Racket 中的延续及其应用

毛奕夫

目录

1 简介

延续 (continuation) 作为对 “剩下的计算” 的表示，在主流的计算机教材中鲜有提及。主流的编程语言大多缺乏对延续的显式支持。由于 Lisp 家族的语言拥有独特的语法结构及求值方式，在这类语言中理解和应用延续可以变得直观。Scheme 是 Lisp 语言的一个方言，其函数调用为按值调用，默认使用词法作用域，支持尾递归优化，延续在其中是一等公民。Racket 是 Scheme 的一种实现及拓展。本文讲解了延续的概念及分类，并列举了延续几种应用，如非本地的跳转、异常处理、协程、引擎以及不确定性编程。

2 Continuation Demystified

在定义延续之前我们先看一个简单的 Racket 代码：

```
(sqrt (+ (* 3 3) (* 4 4)))
```

想要成功运行 sqrt，首先需要解释其子表达式 (+ (* 3 3) (* 4 4))。而如果这个表达式想要成功运行，(* 3 3) 和 (* 4 4) 也得要首先被运行。

假设现在程序运行到了 (* 3 3)，这个表达式可以一步完成，称作规约式 (redex)，规约 (运行) 后获得 9。此时，(* 3 3) 这个表达式的延续就是：

```
(sqrt (+ [] (* 4 4)))
```

方括号会被替换为 (* 3 3) 执行完成后的结果。

当 (* 3 3) 执行完成之后，后面的 (* 4 4) 会继续执行，当执行 (* 4 4) 的时候，它的延续是：

```
(sqrt (+ 9 []))
```

以此类推，执行 (+ 9 16) 时它的延续是：

```
(sqrt [])
```

而当执行 (sqrt 25) 时它的延续就是顶级的 REPL 了。

所以其实可以看出，延续是相对某一个表达式而言的，代表了这个表达式执行完成之后需要进行的计算。在程序运行的某一时刻，只有一条表达式在运行，它的延续等着它的结果，所以延续可以看作一个接受一个参数的函数。上面的三个延续就可以相应被看作：

```
1 (λ (v) (sqrt (+ v (* 4 4))))
2 (λ (v) (sqrt (+ 9 v)))
3 (λ (v) (sqrt v))
```

延续既然和函数[1]、变量、结构等对象一样，是一等公民，那么延续也可以复制给一个变量、可以当作函数的参数、可以作为函数的返回值。

怎样获取 (截获) 一个延续呢？答案是通过 Racket 提供的 call-with-current-continuation 函数。call-with-current-continuation 太长，我们也可以使用简化版的 call/cc。

定义

```
(call-with-current-continuation proc
                                [prompt-tag]) → any

(call/cc proc [prompt-tag]) → any

proc : (continuation? . -> . any)
prompt-tag : continuation-prompt-tag?
           = (default-continuation-prompt-tag)
```

[1]在 Lisp 或者函数式编程语言中，函数习惯性被称为过程 (procedure)。本文中我们还是习惯性将这类代码的封装称作函数。

第二个参数 *prompt-tag* 这里先忽略。*proc* 是一个一元函数，执行 `call/cc` 时 `call/cc` 会调用 *proc* 并将捕获的延续作为参数传给 *proc*。*proc* 执行完毕后的返回值即作为 `call/cc` 的返回值。

在上面的例子中，如果我们想截获 `(* 3 3)` 的延续，可以用如下的代码：

```
(sqrt (+ (call/cc (λ (c) (* 3 3)))
         (* 4 4)))
```

这里，`call/cc` 代替了原先的 `(* 3 3)`。*proc* 在这里是 `(λ (c) (* 3 3))`。`call/cc` 会把当前的延续 (即程序运行到 `(* 3 3)` 时的延续) 传递给这个函数。*proc* 的返回值 `(* 3 3) → 9` 也就是 `call/cc` 的返回值。

在上面的代码中我们并没有用到 `c` 这个参数，在下面的代码中，我们将延续 `c` 保存到一个全局变量并调用它：

```
1 (define k #f)
2
3 (sqrt (+ (call/cc (λ (c) (set! k c) (* 3 3)))
4          (* 4 4)))
5
6 (k 5)
```

```
5
4.58257569495584
```

第一行的 `define` 创建了一个变量，它的初始值是什么无所谓，因为我们在 `call/cc` 中通过 `set!` 将其重新赋值成截获的延续。后面的 `(* 3 3)` 执行完后，生成的 `9` 就成为了 `call/cc` 执行完后的值，所以第一个输出是 `5`。

之后我们运行了 `(k 5)`。前面说过，一个延续可以看成一个单个参数的函数。这里，我们截获的延续就可以看成是 `(λ (v) (sqrt (+ v (* 4 4))))`，所以当 `5` 传递给了这个延续，最终我们获得的值是 `4.58257569495584`。

延续的调用会将当前延续直接替换成正在调用的延续，并将调用的参数传递给捕获的延续 [2]。所以如果我们将上面的 `(k 5)` 改为 `(+ 1 (k 5))`，得到的结果还是 `4.58257569495584`，而不是 `5.58257569495584`，因为当前延续 `(+ 1 [])` 被正在应用的延续 `k` 完全替换掉了。

因为调用通过 `call/cc` 截获的延续会终止当前延续，所以这类延续也被称为终止式延续 (abortive continuation)。

现在有一个问题，为什么在上面的例子中 `call/cc` 没有截获整个剩余的程序作为延续呢？毕竟，整个 Racket 的程序实际上是一个大的列表，第一个元素是 `(define k #f)`，第二个是 `(sqrt (+ (call/cc (λ (c) (set! k c) (* 3 3))) (* 4 4)))`，以此类推。这样一来，截获的延续应该是：

```
(define k #f)

(sqrt (+ []
    (* 4 4)))

(k 5)
```

可现实并不是这样，我们截获的延续仅限于 `(sqrt (+ [] (* 4 4)))`，其原因在于 Racket 解释器内部会在每一个顶级语句外部包上一个延续提示 (prompt)。

[2] 也可以理解为整个 `call/cc` 的值被替换成了传递给延续的参数。

3 延续提示和提示标签

Racket 在解释每一条表达式时都会在外面加一层延续提示 (或直接 “提示”)，以防止捕获的延续拓展到这条表达式外，进而防止形成死循环。如果截获的延续是

```
(define k #f)

(sqrt (+ []
    (* 4 4)))

(k 5)
```

那么截获的延续中就包含了对这个延续的调用 `(k 5)`。当调用这个延续时，程序重新运行到 `(+ 5 (* 4 4))`，当 `sqrt` 执行完后再次执行 `(k 5)`，于是程序又跳到 `(+ 5 (* 4 4))`……

延续提示的作用就是界定截获延续的范围，使截获的延续限定在提示内部。怎样获取特定的，或者说以某个提示界断了的延续呢？不要忘了 `call/cc` 的可选参数 *prompt-tag*，用来指定提示标签，提示标签的作用是定位提示。指定了这个可选参数还不够，`call/cc` 还需要和另一个函数——`call-with-continuation-prompt` 相配合。`call-with-continuation-prompt` 也可缩写为 `call/prompt`。

定义

```
(call-with-continuation-prompt proc
                               [prompt-tag
                               handler]
                               arg ...) → any

(require racket/control)
(call/prompt  proc
     [prompt-tag
     handler]
     arg ...)) → any

proc : procedure?
prompt-tag : continuation-prompt-tag?
           = (default-continuation-prompt-tag)
handler : (or/c procedure? #f) = #f
arg : any/c
```

`call/prompt` 也接受一个提示标签作为可选参数。运行时，`call/prompt` 会在当前位置安装一个提示，之后在这个提示下执行 *proc*。这样一来，只要给定的提示标签是同一个对象，在 *proc* 内部调用的 `call/cc` 截取的延续就只会延伸到这个 `call/prompt` 下。如果指定了 *arg …*，则 `call/prompt` 会把这些参数传给 *proc*。`call/prompt` 的返回值是 *proc* 的返回值。

如果在上面的例子中，当运行到 `(* 3 3)` 时我们不想获取 `(sqrt (+ [] (* (4 4))))`，而是 `(+ [] (* (4 4)))` 作为延续，那就可以用 `call/prompt` 配合 `call/cc` 实现。

```
;; 使用 call/prompt 需要导入 racket/control
(require racket/control)

;; 提示标签通过 make-continuation-prompt-tag 创建
```

```
(define P (make-continuation-prompt-tag))
(define k #f)

(sqrt (call/prompt
       (λ () (+
              ;; 如果没有指定 P，则依然获取整个当前延续
              (call/cc (λ (c) (set! k c) (* 3 3)) P)
              (* 4 4)))
       P #f)) ; 这里 call/prompt 的第二个可选参数 handler 暂时不用，设为#f

;; 注意：调用带有标签的延续同样需要在有和其标签相同的提示下完成
(call/prompt (λ () (k 1)) P #f)
(+ 1 (call/prompt (λ () (k 1)) P #f))
```

```
5
17
18
```

call/cc 的返回值是 (* 3 3) 的值，而 call/prompt 的返回值又是其参数 *proc* 的返回值，所以第一个输出是 5。第二次，我们在用同样的提示标签标记的延续提示下调用截获的延续，输出的值是 17，因为我们截获的延续是 (+ [] (* 4 4))，不包括 (sqrt ...)，所以 (+ 1 (* 4 4)) → 17。另外，我们看到最后一行输出的结果是 18，而不是 17，说明最后一行语句的当前延续 (+ 1 []) 没有被替换，所以这里也可以看出，提示不仅可以界断截获延续的范围，也可以限制应用延续时当前延续被替换的范围。

call/prompt 可以嵌套使用，每个提示也可以用相同或不同的标签标记。如此一来，call/cc 截取的延续会延伸到离它最近的有相同的标签的提示。

在以往支持延续的编程语言实现中，延续并不能被分界。这种被分界的延续也被称为界断延续 (delimited continuation)[5][11]。

如果只是简单地要界断截取的延续的范围，我们可以用 prompt 搭配 control 或者 reset 搭配 shift。prompt 和 reset 作用相同，control 和 reset 同样。在早期的关于程序控制结构的文献中 [2][4][11]，prompt 和 control 或，reset 和 shift 是独立的控制操作符，但 Racket 里它们可以交叉使用，比如 prompt 搭配着 shift，因为它们都有着相同的宏展开。另外，prompt 还可以直接简化为% [4]。

更方便的是通过以上两对操作符获取的界断延续可以直接使用，而不是放在某个用标签标记了的提示里。

```
(require racket/control)
(define c #f)

(* 10 (prompt
       (+ 2 (control k (set! c k) 2))))

(* 10 (prompt
       (+ 2 (shift k 2))))

(* 10 (reset
       (+ 2 (shift k 2))))

(+ 100 (c 5))
```

```
20
20
20
107
```

前三个表达都都输出 20 而不是 40，因为在 prompt/reset 中使用 control/shift 时，整个 prompt/reset 的值会被替换为 control/shift 的尾部表达式的值，这样的操作符有着和后面讲到的 abort/cc 类似的行为。但通过 control/shift 获取的延续却包含了其外部和后面的直到 prompt/reset 的表达式，如上面第一个表达式中 k 对应的就是 (+ 2 [])。

最后一行输出 107 而不是 7，表明当前延续 (+ 100 []) 并没有被截获的延续替换，意味着通过这类操作符截获的延续是可组合的。

4 组合延续

前面说过，当调用 call/cc 捕获的延续时，整个当前延续会被直接替换成正在调用的延续。而如果想要保留当前延续并调用另一个延续，我们可以使用组合式延续 (composable continuation)。组合式延续使用 call-with-comsable-continuation 或 call/comp 截获，截获的延续也是作为参数传递给 *proc*。

定义

```
(call-with-composable-continuation proc
                                   [prompt-tag]) → any

(require racket/control)
(call/comp proc
           [prompt-tag]) → any

proc : (continuation? . -> . any)
prompt-tag : continuation-prompt-tag?
           = (default-continuation-prompt-tag)
```

```
1 (require racket/control)
2 (define k #f)
3
4 (sqrt (+ (call/comp (λ (c) (set! k c) (* 3 3)))
5          (* 4 4)))
6
7 (+ 1 (k 5))
```

```
5
4.58257569495584
5.58257569495584
```

上面的代码中只有第 4 行和第 7 行的两个表达式执行后可以输出值，但实际得到的结果有 3 行。这是因为在截获组合式延续的时候，如果不用提示作界断，call/comp 截获的延续会包含顶层的 REPL，这意味着在调用这样一个组合式延续的时候，这个延续执行完后的值会首先被打印出来，然后这个值再被返回到该延续的调用点，接着当前延续继续执行。

所以上面在输出 5 后会有两个输出，其中 4.58257569495584 是运行 (k 5) 后的输出。之后这个值又返回给 (k 5) 的延续 (+ 1 [])，于是输出了 5.58257569495584。

为了不在应用这种组合式延续的时候打印其结果，我们可以将其界断：

```
(require racket/control)
(define k #f)

(prompt (sqrt (+ (call/comp (λ (c) (set! k c) (* 3 3)))
        (* 4 4))))

(+ 1 (k 5))
```

```
5
5.58257569495584
```

这样就消去了应用组合式延续的“副作用”。

5 终止延续

这里要介绍的不是终止式延续，而是终止当前延续的函数 abort-current-continuation。

定义

```
(abort-current-continuation prompt-tag
                            v ...) → any

(require racket/control)
(abort-cc prompt-tag
          v ...) → any

prompt-tag: any/c
v: any/c
```

可以看到，这个函数的第一个参数 *prompt-tag* 指定一个提示标签，当执行这个函数时，程序会跳转到最近的由这个标签标记的提示。但这还不是全部。后面还有一些可选参数 v ...，它们有何用呢？

前面讲到 call-with-continuation-prompt 时我们暂时忽略了 *handler* 这个参数，这里我们就要用到了。*handler* 是一个自定义的函数，正常执行 call/prompt 时不会被执行，但当我们使用 abort/cc 跳转到一个有 *handler* 的提示时，*handler* 就会被执行，其接受到的参数就是 v ...。因为我们终止了当前延续，又调转到了先前的提示，所以这个提示的返回值，即 call/prompt 的值，就是 *handler* 执行后返回的值。

依旧沿用上面的例子看看 abort-cc 的行为：

```
(require racket/control)
(define P (make-continuation-prompt-tag))

(sqrt (call/prompt
       (λ () (+
              (abort/cc P 25)
              (* 4 4)))
       P
       (λ (x) (printf "Got argument ~a~n" x) x)))
```

```
Got argument 25
5
```

这里，call/prompt 安装一个用 P 标记的提示，之后运行其参数 *proc*，即 5 到 7 行的 lambda。运行到 abort/cc 时，程序跳转到最近的由 P 标记的提示，并将参数 25 传递给该提示的 *handler*，即第 9 行的 lambda。这个 lambda 的返回值作为提示的返回值交给 sqrt，最终输出 5。

6 延续屏障

如果在延续屏障 (continuation barrier) 里截获了一个延续，那么这个屏障也会包含在延续内，这样一来该延续的应用就仅仅局限在这个屏障之内。如果在这个屏障外部调用这个延续则会触发异常。

```
(define k #f)
(+ 1 (call-with-continuation-barrier
       (λ () (+ 1
                (call/cc
                 (λ (c) (set! k c) (k 3)))))))
```

```
5
```

而如果

```
1 (define k #f)
2 (+ 1 (call-with-continuation-barrier
3       (λ () (+ 1
4                (call/cc
5                 (λ (c) (set! k c) (k 3)))))))
6 (k 5)
```

则第 6 行会触发异常，因为正在应用的延续包含一个延续屏障。

延续屏障可以防止在某些上下文内获取的延续在其它地方错误使用。

7 动态缠绕

使用函数式编程语言并不是完全不需要副作用，有时我们还是需要读写文件，或在网络上收发数据。延续意味着程序可以随时跳转。假如应用某个延续时程序跳转到某一位置，而这里的代码执行文件操作，如果此时文件都还没有打开，那势必会报错。所以，我们需要一种机制来确保在使用延续进行程序跳转时执行恰当的代码，为延续中的代码执行作准备。同理，如果打开文件后还没来得及关闭就跳转到了另外的代码，那也可能造成系统资源的浪费以及文件的损坏等问题，所以我们也需要一种机制来确保在跳出某些代码时执行一些代码。

dynamic-wind 接收三个参数，均为无参函数 (thunk)。运行时，dynamic-wind 会依次执行 *pre-thunk*，*value-thunk* 和 *post-thunk*。dynamic-wind 的值就是 *value-thunk* 返回的值。运行 *pre-thunk* 和 *post-thunk* 通常只为它们的副作用，比如在 *pre-thunk* 打开文件或者套接字，然后在 *post-thunk* 关闭它们。

定义

```
(dynamic-wind pre-thunk
              value-thunk
              post-thunk) → any

pre-thunk : (-> any)
```

```
value-thunk : (-> any)
post-thunk : (-> any)
```

如果不使用延续，dynamic-wind 就仅仅依次执行三个参数。而如果使用了延续，事情就变得有趣了：只要在 dynamic-wind 里调用延续，那么跳转之前 *post-thunk* 会先被执行；而只要在 *value-thunk* 里截获的延续在其它地方调用，那么 *pre-thunk* 会先被执行，然后是延续体，再然后就是 *post-thunk*。

```
(require racket/control)
(define c #f)

(dynamic-wind (λ () (displayln "In"))
              (λ () (displayln "Body"))
              (λ () (displayln "Out")))
(displayln "------------------------")

(dynamic-wind (λ () (displayln "In"))
              (λ ()
                (displayln "Body")
                (let/cc k (set! c k))
                (displayln "Tail"))
              (λ () (displayln "Out")))
(displayln "------------------------")

(c)
(displayln "------------------------")

(let/cc k
  (dynamic-wind (λ () (displayln "In"))
                (λ ()
                  (displayln "Body")
                  (k)
                  (displayln "Not executed") )
                (λ () (displayln "Out"))))
```

```
In
Body
Out
------------------------
In
Body
Tail
Out
------------------------
In
Tail
Out
------------------------
In
Body
Out
```

上面的第一个 dynamic-wind(dw) 展示了正常的执行顺序。第二个 dw 依然如此，只不过将其内部的一个延续保存在了全局变量 c 中。接下来 17 行的 (c) 调用保存的延续，可以在输出中看出，首先是 dw 的 *pre-thunk* 被执行，其次是延续部分 (13 行)，之后是 *post-thunk* 部分。最后一个 dw 中间使用了 let/cc 截获的延续跳出，25 行未被执行，*post-thunk*(26 行) 被执行。

8 延续的图形表示

可以借用 [5] 中的示意图来分析上面的延续截获原理。从左向右为程序的运行顺序，方框代表规约式。

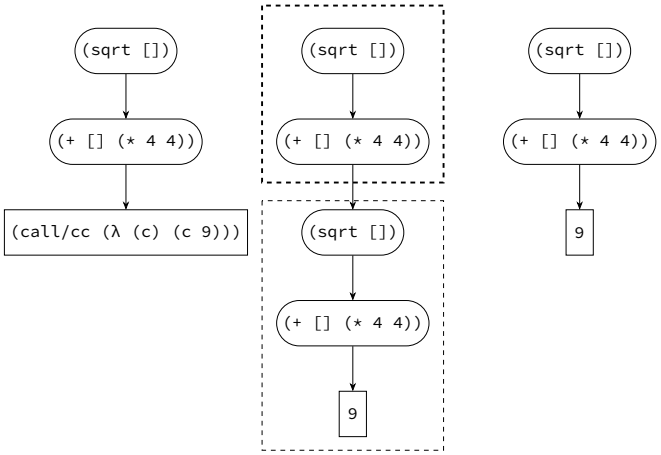

图 1: 默认 (终止式) 延续的捕获和应用，应用延续时当前延续 (粗虚线框) 被正在应用的延续 (细虚线框) 替代。

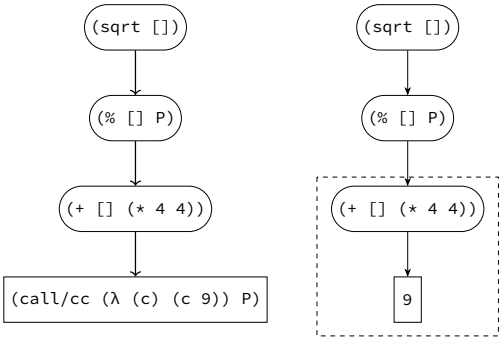

图 2: 提示和标签在延续捕获时的作用。%表示提示，P 为标签，下同。

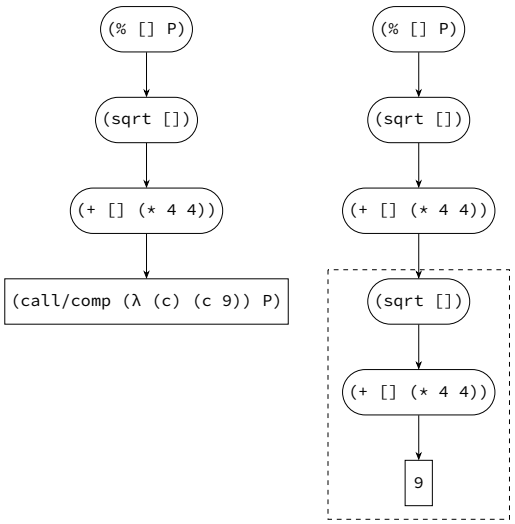

图 3: 组合式延续的捕获和应用。捕获的组合式延续向上延伸到以 P 标记的提示下，应用此延续时，当前延续被正在应用的延续延展，而不是被替换。

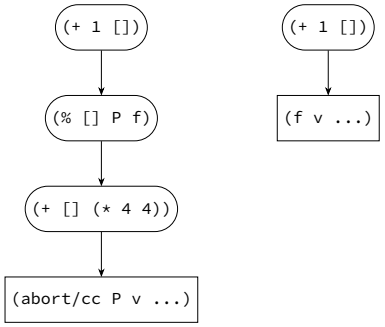

图 4: 延续的终止。f 为提示的处理函数。运行 abort/cc 之后，程序跳转到标签处，参数 v ... 被传递给 f。

9 延续的应用

9.1 非本地跳转

在某些计算情境下，一旦我们读取到某些值，则当前的计算可以直接停止。一个很简单的例子就是将一个数列相乘，一旦碰到 0，则直接返回 0，不需要后续的计算。这样一个乘法函数类似如下：

```
(define (mul args)
  (let/cc k
    (define (loop res l)
      (cond
        [(null? l) res]
        [(= 0 (car l)) (k 0)]
        [else (loop (* res (car l)) (cdr l))]))
  (loop 1 args)))

(mul '(1 2 3 4))
(mul '(1 2 3 0 4))
```

```
24
0
```

这里 let/cc 是 call-with-current-continuation 的又一个简化版，省去了每次编写 lambda 表达式的麻烦，截获的延续直接存放在变量 k 中。

9.2 异常处理

Racket 提供的 call-with-exception-handler 实现了基本的异常处理机制。

定义

```
(call-with-exception-handler f thunk) → any

f : (any/c . -> . any)
thunk : (-> any)
```

当 *thunk* 内部调用 raise 触发一个异常时，异常处理程序 f 被调用，它的参数就是传递给 raise 的 v。

利用延续，我们可以写一个 call-with-handler 来模拟 call-with-exception-handler[3]。当 call-with-handler 内部通过 raise 触发异常时，和其绑定的异常处理函数 *handler* 被执行，之后 call-with-handler 返回。当 call-with-handler 嵌套时，只有距离 raise 最近的 call-with-handler 生效。

```
;; 截获并直接返回当前延续
(define (cc) (call/cc (λ (c) c)))

;; 用于临时存放 call-with-handler 内获取的延续的栈
(define exn-stack '())

(define (call-with-handler handler proc)
  (let ([c (cc)])
    (cond
      [(continuation? c)
         (dynamic-wind
           (λ () (set! exn-stack (cons c exn-stack)))
           proc
           (λ () (set! exn-stack (cdr exn-stack))))]
        [(pair? c) (handler (cdr c))])))

(define (raise e)
  (let ([cont (car exn-stack)])
    (cont (cons 'exn e))))

(define handle (λ (e) (printf "Got exception: ~a~n" e)))

(call-with-handler
 handle
 (λ ()
   (call-with-handler handle
                      (λ ()
                        (displayln "Entering..."))
```

[3] http://matt.might.net/articles/programming-with-continuations–exceptions-backtracking-search-threads-generators-coroutines/

```
                    (raise "Boom!")
                    (displayln "Not displayed")))
  (displayln "Continue..."))
```

```
Entering...
Got exception: Boom!
Continue...
```

call-with-handler 利用了之前讲到的 dynamic-wind。执行 *proc* 之前，call-with-handler 的延续被压进栈中，接着执行 *proc*。无论 *proc* 是正常执行完毕还是中途通过 raise 触发异常，dynamic-wind 的 *post-thunk* 部分都会被执行，将栈顶元素弹出。如果通过 raise 触发异常，那么栈顶的延续被取出并调用，接着程序回到最近的 call-with-handler，此时 c 不再是延续，而是在 raise 里构建的一个对子 (pair)，之后异常处理函数被执行，最近的 call-with-handler 在异常处理函数执行完后退出，所以上面的代码输出了 Continue...。异常处理函数的返回值也是这个 call-with-handler 的返回值。类似的异常处理机制也可以用延续提示配合终止延续实现。

9.3 协程

协程是 (coroutine) 一种在用户空间实现并发的机制 [7] [8]。一个的协程可以在一定情况下 (如遇到阻塞操作时) 将自身挂起，同时启动另外一个协程。挂起的协程状态被保存，在下次启动时从挂起的地方继续执行。和进程、线程不同，协程的调度一般是非抢占式的，因为它们自己决定何时交出控制权并挂起自身。相比之下，操作系统调度线程对线程本身来说是透明的，只要分配给这个线程的时间片耗尽，操作系统就暂停这个线程的执行，转而切换到下一个线程。

[9] 中区分了对称和非对称协程。对称协程往往只用一个函数来实现不同协程间的控制传递，这种传递往往需要一个中心调度器。非对称协程使用两个函数：一个用来挂起自身，一个用来启动其它协程。这里实现的是文中所讲的非对称协程 [4]。

我们需要实现三个函数：create、yield 和 resume。协程通过 create 创建，其接受一个无参函数作为参数并返回一个创建的协程对象。协程主动调用 yield 交出控制权，也可以通过 resume 调用其它协程。yield 有一个可选参数，调用后这个参数会传给这个协程的调用方作为 resume 的返回值。resume 有两个参数，分别是待启动的协程以及要传递的参数。如果启动的协程处于挂起状态，那么被 resume 重新启动后会从上一次调用 yield 的地方继续执行，yield 返回的值即是 resume 的第二个参数。此外，协程正常执行完毕退出后其最后一个表达式的值也将作为其调用方的 resume 的值。

我们用一个结构 coroutine 来表示创建的协程对象，定义如下：

```
(struct coroutine (body status) #:mutable)
```

body 即待执行的函数本身。同时，当协程挂起时，body 会被重新赋值为其延续。status 记录协程的状态，有三种可能性：'suspended，'running，stopped。#:mutable 选项指定 body 和 status 都可通过 set-coroutine-body!和 set-coroutine-status!重新赋值。

create 实际上就是创建了一个 coroutine 结构：

```
(define (create cr)
  (coroutine cr 'suspended))
```

[4]这里参考了 https://gist.github.com/takikawa/4090648 的实现。

协程在调用 yield 后需要获取当前延续，这个延续记录了协程被再次启动后需要执行的操作。前面说过，这个延续会被赋值给这个协程结构的 body 字段，但问题是协程本身没有办法访问自己的这个结构，所以我们需要其它方法保存获取到的这个延续。另外，我们还需要解决调用 yield v 时 v 的传递问题。

假设协程的延续已经存入了 body 字段，那么用 resume 恢复协程就简单了：直接从传入的 coroutine 结构中提取延续调用即可，调用延续时指定的参数也会被直接作为协程调用 yield 的返回值。

事实上，调用 yield 时的延续保存和参数传递问题可以通过先前讨论的 abort-current-continuation 函数解决。我们已经知道，abort-current-continuation 不仅可以终止当前延续、跳转到指定的提示下，还可以传递一些参数到那个提示的处理程序里。

所以 yield 的机制如下：

1. 获取当前延续 k
2. 终止当前延续，将 k 和可选的参数 v 一并传给指定的提示的处理程序
3. 此时程序已经跳转到 resume 函数里，提示处于其中
4. 处理程序接收到 k 和 v 后，将 k 保存，将 v 返回
5. v 作为调用 resume 的值返回

yield 和 resume 的实现如下：

```
(define (yield v)
  (call-with-current-continuation
   ;; 这里 cr-prompt 用于指定提示
   ;; 截取的延续延伸到本协程的调用方 (resume 函数里) 的 call-with-continuation-prompt
   (λ (k) (abort-current-continuation cr-prompt k v)) cr-prompt))

(define (resume cr . args)
  (call-with-continuation-prompt
   (λ (v)
     (when (equal? 'stopped (status cr)) (error "trying to start a stopped coroutine!"))
     (set-coroutine-status! cr 'running)
     (define val ((coroutine-body cr) v))
     (set-coroutine-status! cr 'stopped)
     val)
   cr-prompt
   ;; 该提示的处理程序
   ;; 将接收到的延续 cont 保存后返回 val 作为 resume 函数的值
   (λ (cont val)
     (set-coroutine-body! cr cont)
     (set-coroutine-status! cr 'suspended)
     val)
   args))
```

别忘了定义 cr-prompt：

```
(define cr-prompt
  (make-continuation-prompt-tag 'coroutine))
```

上面的 yield 还有待改进，因为它必须接受一个参数返回给调用方。我们可以重新将其定义成一个宏以支持带参数和不带参数的调用：

```
(define-syntax yield
(syntax-rules ()
  [(_)
   (call-with-current-continuation
    ;; 不带参数时默认返回'()
    (λ (k) (abort-current-continuation cr-prompt k '())) cr-prompt)]
  [(_ v)
   (call-with-current-continuation
    (λ (k) (abort-current-continuation cr-prompt k v)) cr-prompt)]))
```

总的代码如下：

```
(struct coroutine (body status) #:mutable)

(define cr-prompt (make-continuation-prompt-tag 'coroutine))

(define (create cr)
  (coroutine cr 'suspended))

(define (resume cr . args)
  (call-with-continuation-prompt
   (λ (v)
     (when (equal? 'stopped (status cr)) (error "trying to start a stopped coroutine!"))
     (set-coroutine-status! cr 'running)
     (define val ((coroutine-body cr) v))
     (set-coroutine-status! cr 'stopped)
     val)
   cr-prompt
   (λ (cont val)
     (set-coroutine-body! cr cont)
     (set-coroutine-status! cr 'suspended)
     val)
   args))

(define-syntax yield
  (syntax-rules ()
    [(_)
     (call-with-current-continuation
      (λ (k) (abort-current-continuation cr-prompt k #f)) cr-prompt)]
    [(_ v)
     (call-with-current-continuation
      (λ (k) (abort-current-continuation cr-prompt k v)) cr-prompt)]))

;; 获取协程状态
(define (status cr)
  (coroutine-status cr))
```

9.3.1 例子

这里将 Matthew Might 的一个例子[5] 稍加改变来进行协程的轮流调度。

```
;; 用来存放创建的协程
(define Q '())

;; 将协程加进 Q
(define (add-task t)
  (set! Q (cons t Q)))

;; 方便带参数的协程创建
(define (wrap0 f . args)
  (create (λ (x) (apply f args))))

;; 协程调度函数
;; 依次从 Q 中取出协程并执行
(define (start-coroutines)
  (let/cc k
    (define (loop)
      (cond
        ;; 协程全部执行完毕，退出
        [(null? Q) (k "All done")]
        [(equal? 'stopped (status (car Q))) (set! Q (cdr Q)) (loop)]
        [else
         (let* ([c (car Q)]
                [res (resume c)])
           (if (equal? res #f)
               (begin (set! Q (cdr Q)) (loop))
                ;; 开始下一个协程
               (begin (set! Q (append (cdr Q) (list c))) (loop))))]))
    (loop)))

(define counter 10)
;; 递减全局变量 counter
(define (make-thread-thunk name)
  (define loop (lambda ()
                 (when (< counter 0)
                   (yield #f))
                 (display "in thread ")
                 (display name)
                 (display "; counter = ")
                 (display counter)
                 (newline)
                 (set! counter (- counter 1))
                 (yield 1)
                 (loop)))
  (loop))

;; 创建三个协程来递减 counter
```

[5] http://matt.might.net/articles/programming-with-continuations–exceptions-backtracking-search-threads-generators-coroutines/

```
(add-task (wrap0 make-thread-thunk 'a))
(add-task (wrap0 make-thread-thunk 'b))
(add-task (wrap0 make-thread-thunk 'c))

(start-coroutines)
```

```
in thread c; counter = 10
in thread b; counter = 9
in thread a; counter = 8
in thread c; counter = 7
in thread b; counter = 6
in thread a; counter = 5
in thread c; counter = 4
in thread b; counter = 3
in thread a; counter = 2
in thread c; counter = 1
in thread b; counter = 0
"All done"
```

9.4 生成器

基于协程，我们可以编写生成器 (generator) 用于生成特定的值。生成器每调用一次会返回一个值到其调用方，下次调用时同样如此。首先我们需要一个函数来方便生成器的创建：

```
(define (wrap f . args)
  (define cr (create (λ (x) (apply f args))))
  (λ () (resume cr)))
```

首先是一个生成自然数的例子：

```
(define (naturals)
  (define (loop n)
    (yield n)
    (loop (+ 1 n)))
  (loop 0))

(define n (wrap naturals))
```

自然数太单调，不如生成斐波那契数列：

```
(define (fib)
  (define (loop x y)
    (yield x)
    (loop y (+ x y)))
  (loop 1 1))

(define f (wrap fib))
```

这样一来每执行一次 `(n)` 或 `(f)` 都会得到对应数列的下一个值。

9.5 引擎

在协程一节的例子中我们通过 start-coroutines 函数实现了多任务的非抢占式调度，所谓“非抢占”是因为各个协程是主动调用 yield 来交出控制权，回到 start-coroutines，接着 start-coroutines 执行下一个协程。

引擎是一种定时的、抢占式的多任务执行模型，可以用来模拟多任务并发、操作系统内核，以及不确定性编程 [3][6]。主要的函数是 make-engine。这个函数接收一个无参函数作为参数，包含了需要执行的计算。make-engine 返回另一个三元函数，这个函数就是一个引擎。引擎的三个参数分别是：

- *ticks*：引擎的燃料，即该引擎可以运行多久
- *return*：二元函数。当引擎在 *ticks* 之内完成计算，则将计算结果及剩余时间作为参数传给 *return*
- *expire*：一元函数。如果超时，即当 *ticks* 归零时计算未完成，则该函数被调用，其参数是一个包含剩余未完成计算的新引擎。

这里的 *ticks* 可以是物理时间，也可以是其它便于记录的计算单位。这里，我们沿用 [10] 中的方法，将每一个 *tick* 记录为一次用户自定义函数的调用。计时系统的实现如下：

```
(define clock 0)

(define (timer-handler)
  (call/cc (λ (k) (abort/cc P k)) P))

(define (start-timer ticks)
  (set! clock ticks))

(define (stop-timer)
  (let ([time-left clock])
    (set! clock 0)
    time-left))

(define (decrement-timer)
  (if (> clock 0)
      (set! clock (- clock 1))
      (when (= clock 0)
        (timer-handler))))
```

timer-handler 中的 P 是一个提示标签，用来界断引擎运行超时时截获的延续：

```
(define P (make-continuation-prompt-tag 'engine))
```

为了记录函数的调用，我们自定义一个 t-lambda：

```
(define-syntax t-lambda
  (syntax-rules ()
    [(_ param exp ...) (lambda param (decrement-timer) exp ...)]))
```

这样一来，每当调用使用 t-lambda 编写的函数时，全局时钟 clock 都会减少一个刻度。如果减少到 0，则调用 time-handler 终止当前计算，并把获取的延续传递给 P 标记的提示的处理函数。提示处于 make-engine 函数下：

```
(define (make-engine proc)
  (λ (ticks return expire)
    (set! clock ticks)
    (let ([res (call/prompt proc P id)]
          [time-left (stop-timer)])
      ;; 如果 res 是一个延续则表示引擎运行超时，调用 expire
      ;; 否则将剩余的时间和计算结果传递给 return
      (if (continuation? res)
          (expire (make-engine res))
          (return time-left res)))))
```

其中 id 的定义是：

```
(define (id x) x)
```

意味着如果 *proc* 内部通过 decrement-timer 触发了 timer-handler，则其延续会被作为参数传递给 id，id 直接返回这个延续作为 call/prompt 的值，接着赋值给 res。

下面是一个简单的例子。我们用 t-lambda 编写一个阶乘函数，并使用 total-ticks 统计一个阶乘函数执行结束需要的时间 (递归的次数)：

```
(define (total-ticks eng)
  (define (loop ticks resume total)
    (let ([val (resume ticks list id)])
      (cond [(procedure? val) (loop 100 val (+ total ticks))]
            [(list? val) (- ticks (car val))])))
  ;; 为了避免多次超时、保存延续并重新创建引擎，这里可以给定比 100 更大的值
  (loop 100 eng 0))

(define fac
  (t-lambda (n res)
            (if (= n 0) res
                (fac (- n 1) (* n res)))))

(define fac1 (make-engine (λ () (fac 6 1))))
(define fac2 (make-engine (λ () (fac 10 1))))

(total-ticks fac1)
(total-ticks fac2)
```

```
7
11
```

到这里我们会发现，协程和引擎的原理是类似的，只不过一个是主动截获延续并中断执行，另一个是被下层隐藏的定时器处理程序中断。之后两者的延续都被保存以便后续使用。

9.6 不确定性编程/回溯

在《计算机程序的构造和解释》一书 [1] 的第四章第三节，作者实现了一个可以进行不确定计算 (non-deterministic computing) 的 Scheme 元循环解释器，整个解释器基于延续传递风格 (continuation-passing style, CPS)。解释器在解释每一条指令的时候都会传入一个“成功延续”和一个“失败延续”。每当一个表达式被求得一个值，其成功延续就会被调用。失败延续只在解释赋值表达式和和不确定计算相关的 amb 函数时才会被调用，因为在向一个变量赋值失败时需要调用失败延续恢复变量之前的值，而当 (amb) 生成的值无效，或者已经没有值可以生成时，也需要调用失败延续来回退到之前的语句。这种行为也称为回溯 (backtracking)。

[1] 中的例子是基于元循环解释器的，这里我们不另写一个解释器，而是直接利用宏和延续来实现回溯。

```
(require racket/control)

;; 存储需要重新执行的计算
(define Q '())

(define-syntax amb
  (syntax-rules ()
    [(_) #f]
    [(_ exp) exp]
    [(_ exp1 exp2 ...)
     (call/cc (λ (k)
                (set! Q (cons (λ () (k (amb exp2 ...))) Q))
                exp1))]))

;; 清除之前 amb 表达式的痕迹
(define (clear) (set! Q '()))

(define (try-next)
  (if (null? Q)
      #f
      (let ([c (car Q)])
        (set! Q (cdr Q))
        (c))))

(* 5 (amb 1 2) (amb 3 4))
(try-next)
(try-next)
(try-next)
(clear)
(displayln "-------------------------")

(define (? pred expr)
  (if (pred expr)
      expr
      (try-next)))

(? odd? (+ (amb 4 5 6 7 8) (amb 1 2 3 4 5)))
(try-next)
(try-next)
```

```
(try-next)
(clear)
(displayln "-------------------------")

(define (an-element-of items)
  (if (null? items)
      (try-next)
      (amb (car items) (an-element-of (cdr items)))))

;; 生成水仙花数
(define nums '(0 1 2 3 4 5 6 7 8 9))
(define nums1 '(1 2 3 4 5 6 7 8 9))

(let ([a (an-element-of nums1)]
      [b (an-element-of nums)]
      [c (an-element-of nums)])
  (if (or (eq? a #f) (eq? b #f) (eq? c #f))
      (abort)
      (if (= (+ (* a a a) (* b b b) (* c c c)) (+ (* 100 a) (* 10 b) c))
          (printf "~a~a~a~n" a b c)
          (try-next))))

(try-next)
(try-next)
(try-next)
```

```
15
20
30
40
-------------------------
5
7
9
7
-------------------------
153
370
371
407
```

这里的核心代码是自定义的宏 amb，它可以接受任意多个参数。如果参数只有一个则直接将其返回，否则返回第一个参数并将剩余的参数包装成另一个 amb 表达式放进 Q 中。

因为我们在截获延续时使用的是 call/cc，所以整个当前延续都会被截取，包括 amb 之外的表达式。这样一来，当调用 try-next 时，我们得到的结果是先前整个包含 amb 的表达式的结果。

10 总结

本文主要通过 Racket 编程语言介绍了延续的概念，延续的不同种类及截获方式。Racket 利用延续提示实现了延续的界断。延续作为一种程序状态的保存，有很多应用。文中利用延续实现了非本地跳转、异

常处理、协程、生成器、引擎以及回溯。事实上，Racket 的异常处理机制和参数 (动态绑定) 都是基于界断延续的。本文没有涉及更一般的延续帧和延续标记，其中延续提示和延续屏障为延续帧的特例，提示标签是延续标记的特例。在其它语言中也不乏延续的应用，如 C#中对异步编程的支持，只不过在大部分主流编程语言中延续没有作为一等公民存在。利用延续进行跳转和 `goto` 类似，因此，不恰当地使用延续可能会导致程序的逻辑混乱，应该避免。

参考文献

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 151 – 160, New York, NY, USA, 1990. Association for Computing Machinery.

[3] R.Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[4] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 180 – 190, New York, NY, USA, 1988. Association for Computing Machinery.

[5] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. *SIGPLAN Not.*, 42(9):165 – 176, October 2007.

[6] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[7] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, page 293 – 298, New York, NY, USA, 1984. Association for Computing Machinery.

[8] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3):143–153, 1986.

[9] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2), February 2009.

[10] Dorai Sitaram. Handling control. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 147 – 155, New York, NY, USA, 1993. Association for Computing Machinery.

[11] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation*, 3(1):67–99, Jan 1990.