Table of Contents

Chez Scheme中文手册

Chez Scheme Vesion 9 User's Guide

本手册是本人在实践中根据自己需要而翻译的。

所以优先翻译C语言库引入，以及高级控制结构的部分。

其他章节暂时没有翻译的计划。

更多Scheme相关，请访问 theschemer.org

术语

过程：根据Lisp传统，函数一般被称为过程。对初学者来说可以简单的将其理解为函数，或方法。

动态链接库：Unix体系内的shared object (共享对象)统一翻译为更好理解的动态链接库，从而不与Windows的dynamic linking library加以区分。

4 外部接口

4.1 子进程通信

4.2 导入Scheme

4.3 从Scheme导出

4.4 延续和外部引用

4.5 外部数据

4.6 访问外部过程

可以通过以下几种方式获得外部过程：

- 可以使用加载动态链接库过程 load-shared-object 加载外部过程。
- 一个新的Chez Scheme映像可以用加载外部代码构建的方式链接。Sforeign_symbol Sregister_symbol 参见4.8
- 其他条目可以动态加载或以其他方式由外部代码获得。通常也使用Sforeign_symbol Sregister_symbol进行注册。
- 入口地址，比如函数指针，可以被传递给Scheme，并用作外部过程表达式中入口语句的值。即使没有按照名称注册，也可以使用外部入口点。

```
过程：(foreign-entry? entry-name)
返回：如果entry-name存在，返回#t，否则#f
库：(chezscheme)
```

entry-name 必须是 string，可用于确定是否存在外部程序的过程。

以下示例假定定义strlen的库已经通过load-shared-object加载，或者strlen已经通过本节中介绍的其他方法注册。

```
foreign-entry? "strlen")#  => t
    ((foreign-procedure "strlen"
    (string) size_t)
 "hey!")  => 4
```

```
过程：(foreign-entry entry-name)
返回：以integer返回entry-name名称的address
库：(chezscheme)
```

入口名称必须是一个命名现有外来入口点的字符串。

以下示例假定定义strlen的库已经通过load-shared-object加载，或者strlen已经通过本节中介绍的其他方法注册。

```
(let ([addr (foreign-entry "strlen")])
    (and (integer? addr) (exact? addr))) => #t

(define-ftype strlen-type (function (string) size_t))
(define strlen
    (ftype-ref strlen-type ()
    (make-ftype-pointer strlen-type "strlen")))
(strlen "hey!") => 4
```

```
过程：(foreign-address-name address)
返回：address对应的entry-name。若不存在，返回#f
库：(chezscheme)
```

以下示例假定定义strlen的库已经通过load-shared-object加载，或者strlen已经通过本节中介绍的其他方法注册。

```
(foreign-address-name (foreign-entry "strlen")) => "strlen"
```

```
过程：(load-shared-object path)
返回：unspecified
库：(chezscheme)
```

path必须是一个字符串。load-shared-object加载由path指定的动态链接库。动态链接库可能是系统库或从普通C程序创建的文件。动态链接库中的所有外部符号以及与shared-object链接的其他动态链接库中可用的外部符号都可用作外部条目。

在Chez Scheme运行的大多数平台上都支持这个过程。

如果path不以"."或者"/"开始，shared-object将在系统默认的环境变量中进行搜索。

在大多数Unix系统上，load-shared-object基于系统例程dlopen。在Windows下，load-shared-object基于LoadLibrary。有关这些例程的文档，请参阅C编译器和加载器，以获取有关查找和构建动态链接库的精确规则。

load-shared-object可以用来访问内置的C库函数，比如getenv。动态链接库的名称因系统而异，在Linux系统上：

```
（load-shared-object "libc.so.6"）
```

在Solaris，OpenSolaris，FreeBSD，NetBSD和OpenBSD系统上：

```
（load-shared-object "libc.so"）
```

在MacOS X系统上：

```
(load-shared-object "libc.dylib")
```

在Windows上：

```
(load-shared-object "crtdll.dll")
```

一旦C库被加载，getenv应该可以作为一个外部入口：

```
(foreign-entry? "getenv") => #t
```

可以这样定义和调用等效的Scheme过程：

```
(define getenv
    (foreign-procedure "getenv"
        (string)
        string))
(getenv "HOME") => "/home/elmer/fudd"
(getenv "home") => #f
```

load-shared-object也可以用来访问用户创建的库，

假设C文件"env.c"包含

```
int even(n) int n; { return n == 0 || odd(n - 1); }
```

C文件"odd.c"包含

```
int odd(n) int n; { return n != 0 && even(n - 1); }
```

这些文件必须被编译并链接到动态链接库才能被加载。其过程取决于系统：

在Linux，FreeBSD，OpenBSD和OpenSolaris上：

```
(system "cc -fPIC -shared -o evenodd.so even.c odd.c")
```

根据主机配置的不同，可能需要 -m32 或 -m64 选项来指定32位或64位编译。

在MacOS X（Intel或PowerPC）系统上：

```
(system "cc -dynamiclib -o evenodd.so even.c odd.c")
```

根据主机配置的不同，可能需要 -m32 或 -m64 选项来指定32位或64位编译。

在32位Sparc Solaris上：

```
(system "cc -KPIC -G -o evenodd.so even.c odd.c")
```

在64位Sparc Solaris上：

```
(system "cc -xarch=v9 -KPIC -G -o evenodd.so even.c odd.c")
```

在Windows上，我们构建一个DLL（动态链接库）文件。为了使编译器生成适当的入口点，我们改变even.c来读取

```
#ifdef WIN32
#define EXPORT extern __declspec (dllexport)
#else
#define EXPORT extern
#endif

EXPORT int even(n) int n; { return n == 0 || odd(n - 1); }
```

和odd.c读取

```
#ifdef WIN32
#define EXPORT extern __declspec (dllexport)
#else
#define EXPORT extern
#endif

EXPORT int odd(n) int n; { return n != 0 && even(n - 1); }
```

然后，我们可以按如下所示构建DLL，并为其提供扩展名“.so”而不是“.dll”，以便与其他系统保持一致。

```
(system "cl -c -DWIN32 even.c")
(system "cl -c -DWIN32 odd.c")
(system "link -dll -out:evenodd.so even.obj odd.obj")
```

生成的“.so”文件可以加载到Scheme中，even和odd可以作为外部过程使用：

```
(load-shared-object "./evenodd.so")
(let ([odd (foreign-procedure "odd"
             (integer-32) boolean)]
      [even (foreign-procedure "even"
              (integer-32) boolean)])
  (list (even 100) (odd 100))) => (#t #f)
```

文件名以“./evenodd.so”定义，而不是简单的“evenodd.so”，因为有些系统在不包含当前目录的标准系统目录集中查找共享库。

```
过程：(remove-foreign-entry entry-name)
返回：unspecified
库： (chezscheme)
```

remove-foreign-entry将访问由entry-name指定的入口。如果目标不存在，就会引发异常。可在外部接口建立后用remove-foreign-entry而不影响之前由foreign-procedure建立的接口访问。

使用remove-foreign-entry可删除使用Sforeign_symbol和Sregister_symbol注册的条目，但不能删除由调用load-shared-object而创建的条目。

4.7 使用其他的编程语言

尽管Chez Scheme外部过程接口主要面向C中定义的过程或C库中可用的过程，但也可以调用其他语言中遵循C调用约定的过程。一个难点的来源可能是名字的解释。基于Unix的C编译器通常会在外部名称前加一个下划线，外部接口将尝试以与主机C编译器一致的方式解释条目名称。

偶尔，如汇编代码文件，这个条目的名称可能不以被期望的方式解释。通过在条目名称前添加一个“=”字符来防止。

例如，加载包含过程“foo”的程序集文件后，可能会有

```
(foreign-entry? “foo”) #  => f
(foreign-entry? “= foo”) #  => t
```

4.8 C库过程

4.9 示例：套接字操作

6 控制结构

6.3 延续

6.4 引擎

引擎是一个支持定时抢占[15，24]的高级过程抽象。引擎可能被用来模拟多任务处理，实现操作系统内核，并执行非确定性的计算。

过程：(make-engine thunk)

返回：一个引擎

库：(chezscheme)

引擎是通过传递一个thunk（无参数过程）来制作引擎而创建的。thunk的主体是由引擎执行的计算。引擎本身是一个三参数的过程：

ticks：一个正整数，指定引擎的运行时间，一个引擎会执行直到ticks用完，或计算完成。

complete：一个或多个参数的过程，指定在计算完成时要执行的操作。它的参数是剩余ticks和计算产生的数值。

expire：一个参数的过程，指定在计算完成前如果ticks耗尽该怎么做。它的参数是能够从中断处重新开始计算的新引擎。

当一个引擎应用于它的参数时，它被设定了一个以ticks的计时器。（参见320页 set-timer）如果在ticks耗尽之前引擎计算完成，系统将调用complete，并将剩余的ticks和计算结果生成的值传递给它。如果在计算完成之前ticks耗尽，则系统将从中断计算的继续中创建新的引擎，并使当前的引擎expire。complete和expire将在引擎调用的延续中调用。

Scheme编程语言第12.11节给出了引擎的实现

请勿使用定时器（见 set-timer）中断引擎，因为引擎是定时器实现的。

下面这个例子是一个10ticks的引擎。

```
(define
  (make-engine
    (lambda () 3)))

(eng 10
     (lambda (ticks value) value)
     (lambda (x) x))   => 3
```

```
(define eng
  (make-engine
    (lambda () 3)))

(eng 10
     list
     (lambda (x) x)) => (9 3)
```

```
(define fibonacci
  (lambda (n)
    (let fib ([i n])
      (cond
        [(= i 0) 0]
        [(= i 1) 1]
        [else (+ (fib (- i 1))
                 (fib (- i 2)))]))))

(define eng
  (make-engine
    (lambda ()
      (fibonacci 10))))

(eng 50
     list
     (lambda (new-eng)
       (set! eng new-eng)
       "expired")) => "expired"

(eng 50
     list
     (lambda (new-eng)
       (set! eng new-eng)
       "expired")) => "expired"

(eng 50
     list
     (lambda (new-eng)
       (set! eng new-eng)
       "expired")) => "expired"

(eng 50
     list
     (lambda (new-eng)
       (set! eng new-eng)
       "expired")) => (21 55)
```