Tenon 语言教程

Release 0.1.0

ZDZN

Apr 12, 2019

Contents

Chapter 1

简介

Tenon 是一门用于机械臂程序开发的编程语言，具有可扩展、支持错误处理、中断等特性。它包括独立的语言核心部分，以及机械臂相关的扩展部分。本文档主要介绍其语言核心部分的内容。

1.1 语言特点

Tenon 以传统的机器人编程语言为范本，参考诸多现代语言，形成了自己的语言特点。

与该领域传统语言相比：

- Tenon 大小写敏感，更符合现代编程习惯。
- 语言更为简洁。
- 语言更为自由。Tenon 的记录类型支持函数成员, Tenon 支持自增、自减表达式等。
- 语言具有扩展性。Tenon 支持通过 c 语言等主流语言进行功能扩展。

1.2 语言展示

首先，我们来编写一个简单的例子程序 hello.t：

```
! This is a simple Tenon program 'hello.t'.

module hello

  function void main()
    twrite("Hello World!");
  end
```

(continues on next page)

(continued from previous page)

```
end
```

在命令行中执行该程序，可以看到输出信息如下:

```
$ tenon -m base hello.t
Hello World!
```

这个例子展示了 Tenon 程序的大概模样:

- 程序由模块组成，每个模块作为一个文件单独保存
- 程序的执行入口函数为 `main` 函数
- 注释以 `!` 开头，直到一行结束
- 语句以 `;` 结束
- 代码块，例如模块，函数等，以 `end` 结束

Tenon 语言大小写敏感，标识符由字母、数字和下划线组成，其中起始字符必须是字母或者下划线。

标识符不能与关键字重名。关键字包括:

```
and alias
backward bool
case connect const
default div do
else elseif end error exit
false for from function
goto
if
local
module mod
not num
or
pers private public
raise record retry return
step string switch
task then trap true trynext
undo
var void
while with
xor
```

Chapter 2

数据

2.1 数据种类

数据种类包括：常量、变量、持久量。

- 常量为只读数据，不能被改写
- 变量即可以被读取，也可以被改写
- 持久量也是可以读写的，与变量不同的是，它在程序运行结束后，其值会被持久保存

2.2 数据类型

数据类型包括：基本类型、记录类型和别名类型。

2.2.1 基本类型

有三种基本数据类型：`num`、`bool`、`string` 。分别对应数字类型、布尔类型和字符串类型。

2.2.2 记录类型

记录类型用来表示一个组合类型，具有一个或者多个域。Tenon 语言的记录类型支持函数成员。下面的例子，展示了如何定义一个记录类型：

```
record info
  string name;
  num age;
end
```

这段代码定义了一个记录类型 `info` 。

有 3 个内建记录类型：`pos`，`orient`，`pose` 。它们分别定义如下：

```
record pos
  num x;
  num y;
  num z;
end

record orient
  num q1;
  num q2;
  num q3;
  num q4;
end

record pose
  pos trans;
  orient rot;
end
```

2.2.3 别名类型

别名类型用来表示一个类型的别名，它与所表示的实际类型是等价的。下面的例子，展示了如何定义一个别名类型：

```
alias num score;
```

这段代码定义了一个别名类型 `score` 。

有 2 个内建别名类型：`errnum`，`intnum` 。它们分别定义如下：

```
alias num errnum;
alias num intnum;
```

2.3 数据声明

Tenon 程序中，声明一个数据，需要给出该数据的种类和类型。数据可以是一个 1 维或多维数组，最多是 3 维。

2.3.1 常量声明

常量声明必须要给出初始值，初始值为一个常量表达式。下面的例子，展示了如何声明一个常量：

```
const num pi = 3.14;
```

这段代码声明了一个 num 类型的常量 pi ，其初始值为 3.14 。

2.3.2 变量声明

变量声明的初始值是可选的，如果有则为一个常量表达式。下面的例子，展示了如何声明一个变量：

```
var bool status;
```

这段代码声明了一个 bool 类型的变量 status 。

2.3.3 持久量声明

持久量声明的初始值是可选的，如果有则为一个文字表达式。下面的例子，展示了如何声明一个持久量：

```
pers pos move_point = [0, 0, 0];
```

这段代码声明了一个 pos 类型的持久量 move_point ，其初始值为 [0, 0, 0] 。

Chapter 3

表达式

表达式用来表示一个值，以及值的运算。

3.1 运算符

Tenon 支持如下运算符：

```
+ - * / div mod ++
-- < <= > >= == <>
and xor or not
```

值得注意的是，Tenon 支持自增运算与自减运算。

3.2 数据引用

表达式中，可以通过标识符来引用一个数据。下面的例子展示了如何引用数据：

```
a
b.x
c{1,2}
```

3.3 常量表达式

常量表达式中不能包含变量，持久量，以及函数调用。常量表达式主要用于数据的初始化。

3.4 文字表达式

文字表达式是一种特殊的常量表达式，其只能是一个单个的文字值，或者一个聚合表达式，且表达式的成员都是单个的文字值。文字表达式主要用于持久量的初始化。

3.5 聚合表达式

聚合表达式表示一个组合值，其可以是一个数组，或者一个记录。例如：

```
[1, 2, 3]
```

Chapter 4

语句

Tenon 支持各种常见的语句，包括声明语句，赋值语句，循环，条件分支，函数调用等；同时也支持异常处理和中断处理相关的语句。

4.1 赋值语句

赋值语句是将一个表达式的值赋予一个数据，数据类型和表达式类型必须匹配。

例如：

```
a = 1;
```

4.2 if 语句

if 语句是常用的选择语句。语句以 `if` 开头，以 `end` 结束。由 if、elseif、else 三部分组成。其中 elseif 部分可以有多个或没有, else 部分可省略。

当满足条件时执行 then 语句列表。语法为：

```
if 条件表达式 then 语句列表
{elseif 条件表达式 then 语句列表}
[else 语句列表]
end
```

下面是一个简单示例：

```
if option == "-h" then
  show_help();
```

(continues on next page)

(continued from previous page)

```
elseif option == "-v" then
  show_version();
else
  return;
end
```

此例中的 if 部分由条件表达式 `option == "-h"` 以及 then 语句列表 `show_help();` 组成。当 `option` 的值为 `"-h"` 时，调用函数 `show_help()` 。

4.3 for 语句

for 语句为常用循环语句，以 `for` 开头，以 `end` 结束。由循环变量，起点表达式，终点表达式以及语句列表组成。其中步长表达式可省略，默认值为 `1` 。

语句会重复执行语句列表，每执行一次循环变量的值会增加步长表达式的值，当循环变量的值超过超过终点表达式的值时，循环终止。语法为：

```
for 循环变量 from 起点表达式 to 终点表达式 [步长表达式]    do
  语句列表
end
```

下面是一个简单示例：

```
var num sum = 0;
for i from 1 to 10 do
  sum = sum + i;
end
```

此例中，我们通过 for 循环求得从 1 到 10 的整数之和。

4.4 while 语句

while 语句为常用循环语句，以 `while` 开头，以 `end` 结束。由条件表达式以及语句列表组成。

语句会重复执行语句列表直至不再满足条件。语法为：

```
while 条件表达式   do
  语句列表
end
```

下面是一个简单示例：

```
var num sum = 0;
var num i = 1;
while i <= 10 do
  sum = sum + i;
  i = i + 1;
end
```

此例中，我们通过 while 循环求得从 1 到 10 的整数之和。

4.5 switch 语句

switch 语句是常用的选择语句。语句以 `switch` 开头，以 `end` 结束。由 switch、case、default 三部分组成。其中 case 部分可以有多个或没有, default 部分可省略。

语句分析表达式的值，如果有与之相等的测试值，则执行对应 case 的语句列表，否则执行 default 部分的语句列表。语法为：

```
switch 表达式
{case 测试值: 语句列表}
[default: 语句列表]
end
```

下面是一个简单示例：

```
switch a
case 1:
  return true;
default:
  return false;
end
```

此例中，我们通过 switch 语句判断 a 的值是否为 1。

4.6 return 语句

return 语句用于函数返回一个表达式的值或空值。

下面是一些简单示例：

```
return;
return 1;
return a;
```

4.7 exit 语句

exit 语句用于退出程序。

下面是一个简单示例：

```
var num i = 0;

while true do
  i = i + 1;
  tp_write(i);
  if i >= 10 then
    exit;
  end
end
```

此例用于打印从 1 到 10 的所有整数。

4.8 raise 语句

错误处理的常用语句，用于设置错误或将错误报告到上一层。raise 语句有两种形式，一种为 raise 后带上显式的错误编号，一种为 raise 单独使用。前者用来设置一个显式错误，该显示错误的编号只能是从 1 到 90 这个范围之内的；并且这种形式不能用在错误处理之中。后者用来将错误报告到上一层，只能用在错误处理之中。

下面分别就 raise 的两种形式做具体的演示。raise XXX (XXX 为显式的错误编号) 使用示例：

```
const errnum err10 = 10;

function void test_raise()
  twrite("raise the error\n");
  raise err10;

error
  if ERRNO == err10 then
    twrite("catch the error\n");
```

(continues on next page)

(continued from previous page)

```
  end
end

function void main()
  test_raise();
end
```

该例子中，raise 设置了一个编号 10 的错误，该错误随即在 error 处理程序中被捕获。

raise 直接使用的示例：

```
function num safediv(num x, num y)
  return x / y;

  error
    raise;
end

function void main()
  safediv(2, 0);

error
  trynext;
end
```

此例中函数 safediv() 发生错误时，会将错误报告到 main 函数的错误处理处处理。

4.9 retry 语句

错误处理的常用语句，用于从出错的语句处重新执行。

下面是一个简单示例：

```
function num safediv(num x, num y)
  return x / y;

error
  y = 1;
  retry;
end
```

此例中，如果函数运行出现错误，就会将 y 的值设为 1 然后从出错处重新执行，最终的返回值为 x 的值。

4.10 trynext 语句

错误处理的常用语句，用于从出错的语句的下一条语句处开始执行。

下面是一个简单示例：

```
function num safediv(num x, num y)
  var num a = 0;
  a = x / y;
  return a;
error
  trynext;
end
```

此例中，如果函数运行出现错误，`a = x / y;` 赋值失败，函数会继续执行 return 语句 `return a` ，返回值为 `0` 。

4.11 Connect 语句

用于将中断标识变量与中断处理函数关联。关于中断更多的介绍请参阅 中断这一章节。

下面是一个简单示例：

```
var intnum t1;

function void main()
  connect t1 with traptry;
end

trap traptry
  tp_write("this is a interrupt test");
end
```

4.12 waituntil 语句

用于轮询条件。该语句会持续轮询知道直到条件满足或者超时才会推出轮询执行下一条语句。其多用于轮询基于共享变量的条件，从而实现多任务同步。更多的介绍请参阅 多任务支持这一章节。

语句的格式为：

```
waituntil COND \pollrate = RATE \maxtime = TIME \timeflag = FLAG_VAR;
```

COND 可以为任意复杂的逻辑表达式，为强制参数，接下来介绍的三个参数为可选参数。pollrate 指定轮询的间隔时间，单位为秒。maxtime 指定轮询超时的时间，单位为秒。一旦超时发生，ERR_WAIT_MAXTIME 错误将会被发起，除非 timeflag 同时被指定，在这种情况下，运行时系统不会产生错误，但是设置 FLAG_VAR 为真。注意，FLAG_VAR 必须为 bool 类型的变量。

Chapter 5

函数

Tenon 程序的执行代码位于函数中，其中入口函数为 `main` 函数。

5.1 函数声明

函数可以有 1 个返回值，如果没有返回值则通过 `void` 来指出。函数的参数分为：必选参数和可选参数。下面的例子展示了如何声明一个函数：

```
function num increase(num x, num y = 1)
  return x + y;
end
```

其中 `x` 为必选参数，`y` 为可选参数。可选参数可以给出一个缺省值，在这里，`y` 的缺省值是 `1` 。

5.2 函数调用

调用函数，涉及到函数的传参和返回值。必选参数必须给出，可选参数可以给出，并且必须是位于所有的必选参数之后。下面的例子展示了函数调用：

```
function void main()
  var num a = 1;
  a = increase(a);
  a = increase(a, y=2);
end
```

第一次调用 `increase`，`y` 使用的是缺省值 `1` ； 第二次调用，`y` 使用的是给定值 `2` 。

Chapter 6

模块

Tenon 程序由一个或者多个模块组成。每个模块可以用来实现独立的功能。一个模块具有一个模块名，以及可选的模块属性。

模块属性用来配置该模块在机械臂运行系统中的权限，加载执行模式等，包括：

- sysmodule，表示该模块为系统模块
- noview，表示该模块的源代码对用户是不可见的
- nostepin，表示在单步执行模式下，不能单步进入该模块
- viewonly，表示该模块的源代码不能被用户修改

下面的例子用来声明一个系统模块：

```
module smod(sysmodule, viewonly)


end
```

系统模块在机械臂加电启动之后，会被自动加载到内存中，对所有的程序都可用。缺省没有指定属性的模块为普通模块，其只对当前程序可用。

Chapter 7

错误处理

错误处理是指在函数运行出错时执行的语句，它能够帮助我们从错误中恢复，从而保证任务顺利进行。

错误处理以 `error` 开头，直至函数声明结束。我们可以为错误处理添加编号，并通过 raise 语句将错误提交到指定的错误处理处。

语法为：

```
error
  语句列表

或：
error(表达式数列)
  语句列表
```

7.1 错误编号

Tenon 为每种错误类型提供了一个编号，例如 除数为零的编号为：`ERR_DIVZERO` 。

7.2 错误变量

错误变量 `ERRNO` 是一个用于保存错误编号的系统变量。

当函数执行出错后，错误编号就会被写入到 `ERRNO` 中，我们通过查询 `ERRNO` 的值来确定当前错误类型。

7.3 错误处理代码

下面是一个简单的错误处理示例：

```
function safediv(num x, num y)
  return x / y;

error
  if ERRNO == ERR_DIVZERO then
    return x;
  end
end
```

这个例子展示了函数中错误处理的大概模样：

在执行函数时，如果出现错误，系统会自动将 ERRNO 的值设为该错误对应的错误编号，并进入函数的错误处理部分。

在这一函数中，如果错误类型是除数为零 ($ERRNO == ERR\_DIVZERO$)，就会返回被除数的值。

7.4 长跳转

```
error(表达式数列)
  语句列表
```

长跳转通过表达式数列，有选择的接受同级函数的错误信号，或者是下级函数通过 raise 语句传递上来的错误信号。其所接受的错误信号，由表达式数列来表达。如果错误信号符合表达式数列的要求，则该错误进入错误处理的语句列表中被处理；否则，该错误被继续向上传递，直到到达顶层 main 函数，如果 main 函数仍然无法处理，则返回系统错误。目前的长跳转支持三种形式的表达式数列：数字常量、数字全局变量、long_jmp_all_err 。数字常量和数字全局变量，可以直接根据数字值过滤错误信号，不符合条件的继续向上传递；long_jmp_all_err 则接受所有的错误信号。三种形式的表达式数列既可以单独使用某一种，也可以随机组合混用，其效果和单独使用一致。

数字常量：

```
const errnum err10 = 10;

function void test_raise()
  twrite("raise the error");
  raise err10;

error(10)
  raise;
end
```

数字全局变量：

```
const errnum err10 = 10;

function void test_raise()
  twrite("raise the error");
  raise err10;

error(err10)
  raise;
end
```

long_jmp_all_err:

```
const errnum err10 = 10;

function void test_raise()
  twrite("raise the error");
  raise err10;

error
  raise;
end

function void test_raise1()
  test_raise();
end

function void main()
  test_raise1();

error(long_jmp_all_err)
  twrite("catch the error err10 in main()");
  return;
end
```

Chapter 8

中断

程序在执行任务的过程中有可能触发中断，既内部中断。同时也有可能收到外部中断。我们通过 connect 来关联中断标识变量与其处理函数 (trap 函数)。当中断发生时，程序通过传递中断标识变量来通知运行时环境执行关联的中断处理函数。

中断标识变量并不存放中断号，其作用仅限于标识中断处理函数。通常来说，用户程序会提前通过其他库函数来注册 （中断号，中断标识变量）键值对，这样当某一中断发生时，键值对中的中断标识变量会被举起，运行时环境则执行该标识变量关联的处理函数。

注意，中断可能发生在程序执行中的任何时候，中断处理函数执行完毕后将会返回程序被中断的位置继续执行。同时注意，内部中断仅当前任务可见。

8.1 中断标识变量

中断标识变量是用户声明的用以和 trap 函数关联的特殊变量，变量类型为 intnum 。一个中断标识变量只能与一个 trap 函数关联。

中断标识变量声明语句:

```
var intnum interrupt;
```

8.2 系统中断号变量

在某些时候，用户程序需要查看中断号。系统中断变量 INTNO 就是一个用于保存中断编号的系统变量。

当发生中断后，中断编号就会被写入到 INTNO 中，我们通过查询 INTNO 的值来获得当前中断编号。

8.3 中断处理函数

通过 trap 来声明一个 trap 函数，并以 end 结束声明。

通过 connect 语句将 trap 函数与中断标识变量关联。一个 trap 函数可以与多个中断标识变量关联。

下面是一个简单的中断处理示例：

```
module test

  opaque function void IError(num err_domain, num err_type, alias num err_val)
  end

  var intnum err_int;

  function num div_wrapper(num x, num y)
    return x / y;
  end

  function void main()

    connect err_int with ftrap;
    ! 4 = PROGRAM_ERR
    ! 3 = TYPE_ERR
    IError(4, 3, err_int);

    var num result;
    result = div_wrapper(10, 0);

  end

  trap ftrap
    twrite("caught error interrupt!");
    return;
  end

end
```

Tenon 运行时系统有一系列的内部预定义中断种类和终端号，比如程序除零属于 程序错误 (编号为 4) 这个类别中的 类型错误 (编号为 3)。上面的例子中，我们希望对除零错误进行捕获和处理。

程序首先通过 connect 来关联中断标识变量 err_int 和中断处理函数 ftrap 。然后通过库函数 IError 来通知运行时系统，当程序收到除零错误中断时候举起 err_int 中断错误标识变量，执行标识变量关联的中断

函数。

关于库函数 IError 的更多细节，请参考 语言扩展及 Tenon 函数这两个章节的介绍。

Chapter 9

多文件支持

Tenon 语言和 Tenon 运行时环境支持单任务多文件，即一个任务可以有由多个 Tenon 文件共同实现。而且，实现这个任务的多个文件之间的函数可以互相调用。下面举一个具体实例：

mult-0.t 文件：

```
module main_module
  function void main()
    var num local_a = 20;
    print(global_a, local_a);
  end
end
```

mult-1.t 文件：

```
module test_module
  var num global_a = 10;

  function void print(num a, num b)
    twrite_num(a);
    twrite_num(b);
  end
end
```

mult-0.t 和 mult-1.t 共同构成了一个任务。mult-0.t 中调用了 mult-1.t 中的 print() 函数，并且直接使用了 mult-1.t 里的 global_a 全局变量。在编译和运行这个多文件构成的任务的时候，可以有两种做法。第一种，使用 Tenon 编译器分别将多个文件编译成对应的多个 asm 文件，然后交给 Tenon 运行时环境去运行。第二种，在使用 Tenon 编译器将多个文件编译成一个 asm 文件，然后将这一个 asm 文件交给 Tenon 运行时环境去运行。目前，多个文件之间的函数引用可以支持多个文件分开编译成多个 asm 文件，而全局变量引用只能支持多个文件编译成一个 asm 文件。另外，Tenon 运行时环境既能支持 asm 文件输入，也能支持二进制

文件输入，这个功能在多文件情况下也可以使用。

Chapter 10

多任务支持

Tenon 运行时环境支持多任务并发运行，且会为每一个任务创建独立的运行时上下文。各任务拥有独立的代码和数据，在运行时互不干涉，并且每个任务也可以支持多文件。同时，Tenon 从语言和运行时也提供了必要的多任务通信机制，包括多任务共享变量，多任务中断通信等。子章节对这些机制进行近一步的介绍。

在这里，运行可以是执行，也可以是调试，还可以是程序反汇编等任何 Tenon 运行时环境所支持的运行模式。Tenon 的运行时环境提供相应的接口，用户可以自由的为各任务指定加载和运行模式。这种执行任务与非执行任务任意组合的模式提供给用户最大的使用自由。比如，用户编写了两个任务，第一个任务，负责监控某些环境参数，并由此来控制硬件 (比如机械臂)。而第二个任务则负责设置这些环境参数。现在，假设用户对第二个程序的部分代码有疑虑，那么可以同时加载这两个任务。第一个任务加载为执行任务，第二个任务加载为远程调试任务。这样用户可以远程调试第二个程序，并实时观察其对第一个程序的影响。

Tenon 运行时环境同时提供一个默认的主控后台任务。该任务于系统设定的网络地址和端口等待远程连接，其拥有所有其他任务的运行时信息。通过连接该主控后台任务，用户可以实时查询各个任务的运行情况，同时可以对各任务执行远程操作，比如挂起某任务或调试某任务等。

10.1 多任务共享变量

Tenon 语言支持 `shared` 作用域，表示被作用对象将在运行时全系统，多任务间可见。该作用域可作用于变量，亦或函数。我们通常可以通过申明 `shared` 变量在多任务间传递信息，实现经典的任务同步机制。Tenon 运行时环境将会保证不同任务对 `shared` 变量操作的原子性。

下例为基于共享变量实现经典的无锁生产者消费者同步。

生产者代码：

```
module producer

  shared var num produced_cnt = 0;
  shared var num consumed_cnt = 0;
```

(continues on next page)

(continued from previous page)

```
  shared var bool producer_terminated = false;

  function void main()
    while 1 == 1 do
      if consumed_cnt < produced_cnt then
        twrite("producer: consumer hasn't consumed the existing coin");
        waittime(0.01);
      else
        if produced_cnt == 10 then
          twrite("producer: produced all ten coins");
          producer_terminated = true;
          return;
        else
          twrite("producer: produced one coin");
          produced_cnt++;
        end
      end
    end
  end

end
```

消费者代码:

```
module consumer

  shared var num produced_cnt = 0;
  shared var num consumed_cnt = 0;
  shared var bool producer_terminated = false;

  function void main()

    while 1 == 1 do
      if produced_cnt > consumed_cnt then
        twrite("consumer: consumed one coin");
        consumed_cnt++;
      else
        if consumed_cnt == 10 then
          if producer_terminated then
            twrite("consumer: consumed all ten coins");
```

(continues on next page)

(continued from previous page)

```
            return;
          else
            twrite("consumer: waiting producer terminating...");
            waittime(0.005);
          end
        else
          twrite("consumer: got nothing in this round, waiting");
          waittime(0.005);
        end
      end
    end
  end

end
```

该用例完全基于共享变量 `produced_cnt`，`consumed_cnt` ， `producer_terminated` 在两个任务间同步信息，没有使用任何互斥锁。

注意，`shared` 变量需要在定义和使用任务中都需要进行申明。通常应具有相同的初始值，如果不同，则初始值为最先加载的任务中定义的值。

10.2 多任务共享变量轮询

一种非常常见的共享变量同步方式是变量轮询 (`polling`)。Tenon 对此有语言层面的直接支持。你可以使用 `waituntil` 语句来轮询某基于共享变量的条件，该语句将持续轮询，直到条件满足或超时才会继续执行下一条语句。

比如 任务 1 采用 `waituntil` 持续轮询 `startsync` 变量的值，只有当其值为真时，才会推出轮询转而执行下一条语句。

```
module task1

  shared var bool startsync = false;

  function void main()
    waituntil startsync;
  end

end
```

任务 2 在 10 秒之后设置共享变量 `startsync` 为真。任务 1 的轮询随之成功，任务退出轮询继续执行。

```
module task2

  shared var bool startsync = false;

  function void main()
    waittime(10);
    startsync = true;
  end

end
```

waituntil 还支持其他参数，详细信息请参阅语句介绍的章节。

10.3 多任务中断通信

在前面的中断章节我们提到 Tenon 支持内部中断和外部中断，后者主要用于多任务通信。当中断用于多任务间通信时，发出的中断需要在全系统可见。

Tenon 的运行时环境提供 signal 和 sigaction 这两个库函数。signal 负责发起指定的中断号，并且在全系统广播。sigaction 则是潜在的中断接收任务用来注册中断处理函数。

比如如下的例子演示如何在两个任务之间通过中断进行通信：

假设第一个任务负责监视硬件的管脚电平信号，当信号变化时候则发送全局中断，我们同时假设电平信号变化的中断号为 0 。

```
module sender

  function void main()
    var num sleep_sec = 1;
    waittime(sleep_sec);
    ! 0 = AIO_PIN_LEVEL_CHANGE
    var num signum = 0;
    ! Raise global interrupt
    signal(signum);
  end

end
```

第二个任务则等待系统电平信号改变，并在收到信息后打印信息。

```
module receiver

  var intnum irq_int;

  function void main()
    connect irq_int with ftrap;
    ! 0 = AIO_PIN_LEVEL_CHANGE
    sigaction(0, irq_int);
    var num sleep_sec = 2;
    waittime(sleep_sec);
  end

  trap ftrap
    twrite("caught extern pin leval change!");
    return;
  end

end
```

signal 和 sigaction 可以看作是对 connect 和其他中断库函数的简化。二者提供更加简单易用的中断通信接口。

通常来说，我们可以把对硬件的监测集中放置在一个任务中，然后通过 signal 将监测的结果广播给系统中的其他任务。

10.4 多任务多文件

Tenon 运行时环境可以支持多任务多文件，即：支持多个任务，而每个任务又支持多个文件。多任务多文件状态下，既要考虑 Tenon 语言对多任务的约束，也要考虑其对多文件的约束。

Chapter 11

Logging 系统

Logging 系统实现了 Tenon 用户输出 Log 信息的功能。目前，Logging 系统支持将用户所要输出的 Log 信息输出到控制台或是 Log 文件中。Log 文件通常存储在用户主目录下，以 tenon.log 命名；随着文件内容的不断增加，会有 tenon.logx 名字的 Log 文件不断的产生，其中 x 代表着 1、2、3…。

Logging 系统有一个 Log 控制器，用来控制整个系统的过滤机制和输出选项等。Logging 系统中的 Log 输出，可以通过 Log 控制器中的 Level 和 Marker 过滤。

11.1 Log 的 Level

Log 的 Level 分为：NOTSET, DEBUG, INFO, WARNING, ERROR 和 CRITICAL。每条 Log 信息中都包含着本条 Log 信息的 Level。同时，Log 控制器中也有一个 Level 用于过滤 Log 信息的输出。通过 getLogDefaultLevel() 可以获取当前 Log 控制器所设置的默认 Level，初始情况下该默认 Level 为 INFO。这种设置下，Logging 系统只会输出 Level 大于等于 INFO 的 Log 信息。而通过 setDefaultLevel(LogLevel level)，可以设置 Log 控制器的默认 Level，输入的 Level 必须严格等于 Log 的 Level，并且大小写敏感。其具体使用例子如下：

```
var string str = getLogDefaultLevel();
twrite_string(str);

var string str1 = "ERROR";
setLogDefaultLevel(str1);
```

11.2 Log 的 Marker

Log 的 Marker 是标示着 Log 信息的来源，可以用来标示该 Log 信息来源于一个程序中的某个部分，或者是某块业务。Marker 的值并不像 Level 一样是提前定义好的，而是一个开放的数据项，用户可以根据自己的需

求定义。每条 Log 信息都包含着本条信息的 Marker。同时，Log 控制器中也包含着一个用于过滤的 Marker 信息，只有当 Log 信息的 Marker 等于 Log 控制器中的 Marker 的时候，该条 Log 信息才会被输出。和 Level 一样，Log 控制器的 Marker 也可以通过接口获取和设置，具体例子如下：

```
var string str = getLogDefaultMarker();
twrite_string(str);

var string str1 = "TASK1";
setLogDefaultMarker(str1);
str = getLogDefaultMarker();
```

11.3 Log 的输出方式

前文提到 Log 可以被输出到控制台或者是 Log 文件中，这个也是通过对 Log 控制器中的 Handler 进行设置取控制的。Log 控制器中有一个 Handler，默认设置为 CONSOLE，这种情况下会默认将 Log 信息输出到控制台，如果将该设置项设置为 LOG_FILE，那么 Log 信息将会被输出到 Log 文件中。与前文 Level 和 Marker 不同的是，Handler 只存在于 Log 控制器中，具体的 Log 信息中不包含 Handler。Log 控制器中的 Handler 的获取和设置，具体例子如下：

```
var string str = getLogDefaultHandler();
twrite_string(str);

var string str1 = "LOG_FILE";
setLogDefaultHandler(str1);
```

11.4 Log 的输出函数

Log 的输出函数，根据所需要提供的信息不同，可以分为三类。第一类，用户需要提供 Log 信息所需的 Marker 信息、Level 信息和 Message 信息。函数原型及调用方式如下：

```
void log_marker(string marker, string level, string message)

log_marker("SYSTEM", "INFO", "Test the info level log message.");

log_marker("TASK1", "ERROR", "Test the error level log message.");
```

第二类，用户需要提供 Level 信息和 Message 信息，Marker 信息直接设置为 SYSTEM，方便不需要使用 Marker 选项的用户。函数原型及调用方式如下：

```
void log(string level, string message)

log("WARNING", "Test the warning level log message.");

log("ERROR", "Test the error level log message.");
```

第三类，用户只需要提供 Message 信息，Marker 信息直接设置为 SYSTEM，Level 信息则根据用户选择的函数直接确定。函数原型及调用方式如下：

```
void log_notset(string message)
void log_debug(string message)
void log_info(string message)
void log_warning(string message)
void log_error(string message)
void log_critical(string message)

log_notset("Test the notset level log message.");
log_debug("Test the debug level log message.");
log_info("Test the info level log message.");
log_warning("Test the warning level log message.");
log_error("Test the error level log message.");
log_critical("Test the critical level log message.");
```

Chapter 12

语言扩展

Tenon 具有很强的扩展性。它支持使用 opaque 函数声明一个接口，通过 C 语言进行扩展。

opaque 函数以 `opaque function` 开头，以 `end` 结束。与一般函数相比缺省了语句部分。

C 文件:

```
#include "tenon.h"

static void twrite_bool(TenonData data)
{
  bool v = tenon_get_bool_value(data);
  char *str = v ? "true" : "false";
  printf(str);
}

static void twrite_num(TenonData data)
{
  double v = tenon_get_num_value(data);
  printf("%0.14g", v);
}

static const TenonExtFunc base_funcs[] = {
  {"twrite_bool", twrite_bool},
  {"twrite_num", twrite_num},
  {NULL, NULL}, /* sentinel */
};

int init_base()
{
```

(continues on next page)

(continued from previous page)

```
    return tenon_init_lib("base", base_funcs);
  }

Tenon 文件:
  opaque function void twrite_num(num x)
  end

  function void main()
    twrite_num(1);
  end
```

首先我们在 Tenon 文件中声明一个 `opaque` 函数，在 C 语言中将其实现，并将其注册到 Tenon 中，最后就可以调用这个函数了。

Tenon 提供两种注册 C 语言扩展函数的方式：

- 静态注册
- 动态注册

对于静态注册，需要将该 C 语言扩展函数与 Tenon 运行时环境的代码一起编译，这样 Tenon 运行时环境将会默认支持该扩展函数。

相较于静态注册，动态注册是一种更加方便的使用方式。在动态注册模式下，用户不需要修改 Tenon 运行时环境的代码，而是将 C 扩展函数编译为动态库并通知运行时环境进行动态加载。例如, 对于上面例子中的 C 文件，如果我们保存为 my_ext.c，那么可以用如下的命令将其编译为动态库：

```
gcc -fPIC -shared -IPATH_TO_TENON_INCLUDE -o my_ext.so my_ext.c
```

命中中的 `-IPATH_TO_TENON_INCLUDE` 是指定 Tenon 开发头文件的位置。比如 Tenon 环境安装在/home/testusr/tenon，那么头文件在/home/testusr/tenon/include，所以需要指定 `-I/home/testusr/tenon/include` 。

接下来当运行 Tenon 程序的时候，使用 `-p` 参数通知运行时环境加载该动态库：

```
tvm -p my_ext.so prog.asm
```

此示例最后会打印 `1` 。

Chapter 13

Tenon 二进制存储格式

13.1 简介

Tenon 的运行时环境主要围绕 Tenon 指令集打造。Tenon 语言经编译器编译为以 Tenon 指令为主体的汇编文件后，以文本或二进制格式进行存储。Tenon 虚拟机对二者都可以识别并加载。文本存储同其他常见的汇编格式并无太大区别，具有很强的自释性，但虚拟机需要进行文本分析，所以加载速度较慢。与之对应，二进制存储加载速度较快，但不具有可读性。用户需要使用二进制分析工具对二进制文件进行查看。本文档主要针对二进制存储格式进行介绍。

Tenon 的二进制存储以 ELF 文件作为容器，利用 ELF 的各段来存储 Tenon 程序的不同部份, 同时利用符号表来呈现程序的逻辑关系。在理解 Tenon 的二进制格式后，用户可以使用常见的 ELF 工具对程序 (readelf, objdump 等) 进行概要查看亦或详细分析。比如，对于下一章中的 test.t 用例，其二进制存储概览如下：

```
readelf -Wa tvm.out 2>/dev/null

节头：
  [Nr] Name              Type
  [ 0]                   NULL
  [ 1] .text             PROGBITS
  [ 2] .erange           PROGBITS
  [ 3] .gdata            NOBITS
  [ 4] .stack            NOBITS
  [ 5] .rodata           PROGBITS
  [ 6] .nconst           NOBITS
  [ 7] .bconst           NOBITS
  [ 8] .sconst           NOBITS
  [ 9] .rel.text         REL
  [10] .symtab           SYMTAB
```

(continues on next page)

(continued from previous page)

```
  [11] .ettype           PROGBITS
  [12] .ttype            PROGBITS
  [13] .strtab           STRTAB

Symbol table '.symtab' contains 48 entries:
 Num:    Value  Size Type    Bind   Vis      Ndx Name
   ..
   3: 00000001     0 OBJECT  GLOBAL DEFAULT    5 ERR_ALRDYCNT
  ...
  35: 00000000     0 OBJECT  GLOBAL DEFAULT    3 global_addend
  36: 00000000     4 FUNC    WEAK   DEFAULT    1 foo
  37: 00000000     4 FUNC    WEAK   DEFAULT    1 main
  ...
  41: 00000000     0 MODULE  GLOBAL DEFAULT    1 test
  42: 00000000     8 FUNC    GLOBAL DEFAULT    1 __tvm_module_onload
  43: 00000008    36 FUNC    GLOBAL DEFAULT    1 foo
  44: 00000000     0 OBJECT  LOCAL  DEFAULT    4 input
  45: 00000000     0 OBJECT  LOCAL  DEFAULT    4 addend
  46: 0000002c    48 FUNC    GLOBAL DEFAULT    1 main
  47: 00000000     0 OBJECT  LOCAL  DEFAULT    4 sum
```

程序的层级结构和基本信息可以很方便的从 readelf 等通用 ELF 工具获取。

```
! 显示所有的信息
readelf -Wa BINARY_NAME
```

```
! 只显示重定位信息
readelf -Wr BINARY_NAME
```

本文档不要求读者预先对 ELF 格式有了解，但理解后者可以帮助深入理解 Tenon 的二进制存储格式。ELF 格式可以参阅 http://www.skyfree.org/linux/references/ELF_Format.pdf。

13.2 Tenon 程序的构成

一个标准的 Tenon 程序，由模组 (MODULE) 及其作用域内的函数 (FUNCTION) 组成，包含各个层次的数据和代码。比如对于下图所示的代码 test.t：

```
module test


  var num global_addend = 20;
```

(continues on next page)

(continued from previous page)

```
  function num foo(num input)
    var num addend = 10;

    return addend + input;
  end

  function num main()
    var num sum = 3;

    sum = foo(sum);

    return sum + global_addend;
  end

end
```

其上层组织结构为一个模组，下辖一个全局变量和两个函数。每个函数内部又包含局部变量。

通常来说，一个 Tenon 程序可以包含多个模组，每个模组又可以包含多个函数。全局变量存放在全局变量表里面，全程序可见。程序里所有的数字和字符串常量存放在全局常量表里，全程序可见。每个函数有自己的局部变量表。同时每个函数还有自己的异常处理地址表。程序还可能包含重定位信息，有些是局部重定位，有些是全局重定位，通过重定位类型体现。另外，Tenon 程序的变量在运行时仍然有类型信息。

Tenon 的二进制存储格式需要能承载以上所描述的所有信息。

13.3 Tenon 二进制存储格式详解

在前面的章节中我们已经提到 Tenon 的二进制存储格式复用了业界标准的 ELF 格式作为容器。通过使用各种标准或者扩展的 ELF 特性，我们可以把构成 Tenon 程序各个组件完整的映射到 ELF 文件。

13.3.1 ELF 段详解

程序代码和数据实际存放在 ELF 段中。使用标准的段类型来标识一个段是（真段）否（伪段）包含实际内容。PROGBITS 意为包含实际内容，NOBITS 表示该段不包含实际内容。Tenon 所有的变量都在运行时分配存储空间，所以变量相关的段都为伪段。以下为所有用到的 ELF 段，可以通过下面的命令显示。

```
! 只显示文件段信息
readelf -WS BINARY_NAME
```

ELF 段	类型	含义
.text	PROGBITS	存放所有的程序代码
.erange	PROGBITS	存放所有的函数异常处理地址信息
.gdata	NOBITS	伪段，指向该段表示一个变量是全局变量
.stack	NOBITS	伪段，指向该段表示一个变量是局部变量
.rodata	PROGBITS	存放所有的常量
.nconst	NOBITS	伪段，指向该段表示一个常量是数字类型
.bconst	NOBITS	伪段，指向该段表示一个常量是布尔类型
.sconst	NOBITS	伪段，指向该段表示一个常量是字符串类型
.rel.text	REL	存放所有的重定位项
.symtab	SYMTAB	符号表
.ettype	PROGBITS	扩展类型描述段
.ttype	PROGBITS	类型段
.strtab	STRTAB	字符串表

13.3.2 ELF 符号详解

程序的组成单元和相关逻辑关系通过 ELF 符号体现。理解 ELF 符号表对理解 Tenon 的二进制存储格式至关重要。可以通过如下的命令显示符号表。

```
! 只显示符号表信息
readelf -Ws BINARY_NAME
```

模组（MODULE）

每一个模组都在 ELF 符号表中拥有一个全局符号，且符号类型为 STT_MODULE（Teonon 扩展）

```
41: 00000000     0 MODULE  GLOBAL DEFAULT    1 test
```

标准的 readelf 目前没有 Tenon 扩展支持，所以实际显示为：

```
41: 00000000     0 < 处理器专用>: 13 GLOBAL DEFAULT    1 test
```

ELF 符号域	含义
Value	保留
Size	保留
TYPE	STT_MODULE (Tenon 扩展，实际值为 STT_LOPROC
Bind	STB_GLOBAL
Vis	保留
Ndx	对应的代码段索引
Name	模组名字

函数（FUNCTION）

每一个函数都在 ELF 符号表中拥有一个全局符号，且符号类型为 STT_FUNC

```
43: 00000008    36 FUNC    GLOBAL DEFAULT    1 foo
```

ELF 符号域	含义
Value	函数起始地址在代码段中的偏移
Size	函数代码段的大小
TYPE	STT_FUNC
Bind	STB_GLOBAL
Vis	保留
Ndx	对应的代码段索引
Name	函数名字

以上的函数符号同前面介绍过的模组符号用于二进制的消费者理解并建立程序的运行时单元。

全局变量

每一个全局变量都在 ELF 符号表中拥有一个全局符号，且符号类型为 STT_OBJECT

```
35: 00000000     0 OBJECT  GLOBAL DEFAULT    3 global_addend
```

ELF 符号域	含义
Value	变量的私有标识
Size	变量类型在类型表中的索引
TYPE	STT_OBJECT
Bind	STB_GLOBAL
Vis	保留
Ndx	对应的数据段索引
Name	变量名字

局部变量

每一个局部变量都在 ELF 符号表中拥有一个局部符号，且符号类型为 STT_OBJECT。注意，函数的局部变量符号会紧跟函数符号依序输出。

```
44: 00000000     0 OBJECT  LOCAL  DEFAULT    4 input
```

ELF 符号域	含义
Value	变量的私有标识
Size	变量类型在类型表中的索引
TYPE	STT_OBJECT
Bind	STB_LOCAL
Vis	保留
Ndx	对应的数据段索引
Name	变量名字

以下为局部变量依序输出的例子：

```
239: 00000744    56 FUNC    GLOBAL DEFAULT    1 distance
240: 00000000     6 OBJECT  LOCAL  DEFAULT    4 point1
241: 00000000     6 OBJECT  LOCAL  DEFAULT    4 point2
242: 00000000     0 OBJECT  LOCAL  DEFAULT    4 dist
243: 0000077c    56 FUNC    GLOBAL DEFAULT    1 dotprod
244: 00000000     6 OBJECT  LOCAL  DEFAULT    4 vector1
245: 00000000     6 OBJECT  LOCAL  DEFAULT    4 vector2
246: 00000000     0 OBJECT  LOCAL  DEFAULT    4 val
247: 000007b4    40 FUNC    GLOBAL DEFAULT    1 exp
248: 00000000     0 OBJECT  LOCAL  DEFAULT    4 exponent
249: 00000000     0 OBJECT  LOCAL  DEFAULT    4 value
```

在上表中，有三个函数 distance，dotprod 和 exp，分别包含有各自的局部变量。可以看到，这些局部变量都

是紧跟各自的函数符号输出的。

13.3.3 常量

Tenon 里面有数字，布尔和字符串这三种常量。常量只有内容，没有名字，比如 1, false, “hello”。每一个常量在符号表中都拥有一个全局符号。通过根据该常量的类型，符号的各域有不同的解释。

```
105: 00000000 0x3ff00000 OBJECT  GLOBAL DEFAULT    6 __tvm_const_tbl_entry_1
107: 0000048e     0 OBJECT  GLOBAL DEFAULT    8 __tvm_const_tbl_entry_3
116: 00000000     0 OBJECT  GLOBAL DEFAULT    7 __tvm_const_tbl_entry_12
```

ELF 符号域	含义
Value	对于数字常量，表示低 32 位。对于布尔常量，表示布尔值。对于字符串常量，表示字符串表偏移。
Size	对于数字常量，表示高 32 位
TYPE	STT_OBJECT
Bind	STB_GLOBAL
Vis	保留
Ndx	对应的常量段索引，可以区分常量类型。索引为.nconst，表示数字类型。索引为.bconst，表示布尔类型。索引为.sconst，表示字符串类型。
Name	常量的伪名

13.3.4 变量类型映射

类型段

上面的介绍提到，全局和局部变量符号的 Size 域均存储该变量在类型表中的索引。Tenon 程序中的所有类型都是存储在.ttype 和.ettype 这两个段中的。

.ttype 是基础类型段。实际由一项项的基本类型描述构成。每一项的结构如 TExtType 所示。

```
typedef struct ext_type
{
  TType t;
  union
  {
    unsigned n_record;
    ArrayType t_array;
    unsigned t_ptr;
```

(continues on next page)

(continued from previous page)

```
  };
} TExtType;
```

TType 的编码为：

```
typedef enum
{
  TYPE_UNKNOWN,
  TYPE_BOOL,
  TYPE_NUM,
  TYPE_OPAQUE,
  TYPE_STR,
  TYPE_RECORD,
  TYPE_ARRAY,
  TYPE_FUNC,
  TYPE_PTR,
  TYPE_MAX,
} TType;
```

TExtType 域	含义
t	TType 编码
n_record	如果 TType 为 RECORD，记录 RECORD 类型的名字在字符表中的偏移
t_array	如果 TType 为 ARRAY，该项描述数组的构成
t_ptr	如果 TType 为 PTR，该项为所指类型在表中的索引

数组类型的描述为：

```
typedef struct array_type
{
  TType elem_t;
  unsigned elem_t_name;
  int x;
  int y;
  int z;
} ArrayType;
```

array_type 域	含义
elem_t	数组元素的 TType 编码
elem_t_name	如果数组元素为 RECORD，记录 RECORD 类型的名字在字符表中的偏移
x	数组的 X 维度
y	数组的 Y 维度
z	数组的 Z 维度

扩展类型描述段

在上面的章节，我们没有列出记录（RECORD）类型的描述。这是因为 Record 类型具有不定的长度，可以包含任意的子域组合，所以需要单独的变长描述结构。该描述单独放在.ettype 段中。

.ettype 段由一系列变长描述单元组成。每一个单元包含一个固定的描述头和任意长度的子域。其中描述头的定义为：

```
typedef struct
{
  unsigned int flag;
  unsigned int name;
  unsigned int field_cnt;
} ElfETypeRecord;
```

ElfETypeRecord 域	含义
flag	类型标志位
name	类型名字在字符串表中的偏移
field_cnt	子域的数量

子域描述为：

```
typedef struct
{
  unsigned int name;
  unsigned int t_idx;
} ElfETypeField;
```

ElfETypeField 域	含义
name	子域名字在字符串表中的偏移
t_idx	子域类型索引

13.3.5 重定位信息

Tenon 的重定位描述完全复用 ELF，且使用 REL 描述，不使用 RELA。.rel.text 段由若干 ElfRel 组成。

```
typedef struct
{
  unsigned char r_offset[4];
  unsigned char r_info[4];
} ElfRel;

#define ELF_R_SYM(i)      ((i) >> 8)
#define ELF_R_TYPE(i)     ((i) & 0xff)
#define ELF_R_INFO(s, t)  (((s) << 8) + ((t) & 0xff))
```

ElfRel 域	含义
r_offset	重定位需要改写的代码的函数相对偏移
r_info	TYPE 子域包含重定位类型，SYM 子域包含重定位符号索引

Tenon 目前支持的重定位类型有：

```
typedef enum {
  TVM_RELOC_NONE,
  TVM_RELOC_J_7,
  TVM_RELOC_J_14,
  TVM_RELOC_C_7,
  TVM_RELOC_C_20,
  TVM_RELOC_DYN_C_7,
  TVM_RELOC_LGR_GDATA_7,
  TVM_RELOC_LGR_GDATA_14,
  TVM_RELOC_LGR_GDATA_19,
  TVM_RELOC_LGR_GDATA_23,
  TVM_RELOC_G_GDATA_7,
} RelocType;
```

其中带有 DYN 关键字的为运行时重定位。

13.3.6 函数异常处理地址表

Tenon 函数可能有异常处理地址表，也可能没有。所有的的异常处理地址表都存放在.erange 段中，该段由如下的表项组成：

```
typedef struct elf_erange
{
  unsigned int start;
  unsigned int end;
} ElfRange;
```

一个函数的异常处理地址对依序输出到.erange 段，且以一个特俗对 {-1，0} 表示该函数的地址对输出结束。如果一个函数没有异常处理地址对，那么其也会输出一个 {-1，0} 特殊对。

13.3.7 函数错误处理钩子函数地址

Tenon 函数可能有错误处理钩子函数，其地址通过一个特殊的符号 ___err_handler_FUNC_NAME 来标识。比如：

```
43: 00000004    52 FUNC    GLOBAL DEFAULT    1 test_raise
44: 00000018     0 NOTYPE  LOCAL  DEFAULT  ABS __err_handler_test_raise
```

标识 test_raise 这个函数有错误处理钩子函数，其实地址为 test_raise 函数的基地址偏移 0x18，该偏移放在符号的 Value 域中。

Chapter 14

Tenon 调试器

14.1 简介

Tenon 语言支持汇编级和源代码级调试。通过设置断点，你可以挂起被调试程序。然后进行变量检查，函数调用栈回溯，代码查看等常用调试操作。结束检查后可以继续执行被调试程序。调试命令跟 Linux 下常用的调试器 Gdb 类似。

Tenon 调试器提供最大化的命令自动补全支持。无论当你记不清楚一个命令的完整拼写，抑或是不知道命令参数，在或是不知道调试器在当前的上下文中可以提供何种的调试支持，你都可以使用 tab 来进行补全问询。调试器将会自动补全你的命令，参数，或是列出在当前上下文中所能支持的功能和对应的命令。

在下面的章节中，我们将使用如下的 Tenon 程序作为调试范例。

```
demo.t

module debug_demo
  num global=100;
  function void numaddref(alias num x)
    x = x + 1;
  end

  function void numaddval(num x)
    x = x + 1;
  end

  function void main()
    var num i = 1;
    numaddref(i);
    numaddval(i);
```

(continues on next page)

(continued from previous page)

```
    numaddref(i);
  end
end
```

当编译 Tenon 程序的时候，使用-g 可以生成源代码级调试信息。目前的调试信息主要包括行号信息和源文件路径。

```
tenonc -g demo.t
```

检查生成的汇编文件，你可以看到源文件路径信息：

```
.file "/home/jwang/core/testsuite/tasm/test.t"
```

还可以看到行号信息：

```
.loc 2 37
alloca  num* %x
poparg  %x
```

这些信息是源代码级调试所必须的。如果调试器在加载被调试程序时不能找到这些信息，则会使用汇编级调试模式。

Tenon 调试器集成在 Tenon 虚拟机中。加载程序时，指定-i 会打开调试模式：

```
tvm -i task0.asm
```

14.2 调试命令

本部分内容介绍 Tenon 调试器支持的调试命令。当描述命令格式的时候，文档遵循如下的约定：

- *[]* 描述必须参数。
- *{}* 描述可选参数。
- | 描述不同的参数选择。

Tenon 调试器支持调试命令局部匹配，当输入的字符串是且仅是某一命令名的前置部分时，调试器会成功匹配该命令。比如，*hi* 会匹配 *history* , 因为它是后者的前两个字符, 而且没有任何其他命令的前两个字符是 *hi* 。

同时，某些命令有固定的特殊简写，这些简写是为了跟 *GDB* 保持兼容，比如 *b* 可同时匹配多个命令，但我们将其绑定到 *breakpoint* 。这些固定的简写在 简写行列出。

14.2.1 backtrace

功能介绍：打印出当前的函数调用回溯栈。

简写：*bt*

命令格式：无参数

样例：

```
tvm ) bt
#0: numaddval +0
#1: main +8
```

14.2.2 breakpoint

功能介绍：在指定的位置设置断点。

简写：*b*

命令格式：

- *breakpoint* ，不带任何参数，在当前 *pc* 设置断点。
- *breakpoint {模组名:}{函数名:}[行号:* * *指令索引]* ，在指定的函数行号或者指令索引设定断点。

样例：

```
//在当前 PC 设置断点
tvm ) b
```

```
//在 test 模组的 main 函数起始设置断点
tvm ) b debug_demo:main
```

```
//在 test 模组的 main 函数的第十行设置断点
tvm ) b debug_demo:main:10
```

```
//在 test 模组的 main 函数的第三条指令设置断点
tvm ) b debug_demo:main:*3
```

14.2.3 continue

功能介绍：继续执行被调试程序。

简写：*c*

命令格式：无参数

样例：

```
tvm ) c
```

14.2.4 delete

功能介绍：永久删除所有的断点。

简写：*d*

命令格式：*{无参数 | 断点索引}*

样例：

```
// 删除所有断点
tvm ) delete
// 删除第三个断点
tvm ) delete 3
```

14.2.5 disable

功能介绍：临时关闭指定的断点。

简写：*disa*

命令格式：*disable {无参数 | 断点索引}*

样例：

```
// 关闭所有断点
tvm ) disable
// 关闭第三个断点
tvm ) disable 3
```

14.2.6 disassemble

功能介绍：打印出指定函数的汇编指令。

简写：无

命令格式：*disassemble {函数名}*，当不带函数名或者函数名为 *all* 时，打印出所有函数的汇编指令。

样例：

```
tvm ) disassemble
debug_demo:main:
->      0: alloca       *num %i
        1: move %1, 1
        2: store        %i, %1
```

-> 所指为当前 *pc* 对应的指令。

14.2.7 enable

功能介绍：重新使能指定的断点。

简写：无

命令格式：*enable {无参数 | 断点索引}*

样例：

```
// 使能所有断点
tvm ) enable
// 使能第三个断点
tvm ) enable 3
```

14.2.8 help

功能介绍：打印帮助信息。

简写：*h*

命令格式：无参数

样例：

```
tvm ) help
Commands list:
        backtrace       - backtrace call frames
        breakpoint      - set breakpoint at func:offset
        continue        - continue the execution to the end
        delete          - delete all breakpoints
        disable         - disable one breakpoint
        disassemble     - disassemble the specified function
        ...
```

14.2.9 history

功能介绍：打印或者重新执行历史命令。

简写：无

命令格式：*history {命令索引}*

样例：

```
//打印历史命令
tvm ) history
   1:   help
   2:   history
```

```
//重新执行历史命令
tvm ) history 1
```

14.2.10 list

功能介绍：列出源代码。可以从当前行号列出附近的十行源代码，或者从指定的行号开始列出代码，或者从当前行号 *h* 倒退指定的行数列出代码。

简写：*l*

命令格式：*list {起始行号 | -倒退行数 }*

样例：

```
tvm ) l 14
  10:   function void main()
  11:     var num i = 1;
  12:     numaddref(i);
  13:     numaddval(i);
  14:     numaddref(i);
  15:     end
tvm ) l -15
   1: module debug_demo
   2:   function void numaddref(alias num x)
   3:     x = x + 1;
   4:   end
```

当不带任何行号参数时，*list* 从当前行号继续列出代码。

14.2.11 next

功能介绍：源代码级单步。如果遇到函数调用，不会单步进入该被调用函数。

简写：*n*

命令格式：*next {行数量}*，指定行数量可以进行多行单步。

样例：

```
tvm ) b main
tvm ) r
Breakpoint 0, debug_demo:main +0 at test.t:11
  11:     var num i = 1;
tvm ) n
  12:     numaddref(i);
tvm ) n
  13:     numaddval(i);
```

上面的例子中，第十二行是一个函数调用，*next* 单步经过该调用，没有进入被调函数。

14.2.12 print

功能介绍：打印程序变量信息，或者执行函数调用 *(dummy call)*。

简写：*p*

命令格式：*print {@ 全局变量名 | % 局部变量或寄存器名 | 函数调用}*

样例：

```
// 打印局部变量
tvm ) p %x
*num           0
// 打印寄存器
tvm ) p %1
num            1
// 打印全局变量
tvm ) p @global
*num           7
// 执行函数调用 (dummy call)
tvm ) p main()
```

14.2.13 quit

功能介绍：退出调试器终端。

简写：*q*

命令格式：无参数

14.2.14 run

功能介绍：从头重新执行调试程序

简写：*r*

命令格式：无参数

样例：

```
tvm ) b main
tvm ) r
Breakpoint 0, debug_demo:main +0 at test.t:11
  11:     var num i = 1;
```

14.2.15 show

功能介绍：显示调试器内部状态，目前支持显示已经设置的断点。

简写：无

命令格式：*show {breakpoint}*

样例：

```
tvm ) b main
tvm ) b main:*1
tvm ) b main:*3
tvm ) show breakpoint
1:      debug_demo:main +0000   active
2:      debug_demo:main +0001   active
3:      debug_demo:main +0003   active
```

14.2.16 step

功能介绍：源代码级单步 *(*单步进入函数*)*。该命令跟 *next* 相仿，但遇到函数调用，会 单步进入该被调用函数。

简写：*s*

命令格式：*step {行数量}*，指定行数量可以进行多行单步。

样例：

```
tvm ) b main
tvm ) r
Breakpoint 0, debug_demo:main +0 at test.t:11
  11:     var num i = 1;
tvm ) n
Breakpoint 3, debug_demo:main +3 at test.t:12
  12:     numaddref(i);
tvm ) s
  2:   function void numaddref(alias num x)
tvm ) bt
#0: numaddref +0
#1: main +5
```

上面的例子中，第十二行是一个函数调用，*step* 单步经过该调用，进入了被调函数。

14.2.17 stepi

功能介绍：汇编级单步。该命令跟 *next* 和 *step* 相仿，但单步单条汇编指令，遇到函数调用，会始终 单步进入该被调用函数。

简写：*si*

命令格式：*stepi {指令数量}*，指定指令数量可以进行多指令单步。

14.3 调试脚本

调试脚本指包含一系列调试命令的文本文件。当启动调试器的命令行参数中带有调试脚本时，Tenon 调试器会首先加载该脚本，逐条执行脚本中的命令，然后才会进入到交互式命令行。调试脚本通常包含一系列调试准备命令，避免重复性的手工输入。

调试脚本通过 *-x* 指定。比如我们希望对程序运行到第 20 行时的各个变量进行检查，但是在这之前，我们也希望知道变量 *a* 在第 3，6，9 行的变量值。那么我们可以编写如下的脚本:

```
cat demo.x

! 设置各断点
br main:3
br main:6
```

(continues on next page)

(continued from previous page)

```
br main:9
br main:20
! 启动程序
run
! 打印 a 在第 3, 6, 9 行的值
print a
continue
print a
continue
print a
! 继续执行程序, 程序触发第 20 行的断点
continue
! 脚本结束, 进入交互模式
```

! 在脚本中是注释标识符。然后，通过如下的命令通知调试器加载调试脚本

```
tvm -x demo.x -i demo.t
```

如果你希望调试器将脚本中的命令运行完毕后直接退出，不进入交互模式，那么只需要将 *quit* 作为最后一条命令，比如：

```
cat demo.x

! 设置各断点
br main:3
...
quit
```