

8.3.1 x

8.4 Lua 语法基础

8.4.1 注释

Lua 的行注释为两个连续的减号，段注释以--[[开头，以--]]结尾。

不过，在调试过程中如果想临时取消段注释，而直接将其标识删除，这样做其实并不好。因为有可能还需要再添加上。而段注释的写法相对较麻烦。所以，Lua 给出了一种简单处理方式：在开头的--[[前再加一个减号，即可使段注释不起作用。其实就是使两个段注释标识变为了两个行注释。

8.4.2 数据类型

Lua 中有 8 种类型，分别为：nil、boolean、number、string、userdata、function、thread 和 table。通过 type() 函数可以查看一个数据的类型，例如，type(nil) 的结果为 nil，type(123) 的结果为 number。

数据类型	描述
nil	只有值 nil 属于该类，表示一个无效值，与 Java 中的 null 类似。但在条件表达式中相当于 false。
boolean	包含两个值：false 和 true。
number	表示双精度类型的实浮点数。
string	字符串，由一对双引号或单引号括起来。当一个字符串包含多行时，可以在第一行中以[[开头，在最后一行中以]]结尾，那么在[[与]]括起来的这多行内容就是一个字符串。换行符为字符串"\n"。
table	类似于 Java 中的数组，但比数组的功能更强大，更灵活。
function	由 C 或 Lua 编写的函数。
thread	协同线程，是协同函数的执行体，即正在执行的协同函数。
userdata	一种用户自定义数据，用于表示一种由应用程序或 C/C++ 语言库所创建的类型，可以将任意 C/C++ 的任意数据类型的数据存储到 Lua 变量中调用。

8.4.3 标识符

程序设计语言中的标识符主要包含保留字、变量、常量、方法名、函数名、类名等。Lua 的标识符由字母、数字与下划线组成，但不能以数字开头。Lua 是大小写敏感的。

(1) 保留字

Lua 常见的保留字共有 22 个。不过，除了这 22 个外，Lua 中还定义了很多的内置全局变量，这些内置全局变量的一个共同特征是，以下划线开头后跟全大写字母。所以我们在定义自己的标识符时不能与这些保留字、内置全局变量重复。

and	break	do	else
elseif	end	false	for
function	if	in	local
nil	not	or	repeat
return	then	true	until
while	goto		

(2) 变量

Lua 是弱类型语言，变量无需类型声明即可直接使用。变量分为全局变量与局部变量。Lua 中的变量默认都是全局变量，即使声明在语句块或函数里。全局变量一旦声明，在当前文件中的任何地方都可访问。局部变量 local 相当于 Java 中的 private 变量，只能在声明的语句块中使用。

(3) 动态类型

Lua 是动态类型语言，变量的类型可以随时改变，无需声明。

8.4.4 运算符

运算符是一个特殊的符号，用于告诉解释器执行特定的数学或逻辑运算。Lua 提供了以下几种运算符类型：

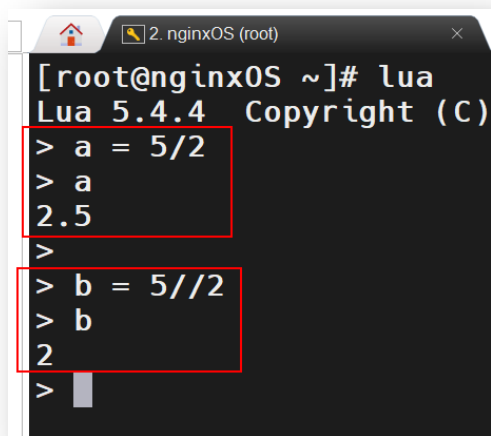
- 算术运算符
- 关系运算符

- 逻辑运算符
- 其他运算符

(1) 算术运算符

下表列出了 Lua 语言中的常用算术运算符，设定 A 的值为 10，B 的值为 20：

操作符	描述	示例
+	加法	A + B 输出结果 30
-	减法	A - B 输出结果 -10
*	乘法	A * B 输出结果 200
/	除法	5 / 2 输出结果 2.5
%	取余	B % A 输出结果 0
^	乘幂	A^2 输出结果 100
-	负号	-A 输出结果 -10
//	整除运算符(>=lua5.3)	5//2 输出结果 2



```

[root@nginx05 ~]# lua
Lua 5.4.4 Copyright (C)
> a = 5/2
> a
2.5
>
> b = 5//2
> b
2
>
  
```

注意，

- SciTE 对 Lua 支持的目前最高版本为 5.1，而整除运算符//需要在 Lua5.3 版本以上，所以当前 SciTE 中无法看到效果。
- 在命令行模式中，直接输入变量名回车，即相当于 print()函数输出该变量。

(2) 关系运算符

下表列出了 Lua 语言中的常用关系运算符，设定 A 的值为 10，B 的值为 20：

操作符	描述	示例
<code>==</code>	等于	<code>(A == B)</code> 为 <code>false</code> 。
<code>~=</code>	不等于	<code>(A ~= B)</code> 为 <code>true</code> 。
<code>></code>	大于	<code>(A > B)</code> 为 <code>false</code> 。
<code><</code>	小于	<code>(A < B)</code> 为 <code>true</code> 。
<code>>=</code>	大于等于	<code>(A >= B)</code> 返回 <code>false</code> 。
<code><=</code>	小于等于	<code>(A <= B)</code> 返回 <code>true</code> 。

(3) 逻辑运算符

注意，Lua 系统将 `false` 与 `nil` 作为假，将 `true` 与非 `nil` 作为真，即使是 0 也是真。

下表列出了 Lua 语言中的常用逻辑运算符，设定 A 的值为 `true`，B 的值为 `false`：

操作符	描述	示例
<code>and</code>	逻辑与	<code>(A and B)</code> 为 <code>false</code> 。
<code>or</code>	逻辑或	<code>(A or B)</code> 为 <code>true</code> 。
<code>not</code>	逻辑非	<code>not(A and B)</code> 为 <code>true</code> 。

(4) 其他运算符

下表列出了 Lua 语言中的连接运算符与计算表或字符串长度的运算符：

操作符	描述	示例
<code>..</code>	字符串连接符。 (两个点)	<code>a..b</code> ，其中 a 为 "Hello"，b 为 "World"，输出结果为 "Hello World"。
<code>#</code>	返回字符串或表的长度。	<code>#"Hello"</code> 返回 5。

8.4.5 函数

Lua 中函数的定义是以 `function` 开头，后跟函数名与参数列表，以 `end` 结尾。其可以没有返回值，也可以一次返回多个值。

（1）固定参函数

Lua 中的函数在调用时与 Java 语言中方法的调用是不同的，其不要求实参的个数必须与函数中形参的个数相同。如果实参个数少于形参个数，则系统自动使用 `nil` 填充；如果实参个数多于形参个数，多出的将被系统自动忽略。

（2）可变参函数

在函数定义时不给出具体形参的个数，而是使用三个连续的点号。在函数调用时就可以向该函数传递任意个数的参数，函数可以全部接收。

（3）可返回多个值

Lua 中的函数一次可以返回多个值，但需要多个变量来同时接收。

（4）函数作为参数

Lua 的函数中，允许函数作为参数。而作为参数的函数，可以是已经定义好的普通函数，也可以是匿名函数。

8.4.6 流程控制语句

Lua 提供了 `if` 作为流程控制语句。

（1）if 语句

Lua 提供了 `if...then` 用于表示条件判断，其中 `if` 的判断条件可以是任意表达式。Lua 系统将 `false` 与 `nil` 作为假，将 `true` 与非 `nil` 作为真，即使是 `0` 也是真。

需要注意，Lua 中的 `if` 语句的判断条件可以使用小括号括起来，也可以不使用。

(2) if 嵌套语句

Lua 中提供了专门的关键字 `elseif` 来做 if 嵌套语句。注意，不能使用 `else` 与 `if` 两个关键字的联用形式，即不能使用 `else if` 来嵌套 if 语句。

8.4.7 循环控制语句

Lua 提供了四种循环控制语句：`while...do` 循环、`repeat...until` 循环、数值 `for` 循环，及泛型 `for` 循环。同时，Lua 还提供了 `break` 与 `goto` 两种循环流程控制语句。

(1) while...do

只要 `while` 中的条件成立就一直循环。

(2) repeat...until

`until` 中的条件成立了，循环就要停止。

(3) 数值 for

这种 `for` 循环只参用于循环变量为数值型的情况。其语法格式为：

```
for var=exp1, exp2, exp3 do
    循环体
end
```

`var` 为指定的循环变量，`exp1` 为循环起始值，`exp2` 为循环结束值，`exp3` 为循环步长。步长可省略不写，默认为 1。每循环一次，系统内部都会做一次当前循环变量 `var` 的值与 `exp2` 的比较，如果 `var` 小于等于 `exp2` 的值，则继续循环，否则结束循环。

(4) 泛型 for

泛型 `for` 用于遍历 `table` 中的所有值，其需要与 Lua 的迭代器联合使用。后面 `table` 学习时再详解。

(5) break

`break` 语句可以提前终止循环。其只能用于循环之中。

(6) goto

goto 语句可以将执行流程无条件地跳转到指定的标记语句处开始执行，注意，是开始执行，并非仅执行这一句，而是从这句开始后面的语句都会重新执行。当然，该标识语句在第一次经过时也是会执行的，并非是必须由 goto 跳转时才执行。

语句标记使用一对双冒号括起来，置于语句前面。goto 语句可以使用在循环之外。

注意，Lua5.1 中不支持双冒号的语句标记。

8.5 Lua 语法进阶

8.5.1 table

(1) 数组

使用 table 可以定义一维、二维、多维数组。不过，需要注意，Lua 中的数组索引是从 1 开始的，且无需声明数组长度，可以随时增加元素。当然，同一数组中的元素可以是任意类型。

(2) map

使用 table 也可以定义出类似 map 的 key-value 数据结构。其可以定义 table 时直接指定 key-value，也可单独指定 key-value。而访问时，一般都是通过 table 的 key 直接访问，也可以数组索引方式来访问，此时的 key 即为索引。

(3) 混合结构

Lua 允许将数组与 key-value 混合在同一个 table 中进行定义。key-value 不会占用数组的数字索引值。

(4) table 操作函数

Lua 中提供了对 table 进行操作的函数。

A、table.concat()

【函数】table.concat (table [, sep [, start [, end]]]):

【解析】该函数用于将指定的 table 数组元素进行字符串连接。连接从 start 索引位置到 end

索引位置的所有数组元素，元素间使用指定的分隔符 `sep` 隔开。如果 `table` 是一个混合结构，那么这个连接与 `key-value` 无关，仅是连接数组元素。

B、`table.unpack()`

【函数】`table.unpack (table [, i [, j]])`

【解析】拆包。该函数返回指定 `table` 的数组中的从第 `i` 个元素到第 `j` 个元素值。`i` 与 `j` 是可选的，默认 `i` 为 1，`j` 为数组的最后一个元素。Lua5.1 不支持该函数。

C、`table.pack()`

【函数】`table.pack (...)`

【解析】打包。该函数的参数是一个可变参，其可将指定的参数打包为一个 `table` 返回。这个返回的 `table` 中具有一个属性 `n`，用于表示该 `table` 包含的元素个数。Lua5.1 不支持该函数。

D、`table.maxn()`

【函数】`table.maxn(table)`

【解析】该函数返回指定 `table` 的数组中的最大索引值，即数组包含元素的个数。

E、`table.insert()`

【函数】`table.insert (table, [pos,] value):`

【解析】该函数用于在指定 `table` 的数组部分指定位置 `pos` 插入值为 `value` 的一个元素。其后的元素会被后移。`pos` 参数可选，默认为数组部分末尾。

F、`table.remove ()`

【函数】`table.remove (table [, pos])`

【解析】该函数用于删除并返回指定 `table` 中数组部分位于 `pos` 位置的元素。其后的元素会被前移。`pos` 参数可选，默认删除数组中的最后一个元素。

G、`table.sort()`

【函数】`table.sort(table [,fun(a,b)])`

【解析】该函数用于对指定的 `table` 的数组元素进行升序排序，也可按照指定函数 `fun(a,b)` 中指定的规则进行排序。`fun(a,b)` 是一个用于比较 `a` 与 `b` 的函数，`a` 与 `b` 分别代表数组中的两个相邻元素。

注意：

- 如果 `arr` 中的元素既有字符串又有数值型，那么对其进行排序会报错。
- 如果数组中多个元素相同，则其相同的多个元素的排序结果不确定，即这些元素的索引

谁排前谁排后，不确定。

- 如果数组元素中包含 nil，则排序会报错。

8.5.2 迭代器

Lua 提供了两个迭代器 `pairs(table)` 与 `ipairs(table)`。这两个迭代器通常会应用于泛型 for 循环中，用于遍历指定的 table。这两个迭代器的不同是：

- `ipairs(table)`：仅会迭代指定 table 中的数组元素。
- `pairs(table)`：会迭代整个 table 元素，无论是数组元素，还是 key-value。

8.5.3 模块

模块是 Lua 中特有的一种数据结构。从 Lua 5.1 开始，Lua 加入了标准的模块管理机制，可以把一些公用的代码放在一个文件里，以 API 接口的形式在其他地方调用，有利于代码的重用和降低代码耦合度。

模块文件主要由 table 组成。在 table 中添加相应的变量、函数，最后文件返回该 table 即可。如果其它文件中需要使用该模块，只需通过 `require` 将该模块导入即可。

（1）定义一个模块

模块是一个 lua 文件，其中会包含一个 table。一般情况下该文件名与该 table 名称相同，但其并不是必须的。

（2）使用模块

这里要用到一个函数 `require("文件路径")`，其中文件名是不能写 .lua 扩展名的。该函数可以将指定的 lua 文件静态导入（合并为一个文件）。不过需要注意的是，该函数的使用可以省略小括号，写为 `require "文件路径"`。

`require()` 函数是有返回值的，返回的就是模块文件最后 `return` 的 table。可以使用一个变量接收该 table 值作为模块的别名，就可以使用别名来访问模块了。

（3）再看模块

模块文件中一般定义的变量与函数都是模块 table 相关内容，但也可以定义其它与 table 无关的内容。这些全局变量与函数就是普通的全局变量与函数，与模块无关，但会随着模块的导入而同时导入。所以在使用时可以直接使用，而无需也不能添加模块名称。

8.5.4 元表与元方法

元表，即 Lua 中普通 table 的元数据表，而元方法则是元表中定义的普通表的默认行为。Lua 中的每个普通 table 都可为其定义一个元表，用于扩展该普通 table 的行为功能。例如，对于 table 与数值相加的行为，Lua 中是没有定义的，但用户可通过为其指定元表来扩展这种行为；再如，用户访问不存在的 table 元素，Lua 默认返回的是 nil，但用户可能并不知道发生了什么。此时可以通过为该 table 指定元表来扩展该行为：给用户提示信息，并返回用户指定的值。

(1) 重要函数

元表中有两个重要函数：

- setmetatable(table,metatable)：将 metatable 指定为普通表 table 的元表。
- getmetatable(table)：获取指定普通表 table 的元表。

(2) __index 元方法

当用户在对 table 进行读取访问时，如果访问的数组索引或 key 不存在，那么系统就会自动调用元表的 __index 元方法。该重写的方法可以是一个函数，也可以是另一个表。如果重写的 __index 元方法是函数，且有返回值，则直接返回；如果没有返回值，则返回 nil。

```
emp = {"北京", nil, name="张三", age=23, "上海", depart="销售部", "广州", "深圳"}
print(emp.x)

-- 声明一个元表
meta = {};

-- 将原始表与元表相关联
setmetatable(emp, meta)

-- 有返回值的情况
--~ meta.__index = function(tab, key)
--~     return "通过【"..key.."】访问的值不存在"
--~ end

-- 无返回值的情况
--meta.__index = function(tab, key)
--    print("通过【"..key.."】访问的值不存在")
--end

print(emp.x)
print(emp[2])
```

```
emp = {"北京", name="张三", age=23, "上海", depart="销售部", "广州", "深圳"}
print(emp[5])

-- 声明一个元表
meta = {};

-- 将原始表与元表相关联
setmetatable(emp, meta)

-- 再定义一个普通表
other = {}

other[5] = "天津"
other[6] = "西安"

-- 指定元表为另一个普通表
meta.__index = other

-- 在原始表中若找不到，则会到元表指定的普通表中查找
print(emp[6])
```

(3) __newindex 元方法

当用户为 table 中一个不存在的索引或 key 赋值时，就会自动调用元表的 __newindex 元方法。该重写的方法可以是一个函数，也可以是另一个表。如果重写的 __newindex 元方法是函数，且有返回值，则直接返回；如果没有返回值，则返回 nil。

```
emp = {"北京", name="张三", age=23, "上海", depart="销售部", "广州", "深圳"}
print(emp[5])

-- 声明一个元表
meta = {};

-- 将原始表与元表相关联
setmetatable(emp, meta)

-- 无返回值的情况
function meta.__newindex(tab, key, value)
    print("新增的key为"..key.."，value为"..value)
    -- 将新增的key-value写入到原始表
    rawset(tab, key, value)
end

emp.x = "天津"
print(emp.x)
```

```
emp = {"北京", name="张三", age=23, "上海", depart="销售部", "广州", "深圳"}
-- 声明一个元表
meta = {};
-- 将原始表与元表相关联
setmetatable(emp, meta)
-- 再定义一个普通表
other = {}
-- 元表指定的另一个普通表的作用是，暂存新增加的数据
meta.__newindex = other
emp.x = "天津"
print(emp.x)
print(other.x)
```

(4) 运算符元方法

如果要为一个表扩展加号(+)、减号(-)、等于(==)、小于(<)等运算功能，则可重写相应的元方法。

例如，如果要为一个 table 扩展加号(+)运算功能，则可重写该 table 元表的__add 元方法，而具体的运算规则，则是定义在该重写的元方法中的。这样，当一个 table 在进行加法(+)运算时，就会自动调用其元表的__add 元方法。

```
emp = {"北京", name="张三", age=23, "上海", depart="销售部", "广州", 12, "深圳"}
-- 声明一个元表
meta = {
  -- 遍历tab中的所有元素
  -- 若value为数值类型，则做算术加法
  -- 若value为string，则做字符串拼接
  __add = function(tab, num)
    for k, v in pairs(tab) do
      if type(v) == "number" then
        tab[k] = v + num
      elseif type(v) == "string" then
        tab[k] = v..num
      end
    end
    -- 返回变化过的table
    return tab
  end
};
-- 将原始表与元表相关联
setmetatable(emp, meta)
empsum = emp + 5
for k, v in pairs(empsum) do
  print(k..": "..v)
end
```

类似于加法操作的其它操作，Lua 中还包含很多：

元方法	说明	元方法	说明
__add	加法， +	__band	按位与， &
__sub	减法， -	__bor	按位或，

__mul	乘法, *	__bxor	按位异或, ~
__div	除法, /	__bnot	按位非, ~
__mod	取模, %	__shl	按位左移, <<
__pow	次幂, ^	__shr	按位右移, >>
__unm	取反, -		
__idiv	取整除法, //	__eq	等于, ==
		__lt	小于, <
__concat	字符串连接, ..	__le	小于等于, <=
__len	字符串长度, #		

(5) __tostring 元方法

直接输出一个 table，其输出的内容为类型与 table 的存放地址。如果想让其输出 table 中的内容，可重写 __tostring 元方法。

```
emp = {"北京", name="张三", age=23, "上海", depart="销售部", "广州", 12, "深圳"}

-- 声明一个元表
meta = {
    __add = function(tab, num)
        -- 遍历tab中的所有元素
        for k, v in pairs(tab) do
            -- 若value为数值类型, 则做算术加法
            if type(v) == "number" then
                tab[k] = v + num
            -- 若value为string, 则做字符串拼接
            elseif type(v) == "string" then
                tab[k] = v..num
            end
        end
        -- 返回变化过的table
        return tab
    end, -- 注意, 这里必须要添加一个逗号

    __tostring = function(tab)
        str = ""
        -- 字符串拼接
        for k, v in pairs(empsum) do
            str = str.." "..k.." ":"..v
        end
        return str
    end
};
```

```
-- 将原始表与元表相关联
setmetatable(emp, meta)

empsum = emp + 5

print(emp)
print(empsum)
```

(6) __call 元方法

当将一个 table 以函数形式来使用时，系统会自动调用重写的__call 元方法。该用法主要是可以简化对 table 的相关操作，将对 table 的操作与函数直接相结合。

```
emp = {"北京", name="张三", age=23, "上海", depart="销售部", 59, "广州", "深圳"}
-- 将原始表与匿名元表相关联
setmetatable(emp, {
  __call = function(tab, num, str)
    -- 遍历table
    for k, v in pairs(tab) do
      if type(v) == "number" then
        tab[k] = v + num
      elseif type(v) == "string" then
        tab[k] = v..str
      end
    end
    return tab
  end
})
newemp = emp(5, "-hello")
for k, v in pairs(newemp) do
  print(k.."":..v)
end
```

(7) 元表单独定义

为了便于管理与复用，可以将元素单独定义为一个文件。该文件中仅可定义一个元表，且一般文件名与元表名称相同。

若一个文件要使用其它文件中定义的元表，只需使用 require “元表文件名”即可将元表导入使用。

如果用户想扩展该元表而又不想修改元表文件，则可在用户自己文件中重写其相应功能的元方法即可。

8.5.5 面向对象

Lua 中没有类的概念，但通过 table、function 与元表可以模拟和构造出具有类这样功能的结构。

(1) 简单对象的创建

Lua 中通过 table 与 function 可以创建一个简单的 Lua 对象：table 为 Lua 对象赋予属性，通过 function 为 Lua 对象赋予行为，即方法。

（2）类的创建

Lua 中使用 `table`、`function` 与元表可以定义出类：使用一个表作为基础类，使用一个 `function` 作为该基础类的 `new()` 方法。在该 `new()` 方法中创建一个空表，再为该空表指定一个元表。该元表重写 `__index` 元方法，且将基础表指定为重写的 `__index` 元方法。由于 `new()` 中的表是空表，所以用户访问的所有 `key` 都会从基础类（表）中查找。

8.5.6 协同线程与协同函数

（1）协同线程

Lua 中有一种特殊的线程，称为 `coroutine`，协同线程，简称协程。其可以在运行时暂停执行，然后转去执行其它线程，然后还可返回再继续执行没有执行完毕的内容。即可以“走走停停，停停再走走”。

协同线程也称为协作多线程，在 Lua 中表示独立的执行线程。任意时刻只会有一个协程执行，而不会出现多个协程同时执行的情况。

协同线程的类型为 `thread`，其启动、暂停、重启等，都需要通过函数来控制。下表是用于控制协同线程的基本方法。

方法	描述
create(function)	创建一个协同线程实例，即返回的是 <code>thread</code> 类型。参数是一个 <code>function</code> 。其需要通过 <code>resume()</code> 来启动协同线程的执行
resume(thread, ...)	启动指定的协同线程的执行，使其从开始处或前面挂起处开始执行。可以向 <code>create()</code> 的内置函数传递相应的参数。如果内置函数具有返回值， <code>resume()</code> 会全部接收并返回。
running()	返回正在运行的协同线程实例，即 <code>thread</code> 类型值
yield()	挂起协同线程，并将协同线程设置为挂起状态。 <code>resume()</code> 可从挂起处重启被挂起的协同线程
status(thread)	查看协同线程的状态。状态有三种：运行态 <code>running</code> ，挂起态 <code>suspended</code> ，消亡态 <code>dead</code>
close()	关闭协同线程
wrap(function)	创建一个协同函数，返回的是 <code>function</code> 类型。一旦调用该函数就会创建并执行一个协同线程实例

(2) 协同函数

协同线程可以单独创建执行，也可以通过协同函数的调用启动执行。使用 `coroutine` 的 `wrap()` 函数创建的就是协同函数，其类型为 `function`。

由于协同函数的本质就是函数，所以协同函数的调用方式就是标准的函数调用方式。只不过，协同函数的调用会启动其内置的协同线程。

```
-- 创建一个协同函数
cf = coroutine.wrap(
    function (a, b)
        print(a, b)

        -- 获取当前协同函数创建的协同线程
        tr = coroutine.running()

        print("tr的类型为: ".. type(tr))

        -- 挂起当前的协同线程
        coroutine.yield(a+1, b+1)

        print("又重新返回到了协同线程")

        return a+b, a*b
    end
)

-- 调用协同函数，启动协同线程
result1, result2 = cf(3, 5)
print(result1, result2)

print("cf的类型为: ".. type(cf))

-- 重启挂起的协同线程
result1, result2 = cf(3, 5)
print(result1, result2)
```

8.5.7 文件 IO

Lua 中提供了大量对文件进行 IO 操作的函数。这些函数分为两类：静态函数与实例函

数。所谓静态函数是指通过 `io.xxx()` 方式对文件进行操作的函数，而实例函数则是通过 Lua 中面向对象方式操作的函数。

(1) 常用静态函数

A、`io.open()`

【格式】`io.open (filename [, mode])`

【解析】以指定模式打开指定文件，返回要打开文件的句柄，就是一个对象（后面会讲 Lua 中的对象）。其中模式 `mode` 有三种，但同时还可配合两个符号使用：

- `r`: 只读，默认模式
- `w`: 只写，写入内容会覆盖文件原有内容
- `a`: 只写，以追加方式写入内容
- `+`: 增加符，在 `r+`、`w+`、`a+` 均变为了读写
- `b`: 二进制表示符。如果要操作的文件为二进制文件，则需要变为 `rb`、`wb`、`ab`。

B、`io.input()`

【格式】`io.input (file)`

【解析】指定要读取的文件。

C、`io.output()`

【格式】`io.output (file)`

【解析】指定要写入的文件。

D、`io.read()`

【格式】`io.read([format])`

【解析】以指定格式读取 `io.input()` 中指定的输入文件。其中 `format` 格式有：

- `*l`: 从当前位置的下一个位置开始读取整个行，默认格式
- `*n`: 读取下一个数字，其将作为浮点数或整数
- `*a`: 从当前位置的下一个位置开始读取整个文件
- `number`: 这是一个数字，表示要读取的字符的个数

E、`io.write()`

【格式】`io.write(data)`

【解析】将指定的数据 `data` 写入到 `io.output()` 中指定的输出文件。

(2) 常用实例函数

A、file:read()

这里的 file 使用的是 io.open() 函数返回的 file，其实际就是 Lua 中的一个对象。其用法与 io.read() 的相同。

B、file:write()

用法与 io.write() 的相同。

C、file:seek()

【格式】file:seek ([whence [, offset]])

【解析】该函数用于获取或设置文件读写指针的当前位置。位置从 1 开始计数，除文件最后一行外，每行都有行结束符，其会占两个字符位置。位置 0 表示文件第一个位置的前面位置。

当 seek() 为无参时会返回读写指针的当前位置。参数 whence 的值有三种，表示将指针定位的不同位置。而 offset 则表示相对于 whence 指定位置的偏移量，offset 的默认值为 0，为正表示向后偏移，为负表示向前偏移。

- set: 表示将指针定位到文件开头处，即 0 位置处
- cur: 表示指针保持当前位置不变，默认值
- end: 表示将指针定位到文件结尾处