**目 录**

[1 CZL简介 1](#_Toc528603725)

[2 CZL外貌 2](#_Toc528603726)

[2.1 函数指针的应用 2](#_Toc528603727)

[2.2 哈希表的应用 3](#_Toc528603728)

[2.3 类的应用 3](#_Toc528603729)

[2.4 协程的应用 4](#_Toc528603730)

[2.5 递归的应用 5](#_Toc528603731)

[2.6 字符串的应用 5](#_Toc528603732)

[2.7 多重循环的应用 6](#_Toc528603733)

[3 数据类型 7](#_Toc528603734)

[3.1 var 7](#_Toc528603735)

[3.2 int 7](#_Toc528603736)

[3.3 float 8](#_Toc528603737)

[3.4 num 8](#_Toc528603738)

[3.5 str 8](#_Toc528603739)

[3.6 fil 8](#_Toc528603740)

[3.7 arr 9](#_Toc528603741)

[3.8 tab 9](#_Toc528603742)

[3.9 sta 9](#_Toc528603743)

[3.10 que 9](#_Toc528603744)

[3.11 lis 10](#_Toc528603745)

[3.12 enum 10](#_Toc528603746)

[3.13 class 10](#_Toc528603747)

[3.14 fun 12](#_Toc528603748)

[3.15 static 13](#_Toc528603749)

[4 运算符 14](#_Toc528603750)

[5 语句 15](#_Toc528603751)

[5.1 for万金油 16](#_Toc528603752)

[5.2 yeild 18](#_Toc528603753)

[5.3 try 20](#_Toc528603754)

[5.4 load 22](#_Toc528603755)

[6 变量 22](#_Toc528603756)

[7 函数 24](#_Toc528603757)

[8 引用 25](#_Toc528603758)

[9 系统库 27](#_Toc528603759)

[10 多线程 29](#_Toc528603760)

[11 编译过程 36](#_Toc528603761)

[12 运行时 37](#_Toc528603762)

[13 内存管理 40](#_Toc528603763)

[14 扩展编写 43](#_Toc528603764)

[15 作为库使用 46](#_Toc528603765)

[16 如何编译CZL 47](#_Toc528603766)

[结束语 48](#_Toc528603767)

# 1 CZL简介

CZL是一款跨平台、高性能、省内存、功能强大和易于使用的解释型编程语言，它的语法风格和C/C++相似度高达90%以上，具有C/C++编程经验的人，几乎不需要学习即可马上使用。它既可以作为独立开发的编程语言，也可以作为扩展应用于C/C++。CZL内核由100%的标准C写成，它只有几百KB，与具体平台相关的API由宏控制，源码编译支持所有的C/C++编译器。

CZL天生支持多线程并能友好地应用于C/C++多线程运行环境。CZL的多线程框架对用户是非常友好的，因为它对用户屏蔽了所有锁相关的操作细节，用户只需要简单地操作几个API就能实现几乎所有的多线程功能。CZL运行时没有涉及到任何全局变量的写操作，所以它能友好地应用于C/C++多线程环境。需要说明的是，CZL多线程是能够利用多核的真多线程，这点与Python不同。

CZL支持用C/C++编写扩展。扩展框架非常简单，熟练掌握后一分钟就能完成一个新库的添加，但需要注意的是，个人库用到的所有的全局变量必须放到指定位置，个人库必须针对具体平台用宏控制，因为新添加的库必须支持跨平台和多线程。

CZL的VM是基于寄存器而不是栈，这点与LUA类似。CZL读写变量是直接寻址，不需要进行哈希，要知道，PHP和LUA等基于寄存器的解释型语言在进行变量寻址时都是需要进行哈希操作的，直接寻址无疑是最快的寻址方式。CZL没有空类型，在进行无效变量（可以理解为脱离生命周期或被GC掉的垃圾变量）读写操作时，PHP和LUA等通过空类型处理，CZL通过锁处理，锁比空类型更符合正常逻辑并且能实现垃圾变量的实时回收。锁的引入对变量计算性能是有一定影响的，尽管如此，CZL还是非常高效的，性能不会比PHP7.0和LUA5.3差。

CZL是完全自动内存管理的，这点与很多流行的脚本语言一样。CZL的内存管理模块采用引用计数、写时复制、内存池和标记算法等构成，整个内存管理框架和算法完全重新设计，与现有的所有编程语言都不同，可以说内存管理模块是CZL的灵魂。虽然CZL采用了引用计数，但不存在循环引用问题，具体细节很复杂，这里就不多说。CZL的内存是实时回收的，这得益于上面提到的锁的设计理念。CZL只有在内存不足的情况下才会进行GC，注意，这里的GC与传统上的垃圾回收不是一个概念，CZL的GC表示对内存的压缩，压缩完毕就能腾出多余的空间进行分配。综上所述，CZL非常适合实时的应用场景。

CZL的数据类型可动态、可静态。动静结合有非常大的优势，比如在大型项目中对函数的形参和返回值声明类型有助于代码的构建、维护和排错，在一些非核心的简单的功能中动态类型能非常灵活和快速地实现功能测试等，另外，静态类型还比较容易实现JIT（CZL暂时还没有引入JIT）。CZL支持的数据类型非常多，你能在PHP和LUA中找到的CZL都有，找不到的CZL也有。需要说明的是，CZL支持的类是完整的类，与C++几乎一模一样，只不过针对性地进行了一些简化，因为C++的类真的是过分复杂了。总之，CZL的数据类型一定会让你感受到用程序实现想法是一件很简单的事情。

CZL支持丰富的语句和运算符。首先包括了所有C/C++语句，并且针对for进行了扩展升级，把for做成了易用性非常高的万金油。此外，CZL还有yeild语句，yeild能很好地实现协程，CZL就是利用yeild作为协程框架基础的，协程部分后面会具体介绍。其次，CZL的运算符非常丰富，表达能力非常强。如果说数据类型是编程语言的柜子，那么语句和运算符就是编程语言的各种工具了，像LUA就很少柜子和工具，所以使用上就没有那么方便。

综上所述，CZL在各个方面都追求做到极致，功能强大并易于使用，小巧且高性能。如果你是编程的入门人员，你觉得C的指针打击了你学编程的积极性，那么CZL一定是很好的过渡；如果你是编程的老友，那么CZL能让你的工作变得更加轻松。下面，让我们进入CZL的世界吧！

# 2 CZL外貌

## 2.1 函数指针的应用

//返回a和b的大者

max(a, b)

{

return a > b ? a : b;

}

//比较a和b谁大

cmp(a, b)

{

return a ?? b;

}

//main函数是第一个被执行的函数

main()

{

f = @max; //变量f保存函数max的地址

echo(f(1, 2)); //结果是 2

f = @cmp; //变量f保存函数cmp的地址

echo(f("foo", "bar")); //结果是 1

}

## 2.2 哈希表的应用

main()

{

fruits = {"apple":3.5}; //初始化一张哈希表

fruits["orange"] = 3; //插入一个键值对

for name,price in fruits //遍历打印哈希表的键值对

print("name:%s, price:%f\n", name, price);

}

## 2.3 类的应用

class fruit

{

str name;

float price;

fruit(str name, float price=3)

{

this.name = name;

this.price = price;

}

};

main()

{

fruit apple("apple", 3.5); //定义并初始化一个apple实例

orange = new fruit("orange"); //new并初始化一个orange实例

echo(apple); //打印apple实例的值

echo(orange); //打印orange实例的值

}

## 2.4 协程的应用

//每调用一次fruit()返回一个水果

fruit()

{

static arr fruits = ["apple", "orange"];

for fruit in fruits

yeild fruit; //返回一个水果并保存下一条语句代码地址

return "no fruit any more";

}

main()

{

fruit = fruit();

echo(fruit);

fruit = fruit();

echo(fruit);

fruit = fruit();

echo(fruit);

}

## 2.5 递归的应用

//求斐波那契数列

int fbnl(int N)

{

return N < 2 ? N : fbnl(N - 1) + fbnl(N - 2);

}

int main()

{

int ret = fbnl(7);

echo(ret);

return 0;

}

## 2.6 字符串的应用

main()

{

s = "foobar";

//把s由小写转为大写

len = sz(s); //系统函数sz求字符串的长度

for (i = 0; i < len; i++)

if (s[i] >= 'a' && s[i] <= 'z')

s[i] -= 32;

echo(s); //结果是 FOOBAR

}

## 2.7 多重循环的应用

main()

{

//打印一张九九表

for i in (1,9)

{

for j in (i,9)

print("%d\*%d=%2d\t", i, j, i\*j);

print("\n");

}

}

上面列举了一些程序例子，方便读者快速了解CZL的样子。怎么样，CZL是不是跟C/C++长得几乎一模一样，2.5甚至可以被C/C++编译器直接编译。所以，如果你有C/C++编程经验，学习CZL的成本几乎是零。

# 3 数据类型

|  |  |
| --- | --- |
| **类型** | **说明** |
| var | 万能数据类型 |
| int | 有符号整形，32/64位与系统有关 |
| float | 64位浮点型 |
| num | 数值类型，int或float |
| str | 字符串 |
| fil | 文件 |
| arr | 数组 |
| tab | 哈希表 |
| sta | 栈 |
| que | 队列 |
| lis | 链表 |
| enum | 枚举 |
| class | 类 |
| fun | 函数指针 |
| static | 静态类型声明，需要与具体类型组合使用 |
| 引用 | 本章暂不介绍，后面会独立说明 |

## 3.1 var

var a = {0:1};

a = 1;

a = "foobar";

## 3.2 int

int a = 1.5; //结果是 1

a = 0B1010; //二进制整形

a = 0XA1F4; //十六进制整形

a = -2e3; //结果是 -2000

## 3.3 float

float a = 1; //结果是 1.0

a = 3.8; //结果是 3.8

a = 2e-1; //结果是 0.2

## 3.4 num

num a = 1; //结果是 1

a = 3.8; //结果是 3.8

## 3.5 str

str s[128] = "foobar";

s = new str[32]"czl";

s = "fruit";

ch = s[0]; //把字符串s的第0个元素保存到ch

## 3.6 fil

main()

{

fil f; //定义一个文件变量

open(f, "datas.txt", "wb+"); //以覆盖方式打开datas.txt文件

write(f, {"apple":3.5, "orange":3.0}); //写入一张水果表

var ret;

read(f, ret); //读取刚刚写入的水果表

echo(ret);

close(f); //关闭文件

}

## 3.7 arr

arr a[10] = [1,2,3]; //定义并初始化一个声明长度的数组

a = new arr[10][1,2,3]; //new并初始化一个声明长度的数组

a = [1,2,3]; //赋值一个数组

ele = a[0]; //把第0个元素保存到ele

## 3.8 tab

//哈希表不能声明长度，因为没有意义

tab t = {"apple":3.5, "orange":3.0}; //定义一个哈希表

a = new tab{"apple":3.5, "orange":3.0}; //new一个哈希表

a = {"apple":3.5, "orange":3.0}; //赋值一个哈希表

ele = a["apple"]; //把键为apple的元素的值保存到ele

a => "apple"; //查询键apple是否在哈希表中

"apple" => a; //删除键为apple的元素

## 3.9 sta

sta s[10] = [1,2]; //定义并初始化一个声明长度的栈

push(s, "foobar"); //向栈s插入一个元素

ele = pop(s); //结果是 "foobar"

ele = pop(s); //结果是 1

ele = pop(s); //结果是 2

ele = pop(s); //结果是 0

## 3.10 que

que q = [1,2]; //定义并初始化一个队列

in(q, "foobar"); //向队列插入一个元素

ele = out(q); //结果是 1

ele = out(q); //结果是 2

ele = out(q); //结果是 "foobar"

ele = out(q); //结果是 0

## 3.11 lis

lis l[10] = [1,2]; //定义并初始化一个声明长度的链表

ins(l, 2, "foobar"); //在位置2后插入一个元素

ele = at(l, 1); //结果是 1

ele = at(l, 2); //结果是 2

ele = at(l, 3); //结果是 "foobar"

del(l, 3); //删除位置3的元素，即删除"foobar"

## 3.12 enum

enum {A, B="foobar", C, D=[1,2], E=1.1, F};

//A等于0， C等于1， F等于2.1

//enum里的每个成员都是只读属性，不能进行修改

## 3.13 class

class b

{

pro: //受保护访问权限

var b;

pub:

var c = 2; //设置默认值为2

};

final class a : pub b //a以公开方式继承b，final表示a不能被继承

{

pri: //私有访问权限

// func关键字专门用于类的内部函数声明

func get, set; //声明函数 get 和 set

static var x; //定义一个共享变量x

pub: //公开访问权限

var a;

var c;

a(a, b) //类a的构造函数

{

this.a = a; //必须用this访问实例成员变量

this.b = b;

}

class c

{

var a = "foobar";

} d;

};

a::get() //定义类a的get函数

{

}

a::set() //定义类a的set函数

{

}

main()

{

a a1(1,2); //定义并初始化一个a的实例a1

a2 = new a("foo", "bar"); //new并初始化一个a的实例a2

a as1[10](1,2); //定义并初始化一个a的实例数组as1

as2 = new a[10]("foo", "bar"); //new并初始化一个a的实例数组a2

echo(a1.a); //结果是 1

echo(a1.d.a); //结果是 "foobar"

echo(a1.c); //结果是 0

echo(a1.b::c); //结果是 2，这里通过指定b来访问b的成员c

echo(a1.b); //抛出异常，因为成员b是受保护的

}

## 3.14 fun

fun1()

{

echo("I am fun1");

}

fun2()

{

echo("I am fun2");

}

main()

{

fun f = @fun1; //@是取函数地址运算符

f(); //打印 I am fun1

f = @fun2;

f(); //打印 I am fun2

}

## 3.15 static

class a

{

static int cnt;

};

int test()

{

static int cnt;

return cnt++;

}

main()

{

echo(test()); //打印 0

echo(test()); //打印 1

a1 = new a;

a2 = new a;

a1.cnt = 10;

echo(a2.cnt); //打印10

}

补充说明：非var声明变量，除了int和float型变量可以互相赋值，其他类型变量不能相互赋值，否则会抛出异常。

# 4 运算符

|  |  |  |
| --- | --- | --- |
| 标识 | 说明 | 目数、结合性 |
| ++ | 自加运算符，只支持数值类型，注意区分i++和++i | 单目 |
| -- | 自减运算符，只支持数值类型，注意区分i--和--i | 单目 |
| - | 取负运算符，只支持数值类型 | 单目 |
| ! | 取非运算符，只支持数值类型 | 单目 |
| ~ | 按位取反运算符，只支持数值类型 | 单目 |
| & | 取变量地址运算符，必须和变量名使用 | 单目 |
| @ | 取函数地址运算符，必须和静态函数名使用 | 单目 |
| || | 条件或运算符 | 双目，自左向右 |
| && | 条件与运算符 | 双目，自左向右 |
| >< | 值交换运算符，只支持数值和字符串类型 | 双目，自右向左 |
| = | 赋值运算符，支持任意类型 | 双目，自右向左 |
| += | 复合赋值加运算符，支持数值、字符串和数组类型 | 双目，自右向左 |
| -= | 复合赋值减运算符，只支持数值类型 | 双目，自右向左 |
| \*= | 复合赋值乘运算符，只支持数值类型 | 双目，自右向左 |
| /= | 复合赋值除运算符，只支持数值类型 | 双目，自右向左 |
| %= | 复合赋值求余运算符，只支持整形 | 双目，自右向左 |
| |= | 复合赋值按位或运算符，只支持整形 | 双目，自右向左 |
| ^= | 复合赋值按位亦或运算符，只支持整形 | 双目，自右向左 |
| &= | 复合赋值按位与运算符，只支持整形 | 双目，自右向左 |
| <<= | 复合赋值按位左移运算符，只支持整形 | 双目，自右向左 |
| >>= | 复合赋值按位右移运算符，只支持整形 | 双目，自右向左 |
| > | 大于运算符，支持数值和字符串类型 | 双目，自左向右 |
| >= | 大于或等于运算符，支持数值和字符串类型 | 双目，自左向右 |
| < | 小于运算符，支持数值和字符串类型 | 双目，自左向右 |
| <= | 大于或等于运算符，支持数值和字符串类型 | 双目，自左向右 |
| == | 等于运算符，支持任意类型 | 双目，自左向右 |
| != | 不等于运算符，支持任意类型 | 双目，自左向右 |
| === | 恒等于运算符，支持任意类型 | 双目，自左向右 |
| ?? | 大小比较运算符，支持数值和字符串类型 | 双目，自左向右 |
| => | 哈希表运算符，只能用于哈希表操作 | 双目，自左向右 |
| + | 加运算符，支持数值、字符串和数组类型 | 双目，自左向右 |
| - | 减运算符，只支持数值类型 | 双目，自左向右 |
| \* | 乘运算符，只支持数值类型 | 双目，自左向右 |
| / | 除运算符，只支持数值类型 | 双目，自左向右 |
| % | 求余运算符，只支持数值类型 | 双目，自左向右 |
| | | 按位或运算符，只支持整形 | 双目，自左向右 |
| ^ | 按位亦或运算符，只支持整形 | 双目，自左向右 |
| & | 按位与运算符，只支持整形 | 双目，自左向右 |
| << | 按位左移运算符，只支持整形 | 双目，自左向右 |
| >> | 按位右移运算符，只支持整形 | 双目，自左向右 |
| ?: | 条件运算符 | 双目，自左向右 |

补充说明：

1. 双目运算符的结果类型与左操作数保持一致，这样做能在表达式计算过程中更方便开发者确定变量类型而不需要复杂的推导过程，C/C++的类型转换过分复杂了，这里进行了简化；

2. 各运算符优先级和C/C++保持一致。

# 5 语句

|  |  |
| --- | --- |
| **标识** | 说明 |
| if-elif-else | 分支语句，其中elif等效else if |
| switch-case-default | 分支语句，其中case后可以是任意数据类型或表达式 |
| while/do-while | 循环语句 |
| for | 循环语句，其中for进行了万金油设计 |
| break | 跳出上层循环 |
| continue | 跳到下一次循环 |
| goto | 跳到指定标识位置 |
| return | 函数返回 |
| yield | 协程断点返回 |
| try | 异常处理 |
| load | 脚本文件载入 |

补充说明：if/switch/for/break/continue/goto/return语句与C/C++保持一致，下面主要对for的万金油、yield、try和load进行详细介绍。

## 5.1 for万金油

for i in (1, 10, 2) //i从1开始，每循环一次累加2，直到大于10就退出

echo(i);

for i in (1, 10) //i从1开始，每循环一次累加1，直到大于10就退出

echo(i);

for i in "foobar" //遍历一个字符串

print("%c\n", i);

for i in [1,2,3] //遍历一个数组

echo(i);

fil f;

open(f, "datas.txt", "wb+", 2);

write(f, "apple", 3.5);

write(f, "orange", 3.0);

close(f);

open(f, "datas.txt", "rb+", 2);

for i in f //每循环一次读取一行文件数据

echo(i);

a = [1,2];

for i in a //遍历数组

echo(i);

for i in obj //obj可以是字符串、数组、哈希表、栈、队列、链表和文件

echo(i);

for i in &obj //&表示获取obj元素的引用，当obj是字符串或文件是无效

i = 0;

t = {0:"foo", 1:"bar"};

for k,v in t //k是哈希表元素的键，v是哈希表元素的值

echo(k, " ", v);

t = {0:"foo", 1:"bar"};

for k key t //key声明获取的是哈希表元素的键

echo(k);

补充说明：for循环获取对象引用时，如果没有自然循环结束而是中途跳转出循环，那么for i in &obj的i是一个引用变量，如果是自然循环结束，那么i是类型为int、数值等于0的普通变量。

## 5.2 yeild

yeild和return很像，区别是return直接从函数返回并释放所有临时变量内存，而yeild保存临时变量内存和yeild的下一条语句地址、当再次调用时会从该地址开始运行、直到函数返回就释放临时变量和重置语句地址为函数的第一条。

包含有yeild的函数就表示该函数是一个协程。协程与函数的区别有两点：协程的新参列表不能加默认参数，函数可以；协程不支持递归，函数支持。协程有两种工作方式，一种是直接使用本体运行，另一种是通过crecor函数来fork多个实例、然后通过返回的id来指定运行哪个协程实例。

每次调用协程的参数不是固定的，只要少于或等于形参个数就行，声明了参数的部分会被覆盖掉、没声明的保持上一次的值。这个特性保证协程是能够通信的，结合yeild能实现协程返回值就保证了协程的完备特性。

本体方式运行：

cor(v)

{

for i in (1, 10)

{

echo("v: ", v);

yeild i;

}

}

main()

{

echo(cor());

echo(cor(1));

echo(cor());

}

实例方式运行：

cor(v)

{

for i in (1, 10)

{

echo("v: ", v);

yeild i;

}

}

main()

{

c1 = newcor(@cor);

echo(resume(c1));

echo(resume(c1, 1));

c2 = newcor(@cor);

echo(resume(c2));

echo(resume(c2, 2));

corsta(c1); //查看c1协程是否运行结束

reset(c1); //重置协程c1

kill(c1); //删除协程c1

}

## 5.3 try

try专门用于异常处理。CZL没有catch，但是能捕获所有异常。try的使用格式如下：

try(continue/break/exit,异常处理代码) {异常触发代码}

continue: 异常处理完毕继续执行异常触发代码的下一条语句；

break: 异常处理完毕跳出“{异常触发代码}”继续执行try的下一条语句；

exit: 异常处理完毕直接退出程序。

异常处理常用的三个函数如下：

czl\_sys\_errfile(): 获取异常触发的脚本文件；

czl\_sys\_errline(): 获取异常触发的脚本行号；

czl\_sys\_errcode(): 获取异常触发的异常类型；

使用上面三个函数能快速定位出错位置和出错原因，非常方便。同时CZL在作为扩展使用时还提供了log打印到文件的功能，也能非常快的定位问题，后面会独立介绍log模块。

try-continue例子:

try(continue, echo("error code: ", errcode()))

{

var a;

a/0; //除0异常

echo("continue run"); //会执行

}

echo("see me"); //会执行

try-break例子:

try(break, echo("error code: ", errcode()))

{

var a;

a/0; //除0异常

echo("continue run"); //不会执行

}

echo("see me"); //会执行

try-exit例子:

callback()

{

echo("error code: ", errcode());

}

try(exit,callback())

{

var a;

a/0; //除0异常

echo("continue run"); //不会执行

}

echo("see me"); //不会执行

补充说明：try语句和分支、循环语句一样，是可以嵌套使用的。

## 5.4 load

load和C/C++的include差不多，都是用来把工程分解为多个文件，有了load，就可以构建大型项目进行分工合作了。load和include的区别是：load对应的脚本文件只能包含一次，而include可以在不同源文件（即.c/.cpp文件）多次包含同一个头文件（即.h文件），之所以load只能包含同一个脚本文件一次，是因为CZL没有源文件和头文件之分，所有脚本文件共享同一个全局作用域，函数的调用也不需要先声明才能调用（类成员函数除外），每个脚本被load后都会被编译生成opcode，这样在后续的任何文件都可以识别这些opcode了，所以没必要重复load。由于CZL的所有脚本文件共享同一个全局作用域，所以不能在两个脚本定义同名的全局变量、函数或类（C/C++是可以的），当不同的人负责不同脚本文件时，全局对象名最好以负责人名字的缩写作为前缀，这样在各文件调用时，就能直观的看到当前调用对象是谁负责的，出错后找负责人就非常快了。C/C++通过在头文件以extern方式共享对象在文件依赖比较复杂时存在各种各样的编译问题，这对项目的划分就造成非常大的困扰，C/C++老友应该深有体会。

补充说明：CZL的变量、函数和类都是可以同名的。

举例：

load "class.czl" //遇到load，就会跳转到class.czl文件进行解析编译，

//完毕后回到当前文件继续解析。

load "thread.czl"

main()

{

}

# 6 变量

CZL的变量分为局部变量、全局变量、类成员变量、函数内部静态变量、类内部静态变量，CZL没有全局静态变量（因为所有脚本共享同一个全局作用域），下面具体介绍各个作用域变量的定义方式和调用方法。

举例1：

class a

{

static float price = 6.666; //定义一个类内部静态变量

var any; //定义一个类内部全局变量

a()

{

this.any = 0; //类内部变量调用必须加this作为声明

}

};

float price = 6.666; //定义一个全局变量

main()

{

static float price; //定义一个函数内部静态变量

var any; //定义一个函数内部局部变量

echo(price); //打印局部变量

echo($price); //打印全局变量，全局变量的调用必须加$作为声明

}

举例2：

main()

{

tmp = 0; //函数局部变量可以通过赋值方式定义变量，全局作用域不行

if (1)

{

var tmp = 1; //函数局部变量支持块作用域语句内部定义

echo(tmp);

}

echo(tmp);

}

全局变量以$方式调用、类成员变量以this方式调用的好处是：查看一个变量的调用出处非常直观。毕竟一个项目的后期修改和维护时间远远比开发时多，其实CZL的所有语法设计都是基于简洁性、可读性和可维护性考量的。

# 7 函数

CZL的函数支持参数的默认值设置（协程函数除外）和指定参数类型与返回值类型，但是不支持重载，也不支持闭包。不支持重载的原因是：CZL是动态类型语言。不支持闭包的原因是：闭包的引入会导致函数嵌套层次过深，可读性差，并且闭包的作用无非是增加变量的局部作用域而已，并不是非有不可，引入闭包并不能添加任何新功能。

举例：

int test(int a=0, b="foobar")

{

echo(a);

echo(b);

echo("===========");

}

main()

{

test(); //都使用默认值

test(1); //后一个参数使用默认值

test(,"fruit"); //前一个参数使用默认值

}

# 8 引用

任何一门实用的编程语言都应该包含引用。C/C++的指针就是面向操作系统内存管理单元的引用，而脚本语言的引用则是面向自己设计的内存管理模块的对象。任何软件的性能在相同机器上都取决于内存管理模块的设计方案，而引用能很好地减少内存管理的开销，主要是内存搬移和访问速度。由此可见，引用是多麽的重要。

C高手一定是指针使用高手、内存管理高手。许多C程序员新手对指针望而却步，因为指针管理不好就会出现段错误（内存脏读、脏写）导致程序崩溃，这种情况是最头疼的，谁都不想单步调试找内存Bug，当内存Bug出现在一个超大循环时单步调试根本没法用，等到猴年马月也调不完，所以指针真的很头痛，并且你一点办法也没有。CZL的引用就不一样了，它不可能出现段错误，因为内存是后台自动管理的，用户没有办法直接操作系统内存，也不可能产生野指针，因为CZL的引用也是后台自动管理的，一旦某个被引用的变量脱离了生命周期，那么对应的引用变量会自动变成普通变量并且值等于该变量的值、类型也等于该变量的类型。

综上所述，使用CZL的用户，无需考虑内存申请和释放问题，也无需考虑野指针问题，你只需要根据目标问题选用合适的数据类型就能解决所有的应用问题。如果你想操作实际的系统内存，那么可以通过编写扩展来实现，扩展部分后面会详细介绍。下面通过举例说明引用的使用方法。

举例1：

main()

{

a = 0; //定义一个普通变量

pa = &a; //定义一个引用变量并指向a

pa = 1; //把1赋值到pa对应的实体变量a

echo(pa); //打印pa对应的实体变量a的值，结果是1

echo(a); //打印a的值，结果是1

pa = null; //断开pa与a的映射关系，pa变回普通变量并且值为0

pa = 2; //把2赋值到pa本身

echo(pa); //打印pa的值，结果是2

echo(a); //打印a的值，结果是1

}

举例2：

main()

{

//引用变量可以指向任意类型变量

int a;

str b;

p = &a; //p指向一个int型的变量

p = 1;

p = &b; //p指向一个str型的变量

p = "foo";

echo(a);

echo(b);

}

举例3：

main()

{

a = 0;

p1 = &a;

p2 = &p1; //p2指向的不是p1，而是a

p2 = 2;

echo(a);

echo(p1);

echo(p2);

}

举例4：

test(int &pa)

{

pa = 1;

pa = null;

}

main()

{

int a = 0;

test(a);

echo(a); //打印1

}

补充说明：CZL对象成员不能是引用类型，CZL函数参数的传递方式默认是值传递，与数据类型无关，这点与Python不同，Python在对象传递时是引用传递，因为Python没有&运算符。

# 9 系统库

系统库就是系统内置函数的集合，如：echo、print等。CZL的系统库由作者本身设计的os库和第三方编写的扩展库组成，每个库都包含一组函数集合，不同的库函数是可以同名的，第三方扩展库都有唯一的库名。

系统函数的调用方法是：库名.函数名(…)，当库是os库时，库名可省略。CZL没有类似Python的import-from-as，非os库函数必须指定库名来防止命名冲突。当用户自定义对象和系统库或系统库函数重名时，按照用户自定义对象算。之所以没有引入import，是因为当工程较大时，维护者B查看代码无法直接识别开发者A写的同名系统函数，这会造成不少困扰，维护者每查看一个函数调用都要寻找其出处，非常不利于后期维护和升级。而强制声明库名就能直观地识别当前函数是系统函数还是用户自定义函数、系统函数来自哪个库，这能大大提高代码的可阅读性和可维护性，还是那句话“研发的时间远远小于后期维护和升级的时间”。

举例1：

echo()

{

os.echo("I am not system function");

}

main()

{

echo(); //调用用户自定义函数echo

os.echo("I am system function"); //指定调用os库的系统函数echo

}

举例2：

class a

{

echo(v)

{

os.echo(v);

}

};

main()

{

a os; //定义一个类a的实例os

os.echo("I am class function"); //指定调用实例os的成员函数echo

}

不建议开发者把用户自定对象设计成和系统库或系统库函数同名，作者不禁止用户自定义对象和系统对象同名的原因是：系统库和函数是不断增加的，如果后期加入的系统库或函数与用户自定对象同名，那么之前的所有代码都不能编译通过，这样会直接丧失掉旧版本的兼容性。综上所述，CZL在命名冲突上做了最精细的考量，灵活且不失兼容性。

os库是作者单独开发维护的，其中有很多实用的函数，大部分与系统本身有关，还有一些数学计算函数和字符串处理函数，但非常有限。目前负责CZL开发和维护的人员暂时只有作者一个，一个人的精力是很有限的，所以它缺少各种应用库，如：网络、数据库和正则表达式等，由于这是一款新的编程语言，还没有社区，欢迎广大网友为它编写扩展，大家一起组建属于CZL的社区，关于扩展的编写方法，后面会详细介绍。

# 10 多线程

CZL的多线程使用方法以系统库函数的方式提供，CZL多线程能真正利用多核。每个线程运行一个独立的CZL虚拟机，每个线程有独立的内存空间，任何一个线程的异常退出都不会影响其他线程，线程间通信通过信号和共享内存实现，线程间同步通过临界区和互斥量实现，任意一对主、子线程对都有唯一的双向通信加锁管道。综上所述，CZL线程相当于进程，但它比操作系统提供的进程要轻量得多、通信效率高得多，因为CZL进程是利用操作系统的线程模拟的。其实，CZL的整体设计就是一个高阶的操作系统，它屏蔽了多线程复杂的锁机制和容易出现段错误的手动内存管理，C/C++程序员大部分时间都花在了这两点上，从而导致产品开发周期过长、Bug多等问题，而CZL用户只需要关心业务逻辑应用就好了，对于计算密集型部分代码可以通过用C/C++编写扩展实现。

CZL多线程框架是基于主子线程多叉树实现的，每棵子树的根节点都是主线程，叶节点都是子线程，只有主子线程可以直接使用API通信，子线程和子线程不能直接通信。为什么设计成这样呢？现实生活中基于树型结构的逻辑关系非常多，如一个公司的内部人员组织关系，以老板为根节点，老板就是主线程，那么各个部门的部长就是子线程；以部长为根节点，部长就是主线程，而各个科长就是子线程；以科长为根节点，科长就是主线程，那么该科的各个经理就是子线程；以经理为根节点，经理就是主线程，那么各个普通职工就是子线程。正常情况下，职工向老板汇报信息的方式是：职工向经理、经理向科长、科长向部长、部长向老板汇报。再比如QT框架，各个数据子线程向界面主线程发送数据，主线程再将数据展现在界面上，期间子线程向主线程的槽发送信号。类似的树形关系还有非常多，所以，CZL也采用树形结构实现多线程框架。

CZL线程框架图：

多线程API：

newshl(str,var): 创建线程；

wait(int): 阻塞方式等待数据，可用于主线程等待子线程或子线程等待主线程；

listen(int): 非阻塞方式等待子线程数据；

waitshl(int): 阻塞方式等待子线程结束；

report(var): 阻塞方式上报数据给主线程；

send(var): 非阻塞方式上报数据给主线程；

notify(int,var): 非阻塞方式通知子线程数据；

notifyall(var): 非阻塞方式通知所有子线程数据；

shlsta(int): 查看子线程状态。

通过以上9个API就能实现同步或异步方式多线程编程，多线程通信数据类型var必须是int、float、str或arr（arr的元素类型必须是int、float或str，否则全部视为int-0），数据接收函数的返回值固定是数组类型，下面通过具体例子说明如何使用这些API。

例子1：

说明：主线程循环发送数据给子线程、子线程循环接收数据返回给主线程。

主线程脚本：

main()

{

t = newshl("thread.txt", 1); //创建子线程t，入参是1

a = []; //定义一个空数组用于接收子线程返回的数据

for i in (1, 100) //循环发送1到100个数字给子线程t

{

notify(t, i); //发送数据i给子线程t

ret = listen(t); //监听子线程t是否有数据返回

if (ret) //如果有数据返回

a += ret; //把数组ret连接到数组a中

}

notify(t, -1); //发送-1给子线程t表示数据发送完毕

ret = waitshl(t); //等待子线程t结束

if (ret) //如果子线程结束有数据返回

a += ret; //把数组ret连接到数组a中

echo(a); //打印数组a

}

子线程脚本：

main()

{

t = newshl("thread.txt", 1); //创建子线程t，入参是1

a = []; //定义一个空数组用于接收子线程返回的数据

for i in (1, 100) //循环发送1到100个数字给子线程t

{

notify(t, i); //发送数据i给子线程t

ret = listen(t); //监听子线程t是否有数据返回

if (ret) //如果有数据返回

a += ret; //把数组ret连接到数组a中

}

notify(t, -1); //发送-1给子线程t表示数据发送完毕

ret = waitshl(t); //等待子线程t结束

if (ret) //如果子线程结束有数据返回

a += ret; //把数组ret连接到数组a中

echo(a); //打印数组a

}

例子2：

说明：开4个线程计算0到100000000的累加求和，每个子线程计算1/4，最后主线程累加4个子线程的计算结果。该例子在作者的4核处理器上CPU使用率达到100%，用时比单线程少一倍以上。

main()

{

//创建4个子线程并划分好计算任务

t1 = newshl("thread.txt", [0, 25000000]);

t2 = newshl("thread.txt", [25000000, 50000000]);

t3 = newshl("thread.txt", [50000000, 75000000]);

t4 = newshl("thread.txt", [75000000, 100000000]);

sum = 0;

//阻塞方式接收结果

ret = wait(t1);

print("%d\n", ret[0]);

sum += ret[0];

ret = wait(t2);

print("%d\n", ret[0]);

sum += ret[0];

ret = wait(t3);

print("%d\n", ret[0]);

sum += ret[0];

ret = wait(t4);

print("%d\n", ret[0]);

sum += ret[0];

/\*

//非阻塞方式接收结果

cnt = 0;

while (cnt < 4)

{

if ((ret=listen(t1)))

{

print("%d\n", ret[0]);

sum += ret[0];

cnt++;

}

if ((ret=listen(t2)))

{

print("%d\n", ret[0]);

sum += ret[0];

cnt++;

}

if ((ret=listen(t3)))

{

print("%d\n", ret[0]);

sum += ret[0];

cnt++;

}

if ((ret=listen(t4)))

{

print("%d\n", ret[0]);

sum += ret[0];

cnt++;

}

sleep(10);

}

\*/

print("sum: %d\n", sum);

print("time: %dms\n", clock());

}

子线程脚本：

sum(a, b)

{

sum = 0;

for i in (a, b)

{

sum += i;

}

return sum;

}

main(v)

{

sum = sum(v[0], v[1]);

report([sum]);

}

以上两个例子表明：CZL多线程的使用简单方便，短短几句代码就能实现丰富的多线程功能，畏惧多线程锁机制的读者可以在CZL这里找到非常舒适的体验。

# 11 编译过程

编译就是将人写的代码翻译为机器或虚拟机能够识别的代码，期间涉及到词法分析、语法分析、语义分析、AST（抽象语法树）生成、表达式AST转波兰式、表达式波兰式转表达式寄存器序列，最后把整个AST和所有表达式寄存器序列映射到一块连续的内存就能的到opcode，opcode就是最终能被机器或虚拟机直接执行的代码。上面的编译方式是基于寄存器模型的，基于栈模型没有生成寄存器序列这步，实测表明，基于寄存器模型比基于栈模型运行时性能高两倍以上，像PHP7和LUA5就是基于寄存器模型的，所以他俩在解释型编程语言里面是最快的，所以，我建议其他语言的作者也将模型设计为寄存器方式。

寄存器模型就是操作数是可以被运算符直接计算的，栈模型操作数必须先入栈才能进行计算；寄存器模型的操作数是可以被随机访问的，栈模型操作数不能随机访问。从上面两点对比可知，寄存器模型对于相同一段代码生成的指令数更少，因为栈模型包含了太多数据出入栈指令，因此寄存器模型比栈模型更高效。其实，寄存器模型只需要在编译期间通过栈模型方式计算出所有用户变量和临时变量的地址并将指令和这些变量地址封装成寄存器序列即可，只需要多做这一步就能在运行时大大提高效率，何乐不为呢。

CZL的编译过程是高效的，任何一个脚本只需要扫描一遍就能生成opcode，各个部分都是分开编译组织的，比如任意一个函数的所有语句会被编译生成独立的一份opcode，全局初始化语句会被编译生成一份opcode，所有的类会被编译生成一份opcode，所有的枚举会被编译生成一份opcode，……等等，分类编译组织条理清晰，这种编译方法不仅快，还非常利于编译过程的代码阅读和维护，如果后期加入一个新的编译对象，那么它将不会影响原有的任何代码逻辑。

编译原理并不难，只要熟练掌握递归、状态机和常用的数据结构，就能编译任意复杂的文法，觉得编译原理难的根本原因就是基础不扎实，所以希望学习计算机的同学要打牢基础知识，不要跳过基础直接学习应用开发，得不偿失。编程语言开发、甚至可以说任何基础软件的开发，真正核心的内容是运行时和内存管理，系统的稳定性和健壮性完全取决于他俩，比如一台长期运行的服务器不能有内存泄露、内存不足时能进行内存碎片整理并重新利用，否则这台服务器就会频繁发生重启；比如对实时性要求高的游戏不能经常发生GC、无用内存应该实时回收，否则就会导致频繁卡顿现象；比如对内存使用率要求高的应用场合，多出来一些内存就能容纳更多的业务请求。运行时和内存管理不像编译原理，他们没有固定的套路，给你一块连续的内存，就看你怎么用了，就像给你一百块钱，就看你怎么用了是一个道理。下面两章将详细介绍CZL的运行时和内存管理，他们是CZL的灵魂。

# 12 运行时

CZL运行时主要做两件事情，分别是异常检测和内存管理。CZL运行时的所有异常都是可捕获的，CZL虚拟机不会放生异常崩溃现象，如果发生崩溃说明CZL自身存在Bug。对于动态类型并且是自动内存管理的系统，由于不像静态类型的编译型系统那样能在运行前排查绝大部分错误，所以CZL运行时触发异常的途径会非常多，比如内存读写错误、类型不匹配、非法访问和栈溢出等，所以运行时的异常捕获和处理功能就显得非常必要。CZL运行时还涉及到临时内存的分配和回收问题，比如数组连接操作，如果没有指定连接结果的保存变量系统会将结果保存在用户不可见的临时变量中，那么这个临时变量的内存什么时候释放是个问题，由于CZL是基于寄存器的VM，在编译期间就已经确定每个函数需要的临时变量个数，所以任何临时变量对于系统来说都是可控的，CZL临时变量的内存释放时机是下一次用到该临时变量，如果检测到该变量是对象，那么立即释放内存，并且当函数调用结束时系统会扫描该函数的所有临时变量并释放内存，这种时机机制即保证不会产生内存泄露，也能及时地释放内存，由于临时内存的释放不需要定时检查，所以CZL的临时内存管理机制是高效的。再比如函数调用，如果返回值是对象类型并且没有指定返回值的保存变量，那这个对象就会一直存在该函数的返回值临时变量中，直到下一次调用该函数，才能释放这个对象的内存。由于CZL支持的运算符和数据类型多，所以类似的例子还有很多，其中还涉及到临时变量的结果搬移与合并等问题，这里就不多做介绍了，但设计目标是确定的，内存的操作成本一定要低，比如内存搬移次数少、内存的分配和回收速度快等。下面通过具体例子说明CZL运行时的一些行为特点。

例子1：

main()

{

int a;

str b;

a = b; //类型不匹配异常

}

例子2：

rec()

{

rec(); //无限制递归，直到吃完内存异常退出，但是不会出现程序崩溃现象

}

main()

{

rec();

}

例子3：

main()

{

a = "foo";

b = "bar";

echo(a+b); //打印字符串a+b的结果，其中结果保存在系统的临时变量中

a = 1;

b = 2;

echo(a+b); //临时变量在这里再次被使用，这个时候就会释放字符串连接结果

}

例子4：

class a

{

int v = 1;

};

test()

{

return new a;

}

main()

{

echo(test());

a = test(); //上一次test()的临时返回值对象在test再次调用时被释放

}

上面列举了几个能够体现CZL运行时的行为特点的例子，方便读者对CZL的运行时有个大概了解，其实CZL的内部运行机制远不止这些，所以评测一门语言的好坏不能只看表象的东西，同样一个功能可能实现上完全不一样，那么带来的性能和稳定性也不一样。下面介绍CZL运行时的锁机制。

动态类型语言在进行对象的某些操作时，需要加锁保护，先看下面两个例子：

例子1：

main()

{

a = [0,1];

b = a[a=0]; //把数组a的0元素赋值给b，但期间a被释放了，就会发生异常

}

例子2：

main()

{

a = [0,1];

a = &a[0]; //取数组a的0元素引用赋值给a，如果允许赋值成功那么a[0]已经//不存在，所以应该抛出异常

}

上面两个例子表明，在进行某些对象操作时，必须加锁，否则无法发现异常从而导致程序异常崩溃，如果操作成功还需要及时解锁，否则对象内存无法释放，这就是CZL的锁机制。锁对使用者是不可见的，无需关心，用户只需要在发生异常时通过errcode()函数查看异常码就行了。细心的读者可能注意到，作者在介绍CZL运行时过程中，一直是围绕内存讲解的，没错，因为内存真的很重要，一个优秀的系统软件，内存模型一定设计得非常精妙，比如NGX的内存池和Redis的键值对，CZL也不例外，同样对内存模型的设计追求极致，下一章将详细介绍CZL的内存管理机制。

# 13 内存管理

几乎所有的高级语言都用C语言实现，因为C语言具有很高的性能并且易于使用，CZL也是由C语言编写成，CZL所有的结构体设计，都充分利用了C语言特性，同时还需要充分利用C编译器的优化行为来写适合优化的C代码，另外，选用合适的算法和数据结构往往能得到意想不到的效果，最后，还需要对CPU的cache和操作系统的内存管理有一定了解，只有这样才能让CZL性能达到最高。C语言学习不难，但要用好却很难，因为它涉及到很多系统和硬件的相关知识，只有当你把C语言和系统、硬件融为一体的时候，才真正把C语言掌握了。比如你看到某条C语句，你能联想到它是如何被编译的，编译后它是如何与系统交互的，它用到CPU里的哪个部件。比如为什么说C语言的移位运算高效，而乘法运算相对低效得多，因为移位运算在CPU里是一个移位寄存器，而乘法运算是乘法器，相同位数下移位寄存器比乘法器需要的时钟周期少得多。所以，专门学习C语言的人员要做好打持久战的心里准备，待学成归来时，你会发现任何软件在你看来都是透明的。

CZL的内存管理以内存池作为基本单位，比如编译时用到的临时内存由临时内存池分配，字符串由字符串内存池分配，哈希表由哈希表内存池分配，……等等，分类内存池的优点明显，不同对象的内存申请和释放互不影响，它能有效地减少内存碎片，申请和释放内存的速度比直接malloc快得多，并且在删除某些对象时能以内存池为单位一次性释放，速度比一个一个对象成员释放快很多。CZL由内存池实现了内部的malloc、calloc、realloc和free函数，这些函数不会直接操作系统内存，而是操作内存池，当内存池不够时，才会向系统申请新的内存用来构建内存池。CZL的内存池分为定长分配和不定长分配两种，其中又分为128字节以内、256字节以内和512字节以内三种，内存池大小分别对应4KB、20KB和40KB，一次申请内存超过512字节则直接向系统申请，CZL只自动管理512字节以内的原因有两个：原因一是在实际生活当中，小内存往往被高频使用，并且小内存才是操作系统堆积内存碎片的罪魁祸首，所以只要有效地管理小内存的申请和释放，就能提高整个系统的性能；原因二是超过512字节的内存池会很大，导致内存的利用率低，并且操作系统对于超过512字节的内存已经有很好地管理方案了。CZL内存池采用三色标记算法标记已分配和空闲的heap，该算法具有很高的性能，在碎片整合中表现优异。

一个理想的内存池，应该具备如下特征：1.申请和释放内存的平均时间复杂度为O(1)，2.内存的利用率高，3.具有通用性。1和2往往是一对矛盾体，CZL通过牺牲25%的内存巧妙地解决了两者的冲突，也就是说在最糟糕的情况下，内存池最多有25%的内存是没有使用的，但这保证了在最糟糕情况下能以平均时间复杂度为0（4）找到待分配的内存。另外，CZL在定长内存分配中能保证最多需要O(100)时间就能找到待分配内存，在非定长分配中无法保证搜索次数，但CZL在重用内存碎片过程中总是在保证有碎片的池中找，并且保证在碎片池的上一次位置往下找，而不是盲目地遍历，再结合保证有25%的空闲内存，所以在非定长分配中，CZL内存池具有很好的统计特性。其实在运行时非定长分配的内存池只有字符串内存池，其他对象要么是定长、要么超过512字节，所以CZL内存池在实际测试中工作得很好，对比LUA5.3的字符串连接和哈希表插入操作，CZL胜出，两者消耗的时间竟然相差一个数量级，作者觉得不可思议。CZL内存池在删除操作中保证时间复杂度为0(1)，这个达到理想效果，实现原理是在每个分配的heap前包含该heap地址到该内存池的起始地址的偏移量，释放heap时通过偏移量直接找到它的内存池结构体地址，修改内存池结构体参数即可释放该heap。CZL内存池封装了类似操作系统的内存申请和释放函数集，所以CZL内存池具有通用性。综上所述，CZL内存池除了非定长分配依赖统计性外，其他所有特征都达到了理想效果，由于非定长分配被设计为具有良好的统计特性，所以总体上CZL内存池是非常优秀的。

CZL结构体内存设计非常紧凑，力求能省则省、能重用则重用、能少做内存分配则少做分配，但不能影响通用性，这就需要很强的C语言掌控能力了，所以本章一开始为什么强调学习C语言的重要性，学好C语言能培养精打细算的编程习惯，这是一个优秀程序员必备的专业素质。对比Python2.7的类，new多个实例，CZL需要的内存不到Python的三分之一，作者觉得不可思议。

CZL对象的内存管理采用引用计数和写时复制技术，引用计数意思是对象赋值过程中，只对对象的计数器加一，写时复制意思是当对象的成员发生改变时才分配内存将原对象内容拷贝到新对象中并将原对象的计数器减一。这些技术最早出现在操作系统中并且沿用至今，事实表明这些技术工作得很好，但是引用计数存在循环引用的问题，不做特殊处理会导致存在循环引用的对象无法实时释放内存，幸运的是CZL通过特殊手段消除了循环引用，这样一来，CZL的任意对象都能实时释放了，这个特征是非常重要的，很多对实时性要求高的场景使用CZL就能解决内存堆积导致GC卡顿问题。

最后讨论一种现象，在多次malloc后free掉一部分，被free掉的内存会真正回归操作系统吗？实际表明，很多时候被free掉的内存还驻留在进程中，并没有被操作系统回收，有人会认为是操作系统故意不回收为了方便下次malloc时重用的，其实不是，操作系统无法立即回收这些内存供其他进程使用，因为操作系统为进程分配的内存是以页为单位的，如果释放的内存没有达到连续的一页，那么操作系统就无法实现整页回收，页内被free掉的内存就成了碎片，碎片是无法回收的，只有页才能回收，所以本章开头说了学好C必须了解操作系统。CZL的内存池和操作系统的页一样，释放量必须达到一个池时，才会将整个池的内存free掉，其实几乎所有自动内存管理的编程语言都有这个问题，比如你明明删除了一个数组的一部分成员，但查看内存并没有变，既然问题客观存在并没有根本的解决方法，那么我们只能改善现状。CZL的改善方法是，当内存不足时，会将运行时用到的所有可压缩的内存池进行内存压缩，CZL内存池绝大部分是可压缩的，只有很少特殊的内存池不可压缩，并且不可压缩的内存池往往个数很少，所以无需担心。CZL内存池的平均压缩率达到75%，也就是说压缩完毕后每个内存池的空闲内存平均不会超过25%，举个例子，当CZL的很多内存池的使用率不到50%时，某个对象申请内存不足，这时就会触发内存池压缩程序，压缩完毕后就会多出25%的内存用于新对象的内存分配。CZL还提供了手动压缩内存池的函数gc()，下面举个例子：

main()

{

t = {};

for i in (1, 1000000) //向哈希表插入一百万个元素

t[i] = i;

echo(usemem()); //查看当前内存: 49566315

for i in (1, 1000000, 2) //以间隔为2删除五十万个元素

i => t;

echo(usemem()); //查看当前内存: 49566315，可见内存占用并没有变少

gc(); //压缩内存

echo(usemem()); //查看当前内存: 34461403，压缩后内存占用明显变少

}

综上所述，CZL内存管理设计精良、完备，它贯穿整个运行时，除了基于寄存器模型外，这也是CZL高性能的重要原因。至此，CZL各部分都介绍完毕了，期间都是以文字形式说明，没有附图，也没有具体说明某些特征的具体实现方法，因为本文档主要介绍CZL语言的使用方法，对实现方法以后可能会另外写文档补充说明。接下来两章会介绍CZL的扩展编写方法和作为库的使用方法，这两个实用性高，希望读者熟练掌握。

# 14 扩展编写

CZL本身由C编写而成，所以它支持用C进行扩展，当然也可以用C++扩展。扩展编写有完整的框架，扩展编写人员只需要了解框架的规则和CZL的常用数据类型就行了，需要说明的是，CZL本身是线程安全的，扩展编写人员写的扩展库也应该是线程安全的，如果扩展不安全，就会导致CZL的多线程使用无法得到保证。另外，CZL本身是跨平台的，所以还需要用宏控制扩展支持的操作系统平台。最后，所有全局量命名都要以CZL作为前缀，这样当CZL作为扩展使用时就不会导致命名冲突。

CZL源码包含了一个作者自己开发的第三方扩展库czl\_com.c、czl\_com.h作为参考样例，该库是windows平台的串口通信库，非常小，只有300行代码，下面将通过该扩展详细介绍CZL扩展的编写方法。

czl\_com.h源码截图如下：

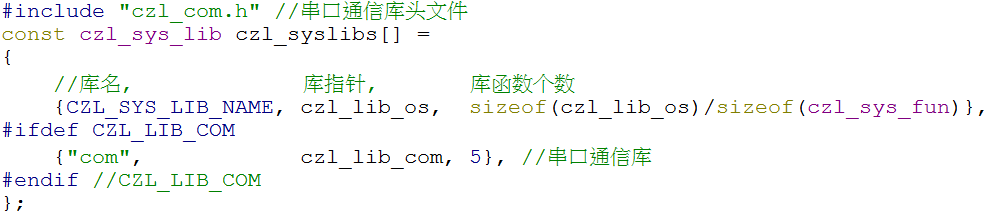
补充说明：CZL\_SYSTEM\_WINDOWS宏是内部宏，表示运行平台是windows平台，CZL\_SYSTEM\_LINUX则表示linux平台，它们在czl\_mm.h中被定义。

czl\_com.c源码截图如下：

补充说明：库函数表定义中，函数名是CZL代码能调用的系统函数名，函数指针

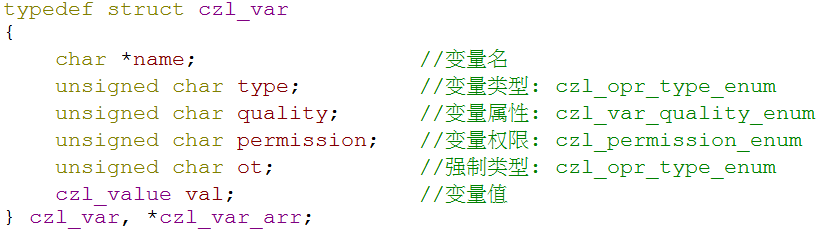
对应库函数声明的内部实现函数，参数个数是该函数的入参个数，参数声明是该函数的入参类型、默认值声明（没有则为NULL，并且字符串不允许有空格、换行等多余符号，目前默认值只允许数值或字符串类型）。

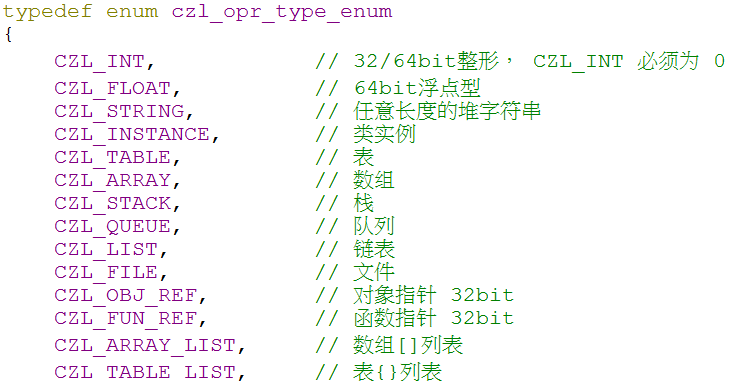
上面两张截图只完成了扩展库的声明和定义，我们还需要将扩展库czl\_lib\_com添加到系统库表中，方法如下：

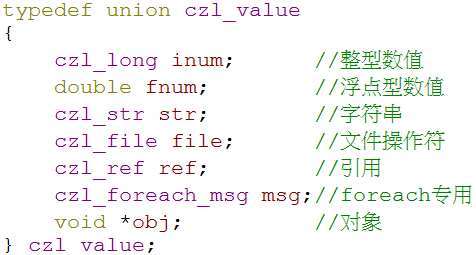
czl\_lib.c源码截图如下：

至此，一个完整的扩展库添加完毕，步骤非常清晰，只需要三步走就能完成一个扩展库的添加，所以，CZL的扩展编写是非常简单的，只需要你有C语言基础，分分钟爽快写扩展。

一个系统函数的内部实现需要遵循一些规则，下面以czl\_com\_close为例进行说明，截图如下：

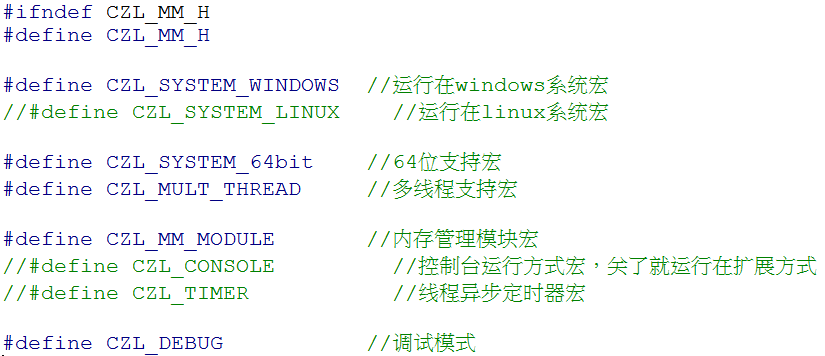
我们只需要搞清楚系统函数的入参怎么获取、返回值怎么设置就行了，对于close函数，它的入参是一个数组类型的引用，我们通过CZL\_GCRV宏获取引用对应的实际变量，其中fun->vars就是入参对应的变量数组，返回值变量是fun->ret，它和fun->vars都是czl\_var类型，下面我们看czl\_var的具体定义，如下图：

我们只需要读写type和val就行，其中type如下图：

val如下图：

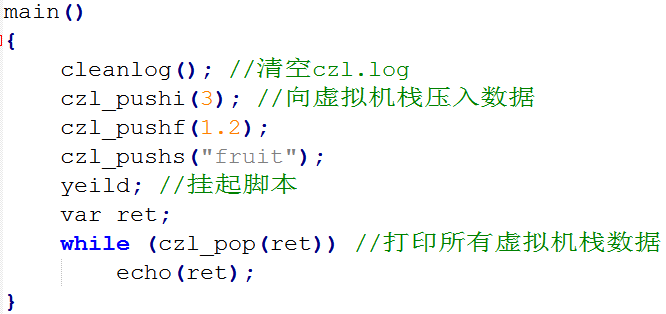
至于具体怎么操作czl\_var、其中用到哪些系统内置函数等问题，建议扩展开发人员通过阅读源码慢慢熟悉，推荐阅读czl\_lib.c中作者自己维护的库czl\_lib\_os，只要你有C语言基础，很容易就能看懂。好了，扩展的编写方法介绍完毕，下一章将介绍如何把CZL作为C/C++的库使用。

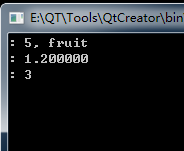
# 15 作为库使用

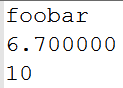
CZL不仅可以独立运行，还可以作为C/C++语言的库使用，我们只需要在czl\_mm.h中把宏CZL\_CONSOLE注释掉，那么CZL就会以服务的形式运行，截图如下：

下面通过具体例子说明CZL作为库时如何使用。因为CZL作为库使用时，没有main函数，所以我们新建一个test.c文件，添加如下代码：

16 编译CZL

新建shell.txt，往里面添加如下代码：

test.c的运行结果如下图：

打开czl.log文件结果如下图：

CZL作为库使用时，通过内部的虚拟机栈与C语言进行数据交互，CZL通过yeild关键字挂起脚本，C语言通过czl\_exec函数唤醒脚本。当C语言运行在多线程环境时，如果同一个gp在多个线程中使用，那么需要对gp加锁，如果多个gp在不同的线程中使用，则不需要加锁，因为CZL虚拟机本身是线程安全的。

# 16 如何编译CZL

C:\Users\Administrator\Desktop\2018-10-29_044247.pngCZL采用标准C编写而成，与操作系统有关的API由宏控制，所以，编译CZL只需要在czl\_mm.h中打开指定操作系统平台宏、然后用对应平台的C/C++编译器编译即可。在windows上编译CZL的czl\_mm.h配置图如下：

C:\Users\Administrator\Desktop\2018-10-29_044259.png在linux上编译CZL的czl\_mm.h配置图如下：

在非windows和linux上编译CZL 的czl\_mm.h配置图如下：

C:\Users\Administrator\Desktop\2018-10-29_044324.png

# 结束语

作者萌生设计编程语言想法是基于无线通信模块测试，无线模块采用AT指令通过串口通信交互，原来的公司用java写测试脚本，很不方便，于是自己就想设计一门哪怕很简单的脚本语言，只要它能灵活地测试AT指令就行，后来一发不可收拾，把语言做得越来越大，最终就变成了CZL。

设计CZL的整个过程中，作者没有查看任何开源的编程语言源码，所有想法都是作者独立设计的。设计语言之前，作者并没有系统学习过设计一门语言需要哪些学科知识，甚至没有学过编译原理，每当遇到问题，作者都是先从原理上想清楚，然后凭借扎实的编程基础，一步步实现，从能运行到高效地运行，CZL反复迭代了多次，由于作者追求完美，所以在一些核心设计上绞尽脑汁，说白了，CZL就是硬刚刚出来的，因此，CZL是完全创新的一门语言，也因此它做到了一些其他编程语言不能做到的事情。

在研发过程中，作者经常拿CZL和其他编程语言做对比测试，主要比较执行性能、内存占用、功能特性和易用性几点，只要有任何一点比他们差，作者就会想方设法去达到、甚至超越他们。这一年的时间里，作者变强了很多，无论是深度、还是广度都有了很大提高，感谢自己的努力。

作者非211和985高校毕业，并且非计算机专业出身，但作者把握住了大学4年的美好时光（主要原因是校区在山沟沟里面，所以作者挺感谢母校的），和技术有关的东西作者多多少少都有点兴趣。编程是一件很苦逼的事情，作者并不喜欢编程，作者只是乐于把想法变成现实，尽管有些想法看似天马行空，其实只要掌握了规律，它就会变得很普通。所以，像作者这么普通的人都能设计出一门不错的编程语言，相信大部分人只要把握时间，都能做出优秀的基础软件。

希望能有更多人投入到基础软件、基础科学研究中，其实它们并没有想象中这么难，它们只是不常见，当你心中有光的时候，一切看不见的东西，最终都会清清楚楚地展现在你面前。