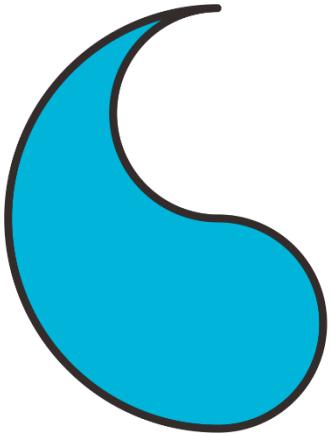


Covariant Script

Covariant Script 编程语言

参考文档(STD20171102)

类型参考

number（数值）类型

number 的字面量由 0~9 十个数和小数点组成，如 12，3.14 等
number 表示的数仅限于全体实数
number 的初始值为 0

boolean（逻辑）类型

boolean 的字面量只有两个，分别是 true（真）和 false（假）
boolean 的初始值为 true（真）

pointer（指针）类型

pointer 没有字面量
pointer 指向一块内存空间，可使用 gcnew 运算符申请一块内存空间
pointer 的初始值为 null（空指针）
null（空指针）指向一块特殊的内存空间，不允许解引用一个空指针
不再使用的内存空间将由 GC（垃圾回收器）自动回收

char（字符）类型

char 的字面量是由单引号括起的单个 ASCII 字符，如'A'，'C'等
特殊符号需使用转义序列来表示

转义序列	符号
\a	响铃
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表
\v	垂直制表
\\	反斜杠
\'	单引号
\"	双引号
\0	空字符

char 的初始值为'\0'（空字符）

string（文字）类型

string 的字面量是由双引号括起的任意个数 ASCII 字符，如"Hello"
string 不支持 Unicode 文字
string 的初始值为""（空串）

array（数组）类型

array 的字面量为大括号扩起的以逗号分隔的任意个数元素，如{1,2,3}
array 是一种元素均匀分布的顺序容器
array 的初始值为{}（空数组）

list（线性表）类型

list 没有字面量
list 是一种元素不均匀分布的顺序容器
list 的初始值为空表

pair（映射）类型

pair 的字面量为冒号对应的一个键值对，如 2:3
pair 的初始值为 0:0（空映射）

hash_map（散列表）类型

hash_map 的字面量为大括号扩起的以逗号分隔的一个或以上映射，如{2:3,4:5}
hash_map 是一种元素均匀分布的无序容器
hash_map 要求其存储的映射的键的类型必须支持生成哈希值
hash_map 的初始值为空表

语法参考

说明

本文档格式为：正文内容 是代码，*斜体内容* 是说明，**加粗内容** 是解释

预处理指令

 # 注释 **以#开头的一行为注释**
@begin
    语句
@end
在@begin 和@end 之间的代码将视为一行语句
也就是说，@begin 和@end 之间的所有换行符都将会被忽略

引入 Package

import *Package 名* **引入一个 Package**
引入的 Package 可以是*.csp 文件(CovScript 包)或者是*.cse 文件(CovScript 扩展)
当两者同时存在时会优先引入*.csp 文件(CovScript 包)

声明 Package

package *Package 名* **声明一个 Package**
原则上包名应和文件名相同

变量定义

var *变量名=表达式* **定义一个变量，初始值为表达式的值**
const var *常量名=表达式* **定义一个常量，其值为表达式的值**

新建一个对象

new *类型* **新建一个指定类型的对象**

申请一块内存

gcnew *类型* **申请一块内存，其中存储的变量为指定类型，返回指向这块内存的指针**
申请的内存将会由垃圾回收器自动释放

运算符与表达式

表达式由操作数和运算符组成
操作数 运算符 操作数
一般有左右两个操作数的运算符是二元运算符
只有一个操作数的运算符是一元运算符
二元运算符有结合律，左结合是从右向左运算，右结合是从左向右运算
所有的运算符都有优先级，优先级越高越先计算

条件运算符

逻辑表达式 ? 表达式 1 : 表达式 2
逻辑表达式的值为真时整个表达式的值为表达式 1
逻辑表达式的值为假时整个表达式的值为表达式 2

临时作用域与名称空间

block
    语句块
end
定义一个临时作用域
临时作用域中的变量会在离开作用域后销毁
namespace *名称空间名*
    语句块
end
定义一个名称空间
名称空间中只允许变量定义，函数定义，类型定义以及名称空间定义

分支语句

if *逻辑表达式*
    语句块
end
逻辑表达式的值为真则执行语句块
if *逻辑表达式*
    语句块 1
else
    语句块 2
end
逻辑表达式的值为真则执行语句块 1
逻辑表达式的值为假则执行语句块 2
switch *表达式*
    case *常量标签*
        语句块
    end
    default
        语句块
    end
end
执行与表达式的值相等的常量标签对应的 case 中的语句块
当无匹配的常量标签时会执行 default 中的语句块，如 default 未找到则跳出
常量标签的类型必须支持生成哈希值

循环语句

while *逻辑表达式*
    语句块
end
当逻辑表达式的值为真时循环执行语句块
loop
    语句块
until *逻辑表达式*
end
直到逻辑表达式的值为真时跳出循环
loop
    语句块
end
循环执行语句块直到用户手动跳出
for *变量名=表达式* to *表达式*
    语句块
end
定义一个变量在闭区间中遍历，步长为 1
for *变量名=表达式* to *表达式* step *表达式*
    语句块
end
定义一个变量在闭区间中遍历，步长为用户指定的值
for *变量名* iterate *表达式*
    语句块
end
表达式的值必须是一个支持 for 遍历的容器
定义一个变量正序遍历容器

控制语句

break	**跳出循环**
continue	**进入下一轮循环**
return	**结束函数并返回 0**
return *表达式*	**结束函数并返回表达式的值**

异常处理

throw *异常*	**抛出一个异常**

try
    语句块
catch *异常名*
    语句块
end
测试代码是否会抛出异常，如抛出则抓取异常

函数

function *函数名(参数列表（可选）)*
    语句块
end
定义一个函数
参数列表中的参数只能指定名称，参数名不可重复，各参数之间以逗号分隔，如：
function test(a0,a1,a2)

Lambda 表达式

[ ](*参数列表（可选）*)->*表达式*　　**定义一个 Lambda 表达式**
参数列表中的参数只能指定名称，参数名不可重复，各参数之间以逗号分隔
Lambda 表达式是一种匿名函数，调用 Lambda 表达式将计算表达式的值并返回

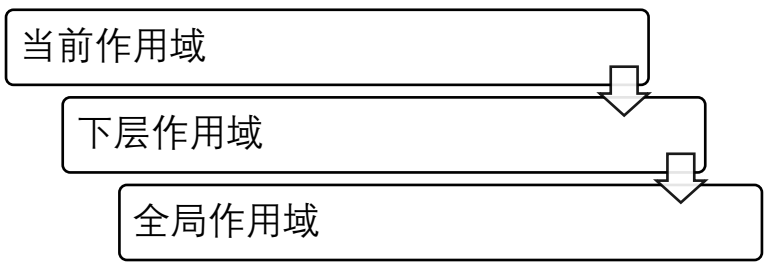
结构

struct *结构名*
    结构体
end
结构定义后结构名就可以作为类型名使用
结构体中只允许变量定义和函数定义
结构体中的变量或函数称为结构体的成员
编译器会为成员函数插入一个隐式的 this 参数
this 指的是调用成员函数的结构实例本身
this 只在成员函数中可用

名称查找

变量名	**从最上层作用域开始向下查找变量**
current.*变量名*	**查找当前作用域中的变量**
global.*变量名*	**查找全局作用域中的变量**
名称空间名. 变量名	**查找名称空间中的变量**
变量名.变量名	**查找结构体或扩展中的变量**

作用与结构以及变量查找方式如图所示



注意：对于最后一种访问方法，仅变量类型为结构或支持扩展的类型时可用，如访问的是扩展或结构中的函数，将会把点运算符左边的变量作为函数的第一个参数传入。
也就是说： char.isspace(ch) **等价于** ch.isspace()

内建类型，函数及变量参考

说明

函数在本文档中表示为　*返回类型 函数名(参数列表（可选）)*
参数列表中的参数表示为　*参数类型 参数名（可选）*
参数类型和返回类型如用中括号([ ])标记，则表示这个类型不是 Covariant Script 内建类型

全局

context	运行环境
char	字符类型
number	数字类型
boolean	逻辑类型
pointer	指针类型
string	文字类型
list	链表类型
array	数组类型
pair	映射类型
hash_map	哈希表类型
exception	异常名称空间
iostream	输入输出流名称空间
system	系统名称空间
runtime	运行时名称空间
math	数学名称空间
number to_integer(var)	将一个变量转换为整数
string to_string(var)	将一个变量转换为文字
string type(var)	获取一个变量的类型名
var clone(var)	复制一个变量并返回
void swap(var,var)	交换两个变量的值

exception 名称空间

string what([exception])	获取异常详情

iostream 名称空间

seekdir	寻位方向名称空间
openmode	打开方式名称空间
istream	输入流名称空间
ostream	输出流名称空间
[istream/ostream] fstream(string path,[openmode] mode)	新建一个文件流，具体类型取决于打开方式
void setprecision(number)	设置输出精度（to_string 的精度）

寻位方向名称空间

start	流的开始
finish	流的结尾
present	当前位置

打开方式名称空间

in	为读打开（输入流）
out	为写打开（清空内容，输出流）
app	为写打开（追加内容，输出流）

输入流名称空间

char get([istream])	读取字符
char peek([istream])	读取下一个字符而不删除
void unget([istream])	放回字符
string getline([istream])	读取一行
number tell([istream])	返回流位置指示器
void seek([istream],number pos)	设置流位置
void seek_from([istream],[seekdir],number offset)	设置相对寻位方向的流位置
boolean good([istream])	检查是否有错误
boolean eof([istream])	检查是否到达文件结尾
var input([istream])	从流中获取输入（格式化）

输出流名称空间

void put([ostream],char)	插入字符
number tell([ostream])	返回流位置指示器
void seek([ostream],number pos)	设置流位置
void seek_from([ostream],[seekdir],number offset)	设置相对寻位方向的流位置
void flush([ostream])	与底层存储设备同步
boolean good([ostream])	检查是否有错误
void print([ostream],var)	向流中输出内容，仅可输出支持 to_string 的类型（不换行）
void println([ostream],var)	向流中输出内容，仅可输出支持 to_string 的类型（换行）

System 名称空间

console	控制台名称空间
args	运行参数(Array)
max	数字类型最大值(Number)
inf	数字类型正无穷(Number)
in	标准输入流
out	标准输出流
number run(string)	在系统环境中运行一条指令，返回错误码
string getenv(string)	获取环境变量的值并返回
void exit(number)	清理资源并退出

控制台名称空间

number terminal_width()	获取控制台宽度
number terminal_height()	获取控制台高度
void gotoxy(number x, number y)	移动光标
void echo(boolean)	设置光标可见性
void clrscr()	清屏
char getch()	获取键盘输入
bool kbhit()	判断是否有键盘输入

Runtime 名称空间

std_version	标准版本号(Number)
void info()	解释器版本信息
number time()	获取计时器的读数，单位毫秒
void delay(number)	使程序暂停一段时间，单位毫秒
number rand(number,number)	获取区间内的伪随机数
number randint(number,number)	获取区间内的伪随机整数
[exception] exception(string)	新建运行时异常
[hash_value] hash(var)	计算一个变量的哈希值
[expression] build([context],string)	构建一个可用于计算的表达式
var solve([context],[expression])	计算一个表达式

字符类型扩展

boolean isalnum(char)	检查字符是否是字母或数字
boolean isalpha(char)	检查字符是否是字母
boolean islower(char)	检查字符是否是小写字母
boolean isupper(char)	检查字符是否是大写字母
boolean isdigit(char)	检查字符是否是数字
boolean iscntrl(char)	检查字符是否是控制字符
boolean isgraph(char)	检查字符是否是图形字符
boolean isspace(char)	检查字符是否是空白字符
boolean isblank(char)	检查字符是否是空格或 tab
boolean isprint(char)	检查字符是否是打印字符
boolean ispunct(char)	检查字符是否是标点符号
char tolower(char)	将字符转换为小写
char toupper(char)	将字符转换为大写

文字类型扩展

string append(string,var)	在尾部追加内容
string insert(string,number,var)	在指定位置处插入内容
string erase(string,number,number)	将范围内的字符删除
string replace(string,number,number,var)	将从指定位置开始的指定个数字符替换
string substr(string,number,number)	从指定位置截取指定长度的子文字
number find(string,string,number)	从指定位置开始从左向右查找一段文字
number rfind(string,string,number)	从指定位置开始从右向左查找一段文字
string cut(string,number)	从尾部删除指定长度的文字
boolean empty(string)	检查文字是否为空
void clear(string)	清空
number size(string)	获取字符个数
string tolower(string)	将文字转换为小写
string toupper(string)	将文字转换为大写
number to_number(string)	将文字转换为数值
array split(string,array)	使用指定的字符集合分割文字

链表类型扩展

iterator	链表迭代器名称空间
var front(list)	访问第一个元素
var back(list)	访问最后一个元素
[iterator] begin(list)	返回指向容器第一个元素的迭代器
[iterator] term(list)	返回指向容器尾端的迭代器
boolean empty(list)	检查容器是否为空
number size(list)	返回容纳的元素数
void clear(list)	删除全部内容
[iterator] insert(list,[iterator],var)	插入元素，插入到迭代器指向的元素之前，返回指向插入的元素的迭代器
[iterator] erase(list,[iterator])	删除元素，返回指向要删除的元素的下一个元素的迭代器
void push_front(list,var)	在容器的开始处插入新元素
void pop_front(list)	删除第一个元素
void push_back(list,var)	将元素添加到容器末尾
void pop_back(list)	删除最后一个元素
void remove(list,var)	删除所有与指定变量相等的元素
void reverse(list)	将该链表的所有元素的顺序反转
void unique(list)	删除连续的重复元素

链表迭代器名称空间

[iterator] forward([iterator])	向前移动迭代器
[iterator] backward([iterator])	向后移动迭代器
var data([iterator])	访问迭代器指向的元素

数组类型扩展

iterator	数组迭代器名称空间
var at(array,number)	访问指定的元素，同时进行越界检查
var front(array)	访问第一个元素
var back(array)	访问最后一个元素
[iterator] begin(array)	返回指向容器第一个元素的迭代器
[iterator] term(array)	返回指向容器尾端的迭代器
boolean empty(array)	检查容器是否为空
number size(array)	返回容纳的元素数
void clear(array)	删除全部内容
[iterator] insert(array,[iterator],var)	插入元素，插入到迭代器指向的元素之前，返回指向插入的元素的迭代器
[iterator] erase(array,[iterator])	删除元素，返回指向要删除的元素的下一个元素的迭代器
void push_front(array,var)	在容器的开始处插入新元素
void pop_front(array)	删除第一个元素
void push_back(array,var)	将元素添加到容器末尾
void pop_back(array)	删除最后一个元素
list to_list(array)	将数组转换为链表

数组迭代器名称空间

[iterator] forward([iterator])	向前移动迭代器
[iterator] backward([iterator])	向后移动迭代器
var data([iterator])	访问迭代器指向的元素

映射类型扩展

var first(pair)	获取第一个元素
var second(pair)	获取第二个元素

哈希表类型扩展

boolean empty(hash_map)	检查容器是否为空
number size(hash_map)	返回容纳的元素数
void clear(hash_map)	删除全部内容
void insert(hash_map,var,var)	插入一对映射
void erase(hash_map,var)	删除键对应的映射
var at(hash_map,var)	访问指定的元素，同时进行越界检查
boolean exist(hash_map,var)	查找是否存在映射

数学名称空间

pi	圆周率(Number)
e	自然底数(Number)
number abs(number)	绝对值
number ln(number)	以 e 为底的对数
number log10(number)	以 10 为底的对数
number log(number a,number b)	以 a 为底 b 的对数
number sin(number)	正弦
number cos(number)	余弦
number tan(number)	正切
number asin(number)	反正弦
number acos(number)	反余弦
number atan(number)	反正切
number sqrt(number)	开方
number root(number a,number b)	a 的 b 次方根
number pow(number a,number b)	a 的 b 次方
number min(number a,number b)	a 和 b 的最小值
number max(number a,number b)	a 和 b 的最大值

扩展类型，函数及变量参考

说明

函数在本文档中表示为　　*返回类型 函数名(参数列表（可选）)*
参数列表中的参数表示为　*参数类型 参数名（可选）*
参数类型和返回类型如用中括号([ ])标记，则表示这个类型不是 Covariant Script 内建类型
扩展需要单独下载并使用 import 语句引入

Regex 扩展

result	Regex Result 名称空间
[regex] build(string)	构建一个正则表达式
[result] match([regex],string)	匹配正则表达式到整个字符序列
[result] search([regex],string)	匹配正则表达式到字符序列的任何部分
string replace([regex],string str,string fmt)	以格式化的替换文本来替换正则表达式匹配的出现位置

Regex Result 名称空间

boolean ready([result])	检查结果是否合法
boolean empty([result])	检查匹配是否成功
number size([result])	返回完全建立的结果状态中的匹配数
number length([result],number)	返回特定子匹配的长度
number position([result],number)	返回特定子匹配首字符的位置
string str([result],number)	返回特定子匹配的字符序列
string prefix([result])	返回目标序列起始和完整匹配起始之间的子序列
string suffix([result])	返回完整匹配结尾和目标序列结尾之间的子序列

Darwin 扩展

ui	Darwin UI 名称空间
drawable	Darwin Drawable 名称空间
black	黑色
white	白色
red	红色
green	绿色
blue	蓝色
pink	粉色
yellow	黄色
cyan	青色

[pixel] pixel(char,[color] front,[color] back)	创建一个像素
[drawable] picture(number width,number height)	创建一幅图片(Drawable)
void load()	加载 Darwin 功能
void exit()	退出 Darwin 功能
boolean is_kb_hit()	判断是否有按键按下
char get_kb_hit()	获取按下的按键
void fit_drawable()	使画布适合当前屏幕大小
[drawable] get_drawable()	获取画布
void update_drawable()	将画布中的内容更新至屏幕上
void set_frame_limit(number fps)	设置帧率
void set_draw_line_precision(number)	设置画线精度

Darwin UI 名称空间

void message_box(string title,string message,string button)	弹出一个消息对话框
var input_box(string title,string message,string default,boolean format)	弹出一个输入对话框

Darwin Drawable 名称空间

void load_from_file([drawable],string path)	从指定路径加载图片(Darwin CDPF 图片文件)
void save_to_file([drawable],string path)	将图片保存至指定路径(Darwin CDPF 图片文件)
void clear([drawable])	清空画布
void fill([drawable],[pixel])	填充画布
void resize([drawable],number width,number height)	重新设置画布大小
number get_width([drawable])	获取画布宽度
number get_height([drawable])	获取画布高度
[pixel] get_pixel([drawable],number x,number y)	获取画布上的点
void draw_pixel([drawable],number x,number y,[pixel])	在画布上画点
void draw_line([drawable],number x1,number y1,number x2,number y2,[pixel])	在画布上画线
void draw_rect([drawable],number x,number y,number width,number height,[pixel])	在画布上绘制线框
void fill_rect([drawable],number x,number y,number width,number height,[pixel])	在画布上填充矩形
void draw_triangle([drawable],number x1,number y1,number x2,number y2,number x3,number y3,[pixel])	在画布上绘制三角形
void fill_triangle([drawable],number x1,number y1,number x2,number y2,number x3,number y3,[pixel])	在画布上填充三角形
void draw_string([drawable],number x,number y,string,[pixel])	在画布上绘制文字
void draw_picture([drawable],number x,number y,[drawable])	将一幅图片绘制到画布上

SQLite 扩展

statement	SQLite 语句名称空间
integer	SQLite 整数类型
real	SQLite 浮点类型
text	SQLite 文本类型
[sqlite] open(string path)	打开或创建一个 SQLite 数据库
[statement] prepare([sqlite] database,string sql)	准备一个 SQLite 语句

SQLite 语句名称空间

boolean done([statement])	判断是否执行完毕
void reset([statement])	重置语句
void exec([statement])	执行语句
number column_count([statement])	获取记录数量
[type] column_type([statement],number index)	获取记录类型
string column_name([statement],number index)	获取记录名称
string column_decltype([statement],number index)	推导记录类型
number column_integer([statement],number index)	获取整数记录
number column_real([statement],number index)	获取浮点记录
string column_text([statement],number index)	获取文本记录
number bind_param_count([statement])	绑定参数数量
void bind_integer([statement],number index,number data)	绑定整数参数
void bind_real([statement],number index,number data)	绑定浮点参数
void bind_text([statement],number index,string data)	绑定文本参数
void clear_bindings([statement])	清除所有绑定

扩展指南

注意事项

由于各个编译器生成的二进制之间并不兼容，请一定注意使用与 CovScript 主程序相同的编译器编译扩展。
CovScript 官方二进制文件使用的编译器为：
Windows 32 位： gcc version 7.1.0 (i686-posix-dwarf-rev0, Built by MinGW-W64 project)
Windows 64 位： gcc version 7.1.0 (x86_64-posix-seh-rev0, Built by MinGW-W64 project)

Covariant Script Extension（CSE）简介

Covariant Script Extension（CSE）是 Covariant Script Runtime 与外界交互的唯一方式，Covariant Script 语言核心功能是 Covariant Script Runtime 通过内部 CSE 调用 C++ Runtime 和 System Call 实现的。
本指南介绍的是 Covariant Script 外部 CSE 的编写方法。外部 CSE 实质上为动态链接库（Dynamic Linked Library），由 Covariant Script 主程序在运行时动态链接并读取特定符号实现对主程序的扩展。

Covariant Script Callable 简介

Covariant Script Callable 是 Covariant Script 语言核心中的抽象函数对象，大部分 Covariant Script Runtime 的函数调用都会被抽象为 Callable 调用。

C/C++ Native Interface（CNI）简介

C/C++ Native Interface（CNI）是允许 Covariant Script 调用 C/C++函数的抽象接口，CNI 能够将 Callable 发起的函数调用转发至 C/C++函数。

准备工作

确保你的编译器与 Covariant Script 二进制文件使用的编译器相同，或者你也可以使用你自己的编译器编译 Covariant Script 源代码得到与之兼容的二进制文件，详情请阅读 README。
请访问 Covariant Script Github 主页（https://github.com/covscript/covscript/）下载最新版本源代码
我们需要用到的是 include 和 sources 这两个文件夹

需要的头文件

你需要把整个 include 文件夹加入 include 搜索目录。
covscript/extension.hpp 包含了了必要的入口和函数
covscript/cni.hpp 开启 CNI 支持

Covariant Script 内建类型

所有 Covariant Script 内建类型皆在 cs 命名空间内有定义
包括：cs::number cs::boolean cs::string cs::list cs::array cs::pair cs::hash_map
字符类型即为 C++的 char 类型，无需特意声明

扩展类对象

每一个 CSE 都必须包含一个静态 cs::extension 对象
例如：static cs::extension my_ext;

入口函数

CSE 要求必须定义一个类似于 main 函数的入口函数，这个函数的作用是加载扩展的功能并返回 CSE 包含的静态 cs::extension 对象：

```
cs::extension* cs_extension()
{
    // TODO
    return &my_ext;
}
```

其中，my_ext 可以使用自定义的 cs::extension 静态对象的名称代替。

向扩展中添加变量或函数

cs::extension 包含一个名为 add_var 的方法，只需在入口函数中调用 my_ext.add_var 即可。

```
my_ext.add_var(变量名[std::string],变量[cs_impl::any]);
```

其中，cs_impl::any 支持 make 静态方法显式构造一个变量：

```
cs_impl::any::make<类型>(构造参数)
```

如果要保护一个变量，可以使用其他构造方法：

```
cs_impl::any::make_protect      变量不能被赋值
cs_impl::any::make_constant     变量不能被赋值或修改
cs_impl::any::make_single       变量不能被赋值，修改或复制
```

Covariant Script 优化器支持在编译期优化被保护的变量
对于函数，Covariant Script 会将所有 Callable 对象视为函数。
Callable 对象支持任意形式为 cs_impl::any(std::deque<cs_impl::any>&)的函数，包括 Lambda 表达式。
如果你不想使用 CNI，你可以直接将你的函数接入到 Callable 中，如：

```
#include <covscript/extension.hpp>
#include <iostream>
static cs::extension my_ext;
cs_impl::any printall(std::deque<cs_impl::any>& args)
{
    for(auto& it:args) std::cout<<it<<std::endl;
    return cs::number(0);
}
```

```
cs::extension* cs_extension()
{
    my_ext.add_var("printall",cs_impl::any::make<cs::callable>(printall));
    return &my_ext;
}
```

注意，如果你不想返回值，请务必返回一个 cs::number(0)，不然 Covariant Script 可能会出现错误。

如果使用 CNI，则简单得多，如：

```
#include <covscript/extension.hpp>
#include <covscript/cni.hpp>
#include <iostream>
static cs::extension my_ext:
void print(const cs_impl::any& val)
{
    std::cout<<val<<std::endl;
}
cs::extension* cs_extension()
{
    my_ext.add_var("print",cs_impl::any::make<cs::callable>(cs::cni(print)));
    return &my_ext;
}
```

注意，CNI 必须通过 Callable 接入 Covariant Script 否则将不能被识别为函数。

CNI 理论上支持任意形态的 C/C++函数，但由于语言限制不支持重载函数和模板函数。

为确保兼容性，接入 CNI 的 C/C++函数最好使用 Covariant Script 内建类型作为参数，因为 CNI 并不支持参数类型的隐式转换。

对于引用，CNI 支持将 Covariant Script 中的变量地址直接映射至参数的引用上，也就是说如果你使用了非常量引用将可以修改 Covariant Script 中的变量，但对于被保护的变量此举将引发错误。为提高性能，请尽量使用常量引用和引用。

对于返回类型为 void 的函数，CNI 将默认返回 0。

向优化器请求优化函数的调用

Covariant Script 优化器支持优化函数调用，但此举的前提是函数手动向优化器请求优化。

原则上请求优化的函数应满足以下要求：

1. 不进行 I/O 操作
2. 不堵塞线程
3. 不修改运行时环境
4. 不依赖运行时环境

要注意的是一旦优化器优化了一个函数调用，那么这个函数调用将不会出现在运行时，比如一个计数器函数请求了优化，那么即便这个函数在循环中调用，可能这个计时器的值将一直不变。

要请求优化，只需将 Callable 构造函数的第二个可选参数设置为 true，并将变量设置为保护即可，如：

```
my_ext.add_var("test",cs_impl::make_protect<cs::callable>(cs::cni(test),true));
```

注意，不要将变量设置为 constant 或 single，这将阻止 Covariant Script 调用这个函数。

类型扩展

可以通过特化 CovScript 类型相关函数扩展类型。
需要注意的是，若在扩展中涉及到 CovScript 内建类型，若想开启对这些类型的支持，请务必包含相关头文件并在扩展的主函数中初始化。如字符串支持，请在文件前部加入#include <covscript/extensions/string.hpp>，并在扩展主函数中调用 string_cs_ext::init()。

比较函数：template<typename T>bool cs_impl::compare(const T &, const T &)

特化此函数以支持比较操作，默认情况下如未找到类型定义的 operator==或者特化的 compare 函数将直接比较变量的地址。

整数转换函数：template<typename T>long cs_impl::to_integer(const T &)

特化此函数以支持向整数的转换。

字符串转换函数：template<typename T>std::string cs_impl::to_string(const T &)

特化此函数以支持向字符串的转换。

哈希函数：template<typename T>std::size_t cs_impl::hash(const T &)

特化此函数以支持生成哈希值。

GC 标记函数：template<typename T>void cs_impl::detach(T &)

对于含有 cs::var 的容器，应特化此函数并分别调用每个 cs::var 实例的 detach 方法。

类型名函数：template<typename T>constexpr const char *cs_impl::get_name_of_type()

特化此函数以支持反馈更友好的类型名，如未特化将默认返回 typeid(T).name()。

类型方法扩展函数：template<typename T>cs::extension_t &cs_impl::get_ext()

特化此函数并返回类型方法的扩展将为此类型引入点运算符成员访问功能的支持。
cs::extension_t 是 std::shared_ptr<cs::extension_holder>的别名，调用 cs::make_shared_extension(cs::extension&)可获取共享的扩展类对象。
对于扩展中的函数，CovScript 将会把调用对象作为第一个参数传入。