

Covariant Script编程语言

扩展指南(v1.0.4)



## 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 v1.0.4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/mikecovlee/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

sources/extension.hpp 包含了了必要的入口和函数

sources/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 “./sources/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 “./sources/extension.hpp”

#include “./sources/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。**