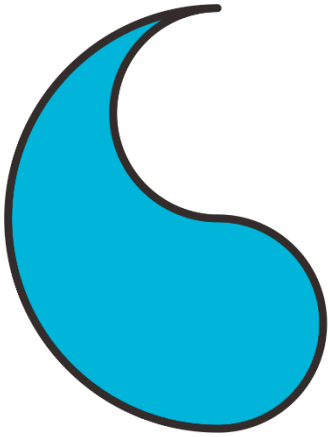

Covariant Script

Covariant Script 编程语言

扩展指南(v1.0.3)

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 v1.0.3 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。
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

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。