

🔑 master ▾

...

[deep-into-node](#) / [chapter2](#) / [chapter2-1.md](#)

yangtuo333 Spelling modification

🕒 History

👤 2 contributors



## C++ 和 JS 交互

本章主要来讲讲如何通过 V8 来实现 JS 调用 C++。JS 调用 C++，分为 JS 调用 C++ 函数（全局），和调用 C++ 类。

### 数据及模板

由于 C++ 原生数据类型与 JavaScript 中数据类型有很大差异，因此 V8 提供了 Value 类，从 JavaScript 到 C++，从 C++ 到 JavaScript 都会用到这个类及其子类，比如：

```
Handle<Value> Add(const Arguments& args){
    int a = args[0]->Uint32Value();
    int b = args[1]->Uint32Value();

    return Integer::New(a+b);
}
```

Integer 即为 Value 的一个子类。

V8 中，有两个模板 (Template) 类 (并非 C++ 中的模板类)：

- 对象模板 (ObjectTemplate)

- 函数模板 (FunctionTemplate) 这两个模板类用以定义 JavaScript 对象和 JavaScript 函数。我们在后续的小节部分将会接触到模板类的实例。通过使用 ObjectTemplate, 可以将 C++ 中的对象暴露给脚本环境, 类似的, FunctionTemplate 用以将 C++ 函数暴露给脚本环境, 以供脚本使用。

## JS 使用 C++ 变量

在 JavaScript 与 V8 间共享变量事实上是很容易的, 基本模板如下:

```
static char sname[512] = {0};

static Handle<Value> NameGetter(Local<String> name, const AccessorInfo& inf
    return String::New((char*)&sname, strlen((char*)&sname));
}

static void NameSetter(Local<String> name, Local<Value> value, const Access
    Local<String> str = value->ToString();
    str->WriteAscii((char*)&sname);
}
```

定义了 NameGetter, NameSetter 之后, 在 main 函数中, 将其注册在 global 上:

```
// Create a template for the global object.
Handle<ObjectTemplate> global = ObjectTemplate::New();
//public the name variable to script
global->SetAccessor(String::New("name"), NameGetter, NameSetter);
```

## JS 调用 C++ 函数

在 JavaScript 中调用 C++ 函数是脚本化最常见的方式, 通过使用 C++ 函数, 可以极大程度的增强 JavaScript 脚本的能力, 如文件读写, 网络 / 数据库访问, 图形 / 图像处理等等, 类似于 JAVA 的 jni 技术。

在 C++ 代码中, 定义以下原型的函数:

```
Handle<Value> func(const Arguments& args){//return something}
```

然后, 再将其公开给脚本: global-

```
>Set(String::New("func"),FunctionTemplate::New(func));
```

## JS 使用 C++ 类

如果从面向对象的视角来分析，最合理的方式是将 C++ 类公开给 JavaScript，这样可以将 JavaScript 内置的对象数量大大增加，从而尽可能少的使用宿主语言，而更大的利用动态语言的灵活性和扩展性。事实上，C++ 语言概念众多，内容繁复，学习曲线较 JavaScript 远为陡峭。最好的应用场景是：既有脚本语言的灵活性，又有 C/C++ 等系统语言的效率。使用 V8 引擎，可以很方便的将 C++ 类“包装”成可供 JavaScript 使用的资源。

我们这里举一个较为简单的例子，定义一个 Person 类，然后将这个类包装并暴露给 JavaScript 脚本，在脚本中新建 Person 类的对象，使用 Person 对象的方法。首先，我们在 C++ 中定义好类 Person：

```
class Person {
private:
    unsigned int age;
    char name[512];

public:
    Person(unsigned int age, char *name) {
        this->age = age;
        strncpy(this->name, name, sizeof(this->name));
    }

    unsigned int getAge() {
        return this->age;
    }

    void setAge(unsigned int nage) {
        this->age = nage;
    }

    char *getName() {
        return this->name;
    }

    void setName(char *nname) {
        strncpy(this->name, nname, sizeof(this->name));
    }
};
```

Person 类的结构很简单，只包含两个字段 age 和 name，并定义了各自的 getter/setter. 然后我们来定义构造器的包装：

```

Handle<Value> PersonConstructor(const Arguments& args){
    Handle<Object> object = args.This();
    HandleScope handle_scope;
    int age = args[0]->Uint32Value();

    String::Utf8Value str(args[1]);
    char* name = ToCString(str);

    Person *person = new Person(age, name);
    object->SetInternalField(0, External::New(person));
    return object;
}

```

≡ 187 lines (152 sloc) | 7.27 KB

...

从函数原型上可以看出，构造器的包装与上一小节中，函数的包装是一致的，因为构造函数在 V8 看来，也是一个函数。需要注意的是，从 args 中获取参数并转换为合适的类型之后，我们根据此参数来调用 Person 类实际的构造函数，并将其设置在 object 的内部字段中。紧接着，我们需要包装 Person 类的 getter/setter：

```

Handle<Value> PersonGetAge(const Arguments& args){
    Local<Object> self = args.Holder();
    Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));

    void *ptr = wrap->Value();

    return Integer::New(static_cast<Person*>(ptr)->getAge());
}

Handle<Value> PersonSetAge(const Arguments& args) {
    Local<Object> self = args.Holder();
    Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));

    void* ptr = wrap->Value();

    static_cast<Person*>(ptr)->setAge(args[0]->Uint32Value());
    return Undefined();
}

```

而 getName 和 setName 的与上例类似。在对函数包装完成之后，需要将 Person 类暴露给脚本环境：首先，创建一个新的函数模板，将其与字符串“Person”绑定，并放入 global：

```
Handle<FunctionTemplate> person_template = FunctionTemplate::New(PersonCons  
person_template->SetClassName(String::New("Person"));  
global->Set(String::New("Person"), person_template);
```

然后定义原型模板:

```
Handle<ObjectTemplate> person_proto = person_template->PrototypeTemplate();  
  
person_proto->Set("getAge", FunctionTemplate::New(PersonGetAge));  
person_proto->Set("setAge", FunctionTemplate::New(PersonSetAge));  
  
person_proto->Set("getName", FunctionTemplate::New(PersonGetName));  
person_proto->Set("setName", FunctionTemplate::New(PersonSetName));
```

最后设置实例模板:

```
Handle<ObjectTemplate> person_inst = person_template->InstanceTemplate();  
person_inst->SetInternalFieldCount(1);
```

## C++ 调用 JS 函数

我们直接看下 src/timer\_wrap.cc 的例子, V8 编译执行 timer.js, 构造了 Timer 对象。

```
static void OnTimeout(uv_timer_t* handle) {  
    TimerWrap* wrap = static_cast<TimerWrap*>(handle->data);  
    Environment* env = wrap->env();  
    HandleScope handle_scope(env->isolate());  
    Context::Scope context_scope(env->context());  
    wrap->MakeCallback(kOnTimeout, 0, nullptr);  
}  
  
inline v8::Local<v8::Value> AsyncWrap::MakeCallback(uint32_t index, int argc  
    v8::Local<v8::Value> cb_v = object()->Get(index);  
    CHECK(cb_v->IsFunction());  
    return MakeCallback(cb_v.As<v8::Function>(), argc, argv);  
}
```

TimerWrap 对象通过数组的索引寻址，找到 Timer 对象索引 0 的对象，而对其赋值的是在 lib/timer.js 里面的 `list._timer[kOnTimeout] = listOnTimeout;`。这边找到的对象是个 Function，后面忽略 domains 异常处理等，就是简单的调用 Function 对象的 Call 方法，并且传入上文提到的 Context 和参数。

```
Local<Value> ret = callback->Call(recv, argc, argv);
```

这就实现了 C++ 对 JS 函数的调用。

## 总结

## 参考

- [1]. <https://www.ibm.com/developerworks/cn/opensource/os-cn-v8engine/>
- [2]. <https://developers.google.com/v8/embed>