Nood.js C++ 模块 addon插件（四） - 程序园

官方node手册 一篇重量级的：

1 return 函数

执行函数返回hello world：

```
#define BUILDING_NODE_EXTENSION
#include <node.h>

using namespace v8;

Handle<Value> MyFunction(const Arguments& args) {
  HandleScope scope;
  return scope.Close(String::New("hello world"));
}

Handle<Value> CreateFunction(const Arguments& args) {
  HandleScope scope;

  Local<FunctionTemplate> tpl =
FunctionTemplate::New(MyFunction);
  Local<Function> fn = tpl->GetFunction();
  fn->SetName(String::NewSymbol("theFunction"));

  return scope.Close(fn);
}

void Init(Handle<Object> target) {
  target->Set(String::NewSymbol("createFunction"),
      FunctionTemplate::New(CreateFunction)->GetFunction());
}

NODE_MODULE(addon, Init)我们着重看下面两行代码


 Local<FunctionTemplate> tpl =
FunctionTemplate::New(MyFunction);
  Local<Function> fn = tpl->GetFunction();
  fn->SetName(String::NewSymbol("theFunction"));
```

FunctionTemplate用来创建函数的，在一个上下文中仅能创建一个FunctionTemplate

创建的函数声明周期等于这个上下文存在的声明周期，一个FunctionTemplate可以拥有属性，当你创建它后，这些属性可以加给他。

一个FunctionTemplate可以由原型模板，这个原型模板是用来创建原型链对象。

下面是V8官网的代码示例：

```
  v8::Local<v8::FunctionTemplate> t = v8::FunctionTemplate::New();
    t->Set("func_property", v8::Number::New(1));


v8::Local<v8::Template> proto_t = t->PrototypeTemplate();
    proto_t->Set("proto_method", v8::FunctionTemplate::New(InvokeCallback));
    proto_t->Set("proto_const", v8::Number::New(2));

    v8::Local<v8::ObjectTemplate> instance_t = t->InstanceTemplate();
    instance_t->SetAccessor("instance_accessor", InstanceAccessorCallback);
    instance_t->SetNamedPropertyHandler(PropertyHandlerCallback, ...);
    instance_t->Set("instance_property", Number::New(3));

    v8::Local<v8::Function> function = t->GetFunction();
    v8::Local<v8::Object> instance = function->NewInstance();
```

下面是js代码：

```
func_property in function == true;
   function.func_property == 1;

   function.prototype.proto_method() invokes 'InvokeCallback'
   function.prototype.proto_const == 2;

   instance instanceof function == true;
   instance.instance_accessor calls
'InstanceAccessorCallback'
   instance.instance_property == 3;
```

具体的我们如何利用c++函数给node，我们的大致思路是：

先写C++函数，然后里哟个函数模板转化成js模板函数，接着为这个函数模板做一些属性设置，比如原型链或者静态属性等，

最后调用这个函数模板指针的GetFunction方法，转化成local<function>类型，供node使用

2　包裹c++对象

官网接下来的例子是用c++的类Myobjet输出给js，然后js通过new的操作来获取它。

先看代码：

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

void InitAll(Handle<Object> target) {
  MyObject::Init(target);
}

NODE_MODULE(addon, InitAll)
```

myobject.h的代码；

```
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Handle<v8::Object> target);

 private:
  MyObject();
  ~MyObject();

  static v8::Handle<v8::Value> New(const v8::Arguments& args);
  static v8::Handle<v8::Value> PlusOne(const v8::Arguments&
args);
  double counter_;
};

#endif
```

myobject.cc：

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

MyObject::MyObject() {};
MyObject::~MyObject() {};

void MyObject::Init(Handle<Object> target) {

  Local<FunctionTemplate> tpl = FunctionTemplate::New(New);
  tpl->SetClassName(String::NewSymbol("MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  tpl->PrototypeTemplate()->Set(String::NewSymbol("plusOne"),
      FunctionTemplate::New(PlusOne)->GetFunction());

  Persistent<Function> constructor = Persistent<Function>::New(tpl-
>GetFunction());

  target->Set(String::NewSymbol("MyObject"), constructor);
}

Handle<Value> MyObject::New(const Arguments& args) {
  HandleScope scope;

  MyObject* obj = new MyObject();
  obj->counter_ = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
  obj->Wrap(args.This());

  return args.This();
}

Handle<Value> MyObject::PlusOne(const Arguments& args) {
  HandleScope scope;

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.This());
  obj->counter_ += 1;

  return scope.Close(Number::New(obj->counter_));
}
```

这里有几个需要我们仔细消化下：

tpl->InstanceTemplate()->SetInternalFieldCount(1);//设置从这个模版实例化的数量

InstanceTemplate方法用来获得tpl模版的实例的模版，返回Local<ObjectTemplate>而SetInternalFieldCount方法用来对这个对象模版设置内部字段的数量，这里设置为1.这里我也不是很明白，起初以为是只能设置对象的一个属性，后来发现不是那样的，谷歌查询之后发现是映射关系的参数。

Persistent<Function> constructor =Persistent<Function>::New(tpl->GetFunction()); //获得构造函数

//Persistent类表示需要明确的调用Persistent::Dispose才回去GC
target->Set(String::NewSymbol("MyObject"), constructor); //设置Myobject为构造函数

Persistent这个类型不同于local，他不会自动的去GC，而要求用户手动去Dispose，所以这里就等于声明了全局变量，但是在node端这里用Local替换Persistent是一样的。

Javascript中的this指针可以通过Arguments::This()得到

js代码：

```
var addon =
require('./build/Release/addon');

var obj = new addon.MyObject(10);
console.log( obj.plusOne() );
console.log( obj.plusOne() );
console.log( obj.plusOne() );
```

3、官网接下来一个例子是用createObject方法来代替new关键字，和上面代码区别只有一块：

```
Handle<Value> MyObject::NewInstance(const Arguments& args) {
  HandleScope scope;

  const unsigned argc = 1;
  Handle<Value> argv[argc] = { args[0] };
  Local<Object> instance = constructor->NewInstance(argc,
argv);

  return scope.Close(instance);
 }
```

官方api对NewInstance的定义：

V8EXPORT Local< Object > NewInstance (int argc, Handle< Value >argv[]) const

注意这里的NewInstance不是MyObject的方法。

js代码：

```
var addon =
require('./build/Release/addon');

 var obj = addon.createObject(10);
 console.log( obj.plusOne() );
 console.log( obj.plusOne() );
 console.log( obj.plusOne() );

 var obj2 = addon.createObject(20);
 console.log( obj2.plusOne() );
 console.log( obj2.plusOne() );
 console.log( obj2.plusOne() );
```

总结一下这2段程序的流程吧：

addon.cc文件

1、定义Init函数，用来初始化，在Init函数中执行Myobject的静态方法Init、

2、Init函数第二行定义一个对外的方法createobject

3、定义createobject方法，执行Myobject类的静态方法NewInstance，然后将其值返回。

myobject.h

1、MYOBJECT_H这个常量如果定义过则不重复定义，如果没有则执行myobject.h，防止重复加载

2、定义Myobject类，继承自node命名空间下的ObjectWrap基类

3、声明2个public的静态方法init和NewInstance

4、声明private的属性和方法，声明构造和析构函数，静态属性 constructor，静态方法new和plusone

最重要的myobject.cc文件，主要是对myobject类各种方法的定义（声明在类中已经做过）

1、将myobject.h包涵进来

2、定义构造和析构函数

3、定义Init函数

3-1、定义一个函数模版，模版来源是c++的myobject类的静态方法new

3-2、设置模版函数tpl的class名，为Myobject

3-3、设置这个class名为myobject的构造函数（js中叫这个吧，不是基类）的原形链上的方法plusone，并且把c++中myobject类的静态私有方法plusone作为函数模版赋值给它

3-4、最后将constructor变量指向tpl转化为的Persistent<Function>类型的js函数对象。

4、定义new函数

4-1、创建一个指向Myobject类的实例的指针变量 obj

4-2、将这个实例的counter_属性初始赋值为0或者参数

4-3、将local<object>对象wrap包装

4-4、返回local<object>，其实这里就是js实例的对象，就是js的createobject()方法的返回值

5、定义NewInstance方法

5-1、生成参数，然后调用

Local<Object> instance =constructor->NewInstance(argc, argv);

上面这段其实就是生成tpl函数模版的实例，同时将参数传进去，就相当于执行了Myobject::new的方法

5-2、最后将生成的实例通过scope.Close返回给客户端

6、定义PlusOne方法，将js的实例对象转化为C++对应的实例对象指针，然后执行属性counter_加1，最后返回数据

4、wrap和unwrap

除了wrap node 对象C++对象以外，我们还可以用过node::ObjectWrap::Unwrap这个方法来unwrap的node对象，供C++使用

wrap.cc代码：

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

Handle<Value> CreateObject(const Arguments& args) {
  HandleScope scope;
  return scope.Close(MyObject::NewInstance(args));
}

Handle<Value> Add(const Arguments& args) {
  HandleScope scope;

  MyObject* obj1 = node::ObjectWrap::Unwrap<MyObject>(
      args[0]->ToObject());
  MyObject* obj2 = node::ObjectWrap::Unwrap<MyObject>(
      args[1]->ToObject());

  double sum = obj1->Val() + obj2->Val();
  return scope.Close(Number::New(sum));
}

void InitAll(Handle<Object> target) {
  MyObject::Init();

  target->Set(String::NewSymbol("createObject"),
      FunctionTemplate::New(CreateObject)-
>GetFunction());

  target->Set(String::NewSymbol("add"),
      FunctionTemplate::New(Add)->GetFunction());
}

NODE_MODULE(addon, InitAll)
```

上面这段代码重点就是：

```
  MyObject* obj1 = node::ObjectWrap::Unwrap<MyObject>
(
      args[0]->ToObject());
```

我们可以利用这段代码，将node端的对象转化C++端的对象

mywrap.h

```
#define BUILDING_NODE_EXTENSION
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>

class MyObject : public node::ObjectWrap {
 public:
  static void Init();
  static v8::Handle<v8::Value> NewInstance(const v8::Arguments&
args);
  double Val() const { return val_; }

 private:
  MyObject();
  ~MyObject();

  static v8::Persistent<v8::Function> constructor;
  static v8::Handle<v8::Value> New(const v8::Arguments& args);
  double val_;
};

#endif
mywrap.cc
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

MyObject::MyObject() {};
MyObject::~MyObject() {};

Persistent<Function> MyObject::constructor;

void MyObject::Init() {

  Local<FunctionTemplate> tpl = FunctionTemplate::New(New);
  tpl->SetClassName(String::NewSymbol("MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  constructor = Persistent<Function>::New(tpl->GetFunction());
}

Handle<Value> MyObject::New(const Arguments& args) {
  HandleScope scope;

  MyObject* obj = new MyObject();
  obj->val_ = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
```

```
  obj->Wrap(args.This());

  return args.This();
}

Handle<Value> MyObject::NewInstance(const Arguments& args) {
  HandleScope scope;

  const unsigned argc = 1;
  Handle<Value> argv[argc] = { args[0] };
  Local<Object> instance = constructor->NewInstance(argc, argv);

  return scope.Close(instance);
}
```

js代码：

```
var addon =
require('./build/Release/addon');

var obj1 = addon.createObject(10);
var obj2 = addon.createObject(20);
var result = addon.add(obj1, obj2);

console.log(result);
```