C++ Addon with Node.js

Author : Rhio Kim
email : rhio.kim@gmail.com
blog : http://rhio.tistory.com
twitter : http://twitter.com/rhiokim

Preface

Node 는 V8 자바스크립트 엔진을 기반으로 동작하는 서버 사이드 자바스크립트로 V8 엔진과 유기적으로 동작할 수 있는 C/C++로 작성된 추가기능을 제공합니다.

이것은 구조적으로 추가 기능과 동적으로 공유 객체를 연결하고 C, C++ 라이브러리를 위한 연결 지점을 제공합니다.

V8 엔진도 C++ 로 작성된 라이브러리로 자바스크립트의 오브젝트를 만들거나 함수 호출등의 인터페이스에 사용됩니다. 문서화는 대부분 v8.h 헤더파일에 기록되어 있어 추가기능을 만들기 위해서는 자주 참고하게 될 것입니다. (Node의 소스 트리중 deps/v8/include/v8.h를 참고하세요.)

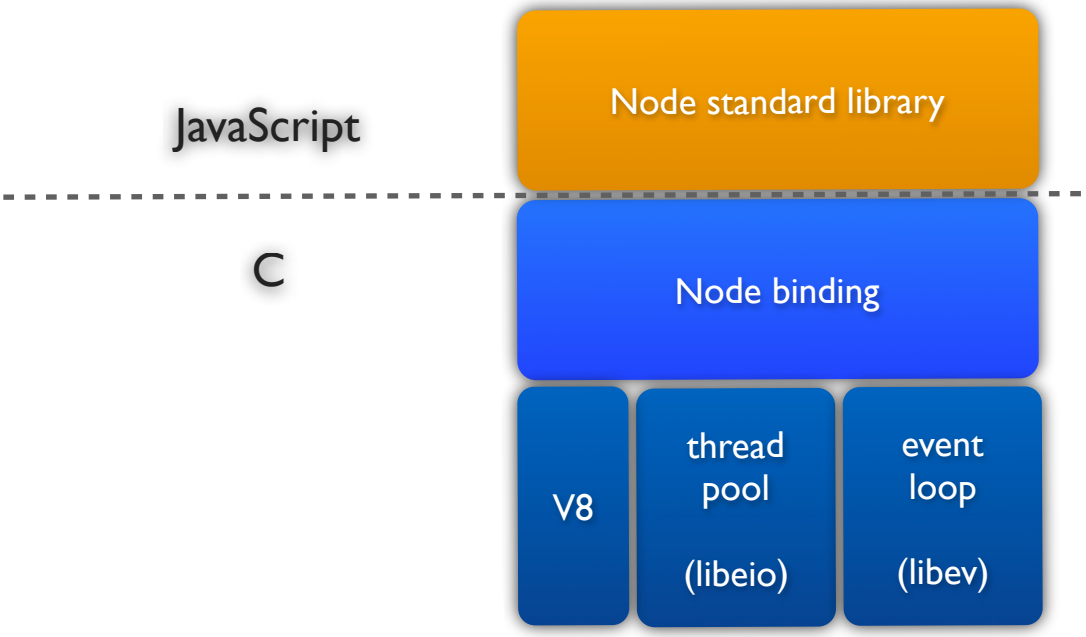
Node 의 동작에 있어서 가장 중요한 역할을 하는 두가지 라이브러리가 있습니다. 그것은 libev 인터페이스와, libeio 쓰레드 풀 라이브러리 입니다.

이 문서를 통해 Node.js 의 Addon 의 구조와 원리를 이해하고 직접 Native Extension 개발을 습득해봅니다.

Addon

Node에서 Addon은 이미 제공하는 라이브러리 외에 사용자가 직접 고성능의 Native Extension을 개발하여 Node의 라이브러리와 동일한 레벨에서 사용할 수 있는 구조를 지원합니다.

Structure



Node는 크게 자바스크립트 언어 레이어와 C 언어 레이어로 나뉜다. C 언어 레이어는 C++로 개발된 구글의 V8 자바스크립트 엔진은 쓰레드 풀 라이브러리인 libeio 와 이벤트 루프 라이브러리인 libev 로 시스템에서 자바스크립트이 가능하도록 하는 코어 역할을 한다.

Node binding 레이어는 Blocking POSIX 시스템을 비동기적으로 호출하기 위한 대부분의 래퍼들을 구현해 놓았다. 이것들은 Node 저장소에 있는 소스 중 src/*.cc 에 포함되어 있고 Node standard library 의 대부분이 래퍼들과 동작하게 된다.

Node standard library 레이어는 Node binding 에 구현해 놓은 래퍼들이 제공하는 API를 이용해 자바스크립트에서 효율적이고 손쉽게 시스템상에서 I/O 프로그래밍을 할 수 있도록 지원하는 서버 사이드 라이브러리 이다.

Libraries

1. C event loop 라이브러리 (libev)
   a. http://cvs.schmorp.de/libev/ev.html
   b. 파일 디스크립터가 읽을 수 있게 되고, 타이머를 기다릴 때 혹은 신호를 받기를 기다릴때 등에서 libev 인터페이스가 필요하다.
   c. 모든 I/O 작업을 할때 libev 인터페이스는 꼭 필요하다.

2. C thread pool 라이브러리 (libeio)
   a. http://pod.tst.eu/http://cvs.schmorp.de/libeio/eio.pod
   b. 블럭킹 POSIX 시스템을 비동기적으로 실행하는데 사용한다.
   c. Non-blocking 프로그램을 가능하게 한다.

1. Creating Modules

CommonJS 커뮤니티에서 주도적으로 잘 알려진 자바스크립트 라이브러리들을 모듈처럼 패키지하는 표준화를 위해 노력하고 있다.

1. Reference sites
   a. Proposal to ECMA TC39 : https://docs.google.com/Doc?id=dfgxb7gk_34gpk37z9v&hl=en
   b. Presentation to ECMA TC39 : https://docs.google.com/present/view?hl=en&id=dcd8d5dk_0cs639jg8

2. Write Module(helloworld.js)

```
var NodeModule = function(){
  this._friendName = "world";
  this._message = "hello";
}

NodeModule.prototype.hello = function(){
  return this._message +" "+ this._friendName +"!";
};

NodeModule.prototype.myNameIs = function(name){
  this._friendName = name || "world";
};

exports.NodeModule = NodeModule;
```

위의 소스는 흔히 자바스크립트로 특정 기능을 수행하는 함수와 속성으로 이뤄진 NodeModule 클래스입니다. 하지만 다른 점이 있다면 마지막 줄에 있는 Node에서 제공하는 exports 객체의 NodeModule 속성에 NodeModule 생성자 함수를 설정하는 것입니다.

이것은 Node 에서 모듈을 사용할 수 있도록 연결 지점을 제공하게 된다.

3. 사용(Usage)

```
$ node
$ Type '.help' for options.
node>
node>var module = require('./helloworld');
node>var NodeModule = new module.NodeModule();
node>NodeModule.hello();
'hello world!'
node>NodeModule.myNameIs('rhio');
node>NodeModule.hello();
'hello rhio'
```

사용하는 방법도 모두 비슷하지만 한가지 특이한 점이 있다면 require 함수입니다. 이것 역시 Node 에서 제공하는 글로벌 함수로 자바스크립트로 작성된 모듈과 연결하게 되고 연결된 모듈은 흔히 자바스크립트 클래스를 다루듯이 사용하게 됩니다.

```
$ cat app.js
var module = require('./helloworld');
var NodeModule = new module.NodeModule();
console.log(NodeModule.hello());

NodeModule.myNameIs('rhio'))
console.log(NodeModule.hello());
$
$ node app.js
hello world!
hello rhio!
```

위의 예제는 미리 작성된 helloworld.js 모듈을 사용하여 작성된 예제입니다. 이 app.js 는 Node를 통해 위와 같이 실행됩니다.

2. Writing Native Extension

먼저 C++ 부가 기능을 만들기에 앞서 Node에서 자바스크립트를 이용한 모듈을 개발하는 방법을 이해해야 합니다. 작성하고 사용하기 위해 몇가지 규칙이 있지만 매우 간단하여 예시를 통해 이해해보도록 합니다.

작은 부가기능을 만들어 보기 위해서 C++에서 다음과 같은 hello.cc 소스코드를 작성합니다.

```
#include <v8.h>

using namespace v8;

extern "C" void
init (Handle<Object> target)
{
  HandleScope scope;
  target->Set(String::New("hello"), String::New("World"));
}
```

위의 소스는 Node 의 부가기능이 동작하기 위한 가장 기본적인 코드입니다. 어떻게 보면 매우 간단해 보이지만 C나 C++의 코드에 친숙하지 않는 개발자들에게는 코드의 의미를 이해하는데 약간의 어려움이 있습니다.

```
extern “C” void init (Handle<Object> target)
```

모든 부가기능에서는 위의 코드와 함께 init 함수가 호출되도록 작성되어야만 합니다. 이 코드가 바로 부가기능이 동적으로 공유 객체에 연결되는 시점이 됩니다.

Node의 모든 부가 기능들은 이 서명과 함께 init 함수 호출되도록 내보내야만 합니다.

3. wscript

Waf는 애플리케이션 설치를 위한 환경설정, 컴파일을 위한 Python 기반의 프레임워크입니다. 이것은 Scons, Autotools, CMake, Ant 와 같은 유사한 빌드툴의 컨셉을 그대로 가져왔고 node-waf 는 바로 Waf 를 사용하여 node 의 부가기능으로 C++ 로 개발되어진 기능을 빌드하는데 사용되어집니다.

자세한 사항은 http://code.google.com/p/waf/[WAF] 을 참고하세요.

위의 예시에서 작성된 hello.cc 파일을 node의 addon 으로 사용하기 위해서는 빌드 과정을 거쳐야 합니다. Waf 를 사용해서 빌드를 하는 Node 에서는 wscript 를 작성해야 해야합니다.

간단한 예시를 통해 wscript 에 대해서 자세히 알아보겠습니다.

```
srcdir = '.'
blddir = 'build'
VERSION = '0.0.1'

def set_options(opt):
  opt.tool_options('compiler_cxx')

def configure(conf):
  conf.check_tool('compiler_cxx')
  conf.check_tool('node_addon')

def build(bld):
  obj = bld.new_task_gen('cxx', 'shlib', 'node_addon')
  obj.target = 'hello'
  obj.source = 'hello.cc'
```

srcdir은 소스 디렉토리 blddir 은 빌드될 디렉토리입니다. 이 설정은 Waf 에서 빌드를 위한 소스 디렉토리와 빌드 디렉토리를 분리하여 서로 섞여 혼란스럽지 않도록 지정한 기본적인 설정입니다.

```
def set_options(opt):
  opt.tool_options('compiler_cxx')
```

build 옵션을 설정하는 명령줄입니다. 이 부분에는 빌드 시 필요한 다양한 커멘드 라인 옵션을 지정할 수 있습니다.

```
def configure(conf):
  conf.check_tool('compiler_cxx')
  conf.check_tool('node_addon')
```

위의 부분은 환경 구성 기능으로 커멘드 라인에서의 빌드 옵션을 입력하였을 때 실행되는 부분입니다. Node 에서는 check_tool 기능을 통해서 컴파일러와 Addon 에 대한 유효성 체크기능을 기본적으로 수행해야 합니다.

```
def build(bld):
  obj = bld.new_task_gen('cxx', 'shlib', 'node_addon')
  obj.target = 'hello'
  obj.source = 'hello.cc'
```

Waf 의 동작원리와 개념적인 이해를 위해서 아래의 문서를 참고하세요.
http://waf.googlecode.com/svn/docs/wafbook/single.html#_waf_project_definition

위와 같이 hello.cc 를 위한 wscript 작성까지 완료되었다면 Addon을 빌드하기 위한 준비는 모두 갖춰진 상태입니다.

4. Build

위의 과정을 거쳐 생성된 hello.cc 와 wscript 를 통해 빌드를 해봅니다.

build 명령 수행

```
$ pwd
/path/to/node-hello/
$ ls
hello.cc            wscript
$ node-waf configure build
```

build 과정

```
Checking for program g++ or c++         : /usr/bin/g++
Checking for program cpp                : /usr/bin/cpp
Checking for program ar                 : /usr/bin/ar
Checking for program ranlib             : /usr/bin/ranlib
Checking for g++                        : ok
Checking for node path                  : ok /Users/rhio/.node_libraries
Checking for node prefix                : ok /Users/rhio/local
'configure' finished successfully (0.040s)
Waf: Entering directory `/Users/rhio/node-apps/node-hello/build'
[1/2] cxx: node-hello.cc -> build/default/node-hello_1.o
[2/2] cxx_link: build/default/node-hello_1.o -> build/default/node-hello.node
Waf: Leaving directory `/Users/rhio/node-apps/node-hello/build'
'build' finished successfully (0.533s)
```

build 확인

```
$ pwd
/path/to/node-hello/build/default
$ ls
hello.node                  hello_1.o
```

위의 과정을 통해 hello.node 가 생성됨을 알 수 있습니다. Native Addon 이 생성되었고 이제 자바스크립트를 이용하여 Native Addon을 사용하는 방법을 알아보도록 하겠습니다.

5.Usage

위의 과정을 통해 build/default/hello.node 에 생성됩니다.

```
$ pwd
/path/to/node-hello/
$ ls
hello.cc        wscript        build
```

지정된 test 하는 규칙은 없지만 addon을 개발하는 개발자들은 tests 디렉토리를 생성하고 테스트에 필요한 작업을 진행합니다.

tests 디렉토리는 빌드 과정에서 생성되는 것이 아닌 테스트를 진행하기 위해서 직접 생성해야 합니다.

```
$ pwd
/path/to/node-hello/
$ mkdir tests
$ cd tests
$ vi hello.js
```

Test

hello.js 작성

```
#hello.js
var addon = require('../build/defaul/hello');
console.log(addon);
console.log(addon.hello);
```

위의 소스는 native addon 을 로드하여 addon 의 변수에 native addon 공유 객체로 연결을 하게 되고 addon 을 통해서 native addon 의 기능을 사용하게 된다.

Native addon 실행

```
$ pwd
/path/to/node-hello/tests
$ ls
hello.js
$ node hello.js
{ hello : 'World' }
World
```

위에서 작성된 hello.js 를 수행하면 다음과 같은 실행된다.

6.References & See more

V8 Embedder's Documentation - http://code.google.com/apis/v8/embed.html
위의 문서는 Node.js 의 부가기능을 작성하기 위해서 꼭 읽어봐야 합니다.

V8 Cookbook - http://create.tpsitulsa.com/wiki/V8_Cookbook

NativeClient - http://code.google.com/p/nativeclient/

Writing Node.js Native Extensions
https://www.cloudkick.com/blog/2010/aug/23/writing-nodejs-native-extensions/

Appendix

POSIX System : 이식 가능 운영 체제 인터페이스. 서로 다른 UNIX OS의 공통 API를 정리하여 이식성이 높은 유닉스 응용 프로그램을 개발하기 위한 목적으로 IEEE가 책정한 애플리케이션 인터페이스 규격이다.
  위키 - http://ko.wikipedia.org/wiki/POSIX
  텀즈코리아 - http://www.terms.co.kr/POSIX.htm

REPL : Read Eval Print Loop 는 독립실행형 프로그램을 손쉽게 다른 프로그램과 통합하여 이용할 수 있도록 해준다. http://nodejs.org/api.html#repl-311

require : 모듈을 요청하기 위한 글로벌 함수

modules : CommonJS의 모듈 시스템을 사용하고 있다. http://nodejs.org/api.html#modules-314