一、编写 Node.js 原生扩展

Node.js 是一个强大的平台，理想状态下一切都都可以用 javascript 写成。然而，你可能还会用到许多遗留的库和系统，这样的话使用 c++编写 Node.JS 扩展会是一个不错的注意。

以下所有例子的源代码可在 node 扩展示例中找到 。

编写 Node.js C + +扩展很大程度上就像是写 V8 的扩展； Node.js 增加了一些接口，但大部分时间你都是在使原始的 V8 数据类型和方法，为了理解以下的代码，你必须首先阅读 V8 引擎嵌入指南。

Javascript 版本的 Hello World

在讲解 C++版本的例子之前，先让我们来看看在 Node.js 中用 Javascript 编写的等价模块是什么样子。这是一个最简单的 Hello World，也不是通过 HTTP，但它展示了 node 模块的结构，而其接口也和大多数 C++扩展要提供的接口差不多：

```
HelloWorldJs = function() {
  this.m_count = 0;
};

HelloWorldJs.prototype.hello = function()
{
  this.m_count++;
  return "Hello World";
};

exports.HelloWorldJs = HelloWorldJs;
```

正如你所看到的，它使用 prototype 为 HelloWorldJs 类创建了一个新的方法。请注意，上述代码通过将 HelloWorldJS 添加到 exports 变量来暴露构造函数。

要在其他地方使用该模块，请使用如下代码：

```
var helloworld = require('helloworld_js');
var hi = new helloworld.HelloWorldJs();
console.log(hi.hello()); // prints "Hello World" to stdout
```

C++版本的 Hello World

要开始编写C++扩展，首先要能够编译Node.js（请注意，我们使用的是Node.js 2.0版本）。本文所讲内容应该兼容所有未来的0.2.x版本。一旦编译安装完node，编译模块就不在需要额外的东西了。

完整的源代码可以在这里找到 。在使用Node.js或V8之前，我们需要包括相关的头文件：

```
#include <v8.h>
#include <node.h>

using namespace node;
using namespace v8;
```

在本例子中我直接使用了V8和node的命名空间，使代码更易于阅读。虽然这种用法和谷歌的自己的C++编程风格指南相悖，但由于你需要不停的使用V8定义的类型，所以目前为止的大多数node的扩展仍然使用了V8的命名空间。

接下来，声明HelloWorld类。它继承自node::ObjectWrap类 ，这个类提供了几个如引用计数、在V8内部传递contex等的实用功能。一般来说，所有对象应该继承ObjectWrap：

```
class HelloWorld: ObjectWrap
{
private:
  int m_count;
public:
```

声明类之后，我们定义了一个静态成员函数，用来初始化对象并将其导入Node.js提供的target对象中。设个函数基本上是告诉Node.js和V8你的类是如何创建的，和它将包含什么方法：

```
static Persistent<FunctionTemplate> s_ct;
  static void Init(Handle<Object> target)
  {
    HandleScope scope;

    Local<FunctionTemplate> t = FunctionTemplate::New(New);

    s_ct = Persistent<FunctionTemplate>::New(t);
    s_ct->InstanceTemplate()->SetInternalFieldCount(1);
    s_ct->SetClassName(String::NewSymbol("HelloWorld"));

    NODE_SET_PROTOTYPE_METHOD(s_ct, "hello", Hello);

    target->Set(String::NewSymbol("HelloWorld"),
```

```
                s_ct->GetFunction());
}
```

在上面这个函数中 target 参数将是模块对象，即你的扩展将要载入的地方。（译著：这个函数将你的对象及其方法连接到这个模块对象，以便外界可以访问）首先我们为 New 方法创建一个 FunctionTemplate，将于稍后解释。我们还为该对象添加一个内部字段，并命名为 HelloWorld。然后使用 NODE_SET_PROTOTYPE_METHOD 宏将 hello 方法绑定到该对象。最后，一旦我们建立好这个函数模板后，将他分配给 target 对象的 HelloWorld 属性，将类暴露给用户。

接下来的部分是一个标准的 C++构造函数：

```
HelloWorld() :
    m_count(0)
{
}

~HelloWorld()
{
}
```

接下来，在::New 方法中 V8 引擎将调用这个简单的 C++构造函数：

```
static Handle<Value> New(const Arguments& args)
{
  HandleScope scope;
  HelloWorld* hw = new HelloWorld();
  hw->Wrap(args.This());
  return args.This();
}
```

此段代码相当于上面 Javascript 代码中使用的构造函数。它调用 new HelloWorld 创造了一个普通的 C++对象，然后调用从 ObjectWrap 继承的 Wrap 方法， 它将一个 C++HelloWorld 类的引用保存到 args.This()的值中。在包装完成后返回 args.This()，整个函数的行为和 javascript 中的 new 运算符类似，返回 this 指向的对象。

现在我们已经建立了对象，下面介绍在 Init 函数中被绑定到 hello 的函数：

```
static Handle<Value> Hello(const Arguments& args)
{
  HandleScope scope;
  HelloWorld* hw = ObjectWrap::Unwrap<HelloWorld>(args.This());
  hw->m_count++;
```

```
    Local<String> result = String::New("Hello World");
    return scope.Close(result);
  }
```

函数中首先使用 ObjectWrap 模板的方法提取出指向 HelloWorld 类的指针，然后和 javascript 版本的 HelloWorld 一样递增计数器。我们新建一个内容为“HelloWorld”的 v8 字符串对象，然后在关闭本地作用域的时候返回这个字符串。

上面的代码实际上只是针对 v8 的接口，最终我们还需要让 Node.js 知道如何动态加载我们的代码。为了使 Node.js 的扩展可以在执行时从动态链接库加载，需要有一个 dlsym 函数可以识别的符号，所以执行编写如下代码：

```
extern "C" {
  static void init (Handle<Object> target)
  {
    HelloWorld::Init(target);
  }

  NODE_MODULE(helloworld, init);
}
```

由于 c++的符号命名规则，我们使用 extern C，以便该符号可以被 dysym 识别。init 方法是 Node.js 加载模块后第一个调用的函数，如果你有多个类型，请全部在这里初始化。NODE_MODULE 宏用来填充一个用于存储模块信息的结构体，存储的信息如模块使用的 API 版本。这些信息可以用来防止未来因 API 不兼容导致的崩溃。

到此，我们已经完成了一个可用的 C++ NodeJS 扩展。

Node.js 也提供了一个用于构建模块的简单工具：node-waf 首先编写一个包含扩展编译方法的 wscript 文件，然后执行 node-waf configure && node-waf build 完成模块的编译和链接工作。对于这个 helloworld 的例子来说，wscript 内容如下：

```
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.cxxflags = ["-g", "-D_FILE_OFFSET_BITS=64", "-D_LARGEFILE_SOURCE",
"-Wall"]
  obj.target = "helloworld"
  obj.source = "helloworld.cc"
```

异步 IO 的 HelloWorld

对于实际的应用来说，HelloWorld 的示例太过简单了一些，Node.js 主要的优势是提供异步 IO。Node.js 内部通过 libeio 将会产生阻塞的操作全都放入线程池中执行。如果需要和遗留的 c 库交互，通常需要使用异步 IO 来为 javascript 代码提供回调接口。

通常的模式是提供一个回调，在异步操作完成时被调用——你可以在整个 Node.js 的 API 中看到这种模式。Node.js 的 filesystem 模块提供了一个很好的例子，其中大多数的函数都在操作完成后通过调用回调函数来传递数据。和许多传统的 GUI 框架一样，Node.js 只在主线程中执行 JavaScript，因此主线程以外的任何操作都不应该直接和 V8 或 Javascript 交互。

同样 helloworld_eio.cc 源代码在 GitHub 上 。我只强调和原来 HelloWorld 之间的差异，其中大部分代码保持不变，变化集中在 Hello 方法中：

```
static Handle<Value> Hello(const Arguments& args)
  {
    HandleScope scope;

    REQ_FUN_ARG(0, cb);

    HelloWorldEio* hw =
ObjectWrap::Unwrap<HelloWorldEio>(args.This());
```

在 Hello 函数的入口处 ，我们使用宏从参数列表的第一个位置获取回调函数，在下一节中将详细介绍。然后，我们使用相同的 Unwarp 方法提取指向类对象的指针。

```
    hello_baton_t *baton = new hello_baton_t();
    baton->hw = hw;
    baton->increment_by = 2;
    baton->sleep_for = 1;
    baton->cb = Persistent<Function>::New(cb);
```

这里我们创建一个 baton 结构，并将各种参数保存在里面。请注意，我们为回调函数创建了一个永久引用，因为我们想要在超出当前函数作用域的地方使用它。如果不这么做，在本函数结束后将无法再调用回调函数。

```
    hw->Ref();

    eio_custom(EIO_Hello, EIO_PRI_DEFAULT, EIO_AfterHello, baton);
    ev_ref(EV_DEFAULT_UC);

    return Undefined();
  }
```

如下代码是真正的重点。首先，我们增加 HelloWorld 对象的引用计数，这样在其他线程执行的时候他就不会被回收。函数 eio_custom 接受两个函数指针作为参数。EIO_Hello 函数将在线程池中执行，然后 EIO_AfterHello 函数将回到在“主线程”中执行。我们的 baton 结构也被传递进各函数，这些函数可以使用 baton 结构中的数据完成相关的操作。同时，我们也增加 event loop 的引用。这很重要，因为如果 event loop 无事可做，Node.js 就会退出。最终，函数返回 Undefined，因为真正的工作将在其他线程中完成。

```
static int EIO_Hello(eio_req *req)
  {
    hello_baton_t *baton = static_cast<hello_baton_t *>(req->data);

    sleep(baton->sleep_for);

    baton->hw->m_count += baton->increment_by;

    return 0;
  }
```

这个回调函数将在 libeio 管理的线程中执行。首先，解析出 baton 结构，这样可以访问之前设置的各种参数。然后 sheep baton->sleep_for 秒，这么做是安全的，因为这个函数运行在独立的线程中并不会阻塞主线程中 javascript 的执行。然后我们的增计数器，在实际的系统中，这些操作通常需要使用 Lock/Mutex 进行同步。

当上述方法返回后，libeio 将会通知主线程它需要在主线成上执行代码，此时 EIO_AfterHello 将会被调用。

```
static int EIO_AfterHello(eio_req *req)
  {
    HandleScope scope;
    hello_baton_t *baton = static_cast<hello_baton_t *>(req->data);
    ev_unref(EV_DEFAULT_UC);
    baton->hw->Unref();
```

进度此函数时，我们提取出 baton 结构，删除事件循环的引用，并减少 HelloWorld 对象的引用。

```
    Local<Value> argv[1];

    argv[0] = String::New("Hello World");

    TryCatch try_catch;

    baton->cb->Call(Context::GetCurrent()->Global(), 1, argv);

    if (try_catch.HasCaught()) {
      FatalException(try_catch);
    }
```

新建要传递给回调函数的字符串参数，并放入字符串数组中。然后我们调用回调传递一个参数，并检测可能抛出的异常。

```
    baton->cb.Dispose();

    delete baton;
    return 0;
  }
```

在执行过回调之后，应该销毁持久引用，然后删除之前创建的 baton 结构。

最后，你可以使用如下形式在 Javascript 中使用该模块：

```
var helloeio = require('./helloworld_eio');
hi = new helloeio.HelloWorldEio();
hi.hello(function(data){
  console.log(data);
});
```

参数传递与解析

除了 HelloWorld 之外，你还需要理解最后一个问题：参数的处理。在 helloWorld EIO 例子中，我们使用一个 REQ_FUN_ARG 宏，然我们看看这个宏到底都做些什么。

```
#define REQ_FUN_ARG(I, VAR)
\
  if (args.Length() <= (I) || !args[I]->IsFunction())
\
    return
ThrowException(Exception::TypeError(                          \
                  String::New("Argument " #I " must be a function")));
\
```

```
Local<Function> VAR = Local<Function>::Cast(args[I]);
```

就像 Javascript 中的 argument 变量，v8 使用数组传递所有的参数。由于没有严格的类型限制，所以传递给函数的参数数目可能和期待的不同。为了对用户友好，使用如下的宏检测一下参数数组的长度并判断参数是否是正确的类型。如果传递了错误的参数类型，该宏将会抛出 TypeError 异常。为简化参数的解析，目前为止大多数的 Node.js 扩展都有一些本地作用域内的宏，用于特定类型参数的检测。

二、揭秘 node. js 事件

要使用 NodeJS，你需要知道一个重要的东西：事件（events）。Node 中有很多对象都可以触发事件，Node 的文档中有很多示例。但文档也许并不能清晰的讲解如何编写自定义事件以及监听函数。对于一些简单的程序你可以不使用自定义事件，但这样很难应对复杂的应用。那么如何编写自定义事件？首先需要了解的是在 node. js 中的’ events’ 模块。

快速概览

要访问此模块，只需使用如下语句：

```
require( ‘events’ )
requires( ‘events’ ).EventEmitter
```

特别说明，node 中所有能触发事件的对象基本上都是后者的实例。让我们创建一个简单的演示程序 Dummy：

dummy. js

view plaincopy to clipboardprint?

```
1.  // basic imports
2.  var events = require('events');
3.
4.  // for us to do a require later
5.  module.exports = Dummy;
6.
7.  function Dummy() {
```

```
8.      events.EventEmitter.call(this);
9.  }
10.
11. // inherit events.EventEmitter
12. Dummy.super_ = events.EventEmitter;
13. Dummy.prototype = Object.create(events.EventEmitter.prototype, {
14.     constructor: {
15.         value: Dummy,
16.         enumerable: false
17.     }
18. });
```

上述代码中重点展示如何使用 EventEmitter 扩充对象，并从中继承所有的原型对象，方法…等等。

现在，我们假设 Dummy 有一个 cooking()的方法，一旦把食物做熟之后它会触发’cooked’事件，并调用一个名为’eat’的回调函数。

dummy-cooking.js

view plaincopy to clipboardprint?

```
1.  Dummy.prototype.cooking = function(chicken) {
2.      var self = this;
3.      self.chicken = chicken;
4.      self.cook = cook(); // assume dummy function that'll do the cooking
5.      self.cook(chicken, function(cooked_chicken) {
6.          self.chicken = cooked_chicken;
7.          self.emit('cooked', self.chicken);
8.      });
9.
10.     return self;
11. }
```

现在，这个模块已经完成了。我们可以在主程序中使用它。

dummy-node.js

view plaincopy to clipboardprint?

```
1.  // A nonsensical node.js program
2.
3.  var Dummy = require('./dummy');
4.  var kenny = new Dummy();
5.  var dinner = kenny.cooking(fried_chix);
```

```
6.  dinner.on('cooked', function(chicken) {
7.      // eat up!
8.  }>
```

所以基本上，node.js 执行脚本，然后等待’cooked’事件被触发，并在事件触发之后调用回调函数并传递返回的参数。

还有什么要注意的

值得注意的是，例子中使用的“子类”和事件有一些极端（a bit of an overkill）。EventEmitter 每次只触发一个事件(EventEmitter for things that only fire one event once)。如果只创造少数几个实例，可以将方法直接加入到实例本身，如果要触发底层事件，可能实用异步函数会更好一些。

关于 events.EventEmitter，你还需注意一个特别的事件：’error’。任何错误发生时此事件都会触发，并且当没有监听程序监听这个事件时，node 将会抛出异常并结束应用程序。（感谢 Tim 指出这一点）

结束之前的话

这篇文章简要介绍了一些好的方法（good runthrough）。在发表之前文章之前，我查阅了我最喜爱的图书馆，参考别人的实现，并且还得到了别人的帮助弄把一些问题搞清楚。为了更好的理解 node，我建议你阅读 Time Caswell 的文章：

Node 中的流程控制（2）
What is “this”？

同时，你也可以看一下我用 node.js 写的机器人混搭应用（bot mashup）tocho，它使用了 ircbot、logger、real-time websockets 技术并支持搜索，但是处于开发的前期还比较粗糙。编写这个程序给我带来了很多乐趣。我还没有将搜索功能发布到演示网站，您以使用支持 web socket 的浏览器访问实时聊天室 #node.js。

简单地说，NodeJS 是一个使用了 Google 高性能 V8 引擎 的服务器端 JavaScript 实现。它提供了一个（几乎）完全非阻塞 I/O 栈，与 JavaScript 提供的闭包和匿名函数相结合，使之成为编写高吞吐量网络服务程序的优秀平台。在我们内部，雅虎邮件队正调研能否使用 NodeJS 开发一些我们即将推出的新服务。我们认为分享我们的劳动成果是一件十分有意义的事情。

三、在多处理器系统上使用 NodeJS 的情况

NodeJS 中并不是完美无缺的。虽然单进程的性能表现相当不错，但一个 CPU 最终还是不够用（由于 JS 引擎自身的运行原理，NodeJS 使用单线程执行 JS 代码，详见“JS 和多线程”）。Node 本身并没有扩展能力来充分利用多 CPU 系统的计算能力。实际上当前版本的 NodeJS 程序只能在一个上 CPU 执行，在 2.5GHz 的英特尔至强处理器下运行 HTTP 代理服务的性能大约为 2100 reqs/s。

虽然 Node 相对稳定，但它仍然会偶尔崩溃。如果你实用一个单独的 NodeJS 进程作为服务，崩溃会对可用性造成不良影响。例如段错误、内存越界等错误在用 C++编写的程序上相当普遍。如果有多个进程同事处理请求，当一个进程出错退出，传入的请求可以被导向给其他进程。

充分利用多处理器的优势

有如下几种方法可以使 NodeJS 利用多处理器，每个方法都有自己的优缺点。

使用软件负载均衡

直到 node-v0.1.98 ，充分利用多处理器的最佳做法是为每个处理器单独启动一个 NodeJS 进程，每个进程都运行 HTTP 服务并绑定到不同的端口。这样需要一个负载均衡软件，将客户端请求转发到各进程，这个软件知道每个服务进程的端口。这样处理性能也不错，但配置管理多进程比较复杂，因此不是最佳方案。

当然，这种架构也有好处，它允许负载均衡软件按照指定的策略将请求路由到不同进程上。（例如，通过 IP，通过 cookie 等）。

使用操作系统内核做负载均衡

在 node-v0.1.98 中 ，雅虎贡献了一个用于传递和重用文件描述符的核心补丁，允许如 Connect 和 multi-node 等 HTTP 框架使用多个进程同时提供 HTTP 服务，而且不需要修改原有的程序代码和配置。

概括地讲，这些框架使用的方法是创建一个进程监听端口（比如说监听 80 端口）。然而，这个进程不是接受 Socket 连接，而是使用 net.Stream.write()将其传递给了其他子进程（其内部是使用 sendmsg(2)发送，并使用 recvmsg 来获取文件表描述符）。每个子进程排队将收到的文件描述符插入自己的事件循环中并在空闲时处理客户端的连接。OS 内核本身负责进程间的负载平衡。

重要的是，这实际上是一个高效但没有策略的 L4 负载平衡器，每个客户端的请求可能被任意一个进程处理。任何处理请求所需的应用程序的状态，都不能像单进程时那样简单的保存在一个 NodeJS 实例当中。

使用 NodeJS 转发请求

某些情况下，你可能可能无法使用或者不想使用上述两种方法。例如，负载均衡程序无法按照应用程序所需的路由规则转发请求（如，有复杂应用逻辑的路由规则或者需要 SELinux 连接信息的路由规则）。在这种情况下，可以使用单个进程接受连接，检查并传递给其他进程处理。

下面的例子需要 node-v0.1.100 或更高版本以及 node-webworker 。node-webworker 是新兴的 HTML5 Web Workers 标准的 NodeJS 实现，这个标准允许并行执行 JavaScript 代码。您可以使用 npm 安装 node-webworker，命令如下 npm install webworker@stable。

详细介绍 Web Workers 的原理超出了这篇文章的范围，你可以认为 Web Worker 是一个独立的执行上下文（类似进程），它可以由 JavaScript 代码生成并来回传递数据。node-webworker 允许使用如下消息传递机制传递文件描述符：

首先，主进程的源代码 master.js：


```
1.  var net = require('net');
2.  var path = require('path');
3.  var sys = require('sys');
4.  var Worker = require('webworker/webworker').Worker;
5.
6.  var NUM_WORKERS = 5;
7.
8.  var workers = [];
9.   var numReqs = 0;
10.
11. for (var i = 0; i < NUM_WORKERS; i++) {
12.  workers[i] = new Worker(path.join(__dirname, 'worker.js'));
13.  }
14.
15. net.createServer(function(s) {
16.  s.pause();
17.
18.  var hv = 0;
19.  s.remoteAddress.split('.').forEach(function(v) {
20.  hv += parseInt(v);
21.  });
22.
23.  var wid = hv % NUM_WORKERS;
24.
25.  sys.debug('Request from ' + s.remoteAddress + ' going to worker ' + wid);

26.
```

```
27.   workers[wid].postMessage(++numReqs, s.fd);
28.   }).listen(80);
```

主进程将执行如下操作：

- 主进程将建立 net.Server 实例并在 80 端口上侦听连接请求。
- 当请求到来时，主进程
    - 根据请求端的 IP 地址决定将请求发送至哪一个 wroker。
    - 调用请求流对象的 net.Stream.pause()方法。这可以防止主进程从读取套接字中读取数据 — wroker 进程应该看到远程端发送的所有数据。
    - 使用 postMessage()方法将（递增后的）全局请求计数器和刚刚收到
      套接字描述符发送到指定的 worker

然后，worker 进程的源代码 worker.js：

view plaincopy to clipboardprint?

```
1.  var http = require('http');
2.  var net = require('net');
3.  var sys = require('sys');
4.
5.  process.setuid('nobody');
6.
7.  var srv = http.createServer(function(req, resp) {
8.   resp.writeHead(200, {'Content-Type' : 'text/plain'});
9.   resp.write(
10.  'process=' + process.pid +
11.  '; reqno=' + req.connection.reqNo + '\n'
12.  );
13.  resp.end();
14.  });
15.
16. onmessage = function(msg) {
17.  var s = new net.Stream(msg.fd);
18.  s.type = srv.type;
19.  s.server = srv;
20.  s.resume();
21.
22.  s.reqNo = msg.data;
23.
24.  srv.emit('connection', s);
25.  };
```

worker 执行如下操作：

- 将自己的权限降为 nobody 用户。
- 创建一个 HTTP 服务器实例但并不调用任何 listen()方法。我们将通过主进程收到的描述符来传递请求。
- 等待从主进程接收套接字描述符和相关信息
- 将从主进程收到的请求计数保存进流对象(stream.object)中，代码有些乱，但让我们可以使用 HTTP 相关的类来处理这些数据。
- 将 net.Stream 实例和收到的 TCP 链接接合，然后通过手动触发事件将其融入 HTTP 请求的处理流程中。
- 现在，我们如上建立的请求处理程序可以正常运行了： HTTP 服务实例完全拥有连接并将像平常一样解理客户端的请求。注意一个小技巧，请求处理程序访问流对象的 reqNo 属性，并根据主进程中的计数变量(既用于记录请求数的全局变量 numReqs)将其设置为实际的请求数。

最后，一定要使用超级用户执行 master.js，因为我们希望程序监听特权端口(80)。然后使用 curl 发出一些请求，并看看是那个进程处理这些请求。

```
% sudo node ./master.js
% curl 'http://localhost:80'
process=13049; reqno=2
```

当然，前面例子用到的基于 IP 的哈希算法是玩具级的，任何一个合格的 HTTP 负载均衡器能可以实现。在现实中，你可能想根据客户端的请求，将连接分派到运行在正确的 SELinux 上下文中的 worker。（参见，node-selinux）根据 HTTP 请求本身的的信息（如：path，vhost）作出路由决策稍微复杂些，且使用类似的技术也可行。

结论

最 后，我希望本文能够说明当前 NodeJS 利用多处理器的情况：一些现有的 HTTP 框架可以给各种 NodeJS 应用提供多处理器支持；node- webworkers 为管理 NodeJS 中的并行机制提供了一个好方法(基于 chlid_proess)；怎样实用 NodeJS 自身实现 L7 HTTP 路由器。