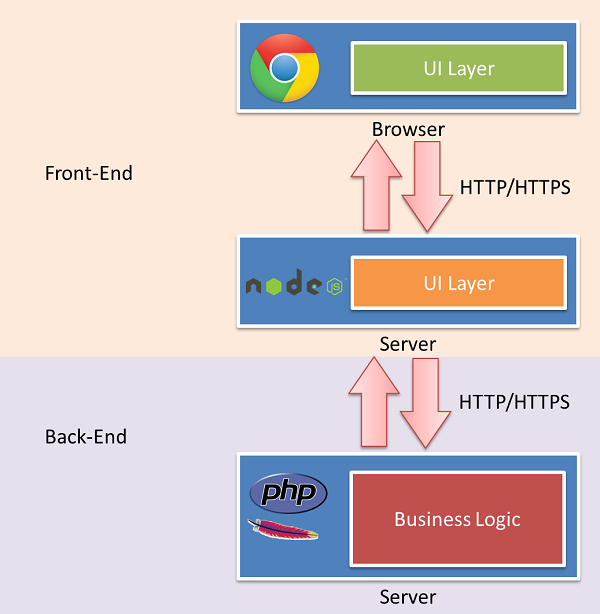
# 用nodejs做Web前端服务器的一些经验

前不久 NCZ 发表了新文章 [Node.js and the new web front-end](http://www.nczonline.net/blog/2013/10/07/node-js-and-the-new-web-front-end/)（[译文](http://www.silverna.org/blog/?p=297)），描述了用 node.js 做 Web 前端服务器的种种优势。NCZ 在文章中推荐了一套服务器模型(图片来源自[Node.js and the new web front-end](http://www.nczonline.net/blog/2013/10/07/node-js-and-the-new-web-front-end/))。



这个模型在传统的后台服务器前，增加了一层 node.js 实现的 Frontend Server 层。这种架构的最大好处是前后端开发人员的依赖分离，让后端开发人员不必再关心数据在页面间如何传递、用户数据获取是通过 Ajax 还是刷新页面等前端开发所涉及的方方面面，前端开发人员也不必再关心数据如何在数据库中存储等等后端问题。

The front-end and back-end now have a perfect split of concerns amongst the engineers who are working on those parts. The front-end has expanded back onto the server where the Node.js UI layer now exists, and the rest of the stack remains the realm of back-end engineers. – Nicholas C. Zakas

碰巧前不久，我在公司内部尝试了这种架构，这里正好分享一些 node.js 做 Web 前端服务器的经验。

## 与后台服务器的交互

在用户的一次请求中，往往需要请求多个不同的后台接口。由于 node.js 的异步特性，写多次 HTTP 请求并处理回调是一件非常痛苦的事情，例如

|  |
| --- |
| var request = require('request');  exports.index = function (req, res) {  request('API\_A', function (err, response, body) {  if (err) {  // ...  }  request('API\_B', function (err, response, body) {  if (err) {  // ...  }  request('API\_C', function (err, response, body) {  if (err) {  // ...  }  // ...  });  });  });  }; |

这种情况通过 [async] 库可以很很好的解决这个问题。[async] 是一个工具包，提供了各种各样的小函数来简化 node.js 的异步回调处理。

|  |
| --- |
| var request = require('request');  var async = require('async');  exports.index = function (req, res) {  async.map(['API\_A', 'API\_B', 'API\_C', /\* ... \*/], request, function (err, results) {  if (err) {  // ...  }  var resultA = results[0];  var resultB = results[1];  var resultC = results[2];  // ...  });  }; |

通过 async.map 可以很轻易的实现并行请求数据。如果需要串行请求数据，可以使用 async.Series 函数。除此之外，还可以使用 async.mapLimit 来限制 node.js 的并发连接数。

### 常用 API 数据的获取

有些 API 数据是几乎每个页面都会用到的，例如当前用户的个人信息等。对于这类数据，可以通过 middleware 的方式来将它传递给 controller。

|  |
| --- |
| var request = require('request');  var async = require('async');  function userdata (req, res, next) {  request('GET\_USER\_API', function (err, response, body) {  if (err) {  next(err);  return;  }  req.user = JSON.parse(body);  next();  });  }  app.get('/pageA', userdata, pageAController);  app.get('/pageB', userdata, pageBController);  app.get('/pageC', userdata, pageCController); |

### Cookie 代理

如果 API 接口需要验证 Cookie，那么 node.js 在发送 API 请求时，需要将用户的 Cookie 信息发到后台服务器。同样的，如果后台 API 接口修改了用户 Cookie，例如登陆 API，那么还需要 node.js 将设置用户 Cookie 的请求转发给用户。这就需要实现一个 cookieRequest 方法。

|  |
| --- |
| var request = require('request');  var cookieRequest = function (userRequest, userResponse, url, callback) {  var options = {  url: url,  headers: {}  };  options.headers.Cookie = userRequest.header('Cookie'); // 将用户的 Cookie 传递给后台服务器  request(options, function (error, response, body) {  userResponse.setHeader('Cookie', response.headers.cookie);  callback.apply(null, arguments);  });  }; |

## 多核优化

由于 node.js 的单进程特性，只启动一个 node.js 实例的话不能充分发挥多核 CPU 的性能。因此 node.js 提供了 cluster 模块来解决这个问题。cluster 可以管理多个服务器进程，充分发挥多核 CPU 的性能。

用法很简单，只需要创建一个新的启动脚本，调用 cluster 模块来启动服务即可，node.js 官方的 API 文档给了一个很简单的例子：

|  |
| --- |
| var cluster = require('cluster');  var http = require('http');  var numCPUs = require('os').cpus().length;  if (cluster.isMaster) {  // Fork workers.  for (var i = 0; i < numCPUs; i++) {  cluster.fork();  }  cluster.on('exit', function(worker, code, signal) {  console.log('worker ' + worker.process.pid + ' died');  });  } else {  // Workers can share any TCP connection  // In this case its a HTTP server  http.createServer(function(req, res) {  res.writeHead(200);  res.end("hello world\n");  }).listen(8000);  } |

## 性能

老的 Web 服务器使用的是 tomcat + java 的架构，用 node.js 重写整个前端层之后，整个服务的性能提升了不少。目前这个项目只是个人娱乐，所以也没有做太专业的性能测试，只是用 ab 随便打了打压力，在我的 iMac(2.7G i5, 12G) 上大约有 20% 的性能提升。这和 node.js HTTP 模块的高性能以及并发请求 API 不无关系。

## 结语

随着项目规模的扩展，Web 前端服务器与后端服务器分离是一个不可避免的趋势。而 node.js 提供了一套对前端开发人员更加友好的 Web 前端服务器方案，这一方案将前后端开发人员从彼此不擅长的领域中解救出来，降低了沟通成本，对于提升开发效率有着非常大的帮助。