
目錄

Introduction	1.1
前言	1.2
1 Node.js 体系结构	1.3
2 JavaScript 那些事	1.4
3 Node 基础	1.5
4 NPM 包管理工具	1.6
5 Node 中使用数据	1.7
6 Express 介绍	1.8
7 Express 进阶	1.9
8 Node.js 单元测试	1.10
9 Node.js 最佳实践	1.11
10 Node.js 的 c++ 扩展	1.12
11 Node.js 调优	1.13
12 Node.js Web 安全	1.14
A1 Node.js 好用工具介绍	1.15
A2 参考资源	1.16
A3 书籍写作规范	1.17

关于本书

这是一本关于 Node.js 技术的开放源码电子书，受到台湾nodejs社区提供（源码参见[github地址](#)）启发编写。感谢台湾nodejs社区的前辈提供这么优秀的教程，不过台湾社区的教程已经停止了维护。为了让大家学习到这么优秀的教程，我决定将其复活，所以才有了这个项目。由于原书写作的时候Node版本是0.6，所以在写作本书的时候，借鉴了原书的目录结构，但是内容上进行了重写。本书的线上阅读网址，与 GitHub 资料同步更新。

<http://nodebook.whyun.com/>

本书适合 Node.js初学者至进阶开发者，也欢迎您在学习时一起参与本书内容撰写。

授权

本书原采用创用CC姓名标示-非商业性授权。您不必为本书付费。

Node.js Wiki Book book is licensed under the Attribution-NonCommercial 3.0 Unported license. **You should not have paid for this book.**

您可以复制、散布及修改本书内容，但请勿将本书用于商业用途。

您可以在以下网址取得授权条款全文。

<http://creativecommons.org/licenses/by-nc/3.0/legalcode>

作者

- yunnysunny <https://github.com/yunnysunny>

源码

本书最新的源码网址如下：

<http://github.com/yunnysunny/nodebook>

编译

本书采用 [gitbook](#) 进行编译，如果你想在本机编译，需要安装 `gitbook-cli`：

```
npm install gitbook-cli -g
```

然后在项目目录中运行 `gitbook serve` 即可编译生成html，并且创建一个本地预览网站，其他详细的编译参数可以参照[官方文档](#)。

前言

Node.js 是 JavaScript 后端开发语言。从诞生之初就备受关注，到如今说到最火的后端 Web 开发，Node 说自己是第二，没有人敢说他是第一。正是 Node 的兴起，还带动了前端 JS 的热度，react.js vue.js 这些前端 JS 也火借东风，烧的很旺，甚至连“全栈”这个词也悄然登上了招聘启事的热搜关键字。也许我能够做的就是在这把大火里再添几把柴。

声明

首先声明本电子书并不是本人创作，nodejs台湾协作电子书（源码参见[github地址](#)），是[台湾nodejs社区](#)发起的一套nodejs入门教程，本人初学nodejs之时，受其启发颇深。现在台湾社区的兄弟们对其已经停止维护，想想这么优秀的教程就这样白白流逝掉，是一件让人惋惜的事情的事情，于是乎我利用业余时间将其复活。

但是在整理的过程中发现了若干问题，首先繁体中文和简体中文的语言使用上还是有差别，之前耳熟能详的词语到了繁体中文中就变了个叫法，搞的自己差点摸不着头脑；其次，当时书籍写作的时候 Node 的版本还是0.6，如今时过境迁 Node 连6.0都出来了，好多知识点也发生了变化；最后，我也对原书的有些章节安排不是很满意。正是以上三点原因促成了我重写此书的决定。

线上预览地址：<http://nodebook.whyun.com>

本书github地址：<https://github.com/yunnysunny/nodebook>

同时欢迎大家提交pull request来完善文档内容。

电子书下载

本书同时提供多种格式的电子书供下载：

- [pdf格式下载](#)
- [epub格式下载](#)
- [mobi格式下载](#)

阅读指引

- 如果你之前有socket和多线程的编程经验，可以看第1章，没有上述经验的编程者也可以阅读此章，只不过在阅读过程中可能会遇到概念一时半会儿理解不了。
- 如果之前没有接触过 javascript 这门语言，可以从第2章开始阅读，第2章讲述

了一些 javascript 的基础语法。

- 如果之前接触过 javascript ，但是没有接触过 Node.js ，可以从第3章开始阅读，第3章讲述了一些 Node.js 的基础 API 。
- 如果之前接触过 Node.js ，但是没有用过数据库操作，可以阅读第5章，第5章讲述了 Node 中操作 redis 、mongodb 的 API 如何使用；如果没有使用 express ，可以阅读第6章和第7章，这两章讲述了如何利用 express 这个 HTTP 编程框架。
- 最后介绍一下一些独立章节，第4章讲述了 npm 命令的使用教程；第8章讲述了如何使用单元测试框架mocha；第9章讲述了一些线上环境的最佳实践，包括配置文件、进程管理、 docker 等内容；第10章讲述如何编写 c++ 扩展。

1 Node.js 体系结构

其实我就是想写一下 Node 的底层架构，但是说道底层这个东西，我就想起了我上学时候的一门课《计算机体系机构》，就是把计算机各个部件的运行原理给串起来来讲，所以我就把这章的名字定为 Node.js 体系结构，但愿讲得够底层。

1.1 网络 IO 模型变迁

Node.js 是一门服务器语言，为了体现 Node 的优越性，我们这里不得不扯一下服务器开发的一些历史。

我们最常见的服务器程序一般是基于 HTTP 和 TCP 协议来提供服务的，HTTP 底层又基于 TCP，所以我们直接来描述 TCP 协议在服务器端实现的逻辑。在操作系统中 TCP 的通信过程又被称之为网络 IO 操作，下面描述的就是网络 IO 操作的简史。

TCP 服务在建立完 socket 监听后，会调用 `accept()` 函数来监听客户端的连接请求，但是这个过程是堵塞的。也就是说如果函数没有返回，当前线程会一直等待，而且在这个等待的过程中无法做任何事情。伪代码如下：

```
while(true) {  
    socket = accept();  
}
```

代码 1.1.1 socket 建立连接伪代码

代码 1.1 中我们通过 `accept` 函数，服务器和客户端之间建立了一个 socket 连接，建立完连接之后，就可以开始发送接收数据的操作。但是当程序运行到 `accept` 函数的时候，是堵塞的，也就是说这个函数不运行完成，代码是没法继续运行的。假设我们现在 `accept` 函数返回了，那么我们就可以读取这个连接发送过来的请求数据了：

```
while(true) {
    socket = accept();
    while(true) {
        data = read(socket);
        //process_data(data); //处理数据
    }
}
```

代码 1.1.2 读取socket数据伪代码

不过和 `accept` 一样，这个 `read` 函数依然是堵塞的。照这个趋势下去，一个服务器只能给一个连接做服务了，其他的连接就干等着。这可不是我们想要的结果。

我们的前辈们想到的解决方案是fork子进程，每次跟客户端建立一个连接，都创建一个新的子进程来维护当前连接，在这个新的子进程中进行发送和接收数据。这种子进程的模型的典型代表就是 Apache 1.x。我们来看一下伪代码：

```
while(true) { //主进程代码
    socket = accept();
    var child = fork(socket);
}

while(true) { //子进程代码
    data = read(socket);
}
```

代码 1.1.3 子进程读取socket数据伪代码

看上去是一个好的解决方案，各个socket连接在读取数据的时候都是在单独的一个进程中完成的，不会互相堵塞。不过进程的创建是一个耗时的操作，而且操作系统对于启动的最大进程数也是有限制的，如果服务器创建大量线程，有可能导致系统其他进程无法启动(所以一般服务器都会限制启动子进程的最大数目，这个时候在程序里面会维护一个socket队列，来决定那些连接被丢入子进程进行处理。)。这个时候线程便进入了大家的视野，它作为cpu的最小调度单位，具有比进程更少的资源占用，最好的性能。线程有进程创建，对于一个进程来说，它所创建的线程共享其内存数据，且可以被统一管理。由于使用线程的逻辑和使用进程的逻辑类似，所以这里不给出伪代码。Apache 从2.x开始增加了对多线程的支持。

即使使用了线程，但是计算机的CPU每次可以处理的线程数是有限的（单核CPU每次处理一个线程，双核可以同时处理两个，Intel使用超线程技术，可以使一个核心处理两个线程，所以说我们常用的i5处理器，虽然是两核但是却可以同时处理四线程），为了让各个线程公平对待，CPU在单位时间内会切换正在处理的线程。但是这个切换动作是比较耗时的，CPU在将处理的线程任务切换走之前要暂存线程的内存，在切换入一个新的要处理的进程之前要读取之前暂存的线程内存，当然还要考虑到CPU内部还要有一套调度算法，来决定什么时候将线程切换到CPU进行处理。所以说使用线程也会遇到性能瓶颈，不会像我们想的那样，线程数起的越多，性能越好。

不过在操作系统中有非堵塞IO（nonblocking IO）的概念，既然它叫这个名，那么我们前面讲的就应该叫堵塞IO（blocking io）了。我们还是通过类比来解释在读取socket数据时两者的区别，同时看看这个传说中的非堵塞IO能否解决我们的问题。

我们把socket通讯过程类比为你在淘宝上买东西的过程，你在淘宝上下单买了件商品（socket连接建立了）。对于堵塞IO来说，你需要给快递员打电话，并且你还不能挂断，在快递员没有通知你商品到之前，你啥事也不能干。

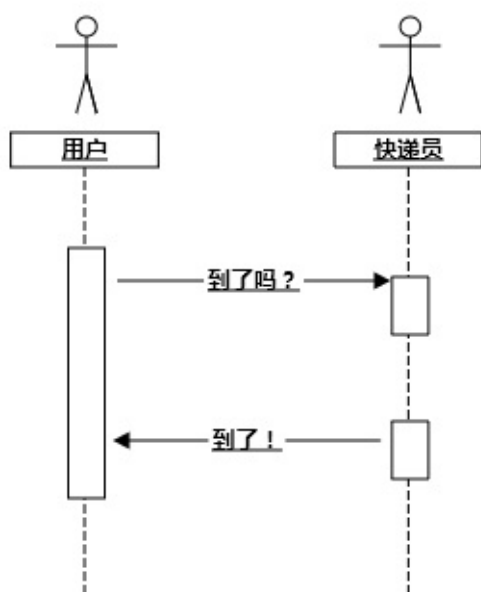


图 1.1 堵塞 IO 类比时序图

我们用一幅图来描述上述过程的话，那他应该如图 1.1 所示，其实在真实场景中，用户就是你的应用程序，而快递员就是你的 Linux 内核。

对于非堵塞 IO 来说，为了了解到包裹是否送达了，你只需要定时给快递员打电话咨询。不过你在打完电话之后还可以忙别的，比如说看会儿书，喝喝茶。然后你想起，我靠还有一个快递呢，于是赶紧再打一个电话，结果发现人家快递员已经在楼下等了半天。所以说，要想尽早得到快递，你得一直跟快递员打电话（俗称呼死他）。

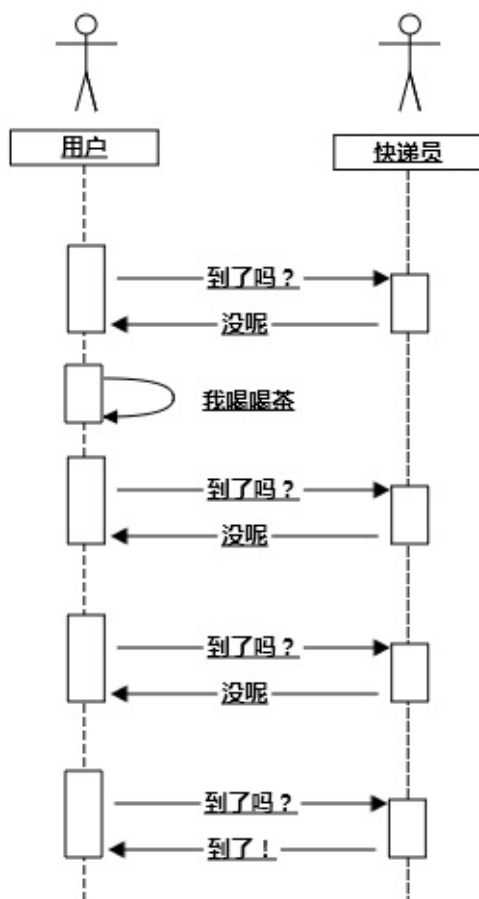


图 1.2 非堵塞 IO 类比时序图

有人问，为什么不是快递员给你打电话，而是你给快递员打电话，首先声明一下，为了简化描述的工作量，我们现在先按照linux操作系统来讲，在linux系统中只能用户去调用内核函数，没有内核函数主动通知用户程序的功能。我们这里内核函数就是快递员，用户程序就是你自己，所以只能你自己打电话给快递员。同时大家需要注意，对于非堵塞 IO 这个定义，有不同的叫法，有的管我们刚才提到的这种方式交非堵塞 IO，但是有的管 IO 多路复用（下面马上讲）叫非堵塞 IO。

OK，下面要轮到我们的 IO 多路复用闪亮登场了。一般你在淘宝上买东西，填写邮寄方式的时候，都是直接写你自己的地址，不过淘宝其实提供了菜鸟驿站这个东西，你可以在不方便的情况下把，把包裹的邮寄地址写成菜鸟驿站。这个样子你的所有快递就都可以由菜鸟驿站来代收了，不过你仍然要打电话询问驿站的工作人员，快递来了吗（因为我们用的是linux，在这个操作系统下，内核是不会主动通知用户程序的，这个步骤在linux中称为事件查询）。

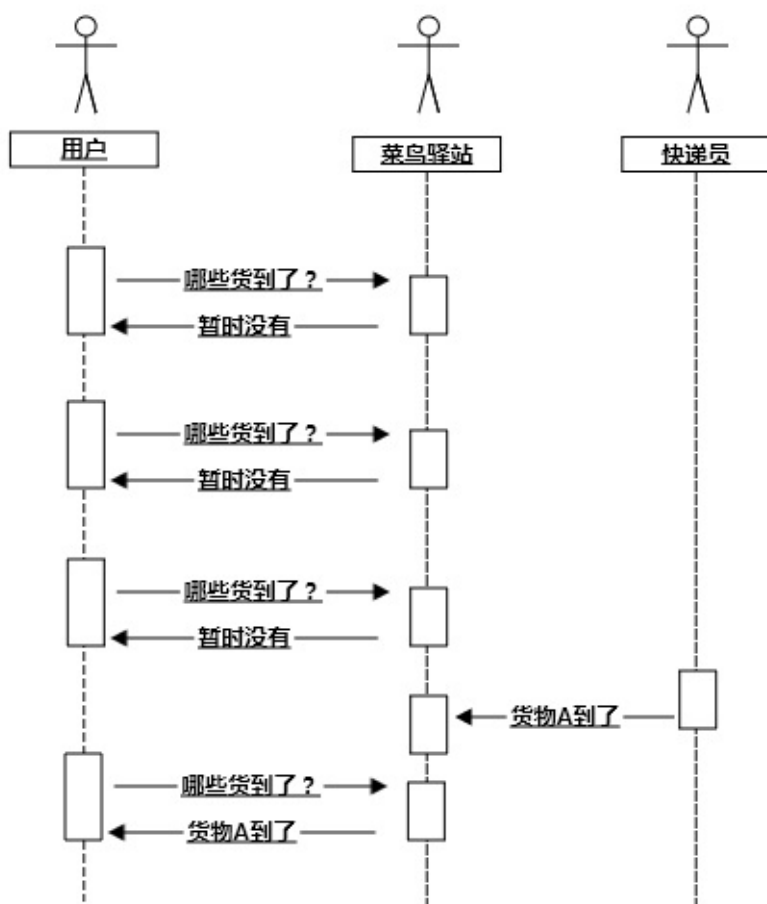


图 1.3 IO 多路复用类比时序图

回到上面的栗子，你可能会问如果单纯一件商品的话，自己直接等快递显然比先送到菜鸟驿站再打电话问要快。是的，没错！但是不要忘了，为了收多件快递，没订购一件商品，都要克隆出另外一个你（fork子进程或者创建线程），来等着收快递，一旦你淘宝上下单量很大，管理这些克隆人的成本就会陡增（主要耗费在进程或线程的上下文切换和调度）。所以说在连接数不大的情况下使用堵塞IO反而效率更高。

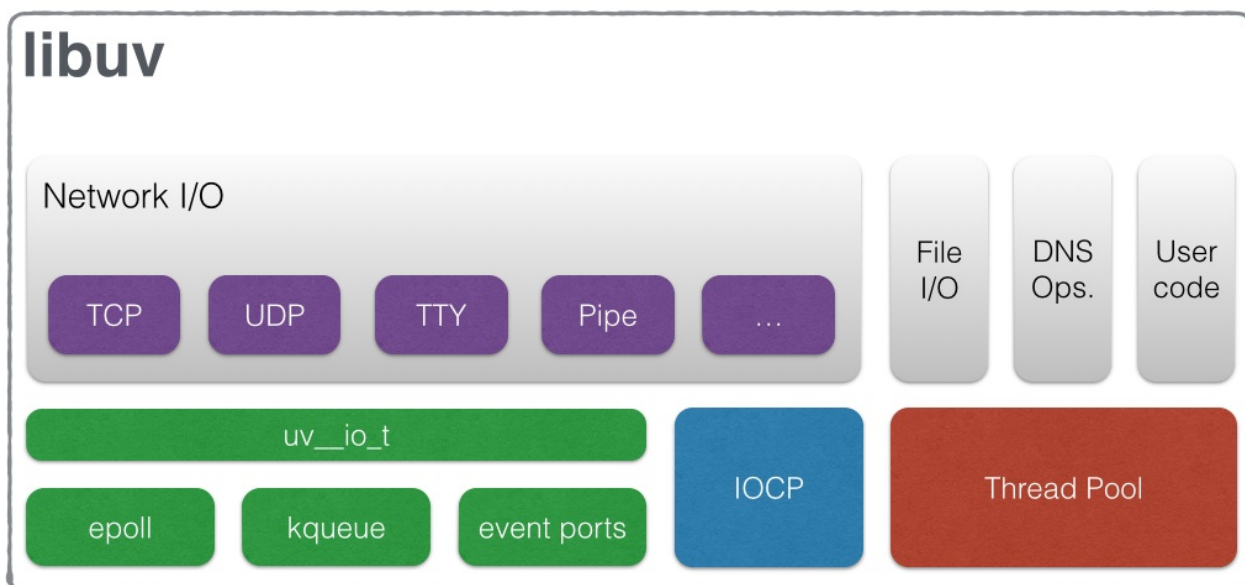
这个IO 多路复用在linux上几经更新，发展到现在，使用的最新技术就是 **epoll**，**nginx**底层就是利用了这个技术。其实通过前面的栗子，我们发现 IO 多路复用中依然有堵塞过程（不断打电话给菜鸟驿站的过程），但是我们在实际编程中可以专门做一个子线程来做打电话的工作（相当于给你请了一个秘书），然后主线程可以该干嘛干嘛。

1.2 libuv

Node 的开发者 Ryan dahl，起初想构建一个可以处理大量HTTP连接的web服务，他知道使用C语言可以实现这个目标，前面章节讲到 IO 多路复用在大量连接数的时候，性能要优于堵塞 IO。但那是C语言开发效率太低了，特别是当你做web开发的时候，当时恰逢08年，谷歌刚推出V8引擎，我们的Ryan dahl 经过各种选型和权衡后，最终选择用C、C++做了一个 IO 处理层，结合V8引擎，组成了 Node。这个 IO 处理层，就是我们现在说到的 libuv。

我们前面的内容是基于 linux 描述的，但是类似于 epoll 的操作，在不同的操作系统实现库函数是不同的，在 windows 上有IOCP，MAC上有kqueue,SunOS上有event ports，这个时候有一个抽象层对外提供统一的 api 是一个好的选择，libuv就解决了这个问题，但是这不是他所有的功能。

libuv的[官方文档](#)在阐述他的架构的时候给出来这么一张图



，但是仅仅凭着这么一张图并不能让你对其内部机制理解得更透彻。

我们知道 node 使用了 V8 引擎，但是在 node 里面 V8 充当的角色更多的是语法解析层面，另外它还充当了 JavaScript 和 c/c++ 的桥梁。但是我们都知道 Node 中一切皆可异步，但这并不是通过 V8 来实现的，充当这个角色的是 libuv。libuv 作为

实现此功能的幕后工作者，一直不显山不露水，今天就要将其请到前台来给大家展示一下。

js 怎样做一个异步代码， `setTimeout` 函数即可搞定：

```
setTimeout(function(){console.log('timeout 0');},0);  
console.log('outter');
```

代码 **1.2.1** 一个简单的js定时器演示

最终输出结果先是打印 `outter` 然后打印 `timeout 0`。

想要深挖为什么会出现这样的结果，要首先来研究一下 libuv 的事件轮询机制。在 libuv 中，有一个句柄（**handle**）的概念，每个句柄中存储数据和回调函数之类的信息，句柄在使用前要添加到对应的队列（**queue**）或者堆（**heap**）中，其实只有定

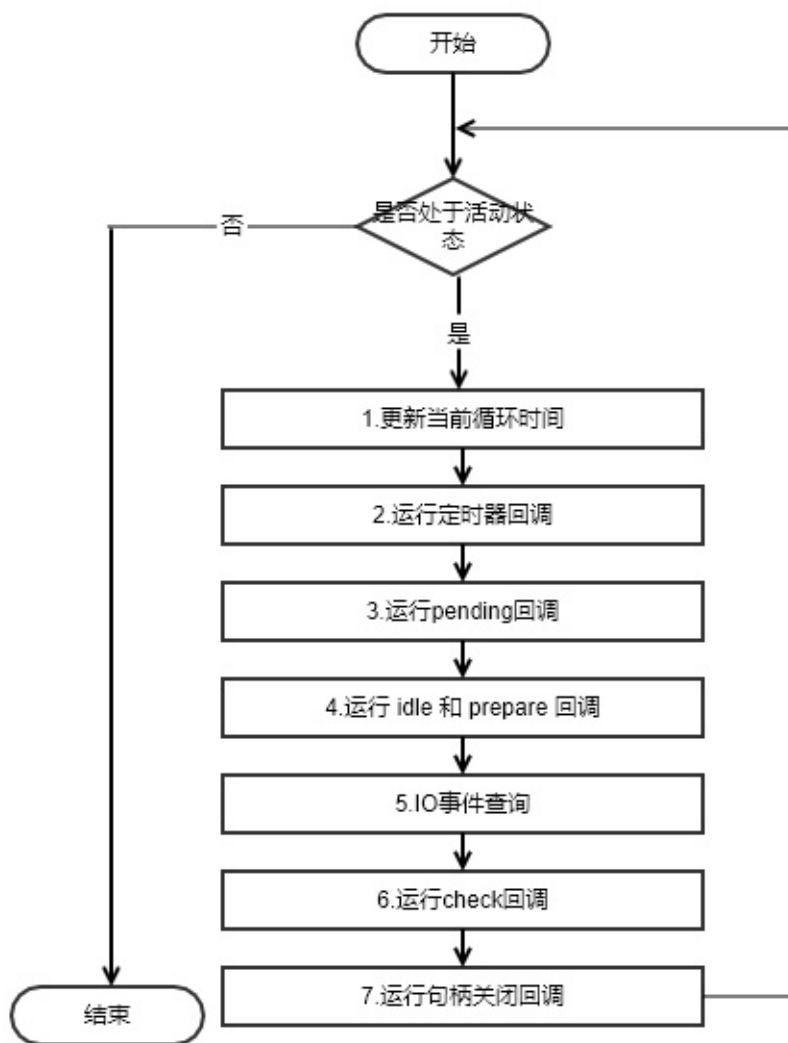


图1.2.1 默认事件轮询流程图

如上图所示，从第2步开始处理事件轮询中的各种类型的堆和队列结构，其中：

- 定时回调，处理 `setTimeout` 和 `setInterval` 的回调。
- pending 回调，处理各种 IO 事件完成的回调函数，不过不包括关闭事件。
- idle 和 prepare 回调，仅仅在内部使用，有兴趣大家可以参见libuv的测试文件 [test_idle.c](#)。
- IO 事件查询，就是前面讲到的检测 IO 多路复用事件的操作。
- check 回调，在 Node.js 中被用作运行 `setImmediate` 回调。
- 句柄关闭回调，用来运行一些类似于 `socket` 句柄关闭的回调函数。

接着，讲述一下libuv的线程模型，因为要想实现一个无堵塞的事件轮询必须依靠线程。libuv 中大体上可以把线程分为两类，一类是事件轮询线程，一类是文件 IO 处理线程。第一类事件轮询线程是单线程；另外一类称其为文件 IO 处理线程多少有

接着，讲述一下libuv的线程模型，因为要想实现一个无堵塞的事件轮询必须依靠线程。libuv 中大体上可以把线程分为两类，一类是事件轮询线程，一类是文件 IO 处理线程。第一类事件轮询线程是单线程；另外一类称其为文件 IO 处理线程多少有些不准确，因为他不仅能处理文件 IO，还能处理 DNS 解析，也能处理用户自己编写的 node 扩展中的逻辑，它是一个线程池，如果你想自己编写一个 c++ 扩展来处理耗时业务的话，就会用上它（我们将在第9章讲c++扩展内容）。

我们这里拿文件IO处理举个栗子，来描述这两类线程之前是怎么通信的。libuv 在处理完一个文件 IO 操作后，会把处理后的结果发送到 pending 队列中；事件轮询线程读取 pending 队列，执行回调函数，也就是图1.2.1中第3步操作。下面是我们演示用的函数：

```
var fs = require('fs');

fs.exists(__filename, function (exists) {
  console.log(exists);
});
```

代码 1.2.3 fs.exists 函数示例

`fs.exists` 是 Node 自带的函数，我们在调用的时候传了两个参数，第一个

`__filename` 是 Node 中的一个全局变量，它的值其实是当前执行文件的所在路径，第二个参数是一个回调函数，回调函数中 `exists` 用来表示当前是否存在，很明显当前这段代码最终打印的结果肯定是 `true`，当然我们这里更关心的是整个流程处理，下面用一副数据流向图来将上面流程总结一下：

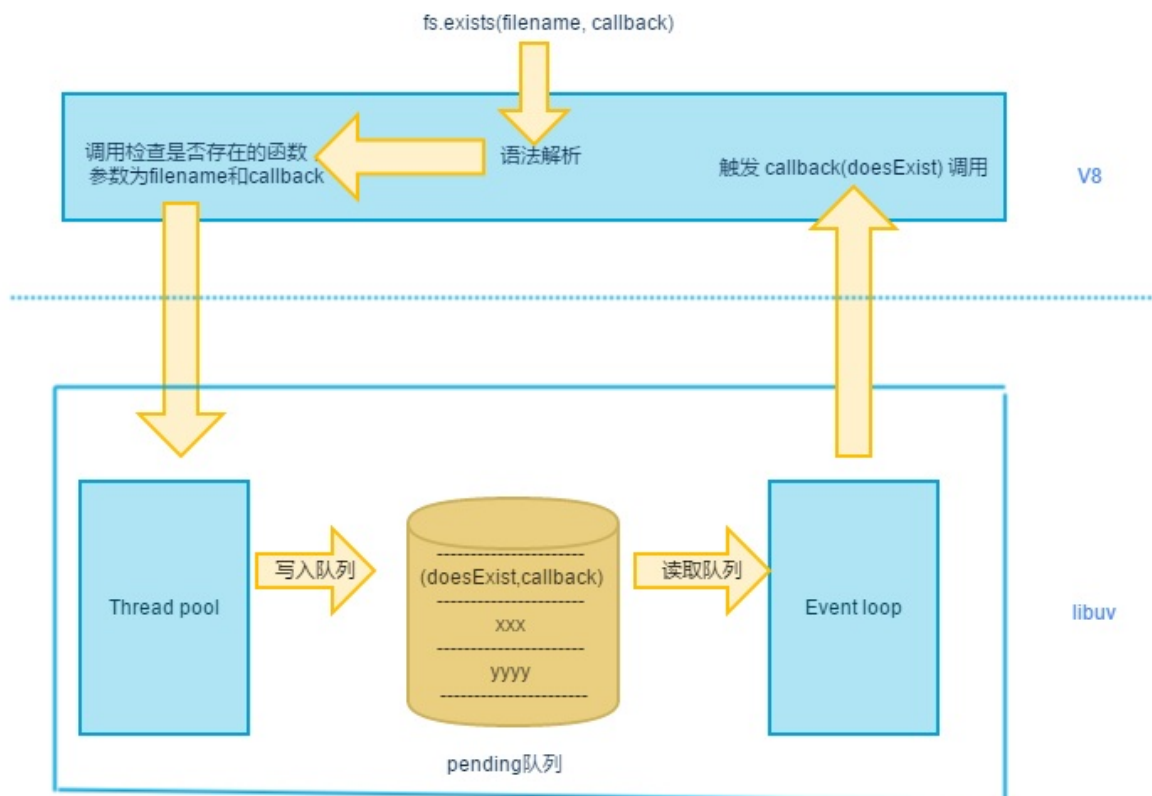


图 1.2.2 Node 中文件IO处理数据流向图

1.3 参考链接

- [Linux IO 概览](#)
- [Libuv 源码阅读](#)
- [The Node.js Event Loop, Timers, and `process.nextTick\(\)`](#)

2 JavaScript那些事

1900年代初期，Java 开始投入市场，并取得巨大成功。作为 Java 的维护者的 Sun 公司，也趁热打铁，开发出了 Applet，其实这是一个合成词，可以拆分成 Application 和 little 两个单词，意译的话就是“小程序”。

同时网景（Netscape）公司想开发一门让网页制作者就能学会的脚本语言，于是他们就委派了 Brendan Eich（布兰登·艾克）这位大神。由于时间紧迫，他用了10天便开发出了一个最初版本，开始的时候这个项目的代号还叫 Mocha（好吧，听起来很熟悉吧），并在第一版发布的时候给这门语言起名 LiveScript。不过，很快网景公司和 Sun 公司开展合作，两者达成协议，将这门语言改名为 Javascript。

从此 Sun 公司的 Applet 可以运行在浏览器上，只需要一个浏览器你的应用就可以随处运行了，而 Javascript 也借着 Java 的名声炒作了一把（虽然其语言本身跟 Java 没有多大关系，不过他成功迷惑了好多语言入门者，就好像我小时候搞不清楚雷锋和雷峰塔一样）。不过后来的结果大家大概都猜到了，小程序活不久，而 js 却被大量使用，直到今天依然无可替代。

2.1 数据类型

JavaScript 由于当初设计之初时间紧迫，所以有好多历史遗留问题，所以对于其有些语法知识点，大家可以完全以吐槽的心态来学习，完全不用去深究。

2.1.1 数字

一般编程语言都有像整数、浮点数之类的数据类型，但 js 将整数按照浮点数来存储，所以 `1 === 1.0`。不过 js 本身提供了一些浮点数转成整数（保证在显示的时候不带小数点）的函数，比如说 `Math.ceil`（向上取整）、`Math.floor`（向下取整）、`parseInt`（将字符串转化成整数，如果参数不是字符串，会先转化为字符串）。

2.1.2 布尔

布尔，也就我们常说的 `true` `false`，本来布尔就是用来做逻辑判断的，但是我们在做逻辑判断的时候也可以这个样子


```
if (undefined) {  
    console.log('undefined is true');  
}  
if (null) {  
    console.log('null is true');  
}  
if ('') {  
    console.log('\'\' is true');  
}  
if (0) {  
    console.log('0 is true');  
}  
if ('0') {  
    console.log('\'0\' is true');  
}  
if (NaN) {  
    console.log('NaN is true');  
}
```

代码 2.1.2.1 逻辑判断语句

你会发现 只有 `'0' is true` 被打印出来，其他的在做逻辑判断的时候都为假。本来这一节是讲布尔型数据类型的，但是我发现实在没啥可讲的，所以我就讲些逻辑判断相关的知识。其中 `undefined` 在 js 里面表示变量未定义；`null` 代表当前变量是一个对象，但是没有初始化；`''` 代表当前是一个字符串，但是字符串中没有任何字符；`0` 表示当前是一个字符串，且字符串就只有一个0字符；`NaN` 代表当前变量不是数字，这个数据类型在调用 `parseInt` 的时候会返回，比如说 `parseInt('a')` 就返回 `NaN`。

js 判断相等有两种方式 `==` 和 `===`，两者的区别是前者在做判断前，会将等号两边的数据类型转化成一致的；而后者在做判断的时候，如果检测到等号两边数据类型不一致，直接返回`false`。例如 `0 == '0'` 为 `true`，而 `0 === '0'` 为 `false`。最好需要注意，判断一个变量是否是`NaN`，不要使用等号来判断，而需要使用函数 `isNaN`。

最后讲述一个小技巧，使用下面方式判断一个变量是否是“不正常”的：

```
if (!x) {  
    console.log('x is dirty');  
}
```

2.1.3 字符串

js 中字符串可以使用"或者"来包裹，`'a' === "a"`，有一些语言（Java或者C）中用 `'a'` 来表示 `字符` 类型，在 js 中是没有 `字符` 类型的。

js 中提供了N多关于字符串的函数，比如说字符串截取函数 `substring`、替换函数 `replace`、查找函数 `indexOf`。

2.1.4 数组

`数组` 是 js 中一项重要的数据结构，我们可以通过如下方式声明一个数组：

```
var array = ['a','bb','cc'];
```

js 提供了一系列函数来对数组进行增删改查。

首先是查，你可以通过下标来访问数组元素，下标从0开始，`array[0]` 返回 `'a'`。同时数组还有一个 `length` 属性，通过这个属性我们可以写一个遍历这个数组的 `for` 循环：

```
for (var i=0,len=array.length;i<len;i++) {  
    console.log(array[i]);  
}
```

代码 **2.1.4.1** `for` 循环遍历数组

我们还可以通过 `indexOf` 函数来查找某一个元素是否在当前数组中。

然后是改，直接举个栗子，设置 0 号元素为 `'11'`，则使用 `array[0] = '11'`，即可。

接着是增加，通过 `shift` 函数删除数组最开头的元素，`var first = array.shift()` 调用完之后，变量 `array` 的值为 `['bb','cc']`，同时 `first` 被赋值 `'a'`。

通过 `unshift` 函数可以往数组头部添加元素，例如 `array.unshift('123')`，那么 `array` 变量的值就变成了 `['123', 'a', 'bb', 'cc']`。

通过 `pop` 可以删除数组最末尾的一个元素，例如 `var last = array.pop()` 调用完之后，`array` 就变成了 `['a', 'bb']` 同时变量 `last` 被赋值 `'cc'`。

通过 `push` 函数可以在数组末尾添加元素，例如 `array.push('333')`，则 `array` 变成 `['a', 'bb', 'cc', '333']`。

2.1.5 函数

作为一门编程语言，我们免不了在使用的时候要把某些功能封装成一个模块，函数作为模块的载体在任何程序语言中都是必不可少的，JavaScript 也不例外。在 js 中定义一个函数很简单：

```
function doAdd(a,b) {  
    return a + b;  
}
```

代码 2.1.5.1 函数定义

虽然函数体只有一行，但是这个函数却将函数三要素都澄清了：函数名 `doAdd`，参数 `a` 和 `b`，返回值 `a + b`。当然三要素并不是必不可缺的：

```
var doEcho = function() {  
    console.log('你好');  
}
```

代码 2.1.5.2 匿名函数

这里其实是定义了一个匿名函数，只不过我们在定义完之后将它赋值给了变量 `doEcho`，同时这个函数在运行的时候可以不用传任何参数，同时函数内部没有任何 `return` 语句，其实这种情况跟 `return undefined` 是等价的。匿名函数一般作用是作为函数参数使用，例如下面这个栗子：

```
function addLater(a,b,callback) {
  setTimeout(function() {
    var sum = a+b;
    callback(sum);
  },1000);
}

addLater(1,2,function(result) {
  console.log('the result:',result);
});
```

代码 2.1.5.3 匿名函数使用示例

上面这个栗子中，`addLater`函数在调用的时候，第三个参数在使用的时候是一个函数，而且它是匿名的。

2.2 对象

其实对象也是一种数据类型，只不过由于它太特殊，所以这里单独拿出来讲。在 ES6 之前 javascript 还是一门基于对象的编程语言，为啥叫基于呢，因为 ES5 和之前版本的 javascript 中原生语法中没有类（`class`）这个关键词，你只能拿原型（`prototype`）来模拟一个类的行为。

ES ([ECMAScript](#))，可以理解为 javascript 的语法标准，2015年6月发布的 ES6（又称 ES2015）版本增加了N多语言特性，其中就包括类和继承的实现。由于 ES6 规避了之前版本中 javascript 中的一些糟粕设计，并且提升了开发效率，所以产生了学习 ES6 的大量前端开发人员，但是现行浏览器对于 ES6 语法的支持能力参差不齐，所以 [babeljs](#) 应运而生，它提供了 ES6 转 ES5 的功能，一时间产生了大量的拥趸。同时国内大神阮一峰也写了一本 [ECMAScript 6 入门](#) 开源图书，我想使用 ES6 语法的程序员，没有一位没有浏览过这本书的。另外 node 从4.x开始逐渐引入 ES6 语法，具体各个版本的实现情况可以参见[Node.js ES2015 Support](#)。

我们这里先讲一下 ES5 中怎样模拟一个类，答案是使用原型：

```
function PersonES5(p) {  
    this.age = p.age;  
    this.name = p.name;  
    this.sex = p.sex;  
}  
  
PersonES5.prototype.showInfo = function() {  
    console.log(this);  
};  
  
var person = new PersonES5({  
    age:18,  
    name: 'tom',  
    sex: 'boy'  
});  
  
person.showInfo();
```

代码 2.2.1 person_es5.js

而在 ES6 中由于直接有类的概念，所以代码语法上还是有差别的：

```
class PersonES6 {
  constructor(p) {
    this.age = p.age;
    this.name = p.name;
    this.sex = p.sex;
  }
  showInfo() {
    console.log(this);
  }
}

var person = new PersonES6({
  age:18,
  name:'tom',
  sex:'boy'
});

person.showInfo();
```

代码 2.2.2 person_es6.js

由于 javascript 长期函数式编程思想盛行，因为我们一般不会在一个网页中呈现过多的 UI 组件，所以它的代码处理流程一般都是线性的。比如说我们在前端使用 javascript 的流程是这样的：加载网页->请求数据->渲染 UI 组件->触发事件监听，后端的流程是这样的：接收请求->数据库操作->返回处理结果。当然你会说，不对，我们处理的流程可比这复杂多了，当然随着单页应用（SPA,Single Page Application）的兴起，前端 js 的处理逻辑会越来越复杂。比如说有一天，你的经理可能会给你分配一个在线 photoshop 的需求，这时候面向对象就派上用场了，你可能需要一个抽象类来描述组件的基本属性和功能，同时派生出若干继承自这个抽象类的具体组件类，比如说矩形类、三角形类、圆形类。我想面对这么复杂需求的时候，开发者肯定会选择 ES6 来实现，更不用说如今流行 mvvm 框架都是采用 ES6 来开发。

上面啰嗦了这么多，其实是为了我自己开脱，我实在不想讲 ES5 中的原型链的知识点，为了搞清楚如何依赖原型链来实现继承，好多人都已经吐血了，这里就略过了，如果出现想用面向对象的场景，还是用 ES6 吧。

2.3 回调嵌套

由于在 javascript 中存在大量的异步操作，函数调用完成之后，不能立马拿到执行结果，必须在回调函数中得到执行结果，如果你在一个函数中要接连做好几次这样的异步处理，是不是画面应该是这样的：



图 2.3.1 代码深层次嵌套的即视感

正是由于考虑到这种问题，所以 ES6 在设计的时候增加 Promise 类，不过这东西在批量处理异步回调时候依然让人不爽，大家可以参考 [A quick guide to JavaScript Promises](#)。我这里给大家介绍的是一个第三方回调流程控制库 `async` (我这算不算开倒车？另外注意不要和 ES7 中的 `async` 关键字相混淆)。

我们在处理异步任务的时候，大体上分为两种情况，一种是串行操作，即处理完一个任务之后才能接着处理下一个任务；一种是并行操作，即各个任务都是独立运行的，大家一起运行，没有前后依赖关系。

对于串行运行在 `async` 中，可以是这样的：

```
var async = require('async');

async.waterfall([
  function(callback) {
    setTimeout(function() {
      callback(false, 2+3);
    }, 100);
  },
  function(sum, callback) {
    setTimeout(function() {
      callback(false, sum-1);
    }, 100);
  },
  function(left, callback) {
    setTimeout(function() {
      callback(false, left * 2);
    }, 100);
  }
], function(err, result) {
  console.log(err, result);
});
```

代码 2.3.1 async waterfall 方法示例

waterfall 函数接受两个参数，第一个参数是 **Array** 类型，用来指明各个需要异步执行的任务，数组的第一个元素为：

```
function(callback) {
  setTimeout(function() {
    callback(false, 2+3);
  }, 100);
}
```

注意 `callback(false, 2+3);` 这一句，调用完这一句，它就参数 `2+3` 这个值传递到下一个任务中去了，然后数组的第二个元素：


```
function(sum, callback) {  
  setTimeout(function() {  
    callback(false, sum-1);  
  }, 100);  
}
```

其中里面的 `sum` 正是刚才我们在第一个函数中传递过来的 `3+2`，同理可得我们最终将 `4` 作为参数 `left` 传递到了第三方函数中。`waterfall` 的第二个参数是一个回调函数：

```
function(err, result) {  
  console.log(err, result);  
}
```

其第一个参数 `err` 代表错误信息，假设我们在处理任何一个异步任务的回调时写了一个 `callback(errorInfo);`，整个 `waterfall` 函数会提前结束，并且将这个 `errorInfo` 传递到第一个参数 `err` 上；第二个参数 `result` 代表最终处理得到的结果，具体到上面那个栗子，最终的结构就应该是 `4*2` 得 `8`。

接着将并行处理，也就是 `parallel`，我们再举个栗子：

```
var async = require('async');

async.parallel([
  function(callback) {
    setTimeout(function() {
      callback(null, 1);
    }, 200);
  },
  function(callback) {
    setTimeout(function() {
      callback(null, 2);
    }, 100);
  }
],
function(err, results) {
  console.log(err, results);
  //最终打印结果：null [1,2]
});
```

代码 2.3.2 async parallel 函数示例

和 `waterfall` 类似，只要其中有一个任务在 `callback` 的时候传递了一个 `error` 对象，就会导致整个 `parallel` 函数立马结束。

最后大家可能留意到我们的第一行使用了 `require('async')`，这个 `require` 函数用来加载第三方包，我们需要在代码文件所在目录运行 `npm install async --save` 来安装这个第三方包。更多关于 `npm` 的知识可以参见本书第4章。

本章部分代码：<https://github.com/yunnysunny/nodebook-sample/tree/master/chapter2>

2.4 参考文献

- <https://www.twilio.com/blog/2016/10/guide-to-javascript-promises.html>

3 Node 基础

3.1 安装

打开 [Node官网](#) ,引入眼帘的就是它的下载地址了，windows下提供的是安装程序（下载完之后直接双击安装），linux下提供的是源码包（需要编译安装），详细安装流程这里省略掉，我想这个不会难倒各位好汉。

3.2 旋风开始

在讲 Node 语法之前先直接引入一段 Node 的小例子，我们就从这个例子着手。首先我们在随意目录下创建两个文件 `a.js` `b.js` 。

```
exports.doAdd = function(x,y) {  
    return x+y;  
};
```

代码 3.2.1 a.js

```
const a = require('./a');  
  
console.log(a.doAdd(1,2));
```

代码 3.2.2 b.js

和普通前端 javascript 不同的是，这里有两个关键字 `exports` 和 `require` 。这就牵扯到模块化的概念了，javascript 这门语言设计的初衷是开发一门脚本语言，让美工等从业人员也能快速掌握并做出各种网页特效来，加之当初语言创作者开发这门语言的周期非常之短，所以在 javascript 漫长的发展过程中一直是没有模块这个语言特性的（直到最近ES6的出现才打破了这个格局）。

> Node 是最近几年才发展起来的语言，前端 js 发展的历史要远远长于他，2000年以后随着 [Ajax](#)技术越来越流行，js的代码开始和后端代码进行交互，逻辑越来越复杂，也越来越需要以工程化的角度去组织它的代码。模块化就是其中一项亟待解决的问题，期间出现了很多模块化的规范，[CommonJS](#)就是其中的一个解决方案。由

于其采用同步的模式加载模块，逐渐被前端所抛弃，但是却特别适合服务器端的架构，服务器端只需要在启动前的时候把所有模块加载到内存，启动完成后所有模块就都可以被调用了。

我们在命令行中进入刚才我们新创建的那个文件夹下，然后运行 `node b.js`，会输出 `3`，这就意味着你的第一个node程序编写成功了。

在exports 对象会被 导出，在 `b.js` 中通过`require` 就能得到这个被导出的对象，所以我们能访问这个被导出对象的 `doAdd` 函数。假设我们在 `a.js` 中还有一个局部变量：

```
var tag = 'in a.js';
exports.doAdd = function(x,y) {
  console.log(tag,x,y);
  return x+y;
};
```

代码 3.2.3 a.js

这里定义的 `tag` 变量是没法在 `b.js` 中读取的，其作用区域仅仅被局限在 `a.js` 中。如果在 `b.js` 中打印 `console.log(a.tag)` 会输出 `undefined`。

这里 `b.js` 里我们引用 `a.js` 时用 `require('./a')`，假设我们现在的目录结构是这样的

```
---a.js
---b.js
---lib
|-----c.js
```

目录3.2.1

这个时候我们在`b.js`中就可以通过 `const c = require('./lib/c');` 来引入 `c.js`。同时 `node` 本身还包含了各种系统API。比如通过`require('fs')`，可以引入系统自带的 [文件操作库](#)。下面就举一个操作文件的栗子：

```
const fs = require('fs');

exports.getData = function(path, callback) {
  fs.exists(path, statCallback);

  function statCallback(exists) {
    if (!exists) {
      return callback(path+'不存在');
    }
    const stream = fs.createReadStream(path);
    let data = '';
    stream.on('data', function(chunk) {
      data += chunk;
    });
    stream.on('end', function() {
      callback(false, data);
    });
  }
};
```

代码 3.2.4 c.js

代码 3.2.4 中 函数 `exists` 用来判断文件是否存在，`createReadStream` 函数返回一个 **readable stream**(可读流)，node 中 IO（包括文件 IO 和网络 IO）处理采用 **stream**(流) 的方式进行处理。同时在流的内部还使用 **EventEmitter** 来触发事件，具体到 代码 3.2.4 中，我们会看到 `data` 事件和 `end` 事件，分别表示当前有新读入的数据、当前的数据全都读取完毕了。

接下来我们写一个测试代码来对 `c.js` 进行测试：

```
const path = require('path');
const c = require('./lib/c');

c.getData(path.join(__dirname, 'test.txt'), function(err, data) {
  console.log(err, data);
});
```

代码 3.2.5 fs_test.js

注意上述代码中的全局变量 `__dirname` 他获取的是当前代码文件所在的路径，我们在 `fs_test.js` 同级目录下放了一个 `test.txt`，所以这里使用 `path.join(__dirname, 'test.txt')` 来获取一个绝对路径。假设我们的 `test.txt` 在目录 `data` 中，

```
---fs_test.js
---data
|-----test.txt
```

目录 3.2.2

那么就写 `path.join(__dirname, 'data/test.txt')`。

3.3 做一个Apache

现在我们做个更让人兴奋的栗子，做一个 **Apache**，当然这里的 **Apache** 不是武装直升机，而是一个服务器，熟悉 **php** 的人对他肯定不会陌生。你在本地安装它之后，然后在其默认的网站目录中放一张图片，我们假设它为 `a.jpg`，然后你可以通过 <http://localhost/a.jpg> 来访问它了。下面的内容就是要模拟这个过程。

要做这个处理，我们首先要搞懂 **node** 中的 **http** 包。我们抄一段 **node** 官网给出的快速搭建 **http** 服务器的代码吧：

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

代码 3.3.1 example.js

直接运行 `node example.js`，然后我们打开 chrome，输入网址

`http://localhost:3000`,就会在网页上看到 `Hello world`。OK，我们回头看一下代码，关键部分在于 `createServer` 的回调函数上，这里有两个参数 `req` 和 `res`，这两个变量也是 `stream` 类型，前者是 **readable stream**(可读流)，后者是 **writable stream**(可写流)，从字面意思上推测出前者是用来读取数据的，而后者是用来写入数据的。大家还有没有记得我们在代码 3.2.4 中函

数 `fs.createReadStream` 也返回一个 `readable stream`。接下来就是一个见证奇迹的时刻，`stream` 类上有一个成员函数叫做 `pipe`，就像它的名字 管道 一样，他可以将两个流通过管子连接起来：

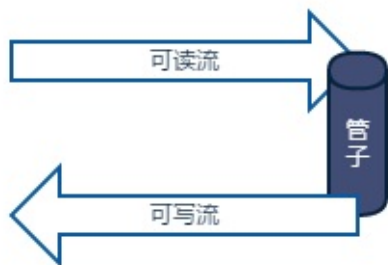


图 3.3.1 pipe原理

有了 `pipe` 这个功能，我们就能将 `fs.createReadStream` 函数得到的可读流转接到 `res` 这个可写流上去了。说干就干，我们简单修改一下代码 3.3.1，就可以让其成为一个 Apache：

```
const http = require('http');
const fs = require('fs');
const path = require('path');

const hostname = '127.0.0.1';
const port = 3000;
const imageDir = __dirname + '/images';

const server = http.createServer((req, res) => {
  const url = req.url;
  const _path = path.join(imageDir, url);
  fs.exists(_path, function(exists) {
    if (exists) {
      res.statusCode = 200;
      res.setHeader('Content-Type', `image/${path.extname(
url).replace('.', '')}`);
      fs.createReadStream(_path).pipe(res);
    } else {
      res.statusCode = 404;
      res.end('Not Found');
    }
  });
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

代码3.3.2 app.js

我们仅仅使用了一句 `fs.createReadStream(_path).pipe(res);` ,就便捷的将文件流输出到HTTP的响应流中了，是不是很强大。OK来看一下效果，运行 `node app.js` ，在浏览器中打开 `http://localhost:3000/a.png` 就能看到显示效果。

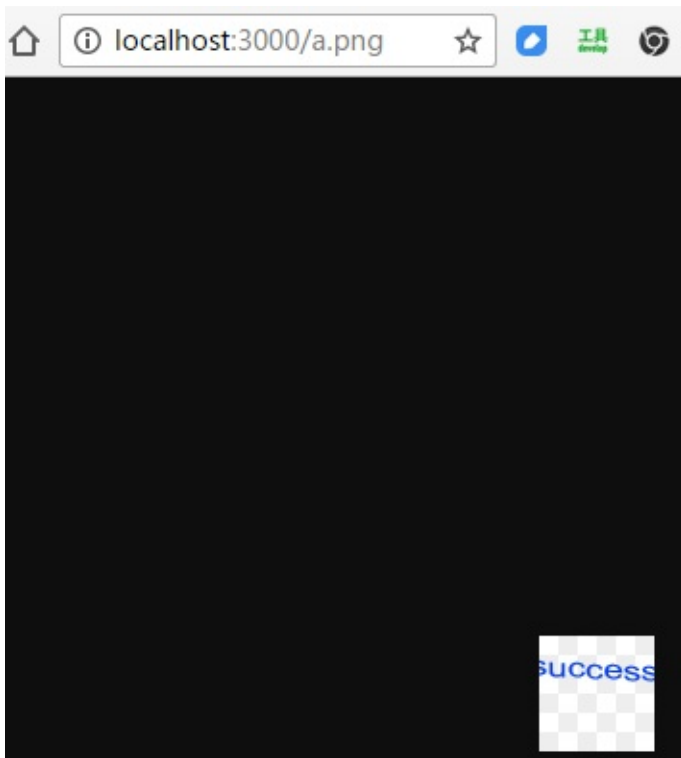


图 3.3.2 最终我们的apache显示效果

3.4 HTTP请求参数

既然我们称 Node.js 是一门后端语言，那他就应该能处理 HTTP 请求中的请求参数，比如说我在 URL 上添加查询参数（类似于这种 `/xxx?a=1&b=2` ），再比如说通过表单提交数据。Node 确实提供了处理这两种数据的能力，只不过让人感觉到稍显“低级”。为什么这么说来，下面就一一道来。

3.4.1 GET 请求

首先我们来简单描述一下 HTTP 请求，打开浏览器，并且打开开发者工具，



图3.4.1.1 通过菜单打开控制台

然后使用谷歌搜 `node`，我们定向到开发者工具的 **Network** 标签页，然后开第一条网络请求，鼠标单击点开这条网络请求，会显示格式化好的HTTP 请求和响应的数据包内容：

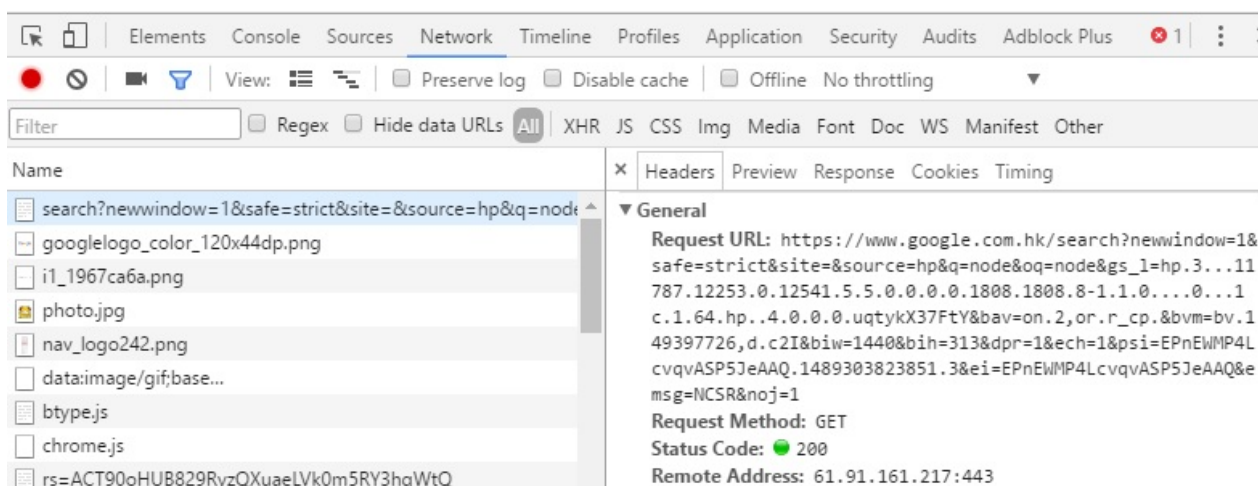


图3.4.1.2 使用谷歌搜索 `node`

我们这里仅仅关注一下 **General** 部分，**Request URL** 中 `?` 后面是一大串请求参数，在我们这里是

```
?newwindow=1&safe=strict&site=&source=hp&q=node&oq=node
```

，由于参数太长了，我们故意省略掉一些。然后还有一个参数 `Request Method`（请求方法）为 `GET`，HTTP 协议中很几种常用的请求方法：`GET POST HEAD PUT DELETE`，其中最常用的就是前两者。这个栗子中给出的请求方法正是 `GET` 请求，它将参数类似 `?key1=value&key2=value2` 方式组织在一起，拼接在请求地址的后面，发送到服务器端，服务器要解析这个请求参数，然后得到参数 `key1` 的值是 `value1`，`key2` 的值是 `value2`，继而进行逻辑处理。

具体到上面这个栗子中，我们要在后端取到 `source` 这个参数的值，应该怎么做呢？可能很多之前有 `php` 或者 `java` 经验的开发者，会通过 `$_GET['source']` 或者 `request.getParamater("source")` 能够轻而易举的把这件事给办了，但是 `node` 不行。下面我们对代码 **3.3.1**稍加改造来实现这个目的：

```
const http = require('http');
const url = require('url');
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  const query = url.parse(req.url, true).query;
  console.log(query);
  res.setHeader('Content-Type', 'text/plain');
  res.end(`Hello World from ${query['source']}\n`);
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

代码 3.4.1.1 get.js GET 请求参数处理

运行 `node get.js`，打开浏览器输入 `http://127.0.0.1:3000/?newwindow=1&safe=strict&site=&source=hp&q=node&oq=node`，会打印出 `Hello World from hp`。

注意代码 **3.4.1.1** 中的 `url` 库的 `parse` 函数，它对 URL 地址进行解析，得到 URL 中的域名、请求路径等参数，其中第二参数在这里要设置 `true`，否则它不会解析请求参数。

3.4.2 POST 请求

刚才我们在演示 GET 请求的使用的是谷歌搜索的栗子，它的请求参数在地址栏中都可以看到，这种方式简介明了，你甚至可以给朋友发一个网址，他就可以搜出来跟你一样的内容。但是有时候，我们还要在互联网上做一些比较隐私的事情，比如说登录某个网站，你肯定不想让你的用户名、密码出现在浏览器的地址栏中，这个时候 POST 请求就派上用场了。

我们把代码 **3.4.1.1** 稍微修改一番，来演示一下后端怎样读取 POST 参数：

```
const http = require('http');
const url = require('url');
const qs = require('querystring');
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  const query = url.parse(req.url, true).query;
  let postStr = '';
  req.on('data', function(data) {
    postStr += data;
  });
  req.on('end', function() {
    const post = qs.parse(postStr);
    res.setHeader('Content-Type', 'text/plain');
    res.end(`Hello ${post['name']} World from ${query['source']}
    || 'unknown'\n`);
  });
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

代码 **3.4.2.1 post.js** POST 参数处理

从代码 **3.4.2.1** 可以看出处理 POST 参数时竟然要需要从流中读取数据，这是由于在 HTTP 协议中 POST 数据被认为可以发送大容量的请求数据，所以为了防止堵塞，Node 强制使用流来处理这部分数据。

由于不能直接通过浏览器输入地址来测试 POST 请求，所以推荐大家使用一个 chrome app 来测试 POST 请求，它就是 **Postman**。安装完成之后，打开主界面，下拉菜单区域选择 **POST**，然后输入地址 `http://localhost:3000`，打开 **Body** 标签，选择 **x-www-form-urlencoded**，最后点击 **Send** 按钮，完成后就会输出 `Hello sunny World from unknown`。

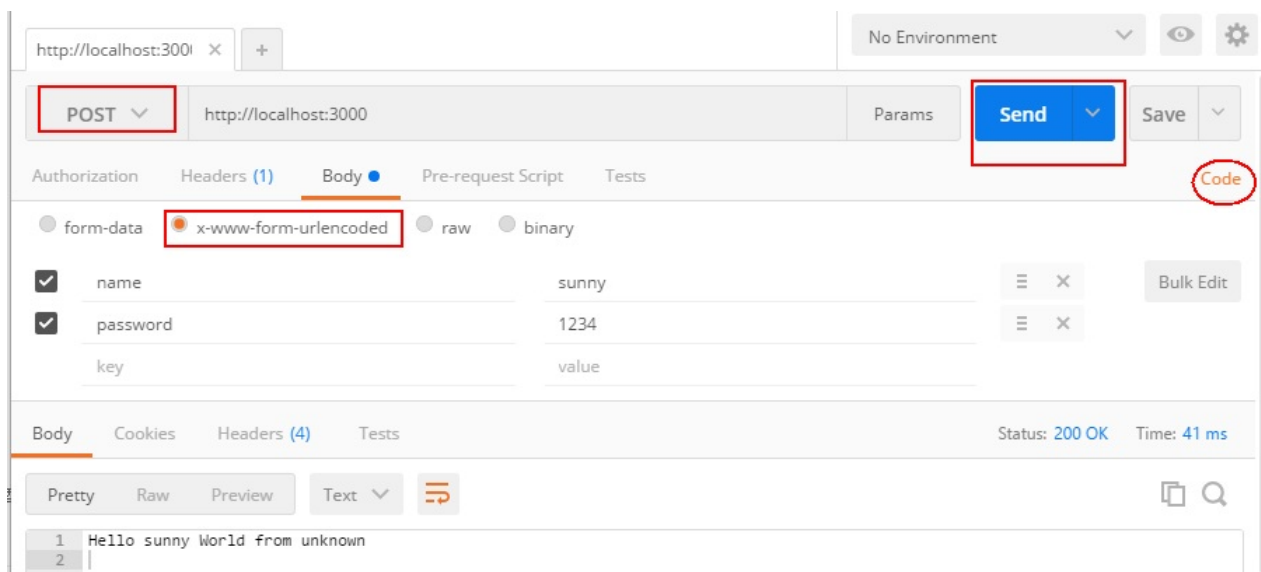


图3.4.2.1 Postman 发送 POST 请求

这里之所以需要选择 **x-www-form-urlencoded**，是由于 POST 数据包有很多组织方式，我们最常用的就是这种方式，其数据格式依然是 `key1=value1&key2=value2`，而 Node 自带的 `querystring.parse` 函数正是用来处理这种字符串的。同时我们可以点击 **Code** 按钮，来查看 POST 请求发送的请求数据包：

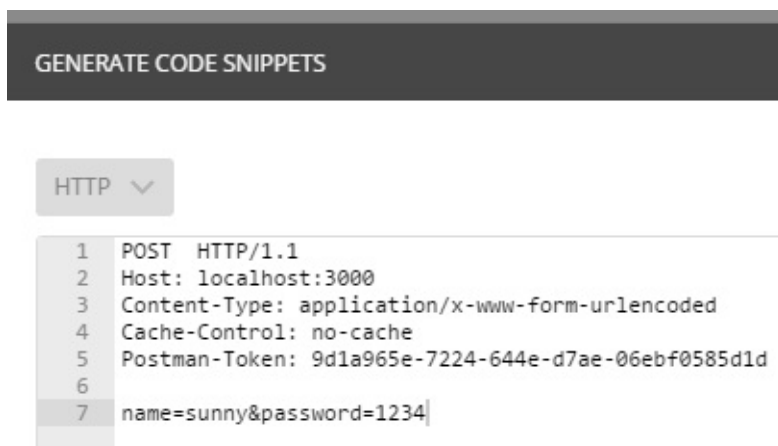


图3.4.2.2 POST 请求数据包

你可以留意到请求包中当中有一个空行（HTTP协议中用空行来分割请求头和请求正文），空格下面的 `name=sunny&password=1234` 就是我们 POST 到后台的数据。

3.5 总结

我们用两个小节讲述了 Node 中如何处理静态资源和动态请求，看完这些之后，如果你是一个初学者，可能会因此打退堂鼓，这也太麻烦了，如果通过这种方式来处理数据，跟 php java 之类的比起来毫无优势可言嘛。大家不要着急，Node 社区已经给大家准备了各种优秀的 Web 开发框架，比如说 [Express](#)、[Koa](#)，绝对让你爱不释手。你可以从本书的第5章中学习到 `express` 基本知识。

本章示例代码可以从这里找到：<https://github.com/yunnysunny/nodebook-sample/tree/master/chapter2>。

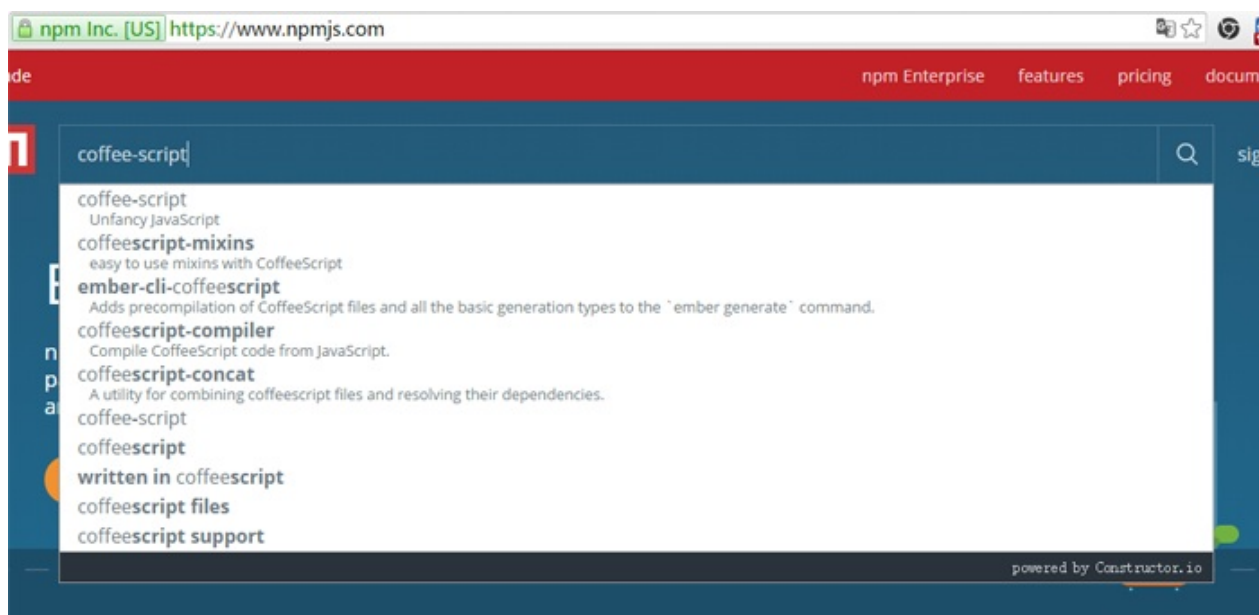
4 NPM 包管理工具

在Node出现之前，我们在做前端的时候经常用到一些开发工具使用ruby和python开发的（比如说sass，一种CSS预编译语言，依赖于ruby；(Pygments) [<http://pygments.org/>]，一种代码语法高亮插件，依赖于python），这个时候就会看到gem和pip的身影。熟悉java的同学，应该也对maven如数家珍。和这些语言类似，Node 也提供了包管理工具，它就是 npm，全名为 **Node Package Manager**，集成于 Node 的安装程序中。

4.1 使用NPM

npm 不仅可用于安装新的包，它也支持搜寻、列出已安装模块及更新的功能。

npm 目前拥有数以百万计的包，可以在 <https://www.npmjs.com/> 使用关键字搜寻包。举例来说，在关键字栏位输入“coffee-script”，下方的清单就会自动列出包含 coffee-script 关键字的包。



虽然也可以通过 `npm search` 来在命令行中查询，但是初次查询过程中要在本地建立索引，等待的时间巨漫长，还是不介绍的为好。

找到需要的包后，即可使用以下指令安装：

```
npm install coffee-script
```


运行完之后，就会在当前目录下的 `node_modules` 目录下安装 `coffee-script` 包。

```
├─ node_modules
│  └─ coffee-script
```

目录结构 **4.1.1** 将包安装到本地后的目录结构

一般情况下，我们在 `node` 项目目录下创建 `package.json`，里面包含项目名称、作者、依赖包等配置，我们可以通过 `npm init` 快速创建一个 `package.json` 文件，我们新建一个目录，然后在命令执行 `npm init`，则会要求你填入若干信息：

```
name: (app) test
version: (1.0.0) 0.0.1
description: test app
entry point: (index.js)
test command:
git repository:
keywords:
author: yunnysunny
license: (ISC) MIT
About to write to I:\node\app\package.json:

{
  "name": "test",
  "version": "0.0.1",
  "description": "test app",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "yunnysunny",
  "license": "MIT"
}

Is this ok? (yes)
```


命令输入 4.1.1 npm init 命令输入示例

我们接着在项目中安装 **express** 包（在第5章会讲到这个包的使用），不过我们执行命令的时候加个参数：`npm install express --save`。命令执行完成之后，再看 `package.json`，发现多了一个配置属性：

```
"dependencies": {  
  "express": "^4.14.0"  
}
```

这个 `dependencies` 属性里面描述的就是当前项目依赖的各种包，你可以通过运行 `npm install packageName --save` 来将其安装到本地的同时在 `package.json` 中同时添加依赖声明。当你代码开发完成时，要把项目往服务器上部署，那么这时候 `package.json` 中的依赖声明都已经写好了，这时候，你直接在项目目录运行 `npm install`，就可以自动将声明中的文件全部下载安装到项目目录的 `node_modules` 子目录下。

我们在来稍微留意一下，我们配置的这个 **express** 的版本号，咦，`^` 是个什么鬼？讲这个东东，还要从 [Semantic Versioning](#) 这个概念讲起，它将版本号分为三段：

1. 主版本，你可以在这个版本中做不兼容性改动
2. 小版本，你可以在这个版本上增加共嗯那个，不过要向后兼容
3. 补丁版本，在这里可以做一些bug修复，不过依然要保持向后兼容

在这里对于 **express** 来说，主版本号是 `4`，小版本号是 `14`，补丁版本号是 `0`。啰嗦了这么多，那么 `^` 呢，它告诉你使用从 `4.14.0` 到 `5.0.0`（不包括 `5.0.0`）之间的最新版本，也就是说它选择的版本号 `x` 的取值范围：`4.14.0 <= x < 5.0.0`。

除了 `^`，还有一个版本号标识符 `~` 也很常用，假设我们将这里 **express** 的版本号设置为 `~4.14.0`，那么它表示从 `4.14.0` 到 `4.15.0`（不包括 `4.15.0`）之间的最新版本，也就是说它选择的版本号 `x` 的取值范围：`4.14.0 <= x < 4.15.0`。

另外还有一些版本号的特殊标志符，由于不常用，有需要的可以参

考 <https://docs.npmjs.com/misc/semver>。一般情况下，我们通过将依赖安装到项目目录下，但是有时候我们需要做全局安装，这种全局安装的包一般都是些命令行程序，这些命令行程序安装到全局后就可以保证我们通过 `cmd.exe`（或者 `bash`）中调用这些程序了。下面我们演示一下如何全局安装 [express-generator](#)：

```
npm install -g express-generator
```

安装完成后会提示安装到了目录 `C:\Users\[用户名]\AppData\Roaming\npm\node_modules` 目录下，其实这个安装目录是可以指定的，老是往系统盘安装会让人抓狂，下面要讲到这个问题。

安装完 `express-generator`，我们在命令行中新建一个目录 `mkdir first-express`，然后进入这个目录运行 `express`，如果发现生成了一堆 `express` 项目文件，恭喜你成功了！

4.2 NPM用不了怎么办

互联网拉近了整个世界的距离，有时候让你感觉到近到只有一墙之隔。前面讲了很多 `npm` 的使用方法，但是我们要想到 `npmjs` 毕竟是一个外国网站，作为一个开发人员，相信你也许经历过很多技术网站，安安静静的躺在那里，但是就是无法访问的问题，但是谁又能保证 `npmjs` 不会是下一个中枪者呢？

幸好，阿里开发出了 `cnpm`，一个完整 `npmjs.org` 镜像，每隔10分钟和官方库进行一次同步。其安装命令很简单：

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

命令 4.2.1

不过你需要注意，由于最新版本的 `cnpm` 不兼容低版本 `node`，如果你当前使用的 `node` 版本低于 `4.x`，那么你需要在安装的时候指定版本号：

```
npm install -g cnpm@3.4.1 --registry=https://registry.npm.taobao.org
```

命令 4.2.2

否则的话，安装完之后运行命令会报错。

接着你可以使用 `cnpm` 来代替 `npm`，比如说 `cnpm install` 来代替 `npm install`，又可以愉快的玩耍了。

接着，我们尝试使用 `cnpm` 全局安装 `lodash`，运行假设你的 `nodejs` 安装在 `windows` 的 `C` 盘的话，运行完 `cnpm install lodash -g` 后，你会惊奇的发现报错了：

```
npm ERR! Error: EPERM, mkdir 'C:\Program Files (x86)\nodejs\node_modules\lodash'
npm ERR!   { [Error: EPERM, mkdir 'C:\Program Files (x86)\nodejs\node_modules\lodash']
npm ERR!     errno: 50,
npm ERR!     code: 'EPERM',
npm ERR!     path: 'C:\\Program Files (x86)\\nodejs\\node_modules\\lodash',
npm ERR!     fstream_type: 'Directory',
npm ERR!     fstream_path: 'C:\\Program Files (x86)\\nodejs\\node_modules\\lodash',
npm ERR!     fstream_class: 'DirWriter',
npm ERR!     fstream_stack:
npm ERR!       [ 'C:\\Users\\sunny\\AppData\\Roaming\\npm\\node_modules\\cnpm\\node_modules\\npm\\node_modules\\fstream\\lib\\dir-writer.js:35:25',
npm ERR!         'C:\\Users\\sunny\\AppData\\Roaming\\npm\\node_modules\\cnpm\\node_modules\\npm\\node_modules\\mkdirp\\index.js:47:53',
npm ERR!         'Object.oncomplete (fs.js:108:15)' ] }
npm ERR!
npm ERR! Please try running this command again as root/Administrator.

npm ERR! Please include the following file with any support request:
```

输出 4.2.1 cnpm全局安装错误输出

为啥呢？首先，cnpm 会在将包默认安装在nodejs安装目录下的 node_modules 子文件夹中，其次我们这里将node安装到了系统盘 C:\Program Files (x86) 目录下，最后写入这个目录需要超级管理员权限。

本来就讨厌往系统盘写入数据文件，这下子非要改掉它这个默认设置不可了。我们命令

```
npm config set prefix "D:\npm"
```

命令 4.2.3

这样你使用 `npm install -g package` 命令安装的包就会被放置到 `${prefix}/node_modules` 下。同时使用命令

```
npm config set cache "D:\npm\node_modules\npm-cache"
```

命令 4.2.4

可以设置npm的缓存路径，否则的话它默认会缓存一部分下载的包到系统目录中。

我们推荐仅仅在全局安装命令行工具类型的包，因为同一个包在不同项目中很有可能使用不同的版本，所以如果将其安装在全局的话，就没办法使用不同版本了。如果你非要将某一个类库安装到全局的话，那就增加一个 `NODE_PATH` 环境变量，指向我们刚才设置的目录 `D:\npm`。

4.3 其他一些命令

如果你想查看当前全局安装了哪些包，可以使用 `npm list -g` 命令，运行完成后会打印一个目录树，但是如果安装的包比较多的话，在命令行中会打印不全，所以可以采用重定向的方式将打印结果输出到硬盘，例如 `npm list -g > d:\package.txt`。如果不加 `-g` 参数就是打印当前目录下 `node_modules` 文件夹下的包结构。

有时候，我们需要将安装好的包删除掉，如果包是安装在项目目录下的话，其实直接可以把 `node_modules` 下对应的文件夹删除即可，如果是全局安装的话，那还是使用命令进行卸载吧，比如卸载我们上面安装的 `express-generator`：`npm uninstall express-generator -g`。同样这里的 `-g` 是说卸载全局安装的 `express-generator` 包。

4.4 yarn

我们在4.1节提到 `Semantic Versioning` 这个概念，但是这个约定全凭开发者去自觉遵守显然不现实。在之前的开发中，我使用 `supertest` 这个包来做单元测试，当时安装的是 `2.0.0` 版本，过了几天我新建了一个项目使用 `npm install` 来安装依赖的时候被安装的是 `2.0.1` 版本。按理来说依照规则，小版本变更应该是用来修改bug的，没想到我运行单元测试之后直接在 `supertest` 中报了语法错误。去网上一查，当前报错代码在 `node 4.x` 中才能避免，而我用的是 `0.10.x`，然后手动查看了一下

supertest 的 package.json 文件，发现它在这一个小版本改动中悄悄的将 engine 属性的 node 版本改为 `>=4.0.0`。单纯使用一个包都可能会导致风险，更不用说在一个庞大的项目中使用大量的依赖，那个时候可真叫牵一发而动全身。

yarn 正是 facebook 的开发人员在开发 React Native 的时候实在无法忍受第三方包版本号变更带来的兼容问题，怒而开发之。

需要注意的 yarn 需要 node 版本大于 4.0.0。

yarn 很多命令和 npm 相似，比如说 `yarn init` 对应 `npm init` 来初始化项目，`yarn install` 对应 `npm install` 来读取配置文件安装依赖包。

不过通过 `yarn install` 安装过程中，yarn 将下载下来的包都缓存到了本地（这个缓存路径在 windows 下默认为 `C:\Users\[user]\AppData\Local\Yarn\cache`），下次如果换个项目再安装相同包，并且版本号也跟之前安装的一样的话，它就直接从缓存中读取出来。同时，yarn 在安装包的时候是并行的，而 npm 在安装包时串行的，必须第一个包安装完成之后才能安装第二个包。所以综上所述，yarn 安装包的速度要比 npm 快。

我们通过 `npm save [package] --save` 来将依赖包安装到当前项目下，同时将配置写入 package.json 文件，在 yarn 中使用 `yarn add [package]` 即可，如果你要想全局安装包可以用 `yarn global add [package]`（之前我们通过命令 4.2.3 设置过全局包安装的路径，yarn 也会读取这个设置）。此外 yarn global 还有一些很有用的命令，大家可以参见[这里](#)。

不过我们通过 `yarn add [package]` 后，他还会在当前目录下生成或者修改一个 yarn.lock 文件。例如我们运行 `yarn add express` 后就会发现文件 yarn.lock 中内容如下：

```
# THIS IS AN AUTOGENERATED FILE. DO NOT EDIT THIS FILE DIRECTLY.
# yarn lockfile v1

accepts@~1.3.3:
  version "1.3.3"
  resolved "https://registry.yarnpkg.com/accepts/-/accepts-1.3.3.tgz#c3ca7434938648c3e0d9c1e328dd68b622c284ca"
  dependencies:
    mime-types "~2.1.11"
    negotiator "0.6.1"

array-flatten@1.1.1:
  version "1.1.1"
  resolved "https://registry.yarnpkg.com/array-flatten/-/array-flatten-1.1.1.tgz#9a5f699051b1e7073328f2a008968b64ea2955d2"

content-disposition@0.5.1:
  version "0.5.1"
  resolved "https://registry.yarnpkg.com/content-disposition/-/content-disposition-0.5.1.tgz#87476c6a67c8daa87e32e87616df883ba7fb071b"
```

代码 4.2.1 yarn.lock 示例

你会发现里面罗列了 **express** 各个依赖的版本号（**version** 字段），下载地址（**resolved** 字段），我们仅仅截取了前面几行，因为 **express** 包中依赖关系比较复杂，生成的这个 **lock** 文件也比较长。项目初始化的老兄，通过 **yarn add** 的方式安装好包之后，需要将这个 **yarn.lock** 提交到版本库，这样你的小伙伴通过 **yarn install** 安装的各个依赖就和初始化的老兄用的一样了，这样就避免了团队中各个开发者通过 **npm install** 安装到本地的包的版本号不一致而导致的各种难以排查的问题了。

更多关于 **npm** 和 **yarn** 的对比可以参见[官方文档](#)。

不过我们在使用 **yarn** 的时候，因为 **yarn** 在底层依然得使用 **npm** 进行安装，所以依然无法避免因网络原因导致的包无法下载的问题，不过我们可以直接将 **npm** 的安装源设置为 **cnpm** 提供的安装源：

```
yarn config set registry http://registry.cnpmjs.org
```

命令4.4.1

4.5 发布自己的包到 npmjs

刚才演示了这么多命令都是安装别人的包，现在我们自己开发一个包。首先你要注册一个npmjs的账号（注册地址：<https://www.npmjs.com/signup>）。注册完成后，通过 `npm adduser` 命令来将注册的账号绑定到本地机器上，运行完改命令后会让你输入 npmjs 的注册账号和密码。

要想在 npmjs 上发布自己的包，首先要做的是明确你发布的包名在这个网站上有没有存在，在4.1小节，我们上来就介绍了怎么通过包名搜索npmjs上的包。不过，这里提供一个简单暴力的方法，就是直接在浏览器里输入：

`npmjs.com/package/packageName`，将packageName替换成你所想创建的包名，然后回车，如果打开的网页中有404映入你的眼帘，恭喜你，这个包名没有被占用。

我这里演示一下，我开发包slogger的过程，首先在浏览器地址栏里输入：

`npmjs.com/package/slogger`，很不幸，slogger 这个包名已经被占用了。于是乎我输入 `npmjs.com/package/node-slogger`，咦没有被占用（我们应该用发展的眼光的看待问题）。接着新建一个目录node-slogger，在命令行中进入这个目录，运行

```
npm init :
```

```
name: (node-slogger) node-slogger
```

```
version: (1.0.0) 0.0.1
```

```
description: A wrapper of logger package , which can write same  
code even if you change you logger api.
```

```
entry point: (index.js)
```

```
test command:
```

```
git repository: git@github.com:yunnysunny/slogger.git
```

```
keywords:  logger
```

```
author: yunnysunny
```

```
license: (ISC) MIT
```

输出 **4.5.1** 运行 `npm init` 后的部分输出

注意我们在 `git repository` 位置填写了一个 `git` 地址，这就意味着当前的代码要托管在github上。接着我们编写代码，然后将代码push到github，接着给预发布的代码打一个tag，最后运行 `npm publish`，打完收工，现在我们看

<https://npmjs.com/package/node-slogger>，包已经可以访问了！

5 数据库

我们在使用 node 处理业务逻辑的时候难免要和数据打交道，这时候数据库就派上用场了。在 node 中最常用的数据库有两种，redis 和 mongodb。本章也正是围绕这两个数据库展开讲解。

5.1 redis

redis 提供 key-value 类型的存储结构，是一种内存数据库，因此数据查询速度特别快，而且它还可以通过配置来实现将数据定期备份到磁盘上的功能，一定程度上解决进程掉线后数据恢复的问题。

node 中推荐使用 ioredis 这个驱动来对 redis 进行操作。redis 这个驱动虽然使用人数更多，但是从 redis 3.x 开始增加了 cluster 模式，但是这个驱动并不支持这种模式，所以不推荐使用。

```
var Redis = require('ioredis');
/**
 * 如果不传参数默认连接127.0.0.1:6379端口
 * */
var redis = new Redis(/*{"port" : 6379,"host" : "127.0.0.1",password: 'auth'}*/); //没有密码不需要传password参数
/*
var clusterRedis = [
  {
    "host": "127.0.0.1",
    "port": 6379
  },
  {
    "host": "127.0.0.1",
    "port": 6380
  }
];
var redis = new Redis.Cluster(clusterRedis, {redisOptions: {password: 'auth'}}); //集群连接方式
*/

redis.set('foo', 'bar', function(err, reply) {
  console.log(err, reply); //正常情况打印 null 'OK'
});
redis.get('foo', function (err, result) {
  console.log(err, result); //正常情况打印 null 'bar'
});
```

代码 5.1.1 redis 命令基本演示

redis 中大多数的命令格式都是这样的 `command key param1 param2 ...` 对应 ioredis 中的函数就是 `redis.command(key, param1, param2, ...)` 比如说代码 5.1.1 中的栗子，我们在 redis-cli 中执行 `set foo bar` 命令就对应我们的 `redis.set('foo', 'bar')` 这行代码。注意到我们这里在接收处理结果的时候都是使用 `callback` 的方式，ioredis 内部也支持 `promise` 方式来接收处理结构，你只需要将回调函数去掉，改成 `then` 函数：

```
redis.set('foo', 'bar').then(function(reply) {  
  
});
```

代码 5.1.2 使用 **promise** 方式接收返回数据

有时候我们在使用 **redis** 的时候，在一个处理逻辑中要连续发送多条 **redis** 命令，这时候你可以考虑用 **ioredis** 中提供的 **pipeline** 或者 **multi** 函数。

使用 **pipeline** 时 **ioredis** 内部将一系列指令缓存到内存，最后通过 **exec** 函数执行后打包发送到 **redis** 服务器，而且它支持链式的调用方式：

```
redis.pipeline().set('foo', 'bar').get('foo').exec(function (err  
, results) {  
});
```

代码 5.1.3 **pipeline** 链式调用

甚至可以在调用每个命令的时候都加一个回调函数，这里在 **get** 位置加一个回调函数：

```
redis.pipeline().set('foo', 'bar').get('foo', function(err, result  
) {  
    console.log('get foo', err, result);  
}).exec(function (err, results) {  
    console.log('with single callback', err, results);  
});
```

代码 5.1.4 **pipeline** 链式函数中加回调

当然这里还有一种更加简洁的调用方式，就是都把参数放到数组里：

```
redis.pipeline([
  ['set', 'foo', 'bar'],
  ['get', 'foo']
]).exec(function(err, results) {
  console.log('array params', err, results);
});
```

代码 5.1.5 pipeline 数组参数调用方式

multi 函数跟 pipeline 函数的区别是，multi 提供了事务的功能，提交到 redis 服务器的命令的会被依次执行，pipeline 则是批量执行一批提交一批指令，但是在 redis 内部都是独立执行的，没有先后顺序，只是最终服务器将所有处理结果一起返回给了调用者。不过要想完全保证事务的原子性，我们还需要使用 watch 函数，防止我们在事务中操作一个事务的过程中，当前操作的某一个键值又被其他连接的客户端给修改了：

```
redis.watch('foo');
redis.multi().set('foo', 'bar').get('foo').exec(function (err, results) {
  redis.unwatch();
  console.log('chain', err, results);
});
```

代码 5.1.6 multi 事务操作代码

最后一件需要重点指明的事情是，如果你当前使用了 cluster 方式连接 redis，那么最好不要使用 pipeline 和 multi 因为，ioredis 在调用这两个函数的时候，仅仅会往一个节点发送指令，但是你又不能保证你这里面操作的所有键值都在一个节点上，所以说调用这两个函数的时候很有可能会失败。

5.2 mongodb

[mongodb](#)官方提供了 Node.js 的 mongodb 驱动，不过鉴于其提供驱动的功能太过于简单，所以又涌现了许多基于官方驱动上开发的第三方驱动。下面要讲两个使用广泛的第三方驱动，[mongoskin](#) 和 [mongoose](#)。

5.2.1 mongoskin

我们这里先讲 **mongoose**，在介绍之前先讲清楚一个概念，传统关系型数据库中，有表的概念，**mongodb**有**collection**的概念，其实是同一种东西，我在这里仍然称呼**collection**为 表。

为了演示它的用法，我们先不在 **express** 中使用它，而是写个简单的测试函数。

```
var mongo = require('mongoose');
var db = mongo.db("mongodb://localhost:27017/live", {native_parser:false});
db.bind('article');
```

代码 5.2.1.1 初始化**mongoose**

在代码 5.2.1.1 中，可以看出我们创建了一个**mongo**连接，服务器地址为 **localhost** ,端口为 **27017** ,参数 **native_parser** 代表是否使用原生代码来解析**mongodb**的**bson**数据。如果开启这个选项，需要在安装 **mongoose** 模块的时候可以编译原生代码，如果你的开发环境是 **Windows**，且没有安装臃肿的**Visual Studio**的话，是没法编译原生代码的，那么这个参数就设置为 **false**。

注意最后一句 **db.bind('article')** 函数**bind**返回一个 **Collection** 对象，它在**db**对象上绑定一个 **article** 属性，指向刚才返回的对象，这句话在 **mongoose**中等同于 **db.article = db.collection('article');**（**collection** 其实为原生**mongodb**驱动里面获取一个 **Collection** 的封装函数），调用完 **bind** 函数后我们就可以通过 **db.article** 来操作一系列的增删改查了。

首先是插入单条数据：

```
db.article.insert({
  name: 'chapter5', content: 'Express.js 基础', createTime: new Date('2016/07/03')
}, function(err, ret) {
  console.log('单条插入', err, ret);
});
```

代码 5.2.1.2 **mongoose**插入单条数据

接着是插入多条数据，仅仅把第一个参数改成数组就可以了：

```
db.article.insert([
  {name:'chapter1',content:'Node.js 简介',createTime:new Date('2016/07/01')},
  {name:'chapter2',content:'Node.js 基础',createTime:new Date('2016/07/02')}
],function(err,ret) {
  console.log('插入数组',err,ret);
});
```

代码 **5.2.1.3 mongoskin**插入多条数据

修改单条数据：

```
db.article.update({name:'chapter2'},{
  $set:{content:'Node.js 入门'}
},function(err,ret) {
  console.log('更新单条数据',err,ret);
});
```

代码 **5.2.1.4 mogonskin**修改单条数据

注意第二个参数中需要有一个 `$set` 属性，否则整条数据将会被替换掉，如果直接将第二个参数写成了 `{content:'Node.js 入门'}`，则操作完成之后，数据库里当前记录就变成了 `{_id:主键值,content:'Node.js 入门'}`，之前的属性 `name` 和 `createTime` 就都丢失了。代码 **5.2.1.5 mongoskin**修改单条数据
如果想修改多条数据，只需要增加一个参数：

```
db.article.update({name:'chapter2'},{
  $set:{content:'Node.js 入门'}
},{multi:true},function(err,ret) {
  console.log('更新单条数据',err,ret);
});
```

相比较代码 **5.2.1.4** 这里多了一个参数，`{multi:true}` 告诉数据库服务器，要更新多条数据。代码 **5.2.1.6 mongoskin**修改多条数据
删除和更新相反，默认情况下是删除多条记录：

```
db.article.remove({name:'chapter1'},function(err,ret) {  
    console.log('删除数据',err,ret);  
});
```

代码 **5.2.1.7 mongoskin** 删除多条记录 如果想删除一条记录，则增加一个参数 `{justone:true}`，即改成：

```
db.article.remove({name:'chapter1'},{justone:true},function(err,  
ret) {  
    console.log('删除数据',err,ret);  
});
```

代码 **5.2.1.8 mongoskin** 删除单条记录 查询一条记录：

```
db.article.findOne({name:'chapter2'},function(err,item) {  
    console.log('查询单条数据',err,item);  
});
```

代码 **5.2.1.9 mongoskin** 查询单体记录

代码 5.2.1.7 中回调函数得到 `item` 变量即为查询后得到的记录。

查询多条记录：

```
db.article.findItems({},function(err, items) {  
    console.log('查询多条数据',err,items);  
});
```

代码 **5.2.1.10 mongoskin** 查询多条记录 上面演练了一遍mongoskin的增删改查，不过，我们将其和传统的关系型数据库做对比，发现还少了点东西。比如说：一条记录有多个字段，我只想返回若干字段怎么弄；表中的数据过多，想分页显示怎么弄；对于需要使用事务的情形，怎么弄。

对于前两个问题，只需要在查询的时候加参数即可。比如我们想查询7月1日都有哪些文章发布，我们只关心文章名称，同时由于数据量很大，无法全部显示出来，需要做分页，那么查询语句可以这么写：

```
db.article.findItems({
  createTime:{$gte:new Date(2016,6,1),$lt:new Date(2016,6,2)}
},{
  fields:{name:1,createTime:1},skip:1,limit:1,sort:{createTime:
-1}
},function(err, items) {
  console.log('查询多条数据',err,items);
});
```

代码 5.2.1.11 mongoskin自定义查询选项

在这里我们通过 `fields` 来控制返回字段名，`skip:1` 代表跳过第一条记录，`limit:1` 代表从跳过的记录后面取1条记录，同时我们还增加了 `sort` 属性，按照 `createTime` 字段的倒序排列。

对于最后一个问题，很遗憾，`mongodb`中确实没有事务，不过它还是提供了一个带有锁功能的操作函数，就是 `findAndModify`。这个函数的参数比较多，下面直接给出参数列表：

函数声明

`findAndModify(query, sort, update , options , callback)`

参数声明

- `query{Object}`查询条件
- `sort{Array}`排序
- `update{Object}`更新的内容
- `option`

```
{
  new:{Boolean} 是否返回更新后的内容,默认为false
  upsert: {Boolean} 不存在时是否插入,默认为false
  remove : {Boolean} 查询到结果后将其删除，此字段优先级高于 upsert,默认为false
  fields:{Object} 指定查询返回结果中要返回的字符，默认为null
}
```


- `callback{Function}` 此回调函数有两个参数：
 - `error {Error}` 错误对象，成功时为`null`
 - `result {Object}`

```
{
  value : {Object} 函数findAndModify返回的数据记录
  lastErrorObject : { updatedExisting: true, n: 1, connectionId: 14, err: null, ok: 1 } 大体格式是这样的
  ok : {Number} 成功返回1
}
```

举个例子，现在有一张表`comment`，它的数据结构是这样的：

```
{
  "_id" : ObjectId("5792f2db03d07723cff9ab35"),
  "author" : "1vglr42hwqb",
  "content" : "twicgcbk7xd",
  "articleId" : ObjectId("5792288c0c0c422b282f2f93"),
  "createTime" : 1468304429853.0000000000000000
}
```

现在我们将使用函数 `findAndModify` 查询文章ID

为 `ObjectId("5792288c0c0c422b282f2f93")` 最近的一条评论，将其 `content` 字段改成 评论已被删除：

```
db.comment.findAndModify({
  articleId:mongo.helper.toObjectID("5792288c0c0c422b282f2f93")
},{
  'createTime':-1},{
  $set:{content:'评论已被删除'}},{
  fields:{author:1,content:1},new:true,upsert:false,remove:false
},function(err, result) {
  console.log('修改后数据',err,result);
});
```

代码 5.2.1.12 函数findAndModify演示

但是要注意，`findAndModify` 会同时持有读写锁，也就是在这个函数操作过程中，其他命令是会被堵塞住，一直等待这个函数操作完成或者出错，其他命令才会有机会接着执行。所以一般情况下，不要使用这个函数，除非是非常关键的数据，要求精度很高才使用这个函数，比如说订单状态修改之类的操作，但是就像 代码 5.2.1.10 之中的操作，纯属拿大炮打蚊子了。即使非要用到这个函数的情况，也尽量保证查询的时候可以使用索引，尽量减少锁持有的时间。

对于一些非关键性数据，但是又必须保证某个字段在写入的时候保持唯一性，那么在这个字段上增加一个唯一索引，就可以了。在mongodb的命令行中执行如下命令：

```
db.collectionName.ensureIndex({fieldName:1},{unique:true});
```

命令 5.2.1.1 创建唯一索引

之前已经提过，`mongoskin`只不过是对与原生mongodb node驱动的封装，其基于 `collection` 的操作函数的各个参数都是通用的，另一方面`mongoskin`官方给出的API文档并不详尽，所以如果想要了解详细的各个操作函数的说明，可以参考[mongodb node驱动的官方文档](#)。

5.2.2 mongoose

前面讲了 `mongoskin`，算是 `mongodb` 知识点的开胃菜，`mongoskin` 中的函数绝大部分和 `mongodb` 命令行是类似的。下面要讲的 `mongoose` 却稍有不同，因为其有一个 ODM (Object Data Model) 的概念，类似于 `hibernate` 开发中用到的 `ORM (Object Relational Mapping)` 的概念，它提供了一种将 `mongodb` 中字段映射为 JavaScript 对象属性的能力。如果我们用 `mongoose` 来实现一系列的增删改查操作，就必须先定义一个 `Schema`，不过下面要先讲怎样在 `mongoose` 中建立连接，否则接下来的例子就没法运行了：

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/live', {/*user:'username',
pass:'password'*/}); // connect to database
```

代码 5.2.2.1 mongoose 建立连接代码

在 `mongoose` 中使用 `connect` 函数可以初始化 `mongodb` 连接，第一个参数代表 `mongodb` 的连接字符串，第二个参数存放连接控制参数，比如说用户名、密码之类的。其实第一个字符串中有更多连接参数控制，可以参考 `mongodb` 的 [官方文档](#)，其中就包括用户名和密码信息（格式

为 `mongodb://username:password@host:port/database?options...` ），但是如果你的密码中有特殊字符的话（比如说 `@` ），就比较难办了，所以将用户名和密码放到第二个参数中比较保险。

接下来就将 `mongoose` 中非常之重要的 `Schema`，首先直接构造一个我们在 5.2.1 小节中使用过的 `article` 的 `schema` 声明：

```
var mongoose = require('mongoose');
require('./conn');//代码5.2.2.1对应的代码

var Schema = mongoose.Schema;

var articleSchema = new Schema({
  name: String,
  content: String,
  comments: [{ body: String, date: Date }],
  create_at: { type: Date, default: Date.now }
});
var Article = mongoose.model('article', articleSchema);
```

代码 5.2.2.2 声明 `Schema`

通过以上代码可以总结出 `shema` 干的事情就是把数据库的各个字段的数据类型定义出来，最后我们还通过 `model` 函数获得了一个 `mongoose` 中的 `Model` 类，`mongoose` 的增删改查都通过这个类来进行。注意第一个参数代表表名。

```
new Article({
  name: 'chapter5',
  content: 'Express.js 基础',
  comments : [
    {body: '写的不多', date: new Date('2016-10-11')},
    {body: '我顶', date: new Date('2017-01-01')}
  ],
  create_at: new Date('2016/07/03')
}).save(function(err, item) {
  console.log(err, item);
});
```

代码 5.2.2.3 mongoose 插入操作

为啥说 `model` 函数得到的是类呢，通过 代码 5.2.2.3 就可以看出，我们通过 `new` 生成一个对象实例，然后调用其 `save` 函数将其插入数据库。如果我们将

`create_at` 属性去掉，那么其值就会自动取当前时间。不过等你执行完上述代码后，查看数据库，咦，**surprise**，数据库里竟然多了一个名字叫 **articles** 的表，不是说 `model` 的第一个参数是执行关联的表明吗，明明在 代码 5.2.2.2 中指定的表明是 **article** 啊？是的，不要惊讶，**mongoose** 默认就是这么设计的，如果你想绑定到一个自定义的一个表明上，可以在实例化 **Schema** 的时候，传入一个可选参数：

```
var articleSchema = new Schema({/*此处省略字段定义*/},{collection: 'article'});
```

这样将 `articleSchema` 插入 `model` 后得到的 `Article` 就绑定表 `article` 上了。

说了插入单条，再说一下批量插入，这时候使用 `insertMany` 函数即可：

```
Article.insertMany([
  {name:'chapter1',content:'Node.js 简介1',create_at:new Date('2016/07/01')},
  {name:'chapter1',content:'Node.js 简介2',create_at:new Date('2016/07/01')},
  {name:'chapter1',content:'Node.js 简介3',create_at:new Date('2016/07/01')},
  {name:'chapter2',content:'Node.js 基础4',create_at:new Date('2016/07/02')},
  {name:'chapter2',content:'Node.js 基础5',create_at:new Date('2016/07/02')}
],function(err,ret) {
  console.log('插入数组',err,ret);
});
```

代码 **5.2.2.4 mongoose** 批量插入操作

mongoose 的修改操作和官方 API 差不多：

```
Article.update({name:'chapter2'},{
  $set:{content:'Node.js 入门'}
},function(err,ret) {
  console.log('更新单条数据',err,ret);
});
Article.update({name:'chapter2'},{
  $set:{content:'Node.js 入门'}
},{multi:true},function(err,ret) {
  console.log('更新多条数据',err,ret);
});
```

代码 **5.2.2.5 mongoose** 修改操作

不过它的删除稍微有些不同，就是删除的时候仅仅只能指定一个查询参数，如果你想仅仅删除一条的话，那就需要先查询出来，然后再删除。

```
Article.findOne({name: 'chapter1'}).remove().exec(function(err, ret) {
  console.log('删除数据', err, ret);
});
Article.remove({name: 'chapter1'}, function(err, ret) {
  console.log('删除数据', err, ret);
});
```

代码 5.2.2.6 mongoose 删除操作

上面总结了一下 mongoose 的一些基本用法，不过前面的描述还不足以体现 mongoose 的强大，下面讲到的一些高级用法，绝对能让你感到惊艳。

首先 mongoose 提供了中间件（middleware）的功能，我们可以在执行数据命令前和执行后添加钩子函数，先上代码：

```
var mongoose = require('mongoose');
require('./conn');//代码6.2.2.1对应的代码

var Schema = mongoose.Schema;

var articleSchema = new Schema({
  name: String,
  content: String,
  comments: [{ body: String, date: Date }],
  create_at: { type: Date, default: Date.now }
});

articleSchema.pre('save',function(next) {
  this.content = this.name + '\n' + this.content;
  next();
});

articleSchema.post('save', function(doc) {
  console.log('%s has been saved', doc._id);
});

var Article = mongoose.model('article', articleSchema);

new Article({
  name: 'chapter5',
  content: 'Node 中使用数据库',
  comments : [
    {body: '写的不多', date: new Date('2016-10-11')},
    {body: '我顶', date: new Date('2017-01-01')}
  ],
  create_at: '2017-02-11'
}).save(function(err,item) {
  console.log(err,item);
});
```

代码 5.2.2.7 save 的中间件函数演示

我们创建了一个 `article` 的 `schema` 定义，同时定义了两个中间件。通过 `pre('save')` 操作，我们在文章的第一行拼接了文章的标题，然后注意一定要调用 `next` 函数，否则当前数据库操作就不会得到执行。通过 `post('save')` 操作用来在数据库操作完成之后执行一些级联操作，这里我们简单的打印了一下日志。这两个中间件函数会先于 `save` 函数的回调函数前执行。

在调用 `save` 函数时，`mongoose` 中还提供了一个 `validate` 中间件，他会在 `pre('save')` 之前被触发，用来校验传入 `save` 函数的各个属性是不是合法：

```
articleSchema.pre('validate',function(next) {
  if (/<script>/.test(this.content)) {
    return next(new Error('文章内容非法'));
  }
  next();
});
new Article({
  name: 'chapter5',
  content: 'Node 中使用数据库<script>alert(document.cookie)</script>',
}).save(function(err,item) {
  console.log(err,item);
});
```

代码 5.2.2.8 `save` 的 `validate` 中间件函数演示

上面的代码执行后，会抛出异常，因为我们的 `article content` 字段中包含 `script` 标签。令人欣喜的是，`mongoose` 还提供将 `validate` 中间件直接加到 `schema` 定义上的功能：

```
var mongoose = require('mongoose');
require('./conn');//代码6.2.2.1对应的代码

var Schema = mongoose.Schema;

var articleSchema = new Schema({
  name: {
    type:String,
    required: [true, '必须提供文章标题'],
    maxlength : [50, '文章标题不能多于50个字符']
```



```
},
isbn : {
  type:String,
  unique:true,
  sparse: true
},
content: {
  type:String,
  validate:{
    validator : function() {
      return !(</script>/.test(this.content));
    },
    message : '文章内容非法'
  }
},
starts : {
  type:Number,
  min:0,
  max:[5, '最多只能给5颗星'],
  default:0
},
level : {
  type:String,
  enum:['专家推荐', '潜力无限', '家有作家初长成', '我只是个小学生']
},
category : {
  type:String,
  enum:{
    values:['诗歌', '散文', '杂文', '议论文', '小说'],
    message:'当前标签不支持'
  }
},
cover_url : {
  type:String,
  match:[/^http(s?):\/\//, '封面图格式非法']
},
comments: [{ body: String, date: Date }],
create_at: { type: Date, default: Date.now }
});
```

```
articleSchema.pre('save',function(next) {
    this.content = this.name + '\n' + this.content;
    next();
});

articleSchema.post('save', function(doc) {
    console.log('%s has been saved', doc._id);
});

var Article = mongoose.model('article', articleSchema);

new Article({
    name:'chapter5',
    content:'Node 中使用数据库<script>alert(document.cookie)</script>',
}).save(function(err,item) {
    if (err && err.name === 'ValidationError') {
        for (var field in err.errors) {
            var error = err.errors[field];
            console.error(error.message,error.path,error.value);
        }
    }
});
```

代码 5.2.2.9 在 schema 中使用校验器

mongoose 内建了好多校验器 (validator)，多余所有类型字段来说都可以使用 **required** 校验器，对于 Number 类型字段来说，可以使用 **min** 和 **max** 校验器，对于 String 类型字段来说，可以使用 **enum match maxlength minlength** 校验器。

所有校验器都可以设置在校验失败后的错误提示信息，如果相对某一个字段设置 **required** 约束，那么可以写成 `required:true`，还可以进一步指定校验失败后的提示信息，也就是写成这样 `required:[true, '这个字段必须指定']`。但是对于 **enum** 来说，由于本身定义的时候就是一个数组结构（参见上面代码中 `level` 字段的定义），所以 mongoose 内部在定义其 **message** 属性时使用这样一个 Object 结构：`{values:[/*枚举字段定义*/],message:'出错提示信息'}`。

还记得在代码 **5.2.2.8** 中我们自定义的那个 `content` 字段的校验中间件不？这个中间件可以直接写到 `schema` 定义中，在代码 **5.2.2.9** 中的 `content` 字段中的 `validate` 属性，就能替换掉之前我们写过的校验中间件。

最终你在调用 `save` 函数之前，这层层字段定义约束都会被执行，如果校验出错，那么 `save` 回调函数返回的第一个参数中的 `name` 属性的值将是

`ValidationError`，让你后其 `errors` 属性中保存着字段的详细信息的一个 `key-value` 数据结构，键名是出错的字段名，值是一个包含错误详情的对象，这个对象中 `message` 属性就是我们在 `schema` 中设置的出错信息，`path` 是出错的字段名，`value` 是引起出错的具体的设置的值。

最终需要注意，`unique` 这个约束并不是一个 `ValidationError`（实际上其 `name` 属性值为 `MongoError`），所以你 `save` 失败后得到的 `error` 对象中没有 `errors` 属性。`unique` 和 `sparse` 仅仅是 `schema` 调用 `mongodb` 的驱动创建了数据库索引而已。代码 **5.2.2.9** 中关于 `isbn` 的约束，也可以通过 `schema` 中的 `index` 函数来实现：

```
articleSchema.index('isbn',{unique:true,sparse:true});
```

代码 5.2.2.10

前面讲了许多 `mongoose` 的插入、修改之类的操作，一直没有提到查询操作，下面就来讲一下查询。

在讲查询之前，需要先将我们在代码 **5.2.2.9** 中定义的 `articleSchema` 进行一下扩充，增加下面这个字段：

```
_author : {type:Schema.Types.ObjectId,ref:'user'},
```

代码 5.2.2.11

至于其中的 `_ref` 属性是怎么回事，我们先买个关子，一会儿再说。

`mongoose` 在查询方面，有好多细节做了优化，比如说在筛选返回字段的时候可以直接通过字符串来指定：

```
Article.findOne({name:nameRand}, 'name -_id', function(err,item) {
  if (err) {
    return console.error('findOne',err);
  }
  console.log('findOne',item && item.name === nameRand);
});
```

代码 5.2.2.12 mongoose 查询使用字符串筛选字段

mongoose 的查询中的各个控制参数都可以链式的调用各个函数来解决，比如说上例中用到的字段筛选可以使用 `select` 函数来替代，即改成

```
Article.findOne({name:nameRand}).select('name -_id').exec(function(err,item) {});
```

当中可以添加无数个链式函数来控制查询行为，比如说 `limit skip lean` 等等，最后以 `exec` 函数结尾添加回调函数。

mongoose 查询默认返回的是 `MongooseDocuments` 类型对象，使用 `lean` 函数后可以将其转成普通 javascript 对象：

```
Article.find({name:/^name/}).select('_author').lean().exec(function(err,items) {
  if (err) {
    return console.error('find',err);
  }
  console.log('find',items);
});
```

代码 5.2.2.13 mongoose 查询返回纯 javascript 对象

转纯 javascript 对象的使用场景一般比较少见，当我们拿查询的结果作为参数来调用一些第三方库（比如说 `protobufjs`）时，不调用 `lean` 的情况下会出错。

最后还要暴一下 mongoose 中的大杀器，就是联合查询，其实 mongdb 本身是没有联合查询功能的，这个功能是在 mongoose 层面延伸的功能：

```
Article
  .findById(articleId)
  .select('name _author')
  .populate('_author', 'nickname -_id')
  .exec(function(err, item) {
    if (err) {
      return console.error('findById', err);
    }
    console.log('populate', item);
  });
```

代码 5.2.2.14 mongoose 联合查询功能

还记得我们在代码 5.2.2.10 中卖的关子不，我们看到其中有一个 `_ref` 属性，它的作用就是告诉 mongoose `_author` 字段的值对应 `users` 表中的主键字段，如果在查询的时候使用 `populate` 函数，则 mongoose 将在底层做两次查询（查询 `articles` 表和 `users` 表），然后把查询结果合并。最终得到的结构演示如下：

```
{ name: 'name0.6169953700982793',
  _author: { nickname: 'nick0.09724390163323227' },
  _id: 5916e9178be9f133b4798002 }
```

5.3 代码

本章代码参见这里：<https://github.com/yunnysunny/nodebook-sample/tree/master/chapter5>

6 Express 介绍

提到node，那么就不得不提大名鼎鼎的 [express](#)，作为一个web framework，它几乎满足了你所有的愿望。本篇的内容主要讲述express的基本使用。

6.1 Express 安装

当然作为一个web framework，必然要牵扯到各种配置。聪明人肯定不是吧所有配置代码从头到尾敲出来，这就要提到 [express-generator](#)。首先运行

```
npm install -g express-generator
```

来安装，这里用 `-g` 参数来将其安装为全局位置，因为这个样子我们就能将其安装后生成的可执行程序添加到环境变量中了。

接着运行 `express -e first-app && cd first-app`，其中命令中 `-e` 参数是说使用 [ejs](#)模板引擎来渲染视图。`first-app` 就是我们生成程序生成的目录，紧接着我们通过 `cd` 命令进入了这个目录。最后我们运行 `npm install` 命令来安装所需依赖。最终在图形化界面中进入这个目录，会看到如下文件列表：

```
--bin
----www
--public
----images
----javascripts
----stylesheets
--routes
--views
--app.js
--package.json
```

目录 6.1.1

6.2 Express 基本操作

express的所有配置信息在app.js中：

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico'))
);
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
app.use('/users', users);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handlers

// development error handler
// will print stacktrace
```

```
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});

module.exports = app;
```

代码 6.2.1 app.js

express处理网络请求，是以一种被称之为 `middleware`(也翻译为中间件) 机制进行的，即网络请求先经过第一个`middleware`，如果处理完成则直接返回，否则调用 `next()` 函数将当前请求丢给下一个`middleware`进行处理。我们看到`app.js`中有很多 `app.use` 函数的调用，正是这个函数配置了一个个的`middleware`。

其中 `app.use(bodyParser.json());` 处理请求头为 `application/json` 的数据，其实这个`middleware`一般用不到；`app.use(bodyParser.urlencoded({ extended: false }));` 这句话是处理`form`表单数据的，这个用处就大了。`app.use(cookieParser());` 是用来处理`cookie`的，没有这个`middleware`的话，无法读取`http`请求中的`cookie`数据。

据。`app.use(express.static(path.join(__dirname, 'public')));` 是定义从 `public` 路径读取静态文件。之前讲过当前项目目录中存在 `public` 文件夹，假设我们在其下 `javascripts` 目录中放置一个 `query.js` 文件，那么我们在`html`中就可以这么引用这个静态文件：


```
<script type="text/javascript" src="/javascripts/jquery.js">
</script>
```

其他静态文件依次类推。

如果是做网站项目的话，还缺少对于session的支持，这个要在后面单独讲。接下来是很重要的路由映射部分，因为项目中的url映射都是在这里配置的。我们这里只看

routes/index.js 文件：

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

代码 6.2.2 routes/index.js

在express 3.x中，定义路由需要使用到在 app.js 中定义的 app 对象，写成 app.get('/path', function(req, res){}) 的样式，不过经过本人测试在express 4.x中依然可用。如果采用express 3.x的编写方式，那么 routes/index.js可以写成这样：

```
module.exports = function (app) {
  app.get('/', function(req, res) {
    res.render('index', { title: 'Express' });
  });
}
```

对应app.js中，需要将 app.use('/', routes); 替换成 routes(app); 。两种写作手法而已，看个人喜好。

我们看到这里仅仅只有一个根路径的映射，假设我们网站上还有一个 /about 的路径。那么就可以在 index.js 中再追加一条：

```
router.get('/about', function(req, res) {  
  res.render('index', { title: 'about' });  
});
```

代码 6.2.3

app.js中定义的 `app.use('/', routes);`，其实其中的 `/` 仅仅是定义路由的路径前缀而已，从这种意义上来讲，`routes/index.js` 和 `routes/user.js` 的代码是可以合并的。我们删除 `user.js` 文件，然后在`index.js`中追加一段代码：

```
/* GET users listing. */  
router.get('/user', function(req, res) {  
  res.send('respond with a resource');  
});
```

代码 6.2.4

这样我们就可以删除掉 `app.use('/users', users);` 了。其实如果看完上述关于路由器的介绍，熟悉express 3的用户会发现，除了语法和3.x不一样以外，功能上没啥不同。不过事实并非如此，`index.js` 中的`router`对象还可以直接用来定义`middleware`，我们在 `index.js` 开头再添加一段代码：

```
// middleware specific to this router  
router.use(function timeLog(req, res, next) {  
  console.log('Time: ', Date.now());  
  next();  
});
```

代码 6.2.5

那么上述代码定义的这个`middleware`就仅仅对 `index.js` 内部定义的地址起作用，对于这个路由器文件外的代码是不起作用的，这个设计就比较灵活了。之前咱们在 `app.js` 中通过 `app.use` 来定义`middleware`，那么理论上所有的请求都要经过这种`middleware`进行处理的，除非在经过这个`middleware`之前，已经有其他的`middleware`把HTTP请求处理完成了。

最后看错误捕获这一块了，`app.js` 中对于代码捕获区分了两种情况，如果当前是开发环境就在出错的时候打印堆栈，否则只显示错误名称。我们现在修改一下

`/user` 的路由代码：

```
router.get('/user', function(req, res) {  
  console.log(noneExistVar.pp);  
  res.send('respond with a resource');  
});
```

代码 6.2.6

接着运行项目（关于如何运行项目，将在下面讲到），然后在浏览器中打开 <http://localhost:3000/user>，浏览器直接显示错误堆栈：

```
noneExistVar is not defined
```

```
ReferenceError: noneExistVar is not defined
```

```
    at D:\code\eapp\first-app\routes\index.js:15:14  
    at Layer.handle [as handle_request] (D:\code\eapp\first-app\  
node_modules\express\lib\router\layer.js:95:5)  
    at next (D:\code\eapp\first-app\node_modules\express\lib\rou  
ter\route.js:131:13)  
    at Route.dispatch (D:\code\eapp\first-app\node_modules\expre  
ss\lib\router\route.js:112:3)  
    at Layer.handle [as handle_request] (D:\code\eapp\first-app\  
node_modules\express\lib\router\layer.js:95:5)  
    at D:\code\eapp\first-app\node_modules\express\lib\router\in  
dex.js:277:22  
    at Function.process_params (D:\code\eapp\first-app\node_modu  
les\express\lib\router\index.js:330:12)  
    at next (D:\code\eapp\first-app\node_modules\express\lib\rou  
ter\index.js:271:10)  
    at Function.handle (D:\code\eapp\first-app\node_modules\expr  
ess\lib\router\index.js:176:3)  
    at router (D:\code\eapp\first-app\node_modules\express\lib\r  
outer\index.js:46:12)
```

输出 6.2.1

这说明，程序默认走到 `app.get('env') === 'development'` 这个条件中去了。 `app.get('env')` 其实是读取的环境变量 `NODE_ENV`，这是一个express专用的环境变量，express官方推荐在生产环境将其设置为 `production`（参考[这](#)

里) 后会带来三倍的性能提升。官方推荐使用 `systemd` 或者 `Upstart` 来设置环境变量, 不过如果你的程序不是开机自启动的话, 直接配置 `.bash_profile` 文件即可, 也就是说直接在该文件中添加 `export NODE_ENV=production`。

6.6 模板引擎

在代码6.2.2中遇到了一个 `render` 函数, 这个函数就是`express`中用于加载模板的函数。通过代码也可以大体看出, 第一个参数是模板的名字, 它所代表的文件位于视图文件夹 `views` 目录下的 `index.ejs` (`ejs` 文件后缀是 `ejs` 模板引擎的默认后缀); 而第二个参数即为传递给这个模板的参数。接着看一下在模板中, 是怎样使用刚才传递的那个 `titile` 参数的, 打开 `views` 文件夹下的 `index.ejs` :

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

可以看到使用`<%=titile%>`的方式就可以把之前`render`函数中传递的`title`参数读取出来。扩展一下在`ejs`中还有两个常见的标签: `<%- %>`:读取变量中的值且对于变量中的`html`特殊符号(比如`<`、`>`、`&`、`"`等)不进行转义, 如果使用`<%= %>`就会把特殊符号转义, `<%%>`:写在这个标签里的语句会直接当成代码来解析, 比如说如下代码:

```
<% if (status == 0) { %>
<input type="button" value="启用" />
<% } else { %>
<input type="button" value="禁用" />
<% } %>
```

6.7 Express 中的GET和POST

接下来的内容来讲一下express中怎样使用get和post，首先我们在views文件夹下新建目录user,然后在user目录下新建文件sign.ejs(当然你也可以把它当成静态页，放到public中；但是正常环境下，对于html一般都是通过视图的方式来加载)。

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=
utf-8" />
  <title>Node.js注册演示</title>
</head>
<body>
  <h1>注册</h1>
  <form id="signup" method="get" action="/users/do/sign">
    <label> 帐号 : </label><input type="text" name="username" />
    <label> Email : </label><input type="text" name="email" />
    <input type="submit" value="注册" /><br>
  </form>
</body>
</html>
```

代码6.7.1 sign.ejs代码

这里表单method是get（虽然一般情况下网服务器添加数据都是用post方式，但是这里为了演示方便，现将其写成get）。接下来看一下express中怎样在GET方式下获取表单中的数据。为了演示用户注册这个流程，我们新建 controllers 目录，在里面创建user_controller.js文件：

```
exports.showSign = function(req, res) {
  res.render('user/sign');
}

exports.doSign = function(req, res) {
  var name = req.query.name;
  var email = req.query.email;
  res.send('恭喜' + name + '注册成功，你的邮箱为:' + email);
}
```

代码6.7.2 user_controller.js文件中处理函数

web编程中广泛使用MVC（模型Model、视图View、控制器Controller，这三个单词的缩写）的设计模式，在项目创建 `controllers` 正是为了符合这一模式，同时你还需要创建一个`models`文件夹，专门负责处理数据。具体的使用流程是这样的：`controllers` 里面放置请求处理的代码，即接收请求参数，对其进行有效性校验，然后调用 `models` 里面的代码进行数据操作（比如说数据库的增删改查等操作），拿到处理结果后加载视图进行渲染。关于MVC的介绍，可以参见[维基百科](#)。

然后添加相应的路由如下：

```
router.get('/users/sign', user.showSign);
router.get('/users/do/sign', user.doSign);
```

代码6.7.3 新增路由配置

运行 `npm start`，即可查看效果，打开<http://localhost:3000/users/sign>，可看到如下界面：

输入数据后，点击注册，显示提示信息：

这就完成了`get`操作，但是前面提到了类似于这种注册操作一般都是用`post`的，将上面的代码改成`post`是很简单的，只需在代码代码6.7.1 中将表单的`method`改成`post`，代码6.7.2中获取请求数据是这么写的：

```
var name = req.query.name;  
var email = req.query.email;
```

如果改成post，只需将其改为

```
var name = req.body.name;  
var email = req.body.email;
```

6.8 Express AJAX 应用示例

还是上面的例子，只不过这次换成用ajax来提交数据，我们在views/user文件夹下再新建文件sign2.ejs：

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=
utf-8" />
  <title>Node.js注册演示</title>
  <script language="javascript" src="/javascripts/jquery-1.10.
2.js"></script>
</head>
<body>
<h1>注册</h1>
<form id="signup" method="post" action="/users/sign2">
<label>帐号 : </label><input type="text" name="name" /><br />
<label>Email : </label><input type="text" name="email" /><br
/>
<input type="submit" value="注册" /><br>
</form>
<script language="javascript">
  $(document).ready(function() {
    $('#signup').submit(function() {
      $.post($(this).attr('action'),$(this).serialize(
),function(result) {
        if (result.code == 0) {
          alert('注册成功');
        } else {
          if (result.msg) {
            alert(result.msg);
          } else {
            alert('服务器异常');
          }
        }
      }, 'json')
      return false;
    });
  });
</script>
</body>
</html>
```


代码6.8.1 sign2.ejs

为了使用ajax，我们引入了jquery，并将jquery-1.10.2.js放到了public/javascripts文件夹下，为了演示ajax和普通请求处理的区别，这里仅仅给出处理post请求的代码：

```
exports.doSign2 = function(req, res) {
  var name = req.body.name;
  var result = {};
  if (!name) {
    result.code = 1;
    result.msg = '账号不能为空';
    res.send(result);
    return;
  }
  var email = req.body.email;
  if (!email) {
    result.code = 2;
    result.msg = '邮箱不能为空';
    res.send(result);
    return;
  }
  res.send({code : 0});
}
```

代码6.8.2 ajax后台处理代码

express中res的send函数中传一个json对象，则发送给浏览器的时候会自动序列化成json字符串。

我们继续添加两个路由：

```
router.get('/users/sign2', user.showSign2);
router.post('/users/do/sign2', user.doSign2);
```

代码6.8.3 ajax相关路由

重启项目后访问地址<http://localhost:3000/users/sign2> 即可进行测试。

6.9 代码

本章用的部分代

码：<https://github.com/yunnysunny/expressdemo/tree/master/chapter6>

7 Express 进阶

第5章讲了Express的入门知识，这一节趁热打铁要讲一下高级技术。

7.1 使用 session

对于一个网站来说，一个不可避免的问题就是用户登录，这就牵扯到 session 问题了。为此我们需要在app.js中引入两个middleware，[cookie-parser](#)和[express-session](#)，上一章的代码6.2.1已经介绍过cookie-parser，接下来重点介绍一下 `express-session`，在app.js中添加如下代码：

```
var cookieParser = require('cookie-parser');
var session = require('express-session');
var RedisStore = require('connect-redis')(session);
var redis = require('redis');

app.use(cookieParser());//
app.use(session({
  secret: 'GG##@$',
  cookie:{domain:'localhost'},
  key:'express_chapter6',
  resave:false,
  saveUninitialized:false,
  store: new RedisStore({
    client:redis.createClient(6379, '127.0.0.1');,
    ttl:3600*72,
    db:2,
    prefix:'session:chapter6:'
  })
}));
```

代码 7.1.1 引入session

其实express默认的session是存储在内存里的，但是这种做法不适合在生产环境使用，首先node如果使用cluster模式的话，内存无法共享，也就是说只能使用单进程；其次，如果在线人数一直增多的话，会造成内存猛增。所以这里的 `store` 参

数使用了redis，这同样也意味着你需要在你本地（或者远程机器上）启动redis服务，否则程序会报错。session 函数的 key 参数代表生成cookie的名称。resave 参数设置默认为 true，代表每次请求结束都会重写 store 中的数据，不管当前的session有没有被修改。saveUninitialized 参数值默认为 true，代表将未赋值过的session写入到store，也就是说假设我们的网站需要登录，那么在未登陆之前，也会往 store 写入数据。所以我们将 resave 和 saveUninitialized 都设置为了false。

如果你对 session 原理不是很清楚的话，可以参见我的博文 [session的安全性](#)，里面提到了session的基本原理，安全性及攻击防范。

为了减少篇幅，给出的代码都是片段形式：

```
<form method="post" action="/user/login" id="loginForm">
  <p><input name="username" /><label for="username">用户名</label><p/>
  <p><input name="password" type="password" /><label for="password">密码</label><p/>
  <p><input type="submit" value="登陆" /><p/>
</form>
<script src="//upcdn.b0.upaiyun.com/libs/jquery/jquery-1.10.2.min.js" type="text/javascript"></script>
<script type="text/javascript">
  $(document).ready(function() {
    $('#loginForm').submit(function() {
      var $this = $(this);
      $.ajax({
        method:$this.attr('method'),
        url:$this.attr('action'),
        data:$this.serialize(),
        dataType:'json'
      }).done(function(result) {
        if (result.code == 0) {
          return location.href = '/user/admin';
        }
        alert(result.msg || '服务器异常');
      }).fail(function() {
        alert('网络异常');
      });
      return false;
    });
  });
</script>
```

代码7.1.2 登陆前端代码

```
exports.login = function(req, res) {
  var _body = req.body;
  var username = _body.username;
  var password = _body.password;
  if (username === 'admin' && password === 'admin') {
    req.session.user = {account:username};
    return res.send({code:0});
  }
  res.send({code:1,msg:'用户名或者密码错误'});
}
```

代码7.1.3 登陆后端代码

在 代码7.1.3 中通过 `req.session.user` 来给session增加一个user的属性，在 代码7.1.2 中登陆成功后要跳转到 `/user/admin` 地址上去，我们接下来看这个地址映射的后端代码：

```
exports.admin = function(req, res) {
  var user = req.session.user;
  res.render('user/admin', {user:user});
}
```

代码 7.1.4 读取session

通过 `req.session.user`，就可以方便的将之前存储的 `user` 属性给读取出来。

7.3 改善我们的登陆

在7.2中，我们已经讲述了怎样使用mongodb了，下面就有机会对于我们7.1中提到的登陆进行改进了。虽然登陆在前端页面登陆仅仅是一个表单，但是在后台处理的流程就不是那么简单。这次我们决定读取数据库中的账号信息进行登陆验证，为此我们创建文件夹 `models`，在其内新建文件 `user_model.js`：

```
var crypto = require('crypto');
var collection = require('./index');
var users = collection.users;

var passwordValid = exports.passwordValid = function(passwordInput, passwordDb) {
    return crypto.createHash('sha256').update(passwordInput).digest('base64') === passwordDb;
}

exports.loginCheck = function(username, password, callback) {
    users.findOne({account:username}, function(err, item) {
        if (err) {
            console.error('查询用户时失败', err);
            return callback('查询用户时失败');
        }
        if (!item) {
            return callback('当前用户不存在');
        }
        if (!passwordValid(password, item.passwd)) {
            return callback('用户名或者密码错误');
        }
        item.passwd = undefined;
        callback(false, item);
    });
};
```

代码 7.3.1 登陆验证逻辑

其实单纯用md5/sha1/sha256这些算法来说，都存在被破解的可能性，国内有网站<http://cmd5.com>几乎可以破解一切弱密码，解决方案就是使用更长的密码（可以通过用户名和密码进行拼接来计算哈希值），或者使用hmac算法。这里为了演示，使用了最简单的方式。

那么现在控制器中的代码就可以这么写：

```
exports.loginWithDb = function(req, res) {
  var _body = req.body;
  var username = _body.username;
  var password = _body.password;
  if (!username) {
    return res.send({code:1,msg:'用户名不能为空'});
  }
  userModel.loginCheck(username,password,function(err,item) {
    if (err) {
      return res.send({code:1,msg:err});
    }
    req.session.user = item;
    res.send({code:0});
  });
};
```

代码 7.3.2 登陆验证控制器代码

我们在路由器中增加一个链接 `/user/login-with-db` 指向代码 7.3.2 中的控制器函数，修改代码 7.1.2 中的表单提交地址即可。

7.4 使用拦截器

之前的章节中介绍过 **express** 的 **middleware**，翻译一下就是中间件，下面的内容其实是做一个中间件，但是为啥我给它起名叫拦截器呢，因为我认为对于业务逻辑处理叫拦截器更贴切，因为我理解的 **middleware** 仅仅负责解析 **http** 数据，不处理业务逻辑。仅仅是个人见解。

之前我们已经做了登陆操作，但是对于一个网站的若干地址（比如说后台地址），不登录是没法用的，我们需要用户在加载这些地址的时候，如果检测到当前处于未登录状态，就统一跳转到登陆页。在每一个控制器中都做一遍登陆状态判断来决定是否跳转，显然是一个笨拙的方法。但是如果使用了拦截器，一切问题就显得简单了。

我们新建文件夹 `filters`，然后在其内新建文件 `auth_filter.js`：


```
const ERROR_NO_LOGIN = 0xffff0000;
module.exports = function(req, res, next) {
  var path = req.path;
  if (path === '/' || path === '/user/login') {//这些路径不需要做
    登陆验证
    return next();
  }
  if (req.session && req.session.user) {//已经登陆了
    return next();
  }
  //以下为没有登陆的处理逻辑
  if (req.xhr) {//当前请求为ajax请求
    return res.send({code:ERROR_NO_LOGIN,msg:'尚未登陆'});
  }
  res.redirect('/');//普通请求
};
```

代码 7.4.1 授权拦截器逻辑

同时我们在 `app.js` 中引入这段代码：

```
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var session = require('express-session');
var RedisStore = require('connect-redis')(session);
var redis = require('redis');

var routes = require('./routes/index');
var authFilter = require('./filters/auth_filter');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use(session({
  secret: 'GG##@$',
  cookie: {domain: 'localhost'},
  key: 'express_chapter6',
  resave: false,
  saveUninitialized: false,
  store: new RedisStore({
    client: redis.createClient(6379, '127.0.0.1'),
    ttl: 3600*72,
    db: 2,
    prefix: 'session:chapter6:'
  })
}));

app.use(authFilter);
app.use('/', routes);
```

代码 7.4.2 引入授权拦截器

将其放到`session`中间件的后面是由于，我们在这个拦截器中需要读取 `req.session` 变量，如果放到`session`中间件前面，则这个变量不存在。现在我们在未登录的情况下访问<http://localhost:3000/user/admin>，则会直接跳转到登陆页。

本章节代码可以从这里获

取：<https://github.com/yunnysunny/expressdemo/tree/master/chapter7>。

8 Node.js 单元测试

8.1 单元测试

对于一个程序员来说不仅要写代码，还要验证一下代码写得到底对不对，写单元测试就是一个通用且有效的解决方案。单元测试很重要，可以将错误扼杀在摇篮中，如果你认为没有写单元测试也过得很好，也许等我介绍完 [mocha](#) 之后，你会改变主意的。

下面给出一个栗子，领导给了小明一个计算器的项目，不过这个项目周期比较长，后期需要增加更多的人手，所以对于每一个模块都要有相应的测试用例。下面是小明开工后写的第一个函数，一个加法函数：

```
function add(a,b) {  
    return a+b;  
}
```

代码 8.1.1 add 函数

小明之前没有接触过单元测试，于是乎他写出的测试用例是这样的：

```
var s = add(1,2);  
if (s == 3) {  
    console.log('恭喜你，加法测试成功了');  
} else {  
    console.error('哎哟，加法测试出错了啊');  
}
```

代码 8.1.2 不使用测试框架的测试用例

接着他又写了减法函数、乘法函数、除法函数，但是随着模块的增加，他逐渐的意识到一个问题，如果按照代码8.1.2的模式的话，一则输出格式比较乱，而且测试结果没有最终统计信息，不能一下子得出成功数和失败数。于是乎他去网上找资料，然后他就发现了大名鼎鼎的 [mocha](#)。

研究了一番，小明发现 [mocha](#) 提供了测试结果输出格式化和测试结果统计的功能，也就是说对于上面那个加法测试用例，就可以这么写了：

```
var assert = require('assert');
describe('Calculator', function() {
  describe('#add()', function() {
    it('should get 3 when 1 add 2', function() {
      assert.equal(3, add(1,2));
    });
  });
});
```

代码 8.1.3 使用 **mocha** 改写测试用例

官网上说，**mocha** 在使用的时候，可以使用全局安装的模式，也可以安装为当前项目的开发依赖包（即在安装的时候使用 `--save-dev` 参数），不过小明考虑到以后各个项目都要用得到，于是决定进行全局安装（`npm install mocha -g`）。

安装完之后，小明兴奋的新建了文件 `calculator_test.js`：

```
var assert = require('assert');
var calculator = require('./calculator');

describe('Calculator', function() {
  describe('#add()', function() {
    it('should get 3 when 1 add 2', function() {
      assert.equal(3, calculator.add(1,2));
    });
  });
});
```

代码 8.1.4 文件 **calculator_test.js** 部分代码

接着他运行 `node calculator_test.js`，没想到结果打出他的意料，报错了：

```
e:\kuaipan\code\node\myapp\chapter7\src\test\mocha>node calculator_test.js
```

```
e:\kuaipan\code\node\myapp\chapter7\src\test\mocha\calculator_test.js:4
```

```
describe('Calculator', function() {
```

```
^
```

```
ReferenceError: describe is not defined
```

```
    at Object.<anonymous> (e:\kuaipan\code\node\myapp\chapter7\src\test\mocha\calculator_test.js:4:1)
```

```
    at Module._compile (module.js:456:26)
```

```
    at Object.Module._extensions..js (module.js:474:10)
```

```
    at Module.load (module.js:356:32)
```

```
    at Function.Module._load (module.js:312:12)
```

```
    at Function.Module.runMain (module.js:497:10)
```

```
    at startup (node.js:119:16)
```

```
    at node.js:929:3
```

输出 **8.1.1** 命令 **node calculator_test.js** 的输出

一定是小问题，小明心里嘀咕着，然后顺手打开了 **google**，直接搜

索 **ReferenceError: describe is not defined** 在第一页就发现了答案，原来要运行 **mocha calculator_test.js**（还是谷歌靠谱）。运行完正确的命令后，果然看到想要的结果了：

```
e:\kuaipan\code\node\myapp\chapter7\src\test\mocha>mocha calculator_test.js
```

```
Calculator
```

```
  #add()
```

```
    ✓ should get 3 when 1 add 2
```

```
1 passing (7ms)
```

输出 **8.1.2 命令**`mocha calculator_test.js`的输出

小明发现，这里之所以使用 node 自带的 `assert` 包，是由于 `mocha` 仅仅本身没有提供断言（Assertion）库，所以他又尝试了几种常用的断言库。

1.should.js

他使用的格式是 `(something).should` 或者 `should(something)`，更多使用方法还得参阅其[github文档](#)，例如上面我们使用 `assert` 进行判断的代码就可以写成：

```
var should = require('should');

(calculator.add(1,2)).should.be.exactly(3).and.be.a.Number();
```

代码 **8.1.5 should判断1**

或者：

```
var should = require('should/as-function');

should(calculator.add(1,2)).be.exactly(3).and.be.a.Number();
```

代码 **8.1.6 should判断2**

画外音，上面仅仅是简单说明使用方法，完整的测试用例大家可以参见第七章源码 `test/mocha/calculator_should1.js` 和 `test/mocha/calculator_should2.js`。

2.expect.js

首先给出expect.js的[github地址](#)，下面是用expect重写的测试用例：

```
var expect = require('expect.js');
expect(calculator.add(1,2)).to.be(3);
```

代码 8.1.7 expect判断

3.chai

chai将前面提到的assert should expect融合到了一起，你仅仅需要使用 chai 这一个包就能享用以上三者的功能，所以前面讲到的三种判断在chai中是这么实现的：

```
var chai = require('chai');

var assert = chai.assert; //use assert
assert.equal(3, calculator.add(1,2));

chai.should(); //use should
(calculator.add(1,2)).should.be.equal(3).and.be.a('number');

var expect = chai.expect; //use expect
expect(calculator.add(1,2)).to.equal(3);
```

代码 8.1.8 使用chai判断

这个 chai 还真是个大杀器呢。不过注意，在 chai 中


```
to
be
been
is
that
which
and
has
have
with
at
of
same
```

只能作为属性使用，不能作为函数使用（除非你自己写代码把这些属性覆盖掉），所以 `to.be(3)` 要写作 `to.equal(3)`，另外 `chai` 中也没有 `exactly` 这个函数，所以这里也是用 `equal` 来替代，同时在 `chai` 中 `a` 只能作为函数使用，其函数声明为 `a(type)`，所以这里用了 `a('number')`，其他技术细节，请移步官方 [API BDD部分](#)。

4.supertest

小明将计算器的任务完成了，测试用例写的也很完备，经理对其进行了表扬。不过他同时分配了下一期的任务，公司要提供一个计算器云的业务，要将小明之前做的功能以接口的形式提供给用户调用，不过这个项目同样要给出测试用例。

HTTP请求和本地调用从流程是有很多不一样的地方的：调用的过程中服务器可能会出现异常，返回数据有可能格式不正确，传递参数有可能出现偏差；而本地代码测试只要关心调用得到的结果是否正常就够了。不过，还好，小明找到了 [supertest](#) 这个 `mocha` 断言工具。

这里先给出小明写的一个HTTP接口：

```
exports.doAdd = function(req, res) {
  var _body = req.body;
  var a = parseInt(_body.a, 10);
  if (isNaN(a)) {
    return res.send({code:1,msg:'a值非法'});
  }
  var b = parseInt(_body.b, 10);
  if (isNaN(b)) {
    return res.send({code:2,msg:'b值非法'});
  }
  res.send({code:0,data:a+b});
};
```

代码 8.1.9 加法运算的HTTP接口

对应的supertest的测试用例代码就是这样的：

```
var request = require('supertest');
var app = require('../app');

describe('POST /calculator/add', function() {
  it('respond with json', function(done) {
    request(app)
      .post('/calculator/add')
      .send({a: 1,b:2})
      .expect(200, {
        code:0,data:3
      },done);
  });
});
```

代码 8.1.10 加法运算HTTP接口的单元测试代码

此代码放置于第七章代码的目录 /src/test/http目录下，变量app其实是引用的项目根目录的app.js。

完成了这个所谓的计算器云的项目后，小明又接到了新的任务，要做一个开发者平台，这个平台允许开发者创建基于计算云的开发者帐号和应用，然后登录进去管理自己的应用。

为了简化我们的教程，我们姑且认为这个平台使用的登录请求是我们之前用过的 `/user/login` 请求，管理后台首页是 `/user/admin`。我们现在来测试登录到后台这个动作。

登录过程要牵扯到我们在第6章讲过的 `session` 的知识，但是 `session` 在前端要有 `sessionid` 写入 `cookie` 中，我们的测试运行环境是命令行，不是浏览器，这就需要我们自己来维护 `cookie` 的读写操作。同时我们还需要用到 `mocha` 中的钩子（Hook）函数。由于我们的后台在进入之前需要先登录，所以我们要用到 `mocha` 中的 `before` 函数，它表示在所有测试用例之前执行，当然还有 `beforeEach` 函数，它表示在每次测试用例执行前都执行一次。显然两者的区别是前者只在所有测试用例前执行一次，而后者要在每个测试用例执行前都要重新执行。先看这个钩子函数：

```
var request = require('supertest');
var app = require('../..../app');
var cookie = exports.cookie = '';

before(function(done) {
  request(app)
    .post('/user/login')
    .send({username:'admin', password:'admin'})
    .expect(200,{code:0})
    .end(function(err,res) {
      if (err) {
        return done(err);
      }
      var header = res.header;
      var setCookieArray = header['set-cookie'];

      for (var i=0,len=setCookieArray.length;i<len;i++) {
        var value = setCookieArray[i];
        var result = value.match(/^express_chapter7=([a-zA-Z0-9%\.\-\_]+);\s/);
        if (result && result.length > 1) {
          exports.cookie = cookie = result[1];
          break;
        }
      }
      if (!cookie) {
        return done(new Error('查找cookie失败'));
      }
      done();
    });
});
```

代码 8.1.11 钩子函数

这里我们使用全局变量`cookie`来存储我们提到的 `sessionid`，在 `supertest` 的 `end` 函数中，我们读取响应体变量`res`中的 `set-cookie` 头信息，通过正则把 `sessionid` 读取出来。注意我们这里将 `cookie` 这个变量还专门做导出了，这样子我们就可以在其他测试文件中引用这个变量了。

接着就可以使用这个读取到的`cookie`来进入后台了：

```
var request = require('supertest');
var app = require('../..../app');
var before = require('./login_before');

describe('Backend', function() {

  it('first test', function(done) {
    request(app)
      .get('/user/admin')
      .set('Cookie', 'express_chapter7=' + before.cookie)
      .expect(200, /<title>admin<\|title>\/, done);
  });
});
```

代码 8.1.12 请求中使用 cookie

我们在模板 `user/admin.ejs` 有这么一句：`<title><%=user.account%></title>`，所以我们这里在测试时使用正则 `/<title>admin<\|title>\/` 来验证是否真的进入后台了。还有就是在读取 `cookie` 变量的时候没有直接在 `require` 完成之后立即读取，因为那个时候，这个变量还没有被赋值，而是在测试用例内部通过 `before.cookie` 读取。

注意，由于我们使用了 `redis` 来存储 `session` 数据，所以如果你忘记启动 `redis` 服务器的话，我们的登录操作会失败，而且在 `mocha` 中给出的报错提示是请求超时，这个问题比较隐蔽，大家一定要注意。

经过一番实践，小明的熟练掌握了各种测试技能，不过某天经理找打了他，“小明啊，要不你转到测试组吧”，.....

本章配套代

码：<https://github.com/yunnysunny/expressdemo/tree/master/chapter8>

9 Node.js 最佳实践

9.1 配置文件

一般代码的运行的环境起码应该包括本地开发环境和线上运行环境，那么问题来了，你开发环境用的配置信息可是跟线上环境不一样的。那么已经存储这个配置信息呢？在代码中写死肯定是最low的方式。更通用的方式是使用配置文件，可是你一旦将这个配置文件就面临一个问题，你这个配置文件一旦提交到了 git 之后，你的同事 pull 代码之后，就有可能就他本地配置文件覆盖掉，而这个配置文件中又包含了本地文件路径有关的配置，但是你和你的同事用的还不是一个操作系统（一个 windows，一个 mac），想想这个场景就恶心。那如果我们不将这个配置文件提交 git 呢？每次新增配置文件选项，都需要口头通知你的同事们，又比如有新同事加入到这个项目来了，你只能把你电脑上存储的配置文件发他一份，让他做相应的修改再使用，想想这个场景就恶心。

其实解决这个问题的办法也很简单，就是在 git 上放置一个配置文件的示例文件，我们就假设它为 `config.example.json`，里面写入所有的示例配置项。然后在 `.gitignore` 中将 `config.json` 添加进去，最后你在代码中加载 `config.json` 这个配置文件，这样的话大家都可以使用自己的配置文件，不会相互干扰，同时在 git 还存留了一份示例文件，配置项的更改都可以呈现到这个示例文件上。貌似是个完美的解决方案，其实这种解决方案不仅适用用 node，任何语言都适用。

但，这并不是问题的终点，我们是给出了一个配置示例文件

`config.example.json`，但是如果配置项有修改，你的同事不修改 `config.example.json` 怎么办？那就把错误扼杀在摇篮中吧，在我们的项目中会引入一个 `setting.js` 的文件来负责做配置项校验，在应用加载时，如果检测到某个参数不存在或者非法，就直接退出当前进程，让你启动不起来：

```
var log4js = require('log4js');
var mongoskin = require('mongoskin');
var redis = require('redis');
var slogger = require('node-slogger');

var configObj = require('../config.json');
var settings = require('./lib/settings').init(configObj);
exports.port = settings.loadNecessaryInt('port');
```

```
//保证配置文件中的debugfilename属性存在，且其所在目录在当前硬盘中存在
var debugFile = settings.loadNecessaryFile('debuglogfilename', true);
var traceFile = settings.loadNecessaryFile('tracelogfilename', true);
var errorFile = settings.loadNecessaryFile('errorlogfilename', true);

log4js.configure({
  appenders: [
    {type: 'console'},
    {type: 'dateFile', filename: debugFile, 'pattern': 'dd',
    backups: 10, category: 'debug'}, //
    {type: 'dateFile', filename: traceFile, 'pattern': 'dd',
    category: 'trace'},
    {type: 'file', filename: errorFile, maxLogSize: 1024000,
    backups: 10, category: 'error'}
  ],
  replaceConsole: true
});

var debugLogger = exports.debuglogger = log4js.getLogger('debug');
var traceLogger = exports.tracelogger = log4js.getLogger('trace');
var errorLogger = exports.errorlogger = log4js.getLogger('error');
slogger.init({
  debugLogger:debugLogger,
  traceLogger:traceLogger,
  errorLogger:errorLogger
});

var dbConfig = settings.loadNecessaryObject('db');//保证配置文件中的db属性存在
if (dbConfig.url instanceof Array) {
  exports.db = mongoskin.db(dbConfig.url, dbConfig.dbOption, dbConfig.reconnectOption);
}
```

```
} else {
    exports.db = mongoskin.db(dbConfig.url, dbConfig.dbOption);
}

var redisConfig = settings.loadNecessaryObject('redis');//保证配置
文件中的redis属性存在
exports.redis = redis.createClient(redisConfig.port, redisConfig
    .host);
```

代码 9.1 使用配置文件

9.2 自动重启

作为一个健壮的线上环境，肯定不希望自己的应用程序垮掉。然而，现实开发中在代码中总是会时不时出现未捕获的异常导致程序崩溃，真实编程实践中，我们肯定会对代码慎之又慎，但是想要代码100%无bug是不可能的，想想那个整天升级打补丁的微软。

我们用下面代码监听未捕获异常：

```
process.on('uncaughtException', function(err) {
    try {
        errorlogger.error('出现重大异常，重启当前进程',err);
    } catch(e) {
        console.log('请检查日志文件是否存在',e);
    }

    console.log('kill current proccess:'+process.pid);
    process.exit();
});
```

代码 9.2.1 监听未捕获异常

在 代码 9.2.1 中最后一行将当前进程强制退出，这是由于如果如果不这么做的话，很有可能会触发内存泄漏。我们肯定希望进程在意外退出的时候，能够重新再启动。这种需求其实可以使用 Node 的 cluster 来实现，这里我们不讲如何通过代码来达到如上需求，我们介绍一个功能十分之完备的工具——[pm2](#)。

首先我们运行 `cnpm install pm2 -g` 对其进行全局安装。为了做对比，我们首

先来观察不用pm2的效果。本章用的源码是第6章的基础上完成的，由于在第6章中我们使用了登陆拦截器，为了不破坏这个结构，我们新生成一个路由器，放置在 `routes/test.js`，然后在 `app.js` 中引入这个拦截器：

```
app.use('/test', testRoutes);
app.use(authFilter);
app.use('/', routes);
```

代码 9.2.2 添加测试路由器

然后在 `routes/test.js` 中添加让程序崩溃的代码：

```
router.get('/user', function(req, res) {
  setTimeout(function() {
    console.log(noneExistVar.pp);
    res.send('respond with a resource');
  }, 0);
});
```

代码 9.2.3 导致进程崩溃

可能你要问，这个地方为啥要加个 `setTimeout`，因为如果你不把这个错误放到异步代码中，就会像代码 5.2.6那样被express本身捕获到，就不会触发未捕获异常了。

最后启动应用，访问 `/test/user` 路径，不出意外，程序崩溃了。

然后我们用 pm2 来启动：

```
pm2 start src/bin/www
```

运行成功后会有如下输出：

```
[PM2] Spawning PM2 daemon
```

```
[PM2] PM2 Successfully daemonized
```

```
[PM2] Starting src/bin/www in fork_mode (1 instance)
```

[PM2] Done.

App name	id	mode	pid	status	restart	uptime	memory
www	0	fork	5804	online	0	10s	29.3 MB
		disabled					

Use ``pm2 show <id|name>`` to get more details about an app

输出 9.2.1

pm2 命令还有好多命令行参数，如果单纯手敲的话就太麻烦了，幸好它还提供了通过配置文件的形式来指定各个参数值，它支持使用 json 或者 yaml 格式来书写配置文件，下面给出一个 json 格式的配置文件：

```
{
  apps : [{
    name      : "chapter7",
    script    : "./src/bin/www",
    instances : 2,
    watch     : true,
    error_file : "/temp/log/pm2/chapter7/error.log",
    out_file  : "/temp/log/pm2/chapter7/out.log",
    env: {
      "NODE_ENV": "development",
    },
    env_production : {
      "NODE_ENV": "production"
    }
  }]
}
```

配置文件 9.2.1 process.json

我为啥要在日志文件的路径配置项上写linux路径呢，因为在 windows 下使用 pm2，一旦出现未捕获异常，进程重启的时候，都会弹出命令行窗口来抢占当前的桌面。所以我只能在 linux 下进行测试。并且经过测试，如果使用 node 0.10.x 版本的话，遇到未捕获异常时，进程无法重启，会僵死，所以推荐使用 4.x+ 版本。

接着运行如下命令来启动项目：

```
pm2 start process.json
```

命令 9.2.1

如果你想重启当前项目，运行：

```
pm2 restart process.json
```

命令 9.2.2

如果想关闭当前进程，运行：

```
pm2 stop process.json
```

命令 9.2.3

你还可以使用命令 `pm2 logs chapter7` 来查看当前项目的日志。最后我们来测试一下，访问我们故意为之的错误页面 `http://localhost:8100/test/user`，会看到控制台中会打印重启日志：

```
chapter7-0 ReferenceError: noneExistVar is not defined
chapter7-0      at null._onTimeout (/home/gaoyang/code/expressdemo/chapter7/src/routes/test.js:7:17)
chapter7-0      at Timer.listOnTimeout (timers.js:92:15)
chapter7-0 [2016-09-16 23:28:14.016] [ERROR] error - 出现重大异常
，重启当前进程 [ReferenceError: noneExistVar is not defined]
chapter7-0 ReferenceError: noneExistVar is not defined
chapter7-0      at null._onTimeout (/home/gaoyang/code/expressdemo/chapter7/src/routes/test.js:7:17)
chapter7-0      at Timer.listOnTimeout (timers.js:92:15)
chapter7-0 [2016-09-16 23:28:14.025] [INFO] console - kill current process:6053
chapter7-0 load var [port],value: 8100
chapter7-0 load var [debuglogfilename],value: /tmp/debug.log
chapter7-0 load var [tracelogfilename],value: /tmp/trace.log
chapter7-0 load var [errorlogfilename],value: /tmp/error.log
chapter7-0 [2016-09-16 23:28:14.908] [INFO] console - load var [db],value: { url: 'mongodb://localhost:27017/live',
chapter7-0   dbOption: { safe: true } }
chapter7-0 [2016-09-16 23:28:14.934] [INFO] console - load var [redis],value: { port: 6379, host: '127.0.0.1' }
chapter7-0 [2016-09-16 23:28:15.003] [INFO] console - Listening on port 8100
```

输出 9.2.1

我们看到进程自己重启了，最终实现了我们的目的。

9.3 开机自启动

虽然我们在服务上线的时候，可以请高僧来给服务器开光，其实只要不是傻子就看得出来那只不过博眼球的无耻炒作而已。机器不是你想不宕就不宕，所以说给你的服务加一个开机自启动，是绝对有必要的，庆幸的是 `pm2` 也提供了这种功能。

以下演示命令是在 `Ubuntu 16.04` 做的，其他服务器差别不大，首先运行 `pm2 startup`，正常情况会有如下输出：

```
[PM2] Writing init configuration in /etc/init.d/pm2-root
[PM2] Making script booting at startup...
>>> Executing chmod +x /etc/init.d/pm2-root
[DONE]
>>> Executing mkdir -p /var/lock/subsys
[DONE]
>>> Executing touch /var/lock/subsys/pm2-root
[DONE]
>>> Executing chkconfig --add pm2-root
[DONE]
>>> Executing chkconfig pm2-root on
[DONE]
>>> Executing initctl list
rc stop/waiting
tty (/dev/tty3) start/running, process 2312
tty (/dev/tty2) start/running, process 2310
tty (/dev/tty1) start/running, process 2308
tty (/dev/tty6) start/running, process 2318
tty (/dev/tty5) start/running, process 2316
tty (/dev/tty4) start/running, process 2314
plymouth-shutdown stop/waiting
control-alt-delete stop/waiting
rcS-emergency stop/waiting
kexec-disable stop/waiting
quit-plymouth stop/waiting
rcS stop/waiting
prefdm stop/waiting
init-system-dbus stop/waiting
splash-manager stop/waiting
start-ttys stop/waiting
rcS-sulogin stop/waiting
serial stop/waiting
[DONE]
+-----+
[PM2] Freeze a process list on reboot via:
$ pm2 save

[PM2] Remove init script via:
$ pm2 unstartup systemv
```

按照上面的提示，用 `pm2 save` 产生当前所有已经启动的 `pm2` 应用列表，这样下次服务器在重启的时候就会加载这个列表，把应用再重新启动起来。

最后，如果不想再使用开机启动功能，运行 `pm2 unstartup systemd` 即可取消。

9.4 使用docker

随着智能设备的蓬勃发展，整个互联网的网民总数出现了井喷，对于软件开发者来说，面对的用户群体越来越庞大，需求变化原来越快，导致软件开发的规模越来越大，复杂度越来越高。为了应对这些趋势，最近几年一些新的技术渐渐被大家接受，比如说 `devops`，比如说我们接下来要讲的 `docker` 容器。

有了 `docker`，大家就可以本地开发代码，然后开发完成之后直接打一个包扔到服务器上运行，这个包就是我们所说的容器，它跟宿主机无关，不管运行在何种宿主机上，它的内部环境都是一致。所以说有了 `docker`，我们在也不用担心在本地跑的好好的，结果一到服务器就出错的问题了。

当然如果你们服务器使用了 `Docker` 技术的话，9.3小节的内容就没有必要使用了。因为在 `Docker` 上是没法设置开机服务的。

`pm2` 提供了生成 `Dockerfile` 的功能，不过生成的文件实用性不是很强，我需要稍加改造了一下。另外为了方便演示 `docker` 使用，专门在 `oschina` 新建一个 `代码仓库` 用于第8章代码。下面演示一下 `dockerfile` 的编写，具体流程是在 `docker` 构建的时候，使用 `git clone` 从仓库中拿去代码，然后安装所需的依赖。构建完成之后，每次启动这个 `docker` 容器的使用使用 `pm2` 命令启动当前应用。`dockerfile` 的示例代码如下：

```
FROM mhart/alpine-node:latest

RUN sed -i 's/dl-cdn.alpinelinux.org/mirrors.ustc.edu.cn/g' /etc
/apk/repositories
RUN apk update && apk add git && apk add openssh-client && rm -r
f /var/cache/apk/*

#创建应用目录
RUN mkdir -p /var/app
RUN mkdir -p /var/log/app
#将git clone用的sshkey的私钥拷贝到.ssh目录下
COPY deploy_key /root/.ssh/id_rsa
RUN chmod 600 ~/.ssh/id_rsa
#将当前git服务器域名添加到可信列表
RUN ssh-keyscan -p 22 -t rsa git.oschina.net >> /root/.ssh/know
n_hosts

WORKDIR /var/app

#clone代码
RUN git clone git@git.oschina.net:nodebook/chapter9.git .
#拷贝配置文件
COPY config.production.json config.json
COPY process.production.json process.json

#安装cnpm
RUN npm install -g cnpm --registry=https://registry.npm.taobao.o
rg
#安装pm2
RUN cnpm install pm2 -g
RUN cnpm install

#向外暴露当前应用的端口
EXPOSE 8100:8100

## 设置环境变量
ENV NODE_ENV=production
# 启动命令
CMD ["pm2-docker", "process.json"]
```


代码 9.4.1 Dockerfile 示例

其中 `From` 代表使用的基础镜像，`alpine` 是一个非常轻量级的 linux 发行版本，所以基于其制作的 `docker` 镜像非常小，特别利于安装。这里的 `alpine-node` 在 `alpine` 操作系统上集成了 `node`，单纯 `pull` 安装的话也非常小。然后 `RUN` 和 `COPY` 两个命令是在构建的时候执行命令和拷贝文件，注意 `COPY` 命令仅仅只能拷贝当前执行 `docker` 命令的目录下的文件，也就是说拷贝的时候不能使用相对路径，比如说你要执行 `COPY xxx/yyy /tmp/yyy` 或者 `COPY ../zzz /tmp/zzz` 都是不允许的。为了正确的 `clone git` 服务器上的代码，我们还需要配置一下部署密钥。谈到部署密钥的概念，这里还要多说几句。我们一般从 `git` 服务器上 `clone` 下来代码后，会对代码进行编写，然后 `push` 你编写后的新代码。但是服务器上显然是不适合在其上面进行直接改动代码的，所以就有了部署密钥的概念，使用部署密钥你可以做 `clone` 和 `pull` 操作，但是你不能做 `push` 操作。

```
$ ssh-keygen -f deploy_key -C "somebody@some site.com"
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in deploy_key.
Your public key has been saved in deploy_key.pub.
The key fingerprint is:
SHA256:S3JbyWc68K43kifBwYcJJx1IF1D1Xz9MJDGI6gEhFKw somebody@some
site.com
The key's randomart image is:
+---[RSA 2048]-----+
|+0+==+0+ .+..      |
| 0....= 0  +       |
|. . . .= 0. .      |
|E  o  . =0.=       |
| . . . .S0+ *       |
| .  +0* + .        |
|      oo+          |
|      +.+          |
|      .*..         |
+----[SHA256]-----+
```

命令 9.4.1 生成密钥对

我们在第8章项目代码根目录下新建一个 `deploy` 文件夹，进入这个文件夹然后运行命令 **9.4.1**，一路回车即可。然后我们就得到了 代码 **9.4.1** 中的 `deploy_key` 了。生成完了之后去 git.oschina.com 上配置一下公钥（也就是我们生成的 `deploy_key.pub` 文件），在项目页（在这里是 <http://git.oschina.net/nodebook/chapter8>）上点击 管理 导航链接（），在打开的页面中点击 部署公钥管理，然后选择 添加公钥，用记事本打开刚才生成的 `deploy_key.pub` 文件，全选复制，然后贴到输入框中：



图 9.4.1 添加部署公钥

最后要注意一下 `EXPOSE` 命令，他代表 `docker` 及向宿主机暴露的端口号，如果不暴露端口的话，在宿主机上没法访问我们应用监听的端口。我们运行 `docker build -t someone/chapter8 .` 其中 `-t` 参数指定当前镜像的 `tag` 名称，`someone` 是指你在 [docker hub](https://hub.docker.com/) 网站上注册的用户，`build` 成功后你可以通过 `docker push someone/chapter8` 将构建后的结构 `push` 到 `docker hub` 网站上去，然后在服务器上运行 `docker pull someone/chapter8` 来拿取你当初 `push` 的仓库。当然你可以直接将 `Dockerfile` 拿到你的服务器上执行 `build` 命令，这时候 `-t` 参数可以随便指定，甚至不写。

鉴于国内的网络环境问题，在做 `build` 的时候，`pull` 基础镜像很有可能会失败，这时候你就只能求助于国内的 `docker` 镜像站了，比如说 [daocloud](https://www.daocloud.io/)。

`build` 命令运行完成之后，运行 `docker images` 会输出：

REPOSITORY TUAL SIZE	TAG	IMAGE ID	CREATED	VIR
someone/chapter8 .7 MB	latest	2a1a00cc1b41	4 minutes ago	147

最后我们通过 `docker run -d --name chapter8 someone/chapter8` 即可生成一个 `docker` 容器。其中 `-d` 参数代表在后台运行，`--name` 指定当前 `docker` 容器的名称，`someone/chapter8` 说明我们使用刚才 `build` 的镜像来生成容器。通过 `docker ps` 命令的输出，我们可以查看生成的 `docker` 容器：

CONTAINER ID	IMAGE	COMMAND	C
REATED	STATUS	PORTS	NAMES
fb0d726a86dc	someone/chapter8	"pm2-docker process.	4
seconds ago	Up 4 seconds	8100/tcp	chapt
er8			

9.5 代码

本章代码9.1、9.2小节代码和第8章存储在相同位

置：<https://github.com/yunnysunny/nodebook-sample/tree/master/chapter8>，9.4
章节代码为演示方便专门做了一个仓库，位

于：<http://git.oschina.net/nodebook/chapter8>。

10 Node.js 的 c++ 扩展

Node 的优点是处理 IO 密集型操作，对于互联网应用来说，很大一部分内容都是 IO 处理（包括文件 IO 和网络 IO），但是还是有部分功能属于计算密集型操作。如果遇到这种计算密集型操作，推荐的解决方案是使用其他语言来实现，然后提供一个服务，让 Node 来进行调用。不过我们这章要讲的是 Node 的 C++ 扩展，也就是说，我们可以通过这种方式是 Node 代码直接“桥接”到 C++ 上，以此来解决计算密集型操作。但是我在前面为什么推荐使用其他语言来提供计算密集型的服务呢，因为一旦你的这个这个计算服务稍微一上规模，你的代码开发就要面临横跨两个语种的境况，给程序调试增加了很多不确定性。所以说，如果你要做的计算应用的功能比较单一的话，可以考虑做成 C++ 扩展。

Node 的 C++ 扩展功能是依赖于 V8 来实现的，但是在 Node 每次做大的版本升级的时候，都会有可能对应升级 V8 的版本，相应的扩展 API 的定义也很有可能发生变化，所以下面要重点介绍 nan 这个第三方包的，它提供了一系列的宏定义和包装函数，来对这些不同版本的扩展 API 进行封装。

10.1 准备工作

为了能够编译我们的 C++ 扩展，我们需要做一些准备工作，首先需要全局安装 `node-gyp` 这个包：`npm install -g node-gyp`。不过此包还依赖于 python 2.7（必须得用 2.7 版本，安装 3.0 是不管用的）。同时需要安装 C++ 编译工具，在 linux 下需要使用 `GCC`，Mac 下需要使用 `Xcode`，Windows 下需要安装 `Visual Studio`（版本要求是 2015，低于此版本的不可以，高于此版本的作者本身没有做过测试），大家可以选择安装社区版，因为专业版和旗舰版都是收费的，如果想进一步减小安装后占用磁盘的体积可以安装 `Visual C++ Build Tools`。按照官方说明在 windows 下安装完 `Visual Studio` 和 `node-gyp` 后，还需要使用命令 `npm config set msvs_version 2015` 来指定 `node-gyp` 使用的 VS 版本。

不过经笔者测试发现，如果在 Windows 中同时安装了多个 VS 工具时，`node-gyp` 有可能使用错误的版本进行编译，笔者电脑上同时安装了 `build tool 2015` 和 `vs 2012`，编译的时候老会选择使用 `vs 2012`，不得已将 `vs 2012` 卸载掉，但是依然报错，最后在 `node-gyp` 命令中添加参数 `--msvs_version=2015` 才解决。

10.2 hello world

为了演示如何编译一个 C++ 扩展，我们从亘古不变的 hello world 程序入手，这个程序取自 Node [C++扩展的官方文档](#)。我们的目的是在 C++ 扩展中实现如下代码：

```
exports.hello = () => 'world';
```

代码 10.2.1

这看上去有些拿大炮打蚊子的味道，这段代码太简单了，而我们竟然要用 C++ 将其实现一番，是的这一节关注的并不是代码本身，还是如何使用工具进行编译，所以我们选择了最简单的代码。首先我们创建 hello.cc 文件：

```
// hello.cc
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    args.GetReturnValue().Set(String::NewFromUtf8(isolate, "world"
));
}

void init(Local<Object> exports) {
    NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(addon, init)

} // namespace demo
```

代码 10.2.2 hello.cc

然后创建 `binding.gyp`，注意这里的 `target_name` 属性要和 `NODE_MODULE` 宏定义中的第一个参数保持相同。

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

配置文件 10.2.1

gyp (**Generate Your Project**) 是一种跨平台的项目构建工具，是谷歌员工在开发 **chromium** 项目时衍生出来的工具。Node.js 扩展说白了也是基于 V8 的 API 基础上的，所以它也采用 gyp 技术。

编写完成之后在 `hello.cc` 目录下运行命令行 `node-gyp configure`，成功之后会生成一个 `build` 文件夹，里面包含当前代码生成的 c++ 编译用文件。接着运行 `node-gyp build` 就能生成扩展包了。

编译成功后，我们就可以在 js 代码中引用这个扩展库了：

```
// hello.js
const addon = require('./build/Release/addon');

console.log(addon.hello());
// Prints: 'world'
```

代码 10.2.3 hello.js

之前讲过本章的重点是使用 `nan` 这个包来实现扩展编写，所以我们就先拿这个 `hello world` 下手。首先是安装 `nan` 包：`npm install nan --save`。然后编写 `hello.cc`：

```
// hello.cc
#include <nan.h>

using namespace v8;

NAN_METHOD(Method) {
    info.GetReturnValue().Set(Nan::New<String>("world").ToLocalChecked());
}

NAN_MODULE_INIT (Init) {
    Nan::Export(target, "hello", Method);
}

NODE_MODULE(hello_nan, Init)
```

代码 **10.2.4 hello.cc** 的 **nan** 版

可以看到和代码**10.2.2**相比代码**10.2.4**要简洁不少，这里 `NAN_METHOD(Method)` 经过宏定义解析为 `void Method(const Nan::FunctionCallbackInfo<v8::Value>& info)`，所以你在函数 `Method` 内部会有一个 `info` 对象，能够在编译的时候被正确识别。同时宏定义 `NAN_MODULE_INIT(Init)` 会被转化为 `void Init(v8::Local<v8::Object> target)` 所以你会在函数内部看到一个 `target` 对象。同时代码 **10.2.2** 第13行中 `Isolate* isolate = args.GetIsolate();` 这个代码在函数 `Nan::New<String>` 中被封装在其内部，所以在代码 **10.2.4** 中没有看到这段代码。

10.3 映射 C++ 类

C++ addone 最精髓的地方，就是将一个 JavaScript 类映射为一个 C++ 类，这样就会产生一个有趣的效果，你通过 `new` 构建的 js 对象，它的成员函数都被映射成 C++ 类中的成员函数。

下面举的例子可能可能看上去很傻，因为我们又写对 `a+b` 求值的函数了，但是这种很天真的代码最好理解不过了。注意，这个例子改写自项目 [node-addon-examples](#) 中的 `object_wrap` 小节。首先是 C++ 头文件定义：


```

#ifndef MY_CALC_H
#define MY_CALC_H

#include <nan.h>

class MyCalc : public Nan::ObjectWrap {
public:
    static void Init(v8::Handle<v8::Object> module);
private:
    explicit MyCalc(double value=0);
    ~MyCalc();

    static NAN_METHOD(New);
    static NAN_METHOD(PlusOne);
    static NAN_METHOD(GetValue);
    static Nan::Persistent<v8::Function> constructor;
    double _value;
};

#endif

```

代码 10.3.1 MyCalc.h

首先注意的一点是，类 `MyCalc` 要继承自 `Nan::ObjectWrap`，按照惯例这个类中还要有一个 `Persistent` 类型的句柄用来承载 js 类的构造函数，`MyCalc` 类中唯一对外公开的函数就是 `Init`，其参数 `module` 正是对应的是 Node 中的 `module` 对象。

```

#include "MyCalc.h"

Nan::Persistent<v8::Function> MyCalc::constructor;

MyCalc::MyCalc(double value): _value(value) {

}

MyCalc::~~MyCalc() {

}

```



```

void MyCalc::Init(v8::Handle<v8::Object> module) {
    // Prepare constructor template
    v8::Local<v8::FunctionTemplate> tpl = Nan::New<v8::FunctionT
emplate>(New); //使用ShmldbNan::New<Object>函数作为构造函数
    tpl->SetClassName(Nan::New<v8::String>("MyCalc").ToLocalChec
ked()); //js中的类名为MyCalc
    tpl->InstanceTemplate()->SetInternalFieldCount(1); //指定js类
的成员字段个数

    Nan::SetPrototypeMethod(tpl, "addOne", PlusOne); //js类的成员函数
名为addOne, 我们将其映射为 C++中的PlusOne函数
    Nan::SetPrototypeMethod(tpl, "getValue", GetValue); //js类的成员
函数名为getValue, 我们将其映射为 C++中的GetValue函数

    //Persistent<Function> constructor = Persistent<Function>::N
ew/*New等价于js中的new*/(tpl->GetFunction()); //new一个js实例
    constructor.Reset(tpl->GetFunction());
    module->Set(Nan::New<v8::String>("exports").ToLocalChecked()
, tpl->GetFunction());
}

NAN_METHOD(MyCalc::New) {
    if (info.IsConstructCall()) {
        // 通过 `new MyCalc(...)` 方式调用
        double value = info[0]->IsUndefined() ? 0 : info[0]->Num
berValue();
        MyCalc* obj = new MyCalc(value);
        obj->Wrap(info.This());
        info.GetReturnValue().Set(info.This());
    } else {
        // 通过 `MyCalc(...)` 方式调用, 转成使用构造函数方式调用
        const int argc = 1;
        v8::Local<v8::Value> argv[argc] = { info[0] };
        v8::Local<v8::Function> cons = Nan::New<v8::Function>(co
nstructor);
        info.GetReturnValue().Set(cons->NewInstance(argc, argv))
;
    }
}

```

```

NAN_METHOD(MyCalc::GetValue) {
    MyCalc* obj = ObjectWrap::Unwrap<MyCalc>(info.Holder());
    info.GetReturnValue().Set(Nan::New(obj->_value));
}

NAN_METHOD(MyCalc::PlusOne) {
    MyCalc* obj = ObjectWrap::Unwrap<MyCalc>(info.Holder());
    double wannaAddValue = info[0]->IsUndefined() ? 1 : info[0]-
>NumberValue();

    obj->_value += wannaAddValue;
    info.GetReturnValue().Set(Nan::New(obj->_value));
}

```

代码 10.3.2 MyCalc.cc

注意到在 Init 函数中，定义了一个 js 类的实现，并且将其成员函数和 C++ 类的成员函数做了绑定，其中构造函数绑定为 New，getValue 绑定为 GetValue，addOne 绑定为 PlusOne。Init 函数中的最后一行类似于我们在 js 中的 `module.exports = MyCalc` 操作。对于 C++ 函数 New GetValue PlusOne 来说，我们在定义的时候都使用了宏 NAN_METHOD，这样我们在函数内部就直接拥有了 info 这个变量，这个跟 代码 10.2.4 中的使用方法是一样的。

同时留意到在函数 GetValue 和 PlusOne 中，`MyCalc* obj = ObjectWrap::Unwrap<MyCalc>(info.Holder());`，这一句将 js 对象转化为 C++ 对象，然后操作 C++ 对象的属性。相反在函数 New 中 `obj->Wrap(info.This());` 是一个相反的过程，将 C++ 对象转化为 js 对象。

10.4 使用线程池

前面几小节介绍了 Nan 的基本使用，可是即使使用了 C++ addon 技术，默认情况下，你所写的代码依然运行在 V8 主线程上，所以说在面对高并发的情况下，如果你的 C++ 代码是计算密集型的，它依然会抢占 V8 主线程的 CPU 时间，最严重的后果当然就是事件轮询的 CPU 时间被抢占导致整个 Node 处理效率下降。所以说釜底抽薪之术还是使用线程。

Nan 中提供了 AsyncWorker 类，它内部封装了 libuv 中的 uv_queue_work，可以在将计算代码直接丢到 libuv 的线程做处理，处理完成之后再通知 V8 主线程。

下面是一个简单的小例子：

```
#include <string>
#include <nan.h>
#include <sstream>

#ifdef WINDOWS_SPECIFIC_DEFINE
#include <windows.h>
typedef DWORD ThreadId;
#else
#include <unistd.h>
#include <pthread.h>
typedef unsigned int ThreadId;
#endif
using v8::Function;
using v8::FunctionTemplate;
using v8::Local;
using v8::Value;
using v8::String;

using Nan::AsyncQueueWorker;
using Nan::AsyncWorker;
using Nan::Callback;
using Nan::HandleScope;
using Nan::New;
using Nan::Null;
using Nan::ThrowError;
using Nan::Set;
using Nan::GetFunction;

NAN_METHOD(doAsyncWork);

static ThreadId __getThreadId() {
    ThreadId nThreadId;
#ifdef WINDOWS_SPECIFIC_DEFINE
```

```

        nThreadID = GetCurrentProcessId();
        nThreadID = (nThreadID << 16) + GetCurrentThreadId();
    #else
        nThreadID = getpid();
        nThreadID = (nThreadID << 16) + pthread_self();
    #endif
    return nThreadID;
}

static void __tsleep(unsigned int millisecond) {
#ifdef WINDOWS_SPECIFIC_DEFINE
    ::Sleep(millisecond);
#else
    usleep(millisecond*1000);
#endif
}

class ThreadWoker : public AsyncWorker {
private:
    std::string str;
public:
    ThreadWoker(Callback *callback, std::string str)
        : AsyncWorker(callback), str(str) {}
    ~ThreadWoker() {}
    void Execute() {
        ThreadId tid = __getThreadId();
        printf("[%s]: Thread in uv_worker: %d\n", __FUNCTION__, tid);

        __tsleep(1000);
        printf("sleep 1 seconds in uv_work\n");
        std::stringstream ss;
        ss << " worker function: ";
        ss << __FUNCTION__;
        ss << " worker thread id ";
        ss << tid;
        str += ss.str();
    }
    void HandleOKCallback () {

```

```

        HandleScope scope;

        Local<Value> argv[] = {
            Null(),
            Nan::New<String>("the result:"+str).ToLocalCheck
ed()
        };

        callback->Call(2, argv);
    };
};

NAN_METHOD(doAsyncWork) {
    printf("[%s]: Thread id in V8: %d\n",__FUNCTION__,__getThrea
dId());
    if(info.Length() < 2) {
        ThrowError("Wrong number of arguments");
        return info.GetReturnValue().Set(Nan::Undefined());
    }

    if (!info[0]->IsString() || !info[1]->IsFunction()) {
        ThrowError("Wrong number of arguments");
        return info.GetReturnValue().Set(Nan::Undefined());
    }

    //
    Callback *callback = new Callback(info[1].As<Function>());
    Nan::Utf8String param1(info[0]);
    std::string str = std::string(*param1);
    AsyncQueueWorker(new ThreadWoker(callback, str));
    info.GetReturnValue().Set(Nan::Undefined());
}

NAN_MODULE_INIT(InitAll) {
    Set(target, New<String>("doAsyncWork").ToLocalChecked(),
        GetFunction(New<FunctionTemplate>(doAsyncWork)).ToLocalCheck
ed());
}

```

```
}  
  
NODE_MODULE(binding, InitAll)
```

代码 10.4.1 `async_simple.cc`

为了简单起见，将所有代码写到一个 `c++` 文件中，注意到在代码 **10.4.1**中使用了自定义宏定义 `WINDOWS_SPECIFIC_DEFINE`，在 `binding.gyp` 中是支持添加自定义宏定义和编译参数的，下面是这个项目中用到的 `binding.gyp` 文件：

```
{
  'targets': [
    {
      'target_name': 'async-simple',
      'defines': [
        'DEFINE_FOO',
        'DEFINE_A_VALUE=value',
      ],
      "include_dirs" : [
        "<!(node -e \"require('nan')\")"
      ],
      'conditions' : [
        ['OS=="linux"', {
          'defines': [
            'LINUX_DEFINE',
          ],

          'libraries':[
            '-lpthread'
          ],
          'sources': [ 'async_simple.cc' ]
        }],
        ['OS=="win"', {
          'defines': [
            'WINDOWS_SPECIFIC_DEFINE',
          ],
          'sources': [ 'async_simple.cc' ]
        }]
      ]
    }
  ]
}
```

代码 10.4.2 binding.gyp

`binding.gyp` 中的最外层的 `defines` 变量是全局环境变量，`conditions` 中可以放置各种条件判断，`OS=="linux"` 代表当前的操作系统是 linux，其下的 `defines` 下定义的宏定义，只有在 linux 系统下才起作用，所以在代码 10.4.1 中

的环境变量 `WINDOWS_SPECIFIC_DEFINE` 只用在 windows 上才起作用，我们使用这个宏定义来做条件编译，以保证能够正确使用线程函数（其实libuv 中封装了各种跨平台的线程函数，这里不做多讨论）。

继续回到代码 **10.4.1**，类 `ThreadWorker` 中，函数 `Execute` 用来执行耗时函数，它将在 libuv 中的线程池中执行，函数 `HandleOKCallback` 在函数 `Execute` 执行完成后被调用，用来将处理结果通知 libuv 的事件轮询，它在 V8 主线程中执行。

最终给出测试用的 js 代码，又回到了我们熟悉的回调函数模式：

```
var asyncSimple = require('./build/Release/async-simple');

asyncSimple.doAsyncWork('prefix:', function(err, result) {
    console.log(err, result);
});
```

代码 **10.4.3** `addon.js`

10.5 代码

本章代码位于 <https://github.com/yunnysunny/nodebook-sample/tree/master/chapter10>

Node.js 调优

作为一门后端语言，肯定要求运行的效率最优化，以实现对于资源最小损耗，这一章正是围绕这个话题展开。调优是一个有挑战性的活儿，可能经常让人摸不着头脑，下面的内容尽量使用实例来带入一个个调优的场景。

11.1 准备工作

首先我们准备一个 http server 端代码，请求后返回一个二维码图片：

```
var http = require('http');
var ccap = require('ccap')();//Instantiated ccap class

http.createServer(function (request, response) {
    if(request.url == '/favicon.ico')return response.end('');//I
ntercept request favicon.ico
    var ary = ccap.get();
    var txt = ary[0];
    var buf = ary[1];
    response.end(buf);
    console.log(txt);
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

代码 11.1.1 app.js

对于 node 版本大于 6.3 的会比较简单，运行

```
node --inspect=9229 app.js
```

命令 11.1.1

启动的 node 进程会带有调试功能，使用 Chrome DevTools 即可远程查看当前运行的 js 源码，并且能够生成 CPU 和 内存快照，这对于我们分析性能十分有帮助。

运行完命令 11.1.1 之后，我们在控制台上会看到如下输出：

```
> node --inspect=9229 app.js
```

Debugger listening on port 9229.

Warning: This is an experimental feature and could change at any time.

To start debugging, open the following URL in Chrome:

```
chrome-devtools://devtools/remote/serve_file/@60cd6e859b9f557d2312f5bf532f6a5f284980/inspector.html?experiments=true&v8only=true
```

```
&ws=127.0.0.1:9229/1a19bc9d-7175-4df3-b131-2eca35c7c844
```

Server running at http://127.0.0.1:8124/

Debugger attached

输出 11.1.1

我们将输出的地址 `chrome-`

`devtools://devtools/remote/serve_file/@60cd6e859b9f557d2312f5bf532f6a5f284980/inspector.html?experiments=true&v8only=true`

`&ws=127.0.0.1:9229/1a19bc9d-7175-4df3-b131-2eca35c7c844` 贴到 chrome 地址栏中访问，然后我们看到的竟然是一个空白页，好多教程中都说会直接打开一个 chrome 开发面板，然而并没有（我使用的是 Chrome 62 版本）。这时候，你在任何一个网页中手动打开一个开发面板，



图 11.1.1

你会发现多了一个 Node 的小图标，这个图标就是之前我们输入 `chrome-devtools` 地址后生成的，我用红色矩形专门标记了出来，点击这个图标又弹出了一个面板，

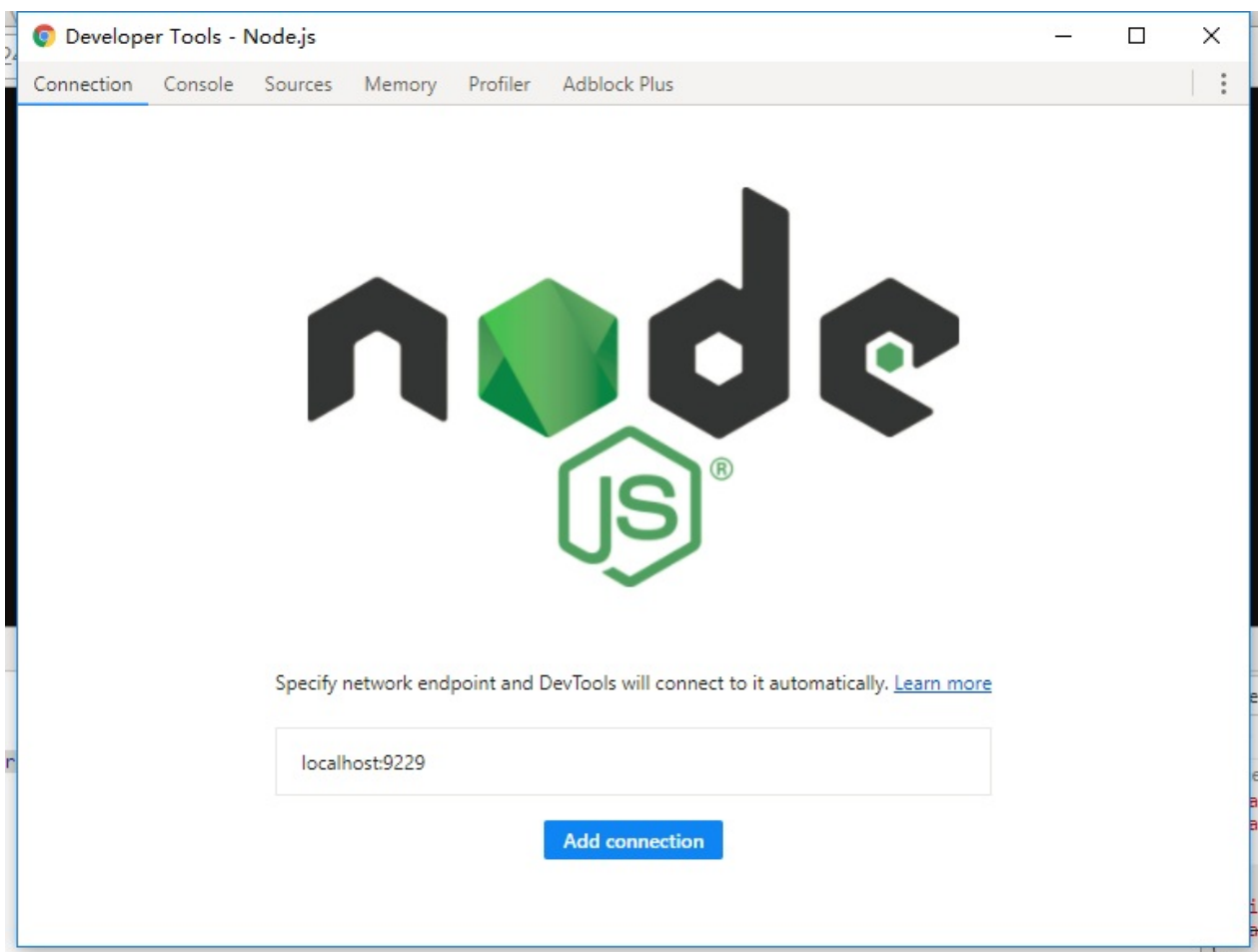


图 11.1.2

或者你在开发面板中选择 **Remote Devices**：

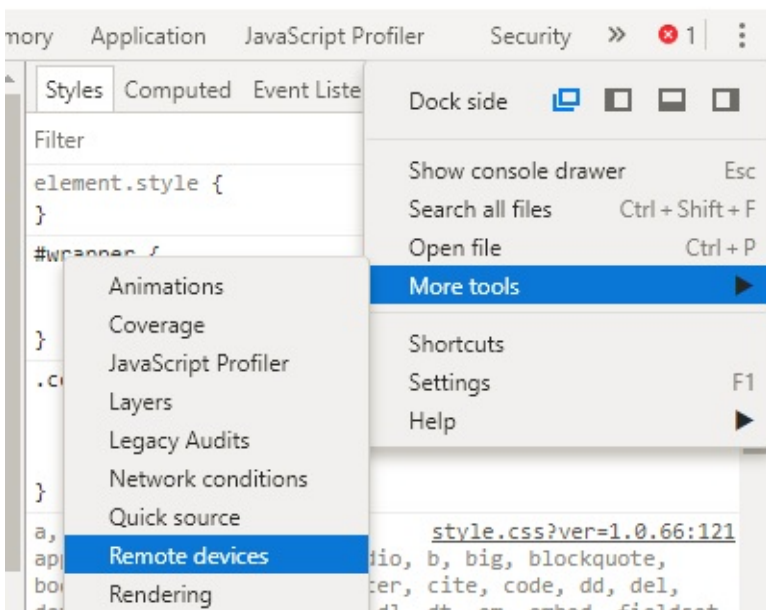


图 11.1.3

然后在弹出的 Tab 页中找到 **Add Address** 按钮，点击然后输入 Node 应用部署的 ip 和 inspect 端口号（即启动 node 程序的 **inspect** 参数），甚至可以监听任意远程电脑的 Node 应用（如图 11.1.4 所示），添加成功后，同样可以触发开发面板上出现图 11.1.1 中的 Node 小图标。

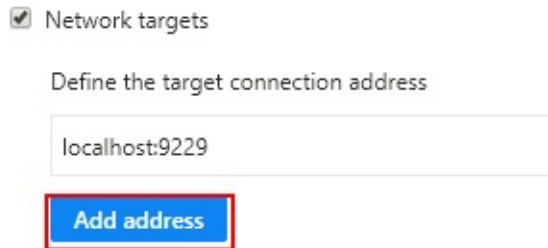


图 11.1.4

继续回到图 11.1.2，里面有一个按钮 **Add connection**，其实我们在这里一样可以添加远程 Node 应用的 inspect 信息，这里的功能和图 11.1.4 中是一样的。我们点开他的 **Source** tab 页，能够找到我们启动的 app.js 的源码：

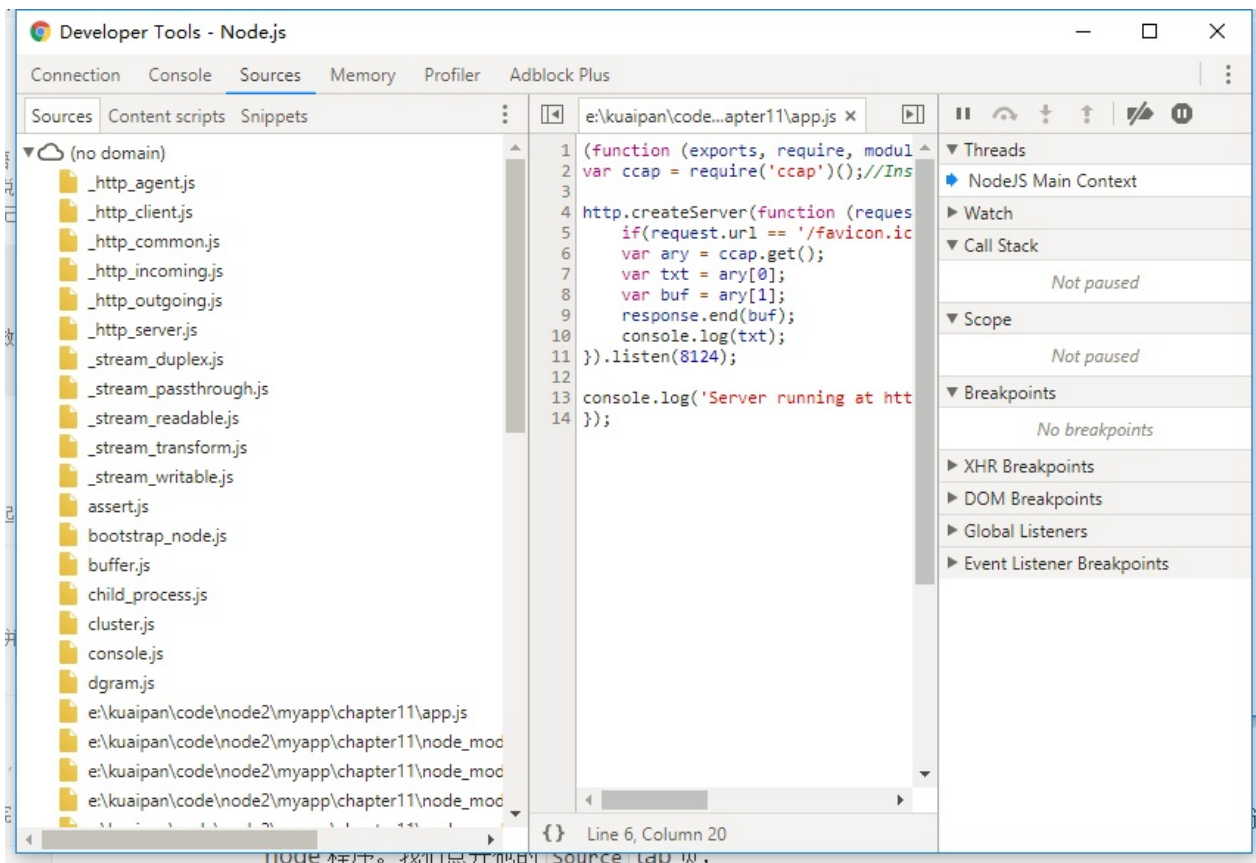


图 11.1.5

这项功能对于 chrome 来说还处于实验状态，所以在操作过程中如果你的浏览器崩溃，不要怀疑，这属于正常现象。

11.2 压力测试

要想知道性能如何，需要首先借助压力测试工具，这里我们选择开源的 [JMeter](#)。

打开 JMeter 后，首先创建一个线程组，用过 LoadRunner 的同学，可能会比较熟悉 用户数，这个概念对应到 JMeter 的话，就是线程组中的线程



图 11.2.1 添加线程组

接着我们添加一个 HTTP 请求的默认参数

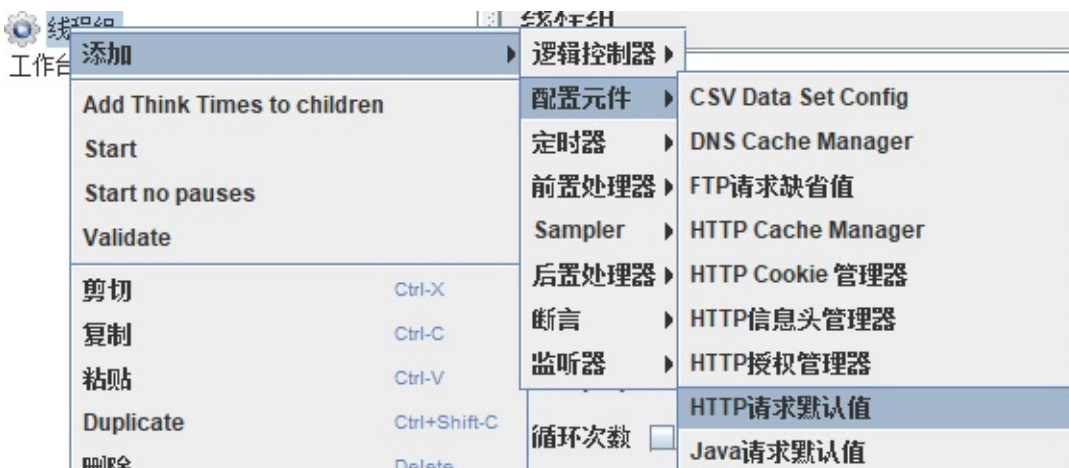


图 11.2.2 HTTP 请求默认参数设置

在这里我们仅仅需要设置 ip 和端口号即可：

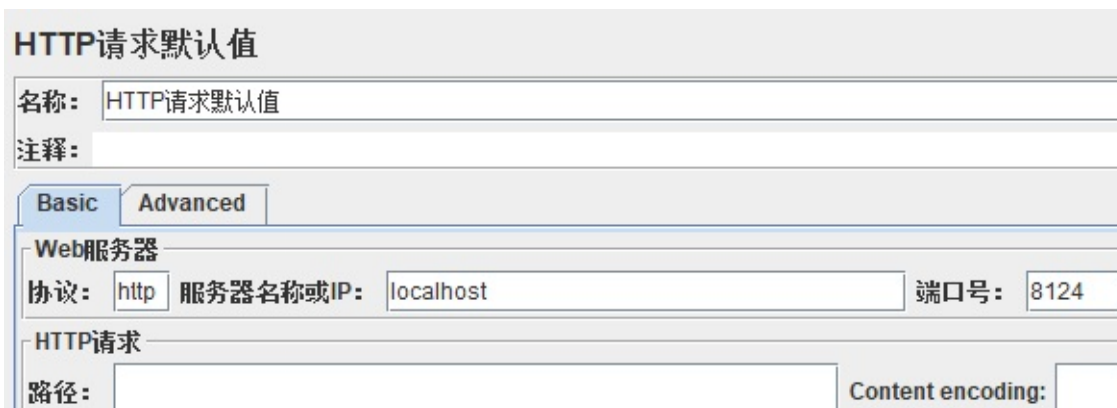


图 11.2.3 设置 HTTP 请求默认参数

其实这个步骤并不是必须的，下面我们要设置一个 http 请求的 sampler，在那里设置请求的 ip 和端口号也是可以的，只不过这里提前设置好了，这样如果你想对当前网站配置多个请求 sampler 的时候，可以省去公共部分的配置，特别当你的请求都有公共参数的时候，不过由于我们这里仅仅测试一个 URL，所以优势没有体现出来。

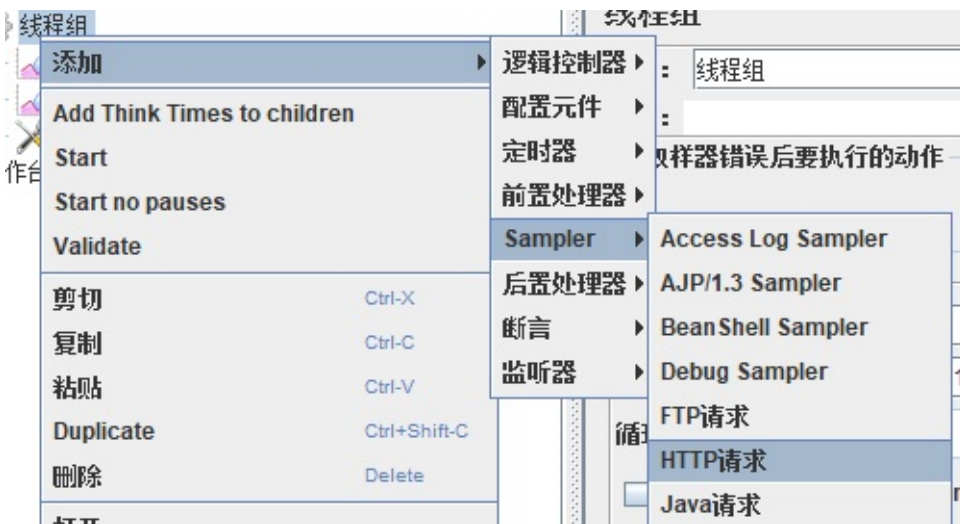


图 11.2.4 添加 HTTP 请求 sampler

由于前面在图 11.2.3 中已经设置了请求的 ip 和端口，这里仅仅设置一下请求路径即可：

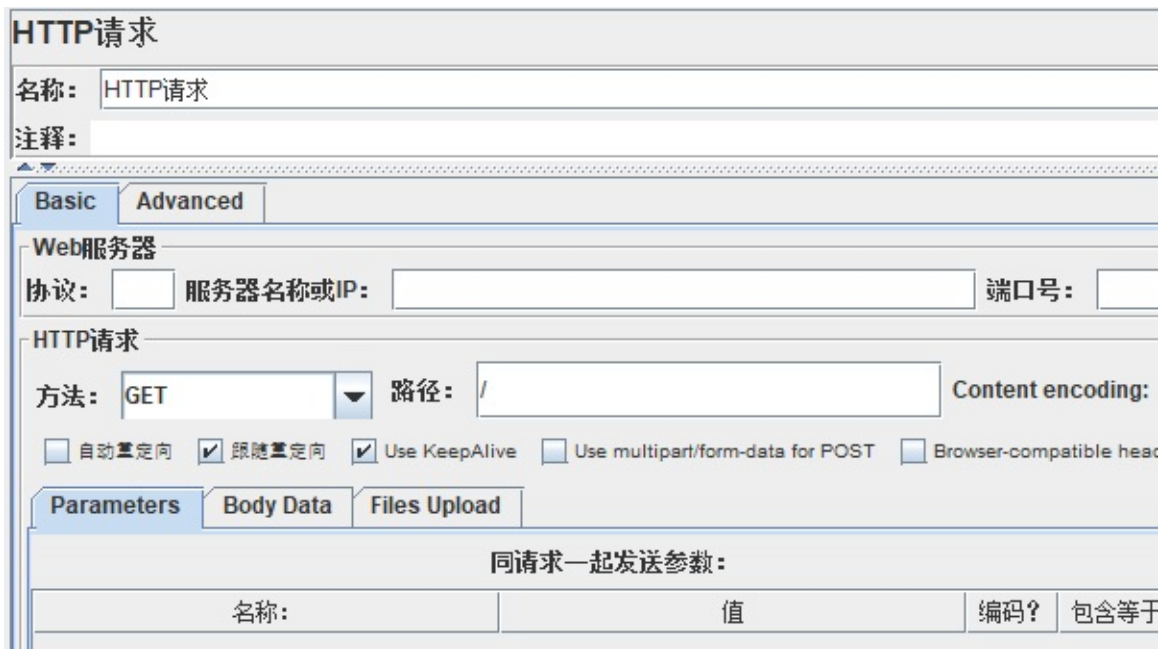



图 11.2.5 配置 HTTP 请求 sampler

最后点击工具栏中的按钮  即可开始测试。

接着我们回到线程组的配置，



里面配置的线程数，可以控制同时发送请求的线程数，我们在做测试的时候逐步增加线程数，然后查看 Node 进程的 CPU 占有率，直到达到 100%（windows 下多核 CPU 的操作系统，显示看是否达到 100%/n，其中 n 为 CPU 核心数，比如说常见的 4 核心的 windows，达到 25%，则代表一颗 CPU 核心被吃满）。

当你的 Node 进程达到满负载之后，回到图 11.1.2 的 Profiler 标签页，选择 Start 按钮开始生成 CPU profile 文件，等待一段时间时候，选择 Stop 则会得到一份 CPU profile，

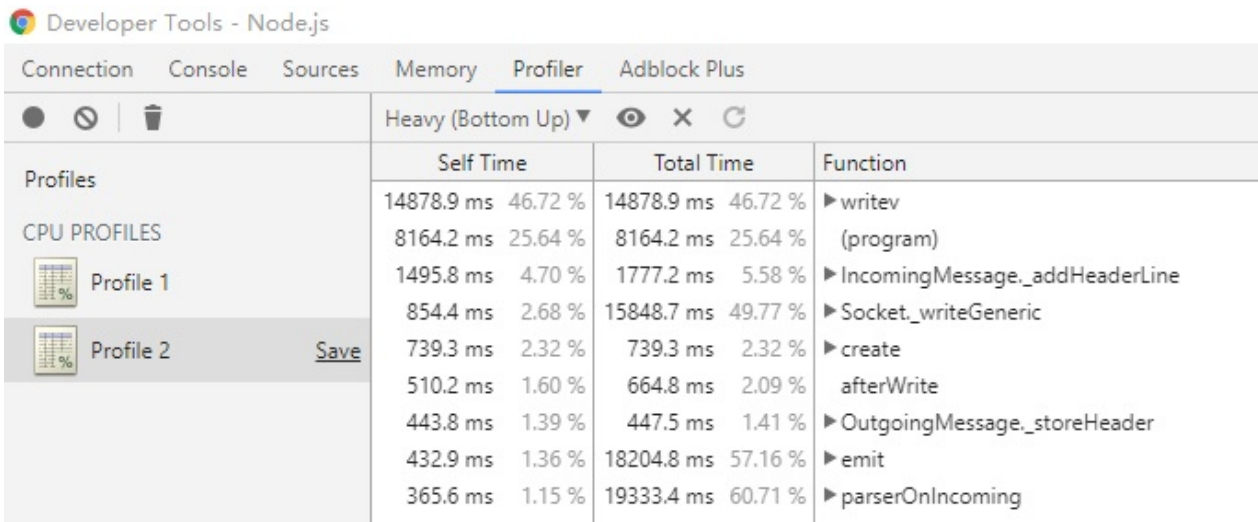


图 11.2.6 生成的 CPU profile 文件

通过分析可知 Socket 的 write 操作和原生代码操作（即第二行标记为 program 的调用）比较耗费 CPU，由于我们在代码 11.1.1 使用了 ccap 这个验证码生成库，而这个库的核心代码是 C/C++ 编写的，所以耗时操作统一算在 program 身上。

回到代码 11.1.1，我们将产生的二维码图片数据作为 HTTP 的响应数据，鉴于一个二维码有几十KB的数据，服务端短时间内大批量的写操作，我们在 JMeter 中添加一个聚合报告（如图 11.2.6所示），然后在压力测试的过程中就会发现网络吞吐量是非常大的（如图 11.2.7 所示）。

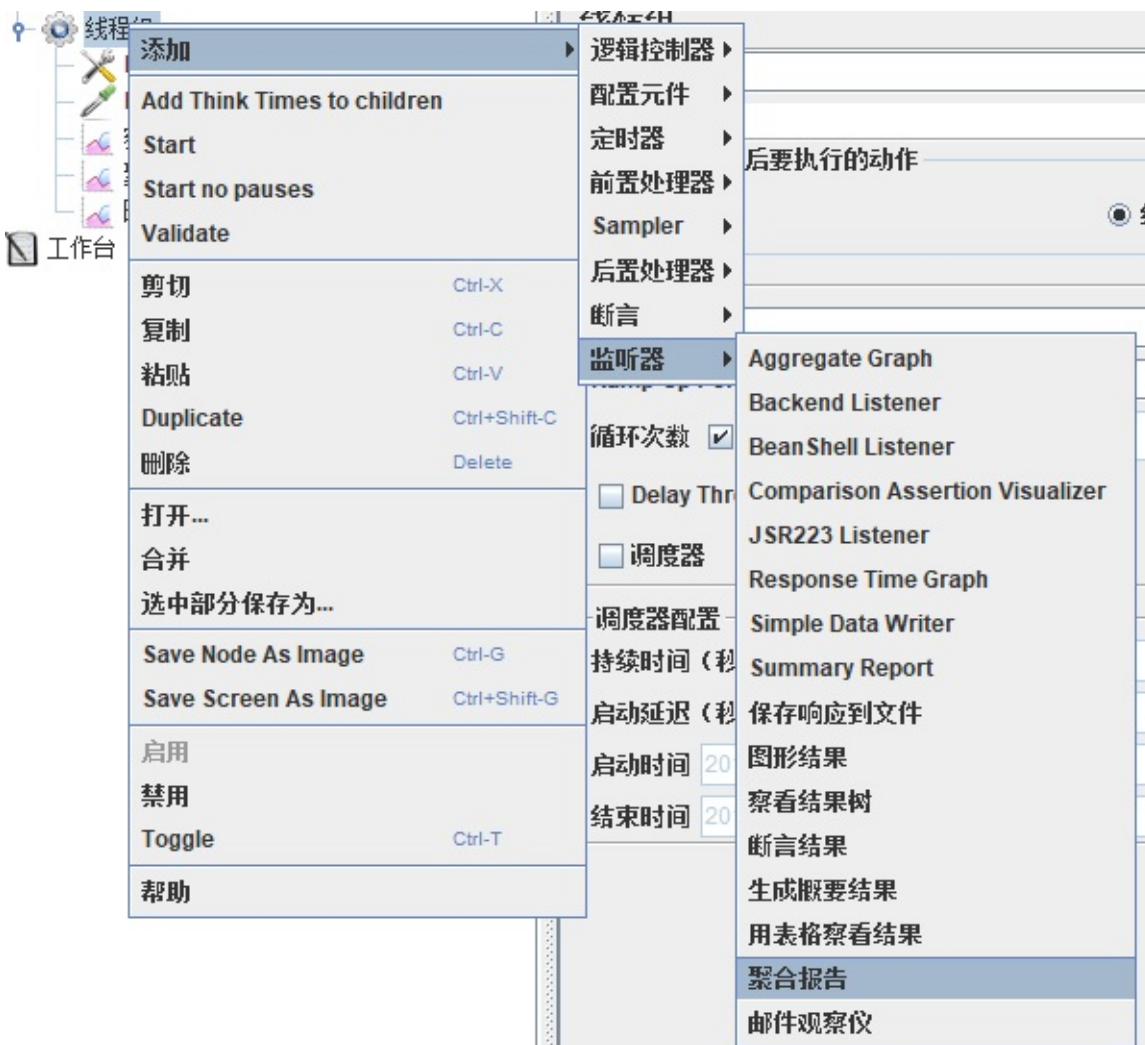


图 11.2.6

聚合报告

名称:聚合报告

注释:

所有数据写入一个文件

文件名

浏览...

Log/Display Only:

☐ 仅日志错误

☐ Successes

Configure

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Err...	Throughput	Received KB/...	Sent KB/sec
HTTP请求	1960803	0	1	1	2	3	0	826	0.0...	2048.0/sec	62294.99	234.0
总体	1960803	0	1	1	2	3	0	826	0.0...	2048.0/sec	62294.99	234.0

图 11.2.7

综上两点，做一个实时的验证码生成程序是不合适的，所以这里推荐的解决方案是提前预生成一批验证码图片，然后放到 **CDN** 上，并且将关联数据存入数据库（比如说 **redis**），等浏览器请求过来的时候再随机从数据库中抽取一个，至于实现代码留给各位同学自己实现了。

11.3 内存分析

Node 的程序中一般不会操作大内存，一般是一个请求过来，处理完数据，变量的生命周期就结束了，会被垃圾回收器回收掉。但是如果有高并发需求时，我们希望从数据库中查询到的数据能够在内存中得到缓存，这样就能减轻对于数据库的压力，提高吞吐率。

我们看下面一段代码：

```
const db = require('./db');
const cacheResult = new Map();
const DEFAULT_CACHE_AGE = 1000;

class CacheItem {
  constructor(data, expire) {
    this.expire = expire || (new Date().getTime() + DEFAULT_CACHE_AGE);
    this.data = data;
  }
}

/*const queryWithCache = */exports.queryWithCache = function(itemName) {
  const item = cacheResult.get(itemName); // console.log(cacheResult.size);

  if (item) {
    if (item.expire > new Date().getTime()) {
      return (item.data);
    }
    cacheResult.delete(item); // console.log('expired cache...');
  }

  const value = db[itemName];
  cacheResult.set(itemName, new CacheItem(itemName, value));
  return (value);
};
```

代码 11.3.1

将从数据库中查询到的数据缓存到内存，不过设置了一个过期时间，下次查询的时候，先判断内存中有没有，如果内存中存在且数据尚未过期，就直接返回。如果内存中存在数据，但是过期了，就将其删除。看上去这个算法比较简单高效，但是却存在严重的内存泄漏问题，缓存的数据只有再下次被重新请求到的时候才有可能被删除，如果每次请求恰好都是新数据，那么之前缓存的数据永远得不到机会删除，导致内存暴涨。

为了方便的复现这个问题，我们对代码**11.1.1**进行改造：

```
const http = require('http');
const ccap = require('ccap')();//Instantiated ccap class
const cache = require('./lib/cache');
const rand = require('./lib/rand');

http.createServer(function (request, response) {
  const url = request.url;
  if(url === '/favicon.ico')return response.end('');//Intercept request favicon.ico
  if (url === '/cache-test') {
    return response.end(cache.queryWithCache(rand.create(3))
    || '___');
  }
  const ary = ccap.get();
  const txt = ary[0];
  const buf = ary[1];
  response.end(buf);
  // response.end('ok');
  //console.log(txt);
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

代码 11.3.2

增加一个地址 `/cache-test` 来模拟这个请求，这里使用随机字符串来达到每次请求都使用新数据的目的。回到 Chrome 开发面板，选择 **Memory** 标签页，然后再选择 **Take heap snapshot**，点击 **Take snapshot** 按钮即可生成堆快照：

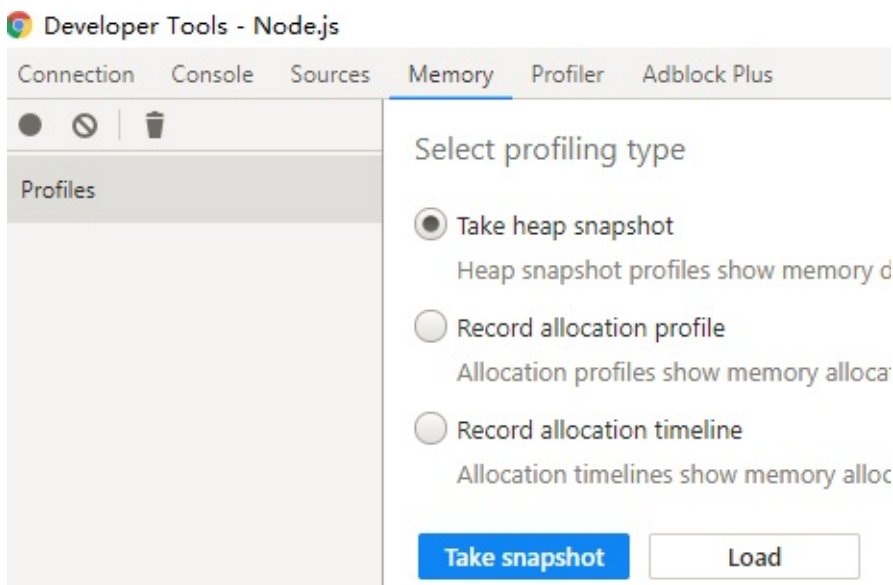


图 11.3.1

回到我们的 JMeter，再新建一个 HTTP sampler，



图 11.3.2

为了单测试内存泄漏，我们在测试之前先把之前的验证码的测试用例禁用掉，然后点击启动按钮。在测试之前，测试过程中，测试结束之后过一段时间之后，分别点击 **Take snapshot** 按钮：

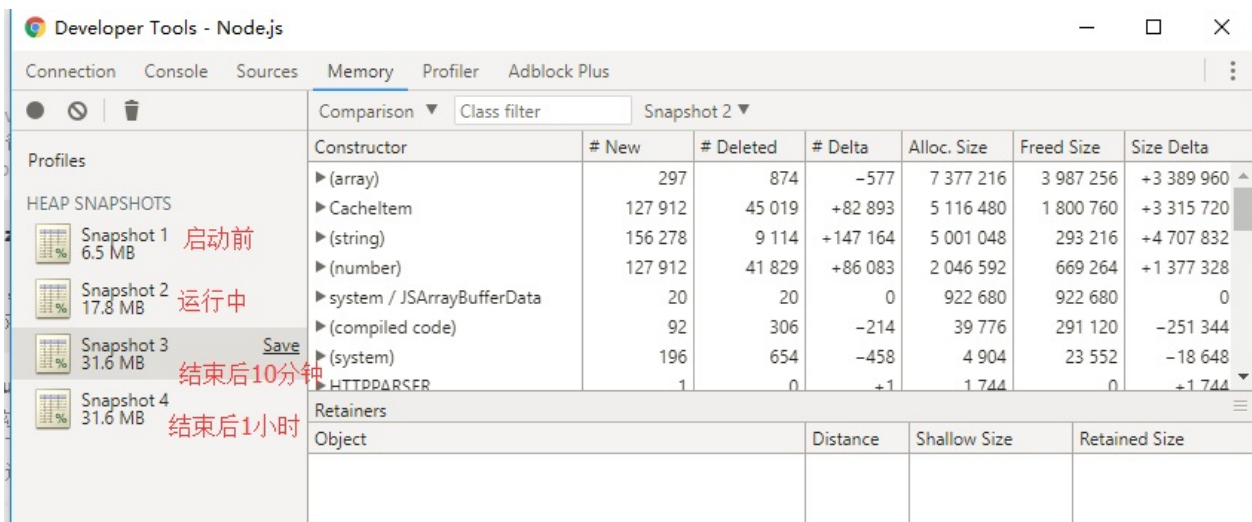


图 11.3.3

Snapshot 1 对应开始前，Snapshot 2 为进行中和 Snapshot 3、4 结束后的堆快照，最终发现即使在测试结束后，堆内存依然没有释放，我们选择 Snapshot 3，然后在 Class filter 左边的下拉框中选择 Comparison，它会自动和上一个快照做对比，我们发现 CacheItem 类型的对象在这当中新增了 82893 个，然后再对比 Snapshot 4 和 Snapshot 3 发现，CacheItem 类型对象完全没有被回收掉。以此我们可以认定，我们写的代码有内存泄漏了。

为了解决这个问题，我单独做了一个闪存处理的包，借鉴了 JVM 或者 V8 中在 GC 中所使用的新生代、老生带的算法，

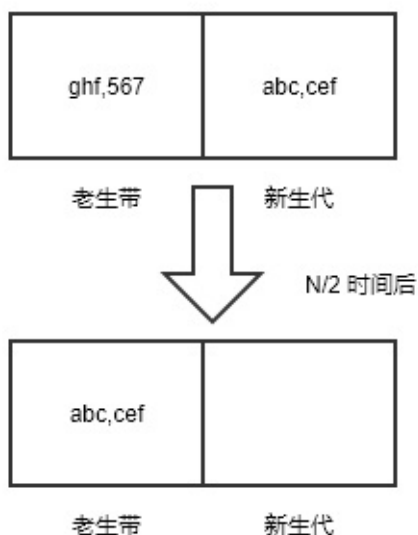


图 11.3.4

假设缓存声明周期为 N ，内部会预置一个定时器，每隔 $N/2$ 时间后将新生代拷贝到老生带，并且清空新生代，具体代码参考包 [flash-cache](#)。

A1 Node.js 好用工具介绍

工欲善其事必先利其器，Node 语言的发展离不开一帮优质第三方库。下面提到的各个包基本上在书中都有提到，这里仅仅是列一个清单，供大家查阅。

A1.1 log.io

Real-time log monitoring in your browser

<http://logio.org/>

```
npm config set unsafe-perm true
npm install -g --prefix=/usr/local log.io
log.io server start
```

```
http://localhost:8998
```

A1.2 log4js

node 日志打印工具，可以在控制台格式化输出日志，可以将日志打印到指定文件，日志文件可以按照日期或者大小进行拆分。我们在代码 **7.1** 中曾经出现过它的身影。

<https://github.com/nomiddlename/log4js-node>

```
npm install log4js
```

A1.3 mongoskin

对于原生 mongo node 驱动进行封装，使其对开发者更友好。在 6.2 章节曾经拿出一整节来讲它的使用。

<https://github.com/kissjs/node-mongoskin>

```
npm install mongoskin
```

A1.4 socket.io

websocket给前端带来了变革，从此前端也可以光明正大的用上长连接，socket.io正是顺应此时势而生的。它在高版本浏览器上使用 websocket，在低版本浏览器上使用 ajax 轮询，保证对所有浏览器的兼容。虽然本书没有对其拿出专门的章节进行介绍，但是它真的很重要。

<http://socket.io/>

```
npm install socket.io
```

A1.5 mocha

鼎鼎大名js单元测试框架，本书专门拿出7.3一个章节对其进行介绍。

<http://mochajs.org/>

```
npm install --global mocha
```

A1.6 nan

如果你是个 node 原生扩展的开发者，一定曾经对于V8各个版本API接口不兼容而大为光火，幸好有了 nan 这个包，它抽象出来了一个头文件来解决这个问题，从此扩展开发者就可以写一套代码运行在各个版本的node上了。本书第8章有对其的内容介绍。

```
npm install nan
```

A1.7 express

在 Node 的web框架领域，如果express敢称第二，没有人敢称第一。虽然我们把他列到了最后，但是不代表他是不重要的。本书专门拿出第5、6两章来讲述他的使用。

```
npm install express
```


A2 参考资源

A2.1 Node.js 书籍

- [深入浅出Node.js](#), 2013-12-01, 朴灵
- [Node与Express开发](#), 2015-01-01, [美] Ethan Brown, 吴海星, 苏文 译
- [ES 6 标准入门（第2版）](#), 2015-12-01, 阮一峰

A2.2 Node.js 影音教学

- [The Node Sessions: The Best of OSCON 2011](#), August 2011, O'Reilly Media (Video)
- [Tom Hughes-Croucher on Node](#), March 2011, O'Reilly Media (Video)

A2.3 Node.js 教学网站

- [Node School](#)

A2.4 Node.js 课程

- [Introduction To Node.js](#), Van Nguyen, CodeLesson

A3 书籍写作规范

A3.1 专有名词

技术类专有名词依循官方常见用法。

- Node.js
- npm
- JavaScript

A3.2 标点符号

以中文全角标点符号为主。

， 、 。 ! “ ” ()

遇英文段落（整句）采用一般（半角）标点符号。中文与英文单字之间以一个空白字符隔开，标点符号与英文单字之间不需要空隔。这是为了让排版显示的自动断词断句可以正确运作，以及增加中英文混杂段落的阅读舒适。举个例子：

Node.js 是一种适合用于 Server-side 的开发框架（Framework），相当 Nice !

A3.3 特殊排版说明

中文同一段落为了方便编辑，可以将句子断行；但行尾必须加上半角“”倒斜线符号。因为在 HTML、EPUB 或 MOBI 格式中，换行字符会被当做空白字符处理，导致增加过多影响版面美观的空隔。

A3.4 代码

片段程序码（snippets），使用 inline code block，并指定 language type。

```
if (something) {  
    alert('test');  
}
```