Node.js 进阶之路

尤嘉　编著

清華大學出版社
北　京

内 容 简 介

本书内容涵盖了 Node.js 高并发的原理、源码分析以及使用 Node.js 开发应用所需要的不同层面的技术实践。具体来讲，本书包括 Node.js 异步机制（配以源码分析）、编辑与调试、测试技术、Docker 部署、模块机制、V8 引擎与代码优化、Promise 和 ES6 Generator、LoopBack 开源框架、使用 C++ 编写扩展、JavaScript 严格模式、编码规范等内容。在 LoopBack 章节，本书详细介绍了使用此框架开发企业级 Web 应用的步骤，帮助读者迅速掌握使用这个强大框架的诀窍。最后一章详细介绍了编写不同类型的 C++ 模块的知识，并对堆内存管理等内容做了深入探讨。

本书适合所有前端和后端的开发人员阅读。

图书在版编目(CIP)数据

Node.js进阶之路 / 尤嘉 编著. — 北京：清华大学出版社，2017
ISBN 978-7-302-45693-3

Ⅰ. ①N… Ⅱ. ①尤… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2016）第 289152 号

责任编辑：袁金敏
封面设计：刘新新
责任校对：徐俊伟
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185mm×230mm　　**印　　张：**12.75　　**字　　数：**211 千字
版　　次：2017 年 1 月第 1 版　　**印　　次：**2017 年 1 月第 1 次印刷
印　　数：1 ～ 3500
定　　价：35.00 元

产品编号：071875-01

前　言

本书写给那些打算或者正在使用 Node.js（简称Node，后文均用此简称）创建 Web 应用的开发者。众所周知，JavaScript 的灵活易用以及 V8 引擎的加速，再加上活跃的社区支持，使得用 Node 开发应用的成本低，收益大。2015 年 ES6 标准的确立，为JavaScript成为企业级开发语言扫除了不确定性。这本书的选材契合这个领域最新的技术进展，深浅适宜地介绍了 Node 技术栈的全貌。

本书共分9章。第1章概述，介绍 Node 异步实现的原理，涵盖了 Node 实现异步的两种方式。这部分引用了 Node 源码，以求逻辑清晰与内容翔实。第2章～第7章是站在 JavaScript 的角度，介绍了用 Node 开发应用的方方面面，包括编辑与调试、测试技术、Docker 部署、模块机制、V8 引擎与代码优化、Promise 和 ES6 generator 等内容。第8章介绍了 LoopBack 开源框架的使用。本书没有介绍 Express（可能读者早已熟悉），因为本书希望为读者引荐一个更加强大易用的企业级 Web 框架。第9章则从 C++ 的角度介绍了 Node 扩展模块的编写，这部分适合那些想要了解 V8 引擎的读者。可以说 C++ 是 Node 技术栈的基石。本书希望向读者呈现构成 Node 技术栈的JavaScript和C++ 全貌。

本书不假定读者有 Node 研发经验，但需熟悉 JavaScript。如果读者最近才接触编程，建议选一本更初级的教程，或者先到 W3School（http://www.w3school.com.cn/js/index.asp）上看看。本书每一章都有源码示例，这些示例大部分可以在 Node 支持的任何系统上运行，但也有例外。建议使用本书第3章介绍的容器，在 Linux 环境下运行本书示例。大部分示例代码可以从 https://github.com/classfellow/node-AdProgramming 下载。

饮半盏湖水，当知江河滋味；拾一片落叶，尽享人间秋凉。希望本书成为读者熟练掌握 Node 技术栈的那一盏湖水、一片落叶。

致谢

感谢 CNode 社区，它提供了一个非常好的平台，本书前期的一些章节从中得到了积极的反馈，使笔者有了继续写下去的动力。首都师范大学的刘晓莲同学，利用周末时间审阅了本书的稿件，提出的一些见解，使得本书在内容安排上更合理，更容易看懂，在此表示感谢。笔者周围的一些同事部分地阅读了初稿并给出了积极的反馈，在此一并谢过！

作者邮箱

classfellow@qq.com

目　录

第1章
Node异步编程范式

本章通过实际案例，向读者介绍 Node 异步编程的优势和原理，这些内容帮助读者理解Node运行的本质。本章还就 Node 实现异步调用的两种机制进行详细的介绍，并引用源码，剖析其内部实现的流程。

1.1 同步与异步的比较

Node是一个JavaScript 运行时环境，它为 JavaScript 提供了一个异步 I/O 编程框架，较其他语言通常使用的同步式方案，其性能好比“搭载上了火箭”。Node的指导思想说起来也简单——CPU执行指令是非常快速的，但 I/O 操作相对而言是极其缓慢的。可以说，Node 要解决的也是这类问题，即给 CPU 执行的算法容易，I/O请求却频繁的情况。

请求到了，相对于传统的进程或者线程同步处理的方式，Node 只在主线程中处理请求。如果遇到 I/O 操作，则以异步方式发起调用，主线程立即返回，继续处理之后的任务。由于异步，一次客户请求的处理方式由流式变为阶段式。我们使用 Node 编写的 JavaScript 代码都运行在主线程。

假设一次客户请求分为三个阶段——执行函数 a，一次 I/O 操作，执行函数 b。如图1-1代表了同步式的处理流程。

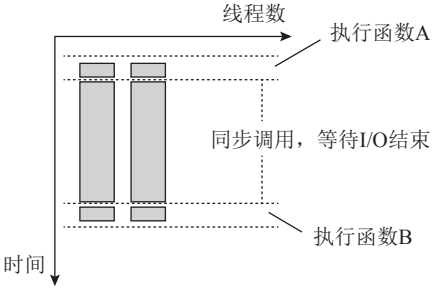


图1-1　同步式的处理流程

可以用一段伪代码描述同步请求的过程，如下所示：

```
// 代码 1-1
function request() {
  //开始执行函数 a
  $rel_a = stage_a();
  //读文件，将文件内容返回到 $data
  $data = readfile();
  //将前两步的结果作为参数，调用函数 b
  stage_b($rel_a, &data);
}
```

可见，同步式处理方式中，每个请求用一个线程（或进程）处理。一次请求处理完毕之后，线程被回收。同步式的方式只画出了两个线程，如果有更多客户请求，线程数还要增加。与之相比较，如图1-2所示则是 Node 异步执行的示意图。可见，Node 一个主线程解决了所有的问题。这种异步式处理流程中，每一个方块代表了一个阶段任务的执行。

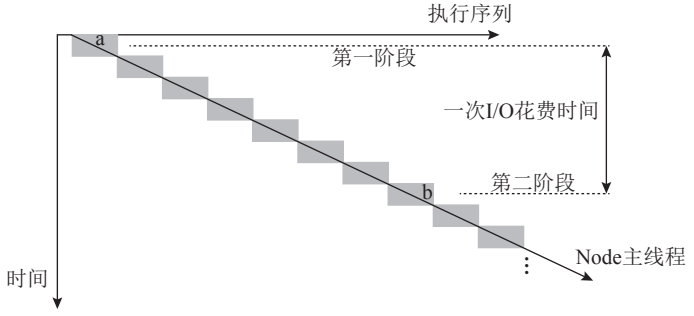


图1-2　Node异步执行示意图

对比同步，异步同样可以用一段伪代码表达Node异步的处理方式，如下所示：

```
// 代码 1-2
var request = function(){
  //开始执行函数 a
```

```
  var rel_a = stage_a();
  //发起异步读取，主线程立即返回，处理之后的任务
  readfileAsync(function(data){
    //在随后的循环中，执行回调函数
    stage_b(rel_a, data);
  });
}
```

Node也“站在巨人的肩上”。这个“巨人”是大名鼎鼎的 V8 引擎，有这样一个强大的“心脏”，再配合基于高阶函数和闭包的异步编码范式，使得用 Node 构建的程序在性能上有着出色的表现。

高阶函数与闭包是两个联系非常紧密的概念。如果一个函数以一个或多个函数作为参数，或者返回一个函数，那么称此函数为高级函数。Node 中大部分的异步函数，参数列表的最后一项接受一个回调，这类异步函数就符合高阶函数的定义。高阶函数执行后返回的函数，或者接受的函数参数，则被称为闭包。闭包的最大特点是引用了它之外的变量，这些变量既不是全局的，也不是参数和局部的，而是作为闭包执行时的上下文环境存在。如下所示：

```
// 代码 1-3
function wrapper(price){
  var freeVal = price;
  function closure (delta){
      return freeVal * delta;
```

```
  }
  return closure;
}
var clo1 = wrapper(100);
var clo2 = wrapper(200);
setTimeout(function(){
  console.log(clo1(1));
  console.log(clo2(1));
}, 500);
```

代码1-3运行后输出的结果如下：

```
100
200
```

函数 wrapper 是一个高阶函数，执行后返回一个闭包，这个闭包将 price 纳入自己的作用域，price 就不再是函数内部的局部变量，它有一个名字叫自由变量，其生命期与闭包绑定。price 这样的自由变量被闭包内的代码引用，成为闭包执行的上下文。

Node高性能的来源，得益于它异步的运行方式。可以举一个例子来理解异步对性能的提升。按照目前北京出入车辆管理规定，外地来的车需要办理进京证，而办进京证需等待一定时间。如果每个人都自己跑去办理，就好比开启多个线程同步处理，办理窗口有限，就得排队。而把这件事委托给第三方，就好比不开启线程或进程，把耗时的I/O请求委托给操作系统。这种情况下人们从办证的任务中解放出来，因而能继续做其他事情。若来了一个任务，交给第三方去处理，则第三方就有一个接单队列，其只需拿着所有的接单，去办理地点逐个办理即可。

如图1-3所示是办理进京证的示意图。假设有20个人，每人开车来回花费2小时，往返油费60元，办理窗口有两个，在窗口办完一个进京证平均需要5分钟。不考

虑排队时间，则20个人办证花费的总时间至少是20×2 + 20×5÷60小时，总油费是20×60元。按照目前北京市最低工资标准推断，假设有车族时薪为100元，则总成本为(20×2+20×5÷60)×100+20×60元，这个数字大概是5367元。下面来对比一下异步办理进京证的花费。

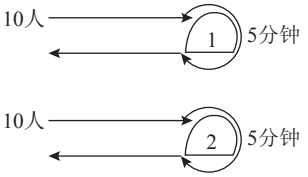


图 1-3　同步办理进京证示意图

客户将资料交给代理者，此人拿着 20 个客户的资料，一个人办理完所有事情。同样假设此代理者的时薪为 100元，则总花费为(20×5÷60)×100+60元，这个数字大概是 227元。还应该注意到，这种方式还省去了一个办理窗口，如图1-4所示。

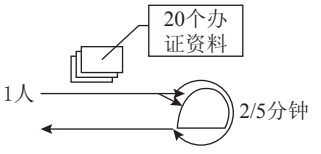


图 1-4　异步办理进京证示意图

在上面的讨论中，将任务委托给第三方，至少涉及两个细节，一是上下文，二是后续动作如何对接。以办证为例，客户需要把自己的一些资料交给代理人。办完之后，代理人需要有一种方式将结果交给客户。在同步办证逻辑中，所谈的这两个细节是不需要特别提出的。人们排队去窗口办理，轮到的人，说明来意，交出资料，耐心等待；办证窗口里面的工作人员办理好后，把新证交给窗口处的等待者即可。但在异步办证的逻辑中，所谈的这两个细节就需要认真考虑了。代理者去一个窗口办理，因为办证处仍然具有同时处理两个请求的能力，所以代理者同时交出两个客户资料请求办理。假设这个代理人把 A 和 B 的资料交给窗口，只过了 4 分钟，就办好了一个，此人需要判断此证属于 A 还是 B，不能搞混了，否则他后续的步骤将会出错。这种因为异步带来的返回结果次序的不确定性，是异步编程框架需要解决的一个问题。

假设有一个异步读取文件函数 readAsync，在主线程中，调用此函数发起了一个异步读取操作。因为执行的是异步函数，所以主线程立即返回，继续处理其他任务。操作系统执行完具体的读取操作后，将数据准备好，通知主线程。这个过程需要解决的一个问题是，当主线程得到通知后，如何识别异步请求是谁发起的，以及得到数据之后下一步该做什么。前面讲到了闭包，这里恰恰就需要闭包的特性对接执行流程。因为闭包保留了异步调用发起时的上下文信息，于是执行闭包，将结果传入，就可沿着发起读取文件时的执行环境继续运行后续逻辑。

设想如果来了1000个并发请求，按照创建进程或线程的同步处理方式，一个服务器运行这么多进程或者线程，使CPU频繁地在上下文切换是多么低效。CPU 和内存资源的内耗挤压了真正花费在处理业务逻辑上的时间，造成服务器性能下降，但假如服务器少开线程，则又会造成请求排队等待。而在异步方式下，不需要开多余的线程，所有请求均由主线程承担。一旦遇到耗时的 I/O 请求，以异步的方式发起调用。一次异步调用同时也会创建一个闭包传进去，操作系统执行具体的 I/O 操作，将结果放入指定缓存，然后将本次 I/O 置为就绪状态。主线程在每一次循环中，收集就绪的请求，取出原先的闭包，然后调用回调函数并将结果传入。这个过程不需要创建任何多余的线程或进程。在1.2节，将以实例从脚本和 C++ 层面描述这个过程。

Node能够最大限度利用硬件资源，其原因即如此。换句话说，CPU 把绝大多数时间花在处理实际业务逻辑上，而不是线程或进程的等待和上下文切换上。在这种处理模式下，假如主线程阻塞，那说明真的是没有任务需要处理，而不是等待 I/O 结束。

1.2　Node异步的实现

本节从一次HTTP请求来跟踪Node内部的执行过程，读者可以从中了解 Node 是如何工作的，然后再将网络 I/O 与本地磁盘 I/O 做对比，指出 Node 实现异步调用的两种机

制。读者在阅读本节时，可以比照着 Node 源码来看，以加深理解。

1.2.1 HTTP请求——完全异步的例子

HTTP是一个应用层协议，在传输层使用TCP协议与服务器进行数据交互。一次HTTP请求分四个阶段，分别是连接、请求、应答、结束。如下代码是一段发起HTTP请求的代码，下面就以这段代码为例，看看 Node 是如何以异步的方式完成上述四个过程的。

```
// 代码 1-4
"use strict";
var http  = require('http');
var options = {
  host: 'cnodejs.org'
  ,port: 80
  ,path: '/'
  ,method: 'GET'
  ,headers: {
    'Content-Type': 'application/json'
  }
};
var req = http.request(options);
req.once('response', (res)=>{
  var result = '';
  res.on('data', function (chunk) {
    result += chunk.toString();
  });
```

```
  res.on('end', function () {
    console.log(result);
  });
});
req.on("error", (e)=>{
  console.error(e.message);
});
req.end();
```

response 事件回调函数的写法使用了 ES6 标准的箭头函数，如果不关心函数名，这种写法可以使代码更加简洁。ES6 引入了大量新的语法特性，小到能够简化代码的“语法糖”，例如箭头函数（arrow functions）和解构赋值 (destructing)；大到原生支持 Promise 对象和生成器 (Generator)。其目标是使得 JavaScript 可以用来编写大型的、复杂的应用程序，成为企业级开发语言。关于 ES6 的更多知识，读者可以参考http://es6.ruanyifeng.com/#README。

文件开始的 "use strict" 语句，表示以严格模式运行。这里为了简单一些，将 Buffer 对象转换成字符串处理了。在实际中，应该始终以 Buffer 对象操作数据。下面我们先来看连接阶段的执行。

1. 连接阶段

当调用 http.request 函数发起请求之后，Node 会建立与服务器的 socket 连接，而socket连接是需要IP地址和端口号的，所以中间还有一个 DNS 解析过程。http.request 函数返回的对象req代表了这次HTTP请求，其监听的response事件在连接建立之后，服务器的HTTP

头信息解析完毕之后触发。此对象的构造函数源码位于_http_client.js，这个构造函数将近200行代码。为了方便说明，在上述示例代码实际执行的路径上，截留出关键的代码段，如下所示：

```
// 代码 1-5
function ClientRequest(options, cb) {
  var self = this;
  OutgoingMessage.call(self);
  var agent = options.agent;
  var defaultAgent = options._defaultAgent || Agent.globalAgent;
  self.agent = defaultAgent;
 if (self.agent) {
   if (!self.agent.keepAlive && !Number.isFinite(self.agent.maxSockets)) {
     self._last = true;
     self.shouldKeepAlive = false;
    } else {
      self._last = false;
      self.shouldKeepAlive = true;
    }
    self.agent.addRequest(self, options);
  }
  self._deferToConnect(null, null, function() {
    self._flush();
    self = null;
  });
}
```

代码1-5获取了一个全局的 Agent 对象，这个全局的对象维护了一个 socket 连接池。addRequest()方法先寻找是否有备用的连接，如果有，则直接获取一个与被连服务器保持 TCP 连接的 socket 对象，否则调用 createSocket()方法，创建一个新的socket对象。大致过程如下所示：

```
// 代码 1-6
Agent.prototype.addRequest = function(req, options) {
  var freeLen = this.freeSockets[name] ? this.freeSockets[name].length : 0;
  var sockLen = freeLen + this.sockets[name].length;
  if (sockLen < this.maxSockets) {
    var newSocket = this.createSocket(req, options);
    req.onSocket(newSocket);
  } else {
    debug('wait for socket');
    //We are over limit so we'll add it to the queue.
    if (!this.requests[name]) {
      this.requests[name] = [];
    }
    this.requests[name].push(req);
  }
};
```

我们看到，createSocket()方法返回一个新创建的socket对象newSocket，然后以这个对象为参数，立即调用 req 对象的 onSocket()方法。req就是构造函数 ClientRequest()正在构造的对象，onSocket()方法只有一行代码：

```
process.nextTick(onSocketNT, this, socket);
```

这意味着在下一轮循环才执行 onSocketNT()函数，这个函数的执行会触发socket事件。因为在调用_deferToConnect()函数的过程中，req对象监听了此事件，而监听这个事件的意义在于，在执行此事件的回调函数时，让newSocket对象监听connect事件。

这里的重头戏在createSocket()函数的调用上，这个函数实际上调用了net.js中定义的构造函数createConnection()，这个函数的定义如下所示：

```
// 代码 1-7
exports.connect = exports.createConnection = function() {
  const argsLen = arguments.length;
  var args = new Array(argsLen);
  for (var i = 0; i < argsLen; i++)
    args[i] = arguments[i];
  args = normalizeConnectArgs(args);
  var s = new Socket(args[0]);
  return Socket.prototype.connect.apply(s, args);
};
```

代码1-7中的倒数第二行创建了 newSocket 对象，此对象代表此次HTTP请求的socket连接，记录着连接的各种状态信息，例如是否可写、可读，并负责监听与 socket 状态相关的事件。最后一行明确以这个对象为上下文，调用 connect()方法。这个方法的大致流程如下所示：

```
// 代码 1-8
Socket.prototype.connect = function(options, cb) {
    if (!this._handle) {
       this._handle = pipe ? createPipe() : createTCP();
       initSocketHandle(this);
```

```
    }
    var dns = require('dns');
    dns.lookup(host, dnsopts, function(err, ip, addressType) {
      if (err) {
      } else {
        self._unrefTimer();
        connect(self,
                ip,
                port,
                addressType,
                localAddress,
                localPort);
      }
    });
  return self;
};
```

上面只写出了主要的执行流程，关键的步骤是调用createTCP()函数，构造了一个 TCP 对象，然后调用dns模块的lookup()方法进行DNS解析。createTCP()函数的定义如下所示：

```
// 代码 1-9
function createTCP() {
  var TCP = process.binding('tcp_wrap').TCP;
  return new TCP();
}
```

可见，TCP 对象的构造函数是在 tcp_wrap 模块中定义的，对应于函数 TCPWrap::New()，源码位于 tcp_wrap.cc 中。这个由 C++ 为 JavaScript 编写的构造函数附

带了诸多的原型方法，JavaScript 层面的对象能够直接调用这些原型函数。因此，借由这些 C++ 版本的方法，JavaScript 拥有了异步读写操作系统层面 socket 描述符的能力。这些原型函数大体分两类，一类维护 socket 描述符状态，例如 connect、listen、bind等。还有一类是读写 socket，包括 writev、readStart 等。创建了 TCP 对象之后，接下来调用dns模块的 lookup()方法，执行此函数会立即返回。于是本轮调用栈依次退出，req 对象的构造过程结束。

http.request 调用返回之后，紧接着调用了req的end()方法。调用此方法意味着客户端要发送的数据已经全部写完。事实上，数据仅仅是被缓存，因为连接还没建立。例子中请求类型是GET，因此这里的数据仅仅是HTTP的请求头信息。随后我们将看到，客户端真正向服务器发送数据的时机是在socket连接完毕执行回调函数的时候。

回到代码1-8，在DNS解析完毕之后，逻辑继续往下走，调用了 net.js 内部的 connect() 函数。此时一切就绪，开始用端口号和解析得到的IP地址发起 socket 连接请求。

```
// 代码 1-10
function connect(self, address, port, addressType, localAddress, localPort) {
     //...
     const req = new TCPConnectWrap();
    req.oncomplete = afterConnect;
    req.address = address;
    req.port = port;
    req.localAddress = localAddress;
    req.localPort = localPort;
    if (addressType === 4)
        err = self._handle.connect(req, address, port);
}
```

self._handle是之前创建的TCP对象，在JavaScript 层面调用connect()方法，会直接引起

C++ 层面的 TCPWrap::Connect 方法的调用。到达这个函数之后，再继续往下执行，则涉及调用操作系统的 API 建立连接。这里以 Linux操作系统为例，Node使用Epoll处理异步事件。下面简要描述进入TCPWrap::Connect ()函数之后的过程。大致分为四步。

（1）创建一个非阻塞的 socket；

（2）调用系统函数 connect()；

（3）调用 uv_epoll_ctl()将创建的非阻塞 socket与一个已存在的Epoll的句柄关联起来；

（4）调用 uv_epoll_wait()函数，收集就绪的描述符，然后执行相应的回调函数。

前两步体现在tcp.c的uv_tcp_connect()函数中。第（3）步和第（4）步稍微复杂一些，调用完函数uv_tcp_connect()之后，其实并未关联Epoll句柄。第（3）步和第（4）步的操作在后一个线程循环中进行。具体代码在linux-core.cc的uv_io_poll()函数中。

继续看代码1-10，传给connect()函数的第一个参数是TCPConnectWrap构造函数创建的 req 对象，此对象代表这次 TCP 连接阶段。当连接过程结束，此对象的生命期也终结。这个对象的构造函数也是在 C++ 文件中定义的。这是因为此类对象未来要在 C++ 层面访问，其一些需要的特性需要在构造的时候配置，这就需要借助 C++ 层面的构造函数实现。之后为这个 req 对象增加了一个 oncomplete 属性，其值是一个名为 afterConnect()的函数。传给connect()方法的参数中没有回调函数，连接过程结束，JavaScript 的代码是如何得到通知的呢？在上述第（4）步中，回调函数的调用栈最终会到达 TCPWrap::AfterConnect()，这个函数中执行了下面的语句：

```
req_wrap->MakeCallback(env->oncomplete_string(), arraysize(argv),
argv);
```

env->oncomplete_string()代表的字符串是 oncomplete ，这个调用实际上从 C++ 层面调用了 req 对象的 afterConnect()方法(此处req是构造函数TCPConnectWrap创建的对象)。因此afterConnect相当于回调函数，该函数内部触发connect事件。之前提到socket 对象监听了connect事件，因此一旦连接成功，此事件的回调函数被调用。在该函数中，调用了_flush方法，客户端开始向服务器发送数据。至此为止，连接阶段结束。

从对连接阶段的分析知，Node 实现异步流程的基本方式是从 JavaScript 代码发起请求，借助 V8 的编程接口，进入 C++ 代码，使用操作系统提供的异步编程机制，例如 Linux 下的 Epoll，向操作系统发起异步调用，然后立即返回。主线的消息循环会调用uv_io_poll()函数，此函数的主要任务就是不断收集已经处于就绪状态的描述符，顺序调用相应的回调函数，执行回调函数会从 C++ 回到上层 JavaScript 层面，最终调用相应的JavaScript 版本的回调函数。这样一圈下来，逻辑闭合。

2. 请求与应答阶段

前面比较详细地介绍了 TCP 的连接过程。Node 读写 socket 也是异步方式，流程与上述类似。上面的讨论中提到 afterConnect()方法会触发connect事件，此事件代表连接过程完成，客户可以向 socket 写数据了。socket 对象创建之后，同时还监听了data事件，其回调函数是 socketOnData()，此方法在 _http.client.js 中定义。服务器发送过来的任何数据，都会触发这个函数执行。传输层不考虑数据格式，对数据格式的解析应该在获取数据之后开始。我们看到 socketOnData 中调用了HTTP协议解析器，边接收数据边解析，如下所示：

```
// 代码 1-11
function socketOnData(d) {
  var socket = this;
  var req = this._httpMessage;
  var parser = this.parser;
  var ret = parser.execute(d);
  //...
```

parser 是一个由 C++ 层面的构造函数创建的对象，相关源码在 node_http_parser.cc 中。解析的工作由 C++ 的代码完成，parser 对象的 execute()方法对应于 C++ 的Parser::Execut()函数。一旦服务器的HTTP头解析完毕，会触发 parserOnIncomingClient 函数的执行，此函数也定义在 _http.client.js，这个函数会触发response事件。代码1-11在调用http.request()函数返回后，监听了此事件。在事件函数中，传入的 res 对象又监听了data和

end事件，后续便可获取数据和得到数据发送完毕的通知。

3. 结束阶段

如果HTTP请求头包含 Connection:keep-alive（默认自动添加），那么当客户端收到服务器端的所有应答数据之后，与服务器的 socket 连接仍然保持。此 socket 对象将会缓存在全局 Agent 对象的 freeSockets 中，下次请求相同的地址时，直接取用，节省 DNS 解析和连接服务器的时间。

1.2.2　本地磁盘I/O——多线程模拟

在1.2.1节HTTP请求的例子中，我们看到 Node 的异步调用，其背后的机制是 Linux 的 Epoll。在 Windows 下，则是采用完成端口（IOCP）。这两种方式的共同点是没有启用任何其他的线程。首先，Node 的 C++ 代码在执行异步请求时没有创建线程，也没有占用线程池的资源；其次，使用 Epoll 或 IOCP，没有隐式地引起更多线程的创建，也不存在线程被阻塞等待 I/O 完成的情况，我们可以称上述过程完全异步。对于网络 I/O是这样，但对于本地的磁盘 I/O，Node 使用了不同的策略。与远程调用相比，本地磁盘 I/O 具有不同的特点：

（1）本地磁盘读写要比网络请求快得多；

（2）磁盘文件按块儿访问，操作系统的缓存机制使得顺序读写文件的效率极高；

（3）相对于同步的读写，完全异步的处理流程复杂得多。

关于上述第（3）点，读者若仔细看了第一个示例代码1-4中列举的 TCP 连接过程，应该会同意。但也许真正值得考虑的是第（1）和第（2）点。Node 在非主线程中执行同步代码，用多线程的方式模拟出异步的效果，基于上述前两点考虑，比起完全异步的方案，效率应该不会低多少。除本地磁盘操作，第一个示例提到的异步解析 DNS也是如此。本书第9章将介绍使用线程对象输入日志的方案，同样也是基于上述考虑。

Linux 下 Node 在启动时，会维护一个线程池，Node 使用这个线程池的线程，同步地读写文件；而在 Windows 下，则是利用 IOCP 的通知机制，把同步代码交给由操作系统管

理的线程池运行。

读者可能会困惑为什么在 Windows 下，主线程的异步调用都使用了 IOCP 的通知机制，怎么区分完全异步和模拟出的异步？因为读写本地文件时，是以同步的方式调用 ReadFile 或者 WriteFile 这类 API，线程会等待，直到 I/O 过程结束，才会执行下面的语句。虽然这个过程没有使用 Node 自身的线程池，但一样会消耗操作系统线程池的资源。而对于 socket I/O，主线程调用完这类函数会立即返回，不存在线程因为等待 I/O 而无法处理其他任务的情况。本章参考资料（5）包含一份代码，提供了一个使用 IOCP 但用完全异步（将文件句柄与 IOCP 句柄关联，以 OVERLAPPED 方式调用 ReadFile）的方式读取文件内容的一个例子。

1.3 事件驱动

根据上面的讨论，读者应该对事件驱动这个概念有了更切实的体会。我们平常在电脑上使用的应用程序、鼠标或者键盘事件驱动着程序状态的改变。对 Node 来讲，以1.2.1节讨论的 http 请求为例，JavaScript 代码中出现的 connect、data、end、close 等事件，驱动着程序的执行。从真实的编程角度看，所谓的事件就是向操作系统发起异步调用之后，在以后某个时刻，所期待的结果发生了，这便是一个事件。

Node 内部函数 uv_epoll_wait 收集所有就绪的事件，然后依次调用回调函数。如果上层监听了相应事件，则依次调用对应的事件函数。例如发起一个 socket 连接请求，当连接成功后，从 C++ 进入到 JavaScript 层面后，会触发 connect 事件，其事件函数被依次调用。因为 JavaScript 的闭包机制，调用之前的上下文依然保留，程序可以接着向下执行已经连接之后的逻辑。

在1.1节，我们讨论了异步编程的诸多好处，主要体现在节约 CPU 资源，不阻塞，快速，高响应。但还有一点没有谈到，就是异步的编码范式几乎用不到锁，这在 C++ 层面

尤其体现出它的优势。我们惊奇地发现，不仅所有 JavaScript 代码均运行于主线程，对于完全异步的情况，C++ 代码也不需要使用锁。因为所有代码也运行在主线程，对象内部状态都在一个线程中维护。

本章讨论了一些基础的概念，并在源码的基础上，以实例对 Node 的运行机制进行了一个较完整的展示。现在我们已经对 Node 有了相当的理解，也折服于其事件驱动带来的高效能。在接下来几章将探讨更多的内容，以更好地使用这门技术，在以后的工程实践中，做到游刃有余。

参考资料

（1）https://github.com/nodejs/node

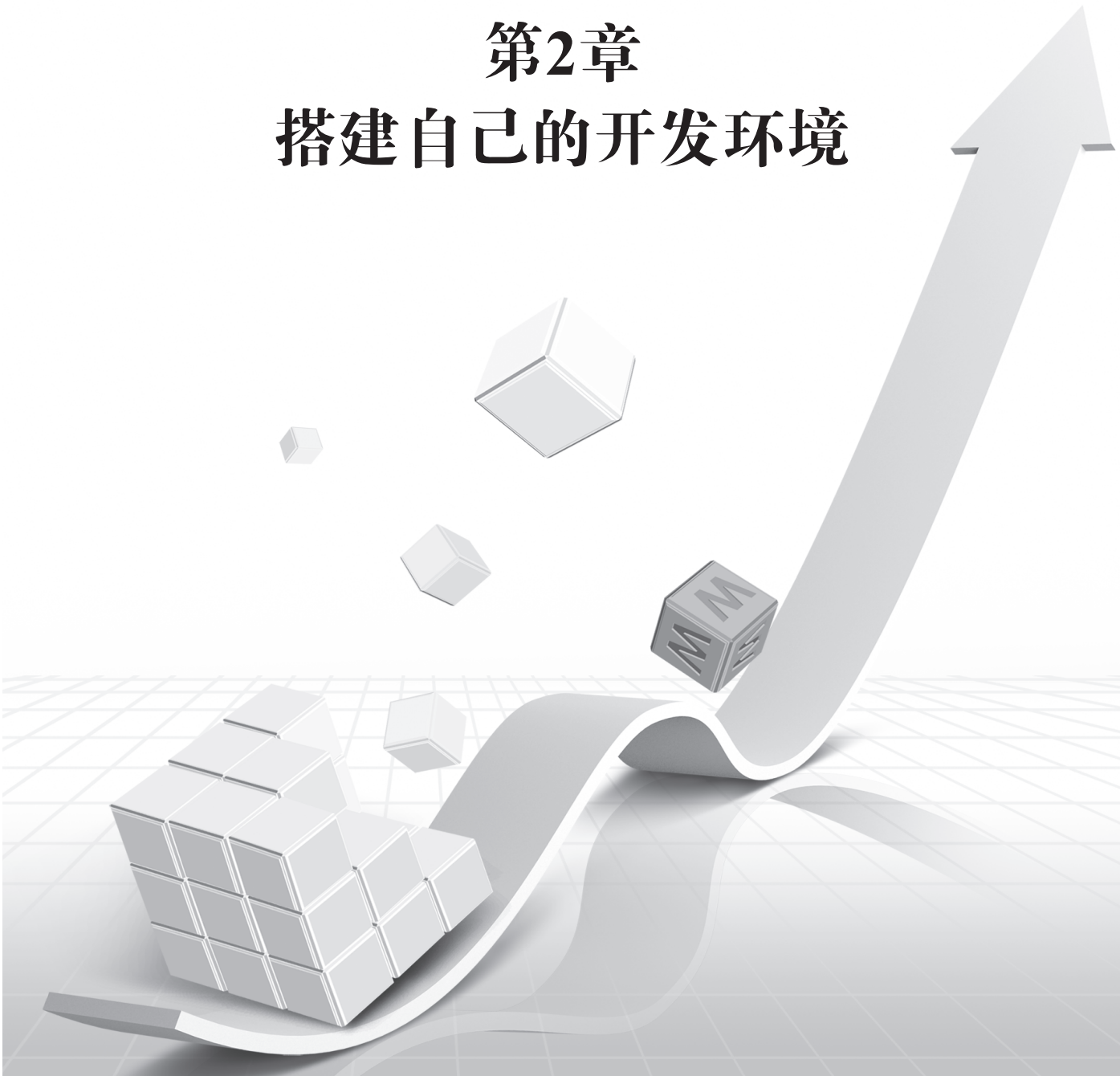
（2）http://docs.libuv.org/en/v1.x/index.html

（3）https://nodejs.org/dist/latest-v4.x/docs/api/

（4）http://blog.csdn.net/xiajun07061225/article/details/9250579

（5）https://github.com/classfellow/async-read

第2章
搭建自己的开发环境



本章的内容包括 Node 的编译和安装，IDE 开发环境，代码调试，编写单元测试等内容。编译与安装以及对 IDE 的介绍，适合从零开始学习的读者，而对于已经能非常顺手地搭建开发调试环境的读者，可酌情跳过。在 IDE 开发环境和代码调试的部分，本书选取 Visual Studio Code 介绍。最后一节将介绍测试框架 Mocha 的使用。

2.1 Node的编译与安装

下面以 Linux 平台为例，介绍如何编译安装。Node 的编译环境要求如下。

- gcc/g++ 的版本不能低于 4.8；
- Python 2.6 或者 2.7；
- GNU Make 的版本不低于 3.81。

可以分别运行如下指令检查是否满足。

- gcc -v；
- g++ -v；
- python -V；
- make –v。

读者可以从 Node 官网下载源码包，下载完毕并解压之后，进入源码包的文件夹node-< version > 按顺序分别运行如下命令。

- ./configure --prefix=/usr/local/Node-< version >；
- make；
- make install。

这里需要注意的是，执行上述命令时，应该在 root 权限下。Node 的编译过程比较耗时，等待结束之后，进入 /usr/local 目录可以看到文件夹node-< version >。然后进入 /usr/local/node-< version >/bin，可以看到编译好的可执行文件 Node。进入 /usr/bin 目录，为

node和npm建立软链接，使之在任何目录下均可运行。

读者也不必非得亲自编译，只需去官网找到对应操作系统的二进制版本下载，解压到/usr/local 中，然后建立软链接即可。

使用 Node 开发程序，经常下载使用第三方模块。有些模块包含 C++ 代码，使用npm命令安装的时候会编译这些C++文件。此时如果操作系统的 GCC 版本过低，则编译 C++模块会出错。而编译高版本的 GCC 非常耗时，并且步骤烦琐，这种情况下，建议读者使用容器运行 Node 程序。下一章将介绍相关内容。

2.2　开发与调试

“工欲善其事，必先利其器”，一款好的 IDE 可以极大地提高我们的工作效率。Visual Studio Code 是微软推出的一个跨平台的 Web 开发环境。其编辑功能支持代码补全、自动语法检查、括号匹配、语法高亮等。它还支持插件扩展，并且完全免费。VS Code 也是一个非常优秀的调试环境，在图形界面下支持鼠标下断点、单步执行、展示变量信息和堆栈信息。在实际的开发工作中，我们可以在本地编辑调试代码，使用 Git 提交之后，在服务器端 pull 代码重启或运行。如果工程根目录下有 .git 目录，则可以借由它的界面执行 git add -A、git commit、git push 等命令。

在 Visual Studio Code 官网上下载对应平台的最新版本，安装完成之后启动程序，可以看到它的界面，如图2-1所示。

双击打开文件夹，选择一个项目的根目录，假设目录中包含一个 test/test.js 的文件，我们看一下如何使用 VS Code 调试此文件。

VS 系列的开发环境中，F5 代表调试运行。当按下F5键，第一次会弹出如图2-2所示的菜单。

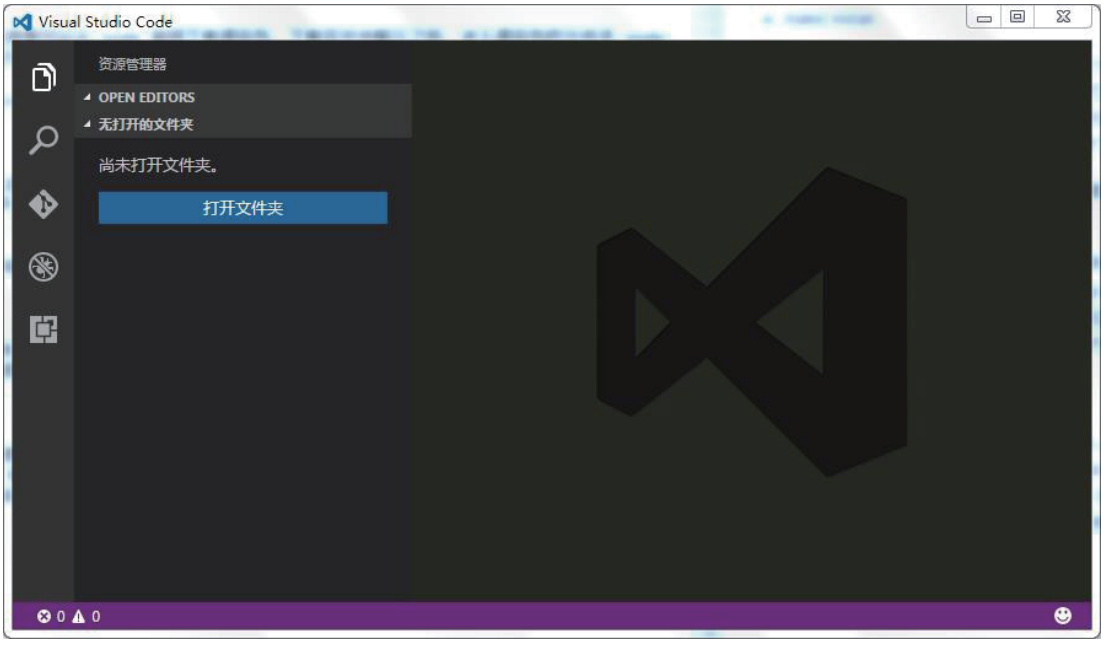


图 2-1　Visual Studio Code界面

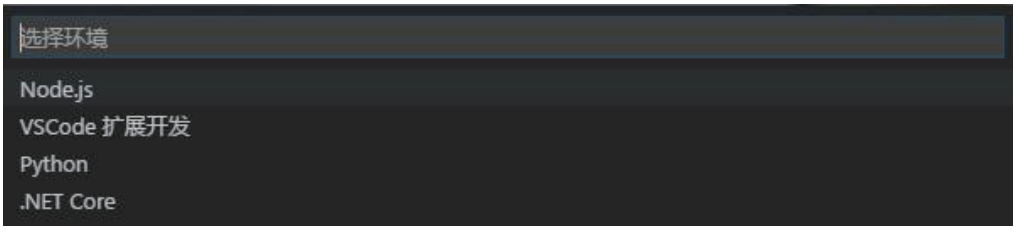


图2-2　打开的菜单

双击 Node.js，这样会在工程根目录下生成 .vscode目录，并在其中创建了一个名为 launch.json 的文件。我们可以在 VS Code 中查看这个 JSON 文件。当鼠标移动到这个文件的键名上时，会出现对键名含义的说明。configurations 是一个配置列表，初始的内容如下：

```
// 代码 2-1
[
        {
```

```
            "name": "启动",
            "type": "Node",
            "request": "launch",
            "program": "${workspaceRoot}/app.js",
            "stopOnEntry": false,
            "args": [],
            "cwd": "${workspaceRoot}",
            "preLaunchTask": null,
            "runtimeExecutable": null,
            "runtimeArgs": [
                "--nolazy"
            ],
            "env": {
                "NODE_ENV": "development"
            },
            "externalConsole": false,
            "sourceMaps": false,
            "outDir": null
        },
        {
            "name": "附加",
            "type": "Node",
            "request": "attach",
            "port": 5858,
            "address": "localhost",
            "restart": false,
```

```
            "sourceMaps": false,
            "outDir": null,
            "localRoot": "${workspaceRoot}",
            "remoteRoot": null
        },
        {
            "name": "Attach to Process",
            "type": "Node",
            "request": "attach",
            "processId": "${command.PickProcess}",
            "port": 5858,
            "sourceMaps": false,
            "outDir": null
        }
    ]
```

每一项的 name 代表一种调试方式的配置名。此名称对应于调试面板左上方的下拉列表。下面以不同的方式调试一个示例程序，然后再来比较这三种方式的异同。先将 program 这个键的值修改为${workspaceRoot}/test/test.js。

这个键指明了入口文件相对于工程目录的位置。修改完之后，按 F5 以调试方式运行。程序在断点处停下来，如图2-3所示。

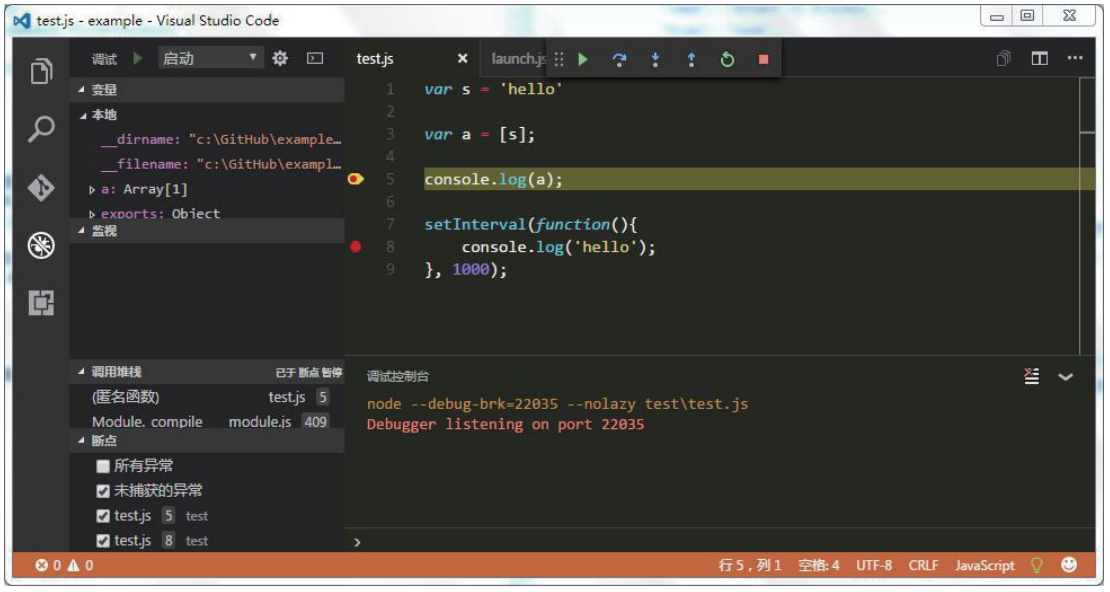


图2-3　程序在断点处停下来

按F10键单步执行，F5键让程序运行，直到遇到下一个断点。在右下方的“调试控制台”窗口，看到如下信息：

```
node --debug-brk=22035 --nolazy test\test.js
```

这个字符串是启动调试服务端的命令行。--debug-brk 表示启动调试服务端后，在客户端连接之前，不执行脚本。此时，如果 stopOnEntry 为 true，则客户端连接成功之后，仍然不执行脚本，否则开始运行，直到遇到断点。22035 代表客户端连接的端口号，随机分配。Node 程序的调试过程符合客户端—服务器模型，两者使用 TCP 通信，因此也支持远程调试。--nolazy 这个选项指定 V8 采取非延迟编译的方式生成可执行的机器码。

接下来按一次F5键，让程序运行起来，之后程序在第8行处断下来。可以查看断点处的变量值、堆栈信息。程序中调用 console 打印出的信息在 VS Code 的控制台窗口显示。在 VS 系列的开发环境中，退出调试的快捷键是 Shift+F5。

以上演示了以启动方式调试程序的过程。按快捷键 Shift+F5 退出调试后，在调试面板

左上方的下拉列表中，选择“Attach to Process”，此选项对应于 configurations 列表的第3项。我们进入工程目录的 test 文件夹，运行 node test.js。这段代码运行之后，打印出如下的字符串：

```
['hello']
hello
```

之后每隔1秒钟，打印出一个 hello。打开任务管理器，查看这个 Node 进程的信息，如图2-4所示。

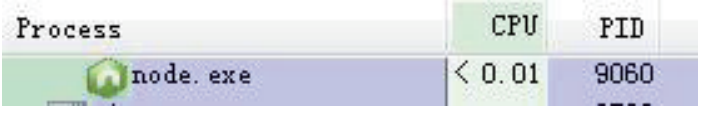


图 2-4　Node进程信息

其 PID 为 9060。我们将列表中第3项的“processed”的值改为 9060，保存文件后，按下F5键。在系统控制台窗口，立即打印出如下信息：

```
Starting debugger agent.
Debugger listening on port 5858
```

程序断下来之后，按一次F5键，以后每次运行到

```
console.log('hello');
```

语句时，程序会在断点处停下来。

launch.json 文件中的 configurations 键包含的三种调试方式各有各的适用场景。以第一种方式调试程序，VS Code 自动启动一个调试服务端，运行我们指定的入口文件。此种方式适合开发过程中，想要跟踪代码的运行，或查看运行时的变量值的情况，以确定程序执行是否符合预期。第二种方式主要用于远程调试，需要远程调试服务端事先运行起来。例如运行

```
node --debug[=port] yourfile.js
```

然后为本地的客户端指定 port 和 address 参数连接远程的调试服务进程。但第二种方式只能在Node 5.0及以上版本中使用，对于低于 5.0 的版本，address 参数应该使用默认值 localhost。如果一定要远程调试，就需要在服务端运行一个 TCP 代理，用于转发客户端的调试指令和返回信息。第三种是附加到进程的方式，这个方式的特点是起初 Node 程序并非以调试方式运行。因此这个方式适合在程序运行一段时间遇到问题时，开发者可以用断点在某些地方让程序暂停，以便查看内部状态，定位问题。退出调试之后，程序还可继续执行。

2.3　单元测试

错误不可避免，但发现得越早，带来的风险和成本就越低。我们应该尽可能早地考虑如何检验模块运行是否良好，是否符合预期。在整个开发周期中，写与测应当交替进行。如果说写是在往前冲，那测就是停下来安静一会儿。Node 中每一个文件皆是模块，可酌情选取，编写对应的测试文件。下面介绍Mocha这个测试框架，它将测试变成了一个有趣的过程。

2.3.1　Mocha 测试框架

Mocha 是一个用于前端和 Node的JavaScript测试框架。它集合了丰富的特性，让异步测试变得简单有趣。Mocha 依次串行执行我们编写的每一个测试用例，在将未捕获异常与相关的用例对应起来的同时，生成灵活准确的测试报告。Mocha 支持 TDD 和 BDD 两种测试风格。

TDD (Test-Driven Development) 即测试驱动开发，测试驱动开发的流程如下。

（1）编写测试代码；

（2）运行测试文件，跑一遍其中的测试用例。此时无一例外会全部失败，因为要测试的对象还没有；

（3）实现被测对象；

（4）重新运行测试文件，修改问题，最后全部通过。

TDD 是一种编程技术，它引导程序员思考自己的代码是如何被其他代码所使用的。首先要写一个测试来描述如何使用下一块代码，然后实现这块代码，并保证它通过测试。TDD 的表述方式类似于说明书，旨在阐明你希望代码如何表现。

BDD (Behavior-Driven Development) 即行为驱动开发。在TDD中，根据设计所编写的测试即便完全通过，也不能保证就是用户想要的功能。BDD 的作用是把利益关系人、交付团队等不同方面的项目相关人员集中到一起，形成共同的理解、共同的价值观以及共同的期望值，它要求开发者在整个项目的层面上思考问题。BDD 关注整体行为是否符合预期，更接近人的思考方式，其表述方式更接近自然语言。

运行

```
npm install mocha -g
```

将 Mocha 安装为全局模块，运行

```
mocha -help
```

显示出帮助内容。

在项目的根目录新建一个 test 文件夹，在其中新建一个 js 文件，内容如下：

```
// 代码 2-2
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal(-1, [1,2,3].indexOf(4));
    });
  });
});
```

回到根目录下，运行 Mocha，控制台显示如下：

```
 Array
   #indexOf()
     √ should return -1 when the value is not present
 1 passing (10ms)
```

上述测试文件第一行引用了 assert 模块，此模块是 Node 的原生模块，实现了断言的功能。断言是程序中的一种语句，其作用是声明预期的结果必须满足。如果运行时不成立，则抛异常。用程序语句表达如下：

```
// 代码 2-3
if(测试条件满足)
  ;
else
  throw new Error("AssertionError");
```

assert 模块定义了一系列的方法，以下的例子列举了这些方法的使用。

```
// 代码 2-4
var assert = require('assert');
var testarr1 = [1,2,'3'];
var testarr2 = [1,2,3];
var testarr3 = [1,2,4]
assert.ok([]);                                  //断言为真
assert.equal(1, '1');                           //断言相等
assert.notEqual(1, 2);                          //断言不相等
assert.strictEqual(1, 1);                       //断言严格相等
assert.notStrictEqual(1, '1');                  //断言不严格相等
assert.deepEqual(testarr1, testarr2);           //断言深度相等
assert.notDeepEqual(testarr1, testarr3);  //断言深度不相等
assert.throws(function(err){
   throw new Error('throw error intentionally')
});                                             //断言代码块抛出异常
assert.doesNotThrow(function(err){ });    //断言代码块不抛异常
assert.ifError(false);  //断言值为假——false, null, undefined, 0, '', NaN
```

除了内建的断言模块，Mocha 测试用例中还可以使用第三方的断言库，例如 should.js。关于此模块的使用，感兴趣的读者可参考 https://github.com/shouldjs/should.js。

2.3.2 TDD 风格

在TDD 风格下，可用的接口包括 suite()、test()、suiteSetup()、suiteTeardown()、setup()和 teardown()函数，用例写法如下：

```
// 代码 2-5
var assert = require('assert');
suite('Array', function() {
  setup(function(done) {
  setTimeout(function(){
    done();
  }, 1000)
  });
  suite('#indexOf()', function() {
    test('should return -1 when not present', function() {
      assert.equal(-1, [1,2,3].indexOf(4));
    });
  });
});
```

TDD 使用 suite 和 test 组织测试用例，suite 可以多级嵌套。setup 在进入 suite 之后触发执行，执行完全部用例后，teardown 被触发执行。使用 Mocha 测试异步代码，只需要在用例函数里面加一个参数 done，异步过程执行完毕之后，调用 done()，通知 Mocha 继续执行下一个测试用例。读者需要知道，异步过程的超时时间默认为2秒，超过2秒就代表执行失败了。也可以修改这个时间，例如：

```
// 代码 2-6
setup(function(done) {
  this.timeout(3000)
  setTimeout(function(){
    done();
  }, 2100)
```

```
  });
```

Mocha 默认识别 BDD 风格的测试代码，因此需要添加 --ui tdd 参数运行上面的例子。

2.3.3 BDD 风格

BDD 的接口有 describe()、it()。同时支持4个钩子函数 before()、after()、beforeEach()和 afterEach()。

describe()和 it()还有别名，分别是 context()和 specify()。before()和after()分别在进入和退出describe()的时候被触发。beforeEach()和afterEach()在执行每一个测试用例的前后被触发。

下面是 BDD 风格的例子。

```
// 代码 2-7
var assert = require('assert');
var fs = require('fs');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal([1,2,3].indexOf(5), -1);
      assert.notEqual([1,2,3].indexOf(1), -1);
    });
  });
});
describe('fs', function(){
  describe('#readdir()', function(){
    it('should not return error', function(done){
```

```
      fs.readdir(__dirname, function(err){
        assert.ifError(err);
        done();
      })
    })
  })
})
```

可见，仅从用法上，BDD与TDD非常一致。

Mocha 会自动读取 test 目录中文件名为 mocha.opts 的内容，这个文件的每一行代表一个配置，例如下例：

```
--require should
--reporter dot
--ui tdd
```

上面的配置就会让 Mocha 引入 should 模块，报告样式设置为 dot，并且使用 tdd 的测试接口。运行 mocha 命令的时候，指定的配置参数与这个配置文件中的配置若有冲突，以命令中的为准。

2.3.4　生成不同形式的测试报告

--reporter 设置项指定了测试报告的样式，默认为 spec ，代表一个嵌套的分级视图。Mocha 提供了一些其他有趣或有用的视图。例如，运行如下命令：

```
mocha --reporter landing --ui tdd test.js
```

看到的是一个起跑的飞机。

```
  ------------------------------------------------------
  ·····················------------------------------·····✈
  ------------------------------------------------------
  1 passing (1s)
```

还可以把测试报告导入到文件，例如以 markdown 格式保存。

```
mocha --reporter markdown --ui tdd test.js > test.md
```

2.3.5 代码覆盖率工具Istanbul

Istanbul（伊斯坦布尔）是土耳其名城，土耳其盛产地毯，地毯用于覆盖，这个工具名称由来于此。其实地毯（blanket）是另外一个类似工具的名字。Istanbul 由雅虎出品，它容易使用，能够生成漂亮的HTML报告，结果直观，查看方便。值得一提的是，它也支持浏览器端的测试。

root权限下，运行如下指令进行安装：

```
npm install -g istanbul
```

安装成功后，运行 istanbul help 可查看它的帮助信息。下面举例说明它的使用方法。首先在根目录的 lib 文件夹内，新建一个 sqrt.js，输入以下代码：

```
// 代码 2-8
module.exports.sqrt = function(x) {
  if(x >= 0)
    return Math.sqrt(x);
  else throw new Error('x < 0');
}
```

```
function check(x) {
    return x < 0 ? false : true;
}
```

然后进入根目录下的 test 目录，新建一个 sqrt_test.js 文件，测试用例如下：

```
// 代码 2-9
'use strict';
let assert = require('assert');
let sqrt = require('./../lib/sqrt').sqrt;
describe('#sqrt()', function (){
    it('sqrt(4) should equal 2', function () {
        assert.equal(sqrt(4), 2);
    });
    it('#sqrt(-1) should throw an Error', function () {
        assert.throws(function(){
            sqrt(-1);
        });
    });
});
```

最后，在工程根目录下，再新建一个名为 .istanbul.yml 的文件，这个文件的内容是 Istanbul 的配置项。例如 excludes: ['sitelists/*'] 表示将忽略 sitelists 目录的内容。

```
// 代码 2-10
verbose: false
instrumentation:
    root: .
```

```
    default-excludes: true
    excludes: []
    include-all-sources: true
reporting:
    print: summary
    reports:
        - lcov
    dir: ./coverage
```

之后运行如下命令：

```
istanbul cover _mocha test/sqrt_test.js
```

运行时如果报错，建立 _mocha 的软链接即可。_mocha 代表在相同进程（即 Istanbul 所在的进程）运行测试用例。Istanbul 会在 coverage/lcov-report 目录下生成HTML格式的报告，如图2-5所示。

图2-5　生成的报告

因为没有调用 check 函数，因此函数的覆盖率是 50% 。此外，它的统计结果还包含行覆盖率、语句覆盖率和分支覆盖率。

Istanbul 提供了 Hook 机制，能够在运行阶段向源文件注入代码。这使得用户能够测试文件内部的非导出函数，例如下面的例子：

```
// 代码 2-11
var assert = require('assert');
var istanbul = require('istanbul');
var hook = istanbul.hook,
     sqrtMatcher = function (file) { return file.match(/sqrt/); },
     sqrtTransformer = function (code, file) { return code +
         '\n module.exports.check = function(x) { return check(x); }'; };
 hook.hookRequire(sqrtMatcher, sqrtTransformer);
var sqrt = require('./../lib/sqrt.js').sqrt;
var check = require('./../lib/sqrt.js').check;
suite('#sqrt()', function (){
    test('sqrt(4) should equal 2', function () {
        assert.equal(sqrt(4), 2);
    });
    test('#sqrt(-1) should throw an Error', function () {
        assert.throws(function(){
            sqrt(-1);
        });
    });
});
//测试非导出函数check
```

```
suite('#check()', function(){
  test('should be false when < 0', function(){
      assert.ifError(check(-1));
  });
});
```

Istanbul模块Hook住了 require 函数。如果引用的模块名称包含 sqrt ，则向读出的源文件加一行字符串，这行字符串的作用是将内部的 check 函数导出。 将上述代码保存为sqrt_test.js文件，执行如下命令：

```
istanbul cover _mocha -- --ui tdd test/sqrt_test.js
```

随后在控制台显示出的信息中，结果如下：

```
#sqrt()
√ sqrt(4) should equal 2
√ #sqrt(-1) should throw an Error
#check()
√ should be false when < 0
3 passing (11ms)
```

参考资料

（1）https://code.visualstudio.com/docs

（2）http://mochajs.org/

（3）http://www.infoq.com/cn/articles/virtual-panel-tdd-bdd

（4）http://www.ibm.com/developerworks/cn/java/j-cq09187/

（5）https://github.com/shouldjs/should.js

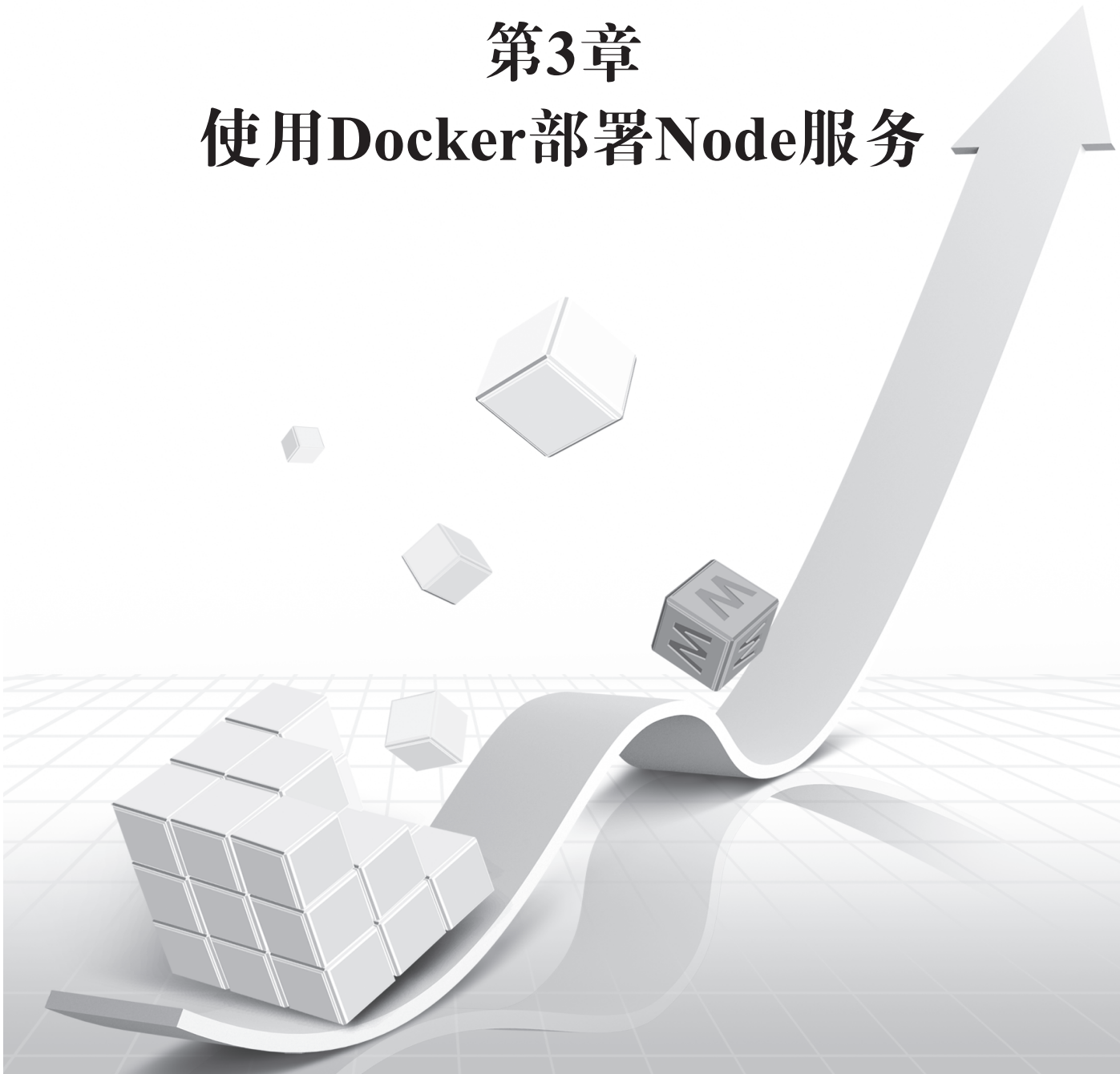
（6）https://github.com/mochajs/mocha/wiki

（7）https://github.com/gotwarlost/istanbul

（8）https://gotwarlost.github.io/istanbul/public/apidocs/index.html

（9）https://github.com/guyellis/http-status-check

第3章
使用Docker部署Node服务



Docker 是一个开源的容器引擎。开发者可以将自己的应用以及依赖打包为一个可移植的容器，然后发布到 Linux 机器上。它类似于一个轻量级的虚拟机，极大地方便了用户在服务端部署和管理应用环境。

3.1 Docker基础

配置 Node 的运行环境，有时候需要编译 Node 的 C++ 模块。Node的编译环境要求GCC/g++ 4.8 或以上的版本。在一些较低版本的 Linux 服务器上，编译安装GCC是一件非常耗时的事情。使用 Docker 可以省去这些麻烦，快速部署应用。可以使用 docker pull 命令下载一个支持Node运行和编译的 Linux 镜像，基于此镜像制作一个包含 Node 程序运行环境的新镜像，以后就可以直接使用这个镜像部署 Node 服务。

如果还没有安装Docker，需要先安装。例如在 Linux 下使用 root 登录后，运行

```
wget -qO- https://get.docker.com/ | sh
```

命令完成安装。安装完毕之后可以运行 docker version 查看版本，结果如下：

```
Client:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.4
 Built:        Wed Jun  1 21:47:50 2016
 OS/Arch:      linux/amd64
Server:
 Version:      1.11.2
```

```
API version:  1.23
Go version:   go1.5.4
Built:        Wed Jun  1 21:47:50 2016
OS/Arch:      linux/amd64
```

Docker 使用客户端/服务器 (C/S) 模型。其守护进程接收客户端的指令，例如运行、提交、发布等。Docker 客户端和守护进程可以运行在同一个系统内，也可以使用 Docker 客户端去连接一个远程的守护进程，此时客户端和守护进程之间通过 socket 或者 RESTful API 进行通信。默认情况下，Docker 守护进程会生成一个 socket（/var/run/docker.sock）文件来与本地的客户端通信，而不会监听任何端口。可以编辑文件 /etc/default/docker，然后重启服务实现远程通信。

相比于虚拟机，Docker 是一种轻量级的虚拟技术。Docker 相对于裸机，其计算能力几乎没有损耗。它直接利用宿主机的系统内核，启动一个 Docker 容器就像启动一个进程一样轻便。Image、container 和 registery 这些概念都可以对应到Git下。下一节将使用一个已有的镜像构建我们的运行环境。

3.2　在Docker中运行Node

笔者基于一个 Ubuntu 14.04的镜像制作了一个包含 Node 程序运行环境的新镜像。安装好 docker 后，运行如下命令：

```
docker pull banz/ubuntu14.04-ansible-nodev4.4.7
```

该命令非常类似于 git clone，会加载这个镜像到本地。下载完毕后运行

```
docker images
```

命令，查看本地包含的所有镜像：

```
root@ubuntu:~# docker images
REPOSITORY              TAG     IMAGE ID      CREATED        SIZE
banz/ubuntu14.04-ansible-nodev4.4.7
                       latest  cc27126cb860  46 hours ago   633.2 MB
```

此镜像是基于 Ubuntu14.04-ansible 制作，增加了 Node 二进制程序和几个常用全局模块。然后可以运行镜像：

```
docker run -t -i -v /data:/root -p 8079:3000 banz/ubuntu14.04-
ansible-nodev4.4.7 /bin/bash
```

该指令创建了一个 container，并且把宿主的 /data 目录挂载到 container 的 /root 下。在容器中，可以使用 ping 命令测试一下是否可以联网。如果 ping 不通，可能是由于系统本地转发支持没有打开。容器中运行 exit 命令退出后，可运行 sysctl net.ipv4.ip_forward 查看，如果是 0，则需要手动打开，可以运行如下命令：

```
sysctl -w net.ipv4.ip_forward=1
```

上面的命令同时将宿主的 8079 端口映射为容器的 3000 端口。只要容器内的服务监听 3000 端口，外部对宿主 8079 端口的请求便能访问到容器内部的服务。回到宿主的命令行下运行如下命令：

```
docker ps -a
```

查看新生成的 container，如下所示：

```
CONTAINER ID IMAGE                    COMMAND      CREATED    NAMES
42cb1c8c730d banz/ubuntu14.04-ansible-nodev4.4.7
```

```
                                    "/bin/bash"  5 min ago  git_gold
```

以后可以使用

```
docker start -ai git_gold
```

来启动这个 container。git_gold 为容器的 name，此值随机生成。

我们重新启动这个容器，运行如下命令：

```
node -v
node-gyp -v
pm2 -v
gulp -v
```

可以看到相应的版本号。在写作本书时，最新的 Node LTS 版本为 4.4.7。以后我们可以把工程源码放到宿主的 /data 目录下，然后在 container 启动后到 /root下使用。在容器中，切换到 /data目录下，将其中的 book 目录复制到～目录下。然后运行 cd ~/book，这个目录下的 node_modules 里包含了以后章节中需要的一些模块。本书之后的一些用例，建议在这个容器中运行。

3.3　导出配置好的容器

上节讲到可以把宿主的 /data 目录挂载到 container 的 /root 下。源码可以使用 Git 仓库托管，在宿主机器上直接拉取到 /data，这样进入容器后就可以直接访问。进入容器，安装好运行环境需要的程序与模块后，可以将容器保存成一个文件，以后需要在其他机器上部署的时候可直接导入。这种方式使得我们对 Node 程序的运行环境实现一次配置、多处部署。

要将一个容器导出，运行如下命令：

```
docker export < CONTAINER-ID > > ~/export.tar
```

该命令将容器ID为 CONTAINER-ID 的容器导出为 export.tar文件，可以将该文件复制到其他机器。如果这些机器也安装了Docker，则可以通过下面的命令：

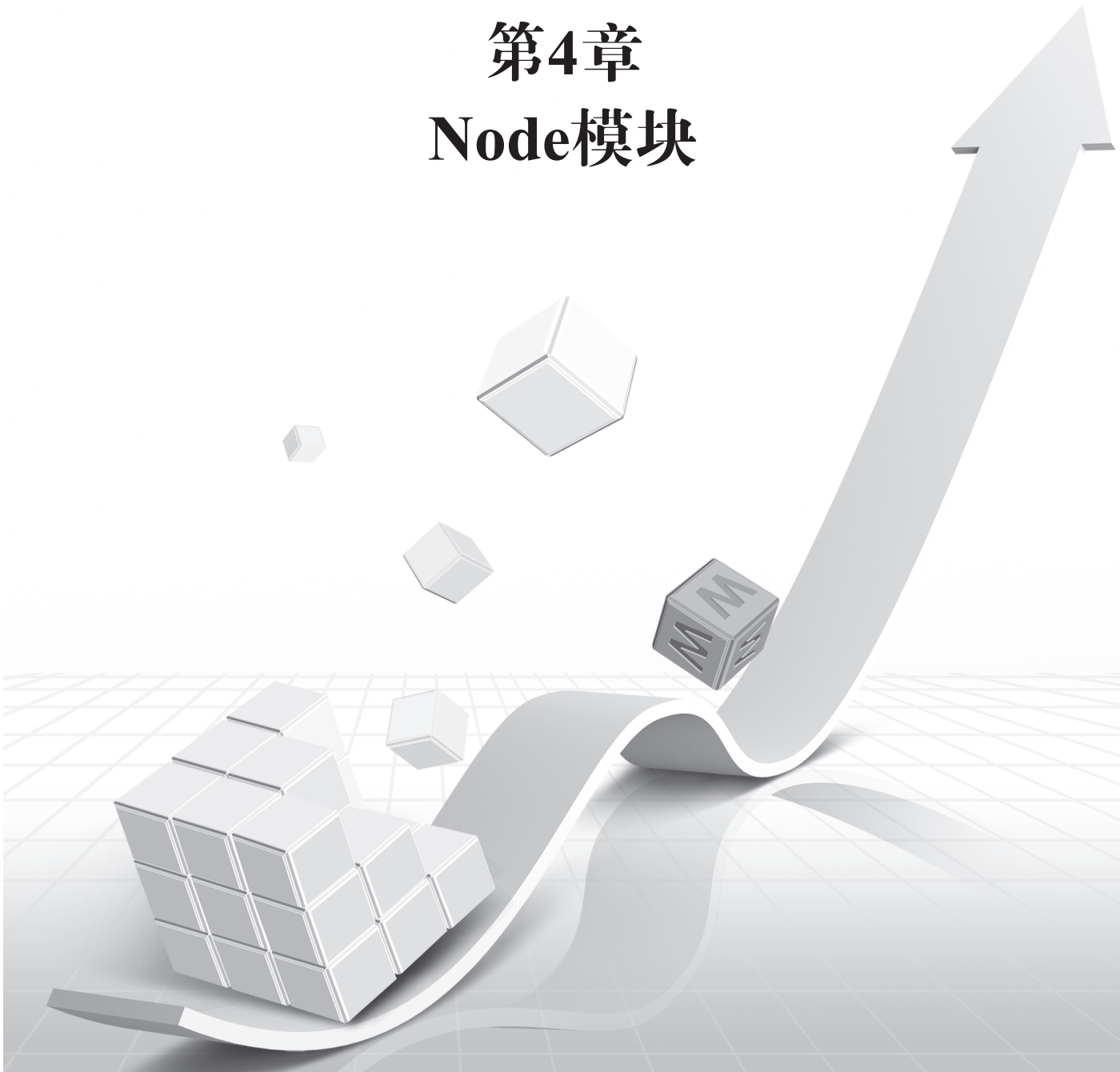
```
cat export.tar | docker import - dev/ubuntu:latest
```

将导出的容器导入为镜像，然后运行这个镜像即可。

参考资料

http://udn.yyuap.com/doc/docker_practice/introduction/index.html

第4章
Node模块



模块是 Node 组织应用程序的单元。本章介绍了 Node 加载模块的机制和导出对象的不同方法以及注意事项。

4.1 程序入口

了解 Node 模块的加载过程，有助于从宏观上把握一个 Node 服务器程序是如何组织起来的。首先编辑一个 hello.js 的文件，内容如下：

```
// 代码 4-1
(function (){
  console.log('hello world!')
})()
```

在命令行下执行 node hello.js。Node 在启动的时候，根据第二个参数加载 JavaScript 文件并运行(node 是第一个参数)。Node没有 C++ 或 Java 那样的 main 函数，但 hello.js如 main 函数一样，是服务端程序运行的总入口。

4.2 VM模块

在本节中，编辑两个 JavaScript 文件。第一个文件内容如下：

```
// 代码 4-2
(function(){
   console.log('hello world~')})
```

将此文件保存为 a.js。第二个文件内容如下：

```
// 代码 4-3
const vm = require('vm');
const fs = require('fs');
const path = require('path');
var prt = path.resolve(__dirname, '.', 'a.js');
function stripBOM(content) {
  if (content.charCodeAt(0) === 0xFEFF) {
    content = content.slice(1);
  }
  return content;
}
var wrapper = stripBOM(fs.readFileSync(prt, 'utf8'));
var compiledWrapper = vm.runInThisContext(wrapper, {
  filename: prt,
  lineOffset: 0,
  displayErrors: true
});
compiledWrapper();
```

将此文件保存为 main.js，然后运行这个文件，得到：

```
hello world~
```

可见使用 VM 模块就能直接运行 JavaScript 文件。在使用 require 加载脚本的时候，Node 内部也是如此读取、编译、运行JavaScript文件。

4.3 模块加载与缓存

用 Node 编写程序，一个 JavaScript 文件对应一个模块，在 module.js 文件中，其构造函数定义如下：

```
// 代码 4-4
function Module(id, parent) {
      this.id = id;
      this.exports = {};
      this.parent = parent;
      if (parent && parent.children) {
            parent.children.push(this);
      }
      this.filename = null;
      this.loaded = false;
      this.children = [];
    }
```

Node 加载一个 JavaScript 文件时，会先构造一个 Module 对象：

```
var module = new Module(filename, parent);
```

然后读入 JavaScript 代码并对文件进行头尾包装。之后，JavaScript 文件里面的代码变成这个匿名函数内部的语句。

```
// 代码 4-5
(function (exports, require, module, _filename, _dirname) {
            //原始文件内容
```

```
    });
```

上述形式的代码实际上是一个函数字面量，也就是定义了一个匿名的函数表达式。Node 使用 V8 编译并运行上面的代码，也就是对以上表达式求值，其值是一个函数对象。V8 将结果返回给 Node。

```
var fn = (function(){}) ; //右侧表达式的值是一个函数对象
```

Node 得到返回的函数对象，使用函数对象的 apply 方法，指定上下文，以

```
module.exports
require
module
_filename
_dirname
```

作为参数，执行函数。在这一步，开始执行 JavaScript 文件内部的代码。

```
var args = [this.exports, require, this, filename, dirname];
var result = compiledWrapper.apply(this.exports, args);
```

由源码可知，JavaScript 文件运行的上下文环境是 module.exports，因此在文件中，也可以直接使用 this 导出对象。一个文件一旦加载之后，其对应的模块被缓存。其他文件在 require 的时候，直接取缓存。

```
// 代码 4-6
var cachedModule = Module._cache[filename];
if (cachedModule) {
    return cachedModule.exports;
}
```