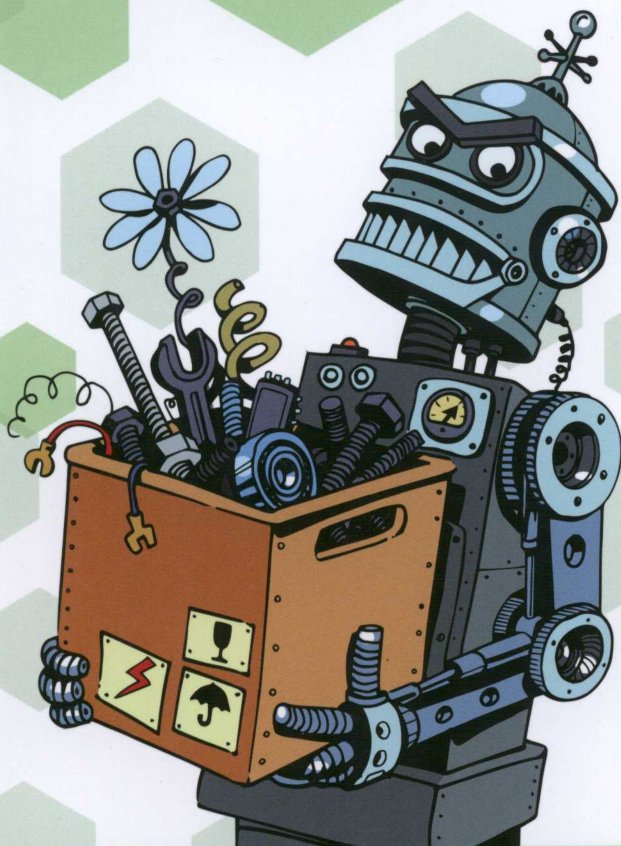
Broadview®
www.broadview.com.cn



Node.js 调试指南

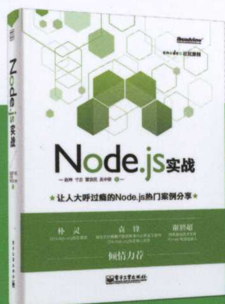
赵坤 编著

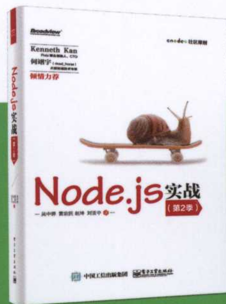
作者简介

/ 赵 坤 /

网名nswbmw，资深Node.js开发者，开源项目Paloma、Mongolass和EverBlog作者，崇尚开源，热爱分享。

曾出版《Node.js实战（双色）》和《Node.js实战（第2季）》。





仅供非商业用途或交流学习使用

Node.js 调试指南

赵坤 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

内容简介

本书从CPU、内存、代码、工具、APM、日志、监控、应用这8个方面讲解如何调试Node.js，大部分小节都会以一段经典的问题代码为例进行分析并给出解决方案。其中，第1章讲解CPU相关的知识，涉及各种CPU性能分析工具及火焰图的使用；第2章讲解内存相关的知识，例如Core Dump及如何分析heapsnapshot文件；第3章讲解代码相关的知识，例如如何从代码层面避免写出难以调试的代码，并涉及部分性能调优知识；第4章讲解工具相关的知识，涉及常用的Node.js调试工具和模块；第5章讲解APM（Application Performance Management）相关的知识，例如两个不同的应用程序性能管理工具的使用；第6章讲解日志相关的知识，例如如何使用Node.js的async_hooks模块实现自动日志打点，并结合各种工具进行使用；第7章讲解监控相关的知识，例如如何使用Telegraf + InfluxDB + Grafana搭建一个完整的Node.js监控系统；第8章讲解应用相关的知识，给出了两个完整的Node.js应用程序的性能解决方案。

本书并不适合Node.js初学者，适合有一定Node.js开发经验的人阅读。笔者倾向于将本书定位成参考书，每一小节基本独立，如果遇到相关问题，则可以随时翻到相应的章节进行阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Node.js调试指南 / 赵坤编著 . —北京：电子工业出版社，2018.6
ISBN 978-7-121-34146-5

Ⅰ. ①N… Ⅱ. ①赵… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字（2018）第088378号

策划编辑：张国霞
责任编辑：徐津平
印　　刷：北京富诚彩色印刷有限公司
装　　订：北京富诚彩色印刷有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：787×980　1/16　印张：15.25　字数：320千字
版　　次：2018年6月第1版
印　　次：2018年6月第1次印刷
印　　数：3000册　定价：89.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。
质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

前言

笔者和同事在过去一年多主要把工作精力放在如何提高 Node.js 服务端的性能、稳定性和基础设施搭建上，随着公司业务量的快速增长，我们遇到了各种各样的挑战，也取得了不错的成绩：从起初啥都没有，到后来建立了比较完善的日志、监控和报警系统；从起初遇到问题不知道如何下手，到后来在遇到问题时能及时发现并定位问题。总之，付出得到了回报。

笔者曾将在这一段时间内遇到的调试、调优过程整理并记录成文章，发表在公司的知乎专栏上，本书就是在其基础上修改、补充和润色而成的，算是笔者对这几年在 Node.js 开发中进行调试的经验和思路的总结，笔者希望授人以鱼，亦能授人以渔。

本书概要

本书从 CPU、内存、代码、工具、APM、日志、监控、应用这 8 个方面讲解如何调试 Node.js，大部分小节都会以一段经典的问题代码为例进行分析并给出解决方案。

第 1 章讲解 CPU 相关的知识，涉及各种 CPU 性能分析工具及火焰图的使用。

第 2 章讲解内存相关的知识，例如 Core Dump 及如何分析 heapsnapshot 文件。

第 3 章讲解代码相关的知识，例如如何避免在代码层面写出难以调试的代码，并涉及部分性能调优知识。

第 4 章讲解工具相关的知识，涉及常用的 Node.js 调试工具和模块。

第 5 章讲解 APM（Application Performance Management）相关的知识，例如两个不同的应用程序性能管理工具的使用。

第 6 章讲解日志相关的知识，例如如何使用 Node.js 的 async_hooks 模块实现自动日

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

志打点，并结合各种工具进行使用。

第 7 章讲解监控相关的知识，例如如何使用 Telegraf + InfluxDB + Grafana 搭建一个完整的 Node.js 监控系统。

第 8 章讲解应用相关的知识，给出了两个完整的 Node.js 应用程序的性能解决方案。

本书定位

本书并不适合 Node.js 初学者，适合有一定 Node.js 开发经验的人阅读。笔者倾向于将本书定位成参考书，每一小节基本独立，如果遇到相关问题，则可以随时翻到相应的章节进行阅读。

开发环境

- MacOS 或 Linux（Ubuntu@16.04 64 位），Windows 用户请在使用虚拟机安装 Ubuntu 后进行操作。
- Node.js@8.9.4。

致谢

感谢石墨文档为笔者提供了良好的成长环境和技术氛围，感谢一起努力并解决问题的同事们，感谢张国霞编辑的耐心指导，感谢寸志、老雷、Yorkie、王政、杨海剑、黄一君在百忙之中抽出时间审阅本书并给出反馈。谢谢你们。

交流 & 勘误

扫描下方的二维码，便可与笔者交流并提交勘误，您的反馈及意见对笔者来说非常重要，再次感谢！

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

目录

第 1 章　CPU　1

1.1　理解 perf 与火焰图（FlameGraph）　2

1.1.1　perf　2

1.1.2　火焰图　6

1.1.3　红蓝差分火焰图　8

1.2　使用 v8-profiler 分析 CPU 的使用情况　11

1.3　Tick Processor 及 Web UI　16

1.3.1　Tick Processor　16

1.3.2　Web UI　21

第 2 章　内存　23

2.1　gcore 与 llnode　24

2.1.1　Core 和 Core Dump　24

2.1.2　gcore　25

2.1.3　llnode　25

2.1.4　测试 Core Dump　26

2.1.5　分析 Core 文件　27

2.1.6　--abort-on-uncaught-exception　29

2.1.7　小结　30

2.2　heapdump　30

2.2.1　使用 heapdump　30

2.2.2　Chrome DevTools　32

2.2.3　对比快照　34

2.3　memwatch-next　35

2.3.1　使用 memwatch-next　35

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

2.3.2 使用 Heap Diff 38
2.3.3 结合 heapdump 使用 40
2.4 cpu-memory-monitor 41
2.4.1 使用 cpu-memory-monitor 41
2.4.2 cpu-memory-monitor 源码解读 43

第 3 章 代码 46

3.1 Promise 47
3.1.1 Promise/A+ 规范 48
3.1.2 从零开始实现 Promise 48
3.1.3 Promise 的实现原理 50
3.1.4 safelyResolveThen 52
3.1.5 doResolve 和 doReject 54
3.1.6 Promise.prototype.then 和 Promise.prototype.catch 55
3.1.7 值穿透 58
3.1.8 Promise.resolve 和 Promise.reject 60
3.1.9 Promise.all 61
3.1.10 Promise.race 62
3.1.11 代码解析 63
3.2 Async + Await 69
3.2.1 例 1：async + await 70
3.2.2 例 2：co + yield 71
3.2.3 例 3：co + yield* 72
3.2.4 例 4：co + bluebird 73
3.2.5 从 yield 转为 yield* 遇到的坑 75
3.2.6 async + bluebird 76
3.3 Error Stack 77
3.3.1 Stack Trace 78
3.3.2 Error.captureStackTrace 80
3.3.3 captureStackTrace 在 Mongolass 中的应用 83
3.3.4 Error.prepareStackTrace 84
3.3.5 Error.prepareStackTrace 的其他用法 86

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

3.3.6 Error.stackTraceLimit 88
3.3.7 Long Stack Trace 88
3.4 node@8 89
3.4.1 Ignition + Turbofan 90
3.4.2 版本的对应关系 91
3.4.3 try/catch 91
3.4.4 delete 93
3.4.5 arguments 95
3.4.6 async 性能提升 97
3.4.7 不会优化的特性 98
3.5 Rust Addons 100
3.5.1 Rust 100
3.5.2 FFI 100
3.5.3 Neon 103
3.5.4 NAPI 108
3.6 Event Loop 110
3.6.1 什么是 Event Loop 110
3.6.2 poll 阶段 112
3.6.3 process.nextTick() 112
3.6.4 代码解析 113
3.7 处理 uncaughtException 120
3.7.1 uncaughtException 120
3.7.2 使用 llnode 121
3.7.3 ReDoS 122

第 4 章 工具 125

4.1 Source Map 126
4.1.1 uglify-es 126
4.1.2 TypeScript 128
4.1.3 source-map-support 的高级用法 129
4.2 Chrome DevTools 129
4.2.1 使用 Chrome DevTools 130

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

4.2.2 NIM 132
4.2.3 inspect-process 133
4.2.4 process._debugProcess 133
4.3 Visual Studio Code 134
4.3.1 基本调试 134
4.3.2 launch.json 136
4.3.3 技巧 1——条件断点 138
4.3.4 技巧 2——skipFiles 139
4.3.5 技巧 3——自动重启 140
4.3.6 技巧 4——对特定操作系统的设置 142
4.3.7 技巧 5——多配置 142
4.3.8 总结 144
4.4 debug + repl2 + power-assert 144
4.4.1 debug 144
4.4.2 repl2 146
4.4.3 power-assert 148
4.5 supervisor-hot-reload 151
4.5.1 Proxy 151
4.5.2 用 Proxy 实现 Hot Reload 153
4.5.3 supervisor-hot-reload 155
4.5.4 内存泄漏问题 160

第 5 章 日志 161

5.1 koa-await-breakpoint 162
5.1.1 koa-await-breakpoint 的实现原理 162
5.1.2 使用 koa-await-breakpoint 165
5.1.3 自定义日志存储 167
5.2 使用 async_hooks 168
5.3 ELK 177
5.3.1 安装 ELK 177
5.3.2 使用 ELK 178

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

5.4 OpenTracing + Jaeger 182
5.4.1 什么是 OpenTracing 182
5.4.2 什么是 Jaeger 184
5.4.3 启动 Jaeger 及 Jaeger UI 184
5.4.4 使用 OpenTracing 及 Jaeger 185
5.4.5 koa-await-breakpoint-jaeger 187
5.5 使用 Sentry 190

第 6 章 APM 197

6.1 使用 NewRelic 198
6.2 Elastic APM 201
6.2.1 什么是 Elastic APM 201
6.2.2 启动 ELK 203
6.2.3 启动 APM Server 203
6.2.4 使用 Elastic APM 203
6.2.5 错误日志 205

第 7 章 监控 207

7.1 Telegraf + InfluxDB + Grafana（上） 208
7.1.1 Telegraf（StatsD）+ InfluxDB + Grafana 简介 208
7.1.2 启动 docker-statsd-influxdb-grafana 208
7.1.3 熟悉 InfluxDB 209
7.1.4 配置 Grafana 210
7.1.5 node-statsd 211
7.1.6 创建 Grafana 图表 213
7.1.7 模拟真实环境 214
7.2 Telegraf + InfluxDB + Grafana（下） 217
7.2.1 Grafana + ELK 217
7.2.2 监控报警 220
7.2.3 脚本一键生成图表 222

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

第 8 章　应用　224

8.1　使用 node-clinic　225
8.2　alinode　227
　　8.2.1　什么是 alinode　227
　　8.2.2　创建 alinode 应用　228
　　8.2.3　安装 alinode　228
　　8.2.4　使用 alinode 诊断内存泄露　229
　　8.2.5　使用 alinode 诊断 CPU 性能瓶颈　232

书籍是人类进步的阶梯

书籍是人类进步的阶梯

第1章

CPU

Node.js 是单线程异步 I/O 模型，适合 I/O 密集型的场景，这也意味着 Node.js 并不适合 CPU 密集型计算的场景，因为 Node.js 不是收到一个请求就启动一个线程，假如有一个计算任务长时间地占用 CPU，整个应用就会“卡住”，无法处理其他请求。本章将讲解在 CPU 出现性能瓶颈时如何调试、发现及解决问题，并讲解相关的基础知识。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

1.1 理解 perf 与火焰图（FlameGraph）

当程序出现性能瓶颈时，我们通常通过表象（比如在请求某个接口时 CPU 使用率飙涨）并结合代码去推测可能出问题的地方，却不知道问题是由什么引起的。如果有一个可视化的工具能直观地展现程序的性能瓶颈就好了，幸好 Brendan D. Gregg 发明了火焰图。

火焰图（Flame Graph）看起来就像一团跳动的火焰，因此得名，它可以将 CPU 的使用情况可视化，使我们直观地了解到程序的性能瓶颈。我们通常要结合操作系统的性能分析工具（Profiling Tracer）使用火焰图，常见的操作系统的性能分析工具如下。

- Linux：perf、eBPF、SystemTap 和 ktap。
- Solaris：illumos、FreeBSD 和 DTrace。
- Mac OS X：DTrace 和 Instruments。
- Windows：Xperf.exe。

1.1.1 perf

perf_events（简称 perf）是 Linux Kernal 自带的系统性能分析工具，能够进行函数级与指令级的热点查找。它基于事件采样原理，以性能事件为基础，支持针对处理器与操作系统相关的性能指标的性能剖析，常用于查找性能瓶颈及定位热点代码。

测试机器：

```
$ uname -a
Linux nswbmw-VirtualBox 4.10.0-28-generic #32~16.04.2-Ubuntu SMP Thu Jul 20
10:19:48 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
```

> **注　意**
>
> 非 Linux 用户需要在通过虚拟机安装 Ubuntu 16.04 和 node@8.9.4 后再进行后面的操作。

安装 perf：

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
$ sudo apt install linux-tools-common
$ perf # 根据提示安装对应的内核版本的 tools，如下
$ sudo apt install linux-tools-4.10.0-28-generic linux-cloud-tools-4.10.0-28-
generic
```

创建测试目录 ~/test 和测试代码。

app.js：

```
const crypto = require('crypto')
const Paloma = require('paloma')
const app = new Paloma()
const users = {}

app.route({ method: 'GET', path: '/newUser', controller (ctx) {
  const username = ctx.query.username || 'test'
  const password = ctx.query.password || 'test'

  const salt = crypto.randomBytes(128).toString('base64')
  const hash = crypto.pbkdf2Sync(password, salt, 10000, 64, 'sha512').
toString('hex')

  users[username] = { salt, hash }

  ctx.status = 204
}})

app.route({ method: 'GET', path: '/auth', controller (ctx) {
  const username = ctx.query.username || 'test'
  const password = ctx.query.password || 'test'

  if (!users[username]) {
    ctx.throw(400)
  }
  const hash = crypto.pbkdf2Sync(password, users[username].salt, 10000, 64,
'sha512').toString('hex')

  if (users[username].hash === hash) {
    ctx.status = 204
  } else {
    ctx.throw(403)
  }
}})
```

书籍是人类进步的阶梯

```
app.listen(3000)
```

添加 --perf_basic_prof（或者 --perf-basic-prof）参数运行此程序，会对应生成一个 /tmp/perf-<PID>.map 文件。命令如下：

```
$ node --perf_basic_prof app.js &
[1] 3590
$ tail /tmp/perf-3590.map
51b87a7b93e 18 Function:~emitListeningNT net.js:1375
51b87a7b93e 18 LazyCompile:~emitListeningNT net.js:1375
51b87a7bad6 39 Function:~emitAfterScript async_hooks.js:443
51b87a7bad6 39 LazyCompile:~emitAfterScript async_hooks.js:443
51b87a7bcbe 77 Function:~tickDone internal/process/next_tick.js:88
51b87a7bcbe 77 LazyCompile:~tickDone internal/process/next_tick.js:88
51b87a7bf36 12 Function:~clear internal/process/next_tick.js:42
51b87a7bf36 12 LazyCompile:~clear internal/process/next_tick.js:42
51b87a7c126 b8 Function:~emitPendingUnhandledRejections internal/process/
promises.js:86
51b87a7c126 b8 LazyCompile:~emitPendingUnhandledRejections internal/process/
promises.js:86
```

map 文件中的三列依次为：16 进制的符号地址（Symbol Address）、大小（Size）和符号名（Symbol Names）。perf 会尝试查找 /tmp/perf-<PID>.map 文件，用来做符号转换，即把 16 进制的符号地址转换成人能读懂的符号名。

当然，在这里使用 --perf_basic_prof_only_functions 参数也可以，但笔者在尝试后发现生成的火焰图信息不全（不全的地方显示 [perf-<PID>.map]），所以使用了 --perf_basic_prof。但是，使用 --perf_basic_prof 有个缺点，就是会导致 map 文件一直增大，这是由于符号（symbols）地址的不断变换导致的，用 --perf_basic_prof_only_functions 可以缓解这个问题。关于如何取舍，还请读者自行尝试。

接下来 clone（克隆）用来生成火焰图的工具：

```
$ git clone http://github.com/brendangregg/FlameGraph ~/FlameGraph
```

我们先用 ab 压测：

```
$ curl "http://localhost:3000/newUser?username=admin&password=123456"
$ ab -k -c 10 -n 2000 "http://localhost:3000/auth?username=admin&passwo
```

书籍是人类进步的阶梯

```
rd=123456"
```

重新打开另一个终端，在 ab 开始压测后立即运行：

```
$ sudo perf record -F 99 -p 3590 -g -- sleep 30
$ sudo chown root /tmp/perf-3590.map
$ sudo perf script > perf.stacks
$ ~/FlameGraph/stackcollapse-perf.pl --kernel < ~/perf.stacks | ~/FlameGraph/
flamegraph.pl --color=js --hash> ~/flamegraph.svg
```

注 意

第 1 次生成的 svg 可能不太准确，最好重复几次以上步骤，使用第 2 次及以后生成的 flamegraph.svg。

有几点需要解释一下。

（1）关于 perf record：

- -F 指定采样频率为 99Hz（即每秒 99 次，如果 99 次都返回同一个函数名，就说明 CPU 在这一秒钟都在执行同一个函数，可能存在性能问题）；
- -p 指定进程的 pid；
- -g 启用 call-graph 记录；
- sleep 30 指定记录 30s。

（2）sudo chown root /tmp/perf-3009.map 用于将 map 文件更改为 root 权限，否则会报如下错误：

```
 > File /tmp/perf-PID.map not owned by current user or root, ignoring it
(use -f to override). > Failed to open /tmp/perf-PID.map, continuing without
symbols
```

（3）perf record 会将记录的信息保存到当前执行目录的 perf.data 文件中，将使用 perf script 读取的 perf.data 的 trace 信息写入 perf.stacks。

（4）--color=js 指定生成针对 JavaScript 配色的 svg，其中，green 代表 JavaScript，blue 代表 Builtin，yellow 代表 C++，red 代表 System。

书籍是人类进步的阶梯

ab 压测用了 30s 左右，用浏览器打开 flamegraph.svg，截取关键的部分，如图 1-1 所示。

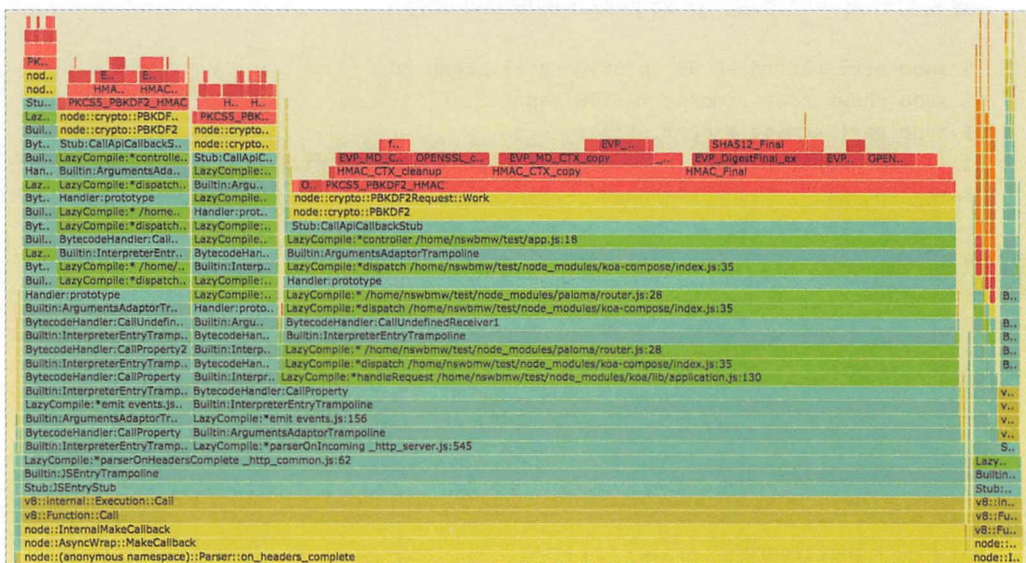


图 1-1

1.1.2 火焰图

火焰图的含义如下。

- 每一个小块都代表一个函数在栈中的位置（即一个栈帧）。
- Y 轴代表栈的深度（栈上的帧数），顶端的小块显示了占据 CPU 的函数。每个小块的下面是它的祖先（即父函数）。
- X 轴代表总的样例群体。它不像绝大多数图表那样从左到右表示时间的流逝，其左右顺序没有特殊含义，仅仅按照字母表的顺序排列。
- 小块的宽度代表 CPU 的使用时间，或者说相对于父函数而言使用 CPU 的比例（基于所有样例），越宽则代表占用 CPU 的时间越长，或者使用 CPU 很频繁。
- 如果采取多线程并发运行取样，则取样数量会超过运行时间。

书籍是人类进步的阶梯

从图 1-1 可以看出，最上面的绿色小块（即 JavaScript 代码）指向 test/app.js 的第 18 行，即 GET /auth 这个路由；再往上看，黄色的小块（即 C++ 代码）node::crypto::PBKDF2 占用了大量的 CPU 时间。

解决方法：将同步改为异步，即将 crypto.pbkdf2Sync 改为 crypto.pbkdf2。修改如下：

```
app.route({ method: 'GET', path: '/auth', async controller (ctx) {
  const username = ctx.query.username || 'test'
  const password = ctx.query.password || 'test'

  if (!users[username]) {
    ctx.throw(400)
  }
  const hash = await new Promise((resolve, reject) => {
    crypto.pbkdf2(password, users[username].salt, 10000, 64, 'sha512', (err,
derivedKey) => {
      if (err) {
        return reject(err)
      }
      resolve(derivedKey.toString('hex'))
    })
  })

  if (users[username].hash === hash) {
    ctx.status = 204
  } else {
    ctx.throw(403)
  }
}})
```

用 ab 重新压测，结果用了 16s。重新生成的火焰图如图 1-2 所示。

书籍是人类进步的阶梯

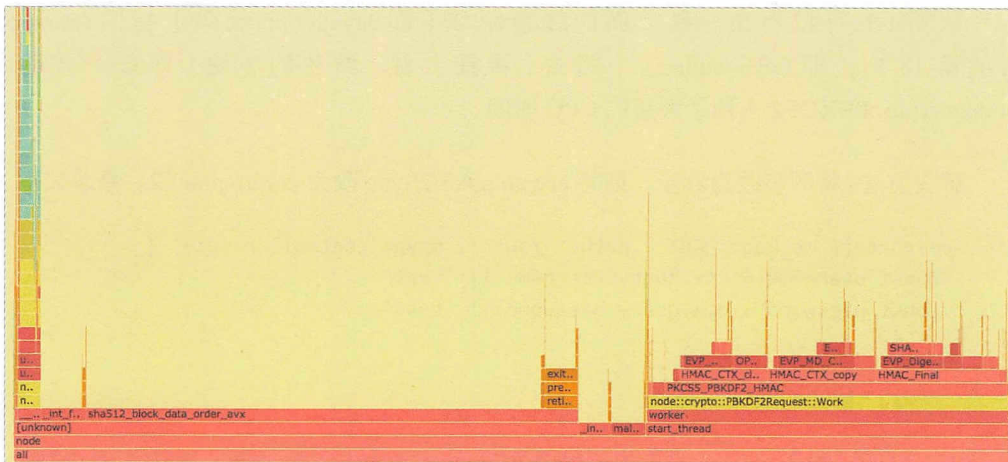


图 1-2

可以看出，只有在左侧极窄的绿色小块中可以看到 JavaScript 代码，我们不关心也无法优化红色的部分。那么，为什么异步比同步的 QPS 要高呢？原因是 Node.js 底层的 libuv 用了多个线程进行计算，这里就不再深入介绍了。

svg 火焰图的其他小技巧如下。

- 单击任意一个小块即可展开，即被单击的小块宽度变宽，它的子函数也按比例变宽，方便查看。
- 可单击 svg 右上角的 search 按钮进行搜索，被搜索的关键词会高亮显示，在有目的地查找某个函数时比较有用。

1.1.3 红蓝差分火焰图

虽然我们有了火焰图，但要处理性能回退问题，还需要在修改代码前后的火焰图之间不断切换和对比，来找出问题所在，很不方便。于是 Brendan D. Gregg 又发明了红蓝差分火焰图（Red/Blue Differential Flame Graphs），如图 1-3 所示，其中，红色表示增长，蓝色表示衰减。

书籍是人类进步的阶梯

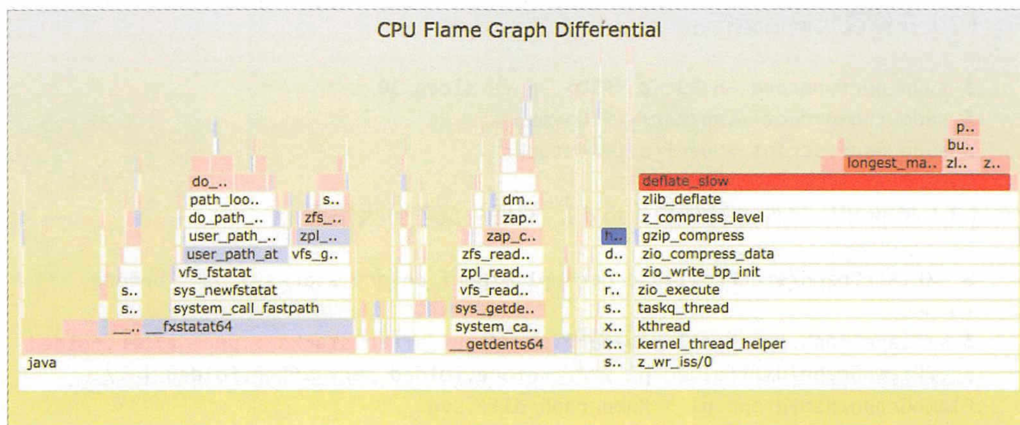


图 1-3

红蓝差分火焰图的工作原理如下。

（1）抓取修改前的栈 profile1 文件。

（2）抓取修改后的栈 profile2 文件。

（3）使用 profile2 来生成火焰图，这样栈帧的宽度就是以 profile2 文件为基准的。

（4）将 profile2 与 profile1 进行比较，并根据其差异来生成新的火焰图，并对新的火焰图重新上色。上色的原则是：如果栈帧在 profile2 中出现的次数更多，则将该栈帧标为红色，否则将其标为蓝色。色彩是根据修改前后的差异来填充的。

这样，通过红蓝差分火焰图，我们就可以清楚地看到当前系统的性能差异。

生成红蓝差分火焰图的流程如下。

（1）在修改代码前运行：

```
$ sudo perf record -F 99 -p <PID> -g -- sleep 30
$ sudo chown root /tmp/perf-<PID>.map
$ sudo perf script > perf_before.stacks
```

书籍是人类进步的阶梯

（2）在修改代码后运行：

```
$ sudo perf record -F 99 -p <PID> -g -- sleep 30
$ sudo chown root /tmp/perf-<PID>.map
$ sudo perf script > perf_after.stacks
```

（3）将 profile 文件进行折叠（fold），然后生成差分火焰图：

```
$ ~/FlameGraph/stackcollapse-perf.pl ~/perf_before.stacks > perf_before.
folded
$ ~/FlameGraph/stackcollapse-perf.pl ~/perf_after.stacks > perf_after.folded
$ ./FlameGraph/difffolded.pl perf_before.folded perf_after.folded | ./
FlameGraph/flamegraph.pl > flamegraph_diff.svg
```

其中的缺点是：如果一个代码执行路径完全消失了，在火焰图中就找不到地方来标注蓝色，我们只能看到当前的 CPU 使用情况，却不知道为什么会变成这样。

一种解决办法是：生成一个相反的差分火焰图，即基于 profile1 生成 profile1 与 profile2 的差分火焰图。对应的命令如下：

```
$ ./FlameGraph/difffolded.pl perf_after.folded perf_before.folded | ./
FlameGraph/flamegraph.pl --negate > flamegraph_diff2.svg
```

其中，--negate 用于颠倒红、蓝配色。我们最终得到 flamegraph_diff.svg 和 flamegraph_diff2.svg，如下所述。

- flamegraph_diff.svg 的宽度以修改前的 profile 文件为基准，其颜色表明将要发生的情况。
- flamegraph_diff2.svg 的宽度以修改后的 profile 文件为基准，其颜色表明已经发生的情况。

总之，红蓝差分火焰图可能只在代码变化不大的情况下使用时效果明显，在代码变化较大的情况下使用时效果可能就不明显了。

本节的参考链接如下。

- https://yunong.io/2015/11/23/generating-node-js-flame-graphs/
- http://www.brendangregg.com/perf.html
- http://www.brendangregg.com/blog/2014-09-17/node-flame-graphs-on-linux.html

书籍是人类进步的阶梯

- https://linux.cn/article-4670-1.html
- http://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html
- http://www.ruanyifeng.com/blog/2017/09/flame-graph.html

1.2 使用 v8-profiler 分析 CPU 的使用情况

我们知道 Node.js 是基于 V8 引擎的，V8 暴露了一些 profiler API，我们可以通过 v8-profiler 收集一些运行时数据（例如 CPU 和内存）。本节将介绍如何使用 v8-profiler 分析 CPU 的使用情况。

首先，创建测试代码。

app.js：

```
const fs = require('fs')
const crypto = require('crypto')
const Bluebird = require('bluebird')
const profiler = require('v8-profiler')
const Paloma = require('paloma')
const app = new Paloma()

app.route({ method: 'GET', path: '/encrypt', controller: function
encryptRouter (ctx) {
  const password = ctx.query.password || 'test'
  const salt = crypto.randomBytes(128).toString('base64')
  const encryptedPassword = crypto.pbkdf2Sync(password, salt, 10000, 64,
'sha512').toString('hex')

  ctx.body = encryptedPassword
}})

app.route({ method: 'GET', path: '/cpuprofile', async controller (ctx) {
   //Start Profiling
   profiler.startProfiling('CPU profile')
   await Bluebird.delay(30000)
   //Stop Profiling after 30s
```

书籍是人类进步的阶梯

```
    const profile = profiler.stopProfiling()
    profile.export()
      .pipe(fs.createWriteStream(`cpuprofile-${Date.now()}.cpuprofile`))
      .on('finish', () => profile.delete())
    ctx.status = 204
}})

app.listen(3000)
```

GET /encrypt 有一个 CPU 密集型的计算函数 crypto.pbkdf2Sync，GET /cpuprofile 用来收集 30s 的 V8 log，然后将其 dump（转储）到一个文件中。

运行该程序，打开两个终端窗口。在一个终端运行如下命令来触发 CPU profiling：

```
$ curl localhost:3000/cpuprofile
```

然后在另一个终端立即运行如下命令来触发 CPU 密集计算：

```
$ ab -c 20 -n 2000 "http://localhost:3000/encrypt?password=123456"
```

最后生成 cpuprofile-xxx.cpuprofile 文件，该文件的内容其实就是一个大的 JSON 对象，大体如下：

```
{
  "typeId": "CPU",
  "uid": "1",
  "title": "CPU profile",
  "head":
   { "functionName": "(root)",
     "url": "",
     "lineNumber": 0,
     "callUID": 154,
     "bailoutReason": "",
     "id": 1,
     "scriptId": 0,
     "hitCount": 0,
     "children": [ ... ] },
  "startTime": 276245,
  "endTime": 276306,
  "samples": [ ... ],
  "timestamps": [ ... ]
}
```

书籍是人类进步的阶梯

这个 JSON 对象记录了函数调用栈、路径、时间戳和其他信息，samples 节点数组与 timestamps 节点数组中的时间戳是一一对应的，并且 samples 节点数组中的每一个值其实对应了 head 节点的深度优先遍历 ID。这里我们就不深究每个字段的含义了，先来看看如何可视化这些数据。

方法 1：使用 Chrome DevTools

Chrome 自带了分析 CPU profile 日志的工具。打开 Chrome，调出开发者工具（DevTools），单击右上角三个点的按钮，依次单击 More tools → JavaScript Profiler → Load，加载刚才生成的 cpuprofile 文件。在左上角的下拉菜单中可以选择如下三种模式。

- Chart：显示按时间顺序排列的火焰图。
- Heavy（Bottom Up）：按照函数对性能的影响排列，同时可以检查函数的调用路径。
- Tree（Top Down）：显示调用结构的总体状况，从调用堆栈的顶端开始。

这里我们选择 Tree（Top Down）模式，按 Total Time 降序排列。可以看到有如下三列。

- Self Time：指函数调用所耗费的时间，仅包含函数本身的声明，不包含任何子函数的执行时间。
- Total Time：指函数调用所耗费的总时间，包含函数本身的声明及所有子函数的执行时间。即父函数的 Total Time 为父函数的 Self Time 与所有子函数的 Total Time 之和。
- Function：函数名及路径，可展开查看子函数。

我们不断地展开，并定位到了 encryptRouter，如图 1-4 所示。

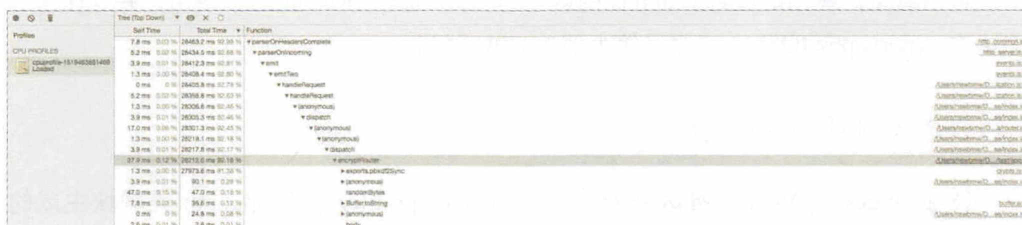


图 1-4

书籍是人类进步的阶梯

可以看出，我们定位到了 encryptRouter 这个路由，并且 exports.pbkdf2Sync 在这个路由中占据了绝大部分 CPU 时间。

方法 2：使用火焰图

我们也可以用火焰图来展示 cpuprofile 数据。

首先，全局安装 flamegraph 模块：

```
$ npm i flamegraph -g
```

运行以下命令，将 cpuprofile 文件生成 svg 文件：

```
$ flamegraph -t cpuprofile -f cpuprofile-xxx.cpuprofile -o cpuprofile.svg
```

用浏览器打开 cpuprofile.svg，如图 1-5 所示。

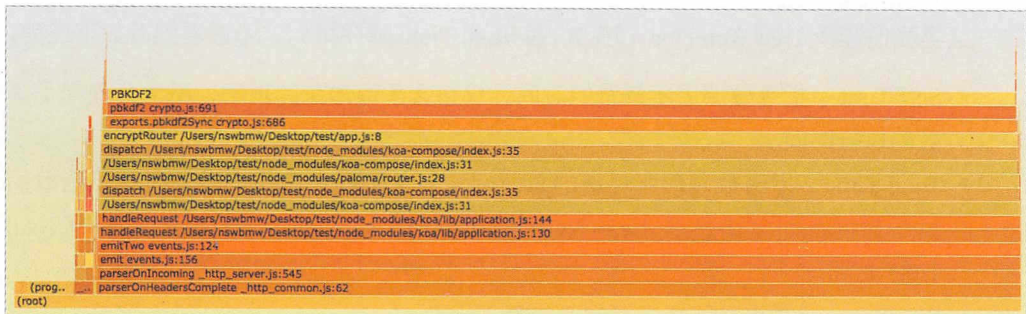


图 1-5

可以看出，我们定位到了 app.js 的第 8 行，即 encryptRouter 这个路由，并且在这个路由中 exports.pbkdf2Sync 占据了绝大部分 CPU 时间。

方法 3：使用 v8-analytics

v8-analytics 是 Node.js 社区开源的一个解析 v8-profiler 和 heapdump 等模块生成的 CPU 和 heap-memory 日志的工具。它提供了以下功能。

（1）将 V8 引擎逆优化或者优化失败的函数标红展示，并展示优化失败的原因。

书籍是人类进步的阶梯

（2）在函数执行时长超过预期时标红展示。

（3）展示当前项目中可疑的内存泄漏点。

我们以上述第 2 个功能为例，使用 v8-analytics 分析 CPU 的使用情况。

首先，全局安装 v8-analytics：

```
$ npm i v8-analytics -g
```

使用以下命令查看执行时间大于 200ms 的函数：

```
$ va timeout cpuprofile-xxx.cpuprofile 200 --only
```

结果如图 1-6 所示。

```
Function Execute Time > 200ms List:
1. (idle) (312.0ms 1.03%)
2. parserOnHeadersComplete (207.1ms 0.68%) (_http_common.js 62)
3. parserOnHeadersComplete (279.0ms 0.92%) (_http_common.js 62)
4. parserOnIncoming (233.7ms 83.76%) (_http_server.js 545)
5. emit (233.7ms 100.00%) (events.js 156)
6. emitTwo (233.7ms 100.00%) (events.js 124)
7. handleRequest (233.7ms 100.00%) (/Users/nswbmw/Desktop/test/node_modules/koa/lib/application.js 130)
8. handleRequest (233.7ms 100.00%) (/Users/nswbmw/Desktop/test/node_modules/koa/lib/application.js 144)
9. anonymous (210.6ms 90.13%) (/Users/nswbmw/Desktop/test/node_modules/koa-compose/index.js 31)
10. dispatch (210.6ms 100.00%) (/Users/nswbmw/Desktop/test/node_modules/koa-compose/index.js 35)
11. anonymous (209.5ms 99.44%) (/Users/nswbmw/Desktop/test/node_modules/paloma/router.js 28)
12. anonymous (209.5ms 100.00%) (/Users/nswbmw/Desktop/test/node_modules/koa-compose/index.js 31)
13. dispatch (209.5ms 100.00%) (/Users/nswbmw/Desktop/test/node_modules/koa-compose/index.js 35)
14. encryptRouter (209.5ms 100.00%) (/Users/nswbmw/Desktop/test/app.js 8)
15. exports.pbkdf2Sync (209.5ms 100.00%) (crypto.js 686)
16. pbkdf2 (209.5ms 100.00%) (crypto.js 691)
17. PBKDF2 (209.5ms 100.00%)
```

图 1-6

可以看出，我们依然能够定位到 encryptRouter 和 exports.pbkdf2Sync。

本节的参考链接如下。

- https://developers.google.com/web/tools/chrome-devtools/rendering-tools/js-execution
- http://www.ebaytechblog.com/2016/06/15/igniting-node-js-flames/
- https://cnodejs.org/topic/58b562f97872ea0864fee1a7
- https://github.com/hyj1991/v8-analytics/blob/master/README_ZH.md

书籍是人类进步的阶梯

1.3 Tick Processor 及 Web UI

1.3.1 Tick Processor

V8 内置了一个性能分析工具——Tick Processor，可以记录 JavaScript、C、C++ 代码的堆栈信息，该功能默认是关闭的，可以通过添加命令行参数 --prof 开启。

首先，创建测试代码。

app.js：

```
const crypto = require('crypto')

function hash (password) {
  const salt = crypto.randomBytes(128).toString('base64')
  const hash = crypto.pbkdf2Sync(password, salt, 10000, 64, 'sha512')
  return hash
}

console.time('pbkdf2Sync')
for (let i = 0; i < 100; i++) {
  hash('random_password')
}
console.timeEnd('pbkdf2Sync')
```

运行：

```
$ node --prof app
pbkdf2Sync: 1375.582ms
```

可以看出，执行 100 次 hash 函数总共用了 1375.585ms，并且在当前目录下多了一个 isolate-xxx-v8.log 文件。该文件记录了 V8 的性能日志，内容如下：

```
v8-version,6,1,534,50,0
shared-library,"/usr/local/bin/node",0x100001800,0x100bbb69a,0
...
code-creation,Function,18,111912,0x37d07c7246a8,144,"hash /Users/nswbmw/
Desktop/test/app.js:3:15",0x37d07c7076d0,~
code-creation,LazyCompile,18,111927,0x37d07c7246a8,144,"hash /Users/nswbmw/
```

书籍是人类进步的阶梯

```
Desktop/test/app.js:3:15",0x37d07c7076d0,~
code-creation,Function,18,112058,0x37d07c725690,80,"exports.pbkdf2Sync
crypto.js:686:30",0x37d07c70cb58,~
code-creation,LazyCompile,18,112074,0x37d07c725690,80,"exports.pbkdf2Sync
crypto.js:686:30",0x37d07c70cb58,~
...
```

我们在早期需要借助 node-tick-processor 这样的工具解析 v8.log，但 Node.js 在 v5.2.0 之后包含了 v8.log 处理器，可添加命令行参数 --prof-process 进行开启。

运行：

```
$ node --prof-process isolate-0x103000000-v8.log
```

结果如下：

```
Statistical profiling result from isolate-0x103000000-v8.log, (1152 ticks, 44
unaccounted, 0 excluded).

 [Shared libraries]:
   ticks  total  nonlib   name

 [JavaScript]:
   ticks  total  nonlib   name
      1    0.1%    0.1%  Function: ~Uint8Array native typedarray.js:158:31
      1    0.1%    0.1%  Function: ~NativeModule.cache bootstrap_node.
js:604:42
      1    0.1%    0.1%  Function: ~Buffer.toString buffer.js:609:37

 [C++]:
   ticks  total  nonlib   name
   1023   88.8%   88.8%  T node::crypto::PBKDF2(v8::FunctionCallbackInfo<v8::
Value> const&)
     27    2.3%    2.3%  t node::(anonymous namespace)::ContextifyScript::New
(v8::FunctionCallbackInfo<v8::Value> const&)
     ...

 [Summary]:
   ticks  total  nonlib   name
      3    0.3%    0.3%  JavaScript
   1105   95.9%   95.9%  C++
      3    0.3%    0.3%  GC
      0    0.0%          Shared libraries
```

书籍是人类进步的阶梯

```
     44    3.8%          Unaccounted

 [C++ entry points]:
   ticks    cpp   total   name
   1062   98.2%   92.2%  T v8::internal::Builtin_HandleApiCall(int,
v8::internal::Object**, v8::internal::Isolate*)
     13    1.2%    1.1%  T v8::internal::Runtime_CompileLazy(int,
v8::internal::Object**, v8::internal::Isolate*)
     ...

 [Bottom up (heavy) profile]:
  Note: percentage shows a share of a particular caller in the total
  amount of its parent calls.
  Callers occupying less than 1.0% are not shown.

   ticks parent  name
   1023   88.8%  T node::crypto::PBKDF2(v8::FunctionCallbackInfo<v8::Value>
const&)
   1023  100.0%    T v8::internal::Builtin_HandleApiCall(int,
v8::internal::Object**, v8::internal::Isolate*)
   1023  100.0%      Function: ~pbkdf2 crypto.js:691:16
   1023  100.0%        Function: ~exports.pbkdf2Sync crypto.js:686:30
   1023  100.0%          Function: ~hash /Users/nswbmw/Desktop/test/app.
js:3:15
   1023  100.0%            Function: ~<anonymous> /Users/nswbmw/Desktop/test/
app.js:1:11
   ...
```

打印结果包含 6 部分：Shared libraries、JavaScript、C++、Summary、C++ entry points 和 Bottom up (heavy) profile。在 JavaScript 部分列出了 JavaScript 代码执行时所占用的 CPU ticks（CPU 时钟周期）；在 C++ 部分列出了 C++ 代码执行时所占用的 CPU ticks；在 Summary 部分列出了各个部分的占比；在 Bottom up 部分列出了所有 CPU 的占用时间从大到小的函数及堆栈信息，小于 1% 的则不予显示。

可以看出，88.8% 的 CPU 时间都花在了 crypto.js 文件的 pbkdf2Sync 函数上，hash 函数在 app.js 的第 3 行被调用。

解决方法：将同步的 pbkdf2Sync 改为异步的 pbkdf2。将代码修改如下：

```
const crypto = require('crypto')
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
function hash (password, cb) {
  const salt = crypto.randomBytes(128).toString('base64')
  crypto.pbkdf2(password, salt, 10000, 64, 'sha512', cb)
}

let count = 0
console.time('pbkdf2')
for (let i = 0; i < 100; i++) {
  hash('random_password', () => {
    count++
    if (count === 100) {
      console.timeEnd('pbkdf2')
    }
  })
}
```

运行结果：

```
$ node --prof app
pbkdf2: 656.332ms
...
```

可以看出，程序运行了 656.332ms，相比较于之前的 1375.585ms，性能提升了 1 倍。我们继续看看 v8.log 的分析结果，运行：

```
$ node --prof-process isolate-0x102802400-v8.log
Statistical profiling result from isolate-0x103001a00-v8.log, (198 ticks, 19
unaccounted, 0 excluded).

[Shared libraries]:
   ticks  total  nonlib   name

 [JavaScript]:
   ticks  total  nonlib   name
      1    0.5%    0.5%  StoreIC: A store IC from the snapshot
      1    0.5%    0.5%  Function: ~set native collection.js:149:4
      1    0.5%    0.5%  Function: ~pbkdf2 crypto.js:691:16
      1    0.5%    0.5%  Function: ~inherits util.js:962:18
      1    0.5%    0.5%  Builtin: ArrayIteratorPrototypeNext

 [C++]:
   ticks  total  nonlib   name
     83   41.9%   41.9%  T ___kdebug_trace_string
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
     31   15.7%   15.7%  t node::(anonymous namespace)::ContextifyScript::New
(v8::FunctionCallbackInfo<v8::Value> const&)
     14    7.1%    7.1%  T ___pthread_sigmask
     ...

 [Summary]:
   ticks  total  nonlib   name
      5    2.5%    2.5%  JavaScript
    174   87.9%   87.9%  C++
      3    1.5%    1.5%  GC
      0    0.0%          Shared libraries
     19    9.6%          Unaccounted

 [C++ entry points]:
   ticks    cpp   total   name
     41   60.3%   20.7%  T v8::internal::Builtin_HandleApiCall(int,
v8::internal::Object**, v8::internal::Isolate*)
     17   25.0%    8.6%  T v8::internal::Runtime_CompileLazy(int,
v8::internal::Object**, v8::internal::Isolate*)
     ...

 [Bottom up (heavy) profile]:
  Note: percentage shows a share of a particular caller in the total
  amount of its parent calls.
  Callers occupying less than 1.0% are not shown.

   ticks parent  name
     83   41.9%  T ___kdebug_trace_string

     31   15.7%  t node::(anonymous namespace)::ContextifyScript::New(v8::Fun
ctionCallbackInfo<v8::Value> const&)
     31  100.0%    T v8::internal::Builtin_HandleApiCall(int,
v8::internal::Object**, v8::internal::Isolate*)
     31  100.0%      Function: ~runInThisContext bootstrap_node.js:495:28
     31  100.0%        Function: ~NativeModule.compile bootstrap_node.
js:584:44
     31  100.0%          Function: ~NativeModule.require bootstrap_node.
js:516:34
     ...
```

可以看出，在 Bottom up 部分没有很多的 ticks，而且不再有 pbkdf2 这种堆栈信息。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

1.3.2 Web UI

V8 还提供了一个 Web 可视化工具来查看生成的 v8 日志。首先，将代码还原到使用 pbkdf2Sync 的版本，运行：

```
$ node --prof app # 生成 isolate-0x103000000-v8.log
$ node --prof-process --preprocess isolate-0x103000000-v8.log > v8.json
# 格式化成 JSON 文件
$ git clone https://github.com/v8/v8.git # 克隆 v8 仓库
$ open v8/tools/profview/index.html # 打开 V8 profiling log processor
```

单击“选择文件”，选择刚才生成的 v8.json 文件，单击“Bottom up”视图，如图1-7所示。

图 1-7

有以下两点需要解释。

- 图 1-7 中的上半部分展示了 CPU 的 timeline，X 轴代表时间的流逝，Y 轴代表当前时间点的不同部分占用 CPU 的比例，可以在 timeline 图表上用鼠标左键拖动，选择时间区间。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

- 图 1-7 中的下半部分展示了在当前时间段内 CPU 占用比从大到小排列的函数，将其展开后可查看堆栈信息。不同的颜色代表了不同的部分，单击任意一个函数，在 timeline 底部会展示该函数的执行时间分布。

本节的参考链接如下。

- https://github.com/v8/v8/wiki/V8-Profiler
- https://blog.ghaiklor.com/profiling-nodejs-applications-1609b77afe4e
- https://stackoverflow.com/questions/23934451/how-to-read-nodejs-internal-profiler-tick-processor-output

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

第2章 内存

内存泄露可能是最常见的 Node.js 问题了，不当的全局缓存、事件监听、闭包等都可能导致内存泄露，本章将针对这些情况给出多种调试方案。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

2.1 gcore 与 llnode

2.1.1 Core 和 Core Dump

在开始之前，我们先了解下什么是 Core 和 Core Dump。

什么是 Core？在使用半导体作为内存材料前，人们将线圈作为内存的材料，线圈就叫作 Core，用线圈做的内存就叫作 Core Memory。如今，半导体工业蓬勃发展，已经没有人用 Core Memory 了，不过在许多情况下，人们还是把记忆体叫作 Core。

什么是 Core Dump？当程序在运行的过程中异常终止或崩溃时，操作系统会将程序当时的内存状态记录下来，保存在一个文件中，这种行为就叫作 Core Dump（有时被翻译成“核心转储”）。我们可以认为 Core Dump 是“内存快照”，但实际上，除了内存信息，还有些关键的程序运行状态会被同时 dump 下来，例如寄存器信息（包括程序指针、栈指针等）、内存管理信息、其他处理器和操作系统的状态及信息。Core Dump 对于编程人员诊断和调试程序是非常有帮助的，因为某些程序中的错误是很难重现的，例如指针异常，而 Core Dump 文件可以再现程序出错时的情景。

测试环境：

```
$ uname -a
Linux nswbmw-VirtualBox 4.13.0-36-generic #40~16.04.1-Ubuntu SMP Fri Feb 16
23:25:58 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
```

开启 Core Dump，在终端输入：

```
$ ulimit -c
```

查看允许 Core Dump 生成的文件的大小，如果是 0，则表示关闭了 Core Dump。使用以下命令开启 Core Dump，并且不限制 Core Dump 生成的文件大小：

```
$ ulimit -c unlimited
```

以上命令只在当前的终端环境下有效，如果想永久生效，就需要修改 /etc/security/limits.conf 文件，如图 2-1 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
# /etc/security/limits.conf
#
#Each line describes a limit for a user in the form:
#
#<domain>        <type>  <item>  <value>
    *            soft    core    unlimited
```

图 2-1

2.1.2 gcore

在使用 gcore 时可以不重启程序就 dump 出特定进程的 core 文件。gcore 的使用方法如下：

```
$ gcore [-o filename] pid
```

在发生 Core Dump 时，默认会在执行 gcore 命令的目录下生成 core.<PID> 文件。

2.1.3 llnode

lldb 是下一代高性能调试器，llnode 就是 LLDB 的一个支持调试 Node.js v4+ 的插件。首先，安装 llnode 和 lldb：

```
$ sudo apt-get update

# Clone llnode
$ git clone https://github.com/nodejs/llnode.git ~/llnode && cd ~/llnode

# Install lldb and headers
$ sudo apt-get install lldb-4.0 liblldb-4.0-dev

# Initialize GYP
$ git clone https://github.com/bnoordhuis/gyp.git tools/gyp

# Configure
$ ./gyp_llnode -Dlldb_dir=/usr/lib/llvm-4.0/
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
# Build
$ make -C out/ -j9

# Install
$ sudo make install-linux
```

如果 sudo apt-get update 遇到以下错误：

```
W: GPG error: xxx stable Release: The following signatures couldn't be verified
because the public key is not available: NO_PUBKEY 6DA62DE462C7DA6D
```

则可以用以下命令解决：

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
6DA62DE462C7DA6D
```

在 --recv-keys 后面跟着的是在前面报错时提示的 PUBKEY。

2.1.4 测试 Core Dump

下面通过一个典型的因全局变量缓存导致内存泄漏的例子来测试 llnode 的用法。代码如下。

app.js：

```
const leaks = []

function LeakingClass() {
  this.name = Math.random().toString(36)
  this.age = Math.floor(Math.random() * 100)
}

setInterval(() => {
  for (let i = 0; i < 100; i++) {
    leaks.push(new LeakingClass)
  }

  console.warn('Leaks: %d', leaks.length)
}, 1000)
```

运行该程序：

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
$ node app.js
```

等待几秒，打开另一个终端，运行 gcore：

```
$ ulimit -c unlimited
$ sudo gcore `pgrep -n node`
```

生成 core.2763 文件。

2.1.5 分析 Core 文件

使用 lldb 加载刚才生成的 Core 文件：

```
$ lldb-4.0 -c ./core.2763
(lldb) target create --core "./core.2763"
Core file '/home/nswbmw/test/./core.2763' (x86_64) was loaded.
(lldb)
```

输入 v8 来查看使用文档，有以下几条命令：bt、findjsinstances、findjsobjects、findrefs、inspect、nodeinfo、print、source，下面讲解这些命令的用法。

运行 v8 findjsobjects 查看所有的对象实例及总共占用的内存大小：

```
(lldb) v8 findjsobjects
 Instances  Total Size Name
 ---------- ---------- ----
       ...
       2100      84000 LeakingClass
       8834      39792 (String)
 ---------- ----------
      12088     181320
```

可以看出，LeakingClass 有 2100 个实例，占内存 84000 Byte。使用 v8 findjsinstances 查看所有的 LeakingClass 实例：

```
(lldb) v8 findjsinstances LeakingClass
0x000022aaa118ab19:<Object: LeakingClass>
0x000022aaa118acf9:<Object: LeakingClass>
0x000022aaa118ade1:<Object: LeakingClass>
...
```

书籍是人类进步的阶梯

使用 v8 i 检索实例的具体内容：

```
(lldb) v8 i 0x000022aaa118ab19
0x000022aaa118ab19:<Object: LeakingClass properties {
    .name=0x000022aaa118ab91:<String: "0.4tx00cipe8">,
    .age=<Smi: 71>}>
(lldb) v8 i 0x000022aaa118acf9
0x000022aaa118acf9:<Object: LeakingClass properties {
    .name=0x000022aaa118ad71:<String: "0.48563ixsblf">,
    .age=<Smi: 70>}>
(lldb) v8 i 0x000022aaa118ade1
0x000022aaa118ade1:<Object: LeakingClass properties {
    .name=0x000022aaa118ae59:<String: "0.w1nel407zj">,
    .age=<Smi: 80>}>
```

这样可以看到每个 LeakingClass 实例的 name 和 age 字段的值。

使用 v8 findrefs 查看引用：

```
(lldb) v8 findrefs 0x000022aaa118ab19
0x22aaa1189729: (Array)[0]=0x22aaa118ab19
(lldb) v8 i 0x22aaa1189729
0x000022aaa1189729:<Array: length=2100 {
    [0]=0x000022aaa118ab19:<Object: LeakingClass>,
    [1]=0x000022aaa118acf9:<Object: LeakingClass>,
    [2]=0x000022aaa118ade1:<Object: LeakingClass>,
    [3]=0x000022aaa118aea1:<Object: LeakingClass>,
    [4]=0x000022aaa118af61:<Object: LeakingClass>,
    [5]=0x000022aaa118b021:<Object: LeakingClass>,
    [6]=0x000022aaa118b0e1:<Object: LeakingClass>,
    [7]=0x000022aaa118b1a1:<Object: LeakingClass>,
    [8]=0x000022aaa118b221:<Object: LeakingClass>,
    [9]=0x000022aaa118b2a1:<Object: LeakingClass>,
    [10]=0x000022aaa118b321:<Object: LeakingClass>,
    [11]=0x000022aaa118b3a1:<Object: LeakingClass>,
    [12]=0x000022aaa118b421:<Object: LeakingClass>,
    [13]=0x000022aaa118b4a1:<Object: LeakingClass>,
    [14]=0x000022aaa118b521:<Object: LeakingClass>,
    [15]=0x000022aaa118b5a1:<Object: LeakingClass>}>
```

可以看出，通过一个 LeakingClass 实例的内存地址，我们使用 v8 findrefs 找到了引用它的数组的内存地址，然后通过这个地址去检索数组，得到这个数组长度为 2100，每一项都是一个 LeakingClass 实例，这不就是我们代码中的 leaks 数组吗？

书籍是人类进步的阶梯

> **小提示**
>
> v8 i 是 v8 inspect 的缩写，v8 p 是 v8 print 的缩写。

2.1.6 --abort-on-uncaught-exception

在 Node.js 程序启动时添加 --abort-on-uncaught-exception 参数，在程序崩溃时会自动 Core Dump，方便“死后验尸”。

添加 --abort-on-uncaught-exception 参数，启动测试程序：

```
$ ulimit -c unlimited
$ node --abort-on-uncaught-exception app.js
```

启动第 2 个终端，运行：

```
$ kill -BUS `pgrep -n node`
```

第 1 个终端会显示：

```
Leaks: 100
Leaks: 200
Leaks: 300
Leaks: 400
Leaks: 500
Leaks: 600
Leaks: 700
Leaks: 800
Bus error (core dumped)
```

调试步骤与上面的一致：

```
$ lldb-4.0 -c ./core
(lldb) target create --core "./core"
Core file '/home/nswbmw/test/./core' (x86_64) was loaded.
(lldb) v8 findjsobjects
 Instances  Total Size Name
 ---------- ---------- ----
        ...
```

书籍是人类进步的阶梯

```
        800       32000 LeakingClass
       7519       38512 (String)
 ---------- ----------
       9440      126368
```

2.1.7 小结

我们的测试代码很简单，没有引用任何第三方模块，如果项目较大且引用的模块较多，则将难以甄别 v8 findjsobjects 的结果，这时可以多次使用 gcore 进行 Core Dump，对比发现增长的对象，再进行诊断。

本节的参考链接如下。

- http://www.cnblogs.com/Anker/p/6079580.html
- http://www.brendangregg.com/blog/2016-07-13/llnode-nodejs-memory-leak-analysis.html

2.2 heapdump

heapdump 是一个 dump V8 堆信息的工具，v8-profiler 也具有 dump V8 堆信息的功能，这两个工具的原理都是一致的，但是 heapdump 的使用更简单些。下面以 heapdump 为例讲解如何分析 Node.js 的内存泄漏。

2.2.1 使用 heapdump

这里以一段经典的内存泄漏代码作为测试代码。

app.js：

```
const heapdump = require('heapdump')
let leakObject = null
let count = 0

setInterval(function testMemoryLeak() {
```

书籍是人类进步的阶梯

```
    const originLeakObject = leakObject
    const unused = function () {
      if (originLeakObject) {
        console.log('originLeakObject')
      }
    }
    leakObject = {
      count: String(count++),
      leakStr: new Array(1e7).join('*'),
      leakMethod: function () {
        console.log('leakMessage')
      }
    }
  }, 1000)
```

为什么这段程序会发生内存泄漏呢？首先我们要明白闭包的原理：在同一个函数内部闭包作用域只有一个，所有闭包共享。在执行函数的时候，如果遇到闭包，则会创建闭包作用域的内存空间，将该闭包所用到的局部变量添加进去，再遇到闭包时，会在之前创建好的作用域空间，添加此闭包会用到而前闭包没用到的变量。在函数结束时，会清除没有被闭包作用域引用的变量。

这段代码内存泄露原因是：在 testMemoryLeak 函数内有两个闭包：unused 和 leakMethod。unused 这个闭包引用了父作用域中的 originLeakObject 变量，如果没有后面的 leakMethod，则会在函数结束后被清除，闭包作用域也跟着被清除了。因为后面的 leakObject 是全局变量，即 leakMethod 是全局变量，它引用的闭包作用域（包含了 unused 所引用的 originLeakObject）不会被释放。而随着 testMemoryLeak 的不断调用，originLeakObject 指向前一次的 leakObject，下一次的 leakObject.leakMethod 又会引用之前的 originLeakObject，从而形成一个闭包引用链，而 leakStr 是一个大字符串，得不到释放，从而造成了内存泄漏。

解决方法：在 testMemoryLeak 函数内部的最后添加 originLeakObject = null 。

运行测试代码：

```
$ node app
```

然后先后执行两次：

```
$ kill -USR2 `pgrep -n node`
```

书籍是人类进步的阶梯

最终，在当前目录下生成了两个 heapsnapshot 文件：

```
heapdump-100427359.61348.heapsnapshot
heapdump-100438986.797085.heapsnapshot
```

2.2.2 Chrome DevTools

我们使用 Chrome DevTools 来分析前面生成的 heapsnapshot 文件。调出 Chrome DevTools → Memory → Load，按顺序依次加载前面生成的 heapsnapshot 文件。单击第 2 个堆快照，在左上角有个下拉菜单，有如下 4 个选项。

- Summary：以构造函数名分类显示。
- Comparison：比较多个快照之间的差异。
- Containment：查看整个 GC 路径。
- Statistics：以饼状图显示内存占用信息。

通常我们只会用前两个选项，一般用不到第 3 个选项，因为在展开 Summary 和 Comparison 中的每一项时，都可以看到从 GC roots 到这个对象的路径；从第 4 个选项中只能看到内存占比，如图 2-2 所示。

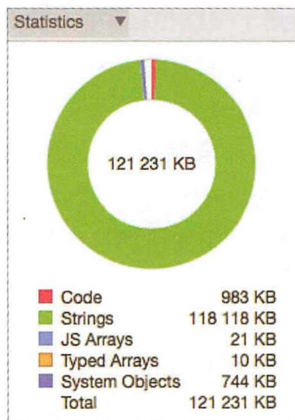


图 2-2

切换到 Summary 页，可以看到有如下 5 个属性。

- Contructor：指构造函数名，例如 Object、Module、Socket、(array)、(string)、(regexp) 等，加了括号的分别代表内置的 Array、String 和 Regexp。

书籍是人类进步的阶梯

- Distance：指到 GC roots（GC 根对象）的距离。GC 根对象在浏览器中一般是 window 对象，在 Node.js 中是 global 对象。距离越大，则说明引用越深。
- Objects Count：指对象的个数。
- Shallow Size：指对象自身的大小，不包括它引用的对象。
- Retained Size：指对象自身的大小和它引用的对象的大小，即该对象被 GC 之后所能回收的内存大小。

小提示

一个对象的 Retained Size 为该对象的 Shallow Size 与该对象支配树上其子节点的 Retained Size 之和。Shallow Size 等于 Retained Size 的有 (boolean)、(number)、(string)，它们无法引用其他值，并且始终是叶子节点。

单击 Retained Size，选择降序展示，可以看到 (closure) 这一项引用的内容达到 98%，继续展开后如图 2-3 所示。

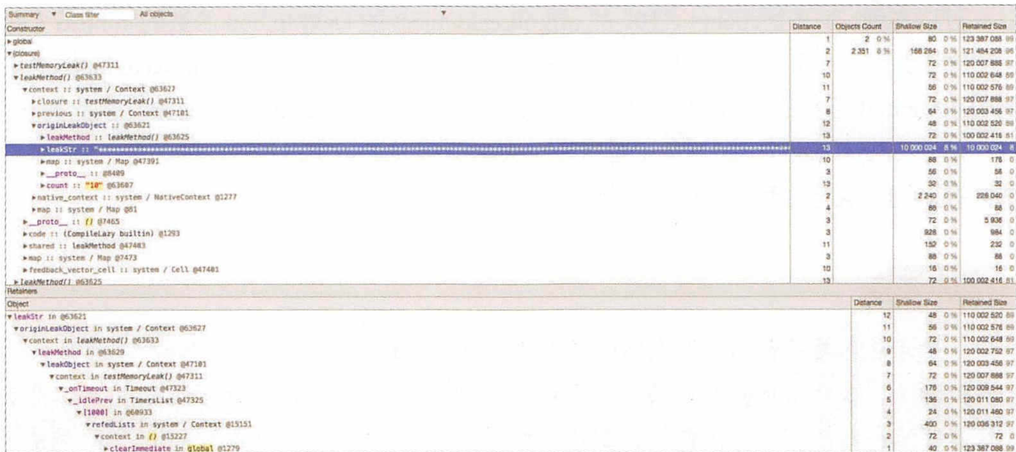


图 2-3

可以看到，一个 leakStr 占用了 8% 的内存，而 leakMethod 引用了 81% 的内存。对象保留树（Retainers，在老版本的 Chrome 中叫作 Object's retaining tree）展示了对象的 GC path，单击如图 2-3 所示的 leakStr（Distance 是 13），Retainers 会自动展开，Distance 从 13 递减到 1。

书籍是人类进步的阶梯

继续展开 leakMethod，如图 2-4 所示。

图 2-4

可以看到，有一个 count= "10" 的 originLeakObject 的 leakMethod 函数的 context（即上下文）引用了一个 count= "9" 的 originLeakObject 对象，而这个 originLeakObject 对象的 leakMethod 函数的 context 又引用了 count= "8" 的 originLeakObject 对象，以此类推。而在每个 originLeakObject 对象上都有一个大字符串 leakStr（占用 8% 的内存），从而造成内存泄漏，这符合我们之前的推断。

小提示

若背景色是黄色的，则表示这个对象在 JavaScript 中还存在引用，所以可能没有被清除。若背景色是红色的，则表示这个对象在 JavaScript 中不存在引用，但是依然存活在内存中，一般常见于 DOM 对象，它们存放的位置和 JavaScript 中的对象还是不同的，在 Node.js 中很少遇见。

2.2.3 对比快照

切换到 Comparison 视图下，可以看到 #New、#Deleted、#Delta 等属性，+ 和 – 表

书籍是人类进步的阶梯

示相对于比较的堆快照而言。我们对比第 2 个快照和第 1 个快照，如图 2-5 所示。

图 2-5

可以看出，(string) 增加了 10 个，每个 string 的大小为 10 000 024 字节。

本节的参考链接如下。

- https://blog.meteor.com/an-interesting-kind-of-javascript-memory-leak-8b47d2e7f156
- https://www.zhihu.com/question/56806069
- http://taobaofed.org/blog/2016/04/15/how-to-find-memory-leak/
- https://developers.google.com/web/tools/chrome-devtools/memory-problems/memory-101

2.3　memwatch-next

memwatch-next（以下简称 memwatch）是一个用来监测 Node.js 的内存泄漏和比较堆信息的模块。下面以一段事件监听器导致内存泄漏的代码为例，讲解如何使用 memwatch。

2.3.1　使用 memwatch-next

测试代码如下。

书籍是人类进步的阶梯

app.js：

```
let count = 1
const memwatch = require('memwatch-next')
memwatch.on('stats', (stats) => {
  console.log(count++, stats)
})
memwatch.on('leak', (info) => {
  console.log('---')
  console.log(info)
  console.log('---')
})

const http = require('http')
const server = http.createServer((req, res) => {
  for (let i = 0; i < 10000; i++) {
    server.on('request', function leakEventCallback() {})
  }
  res.end('Hello World')
  global.gc()
}).listen(3000)
```

在每个请求到来时，为 server 注册 10000 个 request 事件的监听函数（有大量的事件监听函数被存储到内存中，从而造成内存泄漏），然后手动触发一次 GC。

运行该程序：

```
$ node --expose-gc app.js
10:19:48 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
```

注 意

这里添加 --expose-gc 参数来启动程序，这样我们才可以在程序中手动触发 GC。

memwatch 监听以下两个事件。

（1）stats：GC 事件，每执行一次 GC，都会触发该函数并打印 heap 相关的信息。代码如下：

```
{
  num_full_gc: 1,// 完整的垃圾回收次数
```

书籍是人类进步的阶梯

```
    num_inc_gc: 1,// 增长的垃圾回收次数
    heap_compactions: 1,// 内存压缩次数
    usage_trend: 0,// 使用趋势
    estimated_base: 5350136,// 预期基数
    current_base: 5350136,// 当前基数
    min: 0,// 最小值
    max: 0// 最大值
  }
```

（2）leak：为可疑的内存泄露事件，触发该事件的条件是内存在连续 5 次 GC 后都是增长的。打印的内容如下：

```
  {
    growth: 4051464,
    reason: 'heap growth over 5 consecutive GCs (2s) - -2147483648 bytes/hr'
  }
```

运行：

```
$ ab -c 1 -n 5 http://localhost:3000/
```

输出：

```
(node:20989) MaxListenersExceededWarning: Possible EventEmitter memory leak
detected. 11 request listeners added. Use emitter.setMaxListeners() to
increase limit
1 { num_full_gc: 1,
  num_inc_gc: 1,
  heap_compactions: 1,
  usage_trend: 0,
  estimated_base: 5720064,
  current_base: 5720064,
  min: 0,
  max: 0 }
2 { num_full_gc: 2,
  num_inc_gc: 1,
  heap_compactions: 2,
  usage_trend: 0,
  estimated_base: 7073824,
  current_base: 7073824,
  min: 0,
  max: 0 }
3 { num_full_gc: 3,
```

书籍是人类进步的阶梯

```
    num_inc_gc: 1,
    heap_compactions: 3,
    usage_trend: 0,
    estimated_base: 7826368,
    current_base: 7826368,
    min: 7826368,
    max: 7826368 }
  4 { num_full_gc: 4,
    num_inc_gc: 1,
    heap_compactions: 4,
    usage_trend: 0,
    estimated_base: 8964784,
    current_base: 8964784,
    min: 7826368,
    max: 8964784 }
  ---
  { growth: 3820272,
    reason: 'heap growth over 5 consecutive GCs (0s) - -2147483648 bytes/hr' }
  ---
  5 { num_full_gc: 5,
    num_inc_gc: 1,
    heap_compactions: 5,
    usage_trend: 0,
    estimated_base: 9540336,
    current_base: 9540336,
    min: 7826368,
    max: 9540336 }
```

可以看出，Node.js 已经警告我们事件监听器超过了 11 个，可能造成内存泄露。连续的 5 次内存增长触发了 leak 事件，打印出增长了多少内存（Bytes）和预估每小时增长多少 Bytes。

2.3.2 使用 Heap Diff

memwatch 有一个 HeapDiff 函数，用来对比、计算两次堆快照的差异。将测试代码修改如下：

```
const memwatch = require('memwatch-next')
const http = require('http')
const server = http.createServer((req, res) => {
  for (let i = 0; i < 10000; i++) {
```

书籍是人类进步的阶梯

```
    server.on('request', function leakEventCallback() {})
  }
  res.end('Hello World')
  global.gc()
}).listen(3000)

const hd = new memwatch.HeapDiff()
memwatch.on('leak', (info) => {
  const diff = hd.end()
  console.dir(diff, { depth: 10 })
})
```

运行这段代码并执行同样的 ab 命令，打印如下：

```
{ before: { nodes: 35727, size_bytes: 4725128, size: '4.51 mb' },
  after: { nodes: 87329, size_bytes: 8929792, size: '8.52 mb' },
  change:
   { size_bytes: 4204664,
     size: '4.01 mb',
     freed_nodes: 862,
     allocated_nodes: 52464,
     details:
      [ ...
        { what: 'Array',
          size_bytes: 530200,
          size: '517.77 kb',
          '+': 1023,
          '-': 510 },
        { what: 'Closure',
          size_bytes: 3599856,
          size: '3.43 mb',
          '+': 50001,
          '-': 3 },
        ...
      ]
   }
}
```

可以看出，内存由 4.51MB 增加到 8.52MB，其中 Closure 和 Array 的增加占了很大的比例，而我们知道注册事件监听函数的本质就是将事件函数（Closure）push 到相应的数组（Array）里。

书籍是人类进步的阶梯

2.3.3 结合 heapdump 使用

memwatch 在结合 heapdump 使用时才能发挥更好的作用。通常用 memwatch 监测到内存泄漏，用 heapdump 导出多份堆快照，然后用 Chrome DevTools 分析和比较，定位内存泄漏的元凶。

将代码修改如下：

```
const memwatch = require('memwatch-next')
const heapdump = require('heapdump')

const http = require('http')
const server = http.createServer((req, res) => {
  for (let i = 0; i < 10000; i++) {
    server.on('request', function leakEventCallback() {})
  }
  res.end('Hello World')
  global.gc()
}).listen(3000)

dump()
memwatch.on('leak', () => {
  dump()
})

function dump() {
  const filename = `${__dirname}/heapdump-${process.pid}-${Date.now()}.
heapsnapshot`

  heapdump.writeSnapshot(filename, () => {
    console.log(`${filename} dump completed.`)
  })
}
```

以上程序在启动后先执行一次 heap dump，在触发 leak 事件时再执行一次 heap dump。运行这段代码并执行同样的 ab 命令，生成两个 heapsnapshot 文件：heapdump-21126-1519545957879.heapsnapshot 和 heapdump-21126-1519545975702.heapsnapshot。

书籍是人类进步的阶梯

用 Chrome DevTools 加载这两个 heapsnapshot 文件，选择 comparison 比较视图，如图 2-6 所示。

Comparison ▼　Class filter　heapdump-21126-1518545957879 ▼

Constructor	# New	# Deleted	# Delta	Alloc. Size	Freed Size	Size Delta
▼(closure)	50 002	4	+49 998	3 600 144	288	+3 599 8
▶() @71937	•			72		
▶() @71947	•			72		
▶leakEventCallback() @71965	•			72		
▶leakEventCallback() @71973	•			72		
▶leakEventCallback() @71975	•			72		
▶leakEventCallback() @71977	•			72		
▶leakEventCallback() @71979	•			72		
▶leakEventCallback() @71981	•			72		
▶leakEventCallback() @71983	•			72		
▶leakEventCallback() @71985	•			72		
▶leakEventCallback() @71987	•			72		
▶leakEventCallback() @71989	•			72		
▶leakEventCallback() @71991	•			72		
▶leakEventCallback() @71993	•			72		
▶leakEventCallback() @71995	•			72		
▶leakEventCallback() @71997	•			72		
▶leakEventCallback() @71999	•			72		
▶leakEventCallback() @72001	•			72		

图 2-6

可以看出，增加了 5 万个 leakEventCallback 函数，单击其中任意一个，都可以从 Retainers 中看到更详细的信息，例如 GC path 和所在的文件等信息。

本节的参考链接如下。

- https://github.com/marcominetti/node-memwatch

2.4　cpu-memory-monitor

前面介绍了 heapdump 和 memwatch-next 的用法，但在实际使用时并不那么方便，我们总不能一直盯着服务器的状况，在发现内存持续增长并超过心里的阈值时，再手动去触发 Core Dump 吧？因为在大多数情况下，在发现问题时就已经错过了现场。所以，我们可能需要 cpu-memory-monitor。顾名思义，这个模块可以用来监控 CPU 和 Memory 的使用情况，并可以根据配置策略自动 dump CPU 的使用情况（Cpuprofile）和内存快照（Heapsnapshot）。

2.4.1　使用 cpu-memory-monitor

先来看看如何使用 cpu-memory-monitor，其实很简单，只需在进程启动的入口文件中引入以下代码：

书籍是人类进步的阶梯

```
require('cpu-memory-monitor')({
  cpu: {
    interval: 1000,
    duration: 30000,
    threshold: 60,
    profileDir: '/tmp',
    counter: 3,
    limiter: [5, 'hour']
  }
})
```

以上代码的作用是：每 1000ms（interval）检查一次 CPU 的使用情况，如果发现连续 3（counter）次 CPU 使用率大于 60%（threshold），则 dump 30000ms（duration）CPU 的使用情况，在 /tmp（profileDir）目录下生成 cpu-${process.pid}-${Date.now()}.cpuprofile，1（limiter[1]）小时最多 dump 5（limiter[0]）次。

以上是自动 dump CPU 使用情况的策略。dump Memory 使用情况的策略同理：

```
require('cpu-memory-monitor')({
  memory: {
    interval: 1000,
    threshold: '1.2gb',
    profileDir: '/tmp',
    counter: 3,
    limiter: [3, 'hour']
  }
})
```

上述代码的作用是：每 1000ms（interval）检查一次 Memory 的使用情况，如果发现连续 3（counter）次 Memory 大于 1.2GB（threshold），则 dump 一次 Memory，在 /tmp（profileDir）目录下生成 memory-${process.pid}-${Date.now()}.heapsnapshot，1（limiter[1]）小时最多 dump 3（limiter[0]）次。

注 意

Memory 的配置没有 duration 参数，因为 Memroy 的 dump 只是某一时刻的，而不是一段时间的。

书籍是人类进步的阶梯

聪明的你肯定会问了：能不能将 CPU 和 Memory 配置一块使用? 代码如下。

```
require('cpu-memory-monitor')({
  cpu: {
    interval: 1000,
    duration: 30000,
    threshold: 60,
    ...
  },
  memory: {
    interval: 10000,
    threshold: '1.2gb',
    ...
  }
})
```

答案是：可以，但不要这么做。因为这样做可能会出现这种情况：首先，内存增加且达到设定的阈值，会触发 Memory Dump/GC，导致 CPU 的使用率增加且达到设定的阈值；然后，会触发 CPU Dump，导致堆积的请求越来越多（比如在内存中堆积了很多 SQL 查询）；最后，会触发 Memory Dump，导致雪崩。

所以，在通常情况下只使用其中一种就可以了。

2.4.2　cpu-memory-monitor 源码解读

cpu-memory-monitor 的源码不过百余行，大体逻辑如下：

```
...
const processing = {
  cpu: false,
  memory: false
}

const counter = {
  cpu: 0,
  memory: 0
}

function dumpCpu(cpuProfileDir, cpuDuration) { ... }
function dumpMemory(memProfileDir) { ... }
```

书籍是人类进步的阶梯

```
module.exports = function cpuMemoryMonitor(options = {}) {
  ...
  if (options.cpu) {
    const cpuTimer = setInterval(() => {
      if (processing.cpu) {
        return
      }
      pusage.stat(process.pid, (err, stat) => {
        if (err) {
          clearInterval(cpuTimer)
          return
        }
        if (stat.cpu > cpuThreshold) {
          counter.cpu += 1
          if (counter.cpu >= cpuCounter) {
            memLimiter.removeTokens(1, (limiterErr, remaining) => {
              if (limiterErr) {
                return
              }
              if (remaining > -1) {
                dumpCpu(cpuProfileDir, cpuDuration)
                counter.cpu = 0
              }
            })
          } else {
            counter.cpu = 0
          }
        }
      })
    }, cpuInterval)
  }

  if (options.memory) {
    ...
    memwatch.on('leak', () => {
      dumpMemory(...)
    })
  }
}
```

可以看出，cpu-memory-monitor 没有用到什么新鲜的内容，还是之前讲解过的 v8-profiler、heapdump、memwatch-next 的组合使用而已。

书籍是人类进步的阶梯

有以下几点需要注意：

- 只有传入了 CPU 或者 Memory 的配置，才会去监听相应的 CPU 或者 Memory；
- 在传入 Memory 配置时，用 memwatch-next 额外监听了 leak 事件，也会 dump Memory，格式是 leak-memory-${process.pid}-${Date.now()}.heapsnapshot；
- 在顶部引入了 heapdump，所以即使没有 Memory 配置，也可以通过 kill -USR2 <PID> 手动触发 Memory Dump。

本节的参考链接如下。

- https://github.com/node-inspector/v8-profiler
- https://github.com/bnoordhuis/node-heapdump
- https://github.com/marcominetti/node-memwatch

书籍是人类进步的阶梯

第3章

代码

如何写出优雅、高性能和错误栈清晰的代码也是调试环节中重要的一部分。本章将会从多个方面讲解 Node.js 的调试和调优方案，还会对涉及的部分 ES6 特性进行深入讲解。

书籍是人类进步的阶梯

3.1 Promise

如何写出清晰、优雅的代码也是调试环节中重要的一部分，而在过去很长一段时间内，JavaScript 最令人吐槽的就是回调地狱（callback hell）了。先看一段代码：

```
step1(function (err, value1) {
  if (err) {
    ...
    return
  }
  step2(value1, function (err, value2) {
    if (err) {
      ...
      return
    }
    step3(value2, function (err, value3) {
      if (err) {
        ...
        return
      }
      // Do something with value3
    })
  })
})
```

上面的代码依次执行了 step1、step2 和 step3，且后一个函数用到了前一个函数执行的结果。这只是一个简单的例子，在真实环境下可能会写出嵌套更深的回调函数，代码形成一个倒金字塔。如果使用 Promise，代码就会优雅很多，如下所示：

```
step1()
  .then(step2)
  .then(step3)
  .catch((e) => {
    // Do something with error
  })
```

Promise 的出现就是为了解决回调地狱的问题，它最早是由社区提出和实现的，衍生的规范也有很多，最终 ES6 采用了 Promise/A+ 规范，并将其写进了语言标准，统一了用法。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

3.1.1 Promise/A+ 规范

Promise 规范有很多，例如 Promise/A，Promise/B，Promise/D 及 Promise/A 的升级版 Promise/A+，细节各有不同，最终在 ES6 中采用了 Promise/A+ 规范。在讲解 Promise 的实现之前，当然要先了解 Promise/A+ 规范。Promise/A+ 规范的参考如下。

- 英文版：https://promisesaplus.com/
- 中文版：http://www.ituring.com.cn/article/66566

该规范虽然不长，但细节比较多，笔者挑出以下要点简单说明一下。

- Promise 本质是一个状态机。每个 promise 只能是 3 种状态中的一种：pending、fulfilled 或 rejected。状态转变只能是 pending -> fulfilled 或者 pending -> rejected，状态转变不可逆。
- then 方法可以被同一个 promise 调用多次。
- then 方法必须返回一个 promise，从而实现链式调用。
- 值穿透。

Promise 的 API 并不多，但是 Promise 并不简单，如何透彻理解并玩转 Promise 呢？当然是从头实现一遍 Promise 啦。

3.1.2 从零开始实现 Promise

我们知道 Promise 在本质上是一个构造函数，需要用 new 调用，并有以下几个 api：

```
function Promise (resolver) {}

Promise.prototype.then = function () {}
Promise.prototype.catch = function () {}

Promise.resolve = function () {}
Promise.reject = function () {}
Promise.all = function () {}
Promise.race = function () {}
```

创建以下初始代码，然后开始一步一步地构建完整的 Promise 实现。如下所示：

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
function INTERNAL () {}
function isFunction (func) {
  return typeof func === 'function'
}
function isObject (obj) {
  return typeof obj === 'object'
}
function isArray (arr) {
  return Array.isArray(arr)
}

const PENDING = 'pending'
const FULFILLED = 'fulfilled'
const REJECTED = 'rejected'

module.exports = Promise

function Promise (resolver) {
  if (!isFunction(resolver)) {
    throw new TypeError('resolver must be a function')
  }
  this.state = PENDING
  this.value = void 0
  this.queue = []
  if (resolver !== INTERNAL) {
    safelyResolveThen(this, resolver)
  }
}
```

注 意

以下 promise 均指代 Promise 实例。

INTERNAL 就是一个空函数，后面会用来传入 Promise 构造函数来生成一个 promise 实例。这里定义了 3 个辅助函数：isFunction、isObject 和 isArray，并定义了 3 种状态：PENDING、FULFILLED 和 REJECTED。promise 内部有三个变量，如下所述。

- state：指当前 promise 的状态，初始值为 PENDING。状态改变只能是 PENDING -> FULFILLED 或 PENDING -> REJECTED。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

- value：初始值是 void 0（即 undefined），当 state 是 FULFILLED 时存储返回值，当 state 是 REJECTED 时存储错误。
- queue：promise 内部的回调队列，后面会讲它的作用。

3.1.3 Promise 的实现原理

笔者发布了一个以 Promise/A+ 规范实现的模块——appoint，我们通过这个模块研究一下它是如何实现 Promise 的。看一段代码：

```
const Promise = require('appoint')
const promise = new Promise((resolve) => {
  setTimeout(() => {
    resolve('haha')
  }, 1000)
})
const a = promise.then(function onSuccess () {})
const b = promise.catch(function onError () {})
console.dir(promise, { depth: 10 })
console.log(promise.queue[0].promise === a)
console.log(promise.queue[1].promise === b)
```

运行后打印出：

```
Promise {
  state: 'pending',
  value: undefined,
  queue:
   [ QueueItem {
       promise: Promise { state: 'pending', value: undefined, queue: [] },
       callFulfilled: [Function],
       callRejected: [Function] },
     QueueItem {
       promise: Promise { state: 'pending', value: undefined, queue: [] },
       callFulfilled: [Function],
       callRejected: [Function] } ] }
true
true
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

注 意

原生 Promise 是没有 queue 属性的，在 appoint 的实现中添加了这个属性。

可以看出，在 queue 数组中有两个对象，因为规范中规定 then 方法可以被同一个 promise 调用多次。在上个例子中在调用 .then 和 .catch 时 promise 并没有被 resolve，所以将 .then 和 .catch 生成的新 promise（a 和 b）和正确时的回调（将 onSuccess 包装成 callFulfilled）和错误时的回调（将 onError 包装成 callRejected）生成一个 QueueItem 实例并 push 到 queue 数组里，所以两个 console.log 都打印了 true。当 promise 状态改变时遍历内部 queue 数组，统一执行成功（callFulfilled）或失败（callRejected）的回调（传入 promise 的 value 值），以生成的结果分别设置 a 和 b 的 state 和 value，这就是 Promise 实现的基本原理。再来看另一个例子：

```
const Promise = require('appoint')
const promise = new Promise((resolve) => {
  setTimeout(() => {
    resolve('haha')
  }, 1000)
})
promise
  .then(() => {})
  .then(() => {})
  .then(() => {})
console.dir(promise, { depth: 10 })
```

打印出：

```
Promise {
  state: 'pending',
  value: undefined,
  queue:
   [ QueueItem {
       promise:
        Promise {
          state: 'pending',
          value: undefined,
          queue:
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
              [ QueueItem {
                  promise:
                   Promise {
                     state: 'pending',
                     value: undefined,
                     queue:
                      [ QueueItem {
                          promise: Promise { state: 'pending', value: undefined,
    queue: [] },
                          callFulfilled: [Function],
                          callRejected: [Function] } ] },
                  callFulfilled: [Function],
                  callRejected: [Function] } ] },
          callFulfilled: [Function],
          callRejected: [Function] } ] }
```

链式调用了 3 次 .then，在每次调用 .then 时都将它生成的 promise 放到了调用它的 promise 队列里，形成了 3 层调用关系。当最外层的 promise 状态改变时，遍历它的 queue 数组会调用对应的回调，设置子 promise 的 state 和 value 并遍历它的 queue 数组调用对应的回调，以此类推。

> **注　意**
>
> 这里 queue 是嵌套的，而不是像上个例子中 queue 是平铺的。

3.1.4　safelyResolveThen

接下来完成 safelyResolveThen 的逻辑，代码如下：

```
function safelyResolveThen (self, then) {
  let called = false
  try {
    then(function (value) {
      if (called) {
        return
      }
      called = true
      doResolve(self, value)
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    }, function (error) {
      if (called) {
        return
      }
      called = true
      doReject(self, error)
    })
  } catch (error) {
    if (called) {
      return
    }
    called = true
    doReject(self, error)
  }
}
```

safelyResolveThen 顾名思义用于“安全地执行 then 函数”，这里的 then 函数指“第 1 个参数是 resolve 函数，第 2 个参数是 reject 函数的函数”，适用于以下两种情况。

（1）构造函数的参数，即这里的 resolver：

```
new Promise(function resolver (resolve, reject) {
  setTimeout(() => {
    resolve('haha')
  }, 1000)
})
```

（2）promise 的 then：

```
promise.then(resolve, reject)
```

safelyResolveThen 有如下 3 个作用。

（1）try…catch 用来捕获函数内抛出的异常，例如在构造函数内抛出异常：

```
new Promise(function resolver (resolve, reject) {
  throw new Error('Oops')
})
```

（2）called 控制 resolve 或 reject 只执行一次，多次调用没有任何作用。即：

```
const Promise = require('appoint')
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
const promise = new Promise(function resolver (resolve, reject) {
  setTimeout(() => {
    resolve('haha')
  }, 1000)
  reject('error')
})
promise.then(console.log)
promise.catch(console.error)
```

可看到，打印了 error，不会再打印 haha。

（3）若没有错误则执行 doResolve，若有错误则执行 doReject。

3.1.5 doResolve 和 doReject

doResolve 和 doReject 的相关代码如下：

```
function doResolve (self, value) {
  try {
    const then = getThen(value)
    if (then) {
      safelyResolveThen(self, then)
    } else {
      self.state = FULFILLED
      self.value = value
      self.queue.forEach(function (queueItem) {
        queueItem.callFulfilled(value)
      })
    }
    return self
  } catch (error) {
    return doReject(self, error)
  }
}

function doReject (self, error) {
  self.state = REJECTED
  self.value = error
  self.queue.forEach(function (queueItem) {
    queueItem.callRejected(error)
  })
  return self
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
}
```

doReject 用来设置 promise 的 state 为 REJECTED，value 为 error，然后遍历 queue，设置所有子 promise 的状态为 REJECTED 和值为 error。doResolve 结合 safelyResolveThen 使用，不断地解包 promise，直至返回值是非 promise 对象，再设置 promise 的状态和值，然后设置子 promise 的状态和值。

这里有个辅助函数 getThen：

```
function getThen (promise) {
  const then = promise && promise.then
  if (promise && (isObject(promise) || isFunction(promise)) &&
isFunction(then)) {
    return function applyThen () {
      then.apply(promise, arguments)
    }
  }
}
```

getThen 遵循了规范中的规定：如果 then 是函数，则将 x（即被调用的 promise）作为函数的 this 调用。

3.1.6 Promise.prototype.then 和 Promise.prototype.catch

接下来实现 Promise.prototype.then 和 Promise.prototype.catch，代码如下：

```
Promise.prototype.then = function (onFulfilled, onRejected) {
  if ((!isFunction(onFulfilled) && this.state === FULFILLED) ||
    (!isFunction(onRejected) && this.state === REJECTED)) {
    return this
  }
  const promise = new this.constructor(INTERNAL)
  if (this.state !== PENDING) {
    const resolver = this.state === FULFILLED ? onFulfilled : onRejected
    unwrap(promise, resolver, this.value)
  } else {
    this.queue.push(new QueueItem(promise, onFulfilled, onRejected))
  }
  return promise
}
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
Promise.prototype.catch = function (onRejected) {
  return this.then(null, onRejected)
}
```

以上代码中的 return this 实现了值穿透，后面会细讲。可以看出，在 then 方法中生成了一个新的 promise 然后返回。如果 promise 的状态改变了，则调用 unwrap，否则将生成的 promise 加入当前 promise 的回调队列 queue 里，之前已经讲解了如何消费 queue。有以下 3 点需要讲解。

（1）Promise 构造函数传入了一个 INTERNAL 空函数，因为这个新产生的 promise 可以被认为是内部的 promise，需要根据外部的 promise 的状态和值产生自身的状态和值，不需要传入回调函数；而外部 Promise 需要传入回调函数来决定它的状态和值。所以之前在 Promise 的构造函数里做了判断，以区分是外部调用还是内部调用：

```
if (resolver !== INTERNAL) {
  safelyResolveThen(this, resolver)
}
```

QueueItem 的代码如下：

```
function QueueItem (promise, onFulfilled, onRejected) {
  this.promise = promise
  this.callFulfilled = function (value) {
    doResolve(this.promise, value)
  }
  this.callRejected = function (error) {
    doReject(this.promise, error)
  }
  if (isFunction(onFulfilled)) {
    this.callFulfilled = function (value) {
      unwrap(this.promise, onFulfilled, value)
    }
  }
  if (isFunction(onRejected)) {
    this.callRejected = function (error) {
      unwrap(this.promise, onRejected, error)
    }
  }
}
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

（2）promise 为 then 生成的新 promise，onFulfilled 和 onRejected 是 then 参数中的 onFulfilled 和 onRejected。从上面的代码可以看出，当 promise 的状态变为 FULFILLED 时，之前注册的 then 函数通过 callFulfilled 调用 unwrap 进行解包，最终得出 promise 的状态和值；之前注册的 catch 函数用 callRejected 直接调用 doReject，设置队列里 promise 的状态和值。当 promise 的状态变为 REJECTED 时类似。

unwrap 的代码如下：

```
function unwrap (promise, func, value) {
  process.nextTick(function () {
    let returnValue
    try {
      returnValue = func(value)
    } catch (error) {
      return doReject(promise, error)
    }
    if (returnValue === promise) {
      doReject(promise, new TypeError('Cannot resolve promise with itself'))
    } else {
      doResolve(promise, returnValue)
    }
  })
}
```

（3）unwrap 函数顾名思义是用来解包的，即用父 promise 的结果设置当前 promise 的状态和值。第 1 个参数是 promise，第 2 个参数是父 promise 的 then 的回调（onFulfilled/onRejected），第 3 个参数是父 promise 的值（正常值 / 错误）。

有以下 3 点需要说明。

（1）使用 process.nextTick 将代码异步执行，这也是在规范里明确规定的。看一段代码：

```
const Promise = require('appoint')
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('haha')
  }, 1000)
})
promise.then(() => {
  promise.then(() => {
```

书籍是人类进步的阶梯

```
    console.log('1')
  })
  console.log('2')
})
```

打印 2 1，去掉 process.nextTick 则打印 1 2。

（2）try…catch 用来捕获 then/catch 函数内抛出的异常，并调用 doReject，例如：

```
promise.then(() => {
  throw new Error('haha')
})
promise.catch(() => {
  throw new Error('haha')
})
```

（3）返回的值不能是 promise 本身，否则会造成死循环，代码如下：

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('haha')
  }, 1000)
})
const a = promise.then(() => {
  return a
})

a.catch(console.log)// [TypeError: Chaining cycle detected for promise
#<Promise>]
```

注 意

promise.catch(onRejected) 是 promise.then(null, onRejected) 的语法糖。

至此，Promise 的核心部分就实现了。

3.1.7 值穿透

上面提到好几次值穿透，什么是值穿透呢？在上面的 Promise.prototype.then 的实现中有这么一段代码：

书籍是人类进步的阶梯

```
Promise.prototype.then = function (onFulfilled, onRejected) {
  if ((!isFunction(onFulfilled) && this.state === FULFILLED) ||
    (!isFunction(onRejected) && this.state === REJECTED)) {
    return this
  }
  ...
};
```

值穿透即传入 then/catch 的参数如果不为函数，则忽略该值，返回上一个 promise 的结果。看一段代码：

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('haha')
  }, 1000)
})
promise
  .then('hehe')
  .then(console.log)
```

最终打印 haha 而不是 hehe。

通过 return this 只实现了值穿透的一种情况，其实值穿透有如下两种情况。

（1）在 promise 已经是 FULFILLED/REJECTED 时，通过 return this 实现值穿透：

```
const Promise = require('appoint')
const promise = new Promise(function (resolve) {
  setTimeout(() => {
    resolve('haha')
  }, 1000)
})
promise.then(() => {
  promise.then().then((res) => {// [1]
    console.log(res)// haha
  })
  promise.catch().then((res) => {// [2]
    console.log(res)// haha
  })
  console.log(promise.then() === promise.catch())// true
  console.log(promise.then(1) === promise.catch({ name: 'nswbmw' }))// true
})
```

书籍是人类进步的阶梯

上述代码 [1]、[2] 处的 promise 已经是 FULFILLED 了，符合条件，所以执行了 return this。

（2）在 promise 是 PENDING 时，通过生成新的 promise 加入父 promise 的 queue 中，在父 promise 状态改变时调用 callFulfilled->doResolve 或 callRejected->doReject（因为 then/catch 传入的参数不是函数）设置子 promise 的状态和值为父 promise 的状态和值。

看一段代码：

```
const Promise = require('appoint')
const promise = new Promise((resolve) => {
  setTimeout(() => {
    resolve('haha')
  }, 1000)
})
const a = promise.then()
a.then((res) => {
  console.log(res)// haha
})
const b = promise.catch()
b.then((res) => {
  console.log(res)// haha
})
console.log(a === b)// false
```

3.1.8 Promise.resolve 和 Promise.reject

Promise.resolve 和 Promise.reject 是 Promise 的两个静态方法，用来快捷地生成一个状态为 fulfilled 或者 rejected 的 promise 实例。代码如下：

```
Promise.resolve = resolve
function resolve (value) {
  if (value instanceof this) {
    return value
  }
  return doResolve(new this(INTERNAL), value)
}

Promise.reject = reject
function reject (reason) {
```

书籍是人类进步的阶梯

```
    return doReject(new this(INTERNAL), reason)
  }
```

当 Promise.resolve 参数是一个 promise 时，直接返回该值。

3.1.9 Promise.all

Promise.all 接收一个数组，用来并行执行一组 promise。代码如下：

```
Promise.all = all
function all (iterable) {
  const self = this
  if (!isArray(iterable)) {
    return this.reject(new TypeError('must be an array'))
  }

  const len = iterable.length
  let called = false
  if (!len) {
    return this.resolve([])
  }

  const values = new Array(len)
  let resolved = 0
  let i = -1
  const promise = new this(INTERNAL)

  while (++i < len) {
    allResolver(iterable[i], i)
  }
  return promise
  function allResolver (value, i) {
    self.resolve(value).then(resolveFromAll, function (error) {
      if (!called) {
        called = true
        doReject(promise, error)
      }
    })
    function resolveFromAll (outValue) {
      values[i] = outValue
      if (++resolved === len && !called) {
        called = true
```

书籍是人类进步的阶梯

```
            doResolve(promise, values)
          }
        }
      }
    }
```

Promise.all 用来并行执行多个 promise/ 值，当所有 promise/ 值执行完毕或有一个 promise 状态变为 rejected 时返回。从以上代码可以看出，

- 在 Promise.all 内部生成了一个新的 promise 返回；
- called 用来控制即使有多个 promise rejected 也只有第 1 个生效；
- values 用来存储执行结果；
- 在最后一个 promise 状态改变后，使用 doResolve(promise, values) 设置 promise 的 state 为 FULFILLED，value 为结果数组 values。

3.1.10 Promise.race

Promise.race 接收一个数组，在数组中有一个 promise 状态发生改变（pending -> fulfilled/rejected）时返回：

```
Promise.race = race
function race (iterable) {
  const self = this
  if (!isArray(iterable)) {
    return this.reject(new TypeError('must be an array'))
  }

  const len = iterable.length
  let called = false
  if (!len) {
    return this.resolve([])
  }

  let i = -1
  const promise = new this(INTERNAL)

  while (++i < len) {
    resolver(iterable[i])
  }
  return promise
```

书籍是人类进步的阶梯

```
    function resolver (value) {
      self.resolve(value).then(function (response) {
        if (!called) {
          called = true
          doResolve(promise, response)
        }
      }, function (error) {
        if (!called) {
          called = true
          doReject(promise, error)
        }
      })
    }
  }
```

Promise.race 与 Promise.all 的代码相近，只不过这里用 called 控制只要有任意一个 promise 状态改变，则立即设置返回的 promise 的状态和值。

至此，Promise 的实现全部讲解完毕。

3.1.11 代码解析

下面，我们通过对 10 段代码进行解析，来巩固在前面学到的 Promise 知识点。

代码一

```
const promise = new Promise((resolve, reject) => {
  console.log(1)
  resolve()
  console.log(2)
})
promise.then(() => {
  console.log(3)
})
console.log(4)
```

运行结果：

```
1
2
4
```

书籍是人类进步的阶梯

```
3
```

代码解析：Promise 构造函数是同步执行的，而 promise.then 中的函数是异步执行的。

代码二

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('success')
  }, 1000)
})
const promise2 = promise1.then(() => {
  throw new Error('error!!!')
})

console.log('promise1', promise1)
console.log('promise2', promise2)

setTimeout(() => {
  console.log('promise1', promise1)
  console.log('promise2', promise2)
}, 2000)
```

运行结果：

```
promise1 Promise { <pending> }
promise2 Promise { <pending> }
(node:50928) UnhandledPromiseRejectionWarning: Unhandled promise rejection
(rejection id: 1): Error: error!!!
(node:50928) [DEP0018] DeprecationWarning: Unhandled promise rejections are
deprecated. In the future, promise rejections that are not handled will
terminate the Node.js process with a non-zero exit code.
promise1 Promise { 'success' }
promise2 Promise {
  <rejected> Error: error!!!
    at promise.then (...)
    at <anonymous> }
```

代码解析：promise 有 3 种状态，即 pending、fulfilled 或 rejected。状态改变只能是 从 pending 到 fulfilled 或者从 pending 到 rejected，状态一旦改变则不能再变。上面的 promise2 并不是 promise1，而是返回的一个新的 Promise 实例。

书籍是人类进步的阶梯

代码三

```
const promise = new Promise((resolve, reject) => {
  resolve('success1')
  reject('error')
  resolve('success2')
})

promise
  .then((res) => {
    console.log('then: ', res)
  })
  .catch((err) => {
    console.log('catch: ', err)
  })
```

运行结果：

```
then: success1
```

代码解析：构造函数中的 resolve 或 reject 只有在第 1 次执行时有效，多次调用没有任何作用，再次印证了代码二的结论：promise 的状态一旦改变则不能再变。

代码四

```
Promise.resolve(1)
  .then((res) => {
    console.log(res)
    return 2
  })
  .catch((err) => {
    return 3
  })
  .then((res) => {
    console.log(res)
  })
```

运行结果：

```
1
2
```

代码解析：promise 可以链式调用。提起链式调用，我们通常会想到通过 return this

书籍是人类进步的阶梯

实现，不过 Promise 并不是这样实现的。promise 在每次调用 .then 或者 .catch 时都会返回一个新的 promise，从而实现链式调用。

代码五

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('once')
    resolve('success')
  }, 1000)
})

const start = Date.now()
promise.then((res) => {
  console.log(res, Date.now() - start)
})
promise.then((res) => {
  console.log(res, Date.now() - start)
})
```

运行结果：

```
once
success 1005
success 1007
```

代码解析：promise 的 .then 或者 .catch 可以被调用多次，但在这里 Promise 构造函数只执行一次。或者说，promise 的内部状态一旦改变并且有了一个值，则后续在每次调用 .then 或者 .catch 时都会直接拿到该值。

代码六

```
Promise.resolve()
  .then(() => {
    return new Error('error!!!')
  })
  .then((res) => {
    console.log('then: ', res)
  })
  .catch((err) => {
    console.log('catch: ', err)
  })
```

书籍是人类进步的阶梯

运行结果：

```
then: Error: error!!!
    at Promise.resolve.then (...)
    at ...
```

代码解析：在 .then 或者 .catch 中返回一个 error 对象并不会抛出错误，所以不会被后续的 .catch 捕获，需要改成 return Promise.reject(new Error('error!!!')) 或者 throw new Error('error!!!')。返回的任意一个非 promise 的值都会被包裹成 promise 对象，即 return new Error('error!!!') 等价于 return Promise.resolve(new Error('error!!!'))。

代码七

```
const promise = Promise.resolve()
  .then(() => {
    return promise
  })
promise.catch(console.error)
```

运行结果：

```
TypeError: Chaining cycle detected for promise #<Promise>
    at <anonymous>
    at process._tickCallback (internal/process/next_tick.js:188:7)
    at Function.Module.runMain (module.js:667:11)
    at startup (bootstrap_node.js:187:16)
    at bootstrap_node.js:607:3
```

代码解析：.then 或 .catch 返回的值不能是 promise 本身，否则会造成死循环。类似于：

```
process.nextTick(function tick () {
  console.log('tick')
  process.nextTick(tick)
})
```

代码八

```
Promise.resolve(1)
  .then(2)
  .then(Promise.resolve(3))
  .then(console.log)
```

书籍是人类进步的阶梯

运行结果：

```
1
```

代码解析：.then 或者 .catch 的参数期望是函数，传入非函数则会发生值穿透。

代码九

```
Promise.resolve()
  .then(function success (res) {
    throw new Error('error')
  }, function fail1 (e) {
    console.error('fail1: ', e)
  })
  .catch(function fail2 (e) {
    console.error('fail2: ', e)
  })
```

运行结果：

```
fail2: Error: error
    at success (...)
    at ...
```

代码解析：.then 可以接收两个参数，第 1 个是处理成功的函数，第 2 个是处理错误的函数。

.catch 是 .then 的第 2 个参数的简便写法，但是在用法上有一点需要注意，.then 的第 2 个处理错误的函数（fail1）捕获不了第 1 个处理成功的函数（success）抛出的错误，而后续的 .catch 方法（fail2）可以捕获之前的错误。当然，也可以采用以下代码：

```
Promise.resolve()
  .then(function success1 (res) {
    throw new Error('error')
  }, function fail1 (e) {
    console.error('fail1: ', e)
  })
  .then(function success2 (res) {
  }, function fail2 (e) {
    console.error('fail2: ', e)
  })
```

书籍是人类进步的阶梯

代码十

```
process.nextTick(() => {
  console.log('nextTick')
})
Promise.resolve()
  .then(() => {
    console.log('then')
  })
setImmediate(() => {
  console.log('setImmediate')
})
console.log('end')
```

运行结果：

```
end
nextTick
then
setImmediate
```

代码解析：process.nextTick 和 promise.then 都属于 microtask，而 setImmediate 属于 macrotask，在事件循环的 check 阶段执行。在事件循环的每个阶段（macrotask）之间都会执行 microtask，以上代码本身（macrotask）在执行完后会执行一次 microtask。

本节的参考链接如下。

- https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise
- https://promisesaplus.com/

3.2 Async + Await

笔者在很长一段时间内都在使用 koa@1 +(generator|bluebird)与 sequelize 这个组合，这个组合并没有什么问题，也很常见，但是在滥用时会导致后面的维护和调试很痛苦。现在，排除我们不得不用的模块 sequelize，从调试 cpuprofile 的角度讲讲为什么笔者认为应该用 async/await + Promise 替代 co + generator|bluebird。

书籍是人类进步的阶梯

笔者的观点是：使用原生模块具有更清晰的调用栈。

下面用 4 个例子进行对比，看看通过实现相同逻辑的不同代码生成的 cpuprofile 中的调用栈的信息。

3.2.1 例 1：async + await

async.js 的代码如下：

```
const fs = require('fs')
const profiler = require('v8-profiler')

async function A () {
  return await Promise.resolve('A')
}

async function B () {
  return await A()
}

(async function asyncWrap () {
  const start = Date.now()
  profiler.startProfiling()
  while (Date.now() - start < 10000) {
    await B()
  }
  const profile = profiler.stopProfiling()
  profile.export()
    .pipe(fs.createWriteStream('async.cpuprofile'))
    .on('finish', () => {
      profile.delete()
      console.error('async.cpuprofile export success')
    })
})()
```

加载运行后生成的 async.cpuprofile，如图 3-1 所示。

Self Time		Total Time		Function	
799.9 ms	8.00 %	799.9 ms	8.00 %	(program)	
281.3 ms	2.81 %	281.3 ms	2.81 %	(garbage collector)	
0 ms	0 %	8918.7 ms	89.19 %	▼(anonymous)	bootstrap_node.js:10
0 ms	0 %	8918.7 ms	89.19 %	▼startup	bootstrap_node.js:12
0 ms	0 %	8918.7 ms	89.19 %	▼Module.runMain	module.js:662
646.7 ms	6.47 %	8918.7 ms	89.19 %	▼_tickCallback	internal/proces...xt_tick.js:181
2595.8 ms	25.96 %	8272.0 ms	82.72 %	▼runMicrotasks	
862.7 ms	8.63 %	869.0 ms	8.69 %	▶then	
538.7 ms	5.39 %	4607.2 ms	46.07 %	▼(anonymous)	
707.0 ms	7.07 %	4268.5 ms	42.68 %	▼(anonymous)	
894.1 ms	8.94 %	3034.0 ms	30.34 %	▼asyncWrap	/Users/nswbmw/D...st/async.js:12
568.9 ms	5.69 %	568.9 ms	5.59 %	(anonymous)	
0 ms	0 %	1.3 ms	0.01 %	▶stopProfiling	/Users/nswbmw/D...rofiler.js:218
0 ms	0 %	1569.8 ms	15.70 %	▼B	/Users/nswbmw/D...est/async.js:8
707.0 ms	7.07 %	707.0 ms	7.07 %	(anonymous)	
0 ms	0 %	862.7 ms	8.63 %	▼A	/Users/nswbmw/D...est/async.js:0
693.2 ms	6.93 %	693.2 ms	6.93 %	(anonymous)	
169.5 ms	1.70 %	169.5 ms	1.70 %	resolve	
74.1 ms	0.74 %	150.7 ms	1.51 %	▶A	/Users/nswbmw/D...est/async.js:4
71.6 ms	0.72 %	376.7 ms	3.77 %	▶B	/Users/nswbmw/D...est/async.js:8
0 ms	0 %	0 ms	0 %	▶Module._load	module.js:442

图 3-1

可以看出，在 asyncWrap 中调用了 B 函数，B 函数调用了 A 函数，在 A 函数中 resolve 了一个值，在 asyncWrap 中还调用了 stopProfiling 函数。

3.2.2 例 2：co + yield

co.js 的代码如下：

```
const fs = require('fs')
const co = require('co')
const profiler = require('v8-profiler')

function * A () {
  return yield Promise.resolve('A')
}

function * B () {
  return yield A()
}

co(function * coWrap () {
  const start = Date.now()
  profiler.startProfiling()
  while (Date.now() - start < 10000) {
    yield B()
  }
  const profile = profiler.stopProfiling()
  profile.export()
    .pipe(fs.createWriteStream('co.cpuprofile'))
    .on('finish', () => {
```

```
      profile.delete()
      console.error('co.cpuprofile export success')
    })
})
```

加载运行后生成的 co.cpuprofile，如图 3-2 所示。

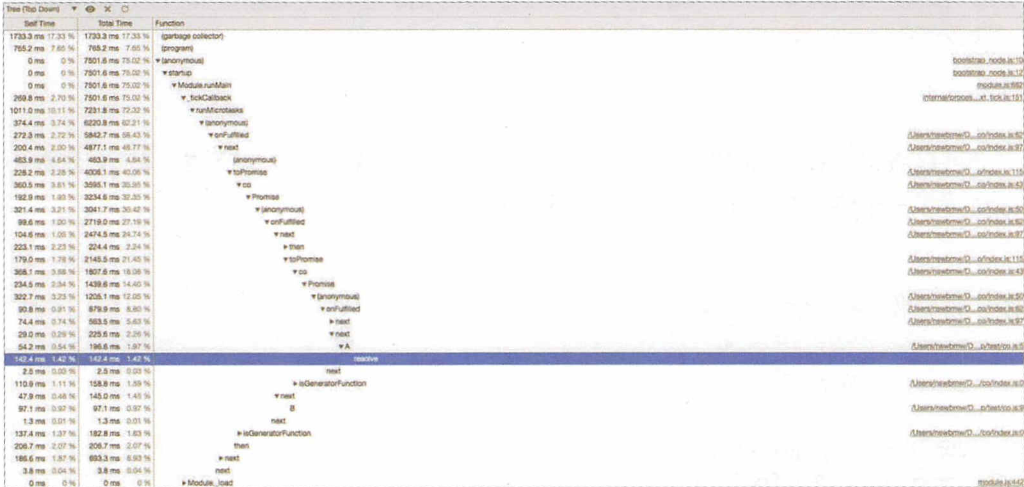


图 3-2

可以看出，调用栈非常深，有太多没有用的 co 相关的调用栈。如果有 n 个 generator 层层嵌套，就会出现 n 倍的 (anonymous)→onFullfiled→next→toPromise→co→Promise→(anonymous) 调用栈。如果你读过 co 的源码，就可能知道，这是 co 将 generator 解包的过程。其实这个可以通过将 yield generator 替换成 yield* generator 来优化。

3.2.3 例 3：co + yield*

co_better.js 的代码如下：

```
const fs = require('fs')
const co = require('co')
const profiler = require('v8-profiler')

function * A () {
  return yield Promise.resolve('A')
```

```
}

function * B () {
  return yield * A()
}

co(function * coWrap () {
  const start = Date.now()
  profiler.startProfiling()
  while (Date.now() - start < 10000) {
    yield * B()
  }
  const profile = profiler.stopProfiling()
  profile.export()
    .pipe(fs.createWriteStream('co_better.cpuprofile'))
    .on('finish', () => {
      profile.delete()
      console.error('co_better.cpuprofile export success')
    })
})
```

加载运行后生成的 co_better.cpuprofile，如图 3-3 所示。

Tree (Top Down)

Self Time		Total Time		Function	
923.7 ms	9.24 %	923.7 ms	9.24 %	(program)	
293.7 ms	2.94 %	293.7 ms	2.94 %	(garbage collector)	
0 ms	0 %	8782.6 ms	87.83 %	▼(anonymous)	bootstrap_node.js:10
0 ms	0 %	8782.6 ms	87.83 %	▼startup	bootstrap_node.js:12
0 ms	0 %	8782.6 ms	87.83 %	▼Module.runMain	module.js:662
496.0 ms	4.96 %	8782.6 ms	87.83 %	▼_tickCallback	internal/proces...xt_tick.js:151
1335.6 ms	13.36 %	8266.6 ms	82.67 %	▼runMicrotasks	
304.7 ms	3.05 %	6951.0 ms	69.51 %	▼(anonymous)	
191.3 ms	1.91 %	6641.5 ms	66.41 %	▼onFulfilled	/Users/nswbmw/D...co/index.js:82
141.4 ms	1.41 %	4805.0 ms	48.05 %	▼next	
2516.9 ms	25.19 %	4653.9 ms	46.54 %	▼coWrap	/Users/nswbmw/D...o_better.js:13
414.3 ms	4.14 %	2135.0 ms	21.35 %	▼next	
766.5 ms	7.67 %	1706.1 ms	17.06 %	▼B	/Users/nswbmw/D...co_better.js:9
316.8 ms	3.17 %	939.6 ms	9.40 %	▼next	
270.5 ms	2.71 %	622.7 ms	6.23 %	▼A	/Users/nswbmw/D...co_better.js:5
352.2 ms	3.52 %	352.2 ms	3.52 %	resolve	
14.6 ms	0.15 %	14.6 ms	0.15 %	next	
9.7 ms	0.10 %	9.7 ms	0.10 %	next	
1.2 ms	0.01 %	1645.1 ms	16.45 %	►next	/Users/nswbmw/D...co/index.js:97
4.9 ms	0.05 %	4.9 ms	0.05 %	next	
0 ms	0 %	0 ms	0 %	►Module._load	module.js:442

图 3-3

可以看出，与 co.js 相比，调用栈清晰了很多，不过与使用 async/await 相比，还是多了些 onFulfilled、next。

3.2.4 例 4：co + bluebird

co_bluebird.js 的代码如下：

```
const fs = require('fs')
const co = require('co')
const Promise = require('bluebird')
const profiler = require('v8-profiler')

function * A () {
  return yield Promise.resolve('A')
}

function * B () {
  return yield * A()
}

co(function * coBluebirdWrap () {
  const start = Date.now()
  profiler.startProfiling()
  while (Date.now() - start < 10000) {
    yield * B()
  }
  const profile = profiler.stopProfiling()
  profile.export()
    .pipe(fs.createWriteStream('co_bluebird.cpuprofile'))
    .on('finish', () => {
      profile.delete()
      console.error('co_bluebird.cpuprofile export success')
    })
})
```

加载运行后生成的 co_bluebird.cpuprofile，如图 3-4 所示。

图 3-4

可以看出，与 co_better.js 相比，在调用栈中多了许多 bluebird 模块的无用信息。而且这只是非常简单的示例代码，若在复杂的业务逻辑中大量使用 bluebird 代码生成的 cpuprofile，就很复杂了。

结论：在使用 async/await + Promise + 命名函数时会有更清晰的调用栈，在分析 cpuprofile 时更加顺畅。

聪明的你可能会有以下问题，这里一一对这些问题进行解答。

（1）为什么不建议用 bluebird？

答：

- 随着 V8 的不断优化，原生 Promise 的性能逐渐提高，bluebird 的性能优势不明显；
- 原生 Promise 的 API 足够用，至少能覆盖大部分使用场景，而且在不断完善，未来还会添加新的 API，例如 Promise.prototype.finally；
- 具有更清晰的调用栈。

（2）由于历史遗留原因，在现在的代码中大量使用了 yield + generator，该怎么办?

答：

- 将所有 yield generator 替换成 yield * generator；
- 升级到 node@8+，逐步用 async/await 替换，毕竟 async 函数在调用后返回的也是一个 promise，也是 yieldable 的。

（3）async 和 co 的性能比较如何呢?

答：在 node@8+ 下 async/await 完胜 co。

3.2.5 从 yield 转为 yield* 遇到的坑

上面讲到，可以将 yield generator 转成 yield * generator，这里面有一个坑，是由于不明白 co 的原理而滥用 co 导致的。代码如下：

```
const co = require('co')
```

仅供非商业用途或交流学习使用


```
function * genFunc () {
  return Promise.resolve('genFunc')
}

co(function * () {
  console.log(yield genFunc()) // => genFunc
  console.log(yield * genFunc()) // => Promise { 'genFunc' }
})
```

可以看出，genFunc 这个 generatorFunction 在执行后会返回一个 promise，在使用 yield genFunc() 时，若 co 判断返回了一个 promise，则会继续帮我们调用它的 then，从而得到真正的字符串。如果使用 yield * genFunc()，就用到了语言原生的特性而不经过 co，直接返回一个 promise。

解决方法（任选其一）如下。

- 将 function * genFunc 改为 function genFunc，用 yield genFunc()。
- 将 return Promise.resolve('genFunc') 改为 return yield Promise.resolve('genFunc')，用 yield* genFunc()。

不过，建议最终转换为 async/await + Promise，毕竟 co + generator 只是一个过渡产物。

3.2.6 async + bluebird

如果是使用 async/await + bluebird 的情况呢？ async_bluebird.js 的代码如下：

```
const fs = require('fs')
const profiler = require('v8-profiler')
const Promise = require('bluebird')

async function A () {
  return await Promise.resolve('A')
}

async function B () {
  return await A()
}

(async function asyncBluebirdWrap () {
  const start = Date.now()
```

仅供非商业用途或交流学习使用


```
  profiler.startProfiling()
  while (Date.now() - start < 10000) {
    await B()
  }
  const profile = profiler.stopProfiling()
  profile.export()
    .pipe(fs.createWriteStream('async_bluebird.cpuprofile'))
    .on('finish', () => {
      profile.delete()
      console.error('async_bluebird.cpuprofile export success')
    })
})()
```

结论：其调用栈比 co_blueblird.js 的调用栈还乱。

本节的参考链接如下。

- https://medium.com/@markherhold/generators-vs-async-await-performance-806d8375a01a

3.3 Error Stack

Node.js 内置的 Error 类型如下。

- Error：通用的错误类型，例如 new Error('error!!!')。
- SyntaxError：语法错误，例如 require('vm').runInThisContext('binary ! isNotOk')。
- ReferenceError：引用错误，比如引用一个未定义的变量，例如 doesNotExist。
- TypeError：类型错误，例如 require('url').parse(() => {})。
- URIError：全局的 URI 处理函数抛出的错误，例如 encodeURI('\uD800')。
- AssertError：在使用 assert 模块时抛出的错误，例如 assert(false)。

每个 Error 对象都通常有 name、message、stack、constructor 等属性。当程序抛出异常时，我们需要根据错误栈（error.stack）定位到出错的代码。希望本节能够帮助读者理解并玩转错误栈，写出错误栈清晰的代码，方便调试。

仅供非商业用途或交流学习使用

3.3.1 Stack Trace

错误栈在本质上就是调用栈（或者叫作堆栈追踪）。在 JavaScript 中，调用栈指每当有一个函数调用出现，就会将该函数压入栈顶，在调用结束时再将其从栈顶移出。

来看一段代码：

```
function c () {
  console.log('c')
  console.trace()
}

function b () {
  console.log('b')
  c()
}

function a () {
  console.log('a')
  b()
}

a()
```

执行后打印出：

```
a
b
c
Trace
    at c (/Users/nswbmw/Desktop/test/app.js:3:11)
    at b (/Users/nswbmw/Desktop/test/app.js:8:3)
    at a (/Users/nswbmw/Desktop/test/app.js:13:3)
    at Object.<anonymous> (/Users/nswbmw/Desktop/test/app.js:16:1)
    at ...
```

可以看出，在 c 函数中 console.trace() 打印出的堆栈追踪依次为 c、b、a，即 a 调用了 b，b 调用了 c。

稍微修改下上面的例子：

仅供非商业用途或交流学习使用


```
function c () {
  console.log('c')
}

function b () {
  console.log('b')
  c()
  console.trace()
}

function a () {
  console.log('a')
  b()
}

a()
```

在执行后打印出：

```
a
b
c
Trace
    at b (/Users/nswbmw/Desktop/test/app.js:8:11)
    at a (/Users/nswbmw/Desktop/test/app.js:13:3)
    at Object.<anonymous> (/Users/nswbmw/Desktop/test/app.js:16:1)
    at ...
```

可以看出，c() 在 console.trace() 之前执行完毕，从栈中移除，所以在栈中从上往下为 b、a。

以上示例的代码过于简单，在实际情况下错误栈并没有这么直观。以常用的 mongoose 为例，mongoose 的错误栈并不友好：

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema
mongoose.connect('mongodb://localhost/test')

const UserSchema = new Schema({
  id: mongoose.Schema.Types.ObjectId
})
const User = mongoose.model('User', UserSchema)
```

仅供非商业用途或交流学习使用


```
User
  .create({ id: 'xxx' })
  .then(console.log)
  .catch(console.error)
```

运行后打印出：

```
{ ValidationError: User validation failed: id: Cast to ObjectID failed for
value "xxx" at path "id"
    at ValidationError.inspect (/Users/nswbmw/Desktop/test/node_modules/
mongoose/lib/error/validation.js:56:24)
    at ...
  errors:
   { id:
      { CastError: Cast to ObjectID failed for value "xxx" at path "id"
    at new CastError (/Users/nswbmw/Desktop/test/node_modules/mongoose/lib/
error/cast.js:27:11)
    at model.$set (/Users/nswbmw/Desktop/test/node_modules/mongoose/lib/
document.js:792:7)
    at ...
        message: 'Cast to ObjectID failed for value "xxx" at path "id"',
        name: 'CastError',
        stringValue: '"xxx"',
        kind: 'ObjectID',
        value: 'xxx',
        path: 'id',
        reason: [Object] } },
  _message: 'User validation failed',
  name: 'ValidationError' }
```

从 mongoose 给出的 error.stack 中看不到任何有用的信息，error.message 告诉我们“xxx”不匹配 User 这个 Model 的 id（ObjectID）类型，其他字段基本上也是这个结论的补充，却没有给出我们最关心的问题：在我们写的代码中，到底哪一行出了问题?

如何解决这个问题呢? 我们先看看 Error.captureStackTrace 的用法。

3.3.2 Error.captureStackTrace

Error.captureStackTrace 是 V8 提供的一个 API，可以传入两个参数，如下所示：

```
Error.captureStackTrace(targetObject[, constructorOpt])
```

仅供非商业用途或交流学习使用

Error.captureStackTrace 会在 targetObject 中添加一个 stack 属性，在对该属性进行访问时，将以字符串的形式返回 Error.captureStackTrace() 语句被调用时的代码位置信息（即调用栈历史）。

举个简单的例子：

```
const myObject = {}
Error.captureStackTrace(myObject)
console.log(myObject.stack)
// 输出
Error
    at Object.<anonymous> (/Users/nswbmw/Desktop/test/app.js:2:7)
    at ...
```

除了 targetObject，captureStackTrace 还接收了一个类型为 function 的可选参数 constructorOpt，在传递该参数时，在调用栈中所有 constructorOpt 函数之上的信息（包括 constructorOpt 函数自身），都会在访问 targetObject.stack 时被忽略。当需要对终端用户隐藏内部的实现细节时，constructorOpt 参数会很有用。传入的第 2 个参数通常用于自定义错误，例如：

```
function MyError() {
  Error.captureStackTrace(this, MyError)
  this.name = this.constructor.name
  this.message = 'you got MyError'
}

const myError = new MyError()
console.log(myError)
console.log(myError.stack)
// 输出
MyError { name: 'MyError', message: 'you got MyError' }
Error
    at Object.<anonymous> (/Users/nswbmw/Desktop/test/app.js:7:17)
    at ...
```

如果去掉 captureStackTrace 的第 2 个参数，则出现了 MyError 相关的调用栈，但我们并不关心 MyError 及其内部是如何实现的：

```
function MyError() {
  Error.captureStackTrace(this)
```

仅供非商业用途或交流学习使用


```
    this.name = this.constructor.name
    this.message = 'you got MyError'
  }

  const myError = new MyError()
  console.log(myError)
  console.log(myError.stack)
  // 输出
  MyError { name: 'MyError', message: 'you got MyError' }
  Error
      at new MyError (/Users/nswbmw/Desktop/test/app.js:2:9)
      at Object.<anonymous> (/Users/nswbmw/Desktop/test/app.js:7:17)
      at ...
```

captureStackTrace 的第 2 个参数可以传入调用链上的其他函数，不一定是当前函数，例如：

```
const myObj = {}

function c () {
  Error.captureStackTrace(myObj, b)
}

function b () {
  c()
}

function a () {
  b()
}

a()
console.log(myObj.stack)
// 输出
Error
    at a (/Users/nswbmw/Desktop/test/app.js:12:3)
    at Object.<anonymous> (/Users/nswbmw/Desktop/test/app.js:15:1)
    at ...
```

可以看出，captureStackTrace 的第 2 个参数传入了函数 b，在调用栈中隐藏了函数 b 及其以上的所有堆栈帧。

仅供非商业用途或交流学习使用

讲到这里，相信读者都明白了 captureStackTrace 的用法。但这具体有什么用呢？其实在上面提到了：隐藏内部的实现细节，优化错误栈。

下面以笔者写的一个模块 Mongolass 为例，讲解如何应用 captureStackTrace。

3.3.3 captureStackTrace 在 Mongolass 中的应用

这里先大体讲讲 Mongolass 的用法。Mongolass 与 Mongoose 类似，有 Model 的概念。在 Model 上挂载的方法与对 MongoDB 的 collections 的操作对应，例如 User.insert。User 是一个 Model 实例，User.insert 方法返回的是一个 Query 实例。Query 的代码如下：

```
class Query {
  constructor(op, args) {
    Error.captureStackTrace(this, this.constructor);
    ...
  }
}
```

这里用 Error.captureStackTrace 隐藏了 Query 内部的错误栈细节，但这样带来一个问题：丢失了原来的 error.stack，在 Mongolass 中可以自定义插件，而插件函数的执行是在 Query 内部的，假如在插件中抛出错误，则会丢失相关的错误栈信息。如何弥补呢？Mongolass 的做法是：当在 Query 内部抛出错误（error）时，截取有用的 error.stack，然后拼接到 Query 实例通过 Error.captureStackTrace 生成的 stack 上。

来看一段 Mongolass 代码：

```
const Mongolass = require('mongolass')
const Schema = Mongolass.Schema
const mongolass = new Mongolass('mongodb://localhost:27017/test')

const UserSchema = new Schema('UserSchema', {
  name: { type: 'string' },
  age: { type: 'number' }
})
const User = mongolass.model('User', UserSchema)

User
  .insertOne({ name: 'nswbmw', age: 'wrong age' })
  .exec()
```

仅供非商业用途或交流学习使用


```
  .then(console.log)
  .catch(console.error)
```

运行后打印的错误信息如下：

```
{ TypeError: ($.age: "wrong age") ✖ (type: number)
    at Model.insertOne (/Users/nswbmw/Desktop/test/node_modules/mongolass/
lib/query.js:104:16)
    at Object.<anonymous> (/Users/nswbmw/Desktop/test/app.js:12:4)
    at ...
  validator: 'type',
  actual: 'wrong age',
  expected: { type: 'number' },
  path: '$.age',
  schema: 'UserSchema',
  model: 'User',
  op: 'insertOne',
  args: [ { name: 'nswbmw', age: 'wrong age' } ],
  pluginName: 'MongolassSchema',
  pluginOp: 'beforeInsertOne',
  pluginArgs: [] }
```

可以看出，app.js 第 12 行的 insertOne 报错，报错的原因是 age 字段是字符串 “wrong age”，而我们期望的是 number 类型的值。

3.3.4 Error.prepareStackTrace

V8 暴露了另外一个接口——Error.prepareStackTrace。简单来讲，该接口的作用就是定制 stack。其用法如下：

```
Error.prepareStackTrace(error, structuredStackTrace)
```

第 1 个参数是一个 Error 对象，第 2 个参数是一个数组，每一项都是一个 CallSite 对象，包含错误的函数名、行数等信息。下面对比两种代码。

正常的 throw error 代码如下：

```
function c () {
  throw new Error('error!!!')
}
```

```
function b () {
  c()
}

function a () {
  b()
}

try {
  a()
} catch (e) {
  console.log(e.stack)
}
// 输出
Error: error!!!
    at c (/Users/nswbmw/Desktop/test/app.js:2:9)
    at b (/Users/nswbmw/Desktop/test/app.js:6:3)
    at a (/Users/nswbmw/Desktop/test/app.js:10:3)
    at Object.<anonymous> (/Users/nswbmw/Desktop/test/app.js:14:3)
    at ...
```

使用 Error.prepareStackTrace 格式化 stack，代码如下：

```
Error.prepareStackTrace = function (error, callSites) {
  return error.toString() + '\n' + callSites.map(callSite => {
    return '    -> ' + callSite.getFunctionName() + ' ('
      + callSite.getFileName() + ':'
      + callSite.getLineNumber() + ':'
      + callSite.getColumnNumber() + ')'
  }).join('\n')
}

function c () {
  throw new Error('error!!!')
}

function b () {
  c()
}

function a () {
  b()
```

书籍是人类进步的阶梯

```
}

try {
  a()
} catch (e) {
  console.log(e.stack)
}
// 输出
Error: error!!!
    -> c (/Users/nswbmw/Desktop/test/app.js:11:9)
    -> b (/Users/nswbmw/Desktop/test/app.js:15:3)
    -> a (/Users/nswbmw/Desktop/test/app.js:19:3)
    -> null (/Users/nswbmw/Desktop/test/app.js:23:3)
    -> ...
```

可以看出，我们自定义了一个 Error.prepareStackTrace，用其格式化并打印出了 stack。

CallSite 对象还有许多 API，例如：getThis、getTypeName、getFunction、getFunctionName、getMethodName、getFileName、getLineNumber、getColumnNumber、getEvalOrigin、isToplevel、isEval、isNative 和 isConstructor，这里不一一介绍了，有兴趣的读者可查看参考链接。

在使用 Error.prepareStackTrace 时需要注意以下两点。

- 这个方法是 V8 暴露出来的，所以只能在基于 V8 的 Node.js 或者 Chrome 里使用。
- 这个方法会修改全局 Error 的行为。

3.3.5 Error.prepareStackTrace 的其他用法

Error.prepareStackTrace 除格式化错误栈外还有什么作用呢？ sindresorhus 大神还写了一个 callsites 模块，可以用来获取函数调用相关的信息，例如获取执行的函数所在的文件名：

```
const callsites = require('callsites')

function getFileName() {
  console.log(callsites()[0].getFileName())
  //=> '/Users/nswbmw/Desktop/test/app.js'
```

书籍是人类进步的阶梯

```
}

getFileName()
```

来看看源码：

```
module.exports = () => {
  const _ = Error.prepareStackTrace
  Error.prepareStackTrace = (_, stack) => stack
  const stack = new Error().stack.slice(1)
  Error.prepareStackTrace = _
  return stack
}
```

请注意以下几点。

- 因为修改 Error.prepareStackTrace 会全局生效，所以将原来的 Error.prepareStackTrace 存到一个变量中，在函数执行完后再重置回去，避免影响全局的 Error。
- Error.prepareStackTrace 函数直接返回 CallSite 对象数组，而不是格式化后的 stack 字符串。
- new 一个 Error，stack 是返回的 CallSite 对象数组，因为第 1 项是 callsites，它总是这个模块的 CallSite，所以通过 slice(1) 去掉。

假如我们想获取当前函数的父函数名，则可以这样用：

```
const callsites = require('callsites')

function b () {
  console.log(callsites()[1].getFunctionName())
  // => 'a'
}

function a () {
  b()
}
a()
```

书籍是人类进步的阶梯

3.3.6 Error.stackTraceLimit

Node.js 还暴露了一个 Error.stackTraceLimit 的设置，可以通过设置这个值来改变输出的 stack 的行数，默认值是 10。

3.3.7 Long Stack Trace

stack trace 也有短板，问题出在异步操作上。若在异步回调中抛出错误，就会丢失绑定回调前的调用栈信息，来看个例子：

```
const foo = function () {
  throw new Error('error!!!')
}
const bar = function () {
  setTimeout(foo)
}
bar()
// 输出
/Users/nswbmw/Desktop/test/app.js:2
  throw new Error('error!!!')
  ^

Error: error!!!
    at Timeout.foo [as _onTimeout] (/Users/nswbmw/Desktop/test/app.js:2:9)
    at ontimeout (timers.js:469:11)
    at tryOnTimeout (timers.js:304:5)
    at Timer.listOnTimeout (timers.js:264:5)
```

通过该例可以看出，其中丢失了 bar 的调用栈。

在实际开发过程中，异步回调的例子数不胜数，如果不能知道异步回调之前的触发位置，则会给调试带来很大的难度。这时，出现了一个 long Stack Trace 的概念。

long Stack Trace 并不是 JavaScript 原生就支持的功能，所以要拥有这样的功能，就需要我们做一些 hack。幸好在 V8 环境下已经提供了所有 hack 所需的 API。

对于异步回调，目前能做的就是在所有会产生异步操作的 API 上做一些手脚，这些 API 包括：

书籍是人类进步的阶梯

（1）setTimeout、setInterval、setImmediate；

（2）nextTick、nextDomainTick；

（3）EventEmitter.addEventListener；

（4）EventEmitter.on；

（5）Ajax XHR。

Long Stack Trace 相关的库可以参考：

- https://github.com/AndreasMadsen/trace
- https://github.com/mattinsler/longjohn
- https://github.com/tlrobinson/long-stack-traces

node@8+ 提供了强大的 async_hooks 模块，本书会在后面的章节中介绍如何使用 async_hooks 模块。

本节的参考链接如下。

- https://zhuanlan.zhihu.com/p/25338849
- https://segmentfault.com/a/1190000007076507
- https://github.com/v8/v8/wiki/Stack-Trace-API
- https://www.jianshu.com/p/1d5120ad62bb

3.4　node@8

如果你想以最简单的方式提升 Node.js 程序的性能，那就升级到 node@8+ 吧。这不是一个玩笑，多少 JavaScript 前辈们以血的教训总结出了一长列 Optimization killer，典型的如下。

- 在 try 里面不要写过多的代码，V8 无法优化，最好将这些代码放到一个函数里，然后在 try 里执行这个函数。

书籍是人类进步的阶梯

- 少用 delete。
- 少用 arguments。

然而，随着 V8 彻底采用了新的 JIT 编译器—— Turbofan，大多数 Optimization killers 都成了过去时，所以在本节中我们来看看过去哪些常见的 Optimization killers 可以被 V8 优化。

3.4.1 Ignition + Turbofan

之前 V8 使用的是名为 Crankshaft 的编译器，这个编译器后来逐渐暴露出一些缺点，如下所述。

- 不能优化所有现代语言特性（如 try-catch、for-of、generators、async/await 等）。
- 默认逆优化（性能断崖、逆优化循环）。
- 有限的优化潜力、有限的静态分析等。

而引入 Turbofan 的好处如下。

- 对全部的 ESNext 语言特性的支持（例如 try-catch、class、generators、async/await、模块和解构赋值等）。
- 没有逆优化循环，只有在真正有益的情况下才会进行逆优化。
- 性能提升等。

Ignition 是 V8 新引入的解释器，用来将代码编译成简洁的字节码，而不是之前的机器码，这大大减少了结果代码，减少了系统的内存使用。由于字节码较小，所以可以编译全部源码，而不用避免编译未使用的代码。也就是说，脚本只需要解析一次，而不是像之前的编译过程那样解析多次。

Ignition 与 TurboFan 的关系为：Ignition 解释器使用低级的体系结构无关的 TurboFan 宏汇编指令为每个操作码生成字节码处理程序，TurboFan 将这些指令编译成目标平台的代码，并在这个过程中执行低级的指令选择和机器寄存器分配。

补充一点，之前的 V8 将代码编译成机器码执行，而新的 V8 将代码编译成字节码解释执行，动机是什么呢？可能如下。

书籍是人类进步的阶梯

- 减少机器码占用的内存空间，即牺牲时间换空间（主要动机）；
- 加快代码的启动速度；
- 对 V8 的代码进行重构，降低 V8 的代码复杂度。

3.4.2 版本的对应关系

Node.js 版本、V8 版本与 V8 编译器的对应关系如下：

```
node@6         -> V8@5.1 -> Crankshaft
node@8.0-8.2 -> V8@5.8 -> Crankshaft + Turbofan
                V8@5.9 -> Turbofan
node@8.3-8.4 -> V8@6.0 -> Turbofan
```

3.4.3 try/catch

著名的去优化操作是使用 try/catch 代码块。下面通过 4 种场景比较在不同的 V8 版本下执行的效率：

```
var benchmark = require('benchmark')
var suite = new benchmark.Suite()

function sum (base, max) {
  var total = 0

  for (var i = base; i < max; i++) {
    total += i
  }
}

suite.add('sum with try catch', function sumTryCatch () {
  try {
    var base = 0
    var max = 65535

    var total = 0

    for (var i = base; i < max; i++) {
      total += i
    }
  } catch (err) {
```

书籍是人类进步的阶梯

```
    console.log(err.message)
  }
})

suite.add('sum without try catch', function noTryCatch () {
  var base = 0
  var max = 65535

  var total = 0

  for (var i = base; i < max; i++) {
    total += i
  }
})

suite.add('sum wrapped', function wrapped () {
  var base = 0
  var max = 65535

  try {
    sum(base, max)
  } catch (err) {
    console.log(err.message)
  }
})

suite.add('sum function', function func () {
  var base = 0
  var max = 65535

  sum(base, max)
})

suite.on('complete', require('./print'))
suite.run()
```

运行结果如图 3-5 所示。

书籍是人类进步的阶梯

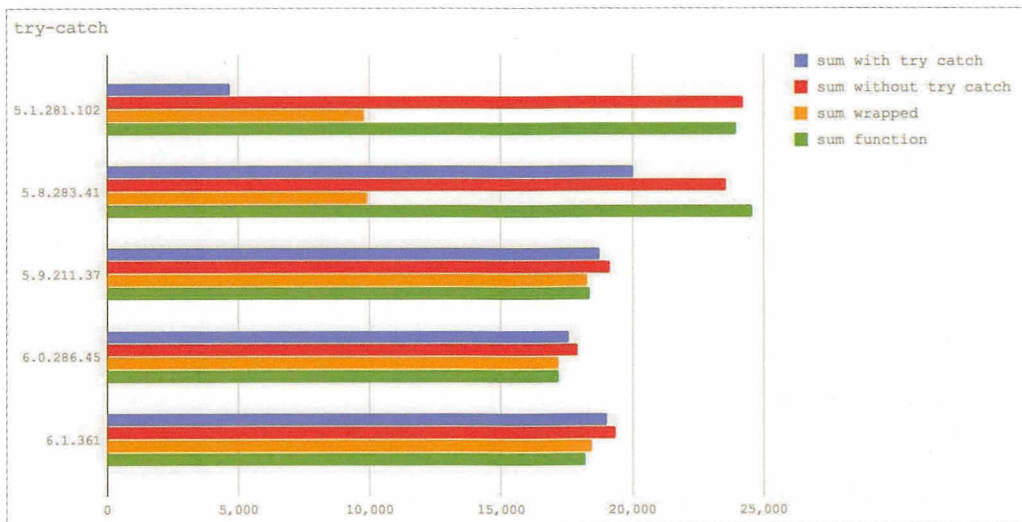


图 3-5

结论：在 node@8.3 及以上版本中，在 try 块内写代码的性能损耗可以忽略不计。

3.4.4 delete

多年以来，delete 对于任何希望编写高性能 JavaScript 的人来说都是受限制的，我们通常用赋值 undefined 替代。delete 的问题可归结为 V8 处理 JavaScript 对象的动态特性和原型链方式，使得属性查找在实现上变得复杂。下面通过 3 种场景比较在不同的 V8 版本下执行的效率：

```
var benchmark = require('benchmark')
var suite = new benchmark.Suite()

function MyClass (x, y) {
  this.x = x
  this.y = y
}

function MyClassLast (x, y) {
  this.y = y
  this.x = x
}
```

书籍是人类进步的阶梯

```
suite.add('setting to undefined', function undefProp () {
  var obj = new MyClass(2, 3)
  obj.x = undefined

  JSON.stringify(obj)
})

suite.add('delete', function deleteProp () {
  var obj = new MyClass(2, 3)
  delete obj.x

  JSON.stringify(obj)
})

suite.add('delete last property', function deleteProp () {
  var obj = new MyClassLast(2, 3)
  delete obj.x

  JSON.stringify(obj)
})

suite.add('setting to undefined literal', function undefPropLit () {
  var obj = { x: 2, y: 3 }
  obj.x = undefined

  JSON.stringify(obj)
})

suite.add('delete property literal', function deletePropLit () {
  var obj = { x: 2, y: 3 }
  delete obj.x

  JSON.stringify(obj)
})

suite.add('delete last property literal', function deletePropLit () {
  var obj = { y: 3, x: 2 }
  delete obj.x

  JSON.stringify(obj)
})
```

书籍是人类进步的阶梯

```
suite.on('complete', require('./print'))
suite.run()
```

运行结果如图 3-6 所示。

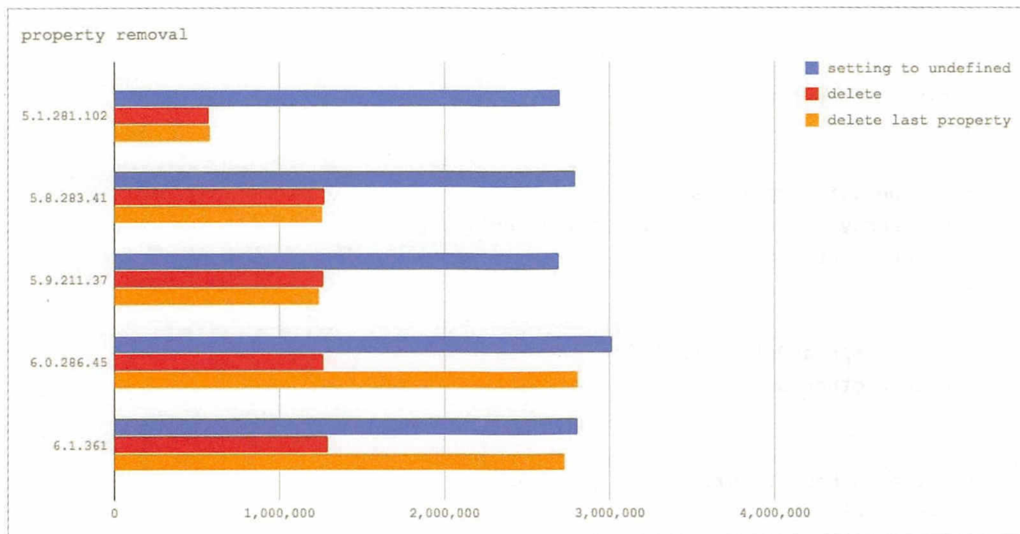


图 3-6

结论：在 node@8 及以上版本中 delete 一个对象上的属性比 node@6 快了一倍。在 node@8.3 及以上版本中 delete 一个对象上的最后一个属性几乎与赋值 undefined 同样快。

3.4.5 arguments

我们知道 arguments 是个类数组，所以我们通常使用 Array.prototype.slice.call(arguments) 将它转化成数组再使用，这样会有一定的性能损耗。下面通过 4 种场景比较在不同的 V8 版本下执行的效率：

```
var benchmark = require('benchmark')
var suite = new benchmark.Suite()

function leakyArguments () {
  return other(arguments)
}
```

书籍是人类进步的阶梯

```
function copyArgs () {
  var array = new Array(arguments.length)

  for (var i = 0; i < array.length; i++) {
    array[i] = arguments[i]
  }

  return other(array)
}

function sliceArguments () {
  var array = Array.prototype.slice.apply(arguments)
  return other(array)
}

function spreadOp(...args) {
  return other(args)
}

function other (toSum) {
  var total = 0
  for (var i = 0; i < toSum.length; i++) {
    total += toSum[i]
  }
  return total
}

suite.add('leaky arguments', () => {
  leakyArguments(1, 2, 3)
})

suite.add('Array.prototype.slice arguments', () => {
  sliceArguments(1, 2, 3)
})

suite.add('for-loop copy arguments', () => {
  copyArgs(1, 2, 3)
})

suite.add('spread operator', () => {
  spreadOp(1, 2, 3)
})
```

书籍是人类进步的阶梯

```
suite.on('complete', require('./print'))
suite.run()
```

运行结果如图 3-7 所示。

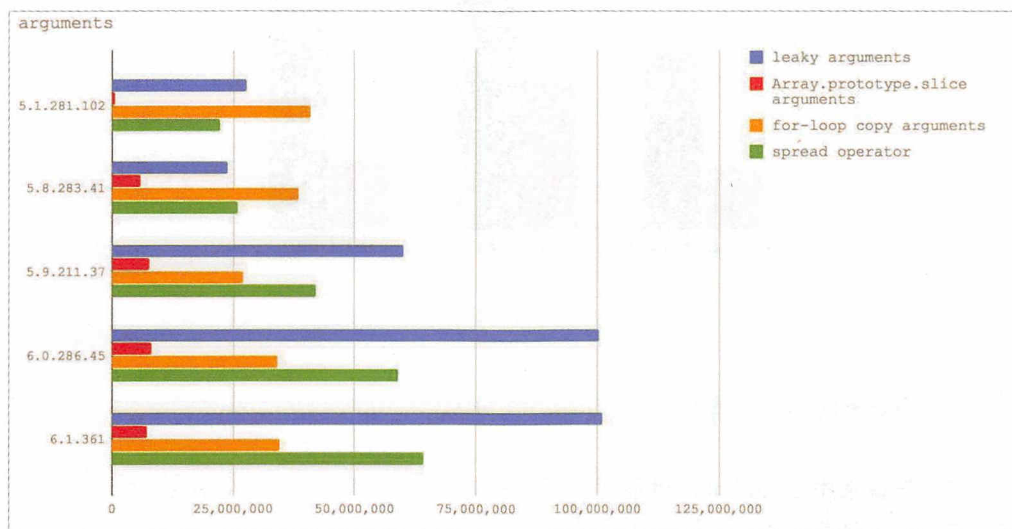


图 3-7

结论：在 node@8.3 及以上版本中使用对象展开运算符是除直接使用 arguments 外最快的方案；对于 node@8.2 及以下版本，我们应该使用一个 for 循环将 key 从 arguments 复制到一个新的（预先分配的）数组中。总之，是时候抛弃 Array.prototype.slice.call 了。

3.4.6 async 性能提升

在 V8@5.7 发布后，原生的 async 函数与 Promise 一样快了，同时，Promise 的性能比 V8@5.6 快了一倍，如图 3-8 所示。

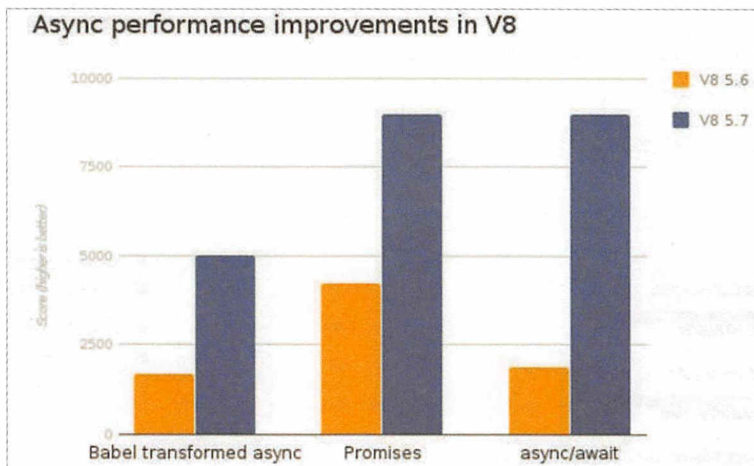


图 3-8

3.4.7 不会优化的特性

并不是说采用了 Turbofan 就能优化所有的 JavaScript 语法，V8 是不会去优化（也没有必要优化）某些语法的，例如 debugger、eval 和 with。

这里以 debugger 为例，比较使用和不使用 debugger 时的性能：

```
var benchmark = require('benchmark')
var suite = new benchmark.Suite()

suite.add('with debugger', function withDebugger () {
  var base = 0
  var max = 65535

  var total = 0

  for (var i = base; i < max; i++) {
    debugger
    total += i
  }
})

suite.add('without debugger', function withoutDebugger () {
  var base = 0
```

```
    var max = 65535

    var total = 0

    for (var i = base; i < max; i++) {
      total += i
    }
  })

suite.on('complete', require('./print'))
suite.run()
```

运行结果如图 3-9 所示。

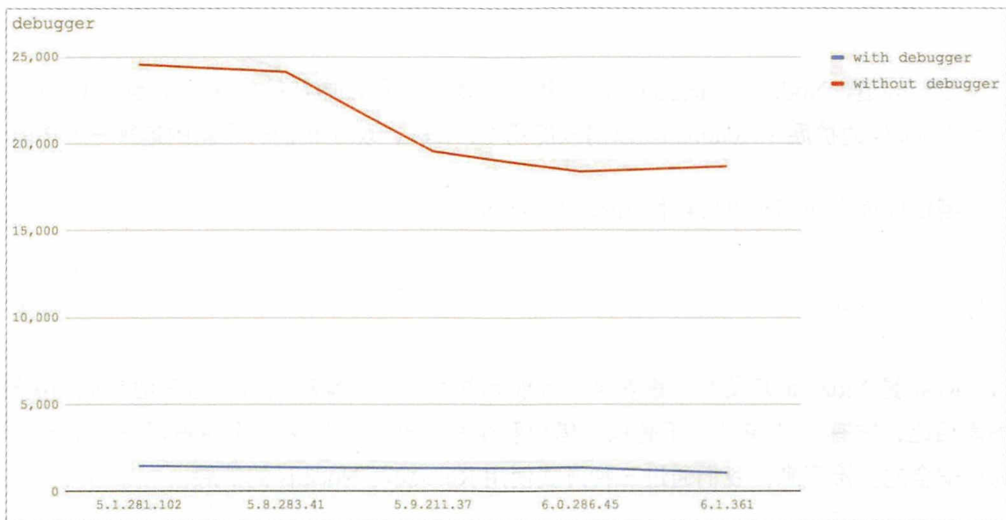


图 3-9

结论：在所有测试的 V8 版本中，debugger 一直都很慢，所以记得在打点测试完后一定要删掉 debugger。

总结如下。

- 要使用最新 LTS 版本的 Node.js。
- 关注 V8 团队的博客（https://v8project.blogspot.com），了解第一手资讯。
- 书写清晰的代码远比使用各种奇技淫巧来提升一点点性能重要得多。

本节的参考链接如下。

- https://github.com/davidmarkclements/v8-perf
- http://www.infoq.com/cn/news/2016/08/v8-ignition-javascript-inteprete
- https://docs.google.com/presentation/d/1H1lLsbclvzyOF3IUR05ZUaZcqDxo7_-8f4yJoxdMooU/edit#slide=id.g18ceb14729_0_59
- https://www.nearform.com/blog/node-js-is-getting-a-new-v8-with-turbofan
- https://zhuanlan.zhihu.com/p/26669846

3.5 Rust Addons

我们知道，Node.js 不适合 CPU 密集型计算的场景，通常的解决方法是用 C、C++ 编写 Node.js 的扩展（Addons）。以前只能用 C、C++，现在我们有了新的选择——Rust。

测试环境为 node@8.9.4 和 rust@1.26.0-nightly。

3.5.1 Rust

Rust 是 Mozilla 开发的注重安全、性能和并发的现代编程语言。与其他常见的编程语言相比，它有 3 个特性：所有权、借用和生命周期，正是这 3 个特性保证了 Rust 是内存安全的。接下来，我们通过三种方式使用 Rust 编写 Node.js 的扩展。

3.5.2 FFI

FFI（Foreign Function Interface）是用来与其他语言交互的接口，即可以用 Node.js 调用动态链接库。

运行以下命令：

```
$ cargo new ffi-demo && cd ffi-demo
$ npm init -y
$ npm i ffi --save
```

```
$ touch index.js
```

对部分文件的修改如下。

src/lib.rs：

```
#[no_mangle]
pub extern fn fib(n: i64) -> i64 {
  return match n {
    1 | 2 => 1,
    n => fib(n - 1) + fib(n - 2)
  }
}
```

Cargo.toml：

```
[package]
name = "ffi-demo"
version = "0.1.0"

[lib]
name = "ffi"
crate-type = ["dylib"]
```

Cargo.toml 是 Rust 项目的配置文件，相当于 Node.js 中的 package.json。这里指定编译生成的类型是 dylib（动态链接库），其名称在 Linux 下是 libffi，在 Windows 下是 ffi。

cargo 是 Rust 的构建工具和包管理工具，负责构建代码、下载依赖库并编译它们。使用 cargo 编译代码：

```
$ cargo build #开发环境用
或者：
$ cargo build --release #生产环境用，编译器做了更多优化，但编译慢
```

此时会生成一个 target 的目录，在该目录下会有 debug（不加 --release）或者 release（加 --release）目录，存放了生成的动态链接库。

index.js 的代码如下：

```
const ffi = require('ffi')
const isWin = /^win/.test(process.platform)
```

```
const rust = ffi.Library('target/debug/' + (!isWin ? 'lib' : '') + 'ffi', {
  fib: ['int', ['int']]
})

function fib(n) {
  if (n === 1 || n === 2) {
    return 1
  }
  return fib(n - 1) + fib(n - 2)
}

// js
console.time('node')
console.log(fib(40))
console.timeEnd('node')

// rust
console.time('rust')
console.log(rust.fib(40))
console.timeEnd('rust')
```

运行 index.js：

```
$ node index.js
102334155
node: 1053.743ms
102334155
rust: 1092.570ms
```

将 index.js 中的 debug 改为 release，运行：

```
$ cargo build --release
$ node index.js
102334155
node: 1050.467ms
102334155
rust: 273.508ms
```

可以看出，添加了 --release 编译后的代码，执行效率有了很明显的提升。

3.5.3 Neon

Neon 是一个可以使用 Rust 编写 Node.js 插件的模块。

使用方法如下：

```
$ npm i neon-cli -g
$ neon new neon-demo
$ cd neon-demo
$ tree .
.
├── README.md
├── lib
│   └── index.js
├── native
│   ├── Cargo.toml
│   ├── build.rs
│   └── src
│       └── lib.rs
└── package.json

3 directories, 6 files
$ npm i # 触发 neon build
$ node lib/index.js
hello node
```

接下来看看关键的代码文件。

lib/index.js：

```
var addon = require('../native');
console.log(addon.hello());
```

native/src/lib.rs：

```
#[macro_use]
extern crate neon;

use neon::vm::{Call, JsResult};
use neon::js::JsString;

fn hello(call: Call) -> JsResult<JsString> {
```

```
    let scope = call.scope;
    Ok(JsString::new(scope, "hello node").unwrap())
}

register_module!(m, {
    m.export("hello", hello)
});
```

native/build.rs

```
extern crate neon_build;

fn main() {
    neon_build::setup(); // must be called in build.rs

    // add project-specific build logic here...
}
```

native/Cargo.toml：

```
[package]
name = "neon-demo"
version = "0.1.0"
authors = ["nswbmw <gxqzk@126.com>"]
license = "MIT"
build = "build.rs"

[lib]
name = "neon_demo"
crate-type = ["dylib"]

[build-dependencies]
neon-build = "0.1.22"

[dependencies]
neon = "0.1.22"
```

在运行 neon build 时，会根据 native/Cargo.toml 中 build 字段指定的文件（这里是 build.rs）编译，并且生成的类型是 dylib（动态链接库）。native/src/lib.rs 存放了扩展的代码逻辑，通过 register_module 注册了一个 hello 方法，返回 hello node 字符串。

接下来测试原生 Node.js 和 Neon 编写的扩展运行斐波那契数列的执行效率。

将对应的文件修改如下。

native/src/lib.rs：

```
#[macro_use]
extern crate neon;

use neon::vm::{Call, JsResult};
use neon::mem::Handle;
use neon::js::JsInteger;

fn fib(call: Call) -> JsResult<JsInteger> {
  let scope = call.scope;
  let index: Handle<JsInteger> = try!(try!(call.arguments.require(scope, 0)).
check::<JsInteger>());
  let index: i32 = index.value() as i32;
  let result: i32 = fibonacci(index);
  Ok(JsInteger::new(scope, result))
}

fn fibonacci(n: i32) -> i32 {
  match n {
    1 | 2 => 1,
    _ => fibonacci(n - 1) + fibonacci(n - 2)
  }
}

register_module!(m, {
  m.export("fib", fib)
});
```

lib/index.js：

```
const rust = require('../native')

function fib (n) {
  if (n === 1 || n === 2) {
    return 1
  }
  return fib(n - 1) + fib(n - 2)
}

// js
```

```
console.time('node')
console.log(fib(40))
console.timeEnd('node')

// rust
console.time('rust')
console.log(rust.fib(40))
console.timeEnd('rust')
```

运行：

```
$ neon build
$ node lib/index.js
102334155
node: 1030.681ms
102334155
rust: 270.417ms
```

接下来看一个复杂点的例子，用 Neon 编写一个 User 类，可传入一个含有 first_name 和 last_name 的对象，暴露出一个 get_full_name 方法。

将对应的文件修改如下。

native/src/lib.rs：

```
#[macro_use]
extern crate neon;

use neon::js::{JsFunction, JsString, Object, JsObject};
use neon::js::class::{Class, JsClass};
use neon::mem::Handle;
use neon::vm::Lock;

pub struct User {
  first_name: String,
  last_name: String,
}

declare_types! {
  pub class JsUser for User {
    init(call) {
      let scope = call.scope;
```

```
        let user = try!(try!(call.arguments.require(scope, 0)).
check::<JsObject>());
        let first_name: Handle<JsString> = try!(try!(user.get(scope, "first_
name")).check::<JsString>());
        let last_name: Handle<JsString> = try!(try!(user.get(scope, "last_
name")).check::<JsString>());

        Ok(User {
          first_name: first_name.value(),
          last_name: last_name.value(),
        })
      }

      method get_full_name(call) {
        let scope = call.scope;
        let first_name = call.arguments.this(scope).grab(|user| { user.first_
name.clone() });
        let last_name = call.arguments.this(scope).grab(|user| { user.last_
name.clone() });
        Ok(try!(JsString::new_or_throw(scope, &(first_name + &last_name))).
upcast())
      }
    }
}

register_module!(m, {
  let class: Handle<JsClass<JsUser>> = try!(JsUser::class(m.scope));
  let constructor: Handle<JsFunction<JsUser>> = try!(class.constructor(m.
scope));
  try!(m.exports.set("User", constructor));
  Ok(())
});
```

lib/index.js：

```
const rust = require('../native')
const User = rust.User

const user = new User({
  first_name: 'zhang',
  last_name: 'san'
})
```

```
console.log(user.get_full_name())
```

运行：

```
$ neon build
$ node lib/index.js
zhangsan
```

3.5.4 NAPI

不少 Node.js 开发者可能都遇到过升级 Node.js 版本导致程序运行不起来的情况，这时需要重新安装依赖如 node-sass 模块来解决问题。因为之前在编写 Node.js 扩展时严重依赖 V8 暴露的 API，而不同版本的 Node.js 依赖的 V8 版本可能不同，所以一旦升级 Node.js 的版本，原先正常运行的 Node.js 扩展就可能失效了。

NAPI 是 node@8 新添加的用于原生模块开发的接口，相较于以前的开发方式，NAPI 提供了稳定的 ABI 接口，消除了 Node.js 版本差异、引擎差异等编译后不兼容的问题，解决了编写 Node.js 插件时最让人头疼的问题。

目前 NAPI 还处于试验阶段，所以相关资料并不多，笔者写了一个 demo 并将其放到了 GitHub 上，这里直接 clone 下来运行：

```
$ git clone https://github.com/nswbmw/rust-napi-demo
```

主要文件的代码如下。

src/lib.rs：

```
#[macro_use]
extern crate napi;
#[macro_use]
extern crate napi_derive;

use napi::{NapiEnv, NapiNumber, NapiResult};

#[derive(NapiArgs)]
struct Args<'a> {
  n: NapiNumber<'a>
}
```

```
fn fibonacci<'a>(env: &'a NapiEnv, args: &Args<'a>) ->
NapiResult<NapiNumber<'a>> {
  let number = args.n.to_i32()?;
  NapiNumber::from_i32(env, _fibonacci(number))
}

napi_callback!(export_fibonacci, fibonacci);

fn _fibonacci(n: i32) -> i32 {
  match n {
    1 | 2 => 1,
    _ => _fibonacci(n - 1) + _fibonacci(n - 2)
  }
}
```

index.js：

```
const rust = require('./build/Release/example.node')

function fib (n) {
  if (n === 1 || n === 2) {
    return 1
  }
  return fib(n - 1) + fib(n - 2)
}

// js
console.time('node')
console.log(fib(40))
console.timeEnd('node')

// rust
console.time('rust')
console.log(rust.fibonacci(40))
console.timeEnd('rust')
```

运行结果：

```
$ npm start
102334155
node: 1087.650ms
102334155
```

```
rust: 268.395ms
(node:33302) Warning: N-API is an experimental feature and could change at
any time.
```

本节的参考链接如下。

- https://github.com/neon-bindings/neon
- https://github.com/napi-rs/napi
- https://zhuanlan.zhihu.com/p/27650526

3.6 Event Loop

事件循环（Event Loop）是 Node.js 最核心的概念，所以理解 Event Loop 如何运作对于写出正确的代码和调试是非常重要的。比如考虑以下代码：

```
setTimeout(() => {
  console.log('hi')
}, 1000)
...
```

我们期望程序在运行 1s 后打印出 hi，但是实际情况可能是在远大于 1s 后才打印出 hi。这时如果理解 Event Loop 就可以轻易地发现问题，否则任凭怎么调试都是发现不了问题的。

3.6.1 什么是 Event Loop

我们可以简单理解 Event Loop 如下。

- 所有任务都在主线程上执行，形成一个执行栈（Execution Context Stack）。
- 在主线程之外还存在一个“任务队列”（Task Queue），系统把异步任务放到“任务队列”中，然后主线程继续执行后续的任务。
- 一旦“执行栈”中的所有任务执行完毕，系统就会读取“任务队列”。如果这时异步任务已经结束了等待状态，就会从“任务队列”进入执行栈，恢复执行。
- 主线程不断重复上面的第三步。

仅供非商业用途或交流学习使用

> **小提示**
>
> 我们常说 Node.js 是单线程的，但为何能达到高并发呢？原因就在于底层的 Libuv 维护一个 I/O 线程池（即上述的"任务队列"），结合 Node.js 异步 I/O 的特性，单线程也能达到高并发。

上面提到了"读取任务队列"，这样讲有点笼统，其实 Event Loop 的"读取任务队列"有 6 个阶段，如图 3-10 所示。

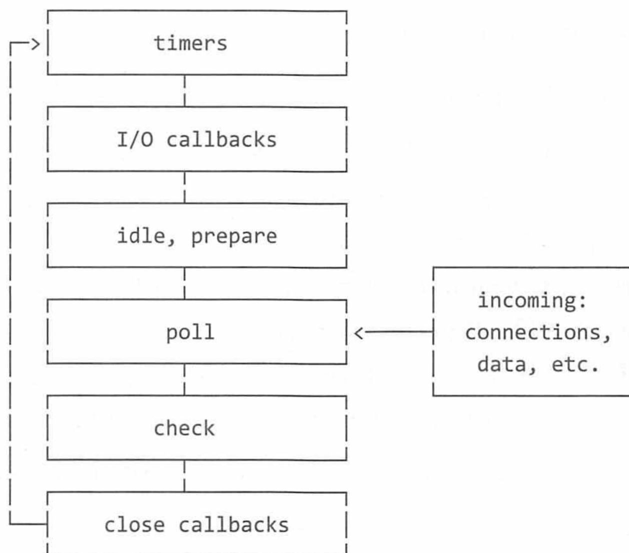


图 3-10

每个阶段都有一个 FIFO 的回调队列（queue），当 Event Loop 执行到这个阶段时，就会从当前阶段的队列里拿出一个任务放到栈中执行，在队列任务清空或者执行的回调数量达到上限后，Event Loop 就会进入下一个阶段。

每个阶段（phase）的作用如下。

- timers：执行 setTimeout() 和 setInterval() 中到期的 callback。
- I/O callbacks：上一轮循环中有少数的 I/O callback 会被延迟到这一轮的这一阶段执行。

仅供非商业用途或交流学习使用

- idle, prepare：仅内部使用。
- poll：为最重要的阶段，执行 I/O callback，在适当的条件下 node 会阻塞在这个阶段。
- check：执行 setImmediate() 的 callback。
- close callbacks：执行 close 事件的 callback，例如 socket.on('close',func)。

3.6.2 poll 阶段

poll 阶段主要有两个功能，如下所述。

- 当 timers 的定时器到期后，执行定时器（setTimeout 和 setInterval）的 callback。
- 执行 poll 队列里面的 I/O callback。

如果 Event Loop 进入了 poll 阶段，且代码未设定 timer，则可能发生以下情况。

- 如果 poll queue 不为空，则 Event Loop 将同步执行 queue 里的 callback，直至 queue 为空，或者执行的 callback 到达系统上限。
- 如果 poll queue 为空，则可能发生以下情况。
  - 如果代码使用 setImmediate() 设定了 callback，则 Event Loop 将结束 poll 阶段并进入 check 阶段，执行 check 阶段的 queue。
  - 如果代码没有使用 setImmediate()，则 Event Loop 将阻塞在该阶段，等待 callbacks 加入 poll queue，如果有 callback 进来则立即执行。

一旦 poll queue 为空，则 Event Loop 将检查 timers，如果有 timer 的时间到期，则 Event Loop 将回到 timers 阶段，然后执行 timer queue。

3.6.3 process.nextTick()

上面的 6 个阶段并没有出现 process.nextTick()，process.nextTick() 不在 Event Loop 的任何阶段执行，而是在各个阶段切换的中间执行，即从一个阶段切换到下个阶段前执行。这里还需要提一下 macrotask 和 microtask 的概念，macrotask（宏任务）指 Event Loop 在每个阶段执行的任务，microtask（微任务）指在每个阶段之间执行的任务。即上述 6 个阶段都属于 macrotask，process.nextTick() 属于 microtask。

仅供非商业用途或交流学习使用

小提示

process.nextTick() 的实现和 V8 的 microtask 并无关系，是 Node.js 层面的东西，应该说 process.nextTick() 的行为接近于 microtask。Promise.then 也属于 microtask 的一种。

最后给出一张关于 Event Loop 的非常直观的图，如图 3-11 所示。

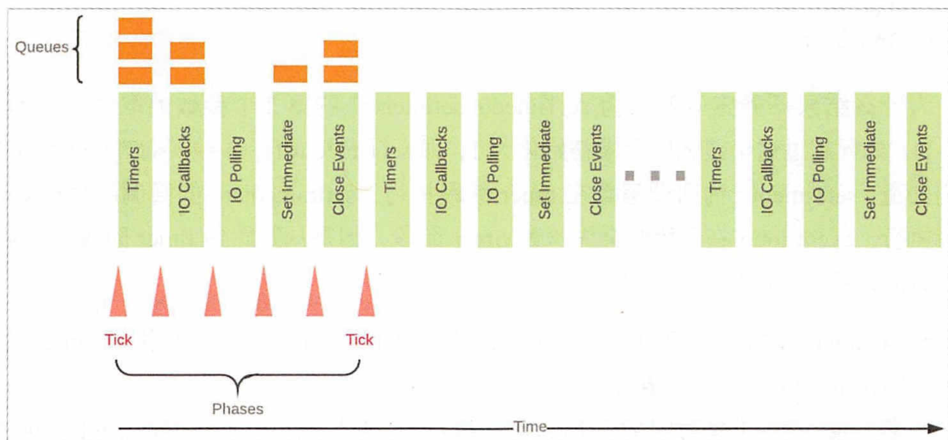


图 3-11

绿色小块表示 Event Loop 的各个阶段，执行的是 macrotask，macrotask 中间的粉红箭头表示执行的是 microtask。

3.6.4 代码解析

下面我们通过解析 6 段代码来巩固在前面学到的 Event Loop 的知识。

代码一

```
setTimeout(() => {
  console.log('setTimeout')
}, 0)

setImmediate(() => {
```

仅供非商业用途或交流学习使用


```
  console.log('setImmediate')
})
```

运行结果为：

```
setImmediate
setTimeout
```

或者：

```
setTimeout
setImmediate
```

为什么结果不确定呢？因为 setTimeout/setInterval 的第 2 个参数的取值范围是 $[1, 2^{31}-1]$，如果超过这个范围则会被初始化为 1，即 setTimeout(fn, 0) === setTimeout(fn, 1)。我们知道，setTimeout 的回调函数在 timer 阶段执行，setImmediate 的回调函数在 check 阶段执行，event loop 在开始时会先检查 timer 阶段，但是从开始到 timer 阶段会消耗一定的时间，所以会出现如下两种情况。

- 若 timer 前的准备时间超过 1ms，满足 loop->time >= 1，则执行 timer 阶段（setTimeout）的回调函数。
- 若 timer 前的准备时间少于 1ms，则先执行 check 阶段（setImmediate）的回调函数，下一次 event loop 执行 timer 阶段（setTimeout）的回调函数。

再看个例子：

```
setTimeout(() => {
  console.log('setTimeout')
}, 0)

setImmediate(() => {
  console.log('setImmediate')
})

const start = Date.now()
while (Date.now() - start < 10);
```

运行结果一定是：

```
setTimeout
setImmediate
```

仅供非商业用途或交流学习使用

代码二

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

fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('setTimeout')
  }, 0)

  setImmediate(() => {
    console.log('setImmediate')
  })
})
```

运行结果如下：

```
setImmediate
setTimeout
```

解释：fs.readFile 的回调函数在执行后，执行以下操作。

- 将 setTimeout 的回调函数注册到 timer 阶段。
- 将 setImmediate 的回调函数注册到 check 阶段。
- event loop 从 pool 阶段出来继续向下一个阶段执行，恰好是 check 阶段，所以 setImmediate 的回调函数先执行。
- 在本次 event loop 结束后，进入下一次 event loop，执行 setTimeout 的回调函数。

所以，在 I/O Callbacks 中注册的 setTimeout 和 setImmediate 永远都是 setImmediate 先执行。

代码三

```
setInterval(() => {
  console.log('setInterval')
}, 100)

process.nextTick(function tick () {
  process.nextTick(tick)
})
```

运行结果：永远不会打印出 setInterval。

仅供非商业用途或交流学习使用

解释：process.nextTick 会无限循环，将 event loop 阻塞在 microtask 阶段，导致 event loop 上其他 macrotask 阶段的回调函数没有机会执行。

解决方法通常是用 setImmediate 代替 process.nextTick：

```
setInterval(() => {
  console.log('setInterval')
}, 100)

setImmediate(function immediate () {
  setImmediate(immediate)
})
```

运行结果：每 100ms 打印一次 setInterval。

解释：在 process.nextTick 内执行 process.nextTick 时仍然将 tick 函数注册到当前 microtask 的尾部，所以导致 microtask 永远执行不完；在 setImmediate 内执行 setImmediate 时会将 immediate 函数注册到下一次 event loop 的 check 阶段，而不是当前执行的 check 阶段，所以给了 event loop 上其他 macrotask 执行的机会。

再看个例子：

```
setImmediate(() => {
  console.log('setImmediate1')
  setImmediate(() => {
    console.log('setImmediate2')
  })
  process.nextTick(() => {
    console.log('nextTick')
  })
})

setImmediate(() => {
  console.log('setImmediate3')
})
```

运行结果：

```
setImmediate1
setImmediate3
nextTick
setImmediate2
```

仅供非商业用途或交流学习使用

注　意

并不是说 setImmediate 可以完全代替 process.nextTick，process.nextTick 在特定场景下还是无法被代替的，比如我们只想将一些操作放到最近的 microtask 里执行的场景。

代码四

```
const promise = Promise.resolve()
  .then(() => {
    return promise
  })
promise.catch(console.error)
```

运行结果：

```
TypeError: Chaining cycle detected for promise #<Promise>
    at <anonymous>
    at process._tickCallback (internal/process/next_tick.js:188:7)
    at Function.Module.runMain (module.js:667:11)
    at startup (bootstrap_node.js:187:16)
    at bootstrap_node.js:607:3
```

解释：在 Promise A+ 的规范里规定了 promise 不能返回自己。仔细想想，即使在规范里不规定，promise.then 因为类似于 process.nextTick，所以都会将回调函数注册到 microtask 阶段。上面的代码也会导致死循环，类似前面提到的：

```
process.nextTick(function tick () {
  process.nextTick(tick)
})
```

再看个例子：

```
const promise = Promise.resolve()

promise.then(() => {
  console.log('promise')
})
```

仅供非商业用途或交流学习使用


```
process.nextTick(() => {
  console.log('nextTick')
})
```

运行结果：

```
nextTick
promise
```

解释：虽然和 process.nextTick 一样，promise.then 也将回调函数注册到 microtask，但其优先级不一样，process.nextTick 的 microtask queue 总是优先于 promise 的 microtask queue 执行的。

代码五

```
setTimeout(() => {
  console.log(1)
}, 0)
new Promise((resolve, reject) => {
  console.log(2)
  for (let i = 0; i < 10000; i++) {
    i === 9999 && resolve()
  }
  console.log(3)
}).then(() => {
  console.log(4)
})
console.log(5)
```

运行结果：

```
2
3
5
4
1
```

解释：Promise 构造函数是同步执行的，所以先打印 2、3，然后打印 5；接下来 event loop 进入执行 microtask 的阶段，执行 promise.then 的回调函数，打印出 4，；然后执行下一个 macrotask，恰好是 timer 阶段的 setTimeout 的回调函数，打印出 1。

仅供非商业用途或交流学习使用

代码六

```
setImmediate(() => {
  console.log(1)
  setTimeout(() => {
    console.log(2)
  }, 100)
  setImmediate(() => {
    console.log(3)
  })
  process.nextTick(() => {
    console.log(4)
  })
})
process.nextTick(() => {
  console.log(5)
  setTimeout(() => {
    console.log(6)
  }, 100)
  setImmediate(() => {
    console.log(7)
  })
  process.nextTick(() => {
    console.log(8)
  })
})
console.log(9)
```

运行结果：

```
9
5
8
1
7
4
3
6
2
```

关于 process.nextTick、setTimeout 和 setImmediate 的组合，请读者自行推理吧。

本节的参考链接如下。

- https://medium.com/the-node-js-collection/what-you-should-know-to-really-understand-the-node-js-event-loop-and-its-metrics-c4907b19da4c
- https://cnodejs.org/topic/57d68794cb6f605d360105bf

3.7 处理 uncaughtException

相信所有 Node.js 开发者都对 TypeError: Cannot read property 'xxx' of undefined/null 这种错误并不陌生，发生这种错误是因为程序期望从一个对象上获取 xxx 属性，结果这个对象的值是 undefined 或者 null。

3.7.1 uncaughtException

下面看一段代码：

```
const article = { title: 'Node.js', content: 'Hello, Node.js' }
setImmediate(() => {
  console.log(article.author.name)
})
```

运行以上代码会打印：

```
/Users/nswbmw/Desktop/test/app.js:3
  console.log(article.author.name)
                             ^
TypeError: Cannot read property 'name' of undefined
    at Timeout.setInterval [as _onTimeout] (/Users/nswbmw/Desktop/test/app.
js:3:30)
    at ontimeout (timers.js:475:11)
    at tryOnTimeout (timers.js:310:5)
    at Timer.listOnTimeout (timers.js:270:5)
```

article 是一个文章对象，有 title 和 content 属性，没有 author 属性，所以 article.author 是 undefined，调用 article.author.name 会报错。这个运行时错误是在一个异步函数（setImmediate）内抛出的，所以这个错误是一个“uncaught exception”，如果没有 process.on('uncaughtException', () => {}) 事件监听器，则程序会 crash。

调试这种错误没有较好的方法，通常只能添加 console.log 来打印 article 的值。我们前面介绍过 llnode 的用法，是否可以使用 llnode 调试这类错误呢？答案是肯定的。

3.7.2　使用 llnode

我们添加 --abort-on-uncaught-exception 参数重新运行程序，在程序 crash 的时候，会自动 Core Dump：

```
$ ulimit -c unlimited
$ node --abort-on-uncaught-exception app.js
Uncaught TypeError: Cannot read property 'name' of undefined

FROM
Immediate.setImmediate (/home/nswbmw/test/app.js:1:1)
runCallback (timers.js:1:1)
tryOnImmediate (timers.js:1:1)
processImmediate [as _immediateCallback] (timers.js:1:1)
Illegal instruction (core dumped)
```

此时会生成一个 core 文件，我们使用 llnode 加载并诊断这个 core 文件：

```
$ lldb-4.0 -c ./core
(lldb) target create --core "./core"
Core file '/home/nswbmw/test/./core' (x86_64) was loaded.
(lldb)
```

使用 v8 bt 查看最近的 backtrace：

```
(lldb) v8 bt
 * thread #1: tid = 4750, 0x00007ffd905d5b39 node`v8::base::OS::Abort() + 9,
name = 'node', stop reason = signal SIGILL
  * frame #0: 0x00007ffd905d5b39 node`v8::base::OS::Abort() + 9
    frame #1: 0x00007ffd900a4d19 node`v8::internal::Isolate::Throw(v8::intern
al::Object*, v8::internal::MessageLocation*) + 489
    frame #2: 0x00007ffd9005e7f9 node`v8::internal::LoadIC::Load(v8::interna
l::Handle<v8::internal::Object>, v8::internal::Handle<v8::internal::Name>) +
569
    frame #3: 0x00007ffd9005f759 node`v8::internal::Runtime_LoadIC_Miss(int,
v8::internal::Object**, v8::internal::Isolate*) + 633
    frame #4: 0x000018e6e710463d <exit>
    frame #5: 0x000018e6e71ecce4 <stub>
```

```
    frame #6: 0x000018e6e71bf9ce setImmediate(this=0x0000154a3ae09429:<Obje
ct: Immediate>) at /home/nswbmw/test/app.js:2:14 fn=0x0000154a3ae09281
    ...
```

可以看出，在 frame #4 处程序触发了 exit。往上追溯到 frame #6，发现有一个 setImmediate（在 app.js 第 2 行）抛出了错误，setImmediate 的回调函数的地址为 0x0000154a3ae09281，我们使用 v8 i 检索这个函数：

```
(lldb) v8 i 0x0000154a3ae09281
0x0000154a3ae09281:<function: setImmediate at /home/nswbmw/test/app.js:2:14
  context=0x0000154a3ae09199{
    (previous)=0x000017849b703d89
    (closure)=0x0000154a3ae08d81 {<function: (anonymous) at /home/nswbmw/
test/app.js:1:10>},
    article=0x0000154a3ae091d1:<Object: Object>}>
(lldb) v8 i 0x0000154a3ae091d1
0x0000154a3ae091d1:<Object: Object properties {
    .title=0x000014117e04c9e9:<String: "Node.js">,
    .content=0x000014117e04ca09:<String: "Hello, Node.js">}>
```

可以发现，在 setImmediate 函数内有一个 article 对象。我们继续通过 v8 i 检索，得知 article 的值为 { title: "Node.js", content: "Hello, Node.js" }，并没有 author 属性，真相大白。

3.7.3 ReDoS

ReDoS（RegExp Denial of Service，正则表达式拒绝服务攻击）是由于正则表达式的写法有缺陷，所以在使用正则匹配时，会出现大量占用 CPU 的情况，导致服务不可用，而导致正则表达式匹配“卡住”的原因正是正则表达式的“回溯”特性。

看一个简单的例子：

```
/a.*b/g.test('aaaf')
```

其匹配过程如图 3-12 所示。

```
MATCH 1 - FINISHED IN 18 STEPS

 1  /a.*b/g  aaaf
 2  /a.*b/g  aaaf
 3  /a.*b/g  aaaf
 4  /a.*b/g  aaaf BACKTRACK
 5  /a.*b/g  aaaf BACKTRACK
 6  /a.*b/g  aaaf BACKTRACK
 7  /a.*b/g  aaaf
 8  /a.*b/g  aaaf BACKTRACK
 9  /a.*b/g  aaaf
10  /a.*b/g  aaaf BACKTRACK
11  /a.*b/g  aaaf BACKTRACK
12  /a.*b/g  aaaf
13  /a.*b/g  aaaf BACKTRACK
14  /a.*b/g  aaaf
15  /a.*b/g  aaaf BACKTRACK
16  /a.*b/g  aaaf
17  /a.*b/g  aaaf BACKTRACK
18  /a.*b/g  aaaf
 #  Match failed in 18 step(s)
```

图 3-12

可以看出，因为“*”是贪婪匹配，所以第 3 步 .* 匹配了字符串的末尾，由于剩下的一个 b 无法匹配，所以先“吐出”一个字符再尝试匹配（第 4 步），结果仍然不匹配（第 5 步），所以继续“吐出”一个字符……，这个“吐出”一个字符的过程就是回溯（backtrack）。

再看个例子：

```
const reg = /(a*)+b/

console.time('reg')
reg.test('aaaaaaaaaaaaaaaaaaaaaaaaaaaf') // reg: 2572.022ms
// reg.test('aaaaaaaaaaaaaaaaaaaaaaaaaaaf') // reg: 5048.735ms
// reg.test('aaaaaaaaaaaaaaaaaaaaaaaaaaaf') // reg: 10710.070ms
console.timeEnd('reg')
```

运行以上代码，每添加一个字母 a，程序的运行时间就翻倍，这正是由于正则表达式的回溯导致的，这个正则表达式的时间复杂度为 $O(2^n)$。

具有回溯的正则表达式是比较常见的，在这种情况下程序会“卡住”，我们使用 llnode 也可以调试这类问题。

运行以下代码：

```
$ echo "/(a*)+b/.test('aaaaaaaaaaaaaaaaaaaaaaaaaaaaaf')" > app.js
$ node --abort-on-uncaught-exception app.js &
$ kill -BUS `pgrep -n node`
```

生成 core 文件，使用 llnode 调试：

```
$ lldb-4.0 -c ./core
(lldb) target create --core "./core"
Core file '/home/nswbmw/test/./core' (x86_64) was loaded.
(lldb) v8 bt
 * thread #1: tid = 5381, 0x000036a6db804f6b, name = 'node', stop reason =
signal SIGBUS
  * frame #0: 0x000036a6db804f6b <builtin>
    ...
    frame #6: 0x000036a6db68463d <exit>
    frame #7: 0x000036a6db7135f4 test(this=0x0000038344709119:<JSRegExp
source=/(a*)+b/>, 0x0000098ccdb4c9e9:<String: "aaaaaaaaaaaaaaaa...">) at (no
script) fn=0x0000183ca0c134a1
    ...
(lldb) v8 i -F 0x0000098ccdb4c9e9
0x0000098ccdb4c9e9:<String: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaf">
```

可以看出，程序在退出前，正在执行一个正则表达式（/(a*)+b/）的 test 方法，参数是字符串 (aaaaaaaaaaaaaaaaaaaaaaaaaaaaaf)。

减少正则表达式回溯的简单方法就是合并不必要的量词，例如将上面的正则表达式 /(a*)+b/ 修改为 /a*b/。

本节的参考链接如下。

- https://www.rawidn.com/posts/ddos-and-ddos-in-regular-expression.html

第4章 工具

工欲善其事，必先利其器。本章将会讲解多个调试相关的模块和工具的用法，最后会讲解一个黑魔法——Proxy，希望能帮助读者打开调试思路。

4.1 Source Map

对于 Source Map，想必大家并不陌生，在前端开发中通常要压缩 JavaScript、CSS 以减小体积，加快网页显示，但会导致在出现错误时无法对错误进行定位，Source Map 应运而生。举个例子，jQuery 1.9 引入了 Source Map，打开 http://ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js，最后一行是这样的：

```
//@ sourceMappingURL=jquery.min.map
```

这就是 Source Map。它是一个独立的 map（其实就是 JSON）文件，通常与源码在同一个目录下。

Source Map 常用于以下几个场景中。

（1）压缩代码，减小体积。比如 jQuery 1.9 的源码在压缩前是 252KB，在压缩后是 32KB。

（2）多个文件合并，减少 HTTP 请求数，仅用于前端。

（3）将其他语言编译成 JavaScript，例如 CoffeeScript、TypeScript 等。

本节只讲解如何使用 Source Map，不会解释 map 文件中的字段含义，有兴趣的读者可以查看参考链接中的文章。接下来在 Node.js 环境下以场景 1、3 为例，分别介绍如何将 uglify-es 和 TypeScript 结合 Source Map 使用。

4.1.1 uglify-es

uglify-js 是最常用的 JavaScript 代码压缩工具，但只支持到 ES 5 版本。uglify-es 支持 ES6 及以上版本并且兼容 uglify-js，所以本节使用了 uglify-es。

source-map-support 是一个在 Node.js 环境下支持 Source Map 的模块。

安装 uglify-es 和 source-map-support：

```
$ npm i uglify-es -g
```

```
$ npm i source-map-support
```

创建测试代码，如下所示。

app.js：

```
require('source-map-support').install()

function sayHello (name) {
  throw new Error('error!!!')
  console.log(`Hello, ${name}`)
}

sayHello('World')
```

使用 uglify-es 压缩代码文件并生成 map 文件：

```
$ uglifyjs app.js -o app.min.js --source-map "url=app.min.js.map"
```

生成 app.min.js 和 app.min.js.map 文件。

App.min.js 文件的内容如下：

```
require("source-map-support").install();function sayHello(name){throw new
Error("error!!!");console.log(`Hello, ${name}`)}sayHello("World");
//# sourceMappingURL=app.min.js.map
```

app.min.js.map 文件的内容如下：

```
{"version":3,"sources":["app.js"],"names":["require","install","sayHello","na
me","Error","console","log"],"mappings":"AAAAA,QAAQ,sBAAsBC,UAE9B,SAASC,SAAUC
,MACjB,MAAM,IAAIC,MAAM,YAChBC,QAAQC,cAAcH,QAGxBD,SAAS"}
```

此时运行 app.min.js，便可以显示正确的错误栈：

```
$ node app.min.js

/Users/nswbmw/Desktop/test/app.js:4
  throw new Error('error!!!')
        ^
Error: error!!!
    at sayHello (/Users/nswbmw/Desktop/test/app.js:4:9)
```

如果删除 app.min.js 最后的一行注释，则在重新运行后无法显示正确的错误栈：

```
$ node app.min.js

/Users/nswbmw/Desktop/test/app.min.js:1
require("source-map-support").install();function sayHello(name){throw new
Error("error!!!");console.log(`Hello, ${name}`)}sayHello("World");
                                                                  ^
Error: error!!!
    at sayHello (/Users/nswbmw/Desktop/test/app.min.js:1:71)
```

source-map-support 是通过 Error.prepareStackTrace 实现的，在 3.3 节讲解过它的用法，这里不再赘述。

4.1.2 TypeScript

全局安装 TypeScript：

```
$ npm i typescript -g
```

创建测试代码，如下所示。

app_ts.ts：

```
declare function require(name: string)
require('source-map-support').install()

function sayHello (name: string): any {
  throw new Error('error!!!')
}

sayHello('World')
```

运行如下代码，生成 app_ts.js 和 app_ts.js.map：

```
$ tsc --sourceMap app_ts.ts
```

运行 app_ts.js：

```
$ node app_ts.js
```

```
/Users/nswbmw/Desktop/test/app_ts.ts:5
  throw new Error('error!!!')
        ^
Error: error!!!
    at sayHello (/Users/nswbmw/Desktop/test/app_ts.ts:5:9)
```

4.1.3 source-map-support 的高级用法

我们可以在调用 install 方法时传入一个 retrieveSourceMap 参数，用来自定义处理 Source Map：

```
require('source-map-support').install({
  retrieveSourceMap: function(source) {
    if (source === 'compiled.js') {
      return {
        url: 'original.js',
        map: fs.readFileSync('compiled.js.map', 'utf8')
      }
    }
    return null
  }
})
```

比如将所有 map 文件缓存到内存中，而不是磁盘中。

本节的参考链接如下。

- http://www.ruanyifeng.com/blog/2013/01/javascript_source_map.html
- https://yq.aliyun.com/articles/73529
- https://github.com/v8/v8/wiki/Stack-Trace-API
- https://github.com/evanw/node-source-map-support

4.2 Chrome DevTools

调试是每个程序员的必备技能，因此选择合适的调试工具能极大地方便我们调试代码。Node.js 的调试工具有很多，常见的有万能的 console.log、debugger 和 node-

书籍是人类进步的阶梯

inspector。console.log 就不用说了；我们不推荐使用 debugger，因为它的使用比较烦琐，需手动打点，若忘记删除 debugger，则还会引起性能问题；另外，node-inspector 已经退出历史的舞台。

node@6.3 及以上版本内置了一个调试器，可以结合 Chrome DevTools 使用，而且比 node-inspector 更强大。

下面讲讲 Chrome DevTools 的用法。

4.2.1 使用 Chrome DevTools

创建示例代码，如下所示。

app.js：

```
const Paloma = require('paloma')
const app = new Paloma()

app.use(ctx => {
  ctx.body = 'hello world!'
})

app.listen(3000)
```

运行：

```
$ node --inspect app.js
```

注 意

如果想让代码在第 1 行就暂停执行，就需要使用 --inspect-brk 参数启动，即 node --inspect-brk app.js。

打开 Chrome 浏览器，访问 chrome://inspect，如图 4-1 所示。

书籍是人类进步的阶梯

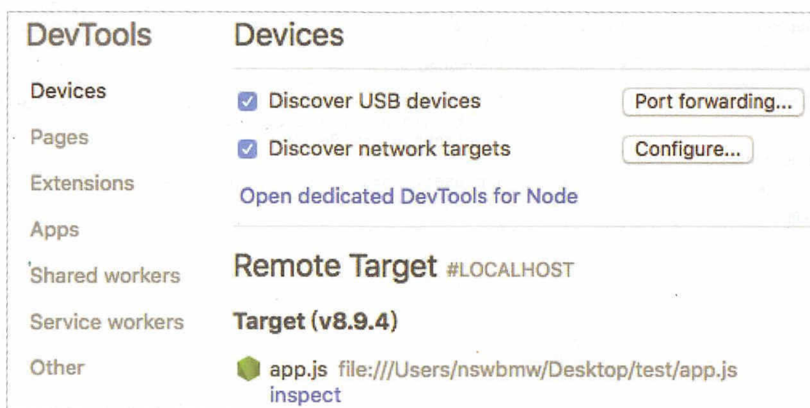


图 4-1

单击 Remote Target 下的 inspect，选择 Sources，如图 4-2 所示。

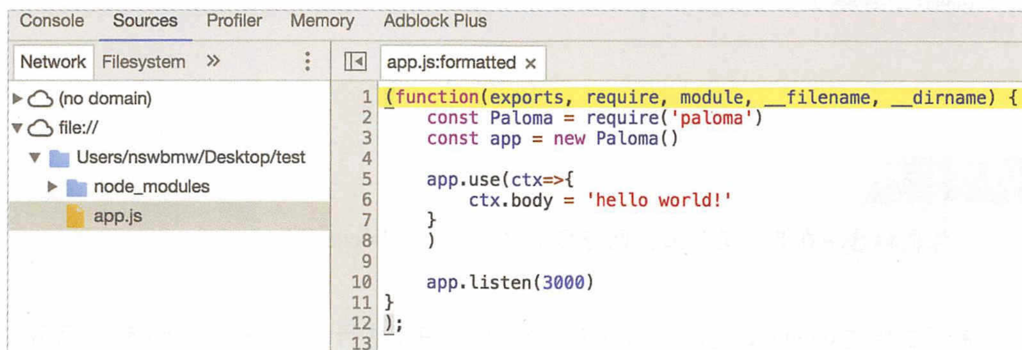


图 4-2

其使用方式与 node-inspector 类似，可以添加断点，然后在 Console 里直接输入变量名来打印该变量的值。如下所示，在第 6 行添加断点，然后运行 curl localhost:3000?name=nswbmw，代码在执行到第 6 行时暂停执行，在 Console 里打印 ctx.query 的值，如图 4-3 所示。

书籍是人类进步的阶梯

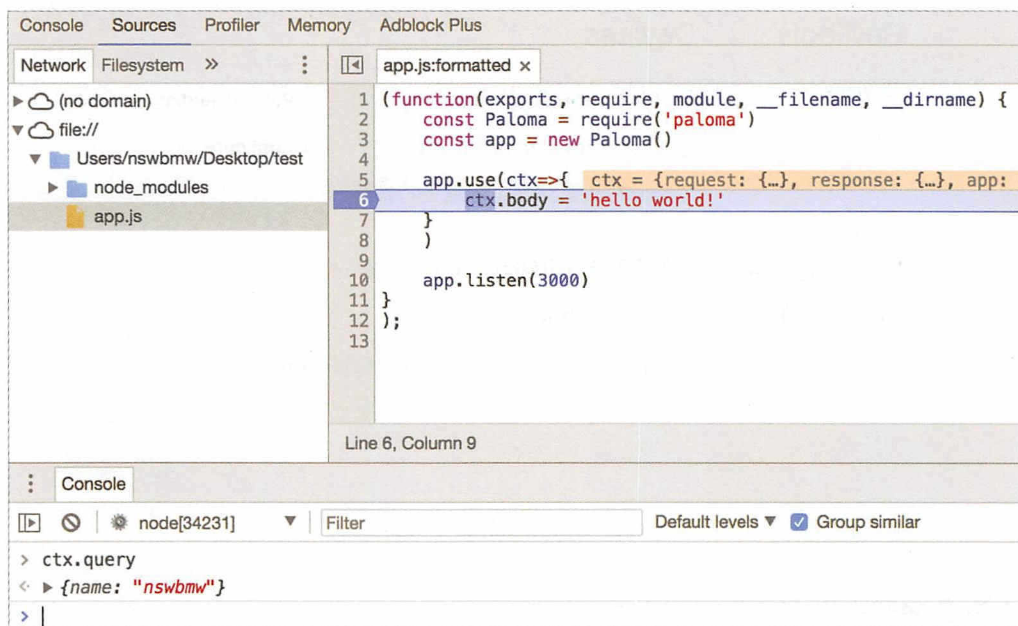


图 4-3

> ① 小提示
>
> 将鼠标悬浮在某个变量上，也会显示它的值，例如 ctx。

展开右侧的 debugger，会显示更多的功能，例如单步执行、单步进入、单步退出，等等，这里不再详细讲解。

4.2.2 NIM

在每次调试 Node.js 时都需要打开隐藏很深的入口，这很烦人，还好我们有了 NIM。NIM（Node Inspector Manager）是一个 Chrome 插件，可以帮助我们快捷地打开 DevTools，也可以设置自动发现并打开 DevTools，如图 4-4 所示。

书籍是人类进步的阶梯

图 4-4

4.2.3 inspect-process

如果你觉得 NIM 用起来也麻烦，那么你可能需要 inspect-process。

全局安装：

```
$ npm i inspect-process -g
```

使用：

```
$ inspect app.js
```

inspect-process 会自动调用 Chrome DevTools，然后定位到 app.js，其余用法与 Chrome DevTools 一致。

4.2.4 process._debugProcess

如果一个 Node.js 进程已经启动，没有添加 --inspect 参数，而我们不想重启（会丢失现场）又想调试，则这时可以使用 process._debugProcess。使用方法如下。

- 通过 ps 命令或者 pgrep -n node 查看当前启动的 Node.js 进程的 pid，例如 53911。
- 打开新的终端，运行 node -e "process._debugProcess(53911)"，原来的 Node.js 进程会打印出 Debugger listening on ws://127.0.0.1:9229/2331fa07-32af-45eb-a1a8-

书籍是人类进步的阶梯

bead7a0ab905。

- 调出 Chrome DevTools 进行调试。

本节的参考链接如下。

- https://medium.com/@paul_irish/debugging-node-js-nightlies-with-chrome-devtools-7c4a1b95ae27

4.3 Visual Studio Code

Visual Studio Code（简称 VS Code）是一款微软开源的现代化、跨平台、轻量级的代码编辑器。VS Code 很好用也很强大，本节将介绍如何使用 VS Code 来调试 Node.js 代码。

4.3.1 基本调试

示例代码如下。

app.js：

```
const Paloma = require('paloma')
const app = new Paloma()

app.use(ctx => {
  ctx.body = 'hello world!'
})

app.listen(3000)
```

用 VS Code 加载 test 文件夹，打开 app.js，然后进行如下操作。

- 单击左侧第 4 个 Tab，切换到调试模式。
- 单击代码第 5 行 ctx.body='hello world!'，在左侧空白处添加断点。
- 单击左上角“调试”的绿色三角按钮启动调试。
- 单击左上角的终端图标打开调试控制台。

结果如图 4-5 所示。

书籍是人类进步的阶梯

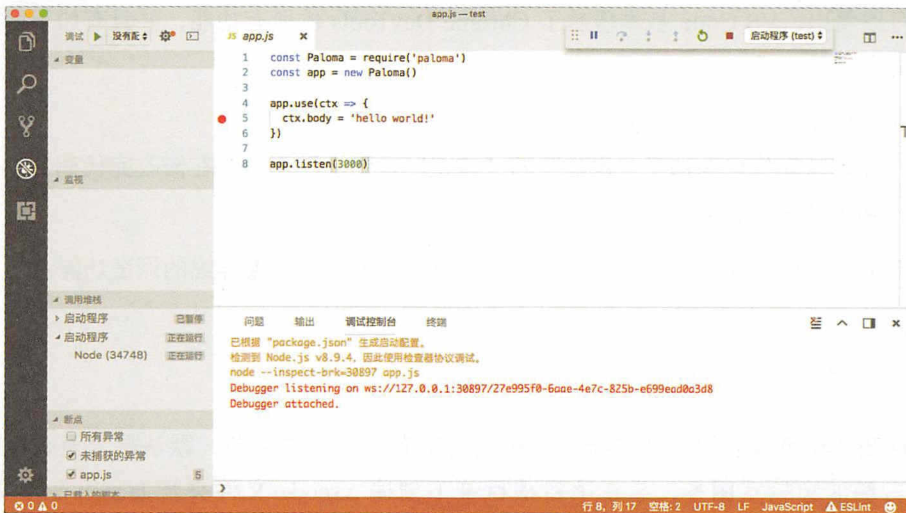


图 4-5

从调试控制台切换到终端，运行：

```
$ curl localhost:3000
```

运行结果如图 4-6 所示。

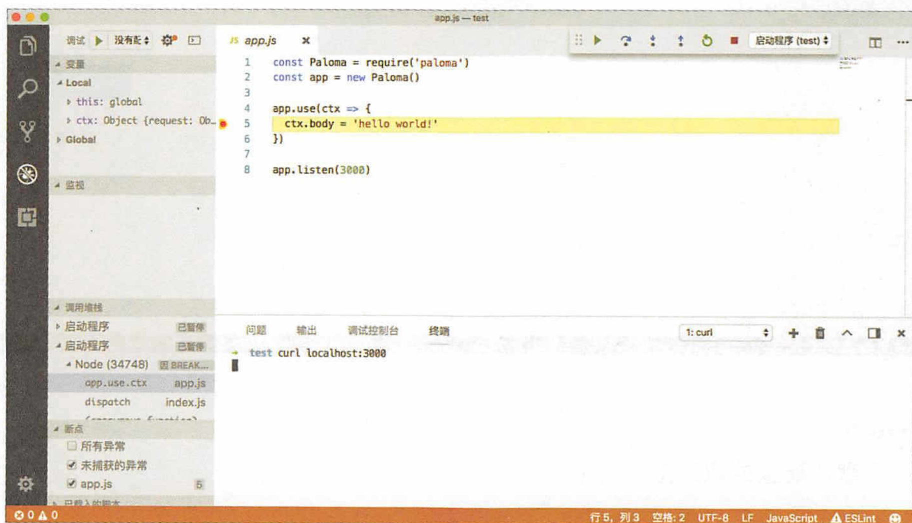


图 4-6

书籍是人类进步的阶梯

可以看出，VS Code 基本覆盖了 Chrome DevTools 的所有功能，并且有如下两个额外的优点。

- 集成了终端，不用再打开新的终端输入命令了。
- 在调试动作里添加了“重启”和“停止”按钮，不用每次在修改完代码后切回终端去重启了。

但 VS Code 的强大远不止于此，通过 launch.json 可以配置详细的调试功能。

4.3.2 launch.json

从图 4-6 可以看出，在“调试”按钮右边有一个下拉菜单，默认是“没有配置”。单击右侧的齿轮状图标，会在项目根目录下创建 .vscode 文件夹及 launch.json 文件。launch.json 的内容如图 4-7 所示。

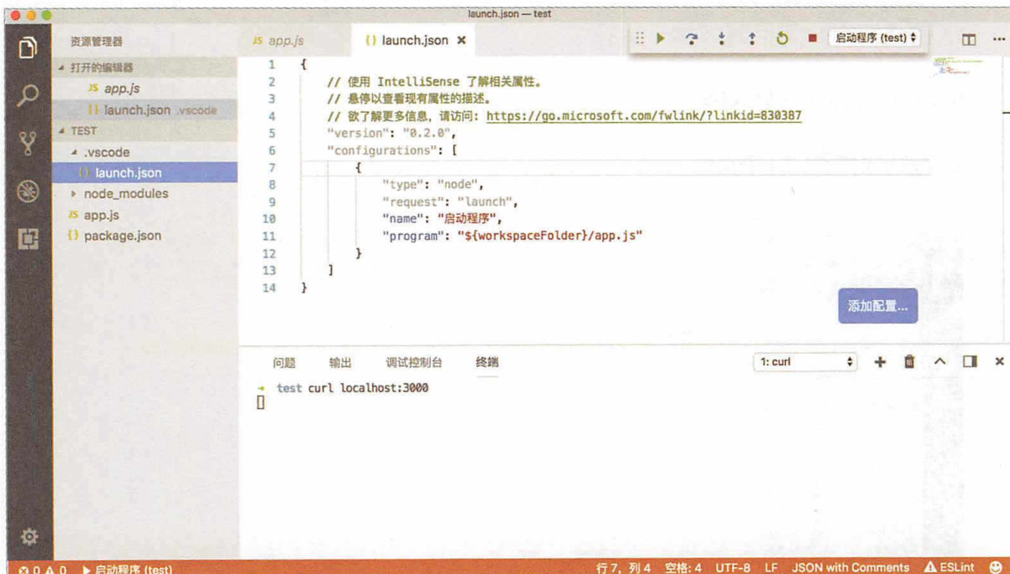


图 4-7

这个默认配置的意思是执行：

```
$ node ${workspaceFolder}/app.js
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

launch.json 其实存储了调试相关的一些配置，VS Code 在启动调试时，会读取 launch.json 来决定以何种方式调试。launch.json 有以下常用的选项，如下所述。

（1）必需的字段如下。

- type：指调试器的类型。这里是 node（内置的调试器），如果安装了 Go 和 PHP 的扩展，则对应的 type 分别为 go 和 php。
- request：指请求的类型，支持 launch 和 attach。launch 就是以 debug 模式启动调试，attach 就是附加到已经启动的进程开启 debug 模式并调试，跟在 4.3.1 节中提到的 node -e "process._debugProcess(PID)" 的作用一样。
- name：指下拉菜单显示的名字。

（2）可选的字段（在括号里的表示适用的类型）如下。

- program：指可执行的文件或者调试器要运行的文件（launch）。
- args：指要传递给调试程序的参数（launch）。
- env：指环境变量（launch）。
- cwd：指当前执行的目录（launch）。
- address：指 IP 地址（launch & attach）。
- port：指端口号（launch & attach）。
- skipFiles：指想要忽略的文件，为数组类型（launch & attach）。
- processId：指进程 PID（attach）。
- ……

（3）选项里可用的变量替换如下。

- ${workspaceFolder}：指当前打开工程的路径。
- ${file}：指当前打开文件的路径。
- ${fileBasename}：指当前打开文件的名字，包含后缀名。
- ${fileDirname}：指当前打开文件所在的文件夹的路径。
- ${fileExtname}：指当前打开文件的后缀名。
- ${cwd}：指当前执行的目录。
- ……

如果当前打开的文件是 app.js，则以下配置与默认配置是等效的：

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "启动程序",
            "program": "${file}"
        }
    ]
}
```

若想了解更多的 launch.json 选项，则请查阅 VS Code 官方文档。

下面以 5 个实用的技巧讲解部分 launch.json 配置的作用。

4.3.3 技巧 1——条件断点

VS Code 可以添加条件断点，即执行到该行代码满足特定条件后程序才会中断。在断点的小红点上用鼠标右键选择“编辑断点”，可以选择以下两种条件。

（1）表达式：当表达式的计算结果为 true 时中断，例如设置 ctx.query.name === 'nswbmw' 表示当访问 localhost:3000?name=nswbmw 时断点才会生效，其余请求断点无效。

（2）命中次数：同样在表达式计算结果为 true 时中断，支持运算符 <、<=、==、>、>= 和 %。下面举例说明。

- >10：在执行 10 次以后断点才会生效。
- <3：只有前两次断点会生效。
- 10：等价于≥ 10。
- %2：隔一次中断一次。

注　意

可以组合表达式和命中次数条件一起使用。在切换条件类型时，需要将原来的条件清空，否则会添加两种条件。将鼠标悬浮在断点上，可以查看设置了哪些条件。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

4.3.4 技巧 2——skipFiles

从图 4-7 可以看到，在 VS Code 左侧有一个“调用堆栈”面板，显示了当前断点的调用堆栈，但无法直观地看出哪些是我们的项目的代码，哪些是 node_modules 里的模块的代码，而且在单步调试时会进入 node_modules 里。总之，我们不关心 node_modules 里的代码，只关心项目本身的代码，这时，skipFiles 就派上用场了。

skipFiles 顾名思义就是忽略我们不关心的文件。修改 launch.json 如下：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": " 启动程序 ",
            "program": "${workspaceFolder}/app.js",
            "skipFiles": [
                "${workspaceFolder}/node_modules/**/*.js",
                "<node_internals>/**/*.js"
            ]
        }
    ]
}
```

有以下几点需要解释。

- 支持 ${xxx} 这种变量替换。
- 支持 glob 模式匹配。
- 用来忽略 Node.js 核心模块。

在重启调试后，结果如图 4-8 所示。

仅供非商业用途或交流学习使用

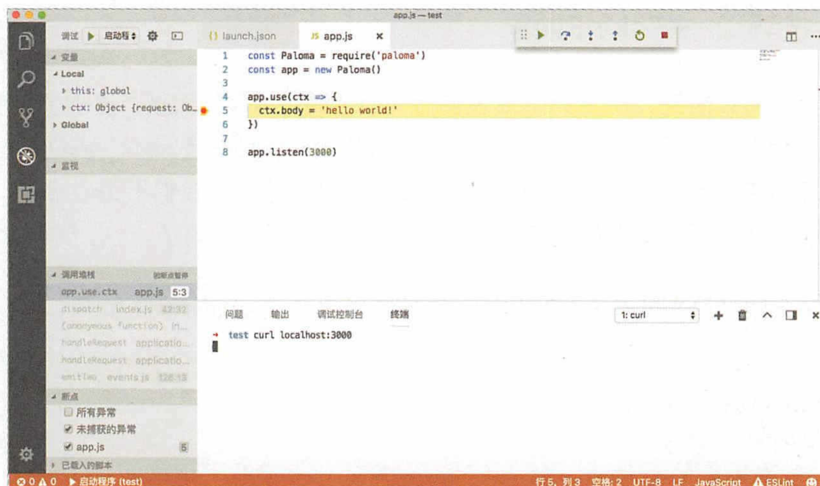


图 4-8

可以看出，在左侧的“调用堆栈”中，我们不关心的调用栈都变灰了，而且单步调试也不会进入 skipFiles 所匹配的文件里。

4.3.5　技巧 3——自动重启

在每次修改代码并保存后都要手动重启，否则修改后的代码和断点都不会生效。VS Code 开发者们想到了这一点，通过添加配置可以实现在修改代码并保存后自动重启调试，需要结合 nodemon 一起使用。

首先，全局安装 nodemon：

```
$ npm i nodemon -g
```

然后，修改 launch.json：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": " 启动程序 ",
```

仅供非商业用途或交流学习使用


```
                "runtimeExecutable": "nodemon",
                "program": "${workspaceFolder}/app.js",
                "restart": true,
                "console": "integratedTerminal",
                "skipFiles": [
                    "${workspaceFolder}/node_modules/**/*.js",
                    "<node_internals>/**/*.js"
                ]
            }
        ]
    }
```

当前的 launch.json 与上一个版本相比，多了以下几个字段。

- runtimeExecutable：指用什么命令执行 app.js，在这里设置为 nodemon。
- restart：设置为 true，在修改代码并保存后会自动重启调试。
- console：在单击停止按钮或者修改代码并保存后自动重启调试，而 nodemon 仍在运行，通过设置 console 为 integratedTerminal 便可以解决这个问题。此时 VS Code 终端将会打印 nodemon 的 log，可以在终端右侧的下拉菜单中选择返回第 1 个终端，然后运行 curl localhost:3000 进行调试。

对于已经使用 nodemon 运行的程序，例如：

```
$ nodemon --inspect app.js
```

可使用 attach 模式启动调试。launch.json 如下：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Attach to node",
            "type": "node",
            "request": "attach",
            "restart": true,
            "processId": "${command:PickProcess}"
        }
    ]
}
```

仅供非商业用途或交流学习使用

在运行 Attach to node 配置进行调试时，VS Code 会列出正在执行的 Node.js 进程及对应的 PID 以供选择。也可以通过 address 和 port 参数设置 attach 到具体的进程来开启调试。

4.3.6 技巧 4——对特定操作系统的设置

针对不同的操作系统，我们可能会用到不同的调试配置。可选的参数有 windows、linux 和 osx。

示例如下：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": " 启动调试 ",
            "program": "./node_modules/gulp/bin/gulpfile.js",
            "args": ["/path/to/app.js"],
            "windows": {
                "args": ["\\path\\to\\app.js"]
            }
        }
    ]
}
```

4.3.7 技巧 5——多配置

configurations 是一个数组而不是一个对象，这样的设计是为了添加多个调试配置。打开 launch.json，单击右下角的“添加配置…”，会弹出配置模板，如图 4-9 所示。

仅供非商业用途或交流学习使用

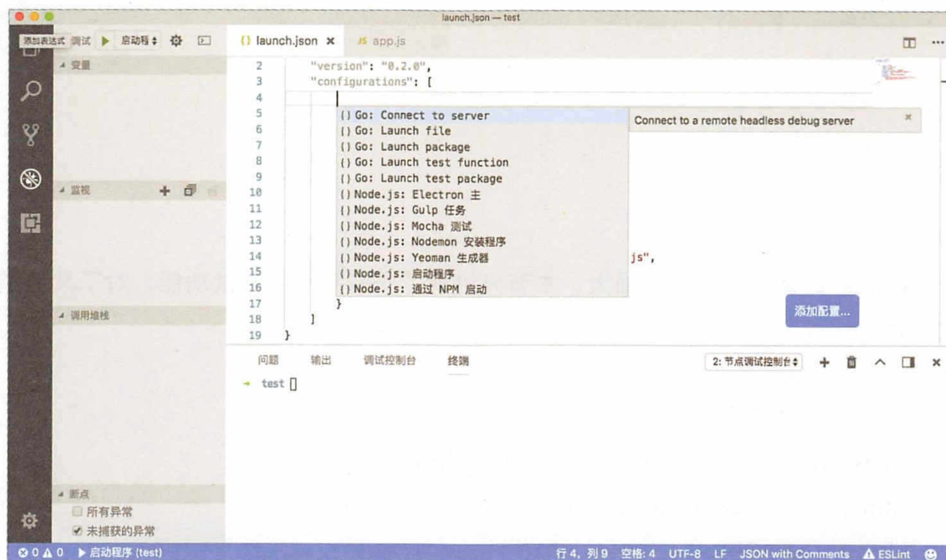


图 4-9

configurations 可以用来配置不同的调试规则，比如最终将 launch.json 修改如下：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "attach",
            "name": "Attach to node",
            "restart": true,
            "processId": "${command:PickProcess}"
        },
        {
            "type": "node",
            "request": "launch",
            "name": " 启动程序 ",
            "runtimeExecutable": "nodemon",
            "program": "${workspaceFolder}/app.js",
            "restart": true,
            "console": "integratedTerminal",
            "skipFiles": [
                "${workspaceFolder}/node_modules/**/*.js",
                "<node_internals>/**/*.js"
```

仅供非商业用途或交流学习使用


```
            ]
        }
    ]
}
```

4.3.8 总结

VS Code 的调试功能十分强大，本节只讲解了一些常用的调试功能，对于其他调试功能，还请读者自行尝试。

本节的参考链接如下。

- https://code.visualstudio.com/docs/editor/debugging
- https://code.visualstudio.com/docs/nodejs/nodejs-debugging

4.4 debug + repl2 + power–assert

在 4.3 节讲解了如何使用 VS Code 调试 Node.js 代码，但调试不只是打点，比如：

- 如何快速地切换输出的日志类型（或级别）？
- 我想用 moment 打印出年份，是使用 moment().format('YYYY') 还是使用 moment().format('yyyy')？还是两种写法都可以?
- 断言报错 AssertionError: false == true，但没有有用的信息，怎么调试?

本节将介绍三款实用的调试工具，分别解决以上三种问题，来提高我们的调试效率。

4.4.1 debug

debug 是一个小巧却非常实用的日志模块，可以根据环境变量打印不同类型（或级别）的日志，代码如下。

app.js：

```
const normalLog = require('debug')('log')
```

仅供非商业用途或交流学习使用


```
const errorLowLog = require('debug')('error:low')
const errorNormalLog = require('debug')('error:normal')
const errorHighLog = require('debug')('error:high')

setInterval(() => {
  const value = Math.random()
  switch (true) {
    case value < 0.5: normalLog(value); break
    case value >= 0.5 && value < 0.7: errorLowLog(value); break
    case value >= 0.7 && value < 0.9: errorNormalLog(value); break
    case value >= 0.9: errorHighLog(value); break
    default: normalLog(value)
  }
}, 1000)
```

运行上面的代码，可每一秒生成一个随机数，根据随机数的值模拟不同级别的日志输出。

- < 0.5：正常的日志。
- 0.5 ~ 0.7：低级别的错误日志。
- 0.7 ~ 0.9：一般级别的错误日志。
- ≥ 0.9：严重级别的错误日志。

运行：

```
$ DEBUG=* node app.js
```

打印的效果如图 4-10 所示。

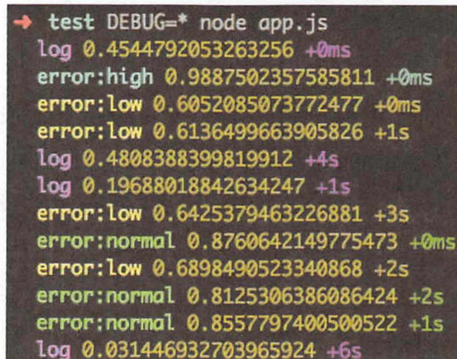


图 4-10

仅供非商业用途或交流学习使用

可以看出，debug 模块打印的日志与 console.log 相比，有以下几个特点。

- 对不同的日志类型分配了不同的颜色加以区分，更直观。
- 添加了日志类型的前缀。
- 添加了自上一次该类型的日志打印到这次日志打印经历了多长时间的后缀。

debug 模块支持以下用法。

- DEBUG=*：打印所有类型的日志。
- DEBUG=log：只打印 log 类型的日志。
- DEBUG=error:*：打印所有以 error: 开头的日志。
- DEBUG=error:*,-error:low：打印所有以 error: 开头并且过滤掉 error:low 类型的日志。

下面演示第 4 种用法，运行：

```
$ DEBUG=error:*,-error:low node app.js
```

打印的效果如图 4-11 所示。

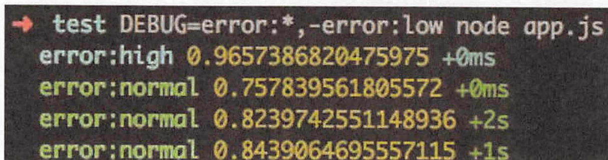


图 4-11

4.4.2 repl2

我们在写代码时，有时不太能记清楚某个模块的某个方法的具体用法，比如，用 moment 格式化年份，是用 moment().format('YYYY')，还是用 moment().format('yyyy')，还是两种写法都可以？ lodash 的 _.pick 方法能否接收数组作为参数？这时相对于翻阅官方文档，在 REPL 里试一下可能会更快，通常的步骤是：

```
$ npm i moment
$ node
> const moment = require('moment')
> moment().format('YYYY')
```

```
'2017'
> moment().format('yyyy')
'yyyy'
```

这种操作进行一次还好，次数多了也略微烦琐，repl2 模块应运而生。

repl2 顾名思义是 REPL 的增强版，会根据一个用户配置（~/.noderc）预先加载模块到 REPL 中，省去了我们手动在 REPL 中 require 模块的过程。

全局安装 repl2：

```
$ npm i repl2 -g
```

使用方式很简单，如下所述。

（1）将常用的模块进行全局安装，例如：

```
$ npm i lodash validator moment -g
```

（2）添加配置到 ~/.noderc：

```
{
  "lodash": "__",
  "moment": "moment",
  "validator": "validator"
}
```

（3）运行 noder：

```
$ noder
__ = lodash@4.17.4 -> local
moment = moment@2.18.1 -> global
validator = validator@7.0.0 -> global
> moment().format('YYYY')
'2017'
> __.random(0, 5)
3
> validator.isEmail('foo@bar.com')
true
```

这里需要讲解以下两点。

- ~/.noderc 是一个 JSON 文件，key 是模块的名字，value 是在 require 这个模块后加载到 REPL 中的变量名。这里给 lodash 命名的变量名是“__”而不是“_”，这是因为在 REPL 中“_”有特殊含义，表示上一个表达式的结果。
- repl2 会优先加载当前目录下的模块，如果没有找到，则再去加载全局安装的模块。上面的结果显示 lodash 是从本地目录加载的，因为在 test 目录下已经安装了 lodash，所以其余模块在从本地目录下没有找到时会尝试从全局 npm 目录下加载。如果都没有找到，则不会加载。

4.4.3 power-assert

我们常用的断言库有 should.js、expect.js 和 chai。但这类断言库都有一些通病，比如过分追求语义化、API 复杂、错误信息不足等。

先看一段代码。

test.js：

```
const assert = require('assert')
const should = require('should')
const expect = require('expect.js')

const tom = { id: 1, age: 18 }
const bob = { id: 2, age: 20 }

describe('app.js', () => {
  it('assert', () => {
    assert(tom.age > bob.age)
  })
  it('should.js', () => {
    tom.age.should.be.above(bob.age)
  })
  it('expect.js', () => {
    expect(tom.age).be.above(bob.age)
  })
})
```

运行：

```
$ mocha
```

结果如下：

```
app.js
  1) assert
  2) should.js
  3) expect.js


0 passing (13ms)
3 failing

1) app.js
     assert:

    AssertionError [ERR_ASSERTION]: false == true
    + expected - actual

    -false
    +true

    at Context.it (test.js:10:5)

2) app.js
     should.js:
   AssertionError: expected 18 to be above 20
    at Assertion.fail (node_modules/should/cjs/should.js:275:17)
    at Assertion.value (node_modules/should/cjs/should.js:356:19)
    at Context.it (test.js:13:23)

3) app.js
     expect.js:
   Error: expected 18 to be above 20
    at Assertion.assert (node_modules/expect.js/index.js:96:13)
    at Assertion.greaterThan.Assertion.above (node_modules/expect.js/index.
js:297:10)
    at Function.above (node_modules/expect.js/index.js:499:17)
    at Context.it (test.js:16:24)
```

可以看出，基本没有有用的信息。这时，power-assert 粉墨登场。

power-assert 使用起来很简单，理论上只用一个 assert 就可以了，而且可以无缝地迁移。

> **注 意**
>
> 在使用 intelli-espower-loader 时，必须将测试文件放到 test/ 目录下，所以我们在 test 目录下创建 test/app.js，将原来的 test.js 代码粘贴过去。

安装 power-assert 和 intelli-espower-loader，然后运行测试：

```
$ npm i power-assert intelli-espower-loader --save-dev
$ mocha -r intelli-espower-loader
```

结果如下：

app.js：

```
  1) assert
  2) should.js
  3) expect.js


0 passing (42ms)
3 failing

1) app.js
     assert:

    AssertionError [ERR_ASSERTION]:   # test/app.js:10

assert(tom.age > bob.age)
       |   |   | |   |
       |   |   | |   20
       |   |   | Object{id:2,age:20}
       |   18  false
       Object{id:1,age:18}

    + expected - actual

    -false
    +true
    ...
```

错误的信息非常直观，有以下两点需要说明。

- mocha 需要引入 intelli-espower-loader，主要用于转译代码，在转译后，require('assert') 都不需要更改。
- intelli-espower-loader 可有选择性地在 package.json 中添加 directories.test 配置，例如：

```
"directories": {
  "test": "mytest/"
}
```

如果没有 directories.test 配置，则默认是 test/。

本节的参考链接如下。

- https://zhuanlan.zhihu.com/p/25956323
- https://www.npmjs.com/package/intelli-espower-loader

4.5 supervisor-hot-reload

我们在本地开发 Node.js 程序时通常会使用 nodemon 或者 supervisor 这种进程管理工具，当有文件修改时自动重启应用。小项目还好，若项目大了（尤其是前端应用），则每次重启应用都会用几秒到几十秒，大部分时间都被花在了加载及编译代码上。

这让笔者联想到前端比较火的一个名词——Hot Reload（热加载），比如 React 静态资源的热加载通过 webpack-dev-server 和 react-hot-loader 实现，webpack-dev-server 负责重新编译代码，react-hot-loader 负责热加载。

那么在 Node.js 应用中如何实现 Hot Reload 呢？最好能实现不重启应用便使新代码生效。幸好 ES6 引入了一个新特性——Proxy。

4.5.1 Proxy

Proxy 用于修改对象的默认行为，等同于在语言层面做出修改，属于一种“元编程”。Proxy 在要访问的对象之前架设一层拦截，在要访问该对象成员时必须先经过这层拦截。示例代码如下：

```
const obj = new Proxy({}, {
  get: function (target, key) {
    console.log(`getting ${key}!`)
    return 'haha'
  }
})

console.log(obj.name)
// getting name!
// haha
console.log(obj.age)
// getting age!
// haha
```

可以看出，我们并没有在 obj 上定义 name 和 age 属性，在获取 obj 上的属性时都会执行 get 方法，然后打印 getting xxx! 和返回 haha。

以上代码中的 Proxy 的第 1 个参数是一个空对象，也可以是一个其他对象，比如函数（毕竟在 JavaScript 中函数也是对象）。

```
function user () {}

const obj = new Proxy(user, {
  get: function (target, key) {
    console.log(`getting ${key}!`)
    return 'haha'
  }
})

console.log(user.name)
// user
console.log(user.age)
// undefined
console.log(obj.name)
// getting name!
// haha
console.log(obj.age)
// getting age!
// haha
new Proxy(1, {})
// TypeError: Cannot create proxy with a non-object as target or handler
```

4.5.2 用 Proxy 实现 Hot Reload

用 Proxy 实现 Hot Reload 的核心原理是：使用 Proxy 将模块导出的对象包装一层“代理”，即 module.exports 导出的是一个 Proxy 实例。定义一个 get 方法，使得在获取实例上的属性时其实是获取最新的 require.cache 中的对象上的属性；同时，监听代码文件，如果有修改，则更新 require.cache。

简而言之，我们在获取对象的属性时在中间加了一层代理，通过代理间接地获取原有属性的值，如果属性值有更新，则会更新 require.cache 的缓存；若下次再获取对象的属性，则通过代理获取该属性的最新的值。可见，Proxy 可以实现属性访问拦截，也可以实现断开强引用。

笔者发布了一个 proxy-hot-reload 模块，核心代码如下：

```
module.exports = function proxyHotReload(opts) {
  const includes = [ ... ]
  const excludes = [ ... ]
  const filenames = _.difference(includes, excludes)

  chokidar
    .watch(filenames, {
      usePolling: true
    })
    .on('change', (path) => {
      try {
        if (require.cache[path]) {
          const _exports = require.cache[path].exports
          if (_.isPlainObject(_exports) && !_.isEmpty(_exports)) {
            delete require.cache[path]
            require(path)
          }
        }
      } catch (e) { ... }
    })
    .on('error', (error) => console.error(error))

  shimmer.wrap(Module.prototype, '_compile', function (__compile) {
    return function proxyHotReloadCompile(content, filename) {
      if (!_.includes(filenames, filename)) {
        try {
```

```
        return __compile.call(this, content, filename)
      } catch (e) { ... }
    } else {
      const result = __compile.call(this, content, filename)
      this._exports = this.exports
      // non-object return original compiled code
      if (!_.isPlainObject(this._exports)) {
        return result
      }
      try {
        this.exports =  new Proxy(this._exports, {
          get: function (target, key, receiver) {
            try {
              if (require.cache[filename]) {
                return require.cache[filename]._exports[key]
              } else {
                return Reflect.get(target, key, receiver)
              }
            } catch (e) { ... }
          }
        })
      } catch (e) { ... }
    }
  }
 })
}
```

对以上代码简单讲解如下。

- 可传入 includes 和 excludes 参数，支持 glob 写法，用来设置监听哪些代码文件。
- 用 chokidar 模块监听文件，如果有改动，则重新加载该文件。这里只针对 module.exports 导出的纯对象的模块有用。做这个限制的原因是：对于非对象如函数，我们一般在导出一个函数时直接调用执行而不是获取函数上的属性或方法，这种导出非纯对象模块的方式即使重建缓存也不会生效，所以干脆忽略。幸运的是，在大多数场景下，module.exports 导出的是对象。
- 用 shimmer 模块重载 Module.prototype._compile 方法，如果是被监听的文件并且导出的是纯对象，则尝试将导出的对象包装成 Proxy 实例。这样，在获取该对象上的属性时，将从 require.cache 中读取最新的值。

使用示例如下。

user.js：

```
module.exports = {
  id: 1,
  name: 'nswbmw'
}
```

app.js：

```
if (process.env.NODE_ENV !== 'production') {
  require('proxy-hot-reload')({
    includes: '**/*.js'
  })
}

const Paloma = require('paloma')
const app = new Paloma()
const user = require('./user')

app.route({ method: 'GET', path: '/', controller (ctx) {
  ctx.body = user
}})

app.listen(3000)
```

浏览器将访问 localhost:3000 来查看结果，修改 user.js 中的字段的值，然后刷新浏览器来查看结果。

proxy-hot-reload 有个非常明显的缺点：只支持对导出的纯对象的文件做代理，而且程序入口文件不会生效，比如上面的 app.js，修改端口号只能在重启后生效。Proxy 再怎么“黑魔法”也只能做到这个地步了。退一步想，如果修改了 proxy-hot-reload 覆盖不到的文件（例如 app.js）并降级成自动重启就好了，如果将 proxy-hot-reload 和 supervisor 结合，则会怎样呢?

4.5.3 supervisor-hot-reload

如果要将 proxy-hot-reload 结合 supervisor 使用，则需要解决以下几个难点。

（1）非侵入式，即在代码里不再写：

```
if (process.env.NODE_ENV !== 'production') {
  require('proxy-hot-reload')({
    includes: '**/*.js'
  })
}
```

（2）参数统一。supervisor 可接收的 -w 参数表明监听哪些文件；-i 参数表明忽略哪些文件。这两个参数怎么与 proxy-hot-reload 的 includes 和 excludes 参数整合呢?

（3）职责分明。在修改代码文件并保存后，优先尝试 proxy-hot-reload 的热更新，如果 proxy-hot-reload 热更新不了，则使用 supervisor 重启。

首先，我们来看下 supervisor 的源码（lib/supervisor.js），在源码中有以下代码，作用是遍历找到所有需要监听的文件，然后调用 watchGivenFile 监听文件：

```
var watchItems = watch.split(',');
watchItems.forEach(function (watchItem) {
    watchItem = path.resolve(watchItem);

    if ( ! ignoredPaths[watchItem] ) {
        log("Watching directory '" + watchItem + "' for changes.");
        if(interactive) {
            log("Press rs for restarting the process.");
        }
        findAllWatchFiles(watchItem, function(f) {
            watchGivenFile( f, poll_interval );
        });
    }
});
```

watchGivenFile 的代码如下：

```
function watchGivenFile (watch, poll_interval) {
    if (isWindowsWithoutWatchFile || forceWatchFlag) {
        fs.watch(watch, { persistent: true, interval: poll_interval },
crashWin);
    } else {
        fs.watchFile(watch, { persistent: true, interval: poll_interval },
function(oldStat, newStat) {
            // we only care about modification time, not access time.
            if ( newStat.mtime.getTime() !== oldStat.mtime.getTime() ) {
                if (verbose) {
```

仅供非商业用途或交流学习使用


```
                    log("file changed: " + watch);
                }
            }
            crash();
        });
    }
    if (verbose) {
        log("watching file '" + watch + "'");
    }
}
```

watchGivenFile 的作用是：用 fs.watch/fs.watchFile 监听文件，如果有改动则调用 crashWin/crash 程序退出。supervisor 使用 child_process.spawn 将程序运行在子进程中，在子进程退出后会被 supervisor 重新启动。相关代码如下：

```
function startProgram (prog, exec) {
    var child = exports.child = spawn(exec, prog, {stdio: 'inherit'});
    ...
    child.addListener("exit", function (code) {
        ...
        startProgram(prog, exec);
    });
}
```

在大体理清 supervisor 的关键源码后，我们就知道如何解决上面提到的几个难点了。

首先需要修改 proxy-hot-reload，添加以下几个功能。

- 添加 includeFiles 和 excludeFiles 选项，值为数组，用来接收 supervisor 传来的文件列表。
- 添加 watchedFileChangedButNotReloadCache 参数。proxy-hot-reload 可以知道哪些代码文件可以热更新，哪些不可以。当监听到不能热更新的文件有修改时，则调用 watchedFileChangedButNotReloadCache 函数，在这个函数里有 process.exit() 可以使进程退出。

相应的解决方案如下。

（1）非侵入式。因为真正的程序试运行在 supervisor 创建的子进程中，所以我们无法在 supervisor 进程中引入 proxy-hot-reload，只能通过子进程用 node -r xxx 提前引入并覆盖 Module.prototype._compile。解决方案为：将 supervisor 需要监听的文件数组

仅供非商业用途或交流学习使用

（watchFiles）和 proxy-hot-reload 配置写到一个文件（例如 proxy-hot-reload.js）中，子进程通过 node -r proxy-hot-reload.js app.js 预加载此文件并启动。

supervisor 的相关代码如下：

```
// 获取 watchFiles
fs.writeFileSync(path.join(__dirname, 'proxy-hot-reload.js'), `
    require('${path.join(__dirname, "..", "node_modules", "proxy-hot-
reload")}')({
    includeFiles: ${JSON.stringify(watchFiles)},
    excludeFiles: [],
    watchedFileChangedButNotReloadCache: function (filename) {
        console.log(filename + ' changed, restarting...');
        setTimeout(function () {
            process.exit();
        }, ${poll_interval});
    }
});`);
// startChildProcess()
```

（2）参数统一。将上面的在 watchItems.forEach 内异步遍历时需要监听的文件列表修改为同步，代码如下：

```
var watchFiles = []
var watchItems = watch.split(',');
watchItems.forEach(function (watchItem) {
    watchItem = path.resolve(watchItem);
    if ( ! ignoredPaths[watchItem] ) {
        log("Watching directory '" + watchItem + "' for changes.");
        if(interactive) {
            log("Press rs for restarting the process.");
        }
        findAllWatchFiles(watchItem, function(f) {
            watchFiles.push(f)
            // watchGivenFile( f, poll_interval );
        });
    }
});
```

仅供非商业用途或交流学习使用

注 意

这里 findAllWatchFiles 虽然有回调函数，却是同步的。将 findAllWatchFiles 内的 fs.lstat/fs.stat/fs.readdir 分别改为 fs.lstatSync/fs.statSync/fs.readdirSync，这里就不贴代码了。

（3）职责分明。在子进程使用 node -r proxy-hot-reload.js app.js 启动后，能热更新的则热更新，不能热更新的则执行 watchedFileChangedButNotReloadCache，子进程退出，supervisor 会启动一个新的子进程，实现了职责分明。

笔者将改进后的 supervisor 发布成一个新的包——supervisor-hot-reload 。使用方法如下。

user.js：

```
module.exports = {
  id: 1,
  name: 'nswbmw'
}
```

app.js：

```
const Paloma = require('paloma')
const app = new Paloma()
const user = require('./user')

app.route({ method: 'GET', path: '/', controller (ctx) {
  ctx.body = user
}})

app.listen(3000)
```

全局安装并使用 supervisor-hot-reload：

```
$ npm i supervisor-hot-reload -g
$ DEBUG=proxy-hot-reload supervisor-hot-reload app.js
```

仅供非商业用途或交流学习使用

修改 user.js，程序不会重启，打印：

```
proxy-hot-reload Reload file: /Users/nswbmw/Desktop/test/user.js
```

修改 app.js，程序会重启，打印：

```
/Users/nswbmw/Desktop/test/app.js changed, restarting...
Program node app.js exited with code 0

Starting child process with 'node app.js'
...
```

4.5.4 内存泄漏问题

这里需要声明一下，虽然修改 require.cache + Proxy 实现了我们想要的功能，但这样做存在内存泄漏问题，因为即使删除了一个模块的缓存，但在父模块的缓存中还引用着旧的模块导出的对象。对于这个问题，我们不用太关心，知道为什么就好，因为我们只是在开发环境中使用 proxy-hot-reload。

本节的参考链接如下。

- https://nodejs.org/dist/latest-v8.x/docs/api/async_hooks.html

第5章

日志

在应用发生问题后，日志通常是我们的救命稻草，我们可以在事后通过分析日志定位问题原因和代码。如何做日志打点则又是个哲学问题了，本章将会以自动打点为基础，结合多种工具收集和可视化日志。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

5.1 koa-await-breakpoint

日志打点一直是调试工作中最让人头疼的问题，如果直接在代码中插入埋点代码，则不仅侵入性强，而且工作量大，也不够灵活。如果能做到智能打点，就会让调试工作轻松许多，koa-await-breakpoint 正是我们需要的。

5.1.1 koa-await-breakpoint 的实现原理

koa-await-breakpoint 是 Koa 的一个中间件，是一个在 routes/controllers 里（作用域包含 ctx）await 表达式前后自动打点的工具，不用插入一行日志打点代码，只需在引入时配置一下，就可以记录每个请求到来时 await 表达式前后的现场，例如：

- await 表达式所在的文件及行列号（filename）；
- await 表达式执行的是第几步（step）；
- await 表达式的字符串形式（fn）；
- 执行 await 表达式所花费的毫秒（take）；
- 执行 await 表达式的结果（result）；
- 当前请求的 ctx。

使用方法如下：

```
// On top of the main file
const koaAwaitBreakpoint = require('koa-await-breakpoint')({
  name: 'api',
  files: ['./routes/*.js']
})

const Koa = require('koa')
const app = new Koa()

// Generally, above other middlewares
app.use(koaAwaitBreakpoint)
...

app.listen(3000)
```

重载 Module.prototype._compile 相当于 hack 了 require，如果发现 require 了配置里

仅供非商业用途或交流学习使用

指定的文件，则进行下一步，否则返回原始代码的内容，相关源码如下：

```
shimmer.wrap(Module.prototype, '_compile', function (__compile) {
  return function koaBreakpointCompile(content, filename) {
    if (!_.includes(filenames, filename)) {
      return __compile.call(this, content, filename);
    }
    ...
  };
});
```

用 esprima 解析代码，生成 AST。例如：

```
const Mongolass = require('mongolass')
const mongolass = new Mongolass('mongodb://localhost:27017/test')
const User = mongolass.model('users')

exports.getUsers = async function getUsers(ctx) {
  await User.create({
    name: 'xx',
    age: 18
  })
  const users = await User.find()
  return users
}
```

则会生成如下 AST，只截取了 await User.create(...) 相关的 AST：

```
Script {
    ...
    AwaitExpression {
      type: 'AwaitExpression',
      argument:
       CallExpression {
         type: 'CallExpression',
         callee:
          StaticMemberExpression {
            type: 'MemberExpression',
            object:
             Identifier {
               type: 'Identifier',
               name: 'User'},
            property:
```

仅供非商业用途或交流学习使用


```
                Identifier {
                  type: 'Identifier',
                  name: 'create' } },
            arguments:
             [ ObjectExpression {
                 type: 'ObjectExpression',
                 properties:
                  [ Property {
                      type: 'Property',
                      key:
                       Identifier {
                         type: 'Identifier',
                         name: 'name' },
                      computed: false,
                      value:
                       Literal {
                         type: 'Literal',
                         value: 'xx',
                         raw: '\'xx\'' } },
                    Property { ... } ]
    ...
```

遍历找到 awaitExpression 节点，在进行以下包装后生成 AST，替换原来的节点：

```
global.logger(
  (typeof ctx !== 'undefined' ? ctx : this),
  function(){
    return awaitExpression
  },
  awaitExpressionString,
  filename
)
```

相关源码如下：

```
findAwaitAndWrapLogger(parsedCodes)
try {
  content = escodegen.generate(parsedCodes, {
    format: { indent: { style: '  ' } },
    sourceMap: filename,
    sourceMapWithCode: true
  })
} catch(e) {
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    console.error('cannot generate code for file: %s', filename)
    console.error(e.stack)
    process.exit(1)
  }
  debug('file %s regenerate codes:\n%s', filename, content.code)
```

findAwaitAndWrapLogger 的作用就是遍历 AST，将 awaitExpression 替换成用日志函数包裹后的新的 awaitExpression 的 AST。最后用 escodegen 将 AST 生成代码（支持 soucemap，所以错误栈对应的行数是正确的）。

其核心内容为：在每个请求到来时生成一个 requestId（可自定义，默认为 uuid）并将其挂载到 ctx 上，这样就可以通过 requestId 将日志串起来了。其特点是可以记录每个请求的每一步（await 表达式）的现场及返回值，方便查找日志。

5.1.2　使用 koa-await-breakpoint

测试代码如下。

app.js：

```
const koaAwaitBreakpoint = require('koa-await-breakpoint')({
  name: 'api',
  files: ['./routes/*.js']
})

const Paloma = require('paloma')
const app = new Paloma()
const userRouter = require('./routes/user')

app.use(koaAwaitBreakpoint)
app.route({ method: 'GET', path: '/users', controller: userRouter.getUsers })

app.listen(3000)
```

routes/user.js：

```
const Mongolass = require('mongolass')
const mongolass = new Mongolass('mongodb://localhost:27017/test')
const User = mongolass.model('users')
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
exports.getUsers = async function getUsers (ctx) {
  await User.create({
    name: 'xx',
    age: 18
  })

  const users = await User.find()
  ctx.body = users
}
```

运行：

```
$ DEBUG=koa-await-breakpoint node app.js
```

终端会打印出转换后的代码，可以看出 routes/users.js 被转换成了：

```
const Mongolass = require('mongolass');
const mongolass = new Mongolass('mongodb://localhost:27017/test');
const User = mongolass.model('users');
exports.getUsers = async function getUsers(ctx) {
  await global.logger(typeof ctx !== 'undefined' ? ctx : this, function () {
    return User.create({
      name: 'xx',
      age: 18
    });
  }, 'User.create({\n    name: \'xx\',\n    age: 18\n})', '/Users/nswbmw/
Desktop/test/routes/user.js:6:2');
  const users = await global.logger(typeof ctx !== 'undefined' ? ctx : this,
function () {
    return User.find();
  }, 'User.find()', '/Users/nswbmw/Desktop/test/routes/user.js:11:16');
  ctx.body = users;
};
```

访问 localhost:3000/users，终端打印出：

```
{"name":"api","requestId":"50dbda0c-9e13-4659-acce-
b237bc5178b7","timestamp":"2018-02-26T06:31:31.100Z","this":...,"type":"start
","step":1,"take":0}
{"name":"api","requestId":"50dbda0c-9e13-4659-acce-b237bc5178b7","step":2,"
filename":"/Users/nswbmw/Desktop/test/routes/user.js:6:2","timestamp":"2018-
02-26T06:31:31.104Z","this":...,"type":"beforeAwait","fn":"User.create({\n
name: 'xx',\n    age: 18\n})","take":4}
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
{"name":"api","requestId":"50dbda0c-9e13-4659-acce-b237bc5178b7","step":3,"
filename":"/Users/nswbmw/Desktop/test/routes/user.js:6:2","timestamp":"2018-
02-26T06:31:31.175Z","this":...,"type":"afterAwait","fn":"User.create({\n
name:  'xx',\n     age: 18\n})","result":{"result":{"ok":1,"n":1},"ops":[{"name
":"xx","age":18,"_id":"5a93a9c3cf8c8797c9b47482"}],"insertedCount":1,"inserte
dIds":["5a93a9c3cf8c8797c9b47482"]},"take":71}
{"name":"api","requestId":"50dbda0c-9e13-4659-acce-b237bc5178b7","step":4,"fi
lename":"/Users/nswbmw/Desktop/test/routes/user.js:11:16","timestamp":"2018-
02-26T06:31:31.175Z","this":...,"type":"beforeAwait","fn":"User.
find()","take":0}
{"name":"api","requestId":"50dbda0c-9e13-4659-acce-b237bc5178b7","step":5,"fi
lename":"/Users/nswbmw/Desktop/test/routes/user.js:11:16","timestamp":"2018-
02-26T06:31:31.180Z","this":...,"type":"afterAwait","fn":"User.
find()","result":[{"_id":"5a93a9c3cf8c8797c9b47482","name":"xx","age":18}],"ta
ke":5}
{"name":"api","requestId":"50dbda0c-9e13-4659-acce-
b237bc5178b7","timestamp":"2018-02-26T06:31:31.181Z","this":...,"type":"end",
"step":6,"take":1}
```

注　意

type 是以下中的一种，take 的单位是 ms。

- start：请求到来时的第 1 次打点。
- beforeAwait：从上一个 awaitExpression 之后到这个 awaitExpression 之前。
- afterAwait：从这个 awaitExpression 开始到结束。
- error：错误日志，包含了错误信息。
- end：在请求结束时打点。

5.1.3　自定义日志存储

在使用 koa-await-breakpoint 时最好自己定义 store 参数（默认打印日志到 stdout），该参数是一个对象并且有一个 save 方法即可。在 save 方法内可做一些逻辑修改或者日志策略，比如：

- 添加日志标识（如 name），以方便区分不同服务的日志；
- 针对错误日志，添加一些额外的字段以方便追踪现场；
- 将日志发送到 Logstash 或其他日志服务；

书籍是人类进步的阶梯

- 限制日志频率，比如只有响应时间大于 500ms 的请求日志才会被记录。

示例代码如下。

koa_await_breakpoint_store.js：

```
exports.save = function save(record, ctx) {
  record.name = 'app name'
  record.env = process.env.NODE_ENV

  if (record.error) {
    record.error = {
      message: record.error.message,
      stack: record.error.stack,
      status: record.error.status || record.error.statusCode || 500
    }
  }
  ...
  logstash.send(record)
}
```

本节的参考链接如下。

- https://github.com/jquery/esprima
- https://github.com/estools/escodegen

5.2 使用 async_hooks

6.1 节讲解了 koa-await-breakpoint 的用法，但 koa-await-breakpoint 仍然有一个很大的缺憾，即无法记录除 routes/controllers 外的函数的执行时间（因为获取不到当前请求的 ctx）。举个通俗的例子：在一个路由的 controller 里面调用了 A，A 调用了其他文件的 B，B 又调用了其他文件的 C……这是非常常见的用法，但之前使用 koa-await-breakpoint 只能获取 A 的执行时间，无法获取 B 和 C 的执行时间。

其根本原因在于：无法知道函数之间的调用关系，即 B 不知道是 A 调用的它，即便知道也不知道是哪次请求到来时执行的 A 调用的它。

书籍是人类进步的阶梯

但是，node@8.1 引入了一个黑魔法——async_hooks，async_hooks 用来追踪 Node.js 中异步资源的生命周期。

先来看一段测试代码：

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

async_hooks.createHook({
  init (asyncId, type, triggerAsyncId, resource) {
    fs.writeSync(1, `${type}(${asyncId}): trigger: ${triggerAsyncId}\n`)
  },
  destroy (asyncId) {
    fs.writeSync(1, `destroy: ${asyncId}\n`);
  }
}).enable()

async function A () {
  fs.writeSync(1, `A -> ${async_hooks.executionAsyncId()}\n`)
  setTimeout(() => {
    fs.writeSync(1, `A in setTimeout -> ${async_hooks.executionAsyncId()}\n`)
    B()
  })
}

async function B () {
  fs.writeSync(1, `B -> ${async_hooks.executionAsyncId()}\n`)
  process.nextTick(() => {
    fs.writeSync(1, `B in process.nextTick -> ${async_hooks.
executionAsyncId()}\n`)
    C()
    C()
  })
}

function C () {
  fs.writeSync(1, `C -> ${async_hooks.executionAsyncId()}\n`)
  Promise.resolve().then(() => {
    fs.writeSync(1, `C in promise.then -> ${async_hooks.
executionAsyncId()}\n`)
  })
}
```

书籍是人类进步的阶梯

```
fs.writeSync(1, `top level -> ${async_hooks.executionAsyncId()}\n`)
A()
```

async_hooks.createHook 可以注册 4 个方法来跟踪所有异步资源的初始化（init）、回调之前（before）、回调之后（after）、销毁后（destroy）事件，并通过调用 .enable() 启用，通过调用 .disable() 关闭。

这里我们只关心异步资源的初始化和销毁的事件，并使用 fs.writeSync(1, msg) 打印到标准输出，writeSync 的第 1 个参数接收文件描述符，1 表示标准输出。为什么不使用 console.log 呢？因为 console.log 是一个异步操作，如果在 init、before、after 和 destroy 事件处理函数中出现，就会导致无限循环，同理，也不能使用任何其他异步操作。

运行该程序，打印如下：

```
top level -> 1
PROMISE(6): trigger: 1
A -> 1
Timeout(7): trigger: 1
TIMERWRAP(8): trigger: 1
A in setTimeout -> 7
PROMISE(9): trigger: 7
B -> 7
TickObject(10): trigger: 7
B in process.nextTick -> 10
C -> 10
PROMISE(11): trigger: 10
PROMISE(12): trigger: 11
C -> 10
PROMISE(13): trigger: 10
PROMISE(14): trigger: 13
C in promise.then -> 12
C in promise.then -> 14
destroy: 7
destroy: 10
destroy: 8
```

这段程序的打印结果包含了很多信息，下面逐一进行解释。

- 为了实现对异步资源的跟踪，Node.js 对每一个函数（不论异步还是同步）提供了一个 async scope，我们可以通过调用 async_hooks.executionAsyncId() 来

书籍是人类进步的阶梯

获取函数当前的 async scope 的 id（称之为 asyncId），通过调用 async_hooks.triggerAsyncId() 来获取当前函数调用者的 asyncId。

- 异步资源在创建时触发 init 事件函数，init 函数中的第 1 个参数代表该异步资源的 asyncId；type 代表异步资源的类型（例如 TCPWRAP、PROMISE、Timeout、Immediate、TickObject 等）；triggerAsyncId 代表该异步资源的调用者的 asyncId。异步资源在销毁时触发 destroy 事件函数，该函数只接收一个参数，即该异步资源的 asyncId。
- 函数调用关系明确。我们通过上面的打印结果可以很容易地看出（从下往上看），C（asyncId: 10）被 B（asyncId: 7）调用，B（asyncId: 7）被 A（asyncId: 1）调用，而且 C 的 promise.then 里的 asyncId（值为 12/14）也可以通过 12/14 -> 11/13 -> 10 定位到 C 的 asyncId（值为 10）。
- 同步函数每次调用的 asyncId 都一样，如上面的代码所示，C 调用了两次，都打印了 C -> 10，与调用方的作用域的 asyncId 一致，即在以上代码中打印的 B in process.nextTick -> 10。异步函数每次调用的 asyncId 都不一样，即在以上代码中打印的 C in promise.then -> 12 和 C in promise.then -> 14。
- 最外层作用域的 asyncId 总是 1，每个异步资源在创建时 asyncId 全局递增。

上面的结论非常重要。接下来我们看看如何使用 async_hooks 改造 koa-await-breakpoint。

我们通过前面的结论已经知道，在使用 async_hooks 时可以通过 asyncId 串起函数的调用关系，但是如何将这些函数的调用链与 Koa 接收的每个请求关联起来呢？

首先，定义一个全局 Map，存储函数的调用关系：

```
const async_hooks = require('async_hooks')
const asyncIdMap = new Map()

async_hooks.createHook({
  init (asyncId, type, triggerAsyncId) {
    const ctx = getCtx(triggerAsyncId)
    if (ctx) {
      asyncIdMap.set(asyncId, ctx)
    } else {
      asyncIdMap.set(asyncId, triggerAsyncId)
    }
  },
```

书籍是人类进步的阶梯

```
    destroy (asyncId) {
      asyncIdMap.delete(asyncId)
    }
  }).enable()

  function getCtx (asyncId) {
    if (!asyncId) {
      return
    }
    if (typeof asyncId === 'object' && asyncId.app) {
      return asyncId
    }
    return getCtx(asyncIdMap.get(asyncId))
  }
```

有以下三点需要解释。

- 定义了一个全局 Map 来存储函数的调用关系，在适当的地方（下面会讲到）将当前请求的 ctx 存储到 Map 中，key 是 asyncId。
- 每个异步资源在初始化时，会尝试通过 asyncId 向上寻找祖先的 value 是否是 ctx（koa 应用中每个请求的 ctx），如果是，则直接将 value 设置为 ctx，否则将 value 设置为调用者的 asyncId（即 triggerAsyncId）。
- 在 destroy 事件函数里直接删除调用关系，可保证不会引起内存泄漏，即杜绝引用了 ctx 但没有释放的情况。

然后，修改 global[loggerName] 如下：

```
global[loggerName] = async function (ctx, fn, fnStr, filename) {
  const originalContext = ctx
  let requestId = _getRequestId()

  const asyncId = async_hooks.executionAsyncId()
  if (!requestId) {
    const _ctx = getCtx(asyncId)
    if (_ctx) {
      ctx = _ctx
      requestId = _getRequestId()
    }
  } else {
    asyncIdMap.set(asyncId, ctx)
  }
```

书籍是人类进步的阶梯

```
    if (requestId) {
      _logger('beforeAwait')
    }
    const result = await fn.call(originalContext)
    if (requestId) {
      _logger('afterAwait', result)
    }
    return result

    function _getRequestId () {
      return ctx && ctx.app && _.get(ctx, requestIdPath)
    }

    function _logger (type, result) {
      ...
    }
  }
```

有以下两点需要解释。

- logger 函数传入的第 1 个参数 ctx，之前是每个请求的 ctx，现在可能是当前执行上下文的 this，所以先将 ctx 赋值给 originalContext，然后通过 await fn.call(originalContext) 让函数在执行时有正确的上下文。
- 如果传入的 ctx 是来自请求的 ctx 且能拿到 requestId，那么将当前的 asyncId 和 ctx 写入 Map；如果不是来自请求的 ctx，则尝试从 Map 里向上寻找祖先的 value 是否是 ctx。如果找到，则覆盖当前的 ctx 并拿到 requestId。

至此，koa-await-breakpoint 全部改造完毕。接下来我们通过一个例子验证升级后的 koa-await-breakpoint。

app.js：

```
const koaAwaitBreakpoint = require('koa-await-breakpoint')({
  files: ['./routes/*.js']
})

const Paloma = require('paloma')
const app = new Paloma()

app.use(koaAwaitBreakpoint)
app.route({ method: 'POST', path: '/users', controller: require('./routes/
```

```
user').createUser })

app.listen(3000)
```

routes/users.js：

```
const Mongolass = require('mongolass')
const mongolass = new Mongolass('mongodb://localhost:27017/test')
const User = mongolass.model('User')
const Post = mongolass.model('Post')
const Comment = mongolass.model('Comment')

exports.createUser = async function (ctx) {
  const name = ctx.query.name || 'default'
  const age = +ctx.query.age || 18
  await createUser(name, age)
  ctx.status = 204
}

async function createUser (name, age) {
  const user = (await User.create({
    name,
    age
  })).ops[0]
  await createPost(user)
}

async function createPost (user) {
  const post = (await Post.create({
    uid: user._id,
    title: 'post',
    content: 'post'
  })).ops[0]

  await createComment(user, post)
}

async function createComment (user, post) {
  await Comment.create({
    userId: user._id,
    postId: post._id,
    content: 'comment'
  })
}
```

以上代码的意思是：在访问创建用户接口时调用了 createUser，在 createUser 里又调用了 createPost，在 createPost 里又调用了 createComment。

运行：

```
$ curl -XPOST localhost:3000/users
```

打印如下：

```
{ type: 'start',
  step: 1,
  take: 0 ... }
{ type: 'beforeAwait',
  step: 2,
  fn: 'createUser(name, age)',
  take: 1 ... }
{ type: 'beforeAwait',
  step: 3,
  fn: 'User.create(...)',
  take: 1 ... }
{ type: 'afterAwait',
  step: 4,
  fn: 'User.create(...)',
  take: 36 ... }
{ type: 'beforeAwait',
  step: 5,
  fn: 'createPost(user)',
  take: 1 ... }
{ type: 'beforeAwait',
  step: 6,
  fn: 'Post.create(...)',
  take: 0 ... }
{ type: 'afterAwait',
  step: 7,
  fn: 'Post.create(...)',
  take: 3 ... }
{ type: 'beforeAwait',
  step: 8,
  fn: 'createComment(user, post)',
  take: 1 ... }
{ type: 'beforeAwait',
  step: 9,
```

```
    fn: 'Comment.create(...)',
    take: 0 ... }
  { type: 'afterAwait',
    step: 10,
    fn: 'Comment.create(...)',
    take: 1 ... }
  { type: 'afterAwait',
    step: 11,
    fn: 'createComment(user, post)',
    take: 1 ... }
  { type: 'afterAwait',
    step: 12,
    fn: 'createPost(user)',
    take: 6 ... }
  { type: 'afterAwait',
    step: 13,
    fn: 'createUser(name, age)',
    take: 44 ... }
  { type: 'end',
    step: 14,
    take: 0 ... }
```

至此，一个全链路、无侵入、强大的日志打点工具就完成了。

注 意

使用 async_hooks 在目前有较严重的性能损耗，可参见 https://github.com/bmeurer/async-hooks-performance-impact，在生产环境中请慎重使用。

本节的参考链接如下。

- https://nodejs.org/dist/latest-v8.x/docs/api/async_hooks.html
- https://zhuanlan.zhihu.com/p/27394440
- https://www.jianshu.com/p/4a568dac41ed

5.3 ELK

ELK 是 Elasticsearch + Logstash + Kibana 这套组合工具的简称，是一个常用的日志系统。其中，Elasticsearch 是一款开源的基于 Lucene 实现的一个分布式搜索引擎，也是一个存储引擎（例如日志），它的特点有：分布式、零配置、自动发现、索引自动分片、索引副本机制、有 Restful 风格的接口、多数据源和自动搜索负载等；Logstash 是一款开源的日志收集工具，可以对日志进行收集、分析、过滤，并将其存储（如 Elasticsearch）起来供以后使用；Kibana 是一款开源的可视化工具，可以为 Elasticsearch 提供日志分析友好的 Web 界面，可以汇总、分析和搜索重要的数据日志。

5.3.1 安装 ELK

使用 Docker 安装 ELK，运行如下命令：

```
$ docker run -p 5601:5601 \
    -p 9200:9200 \
    -p 5044:5044 \
    -p 15044:15044/udp \
    -it --name elk sebp/elk
```

进入容器：

```
$ docker exec -it elk /bin/bash
```

运行以下命令，设置 logstash 的 input 和 output：

```
# /opt/logstash/bin/logstash --path.data /tmp/logstash/data \
  -e 'input { udp { codec => "json" port => 15044 } } output { elasticsearch {
hosts => ["localhost"] } }'
```

这里启动了一个 15044 的 UDP 端口，用来接收通过 UDP 发送到 Logstash 的日志。

用浏览器打开 localhost:5601，效果如图 5-1 所示。

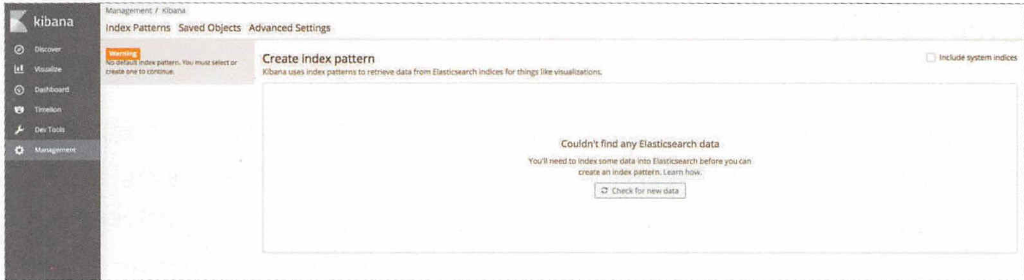


图 5-1

目前还没有指定 index（Elasticsearch 的 index 类似于 MySQL 或 MongoDB 中的 database），即日志来源。下面我们尝试向 ELK 中写入一些日志。

5.3.2 使用 ELK

这里仍然以使用 koa-await-breakpoint 为例，来演示如何将日志发送到 ELK。

app.js：

```
const koaAwaitBreakpoint = require('koa-await-breakpoint')({
  name: 'api',
  files: ['./routes/*.js'],
  store: require('./logger')
})

const Paloma = require('paloma')
const app = new Paloma()

app.use(koaAwaitBreakpoint)
app.route({ method: 'POST', path: '/users', controller: require('./routes/
user').createUser })

app.listen(3000)
```

logger.js：

```
const Logstash = require('logstash-client')

const logstash = new Logstash({
```

```
    type: 'udp',
    host: 'localhost',
    port: 15044
  })

  module.exports = {
    save (log) {
      if (log.error) {
        log.errMsg = log.error.message
        log.errStack = log.error.stack
      }
      logstash.send(log)
    }
  }
```

routes/user.js：

```
const Mongolass = require('mongolass')
const mongolass = new Mongolass('mongodb://localhost:27017/test')
const User = mongolass.model('User')
const Post = mongolass.model('Post')
const Comment = mongolass.model('Comment')

exports.createUser = async function (ctx) {
  const name = ctx.query.name || 'default'
  const age = +ctx.query.age || 18
  await createUser(name, age)
  ctx.status = 204
}

async function createUser (name, age) {
  const user = (await User.create({
    name,
    age
  })).ops[0]
  await createPost(user)
}

async function createPost (user) {
  const post = (await Post.create({
    uid: user._id,
    title: 'post',
    content: 'post'
```

```
    })).ops[0]

    await createComment(user, post)
  }

  async function createComment (user, post) {
    await Comment.create({
      userId: user._id,
      postId: post._id,
      content: 'comment'
    })
  }
```

运行：

```
$ curl -XPOST localhost:3000/users
```

在此时刷新 Kibana，如图 5-2 所示。

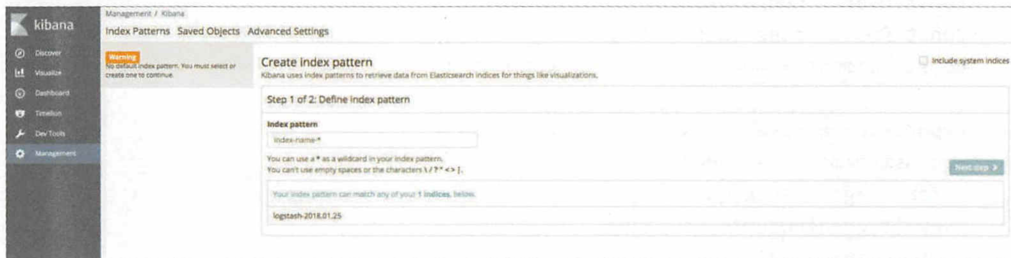


图 5-2

在初次使用 Kibana 时，需要配置 Kibana 从 Elasticsearch 的哪些 index 中搜索日志，我们在 Index pattern 处填 logstash-*，然后单击 Next step 按钮，在 Time Filter field name 中选择 timestamp，单击 Create index pattern 完成配置。

注意

我们选择 timestamp 而不是默认的 @timestamp，是因为在 koa-await-breakpoint 的日志中有 timestamp 字段。

单击左侧目录的 Discover，我们发现已经有日志了。分别单击左侧出现的 Available

Fields 的 fn、type、step、take，然后按 step 升序展示，如图 5-3 所示。

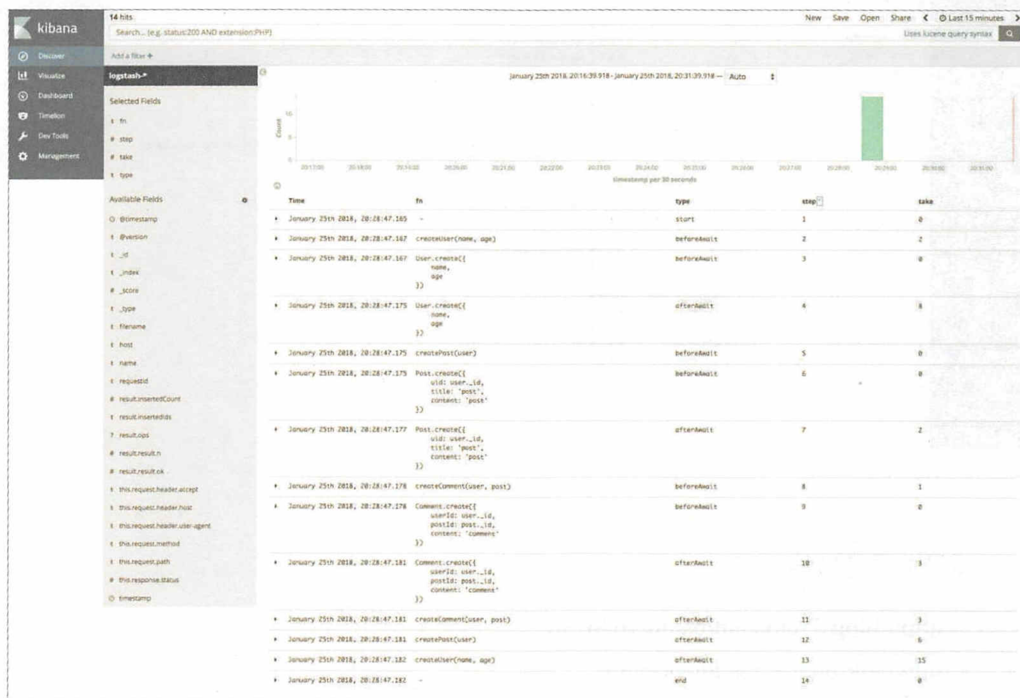


图 5-3

是不是一目了然？我们把每个请求的每一步的函数及其执行时间都记录下来了。

修改 routes/users.js 的 createComment，抛出一个 new Error('test')。重启程序并发起一个请求，ELK 界面如图 5-4 所示。

注 意

在实际应用中会有非常多的日志，我们可以通过 requestId 找到一个请求的所有日志，在 7.2 节会进行讲解。

ELK 非常强大，基本能满足所有日志查询需求，Kibana 的查询使用了 Lucene 语法，用 10 分钟左右就能基本上手。Kibana 还能创建各种仪表盘和聚合图表，读者可自行尝试。

图 5-4

本节的参考链接如下。

- http://blog.51cto.com/baidu/1676798
- http://elk-docker.readthedocs.io

5.4 OpenTracing + Jaeger

5.4.1 什么是 OpenTracing

OpenTracing 是一个分布式追踪规范，通过提供平台、厂商无关的 API，为分布式追踪提供了统一的概念和数据标准，使得开发人员能够方便地添加（或更换）追踪系统的实现。OpenTracing 定义了如下几个术语。

（1）Span：代表系统中的一个逻辑工作单元，具有操作名、操作开始时间及持续时长。Span 可能会有嵌套或排序，从而对因果关系建模。

- Tags：每个 Span 可以有多个键值对（key: value）形式的 Tags，Tags 是没有时

间戳的，支持简单地对 Span 进行注解和补充。

- Logs：每个 Span 可以进行多次 Log 操作，每一次 Log 操作都需要一个带时间戳的时间名称，以及可选的任意大小的存储结构。

（2）Trace：代表了系统中的一个数据 / 执行路径（一个或多个 Span），可以将其理解为 Span 的有向无环图。

OpenTracing 还有其他一些概念，这里就不过多解释了。我们来看一个传统的调用关系的例子，如图 5-5 所示。

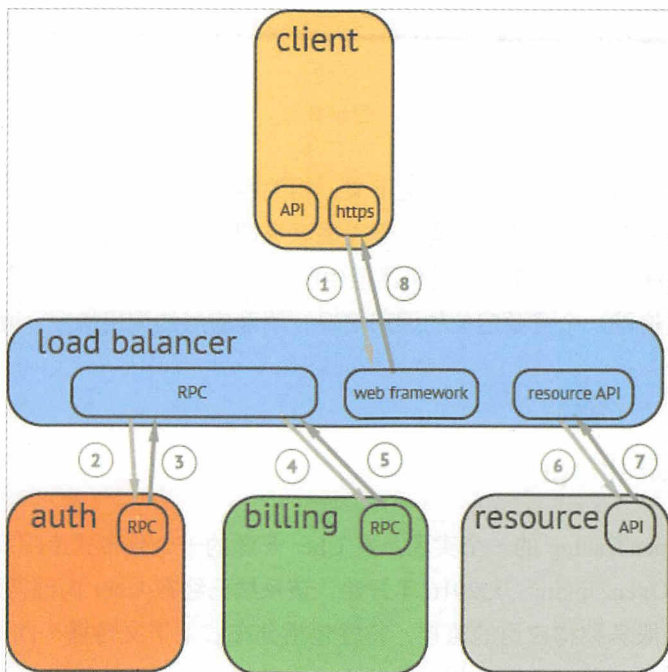


图 5-5

在一个分布式系统中追踪一个事务或者调用流的流程一般如图 5-5 所示。虽然这种图对于看清楚各组件的组合关系很有用，但是不能很好地显示组件的调用时间，以及是串行调用还是并行调用。如果展现更复杂的调用关系，则会更加复杂，甚至无法画出这样的图。另外，这种图也无法显示调用间的时间间隔及是否通过定时调用来启动调用。如下方式可以更有效地展现一个典型的 Trace 过程，如图 5-6 所示。

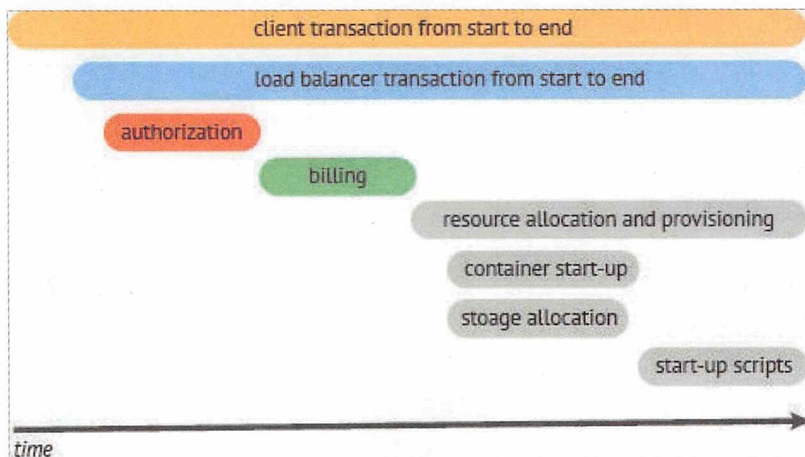


图 5-6

这种展现方式增加了执行时间的上下文、相关服务间的层次关系、进程或者任务的串行或并行调用关系。这样的视图有助于发现系统调用的关键路径。通过关注关键路径的执行过程，项目团队可以专注于优化路径中的关键位置，最大幅度地提升系统的性能。例如：可以通过追踪一个资源定位的调用情况，明确底层的调用情况，发现哪些操作有阻塞的情况。

5.4.2 什么是 Jaeger

Jaeger 是 OpenTracing 的一个实现，是 Uber 开源的一个分布式追踪系统，其灵感来源于 Dapper 和 OpenZipkin。从 2016 年开始，该系统已经在 Uber 内部得到了广泛应用，它可以用于对微服务架构应用的监控，特性包括分布式上下文传播（Distributed context propagation）、分布式事务监控、根原因分析、服务依赖分析及性能优化。该项目已经被云原生计算基金会（Cloud Native Computing Foundation，CNCF）接纳为第 12 个项目。

5.4.3 启动 Jaeger 及 Jaeger UI

我们使用 Docker 启动 Jaeger 及 Jaeger UI（即 Jaeger 可视化 Web 控制台），运行如下命令：

仅供非商业用途或交流学习使用


```
$ docker run -d -p5775:5775/udp \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 14268:14268 \
  jaegertracing/all-in-one:latest
```

用浏览器打开 localhost:16686，如图 5-7 所示。

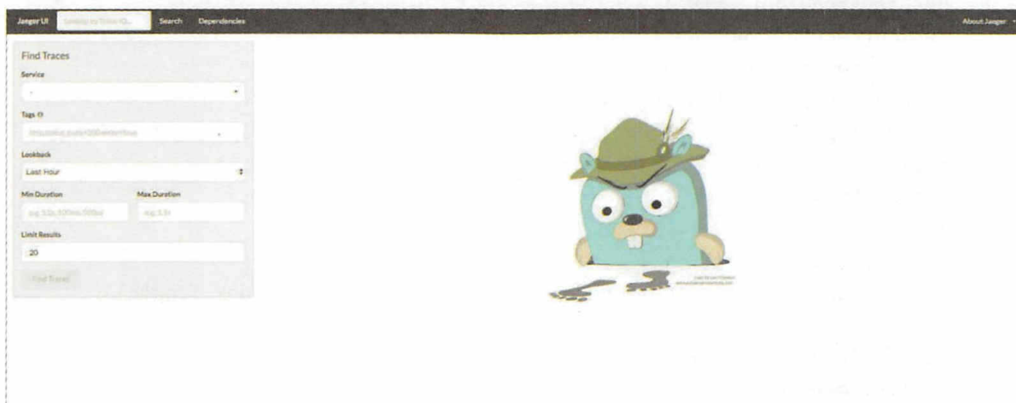


图 5-7

现在并没有任何数据，接下来我们看看如何使用 Jaeger 接收并查询日志。

5.4.4 使用 OpenTracing 及 Jaeger

OpenTracing 和 Jaeger 分别提供了 JavaScript、Node.js 的 SDK：opentracing/opentracing-javascript 和 jaegertracing/jaeger-client-node。

opentracing 的示例代码如下：

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

// NOTE: the default OpenTracing tracer does not record any tracing
information.
// Replace this line with the tracer implementation of your choice.
```

仅供非商业用途或交流学习使用


```
const tracer = new opentracing.Tracer()

const span = tracer.startSpan('http_request')
const opts = {
  host : 'example.com',
  method: 'GET',
  port : '80',
  path: '/',
}
http.request(opts, res => {
  res.setEncoding('utf8')
  res.on('error', err => {
    // assuming no retries, mark the span as failed
    span.setTag(opentracing.Tags.ERROR, true)
    span.log({'event': 'error', 'error.object': err, 'message': err.message,
'stack': err.stack})
    span.finish()
  })
  res.on('data', chunk => {
    span.log({'event': 'data_received', 'chunk_length': chunk.length})
  })
  res.on('end', () => {
    span.log({'event': 'request_end'})
    span.finish()
  })
}).end()
```

有以下两点需要解释。

- 需要将上面的 const tracer = new opentracing.Tracer() 替换成自己的 tracer 实现，即 Jaeger 的实现。
- 通过 tracer.startSpan 启动一个 Span，span.setTag 用来设置 Tags，span.log 用来设置 Logs，span.finish 用来结束一个 Span。

这有点类似于我们的手动埋点，只不过变成了一个规范而已。但 OpenTracing 的功能不止于此，上面只是一个 Span 的用法，在 Span 之间还可以关联调用关系，最后得到一个 DAG（有向无环图）。

举个例子，假如我们正在做微服务，多个服务之间有调用关系（不管是 HTTP 还是 RPC 等），每次调用服务在内部可能产生多个 Span，最终会在 Jaeger 控制台页面看到一个完整的 Trace 和 DAG 图（指微服务之间的调用关系）。

仅供非商业用途或交流学习使用

jaeger-client-node 的使用如下：

```
const tracer = new jaeger.Tracer(
  serviceName,
  new jaeger.RemoteReporter(new UDPSender()),
  new jaeger.RateLimitingSampler(1)
)
```

创建一个 tracer，可以接收如下 3 个参数。

- serviceName：服务名。
- Reporter：上报器，即向哪里发送日志，例如上述代码是通过 UDP 发送日志的，默认的地址是 localhost:6832。
- Sampler：采样器，即日志如何采样，例如上述代码是限制 1 秒采样一次的。

这里不再详细介绍其他选项，读者可自行查阅 jaeger-client-node 的文档。

5.4.5 koa-await-breakpoint-jaeger

我们通过上面的例子可以知道，在使用 Jaeger 时需要手动埋点。前面介绍了 koa-await-breakpoint 日志自动打点，可自定义 store，koa-await-breakpoint-jaeger 是为 koa-await-breakpoint 写的 store 的 adaptor，在实现上有一些小技巧，有兴趣的读者可以去读其源码。

还是以 koa-await-breakpoint 的 example 为例，只添加了两行代码来使用 Jaeger，代码如下。

app.js：

```
const JaegerStore = require('koa-await-breakpoint-jaeger')
const koaAwaitBreakpoint = require('koa-await-breakpoint')({
  name: 'api',
  files: ['./routes/*.js'],
  store: new JaegerStore()
})

const Paloma = require('paloma')
const app = new Paloma()
```

仅供非商业用途或交流学习使用


```
app.use(koaAwaitBreakpoint)
app.route({ method: 'POST', path: '/users', controller: require('./routes/
user').createUser })

app.listen(3000)
```

运行：

```
$ curl -XPOST localhost:3000/users
```

在刷新 localhost:16686 后便可以看到日志了，如图 5-8 所示。

图 5-8

选择 Sercice → api，并选择 Operation → POST /users，单击 Find Traces 来查看所有结果，在右侧展示了一条日志，单击该日志，效果如图 5-9 所示。

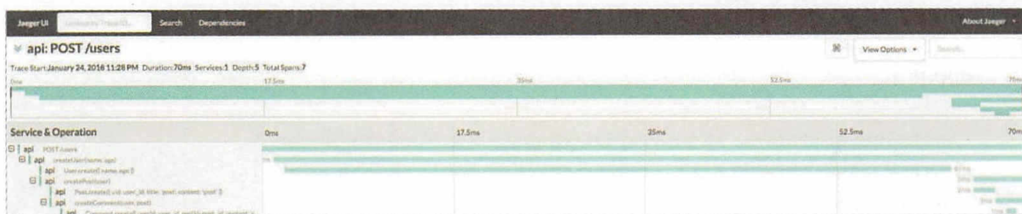


图 5-9

小提示

可以根据 tags 过滤结果。

仅供非商业用途或交流学习使用

注 意

Jaeger 是分布式追踪系统，通常用来追踪多个服务之间的调用关系，在这里用来追踪一个服务的多个函数之间的调用关系。

修改 routes/user.js 的 createComment 函数，抛出一个 new Error('test')，重新运行，如图 5-10 所示。

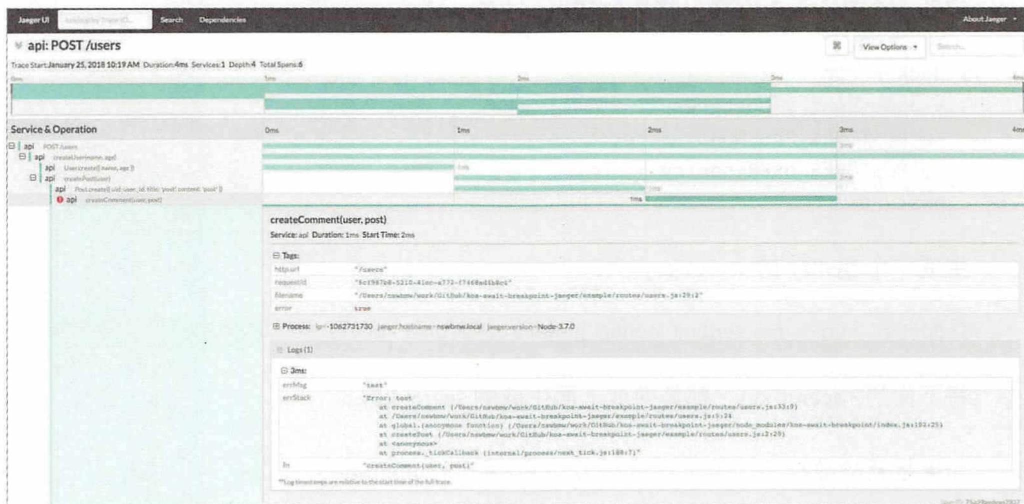


图 5-10

可以看出，Jaeger 完美地展现了在一个请求到来时函数之间的调用关系、层级关系及耗时，甚至函数体和错误栈！之后，我们就可以用 requestId 去 ELK 中查询日志了。

本节的参考链接如下。

- https://wu-sheng.gitbooks.io/opentracing-io/content/
- https://segmentfault.com/a/1190000008895129
- http://www.infoq.com/cn/news/2017/11/Uber-open-spurce-Jaeger

仅供非商业用途或交流学习使用

5.5 使用 Sentry

Sentry 是一个开源的实时错误日志收集平台。下面使用 Docker 安装并启动 Sentry。

启动一个 Redis 容器，将其命名为 sentry-redis：

```
$ docker run -d --name sentry-redis redis
```

启动一个 Postgres 容器，将其命名为 sentry-postgres：

```
$ docker run -d \
  --name sentry-postgres \
  -e POSTGRES_PASSWORD=secret \
  -e POSTGRES_USER=sentry \
  postgres
```

生成一个 Sentry 的 secret key：

```
$ docker run --rm sentry config generate-secret-key
```

将下面的 <secret-key> 都替换成上面生成的 secret key。

如果是新的数据库（第 1 次运行），则需要运行 upgrade：

```
$ docker run -it --rm \
  -e SENTRY_SECRET_KEY='<secret-key>' \
  --link sentry-postgres:postgres \
  --link sentry-redis:redis \
  sentry upgrade
```

按步骤填写自己的信息，如图 5-11 所示。

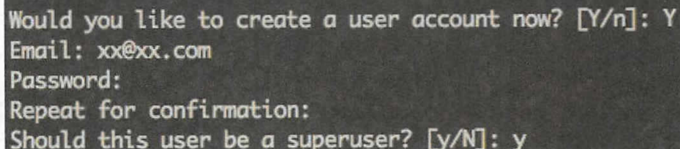


图 5-11

仅供非商业用途或交流学习使用

最终创建了一个超级管理员和一个默认的名为 Sentry 的组织（organization）。

启动 Sentry，并对外暴露 9000 端口：

```
$ docker run -d \
  --name my-sentry \
  -e SENTRY_SECRET_KEY='<secret-key>' \
  --link sentry-redis:redis \
  --link sentry-postgres:postgres \
  -p 9000:9000 \
  sentry
```

启动 Celery cron 和 Celery workers：

```
$ docker run -d \
  --name sentry-cron \
  -e SENTRY_SECRET_KEY='<secret-key>' \
  --link sentry-postgres:postgres \
  --link sentry-redis:redis \
  sentry run cron

$ docker run -d \
  --name sentry-worker-1 \
  -e SENTRY_SECRET_KEY='<secret-key>' \
  --link sentry-postgres:postgres \
  --link sentry-redis:redis \
  sentry run worker
```

> 小提示
>
> Celery 是用 Python 写的一个分布式任务调度模块。

至此便完成了 Sentry 的安装和启动了。

用浏览器打开 localhost:9000 就能看到 Sentry 的登录页面了，如图 5-12 所示。

在首次登录时需要填写一些必要的信息，如图 5-13 所示。

仅供非商业用途或交流学习使用

图 5-12

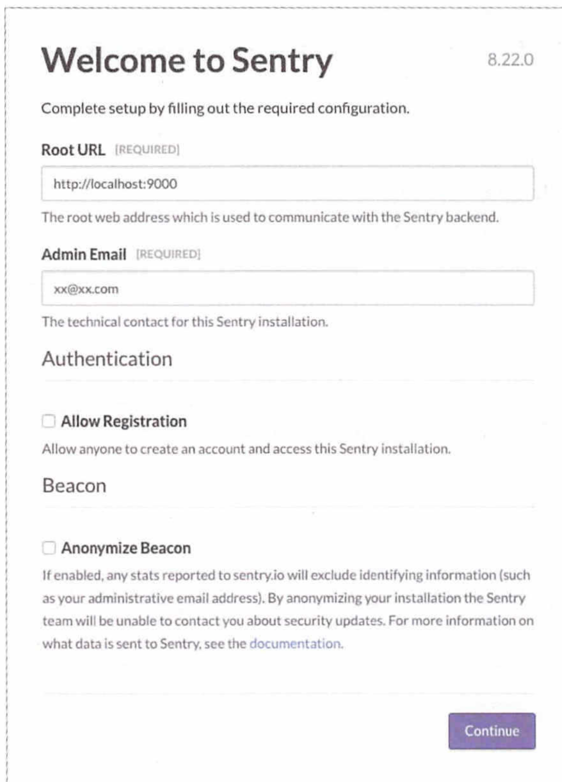


图 5-13

仅供非商业用途或交流学习使用

单击 Continue，进入 Sentry 仪表盘（Dashboard）。单击右上角的 New Project 按钮创建一个项目，选择 Node.js 并填写项目名称为 API，然后单击 Create Project 按钮创建项目。如图 5-14 所示。

图 5-14

在创建成功后进入 Node.js 使用示例的页面，选择使用 Koa 测试，在右侧选择 Koa，如图 5-15 所示是 koa@1 的示例代码。

图 5-15

我们以 Paloma（基于 koa@2）为例，编写测试代码：

```
const Raven = require('raven')
const Paloma = require('paloma')
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
const app = new Paloma()

Raven.config(DSN).install()

app.on('error', (err) => {
  Raven.captureException(err, (err, eventId) => {
    console.log('Reported error ' + eventId)
  })
})

app.use((ctx) => {
  throw new Error('test')
})

app.listen(3000)
```

raven 是 Node.js 版的 Sentry SDK，用来收集和发送错误日志。

小提示

将 DSN 替换为图 5-15 中的 http://xxx@localhost:9000/2，DSN 既告诉客户端 Sentry 服务器的地址，也被用作身份认证的 token。

运行以上测试代码，访问 localhost:3000，错误信息会被发送到 Sentry，Sentry 界面如图 5-16 所示。

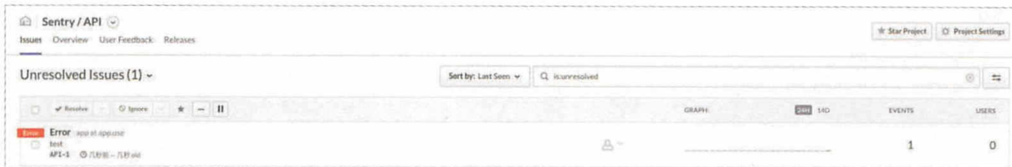


图 5-16

点进去便可以看到详细的信息，如图 5-17 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

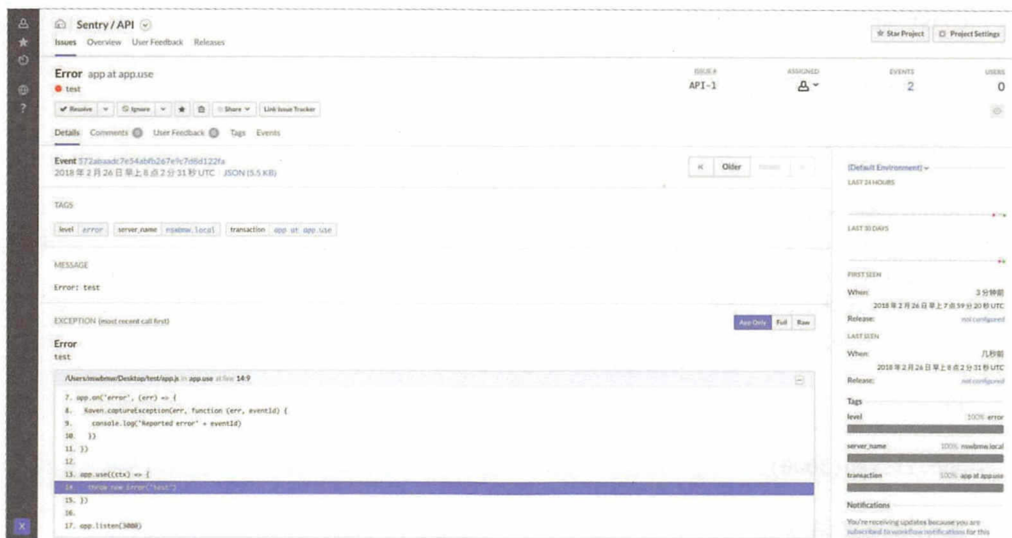


图 5-17

Sentry 还有许多功能，比如：错误归类、展示错误的频率柱状图、将错误指派给组织中的某个人、给错误添加标签、查看这类错误事件的历史、标记错误为已解决、在错误下发表评论、警报等，读者可自行尝试了解。

笔者将 raven 封装成 Koa 的一个中间件，将其命名为 koa-raven，使用如下：

```
const raven = require('koa-raven')
const Paloma = require('paloma')
const app = new Paloma()

app.use(raven(DSN))

app.use((ctx) => {
  throw new Error('test')
})

app.listen(3000)
```

或者使用 ctx.raven：

```
const raven = require('koa-raven')
const Paloma = require('paloma')
```

书籍是人类进步的阶梯

```
const app = new Paloma()

app.use(raven(DSN))

app.use((ctx) => {
  try {
    throw new Error('test')
  } catch (e) {
    ctx.raven.captureException(e, { extra: { name: 'tom' } })
    ctx.status = 500
    ctx.body = e.stack
  }
})

app.listen(3000)
```

本节的参考链接如下。

- https://sentry.io/

书籍是人类进步的阶梯

书籍是人类进步的阶梯

第6章

APM

APM（Application Performance Management），即应用程序性能管理工具，通过探针的方式无侵入式地植入埋点，监测和诊断应用程序的性能问题。本章将会讲解如何使用两种不同类型的 APM。

6.1 使用 NewRelic

NewRelic 是一个老牌的应用性能监测工具，提供了 14 天的免费试用期，本节将讲解如何使用 NewRelic 监控 Node.js 程序的性能。

测试代码如下。

app.js：

```
require('newrelic')

const crypto = require('crypto')
const express = require('express')
const app = express()
const createUser = require('./routes/users').createUser

app.get('/', (req, res) => {
  const salt = crypto.randomBytes(128).toString('base64')
  const hash = crypto.pbkdf2Sync(String(Math.random()), salt, 10000, 512,
'sha512').toString('hex')
  res.json({ salt, hash })
})

app.get('/error', (req, res, next) => {
  next(new Error('error!!!'))
})

app.post('/users/:user', async (req, res) => {
  const user = await createUser(req.params.user, 18)
  res.json(user)
})

app.listen(3000)
```

routes/users.js：

```
const Mongolass = require('mongolass')
const mongolass = new Mongolass('mongodb://localhost:27017/test')
const User = mongolass.model('User')

exports.createUser = async function (ctx) {
```

书籍是人类进步的阶梯

```
  const name = ctx.query.name || 'default'
  const age = +ctx.query.age || 18
  const user = await createUser(name, age)
  ctx.status = user
}

async function createUser (name, age) {
  const user = (await User.create({
    name,
    age
  })).ops[0]
  return user
}
```

首先，注册一个 NewRelic 账号。创建一个应用，如图 6-1 所示。

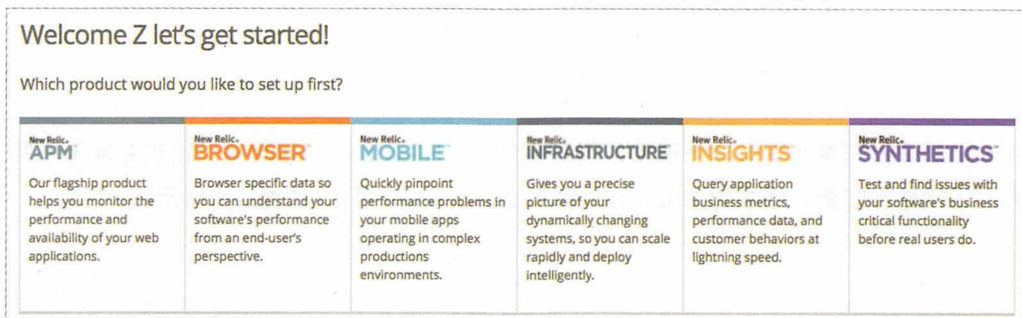


图 6-1

选择 APM，进入下一步，选择 Node.js 应用，并拿到 license key，如图 6-2 所示。

在 Node.js 中使用 NewRelic 的步骤如下：

```
$ npm i newrelic --save # 安装 NewRelic 的 Node.js SDK
$ cp node_modules/newrelic/newrelic.js . # 将默认配置文件复制到项目根目录下
```

修改 newrelic.js，在 app_name 处填写我们的应用名（例如 api），在 license_key 处填写刚才生成的 license key。

书籍是人类进步的阶梯

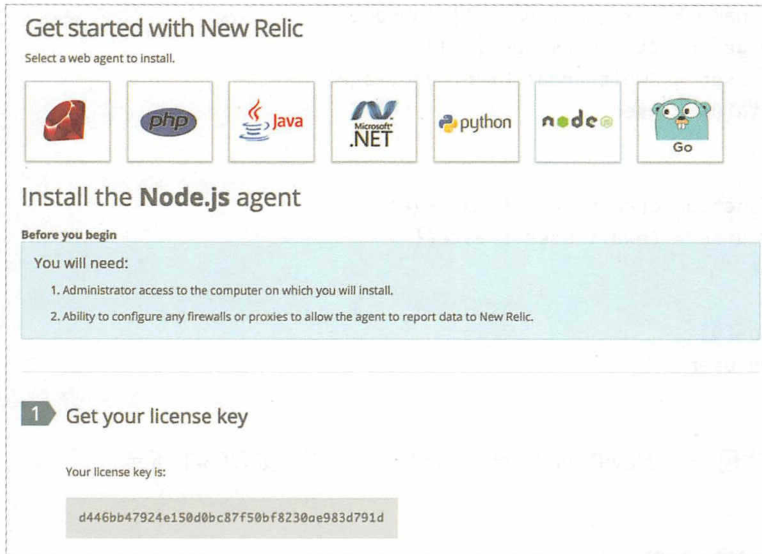


图 6-2

启动测试程序并发起几个请求，稍等几分钟，NewRelic 的后台将会收到并展示一些数据（例如吞吐量、请求的 Urls、错误率和 Apdex score 等），如图 6-3 所示。

图 6-3

书籍是人类进步的阶梯

试用版的功能有限，升级到付费版可解锁更多的功能，例如数据库分析、错误分析，甚至 Node.js VM 监控（CPU、内存、GC、Event Loop），等等。

类似的其他 APM 还有 AppDynamics、OneAPM、DataDog、atatus 和 opbeat，用法大同小异，这里就不一一介绍了。

本节的参考链接如下。

- https://newrelic.com/

6.2 Elastic APM

6.2.1 什么是 Elastic APM

Elastic APM 是 Elastic 公司开源的一款 APM 工具，目前还处于 Beta 阶段，它有以下几个优势。

- 开源。我们可以免费使用，像使用 ELK 一样。
- 功能完善。API 比较完善，有 Agent、Transaction 和 Trace，默认创建响应时间和每分钟请求数这两种图表，且可以使用 Kibana 的 Filter 过滤生成我们所关心的数据的图表。
- 监控与日志统一。Elastic APM 依赖 Elasticsearch 和 Kibana，所以可以结合 ELK 使用，可以在 Kibana 中查看监控及直接查询日志。

Elastic APM 的架构如图 6-4 所示 。

书籍是人类进步的阶梯

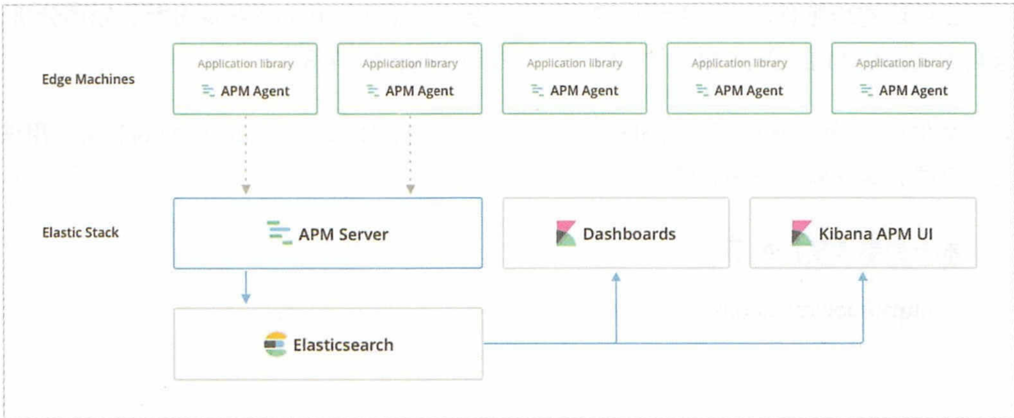


图 6-4

可以看出，APM Agent（即在应用端引入的探针）将收集的日志发送到 APM Server（Go 写的 HTTP 服务），APM Server 将数据存储到 Elasticsearch 中，然后通过 Kibana 展示。

Kibana 的界面如图 6-5 所示。

图 6-5

书籍是人类进步的阶梯

6.2.2 启动 ELK

我们使用 Docker 安装并启动 ELK，运行如下命令：

```
$ docker run -p 5601:5601 \
    -p 9200:9200 \
    -p 5044:5044 \
    -it --name elk sebp/elk
```

6.2.3 启动 APM Server

下载 APM Server 并解压，然后运行以下命令：

```
$ ./apm-server setup # 导入 APM 仪表盘到 Kibana
$ ./apm-server -e # 启动 APM Server，默认监听 8200 端口
```

用浏览器打开 localhost:5601，进入 Dashboard 页，如图 6-6 所示。

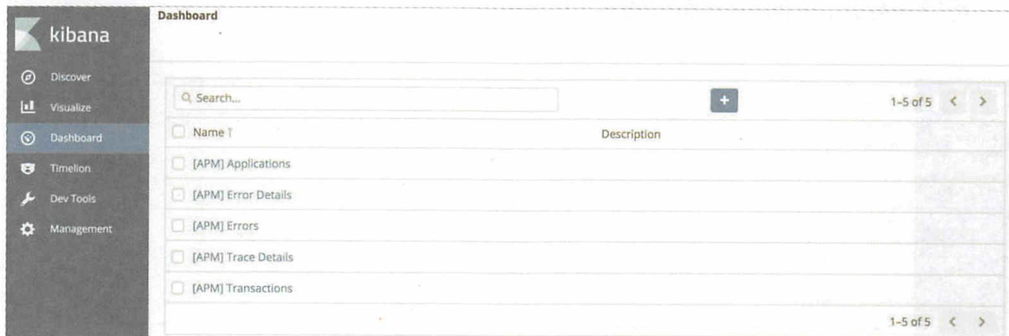


图 6-6

6.2.4 使用 Elastic APM

测试代码如下：

```
const apm = require('elastic-apm-node').start({
  appName: 'test'
})
```

```
const Paloma = require('paloma')
const app = new Paloma()

app.route({ method: 'GET', path: '/', controller (ctx) {
  apm.setTransactionName(`${ctx.method} ${ctx._matchedRoute}`)
  ctx.status = 200
}})

app.route({ method: 'GET', path: '/:name', controller (ctx) {
  apm.setTransactionName(`${ctx.method} ${ctx._matchedRoute}`)
  ctx.status = 200
}})

app.listen(3000)
```

运行该程序，发起两个请求：

```
$ curl localhost:3000/
$ curl localhost:3000/nswbmw
```

等待一会儿，Kibana 界面如图 6-7 所示。

图 6-7

在 Elastic APM 中有如下两个术语。

- transaction：指一组 traces 的集合，例如一个 HTTP 请求。
- trace：指一个事件及持续时间，例如一个 SQL 查询。

6.2.5 错误日志

现在，我们来测试一下 Elastic APM 的错误收集功能。将测试代码修改为：

```
const apm = require('elastic-apm-node').start({
  appName: 'test'
})

const Paloma = require('paloma')
const app = new Paloma()

app.use(async (ctx, next) => {
  try {
    await next()
  } catch (e) {
    apm.captureError(e)
    ctx.status = 500
    ctx.message = e.message
  }
})

app.route({ method: 'GET', path: '/', controller: function indexRouter (ctx) {
  apm.setTransactionName(`${ctx.method} ${ctx._matchedRoute}`)
  throw new Error('error!!!')
}})

app.listen(3000)
```

重启测试程序并发起一次请求。回到 Kibana，单击 Dashboard → [APM] Errors，可以看到错误日志记录（自动聚合）和图表，如图 6-8 所示。

图 6-8

单击 View Error Details，进入错误详情页，如图 6-9 所示。

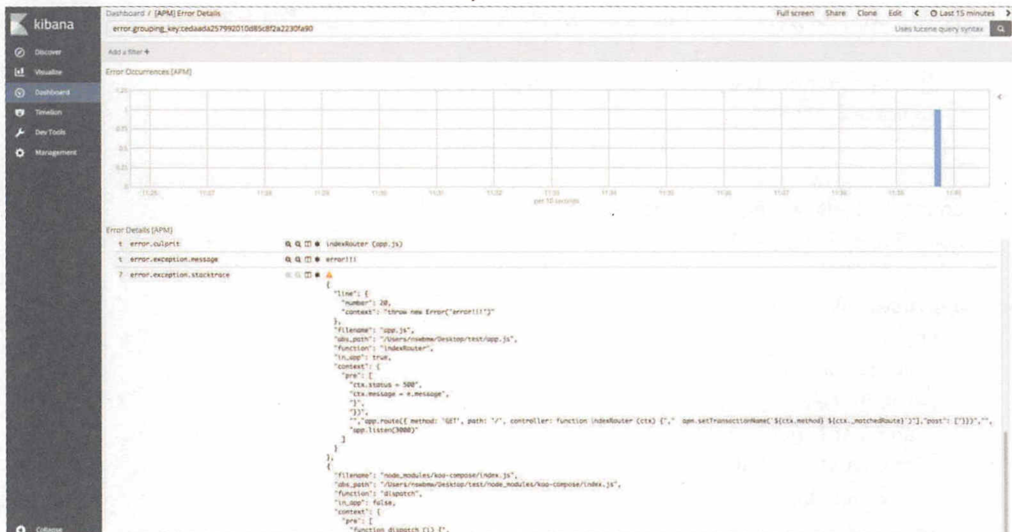


图 6-9

可以看出，在错误日志中展示了错误的代码及行数、上下几行代码、父级函数名和所在文件等信息。

本节的参考链接如下。

- https://www.elastic.co/guide/en/apm/agent/nodejs/0.x/index.html
- https://www.elastic.co/guide/en/apm/agent/nodejs/0.x/custom-stack.html

第7章 监控

监控可以将应用状态可视化，报警通常是我们首先发现应用程序问题的途径，本章将会讲解如何使用 Telegraf + InfluxDB + Grafana 这个组合搭建完整的 Node.js 应用监控和报警。

7.1 Telegraf + InfluxDB + Grafana（上）

本节将会讲解如何使用 Telegraf（StatsD）+ InfluxDB + Grafana 搭建一套完整的监控系统。

7.1.1 Telegraf（StatsD）+ InfluxDB + Grafana 简介

Telegraf 是一个使用 Go 语言开发的代理程序，可收集系统和服务或者其他来源（inputs）的数据，并将其写入 InfluxDB（outputs）数据库，支持多种 inputs 和 outputs 插件。StatsD 是一个使用 Node.js 开发的网络守护进程，通过 UDP 或者 TCP 方式收集各种统计信息，包括计数器和定时器等。

InfluxDB 是一个使用 Go 语言开发的开源的分布式时序、事件和指标数据库，无须外部依赖，其设计目标是实现分布式和水平伸缩扩展。

Grafana 是一个使用 Angular + Go 语言开发的开源的、功能齐全的、漂亮的仪表盘和图表的编辑器，可用来做日志的分析与展示曲线图（如 api 的请求日志），支持多种 backend，例如 Elasticsearch 、InfluxDB、OpenTSDB 等。

其工作流程为：Telegraf 将 StatsD（inputs）和 InfluxDB（outputs）结合起来，将发往 StatsD 的数据，最终通过 Telegraf 写入 InfluxDB，然后 Grafana 读取 InfluxDB 的数据并将其以图表的形式展示。

7.1.2 启动 docker-statsd-influxdb-grafana

我们使用 Docker 一键启动 Telegraf（StatsD）+ InfluxDB + Grafana，缩短搭建环境的时间：

```
$ docker run -d \
  --name docker-statsd-influxdb-grafana \
  -p 3003:3003 \
  -p 3004:8083 \
  -p 8086:8086 \
  -p 22022:22 \
```

```
  -p 8125:8125/udp \
  samuelebistoletti/docker-statsd-influxdb-grafana:latest
```

端口映射关系如下：

```
Host         Container        Service
-----------------------------------
3003         3003             grafana
3004         8083             influxdb-admin
8086         8086             influxdb
8125         8125             statsd
22022        22               sshd
```

7.1.3 熟悉 InfluxDB

在容器启动后，用浏览器访问 localhost:3004（以下称之为 influxdb-admin），如图 7-1 所示。

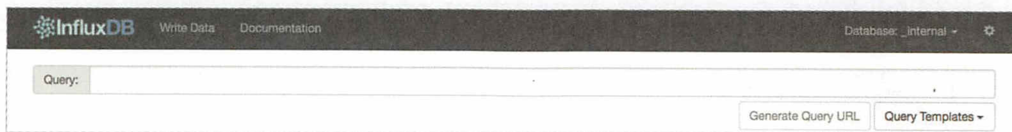


图 7-1

InfluxDB 的基本概念如下。

- database：数据库。
- measurement：数据库中的表。
- point：表里面的一行数据，由时间戳（time）、数据（field）和标签（tag）组成。其中，time 是每条数据记录的时间戳，是数据库中的主索引（会自动生成）；field 是各种记录的值（没有索引的属性）；tag 是各种有索引的属性。

InfluxDB 采用了类 SQL 的查询语法，如下所述。

- show databases：列出所有数据库。
- show measurements：列出当前数据库的所有表。
- select * from xxx：列出 xxx 表的所有数据。

我们在 Query 中输入：

```
show databases
```

查询结果如图 7-2 所示。

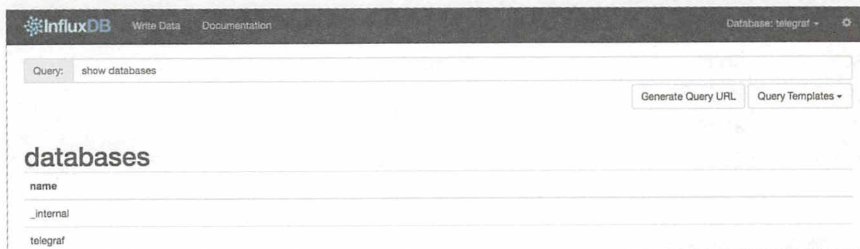


图 7-2

_interal 是 InfluxDB 内部使用的数据库，telegraf 是当前的 Docker 容器在启动后默认创建的测试数据库。

7.1.4 配置 Grafana

用浏览器打开 localhost:3003，如图 7-3 所示。

图 7-3

输入用户名 root 和密码 root 登录，进入初始化配置页，单击“Add data source”，填写的内容如图 7-4 所示。

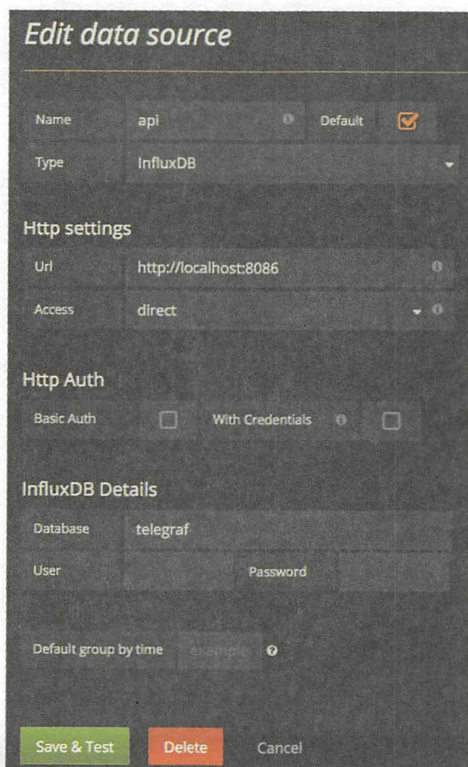


图 7-4

单击“Save & Test”保存配置。目前配置 Grafana 默认的 datasource 是名为 api 的 InfluxDB。接下来创建测试代码，生产测试数据。

7.1.5 node-statsd

node-statsd 是一个 statsd 的 Node.js client。创建以下测试代码：

```
const StatsD = require('node-statsd')
const statsdClient = new StatsD({
  host: 'localhost',
```

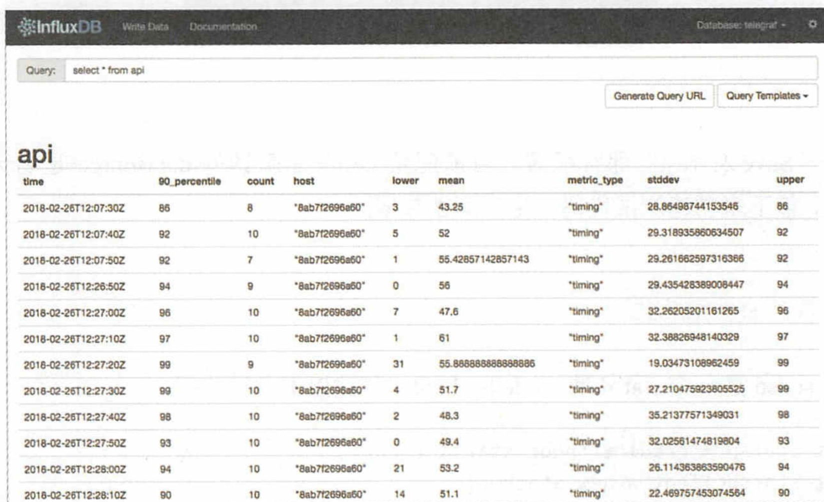
```
    port: 8125
  })

  setInterval(() => {
    const responseTime = Math.floor(Math.random() * 100)
    statsdClient.timing('api', responseTime, function (error, bytes) {
      if (error) {
        console.error(error)
      } else {
        console.log(`Successfully sent ${bytes} bytes, responseTime
  ${responseTime}ms`)
      }
    })
  }, 1000)
```

运行以上代码，每秒钟会产生一个 0 ~ 99 的随机值（模拟响应时间，单位为 ms）并将其发送到 StatsD，StatsD 会通过 Telegraf 将这些数据写入 InfluxDB 的 telegraf 数据库。

回到 influxdb-admin，单击右上角的下拉菜单，切换到 telegraf 数据库，然后输入 show measurements 查看已经存在 api 表，输入：

```
select * from api
```

查询结果如图 7-5 所示。



time	90_percentile	count	host	lower	mean	metric_type	stddev	upper
2018-02-26T12:07:30Z	86	8	"8ab7f2696a60"	3	43.25	"timing"	28.86498744153546	86
2018-02-26T12:07:40Z	92	10	"8ab7f2696a60"	5	52	"timing"	29.318935860634507	92
2018-02-26T12:07:50Z	92	7	"8ab7f2696a60"	1	55.42857142857143	"timing"	29.261662597316366	92
2018-02-26T12:26:50Z	94	9	"8ab7f2696a60"	0	56	"timing"	29.435428389008447	94
2018-02-26T12:27:00Z	96	10	"8ab7f2696a60"	7	47.6	"timing"	32.26205201161265	96
2018-02-26T12:27:10Z	97	10	"8ab7f2696a60"	1	61	"timing"	32.38826948140329	97
2018-02-26T12:27:20Z	99	9	"8ab7f2696a60"	31	55.888888888888886	"timing"	19.03473108962459	99
2018-02-26T12:27:30Z	99	10	"8ab7f2696a60"	4	51.7	"timing"	27.210475923805525	99
2018-02-26T12:27:40Z	98	10	"8ab7f2696a60"	2	48.3	"timing"	35.21377571349031	98
2018-02-26T12:27:50Z	93	10	"8ab7f2696a60"	0	49.4	"timing"	32.02561474819804	93
2018-02-26T12:28:00Z	94	10	"8ab7f2696a60"	21	53.2	"timing"	26.114363863590476	94
2018-02-26T12:28:10Z	90	10	"8ab7f2696a60"	14	51.1	"timing"	22.469757453074564	90

图 7-5

可以看出 api 表有以下几个字段。

- time：InfluxDB 默认添加的时间戳。
- 90_percentile：所有记录中从小到大排列的 90% 那个点的值。
- count：一次收集的日志数量，可以看出每条记录（point）的 count 值接近或等于 10，而我们的测试代码是 1s 发送一条数据，也就说明 Telegraf 的默认设置是 10s 收集一次数据，默认的配置也的确是这样的，可参见 https://github.com/samuelebistoletti/docker-statsd-influxdb-grafana/blob/master/telegraf/telegraf.conf。
- host：机器的地址。
- lower：最小的那条记录的值。
- mean：所有记录的平均值。
- metric_type：metric 类型。
- stddev：所有记录的标准差。
- upper：最大的那条记录的值。

7.1.6 创建 Grafana 图表

回到 Grafana，单击左上角 Grafana 图标的下拉菜单，单击 Dashboards 回到仪表盘页继续完成配置，单击“New dashboard”，然后单击创建 Graph 类型的图表，就创建了一个空的图表，如图 7-6 所示。

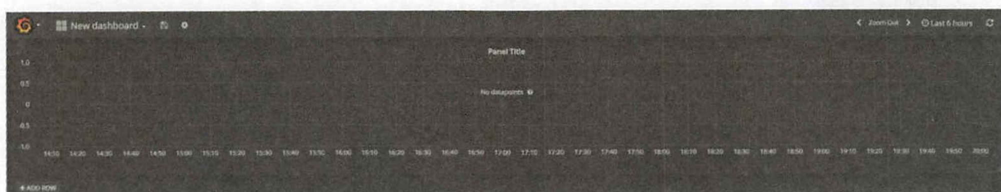


图 7-6

单击当前的图表，选择 Edit，修改如下几个地方。

- 在 Metrics 配置中选择 FROM → api 表，SELECT → field(mean) 字段。
- 在 Display 配置的“Null value”中选择 connected，将每个点连成折线。

效果如图 7-7 所示。

仅供非商业用途或交流学习使用

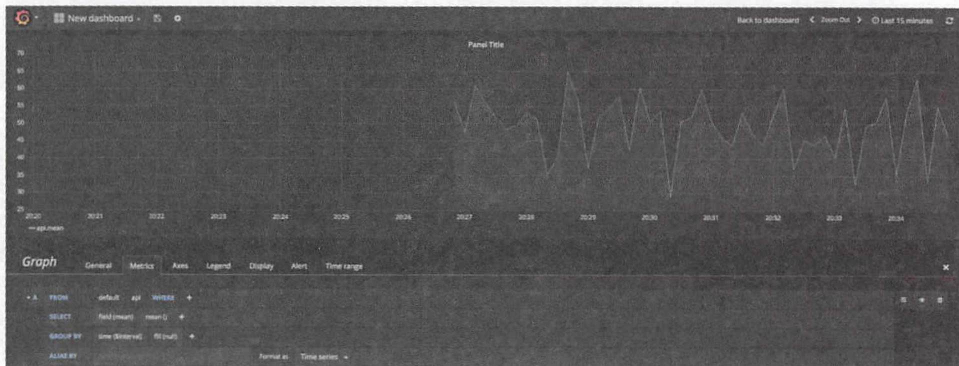


图 7-7

7.1.7　模拟真实环境

下面我们用代码模拟一下在真实的环境下如何使用刚才搭建的监控系统。首先，分别创建 middlewares/statsd.js、server.js 和 client.js 文件。

middlewares/statsd.js：

```
const StatsD = require('node-statsd')
const statsdClient = new StatsD({
  host: 'localhost',
  port: 8125
})

module.exports = function (routerName) {
  return async function statsdMiddleware (ctx, next) {
    const start = Date.now()

    try {
      await next()
      const spent = Date.now() - start
      statsdClient.timing(`api_${routerName}`, spent)
    } catch (e) {
      statsdClient.increment(`api_${routerName}_${e.status || (ctx.status !==
404 ? ctx.status : 500)}`)
      throw e
    }
  }
```

仅供非商业用途或交流学习使用


```
}
```

server.js：

```
const Bluebird = require('bluebird')
const Paloma = require('paloma')
const app = new Paloma()
const statsd = require('./middlewares/statsd')

app.route({ method: 'GET', path: '/', controller: [
  statsd('getHome'),
  async (ctx) => {
    // 模拟十分之一出错概率
    if (Math.random() < 0.1) {
      console.error('error')
      ctx.throw(400)
    }
    // 模拟 1 ~ 100 毫秒响应时间
    const responseTime = Math.floor(Math.random() * 100 + 1)
    await Bluebird.delay(responseTime)
    console.log(`Spent ${responseTime}ms`)
    ctx.status = 200
  }
]})

app.listen(3000)
```

client.js：

```
const axios = require('axios')

setInterval(() => {
  // 模拟 1-10 的 tps
  const tps = Math.floor(Math.random() * 10 + 1)
  for (let i = 0; i < tps; i++) {
    axios.get('http://localhost:3000')
  }
}, 1000)
```

打开两个终端，分别运行：

```
$ node server.js
$ node client.js
```

仅供非商业用途或交流学习使用

回到 influxdb-admin，输入：

```
show measurements
```

可以看到已经有 api_getHome 和 api_getHome_400 表了。回到 Grafana，在一行（row）里创建两个图表，分别如下。

- 请求量：包含了正常请求（200）和错误请求（4xx、5xx 等等）请求量的折线图。
- 响应时间：正常请求的最低（lower）、平均（mean）、最高（upper）响应时间的折线图。

效果如图 7-8 所示。

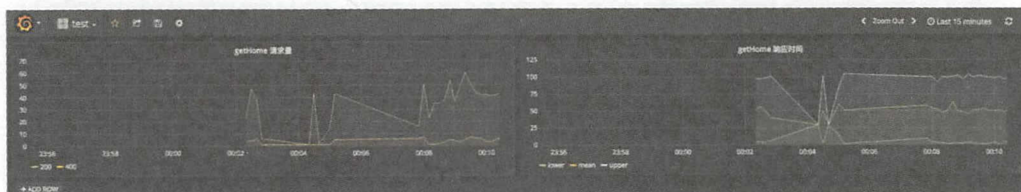


图 7-8

以“getHome 响应时间”的图表为例，Metrics 的配置截图如图 7-9 所示。

▾ A	FROM	default	api_getHome	WHERE	+		
	SELECT	field (lower)	mean ()	+			
	GROUP BY	time ($interval)	fill (null)	+			
	ALIAS BY	lower				Format as	Time series ▾
▾ B	FROM	default	api_getHome	WHERE	+		
	SELECT	field (mean)	mean ()	+			
	GROUP BY	time ($interval)	fill (null)	+			
	ALIAS BY	mean				Format as	Time series ▾
▾ C	FROM	default	api_getHome	WHERE	+		
	SELECT	field (upper)	mean ()	+			
	GROUP BY	time ($interval)	fill (null)	+			
	ALIAS BY	upper				Format as	Time series ▾

图 7-9

仅供非商业用途或交流学习使用

本节的参考链接如下。

- https://www.cnblogs.com/shhnwangjian/p/6897216.html

7.2 Telegraf + InfluxDB + Grafana（下）

7.1 节主要讲解了 Telegraf（StatsD）+ InfluxDB + Grafana 的搭建和基本用法，并创建了请求量和响应时间这两种图表。本节讲解如下几个高级用法。

- 如何将 Grafana（监控）与 ELK（日志）结合起来。
- Grafana 监控报警。
- 用脚本一键生成图表。

7.2.1 Grafana + ELK

在观察 Grafana 监控时，我们发现某个 api 接口的响应时间突然有一个尖刺，这时想查一查到底是什么原因导致的。我们在前面了解了 koa-await-breakpoint + ELK 的用法，那这时是否可以结合 Grafana 使用呢？答案是可以。

因为涉及的代码量大，所以笔者写了一个 demo 托管到了 GitHub 上，有两个 repo，分别如下。

（1）grafana-to-elk：包含 web server 和模拟请求的 client，分别将统计信息发送到 StatsD 和将日志发送到 ELK。

（2）grafana-to-elk-extension：为 Chrome 扩展，作用如下。

- 格式化从 Grafana 跳转到 ELK 的时间范围。
- 添加 requestId 的链接。
- 高亮显示重要的字段。

首先将其 clone 到本地，然后进行测试：

```
$ git clone https://github.com/nswbmw/grafana-to-elk.git
$ git clone https://github.com/nswbmw/grafana-to-elk-extension.git
```

仅供非商业用途或交流学习使用

按照 7.2 节启动 Telegraf（StatsD）+ InfluxDB + Grafana。

按照 6.3 节启动 ELK。

到 grafana-to-elk 目录下运行：

```
$ npm i
$ node server
```

打开另外一个终端并运行：

```
$ node client
```

此时，ELK 应该有日志了。

加载 Chrome 扩展。打开 Chrome 扩展程序页，加载已解压的扩展程序，加载 grafana-to-elk-extension（在非测试环境下需要修改 manifest.json 的 matches 字段）。

回到 Grafana 的"getHome 响应时间"图表，进入编辑页的 General tab，填写的内容如图 7-10 所示。

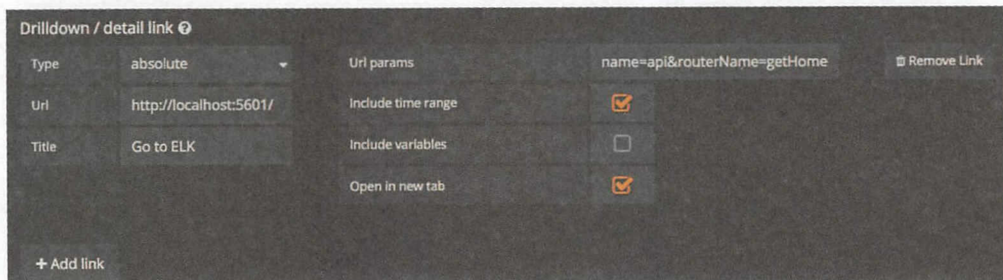


图 7-10

在保存后，在图表的左上角会出现一个类似分享的按钮，单击 Go to ELK，跳转到 ELK。

仅供非商业用途或交流学习使用

ELK 界面如图 7-11 所示。

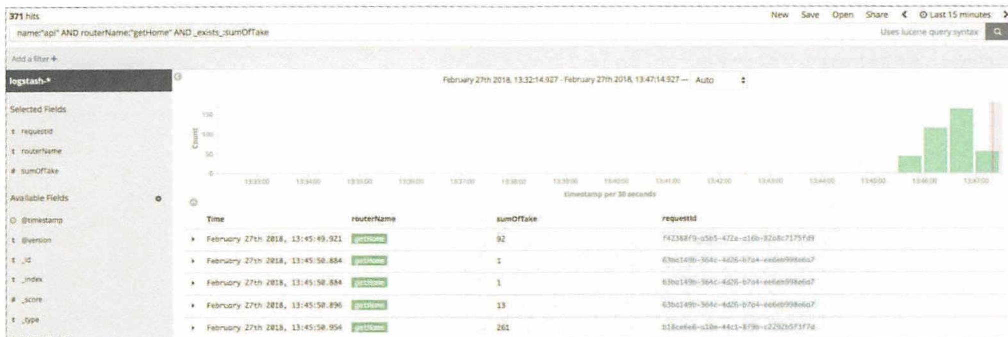


图 7-11

grafana-to-elk-extension 插件会自动处理并跳转到对应的 Grafana 中的时间段并且查询出我们关心的结果。单击第 1 个 requestId，将会跳转并显示该请求的所有日志，如图 7-12 所示。

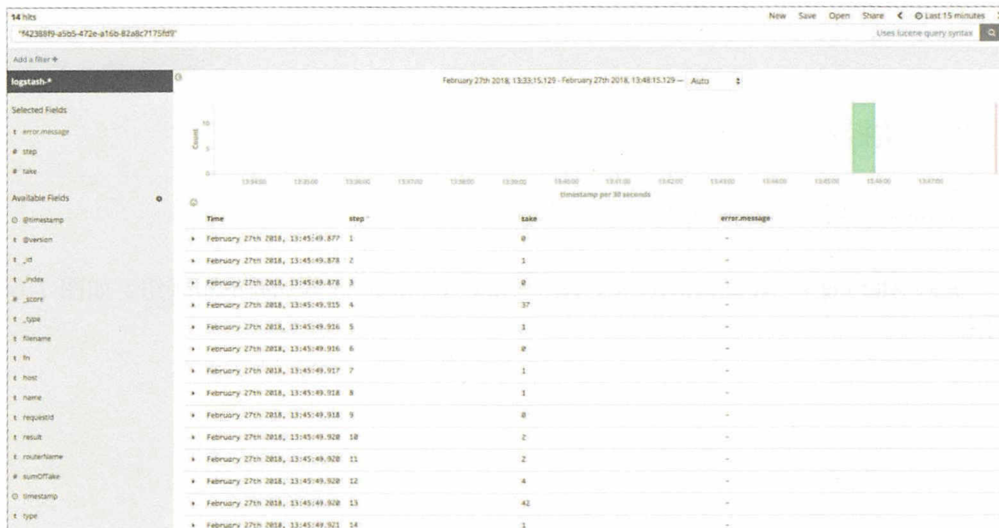


图 7-12

仅供非商业用途或交流学习使用

错误请求的日志如图 7-13 所示。

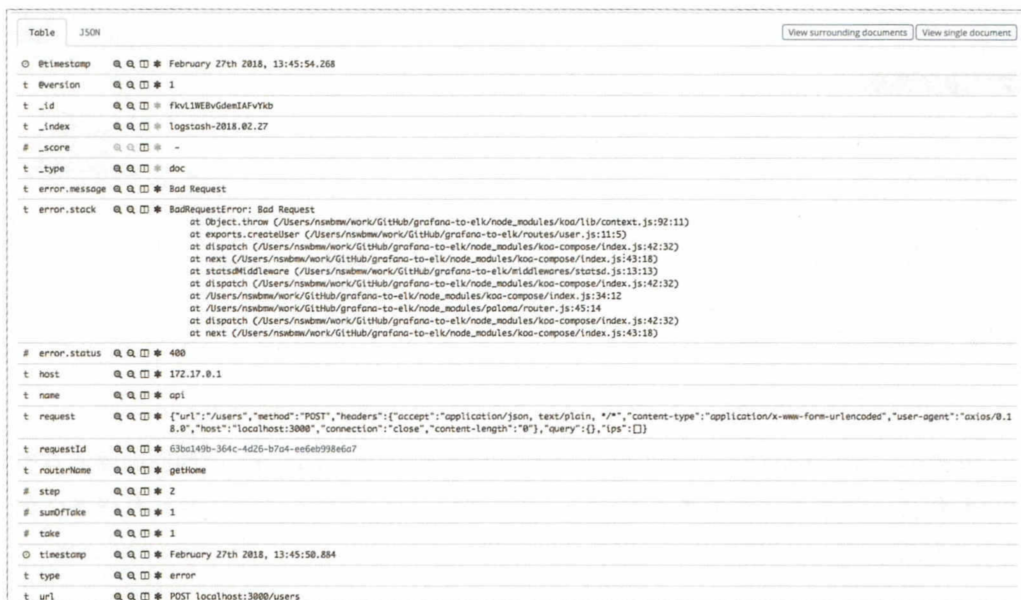


图 7-13

7.2.2 监控报警

Grafana 有内置的监控报警，设置步骤如下。

进入 Alerting -> Notifications 页，单击 New Notification 添加新的报警组，如图 7-14 所示。

仅供非商业用途或交流学习使用

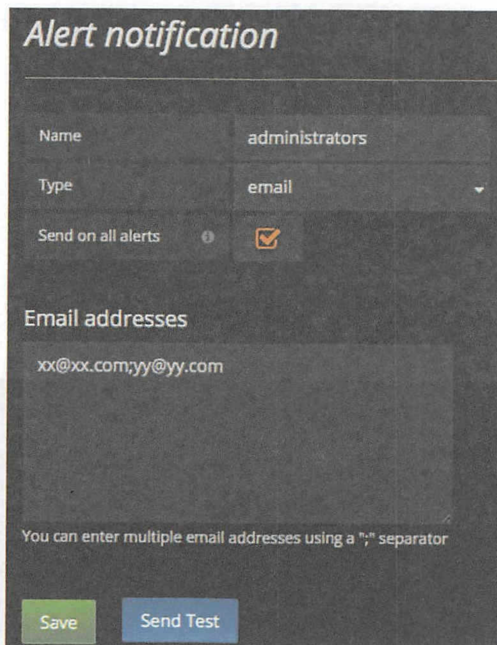


图 7-14

回到“getHome 响应时间”图表，进入编辑页的 Alert tab，单击 Create Alert 创建报警规则，如图 7-15 所示。

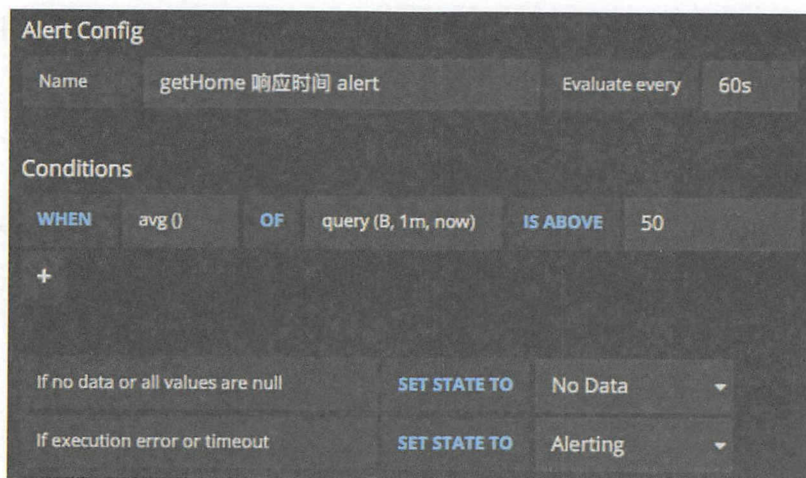


图 7-15

仅供非商业用途或交流学习使用

报警规则为：每 60s 检查一次过去 1 分钟的 mean（B 在 Metrics 里面代表了别名为 mean 的折线图）折线图的平均值是否大于 50，如果是则触发报警。

注 意

如需发邮件，则需要设置 Grafana 的 SMTP settings。

我们还可以给“getHome 请求量”设置错误报警监控，如图 7-16 所示。

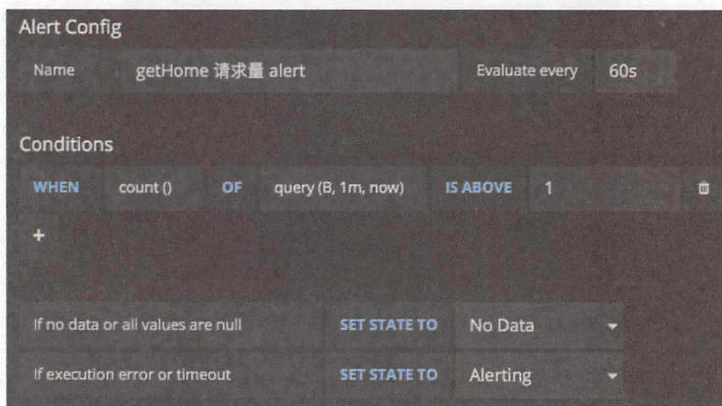


图 7-16

每 60s 检查一次在过去 1 分钟内是否有 400 报错，如果有则触发报警，其中 B 代表了别名为 400 的折线图。

小 提 示

可以将报警信息发送到 Email、Slack、DingTalk 或者 Webhook 等。报警的内容可以包含图表的截图，需要配置 external image uploader。

7.2.3　脚本一键生成图表

我们只创建了一个接口的两种（请求量和响应时间）图表，对每个图表都设置 link、alert 等就很麻烦了。如果我们的 api 有几百个接口，这岂不成了灾难?

虽然我们可以使用 Grafana 的 Template 的功能，但是该功能不够强大，我们在这里使用另一个技巧。

我们在保存图表时，从 Chrome DevTools 的 Network 中看到发起了一个 Ajax 请求，如图 7-17 所示。

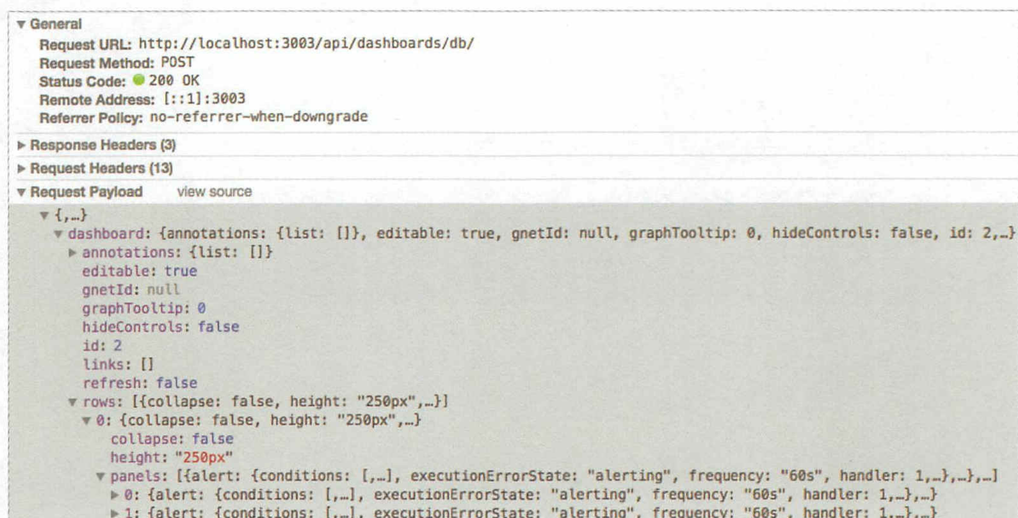


图 7-17

dashboard 就是包含了当前仪表盘页所有图表的完整的 JSON，JSON 中的字段如下。

- dashboard：包含一到多行 row。
- rows：一行 row 包含一到多个 panel。
- panels：一个 panel 就是一个具体的图表。

在拿到这个 JSON 后，我们就可以不断地尝试修改它，然后用 axios 带上在浏览器中拿到的 Cookie 发送到图中的 URL，模拟浏览器的保存操作，这里就不再展开讲解了。

本节的参考链接如下。

- http://docs.grafana.org/alerting/notifications/

第8章
应用

本章将会讲解两个 Node.js 应用服务整体解决方案。

书籍是人类进步的阶梯

8.1 使用 node-clinic

node-clinic（简称 clinic）是一个开箱即用的 Node.js 应用诊断工具。

首先，安装 node@9+：

```
$ nvm install 9
```

全局安装 clinic：

```
$ npm i clinic -g
```

然后创建测试代码。

app.js：

```
const Paloma = require('paloma')
const app = new Paloma()

function sleep (ms) {
  const future = Date.now() + ms
  while (Date.now() < future);
}

app.use(() => {
  sleep(50)
})

app.listen(3000)
```

使用 clinic doctor 启动并诊断 Node.js 应用：

```
$ clinic doctor -- node app.js
```

使用 ab 压测：

```
$ ab -c 10 -n 200 "http://localhost:3000/"
```

通过 CTRL+C 终止测试程序，终端打印出：

```
Warning: Trace event is an experimental feature and could change at any time.
```

书籍是人类进步的阶梯

```
^Canalysing data
generated HTML file is 51485.clinic-doctor.html
```

用浏览器打开 51485.clinic-doctor.html，结果如图 8-1 所示。

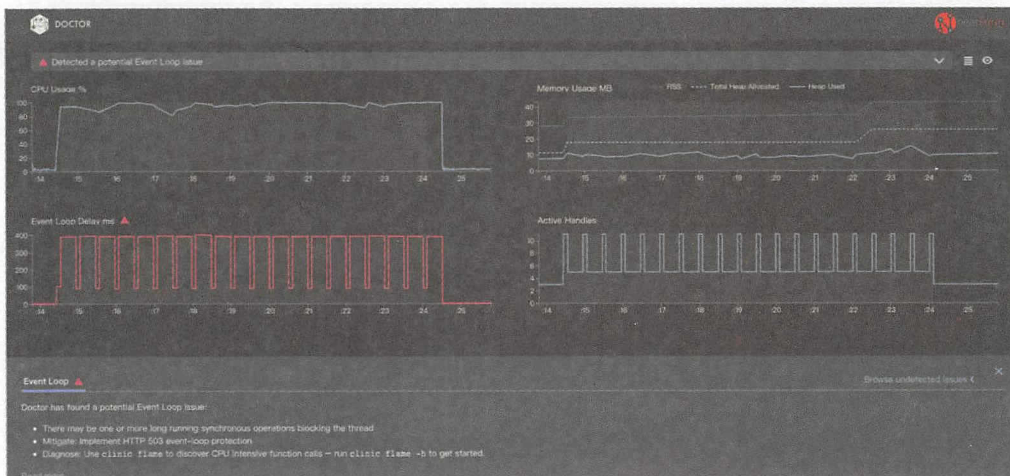


图 8-1

可以看出，Event Loop 被阻塞，CPU Usage 也居高不下，一定是有 CPU 密集型计算，与我们的测试代码吻合。

clinic 也给出了猜测和解决方案，我们尝试使用 clinic flame 生成火焰图：

```
$ clinic flame -- node app.js
```

也可以用以下命令代替：

```
$ clinic flame --collect-only -- node app.js # 只收集数据
$ clinic flame --visualize-only PID.flamegraph # 将数据生成火焰图
```

在使用同样的 ab 命令压测后，生成的火焰图如图 8-2 所示。

书籍是人类进步的阶梯

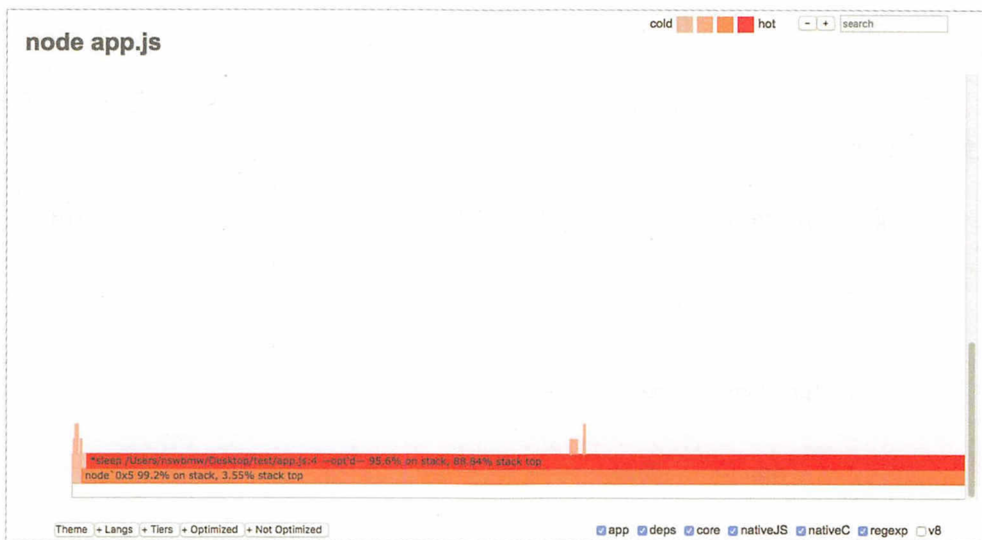


图 8-2

可以看出，app.js 的第 4 行的 sleep 函数占用了大量的 CPU 计算。

本节的参考链接如下。

- https://www.nearform.com/blog/introducing-node-clinic-a-performance-toolkit-for-node-js-developers/

8.2 alinode

8.2.1 什么是 alinode

Node.js 性能平台（原 alinode）是为中大型 Node.js 应用提供性能监控、安全提醒、故障排查、性能优化等服务的整体性解决方案。alinode 团队凭借对 Node.js 内核的深入理解，提供了完善的工具链和服务，协助客户主动、快速地发现和定位线上问题。

书籍是人类进步的阶梯

8.2.2 创建 alinode 应用

访问官网 https://www.aliyun.com/product/nodejs，如未开通，则使用阿里云账号登录并免费开通即可。

在登录后进入控制台，单击“创建新应用”，创建一个名为 test_alinode 的应用。

进入设置页面，如图 8-3 所示。

图 8-3

在后面会用到 App ID 和 App Secret。

8.2.3 安装 alinode

alinode 的整套服务由 alinode 运行时、agenthub（原 agentx + commdx 命令集）和服务平台组成，所以在自己的服务器上部署时需要安装 alinode 运行时和 agenthub。

我们使用交互式一键安装 alinode 和 agenthub：

```
$ uname -a # 阿里云 ECS Ubuntu@16.04
Linux nswbmw 4.4.0-105-generic #128-Ubuntu SMP Thu Dec 14 12:42:11 UTC 2017
x86_64 x86_64 x86_64 GNU/Linux
$ wget https://raw.githubusercontent.com/aliyun-node/alinode-all-in-one/
master/alinode_all.sh
$ bash -i alinode_all.sh # App ID 和 App Secret 填写上面生成的
...
$ node -p 'process.alinode' # 查看 alinode 版本
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

注 意

如果遇到 wget 报错：wget: unable to resolve host address 'raw.githubusercontent.com'，则需要修改 DNS 配置，在 /etc/resolv.conf 最上面添加 nameserver 8.8.8.8。

生成一个 yourconfig.json 配置文件，内容如下：

```
{
  "server": "agentserver.node.aliyun.com:8080",
  "appid": "xxx",
  "secret": "xxx",
  "logdir": "/tmp/",
  "reconnectDelay": 10,
  "heartbeatInterval": 60,
  "reportInterval": 60,
  "error_log": [],
  "packages": []
}
```

使用该配置启动 agenthub：

```
$ nohup agenthub yourconfig.json &
```

agenthub 将以常驻进程的方式运行。

8.2.4 使用 alinode 诊断内存泄露

这里以一段内存泄露代码为例，演示如何使用 alinode 调试内存泄漏的问题。代码如下。

server.js：

```
const Paloma = require('paloma')
const session = require('koa-generic-session')
const app = new Paloma()

app.keys = ['some secret']
app.use(session())
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
class User {
  constructor () {
    this.name = new Array(1e6).join('*')
  }
}

app.use((ctx) => {
  ctx.session.user = new User()
  ctx.status = 204
})

app.listen(3000)
```

在这段代码中内存泄露的原因是：koa-generic-session 默认将 session 信息存储到了内存中。

client.js：

```
const axios = require('axios')

setInterval(() => {
  axios.get('http://localhost:3000')
}, 1000)
```

打开两个终端，分别运行：

```
$ ENABLE_NODE_LOG=YES node server
# 开启 alinode 的 log 功能，使得 agenthub 可以监控内核级的性能数据
$ node client # 1s 发起一次请求
```

过一会儿就可以在 alinode 控制台中看到数据了，如图 8-4 所示。

书籍是人类进步的阶梯

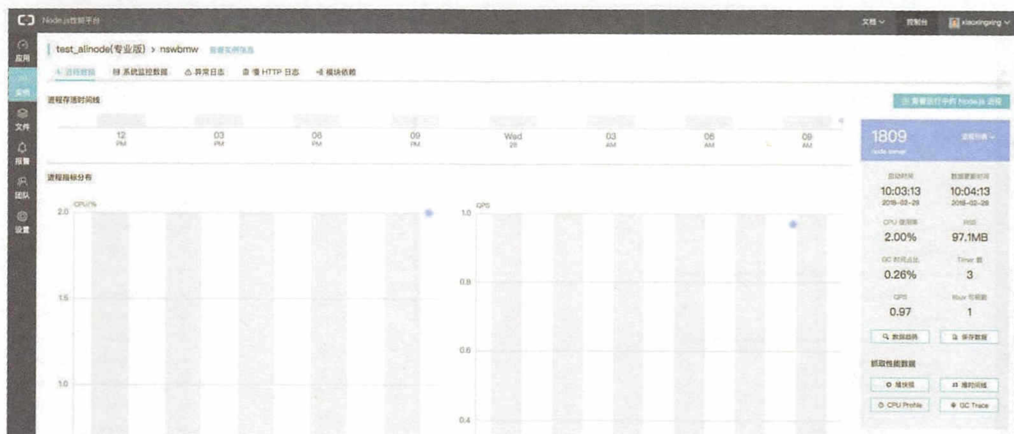


图 8-4

可以看出，alinode 监控了异常日志、慢 HTTP 日志、模块依赖和系统监控数据（包含非常详尽的图表数据，有 Memory、CPU、Load、QPS、GC、Apdex、Apdex detail、node 进程数和磁盘）。

单击“堆快照”，生成一个 heapsnapshot 文件，单击左侧的“文件”，查看刚才生成的堆快照，如图 8-5 所示。

图 8-5

在转储后单击“分析”，选择“对象簇视图”的树状列表，展开后如图 8-6 所示。

书籍是人类进步的阶梯

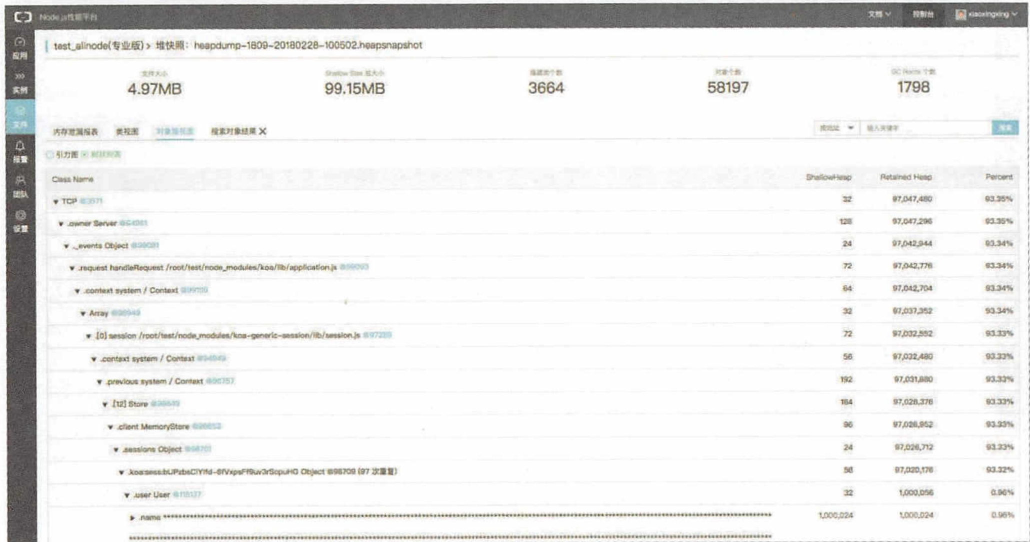


图 8-6

可以看出，在 MemoryStore 的 sessions 对象中存储了 97 个 session，并且在每个 session.user 上有一个 name 字段是长字符串。

8.2.5 使用 alinode 诊断 CPU 性能瓶颈

测试代码如下。

server.js：

```
const crypto = require('crypto')
const Paloma = require('paloma')
const app = new Paloma()

app.route({ method: 'GET', path: '/encrypt', controller: function
encryptRouter (ctx) {
  const password = ctx.query.password || 'test'
  const salt = crypto.randomBytes(128).toString('base64')
  const encryptedPassword = crypto.pbkdf2Sync(password, salt, 10000, 64,
'sha512').toString('hex')

  ctx.body = encryptedPassword
```

书籍是人类进步的阶梯

```
}})

app.listen(3000)
```

client.js：

```
const axios = require('axios')

setInterval(() => {
  const tps = Math.floor(Math.random() * 10)
  for (let i = 0; i < tps; i++) {
    axios.get('http://localhost:3000/encrypt?password=123456')
  }
  console.log(`Sent ${tps} requests`)
}, 1000)
```

打开两个终端，分别运行：

```
$ ENABLE_NODE_LOG=YES node server
$ node client
```

回到 alinode 控制台，单击“CPU Profile”，然后到“文件”查看刚才生成的 cpuprofile 文件，转储后单击“分析”，可以看到生成的火焰图。展开后如图 8-7 所示。

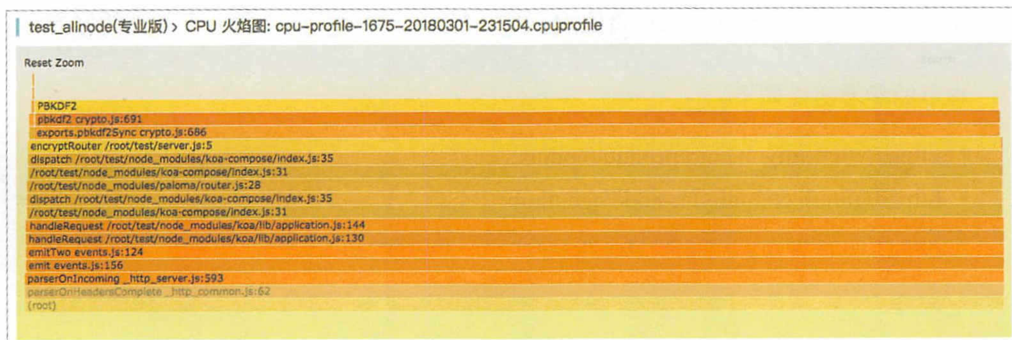


图 8-7

可以看出，server.js 的第 5 行即 encryptRouter 占用的 CPU 较多，而 encryptRouter 里的 exports.pbkdf2Sync 占用了 encryptRouter 的绝大部分 CPU 时间。

回到“文件”，选择“devtools 分析”，如图 8-8 所示。

书籍是人类进步的阶梯

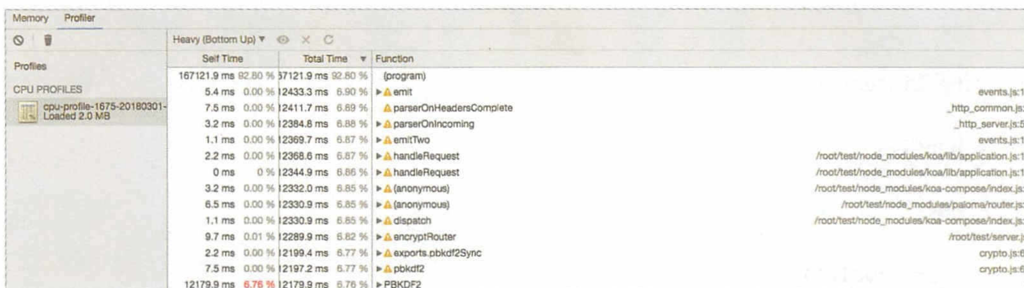


图 8-8

可以看出，alinode 已经帮我们把可疑的 CPU 性能瓶颈的元凶标红显示了。

小提示

不管是生成的 heapsnapshot 还是 cpuprofile，都可以选择在下载后使用 Chrome DevTools 分析。

我们在上面只演示了堆快照和 CPU Profile 的使用，alinode 支持抓取堆快照、堆时间线、CPU Profile、GC Trace 和 Heap Profile 这 5 种数据，本节就不一一演示了。

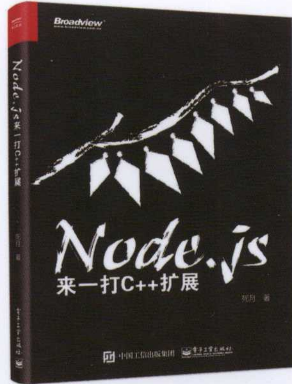
alinode 如此强大，而且可以免费使用，可以说是开发 Node.js 应用必不可少的好方案。

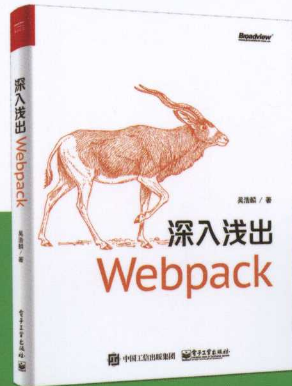
本节的参考链接如下。

- https://www.aliyun.com/product/nodejs
- https://github.com/aliyun-node/agenthub
- https://cnodejs.org/topic/561f289b4928c5872abc18ee

书籍是人类进步的阶梯

好书分享





Broadview®
www.broadview.com.cn

博文视点 · IT出版旗舰品牌

技术凝聚实力 · 专业创新出版

本书从CPU、内存、代码、工具、APM、日志、监控、应用这8个方面讲解如何调试Node.js，大部分小节都会以一段经典的问题代码为例进行分析并给出解决方案。

- 第1章 讲解CPU相关的知识，涉及各种CPU性能分析工具及火焰图的使用；
- 第2章 讲解内存相关的知识，例如Core Dump及如何分析heapsnapshot文件；
- 第3章 讲解代码相关的知识，例如如何从代码层面避免写出难以调试的代码，并涉及部分性能调优知识；
- 第4章 讲解工具相关的知识，涉及常用的Node.js调试工具和模块；
- 第5章 讲解APM（Application Performance Management）相关的知识，例如两个不同的应用程序性能管理工具的使用；
- 第6章 讲解日志相关的知识，例如如何使用Node.js的async_hooks模块实现自动日志打点，并结合各种工具进行使用；
- 第7章 讲解监控相关的知识，例如如何使用Telegraf + InfluxDB + Grafana搭建一个完整的Node.js监控系统；
- 第8章 讲解应用相关的知识，给出了两个完整的Node.js应用程序的性能解决方案。

本书并不适合Node.js初学者，适合有一定Node.js开发经验的人阅读。可将本书定位成参考书，书中的每一小节基本独立，如果遇到相关问题，则可以随时翻到相应的章节进行阅读。

欢迎加入Node.js实战QQ群：156627943

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

· 提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

· 交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34146

上架建议：JavaScript > Node.js

博文视点Broadview

@博文视点Broadview

ISBN 978-7-121-34146-5

定价：89.00元

策划编辑：张国霞
责任编辑：徐津平
封面设计：侯士卿

欢迎投稿
邮箱：zhanggx@phei.com.cn
微信：zgx228