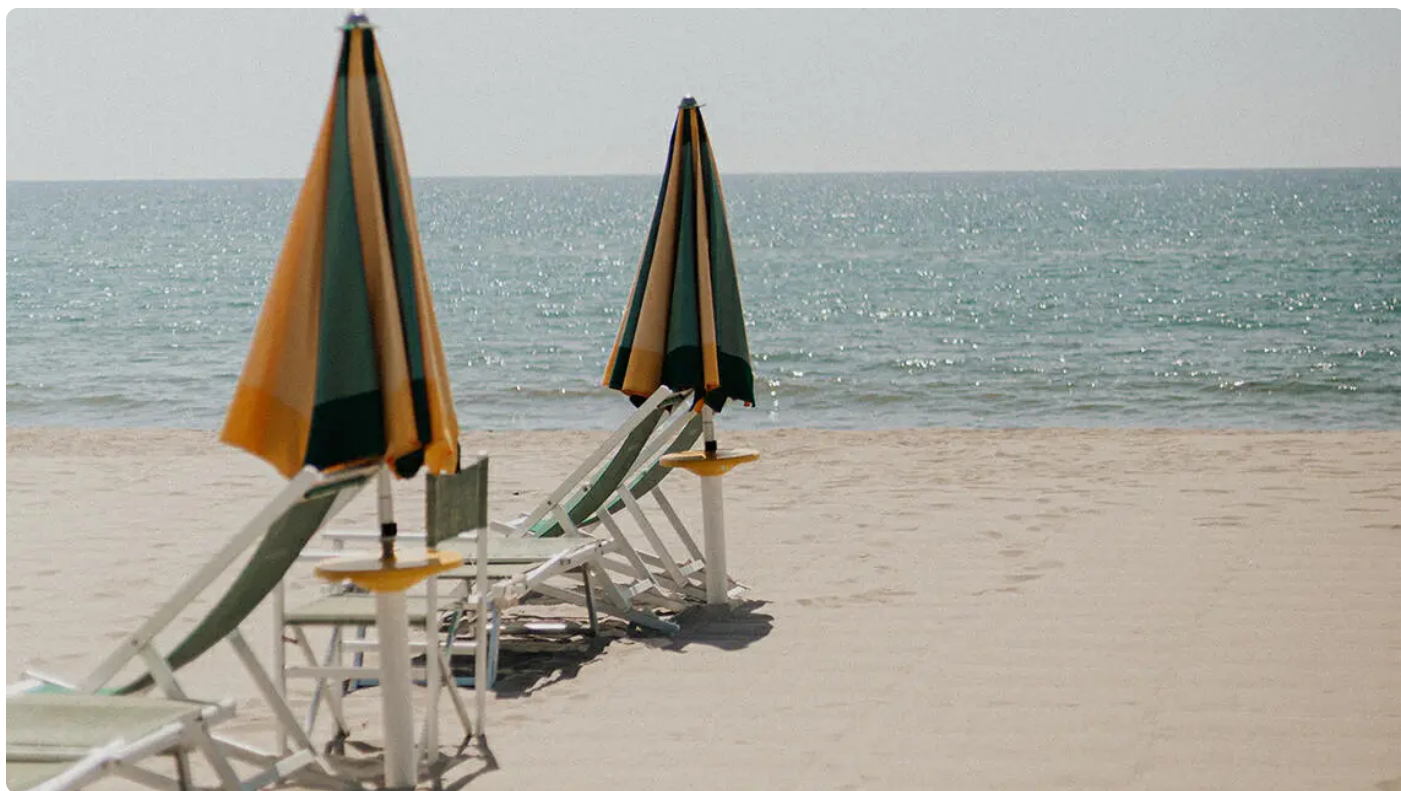


36 | 日志收集与问题排错：如何守护好Vue.js和Node.js的全栈项目？

2023-03-06 杨文坚 来自北京

《Vue 3 企业级项目实战课》

课程介绍 >



讲述：杨文坚

时长 17:50 大小 16.29M



你好，我是杨文坚。

经过前面的学习，我们从 Vue.js 的常见开发使用场景开始，建设了自己的 Vue.js 组件库，并在运营搭建平台的开发实践中，实际使用了组件库，然后我们对搭建平台全栈项目进行了功能设计和技术方案实现，最后也就是增强篇中，我们对搭建平台进行了功能增强，扩展了全栈项目的日志监控、页面性能优化，以及功能扩展建设。

这么一看，你是不是感觉基于 Vue.js 和 Node.js 的搭建平台项目大功告成，项目可以宣告完结了呢？其实还不算。在企业日常工作中，只要项目一天在线运行，并且提供功能给用户或客户使用，就不算完结。除非业务方确认项目完成历史使命，下线功能不再维护，那才算是项目的完结时刻。

所以现阶段，我们充其量是满足了运营搭建平台的功能需求。实现完功能需求后，接下来的主要工作有两大部分。

- 功能扩展
- 功能维护

前面我们花两节课学习了搭建平台的功能扩展，今天就围绕最后一个工作重点——运营搭建平台的功能维护，来学习如何守护好运营搭建平台的全栈项目。

如何理解搭建平台的功能维护

不过“功能维护”这个词太宽泛了，放在不同项目语境里，都有不同的理解。如果想理解具体内容，我们需要先限定在一个具体的项目背景里。那么围绕我们的运营搭建平台，功能维护的操作有哪些呢？

主要可以分成这样三类。

- 依赖升级维护
- 全栈问题处理
- 系统稳定运维

1. 依赖升级维护

依赖升级维护，主要是定期对 Node.js 版本、项目依赖的 npm 模块做升级，以及升级后的代码兼容处理。

这里可能你会有疑问，**如果项目代码一直运行稳定，功能也够用，Node.js 和 npm 模块还有升级的必要吗？**我的回答是“有必要”，这是从安全角度出发考虑的。

Node.js 按照官方正常的维护节奏，每年都会发布两个大版本号，主要是新增或废弃功能、提升性能或者解决重大问题。按照惯例，奇数版本主要新增实验新功能，偶数版本为稳定功能版本。在此期间，也会不定期发布一些小版本号迭代，主要是解决一些小问题和日常功能维护优化。

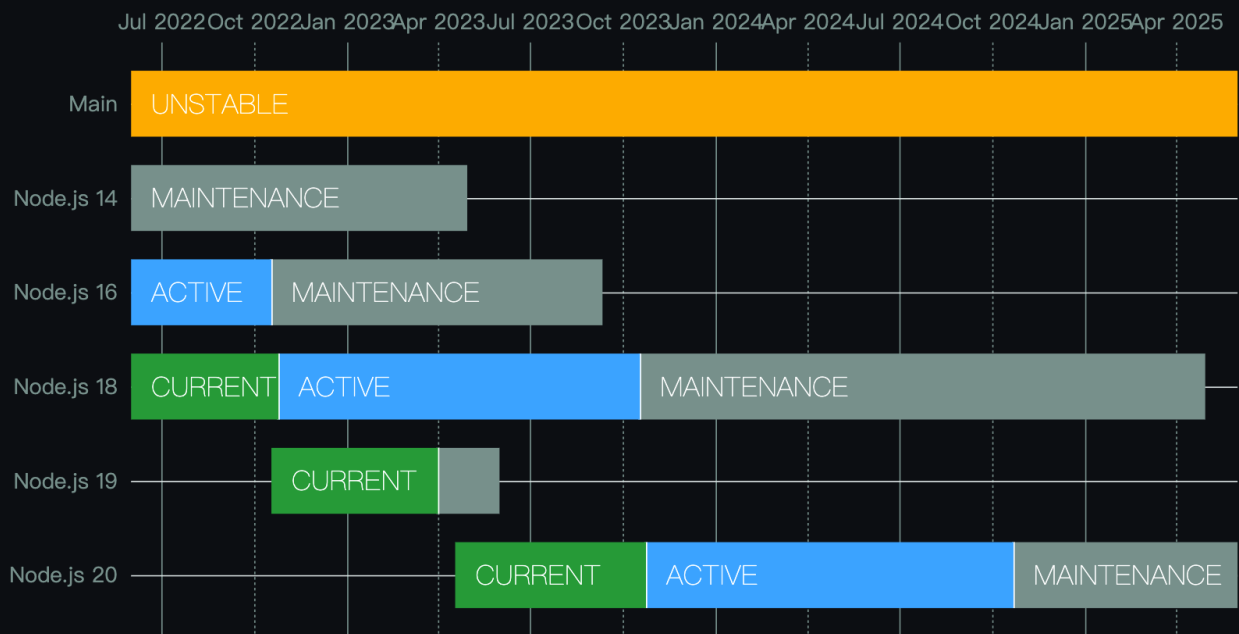
而且，每个大版本号都有对应维护的排期（截图来自 [Node.js 的 GitHub 官方文档](#)）。

Node.js Release Working Group

Release schedule

Release	Status	Codename	Initial Release	Active LTS Start	Maintenance Start	End-of-life
14.x	Maintenance	Fermium	2020-04-21	2020-10-27	2021-10-19	2023-04-30
16.x	Maintenance	Gallium	2021-04-20	2021-10-26	2022-10-18	2023-09-11
18.x	LTS	Hydrogen	2022-04-19	2022-10-25	2023-10-18	2025-04-30
19.x	Current		2022-10-18	-	2023-04-01	2023-06-01
20.x	Pending		2023-04-18	2023-10-24	2024-10-22	2026-04-30

Dates are subject to change.



我们从截图可以看到，Node.js 的第 16 版本定在“2023-09-11”后停止维护，这意味着用到 16.x 版本的 Node.js 需要在这个时间前升级版本。如果你的项目使用的是 Node.js 16.x 版本，但没有及时升级版本，一旦出现安全漏洞，官方是不会修复问题的，这时候你能做的事情就是升级项目的 Node.js 版本。

其它 npm 模块的版本情况也类似，我们要定期查询最新 npm 的更新状态，然后做好升级工作和兼容测试工作。

2. 全栈问题处理

功能维护操作的第二个内容就是全栈问题处理，可以分成两个部分，问题收集和处理问题。

问题收集主要靠记录日志，Node.js 服务在系统中运行的时候，我们要准备好日志记录的代码，在服务通用错误监听和业务关键环节上，做好日志收集工作。

有了收集到的日志数据，一旦 Node.js 服务发生异常问题，导致用户功能不可用，我们就可以基于日志数据辅助问题定位。

3. 系统稳定运维

功能维护操作的最后一点，系统稳定运维，主要是维持系统长时间稳定运行，保证系统的高可用性。

系统运维的关注点是 Node.js 服务在系统中的资源使用情况，以及服务的健康状况。我们需要根据情况信息，做出对应的处理措施，比如 Node.js 服务运行日志文件过多，磁盘空间不够用，就要及时清理日志文件，腾出磁盘空间。

功能维护，总的来说，**第一点“依赖升级维护”就是日常环境和模块的升级迭代**，除了一些 API 发生重大变更，需要特殊情况特殊处理，剩下的都是常规操作，例如升级版本号、回归测试验证功能。

后两点“全栈问题处理”“系统稳定运维”其实目标是一致的，实际工作内容也基本一致，**都是保证整个系统和服务应用的高可用性。**

我们在课程里提到的“功能维护”，主要集中在系统的日志收集和稳定维护上，这些工作内容，不知道你会不会觉得“既陌生又熟悉”呢？“陌生”是因为这些内容完全不是我们前端工程师的职责范围，在工作分工中，应该是运维工程师的职责工作，所以也很“熟悉”。

那么问题来了，作为前端程序员，我们需要懂系统运维吗？

为什么前端程序员要懂运维工作

先给出答案，作为前端程序员，我们需要懂系统运维。站在项目和企业的综合视角上看，有三个关键因素。

1. 全栈项目包括了服务端内容。
2. 企业分工现状的局限，开发也即运维。
3. 项目技术的第一责任人划分。

第一点，我们课程的搭建平台项目，是围绕着 **Vue.js** 来做页面搭建基础，离不开用 **Node.js** 搭建服务处理 **Vue.js** 代码。因此涉及 **Node.js** 服务端代码的上线部署工作，是离不开运维知识的。

第二点，在目前企业里技术岗位的分工理念中，虽然常说“专业的事，交给专业的人来做”，但是这种说法过于理想化。即使企业有运维工程师这一个角色，也不是所有项目都能被兼顾到。

因为，企业需要考虑成本问题，不会对每个项目分配专业的运维工程师。而且对大厂来说，运维工程师数量也比较稀少，他们主要是做一些大方向的运维工作，比如辅助开发程序员处理机器资源申请、数据库扩容、域名配置和系统安全维护等等。如果涉及具体项目的功能维护，或者更细致的系统运维工作，就是由项目主导的程序员来承当。

我们课程的运营搭建平台，是由前端程序员主导的 **Node.js** 全栈项目，放在大厂环境里，主要的功能运维工作也是前端来承担。

最后也是最重要的一点，项目技术第一责任人的划分。我们课程的项目毫无疑问，前端程序员是第一责任人，所以系统运维，也会由前端工程师，作为第一责任人。

好，理清了为什么要懂运维工作，我们回到课程的搭建平台项目，看看前端程序员在系统运维的过程里，具体需要关注什么。

搭建平台的系统运维需要关注什么

系统运维，是一个很宽泛的技术领域，涉及的知识面非常广，不过围绕搭建平台这个项目，对于我们前端程序员来讲，系统运维的关注点有三个。

- 机器配置
- 机器资源使用情况
- **Node.js** 服务的健康情况

第一点，系统运维要关注服务机器的配置情况，这里的“配置”指 **CPU** 核数、内存大小、磁盘空间大小等具体参数。关注这些，主要让我们对机器配置有概念，让自己心里有底，知道机器是否能扛得住用户使用量。

比如搭建平台的后台服务，使用单机“**4 核 CPU+2G 内存 +20G 磁盘空间**”的机器，运行 **Node.js** 服务，提供给企业内部几十人使用，性能已经绰绰有余了。如果是搭建平台的前台服务，面对的是十几万级别的外部客户，就需要集群机器来部署服务了。

第二点，机器资源使用情况，指的是 **Node.js** 服务在正常运行状态下，**CPU**、内存和磁盘空间的使用情况。我们需要对这些数据有最基本的概念，项目正常运行时，单机 **CPU**、内存和磁盘空间的使用率是多少，才能维持在稳定状态，不会出现异常起伏。

比如早上 **9 至 10** 点左右一般是流量高峰，今天的机器资源数据，同比前几天的数据是否差不多，如果异常，就需要保持对服务健康状态的关注了；比如观察机器 **Nginx** 等网关服务的流量数据情况，同比前几天数据，是否稳定。

第三点，关注 **Node.js** 服务的健康状态情况。这里的“健康”没有一个准确、通用的指标，我们只有通过对比数据来判断。比如搭建平台正常运行时，每天错误日志文件大致占用多少磁盘空间，以此作为参考系来定义为“正常状态”，如果某天错误日志文件剧增，就是“不健康状态”。

对于 **Node.js** 服务来说，“健康情况”可以分成“服务是否正常”和“日志是否异常”两种场景。

- “服务是否正常”，指 **Node.js** 的 **Web** 服务进程是否存在，进程资源使用情况同比前几天数据是否维持稳定。
- “日志是否异常”，指 **Node.js** 代码本身记录日志数据是否出现激增的情况，或者日志文件是否突然占用大量磁盘空间。如果是，意味着搭建平台存在问题且问题都记录在磁盘日志文件中。

前二点机器配置、机器资源使用情况，我们其实可以通过服务器或云服务厂商提供的运维数据、运维建议和运维工具来管理。

第三个关注点，**Node.js** 服务的健康情况，就需要程序员做一些开发工作，来辅助 **Node.js** 服务的健康状况的维护，这个工作就是我们接下来要学习的——检查 **Node.js** 服务健康状况。

如何检查 Node.js 服务健康状况

Node.js 服务健康状况的检查，主要就是检查 Node.js 服务进程的资源使用情况。其中，最基本的内容就是进程的内存和 CPU 使用数据，通常有两种方式。

- Node.js 服务自己检查自己
- 用系统工具来检查 Node.js 服务

第一种方式，Node.js 服务自身监控，主要是利用 Node.js 自带的能力，来获取当前进程状态数据。

我们可以使用 Node.js 进程的 API “process.memoryUsage()” 和 “process.cpuUsage()”，分别获取当前内存和 CPU 的使用情况。然后在 Node.js 服务中，设置定时任务，通过定时调用这两个 API，把进程的 CPU 和内存使用情况，写入日志，方便检查和观察进程监控状态趋势。

第二种方式，用系统工具来监控 Node.js 服务。其实，主流的操作系统都提供了工具，方便使用者查看进程资源使用情况。

查找方式主要是通过进程 id 来进行关联搜索。比如在 Linux 或者 MacOS 操作系统中，我们可以借助“ps”命令，根据进程 id 查看内存和 CPU 使用情况。

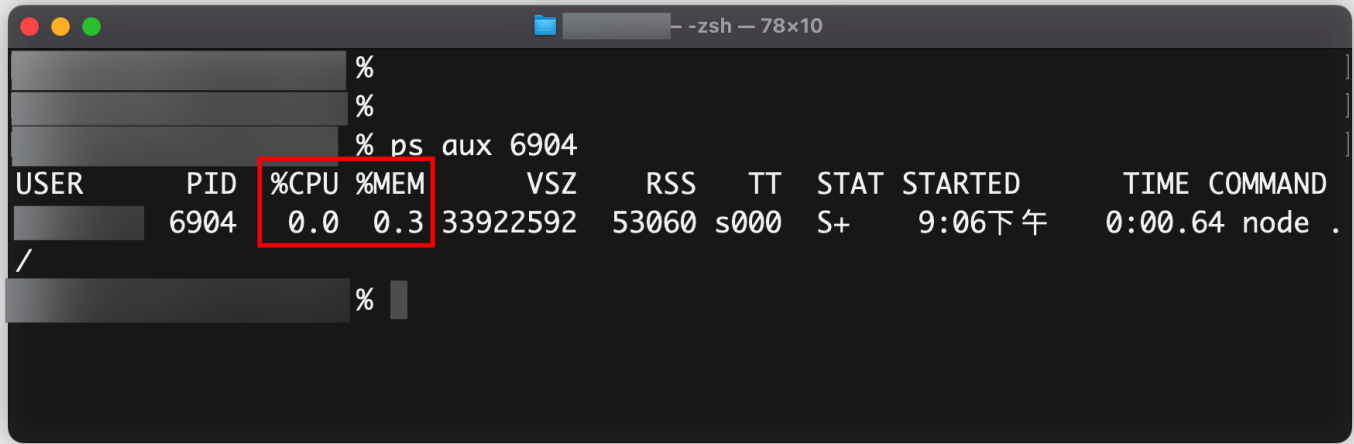
“ps”命令，是 Process Status 的简称，在大部分 Linux 和 MacOS 操作系统中，都有内置这个命令工具。

简单的使用方式比如：

```
1 ps aux ${pid}
2 ## ${pid} 为具体服务进程的id
```

 复制代码

我们试一试，查询 Node.js 服务的进程 id。

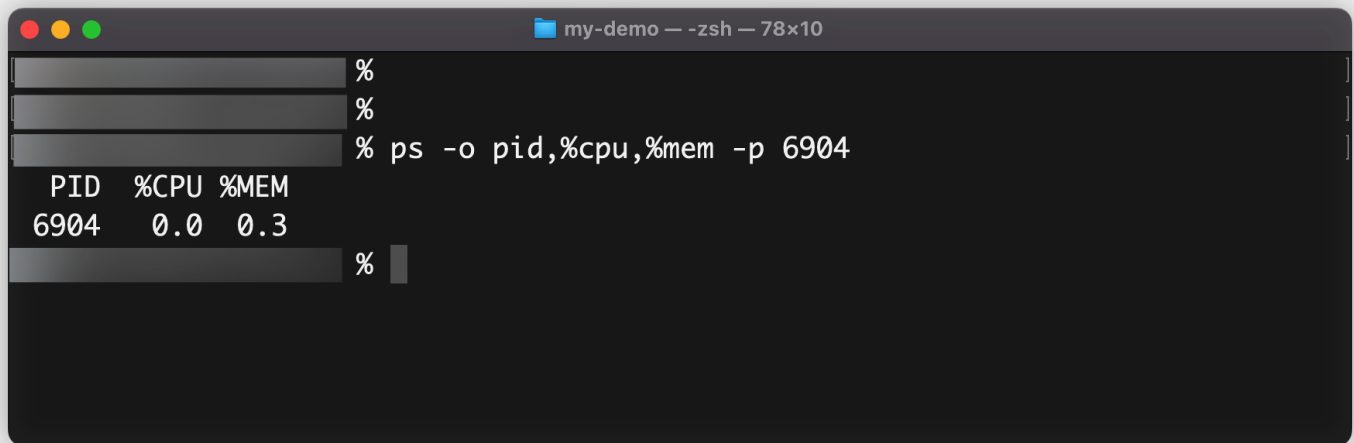


```
zsh -- 78x10
%
%
% ps aux 6904
USER      PID    %CPU %MEM    VSZ   RSS  TT  STAT STARTED  TIME COMMAND
          6904    0.0  0.3 33922592 53060 s000 S+    9:06下午 0:00.64 node .
/
%
```

从截图可以看出，通过 `ps` 命令，可以查询服务进程的资源使用情况，如果你想以其它格式查看数据，可以添加格式化配置。

 复制代码

```
1 ps -o pid,%cpu,%mem -p ${pid}
2 ## ${pid} 为具体服务进程的id
```



```
my-demo -- zsh -- 78x10
%
%
% ps -o pid,%cpu,%mem -p 6904
PID  %CPU %MEM
6904  0.0  0.3
%
```

另外，我们也可以用于子进程的方式执行“ps”命令，获取进程的内存和 CPU 数据，并记录在日志里。子进程的命令操作参考代码。

 复制代码

```
1 import childProcess from 'node:child_process';
2
3 const pid = 6904;
4 const cmd = `ps -o pid,%cpu,%mem -p ${pid}`;
5 const output = childProcess.execSync(cmd, {
6   encoding: 'utf-8'
7 });
8 console.log(output);
```


不过，这两种方式各有优劣。

- 使用 **Node.js** 的进程 API，优点是方便，但是存在风险，毕竟是在 **Node.js** 服务进程里，自己获取自己的数据，如果进程异常退出，就无法继续记录相关数据。
- 使用操作系统命令工具，优点是稳定，而且可以用其它语言启动子进程来定时执行命令，从而达到定时记录 **Node.js** 服务数据的效果。但是，系统命令的方式做好要兼容不同平台的操作，比如 **Windows** 和 **Linux** 系统查看进程状态的命令就不一样。

那么有没有折中的办法呢？答案是有的，我们也可以使用成熟工具，获取系统的 **CPU** 和内存情况。比如使用“**pidusage**”这个 **npm** 模块，内置了多种操作系统的进程查询命令，封装了统一 API 的调用方式来查看进程状态。具体 **npm** 模块链接为：

🔗 <https://www.npmjs.com/package/pidusage>。

现在，我们能通过多种方式，检查 **Node.js** 进程的健康状态了，接下来要做的事就是根据健康状态的指标，判断和记录进程“不健康情况”的快照日志。

如何判断和记录进程异常快照日志

判断和记录进程异常快照日志，我们分成两个步骤。

- 第一步，判断进程异常状态。
- 第二步，记录进程状态快照。

第一步，判断进程异常，主要建立在前面进程监控状态检查的基础上，定时获取进程的 **CPU** 和内存数据，同时，要判断 **CPU** 和内容是否超出正常指标范围。这里“正常指标”，一般指 **CPU** 或内存占用整体系统的资源比例。

对于具体“正常指标”的确定，我们要具体项目具体分析。在企业里一般认为，一些性能指标，日常资源占用情况在 **15%~20%** 以内，是健康稳定的状态。这里留了比较大的爆发增长空间作为缓冲，应付突发情况的出现，如果资源占用超过 **50%**，就属于比较严重的发展趋势，需要开始执行应急方案。

CPU 或者内存的占用红线，我们可以定为 50%，如果超过这条红线，就记录当前的 CPU 和内存快照，把当时整个 Node.js 的进程状态记录下来。需要用到两个 npm 模块，“heapdump”和“v8-profiler-next”。

- heapdump，npm 模块地址为：🔗 <https://www.npmjs.com/package/heapdump>，生成 Node.js 某一个时刻的内存快照。
- v8-profiler-next，npm 模块地址为：🔗 <https://www.npmjs.com/package/v8-profiler-next>，生成某个时刻 CPU 或内存的快照。

具体快照记录模块的使用，你可以参考 npm 各自模块的说明。

有了快照日志，我们接下来要做的是**分析快照数据**。

不过，内存、CPU 快照日志的内容和查看方式有很大差异。

- heapdump 记录的快照数据，是 Node.js V8 记录的内存堆信息。我们可以使用 Chrome 的开发者工具来打开快照日志文件，定位到某个 JavaScript 代码变量内存情况。
- v8-profiler-next 记录的 CPU 快照日志数据，是某个时刻 JavaScript 代码执行时候占用 CPU 的时间情况。我们可以使用 Chrome 的开发者工具，来打开快照日志文件，定位到某段 JavaScript 代码的耗时情况。

到这里，我们解决了 Node.js 服务进程的健康检查和异常资源问题排查，那么运维工作是否已经到尾声了呢？其实还没有，只要项目没完成业务使命下线，就会一直保持系统运维工作。

我们之前提到过，**系统运维工作，核心是保证服务高可用，这意味着要保持 Node.js 服务进程的持续稳定在线**。比如，我们用多进程方式启动服务，即使个别服务异常退出，还有剩余的子进程保持服务可用，但是碰到极端情况，多个服务进程都全部异常退出了，等于服务全都不可用了，那该怎么办呢？

这时候，我们就需要用到“进程守护”（或称“守护进程”）。

如何进行 Node.js 进程守护操作

进程守护，不是保护进程一直稳定不中断或者崩溃，而是让进程数量一直保持在指定的数量。具体就是启动多进程后，监听每一个进程的“健康状态”。如果出现个别进程中断或者退出，就自动开启一个新进程补位，让多进程的进程数保持不变。

看具体代码案例。

 复制代码

```
1 // packages/work-server/start-daemon.cjs
2 const path = require('path');
3 const cluster = require('cluster');
4 const os = require('os');
5
6 const maxProcessCount = os.cpus().length;
7 // 待启动多进程的 后台服务入口文件
8 const entryFile = path.join(__dirname, 'dist', 'index.cjs');
9
10 let reforkCount = 0;
11 const maxTryCount = 1000;
12
13 function startDaemon() {
14   if (cluster.isWorker) {
15     return;
16   }
17
18   // 在线的进程数量
19   let onlineProcessCount = 0;
20
21   // 启动主线程
22   cluster.setupMaster({ exec: entryFile });
23
24   for (let i = 0; i < maxProcessCount; i++) {
25     // 创建多个子进程
26     forkWorker();
27   }
28
29   function forkWorker() {
30     // 保证在线进程数量小于限制的进程数量
31     if (onlineProcessCount >= maxProcessCount) {
32       return;
33     }
34     onlineProcessCount++;
35     return cluster.fork();
36   }
37
38   function reforkWorker() {
39     console.log('尝试重新开启新线程 ...');
40     reforkCount++;
41     if (reforkCount >= maxTryCount) {
42       throw Error('已经超出最多尝试次数');
```

```

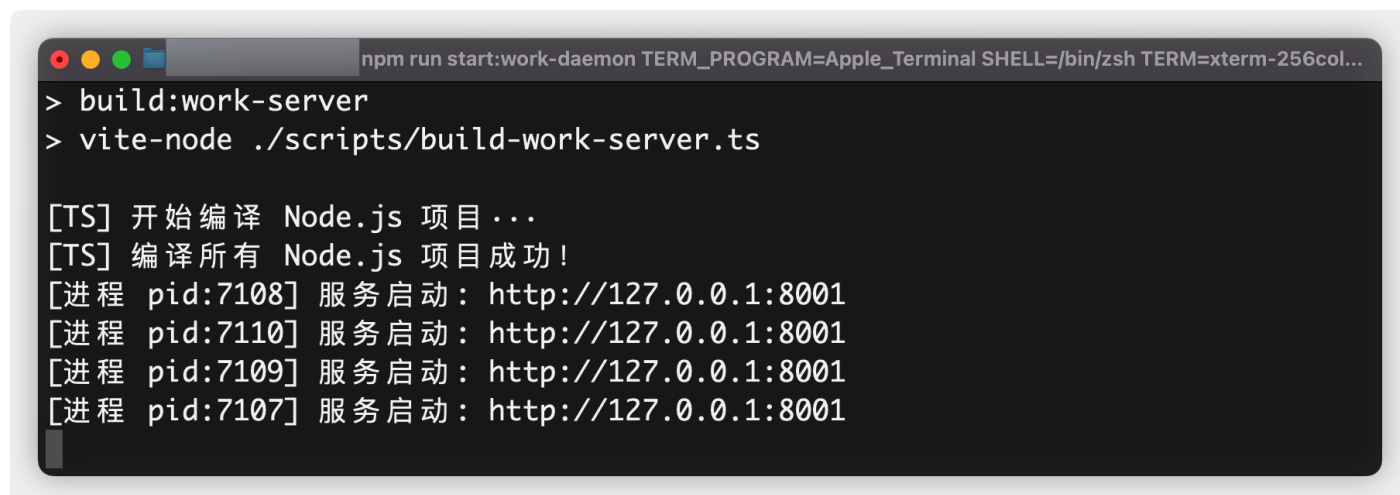
43     }
44     return forkWorker();
45 }
46
47 cluster.on('exit', (worker) => {
48     console.log(`进程 pid=${worker.process.pid} 已经退出`);
49     reforkWorker();
50 });
51
52 cluster.on('disconnect', (worker) => {
53     console.log(`进程 pid=${worker.process.pid} 已经断开连接`);
54     if (worker.isDead && worker.isDead()) {
55         console.log(`进程 pid=${worker.process.pid} 已经挂掉了`);
56         return;
57     }
58     onlineProcessCount--;
59     reforkWorker();
60 });
61 }
62
63 startDaemon();

```

在代码中，我们通过进程间的 IPC 通信，进行进程的守护管理。当个别进程出现异常中断退出的时候，就发消息给主进程，主进程就会重新开启一个新的子进程。

基于这个案例，我们可以测试一下。在开启多进程后，根据进程 id，我们强制退出进程。当子进程被强制退出后，看一看，守护进程是否会收到通信消息，自动开启新的进程。

第一张截图，我们用进程守护代码，启动了搭建平台的后台多进程服务。



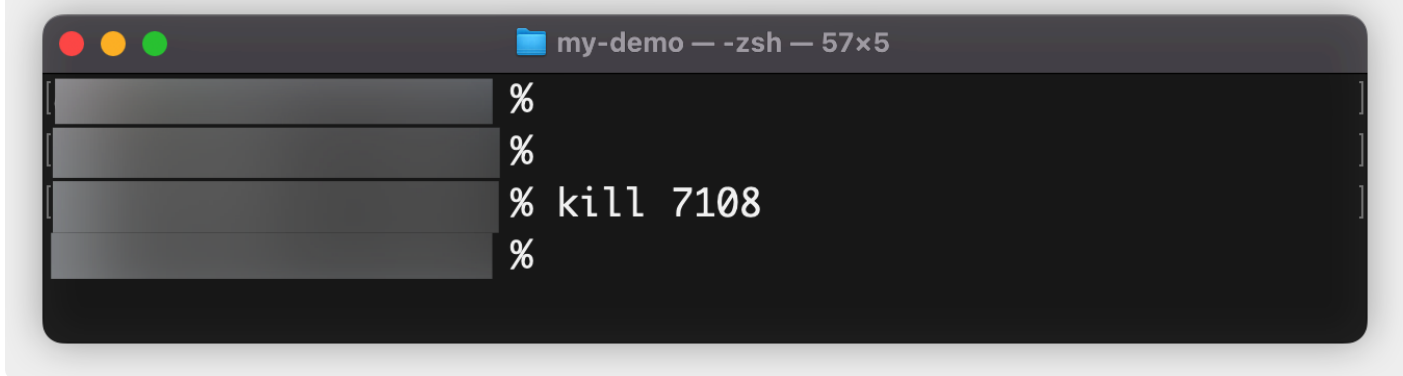
```

npm run start:work-daemon TERM_PROGRAM=Apple_Terminal SHELL=/bin/zsh TERM=xterm-256col...
> build:work-server
> vite-node ./scripts/build-work-server.ts

[TS] 开始编译 Node.js 项目...
[TS] 编译所有 Node.js 项目成功!
[进程 pid:7108] 服务启动: http://127.0.0.1:8001
[进程 pid:7110] 服务启动: http://127.0.0.1:8001
[进程 pid:7109] 服务启动: http://127.0.0.1:8001
[进程 pid:7107] 服务启动: http://127.0.0.1:8001

```

第二张截图，我们用 kill 命令，让其中一个进程退出。



第三个截图，进程守护代码，监听到子进程退出信息，自动创建新的子进程补位，保持固定进程数量。



总结

系统运维，也是我们作为程序员的必备技能。无论你的工作头衔是前端工程师，还是服务端工程师，都不能局限于前端和服务端这些纯开发领域的内容，更重要是看到项目在产线运行的维护技术领域。

系统运维，核心是要保证服务的能稳定在线，没有什么“绝对的技术方案”，都是尽量结合技术手段和人工手段。

- 技术手段，例如通过线程守护，维持服务正常在线。
- 人工手段，例如出现问题马上手动重启服务。

系统功能运维，说到底就是服务的健康状态运维。监控状态的数据内容没有固定的格式，但是有基础的指标，例如服务进程的 CPU、内存使用情况和 HTTP 请求的超时情况。可以归纳成

四个运维步骤。

- 第一步，根据项目的服务特点，做好系统服务端日志收集。
- 第二步，当服务出现问题完全不能使用时，第一时间重启服务，恢复使用状态。
- 第三步，排查服务端里的日志数据，一般从资源日志、错误日志和请求超时日志进行排查。
- 第四步，如果没找到问题原因，就需要补充日志收集代码逻辑，收集更多日志信息。

通过今天的学习，希望你能掌握服务端运维的相关知识点，实际工作中，这个部分不需要你有很专业的知识，只要能从实用主义视角出发，解决日常问题、保持服务稳定运行，面对绝大部分系统问题，能做到“兵来将挡，水来土掩”就可以了。


思考题

业界经常提到云服务、服务容器化、Serverless 等方案能解决运维成本问题，那么 Node.js 全栈项目如何选择相关服务方案呢？

期待在留言区看到你的思考。我们结束语见。

[🔗 完整的代码在这里](#)

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 35 | 多进程部署：如何最大限度利用服务器资源运行Node.js服务？

[下一篇](#) 加餐 | 增强篇思考题答疑

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。