面试官：在node中创建线程的都有哪几种方法?

在 Node.js 中，由于其单线程的特性，主线程用于执行非阻塞的 I/O 操作。但在需要执行 CPU 密集型任务时，仅依靠单线程可能会导致性能瓶颈。幸运的是，Node.js 提供了几种方法来开启和管理线程，以利用多核 CPU 的优势。

为什么要开启子线程

在 Node.js 中开启子线程的原因主要是为了处理并发任务和提高应用性能。Node.js 本身是基于事件循环的单线程模型，这意味着所有的 I/O 操作（如文件读写、网络请求等）都是非阻塞的，而 CPU 密集型任务（如大量计算）则可能会阻塞事件循环，影响整个应用的性能。

开启子线程可以帮助解决以下问题：

1. 非阻塞操作：Node.js 的设计哲学是非阻塞 I/O，但如果在主线程中直接执行外部命令，那么命令执行的过程可能会阻塞主线程，影响应用的响应性能。通过子线程执行这些命令，可以保持主线程的非阻塞特性，从而不会影响到其他并发操作。
2. 充分利用系统资源：通过使用子进程或工作线程，Node.js 应用可以更好地利用多核 CPU 的计算能力。这对于执行 CPU 密集型的外部命令尤其有用，因为它们可以在单独的 CPU 核心上运行，而不会影响到 Node.js 的主事件循环。
3. 隔离和安全：在子线程中执行外部命令可以为应用提供额外的安全层。如果外部命令执行失败或导致崩溃，这种隔离可以帮助保护主 Node.js 进程不受影响，从而提高应用的稳定性。
4. 灵活的数据处理和通信：通过子线程，你可以更灵活地处理来自外部命令的数据。例如，你可以在子进程中对命令的输出进行处理，然后将结果传回主进程。Node.js 提供了多种方式来实现进程间的通信（IPC），这使得数据交换变得容易。

最常用的场景就是我目前正在开发的 create-neat 脚手架 就经常要使用到这些开启子线层的场景，例如创建文件，执行 `npm install` 等命令。

开启子线程的几种方式

接下来我们就来讲解一下 Node 中开启子线程的几种方式。

Child Processes（子进程）

Node.js 的 child_process 模块允许你运行系统命令或其他程序，并通过创建子进程与它们通信。这可以用来执行 CPU 密集型任务或运行其他程序。

spawn

child_process 模块的 spawn()方法在 Node.js 中用于创建新的子进程，以执行指定的命令。spawn()方法返回一个带有 stdout 和 stderr 流的对象，你可以用它来与子进程进行交互。这个方法非常适合于需要处理大量输出数据的长时间运行的进程，因为它以流的形式处理数据，这意味着数据可以被逐渐读取，而不是一次性地存放在缓冲区中。

spawn()函数的基本语法如下：

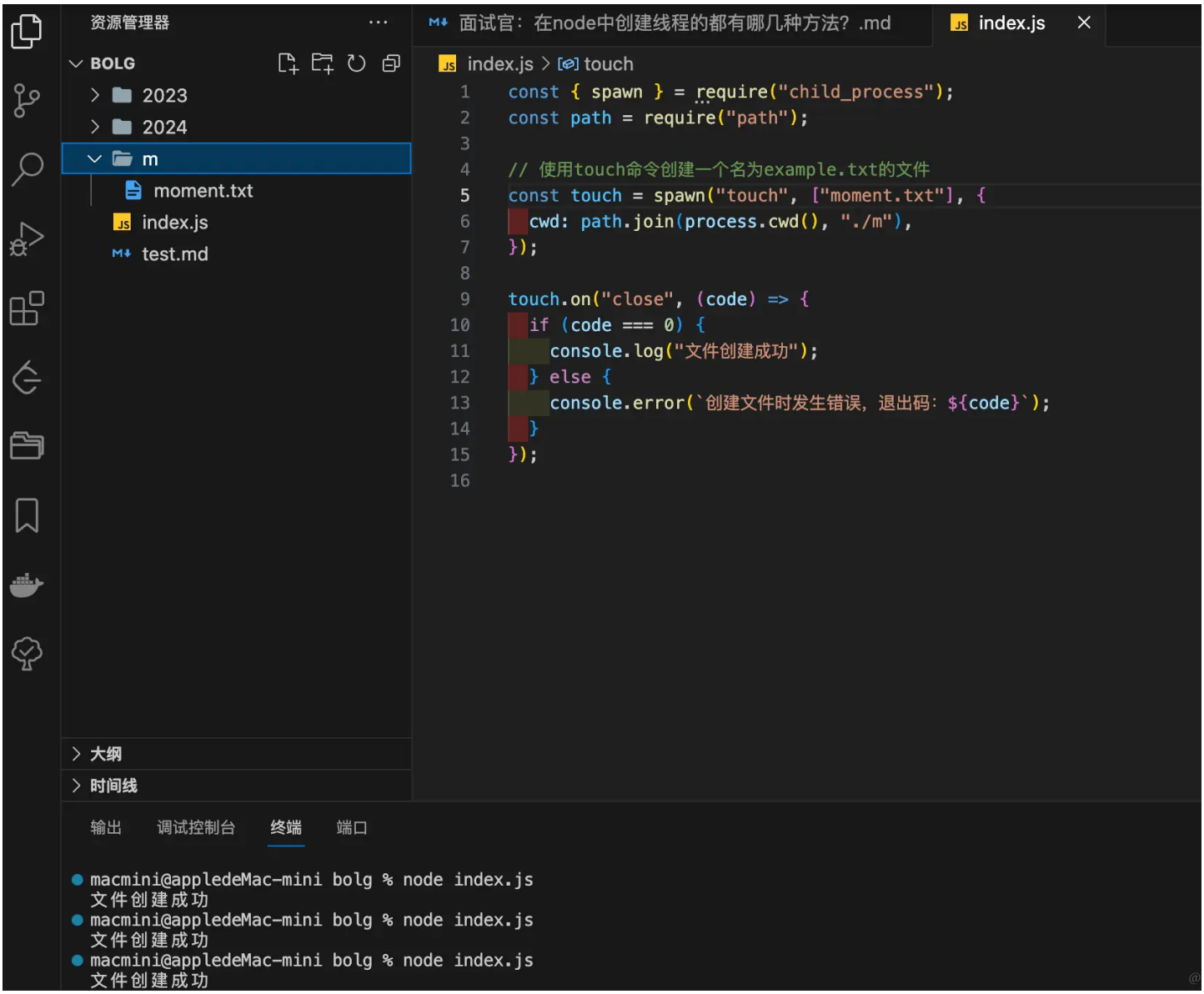
```
const { spawn } = require("child_process");
const child = spawn(command, [args], [options]);
```

1. command：一个字符串，表示要运行的命令。
2. args：一个字符串数组，列出了所有的命令行参数。
3. options：一个可选的对象，用于配置子进程的创建方式。常用选项包括：
   - cwd：子进程的当前工作目录。
   - env：环境变量键值对的对象。
   - stdio：配置子进程的标准输入输出。通常用于管道操作或文件重定向。
   - shell：如果为 true，将在 shell 中运行命令，这可以是一个指定的 shell 路径。默认情况下，Unix 上是/bin/sh，Windows 上是 cmd.exe。
   - detached：如果设置为 true，则子进程将会在其父进程独立之外运行。可以使其在父进程退出后继续运行。

以下是使用 `spawn()` 方法的一个简单例子，如下代码所示：

```
const { spawn } = require("child_process");
const path = require("path");

// 使用touch命令创建一个名为moment.txt的文件
const touch = spawn("touch", ["moment.txt"], {
  cwd: path.join(process.cwd(), "./m"),
});

touch.on("close", (code) => {
  if (code === 0) {
    console.log("文件创建成功");
  } else {
    console.error(`创建文件时发生错误，退出码：${code}`);
  }
});
```

在上面这段代码的主要目的是在当前工作目录的 m 子目录创建一个名为 `moment.txt` 的空文件，如果创建成功，你将在控制台看到 `文件创建成功` 的消息。如果有任何错误（例如，如果 m 目录不存在），则会在控制台中看到错误消息。



exec

Node.js 的 child_process.exec() 方法用于创建新的子进程，以执行给定的命令，并缓冲任何产生的输出。与 spawn() 方法不同，exec() 适用于那些输出量不大的场合，因为它会把子进程的标准输出（stdout）和标准错误输出（stderr）存储在缓冲区中。

exec() 函数的基本语法如下：

```
const { exec } = require("child_process");

exec(command, [options], callback);
```

- command：要运行的命令，作为一个字符串。

- options：可选参数，用于自定义执行环境的各种设置。
- callback：命令执行完成后的回调函数，带有 (error, stdout, stderr) 参数。

exec() 的 options 对象可以包含多种属性，比如：

- cwd：设置子进程的当前工作目录。
- env：指定环境变量对象。
- encoding：字符串编码。
- shell：用于执行命令的 shell，默认是 /bin/sh 在 UNIX 上，cmd.exe 在 Windows 上，可以被覆盖。
- timeout：设置超时时间（以毫秒为单位），当子进程运行时间超过此值时，子进程会被杀死。
- maxBuffer：指定 stdout 和 stderr 的最大缓冲区大小，默认是 `1024 * 1024`（约 1MB）。如果超出这个限制，子进程会被杀死。
- killSignal：当超时或关闭子进程时使用的信号，默认是'SIGTERM'。

exec() 方法的回调函数有三个参数：error、stdout 和 stderr：

- error：如果执行命令出错或返回非零值，则为 Error 对象；否则为 null。
- stdout：命令的标准输出。
- stderr：命令的标准错误输出。

下面是一个使用 exec() 方法执行命令的例子：

```
const { exec } = require("child_process");
const path = require("path");

// 指定要执行的命令，包括命令路径
const command = `touch ${path.join("./m", "moment.txt")}`;

exec(command, { cwd: process.cwd() }, (error, stdout, stderr) => {
  if (error) {
    console.error(`执行命令时发生错误: ${error}`);
    return;
  }
  if (stderr) {
    console.error(`标准错误输出: ${stderr}`);
    return;
  }
  console.log("文件创建成功");
});

```

最后执行代码就能看到文件的输出了。

fork()

child_process.fork() 方法是 Node.js 中 child_process 模块的一部分，它用于创建一个新的 Node.js 进程，与原进程（父进程）之间通过 IPC（Inter-Process Communication，进程间通信）通道进行通信。fork() 方法特别适用于运行 Node.js 模块的场景，在多核 CPU 上实现并行操作时尤其有用。

fork() 函数的基本语法如下：

```
1 const { fork } = require("child_process");
2
3 const child = fork(modulePath, [args], [options]);
4
```

- modulePath：一个字符串，表示要在子进程中运行的模块的路径。
- args：一个字符串数组，包含传递给模块的参数。
- options：可选参数，一个对象，用于配置子进程的创建方式。

fork() 方法的 options 对象可以包含以下属性：

- cwd：子进程的当前工作目录。
- env：环境变量键值对的对象。
- execPath：用于创建子进程的 Node.js 可执行文件的路径。
- execArgv：传递给 Node.js 可执行文件的参数列表，但不会传递给子模块。
- silent：如果设置为 true，则子进程的 stdin、stdout 和 stderr 会被重定向到父进程，否则，它们会继承自父进程。
- stdio：用于配置子进程的标准输入输出流。
- ipc：创建一个用于父进程和子进程通信的 IPC 通道。

fork() 创建的子进程自动地建立了一个 IPC 通道，允许父进程和子进程之间相互发送消息。父进程可以使用 child.send(message) 方法发送消息到子进程，子进程可以通过监听 process.on('message', callback) 事件来接收消息。同样地，子进程也可以使用 process.send(message) 向父进程发送消息。

下面的例子演示了如何使用 fork() 方法创建子进程，并通过 IPC 通道进行通信：

index.js 文件代码如下所示：

```
1 const { fork } = require("child_process");
2
3 const child = fork("./child.js");
4
```

```
child.on("message", (message) => {
  console.log("来自子进程的消息:", message);
});

child.send({ hello: "world" });

setInterval(() => {
  child.send({ hello: "world" });
}, 1000);

```

child.js 文件代码如下所示：

```
process.on("message", (message) => {
  console.log("来自父进程的消息:", message);
});

process.send({ foo: "bar" });

setInterval(() => {
  process.send({ hello: "world" });
}, 1000);

```

在这个示例中，父进程 parent.js 创建了一个子进程来运行 child.js 模块。父进程向子进程发送了一个消息，子进程接收到消息后，打印出消息内容，并回送一个消息到父进程。父进程接收到子进程发送的消息后，也打印出来。之后还有一个定时器在固定时间里互相传送消息。

最终输出结果如下图所示：

使用 fork() 时，每个子进程都是一个独立的 Node.js 实例，具有独立的 V8 实例和事件循环。这意味着创建大量子进程可能会导致大量的资源消耗。

Worker Threads（工作线程）

worker_threads 模块是 Node.js 提供的一种在单个进程中运行多个 JavaScript 任务的并行执行机制。这使得 Node.js 应用可以更有效地利用多核 CPU 资源，尤其适合处理 CPU 密集型任务，而不必创建多个进程。使用 worker_threads 可以显著提高应用性能，并使其能够处理更复杂的计算任务。

以下是 Worker Threads 的一些基本概念：

1. Worker：一个独立的线程，可以执行 JavaScript 代码。每个 Worker 运行在自己的 V8 实例中，有自己的事件循环和局部变量，这意味着它可以独立于其他 Worker 或主线程执行任务。
2. 主线程：启动 Worker 的线程通常被认为是主线程。在一个典型的 Node.js 应用中，初始的 JavaScript 执行环境（或者说是初始的事件循环）就是在主线程上。
3. 通信：Worker 和主线程之间可以通过消息传递来通信。它们可以相互发送 JavaScript 值，包括 ArrayBuffer 和其他可转移对象（transferable objects），这允许高效地在不同线程间传递数据。

以下是一个简单的示例，展示了如何创建一个 Worker，以及主线程和 Worker 之间如何通信：

```
const { Worker, isMainThread, parentPort } = require("worker_threads");

if (isMainThread) {
  // 主线程代码
  const worker = new Worker(__filename);
  worker.on("message", (message) => {
    console.log("来自Worker的消息:", message);
  });
  worker.postMessage("Hello Worker!");
} else {
  // Worker线程代码
  parentPort.on("message", (message) => {
    console.log("来自主线程的消息:", message);
    parentPort.postMessage("Hello Main Thread!");
  });
}

```

在这个示例中，index.js 文件即作为主线程运行时的入口点，也定义了 Worker 线程要执行的代码。通过检查 isMainThread 变量，我们可以区分代码是在主线程中运行还是在 Worker 线程中执行。主线程创建了一个 Worker 来执行同一个脚本，然后通过 postMessage() 方法发送消息给 Worker。Worker 接收到消息后，也通过 postMessage()回应主线程。

worker_threads 和 fork 的区别

1. 基本概念：

- worker_threads：属于 worker_threads 模块，是 Node.js 为了充分利用多核 CPU 而引入的。工作线程（Worker Threads）允许 JavaScript 和 WebAssembly 代码在新的 V8 实例上并行运行，与主线程相隔离，但可以共享内存。
- fork：是 child_process 模块的一个方法，用于创建一个新的 Node.js 进程。被 fork 的脚本与原进程相独立，拥有自己的 V8 实例，环境变量，内存空间等。通过 IPC（进程间通信）通道实现父子进程间的消息传递。

2. 通信机制：
   - worker_threads：工作线程之间以及工作线程与主线程之间的通信，主要通过 MessagePort 进行，可以传输几乎任何 JavaScript 值，包括 ArrayBuffer 和 MessageChannel 等。
   - fork：父进程与通过 fork 创建的子进程之间的通信，是通过 IPC 通道实现的。双方可以通过 process.send()方法和 message 事件进行消息传递，通信内容主要是序列化后的 JSON 对象。
3. 内存和性能：
   - worker_threads：由于工作线程可以共享内存，对于某些任务来说，这可以减少内存使用并提高性能。例如，使用 SharedArrayBuffer 实现线程间的数据共享，减少数据复制的需要。
   - fork：每个通过 fork 创建的子进程都有自己独立的内存空间和 V8 实例。这意味着，相对于 worker_threads，fork 可能会占用更多的系统资源，特别是在创建大量子进程时。
4. 适用场景：
   - worker_threads：更适合用于执行 CPU 密集型任务，比如大量计算，数据处理等，并行计算场景。
   - fork：由于能够创建独立的 Node.js 进程，更适合于需要完全隔离环境的场景，如运行独立的服务，执行与主进程环境完全不同的任务等。

总的来说，选择 worker_threads 还是 fork，主要取决于你的应用场景和对资源隔离、内存共享的需求。对于需要高度隔离的应用，或者是完全独立运行的 Node.js 应用，fork 可能是更好的选择。而对于需要高效进行并行计算、数据处理的场景，worker_threads 可能会提供更好的性能和资源利用率。

Cluster（集群）

Node.js 的 cluster 模块允许你简单地创建共享服务器端口的子进程。这是一种将单个 Node.js 实例运行在多核系统的多个 CPU 核心上的方法，从而提高应用的性能和吞吐量。在单线程的 Node.js 中，尽管非阻塞 I/O 操作使得它在处理大量并发连接时表现良好，但是 CPU 密集型的任务或简单地希望跨多核心扩展性能时，使用 cluster 模块就显得尤为重要了。

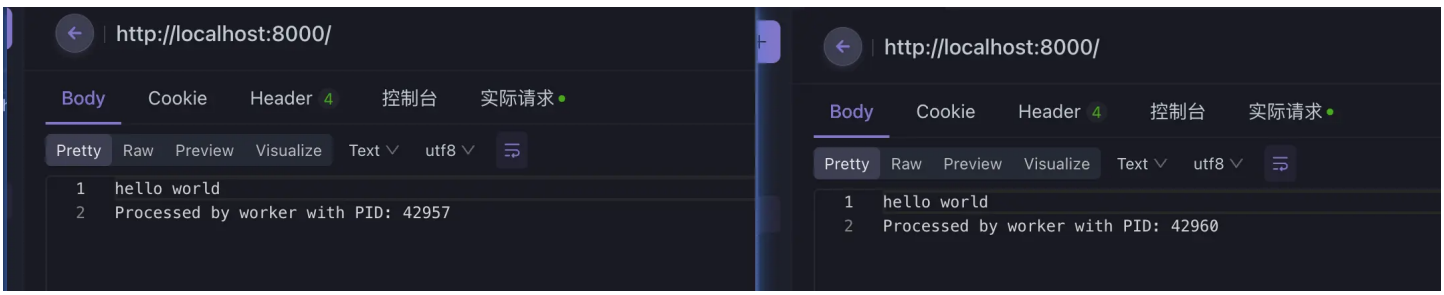
cluster 模块工作的基本原理是它允许主进程（通常称为"master"）创建多个工作进程（称为"workers"），这些工作进程实际上是当前主进程的副本。主进程管理这些工作进程，并将接收到的网络连接分发给它们。

在内部，它使用了 child_process.fork 方法来创建工作进程，这使得工作进程能够运行相同的应用代码。不同的是，它们可以通过 IPC（进程间通信）与主进程通信，接收任务或发送操作结果。

下面是一个使用 cluster 模块的简单例子：

```
const cluster = require("cluster");
const http = require("http");
const numCPUs = require("os").cpus().length;

if (cluster.isMaster) {
  console.log(`主进程 ${process.pid} 正在运行`);

  // 衍生工作进程。
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on("exit", (worker, code, signal) => {
    console.log(`工作进程 ${worker.process.pid} 已退出`);
  });
} else {
  // 工作进程可以共享任何TCP连接。
  // 在本例中，它是一个HTTP服务器
  http
    .createServer((req, res) => {
      res.writeHead(200);
      res.end("hello world\n");
    })
    .listen(8000);

  console.log(`工作进程 ${process.pid} 已启动`);
}

```

通过运行上面的代码并最终使用接口测试工具来进行访问，你会看到它输出的进行 id 是不同的：



在这个例子中，主进程创建了与 CPU 核心数量相等的工作进程，每个工作进程都是运行相同代码的独立进程。当工作进程退出时，主进程会收到 exit 事件。

尽管 cluster 可以提高应用的性能和可靠性，但它也增加了应用的复杂度，比如需要管理工作进程的生命周期、处理工作进程之间的通信等。在某些情况下，其他解决方案（如使用 pm2 等进程管理器）可

能更加适合。

cluster 模块并不适用于所有场景。例如，对于非 CPU 密集型的应用，单个 Node.js 实例可能已经足够处理所有的工作负载。

总结

子进程允许 Node.js 应用执行操作系统命令或独立运行其他 Node.js 模块，提高应用的并发处理能力。通过 exec、spawn、和 fork 等 API，开发者可以灵活地创建和管理子进程，实现复杂的异步非阻塞操作，从而在不干扰主事件循环的前提下，充分利用系统资源和多核 CPU 的优势。