进程：process模块

process 模块是 nodejs 提供给开发者用来和当前进程交互的工具，它的提供了很多实用的 API。从文档出发，管中窥豹，进一步认识和学习 process 模块：

- 如何处理命令参数?
- 如何处理工作目录?
- 如何处理异常?
- 如何处理进程退出?
- process 的标准流对象
- 深入理解 process.nextTick

如何处理命令参数?

命令行参数指的是 2 个方面：

- 传给 node 的参数。例如 `node --harmony script.js --version` 中，`--harmony` 就是传给 node 的参数
- 传给进程的参数。例如 `node script.js --version --help` 中，`--version --help` 就是传给进程的参数

它们分别通过 `process.argv` 和 `process.execArgv` 来获得。

如何处理工作目录?

通过`process.cwd()`可以获取当前的工作目录。

通过`process.chdir(directory)`可以切换当前的工作目录，失败后会抛出异常。实践如下：

```
function safeChdir(dir) {
try {
process.chdir(dir);
return true;
} catch (error) {
return false;
}
}
```

如何处理异常?

uncaughtException 事件

Nodejs 可以通过 try-catch 来捕获异常。如果异常未捕获，则会一直从底向事件循环冒泡。如是冒泡到事件循环的异常没被处理，那么就会导致当前进程异常退出。

根据文档，可以通过监听 process 的 uncaughtException 事件，来处理未捕获的异常：

```
process.on("uncaughtException", (err, origin) => {
console.log(err.message);
});
```

```
const a = 1 / b;
console.log("abc"); // 不会执行
```

上面的代码，控制台的输出是：`b is not defined`。捕获了错误信息，并且进程以`0`退出。开发者可以在 uncaughtException 事件中，清除一些已经分配的资源（文件描述符、句柄等），不推荐在其中重启进程。

unhandledRejection 事件

如果一个 Promise 回调的异常没有被`.catch()`捕获，那么就会触发 process 的 unhandledRejection 事件：

```
process.on("unhandledRejection", (err, promise) => {
console.log(err.message);
});

Promise.reject(new Error("错误信息")); // 未被catch捕获的异常，交由unhandledRejection事件处理
```

warning 事件

告警不是 Node.js 和 Javascript 错误处理流程的正式组成部分。 一旦探测到可能导致应用性能问题，缺陷或安全隐患相关的代码实践，Node.js 就可发出告警。

比如前一段代码中，如果出现未被捕获的 promise 回调的异常，那么就会触发 warning 事件。

如何处理进程退出?

process.exit() vs process.exitCode

一个 nodejs 进程，可以通过 process.exit() 来指定退出代码，直接退出。**不推荐直接使用 process.exit()**，这会导致事件循环中的任务直接不被处理，以及可能导致数据的截断和丢失（例如 stdout 的写入）。

```
setTimeout(() => {
console.log("我不会执行");
});

process.exit(0);
```

正确安全的处理是，设置 process.exitCode，并允许进程自然退出。

```
setTimeout(() => {
console.log("我不会执行");
});

process.exitCode = 1;
```

beforeExit 事件

用于处理进程退出的事件有：beforeExit 事件 和 exit 事件。

当 Node.js 清空其事件循环并且没有其他工作要安排时，会触发 beforeExit 事件。例如在退出前需要一些异步操作，那么可以写在 beforeExit 事件中：

```
let hasSend = false;
process.on("beforeExit", () => {
if (hasSend) return; // 避免死循环

setTimeout(() => {
console.log("mock send data to serve");
hasSend = true;
}, 500);
});

console.log(".......");
// 输出：
// .......
// mock send data to serve
```

注意：在 beforeExit 事件中如果是异步任务，那么又会被添加到任务队列。此时，任务队列完成所有任务后，又回触发 beforeExit 事件。因此，不处理的话，**可能出现死循环的情况**。如果是显式调用 exit()，那么不会触发此事件。

exit 事件

在 exit 事件中，只能执行同步操作。在调用 'exit' 事件监听器之后，Node.js 进程将立即退出，从而导致在事件循环中仍排队的任何其他工作被放弃。

process 的标准流对象

process 提供了 3 个标准流。需要注意的是，它们有些在某些时候是同步阻塞的（请见文档）。

- process.stderr：WriteStream 类型，`console.error`的底层实现，默认对应屏幕
- process.stdout：WriteStream 类型，`console.log`的底层实现，默认对应屏幕
- process.stdin：ReadStream 类型，默认对应键盘输入

下面是基于“生产者-消费者模型”的读取控制台输入并且及时输出的代码：

```
process.stdin.setEncoding("utf8"); process.stdin.on("readable", () => { let chunk; while ((chunk = process.stdin.read()) !== null) { process.stdout.write(`>>> ${chunk}`); } }); process.stdin.on("end", () => { process.stdout.write("结束"); });
```

关于事件的含义，还是请看stream 的文档。

深入理解 process.nextTick

我第一次看到 process.nextTick 的时候是比较懵的，看文档可以知道，它的用途是：把回调函数作为微任务，放入事件循环的任务队列中。但这么做的意义是什么呢?

因为 nodejs 并不适合计算密集型的应用，一个进程就一个线程，在当下时间点上，就一个事件在执行。那么，如果我们的事件占用了很多 cpu 时间，那么之后的事件就要等待非常久。所以，**nodejs 的一个编程原则是尽量缩短每一个事件的执行事件**。process.nextTick 的作用就在这，**将一个大的任务分解成多个小的任务**。示例代码如下：

```
// 被拆分成2个函数执行
function BigThing() {
doPartThing();

process.nextTick(() => finishThing());
}
```

在事件循环中，何时执行 nextTick 注册的任务呢？请看下面的代码：

```
setTimeout(function() {
console.log("第一个1秒");
process.nextTick(function() {
console.log("第一个1秒：nextTick");
});
}, 1000);

setTimeout(function() {
console.log("第2个1秒");
}, 1000);

console.log("我要输出1");

process.nextTick(function() {
console.log("nextTick");
});

console.log("我要输出2");
```

输出的结果如下，nextTick 是早于 setTimeout：

```
我要输出1
我要输出2
nextTick
第一个1秒
第一个1秒：nextTick
第2个1秒
```

在浏览器端，nextTick 会退化成 `setTimeout(callback, 0)`。但在 nodejs 中请使用 nextTick 而不是 setTimeout，前者效率更高，并且严格来说，两者创建的事件在任务队列中顺序并不一样（请看前面的代码）。

子进程：child_process模块

掌握 nodejs 的 child_process 模块能够极大提高 nodejs 的开发能力，例如主从进程来优化 CPU 计算的问题，多进程开发等等。本文从以下几个方面介绍 child_process 模块的使用：

- 创建子进程
- 父子进程通信
- 独立子进程
- 进程管道

创建子进程

nodejs 的 child_process 模块创建子进程的方法：spawn, fork, exec, execFile。它们的关系如下：

- fork, exec, execFile 都是通过 spawn 来实现的。
- exec 默认会创建 shell。execFile 默认不会创建 shell，意味着不能使用 I/O 重定向、file glob，但效率更高。
- spawn、exec、execFile 都有同步版本，可能会造成进程阻塞。

`child_process.spawn()`的使用：

```
const { spawn } = require("child_process");
// 返回ChildProcess对象，默认情况下其上的stdio不为null
const ls = spawn("ls", ["-lh"]);

ls.stdout.on("data", data => {
console.log(`stdout: ${data}`);
});

ls.stderr.on("data", data => {
console.error(`stderr: ${data}`);
});

ls.on("close", code => {
console.log(`子进程退出，退出码 ${code}`);
});
```

`child_process.exec()`的使用：

```
const { exec } = require("child_process");
// 通过回调函数来操作stdio
exec("ls -lh", (err, stdout, stderr) => {
```

```
if (err) {
console.error(`执行的错误: ${err}`);
return;
}
console.log(`stdout: ${stdout}`);
console.error(`stderr: ${stderr}`);
});
```

父子进程通信

`fork()`返回的 ChildProcess 对象，监听其上的 message 事件，来接受子进程消息；调用 send 方法，来实现 IPC。

parent.js 代码如下：

```
const { fork } = require("child_process");
const cp = fork("./sub.js");
cp.on("message", msg => {
    console.log("父进程收到消息: ", msg);
});
cp.send("我是父进程");
```

sub.js 代码如下：

```
process.on("message", m => {
console.log("子进程收到消息：", m);
});

process.send("我是子进程");
```

运行后结果：

```
父进程收到消息：  我是子进程
子进程收到消息：  我是父进程复制代码
```

独立子进程

在正常情况下，父进程一定会等待子进程退出后，才退出。如果想让父进程先退出，不受到子进程的影响，那么应该：

- 调用 ChildProcess 对象上的`unref()`
- `options.detached` 设置为 true
- 子进程的 stdio 不能是连接到父进程

main.js 代码如下：

```
const { spawn } = require("child_process");
const subprocess = spawn(process.argv0, ["sub.js"], {
detached: true,
```

```
stdio: "ignore"
});

subprocess.unref();
```

sub.js 代码如下：

```
setInterval(() => {}, 1000);
```

进程管道

options.stdio 选项用于配置在父进程和子进程之间建立的管道。 默认情况下，子进程的 stdin、stdout 和 stderr 会被重定向到 ChildProcess 对象上相应的 subprocess.stdin、subprocess.stdout 和 subprocess.stderr 流。 这意味着可以通过监听其上的 `data`事件，在父进程中获取子进程的 I/O 。

可以用来实现“重定向”：

```
const fs = require("fs");
const child_process = require("child_process");

const subprocess = child_process.spawn("ls", {
stdio: [
0, // 使用父进程的 stdin 用于子进程。
"pipe", // 把子进程的 stdout 通过管道传到父进程 。
fs.openSync("err.out", "w") // 把子进程的 stderr 定向到一个文件。
]
});
```

也可以用来实现"管道运算符"：

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

const ps = spawn("ps", ["ax"]);
const grep = spawn("grep", ["ssh"]);

ps.stdout.on("data", data => {
    grep.stdin.write(data);
});

ps.stderr.on("data", err => {
    console.error(`ps stderr: ${err}`);
});

```

```
ps.on("close", code => {
    if (code !== 0) {
        console.log(`ps 进程退出，退出码 ${code}`);
    }
    grep.stdin.end();
});

grep.stdout.on("data", data => {
    console.log(data.toString());
});

grep.stderr.on("data", data => {
    console.error(`grep stderr: ${data}`);
});

grep.on("close", code => {
    if (code !== 0) {
        console.log(`grep 进程退出，退出码 ${code}`);
    }
});
```