

前端模块化

| | | | |
|--------|--------------|---------|-----|
| 知识贡献者： | 研发中心 移动研发部王健 | 推荐人： | 马成平 |
| 推荐读者： | | 建议阅读时间： | 分钟 |

前言

在平常开发时，大家经常会写`require('xxxxxx')` 或者`import xxx form 'xxx'` 他么你有什么不同呢？ 还有用过 `webpack` 或者 `rollup.js` 等工具 的同学们也会对 `amd`、`cjs`、`es`、`umd` 等词语比较耳熟，他们都代表上面含义呢？；打开`npm`下载的包，你可能会发现有`dist`，`es`和`lib`目录，为啥这么多目录呢？

很多同学都已经看出来了，没错这些都和前端的模块化规范有着关系，那么今天我们就一起来了解和学习前端模块化方面的知识~

前端模块化的发展

在JavaScript发展初期，为了实现简单的页面交互逻辑，js代码寥寥数语即可，代码的依赖抽离放在一个文件内完成也没有什么问题。而如今各种电子设备的CPU、浏览器性能都有了巨大提升，很多的页面逻辑处理都迁移到了客户端，这就造成了前端项目越来越复杂。从而前端代码间相互依赖、引用等问题越来越不可忽视，这时就需要制定一套规范制度来约束代码间的相互关系。遗憾的是当时的js并没有像 Ruby 的 `require`、Python 的 `import` 这样的东西。可是连当时的 CSS 都有 `@import` 来引用其他 css 文件...

所以js开发者急需模拟出类似的功能，来隔离、组织复杂的Javascript代码，即**前端模块化**。下面我来介绍一下**前端模块化**的发展历程：

- **立即执行函数IIFE**（Immediately Invoked Function Expression）
- **Common.js**
- **AMD/CMD**
- **ES6 Module**

立即执行函数IIFE

IIFE 即 立即执行函数，最开始，我们对于模块区分概念，可能是从独立的文件区分开始的，在一个简易的项目中，编程的习惯是通过一个 HTML 文件加上若干个 JavaScript 文件来区分不同的模块，就像这样：

HTML

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>demo</title>
  </head>

  <script src="main.js"></script>
  <script src="header.js"></script>

  <body>XXXX</body>

</html>
```

在不同的JS中我们定义不同的变量，来做一些事情，比如这样：

JS

```
//header.js
var header = ''
// window.header => ''
// balabalabala

//main.js
var main_message = '';
// window.main_message => ''
// balabalabala
```

但是这样会导致严重的问题，因为这些变量都挂在window对象下面，如果有其他人来维护项目很可能把其中的变量覆盖。所以仅仅通过不同的文件，我们是无法做到将这些变量**隔离**，因为它们都被绑在了同一个 window 变量上。那我们怎么去解决呢？

我们都知道当在一个JS文件中是，函数体内的变量在函数体外是访问不到的，像这样：

JS

```
function bar() {  
  var foo = 'foo';  
  console.log(foo); // foo  
}  
bar();  
  
foo // undefined
```

所以隔离的方案找到了，只要把整个JS内容放到函数里面，这样就可以把各自的变量**隔离**开，但是现在又有了新的问题：

1. bar 函数需要执行，总不能每个文件的末尾都要执行一下吧？
2. 最关键的问题，上述代码的函数 bar 不是也和之前的 header 变量一样嘛，挂载在 window 上，并未做到隔离

面对上面的问题，我们思考片刻后给出了答案：能否让**匿名函数自动执行**，像这样：

IIFE

```
function(){  
  //  
}()
```

天才的想法！！！可是报错了~ 为什么呢？

原因是：function关键字，既可以用作语句，也可以用作表达式。如果不加以区分或限定，代码在解析时，就会“不知所措”，分不清是当作语句，还是当作表达式来处理。为了避免这种歧义的产生，JS引擎规定，如果function关键字出现在一行代码的最前部，一律解析为语句。

那么聪明的同学立刻就会想到，只要 function 不放在行首就行了：

IIFE

```
(function(){  
  //  
})();
```

是的，完美解决了隔离的问题！不会污染全局对象。并且匿名函数执行完后很快就会被释放

当然如果JS间有互相引用的部分，就把需要暴露的变量手动挂载的 window 对象下，像这样：window.foo = 'foo'

IIFE

```
// IIFE
(function(
  //
){})();

//~
~function(){
  //
}();

//+
+function(){
  //
}();
//
```

虽然今天已经很少有人会在开发中使用IIFE，但是由于其兼容性好，IIFE一直存在于许多项目最终编译打包生成的JS文件中

Common.js

在 2009 年的一个冬天，一名来自 Mozilla 团队的工程师 Kevin Dangoor 开始捣鼓了一个叫 ServerJS 的项目，这个项目在 2009 年的 8 月份更名为今天我们熟悉的 CommonJS 以显示 API 更广泛的适用性。那时他可能并没有料到，这一规则的制定会让整个前端发生翻天覆地的变化。

CommonJS 是一个旨在Web 浏览器之外，为 JavaScript 建立模块生态系统的约定的项目。其创建的主要原因是缺乏普遍接受的 JavaScript 脚本模块单元形式，而这一形式可以让 JavaScript 在不同于传统网络浏览器提供的环境中重复使用，例如，运行 JavaScript 脚本的 Web 服务器或本机桌面应用程序。

通过上面这些描述，相信你已经知道 CommonJS 是诞生于怎样的背景，但是这里所说的CommonJS 是一套**通用的规范**，与之对应的有非常多**不同的实现**：

下面我们来看看

Implementations [\[edit \]](#)

- [Akshell](#)^[7]
- [Common Node](#)^[8]
- [CommonJS Compiler](#) - a command-line tool that makes Common JS modules suitable for in-browser use^[9]
- [CommonJS for PHP](#) - a light-weight CommonJS implementation for PHP 5.3+^[10]
- [CouchDB](#)^[11]
- [Flusspferd](#)^[12]
- [GPSEE](#)^[13]
- [Jetpack](#)
- [Joyent Smart Platform](#)^[14]
- [JSBuild](#)^[15]
- [MongoDB](#)^[16]
- [Narwhal](#) (JavaScript platform)^[17]
- [Node.js](#)^[18]
- [Persevere](#)^[19]
- [PINF JavaScript Loader](#)^[20]
- [RingoJS](#)^[21]
- [SilkJS](#)^[22]
- [SproutCore](#)^[23]
- [TeaJS](#)^[24]
- [Wakanda](#)^[25]
- [XULJet](#)^[26]

但这其中最受关注的是其中Node.js 的实现部分，我们先看一下代码

common.js

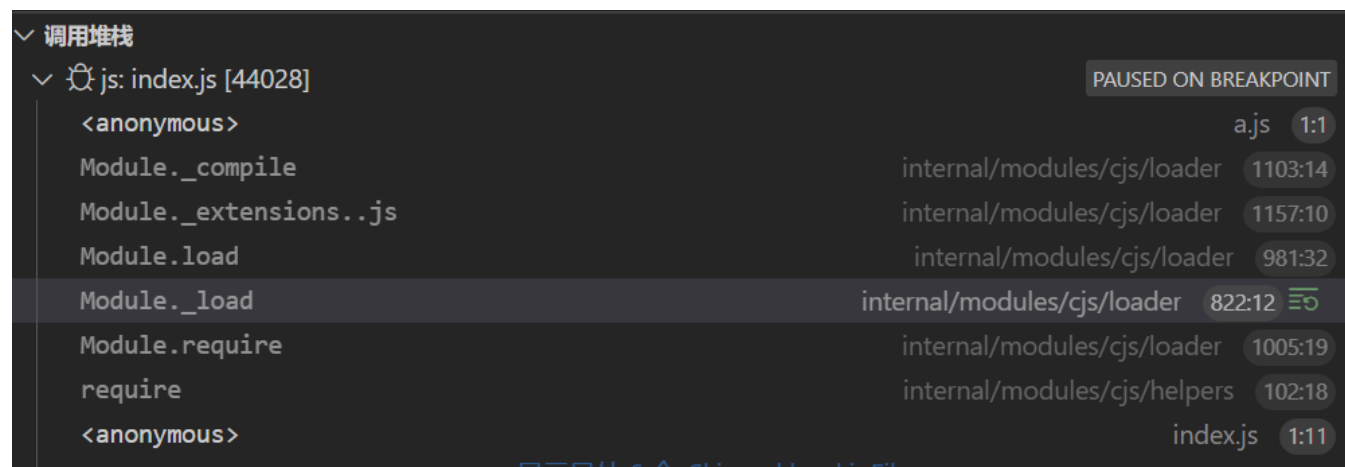
```
// index.js
const moduleA = require('./a')
console.log('a', moduleA.a)

// a.js
module.exports = {a: 'a'};
//
// exports.a = 'a'
```

`module.exports`（或者 `exports`）导出变量；`require` 导入变量；这里要注意一下 `module.exports` 和 `exports` 导出的用法

相信大家已经很了解 `commonjs` 的用法；下面我们根据 VScode 提供的调试功能（调用栈），来简单的看一下 `commonjs` 的部分源码：

当我们再 `a.js` 打了断点之后，发现调用栈有 6 层，如下图：



所以下面我们从栈的最底部开始看

require

common.js

```
//...
} else {
  require = function require(path) {
    return mod.require(path);
  };
}

// mod
//
```

这里 `mod` 就可以认为是调用的模块（`index.js` 模块的各种信息），`path` 就是被调用的文件路径（`./a`），然后就调用了 `Module.require` 方法（`mod` 的构造函数就是 `Module`）；

Module.require

common.js

```
Module.prototype.require = function(id) {
  validateString(id, 'id');
  if (id === '') {
    throw new ERR_INVALID_ARG_VALUE('id', id, 'must be a non-empty string');
  }

  requireDepth++;

  try {
    return Module._load(id, this, /* isMain */ false);
  } finally {
    requireDepth--;
  }
};
```

Module.require 做了2件事情，第一是判断 id 也就是 require 的路径是否合法；第二是执行Module._load 方法；

Module._load

common.js

```
// Check the cache for the requested file.
// 1. If a module already exists in the cache: return its exports object.
// 2. If the module is native: call
//    `NativeModule.prototype.compileForPublicLoader()` and return the exports.
// 3. Otherwise, create a new module for the file and save it to the cache.
//    Then have it load the file contents before returning its exports
//    object.
Module._load = function(request, parent, isMain) {
  let relResolveCacheIdentifier;
  if (parent) {
    debug('Module._load REQUEST %s parent: %s', request, parent.id);
    // Fast path for (lazy loaded) modules in the same directory. The indirect
    // caching is required to allow cache invalidation without changing the old
    // cache key names.
    relResolveCacheIdentifier = `${parent.path}\\x00${request}`;
    const filename = relativeResolveCache[relResolveCacheIdentifier];
    if (filename !== undefined) {
      const cachedModule = Module._cache[filename];
      if (cachedModule !== undefined) {
        updateChildren(parent, cachedModule, true);
        if (!cachedModule.loaded)
          return getExportsForCircularRequire(cachedModule);
        return cachedModule.exports;
      }
      delete relativeResolveCache[relResolveCacheIdentifier];
    }
  }

  const filename = Module._resolveFilename(request, parent, isMain);
  if (StringPrototypeStartsWith(filename, 'node:')) {
    // Slice 'node:' prefix
    const id = StringPrototypeSlice(filename, 5);

    const module = loadNativeModule(id, request);
    if (!module?.canBeRequiredByUsers) {
      throw new ERR_UNKNOWN_BUILTIN_MODULE(filename);
    }

    return module.exports;
  }

  const cachedModule = Module._cache[filename];
  if (cachedModule !== undefined) {
    updateChildren(parent, cachedModule, true);
```

```

    if (!cachedModule.loaded) {
      const parseCachedModule = cjsParseCache.get(cachedModule);
      if (!parseCachedModule || parseCachedModule.loaded)
        return getExportsForCircularRequire(cachedModule);
      parseCachedModule.loaded = true;
    } else {
      return cachedModule.exports;
    }
  }

  const mod = loadNativeModule(filename, request);
  if (mod?.canBeRequiredByUsers) return mod.exports;

  // Don't call updateChildren(), Module constructor already does.
  const module = cachedModule || new Module(filename, parent);

  if (isMain) {
    process.mainModule = module;
    module.id = '.';
  }

  Module._cache[filename] = module;
  if (parent !== undefined) {
    relativeResolveCache[relResolveCacheIdentifier] = filename;
  }

  let threw = true;
  try {
    module.load(filename);
    threw = false;
  } finally {
    if (threw) {
      delete Module._cache[filename];
      if (parent !== undefined) {
        delete relativeResolveCache[relResolveCacheIdentifier];
        const children = parent?.children;
        if (Array.isArray(children)) {
          const index = Array.prototype.indexOf(children, module);
          if (index !== -1) {
            Array.prototype.splice(children, index, 1);
          }
        }
      }
    }
  }
  } else if (module.exports &&
    !isProxy(module.exports) &&
    Object.getPrototypeOf(module.exports) ===
      CircularRequire.prototypeWarningProxy) {
    Object.setPrototypeOf(module.exports, Object.prototype);
  }
}

return module.exports;
};

```

`Module._load` 方法比较长，好在在注释的地方，作者已经署名了 `_load` 方法干了什么：

1. 如果缓存存在该模块，直接从缓存中返回
2. 如果该模块为原生模块，调用 `NativeModule.prototype.compileForPublicLoader` 来加载模块，若加载成功，返回此原生模块
3. 否则，根据路径创建一个新的模块，同时存入缓存中，返回新创建模块

我们分析一下：

- 42 行根据 29 行得到的绝对路径地址 `filename`，去判断该模块是否已经缓存，如果缓存了，直接返回
- 55 行加载原生模块
- 如果碰到新的模块，66 行存入缓存中（键值是 `filename`），73 行调用了 `load` 方法传入参数 `filename`，来加载新的模块
- 代码最后将 `module.exports` 返回

Module.load

common.js

```
// Given a file name, pass it to the proper extension handler.
Module.prototype.load = function(filename) {
  debug('load %j for module %j', filename, this.id);

  assert(!this.loaded);
  this.filename = filename;
  this.paths = Module._nodeModulePaths(path.dirname(filename));

  const extension = findLongestRegisteredExtension(filename);
  // allow .mjs to be overridden
  if (StringPrototypeEndsWith(filename, '.mjs') && !Module._extensions['.mjs'])
    throw new ERR_REQUIRE_ESM(filename, true);

  Module._extensions[extension](this, filename);
  this.loaded = true;

  const esmLoader = asyncESM.esmLoader;
  // Create module entry at load time to snapshot exports correctly
  const exports = this.exports;
  // Preemptively cache
  if ((module?.module === undefined ||
    module.module.getStatus() < kEvaluated) &&
    !esmLoader.cjsCache.has(this))
    esmLoader.cjsCache.set(this, exports);
};
```

注释里面已经写明白了：**给定文件名，将其传递给适当的扩展处理程序**；可以理解为**根据不同的文件拓展名调用对应的处理函数**，而处理的函数就是**Module._extensions[extension](this, filename)**，传递了 **this** 和 **filename**；

我们可以找到第9行**findLongestRegisteredExtension** 定义的地方：

common.js

```
// Find the longest (possibly multi-dot) extension registered in
// Module._extensions
function findLongestRegisteredExtension(filename) {
  .....
```

作用就是找出 filename 文件的拓展名；

11行对 .mjs 后缀的文件也做了处理，当 filename 文件后缀是 mjs 并且 Module._extensions 没有对应 mjs 处理函数时，就会抛出一个错误；

此外，load 函数还重新赋值了 **this.filename**，**this.paths**，**this.loaded**；

Module._extensions

上文说了处理函数**Module._extensions[extension](this, filename)** 根据不同的**extension** 调用不同的处理函数，**Module._extensions[extension]** 这种形式，大家很容易想到**Module._extensions** 应该是个对象，**extension** 就是键值；

果不其然，定位到代码后，会看到：

common.js

```
// Native extension for .js
Module._extensions['.js'] = function(module, filename) {
  // If already analyzed the source, then it will be cached.
  const cached = cjsParseCache.get(module);
  let content;
  if (cached?.source) {
    content = cached.source;
    cached.source = undefined;
  } else {
    content = fs.readFileSync(filename, 'utf8');
  }
```

```

}
if (StringPrototypeEndsWith(filename, '.js')) {
  const pkg = readPackageScope(filename);
  // Function require shouldn't be used in ES modules.
  if (pkg?.data?.type === 'module') {
    const parent = moduleParentCache.get(module);
    const parentPath = parent?.filename;
    const packageJsonPath = path.resolve(pkg.path, 'package.json');
    const usesEsm = hasEsmSyntax(content);
    const err = new ERR_REQUIRE_ESM(filename, usesEsm, parentPath,
      packageJsonPath);

    // Attempt to reconstruct the parent require frame.
    if (Module._cache[parentPath]) {
      let parentSource;
      try {
        parentSource = fs.readFileSync(parentPath, 'utf8');
      } catch {
        // Continue regardless of error.
      }
      if (parentSource) {
        const errLine = StringPrototypeSplit(
          StringPrototypeSlice(err.stack, StringPrototypeIndexOf(
            err.stack, '    at ')), '\n', 1)[0];
        const { 1: line, 2: col } =
          RegExpPrototypeExec(/(\d+):(\d+)\)/, errLine) || [];
        if (line && col) {
          const srcLine = StringPrototypeSplit(parentSource, '\n')[line - 1];
          const frame = `${parentPath}:${line}\n${srcLine}\n${
            StringPrototypeRepeat(' ', col - 1)}^`;
          setArrowMessage(err, frame);
        }
      }
    }
    throw err;
  }
}
module._compile(content, filename);
}

// Native extension for .json
Module._extensions['.json'] = function(module, filename) {
  const content = fs.readFileSync(filename, 'utf8');

  if (policy?.manifest) {
    const moduleURL = pathToFileURL(filename);
    policy.manifest.assertIntegrity(moduleURL, content);
  }

  try {
    module.exports = JSONParse(stripBOM(content));
  } catch (err) {
    err.message = filename + ': ' + err.message;
    throw err;
  }
}

// Native extension for .node
Module._extensions['.node'] = function(module, filename) {
  if (policy?.manifest) {
    const content = fs.readFileSync(filename);
    const moduleURL = pathToFileURL(filename);
    policy.manifest.assertIntegrity(moduleURL, content);
  }
  // Be aware this doesn't use `content`
  return process.dlopen(module, path.toNamespacedPath(filename));
}

```

目前源码中只支持处理这三种后缀的文件（后面有需要增加处理函数的话只要添加对应的后缀名就ok了）：

- node文件 最后调用了 `process.dlopen(module, path.toNamespacedPath(filename))`，后面不做深入研究

- json文件 调用了JSONParse 解析文件内容，得到对象然后复赋值给module.exports
- js文件 最后调用了module._compile(content, filename)，当然前面也做了一些 es module 的判断

module._compile

common.js

```
// Run the file contents in the correct scope or sandbox. Expose
// the correct helper variables (require, module, exports) to
// the file.
// Returns exception, if any.
Module.prototype._compile = function(content, filename) {
  let moduleURL;
  let redirects;
  if (policy?.manifest) {
    moduleURL = pathToFileURL(filename);
    redirects = policy.manifest.getDependencyMapper(moduleURL);
    policy.manifest.assertIntegrity(moduleURL, content);
  }

  maybeCacheSourceMap(filename, content, this);
  const compiledWrapper = wrapSafe(filename, content, this);

  let inspectorWrapper = null;
  if (getOptionValue('--inspect-brk') && process._eval == null) {
    if (!resolvedArgv) {
      // We enter the repl if we're not given a filename argument.
      if (process.argv[1]) {
        try {
          resolvedArgv = Module._resolveFilename(process.argv[1], null, false);
        } catch {
          // We only expect this codepath to be reached in the case of a
          // preloaded module (it will fail earlier with the main entry)
          assert(Array.isArray(getOptionValue('--require')));
        }
      } else {
        resolvedArgv = 'repl';
      }
    }

    // Set breakpoint on module start
    if (resolvedArgv && !hasPausedEntry && filename === resolvedArgv) {
      hasPausedEntry = true;
      inspectorWrapper = internalBinding('inspector').callAndPauseOnStart;
    }
  }
  const dirname = path.dirname(filename);
  const require = makeRequireFunction(this, redirects);
  let result;
  const exports = this.exports;
  const thisValue = exports;
  const module = this;
  if (requireDepth === 0) statCache = new SafeMap();
  if (inspectorWrapper) {
    result = inspectorWrapper(compiledWrapper, thisValue, exports,
                              require, module, filename, dirname);
  } else {
    result = ReflectApply(compiledWrapper, thisValue,
                          [exports, require, module, filename, dirname]);
  }
  hasLoadedAnyUserCJSModule = true;
  if (requireDepth === 0) statCache = null;
  return result;
};
```

注释中这样写道：**在正确的范围或沙箱中运行文件内容。将正确的辅助变量（require、module、exports）暴露给文件。**返回异常（如果有），其实就是把读取的js文件在沙箱中结合辅助函数运行；

在看源码之前，我们思考一个问题，fs.readFileSync 得到的 js 文件的字符串（文件是**同步读取**的哦，所以模块的加载也是同步运行的），然后是怎么执行的？

1. 第一个很容易想到的是: `eval` 方法, 负面评论很多的方法, 已经不建议使用了
2. 第二个 `new Function()` 方法, 下面这样使用:

common.js

```
// argN
new Function(arg1, arg2, ...argN, functionBody);
```

但是, 这个方法并不完美, 使用 `new Function` 创建的函数不会创建当前环境的闭包, 它们总是被创建在全局环境。也就是说, 使用 `new Function` 定义的函数在运行时能访问全局变量和自己的局部变量, 而不能访问它们被 `Function` 构造函数创建时所在的作用域的变量

3. 第三个是 `node vm` 模块, 即沙箱模式。它可以这样使用:

common.js

```
const vm = require("vm");
global.a = 100;
// []
vm.runInThisContext("console.log(a)"); // 100
// []
vm.runInNewContext("console.log(a)"); // a is not defined
```

属于完美的解决方案。当然他还有很多api, 比如 `vm.compileFunction`

现在, 我们看一下源码。第15行和51行, 调用了 `wrapSafe`:

common.js

```
function wrapSafe(filename, content, cjsModuleInstance) {
  if (patched) {
    const wrapper = Module.wrap(content);
    return vm.runInThisContext(wrapper, {
      filename,
      lineOffset: 0,
      displayErrors: true,
      importModuleDynamically: async (specifier, _, importAssertions) => {
        const loader = asyncESM.esmLoader;
        return loader.import(specifier, normalizeReferrerURL(filename),
          importAssertions);
      },
    });
  }
  try {
    return vm.compileFunction(content, [
      'exports',
      'require',
      'module',
      '__filename',
      '__dirname',
    ], {
      filename,
      importModuleDynamically(specifier, _, importAssertions) {
        const loader = asyncESM.esmLoader;
        return loader.import(specifier, normalizeReferrerURL(filename),
          importAssertions);
      },
    });
  } catch (err) {
    if (process.mainModule === cjsModuleInstance)
      enrichCJSError(err, content);
    throw err;
  }
}
```

这里使用了`vm.compileFunction` 方法对代码字符串 进行了处理, 处理后的结果如下图代码所示: 代码被包裹了一层, 变成了上节我们所说的 IIFE, 参数即上文所说的辅助变量; 变成了可执行的 `js` 函数, 将值赋值给 `module.exports`, 最后再看回`Module._load`方法内, 把`module.exports` 返回;

common.js

```
// Node.js
(function(exports, require, module, __filename, __dirname) {
// exports    module
// require
// __filename  __dirname

// ~
});
```

module.exports 与 exports

这里可能会有人好奇，`module.exports={a: XXX}` 与 `exports.a = XXX` 有什么不一样，能不能写成`exports={a: XXX}`，我们来看一段伪代码：

common.js

```
function require(/* ... */) {
  const module = {
    exports: {}
  };

  ((exports, require, module, __filename, __dirname) => {
    //
    function a() {}
    function b() {}
    function c() {}
    //exports = a; //
    //exports.b = b; //
    //module.exports = b; //

  })(module, module.exports);

  return module.exports;
}
```

上述标记1、2、3，哪些是正确的，哪些是错误的呢？

我们稍加测试之后就会发现，标记①是错误的，标记②和标记③都是正确的，这是应为`exports` 是 `module.exports` 的引用，指针指向同一地址，但是当你给`exports` 赋值时，`exports` 的性质就变了，此时`exports` 不再是 `module.exports` 的引用，它已经与`module.exports` 没有任何关系了，而`exports.a` 这种写法却完全没有问题：

循环引用

我们先看一个小例子：

common.js

```
// index.js
console.log('index-');
const a = require('./a');
console.log('index'+a);

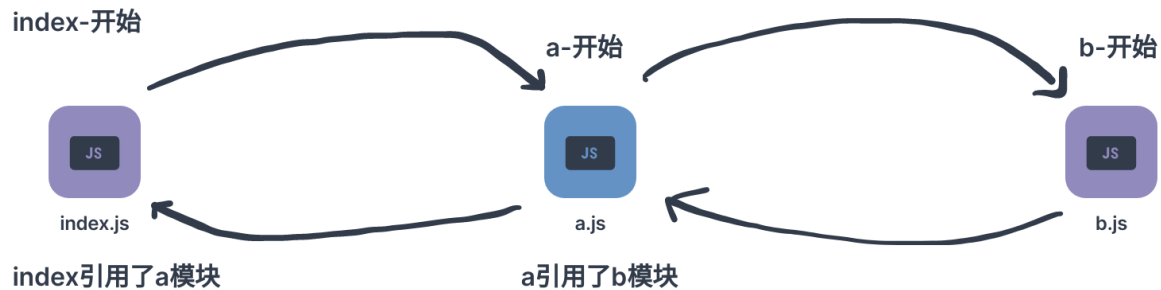
// a.js
console.log('a-');
const b = require('./b');
console.log('a'+b);
module.exports.text = 'a';

// b.js
console.log('b-');
module.exports.text = 'b';
```

此时会打印出：

common.js

```
index-  
a-  
b-  
ab  
indexa
```



结合我们之前讲的源码，js 文件遇到`require('依赖文件')`就会进行加载和缓存，然后依赖文件执行完毕就会 `return module.exports`并回到之前的位置继续往下执行，所以这显然是 **深度优先遍历**，即 **父→子→父** 这种形式：

那么思考一个问题，如果代码中出现了模块间的循环引用呢？改成下面这样试一下：

common.js

```
// index.js  
console.log('index-');  
const a = require('./a');  
console.log('index', a);  
  
// a b  
// a.js  
console.log('a-');  
const b = require('./b');  
console.log('a', b);  
module.exports = 'a';  
  
// b.js  
console.log('b-');  
const a = require('./a');  
console.log('b', a);  
module.exports = 'b';
```

此时会打印出：

common.js

```
index-  
a-  
b-  
b {}  
a b  
index a
```

a.js 引用了 b.js，b.js 又引用了 b.js，按道理他们会无限循环的引用，但其实并没有；按照上面的源码介绍，我们走一下流程：

- 执行 index.js 打印 “index-开始”，然后走到 require('a.js') 处；
- 进入 a.js 文件中，此时 a.js 属于新的模块，所以 a.js 会被加载并会被缓存，我们看一下变量情况：

```

变量
  Local
    __dirname: 'c:\Users\wangjian8818\Desktop\tcptest'
    __filename: 'c:\Users\wangjian8818\Desktop\tcptest\la.js'
    b: undefined
  > exports: {}
  > module: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\la.js', path: 'c:\Users\wangjian8818\Desktop\tcptest', exports: {}, filename: 'c:\Users\wangjian8818\Desktop\tcptest\la.js')
  > require: f require(path) {
    arguments: ④ f ()
    > cache: {d:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js, c:\Users\wangjian8818\Desktop\tcptest\index.js: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\index.js', path: 'c:\Users\wangjian8818\Desktop\tcptest\index.js', exports: {}), c:\Users\wangjian8818\Desktop\tcptest\la.js: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\la.js', path: 'c:\Users\wangjian8818\Desktop\tcptest\la.js', exports: {})}
    > children: (0) []
    > exports: {}
    filename: 'c:\Users\wangjian8818\Desktop\tcptest\la.js'
    id: 'c:\Users\wangjian8818\Desktop\tcptest\la.js'
    isPreloading: ④ f get() { return isPreloading; }
    loaded: false
    parent: ④ f getModuleParent() {\r\n return moduleParentCache.get(this);\r\n}
    path: 'c:\Users\wangjian8818\Desktop\tcptest'
    > paths: (5) ['c:\Users\wangjian8818\Desktop\tcptest\node_modules', 'c:\Users\wangjian8818\Desktop\node_modules', 'c:\Users\wangjian8818\node_modules', 'c:\Users\wangjian8818\node_modules', 'c:\Users\wangjian8818\node_modules']
    > [[Prototype]]: Object
    > c:\Users\wangjian8818\Desktop\tcptest\index.js: Module (id: '.', path: 'c:\Users\wangjian8818\Desktop\tcptest', exports: {}), filename: 'c:\Users\wangjian8818\Desktop\tcptest\index.js'
    > d:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js: Module (id: 'd:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js', path: 'd:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js', exports: {}), filename: 'd:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js'
    caller: ④ f ()
    > extensions: {js: f, json: f, node: f}
  
```

- 此时 a.js 还没有加载到 module.exports，所以缓存cache的 a.js 的 exports 是空对象{}；打印 “a-开始”，然后走到 require('b.js') 处；
- 进入 b.js 文件中，此时 b.js 属于新的模块，所以 b.js 会被加载并会被缓存，我们再看一下变量情况：

```

变量
  Local
    __dirname: 'c:\Users\wangjian8818\Desktop\tcptest'
    __filename: 'c:\Users\wangjian8818\Desktop\tcptest\b.js'
    a: undefined
  > exports: {}
  > module: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\b.js', path: 'c:\Users\wangjian8818\Desktop\tcptest', exports: {}, filename: 'c:\Users\wangjian8818\Desktop\tcptest\b.js')
  > require: f require(path) {
    arguments: ④ f ()
    > cache: {d:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js, c:\Users\wangjian8818\Desktop\tcptest\index.js: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\index.js', path: 'c:\Users\wangjian8818\Desktop\tcptest\index.js', exports: {}), c:\Users\wangjian8818\Desktop\tcptest\la.js: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\la.js', path: 'c:\Users\wangjian8818\Desktop\tcptest\la.js', exports: {}), c:\Users\wangjian8818\Desktop\tcptest\b.js: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\b.js', path: 'c:\Users\wangjian8818\Desktop\tcptest\b.js', exports: {})}
    > children: (1) [Module]
    > exports: {}
    filename: 'c:\Users\wangjian8818\Desktop\tcptest\b.js'
    id: 'c:\Users\wangjian8818\Desktop\tcptest\b.js'
    isPreloading: ④ f get() { return isPreloading; }
    loaded: false
    parent: ④ f getModuleParent() {\r\n return moduleParentCache.get(this);\r\n}
    path: 'c:\Users\wangjian8818\Desktop\tcptest'
    > paths: (5) ['c:\Users\wangjian8818\Desktop\tcptest\node_modules', 'c:\Users\wangjian8818\Desktop\node_modules', 'c:\Users\wangjian8818\node_modules', 'c:\Users\wangjian8818\node_modules', 'c:\Users\wangjian8818\node_modules']
    > [[Prototype]]: Object
    > c:\Users\wangjian8818\Desktop\tcptest\index.js: Module (id: '.', path: 'c:\Users\wangjian8818\Desktop\tcptest', exports: {}), filename: 'c:\Users\wangjian8818\Desktop\tcptest\index.js'
    > d:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js: Module (id: 'd:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js', path: 'd:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js', exports: {}), filename: 'd:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js'
    caller: ④ f ()
    > extensions: {js: f, json: f, node: f}
  
```

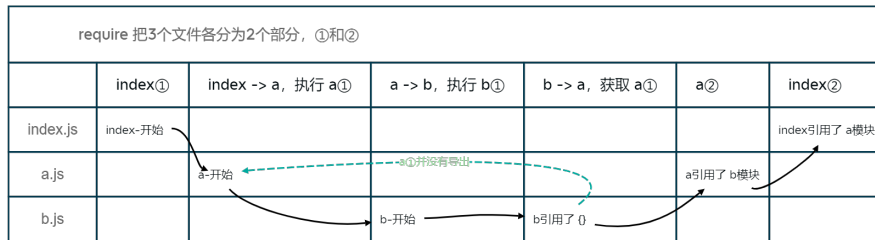
- 此时缓存的 a.js 和 b.js 的 exports 还是空对象{}；打印 “b-开始”，继续往下走；
- 此时 b.js 又 require('a.js')，但是上面我们讲了，a.js 的 exports 还是空对象，所以打印出来 “b引用了 {}”；继续往下走，b.js 导出了字符串 “b模块”，b.js 文件执行结束，所以又回到了父文件 a.js 中继续执行，此时在看一下变量：

```

变量
  Local
    __dirname: 'c:\Users\wangjian8818\Desktop\tcptest'
    __filename: 'c:\Users\wangjian8818\Desktop\tcptest\la.js'
    b: 'b模块'
  > exports: {}
  > module: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\la.js', path: 'c:\Users\wangjian8818\Desktop\tcptest', exports: {}, filename: 'c:\Users\wangjian8818\Desktop\tcptest\la.js')
  > require: f require(path) {
    arguments: ④ f ()
    > cache: {d:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js, c:\Users\wangjian8818\Desktop\tcptest\index.js: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\index.js', path: 'c:\Users\wangjian8818\Desktop\tcptest\index.js', exports: {}), c:\Users\wangjian8818\Desktop\tcptest\la.js: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\la.js', path: 'c:\Users\wangjian8818\Desktop\tcptest\la.js', exports: {}), c:\Users\wangjian8818\Desktop\tcptest\b.js: Module (id: 'c:\Users\wangjian8818\Desktop\tcptest\b.js', path: 'c:\Users\wangjian8818\Desktop\tcptest\b.js', exports: 'b模块')}
    > children: (1) [Module]
    > exports: {}
    filename: 'c:\Users\wangjian8818\Desktop\tcptest\la.js'
    id: 'c:\Users\wangjian8818\Desktop\tcptest\la.js'
    isPreloading: ④ f get() { return isPreloading; }
    loaded: false
    parent: ④ f getModuleParent() {\r\n return moduleParentCache.get(this);\r\n}
    path: 'c:\Users\wangjian8818\Desktop\tcptest'
    > paths: (5) ['c:\Users\wangjian8818\Desktop\tcptest\node_modules', 'c:\Users\wangjian8818\Desktop\node_modules', 'c:\Users\wangjian8818\node_modules', 'c:\Users\wangjian8818\node_modules', 'c:\Users\wangjian8818\node_modules']
    > [[Prototype]]: Object
    > c:\Users\wangjian8818\Desktop\tcptest\index.js: Module (id: '.', path: 'c:\Users\wangjian8818\Desktop\tcptest', exports: {}), filename: 'c:\Users\wangjian8818\Desktop\tcptest\index.js'
    > d:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js: Module (id: 'd:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js', path: 'd:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js', exports: {}), filename: 'd:\Microsoft\Microsoft VS Code\resources\app\extensions\ms-vscode.js-debug\src\bootstrap.bundle.js'
    caller: ④ f ()
    > extensions: {js: f, json: f, node: f}
  
```

此时缓存的 a.js 的 exports 还是空对象 {}, 而 b.js 的 exports 则变成了 “b模块”; 因为上一步 b.js 最后导出了字符串 “b模块”, 所以此时打印了 “a引用了 b模块”:

- 下一步, a.js 导出字符串 “a模块”, a.js 执行结束, 然后回到父文件 index.js 继续执行; 此时同理, 缓存中的 a.js 的 exports 变成字符串 “a模块”, 所以打印 “index引用了 a模块”, 执行结束; 示意图如下:



所以, 可以看出: commonjs中文件的循环引用一般不会导致报错, 这个得益于commonjs 的缓存策略, 但引用方可能会得到 空对象 或者 undefined, 大家开发的时候要多加留意。

总结

- CommonJS 由 JS **运行时**实现, 它是js语言缺失底层模块机制的情况下在上层做的弥补, 他的本质还是建立在 js 语言对象上的; 由于其这一特性, 你甚至可以这样写:

common.js

```
var a = require('./a');
var qq;
if(a.b === 'lalala'){
    qq = require('./qq');
};
// XXX
```

- CommonJS 是**同步加载并执行**模块文件的, 采用深度优先遍历, 执行顺序是 父 -> 子 -> 父;
- CommonJS **循环引用**一般不会导致 JS 错误, 但还是要多加小心;
- CommonJS 导入导出语句位置可以任意放置, 并且导入导出语句位置会影响模块代码语句执行结果

最后, 有的同学可能会问: commonjs不是用在node端的吗? 怎么我在前端代码中也可以用 require(“xxx”) 呢?

这是因为后来出现了 Browserify 这样的实现, 可以将 commonjs 的代码转换成浏览器能识别的代码。有兴趣的同学可以[了解一下](#)

AMD (Asynchronous Module Definition) /CMD (Common Module Definition) 规范

这里我们顺带了解一下AMD和CMD规范:

AMD& RequireJS

因为我们已经了解了 require() 的实现, 所以你会发现这其实是一个 “复制” 的过程, 将被 require 的内容, 赋值到一个 module 对象的属性上, 然后返回这个对象的 exports 属性。这样做会有什么问题呢? 在我们还没有完成 “复制” 的时候, 无法使用被引用的模块中的方法和属性。在服务端可能这不是一个问题(因为服务器的文件都是存放在本地, 并且是有缓存的), 但在浏览器环境下 (网络请求文件有延迟), 这会导致阻塞, 使得我们后面的步骤无法进行下去, 还可能会执行一个未定义的方法而导致出错。

相对于服务端的模块化, 浏览器环境下, 模块化的标准必须满足一个新的需求: 异步的模块管理。在这样的背景下, RequireJS 出现了, RequireJS 是基于 AMD 规范 实现的, 我们简单的了解一下它最核心的部分:

- 引入其他模块: require()
- 定义新的模块: define()

看一个AMD的例子:

AMD

```
define('math',['jquery'], function ($) {  
    //  
  
    return {  
        add: function(x,y){  
            return x + y;  
        }  
    };  
});  
  
require(['jquery','math'], function ($,math) {  
    console.log(math.add(10,100));//110  
});
```

AMD特点

- 以函数的形式返回模块的值，尤其是构造函数，可以更好的实现API 设计，Node 中通过 `module.exports` 来支持这个，但使用 “`return function () {}`” 会更清晰。这意味着，我们不必通过处理 “`module`” 来实现 “`module.exports`”，它是一个更清晰的代码表达式；
- **异步代码加载**（在AMD系统中通过`require(['xx1','xx2']functionxx1,xx2{ // })`），并且一次可以并行加载多个模块；
- 通过上面的语法说明，我们会发现一个很明显的问题，在使用 RequireJS 声明一个模块时，必须指定所有的依赖项，这些依赖项会被当做形参传到函数中，对于依赖的模块会提前执行（在 RequireJS 2.0 也可以选择延迟执行），这被称为：**依赖前置**；

CMD & SeaJS

针对 AMD 规范中可以优化的部分，[CMD 规范](#)出现了，而 SeaJS 则作为它的具体实现之一，与 AMD 十分相似：

CMD

```
// AMD  
define('amd',['header', 'main', 'footer'], function(header, main, footer) {  
    if (xxx) {  
        header.setHeader('new-title')  
    }  
    if (xxx) {  
        main.setMain('new-content')  
    }  
    if (xxx) {  
        footer.setFooter('new-footer')  
    }  
});  
  
// CMD  
define(function(require, exports, module) {  
    if (xxx) {  
        var header = require('./header')  
        header.setHeader('new-title')  
    }  
    if (xxx) {  
        var main = require('./main')  
        main.setMain('new-content')  
    }  
    if (xxx) {  
        var footer = require('./footer')  
        footer.setFooter('new-footer')  
    }  
});
```

看一个CMD的例子：

CMD

```
// myModule.js
define(function(require, exports, module) {
  var $ = require('jquery.js')
  $('div').addClass('active');
  module.exports = {
    data: 1
  };

  // exports = { // seajs
  //   data: 1
  // };
  // return {
  //   data: 1
  // };
});

//
seajs.use(['myModule.js'], function(my){
  var star= my.data;
  console.log(star); //1
});
```

CMD特点

- **依赖就近-延迟执行**这正是CMD所推崇的。只有当我们用到了某个外部模块的时候，它才会去引入。这解决了我们上一小节中遗留的问题。
- 当你看到`require(); module.exports = {};`的时候，是不是有一种似曾相识的感觉，对！没错！与 `commonjs` 怎么这么像啊！！所以它与 CommonJS 的 Node.js Modules 规范保持了很大的兼容性。

ECMAScript6 Module

突然，时间到达了2015年6月17日，ECMAScript 6发布正式版本，即ECMAScript 2015；ECMAScript6 标准增加了 JavaScript 语言层面的模块体系定义，作为浏览器和服务端通用的模块解决方案它可以取代我们之前提到的 AMD，CMD，CommonJS。适用于前后端。至此，终于有了官方版了...

关于 ES Module 相信大家每天的工作中都会用到，看一个示例：

ESM

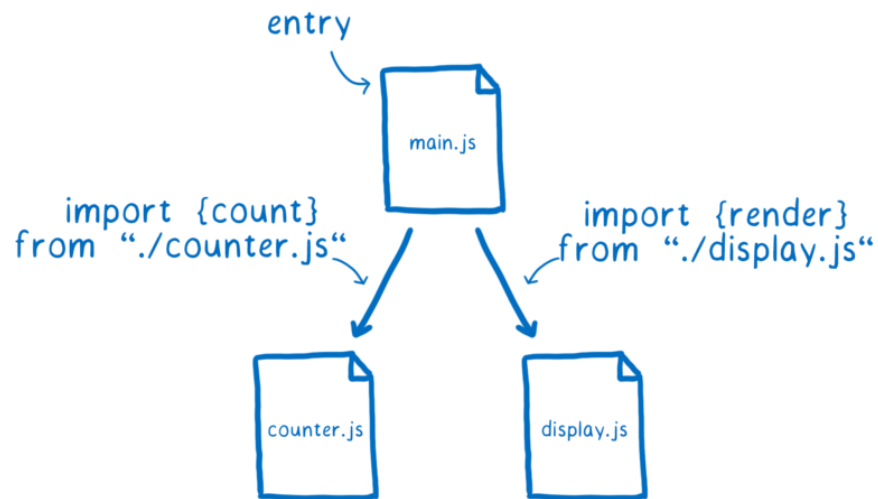
```
// index.js
import moduleA from './a';
//
// import {a} from './a';
console.log('a', moduleA);

// a.js
export default 'a';
//
// export const a = 'a';
//
// const a = 'a';
// export { a };
// ...
```

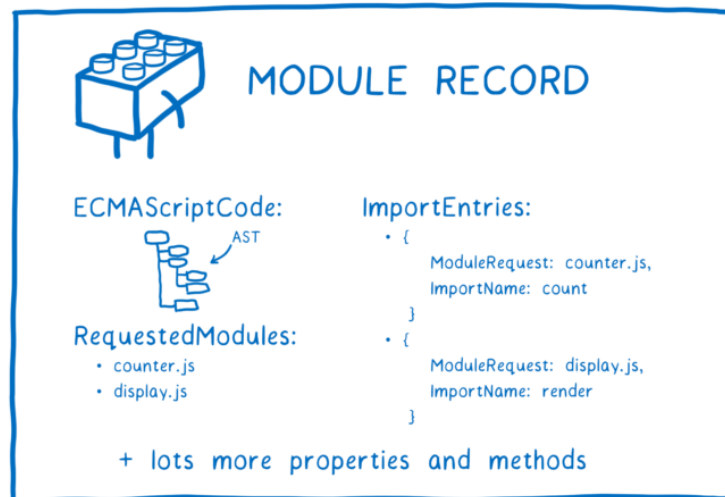
`import` 是导入；`export` 和 `export default` 是导出，`export` 是普通导出，可以导出 对象、字面量等，`export default` 的含义也是导出，不过它是默认导出，一个文件有且仅有一个 默认导出，`export` 和 `export default` 导出时，他们的 `import` 也会不一样，即如上述代码所示

ES Modules 如何工作

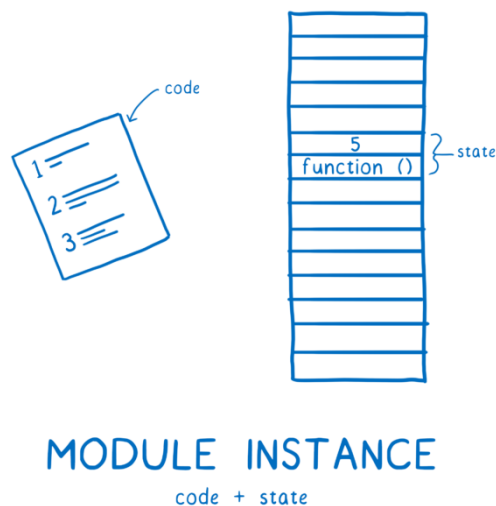
当我们使用 **ES Modules** 进行开发时，实际上是在构建一个依赖关系图（**模块地图**）。不同依赖项之间通过**导入语句**来进行关联。浏览器 或 Node 通过这些 **import语句** 判断加载哪些代码。从**入口文件**开始查找其余文件代码



但是文件本身并不是浏览器可以使用的。它需要解析所有这些文件以将它们转换为称为**模块记录**的数据结构。这样，它实际上知道文件中发生了什么



之后，需要将模块记录转化为**模块实例**，它包含两个部分：**代码**和**状态**：代码基本上是一组指令。这就像一个如何制作东西的食谱，而状态好比制作东西的原材料。所以**模块实例**是将代码（指令列表）与状态（所有变量的值）结合起来

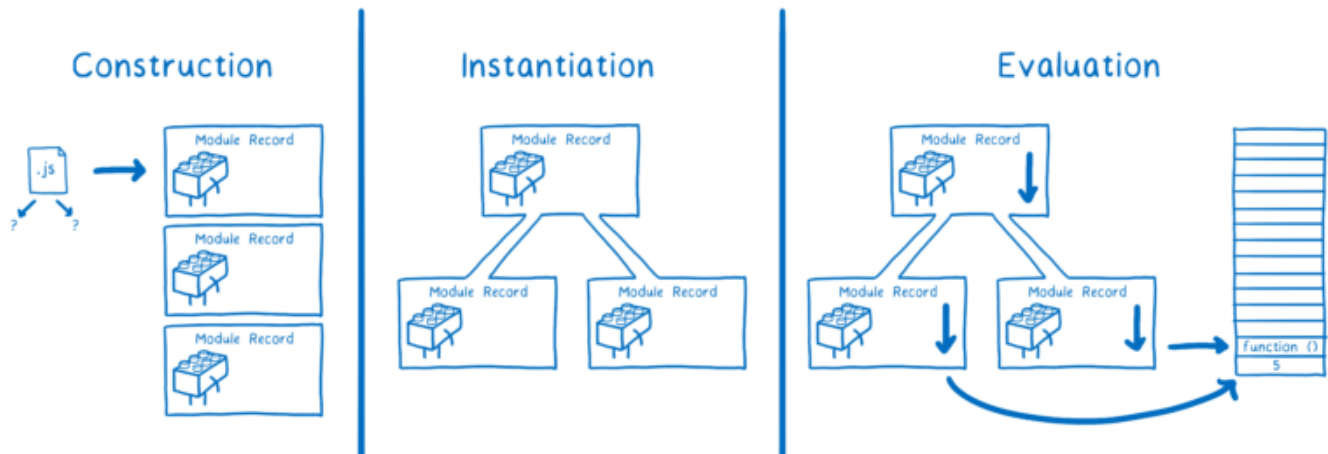


MODULE INSTANCE
code + state

我们需要的就是最后生成的**模块实例**。模块加载的过程就是从入口文件到拥有一个完整的**模块实例图**的过程，对于 ES 模块来说，分三步进行：

1. **构造** — 查找、下载并解析所有文件到模块记录中

2. **实例化** —在内存中寻找一块区域来存储所有导出的变量（但还没有填充值）。然后让 `export` 和 `import` 都指向这些内存块。这个过程叫做链接（linking）
3. **求值** —在内存块中填入变量的实际值

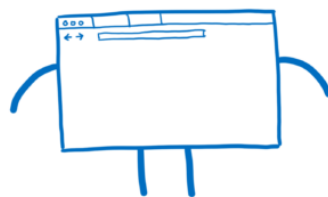


大家都在说 ES 模块是异步的。的确，你可以将其视为异步，因为它运作分为三个不同的阶段——**加载**（即构造）、**实例化**和**求值**——并且这些阶段可以单独完成。

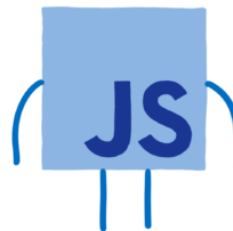
这意味着规范确实引入了一种 CommonJS 中不存在的异步，在CommonJS 中，模块及其依赖项是一次性加载、实例化和求值的，中间没有任何中断。

但是，这些步骤本身不一定是异步的。它们可以以同步的方式完成。这取决于加载的内容。那是因为并非一切都受 ES 模块规范控制，实际上再 ES Modules 的运作过程中，会由2个不同的规范覆盖，**ES 模块规范**说明了如何将文件解析为**模块记录**，以及如何**实例化**和**求值**。但是，它并没有说明如何首先获取文件（加载）。**获取文件**需要加载器，并且它在不同的规范中指定的不同。对于浏览器，该规范是**HTML 规范**，不同的平台有不同的加载器

Spec that says ES module spec
how to load modules
(i.e. HTML spec)

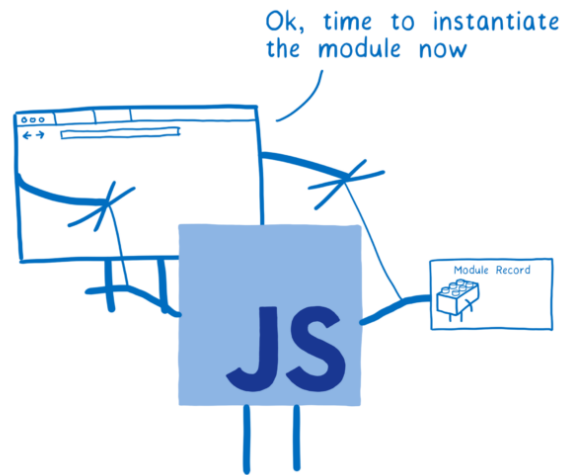


knows how to find
and download module files



knows how to
parse, instantiate,
and evaluate modules

加载器还精确控制模块的加载方式。它调用 ES 模块方法 `ParseModule`、`Module.Instantiate`和`Module.Evaluate`. 这有点像控制 JS 引擎字符串的木偶师



现在让我们更详细地介绍每个步骤：

构造阶段（Construction）

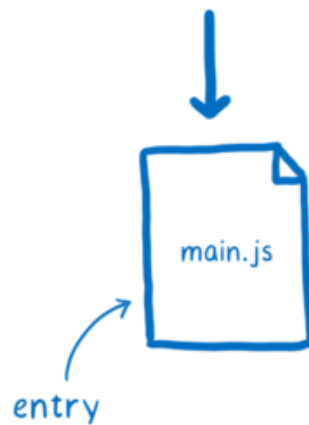
在构造阶段，每个模块都会经历三件事情：

- **查找文件：**找出从哪里下载包含该模块的文件（也称为模块解析）
- **下载文件：**获取文件（从 URL 下载或从文件系统加载）
- **解析文件：**将文件解析为模块记录

查找文件

加载器将负责查找文件并下载它。首先它需要找到入口点文件。在 HTML 中，您使用脚本标记告诉加载程序在哪里找到它

```
<script src="main.js" type="module">
```



但是它如何找到下一组模块（即main.js直接依赖的模块）呢？

这就是**导入语句**的用武之地。导入语句的一部分称为**模块说明符**。它告诉加载器在哪里可以找到每个下一个模块

module specifier

```
import {count} from “./counter.js”
```

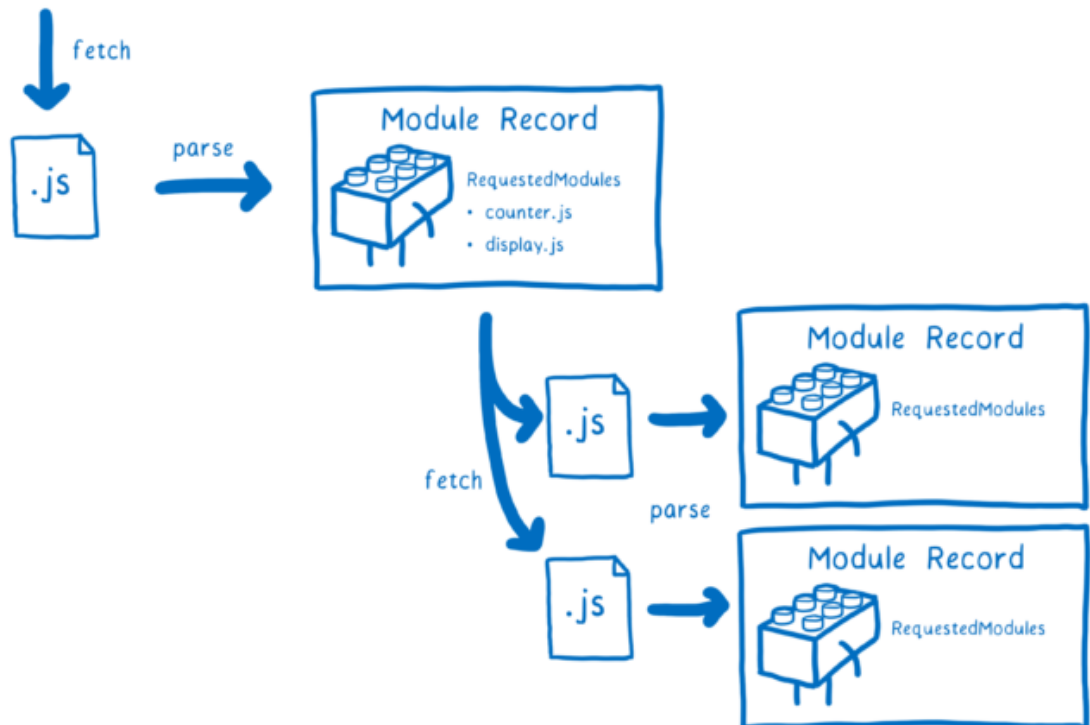
下载文件

关于**模块说明符**需要注意的一件事：它们有时需要在**浏览器**和 **Node.js** 之间进行不同的处理。每个环境都有自己**模块说明符**的运行方式。为此，它使用了一种称为**模块解析算法**的东西，该算法因平台而异

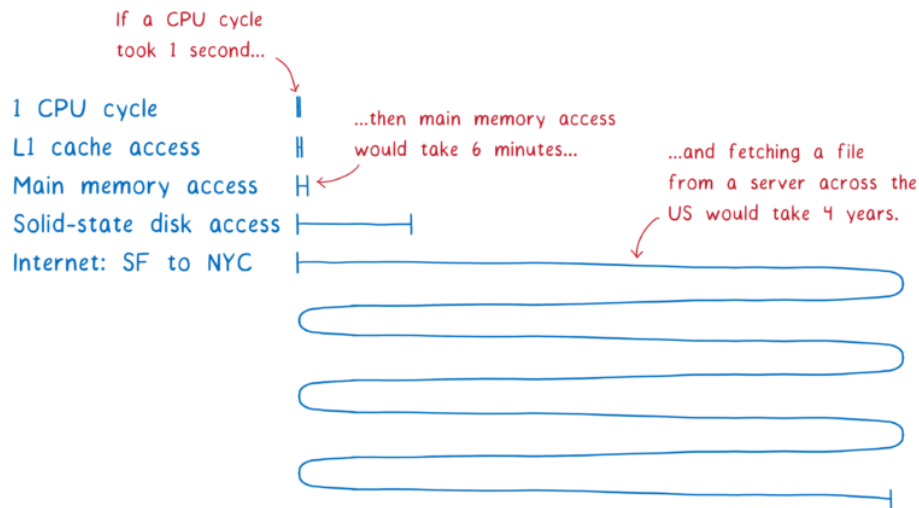
目前浏览器只接受 URL 作为模块说明符。他们将从该 URL 加载模块文件，但这一加载动作在整个模块地图中不会同时发生。因为在解析文件之前，你不知道模块需要获取哪些依赖项……所以在获取文件之前无法解析文件

这意味着我们必须逐层遍历树，解析一个文件，然后找出它们的**依赖关系**，然后找到并加载这些依赖关系

```
<script src=“main.js“ type=“module“>
```



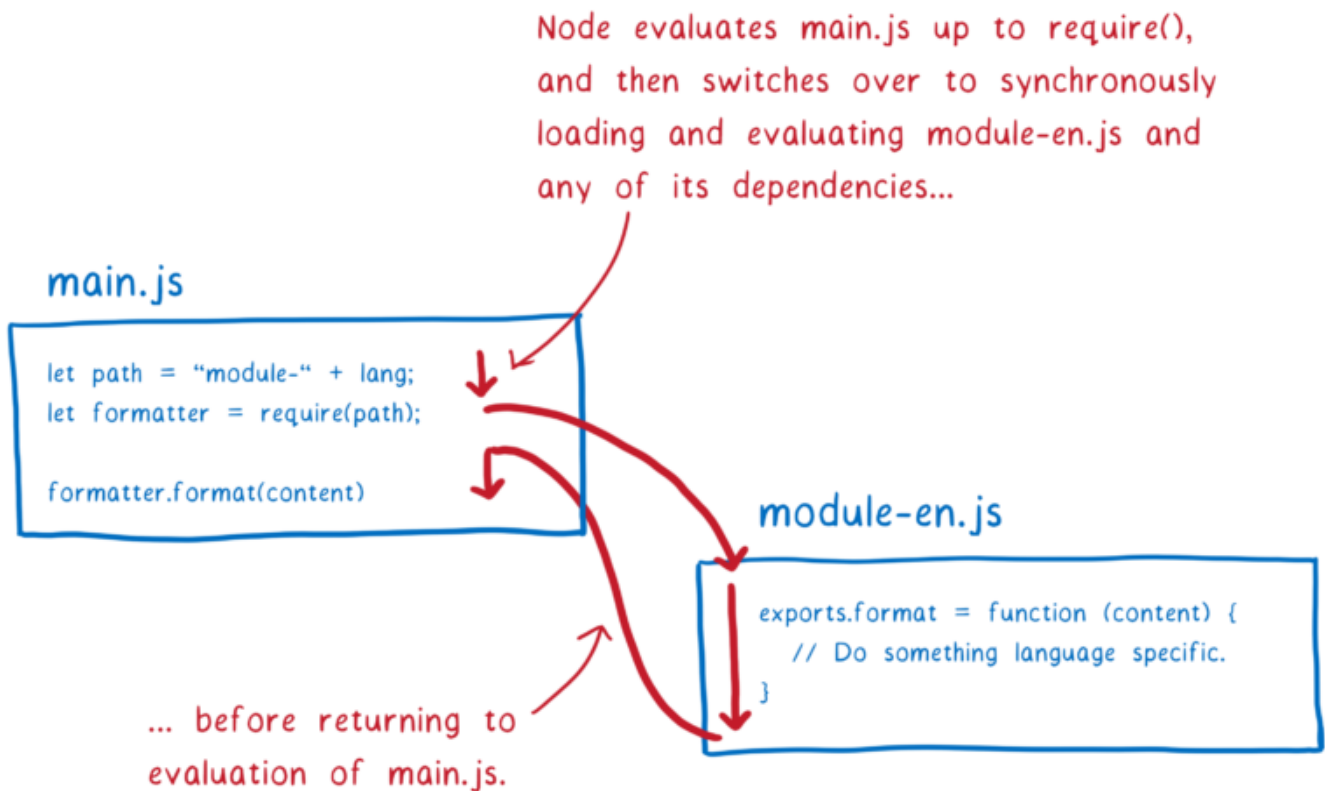
如果主线程要等待每一个文件下载完成，许多其他任务就会堆积在它的队列中，那是因为文件**下载**需要相对较长的时间（下图用等比放大的方式对比了下载文件需要的时间与cpu处理、L1缓存、内存、SSD硬盘所消耗的时间对比）



像这样**阻塞主线程**会使使用模块的应用程序太慢而无法使用。这是 ES 模块规范将**算法拆分**为多个阶段的原因之一。将构造拆分为独立的阶段允许浏览器在进行同步的实例化工作之前获取文件并建立**模块地图**

这种方法 — 将算法分成多个阶段 — 是 ES 模块和 CommonJS 模块之间的主要区别之一

CommonJS 可以做不同的事情，因为从文件系统加载文件所花费的时间比通过 Internet 下载要少得多。这意味着 Node 可以在加载文件时不用考虑是否阻塞主线程。而且由于文件已经加载，所以只实例化和求值（在 CommonJS 中不是单独的阶段）是有意义的。这也意味着在返回模块实例之前，你可以遍历整个树，并且可以对任何依赖项进行加载、实例化和求值



上图可以看出，CommonJS 在 Node 中使用时，您可以在模块说明符中使用变量。在进入下个模块之前，您将执行此模块的所有代码（直到碰到 require）。这意味着当您进行模块解析时，该变量将有一个具体的值。

但是使用 ES 模块时，在你进行任何**求值**之前，需要预先构建整个**模块地图**……这意味着你的**模块说明符**中**不能有变量**，因为这些变量还没有确切的值~

✓ `require(`${path}/counter.js`).count;`

✗ `import {count} from `${path}/counter.js` ;`

can't use variables
in module specifiers
(when using static import statements)

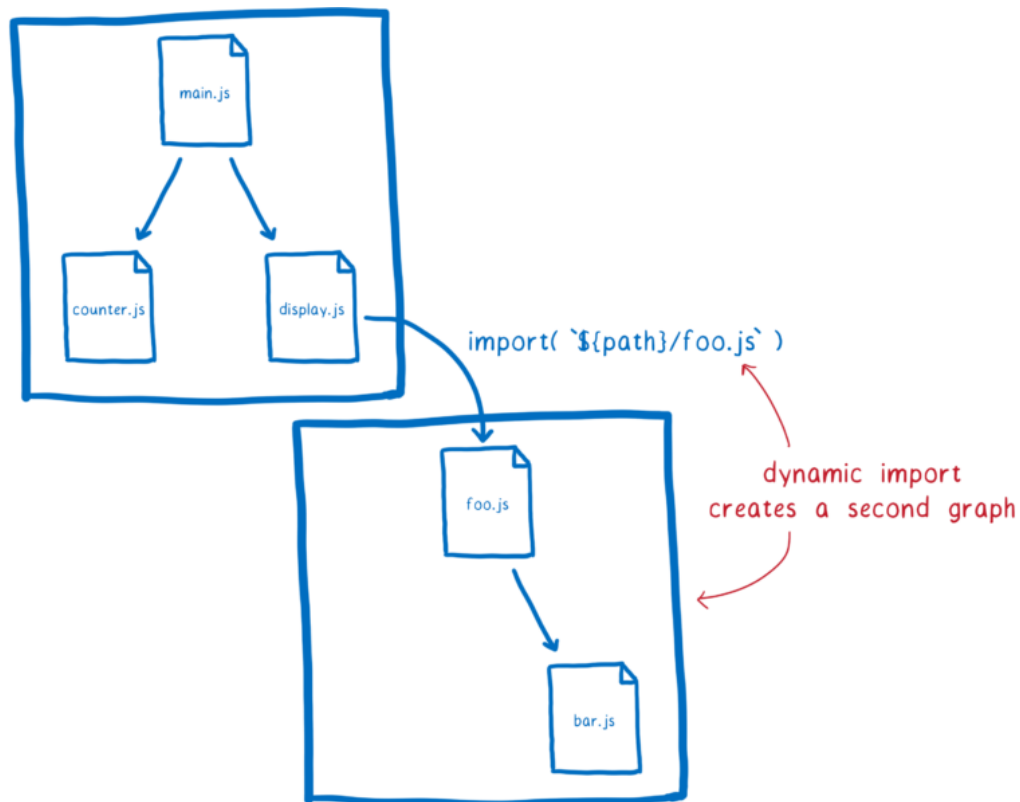
但有时将变量用于模块路径确实很有用！例如，您可能希望根据代码正在执行的环境来切换加载的模块~

为了使 ES 模块成为可能，有一个**动态导入**的提议。有了它，您可以使用类似的导入语句：

ES Modules

```
import(`${path}/foo.js`)
```

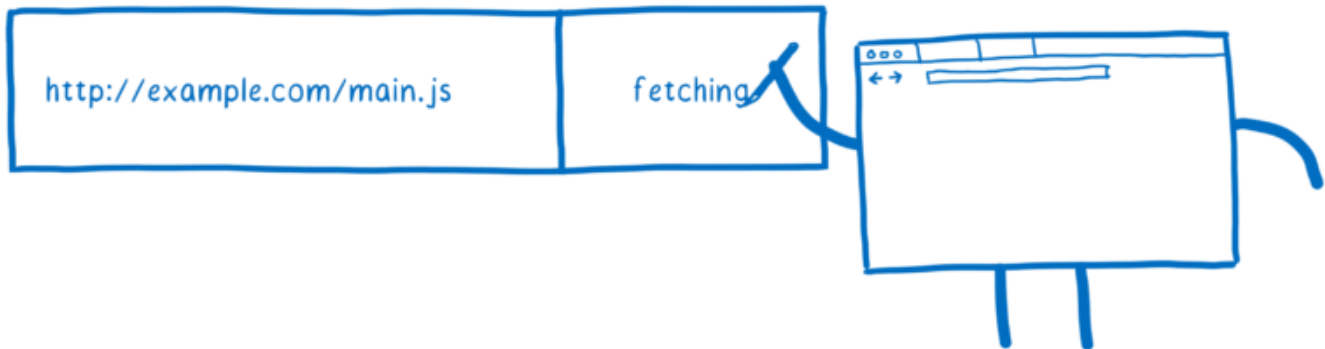
其工作方式任何文件使用 `import()` 加载都被处理为单独**模块地图**的入口。动态导入的模块会启动一个新的**模块地图**~



不过需要注意的一点是：在这两个模块地图中都存在的任何模块都将共享一个模块实例。这是因为加载器缓存了模块实例。对于特定全局范围内的每个模块，将只有一个**模块实例**，这意味着引擎的工作量更少。例如，即使有多个模块依赖，模块文件也只會被获取一次（这是缓存模块原因的其中之一，我们将在求值部分看到另一个）

加载器使用称为**模块映射**的东西来管理这个缓存。每个全局都在单独的**模块映射**中跟踪其模块。当加载器去获取一个 URL 时，它会将该 URL 放入模块映射中并记下它当前**正在获取**（Fetching）该文件。然后它将发出请求并继续开始获取下一个文件

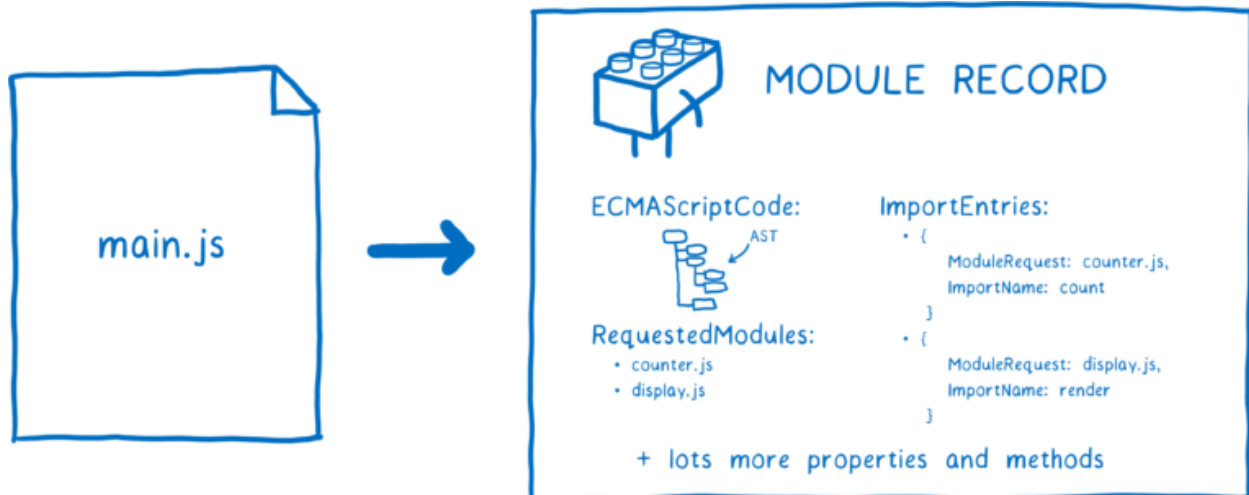
MODULE MAP



如果另一个模块依赖于同一个文件会发生什么？加载器将在**模块映射**中查找每个 URL。如果发现该文件正在 `fetching`，它将继续处理下一个 URL。但是模块映射不只是跟踪**正在获取的文件**。模块映射还用作模块的**缓存**，我们将在接下来看到

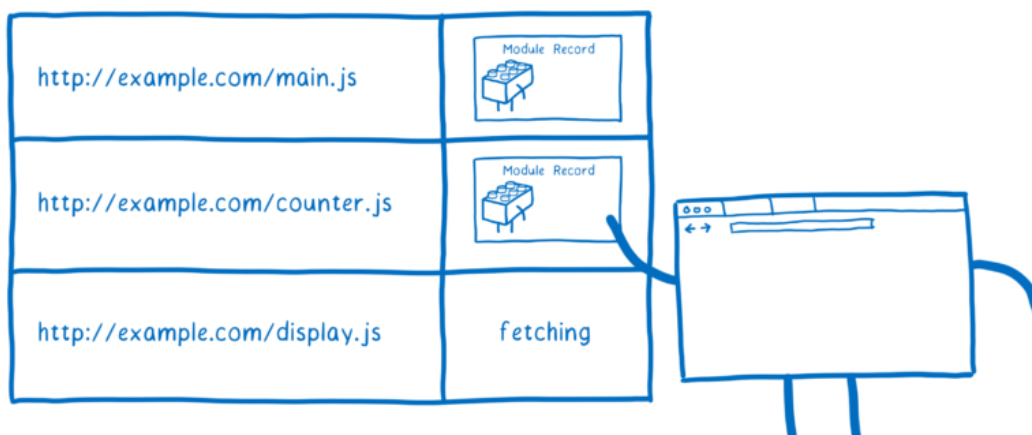
解析文件

现在我们已经获取了这个文件，我们需要将它解析成一个**模块记录**。这有助于浏览器了解模块的不同部分是什么~



一旦创建了**模块记录**，它就会被放置在**模块映射**中。这意味着无论何时从这里请求它，加载程序都可以从该地图中把它重新拉出来~

MODULE MAP

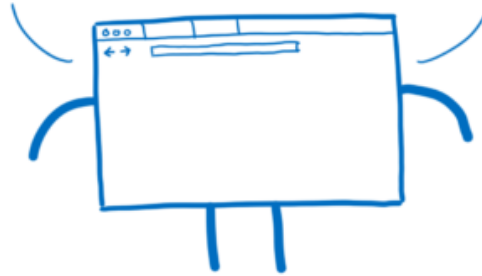


解析中有一个细节看似微不足道，但实际上具有相当大的影响。所有模块都被解析为顶部有“`use strict`”。还有其他一些细微的差别，例如：关键字 `await` 在模块的顶层代码中保留，并且 `this` 值为 `undefined`

这种不同的解析方式称为“parse goal”（目标解析？）。如果你解析同一个文件但使用不同的“goal”，你最终会得到不同的结果。所以你在开始解析之前要知道正在解析什么类型的文件——并且它是否是一个支持 ES Modules 的文件~

Ok, main.js is a module because the type attribute says so...

...and counter.js must be a module because it's imported.



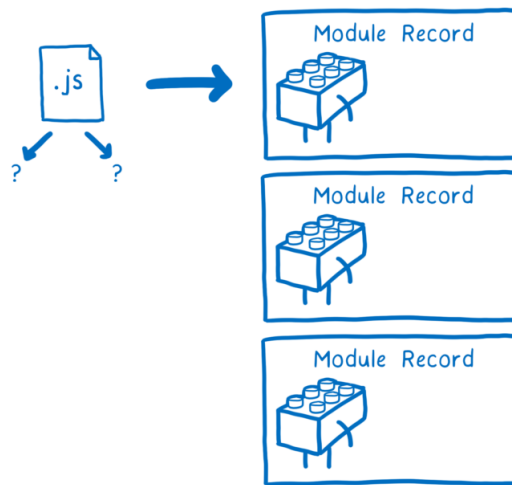
```
<script src="main.js" type="module">
```



但是在 Node 中，您不使用 HTML 标记，因此您没有使用 type 属性的选项。社区尝试解决此问题的一种方法是使用 .mjs 扩展。使用该扩展名告诉 Node，“这个文件支持 ES Modules”

无论哪种方式，加载器都会确定是否将文件解析为模块。如果它是一个模块并且有导入，它将重新开始该过程，直到所有文件都被**下载和解析**

我们完成了！在加载过程结束时，您已经从一个入口文件变为拥有一堆**模块记录**~



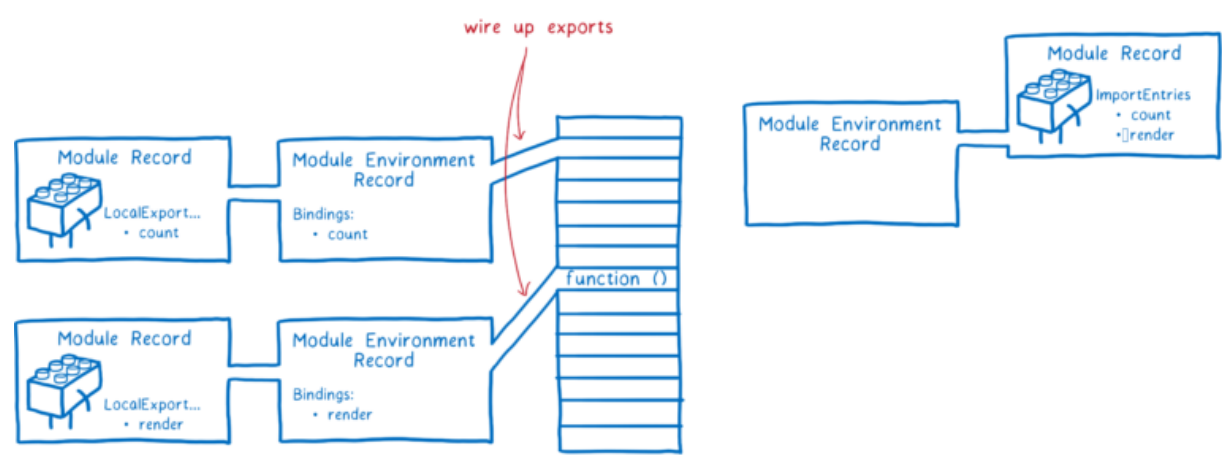
下一步是实例化这个模块并将所有实例链接在一起~

【实例化】阶段（Instantiation）

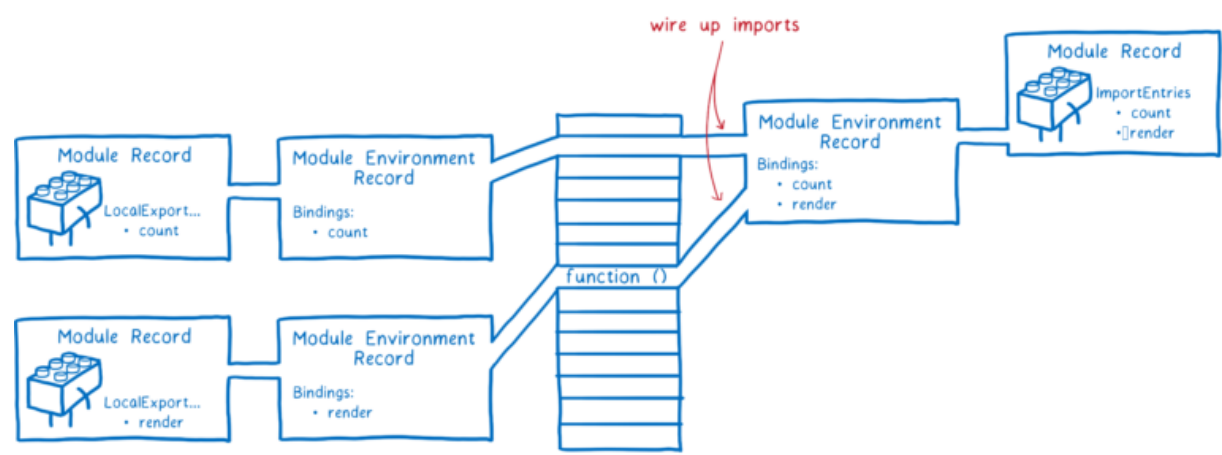
就像之前提到的，一个实例将**代码**与**状态**结合起来。该状态存在于内存中，因此实例化步骤就是写一些东西到内存~

首先，JS引擎创建一个模块环境记录。这里管理模块记录的变量，然后它会在内存中找到所有**导出**内容的区域。模块环境记录将**跟踪**内存中的此区域并与每个导出**相关联**，此时内存中的这些区域还没有值。只有在**求值**之后才会填写它们的实际值。这个规则有一个警告：任何导出的函数声明都在这个阶段**初始化**。这使**求值**变得更容易

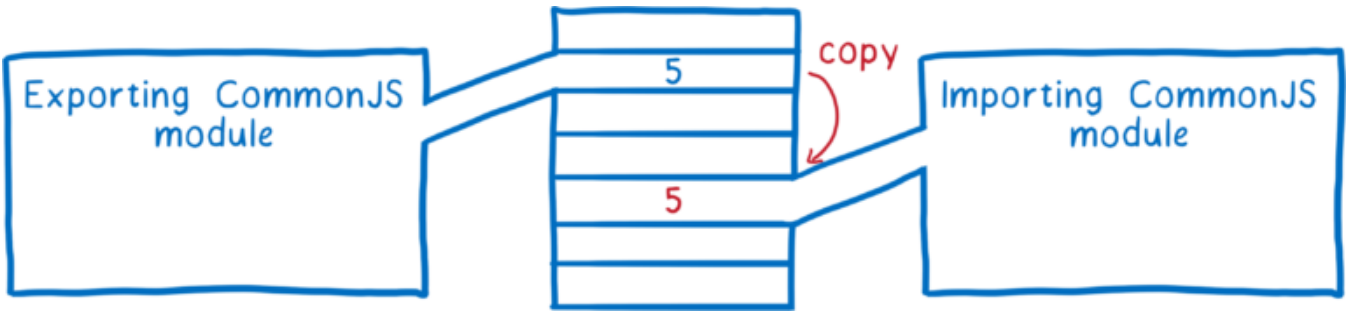
为了实例化**模块地图**，引擎将执行所谓的**深度优先后序遍历**。这意味着它将先遍历到**模块地图**的底部（**从子模块开始实例化**）——到达底部不依赖任何其他东西的依赖项——并设置它们的导出~



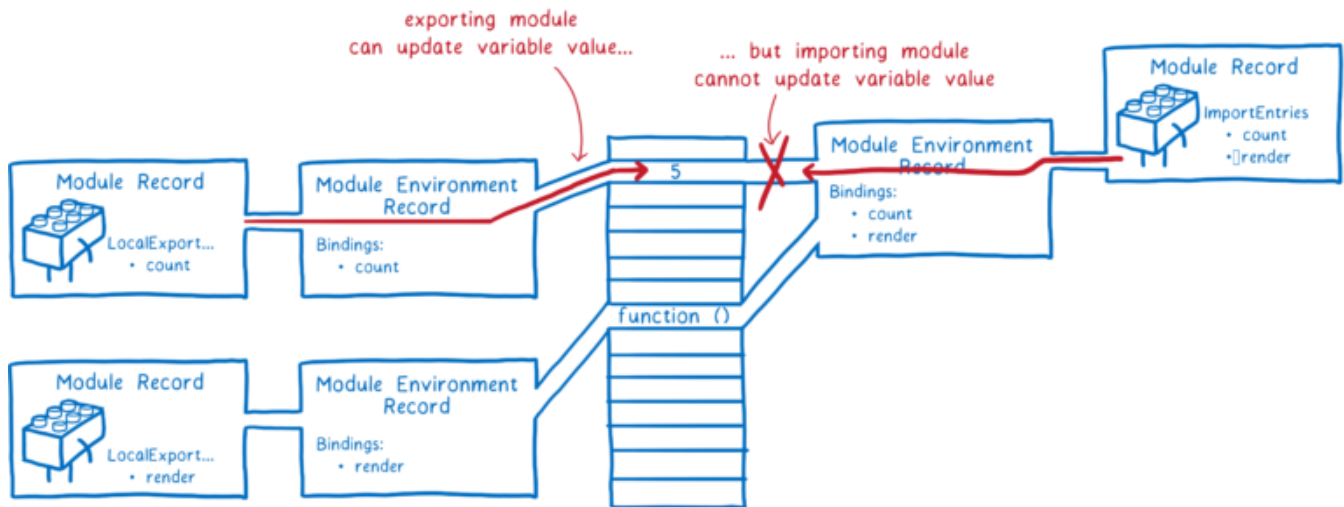
引擎完成了模块下所有导出的连接——模块所依赖的所有**导出**。然后返回一级连接来自上个模块的**导入**。请注意，导出和导入都指向内存中的同一位置。首先连接**导出**可以保证所有**导入**都可以连接到匹配的**导出**



这与 CommonJS 模块不同。在 CommonJS 中，整个导出对象在导出时被复制。这意味着导出的任何值（如数字）都是**副本**。这意味着如果导出模块稍后更改该值，则导入模块不会看到该更改



相比之下，ES 模块使用称为**实时绑定**的东西。两个模块都指向内存中的相同位置。这意味着当导出模块更改值时，该更改将显示在导入模块中。导出值的模块可以随时更改这些值，但导入模块不能更改其导入的值。话虽如此，如果模块导入一个对象，它可以更改该对象上的属性值



像这样进行实时绑定的原因是，您可以在不运行任何代码的情况下连接所有模块。当你的代码出现循环依赖时这有助于你求值，我将在下面解释。所以在这一步结束时，我们已经连接了导出/导入变量的所有实例和内存位置

现在我们可以开始对代码求值并用它们的值填充这些内存位置~

【求值】阶段 (Evaluation)

最后一步是在内存中填充值。JS 引擎通过执行顶层代码（**函数之外的代码，即 import 语句，声明语句，执行语句等等**）来做到这一点。求值阶段除了在内存中填充这些值以外，还可以触发副作用。例如：一个模块可能会调用一个服务~

top level code

```
let count = 5;
updateCountOnServer();
export {count, updateCount};

function updateCount() { ... }
function updateCountOnServer { ... }
```

由于潜在的副作用，您只想对该模块进行一次**求值**。这是使用**模块地图**的原因之一，模块映射通过 URL 缓存模块，以便每个模块只有一个**模块记录**，这确保每个模块只执行一次。就像实例化一样（**从子模块开始求值**），这是作为**深度优先遍历**完成的

循环依赖

下面我们来模拟一下在 ES Modules 中循环依赖的情况：

ES Modules

```
// a.mjs
import b from './b.mjs'

console.log('b', b.message);

export default {
  message: "a Evaluation ",
};
```

ES Modules

```
//b.mjs
import a from './a.mjs';

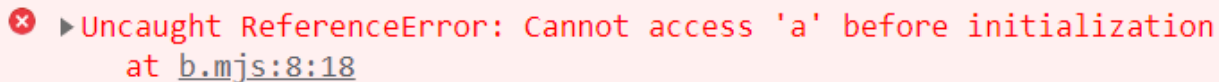
export default {
  message: "b Evaluation ",
};

// [1]
// console.log('a', a.message)
// [2]
// setTimeout(() => console.log('a', a.message), 0)
```

ES Modules

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="./a.mjs" type="module"></script>
</head>
<body>
  66666
</body>
</html>
```

这里人为制造了2个循环依赖的js文件，当我只把 b.mjs 文件 标记[1] 放开时，控制台会有以下报错：“在初始化之前不能访问a”

A screenshot of a browser console error. It shows a red 'x' icon followed by the text 'Uncaught ReferenceError: Cannot access 'a' before initialization' in red. Below this, it says 'at b.mjs:8:18' in blue, with 'b.mjs:8:18' underlined.

出现这样的原因很好理解，根据上述文章，当 ES Modules 被构造和实例化之后，虽然生成了模块地图和模块记录，并且也在内存中开辟了“b.mjs”模块的区域，但是到求值阶段，从子模块开始，此时父模块 a.mjs 还没有执行，a.mjs的导出变量未被初始化，所以 b.mjs 中的导入变量a也就没有被初始化，就会导致 JS 错误

而当我只把 b.mjs 文件 标记[2] 放开时，此时代码就可以成功执行，这是因为异步代码执行的时候父模块已经被执行了~就不会报错

总结

- ES Modules 是由JS 引擎实现，即**语言层面**的底层的实现（当然上层也做了一些适配，比如文件加载那块）
- ES Modules 是提前加载并执行模块文件，在预处理阶段分析模块依赖，在执行阶段执行模块；采用**深度优先遍历**，执行顺序是 子 -> 父；
- ES Modules **循环引用**一般会导致 JS 错误；
- ES Modules 导入导出语句只能写在代码顶层，并且导入导出语句位置不影响模块代码语句执行结果