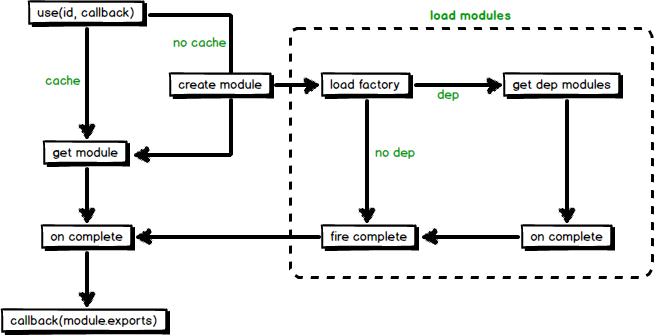
cmd 是阿里大神玉伯提出的基于浏览器的前端模块化规范，并在 seajs 中实现了这个规范。相对于另一个在国外比较流行的前端模块化规范 amd，cmd 对于 nodejs 的使用者来说更加友好，使得类似 commonJS 模块的写法可以在浏览器中使用，同时解决了浏览器中模块异步加载的困扰。 关于 cmd 更详细的内容可以移步 <https://github.com/cmdjs/specification/blob/master/draft/module.md> 今天，我们一起来学习如何实现一个浏览器端的简单的 cmd loader。

**模块加载流程**

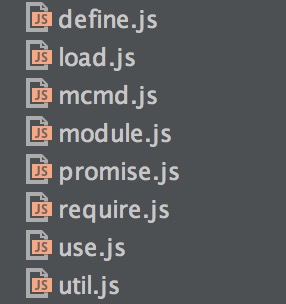
下图展示了一个 cmd loader 的模块加载大体流程：



1. 首先，通过 use 方法来加载入口模块，并接收一个回调函数， 当模块加载完成， 会调用回调函数，并传入对应的模块。use 方法会 check 模块有没有缓存，如果有，则从缓存中获取模块，如果没有，则创建并加载模块。
2. 获取到模块后，模块可能还没有 load 完成，所以需要在模块上绑定一个 "complete" 事件，模块加载完成会触发这个事件，这时候才调用回调函数。
3. 创建一个模块时，id就是模块的地址，通过创建 script 标签的方式异步加载模块的代码（factory），factory 加载完成后，会 check factory 中有没有 require 别的子模块:
   * 如果有，继续加载其子模块，并在子模块上绑定 "complete" 事件，来触发本身 的 "complete" 事件；
   * 如果没有则直接触发本身的 "complete" 事件。
4. 如果子模块中还有依赖，则会递归这个过程。
5. 通过事件由里到外的传递，当所有依赖的模块都 complete 的时候，最外层的入口模块才会触发 "complete" 事件，use 方法中的回调函数才会被调用。

**功能划分**

理解了整个过程，那么我们就来开始实现我们的代码，我们暂且给这个加载器命名为 mcmd 吧。首先是加载器的功能模块划分：



* mcmd：入口文件，用于定义默认配置，参数，常量等，同时使用或加载其他的功能模块；
* define：实现 cmd 中的 "define" 方法；
* require：实现 cmd 中的 "require" 方法；
* use：实现 cmd 中的 "use" 方法；
* module：模块类，实现模块的创建、加载、事件等功能；
* load：用于获取模块，把模块从新建和从 cache 中获取封装成统一的接口；
* promise：异步任务处理器；
* util：工具类函数；

**构建**

我们使用 commonJS 的方式进行编码，并使用 browserify 配合 gulp 来构建我们的项目。

var gulp = require('gulp');

var uglify = require('gulp-uglify');

var concat = require('gulp-concat');

var browserify = require('browserify');

var source = require('vinyl-source-stream');

var buffer = require('vinyl-buffer')

var pg = require('./package');

var versionName = pg.name + '.' + pg.version

gulp.task('default', ['build']);

gulp.task('build', function () {

browserify('./src/mcmd.js')

.bundle()

.pipe(source(versionName))

.pipe(buffer())

.pipe(concat(versionName + '.js'))

.pipe(gulp.dest('./prd'))

.pipe(uglify())

.pipe(concat(versionName + '.min.js'))

.pipe(gulp.dest('./prd'));

});

确定好了功能划分和构建方式，下面我们就来实现每一个功能模块：

**入口文件**

将我们的 cmd loader 挂在 window.mcmd 上，把 define 方法也挂在 window.define 上，初始化其他的方法和配置。

var g = window;

g.define = require('./define');

g.mcmd = {

use: require('./use'),

require: require('./require'),

// 模块缓存

modules: {},

// 默认配置

config: {

root: '/'

},

// 修改配置

setConfig: function (obj) {

for (var key in obj) {

this.config[key] = obj[key];

}

},

// 模块状态常量

MODULE\_STATUS: {

PENDDING: 0,

LOADING: 1,

COMPLETED: 2,

ERROR: 3

}

};

**use.js**

实现了 mcmd.use 方法，接收两个参数，第一个是id或者id数组，第二个是回调函数。内部会使用 load.js 来获取模块，并通过 promise 来处理获取多个模块的并发异步场景。

var Promise = require('./promise');

var load = require('./load');

module.exports = function use(ids, callback) {

if (!Array.isArray(ids)) {

ids = [ids]

}

Promise.all(ids.map(function (id) {

return load(mcmd.config.root + id);

})).then(function (list) {

if (typeof callback === 'function') {

callback.apply(window, list);

}

}, function (errorInfo) {

throw errorInfo;

});

}

**load.js**

获取一个模块，并绑定事件，接收两个参数，一个是模块id，一个是回调函数，并返回一个 promise 对象。当模块 complete（加载完成）时，执行回调，同时 resolve 返回的 promise 对象。

var Promise = require('./promise');

var Module = require('./module');

var util = require('./util');

module.exports = function (id, callback) {

return new Promise(function (resolve, reject) {

var mod = mcmd.modules[id] || Module.create(id);

mod.on('complete', function () {

var exp = util.getModuleExports(mod);

if (typeof callback === 'function') {

callback(exp);

}

resolve(exp);

});

mod.on('error', reject);

});

}

**promise.js**

详见： <http://annn.me/ecmascript6-promise>

**module.js**

模块的构造函数，实现了模块的创建，加载，事件传递，状态维护等。

// 构造函数

function Module(id) {

mcmd.modules[id] = this; // 缓存模块

this.id = id;

this.status = mcmd.MODULE\_STATUS.PENDDING; // 状态

this.factory = null; // 执行代码

this.dependences = null; //依赖

this.callbacks = {}; // 绑定的事件回调函数

this.load();

}

// 静态方法创建模块

Module.create = function (id) {

return new Module(id);

}

// 通过创建 script 标签异步加载模块

Module.prototype.load = function () {

var id = this.id;

var script = document.createElement('script');

script.src = id;

script.onerror = function (event) {

this.setStatus(mcmd.MODULE\_STATUS.ERROR, {

id: id,

error: (this.error = new Error('module can not load.'))

});

}.bind(this);

document.head.appendChild(script);

this.setStatus(mcmd.MODULE\_STATUS.LOADING);

}

// 事件绑定方法

Module.prototype.on = function (event, callback) {

(this.callbacks[event] || (this.callbacks[event] = [])).push(callback);

if (

(this.status === mcmd.MODULE\_STATUS.LOADING && event === 'load') ||

(this.status === mcmd.MODULE\_STATUS.COMPLETED && event === 'complete')

) {

callback(this);

}

if (this.status === mcmd.MODULE\_STATUS.ERROR && event === 'error') {

callback(this, this.error);

}

}

// 事件触发方法

Module.prototype.fire = function (event, arg) {

(this.callbacks[event] || []).forEach(function (callback) {

callback(arg || this);

}.bind(this));

}

// 设置状态方法，并抛出相应的事件

Module.prototype.setStatus = function (status, info) {

if (this.status !== status) {

this.status = status;

switch (status) {

case mcmd.MODULE\_STATUS.LOADING:

this.fire('load');

break;

case mcmd.MODULE\_STATUS.COMPLETED:

this.fire('complete');

break;

case mcmd.MODULE\_STATUS.ERROR:

this.fire('error', info);

break;

default:

break;

}

}

}

module.exports = Module;

**define.js**

实现 window.define 方法。接收一个参数 factory（cmd规范中不止一个，为了保持简单，我们只实现一个），即模块的代码包裹函数。通过 getCurrentScript 这个函数获取到当前执行脚本的 script 节点 src ，提取出模块 id ，找到模块对象。然后提取出 factory 中的依赖子模块，如果没有依赖，则直接触发模块的 "complete" 事件， 如果有依赖，则创建依赖的模块，绑定事件并加载，等依赖的模块加载完成后，再触发 "complete" 事件。

var util = require('./util');

var Promise = require('./promise');

var Module = require('./module');

module.exports = function (factory) {

var id = getCurrentScript().replace(location.origin, '');

var mod = mcmd.modules[id];

var dependences = mod.dependences = getDenpendence(factory.toString());

mod.factory = factory;

if (dependences) {

Promise.all(dependences.map(function (id) {

return new Promise(function (resolve, reject) {

id = mcmd.config.root + id;

var depMode = mcmd.modules[id] || Module.create(id);

depMode.on('complete', resolve);

depMode.on('error', reject);

});

})).then(function () {

mod.setStatus(mcmd.MODULE\_STATUS.COMPLETED);

}, function (error) {

mod.setStatus(mcmd.MODULE\_STATUS.ERROR, error);

});

}

else {

mod.setStatus(mcmd.MODULE\_STATUS.COMPLETED);

}

}

// 获取当前执行的script节点

// 参考 http://www.cnblogs.com/rubylouvre/archive/2013/01/23/2872618.html

function getCurrentScript() {

var doc = document;

if(doc.currentScript) {

return doc.currentScript.src;

}

var stack;

try {

a.b.c();

} catch(e) {

stack = e.stack;

if(!stack && window.opera){

stack = (String(e).match(/of linked script \S+/g) || []).join(" ");

}

}

if(stack) {

stack = stack.split( /[@ ]/g).pop();

stack = stack[0] == "(" ? stack.slice(1,-1) : stack;

return stack.replace(/(:\d+)?:\d+$/i, "");

}

var nodes = head.getElementsByTagName("script");

for(var i = 0, node; node = nodes[i++];) {

if(node.readyState === "interactive") {

return node.className = node.src;

}

}

}

// 解析依赖，这里只做简单的提取，实际需要考虑更多情况，参考seajs

function getDenpendence(factory) {

var list = factory.match(/require\(.+?\)/g);

if (list) {

list = list.map(function (dep) {

return dep.replace(/(^require\(['"])|(['"]\)$)/g, '');

});

}

return list;

}

**require.js**

返回模块的 exports 属性， 这里通过封装的 util.getModuleExports 方法获取并返回。

var util = require('./util');

module.exports = function (id) {

id = mcmd.config.root + id;

var mod = mcmd.modules[id];

if (mod) {

return util.getModuleExports(mod);

}

else {

throw 'can not get module by from:' + id;

}

}

module.exports.async = function (ids, callback) {

mcmd.use(ids, callback);

}

**util.js**

这里只有一个 getModuleExports 方法， 接收一个模块，返回模块的接口。当模块的 exports 属性不存在时，说明模块的 factory 没有被执行过。这时我们需要执行下 factory，传入 require， 创建的exports，以及 module 本身作为参数。最后获取模块的暴露的数据并返回。

module.exports = {

getModuleExports: function (mod) {

if (!mod.exports) {

mod.exports = {};

mod.factory(mcmd.require, mod.exports, mod);

}

return mod.exports;

}

};

这样，整个 cmd loader 就基本完成了。这只是一个非常基础的模块加载器，主要是为了理解 cmd 的原理和实现方式，对于生产环境，推荐使用成熟的 seajs。

整个 mcmd 项目我都放在了 github 上，大家可以去看看：<https://github.com/hanan198501/mcmd>。