通行的Javascript模块规范共有两种：CommonJS和AMD。

CommonJS

require()，用于加载模块

```
var math = require('math');
math.add(2,3); // 5
```

第二行math.add(2, 3)，在第一行require('math')之后运行，因此必须等math.js加载完成。也就是说，如果加载时间很长，整个应用就会停在那里等。这对服务器端不是一个问题，因为所有的模块都存放在本地硬盘，可以同步加载完成，等待时间就是硬盘的读取时间。但是，对于浏览器，这却是一个大问题，因为模块都放在服务器端，等待时间取决于网速的快慢，可能要等很长时间，浏览器处于"假死"状态。
因此，浏览器端的模块，不能采用"同步加载"（synchronous），只能采用"异步加载"（asynchronous）。这就是AMD规范诞生的背景

AMD

AMD是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义"。它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行。

AMD也采用require()语句加载模块，但是不同于CommonJS，它要求两个参数：

```
require([module], callback);
```

第一个参数[module]，是一个数组，里面的成员就是要加载的模块；第二个参数callback，则是加载成功之后的回调函数。如果将前面的代码改写成AMD形式，就是下面这样：

```
  require(['math'], function (math) {
       math.add(2, 3);
    });
```

math.add()与math模块加载不是同步的，浏览器不会发生假死。所以很显然，AMD比较适合浏览器环境。

为什么要用require.js?

最早的时候，所有Javascript代码都写在一个文件里面，只要加载这一个文件就够了。后来，代码越来越多，一个文件不够了，必须分成多个文件，依次加载。

```
<script src="1.js"></script>
<script src="2.js"></script>
<script src="3.js"></script>
<script src="4.js"></script>
```

这段代码依次加载多个js文件。这样的写法有很大的缺点。
首先，加载的时候，浏览器会停止网页渲染，加载文件越多，网页失去响应的时间就会越长；
其次，由于js文件之间存在依赖关系，因此必须严格保证加载顺序（比如上例的1.js要在2.js的前面），依赖性最大的模块一定要放到最后加载，当依赖关系很复杂的时候，代码的编写和维护都会变得困难。

require.js的诞生，就是为了解决这两个问题：
（1）实现js文件的异步加载，避免网页失去响应；
（2）管理模块之间的依赖性，便于代码的编写和维护。

require.js的加载

使用require.js

```
  <script src="js/require.js"></script>
```

有人可能会想到，加载这个文件，也可能造成网页失去响应。解决办法有两个，一个是把它放在网页底部加载，另一个是写成下面这样：

```
<script src="js/require.js" defer async="true" ></script>
```

async属性表明这个文件需要异步加载，避免网页失去响应。IE不支持这个属性，只支持defer，所以把defer也写上。

加载require.js以后，下一步就要加载我们自己的代码了。假定我们自己的代码文件是main.js，也放在js目录下面。那么，只需要写成下面这样就行了：

```
<script src="js/require.js" data-main="js/main"></script>
```

data-main属性的作用是，指定网页程序的主模块。在上例中，就是js目录下面的main.js，这个文件会第一个被require.js加载。由于require.js默认的文件后缀名是js，所以可以把main.js简写成main。

主模块的写法

上一节的main.js，我把它称为"主模块"，意思是整个网页的入口代码。它有点像C语言的main()函数，所有代码都从这儿开始运行。
主模块依赖于其他模块，这时就要使用AMD规范定义的的require()函数。

```
// main.js
    require(['moduleA', 'moduleB', 'moduleC'], function (moduleA, moduleB,
moduleC){
        // some code here
    });
```

require()函数接受两个参数。第一个参数是一个数组，表示所依赖的模块，上例就是['moduleA', 'moduleB', 'moduleC']，即主模块依赖这三个模块；第二个参数是一个回调函数，当前面指定的模块都加载成功后，它将被调用。加载的模块会以参数形式传入该函数，从而在回调函数内部就可以使用这些模块。

require()异步加载moduleA，moduleB和moduleC，浏览器不会失去响应；它指定的回调函数，只有前面的模块都加载成功后，才会运行，解决了依赖性的问题。

假定主模块依赖jquery、underscore和backbone这三个模块，main.js就可以这样写：

```
require(['jquery', 'underscore', 'backbone'], function ($, _, Backbone){
        // some code here
    });
```

require.js会先加载jQuery、underscore和backbone，然后再运行回调函数。主模块的代码就写在回调函数中。

模块的加载

上一节最后的示例中，主模块的依赖模块是['jquery', 'underscore', 'backbone']。默认情况下，require.js假定这三个模块与main.js在同一个目录，文件名分别为jquery.js，underscore.js和backbone.js，然后自动加载。

使用require.config()方法，我们可以对模块的加载行为进行自定义。require.config()就写在主模块（main.js）的头部。参数就是一个对象，这个对象的paths属性指定各个模块的加载路径。

```
require.config({
       paths: {
          "jquery": "jquery.min",
          "underscore": "underscore.min",
          "backbone": "backbone.min"
       }
   });
```

上面的代码给出了三个模块的文件名，路径默认与main.js在同一个目录（js子目录）。如果这些模块在其他目录，比如js/lib目录，则有两种写法。一种是逐一指定路径。

```
require.config({
       paths: {
          "jquery": "lib/jquery.min",
          "underscore": "lib/underscore.min",
          "backbone": "lib/backbone.min"
       }
   });
```

另一种则是直接改变基目录（baseUrl）。

```
require.config({
       baseUrl: "js/lib",
       paths: {
          "jquery": "jquery.min",
          "underscore": "underscore.min",
          "backbone": "backbone.min"
       }
   });
```

```
require.config({
       paths: {
          "jquery":
"https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min"
       }
   });
```

require.js要求，每个模块是一个单独的js文件。这样的话，如果加载多个模块，就会发出多次HTTP请求，会影响网页的加载速度。因此，require.js提供了一个优化工具，当模块部署完毕以后，可以用这个工具将多个模块合并在一个文件中，减少HTTP请求数。

AMD模块的写法

require.js加载的模块，采用AMD规范。
就是模块必须采用特定的define()函数来定义。如果一个模块不依赖其他模块，那么可以直接定义在define()函数之中。
假定现在有一个math.js文件，它定义了一个math模块。那么，math.js就要这样写：

```
// math.js
define(function (){
    var add = function (x,y){
        return x+y;
    };
    return {
        add: add
    };
});
```

加载方法

```
// main.js
require(['math'], function (math){
    alert(math.add(1,1));
});
```

如果这个模块还依赖其他模块，那么define()函数的第一个参数，必须是一个数组，指明该模块的依赖性。

```
define(['myLib'], function(myLib){
        function foo(){
            myLib.doSomething();
        }
        return {
            foo : foo
        };
    });
```

当require()函数加载上面这个模块的时候，就会先加载myLib.js文件。

加载非规范的模块

理论上，require.js加载的模块，必须是按照AMD规范、用define()函数定义的模块。但是实际上，虽然已经有一部分流行的函数库（比如jQuery）符合AMD规范，更多的库并不符合。那么，require.js是否能够加载非规范的模块呢？回答是可以的。

这样的模块在用require()加载之前，要先用require.config()方法，定义它们的一些特征。
举例来说，underscore和backbone这两个库，都没有采用AMD规范编写。如果要加载它们的话，必须先定义它们的特征。

```
require.config({
       shim: {
          'underscore':{
             exports: '_'
          },
          'backbone': {
             deps: ['underscore', 'jquery'],
             exports: 'Backbone'
          }

       }

    });
```

require.config()接受一个配置对象，这个对象除了有前面说过的paths属性之外，还有一个shim属性，专门用来配置不兼容的模块。具体来说，每个模块要定义（1）exports值（输出的变量名），表明这个模块外部调用时的名称；（2）deps数组，表明该模块的依赖性。

比如，jQuery的插件可以这样定义：

```
shim: {
       'jquery.scroll': {
          deps: ['jquery'],
          exports: 'jQuery.fn.scroll'
       }
    }
```

require.js插件
require.js还提供一系列插件，实现一些特定的功能。
domready插件，可以让回调函数在页面DOM结构加载完成后再运行。

```
  require(['domready!'], function (doc){
       // called once the DOM is ready
    });
```

```
  define([
       'text!review.txt',
       'image!cat.jpg'
       ],
       function(review,cat){
          console.log(review);
          document.body.appendChild(cat);
       }
    );
```

require() 源码解读

浏览器加载 CommonJS 模块的原理与实现

AMD与CMD的区别

AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。
CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。
类似的还有 CommonJS Modules/2.0 规范，是 BravoJS 在推广过程中对模块定义的规范化产出。
这些规范的目的都是为了 JavaScript 的模块化开发，特别是在浏览器端的。
目前这些规范的实现都能达成浏览器端模块化开发的目的。
区别：

1. 对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）。CMD 推崇 as lazy as possible.
2. CMD 推崇依赖就近，AMD 推崇依赖前置。

```
// CMD
define(function(require, exports, module) {
    var a = require('./a')
    a.doSomething() // 此处略去 100 行
    var b = require('./b') // 依赖可以就近书写
    b.doSomething()
    // ...
})
```

```
// AMD 默认推荐的是
define(['./a', './b'], function(a, b) {
    // 依赖必须一开始就写好 a.doSomething()
    // 此处略去 100 行
    b.doSomething()
    //...
})
```

虽然 AMD 也支持 CMD 的写法，同时还支持将 require 作为依赖项传递，但 RequireJS 的作者默认是最喜欢上面的写法，也是官方文档里默认的模块定义写法。

```
//require.js案例
require.config({
    paths: {
        "m1": "m1",
        "underscore": "m2",
        "backbone": "m3"
    }
});
require(['m1', 'underscore', 'backbone'], function(m0, m2, m3) {
    console.log(m0)
    console.log(m2)
    console.log(m3)
    console.log( m0.add(1,2) )
    console.log( m2.remove(3) )
    console.log( m3.getname(5) )
})
```

```
//m1.js
define(function (){
       var add = function (x,y){
          return x+y;
       };
       return {
          add: add
       };
   });
```

```
//m2.js
define(function (){
       var remove = function (x){
          return x;
       };
       return {
            remove: remove
       };
  });
```

```
//m3.js
define(['m4'], function (m4){
       var getname = function (x){
          return m4.setname;
       };
       return {
            getname: getname
       };
   });
```

```
//m4.js
define(function (){
       var setname = function (x){
          return x
       };
       return {
            setname: setname
       };
   });
```

```
//sea.js
....
```

```
//main.js
define(function(require) {
    var m1 = require("./m1.js")
    var m2 = require("./m2.js")
    console.log(m1)  //{getName , setName }
    console.log(m2) //{a, aa}
})
```

```
//m1.js
define(function(require, exports) {
    var m3 = require("./m3.js")

    exports.getName = function() {
        return m3.getName
    }
    exports.setName = function(str) {
        return str
    }
})
```

```
//m2.js
define(function(require, exports) {
    var m3 = require("./m3.js")
    var a = 'cxh'
    console.log(m3)   // {getName }
    return {
        a: a,
        aa: m3.getName
    }
})
```

```
//m3.js
define(function(require, exports) {
    exports.getName = function(str) {
        return str
    }
})
```

ES6 Module

ES6 浏览器的模块化标准。

模块引入

浏览器使用以下方式引入一个ES6模块化文件

```
<script src="./xxx.js" type="module"></script>
```

模块导出

1. 模块导出分为两种， 基本导出和默认导出

   ```
   export var a = 1 // 基本导出a=1
   var c = 3; export {c} // 基本导出c=3.
   var c = 3; export {c as temp} // 基本导出temp=3.
   export {c as default} // 默认导出default = 3。
   ```

   ```
   export default 3 // 默认导出default = 3
   ```

   我们平时 直接 export default 一把梭。

模块导入

```
import {a, b} from "模块路径"
import {a as temp1 , b as temp2} from "模块路径"
import {defalut as a} from "模块路径"
import c from "模块路径"  //相当于import {defalut as c} from "模块路径"
import * as obj from "模块路径"
```

模块导入时注意

1. ES6 module 采用依赖预加载模式， 所有模块导入均会提升到代码顶部
2. 不能将导入代码放置到判断， 循环中
3. 导入的内容放置到常量中， 不可更改
4. ES6 module使用了缓存， 保证每个模块仅加载一次

- loader：webpack 自身只理解 JavaScript，loader 能够去处理非 JavaScript 文件并转化 JavaScript，处理源文件，一次处理一个。
- plugins：用来扩展 webpack 功能，插件能够执行很多任务。如：打包优化、压缩等。

  常用 Plugin:

  - UglifyJsPlugin: 压缩、混淆代码；
  - CommonsChunkPlugin: 代码分割；
  - ProvidePlugin: 自动加载模块；
  - html-webpack-plugin: 加载 html 文件，并引入 css / js 文件；
  - extract-text-webpack-plugin / mini-css-extract-plugin: 抽离样式，生成 css 文件；
  - DefinePlugin: 定义全局变量；
  - optimize-css-assets-webpack-plugin: CSS 代码去重；
  - webpack-bundle-analyzer: 代码分析；
  - compression-webpack-plugin: 使用 gzip 压缩 js 和 css；
  - happypack: 使用多进程，加速代码构建；
  - EnvironmentPlugin: 定义环境变量；

  Webpack 就像工厂中的一条产品流水线。原材料经过 Loader 与 Plugin 的一道道处理，最后输出结果。

  - 通过链式调用，按顺序串起一个个 Loader；
  - 通过事件流机制，让 Plugin 可以插入到整个生产过程中的每个步骤中；

构建流程

- 生成 options（将 webpack.config.js 和 shell 中的参数合并到 options 对象）。
- 实例化 compiler 对象 （webpack 全局对象，包含 entry、output、loader、plugins等所有配置对象）。
- 实例化 compilation 对象（compiler.run 方法执行，开始编译过程，生成 compilation 对象）。
- 分析入口 js 文件，调用 AST 引擎处理入口文件，生成抽象语法树 AST，根据 AST 构建模块的所有依赖。
- 通过 loader 处理入口文件的所有依赖，转换为 js 模块，生成 AST，然后继续递归遍历，直至所有依赖被分析完毕。
- 对生成的所有 module 进行处理，调用 plugins，合并，拆分，生成 chunk。
- 将 chunk 生成为对应 bundle 文件，输出到目录。

使用less
定义全局 css作用域包裹 避免冲突

饿了么团队开源一个基于vue 组件库
elementUI　PC官网 http://element.eleme.io/
MintUI 移动端 http://mint-ui.github.io/
iView 是一套基于 Vue.js 的 UI 组件库，主要服务于 PC 界面的中后台产品

Gulp是一个工具，webpack是模块化方案

gulp

gulp强调的是前端开发的工作流程，我们可以通过配置一系列的task，定义task处理的事务（例如文件压缩合并、雪碧图、启动server、版本控制等），然后定义执行顺序，来让gulp执行这些task，从而构建项目的整个前端开发流程。
PS：简单说就一个Task Runner

webpack

webpack是一个前端模块化方案，更侧重模块打包，我们可以把开发中的所有资源（图片、js文件、css文件等）都看成模块，通过loader（加载器）和plugins（插件）对资源进行处理，打包成符合生产环境部署的前端资源。
PS：webpack is a module bundle