**export\_1516687482071**

exports与 module.exports属于 commonJS规范

export与export default 属于 ES6规范

都属于导出语法

分别对应的导入语法： require（） import

exports 与 module.exports的区别

commonJS将每个文件视为一个模块，每个模块内部，module变量代表当前模块，这个变量是一个对象，它的exports属性（module.exports）是对外的接口。加载一个模块，就是加载这个module.exports属性。

而exports指向 module.exports，可以改变exports对象的属性，但是不能直接将exports指向另一个值。

export 与 export default的区别和联系

是ES6中导出的接口，一个是导出一个一个的单独接口，一个是导出整体的接口。

**absolute\_1516945381838**

absolute定位于scroll元素的表现

定位元素会固定在一开始的位置，并不会随滚动而滚动。（滚动的部分为超出部分）

#### [anonymous function](C:/Users/Administrator/AppData/Local/youdao/dict/Application/7.5.2.0/resultui/dict/javascript:;)\_1519290561525

实际上的匿名函数也包括表达式函数（但是）

**recursion\_1519611483438**

一、函数调用

function isPalindrome( text ) {

if ( text.length <=1 ) return true

if( text.charAt(0) !== text.charAt(text.length -1) ) return false

return isPalindrome( text.substr( 1, text.length - 2 ) )

}

二、方法调用

var ninja = {

chirp: function(n) {

return n > 1 ? ninja.chirp(n - 1) + "-chirp" : "chirp";

}

};

var samurai = { chirp: ninja.chirp };

**优化**

return n > 1 ? this.chirp(n - 1) + "-chirp" : "chirp";

1. **内联命名函数**

上面的问题在于，其他对象引用时必须给方法命名为chirp，而用内联命令函数则不会有这个问题

var ninja = {

chirp: function signal(n) { //#1

return n > 1 ? signal(n - 1) + "-chirp" : "chirp";

}

};

内联函数的名称在函数外是不可用的。只在函数内部是可用的。如：

var a = function b() {

}

b -- 》 undefined

这也是为什么全局属性会被作为window的方法进行创建的原因，不使用window的属性，无法引用这些函数。

注意： arguments.callee() 也可以调用当前的函数，但是，es5规范中禁止使用这个属性。

fetch\_1519795453831

ajax的取代品 ：fetch

fetch是基于Promise设计

一、用await

    try {

      const response = await fetch(url, requestConfig);

      const responseJson = await response.json();

      return responseJson

    } catch (error) {

      throw new Error(error)

    }

**二、promise形式**

fetch(url).then(response => response.json())

.then(data => console.log(data))

.catch(e => console.log("Oops, error", e))

**问题：**

1. Fetch 请求默认是不带 cookie 的，需要设置 fetch(url, {credentials: 'include'})

2.服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。

3. IE不支持原生Fetch。

cross-origin\_1520389506693

为什么存在同源策略

简单来说：

1. 非同源的页面不能产生交互（iframe）

主要是防止网站伪造，如果伪造者用一个irame将目标网站防止其网站内部，通过样式设置起来看起来没有任何不同，如果非同源页面可以交互，用户的输入都能被获取，包括密码。

1. 非同源的请求需要在授权的情况下才能读取数据（cors）,包括cookie传递

这是避免攻击者直接发送非同源请求（附带权限认证的cookie，如果他能拿到的h话），效果爆炸

但是针对cookie这个特殊的数据，非相同二级域下，无法共享。

1、 document.domain + iframe跨域

**能实现同一二级域下的子域之间的iframe交互**

缺点：

必须在两个页面都设置同一个domain，且只能设置为自身或者上一级domain，而且都要设置

document.domain = ‘dongyuewu.me’

2、 location.hash + iframe

**能实现不同二级域之间的iframe数据交互**

不同的二级域，父页面a可以操作子页面b的hash，但是子页面不能操作父页面的hash，因此需要在b页面引入与父页面在同一域的孙页面c。

a ------> b ----> c ----> a

1. a页面改变b页面的hash
2. b页面监控到hash改变，然后变c页面的hash
3. c页面监控到hash改变，然后将数据传入a页面

3.window.name + iframe跨域

**能实现不同二级域的单向数据获取**

原理：window.name有一个特殊的表现，在一个框架中，就算其location切换了，这个框架的window.name亦然会存在。但是在父级页面a不能获取非同源子页面b的框架的name.所以这里也需要一个与a页面同源的中间页c。

4. postMessage跨域

**能实现不同二级域的数据交互**

**包括多窗口之间，iframe嵌套 之间的数据交互**

**获取到目标**

H5新方法

otherWindow..postMessage(data,origin)

window. addEventListener('message', function(e) {

alert('data from domain2 ---> ' + e.data);

}, false);

otherWindow：其他窗口的引用。

如 window.frames iframe.contentWindow 等等

origin:

‘\*’任意页面

‘/’同源页面

‘domain.com’具体页面

1. 通过jsonp跨域

**能实现不同二级域之间的通信**

根本原理在于，服务端监听到传过来的回调函数fn

返回一个全局脚本

fn（{...}）

缺点：

必须有一个fn全局函数。

6. 跨域资源共享（CORS） -- 重点

一般而言，不需要前端请求有任何设置，浏览器会帮我们设置好一些需要的

请求头部，但是如果想要传递cookie时，需要设置withCredentials属性（有些浏览器会默认发送cookie）。

简单请求

浏览器直接发送请求

浏览器会在头部信息汇总加一个origin字段，表示发起资源请求的源是什么

服务器会更具origin，决定是否同意请求

同意的话：

返回的头部信息会有

Access-Control-Allow-Origin(请求该资源支持的域)

‘\*’表示所有域都可以请求这个资源

Access-Control-Allow-Credentials

返回是否携带cookie，默认不发送cookie，如果是true，则Access-Control-Allow-Origin不能为‘\*’，Access-Control-Allow-Origin必须设置与请求域一致的域。

但是客户端的cookie也会遵循同源策略，不是上传给非同源的服务器

Access-Control-Expose-Headers（可以被获取的头部）

复杂请求

7、 nginx代理跨域

没有弄，有兴趣的同学可以搞一个，分享出来

8、 nodejs中间件代理跨域

大致原理与Nginx相似， 也是开启一个代理服务器。实现数据转发

在vue的开发环境中可以通过webpack-dev-server开启中间代理

9、 WebSocket协议跨域

H5新协议，实现了浏览器与服务双向通信，同时也允许跨域通讯。

使用socket.io