1. **浏览器跨域问题**

<https://segmentfault.com/a/1190000011145364>

同源策略是指‘协议+域名+端口’三者相同，即便两个不同的域名指向同一个IP，也不属于同源。

同源策略限制以下几种行为：

1. Cookie、LocalStorage和IndexDB无法读取。
2. DOM和JS对象无法获得
3. AJAX请求不能发送

跨域解决方案：

1. 通过jsonp跨域。原理：利用浏览器允许html页面通过不同的标签加载来自不同源的静态资源的特性。实现：动态创建script标签，请求一个带参网址实现跨域请求。





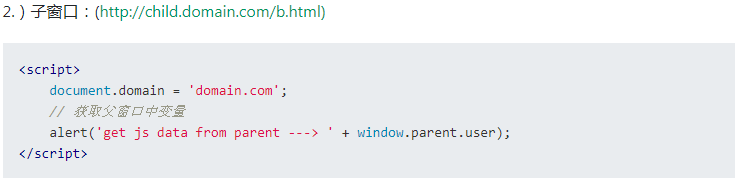


缺点：只能实现get一种请求方式

1. document.domain + iframe 跨域

缺点：此方案只限主域相同，子域不同的应用场景。扩展：什么样的地址是主域相同，子域不同？例：（<http://www.domain.com>和<http://child.domain.com> ）





1. location.hash + iframe 跨域

location.hash属性是一个可读可写的字符串，该字符是URL的锚部分（从#开始的部分）

实现功能：两个不同域的页面可以相互通信。

实现原理：a欲与b相互通信，通过中间页c来实现。三个页面，不同域之间用iframe的location.hash来传值，相同域之间直接用js访问。（window.parent.parent可通过c访问a页面的所有对象）注：a与b属于不同域，a与c属于同域。







1. postMessage 跨域

postMessage是html5 XMLHttpRequest中新增的API。可以解决如下问题：a、页面及打开的新窗口的数据传递b、多窗口之间通信c、页面与嵌套的iframe消息传递d、一上三个场景的跨域数据传递。

用法：postMessage(data,origin)。data：html5规范支持任意基本类型和可复制的对象。但部分浏览器只支持字符串，所以传参时，最好用JSON.stringfy()序列化。origin：‘协议+主机+端口号’，也可以设置为‘\*’，表示可以传递给任意窗口，如果要设置指定和当前窗口同源的话，设置为‘/’。





1. 跨域资源共享CORS

普通跨域请求，只需服务端设置Access-Control-Allow-Origin为true即可。若要带cookie请求，则前后台都需要设置。需要注意的是由于同源策略的限制，所读取的cookie为跨域请求接口所在域的cookie，而非跳转到的当前页面的cookie。若要想实现当前页面的cookie写入。则需在nginx反向代理中设置proxy\_cookie\_domain。

<http://www.ruanyifeng.com/blog/2016/04/cors.html> --CORS跨域详解

允许cookie传递，

前端设置：xhr.withCredentials = true;(用于设置浏览器发送cookie信息)。

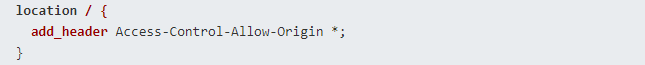
后端设置：

response.setHeader("Access-Control-Allow-Origin", "http://www.domain1.com"); (允许跨域访问的域名：若有端口需写全（协议+域名+端口），若没有端口末尾不用加'/')；

response.setHeader("Access-Control-Allow-Credentials", "true"); (允许前端带认证cookie：启用此项后，上面的域名不能为'\*'，必须指定具体的域名，否则浏览器会提示)

1. nginx代理跨域

a)、nginx配置解决iconfont跨域。浏览器跨域访问js、css、img等常规静态资源被同源策略许可，但iconfont字体文件(eot|otf|ttf|woff|svg)例外，此时可在nginx的静态资源服务器中加入以下配置。



b)、nginx反向代理解决接口跨域



1. Node中间件代理跨域
2. WebSocket协议跨域
3. **关于http请求是否存在跨域的问题小记。**

若从aaa.com上的JS请求bbb.com上某一个接口，这种情况下属于跨域。但是如果直接从浏览器地址栏里，或者用浏览器的扩展（postMan），使用vscode的插件（如REST Client），或者postMan的桌面版，或者CURL的命令行工具去请求bbb.com上的接口时，不会产生跨域。因为这些本地客户端不存在域。

1. **讲讲js数据类型？基本和引用的区别？Symbol和BigInt讲一讲应用场景？**

JS的数据类型有8种。分别是：Null/Boolean/String/Number/Object/undefiend

/BigInt(ES10新增)/Symbol（ES6新增）。

区别：

|  |  |  |
| --- | --- | --- |
| 区别 | 基本类型 | 引用类型 |
| 访问方式不同 | 操作和保存变量实际的值 | 值保存在内存中，JS无法直接访问内存，访问的变量的引用 |
| 存储方式不同 | 存储在栈中 | 引用存储在栈中，值存储在堆中 |

Symbol的应用场景：

1. 使用Symbol作为对象的属性名

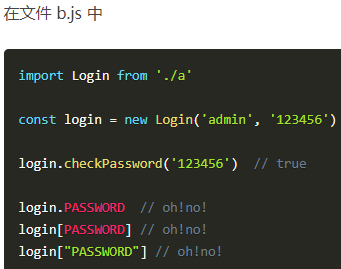
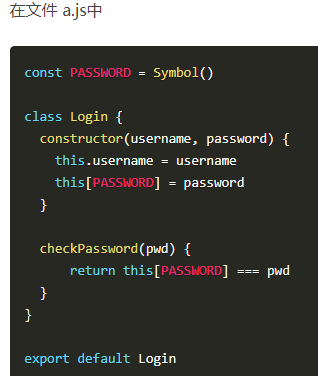
由于Symbol类型的Key无法被Object.keys()和for…in枚举，可以利用这一特性，可以将一些不需要对外访问和操作的属性使用Symbol定义。注意，使用JSON.stringify方法将对象转换成字符串时，Symbol类型也会被排除。同时，如果使用Symbol类型作为对象的key，只能使用’[]’访问，不能使用’.’运算符。

扩展：如何访问一个对象的Symbol类型的key？

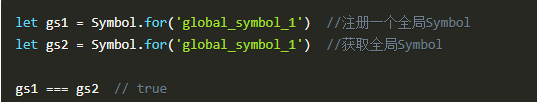
Object.getOwnPropertySymbols(obj) // [Symbol(name)] 或

Reflect.ownKeys(obj) // [Symbol(name),any…]

1. 使用Symbol代替只需校验其唯一性的常量
2. 使用Symbol定义类的私有方法和属性。



注：可以使用Symbol.for()注册或获取一个window间的全局的Symbol实例。



BigInt的应用场景：在JS中，Number类型无法存储超大的整数形式的ID和高精度时间戳，BigInt可以用来存储超过最大安全数（2\*\*52-1）的整数就可以解决这一问题。超过最大安全数的运算问题。

扩展：如何创建BigInt类型？BigInt(123)或 123n。123n !== 123;(不严格相等)123n == 123;（抽象相等）

1. **判断数据类型的方法？instanceof原理?判断空对象？ typof null？typeof NaN？**

**判断数据类型的方法：**

<https://blog.csdn.net/xiaolinlife/article/details/90518849> --四种方法及原理

a) typeof xxx 或 typeof(xxx);该方法可正确判别出Number/undefiend/Object

/Boolean/String/Function/Symbol/BigInt等

b) xxx instanceof Y;Y一定是对象类型，并且大小写不能错。可以检测对象（Date）、数组、函数等。

c) xxx.constructor;constructor是Object类型的原型属性，返回当前对象的构造器（类型函数）。可以检测对象（Date）、数组、函数等。注：该方法是不稳定的，这个主要体现在自定义对象上，当开发者重写prototype后，原有的constructor会丢失，constructor会默认为Object。

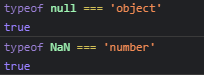
d) Object.prototype.toString.call(xxx);可以检测任何数据类型。

**instanceof原理:**右边变量的prototype在左边变量的原型链上即可返回true。因此，instanceof在查找过程中，会遍历左边变量的原形链，去查找右边变量的prototype；直到左边变量等于null，说明匹配失败，右边变量并非左边变量的实例，返回false；

**判断空对象：**

[**https://zhuanlan.zhihu.com/p/52684391**](https://zhuanlan.zhihu.com/p/52684391) --如何判断空对象

1. 使用for…in遍历对象，在方法体中return true；方法体外 return false;
2. JSON.stringify(obj) === ‘{}’
3. Object.keys(obj)（ES6新增）方法会返回一个obj所有可遍历的属性list，判断list的length属性。
4. Object.getOwnPropertyNames(obj) 方法会返回一个obj所有可遍历的属性list，判断list的length属性。

** --typeof null和typeof NaN无法正确判断类型**

**5、var/let/const 区别？暂时性死区？块级作用域？const a = {}; a.x = 1 能不能修改？**

**var/let/const 区别:**

[**https://zhuanlan.zhihu.com/p/194311856**](https://zhuanlan.zhihu.com/p/194311856) **--区别及特点**

var声明的变量，其作用域在声明该变量的函数中或全局，声明于函数外的作用于全局。存在变量提升，可重新赋值，也可在同作用域中重复声明。

let声明的变量，其作用域在声明该变量的代码块中，变量仅在该代码块内生效，写在顶端的在全局生效。不存在变量提示，会出现暂时性死区，可重新赋值，但不可在同作用域中重复声明。

const声明的常量，其作用域在声明该变量的代码块中，变量仅在该代码块内生效，写在顶端的在全局生效。必须在声明时赋值，并且不允许二次赋值，也不可在同作用域中重复声明

**暂时性死区：**let(const)声明的变量/常量不能在声明代码位置前使用。事实上，在"预处理"时，所有的声明被“注册”，var声明的变量会分配一个初始值（undefined），而let不会，只有执行到声明语句时，let声明的变量才会被初始化，而未初始化的变量不能使用。从let变量作用的块顶部到声明位置被称作“暂存死区”。

**块级作用域：**ECMAScript 6(简称ES6)中新增了块级作用域。块作用域由 { } 包括，if语句和for语句里面的{ }也属于块作用域。在块级作用域中，使用let/const声明的变量/常量只能在该块中使用，不能跨块使用。

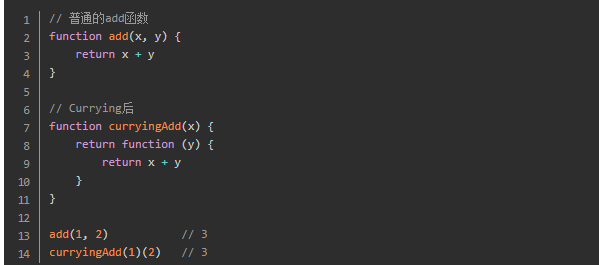
**const a = {}; a.x = 1 能不能修改？**

可以，因为const声明对象、数组等引用类型时，存储的是引用类型的地址，而非数据。不可重新赋值，但可修改对象属性。

1. **说说你对函数式编程的理解？函数柯里化的理解？平时的使用场景？（未完）**

**函数式编程的理解：**以函数为单位处理各种业务逻辑，函数既可以当做参数传来传去，也可以当做返回值。它属于结构化编程的一种，主要思想是把运算过程写成一系列的函数嵌套调用。对函数外部环境没有任何依赖。

**函数柯里化的理解：**把接收多个参数的函数变成一个接收单一参数（原函数的第一个参数）的函数，并返回接收剩余参数的函数而且返回结果的新函数的技术。

**7、在for循环中使用setTimeout的四种解决方案**

1、利用闭包，创建立即执行函数；

2、利用setTimeout方法的第三个参数，需要回调函数接收

3、利用let声明变量会产生块级作用域

4、拆分函数，将setTimeout方法单独拆出来声明。

****

[..\学习小记\code.js](../学习小记/code.js) –代码

1. **解释为什么[]==![]？**

①、根据运算符优先级 ，！ 的优先级是大于 == 的，所以先会执行 ![]

！可将变量转换成boolean类型，null、undefined、NaN以及空字符串('')取反都为true，其余都为false。

所以 ! [] 运算后的结果就是 false

也就是 [] == ! [] 相当于 [] == false

②、根据上面提到的规则（如果有一个操作数是布尔值，则在比较相等性之前先将其转换为数值——false转换为0，而true转换为1），则需要把 false 转成 0

也就是 [] == ! [] 相当于 [] == false 相当于 [] == 0

③、根据上面提到的规则（如果一个操作数是对象，另一个操作数不是，则调用对象的valueOf()方法，用得到的基本类型值按照前面的规则进行比较，如果对象没有valueOf()方法，则调用 toString()）

而对于空数组，[].toString() ->  '' (返回的是空字符串)

也就是  [] == 0 相当于 '' == 0

④、根据上面提到的规则（如果一个操作数是字符串，另一个操作数是数值，在比较相等性之前先将字符串转换为数值）

Number('') -> 返回的是 0

相当于 0 == 0 自然就返回 true了

总结一下：[] == ! []   ->   [] == false  ->  [] == 0  ->   '' == 0   ->  0 == 0   ->  true

那么对于 {} == !{} 也是同理的。关键在于  {}.toString() -> Number("[object Object]")->  NaN(返回的是NaN)

根据上面的规则（如果有一个操作数是NaN，则相等操作符返回 false）

总结一下：{} == ! {}   ->   {} == false  ->  {} == 0  -> "[object Object]" == 0 ->  NaN == 0    ->  false

**9、什么是作用域？**

JS作用域是可访问变量、对象和函数的集合。分为三类：全局作用域、函数作用域和块级作用域。

变量在函数内声明为局部变量，在函数执行时创建，在函数执行完成后自动销毁；变量在函数内部没有使用var关键字声明或在函数外部使用var关键字声明的为全局变量，全局变量会在页面关闭时销毁。使用let关键字声明的变量会产生块级作用域。

1. **什么是作用域链？**

一般情况下，变量取值会到创建该变量的函数的作用域中获取值。但是如果在当前作用域中没有找到该变量，则会一直向上查找知道找到全局作用域，这一查找过程形成的链就叫做作用域链。

1. **原型及原型链**

每个对象都有一个\_\_proto\_\_指向该对象的原型对象；

每个原型对象都可以通过constructor找到该对象的构造函数，构造函数可以通过prototype属性找到原型

所有函数都可以通过\_\_proto\_\_找到Function

所有对象都可以通过\_\_proto\_\_找到Object

对象之间通过\_\_proto\_\_连接起来，这样称之为原型链。当前对象不存在的属性，可通过原型链一层层向上找，直到找到Object对象，最上层为null。

1. **事件循环**

同步任务和异步任务分别进入不同的执行环境。同步任务进入主线程，异步任务被推入到任务队列。主线程的任务执行完毕后，会去任务队列中读取对应的任务，推入主线程中。这样不断重复的执行任务的过程就是事件循环（event loop）。

主线程：也就是JS引擎执行的线程，这个线程只有一个，页面渲染和函数处理都在这个线程上。

工作线程：也叫幕后线程，这个线程存在于浏览器或JS引擎中，与主线程是分开的，用于处理网络请求或文件读取等异步任务。

任务队列：异步任务会通过先进先出的机制存储在任务队列中。

宏任务（Macro Task）：同步代码、setTimeout回调、setInterval回调、IO、UI交互事件、postmessage、MessgaeChannel。

微任务（Micro Task）：Promise状态改变后的回调（then函数执行，如果此时状态没有改变，回调只会被缓存，只有当状态改变，缓存的回调才会被丢到任务队列）、Mutation Observer回调函数、queueMicroTask回调函数（新增的API）。

宏任务会被丢到下一次循环，并且宏任务队列一次只执行一个任务。

微任务会被丢到本次循环，微任务执行直到清空微任务执行队列。也就是说，微任务执行如果返回一个微任务，则该任务将永远不会停止。

事件循环执行顺序：1、执行同步代码；2、执行完所有同步代码且执行栈为空，判断是否有微任务需要执行；3、执行所有微任务直至执行队列为空；4、是否有必要渲染页面；5、执行一个宏任务。