**01、Javascript 的几种数据类型与区别**

**数据类型：**

简单数据类型：

Number、String、Boolean、Symbol，BIgInt、Undefined、Null；

栈内存(Stack)；

没有属性和方法，但通常浏览器会自行为我们创建包装类，再调用包装类的属性和方法。

引用数据类型：

Object；

堆内存(Heap)；

有属性和方法。

**类型转化：**

在 js 中，我们可以先声明变量但不为变量赋值，那么变量的具体类型通常需要等到程序运行时才能知道，虽然变量的数据类型是不确定的，但各种运算符对数据类型是有要求的，于是就会触发类型转换机制

**显示转化**

Number()

String()

Boolean()

parseInt()

**隐式转化**

比较运算符（==、!=、<、>）if、else 需要布尔值的地方

NaN、0、null、''、undefined、false 转化为 false，其余皆为 true。

算数运算（+、-、\*、/、%）

**02、js 数组常用的方法有哪些**

**操作方法**

增：unshift、push、splice、concat(不会对原数组产生影响)

删：shift、pop、splice、slice(不会对原数组产生影响)

查：indexOf、includes、find、findIndex

改：splice(起始位置、删除元素个数、插入的元素)

**排序方法**

reverse

sort

**转化方法**

join

**迭代方法**

some

every

filter

map

forEach

**03、JS 字符串常用的方法有哪些？**

字符串的特点是一旦创建，就不可更改。所以当我们对字符串进行操作时，实际上都是新建一个副本，再对副本进行操作。

基本操作

增

除了常用的 + 、${} 外，我们也可以使用 concat 来进行字符串的拼接。

删

"abcd".slice(1,2) //b

"abcd".substring(1,2) //b

"abcd".substr(1,2) //bc

查

charAt()

indexOf()

includes()

startWith("abc") // 是否以 "abc" 开头，返回布尔值

改

trim(), trimLeft(), trimRight()

toUpperCase(), toLowerCase()

padStart(), padEnd()

转换方法

字符串转化为数组：

"12+23+34".split('+') // [12,34,56]

Array.from("123") // ['1','2','3']

模板匹配方法：

search()

返回number："hello world".search("el") // 1

match()

返回数组: "hello world".match(/el/)[0] // el

replace()

"hello world".replace("hello","hi") // hi world

**04、== 和 === 的区别，分别在什么情况下使用**

== 是一种弱类型比较，不同类型之间进行比较，通常会先进行隐式类型转换，然后在进行比较：

基本类型之间进行比较：

先转换为数值类型再进行对比

0 == 'false' // true 此处会把字符串 false 转化为数值类型，得到 NaN。

基本数据类型与引用类型之间进行比较：

先得到引用类型的返回值，再进行比较

let obj = {valueOf:function(){return 1}}

let result1 = (obj == 1); // true

引用类型之间进行比较：

比较内存中的地址是否一致

null 与 undefined 比较结果为 true

存在 NaN 则返回 false

=== 比较中，不会进行类型转换，它在数据类型一致的情况下，才会进行值的比较。

**05、深拷贝和浅拷贝的区别？如何实现一个深拷贝？**

基本类型赋值时，赋的时数据，所以两者互不干扰

当对象的属性是对象时：

浅拷贝拷贝的是地址，于是更改副本，源数据也会发生改变

浅拷贝会将变量存在栈区的值复制给别人

基本类型在栈区存储的是值，而引用类型在栈区存储的是地址

Object.assign()、slice()、concat()、拓展运算符(...)

深拷贝拷贝的是数据，是两个互不相干扰的数据

\_.cloneDeep(拷贝对象)

jQuery.extend(true, {}, obj1)

JSON.stringfy()

会忽略undefined、symbol、函数

递归循环

**06、说说你对闭包的理解？闭包使用场景？**

概念：

闭包指有权访问另一个函数作用域中的局部变量函数。声明在一个函数内部的局部变量函数称为闭包函数。而且内部函数始终可以访问在其外部函数声明的变量和参数，即使在其外部函数被返回（寿命终结）之后。

特点：

让外部访问函数内部变量成为可能；

局部变量会常驻在内存中；

避免造成全局变量污染；

造成内存污染（有一块内存空间长期被占用，无法释放）

**07、说说你对作用域链的理解**

作用域：即变量及其能够被访问的区域的集合（变量作用域也称作上下文）。作用域决定我们的数据资源的可见性。

分类

全局作用域：不在函数或块级作用域定义的变量，都是在全局作用域下。

函数作用域：也称作局部作用域，在函数内部定义的变量，只能在该函数内部被访问得到，不能在函数外的其他区域所访问（闭包除外）。

块级作用域：ES6 中新增的 let 和 const 关键字，在大括号 {} 内部定义的变量，只能在大括号内部被访问，而不能再大括号外部被访问。（var 定义的变量例外：变量提升）

词法作用域：也称静态作用域。即变量在创建时就已经确定好了其作用域。

作用域链：

当函数内部查找不到所需变量时，会向上一层作用域进行查找，直到到达全局作用域下，若还找不到，则报错 undefined。

**08、JavaScript 原型，原型链 ? 有什么特点？**

原型：

prototype 存储着原型的地址，原型是一堆共有属性的集合

所有函数都有一个 prototype（原型）属性，属性值是一个普通对象；

所有的应用类型都有一个 \_\_proto\_\_ （隐式原型）属性，属性值是一个普通对象；

所有引用类型的 \_\_proto\_\_ 属性都指向其构造函数的 prototype 属性。

原型链：

原型对象也可以拥有原型，并从中继承方法和属性，一层一层以此类推，这样一种联系称为原型链，原型链的顶端值为 null

总结：

一切对象都继承于 Object 对象，Object 对象直接继承根源对象 null；

一切的函数对象（包括 Object 对象），都继承于 Function 对象；

Object 对象由 Function 构造；

Function 对象的 \_\_proto\_\_ 会指向自己的原型对象，最终还是继承自 Object 对象。

**09、Javascript 如何实现继承**

概念：继承是面向对象软件技术里的一个概念，如果一个类B继承自类A，那么A称为父类（超类），B称为子类。

继承的优点：

继承使得子类拥有父类的各种属性和方法，而不要需要再次编程，同时，子类可以重新定义某些属性，重写某些方法，即覆盖父类的方法，使其获得与夫类别不同的功能

常见的继承方式：

原型链继承 ：Child1.prototype = new Parent();

构造函数、原型对象、实例对象三者联系

可以继承父类的属性和方法，包括原型身上---造成数据污染

构造函数继承：Parent.call(this)

可以继承父类身上的普通属性和方法，但不能继承父类原型上的属性和方法

父类的引用属性不会被共享，不会造成数据污染

组合式继承

将上面两种方式组合起来，使得方式一和方式而的问题都解决了，但会调用两次超类，造成多构造一次的性能开销。

原型式继承

使用 Object.create 方法实现普通对象的继承

Object.create 是浅拷贝的方式，多个实例的引用类型属性指向相同的内存空间，存在数据纂改的可能性

寄生式继承

在原型式继承的基础上进行优化，利用这个浅拷贝的能力再进行增强，添加一些方法

具有和原型式继承相同的缺点

寄生组合式继承

寄生组合式继承，借助解决普通对象的继承问题的 Object.create 方法，在前面几种继承方式的优缺点基础上进行改造，这也是所有继承方式里面相对较优的继承方式

es6 中继承的关键字 extends 也是采用的寄生组合式继承

**10、谈谈 this 对象的理解**

概念：

this 是在函数运行时自动创建的一个内部对象，只能在函数内部使用，总是指向它的调用对象。在不同的使用场景下，有不同的绑定方式：

绑定方式：

默认绑定：

在全局环境中定义并使用一个函数，那么该 this 指向 window

隐式绑定：

当函数作为某个对象的方法调用时， this 始终指向它的上一级运行对象

注意：若只是声明，没有调用则不作数：

在下述例子中，this 指向 window

var o = {

a:10,

b:{

a:12,

fn:function(){

console.log(this.a); //undefined

console.log(this); //window

}

}

}

var j = o.b.fn;

j(); // 在全局环境下调用

显示绑定：

使用 call() , bind(), apply() 显示指定指向的对象

new 绑定：

this 指向使用 new 关键字创建的实例对象

注意当函数有返回值时：

返回值为基本类型：则 this 指向示例对象

返回值为引用类型：则 this 指向该返回对象

注意：

箭头函数没有 this 和 arguments ，其 this 始终指向window

优先级：

new 绑定 > 显示绑定 > 隐式绑定 > 默认绑定

**11、JS 中的执行上下文和执行栈是什么？**

执行上下文：执行上下文是一种对 js 代码运行环境的抽象概念，只要有 js 代码执行，那么就一定存在执行上下文

分类：

全局执行上下文：在window环境下

函数执行上下文：存在无数个，只有在函数被调用的时候才会创建，每次调用函数时都会创建一个新的执行上下文。

Eval 函数执行上下文：指在eval函数中的代码，很少用且不建议使用

声明周期：创建阶段、执行阶段、回收阶段

执行栈：也称为调用栈，具有 LIFO (后进先出) 的结构，用于存储代码在执行期间创建的所有执行上下文

当引擎碰上一个函数时，会自动为该函数创建一个函数执行上下文，然后将这个执行上下文压入栈中，当该函数执行完毕，弹出该函数的执行上下文。

**12、typeof 和 instanceof 的区别**

区别：

typeof 返回一个数据类型，是一个字符串；instanceof 返回一个布尔值

instanceof 用于判断构造函数的prototype属性是否在子类对象的原型身上出现。 可以准确地判断复杂地引用数据类型，但不能判断基本数据类型

type可以判断基本数据类型，null除外。但引用数据类型只能识别function，其余皆返回object

**Object.prototype.toString.call() 方法：统一返回 [Object Xxx] 字符串**

Object.prototype.toString({}) // "[object Object]"

Object.prototype.toString.call({}) // 同上结果，加上call也ok

Object.prototype.toString.call(1) // "[object Number]"

Object.prototype.toString.call('1') // "[object String]"

Object.prototype.toString.call(true) // "[object Boolean]"

Object.prototype.toString.call(function(){}) // "[object Function]"

Object.prototype.toString.call(null) //"[object Null]"

Object.prototype.toString.call(undefined) //"[object Undefined]"

Object.prototype.toString.call(/123/g) //"[object RegExp]"

Object.prototype.toString.call(new Date()) //"[object Date]"

Object.prototype.toString.call([]) //"[object Array]"

Object.prototype.toString.call(document) //"[object HTMLDocument]"

Object.prototype.toString.call(window) //"[object Window]"

封装一个全局通用的数据类型判断方法

function getType(obj){

let type = typeof obj;

if (type !== "object") { // 先进行typeof判断，如果是基础数据类型，直接返回

return type;

}

// 对于typeof返回结果是object的，再进行如下的判断，正则返回结果

return Object.prototype.toString.call(obj).replace(/^\[object (\S+)\]$/, '$1');

}

**13、说说new操作符具体干了什么**

自动创建一个空对象

自动为空对象关联原型，原型地址为X.prototype

自动将this作为关键字运行构造函数

自动 return this

**14、事件委托**

<https://github.com/febobo/web-interview/issues/66>

把一个元素的响应事件绑定到另外一个函数身上

事件流：捕获阶段--》目标阶段--》冒泡阶段，事件委托于冒泡阶段完成

减少整个页面所需的内存，提升整体性能

适合事件委托的事件有：click，mousedown，mouseup，keydown，keyup，keypress

focus、blur 这些事件没有事件冒泡机制，所以无法进行委托绑定事件

mousemove、mouseout 这样的事件，虽然有事件冒泡，但是只能不断通过位置去计算定位，对性能消耗高，因此也是不适合于事件委托的

**15、ajax的原理**

异步的 js 和 xml，是一种创建交互式网页应用的网页开发技术，可以再不重新刷新整个页面的情况下，与服务器交换数据，并更新部分网页

简单来说是通过 XMLHttpRequest 对象来向服务器发起异步请求，从服务器获得数据，最后使用js来操作dom从而更新页面

实现：

创建xhr对象

调用xhr对象的open方法

监听服务器端的通信状态

调用xhr的send方法

onreadystatechange 监听服务器端的通信状态：

0 ---- 创建xhr对象

1 ---- 调用open方法

2 ---- 调用send方法

3 ---- 服务器端返回的数据开始下载

4 ---- 数据下载完成

**16、bind、call、apply 区别？如何实现一个bind?**

三者都可以用于改变函数执行的上下文，也就是this指向；

三者接收的第一个参数都是this要指向的对象，

三者都可以传参，apply的第二个参数是数组，call是列表，且是一次性传入；而 bind 可以分多次传入

apply 和 call 改变 this 指向后会立即执行，并且是一次性的；而 bind 不会立即执行，取而代之的是返回一个改变 this 指向后的函数（永久性）

**17、正则表达式**

<https://github.com/febobo/web-interview/issues/72>

正则表达式的设计思想是用一种描述性的语言去定义一个规则，凡是符合规则的字符串，就认为是匹配的，否则该字符串就是不合法的







**18、事件循环**

JavaScript 中的任务分为同步任务和异步任务，通常来说，同步任务会直接压入主线程，而异步任务则进入队列里面；当主线程全部执行完毕后，会去队列里面找异步任务并且压入主线程中，整个过程不断循环，这种运行机制称作事件循环；

异步任务分为宏任务和微任务，所在的队列具有FIFO的结构特点，微任务的执行循序优于宏任务

宏任务：settimeout、script、UI事件、等

微任务：Promise.then、Proxy代理、process.nextTick、MutationObserver等

async & await

async用于声明一个异步方法，而await用于等待异步方法执行

async函数返回一个 promise 对象

正常情况下，await 命令后面跟着一个 promise 对象，如没有，则返回对应的值

await 总是会阻塞后面的代码（压入微任务队列）

**19、DOM 常见操作有哪些**

概述：

文档对象模型是 HTML 和 XML 文档的编程接口，简单来说，dom 就是 js 操作 html 的 api

dom 常见操作：

创建节点

createElement、createTextNode、createAttribute、createDocumentFragment（存储临时节点，插入的是子孙节点）

添加节点

innerHTML、appendChild、insertBefore(newElement, referenceElement)、setAttribute(div.setAttribute('class', 'white');//第一个参数属性名，第二个参数属性值。)（添加一个属性节点）

查询节点

querySelector

更新节点

innerHTML、innerText、textContent

删除节点

removeChild

**20、说说你对 BOM 的理解，常见的 BOM 对象有哪些？**

概述：

bom 作用是与浏览器进行交互，比如页面的后退、前进、刷新，浏览器窗口发生变化、滚动条的滚动、获取客户的一些信息：浏览器版本、屏幕分辨率等等

常见的 bom 对象：

window：核心对象：即是浏览器窗口的一个接口、又是全局对象

控制窗口常用方法：

location：获取url相关的参数moveBy(x,y)：从当前位置水平移动窗体x个像素，垂直移动窗体y个像素，x为负数，将向左移动窗体，y为负数，将向上移动窗体

moveTo(x,y)：移动窗体左上角到相对于屏幕左上角的(x,y)点

resizeBy(w,h)：相对窗体当前的大小，宽度调整w个像素，高度调整h个像素。如果参数为负值，将缩小窗体，反之扩大窗体

resizeTo(w,h)：把窗体宽度调整为w个像素，高度调整为h个像素

scrollTo(x,y)：如果有滚动条，将横向滚动条移动到相对于窗体宽度为x个像素的位置，将纵向滚动条移动到相对于窗体高度为y个像素的位置

scrollBy(x,y)： 如果有滚动条，将横向滚动条向左移动x个像素，将纵向滚动条向下移动y个像素

window.open() 既可以导航到一个特定的url，也可以打开一个新的浏览器窗口。如果 window.open() 传递了第二个参数，且该参数是已有窗口或者框架的名称，那么就会在目标窗口加载第一个参数指定的URL

window.close() 仅用于通过 window.open() 打开的窗口

location：获取url各部分属性

protocol：使用协议

port：端口号

href：完整url

location.reload()，此方法可以重新刷新当前页面。这个方法会根据最有效的方式刷新页面，如果页面自上一次请求以来没有改变过，页面就会从浏览器缓存中重新加载，如果要强制从服务器中重新加载，传递一个参数true即可

navigator：获取浏览器相关的属性

screen：客户端能力信息，如像素、高度、宽度等。

history：url访问历史

history.go()：跳转到指定页面

history.forward()：向前跳转一个页面

history.back()：向后跳转一个页面

history.length：获取历史记录

**21、js 的本地存储方式有哪些？区别以及应用场景**

方式：

cookie

可设置活性（存活一段时间）

4k左右

localStorage：

h5新方法、ie8及以上都支持

永久存储

5M 左右，受浏览器厂商影响

受同源策略影响

sessionStorage

跟localStorage操作基本相同

会话关闭即失效

indexedDB

存储量理论上没有上限

所有操作都是异步的，相比ls同步操作性能更高，尤其是大量数据时

支持存储js对象

数据库

操作非常繁琐

区别（cookie、ls、ss）：

存储大小：

cookie数据不能超过4k，ls和ss可达5M

有效时间：

sessionStorage：会话关闭即失效

cookie：可设置存活时间

localStorage：永久存活

数据与服务器之间的交互方式：

ls 和 ss 存储的数据仅在客户端保存

cookie 的数据会自动的传递到服务器，服务器端也可以写cookie到客户端

应用场景：

敏感账号一次性登录：sessionStorage

标记用户、跟踪用户行为：cookie

长期保存数据在本地：localStorage

存储大量数据：indexedDB

**22、尾递归的理解？应用场景**

<https://github.com/febobo/web-interview/issues/77>

递归：在函数的定义中调用行数自身的方法

尾递归：在尾部调用函数自身

对于优化前返回的是n \* factorial(n - 1)，这里会先执行后面的函数，而前面的n \* 操作则需要保存下来，这样就需要保存整个函数的执行上下文。直到最后一次n === 1，然后再从堆栈中取出上一次执行上下文的n，进行乘法操作，所以最后factorial(5)等价于 5 \* factorial(4) \* factorial(3) \* factorial(2) \* factorial(1)。优化后直接返回函数，所以无需保存函数的执行上下文，进入下一个函数后垃圾清理机制会自动清理掉上个函数的执行上下文。

应用场景：

数组求和

优化斐波那契额数列

数组扁平化

数组对象格式化

**23、函数式编程的理解**

概述：

是一种编程范式，主要的编程范式有：命令式编程、声明式编程、函数式编程

同命令式编程相比，函数式编程更强调的是程序的执行结果而非执行过程（封装）

简单来说就是把过程逻辑写成函数，定义好输入参数，只关心它的输出结果

内容：

纯函数

高阶函数

柯里化

组合与管道

优缺点：

优点

更好的管理状态：因为它的宗旨是无状态，或者说更少的状态，能最大化的减少这些未知、优化代码、减少出错情况

更简单的复用：固定输入->固定输出，没有其他外部变量影响，并且无副作用。这样代码复用时，完全不需要考虑它的内部实现和外部影响

更优雅的组合：往大的说，网页是由各个组件组成的。往小的说，一个函数也可能是由多个小函数组成的。更强的复用性，带来更强大的组合性

隐性好处。减少代码量，提高维护性

缺点：

性能：函数式编程相对于指令式编程，性能绝对是一个短板，因为它往往会对一个方法进行过度包装，从而产生上下文切换的性能开销

资源占用：在 JS 中为了实现对象状态的不可变，往往会创建新的对象，因此，它对垃圾回收所产生的压力远远超过其他编程方式

递归陷阱：在函数式编程中，为了实现迭代，通常会采用递归操作

**24、js 中如何实现函数缓存？函数缓存有哪些应用场景？**

是什么？

函数缓存：就是将函数的运行结果进行缓存

本质是以空间换时间

常用于缓存数据计算结果和缓存对象

如何实现？

依靠闭包、柯里化、高阶函数完成

将参数和对应的函数运行结果存储于一个对象中，调用时判断该对象中的某一个参数是否存在，存在则返回结果；不存在就返回计算结果，并对数据进行缓存

应用场景？

常用于昂贵的函数调用，如计算过程复杂的函数

一定范围内，高度重复输入的函数

重复输入值的递归函数

对于纯函数来说，输入特定的参数会返回特点的结果

**25、js 精度丢失？如何解决**

浮点数

计算机存储64位双精度浮点数，需要先把十进制数转化为二进制

转化为二进制数时出现无限循环数，

舍0进1造成误差

解决办法

对于存储单个小数：

使用toprecision凑整，在使用parseFloat转化为数字

对于四则运算：

将小数转化为整数，再进行四则运算，而后转化回小数

使用第三方库：Math.js、BigDecimal.js

**26、防抖与节流**

是什么

防抖：回城被打断

节流：技能冷却

区别

相同点：

都可以使用定时器来完成

目的都是降低回调执行频率，节省计算资源

不同点：

防抖在一段连续操作结束后，处理回调（先清除定时器，再调用定时器）；节流再一段连续的操作中，只执行一次回调

防抖关注的是一段时间内的连续触发事件，只执行最后一次

节流是在一段时间内只执行一次

应用场景

防抖：

搜索框的输入

提交按钮事件

scroll滚动

浏览器窗口的resize事件

节流：

dom元素的拖拽

射击游戏

计算鼠标的距离

**27、判断一个元素是否在可视区**

可视区：

浏览页面肉眼可见的区域

方法：

offsetTop+scrollTop

getBoundingClientRect(top>0,top<innerHTML)

intersection observer 重叠观察者

**28、大文件如何做断点续传**

是什么？

影响用户体验的因素：

服务器处理数据的能力

请求超时

网络波动

对大文件单独处理：分片上传、断点续传

分片上传：文件按照一定的大小分割成多个数据块进行上传，上传成功后，再有服务器进行汇总，还原文件

断点续传：指在分片上传的过程中，将下载或上传任务人为地划分为几个部分（打断分片上传），而需要继续上传时只需要从被打断的部分开始上传，而不需要重新上传

服务器端返回，告知从哪开始

浏览器自行处理（上传切片后及时将其从数据块集合中删除，再存储到本地中）

实现思路？

拿到文件，保存文件的唯一标识（用于判断文件的上传速度）

切割文件，将切割后的数据块保存到一个集合里

分片上传文件，修改唯一标识。上传后及时将其从数据块集合中删除，再存储到本地中

上传完毕后，服务器将数据块进行拼接，得到一个完整的资源文件

若需要实现断点（暂停切片上传）：xhr.abort()可以实现

应用场景？

上传过程中刷新页面怎么办

如何进行并行上传

切片什么时候按数量切，什么时候按大小切

如何结合 Web Worker 处理大文件上传

如何实现秒传

**29、如何实现上拉加载、下拉刷新？**

实现原理

这两种交互常出现在移动端中（PC的分页）

依赖于用户的交互

本质是页面触及顶部或者是底部时触发的动作

实现方法

开源社区有很多优秀的解决方案：iscroll、better-scroll等等的库

**30、什么是单点登录SSO？如何实现**

是什么？

单点登录是指在多个应用系统中，用户只需要登陆一次就可以访问所有互相信任的应用系统

sso一般需要一个独立的认证中心，子系统的登陆均通过passport，而子系统本身不再参与登录操作

将登录从系统中抽离出来，简化了用户在个业务系统之间反复登录的繁琐操作

如何实现？

同域名下的单点登录：tieba.baidu.com 和 map.baidu.com，它们都建立在 baidu.com这个主域名之下

将 cookie 的 domain 属性设置为父域的域名（主域名）

将 cookie 的 path 属性设置为根路径

将 sessionID（Token）保存到父域里

在不同域名下：将登录抽离出来成为一个独立的系统 SSO

将 sessionID（Token）保存到 localStorage 中

向后端请求时，主动将localStorage的数据传递给后端

登录成功后，后端将 sessionID（Token）放在响应体里传递给前端

通过 iframe + postMessage() 将同一份 token 写入多个域下的localStorage中

// 获取 token

var token = result.data.token;

// 动态创建一个不可见的iframe，在iframe中加载一个跨域HTML

var iframe = document.createElement("iframe");

iframe.src = "http://app1.com/localstorage.html";

document.body.append(iframe);

// 使用postMessage()方法将token传递给iframe

setTimeout(function () {

iframe.contentWindow.postMessage(token, "http://app1.com");

}, 4000);

setTimeout(function () {

iframe.remove();

}, 6000);

// 在这个iframe所加载的HTML中绑定一个事件监听器，当事件被触发时，把接收到的token数据写入localStorage

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

localStorage.setItem('token', event.data)

}, false);

流程？

用户访问系统1的受保护资源，系统1发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数

sso认证中心发现用户未登录，将用户引导至登录页面

用户输入用户名密码提交登录申请

sso认证中心校验用户信息，创建用户与sso认证中心之间的会话，称为全局会话，同时创建授权令牌

sso认证中心带着令牌跳转会最初的请求地址（系统1）

系统1拿到令牌，去sso认证中心校验令牌是否有效

sso认证中心校验令牌，返回有效，注册系统1

系统1使用该令牌创建与用户的会话，称为局部会话，返回受保护资源

用户访问系统2的受保护资源

系统2发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数

sso认证中心发现用户已登录，跳转回系统2的地址，并附上令牌

系统2拿到令牌，去sso认证中心校验令牌是否有效

sso认证中心校验令牌，返回有效，注册系统2

系统2使用该令牌创建与用户的局部会话，返回受保护资源

**31、web常见的攻击方式有哪些？如何防御？**

是什么？

指针对用户上网，或者对网站服务器等设备进行攻击的行为

植入恶意代码、修改网站权限、获取网站用户隐私信息等等

XSS？跨站脚本攻击：攻击者将恶意代码植入到用户使用的页面中

反射性

伪装后的url（包含恶意代码）诱惑用户点击请求

存储型

恶意代码提交到数据库中

DOM型

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞

CSRF？跨站请求伪造

攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求

整个过程攻击者并不能获取到受害者的登录凭证，仅仅是“冒用”

CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性

防止csrf常用方案如下：

阻止不明外域的访问

同源检测

Samesite Cookie

提交时要求附加本域才能获取的信息

CSRF Token

双重Cookie验证

SQL注入: 是通过将恶意的 Sql 查询或添加语句插入到应用的输入参数中，再在后台 Sql 服务器上解析执行进行的攻击

流程如下所示：

找出SQL漏洞的注入点

判断数据库的类型以及版本

猜解用户名和密码

利用工具查找Web后台管理入口

入侵和破坏

预防方式如下：

严格检查输入变量的类型和格式

过滤和转义特殊字符

对访问数据库的Web应用程序采用Web应用防火墙

**32、垃圾回收**

垃圾是什么？

如果一段内存空间没有被其他变量所引用，则称之为垃圾。注意环引用也是垃圾

如何捡垃圾？

标记清除法

策略：

垃圾收集器运行时会给内存中的所有变量都加上一个标记，假设全为垃圾0；

从各个根对象开始遍历，把不是垃圾的对象节点改为1

清除所有标记为0的垃圾，销毁并回收内存空间

最后，把所有内存中的对象标记修改为0，等待下一轮垃圾回收、

优点：实现比较简单，打标记无非也就两种

缺点：

内存碎片化：空闲内存不连续，新建对象时可能找不到合适的块

分配速度慢：新建对象时，需要从头开始遍历，查看内存空间

解决办法：使用标记整理算法：在标记清除阶段结束后，标记整理算法会把活着的对象向内存的一侧移动，整理掉边界的内存

引用计数法：

记录引用对象被引用的次数，若引用数为0，则垃圾收集器在运行时会清理掉引用次数为0所占的内存空间

优点：

对比标记清除算法来说更加清晰，尤其是引用数为0时，会被立马清除掉

标记清除算法则需要每隔一段时间进行一次遍历

缺点：

需要一个计数器，由于我们不知道引用数的上限，计数器器需要占很大的内存空间

无法解决环引用的问题

特殊情况？

全局遍历下声明的变量都不是垃圾

全局作用域下，某变量=null，不是垃圾，有可能被dom引用。