# JavaScript复习资料

1. **怎么判断变量类型是否为引用类型**
2. Variant !== null && (typeof Variant === ‘object’ || typeof Variant === ‘function’ )
3. Variant instanceof Object
4. Object.prototype.toString.call(Variant).slice(8, -1)
5. **检查是否是数组**
6. Object.prototype.toString.call().slice(8, -1)
7. Instanceof
8. Map、 slice等数组方法进行能力检测
9. Array.isArray
10. **Typeof null === ‘object’**

对于 null 来说，很多人会认为他是个对象类型，其实这是错误的。虽然 typeof null 会输出 object，但是这只是 JS 存在的一个悠久 Bug。在 JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，000 开头代表是对象，然而 null 表示为全零，所以将它错误的判断为 object 。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却是一直流传下来

1. **== 与 === 的区别**

**==：**

如果对比双方的类型不一样的话，就会进行类型转换，null和undefined 除非这两个之间对比否则原始类型都不相同，对象转换为String和 Number，String 和 Boolean 转换为 Number，

**===：**

先判断类型，再判断值

1. **深拷贝和浅拷贝**

**浅拷贝：**

拷贝所有属性到新的对象，如果是对象那么拷贝的是地址，比如Object.assign、展开运算符...

**深拷贝：**

深拷贝遇到对象的属性值如果还是一个对象那么会重新创建一个全新的对象，对象的属性名相同，属性值是再一次的“深拷贝”。常见的就是JSON.parse(JSON.stringify())

**JSON.parse(JSON.stringify())：**

1. 无法对undefined和Symbol类型处理
2. 无法处理序列化函数
3. 无法解决循环引用的问题
4. **This值的指向**
5. 全局环境的this => window，非箭头函数的普通函数调用 this => window
6. 箭头函数，是在定义的函数的环境上下文的this值
7. 函数被作为对象属性调用，this => 该对象
8. Call、Apply、Bind 函数
9. **箭头函数**
10. 箭头函数体中的 this 对象，是定义函数时的对象，而不是使用函数时的对象
11. 没有 this、super、arguments 和 new.target 绑定（不能new）
12. 在class关键字里，绑定的箭头函数是在实例对象上的，而不是原型上的
13. 无法改变this的值，call，bind，apply无效
14. **闭包**

**形成闭包的条件：**

1. 函数A内能够定义函数B
2. 函数B 能访问函数A的变量
3. 函数调用

**闭包的三个特性：**

1. JS在函数内部运行定义新的函数
2. 在函数内部可以访问父亲函数的变量
3. 函数作用第一公民，可以作为返回值（也可以直接向外暴露）

**闭包的使用场景：**

1. 模块化，解决的问题有私有变量的隐藏，可以避免污染全局变量
2. 柯里化、节流和防抖
3. 一些回调函数、ajax、定时器、事件绑定方法

**闭包的几个问题：**

1. 代码难以维护，函数可以一层嵌套一层，内函数可以访问祖先函数的变量
2. 内存泄漏，需要及时的去清理，一些变量一直和闭包共存亡，占用内存空间
3. **arguments**

arguments 是一个对应于传递给函数的参数的类数组对象

**几个属性：**

1. callee JS当前执行到该段代码时，正在执行的函数，匿名函数自调用
2. length 函数的参数的长度
3. 生成迭代对象属性，Symbol.iterator
4. **内存泄漏并且如何解决**

**垃圾回收机制：**

找出那些不再使用的变量，然后释放其占用的内存。为此，垃圾收集器会按照固定的时间间隔（或代码 执行中预定的收集时间），周期性的执行这一操作

**几种方法：**

1. 标记清除。当变量进入环境（例如，在函数中声明一个变量）时，将这个变量标记为 “进入环境” 。从逻辑上讲， 永远不能释放进入环境的变量所占用的内存，因为我们在这个环境中可能随时会用到它们。当变量离开 环境时，则将其标记为 “离开环境
2. 引用计数。跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型值赋给该变量时，这个值的引用次 数就是 1。如果同一个值又被赋值给另一个变量，则引用次数加 1。相反，如果包含对这个值的引用的 变量有取了另一个值，则引用次数减 1。当这个值的引用次数变为 0 时，说明已经没法再访问这个值 了，因此可以将其占用的内存回收了。引用计数策略有一个很严重的问题：循环引用。所以不常用。 如果对象 A 中包含一个指针指向对象 B，而对象 B 中也包含一个指针指向对象 A。那么这两个对象引用 次数都是 2，但实际上已经可以回收了。若这种函数被反复多次调用，会导致大量内存得不到回收
3. **原型和原型链**

**原型：**是一个对象，我们可以通过new原型的构造函数的创建的实例对象，当我们访问该对象的方法和属性是，先从该对象上查找，如果找不到，就会往它的原型上查找

**原型链：**由于对象的原型也可能是另外一个原型对象的构造函数的实例，那么如果访问对象的属性和方法在原型上也找不到，就会继续的往原型的原型上查找，如此递归下去，最终找到Object.prototype，并且它的原型为null，这样的一条由对象原型组成的节点就像链表一样，称为原型链

1. **执行上下文**

分为全局环境执行上下文，和函数环境执行上下文

**做了什么？**

1. 组合实参和新参，形成arguments
2. 函数声明提前，变量声明提前（多次声明忽略）
3. 确定当前上下文的变量对象
4. 确定this值
5. 将当前变量对象加入到作用域链的最前端

当前真正处于的执行上下文的变量对象叫做活动对象

1. **继承**

**组合继承：**

**核心**是在子类的构造函数中通过 Parent.call(this) 继承父类的属性，然后改变子 类的原型为 new Parent() 来继承父类的函数；**缺点**是在继承父类函数的时候调用了父类构造函数，导致子类的原型上多了不需要的父类属性，存在内存上的浪费

**寄生组合继承：**

相较于组合继承，在进行设置Child.prototype时，不通过new Parent进行设置，而是Object.creat(Parent.prototype, { constructor: { value: Child, enumerable: false }}

**Class 继承：**

class 实现继承的核心在于使用 extends 表明继承自哪个父类，并且在子类构造函数中必须调用 super ，因为这段代码可以看成 Parent.call(this, value)

1. **浏览器渲染机制**

现在html文档结束之后：

1. 解析html，构建DOM Tree
2. 解析CSS文件，构建CSSOM
3. 将DOM和COM连接，合成渲染树Render Tree，只有需要展示的节点和其样式信息
4. 根据Render树，计算元素大小和位置，称为布局layout也叫做回流
5. 根据布局树，计算元素对应的像素，称为painting也叫做重绘
6. 最终交给CPU进程渲染，合成图层，呈现页面元素
7. **什么是回流和重绘**

**回流：**

当render tree中的一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建

**重绘：**

当render tree中的一部分(或全部)因为元素的外观风格改变而需要重新构建

**Load、DOMContentLoaded和onreadystatechange区别：**

Window.load事件等待文档的所有资源（样式、图片、视频、脚本）都加载完成才触发；Document.DOMContentLoaded 文档的DOM树构建完成的时候触发；Document.onreadystatechange是DOM构建之后，资源Load之前触发

**Script的async和defer的区别：**

1. Aysnc和Defer都可以异步的下载JS文件，不阻塞DOM的构建
2. Async 一旦脚本下载完成之后立即执行，这也说明了Async脚本执行是无序的
3. Defer 有延迟执行的效果，等到DOM构建完成但DOMContentLoaded事件之前执行脚本，它的执行是有序的