JS

面试方式

特点：逐步挖掘、层层深入

类型检测 & 快速区分

1. JS有几种基础数据类型？几种新增？ *

- JS 8种基础数据类型：undefined null boolean number string object | symbol bigInt
- symbol 独一无二 且 不可变 => 全局变量冲突、内部变量覆盖
- bigInt 任意精度正数 安全地存储和操作大数据，即便超出了number的安全整数范围

2. 基础数据类型通常会如何进行分类？使用起来有什么区别？使用过程中你是如何区分他们的？ ***

- 可以分为：原始数据类型 + 引用数据类型
  - 原始数据类型：undefined null boolean number string
  - 引用数据类型：对象、数组、函数
- 效果不同：
  - 原始数据类型直接赋值后，不存在引用关系 属性引用关系
  - 存储位置不同：
    - 栈：原始数据类型 => 先进后出栈维护结构 => 栈区由编译器自动分配释放 => 临时变量方式
    - 堆：引用数据类型 => 堆内存由开发者进行分配 => 直到应用结束
  - 原始数据放置在栈中，空间小、大小固定、操作频繁
  - 引用类型数据量大、大小不固定，赋值给的是地址

3. 如何进行类型区分判断？几种对类型做判断区分的方式？ *

````
typeof
```js
    typeof 2  // number
    typeof true // boolean

    // 问题
    typeof {}  // object
````
````

```
    typeof []  // object
```
=> 有哪些需要注意的特例?
```js
    typeof null;  // object
    typeof NaN;  // number
```

instanceof
```js
    2 instanceof Number  // true
    [] instanceOf Array   // true
```

- 那你能说说或者手写一下instanceof的原理实现? *** 通过翻户口本，查家庭信息

```
function myInstance (left, right) {
    // 获取对象的原型
    let _proto = Object.getPrototypeOf(left);
    // 构造函数的prototype
    let _prototype = right.prototype;

    while(true) {
        if (!_proto) {
            return false;
        }

        if (_proto === _prototype) {
            return true;
        }

        _proto = Object.getPrototypeOf(_proto);
    }
}
```

constructor

```
(2).constructor === Number  // true
([]).constructor === Array  // true
```

=> 隐患? *** constructor代表的是构造函数指向的类型，可以被修改的

```
function Fn() {}
Fn.prototype = new Array();

var f = new Fn();
```

Object.prototype.toString.call()

```
let a = Object.prototype.toString;

a.call(2)
a.call([])
```

=> 为啥这里要用call？同样是检测obj调用toString，obj.toString()的结果Object.prototype.toString.call(obj)结果不一样？为什么？ **

```
保证toString是Object上的原型方法，根据原型链知识，优先调用本对象属性 => 原型链

=> 当对象中有某个属性和Object的属性重名时，使用的顺序是什么样的？如果说优先使用Object属性，如何做？**
```

类型转换

1. isNaN 和 Number.isNaN 的区别？ **

- isNaN 包含了一个**隐式转化**。 isNaN => 接收参数 => 尝试参数**转成数值型** => 不能被转数值的参数 返回true => 非数字值传入返回true
- Number.isNaN => 接收参数 => **判断参数是否为数字** => 判断是否为NaN => 不会进行数据类型转换

2. 既然说到了类型转换，有没有其他的类型转换场景？ ***

```
- 转换成字符串：
    Null 和 Undefined => 'null' 'undefined'
    Boolean => 'true'  'false'
    Number => '数字' 大数据 会转换成带有指数形式
    Symbol => '内容'
    普通对象 => '[Object Object]'

- 转成数字：
    undefined => NaN
    Null => 0
    Boolean => true | 1  false | 0
    String => 包含非数字的值NaN  空 0
    Symbol => 报错
    对象 => 相应的基本值类型 => 相应的转换
```

```
- 转成Boolean:
    undefined | null | false | +0 -0 | NaN | "" => false
```

3. 原始数据类型如何具有属性操作的？ ***

- 前置知识：js的包装类型
    - 原始数据类型，在调用属性和方法时，js会在后台隐式地将基本类型转换成对象

```
let a = 'zhaowa'

a.length; // 6

// js在收集阶段
Object(a); // String { 'zhaowa' }

// => 去包装
let a = 'zhaowa'
let b = Object(a)
let c = b.valueOf() // 'zhaowa'
```

=> 说说下面代码执行结果?

```
let a = new Boolean(false);  // => Boolean {}
if (!a) {
    console.log('hi zhaowa');
}
// never print
```

数组操作的相关问题

1. 数组的操作基本方法？如何使用？*

```
转换方法: toString() toLocalString() join()
尾操作: pop() push()
首操作: shift() unshift()
排序: reverse() sort()
连接: concat()
截取: slice()
插入: splice()
索引: indexOf()
```

```
迭代方法: every() some() filter() map() forEach()
归并: reduce()
```

> splice()：删除、插入和替换

```
arr.splice( 0 , 2 ) // 数组arr从索引0开始删除，共删除两项

arr.splice( 2,0,4,6 ) // 数组arr从索引2开始插入，共插入两项元素，即4，6（参数0的意思就是没有要删除的项）
```

变量提升 & 作用域

1. 谈谈对于变量提升以及作用域的理解?

- 现象： -无论在任何位置声明的 函数、变量 都被提升到模块、函数的顶部
  - JS实现原理：
    - 解析 | 执行
    - 解析：检查语法、预编译，代码中即将执行的变量和函数声明调整到全局顶部，并且赋值为undefined，上下文、arguments、函数参数 全局上下文：变量定义，函数声明 函数上下文：变量定义，函数声明，this，arguments
    - 再去执行阶段，按照代码顺序从上而下逐行运行
  - 变量提升存在意义?
    - 提高性能 解析引用提升了性能，不需要执行到时重新解析
    - 更加灵活 补充定义这样一种玩法
- 指出特殊case
  - let const 取消了变量提升机制的玩法

闭包

1. 什么是闭包？闭包的作用？ *

- 在一个函数中访问另一个**函数作用域中变量的方法**
- 闭包的作用：
    - 函数外部可以访问到函数内部的变量。
    - 跨作用域，创建私有变量
    - 已经运行结束的逻辑，依然残留在闭包里，变量得不到回收

2. 闭包经典题目结果和改造方式？ *

```
for (var i = 1; i < 9; i++) {
    setTimeout((function a() {
        console.log(i)
    }, i * 1000))
}

// 利用闭包解决
 for (var i = 1; i < 9; i++) {
    (function(j) {
        setTimeout((function a() {
            console.log(j)
        }, j * 1000))
    })(i)
}

// 利用作用域
for (let i = 1; i < 9; i++) {
    setTimeout((function a() {
        console.log(i)
    }, i * 1000))
}
```

ES6

**1. 叛逆型问题：const对象的属性可以修改吗？new一个箭头函数会发生什么呢？ **

- const 只能保证指针固定不变的，指向的数据结构属性，无法控制是否变化的
- new执行全过程：
    - 创建一个对象
    - 构造函数作用域付给新对象
    - 指向构造函数后，构造函数中的this指向该对象 返回一个新的对象
- 箭头函数 没有prototype，也没有独立的this指向，更没有arguments

2. JS ES内置对象有哪些？ **

- 内置对象
  - 值属性类： Infinity, NaN, undefined, null
  - 函数属性：eval(), parseInt()
  - 对象：Object, Function, Boolean, Symbol, Error
  - 数字：Number, Math, Date
  - 字符串：String, RegExp
  - 集合：Map, set, weakMap
  - 抽象控制：promise
  - 映射：proxy

Proxy的设计初衷:
在ES6中，新增了一个Proxy类，Proxy翻译为“代理”，是用于帮助我们创建一个代理的

如果我们希望监听一个对象的相关操作，那么我们可以先创建一个代理对象（Proxy对象）

```
const objProxy = new Proxy(obj, handler);
```

Proxy捕获器

```
const obj = {
  name: "xs",
  age: 18,
};

const objProxy = new Proxy(obj, {
  // 获取值 捕获器 target:obj,
  get(target, key) {
    console.log(`监听到对象的${key}属性被访问了`, target);
    return target[key];
  },

  // 设置值 捕获器
  set(target, key, newValue) {
    console.log(`监听到对象的${key}属性被赋值了`, target);
    target[key] = newValue;
  },

  // has 捕获器
  has(target, key) {
    console.log(`监听到对象的${key}属性in操作`, target);
    target.has(key)
  },

 // delete 捕获器
  deleteProperty(target, key) {
    console.log(`监听到对象的${key}属性删除操作`, target);
    delete target[key];
  },
```

```
});
```

原型 & 原型链

1. 简单说说原型原型链理解？ *

- 构造函数：
  - Js中用来构造新建一个对象的
  - 构造函数内部有一个属性prototype, 这个prototype值是一个对象，包含了共享的属性和方法
  - 使用构造函数创建对象后，被创建对象内部会存在一个指针（**proto**） => 指向构造函数prototype属性的对应值
- 链式获取属性规则：
  - 对象的属性 => 对象内部本身是否包含该属性
  - 对象的属性 => 顺着指针去原型对象里查找 => 在往上层级里去查找

1. 继承方式？ *

> （1）第一种是以**原型链**的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被**所有的实例对象所共享**，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

```
// Game类
function Game() {
    this.name = 'lol'
}
Game.prototype.getName = function() {
    return this.name;
}
// LOL类
function LOL() {}
LOL.prototype = new Game();
LOL.prototype.constructor = LOL;
const game = new LOL();
// 本质：重写原型对象方式，将父对象的属性方法，作为子对象原型对象的属性和方法
```

（2）第二种方式是使用借用**构造函数**的方式，这种方式是通过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在的一个问题就是**无法实现函数方法的复用**，并且**超类型原型定义的方法子类型也没有办法访问到**。

```
function Game(arg) {
    this.name = 'lol';
    this.skin = ['s'];
}
Game.prototype.getName = function() {
    return this.name;
}

// LOL类
function LOL(arg) {
    Game.call(this, arg);
}

const game3 = new LOL('arg');
// 解决了共享属性的问题 + 子向父传参问题
```

（3）第三种方式是**组合继承**，组合继承是将**原型链和借用构造函数组合起来**使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以**调用了两次超类的构造函数**，造成了子类型的原型中多了很多不必要的属性。

```
function Game(arg) {
    this.name = 'lol';
    this.skin = ['s'];
}
Game.prototype.getName = function() {
    return this.name;
}

// LOL类
function LOL(arg) {
    Game.call(this, arg);
}
LOL.prototype = new Game();
LOL.prototype.constructor = LOL;

const game3 = new LOL();
```

> （4）第四种方式是**寄生式组合继承**，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的**原型的副本**来作为子类型的原型，这样就避免了创建不必要的属性。

```
function Game(arg) {
    this.name = 'lol';
    this.skin = ['s'];
}
Game.prototype.getName = function() {
    return this.name;
}

// LOL类
function LOL(arg) {
    Game.call(this, arg);
}
LOL.prototype = Object.create(Game.prototype);
LOL.prototype.constructor = LOL;
```

异步编程

1. 聊聊遇到哪些异步执行方式？ *

- 回调函数 => cb 回调地狱
- promise => 链式调用 => 语义不明确
- generator => 考虑如何控制执行 co库
- async await => 不改变同步书写习惯的前提下，异步处理

2. 聊聊promise的理解 *

- 一个对象、一个容器 => 触发操作
- 三个状态： pending | resolved | rejected
- 两个过程： pending => resolved pending => rejected
- promise缺点：
  - 无法取消
  - pending状态，无细分状态

内存 & 浏览器执行相关

1. 简单说说看对垃圾回收的理解？ *

- 垃圾回收概念：

  > jS具有自动垃圾回收机制，找到不再使用的变量，释放其占用的内存空间

- JS存在两种变量：
  - 局部变量 + 全局变量

2. 现代浏览器如何处理垃圾回收？ **

- 标记清除、引用计数
- 内存中所有变量加上标记，当前环境状态。定期进行标记变量的回收。
- 变量加上的是被引用使用的使用方个数。降低到0时自动清除

3. 减少垃圾的方案 ***

- 数组优化：清空数组时，赋值一个[] => length = 0
- object优化：对象尽量复用，减少深拷贝
- 函数优化：循环中的函数表达式，尽量统一放在外面