JavaScript 面试题汇总（上篇）

1. 根据下面 *ES6* 构造函数的书写方式，要求写出 *ES5* 的

```
class Example {
  constructor(name) {
    this.name = name;
  }
  init() {
    const fun = () => { console.log(this.name) }
    fun();
  }
}
const e = new Example('Hello');
e.init();
```

参考答案：

```
function Example(name) {
   'use strict';
   if (!new.target) {
        throw new TypeError('Class constructor cannot be invoked without new');
   }
   this.name = name;
}

Object.defineProperty(Example.prototype, 'init', {
   enumerable: false,
   value: function () {
        'use strict';
        if (new.target) {
            throw new TypeError('init is not a constructor');
        }
        var fun = function () {
            console.log(this.name);
        }
        fun.call(this);
   }
})
```

解析：

此题的关键在于是否清楚 *ES6* 的 *class* 和普通构造函数的区别，记住它们有以下区别，就不会有遗漏：

1. *ES6* 中的 *class* 必须通过 *new* 来调用，不能当做普通函数调用，否则报错

   因此，在答案中，加入了 *new.target* 来判断调用方式

2. *ES6* 的 *class* 中的所有代码均处于严格模式之下

> 因此，在答案中，无论是构造函数本身，还是原型方法，都使用了严格模式
>
> 1. *ES6* 中的原型方法是不可被枚举的
>
>   因此，在答案中，定义原型方法使用了属性描述符，让其不可枚举
>
> 2. 原型上的方法不允许通过 *new* 来调用
>
>   因此，在答案中，原型方法中加入了 *new.target* 来判断调用方式

2. 数组去重有哪些方法？（美团 *19* 年）

> 参考答案：

```
// 数字或字符串数组去重，效率高
function unique(arr) {
   var result = {}; // 利用对象属性名的唯一性来保证不重复
   for (var i = 0; i < arr.length; i++) {
        if (!result[arr[i]]) {
            result[arr[i]] = true;
        }
   }
   return Object.keys(result); // 获取对象所有属性名的数组
}

// 任意数组去重，适配范围光，效率低
function unique(arr) {
   var result = []; // 结果数组
   for (var i = 0; i < arr.length; i++) {
        if (!result.includes(arr[i])) {
            result.push(arr[i]);
        }
   }
   return result;
}

// 利用ES6的Set去重，适配范围广，效率一般，书写简单
function unique(arr) {
   return [...new Set(arr)]
}
```

3. 描述下列代码的执行结果

```
foo(typeof a);
function foo(p) {
    console.log(this);
    console.log(p);
    console.log(typeof b);
    let b = 0;
}
```

> 参考答案：
>
> 报错，报错的位置在 `console.log(typeof b);`
>
> 报错原因：*ReferenceError: Cannot access 'b' before initialization*

> 解析：
>
> 这道题考查的是 *ES6* 新增的声明变量关键字 *let* 以及暂时性死区的知识。*let* 和以前的 *var* 关键字不一样，无法在 *let* 声明变量之前访问到该变量，所以在 *typeof b* 的地方就会报错。

4. 描述下列代码的执行结果

```
class Foo {
    constructor(arr) {
        this.arr = arr;
    }
    bar(n) {
        return this.arr.slice(0, n);
    }
}
var f = new Foo([0, 1, 2, 3]);
console.log(f.bar(1));
console.log(f.bar(2).splice(1, 1));
console.log(f.arr);
```

> 参考答案：
>
> [0]
> [1]
> [0, 1, 2, 3]

> 解析：
>
> 主要考察的是数组相关的知识。*f* 对象上面有一个属性 *arr*，*arr* 的值在初始化的时候会被初始化为 *[0, 1, 2, 3]*，之后就完全是考察数组以及数组方法的使用了。

5. 描述下列代码的执行结果

```
01 function f(count) {
02    console.log(`foo${count}`);
03    setTimeout(() => { console.log(`bar${count}`); });
04 }
05 f(1);
06 f(2);
07 setTimeout(() => { f(3); });
```

> 参考答案：

> foo1
> foo2
> bar1
> bar2
> foo3
> bar3

> 解析：
>
> 这个完全是考察的异步的知识。调用 *f(1)* 的时候，会执行同步代码，打印出 *foo1*，然后 *03* 行的 *setTimeout* 被放入到异步执行队列，接下来调用 *f(2)* 的时候，打印出 *foo2*，后面 *03* 行的 *setTimeout* 又被放入到异步执行队列。然后执行 *07* 行的语句，被放入到异步执行队列。至此，所有同步代码就都执行完毕了。
>
> 接下来开始执行异步代码，那么大家时间没写，就都是相同的，所以谁先被放入到异步队列，谁就先执行，所以先打印出 *bar1*、然后是 *bar2*，接下来执行之前 *07* 行放入到异步队列里面的 *setTimeout*，先执行 *f* 函数里面的同步代码，打印出 *foo3*，然后是最后一个异步，打印出 *bar3*

6. 描述下列代码的执行结果

```
var a = 2;
var b = 5;
console.log(a === 2 || 1 && b === 3 || 4);
```

> 参考答案：
>
> *true*
>
> 考察的是逻辑运算符。在 || 里面，只要有一个为真，后面的直接短路，都不用去计算。所以 *a === 2* 得到 *true* 之后直接短路了，返回 *true*。

7. 描述下列代码的执行结果

```
export class ButtonWrapper {
    constructor(domBtnEl, hash) {
        this.domBtnEl = domBtnEl;
        this.hash = hash;
        this.bindEvent();
    }
    bindEvent() {
        this.domBtnEl.addEventListener('click', this.clickEvent, false);
    }
    detachEvent() {
        this.domBtnEl.removeEventListener('click', this.clickEvent);
    }
    clickEvent() {
        console.log(`The hash of the button is: ${this.hash}`);
    }
}
```

> 参考答案：
>
> 上面的代码导出了一个 *ButtonWrapper* 类，该类在被实例化的时候，实例化对象上面有两个属性，分别是 *domBtnEl* 和 *hash*，*domBtnEl* 是一个 *DOM* 节点，之后为这个 *domBtnEl* 绑定了点击事件，点击后打印出 *The hash of the button is: hash* 那句话。*detachEvent* 是移除点击事件，当调用实例化对象的 *detachEvent* 方法时，点击事件就会被移除。

8. 箭头函数有哪些特点

> 参考答案：
>
> 1. 更简洁的语法，例如
>   - 只有一个形参就不需要用括号括起来
>   - 如果函数体只有一行，就不需要放到一个块中
>   - 如果 *return* 语句是函数体内唯一的语句，就不需要 *return* 关键字
> 2. 箭头函数没有自己的 *this*，*arguments*，*super*
> 3. 箭头函数 *this* 只会从自己的作用域链的上一层继承 *this*。

9. 说一说类的继承

> 参考答案：
>
> 继承是面向对象编程中的三大特性之一。
>
> *JavaScript* 中的继承经过不断的发展，从最初的对象冒充慢慢发展到了今天的圣杯模式继承。
>
> 其中最需要掌握的就是**伪经典继承**和**圣杯模式**的继承。
>
> 很长一段时间，JS 继承使用的都是**组合继承**。这种继承也被称之为伪经典继承，该继承方式综合了原型链和盗用构造函数的方式，将两者的优点集中了起来。
>
> 组合继承弥补了之前原型链和盗用构造函数这两种方式各自的不足，是 *JavaScript* 中使用最多的继承方式。
>
> 组合继承最大的问题就是效率问题。最主要就是父类的构造函数始终会被调用两次：一次是在创建子类原型时调用，另一次是在子类构造函数中调用。
>
> 本质上，子类原型最终是要包含超类对象的所有实例属性，子类构造函数只要在执行时重写自己的原型就行了。
>
> 圣杯模式的继承解决了这一问题，其基本思路就是不通过调用父类构造函数来给子类原型赋值，而是取得父类原型的一个副本，然后将返回的新对象赋值给子类原型。

> 解析：该题主要考察就是对 *js* 中的继承是否了解，以及常见的继承的形式有哪些。最常用的继承就是**组合继承**（伪经典继承）和圣杯模式继承。下面附上 *js* 中这两种继承模式的详细解析。
>
> 下面是一个组合继承的例子：

```
// 基类
var Person = function (name, age) {
 this.name = name;
 this.age = age;
}
```

```
Person.prototype.test = "this is a test";
Person.prototype.testFunc = function () {
 console.log('this is a testFunc');
}

// 子类
var Student = function (name, age, gender, score) {
 Person.apply(this, [name, age]); // 盗用构造函数
 this.gender = gender;
 this.score = score;
}
Student.prototype = new Person(); // 改变 Student 构造函数的原型对象
Student.prototype.testStuFunc = function () {
 console.log('this is a testStuFunc');
}

// 测试
var zhangsan = new Student("张三", 18, "男", 100);
console.log(zhangsan.name); // 张三
console.log(zhangsan.age); // 18
console.log(zhangsan.gender); // 男
console.log(zhangsan.score); // 100
console.log(zhangsan.test); // this is a test
zhangsan.testFunc(); // this is a testFunc
zhangsan.testStuFunc(); // this is a testStuFunc
```

在上面的例子中，我们使用了组合继承的方式来实现继承，可以看到无论是基类上面的属性和方法，还是子类自己的属性和方法，都得到了很好的实现。

但是在组合继承中存在效率问题，比如在上面的代码中，我们其实调用了两次 *Person*，产生了两组 *name* 和 *age* 属性，一组在原型上，一组在实例上。

也就是说，我们在执行 *Student.prototype = new Person( )* 的时候，我们是想要 *Person* 原型上面的方法，属性是不需要的，因为属性之后可以通过 *Person.apply(this, [name, age])* 拿到，但是当你 *new Person( )* 的时候，会实例化一个 *Person* 对象出来，这个对象上面，属性和方法都有。

圣杯模式的继承解决了这一问题，其基本思路就是不通过调用父类构造函数来给子类原型赋值，而是取得父类原型的一个副本，然后将返回的新对象赋值给子类原型。

下面是一个圣杯模式的示例：

```
// target 是子类，origin 是基类
// target ---> Student, origin ---> Person
function inherit(target, origin) {
 function F() { }; // 没有任何多余的属性

 // origin.prototype === Person.prototype, origin.prototype.constructor === Person 构造函数
 F.prototype = origin.prototype;
```

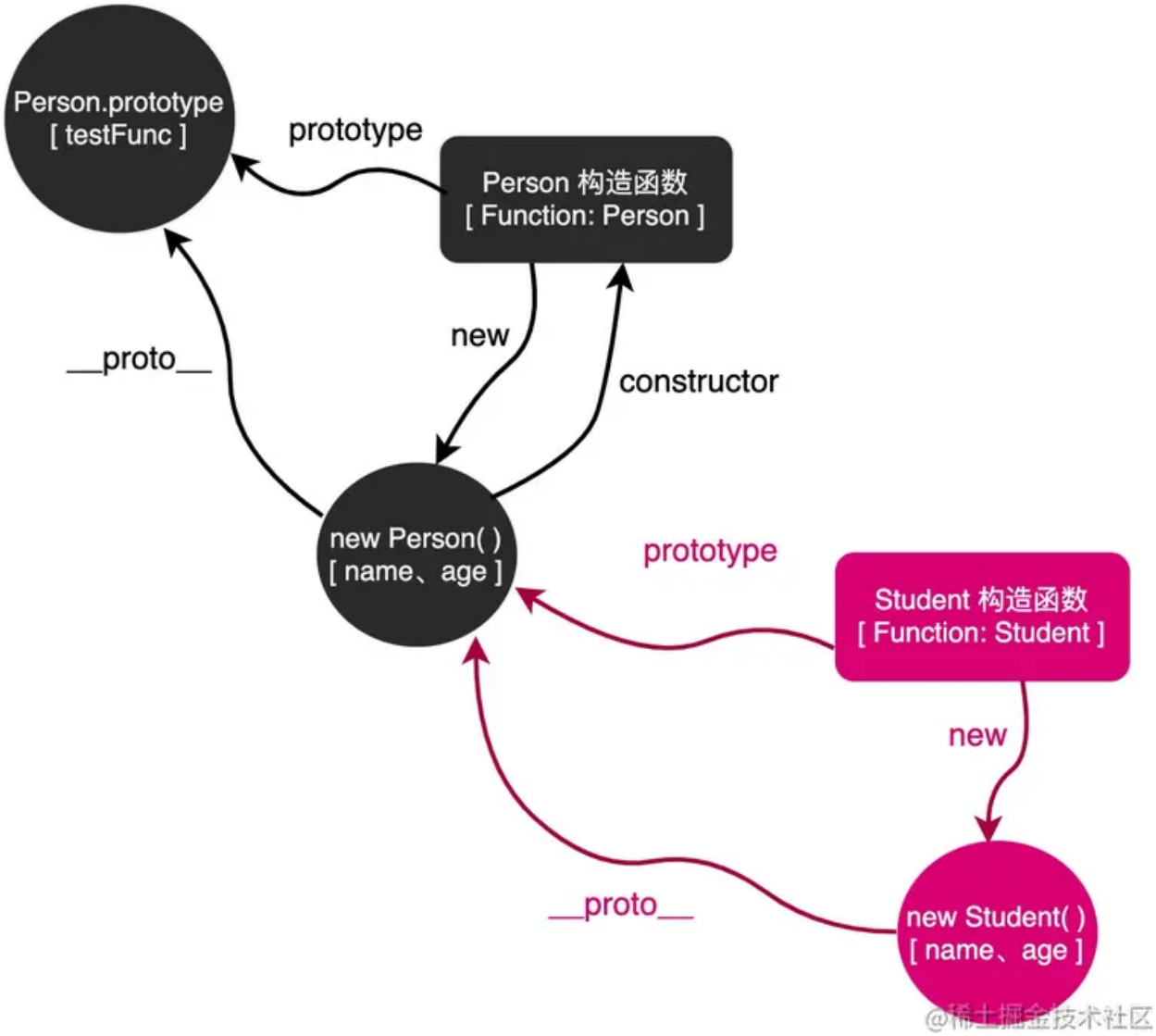
```
 // 假设 new F() 出来的对象叫小 f
 // 那么这个 f 的原型对象 === F.prototype === Person.prototype
 // 那么 f.constructor === Person.prototype.constructor === Person 的构造函数
 target.prototype = new F();

 // 而 f 这个对象又是 target 对象的原型对象
 // 这意味着 target.prototype.constructor === f.constructor
 // 所以 target 的 constructor 会指向 Person 构造函数

 // 我们要让子类的 constructor 重新指向自己
 // 若不修改则会发现 constructor 指向的是父类的构造函数
 target.prototype.constructor = target;
}


// 基类
var Person = function (name, age) {
 this.name = name;
 this.age = age;
}
Person.prototype.test = "this is a test";
Person.prototype.testFunc = function () {
 console.log('this is a testFunc');
}


// 子类
var Student = function (name, age, gender, score) {
 Person.apply(this, [name, age]);
 this.gender = gender;
 this.score = score;
}
inherit(Student, Person); // 使用圣杯模式实现继承
// 在子类上面添加方法
Student.prototype.testStuFunc = function () {
 console.log('this is a testStuFunc');
}

// 测试
var zhangsan = new Student("张三", 18, "男", 100);

console.log(zhangsan.name); // 张三
console.log(zhangsan.age); // 18
console.log(zhangsan.gender); // 男
console.log(zhangsan.score); // 100
console.log(zhangsan.test); // this is a test
zhangsan.testFunc(); // this is a testFunc
zhangsan.testStuFunc(); // this is a testStuFunc
```

在上面的代码中，我们在 *inherit* 方法中创建了一个中间层，之后让 *F* 的原型和父类的原型指向同一地址，再让子类的原型指向这个 *F* 的实例化对象来实现了继承。

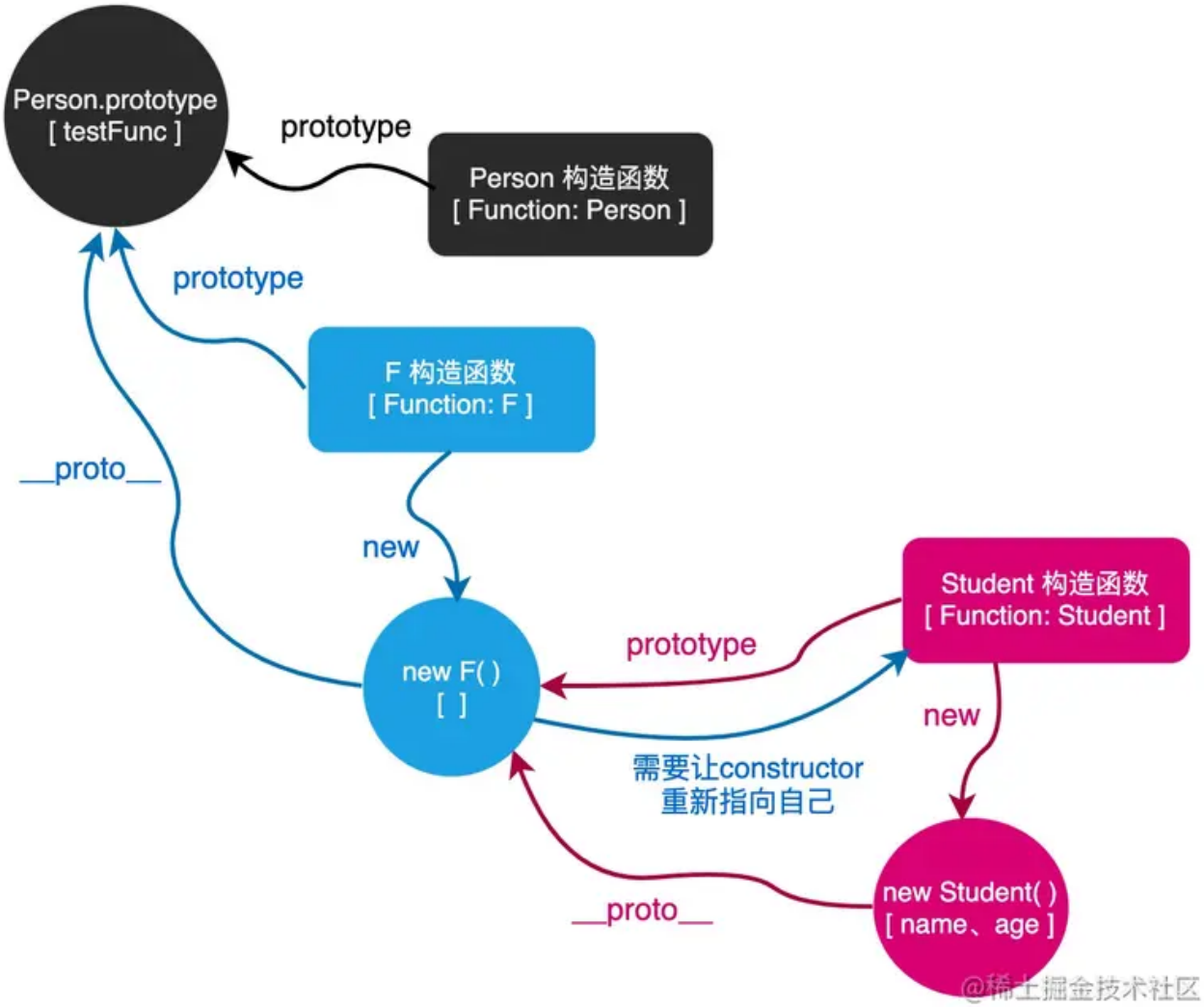
这样我们的继承，属性就不会像之前那样实例对象上一份，原型对象上一份，拥有两份。圣杯模式继承是目前 *js* 继承的最优解。

最后我再画个图帮助大家理解，如下图：

组合模式（伪经典模式）下的继承示意图：



圣杯模式下的继承示意图：



10. *new* 操作符都做了哪些事?

参考答案：

new 运算符创建一个用户定义的对象类型的实例或具有构造函数的内置对象的实例。

new 关键字会进行如下的操作：
步骤 *1*：创建一个空的简单 *JavaScript* 对象，即 {}；
步骤 *2*：链接该对象到另一个对象（即设置该对象的原型对象）；
步骤 *3*：将步骤 *1* 新创建的对象作为 *this* 的上下文；
步骤 *4*：如果该函数没有返回对象，则返回 *this*。

11. *call*、*apply*、*bind* 的区别？

参考答案：

call 和 *apply* 的功能相同，区别在于传参的方式不一样:

- *fn.call(obj, arg1, arg2, ...)* 调用一个函数, 具有一个指定的 *this* 值和分别地提供的参数(参数的列表)。
- *fn.apply(obj, [argsArray])* 调用一个函数，具有一个指定的 *this* 值，以及作为一个数组（或类数组对象）提供的参数。

> *bind* 和 *call/apply* 有一个很重要的区别，一个函数被 *call/apply* 的时候，会直接调用，但是 *bind* 会创建一个新函数。当这个新函数被调用时，*bind( )* 的第一个参数将作为它运行时的 *this*，之后的一序列参数将会在传递的实参前传入作为它的参数。

12. 事件循环机制（宏任务、微任务）

> 参考答案：
>
> 在 *js* 中任务会分为同步任务和异步任务。
>
> 如果是同步任务，则会在主线程（也就是 *js* 引擎线程）上进行执行，形成一个执行栈。但是一旦遇到异步任务，则会将这些异步任务交给异步模块去处理，然后主线程继续执行后面的同步代码。
>
> 当异步任务有了运行结果以后，就会在任务队列里面放置一个事件，这个任务队列由事件触发线程来进行管理。
>
> 一旦执行栈中所有的同步任务执行完毕，就代表着当前的主线程（*js* 引擎线程）空闲了，系统就会读取任务队列，将可以运行的异步任务添加到执行栈中，开始执行。
>
> 在 *js* 中，任务队列中的任务又可以被分为 *2* 种类型：宏任务（*macrotask*）与微任务（*microtask*）
>
> 宏任务可以理解为每次执行栈所执行的代码就是一个宏任务，包括每次从事件队列中获取一个事件回调并放到执行栈中所执行的任务。
>
> 微任务可以理解为当前宏任务执行结束后立即执行的任务。

13. 你了解 *node* 中的事件循环机制吗？*node11* 版本以后有什么改变

> 参考答案：
>
> *Node.js* 在主线程里维护了一个**事件队列，**当接到请求后，就将该请求作为一个事件放入这个队列中，然后继续接收其他请求。当主线程空闲时（没有请求接入时），就开始循环事件队列，检查队列中是否有要处理的事件，这时要分两种情况：如果是非 *I/O* 任务，就亲自处理，并通过回调函数返回到上层调用；如果是 *I/O* 任务，就从*线程池**中拿出一个线程来处理这个事件，并指定回调函数，然后继续循环队列中的其他事件。
>
> 当线程中的 *I/O* 任务完成以后，就执行指定的回调函数，并把这个完成的事件放到事件队列的尾部，等待事件循环，当主线程再次循环到该事件时，就直接处理并返回给上层调用。 这个过程就叫 **事件循环** (*Event Loop*)。
>
> 无论是 *Linux* 平台还是 *Windows* 平台，*Node.js* 内部都是通过**线程池**来完成异步 *I/O* 操作的，而 *LIBUV* 针对不同平台的差异性实现了统一调用。因此，***Node.js* 的单线程仅仅是指 *JavaScript* 运行在单线程中，而并非 *Node.js* 是单线程。**
>
> *Node.JS* 的事件循环分为 *6* 个阶段：
>
> - *timers* 阶段：这个阶段执行 *timer*（ *setTimeout*、*setInterval* ）的回调
> - *I/O callbacks* 阶段：处理一些上一轮循环中的少数未执行的 *I/O* 回调
> - *idle*、*prepare* 阶段：仅 *Node.js* 内部使用
> - *poll* 阶段：获取新的 *I/O* 事件, 适当的条件下 *Node.js* 将阻塞在这里
> - *check* 阶段：执行 *setImmediate( )* 的回调
> - *close callbacks* 阶段：执行 *socket* 的 *close* 事件回调

事件循环的执行顺序为：

外部输入数据 –-> 轮询阶段（ *poll* ） –-> 检查阶段（ *check* ） –-> 关闭事件回调阶段（ *close callback* ） –-> 定时器检测阶段（ *timer* ） –-> *I/O* 事件回调阶段（ *I/O callbacks* ） –->闲置阶段（ *idle*、*prepare* ） –->轮询阶段（按照该顺序反复运行）...

浏览器和 *Node.js* 环境下，微任务任务队列的执行时机不同

- *Node.js* 端，微任务在事件循环的各个阶段之间执行
- 浏览器端，微任务在事件循环的宏任务执行完之后执行

Node.js v11.0.0 版本于 *2018* 年 *10* 月，主要有以下变化：

1. *V8* 引擎更新至版本 *7.0*
2. *http*、*https* 和 *tls* 模块默认使用 *WHESWG URL* 解析器。
3. 隐藏子进程的控制台窗口默认改为了 *true*。
4. *FreeBSD 10*不再支持。
5. 增加了多线程 *Worker Threads*

14. 什么是函数柯里化?

参考答案：

柯里化（*currying*）又称部分求值。一个柯里化的函数首先会接受一些参数，接受了这些参数之后，该函数并不会立即求值，而是继续返回另外一个函数，刚才传入的参数在函数形成的闭包中被保存起来。待到函数被真正需要求值的时候，之前传入的所有参数都会被一次性用于求值。

举个例子，就是把原本：

function(arg1,arg2) 变成 *function(arg1)(arg2)*
function(arg1,arg2,arg3) 变成 *function(arg1)(arg2)(arg3)*
function(arg1,arg2,arg3,arg4) 变成 *function(arg1)(arg2)(arg3)(arg4)*

总而言之，就是将：

function(arg1,arg2,...,argn) 变成 *function(arg1)(arg2)...(argn)*

15. *promise.all* 方法的使用场景？数组中必须每一项都是 *promise* 对象吗？不是 *promise* 对象会如何处理？

参考答案：

***promise.all(promiseArray)** * 方法是 *promise* 对象上的静态方法，该方法的作用是将多个 *promise* 对象实例包装，生成并返回一个新的 *promise* 实例。

此方法在集合多个 *promise* 的返回结果时很有用。

返回值将会按照参数内的 *promise* 顺序排列，而不是由调用 *promise* 的完成顺序决定。

***promise.all* 的特点**

接收一个*Promise*实例的数组或具有*Iterator*接口的对象

> 如果元素不是*Promise*对象，则使用*Promise.resolve*转成*Promise*对象
>
> 如果全部成功，状态变为*resolved*，返回值将组成一个数组传给回调
>
> 只有有一个失败，状态就变为 *rejected*，返回值将直接传递给回调 *all( )*的返回值，也是新的 *promise* 对象

16. *this* 的指向哪几种？

> 参考答案：
>
> 总结起来，*this* 的指向规律有如下几条：
>
> - 在函数体中，非显式或隐式地简单调用函数时，在严格模式下，函数内的 *this* 会被绑定到 *undefined* 上，在非严格模式下则会被绑定到全局对象 *window/global* 上。
> - 一般使用 *new* 方法调用构造函数时，构造函数内的 *this* 会被绑定到新创建的对象上。
> - 一般通过 *call/apply/bind* 方法显式调用函数时，函数体内的 *this* 会被绑定到指定参数的对象上。
> - 一般通过上下文对象调用函数时，函数体内的 *this* 会被绑定到该对象上。
> - 在箭头函数中，*this* 的指向是由外层（函数或全局）作用域来决定的。

17. *JS* 中继承实现的几种方式

> 参考答案：
>
> *JS* 的继承随着语言的发展，从最早的对象冒充到现在的圣杯模式，涌现出了很多不同的继承方式。每一种新的继承方式都是对前一种继承方式不足的一种补充。
>
> 1. 原型链继承
>
> - 重点：让新实例的原型等于父类的实例。
> - 特点：实例可继承的属性有：实例的构造函数的属性，父类构造函数属性，父类原型的属性。（新实例不会继承父类实例的属性！）
> - 缺点：
>   - 1、新实例无法向父类构造函数传参。
>   - 2、继承单一。
>   - 3、所有新实例都会共享父类实例的属性。（原型上的属性是共享的，一个实例修改了原型属性，另一个实例的原型属性也会被修改！）
>
> 1. 借用构造函数继承
>
> - 重点：用 *call( )* 和 *apply( )* 将父类构造函数引入子类函数（在子类函数中做了父类函数的自执行（复制））
> - 特点：
>   - 1、只继承了父类构造函数的属性，没有继承父类原型的属性。
>   - 2、解决了原型链继承缺点1、2、3。
>   - 3、可以继承多个构造函数属性（call多个）。
>   - 4、在子实例中可向父实例传参。

- 缺点：
  - 1、只能继承父类构造函数的属性。
    - 2、无法实现构造函数的复用。（每次用每次都要重新调用）
    - 3、每个新实例都有父类构造函数的副本，臃肿。

1. 组合模式（又被称之为伪经典模式）

- 重点：结合了两种模式的优点，传参和复用
- 特点：
  - 1、可以继承父类原型上的属性，可以传参，可复用。
  - 2、每个新实例引入的构造函数属性是私有的。
- 缺点：调用了两次父类构造函数（耗内存），子类的构造函数会代替原型上的那个父类构造函数。

1. 寄生组合式继承（圣杯模式）

- 重点：修复了组合继承的问题

18. 什么是事件监听

参考答案：

首先需要区别清楚事件监听和事件监听器。

在绑定事件的时候，我们需要对应的书写一个事件处理程序，来应对事件发生时的具体行为。

这个事件处理程序我们也称之为事件监听器。

当事件绑定好后，程序就会对事件进行监听，当用户触发事件时，就会执行对应的事件处理程序。

关于事件监听，*W3C* 规范中定义了 *3* 个事件阶段，依次是捕获阶段、目标阶段、冒泡阶段。

- **捕获**阶段：在事件对象到达事件目标之前，事件对象必须从 *window* 经过目标的祖先节点传播到事件目标。 这个阶段被我们称之为捕获阶段。在这个阶段注册的事件监听器在事件到达其目标前必须先处理事件。
- **目标** 阶段：事件对象到达其事件目标。 这个阶段被我们称为目标阶段。一旦事件对象到达事件目标，该阶段的事件监听器就要对它进行处理。如果一个事件对象类型被标志为不能冒泡。那么对应的事件对象在到达此阶段时就会终止传播。
- **冒泡** 阶段：事件对象以一个与捕获阶段相反的方向从事件目标传播经过其祖先节点传播到 *window*。这个阶段被称之为冒泡阶段。在此阶段注册的事件监听器会对相应的冒泡事件进行处理。

19. 什么是 *js* 的闭包？有什么作用？

参考答案：

一个函数和对其周围状态（*lexical environment*，词法环境）的引用捆绑在一起（或者说函数被引用包围），这样的组合就是**闭包**（*closure*）。也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域。在 *JavaScript* 中，每当创建一个函数，闭包就会在函数创建的同时被创建出来。

闭包的用处：

1. 匿名自执行函数

2. 结果缓存
3. 封装
4. 实现类和继承

20. 事件委托以及冒泡原理

参考答案：

事件委托，又被称之为事件代理。在 *JavaScript* 中，添加到页面上的事件处理程序数量将直接关系到页面整体的运行性能。导致这一问题的原因是多方面的。

首先，每个函数都是对象，都会占用内存。内存中的对象越多，性能就越差。其次，必须事先指定所有事件处理程序而导致的 *DOM* 访问次数，会延迟整个页面的交互就绪时间。

对事件处理程序过多问题的解决方案就是事件委托。

事件委托利用了事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。例如，*click* 事件会一直冒泡到 *document* 层次。也就是说，我们可以为整个页面指定一个 *onclick* 事件处理程序，而不必给每个可单击的元素分别添加事件处理程序。

事件冒泡（*event bubbling*），是指事件开始时由最具体的元素（文档中嵌套层次最深的那个节点）接收，然后逐级向上传播到较为不具体的节点（文档）。

21. *let const var* 的区别？什么是块级作用域？如何用？

参考答案：

1. *var* 定义的变量，没有块的概念，可以跨块访问, 不能跨函数访问，有变量提升。
2. *let* 定义的变量，只能在块作用域里访问，不能跨块访问，也不能跨函数访问，无变量提升，不可以重复声明。
3. *const* 用来定义常量，使用时必须初始化(即必须赋值)，只能在块作用域里访问，而且不能修改，无变量提升，不可以重复声明。

最初在 *JS* 中作用域有：全局作用域、函数作用域。没有块作用域的概念。

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。

在以前没有块作用域的时候，在 if 或者 for 循环中声明的变量会泄露成全局变量，其次就是 { } 中的内层变量可能会覆盖外层变量。块级作用域的出现解决了这些问题。

22. *ES5* 的方法实现块级作用域（立即执行函数） *ES6* 呢？

参考答案：

ES6 原生支持块级作用域。块作用域由 { } 包括，*if* 语句和 *for* 语句里面的 { } 也属于块作用域。

使用 *let* 声明的变量或者使用 *const* 声明的常量，只能在块作用域里访问，不能跨块访问。

23. *ES6* 箭头函数的特性

> 参考答案：
>
> 1. 更简洁的语法，例如
>   - 只有一个形参就不需要用括号括起来
>   - 如果函数体只有一行，就不需要放到一个块中
>   - 如果 *return* 语句是函数体内唯一的语句，就不需要 *return* 关键字
> 2. 箭头函数没有自己的 *this*，*arguments*，*super*
> 3. 箭头函数 *this* 只会从自己的作用域链的上一层继承 *this*。

24. 箭头函数与普通函数的区别？

> 参考答案：
>
> 1. 外形不同。箭头函数使用箭头定义，普通函数中没有
> 2. 普通函数可以有匿名函数，也可以有具体名函数，但是箭头函数都是匿名函数。
> 3. **箭头函数不能用于构造函数，不能使用 *new*，**普通函数可以用于构造函数，以此创建对象实例。
> 4. **箭头函数中 *this* 的指向不同，**在普通函数中，*this* 总是指向调用它的对象，如果用作构造函数，*this* 指向创建的对象实例。
>   箭头函数本身不创建 *this*，也可以说箭头函数本身没有 *this*，但是它在声明时可以捕获其所在上下文的 *this* 供自己使用。
> 5. 每一个普通函数调用后都具有一个 *arguments* 对象，用来存储实际传递的参数。
>
>   但是箭头函数并没有此对象。**取而代之用rest参数来解决。**
> 6. 箭头函数不能用于 *Generator* 函数，不能使用 *yeild* 关键字。
> 7. 箭头函数不具有 *prototype* 原型对象。而普通函数具有 *prototype* 原型对象。
> 8. 箭头函数不具有 *super*。
> 9. 箭头函数不具有 *new.target*。

25. *JS* 的基本数据类型有哪些？基本数据类型和引用数据类型的区别

> 参考答案：
>
> 在 *JavaScript* 中，数据类型整体上来讲可以分为两大类：**基本类型**和**引用数据类型**
>
> 基本数据类型，一共有 *6* 种：
>
> ```
> string, symbol, number, boolean, undefined, null
> ```
>
> 其中 *symbol* 类型是在 *ES6* 里面新添加的基本数据类型。
>
> 引用数据类型，就只有 *1* 种：
>
> ```
> object
> ```

> 基本数据类型的值又被称之为原始值或简单值，而引用数据类型的值又被称之为复杂值或引用值。
>
> 两者的区别在于：
>
> **原始值是表示 *JavaScript* 中可用的数据或信息的最底层形式或最简单形式。**简单类型的值被称为原始值，是因为它们是*不可细化**的。
>
> 也就是说，数字是数字，字符是字符，布尔值是 *true* 或 *false*，*null* 和 *undefined* 就是 *null* 和 *undefined*。这些值本身很简单，不能够再进行拆分。由于原始值的数据大小是固定的，所以**原始值的数据是存储于内存中的栈区里面的。**
>
> 在 *JavaScript* 中，对象就是一个引用值。因为对象可以向下拆分，拆分成多个简单值或者复杂值。**引用值在内存中的大小是未知的，因为引用值可以包含任何值，而不是一个特定的已知值，所以引用值的数据都是存储于堆区里面。**
>
> 最后总结一下两者的区别：
>
> 1. 访问方式
>   - 原始值：访问到的是值
>   - 引用值：访问到的是引用地址
> 2. 比较方式
>   - 原始值：比较的是值
>   - 引用值：比较的是地址
> 3. 动态属性
>   - 原始值：无法添加动态属性
>   - 引用值：可以添加动态属性
> 4. 变量赋值
>   - 原始值：赋值的是值
>   - 引用值：赋值的是地址

26. *NaN* 是什么的缩写

> 参考答案：
>
> *NaN* 的全称为 *Not a Number*，表示非数，或者说不是一个数。虽然 NaN 表示非数，但是它却属于 *number* 类型。
>
> *NaN* 有两个特点：
>
> 1. 任何涉及 *NaN* 的操作都会返回 *NaN*
> 2. *NaN* 和任何值都不相等，包括它自己本身

27. *JS* 的作用域类型

> 参考答案：
>
> 在 *JavaScript* 里面，作用域一共有 4 种：全局作用域，局部作用域、函数作用域以及 *eval* 作用域。
>
> **全局作用域：**这个是默认的代码运行环境，一旦代码被载入，引擎最先进入的就是这个环境。
>
> **局部作用域：**当使用 *let* 或者 *const* 声明变量时，这些变量在一对花括号中存在局部作用域，只能够在花括号内部进行访问使用。
>
> **函数作用域：**当进入到一个函数的时候，就会产生一个函数作用域。函数作用域里面所声明的变量只在函数中提供访问使用。
>
> ***eval* 作用域：**当调用 *eval( )* 函数的时候，就会产生一个 *eval* 作用域。

28. *undefined==null* 返回的结果是什么？*undefined* 与 *null* 的区别在哪?

> 参考答案：
>
> 返回 *true*。
>
> 这两个值都表示“无”的意思。
>
> 通常情况下， 当我们试图访问某个不存在的或者没有赋值的变量时，就会得到一个 *undefined* 值。*Javascript* 会自动将声明是没有进行初始化的变量设为 *undifined*。
>
> 而 *null* 值表示空，*null* 不能通过 *Javascript* 来自动赋值，也就是说必须要我们自己手动来给某个变量赋值为 *null*。

> 解析：
>
> 那么为什么 *JavaScript* 要设置两个表示"无"的值呢？这其实是历史原因。
>
> *1995* 年 *JavaScript* 诞生时，最初像 *Java* 一样，只设置了 *null* 作为表示"无"的值。根据 *C* 语言的传统，*null* 被设计成可以自动转为*0*。
>
> 但是，*JavaScript* 的设计者，觉得这样做还不够，主要有以下两个原因。
>
> 1. *null* 像在 *Java* 里一样，被当成一个对象。但是，*JavaScript* 的数据类型分成原始类型（*primitive*）和合成类型（*complex*）两大类，作者觉得表示"无"的值最好不是对象。
> 2. *JavaScript* 的最初版本没有包括错误处理机制，发生数据类型不匹配时，往往是自动转换类型或者默默地失败。作者觉得，如果 *null* 自动转为 *0*，很不容易发现错误。
>
> 因此，作者又设计了一个 *undefined*。
>
> **这里注意：先有 *null* 后有 *undefined* 出来，*undefined* 是为了填补之前的坑。**
>
> *JavaScript* 的最初版本是这样区分的：
>
> *null* 是一个表示"无"的对象（空对象指针），转为数值时为 *0*；
>
> 典型用法是：
>
> - 作为函数的参数，表示该函数的参数不是对象。

- 作为对象原型链的终点。

undefined 是一个表示"无"的原始值，转为数值时为 *NaN*。

典型用法是：

- 变量被声明了，但没有赋值时，就等于 *undefined*。
- 调用函数时，应该提供的参数没有提供，该参数等于 *undefined*。
- 对象没有赋值的属性，该属性的值为 *undefined*。
- 函数没有返回值时，默认返回 *undefined*。

29. 写一个函数判断变量类型

参考答案：

```
function getType(data){
 let type = typeof data;
 if(type !== "object"){
     return type
 }
 return Object.prototype.toString.call(data).replace(/^[object (\S+)]$/,'$1')
}
function Person(){}
console.log(getType(1)); // number
console.log(getType(true)); // boolean
console.log(getType([1,2,3])); // Array
console.log(getType(/abc/)); // RegExp
console.log(getType(new Date)); // Date
console.log(getType(new Person)); // Object
console.log(getType({})); // Object
```

30. *js* 的异步处理函数

参考答案：

在最早期的时候，*JavaScript* 中要实现异步操作，使用的就是 *Callback* 回调函数。

但是回调函数会产生回调地狱（*Callback Hell*）

之后 ES6 推出了 *Promise* 解决方案来解决回调地狱的问题。不过，虽然 *Promise* 作为 *ES6* 中提供的一种新的异步编程解决方案，但是它也有问题。比如，代码并没有因为新方法的出现而减少，反而变得更加复杂，同时理解难度也加大。

之后，就出现了基于 *Generator* 的异步解决方案。不过，这种方式需要编写外部的执行器，而执行器的代码写起来一点也不简单。当然也可以使用一些插件，比如 *co* 模块来简化执行器的编写。

ES7 提出的 *async* 函数，终于让 *JavaScript* 对于异步操作有了终极解决方案。

实际上，*async* 只是生成器的一种语法糖而已，简化了外部执行器的代码，同时利用 *await* 替代 *yield*，*async* 替代生成器的`*`号。

31. *defer* 与 *async* 的区别

> 参考答案：
>
> 按照惯例，所有 *script* 元素都应该放在页面的 *head* 元素中。这种做法的目的就是把**所有外部文件（*CSS* 文件和 *JavaScript* 文件）的引用都放在相同的地方**。可是，在文档的 *head* 元素中包含所有 *JavaScript* 文件，意味着必须等到全部 *JavaScript* 代码都被下载、解析和执行完成以后，才能开始呈现页面的内容（浏览器在遇到 *body* 标签时才开始呈现内容）。
>
> 对于那些需要很多 *JavaScript* 代码的页面来说，这无疑会导致浏览器在呈现页面时出现明显的延迟，而延迟期间的浏览器窗口中将是一片空白。为了避免这个问题，现在 ***Web* 应用程序一般都全部 *JavaScript* 引用放在 *body* 元素中页面的内容后面**。这样一来，在解析包含的 *JavaScript* 代码之前，页面的内容将完全呈现在浏览器中。而用户也会因为浏览器窗口显示空白页面的时间缩短而感到打开页面的速度加快了。
>
> 有了 *defer* 和 *async* 后，这种局面得到了改善。
>
> ***defer*（延迟脚本）**
>
> 延迟脚本：*defer* 属性只适用于外部脚本文件。
>
> 如果给 *script* 标签定义了*defer* 属性，这个属性的作用是表明脚本在执行时不会影响页面的构造。也就是说，脚本会被延迟到整个页面都解析完毕后再运行。因此，如果 *script* 元素中设置了 *defer* 属性，相当于告诉浏览器立即下载，但延迟执行。
>
> ***async*（异步脚本）**
>
> 异步脚本：*async* 属性也只适用于外部脚本文件，并告诉浏览器立即下载文件。
>
> **但与 *defer* 不同的是：标记为 *async* 的脚本并不保证按照指定它们的先后顺序执行。**
>
> 所以总结起来，两者之间最大的差异就是在于脚本下载完之后何时执行，显然 *defer* 是最接近我们对于应用脚本加载和执行的要求的。
>
> *defer* 是立即下载但延迟执行，加载后续文档元素的过程将和脚本的加载并行进行（异步），但是脚本的执行要在所有元素解析完成之后，*DOMContentLoaded* 事件触发之前完成。*async* 是立即下载并执行，加载和渲染后续文档元素的过程将和 *js* 脚本的加载与执行并行进行（异步）。

32. 浏览器事件循环和任务队列

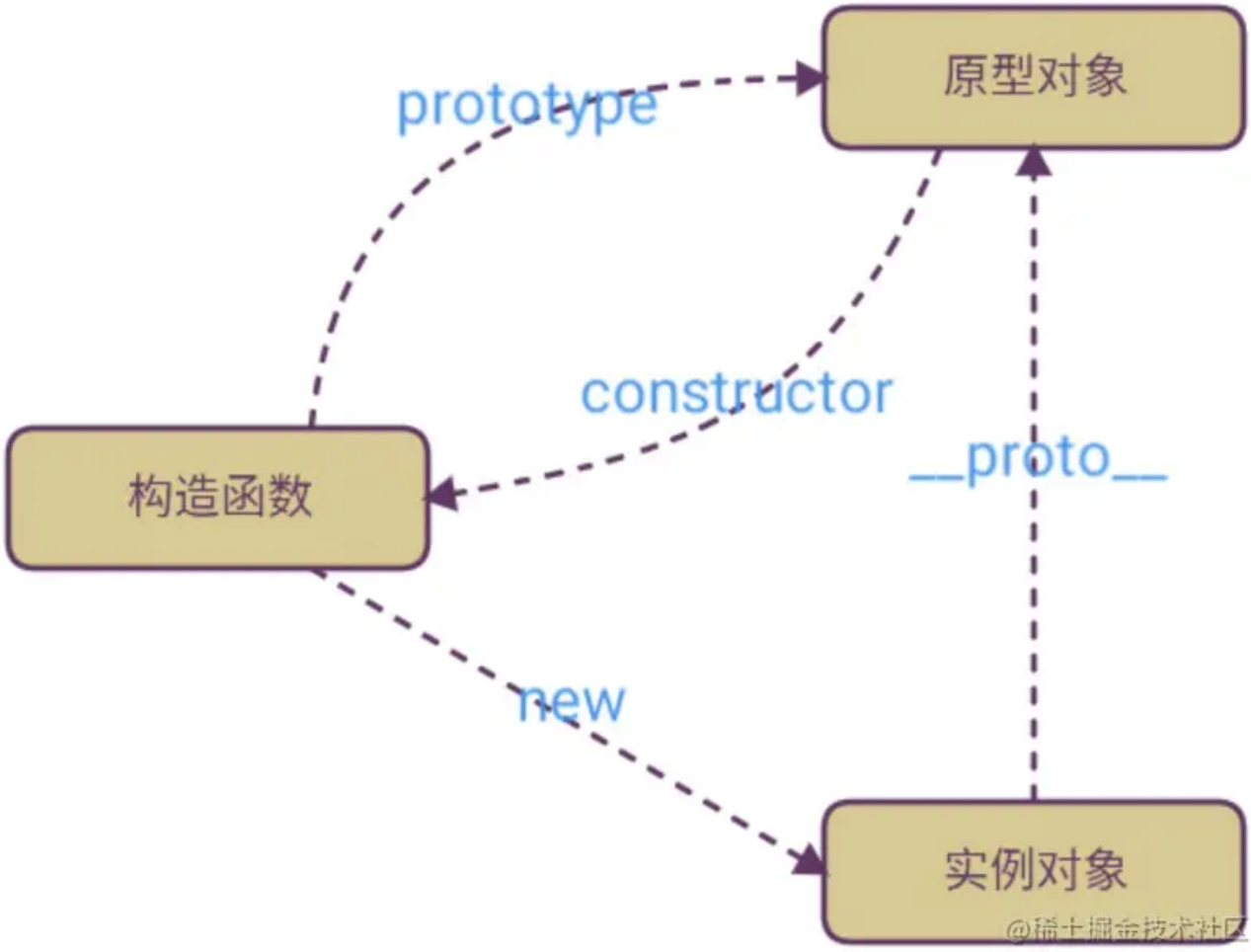
> 参考答案：
>
> *JavaScript* 的异步机制由事件循环和任务队列构成。
>
> *JavaScript* 本身是单线程语言，所谓异步依赖于浏览器或者操作系统等完成。*JavaScript* 主线程拥有一个执行栈以及一个任务队列，主线程会依次执行代码，当遇到函数时，会先将函数入栈，函数运行完毕后再将该函数出栈，直到所有代码执行完毕。
>
> 遇到异步操作（例如：*setTimeout*、*Ajax*）时，异步操作会由浏览器(*OS*)执行，浏览器会在这些任务完成后，将事先定义的回调函数推入主线程的任务队列(*task queue*)中,当主线程的执行栈清空之后会读取任务队列中的回调函数,当任务队列被读取完毕之后,主线程接着执行,从而进入一个无限的循环，这就是事件循环。

33. 原型与原型链 （美团 19年）

参考答案：

- 每个对象都有一个 `__proto__` 属性，该属性指向自己的原型对象
- 每个构造函数都有一个 `prototype` 属性，该属性指向实例对象的原型对象
- 原型对象里的 `constructor` 指向构造函数本身

如下图：



每个对象都有自己的原型对象，而原型对象本身，也有自己的原型对象，从而形成了一条原型链条。

当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

34. 作用域与作用域链 （美团 19年）

参考答案：

作用域是在运行时代码中的某些特定部分中变量，函数和对象的可访问性。换句话说，作用域决定了代码区块中变量和其他资源的可见性。*ES6* 之前 *JavaScript* 没有块级作用域，只有全局作用域和函数作用域。*ES6* 的到来，为我们提供了块级作用域。

> 作用域链指的是作用域与作用域之间形成的链条。当我们查找一个当前作用域没有定义的变量（自由变量）的时候，就会向上一级作用域寻找，如果上一级也没有，就再一层一层向上寻找，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是作用域链 。

35. 闭包及应用场景以及闭包缺点 （美团 19年）

> 参考答案：
>
> 闭包的应用场景：
>
> 1. 匿名自执行函数
> 2. 结果缓存
> 3. 封装
> 4. 实现类和继承
>
> 闭包的缺点：
>
> 因为闭包的作用域链会引用包含它的函数的活动对象，导致这些活动对象不会被销毁，因此会占用更多的内存。

36. 继承方式 （美团 19年）

> 参考答案：
>
> 参阅前面第 *9* 题以及第 *18* 题答案。

37. 原始值与引用值 （美团 19年）

> 参考答案：
>
> **原始值是表示 *JavaScript* 中可用的数据或信息的最底层形式或最简单形式。**简单类型的值被称为原始值，是因为它们是****不可细化的。
>
> 也就是说，数字是数字，字符是字符，布尔值是 *true* 或 *false*，*null* 和 *undefined* 就是 *null* 和 *undefined*。这些值本身很简单，不能够再进行拆分。由于原始值的数据大小是固定的，所以**原始值的数据是存储于内存中的栈区里面的。**
>
> 在 *JavaScript* 中，对象就是一个引用值。因为对象可以向下拆分，拆分成多个简单值或者复杂值。**引用值在内存中的大小是未知的，因为引用值可以包含任何值，而不是一个特定的已知值，所以引用值的数据都是存储于堆区里面。**
>
> 最后总结一下两者的区别：
>
> 1. 访问方式
>   - 原始值：访问到的是值
>   - 引用值：访问到的是引用地址
> 2. 比较方式
>   - 原始值：比较的是值
>   - 引用值：比较的是地址
> 3. 动态属性

    - 原始值：无法添加动态属性
    - 引用值：可以添加动态属性
4. 变量赋值
    - 原始值：赋值的是值
    - 引用值：赋值的是地址

38. 描述下列代码的执行结果

```
const first = () => (new Promise((resolve, reject) => {
    console.log(3);
    let p = new Promise((resolve, reject) => {
        console.log(7);
        setTimeout(() => {
            console.log(1);
        }, 0);
        setTimeout(() => {
            console.log(2);
            resolve(3);
        }, 0)
        resolve(4);
    });
    resolve(2);
    p.then((arg) => {
        console.log(arg, 5); // 1 bb
    });
    setTimeout(() => {
        console.log(6);
    }, 0);
}))
first().then((arg) => {
    console.log(arg, 7); // 2 aa
    setTimeout(() => {
        console.log(8);
    }, 0);
});
setTimeout(() => {
    console.log(9);
}, 0);
console.log(10);
```

参考答案：

```
3
7
10
4 5
2 7
```

```
1
2
6
9
8
```

39. 如何判断数组或对象（美团 19年）

参考答案：

1. 通过 *instanceof* 进行判断

```
var arr = [1,2,3,1];
console.log(arr instanceof Array) // true
```

1. 通过对象的 *constructor* 属性

```
var arr = [1,2,3,1];
console.log(arr.constructor === Array) // true
```

1. *Object.prototype.toString.call(arr)*

```
console.log(Object.prototype.toString.call({name: "jerry"}));//[object Object]
console.log(Object.prototype.toString.call([]));//[object Array]
```

1. 可以通过 *ES6* 新提供的方法 *Array.isArray( )*

```
Array.isArray([]) //true
```

40. 对象深拷贝与浅拷贝，单独问了 *Object.assign*（美团 19年）

参考答案：

- **浅拷贝**：只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做浅拷贝（浅复制）

  浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存。
- **深拷贝**：在堆中重新分配内存，并且把源对象所有属性都进行新建拷贝，以保证深拷贝的对象的引用图不包含任何原有对象或对象图上的任何对象，拷贝后的对象与原来的对象是完全隔离，互不影响。

Object.assign 方法可以把任意多个的源对象自身的可枚举属性拷贝给目标对象，然后返回目标对象。但是 *Object.assign* 方法进行的是浅拷贝，拷贝的是对象的属性的引用，而不是对象本身。

42. 说说 *instanceof* 原理，并回答下面的题目（美团 19年）

```
function A(){}
function B(){}
A.prototype = new B();
let a = new A();
console.log(a instanceof B) // true of false ?
```

> 参考答案：
>
> 答案为 *true*。
>
> *instanceof* 原理：
>
> *instanceof* 用于检测一个对象是否为某个构造函数的实例。
>
> 例如：*A instanceof B*
> *instanceof* 用于检测对象 *A* 是不是 *B* 的实例，而检测是基于原型链进行查找的，也就是说 *B* 的 *prototype* 有没有在对象 *A* 的***proto*** 原型链上，如果有就返回 *true*，否则返回 *false*

###

43. 内存泄漏（美团 19 年）

> 参考答案：
>
> 内存泄漏（*Memory Leak*）是指程序中己动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。
>
> *Javascript* 是一种高级语言，它不像 *C* 语言那样要手动申请内存，然后手动释放，*Javascript* 在声明变量的时候自动会分配内存，普通的类型比如 *number*，一般放在栈内存里，对象放在堆内存里，声明一个变量，就分配一些内存，然后定时进行垃圾回收。垃圾回收的任务由 *JavaScript* 引擎中的垃圾回收器来完成，它监视所有对象，并删除那些不可访问的对象。
>
> 基本的垃圾回收算法称为**“标记-清除”**，定期执行以下“垃圾回收”步骤:
>
> - 垃圾回收器获取根并**“标记”**(记住)它们。
> - 然后它访问并“标记”所有来自它们的引用。
> - 然后它访问标记的对象并标记它们的引用。所有被访问的对象都被记住，以便以后不再访问同一个对象两次。
> - 以此类推，直到有未访问的引用(可以从根访问)为止。
> - 除标记的对象外，所有对象都被删除。

44. *ES6* 新增哪些东西？让你自己说（美团 19 年）

> 参考答案：
>
> *ES6* 新增内容众多，这里列举出一些关键的以及平时常用的新增内容：
>
> 1. 箭头函数
> 2. 字符串模板

3. 支持模块化（*import*、*export*）
4. 类（*class*、*constructor*、*extends*）
5. *let*、*const* 关键字
6. 新增一些数组、字符串等内置构造函数方法，例如 *Array.from*、*Array.of* 、*Math.sign*、*Math.trunc* 等
7. 新增一些语法，例如扩展操作符、解构、函数默认参数等
8. 新增一种基本数据类型 *Symbol*
9. 新增元编程相关，例如 *proxy*、*Reflect*
10. *Set* 和 *Map* 数据结构
11. *Promise*
12. *Generator* 生成器

45. *weakmap*、*weakset*（美团 *19* 年）

参考答案：

WeakSet 对象是一些对象值的集合, 并且其中的每个对象值都只能出现一次。在 *WeakSet* 的集合中是唯一的

它和 *Set* 对象的区别有两点:

- 与 *Set* 相比，*WeakSet* 只能是**对象的集合**，而不能是任何类型的任意值。
- *WeakSet* 持弱引用：集合中对象的引用为弱引用。 如果没有其他的对 *WeakSet* 中对象的引用，那么这些对象会被当成垃圾回收掉。 这也意味着 *WeakSet* 中没有存储当前对象的列表。 正因为这样，*WeakSet* 是不可枚举的。

WeakMap 对象也是键值对的集合。它的**键必须是对象类型**，值可以是任意类型。它的键被弱保持，也就是说，当其键所指对象没有其他地方引用的时候，它会被 *GC* 回收掉。*WeakMap* 提供的接口与 *Map* 相同。

与 *Map* 对象不同的是，*WeakMap* 的键是不可枚举的。不提供列出其键的方法。列表是否存在取决于垃圾回收器的状态，是不可预知的。

46. 为什么 *ES6* 会新增 *Promise*（美团 19年）

参考答案：

在 *ES6* 以前，解决异步的方法是回调函数。但是回调函数有一个最大的问题就是回调地狱（*callback hell*），当我们的回调函数嵌套的层数过多时，就会导致代码横向发展。

Promise 的出现就是为了解决回调地狱的问题。

47. *ES5* 实现继承？（虾皮）

参考答案：

1. 借用构造函数实现继承

```
function Parent1(){
    this.name = "parent1"
}
function Child1(){
    Parent1.call(this);
    this.type = "child1";
}
```

缺点：*Child1* 无法继承 *Parent1* 的原型对象，并没有真正的实现继承 (部分继承)。

1. 借用原型链实现继承

```
function Parent2(){
    this.name = "parent2";
    this.play = [1,2,3];
}
function Child2(){
    this.type = "child2";
}
Child2.prototype = new Parent2();
```

缺点：原型对象的属性是共享的。

1. 组合式继承

```
function Parent3(){
    this.name = "parent3";
    this.play = [1,2,3];
}
function Child3(){
    Parent3.call(this);
    this.type = "child3";
}
Child3.prototype = Object.create(Parent3.prototype);
Child3.prototype.constructor = Child3;
```

48. 科里化？（搜狗）

参考答案：

柯里化，英语全称 *Currying*，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

举个例子，就是把原本：

function(arg1,arg2) 变成 *function(arg1)(arg2)*
function(arg1,arg2,arg3) 变成 *function(arg1)(arg2)(arg3)*
function(arg1,arg2,arg3,arg4) 变成 *function(arg1)(arg2)(arg3)(arg4)*

总而言之，就是将：

> *function(arg1,arg2,...,argn)* 变成 *function(arg1)(arg2)...(argn)*

49. 防抖和节流？（虾皮）

> 参考答案：
>
> 我们在平时开发的时候，会有很多场景会频繁触发事件，比如说搜索框实时发请求，*onmousemove*、*resize*、*onscroll* 等，有些时候，我们并不能或者不想频繁触发事件，这时候就应该用到函数防抖和函数节流。
>
> 函数防抖(*debounce*)，指的是短时间内多次触发同一事件，只执行最后一次，或者只执行最开始的一次，中间的不执行。
>
> 函数节流(*throttle*)，指连续触发事件但是在 *n* 秒中只执行一次函数。即 *2n* 秒内执行 *2* 次... 。节流如字面意思，会稀释函数的执行频率。

50. 闭包？（好未来---探讨了 *40* 分钟）

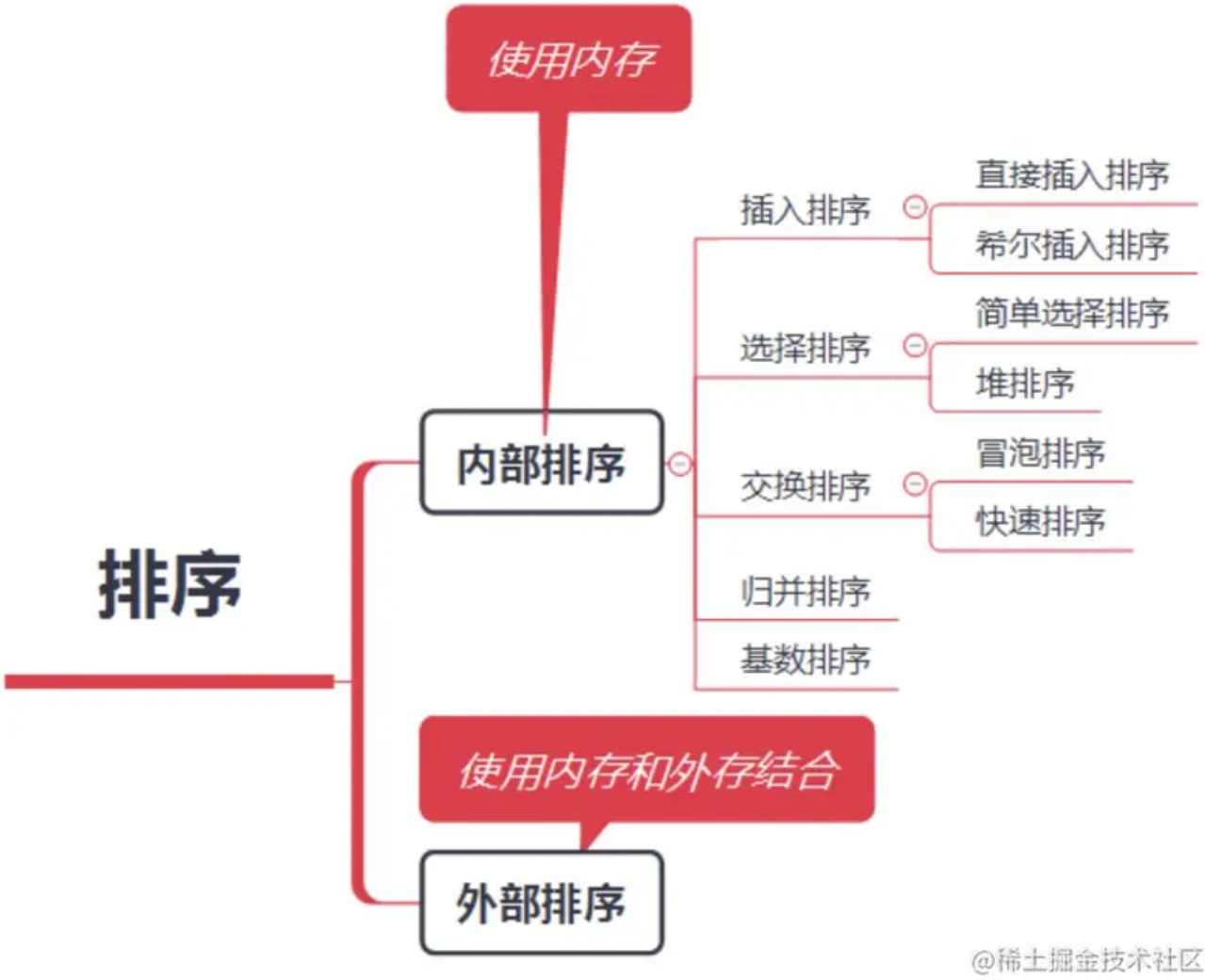
> 参考答案：
>
> 请参阅前面第 *20* 题以及第 *36* 题答案。

51. 原型和原型链？（字节）

> 参考答案：
>
> 请参阅前面第 *34* 题答案。

52. 排序算法---（时间复杂度、空间复杂度）

> 参考答案：
>
> 算法（*Algorithm*）是指用来操作数据、解决程序问题的一组方法。对于同一个问题，使用不同的算法，也许最终得到的结果是一样的，但在过程中消耗的资源和时间却会有很大的区别。
>
> 主要还是从算法所占用的「时间」和「空间」两个维度去考量。
>
> - 时间维度：是指执行当前算法所消耗的时间，我们通常用「时间复杂度」来描述。
> - 空间维度：是指执行当前算法需要占用多少内存空间，我们通常用「空间复杂度」来描述。
>
> 因此，评价一个算法的效率主要是看它的时间复杂度和空间复杂度情况。然而，有的时候时间和空间却又是「鱼和熊掌」，不可兼得的，那么我们就需要从中去取一个平衡点。
>
> 排序也称排序算法(*Sort Algorithm*)，排序是将**一组数据**，依**指定的顺序**进行**排列的过程**。
>
> 排序的分类分为**内部排序**和**外部排序法**。
>
> - 内部排序：指将需要处理的所有数据都加载到**内部存储器(内存)**中进行排序。
> - 外部排序：**数据量过大**，无法全部加载到内存中，需要借助**外部存储(文件等)**进行排序。



53. 浏览器事件循环和 *node* 事件循环（搜狗）

参考答案：

1. 浏览器中的 *Event Loop*

事件循环中的异步队列有两种：*macro*（宏任务）队列和 *micro*（微任务）队列。**宏任务队列可以有多个，微任务队列只有一个**。

- 常见的 *macro-task* 比如：*setTimeout*、*setInterval*、 *setImmediate*、*script*（整体代码）、 *I/O* 操作、*UI* 渲染等。
- 常见的 *micro-task* 比如: *process.nextTick*、*new Promise( ).then*(回调)、*MutationObserver*(*html5* 新特性) 等。

当某个宏任务执行完后,会查看是否有微任务队列。如果有，先执行微任务队列中的所有任务，如果没有，会读取宏任务队列中排在最前的任务，执行宏任务的过程中，遇到微任务，依次加入微任务队列。栈空后，再次读取微任务队列里的任务，依次类推。

1. *Node* 中的事件循环

Node 中的 *Event Loop* 和浏览器中的是完全不相同的东西。*Node.js* 采用 *V8* 作为 *js* 的解析引擎，而 *I/O* 处理方面使用了自己设计的 *libuv*，*libuv* 是一个基于事件驱动的跨平台抽象层，封装了不同操作系统一些底层特性，对外提供统一的 *API*，事件循环机制也是它里面的实现。

> *Node.JS* 的事件循环分为 *6* 个阶段：
>
> - *timers* 阶段：这个阶段执行 *timer*（ *setTimeout*、*setInterval* ）的回调
> - *I/O callbacks* 阶段：处理一些上一轮循环中的少数未执行的 *I/O* 回调
> - *idle*、*prepare* 阶段：仅 *Node.js* 内部使用
> - *poll* 阶段：获取新的 *I/O* 事件, 适当的条件下 *Node.js* 将阻塞在这里
> - *check* 阶段：执行 *setImmediate( )* 的回调
> - *close callbacks* 阶段：执行 *socket* 的 *close* 事件回调
>
> *Node.js* 的运行机制如下:
>
> - *V8* 引擎解析 *JavaScript* 脚本。
> - 解析后的代码，调用 *Node API*。
> - *libuv* 库负责 *Node API* 的执行。它将不同的任务分配给不同的线程，形成一个 *Event Loop*（事件循环），以异步的方式将任务的执行结果返回给 *V8* 引擎。
> - *V8* 引擎再将结果返回给用户。

54. 闭包的好处

> 参考答案：
>
> 请参阅前面第 *20* 题以及第 *36* 题答案。

55. *let*、*const*、*var* 的区别

> 参考答案：
>
> 1. *var* 定义的变量，没有块的概念，可以跨块访问, 不能跨函数访问，有变量提升。
> 2. *let* 定义的变量，只能在块作用域里访问，不能跨块访问，也不能跨函数访问，无变量提升，不可以重复声明。
> 3. *const* 用来定义常量，使用时必须初始化(即必须赋值)，只能在块作用域里访问，而且不能修改，无变量提升，不可以重复声明。

56. 闭包、作用域（可以扩充到作用域链）

> 参考答案：
>
> **什么是作业域?**
>
> ES5 中只存在两种作用域：全局作用域和函数作用域。在 JavaScript 中，我们将作用域定义为一套规则，这套规则用来管理引擎如何在当前作用域以及嵌套子作用域中根据标识符名称进行变量(变量名或者函数名)查找。
>
> **什么是作用域链?**
>
> 当访问一个变量时，编译器在执行这段代码时，会首先从当前的作用域中查找是否有这个标识符，如果没有找到，就会去父作用域查找，如果父作用域还没找到继续向上查找，直到全局作用域为止,，而作用域链，就是有当前作用域与上层作用域的一系列变量对象组成，它保证了当前执行的作用域对符合访问权限的变量和函数的有序访问。

闭包产生的本质

当前环境中存在指向父级作用域的引用

什么是闭包

闭包是一种特殊的对象，它由两部分组成：执行上下文(代号 A)，以及在该执行上下文中创建的函数 (代号 B)，当 B 执行时，如果访问了 A 中变量对象的值，那么闭包就会产生，且在 Chrome 中使用这个执行上下文 A 的函数名代指闭包。

一般如何产生闭包

- 返回函数
- 函数当做参数传递

闭包的应用场景

- 柯里化 bind
- 模块

57. *Promise*

参考答案：

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理且更强大。它最早由社区提出并实现，*ES6*将其写进了语言标准，统一了用法，并原生提供了*Promise*对象。

特点

1. 对象的状态不受外界影响 （*3* 种状态）
    - *Pending* 状态（进行中）
    - *Fulfilled* 状态（已成功）
    - *Rejected* 状态（已失败）
2. 一旦状态改变就不会再变 （两种状态改变：成功或失败）
    - *Pending -> Fulfilled*
    - *Pending -> Rejected*

用法

```
var promise = new Promise(function(resolve, reject){
    // ... some code

    if (/* 异步操作成功 */) {
        resolve(value);
    } else {
        reject(error);
    }
})
```

58. 实现一个函数,对一个url进行请求,失败就再次请求,超过最大次数就走失败回调,任何一次成功都走成功回调

参考答案:

示例代码如下:

```
/**
 @params url: 请求接口地址;
 @params body: 设置的请求体;
 @params succ: 请求成功后的回调
 @params error: 请求失败后的回调
 @params maxCount: 设置请求的数量
*/
function request(url, body, succ, error, maxCount = 5) {
 return fetch(url, body)
      .then(res => succ(res))
      .catch(err => {
          if (maxCount <= 0) return error('请求超时');
          return request(url, body, succ, error, --maxCount);
      });
}

// 调用请求函数
request('https://java.some.com/pc/reqCount', { method: 'GET', headers: {} },
 (res) => {
     console.log(res.data);
 },
 (err) => {
     console.log(err);
 })
```

59. 冒泡排序

参考答案:

冒泡排序的核心思想是:

1. 比较相邻的两个元素，如果前一个比后一个大或者小（取决于排序的顺序是小到大还是大到小），则交换位置。
2. 比较完第一轮的时候，最后一个元素是最大或最小的元素。
3. 这时候最后一个元素已经是最大或最小的了，所以下一次冒泡的时候最后一个元素不需要参与比较。

示例代码:

```
function bSort(arr) {
    var len = arr.length;
    // 外层 for 循环控制冒泡的次数
    for (var i = 0; i < len - 1; i++) {
```

```
        for (var j = 0; j < len - 1 - i; j++) {
            // 内层 for 循环控制每一次冒泡需要比较的次数
            // 因为之后每一次冒泡的两两比较次数会越来越少，所以 -i
            if (arr[j] > arr[j + 1]) {
                var temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
}

//举个数组
myArr = [20, -1, 27, -7, 35];
//使用函数
console.log(bSort(myArr)); // [ -7, -1, 20, 27, 35 ]
```

60. 数组降维

参考答案：

数组降维就是将一个嵌套多层的数组进行降维操作，也就是对数组进行扁平化。在 *ES5* 时代我们需要自己手写方法或者借助函数库来完成，但是现在可以使用 *ES6* 新提供的数组方法 *flat* 来完成数组降维操作。

解析：使用 *flat* 方法会接收一个参数，这个参数是数值类型，是要处理扁平化数组的深度，生成后的新数组是独立存在的，不会对原数组产生影响。

flat 方法的语法如下：

```
var newArray = arr.flat([depth])
```

其中 *depth* 指定要提取嵌套数组结构的深度，默认值为 *1*。

示例如下：

```
var arr = [1, 2, [3, 4, [5, 6]]];
console.log(arr.flat());      // [1, 2, 3, 4, [5, 6]]
console.log(arr.flat(2));     // [1, 2, 3, 4, 5, 6]
```

上面的代码定义了一个层嵌套的数组，默认情况下只会拍平一层数组，也就是把原来的三维数组降低到了二维数组。在传入的参数为 *2* 时，则会降低两维，成为一个一维数组。

使用 *Infinity*，可展开任意深度的嵌套数组，示例如下：

```
var arr = [1, 2, [3, 4, [5, 6, [7, 8]]]];
console.log(arr.flat(Infinity));  // [1, 2, 3, 4, 5, 6, 7, 8]
```

在数组中有空项的时候，使用 *flat* 方法会将中的空项进行移除。

```
var arr = [1, 2, , 4, 5];
console.log(arr.flat()); // [1, 2, 4, 5]
```

上面的代码中，数组中第三项是空值，在使用 *flat* 后会对空项进行移除。

61. *call apply bind*

参考答案：

请参阅前面第 *11* 题答案。

62. promise 代码题

```
new Promise((resolve, reject) => {
    reject(1);
    console.log(2);
    resolve(3);
    console.log(4);
}).then((res) => { console.log(res) })
    .catch(res => { console.log('reject1') })
try {
    new Promise((resolve, reject) => {
        throw 'error'
    }).then((res) => { console.log(res) })
        .catch(res => { console.log('reject2') })
} catch (err) {
    console.log(err)
}
```

参考答案：

2
4
reject1
reject2

直播课或者录播课进行解析。

63. *proxy* 是实现代理，可以改变 *js* 底层的实现方式, 然后说了一下和 *Object.defineProperty* 的区别

参考答案：

两者的区别总结如下：

- 代理原理：Object.defineProperty的原理是通过将数据属性转变为存取器属性的方式实现的属性读写代理。而Proxy则是因为这个内置的Proxy对象内部有一套监听机制，在传入handler对象作为参数构造代理对象后，一旦代理对象的某个操作触发，就会进入handler中对应注册的处理函数，此时我们就可以有选择的使用Reflect将操作转发被代理对象上。

- 代理局限性：Object.defineProperty始终还是局限于属性层面的读写代理，对于对象层面以及属性的其它操作代理它都无法实现。鉴于此，由于数组对象push、pop等方法的存在，它对于数组元素的读写代理实现的并不完全。而使用Proxy则可以很方便的监视数组操作。
- 自我代理：Object.defineProperty方式可以代理到自身（代理之后使用对象本身即可），也可以代理到别的对象身上（代理之后需要使用代理对象）。Proxy方式只能代理到Proxy实例对象上。这一点在其它说法中是Proxy对象不需要侵入对象就可以实现代理，实际上Object.defineProperty方式也可以不侵入。

64. 使用 *ES5* 与 *ES6* 分别实现继承

参考答案：

如果是使用 *ES5* 来实现继承，那么现在的最优解是使用圣杯模式。圣杯模式的核心思想就是不通过调用父类构造函数来给子类原型赋值，而是取得父类原型的一个副本，然后将返回的新对象赋值给子类原型。具体代码可以参阅前面第 *9* 题的解析。

ES6 新增了 *extends* 关键字，直接使用该关键字就能够实现继承。

65. 深拷贝

参考答案：

有深拷贝就有浅拷贝。

浅拷贝就是只拷贝对象的引用，而不深层次的拷贝对象的值，多个对象指向堆内存中的同一对象，任何一个修改都会使得所有对象的值修改，因为它们共用一条数据。

深拷贝不是单纯的拷贝一份引用数据类型的引用地址，而是将引用类型的值全部拷贝一份，形成一个新的引用类型，这样就不会发生引用错乱的问题，使得我们可以多次使用同样的数据，而不用担心数据之间会起冲突。

解析：

「深拷贝」就是在拷贝数据的时候，将数据的所有**引用结构**都拷贝一份。简单的说就是，在内存中存在两个数据结构完全相同又相互独立的数据，将引用型类型进行复制，而不是只复制其引用关系。

分析下怎么做「深拷贝」：

1. 首先假设深拷贝这个方法已经完成，为 deepClone
2. 要拷贝一个数据，我们肯定要去遍历它的属性，如果这个对象的属性仍是对象，继续使用这个方法，如此往复

```
function deepClone(o1, o2) {
    for (let k in o2) {
        if (typeof o2[k] === 'object') {
            o1[k] = {};
            deepClone(o1[k], o2[k]);
        } else {
            o1[k] = o2[k];
        }
    }
```

```
}
// 测试用例
let obj = {
    a: 1,
    b: [1, 2, 3],
    c: {}
};
let emptyObj = Object.create(null);
deepClone(emptyObj, obj);
console.log(emptyObj.a == obj.a);
console.log(emptyObj.b == obj.b);
```

递归容易造成爆栈，尾部调用可以解决递归的这个问题，*Chrome* 的 *V8* 引擎做了尾部调用优化，我们在写代码的时候也要注意尾部调用写法。递归的爆栈问题可以通过将递归改写成枚举的方式来解决，就是通过 *for* 或者 *while* 来代替递归。

66. *async* 与 *await* 的作用

参考答案：

async 是一个修饰符，*async* 定义的函数会默认的返回一个 *Promise* 对象 *resolve* 的值，因此对 *async* 函数可以直接进行 *then* 操作，返回的值即为 *then* 方法的传入函数。

await 关键字只能放在 *async* 函数内部， *await* 关键字的作用就是获取 *Promise* 中返回的内容， 获取的是 *Promise* 函数中 *resolve* 或者 *reject* 的值。

67. 数据的基础类型（原始类型）有哪些

参考答案：

JavaScript 中的基础数据类型，一共有 *6* 种：

string，*symbol*，*number*，*boolean*，*undefined*，*null*

其中 *symbol* 类型是在 *ES6* 里面新添加的基本数据类型。

68. *typeof null* 返回结果

参考答案：

返回 *object*

解析：至于为什么会返回 *object*，这实际上是来源于 *JavaScript* 从第一个版本开始时的一个 *bug*，并且这个 *bug* 无法被修复。修复会破坏现有的代码。

原理这是这样的，不同的对象在底层都表现为二进制，在 *JavaScript* 中二进制前三位都为 *0* 的话会被判断为 *object* 类型，*null* 的二进制全部为 *0*，自然前三位也是 *0*，所以执行 *typeof* 值会返回 *object*。

69. 对变量进行类型判断的方式有哪些

> 参考答案：
>
> 常用的方法有 *4* 种：
>
> 1. *typeof*
>
> *typeof* 是一个操作符，其右侧跟一个一元表达式，并返回这个表达式的数据类型。返回的结果用该类型的字符串(全小写字母)形式表示，包括以下 *7* 种：*number*、*boolean*、*symbol*、*string*、*object*、*undefined*、*function* 等。
>
> 1. *instanceof*
>
> *instanceof* 是用来判断 *A* 是否为 *B* 的实例，表达式为：*A instanceof B*，如果 *A* 是 *B* 的实例，则返回 *true*,否则返回 *false*。 在这里需要特别注意的是：*instanceof* 检测的是原型。
>
> 1. *constructor*
>
> 当一个函数 *F* 被定义时，*JS* 引擎会为 *F* 添加 *prototype* 原型，然后再在 *prototype* 上添加一个 *constructor* 属性，并让其指向 *F* 的引用。
>
> 1. *toString*
>
> *toString( )* 是 *Object* 的原型方法，调用该方法，默认返回当前对象的 *Class* 。这是一个内部属性，其格式为 *[object Xxx]* ，其中 *Xxx* 就是对象的类型。
>
> 对于 *Object* 对象，直接调用 *toString( )* 就能返回 *[object Object]* 。而对于其他对象，则需要通过 *call / apply* 来调用才能返回正确的类型信息。例如：

```
Object.prototype.toString.call('') ;  // [object String]
Object.prototype.toString.call(1) ;   // [object Number]
Object.prototype.toString.call(true) ;// [object Boolean]
Object.prototype.toString.call(Symbol());//[object Symbol]
Object.prototype.toString.call(undefined) ;// [object Undefined]
Object.prototype.toString.call(null) ;// [object Null]
```

70. *typeof* 与 *instanceof* 的区别？ *instanceof* 是如何实现?

> 参考答案：
>
> 1. *typeof*
>
> *typeof* 是一个操作符，其右侧跟一个一元表达式，并返回这个表达式的数据类型。返回的结果用该类型的字符串(全小写字母)形式表示，包括以下 *7* 种：*number*、*boolean*、*symbol*、*string*、*object*、*undefined*、*function* 等。
>
> 1. *instanceof*
>
> *instanceof* 是用来判断 *A* 是否为 *B* 的实例，表达式为：*A instanceof B*，如果 *A* 是 *B* 的实例，则返回 *true*,否则返回 *false*。 在这里需要特别注意的是：*instanceof* 检测的是原型。
>
> 用一段伪代码来模拟其内部执行过程：

```
instanceof (A,B) = {
    varL = A.__proto__;
    varR = B.prototype;
    if(L === R) {
        // A的内部属性 __proto__ 指向 B 的原型对象
        return true;
    }
    return false;
}
```

从上述过程可以看出，当 *A* 的 ***proto*** 指向 *B* 的 *prototype* 时，就认为 *A* 就是 *B* 的实例。

需要注意的是，*instanceof* 只能用来判断两个对象是否属于实例关系， 而不能判断一个对象实例具体属于哪种类型。

例如： *[ ] instanceof Object* 返回的也会是 *true*。

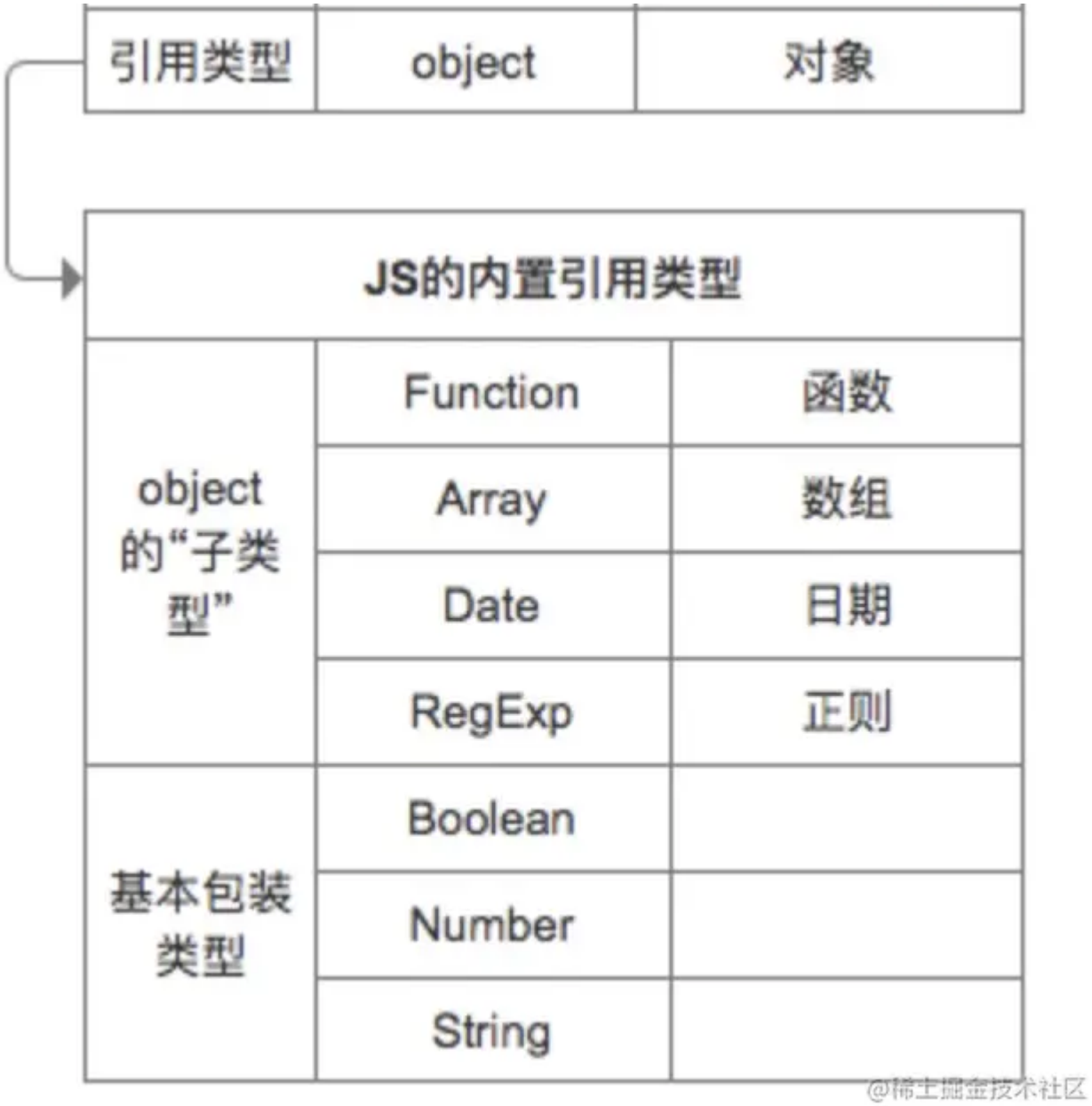
71. 引用类型有哪些，有什么特点

参考答案：

JS 中七种内置类型（*null*，*undefined*，*boolean*，*number*，*string*，*symbol*，*object*）又分为两大类型

两大类型：

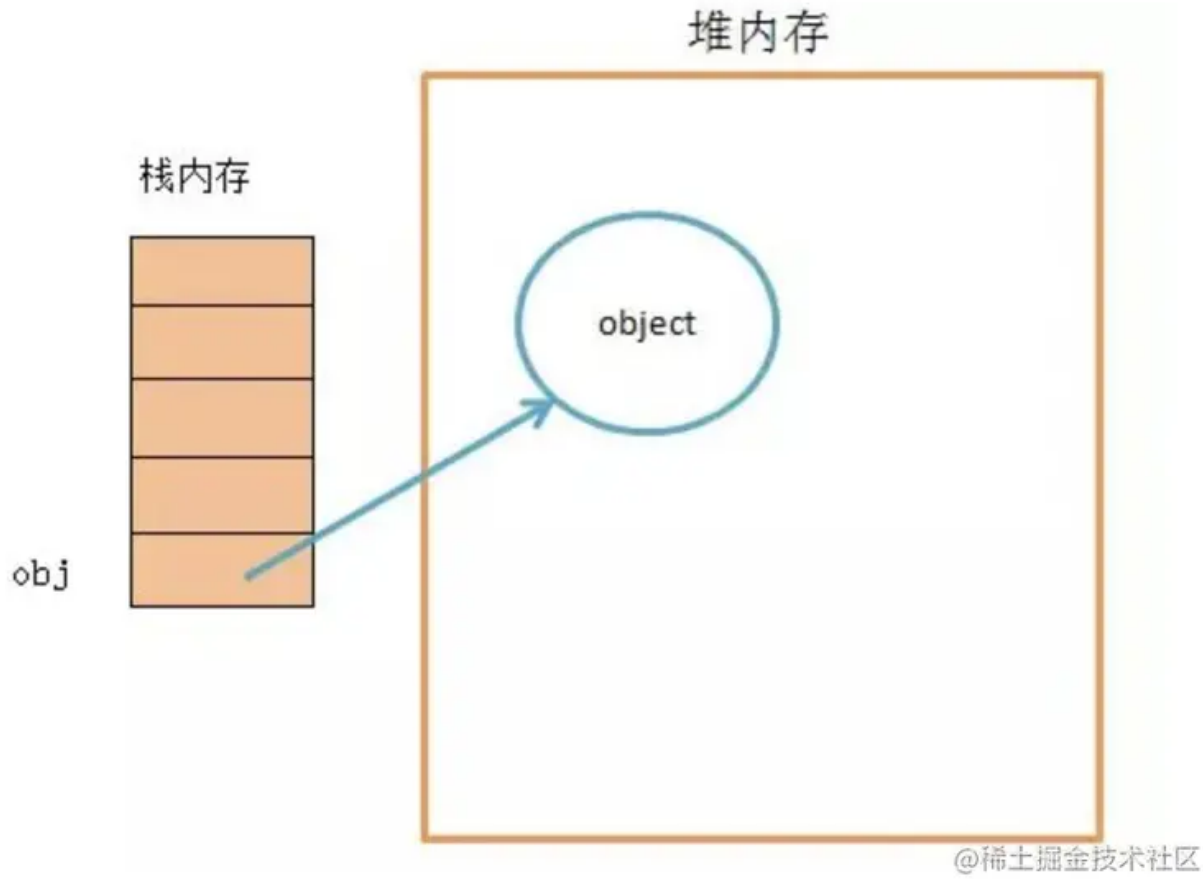
- 基本类型： `null`， `undefined`， `boolean`， `number`， `string`， `symbol`
- 引用类型Object： `Array` ， `Function`， `Date`， `RegExp`等



基本类型和引用类型的主要区别有以下几点：

存放位置：

- 基本数据类型：基本类型值在内存中占据固定大小，直接存储在**栈内存**中的数据
- 引用数据类型：引用类型在栈中存储了指针，这个指针指向堆内存中的地址，真实的数据存放在**堆内存**里。



值的可变性：

- 基本数据类型： 值不可变，*javascript* 中的原始值（*undefined*、*null*、布尔值、数字和字符串）是不可更改的
- 引用数据类型：引用类型是可以直接改变其值的

比较：

- 基本数据类型： 基本类型的比较是值的比较，只要它们的值相等就认为他们是相等的
- 引用数据类型： 引用数据类型的比较是引用的比较，看其的引用是否指向同一个对象

72. 如何得到一个变量的类型---指函数封装实现

参考答案：

请参阅前面第 *30* 题答案。

73. 什么是作用域、闭包

参考答案：

请参阅前面第 *56* 题。

74. 闭包的缺点是什么？闭包的应用场景有哪些？怎么销毁闭包?

> 参考答案：
>
> 闭包是指有权访问另外一个函数作用域中的变量的函数。
>
> 因为闭包引用着另一个函数的变量，导致另一个函数已经不使用了也无法销毁，所以**闭包使用过多，会占用较多的内存，这也是一个副作用，内存泄漏。**
>
> 如果要销毁一个闭包，可以 把被引用的变量设置为*null*，即手动清除变量，这样下次 *js* 垃圾回收机制回收时，就会把设为 *null* 的量给回收了。
>
> 闭包的应用场景：
>
> 1. 匿名自执行函数
> 2. 结果缓存
> 3. 封装
> 4. 实现类和继承

75. *JS*的垃圾回收站机制

> 参考答案：
>
> *JS* 具有自动垃圾回收机制。垃圾收集器会按照固定的时间间隔周期性的执行。
>
> *JS* 常见的垃圾回收方式：标记清除、引用计数方式。
>
> 1、标记清除方式：
>
> - 工作原理：当变量进入环境时，将这个变量标记为“进入环境”。当变量离开环境时，则将其标记为“离开环境”。标记“离开环境”的就回收内存。
> - 工作流程：
> - 垃圾回收器，在运行的时候会给存储在内存中的所有变量都加上标记；
> - 去掉环境中的变量以及被环境中的变量引用的变量的标记；
> - 被加上标记的会被视为准备删除的变量；
> - 垃圾回收器完成内存清理工作，销毁那些带标记的值并回收他们所占用的内存空间。
>
> 2、引用计数方式：
>
> - 工作原理：跟踪记录每个值被引用的次数。
> - 工作流程：
> - 声明了一个变量并将一个引用类型的值赋值给这个变量，这个引用类型值的引用次数就是 *1*；
> - 同一个值又被赋值给另一个变量，这个引用类型值的引用次数加1；
> - 当包含这个引用类型值的变量又被赋值成另一个值了，那么这个引用类型值的引用次数减 *1*；
> - 当引用次数变成 *0* 时，说明没办法访问这个值了；
> - 当垃圾收集器下一次运行时，它就会释放引用次数是0的值所占的内存。

76. 什么是作用域链、原型链

> 参考答案：
>
> **什么是作用域链?**
>
> 当访问一个变量时，编译器在执行这段代码时，会首先从当前的作用域中查找是否有这个标识符，如果没有找到，就会去父作用域查找，如果父作用域还没找到继续向上查找，直到全局作用域为止,，而作用域链，就是有当前作用域与上层作用域的一系列变量对象组成，它保证了当前执行的作用域对符合访问权限的变量和函数的有序访问。
>
> **什么原型链?**
>
> 每个对象都可以有一个原型***proto***，这个原型还可以有它自己的原型，以此类推，形成一个原型链。查找特定属性的时候，我们先去这个对象里去找，如果没有的话就去它的原型对象里面去，如果还是没有的话再去向原型对象的原型对象里去寻找。这个操作被委托在整个原型链上，这个就是我们说的原型链。

77. *new* 一个构造函数发生了什么

> 参考答案：
>
> *new* 运算符创建一个用户定义的对象类型的实例或具有构造函数的内置对象的实例。
>
> *new* 关键字会进行如下的操作：
> 步骤 *1*：创建一个空的简单 *JavaScript* 对象，即 { }；
> 步骤 *2*：链接该对象到另一个对象（即设置该对象的原型对象）；
> 步骤 *3*：将步骤 *1* 新创建的对象作为 *this* 的上下文；
> 步骤 *4*：如果该函数没有返回对象，则返回 *this*。

78. 对一个构造函数实例化后. 它的原型链指向什么

> 参考答案：
>
> 指向该构造函数实例化出来对象的原型对象。
>
> 对于构造函数来讲，可以通过 *prototype* 访问到该对象。
>
> 对于实例对象来讲，可以通过隐式属性 ***proto*** 来访问到。

79. 什么是变量提升

> 参考答案：
>
> 当 *JavaScript* 编译所有代码时，所有使用 *var* 的变量声明都被提升到它们的函数/局部作用域的顶部(如果在函数内部声明的话)，或者提升到它们的全局作用域的顶部(如果在函数外部声明的话)，而不管实际的声明是在哪里进行的。这就是我们所说的“提升”。
>
> 请记住，这种“提升”实际上并不发生在你的代码中，而只是一种比喻，与 *JavaScript* 编译器如何读取你的代码有关。记住当我们想到“提升”的时候，我们可以想象任何被提升的东西都会被移动到顶部，但是实际上你的代码并不会被修改。
>
> 函数声明也会被提升，但是被提升到了最顶端，所以将位于所有变量声明之上。
>
> 在编译阶段变量和函数声明会被放入内存中，但是你在代码中编写它们的位置会保持不变。

80. == 和 === 的区别是什么

> 参考答案：
>
> 简单来说： == 代表相同， === 代表严格相同（数据类型和值都相等）。
>
> 当进行双等号比较时候，先检查两个操作数数据类型，如果相同，则进行===比较，如果不同，则愿意为你进行一次类型转换，转换成相同类型后再进行比较，而 === 比较时，如果类型不同，直接就是false。
>
> 从这个过程来看，大家也能发现，某些情况下我们使用 === 进行比较效率要高些，因此，没有歧义的情况下，不会影响结果的情况下，在 *JS* 中首选 === 进行逻辑比较。

81. *Object.is* 方法比较的是什么

> 参考答案：
>
> *Object.is* 方法是 *ES6* 新增的用来比较两个值是否严格相等的方法，与 === (严格相等)的行为基本一致。不过有两处不同：
>
> - +0 不等于 -0。
> - *NaN* 等于自身。
>
> 所以可以将*Object.is* 方法看作是加强版的严格相等。

82. 基础数据类型和引用数据类型，哪个是保存在栈内存中？哪个是在堆内存中?

> 参考答案：
>
> 在 *ECMAScript* 规范中，共定义了 *7* 种数据类型，分为 **基本类型** 和 **引用类型** 两大类，如下所示：
>
> - **基本类型**：*String*、*Number*、*Boolean*、*Symbol*、*Undefined*、*Null*
> - **引用类型**：*Object*
>
> 基本类型也称为简单类型，由于其占据空间固定，是简单的数据段，为了便于提升变量查询速度，将其存储在栈中，即按值访问。
>
> 引用类型也称为复杂类型，由于其值的大小会改变，所以不能将其存放在栈中，否则会降低变量查询速度，因此，其值存储在堆(*heap*)中，而存储在变量处的值，是一个指针，指向存储对象的内存处，即按址访问。引用类型除 *Object* 外，还包括 *Function* 、*Array*、*RegExp*、*Date* 等等。

83. 箭头函数解决了什么问题?

> 参考答案：
>
> 箭头函数主要解决了 *this* 的指向问题。

> 解析：
>
> 在 *ES5* 时代，一旦对象的方法里面又存在函数，则 *this* 的指向往往会让开发人员抓狂。
>
> 例如：

```
//错误案例，this 指向会指向 Windows 或者 undefined
var obj = {
 age: 18,
 getAge: function () {
     var a = this.age; // 18
     var fn = function () {
         return new Date().getFullYear() - this.age; // this 指向 window 或
undefined
     };
     return fn();
 }
};
console.log(obj.getAge()); // NaN
```

然而，箭头函数没有 *this*，箭头函数的 *this* 是继承父执行上下文里面的 *this*

```
var obj = {
 age: 18,
 getAge: function () {
     var a = this.age; // 18
     var fn = () => new Date().getFullYear() - this.age; // this 指向 obj 对象
     return fn();
 }
};

console.log(obj.getAge()); // 2003
```

84. *new* 一个箭头函数后，它的 *this* 指向什么?

参考答案：

我不知道这道题是出题人写错了还是故意为之。

箭头函数无法用来充当构造函数，所以是无法 *new* 一个箭头函数的。

当然，也有可能是面试官故意挖的一个坑，等着你往里面跳。

85. *promise* 的其他方法有用过吗？如 *all*、*race*。请说下这两者的区别

参考答案：

promise.all 方法参数是一个 *promise* 的数组,只有当所有的 *promise* 都完成并返回成功，才会调用 *resolve*，当有一个失败，都会进*catch*，被捕获错误，*promise.all* 调用成功返回的结果是每个 *promise* 单独调用成功之后返回的结果组成的数组,如果调用失败的话，返回的则是第一个 *reject* 的结果

promise.race 也会调用所有的 *promise*，返回的结果则是所有 *promise* 中最先返回的结果，不关心是成功还是失败。

86. *class* 是如何实现的

参考答案：

class 是 *ES6* 新推出的关键字，它是一个语法糖，本质上就是基于这个原型实现的。只不过在以前 *ES5* 原型实现的基础上，添加了一些 *classCallCheck*、*defineProperties*、*_createClass*等方法来做出了一些特殊的处理。

例如：

```
class Hello {
constructor(x) {
    this.x = x;
}
greet() {
    console.log("Hello, " + this.x)
}
}
"use strict";

function _classCallCheck(instance, Constructor) {
  if (!(instance instanceof Constructor)) {
      throw new TypeError("Cannot call a class as a function");
  }
}

function _defineProperties(target, props) {
  for (var i = 0; i < props.length; i++) {
      var descriptor = props[i];
      descriptor.enumerable = descriptor.enumerable || false;
      descriptor.configurable = true;
      if ("value" in descriptor)
          descriptor.writable = true;
      Object.defineProperty(target, descriptor.key, descriptor);
  }
}

function _createClass(Constructor, protoProps, staticProps) {
  console.log("Constructor::",Constructor);
  console.log("protoProps::",protoProps);
  console.log("staticProps::",staticProps);
  if (protoProps)
      _defineProperties(Constructor.prototype, protoProps);
  if (staticProps)
      _defineProperties(Constructor, staticProps);
  return Constructor;
}

var Hello = /*#__PURE__*/function () {
function Hello(x) {
    _classCallCheck(this, Hello);
```

```
    this.x = x;
}

_createClass(Hello, [{
    key: "greet",
    value: function greet() {
      console.log("Hello, " + this.x);
    }
  }]);

  return Hello;
}();
```

87. *let*、*const*、*var* 的区别

参考答案：

请参阅前面第 *22* 题答案。

88. *ES6* 中模块化导入和导出与 *common.js* 有什么区别

参考答案：

CommonJs模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化不会影响到这个值.

```
// common.js
var count = 1;

var printCount = () =>{
return ++count;
}

module.exports = {
 printCount: printCount,
 count: count
};
// index.js
let v = require('./common');
console.log(v.count); // 1
console.log(v.printCount()); // 2
console.log(v.count); // 1
```

你可以看到明明common.js里面改变了count，但是输出的结果还是原来的。这是因为count是一个原始类型的值，会被缓存。除非写成一个函数，才能得到内部变动的值。将common.js里面的module.exports 改写成

```
module.exports = {
 printCount: printCount,
 get count(){
     return count
 }
};
```

这样子的输出结果是 1，2，2

而在ES6当中，写法是这样的，是利用export 和import导入的

```
// es6.js
export let count = 1;
export function printCount() {
 ++count;
}
// main1.js
import  { count, printCount } from './es6';
console.log(count)
console.log(printCount());
console.log(count)
```

ES6 模块是动态引用，并且不会缓存，模块里面的变量绑定其所有的模块，而是动态地去加载值，并且不能重新赋值，

ES6 输入的模块变量，只是一个“符号连接符”，所以这个变量是只读的，对它进行重新赋值会报错。如果是引用类型，变量指向的地址是只读的，但是可以为其添加属性或成员。

另外还想说一个 *export default*

```
let count = 1;
function printCount() {
 ++count;
}
export default { count, printCount}
// main3.js
import res form './main3.js'
console.log(res.count)
```

export与export default的区别及联系：

1. export与export default均可用于导出常量、函数、文件、模块等
2. 你可以在其它文件或模块中通过 import + (常量 | 函数 | 文件 | 模块)名的方式，将其导入，以便能够对其进行使用
3. 在一个文件或模块中，export、import可以有多个，export default仅有一个
4. 通过export方式导出，在导入时要加{ }，export default则不需要。

89. 说一下普通函数和箭头函数的区别

> 参考答案：
>
> 请参阅前面第 *8*、*25*、*83* 题答案。

90. 说一下 *promise* 和 *async* 和 *await* 什么关系

> 参考答案：
>
> *await* 表达式会造成异步函数停止执行并且等待*promise*的解决，当值被*resolved*，异步函数会恢复执行以及返回*resolved*值。如果该值不是一个*promise*，它将会被转换成一个*resolved*后的*promise*。如果*promise*被*rejected*，*await* 表达式会抛出异常值。

91. 说一下你学习过的有关 *ES6* 的知识点

> 参考答案：
>
> 这种题目是开放题，可以简单列举一下 *ES6* 的新增知识点。（ *ES6* 的新增知识点参阅前面第 *44* 题）
>
> 然后说一下自己平时开发中用得比较多的是哪些即可。
>
> 一般面试官会针对你所说的内容进行二次提问。例如：你回答平时开发中箭头函数用得比较多，那么面试官极大可能针对箭头函数展开二次提问，询问你箭头函数有哪些特性？箭头函数 *this* 特点之类的问题。

92. 了解过 *js* 中 *arguments* 吗？接收的是实参还是形参?

> 参考答案：
>
> *JS* 中的 *arguments* 是一个伪数组对象。这个伪数组对象将包含调用函数时传递的所有的实参。
>
> 与之相对的，*JS* 中的函数还有一个 *length* 属性，返回的是函数形参的个数。

93. *ES6* 相比于 *ES5* 有什么变化

> 参考答案：
>
> *ES6* 相比 *ES5* 新增了很多新特性，这里可以自己简述几个。
>
> 具体的新增特性可以参阅前面第 *44* 题。

94. 强制类型转换方法有哪些?

> 参考答案：
>
> JavaScript 中的数据类型转换，主要有三种方式：
>
> 1. 转换函数
>
> *js* 提供了诸如 *parseInt* 和 *parseFloat* 这些转换函数，通过这些转换函数可以进行数据类型的转换 。
>
> 1. 强制类型转换
>
> 还可使用强制类型转换（*type casting*）处理转换值的类型。

例如：

- *Boolean*(*value*) 把给定的值转换成 *Boolean* 型；
- *Number*(*value*)——把给定的值转换成数字（可以是整数或浮点数）；
- *String*(*value*)——把给定的值转换成字符串。

1. 利用 *js* 变量弱类型转换。

例如：

- 转换字符串：直接和一个空字符串拼接，例如：`a = "" + 数据`
- 转换布尔：!!数据类型，例如：`!!"Hello"`
- 转换数值：数据*1 或 /1，例如：`"Hello * 1"`

95. 纯函数

参考答案：

一个函数，如果符合以下两个特点，那么它就可以称之为**纯函数**：

1. 对于相同的输入，永远得到相同的输出
2. 没有任何可观察到的副作用

解析：

针对上面的两个特点，我们一个一个来看。

- 相同输入得到相同输出

我们先来看一个不纯的反面典型：

```
let greeting = 'Hello'

function greet (name) {
  return greeting + ' ' + name
}

console.log(greet('World')) // Hello World
```

上面的代码中，*greet('World')* 是不是永远返回 *Hello World*？显然不是，假如我们修改 *greeting* 的值，就会影响 *greet* 函数的输出。即函数 *greet* 其实是 **依赖外部状态** 的。

那我们做以下修改：

```
function greet (greeting, name) {
  return greeting + ' ' + name
}

console.log(greet('Hi', 'Savo')) // Hi Savo
```

将 *greeting* 参数也传入，这样对于任何输入参数，都有与之对应的唯一的输出参数了，该函数就符合了第一个特点。

- 没有副作用

副作用的意思是，这个函数的运行，**不会修改外部的状态**。

下面再看反面典型：

```
const user = {
  username: 'savokiss'
}

let isValid = false

function validate (user) {
  if (user.username.length > 4) {
    isValid = true
  }
}
```

可见，执行函数的时候会修改到 *isValid* 的值（注意：如果你的函数没有任何返回值，那么它很可能就具有副作用！）

那么我们如何移除这个副作用呢？其实不需要修改外部的 *isValid* 变量，我们只需要在函数中将验证的结果 *return* 出来：

```
const user = {
  username: 'savokiss'
}

function validate (user) {
  return user.username.length > 4;
}

const isValid = validate(user)
```

这样 *validate* 函数就不会修改任何外部的状态了~

96. *JS* 模块化

参考答案：

模块化主要是用来抽离公共代码，隔离作用域，避免变量冲突等。

模块化的整个发展历史如下：

IIFE：使用自执行函数来编写模块化，特点：**在一个单独的函数作用域中执行代码，避免变量冲突。**

```
(function(){
return {
  data:[]
}
})()
```

AMD： 使用requireJS 来编写模块化，特点：**依赖必须提前声明好**。

```
define('./index.js',function(code){
  // code 就是index.js 返回的内容
})
```

CMD： 使用seaJS 来编写模块化，特点：**支持动态引入依赖文件**。

```
define(function(require, exports, module) {
var indexCode = require('./index.js');
});
```

CommonJS： nodejs 中自带的模块化。

```
var fs = require('fs');
```

UMD：兼容AMD，CommonJS 模块化语法。

webpack(require.ensure)：webpack 2.x 版本中的代码分割。

ES Modules： ES6 引入的模块化，支持import 来引入另一个 js 。

```
import a from 'a';
```

97. 看过 *jquery* 源码吗?

参考答案：

开放题，但是需要注意的是，如果看过 *jquery* 源码，不要简单的回答一个“看过”就完了，应该继续乘胜追击，告诉面试官例如哪个哪个部分是怎么怎么实现的，并针对这部分的源码实现，可以发表一些自己的看法和感想。

98. 说一下 *js* 中的 *this*

参考答案：

请参阅前面第 *17* 题答案。

99. *apply call bind* 区别，手写

参考答案：

apply call bind 区别？

call 和 *apply* 的功能相同，区别在于传参的方式不一样:

- *fn.call(obj, arg1, arg2, ...)* 调用一个函数, 具有一个指定的 *this* 值和分别地提供的参数(参数的列表)。
- *fn.apply(obj, [argsArray])* 调用一个函数，具有一个指定的 *this* 值，以及作为一个数组（或类数组对象）提供的参数。

bind 和 *call/apply* 有一个很重要的区别，一个函数被 *call/apply* 的时候，会直接调用，但是 *bind* 会创建一个新函数。当这个新函数被调用时，*bind( )* 的第一个参数将作为它运行时的 *this*，之后的一序列参数将会在传递的实参前传入作为它的参数。

实现 *call* 方法：

```
Function.prototype.call2 = function (context) {
    //没传参数或者为 null 是默认是 window
    var context = context || (typeof window !== 'undefined' ? window : global)
    // 首先要获取调用 call 的函数，用 this 可以获取
    context.fn = this
    var args = []
    for (var i = 1; i < arguments.length; i++) {
        args.push('arguments[' + i + ']')
    }
    eval('context.fn(' + args + ')')
    delete context.fn
}

// 测试
var value = 3
var foo = {
    value: 2
}

function bar(name, age) {
    console.log(this.value)
    console.log(name)
    console.log(age)
}
bar.call2(null)
// 浏览器环境:  3 undefinde undefinde
// Node环境: undefinde undefinde undefinde

bar.call2(foo, 'cc', 18) // 2  cc 18
```

实现 *apply* 方法：

```
Function.prototype.apply2 = function (context, arr) {
    var context = context || (typeof window !== 'undefined' ? window : global)
    context.fn = this;

    var result;
    if (!arr) {
        result = context.fn();
    }
    else {
        var args = [];
        for (var i = 0, len = arr.length; i < len; i++) {
            args.push('arr[' + i + ']');
        }
        result = eval('context.fn(' + args + ')')
    }

    delete context.fn
    return result;
}

// 测试:

var value = 3
var foo = {
    value: 2
}

function bar(name, age) {
    console.log(this.value)
    console.log(name)
    console.log(age)
}
bar.apply2(null)
// 浏览器环境:  3 undefinde undefinde
// Node环境: undefinde undefinde undefinde

bar.apply2(foo, ['cc', 18]) // 2  cc 18
```

实现 *bind* 方法:

```
Function.prototype.bind2 = function (oThis) {
    if (typeof this !== "function") {
        // closest thing possible to the ECMAScript 5 internal IsCallable function
        throw new TypeError("Function.prototype.bind - what is trying to be bound is
not callable");
    }
    var aArgs = Array.prototype.slice.call(arguments, 1),
        fToBind = this,
```

```
        fNOP = function () { },
        fBound = function () {
            return fToBind.apply(this instanceof fNOP && oThis
                ? this
                : oThis || window,
                aArgs.concat(Array.prototype.slice.call(arguments)));
        };

    fNOP.prototype = this.prototype;
    fBound.prototype = new fNOP();

    return fBound;
}

// 测试
var test = {
    name: "jack"
}
var demo = {
    name: "rose",
    getName: function () { return this.name; }
}

console.log(demo.getName()); // 输出 rose  这里的 this 指向 demo

// 运用 bind 方法更改 this 指向
var another2 = demo.getName.bind2(test);
console.log(another2()); // 输出 jack  这里 this 指向了 test 对象了
```

100. 手写 *reduce flat*

参考答案：

reduce 实现：

```
Array.prototype.my_reduce = function (callback, initialValue) {
 if (!Array.isArray(this) || !this.length || typeof callback !== 'function') {
     return []
 } else {
     // 判断是否有初始值
     let hasInitialValue = initialValue !== undefined;
     let value = hasInitialValue ? initialValue : tihs[0];
     for (let index = hasInitialValue ? 0 : 1; index < this.length; index++) {
         const element = this[index];
         value = callback(value, element, index, this)
     }
     return value
 }
}
```

```
let arr = [1, 2, 3, 4, 5]
let res = arr.my_reduce((pre, cur, i, arr) => {
 console.log(pre, cur, i, arr)
 return pre + cur
}, 10)
console.log(res)//25
```

flat 实现：

```
let arr = [1, [2, 3, [4, 5, [12, 3, "zs"], 7, [8, 9, [10, 11, [1, 2, [3, 4]]]]]]];

//万能的类型检测方法
const checkType = (arr) => {
 return Object.prototype.toString.call(arr).slice(8, -1);
}
//自定义flat方法，注意：不可以使用箭头函数，使用后内部的this会指向window
Array.prototype.myFlat = function (num) {
 //判断第一层数组的类型
 let type = checkType(this);
 //创建一个新数组，用于保存拆分后的数组
 let result = [];
 //若当前对象非数组则返回undefined
 if (!Object.is(type, "Array")) {
     return;
 }
 //遍历所有子元素并判断类型，若为数组则继续递归，若不为数组则直接加入新数组
 this.forEach((item) => {
     let cellType = checkType(item);
     if (Object.is(cellType, "Array")) {
         //形参num，表示当前需要拆分多少层数组，传入Infinity则将多维直接降为一维
         num--;
         if (num < 0) {
             let newArr = result.push(item);
             return newArr;
         }
         //使用三点运算符解构，递归函数返回的数组，并加入新数组
         result.push(...item.myFlat(num));
     } else {
         result.push(item);
     }
 })
 return result;
}
console.time();

console.log(arr.flat(Infinity)); //[1, 2, 3, 4, 5, 12, 3, "zs", 7, 8, 9, 10, 11, 1,
2, 3, 4];
```

```
console.log(arr.myFlat(Infinity)); //[1, 2, 3, 4, 5, 12, 3, "zs", 7, 8, 9, 10, 11, 1, 2, 3, 4];
//自定义方法和自带的flat返回结果一致!!!!
console.timeEnd();
```

注： 由于字数限制，剩余内容在下篇进行总结哦。🤞，大家认真看哦，奥利给！💪