```
            }
        }
    }
    return arr;
}

console.log(bubbleSort(arr));
```

4 快速排序

- 采用二分法， 取出中间数，数组每次和中间数比较，小的放到左边，大的放到右边

```js
var arr = [3, 1, 4, 6, 5, 7, 2];

function quickSort(arr) {
    if(arr.length == 0) {
        return [];    // 返回空数组
    }

    var cIndex = Math.floor(arr.length / 2);
    var c = arr.splice(cIndex, 1);
    var l = [];
    var r = [];

    for (var i = 0; i < arr.length; i++) {
        if(arr[i] < c) {
            l.push(arr[i]);
        } else {
            r.push(arr[i]);
        }
    }

    return quickSort(l).concat(c, quickSort(r));
}

console.log(quickSort(arr));
```

5 编写一个方法 求一个字符串的字节长度

- 假设：一个英文字符占用一个字节，一个中文字符占用两个字节

```js
function GetBytes(str) {
        var len = str.length;
        var bytes = len;
        for(var i=0; i<len; i++){
            if (str.charCodeAt(i) > 255) bytes++;
        }
        return bytes;
    }
alert(GetBytes("你好,as"));
```

6 bind的用法， 以及如何实现bind的函数和需要注意的点

- bind 的作用与 call 和 apply 相同， 区别是 call 和 apply 是立即调用函数， 而 bind 是返回了一个函数， 需要调用的时候再执行 。 一个简单的 bind 函数实现如下

```js
Function.prototype.bind = function(ctx) {
    var fn = this;
    return function() {
        fn.apply(ctx, arguments);
    };
};
```

7 实现一个函数clone

> 可以对 JavaScript 中的5种主要的数据类型,包括 Number 、 String 、 Object 、 Array 、 Boolean ）进行值复

- 考察点1：对于基本数据类型和引用数据类型在内存中存放的是值还是指针这一区别是否清楚
- 考察点2：是否知道如何判断一个变量是什么类型的

考察点3：递归算法的设计

```js
// 方法一：
  Object.prototype.clone = function(){
          var o = this.constructor === Array ? [] : {};
          for(var e in this){
                  o[e] = typeof this [e] === "object" ? this [e].clone() : th
          }
          return o;
  }

 //方法二：
   /**
       * 克隆一个对象
       * @param Obj
       * @returns
       */
      function clone(Obj) {
          var buf;
          if (Obj instanceof Array) {
              buf = [];                    //创建一个空的数组
              var i = Obj.length;
              while (i--) {
                  buf [i] = clone(Obj [i]);
              }
              return buf;
          }else if (Obj instanceof Object){
              buf = {};                    //创建一个空对象
              for (var k in Obj) {            //为这个对象添加新的属性
                  buf [k] = clone(Obj [k]);
              }
              return buf;
          }else{                           //普通变量直接赋值
              return Obj;
          }
      }
```

8 下面这个ul， 如何点击每一列的时候alert其index

> 考察闭包

```html
<ul id="test">
    <li>这是第一条</li>
    <li>这是第二条</li>
    <li>这是第三条</li>
</ul>
```

```js
// 方法一:
var lis=document.getElementById( '2223').getElementsByTagName( 'li');
for(var i=0;i<3;i++)
{
    lis [i].index=i;
    lis [i].onclick=function(){
        alert(this.index);
}

//方法二:
var lis=document.getElementById( '2223').getElementsByTagName( 'li');
for(var i=0;i<3;i++)
{
    lis [i].index=i;
    lis [i].onclick=(function(a){
        return function() {
            alert(a);
        }
    })(i);
}
```

9 定义一个log方法，让它可以代理console.log的方法

```js
可行的方法一:

function log(msg) {
    console.log(msg);
}

log("hello world!") // hello world!
```

> 如果要传入多个参数呢？显然上面的方法不能满足要求，所以更好的方法是:

```js
function log(){
    console.log.apply(console, arguments);
};
```

10 输出今天的日期

> 以 YYYY-MM-DD 的方式， 比如今天是2014年9月26日，则输出2014-09-26

```js
var d = new Date();
 // 获取年， getFullYear()返回4位的数字
 var year = d.getFullYear();
 // 获取月， 月份比较特殊， 0是1月， 11是12月
 var month = d.getMonth() + 1;
 // 变成两位
 month = month < 10 ? '0' + month : month;
 // 获取日
 var day = d.getDate();
day = day < 10 ? '0' + day : day;
alert(year + '- ' + month + '- ' + day);
```

11 用js实现随机选取10– 100之间的10个数字，存入一个数组， 并排序

```js
var iArray = [];
 funtion getRandom(istart, iend){
        var iChoice = istart - iend +1;
        return Math.floor(Math.random() * iChoice + istart;
 }
 for(var i=0; i<10; i++){
        iArray.push(getRandom(10,100));
 }
 iArray.sort();
```

12 写一段JS程序提取URL中的各个GET参数

> 有这样一个 URL ： http://item.taobao.com/item.htm?a=1&b=2&c=&d=xxx&e ，请写一段JS程序提取URL中的各个GET参数(参数名和

> 参数个数不确定)，将其按 key-value 形式返回到一个 json 结构中， 如 {a:'1', b:'2', c:'', d:'xxx', e:undefined}

```js
function serilizeUrl(url) {
    var result = {};
    url = url.split("?") [1];
    var map = url.split("&");
    for(var i = 0, len = map.length; i < len; i++) {
        result[map[i].split("=") [0]] = map[i].split("=") [1];
    }
    return result;
}
```

13 写一个 function ， 清除字符串前后的空格

> 使用自带接口 trim() ，考虑兼容性：

```js
if ( !String.prototype.trim) {
    String.prototype.trim = function() {
        return this.replace(/^\s+/, "").replace(/\s+$/,"");
    }
}

// test the function
var str = " \t\n test string ".trim();
alert(str == "test string"); // alerts "true"
```

14 实现每隔一秒钟输出1,2,3...数字

```js
for(var i=0;i<10;i++){
  (function(j){
    setTimeout(function(){
      console.log(j+1)
    },j*1000)
  })(i)
}
```

15 实现一个函数， 判断输入是不是回文字符串

```js
function run( input)  {
  if (typeof input !== 'string') return false;
  return input.split( '').reverse().join( '') === input;
}
```

16 、数组扁平化处理

> 实现一个 flatten 方法，使得输入一个数组，该数组里面的元素也可以是数组，该方法会输出一个扁平化的数组

```js
function flatten(arr){
    return arr.reduce(function(prev,item){
        return prev.concat(Array.isArray(item)?flatten(item):item);
    }, []);
}
```

九、其他

1 负载均衡

> 多台服务器共同协作，不让其中某一台或几台超额工作，发挥服务器的最大作用

- http 重定向负载均衡：调度者根据策略选择服务器以302响应请求， 缺点只有第一次有效果，后续操作维持在该服务器 dns负载均衡：解析域名时，访问多个 ip 服务器中的一个(可监控性较弱)
- 反向代理负载均衡：访问统一的服务器， 由服务器进行调度访问实际的某个服务器，对统一的服务器要求大，性能受到 服务器群的数量

2 CDN

> 内容分发网络， 基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快 、更稳定。

3 内存泄漏

> 定义：程序中已动态分配的堆内存由于某种原因程序未释放或无法释放引发的各种问题。

js中可能出现的内存泄漏情况

> 结果：变慢， 崩溃，延迟大等，原因：

- 全局变量
- `dom` 清空时， 还存在引用
- `ie` 中使用闭包
- 定时器未清除
- 子元素存在引起的内存泄露

避免策略

- 减少不必要的全局变量， 或者生命周期较长的对象，及时对无用的数据进行垃圾回收；
- 注意程序逻辑， 避免“死循环”之类的；
- 避免创建过多的对象 原则：不用了的东西要及时归还。
- 减少层级过多的引用

4 babel原理

> `ES6、7` 代码输入 -> `babylon` 进行解析 -> 得到 `AST` （抽象语法树） -> `plugin` 用b `abel-traverse` 对 `AST` 树进行遍历转译 ->得到新的 `AST` 树-> 用 `babel-generator` 通过 `AST` 树生成 `ES5` 代码

5 js自定义事件

> 三要素： `document.createEvent()` `event.initEvent()` `element.dispatchEvent()`

```js
// (en:自定义事件名称， fn:事件处理函数， addEvent:为DOM元素添加自定义事件， triggerEv
window.onload = function(){
    var demo = document.getElementById("demo");
    demo.addEvent("test",function(){console.log("handler1")});
    demo.addEvent("test",function(){console.log("handler2")});
    demo.onclick = function(){
        this.triggerEvent("test");
    }
}
Element.prototype.addEvent = function(en,fn){
    this.pools = this.pools || {};
    if(en in this.pools){
        this.pools [en].push(fn);
    }else{
        this.pools [en] = [];
        this.pools [en].push(fn);
    }
}
Element.prototype.triggerEvent  = function(en){
    if(en in this.pools){
        var fns = this.pools [en];
        for(var i=0,il=fns.length;i<il;i++){
            fns [i]();
        }
    }else{
        return;
    }
}
```

6 前后端路由差别

- 后端每次路由请求都是重新访问服务器
- 前端路由实际上只是 JS 根据 URL 来操作 DOM 元素，根据每个页面需要的去服务端请求数据， 返回数据后和模板进行组合

十、综合

1 谈谈你对重构的理解

- 网站重构：在不改变外部行为的前提下， 简化结构 、添加可读性， 而在网站前端保持一致的行为 。也就是说是在不改变UI的情况下， 对网站进行优化， 在扩展的同时保持一致的UI
- 对于传统的网站来说重构通常是：

  表格(`table`)布局改为 `DIV+CSS`

  使网站前端兼容于现代浏览器(针对于不合规范的 `CSS` 、如对Im6有效的)

  对于移动平台的优化
  针对于 `SEO` 进行优化

2 什么样的前端代码是好的

- 高复用低耦合， 这样文件小， 好维护， 而且好扩展。
- 具有可用性 、健壮性 、可靠性 、宽容性等特点
- 遵循设计模式的六大原则

3 对前端工程师这个职位是怎么样理解的？ 它的前景会怎么样

- 前端是最贴近用户的程序员， 比后端 、数据库 、产品经理 、运营 、安全都近

  实现界面交互
  提升用户体验

  基于NodeJS， 可跨平台开发

- 前端是最贴近用户的程序员， 前端的能力就是能让产品从 90分进化到 100 分， 甚至更好，
- 与团队成员， `UI` 设计，产品经理的沟通；
- 做好的页面结构， 页面重构和用户体验；

4 你觉得前端工程的价值体现在哪

- 为简化用户使用提供技术支持 （ 交互部分)
- 为多个浏览器兼容性提供支持
- 为提高用户浏览速度 （ 浏览器性能）提供支持
- 为跨平台或者其他基于webkit或其他渲染引擎的应用提供支持
- 为展示数据提供支持 （ 数据接口)

5 平时如何管理你的项目

- 先期团队必须确定好全局样式 （ `globe.css` ）， 编码模式(`utf-8`) 等；
- 编写习惯必须一致 （例如都是采用继承式的写法， 单样式都写成一行)；
- 标注样式编写人 ，各模块都及时标注 （标注关键样式调用的地方)；
- 页面进行标注 （例如 页面 模块 开始和结束)；

- CSS 跟 HTML 分文件夹并行存放，命名都得统一(例如 style.css)；
- JS 分文件夹存放 命名以该 JS 功能为准的英文翻译。
- 图片采用整合的 images.png png8 格式文件使用 - 尽量整合在一起使用方便将来的管理

6 组件封装

> 目的：为了重用，提高开发效率和代码质量 注意：低耦合， 单一职责， 可复用性， 可维护性 常用操作

- 分析布局
- 初步开发
- 化繁为简
- 组件抽象

十一、一些常见问题

- 自我介绍
- 面试完你还有什么问题要问的吗
- 你有什么爱好?
- 你最大的优点和缺点是什么?
- 你为什么会选择这个行业， 职位?
- 你觉得你适合从事这个岗位吗?
- 你有什么职业规划?
- 你对工资有什么要求?
- 如何看待前端开发?
- 未来三到五年的规划是怎样的?
- 你的项目中技术难点是什么？遇到了什么问题？你是怎么解决的?
- 你们部门的开发流程是怎样的
- 你认为哪个项目做得最好?
- 说下工作中你做过的一些性能优化处理
- 最近在看哪些前端方面的书?
- 平时是如何学习前端开发的?
- 你最有成就感的一件事
- 你为什么要离开前一家公司?
- 你对加班的看法
- 你希望通过这份工作获得什么?

第二部分： 进阶篇

一、JS

1 谈谈变量提升

> 当执行 JS 代码时，会生成执行环境， 只要代码不是写在函数中的，就是在全局执行环境中， 函数中的代码会产生函数执行环境， 只此两种执行环境。

```js
b() // call b
console.log(a) // undefined

var a = 'Hello world'

function b() {
    console.log( 'call b')
}
```

> 想必以上的输出大家肯定都已经明白了， 这是因为函数和变量提升的原因 。通常提升的解释是说将声明的代码移动到了顶部， 这其实没有什么错误，便于大家理解 。但是更准确的解释应该是：在生成执行环境时，会有两个阶段 。第一个阶段是创建的阶段， JS 解释器会找出需要提升的变量和函数， 并且给他们提前在内存中开辟好空间， 函数的话会将整个函数存入内存中， 变量只声明并且赋值为 undefined ，所以在第二个阶段，也就是代码执行阶段， 我们可以直接提前使用

- 在提升的过程中，相同的函数会覆盖上一个函数， 并且函数优先于变量提升

```js
b() // call b second

function b() {
    console.log( 'call b fist')
}
function b() {
```

```
    console.log( 'call b second')
}
var b = 'Hello world'
```

> var 会产生很多错误，所以在 ES6中引入了 let 。 let 不能在声明前使用，但是这并不是常说的 let 不会提升， let 提升了，在第一阶段内存也已经为他开辟好了空间，但是因为这个声明的特性导致了并不能在声明前使用

2 bind 、call 、apply 区别

- call 和 apply 都是为了解决改变 this 的指向 。作用都是相同的， 只是传参的方式不同。
- 除了第一个参数外， call 可以接收一个参数列表， apply 只接受一个参数数组

```
let a = {
    value: 1
}
function getValue(name, age) {
    console.log(name)
    console.log(age)
    console.log(this.value)
}
getValue.call(a, 'yck', '24')
getValue.apply(a, [ 'yck', '24'])
```

> bind 和其他两个方法作用也是一致的， 只是该方法会返回一个函数 。并且我们可以通过 bind 实现柯里化

3 如何实现一个bind 函数

对于实现以下几个函数， 可以从几个方面思考

- 不传入第一个参数，那么默认为 window
- 改变了 this 指向，让新的对象可以执行该函数 。那么思路是否可以变成给新的对象添加一个函数，然后在执行完以后删除?

```js
Function.prototype.myBind = function (context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  var _this = this
  var args = [...arguments].slice(1)
  // 返回一个函数
  return function F() {
    // 因为返回了一个函数，我们可以 new F()，所以需要判断
    if (this instanceof F) {
      return new _this(...args, ...arguments)
    }
    return _this.apply(context, args.concat(...arguments))
  }
}
```

4 如何实现一个 call 函数

```js
Function.prototype.myCall = function (context) {
  var context = context || window
  // 给 context 添加一个属性
  // getValue.call(a, 'yck', '24') => a.fn = getValue
  context.fn = this
  // 将 context 后面的参数取出来
  var args = [...arguments].slice(1)
  // getValue.call(a, 'yck', '24') => a.fn('yck', '24')
  var result = context.fn(...args)
  // 删除 fn
  delete context.fn
  return result
}
```

5 如何实现一个 apply 函数

```js
Function.prototype.myApply = function (context) {
  var context = context || window
  context.fn = this

  var result
  // 需要判断是否存储第二个参数
  // 如果存在，就将第二个参数展开
  if (arguments[1]) {
```

```
    result = context.fn(...arguments [1])
  } else {
    result = context.fn()
  }

  delete context.fn
  return result
}
```

6 简单说下原型链?

- 每个函数都有 `prototype` 属性， 除了 `Function.prototype.bind()` ，该属性指向原型。
- 每个对象都有 `__proto__` 属性，指向了创建该对象的构造函数的原型 。其实这个属性指向了 `[[prototype]]` ，但是 `[[prototype]]` 是内部属性， 我们并不能访问到，所以使用 `_proto_` 来访问。
- 对象可以通过 `__proto__` 来寻找不属于该对象的属性， `__proto__` 将对象连接起来组成了原型链。

7 怎么判断对象类型

- 可以通过 `Object.prototype.toString.call(xx)` 。这样我们就可以获得类似 `[object Type]` 的字符串。
- `instanceof` 可以正确的判断对象的类型， 因为内部机制是通过判断对象的原型链中是不是能找到类型的 `prototype`

8 箭头函数的特点

```js
function a() {
    return () => {
        return () => {
            console.log(this)
        }
    }
}
console.log(a()()())
```

> 箭头函数其实是没有 this 的， 这个函数中的 this 只取决于他外面的第一个不是箭头函数的函数的 this 。在这个例子中， 因为调用 a 符合前面代码中的第一个情况，所以 this 是 window 。并且 this 一旦绑定了上下文，就不会被任何代码改变

9 This

```js
function foo() {
    console.log(this.a)
}
var a = 1
foo()

var obj = {
    a: 2,
    foo: foo
}
obj.foo()

// 以上两者情况 `this` 只依赖于调用函数前的对象，优先级是第二个情况大于第一个情况

// 以下情况是优先级最高的， `this` 只会绑定在 `c` 上，不会被任何方式修改 `this` 指向
var c = new foo()
c.a = 3
console.log(c.a)

// 还有种就是利用 call， apply， bind 改变 this， 这个优先级仅次于 new
```

10 async 、await 优缺点

> async 和 await 相比直接使用 Promise 来说，优势在于处理 then 的调用链， 能够更清晰准确的写出代码 。缺点在于滥用 await 可能会导致性能问题， 因为 await 会阻塞代码，也许之后的异步代码并不依赖于前者，但仍然需要等待前者完成， 导致代码失去了并发性

下面来看一个使用 await 的代码。

```js
var a = 0
var b = async () => {
  a = a + await 10
  console.log( '2', a) // -> '2' 10
  a = (await 10) + a
  console.log( '3', a) // -> '3' 20
}
b()
a++
console.log( '1', a) // -> '1' 1
```

- 首先函数 b 先执行，在执行到 await 10 之前变量 a 还是 0 ， 因为在 await 内部实现了 generators ， generators 会保留堆栈中东西，所以这时候 a = 0 被保存了下来
- 因为 await 是异步操作， 遇到 await 就会立即返回一个 pending 状态的 Promise 对象， 暂时返回执行代码的控制权，使得函数外的代码得以继续执行，所以会先执行 console.log('1', a)
- 这时候同步代码执行完毕， 开始执行异步代码，将保存下来的值拿出来使用， 这时候 a = 10
- 然后后面就是常规执行代码了

11 generator 原理

> Generator 是 ES6 中新增的语法，和 Promise 一样，都可以用来异步编程

```js
// 使用 * 表示这是一个 Generator 函数
// 内部可以通过 yield 暂停代码
// 通过调用 next 恢复执行
function* test() {
  let a = 1 + 2;
  yield 2;
  yield 3;
}
let b = test();
console.log(b.next()); // >  { value: 2, done: false }
console.log(b.next()); // >  { value: 3, done: false }
console.log(b.next()); // >  { value: undefined, done: true }
```

> 从以上代码可以发现，加上 * 的函数执行后拥有了 next 函数，也就是说函数执行后返回了一个对象 。每次调用 next 函数可以继续执行被暂停的代码 。以下是 Generator 函数的简单实现

```js
// cb 也就是编译过的 test 函数
function generator(cb) {
  return (function() {
    var object = {
      next: 0,
      stop: function() {}
    };

    return {
      next: function() {
        var ret = cb(object);
        if (ret === undefined) return { value: undefined, done: true };
        return {
          value: ret,
          done: false
        };
      }
    };
  })();
}
// 如果你使用 babel 编译后可以发现 test 函数变成了这样
function test() {
  var a;
  return generator(function(_context) {
    while (1) {
      switch ((_context.prev = _context.next)) {
        // 可以发现通过 yield 将代码分割成几块
        // 每次执行 next 函数就执行一块代码
        // 并且表明下次需要执行哪块代码
        case 0:
          a = 1 + 2;
          _context.next = 4;
          return 2;
        case 4:
          _context.next = 6;
          return 3;
        // 执行完毕
        case 6:
        case "end":
          return _context.stop();
```

```
      }
    }
  });
}
```

12 Promise

- Promise 是 ES6 新增的语法，解决了回调地狱的问题。
- 可以把 Promise 看成一个状态机。初始是 pending 状态， 可以通过函数 resolve 和 reject ，将状态转变为 resolved 或者 rejected 状态，状态一旦改变就不能再次变化。
- then 函数会返回一个 Promise 实例， 并且该返回值是一个新的实例而不是之前的实例。因为 Promise 规范规定除了 pending 状态， 其他状态是不可以改变的， 如果返回的是一个相同实例的话， 多个 then 调用就失去意义了。 对于 then 来说，本质上可以把它看成是 flatMap

13 如何实现一个 Promise

```js
// 三种状态
const PENDING = "pending";
const RESOLVED = "resolved";
const REJECTED = "rejected";
// promise 接收一个函数参数，该函数会立即执行
function MyPromise(fn) {
  let _this = this;
  _this.currentState = PENDING;
  _this.value = undefined;
  // 用于保存 then 中的回调， 只有当 promise
  // 状态为 pending 时才会缓存，并且每个实例至多缓存一个
  _this.resolvedCallbacks = [];
  _this.rejectedCallbacks = [];

  _this.resolve = function (value) {
    if (value instanceof MyPromise) {
      // 如果 value 是个 Promise，递归执行
      return value.then(_this.resolve, _this.reject)
    }
    setTimeout(() => { // 异步执行，保证执行顺序
      if (_this.currentState === PENDING) {
        _this.currentState = RESOLVED;
        _this.value = value;
        _this.resolvedCallbacks.forEach(cb => cb());
```

```
      }
    })
  };

  _this.reject = function (reason) {
    setTimeout(() => { // 异步执行，保证执行顺序
      if (_this.currentState === PENDING) {
        _this.currentState = REJECTED;
        _this.value = reason;
        _this.rejectedCallbacks.forEach(cb => cb());
      }
    })
  }
  // 用于解决以下问题
  // new Promise(() => throw Error('error))
  try {
    fn(_this.resolve, _this.reject);
  } catch (e) {
    _this.reject(e);
  }
}

MyPromise.prototype.then = function (onResolved, onRejected) {
  var self = this;
  // 规范 2.2.7，then 必须返回一个新的 promise
  var promise2;
  // 规范 2.2.onResolved 和 onRejected 都为可选参数
  // 如果类型不是函数需要忽略，同时也实现了透传
  // Promise.resolve(4).then().then((value) => console.log(value))
  onResolved = typeof onResolved === 'function' ? onResolved : v => v;
  onRejected = typeof onRejected === 'function' ? onRejected : r => throw r

  if (self.currentState === RESOLVED) {
    return (promise2 = new MyPromise(function (resolve, reject) {
      // 规范 2.2.4，保证 onFulfilled，onRjected 异步执行
      // 所以用了 setTimeout 包裹下
      setTimeout(function () {
        try {
          var x = onResolved(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (reason) {
          reject(reason);
        }
      });
    }));
  }
```

```
  if (self.currentState === REJECTED) {
    return (promise2 = new MyPromise(function (resolve, reject) {
      setTimeout(function () {
        // 异步执行onRejected
        try {
          var x = onRejected(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (reason) {
          reject(reason);
        }
      });
    }));
  }

  if (self.currentState === PENDING) {
    return (promise2 = new MyPromise(function (resolve, reject) {
      self.resolvedCallbacks.push(function () {
        // 考虑到可能会有报错，所以使用 try/catch 包裹
        try {
          var x = onResolved(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (r) {
          reject(r);
        }
      });

      self.rejectedCallbacks.push(function () {
        try {
          var x = onRejected(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (r) {
          reject(r);
        }
      });
    }));
  }
};
// 规范 2.3
function resolutionProcedure(promise2, x, resolve, reject) {
  // 规范 2.3.1，x 不能和 promise2 相同，避免循环引用
  if (promise2 === x) {
    return reject(new TypeError("Error"));
  }
  // 规范 2.3.2
  // 如果 x 为 Promise，状态为 pending 需要继续等待否则执行
  if (x instanceof MyPromise) {
    if (x.currentState === PENDING) {
```

```
      x.then(function (value) {
        // 再次调用该函数是为了确认 x resolve 的
        // 参数是什么类型，如果是基本类型就再次 resolve
        // 把值传给下个 then
        resolutionProcedure(promise2, value, resolve, reject);
      }, reject);
    } else {
      x.then(resolve, reject);
    }
    return;
  }
  // 规范 2.3.3.3
  // reject 或者 resolve 其中一个执行过得话， 忽略其他的
  let called = false;
  // 规范 2.3.3，判断 x 是否为对象或者函数
  if (x !== null && (typeof x === "object" || typeof x === "function")) {
    // 规范 2.3.3.2，如果不能取出 then，就 reject
    try {
      // 规范 2.3.3.1
      let then = x.then;
      // 如果 then 是函数，调用 x.then
      if (typeof then === "function") {
        // 规范 2.3.3.3
        then.call(
          x,
          y => {
            if (called) return;
            called = true;
            // 规范 2.3.3.3.1
            resolutionProcedure(promise2, y, resolve, reject);
          },
          e => {
            if (called) return;
            called = true;
            reject(e);
          }
        );
      } else {
        // 规范 2.3.3.4
        resolve(x);
      }
    } catch (e) {
      if (called) return;
      called = true;
      reject(e);
    }
  } else {
```

```
    // 规范 2.3.4, x 为基本类型
    resolve(x);
  }
}
```

14 == 和 ===区别，什么情况用 ==

> 这里来解析一道题目 `[] == ![] // -> true` ，下面是这个表达式为何为 `true` 的步骤

```js
// [] 转成 true， 然后取反变成 false
[] == false
// 根据第 8 条得出
[] == ToNumber(false)
[] == 0
// 根据第 10 条得出
ToPrimitive( []) == 0
// [].toString() -> ''
'' == 0
// 根据第 6 条得出
0 == 0 // -> true
```

> `===` 用于判断两者类型和值是否相同 。 在开发中，对于后端返回的 `code` ，可以通过 `==` 去判断

15 基本数据类型和引用类型在存储上的差别

> 前者存储在栈上， 后者存储在堆上

16 浏览器 Eventloop 和 Node 中的有什么区别

> 众所周知 JS 是门非阻塞单线程语言， 因为在最初 JS 就是为了和浏览器交互而诞生的 。如果 JS 是门多线程的语言话， 我们在多个线程中处理 DOM 就可能