- 如果有右节点，就找到该右节点的最左节点 。对于节点 1 来说，他有右节点 3 ，那么节点 3 的最左节点就是 6
- 如果没有右节点，就向上遍历直到找到一个节点是父节点的左节点 。对于节点 5 来说，没有右节点，就向上寻找到节点 2 ，该节点是父节点 1 的左节点，所以节点 1 是后继节点 以下是算法实现

```js
function successor(node) {
  if (!node) return
  // 结论 1
  if (node.right) {
    return getLeft(node.right)
  } else {
    // 结论 2
    let parent = node.parent
    // 判断 parent 为空
    while(parent && parent.left === node) {
      node = parent
      parent = node.parent
    }
    return parent
  }
}
function getLeft(node) {
  if (!node) return
  node = node.left
  while(node) node = node.left
  return node
}
```

树的深度

> 树的最大深度：该题目来自 Leetcode，题目需要求出一颗二叉树的最大深度

以下是算法实现

```js
var maxDepth = function(root) {
    if (!root) return 0
    return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1
};
```

> 对于该递归函数可以这样理解：一旦没有找到节点就会返回 0，每弹出一次递归函数就会加一， 树有三层就会得到3

第五部分： 高频考点

1 typeof类型判断

> typeof 是否能正确判断类型？ instanceof 能正确判断对象的原理是什么

- typeof 对于原始类型来说， 除了 null 都可以显示正确的类型

```js
typeof 1 // 'number'
typeof '1' // 'string'
typeof undefined // 'undefined'
typeof true // 'boolean'
typeof Symbol() // 'symbol'
```

> typeof 对于对象来说， 除了函数都会显示 object ，所以说 typeof 并不能准确判断变量到底是什么类型

```js
typeof [] // 'object'
typeof {} // 'object'
typeof console.log // 'function'
```

> 如果我们想判断一个对象的正确类型， 这时候可以考虑使用 instanceof ，因为内部机制是通过原型链来判断的

```js
const Person = function() {}
const p1 = new Person()
p1 instanceof Person // true

var str = 'hello world'
str instanceof String // false
```

```
var str1 = new String( 'hello world')
str1 instanceof String // true
```

> 对于原始类型来说，你想直接通过 `instanceof` 来判断类型是不行的

2 类型转换

> 首先我们要知道，在 `JS` 中类型转换只有三种情况，分别是：

- 转换为布尔值
- 转换为数字
- 转换为字符串

转Boolean

> 在条件判断时， 除了 `undefined` ， `null` ， `false` ， `NaN` ， `''` ， `0` ， `-0` ， 其他所有值都转为 `true` ， 包括所有对象

对象转原始类型

> 对象在转换类型的时候，会调用内置的 `[[ToPrimitive]]` 函数，对于该函数来说， 算法逻辑一般来说如下

- 如果已经是原始类型了，那就不需要转换了
- 调用 `x.valueOf()` ， 如果转换为基础类型，就返回转换的值
- 调用 `x.toString()` ， 如果转换为基础类型，就返回转换的值
- 如果都没有返回原始类型，就会报错

> 当然你也可以重写 `Symbol.toPrimitive` ，该方法在转原始类型时调用优先级最高。

```
let a = {
 valueOf() {
```

```
    return 0
  },
  toString() {
    return '1'
  },
  [Symbol.toPrimitive]() {
    return 2
  }
}
1 + a // => 3
```

四则运算符

> 它有以下几个特点：

- 运算中其中一方为字符串，那么就会把另一方也转换为字符串
- 如果一方不是字符串或者数字，那么会将它转换为数字或者字符串

```js
1 + '1' // '11'
true + true // 2
4 + [1,2,3] // "41,2,3"
```

- 对于第一行代码来说，触发特点一， 所以将数字 `1` 转换为字符串，得到结果 `'11'`
- 对于第二行代码来说，触发特点二，所以将 `true` 转为数字 `1`
- 对于第三行代码来说，触发特点二，所以将数组通过 `toString` 转为字符串 `1,2,3` ，得到结果 `41,2,3`

> 另外对于加法还需要注意这个表达式 `'a' + + 'b'`

```
'a' + + 'b' // -> "aNaN"
```

- 因为 `+ 'b'` 等于 `NaN` ，所以结果为 `"aNaN"` ，你可能也会在一些代码中看到过 `+ '1'` 的形式来快速获取 `number` 类型。
- 那么对于除了加法的运算符来说， 只要其中一方是数字，那么另一方就会被转为数字

```js
4 * '3' // 12
4 * [] // 0
4 * [1, 2] // NaN
```

比较运算符

- 如果是对象，就通过 `toPrimitive` 转换对象
- 如果是字符串，就通过 `unicode` 字符索引来比较

```js
let a = {
  valueOf() {
    return 0
  },
  toString() {
    return '1'
  }
}
a > -1 // true
```

> 在以上代码中，因为 `a` 是对象，所以会通过 `valueOf` 转换为原始类型再比较值。

3 This

> 我们先来看几个函数调用的场景

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

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

const c = new foo()
```

- 对于直接调用 `foo` 来说，不管 `foo` 函数被放在了什么地方，`this` 一定是 `window`

- 对于 obj.foo() 来说， 我们只需要记住，谁调用了函数，谁就是 this ，所以在这个场景下 foo 函数中的 this 就是 obj 对象
- 对于 new 的方式来说， this 被永远绑定在了 c 上面，不会被任何方式改变 this

> 说完了以上几种情况， 其实很多代码中的 this 应该就没什么问题了，下面让我们看看箭头函数中的 this

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

- 首先箭头函数其实是没有 this 的， 箭头函数中的 this 只取决包裹箭头函数的第一个普通函数的 this 。在这个例子中， 因为包裹箭头函数的第一个普通函数是 a ， 所以此时的 this 是 window 。另外对箭头函数使用 bind 这类函数是无效的 。
- 最后种情况也就是 bind 这些改变上下文的 API 了， 对于这些函数来说， this 取决于第一个参数， 如果第一个参数为空，那么就是 window 。
- 那么说到 bind ，不知道大家是否考虑过， 如果对一个函数进行多次 bind ，那么上下文会是什么呢?

```js
let a = {}
let fn = function () { console.log(this) }
fn.bind().bind(a)() // => ?
```

> 如果你认为输出结果是 a ，那么你就错了， 其实我们可以把上述代码转换成另一种形式

```js
// fn.bind().bind(a) 等于
let fn2 = function fn1() {
  return function() {
    return fn.apply()
  }.apply(a)
}
fn2()
```

> 可以从上述代码中发现，不管我们给函数 bind 几次， fn 中的 this 永远由第一次 bind 决定，所以结果永远是 window

```js
let a = { name : 'poetries' }
function foo() {
  console. log( this. name)
}
foo .bind(a)( ) // => 'poetries'
```

> 以上就是 this 的规则了，但是可能会发生多个规则同时出现的情况， 这时候不同的规则之间会根据优先级最高的来决定 this 最终指向哪里。

> 首先， new 的方式优先级最高，接下来是 bind 这些函数，然后是 obj.foo () 这种调用方式， 最后是 foo 这种调用方式， 同时， 箭头函数的 this 一旦被绑定，就不会再被任何方式所改变。

image.png

4 == 和 === 有什么区别

> 对于 == 来说， 如果对比双方的类型不一样的话，就会进行类型转换

假如我们需要对比 x 和 y 是否相同，就会进行如下判断流程

1. 首先会判断两者类型是否相同 。相同的话就是比大小了
2. 类型不相同的话，那么就会进行类型转换
3. 会先判断是否在对比 null 和 undefined ， 是的话就会返回 true
4. 判断两者类型是否为 string 和 number ， 是的话就会将字符串转换为 number

```
1 == '1'
     ↓
1 == 1
```

5. 判断其中一方是否为 boolean ， 是的话就会把 boolean 转为 number 再进行判断

```
'1' == true
        ↓
'1' == 1
        ↓
 1  == 1
```

6. 判断其中一方是否为 object 且另一方为 string 、 number 或者 symbol ， 是的话就会把 object 转为原始类型再进行判断

```
'1' == { name: 'yck' }
        ↓
'1' == '[object Object]'
```

> 对于 === 来说就简单多了，就是判断两者类型和值是否相同

5 闭包

> 闭包的定义其实很简单： 函数 A 内部有一个函数 B ， 函数 B 可以访问到函数 A 中的变量，那么函数 B 就是闭包

```js
function A() {
  let a = 1
  window.B = function () {
    console.log(a)
  }
}
A()
B() // 1
```

闭包存在的意义就是让我们可以间接访问函数内部的变量

> 经典面试题，循环中使用闭包解决 var 定义函数的问题

```js
for (var i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)
  }, i * 1000)
}
```

> 首先因为 setTimeout 是个异步函数，所以会先把循环全部执行完毕， 这时候 i 就是 6 了，所以会输出一堆 6

解决办法有三种

1. 第一种是使用闭包的方式

```js
for (var i = 1; i <= 5; i++) {
  ;(function(j) {
    setTimeout(function timer() {
      console.log(j)
    }, j * 1000)
  })(i)
}
```

> 在上述代码中， 我们首先使用了立即执行函数将 i 传入函数内部， 这个时候值就被固定在了参数 j 上面不会改变， 当下次执行 timer 这个闭包的时候，就可以使用外部函数的变量 j ，从而达到目的

2. 第二种就是使用 setTimeout 的第三个参数， 这个参数会被当成 timer 函数的参数传入

```js
for (var i = 1; i <= 5; i++) {
  setTimeout(
    function timer(j) {
      console.log(j)
    },
    i * 1000,
    i
  )
}
```

3. 第三种就是使用 `let` 定义 `i` 了来解决问题了， 这个也是最为推荐的方式

```js
for (let i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)
  }, i * 1000)
}
```

6 深浅拷贝

浅拷贝

> 首先可以通过 `Object.assign` 来解决这个问题，很多人认为这个函数是用来深拷贝的 。其实并不是， `Object.assign` 只会拷贝所有的属性值到新的对象中， 如果属性值是对象的话，拷贝的是地址，所以并不是深拷贝

```js
let a = {
  age: 1
}
let b = Object.assign({}, a)
a.age = 2
console.log(b.age) // 1
```

> 另外我们还可以通过展开运算符 `...` 来实现浅拷贝

```js
let a = {
  age: 1
}
let b = { ...a }
a.age = 2
console.log(b.age) // 1
```

> 通常浅拷贝就能解决大部分问题了，但是当我们遇到如下情况就可能需要使用到深拷贝了

```js
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}
let b = { ...a }
a.jobs.first = 'native'
console.log(b.jobs.first) // native
```

> 浅拷贝只解决了第一层的问题， 如果接下去的值中还有对象的话，那么就又回到最开始的话题了，两者享有相同的地址 。要解决这个问题， 我们就得使用深拷贝了。

深拷贝

> 这个问题通常可以通过 `JSON.parse(JSON.stringify(object))` 来解决。

```js
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}
let b = JSON.parse(JSON.stringify(a))
a.jobs.first = 'native'
console.log(b.jobs.first) // FE
```

但是该方法也是有局限性的：

- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能序列化函数
- 不能解决循环引用的对象

```js
let obj = {
  a: 1,
  b: {
    c: 2,
```

```
    d: 3,
  },
}
obj.c = obj.b
obj.e = obj.a
obj.b.c = obj.c
obj.b.d = obj.b
obj.b.e = obj.b.c
let newObj = JSON.parse(JSON.stringify(obj))
console.log(newObj)
```

> 更多详情 https://www.jianshu.com/p/2d8a26b3958f

7 原型

> 原型链就是多个对象通过 `__proto__` 的方式连接了起来 。为什么 `obj` 可以访问到 `valueOf` 函数，就是因为 `obj` 通过原型链找到了 `valueOf` 函数

- `Object` 是所有对象的爸爸，所有对象都可以通过 `__proto__` 找到它
- `Function` 是所有函数的爸爸，所有函数都可以通过 `__proto__` 找到它
- 函数的 `prototype` 是一个对象
- 对象的 `__proto__` 属性指向原型， `__proto__` 将对象和原型连接起来组成了原型链

8 var 、let 及 const 区别

> 涉及面试题：什么是提升？什么是暂时性死区？var、let 及 const 区别？

- 函数提升优先于变量提升， 函数提升会把整个函数挪到作用域顶部， 变量提升只会把声明挪到作用域顶部
- `var` 存在提升， 我们能在声明之前使用 。 `let` 、 `const` 因为暂时性死区的原因，不能在声明前使用
- `var` 在全局作用域下声明变量会导致变量挂载在 `window` 上， 其他两者不会
- `let` 和 `const` 作用基本一致，但是后者声明的变量不能再次赋值

9 原型继承和 Class 继承

> 涉及面试题：原型如何实现继承？ Class 如何实现继承？ Class 本质是什么？

首先先来讲下 class ， 其实在 JS 中并不存在类， class 只是语法糖，本质还是函数

```js
class Person {}
Person instanceof Function // true
```

组合继承

> 组合继承是最常用的继承方式

```js
function Parent(value) {
  this.val = value
}
Parent.prototype.getValue = function() {
  console.log(this.val)
}
function Child(value) {
  Parent.call(this, value)
}
Child.prototype = new Parent()

const child = new Child(1)

child.getValue() // 1
child instanceof Parent // true
```

- 以上继承的方式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。
- 这种继承方式优点在于构造函数可以传参，不会与父类引用属性共享， 可以复用父类的函数，但是也存在一个缺点就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

> 这种继承方式对组合继承进行了优化， 组合继承缺点在于继承父类函数时调用了构造函数， 我们只需要优化掉这点就行了

```js
function Parent(value) {
  this.val = value
}
Parent.prototype.getValue = function() {
  console.log(this.val)
}

function Child(value) {
  Parent.call(this, value)
}
Child.prototype = Object.create(Parent.prototype, {
  constructor: {
    value: Child,
    enumerable: false,
    writable: true,
    configurable: true
  }
})

const child = new Child(1)

child.getValue() // 1
child instanceof Parent // true
```

> 以上继承实现的核心就是将父类的原型赋值给了子类， 并且将构造函数设置为子类， 这样既解决了无用的父类属性问题， 还能正确的找到子类的构造函数。

Class 继承

> 以上两种继承方式都是通过原型去解决的，在 ES6 中， 我们可以使用 class 去实现继承， 并且实现起来很简单

```js
class Parent {
  constructor(value) {
    this.val = value
  }
  getValue() {
    console.log(this.val)
  }
}
class Child extends Parent {
  constructor(value) {
    super(value)
    this.val = value
  }
}
let child = new Child(1)
child.getValue() // 1
child instanceof Parent // true
```

> `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)` 。

10 模块化

> 涉及面试题：为什么要使用模块化？都有哪几种方式可以实现模块化，各有什么特点?

使用一个技术肯定是有原因的，那么使用模块化可以给我们带来以下好处

- 解决命名冲突
- 提供复用性
- 提高代码可维护性

立即执行函数

> 在早期，使用立即执行函数实现模块化是常见的手段， 通过函数作用域解决了命名冲突 、污染全局作用域的问题

```
(function(globalVariable){
    globalVariable.test = function() {}
    // ... 声明各种变量 、函数都不会污染全局作用域
})(globalVariable)
```

AMD 和 CMD

> 鉴于目前这两种实现方式已经很少见到，所以不再对具体特性细聊， 只需要了解这两者是如何使用的。

```js
// AMD
define( [ './a', './b'], function(a, b) {
    // 加载模块完毕可以使用
    a.do()
    b.do()
})
// CMD
define(function(require, exports, module) {
    // 加载模块
    // 可以把 require 写在函数体的任意地方实现延迟加载
    var a = require( './a')
    a.doSomething()
})
```

CommonJS

> CommonJS 最早是 Node 在使用， 目前也仍然广泛使用， 比如在 Webpack 中你就能见到它， 当然目前在 Node 中的模块管理已经和 CommonJS 有一些区别了

```js
// a.js
module.exports = {
    a: 1
}
// or
exports.a = 1

// b.js
```

```
var module = require( './a.js')
module.a // -> log 1
```

```
ar module = require( './a.js')
module.a
// 这里其实就是包装了一层立即执行函数， 这样就不会污染全局变量了，
// 重要的是 module 这里，module 是 Node 独有的一个变量
module.exports = {
    a: 1
}
// module 基本实现
var module = {
  id: 'xxxx', // 我总得知道怎么去找到他吧
  exports: {} // exports 就是个空对象
}
// 这个是为什么 exports 和 module.exports 用法相似的原因
var exports = module.exports
var load = function (module) {
    // 导出的东西
    var a = 1
    module.exports = a
    return module.exports
};
// 然后当我 require 的时候去找到独特的
// id，然后将要使用的东西用立即执行函数包装下， over
```

> 另外虽然 `exports` 和 `module.exports` 用法相似，但是不能对 `exports` 直接赋值 。因为 `var exports = module.exports` 这句代码表明了 `exports` 和 `module.exports` 享有相同地址， 通过改变对象的属性值会对两者都起效，但是如果直接对 `exports` 赋值就会导致两者不再指向同一个内存地址，修改并不会对 `module.exports` 起效

ES Module

> `ES Module` 是原生实现的模块化方案，与 `CommonJS` 有以下几个区别

1. `CommonJS` 支持动态导入，也就是 `require(${path}/xx.js)` ，后者目前不支持，但是已有提案
2. `CommonJS` 是同步导入， 因为用于服务端，文件都在本地， 同步导入即使卡住主线程影响也不大 。而后者是异步导入， 因为用于浏览器， 需要下载文件， 如果也采用同步导入会对

渲染有很大影响

3. CommonJS 在导出时都是值拷贝，就算导出的值变了， 导入的值也不会改变，所以如果想更新值，必须重新导入一次 。但是 ES Module 采用实时绑定的方式， 导入导出的值都指向同一个内存地址，所以导入值会跟随导出值变化
4. ES Module 会编译成 require/exports 来执行的

```js
// 引入模块 API
import XXX from './a.js'
import { XXX } from './a.js'
// 导出模块 API
export function a() {}
export default function() {}
```

11 实现一个简洁版的promise

```js
// 三个常量用于表示状态
const PENDING = 'pending'
const RESOLVED = 'resolved'
const REJECTED = 'rejected'

function MyPromise(fn) {
    const that = this
    this.state = PENDING

    // value 变量用于保存 resolve 或者 reject 中传入的值
    this.value = null

    // 用于保存 then 中的回调， 因为当执行完 Promise 时状态可能还是等待中， 这时候应该把
    that.resolvedCallbacks = []
    that.rejectedCallbacks = []


    function resolve(value) {
         // 首先两个函数都得判断当前状态是否为等待中
        if(that.state === PENDING) {
            that.state = RESOLVED
            that.value = value

            // 遍历回调数组并执行
            that.resolvedCallbacks.map(cb=>cb(that.value))
        }
    }
    function reject(value) {
```

```
            if(that.state === PENDING) {
                that.state = REJECTED
                that.value = value
                that.rejectedCallbacks.map(cb=>cb(that.value))
            }
        }

        // 完成以上两个函数以后，我们就该实现如何执行 Promise 中传入的函数了
        try {
            fn(resolve,reject)
        }cach(e){
            reject(e)
        }
    }

    // 最后我们来实现较为复杂的 then 函数
    MyPromise.prototype.then = function(onFulfilled,onRejected){
      const that = this

      // 判断两个参数是否为函数类型， 因为这两个参数是可选参数
      onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : v=>v
      onRejected = typeof onRejected === 'function' ? onRejected : e=>throw e

      // 当状态不是等待态时，就去执行相对应的函数 。如果状态是等待态的话，就往回调函数中 push
      if(this.state === PENDING) {
          this.resolvedCallbacks.push(onFulfilled)
          this.rejectedCallbacks.push(onRejected)
      }
      if(this.state === RESOLVED) {
          onFulfilled(that.value)
      }
      if(this.state === REJECTED) {
          onRejected(that.value)
      }
    }
```

12 Event Loop

12.1 进程与线程

> 涉及面试题：进程与线程区别？ JS 单线程带来的好处？