

小册食用指南

这本小册涉及的内容十分广，在这里我提供了一份小册的**食用指南**，帮助你更好地阅读小册。

无论你是刚刚开始学习小册还是想复习小册内容，这份食用指南都能很好地帮助到你。

购买前警告⚠

- 此小册不适合完全沒有前端基础的人阅读。需要各位掌握基本的 HTML、JS，担心小册质量或内容是否适合自己的，请先浏览试读章节再做购买决定。
- 小册已完结，无法继续购买。
- 此小册不会让你的技术突飞猛进，直接从三线项目跳到一线大厂，想进大公司必定是需要个人实力才行，当然小册的内容应付一般公司完全没问题。

发售福利

小册售价为 **49.9 元**

欢迎大家在交流群中多交流学习及面试相关的内容，目前群内交流气氛浓厚，通过交流可以了解到很多面试相关的内容。



内容

我一直认为，面试题目对面试来说是没什么帮助的，即使有，也只是拾遗不拾遗。

因为每道面试题背后都会涉及到几个知识点，如果我们能够扎实地学习这些知识点的话，那么无论题目怎么变，只要涉及的知识点不变，我们就能以不变应万变。

所以，我收集到了大量的面试题，把它们背后所涉及的知识点一一提炼出来，并整理出了常考知识点。当然小册所涉及的知识点非常考的知识点，还包含了一部分我认为重要的知识点（虽然考的不多）、面试技巧和学习资料。

总的来说，整本小册涉及十四个模块，每一模块中又包含许多的知识点，每一模块都自成体系但是又会与其他模块中的内容有交叉。比如浏览器、Webpack、网络协议这几个模块中涉及的部分内容和性能优化模块是相互关联的。

如果你是刚接触开始阅读小册的内容，可以根据自己的薄弱点对症下药来学习相应的模块，但是学习单个模块时不能推荐跳着阅读，因为很可能后面的内容与之前的有所联系，如果你没有理解之前的知识点的话，又会影响到后续的学习，所以你需要跟着前面的内容一起学习。

在学习的过程中，我又给大部分的知识点提供了 1~3 道思考题，你可以通过学习知识点的方式尝试自己攻克面试题。当你学习完整个模块后，我又提供了几道思考题，帮助你检验自身的学习成果，查漏补缺。

小册的内容会持续更新（更新日志都在首页），毕竟面试涉及的知识点很广，内容可能会存在疏忽或者不清楚的地方，并且前端技术更新很快，我会尽可能的让小册内容符合当下最新的技术，可预见的是 Vue 3.0 的内容势必会更新，如果你是在非官方渠道阅读到这本小册的话，为了你学到的内容符合当下，你可以选择 [支持正版](#)、[购买小册](#)。

最后，学习知识一定要配合实践，没有实践的知识是没有灵魂的。另外，碍于篇幅，我不可能深挖每个知识点，所以推荐大家去尝试挖掘我没有涉及的内容。

思考题

在大部分的模块内容结束后，我都提供了几道思考题。虽然每道思考题通过题目我们可能只能理解到背后所涉及的 1~2 个知识点，但是其实很多知识点是有串联关系的。

在面试过程中，如果经常被面试官问出一问一答的情况的话，其实是不合理理想的。虽然一道面试题看起来只涉及了一个知识点，但如果将面试中的知识点是串联起来的话，就可以引申出其他的知识点，这样能给到面试官一个好的印象。

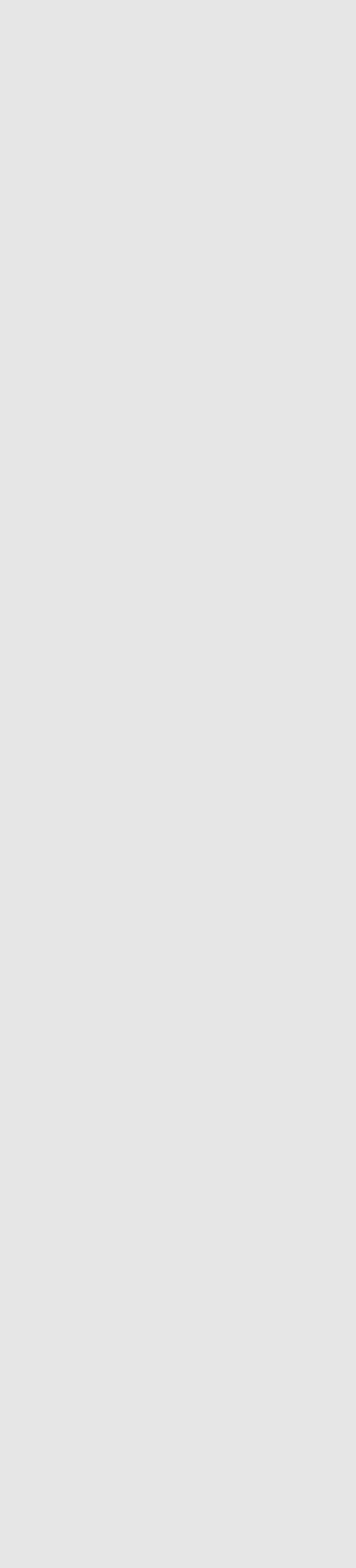
每道思考题我都给出了一些个人的思考引导，帮助大家建立起知识点之间的串联关系，彻底理解这个模块中所涉及的知识点。

记录与分享

我个人习惯已经持续了三年了，写博客是一个很好的习惯，一方面能帮助自己理解知识，另一方面也能锻炼个人的影响力，所以我也很推荐大家能养成这个习惯。

为了激励大家更有动力的去记录与分享，我后面会单独用一个章节的内容去存放我认为写得不错的博客，如果你想对知识点有所记录，或者分享解答面试题、思考题的个人理解，都可以在评论中给出你的分享地址，我都会认真地去阅读，挑出好的内容单独放入一个章节中，这样就有更多的人能看到你的分享。

最后，这本小册不一定能让你在很短的时间内就让你的技术一夜突飞猛进，但是如果你能按照我说的话，绝对能让你醍醐灌顶。好了，食用指南结束了，接下来让我们进入小册的知识海洋吧。



JS 基础知识点及常考面试题 (一)

JS 对于每位前端开发都是必备技能，在小册中我们也会有多个章节去讲述这部分的知识。首先我们先来熟悉下 JS 的一些常见和容易混淆的基础知识点。

原始 (Primitive) 类型

涉及到面试题：原始类型有哪些？null 是对象吗？

在 JS 中，存在着 6 种原始值，分别是：

- boolean
- null
- undefined
- number
- string
- symbol

首先原始类型的都是值，是没有函数可以调用的，比如 `undefined.toString()`

```
> undefined.toString()
✖ TypeError: Cannot read property 'toString' of undefined
```

此时你肯定会有疑问，这不对啊，明明 `'1'.toString()` 是可以使用的。其实在这种情况下，`'1'` 已经不是原始类型了，而是被强制转换成了 `string` 类型也就是对象类型，所以可以调用 `toString` 函数。

除了会在必要的情况下强制类型以外，原始类型还有一些坑。

其中 JS 的 `number` 类型是浮点类型的，在使用中会遇到某些 Bug，比如 `0.1 + 0.2 != 0.3`，但说这一块的内容会在进阶部分讲到。`string` 类型是不可变的，无论你在 `string` 类型上调用任何方法，都不会对值有改变。

另外对于 `null` 来说，很多人会认为这是个对象类型，其实这是错误的。虽然 `typeof null` 会输出 `object`，但是 JS 存在的一个 bug，当你创建了一个对象类型的时候，计算机会在内存中帮我们开辟一个空间来存放值，但是我们需要找到这个空间，这个空间会拥有一个地址（指针）。

```
const a = []
```

对于常量 `a` 来说，假设内存地址（指针）为 `#001`，那么在地址 `#001` 的位置存放了值 `[]`，常量 `a` 存放了地址（指针） `#001`，再看以下代码

```
const a = []
const b = a
b.push()
```

当我们把变量 `a` 复制给另外一个变量时，复制的是原本变量的地址（指针），也就是说当前变量 `b` 存放的地址（指针）也是 `#001`，当我们进行数据修改的时候，就会修改存放在地址（指针） `#001` 的值，也就导致了两个变量的值都发生了改变。

接下来我们再看函数参数是对象的情况

```
function test(person) {
  person.age = 20
  person.name = 'l'
  age: 20
}

return person
}

const p1 = {
  name: 'yin',
  age: 20
}

const p2 = test(p1)
console.log(p1) // => {name: "yin", age: 20}
console.log(p2) // => {name: "yin", age: 20}
```

对于以上代码，你是否能正确的写出结果呢？接下来让我为你解析一番：

- 首先，函数参数是对象类型的属性，我们都知道，当前 `p1` 的值也被修改了
- 但是当我们重新为 `person` 分配了一个对象时就出现了分歧，请看下图



所以最后 `person` 拥有了一个新的地址（指针），也就和 `p1` 没有任何关系了，导致了最终两个变量的值不相同的。

typeof vs instanceof

涉及到面试题：typeof 是什么类型的判断？instanceof 是什么类型的判断？为什么？

`typeof` 对于原始类型来说，除了 `null` 都可以显示正确的类型

```
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

```
typeof {} // "object"
typeof Object // "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，在后面的章节中我们也会自己去实现一个 `instanceof`。

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

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

```
var str = new String('Hello world')
str instanceof String // true
```

对于原生类型来说，你想直接通过 `instanceof` 来判断类型是不行的，当然我们还是有办法让 `instanceof` 判断原生类型的

```
class PrimitiveString {
  static [Symbol.hasInstance](x) {
    return typeof x === 'string'
  }
}
console.log('Hello world' instanceof PrimitiveString) // true
```

你可能不知道 `Symbol.hasInstance` 是什么东西，其实就是一个让你自定义 `instanceof` 行为的代理参见 `Object.create()`，所以如果想要 `instanceof` 行为和字符串一样，那么你只需要重写 `Symbol.hasInstance` 方法，然后是 `true` 了，这其实也侧面反映了一个问题，`instanceof` 也不是百分之百可靠的。

类型转换

涉及到面试题：请问如何将对象转换为字符串？

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

我们先来看一个类型转换表格，然后进入正题

涉及到面试题：请说出 `NaN` 和 `undefined` 在判断真假时的区别

```
let a = {}
a.valueOf() {
  return 0
}
a.toString() {
  return '1'
}
a + 1 // 1
```

如果你对于原生类型有疑惑的话，请看解析：

- 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，触发特点三，所以将数据通过 `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' // "ab"
```

因为 `'a' + 'b'` 等于 `NaN`，所以结果为 `"NaN"`，你可能也会在一些代码中看到过 `'+' + ''` 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

```
a * 1 // 12
a + 1 // 13
a [1,2,3] // 123
```

如果你还是觉得有点懵，那么就看以下的这张流程图吧，图中的流程只针对单个规则。

小结

以上就是我们 JS 基础知识点的第一部分内容了。这一小节中涉及到的知识点在我们日常的开发中经常可以看到，并且很容易出现的坑，熟悉了就能轻松解决这些问题了。

`this` 是很多人会混淆的概念，但是其实它一点都不难，只是网上很多文章把简单的东西说复杂了。在这一小节中，你一定会彻底明白 `this` 这个概念的。

我们先来看几个函数调用的场景

```
function foo() {
  console.log(this.a)
}

var a = 1
foo()

const obj = {
  a: 2,
  foo: foo
}

obj.foo() // undefined
```

涉及到面试题：请问如何判断 `this` 是什么？

```
let a = {
  valueOf() {
    return 0
  },
  toString() {
    return '1'
  }
}
a instanceof String // false
```

首先我们知道其实没有 `this` 的，箭头函数中的 `this` 只取决于前头函数的第一个普通函数的 `this`，在这个例子中，因为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` 这类方法是无效的。

最后这种情况也就是 `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？

```
let a = {
  valueOf() {
    return 0
  },
  toString() {
    return '1'
  }
}
a.bind() // => bind
```

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

```
//箭头函数
let a = () => {
  return '1'
}
a.bind() // => bind
```

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

```
let a = {
  name: 'yin'
}
function foo() {
  console.log(this.name)
}
a.foo() // undefined
```

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决定 `this` 最终指向哪里。

首先我们先看一个具体的例子，我们在前面已经提到了 `new` 这个操作符，所以 `typeof new` 就会返回 `object`。

```
new undefined // undefined
```

所以对于原生类型的 `undefined` 来说，`new` 之后的值就是 `undefined`，所以结果自然是 `undefined` 了。

```
new null // null
```

所以对于原生类型的 `null` 来说，`new` 之后的值就是 `null`，所以结果自然是 `null` 了。

```
new true // true
```

所以对于原生类型的 `true` 来说，`new` 之后的值就是 `true`，所以结果自然是 `true` 了。

```
new false // false
```

所以对于原生类型的 `false` 来说，`new` 之后的值就是 `false`，所以结果自然是 `false` 了。

```
new NaN // NaN
```

所以对于原生类型的 `Nan` 来说，`new` 之后的值就是 `Nan`，所以结果自然是 `Nan` 了。

```
new undefined // undefined
```

所以对于原生类型的 `undefined` 来说，`new` 之后的值就是 `undefined`，所以结果自然是 `undefined` 了。

```
new null // null
```

所以对于原生类型的 `null` 来说，`new` 之后的值就是 `null`，所以结果自然是 `null` 了。

```
new true // true
```

所以对于原生类型的 `true` 来说，`new` 之后的值就是 `true`，所以结果自然是 `true` 了。

```
new false // false
```

所以对于原生类型的 `false` 来说，`new` 之后的值就是 `false`，所以结果自然是 `false` 了。

```
new NaN // NaN
```

所以对于原生类型的 `Nan` 来说，`new` 之后的值就是 `Nan`，所以结果自然是 `Nan` 了。

```
new undefined // undefined
```

所以对于原生类型的 `undefined` 来说，`new` 之后的值就是 `undefined`，所以结果自然是 `undefined` 了。

```
new null // null
```

所以对于原生类型的 `null` 来说，`new` 之后的值就是 `null`，所以结果自然是 `null` 了。

```
new true // true
```

所以对于原生类型的 `true` 来说，`new` 之后的值就是 `true`，所以结果自然是 `true` 了。

```
new undefined // undefined
```

所以对于原生类型的 `undefined` 来说，`new` 之后的值就是 `undefined`，所以结果自然是 `undefined` 了。

```
new null // null
```

所以对于原生类型的 `null` 来说，`new` 之后的值就是 `null`，所以结果自然是 `null` 了。

```
new true // true
```

所以对于原生类型的 `true` 来说，`new` 之后的值就是 `true`，所以结果自然是 `true` 了。

```
new false // false
```

所以对于原生类型的 `false` 来说，`new` 之后的值就是 `false`，所以结果自然是 `false` 了。

```
new NaN // NaN
```

所以对于原生类型的 `Nan` 来说，`new` 之后的值就是 `Nan`，所以结果自然是 `Nan` 了。

```
new undefined // undefined
```

所以对于原生类型的 `undefined` 来说，`new` 之后的值就是 `undefined`，所以结果自然是 `undefined` 了。

```
new null // null
```

所以对于原生类型的 `null` 来说，`new` 之后的值就是 `null`，所以结果自然是 `null` 了。

```
new true // true
```

所以对于原生类型的 `true` 来说，`new` 之后的值就是 `true`，所以结果自然是 `true` 了。

```
new false // false
```

所以对于原生类型的 `false` 来说，`new` 之后的值就是 `false`，所以结果自然是 `false` 了。

```
new NaN // NaN
```

所以对于原生类型的 `Nan` 来说，`new` 之后的值就是 `Nan`，所以结果自然是 `Nan` 了。

```
new undefined // undefined
```

所以对于原生类型的 `undefined` 来说，`new` 之后的值就是 `undefined`，所以结果自然是 `undefined` 了。

```
new null // null
```

所以对于原生类型的 `null` 来说，`new` 之后的值就是 `null`，所以结果自然是 `null` 了。

```
new true // true
```

所以对于原生类型的 `true` 来说，`new` 之后的值就是 `true`，所以结果自然是 `true` 了。

```
new false // false
```

所以对于原生类型的 `false` 来说，`new`

JS 基础知识点及常考面试题 (二)

在这一章节中我们继续来了解 JS 的一些常考和易混的基础知识点。

`==` `===`

对于 `==` 来说，如果对双方的类型不一样的话，就会进行类型转换。这也就用到了我们上一章节讲的内容。

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：

- 首先会判断两者类型是否相同，相同的话就是比大小了
- 类型不相同的话，那么就会进行类型转换
- 会先检测是否在对比 `null` 和 `undefined`，是的话就返回 `true`
- 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`

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  
    ↓  
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: "yihui" }  
    ↓  
1 == object  
    ↓  
1 == object
```

思考：看完了上面的流程，对于 `1 == 1` 你是否能解释清楚呢？

如果你觉得记忆步骤太麻烦的话，我还提供了流程图供大家使用：



当然了，这个流程图并没有将所有的情况都列举出来，我这里只将常用到的情况列举了，如果你想了解更多的内容可以参考 [标准文档](#)。

对于 `==` 來說就简单多了，就是判断两者类型和值是否相同。

更多的对比可以阅读这篇 [文章](#)

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。

```
function A() {  
  let a = 1;  
  return B => function () {  
    console.log(a);  
  };  
}  
A();  
B()
```

很多人对于闭包的解释可能是函数嵌套了函数，然后返回一个函数。其实这个解释是不完整的，就比如上面这个例子就可以反驳这个观点。

在 JS 中，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。

```
for (var i = 1; i < 5; i++) {  
  var timer = function timer() {  
    console.log(i);  
  };  
  i++;  
  timer();  
}
```

首先因为 `setTimeout` 是一个异步函数，所以会先执行循环全部执行完毕。这时候 `i` 就是 6 了，所以会输出一串 6。

解决办法有三种，第一种是使用闭包的方式

```
for (var i = 1; i < 5; i++) {  
  (function(i) {  
    setTimeout(function timer() {  
      console.log(i);  
    }, i * 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 `i` 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 `i` 从而达到目的。

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。

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

第三种就是使用 `Ie` 定义 `i` 了来解决这个问题。这个也是最为推荐的方式

```
for (var i = 1; i < 5; i++) {  
  let timer = function timer() {  
    console.log(i);  
  };  
  i++;  
  timer();  
}
```

对于这一小节的知识点，总结起来就是以下几点：

- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能识别函数
- 不能识别循环引用的对象

```
let obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
let result = JSON.stringify(obj);  
console.log(result) // {}
```

当然你可能想自己来实现一个深拷贝。但是其实实现一个深拷贝是很困难的，需要我们考虑好多种边界情况，比如原型链如何处理，DOM 如何处理等等。所以这里我们实现的深拷贝只是最基础，并且我其实更推荐使用 `lodash` 的深拷贝函数。

```
function deepClone(obj) {  
  return new Promise(resolve => {  
    const [part1, part2] = new MessageChannel();  
    part1.onmessage = ev => resolve(ev.data);  
    part2.postMessage(obj);  
  });  
}  
  
var obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
deepClone(obj).then(result => console.log(result)) // {}
```

深拷贝只能解决了第一层的问题，如果递归下去的话还有对象的话，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，我们就得使用浅拷贝了。

但是在通常情况下，复杂数据都是可以序列化的，所以这个函数可以解决大部分问题。

如果你所需的对象含有内部类型并且不包含函数，可以使用 `MessageChannel`

```
#section structureClone(obj) {  
  return new Promise(resolve => {  
    const [part1, part2] = new MessageChannel();  
    part1.onmessage = ev => resolve(ev.data);  
    part2.postMessage(obj);  
  });  
}  
  
var obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
structureClone(obj).then(result => console.log(result)) // {}
```

对于这一小节的知识点，总结起来就是以下几点：

- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能识别函数
- 不能识别循环引用的对象

```
let obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
let result = JSON.stringify(obj);  
console.log(result) // {}
```

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

```
let a = {  
  age: 1,  
  job: 1,  
  first: '1st'  
};  
let b = {  
  ...a  
};  
a.job.first = '2nd';  
console.log(b.job.first) // 1st
```

浅拷贝只是将所有的属性都复制了一份，所以当我们修改其中一个属性时，另一个属性也会跟着变。

我们在浏览器中打印 `obj` 你会发现还有一个 `__proto__` 属性，那么看来之前的结论就和这个属性有关了。

其实每个 JS 对象又可以发现其中还有一个 `prototype` 属性，而且这个属性对应的值我们之前在 `Object` 中看到的一样。所以我们又可以得出一个结论：原型的 `constructor` 属性指向构造函数，函数函数又通过 `prototype` 属性指向原型，但是并不是所有的函数都具有这个属性。`Function.prototype.bind()` 就没有这个属性。

```
#section深拷贝  
function deepClone(obj) {  
  return new Promise(resolve => {  
    const [part1, part2] = new MessageChannel();  
    part1.onmessage = ev => resolve(ev.data);  
    part2.postMessage(obj);  
  });  
}  
  
var obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
deepClone(obj).then(result => console.log(result)) // {}
```

深拷贝解决不了第一层的问题，如果递归下去的话还有对象的话，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，我们就得使用浅拷贝了。

但是在通常情况下，复杂数据都是可以序列化的，所以这个函数可以解决大部分问题。

如果你所需的对象含有内部类型并且不包含函数，可以使用 `MessageChannel`

```
#section structureClone(obj) {  
  return new Promise(resolve => {  
    const [part1, part2] = new MessageChannel();  
    part1.onmessage = ev => resolve(ev.data);  
    part2.postMessage(obj);  
  });  
}  
  
var obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
structureClone(obj).then(result => console.log(result)) // {}
```

对于这一小节的知识点，总结起来就是以下几点：

- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能识别函数
- 不能识别循环引用的对象

```
let obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
let result = JSON.stringify(obj);  
console.log(result) // {}
```

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

```
let a = {  
  age: 1,  
  job: 1,  
  first: '1st'  
};  
let b = {  
  ...a  
};  
a.job.first = '2nd';  
console.log(b.job.first) // 1st
```

浅拷贝只是将所有的属性都复制了一份，所以当我们修改其中一个属性时，另一个属性也会跟着变。

我们在浏览器中打印 `obj` 你会发现还有一个 `__proto__` 属性，那么看来之前的结论就和这个属性有关了。

其实每个 JS 对象又可以发现其中还有一个 `prototype` 属性，而且这个属性对应的值我们之前在 `Object` 中看到的一样。所以我们又可以得出一个结论：原型的 `constructor` 属性指向构造函数，函数函数又通过 `prototype` 属性指向原型，但是并不是所有的函数都具有这个属性。`Function.prototype.bind()` 就没有这个属性。

```
#section深拷贝  
function deepClone(obj) {  
  return new Promise(resolve => {  
    const [part1, part2] = new MessageChannel();  
    part1.onmessage = ev => resolve(ev.data);  
    part2.postMessage(obj);  
  });  
}  
  
var obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
deepClone(obj).then(result => console.log(result)) // {}
```

深拷贝解决不了第一层的问题，如果递归下去的话还有对象的话，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，我们就得使用浅拷贝了。

但是在通常情况下，复杂数据都是可以序列化的，所以这个函数可以解决大部分问题。

如果你所需的对象含有内部类型并且不包含函数，可以使用 `MessageChannel`

```
#section structureClone(obj) {  
  return new Promise(resolve => {  
    const [part1, part2] = new MessageChannel();  
    part1.onmessage = ev => resolve(ev.data);  
    part2.postMessage(obj);  
  });  
}  
  
var obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
structureClone(obj).then(result => console.log(result)) // {}
```

对于这一小节的知识点，总结起来就是以下几点：

- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能识别函数
- 不能识别循环引用的对象

```
let obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
let result = JSON.stringify(obj);  
console.log(result) // {}
```

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

```
let a = {  
  age: 1,  
  job: 1,  
  first: '1st'  
};  
let b = {  
  ...a  
};  
a.job.first = '2nd';  
console.log(b.job.first) // 1st
```

浅拷贝只是将所有的属性都复制了一份，所以当我们修改其中一个属性时，另一个属性也会跟着变。

我们在浏览器中打印 `obj` 你会发现还有一个 `__proto__` 属性，那么看来之前的结论就和这个属性有关了。

其实每个 JS 对象又可以发现其中还有一个 `prototype` 属性，而且这个属性对应的值我们之前在 `Object` 中看到的一样。所以我们又可以得出一个结论：原型的 `constructor` 属性指向构造函数，函数函数又通过 `prototype` 属性指向原型，但是并不是所有的函数都具有这个属性。`Function.prototype.bind()` 就没有这个属性。

```
#section深拷贝  
function deepClone(obj) {  
  return new Promise(resolve => {  
    const [part1, part2] = new MessageChannel();  
    part1.onmessage = ev => resolve(ev.data);  
    part2.postMessage(obj);  
  });  
}  
  
var obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
deepClone(obj).then(result => console.log(result)) // {}
```

深拷贝解决不了第一层的问题，如果递归下去的话还有对象的话，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，我们就得使用浅拷贝了。

但是在通常情况下，复杂数据都是可以序列化的，所以这个函数可以解决大部分问题。

如果你所需的对象含有内部类型并且不包含函数，可以使用 `MessageChannel`

```
#section structureClone(obj) {  
  return new Promise(resolve => {  
    const [part1, part2] = new MessageChannel();  
    part1.onmessage = ev => resolve(ev.data);  
    part2.postMessage(obj);  
  });  
}  
  
var obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
structureClone(obj).then(result => console.log(result)) // {}
```

对于这一小节的知识点，总结起来就是以下几点：

- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能识别函数
- 不能识别循环引用的对象

```
let obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
let result = JSON.stringify(obj);  
console.log(result) // {}
```

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

```
let a = {  
  age: 1,  
  job: 1,  
  first: '1st'  
};  
let b = {  
  ...a  
};  
a.job.first = '2nd';  
console.log(b.job.first) // 1st
```

浅拷贝只是将所有的属性都复制了一份，所以当我们修改其中一个属性时，另一个属性也会跟着变。

我们在浏览器中打印 `obj` 你会发现还有一个 `__proto__` 属性，那么看来之前的结论就和这个属性有关了。

其实每个 JS 对象又可以发现其中还有一个 `prototype` 属性，而且这个属性对应的值我们之前在 `Object` 中看到的一样。所以我们又可以得出一个结论：原型的 `constructor` 属性指向构造函数，函数函数又通过 `prototype` 属性指向原型，但是并不是所有的函数都具有这个属性。`Function.prototype.bind()` 就没有这个属性。

```
#section深拷贝  
function deepClone(obj) {  
  return new Promise(resolve => {  
    const [part1, part2] = new MessageChannel();  
    part1.onmessage = ev => resolve(ev.data);  
    part2.postMessage(obj);  
  });  
}  
  
var obj = {  
  a: 1,  
  b: 1,  
  c: 1,  
  d: 1,  
};  
obj.c = obj.b;  
obj.a = obj.c;  
obj.b = obj.c;  
obj.d = obj.b;  
deepClone(obj).then(result => console.log(result)) // {}
```

ES6 知识点及常考面试题

本章节我们来学习 ES6 部分的内容。

var, let 及 const 区别

相关面试题：什么是提升？什么是暂时性死区？var, let 及 const 区别？

对于这个问题，我们应该先来了解提升（hoisting）这个概念。

```
console.log(a) // undefined
var a = 1
```

从上述代码中我们可以发现，虽然变量还没有被声明，但是我们却可以使用这个未被声明的变量，这种情况就叫做提升。并且提升的是声明。

对于这种情況，我们可以把代码这样来看：

```
var a
console.log(a) // undefined
a = 1
```

接下来我们再来看一个例子：

```
var a = 10
var a
console.log(a)
```

对于这个例子，如果你认为打印的值为 `undefined` 那么就错了。答案应该是 `10`，对于这种情况，我们这样来看代码：

```
var a
var a
a = 10
console.log(a)
```

到这里为止，我们已经了解了 `var` 声明的变量会发生提升的情况，其实不仅变量会提升函数也会提升：

```
function f() {
  console.log(a)
}
function f() {
  console.log(a)
}
var a = 1
```

对于上述代码，打印结果会是 `f a ()`，即使变量声明在函数之后，这也说明了函数会被提升，并且优先于变量提升。

说完了这些，想必大家也知道 `var` 存在的问题了，使用 `var` 声明的变量会被提升到作用域的顶部，接下来我们再来看 `let` 和 `const`。

我们先来看一个例子：

```
let a = 1
let b = 1
const c = 1
console.log(window.a) // undefined
console.log(window.c) // undefined
```

```
function test1() {
  console.log(a)
}
function test2() {
  console.log(a)
}
test1()
test2()
```

首先在全局作用域下使用 `let` 和 `const` 声明变量，变量并不会被挂载到 `window` 上，这一点和 `var` 声明有了区别。

再者当我们在声明 `a` 之前如果使用了 `a`，就会出现报错的情况。

```
① Uncaught ReferenceError: a is not defined
at test1:1:1
at window.onload:1:1
```

你可能会认为这里也出现了提升的情况，但是因为某些原因导致不能访问。

首先报错的原因是因为存在暂时性死区，我们不能在声明前就使用变量，这也是 `let` 和 `const` 优于 `var` 的一点，然后这里你认为的提升和 `var` 的提升是有区别的，虽然变量在编译的环节中被告知在块作用域中可以访问，但是访问是受限限制的。

那么到这里，想必大家也都明白 `var`、`let` 及 `const` 这区别了，不知道你是否会有这么一个疑问，为什么要存在提升这个事情呢？其实提升存在的根本原因是为了解决函数互相调用的情况。

```
function test1() {
  test2();
}
function test2() {
  test1();
}
```

假如不存在提升这个情况，那么就实现不了上述的代码，因为不可能存在 `test1` 在 `test2` 前面然后 `test2` 又在 `test1` 前面。

那么最后我们总结下这一小节的内容：

- 函数提升优先于变量提升，函数提升会把整个函数都作用域顶部，变量提升只会把声明提升到作用域顶部
- `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赋值

原型继承和 Class 继承

相关面试题：类型如何实现继承？class 如何实现继承？class 本质是什么？

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

```
class Person {
  constructor() {
    this.name = 'John';
  }
}
```

在上一章节中我们讲解了原型的知识点，在这一小节中我们将会分别使用原型和 `class` 的方式来实现继承。

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，我们只需要优化这点就行了。

```
function Parent(value) {
  this.val = value;
}
Parent.prototype.getVal = function() {
  console.log(this.val);
}

function Child(value) {
  Parent.call(this, value);
}
Child.prototype = new Parent();
Child.prototype.constructor = Child;

const child = new Child(1);

child.getVal(); // 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，并将构造函数设置为子类，这样既解决了无用的父类属性，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，我们只需要优化这点就行了。

```
function Parent(value) {
  this.val = value;
}
Parent.prototype.getVal = function() {
  console.log(this.val);
}

function Child(value) {
  Parent.call(this, value);
}
Child.prototype = Object.create(Parent.prototype);
Child.prototype.constructor = Child;
```

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复用，不会与父类引起属性共享，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。

Class 继承

以上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。

```
class Parent {
  constructor(value) {
    this.val = value;
  }
  getVal() {
    console.log(this.val);
  }
}

class Child extends Parent {
  constructor(value) {
    super(value);
  }
}

Child.prototype = new Parent();
Child.prototype.constructor = Child;
```

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

Class 混合

以上实现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。

```
class Parent {
  constructor(value) {
    this.val = value;
  }
  getVal() {
    console.log(this.val);
  }
}

class Child extends Parent {
  constructor(value) {
    super(value);
  }
}

Child.prototype = Object.create(Parent.prototype);
Child.prototype.constructor = Child;
```

以上继承实现的核心在于使用 `Object.create` 表现继承自多个父类，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)`。

当然了，之前也说了在 JS 中并不存在类，`class` 的本质就是函数。

模块化

相关面试题：为什么使用模块化？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
- 提供可复用性
- 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() {
    // ...
  }
})(globalVariable)
```

如果你平时有关注 Vue 的进阶的话，可能已经知道了在 Vue3.0 中将会通过 `Proxy` 来替换原本的 `Object.defineProperty` 来实现数据响应式。`Proxy` 是 ES6 中新增的功能，它可以用来自定义对对象中的操作。

```
let p = new Proxy(target, handler)
```

`target` 代表需要添加代理的对象，`handler` 用来自定义对象的操作，比如可以用来自定义 `set` 或者 `get` 函数。

接下来我们通过 `Proxy` 来实现一个数据响应式。

```
let search = (obj, setObj, genObj) => {
  let handle = {
    get(target, property) {
      getObj(target, property);
    },
    set(target, property, receiver) {
      setObj(target, property, receiver);
    }
  };
  return new Proxy(obj, handle);
}
```

在上述代码中，我们通过自定义 `set` 和 `get` 函数的方式，在原本的逻辑中插入了我们的函数逻辑，实现了对对象属性进行读写时发出通知。

AMD 和 CMD

至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。

```
// AMD
define(['./a', './b'], function(a, b) {
  // ...
})
```

这种实现方式是将模块的入口放在最前面，然后自己再去往里加载了库，单单靠这个小例子是远远不够的。

```
// CMD
define(function(require, exports, module) {
  // ...
});
```

在上一章节中我们讲解了模块的知识点，在这一小节中我们将会分别使用原型和 `class` 的方式来实现继承。

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。

```
// a.js
module.exports = {
  a: 1
}
// b.js
var a = require('./a');
a.a // 1
```

因为 CommonJS 还是会使用到的，所以这里会对一些疑难点进行解析。

先说 `require` 吧

```
var module = require('./a');
module.a // 1
```

这时你可能会疑惑，为什么使用模块化后都像从全局对象中取值一样，而不是像 `var` 一样直接取值呢？

```
var module = require('./a');
module.a // 1
```

这是因为 `require` 语句会自动将模块的路径转换为相对路径，所以 `require` 语句的值实际上是模块的绝对路径。

```
var module = require('./a');
module.a // 1
```

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 = {}`，所以如果直接对 `exports` 赋值，那么在全局范围内就无法再访问到 `module.exports` 了。

ES Module

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

- 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
- CommonJS 是同步导入，因为用于服务器端，文件都在本地，同步导入即使卡住线程影响也不大。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影响到同一个内存地址，必须重新导入一次。但是 ES Module 异步导入时绑定的值也不会指向同一内存地址，修改并不会对 `module.exports` 造成影响。
- ES Module 会将模块 `require,exports` 来执行的

```
// 先从模块 a
import a from './a';
// 然后从模块 b 导入 a
import { a } from './b';
// 然后从模块 c 导入 b
import { a } from './c';
// 然后从模块 d 导入 c
import { a } from './d';
```

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

Class 继承

以上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。

```
class Parent {
  constructor(value) {
    this.val = value;
  }
  getVal() {
    console.log(this.val);
  }
}

class Child extends Parent {
  constructor(value) {
    super(value);
  }
}

Child.prototype = new Parent();
Child.prototype.constructor = Child;
```

以上继承实现的核心在于使用 `Object.create` 表现继承自多个父类，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)`。

当然了，之前也说了在 JS 中并不存在类，`class` 的本质就是函数。

模块化

相关面试题：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
- 提供可复用性
- 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() {
    // ...
  }
})(globalVariable)
```

如果你平时有关注 Vue 的进阶的话，可能已经知道了在 Vue3.0 中将会通过 `Proxy` 来替换原本的 `Object.defineProperty` 来实现数据响应式。`Proxy` 是 ES6 中新增的功能，它可以用来自定义对对象中的操作。

```
let p = new Proxy(target, handler)
```

`target` 代表需要添加代理的对象，`handler` 用来自定义对象的操作，比如可以用来自定义 `set` 或者 `get` 函数。

接下来我们通过 `Proxy` 来实现一个数据响应式。

```
let search = (obj, setObj, genObj) => {
  let handle = {
    get(target, property) {
      getObj(target, property);
    },
    set(target, property, receiver) {
      setObj(target, property, receiver);
    }
  };
  return new Proxy(obj, handle);
}
```

在上述代码中，我们通过自定义 `set` 和 `get` 函数的方式，在原本的逻辑中插入了我们的函数逻辑，实现了对对象属性进行读写时发出通知。

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。

```
// a.js
module.exports = {
  a: 1
}
// b.js
var a = require('./a');
a.a // 1
```

因为 CommonJS 还是会使用到的，所以这里会对一些疑难点进行解析。

先说 `require` 吧

```
var module = require('./a');
module.a // 1
```

这时你可能会疑惑，为什么使用模块化后都像从全局对象中取值一样，而不是像 `var` 一样直接取值呢？

```
var module = require('./a');
module.a // 1
```

这是因为 `require` 语句会自动将模块的路径转换为相对路径，所以 `require` 语句的值实际上是模块的绝对路径。

ES Module

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

- 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
- CommonJS 是同步导入，因为用于服务器端，文件都在本地，同步导入即使卡住线程影响也不大。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影响到同一个内存地址，修改并不会对 `module.exports` 造成影响。
- ES Module 会将模块 `require,exports` 来执行的

```
// 先从模块 a
import a from './a';
// 然后从模块 b 导入 a
import { a } from './b';
// 然后从模块 c 导入 b
import { a } from './c';
// 然后从模块 d 导入 c
import { a } from './d';
```

以上继承实现的核心就是将父类的属性赋值给了子类，并将构造函数设置为子类，这样既解决了无用的父类属性，还能正确的

查看更多回复

sophie组 | 2年前

CommonJS 和 ES6 module 区别: [blog.csdn.net](#)

0 回复 0 评论

sophie组 | 2年前

commonjs实现: [blog.csdn.net](#), 可以看到 6

0 回复 0 评论

sophie组 | 2年前

class extends 会创建第二张原型链 [blog.csdn.net](#)

0 回复 0 评论

sophie组 | 2年前

优化类 事件流包的 5 个技巧 [blog.jobbole.com](#) 有关于 Map, 链表, Array.some, Array.every,

Array.includes 的用处

0 回复 0 评论

查看全部 164 条回复

JS 异步编程及常考面试题

在上一章节中我们了解了常见 ES6 语句的一些知识点。这一章节我们将会学习异步编程这一块的内容，这是异步编程是 JS 中至关重要的内容，所以面试官会用三个章节来学习异步编程涉及到的重点和难点，同时这一块内容也是面试常考范围，希望大家认真学习。

并发 (concurrency) 和并行 (parallelism) 区别

面试问题：并发与并行的区别？

并发和并行这两个词其实并不是一个概念，但是这两个名词确实是很多人都会混淆的知识点。其实根本的原因可能只是两个名词在中文上的相似，在英文上来说完全是不同的单词。

并发是宏观概念，我们分派的任务 A 和任务 B，在一段时间内通过任务间的切换完成了这两个任务，这种情况下就可以称之为并发。

并行是微观概念，假设 CPU 中存在两个核心，那么我就可以同时完成任务 A、B，同时完成多个任务的情况就可以称之为并行。

回调函数 (Callback)

面试问题：什么是回调函数？回调函数有什么缺点？如何解决回调地狱问题？

回调函数应该是大家经常使用到的，以下代码就是一个回调函数的例子：

```
ajax(url, () => {
  // 处理逻辑
})
```

但是回调函数有一个致命的弱点，就是容易写出回调地狱（Callback hell），假设多个请求存在依赖性，你可能会写出如下代码：

```
ajax(url1, () => {
  // 处理逻辑
  ajax(url2, () => {
    // 处理逻辑
    ajax(url3, () => {
      // 处理逻辑
    })
  })
})
```

以上代码看起来不利于阅读和维护，当然，你可能会想说解决这个问题还很简单，把函数分开来写不就得了。

```
function firstAjax() {
  ajax(url1, () => {
    // 处理逻辑
    secondAjax()
  })
}

function secondAjax() {
  ajax(url2, () => {
    // 处理逻辑
    thirdAjax()
  })
}

function thirdAjax() {
  // 处理逻辑
}
```

以上的代码虽然看上去利于阅读了，但是还是没有解决根本问题。

回调地狱的根源问题就是：

1. 序列化操作存在耦合性，一旦有所改动，就会牵一发动全身
2. 序列化操作多，容易处理错误

当然，回调函数还存在别的几个缺点，比如不能使用 `try catch` 捕获错误，不能直接 `return`，在接下来的几小节中，我们将来学习通过别的技术解决这些问题。

Generator

面试问题：你理解 generator 是怎么工作的？

Generator 是 ES6 中挺有趣的概念之一了，`Generator` 最大的特点就是可以控制函数的执行，在这一小节中我们不会去讲什么是 `Generator`，而是把重点放在 `Generator` 的一些易用图标的地方。

```
function *foo(x) {
  let y = 2 * (yield (x + 5));
  return (x + y + 1)
}

let it = foo();
console.log(it.next()) // { value: undefined, done: false}
console.log(it.next(12)) // { value: 12, done: false}
console.log(it.next(12)) // { value: 28, done: true}
```

你也许会疑惑为什么会产生与你预期不同的值，接下来就让我为你进行代码分析原因

- 首先 `Generator` 函数实现和普通函数不同，它会返回一个迭代器
- 当执行第一次 `next` 时，参数会被忽略，并且函数暂停在 `yield` 处，所以返回 `{ value: undefined, done: false}`
- 当执行第二次 `next` 时，传入的参数等于上一个 `yield` 的返回值，如果你不传参，`yield` 永远返回 `undefined`，此时 `let y = 2 * 12`，所以第二个 `yield` 等于 `2 * 12 / 1 = 24`
- 当执行第三次 `next` 时，传入的参数会传递给 `x`，所以 `x = 12, y = 24`，相加等于 `48`

`Generator` 函数一般见到的不多，其实也和有点关系，而且一般会配合 `co` 库去使用，当然，我们可以通过 `Generator` 函数解决回调地狱的问题，可以把之前的回调地狱例子改写为如下代码：

```
function *foo(x) {
  yield x;
  yield x + 1;
  yield x + 2;
}

let it = foo();
let result = it.next();
let result = it.next();
let result = it.next();
```

当我们在构造 `Promise` 的时候，构造函数内部的代码是立即执行的

```
new Promise((resolve, reject) => {
  resolve('success')
}) // 无法
reject('reject')
})
```

`Promise` 实现了链式调用，也就是说每次调用 `then` 之后返回的都是一个 `Promise`，并且是一个全新的 `Promise`，原因也是因为状态不可变。如果你在 `then` 中使用了 `return`，那么 `return` 的值会被 `Promise.resolve()` 包装

```
Promise.resolve(1)
.then(res => {
  console.log(res) // => 1
  return res
})
.then(res => {
  console.log(res) // => 1
  return res
})
```

当然了，`Promise` 也很好地解决了回调地狱的问题，可以把之前的回调地狱例子改写为如下代码：

```
ajax(url)
.then(res => {
  console.log(res)
  return res
})
.then(res => {
  console.log(res)
  return res
})
```

以上代码在浏览器环境中，如果定时器执行过程中出现了耗时操作，多个回调函数会在耗时操作结束后同时执行，这样可能就会带来性能上的问题。

如果你有微弱定时器的需求，其实完全可以通过 `requestAnimationFrame` 来实现

面试问题：`async` 及 `await` 的特点，它们的优点和缺点分别是什么？`await` 原理是什么？

一个函数如果加上 `async`，那么该函数就会返回一个 `Promise`

```
async function test() {
  return "1"
}

console.log(test()) // => Promise {<resolved>: "1"}
```

`async` 就是将函数返回值用 `Promise.resolve()` 包裹了下，和 `then` 中处理返回值一样。

并且 `await` 只能配套 `async` 使用

```
async function test() {
  let value = await sleep()
  return value
}
```

`async` 和 `await` 可以说是异步将极解决方案，相比直接使用 `Promise` 来说，优势在于处理 `then` 的调用链，能够更清晰地写出代码。毕竟写一大块 `then` 也很恶心，并且也能优雅地解决回调地狱问题。当然也存在一些缺点，因为 `await` 将异步代码改造成了同步代码，如果多个异步操作没有依赖性却使用了 `await` 会导致性能上的降低。

```
// 代码块 test() {
  // 代码块
  await sleep();
  // 代码块
}
```

下面来看一个使用 `await` 的例子：

```
let a = 0
let b = await 10
a = a + b
console.log(`2: ${a}`) // => 2: 10
}
```

对于以上代码你可能会有疑惑，让我来解释下原因

- 首先 `await` 先执行，在执行到 `await 10` 之前变量 `a` 还是 0，因为 `await` 内部实现了 `generator`，`generator` 会保留操作的东西，所以这时候 `a = 0` 被保存了下来
- 因为 `await` 是异步操作，后来的表达式不返回 `Promise` 的话，就会包装成 `Promise.resolve()`，然后会去执行函数外的同步代码
- 同步代码执行完毕后开始执行异步代码，将保存下来的值拿出来使用，这时候 `a = 0 + 10`

上述解析的逻辑，内部实现了 `generator`，其实 `await` 就是 `generator` 加上 `Promise` 的结合，内部实现了自动执行，如果在 `await` 之前调用了 `return`，其实自己就可以实现这样的语法糖。

常用定时器函数

面试问题：`setInterval`、`setInterval`、`requestAnimationFrame` 各有什么特点？

异步编程当然并非一无是处，常见的定时器函数有 `setInterval`、`setInterval`、`requestAnimationFrame`，我们先来讲最常用的 `setInterval`，很多人认为 `setTimeout` 是延时多久，那就应该是多久后执行。

其实这个观点是错误的，因为 JS 是单线程执行的，如果前面的代码影响了性能，就会导致 `setTimeout` 不会按期执行，当然了，我们可以通过代码去修正 `setTimeout`，从而使定时器对准时。

如果你有微弱定时器的需求，其实完全可以通过 `requestAnimationFrame` 来实现

```
function draw() {
  requestAnimationFrame(draw);
  console.log('draw')
}, 1000
sleep(1000)
})
```

首先 `requestAnimationFrame` 有自带递归功能，基本可以保证在 16ms 高帧内只执行一次（不掉帧的情况下），并且该函数的延时效果是精确的。没有其他定时器时间不准的问题，当然你也可以通过读取函数来实现 `setTimeout`。

小结

异步编程是 JS 中比较简单的一块内容，同时也是很重要的知识点，以上提到的每个知识点其实都可以作为一道面试题。希望大家可以好好掌握以上内容如果大家对于这个章节的内容存在疑问，欢迎在评论区与我互动。

附录相关的内容并非一无是处，需要慢慢阅读和积累。

面试问题：`Promise` 的特点是什么？分别有什么优缺点？什么是 `Promise` 错误？`Promise` 错误通常执行和 `then` 阶段执行

`Promise` 跟过来就是承诺的意思，这个承诺会在未来有一个确切的答复，并且该承诺有三种状态，分别是：

1. 等待中 (`pending`)
2. 完成了 (`resolved`)
3. 抛出了 (`rejected`)

这个承诺一旦等待状态变成其他状态就永远不能更改状态了，也就是说一旦状态变为 `resolved` 后，就不能再次改变

```
new Promise((resolve, reject) => {
  resolve('success')
}) // 无法
reject('reject')
})
```

你也许会疑惑为什么会产生与你预期不同的值，接下来就让我为你进行代码分析原因

- 首先 `Promise` 函数实现和普通函数不同，它会返回一个迭代器
- 当执行第一次 `next` 时，参数会被忽略，并且函数暂停在 `yield` 处，所以返回 `{ value: undefined, done: false}`
- 当执行第二次 `next` 时，传入的参数等于上一个 `yield` 的返回值，如果你不传参，`yield` 永远返回 `undefined`，此时 `let y = 2 * 12`，所以第二个 `yield` 等于 `2 * 12 / 1 = 24`
- 当执行第三次 `next` 时，传入的参数会传递给 `x`，所以 `x = 12, y = 24`，相加等于 `48`

`Promise` 函数一般见到的不多，其实也和有点关系，而且一般会配合 `co` 库去使用，当然，我们可以通过 `Generator` 函数解决回调地狱的问题，可以把之前的回调地狱例子改写为如下代码：

```
function *foo(x) {
  yield x;
  yield x + 1;
  yield x + 2;
}

let it = foo();
let result = it.next();
let result = it.next();
let result = it.next();
```

以上的代码虽然看上去利于阅读了，但是还是没有解决根本问题。

回调地狱的根源问题就是：

1. 序列化操作存在耦合性，一旦有所改动，就会牵一发动全身
2. 序列化操作多，容易处理错误

当然，回调函数还存在别的几个缺点，比如不能使用 `try catch` 捕获错误，不能直接 `return`，在接下来的几小节中，我们将来学习通过别的技术解决这些问题。

Generator

面试问题：你理解 generator 是怎么工作的？

Generator 是 ES6 中挺有趣的概念之一了，`Generator` 最大的特点就是可以控制函数的执行，在这一小节中我们不会去讲什么是 `Generator`，而是把重点放在 `Generator` 的一些易用图标的地方。

```
function *foo(x) {
  let y = 2 * (yield (x + 5));
  return (x + y + 1)
}

let it = foo();
console.log(it.next()) // { value: undefined, done: false}
console.log(it.next(12)) // { value: 12, done: false}
console.log(it.next(12)) // { value: 28, done: true}
```

以上的代码看起来不利于阅读和维护，当然，你可能会想说解决这个问题还很简单，把函数分开来写不就得了。

```
function firstAjax() {
  ajax(url1, () => {
    // 处理逻辑
    secondAjax()
  })
}

function secondAjax() {
  ajax(url2, () => {
    // 处理逻辑
    thirdAjax()
  })
}

function thirdAjax() {
  // 处理逻辑
}
```

以上的代码虽然看上去利于阅读了，但是还是没有解决根本问题。

回调地狱的根源问题就是：

1. 序列化操作存在耦合性，一旦有所改动，就会牵一发动全身
2. 序列化操作多，容易处理错误

当然，回调函数还存在别的几个缺点，比如不能使用 `try catch` 捕获错误，不能直接 `return`，在接下来的几小节中，我们将来学习通过别的技术解决这些问题。

Generator

面试问题：你理解 generator 是怎么工作的？

Generator 是 ES6 中挺有趣的概念之一了，`Generator` 最大的特点就是可以控制函数的执行，在这一小节中我们不会去讲什么是 `Generator`，而是把重点放在 `Generator` 的一些易用图标的地方。

```
function *foo(x) {
  let y = 2 * (yield (x + 5));
  return (x + y + 1)
}

let it = foo();
console.log(it.next()) // { value: undefined, done: false}
console.log(it.next(12)) // { value: 12, done: false}
console.log(it.next(12)) // { value: 28, done: true}
```

以上的代码看起来不利于阅读和维护，当然，你可能会想说解决这个问题还很简单，把函数分开来写不就得了。

```
function firstAjax() {
  ajax(url1, () => {
    // 处理逻辑
    secondAjax()
  })
}

function secondAjax() {
  ajax(url2, () => {
    // 处理逻辑
    thirdAjax()
  })
}

function thirdAjax() {
  // 处理逻辑
}
```

以上的代码虽然看上去利于阅读了，但是还是没有解决根本问题。

回调地狱的根源问题就是：

1. 序列化操作存在耦合性，一旦有所改动，就会牵一发动全身
2. 序列化操作多，容易处理错误

当然，回调函数还存在别的几个缺点，比如不能使用 `try catch` 捕获错误，不能直接 `return`，在接下来的几小节中，我们将来学习通过别的技术解决这些问题。

Generator

面试问题：你理解 generator 是怎么工作的？

Generator 是 ES6 中挺有趣的概念之一了，`Generator` 最大的特点就是可以控制函数的执行，在这一小节中我们不会去讲什么是 `Generator`，而是把重点放在 `Generator` 的一些易用图标的地方。

```
function *foo(x) {
  let y = 2 * (yield (x + 5));
  return (x + y + 1)
}

let it = foo();
console.log(it.next()) // { value: undefined, done: false}
console.log(it.next(12)) // { value: 12, done: false}
console.log(it.next(12)) // { value: 28, done: true}
```

以上的代码看起来不利于阅读和维护，当然，你可能会想说解决这个问题还很简单，把函数分开来写不就得了。

```
function firstAjax() {
  ajax(url1, () => {
    // 处理逻辑
    secondAjax()
  })
}

function secondAjax() {
  ajax(url2, () => {
    // 处理逻辑
    thirdAjax()
  })
}

function thirdAjax() {
  // 处理逻辑
}
```

以上的代码虽然看上去利于阅读了，但是还是没有解决根本问题。

回调地狱的根源问题就是：

1. 序列化操作存在耦合性，一旦有所改动，就会牵一发动全身
2. 序列化操作多，容易处理错误

当然，回调函数还存在别的几个缺点，比如不能使用 `try catch` 捕获错误，不能直接 `return`，在接下来的几小节中，我们将来学习通过别的技术解决这些问题。

Generator

面试问题：你理解 generator 是怎么工作的？

Generator 是 ES6 中挺有趣的概念之一了，`Generator` 最大的特点就是可以控制函数的执行，在这一小节中我们不会去讲什么是 `Generator`，而是把重点放在 `Generator` 的一些易用图标的地方。

```
function *foo(x) {
  let y = 2 * (yield (x + 5));
  return (x + y + 1)
}

let it = foo();
console.log(it.next()) // { value: undefined
```

手写 Promise

在上一章节中我们了解了 `Promise` 的一些基础点，在这一章节中，我们会通过手写一个符合 `Promise/A+` 规范的 `Promise` 来深入理解它，并且手写 `Promise` 也是一道大厂常考题，在进入正题之前，推荐各位阅读一下 `Promise/A+ 规范`，这样才能更好地理解这个章节的代码。

实现一个简易版 `Promise`

在完成符合 `Promise/A+` 规范的 `Promise` 之前，我们可以先来实现一个简易版 `Promise`，因为在面试中，如果你能实现出一个简层级的 `Promise` 基本可以过关了。

那么我们先来搭建构建函数的大体框架

```
const PENDING = 'pending'
const RESOLVED = 'resolved'
const REJECTED = 'rejected'

function MyPromise(...args) {
  const that = this
  that.state = PENDING
  that.value = null
  that.resolveCallbacks = []
  that.rejectCallbacks = []
  // 将参数 resolve 和 reject 传给
}

首先我们创建了三个常量用于表示状态，对于经常使用的一些值都应该通过常量来管理，便于开发和后期维护
在函数内部我们创建了三个常量: that, 因为代码可能会异步执行，用于获取正确的 this 对象
一开始 Promise 的状态应该是 PENDING
将参数 resolve 和 reject 作为参数，将它们放入 that 中
resolveCallbacks 和 rejectCallbacks 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 then 中的回调保存起来用于状态改变时使用
```

接下来我们来完善 `resolve` 和 `reject` 函数，添加在 `MyPromise` 构造函数内部

```
function resolve(value) {
  if (that.state === PENDING) {
    that.state = RESOLVED
    that.value = value
    that.resolveCallbacks.map(cb => cb(that.value))
  }
}

function reject(value) {
  if (that.state === PENDING) {
    that.state = REJECTED
    that.value = value
    that.rejectCallbacks.map(cb => cb(that.value))
  }
}

这两个函数代码类似，就一起解析了
```

首先两个函数都得判断当前状态是否为等待中，因为规范规定只有等待态才可以改变状态
将当前状态更改为对应状态，并且将传入的值赋值给 `value`

遍历回调数组并执行

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 {
  // resolve, reject
} catch(e) {
  reject(e)
}
```

实现很简单，执行传入的参数并将之前两个函数当成参数传进去
要注意的是，可能执行函数过程中会遇到错误，需要捕获错误并且执行 `reject` 函数

最后我们来实现较为复杂的 `then` 函数

```
MyPromise.prototype.then = function(resolve, reject) {
  const that = this
  const fulfilled = fulfilled => fulfilled ? fulfilled : <=>
  const rejected =
    typeof fulfilled == 'function'
      ? fulfilled
      : <=> {
        throw e
      }
  if (that.state === PENDING) {
    that.resolveCallbacks.push(fulfilled)
    that.rejectCallbacks.push(rejected)
  }
  if (that.state === RESOLVED) {
    fulfilled(that.value)
  }
  if (that.state === REJECTED) {
    rejected(that.value)
  }
}

首先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  
当参数不是函数类型时，需要创建一个函数赋值给对应的参数，同时也实现了透传，比如如下代码
```

```
// 代码出自高亮老师的课件
// 只是举个例子
Promise.resolve(x).then(y => console.log(y))
```

接下来就是一系列判断状态的逻辑，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` 函数，比如如下代码就会进入等待态的逻辑

```
if (that.state === PENDING) {
  return new MyPromise((resolve, reject) => {
    that.resolveCallbacks.push(() => {
      try {
        const x = fulfilled(that.value)
        resolveProcedure(promised, x, resolve, reject)
      } catch (e) {
        reject(e)
      }
    })
  })
}

首先我们返回了一个新的 Promise 对象，并在 Promise 中嵌入了一个函数  
函数的基本逻辑还是和之前一样，往回调数组中 push 函数  
同样，在执行函数的过程中可能会遇到错误，所以使用了 try...catch 包裹
```

规范规定，执行 `onFulfilled` 或者 `onRejected` 函数时会返回一个 `x`，并且执行 `Promise` 解决过程，这是为了不同的 `Promise` 都可以兼容使用，比如 `jQuery` 的 `Promise` 能兼容 `ES6` 的 `Promise`

接下来我们改造判断执行态的逻辑

```
if (that.state === RESOLVED) {
  return (x => new MyPromise((resolve, reject) => {
    that.resolveCallbacks.push(() => {
      try {
        const x = fulfilled(that.value)
        resolveProcedure(promised, x, resolve, reject)
      } catch (e) {
        reject(e)
      }
    })
  }))
}

对于 resolve 函数来说，首先需要判断传入的值是否为 Promise 类型  
为了保证函数执行顺序，需要将两个函数体代码使用 setTimeout 包裹起来
```

接下来继续改造 `then` 函数中的代码。首先我们需要新增一个变量 `promise2`，因为每个 `then` 函数都需要指向一个新的 `Promise` 对象。该变量用于保存新的返回对象，然后我们先来改造判断等待态的逻辑

```
if (that.state === PENDING) {
  return (x => new MyPromise((resolve, reject) => {
    that.resolveCallbacks.push(() => {
      try {
        const x = fulfilled(that.value)
        resolveProcedure(promised, x, resolve, reject)
      } catch (e) {
        reject(e)
      }
    })
  }))
}

首先我们返回了一个新的 Promise 对象，并在 Promise 中嵌入了一个函数  
函数的基本逻辑还是和之前一样，往回调数组中 push 函数  
同样，在执行函数的过程中可能会遇到错误，所以使用了 try...catch 包裹
```

规范规定，执行 `onFulfilled` 或者 `onRejected` 函数时会返回一个 `x`，并且执行 `Promise` 解决过程，这是为了不同的 `Promise` 都可以兼容使用，比如 `jQuery` 的 `Promise` 能兼容 `ES6` 的 `Promise`

完成以上两个改造以后，我们就可以开始实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x` 处于等待态，`Promise` 需保持为等待态直至 `x` 执行或拒绝

2. 如果 `x` 处于其他状态，则用相同的值处理 `Promise`

当然以上这些是规范需要我们判断的情况，实际上我们不判断状态也是可行的。

接下来我们继续按照规范来实现剩余的代码

```
let called = false
if (x === null || typeof x === 'object' || typeof x === 'function') {
  try {
    let value = x.value
    if (value === undefined) {
      value = x
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
    if (value === null) {
      value = undefined
    }
    if (value === undefined) {
      value = null
    }
  } catch (e) {
    reject(e)
  }
}

首先我们返回一个 Promise，如果 x 为 Promise 的话，需要判断以下几个情况：
```

1. 如果 `x</code`

Event Loop

在前两章节中我们了解了 JS 异步相关的知识。在实践的过程中，你是否遇到过以下场景：为什么 `settimeout` 会比 `Promise` 后执行，明明代码写在 `Promise` 之前，这其实涉及到了 Event Loop 相关的知识。这一章节我们将详细地了解 Event Loop 相关知识，知道 JS 异步运行代码的原理，并且这一章节也是面试常考知识点。

进程与线程

浏览器面试题：进程与线程区别？两者有什么好处？

相信大家经常会听到 JS 是单线程执行的，但是你是否疑惑过什么是线程？

浏览器面试题：线程肯定是指的一个线程，本质上来说，两个名词都是 CPU 工作时间片的一个描述。

进程描述了 CPU 在运行指令及数据时保存上下文所需的时间，放在应用上来说就代表了一个程序，线程是进程中的更小单位。相当于执行一段指令所需的时间。

把这些概念放到浏览器中来说，当你打开一个 Tab 页面时，其实就是创建了一个进程，一个进程中可以有多个线程，比如渲染线程、JS 引擎线程、HTTP 请求数量等。当你发送一个请求时，其实就是在调用了一个线程。当请求结束后，该线程可能就会被销毁。

上面说明了 JS 引擎线程和渲染线程，大家应该都知道，在 JS 运行的时候可能会阻止 UI 渲染，这说明了两个线程是互斥的。但其实的原因是因为 JS 可以修改 DOM，如果在 JS 执行的时候 UI 挂起的话，就很可能导致不能安全的渲染 UI，这其实也是一种线程的好处，得益于 JS 是单线程运行的，可以达到节省内存，节约上下文切换时间，没有线程的问题的好处。当然前面两点在服务端中更容易体现。对于锁的问题，形象的来说就是当我读取一个数字 15 的时候，同时有两个操作对数字进行了加减，这时候结果就出现了错误。解决这个问题也不难，只需要在读取的时候加锁，直到读取完毕之前都不能进行写入操作。

执行栈

浏览器面试题：什么是执行栈？

可以把执行栈认为是一个存储函数调用的线结构，遵循先进后出的原则。

Call Stack
console.log('bar!')
main()

输出：
@阿里技术社区

大家可以在上面清晰的看到嵌套在 `foo` 函数，`foo` 函数又是在 `bar` 函数中调用的。

当我们使用 `return` 的时候，因为返回值存放的函数是有限制的，一旦存放过多的函数且没有得到释放的话，就会出现爆堆的问题。

@阿里技术社区

所以我们在开发中，大家也可以在报错中找到执行栈的概念。

上一小节我们讲到了什么是执行栈，大家也知道了当我们执行 JS 代码的时候其实就是往执行栈中放入函数，那么遇到同步代码的时候该怎么处理呢？其实当遇到异步代码时，会继续趣并在线队列中等待执行的时侯，会将任务放入 Task 队列中。一旦执行为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为。

The diagram illustrates the Event Loop's execution flow. It features a 'Main Queue' at the top where synchronous code like 'script start' and 'script end' runs sequentially. Below it is the 'Event Loop' which handles asynchronous tasks. The 'Task Queue' contains tasks like 'setInterval', 'settimeout', and 'idle'. Promises are shown moving between the Main Queue and the Task Queue, with 'Promise.resolve()' and 'Promise.reject()' being processed by the Event Loop.

不同的任务会被分配到不同的 Task 队列中，任务源可以分为 微任务 (microtask) 和 宏任务 (macrotask)。在 ES6 规范中，microtask 称为 `jobs`，macrotask 称为 `tasks`。下面来看以下代码的执行顺序：

@阿里技术社区

也就是说，如果 `await` 后面跟着 `Promise` 的话，`async end` 需要等待三个 tick 才能执行到。那么其实这个性能对来说还是略慢的，所以 V8 团队修复了 Node 8 中的一个 Bug。在引擎底层将三次 tick 减少到了二次 tick，但是这样做其实是违反了规范的，当然规范也是可以更改的。这是 V8 团队的一个 PR，目前已被同意这种做法。

所以 Event Loop 执行顺序如下所示：

- 首先执行同步代码，这属于宏任务
- 当执行所有同步代码后，执行栈为空，这时是否有异步代码需要执行
- 执行宏任务
- 当执行所有宏任务后，如有必要会渲染页面
- 然后开始下一轮 Event Loop，执行宏任务中的异步代码，也就是 `setTimeout` 中的回调函数

所以上述代码虽然 `settimeout` 可以在 `Promise` 之前，但是因为 `Promise` 属于微任务而 `settimeout` 属于宏任务，所以会有以上的打印。

微任务包括 `process.nextTick`，`promise`，`MutationObserver`，其中 `process.nextTick` 为 Node 独有。

宏任务包括 `script`，`setTimeout`，`setInterval`，`setImmediate`，`I/O`，`UI rendering`。

这里很多人会有个误区，认为微任务快于宏任务，其实这是错误的。因为宏任务中包括了 `script`，浏览器会先执行一个宏任务，接下来再执行微任务。

Node 中的 Event Loop

浏览器面试题：Node 中的 Event Loop 和浏览器中的有什么区别？`process.nextTick` 执行顺序？

Node 中的 Event Loop 和浏览器中的完全不同，完全不同的东西。

Node 的 Event Loop 分为 6 个阶段，它们会按照顺序依次执行，每当进入某一个阶段的时候，都会从对应的回调队列中取出函数去执行。当队列为空或者执行的回调函数数量到达系统设置的阈值，就会进入下一阶段。

timer

timers 阶段会执行 `setTimeout` 和 `setInterval` 回调，并且是由 poll 阶段控制的。

同样，在 Node 中定时器指定的时间也不是准确时间，只能是居模执行。

I/O

I/O 阶段会处理一些上一轮循环中的数据未执行的 I/O 回调。

idle, prepare

idle, prepare 阶段内部实现，这里就不讲了。

poll

poll 是一个至关重要的阶段，这一阶段中，系统会做两件事情

- 回到 timer 阶段执行回调
- 执行 I/O 回调

并且在进入 I/O 阶段时如果没有设置 `timer` 的话，会发生以下两件事

- 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者达到系统限制
- 如果 poll 队列为空时，会有两件事发生
 - 如果有 `setImmediate` 回调需要执行，poll 阶段会停止进入到 check 阶段执行回调
 - 如果没有 `setImmediate` 回调需要执行，则将回调加入到队列中并立即执行回调

这里同样有一个避坑时间设置防止一直等待下去

当然设定了 `timer` 的话且 poll 队列为空，则会判断是否有 timer 超时，如果有的话会回到 timer 阶段执行回调。

check

check 阶段执行 `setImmediate`。

close callbacks 阶段执行 close 事件。

在以上的內容中，我们了解了 Node 中的 Event Loop 的执行顺序，接下来我们将会通过代码的方式来深入理解这块内容。

首先在有些情况下，定时器的执行顺序其实是随机的。

```
new Promise((resolve, reject) => {
  console.log('script start')
  // Promise.resolve() 语句插入任务队列
  // 之后再执行
  resolve()
}).then(() => {
  console.log('script end')
})
```

也就是说，如果 `script end` 后面跟着 `Promise` 的话，`script end` 需要等待三个 tick 才能执行到。那么其实这个性能对来说还是略慢的，所以 V8 团队修复了 Node 8 中的一个 Bug。在引擎底层将三次 tick 减少到了二次 tick，但是这样做其实是违反了规范的，当然规范也是可以更改的。这是 V8 团队的一个 PR，目前已被同意这种做法。

所以 Event Loop 执行顺序如下所示：

- 首先执行同步代码，这属于宏任务
- 当执行所有同步代码后，执行栈为空，这时是否有异步代码需要执行
- 执行宏任务
- 当执行所有宏任务后，如有必要会渲染页面
- 然后开始下一轮 Event Loop，执行宏任务中的异步代码，也就是 `setTimeout` 中的回调函数

所以上述代码虽然 `settimeout` 可以在 `Promise` 之前，但是因为 `Promise` 属于微任务而 `settimeout` 属于宏任务，所以会有以上的打印。

微任务包括 `process.nextTick`，`promise`，`MutationObserver`，其中 `process.nextTick` 为 Node 独有。

宏任务包括 `script`，`setTimeout`，`setInterval`，`setImmediate`，`I/O`，`UI rendering`。

这里很多人会有个误区，认为微任务快于宏任务，其实这是错误的。因为宏任务中包括了 `script`，浏览器会先执行一个宏任务，接下来再执行微任务。

Node 中的 Event Loop 和浏览器中的完全不同，完全不同的东西。

浏览器面试题：Node 中的 Event Loop 和浏览器中的有什么区别？`process.nextTick` 执行顺序？

Node 中的 Event Loop 和浏览器中的完全不同，完全不同的东西。

Node 的 Event Loop 分为 6 个阶段，它们会按照顺序依次执行，每当进入某一个阶段的时候，都会从对应的回调队列中取出函数去执行。当队列为空或者执行的回调函数数量到达系统设置的阈值，就会进入下一阶段。

timer

timers 阶段会执行 `setTimeout` 和 `setInterval` 回调，并且是由 poll 阶段控制的。

同样，在 Node 中定时器指定的时间也不是准确时间，只能是居模执行。

I/O

I/O 阶段会处理一些上一轮循环中的数据未执行的 I/O 回调。

idle, prepare

idle, prepare 阶段内部实现，这里就不讲了。

poll

poll 是一个至关重要的阶段，这一阶段中，系统会做两件事情

- 回到 timer 阶段不为空，会遍历回调队列并同步执行，直到队列为空或者达到系统限制
- 如果 poll 队列为空时，会有两件事发生
 - 如果有 `setImmediate` 回调需要执行，poll 阶段会停止进入到 check 阶段执行回调
 - 如果没有 `setImmediate` 回调需要执行，则将回调加入到队列中并立即执行回调

这里同样有一个避坑时间设置防止一直等待下去

当然设定了 `timer` 的话且 poll 队列为空，则会判断是否有 timer 超时，如果有的话会回到 timer 阶段执行回调。

check

check 阶段执行 `setImmediate`。

close callbacks 阶段执行 close 事件。

在以上的內容中，我们了解了 Node 中的 Event Loop 的执行顺序，接下来我们将会通过代码的方式来深入理解这块内容。

首先在有些情况下，定时器的执行顺序其实是随机的。

```
new Promise((resolve, reject) => {
  console.log('script start')
  // Promise.resolve() 语句插入任务队列
  // 之后再执行
  resolve()
}).then(() => {
  console.log('script end')
})
```

也就是说，如果 `script end` 后面跟着 `Promise` 的话，`script end` 需要等待三个 tick 才能执行到。那么其实这个性能对来说还是略慢的，所以 V8 团队修复了 Node 8 中的一个 Bug。在引擎底层将三次 tick 减少到了二次 tick，但是这样做其实是违反了规范的，当然规范也是可以更改的。这是 V8 团队的一个 PR，目前已被同意这种做法。

所以 Event Loop 执行顺序如下所示：

- 首先执行同步代码，这属于宏任务
- 当执行所有同步代码后，执行栈为空，这时是否有异步代码需要执行
- 执行宏任务
- 当执行所有宏任务后，如有必要会渲染页面
- 然后开始下一轮 Event Loop，执行宏任务中的异步代码，也就是 `setTimeout` 中的回调函数

所以上述代码虽然 `settimeout` 可以在 `Promise` 之前，但是因为 `Promise` 属于微任务而 `settimeout` 属于宏任务，所以会有以上的打印。

微任务包括 `process.nextTick`，`promise`，`MutationObserver`，其中 `process.nextTick` 为 Node 独有。

宏任务包括 `script`，`setTimeout`，`setInterval`，`setImmediate`，`I/O`，`UI rendering`。

这里很多人会有个误区，认为微任务快于宏任务，其实这是错误的。因为宏任务中包括了 `script`，浏览器会先执行一个宏任务，接下来再执行微任务。

Node 中的 Event Loop 和浏览器中的完全不同，完全不同的东西。

Node 的 Event Loop 分为 6 个阶段，它们会按照顺序依次执行，每当进入某一个阶段的时候，都会从对应的回调队列中取出函数去执行。当队列为空或者执行的回调函数数量到达系统设置的阈值，就会进入下一阶段。

timer

timers 阶段会执行 `setTimeout` 和 `setInterval` 回调，并且是由 poll 阶段控制的。

同样，在 Node 中定时器指定的时间也不是准确时间，只能是居模执行。

I/O

I/O 阶段会处理一些上一轮循环中的数据未执行的 I/O 回调。

idle, prepare

idle, prepare 阶段内部实现，这里就不讲了。

poll

poll 是一个至关重要的阶段，这一阶段中，系统会做两件事情

- 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者达到系统限制
- 如果 poll 队列为空时，会有两件事发生
 - 如果有 `setImmediate` 回调需要执行，poll 阶段会停止进入到 check 阶段执行回调
 - 如果没有 `setImmediate` 回调需要执行，则将回调加入到队列中并立即执行回调

这里同样有一个避坑时间设置防止一直等待下去

当然设定了 `timer` 的话且 poll 队列为空，则会判断是否有 timer 超时，如果有的话会回到 timer 阶段执行回调。

check

check 阶段执行 `setImmediate`。

close callbacks 阶段执行 close 事件。

在以上的內容中，我们了解了 Node 中的 Event Loop 的执行顺序，接下来我们将会通过代码的方式来深入理解这块内容。

首先在有些情况下，定时器的执行顺序其实是随机的。

```
new Promise((resolve, reject) => {
  console.log('script start')
  // Promise.resolve() 语句插入任务队列
  // 之后再执行
  resolve()
}).then(() => {
  console.log('script end')
})
```

也就是说，如果 `script end` 后面跟着 `Promise` 的话，`script end` 需要等待三个 tick 才能执行到。那么其实这个性能对来说还是略慢的，所以 V8 团队修复了 Node 8 中的一个 Bug。在引擎底层将三次 tick 减少到了二次 tick，但是这样做其实是违反了规范的，当然规范也是可以更改的。这是 V8 团队的一个 PR，目前已被同意这种做法。

所以 Event Loop 执行顺序如下所示：

- 首先执行同步代码，这属于宏任务
- 当执行所有同步代码后，执行栈为空，这时是否有异步代码需要执行
- 执行宏任务
- 当执行所有宏任务后，如有必要会渲染页面
- 然后开始下一轮 Event Loop，执行宏任务中的

JS 进阶知识点及常考面试题

在这一章节中，我们将会学习到一些原理相关的知识，不会解释涉及到的知识点的作用及用法，希望大家对于这些内容还都不怎么熟悉，推荐先去学习相关的知识点再来学习原理知识。

手写 call, apply 及 bind 函数

浏览器面试：call, apply 及 bind 前面实现完之后有什么样的区别？

首先从以下几点来考虑如何实现这几个函数：

- 不传入第一个参数，那么上下文默认为 `window`
- 改变了 `this` 指向，让新的对象可以执行该函数，并能接受参数

那么我们先来实现 `call`：

```
function.prototype.myCall = function(context) {  
    if (typeof this != 'function') {  
        throw new TypeError('Error')  
    }  
    context = context || window  
    context._this = this  
    const args = [...arguments].slice(1)  
    const result = context._this(...args)  
    delete context._this  
    return result  
}
```

以下是对实现的分析：

- 首先 `context` 为可选参数，如果不传的话默认上下文为 `window`
- 接下来将 `context` 创建一个 `_this` 属性并将其设置为需要调用的函数
- 因为 `call` 可以传入多个参数作为调用函数的参数，所以需要将参数剥离开来
- 然后将函数并将其对象的参数删除

以上就是实现 `call` 的思路，`apply` 的实现也类似，区别在于对参数的处理。所以就不一一分析思路了。

```
function.prototype.myApply = myCall.bind(this);  
if (typeof this != 'function') {  
    throw new TypeError('Error')  
}  
const context = context || window  
context._this = this  
let result;  
// 先判断 call 有没有  
if (arguments[1]) {  
    result = context._this(...arguments[1])  
} else {  
    result = context._this()  
}  
delete context._this  
return result  
}
```

`bind` 的实现对比其他两个函数略微地复杂了一点，因为 `bind` 需要返回一个函数，需要判断一些边界问题。以下是 `bind` 的实现：

```
function.prototype.myBind = function(context) {  
    if (typeof this != 'function') {  
        throw new TypeError('Error')  
    }  
    const _this = this  
    const args = [...arguments].slice(1)  
    // 去除第一个参数  
    return function(...args) {  
        // 因为第一个参数，我们可以通过 new F()，所以需要判断  
        if (this instanceof F) {  
            return new _this(...args, ...arguments)  
        }  
        return _this.apply(context, args.concat(...arguments))  
    }  
}
```

以下是对实现的分析：

- 前几步跟之前的实现差不多，就不赘述了
- `bind` 返回了一个函数，对于函数来说有两种方式调用，一种是直接调用，一种是通过 `new` 的方式。我们来先说直接调用的方式
- 对于直接调用来说，这里选择了 `apply` 的方式实现，但是对于参数需要注意以下情况：因为 `bind` 可以实现类似这样的代码 `f.bind(obj, 1)(2)`，所以我们需要将两边的参数拼接起来，于是就有了这样的实现 `args.concat(...arguments)`
- 最后来说通过 `new` 的方式，在之前的章节中我们学习过如何判断 `this`，对于 `new` 的情况来说，不会有任何方式改变 `this`，所以对于这种情况我们需要判断 `this`，对于 `new` 的情况来说，不会有任何方式改变 `this`，所以对于这种情况我们需要判断 `this`。

new

浏览器面试：`new` 的作用是什么？通过 `new` 的方式创建对象和通过字面量创建有什么区别？

在调用 `new` 的过程中会发生以上四件事情：

- 新生成了一个对象
- 链接到原型
- 绑定 `this`
- 返回新对象

根据以上几个过程，我们也可以试着来自己实现一个 `new`：

```
function create() {  
    let obj = {}  
    let con = [].shift.call(arguments)  
    obj.__proto__ = con.prototype  
    let result = con.apply(obj, arguments)  
    return result instanceof Object ? result : obj  
}
```

以下是对实现的分析：

- 首先获取类型的原型
- 然后读取对象的原型
- 然后一直循环判断对象的原型是否等于类型的原型，直到对象原型为 `null`，因为原型链最终指向为 `null`

为什么 $0.1 + 0.2 \neq 0.3$ ？

浏览器面试：为什么 $0.1 + 0.2 \neq 0.3$ ？如何解决这个问题？

先说原因，因为 JS 采用 IEEE 754 双精度版本（64位），并且只要采用 IEEE 754 的语法规都有该问题。

我们都应该知道计算机是通过二进制来存储东西的，那么 0.1 在二进制中会表示为

```
// 0.1是十进制  
0.1 = 2^-4 + 1.2857E-16 * 2^30
```

我们可以发现， 0.1 在二进制中是无限循环的一些数字，其实不只是 0.1 ，其实很多十进制小数用二进制表示都是无限循环的，这样其实没什么问题，但是 JS 采用的浮点数标准却会裁掉我们的数字。

IEEE 754 双精度版本（64位）将 64 位分为三段

- 第一位用来表示符号
- 接下来的 11 位用来表示指数
- 其他的位数用来表示有效位，也就是用二进制表示 0.1 中的 `1.2857E-16`

那么这些循环的数字被裁剪了，就会出现精度丢失的问题，也就造成了 0.1 不再是 0.1 了，而是变成了 `0.12857E-16 * 2^30 + 1.2857E-16 * 2^30`

那么同样的， 0.2 在二进制也是无限循环的，被裁剪后也失去了精度变成了 `0.20000000000000001`

所以这两者相加等于 0.3 而不是 0.30000000000000001 。

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
console.log(0.1) // 0.1  
0.1 === 0.1 // true
```

那么同样的， 0.2 在二进制也是无限循环的，被裁剪后也失去了精度变成了 `0.20000000000000001`

所以这两者相加等于 0.3 而不是 `0.30000000000000001。`

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制转换为了十进制，十进制又转换为了字符串，在这个转换的过程中发生了取近似值的过程。所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
0.1 + 0.2 === 0.3 // true  
0.1 === 0.1 // true
```

那么可能你又会有一个疑问，既然 0.1 不是 0.1 ，那为什么

JS 思考题

之前我们通过了七个章节来学习关于 JS 这部分的内容，那么接下来，会以几道思考题的方式来确保大家理解这部分的内容。

这种方式不仅能够加深你对知识点的理解，同时也能帮助你串联起多个碎片知识点。一旦你将多个知识点串联起来的能力，在面试中就不会经常出现一问一答的情况。如果面试官问的每个问题你都能够引申出一些相关联的知识点，那么面试一定会提高对你的评价。

思考题一：分为哪两大类型？都有什么各自的优缺点？你该如何判断正确的类型？

首先这几道题目想必很多人都能够很好的答出来，接下来就给大家一点思路讲出与众不同的东西。

思路引导：

- 对于原始型来说，你可以指出 `null` 和 `number` 存在的一些问题，对于对象类型来说，你可以从垃圾回收的角度指出。
- 对于判断新类型来说，你可以去对比一下 `typeof` 和 `instanceof` 之间的区别，也可以指出 `instanceof` 判断类型也不是完全准确的。

以上就是这道题目的回答思路。当然不是说让大家完全按照这个思路去答题，而是存在一个意识，当回答面试题的时候，尽量去引申出这个知识点的某些坑或者与这个知识点相关联的东西。

思考题二：你理解的原型是什么？

思路引导：

起码要列出原型小节中的总结内容，然后还可以指出一些小点，比如并不是所有函数都有 `prototype` 属性，然后引申原型链的概念，提出如何使用原型实现继承，进而可以引申出 ES6 中的 `class` 实现继承。

思考题三：bind, call 和 apply 各有什么区别？

思路引导：

首先肯定是要说出三者的不同，如果自己实现过其中的函数，可以尝试说出自己的思路，然后可以聊一聊 `this` 的内容，有几种规则判断 `this` 到底是什么，`this` 规则会涉及到 `new`，那么最后可以说下自己对于 `new` 的理解。

思考题四：ES6 中有加粗过什么？

思路引导：

这边可说的实在太多，你可以先讲 JS 是单线程运行的，这里就可以锻炼你理解的浅层和进阶的区别，然后讲到异步性，接下来的内容就是涉及 Eventloop 了，微任务和宏任务的区别，哪些是微任务，哪些又是宏任务，还可以谈及浏览器和 Node 中的 Eventloop 的不同，最后还可以聊一聊 JS 中的垃圾回收。

小结

虽然思考题不多，但是其实每一道思考题背后都可以引申出很多内容，大家接下来看学习的过程也应该是会有一定的意识，你学习的这块内容到底和你现在脑海里的哪一个知识点有关联，同时也欢迎大家总结这些思考题，并且把总结的内容链接放在评论中，我会挑选出不错的文章单独放入一章节给大家参考。

留言

全部评论 (32)

路过科学园学习 | 1年前
第一题的题目到底是什么意思
宍 点赞 回复

西铁师 | 1年前
这种套路的问题很可能成为作祟中的挂不到点子上，很容易拉了一大堆，面试官并没有找到自己想听的部分。
宍 19 回复

江上酒 | 1年前
那你就一问一答吧
宍 点赞 回复

华少_月下光风乍暖 | 前端 | 3年前
宍 1 点赞 回复

zenghenglu | 基础 | 基础 | 3年前
第一题感觉应该是基本类型和引用类型
宍 15 回复

littlebirdfly_ | 2年前
借类型
宍 点赞 回复

最小怪 | 2年前
谁说只有两种类型的，看的我一愣
宍 点赞 回复

查看更多回复 ↴

zy_ch | FE @ 东哥兄弟 | 3年前
666
宍 点赞 回复

jeffacode同学 | 前端开发 | 不好意思... | 3年前
每次都是同一答，原来我的问题在这里
宍 1 点赞 回复

MingYuan | 回复 | 二道恩无罪 | 3年前
谁易的还是你说的比较清楚
“应该说的是 `typeof null`, `0.1+0.2 != 0.3`”
宍 点赞 回复

查看更多回复 ↴

SwiftAcer | 前端开发 | 3年前
每次都是同一答，原来我的问题在这里
宍 1 点赞 回复

Murphy君 | 3年前
比如并不是所有函数都有 `prototype` 属性，这个怎么理解？是所有的构造函数才有 `prototype` 属性？
宍 2 点赞

yck | yck (作者) | 3年前
回去看之前的章节
宍 点赞 回复

function.prototype.bind | 3年前
“回去看之前的章节”
宍 点赞 回复

查看更多回复 ↴

上完同学 | outlook @ polarisdu | 3年前
我还以为小白能问的东西能有多粗，原来也是逗号。。。
宍 1 点赞 回复

老铁面无表情 | 2年前
考试大纲 哈哈哈
宍 点赞 回复

春家老夫 | 2年前
不然为啥叫小白
宍 点赞 回复

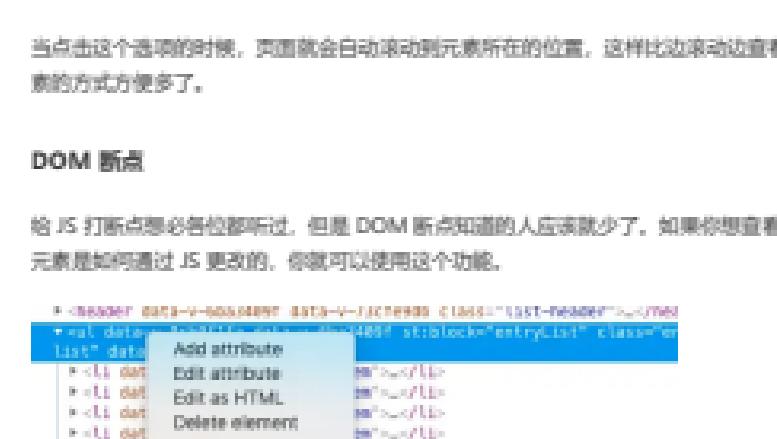
老铁面无表情 | 2年前
考试大纲 哈哈哈
宍 点赞 回复

路过科学园学习 | 1年前
第一题的题目到底是什么意思
宍 点赞 回复

DevTools Tips

这一章节的内容可能和面试没有太大关系，但是如果你能很好地使用 DevTools 的话，它能够很好地帮助你提高生产力和解决一些问题的能力。在这一章节中，我不会去介绍大家经常使用的功能，重点在于让大家学习到一些使用 DevTools 的技巧。

Elements



这个功能肯定也是大家经常用到的，我们可以通过它来可视化所有的 DOM 标签，可以查看任何 DOM 的属性。接下来我们就来学习一下关于这方面的 Tips。

Element 状态

你可能会在开发中遇到这么一个场景：给一个 `a` 标签设置了多种状态下的样式，但是如果手动去改变状态的话就有点麻烦，这时候这个 Tips 就能帮你解决这个问题。



从上图中看到，无论你想看到元素的何种状态下的样式，都只需要勾选相对应的状态就可以了，这是不是比手动更改方便多了？

快速定位 Element

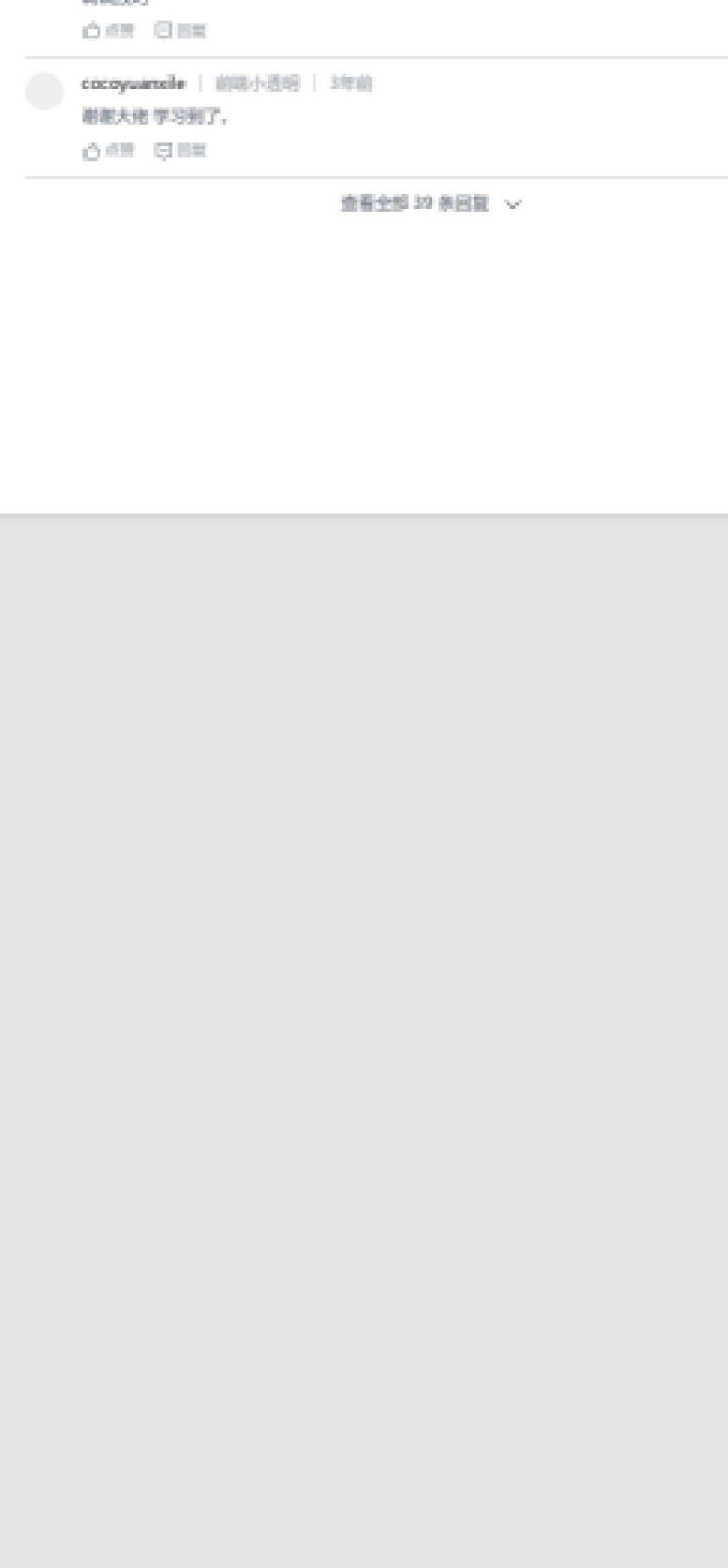
通常都是可以快速定位的，那么如果想查看的元素不在当前窗口的话，你还需要滚动页面才能找到元素，这时候这个 Tip 就能帮到你解决这个问题。



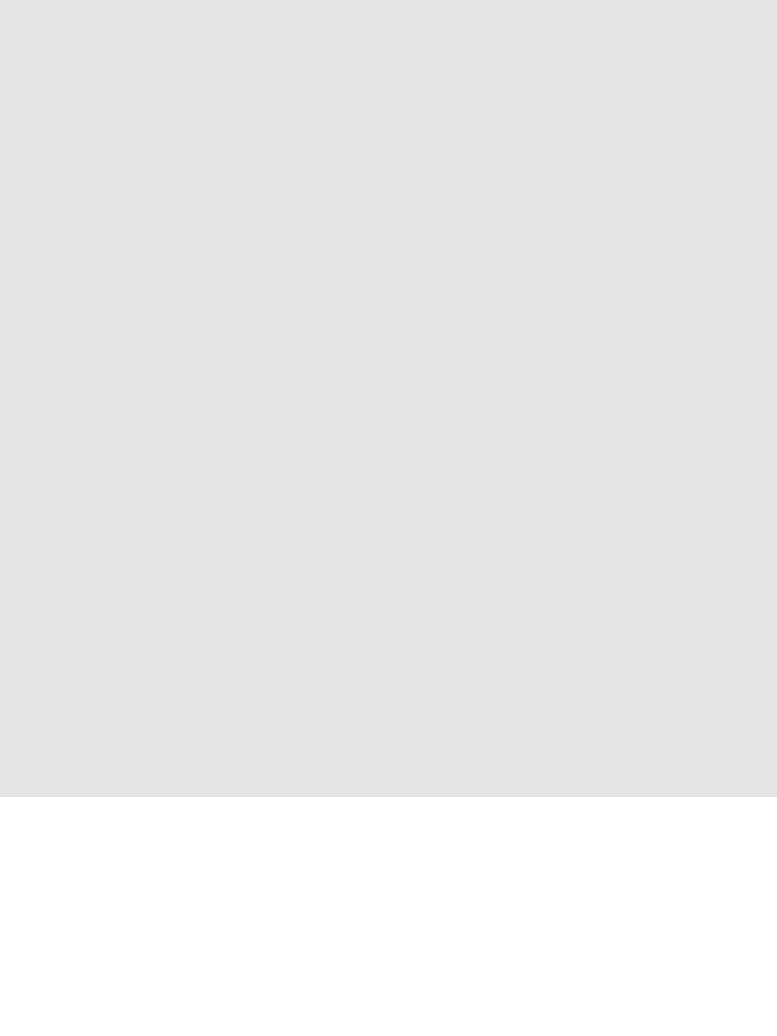
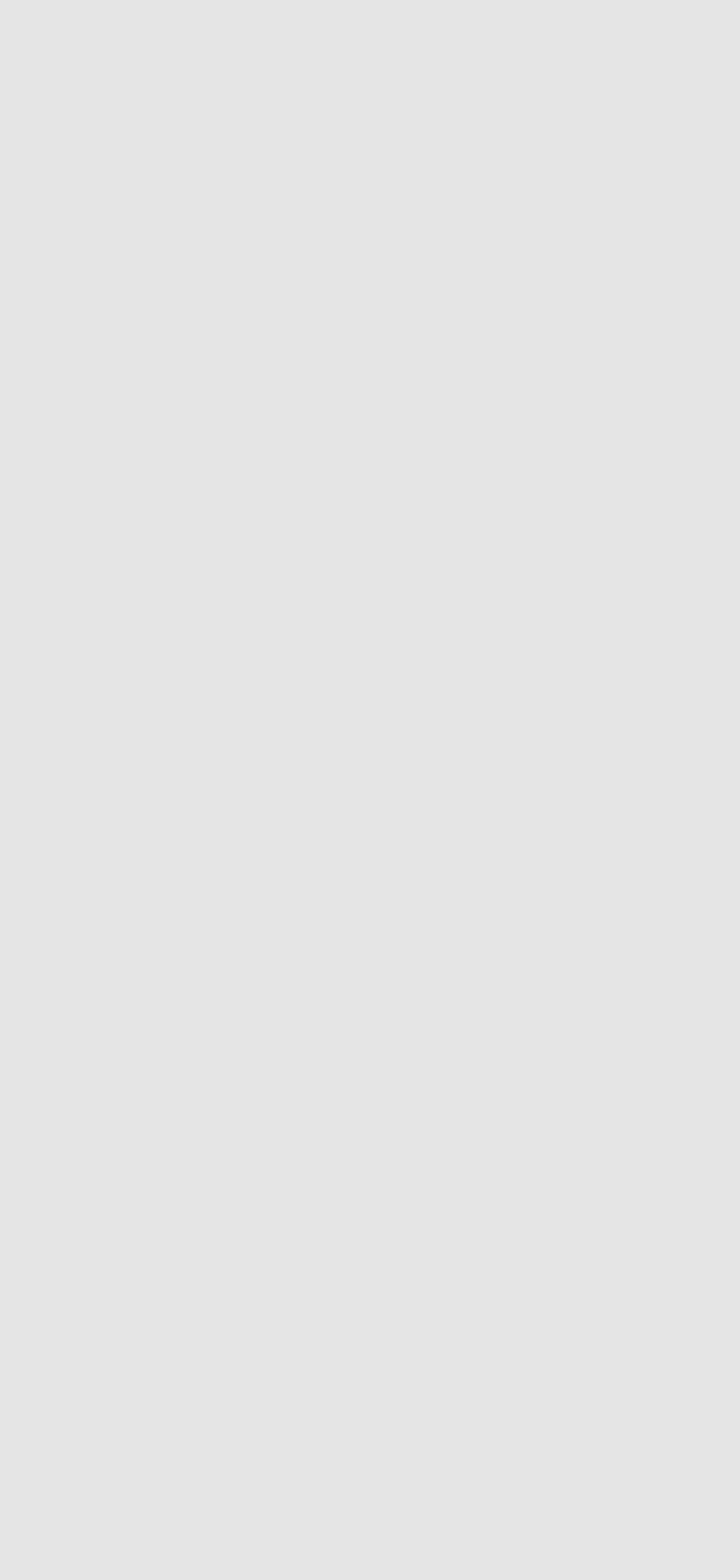
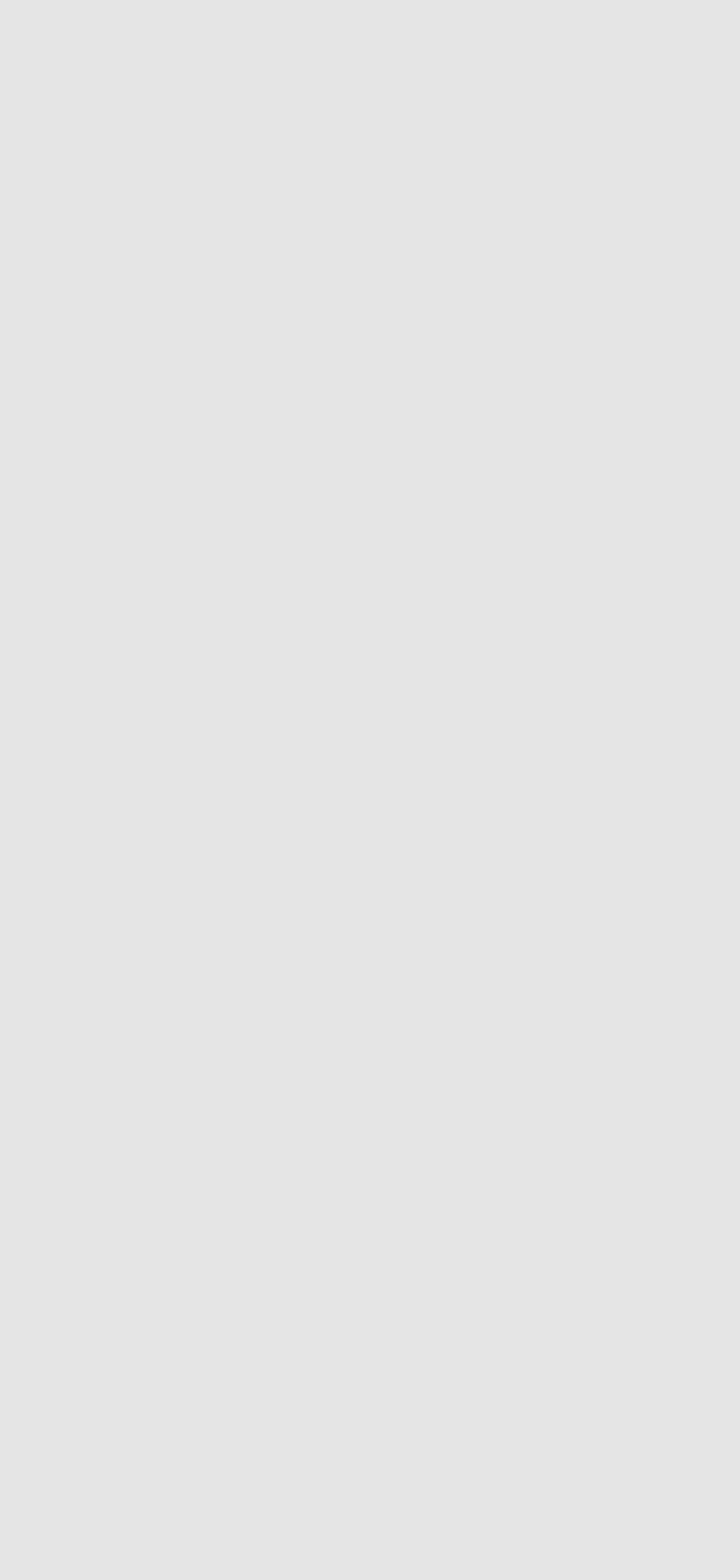
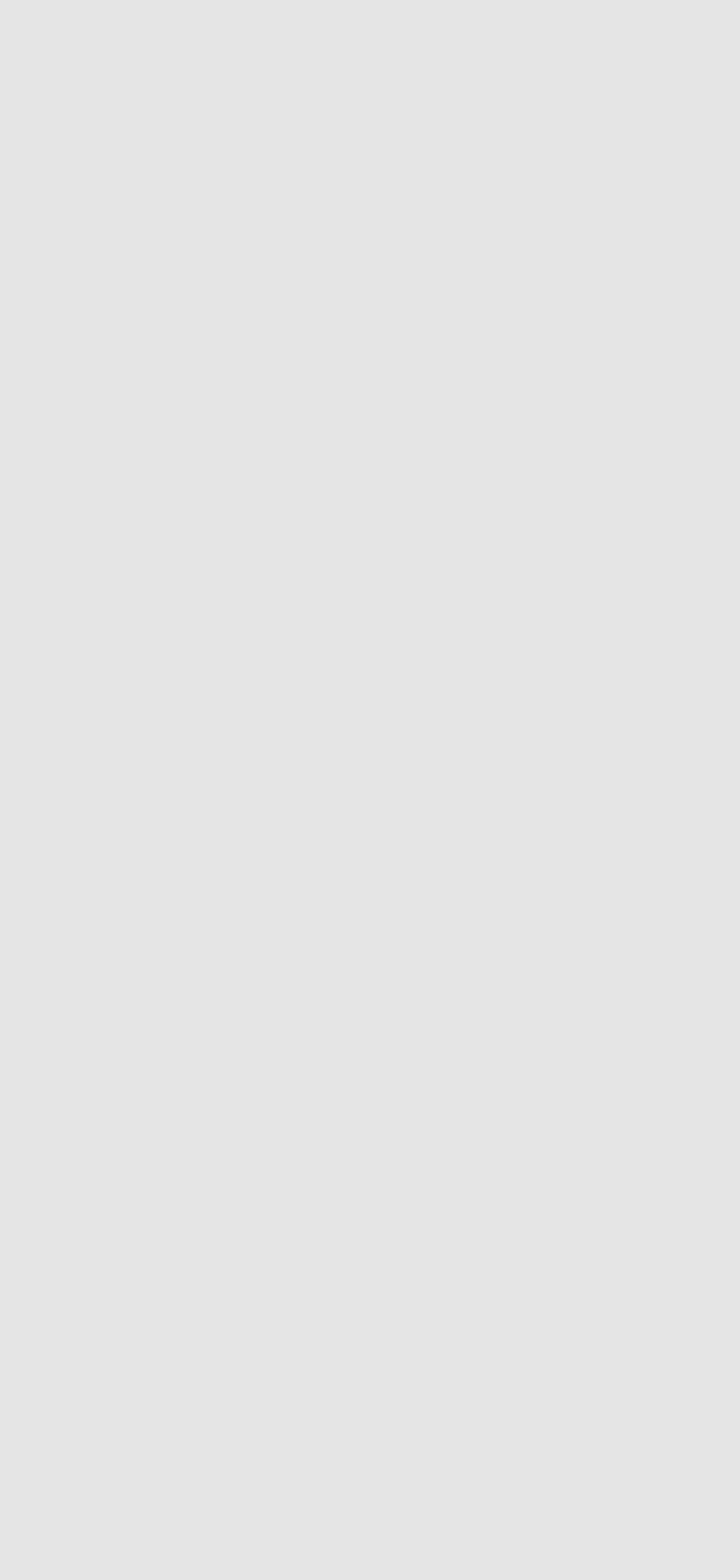
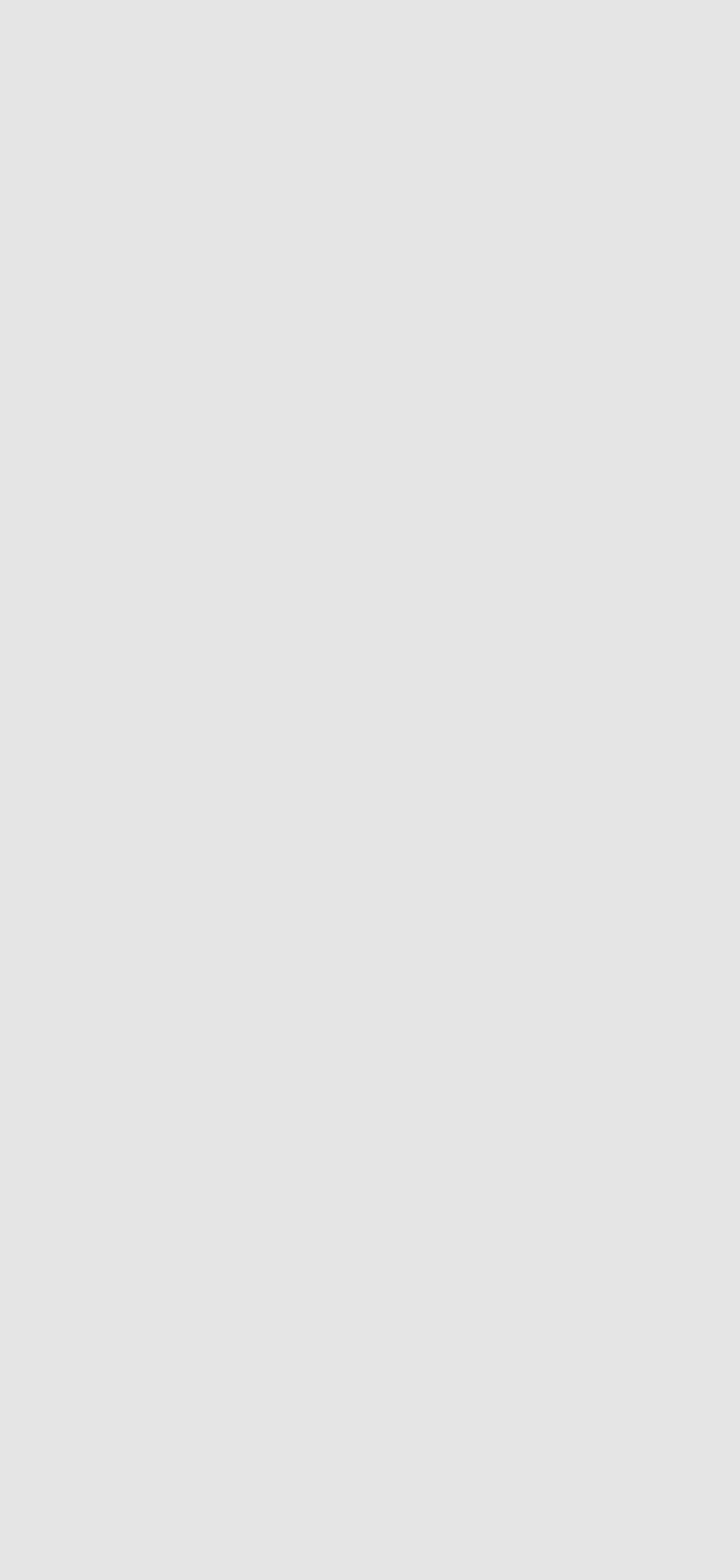
当点击这个选项的时候，页面就会自动滚动到元素所在的位置，这样比滚动边框看是否找到元素的方式方便多了。

DOM 断点

给 JS 打断点想必各位都听过，但是 DOM 断点知道的人应该就少了。如果你想查看一个 DOM 元素是如何通过 JS 更改的，你就可以使用这个功能。



当我们给 `ul` 添加该断点以后，一旦 `ul` 子元素发生了改动，比如说增加了子元素的个数，那么就会自动捕获到对应的 JS 代码。



其实光可以给 DOM 断点，我们还可以给 Ajax 或者 Event Listener 断点。

首先我们先右键断点，然后选择 `Edit Breakpoint...` 选项。

在弹框内输入 `index === 5`，这样断点就会变为橙色，并且只有当符合表达式的情况时断点才会被执行。

小结

虽然这一章的内容并不多，但是涉及到的几个场景都是日常经常会遇到的，希望这一章节的内容会对大家有所帮助。如果大家对于这个章节的内容存在疑问，欢迎在评论区与我互动。

留言

输入评论 (Enter 按键, Ctrl + Enter 提交)

发表评论

全部评论 (39)

zy_ch | FE @ 东南兄弟 | 2年前

牛皮

McHitoTwo | 4年前

加油

浏览器基础知识点及常考面试题

这一章节我们将会来学习浏览器的一些基础知识点。包括：事件机制、跨域、存储相关，这几个知识点也是面试经常会考到的内容。

事件机制

浏览器面试题：事件的触发过程是怎么样的？知道什么是事件代理嘛？

事件触发三阶段

事件触发有三个阶段：

- **window** 事件触发处传播，遇到注册的捕获事件会触发
- 传播到事件触发处时触发注册的事件
- 从事件触发处往 **window** 传播，遇到注册的冒泡事件会触发

事件触发一般来说会按上面的顺序进行，但是也有例外。如果给一个 **body** 中的子节点同时设置冒泡和捕获事件，事件触发会按碰撞的顺序执行。

```
// 以下代码输出的结果是捕获
node.addEventListener(
  'click',
  event => {
    console.log('捕获')
  },
  true
)
```

注册事件

通常我们使用 **addEventListener** 注册事件，该函数的第三个参数可以是布尔值，也可以是对象。对于布尔值 **useCapture** 参数来说，该参数默认值为 **false**，**useCapture** 决定了注册的事件是捕获事件还是冒泡事件。对于对象参数来说，可以使用以下几个属性：

- **capture**: 布尔值，和 **useCapture** 作用一样
- **once**: 表示该回调只会调用一次，调用后会移除监听
- **passive**: 布尔值，表示永远不会调用 **preventDefault**

一般来说，如果我们只希望事件只发生在目标上，这时候可以使用 **stopPropagation** 来阻止事件的进一步传播。通常我们认为 **stopPropagation** 是用来阻止事件冒泡的，其实该函数也可以阻止捕获事件。**stopImmediatePropagation** 同样也能实现阻止事件，但是还能阻止该事件目标执行到的注册事件。

```
node.addEventListener(
  'click',
  event => {
    event.stopPropagation();
    console.log('冒泡')
  },
  false
)

// 从 node 会向上冒泡的看，该函数不会执行
node.addEventListener(
  'click',
  event => {
    console.log('捕获')
  },
  true
)
```

事件代理

如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该挂在父节点上。

```
a1 = document.createElement('a');
a1.setAttribute('href', '#');
document.body.appendChild(a1);
a1.addEventListener('click', (event) => {
  console.log(event.target);
});

a2 = document.createElement('a');
a2.setAttribute('href', '#');
a1.appendChild(a2);
a2.addEventListener('click', (event) => {
  console.log(event.target);
});
```

事件代理的方式相较于直接给目标挂事件来说，有以下优点：

- 节省内存
- 不需要给子节点挂事件

跨域

浏览器面试题：什么是跨域？为什么浏览器限制跨域请求？你有几种方式可以解决跨域问题？了解明了跨域之后，Ajax 请求会失败。

那么会出现什么安全考虑会引入这种机制呢？其实主要是用来防止 CSRF 攻击的。简单点说，CSRF 攻击是利用用户的登录态发起恶意请求。

也就是说，没有同源策略的情况下，A 网站可以任意其他来源的 Ajax 访问到内容。如果你当前 A 网站还在登录态，那么对方就可以通过 Ajax 获得你的任何信息。当然跨域并不能完全阻止 CSRF。

然后我们来考虑一个问题。请求跨域了，那么请求到底发出去没有？请求必然是发出去了，但是浏览器拦截了响应。你可能会疑惑明明通过表单的方式可以发起跨域请求，为什么 Ajax 就不会，因为归根到底，跨域时候用户读取到另一个域名下的内容。Ajax 可以获取响应，浏览器认为这是恶意，所以拦截了响应。但是表单并不会获取新的内容，所以可以发起跨域请求，同时也说明了跨域并不能完全防止 CSRF，因为请求毕竟是发出去了。

接下来我们将学习几种常见的方法来解决跨域的问题。

JSONP

JSONP 的原理很简单，就是利用 **<script>** 标签没有跨域限制的漏洞，通过 **<script>** 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时。

```
<script src="http://www.baidu.com/api/getip/getip.php?cb=jsonp"></script>
<script>
  function jsonp(data) {
    console.log(data)
  }
</script>
```

JSONP 使用简单且兼容性不错。但是只能于 **get** 请求。

在开发中可能会遇到多个 JSONP 请求的回调函数名是相同的，这时候就需要自己封装一个 JSONP。以下以简单实现：

```
function jsonp(url, jsonpCallback, success) {
  let script = document.createElement('script')
  script.src = url
  script.type = 'text/javascript'
  script.jsonpCallback = jsonpCallback
  script.onreadystatechange = function() {
    if (script.readyState === 'loaded' || script.readyState === 'complete') {
      success(script.responseText)
    }
  }
  document.body.appendChild(script)
}
jsonp('http://www.baidu.com/api/getip/getip.php?cb=jsonp', 'callback', function(value) {
  console.log(value)
})
```

CORS

CORS 需要浏览器和后端同时支持，IE 8 和 9 需要通过 **XDomainRequest** 来实现。

浏览器会自动进行 CORS 通信，实现 CORS 通信的关键是后端。只要后端实现了 CORS，就实现了跨域。

服务器端设置 **Access-Control-Allow-Origin** 就可以开启 CORS，该属性表示哪些域名可以访问资源，如果设置为 * 表示所有网站都可以访问资源。

虽然设置 CORS 和前端没什么关系，但是通过这种方式解决跨域问题的话，会在发送请求时出现两种情况，分别为简单请求和复杂请求。

简单请求

以 Ajax 为例，当满足以下条件时，会触发简单请求：

1. 使用下列方法之一：
 - **GET**
 - **HEAD**
 - **POST**
2. **Content-Type** 的值限制于下列三者之一：
 - **text/plain**
 - **multipart/form-data**
 - **application/x-www-form-urlencoded**

请求中的任意 **XMLHttpRequestUpload** 对象都没有注册任何事件监听器；**XMLHttpRequestUpload** 对象可以使用 **XMLHttpRequest.upload** 属性访问。

复杂请求

那么很显然，不符合以上条件的请求就肯定是复杂请求了。

对于复杂请求来说，首先会发起一个预检请求，该请求是 **option** 方法的，通过该请求来知道服务器是否允许跨域请求。

对于预检请求来说，如果你使用过 Node 来设置 CORS 的话，可能会遇到过这么一个坑。

以下以 express 框架举例：

```
app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', '*')
  res.setHeader('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS')
  res.setHeader('Access-Control-Allow-Headers',
    'Origin, X-Requested-With, Content-Type, Accept, Authorization, Access-Control-Allow-Credentials')
})
next()
```

该请求会忽略你的 **Authorization** 字段，没有的话就会报错。

当前端发起一个请求，浏览器中能看到请求头，你会发现就算你设置的 **Content-Type** 是正确的，返回结果永远是报错的，因为预检请求也会进入回溯中，也会触发 **next** 方法，因为预检请求并不包含 **Authorization** 字段，所以服务器会报错。

想解决这个问题很简单，只需要在回溯中过滤 **option** 方法即可。

```
res.statusCode = 200
res.setheader('Content-Length', '0')
res.end()
```

document.domain

该方式只能用于二级域名相同的情况下，比如 **a.test.com** 和 **b.test.com** 适用于该方式。

只需要将页面添加 **document.domain = 'test.com'** 表示二级域名都相同就可以实现跨域。

postMessage

这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息。

```
// 发送消息
window.parent.postMessage('message', 'http://test.com')
// 接收消息
var mc = new MessageChannel()
mc.addEventListener('message', event => {
  var origin = event.origin || event.originalEvent.origin
  if (origin === 'http://test.com') {
    console.log('测试通过')
  }
})
mc.port1.onmessage = event => {
  console.log(event.data)
}
```

打开页面，可以在开发者工具中的 Application 看到 Service Worker 已经启动了！

在 Cache 中也可以发现我们所需的文件已经被缓存。

当我们重新刷新页面可以发现我们缓存的数据是从 Service Worker 中读取的。

小结

以上就是浏览器基础知识点的内容了，如果大家对于这个章节的内容存在疑问，欢迎在评论区与我互动。

输入评论 (Enter按回车, Ctrl+Enter发送)

发表评论

全部评论 (68)

· 麦锐 | 5月前
从插入一点的东西开始就形同虚设了。 ·

· 麦锐 | 5月前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
same-site错了，是跨站。 ·

· 麦锐 | 5月前
限制代码 | 10小时前
前端也是有一套的，归根结底是权限问题，可操作权限，否则一直受限权限。 ·

· 前端 | 5月前
javascript:alert(1) | 10小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

· 前端 | 5月前
前端 | 11小时前
javascript:alert(1) | 11小时前
我理解的跨域也写的不对。 ·

浏览器缓存机制

注意：该知识属于性能优化领域，并且第一章是一个面试题。

缓存可以说是性能优化中简单高效的一种优化方式了。它可以显著减少网络传输所带来的损耗。

对于一个数据请求来说，可以分为发起网络请求、后端处理、浏览器响应三个步骤。浏览器缓存可以帮助我们在第一和第三步实现优化性能。比如说直接使用缓存而不发起请求，或者发起了请求但后端返回的数据和前一版一致，那么就没有必要再将数据回传回来，这样就减少了响应数据。

接下来的内容中我们将通过以下几个部分来探讨浏览器缓存机制：

- 缓存位置
- 缓存策略
- 实际场景应用缓存策略

缓存位置

从缓存位置上来说分为四种，并且各自有优先级，当依次查找缓存时没有命中的话，才会去请求网络。

1. Service Worker
2. Memory Cache
3. Disk Cache
4. Push Cache
5. 网络请求

Service Worker

在上一章节中我们已经介绍了 Service Worker 的内容，这里就不展示相关的代码了。

Service Worker 的缓存与浏览器其他内部的缓存机制不同，它可以让我们在控制缓存哪些文件、如何匹配缓存、如何读取缓存，并且缓存是持久性的。

当 Service Worker 没有命中缓存的时候，我们需要去调用 `fetch` 函数获取数据，也就是说，如果我们没有在 Service Worker 命中缓存的话，会根据缓存 `fetch` 先级去查找数据，但是不管我们是从 `Memory Cache` 中还是从网络请求中获取的数据，浏览器都会显示我们是从 Service Worker 中获取的数据。

Memory Cache

Memory Cache 就是内存中的缓存，读取内存中的数据肯定比磁盘快，但是内存缓存虽然读取高效，可是缓存持续的数据，会随着进程的释放而释放。一旦我们关闭 Tab 页面，内存中的缓存也就被释放了。

当我们访问过页面以后，再次刷新页面，可以发现很多数据都来自于内存缓存。

Name	Status	Type	Initiator	Size	...	Waterfall
bundle.js	200	script	Initial	12.2 KB	-	
manifest.json	200	script	Initial	1 B	-	
vendor.js	200	script	Initial	1 B	-	
app.js	200	script	Initial	1 B	-	
favicon.ico	200	image	Initial	3 B	-	
index.html	200	html	Initial	3.6 KB	-	

那么既然内存缓存这么高效，我们是不是能让数据都存储在内存中呢？

先说结论，这是不可能的。首先计算机中的内存一定比硬盘容量小得多，操作系统需要将物理内存的使用，所以能让我们使用的内存其实不多。内存中其实可以存储大部分的文件，比如 JS、HTML、CSS、图片等等。但是浏览器会把很多文件放进内存这个过程就很繁杂了，我查阅了很多资料却没有一个定论。

当然，我通过一些实践和阅读也得出了一些结论：

- 对于大文件来说，大概率是不存储在内存中的，反之优先
- 当前系统内存使用率高的话，文件优先存进硬盘

Disk Cache

Disk Cache 也就是存储在硬盘中的缓存，读取速度最慢，但是什么都能存储到硬盘中，比之 Memory Cache 在容量和存储时效性上。

在所有浏览器缓存中，Disk Cache 覆盖面基本是最大的，它会根据 HTTP Header 中的字节判断哪些资源需要缓存，哪些资源不需要缓存。哪些资源已经过期需要重新请求，並且即使在跨站点的情况下，如果地址的资源一旦被硬盘缓存下来，就不会再次去请求资源了。

Push Cache

Push Cache 是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被使用。并且缓存时间也很短暂，只在会话（Session）中存在。一旦会话结束就被释放。

Push Cache 在国内能够查到的资料很少，也是因为 HTTP/2 在国内还不普及，但是 HTTP/2 将会是日后的趋势，这里推荐阅读 [HTTP/2 push is tougher than I thought](#) 这篇文章，但是文章是英文的，我翻译一下文章中的几个结论，有能力的同学还是推荐自己阅读。

- 所有的资源都被推送到，但是 Edge 和 Safari 浏览器兼容性不怎么样
- 可以发送 `no-cache` 和 `no-store` 的资源
- 一旦连接被关闭，Push Cache 就会被释放
- 多个页面可以使用相同的 HTTP/2 连接，也就是说能使用同样的缓存
- Push Cache 中的缓存只能被使用一次
- 浏览器可以拒绝接收已经存在的资源推送
- 你可以给其他域名推送资源

网络请求

如果所有缓存都沒有命中的话，那么只能发起请求来获取资源了。

那么为了性能上的考虑，大部分的接口都应该选择好缓存策略，接下来我们就来学习缓存策略这部分的内容。

缓存策略

通常浏览器缓存策略分为两种：强缓存和协商缓存，并且缓存策略都是通过设置 HTTP Header 来实现的。

强缓存

强缓存可以通过设置两种 HTTP Header 实现：`Expires` 和 `Cache-Control`，强缓存表示在缓存期间不需要请求，`status code` 为 200。

Expires

`Expires` 是 HTTP/1 的产物，表示资源会在 `Wed, 22 Oct 2018 08:41:00 GMT` 后过期，需要再次请求。并且 `Expires` 受限于本地时间，如果修改了本地时间，可能会造成缓存失效。

Cache-control

`Cache-control` 出现于 HTTP/1.1，优先级高于 `Expires`，该属性值表示资源会在 30 秒后过期，需要再次请求。

`Cache-control` 可以在请求头或者响应头中设置，并且可以组合使用多种指令。

协商缓存

如果缓存过期了，就需要发起请求验证资源是否更新。协商缓存可以通过设置两种 HTTP Header 实现：`Last-Modified` 和 `ETag`。

当浏览器发起请求验证资源时，如果资源没有做改变，那么服务器就会返回 304 状态码，并且更新浏览器缓存有效期。

Last-Modified 和 If-Modified-Since

`Last-Modified` 表示本地文件最后修改日期，`If-Modified-Since` 会将 `Last-Modified` 的值发给浏览器，询问服务器该资源是否有更新，有更新的话就会将新的资源发送回来，否则返回 304 状态码。

但是 `Last-Modified` 存在一些弊端：

- 如果本地打开缓存文件，即使没有对文件进行修改，但还是会造成长 `Last-Modified` 被修改，这样浏览器不能缓存导致发送相同资源
- 因为 `Last-Modified` 只能以秒为单位，如果在不可感知的时间内修改完成文件，那么服务器会认为资源还是命中了，不会返回正确的资源

因为以上这些弊端，所以在 HTTP/1.1 出现了 `ETag`。

ETag 和 If-None-Match

`ETag` 类似于文件指纹，`If-None-Match` 会将当前 `ETag` 发送给服务器，询问该资源 `ETag` 是否变动，有变动的话就将新的资源发回来，并且 `ETag` 优先级比 `Last-Modified` 高。

以上就是缓存策略的所有内容了，看到这里，不知道你是否存在这样一个疑问：如果什么都设置策略，那么浏览器会怎么处理呢？

对于这种情况，浏览器会采用一个启发式的算法，通常会取响应头中的 `Date` 减去 `Last-Modified` 值的 10% 作为缓存时间。

实际场景应用缓存策略

单纯了解理论而不论乎于实践是没有意义的，接下来我们来通过几个场景学习下如何使用这些理论。

频繁变动的资源

对于频繁变动的资源，首先需要使用 `Cache-Control: no-cache` 使浏览器每次请求服务器，然后配合 `ETag` 或者 `Last-Modified` 来验证资源是否有效，这样的做法虽然不能节省请求数量，但是能显著减少响应数据大小。

代码文件

这里特排除了 HTML 外的代码文件，因为 HTML 文件一般不缓存或者缓存时间很短。

一般来说，现在都会使用工具来打包代码，那么我们就可以对文件名进行哈希处理，只有代码修改后才会生成新的文件名。基于此，我们就可以给代码文件设置缓存有效期一年 `Cache-Control: max-age=31536000`，这样只有当 HTML 文件中引入的文件名发生了改变才会去下载最新的代码文件，否则就一直使用缓存。

小结

在这一章节中我们了解了浏览器的缓存机制，并且列举了几个场景来实践我们学到的理论。希望大家对于这个章节的内容存在疑问，欢迎在评论区与我互动。

留言

输入评论 (Enter执行, Ctrl + Enter发送)

发表评论

全部评论 (70)

梁伟 | web艺术家 | 2年前

原文：如果什么缓存策略都没设置，那么浏览器会怎么处理？对于这种情况，浏览器会采用一个启发式的算法，通常会取响应头中的 `Date` 减去 `Last-Modified` 值的 10% 作为缓存时间。我：前提是响应头中有 `Last-Modified` 吧？如果响应头中连 `Last-Modified` 没有设置呢？请求头里是否带 `If-Modified-Since` 和 `If-None-Match` 来看响应头里有没有 `Last-Modified` 或 `ETag`，服务器首先会产生 `Last-Modified` 或 `ETag` 标记。

白洪 | 前端开发 | 5年前

同学都问到了，讲的太深反而觉得不好，我就粗略看过一些讲的深入的文章，但并不想讲的太深，只是想简单的想法。

白洪 | 前端开发 | 5年前

我也是这样想的，但是一直没敢问，怕被鄙视。

浏览器渲染原理

注意：该章节是一个面试题。

在这一章节中，我们将来学习浏览器渲染原理这部分的知识。你可能会有疑问，我又不是做浏览器研发的，为什么要来学习这个？我们学习浏览器渲染原理更多的是为了了解性能的问题，如果你不了解这部分的知识，你就不知道什么情况下会对性能造成损失，并且渲染原理在面试中答得好，也是一个能与其他人拉开差距的一点。

我们知道执行 JS 有一个 JS 引擎，那么执行渲染也有一个渲染引擎。同样，渲染引擎在不同的浏览器中也都相同的。比如在 Firefox 中叫做 Gecko，在 Chrome 和 Safari 中都是基于 WebKit 开发的。在这一章节中，我们也会主要学习关于 WebKit 的这部分渲染引擎内容。

浏览器接收到 HTML 文件并转换为 DOM 树

当我们打开一个网页时，浏览器都会去请求对应的 HTML 文件。虽然平时我们写代码时都会分为 JS、CSS、HTML 文件，也就是字符串，但是计算机硬件是不理解这些字符串的。所以在网站中传输的内容其实都是 0 和 1 这些字节数据。当浏览器接收到这些字符串以后，它会将这些字节数据转换为字符串，也就是我们写的代码。

字节数据 => 字符串

④博士招生技术社区

当数据转换为字符串以后，浏览器会先将这些字符串通过词法分析转换为标记（token），这一过程在阅读分析中叫做标记化（tokenization）。

字节数据 => 字符串 => Token

④博士招生技术社区

那么什么是标记呢？这其实类似于编译原理这一块的内容了，简单来说，标记还是字符串，是构成代码的最小单位。这一过程会将代码拆分成一块块，然后把这些内容打上标记，便于理解这些最小单位的代码是什么意思。

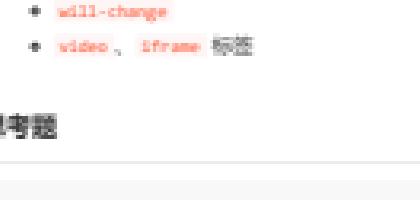
标记为标记内的文本

<a>1

标记为开始一个 a 标签 标记为结束一个 a 标签

④博士招生技术社区

当结束标记以后，这些标记会接着转换为 Node，最后这些 Node 会根据不同 Node 之前的关系构建为一颗 DOM 树。



④博士招生技术社区

以上就是浏览器从网络中接收到 HTML 文件然后一系列的转换过程。

字节数据 => 字符串 => Token => Node => DOM

④博士招生技术社区

当然，在解析 HTML 文件的时候，浏览器还会遇到 CSS 和 JS 文件，这时浏览器会去下载并解析这些文件，接下来就让我们来学习浏览器如何解析 CSS 文件。

将 CSS 文件转换为 CSSOM 树

其实转换 CSS 到 CSSOM 树的过程和上一小节的过程是很类似的。

字节数据 => 字符串 => Token => Node => CSSOM

④博士招生技术社区

在这一过程中，浏览器会确定下每一个节点的样式到底是什么，并且这一过程其实是慢消耗资源的。因为样式你可以自行设置给某个节点，也可以通过继承获得。在这一过程中，浏览器得解析 CSSOM 树，然后确定具体的元素到底是什么样式的。

如果你有点看不懂为什么消耗资源的话，我这里举个例子

```
html
<div>
  <div> <span>span</span> </div>
</div>
<script>
  span {
    color: red;
  }
  div > a > span {
    color: red;
  }
</script>
```

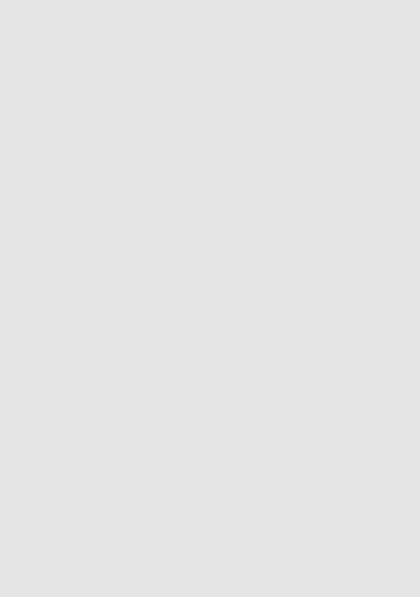
④博士招生技术社区

对于第一种设置样式的方式来说，浏览器只需要找到页面中所有的 span 标签然后设置颜色。但是对于第二种设置样式的方式来来说，浏览器首先需要找到所有的 span 标签，然后找到 span 标签上的 a 标签，然后再去找它的 span 标签，然后把这些内容打上标记，便于理解这些样式的具体实现。这样做的好处就是如果以后我们还想对与之相关的 CSS 选择器，然后对于 HTML 来说也是尽量少的添加无意义标签，保证逻辑清晰。

以上就是浏览器从网络中接收到 CSS 文件然后一系列的转换过程。

生成渲染树

当我们生成 DOM 树和 CSSOM 树以后，就需要把这两棵树组合为渲染树。



④博士招生技术社区

在这一过程中，不是简单的将两者合并就行了，渲染树会包括需要显示的节点和这些节点的样式信息。如果某个节点是 display: none 的，那么就不会在渲染树中显示。

当浏览器生成渲染树以后，会根据渲染树来进行布局（也可以叫做回流），然后调用 GPU 绘制，合成图像，显示在屏幕上。对于这一部分的内容因为过于底层，还涉及到硬件相关知识，这里就不继续展开内容了。

那么通过以上内容，我们已经详细了解了浏览器从接收文件到将内容渲染在屏幕上的这一过程。接下来，我们将会来学习上半部分遗留下来的一些知识点。

为什么操作 DOM 慢

想必大家听说过操作 DOM 性能很慢，但是这其中的原因是什么呢？

因为 DOM 是属于渲染引擎的东西，而 JS 又是 JS 引擎中的东西。当我们通过 JS 操作 DOM 的时候，其实这个操作会涉及到两个线程之间的通信，那么势必会带来一些性能上的损耗。操作 DOM 次数多，也就相当于一直在进行线程之间的通信，而且操作 DOM 可能还会带来重绘的问题，所以也就导致了性能上的问题。

当然面试点会问几个 DOM，如何实现页面不卡顿！

④博士招生技术社区

对于这道题目来说，首先我们肯定不能一次性把几个 DOM 全部插入，这样肯定会造成卡顿，所以解决这个问题的重点应该是如何分批次处理 DOM。大部分人应该可以想到通过 requestAnimationFrame 的方式去逐帧的插入 DOM。其实还有种方式去解决这个问题：虚操作（virtualization），

这种技术的原理就是只能读取区域内的内容，非可见区域的那就完全不渲染了。当用户在滚动的时候就实时去替换渲染的内容。

从上面中我们可以发现，即使列表很长，但是渲染的 DOM 元素永远只有那么几个，当我们滚动的时候会实时去更新新的 DOM，这个技术就能顺利解决这道经典面试题。如果你想了解更多内容可以了解一下这个 react-virtualized。

什么情况阻塞渲染

首先渲染的前提是生成渲染树。所以 HTML 和 CSS 肯定会阻塞渲染。如果你想渲染的越快，你越应该降低一开始需要渲染的文件大小，并且尽量层级，优化选择器。

然后当浏览器在解析到 script 标签时，会暂停构建 DOM，完成后才会从暂停的地方重新开始。也就是说，如果你想部署渲染会更快，就越不应该在首屏就加载 JS 文件，这也是建议将 script 标签放在 body 标签底部的原因。

当然在当下，并不是说 script 标签必须放在底部，因为你可以给 script 标签添加 defer 或者 async 属性。

当 script 标签加上 defer 属性以后，表示该 JS 文件会并行下载，但是会放到 HTML 解析完成后再顺序执行，所以对于这种情况你可以把 script 标签放在任意位置。

对于没有任何依赖的 JS 文件可以加上 async 属性，表示 JS 文件下载和解析不会阻塞渲染。

重绘 (Repaint) 和回流 (Reflow)

重绘和回流会在我们设置节点样式时频繁出现。同时也会很大程度上影响性能。

- 重绘是当节点需要更改外观时会影响到渲染的，比如改变 color 就叫重绘
- 回流是布局或者几何属性需要改变时称为回流。

回流必定会引发重绘，重绘不一定引发回流。回流所需的成本比重绘高得多，改变父节点里的子节点很可能会影响父节点的一系列回流。

以下几个动作可能会导致性能问题：

- 改变 window 大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型

并且很多人不知道的是，重绘和回流其实和 Eventloop 有关。

- 当 Eventloop 执行完 Microtasks 后，会判断 document 是否需要更新。因为浏览器是 60Hz，所以每次问题的重点应该是如何分批次处理 DOM。大部分人应该可以想到通过 requestAnimationFrame 的方式去逐帧的插入 DOM。其实还有种方式去解决这个问题：虚操作（virtualization），
- 然后判断是否有 resize 或者 scroll 事件，如果有适合去触发事件，所以 resize 和 scroll 事件也是至少 16ms 才会触发一次，并且自带流畅功能。

3. 判断是否触发了 media query

4. 更新动画并且发送事件
5. 判断是否完全操作事件
6. 执行 requestAnimationFrame 回调

7. 执行 IntersectionObserver 回调。该方法用于判断元素是否可见。可以用于懒加载上，但是增加性能会带来很多额外的开销。

以上的动作中可能在一秒内会多次触发，如果在一帧中有空闲时间，就会去执行 requestAnimationFrame 回调。

以上内容来自 [HTML 文档](#)。

既然我们已经知道了重绘和回流会影响性能，那么接下来我们将会来学习如何减少重绘和回流的次数。

减少重绘和回流

使用 transform 替代 top:

```
html
<div class="test"></div>
<style>
  .test {
    position: absolute;
    top: 20px;
    width: 100px;
    height: 20px;
    background: red;
  }
</style>
```

④博士招生技术社区

使用 visibility 替换 display: none，因为前者只会引起重绘，后者会引发回流（改变了布局）。

不要把节点的属性放在一个循环里当成循环里的变量

```
js
for(let i = 0; i < 1000; i++) {
  // 重绘
  document.querySelector('.test').style.top = '20px';
}
```

④博士招生技术社区

不要使用 table 布局，可能很小的一个小改动会涵盖整个 table 的重新布局。

动画实现的速度的选择，动画速度越快，回流次数越多，也可以选择使用 requestAnimationFrame 的方式去解决这个问题。

CSS 选择符从有针对性匹配来说，避免节点设置为全局级选择器，全局级选择器会覆盖所有节点，从而导致回流。

将频繁修改的节点设置为局部级选择器，局部级选择器只会影响该节点的渲染行为影响别的节点。

对于 video 标签来说，浏览器会自动将该节点变为图层。

设置节点为图层的方式有很多，我们可以通过以下几个常用属性可以生成新图层

- will-change
- video, iframe 标签

生成渲染树

当我们生成 DOM 树和 CSSOM 树以后，就需要把这两棵树组合为渲染树。

在这一过程中，不是简单的将两者合并就行了，渲染树会包括需要显示的节点和这些节点的样式信息。如果某个节点是 display: none 的，那么就不会在渲染树中显示。

当浏览器生成渲染树以后，会根据渲染树来进行布局（也可以叫做回流），然后调用 GPU 绘制，合成图像，显示在屏幕上。对于这一部分的内容因为过于底层，还涉及到了硬件相关知识，这里就不继续展开内容了。

那么通过以上内容，我们已经详细了解了浏览器从接收文件到将内容渲染在屏幕上的这一过程。接下来，我们将会来学习上半部分遗留下来的一些知识点。

为什么操作 DOM 慢

想必大家听说过操作 DOM 性能很慢，但是这其中的原因是什么呢？

因为 DOM 是属于渲染引擎的东西，而 JS 又是 JS 引擎中的东西。当我们通过 JS 操作 DOM 的时候，其实这个操作会涉及到两个线程之间的通信，那么势必会带来一些性能上的损耗。操作 DOM 次数多，也就相当于一直在进行线程之间的通信，而且操作 DOM 可能还会带来重绘的问题，所以也就导致了性能上的问题。

当然面试点会问几个 DOM，如何实现页面不卡顿！

④博士招生技术社区

对于这道题目来说，首先我们肯定不能一次性把几个 DOM 全部插入，这样肯定会造成卡顿，所以解决这个问题的重点应该是如何分批次处理 DOM。大部分人应该可以想到通过 requestAnimationFrame 的方式去逐帧的插入 DOM。其实还有种方式去解决这个问题：虚操作（virtualization），

这种技术的原理就是只能读取区域内的内容，非可见区域的那就完全不渲染了。当用户在滚动的时候就实时去替换渲染的内容。

从上面中我们可以发现，即使列表很长，但是渲染的 DOM 元素永远只有那么几个，当我们滚动的时候会实时去更新新的 DOM，这个技术就能顺利解决这道经典面试题。如果你想了解更多内容可以了解一下这个 react-virtualized。

什么情况阻塞渲染

首先渲染的前提是生成渲染树。所以 HTML 和 CSS 肯定会阻塞渲染。如果你想渲染的越快，你越应该降低一开始需要渲染的文件大小，并且尽量层级，优化选择器。

然后当浏览器在解析到 script 标签时，会暂停构建 DOM，完成后才会从暂停的地方重新开始。也就是说，如果你想部署渲染会更快，就越不应该在首屏就加载 JS 文件，这也是建议将 script 标签放在 body 标签底部的原因。

当然在当下，并不是说 script 标签必须放在底部，因为你可以给 script 标签添加 defer 或者 async 属性。

当 script 标签加上 defer 属性以后，表示该 JS 文件会并行下载，但是会放到 HTML 解析完成后再顺序执行，所以对于这种情况你可以把 script 标签放在任意位置。

对于没有任何依赖的 JS 文件可以加上 async 属性，表示 JS 文件下载和解析不会阻塞渲染。

重绘 (Repaint) 和回流 (Reflow)

重绘和回流会在我们设置节点样式时频繁出现。同时也会很大程度上影响性能。

- 重绘是当节点需要更改外观时会影响到渲染的，比如改变 color 就叫重绘
- 回流是布局或者几何属性需要改变时称为回流。

回流必定会引发重绘，重绘不一定引发回流。回流所需的成本比重绘高得多，改变父节点里的子节点很可能会影响父节点的一系列回流。

以下几个动作可能会导致性能问题：

- 改变 window 大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型

并且很多人不知道的是，重绘和回流其实和 Eventloop 有关。

- 当 Eventloop 执行完 Microtasks 后，会判断 document 是否需要更新。因为浏览器是 60Hz，所以每次问题的重点应该是如何分批次处理 DOM。大部分人应该可以想到通过 requestAnimationFrame 的方式去逐帧的插入 DOM。其实还有种方式去解决这个问题：虚操作（virtualization），
- 然后判断是否有 resize 或者 scroll 事件，如果有适合去触发事件，所以 resize 和 scroll 事件也是至少 16ms 才会触发一次，并且自带流畅功能。

3. 判断是否触发了 media query

4. 更新动画并且发送事件
5. 判断是否完全操作事件
6. 执行 requestAnimationFrame 回调

7. 执行 IntersectionObserver 回调。该方法用于判断元素是否可见。可以用于懒加载上，但是增加性能会带来很多额外的开销。

以上的动作中可能在一秒内会多次触发，如果在一帧中有空闲时间，就会去执行 requestAnimationFrame 回调。

以上内容来自 [HTML 文档](#)。

既然我们已经知道了重绘和回流会影响性能，那么接下来我们将会来学习如何减少重绘和回流的次数。

减少重绘和回流

使用 transform 替代 top:

```
html
<div class="test"></div>
<style>
  .test {
    position: absolute;
    top: 20px;
    width: 100px;
    height: 20px;
    background: red;
  }
</style>
```

④博士招生技术社区

使用 visibility 替换 display: none，因为前者只会引起重绘，后者会引发回流（改变了布局）。

不要把节点的属性放在一个循环里当成循环里的变量

安全防范知识点

这一章我们将来学习安全防范这一块的知识点。总的来说安全是很复杂的一个领域，不可能通过一个章节就能学习到这部分的内容。在这一章节中，我们会学习到常见的一些安全问题及如何防范的内容。在当下其实安全问题越来越重要，已经逐渐成为前端开发必备的技能了。

XSS

浏览器试想：什么是 XSS 攻击？如何防御 XSS 攻击？什么是 CSRF？

XSS 简单来说，就是攻击者想尽一切办法将可以执行的代码注入到网页中。

XSS 可以分为多种类型，但是总体上我认为分为两类：持久型和非持久型。

持久型也就是攻击的代码被服务器端写入进数据库中，这种攻击危害性很大，因为如果网站访问量很大的话，就会导致大量正常访问页面的用户都受到攻击。

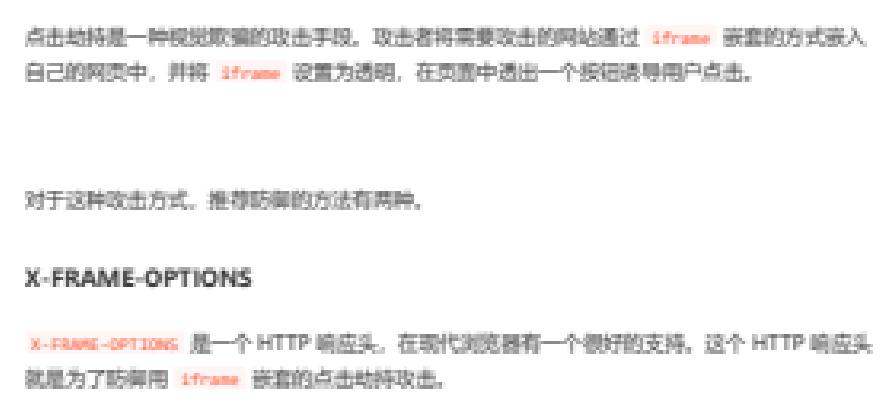
举个例子。对于评论功能来说，就得防范持久型 XSS 攻击，因为我可以在评论中输入以下内容

评论

这种情况下如果前端没有做好防御的话，这段评论就会被存储到数据库中，这样每个打开该页面的用户都会被攻击到。

非持久型相比于前者危害就小的多了，一般通过修改 URL 参数的方式加入攻击代码，诱导用户访问链接从而进行攻击。

举个例子。如果页面需要从 URL 中获取某些参数作为内容的话，不经过过滤就会导致攻击代码被执行

但是对于这种攻击方式来说，如果用户使用 Chrome 这类浏览器的话，浏览器就能够自动帮助用户防御攻击，但是我们不能因此就不防御此类攻击了，因为我不可能确保用户都使用了该类浏览器。

该网页无法正常运作

Chrome 在此网页上检测到了异常代码，为保护您的个人信息（例如密码、电话号码和信用卡信息），Chrome 已将该网页拦截。

请尝试[访问该网站的首页](#)。

ERR_BLOCKED_BY_XSS_AUDITOR

◎安全技术社区

对于 XSS 攻击来说，通常有两种方式可以用来防御。

转义字符

首先，对于用户的输入应该是永远不信任的，最普遍的做法就是转义输入输出的内容，对于引号、尖括号、斜杠进行转义。

通过转义可以将攻击代码 `<script>alert(1)</script>` 变成

但是对于显示文本来说，虽然不能通过上面的办法来转义所有字符。因为这样会把需要的格式也过滤掉。对于这种情况，通常采用白名单过滤的办法，当然也可以通过黑名单过滤，但是考虑到需要过滤的标签和标签属性实在太多，更加推荐使用白名单的方式。

XSS Demo

<script>alert('xss')</script>

CSP

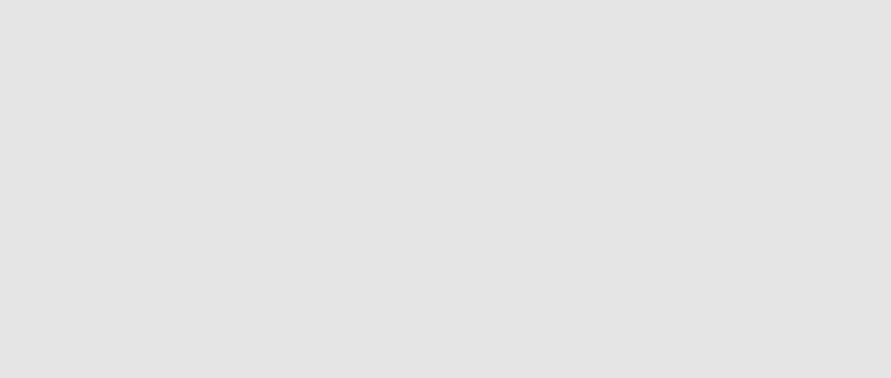
CSP 本质上就是建立白名单，开发者明确告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截是由浏览器自己实现的，我们可以通过这种方式尽量减少 XSS 攻击。

通常可以通过两种方式来开启 CSP：

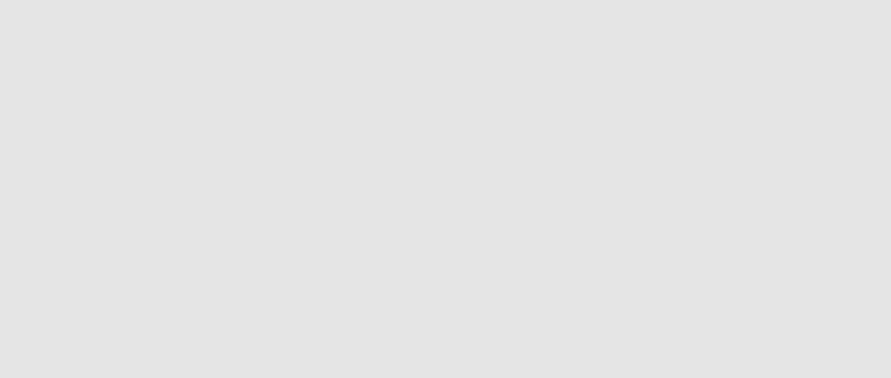
1. 设置 HTTP Header 中的 `Content-Security-Policy`
2. 设置 `meta` 标签的方式 `meta http-equiv="Content-Security-Policy"`

这里以设置 HTTP Header 举例

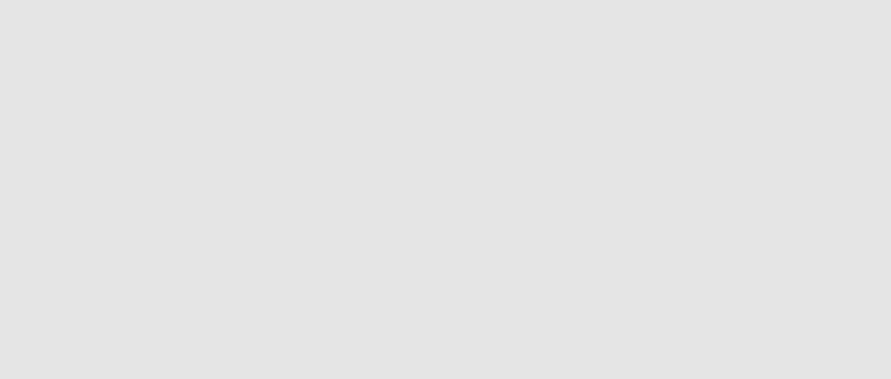
- 只允许加载本站资源



- 只允许加载 HTTPS 协议图片



- 允许加载任何来源框架



当然可以设置的属性远不止这些，你可以通过查阅[文档](#)的方式来学习，这里就不过多赘述其他的属性了。

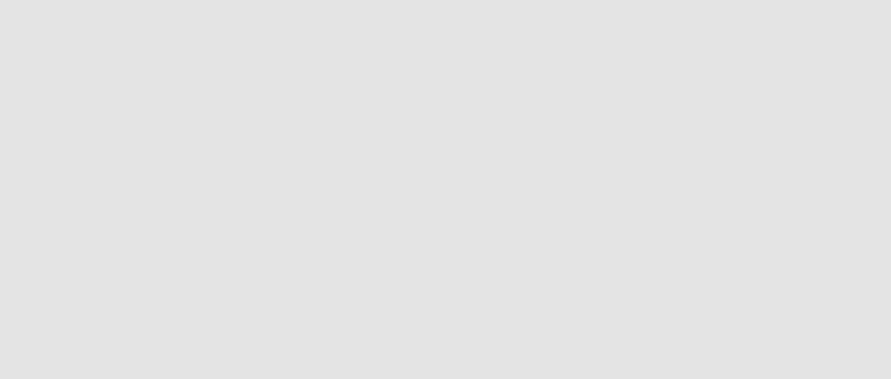
对于这种方式来说，只要开发者配置了正确的规则，那么即使网站存在漏洞，攻击者也不能执行它的攻击代码，并且 CSP 的兼容性也不错。

CSRF

浏览器试想：什么是 CSRF 攻击？如何防御 CSRF 攻击？

CSRF 中文名为什么站请求伪造。原理就是攻击者构造出一个后端请求地址，诱导用户点击或者通过某些逻辑自动发起请求。如果用户是在登录状态下的话，后端就以为是用户在操作，从而进行相应的逻辑。

举个例子。假设网站中有一个通过 `GET` 请求提交用户评论的接口，那么攻击者就可以在钓鱼网站中加入一个图片，图片的地址就是评论接口



那么你是否会觉得使用 `POST` 方式提交请求是不是就没有这个问题了呢？其实并不是，使用这种方式也不是百分百安全的，攻击者同样可以诱导用户进入某个页面，在页面中通过表单提交 `POST` 请求。

如何防御

防范 CSRF 攻击可以遵循以下几种规则：

1. Get 请求不对数据进行修改
2. 不让第三方网站访问到用户 Cookie
3. 阻止第三方网站发起接口
4. 请求时附带验证信息，比如验证码或者 Token

SameSite

可以对 Cookie 设置 `SameSite` 属性，该属性表示 Cookie 不随着跨域请求发送，可以很大程度减少 CSRF 攻击。但是该属性目前并不是所有浏览器都兼容。

验证 Referer

对于需要防范 CSRF 的请求，我们可以通过验证 `Referer` 来判断该请求是否为第三方网站发起的。

Token

服务器下发一个随机 Token，每次发起请求时将 Token 携带上，服务器验证 Token 是否有效。

点击劫持

浏览器试想：什么是点击劫持？如何防御点击劫持？

点击劫持是一种视觉欺骗的攻击手段。攻击者得需要攻击的网站通过 `iframe` 嵌套的方式嵌入自己的网页中，并将 `iframe` 设置为透明，在页面中诱骗用户点击。

对于这种攻击方式，推荐防御的方法有两种。

X-FRAME-OPTIONS

`X-FRAME-OPTIONS` 是一个 HTTP 头部，在现代浏览器有一个很好的支持。这个 HTTP 头部就是用来防御点击劫持的。

该响应头有三个值可选，分别是：

- `DENY`，表示页面不允许通过 `iframe` 的方式展示
- `SAMEORIGIN`，表示页面可以在相同域名下通过 `iframe` 的方式展示
- `ALLOW-FROM`，表示页面可以在指定来源的 `iframe` 中展示

JS 脚本

对于某些旧浏览器来说，并不能支持上面的这种方式，那我们只有通过 JS 的方式来防御点击劫持了。



以上代码的作用就是当通过 `iframe` 的方式加载页面时，攻击者的网页直接不显示所有内容了。

中间人攻击

浏览器试想：什么是中间人攻击？如何防御中间人攻击？

中间人攻击是攻击方同时与服务端和客户端建立了连接，并让对方认为连接是安全的，但是实际上整个通信过程都被攻击者控制了。攻击者不仅能获得双方的通信信息，还能修改通信信息。

通常来说不建议使用公共的 Wi-Fi，因为很可能就会发生中间人攻击的情况。如果你在通信的过程中涉及到了某些敏感信息，就完全暴露给攻击方了。

那么你是否会觉得使用 `POST` 方式提交请求是不是就没有这个问题了呢？其实并不是，使用这种方式也不是百分百安全的，攻击者同样可以诱导用户进入某个页面，在页面中通过表单提交 `POST` 请求。

如何防御

防范 CSRF 攻击可以遵循以下几种规则：

1. Get 请求不对数据进行修改
2. 不让第三方网站访问到用户 Cookie
3. 阻止第三方网站发起接口
4. 请求时附带验证信息，比如验证码或者 Token

SameSite

可以对 Cookie 设置 `SameSite` 属性，该属性表示 Cookie 不随着跨域请求发送，可以很大程度减少 CSRF 攻击。但是该属性目前并不是所有浏览器都兼容。

验证 Referer

对于需要防范 CSRF 的请求，我们可以通过验证 `Referer` 来判断该请求是否为第三方网站发起的。

Token

服务器下发一个随机 Token，每次发起请求时将 Token 携带上，服务器验证 Token 是否有效。

点击劫持

浏览器试想：什么是点击劫持？如何防御点击劫持？

点击劫持是一种视觉欺骗的攻击手段。攻击者得需要攻击的网站通过 `iframe` 嵌套的方式嵌入自己的网页中，并将 `iframe` 设置为透明，在页面中诱骗用户点击。

对于这种攻击方式，推荐防御的方法有两种。

X-FRAME-OPTIONS

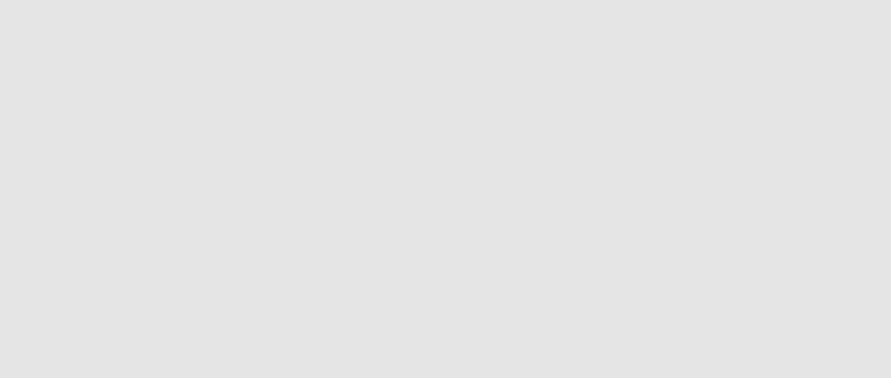
`X-FRAME-OPTIONS` 是一个 HTTP 头部，在现代浏览器有一个很好的支持。这个 HTTP 头部就是用来防御点击劫持的。

该响应头有三个值可选，分别是：

- `DENY`，表示页面不允许通过 `iframe` 的方式展示
- `SAMEORIGIN`，表示页面可以在相同域名下通过 `iframe` 的方式展示
- `ALLOW-FROM`，表示页面可以在指定来源的 `iframe` 中展示

JS 脚本

对于某些旧浏览器来说，并不能支持上面的这种方式，那我们只有通过 JS 的方式来防御点击劫持了。



以上代码的作用就是当通过 `iframe` 的方式加载页面时，攻击者的网页直接不显示所有内容了。

中间人攻击

浏览器试想：什么是中间人攻击？如何防御中间人攻击？

中间人攻击是攻击方同时与服务端和客户端建立了连接，并让对方认为连接是安全的，但是实际上整个通信过程都被攻击者控制了。攻击者不仅能获得双方的通信信息，还能修改通信信息。

通常来说不建议使用公共的 Wi-Fi，因为很可能就会发生中间人攻击的情况。如果你在通信的过程中涉及到了某些敏感信息，就完全暴露给攻击方了。

如何防御

防范 CSRF 攻击可以遵循以下几种规则：

1. Get 请求不对数据进行修改
2. 不让第三方网站访问到用户 Cookie
3. 阻止第三方网站发起接口
4. 请求时附带验证信息，比如验证码或者 Token

SameSite

可以对 Cookie 设置 `SameSite` 属性，该属性表示 Cookie 不随着跨域请求发送，可以很大程度减少 CSRF 攻击。但是该属性目前并不是所有浏览器都兼容。

验证 Referer

对于需要防范 CSRF 的请求，我们可以通过验证 `Referer` 来判断该请求是否为第三方网站发起的。

Token

服务器下发一个随机 Token，每次发起请求时将 Token 携带上，服务器验证 Token 是否有效。

点击劫持

浏览器试想：什么是点击劫持？如何防御点击劫持？

点击劫持是一种视觉欺骗的攻击手段。攻击者得需要攻击的网站通过 `iframe` 嵌套的方式嵌入自己的网页中，并将 `iframe` 设置为透明，在页面中诱骗用户点击。

对于这种攻击方式，推荐防御的方法有两种。

X-FRAME-OPTIONS

`X-FRAME-OPTIONS` 是一个 HTTP 头部，在现代浏览器有一个很好的支持。这个 HTTP 头部就是用来防御点击劫持的。

该响应头有三个值可选，分别是：

- `DENY`，表示页面不允许通过 `iframe` 的方式展示
- `SAMEORIGIN`，表示页面可以在相同域名下通过 `iframe` 的方式展示
- `ALLOW-FROM`，表示页面可以在指定来源的 `iframe` 中展示

JS 脚本

对于某些旧浏览器来说，并不能支持上面的这种方式，那我们只有通过 JS 的方式来防御点击劫持了。

以上代码的作用就是当通过 `iframe` 的方式加载页面时，攻击者的网页直接不显示所有内容了。

中间人攻击

浏览器试想：什么是中间人攻击？如何防御中间人攻击？

中间人攻击是攻击方同时与服务端和客户端建立了连接，并让对方认为连接是安全的，但是实际上整个通信过程都被攻击者控制了。攻击者不仅能获得双方的通信信息，还能修改通信信息。

通常来说不建议使用公共的 Wi-Fi，因为很可能就会发生中间人攻击的情况。如果你在通信的过程中涉及到了某些敏感信息，就完全暴露给攻击方了。

如何防御

防范 CSRF 攻击可以遵循以下几种规则：

1. Get 请求不对数据进行修改
2. 不让第三方网站访问到用户 Cookie
3. 阻止第三方网站发起接口
4. 请求时附带验证信息，比如验证码或者 Token

SameSite

可以对 Cookie 设置 `SameSite` 属性，该属性表示 Cookie 不随着跨域请求发送，可以很大程度减少 CSRF

从 V8 中看 JS 性能优化

注意：该页面属于性能优化领域。

性能问题越来越成为前端火热的话题。因为随着项目的逐步变大，性能问题也逐步体现出来。为了提高用户的体验，减少加载时间，工程师们想尽一切办法去优化细节。

提升之前已经出过一本关于性能的小册子，我在写涉及性能优化的内容时会将地去购买了这本书阅读。目前是为了解决不一样的东西，当性能优化起来还是那几个点。我只能尽可能地写出那本书没有提及的内容，部分内容还是会重叠。当然它超过了十五个章节去介绍性能，肯定会有比我细。有兴趣的可以同时购买还有本「[前端性能优化原理与实践](#)」小册，形成一个互补。

在这些章节中不会提及浏览器、Webpack、网络协议这几块如何优化的内容，因为对应的模块已经讲过了这部分的内容。如果你想学习这几块该如何性能优化的话，可以去对应的章节阅读。

在这一章节中我们将来学习如何让 V8 优化我们的代码。下一章节将会学习性能优化剩余的琐碎点。因为性能优化这个领域所涉及的内容都很碎片化。

在学习如何性能优化之前，我们先来了解下如何测试性能问题，毕竟是先有问题才会去想着该如何改进。

测试性能工具

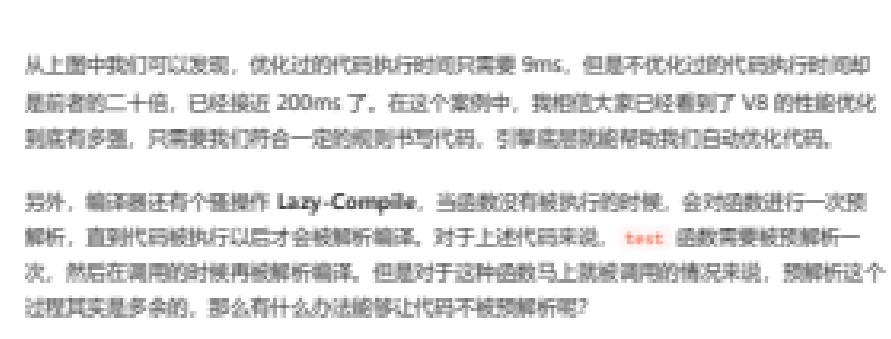
Chrome 已经提供了一个大而全的性能测试工具 [Audits](#)



点我们点击 Audits 后，可以看到如下的界面



在这个界面中，我们可以选择想测试的功能然后点击 Run audits，工具就会自动运行帮助我们测试问题并给出一个完整的报告。



上面是给假金首页测试性能后给出的一个报告，可以看到报告中分为性能、体验、SEO 都给了打分，并且每一个指标都有详细的评估。

Performance

Metrics

First Contentful Paint: 5.490 ms ▲ First Meaningful Paint: 6.870 ms ▲

Speed Index: 11.870 ms ▲ First CPU Idle: 14.980 ms ▲

Time to Interactive: 15.470 ms ▲ Estimated Input Latency: 1.980 ms ▲

[View Trace](#) Values are estimated and may vary.

评估结束后，工具还提供了一些建议便于我们提高这个报告的分数。

Opportunities

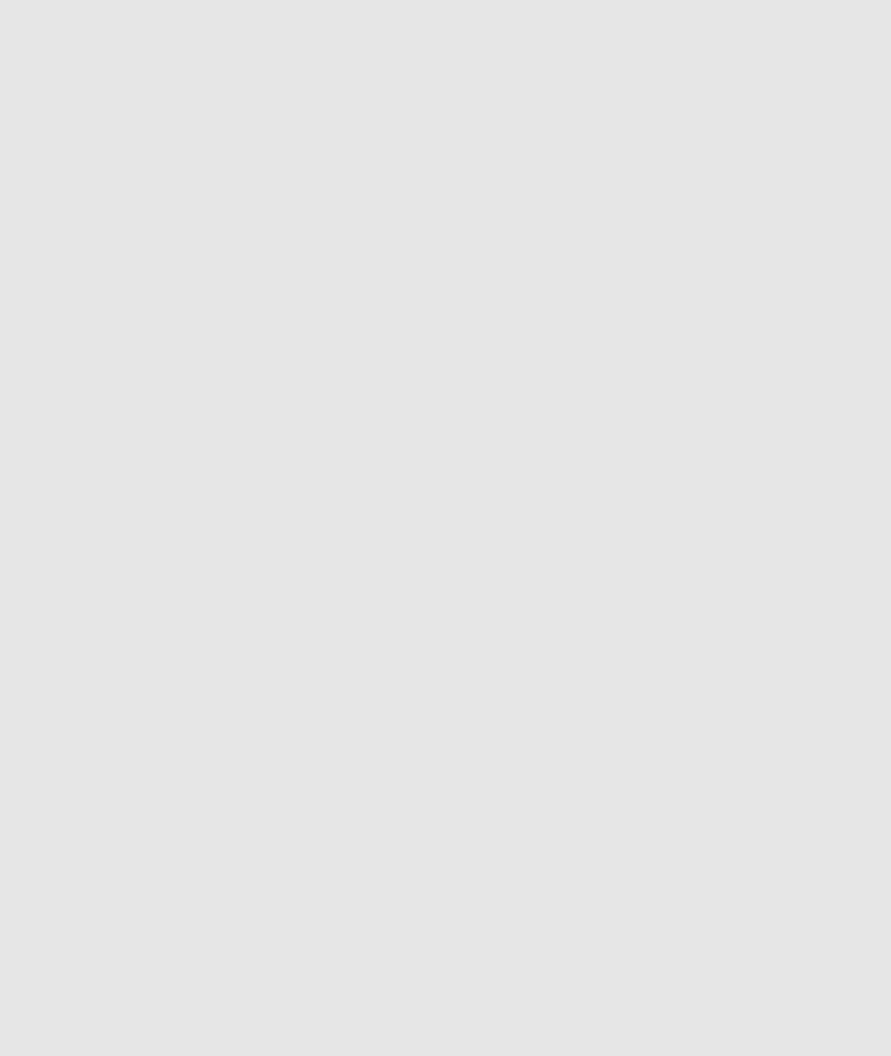
These are opportunities to speed up your application by optimizing the following resources.

Resources to optimize Estimated Savings

1	Defer offscreen images	1.7s
2	Serve images in next-gen formats	1.7s
3	Properly size images	1.38s
4	Defer unused CSS	0.9s
5	Avoid multiple, costly round trips to any origin	0.98s

我们只需要一条条根据建议去优化性能即可。

除了 Audits 工具之外，还有一个 Performance 工具也可以供我们使用。



从上图中我们可以发现，JS 会首先被解析为 AST，解析的过程其实是很慢的。代码越多，解析的过程也就越漫长。这也是我们需要压缩代码的原因之一。另外一种减少解析时间的方式是预解析，会作用于未执行的函数。这个我们在下面再讲。

JS parse time

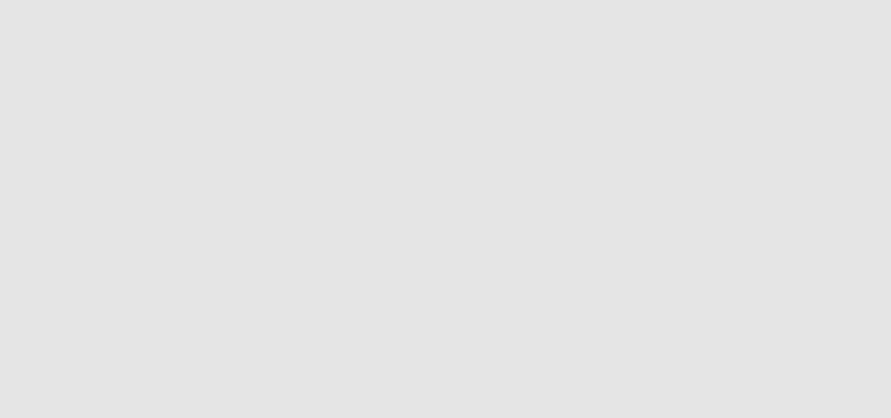
1MB of JS

chrome dev summit 2016 JOIN THE DISCUSSION AT #chromedevsummit

这里需要注意一点，对于函数来说，应该尽可能避免声明嵌套函数（类也是函数），因为这样会面临递归的重新解析。

function test1() {
 // 会将内部的 test2 重新解析
 function test2() {}
}

然后 Ignition 负责将 AST 转化为 Bytecode，TurboFan 负责编译出优化后的 Machine Code，并且 Machine Code 在执行效率上优于 Bytecode。



那么我们就产生了一个疑问，什么情况下代码会编译为 Machine Code？

JS 是编译型还是解释型语言其实并不确定，首先 JS 需要有引擎才能运行起来，无论是浏览器还是在 Node.js 中，这是解释型语言的特性。但是在 V8 引擎下，又引入了 TurboFan 编译器，它会在特定的情况下进行优化，将代码编译成执行效率更高的 Machine Code，当然这个编译器并不是 JS 必须需要的，只是为了提高代码的执行效率，所以总的来说 V8 更偏向于解释型语言。

那么这一小节的内容主要会针对于 Chrome 的 V8 引擎来讲解。

在这一过程中，JS 代码首先会被解析为抽象语法树（AST），然后会通过引擎或者编译器转化为 Bytecode 或者 Machine Code

从上图中我们可以发现，JS 会首先被解析为 AST，解析的过程其实是很慢的。代码越多，解析的过程也就越漫长。这也是我们需要压缩代码的原因之一。另外一种减少解析时间的方式是预解析，会作用于未执行的函数。这个我们在下面再讲。

我们只需要一条条根据建议去优化性能即可。

除了 Audits 工具之外，还有一个 Performance 工具也可以供我们使用。

function test1() {
 return x + x
}

test1();
test1();
test1();
test1();

对于以上代码来说，如果一个函数被多次调用并且参数一直传入 number 类型，那么 V8 就会认为该段代码可以被编译为 Machine Code，因为你固定了类型。不需要再执行很多判断逻辑了。

但是如果一旦我们传入的参数类型改变，那么 Machine Code 就会被 DeOptimized 为 Bytecode，这样就有性能上的一个损耗了。所以如果我们希望代码能被编译为 Machine Code 并且 DeOptimized 的次数减少，就应该尽可能保证传入的类型一致。这也给我们带来了一个思考，这是否也是使用 TypeScript 能够带来的好处之一。

其实很简单，我们只需要给函数加上括号就可以了。

(function test1() {
 return x + x
})()

但是不可能我们为了性能优化，给所有的函数都去套上括号，并且也不是所有函数都需要这样。

我们可以通过 [optimize.js](#) 来测试这个功能，这个库会分析一些函数的使用情况，然后给需要的函数添加括号，当然这个库很久没人维护了，如果需要使用的话，还是需要阅读相关的内容了。

那么我们就可以产生一个疑问，什么情况下代码会编译为 Machine Code？

Machine code

Bytecode

High Level Language

// 会将内部的 test2 重新解析
function test1() {
 function test2() {}
 test1();
}

DeOptimized

Best for humans

Google

Best for machines

@陈士金技术社区

那么我们就产生了一个疑问，什么情况下代码会编译为 Machine Code？

JS 是一门动态类型的语言，并且还有一大堆的规则。简单的加法运算符，内部就需要考虑好几项规则，比如数字相加、字符串相加、对象和字符串相加等等，这样的情况也就势必导致了内部增加很多判断逻辑，降低运行效率。

那么这一小节的内容主要会针对于 Chrome 的 V8 引擎来讲解。

在这一过程中，JS 代码首先会被解析为抽象语法树（AST），然后会通过引擎或者编译器转化为 Bytecode 或者 Machine Code

从上图中我们可以发现，优化过的代码执行时间只需要 9ms，但是不优化过的代码执行时间却是前者的二十倍，已经接近 200ms 了。在这个案例中，我相信大家已经看到了 V8 的性能优化到底有多强，只需要我们符合一定的规则代码，引擎就能帮助我们自动优化代码。

另外，编译器还有一个操作 Lazy-Compile，当函数没有被执行的时候，会对函数进行一次预解析，直到之后被执行才会被重新解析。对于上述代码来说，`test1` 函数会被预解析一次，然后在调用的时候再被解析编译。但是对于这种函数马上就被调用的情况来说，预解析这个过程其实多余的，那么有什么办法能让代码不被预解析呢？

我们只需要一条条根据建议去优化性能即可。

除了 Audits 工具之外，还有一个 Performance 工具也可以供我们使用。

function test1() {
 return x + x
}

test1();
test1();
test1();
test1();

对于以上代码来说，如果一个函数被多次调用并且参数一直传入 number 类型，那么 V8 就会认为该段代码可以被编译为 Machine Code，因为你固定了类型。不需要再执行很多判断逻辑了。

但是如果一旦我们传入的参数类型改变，那么 Machine Code 就会被 DeOptimized 为 Bytecode，这样就有性能上的一个损耗了。所以如果我们希望代码能被编译为 Machine Code 并且 DeOptimized 的次数减少，就应该尽可能保证传入的类型一致。这也给我们带来了一个思考，这是否也是使用 TypeScript 能够带来的好处之一。

其实很简单，我们只需要给函数加上括号就可以了。

(function test1() {
 return x + x
})()

但是不可能我们为了性能优化，给所有的函数都去套上括号，并且也不是所有函数都需要这样。

我们可以通过 [optimize.js](#) 来测试这个功能，这个库会分析一些函数的使用情况，然后给需要的函数添加括号，当然这个库很久没人维护了，如果需要使用的话，还是需要阅读相关的内容了。

那么我们就可以产生一个疑问，什么情况下代码会编译为 Machine Code？

Machine code

Bytecode

High Level Language

// 会将内部的 test2 重新解析
function test1() {
 function test2() {}
 test1();
}

DeOptimized

Best for humans

Google

Best for machines

@陈士金技术社区

那么我们就产生了一个疑问，什么情况下代码会编译为 Machine Code？

JS 是一门动态类型的语言，并且还有一大堆的规则。简单的加法运算符，内部就需要考虑好几项规则，比如数字相加、字符串相加、对象和字符串相加等等，这样的情况也就势必导致了内部增加很多判断逻辑，降低运行效率。

那么这一小节的内容主要会针对于 Chrome 的 V8 引擎来讲解。

在这一过程中，JS 代码首先会被解析为抽象语法树（AST），然后会通过引擎或者编译器转化为 Bytecode 或者 Machine Code

从上图中我们可以发现，优化过的代码执行时间只需要 9ms，但是不优化过的代码执行时间却是前者的二十倍，已经接近 200ms 了。在这个案例中，我相信大家已经看到了 V8 的性能优化到底有多强，只需要我们符合一定的规则代码，引擎就能帮助我们自动优化代码。

另外，编译器还有一个操作 Lazy-Compile，当函数没有被执行的时候，会对函数进行一次预解析，直到之后被执行才会被重新解析。对于上述代码来说，`test1` 函数会被预解析一次，然后在调用的时候再被解析编译。但是对于这种函数马上就被调用的情况来说，预解析这个过程其实多余的，那么有什么办法能让代码不被预解析呢？

我们只需要一条条根据建议去优化性能即可。

除了 Audits 工具之外，还有一个 Performance 工具也可以供我们使用。

function test1() {
 return x + x
}

test1();
test1();
test1();
test1();

对于以上代码来说，如果一个函数被多次调用并且参数一直传入 number 类型，那么 V8 就会认为该段代码可以被编译为 Machine Code，因为你固定了类型。不需要再执行很多判断逻辑了。

但是如果一旦我们传入的参数类型改变，那么 Machine Code 就会被 DeOptimized 为 Bytecode，这样就有性能上的一个损耗了。所以如果我们希望代码能被编译为 Machine Code 并且 DeOptimized 的次数减少，就应该尽可能保证传入的类型一致。这也给我们带来了一个思考，这是否也是使用 TypeScript 能够带来的好处之一。

其实很简单，我们只需要给函数加上括号就可以了。

(function test1() {
 return x + x
})()

但是不可能我们为了性能优化，给所有的函数都去套上括号，并且也不是所有函数都需要这样。

我们可以通过 [optimize.js](#) 来测试这个功能，这个库会分析一些函数的使用情况，然后给需要的函数添加括号，当然这个库很久没人维护了，如果需要使用的话，还是需要阅读相关的内容了。

那么我们就可以产生一个疑问，什么情况下代码会编译为 Machine Code？

Machine code

Bytecode

High Level Language

// 会将内部的 test2 重新解析
function test1() {
 function test2() {}
 test1();
}

DeOptimized

Best for humans

Google

Best for machines

@陈士金技术社区

那么我们就产生了一个疑问，什么情况下代码会编译为 Machine Code？

JS 是一门动态类型的语言，并且还有一大堆的规则。简单的加法运算符，内部就需要考虑好几项规则，比如数字相加、字符串相加、对象和字符串相加等等，这样的情况也就势必导致了内部增加很多判断逻辑，降低运行效率。

那么这一小节的内容主要会针对于 Chrome 的 V8 引擎来讲解。

在这一过程中，JS 代码首先会被解析为抽象语法树（AST），然后会通过引擎或者编译器转化为 Bytecode 或者 Machine Code

从上图中我们可以发现，优化过的代码执行时间只需要 9ms，但是不优化过的代码执行时间却是前者的二十倍，已经接近 200ms 了。在这个案例中，我相信大家已经看到了 V8 的性能优化到底有多强，只需要我们符合一定的规则代码，引擎就能帮助我们自动优化代码。

另外，编译器还有一个操作 Lazy-Compile，当函数没有被执行的时候，会对函数进行一次预解析，直到之后被执行才会被重新解析。对于上述代码来说，`test1` 函数会被预解析一次，然后在调用的时候再被解析编译。但是对于这种函数马上就被调用的情况来说，预解析这个过程其实多余的，那么有什么办法能让代码不被预解析呢？

我们只需要一条条根据建议去优化性能即可。

除了 Audits 工具之外，还有一个 Performance 工具也可以供我们使用。

function test1() {
 return x + x
}

test1();
test1();
test1();
test1();

对于以上代码来说，如果一个函数被多次调用并且参数一直传入 number 类型，那么 V8 就会认为该段代码可以被编译为 Machine Code，因为你固定了类型。不需要再执行很多判断逻辑了。

但是如果一旦我们传入的参数类型改变，那么 Machine Code 就会被 DeOptimized 为 Bytecode，这样就有性能上的一个损耗了。所以如果我们希望代码能被编译为 Machine Code 并且 DeOptimized 的次数减少，就应该尽可能保证传入的类型一致。这也给我们带来了一个思考，这是否也是使用 TypeScript 能够带来的好处之一。

<

图片优化

- 减少像素点
- 减少每个像素点的显示

- 对于能够显示 WebP 格式的浏览器尽量使用 WebP 格式。因为 WebP 格式具有更好的图像数据压缩算法，能带来更小的图片体积。而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好。

DNS

```
if E-  
let h  
return
```

```
    }
}

setInterval(() => {
  console.log(1)
}, 500),
1
)
```

```
// 定义一个方法，用来处理debounce的参数
const debounce = (func, wait = 100) => {
    // 声明一个定时器
    let timer = null
    // 这里相同的函数将取代用户重写调用的原有函数
    // 如果已经設定过定时器了就清空上一次的定时器
    // 开始一个新的定时器，延遲执行用户傳入的方法
    return function(...args) {
        if (timer) clearTimeout(timer)
        timer = setTimeout(() => {
            func.apply(this, args)
        }, wait)
    }
}
```

在开发中，可能会遇到这样的情况：有些资源不需要马上用到，但是希望尽早获取，这时可以使用预加载。

- 预加载可以一定程度上降低载入时间，唯一缺点就是兼容性不好。

执行队
需要在首
用来唤醒

-

函数的拆与函数节流 by 司徒正美

```
// 调用，只执行最后一次  
function clearTimeout(timer){ timer = se-  
Array.prototype.slice.call(arguments);  
throttle(fn, wait) { var timer = n-  
fn.apply(this, Array.prototype.slice
```

回忆 | 前端 | 3年前
黄色感觉的纯跟节流实现功能差不多

△ 点赞 □ 6

MingYuan  | 3年前
jujinim
△ 点赞 □ 回复

墨空  | 3年前
自己写写，看看效果不就懂了

△ 点赞 □ 回复

查看更多回复 ▾

◎ 次數：1
◎ 發文時間：2013-01-10 11:11

 好嗨哟 | web前端开发 | 3年前
按钮点击的时候，加上去之后好像没有作用呢
[点我](#) [回复](#)

 好嗨哟 | 3年前
可以了，发现我是弱鸡了 😅
[点我](#) [回复](#)

 rocky1991 | 前端小菜鸟 @ 创业公司 | 3年前
性能优化的路无尽，极强的压抑性趣，越快越好
[点我](#) [回复](#)

Webpack 性能优化

在这一章节中，我不会浪费篇幅给大家讲如何写配置文件。如果你想学习这方面的内容，那么完全可以去网上学习。在这部分的内容中，我们会聚焦于以下两个知识点，并且每一个知识点都属于高级考点：

- 有哪些方式可以减少 Webpack 的打包时间
- 有哪些方式可以让 Webpack 打出来的包更小

减少 Webpack 打包时间

优化 Loader

对于 Loader 来说，影响打包效率首当其冲必属 Babel 了。因为 Babel 会将代码转为字符串生成 AST，然后对 AST 继续进行转换最后再生成新的代码。项目越大，转换代码越多，效率就越低。当然了，我们是有办法优化的。

首先我们可以优化 Loader 的文件搜索范围

```
module.exports = {
  module: {
    rules: [
      {
        // ts 文件才使用 babel
        test: /\.ts$/,
        loader: 'babel-loader',
        // 只从 src 文件夹下找
        include: [resolve('src')],
        // 不会去用的模块
        exclude: [node_modules]
      }
    ]
  }
}
```

对于 Babel 来说，我们肯定希望它只作用在 JS 代码上的，然后 `node_modules` 中使用的代码都是编译过的，所以我们也完全没有必要再去处理一遍。

当然这样还不够，我们还可以将 Babel 编译过的文件缓存起来，下次只需要编译更改过的代码文件即可。这样可以大幅度加快打包时间

```
loader: 'babel-loader?cacheDirectory=true'
```

HappyPack

受限于 Node 是单线程运行的，所以 Webpack 在打包的过程中也是单线程的，特别是在执行 Loader 的时候，长时间编译的任务很多，这样就会导致等待的情况。

HappyPack 可以将 Loader 的同步执行转换为并行执行，这样就能充分利用系统资源来加快打包效率了

```
module: {
  rules: [
    {
      test: /\.ts$/,
      include: [resolve('src')],
      exclude: [node_modules],
      // id 会指向 happyPack
      loader: 'happyPack-loader?label=happyLabel'
    }
  ]
},
plugins: [
  new happyPack({
    id: 'happyLabel',
    loader: 'babel-loader?cacheDirectory',
    // 并行 4 个线程
    threads: 4
  })
]
```

DllPlugin

DllPlugin 可以将特定的类库提前打包然后引入。这种方式可以极大的减少打包类库的次数，只有当类库更新版本才有需要重新打包，并且也实现了将公共代码抽象成单独文件的优化方案。

接下来我们就来学习如何使用 DllPlugin

```
// 配置其中一个文件
// webpack.dll.conf.js
const path = require('path');
const webpack = require('webpack');
module.exports = {
  entry: {
    // 想打包引用的类库
    vendor: ['react']
  },
  output: {
    path: path.join(__dirname, 'dist'),
    filename: '[name].dll.js',
    library: '[name]-[hash]'
  },
  plugins: [
    new webpack.DllPlugin({
      // name: 会输出的名称
      path: __dirname + '/output/library-one.js',
      // 项目内部的相对路径
      context: __dirname,
      part: path.join(__dirname, 'dist', '[name]-[hash]')
    })
  ]
}
```

然后我们需要执行这个配置文件生成依赖文件，接下来我们需要使用 `DllReferencePlugin` 将依赖文件引入项目中

```
// webpack.conf.js
module.exports = {
  // ... 其他配置
  plugins: [
    new webpack.DllReferencePlugin({
      name: 'vendor',
      // 替换为之前生成的 json 文件
      manifest: require('./dist/vendor-manifest.json')
    })
  ]
}
```

代码压缩

在 Webpack 中，我们一般使用 `uglifyjs` 来压缩代码，但是这个是单线程运行的。为了加快效率，我们可以使用 `webpack-parallel-uglify-plugin` 来并行运行 `uglifyjs`，从而提高效率。

在 Webpack 中，我们就不需要以上这些操作了，只需要将 `mode` 设置为 `production` 就可以默认开启以上功能。代码压缩也是我们必须做的性能优化方案，当然我们不止可以在 JS 代码，还可以压缩 HTML、CSS 代码，并且在压缩 JS 代码的过程中，我们还可以通过配置实现比如删除 `console.log` 这类代码的功能。

一些小的优化点

我们还可以通过一些小的优化点来加快打包速度

- `resolve.extensions`：用来表示文件后缀列表，默认查找顺序是 `['.js', '.json']`，如果你的导入文件没有后缀，那么将多个清空合并了，但是同样也导致了很多并不需要的代码，耗费了很长的JS文件的时间。所以我们尽量减少后缀列表长度。
- `resolve.alias`：可以通过别名的方式映射一个路径，能让 Webpack 更快找到路径
- `module.noParse`：如果你确定一个文件上没有其他依赖，就可以使用该属性让 Webpack 不扫描该文件，这种方式对于大型的类库很有帮助

减少 Webpack 打包后的文件体积

注意：该内容也属于性能优化领域。

按需加载

想必大家在开发 SPA 项目的时候，项目中都会存在十几甚至更多的路由页面。如果我们把这些页面全部放进一个 JS 文件的话，虽然将多个清空合并了，但是同样也导致了很多并不需要的代码，耗费了很长的JS文件的时间。那么为了避免多个清空合并了，但是同样也导致了很多并不需要的代码，耗费了很长的JS文件的时间。

在 Webpack 中，我们就不需要以上这些操作了，只需要将 `mode` 设置为 `production` 就可以默认开启以上功能。代码压缩也是我们必须做的性能优化方案，当然我们不止可以在 JS 代码，还可以压缩 HTML、CSS 代码，并且在压缩 JS 代码的过程中，我们还可以通过配置实现比如删除 `console.log` 这类代码的功能。

一些小的优化点

我们还可以通过一些小的优化点来加快打包速度

- `resolve.extensions`：用来表示文件后缀列表，默认查找顺序是 `['.js', '.json']`，如果你的导入文件没有后缀，那么将多个清空合并了，但是同样也导致了很多并不需要的代码，耗费了很长的JS文件的时间。
- `resolve.alias`：可以通过别名的方式映射一个路径，能让 Webpack 更快找到路径
- `module.noParse`：如果你确定一个文件上没有其他依赖，就可以使用该属性让 Webpack 不扫描该文件，这种方式对于大型的类库很有帮助

Tree Shaking

Tree Shaking 可以实现删除项目中未被引用的代码，比如

```
// test.js
export const a = 1;
export const b = 2;
// index.js
import { a } from './test.js'
```

对于这种情况，我们打包出来的代码会类似这样

```
[/* 什么 */
function (module, exports, require) {
  //...
}
/* 什么 */
function (module, exports, require) {
  //...
}
]
```

但是如果我们使用 Tree Shaking 的话，代码就会尽可能的合并到一个函数中去，也就变成了这样的类似代码

```
[/* 什么 */
function (module, exports, require) {
  //...
}
]
```

这样的打包方式生成的代码量相比之前的少多了。如果在 Webpack4 中你希望开启这个功能，只需要启用 `optimization.concatenateModules` 就可以了。

如果你使用 Webpack 4 的话，开启生产环境就会自动启动这个优化功能。

小结

在这一章节中，我们学习了如何使用 Webpack 去进行性能优化以及如何减少打包时间。

Webpack 的版本更新很快，各个版本之间实现优化的方式可能都会有区别，所以我没有使用过多的代码去展示如何实现一个功能。这一章节的重点是学习到我们可以通过什么方式去优化。具体的代码实现可以查看具体版本对应的代码即可。

留言

输入评论 (Enter键行, Ctrl + Enter键回)

发表评论

全部评论 (18)

一楼 [takim](#) | webpack | 5小时前
优秀的实践

二楼 [yukui](#)

yukui星学习 | 9小时前
是 lodash 吗

三楼 [tom1990algz](#)

tom1990algz | 123 | 2年前
可以通过一个比较完整的 demo 吗

四楼 [受个萝卜](#)

受个萝卜 | 2年前
ParallelUglifyPlugin现在都用这个打包了，支持更好一点

五楼 [小猫猫](#)

小猫猫 | 前端开发 | 2年前
减少文件体积应该再加一个开启zip压缩

六楼 [承受墙右](#)

承受墙右 | FE @ Coding | 2年前
FE @ Coding 应该是直接使用 `optimization.splitChunks`，在 entry 里面明确的提示不要添加 vendors

七楼 [FreePotato](#)

FreePotato | 前端工程师 @ 天上人间 | 3年前
优化全部的分块，有了include是不是就不要include了

八楼 [你是谁在...](#)

你是谁在... | 9小时前
优化结构 include 于前缀

九楼 [zecoding](#)

zecoding | 前端工程师 @ 浏览器切图... | 3年前
可以做一个比较完整的 demo 吗

十楼 [小猫猫](#)

小猫猫 | 前端开发 | 3年前
可以通过一个比较完整的 demo 吗

十一楼 [Tia_Brother](#)

Tia_Brother | 前端开发工程师 @ 捷... | 3年前
可以通过一个比较完整的 demo 吗

十二楼 [不苟言笑](#)

不苟言笑 | 极客工程师 | 3年前
建议阅读 `optimization.concatenateModules`

十三楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

十四楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

十五楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

十六楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

十七楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

十八楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

十九楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十一楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十二楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十三楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十四楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十五楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十六楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十七楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十八楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

二十九楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十一楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十二楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十三楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十四楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十五楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十六楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十七楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十八楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

三十九楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

四十楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

四十一楼 [承受墙右](#)

承受墙右 | 前端工程师 | 3年前
建议阅读 `optimization.concatenateModules`

四十二楼 [承受墙右](#)

实现小型打包工具

原本小册计划中是没有这一章节的，Webpack 工作原理应该是上一章节包含的内容。但是考虑到既然讲到工作原理，必然需要看源码，但是 Webpack 的源码很晦涩，不适合阅读于巴巴读原文又没什么价值，因此在这一章节中，我将会给大家实现一个几十行的迷你打包工具，该工具可以实现以下功能：

- 将 ES6 转换为 ESS
- 支持 JS 文件中 `import` CSS 文件

通过这个工具的实现，大家可以理解到打包工具的原理到底是什么。

实现

因为涉及到 ES6 转 ESS，所以我们首先需要安装一些 Babel 相关的工具：

```
 yarn add babel-loader babel-traverse babel-core babel-preset-env
```

接下来我们将这些工具引入文件中：

```
 const fs = require('fs')
 const path = require('path')
 const babelCore = require('babel-core')
 const traverse = require('babel-traverse').default
 const { transformFromAst } = require('babel-core')
```

首先，我们先来实现如何使用 Babel 转换代码：

```
 function readCode(filePath) {
  // 读取文件内容
  const content = fs.readFileSync(filePath, 'utf-8')
  // 生成 AST
  const ast = babelCore.parse(content, {
    sourceType: 'module'
  })
  // 导出所有文件的依赖关系
  const dependencies = []
  traverse.ast(ast).forEach(({ node }) => {
    dependencies.push(node.source.value)
  })
  // 通过 ast 将代码转为 es6
  const obj = transformFromAst(ast, null, {
    presets: ['env']
  })
  return {
    filePath,
    dependencies,
    code: obj.code
  }
}
```

首先我们读入一个文件路径参数，然后通过 `fs` 将文件中的内容读取出来。

接下来我们通过 `babelCore` 解析代码获取 AST，目的是为了分析代码中是否还引入了别的文件。

通过 `dependencies` 来存储文件中的依赖，然后再将 AST 转换为 ESS 代码。

最后函数返回了一个对象，对象中包含了当前文件路径、当前文件依赖和当前文件转换后的代码。

接下来我们需要实现一个函数，这个函数的功能有以下几点：

- 调用 `readCode` 函数，传入入口文件。
- 分析入口文件的依赖。
- 识别 JS 和 CSS 文件。

```
 function getDependencies(entry) {
  // 读取入口文件
  const entryObject = readCode(entry)
  const dependencies = [entryObject]
  // 遍历所有文件的依赖关系
  entryObject.dependencies.forEach(({ filePath }) => {
    // 读取文件
    const fileContent = fs.readFileSync(filePath, 'utf-8')
    const code = `
      <style>${fileContent}</style>
    `

    dependencies.push({
      filePath,
      relativePath,
      dependencies: [],
      code
    })
  })
  // 如果存在 CSS 文件，那么将相对路径替换成绝对路径
  const child = readCode(filePath)
  child.relativePath = relativePath
  dependencies.push(child)
}

return dependencies
}
```

首先我们读取入口文件，然后创建一个数组，该数组的目的是存储代码中涉及到的所有文件。

接下来我们遍历这个数组，开始把这个数组中只有入口文件，在遍历的过程中，如果入口文件有依赖其他的文件，那么就会被 `push` 到这个数组中。

在遍历的过程中，我们先获得文件对应的目录，然后遍历当前文件的依赖关系。

在遍历当前文件依赖关系的过程中，首先生成依赖文件的绝对路径，然后判断当前文件是 CSS 文件。

如果是 CSS 文件的话，我们就不能用 Babel 去编译了，只需要读取 CSS 文件中的代码，然后生成一个 `style` 标签，将代码插入进标签并放入 `head` 中即可。

如果是 JS 文件的话，我们还需要分析 JS 文件是否还有别的依赖关系。

最后将读取文件后的对象 `push` 进数组中。

现在我们已经读取到了所有的依赖文件，接下来就是实现打包的功能了。

```
 function handle(dependencies, entry) {
  let module = ''
  // 读取文件参数
  // entry.js: function(module, exports, require) { 代码 }
  dependencies.forEach(({ filePath }) => {
    const id = filePath.replace(/\.\w+$/, '')
    module[id] = {
      module: id,
      moduleExports: {},
      moduleRequires: {}
    }
  })
  // 处理 require 语句，目的是为了将模块暴露出来
  const result = ''
  (function(modules) {
    function require(id) {
      const module = modules[id]
      module.exports = module.exports || {}
      module.exports.require = require
      module.exports.id = id
      module.exports.default = module.exports[id]
    }
  })(modules)
  // 生成的内容写入文件中
  fs.writeFileSync(`./bundle.js`, result)
}
```

这段代码要结合着 Babel 转换后的代码来看，这样大家就能理解为什么需要这样写了。

```
// entry.js
var _a = require('./a.js')
var _a$2 = _a._interceptRequireDefault(_a)
function _a_interceptRequireDefault(_a) {
  return obj => obj.___module ? obj : { default: obj }
}
console.log(_a$2.default)
// a.js
object.defineProperty(exports, '__esModule', {
  value: true
})
var a = 1
exports.default = a
```

Babel 将我们 ES6 的模块化代码转换为了 CommonJS (如果你不熟悉 CommonJS 的话，可以阅读这一章中关于 [模块化的知识](#)) 的代码，但是浏览器是不支持 CommonJS 的，所以如果这段代码要在浏览器环境下运行的话，我们需要自己实现支持 CommonJS 相关的代码，这就是 `bundle` 代码做的大部分事情。

接下来我们再来理解下 `bundle` 代码。

首先遍历所有依赖文件，构建出一个函数参数对象。

对参数的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数 `module`, `exports`, `require`。

* `module` 参数对应 CommonJS 中的 `module`。

* `exports` 参数对应 CommonJS 中的 `module.exports`。

* `require` 参数对我们自己创建的 `require` 函数。

接下来就是将 `require` 函数放在 `entry` 对象中，然后用 `require(entry)`，也就是 `require('./entry.js')`，这样就会从函数参数中找到 `./entry.js` 对应的函数并执行，最后将导出的内容通过 `module.exports` 的方式让外部获取到。

最后再将打包出来的内容写入到单独的文件中。

如果你对于上面的实现还有疑惑的话，可以阅读下打包后的部分简化代码。

```
function(module) {
  function require(id) {
    // 读取文件参数
    const moduleExports = module[id] = {}
    // 读取文件相对路径
    moduleExports.id = id
    moduleExports.require = require
    moduleExports.default = moduleExports[id]
  }
  require('./entry.js')
  var _a = module.exports
  var _a$2 = _a._interceptRequireDefault(_a)
  function _a_interceptRequireDefault(_a) {
    return obj => obj.___module ? obj : { default: obj }
  }
  console.log(_a$2.default)
}
// a.js
object.defineProperty(exports, '__esModule', {
  value: true
})
var a = 1
exports.default = a
```

Babel 将我们 ES6 的模块化代码转换为了 CommonJS (如果你不熟悉 CommonJS 的话，可以阅读这一章中关于 [模块化的知识](#)) 的代码，但是浏览器是不支持 CommonJS 的，所以如果这段代码要在浏览器环境下运行的话，我们需要自己实现支持 CommonJS 相关的代码，这就是 `bundle` 代码做的大部分事情。

接下来我们再来理解下 `bundle` 代码。

首先遍历所有依赖文件，构建出一个函数参数对象。

对参数的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数 `module`, `exports`, `require`。

* `module` 参数对应 CommonJS 中的 `module`。

* `exports` 参数对应 CommonJS 中的 `module.exports`。

* `require` 参数对我们自己创建的 `require` 函数。

接下来就是将 `require` 函数放在 `entry` 对象中，然后用 `require(entry)`，也就是 `require('./entry.js')`，这样就会从函数参数中找到 `./entry.js` 对应的函数并执行，最后将导出的内容通过 `module.exports` 的方式让外部获取到。

最后再将打包出来的内容写入到单独的文件中。

如果你对于上面的实现还有疑惑的话，可以阅读下打包后的部分简化代码。

```
function(module) {
  function require(id) {
    // 读取文件参数
    const moduleExports = module[id] = {}
    // 读取文件相对路径
    moduleExports.id = id
    moduleExports.require = require
    moduleExports.default = moduleExports[id]
  }
  require('./entry.js')
  var _a = module.exports
  var _a$2 = _a._interceptRequireDefault(_a)
  function _a_interceptRequireDefault(_a) {
    return obj => obj.___module ? obj : { default: obj }
  }
  console.log(_a$2.default)
}
// a.js
object.defineProperty(exports, '__esModule', {
  value: true
})
var a = 1
exports.default = a
```

Babel 将我们 ES6 的模块化代码转换为了 CommonJS (如果你不熟悉 CommonJS 的话，可以阅读这一章中关于 [模块化的知识](#)) 的代码，但是浏览器是不支持 CommonJS 的，所以如果这段代码要在浏览器环境下运行的话，我们需要自己实现支持 CommonJS 相关的代码，这就是 `bundle` 代码做的大部分事情。

接下来我们再来理解下 `bundle` 代码。

首先遍历所有依赖文件，构建出一个函数参数对象。

对参数的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数 `module`, `exports`, `require`。

* `module` 参数对应 CommonJS 中的 `module`。

* `exports` 参数对应 CommonJS 中的 `module.exports`。

* `require` 参数对我们自己创建的 `require` 函数。

接下来就是将 `require` 函数放在 `entry` 对象中，然后用 `require(entry)`，也就是 `require('./entry.js')`，这样就会从函数参数中找到 `./entry.js` 对应的函数并执行，最后将导出的内容通过 `module.exports` 的方式让外部获取到。

最后再将打包出来的内容写入到单独的文件中。

如果你对于上面的实现还有疑惑的话，可以阅读下打包后的部分简化代码。

```
function(module) {
  function require(id) {
    // 读取文件参数
    const moduleExports = module[id] = {}
    // 读取文件相对路径
    moduleExports.id = id
    moduleExports.require = require
    moduleExports.default = moduleExports[id]
  }
  require('./entry.js')
  var _a = module.exports
  var _a$2 = _a._interceptRequireDefault(_a)
  function _a_interceptRequireDefault(_a) {
    return obj => obj.___module ? obj : { default: obj }
  }
  console.log(_a$2.default)
}
// a.js
object.defineProperty(exports, '__esModule', {
  value: true
})
var a = 1
exports.default = a
```

Babel 将我们 ES6 的模块化代码转换为了 CommonJS (如果你不熟悉 CommonJS 的话，可以阅读这一章中关于 [模块化的知识](#)) 的代码，但是浏览器是不支持 CommonJS 的，所以如果这段代码要在浏览器环境下运行的话，我们需要自己实现支持 CommonJS 相关的代码，这就是 `bundle` 代码做的大部分事情。

接下来我们再来理解下 `bundle` 代码。

首先遍历所有依赖文件，构建出一个函数参数对象。

对参数的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数 `module`, `exports`, `require`。

* `module` 参数对应 CommonJS 中的 `module`。

* `exports` 参数对应 CommonJS 中的 `module.exports`。

* `require` 参数对我们自己创建的 `require` 函数。

接下来就是将 `require` 函数放在 `entry` 对象中，然后用 `require(entry)`，也就是 `require('./entry.js')`，这样就会从函数参数中找到 `./entry.js` 对应的函数并执行，最后将导出的内容通过 `module.exports` 的方式让外部获取到。

最后再将打包出来的内容写入到单独的文件中。

如果你对于上面的实现还有疑惑的话，可以阅读下打包后的部分简化代码。

```
function(module) {
  function require(id) {
    // 读取文件参数
    const moduleExports = module[id] = {}
    // 读取文件相对路径
    moduleExports.id = id
    moduleExports.require = require
    moduleExports.default = moduleExports[id]
  }
  require('./entry.js')
  var _a = module.exports
  var _a$2 = _a._interceptRequireDefault(_a)
  function _a_interceptRequireDefault(_a) {
    return obj => obj.___module ? obj : { default: obj }
  }
  console.log(_a$2.default)
}
// a.js
object.defineProperty(exports, '__esModule', {
  value: true
})
var a = 1
exports.default = a
```

Babel 将我们 ES6 的模块化代码转换为了 CommonJS (如果你不熟悉 CommonJS 的话，可以阅读这一章中关于 [模块化的知识](#)) 的代码，但是浏览器是不支持 CommonJS 的，所以如果这段代码要在浏览器环境下运行的话，我们需要自己实现支持 CommonJS 相关的代码，这就是 `bundle` 代码做的大部分事情。

接下来我们再来理解下 `bundle` 代码。

首先遍历所有依赖文件，构建出一个函数参数对象。

对参数的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数 `module`, `exports`, `require`。

* `module` 参数对应 CommonJS 中的 `module`。

* `exports` 参数对应 CommonJS 中的 `module.exports`。

* `require` 参数对我们自己创建的 `require` 函数。

接下来就是将 `require` 函数放在 `entry` 对象中，然后用 `require(entry)`，也就是 `require('./entry.js')`，这样就会从函数参数中找到 `./entry.js` 对应的函数并执行，最后将导出的内容通过 `module.exports` 的方式让外部获取到。

最后再将打包出来的内容写入到单独的文件中。

如果你对于上面的实现还有疑惑的话，可以阅读下打包后的部分简化代码。

```
function(module) {
  function require(id) {
    // 读取文件参数
    const moduleExports = module[id] = {}
    // 读取文件相对路径
    moduleExports.id = id
    moduleExports.require = require
    moduleExports.default = moduleExports[id]
  }
  require('./entry.js')
  var _a = module.exports
  var _a$2 = _a._interceptRequireDefault(_a)
  function _a_interceptRequireDefault(_a) {
    return obj => obj.___module ? obj : { default: obj }
  }
  console.log(_a$2.default)
}
// a.js
object.defineProperty(exports, '__esModule', {
  value: true
})
var a = 1
exports.default = a
```

Babel 将我们 ES6 的模块化代码转换为了 CommonJS (如果你不熟悉 CommonJS 的话，可以阅读这一章中关于 [模块化的知识](#)) 的代码，但是浏览器是不支持 CommonJS 的，所以如果这段代码要在浏览器环境下运行的话，我们需要自己实现支持 CommonJS 相关的代码，这就是 `bundle` 代码做的大部分事情。

接下来我们再来理解下 `bundle` 代码。

首先遍历所有依赖文件，构建出一个函数参数对象。

对参数的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数 `module`, `exports`, `require`。

* `module` 参数对应 CommonJS 中的 `module`。

* `exports` 参数对应 CommonJS 中的 `module.exports`。

* `require` 参数对我们自己创建的 `require` 函数。

接下来就是将 `require` 函数放在 `entry` 对象中，然后用 `require(entry)`，也就是 `require('./entry.js')`，这样就会从函数参数中找到 `./entry.js` 对应的函数并执行，最后将导出的内容通过 `module.exports` 的方式让外部获取到。

最后再将打包出来的内容写入到单独的文件中。

如果你对于上面的实现还有疑惑的话，可以阅读下打包后的部分简化代码。

```
function(module) {
  function require(id) {
    // 读取文件参数
    const moduleExports = module[id] = {}
    // 读取文件相对路径
    moduleExports.id = id
    moduleExports.require = require
    moduleExports.default = moduleExports[id]
  }
  require('./entry.js')
  var _a = module.exports
  var _a$2 = _a._interceptRequireDefault(_a)
  function _a_interceptRequireDefault(_a) {
    return obj => obj.___module ? obj : { default: obj }
  }
  console.log(_a$2.default)
}
// a.js
object.defineProperty(exports, '__esModule', {
  value: true
})
var a = 1
exports.default = a
```

Babel 将我们 ES6 的模块化代码转换为了 CommonJS (如果你不熟悉 CommonJS 的话，可以阅读这一章中关于 [模块化的知识](#)) 的代码，但是浏览器是不支持 CommonJS 的，所以如果这段代码要在浏览器环境下运行的话，我们需要自己实现支持 CommonJS 相关的代码，这就是 `bundle` 代码做的大部分事情。

接下来我们再来理解下 `bundle` 代码。

首先遍历所有依赖文件，构建出一个函数参数对象。

对参数的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数 `module`, `exports`, `require`。

* `module` 参数对应 CommonJS 中的 `module`。

* `exports` 参数对应 CommonJS 中的 `module.exports`。

* <code

React 和 Vue 两大框架之间的相爱相杀

React 和 Vue 应该是国内当下最火的前端框架，当然 Angular 也是一个不错的框架，但是这个产品因为国内的人很少再加上我对 Angular 也不怎么熟悉，所以框架的章节中不会涉及到 Angular 的内容。

这一章节，我们将会来学习以下几个内容：

- MVVM 是什么
- Virtual DOM 是什么
- 前端路由是如何流转的
- React 和 Vue 之间的区别

MVVM

涉及面试题：什么是 MVVM？比之 MVC 有什么区别？

首先先申明一点，不管是 React 还是 Vue，它们都不是 MVVM 框架，只是有借鉴 MVVM 的思路。文中 Vue 举例也是为了更好地理解 MVVM 的概念。

接下来先说下 View 和 Model：

- View 很简单，就是用户看到的视图
- Model 很简单，一般就是本地数据和数据库中的数据

基本上，我们的产品就是通过接口从数据库中读取数据，然后将数据经过处理映射到用户看到的视图上。当然我们还可以从视图上读取用户的输入，然后将用户的输入写入到数据库中。但是，如何将数据展示到视图上，然后又如何将用户的输入写入到数据库中，不同的人会产生不同的看法，从此也就出现了很多种架构设计。

传统的 MVC 架构通常是使用控制器更新模型，视图从模型中获取数据去渲染。当用户有输入时，会通过控制器去更新模型，并且通知视图进行更新。



但是 MVVM 有一个巨大的缺点就是控制器承担的责任太大了，随着项目愈加复杂，控制器中的代码会越来越臃肿，导致出现不利于维护的情况。

在 MVVM 架构中，引入了 **ViewModel** 的概念，ViewModel 只关心数据和业务的处理，无关 View 如何实现，在这种情况下，View 和 Model 都可以独立出来，任何一方改变了也不一定需要改变另一方，并且可以将一些可复用的逻辑放在一个 ViewModel 中，让多个 View 共享这个 ViewModel。



同样以 Vue 框架来举例，ViewModel 就是组件的实例。View 就是模板，Model 的话在引入 Vuex 的情况下是完全可以和组件分离的。

除了以上三个部分，其实在 MVVM 中还引入了一个微式的 Binder 层，实现了 View 和 ViewModel 的绑定。



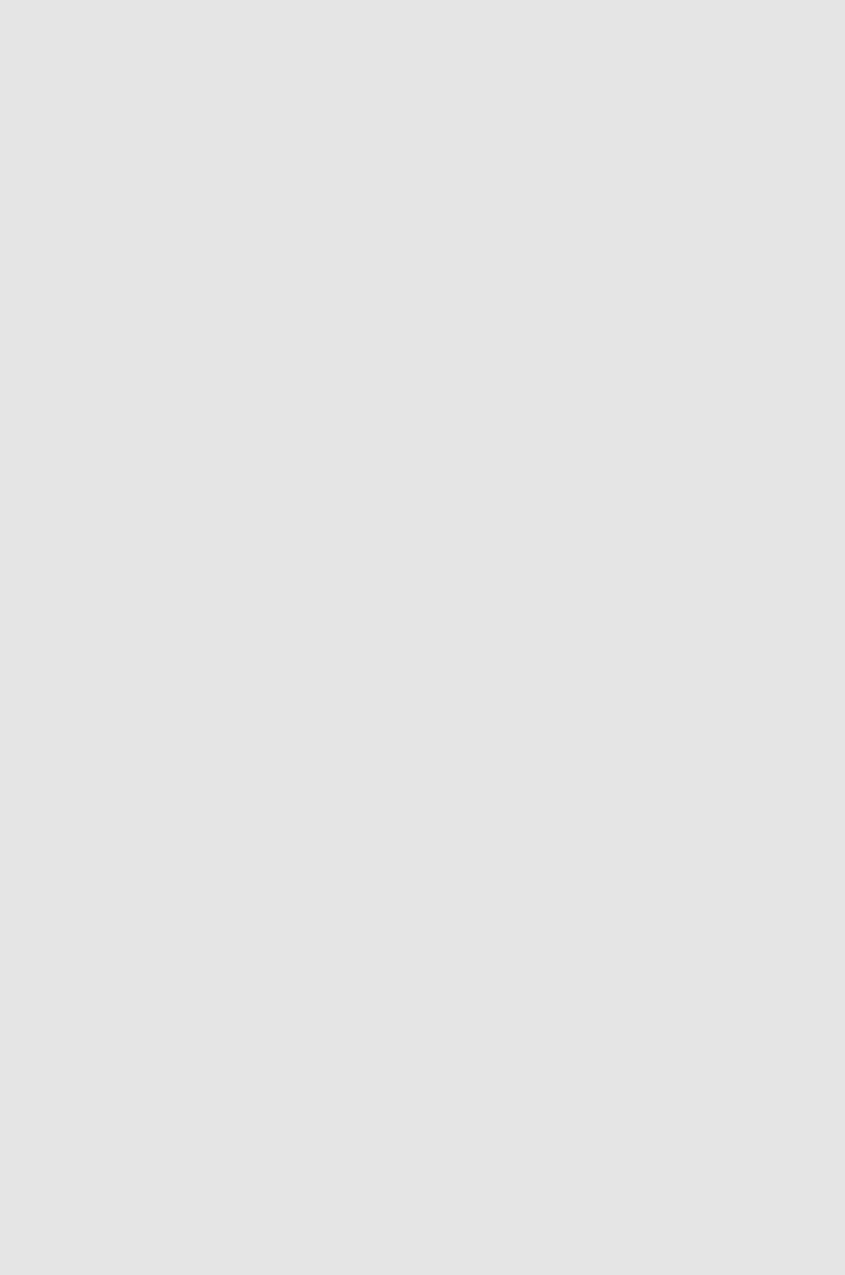
对于 MVVM 来说，其实最重要的并不是通过双向绑定或者其他的方式将 View 与 ViewModel 绑定起来，而是通过 ViewModel 将视图中的状态和用户的行为分离出一个抽象，这才是 MVVM 的精髓。

Virtual DOM

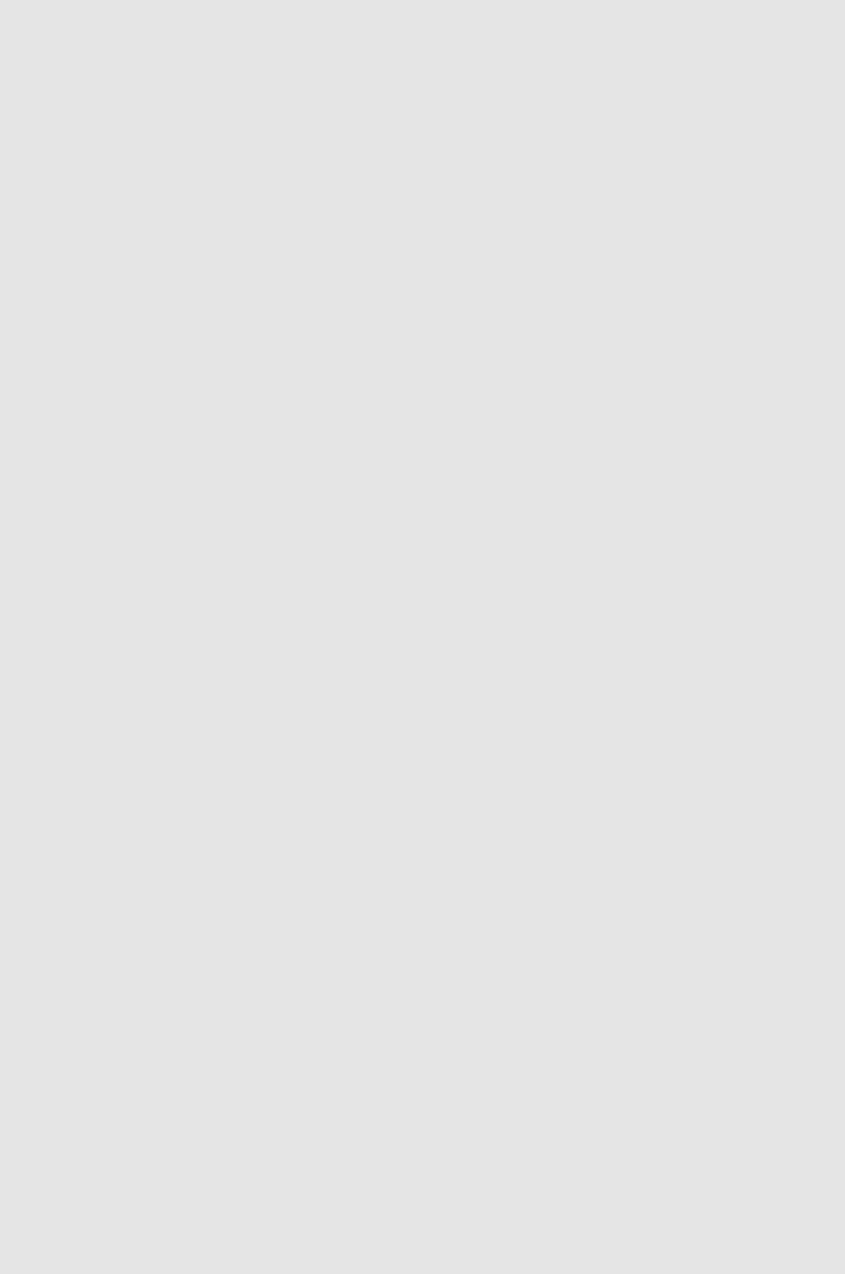
涉及面试题：什么是 Virtual DOM？为什么 Virtual DOM 比原生 DOM 快？

大家都知道操作 DOM 是很慢的，为什么慢的原因已经在「浏览器渲染原理」章节中说过，这里就不再赘述了。

那么相较于 DOM 来说，操作 JS 对象会快很多，并且我们也可以通过 JS 来模拟 DOM



上述代码对应的 DOM 就是



那么既然 DOM 可以通过 JS 对象来模拟，反之也可以通过 JS 对象来渲染出对应的 DOM，当然了，通过 JS 来模拟 DOM 并且渲染对应的 DOM 只是第一步，难点在于如何判断旧两个 JS 对象的最小差异并且实现局部更新 DOM。

首先 DOM 是一个多叉树的结构，如果需要完整的对比两棵树的差异，那么需要的时间复杂度会是 $O(n^2 \cdot 3^h)$ 。这个复杂的算法是不现实的，于是 React 团队优化了算法，实现了 $O(h)$ 的复杂度的差异对比，实现 $O(h)$ 复杂度的关键就是只对比树的节点，而不是跨树对比，这也是考虑到在实际业务中会经常对树的某一个节点进行修改，所以判断差异就分为了两步：

- 首先从上至下，从左到右遍历对象，也就是树的深度遍历，这一步中会将每个节点添加索引，便于最后差异对比。
- 一旦节点有子元素，就去判断子元素是否有不同

在第一步算法中我们需要判断旧节点的 **tagName** 是否相同，如果不相同的话就代表节点被替换了。如果有更改 **tagName** 的话，就需要判断是否有子元素，有的话就进行第二步算法。

在第二步算法中，我们需要判断原本的列表中是否有节点被删除，在新的列表中需要判断是否有新的节点加入，还需要判断节点是否由有变无。

举个例子来说

假设页面中只有一个列表，我们对列表中的元素进行了变更

从上述例子中，我们一眼就可以看出先前的 **ul** 中的第三个 **li** 被移除了，四五替换了位置。

那么在实际的算法中，我们如何去做识别的是哪个节点呢？这就引入了 **key** 这个属性，想必大家在 Vue 或者 React 的列表中都用过这个属性，这个属性是用来给每一个节点打标记的，用于判断是否是同一个节点。

当然在判断以上差异的过程中，我们还需要判断节点的属性是否有变化等。

当我们判断出以上的差异后，就可以把这些差异记录下来。当对比完两棵树以后，就可以通过差异去局部更新 DOM，实现性能的最优化。

另外再来探讨一下为什么 Virtual DOM 比原生 DOM 快这个问题。首先这个问题得分成场景来说。如果纯粹替换所有的 DOM 这样来说，Virtual DOM 的局部更新肯定来的快。但是如果你可以在内部同时替换 DOM，那么 Virtual DOM 必然没有你直接操作 DOM 来的快，毕竟还有一层 diff 算法的损耗。

当然了 Virtual DOM 提高性能是其中一个优势，其实最大的优势还是在于：

- 将 Virtual DOM 作为一个容器，让我们还能对接非 Web 端的系统，实现跨端开发。
- 同样的，通过 Virtual DOM 我们可以渲染到其他的平台，比如实现 SSR、同构渲染等等。
- 实现组件的高度抽象化

路由原理

涉及面试题：前端路由原理？两种实现方式有什么区别？

前端路由实现起来其实很简单，本质就是监听 URL 的变化，然后匹配路由规则，显示相应的页面，并且无须刷新页面，目前前端使用的路由就只有两种实现方式：

Hash 模式

// 前端路由 // Hash 模式，本质上和 React 的 **location** 方式没什么区别，只是 URL 的变化是通过 hash 来实现的，从而实现路由跳转。

那么既然 DOM 可以通过 JS 对象来模拟，反之也可以通过 JS 对象来渲染出对应的 DOM，当然了，通过 JS 来模拟 DOM 并且渲染对应的 DOM 只是第一步，难点在于如何判断旧两个 JS 对象的最小差异并且实现局部更新 DOM。

首先 DOM 是一个多叉树的结构，如果需要完整的对比两棵树的差异，那么需要的时间复杂度会是 $O(n^2 \cdot 3^h)$ 。这个复杂的算法是不现实的，于是 React 团队优化了算法，实现了 $O(h)$ 的复杂度的差异对比，实现 $O(h)$ 复杂度的关键就是只对比树的节点，而不是跨树对比，这也是考虑到在实际业务中会经常对树的某一个节点进行修改，所以判断差异就分为了两步：

- 首先从上至下，从左到右遍历对象，也就是树的深度遍历，这一步中会将每个节点添加索引，便于最后差异对比。
- 一旦节点有子元素，就去判断子元素是否有不同

在第一步算法中我们需要判断旧节点的 **tagName** 是否相同，如果不相同的话就代表节点被替换了。如果有更改 **tagName** 的话，就需要判断是否有子元素，有的话就进行第二步算法。

在第二步算法中，我们需要判断原本的列表中是否有节点被删除，在新的列表中需要判断是否有新的节点加入，还需要判断节点是否由有变无。

History 模式

History 模式是 HTML5 新推出的功能，主要使用 **history.pushState** 和 **history.replaceState** 改变 URL，实现历史记录。

通过 History 模式改变 URL 同样不会引起页面的刷新，只会更新浏览器的历史记录。

// 前端路由 // History 模式，本质上和 React 的 **location** 方式没什么区别，只是 URL 的变化是通过 history 来实现的，从而实现路由跳转。

那么既然 DOM 可以通过 JS 对象来模拟，反之也可以通过 JS 对象来渲染出对应的 DOM，当然了，通过 JS 来模拟 DOM 并且渲染对应的 DOM 只是第一步，难点在于如何判断旧两个 JS 对象的最小差异并且实现局部更新 DOM。

首先 DOM 是一个多叉树的结构，如果需要完整的对比两棵树的差异，那么需要的时间复杂度会是 $O(n^2 \cdot 3^h)$ 。这个复杂的算法是不现实的，于是 React 团队优化了算法，实现了 $O(h)$ 的复杂度的差异对比，实现 $O(h)$ 复杂度的关键就是只对比树的节点，而不是跨树对比，这也是考虑到在实际业务中会经常对树的某一个节点进行修改，所以判断差异就分为了两步：

- 首先从上至下，从左到右遍历对象，也就是树的深度遍历，这一步中会将每个节点添加索引，便于最后差异对比。
- 一旦节点有子元素，就去判断子元素是否有不同

在第一步算法中我们需要判断旧节点的 **tagName** 是否相同，如果不相同的话就代表节点被替换了。如果有更改 **tagName** 的话，就需要判断是否有子元素，有的话就进行第二步算法。

在第二步算法中，我们需要判断原本的列表中是否有节点被删除，在新的列表中需要判断是否有新的节点加入，还需要判断节点是否由有变无。

两者对比

- Hash 模式只可以更改 URL 中的内容，History 模式可以通过 API 设置任意的回源 URL。
- History 模式可以通过 API 添加任意类型的数据到历史记录中，Hash 模式只能更改哈希值，也就是字符串。
- Hash 模式无法配置全局路由，且兼容性不好。History 模式将视图中的状态和用户的动作分离出来一个抽象，这样才是 MVVM 的精髓。

Vue 和 React 之间的区别

Vue 的表单可以使用 **v-model** 支持双向绑定，相比于 React 来说开发上更加方便，当然了 **v-model** 其实就是个语法糖，本质上和 React 表单的方式没什么区别。

改变数据方式不同，Vue 修改数据相对来说要更简单许多，React 需要使用 **useState** 来改变状态，并且使用这个 API 也有不少坑点。并且 Vue 的双向绑定用了劫持属性，页面更新就已经是最优的了，但是 React 还是要用户手动去优化这方面的问题。

React 16 以后，有些钩子函数会执行多次，这是因为引入 Fiber 的原因，这在后续的章节中会讲到。

React 需要使用 JSX，有一定的上手成本，并且需要一套新的工具链支持，但是完全可以通过 JS 来控制页面，更加的灵活。Vue 使用了模板语法，相比于 JSX 来说没有那么灵活，但是完全可以说脱离工具链，通过直接编写 **render** 函数就能在浏览器中运行。

在生态上来说，两者其实没多大的差距，当然 React 的用户是远远高于 Vue 的。

在上手成本上来说，Vue 一开始的定位就是尽可能的降低前端开发的门槛，然而 React 更多的是去改变用户接受它的概念和思维，相较于 Vue 来说上手成本略高。

小结

这一章节中我们学习了几个框架中的相似点，也对比了 React 和 Vue 之间的区别。其实我们可以发现，React 和 Vue 虽然是两个不同的框架，但是他们的底层原理都是很相似的，无非是在上面堆砌了自己的概念上去。所以我们无需对比到底哪个框架牛逼，引用尤大一句话：

说到底，就算你证明了 A 比 B 牛逼，也不意味着你或者你的项目就牛逼了... 比起争这个，不如多想想怎么让自己变得牛逼吧。

留言

输入评论 (Enter 提交, Ctrl + Enter 加速)

发表评论

全部评论 (46)

白宇 (2) | 前端开发工程师 | 1年前
感觉是在说 react?

白宇 (2) | 前端 | 1年前
React和Vue的驼峰命名法是不一样的

白宇 (2) | 前端 | 1年前
前端古力士 (2) | 前端工程师 | 1年前
我觉得小白的回复和你说可以展开讲解一下，只讲了简单的道理，不是很能理解。记得你提过旧版的 ViewModel 就是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白宇 (2) | 前端 | 1年前
感觉这个 ViewModel 只是把所有的东西都放到一个 ViewModel 中，让多个 View 共享这个 ViewModel。

白

Vue 常考基础知识点

这一章节我们将来学习 Vue 的一些经常考到的基础知识点。

生命周期钩子函数

在 `beforeCreate` 钩子函数调用的时候，是获取不到 `props` 或者 `data` 中的数据的，因为这些数据的初始化都在 `initState` 中。

然后会执行 `created` 钩子函数，在这一步的时候已经可以访问到之前不能访问到的数据，但是这时候组件还没被渲染，所以是看不到的。

接下来会先执行 `beforeMount` 钩子函数，开始创建 VDOM，最后执行 `mounted` 钩子，并将 VDOM 渲染为真实 DOM 并且渲染数据，组件中如果有子组件的话，会递归挂载子组件，只有当所有子组件全部加载完毕，才会执行根组件的 `mounted` 钩子。

接下来是数据更新时会调用的钩子函数 `beforeUpdate` 和 `updated`，这两个钩子函数没什么好说的，就是分别在数据更新前和更新后会调用。

另外还有 `keep-alive` 独有的生命周期，分别为 `activated` 和 `deactivated`。用 `keep-alive` 包裹的组件会切换，而是缓存到内存中并执行 `deactivated` 钩子函数，命中缓存渲染后会执行 `activated` 钩子函数。

组件通信

组件通信一般分为以下几种情况：

- 父子组件通信
- 兄弟组件通信
- 跨多层级组件通信
- 任意组件

对于以上每种情况都有多种方式去实现，接下来就来学习下如何实现。

父子通信

父组件通过 `props` 传递数据给子组件，子组件通过 `emit` 发送事件传递数据给父组件，这两种方式是最常用的父子通信实现办法。

这种父子通信方式也就是典型的单向数据流，父组件通过 `props` 传递数据，子组件不能直接修改 `props`，而是必须通过发送事件的方式告知父组件修改数据。

另外这两种方式还可以使用语法糖 `v-model` 来直接实现，因为 `v-model` 默认会解析成名为 `value` 的 `value` 或者 `input` 来直接实现，这是典型的双向绑定，常用于 UI 控件上，但是究其根本，还是通过事件的方法让父组件修改数据。

当然我们还可以通过访问 `$parent` 或者 `$children` 对象来访问组件实例中的方法和数据。

另外如果你使用 Vue 2.3 及以上版本的话还可以使用 `$listeners` 和 `.$sync` 这两个属性。

`$listeners` 属性会将父组件中的（不含 `native` 修饰器的）`v-on` 事件监听器传递给子组件，子组件可以通过访问 `$listeners` 来自定义监听器。

`.$sync` 属性是个透传器，可以很简单的实现子组件与父组件通信

```
<!--父组件-->
<input value="value" />
</div>上面的会同步-->
<input :value="value" @update:value="value = $event.target.value" />
</div>
<script>
this.$emit('update:value', 1)
</script>
```

兄弟组件通信

对于这种情况又可以通过直接父组件中的子组件实现。也就是 `this.$parent.$children`，在 `$children` 中可以通过组件 `name` 查找到需要的组件实例，然后进行通信。

跨多层次组件通信

对于这种情况又可以使用 Vue 2.2 新增的 API `provide / inject`，虽然文档中不推荐直接使用在业务中，但是如果用得好的话还是很有用的。

假设有个组件 A，然后有一个跨多层级的子组件 B

```
//父组件 A
export default {
  provide: {
    data: 1
  }
}
//子组件 B
export default {
  inject: ['data'],
  mounted() {
    // 无论几层都能获得父组件的 data 属性
    console.log(this.data) // => 1
  }
}
```

任意组件

这种方式可以通过 Vuex 或者 Event Bus 解决，另外如果你不怕麻烦的话，可以使用这种方式解决上述所有的问题。

extend 能做什么

这个 API 很少用到，作用是扩展组件生成一个构造器，通常会与 `mount` 一起使用。

```
// 创建并注册组件
let Component = Vue.extend({
  template: ''
})
// 在应用 App 上
new Component().$mount('#app')
// 除了上面的方式，还可以通过扩展已有组件
let superComponent = Vue.extend(Component)
new superComponent({
  created() {
    console.log(1)
  }
})
new superComponent().$mount('#app')
```

mixin 和 mixins 区别

`mixins` 用于全局混入，会影响到每个组件实例，通常插件都是这样做的。

```
Vue.mixin({ // 混入对象
  beforeCreate() {
    // ...逻辑
    // 这样会影响到每个组件的 beforeCreate 钩子函数
  }
})
```

虽然文档不建议我们在应用中直接使用 `mixins`，但是如果不用的话也是很有帮助的，比如可以全局混入装饰好的 `ajax` 或者一些工具函数等等。

`mixins` 应该是我们最常使用的扩展组件的方式了，如果多个组件中有相同的业务逻辑，就可以将这些逻辑抽离出来，通过 `mixins` 混入代码，比如上拉加载数据这种逻辑等等。

另外需要注意的是 `mixins` 插入的钩子函数会优先于组件内的钩子函数执行，并且在遇到同名选项的时候会选择性的进行合并，具体可以阅读 [文档](#)。

computed 和 watch 区别

`computed` 是计算属性，依赖其他属性计算值，并且 `computed` 的值有缓存，只有当计算值变化才会返回内容。

`watch` 监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作。

所以一般来说需要依赖属性来动态获得值的时候可以使用 `computed`，对于监听到值的变化需要做一些复杂业务逻辑的情况可以使用 `watch`。

另外 `computed` 和 `watch` 还都支持对象的写法，这种方式知道的人并不多。

```
vm.$watch('obj', {
  // 所有监听
  deep: true,
  // 只监听对象
  immediate: true,
  // 执行函数
  handler: function(val, oldVal) {
  }
})
var vm = new Vue({
  data: { a: 1 },
  computed: {
    abc: {
      // this.$watch('abc') 时候
      get: function () {
        return this.a + 1
      },
      // this.$watch('abc', 1) 时候
      set: function (v) {
        this.a = v - 1
      }
    }
  }
})
```

keep-alive 组件有什么作用

如果你需要在组件切换的时候，保留一些组件的状态防止多次渲染，就可以使用 `keep-alive` 组件包裹需要保存的组件。

对于 `keep-alive` 组件来说，它拥有两个独有的生命周期钩子函数，分别为 `activated` 和 `deactivated`，用 `keep-alive` 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 `deactivated` 钩子函数，命中缓存渲染后会执行 `activated` 钩子函数。

v-show 与 v-if 区别

`v-show` 只是在 `display: none` 和 `display: block` 之间切换，无论初始条件是什么都会被渲染出来，后面只是显示或隐藏 CSS，DOM 还是一直保留着的，所以总的来说 `v-show` 在初始渲染时有更多的开销，但是切换会非常快，更适合频繁切换的场景。

`v-if` 的话记得识别 Vue 底层的编译了，当属性初始为 `false` 时，组件就不会被渲染，直到条件为 `true`，并且切换条件时会触发销毁/注册组件，所以总的来说在切换时开销更高，更适合不经常切换的场景。

并且基于 `v-if` 的这种惰性渲染机制，可以在必要的时候才去渲染组件，减少整个页面的初始渲染开销。

组件中 data 什么时候可以使用对象

这道题目其实更多考的是 JS 动态。

组件使用时所有组件实例都会共享 `data`，如果 `data` 是对象的话，就会造成一个组件修改 `data` 以后会影响到其他所有组件，所以需要将 `data` 定义成函数，每次用到就调用一次函数获得新的数据。

当我们使用 `new Vue()` 的方式的时候，无论我们将 `data` 设置为对象还是数组都是可以的，因为 `new Vue()` 的方式是生成一个根组件，该组件不会被复用，也就不存在共享 `data` 的情况了。

小结

总的来说这一章节的内容更多的偏向于 Vue 的基础，下一章节我们将来了解一些原理性方面的知识。

留言

输入评论 (Enter执行, Ctrl + Enter提交)

发表评论

全部评论 (24)

不写bug的公子 | 程序员 | 1年前
Vue 基础问题大家可以结合这个一起看，查漏补缺：[@mihanyi/githook](#)

蒋毅 | 3年前
别被吓到666 | 3年前
怎么厉害归属

点点赞 | 3年前
点点赞

lowayYoung | 资深前端 | 2年前
生命周期是不是没更新呢

点点赞 | 2年前
点点赞

GlauberSIE | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

aprilfly_wu | 2年前
`v-show` 不是在 `display:none` 和 `display:block` 之间切换，你没想，如果一个元素是内部标签，`display:block` 就会把它变成块级标签了么

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞

蒋毅 | 前端 | 2年前
求更新

点点赞 | 2年前
点点赞 <img alt="头像" data-bbox="100 1976

React 常考基础知识点

这一章节我们将来学习 React 的一些经常考到的基础知识点。

生命周期

在 V16 版本中引入了 Fiber 机制。这个机制一定程度上影响了部分生命周期的调用，并且也引入了新的 2 个 API 来解决这些问题。关于 Fiber 的内容将会在下一章节中讲到。

在之前的版本中，如果你拥有一个很复杂的复合组件，然后改动了最顶层组件的 `state`，那么调用栈可能很长。



调用栈过长，再加上中间进行了复杂的操作，就可能导致长时间阻塞主线程，带来不好的用户体验。Fiber 就是为了解决这个问题而生的。

Fiber 基本上是一个虚拟的堆栈帧。新的调度器会按照优先级自由调度这些帧，从而将之前的同步渲染改成了异步渲染，在不影响体验的情况下分阶段更新。



对于如何区别优先级，React 有自己的一套逻辑。对于动画这种实时性很高的东西，也就是 16 ms 必须保证一次保证不卡顿的情况下，React 会每 16 ms（以内）暂停一下更新，返回来继续渲染动画。

对于异步渲染，现在渲染有两个阶段：`reconciliation` 和 `commit`，前者过程是可以打断的，后者不能暂停，会一直更新直到界面真正完成。

Reconciliation 阶段

- `componentWillMount`
- `componentDidMount`
- `shouldComponentUpdate`
- `componentWillUpdate`

Commit 阶段

- `componentDidUpdate`
- `componentWillUnmount`

因为 Reconciliation 阶段是可以被打断的，所以 Reconciliation 阶段会执行的生命周期函数就可能会出现调用多次的情况，从而引起 Bug。由于对于 Reconciliation 阶段调用的几个函数，除了 `shouldComponentUpdate` 以外，其他都应该避免去使用。并且 V16 中也引入了新的 API 来解决这个问题。

`getDerivedStateFromProps` 用于替换 `componentWillReceiveProps`，该函数会在初始化和 `update` 时被调用。

```
class ExampleComponent extends React.Component {
  // 初始化 state 在 constructor。
  // 或者通过 a property 初始化。
  state = {}

  static getDerivedStateFromProps(nextProps, prevState) {
    if (nextProps.name === prevState.name) {
      return {
        derivedData: computeDerivedState(nextProps),
        name: nextProps.name
      }
    }
  }

  // 返回 null 表示 no change to state.
  // 或者返回 {}
}

getSnapshotBeforeUpdate 用于替换 componentWillUpdate，该函数会在 update 后 DOM 更新前被调用，用于读取最新的 DOM 数据。
```

setState

`setState` 在 React 中经常使用的一个 API，但是它存在一些的问题经常会导致初学者出错，核心原因是因为这个 API 是异步的。

首先 `setState` 的调用并不会马上引起 `state` 的改变，并且如果你一次调用了多个 `setState`，那么结果可能并不如你期待的一样。

```
handle() {
  console.log('count: ', this.state.count) // => 0
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  console.log(this.state.count) // => 0
}
```

第一次的打印都为 0，因为 `setState` 是个异步 API，只有同步代码运行完毕才会执行。`setState` 异步的原因被认为在于，`setState` 可能会导致 DOM 的重绘，如果调用一次就马上去进行重绘，那么调用多次就会造成不必要的性能损失。设计成异步的话，就可以将多次调用放入一个队列中，在恰当的时候统一进行更新操作。

第二，虽然更新了三次 `setState`，但是 `count` 的值还是为 1。因为多次调用会合并为一次，只有当更新结束后 `state` 才会改变，三次调用等同于如下代码

```
object.assign(
  {},
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
)
```

当然你也可以通过以下方式来实现调用三次 `setState` 使用 `count` 为 3

```
handle() {
  this.setState({count: this.state.count + 1})
  this.setState({count: this.state.count + 1})
  this.setState({count: this.state.count + 1})
}
```

如果你想在每次调用 `setState` 后获得正确的 `state`，可以通过如下代码实现

```
handle() {
  this.setState({count: this.state.count + 1}, () => {
    console.log(this.state)
  })
}
```

性能优化

这一小节内容集中在组件的性能优化上。这一方面的性能优化也基本集中在 `shouldComponentUpdate` 这个钩子函数上做文章。

P5：下面的 state 相当于 state 及 props

在 `shouldComponentUpdate` 函数中我们可以通过返回布尔值来决定当前组件是否需要更新。这层代码逻辑可以是简单的比较一下当前 `state` 和之前的 `state` 是否相同，也可以是判断某个值更新了才触发组件更新。一般来讲不推荐完全地对比当前 `state` 和之前的 `state` 是否相同，因为每次更新触发可能会很浪费。这样的完整对比性能开销会有点大，可能会造成不得不往大的情况。

当然如果真的想完整地对比当前 `state` 和之前的 `state` 是否相同，并且不考虑性能也是行得通的。可以通过 `immutable` 或者 `immer` 这些库来实现不可变对象。这类库对于操作大规模的数据来说会提升不错的性能，并且一旦改变数据就会生成一个新的对象。对比前后 `state` 是否一致也就方便多了，同时也推荐阅读下 `immer` 的源码实现。

另外如果只是单纯的浅比较一下，可以直接使用 `pureComponent`，底层就是实现了浅比较 `state`。

```
class Test extends React.PureComponent {
  render() {
    // ...
  }
}

class Child extends React.Component {
  render() {
    // ...
  }
}
```

这时候你可能会考虑到函数组件就不能使用这种方式了，如果你使用 16.6.0 之后的版本的话，可以使用 `React.memo` 来实现相同的功能。

```
class Test = React.memo(() => {
  // ...
})
```

通过这种方式我们就可以原实现了 `shouldComponentUpdate` 的浅比较，又能够使用函数组件。

通信

其实 React 中的组件通信基本和 Vue 中的一致。同样也分为以下三种情况：

- 父子组件通信

父子组件通过 `props` 传递数据给子组件，子组件通过调用父组件传来的函数作函数给父组件，这两种方式是最常用的父子组件实现办法。

这种父子通信方式也就是典型的单向数据流，父组件通过 `props` 传递数据，子组件不能直接修改 `props`，而是必须通过调用父组件函数的方式告知父组件修改数据。

兄弟组件通信

对于这种情况可以通过共同的父组件来管理状态和事件函数。比如说其中一个兄弟组件调用父组件传递过来的事件函数修改父组件中的状态，然后父组件将状态传递给另一个兄弟组件。

跨多层次组件通信

如果你使用 16.3 以上版本的话，对于这种情况可以使用 Context API。

```
// 创建 Context，可以在开始使用时引入
const StateContext = React.createContext()
class Parent extends React.Component {
  render() {
    // ...
  }
}
class Child extends React.Component {
  render() {
    // ...
  }
}
```

这时候你可能会考虑到函数组件就不能使用这种方式了，如果你使用 16.6.0 之后的版本的话，可以使用 `React.memo` 来实现相同的功能。

```
class Test = React.memo(() => {
  // ...
})
```

通过这种方式我们就可以原实现了 `shouldComponentUpdate` 的浅比较，又能够使用函数组件。

任意组件

这种方式可以通过 Redux 或者 Event Bus 解决，另外如果你不怕麻烦的话，可以使用这种方式解决所有组件通信情况。

小结

总的来说这一章节的内容更多的偏向于 React 的基础，另外 React 的面试题还会经常考到 Virtual DOM 中的内容，所以这块内容大家也需要好好准备。

下一章节我们将来了解一些 React 的进阶知识点。

首先我们进入评论区，输入评论（Enter执行，Ctrl + Enter发送）

发送评论

正在加载评论，请稍候...

暂无评论

回复

React 常考进阶知识点

这一章节我们将来学习 React 的一些经常考到的进阶知识点，并且这章节还需要配合第十九章阅读，其中的内容经常会考到。

HOC 是什么？相比 mixins 有什么优点？

很多人看到高阶组件（HOC）这个概念就被吓到了，认为这东西很难，其实这东西概念真的很简单，我们先来看一个例子。

```
function add(a, b) {
  return a + b
}
```

现在如果我想给这个 `add` 函数添加一个输出结果的功能，那么你可能会考虑直接使用 `console.log` 不就解决问题了么。说的没错，但是如果我們想做的更加优雅并且容易复用和扩展，我们可以这样去做：

```
function withLog(Wrapped) {
  function wrapper(a, b) {
    const result = Wrapped(a, b)
    console.log(result)
    return result
  }
  return wrapper
}
const withLogging = withLog(add)
withLogging(1, 2)
```

其实这个做法在函数式编程里称之为高阶函数。大家都应该知道 React 的思想里是存在函数式编程的，高阶组件和高阶函数就是同一个东西。我们实现一个函数，传入一个组件，然后在函数内部再实现一个函数去扩展现成的组件，最后返回一个新的组件，这就是高阶组件的概念，作用就是为了更好的复用代码。

其实 HOC 和 Vue 中的 mixins 作用是一致的，并且在早期 React 也是使用 mixins 的方式。但是在使用 class 的方式创建组件以后，mixins 的方式就不够使用了，并且其实 mixins 也是存在一些问题的，比如：

- 合成了很多依赖。比如我在组件中写了某个 `state` 并且在 `mixins` 中使用了，那就存在了一个依赖关系。万一下次别人要移除它，就得去 `mixins` 中查找依赖
- 多个 `mixins` 中可能存在相同的命名的函数，同时代碼中也不能出现相同命名的函数，否则就是报错。其实我一个组件还是使用着同一个 `mixins`，但是一个 `mixins` 会被多个组件使用，可能会存在需求使得 `mixins` 修改原本的函数或者新增更多的函数，这样可能就会产生一个维护成本

HOC 解决了这些问题，并且它们达成的效果也是一致的，同时也更加的政治正确（毕竟更加函数式了）。

事件机制

React 其实自己实现了一套事件机制，首先我们考虑一下以下代码：

```
const Test = ({list, handleclick}) => {
  list.map(item, index) => {
    <a href={item.handleClick} key={index}>{item}</a>
  }
}
```

以上类似代码想必大家经常会写到，但是你是否考虑过点击事件是否绑定在了每一个标签上？事实当然不是，JSX 上面的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent），因此我们如果不需要事件冒泡的话，调用 `event.stopPropagation` 是无效的，而应该使用 `event.preventDefault`。

那么实现合成分事件的目的是什么呢？总的来说在我看来好处有两点，分别是：

- 合成事件首先解决了浏览器之间的兼容问题，另外这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力
- 对于浏览器开发者来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，那么就需要分配很多的事件对象，造成高额的内存分配问题。当事件需要被使用时，就会从池子中复用对象。事件回调结束后，就会销毁事件对象的属性，从而便于下次复用事件对象。

更新内容

- [React 进阶系列：Hooks 该怎么用](#)

小结

你可能会惊讶于这一章节的内容并不多的情况，其实你如果将两章 React 以及第十九章的内容全部学习完后，基本上 React 的大部分面试问题都可以解决。

当然你可能会觉得看的不过瘾，这不需要担心，我已经决定写一个免费专栏「React 进阶」，专门讲解一些深度的问题，比如组件的设计模式、新特性、部分双向解绑等等内容。当然这些内容都是需要好好打磨的，所以更新的不会很快，有兴趣的可以持续关注，都会更新链接在这一章节中。

留言

输入评论 (Enter执行 | Ctrl + Enter预览)

发表评论

全部评论 (25)

方舟 | 7小时前
现在不能挂在document上了

白 哥 | 回复

前辈 | 6月前
说的好下一章讲fiber 呢

白 哥 | 回复

mytac | 独立开发 | 独立开发 | 2年前
感觉HOC远不如高阶函数，被鄙视的频率比较高

白 哥 | 回复

DarkKnight | 前端工程师 | 前端工程师 | 2年前
感觉HOC远不如高阶函数的简洁和直观吧，hook应该是函数式的政治正确，而hook的诞生几乎就可以说是大势所趋

白 哥 | 回复

前端古力士 | 前端工程师 | 前端工程师 | 2年前
老哥我想请问下react的生态该考虑些?

白 哥 | 回复

王唯性 | developer | 贡献 | 2年前
另外冒泡到 document 上的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent），因此我们如果不想事件冒泡的话，调用 event.stopPropagation 是无效的，而应该使用 event.preventDefault 才是阻止冒泡的方法，这个地方写错了吧

白 哥 | 回复

王唯性 | developer | 贡献 | 2年前
event.stopPropagation 阻止合成分事件 亂想的方法

白 哥 | 回复

耿晓峰 | 前端工程师 | 前端工程师 | 2年前
感谢！感觉还满意 @字... | 2年前

白 哥 | 回复

白 哥 | 前端工程师 | 前端工程师 | 3年前
阻止冒泡不是 event.stopPropagation() 这个吗？

白 哥 | 回复

我只有三块板 | 前端开发 | 前端开发 | 3年前
博主说的挺对的

白 哥 | 回复

白 哥 | 前端工程师 | 前端工程师 | 3年前
react的高阶感觉没多用啊。

白 哥 | 回复

ZedCoding | 前端 | 杭州高级切图... | 3年前
没使用

白 哥 | 回复

牛哥 | 感谢达人 | 3年前
说实话，受限于篇幅长度，这篇「React 进阶」其实有点浅，不知道加一个 React 进阶的进阶图

白 哥 | 回复

张鹤枫 | 高级前端 | 高级前端 | 3年前
0.0

白 哥 | 回复

小林其 | 软件工程师 | 软件公司 | 3年前
不知道怎么说，想问先讲了HOC，却没深入讲一下高阶化和高阶组件的生命周期；我很不解的render props和render callback也不讲一下吗？

白 哥 | 回复

yeek | 作者 | 3年前
柯里化就更加是 FP 的东西了，说了会有更多人更加模棱。另外高阶组件的生命周期和别的生命周期不一样了！其他你讲的 render props 这点，近期会更新新的文章的

白 哥 | 回复

正直果真 | 2年前
期待React进阶

白 哥 | 回复

查看更多回复 ↴

监控

前端监控一般分为三种，分别为页面埋点、性能监控以及异常监控。

这一章节我们将来学习这些监控相关的内容，但是基本不会涉及到代码，只是让大家了解下前端监控该用什么方式实现。毕竟大部分公司都只是使用到了第三方的监控工具，而不是选择自己造轮子。

页面埋点

页面埋点应该是大家最常用的监控了，一般起码会监控以下几个数据：

- PV / UV
- 停留时长
- 流量来源
- 用户交互

对于这几类统计，一般实现思路大致可以分为两种，分别为手写埋点和无埋点的方式。

相信第一种方式也是大家最常用的方式，可以自由选择需要监控的数据然后在相应的地方写入代码。这种方式的灵活性很大，但是唯一的缺点就是工作量较大，每个需要监控的地方都得插入代码。

另一种无埋点的方式基本不需要开发者手写埋点了，而是统计所有的事件并且定时上报，这种方式虽然没有前一种方式繁琐了，但是因为统计的是所有事件，所以还需要后期过滤出需要的数据。

性能监控

性能监控可以帮助开发者了解到在各种真实环境下，页面的性能情况是如何的。

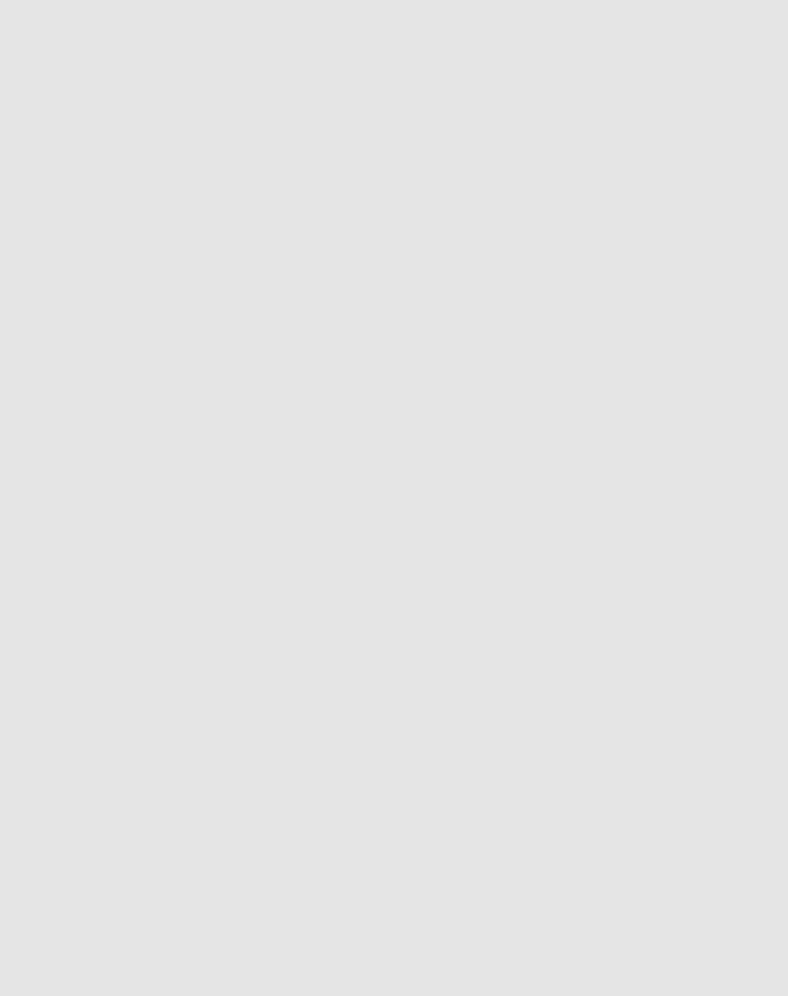
对于性能监控来说，我们可以直接使用浏览器自带的 `Performance API` 来实现这个功能。

对于性能监控来说，其实我们只需要调用 `performance.getEntriesByType('navigation')` 这行代码就可以了，你没看错，一行代码就可以获得页面中各种详细的性能相关信息。

```
> performance.getEntriesByType('navigation')
+-- [PerformanceEntry] {type: "navigation", startTime: 14.1599999991534278, connectEnd: 14.1599999991534278, connectStart: 14.1599999991534278, decodeBodySize: 17581, encodeEnd: 1809.3649999994517, encodeStart: 1791.22888882887, eventContentloadedEventStart: 1805.6849999992084, eventInterconnectLookupEnd: 14.1599999991534278, eventInterconnectLookupStart: 14.1599999991534278, eventLoadEventEnd: 1805.6849999992084, eventLoadEventStart: 14.1599999991534278, fetchEnd: 1805.6849999992084, fetchStart: 14.1599999991534278, fetchType: "navigation", initiatorType: "navigation", loadEventEnd: 1805.6849999992084, loadEventStart: 14.1599999991534278, name: "https://www.w3c.in", redirectCount: 0, redirectEnd: 0, redirectStart: 0, requestEnd: 1805.6849999992084, requestStart: 14.1599999991534278, responseEnd: 1805.6849999992084, responseStart: 14.1599999991534278, secureConnectStart: 14.1599999991534278, secureConnectEnd: 14.1599999991534278, transferSize: 0, unloadEventEnd: 0, unloadEventStart: 0, workerStart: 0}
```

② 国金技术社区

我们可以发现运行代码返回了一个数组，内部包含了相当多的信息，从数据开始在网络中传输到页面加载完成都提供了相应的数据。



异常监控

对于异常监控来说，以下两种监控是必不可少的，分别是代码错误以及接口异常上报。

对于代码运行错误，通常的办法是使用 `window.onerror` 捕获报错。该方法能拦截到大部分的捕获报错信息，但是也有例外

- 对于跨域的代码运行错误会显示 `Script error`，对于这种情况我们需要给 `script` 标签添加 `crossorigin` 属性
- 对于某些浏览器可能不会显示调用纯信息，这种情况可以通过 `arguments.callee.caller` 来做找迹归

对于跨步代码来说，可以使用 `catch` 的方式捕获错误。比如 `Promise` 可以直接使用 `catch` 错误。`async await` 可以使用 `try catch`。

但是要注意线上运行的代码都是压缩过的，需要在打包时生成 `sourceMap` 文件便于 `debug`。

对于捕获的错误需要上传给服务器，通常可以通过 `log` 标签的 `src` 发起一个请求。

另外接口异常相对来说比较简单，可以列举出前面的状态码。一旦出现此类的状态码就可以立即上报出错，接口异常上报可以让开发人员迅速知道哪些接口出现了大面积的报错，以便迅速排查问题。

小结

这一章节内容虽然不多，但是这类基础的知识网上的资料确实不多，相信能给大家一个不错的思路。

留言

输入评论 (Enter换行, Ctrl + Enter发送)

发表评论

全部评论 (20)

内小子 | 前端 | 1年前
useTimeout里面抛出异常怎么捕获
↓ 0 点赞 ↓ 0 回复

breeze | 1年前
... 把数据的
↓ 4 点赞 ↓ 0 回复

回复 9年前
想当然，不是图
↓ 2 点赞 ↓ 0 回复

石头就是我54394 | 1年前
看了demo的文章，看着不离高大上，反而感觉烂烂的
↓ 1 点赞 ↓ 0 回复

CregulaN | 前端学习 | 1年前
异常监控部分，是否能用ErrorBoundary解决的呢? [#zh-hans.reactjs.org](#)
↓ 0 点赞 ↓ 0 回复

页面工具人 | 页面 @ 未知 | 2年前
有点烦，但不得而知时间那么短，要是被问，哑口无言。
↓ 3 点赞 ↓ 0 回复

console_man | Web前端 | 2年前
期待讲一下大埋点内容
↓ 0 点赞 ↓ 0 回复

腾讯用户体验 | 前端 | 2年前
监控分类型的话“页面埋点”和“接口埋点”两个不是同一个类型的东西，后面两个是从监控的需求，“页面埋点”也可以用来做性能监控。个人认为应该把“页面埋点”换成“用户行为监控”。
↓ 0 点赞 ↓ 0 回复

wei | fe dev | 2年前
这属于杠杠知识。。。
↓ 17 点赞 ↓ 0 回复

平百姓海 | 2年前
在过年的时候写完，有那么一丢丢赶进度的感觉
↓ 0 点赞 ↓ 0 回复

石头就是我5_ | 1年前
写的什么啊，谁编的
↓ 0 点赞 ↓ 0 回复

suhangdev | Node @ Ant | 2年前
能不能分享下前端无埋点方案的实现思路
↓ 0 点赞 ↓ 0 回复

PerCye4800 | 3年前
没看够啊，这里内容不够仔细，只是知道个大概
↓ 0 点赞 ↓ 0 回复

anxiplus | web前端 @ Coding | 3年前
感谢后面链接的大佬，其实知识点很多，但需要自己理解一下
↓ 0 点赞 ↓ 0 回复

Bin_zhu | 3年前
药魄大明
↓ 0 点赞 ↓ 0 回复

MingKuan | 公众号: 一只前端 | 3年前
吐到垃圾桶认个怂
↓ 0 点赞 ↓ 0 回复

566 | 3年前
里面的知识也太少了吧
↓ 0 点赞 ↓ 0 回复

crackdlove | 前端攻城师 | 3年前
怎么没人?
↓ 0 点赞 ↓ 0 回复

UDP

网络协议是每个前端工程师都必须要掌握的知识，我们将先来学习传输层中的两个协议：UDP以及TCP，对于大部分工程师来说最常用的协议也就是这两个了，并且面试中经常会被提问的也是关于这两个协议的区别。

我们先来解答这个常考面试题关于 UDP 部分的内容，然后在详细去学习这个协议。

常考面试题：UDP 和 TCP 的区别是什么？

首先 UDP 协议是面向无连接的，也就是说不需要在正式发送数据之前先连接起双方，然后 UDP 协议只是数据报文的搬运工，不保证有序且不丢失的传递到对端，并且 UDP 协议也没有任何控制流量的算法，总的来说 UDP 相较于 TCP 更加的轻便。

面向无连接

首先 UDP 是不需要和 TCP 一样在发送数据前进行三次握手建立连接的，想发数据就可以开始发送了。

并且也只是数据报文的搬运工，不会对数据报文进行任何拆分和接操作。

具体来说就是：

- 在发送端，应用层将数据传递给传输层的 UDP 协议，UDP 只会给数据增加一个 UDP 头标识一下是 UDP 协议，然后就将数据交给应用层了
- 在接收端，将数据传递给传输层的 UDP，UDP 只去除 IP 报文头就将数据交给应用层，不会任何拼接操作

不可靠性

首先不可靠性体现在无连接上，通信都不需要建立连接，想发就发，这样的情况肯定不可靠。

并且收到什么数据就库送什么数据，并且也不会备份数据，发送数据也不会关心对方是否已经正确收到数据了。

再者网络环境不好时坏，但是 UDP 因为没有拥塞控制，一直会以恒定的速度发送数据，即使网络条件不好，也不会对发送速率进行调整，这样实现的弊端就是在网络条件不好的情况下可能会导致丢包，但是优点也很明显，在某些实时性要求高的场景（比如电话会议）就需要使用 UDP 而不是 TCP。

高效

虽然 UDP 收不是那么的可靠，但是正是因为它不是那么的可靠，所以也就没有 TCP 那么复杂了，需要保证数据不丢且有序到达。

因此 UDP 的头部较小，只有八字节，相比 TCP 的至少二十字节要少得多，在传输数据报文时是很高效的。

UDP Header		
0	1	2
Source port		Destination port
Length		Checksum

UDP 头部包含了以下几个数据

- 每个十六位的端口号，分别为源端口（可选字段）和目标端口
- 整个数据报文的长度
- 整个数据报文的检验和（IPv4 可选字段），该字段用于发现头部信息和数据中的错误

传输方式

UDP 不止支持一对一的传输方式，同样支持一对多，多对多，多对一的方式。也就是说 UDP 提供了单播、多播、广播的功能。

适合使用的场景

UDP 虽然比 TCP 有很多缺点，但是正是因为这些缺点造就了它高效的特点，在很多实时性要求高的地方都可以看到 UDP 的身影。

直播

想必大家都看过直播吧，大家可以考虑下如果直播使用了基于 TCP 的协议会发生什么事情？

TCP 会严格控制传输的正确性，一旦有某一个数据对端没有收到，就会停下来直到对端收到这个数据，这种问题在网络条件不好的情况下可能并不会发生什么事情，但是在网络情况差的时候就会变成画面卡住，然后再继续播放下一帧的情况。

但是对于直播来说，用户肯定关注的是最新的画面，而不是因为网络条件差而丢弃的老旧画面，所以 TCP 在这种情况下无用武之地，只会降低用户体验。

王者荣耀

虽然我具体不知道王者荣耀底层使用了什么协议，但是对于这类实时性要求很高的游戏来说，UDP 是离不掉的。

为什么这样说呢？首先对于王者荣耀来说，用户体量是相当大的，如果使用 TCP 连接的话，就可能会出现服务器不够用的情况，因为每台服务器可供支撑的 TCP 连接数量是有限制的。

再者，因为 TCP 会严格控制传输的正确性，如果因为用户网络条件不好就造成页面卡顿然后内存老旧的画面肯定是肯定不能接受的，毕竟对于这类实时性要求很高的游戏来说，最新的游戏画面才是最重要的，而不是老旧的画面，否则角色都不知道死多少次了。

小结

这一章节的内容就到这，因为 UDP 协议相对简单，所以内容并不是很多，但是下一章节会呈现很多关于 TCP 相关的内容，请大家做好准备。

最后总结一下这一章节的内容：

- UDP 相比 TCP 简单的多，不需要建立连接，不需要验证数据完整性，不需要流量控制，只会把要发的数据报文一股脑的发送出去
- 虽然 UDP 并没有 TCP 传输的保障，但是也能在很多实时性要求高的地方有所作为

留言

输入评论 (Enter按行, Ctrl + Enter发送)

发表评论

全部评论 (30)

刚刚回答 | 前端开发 | 3小时前
问题是 UDP 和 TCP 有什么区别?
◎ 点赞 0 回复 0

阿离离吧 | 2小时前
传输层协议有什么关系?
◎ 点赞 0 回复 0

hm496 | 1年前
文字有的图片没了。重新截图
◎ 点赞 0 回复 0

彭金鹏 | 国服第一黄逼 | 1年前
网速较慢，已经是逼你赶紧玩王者荣耀
◎ 点赞 0 回复 0

muzeng | web前端工程师 | 2年前
给大家举个不好说话的例子，后台数据库并不支持 UDP，但是后台数据库只是操作数据(比如插入，修改，删除等)。所以后台数据库是不能接受 UDP 的。所以 UDP 在这种情况下无用武之地，只会降低用户体验。
◎ 点赞 0 回复 0

铁蛋ironEggs | 2年前
这个是比较好的
◎ 点赞 0 回复 0

谷小鑫 | 2年前
牛皮
◎ 点赞 0 回复 0

查看更多回复 ▾

弹幕 | 2年前
王者荣耀这个不好说吧，我也没研究过，但是它后台数据库只是操作数据(比如插入，修改，删除等)，一直都是手机端通过接头实时查询的
◎ 点赞 0 回复 0

Orme小猪 | 1年前
这个传输数据的效率要好些吧，所以是不是会搞个 UDP 版本
◎ 点赞 0 回复 0

iPhone223 | 2年前
看了一下，直播间的 RTMP、RTSP 等网络协议似乎都是基于 TCP 的啊
◎ 点赞 0 回复 0

Server45742 | 2年前
——
◎ 点赞 0 回复 0

ZedCoding | 前端 | 杭州高级工程师 | 3年前
好像是这样
◎ 点赞 0 回复 0

承重墙右 | FE @ SF | 3年前
王者荣耀有的啊 😊
◎ 点赞 0 回复 0

yck | 作者 | 3年前
UDP 本来就没有东西可以说
◎ 点赞 0 回复 0

crackdown | 前端工程师 | 3年前
最后一句虽然 UDP 并没有 TCP 传输来的速度“快”但是性价比还是不错的吧。
◎ 点赞 0 回复 0

joman | 前端工程师 | 广州凡... | 3年前
小弟觉得虽然 UDP 并没有 TCP 传输来的速度“快”，但是性价比还是不错的吧。
◎ 点赞 0 回复 0

yck | 作者 | 3年前
准确
◎ 点赞 0 回复 0

NonXiaoSpace | 17zucye | 3年前
设计模式内部太多 写书卖的
◎ 点赞 0 回复 0

南笙梦 | 对啦
这是多长时间更新一次啊，感觉过了一个世纪
◎ 点赞 0 回复 0

周闻不爱闻 | 3年前
这本书还值得吗？
◎ 点赞 0 回复 0

chechecocomment_ | 前端 | 打杂 | 3年前
设计模式这个会是大头吧，期待作者详细讲解设计模式
◎ 点赞 0 回复 0

法令男 | 3年前
阅读作者下一个更新的章节是TCP吗？
◎ 点赞 0 回复 0

RedLan | 3年前
单独分了一章出来，但是内容也太水了吧
◎ 点赞 0 回复 0

HTTP/2 及 HTTP/3

这一章节我们将学习 HTTP/2 及 HTTP/3 的内容。

HTTP/2 很好地解决了当下最常用的 HTTP/1 存在的一些性能问题。只需要升级到该协议就可以减少很多之前需要做的性能优化工作。当然兼容问题以及如何优雅降级应该是业内还不普遍使用的原因之一。

虽然 HTTP/2 已经解决了很多问题，但是并不代表它已经是完美的了。HTTP/3 就是为了解决 HTTP/2 存在的一些问题而被设计出来的。

HTTP/2

HTTP/2 相比于 HTTP/1，可以说是大幅度提高了网页的性能。

在 HTTP/1 中，为了性能考虑，我们会引入雪崩图、将小图内联、使用多个域名等等的方式。这一切都是因为浏览器限制了同一个域名下的请求数量（Chrome 下一般是限制六个连接），当页面中同时请求很多资源时，队头阻塞（Head of line blocking）会导致在达到最大请求数量时，剩余的资源需要等待其他资源请求完成后才能发起请求。

在 HTTP/2 中引入了多路复用的技术。这个技术可以只通过一个 TCP 连接就可以传输所有的请求数据。多路复用很好的解决了浏览器限制同一个域名下的请求数量的问题，同时也间接更容易实现全连接墙。毕竟新开一个 TCP 连接都需要慢慢提升传输速度。

大家可以通过 [该链接](#) 感受下 HTTP/2 比 HTTP/1 到底快了多少。

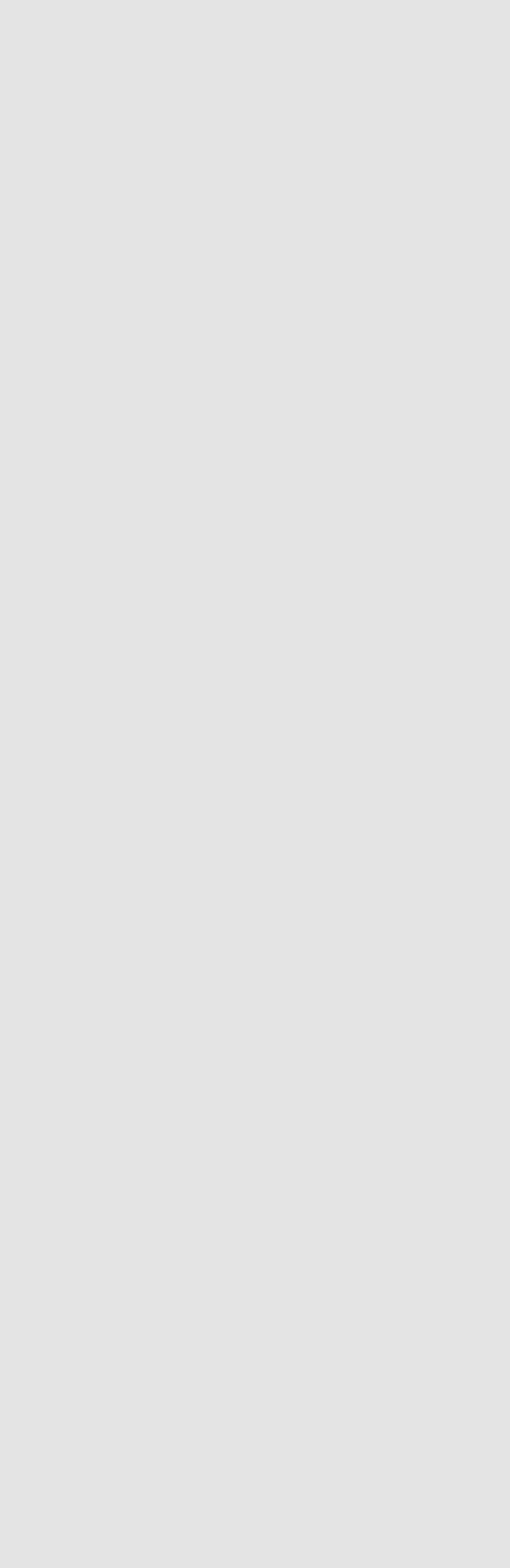


在 HTTP/1 中，因为队头阻塞的原因，你会发现发送请求是长这样的



二进制传输

HTTP/2 中所有加性能的核心点就在于此。在之前的 HTTP 版本中，我们是通过文本的方式传输数据。在 HTTP/2 中引入了新的编码机制，所有传输的数据都会被分割，并采用二进制格式编码。



多路复用

在 HTTP/2 中，有两个非常重要的概念，分别是帧（frame）和流（stream）。

帧代表最小的数据单位。每个帧会标识出该帧属于哪个流，流也就是多个帧组成的数据块。

多路复用，就是在一个 TCP 连接中可以存在多条流。换句话说，也就是可以发送多个请求，对端可以通过流的标识知道属于哪个请求。通过这个技术，可以避免 HTTP/1 版本中的队头阻塞问题，极大的提高传输性能。

那么可能就会有人考虑到修改 TCP 协议，其实这已经是一件不可能完成的任务了。因为 TCP 存在的时间实在太长，已经充斥在各种设备中，并且这个协议是由操作系统实现的，更新起来不太现实。

基于这个原因，Google 就更迫切地搞了一个基于 UDP 协议的 QUIC 协议，并且使用在了 HTTP/3 上，当然 HTTP/3 之前名为 HTTP-over-QUIC，从这个名字中我们可以发现，HTTP/3 最大的改动就是使用了 QUIC，接下来我们就来学习关于这个协议的内容。

QUIC

之前我们学习过 UDP 协议的内容，知道这个协议虽然效率很高，但是并不是那么的可靠。QUIC 虽然基于 UDP，但是在原本的基础上新增了很多功能，比如多路复用、0-RTT、使用 TLS 1.3 加密、流控控制、有序交付、重传等等功能。这里我们就挑选几个重要的功能了解一下这个协议的内容。

多路复用

虽然 HTTP/2 支持多路复用，但是 TCP 协议纯然是没有这个功能的。QUIC 原生就实现了这个功能，并且传输的单个数据流可以保证序交付且不会影响其他的数数据流，这样的技术就解决了之前 TCP 存在的问题。

并且 QUIC 在移动端的表现也会比 TCP 好。因为 TCP 是基于 IP 和端口去识别连接的，这种方式在多变的移动端网络环境下是很脆弱的。但是 QUIC 是通过 ID 的方式去识别一个连接，不管你网络环境如何变化，只要 ID 不变，就能迅速重建上。

0-RTT

通过使用类似 TCP 快速打开的技术，缓存当前会话的上下文，在下次恢复会话的时候，只需要将之前的缓存再送给服务端验证通过就可以进行传输了。

纠错机制

假如说这次我要发送三个包，那么协议会算出这三个包的校验值并单独发出一个校验包，也就是总共发出了四个包。

当出现其中的非校验包的情况时，可以通过另外三个包计算出丢失的数据包的内容。

当然这种技术只能使用在丢失一个包的情况下，如果出现丢失多个包就不能使用纠错机制了。只能使用重传的方式了。

小结

总结一下内容：

- HTTP/2 通过多路复用、二进制流、Header 压缩等等技术，极大地提高了性能，但是还是存在看问题的
- HTTP/2 基于 UDP 实现，是 HTTP/3 中的底层支撑协议，该协议基于 UDP，又取了 TCP 中的精华，实现了即快又可靠的协议

留言

输入评论 (Enter执行, Ctrl + Enter发表)

发表评论

全部评论 (9)

程序员升职记 | 前端研发工程师 @ 零 - 1年前
多路复用的就是怎么在工作中使用的
@点我 1

大象一枚 | 1年前
我也想问这个问题
@点我 1

Kaim | IE | 2年前
服务器端支持HTTP/1还是HTTP/2的么?
@点我 2

铁汉雄 | 1年前
我也想知道这个时间
@点我 1

金博零售布衫 | 1年前
静态服务器端可以配置，比如nginx
@点我 1

腾讯水西哥 | 2年前
“同时，我必须实现全速传输”这句话感觉不太通顺。
@点我 1

易水寒YSP | 3年前
更详细
@点我 1

FatDong1 | 3年前
nice
@点我 1

zy_ch | IE @ 东阳兄弟 | 3年前
真好
@点我 1

输入 URL 到页面渲染的整个流程

之前我们学了那么多章节的内容，是时候找个时间将它们再次复习消化了。就借用这道经典面试题，将之前学习到的知识以及网页几章节的知识联系起来。

首先是 DNS 查询。如果这一步做了智能 DNS 解析的话，会提供访问速度最快的 IP 地址回来，这部分的内容之前没有写过，所以就在这里讲一下。

DNS

DNS 的作用就是通过域名查询到具体的 IP。

因为 IP 存在数字和英文的组合 (IPv6)，很不利于人类记忆，所以就出现了域名。你可以把域名或是某个 IP 联系起来，DNS 就是去查询这个域名的真正名称是什么。

在 TCP 握手之前就已经进行了 DNS 查询，这个查询是操作系统自己做的。当你在浏览器中想访问 www.google.com 时，会进行一下操作：

1. 操作系统首先会在本地缓存中查询 IP
2. 没有的话去系统配置的 DNS 服务器中查询
3. 如果这时候还没得话，会直接去 DNS 根服务器查询，这一步查询会找出负责 .com 这个一级域名的服务器
4. 然后去该服务器查询 www.google.com 这个二级域名
5. 接下来三级域名的查询其实是我们配置的。你可以给 www 这个域名配置一个 IP，然后还可以给其他的三级域名配置一个 IP

以上介绍的是 DNS 迭代查询，还有种是递归查询，区别是前者是由客户端向云端请求，后者是由系统配置的 DNS 服务器做请求，得到结果后将数据返回给客户端。

PS：DNS 是基于 UDP 做的查询，大家也可以考虑下为什么之前不考虑使用 TCP 去实现。

接下来是 TCP 握手，应用层会下发数据给传输层，这里 TCP 协议会指明两端的端口号，然后下发给网络层，网络层中的 IP 协议会确定 IP 地址，并且指示数据传输中如何跳转路由器，然后就会再被代理到数据链路层的数据帧结构中，最后就是物理层的传输了。

在这一部分中，可以详细了解下 TCP 的握手情况以及 TCP 的一些特性。

当 TCP 握手结束后就会进行 TLS 握手，然后就开始正式的数据传输。

在这一部分中，可以详细了解下 TLS 的握手情况以及两种加密方式的内容。

数据在进入服务器之前，可能还会先经过负载均衡的服务器，它的作用就是将请求合理的分发到多台服务器上，这时假设服务器响应一个 HTML 文件。

首先浏览器会判断状态是什么，如果是 400 那就继续解析，如果是 500 或 500 的话就会报错，如果 300 的话会进行重定向，这里会有个重定向计数器，避免过多次数的重定向，超过次数就会报错。

浏览器开始解析文件，如果是 gzip 格式的话会先解压一下，然后通过文件的编码格式知道该如何去解码文件。

文件解码成功后会正式开始渲染流程。先会根据 HTML 构建 DOM 树，有 CSS 的话再去构建 CSSOM 树，如果遇到 script 标签的话，会判断是否存在 async 或者 defer，前者会并行进行下载并执行 JS，后者会先下载文件，然后等待 HTML 解析完成后顺序执行。

如果以上都没有，就会阻塞住渲染流程直到 JS 执行完毕。遇到文件下载的话会去下载文件，这里如果使用 HTTP/2 协议的话会极大的提高多图的下载效率。

CSSOM 树和 DOM 树构建完成后会开始生成 Render 树，这一步就是确定页面元素的布局，样式等等诸多方面的东西。

在生成 Render 树的过程中，浏览器就开始调用 GPU 绘制，合成图层，将内容显示在屏幕上。

这一部分就是渲染原理中讲解到的内容，可以详细的说明下这一过程。并且在下载文件时，也可以通过 HTTP/2 协议来解决头部阻塞的问题。

总的来说这一章节就是带大家从 DNS 查询开始到渲染出画面完整的了解一遍过程，将之前学习到的内容连接起来。

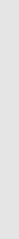
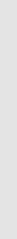
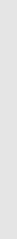
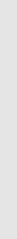
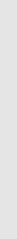
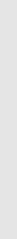
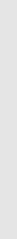
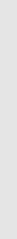
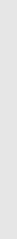
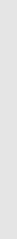
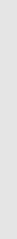
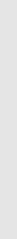
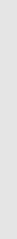
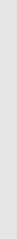
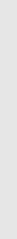
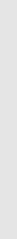
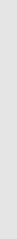
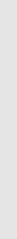
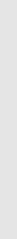
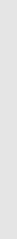
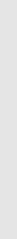
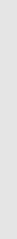
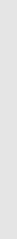
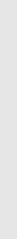
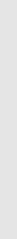
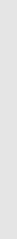
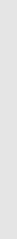
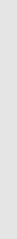
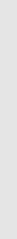
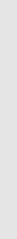
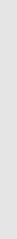
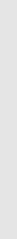
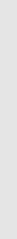
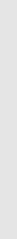
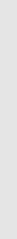
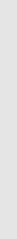
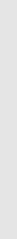
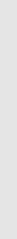
当来这一过程远远不止这些内容，但是对于大部分人能答出这些内容已经很不错了，你如果想了解更多详细的过程，可以阅读这篇文章。

留言

输入评论 (Enter执行, Ctrl + Enter发表)

发表评论

全部评论 (19)

- 砂糖小刀  | 前端 | 3年前
良哥恩这操小破孩是用奇葩式的，又不是百科全书各种知识点都给你讲那么深，就是单纯的让你自己去理解掌握然后举些自身情况解释一下而已，说白了小破孩连打酱油都行，作者也说过这操小破孩是对中小型公司的，中小型企业你摆在这里说的这些已经完全够了，简单点，太简单，那说明你的水平和目标都是月薪上方的一线大厂了，那就还来看这种的破小孩？就这么说，亲来一本初中数学辅导书里都讲的明白还觉得搞笑吗？
- 白4 
- 下海天DY  | 前端开发工程师 @ 小米 | 8月前
太水了
白1 
- 用户664691587...  | 前端开发 @ 美团 | 8月前
真实后面越来越好
白1 
- 阿婆  | 前端 | 9月前
说实话，后面整理的这些真的越来越好了，既然决定了写小破孩要对内负责，这样就只是坏了自己名声的罢了
白1 
- 最美大魔王  | 前端小菜鸡 @ 负责 | 1年前
没必要编，全部写一下，前面这么具体，后面一下就没了，虎头蛇尾都算不上。
白1 
- 你老像风  | 前端开发工程师 | 1年前
讲得挺好，能把这些讲清楚已经很好了。
白1 
- less  | 前端开发工程师 | 1年前
后面整理的这些真的越来越好了一一
白1 
- Lstorie-e君  | 1年前
DNS 是基于 UDP 做的查询，大家也可以考虑下为什么之前不考虑使用 TCP 去实现。现在也有些使用 TCP
白1 
- toored  | 1年前
单理性，属实大差了，感觉很应付，重头也不突出
白1 
- hxtongjun  | 前端开发工程师 | 1年前
整理
白1 
- 别来无恙  | 2年前
AST 这个在什么时候出现的
白1 
- jhongjun  | 前端开发 | 2年前
具体又不全面。
白1 
- Genius同学  | 2年前
就不能先把语言写出来再讲解吗 比如洒洒一大堆也不知道面试的时候怎么回答都比较好
白1 
- 字节串  | 前端开发 | 2年前
总感觉有东西别一下更好，总感觉像是读作文。单理性不是很重要。
白1 
- RandomKing  | 字节跳动 | 2年前
这一过程涉及浏览器缓存机制。好像没有看到。
白1 
- 泉水亭555  | 3年前
手动点赞~
白1 
- 半夜盗贼  | 3年前
这个写得太一致了。美观的都有，深入的却没有。只起了一个指导意义的作用
白1 
- 三只猪新  | 前端 | 3年前
很好
白1 
- MrGack  | 前端工程师 @ 阿里云... | 3年前
最后一段两个字“当然”
白1 

这一章节

```
    alertName() {
        alert(this.name)
    }
}

class Factory {
    static create(name) {
        return new Person(name)
    }
}

Factory.create('yuki').alertName()
```

在 Vue 源码中，你也可以看到工厂模式的影子。在 `Vue.extend` 方法中，如果参数是对象，那么会调用 `Object.create` 方法，将参数对象作为原型，从而实现对原有构造函数的扩展。这种实现方式，与工厂模式非常相似。

102

在上述代码中，我们可以看到我们只需要调用 `createComponent` 传入参数就足够了。但是创建这个实例是很复杂的一个过程，工厂帮助我们隐藏了这个复杂的逻辑，调用的就能实现功能。

单例模式的核心就是保证全局只有一个对象可以访问。因为 JS 是一门无类的语言，所以别的语言实现单例的方式并不能套入 JS 中。我们只需要用一个变量确保实例只创建一次就行。以下是如何实现单例模式的例子

```
class Singleton {
    constructor() {}
}

Singleton.getInstance = (Function) {
    let instance
    return Function() {
        if (!instance) {
            instance = new Singleton()
        }
        return instance
    }
}(function()

let c1 = Singleton.getInstance()
let c2 = Singleton.getInstance()
console.log(c1 === c2) // true
```

在 Vuex 源码中，你也可以看到单例模式的使用。虽然它的实现方式不大一样，通过一个外部变量来控制只安装一次 Vuex。

```
let Vue // bind on install

export function install (_Vue) {
    if (_Vue !== Vue) {
        // 如果发现 _Vue 有值，就不重新创建实例了
        return
    }
    Vue = _Vue
    applyMixin(Vue)
}
```

适配器模式

适配器用来解决两个接口不兼容的情况。不需要改变已有的接口。通过包装一层的方式实现两个接口的正常协作。

以下是如何实现适配器模式的例子

```
class Plug {
    getName() {
        return '连接插头'
    }
}

class Target {
    constructor() {
        this.plug = new Plug()
    }
    getName() {
        return this.plug.getName() + ' 适配器第二层插头'
    }
}
```

内部需要将时间戳转为正常的日期显示。一般会使用 `computed` 来做转换这件事情，这个过程就使用到了适配器模式。

```
name = "yck"
}

let t = new Test()

t.yck = "nnn" // 不可修改
```

代理是为了控制对对象的访问，不让外部直接访问到对象。在现实生活中，也有很多代理的场景，比如你需要买一件国外的产品，这时候你可以通过代购来购买产品。

在实际代码中其实代理的场景很多，也就不举框架中的例子了，比如事件代理就用到了代理模式。

因为
节点

```
<ul id="ul"></ul>
<script>
    let ul = document.querySelector('#ul')
    ul.addEventListener('click', (event) => {
        console.log(event.target);
    })
</script>
```

外观模式提供了一个接口，隐藏了内部的逻辑，更加方便外部调用。

举个例子来说，我们现在需要实现一个兼容多种浏览器的添加事件方法

```
function addEvent(element, evtype, fn, useCapture) {
    if (element.addEventListener) {
        element.addEventListener(evtype, fn, useCapture)
        return true
    } else if (element.attachEvent) {
        var e = element.attachEvent("on" + evtype, fn)
        return e
    } else {
        element["on" + evtype] = fn
    }
}
```

这一章节我们学习了几种常用的设计模式。其实设计模式还有很多，有一些内容很简单，我就没有写在章节中了，比如迭代器模式、原型模式。有一些内容也是不经常使用，所以也就不一一列举了。

如果你还想了解更多关于设计模式的内容，可以阅读[这本书](#)。

[发表评论](#)

全部评论 (32)

于先生  | 前端工程师 @ 英伟达 | 9月前

在发布-订阅模式版本小程序中明确规定到了观察者模式和订阅-发布模式相对区别在于是否有第三方介入，发布-订阅模式需要第三方去维持触发和注册组件所属事件，而观察者由发布者维持订阅关系。

 2 

理想永远不会轻  | 前端 @ 去哪儿网 | 11月前

发布-订阅模式和观察者模式不一样。

 1 

FruitBro  | 前端开发工程师 | 1年前

装饰模式，但在 React 中的例子，connect不是装饰函数么。好像没有涉及到装饰模式。

 2 

你若像风  | 前端开发工程师 | 1年前

讲得很好，了解了很多。

 sky | Web Developer @ 第一 | 1年前
太水了
 回复

 明天下雨不想说话 | 1年前
太垃圾了
 回复

 chiuwingyan  | 前端工程师 | 1年前
发布幻向者模式和观察者模式只能说是类似，但是不是一样的
 回复

 霍星 star | 2年前

正本看下来，太垃圾了
 11 回复

kingli | web前端 | 2年前
使用外联模式在增加子系统的時候修改外观类内部，违反开放封闭原则，具体是否使用还是要结合场景。
 点赞 回复

wenting68456 | 2年前
适配器模式不是很懂，太抽象了
 1 回复

sammui4 | 切图仔 | 2年前
居然没有策略模式。。。

 猪泡泡大 | 2年前
应该是一回事
[点我](#) [回复](#)

 inclosing  | 2年前
不是一回事吧
[点我](#) [回复](#)

[查看更多回答](#) 

那为什么要通过API来创建一个全局对象？这样返回的函数能被当作全局对象。拥有对这个全局对象中instance的访问权限？

 点赞 回复

1  回答 想想看611109 | 2年前
你说的是对的，其实目的就是不想创建一个全局变量instance而已
“那因为要通过API来创建一个作用域吗？这样返回的函数就会形成闭包。拥有对这个...”

 点赞  回复

 zenzplus | 3年前

本来应该绑定在元素上的事件，代理到了父元素以上。这一点是代理模式的体现。而对dom事件的监听 (addEventListener) 是发布订阅模式的体现。这段代码两个模式都有体现。

 点赞 | 回复

 查看更多回复 ▾

常见数据结构

这一章节我们来学习数据结构的内容。经常会有人提问题：学习数据结构或者算法对于前端工程师有什么用？

总的来说，这些基础学科在短期内收获确实甚微，但是我们首先不要将自己局限在前端工程师这点上。笔者之前是做 iOS 开发的，转做前端之后，只有两个技能还对我有用：

- 1. 基础学科内容，比如：网络知识、数据结构、算法
- 2. 编程思维

其他 iOS 上积累的经验，转行以后基本就没多大用处了。所以说，当我们把视野放到编程这个角度去说，数据结构算法一定是有帮助的，并且也是你未来的一个天花板。可以不花时间集中时间去学习这些内容，但是一定需要时常去学习一点，因为这些技能可以实实在在提升你写代码的能力。

这一章节的内容信息量会很大，不适合在非电脑环境下阅读。请各位打开代码阅读器，一行一行的敲代码。单纯阅读是学不会数据结构的。

时间复杂度

在进入正题之前，我们先来了解下什么是时间复杂度。

通常使用最差的时间复杂度衡量一个算法的好坏。

常数时间 $O(1)$ 代表这个操作和数据量没关系，是一个固定时间的操作，比如类的构造。

对于一个操作来说，可能会计算出操作次数为 $aN + b$ ， N 代表数据量。那么该算法的时间复杂度就是 $O(N)$ 级别。我们在计算时间复杂度的时候，数据量通常非常大的，这时高低阶常数项就不计了。

当然可能会出现两个算法都是 $O(N)$ 的时间复杂度，那么对比两个算法的好坏就要通过对比低阶项的数据项了。

栈

概念

栈是一个线性结构，在计算机中是一个相当常见的数据结构。

栈的特点只能在一端添加或删除数据，遵循先进后出的原则。

Push: A blue rectangle labeled "Push" has an arrow pointing down to a stack of four blue rectangles. Pop: A blue rectangle labeled "Pop" has an arrow pointing up from the same stack.

实现

每种数据结构都可以用很多种方式来实现，其实可以把栈看成数组的一个子集。所以这里使用数组来实现。

```
class Stack {
    constructor() {
        this.stack = []
    }
    push(item) {
        this.stack.push(item)
    }
    pop() {
        this.stack.pop()
    }
    peek() {
        return this.stack[this.stack.length - 1]
    }
    getLength() {
        return this.stack.length
    }
    isEmpty() {
        return this.stack.length === 0
    }
}
```

@雷生技术社区

应用

选取了 [LeetCode 上序号为 20 的题目](#)

题目是匹配括号，可以通过栈的特性来完成这道题目

```
var isValid = function(s) {
    let map = {
        '(': ')',
        '[': ']',
        '{': '}'
    }
    let stack = []
    for (let i = 0; i < s.length; i++) {
        if (map[s[i]] >= 0) {
            stack.push(s[i])
        } else {
            let last = stack.pop()
            if (map[last] != map[s[i]]) return false
        }
    }
    if (stack.length > 0) return false
    return true
}
```

其实在 Vue 中关于模板解析的代码，就有应用到匹配括号的内容。

队列

概念

队列是一个线性结构，特点是在某一端添加数据，在另一端删除数据，遵循先进先出的原则。

实现

这里会讲解两种实现队列的方式，分别是单链队列和循环队列。

单链队列

```
class Queue {
    constructor() {
        this.queue = []
    }
    enqueue(item) {
        this.queue.push(item)
    }
    dequeue() {
        return this.queue.shift()
    }
    peek() {
        return this.queue[0]
    }
    getLength() {
        return this.queue.length
    }
    isEmpty() {
        return this.queue.length === 0
    }
}
```

因为单链队列在出队操作的时候需要 $O(n)$ 的时间复杂度，所以引入了循环队列。循环队列的出队操作平均是 $O(1)$ 的时间复杂度。

循环队列

```
class Queue {
    constructor(length) {
        this.queue = new Array(length + 1)
        // 队头
        this.first = 0
        // 队尾
        this.last = 0
        // 当前队列大小
        this.size = 0
    }
    enqueue(item) {
        // 例程 i = 1 是指向队头
        // 例程 i = 2 是指向队尾
        // i = 3, queue.length = 3 是指向队列的尾部
        if ((this.last + 1) % this.queue.length) {
            this.queue[this.last + 1] = item
        }
        this.last += 1
        this.size++
    }
    dequeue() {
        if (this.size === 0) {
            throw Error('Queue is empty')
        }
        let item = this.queue[this.first]
        if (this.size === 1) {
            this.queue[this.first] = null
            this.last = this.first
            this.size = 0
        }
        this.first++
        this.size--
    }
    peek() {
        if (this.size === 0) {
            throw Error('Queue is empty')
        }
        return this.queue[this.first]
    }
    getLength() {
        return this.queue.length - 1
    }
    isEmpty() {
        return this.size === 0
    }
}
```

以上是最基本的二分搜索树实现，接下来实现树的遍历。

对于树的遍历来说，有三种遍历方法，分别是先序遍历、中序遍历、后序遍历。三种遍历的区别在于何时访问节点。在遍历树的过程中，每个节点都会遍历三次，分别是遍历到自己，遍历左子树遍历右子树。如果想要实现先序遍历，那么只需要第一次遍历到节点时进行操作即可。

```
// 先序遍历：用于构建树的结构
// 先序遍历的根节点，然后递归左节点，最后访问右节点。
preorderTraversal() {
    this._preOrder(this.root)
    this._preOrder(this.root)
}
_preOrder() {
    if (node === null) return
    console.log(node.value)
    this._preOrder(node.left)
    this._preOrder(node.right)
}

```

对于中序遍历来说，遍历到左节点时，需要比较当前节点值和左子树的值，当需要直接找左子树的值时，所以只需要在根节点的右子树上寻找。大大提高了搜索效率。

首先对节点的左子节点进行一次遍历，接着对右子节点进行一次遍历，这样就可以实现树的遍历。

```
_inOrder() {
    let node = this._inOrder(this.root)
    return node._inOrder()
}
_inOrder() {
    if (node === null) return
    this._inOrder(node.left)
    console.log(node.value)
    this._inOrder(node.right)
}

```

对于后序遍历来说，相反于左子节点，所以不再赘述。

对于右子节点来说，新增加的节点位于节点 4 的右侧。对于这种情况，需要通过每次遍历来达到目的。

首先对节点的左子节点左子节点进行一次遍历，再对节点进行一次右子节点的遍历。

```
_postOrder() {
    let node = this._postOrder(this.root)
    return node._postOrder()
}
_postOrder() {
    if (node === null) return
    this._postOrder(node.left)
    this._postOrder(node.right)
    console.log(node.value)
}

```

以上的这几种遍历都可以称之为深度遍历，对应的还有种遍历叫做广度遍历，也就是一层层地遍历树。对于广度遍历来说，我们需要利用之前讲过的队列结构来完成。

```
breadthTraversal() {
    if (!this.root) return null
    let q = new Queue()
    q.enqueue(this.root)
    q.dequeue()
    q.enqueue(this.root.left)
    q.enqueue(this.root.right)
    let size = q.size()
    while (size) {
        let node = q.dequeue()
        console.log(node.value)
        if (node.left) {
            q.enqueue(node.left)
        }
        if (node.right) {
            q.enqueue(node.right)
        }
        size--
    }
}

```

接下来讲解的是二分搜索树中最难实现的部分：删除节点。因为对于删除来说，会存在以下几种情况：

- 需要删除的节点没有子树
- 需要删除的节点只有一条子树
- 需要删除的节点有左右两条子树

对于前两种情况来说，比较简单。但是第三种情况就有难度了，所以先来实现相对简单的操作：删除最小节点。对于删除最小节点来说，是不存在第三种情况的。删除最大节点操作是和删除最小节点相对称的，所以这里也就不再赘述。

```
deleteMin() {
    this.root = this._deleteMin(this.root)
    console.log(this.root)
}
_deleteMin(node) {
    // 一直遍历到叶子结点
    if (node.left === null) {
        this._rightDelete(node)
        return node
    }
    if (node.left !== null) {
        node.left = this._deleteMin(node.left)
    }
    return node
}

```

最后讲解的就是如何删除任意节点了。对于这个操作，T.Hibbard 在 1962 年提出了解决这个难题的办法。也就是如何处理三种情况。

当遇到这种情况时，需要取出当前节点的后继节点（也就是当前节点右子树的最小节点）来替换需要删除的节点。然后将后继节点的右子树替换成当前节点的右子树。右子树删除后链接给左子树，左子树的右子树接替右子树。

你如果对这个操作办法疑惑的话，可以这样考虑。因为二分搜索树的特性，父节点一定比所有左子节点大，比所有右子节点小，那么删除父节点，势必需要拿出一个比父节点大的节点来替换父节点。那么节点的值不再大于等于它的左子树，从而判断出左子树的最小节点。如果有的话，继续上面的递归判断。

```
_select(v) {
    let node = this._select(this.root, v)
    return node._getMin()
}
_getMin(node) {
    if (node.left === null) {
        this._rightDelete(node)
        return node
    }
    if (node.left !== null) {
        node.left = this._getMin(node.left)
    }
    return node
}

```

对于左子节点来说，新增加的节点位于节点 2 的左侧，这时树已经不平衡，需要旋转。因为搜索树的特性，节点值比左子节点大，比右子节点小，所以旋转后也要实现这种特性。

旋转之前：new < 2 < 3 < B < 5 < A，旋转之后节点 3 为根节点。这时需要将节点的值比根节点的值大，所以还需要更新根节点的高度。

对于右子节点来说，新增加的节点位于节点 4 的右侧。对于这种情况，需要通过每次旋转来达到目的。

首先对节点的左子节点左子节点进行一次遍历，再对节点进行一次右子节点的遍历。

```
_rotateLeft() {
    let node = this._rotateLeft(this.root)
    return node._rotateRight()
}
_rotateLeft(node) {
    if (node.right === null) return
    let right = node.right
    node.right = right.left
    right.left = node
    return right
}

```

将右子节点的值赋给左子节点，然后将左子节点的值赋给右子节点，最后将右子节点的值赋给原父节点。

```
_rotateRight() {
    let node = this._rotateRight(this.root)
    return node._rotateLeft()
}
_rotateRight(node) {
    if (node.left === null) return
    let left = node.left
    node.left = left.right
    left.right = node
    return left
}

```

对于右子节点来说，相反于左子节点，所以不再赘述。

对于右子节点来说，新增加的节点位于节点 4 的右侧。对于这种情况，需要通过每次遍历来达到目的。

首先对节点的右子节点右子节点进行一次遍历，再对节点进行一次左子节点的遍历。

```
_rotateRight() {
    let node = this._rotateRight(this.root)
    return node._rotateLeft()
}
_rotateRight(node) {
    if (node.left === null) return
    let left = node.left
    node.left = left.right
    left.right = node
    return left
}

```

将左子节点的值赋给右子节点，然后将右子节点的值赋给左子节点，最后将左子节点的值赋给原父节点。

```
_rotateLeft() {
    let node = this._rotateLeft(this.root)
    return node._rotateRight()
}
_rotateLeft(node) {
    if (node.right === null) return
    let right = node.right
    node.right = right.left
    right.left = node
    return right
}

```

将右子节点的值赋给左子节点，然后将左子节点的值赋给右子节点，最后将右子节点的值赋给原父节点。

```
_select(v) {
    let node = this._select(this.root, v)
    return node._getMax()
}
_getMax(node) {
    if (node.right === null) {
        this._leftDelete(node)
        return node
    }
    if (node.right !== null) {
        node.right = this._getMax(node.right)
    }
    return node
}

```

对于右子节点来说，新增加的节点位于节点 4 的右侧。对于这种情况，需要通过每次遍历来达到目的。

```
_rotateRight() {
    let node = this._rotateRight(this.root)
    return node._rotateLeft()
}
_rotateRight(node) {
    if (node.left === null) return
    let left = node.left
    node.left = left.right
    left.right = node
    return left
}

```

将左子节点的值赋给右子节点，然后将右子节点的值赋给左子节点，最后将左子节点的值赋给原父节点。

```
_rotateLeft() {
    let node = this._rotateLeft(this.root)
    return node._rotateRight()
}
_rotateLeft(node) {
    if (node.right === null) return
    let right = node.right
    node.right = right.left
    right.left = node
    return right
}

```

将右子节点的值赋给左子节点，然后将左子节点的值赋给右子节点，最后将右子节点的值赋给原父节点。

```
_select(v) {
    let node = this._select(this.root, v)
    return node._getMin()
}
_getMin(node) {
    if (node.left === null) {
        this._rightDelete(node)
        return node
    }
    if (node.left !== null) {
        node.left = this._getMin(node.left)
    }
    return node
}

```

对于左子节点来说，新增加的节点位于节点 2 的左侧，这时树已经不平衡，需要旋转。因为搜索树的特性，节点值比左子节点大，比右子节点小，所以旋转后也要实现这种特性。

旋转之前：new < 2 < 3 < B < 5 < A，旋转之后节点 2 为根节点。这时需要将节点的值比根节点的值大，所以还需要更新根节点的高度。

```
_rotateRight() {
    let node = this._rotateRight(this.root)
    return node._rotateLeft()
}
_rotateRight(node) {
    if (node.left === null) return
    let left = node.left
    node.left = left.right
    left.right = node
    return left
}

```

将右子节点的值赋给左子节点，然后将左子节点的值赋给右子节点，最后将右子节点的值赋给原父节点。

```
_rotateLeft() {
    let node = this._rotateLeft(this.root)
    return node._rotateRight()
}
_rotateLeft(node) {
    if (node.right === null) return
    let right = node.right
    node.right = right.left
    right.left = node
    return right
}

```

将左子节点的值赋给右子节点，然后将右子节点的值赋给左子节点，最后将左子节点的值赋给原父节点。

```
_select(v) {
    let node = this._select(this.root, v)
    return node._getMax()
}
_getMax(node) {
    if (node.right === null) {
        this._leftDelete(node)
        return node
    }
    if (node.right !== null) {
        node.right = this._getMax(node.right)
    }
    return node
}

```

对于右子节点来说，新增加的节点位于节点 4 的右侧。对于这种情况，需要通过每次遍历来达到目的。

```
_rotateLeft() {
    let node = this._rotateLeft(this.root)
    return node._rotateRight()
}
_rotateLeft(node) {
    if (node.right === null) return
    let right = node.right
    node.right = right.left
    right.left = node
    return right
}

```

将左子节点的值赋给右子节点，然后将右子节点的值赋给左子节点，最后将左子节点的值赋给原父节点。

```
_select(v) {
    let node = this._select(this.root, v)
    return node._getMax()
}
_getMax(node) {
    if (node.right === null) {
        this._leftDelete(node)
        return node
    }
    if (node.right !== null) {
        node.right = this._getMax(node.right)
    }
    return node
}

```

对于右子节点来说，新增加的节点位于节点 4 的右侧。对于这种情况，需要通过每次遍历来达到目的。

```
_rotateRight() {
    let node = this._rotateRight(this.root)
    return node._rotateLeft()
}
_rotateRight(node) {
    if (node.left === null) return
    let left = node.left
    node.left = left.right
    left.right = node
    return left
}

```

将右子节点的值赋给左子节点，然后将左子节点的值赋给右子节点，最后将右子节点的值赋给原父节点。

```
_select(v) {
    let node = this._select(this.root, v)
    return node._getMin()
}
_getMin(node) {
    if (node.left === null) {
        this._rightDelete(node)
        return node
    }
    if (node.left !== null) {
        node.left = this._getMin(node.left)
    }
    return node
}

```

对于左子节点来说，新增加的节点位于节点 2 的左侧，这时树已经不平衡，需要旋转。因为搜索树的特性，节点值比左子节点大，比右子节点小，所以旋转后也要实现这种特性。

旋转之前：new < 2 < 3 < B < 5 < A，旋转之后节点 2 为根节点。这时需要将节点的值比根节点的值大，所以还需要更新根节点的高度。

```
_rotateLeft() {
    let node = this._rotateLeft(this.root)
    return node._rotateRight()
}
_rotateLeft(node) {
    if (node.right === null) return
    let right = node.right
    node.right = right.left
    right.left = node
    return right
}

```

将左子节点的值赋给右子节点，然后将右子节点的值赋给左子节点，最后将右子节点的值赋给原父节点。

```
_select(v) {
    let node = this._select(this.root, v)
    return node._getMax()
}
_getMax(node) {
    if (node.right === null) {
        this._leftDelete(node)
        return node
    }
    if (node.right !== null) {
        node.right = this._getMax(node.right)
    }
    return node
}

```

对于右子节点来说，新增加的节点位于节点 4 的右侧。对于这种情况，需要通过每次遍历来达到目的。

```
_rotateRight() {
    let node = this._rotateRight(this.root)
    return node._rotateLeft()
}
_rotateRight(node) {
    if (node.left === null) return
    let left = node.left
    node.left = left.right
    left.right = node
    return left
}

```

将右子

CSS 常考面试题资料

其实笔者在面试的时候这方面的内容完全没有被问到，并且自己也基本没有准备这一部分的内容。

但是鉴于小白面向的群体是大众，肯定会有人被问到这方面的内容，因此我在这一章节会总结一些面试资料给大家，就不班门弄斧了。

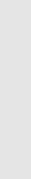
- [50道CSS基础面试题（附答案）](#)
- [《50道CSS基础面试题（附答案）》中的答案真的就只是答案吗？](#)
- [CSS 面试题总结](#)
- [front-end-interview-handbook](#)

留言

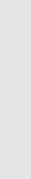
 输入评论 (Enter执行, Ctrl + Enter发送)

发表评论

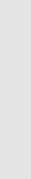
全部评论 (31)

 他的一片海 | 三级前端工 @ 小学生 | 1月前
大佬话 还是海哥口碑好的作者

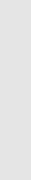
点赞 回复

 RollinLee | 软件开发工程师 @ CIB | 2月前
付费，买的 就这样 非常恶心

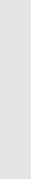
点赞 回复

 跟白痴傻逼 | 9月前
第三个链接打不开了

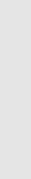
点赞 回复

 卡夫卡 | 前端 @ 未知 | 10月前
真心推荐你们买挺全另一个介绍css的小册子有所值

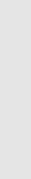
点赞 回复

 liDenggy | 前端开发 | 10月前
直接放几个链接是啥意思呀 😂

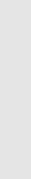
点赞 回复

 jz-ding | web前端 | 1年前
后面两个链接都打不开了

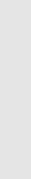
点赞 回复

 AmylyG | 前端 | 1年前
链接都打不开了

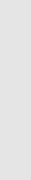
点赞 回复

 前端广成子 | web前端开发 @ 霍比特... | 1年前
前面两个链接失效了，那这一章岂不是GG

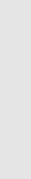
点赞 回复

 helion | 前端 @ tencent | 1年前
前面两个链接失效了，你还是继续写

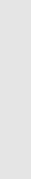
点赞 回复

 不二子 | 2年前
前端怎么可能不问css，作者收录了这样不太好吧

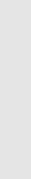
点赞 回复

 赵海峰 | 1年前
或许这就是强者的世界吧

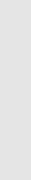
点赞 回复

 楼上一片海 | 回复 | 赵海峰 | 1年前
强者把每一张都贴链接就行了

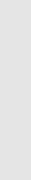
点赞 回复

 "或许这就是强者的世界吧"

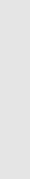
点赞 回复

 我是 | 2年前
这也太敷衍了吧

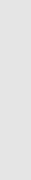
点赞 回复

 ryu_permit | 前端 | 2年前
funtrees.com

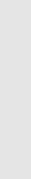
点赞 回复

就是一个春天的花朵  | 深圳前端工程师 | 2年前
阅读这样不好吧。这是收费的小册子。结果里面有一章连专门放别人的链接，这一章直接用放弃了

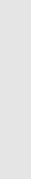
点赞 回复

 wenting66456 | 2年前
这几个链接像很多是重叠?

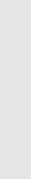
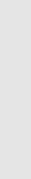
点赞 回复

 sophie姐 | 2年前
水平居中，还有水平面居中各种方法文章：如何居中一个元素 (终级版)

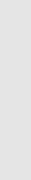
点赞 回复

 Server45742 | 2年前
——

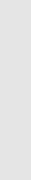
点赞 回复

 善天  | web @ Encompass | 2年前
前面两个链接失效了，你还是继续写

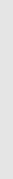
点赞 回复

 Z>华 | Web Fullstack dev | 2年前
前面两个链接失效了，你还是继续写

点赞 回复

 hanmin | 艺人 | 3年前
css部分删掉几个链接吗?

点赞 回复

查看全部 31 条回复 

如何写好一封简历

简历不是一沓记流水账的东西，而是让用人方了解你的亮点的。

平时有在做修改简历的收费服务，也算看过蛮多简历了，很多简历都有以下几个特征：

- 喜欢说自己的特长、优点，用人方真的不关注你的性格是怎么样等等
- 喜欢列举一大堆个人技能，生怕用人方不知道你会些什么。造成的结果就是好多简历的技能都是差不多
- 项目经验流水账，比如使用了什么框架，什么 API 做了什么业务
- 简历页面过多

以上类似简历可以说用人方也看了无数份，完全瞧不到你的亮点。

简历其实就是推销自己，如果你的简历和别人千篇一律，没有亮点，用人方就不会对你产生兴趣。

以下是我经常给别人修改简历的意见：

- 简历页数控制在 2 页以下
- 技术名词注意大小写
- 突出个人亮点，比如在项目中如何找到 Bug，解决 Bug 的过程；比如如何发现的性能问题，如何提升了多少性能；比如为何如此选型，目的为什么，获得了什么优点等等。总体思路就是不要流水账，突出你在项目中具有不错的解决问题的能力和独立思考能力
- 斟酌一些可能引起歧义的字眼，不要自己坑自己
- 确保每一个写上去的技术点自己都能说出点什么，杜绝面试官问你一个技术点，你只能答出会用 API 这种泛泛的情况
- 拿事实说话，你说你学习能力强，那么请列举你能力强的事实；你说你成绩好，那么请写出你得过的排名

做到以上内容，然后在投递简历的过程中加上一份求职信，对你的求职之路相信能上很多忙，当然了，一般来说我推荐尽量走内部通道投递简历，在网上多花点心思就能找到很多内推，比如 V2EX，脉脉等等。

说了这么多，我们还是实战来修改一封简历吧，该简历的作者是一名两年经验的前端开发，关键信息已经全部遮去。

1. 相关职位

web 前端工程师 | 深圳 | 12k-15k | 一个月内面试

2. 教育经历

华南理工大学 | 本科 (连续二次获得奖学金)

2012.09 - 2016.07

就读专业：软件工程
在校期间：担任班级团支书

这算是简历最先被别人看到的地方，最黄金的位置必然要放自己最重要的东西。

比如你是 985、211 毕业的，有不错的成绩、专业排名都可以在这个位置展现出来。但是如果你学历并不那么好，可以考虑把教育经历移到简历的最后，尽量把这块黄金位置让出来。

然后个人优势这块，一般来说就是个人技能，首先杜绝任何精雕细琢的字眼，因为百分之 99 的人都做不到精通，如果你真的精通了，就是一堆工作称赞了。一般来说在个人技能这个区域我推荐再加上几个前端必备的技能就可以了，然后根据投递的公司可以选择性的添加几个对方需求且自己也会的技能块。最后个人技能这块内容同样也可以调整到简历的后半部分，没有必要占据一大块的简历第一部分。

3. 工作经历

前端开发工程师 | 技术经理 | 2016.09 - 至今

负责开发工程师 | 技术经理

负责前端开发工作（移动端、PC）
1. 使用 React Native 和 Webpack 构建项目，提升产品业务表现；
2. 及时协调异地团队处理线上问题，保证产品上线质量；
3. 参与前端技术架构设计，提升新架构节奏的开发时间；
4. 了解新技术和社区动态，熟悉 Java 等相关技术，熟悉 HTTP、HTTPS 协议；
5. 熟练使用 gulp/watch 模块管理打包工具，熟悉 DOM 对象模型标准。

2016.09 - 至今

这时间简历的工作经历这块写的基本没有什么问题，大家在写这一部分的时候需要注意以下几点：

- 在工作中有什么不错的结果都可以在这里表现出来，比如文中的绩效奖励小组 7 人最好等
- 在工作中都解决过什么很困难的问题也可以在这里提一下
- 最后划红点的一点，以上划红线的内容可能会被面试官问到，要做做到心中有数，知道该如何回答，否则还不如不说。比如如果简历中写到了新的架构节省了开发时间，那么这个架构是怎样的，你对这个架构有什么看法等等这些问题都可能会被问到，要准备好一个通用的回答。

4. 项目经历

前端开发工程师 | 技术经理 | 2016.09 - 2018.07

负责开发工程师 | 技术经理

负责前端开发工作（移动端、PC）
1. 使用 React Native 和 Webpack 构建项目，提升产品业务表现；
2. 及时协调异地团队处理线上问题，保证产品上线质量；
3. 参与前端技术架构设计，提升新架构节奏的开发时间；
4. 了解新技术和社区动态，熟悉 Java 等相关技术，熟悉 HTTP、HTTPS 协议；
5. 熟练使用 gulp/watch 模块管理打包工具，熟悉 DOM 对象模型标准。

2016.09 - 2018.07

这时简历的工作经历这块写的就是触碰到很多我之前提到过的问题了。

首先两个项目的经验介绍都挺流水账，都是属于使用了某项技术实现了某项功能。如果可简历的时候实在想不出平时工作中遇到什么困难或者解决了什么问题的话，就要确保以上可到的技术栈能够很好的回答出来。

以上划红的地方可能都会是面试者会重点提问的技术栈。

其实一封简历写的好，一般需要做到以下两点：

- 你让用人方了解到你比其他候选人强
- 不过分夸，确保简历里写的每一个技术点都心中有数

毕竟简历写的很华丽，只是敲开了公司的第一扇门，如果过分夸大了事实，那么其实就是浪费双方的时间了。

大家在写简历的时候可以多多注意以上我提到的几个点，然后在写完之后找出简历中涉及到的所有技术点，并且确保自己能够讲个所以然，这样简历过关就没什么问题了。

留言

输入评论 (Enter 按键, Ctrl + Enter 举报)

发表评论

全部评论 (29)

SWAYING | 前端工程师 | 6月前

还是得靠嘴 😊

回复

dev | 前端工程师 | 1年前

写得好

回复

风清云淡zy | 1年前

选择学霸榜

回复

叶海波风 | 2年前

狗头啊，白字转行的简历太难看了...

回复

shzhyd9 | 前端工程师 | 2年前

不要人机

回复

qianjin_0610 | web前端工程师 | 上海- | 2年前

不要人机

回复

小游戏生 | 前端研发 | 2年前

如果是像我这样连名字都懒得写，就可以放在简历里

回复

yyk | 作者 | 3年前

如果 Github 中有值得看的东西，就可以放在简历里

回复

我不知道我是谁 | 前端工程师 | 3年前

这简历不能读学长啊 😊

回复

小溪里 | https://www.xiaoki.c... | 3年前

简历是自己的脸，反映出自己的表达能力、表现能力，非常用心来做

回复

嘟嘟 | 3年前

想知道该怎么写？

回复

bb老猪 | 三年级 | student | 3年前

大佬，去找实习牛牛分享经验吧

回复

吐司吐司 | FE | 3年前

所长说在简历上面会把面试官都吓跑的啦问题对不对...

回复

yyk | 作者 | 3年前

对的，一般面试官都是照着简历挑一点东西问你，我经历过人家就不知晓问啥了

回复

andria | 3年前

女孩子的话，还可以问你是单身，狗头哈

回复

查看更多回答 ↴

快乐璐 | Web Development En... | 3年前

简历上要放自己的 github 吗？

回复

小游戏生 | 2年前

如果是我的话，就放吧

回复

小游戏生 | 2年前

如果是我的话，就放吧</p

面试常用技巧

这一章节我会介绍一些面试中常用的一些技巧，这些技巧可以帮助大家更好的准备面试，提高面试成功率。

尽早准备简历

找工作时的第一个重要问题就是写简历了。简历就是一个人的门面。简历写的好，用人单位也没有多大兴趣再深入了解你，毕竟行业人太多了。

很多人都会有一个问题就是：不知道简历该写啥。其实我都不推荐当要面试的时候去写简历，因为很多人没有记录的习惯，当去写简历的时候才会发现，在公司呆了那么久好像记不得自己做了哪些事了。

所以简历应该是经常去更新的，隔几个月去更新一次简历，了解自己这几个月以来的成长在哪里，结果是什么。

分批投递简历

当我们准备投递简历的时候，应该先把想投递的几个公司分成几个档次，先投递档次最低的，就算失败了，也就在摸经验。这样多投几次，把握大了就可以开始投递更加心仪的公司了，增加成功几率。

如何粗略判断公司是否靠谱

毕竟不是每个人都去大公司的，所以分辨一个公司是否靠谱是相当重要的，这关系到过来几个月甚至几年的职业道路。

首先一家公司所涉足的行业是很重要的，如果你去一家做社交的公司，很大程度上会以失败而告终。我个人认为目前教育、新能源、生鲜、医疗、数据这几个行业前景不错，当然这只是个人观点仅供参考。

然后我们还应该了解一家公司的情况，这里我推荐使用「天眼查」去查询一家公司的信息，在这里我们可以查到一家公司的几个重要指标：

- 具体的融资情况，一家公司好不好，拥有的资本肯定是要看的一块。一家不错的公司，往往前期融到的金额就很高
- 横向团队的介绍，通过我们可以了解到高管的一个教育背景，行业的经验等等
- 公司涉及到了哪些赛道，经营上的风险

然后还可以在网上查询一下这家公司是否有拖欠工资等等负面的消息。

如何回答面试官的问题

尽量不要止步于问题，也就是面试官问什么你答什么，而是把回答的点发散出去，引导面试官提问，展示自己的水平。

比如面试官提到了一个通过 DNS 查找 IP 的问题，那么在回答好这个问题的同时，可以指出获得 IP 以后就会发生 TCP 三次握手等等的情况，然后就可以引导面试官提问网络协议相关的问题了。

当然回答面试官的前提是你确实熟悉这一块的内容，否则就是给自己挖坑了。

我推荐大家在准备面试的过程中，挖掘出自己擅长的技术内容，然后在面试的过程中，寻找机会引导面试官提问你擅长的技术点。

最后需要注意一点，如果你不能很好的理解面试官的提问，最好先弄明白面试官到底想问什么，而不是直接回答问题导致出现文不对题的情况。

如何应对可能答不好的题目

假如你遇到了一道不会的题目，但是稍微有一点想法，你可以先坦白说这道题目不怎么会，但是愿意尝试回答一下，这样即使回答错了，也不会有什么问题。

但是千万不要不懂装懂，弄巧成拙。

多反思

一场面试结束以后，尽快的将面试中遇到的问题记录下来，然后复盘整个面试。

对于涉及到的题目，可以查一下资料验证自己是否答错了，如果答错了，就应该把这个知识点补充起来。

如果知识点对了，但是语言组织的不好，那么就需要重新组织下措辞和表达方式。

谈钱

我一直认为到手的才是真的，当然老板的大饼有时候也会梦想成真，但是这个更多的就是看个人机遇了。机遇不可求。大部分人还是应该追求到手的这一部分，在薪资满意的情况下，再去追求期权这些东西。

在面试之前应该想好自己想要的薪资，然后在和 HR 谈论工资的时候提高百分之 10 - 15 的样子，最终实话实说，大部分人都没有加班工资或者调休，但是薪资涨幅应该是你当下的百分之 15 以上，这样才能对冲跳槽带来的一个风险，当然如果你实在很想这家公司的活，那么薪资就另谈了。

在和 HR 讨论待遇的时候，应该问清楚以下几点

- 具体的工资（也就是合同上签订的工资），不要杂七杂八什么绩效加起来的那种
- 五险一金缴纳的比例
- 加班费是否有加班工资或者调休
- 是否是 996，我个人很不推荐 996 的公司
- 加班开罚的情况
- 其他各种福利，比如餐补、房补、交通补、节假日福利、另外的保险等等

留言

全部评论 (13) 发表评论

光耀年代 | 前端开发工程师 | 5月前
up 主说的极差。在工作中，要过几个月就忘掉一次自己的收获，而不是每年年初定下目标，干劲满满。年头回头一看，好像啥都没做成，真的是这个样子。现在急需改变~，看了up主的文章收获很多，自己也准备后面写笔记，记录自己工作中遇到的问题以及解决方案。还有自己学习的知识点。为自己加油，而不是抱怨着过每一天！

白日梦 | 6月前

简单 | 6月前
教育除了

白日梦 | 6月前

大家一致 | 老板 @ 深圳xxx有限公司 | 1年前
这篇就是有用的，不过很多地方讲解的不够通俗易懂

白日梦 | 6月前

tocored | 1年前
赞一个，抵制 996

白日梦 | 6月前

代码狂魔 | web前端 | 1年前
看完收获是有，不过很多地方讲解的不够通俗易懂

白日梦 | 6月前

ModelZachary | 前端开发工程师 | 2年前
程序员普世规律，我一般都是摸爬滚打维护我马是怎么协调的

白日梦 | 6月前

用户259247... | 8月前
少玩点手机，下班开始肝

白日梦 | 6月前

春家老大 | 2年前
这里面干货比较多

白日梦 | 6月前

softbone | 前端开发工程师 @ xiao... | 3年前
要对自己负责这些都掌握了在投简历面试吗

白日梦 | 6月前

白日梦 | 全栈工程师 @ vivo | 3年前
这篇文章几个观点很不错：第一：实时更新简历，从而实时看到自己的成长和不足。第二：分批投递简历，从低优先级开始打怪，老哥观点都挺实在

白日梦 | 6月前

前方的路，让我们结伴同行

总结

首先感谢各位购买这本小册。这是我的第一本小册，内容可能会存在瑕疵。感谢大家选择这本小册，选择相信我这个作者。

相信大家都看过一句话：面试连火都工作柠檬丝。诚然，现在大公司的门槛确实高，但这也是因为僧多粥少，供大于求的问题。因此大部分公司会选择优而汰。那么为了进入大公司（我相信大部分人都有这个梦想），我们都需要主动自己去学习，探索更深入的领域，而不是放慢脚步，认为会使用框架 API 能够就行了。

面试时的信心来自于我们面试前的充分准备和平时的积累。有了信心，才能在面试中披荆斩棘，无往不胜。而不是毫无头绪的自我感觉良好，面试失败就是空口刁难、考题太难，责怪于外部因素而不是寻找个人的不足。

我也不说互联网不容易，毕竟物价、房价摆在那里，想做好的，赚的很多，只能好好学，好好干，无论春夏秋冬，很感谢看到最后一章节的各位。相信这本小册能给大家带来一些不小的收获。

展望未来

小册不是我们学习的终点。为了减少大家找寻学习资料的时间，接下来我会提供一些我认为不错的学习资料供大家参考。

当然了资料是一部分，其实我更推荐在工作中学习。深入学习工作中用到的技术并且储备一些未来可能用到的技能，这样对个人职业发展是很有所帮助的。

JS

- [You-Dont-Know-JS](#)，这套书深入的讲解很多 JS 的内容，英文版是开源免费阅读的，如果你英文不好的话，国内这套书已经出版了，可以选择购买。
- [Functional-Light-JS](#)，这本书是讲解函数式编程的，函数式编程也是一种编程范式，轻量级的函数式可以很方便的解决很多问题，有兴趣的可以一读。
- [33-js-concepts](#)，这份资料讲解了 33 个前端开发必须知道的 JS 概念，内容是英文的，如果你英文不好的话，国内这套书已经出版了中文版。
- [前端阅读周刊](#)，这是一份前端技术文章集合，每周都会更新，目前已经更新了 84 篇文章。
- [前端性能清单](#)，这是一份前端性能清单，如果你需要优化一个项目的话，可以根据这份清单一个个来检查优化项。
- [30-seconds-of-code](#)，30 秒系列，很短的代码片段让你了解一个知识点。
- [must-watch-javascript](#)，这份资料包含了很多高质量的前端相关视频，值得一看。

CSS

- [css-protips](#)，通过这份资料你可以了解到很多 CSS 提高性能的技巧。
- [30-seconds-of-css](#)，30 秒系列，很短的代码片段让你了解一个知识点。
- [CSS 世界](#)，张鑫旭出版的书籍，没什么好说的了，看就是了。
- [一些有趣的 CSS 视频](#)，CSS 奇技淫巧，在这里，都有。

框架

框架这里其实我不想推荐任何的资料，如果你单纯想学习一个框架的话，我只推荐阅读官方文档学习，没有任何的必要去阅读其他的入门资料，因为基本上都是照搬文档的。

如果你想进一步学习框架的内容的话，我推荐去阅读框架核心团队成员的博客，比如 React 核心团队成员 Dan Abramov 的 [blog](#)。

Node

Node.js 几乎是资深前端工程师跨不过去的一道坎，也是一个团队的通用底层能力。学习 Node 可以更好的使用工具，建立起一套技能圈层服务于整个项目。

- [Node.js 测试指南](#)，这是一份专注于讲解 Node 测试的书籍，已经出版了，但是可以开源免费阅读。
- [来一打 C++ 扩展](#)，吴月出版的书籍，没什么好说的了，看就是了。
- [Node.js 最佳实践](#)，这是一份 Node.js 最佳实践中排名第一的内容的总结和分享。

安全

- [the-book-of-secret-knowledge](#)，这是一份安全领域的资料，如果你对安全感兴趣的话，可以阅读一下内容。

周报

- [奇闻周刊](#)，每周都会整理一份不错的中文文章合集。
- [TechBrill Weekly](#)，这是一份台湾地区整理的一份多个技术领域的周报。
- [JavaScript Weekly](#)，这是一份很有名气的英文周报，整理的文章质量都很高，如果你只想订阅一份周报，那就是它了。
- [Any Foo Weekly](#)，这也是一份不错的英文周报，文章质量也很高，并且和上一份周报重复的内容不多。

Medium

Medium 上我并没有怎么固定阅读，更多的是订阅它的日报或者从别的阅读上看到的 Medium 的文章，但是如果一定要推荐两个组织的话，我只推荐这两个，毕竟他们的文章质量都很高。

- [freecodecamp](#)
- [hackernoon](#)

Youtube

Youtube 有很多高质量的视频，但是门槛大家都知道，这里我推荐一些值得订阅的频道。

- [JSConf](#)，很多会议的视频你都可以在这里找到。
- [Google Chrome Developers](#)，Google 官方招牌，没啥好说的。
- [Computernphile](#)，内容偏向于计算机领域。
- [Coding Tech](#)，内容偏向于入门。
- [Fun Fun Function](#)，如果你想学习函数式编程的一些内容，这是一个值得订阅的频道。
- [DevTips](#)，每周更新一个视频，能够学到不少开发中的 Tips。

其他

- [互联网公司技术架构](#)，这份资料介绍了当下互联网公司的一个技术架构。
- [javascript-algorithms](#)，这份资料作者使用了 JS 来实现了大部分的数据结构和算法。
- [小型编译器](#)，这份资料告诉了我们该如何去实现一个小型的编译器，很适合前端开发者阅读。
- [every-programmer-should-know](#)，这份资料列举了很多每个开发者都应该知道的知识点。

最后

你可能发现我推荐的很多内容都是英文的或者你并不能打开，这里我只能说一声抱歉。因为我获取学习资料更多的来源于国外。可能不能很好照顾到英文不好的同学，但是说一句脏话之吧，技术需求的英文真的要求不高。花点时间静下心去阅读英文资料，坚持几个月，从此技术的大门就完全敞开了。

如果大家也有不错的资料想要分享，欢迎在留言区留言。

在小册的最后一章，打一个自己公众号的广告。如果你想了解一些前端的热点、新知识、学习感悟等的话，你可以关注我的公众号「前端真好玩」。

最后的最后，真的很感谢购买我小册的朋友，同时也感谢一些业内大佬花时间阅读我的小册，并给我提出了修改意见，是你们促使我一直写下去的。我们，下本书再见！

留言

输入评论 (Enter执行, Ctrl + Enter发送)

发表评论

全部评论 (38)

Runnin (1) | 问题不大 @ W | 3天前
别这么一想
点点赞 回复

HDDW | 14天前
小册知识还是很全面的，一般大部分公司都是问这些，但是讲得不够深入，还需要自己去挖深学习。
点点赞 回复

喜欢蓝色大海 (2) | 2周前
看到评论很多对小白不太满意的，我就属于那个对这本小册特别满意的，这本小册确实帮找到了工作，通过了多家公司的面试，我觉得这本小册写的内容也很好，就喜欢它每一章节的内容少，因为只是一引带十，要真正做到理解，网上讲的代码对我没用，反之，如果这本小册每一章节的内容都讲详细，反而失去了看它的动力，厚积薄发看下去实在没劲动力，所以对原来给作者写的这本小册详细程度拍手叫好，感谢作者，您是最棒的。点点赞 回复

RollinLee (1) | 软件开发工程师 @ CIB | 2月前
非常后悔买了 遗憾，！！
点点赞 回复

前SenSei~ | 前端工程师 @ 上海思勰 | 2月前
别这么一想
点点赞 回复

isgoku (1) | 前端开发工程师 | 7月前
最后一章是质量很高的，整体来说写的很好。
点点赞 回复

zgg03 (1) | zzz | 1年前
感谢 🙏
点点赞 回复

HolyChen | 前端工程师 | 1年前
最喜欢的小册
点点赞 回复

小海看大海 (1) | 前端开发 | 1年前
刚刷，去年11月份买的，今天又看了，发现新增了很多知识点，补充了很多，对于重构我的知识架构有很大帮助，大佬现在善于活用，内推我啊，哈哈。
点点赞 回复

yangxwsming_ | web前端工程师 | 1年前
希望后边能更新些新的内容
点点赞 回复

凸奇皇朝 (1) | 造型设计大赛季军 | 1年前
别这么一想
点点赞 回复

领航 (1) | 前端 @ frontzhm @... | 2年前
前面666
点点赞 回复

已生蛹 | 2年前
收获很大，操作一直写小册，有个建议就是希望小册的深度和广度大一点，读者分享
点点赞 回复

Owen | 前端开发工程师 @ YAHOO | 2年前
看了还是有收获的，有些东西只能靠这本书了。职业生涯本来就应该有自己的方向，应该有自己的一个定位，是想靠自己去学别人的话，自己理解到底才是自己的。
点点赞 回复

syng | 1年前
看了类型转换这一节，感觉小册讲的模棱两可，略显晦涩。关于此类型的转换: II == II 还涉及到运算符的优先级问题。当然还有其他，总体感觉一般。II == II
点点赞 回复

感谢君 (1) | 反向激励战士 | 2年前
感谢作者，收获很大！
点点赞 回复

naturewide | 2年前
感谢作者，真的很实用
点点赞 回复

伊虎 | 2年前
感谢！！！！！
点点赞 回复

安芝 | 2年前
very good
点点赞 回复

已生蛹 | 2年前
大佬，技术的极值。
点点赞 回复

查看全部 38 条回复