

第五部分：高频考点

1 typeof类型判断

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

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

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

js

`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'
typeof console.log // 'function'
```

js

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

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

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

js

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

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

2 类型转换

首先我们要知道，在 `JS` 中类型转换只有三种情况，分别是：

- 转换为布尔值
- 转换为数字
- 转换为字符串



转Boolean

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

对象转原始类型

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

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

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

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

js

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

四则运算符

它有以下几个特点：

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

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

js

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

另外对于加法还需要注意这个表达式 `'a' + + 'b'`

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

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

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

js

比较运算符

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

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

js

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

3 This

我们先来看几个函数调用的场景

```
function foo() {  
  console.log(this.a)  
}  
var a = 1  
foo()  
  
const obj = {  
  a: 2,  
  foo: foo  
}  
obj.foo()  
  
const c = new foo()
```

js

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

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

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

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

js

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

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

js

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

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

js

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

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

js

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

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



4 == 和 === 有什么区别

对于 `==` 来说，如果对比双方的类型不一样的话，就会进行类型转换

假如我们需要对比 `x` 和 `y` 是否相同，就会进行如下判断流程

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

```
1 == '1'  
  ↓  
1 == 1
```

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

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

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

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



对于 `===` 来说就简单多了，就是判断两者类型和值是否相同

5 闭包

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

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

js

闭包存在的意义就是让我们可以间接访问函数内部的变量

经典面试题，循环中使用闭包解决 `var` 定义函数的问题

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

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

解决办法有三种

1. 第一种是使用闭包的方式

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

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

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

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


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

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

js

6 深浅拷贝

浅拷贝

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

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

js

另外我们还可以通过展开运算符 `...` 来实现浅拷贝

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

js

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

js

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

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

深拷贝

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

js

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

但是该方法也是有局限性的：

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

js

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

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

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

7 原型

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

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

8 var、let 及 const 区别

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

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

9 原型继承和 Class 继承

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

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

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

js

组合继承

组合继承是最常用的继承方式

```
function Parent(value) {  
  this.val = value  
}  
Parent.prototype.getValue = function() {  
  console.log(this.val)  
}  
function Child(value) {  
  Parent.call(this, value)  
}  
Child.prototype = new Parent()  
  
const child = new Child(1)  
  
child.getValue() // 1  
child instanceof Parent // true
```

js

- 以上继承的方式核心是在子类的构造函数中通过 `Parent.call(this)` 继承父类的属性，然后改变子类的原型为 `new Parent()` 来继承父类的函数。
- 这种继承方式优点在于构造函数可以传参，不会与父类引用属性共享，可以复用父类的函数，但是也存在一个缺点就是在继承父类函数的时候调用了父类构造函数，导致子类的原型上多了不需要的父类属性，存在内存上的浪费

寄生组合继承

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

```
function Parent(value) {  
  this.val = value  
}  
Parent.prototype.getValue = function() {  
  console.log(this.val)  
}  
  
function Child(value) {  
  Parent.call(this, value)  
}  
Child.prototype = Object.create(Parent.prototype, {  
  constructor: {  
    value: Child,  
    enumerable: false,  
    writable: true,  
    configurable: true  
  }  
})  
  
const child = new Child(1)  
  
child.getValue() // 1  
child instanceof Parent // true
```

js

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

Class 继承

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

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

`class` 实现继承的核心在于使用 `extends` 表明继承自哪个父类，并且在子类构造函数中必须调用 `super`，因为这段代码可以看成 `Parent.call(this, value)`。

10 模块化

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

使用一个技术肯定是有原因的，那么使用模块化可以给我们带来以下好处

- 解决命名冲突
- 提供复用性
- 提高代码可维护性

立即执行函数

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

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

AMD 和 CMD

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

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

js

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
}
// or
exports.a = 1

// b.js
```

js

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

js

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

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

ES Module

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

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

渲染有很大影响

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

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

js

11 实现一个简洁版的promise

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

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

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

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

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

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

function reject(value) {
```

js

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

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

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

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

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

12 Event Loop

12.1 进程与线程

涉及面试题：进程与线程区别？ JS 单线程带来的好处？

- **JS** 是单线程执行的，但是你是否疑惑过什么是线程？
- 讲到线程，那么肯定也得说一下进程。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** 在运行指令及加载和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，描述了执行一段指令所需的时间

把这些概念拿到浏览器中来说，当你打开一个 **Tab** 页时，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** 请求线程等等。当你发起一个请求时，其实就是创建了一个线程，当请求结束后，该线程可能就会被销毁

- 上文说到了 **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。这其中的原因是因为 **JS** 可以修改 **DOM**，如果在 **JS** 执行的时候 **UI** 线程还在工作，就可能导致不能安全的渲染 **UI**。这其实也是一个单线程的好处，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，没有锁的问题的好处

12.2 执行栈

涉及面试题：什么是执行栈？

可以把执行栈认为是一个存储函数调用的栈结构，遵循先进后出的原则

当开始执行 **JS** 代码时，首先会执行一个 **main** 函数，然后执行我们的代码。根据先进后出的原则，后执行的函数会先弹出栈，在图中我们也可以发现，**foo** 函数后执行，当执行完毕后就从栈中弹出了

在开发中，大家也可以在报错中找到执行栈的痕迹

```
function foo() {  
  throw new Error('error')  
}  
function bar() {  
  foo()  
}  
bar()
```

js

大家可以在上图清晰的看到报错在 `foo` 函数，`foo` 函数又是在 `bar` 函数中调用的

当我们使用递归的时候，因为栈可存放的函数是有限制的，一旦存放了过多的函数且没有得到释放的话，就会出现爆栈的问题

```
function bar() {  
  bar()  
}  
bar()
```

js

12.3 浏览器中的 Event Loop

涉及面试题：异步代码执行顺序？解释一下什么是 `Event Loop` ？

众所周知 `JS` 是门非阻塞单线程语言，因为在最初 `JS` 就是为了和浏览器交互而诞生的。如果 `JS` 是门多线程的语言话，我们在多个线程中处理 `DOM` 就可能会发生问题（一个线程中新加节点，另一个线程中删除节点）

- `JS` 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 `Task`（有多种 `task`）队列中。一旦执行栈为空，`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

```
console.log('script start');  
  
setTimeout(function() {  
  console.log('setTimeout');  
}, 0);
```

js

```
console.log('script end');
```

不同的任务源会被分配到不同的 **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 **task**

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
  console.log('Promise')
  resolve()
}).then(function() {
  console.log('promise1');
}).then(function() {
  console.log('promise2');
});

console.log('script end');
// script start => Promise => script end => promise1 => promise2 => setTime
```

以上代码虽然 **setTimeout** 写在 **Promise** 之前，但是因为 **Promise** 属于微任务而 **setTimeout** 属于宏任务

微任务

- **process.nextTick**
- **promise**
- **Object.observe**
- **MutationObserver**

宏任务

- **script**
- **setTimeout**

- `setInterval`
- `setImmediate`
- `I/O`
- `UI rendering`

宏任务中包括了 `script`，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务

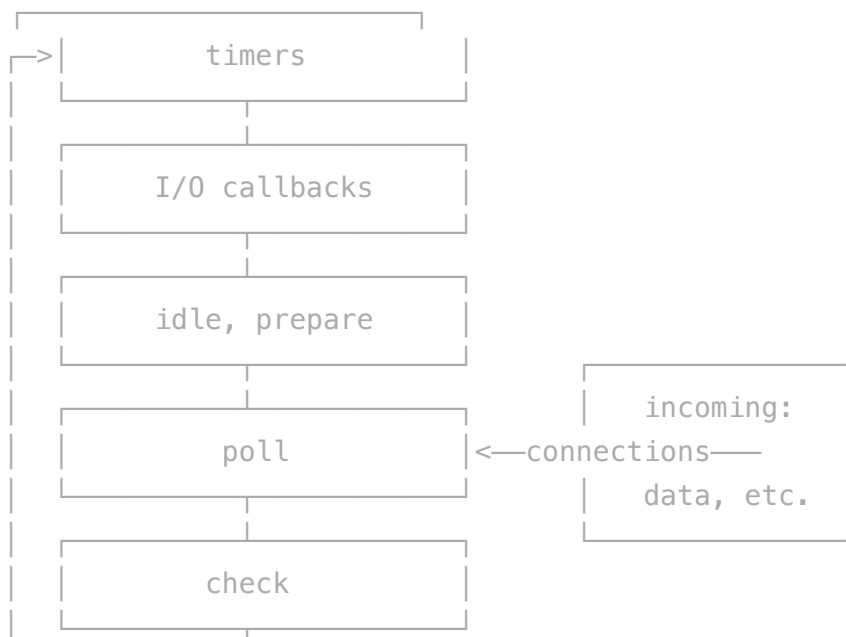
所以正确的一次 Event loop 顺序是这样的

- 执行同步代码，这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 UI
- 然后开始下一轮 `Event loop`，执行宏任务中的异步代码

通过上述的 `Event loop` 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 `DOM` 的话，为了更快的响应界面响应，我们可以把操作 `DOM` 放入微任务中

12.4 Node 中的 Event loop

- `Node` 中的 `Event loop` 和浏览器中的不相同。
- `Node` 的 `Event loop` 分为 6 个阶段，它们会按照顺序反复运行





timer

- **timers** 阶段会执行 **setTimeout** 和 **setInterval**
- 一个 timer 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟

I/O

- **I/O** 阶段会执行除了 **close** 事件，定时器和 **setImmediate** 的回调

poll

- **poll** 阶段很重要，这一阶段中，系统会做两件事情
 - 执行到点的定时器
 - 执行 **poll** 队列中的事件
- 并且当 **poll** 中没有定时器的情况下，会发现以下两件事情
 - 如果 **poll** 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
 - 如果 **poll** 队列为空，会有两件事发生
 - 如果有 **setImmediate** 需要执行，**poll** 阶段会停止并且进入到 **check** 阶段执行 **setImmediate**
 - 如果没有 **setImmediate** 需要执行，会等待回调被加入到队列中并立即执行回调
 - 如果有别的定时器需要被执行，会回到 **timer** 阶段执行回调。

check

- **check** 阶段执行 **setImmediate**

close callbacks

- **close callbacks** 阶段执行 **close** 事件
- 并且在 **Node** 中，有些情况下的定时器执行顺序是随机的

```
setTimeout(() => {  
  console.log('setTimeout');  
}, 0);  
setImmediate(() => {  
  console.log('setImmediate');
```

js

```

})
// 这里可能会输出 setTimeout, setImmediate
// 可能也会相反的输出, 这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒, 这时候会执行 setImmediate
// 否则会执行 setTimeout

```

上面介绍的都是 **macrotask** 的执行情况, **microtask** 会在以上每个阶段完成后立即执行

```

setTimeout(()=>{
  console.log('timer1')

  Promise.resolve().then(function() {
    console.log('promise1')
  })
}, 0)

```

```

setTimeout(()=>{
  console.log('timer2')

  Promise.resolve().then(function() {
    console.log('promise2')
  })
}, 0)

```

```

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中一定打印 timer1, promise1, timer2, promise2
// node 中可能打印 timer1, timer2, promise1, promise2
// 也可能打印 timer1, promise1, timer2, promise2

```

Node 中的 **process.nextTick** 会先于其他 **microtask** 执行

```

setTimeout(() => {
  console.log("timer1");

  Promise.resolve().then(function() {
    console.log("promise1");
  });
}, 0);

process.nextTick(() => {

```

js

js


```
console.log("nextTick");
});
// nextTick, timer1, promise1
```

对于 `microtask` 来说，它会在以上每个阶段完成前清空 `microtask` 队列，下图中的 `Tick` 就代表了 `microtask`

13 手写 call、apply 及 bind 函数

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

实现 call

- 首先 `context` 为可选参数，如果不传的话默认上下文为 `window`
- 接下来给 `context` 创建一个 `fn` 属性，并将值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出来
- 然后调用函数并将对象上的函数删除

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

js

apply实现

`apply` 的实现也类似，区别在于对参数的处理

js

```
Function.prototype.myApply = function(context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  context = context || window
  context.fn = this
  let result
  // 处理参数和 call 有区别
  if (arguments[1]) {
    result = context.fn(...arguments[1])
  } else {
    result = context.fn()
  }
  delete context.fn
  return result
}
```

bind 的实现

`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

- `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，我们先来说直接调用的方式
- 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，不会被任何方式改变 `this`，所以对于这种情况我们需要忽略传入的 `this`

js

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

```
    }  
    return _this.apply(context, args.concat(...arguments))  
  }  
}
```

14 new

涉及面试题： `new` 的原理是什么？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` 的过程中会发生四件事情

- 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`

- 创建一个空对象
- 获取构造函数
- 设置空对象的原型
- 绑定 `this` 并执行构造函数
- 确保返回值为对象

```
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  
}
```

js

- 对于对象来说，其实都是通过 `new` 产生的，无论是 `function Foo()` 还是 `let a = { b : 1 }`。
- 对于创建一个对象来说，更推荐使用字面量的方式创建对象（无论性能上还是可读性）。因为你使用 `new Object()` 的方式创建对象需要通过作用域链一层层找到 `Object`，但是你使用字面量的方式就没这个问题

```
function Foo() {}  
// function 就是个语法糖  
// 内部等同于 new Function()  
let a = { b: 1 }  
// 这个字面量内部也是使用了 new Object()
```

15 instanceof 的原理

涉及面试题： `instanceof` 的原理是什么？

`instanceof` 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 `prototype`

实现一下 instanceof

- 首先获取类型的原型
- 然后获得对象的原型
- 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`，因为原型链最终为 `null`

```
function myInstanceOf(left, right) {  
  let prototype = right.prototype  
  left = left.__proto__  
  while (true) {  
    if (left === null || left === undefined)  
      return false  
    if (prototype === left)  
      return true  
    left = left.__proto__  
  }  
}
```

16 为什么 `0.1 + 0.2 !== 0.3`

涉及面试题：为什么 `0.1 + 0.2 !== 0.3`？如何解决这个问题？

原因，因为 JS 采用 IEEE 754 双精度版本（64 位），并且只要采用 IEEE 754 的语言都有该问题

我们都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

```
// (0011) 表示循环  
0.1 = 2-4 * 1.10011(0011)
```

js

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剪掉我们的数字。

IEEE 754 双精度版本（64位）将 64 位分为了三段

- 第一位用来表示符号
- 接下去的 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 10011(0011)

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 0.100000000000000002

```
0.100000000000000002 === 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 0.200000000000000002

```
0.200000000000000002 === 0.2 // true
```

所以这两者相加不等于 0.3 而是 0.300000000000000004

```
0.1 + 0.2 === 0.300000000000000004 // 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，二进制被转换为了十进制，十进制又被转换为了字符串，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(0.100000000000000002) // 0.1
```

解决

```
parseFloat((0.1 + 0.2).toFixed(10)) === 0.3 // true
```

js

17 事件机制

涉及面试题：事件的触发过程是怎么样的？知道什么是事件代理嘛？

17.1 事件触发三阶段

事件触发有三个阶段：

- `window` 往事件触发处传播，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照上面的顺序进行，但是也有特例，如果给一个 `body` 中的子节点同时注册冒泡和捕获事件，事件触发会按照注册的顺序执行

```
// 以下会先打印冒泡然后是捕获
node.addEventListener(
  'click',
  event => {
```

js

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

17.2 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`：布尔值，值为 `true` 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触发在目标上，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别的注册事件。

```
node.addEventListener(
  'click',
  event => {
    event.stopImmediatePropagation()
    console.log('冒泡')
  },
  false
)
// 点击 node 只会执行上面的函数，该函数不会执行
node.addEventListener(
  'click',
```

js

```
event => {  
  console.log('捕获 ')  
},  
true  
)
```

17.3 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注册在父节点上

```
<ul id="ul">  
  <li>1</li>  
  <li>2</li>  
  <li>3</li>  
  <li>4</li>  
  <li>5</li>  
</ul>  
<script>  
  let ul = document.querySelector('#ul')  
  ul.addEventListener('click', (event) => {  
    console.log(event.target);  
  })  
</script>
```

html

事件代理的方式相较于直接给目标注册事件来说，有以下优点：

- 节省内存
- 不需要给子节点注销事件

18 跨域

涉及面试题：什么是跨域？为什么浏览器要使用同源策略？你有几种方式可以解决跨域问题？了解预检请求嘛？

- 因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域，**Ajax** 请求会失败。

- 那么是出于什么安全考虑才会引入这种机制呢？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** 网站可以被任意其他来源的 **Ajax** 访问到内容。如果你当前 **A** 网站还存在登录态，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。你可能会疑问明明通过表单的方式可以发起跨域请求，为什么 **Ajax** 就不会。因为归根结底，跨域是为了阻止用户读取到另一个域名下的内容，**Ajax** 可以获取响应，浏览器认为这不安全，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阻止 **CSRF**，因为请求毕竟是发出去了。

接下来我们将来学习几种常见的方式来解决跨域的问题

18.1 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://domain/api?param1=a&param2=b&callback=jsonp"></script>
<script>
    function jsonp(data) {
        console.log(data)
    }
</script>
```

JSONP 使用简单且兼容性不错，但是只限于 **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**，以下是简单实现

```
function jsonp(url, jsonpCallback, success) {
  let script = document.createElement('script')
  script.src = url
  script.async = true
  script.type = 'text/javascript'
  window[jsonpCallback] = function(data) {
    success && success(data)
  }
  document.body.appendChild(script)
}
jsonp('http://xxx', 'callback', function(value) {
  console.log(value)
})
```

18.2 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，如果设置通配符则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** 的值仅限于下列三者之一：

- **text/plain**
- **multipart/form-data**
- **application/x-www-form-urlencoded**

请求中的任意 `XMLHttpRequestUpload` 对象均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

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端是否允许跨域请求。

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

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

- 该请求会验证你的 `Authorization` 字段，没有的话就会报错。
- 当前端发起了复杂请求后，你会发现就算你代码是正确的，返回结果也永远是报错的。因为预检请求也会进入回调中，也会触发 `next` 方法，因为预检请求并不包含 `Authorization` 字段，所以服务端会报错。

想解决这个问题很简单，只需要在回调中过滤 `option` 方法即可

```
js
res.statusCode = 204
res.setHeader('Content-Length', '0')
res.end()
```

18.3 document.domain

- 该方式只能用于主域名相同的情况下，比如 `a.test.com` 和 `b.test.com` 适用于该方式。
- 只需要给页面添加 `document.domain = 'test.com'` 表示主域名都相同就可以实现跨域

18.4 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

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

js

19 存储

涉及面试题：有几种方式可以实现存储功能，分别有什么优缺点？什么是 `Service Worker` ？

cookie, localStorage, sessionStorage, indexDB

特性	cookie	localStorage	sessionStorage	indexDB
数据生命周期	一般由服务器生成，可以设置过期时间	除非被清理，否则一直存在	页面关闭就清理	除非被清理，否则一直存在
数据存储大小	4K	5M	5M	无限

特性	cookie	localStorage	sessionStorage	indexDB
与服务端通信	每次都会携带在 header 中，对于请求性能影响	不参与	不参与	不参与

从上表可以看到， **cookie** 已经不建议用于存储。如果没有大量数据存储需求的话，可以使用 **localStorage** 和 **sessionStorage** 。对于不怎么改变的数据尽量使用 **localStorage** 存储，否则可以用 **sessionStorage** 存储

对于 **cookie** 来说，我们还需要注意安全性。

属性	作用
value	如果用于保存用户登录态，应该将该值加密，不能使用明文的用户标识
http-only	不能通过 JS 访问 Cookie ，减少 XSS 攻击
secure	只能在协议为 HTTPS 的请求中携带
same-site	规定浏览器不能在跨域请求中携带 Cookie ，减少 CSRF 攻击

Service Worker

- **Service Worker** 是运行在浏览器背后的独立线程，一般可以用来实现缓存功能。使用 **Service Worker** 的话，传输协议必须为 **HTTPS** 。因为 **Service Worker** 中涉及到请求拦截，所以必须使用 **HTTPS** 协议来保障安全
- **Service Worker** 实现缓存功能一般分为三个步骤：首先需要先注册 **Service Worker** ，然后监听到 **install** 事件以后就可以缓存需要的文件，那么在下次用户访问的时候就可以通过拦截请求的方式查询是否存在缓存，存在缓存的话就可以直接读取缓存文件，否则就去请求数据。以下是这个步骤的实现：

// index.js

js

```
if (navigator.serviceWorker) {
  navigator.serviceWorker
    .register('sw.js')
    .then(function(registration) {
      console.log('service worker 注册成功')
    })
    .catch(function(err) {
      console.log('servcie worker 注册失败')
    })
}
```

```
}  
// sw.js  
// 监听 `install` 事件，回调中缓存所需文件  
self.addEventListener('install', e => {  
  e.waitUntil(  
    caches.open('my-cache').then(function(cache) {  
      return cache.addAll(['./index.html', './index.js'])  
    })  
  )  
})  
  
// 拦截所有请求事件  
// 如果缓存中已经有请求的数据就直接用缓存，否则去请求数据  
self.addEventListener('fetch', e => {  
  e.respondWith(  
    caches.match(e.request).then(function(response) {  
      if (response) {  
        return response  
      }  
      console.log('fetch source')  
    })  
  )  
})
```

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

20 浏览器缓存机制

注意：该知识点属于性能优化领域，并且整一章节都是一个面试题

- 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骤中优化性能。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存储的数据和前端一致，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

20.1 缓存位置

从缓存位置上来说分为四种，并且各自有优先级，当依次查找缓存且都没有命中的时候，才会去请求网络

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

1. Service Worker

- `service Worker` 的缓存与浏览器其他内建的缓存机制不同，它可以让我们自由控制缓存哪些文件、如何匹配缓存、如何读取缓存，并且缓存是持续性的。
- 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，会根据缓存查找优先级去查找数据。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` 中获取的内容。

2. Memory Cache

- **Memory Cache** 也就是内存中的缓存，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，可是缓存持续性很短，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

那么既然内存缓存这么高效，我们是不是能让数据都存放在内存中呢？

- 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，操作系统需要精打细算内存的使用，所以能让我们使用的内存必然不多。内存中其实可以存储大部分的文件，比如说 **JS** 、 **HTML** 、 **CSS** 、 图片等等
- 当然，我通过一些实践和猜测也得出了一些结论：
- 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，文件优先存储进硬盘

3. Disk Cache

- **Disk Cache** 也就是存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 **Memory Cache** 胜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·` 中的字段判断哪些资源需要缓存，哪些资源可以不请求直接使用，哪些资源已经过期需要重新请求。并且即使在跨站点的情况下，相同地址的资源一旦被硬盘缓存下来，就不会再次去请求数据

4. 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** 在国内不够普及，但是 **HTTP/2** 将会是日后的一个趋势

结论

- 所有的资源都能被推送，但是 **Edge** 和 **Safari** 浏览器兼容性不怎么好
- 可以推送 **no-cache** 和 **no-store** 的资源
- 一旦连接被关闭，**Push Cache** 就被释放
- 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

- 浏览器可以拒绝接受已经存在的资源推送
- 你可以给其他域名推送资源

5. 网络请求

- 如果所有缓存都没有命中的话，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

20.2 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

20.2.1 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，**state code** 为 **200**

Expires

`Expires: Wed, 22 Oct 2018 08:41:00 GMT`

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: max-age=30`

- **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

从图中我们可以看到，我们可以将多个指令配合起来一起使用，达到多个目的。比如说我们希望资源能被缓存下来，并且是客户端和代理服务器都能缓存，还能设置缓存失效时间等

一些常见指令的作用

20.2.2 协商缓存

- 如果缓存过期了，就需要发起请求验证资源是否有更新。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，那么服务端就会返回 **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** 的值发送给服务器，询问服务器在该日期后资源是否有更新，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 **Last-Modified** 被修改，服务端不能命中缓存导致发送相同的资源
- 因为 **Last-Modified** 只能以秒计时，如果在不可感知的时间内修改完成文件，那么服务端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送回来。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。

20.3 实际场景应用缓存策略

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` 使浏览器每次都请求服务器，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

这里特指除了 `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，只有当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

更多缓存知识详解 <http://blog.poetries.top/2019/01/02/browser-cache>

21 浏览器渲染原理

注意：该章节都是一个面试题。

21.1 渲染过程

1. 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，所以在网络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节数据以后，它会将这些字节数据转换为字符串，也就是我们写的代码。

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`），这一过程在词法分析中叫做标记化（`tokenization`）

那么什么是标记呢？这其实属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。这一过程会将代码分拆成一块块，并给这些内容打上标记，便于理解这些最小单位的代码是什么意思

当结束标记化后，这些标记会紧接着转换为 `Node`，最后这些 `Node` 会根据不同 `Node` 之前的联系构建为一颗 `DOM` 树

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

当然，在解析 `HTML` 文件的时候，浏览器还会遇到 `CSS` 和 `JS` 文件，这时候浏览器也会去下载并解析这些文件，接下来就让我们先来学习浏览器如何解析 `CSS` 文件

2. 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极其类似的

- 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很消耗资源的。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，浏览器得递归 **CSSOM** 树，然后确定具体的元素到底是什么样式。

如果你有点不理解为什么会消耗资源的话，我这里举个例子

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

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，最后再去找找到 **div** 标签，然后给符合这种条件的 **span** 标签设置颜色，这样的递归过程就很复杂。所以我们应该尽可能的避免写过于具体的 **CSS** 选择器，然后对于 **HTML** 来说也尽量少的添加无意义标签，保证层级扁平

3. 生成渲染树

当我们生成 **DOM** 树和 **CSSOM** 树以后，就需要将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。渲染树只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，就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，合成图层，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的知识，这里就不再继续展开内容了。

21.2 为什么操作 DOM 慢

想必大家都听过操作 DOM 性能很差，但是这其中的原因是什么呢？

- 因为 DOM 是属于渲染引擎中的东西，而 JS 又是 JS 引擎中的东西。当我们通过 JS 操作 DOM 的时候，其实这个操作涉及到了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 次数一多，也就等同于一直在进行线程之间的通信，并且操作 DOM 可能还会带来重绘回流的情况，所以也就导致了性能上的问题。

经典面试题：插入几万个 DOM，如何实现页面不卡顿？

- 对于这道题目来说，首先我们肯定不能一次性把几万个 DOM 全部插入，这样肯定会造成卡顿，所以解决问题的重点应该是怎么分批插入部分渲染 DOM。大部分人应该可以想到通过 requestAnimationFrame 的方式去循环的插入 DOM，其实还有种方式去解决这个问题：虚拟滚动（virtualized scroller）。
- 这种技术的原理就是只渲染可视区域内的内容，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

从上图中我们可以发现，即使列表很长，但是渲染的 DOM 元素永远只有那么几个，当我们滚动页面的时候就会实时去更新 DOM，这个技术就能顺利解决这道经典面试题

21.3 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，并且扁平层级，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，如果你想首屏渲染的越快，就越不应该在首屏就加载 JS 文件，这也是都建议将 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。

21.4 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而不会影响布局的，比如改变 `color` 就叫称为重绘
- 回流是布局或者几何属性需要改变就称为回流。
- 回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变父节点里的子节点很可能会导致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，重绘和回流其实也和 `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` 的刷新率，每 `16.6ms` 才会更新一次。
- 然后判断是否有 `resize` 或者 `scroll` 事件，有的话会去触发事件，所以 `resize` 和 `scroll` 事件也是至少 `16ms` 才会触发一次，并且自带节流功能。
- 判断是否触发了 `media query`
- 更新动画并且发送事件
- 判断是否有全屏操作事件
- 执行 `requestAnimationFrame` 回调
- 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，但是兼容性不好 更新界面
- 以上就是一帧中可能会做的事情。如果在一帧中有空闲时间，就会去执行 `requestIdleCallback` 回调

21.5 减少重绘和回流

1. 使用 `transform` 替代 `top`

```
<div class="test"></div>
<style>
```

html

```
.test {  
  position: absolute;  
  top: 10px;  
  width: 100px;  
  height: 100px;  
  background: red;  
}  
</style>  
<script>  
  setTimeout(() => {  
    // 引起回流  
    document.querySelector('.test').style.top = '100px'  
  }, 1000)  
</script>
```

2. 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）
3. 不要把节点的属性值放在一个循环里当成循环里的变量

```
for(let i = 0; i < 1000; i++) {  
  // 获取 offsetTop 会导致回流，因为需要去获取正确的值  
  console.log(document.querySelector('.test').style.offsetTop)  
}
```

js

4. 不要使用 `table` 布局，可能很小的一个小改动会造成整个 `table` 的重新布局
5. 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`
6. `CSS` 选择符从右往左匹配查找，避免节点层级过多
7. 将频繁重绘或者回流的节点设置为图层，图层能够阻止该节点的渲染行为影响别的节点。比如对于 `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

22 安全防范

22.1 XSS

涉及面试题：什么是 **XSS** 攻击？如何防范 **XSS** 攻击？什么是 **CSP** ？

- **XSS** 简单点来说，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



- 这种情况如果前后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

```
<!-- http://www.domain.com?name=<script>alert(1)</script> -->
<div>{{name}}</div>
```

html

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，因为我不能确保用户都使用了该类浏览器。

对于 **XSS** 攻击来说，通常有两种方式可以用来防御。

1. 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

```
function escape(str) {  
  str = str.replace(/&/g, '&amp;');  
  str = str.replace(/</g, '&lt;');  
  str = str.replace(/>/g, '&gt;');  
  str = str.replace(/"/g, '&quot;');  
  str = str.replace(/'/g, '&#39;');  
  str = str.replace(/`/g, '&#96;');  
  str = str.replace(/\\/g, '&#x2F;');  
  return str  
}
```

js

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

```
// -> &lt;script&gt;alert(1)&lt;&#x2F;script&gt;  
escape('<script>alert(1)</script>')
```

js

但是对于显示富文本来讲，显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

```
const xss = require('xss')  
let html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>'  
// -> <h1>XSS Demo</h1>&lt;script&gt;alert("xss");&lt;/script&gt;  
console.log(html)
```

js

以上示例使用了 `js-xss` 来实现，可以看到在输出中保留了 `h1` 标签且过滤了 `script` 标签

2. 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来尽量减少 **XSS** 攻击。

通常可以通过两种方式来开启 CSP：

- 设置 **HTTP Header** 中的 **Content-Security-Policy**
- 设置 **meta** 标签的方式 `<meta http-equiv="Content-Security-Policy">`

这里以设置 **HTTP Header** 来举例

只允许加载本站资源

```
Content-Security-Policy: default-src 'self'
```

只允许加载 HTTPS 协议图片

```
Content-Security-Policy: img-src https://*
```

允许加载任何来源框架

```
Content-Security-Policy: child-src 'none'
```

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。

22.2 CSRF

涉及面试题：什么是 **CSRF** 攻击？如何防范 **CSRF** 攻击？

CSRF 中文名为跨站请求伪造。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途径自动发起请求。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

```

```

html

那么你是否会想到使用 **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

- **Get** 请求不对数据进行修改
- 不让第三方网站访问到用户 **Cookie**
- 阻止第三方网站请求接口
- 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** 的攻击，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

22.3 点击劫持

涉及面试题：什么是点击劫持？如何防范点击劫持？

点击劫持是一种视觉欺骗的攻击手段。攻击者将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，在页面中透出一个按钮诱导用户点击



对于这种攻击方式，推荐防御的方法有两种

1. X-FRAME-OPTIONS

X-FRAME-OPTIONS 是一个 **HTTP** 响应头，在现代浏览器有一个很好的支持。这个 HTTP 响应头 就是为了防御用 **iframe** 嵌套的点击劫持攻击。

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

2. JS 防御

对于某些远古浏览器来说，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。

html

```
<head>
  <style id="click-jack">
    html {
      display: none !important;
    }
  </style>
</head>
<body>
  <script>
    if (self == top) {
      var style = document.getElementById('click-jack')
      document.body.removeChild(style)
    } else {
      top.location = self.location
    }
  </script>
</body>
```

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

23 从 V8 中看 JS 性能优化

注意：该知识点属于性能优化领域。

23.1 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且给出一个完整的报告

上图是给掘金首页测试性能后给出的一个报告，可以看到报告中分别为性能、体验、SEO 都给出了打分，并且每一个指标都有详细的评估

评估结束后，工具还提供了一些建议便于我们提高这个指标的分数

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。

在这张图中，我们可以详细的看到每个时间段中浏览器在处理什么事情，哪个过程最消耗时间，便于我们更加详细的了解性能瓶颈

23.2 JS 性能优化

JS 是编译型还是解释型语言其实并不固定。首先 **JS** 需要有引擎才能运行起来，无论是浏览器还是在 **Node** 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，他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，只是为了提高代码执行性能，所以总的来说 **JS** 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 代码首先会解析为抽象语法树（AST），然后通过解释器或者编译器转化为 Bytecode 或者 Machine Code

从上图中我们可以发现，JS 会首先被解析为 AST，解析的过程其实是略慢的。代码越多，解析的过程也就耗费越长，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，这个我们下面再谈

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），因为这样会造成函数的重复解析

```
function test1() {  
  // 会被重复解析  
  function test2() {}  
}
```

js

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。简单的加法运算代码，内部就需要考虑好几种规则，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要增加很多判断逻辑，降低运行效率。

```
function test(x) {  
  return x + x  
}
```

js

```
test(1)  
test(2)
```



```
test(3)
test(4)
```

- 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多的编译为 `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。
- 那么你可能会有一个疑问，到底优化前后有多少的提升呢，接下来我们就来实践测试一下到底有多少的提升

```
const { performance, PerformanceObserver } = require('perf_hooks')  
  
function test(x) {  
  return x + x  
}  
  
// node 10 中才有 PerformanceObserver  
// 在这之前的 node 版本可以直接使用 performance 中的 API  
const obs = new PerformanceObserver((list, observer) => {  
  console.log(list.getEntries())  
  observer.disconnect()  
})  
obs.observe({ entryTypes: ['measure'], buffered: true })  
  
performance.mark('start')  
  
let number = 10000000  
// 不优化代码  
%NeverOptimizeFunction(test)  
  
while (number--) {  
  test(1)  
}  
  
performance.mark('end')  
performance.measure('test', 'start', 'end')
```

以上代码中我们使用了 `performance API`，这个 `API` 在性能测试上十分好用。不仅可以用来测量代码的执行时间，还能用来测量各种网络连接中的时间消耗等等，并且这个 `API` 也可以在浏览器中使

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书写代码，引擎底层就能帮助我们自动优化代码。

另外，编译器还有个骚操作 **Lazy-Compile**，当函数没有被执行的时候，会对函数进行一次预解析，直到代码被执行以后才会被解析编译。对于上述代码来说，**test** 函数需要被预解析一次，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是多余的，那么有什么办法能够让代码不被预解析呢？

```
(function test(obj) {  
  return x + x  
})
```

js

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做。我们可以通过 **optimize-js** 实现这个功能，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，还是需要测试过相关内容的。

其实很简单，我们只需要给函数套上括号就可以了

24 性能优化

总的来说性能优化这个领域的很多内容都很碎片化，这一章节我们将来学习这些碎片化的内容。

24.1 图片优化

计算图片大小

对于一张 $100 * 100$ 像素的图片来说，图像上有 10000 个像素点，如果每个像素的值是 **RGBA** 存储的话，那么也就是说每个像素有 4 个通道，每个通道 1 个字节（ 8 位 = 1 个字节），所以该图片大小大概为 $39KB$ （ $10000 * 1 * 4 / 1024$ ）。

- 但是在实际项目中，一张图片可能并不需要使用那么多颜色去显示，我们可以通过减少每个像素的调色板来相应缩小图片的大小。
- 了解了如何计算图片大小的知识，那么对于如何优化图片，想必大家已经有 2 个思路了：

1. 减少像素点
2. 减少每个像素点能够显示的颜色

24.2 图片加载优化

- 不用图片。很多时候会使用到很多修饰类图片，其实这类修饰图片完全可以用 **CSS** 去代替。
- 对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 **CDN** 加载，可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片。
- 小图使用 **base64** 格式
- 将多个图标文件整合到一张图片中（雪碧图）
- 选择正确的图片格式：
 - 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
 - 小图使用 **PNG**，其实对于大部分图标这类图片，完全可以使用 **SVG** 代替
 - 照片使用 **JPEG**

24.3 DNS 预解析

DNS 解析也是需要时间的，可以通过预解析的方式来预先获得域名所对应的 **IP**。

```
<link rel="dns-prefetch" href="//blog.poetries.top">
```

html

24.4 节流

考虑一个场景，滚动事件中会发起网络请求，但是我们并不希望用户在滚动过程中一直发起请求，而是隔一段时间发起一次，对于这种情况我们就可以使用节流。

理解了节流的用途，我们就来实现下这个函数

```
js

// func是用户传入需要防抖的函数
// wait是等待时间
const throttle = (func, wait = 50) => {
  // 上一次执行该函数的时间
  let lastTime = 0
  return function(...args) {
    // 当前时间
    let now = +new Date()
    // 将当前时间和上一次执行函数时间对比
    // 如果差值大于设置的等待时间就执行函数
    if (now - lastTime > wait) {
      lastTime = now
      func.apply(this, args)
    }
  }
}

setInterval(
  throttle(() => {
    console.log(1)
  }, 500),
  1
)
```

24.5 防抖

考虑一个场景，有一个按钮点击会触发网络请求，但是我们并不希望每次点击都发起网络请求，而是当用户点击按钮一段时间后没有再次点击的情况才去发起网络请求，对于这种情况我们就可以使用防抖。

理解了防抖的用途，我们就来实现下这个函数

```
// func是用户传入需要防抖的函数
// wait是等待时间
const debounce = (func, wait = 50) => {
  // 缓存一个定时器id
  let timer = 0
  // 这里返回的函数是每次用户实际调用的防抖函数
  // 如果已经设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，延迟执行用户传入的方法
  return function(...args) {
    if (timer) clearTimeout(timer)
    timer = setTimeout(() => {
      func.apply(this, args)
    }, wait)
  }
}
```

24.6 预加载

- 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，这时候就可以使用预加载。
- 预加载其实是声明式的 `fetch`，强制浏览器请求资源，并且不会阻塞 `onload` 事件，可以使用以下代码开启预加载

```
<link rel="preload" href="http://blog.poetries.top">
```

预加载可以一定程度上降低首屏的加载时间，因为可以将一些不影响首屏但重要的文件延后加载，唯一缺点就是兼容性不好。

24.7 预渲染

可以通过预渲染将下载的文件预先在后台渲染，可以使用以下代码开启预渲染

```
<link rel="prerender" href="http://blog.poetries.top">
```

预渲染虽然可以提高页面的加载速度，但是要确保该页面大概率会被用户在之后打开，否则就是白白浪费资源去渲染。

24.8 懒执行

懒执行就是将某些逻辑延迟到使用时再计算。该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行。懒执行需要唤醒，一般可以通过定时器或者事件的调用来唤醒。

24.9 懒加载

- 懒加载就是将不关键的资源延后加载。
- 懒加载的原理就是只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域内需要加载的东西。对于图片来说，先设置图片标签的 `src` 属性为一张占位图，将真实的图片资源放入一个自定义属性中，当进入自定义区域时，就将自定义属性替换为 `src` 属性，这样图片就会去下载资源，实现了图片懒加载。
- 懒加载不仅可以用于图片，也可以使用在别的资源上。比如进入可视区域才开始播放视频等等。

24.10 CDN

`CDN` 的原理是尽可能的在各个地方分布机房缓存数据，这样即使我们的根服务器远在国外，在国内的用户也可以通过国内的机房迅速加载资源。

因此，我们可以将静态资源尽量使用 `CDN` 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 `CDN` 域名。并且对于 `CDN` 加载静态资源需要注意 `CDN` 域名要与主站不同，否则每次请求都会带上主站的 `Cookie`，白白消耗流量

25 Webpack 性能优化

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频考点：

- 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

25.1 减少 Webpack 打包时间

1. 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** 继续进行转变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: [  
      {  
        // js 文件才使用 babel  
        test: /\.js$/,  
        loader: 'babel-loader',  
        // 只在 src 文件夹下查找  
        include: [resolve('src')],  
        // 不会去查找的路径  
        exclude: /node_modules/  
      }  
    ]  
  }  
}
```

js

对于 **Babel** 来说，我们肯定是希望只作用在 **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

- 当然这样做还不够，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'
```

js

2. 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的，这样就能充分利用系统资源来加快打包效率了

```
module: {
  loaders: [
    {
      test: /\.js$/,
      include: [resolve('src')],
      exclude: /node_modules/,
      // id 后面的内容对应下面
      loader: 'happypack/loader?id=happybabel'
    }
  ]
},
plugins: [
  new HappyPack({
    id: 'happybabel',
    loaders: ['babel-loader?cacheDirectory'],
    // 开启 4 个线程
    threads: 4
  })
]
```

js

3.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离成单独文件的优化方案。

接下来我们就来学习如何使用 `DllPlugin`

```
// 单独配置在一个文件中
// webpack.dll.conf.js
const path = require('path')
const webpack = require('webpack')
module.exports = {
  entry: {
    // 想统一打包的类库
    vendor: ['react']
  },
  output: {
    path: path.join(__dirname, 'dist'),
    filename: '[name].dll.js',
    library: '[name]-[hash]'
  },
  plugins: [
    new webpack.DllPlugin({
      // name 必须和 output.library 一致
      name: '[name]-[hash]',
      // 该属性需要与 DllReferencePlugin 中一致
      context: __dirname,
      path: path.join(__dirname, 'dist', '[name]-manifest.json')
    })
  ]
}
```

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 = {
  // ...省略其他配置
  plugins: [
    new webpack.DllReferencePlugin({
      context: __dirname,
      // manifest 就是之前打包出来的 json 文件
      manifest: require('./dist/vendor-manifest.json'),
    })
  ]
}
```

4. 代码压缩

在 Webpack3 中，我们一般使用 UglifyJS 来压缩代码，但是这个单线程运行的，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 Webpack4 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做的性能优化方案，当然我们不止可以压缩 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。

5. 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：用来表明文件后缀列表，默认查找顺序是 ['.js', '.json']，如果你的导入文件没有添加后缀就会按照这个顺序查找文件。我们应该尽可能减少后缀列表长度，然后将出现频率高的后缀排在前面
- resolve.alias：可以通过别名的方式来映射一个路径，能让 Webpack 更快找到路径
- module.noParse：如果你确定一个文件下没有其他依赖，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

25.2 减少 Webpack 打包后的文件体积

1. 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。如果我们将这些页面全部打包进一个 JS 文件的话，虽然将多个请求合并了，但是同样也加载了很多并不需要的代码，耗费了更长的时间。那么为了首页能更快地呈现给用户，我们肯定是希望首页能加载的文件体积越小越好，这时候我们就可以使用按需加载，将每个路由页面单独打包为一个文件。当然不仅仅路由可以按需加载，对于 loadash 这种大型类库同样可以使用这个功能。

按需加载的代码实现这里就不详细展开了，因为鉴于用的框架不同，实现起来都是不一样的。当然了，虽然他们的用法可能不同，但是底层的机制都是一样的。都是当使用的时候再去下载对应文件，返回一个 `Promise`，当 `Promise` 成功以后去执行回调。

2. Scope Hoisting

`Scope Hoisting` 会分析出模块之间的依赖关系，尽可能的把打包出来的模块合并到一个函数中去。

比如我们希望打包两个文件

```
// test.js
export const a = 1

// index.js
import { a } from './test.js'
```

js

对于这种情况，我们打包出来的代码会类似这样

```
[
  /* 0 */
  function (module, exports, require) {
    //...
  },
  /* 1 */
  function (module, exports, require) {
    //...
  }
]
```

js

但是如果我们使用 `Scope Hoisting` 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

```
[
  /* 0 */
```

js

```
function (module, exports, require) {  
  //...  
}  
]
```

这样的打包方式生成的代码明显比之前的少多了。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。

```
module.exports = {  
  optimization: {  
    concatenateModules: true  
  }  
}
```

js

3. 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'
```

js

- 对于以上情况，**test** 文件中的变量 **b** 如果没有在项目中用到，就不会被打包到文件中。
- 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。

26 实现小型打包工具

该工具可以实现以下两个功能

- 将 **ES6** 转换为 **ES5**
- 支持在 **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 **ES5**，所以我们首先需要安装一些 **Babel** 相关的工具

```
yarn add babylon 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 babylon = require('babylon')
const traverse = require('babel-traverse').default
const { transformFromAst } = require('babel-core')
```

js

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 = babylon.parse(content, {
    sourceType: 'module'
  })
  // 寻找当前文件的依赖关系
  const dependencies = []
  traverse(ast, {
    ImportDeclaration: ({ node }) => {
      dependencies.push(node.source.value)
    }
  })
  // 通过 AST 将代码转为 ES5
  const { code } = transformFromAst(ast, null, {
    presets: ['env']
  })
  return {
    filePath,
    dependencies,
    code
  }
}
```

js

```

    }
  }
}

```

- 首先我们传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 `babylon` 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 `ES5` 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]
  // 遍历所有文件依赖关系
  for (const asset of dependencies) {
    // 获得文件目录
    const dirname = path.dirname(asset.filePath)
    // 遍历当前文件依赖关系
    asset.dependencies.forEach(relativePath => {
      // 获得绝对路径
      const absolutePath = path.join(dirname, relativePath)
      // CSS 文件逻辑就是将代码插入到 `style` 标签中
      if (/\.css$/i.test(absolutePath)) {
        const content = fs.readFileSync(absolutePath, 'utf-8')
        const code = `
          const style = document.createElement('style')
          style.innerText = ${JSON.stringify(content).replace(/\\r\\n/g, '')}
          document.head.appendChild(style)
        `
        dependencies.push({
          filePath: absolutePath,
          relativePath,
          dependencies: [],
          code
        })
      } else {

```

js

```

// JS 代码需要继续查找是否有依赖关系
const child = readCode(absolutePath)
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，一开始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，我们先获得该文件对应的目录，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` 文件还是 `JS` 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，然后创建一个 `style` 标签，将代码插入进标签并且放入 `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
- 现在我们已经获取到了所有的依赖文件，接下来就是实现打包的功能了

```

function bundle(dependencies, entry) {
  let modules = ''
  // 构造函数参数，生成的结构为
  // { './entry.js': function(module, exports, require) { 代码 } }
  dependencies.forEach(dep => {
    const filePath = dep.relativePath || entry
    modules += ` '${filePath}': (
      function (module, exports, require) { ${dep.code} }
    ),`
  })
  // 构建 require 函数，目的是为了获取模块暴露出来的内容
  const result = `
    (function(modules) {
      function require(id) {
        const module = { exports : {} }
        modules[id](module, module.exports, require)
        return module.exports
      }
      require('${entry}')
    `

```

js

```

    })(({modules}))
  },
  // 当生成的内容写入到文件中
  fs.writeFileSync('./bundle.js', result)
}

```

这段代码需要结合着 **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 _a2 = _interopRequireDefault(_a)
function _interopRequireDefault(obj) {
  return obj && obj.__esModule ? obj : { default: obj }
}
console.log(_a2.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** 的代码，但是浏览器是不支持 **CommonJS** 的，所以如果这段代码需要在浏览器环境下运行的话，我们需要自己实现 **CommonJS** 相关的代码，这就是 **bundle** 函数做的大部分事情。

接下来我们再来逐行解析 bundle 函数

- 首先遍历所有依赖文件，构建出一个函数参数对象
- 对象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.export**
 - **require** 参数对应我们自己创建的 **require** 函数
- 接下来就是构造一个使用参数的函数了，函数做的事情很简单，就是内部创建一个 **require** 函数，然后调用 **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.export` 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(modules) {  
  function require(id) {  
    // 构造一个 CommonJS 导出代码  
    const module = { exports: {} }  
    // 去参数中获取文件对应的函数并执行  
    modules[id](module, module.exports, require)  
    return module.exports  
  }  
  require('./entry.js')  
})([{  
  './entry.js': function(module, exports, require) {  
    // 这里继续通过构造的 require 去找到 a.js 文件对应的函数  
    var _a = require('./a.js')  
    console.log(_a2.default)  
  },  
  './a.js': function(module, exports, require) {  
    var a = 1  
    // 将 require 函数中的变量 module 变成了这样的结构  
    // module.exports = 1  
    // 这样就能在外部取到导出的内容了  
    exports.default = a  
  }  
  // 省略  
}])
```

虽然实现这个工具只写了不到 100 行的代码，但是打包工具的核心原理就是这些了

- 找出入口文件所有的依赖关系
- 然后通过构建 CommonJS 代码来获取 exports 导出的内容

27 MVVM/虚拟DOM/前端路由

27.1 MVVM

涉及面试题：什么是 **MVVM**？比之 **MVC** 有什么区别？

首先先来说下 View 和 Model

- **View** 很简单，就是用户看到的视图
- **Model** 同样很简单，一般就是本地数据和数据库中的数据

基本上，我们写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展现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，然后将用户的输入通过接口写入到数据库中。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中，不同的人就产生了不同的看法，从此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 **MVC** 有一个巨大的缺陷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，不关心 **View** 如何处理数据，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** 复用这个 **ViewModel**。
- 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** 中还引入了一个隐式的 **Binder** 层，实现了 **View** 和 **ViewModel** 的绑定
- 同样以 **Vue** 框架来举例，这个隐式的 **Binder** 层就是 **Vue** 通过解析模板中的插值和指令从而实现 **View** 与 **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

27.2 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，为什么慢的原因以及在「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**

```
const ul = {  
  tag: 'ul',  
  props: {  
    class: 'list'  
  },  
  children: {  
    tag: 'li',  
    children: '1'  
  }  
}
```

js

上述代码对应的 **DOM** 就是

```
<ul class='list'>  
  <li>1</li>  
</ul>
```

html

- 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，难点在于如何判断新旧两个 **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颗树的差异，那么需要的时间复杂度会是 $O(n^3)$ ，这个复杂度肯定是不能接受的。于是

React 团队优化了算法，实现了 $O(n)$ 的复杂度来对比差异。实现 $O(n)$ 复杂度的关键就是只对比同层的节点，而不是跨层对比，这也是考虑到在实际业务中很少会去跨层的移动 **DOM** 元素。所以判断差异的算法就分为了两步

- 首先从上至下，从左往右遍历对象，也就是树的深度遍历，这一步中会给每个节点添加索引，便于最后渲染差异
- 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新旧节点的 `tagName` 是否相同，如果不相同的话就代表节点被替换了。如果没有更改 `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移除，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有移动。

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

```
// 假设这里模拟一个 ul，其中包含了 5 个 li
[1, 2, 3, 4, 5]
// 这里替换上面的 li
[1, 2, 5, 4]
```

js

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，我们如何去识别改动的是哪个节点呢？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志的，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。

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` 作为一个兼容层，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

27.3 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
- **History** 模式

1. Hash 模式

www.test.com/#/ 就是 **Hash URL**，当 **#** 后面的哈希值发生变化时，可以通过 **hashchange** 事件来监听到 **URL** 的变化，从而进行跳转页面，并且无论哈希值如何变化，服务端接收到的 **URL** 请求永远是 **www.test.com**

```
window.addEventListener('hashchange', () => {  
  // ... 具体逻辑  
})
```

js

Hash 模式相对来说更简单，并且兼容性也更好

2. 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**

- 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。

```
// 新增历史记录  
history.pushState(stateObject, title, URL)
```

js

```
// 替换当前历史记录  
history.replaceState(stateObject, title, URL)
```

当用户做出浏览器动作时，比如点击后退按钮时会触发 `popState` 事件

```
js  
window.addEventListener('popstate', e => {  
  // e.state 就是 pushState(stateObject) 中的 stateObject  
  console.log(e.state)  
})
```

两种模式对比

- `Hash` 模式只可以更改 `#` 后面的内容，`History` 模式可以通过 `API` 设置任意的同源 `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` 模式无需后端配置，并且兼容性好。`History` 模式在用户手动输入地址或者刷新页面的时候会发起 `URL` 请求，后端需要配置 `index.html` 页面用于匹配不到静态资源的时候

27.4 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` 写表单的方式没什么区别
- 改变数据方式不同，`Vue` 修改状态相比来说要简单许多，`React` 需要使用 `setState` 来改变状态，并且使用这个 `API` 也有一些坑点。并且 `Vue` 的底层使用了依赖追踪，页面更新渲染已经是最优的了，但是 `React` 还是需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
- `React` 需要使用 `JSX`，有一定的上手成本，并且需要一整套的工具链支持，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

28 Vue常考知识点

28.1 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，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挂载，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挂载完毕，才会执行根组件的挂载钩子。
- 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。
- 最后就是销毁组件的钩子函数 `beforeDestroy` 和 `destroyed`。前者适合移除事件、定时器等，否则可能会引起内存泄露的问题。然后进行一系列的销毁操作，如果有子组件的话，也会递归销毁子组件，所有子组件都销毁完毕后会执行根组件的 `destroyed` 钩子函数

28.2 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

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。

1. 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 `prop` 和名为 `input` 的事件。这种语法糖的方式是典型的双向绑定，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` 属性是个语法糖，可以很简单的实现子组件与父组件通信

html

```
<!--父组件中-->
<input :value.sync="value" />
<!--以上写法等同于-->
<input :value="value" @update:value="v => value = v"></comp>
<!--子组件中-->
<script>
  this.$emit('update:value', 1)
</script>
```

2. 兄弟组件通信

对于这种情况可以通过查找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` 查询到需要的组件实例，然后进行通信。

3. 跨多层次组件通信

对于这种情况可以使用 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。

假设有父组件 `A`，然后有一个跨多层级的子组件 `B`

js

```
// 父组件 A
export default {
  provide: {
    data: 1
  }
}
// 子组件 B
export default {
  inject: ['data'],
  mounted() {
```



```
// 无论跨几层都能获得父组件的 data 属性
console.log(this.data) // => 1
}
}
```

终极办法解决一切通信问题

只要你不怕麻烦，可以使用 **Vuex** 或者 **Event Bus** 解决上述所有的通信情况。

28.3 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。

```
// 创建组件构造器
let Component = Vue.extend({
  template: '<div>test</div>'
})
// 挂载到 #app 上
new Component().$mount('#app')
// 除了上面的方式，还可以用来扩展已有的组件
let SuperComponent = Vue.extend(Component)
new SuperComponent({
  created() {
    console.log(1)
  }
})
new SuperComponent().$mount('#app')
```

js

28.4 mixin 和 mixins 区别

mixin 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初始化的

```
Vue.mixin({
  beforeCreate() {
```

js

```
// ...逻辑
// 这种方式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 `mixin`，但是如果不滥用的话也是很有帮助的，比如可以全局混入封装好的 `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，就可以将这些逻辑剥离出来，通过 `mixins` 混入代码，比如上拉下拉加载数据这种逻辑等等。
- 另外需要注意的是 `mixins` 混入的钩子函数会先于组件内的钩子函数执行，并且在遇到同名选项的时候也会有选择性的进行合并，具体可以阅读 文档。

28.5 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别的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 `computer` 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', {
  // 深度遍历
  deep: true,
  // 立即触发
  immediate: true,
  // 执行的函数
  handler: function(val, oldVal) {}
})
var vm = new Vue({
  data: { a: 1 },
  computed: {
    aPlus: {
      // this.aPlus 时触发
      get: function () {
        return this.a + 1
      },
      // this.aPlus = 1 时触发
      set: function (v) {
        this.a = v - 1
      }
    }
  }
})
```

js

```
    }  
  }  
})
```

28.6 keep-alive 组件有什么作用

- 如果你需要在组件切换的时候，保存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。

28.7 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，后面只需要切换 `CSS`，`DOM` 还是一直保留着的。所以总的来说 `v-show` 在初始渲染时有更高的开销，但是切换开销很小，更适合于频繁切换的场景。
- `v-if` 的话就得说到 `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/挂载组件，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。

28.8 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 功底。

- 组件复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` 写成函数，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` 设置为对象还是函数都是可以的，因为 `new Vue()` 的方式是生成一个根组件，该组件不会复用，也就不存在共享 `data` 的情况了

以下是进阶部分

28.9 响应式原理

Vue 内部使用了 `Object.defineProperty()` 来实现数据响应式，通过这个方法可以监听到 `set` 和 `get` 的事件

```
var data = { name: 'poetries' }
observe(data)
let name = data.name // -> get value
data.name = 'yyy' // -> change value

function observe(obj) {
  // 判断类型
  if (!obj || typeof obj !== 'object') {
    return
  }
  Object.keys(obj).forEach(key => {
    defineReactive(obj, key, obj[key])
  })
}

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  Object.defineProperty(obj, key, {
    // 可枚举
    enumerable: true,
    // 可配置
    configurable: true,
    // 自定义函数
    get: function reactiveGetter() {
      console.log('get value')
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
    }
  })
}
```

js

以上代码简单的实现了如何监听数据的 `set` 和 `get` 的事件，但是仅仅如此是不够的，因为自定义的函数一开始是不会执行的。只有先执行了依赖收集，从能在属性更新的时候派发更新，所以接下来我们需要先触发依赖收集

```
<div>
  {{name}}
</div>
```

html

- 在解析如上模板代码时，遇到 `{{name}}` 就会进行依赖收集。
- 接下来我们先来实现一个 `Dep` 类，用于解耦属性的依赖收集和派发更新操作

```
// 通过 Dep 解耦属性的依赖和更新操作
class Dep {
  constructor() {
    this.subs = []
  }
  // 添加依赖
  addSub(sub) {
    this.subs.push(sub)
  }
  // 更新
  notify() {
    this.subs.forEach(sub => {
      sub.update()
    })
  }
}
// 全局属性，通过该属性配置 Watcher
Dep.target = null
```

js

以上的代码实现很简单，当需要依赖收集的时候调用 `addSub`，当需要派发更新的时候调用 `notify`。

接下来我们先来简单的了解下 `Vue` 组件挂载时添加响应式的过程。在组件挂载时，会先对所有需要的属性调用 `Object.defineProperty()`，然后实例化 `Watcher`，传入组件更新的回调。在实例化过程中，会对模板中的属性进行求值，触发依赖收集。

因为这一小节主要目的是学习响应式原理的细节，所以接下来的代码会简略的表达触发依赖收集时的操作。

```
class Watcher {
  constructor(obj, key, cb) {
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
    Dep.target = this
    this.cb = cb
    this.obj = obj
    this.key = key
    this.value = obj[key]
    Dep.target = null
  }
  update() {
    // 获得新值
    this.value = this.obj[this.key]
    // 调用 update 方法更新 Dom
    this.cb(this.value)
  }
}
```

js

以上就是 **Watcher** 的简单实现，在执行构造函数的时候将 **Dep.target** 指向自身，从而使得收集到了对应的 **Watcher**，在派发更新的时候取出对应的 **Watcher** 然后执行 **update** 函数。

接下来，需要对 **defineReactive** 函数进行改造，在自定义函数中添加依赖收集和派发更新相关的代码

```
function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  let dp = new Dep()
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      // 将 Watcher 添加到订阅
      if (Dep.target) {
        dp.addSub(Dep.target)
      }
    }
  })
}
```

js

```

    }
    return val
  },
  set: function reactiveSetter(newVal) {
    console.log('change value')
    val = newVal
    // 执行 watcher 的 update 方法
    dp.notify()
  }
})
}

```

以上所有代码实现了一个简易的数据响应式，核心思路就是手动触发一次属性的 **getter** 来实现依赖收集。

现在我们就来测试下代码的效果，只需要把所有的代码复制到浏览器中执行，就会发现页面的内容全部被替换了

```

var data = { name: 'poetries' }
observe(data)
function update(value) {
  document.querySelector('div').innerText = value
}
// 模拟解析到 `{{name}}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'

```

js

28.9.1 Object.defineProperty 的缺陷

- 以上已经分析完了 **Vue** 的响应式原理，接下来说一点 **Object.defineProperty** 中的缺陷。
- 如果通过下标方式修改数组数据或者给对象新增属性并不会触发组件的重新渲染，因为 **Object.defineProperty** 不能拦截到这些操作，更精确的来说，对于数组而言，大部分操作都是拦截不到的，只是 **Vue** 内部通过重写函数的方式解决了这个问题。
- 对于第一个问题，**Vue** 提供了一个 **API** 解决

```

export function set (target: Array<any> | Object, key: any, val: any): any
// 判断是否为数组且下标是否有效
if (Array.isArray(target) && isValidArrayIndex(key)) {

```

js

```

    // 调用 splice 函数触发派发更新
    // 该函数已被重写
    target.length = Math.max(target.length, key)
    target.splice(key, 1, val)
    return val
  }
  // 判断 key 是否已经存在
  if (key in target && !(key in Object.prototype)) {
    target[key] = val
    return val
  }
  const ob = (target: any).__ob__
  // 如果对象不是响应式对象，就赋值返回
  if (!ob) {
    target[key] = val
    return val
  }
  // 进行双向绑定
  defineReactive(ob.value, key, val)
  // 手动派发更新
  ob.dep.notify()
  return val
}

```

对于数组而言，Vue 内部重写了以下函数实现派发更新

```

// 获得数组原型
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// 重写以下函数
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
methodsToPatch.forEach(function (method) {
  // 缓存原生函数
  const original = arrayProto[method]
  // 重写函数

```

js


```
def(arrayMethods, method, function mutator (...args) {  
  // 先调用原生函数获得结果  
  const result = original.apply(this, args)  
  const ob = this.__ob__  
  let inserted  
  // 调用以下几个函数时，监听新数据  
  switch (method) {  
    case 'push':  
    case 'unshift':  
      inserted = args  
      break  
    case 'splice':  
      inserted = args.slice(2)  
      break  
  }  
  if (inserted) ob.observeArray(inserted)  
  // 手动派发更新  
  ob.dep.notify()  
  return result  
})  
})
```

28.9.2 编译过程

想必大家在使用 Vue 开发的过程中，基本都是使用模板的方式。那么你有过「模板是怎么在浏览器中运行的」这种疑虑嘛？

- 首先直接把模板丢到浏览器中肯定是不能运行的，模板只是为了方便开发者进行开发。
Vue 会通过编译器将模板通过几个阶段最终编译为 `render` 函数，然后通过执行 `render` 函数生成 `Virtual DOM` 最终映射为真实 `DOM`。
- 接下来我们就来学习这个编译的过程，了解这个过程中大概发生了什么事情。这个过程其中又分为三个阶段，分别为：

- 将模板解析为 `AST`
- 优化 `AST`
- 将 `AST` 转换为 `render` 函数

在第一个阶段中，最主要的事情还是通过各种各样的正则表达式去匹配模板中的内容，然后将内容提取出来做各种逻辑操作，接下来会生成一个最基本的 `AST` 对象

```
{
  // 类型
  type: 1,
  // 标签
  tag,
  // 属性列表
  attrsList: attrs,
  // 属性映射
  attrsMap: makeAttrsMap(attrs),
  // 父节点
  parent,
  // 子节点
  children: []
}
```

- 然后会根据这个最基本的 AST 对象中的属性，进一步扩展 AST。
- 当然在这一阶段中，还会进行其他的一些判断逻辑。比如说对比前后开闭标签是否一致，判断根组件是否只存在一个，判断是否符合 HTML5 Content Model 规范等等问题。
- 接下来就是优化 AST 的阶段。在当前版本下，Vue 进行的优化内容其实还是不多的。只是对节点进行了静态内容提取，也就是将永远不会变动的节点提取了出来，实现复用 Virtual DOM，跳过对比算法的功能。在下一个大版本中，Vue 会在优化 AST 的阶段继续发力，实现更多的优化功能，尽可能的在编译阶段压榨更多的性能，比如说提取静态的属性等等优化行为。
- 最后一个阶段就是通过 AST 生成 render 函数了。其实这一阶段虽然分支有很多，但是最主要的目的就是遍历整个 AST，根据不同的条件生成不同的代码罢了。

28.9.3 NextTick 原理分析

nextTick 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于获得更新后的 DOM。

- 在 Vue 2.4 之前都是使用的 microtasks，但是 microtasks 的优先级过高，在某些情况下可能会出现比事件冒泡更快的情况，但如果都使用 macrotasks 又可能会出现渲染的性能问题。所以在新版本中，会默认使用 microtasks，但在特殊情况下会使用 macrotasks，比如 v-on。
- 对于实现 macrotasks，会先判断是否能使用 setImmediate，不能的话降级为 MessageChannel，以上都不行的话就使用 setTimeout。

```
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  macroTimerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else if (
  typeof MessageChannel !== 'undefined' &&
  (isNative(MessageChannel) ||
    // PhantomJS
    MessageChannel.toString() === '[object MessageChannelConstructor]')
) {
  const channel = new MessageChannel()
  const port = channel.port2
  channel.port1.onmessage = flushCallbacks
  macroTimerFunc = () => {
    port.postMessage(1)
  }
} else {
  macroTimerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
```

以上代码很简单，就是判断能不能使用相应的 API

29 React常考知识点

29.1 生命周期

在 V16 版本中引入了 Fiber 机制。这个机制一定程度上的影响了部分生命周期的调用，并且也引入了新的 2 个 API 来解决问题

在之前的版本中，如果你拥有一个很复杂的复合组件，然后改动了最上层组件的 state，那么调用栈可能会很长

- 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 就是为了解决该问题而生

- **Fiber** 本质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去分段计算更新
- 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** 必须渲染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，会一直更新界面直到完成。

1. Reconciliation 阶段

- **componentWillMount**
- **componentWillReceiveProps**
- **shouldComponentUpdate**
- **componentWillUpdate**

2. Commit 阶段

- **componentDidMount**
- **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**。由此对于 **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 {  
  // Initialize state in constructor,  
  // Or with a property initializer.  
  state = {};  
  
  static getDerivedStateFromProps(nextProps, prevState) {  
    if (prevState.someMirroredValue !== nextProps.someValue) {  
      js
```

```

    return {
      derivedData: computeDerivedState(nextProps),
      someMirroredValue: nextProps.someValue
    };
  }

  // Return null to indicate no change to state.
  return null;
}
}

```

`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

更多详情 <http://blog.poetries.top/2018/11/18/react-lifecircle>

29.2 setState

- `setState` 在 `React` 中是经常使用的一个 `API`，但是它存在的一些问题经常会导致初学者出错，核心原因就是因为这个 `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() {
  // 初始化 `count` 为 0
  console.log(this.state.count) // -> 0
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  console.log(this.state.count) // -> 0
}

```

js

- 第一，两次的打印都为 `0`，因为 `setState` 是个异步 `API`，只有同步代码运行完毕才会执行。`setState` 异步的原因我认为在于，`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过程。

```
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` 使得 `count` 为 3

```
handle() {
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.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((prevState) => ({ count: prevState.count + 1 })), () => {
    console.log(this.state)
  })
}
```

更多详情 <http://blog.poetries.top/2018/12/20/react-setState>

29.3 性能优化

- 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地浅比较一下当前 `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。一般来说不推荐完整地对比当前 `state` 和之前的 `state` 是否相同，因为组件更新触发可能会很频繁，这样的完整对比性能开销会有点大，可能会造成得不偿失的情况。

- 当然如果真的想完整对比当前 `state` 和之前的 `state` 是否相同，并且不影响性能也是行得通的，可以通过 `immutable` 或者 `immer` 这些库来生成不可变对象。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，同时也很推荐阅读下 `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() {  
    return (  
      <div>  
        PureComponent  
      </div>  
    )  
  }  
}
```

js

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

```
const Test = React.memo(() => (  
  <div>  
    PureComponent  
  </div>  
))
```

js

通过这种方式我们就可以既实现了 `shouldComponentUpdate` 的浅比较，又能够使用函数组件

29.4 通信

1. 父子通信

- 父组件通过 `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`，而是必须通过调用父组件函数的方式告知父组件修改数据。

2. 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

3. 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，可以在开始就传入值
const StateContext = React.createContext()
class Parent extends React.Component {
  render () {
    return (
      // value 就是传入 Context 中的值
      <StateContext.Provider value='yck'>
        <Child />
      </StateContext.Provider>
    )
  }
}
class Child extends React.Component {
  render () {
    return (
      <ThemeContext.Consumer>
        // 取出值
        {context => (
          name is { context }
        )}
      </ThemeContext.Consumer>
    );
  }
}
```

js

4. 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上述所有的通信情况

29.5 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  
}
```

js

现在如果我想给这个 `add` 函数添加一个输出结果的功能，那么你可能会考虑我直接使用 `console.log` 不就实现了么。说的没错，但是如果我们想做的更加优雅并且容易复用和扩展，我们可以这样做

```
function withLog (fn) {  
  function wrapper(a, b) {  
    const result = fn(a, b)  
    console.log(result)  
    return result  
  }  
  return wrapper  
}  
const withLogAdd = withLog(add)  
withLogAdd(1, 2)
```

js

- 其实这个做法在函数式编程里称之为高阶函数，大家都知道 `React` 的思想中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传入的组件，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` 的方式就不能使用了，并且其实 `mixins` 也是存在一些问题的，比如
 1. 隐含了一些依赖，比如我在组件中写了某个 `state` 并且在 `mixin` 中使用了，就这存在了一个依赖关系。万一下次别人要移除它，就得去 `mixin` 中查找依赖

2. 多个 `mixin` 中可能存在相同命名的函数，同时代码组件中也不能出现相同命名的函数，否则就是重写了，其实我一直觉得命名真的是一件麻烦事。。
3. 雪球效应，虽然我一个组件还是使用着同一个 `mixin`，但是一个 `mixin` 会被多个组件使用，可能会存在需求使得 `mixin` 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）

29.6 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) => (  
    <span onClick={handleClick} key={index}>{index}</span>  
  ))  
})
```

js

- 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` 上写的事件并没有绑定在对应的真实 `DOM` 上，而是通过事件代理的方式，将所有的事件都统一绑定在了 `document` 上。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。
- 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`）。因此我们如果不想要事件冒泡的话，调用 `event.stopPropagation` 是无效的，而应该调用 `event.preventDefault`

那么实现合成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：

1. 合成事件首先抹平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
2. 对于原生浏览器事件来说，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。但是对于合成事件来说，有一个事件池专门来管理它们的创建和销毁，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，就会销毁事件对象上的属性，从而便于下次复用事件对象。

30 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

30.1 页面埋点

页面埋点应该是大家最常写的监控了，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，一般的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，可以自主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。

30.2 性能监控

- 性能监控可以很好的帮助开发者了解在各种真实环境下，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')` 这行代码就行了。对，你没看错，一行代码我们就可以获得页面中各种详细的性能相关信息

我们可以发现这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

30.3 异常监控

- 对于异常监控来说，以下两种监控是必不可少的，分别是代码报错以及接口异常上报。
 - 对于代码运行错误，通常的办法是使用 `window.onerror` 拦截报错。该方法能拦截到大部分的详细报错信息，但是也有例外
1. 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
 2. 对于某些浏览器可能不会显示调用栈信息，这种情况可以通过 `arguments.callee.caller` 来做栈递归
- 对于异步代码来说，可以使用 `catch` 的方式捕获错误。比如 `Promise` 可以直接使用 `catch` 函数，`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 `img` 标签的 `src` 发起一个请求。
 - 另外接口异常就相对来说简单了，可以列举出出错的状态码。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有哪些接口出现了大面积的报错，以便迅速修复问题。

31 TCP/UDP

31.1 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，并且面试中经常会提问的也是关于这两个协议的区别

常考面试题：UDP 与 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传递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

1. 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拼接操作。

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 头标识下是 UDP 协议，然后就传递给网络层了 在接收端，网络层将数据传递给传输层，UDP 只去除 IP 报文头就传递给应用层，不会任何拼接操作

2. 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传递什么数据，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接收到数据了。
- 再者网络环境时好时坏，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

3. 高效

- 虽然 UDP 协议不是那么的可靠，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，需要保证数据不丢失且有序到达。
- 因此 UDP 的头部开销小，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 头部包含了以下几个数据

- 两个十六位的端口号，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

4. 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。

5. 适合使用的场景

UDP 虽然对比 TCP 有很多缺点，但是正是因为这些缺点造就了它高效的特性，在很多实时性要求高的地方都可以看到 UDP 的身影。

5.1 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，就会停止下来直到对端收到这个数据。这种问题在网络条件不错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失的老旧画面，所以 TCP 在这种情况下无用武之地，只会降低用户体验。

5.2 王者荣耀

- 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再传输旧的游戏画面是肯定不能接受的，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需要的，而不是老旧的画面，否则角色都不知道死多少次了。

31.2 TCP

常考面试题：UDP 与 TCP 的区别是什么？

TCP 基本是和 UDP 反着来，建立连接断开连接都需要先需要进行握手。在传输数据的过程中，通过各种算法保证数据的可靠性，当然带来的问题就是相比 UDP 来说不那么的高效

1. 头部

从这个图上我们就可以发现 TCP 头部比 UDP 头部复杂的多

对于 TCP 头部来说，以下几个字段是很重要的

- **Sequence number**，这个序号保证了 TCP 传输的报文都是有序的，对端可以通过序号顺序的拼接报文
- **Acknowledgement Number**，这个序号表示数据接收端期望接收的下一个字节的编号是多少，同时也表示上一个序号的数据已经收到
- **Window Size**，窗口大小，表示还能接收多少字节的数据，用于流量控制
- **标识符**
 - **URG=1**：该字段为一表示本数据报的数据部分包含紧急信息，是一个高优先级数据报文，此时紧急指针有效。紧急数据一定位于当前数据包数据部分的最前面，紧急指针标明了紧急数据的尾部。
 - **ACK=1**：该字段为一表示确认号字段有效。此外，TCP 还规定在连接建立后传送的所有报文段都必须把 ACK 置为一。
 - **PSH=1**：该字段为一表示接收端应该立即将数据 push 给应用层，而不是等到缓冲区满后再提交。
 - **RST=1**：该字段为一表示当前 TCP 连接出现严重问题，可能需要重新建立 TCP 连接，也可以用于拒绝非法的报文段和拒绝连接请求。
 - **SYN=1**：当 SYN=1，ACK=0 时，表示当前报文段是一个连接请求报文。当 SYN=1，ACK=1 时，表示当前报文段是一个同意建立连接的应答报文。
 - **FIN=1**：该字段为一表示此报文段是一个释放连接的请求报文。

2. 状态机

TCP 的状态机是很复杂的，并且与建立断开连接时的握手息息相关，接下来就来详细描述下两种握手

在这之前需要了解一个重要的性能指标 **RTT**。该指标表示发送端发送数据到接收到对端数据所需的往返时间

2.1. 建立连接三次握手

- 首先假设主动发起请求的一端称为客户端，被动连接的一端称为服务端。不管是客户端还是服务端，**TCP** 连接建立完后都能发送和接收数据，所以 **TCP** 是一个全双工的协议。
- 起初，两端都为 **CLOSED** 状态。在通信开始前，双方都会创建 **TCB**。服务器创建完 **TCB** 后便进入 **LISTEN** 状态，此时开始等待客户端发送数据

第一次握手

客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端便进入 **SYN-SENT** 状态

第二次握手

服务端收到连接请求报文段后，如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 **SYN-RECEIVED** 状态

第三次握手

- 当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。客户端发完这个报文段后便进入 **ESTABLISHED** 状态，服务端收到这个应答后也进入 **ESTABLISHED** 状态，此时连接建立成功。
- PS：第三次握手中可以包含数据，通过快速打开（**TFO**）技术就可以实现这一功能。其实只要涉及到握手的协议，都可以使用类似 **TFO** 的方式，客户端和服务端存储相同的 **cookie**，下次握手时发出 **cookie** 达到减少 **RTT** 的目的。

常考面试题：为什么 **TCP** 建立连接需要三次握手，明明两次就可以建立起连接

- 因为这是为了防止出现失效的连接请求报文段被服务端接收的情况，从而产生错误。
- 可以想象如下场景。客户端发送了一个连接请求 **A**，但是因为网络原因造成了超时，这时 **TCP** 会启动超时重传的机制再次发送一个连接请求 **B**。此时请求顺利到达服务端，服务端应答完就建立了请求，然后接收数据后释放了连接。

假设这时候连接请求 **A** 在两端关闭后终于抵达了服务端，那么此时服务端会认为客户端又需要建立 **TCP** 连接，从而应答了该请求并进入 **ESTABLISHED** 状态。但是客户端其实是 **CLOSED** 的状态，那么就会导致服务端一直等待，造成资源的浪费。

PS：在建立连接中，任意一端掉线，**TCP** 都会重发 **SYN** 包，一般会重试五次，在建立连接中可能会遇到 **SYN Flood** 攻击。遇到这种情况你可以选择调低重试次数或者干脆在不能处理的情况下拒绝请求

2.2. 断开链接四次握手

TCP 是全双工的，在断开连接时两端都需要发送 **FIN** 和 **ACK**

第一次握手

若客户端 **A** 认为数据发送完成，则它需要向服务端 **B** 发送连接释放请求。

第二次握手

B 收到连接释放请求后，会告诉应用层要释放 **TCP** 链接。然后会发送 **ACK** 包，并进入 **CLOSE_WAIT** 状态，此时表明 **A** 到 **B** 的连接已经释放，不再接收 **A** 发的数据了。但是因为 **TCP** 连接是双向的，所以 **B** 仍旧可以发送数据给 **A**

3. ARQ 协议

ARQ 协议也就是超时重传机制。通过确认和超时机制保证了数据的正确送达，**ARQ** 协议包含停止等待 **ARQ** 和连续 **ARQ** 两种协议。

停止等待 ARQ

正常传输过程

只要 A 向 B 发送一段报文，都要停止发送并启动一个定时器，等待对端回应，在定时器时间内接收到对端应答就取消定时器并发送下一段报文。

报文丢失或出错

- 在报文传输的过程中可能会出现丢包。这时候超过定时器设定的时间就会再次发送丢失的数据直到对端响应，所以需要每次都备份发送的数据。
- 即使报文正常的传输到对端，也可能出现在传输过程中报文出错的问题。这时候对端会抛弃该报文并等待 A 端重传。
- PS：一般定时器设定的时间都会大于一个 **RTT** 的平均时间。

第三次握手

- B 如果此时还有没发完的数据会继续发送，完毕后会向 A 发送连接释放请求，然后 B 便进入 **LAST-ACK** 状态。
- PS：通过延迟确认的技术（通常有时间限制，否则对方会误认为需要重传），可以将第二次和第三次握手合并，延迟 **ACK** 包的发送。

第四次握手

A 收到释放请求后，向 B 发送确认应答，此时 A 进入 **TIME-WAIT** 状态。该状态会持续 **2MSL**（最大段生存期，指报文段在网络中生存的时间，超时会被抛弃）时间，若该时间段内没有 B 的重发请求的话，就进入 **CLOSED** 状态。当 B 收到确认应答后，也便进入 **CLOSED** 状态。

- 为什么 A 要进入 **TIME-WAIT** 状态，等待 **2MSL** 时间后才进入 **CLOSED** 状态？
- 为了保证 B 能收到 A 的确认应答。若 A 发完确认应答后直接进入 **CLOSED** 状态，如果确认应答因为网络问题一直没有到达，那么会造成 B 不能正常关闭。

ACK 超时或丢失

- 对端传输的应答也可能出现丢失或超时的情况。那么超过定时器时间 A 端照样会重传报文。这时候 B 端收到相同序号的报文会丢弃该报文并重传应答，直到 A 端发送下一个序号的报文。
- 在超时的情况下也可能出现应答很迟到达，这时 A 端会判断该序号是否已经接收过，如果接收过只需要丢弃应答即可。
- 从上面的描述中大家肯定可以发现这肯定不是一个高效的方式。假设在良好的网络环境中，每次发送数据都需要等待片刻肯定是不能接受的。那么既然我们不能接受这个不那么高效的协议，就来继续学习相对高效的协议吧。

连续 ARQ

在连续 ARQ 中，发送端拥有一个发送窗口，可以在没有收到应答的情况下持续发送窗口内的数据，这样相比停止等待 ARQ 协议来说减少了等待时间，提高了效率。

累计确认

连续 ARQ 中，接收端会持续不断收到报文。如果和停止等待 ARQ 中接收一个报文就发送一个应答一样，就太浪费资源了。通过累计确认，可以在收到多个报文以后统一回复一个应答报文。报文中的 ACK 标志位可以用来告诉发送端这个序号之前的数据已经全部接收到了，下次请发送这个序号后的数据。

但是累计确认也有一个弊端。在连续接收报文时，可能会遇到接收到序号 5 的报文后，并未接收到序号 6 的报文，然而序号 7 以后的报文已经接收。遇到这种情况时，ACK 只能回复 6，这样就会造成发送端重复发送数据的情况

4. 滑动窗口

- 上面小节中讲到了发送窗口。在 TCP 中，两端其实都维护着窗口：分别为发送端窗口和接收端窗口。
- 发送端窗口包含已发送但未收到应答的数据和可以发送但是未发送的数据。
- 发送端窗口是由接收窗口剩余大小决定的。接收方会把当前接收窗口的剩余大小写入应答报文，发送端收到应答后根据该值和当前网络拥塞情况设置发送窗口的大小，所以发送窗口的大小是不断变化的。
- 当发送端接收到应答报文后，会随之将窗口进行滑动

滑动窗口是一个很重要的概念，它帮助 TCP 实现了流量控制的功能。接收方通过报文告知发送方还可以发送多少数据，从而保证接收方能够来得及接收数据，防止出现接收方带宽已满，但是发送方还一直发送数据的情况

Zero 窗口

在发送报文的过程中，可能会遇到对端出现零窗口的情况。在该情况下，发送端会停止发送数据，并启动 `persistent timer`。该定时器会定时发送请求给对端，让对端告知窗口大小。在重试次数超过一定次数后，可能会中断

`TCP` 链接

5. 拥塞处理

- 拥塞处理和流量控制不同，后者是作用于接收方，保证接收方来得及接受数据。而前者是作用于网络，防止过多的数据拥塞网络，避免出现网络负载过大的情况。
- 拥塞处理包括了四个算法，分别为：慢开始，拥塞避免，快速重传，快速恢复

慢开始算法

慢开始算法，顾名思义，就是在传输开始时将发送窗口慢慢指数级扩大，从而避免一开始就传输大量数据导致网络拥塞。想必大家都下载过资源，每当我们开始下载的时候都会发现下载速度是慢慢提升的，而不是一蹴而就直接拉满带宽

慢开始算法步骤具体如下

- 连接初始设置拥塞窗口（Congestion Window）为 `1 MSS`（一个分段的最大数据量）
- 每过一个 `RTT` 就将窗口大小乘二
- 指数级增长肯定不能没有限制的，所以有一个阈值限制，当窗口大小大于阈值时就会启动拥塞避免算法。

拥塞避免算法

- 拥塞避免算法相比简单点，每过一个 `RTT` 窗口大小只加一，这样能够避免指数级增长导致网络拥塞，慢慢将大小调整到最佳值。
- 在传输过程中可能定时器超时的情况，这时候 `TCP` 会认为网络拥塞了，会马上进行以下步骤：

1. 将阈值设为当前拥塞窗口的一半
2. 将拥塞窗口设为 1 MSS
3. 启动拥塞避免算法

快速重传

快速重传一般和快恢复一起出现。一旦接收端收到的报文出现失序的情况，接收端只会回复最后一个顺序正确的报文序号。如果发送端收到三个重复的 ACK，无需等待定时器超时而是直接启动快速重传算法。具体算法分为两种：

TCP Tahoe 实现如下

- 将阈值设为当前拥塞窗口的一半
- 将拥塞窗口设为 1 MSS
- 重新开始慢开始算法
- TCP Reno 实现如下

拥塞窗口减半

- 将阈值设为当前拥塞窗口
- 进入快恢复阶段（重发对端需要的包，一旦收到一个新的 ACK 答复就退出该阶段），这种方式在丢失多个包的情况下就不那么好了
- 使用拥塞避免算法

TCP New Ren 改进后的快恢复

- TCP New Reno 算法改进了之前 TCP Reno 算法的缺陷。在之前，快恢复中只要收到一个新的 ACK 包，就会退出快恢复。
- 在 TCP New Reno 中，TCP 发送方先记下三个重复 ACK 的分段的最大序号。

假如我有一个分段数据是 1 ~ 10 这十个序号的报文，其中丢失了序号为 3 和 7 的报文，那么该分段的最大序号就是 10。发送端只会收到 ACK 序号为 3 的应答。这时候重发序号为 3 的报文，接收方顺利接收的话就会发送 ACK 序号为 7 的应答。这时候 TCP 知道对端是有多个包未收到，会继续发送序号为 7 的报文，接收方顺利接收并会发送 ACK 序号为 11 的应答，这时发送端认为这个分段接收端已经顺利接收，接下来会退出快恢复阶段。

32 HTTP/TLS

32.1 HTTP 请求中的内容

HTTP 请求由三部分构成，分别为：

- 请求行
- 首部
- 实体

- 请求行大概长这样 `GET /images/logo.gif HTTP/1.1`，基本由请求方法、URL、协议版本组成，这其中值得一说的就是请求方法了。
- 请求方法分为很多种，最常用的也就是 `Get` 和 `Post` 了。虽然请求方法有很多，但是更多的是传达一个语义，而不是说 `Post` 能做的事情 `Get` 就不能做了。如果你愿意，都使用 `Get` 请求或者 `Post` 请求都是可以的

常考面试题：Post 和 Get 的区别？

- 首先先引入副作用和幂等的概念。
 - 副作用指对服务器上的资源做改变，搜索是无副作用的，注册是副作用的。
 - 幂等指发送 M 和 N 次请求（两者不相同且都大于 1），服务器上资源的状态一致，比如注册 10 个和 11 个帐号是不幂等的，对文章进行更改 10 次和 11 次是幂等的。因为前者是多了一个账号（资源），后者只是更新同一个资源。
 - 在规范的应用场景上说，Get 多用于无副作用，幂等的场景，例如搜索关键字。Post 多用于副作用，不幂等的场景，例如注册。
-
- `Get` 请求能缓存，`Post` 不能
 - `Post` 相对 `Get` 安全一点点，因为 `Get` 请求都包含在 URL 里（当然你想写到 `body` 里也是可以的），且会被浏览器保存历史纪录。`Post` 不会，但是在抓包的情况下都是一样的。
 - URL 有长度限制，会影响 `Get` 请求，但是这个长度限制是浏览器规定的，不是 RFC 规定的
 - `Post` 支持更多的编码类型且不对数据类型限制

1. 首部

首部分为请求首部和响应首部，并且部分首部两种通用，接下来我们就来学习一部分的常用首部。

1.1 通用首部

通用字段	作用
Cache-Control	控制缓存的行为
Connection	浏览器想要优先使用的连接类型，比如 keep-alive
Date	创建报文时间
Pragma	报文指令
Via	代理服务器相关信息
Transfer-Encoding	传输编码方式
Upgrade	要求客户端升级协议
Warning	在内容中可能存在错误

1.2 请求首部

请求首部	作用
Accept	能正确接收的媒体类型
Accept-Charset	能正确接收的字符集
Accept-Encoding	能正确接收的编码格式列表
Accept-Language	能正确接收的语言列表
Expect	期待服务端的指定行为
From	请求方邮箱地址
Host	服务器的域名
If-Match	两端资源标记比较
If-Modified-Since	本地资源未修改返回 304 （比较时间）
If-None-Match	本地资源未修改返回 304 （比较标记）

请求首部	作用
User-Agent	客户端信息
Max-Forwards	限制可被代理及网关转发的次数
Proxy-Authorization	向代理服务器发送验证信息
Range	请求某个内容的一部分
Referer	表示浏览器所访问的前一个页面
TE	传输编码方式

1.3 响应首部

响应首部	作用
Accept-Ranges	是否支持某些种类的范围
Age	资源在代理缓存中存在的时间
ETag	资源标识
Location	客户端重定向到某个 URL
Proxy-Authenticate	向代理服务器发送验证信息
Server	服务器名字
WWW-Authenticate	获取资源需要的验证信息

1.4 实体首部

实体首部	作用
Allow	资源的正确请求方式
Content-Encoding	内容的编码格式
Content-Language	内容使用的语言
Content-Length	request body 长度
Content-Location	返回数据的备用地址
Content-MD5	Base64 加密格式的内容 MD5 检验值

实体首部	作用
Content-Range	内容的位置范围
Content-Type	内容的媒体类型
Expires	内容的过期时间
Last_modified	内容的最后修改时间

2. 常见状态码

状态码表示了响应的一个状态，可以让我们清晰的了解到这一次请求是成功还是失败，如果失败的话，是什么原因导致的，当然状态码也是用于传达语义的。如果胡乱使用状态码，那么它存在的意义就没有了

2XX 成功

- 200 OK，表示从客户端发来的请求在服务器端被正确处理
- 204 No content，表示请求成功，但响应报文不含实体的主体部分
- 205 Reset Content，表示请求成功，但响应报文不含实体的主体部分，但是与 204 响应不同在于要求请求方重置内容
- 206 Partial Content，进行范围请求

3XX 重定向

- 301 moved permanently，永久性重定向，表示资源已被分配了新的 URL
- 302 found，临时性重定向，表示资源临时被分配了新的 URL
- 303 see other，表示资源存在着另一个 URL，应使用 GET 方法获取资源
- 304 not modified，表示服务器允许访问资源，但因发生请求未满足条件的情况
- 307 temporary redirect，临时重定向，和 302 含义类似，但是期望客户端保持请求方法不变向新的地址发出请求

4XX 客户端错误

- 400 bad request，请求报文存在语法错误
- 401 unauthorized，表示发送的请求需要有通过 HTTP 认证的认证信息
- 403 forbidden，表示对请求资源的访问被服务器拒绝
- 404 not found，表示在服务器上没有找到请求的资源

5XX 服务器错误

- **500 internal sever error** , 表示服务器端在执行请求时发生了错误
- **501 Not Implemented** , 表示服务器不支持当前请求所需要的某个功能
- **503 service unavailable** , 表明服务器暂时处于超负载或正在停机维护, 无法处理请求

32.2 TLS

- **HTTPS** 还是通过了 **HTTP** 来传输信息, 但是信息通过 **TLS** 协议进行了加密。
- **TLS** 协议位于传输层之上, 应用层之下。首次进行 **TLS** 协议传输需要两个 **RTT** , 接下来可以通过 **Session Resumption** 减少到一个 **RTT** 。
- 在 **TLS** 中使用了两种加密技术, 分别为: 对称加密和非对称加密。

对称加密

- 对称加密就是两边拥有相同的密钥, 两边都知道如何将密文加密解密。
- 这种加密方式固然很好, 但是问题就在于如何让双方知道密钥。因为传输数据都是走的网络, 如果将密钥通过网络的方式传递的话, 一旦密钥被截获就没有加密的意义的。

非对称加密

- 有公钥私钥之分, 公钥所有人都可以知道, 可以将数据用公钥加密, 但是将数据解密必须使用私钥解密, 私钥只有分发公钥的一方才知道。
- 这种加密方式就可以完美解决对称加密存在的问题。假设现在两端需要使用对称加密, 那么在这之前, 可以先使用非对称加密交换密钥。

简单流程如下: 首先服务端将公钥公布出去, 那么客户端也就知道公钥了。接下来客户端创建一个密钥, 然后通过公钥加密并发送给服务端, 服务端接收到密文以后通过私钥解密出正确的密钥, 这时候两端就都知道密钥是什么了。

TLS 握手过程如下图:

- 客户端发送一个随机值以及需要的协议和加密方式。
- 服务端收到客户端的随机值, 自己也产生一个随机值, 并根据客户端需求的协议和加密方式来使用对应的方式, 并且发送自己的证书 (如果需要验证客户端证书需要说明)
- 客户端收到服务端的证书并验证是否有效, 验证通过会再生成一个随机值, 通过服务端证书的公钥去加密这个随机值并发送给服务端, 如果服务端需要验证客户端证书的话会附带证书

- 服务端收到加密过的随机值并使用私钥解密获得第三个随机值，这时候两端都拥有了三个随机值，可以通过这三个随机值按照之前约定的加密方式生成密钥，接下来的通信就可以通过该密钥来加密解密
- 通过以上步骤可知，在 TLS 握手阶段，两端使用非对称加密的方式来通信，但是因为非对称加密损耗的性能比对称加密大，所以在正式传输数据时，两端使用对称加密的方式通信。

PS：以上说明的都是 **TLS 1.2** 协议的握手情况，在 1.3 协议中，首次建立连接只需要一个 RTT，后面恢复连接不需要 RTT 了

33 HTTP2.0

- **HTTP/2** 很好的解决了当下最常用的 **HTTP/1** 所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国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** 所存在的一些问题而被推出来的。

33.1 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，我们会引入雪碧图、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），当页面中需要请求很多资源的时候，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，同时也接更容易实现全速传输，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

在 **HTTP/2** 中，因为可以复用同一个 **TCP** 连接，你会发现发送请求是长这样的

33.2 二进制传输

HTTP/2 中所有加强性能的核心点在于此。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。

33.3 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）。
- 帧代表着最小的数据单位，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流。
- 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，对端可以通过帧中的标识知道属于哪个请求。通过这个技术，可以避免 **HTTP** 旧版本中的队头阻塞问题，极大的提高传输性能。

33.4 Header 压缩

- 在 **HTTP/1** 中，我们使用文本的形式传输 **header**，在 **header** 携带 cookie 的情况下，可能每次都需要重复传输几百到几千的字节。
- 在 **HTTP / 2** 中，使用了 **HPACK** 压缩格式对传输的 **header** 进行编码，减少了 **header** 的大小。并在两端维护了索引表，用于记录出现过的 **header**，后面在传输过程中就可以传输已经记录过的 **header** 的键名，对端收到数据后就可以通过键名找到对应的值。

33.5 服务端 Push

- 在 **HTTP/2** 中，服务端可以在客户端某个请求后，主动推送其他资源。
- 可以想象以下情况，某些资源客户端是一定会请求的，这时就可以采取服务端 **push** 的技术，提前给客户端推送必要的资源，这样就可以相对减少一点延迟时间。当然在浏览器兼容的情况下你也可以使用 **prefetch**

33.6 HTTP/3

- 虽然 **HTTP/2** 解决了很多之前旧版本的问题，但是它还是存在一个巨大的问题，虽然这个问题并不是它本身造成的，而是底层支撑的 **TCP** 协议的问题。
- 因为 **HTTP/2** 使用了多路复用，一般来说同一域名下只需要使用一个 **TCP** 连接。当这个连接中出现了丢包的情况，那就会导致 **HTTP/2** 的表现情况反倒不如 **HTTP/1** 了。
- 因为在出现丢包的情况下，整个 **TCP** 都要开始等待重传，也就导致了后面的所有数据都被阻塞了。但是对于 **HTTP/1** 来说，可以开启多个 **TCP** 连接，出现这种情况反到只会影响其中一个连接，剩余的 **TCP** 连接还可以正常传输数据。
- 那么可能就会有人考虑到去修改 **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，更新起来不大现实。
- 基于这个原因，Google 就更起炉灶搞了一个基于 **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**，从这个名字中我们也可以发现，**HTTP/3** 最大的改造就是使用了 **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 **TLS1.3** 加密、流量控制、有序交付、重传等等功能。这里我们就挑选几个重要的功能学习下这个协议的内容。

多路复用

虽然 **HTTP/2** 支持了多路复用，但是 **TCP** 协议终究是没有这个功能的。**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有序交付且不会影响其他的数据流，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，就能迅速重连上。

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传递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异或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丢包的情况时，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

34 设计模式

设计模式总的来说是一个抽象的概念，前人通过无数次的实践总结出的一套写代码的方式，通过这种方式写的代码可以让别人更加容易阅读、维护以及复用。

34.1 工厂模式

工厂模式分为好几种，这里就不一一讲解了，以下是一个简单工厂模式的例子

```
class Man {  
  constructor(name) {  
    this.name = name  
  }  
  alertName() {  
    alert(this.name)  
  }  
}  
  
class Factory {  
  static create(name) {  
    return new Man(name)  
  }  
}
```

js

```
Factory.create('yck').alertName()
```

- 当然工厂模式并不仅仅是用来 `new` 出实例。
- 可以想象一个场景。假设有一份很复杂的代码需要用户去调用，但是用户并不关心这些复杂的代码，只需要你提供给我一个接口去调用，用户只负责传递需要的参数，至于这些参数怎么使用，内部有什么逻辑是不关心的，只需要你最后返回我一个实例。这个构造过程就是工厂。
- 工厂起到的作用就是隐藏了创建实例的复杂度，只需要提供一个接口，简单清晰。
- 在 `Vue` 源码中，你也可以看到工厂模式的使用，比如创建异步组件

```
export function createComponent (js  
  Ctor: Class<Component> | Function | Object | void,  
  data: ?VNodeData,  
  context: Component,  
  children: ?Array<VNode>,  
  tag?: string  
): VNode | Array<VNode> | void {  
  
  // 逻辑处理...  
  
  const vnode = new VNode(  
    `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,  
    data, undefined, undefined, undefined, context,  
    { Ctor, propsData, listeners, tag, children },  
    asyncFactory  
  )  
  
  return vnode  
}
```

在上述代码中，我们可以看到我们只需要调用 `createComponent` 传入参数就能创建一个组件实例，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过程，只需要一句代码调用就能实现功能

34.2 单例模式

- 单例模式很常用，比如全局缓存、全局状态管理等等这些只需要一个对象，就可以使用单例模式。

- 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  
  }  
})()  
  
let s1 = Singleton.getInstance()  
let s2 = Singleton.getInstance()  
console.log(s1 === s2) // 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 === Vue) {  
    // 如果发现 Vue 有值，就不重新创建实例了  
    return  
  }  
  Vue = _Vue  
  applyMixin(Vue)  
}
```

34.3 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 '港版插头'
  }
}

class Target {
  constructor() {
    this.plug = new Plug()
  }
  getName() {
    return this.plug.getName() + ' 适配器转二脚插头'
  }
}

let target = new Target()
target.getName() // 港版插头 适配器转二脚插头
```

在 **Vue** 中，我们其实经常使用到适配器模式。比如父组件传递给子组件一个时间戳属性，组件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

34.4 装饰模式

- 装饰模式不需要改变已有的接口，作用是给对象添加功能。就像我们经常需要给手机戴个保护套防摔一样，不改变手机自身，给手机添加了保护套提供防摔功能。
- 以下是如何实现装饰模式的例子，使用了 ES7 中的装饰器语法

```
function readonly(target, key, descriptor) {
  descriptor.writable = false
  return descriptor
}

class Test {
  @readonly
  name = 'yck'
}

let t = new Test()
```

```
t.yck = '111' // 不可修改
```

在 **React** 中，装饰模式其实随处可见

```
import { connect } from 'react-redux'
class MyComponent extends React.Component {
  // ...
}
export default connect(mapStateToProps)(MyComponent)
```

js

34.5 代理模式

- 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

```
<ul id="ul">
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ul>
<script>
  let ul = document.querySelector('#ul')
  ul.addEventListener('click', (event) => {
    console.log(event.target);
  })
</script>
```

html

因为存在太多的 li，不可能每个都去绑定事件。这时候可以通过给父节点绑定一个事件，让父节点作为代理去拿到真实点击的节点。

34.6 发布-订阅模式

- 发布-订阅模式也叫做观察者模式。通过一对一或者一对多的依赖关系，当对象发生改变时，订阅方都会收到通知。在现实生活中，也有很多类似场景，比如我需要在购物网站上购买一个产品，但是发现该产品目前处于缺货状态，这时候我可以点击有货通知的按钮，让网站在产品有货的时候通过短信通知我。
- 在实际代码中其实发布-订阅模式也很常见，比如我们点击一个按钮触发了点击事件就是使用了该模式

html

```
<ul id="ul"></ul>
<script>
  let ul = document.querySelector('#ul')
  ul.addEventListener('click', (event) => {
    console.log(event.target);
  })
</script>
```

在 **Vue** 中，如何实现响应式也是使用了该模式。对于需要实现响应式的对象来说，在 **get** 的时候会进行依赖收集，当改变了对象的属性时，就会触发派发更新。

34.7 外观模式

- 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

js

```
function addEvent(elm, evType, fn, useCapture) {
  if (elm.addEventListener) {
    elm.addEventListener(evType, fn, useCapture)
    return true
  } else if (elm.attachEvent) {
    var r = elm.attachEvent("on" + evType, fn)
    return r
  } else {
    elm["on" + evType] = fn
  }
}
```

对于不同的浏览器，添加事件的方式可能会存在兼容问题。如果每次都需要去这样写一遍的话肯定是不能接受的，所以我们将这些判断逻辑统一封装在一个

接口中，外部需要添加事件只需要调用 `addEventListener` 即可。

35 常见数据结构

35.1 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来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，比如说四则运算。
- 对于一个算法来说，可能会计算出操作次数为 $aN + 1$ ， N 代表数据量。那么该算法的时间复杂度就是 $O(N)$ 。因为我们在计算时间复杂度的时候，数据量通常是非常大的，这时候低阶项和常数项可以忽略不计。
- 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和常数项了

35.2 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。
- 栈的特点是只能在某一端添加或删除数据，遵循先进后出的原则

实现

每种数据结构都可以用很多种方式来实现，其实可以把栈看成是数组的一个子集，所以这里使用数组来实现

```
class Stack {  
  constructor() {  
    this.stack = []  
  }  
  push(item) {  
    this.stack.push(item)  
  }  
  pop() {
```

js

```
    this.stack.pop()
  }
  peek() {
    return this.stack[this.getCount() - 1]
  }
  getCount() {
    return this.stack.length
  }
  isEmpty() {
    return this.getCount() === 0
  }
}
```

35.3 应用

选取了 LeetCode 上序号为 **20 的题** 题意是匹配括号，可以通过栈的特性来完成这道题目

```
var isValid = function (s) {
  let map = {
    '(': -1,
    ')': 1,
    '[': -2,
    ']': 2,
    '{': -3,
    '}': 3
  }
  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]] !== 0) return false
    }
  }
  if (stack.length > 0) return false
  return true
};
```

其实在 **Vue** 中关于模板解析的代码，就有应用到匹配尖括号的内容

35.4 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()  
  }  
  getHeader() {  
    return this.queue[0]  
  }  
  getLength() {  
    return this.queue.length  
  }  
  isEmpty() {  
    return this.getLength() === 0  
  }  
}
```

js

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

js

```
class SqQueue {
  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) {
    // 判断队尾 + 1 是否为队头
    // 如果是就代表需要扩容数组
    // % this.queue.length 是为了防止数组越界
    if (this.first === (this.last + 1) % this.queue.length) {
      this.resize(this.getLength() * 2 + 1)
    }
    this.queue[this.last] = item
    this.size++
    this.last = (this.last + 1) % this.queue.length
  }
  dequeue() {
    if (this.isEmpty()) {
      throw Error('Queue is empty')
    }
    let r = this.queue[this.first]
    this.queue[this.first] = null
    this.first = (this.first + 1) % this.queue.length
    this.size--
    // 判断当前队列大小是否过小
    // 为了保证不浪费空间，在队列空间等于总长度四分之一时
    // 且不为 2 时缩小总长度为当前的一半
    if (this.size === this.getLength() / 4 && this.getLength() / 2 !== 0) {
      this.resize(this.getLength() / 2)
    }
    return r
  }
  getHeader() {
    if (this.isEmpty()) {
      throw Error('Queue is empty')
    }
    return this.queue[this.first]
  }
}
```

```
getLength() {
  return this.queue.length - 1
}
isEmpty() {
  return this.first === this.last
}
resize(length) {
  let q = new Array(length)
  for (let i = 0; i < length; i++) {
    q[i] = this.queue[(i + this.first) % this.queue.length]
  }
  this.queue = q
  this.first = 0
  this.last = this.size
}
}
```

35.5 链表

概念

链表是一个线性结构，同时也是一个天然的递归结构。链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

概念

链表是一个线性结构，同时也是一个天然的递归结构。链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

实现

单向链表

```
class Node {
  constructor(v, next) {
    this.value = v
```

js


```
        this.next = next
    }
}

class LinkList {
    constructor() {
        // 链表长度
        this.size = 0
        // 虚拟头部
        this.dummyNode = new Node(null, null)
    }

    find(header, index, currentIndex) {
        if (index === currentIndex) return header
        return this.find(header.next, index, currentIndex + 1)
    }

    addNode(v, index) {
        this.checkIndex(index)
        // 当往链表末尾插入时, prev.next 为空
        // 其他情况时, 因为要插入节点, 所以插入的节点
        // 的 next 应该是 prev.next
        // 然后设置 prev.next 为插入的节点
        let prev = this.find(this.dummyNode, index, 0)
        prev.next = new Node(v, prev.next)
        this.size++
        return prev.next
    }

    insertNode(v, index) {
        return this.addNode(v, index)
    }

    addToFirst(v) {
        return this.addNode(v, 0)
    }

    addToLast(v) {
        return this.addNode(v, this.size)
    }

    removeNode(index, isLast) {
        this.checkIndex(index)
        index = isLast ? index - 1 : index
        let prev = this.find(this.dummyNode, index, 0)
        let node = prev.next
        prev.next = node.next
        node.next = null
        this.size--
        return node
    }

    removeFirstNode() {
        return this.removeNode(0)
    }
}
```

```

removeLastNode() {
  return this.removeNode(this.size, true)
}
checkIndex(index) {
  if (index < 0 || index > this.size) throw Error('Index error')
}
getNode(index) {
  this.checkIndex(index)
  if (this.isEmpty()) return
  return this.find(this.dummyNode, index, 0).next
}
isEmpty() {
  return this.size === 0
}
getSize() {
  return this.size
}
}

```

35.6 树

二叉树

- 树拥有很多种结构，二叉树是树中最常用的结构，同时也是一个天然的递归结构。
- 二叉树拥有一个根节点，每个节点至多拥有两个子节点，分别为：左节点和右节点。树的最底部节点称之为叶节点，当一颗树的叶数量数量为满时，该树可以称之为满二叉树。

二分搜索树

- 二分搜索树也是二叉树，拥有二叉树的特性。但是区别在于二分搜索树每个节点的值都比他的左子树的值大，比右子树的值小。
- 这种存储方式很适合于数据搜索。如下图所示，当需要查找 6 的时候，因为需要查找的值比根节点的值大，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。

实现

```

class Node {
  constructor(value) {
    this.value = value
    this.left = null

```

js

```

        this.right = null
    }
}
class BST {
    constructor() {
        this.root = null
        this.size = 0
    }
    getSize() {
        return this.size
    }
    isEmpty() {
        return this.size === 0
    }
    addNode(v) {
        this.root = this._addChild(this.root, v)
    }
    // 添加节点时，需要比较添加的节点值和当前
    // 节点值的大小
    _addChild(node, v) {
        if (!node) {
            this.size++
            return new Node(v)
        }
        if (node.value > v) {
            node.left = this._addChild(node.left, v)
        } else if (node.value < v) {
            node.right = this._addChild(node.right, v)
        }
        return node
    }
}

```

- 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，遍历左子树和遍历右子树。如果只需要实现先序遍历，那么只需要第一次遍历到节点时进行操作即可。

```

// 先序遍历可用于打印树的结构
// 先序遍历先访问根节点，然后访问左节点，最后访问右节点。
preTraversal() {
    this._pre(this.root)
}
_pre(node) {

```

js

```

    if (node) {
        console.log(node.value)
        this._pre(node.left)
        this._pre(node.right)
    }
}
// 中序遍历可用于排序
// 对于 BST 来说，中序遍历可以实现一次遍历就
// 得到有序的值
// 中序遍历表示先访问左节点，然后访问根节点，最后访问右节点。
midTraversal() {
    this._mid(this.root)
}
_mid(node) {
    if (node) {
        this._mid(node.left)
        console.log(node.value)
        this._mid(node.right)
    }
}
// 后序遍历可用于先操作子节点
// 再操作父节点的场景
// 后序遍历表示先访问左节点，然后访问右节点，最后访问根节点。
backTraversal() {
    this._back(this.root)
}
_back(node) {
    if (node) {
        this._back(node.left)
        this._back(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()
    // 将根节点入队
    q.enqueue(this.root)
    // 循环判断队列是否为空，为空

```

js

```
// 代表树遍历完毕
while (!q.isEmpty()) {
  // 将队首出队，判断是否有左右子树
  // 有的话，就先左后右入队
  let n = q.dequeue()
  console.log(n.value)
  if (n.left) q.enqueue(n.left)
  if (n.right) q.enqueue(n.right)
}
}
```

接下来先介绍如何在树中寻找最小值或最大数。因为二分搜索树的特性，所以最小值一定在根节点的最左边，最大值相反

```
getMin() {
  return this._getMin(this.root).value
}
_getMin(node) {
  if (!node.left) return node
  return this._getMin(node.left)
}
getMax() {
  return this._getMax(this.root).value
}
_getMax(node) {
  if (!node.right) return node
  return this._getMin(node.right)
}
```

js

向上取整和向下取整，这两个操作是相反的，所以代码也是类似的，这里只介绍如何向下取整。既然是向下取整，那么根据二分搜索树的特性，值一定在根节点的左侧。只需要一直遍历左子树直到当前节点的值不再大于等于需要的值，然后判断节点是否还拥有右子树。如果有的话，继续上面的递归判断。

```
floor(v) {
  let node = this._floor(this.root, v)
  return node ? node.value : null
}
_floor(node, v) {
  if (!node) return null
```

js

```

    if (node.value === v) return v
    // 如果当前节点值还比需要的值大，就继续递归
    if (node.value > v) {
        return this._floor(node.left, v)
    }
    // 判断当前节点是否拥有右子树
    let right = this._floor(node.right, v)
    if (right) return right
    return node
}

```

排名，这是用于获取给定值的排名或者排名第几的节点的值，这两个操作也是相反的，所以这个只介绍如何获取排名第几的节点的值。对于这个操作而言，我们需要略微的改造点代码，让每个节点拥有一个 size 属性。该属性表示该节点下有多少子节点（包含自身）

```

class Node {
    constructor(value) {
        this.value = value
        this.left = null
        this.right = null
        // 修改代码
        this.size = 1
    }
}
// 新增代码
_getSize(node) {
    return node ? node.size : 0
}
_addChild(node, v) {
    if (!node) {
        return new Node(v)
    }
    if (node.value > v) {
        // 修改代码
        node.size++
        node.left = this._addChild(node.left, v)
    } else if (node.value < v) {
        // 修改代码
        node.size++
        node.right = this._addChild(node.right, v)
    }
    return node
}

```

js

```

}
select(k) {
  let node = this._select(this.root, k)
  return node ? node.value : null
}
_select(node, k) {
  if (!node) return null
  // 先获取左子树下有几个节点
  let size = node.left ? node.left.size : 0
  // 判断 size 是否大于 k
  // 如果大于 k, 代表所需要的节点在左节点
  if (size > k) return this._select(node.left, k)
  // 如果小于 k, 代表所需要的节点在右节点
  // 注意这里需要重新计算 k, 减去根节点除了右子树的节点数量
  if (size < k) return this._select(node.right, k - size - 1)
  return node
}

```

接下来讲解的是二分搜索树中最难实现的部分：删除节点。因为对于删除节点来说，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树

对于前两种情况很好解决，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反的，所以这里也就不再赘述。

```

delectMin() {
  this.root = this._delectMin(this.root)
  console.log(this.root)
}
_delectMin(node) {
  // 一直递归左子树
  // 如果左子树为空，就判断节点是否拥有右子树
  // 有右子树的话就把需要删除的节点替换为右子树
  if ((node !== null) & !node.left) return node.right
  node.left = this._delectMin(node.left)
  // 最后需要重新维护下节点的 `size`
  node.size = this._getSize(node.left) + this._getSize(node.right) + 1
}

```

js

```

    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，也就是如何解决第三种情况。
- 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。然后将需要删除节点的左子树赋值给后继节点，右子树删除后继节点后赋值给他。
- 你如果对于这个解决办法有疑问的话，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。那么当需要删除父节点时，势必需要拿出一个比父节点大的节点来替换父节点。这个节点肯定不存在于左子树，必然存在于右子树。然后又需要保持父节点都是比右子节点小的，那么就可以取出右子树中最小的那个节点来替换父节点。

js

```

delete(v) {
  this.root = this._delete(this.root, v)
}
_delete(node, v) {
  if (!node) return null
  // 寻找的节点比当前节点小，去左子树找
  if (node.value < v) {
    node.right = this._delete(node.right, v)
  } else if (node.value > v) {
    // 寻找的节点比当前节点大，去右子树找
    node.left = this._delete(node.left, v)
  } else {
    // 进入这个条件说明已经找到节点
    // 先判断节点是否拥有左右子树中的一个
    // 是的话，将子树返回出去，这里和 `_deleteMin` 的操作一样
    if (!node.left) return node.right
    if (!node.right) return node.left
    // 进入这里，代表节点拥有左右子树
    // 先取出当前节点的后继节点，也就是取当前节点右子树的最小值
    let min = this._getMin(node.right)
    // 取出最小值后，删除最小值
    // 然后把删除节点后的子树赋值给最小值节点
    min.right = this._deleteMin(node.right)
    // 左子树不动
    min.left = node.left
    node = min
  }
  // 维护 size
  node.size = this._getSize(node.left) + this._getSize(node.right) + 1
}

```



```
    return node
}
```

35.7 AVL 树

概念

二分搜索树实际在业务中是受到限制的，因为并不是严格的 $O(\log N)$ ，在极端情况下会退化成长链表，比如加入一组升序的数字就会造成这种情况。

AVL 树改进了二分搜索树，在 AVL 树中任意节点的左右子树的高度差都不大于 1，这样保证了时间复杂度是严格的 $O(\log N)$ 。基于此，对 AVL 树增加或删除节点时可能需要旋转树来达到高度的平衡。

实现

- 因为 **AVL** 树是改进了二分搜索树，所以部分代码是于二分搜索树重复的，对于重复内容不作再次解析。
- 对于 AVL 树来说，添加节点会有四种情况
- 对于左左情况来说，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，节点比左节点大，比右节点小，所以旋转以后也要实现这个特性。
- 旋转之前： $new < 2 < C < 3 < B < 5 < A$ ，右旋之后节点 **3** 为根节点，这时候需要将节点 3 的右节点加到节点 5 的左边，最后还需要更新节点的高度。
- 对于右右情况来说，相反于左左情况，所以不再赘述。
- 对于左右情况来说，新增加的节点位于节点 4 的右侧。对于这种情况，需要通过两次旋转来达到目的。
- 首先对节点的左节点左旋，这时树满足左左的情况，再对节点进行一次右旋就可以达到目的。

```
class Node {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
    this.height = 1
  }
}
```

js

```
}
```

```
class AVL {
  constructor() {
    this.root = null
  }
  addNode(v) {
    this.root = this._addChild(this.root, v)
  }
  _addChild(node, v) {
    if (!node) {
      return new Node(v)
    }
    if (node.value > v) {
      node.left = this._addChild(node.left, v)
    } else if (node.value < v) {
      node.right = this._addChild(node.right, v)
    } else {
      node.value = v
    }
    node.height =
      1 + Math.max(this._getHeight(node.left), this._getHeight(node.right))
    let factor = this._getBalanceFactor(node)
    // 当需要右旋时，根节点的左树一定比右树高度高
    if (factor > 1 && this._getBalanceFactor(node.left) >= 0) {
      return this._rightRotate(node)
    }
    // 当需要左旋时，根节点的左树一定比右树高度矮
    if (factor < -1 && this._getBalanceFactor(node.right) <= 0) {
      return this._leftRotate(node)
    }
    // 左右情况
    // 节点的左树比右树高，且节点的左树的右树比节点的左树的左树高
    if (factor > 1 && this._getBalanceFactor(node.left) < 0) {
      node.left = this._leftRotate(node.left)
      return this._rightRotate(node)
    }
    // 右左情况
    // 节点的左树比右树矮，且节点的右树的右树比节点的右树的左树矮
    if (factor < -1 && this._getBalanceFactor(node.right) > 0) {
      node.right = this._rightRotate(node.right)
      return this._leftRotate(node)
    }

    return node
  }
  _getHeight(node) {
```

```

    if (!node) return 0
    return node.height
}
_getBalanceFactor(node) {
    return this._getHeight(node.left) - this._getHeight(node.right)
}
// 节点右旋
//          5          2
//        /  \        /  \
//       2    6  ==>  1    5
//      /  \        /  \
//     1    3      new  3  6
//    /
//   new
_rightRotate(node) {
    // 旋转后新根节点
    let newRoot = node.left
    // 需要移动的节点
    let moveNode = newRoot.right
    // 节点 2 的右节点改为节点 5
    newRoot.right = node
    // 节点 5 左节点改为节点 3
    node.left = moveNode
    // 更新树的高度
    node.height =
        1 + Math.max(this._getHeight(node.left), this._getHeight(node.right))
    newRoot.height =
        1 +
        Math.max(this._getHeight(newRoot.left), this._getHeight(newRoot.right))

    return newRoot
}
// 节点左旋
//          4          6
//        /  \        /  \
//       2    6  ==>  4    7
//      /  \        /  \  \
//     5    7      2    5   new
//                  \
//                 new
_leftRotate(node) {
    // 旋转后新根节点
    let newRoot = node.right
    // 需要移动的节点
    let moveNode = newRoot.left
    // 节点 6 的左节点改为节点 4
    newRoot.left = node

```

```

// 节点 4 右节点改为节点 5
node.right = moveNode
// 更新树的高度
node.height =
  1 + Math.max(this._getHeight(node.left), this._getHeight(node.right))
newRoot.height =
  1 +
  Math.max(this._getHeight(newRoot.left), this._getHeight(newRoot.right))

return newRoot
}
}

```

35.8 Trie

概念

- 在计算机科学，trie，又称前缀树或字典树，是一种有序树，用于保存关联数组，其中的键通常是字符串。

简单点来说，这个结构的作用大多是为了方便搜索字符串，该树有以下几个特点

- 根节点代表空字符串，每个节点都有 N（假如搜索英文字符，就有 26 条）条链接，每条链接代表一个字符
- 节点不存储字符，只有路径才存储，这点和其他的树结构不同
- 从根节点开始到任意一个节点，将沿途经过的字符连接起来就是该节点对应的字符串

实现

总得来说 **Trie** 的实现相比别的树结构来说简单的很多，实现就以搜索英文字符为例。

```

class TrieNode {
  constructor() {
    // 代表每个字符经过节点的次数
    this.path = 0
    // 代表到该节点的字符串有几个

```

js

```
        this.end = 0
        // 链接
        this.next = new Array(26).fill(null)
    }
}
class Trie {
    constructor() {
        // 根节点，代表空字符
        this.root = new TrieNode()
    }
    // 插入字符串
    insert(str) {
        if (!str) return
        let node = this.root
        for (let i = 0; i < str.length; i++) {
            // 获得字符先对应的索引
            let index = str[i].charCodeAt() - 'a'.charCodeAt()
            // 如果索引对应没有值，就创建
            if (!node.next[index]) {
                node.next[index] = new TrieNode()
            }
            node.path += 1
            node = node.next[index]
        }
        node.end += 1
    }
    // 搜索字符串出现的次数
    search(str) {
        if (!str) return
        let node = this.root
        for (let i = 0; i < str.length; i++) {
            let index = str[i].charCodeAt() - 'a'.charCodeAt()
            // 如果索引对应没有值，代表没有需要搜索的字符串
            if (!node.next[index]) {
                return 0
            }
            node = node.next[index]
        }
        return node.end
    }
    // 删除字符串
    delete(str) {
        if (!this.search(str)) return
        let node = this.root
        for (let i = 0; i < str.length; i++) {
            let index = str[i].charCodeAt() - 'a'.charCodeAt()
            // 如果索引对应的节点的 Path 为 0，代表经过该节点的字符串
```

```

// 已经一个，直接删除即可
if (--node.next[index].path == 0) {
  node.next[index] = null
  return
}
node = node.next[index]
}
node.end -= 1
}
}

```

35.9 并查集

概念

- 并查集是一种特殊的树结构，用于处理一些不交集的合并及查询问题。该结构中每个节点都有一个父节点，如果只有当前一个节点，那么该节点的父节点指向自己。

这个结构中有两个重要的操作，分别是：

- **Find**：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
- **Union**：将两个子集合并成同一个集合。

实现

```

class DisjointSet {
  // 初始化样本
  constructor(count) {
    // 初始化时，每个节点的父节点都是自己
    this.parent = new Array(count)
    // 用于记录树的深度，优化搜索复杂度
    this.rank = new Array(count)
    for (let i = 0; i < count; i++) {
      this.parent[i] = i
      this.rank[i] = 1
    }
  }
  find(p) {
    // 寻找当前节点的父节点是否为自己，不是的话表示还没找到
    // 开始进行路径压缩优化
    // 假设当前节点父节点为 A

```

js

```

// 将当前节点挂载到 A 节点的父节点上，达到压缩深度的目的
while (p !== this.parent[p]) {
  this.parent[p] = this.parent[this.parent[p]]
  p = this.parent[p]
}
return p
}
isConnected(p, q) {
  return this.find(p) === this.find(q)
}
// 合并
union(p, q) {
  // 找到两个数字的父节点
  let i = this.find(p)
  let j = this.find(q)
  if (i === j) return
  // 判断两棵树的深度，深度小的加到深度大的树下面
  // 如果两棵树深度相等，那就无所谓怎么加
  if (this.rank[i] < this.rank[j]) {
    this.parent[i] = j
  } else if (this.rank[i] > this.rank[j]) {
    this.parent[j] = i
  } else {
    this.parent[i] = j
    this.rank[j] += 1
  }
}
}
}

```

35.10 堆

概念

- 堆通常是一个可以被看做一棵树的数组对象。

堆的实现通过构造二叉堆，实为二叉树的一种。这种数据结构具有以下性质。

- 任意节点小于（或大于）它的所有子节点
 - 堆总是一棵完全树。即除了最底层，其他层的节点都被元素填满，且最底层从左到右填入。
- 将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。
 - 优先队列也完全可以用堆来实现，操作是一模一样的。

实现大根堆

- 堆的每个节点的左边子节点索引是 $i * 2 + 1$ ，右边是 $i * 2 + 2$ ，父节点是 $(i - 1) / 2$ 。
- 堆有两个核心的操作，分别是 `shiftUp` 和 `shiftDown`。前者用于添加元素，后者用于删除根节点。
- `shiftUp` 的核心思路是一路将节点与父节点对比大小，如果比父节点大，就和父节点交换位置。
- `shiftDown` 的核心思路是先将根节点和末尾交换位置，然后移除末尾元素。接下来循环判断父节点和两个子节点的大小，如果子节点大，就把最大的子节点和父节点交换。

```
class MaxHeap {
  constructor() {
    this.heap = []
  }
  size() {
    return this.heap.length
  }
  empty() {
    return this.size() == 0
  }
  add(item) {
    this.heap.push(item)
    this._shiftUp(this.size() - 1)
  }
  removeMax() {
    this._shiftDown(0)
  }
  getParentIndex(k) {
    return parseInt((k - 1) / 2)
  }
  getLeftIndex(k) {
    return k * 2 + 1
  }
  _shiftUp(k) {
    // 如果当前节点比父节点大，就交换
    while (this.heap[k] > this.heap[this.getParentIndex(k)]) {
      this._swap(k, this.getParentIndex(k))
      // 将索引变成父节点
      k = this.getParentIndex(k)
    }
  }
  _shiftDown(k) {
```



```
// 交换首位并删除末尾
this._swap(k, this.size() - 1)
this.heap.splice(this.size() - 1, 1)
// 判断节点是否有左孩子，因为二叉堆的特性，有右必有左
while (this.getLeftIndex(k) < this.size()) {
  let j = this.getLeftIndex(k)
  // 判断是否有右孩子，并且右孩子是否大于左孩子
  if (j + 1 < this.size() && this.heap[j + 1] > this.heap[j]) j++
  // 判断父节点是否已经比子节点都大
  if (this.heap[k] >= this.heap[j]) break
  this._swap(k, j)
  k = j
}
}
_swap(left, right) {
  let rightValue = this.heap[right]
  this.heap[right] = this.heap[left]
  this.heap[left] = rightValue
}
}
```

36 常考算法题解析

对于大部分公司的面试来说，排序的内容已经足以应付了，由此为了更好的符合大众需求，排序的内容是最多的。当然如果你还想冲击更好的公司，那么整个章节的内容都是需要掌握的。对于字节跳动这类十分看重算法的公司来说，这一章节是远远不够的，剑指Offer应该是你更好的选择

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，请各位打开代码编辑器，一行行的敲代码，单纯阅读是学习不了算法的

另外学习算法的时候，有一个可视化界面会相对减少点学习的难度，具体可以阅读 [algorithm-visualizer](#) 这个仓库

36.1 位运算

- 在进入正题之前，我们先来学习一下位运算的内容。因为位运算在算法中很有用，速度可以比四则运算快很多。

- 在学习位运算之前应该知道十进制如何转二进制，二进制如何转十进制。这里说明下简单的计算方式

- 十进制 33 可以看成是 $32 + 1$ ，并且 33 应该是六位二进制的（因为 33 近似 32，而 32 是 2 的五次方，所以是六位），那么十进制 33 就是 100001，只要是 2 的次方，那么就是 1 否则都为 0
- 那么二进制 100001 同理，首位是 2^5 ，末位是 2^0 ，相加得出 33

1. 左移 <<

```
10 << 1 // -> 20
```

左移就是将二进制全部往左移动，10 在二进制中表示为 1010，左移一位后变成 10100，转换为十进制也就是 20，所以基本可以把左移看成以下公式 $a * (2 ^ b)$

2. 算数右移 >>

```
10 >> 1 // -> 5
```

算数右移就是将二进制全部往右移动并去除多余的右边，10 在二进制中表示为 1010，右移一位后变成 101，转换为十进制也就是 5，所以基本可以把右移看成以下公式 $\text{int } v = a / (2 ^ b)$

右移很好用，比如可以用在二分算法中取中间值

```
13 >> 1 // -> 6
```

3. 按位操作

3.1 按位与

每一位都为 1，结果才为 1

```
8 & 7 // -> 0
// 1000 & 0111 -> 0000 -> 0
```

3.2 按位或

其中一位为 1，结果就是 1

```
8 | 7 // -> 15
// 1000 | 0111 -> 1111 -> 15
```

3.3 按位异或

每一位都不同，结果才为 1

```
8 ^ 7 // -> 15
8 ^ 8 // -> 0
// 1000 ^ 0111 -> 1111 -> 15
// 1000 ^ 1000 -> 0000 -> 0
```

js

- 从以上代码中可以发现按位异或就是不进位加法
- 面试题：两个数不使用四则运算得出和

这道题中可以按位异或，因为按位异或就是不进位加法， $8 \wedge 8 = 0$ 如果进位了，就是 16 了，所以我们只需要将两个数进行异或操作，然后进位。那么也就是说两个二进制都是 1 的位置，左边应该有一个进位 1，所以可以得出以下公式 $a + b = (a \wedge b) + ((a \& b) \ll 1)$ ，然后通过迭代的方式模拟加法

```
function sum(a, b) {
  if (a == 0) return b
  if (b == 0) return a
  let newA = a ^ b
  let newB = (a & b) << 1
  return sum(newA, newB)
}
```

js

36.2 排序

以下两个函数是排序中会用到的通用函数，就不一一写了

```
function checkArray(array) {  
  if (!array) return  
}  
function swap(array, left, right) {  
  let rightValue = array[right]  
  array[right] = array[left]  
  array[left] = rightValue  
}
```

js

36.2.1 冒泡排序

冒泡排序的原理如下，从第一个元素开始，把当前元素和下一个索引元素进行比较。如果当前元素大，那么就交换位置，重复操作直到比较到最后一个元素，那么此时最后一个元素就是该数组中最大的数。下一轮重复以上操作，但是此时最后一个元素已经是最大数了，所以不需要再比较最后一个元素，只需要比较到 `length - 1` 的位置。

以下是实现该算法的代码

```
function bubble(array) {  
  checkArray(array);  
  for (let i = array.length - 1; i > 0; i--) {  
    // 从 0 到 `length - 1` 遍历  
    for (let j = 0; j < i; j++) {  
      if (array[j] > array[j + 1]) swap(array, j, j + 1)  
    }  
  }  
  return array;  
}
```

js

该算法的操作次数是一个等差数列 $n + (n - 1) + (n - 2) + 1$ ，去掉常数项以后得出时间复杂度是 $O(n * n)$

36.2.2 插入排序

插入排序的原理如下。第一个元素默认是已排序元素，取出下一个元素和当前元素比较，如果当前元素大就交换位置。那么此时第一个元素就是当前的最小数，所以下次取出操作从第三个元素开始，向前对比，重复之前的操作

以下是实现该算法的代码

```
function insertion(array) {  
  checkArray(array);  
  for (let i = 1; i < array.length; i++) {  
    for (let j = i - 1; j >= 0 && array[j] > array[j + 1]; j--)  
      swap(array, j, j + 1);  
  }  
  return array;  
}
```

js

该算法的操作次数是一个等差数列 $n + (n - 1) + (n - 2) + 1$ ，去掉常数项以后得出时间复杂度是 $O(n * n)$

36.2.3 选择排序

选择排序的原理如下。遍历数组，设置最小值的索引为 0，如果取出的值比当前最小值小，就替换最小值索引，遍历完成后，将第一个元素和最小值索引上的值交换。如上操作后，第一个元素就是数组中的最小值，下次遍历就可以从索引 1 开始重复上述操作

以下是实现该算法的代码

```
function selection(array) {
  checkArray(array);
  for (let i = 0; i < array.length - 1; i++) {
    let minIndex = i;
    for (let j = i + 1; j < array.length; j++) {
      minIndex = array[j] < array[minIndex] ? j : minIndex;
    }
    swap(array, i, minIndex);
  }
  return array;
}
```

该算法的操作次数是一个等差数列 $n + (n - 1) + (n - 2) + 1$ ，去掉常数项以后得出时间复杂度是 $O(n * n)$

36.2.4 归并排序

归并排序的原理如下。递归的将数组两两分开直到最多包含两个元素，然后将数组排序合并，最终合并为排序好的数组。假设我有一组数组 $[3, 1, 2, 8, 9, 7, 6]$ ，中间数索引是 3，先排序数组 $[3, 1, 2, 8]$ 。在这个左边数组上，继续拆分直到变成数组包含两个元素（如果数组长度是奇数的话，会有一个拆分数组只包含一个元素）。然后排序数组 $[3, 1]$ 和 $[2, 8]$ ，然后再排序数组 $[1, 3, 2, 8]$ ，这样左边数组就排序完成，然后按照以上思路排序右边数组，最后将数组 $[1, 2, 3, 8]$ 和 $[6, 7, 9]$ 排序

以下是实现该算法的代码

```
function sort(array) {
  checkArray(array);
  mergeSort(array, 0, array.length - 1);
  return array;
}

function mergeSort(array, left, right) {
  // 左右索引相同说明已经只有一个数
  if (left === right) return;
  // 等同于 `left + (right - left) / 2`
```

```

// 相比 `(left + right) / 2` 来说更加安全，不会溢出
// 使用位运算是因为位运算比四则运算快
let mid = parseInt(left + ((right - left) >> 1));
mergeSort(array, left, mid);
mergeSort(array, mid + 1, right);

let help = [];
let i = 0;
let p1 = left;
let p2 = mid + 1;
while (p1 <= mid && p2 <= right) {
  help[i++] = array[p1] < array[p2] ? array[p1++] : array[p2++];
}
while (p1 <= mid) {
  help[i++] = array[p1++];
}
while (p2 <= right) {
  help[i++] = array[p2++];
}
for (let i = 0; i < help.length; i++) {
  array[left + i] = help[i];
}
return array;
}

```

以上算法使用了递归的思想。递归的本质就是压栈，每递归执行一次函数，就将该函数的信息（比如参数，内部的变量，执行到的行数）压栈，直到遇到终止条件，然后出栈并继续执行函数。对于以上递归函数的调用轨迹如下

```

mergeSort(data, 0, 6) // mid = 3
  mergeSort(data, 0, 3) // mid = 1
    mergeSort(data, 0, 1) // mid = 0
      mergeSort(data, 0, 0) // 遇到终止，回退到上一步
      mergeSort(data, 1, 1) // 遇到终止，回退到上一步
      // 排序 p1 = 0, p2 = mid + 1 = 1
      // 回退到 `mergeSort(data, 0, 3)` 执行下一个递归
    mergeSort(2, 3) // mid = 2
      mergeSort(3, 3) // 遇到终止，回退到上一步
      // 排序 p1 = 2, p2 = mid + 1 = 3
      // 回退到 `mergeSort(data, 0, 3)` 执行合并逻辑
      // 排序 p1 = 0, p2 = mid + 1 = 2
      // 执行完毕回退
      // 左边数组排序完毕，右边也是如上轨迹

```

js

该算法的操作次数是可以这样计算：递归了两次，每次数据量是数组的一半，并且最后把整个数组迭代了一次，所以得出表达式 $2T(N / 2) + T(N)$ （ T 代表时间， N 代表数据量）。根据该表达式可以套用该公式得出时间复杂度为 $O(N * \log N)$

36.2.5 快排

快排的原理如下。随机选取一个数组中的值作为基准值，从左至右取值与基准值对比大小。比基准值小的放数组左边，大的放右边，对比完成后将基准值和第一个比基准值大的值交换位置。然后将数组以基准值的位置分为两部分，继续递归以上操作

以下是实现该算法的代码

```
function sort(array) {  
  checkArray(array);  
  quickSort(array, 0, array.length - 1);  
  return array;  
}  
  
function quickSort(array, left, right) {  
  if (left < right) {  
    swap(array, , right)  
    // 随机取值，然后和末尾交换，这样做比固定取一个位置的复杂度略低  
    let indexs = part(array, parseInt(Math.random() * (right - left + 1)) +  
    quickSort(array, left, indexs[0]);  
    quickSort(array, indexs[1] + 1, right);  
  }  
}  
  
function part(array, left, right) {  
  let less = left - 1;  
  let more = right;  
  while (left < more) {  
    if (array[left] < array[right]) {  
      // 当前值比基准值小，`less` 和 `left` 都加一  
      ++less;  
      ++left;  
    } else if (array[left] > array[right]) {  
      // 当前值比基准值大，将当前值和右边的值交换
```

js


```

// 并且不改变 `left`，因为当前换过来的值还没有判断过大小
swap(array, --more, left);
} else {
  // 和基准值相同，只移动下标
  left++;
}
}
// 将基准值和比基准值大的第一个值交换位置
// 这样数组就变成 `[比基准值小, 基准值, 比基准值大]`
swap(array, right, more);
return [less, more];
}

```

该算法的复杂度和归并排序是相同的，但是额外空间复杂度比归并排序少，只需 $O(\log N)$ ，并且相比归并排序来说，所需的常数时间也更少

面试题

Sort Colors：该题目来自 LeetCode，题目需要我们将 `[2,0,2,1,1,0]` 排序成 `[0,0,1,1,2,2]`，这个问题就可以使用三路快排的思想。

以下是代码实现

```

var sortColors = function(nums) {
  let left = -1;
  let right = nums.length;
  let i = 0;
  // 下标如果遇到 right，说明已经排序完成
  while (i < right) {
    if (nums[i] == 0) {
      swap(nums, i++, ++left);
    } else if (nums[i] == 1) {
      i++;
    } else {
      swap(nums, i, --right);
    }
  }
};

```

js

Kth Largest Element in an Array: 该题目来自 LeetCode, 题目需要找出数组中第 K 大的元素, 这问题也可以使用快排的思路。并且因为是找出第 K 大元素, 所以在分离数组的过程中, 可以找出需要的元素在哪边, 然后只需要排序相应的一边数组就好。

以下是代码实现

```
var findKthLargest = function(nums, k) {  
  let l = 0  
  let r = nums.length - 1  
  // 得出第 K 大元素的索引位置  
  k = nums.length - k  
  while (l < r) {  
    // 分离数组后获得比基准树大的第一个元素索引  
    let index = part(nums, l, r)  
    // 判断该索引和 k 的大小  
    if (index < k) {  
      l = index + 1  
    } else if (index > k) {  
      r = index - 1  
    } else {  
      break  
    }  
  }  
  return nums[k]  
};  
function part(array, left, right) {  
  let less = left - 1;  
  let more = right;  
  while (left < more) {  
    if (array[left] < array[right]) {  
      ++less;  
      ++left;  
    } else if (array[left] > array[right]) {  
      swap(array, --more, left);  
    } else {  
      left++;  
    }  
  }  
  swap(array, right, more);  
  return more;  
}
```

js

36.2.6 堆排序

堆排序利用了二叉堆的特性来做，二叉堆通常用数组表示，并且二叉堆是一颗完全二叉树（所有叶节点（最底层的节点）都是从左往右顺序排序，并且其他层的节点都是满的）。二叉堆又分为大根堆与小根堆

- 大根堆是某个节点的所有子节点的值都比他小
- 小根堆是某个节点的所有子节点的值都比他大

堆排序的原理就是组成一个大根堆或者小根堆。以小根堆为例，某个节点的左边子节点索引是 $i * 2 + 1$ ，右边是 $i * 2 + 2$ ，父节点是 $(i - 1) / 2$

1. 首先遍历数组，判断该节点的父节点是否比他小，如果小就交换位置并继续判断，直到他的父节点比他大
2. 重新以上操作 1，直到数组首位是最大值
3. 然后将首位和末尾交换位置并将数组长度减一，表示数组末尾已是最大值，不需要再比较大小
4. 对比左右节点哪个大，然后记住大的节点的索引并且和父节点对比大小，如果子节点大就交换位置
5. 重复以上操作 3 - 4 直到整个数组都是大根堆。

以下是实现该算法的代码

```
function heap(array) {  
  checkArray(array);  
  // 将最大值交换到首位  
  for (let i = 0; i < array.length; i++) {  
    heapInsert(array, i);  
  }  
  let size = array.length;  
  // 交换首位和末尾  
  swap(array, 0, --size);  
  while (size > 0) {  
    heapify(array, 0, size);  
    swap(array, 0, --size);  
  }  
}
```

js

```

    return array;
}

function heapInsert(array, index) {
    // 如果当前节点比父节点大，就交换
    while (array[index] > array[parseInt((index - 1) / 2)]) {
        swap(array, index, parseInt((index - 1) / 2));
        // 将索引变成父节点
        index = parseInt((index - 1) / 2);
    }
}

function heapify(array, index, size) {
    let left = index * 2 + 1;
    while (left < size) {
        // 判断左右节点大小
        let largest =
            left + 1 < size && array[left] < array[left + 1] ? left + 1 : left;
        // 判断子节点和父节点大小
        largest = array[index] < array[largest] ? largest : index;
        if (largest === index) break;
        swap(array, index, largest);
        index = largest;
        left = index * 2 + 1;
    }
}

```

- 以上代码实现了小根堆，如果需实现大根堆，只需要把节点对比反一下就好。
- 该算法的复杂度是 $O(\log N)$

36.3 链表

反转单向链表

该题目来自 LeetCode，题目需要将一个单向链表反转。思路很简单，使用三个变量分别表示当前节点和当前节点的前后节点，虽然这题很简单，但是却是一道面试常考题

以下是实现该算法的代码

```

var reverseList = function(head) {
    // 判断下变量边界问题
    if (!head || !head.next) return head

```

js

```
// 初始设置为空，因为第一个节点反转后就是尾部，尾部节点指向 null
let pre = null
let current = head
let next
// 判断当前节点是否为空
// 不为空就先获取当前节点的下一节点
// 然后把当前节点的 next 设为上一个节点
// 然后把 current 设为下一个节点，pre 设为当前节点
while(current) {
    next = current.next
    current.next = pre
    pre = current
    current = next
}
return pre
};
```

36.4 树

二叉树的先序，中序，后序遍历

- 先序遍历表示先访问根节点，然后访问左节点，最后访问右节点。
- 中序遍历表示先访问左节点，然后访问根节点，最后访问右节点。
- 后序遍历表示先访问左节点，然后访问右节点，最后访问根节点。

递归实现

递归实现相当简单，代码如下

```
function TreeNode(val) {
    this.val = val;
    this.left = this.right = null;
}
var traversal = function(root) {
    if (root) {
        // 先序
        console.log(root);
        traversal(root.left);
        // 中序
        // console.log(root);
        traversal(root.right);
        // 后序
        // console.log(root);
    }
}
```

js

```
}  
};
```

对于递归的实现来说，只需要理解每个节点都会被访问三次就明白为什么这样实现了。

非递归实现

非递归实现使用了栈的结构，通过栈的先进后出模拟递归实现。

以下是先序遍历代码实现

```
function pre(root) {  
  if (root) {  
    let stack = [];  
    // 先将根节点 push  
    stack.push(root);  
    // 判断栈中是否为空  
    while (stack.length > 0) {  
      // 弹出栈顶元素  
      root = stack.pop();  
      console.log(root);  
      // 因为先序遍历是先左后右，栈是先进后出结构  
      // 所以先 push 右边再 push 左边  
      if (root.right) {  
        stack.push(root.right);  
      }  
      if (root.left) {  
        stack.push(root.left);  
      }  
    }  
  }  
}
```

js

以下是中序遍历代码实现

```
function mid(root) {  
  if (root) {  
    let stack = [];  
    // 中序遍历是先左再根最后右  
    // 所以首先应该先把最左边节点遍历到底依次 push 进栈
```

js

```

// 当左边没有节点时，就打印栈顶元素，然后寻找右节点
// 对于最左边的叶节点来说，可以把它看成是两个 null 节点的父节点
// 左边打印不出东西就把父节点拿出来打印，然后再看右节点
while (stack.length > 0 || root) {
  if (root) {
    stack.push(root);
    root = root.left;
  } else {
    root = stack.pop();
    console.log(root);
    root = root.right;
  }
}
}
}
}

```

以下是后序遍历代码实现，该代码使用了两个栈来实现遍历，相比一个栈的遍历来说要容易理解很多

```

function pos(root) {
  if (root) {
    let stack1 = [];
    let stack2 = [];
    // 后序遍历是先左再右最后根
    // 所以对于一个栈来说，应该先 push 根节点
    // 然后 push 右节点，最后 push 左节点
    stack1.push(root);
    while (stack1.length > 0) {
      root = stack1.pop();
      stack2.push(root);
      if (root.left) {
        stack1.push(root.left);
      }
      if (root.right) {
        stack1.push(root.right);
      }
    }
    while (stack2.length > 0) {
      console.log(s2.pop());
    }
  }
}
}

```

js

中序遍历的前驱后继节点

实现这个算法的前提是节点有一个 `parent` 的指针指向父节点，根节点指向 `null`。

如图所示，该树的中序遍历结果是 `4, 2, 5, 1, 6, 3, 7`

前驱节点

对于节点 2 来说，他的前驱节点就是 4，按照中序遍历原则，可以得出以下结论

1. 如果选取的节点的左节点不为空，就找该左节点最右的节点。对于节点 1 来说，他有左节点 2，那么节点 2 的最右节点就是 5
2. 如果左节点为空，且目标节点是父节点的右节点，那么前驱节点为父节点。对于节点 5 来说，没有左节点，且是节点 2 的右节点，所以节点 2 是前驱节点
3. 如果左节点为空，且目标节点是父节点的左节点，向上寻找到第一个是父节点的右节点的节点。对于节点 6 来说，没有左节点，且是节点 3 的左节点，所以向上寻找到节点 1，发现节点 3 是节点 1 的右节点，所以节点 1 是节点 6 的前驱节点

以下是算法实现

```
function predecessor(node) {  
  if (!node) return  
  // 结论 1  
  if (node.left) {  
    return getRight(node.left)  
  } else {  
    let parent = node.parent  
    // 结论 2 3 的判断  
    while(parent && parent.right === node) {  
      node = parent  
      parent = node.parent  
    }  
    return parent  
  }  
}  
  
function getRight(node) {
```

js


```

    if (!node) return
    node = node.right
    while(node) node = node.right
    return node
}

```

后继节点

- 对于节点 2 来说，他的后继节点就是 5，按照中序遍历原则，可以得出以下结论
 1. 如果有右节点，就找到该右节点的最左节点。对于节点 1 来说，他有右节点 3，那么节点 3 的最左节点就是 6
 2. 如果没有右节点，就向上遍历直到找到一个节点是父节点的左节点。对于节点 5 来说，没有右节点，就向上寻找到节点 2，该节点是父节点 1 的左节点，所以节点 1 是后继节点

以下是算法实现

```

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

function getLeft(node) {
  if (!node) return
  node = node.left
  while(node) node = node.left
  return node
}

```

js

树的深度

树的最大深度：该题目来自 Leetcode，题目要求求出一颗二叉树的最大深度

以下是算法实现

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

js

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

36.5 动态规划

- 动态规划背后的基本思想非常简单。就是将一个问题拆分为子问题，一般来说这些子问题都是非常相似的，那么我们可以通过只解决一次每个子问题来达到减少计算量的目的。
- 一旦得出每个子问题的解，就存储该结果以便下次使用。

斐波那契数列

斐波那契数列就是从 0 和 1 开始，后面的数都是前两个数之和

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89....

那么显而易见，我们可以通过递归的方式来完成求解斐波那契数列

```
function fib(n) {  
  if (n < 2 && n >= 0) return n  
  return fib(n - 1) + fib(n - 2)  
}  
fib(10)
```

js

以上代码已经可以完美的解决问题。但是以上解法却存在很严重的性能问题，当 n 越大的时候，需要的时间是指数增长的，这时候就可以通过动态规划来解决这个问题。

动态规划的本质其实就是两点

- 自底向上分解子问题
- 通过变量存储已经计算过的解

根据上面两点，我们的斐波那契数列的动态规划思路也就出来了

- 斐波那契数列从 0 和 1 开始，那么这就是这个子问题的最底层
- 通过数组来存储每一位所对应的斐波那契数列的值

```
function fib(n) {  
  let array = new Array(n + 1).fill(null)  
  array[0] = 0  
  array[1] = 1  
  for (let i = 2; i <= n; i++) {  
    array[i] = array[i - 1] + array[i - 2]  
  }  
  return array[n]  
}  
fib(10)
```

js

0 - 1背包问题

该问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。每个问题只能放入至多一次。

假设我们有以下物品

物品 ID / 重量	价值
1	3
2	7
3	12

- 对于一个总容量为 5 的背包来说，我们可以放入重量 2 和 3 的物品来达到背包内的物品总价值最高。
- 对于这个问题来说，子问题就两个，分别是放物品和不放物品，可以通过以下表格来理解子问题

物品 ID / 剩余容量	0	1	2	3	4	5
1	0	3	3	3	3	3
2	0	3	7	10	10	10
3	0	3	7	12	15	19

直接来分析能放三种物品的情况，也就是最后一行

- 当容量少于 3 时，只取上一行对应的数据，因为当前容量不能容纳物品 3
- 当容量为 3 时，考虑两种情况，分别为放入物品 3 和不放物品 3
 - 不放物品 3 的情况下，总价值为 10
 - 放入物品 3 的情况下，总价值为 12，所以应该放入物品 3
- 当容量为 4 时，考虑两种情况，分别为放入物品 3 和不放物品 3
 - 不放物品 3 的情况下，总价值为 10
 - 放入物品 3 的情况下，和放入物品 1 的价值相加，得出总价值为 15，所以应该放入物品 3
- 当容量为 5 时，考虑两种情况，分别为放入物品 3 和不放物品 3
 - 不放物品 3 的情况下，总价值为 10
 - 放入物品 3 的情况下，和放入物品 2 的价值相加，得出总价值为 19，所以应该放入物品 3

以下代码对照上表更容易理解

```
/**
 * @param {*} w 物品重量
 * @param {*} v 物品价值
 * @param {*} C 总容量
 * @returns
 */
function knapsack(w, v, C) {
  let length = w.length
  if (length === 0) return 0

  // 对照表格，生成的二维数组，第一维代表物品，第二维代表背包剩余容量
  // 第二维中的元素代表背包物品总价值
  let array = new Array(length).fill(new Array(C + 1).fill(null))

  // 完成底部子问题的解
  for (let i = 0; i <= C; i++) {
    // 对照表格第一行， array[0] 代表物品 1
    // i 代表剩余总容量
```

js

```

// 当剩余总容量大于物品 1 的重量时，记录下背包物品总价值，否则价值为 0
array[0][i] = i >= w[0] ? v[0] : 0
}

// 自底向上开始解决子问题，从物品 2 开始
for (let i = 1; i < length; i++) {
  for (let j = 0; j <= C; j++) {
    // 这里求解子问题，分别为不放当前物品和放当前物品
    // 先求不放当前物品的背包总价值，这里的值也就是对应表格中上一行对应的值
    array[i][j] = array[i - 1][j]
    // 判断当前剩余容量是否可以放入当前物品
    if (j >= w[i]) {
      // 可以放入的话，就比大小
      // 放入当前物品和不放入当前物品，哪个背包总价值大
      array[i][j] = Math.max(array[i][j], v[i] + array[i - 1][j - w[i]])
    }
  }
}
return array[length - 1][C]
}

```

最长递增子序列

最长递增子序列意思是在一组数字中，找出最长一串递增的数字，比如

0, 3, 4, 17, 2, 8, 6, 10

对于以上这串数字来说，最长递增子序列就是 0, 3, 4, 8, 10，可以通过以下表格更清晰的理解




数字	0	3	4	17	2	8	6	10
长度	1	2	3	4	2	4 4	5	

通过以上表格可以很清晰的发现一个规律，找出刚好比当前数字小的数，并且在小的数组成的长度基础上加一。

这个问题的动态思路解法很简单，直接上代码

```
function lis(n) {  
  if (n.length === 0) return 0  
  // 创建一个和参数相同大小的数组，并填充值为 1  
  let array = new Array(n.length).fill(1)  
  // 从索引 1 开始遍历，因为数组已经所有都填充为 1 了  
  for (let i = 1; i < n.length; i++) {  
    // 从索引 0 遍历到 i  
    // 判断索引 i 上的值是否大于之前的值  
    for (let j = 0; j < i; j++) {  
      if (n[i] > n[j]) {  
        array[i] = Math.max(array[i], 1 + array[j])  
      }  
    }  
  }  
  let res = 1  
  for (let i = 0; i < array.length; i++) {  
    res = Math.max(res, array[i])  
  }  
  return res  
}
```

37 css常考面试题解析

- [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](#) 
- [前端开发者面试问题 - CSS 部分](#) 