# 🚀 JavaScript 基础面试题

## var, let, const 区别

var: 函数作用域，可重复声明，存在变量提升，可以修改。

var 声明的变量在整个函数内有效，如果在函数外声明，则是全局变量。  
let: 块级作用域，不可重复声明，无变量提升，可以修改。

**块级作用域 (Block Scope)定义：**

**变量的作用范围限定在 {} 花括号内，无论是函数、if、for、while 等块中。**

const: 块级作用域，不可重复声明，无变量提升，声明后不可重新赋值（对象和数组的属性/元素可以改）

**简单建议：**

* 只使用 let 和 const。
* **默认用 const**，如果确定变量值会变，再用 let。
* 避免使用 var，容易引起作用域混乱和提升问题。

## 什么是闭包 (Closure)?

闭包就是一个函数，可以“记住”并访问它定义时的外部作用域中的变量，即使这个外部函数已经执行完毕。可以把闭包想象成 **快照，记住了当时操作时的值。**

**闭包的关键点**

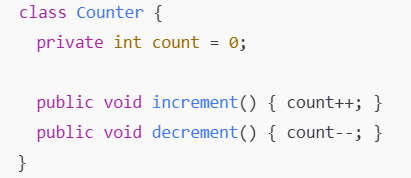
1. **函数嵌套**：需要有一个函数包着另一个函数。
2. **内部函数引用外部函数的变量**。
3. **外部函数已经执行完毕，但内部函数仍然能访问外部变量。**

**闭包的用途**

1. **数据私有化**  
   利用闭包实现“私有变量”，防止外部随意修改。

在 JavaScript 中，默认所有变量、对象、属性都是**公开的**，不像某些语言（如 Java/C#）有 **private** 关键字。但有时候我们希望：**不让外界直接访问或篡改变量，只能通过特定方法读取或修改。**闭包就可以帮我们做到这一点！

此时：count可以看作private变量。可以把它理解为 **Java 的 private 成员变量 + public 方法**：





1. **防止全局污染**  
   可以通过闭包封装变量，避免变量跑到全局作用域。
2. 在异步操作中保存上下文

var输出 3 3 3

let输出0 1 2

**let 的块级作用域**确保每次循环中的 i 值是独立的，即每次 setTimeout 都捕获当前循环的 i 值。

**var 的函数作用域**导致所有的回调函数共享同一个 i，所以会输出相同的值（3）。

总结：

闭包让函数即使在外部函数执行完后，依然可以访问到外部函数的变量。

## == 和 === 的区别

==: 先进行类型转换再比较  
===: 严格相等，不进行类型转换。只有在值和类型都相同的情况下，=== 才会返回 true。

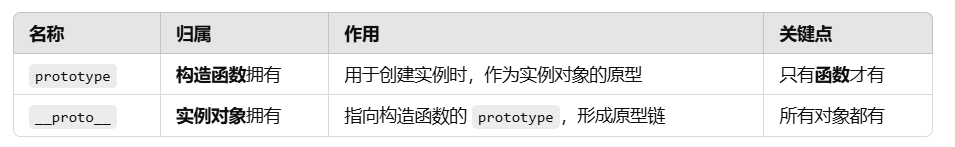
## 原型链 (Prototype Chain) 是什么？

JS 对象通过 \_\_proto\_\_ 指向其原型对象，形成一个链式结构，是继承的基础。

在 JavaScript 中，**函数本质上是一种对象（Object，是一个key value的键值对{}）**，具有特殊的内部方法 [[Call]] 和 [[Construct]]。当一个函数被用作构造函数（使用 new 关键字调用）时，JavaScript 会调用其 [[Construct]] 方法，并返回一个新的对象实例。因此，函数可以使用 new 关键字，是因为函数具有构造行为（constructor behavior）。

不是所有函数都可以用 new！

例如箭头函数不能用 new



**详细讲解：**

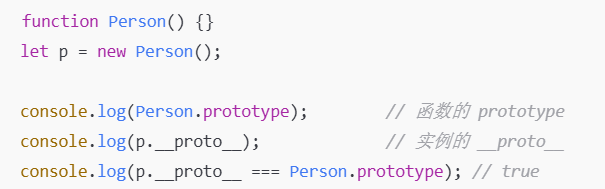
**1️ prototype —— 函数的“模板”**

* **作用**：prototype 是 **构造函数**的一个属性。
* **谁有**：**只有函数（包括自定义函数、内置构造函数 like Function, Object, Array 等）才有 prototype 属性**。
* **用途**：它是实例对象的原型对象，**实例会通过 \_\_proto\_\_ 指向构造函数的 prototype**，用来共享属性和方法。

**2️ \_\_proto\_\_ —— 对象的“指针”**

* **作用**：\_\_proto\_\_ 是 **实例对象**的一个属性。
* **谁有**：**所有对象（包括数组、函数本身、普通对象、实例对象）都有 \_\_proto\_\_**。
* **用途**：它指向创建它的构造函数的 prototype，从而建立 **原型链**。

实例对象的 \_\_proto\_\_ 指向它构造函数的 prototype。实例对象用来找到构造函数的 prototype，形成原型链



## 事件委托 (Event Delegation) 是什么？

利用事件冒泡，将事件绑定在父元素上，通过 event.target 处理子元素事件，减少事件绑定数量，提高性能。

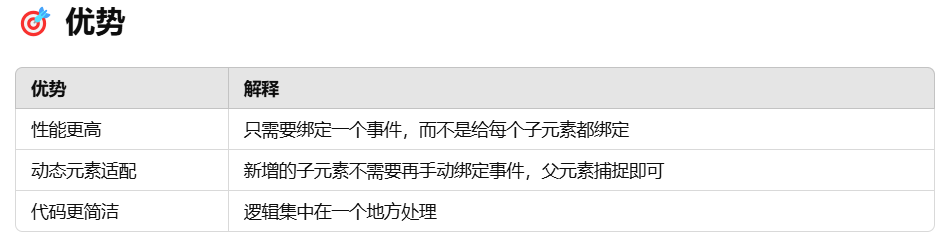
**事件委托** 是指：**利用事件冒泡机制，把子元素的事件处理，统一交给父元素来处理。**

**事件冒泡过程：**

当你点击一个子元素时，事件会从 **子元素** 一直冒泡到 **父元素**、再到 **祖先元素**，最终到达 document。可以使用 stopPropagation() 来阻止冒泡。

换句话说：

👉 **不直接给每个子元素绑定事件**，  
👉 **而是把事件绑定在它的父级或更高层元素上**，  
👉 **事件发生时，通过 event.target 判断具体是哪个子元素触发的，再做相应处理**。



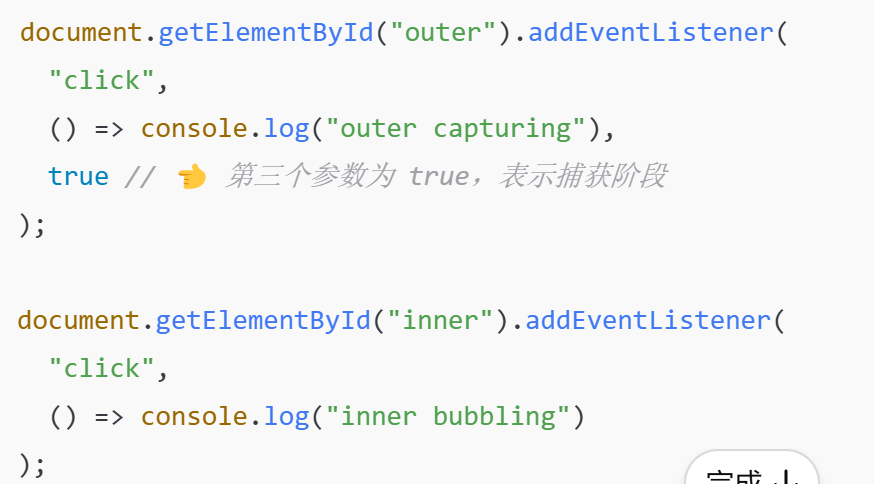
事件委托利用事件冒泡，把子元素的事件绑定在父元素上，减少事件绑定数量，提高性能，适应动态元素。

## 事件捕获

**事件传播的三个阶段：**

1. **捕获阶段（Capturing phase）** 👉 从 document 到事件目标元素（向下）
2. **目标阶段（Target phase）** 👉 到达事件目标元素本身
3. **冒泡阶段（Bubbling phase）** 👉 从目标元素向 document（向上）

事件捕获是事件传播的第一个阶段，事件从最外层节点向目标元素传播。默认情况下事件监听器在冒泡阶段触发，只有显式传入 useCapture = true 才会在捕获阶段触发。



## Promise

Promise 是 JavaScript 中处理 **异步操作** 的一种重要对象和编程模式，它代表一个 **未来可能完成也可能失败的操作的结果**。

简单来说，Promise 就像是一个 **承诺**，告诉你：  
“**我现在可能还没完成（比如数据还没从服务器返回），但将来一定会告诉你是成功还是失败。**”

**Promise 的三个状态**

一个 Promise 一开始是 pending（**进行中**），最终会变成以下两种状态之一：

1. **fulfilled（已完成 / 成功）**  
   异步操作成功完成，返回结果。
2. **rejected（已失败）**  
   异步操作失败，返回错误原因。
3. **pending（未完成）**  
   操作还在进行中，既没有成功，也没有失败。

**一旦状态改变，就不会再变**（从 pending → fulfilled 或 rejected，不会再回到 pending）。

Promise 的设计思想是为了 **异步编程**，异步操作完成后，可能会有 **结果数据** 或 **错误信息** 需要传递出去。

* **resolve(参数)** → 把成功的**结果数据**传给 .then() 的回调
* **reject(参数)** → 把失败的**错误原因**传给 .catch() 的回调

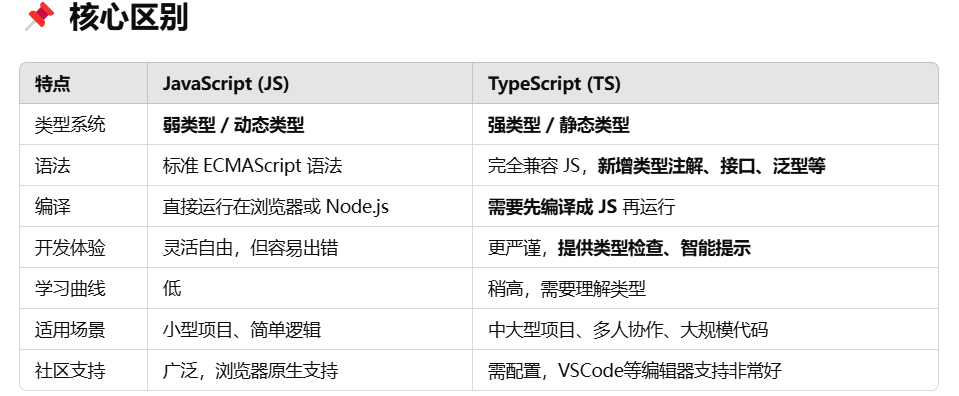


## ES12

# 🔥 TypeScript 基础面试题

## TypeScript 和 JavaScript 区别

TypeScript 是 JavaScript 的超集，支持静态类型检查、接口、泛型、枚举等，编译成 JS 运行。



TypeScript = JavaScript + 类型系统

## interface 和 type 的区别

interface：主要用于定义对象、类的结构，可继承扩展，interface可以声明合并。  
type：可以表示更复杂的类型（如联合类型、交叉类型），但不可重复声明。

**简单对象 → interface 更好**  
**复杂类型组合 → type 更灵活**



## 什么是泛型 (Generics)?

允许定义函数或接口时不指定具体类型，而在调用时传入具体类型，提高代码复用性和类型安全。

## TypeScript 的枚举 (enum)

**JavaScript 中没有enum**

enum Direction { Up, Down, Left, Right }，默认从 0 开始递增，也可以手动赋值。

**数字枚举**：可以自动递增，也支持反向映射。

**字符串枚举**：必须手动赋值，**更安全，但无反向映射**。

**const enum**：适合性能敏感场景，减少编译后代码体积。



# 🌟 Angular 基础面试题

## Angular 的核心概念

组件 (Component)、模块 (NgModule)、服务 (Service) + 依赖注入 (DI)、路由 (Router)、双向绑定 [(ngModel)]、生命周期钩子如 ngOnInit, ngOnDestroy。

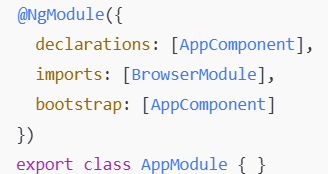
## 1️ Component (组件)

* 核心！Angular 应用是 **由组件树构成**。
* 每个组件包含：
  + **HTML 模板**
  + **CSS 样式**
  + **TypeScript 类 (逻辑)**
  + **元数据 (装饰器 @Component)**

## 2️ Module (模块)

* **@NgModule** 修饰的类
* 组织组件、指令、管道和服务
* **每个 Angular 应用至少一个根模块 AppModule**

模块是 Angular 应用的基础构建块，它将组件、指令、管道、服务以及其他功能相关的部分聚集在一起，使得开发和维护变得更加简洁和高效。



## 3️ Template (模板)

* 组件的 **HTML 部分**
* 支持 **数据绑定、指令、表达式**

## 数据绑定



## 5️ Directive (指令)

**指令就是带有 @Directive 装饰器的类，它可以用来控制 DOM 的显示、样式或行为。**

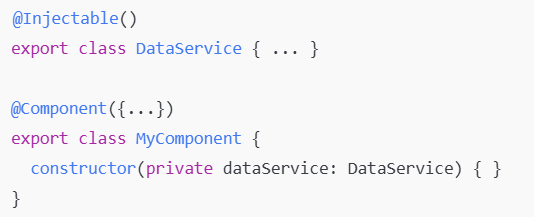
****





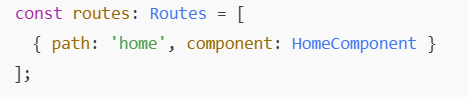
## 6️ Service (服务) & Dependency Injection

* **Service**：封装业务逻辑（如 API 调用）
* **依赖注入**：自动提供和管理服务实例



## 7️ Routing (路由)

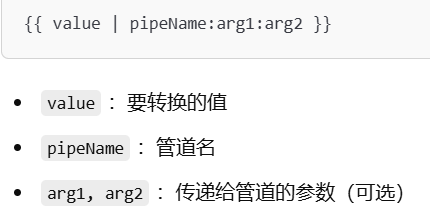
* 配置 URL → 组件映射
* 实现单页应用导航



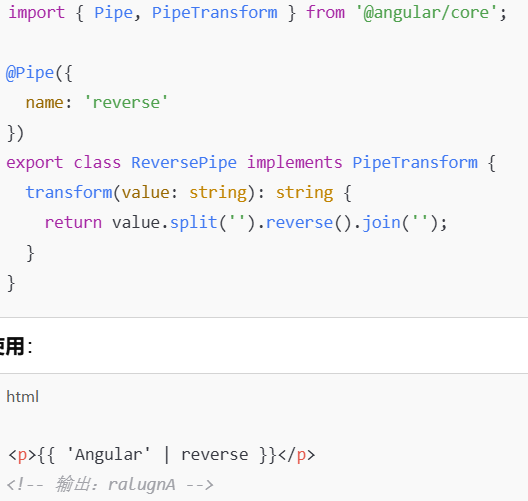
## 8️ Pipe (管道)

**Pipe 是一种数据转换器，可以在模板表达式中对数据进行格式化，比如日期格式化、货币显示、字母大小写转换等。**

* 格式化数据显示
* 常用如：date, currency, uppercase





自定义管道：



在 Angular 中，纯函数是指不会产生副作用（例如修改全局状态、I/O 操作等）、且相同输入永远返回相同输出的函数，常用于自定义管道和优化变更检测（如 OnPush 策略）中，以提升性能。

## 9️ Lifecycle Hooks (生命周期钩子)

* 组件生命周期的重要节点
* 在组件的生命周期内，每个生命周期钩子都有特定的触发时机，你可以在这些时机插入逻辑来处理数据、初始化组件、响应输入的变化、清理资源等任务。这有助于确保你的组件在 **不同的生命周期阶段** 里表现正确，避免出现未初始化、数据不一致、资源泄漏等问题。

**ngAfterViewInit (视图初始化)**

* **调用时机**：当 **组件的视图** 初始化完成时调用。此时视图中所有子组件、DOM 元素已经渲染完成。
* **常用场景**：通常用于访问和操作组件视图的 DOM 元素。

**ngOnInit (组件初始化)**

* **调用时机**：当组件 **初始化时**，即 Angular 初始化完输入属性后调用一次。
* **常用场景**：用于初始化组件的数据或执行一次性的操作。

**ngOnDestroy (销毁)**

* **调用时机**：当组件或指令被销毁时调用。
* **常用场景**：用于清理资源，如解除订阅、销毁定时器、清除事件监听等。

**生命周期钩子调用顺序**

1. ngOnChanges
2. ngOnInit
3. ngDoCheck
4. ngAfterContentInit
5. ngAfterContentChecked
6. ngAfterViewInit
7. ngAfterViewChecked
8. ngOnDestroy（销毁时）

## Observable & RxJS

* 处理异步数据流
* Observable 是一个数据流模型，允许我们处理异步操作和事件流，它是 RxJS 的核心。
* RxJS 是一个功能强大的库，提供了大量的操作符，可以对 Observable 数据流进行转换、组合、过滤等操作，极大地简化了异步代码的编写。
* Angular 使用 RxJS 来处理异步操作，如 HTTP 请求、事件流等，帮助开发者以声明式的方式管理异步操作。

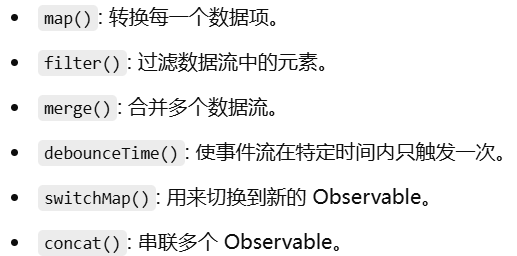
**Observable** 是一种用于表示 **异步数据流** 的模式，它允许我们订阅（observe）某个数据源的变化，并在数据变化时做出反应。你可以把它想象成一个类似于流的数据源，数据不断地流入流中，订阅者能够实时收到更新。

**Observable的基本特性：**

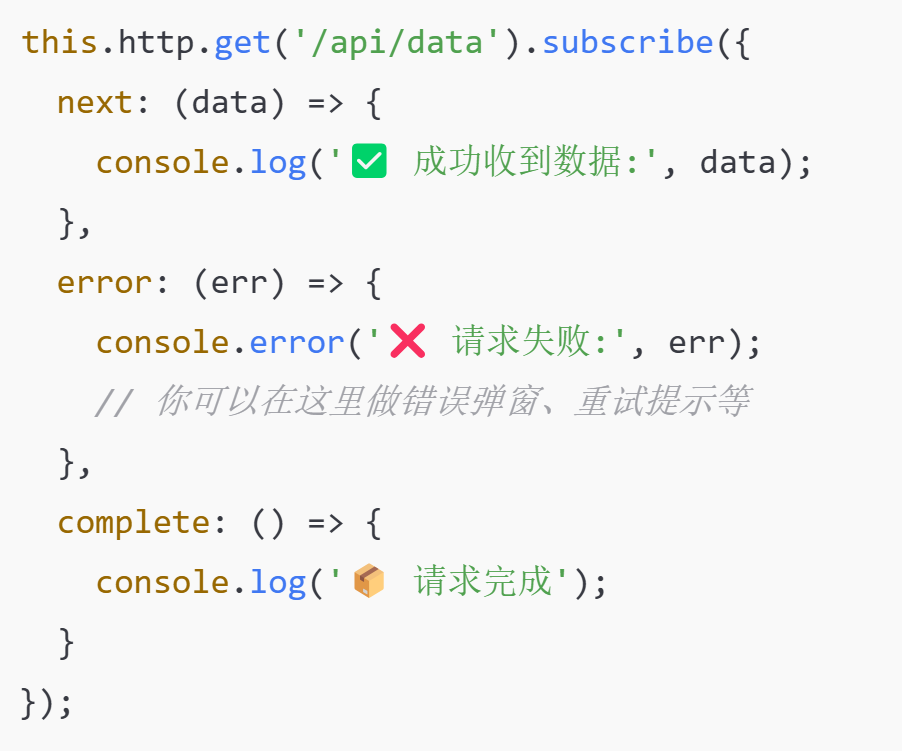
* **推送数据：** Observable 会主动推送数据给订阅者，而不是拉取数据。
* **懒执行：** Observable 是惰性求值的，只有在你订阅它时才会开始执行操作。
* **可以发送多个值：** Observable 可以连续发送多个值，而不像 Promise 只能返回单个值。
* **支持异步操作：** 它非常适合处理异步操作，如 HTTP 请求、用户输入、定时器等。

**关键术语：**

* **Subscriber (订阅者)：** 订阅 Observable 的对象，接收数据流。
* **Observer (观察者)：** 是一个对象，定义了 next(), error(), complete() 等回调来处理不同的事件。
* **Operators (操作符)：** 对 Observable 数据流进行转换的函数，如 map(), filter(), merge(), debounceTime() 等。



例子：http请求：



如果请求失败了，会不会走 complete 回调？

**答：**

不会。只要走了 error，就不会再走 complete，二者互斥。

## 什么是依赖注入 (Dependency Injection)?

Angular 内置 DI 系统，自动提供依赖对象，避免手动创建实例，降低耦合。



**@Injectable()**：这是一个装饰器，表示该类可以被注入到其他类中。通过 { providedIn: 'root' }，Angular 会在根注入器中注册该服务，意味着它是全局可用的，单例的。

在组件中，你可以通过构造函数注入服务的实例。

如果你只希望服务在某个组件的作用域内有效，可以在该组件的 providers 数组中提供服务，这样该服务会在该组件的生命周期内是单例的。

**依赖注入的作用域**

依赖注入的作用域主要由服务的提供方式决定。服务可以有不同的作用域：

* **全局单例（providedIn: 'root'）**：如果服务在根注入器中提供，那么这个服务是全局单例的，整个应用中都能共享同一个实例。
* **模块作用域**：如果服务只在特定模块内提供，那么它的作用域就仅限于该模块，其他模块无法直接访问。
* **组件作用域**：如果服务在组件内提供，那么它只会在该组件及其子组件内有效，属于局部作用域。

**依赖注入的优势**

1. **松耦合**：组件和服务不需要彼此直接依赖，Angular 会负责注入所需的服务。这使得组件之间更加独立，便于维护和测试。
2. **可测试性**：服务的实例可以方便地替换为模拟（mock）服务或不同的实现，方便单元测试。
3. **灵活性和可扩展性**：通过不同的注入方式和提供者，Angular 提供了高度的灵活性，可以根据需求调整服务的作用域。
4. **自动管理依赖**：Angular 会自动管理对象的生命周期，无需手动创建和销毁依赖。

## 什么是单向数据绑定和双向数据绑定？

单向绑定：数据从组件 -> 视图

**单向数据绑定 (One-Way Data Binding)**

**单向数据绑定** 是指数据从 **模型 (Model)** 流向 **视图 (View)**，或者从 **视图** 流向 **模型**，数据只能单向流动，通常是从模型到视图。这意味着当数据发生变化时，视图会自动更新，但视图中的变化不会影响数据。

**单向数据绑定的工作方式：**

1. **从模型到视图**：模型中的数据变化会直接反映到视图中，但用户的输入不会影响模型中的数据。
   * **举例**：在 Angular 中，通过 **{{}}（插值表达式）** 来进行单向数据绑定，将模型的值渲染到视图中：

双向绑定：数据组件 <-> 视图 同步更新

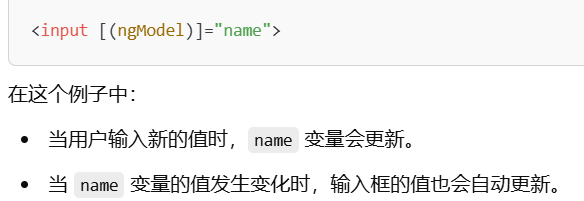
**双向数据绑定 (Two-Way Data Binding)**

**双向数据绑定** 是指数据在 **模型 (Model)** 和 **视图 (View)** 之间是双向流动的，视图的变化会自动反映到模型中，而模型的变化也会自动反映到视图中。这意味着用户输入的数据会更新模型，而模型的变化也会自动更新视图。

**双向数据绑定的工作方式：**

1. **视图到模型**：当用户在视图中修改数据（例如输入框）时，视图中的变化会同步到模型中。
2. **模型到视图**：当模型中的数据发生变化时，视图会自动更新，展示新的数据。

在 Angular 中，双向数据绑定通常通过 **ngModel** 指令实现：



# ⚛️ React 基础面试题

## React 的核心概念

组件 (Function & Class)、Props 和 State、JSX、虚拟 DOM (Virtual DOM)、生命周期方法（Class: componentDidMount / Hooks: useEffect）

## 什么是虚拟 DOM？

React 使用内存中的虚拟 DOM 作为 UI 表达，更新时先对比新旧虚拟 DOM，最后批量更新真实 DOM，提升性能。

## React 中的 key 作用？

提高列表渲染性能，帮助 React 识别元素是否变化，避免不必要的重新渲染。

## 受控组件 vs 非受控组件

受控组件：表单元素值由 state 管控  
非受控组件：使用 ref 直接访问 DOM 节点值

## React Hooks 常用的有哪些？

useState, useEffect, useContext, useRef, useMemo, useCallback

# 💡 前端进阶 & 项目实战题

## JavaScript 中的事件循环 (Event Loop) 是什么？

Event Loop 负责执行代码、收集和处理事件以及执行子任务。  
包括宏任务（setTimeout, setInterval, setImmediate, I/O）和微任务（Promise.then, MutationObserver, queueMicrotask）。微任务队列会在当前宏任务执行完后立即清空，确保优先级更高。

## 什么是防抖 (debounce) 和节流 (throttle)？

防抖：n 秒内只执行最后一次，常用于输入框搜索防止频繁请求。  
节流：每隔 n 秒执行一次，常用于滚动、resize 事件防止高频触发。

## 如何优化 React 应用性能？

- 使用 React.memo 避免不必要的组件重渲染  
- 使用 useMemo, useCallback 记忆化函数和值  
- 列表渲染中合理设置 key  
- 懒加载组件 (React.lazy + Suspense)  
- 虚拟化列表 (如 react-window, react-virtualized)  
- 减少 state 层级，避免过度 lifting state up

## 在 Angular 项目中，如何进行模块懒加载？

通过 Angular Router 的 loadChildren 属性实现模块懒加载，例如：  
{ path: 'admin', loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule) }

## TypeScript 中如何进行类型守卫 (Type Guards)？

- 使用 typeof 检查基本类型  
- 使用 instanceof 检查实例  
- 自定义类型保护函数 (返回值为参数 is 某类型)

## React 项目中常用状态管理方案有哪些？

- React Context API  
- Redux / Redux Toolkit  
- Zustand  
- Recoil  
- MobX

## 项目中常见的前端安全问题及防护措施

- XSS（跨站脚本攻击）：对输入内容进行转义或使用 DOMPurify  
- CSRF（跨站请求伪造）：使用 SameSite Cookie、防止自动提交  
- 点击劫持：设置 X-Frame-Options: DENY  
- 依赖漏洞：定期升级依赖包，使用 npm audit/yarn audit 检查

## 前端性能优化常见手段

- 资源懒加载 (图片/组件/路由)  
- 使用 CDN 加速资源  
- 开启 gzip / brotli 压缩  
- 合理使用缓存 (HTTP Cache, Service Worker)  
- Tree Shaking, 代码拆分  
- 减少重绘与回流  
- 图片优化 (WebP, 图片压缩)

## XSS（跨站脚本攻击，Cross-Site Scripting）

**📌 是什么？**

XSS 是一种通过向网页注入恶意脚本，诱导用户在浏览器上执行攻击者代码的方式。

**📊 分类：**



**⚠️ XSS 危害：**

* 窃取用户的 Cookie
* 伪造用户操作
* 诱导用户跳转钓鱼网站

**✅ 防御 XSS：**

1. **对输入进行过滤/转义** → 例如将 < > & " ' 变成安全字符。
2. **Content Security Policy (CSP)** → 限制脚本来源。
3. **避免直接把用户输入渲染到 HTML 结构中。**

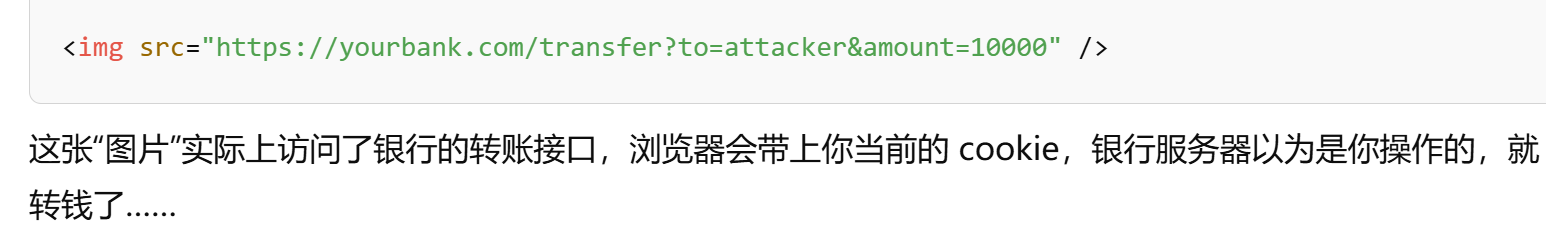
## CSRF（跨站请求伪造，Cross-Site Request Forgery）

**📌 是什么？**

CSRF 是攻击者诱导用户浏览器，**在用户登录某个网站后**，利用你当前的登录状态，伪造请求发给目标网站，执行非用户本意的操作。

**📊 场景：**

你登录了银行网站，还没退出登录。这时你在网站的评论区不小心点击了攻击者精心设计的链接：



**💥 本质：**

* 浏览器会自动带上登录状态（cookie）
* 攻击者伪造请求
* 用户不知情地执行了敏感操作（比如：转账、修改密码、删除账号）

**⚠️ CSRF 危害：**

* 伪造用户发起敏感操作
* 修改用户资料、密码
* 资金交易、购买物品等

**✅ 防御 CSRF：**

1. **Token 验证** → 每个请求带随机生成的 CSRF Token，攻击者无法猜测。
2. **SameSite Cookie 属性** → 阻止第三方网站携带 Cookie。
3. **Referer 检查** → 判断请求来源是否可信。

CSRF 成立的前提是：

攻击者伪造请求，并**自动携带用户的身份凭证**（通常是 Cookie）

浏览器在发送请求时会**自动附带 Cookie**，但**不会自动添加 Authorization 请求头**。

# Sebill-UI

## Token如何设置

通过统一的认证服务：tokengenerator获取token，然后放在appsetting中（保存在内存中）。在每一次发送网络请求的时候，将Bearer Token 放在 Authorization 请求头中可以有效防止 CSRF。

