设计模式（10题）

问题 1：什么是单例模式（Singleton Pattern）？它在前端开发中的典型应用场景是什么？

答案：

单例模式是一种确保一个类只有一个实例，并提供一个全局访问点的设计模式。无论该类被实例化多少次，返回的都是同一个对象。

前端应用场景：

- **全局状态管理：** 像 Vuex 或 Redux 这样的状态管理库，其核心的 `store` 就是一个单例。整个应用程序中只有一个 `store` 实例，用于存储和管理所有组件共享的状态，确保了状态的一致性。
- **全局配置对象：** 应用程序的配置信息（如 API 地址、主题设置、功能开关等）可以被封装在一个单例对象中，方便在任何地方读取和修改，且保持唯一。
- **UI组件库的实例管理：** 像 `Message` 或 `Notification` 这样的全局提示组件，通常会使用单例模式来管理它们的实例。这样可以确保在任何时刻屏幕上只有一个或一组统一管理的提示框，避免了多个实例重叠或冲突。

问题 2：请解释一下工厂模式（Factory Pattern），并举一个在前端中应用的例子。

答案：

工厂模式是一种创建型设计模式，它提供了一种创建对象的最佳方式，而无需向客户端暴露创建逻辑。它定义一个用于创建对象的接口，让子类决定实例化哪一个类。

前端应用例子：

假设你需要根据不同的用户角色（如'admin', 'editor', 'guest'）创建不同的用户操作菜单组件。你可以创建一个菜单工厂函数：

代码块

```
function createMenu(role) {
    switch (role) {
        case 'admin':
            return new AdminMenu(); // 返回管理员菜单组件
        case 'editor':
            return new EditorMenu(); // 返回编辑者菜单组件
        default:
            return new GuestMenu(); // 返回访客菜单组件
    }
}

const userMenu = createMenu(currentUser.role);
```

这样，创建具体菜单的逻辑被封装在工厂函数中，调用者只需关心传入正确的角色即可，代码更易于维护和扩展。

问题 3：观察者模式（Observer Pattern）和发布-订阅模式（Publish-Subscribe Pattern）有什么区别？

答案：

两者都用于处理对象间的解耦和消息传递，但存在一个关键区别：

- **观察者模式：** 观察者（Observer）直接订阅主题（Subject），并维护在主题的订阅者列表中。当主题状态变化时，它会直接通知所有观察者。这是一个紧耦合的关系，主题直接知道它的观察者。
- **发布-订阅模式：** 发布者（Publisher）和订阅者（Subscriber）之间是完全解耦的，它们不直接交互。它们通过一个中心的事件总线（Event Bus / Broker）进行通信。发布者向总线发布事件，而订阅者向总线订阅事件。发布者不知道谁是订阅者，反之亦然。

简单来说： 发布-订阅模式多了一个“中间商”（事件总线），实现了更彻底的解耦。DOM事件监听（`addEventListener`）更接近观察者模式，而像 Vue.js 中的 `$on` 和 `$emit` 或 Node.js 的 `EventEmitter` 则属于发布-订阅模式。

问题 4：什么是模块模式（Module Pattern）？它解决了 JavaScript 中的什么问题？

答案：

模块模式是一种利用闭包（Closure）来创建私有作用域和公有接口的设计模式。它允许你封装变量和函数，只暴露需要对外访问的部分。

它主要解决了 JavaScript 中的两个问题：

1. **全局作用域污染：** 在模块模式出现之前，大量的变量和函数都定义在全局作用域中，容易导致命名冲突和代码难以维护。模块模式将代码封装在函数作用域内，避免了这个问题。
2. **缺乏私有变量：** JavaScript 在ES6之前没有原生的私有属性概念。通过闭包，模块模式可以模拟出私有变量（定义在立即执行函数IIFE内部，但不返回）和公有方法（作为返回对象的方法）。

示例代码：

代码块

```
const MyModule = (function() {
    let privateVariable = '我是私有的';

    function privateMethod() {
        console.log(privateVariable);
    }

    return {
        publicMethod: function() {
            privateMethod();
        }
    };
```

```
})();

MyModule.publicMethod(); // 输出: "我是私有的"
// console.log(MyModule.privateVariable); // undefined，无法访问
```

问题 5：请解释装饰器模式（Decorator Pattern），它在现代前端框架中有什么应用？

答案：

装饰器模式是一种结构型设计模式，它允许在不修改原始对象代码的情况下，动态地为对象添加新的功能或行为。它通过创建一个包装器（wrapper）对象来实现，该包装器持有对原始对象的引用。

在现代前端框架中的应用：

- **React 中的高阶组件 (HOC - Higher-Order Components)：** HOC 本质上就是装饰器模式的一种实现。它是一个函数，接受一个组件作为参数，并返回一个新的增强版组件。例如，React Router 的 `withRouter` 就是一个 HOC，它将路由相关的 props（`match`，`location`，`history`）注入到被包裹的组件中。
- **Angular 和 TypeScript 中的装饰器：** Angular 大量使用 TypeScript 的装饰器语法（`@Component`，`@Injectable`，`@Input` 等）。这些装饰器在编译时为类、属性或方法附加元数据或修改其行为，例如 `@Component` 装饰器将一个普通类标记为 Angular 组件，并为其提供模板、样式等元数据。

问题 6：什么是策略模式（Strategy Pattern）？它如何帮助我们优化代码？

答案：

策略模式是一种行为设计模式，它定义了一系列算法，并将每个算法封装起来，使它们可以互相替换。策略模式让算法的变化独立于使用算法的客户。

如何优化代码：

它可以帮助我们消除冗长、复杂的 `if...else` 或 `switch...case` 语句。当业务逻辑中存在多种条件和多种处理方式时，我们可以将每种处理方式封装成一个独立的策略对象，然后根据上下文动态选择并执行相应的策略。

例子： 表单验证。与其写一长串 `if/else` 来检查不同类型的验证规则（如非空、最小长度、是否为邮件格式），不如将每条规则定义为一个策略对象。

代码块

```
const strategies = {
    isNotEmpty: (value, errorMsg) => value === '' ? errorMsg : void 0,
    minLength: (value, length, errorMsg) => value.length < length ? errorMsg : void 0
};
// 验证时，根据需要动态调用策略
const error = strategies.isNotEmpty(inputValue, '用户名不能为空');
```

问题 7：代理模式（Proxy Pattern）在前端有什么实际用途？请至少举出两个例子。

答案：

代理模式是为一个对象提供一个替代品或占位符，以控制对这个对象的访问。

前端实际用途：

1. **事件代理/事件委托（Event Delegation）：** 这是代理模式最经典的用途之一。当一个容器内有大量子元素需要监听事件时，我们不必为每个子元素都绑定事件处理器，而是将事件处理器绑定在父容器上。利用事件冒泡机制，父容器可以“代理”所有子元素的事件，通过 `event.target` 来判断事件源并执行相应逻辑。这极大地提高了性能并简化了代码。

2. **数据请求代理（API Proxy）：** 在开发环境中，为了解决跨域问题（CORS），我们常常配置一个本地代理服务器。前端发送的 API 请求实际上是发往同源的代理服务器，然后由代理服务器将请求转发给真实的目标 API 服务器。这个代理服务器对前端开发者是透明的，起到了一个中间转发的作用。
3. **ES6 的 `Proxy` 对象：** ** JavaScript ES6 提供了原生的 `Proxy` 对象，可以用来创建一个对象的代理，从而实现对该对象基本操作（如读取、设置属性等）的拦截和自定义行为。Vue 3 的响应式系统就是基于 `Proxy` 实现的，通过代理数据对象，Vue 可以精确地监听到对象属性的读取和修改，从而高效地触发视图更新。

问题 8：在React中，自定义Hooks（Custom Hooks）可以看作是哪种设计模式的应用？请说明理由。

答案：

自定义 Hooks 可以看作是 **策略模式（Strategy Pattern）** 和 **组合模式（Composition Pattern）** 的一种应用。

- **策略模式：** 每个自定义 Hook 封装了一段特定的、可复用的逻辑（一个“策略”），例如数据获取逻辑 (`useFetch`)、订阅事件逻辑 (`useEventListener`) 或表单状态管理逻辑 (`useForm`)。组件可以根据需要“选择”并使用这些策略，而无需关心其内部实现细节。这使得逻辑本身可以独立于使用它的UI组件。
- **组合模式：** React 的核心思想之一就是“组合优于继承”。自定义 Hook 允许你将组件逻辑提取到可重用的函数中，然后通过简单地调用这些函数，将不同的逻辑“组合”到你的组件中。你可以将多个自定义 Hook 组合在一个组件里，也可以在一个自定义 Hook 内部调用另一个自定义 Hook，形成复杂的逻辑组合树。

问题 9：什么是外观模式（Facade Pattern）？jQuery 在其设计中是如何体现外观模式的？

答案：

外观模式是一种结构型设计模式，它为一组复杂的子系统提供一个简化的、统一的接口。它隐藏了系统的复杂性，并向客户端提供一个可以轻松访问的接口。

jQuery 的体现：

jQuery 是外观模式的完美典范。它极大地简化了复杂的原生 DOM API、事件处理和 Ajax 请求。

- **DOM 操作：** 原生 JavaScript 获取元素并修改内容可能需要 `document.getElementById('myId').innerHTML = 'hello'`，而 jQuery 提供了简洁的 `$('#myId').html('hello')`。`$` 函数本身就是一个外观，它背后封装了元素选择、解析等一系列复杂操作。
- **Ajax 请求：** 原生 `XMLHttpRequest` 的使用非常繁琐，需要处理多个步骤和状态。jQuery 的 `$.ajax()`、`$.get()`、`$.post()` 方法提供了一个极其简单的接口，隐藏了底层的实现细节，开发者只需传入配置对象即可完成一个异步请求。

通过提供这些简洁的 API，jQuery 充当了一个“外观”，让开发者可以轻松地与浏览器底层复杂的子系统进行交互。

问题 10：请解释命令模式（Command Pattern），并说明它在前端撤销/重做（Undo/Redo）功能中的应用。

答案：

命令模式是一种行为设计模式，它将一个请求或操作封装成一个独立的对象（即“命令”对象）。这个对象包含了关于请求的所有信息，例如请求的接收者、要执行的动作以及动作的参数。

在撤销/重做功能中的应用：

命令模式是实现撤销/重做功能的理想选择。其工作流程如下：

1. **封装操作：** 将用户的每一个操作（如画图应用中的画一条线、文本编辑器中的输入一个字符、改变颜色等）都封装成一个具体的命令对象。这个命令对象至少包含两个方法：`execute()`（执行操作）和 `undo()`（撤销操作）。

2. **命令历史记录：** 创建一个历史记录栈（`historyStack`）。当用户执行一个操作时，就创建一个对应的命令对象，调用其 `execute()` 方法，并将这个命令对象压入历史记录栈中。
3. **实现撤销：** 当用户点击“撤销”按钮时，从历史记录栈中弹出一个命令对象，并调用该对象的 `undo()` 方法。为了支持“重做”，可以将这个被撤销的命令对象存入另一个“重做栈”（`redoStack`）。
4. **实现重做：** 当用户点击“重做”按钮时，从“重做栈”中弹出一个命令对象，再次调用其 `execute()` 方法，并将其重新压入历史记录栈。