手写JavaScript核心内置方法与Promise实现

你需要手写 `new`、`call`、`apply`、`bind` 操作符/函数以及简易版Promise，下面我将逐一提供完整可运行的实现代码，并附带核心原理说明。

一、手写new操作符

核心原理

1. 创建一个**新的空对象**，该对象的原型会指向构造函数的 `prototype` 属性
2. 将构造函数的 `this` 绑定到这个新创建的对象上
3. 执行构造函数的代码（为新对象添加属性和方法）
4. 判断构造函数的返回值：若返回值是引用类型（对象/函数等），则返回该引用类型；否则返回新创建的对象

实现代码

Code block

```
function myNew(constructor, ...args) {
    // 1. 创建空对象，且让该对象的__proto__指向构造函数的prototype
    const newObj = Object.create(constructor.prototype);
    // 2. 绑定this，执行构造函数
    const result = constructor.apply(newObj, args);
    // 3. 判断返回值，返回对应结果
    return (typeof result === 'object' && result !== null) || typeof result === 'function'
        ? result
        : newObj;
}

// 测试示例
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.sayHi = function() {
        console.log(`我是${this.name}，今年${this.age}岁`);
    }
}
// 使用手写myNew
const p1 = myNew(Person, '张三', 20);
```

```
p1.sayHi(); // 输出：我是张三，今年20岁
console.log(p1 instanceof Person); // true，验证原型链
```

二、手写call函数

核心原理

1. `call` 的作用是**改变函数的this指向**，并立即执行该函数
2. 为目标上下文（要绑定的this）添加一个临时方法，该方法指向当前函数
3. 执行这个临时方法，并传入后续参数
4. 删除临时方法，避免污染目标上下文
5. 返回函数执行的结果

实现代码

Code block

```
Function.prototype.myCall = function(context, ...args) {
    // 1. 处理默认this：若context为null/undefined，默认绑定全局对象（浏览器：window，Node：globalThis）
    const ctx = context || globalThis;
    // 2. 创建唯一临时属性，避免覆盖ctx原有属性
    const tempFnKey = Symbol('tempFn');
    // 3. 将当前函数（this，因为myCall是函数原型方法，this指向调用者函数）挂载到ctx上
    ctx[tempFnKey] = this;
    // 4. 执行临时方法，传入参数
    const result = ctx[tempFnKey](...args);
    // 5. 删除临时方法，清理环境
    delete ctx[tempFnKey];
    // 6. 返回函数执行结果
    return result;
};

// 测试示例
const obj = { name: '李四' };
function fn(a, b) {
    console.log(`this.name: ${this.name}, 参数: ${a} + ${b} = ${a + b}`);
    return a + b;
}
// 使用手写myCall
const res1 = fn.myCall(obj, 10, 20);
// 输出：this.name: 李四，参数：10 + 20 = 30
console.log(res1); // 30
```

三、手写apply函数

核心原理

1. `apply` 与 `call` 功能一致，唯一区别是**参数传递形式**：`apply` 的后续参数是一个数组（或类数组）
2. 其余逻辑与 `call` 一致：绑定this、临时挂载函数、执行并清理、返回结果

实现代码

Code block

```
Function.prototype.myApply = function(context, argsArr) {
    // 1. 处理默认this
    const ctx = context || globalThis;
    // 2. 处理参数：若未传入argsArr，默认是空数组
    const args = Array.isArray(argsArr) ? argsArr : [];
    // 3. 创建唯一临时属性
    const tempFnKey = Symbol('tempFn');
    // 4. 挂载当前函数到ctx
    ctx[tempFnKey] = this;
    // 5. 执行临时方法，展开数组参数
    const result = ctx[tempFnKey](...args);
    // 6. 删除临时方法
    delete ctx[tempFnKey];
    // 7. 返回执行结果
    return result;
};

// 测试示例
const obj2 = { name: '王五' };
function fn2(a, b) {
    console.log(`this.name: ${this.name}, 参数: ${a} * ${b} = ${a * b}`);
    return a * b;
}
// 使用手写myApply
const res2 = fn2.myApply(obj2, [10, 20]);
// 输出：this.name: 王五, 参数: 10 * 20 = 200
console.log(res2); // 200
```

四、手写bind函数

核心原理

1. `bind` 的作用是**改变函数的this指向**，但**不会立即执行函数**，而是返回一个绑定了this的新函数

2. 支持**柯里化参数**：绑定阶段可以传入部分参数，调用新函数时传入剩余参数
3. 新函数可以被当作构造函数使用（通过new调用），此时this指向会失效，指向新创建的实例
4. 新函数的原型要指向原函数的原型，保证原型链继承

实现代码

Code block

```
Function.prototype.myBind = function(context, ...bindArgs) {
    const _this = this; // 保存原函数的引用
    // 1. 返回一个新的绑定函数
    const boundFn = function(...callArgs) {
        // 2. 判断是否通过new调用：若通过new调用，this指向实例；否则指向传入的context
        const isNewCall = this instanceof boundFn;
        const ctx = isNewCall ? this : context || globalThis;
        // 3. 合并绑定阶段参数和调用阶段参数，执行原函数并返回结果
        return _this.apply(ctx, [...bindArgs, ...callArgs]);
    };
    // 4. 让新函数的原型指向原函数的原型，保证原型链继承
    boundFn.prototype = Object.create(_this.prototype);
    // 5. 修正构造函数指向
    boundFn.prototype.constructor = _this;
    return boundFn;
};

// 测试示例
const obj3 = { name: '赵六' };
function fn3(a, b, c) {
    console.log(`this.name: ${this.name}, 参数: ${a} + ${b} + ${c} = ${a + b + c}`);
    return a + b + c;
}
// 1. 绑定this和部分参数
const boundFn = fn3.myBind(obj3, 10);
// 2. 调用新函数，传入剩余参数（柯里化）
const res3 = boundFn(20, 30);
// 输出：this.name: 赵六, 参数: 10 + 20 + 30 = 60
console.log(res3); // 60

// 3. 测试new调用绑定函数
function Person2(name) {
    this.name = name;
}
const boundPerson = Person2.myBind({ name: '默认名' });
const p2 = new boundPerson('孙七');
console.log(p2.name); // 孙七（验证new调用时this失效，指向实例）
```

```
38  console.log(p2 instanceof Person2); // true（验证原型链继承）
```

五、手写简易版Promise

核心原理

1. Promise有三种**状态**：`pending`（等待中）、`fulfilled`（成功）、`rejected`（失败），状态一旦改变就不可逆转
2. 接收一个**执行器函数（executor）**，该函数立即执行，传入`resolve`（成功回调）和`reject`（失败回调）两个参数
3. 拥有`then`方法：接收`onFulfilled`（成功回调）和`onRejected`（失败回调）两个参数，支持链式调用
4. 异步执行：`then`方法的回调会在执行器函数异步操作完成后执行
5. 值的穿透：若`then`方法未传入合法回调，会将结果透传给下一个`then`

实现代码

Code block

```
1   // 定义Promise状态常量
2   const PENDING = 'pending';
3   const FULFILLED = 'fulfilled';
4   const REJECTED = 'rejected';
5
6   class MyPromise {
7       constructor(executor) {
8           try {
9               // 1. 初始化状态、成功值、失败原因
10              this.status = PENDING;
11              this.value = undefined; // 成功时的返回值
12              this.reason = undefined; // 失败时的原因
13              // 2. 存储异步回调（当then方法在状态改变前调用时，缓存回调函数）
14              this.onFulfilledCallbacks = [];
15              this.onRejectedCallbacks = [];
16
17              // 3. 定义resolve函数：将状态改为成功，保存成功值，执行缓存的成功回调
18              const resolve = (value) => {
19                  // 状态不可逆转：只有pending状态才能改变
20                  if (this.status === PENDING) {
21                      this.status = FULFILLED;
22                      this.value = value;
23                      // 批量执行缓存的成功回调
24                      this.onFulfilledCallbacks.forEach(cb => cb());
```

```
                }
            };

            // 4. 定义reject函数：将状态改为失败，保存失败原因，执行缓存的失败回调
            const reject = (reason) => {
                if (this.status === PENDING) {
                    this.status = REJECTED;
                    this.reason = reason;
                    // 批量执行缓存的失败回调
                    this.onRejectedCallbacks.forEach(cb => cb());
                }
            };

            // 5. 立即执行执行器函数
            executor(resolve, reject);
        } catch (error) {
            // 6. 执行器函数报错时，直接调用reject
            reject(error);
        }
    }

    // 实现then方法
    then(onFulfilled, onRejected) {
        // 处理值的穿透：默认回调函数透传结果/原因
        onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : value => value;
        onRejected = typeof onRejected === 'function' ? onRejected : reason => { throw reason; };

        // 7. 返回新的Promise，支持链式调用
        return new MyPromise((resolve, reject) => {
            // 处理成功状态
            if (this.status === FULFILLED) {
                // 异步执行回调（模拟原生Promise的微任务执行）
                setTimeout(() => {
                    try {
                        const result = onFulfilled(this.value);
                        // 处理回调返回值：若返回MyPromise实例，等待其状态改变；否则直接resolve
                        result instanceof MyPromise ? result.then(resolve, reject) : resolve(result);
                    } catch (error) {
                        reject(error);
                    }
                }, 0);
            }

```

```
68              // 处理失败状态
69              if (this.status === REJECTED) {
70                  setTimeout(() => {
71                      try {
72                          const result = onRejected(this.reason);
73                          result instanceof MyPromise ? result.then(resolve,
    reject) : resolve(result);
74                      } catch (error) {
75                          reject(error);
76                      }
77                  }, 0);
78              }
79
80              // 处理等待状态（异步操作未完成时，缓存回调函数）
81              if (this.status === PENDING) {
82                  this.onFulfilledCallbacks.push(() => {
83                      setTimeout(() => {
```

总结

1. `new`：创建新对象、绑定this、执行构造函数、判断返回值
2. `call` / `apply`：临时挂载函数、绑定this、立即执行、清理环境，区别仅在参数形式
3. `bind`：返回绑定this的新函数、支持柯里化、兼容new调用、维护原型链

1. 简易Promise：维护三种不可逆转状态、执行器立即执行、`then`方法支持异步和链式调用、实现值的穿透

六、手写防抖函数（debounce）

核心原理

1. 防抖的核心是“**延迟执行**”：当函数被频繁触发时，只在最后一次触发后，等待指定时间再执行函数
2. 每次触发函数时，都清除之前设置的定时器，重新计时，避免函数多次执行
3. 支持“立即执行”可选参数：开启后，第一次触发时立即执行函数，后续触发则重新延迟
4. 支持“取消防抖”功能：可手动取消尚未执行的延迟函数

实现代码

Code block

```
1   function debounce(fn, delay, immediate = false) {
2       let timer = null; // 存储定时器ID
3       let isExecuted = false; // 标记是否已立即执行（避免重复立即执行）
4
```

```
    // 防抖核心函数
    const debounced = function(...args) {
        const _this = this; // 保存原函数的this指向

        // 每次触发都清除之前的定时器，重新计时
        if (timer) clearTimeout(timer);

        // 处理立即执行逻辑
        if (immediate && !isExecuted) {
            fn.apply(_this, args);
            isExecuted = true; // 执行后标记为已执行
        } else {
            // 延迟执行逻辑
            timer = setTimeout(() => {
                fn.apply(_this, args);
                isExecuted = false; // 延迟执行后重置标记，允许下次立即执行
                timer = null; // 执行完毕清空定时器
            }, delay);
        }
    };

    // 取消防抖：手动清除定时器，重置状态
    debounced.cancel = function() {
        if (timer) clearTimeout(timer);
        timer = null;
        isExecuted = false;
    };

    return debounced;
}

// 测试示例
function handleInput(value) {
    console.log(`输入内容：${value}`);
}

// 1. 普通防抖（延迟500ms执行）
const debouncedInput = debounce(handleInput, 500);
// 模拟频繁输入（仅最后一次输入后500ms执行）
debouncedInput('a');
debouncedInput('ab');
debouncedInput('abc'); // 500ms后输出：输入内容：abc

// 2. 立即执行防抖（第一次输入立即执行，后续输入延迟）
const immediateDebounce = debounce(handleInput, 500, true);
immediateDebounce('1'); // 立即输出：输入内容：1
immediateDebounce('12'); // 500ms内再次触发，清除定时器
```

```
immediateDebounce('123'); // 500ms后输出：输入内容：123

// 3. 取消防抖
const cancelableDebounce = debounce(handleInput, 1000);
cancelableDebounce('需要取消');
cancelableDebounce.cancel(); // 手动取消，函数不会执行

```

七、手写节流函数（throttle）

核心原理

1. 节流的核心是“**控制执行频率**”：当函数被频繁触发时，保证在指定时间内只执行一次函数
2. 两种实现思路：
   - 时间戳法：第一次触发立即执行，后续触发时判断当前时间与上次执行时间的差值，大于等于延迟则执行
   - 定时器法：第一次触发延迟执行，后续触发时若定时器存在则不执行，执行后清空定时器
3. 合并两种思路，实现“第一次立即执行+后续固定频率执行”的完整节流效果
4. 支持“取消节流”功能：手动取消尚未执行的延迟函数

实现代码

Code block

```
function throttle(fn, interval) {
    let lastTime = 0; // 存储上次执行的时间戳
    let timer = null; // 存储定时器ID

    // 节流核心函数
    const throttled = function(...args) {
        const _this = this;
        const now = Date.now(); // 当前时间戳

        // 1. 时间戳法：判断当前时间与上次执行时间的差值是否大于等于间隔
        if (now - lastTime >= interval) {
            // 清除可能存在的定时器（避免时间戳和定时器双重执行）
            if (timer) {
                clearTimeout(timer);
                timer = null;
            }
            // 执行函数，更新上次执行时间
            fn.apply(_this, args);
            lastTime = now;
        } else if (!timer) {
```

```
            // 2. 定时器法：若时间差不足，且无定时器，则设置延迟执行
            timer = setTimeout(() => {
                fn.apply(_this, args);
                lastTime = Date.now(); // 执行后更新上次时间
                timer = null; // 清空定时器
            }, interval - (now - lastTime)); // 计算剩余延迟时间
        }
    };

    // 取消节流：手动清除定时器
    throttled.cancel = function() {
        if (timer) clearTimeout(timer);
        timer = null;
        lastTime = 0;
    };

    return throttled;
}

// 测试示例
function handleScroll() {
    console.log(`滚动触发时间：${new Date().toLocaleTimeString()}`);
}

// 节流函数（间隔1000ms执行一次）
const throttledScroll = throttle(handleScroll, 1000);

// 模拟频繁滚动（1秒内多次触发，仅1秒执行一次）
window.addEventListener('scroll', throttledScroll);

// 取消节流（不需要时移除）
// window.removeEventListener('scroll', throttledScroll);
// throttledScroll.cancel();

```

八、手写Vue响应式原理的代码实现（简易版）

核心原理（Vue2）

1. 核心是“**数据劫持+依赖收集**”：通过Object.defineProperty()劫持数据的getter和setter方法
2. getter：当数据被访问时，收集依赖（记录使用该数据的组件/函数）
3. setter：当数据被修改时，触发依赖更新（通知所有使用该数据的组件/函数重新渲染）
4. Dep类：管理依赖，提供添加依赖（addSub）和触发更新（notify）的方法
5. Watcher类：观察者，关联数据和更新函数，当数据变化时执行更新函数

6. Observer类：递归劫持对象的所有属性（包括子属性），使其变为响应式数据

实现代码

Code block

```
// 1. 依赖管理类：收集依赖、触发更新
class Dep {
    constructor() {
        this.subs = []; // 存储所有依赖（Watcher实例）
    }

    // 添加依赖
    addSub(sub) {
        // 确保依赖是Watcher实例，且不重复添加
        if (sub && sub.update && !this.subs.includes(sub)) {
            this.subs.push(sub);
        }
    }

    // 触发所有依赖更新
    notify() {
        this.subs.forEach(sub => {
            sub.update(); // 调用Watcher的更新方法
        });
    }
}

// 2. 观察者类：关联数据和更新函数
class Watcher {
    /**
     * @param {Object} vm - 组件实例（此处简化为响应式数据对象）
     * @param {string} exp - 数据属性名（如：'name'、'user.age'）
     * @param {Function} cb - 数据变化时的更新函数
     */
    constructor(vm, exp, cb) {
        this.vm = vm;
        this.exp = exp;
        this.cb = cb;
        this.value = this.get(); // 初始化时获取数据值，触发getter收集依赖
    }

    // 获取数据值，触发getter以收集依赖
    get() {
        Dep.target = this; // 标记当前Watcher为待收集的依赖
        const value = this.getVal(this.vm, this.exp); // 访问数据，触发getter
        Dep.target = null; // 收集完成后清空标记
```

```
        return value;
    }

    // 解析属性路径，获取数据值（支持嵌套属性，如'user.age'）
    getVal(vm, exp) {
        return exp.split('.').reduce((data, key) => {
            return data[key];
        }, vm);
    }

    // 数据变化时执行的更新方法
    update() {
        const newValue = this.getVal(this.vm, this.exp);
        const oldValue = this.value;
        if (newValue !== oldValue) {
            this.value = newValue;
            this.cb(newValue, oldValue); // 执行更新回调（如重新渲染视图）
        }
    }
}

// 3. 响应式处理类：递归劫持对象属性
class Observer {
    constructor(data) {
        this.observe(data); // 开始劫持数据
    }

    // 劫持数据主方法
    observe(data) {
        // 只处理对象/数组（基本类型无需劫持）
        if (!data || typeof data !== 'object') return;

        // 遍历对象所有属性，进行劫持
        Object.keys(data).forEach(key => {
            this.defineReactive(data, key, data[key]);
        });
    }

    // 定义响应式属性（核心：Object.defineProperty）
    defineReactive(obj, key, value) {
        const dep = new Dep(); // 每个属性对应一个Dep实例
        this.observe(value); // 递归劫持子属性（支持嵌套对象）

        Object.defineProperty(obj, key, {
            enumerable: true, // 可枚举
            configurable: false, // 不可重新配置
            // 访问属性时触发：收集依赖
```

九、手写浅拷贝函数的代码实现

核心原理

1. 浅拷贝：只复制对象的“表层属性”，若属性值是引用类型（对象/数组等），则复制的是引用地址，原对象和拷贝对象会共享该引用类型数据
2. 支持类型：对象（普通对象、数组、类数组）、基本类型（直接返回原值）
3. 实现思路：遍历原对象的可枚举属性，将属性值直接赋值到新对象中

实现代码

Code block

```
function shallowClone(target) {
    // 1. 处理基本类型和null（基本类型直接返回原值，null无需拷贝）
    if (target === null || typeof target !== 'object') {
        return target;
    }

    // 2. 处理数组：创建新数组，遍历赋值
    if (Array.isArray(target)) {
        return [...target]; // 或 Array.from(target)、target.slice()
    }

    // 3. 处理类数组（如arguments、DOM集合）：转为数组后返回
    if (target.length !== undefined && typeof target.length === 'number') {
        return Array.from(target);
    }

    // 4. 处理普通对象：创建新对象，遍历可枚举属性赋值
    const cloneObj = {};
    for (let key in target) {
        // 只复制自身的可枚举属性（不复制原型链上的属性）
        if (target.hasOwnProperty(key)) {
            cloneObj[key] = target[key];
        }
    }
    return cloneObj;
}

// 测试示例
// 1. 基本类型测试
console.log(shallowClone(123)); // 123
console.log(shallowClone('abc')); // 'abc'
console.log(shallowClone(null)); // null

```

```
// 2. 数组测试（浅拷贝，引用类型共享）
const arr1 = [1, 2, { a: 3 }];
const arr2 = shallowClone(arr1);
arr2[0] = 100; // 修改基本类型属性，不影响原数组
arr2[2].a = 300; // 修改引用类型属性，影响原数组
console.log(arr1); // [1, 2, { a: 300 }]
console.log(arr2); // [100, 2, { a: 300 }]

// 3. 普通对象测试
const obj1 = { name: '浅拷贝', user: { age: 20 } };
const obj2 = shallowClone(obj1);
obj2.name = '修改后'; // 基本类型不共享
obj2.user.age = 21; // 引用类型共享
console.log(obj1); // { name: '浅拷贝', user: { age: 21 } }
console.log(obj2); // { name: '修改后', user: { age: 21 } }

// 4. 类数组测试
function testArgs() {
    const argsClone = shallowClone(arguments);
    console.log(argsClone); // [1, 2, 3]（数组类型）
}
testArgs(1, 2, 3);

```

十、手写深拷贝函数的代码实现

核心原理

1. 深拷贝：复制对象的所有层级属性，包括引用类型属性，原对象和拷贝对象完全独立，互不影响
2. 解决问题：浅拷贝的引用共享问题、循环引用问题（如obj.a = obj）、特殊类型处理（Date、RegExp、Function等）
3. 实现思路：递归遍历原对象，遇到引用类型时，创建新的对应类型对象，继续递归拷贝其属性；遇到基本类型直接赋值

实现代码

Code block

```
function deepClone(target, map = new WeakMap()) {
    // 1. 处理基本类型和null（直接返回原值）
    if (target === null || typeof target !== 'object') {
        return target;
    }

    // 2. 处理循环引用：若目标已被拷贝过，直接返回拷贝后的对象
```

```
    if (map.has(target)) {
        return map.get(target);
    }

    let cloneObj;

    // 3. 处理特殊引用类型
    // 3.1 处理Date
    if (target instanceof Date) {
        cloneObj = new Date(target);
        map.set(target, cloneObj);
        return cloneObj;
    }

    // 3.2 处理RegExp
    if (target instanceof RegExp) {
        cloneObj = new RegExp(target.source, target.flags);
        map.set(target, cloneObj);
        return cloneObj;
    }

    // 3.3 处理数组
    if (target instanceof Array) {
        cloneObj = [];
        map.set(target, cloneObj); // 先缓存，避免循环引用
        // 递归拷贝数组元素
        target.forEach((item, index) => {
            cloneObj[index] = deepClone(item, map);
        });
        return cloneObj;
    }

    // 3.4 处理普通对象（包括自定义类实例）
    if (target instanceof Object) {
        cloneObj = {};
        map.set(target, cloneObj); // 缓存，避免循环引用
        // 遍历对象自身的可枚举属性（包括Symbol属性）
        const keys = [...Object.keys(target),
...Object.getOwnPropertySymbols(target)];
        keys.forEach(key => {
            // 递归拷贝属性值
            cloneObj[key] = deepClone(target[key], map);
        });
        return cloneObj;
    }

    // 其他类型（如Function，直接返回原函数，一般不拷贝函数）
```

```
    return target;
}

// 测试示例
// 1. 基本深拷贝测试（嵌套对象）
const obj1 = {
    name: '深拷贝',
    user: { age: 20 },
    hobbies: ['篮球', '游戏'],
    birth: new Date('2000-01-01'),
    reg: /test/g
};
const obj2 = deepClone(obj1);
obj2.user.age = 21;
obj2.hobbies.push('编程');
obj2.birth.setFullYear(2001);
console.log(obj1.user.age); // 20（不影响原对象）
console.log(obj1.hobbies); // ['篮球', '游戏']（不影响）
console.log(obj1.birth.getFullYear()); // 2000（不影响）
console.log(obj2.reg.flags); // 'g'（RegExp拷贝成功）

// 2. 循环引用测试（obj.a = obj）
const obj3 = { a: 1 };
obj3.b = obj3; // 循环引用：obj3.b指向自身
const obj4 = deepClone(obj3);
console.log(obj4.a); // 1
console.log(obj4.b === obj4); // true（拷贝后仍保持循环引用，且不报错）

// 3. Symbol属性测试
const sym = Symbol('test');
const obj5 = { [sym]: 'symbol值' };
const obj6 = deepClone(obj5);
console.log(obj6[sym]); // 'symbol值'（Symbol属性拷贝成功）

```

补充总结（新增5道面试题）

1. 防抖（debounce）：延迟执行，频繁触发时只执行最后一次，核心是清除定时器重新计时，支持立即执行和取消
2. 节流（throttle）：控制执行频率，指定时间内只执行一次，结合时间戳和定时器实现完整效果，支持取消
3. Vue响应式（Vue2）：核心是数据劫持（Object.defineProperty）+依赖收集（Dep）+观察者（Watcher），递归劫持对象属性，数据变化时触发视图更新
4. 浅拷贝：只复制表层属性，引用类型共享地址，实现简单（遍历赋值、扩展运算符等）

5. 深拷贝：复制所有层级属性，完全独立，需解决循环引用和特殊类型（Date、RegExp），核心是递归+缓存

十一、手写事件总线（EventBus）的代码实现

核心原理

1. 事件总线是一种**发布-订阅模式**的实现，用于组件/模块间的解耦通信，核心是维护一个事件注册表
2. 事件注册表：以事件名为键，对应的值是一个数组，存储该事件的所有订阅回调函数
3. 核心方法：
   - on：订阅事件，向注册表中添加回调函数
   - emit：发布事件，触发对应事件的所有回调函数并传递参数
   - off：取消订阅，从注册表中移除指定回调函数（若未指定则移除该事件所有回调）
   - once：订阅一次性事件，回调函数执行一次后自动取消订阅
4. 需要处理边界情况：重复订阅同一回调、取消不存在的事件/回调、触发未订阅的事件

实现代码

Code block

```
class EventBus {
    constructor() {
        // 初始化事件注册表：key-事件名，value-回调函数数组
        this.events = {};
    }

    /**
     * 订阅事件
     * @param {string} eventName - 事件名
     * @param {Function} callback - 订阅回调
     */
    on(eventName, callback) {
        // 校验参数：事件名必须是字符串，回调必须是函数
        if (typeof eventName !== 'string' || typeof callback !== 'function') {
            throw new Error('事件名必须为字符串，回调必须为函数');
        }
        // 若事件不存在，初始化回调数组
        if (!this.events[eventName]) {
            this.events[eventName] = [];
        }
        // 避免重复添加同一回调
        if (!this.events[eventName].includes(callback)) {
            this.events[eventName].push(callback);
        }
    }
```

```
26
27      /**
28       * 发布事件
29       * @param {string} eventName - 事件名
30       * @param  {...any} args - 传递给回调的参数
31       */
32      emit(eventName, ...args) {
33          if (typeof eventName !== 'string') {
34              throw new Error('事件名必须为字符串');
35          }
36          // 若事件不存在，直接返回
37          if (!this.events[eventName]) return;
38          // 触发所有回调，绑定this为EventBus实例（可选，便于回调内调用其他方法）
39          this.events[eventName].forEach(callback => {
40              callback.apply(this, args);
41          });
42      }
43
44      /**
45       * 取消订阅
46       * @param {string} eventName - 事件名
47       * @param {Function} [callback] - 可选，要取消的回调；未传递则移除该事件所有回调
48       */
49      off(eventName, callback) {
50          if (typeof eventName !== 'string') {
51              throw new Error('事件名必须为字符串');
52          }
53          // 事件不存在，直接返回
54          if (!this.events[eventName]) return;
55          // 未指定回调：移除该事件所有订阅
56          if (!callback) {
57              this.events[eventName] = [];
58              return;
59          }
60          // 指定回调：只移除该回调（过滤掉目标回调）
61          if (typeof callback === 'function') {
62              this.events[eventName] = this.events[eventName].filter(cb => cb
    !== callback);
63          }
64      }
65
66      /**
67       * 订阅一次性事件（执行一次后自动取消）
68       * @param {string} eventName - 事件名
69       * @param {Function} callback - 订阅回调
70       */
71      once(eventName, callback) {
```

```
        // 包装回调：执行原回调后，自动取消当前订阅
        const onceCallback = (...args) => {
            callback.apply(this, args);
            this.off(eventName, onceCallback);
        };
        // 订阅包装后的回调
        this.on(eventName, onceCallback);
    }
}

// 测试示例
const eventBus = new EventBus();

// 1. 普通订阅与发布
function handleTest(arg1, arg2) {
    console.log('普通事件触发：', arg1, arg2);
```

十二、手写函数柯里化（curry）的代码实现

核心原理

1. 柯里化是将**接收多个参数的函数**，转化为**接收单一参数（或部分参数）的函数序列**的过程，最终返回接收剩余参数并执行原函数的函数
2. 核心逻辑：保存已接收的参数，判断已接收参数数量是否达到原函数的参数长度（形参长度），若达到则执行原函数，否则返回一个新函数继续接收参数
3. 支持特性：
   - 自动判断参数是否足够，足够则执行
   - 支持分步传递参数，也支持一次性传递所有参数
   - 兼容原函数的this指向
4. 依赖原函数的 `length` 属性获取形参数量，若原函数是剩余参数（...rest）形式，需手动指定参数长度

实现代码

Code block

```
/**
 * 函数柯里化
 * @param {Function} fn - 待柯里化的原函数
 * @param {number} [arity=fn.length] - 原函数的参数长度（若原函数用rest参数，需手动指定）
 * @returns {Function} 柯里化后的函数
 */
function curry(fn, arity = fn.length) {
```

```
    // 校验：fn必须是函数
    if (typeof fn !== 'function') {
        throw new Error('待柯里化的必须是函数');
    }

    // 递归生成柯里化函数，接收已传递的参数（闭包保存）
    const curried = function(...args) {
        const _this = this; // 保存当前this指向（用于原函数执行时绑定）
        // 情况1：已传递的参数数量 >= 原函数参数长度，执行原函数
        if (args.length >= arity) {
            return fn.apply(_this, args);
        }
        // 情况2：参数不足，返回新函数继续接收参数
        return function(...nextArgs) {
            // 合并已接收的参数和新传递的参数，递归调用curried
            return curried.apply(_this, [...args, ...nextArgs]);
        };
    };

    return curried;
}

// 测试示例
// 1. 普通函数柯里化
function add(a, b, c) {
    return a + b + c;
}
const curriedAdd = curry(add);

// 分步传递参数
console.log(curriedAdd(1)(2)(3)); // 6
console.log(curriedAdd(1, 2)(3)); // 6
console.log(curriedAdd(1)(2, 3)); // 6

// 一次性传递所有参数
console.log(curriedAdd(1, 2, 3)); // 6

// 2. 绑定this的柯里化函数
const obj = {
    name: '柯里化',
    sayHello: function(greet, msg) {
        return `${greet}, ${this.name}: ${msg}`;
    }
};
const curriedSayHello = curry(obj.sayHello);
// 绑定this为obj，分步传递参数
```

```
console.log(curriedSayHello.call(obj, '你好')('欢迎学习')); // 输出：你好，柯里化：欢迎学习

// 3. 处理rest参数函数（需手动指定参数长度）
function sum(...args) {
    return args.reduce((total, curr) => total + curr, 0);
}
// 手动指定需要接收3个参数
const curriedSum = curry(sum, 3);
console.log(curriedSum(1)(2)(3)); // 6
console.log(curriedSum(1, 2)(3)); // 6
```

十三、手写Vue3响应式原理的代码实现（简易版）

核心原理（Vue3 vs Vue2）

1. Vue3 响应式核心从 Vue2 的 `Object.defineProperty` 改为**Proxy**，解决了 Vue2 的缺陷：
   - 支持数组索引/长度变化的响应式
   - 支持新增/删除对象属性的响应式
   - 无需递归劫持初始对象的所有属性（Proxy 是懒代理，访问属性时才触发劫持）
2. 核心模块：
   - Proxy：代理目标对象，拦截属性的读取（get）和设置（set）操作
   - Reflect：配合 Proxy 操作目标对象，确保操作的正确性和规范性
   - effect：副作用函数（如组件渲染函数），依赖收集时记录该函数，数据变化时重新执行
   - track：依赖收集，在 Proxy 的 get 拦截中调用，记录“目标对象-属性-副作用函数”的映射关系
   - trigger：触发更新，在 Proxy 的 set 拦截中调用，找到对应属性的所有副作用函数并执行
3. 依赖存储：用 WeakMap（目标对象 -> Map（属性名 -> Set（副作用函数））） 的结构，避免内存泄漏

实现代码

Code block

```
// 1. 存储依赖的结构：WeakMap<target, Map<key, Set<effect>>>
const targetMap = new WeakMap();

// 2. 当前活跃的副作用函数（依赖收集时需要记录）
let activeEffect = null;

/**
 * 副作用函数：执行时会访问响应式数据，触发依赖收集
 * @param {Function} fn - 要执行的副作用函数（如渲染函数）
```

```
 */
function effect(fn) {
    // 包装副作用函数，便于后续处理（如错误捕获）
    const effectFn = () => {
        // 执行副作用函数前，清除该函数在所有依赖中的记录（避免重复收集）
        cleanup(effectFn);
        activeEffect = effectFn; // 标记当前活跃的副作用函数
        fn(); // 执行副作用函数，触发Proxy的get拦截，进行依赖收集
        activeEffect = null; // 收集完成后清空标记
    };
    // 初始化执行一次副作用函数
    effectFn();
}

/**
 * 清除副作用函数的依赖记录
 * @param {Function} effectFn - 要清除的副作用函数
 */
function cleanup(effectFn) {
    // 遍历targetMap，找到所有包含当前effectFn的依赖集合并删除
    for (const [target, keyMap] of targetMap) {
        for (const [key, effects] of keyMap) {
            effects.delete(effectFn);
        }
    }
}

/**
 * 依赖收集：在响应式数据被访问时调用
 * @param {Object} target - 目标对象
 * @param {string|symbol} key - 被访问的属性名
 */
function track(target, key) {
    // 若没有活跃的副作用函数，无需收集依赖
    if (!activeEffect) return;

    // 1. 从targetMap中获取目标对象对应的keyMap（属性-副作用映射）
    let keyMap = targetMap.get(target);
    if (!keyMap) {
        targetMap.set(target, (keyMap = new Map()));
    }

    // 2. 从keyMap中获取属性对应的副作用集合
    let effects = keyMap.get(key);
    if (!effects) {
        keyMap.set(key, (effects = new Set()));
    }
```

```
57
58      // 3. 将当前活跃的副作用函数添加到集合中
59      effects.add(activeEffect);
60  }
61
62  /**
63   * 触发更新：在响应式数据被修改时调用
64   * @param {Object} target - 目标对象
65   * @param {string|symbol} key - 被修改的属性名
66   */
67  function trigger(target, key) {
68      // 1. 从targetMap中获取目标对象对应的keyMap
69      const keyMap = targetMap.get(target);
70      if (!keyMap) return;
71
72      // 2. 获取属性对应的副作用集合
73      const effects = keyMap.get(key);
74      if (!effects) return;
75
76      // 3. 执行所有副作用函数（创建新数组避免遍历中删除元素导致的问题）
77      [...effects].forEach(effectFn => effectFn());
78  }
79
80  /**
81   * 创建响应式对象（核心：Proxy代理）
82   * @param {Object} target - 要代理的目标对象
83   * @returns {Proxy} 响应式代理对象
84   */
85  function reactive(target) {
86      // 校验：只代理对象/数组（基本类型无需代理）
87      if (target === null || typeof target !== 'object') {
88          return target;
```

补充总结（新增3道面试题）

1. 事件总线（EventBus）：基于发布-订阅模式，核心是维护事件注册表，提供on/emit/off/once方法，实现组件解耦通信
2. 函数柯里化（curry）：将多参数函数转化为单参数函数序列，通过闭包保存已接收参数，参数足够时执行原函数，支持分步传参和this绑定
3. Vue3响应式：核心是Proxy代理+Reflect操作，配合effect/track/trigger实现依赖收集和更新触发，解决了Vue2的数组/新增属性响应式缺陷，依赖存储用WeakMap避免内存泄漏