

1. 手写防抖/节流

防抖(debounce): 触发高频事件后 n 秒内函数只会执行一次, 如果 n 秒内高频事件再次被触发, 则重新计算时间

节流(throttle): 高频事件触发, 但在 n 秒内只会执行一次, 所以节流会稀释函数的执行频率

防抖:

- 加入了防抖以后, 当你在频繁的输入时, 并不会发送请求, 只有当你在指定间隔内没有输入时, 才会执行函数。如果停止输入但是在指定间隔内又输入, 会重新触发计时。
- 函数防抖就是法师发技能的时候要读条, 技能读条没完再按技能就会重新读条。

节流:

- 间隔时间执行
- 规定在一个单位时间内, 只能触发一次函数。如果这个单位时间内触发多次函数, 只有一次生效。
- 函数节流就是fps游戏的射速, 就算一直按着鼠标射击, 也只会在规定射速内射出子弹。

应用场景:

- debounce
 - search搜索联想, 用户在不断输入值时, 用防抖来节约请求资源。
 - window触发resize的时候, 不断的调整浏览器窗口大小会不断的触发这个事件, 用防抖来让其只触发一次
- throttle
 - 鼠标不断点击触发, mousedown(单位时间内只触发一次)
 - 监听滚动事件, 比如是否滑到底部自动加载更多, 用throttle来判断

```

// 防抖函数
function debounce(fn, wait) {
  let timer;
  return function () {
    let _this = this;
    let args = arguments;
    if (timer) {
      clearTimeout(timer);
    }
    timer = setTimeout(function () {
      fn.apply(_this, args);
    }, wait);
  };
}
// 使用
window.onresize = debounce(function () {
  console.log("resize");
}, 500);

```

```

// 节流
// 方式1: 使用时间戳
function throttle1(fn, wait) {
  let time = 0;
  return function () {
    let _this = this;
    let args = arguments;
    let now = Date.now();
    if (now - time > wait) {
      fn.apply(_this, args);
      time = now;
    }
  };
}
// 方式2: 使用定时器
function throttle2(fn, wait) {
  let timer;
  return function () {
    let _this = this;
    let args = arguments;

    if (!timer) {
      timer = setTimeout(function () {
        timer = null;
        fn.apply(_this, args);
      }, wait);
    }
  };
}

```

2. 手写Promise

- 什么是Promise?
 - Promise是JS的异步编程的一种解决方案，在ES6将其写进了语言标准，提供了原生的Promise对象。
 - Promise简单来理解就是一个**容器**，里面存放着某个未来才会结束的事件结果。Promise是一个**对象**，从它可以获取异步操作的消息，Promise提供了统一的API，各种异步操作都可以用同样的方法进行处理。
- 优点/特点
 - 对象的状态不受外界影响。有三种状态：pending(进行中)、fulfilled(成功)、rejected(失败)。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。
 - 一旦状态改变，就不会再变，任何时候都可以得到这个结果。三个状态只有从pending到fulfilled或者从pending到rejected。状态只有从pending改变到fulfilled或者rejected，两种改变。
 - 有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。
 - Promise对象提供统一的接口，使得控制异步操作更加容易。
- 缺点
 - 无法取消Promise，一旦新建它就会立即执行，无法中途取消。
 - 如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。
 - 当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

思路：

1. 当 pending 时，thenable 函数由一个队列维护
2. 当状态变为 resolved(fulfilled) 时，队列中所有 thenable 函数执行
3. 当 resolved 时，thenable 函数直接执行

```

// 1、基本架构：
//  状态
//  then
//  执行器函数 executor

// 2、executor、resolve、reject
// 3、then 同步下调用
// 4、then 异步下调用
// 5、then 链式调用
//  返回 Promise
//  then 函数递归返回常量结果，供下个 then 使用
//  考虑 then 成功的回调为 null 的情况

const PROMISE_STATUS_PENDING = "pending";
const PROMISE_STATUS_FULFILLED = "fulfilled";
const PROMISE_STATUS_REJECTED = "rejected";

// help fun
function execFunctionWithCatchError(execFun, value, resolve, reject) {
  try {
    const result = execFun(value);
    resolve(result);
  } catch (error) {
    reject(error);
  }
}

class MyPromise {
  constructor(executor) {
    this.status = PROMISE_STATUS_PENDING; // 记录promise状态
    this.value = undefined; // resolve返回值
    this.reason = undefined; // reject返回值
    this.onFulfilledFns = []; // 存放成功回调
    this.onRejectedFns = []; // 存放失败回调

    const resolve = value => {
      // if (this.status === PROMISE_STATUS_PENDING) {
      //   queueMicrotask(() => {
      //     if (this.status !== PROMISE_STATUS_PENDING) return;
      //     this.status = PROMISE_STATUS_FULFILLED;
      //     this.value = value;
      //     this.onFulfilledFns.forEach(fn => {
      //       fn(this.value);
      //     });
      //   });
      // }
      if (value instanceof Promise) {
        return value.then(resolve, reject);
      }
    }
  }
}

```

```

        if (this.state === Promise.PENDING) {
            this.state = Promise.RESOLVED;
            this.value = value;
            this.onResolvedCallbacks.forEach((fn) => fn());
        }
    };

    const reject = reason => {
        if (this.status === PROMISE_STATUS_PENDING) {
            queueMicrotask(() => {
                if (this.status !== PROMISE_STATUS_PENDING) return;
                this.status = PROMISE_STATUS_REJECTED;
                this.reason = reason;
                this.onRejectedFns.forEach(fn => {
                    fn(this.reason);
                });
            });
        }
    };

    try {
        executor(resolve, reject);
    } catch (error) {
        reject(error);
    }
}

then(onFulfilled, onRejected) {
    onFulfilled =
        onFulfilled ||
        (value => {
            return value;
        });

    onRejected =
        onRejected ||
        (err => {
            throw err;
        });

    return new MyPromise((resolve, reject) => {
        // 1、 when operate then, status have confirmed
        if (this.status === PROMISE_STATUS_FULFILLED && onFulfilled) {
            execFunctionWithCatchError(onFulfilled, this.value, resolve, reject);
        }
        if (this.status === PROMISE_STATUS_REJECTED && onRejected) {
            execFunctionWithCatchError(onRejected, this.reason, resolve, reject);
        }

        if (this.status === PROMISE_STATUS_PENDING) {
            // this.onFulfilledFns.push(onFulfilled);

```

```

        if (onFulfilled) {
            this.onFulfilledFns.push(() => {
                execFunctionWithCatchError(onFulfilled, this.value, resolve, reject);
            });
        }

        // this.onRejectedFns.push(onRejected);
        if (onRejected) {
            this.onRejectedFns.push(() => {
                execFunctionWithCatchError(onRejected, this.reason, resolve, reject);
            });
        }
    });
}

catch(onRejected) {
    return this.then(undefined, onRejected);
}

finally(onFinally) {
    return this.then(onFinally, onFinally);
}

static resolve(value) {
    return new MyPromise(resolve => resolve(value));
}

static reject(reason) {
    return new MyPromise((resolve, reject) => reject(reason));
}

static all(promises) {
    return new MyPromise((resolve, reject) => {
        const values = [];
        promises.forEach(promise => {
            promise.then(
                res => {
                    values.push(res);
                    if (values.length === promises.length) {
                        resolve(values);
                    }
                },
                err => {
                    reject(err);
                }
            );
        });
    });
}

```

```

// 只关心是否都完成了 无论成功与否
static allSettled(promises) {
    return new MyPromise(resolve => {
        const results = [];
        promises.forEach(promise => {
            promise.then(
                res => {
                    results.push({ status: PROMISE_STATUS_FULFILLED, value: res });
                    if (results.length === promises.length) {
                        resolve(results);
                    }
                },
                err => {
                    results.push({ status: PROMISE_STATUS_REJECTED, value: err });
                    if (results.length === promises.length) {
                        resolve(results);
                    }
                }
            );
        });
    });
}

static race(promises) {
    return new MyPromise((resolve, reject) => {
        promises.forEach(promise => {
            promise.then(
                res => {
                    resolve(res);
                },
                err => {
                    reject(err);
                }
            );
        });
    });
}

static any(promises) {
    return new MyPromise((resolve, reject) => {
        const reasons = [];
        promises.forEach(promise => {
            promise.then(
                res => {
                    resolve(res);
                },
                err => {
                    reasons.push(err);
                    if (reasons.length === promise.length) {
                        // reject(new AggregateError(reasons));
                    }
                }
            );
        });
    });
}

```

```

        reject(reasons);
      }
    }
  );
});
});
}
}

const p1 = new MyPromise((resolve, reject) => {
  setTimeout(() => {
    console.log("--- 1 ---");
    resolve(111);
  });
}).then(res => {
  console.log("p1 res :>> ", res);
});

const p2 = new MyPromise((resolve, reject) => {
  console.log("--- 2 ---");
  resolve(222);
});

const p3 = new MyPromise((resolve, reject) => {
  console.log("--- 3 ---");
  resolve(333);
});

const p4 = new MyPromise((resolve, reject) => {
  console.log("--- 4 ---");
  reject(444);
});

MyPromise.all([p2, p3]).then(res => {
  console.log("p2&p3 res :>> ", res);
});

MyPromise.all([p2, p4])
  .then(res => {
    console.log("p2&p4 res :>> ", res);
  })
  .catch(err => {
    console.log("err :>> ", err);
  });

// --- 2 ---
// --- 3 ---
// --- 4 ---
// p2&p3 res :>> [ 222, 333 ]
// err :>> 444

```



```
// --- 1 ---  
// p1 res :>> 111
```

3. 手写bind / softbind

最简单的写法

```
Function.prototype.myBind = function (obj, ...args) {  
  return (...rest) => this.call(obj, ...args, ...rest);  
};
```

```
Function.prototype.myBind = function (obj, ...args) {  
  return (...rest) =>  
    this.apply(obj, args.concat(rest).slice(0, this.length));  
};
```

```
Function.prototype.bind = function (obj, ...args) {  
  const self = this;  
  const fn = function (...newArgs) {  
    self.apply(this instanceof fn ? this : obj, args.concat(newArgs));  
  };  
};
```

```
fn.prototype = Object.create(this.prototype);  
// 考虑new的情况  
  
return fn;  
};
```

```
Function.prototype.softBind = function (obj, ...args) {  
  const self = this;  
  const fn = function (...args2) {  
    return self.apply(this === global ? obj : this, args.concat(args2));  
  };  
  return fn;  
};
```

// bind函数多次调用会以第一次绑定的this为准，softbind以最后一次绑定传入的this为准；

```
Function.prototype.softBind = function (obj, ...args) {  
  const self = this  
  const fn = function (...newArgs) {  
    const o = !this || this === (window || global) ? obj : this  
    return self.apply(o, [...args, ...newArgs])  
  }  
  
  fn.prototype = Object.create(self.prototype)  
  return fn  
}
```

```
Function.prototype.myCall = function (ctx = globalThis) {  
  const args = Array.from(arguments).slice(1);  
  const key = Symbol("key");  
  ctx[key] = this;  
  const res = ctx[key](...args);  
  delete ctx[key]  
  return res  
};
```

```
Function.prototype.myApply = function (ctx = globalThis) {  
  const args = arguments[1];  
  const key = Symbol("key");  
  ctx[key] = this;  
  const res = ctx[key](...args);  
  delete ctx[key]  
  return res  
};
```

4.实现一个 promise.map，进行并发数控制，有以下测试用例

```
pMap([1, 2, 3, 4, 5], (x) => Promise.resolve(x + 1));  
  
pMap([Promise.resolve(1), Promise.resolve(2)], (x) => x + 1);  
  
// 注意输出时间控制  
pMap([1, 1, 1, 1, 1, 1, 1, 1], (x) => sleep(1000), { concurrency: 2 });
```

```
function pMap(arr, fn, concurrency = Number.MAX_SAFE_INTEGER) {
  return new Promise((resolve) => {
    let ret = [];
    let index = -1;
    function next() {
      ++index;
      Promise.resolve(arr[index])
        .then((val) => fn(val, index))
        .then((res) => {
          ret.push(res);
          if (ret.length === arr.length) {
            resolve(ret);
          } else if (index < arr.length) {
            next();
          }
        });
    }

    for (let i = 0; i < arr.length && i < concurrency; i++) {
      next();
    }
  });
}
```

5. lodash Get 正则表达式

```
function get(source, path, defaultValue = undefined) {
  // a[3].b -> a.3.b -> [a, 3, b]
  const paths =
    path.replace(/\[(\w+)\]/g, '.$1')
      .replace(/\["(\w+)"/g, '.$1')
      .replace(/\['(\w+)'\]/g, '.$1')
      .split('.')
  // 非标准**$1, $2, $3, $4, $5, $6, $7, $8, $9** 属性是包含括号子串匹配的正则表达式的静态和只
  let result = source
  for (const p of paths) {
    result = result?.[p]
  }
  return result === undefined || defaultValue ? defaultValue : result
}

const object = { a: [{ b: { c: 3 } }] };
const result = get(object, 'a[0].b.c',);
console.log(result);
```

6. Deep Clone

方法一：JSON.stringify() 和 JSON.parse()

- undefined、symbol、function类型直接被过滤掉了
- date类型被自动转成了字符串类型

```
var copyObj = JSON.parse(JSON.stringify(obj));
```

方法二：Lodash cloneDeep

方法三：手写实现深拷贝函数

```
/**
 * 深拷贝关注点：
 * 1. JavaScript内置对象的复制：Set、Map、Date、Regex等
 * 2. 循环引用问题
 * @param {*} object
 * @returns
 */
function deepCopy(object, map = new WeakMap()) {
  if (!object || typeof object !== "object") return object;
  // 内置对象的复制
  if (object === null) return object;
  if (object instanceof Date) return new Date(object);
  if (object instanceof RegExp) return new RegExp(object);

  // 解决循环引用问题
  if (map.has(object)) return map.get(object);

  let newObj = Array.isArray(object) ? [] : {};
  map.set(object, newObj);

  for (let key in object) {
    if (object.hasOwnProperty(key)) {
      newObj[key] =
        typeof object[key] === "object" ? deepCopy(object[key]) : object[key];
    }
  }

  return newObj;
}
```

7. 浅拷贝

- Object.assign()
- 扩展运算符 let cloneObj = { ...obj };
- 数组方法实现数组浅拷贝
- Array.prototype.slice
- Array.prototype.concat

- 手写浅拷贝

```
function shallowCopy(object) {
  // 只拷贝对象
  if (!object || typeof object !== "object") return;
  // 根据 object 的类型判断是新建一个数组还是对象
  let newObject = Array.isArray(object) ? [] : {};
  // 遍历 object, 并且判断是 object 的属性才拷贝
  for (let key in object) {
    if (object.hasOwnProperty(key)) {
      newObject[key] = object[key];
      // 区别在这里
    }
  }
  return newObject;
} // 浅拷贝的实现;
```

8. flatMap

- JavaScript中的map和flatMap都是数组方法，用于对数组中的每个元素进行操作。
- map方法可以将数组中的每个元素映射到一个新数组中，并返回该新数组。
- flatMap方法则是先将数组中的每个元素映射到一个新数组中，然后再将所有新数组展平成一个数组。

```
Array.prototype.FlatMap = function (callback, thisArgs) {
  return this.reduce((acc, value) => {
    return (acc = acc.concat(callback.call(thisArgs, value)));
  });
};

// 先map再flat
let myFlatMap = function (fn) {
  let target = this;
  return target.map((i) => fn(i)).flat();
};
```

9. 数组扁平化

递归实现

```
function flatMap(arr) {
  let list = [];
  arr.forEach(item => {
    if (Array.isArray(item)) {
      const l = flatMap(item);
      list.push(...l);
      // list = list.concat(l);
    } else {
      list.push(item);
    }
  })
  return list;
}
```

reduce 函数迭代

```
function flatten(arr) {
  return arr.reduce(function(prev, next){
    return prev.concat(Array.isArray(next) ? flatten(next) : next)
  }, [])
}
```

扩展运算符实现

```
function flatten(arr) {
  while (arr.some(item => Array.isArray(item))) {
    arr = [].concat(...arr);
  }
  return arr;
}
```

split 和 toString

```
function flatten(arr) {
  return arr.toString().split(',');
}
```

ES6 中的 flat

```
function flatten(arr) {
  return arr.flat(Infinity);
}
```

JSON和正则方法

```
function flatten(arr) {  
  let str = JSON.stringify(arr);  
  str = str.replace(/(\[|\])/g, '');  
  str = '[' + str + ']';  
  return JSON.parse(str);  
}
```

10. 如何实现一个无限累加的 sum 函数

```
sum(1, 2, 3).valueOf(); //6  
sum(2, 3)(2).valueOf(); //7  
sum(1)(2)(3)(4).valueOf(); //10  
sum(2)(4, 1)(2).valueOf(); //9  
sum(1)(2)(3)(4)(5)(6).valueOf(); // 21  
思路：
```

1. sum 返回一个函数，收集所有的累加项，使用递归实现
2. 返回函数带有 valueOf 属性，用于统一计算

```
function sum(...args) {  
  const f = (...rest) => sum(...args, ...rest);  
  f.valueOf = () => args.reduce((x, y) => x + y, 0);  
  return f;  
}
```

11. 解析 URL Params 为对象

```
function parseParam(url) {  
  const paramsStr = /.+\?(.+)$/ .exec(url)[1]; // 将 ? 后面的字符串取出来  
  const paramsArr = paramsStr.split('&'); // 将字符串以 & 分割后存到数组中  
  let paramsObj = {};  
  // 将 params 存到对象中  
  paramsArr.forEach(param => {  
    if (/=/ .test(param)) { // 处理有 value 的参数  
      let [key, val] = param.split('='); // 分割 key 和 value  
      val = decodeURIComponent(val); // 解码  
      val = /^\d+$/ .test(val) ? parseFloat(val) : val; // 判断是否转为数字  
      if (paramsObj.hasOwnProperty(key)) { // 如果对象有 key, 则添加一个值  
        paramsObj[key] = [].concat(paramsObj[key], val);  
      } else { // 如果对象没有这个 key, 创建 key 并设置值  
        paramsObj[key] = val;  
      }  
    } else { // 处理没有 value 的参数  
      paramsObj[param] = true;  
    }  
  })  
  return paramsObj;  
}
```


12. Promise.all

```
Promise.all = function (promises) {
  const len = promises.length;
  const result = new Array(len);
  let countDone = 0;
  return new Promise((resolve, reject) => {
    if (len === 0) {
      resolve(result);
    }
    for (let i = 0; i < len; i++) {
      const promise = promises[i];
      Promise.resolve(promise).then(
        (data) => {
          result[i] = data;
          countDone++;
          if (countDone === len) {
            resolve(result);
          }
        },
        (error) => {
          reject(error);
        }
      );
    }
  });
};
```

13 逆序字符串

```
const reverse = (s) => s.split('').reverse().join('');

function reverse(s) {
  let r = '';
  for (const c of s) {
    r = c + r;
  }
  return r;
}
```

14. 给数字添加千位符

```
function numberThousands(number, thousandsSeperator = ",") {  
    return String(number).replace(  
        /(\d)(?=(\d\d\d)+(!\d))/g,  
        "$1" + thousandsSeperator  
    );  
}
```

```
function numberThousands(number, thousandsSeperator = ",") {  
    const reverse = (str) => str.split("").reverse().join("");  
    const str = reverse(String(number)).replace(  
        /\d\d\d(?!d)/g,  
        "$1" + thousandsSeperator  
    );  
    return reverse(str);  
}
```

```
function numberThousands(number, thousandsSeperator = ",") {  
    const s = String(number);  
    let r = "";  
    for (let i = s.length - 1; i >= 0; i--) {  
        const seperator = (s.length - i - 1) % 3 ? "" : thousandsSeperator;  
        r = `${s[i]}${seperator}${r}`;  
    }  
    return r.slice(0, -1);  
}
```

14. Deep Equal

```
function isEqual(x, y) {
  if (x === y) {
    return true;
  } else if (
    typeof x === "object" &&
    x !== null &&
    typeof y === "object" &&
    y !== null
  ) {
    const keysX = Object.keys(x);
    const keysY = Object.keys(y);
    if (keysX.length !== keysY.length) {
      return false;
    }
    for (const key of keysX) {
      if (!isEqual(x[key], y[key])) {
        return false;
      }
    }
    return true;
  } else {
    return false;
  }
}
```

15. 在 JS 中如何监听 Object 某个属性值的变化

- Object.defineProperty()
- Proxy

```

let obj = {
  a: '元素a',
  b: '元素b'
}
const handle = {
  get: (obj, prop) => {
    console.log(`正在获取: ${prop}`);
    return obj[prop];
  },
  set: (obj, prop, value) => {
    console.log(`正在修改元素: 将${prop}属性设置为${value}`);
    obj[prop] = value;
  }
}

const proxy = new Proxy(obj, handle);
console.log(proxy.a)
//正在获取: a
// 元素a
proxy.a = '123'
// 正在修改元素: 将a属性设置为123
console.log(proxy);
//Proxy {a: "123", b: "元素b"}

```

16. 实现函数 promisify

```

function promisify(fn) {
  return function (...args) {
    let hasCb = args.some((v) => typeof v === "function");
    if (hasCb) {
      fn(...args);
    } else {
      return new Promise((resolve, reject) => {
        fn(...args, cb);

        function cb(err, data) {
          if (err) {
            reject(err);
          } else {
            resolve(data);
          }
        }
      });
    }
  };
}

```