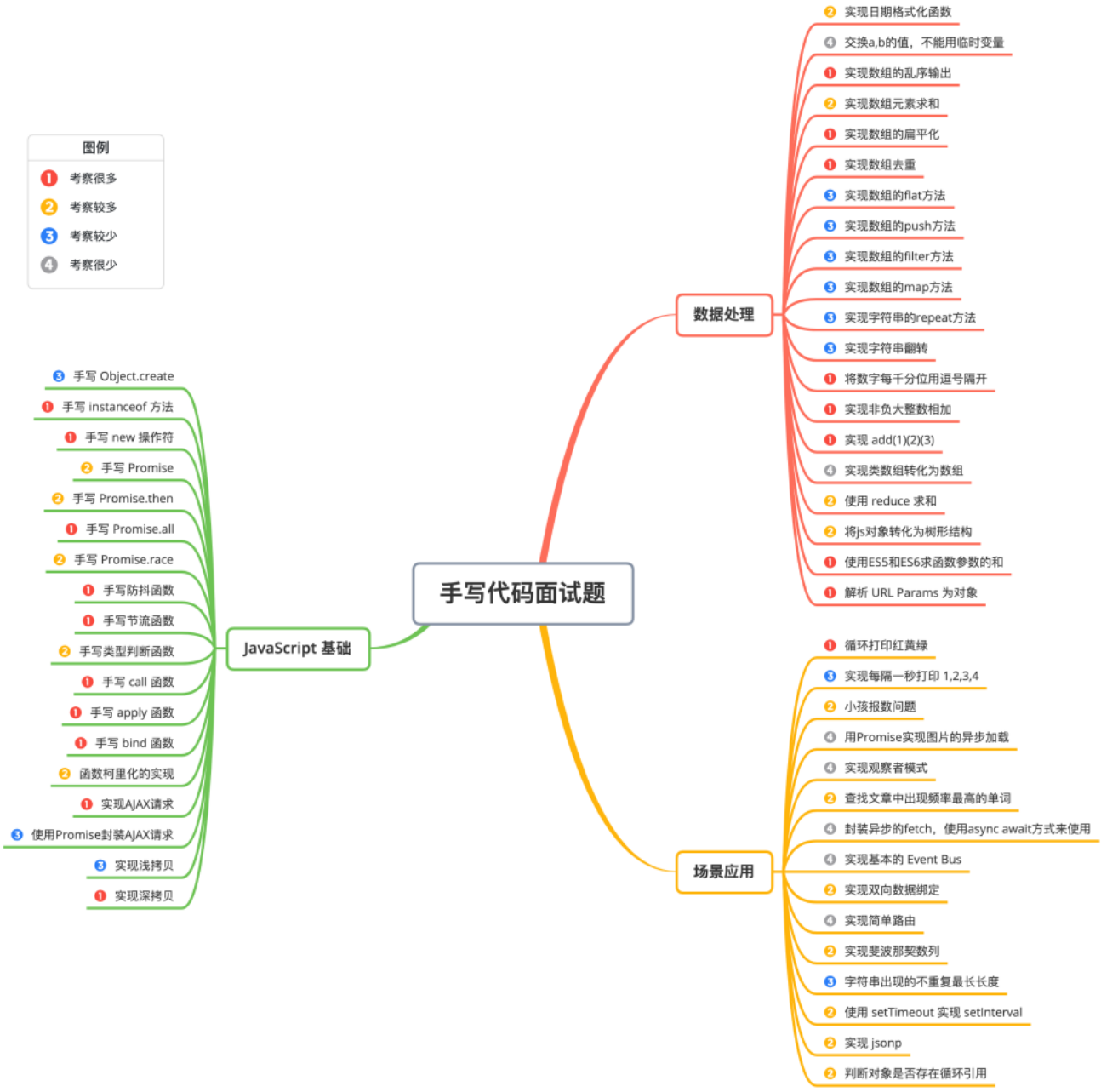
1. 一、JavaScript 基础
    1. 1. 手写 Object.create
    2. 2. 手写 instanceof 方法
    3. 3. 手写 new 操作符
    4. 4. 手写 Promise
    5. 5. 手写 Promise.then
    6. 6. 手写 Promise.all
    7. 7. 手写 Promise.race
    8. 8. 手写防抖函数
    9. 9. 手写节流函数
    10. 10. 手写类型判断函数
    11. 11. 手写 call 函数
    12. 12. 手写 apply 函数
    13. 13. 手写 bind 函数
    14. 14. 函数柯里化的实现
    15. 15. 实现AJAX请求
    16. 16. 使用Promise封装AJAX请求
    17. 17. 实现浅拷贝
        1. （1）Object.assign()
        2. （2）扩展运算符
        3. （3）数组方法实现数组浅拷贝
            1. 1）Array.prototype.slice
            2. 2）Array.prototype.concat
        4. （4）手写实现浅拷贝
    18. 18. 实现深拷贝
        1. （1）JSON.stringify()
        2. （2）函数库lodash的_.cloneDeep方法
        3. （3）手写实现深拷贝函数
2. 二、数据处理
    1. 1. 实现日期格式化函数
    2. 2. 交换a,b的值，不能用临时变量
    3. 3. 实现数组的乱序输出
    4. 4. 实现数组元素求和
    5. 5. 实现数组的扁平化
    6. 6. 实现数组去重
    7. 7. 实现数组的flat方法
    8. 8. 实现数组的push方法
    9. 9. 实现数组的filter方法

10. 10. 实现数组的map方法
11. 11. 实现字符串的repeat方法
12. 12. 实现字符串翻转
13. 13. 将数字每千分位用逗号隔开
14. 14. 实现非负大整数相加
15. 13. 实现 add(1)(2)(3)
16. 14. 实现类数组转化为数组
17. 15. 使用 reduce 求和
18. 16. 将js对象转化为树形结构
19. 17. 使用ES5和ES6求函数参数的和
20. 18. 解析 URL Params 为对象

3. 三、场景应用
    1. 1. 循环打印红黄绿
        1. （1）用 callback 实现
        2. （2）用 promise 实现
        3. （3）用 async/await 实现
    2. 2. 实现每隔一秒打印 1,2,3,4
    3. 3. 小孩报数问题
    4. 4. 用Promise实现图片的异步加载
    5. 5. 实现发布-订阅模式
    6. 6. 查找文章中出现频率最高的单词
    7. 7. 封装异步的fetch，使用async await方式来使用
    8. 8. 实现prototype继承
    9. 9. 实现双向数据绑定
    10. 10. 实现简单路由
    11. 11. 实现斐波那契数列
    12. 12. 字符串出现的不重复最长长度
    13. 13. 使用 setTimeout 实现 setInterval
    14. 14. 实现 jsonp
    15. 15. 判断对象是否存在循环引用



一、JavaScript 基础

1. 手写 Object.create

思路：将传入的对象作为原型

```
function create(obj) {
  function F() {}
  F.prototype = obj
  return new F()
}
```

2. 手写 instanceof 方法

instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

实现步骤：

1. 首先获取类型的原型
2. 然后获得对象的原型
3. 然后一直循环判断对象的原型是否等于类型的原型，直到对象原型为 `null`，因为原型链最终为 `null`

具体实现：

```
function myInstanceof(left, right) {
  let proto = Object.getPrototypeOf(left), // 获取对象的原型
      prototype = right.prototype; // 获取构造函数的 prototype 对象

  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
    if (proto === prototype) return true;

    proto = Object.getPrototypeOf(proto);
  }
}
```

3. 手写 new 操作符

在调用 `new` 的过程中会发生以上四件事情：

（1）首先创建了一个新的空对象

（2）设置原型，将对象的原型设置为函数的 prototype 对象。

（3）让函数的 this 指向这个对象，执行构造函数的代码（为这个新对象添加属性）

（4）判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，就返回这个引用类型的对象。

```
function objectFactory() {
  let newObject = null;
  let constructor = Array.prototype.shift.call(arguments);
  let result = null;
```

```
  // 判断参数是否是一个函数
  if (typeof constructor !== "function") {
    console.error("type error");
    return;
  }
  // 新建一个空对象，对象的原型为构造函数的 prototype 对象
  newObject = Object.create(constructor.prototype);
  // 将 this 指向新建对象，并执行函数
  result = constructor.apply(newObject, arguments);
  // 判断返回对象
  let flag = result && (typeof result === "object" || typeof result ===
"function");
  // 判断返回结果
  return flag ? result : newObject;
}
// 使用方法
objectFactory(构造函数，初始化参数);
```

4. 手写 Promise

```
const PENDING = "pending";
const RESOLVED = "resolved";
const REJECTED = "rejected";

function MyPromise(fn) {
  // 保存初始化状态
  var self = this;

  // 初始化状态
  this.state = PENDING;

  // 用于保存 resolve 或者 rejected 传入的值
  this.value = null;

  // 用于保存 resolve 的回调函数
  this.resolvedCallbacks = [];

  // 用于保存 reject 的回调函数
  this.rejectedCallbacks = [];

  // 状态转变为 resolved 方法
  function resolve(value) {
    // 判断传入元素是否为 Promise 值，如果是，则状态改变必须等待前一个状态改变后再进行改
变
    if (value instanceof MyPromise) {
      return value.then(resolve, reject);
    }

    // 保证代码的执行顺序为本轮事件循环的末尾
    setTimeout(() => {
      // 只有状态为 pending 时才能转变，
      if (self.state === PENDING) {
        // 修改状态
        self.state = RESOLVED;
```

```
        // 设置传入的值
        self.value = value;

        // 执行回调函数
        self.resolvedCallbacks.forEach(callback => {
          callback(value);
        });
      }
    }, 0);
  }

  // 状态转变为 rejected 方法
  function reject(value) {
    // 保证代码的执行顺序为本轮事件循环的末尾
    setTimeout(() => {
      // 只有状态为 pending 时才能转变
      if (self.state === PENDING) {
        // 修改状态
        self.state = REJECTED;

        // 设置传入的值
        self.value = value;

        // 执行回调函数
        self.rejectedCallbacks.forEach(callback => {
          callback(value);
        });
      }
    }, 0);
  }

  // 将两个方法传入函数执行
  try {
    fn(resolve, reject);
  } catch (e) {
    // 遇到错误时，捕获错误，执行 reject 函数
    reject(e);
  }
}

MyPromise.prototype.then = function(onResolved, onRejected) {
  // 首先判断两个参数是否为函数类型，因为这两个参数是可选参数
  onResolved =
    typeof onResolved === "function"
      ? onResolved
      : function(value) {
          return value;
        };

  onRejected =
    typeof onRejected === "function"
      ? onRejected
      : function(error) {
          throw error;
        };

  // 如果是等待状态，则将函数加入对应列表中
  if (this.state === PENDING) {
```

```
    this.resolvedCallbacks.push(onResolved);
    this.rejectedCallbacks.push(onRejected);
  }

  // 如果状态已经凝固，则直接执行对应状态的函数

  if (this.state === RESOLVED) {
    onResolved(this.value);
  }

  if (this.state === REJECTED) {
    onRejected(this.value);
  }
};
```

5. 手写 Promise.then

then 方法返回一个新的 promise 实例，为了在 promise 状态发生变化时（resolve / reject 被调用时）再执行 then 里的函数，我们使用一个 callbacks 数组先把传给 then的函数暂存起来，等状态改变时再调用。

那么，怎么保证后一个 **then 里的方法在前一个 **then**（可能是异步）结束之后再执行呢？**

我们可以将传给 then 的函数和新 promise 的 resolve 一起 push 到前一个 promise 的 callbacks 数组中，达到承前启后的效果：

- 承前：当前一个 promise 完成后，调用其 resolve 变更状态，在这个 resolve 里会依次调用 callbacks 里的回调，这样就执行了 then 里的方法了
- 启后：上一步中，当 then 里的方法执行完成后，返回一个结果，如果这个结果是个简单的值，就直接调用新 promise 的 resolve，让其状态变更，这又会依次调用新 promise 的 callbacks 数组里的方法，循环往复。。如果返回的结果是个 promise，则需要等它完成之后再触发新 promise 的 resolve，所以可以在其结果的 then 里调用新 promise 的 resolve

```
then(onFulfilled, onReject){
    // 保存前一个promise的this
    const self = this;
    return new MyPromise((resolve, reject) => {
      // 封装前一个promise成功时执行的函数
      let fulfilled = () => {
        try{
          const result = onFulfilled(self.value); // 承前
          return result instanceof MyPromise? result.then(resolve, reject) :
resolve(result); //启后
        }catch(err){
          reject(err)
```

```
        }
      }
      // 封装前一个promise失败时执行的函数
      let rejected = () => {
        try{
          const result = onReject(self.reason);
          return result instanceof MyPromise? result.then(resolve, reject) :
reject(result);
        }catch(err){
          reject(err)
        }
      }
      switch(self.status){
        case PENDING:
          self.onFulfilledCallbacks.push(fulfilled);
          self.onRejectedCallbacks.push(rejected);
          break;
        case FULFILLED:
          fulfilled();
          break;
        case REJECT:
          rejected();
          break;
      }
    })
  }
```

注意：

- 连续多个 `then` 里的回调方法是同步注册的，但注册到了不同的 `callbacks` 数组中，因为每次 `then` 都返回新的 `promise` 实例（参考上面的例子和图）
- 注册完成后开始执行构造函数中的异步事件，异步完成之后依次调用 `callbacks` 数组中提前注册的回调

6. 手写 Promise.all

1) 核心思路

1. 接收一个 Promise 实例的数组或具有 Iterator 接口的对象作为参数
2. 这个方法返回一个新的 promise 对象，
3. 遍历传入的参数，用Promise.resolve()将参数"包一层"，使其变成一个promise对象
4. 参数所有回调成功才是成功，返回值数组与参数顺序一致
5. 参数数组其中一个失败，则触发失败状态，第一个触发失败的 Promise 错误信息作为 Promise.all 的错误信息。

2）实现代码

一般来说，Promise.all 用来处理多个并发请求，也是为了页面数据构造的方便，将一个页面所用到的在不同接口的数据一起请求过来，不过，如果其中一个接口失败了，多个请求也就失败了，页面可能啥也出不来，这就看当前页面的耦合程度了

```
function promiseAll(promises) {
  return new Promise(function(resolve, reject) {
    if(!Array.isArray(promises)){
        throw new TypeError(`argument must be a array`)
    }
    var resolvedCounter = 0;
    var promiseNum = promises.length;
    var resolvedResult = [];
    for (let i = 0; i < promiseNum; i++) {
      Promise.resolve(promises[i]).then(value=>{
        resolvedCounter++;
        resolvedResult[i] = value;
        if (resolvedCounter == promiseNum) {
            return resolve(resolvedResult)
          }
      },error=>{
        return reject(error)
      })
    }
  })
}
// test
let p1 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve(1)
    }, 1000)
})
let p2 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve(2)
    }, 2000)
})
let p3 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve(3)
    }, 3000)
})
promiseAll([p3, p1, p2]).then(res => {
    console.log(res) // [3, 1, 2]
})
```

7. 手写 Promise.race

该方法的参数是 Promise 实例数组, 然后其 then 注册的回调方法是数组中的某一个 Promise 的状态变为 fulfilled 的时候就执行. 因为 Promise 的状态只能改变一次, 那么我

们只需要把 Promise.race 中产生的 Promise 对象的 resolve 方法, 注入到数组中的每一个 Promise 实例中的回调函数中即可.

```
Promise.race = function (args) {
  return new Promise((resolve, reject) => {
    for (let i = 0, len = args.length; i < len; i++) {
      args[i].then(resolve, reject)
    }
  })
}
```

8. 手写防抖函数

函数防抖是指在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。这可以使用在一些点击请求的事件上，避免因为用户的多次点击向后端发送多次请求。

```
// 函数防抖的实现
function debounce(fn, wait) {
  let timer = null;

  return function() {
    let context = this,
        args = arguments;

    // 如果此时存在定时器的话，则取消之前的定时器重新记时
    if (timer) {
      clearTimeout(timer);
      timer = null;
    }

    // 设置定时器，使事件间隔指定事件后执行
    timer = setTimeout(() => {
      fn.apply(context, args);
    }, wait);
  };
}
```

9. 手写节流函数

函数节流是指规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。节流可以使用在 scroll 函数的事件监听上，通过事件节流来降低事件调用的频率。

```
// 函数节流的实现;
function throttle(fn, delay) {
  let curTime = Date.now();

  return function() {
    let context = this,
        args = arguments,
        nowTime = Date.now();

    // 如果两次时间间隔超过了指定时间，则执行函数。
    if (nowTime - curTime >= delay) {
      curTime = Date.now();
      return fn.apply(context, args);
    }
  };
}
```

10. 手写类型判断函数

```
function getType(value) {
  // 判断数据是 null 的情况
  if (value === null) {
    return value + "";
  }
  // 判断数据是引用类型的情况
  if (typeof value === "object") {
    let valueClass = Object.prototype.toString.call(value),
      type = valueClass.split(" ")[1].split("");
    type.pop();
    return type.join("").toLowerCase();
  } else {
    // 判断数据是基本数据类型的情况和函数的情况
    return typeof value;
  }
}
```

11. 手写 call 函数

call 函数的实现步骤：

1. 判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
2. 判断传入上下文对象是否存在，如果不存在，则设置为 window 。
3. 处理传入的参数，截取第一个参数后的所有参数。
4. 将函数作为上下文对象的一个属性。
5. 使用上下文对象来调用这个方法，并保存返回结果。

6. 删除刚才新增的属性。
7. 返回结果。

```
// call函数实现
Function.prototype.myCall = function(context) {
  // 判断调用对象
  if (typeof this !== "function") {
    console.error("type error");
  }
  // 获取参数
  let args = [...arguments].slice(1),
      result = null;
  // 判断 context 是否传入，如果未传入则设置为 window
  context = context || window;
  // 将调用函数设为对象的方法
  context.fn = this;
  // 调用函数
  result = context.fn(...args);
  // 将属性删除
  delete context.fn;
  return result;
};
```

12. 手写 apply 函数

apply 函数的实现步骤：

1. 判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
2. 判断传入上下文对象是否存在，如果不存在，则设置为 window 。
3. 将函数作为上下文对象的一个属性。
4. 判断参数值是否传入
5. 使用上下文对象来调用这个方法，并保存返回结果。
6. 删除刚才新增的属性
7. 返回结果

```
// apply 函数实现
Function.prototype.myApply = function(context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }
  let result = null;
  // 判断 context 是否存在，如果未传入则为 window
  context = context || window;
  // 将函数设为对象的方法
  context.fn = this;
```

```
  // 调用方法
  if (arguments[1]) {
    result = context.fn(...arguments[1]);
  } else {
    result = context.fn();
  }
  // 将属性删除
  delete context.fn;
  return result;
};
```

13. 手写 bind 函数

bind 函数的实现步骤：

1. 判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
2. 保存当前函数的引用，获取其余传入参数值。
3. 创建一个函数返回
4. 函数内部使用 apply 来绑定函数调用，需要判断函数作为构造函数的情况，这个时候需要传入当前函数的 this 给 apply 调用，其余情况都传入指定的上下文对象。

```
// bind 函数实现
Function.prototype.myBind = function(context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }
  // 获取参数
  var args = [...arguments].slice(1),
      fn = this;
  return function Fn() {
    // 根据调用方式，传入不同绑定值
    return fn.apply(
      this instanceof Fn ? this : context,
      args.concat(...arguments)
    );
  };
};
```

14. 函数柯里化的实现

函数柯里化指的是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术。

```
function curry(fn, args) {
  // 获取函数需要的参数长度
  let length = fn.length;

  args = args || [];

  return function() {
    let subArgs = args.slice(0);

    // 拼接得到现有的所有参数
    for (let i = 0; i < arguments.length; i++) {
      subArgs.push(arguments[i]);
    }

    // 判断参数的长度是否已经满足函数所需参数的长度
    if (subArgs.length >= length) {
      // 如果满足，执行函数
      return fn.apply(this, subArgs);
    } else {
      // 如果不满足，递归返回科里化的函数，等待参数的传入
      return curry.call(this, fn, subArgs);
    }
  };
}

// es6 实现
function curry(fn, ...args) {
  return fn.length <= args.length ? fn(...args) : curry.bind(null, fn, ...args);
}
```

15. 实现AJAX请求

AJAX是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的 异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。

创建AJAX请求的步骤：

- 创建一个 **XMLHttpRequest** 对象。
- 在这个对象上使用 **open** 方法创建一个 **HTTP** 请求，open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。
- 在发起请求前，可以为这个对象添加一些信息和监听函数。比如说可以通过 setRequestHeader 方法来为请求添加头信息。还可以为这个对象添加一个状态监听函数。一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发 onreadystatechange 事件，可以通过设置监听函数，来处理请求成功后的结果。当对象的 readyState 变为 4 的时候，代表服务器返回的数据接收完成，这个时候可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候就可以通过 response 中的数据来对页面进行更新了。

- 当对象的属性和监听函数设置完成后，最后调用 **sent 方法来向服务器发起请求，**可以传入参数作为发送的数据体。

```
const SERVER_URL = "/server";
let xhr = new XMLHttpRequest();
// 创建 Http 请求
xhr.open("GET", SERVER_URL, true);
// 设置状态监听函数
xhr.onreadystatechange = function() {
  if (this.readyState !== 4) return;
  // 当请求成功时
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
  }
};
// 设置请求失败时的监听函数
xhr.onerror = function() {
  console.error(this.statusText);
};
// 设置请求头信息
xhr.responseType = "json";
xhr.setRequestHeader("Accept", "application/json");
// 发送 Http 请求
xhr.send(null);
```

16. 使用Promise封装AJAX请求

```
// promise 封装实现：
function getJSON(url) {
  // 创建一个 promise 对象
  let promise = new Promise(function(resolve, reject) {
    let xhr = new XMLHttpRequest();
    // 新建一个 http 请求
    xhr.open("GET", url, true);
    // 设置状态的监听函数
    xhr.onreadystatechange = function() {
      if (this.readyState !== 4) return;
      // 当请求成功或失败时，改变 promise 的状态
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    // 设置错误监听函数
    xhr.onerror = function() {
      reject(new Error(this.statusText));
    };
    // 设置响应的数据类型
```

```
    xhr.responseType = "json";
    // 设置请求头信息
    xhr.setRequestHeader("Accept", "application/json");
    // 发送 http 请求
    xhr.send(null);
  });
  return promise;
}
```

17. 实现浅拷贝

浅拷贝是指，一个新的对象对原始对象的属性值进行精确地拷贝，如果拷贝的是基本数据类型，拷贝的就是基本数据类型的值，如果是引用数据类型，拷贝的就是内存地址。如果其中一个对象的引用内存地址发生改变，另一个对象也会发生变化。

（1）Object.assign()

`Object.assign()`是ES6中对象的拷贝方法，接受的第一个参数是目标对象，其余参数是源对象，用法：`Object.assign(target, source_1, ···)`，该方法可以实现浅拷贝，也可以实现一维对象的深拷贝。

注意：

- 如果目标对象和源对象有同名属性，或者多个源对象有同名属性，则后面的属性会覆盖前面的属性。
- 如果该函数只有一个参数，当参数为对象时，直接返回该对象；当参数不是对象时，会先将参数转为对象然后返回。
- 因为`null` 和 `undefined` 不能转化为对象，所以第一个参数不能为`null`或`undefined`，会报错。

```
let target = {a: 1};
let object2 = {b: 2};
let object3 = {c: 3};
Object.assign(target,object2,object3);
console.log(target);  // {a: 1, b: 2, c: 3}
```

（2）扩展运算符

使用扩展运算符可以在构造字面量对象的时候，进行属性的拷贝。语法：`let cloneObj = { ...obj };`

```
```

```
let obj1 = {a:1,b:{c:1}}
let obj2 = {...obj1};
obj1.a = 2;
console.log(obj1); //{a:2,b:{c:1}}
console.log(obj2); //{a:1,b:{c:1}}
obj1.b.c = 2;
console.log(obj1); //{a:2,b:{c:2}}
console.log(obj2); //{a:1,b:{c:2}}
```

（3）数组方法实现数组浅拷贝

1）Array.prototype.slice

- `slice()`方法是JavaScript数组的一个方法，这个方法可以从已有数组中返回选定的元素：用法：`array.slice(start, end)`，该方法不会改变原始数组。
- 该方法有两个参数，两个参数都可选，如果两个参数都不写，就可以实现一个数组的浅拷贝。

```
let arr = [1,2,3,4];
console.log(arr.slice()); // [1,2,3,4]
console.log(arr.slice() === arr); //false
```

2）Array.prototype.concat

- `concat()` 方法用于合并两个或多个数组。此方法不会更改现有数组，而是返回一个新数组。
- 该方法有两个参数，两个参数都可选，如果两个参数都不写，就可以实现一个数组的浅拷贝。

```
let arr = [1,2,3,4];
console.log(arr.concat()); // [1,2,3,4]
console.log(arr.concat() === arr); //false
```

（4）手写实现浅拷贝

```
// 浅拷贝的实现;

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];
    }
  }

  return newObject;
}// 浅拷贝的实现;

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];
    }
  }

  return newObject;
}// 浅拷贝的实现;
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];
    }
  }
  return newObject;
}
```

18. 实现深拷贝

- **浅拷贝：**浅拷贝指的是将一个对象的属性值复制到另一个对象，如果有的属性的值为引用类型的话，那么会将这个引用的地址复制给对象，因此两个对象会有同一个引用类型的引用。浅拷贝可以使用 Object.assign 和展开运算符来实现。
- **深拷贝：**深拷贝相对浅拷贝而言，如果遇到属性值为引用类型的时候，它新建一个引用类型并将对应的值复制给它，因此对象获得的一个新的引用类型而不是一个原有类型的引用。深拷贝对于一些对象可以使用 JSON 的两个函数来实现，但是

由于 JSON 的对象格式比 js 的对象格式更加严格，所以如果属性值里边出现函数或者 Symbol 类型的值时，会转换失败

（1）JSON.stringify()

- `JSON.parse(JSON.stringify(obj))`是目前比较常用的深拷贝方法之一，它的原理就是利用`JSON.stringify` 将`js`对象序列化（JSON字符串），再使用`JSON.parse`来反序列化(还原)`js`对象。
- 这个方法可以简单粗暴的实现深拷贝，但是还存在问题，拷贝的对象中如果有函数，undefined，symbol，当使用过`JSON.stringify()`进行处理之后，都会消失。

```
let obj1 = {  a: 0,
              b: {
                 c: 0
                 }
           };
let obj2 = JSON.parse(JSON.stringify(obj1));
obj1.a = 1;
obj1.b.c = 1;
console.log(obj1); // {a: 1, b: {c: 1}}
console.log(obj2); // {a: 0, b: {c: 0}}
```

（2）函数库lodash的_.cloneDeep方法

该函数库也有提供_.cloneDeep用来做 Deep Copy

```
var _ = require('lodash');
var obj1 = {
    a: 1,
    b: { f: { g: 1 } },
    c: [1, 2, 3]
};
var obj2 = _.cloneDeep(obj1);
console.log(obj1.b.f === obj2.b.f);// false
```

（3）手写实现深拷贝函数

```
// 深拷贝的实现
function deepCopy(object) {
  if (!object || typeof object !== "object") return;

  let newObject = Array.isArray(object) ? [] : {};

  for (let key in object) {
```

```
    if (object.hasOwnProperty(key)) {
      newObject[key] =
        typeof object[key] === "object" ? deepCopy(object[key]) : object[key];
    }
  }

  return newObject;
}
```

二、数据处理

1. 实现日期格式化函数

输入：

```
dateFormat(new Date('2020-12-01'), 'yyyy/MM/dd') // 2020/12/01
dateFormat(new Date('2020-04-01'), 'yyyy/MM/dd') // 2020/04/01
dateFormat(new Date('2020-04-01'), 'yyyy年MM月dd日') // 2020年04月01日
```

```
const dateFormat = (dateInput, format)=>{
    var day = dateInput.getDate()
    var month = dateInput.getMonth() + 1
    var year = dateInput.getFullYear()
    format = format.replace(/yyyy/, year)
    format = format.replace(/MM/,month)
    format = format.replace(/dd/,day)
    return format
}
```

2. 交换a,b的值，不能用临时变量

巧妙的利用两个数的和、差：

```
a = a + b
b = a - b
a = a - b
```

3. 实现数组的乱序输出

主要的实现思路就是：

- 取出数组的第一个元素，随机产生一个索引值，将该第一个元素和这个索引对应的元素进行交换。
- 第二次取出数据数组第二个元素，随机产生一个除了索引为1的之外的索引值，并将第二个元素与该索引值对应的元素进行交换
- 按照上面的规律执行，直到遍历完成

```
var arr = [1,2,3,4,5,6,7,8,9,10];
for (var i = 0; i < arr.length; i++) {
  const randomIndex = Math.round(Math.random() * (arr.length - 1 - i)) + i;
  [arr[i], arr[randomIndex]] = [arr[randomIndex], arr[i]];
}
console.log(arr)
```

还有一方法就是倒序遍历：

```
var arr = [1,2,3,4,5,6,7,8,9,10];
let length = arr.length,
    randomIndex,
    temp;
  while (length) {
    randomIndex = Math.floor(Math.random() * length--);
    temp = arr[length];
    arr[length] = arr[randomIndex];
    arr[randomIndex] = temp;
  }
console.log(arr)
```

4. 实现数组元素求和

- arr=[1,2,3,4,5,6,7,8,9,10]，求和

```
let arr=[1,2,3,4,5,6,7,8,9,10]
let sum = arr.reduce( (total,i) => total += i,0);
console.log(sum);
```

- arr=[1,2,3,[[4,5],6],7,8,9]，求和

```
var = arr=[1,2,3,[[4,5],6],7,8,9]
let arr= arr.toString().split(',').reduce( (total,i) => total += Number(i),0);
console.log(arr);
```

递归实现：

```
let arr = [1, 2, 3, 4, 5, 6]

function add(arr) {
    if (arr.length == 1) return arr[0]
    return arr[0] + add(arr.slice(1))
}
console.log(add(arr)) // 21
```

5. 实现数组的扁平化

（1）递归实现

普通的递归思路很容易理解，就是通过循环递归的方式，一项一项地去遍历，如果每一项还是一个数组，那么就继续往下遍历，利用递归程序的方法，来实现数组的每一项的连接：

```
let arr = [1, [2, [3, 4, 5]]];
function flatten(arr) {
  let result = [];

  for(let i = 0; i < arr.length; i++) {
    if(Array.isArray(arr[i])) {
      result = result.concat(flatten(arr[i]));
    } else {
      result.push(arr[i]);
    }
  }
  return result;
}
flatten(arr);  //  [1, 2, 3, 4, 5]
```

（2）reduce 函数迭代

从上面普通的递归函数中可以看出，其实就是对数组的每一项进行处理，那么其实也可以用reduce 来实现数组的拼接，从而简化第一种方法的代码，改造后的代码如下所示：

```
let arr = [1, [2, [3, 4]]];
function flatten(arr) {
    return arr.reduce(function(prev, next){
        return prev.concat(Array.isArray(next) ? flatten(next) : next)
    }, [])
}
console.log(flatten(arr));//  [1, 2, 3, 4, 5]
```

（3）扩展运算符实现

这个方法的实现，采用了扩展运算符和 some 的方法，两者共同使用，达到数组扁平化的目的：

```
let arr = [1, [2, [3, 4]]];
function flatten(arr) {
    while (arr.some(item => Array.isArray(item))) {
        arr = [].concat(...arr);
    }
    return arr;
}
console.log(flatten(arr)); //  [1, 2, 3, 4, 5]
```

（4）split 和 toString

可以通过 split 和 toString 两个方法来共同实现数组扁平化，由于数组会默认带一个 toString 的方法，所以可以把数组直接转换成逗号分隔的字符串，然后再用 split 方法把字符串重新转换为数组，如下面的代码所示：

```
let arr = [1, [2, [3, 4]]];
function flatten(arr) {
    return arr.toString().split(',');
}
console.log(flatten(arr)); //  [1, 2, 3, 4, 5]
```

通过这两个方法可以将多维数组直接转换成逗号连接的字符串，然后再重新分隔成数组。

（5）ES6 中的 flat

我们还可以直接调用 ES6 中的 flat 方法来实现数组扁平化。flat 方法的语法：`arr.flat([depth])`

其中 depth 是 flat 的参数，depth 是可以传递数组的展开深度（默认不填、数值是 1），即展开一层数组。如果层数不确定，参数可以传进 Infinity，代表不论多少层都要展开：

```
let arr = [1, [2, [3, 4]]];
function flatten(arr) {
  return arr.flat(Infinity);
}
console.log(flatten(arr)); //  [1, 2, 3, 4, 5]
```

可以看出，一个嵌套了两层的数组，通过将 flat 方法的参数设置为 Infinity，达到了我们预期的效果。其实同样也可以设置成 2，也能实现这样的效果。在编程过程中，如果数组的嵌套层数不确定，最好直接使用 Infinity，可以达到扁平化。

（6）正则和 JSON 方法

在第4种方法中已经使用 toString 方法，其中仍然采用了将 JSON.stringify 的方法先转换为字符串，然后通过正则表达式过滤掉字符串中的数组的方括号，最后再利用 JSON.parse 把它转换成数组：

```
let arr = [1, [2, [3, [4, 5]]], 6];
function flatten(arr) {
  let str = JSON.stringify(arr);
  str = str.replace(/(\[|\])/g, '');
  str = '[' + str + ']';
  return JSON.parse(str);
}
console.log(flatten(arr)); //  [1, 2, 3, 4, 5]
```

6. 实现数组去重

给定某无序数组，要求去除数组中的重复数字并且返回新的无重复数组。

ES6方法（使用数据结构集合）：

```
const array = [1, 2, 3, 5, 1, 5, 9, 1, 2, 8];

Array.from(new Set(array)); // [1, 2, 3, 5, 9, 8]
```

ES5方法：使用map存储不重复的数字

```
const array = [1, 2, 3, 5, 1, 5, 9, 1, 2, 8];

uniqueArray(array); // [1, 2, 3, 5, 9, 8]

function uniqueArray(array) {
  let map = {};
  let res = [];
  for(var i = 0; i < array.length; i++) {
    if(!map.hasOwnProperty([array[i]])) {
      map[array[i]] = 1;
      res.push(array[i]);
    }
```

```
  }
  return res;
}
```

7. 实现数组的flat方法

```
function _flat(arr, depth) {
  if(!Array.isArray(arr) || depth <= 0) {
    return arr;
  }
  return arr.reduce((prev, cur) => {
    if (Array.isArray(cur)) {
      return prev.concat(_flat(cur, depth - 1))
    } else {
      return prev.concat(cur);
    }
  }, []);
}
```

8. 实现数组的push方法

```
let arr = [];
Array.prototype.push = function() {
    for( let i = 0 ; i < arguments.length ; i++){
        this[this.length] = arguments[i] ;
    }
    return this.length;
}
```

9. 实现数组的filter方法

```
Array.prototype._filter = function(fn) {
    if (typeof fn !== "function") {
        throw Error('参数必须是一个函数');
    }
    const res = [];
    for (let i = 0, len = this.length; i < len; i++) {
        fn(this[i]) && res.push(this[i]);
    }
    return res;
}
```

10. 实现数组的map方法

```
Array.prototype._map = function(fn) {
   if (typeof fn !== "function") {
        throw Error('参数必须是一个函数');
    }
    const res = [];
    for (let i = 0, len = this.length; i < len; i++) {
        res.push(fn(this[i]));
    }
    return res;
}
```

11. 实现字符串的repeat方法

输入字符串s，以及其重复的次数，输出重复的结果，例如输入abc，2，输出abcabc。

```
function repeat(s, n) {
    return (new Array(n + 1)).join(s);
}
```

递归：

```
function repeat(s, n) {
    return (n > 0) ? s.concat(repeat(s, --n)) : "";
}
```

12. 实现字符串翻转

在字符串的原型链上添加一个方法，实现字符串翻转：

```
String.prototype._reverse = function(a){
    return a.split("").reverse().join("");
}
var obj = new String();
var res = obj._reverse ('hello');
console.log(res);    // olleh
```

需要注意的是，必须通过实例化对象之后再去调用定义的方法，不然找不到该方法。

13. 将数字每千分位用逗号隔开

数字有小数版本：

```
let format = n => {
    let num = n.toString() // 转成字符串
    let decimals = ''
        // 判断是否有小数
    num.indexOf('.') > -1 ? decimals = num.split('.')[1] : decimals
    let len = num.length
    if (len <= 3) {
        return num
    } else {
        let temp = ''
        let remainder = len % 3
        decimals ? temp = '.' + decimals : temp
        if (remainder > 0) { // 不是3的整数倍
            return num.slice(0, remainder) + ',' + num.slice(remainder,
len).match(/\d{3}/g).join(',') + temp
        } else { // 是3的整数倍
            return num.slice(0, len).match(/\d{3}/g).join(',') + temp
        }
    }
}
format(12323.33)  // '12,323.33'
```

数字无小数版本：

```
let format = n => {
    let num = n.toString()
    let len = num.length
    if (len <= 3) {
        return num
    } else {
        let remainder = len % 3
        if (remainder > 0) { // 不是3的整数倍
            return num.slice(0, remainder) + ',' + num.slice(remainder,
len).match(/\d{3}/g).join(',')
        } else { // 是3的整数倍
            return num.slice(0, len).match(/\d{3}/g).join(',')
        }
    }
}
format(1232323)  // '1,232,323'
```

14. 实现非负大整数相加

JavaScript对数值有范围的限制，限制如下：

```
Number.MAX_VALUE // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER // 9007199254740991
Number.MIN_VALUE // 5e-324
Number.MIN_SAFE_INTEGER // -9007199254740991
```

如果想要对一个超大的整数(> Number.MAX_SAFE_INTEGER)进行加法运算，但是又想输出一般形式，那么使用 + 是无法达到的，一旦数字超过 Number.MAX_SAFE_INTEGER 数字会被立即转换为科学计数法，并且数字精度相比以前将会有误差。

实现一个算法进行大数的相加：

```
function sumBigNumber(a, b) {
  let res = '';
  let temp = 0;

  a = a.split('');
  b = b.split('');

  while (a.length || b.length || temp) {
    temp += ~~a.pop() + ~~b.pop();
    res = (temp % 10) + res;
    temp  = temp > 9
  }
  return res.replace(/^0+/, '');
}
```

其主要的思路如下：

- 首先用字符串的方式来保存大数，这样数字在数学表示上就不会发生变化
- 初始化res，temp来保存中间的计算结果，并将两个字符串转化为数组，以便进行每一位的加法运算
- 将两个数组的对应的位进行相加，两个数相加的结果可能大于10，所以可能要仅为，对10进行取余操作，将结果保存在当前位
- 判断当前位是否大于9，也就是是否会进位，若是则将temp赋值为true，因为在加法运算中，true会自动隐式转化为1，以便于下一次相加
- 重复上述操作，直至计算结束

13. 实现 add(1)(2)(3)

函数柯里化概念： 柯里化（Currying）是把接受多个参数的函数转变为接受一个单一参数的函数，并且返回接受余下的参数且返回结果的新函数的技术。

1）粗暴版

```
function add (a) {
return function (b) {
    return function (c) {
      return a + b + c;
    }
}
}
console.log(add(1)(2)(3)); // 6
```

2）柯里化解决方案

- 参数长度固定

```
var add = function (m) {
  var temp = function (n) {
    return add(m + n);
  }
  temp.toString = function () {
    return m;
  }
  return temp;
};
console.log(add(3)(4)(5)); // 12
console.log(add(3)(6)(9)(25)); // 43
```

对于add(3)(4)(5)，其执行过程如下：

1. 先执行add(3)，此时m=3，并且返回temp函数；
2. 执行temp(4)，这个函数内执行add(m+n)，n是此次传进来的数值4，m值还是上一步中的3，所以add(m+n)=add(3+4)=add(7)，此时m=7，并且返回temp函数
3. 执行temp(5)，这个函数内执行add(m+n)，n是此次传进来的数值5，m值还是上一步中的7，所以add(m+n)=add(7+5)=add(12)，此时m=12，并且返回temp函数
4. 由于后面没有传入参数，等于返回的temp函数不被执行而是打印，了解JS的朋友都知道对象的toString是修改对象转换字符串的方法，因此代码中temp函数的toString函数return m值，而m值是最后一步执行函数时的值m=12，所以返回值是12。

- 参数长度不固定

```
function add (...args) {
    //求和
    return args.reduce((a, b) => a + b)
}
function currying (fn) {
    let args = []
    return function temp (...newArgs) {
```

```
        if (newArgs.length) {
            args = [
                ...args,
                ...newArgs
            ]
            return temp
        } else {
            let val = fn.apply(this, args)
            args = [] //保证再次调用时清空
            return val
        }
    }
}
let addCurry = currying(add)
console.log(addCurry(1)(2)(3)(4, 5))  //15
console.log(addCurry(1)(2)(3, 4, 5))  //15
console.log(addCurry(1)(2, 3, 4, 5))  //15
```

14. 实现类数组转化为数组

类数组转换为数组的方法有这样几种：

- 通过 call 调用数组的 slice 方法来实现转换

```
Array.prototype.slice.call(arrayLike);
```

- 通过 call 调用数组的 splice 方法来实现转换

```
Array.prototype.splice.call(arrayLike, 0);
```

- 通过 apply 调用数组的 concat 方法来实现转换

```
Array.prototype.concat.apply([], arrayLike);
```

- 通过 Array.from 方法来实现转换

```
Array.from(arrayLike);
```

15. 使用 reduce 求和

arr = [1,2,3,4,5,6,7,8,9,10]，求和

```
let arr = [1,2,3,4,5,6,7,8,9,10]
arr.reduce((prev, cur) => { return prev + cur }, 0)
```

arr = [1,2,3,[[4,5],6],7,8,9]，求和

```
let arr = [1,2,3,4,5,6,7,8,9,10]
arr.flat(Infinity).reduce((prev, cur) => { return prev + cur }, 0)
```

arr = [{a:1, b:3}, {a:2, b:3, c:4}, {a:3}]，求和

```
let arr = [{a:9, b:3, c:4}, {a:1, b:3}, {a:3}]

arr.reduce((prev, cur) => {
    return prev + cur["a"];
}, 0)
```

16. 将js对象转化为树形结构

```
// 转换前:
source = [{
            id: 1,
            pid: 0,
            name: 'body'
          }, {
            id: 2,
            pid: 1,
            name: 'title'
          }, {
            id: 3,
            pid: 2,
            name: 'div'
          }]
// 转换为:
tree = [{
          id: 1,
          pid: 0,
          name: 'body',
          children: [{
            id: 2,
            pid: 1,
            name: 'title',
            children: [{
              id: 3,
              pid: 1,
```

```
            name: 'div'
          }]
        }
      }]
```

代码实现：

```
function jsonToTree(data) {
  // 初始化结果数组，并判断输入数据的格式
  let result = []
  if(!Array.isArray(data)) {
    return result
  }
  // 使用map，将当前对象的id与当前对象对应存储起来
  let map = {};
  data.forEach(item => {
    map[item.id] = item;
  });
  //
  data.forEach(item => {
    let parent = map[item.pid];
    if(parent) {
      (parent.children || (parent.children = [])).push(item);
    } else {
      result.push(item);
    }
  });
  return result;
}
```

17. 使用ES5和ES6求函数参数的和

ES5：

```
function sum() {
    let sum = 0
    Array.prototype.forEach.call(arguments, function(item) {
        sum += item * 1
    })
    return sum
}
```

ES6：

```
function sum(...nums) {
    let sum = 0
    nums.forEach(function(item) {
```

```
        sum += item * 1
    })
    return sum
}
```

18. 解析 URL Params 为对象

```
let url = 'http://www.domain.com/?
user=anonymous&id=123&id=456&city=%E5%8C%97%E4%BA%AC&enabled';
parseParam(url)
/* 结果
{ user: 'anonymous',
  id: [ 123, 456 ], // 重复出现的 key 要组装成数组，能被转成数字的就转成数字类型
  city: '北京', // 中文需解码
  enabled: true, // 未指定值得 key 约定为 true
}
*/
```

```
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;
}
```

三、场景应用

1. 循环打印红黄绿

下面来看一道比较典型的问题，通过这个问题来对比几种异步编程方法：**红灯 3s 亮一次，绿灯 1s 亮一次，黄灯 2s 亮一次；如何让三个灯不断交替重复亮灯？**

三个亮灯函数：

```
function red() {
    console.log('red');
}
function green() {
    console.log('green');
}
function yellow() {
    console.log('yellow');
}
```

这道题复杂的地方在于**需要“交替重复”亮灯**，而不是“亮完一次”就结束了。

（1）用 callback 实现

```
const task = (timer, light, callback) => {
    setTimeout(() => {
        if (light === 'red') {
            red()
        }
        else if (light === 'green') {
            green()
        }
        else if (light === 'yellow') {
            yellow()
        }
        callback()
    }, timer)
}
task(3000, 'red', () => {
    task(2000, 'green', () => {
        task(1000, 'yellow', Function.prototype)
    })
})
```

这里存在一个 bug：代码只是完成了一次流程，执行后红黄绿灯分别只亮一次。该如何让它交替重复进行呢？

上面提到过递归，可以递归亮灯的一个周期：

```
const step = () => {
    task(3000, 'red', () => {
        task(2000, 'green', () => {
            task(1000, 'yellow', step)
```

```
        })
    })
}
step()
```

注意看黄灯亮的回调里又再次调用了 **step** 方法 以完成循环亮灯。

（2）用 promise 实现

```
const task = (timer, light) =>
    new Promise((resolve, reject) => {
        setTimeout(() => {
            if (light === 'red') {
                red()
            }
            else if (light === 'green') {
                green()
            }
            else if (light === 'yellow') {
                yellow()
            }
            resolve()
        }, timer)
    })
const step = () => {
    task(3000, 'red')
        .then(() => task(2000, 'green'))
        .then(() => task(2100, 'yellow'))
        .then(step)
}
step()
```

这里将回调移除，在一次亮灯结束后，resolve 当前 promise，并依然使用递归进行。

（3）用 async/await 实现

```
const taskRunner =  async () => {
    await task(3000, 'red')
    await task(2000, 'green')
    await task(2100, 'yellow')
    taskRunner()
}
taskRunner()
```

2. 实现每隔一秒打印 1,2,3,4

```
// 使用闭包实现
for (var i = 0; i < 5; i++) {
  (function(i) {
    setTimeout(function() {
      console.log(i);
    }, i * 1000);
  })(i);
}
// 使用 let 块级作用域
for (let i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, i * 1000);
}
```

3. 小孩报数问题

有30个小孩儿，编号从1-30，围成一圈依此报数，1、2、3 数到 3 的小孩儿退出这个圈， 然后下一个小孩 重新报数 1、2、3，问最后剩下的那个小孩儿的编号是多少?

```
function childNum(num, count){
    let allplayer = [];
    for(let i = 0; i < num; i++){
        allplayer[i] = i + 1;
    }

    let exitCount = 0;    // 离开人数
    let counter = 0;      // 记录报数
    let curIndex = 0;     // 当前下标

    while(exitCount < num - 1){
        if(allplayer[curIndex] !== 0) counter++;

        if(counter == count){
            allplayer[curIndex] = 0;
            counter = 0;
            exitCount++;
        }
        curIndex++;
        if(curIndex == num){
            curIndex = 0
        };
    }
    for(i = 0; i < num; i++){
        if(allplayer[i] !== 0){
            return allplayer[i]
        }
    }
}
childNum(30, 3)
```

4. 用Promise实现图片的异步加载

```
let imageAsync=(url)=>{
            return new Promise((resolve,reject)=>{
                let img = new Image();
                img.src = url;
                img.onload=()=>{
                    console.log(`图片请求成功，此处进行通用操作`);
                    resolve(image);
                }
                img.onerror=(err)=>{
                    console.log(`失败，此处进行失败的通用操作`);
                    reject(err);
                }
            })
        }

imageAsync("url").then(()=>{
    console.log("加载成功");
}).catch((error)=>{
    console.log("加载失败");
})
```

5. 实现发布-订阅模式

```
class EventCenter{
  // 1. 定义事件容器，用来装事件数组
    let handlers = {}

  // 2. 添加事件方法，参数：事件名 事件方法
  addEventListener(type, handler) {
    // 创建新数组容器
    if (!this.handlers[type]) {
      this.handlers[type] = []
    }
    // 存入事件
    this.handlers[type].push(handler)
  }

  // 3. 触发事件，参数：事件名 事件参数
  dispatchEvent(type, params) {
    // 若没有注册该事件则抛出错误
    if (!this.handlers[type]) {
      return new Error('该事件未注册')
    }
    // 触发事件
    this.handlers[type].forEach(handler => {
      handler(...params)
    })
  }
```

```
  // 4. 事件移除，参数：事件名 要删除事件，若无第二个参数则删除该事件的订阅和发布
  removeEventListener(type, handler) {
    if (!this.handlers[type]) {
      return new Error('事件无效')
    }
    if (!handler) {
      // 移除事件
      delete this.handlers[type]
    } else {
      const index = this.handlers[type].findIndex(el => el === handler)
      if (index === -1) {
        return new Error('无该绑定事件')
      }
      // 移除事件
      this.handlers[type].splice(index, 1)
      if (this.handlers[type].length === 0) {
        delete this.handlers[type]
      }
    }
  }
}
```

6. 查找文章中出现频率最高的单词

```
function findMostWord(article) {
  // 合法性判断
  if (!article) return;
  // 参数处理
  article = article.trim().toLowerCase();
  let wordList = article.match(/[a-z]+/g),
    visited = [],
    maxNum = 0,
    maxWord = "";
  article = " " + wordList.join("  ") + " ";
  // 遍历判断单词出现次数
  wordList.forEach(function(item) {
    if (visited.indexOf(item) < 0) {
      // 加入 visited
      visited.push(item);
      let word = new RegExp(" " + item + " ", "g"),
        num = article.match(word).length;
      if (num > maxNum) {
        maxNum = num;
        maxWord = item;
      }
    }
  });
  return maxWord + "  " + maxNum;
}
```

7. 封装异步的fetch，使用async await方式来使用

```
(async () => {
    class HttpRequestUtil {
        async get(url) {
            const res = await fetch(url);
            const data = await res.json();
            return data;
        }
        async post(url, data) {
            const res = await fetch(url, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify(data)
            });
            const result = await res.json();
            return result;
        }
        async put(url, data) {
            const res = await fetch(url, {
                method: 'PUT',
                headers: {
                    'Content-Type': 'application/json'
                },
                data: JSON.stringify(data)
            });
            const result = await res.json();
            return result;
        }
        async delete(url, data) {
            const res = await fetch(url, {
                method: 'DELETE',
                headers: {
                    'Content-Type': 'application/json'
                },
                data: JSON.stringify(data)
            });
            const result = await res.json();
            return result;
        }
    }
    const httpRequestUtil = new HttpRequestUtil();
    const res = await httpRequestUtil.get('http://golderbrother.cn/');
    console.log(res);
})();
```

8. 实现prototype继承

所谓的原型链继承就是让新实例的原型等于父类的实例：

```
//父方法
function SupperFunction(flag1){
    this.flag1 = flag1;
}

//子方法
function SubFunction(flag2){
    this.flag2 = flag2;
}

//父实例
var superInstance = new SupperFunction(true);

//子继承父
SubFunction.prototype = superInstance;

//子实例
var subInstance = new SubFunction(false);
//子调用自己和父的属性
subInstance.flag1;   // true
subInstance.flag2;   // false
```

9. 实现双向数据绑定

```
let obj = {}
let input = document.getElementById('input')
let span = document.getElementById('span')
// 数据劫持
Object.defineProperty(obj, 'text', {
  configurable: true,
  enumerable: true,
  get() {
    console.log('获取数据了')
  },
  set(newVal) {
    console.log('数据更新了')
    input.value = newVal
    span.innerHTML = newVal
  }
})
// 输入监听
input.addEventListener('keyup', function(e) {
  obj.text = e.target.value
})
```

10. 实现简单路由

```
// hash路由
class Route{
  constructor(){
    // 路由存储对象
    this.routes = {}
    // 当前hash
    this.currentHash = ''
    // 绑定this，避免监听时this指向改变
    this.freshRoute = this.freshRoute.bind(this)
    // 监听
    window.addEventListener('load', this.freshRoute, false)
    window.addEventListener('hashchange', this.freshRoute, false)
  }
  // 存储
  storeRoute (path, cb) {
    this.routes[path] = cb || function () {}
  }
  // 更新
  freshRoute () {
    this.currentHash = location.hash.slice(1) || '/'
    this.routes[this.currentHash]()
  }
}
```

11. 实现斐波那契数列

```
// 递归
function fn (n){
    if(n==0) return 0
    if(n==1) return 1
    return fn(n-2)+fn(n-1)
}
// 优化
function fibonacci2(n) {
    const arr = [1, 1, 2];
    const arrLen = arr.length;

    if (n <= arrLen) {
        return arr[n];
    }

    for (let i = arrLen; i < n; i++) {
        arr.push(arr[i - 1] + arr[ i - 2]);
    }

    return arr[arr.length - 1];
}
// 非递归
function fn(n) {
    let pre1 = 1;
    let pre2 = 1;
    let current = 2;
```

```
    if (n <= 2) {
        return current;
    }

    for (let i = 2; i < n; i++) {
        pre1 = pre2;
        pre2 = current;
        current = pre1 + pre2;
    }

    return current;
}
```

12. 字符串出现的不重复最长长度

用一个滑动窗口装没有重复的字符，枚举字符记录最大值即可。用 map 维护字符的索引，遇到相同的字符，把左边界移动过去即可。挪动的过程中记录最大长度：

```
var lengthOfLongestSubstring = function (s) {
    let map = new Map();
    let i = -1
    let res = 0
    let n = s.length
    for (let j = 0; j < n; j++) {
        if (map.has(s[j])) {
            i = Math.max(i, map.get(s[j]))
        }
        res = Math.max(res, j - i)
        map.set(s[j], j)
    }
    return res
};
```

13. 使用 setTimeout 实现 setInterval

setInterval 的作用是每隔一段指定时间执行一个函数，但是这个执行不是真的到了时间立即执行，它真正的作用是每隔一段时间将事件加入事件队列中去，只有当当前的执行栈为空的时候，才能去从事件队列中取出事件执行。所以可能会出现这样的情况，就是当前执行栈执行的时间很长，导致事件队列里边积累多个定时器加入的事件，当执行栈结束的时候，这些事件会依次执行，因此就不能到间隔一段时间执行的效果。

针对 setInterval 的这个缺点，我们可以使用 setTimeout 递归调用来模拟 setInterval，这样我们就确保了只有一个事件结束了，我们才会触发下一个定时器事件，这样解决了 setInterval 的问题。

实现思路是使用递归函数，不断地去执行 setTimeout 从而达到 setInterval 的效果

```
function mySetInterval(fn, timeout) {
  // 控制器，控制定时器是否继续执行
  var timer = {
    flag: true
  };
  // 设置递归函数，模拟定时器执行。
  function interval() {
    if (timer.flag) {
      fn();
      setTimeout(interval, timeout);
    }
  }
  // 启动定时器
  setTimeout(interval, timeout);
  // 返回控制器
  return timer;
}
```

14. 实现 jsonp

```
// 动态的加载js文件
function addScript(src) {
  const script = document.createElement('script');
  script.src = src;
  script.type = "text/javascript";
  document.body.appendChild(script);
}
addScript("http://xxx.xxx.com/xxx.js?callback=handleRes");
// 设置一个全局的callback函数来接收回调结果
function handleRes(res) {
  console.log(res);
}
// 接口返回的数据格式
handleRes({a: 1, b: 2});
```

15. 判断对象是否存在循环引用

循环引用对象本来没有什么问题，但是序列化的时候就会发生问题，比如调用`JSON.stringify()`对该类对象进行序列化，就会报错: `Converting circular structure to JSON.`

下面方法可以用来判断一个对象中是否已存在循环引用：

```
const isCycleObject = (obj,parent) => {
    const parentArr = parent || [obj];
    for(let i in obj) {
        if(typeof obj[i] === 'object') {
            let flag = false;
            parentArr.forEach((pObj) => {
                if(pObj === obj[i]){
                    flag = true;
                }
            })
            if(flag) return true;
            flag = isCycleObject(obj[i],[...parentArr,obj[i]]);
            if(flag) return true;
        }
    }
    return false;
}


const a = 1;
const b = {a};
const c = {b};
const o = {d:{a:3},c}
o.c.b.aa = a;

console.log(isCycleObject(o)
```

查找有序二维数组的目标值：

```
var findNumberIn2DArray = function(matrix, target) {
    if (matrix == null || matrix.length == 0) {
        return false;
    }
    let row = 0;
    let column = matrix[0].length - 1;
    while (row < matrix.length && column >= 0) {
        if (matrix[row][column] == target) {
            return true;
        } else if (matrix[row][column] > target) {
            column--;
        } else {
            row++;
        }
    }
    return false;
};
```

二维数组斜向打印：

```
function printMatrix(arr){
  let m = arr.length, n = arr[0].length
    let res = []
```

```
  // 左上角，从0 到 n - 1 列进行打印
  for (let k = 0; k < n; k++) {
    for (let i = 0, j = k; i < m && j >= 0; i++, j--) {
      res.push(arr[i][j]);
    }
  }

  // 右下角，从1 到 n - 1 行进行打印
  for (let k = 1; k < m; k++) {
    for (let i = k, j = n - 1; i < m && j >= 0; i++, j--) {
      res.push(arr[i][j]);
    }
  }
  return res
}
```