### 保留有效小数

    //保留有效小数 方法一

    handleInput(value, name) {

      if (value != '') {

        if (value.indexOf('.') > -1) {

          this.formModel[name] = value.slice(0, value.indexOf('.') + 3)

        } else {

          this.formModel[name] = value

        }

      }

    },

    //保留有效小数 方法er

    HandleInput(val) {

      let number = Number(val)

      console.log(typeof number)

      this.formModel.RentPrice = number.toFixed(3)

    }

    /\*\*

     \* 计算两个日期相差的天数

     \* sDate1和sDate2是2006-12-18格式

     \*/

    DateDiff(sDate1, sDate2) {

      var aDate, oDate1, oDate2, iDays

      aDate = sDate1.split('-')

      oDate1 = new Date(aDate[0], aDate[1] - 1, aDate[2]) //2006-12-18格式

      aDate = sDate2.split('-')

      oDate2 = new Date(aDate[0], aDate[1] - 1, aDate[2])

      iDays = parseInt(Math.abs(oDate1 - oDate2) / 1000 / 60 / 60 / 24)

      return iDays

    },

### 深拷贝问题

    /\* 深拷贝 \*/

    function deepCopy(obj){

      // 如果是 值类型 或 null，则直接return

      if( !obj || typeof obj !== 'object') return obj

      let copy = {}

      if(Object.prototype.toString.call(obj) === '[object Array]'){

          copy = []

        }

      for (const key in obj) {

        // 如果key是对象的自有属性，非继承属性

        if (obj.hasOwnProperty(key)) {

          copy[key] = deepCopy(obj[key])

        }

      }

      return copy

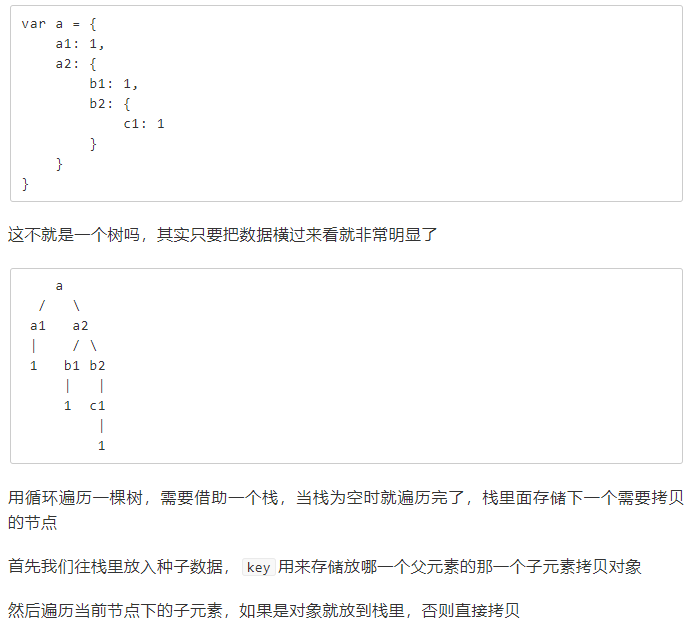
    }

#### 深拷贝爆栈问题

爆栈就是当一个数据深度过深的时候，栈溢出了。

// Maximum call stack size exceeded

解决问题的思路就是：在深度上存在溢出问题，但是在广度上不存在。



    function cloneLoop(x) {

      const root = {};

      debugger

      // 栈

      const loopList = [

        {

          parent: root,

          key: undefined,

          data: x,

        }

      ];

      while (loopList.length) {

        // 深度优先

        const node = loopList.pop();

        const parent = node.parent;

        const key = node.key;

        const data = node.data;

        // 初始化赋值目标，key为undefined则拷贝到父元素，否则拷贝到子元素

        let res = parent;

        if (typeof key !== 'undefined') {

          res = parent[key] = {};

        }

        for (let k in data) {

          if (data.hasOwnProperty(k)) {

            if (typeof data[k] === 'object') {

              // 下一次循环

              loopList.push({

                parent: res,

                key: k,

                data: data[k],

              });

            } else {

              res[k] = data[k];

            }

          }

        }

      }

      return root;

    }

#### 判断循环引用

*function* isHasCircle(*obj*) {

*let* hasCircle **=** false

*const* map **=** **new** *Map*()

*function* loop(*obj*) {

*const* keys **=** *Object*.keys(obj)

        keys.forEach(*key* *=>* {

*const* value **=** obj[key]

**if** (**typeof** value **==** 'object' **&&** value **!==** null) {

**if** (map.has(value)) {

              hasCircle **=** true

**return**

            } **else** {

              map.set(value)

              loop(value)

            }

          }

        })

      }

      loop(obj)

**return** hasCircle

    }

#### JSON中循环引用解决办法

##### 1 JSON.stringify出现 “Converting circular structure to JSON”

上面的错误信息告诉我们，对象中存在循环引用. 解决思路就是通过自定义stringify方法，设置一个全局缓存变量，stringify的第二个参数如果是function时，他会传入每个成员的键和值。使用返回值而不是原始值。如果此函数返回 undefined，则排除成员。根对象的键是一个空字符串：""。所以，便有了如下解决方案:

    //解决对象中存在循环引用问题

    circularReference (row) {

      let cache = [];

      let str = JSON.stringify(row, function (key, value) {

        if (typeof value === 'object' && value !== null) {

          if (cache.indexOf(value) !== -1) {

            // 移除

            return;

          }

          // 收集所有的值

          cache.push(value);

        }

        return value;

      });

      cache = null; // 清空变量，便于垃圾回收机制回收

      return str

    },

用法

        let str = this.circularReference(this.tableData)

        this.parentRow = JSON.parse(str)

##### 2暴力破解（保留引用关系）

    // 保持引用关系

    function cloneForce(x) {

      // =============

      const uniqueList = []; // 用来去重

      // =============

      let root = {};

      // 循环数组

      const loopList = [

        {

          parent: root,

          key: undefined,

          data: x,

        }

      ];

      while (loopList.length) {

        // 深度优先

        const node = loopList.pop();

        const parent = node.parent;

        const key = node.key;

        const data = node.data;

        // 初始化赋值目标，key为undefined则拷贝到父元素，否则拷贝到子元素

        let res = parent;

        if (typeof key !== 'undefined') {

          res = parent[key] = {};

        }

        // =============

        // 数据已经存在

        let uniqueData = find(uniqueList, data);

        if (uniqueData) {

          parent[key] = uniqueData.target;

          continue; // 中断本次循环

        }

        // 数据不存在

        // 保存源数据，在拷贝数据中对应的引用

        uniqueList.push({

          source: data,

          target: res,

        });

        for (let k in data) {

          if (data.hasOwnProperty(k)) {

            if (typeof data[k] === 'object') {

              // 下一次循环

              loopList.push({

                parent: res,

                key: k,

                data: data[k],

              });

            } else {

              res[k] = data[k];

            }

          }

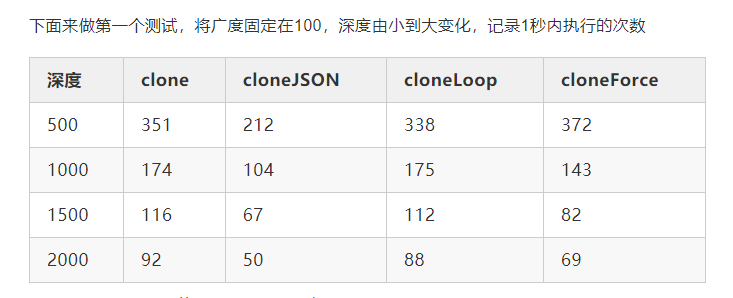
        }

      }

      return root;

}

##### 3性能





### 数组中重名检测

    function isRepeat(arr) {

      var hash = {};

      for (var i in arr) {

        if (hash[arr[i]]) return true;

        hash[arr[i]] = true;

      }

      return false;

    }

### 数组去重

#### 1. Array.filter + Array.indexOf

filter()方法：创建一个新数组，新数组中的元素是指定数组中符合某种条件的所有元素。如果没有符合条件的元素则返回空数组。

语法：array.filter(function(item,index,arr))

filter()不会对空数组进行检测。

filter()不会改变原始数组。

原理：返回item第一次出现的位置等于当前的index的元素。

let newArr = arr.filter((item, index) => arr.indexOf(item) === index);

#### Array.reduce + Array.includes

reduce()方法：接收一个函数作为累加器，数组中的每个值从左到右开始计算，最终计算为一个值。

语法：arr.reduce(function(total, currValue, currIndex, arr), initValue)

reduce()对于空数组是不会执行回调函数的。

total：必需。初始值, 或者计算结束后的返回值

currValue：必需。当前元素

currIndex：可选。当前元素的索引

arr：可选。当前数组对象。

initValue：可选。累加器初始值

* 一个空数组调用reduce()方法且没有提供初始值，会报错。
* 一个空数组调用reduce()方法且提供了初始值，将直接返回该初始值，不会调用callback函数。
* 非空数组调用reduce()提供初始值，则total将会等于初始值，且currValue从第一个元素开始；若没有提供初始值，则total会等于的第一个元素值，且currValue将会从第二个元素开始。

let newArr = arr.reduce((accu, cur) => {

    return accu.includes(cur) ? accu : accu.concat(cur);  // 1. 拼接方法

    // return accu.includes(cur) ? accu : [...accu, cur]; // 2. 扩展运算

}, []);

#### 3. new Set + 扩展运算符 || Array.from

ES6 提供了新的数据结构Set。类似于数组，但是成员的值都是唯一的，没有重复的值。

Set本身是一个构造函数，可以接受一个具有iterable接口数据结构作为参数（如数组，字符串），用来初始化。

let newArr = [...new Set(arr)];      // [1, 2, 4, null, "3", "abc", 3, 5]

let newArr = Array.from(new Set(arr));      // [1, 2, 4, null, "3", "abc", 3, 5]

-----------------------------------------------

let newStr = [...new Set('ababbc')].join('')  //  'abc'

#### 4. new Map

ES6 提供了新的数据结构Map。类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。

set方法设置键名key对应的键值为value，然后返回整个Map结构。如果key已经有值，则键值会被更新，否则就新生成该键。

get方法读取key对应的键值，如果找不到key，返回undefined。

has方法返回一个布尔值，表示某个键是否在当前Map对象之中。

let map = new Map();

let newStr = [];

for (let i = 0; i < arr.length; i++) {

    if (!map.has(arr[i])) {

        map.set(arr[i], true);

        newStr.push(arr[i]);

    }

}

console.log(newArr)  // [1, 2, 4, null, "3", "abc", 3, 5]

### Urlencode类型参数转obj

/\*\*

 \*  urlencodel类型参数转为obj

 \*/

export const getUrlObj =(url) =>{

    let newurl = url.split('?')

    if(newurl.length<2){

        return {}

      }

    let oldarray = newurl[1].split('&')

    let obj = {}

    for (let i = 0; i < oldarray.length; i++) {

      let arritem = oldarray[i]

      let item = arritem.split('=')

      obj[item[0]] = item[1]

    }

    return obj

}

### Promise实战

#### 超时控制

##### 背景

1众所周知，fetch请求是无法设置超时时间的，因此我们需要自己去模拟一个超时控制。

2转盘问题，一个抽奖转盘动画效果有5秒，但是一般来说向后端请求转盘结果只需要不到一秒，因此请求结果至少得等5秒才能展现给用户。

##### 问题分析

首先，超时控制比较简单，和Promise.race()的思想是类似，或者可以直接使用这个函数去解决。

然后，转盘问题如果要答好，需要考虑两种情况。

1转盘动画还未完成，请求结果已经拿到了，此时要等到动画完成再展示结果给用户。

2转盘动画完成了，请求结果还未拿到，此时需要等待结果返回（可以设置请求超时时间）。

所以，转盘问题更适合用Promise.all()来解决。

  /\*\*

 \* 辅助函数，封装一个延时promise

 \* @param {number} delay 延迟时间

 \* @returns {Promise<any>}

 \*/

  function sleep(times) {

    return new Promise((resolve, reject) => {

      setTimeout(() => {

        reject(new Error('超时了'))

      }, times);

    })

  }

/\*\*

 \* 将原promise包装成一个带超时控制的promise

 \* @param {()=>Promise<any>} requestFn 请求函数

 \* @param {number} timeout 最大超时时间

 \* @returns {Promise<any>}

 \*/

  function timeoutPromise(requestFn, timeout) {

    return Promise.race([requestFn(), sleep(timeout)])

  }

// ----------下面是测试用例------------

  // 模拟一个异步请求函数

  function request(delay) {

    return () => {

      return new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve('接口返回了')

        }, delay);

      })

    }

  }

  // 超时的例子

timeoutPromise(createRequest(2000), 1000).catch((error) =>

  console.error(error)

);

// 不超时的例子

timeoutPromise(createRequest(2000), 3000).then((res) => console.log(res));

#### 2. 取消重复请求

##### 背景

当用户频繁点击一个搜索Button时，会在短时间内发出大量的搜索请求，给服务器造成一定的压力，同时也会因请求响应的先后次序不同而导致渲染的数据与预期不符。这里，我们可以使用防抖来减小服务器压力，但是却没法很好地解决后面的问题。

##### 问题分析

这个问题的本质在于，同一类请求是有序发出的（根据按钮点击的次序），但是响应顺序却是无法预测的，我们通常只希望渲染最后一次发出请求响应的数据，而其他数据则丢弃。因此，我们需要丢弃（或不处理）除最后一次请求外的其他请求的响应数据。

    function CancelablePromise() {

      this.pendingPromise = null;

    }

    // 包装一个请求并取消重复请求

    CancelablePromise.prototype.request = function (requestFn) {

      if (this.pendingPromise) {

        this.cancel("取消重复请求");

      }

      const \_promise = new Promise((resolve, reject) => (this.reject = reject));

      //这里把promise的reject这个方法赋给了reject这个属性

      this.pendingPromise = Promise.race([requestFn(), \_promise]);

      return this.pendingPromise;

    };

    // 取消当前请求

    CancelablePromise.prototype.cancel = function (reason) {

      this.reject(new Error(reason));

      //这里执行了这个reject方法，上面的\_promise就会返回值

      // console.log(this.reject);

      this.pendingPromise = null;

    };

    // ----------下面是测试用例------------

    // 模拟一个异步请求函数

    function createRequest(delay) {

      return () =>

        new Promise((resolve) => {

          setTimeout(() => {

            resolve("done");

          }, delay);

        });

    }

    const cancelPromise = new CancelablePromise();

    // 前四个请求将被自动取消

    for (let i = 0; i < 5; i++) {

      cancelPromise

        .request(createRequest(2000))

        .then((res) => console.log(res)) // 最后一个 done

        .catch((err) => console.error(err)); // 前四个 error: 取消重复请求

    }

#### 3限制并发请求数

##### 背景

一般来说，我们不会刻意去控制请求的并发。只有在一些场景下可能会用到，比如，收集用户的批量操作（每个操作对应一次请求），待用户操作完成后一次性发出。另外，为了减小服务器的压力，我们还会**限制并发数**。

##### 问题分析

看上去，Promise.allSettled很适合应对这样的场景，但是稍微想一下就能发现，它能控制的粒度还是太粗了。首先，它必须等待所有Promise都resolve或reject，其次，如果有并发限制的话用它来做还需要分批请求，实际效率也会比较低，短木板效应很明显

   /\*\*

 \* 并发请求限制并发数

 \* @param {()=>Promise<any> []} requestFns 并发请求函数数组

 \* @param {numer} limit 限制最大并发数

 \*/

*function* concurrentRequest(*requestFns*, *limit*) {

  // 递归函数

*function* recursion(*requestFn*) {

      // 前面的请求状态返回后下一个请求才会开始

    requestFn().finally(() *=>* {

**if** (\_requestFns.length **>** 0) {

        recursion(\_requestFns.shift());

      }

    });

  }

  // 这步感觉没什么必要，可能作用就是用一个新的引用地址吧，不是很懂

*const* \_requestFns **=** [**...**requestFns];

  // 限制最大并发量

**for** (*let* i **=** 0; i **<** limit **&&** \_requestFns.length **>** 0; i**++**) {

    recursion(\_requestFns.shift());

  }

}

// ----------下面是测试用例------------

// 模拟一个异步请求函数

*function* createRequest(*delay*,*i*) {

**return** () *=>*

**new** *Promise*((*resolve*) *=>* {

      setTimeout(() *=>* {

        resolve("done:"**+**i);

      }, delay);

    }).then(*r=>*{

        console.log(r);

    })

}

*const* requestFns **=** [];

**for** (*let* i **=** 0; i **<** 10; i**++**) {

    // 这里面的参数i是为了打印查看区分具体请求的

  requestFns.push(createRequest(1000,i));

}

concurrentRequest(requestFns, 3);

#### 4串行化的三种实现方式

使用串行化的常见场景，请求之间有依赖关系或时序关系，如红绿灯

/\*\*

 \* 串行化的三种实现

 \*\*/

// 法一，递归法

*function* runPromiseInSeq1(*requestFns*) {

*function* recursion(*requestFns*) {

**if** (requestFns.length **===** 0) **return**;

    requestFns

      .shift()()

      .finally(() *=>* recursion(requestFns));

  }

*const* \_requestFns **=** [**...**requestFns];

  recursion(\_requestFns);

}

// 法二：迭代法

**async** *function* runPromiseInSeq2(*requestFns*) {

**for** (*const* requestFn **of** requestFns) {

**await** requestFn();

  }

}

// 法三：reduce

*function* runPromiseInSeq3(*requestFns*) {

  requestFns.reduce((*pre*, *cur*) *=>* pre.finally(() *=>* cur()), *Promise*.resolve());

}

// 模拟一个异步请求函数

*function* createRequest(*delay*) {

**return** () *=>*

**new** *Promise*((*resolve*) *=>* {

      setTimeout(() *=>* {

        resolve(delay);

      }, delay);

    }).then((*res*) *=>* {

      console.log(res);

    });

}

// 执行顺序从左至右

*const* requestFns **=** [

  createRequest(3000),

  createRequest(2000),

  createRequest(1000),

];

// 串行调用

runPromiseInSeq1(requestFns);

// runPromiseInSeq2(requestFns);

// runPromiseInSeq3(requestFns);

#### 5. 20行最简异步链式调用

这里模拟了Promise的异步链式调用，代码出处见文章。

*function* Promise(*fn*) {

  this.cbs **=** [];

*const* resolve **=** (*value*) *=>* {

    setTimeout(() *=>* {

      this.data **=** value;

      this.cbs.forEach((*cb*) *=>* cb(value));

    });

  }

  fn(resolve);

}

*Promise*.prototype.then **=** *function* (*onResolved*) {

**return** **new** *Promise*((*resolve*) *=>* {

    this.cbs.push(() *=>* {

*const* res **=** onResolved(this.data);

**if** (res **instanceof** Promise) {

        res.then(resolve);

      } **else** {

        resolve(res);

      }

    });

  });

};

### **随机字符串**

const str = Math.random().toString(36).substr(2, 10);  
console.log(str);   // 'w5jetivt7e'

先是 Math.random() 生成 [0, 1) 的数，也就是 0.123312、0.982931之类的，然后调用 number 的 toString方法将其转换成36进制的，按照MDN的说法，36进制的转换应该是包含了字母 a~z 和 数字0~9的，因为这样生成的是 0.89kjna21sa 类似这样的，所以要截取一下小数部分，即从索引 2 开始截取10个字符就是我们想要的随机字符串了

很多开源库都使用此方式为DOM元素创建随机ID。

### 下载实现

/\*\*

\* 文件流 下载文件

\*/

export const createAFordownLoad = (fileName, steam) => {

if (!steam) {

return

}

let url = window.URL.createObjectURL(new Blob([steam]))

let link = document.createElement('a')

link.style.display = 'none'

link.href = url

link.setAttribute('download', decodeURI(fileName))

document.body.appendChild(link)

link.click()

}

#### **Content-Disposition**

Content-disposition其实可以控制用户请求所得的内容存为一个文件的时候提供一个默认的文件名，文件直接在浏览器上显示或者在访问时弹出文件下载对话框。

****格式说明：**** content-disposition = "Content-Disposition" ":" disposition-type \*( ";" disposition-parm )

****字段说明：****Content-Disposition为属性名disposition-type是以什么方式下载，如attachment为以附件方式下载。disposition-parm为默认保存时的文件名服务端向客户端游览器发送文件时，如果是浏览器支持的文件类型，一般会默认使用浏览器打开，比如txt、jpg等，会直接在浏览器中显示，如果需要提示用户保存，就要利用Content-Disposition进行一下处理，关键在于一定要加上attachment

Response.AppendHeader("Content-Disposition","attachment;filename=FileName.txt");

#### 下载同域名下的资源

通过配置a标签的download属性，这样就可以直接调用浏览器的下载功能，而且还可以通过传参给download来实现文件的重命名功能。

    <a href="download.jpg" download="othername.jpg">下载</a>

但是这有一个问题：

chrome 69.0.3497.92 中已经严格遵循同源策略的限制，如果加载了非同源的内容，download 属性将失效，等效导航功能。

#### 下载网络资源（非同域名下）

    function download2 (url = imgurl, name = '未命名') {

      name = name || url

      // fetch抓取图片数据

      fetch(url).then(response => {

        console.log(response);

        if (response.status == 200)

          // 返回的.blob()为promise，然后生成了blob对象，此方法获得的blob对象包含了数据类型，十分方便

          // blob() 在response的原型上

          return response.blob()

        throw new Error(`status: ${response.status}.`)

      }).then(blob => {

        // 获取到blob对象

        console.log(blob);

        downloadFile(name, blob)

      }).catch(error => {

        console.log("failed. cause:", error)

      })

    }

    function downloadFile (fileName, blob) {

      const anchor = document.createElement("a")

      // 创建指向blob对象地址

      const src = URL.createObjectURL(blob)

      anchor.download = fileName

      anchor.href = src

      anchor.click()

    }

### base64转blob

    base64ToBlob (code) {

      const parts = code.split(";base64,");

      const contentType = parts[0].split(":")[1];

      const raw = window.atob(parts[1]);

      const rawLength = raw.length;

      const uInt8Array = new Uint8Array(rawLength);

      for (let i = 0; i < rawLength; ++i) {

        uInt8Array[i] = raw.charCodeAt(i);

      }

      return new Blob([uInt8Array], {

        type: contentType,

      });

    },

### Canvas绘制 并转化为png挂载在dom上

    init () {

      const canvas = document.createElement("canvas");

      canvas.id = "posterCancas";

      canvas.width = 375;

      canvas.height = 621;

      const context = canvas.getContext("2d");

      context.fillStyle = "#DBA293";

      // context.fill();

      const myImage = new Image();

      myImage.src = "//yun.baoxiaohe.com/activity/inviteActivity/poster.webp"; // 背景图片

      myImage.crossOrigin = "Anonymous";

      myImage.onload = () => {

        context.drawImage(myImage, 0, 0, 375, 621);

        context.font = "13px serif";

        context.fillText("你的好友姜姜", 85, 520);

        const myImage2 = new Image();

        myImage2.src = this.qrcodeUrl;

        console.log(this.qrcodeUrl);

        myImage2.crossOrigin = "Anonymous";

        myImage2.onload = () => {

          context.drawImage(myImage2, 80, 190, 220, 220);

          const base64 = canvas.toDataURL("image/png");

          // const blob = this.base64ToBlob(base64);

          console.log(base64);

          this.canvansUrl = base64;

        };

        // const base64 = canvas.toDataURL("image/png");

        // const aLink = document.createElement("a");

        // const blob = this.base64ToBlob(base64);

        // // const blob = base64ToBlob(this.qrcode); // new Blob([content]);

        // const evt = document.createEvent("HTMLEvents");

        // evt.initEvent("click", true, true); // initEvent 不加后两个参数在FF下会报错  事件类型，是否冒泡，是否阻止浏览器的默认行为

        // aLink.download = Date.now();

        // aLink.href = URL.createObjectURL(blob);

        // aLink.click();

      };

      document.body.appendChild(canvas);

    },