

1. 异步并发调度

```
class Scheduler {
  constructor(limit) {
    this.limit = limit
    this.number = 0
    this.queue = []
  }
  addTask(timeout, str) {
    this.queue.push(
      ()=>{
        return new Promise((resolve,reject)=>{
          setTimeout(()=>{
            resolve(str)
          },timeout)
        })
      })
    console.log(this.queue,'this.queue')
  }
  start() {
    console.log(this.number,'number')
    if (this.number < this.limit&&this.queue.length)
      var run = this.queue.shift()
      this.number++
      run().then((str)=>{
        console.log(str,'str')
        this.number--
        this.start()
      })
      this.start()
    }
  }
}
let sch=new Scheduler(2)
```

2. 深拷贝

```

function deepClone(obj) {
  //排除基本数据类型 和函数 (可以不用深拷贝 不会发生变化)
  if (typeof obj !== 'object' || obj === null) { //null的类型为object
    return obj //null 1 'a' undefined
  }
  //缓存
  if (!deepClone.cached) {
    deepClone.cached = new Map()
  }
  //判断是否缓存过
  if (deepClone.cached.has(obj)) {
    return deepClone.cached.get(obj)
  }
  let temp
  if (obj instanceof Map) {
    temp = new Map()
    deepClone.cached.set(obj, temp)
    for (let [key, value] of obj) {
      temp.set(deepClone(key), deepClone(value)) //map的key也可
    }
  } else if (obj instanceof Set) {
    temp = new Set()
    deepClone.cached.set(obj, temp)
    for (let value of obj) {
      temp.add(deepClone(value))
    }
  } else if (obj instanceof RegExp) {
    temp = new RegExp(obj)
    deepClone.cached.set(obj, temp)
  } else {
    //对象或数组
    temp = new obj.constructor()
    deepClone.cached.set(obj, temp)
    for (let key in obj) {
      temp[key] = deepClone(obj[key])
    }
  }

  return temp
}

```

3. 节流

```

//3.第一次触发 最后一次触发

const __throttle=(fun,delay)=>{
  let timer=null
  let pre=0
  return function(...args){
    let cur=new Date().valueOf()
    if(cur-pre>delay){ //第一次执行,同时清除
      if(timer){
        clearTimeout(timer)
        timer=null
      }
      fun.apply(this,args)
      pre=cur
    }
    if(!timer){
      timer=setTimeout(()=>{
        pre=new Date().valueOf()
        timer=null
        fun.apply(this,args)
      },delay)
    }
  }
}

```

4.防抖

```
//时间响应函数在一段时间后才执行
const debounce=(fun,delay,immediate = false)=>{
  let timer=null
  let result
  return function(...args){
    timer && clearTimeout(timer)
    if(immediate){
      //立即执行
      let isCalledNow=!timer
      timer=setTimeout(()=>{
        timer=null
      },delay)
      if(isCalledNow) result=fun.apply(this,args)
    }else{
      timer=setTimeout(()=>{
        result=fun.apply(this,args)
        timer=null
      },delay)
    }
    return result
  }
}
```

5. 柯里化

```
const currying = (fn) => {
  var arg = []
  return function curry(...args) {
    if (args.length === 0) {
      return fn(...arg)
    } else {
      arg.push(...args)
      return curry
    }
  }
}
```

6. promise.all

```
static all(promises) {
  return new myPromise((resolve, reject) => {
    //参数校验
    if (Array.isArray(promises)) {
      let result = []
      let count = 0
      //传入的是一个空对象时，返回已完成
      if (promises.length === 0) {
        return resolve(promises)
      }
      promises.forEach((item, index) => {
        //判断参数是否为promise
        if (item instanceof myPromise) {
          myPromise.resolve(item).then(value => {
            count++
            //每个promise结果存入数组
            result[index] = value
            count === promises.length && resolve(result)
          }, reason => {
            reject(reason)
          })
        } else { //参数中有非promise的值，原样返回到数组
          count++
          result[index] = item
          count === promises.length && resolve(result)
        }
      })
    } else {
      return reject(new TypeError('Argument is not iterable'))
    }
  })
}
```

7. promise.resolve

```

//要将value解析为promise对象的值
static resolve(value){
    if(value instanceof myPromise){
        return value
    }else if(value instanceof Object && 'then' in value){
        //如果这个值带有then方法 返回的promise会跟随then状态
        return new myPromise((resolve,reject)=>{
            value.then(resolve,reject)
        })
    }
    //其他情况下 都执行resolve
    return new myPromise((resolve,reject)=>{
        resolve(value)
    })
}

```

8. call、apply、bind

```

//this的显示调用：相当于在gt中加入了一个person函数，this则默认指向gt
Function.prototype._call = function (obj=window, ...args) {
    //此时的this指向Function
    obj.fun = this
    const result= obj.fun(...args)
    delete obj.fun
    return result
}
person._call(gt,123,12333)

Function.prototype._apply = function (obj=window, ...args) {
    //此时的this指向Function
    obj.fun = this
    const result= obj.fun(...args)
    delete obj.fun
    return result
}
person._apply(gt,[123,12333])

Function.prototype._bind = function (obj = window, ...args) {
    const thisFn = this //保存当前调用的函数
    let thisFnBind = function (...secondArgs) {
        const isNew = this instanceof thisFnBind
        const thisArg = isNew ? this : obj
        return thisFn.call(thisArg, ...args, ...secondArgs)
    }
    thisFnBind.prototype=Object.create(thisFn.prototype)
    return thisFnBind
}

const per = person._bind(gt, 1, 2, 3)
per(4, 5)

```

9.instanceOf 实现

```

//递归实现
function myInstanceOf(obj,fn){
    if(!obj || typeof obj!=='object'){
        return false
    }else{
        if(obj.__proto__===fn.prototype){ //[].__proto__ === Array.prototype
            return true
        } else{
            return myInstanceOf(obj.__proto__)
        }
    }
}

let arr =[]
console.log(myInstanceOf(arr , Array))

```

10. new实现

```
function _new(fun, ...args) {
  let obj = Object.create({})
  Object.setPrototypeOf(obj, fun.prototype)
  let res = fun.apply(obj, args)
  return res instanceof Object ? res : obj
}
```

11. 环形引用对象序列化

```
let pathRef = new WeakMap()

function isObject(obj) {
  return obj && typeof obj === 'object' && !(obj instanceof Function)
  && !(obj instanceof Date) && !(obj instanceof Map) && !(obj instanceof Set)
}

function formatCircleRef(obj, jsonPath) {
  if (!jsonPath) {
    console.warn('jsonPath不存在')
  }
  if (isObject(obj)) {
    // 存储对象的路径
    let path = pathRef.get(obj)
    if (path) {
      return path
    } else {
      pathRef.set(obj, jsonPath)
    }
    // 遍历对象及子属性
    if (Array.isArray(obj)) {
      // 数组
      return obj.map((item, index) => {
        return formatCircleRef(item, `${jsonPath}["${index}"]`)
      })
    } else {
      // 对象
      const temp = {}
      Object.keys(obj).forEach(key => {
        let value = obj[key]
        temp[key] = formatCircleRef(value, `${jsonPath}["${key}"]`)
      })
      return temp
    }
  } else {
    return obj
  }
}
```