# 实现Promise

function MyPromise (excutor) {

  this.$$status = 'pending'

  this.$$value = undefined

  this.resolveCb = null

  this.rejectCb = null

  this.thenResolveCb = null

  this.thenRejectCb = null

  excutor(this.resolve.bind(this), this.resolve.bind(this))

}

MyPromise.prototype.resolve = function (val) {

  Object.defineProperties(this, {

    '$$status': {

      writable: false,

      value: 'fulfilled',

    },

    '$$value': {

      writable: false,

      value: val,

    },

  })

  if (!this.resolveCb) return

  try {

    const result = this.rejectCb(this.$$value)

    this.thenResolveCb(result)

  } catch (e) {

    this.thenRejectCb(e)

  }

}

MyPromise.prototype.reject = function (val) {

  Object.defineProperties(this, {

    '$$status': {

      writable: false,

      value: 'rejected',

    },

    '$$value': {

      writable: false,

      value: val,

    },

  })

  if (!this.rejectCb) return

  try {

    const result = this.rejectCb(this.$$value)

    this.thenResolveCb(result)

  } catch (e) {

    this.thenRejectCb(e)

  }

}

MyPromise.prototype.then = function (full, fail) {

  let newPromise = new MyPromise(() => {})

  this.resolveCb = full

  this.rejectCb = fail

  this.thenResolveCb = newPromise.resolve.bind(newPromise)

  this.thenRejectCb = newPromise.reject.bind(newPromise)

  return newPromise

}

# 实现Promise.all

Promise.all = function(promises) {

  let count = 0,

    result = []

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

    promises.forEach((promise, index) => {

      Promise.resolve(promise).then(

        (res) => {

          count++

*// 用 index 而不用 push 能保证返回值的顺序对应 promsies数组*

          result[index] = res

          if (count === promises.length) {

            resolve(result)

          }

        },

        (e) => {

          reject(e)

        }

      )

    })

  })

}

# 实现Promise.allSettled

Promise.allSettled = function(promises) {

  const result = []

  let count = 0

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

    promises.forEach((promise, index) => {

      Promise.resolve(promise).then(

        (res) => {

          count++

          result[index] = { status: 'fullfilled', value: res }

        },

        (e) => {

          count++

          result[index] = { status: 'rejected', reason: e }

        }

      ).finally(() => {

        count++

        if (count === promises.length) {

          resolve(result)

        }

      })

    })

  })

}

# 实现async...await

*/\*\**

*\* 该函数的功能是 传入一个生成器函数，返回一个能自执行它的函数*

*\* 该函数能做到能等待 generator中的每一个yield执行完，再执行下一个yield*

*\**@param*generator 需要被自执行的 Generator函数*

*\*/*

function asyncToGenerator(generator) {

  return function () {

    const gen = generator.apply(this, arguments)

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

*/\*\**

*\**

*\**@param*way 两种 next 和 throw*

*\**@param*arg 需要传到下一个 yield的值*

*\*/*

      function step(way, arg) {

        let result

        try {

          result = gen[way](arg)

        } catch(e) {

          reject(e)

        }

        const { value, done } = result

        if (done) {

          return resolve(value)

        } else {

*/\*\**

*\* 当 value为 Promise对象时（即 yield后返回的的是Promise对象）*

*\* Promise.resolve() 会原样返回该 Promise对象，所以 then 会在 val执行完之后执行*

*\* 从而达到了 await 的效果*

*\*/*

          return Promise.resolve(value).then(

            (res) => { step("next", res) },

            (e) => { step("throw", e) }

          )

        }

      }

      step("next")

    })

  }

}

# 实现call

Function.prototype.call2 = function (target, ...args) {

  let copyFn = this,

      name = Symbol('cache'),

      \_target = target || window

  \_target[name] = copyFn

  const result = \_target[name](...args)

  delete \_target[name]

  return result

}

# 实现apply

Function.prototype.apply2 = function (target, args) {

  let fn = this,

    name = Symbol("copyFn")

  target = target || window

  target[name] = fn

  const result = target[name](...(args || []))

  delete target[name]

  return result

}

# 实现bind

Function.prototype.bind = function (target, ...arg1) {

  const self = this

  if (typeof self !== 'function') {

    throw new Error(`${this.name} must be a function`)

  }

  const bindFn = function(...arg2) {

    let context = target

    if (this instanceof binFn) {

      context = this

    }

    return self.apply(context, [...arg1, ...arg2])

  }

  bindFn.prototype = self.prototype

  return bindFn

}

# 实现instanceof

*// 实现 1*

function instance\_of(obj, constructor) {

  let curProto = obj

  while (curProto = curProto.\_\_proto\_\_) {

    if (curProto.constructor === constructor) {

      return true

    }

  }

  return false

}

*// 实现 2*

function instance\_of2(obj, constructor) {

  let proto = constructor.prototype

  let curProto = obj

  while (curProto = curProto.\_\_proto\_\_) {

    if ( curProto === proto ) return true

  }

  return false

}

# 实现new

function newFn(constructor, ...args) {

  let instance,

    proto = null

  if (typeof constructor === 'object') {

    proto = constructor.prototype

  }

  instance = Object.create(proto)

  const res = constructor.apply(instance, args)

  if ( (typeof res === 'object' && res !== null) || typeof res === 'function' ) {

    return res

  }

  return instance

}

# 防抖和节流

*/\*\**

*\* 防抖*

*\*/*

function debounce (fn, delay) {

  let timeout

  return function (...args) {

    clearTimeout(timeout)

    timeout = setTimeout(fn.bind(this), delay, args)

  }

}

*/\*\**

*\* 节流*

*\*/*

function throttle(fn, delay) {

  let isRunning,

    timeout

  return function (...args) {

    if (isRunning) return

    if (timeout) {

      clearTimeout(timeout)

    }

    isRunning = true

    timeout = setTimeout( () => {

      fn.apply(this, args)

      isRunning = false

    }, delay )

  }

}

# 函数柯里化

function currify (fn, ...outerArgs) {

  return function (...innerArgs) {

    fn.apply(this, [...outerArgs, ...innerArgs])

  }

}

# 深拷贝

function deepCopy(target, cache = new WeakMap()) {

  let ob

*/\*\**

*\* 如果是非对象类型 则直接返回原值   null instanceof Object 为false*

*\*/*

  if (!target instanceof Object) return target

  if (cache.get(target)) return cache.get(target);

*/\*\**

*\* 处理 target为函数的情况*

*\*/*

  if (target instanceof Function) {

    return function () {

      target.call(this, ...arguments)

    }

  }

*// 处理一些基本引用类型的复制*

  if (target instanceof Date) {

    return new Date(target)

  }

  if (target instanceof RegExp) {

    return new RegExp(target.source, target.flags)

  }

  ob = Array.isArray(target) ? [] : Object.create(target.\_\_proto\_\_)

*/\*\**

*\* 把 ob 自身存储起来，key为 target（如果 target中有对自己的引用，则 key 会一直为 target）*

*\* 用于在 target有对自身的引用属性时，形成无限循环的出口*

*\*/*

  cache.set(target, ob);

  for (let key in target) {

    if (target[key] instanceof Object) {

      ob[key] = deepCopy(target[key])

    } else {

      ob[key] = target[key]

    }

  }

  return ob

}

# 实现链式调用

let stack = []

function next() {

  let fn = stack.shift()

  fn && (typeof fn === 'function') && fn()

}

function Man(name) {

  stack.push(() => {

    console.log(name)

    next()

  })

  setTimeout(next, 0)

}

Man.prototype.eat = function(food) {

  stack.push(() => {

    console.log(food)

    next()

  })

  return this

}

Man.prototype.sleep = function(delay) {

  stack.push(() => {

    setTimeout(() => {

      console.log('sleep')

      next()

    }, delay)

  })

  return this

}

function LazyMan(name) {

  const p = new Man(name)

  return p

}

LazyMan('Hank').eat('apple').sleep(6000).eat('banana')

# 不同继承模式

*// 原型链 （所有的 Sub实例的原型都是同一个 Sup对象，对原型的更改会互相影响）*

function Sup() {}

function Sub() {}

Sub.prototype = new Sup()

*// 盗用构造函数 （无法继承原型上的方法）*

function Sup() {}

function Sub() {

  Sup.apply(this, arguments)

}

*// 组合继承 （父类构造函数会被调用两次, 并且该实现会使 sub实例 和 sub的prototype中含有相同的实例属性）*

function Sup() {}

function Sub() {

  Sup.apply(this, arguments)

}

Sub.prototype = new Sup()

*// 原型式继承 (即 Object.create())*

function object(origin) {

  function F() {}

  F.prototype = origin

  return new F()

}

*// 寄生式继承 (先使用原型式继承创建一个对象，再在对象上做扩展操作)*

function createAnother(origin) {

  let o = object(origin)

  o.addAct = function() { console.log('添加方法') }

  return o

}

*// 寄生式组合继承 (将组合继承中的第一次调用父构造函数替换成了 inheritPrototype(Sub, Sup) )*

function Sup() {}

function Sub() {

  Sub.apply(this, arguments)

}

inheritPrototype(Sub, Sup)

function inheritPrototype(sub, sup) {

  const prototype = object(sup.prototype)

  prototype.constructor = sub

  sub.prototype = prototype

}

# 自定义事件并触发

*// 创建自定义事件对象*

const myEvent = document.createEvent('CustomEvent')

*// 初始化自定义对象*

myEvent.initCustomEvent('my-event', true, true, 'Hello World')

*// 自定义事件处理器*

document.addEventListener('my-event', function(e) {

    console.log(e.detail) *// Hello World*

})

*// 触发自定义事件对象*

document.dispatchEvent(myEvent)