数据结构

基本数据类型：string boolean undefined null number symbol bigint

引用数据类型：object 。 包括 普通对象{} 数组对象[ ] 函数对象 Function 正则对象 RegEpx 日期对象 Date 数学对象 Math

区别：

- 基本数据类型存在栈中，占据空间、大小固定
- 引用数据类型存在堆中，占据大小 空间 不固定 ，在栈中存储指针，该指针指向堆中该实体的起始位置，堆中存储实体
- 基本数据类型是**值传递** ，拷贝时会创建一个完全相等的变量，在栈中开辟一块内存存储原变量的副本
- 引用数据类型是**引用传递**，拷贝时会在栈中创建一个指针指向原有变量（浅拷贝）

Number和bigInt

- number类型的范围是 2的 -53次方-1 ----- 2的正53次方
- 当超过了number的范围时，会造成数值失真，则需要使用bigInt来进行解决
- 可以用在一个整数字面量后面加 `n` 的方式定义一个 `BigInt` ，如：`10n`，或者调用函数 `BigInt()`（但不包含 `new` 运算符）并传递一个整数值或字符串值。

判断数据类型：

- **typeof** 用于区分基本数据类型，**无法区分null** 和引用数据类型 但是可以区分 **function**

  js的第一个版本中，单个值会在栈中占用32位的存储单元，每个存储单元包含一个**1~3bit的类型标签**，类型标签存储在每个单元的低位

  typeof通过类型标签进行判断数据类型

  ```
  000: object   - 当前存储的数据指向一个对象。
    1: int      - 当前存储的数据是一个 31 位的有符号整数。
  010: double   - 当前存储的数据指向一个双精度的浮点数。
  100: string   - 当前存储的数据指向一个字符串。
  110: boolean  - 当前存储的数据是布尔值。
  ```

  注意： 其中有两个特殊类型： undefined的值是(-2)的30次方 null 是NULL机器码 0-31位均为0 所以typeof null 是Object

  - **instanceof** 用于区分引用数据类型，不能检测基本数据类型

> object instanceof constructor 用来比较一个对象的是否为某一个构造函数的实例

```
 function myInstanceof (obj, constructor) {
    let proto = constructor.prototype
    let object = Object.getPrototypeOf(obj)
    while (1) {
        if (object === null) return false
        if (proto === obj) return true
        object = Object.getPrototypeOf(object)
    }
}
```

```
const myInstanceofEs6 = (obj, constructor) => {
    obj == null ? false : obj == constructor ? true :
myInstanceofEs6(Object.getPrototypeOf(myInstanceofEs6), constructor)
}
```

- **constructor**

```
/**
* x是否为type类型
**/
const isXType = (x,type) => x.constructor.toString().indexOf(type) > -1
```

- **Object.prototype.toString.call(val)** :将val参数转化为字符串

  对象会使用ToPrimitive ，首先（通过内部操作 DefaultValue）检查该值是否有valueOf()方法。如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。

- **Reflect.apply(Object.prototype.toString,val,[])**

Object.is() 与比较操作符 “===”、“==” 的区别?

- 使用双等号（==）进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。
- 使用三等号（===）进行相等判断时，如果两边的类型不一致时，不会做强制类型转换，直接返回 false。
- 使用 Object.is 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 -0 和 +0 不再相等，两个 NaN 是相等的。

0.1+0.2为什么不等于0.3

在 JavaScript 中只有一种数字类型：Number，它的实现遵循IEEE 754标准，使用64位固定长度来表示，也就是标准的double双精度浮点数。在二进制科学表示法中，双精度浮点数的小数部分最多只能保留52位，再加上前面的1，其实就是保留53位有效数字，剩余的需要舍去，遵从“0舍1入”的原则。

根据这个原则，0.1和0.2的二进制数相加，再转化为十进制数就是：`0.30000000000000004`。

如何判断让他们相等

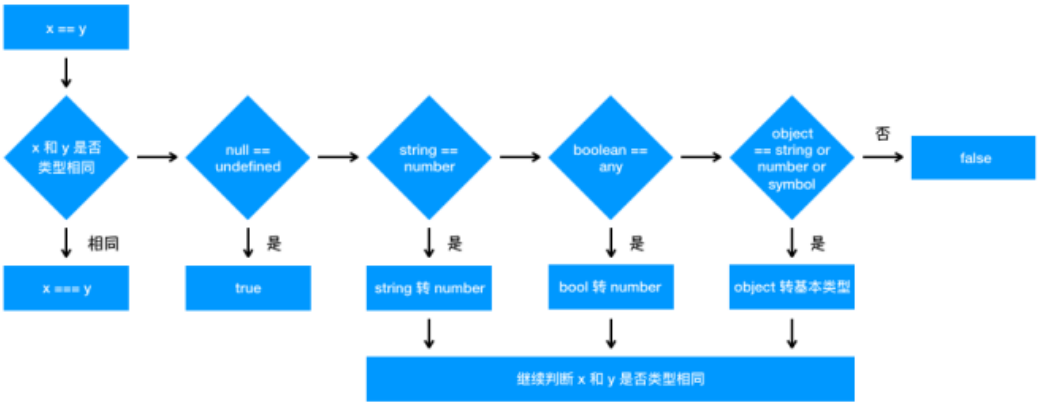
提供了`Number.EPSILON`属性，而它的值就是2-52次方 （机械精度）

```
function numberepsilon(arg1,arg2){
  return Math.abs(arg1 - arg2) < Number.EPSILON;
}

console.log(numberepsilon(0.1 + 0.2, 0.3)); // true
```

== 操作符的强制类型转换规则

判断流程如下：

1. 首先会判断两者类型是否**相同**，相同的话就比较两者的大小；
2. 类型不相同的话，就会进行类型转换；
3. 会先判断是否在对比 `null` 和 `undefined`，是的话就会返回 `true`
4. 判断两者类型是否为 `string` 和 `number`，是的话就会将字符串转换为 `number`
5. 判断其中一方是否为 `boolean`，是的话就会把 `boolean` 转为 `number` 再进行判断
6. 判断其中一方是否为 `object` 且另一方为 `string`、`number` 或者 `symbol`，是的话就会把 `object` 转为原始类型再进行判断



数组 Array

数组方法

改变数组的方法：

- push：**返回值：**数组新长度
- pop：**返回值：**被删除的项
- shift：删除第一个元素　**返回值：**返回被删除的项
- unshift :向数组头部添加一个元素　**返回值：**返回数组新长度
- reverse：反转数组
- splice：　**返回值：**从数组中被删除的元素，三参：起始位置值，删除数量，添加的新元素
- sort：对数组新元素进行排序
- fill：批量复制。三参：填充的值，开始位置，结束位置
- copyWithin：填充数组。三参：复制到指定目标索引位置，元素复制的起始位置，停止复制的索引位置

判断是否为数组

- instanceof
- Array.isArray()
- Object.prototypeof.toString.call()
- constructor ——属性返回数组的构造函数

```
const arr = []
console.log(arr.constructor)  //[Function: Array]
```

数组去重

1. 使用set去重

> 不能去重 {} 对象

```
function unique (arr) {
    return Array.from(new Set(arr))
   // return [...new Set(arr)]
}
```

2. 双重for循环，使用splice去重

> 不能去重NaN和{}

```
function unique (arr) {
    for (let i = 0; i < arr.length; i++) {
        for (let j = i + 1; j < arr.length; j++) {
            if (arr[i] === arr[j]) {
                arr.splice(j, 1)      //前一个数组元素等于后一个，则删除后一个
                j-- // 下标减小一位
            }
        }
    }
    return arr
}
```

3. 使用indexOf去重　　创建新数组，判断新数组中是否有该元素，没有则存入新数组中

> 不能去重 NaN 和 {}

```
function unique (arr) {
    const res = []        //创建一个新数组
    if (!Array.isArray(arr)) {
        return
    }
    for (let i = 0; i < arr.length; i++) {
        if (res.indexOf(arr[i]) === -1) {          //判断新数组中是否存在该元素，如果不存在则存入数组
            res.push(arr[i])
        }
    }
    return res
}
```

4. 使用sort去重　　对数组进行排列，如果后一个和前一个不相等则存入新数组中

> 不能去重NaN和{}

```
function unique (arr) {
     if (!Array.isArray(arr)) {
        return
    }
    arr.sort((a, b) => a - b)
    const res = [arr[0]]
    for (let i = 1; i < arr.length; i++) {
        if (arr[i] !== arr[i - 1]) {
            res.push(arr[i])

        }
    }
    return res
}
```

5. 使用includes　　(和indexOf思路一样)

> 可以去重NaN，不能去重{}

```
function unique (arr) {
    if (!Array.isArray(arr)) {
        return
    }
    const res = []
    for (let i = 0; i < arr.length; i++) {
        if (!res.includes(arr[i])) {
            res.push(arr[i])
        }
    }
    return res
}
```

6. 使用filter　筛选出当前元素在原数组中的第一个索引 等于 当前索引 的元素

> 不能去重空对象 {}

```
function unique (arr) {
    if (!Array.isArray(arr)) {
        return
    }
    return arr.filter((item, index, arr) => {
        // 筛选出当前元素在原数组中的第一个索引 等于 当前索引 的元素
        return arr.indexOf(item, 0) === index
    })
}
```

7. 使用reduce +includes

> 不能去重NaN 和 { }

```
function unique (arr) {
    return arr.reduce((prev, cur) => prev.includes(cur) ? prev : [...prev, cur],
[])
}
```

8. 利用hasOwnProperty　去重全面

> 创建出一个对象，将数组中的元素的类型和值作为属性，值为true 。判断对象是否有相同的属性，如果没有则 存入该属性 并赋值为true

```
function unique (arr) {
    const obj = {}
    return arr.filter((item, index, arr) => {
        // hasOwnProperty() 方法会返回一个布尔值，指示对象自身属性中是否具有指定的属性（也
就是，是否有指定的键）。
        return obj.hasOwnProperty(typeof item + item) ? false : (obj[typeof item
+ item] = true)

    })
}
```

数组扁平化

1. 使用flat( ) 方法进行拍平， 可以传入一个数字作为可以拍平的层数，默认为1， 设置infinty后可以全部拍平
2. 使用递归

```
function myFlat (arr) {
    let res = []
    if (!Array.isArray(arr)) {
        return false
    }
    for (let i = 0; i < arr.length; i++) {
        if (Array.isArray(arr[i])) {
            res = res.concat(myFlat(arr[i]))           //或者
res.push(...myFlat(arr[i]))
        } else {
            res.push(arr[i])
        }
    }
    return res
}
```

3. 使用reduce + 递归

```
function myFlat (arr) {
    if (!Array.isArray(arr)) {
        return false
    }
    let res = arr.reduce((prev, cur) => {
        return prev.concat(Array.isArray(cur) ? myFlat(cur) : cur)
    }, [])
    return res
}
```

4. toString　　toString可以直接去除中括号

```
function myFlat (arr) {
    if (!Array.isArray(arr)) {
        return false
    }
    let res = arr.toString().split(',').map(item => {
        return Number(item)
    })
    return res
}
```

5. 使用 Generator 实现 flat 函数

```
function* myflat (arr, num) {
    if (num === undefined) num = 1
    for (const item of arr) {
        if (Array.isArray(item) && num > 0) {    // num > 0
            yield* myflat(item, num - 1)
        } else {
            yield item
        }
    }
}
const res = [...myflat(arr,Infinity)]              //generator 中内置 iterator 接口则可以使用拓展运算符 ... 进行调用
```

执行上下文与作用域

一、作用域

> **作用域** 指程序中定义变量的区域，它决定了当前执行代码对变量的访问权限。

- **全局作用域**：全局作用域为程序的最外层作用域，并一直存在
- **函数作用域**：函数作用域只有函数被定义时才会创建，包含在父级作用域/全局作用域中
- **作用域链**：当可执行代码内部访问变量时，会先查找本地作用域，如果找到目标变量即返回，否则会去父级作用域查找，一直查找到全局作用域 。 我们把这种**作用域的嵌套机制**，称为**作用域链**

词法作用域： 作用域是由书写代码时函数声明的位置决定的 （意味着函数被定义时，他的作用域已经确定了，和拿到哪里没有关系，所以也称为静态作用域） js使用的就是词法作用域

动态作用域： 基于调用栈，不是代码中的作用域嵌套

二、闭包

闭包的概念：

> 是指有那些引用了另一个函数作用域中变量的函数，通常出现在嵌套函数中

```
function outFn () {
    const data = 'hello world'
    return function innerFn () {
        console.log(data)
    }
}

let bar = outFn()                    //通常情况下，在outFn执行完成之后，函数作用域以及变量都会被销毁掉，但是闭包阻止了其销毁，我们仍可以访问外部函数的变量对象
bar()
```

闭包的使用场景：

一、模拟私有化属性

二、单例模式

单例模式是一种常见的设计模式。他保证了一个类型只有一个实例。实现方法：先判断实例是否存在，如果存在就直接返回，否则就创建了再返回。单例模式的好处就是避免了重复实例化带来的内存开销。

```
function Singleton () {
```

```
    this.data = 'singleton'
}
Singleton.getInstance = (function () {
    let instance
    return function () {
        if (instance) {
            return instance
        } else {
            instance = new Singleton()
            return instance
        }
    }
})()

let sa = Singleton.getInstance()
let sb = Singleton.getInstance()
console.log(sa === sb)    //true
console.log(sa.data)     // singleton
```

三、函数柯里化

函数柯里化，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一参数）的函数，并且返回接受余下的参数而且返回结果的函数的技术

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

四、节流防抖

节流： 一段时间内只能触发一次，如果这段时间内触发多次时间，一段时间后才能再次触发

> 1.鼠标不断点击触发，mousedown(单位时间内只触发一次)
>
> 2.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： 当事件被触发时，会延迟n秒后再触发回调函数，如果n秒内再次触发则重新开始计算时间 （n秒内只生效最后一次）

> 1.search搜索框，用户不断输入输入值，使用防抖来节约请求资源
>
> 2.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

```
// 防抖
function debounce (fn, delay = 1000) {
                let timer = null
                return () => {
                    if (timer) {
                        clearTimeout(timer)
                    }
                    timer = setTimeout(() => {
                        fn.apply(this, arguments)
```

```
                    }, delay)
                }
        }
// 节流
  function thottle (fn, delay = 1000) {
                let flag = true
                return () => {
                    if (flag) {
                        flag = false
                        setTimeout(() => {
                            fn.apply(this, arguments)
                            flag = true
                        }, delay)
                    }
                }
            }
```

垃圾回收机制：

[https://juejin.cn/post/6981588276356317214]:

> 当一些变量没有被引用关系后，垃圾回收机制会定期查找，然后释放其内存

标记清除算法 （ js的垃圾回收机制 ）

引擎在执行 GC（使用标记清除算法）时，需要从出发点去遍历内存中所有的对象去打标记，而这个出发点有很多，我们称之为一组 `根` 对象，而所谓的根对象，其实在浏览器环境中包括又不止于 `全局Window对象`、`文档DOM树` 等

大致过程如下：

- 垃圾收集器会在运行时给内存中的变量都打上一个标记，假设内存中的所有对象都是要被清除的垃圾，都标记上 0
- 从根节点开始遍历，把不是垃圾的节点改为1
- 清理所有标记为0 的节点，销毁并释放其所占的内存空间
- 最后，把所有内存中的对象都标记成0，等待下一轮的垃圾回收

优点：实现起来比较简单

缺点：1. 清除之后，剩余对象的内存的位置是不变的，可能会导致空闲的内存空间不连续，出现**内存碎片**

2. 由于空闲的内存空间并不连续，导致分配空间时需要进行比对，导致**分配内存速度慢**

使用**标记整理算法**可以有效解决这两个缺点

引用计数算法

如果没有引用指向该对象（零引用），对象将被垃圾回收机制回收

大致策略：

- 当声明了一个变量并且将一个引用类型赋值给该对象时这个值的引用次数就加1
- 如果同一个值给另一个变量，则引用次数加1
- 如果该变量的值被其他的值覆盖了，则引用次数减 1
- 当这个值的引用次数变为 0 的时候，说明没有变量在使用，这个值没法被访问了，回收空间，垃圾回收器会在运行的时候清理掉引用次数为 0 的值占用的内存

缺点： 当循环引用时会导致无法回收

闭包存在的问题

闭包可能会导致**内存泄露**

由于闭包的机制，外层函数调用完毕之后，会自动销毁，但是由于在闭包函数内访问了外部函数的变量并且根据垃圾回收机制，被另一个作用域引用的变量不会被回收，导致被访问的变量没有被回收。除非闭包函数解除调用才能销毁。 如果该函数使用的次数很少，不进行销毁的话就会变为闭包产生的内存泄漏。

解决方法： 将函数赋值 null

内存泄漏：

该释放的内存垃圾没有被释放，依然霸占着原有的内存不松手，造成系统内存的浪费，导致性能恶化，系统崩溃等严重后果，这就是所谓的**内存泄漏**

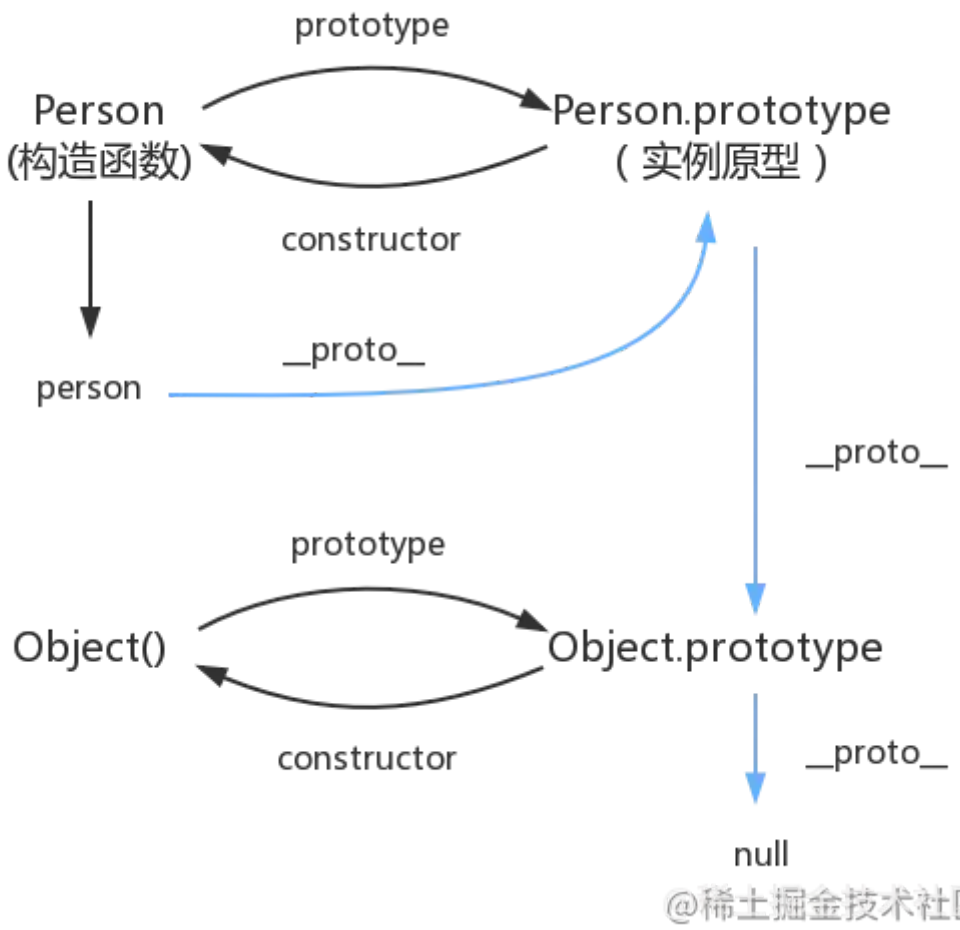
原型原型链

构造函数

如果这个函数可以使用**new**关键字来创建它的实例对象，那么我们就把这种函数称为 **构造函数**

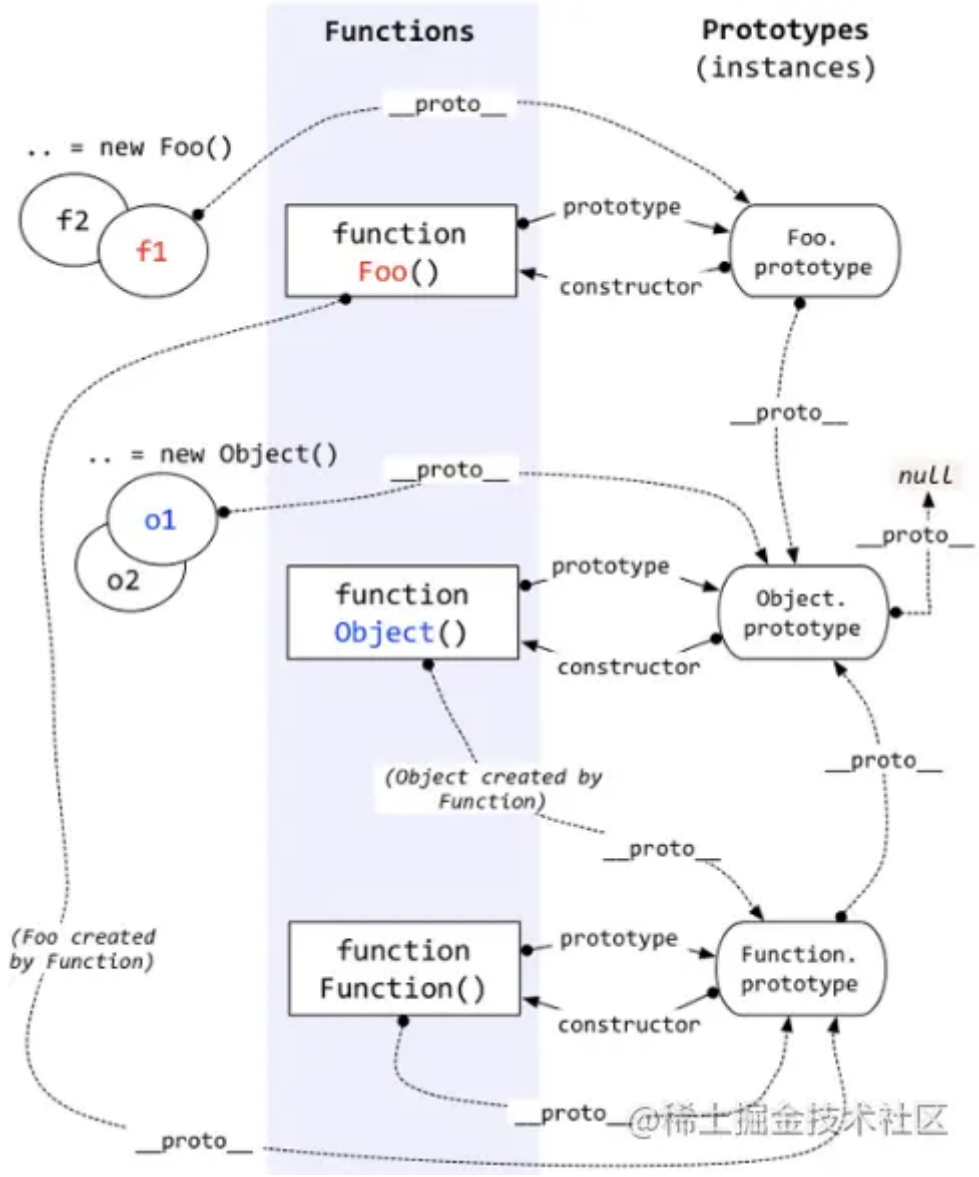
构造函数被声明时，原型对象也一同完成了创建，然后挂载到构造函数的prototype属性上

原型对象被创建出来后，会自动生成一个constructor，constructor指向构造函数



原型链

每个实例对象（object）都有一个私有属性（称之为 **proto** ）指向它的构造函数的原型对象（**prototype**）。该原型对象也有一个自己的原型对象（**proto**），层层向上直到一个对象的原型对象为 `null`,这种由**propto** 连接起来的链式结构，被称为**原型链**



继承

继承是面向对象软件技术当中的一个概念，与多态、封装共为面向对象的三个基本特征。继承可以使得子类具有父类的属性和方法或者重新定义、追加属性和方法等。**与其叫继承，委托的说法反而更准确些**

- **原型链继承**

  原型链继承，就是让对象实例通过原型链的方式串联起来，当访问目标对象的某一属性时，能顺着原型链进行查找，从而达到类似继承的效果。

```
// 父类
function SuperType (colors = ['red', 'blue', 'green']) {
    this.colors = colors;
}

// 子类
function SubType () {}
// 继承父类
SubType.prototype = new SuperType();
// 以这种方式将 constructor 属性指回 SubType 会改变 constructor 为可遍历属性
SubType.prototype.constructor = SubType;
```

- **组合继承**

  组合继承使用 `call` 在子类构造函数中调用父类构造函数

  缺点：父类的构造函数被调用了两次（创建子类原型时调用了一次，创建子类实例时又调用了一次），导致子类原型上会存在父类实例属性，浪费内存。

```
// 组合继承实现

function Parent(value) {
    this.value = value;
}

Parent.prototype.getValue = function() {
    console.log(this.value);
}

function Child(value) {
    Parent.call(this, value)
}

Child.prototype = new Parent();
```

- **寄生组合继承**

  使用 `Object.create(Parent.prototype)` 创建一个新的原型对象赋予子类从而解决组合继承的缺陷：

```


function Parent(value) {
    this.value = value;
}

Parent.prototype.getValue = function() {
    console.log(this.value);
}

function Child(value) {
    Parent.call(this, value)
}

Child.prototype = Object.create(Parent.prototype, {
    constructor: {
```

```
        value: Child,
        enumerable: false, // 不可枚举该属性
        writable: true, // 可改写该属性
        configurable: true // 可用 delete 删除该属性
    }
})
```

事件流和事件委托

事件流

事件流是描述的**从页面接受事件的顺序**，当几个都具有事件的元素层叠在一起的时候， 并不是只有当前被点击的元素会触发事件， 而是所有元素都会触发事件。

一次事件的完整过程： 捕获阶段 ---> 目标阶段 ---->冒泡阶段

通过e.stopPropagation（）来阻止事件冒泡

事件委托

原理：不给每个子节点单独设置事件监听器，而是设置在其父节点上，然后利用冒泡原理设置每个子节点。

优点：

- 可以大量节省内存占用，减少事件注册。
- 可以实现当新增子对象时，无需再对其进行事件绑定，对于动态内容部分尤为合适

缺点：

- 事件代理的常用应用应该仅限于上述需求，如果把所有事件都用事件代理，可能会出现事件误判。即本不该被触发的事件被绑定上了事件。

事件循环机制

事件循环中维护着两个队列，一个宏任务队列，一个微任务队列

所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。

主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。

一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。

主线程不断重复上面的第三步，这个过程就叫做时间循环

一、 浏览器的事件循环

宏任务队列： ajax 、setTimeout 、setInterval、DOM监听事件、UI Rending

微任务队列： Promise的then回调 Mutation Observer API 、queueMicrotask

除了宏任务队列和微任务队列还有一个 DOM渲染任务

执行规则：

1. **main script** 中的代码优先执行
2. 在执行任何一个**宏任务** 之前，都会先查看**微任务队列**中是否有任务要执行

   宏任务执行前，必须要保证微任务队列是空的，如果不为空的话，那就优先执行微任务队列中的任务（回调）
3. async 是一个异步函数 但是里面的代码会同步执行 遇到 await 后 会执行该await后的函数，后面的代码加入微任务队列中

```
async function async1(){
    console.log("async1 start")
    await async2()
    console.log('async1 end')
}
async function async2(){
    console.log('async2')
}

async1()
// 首先打印 async start
// 执行async2 打印async2
// async2 相当于返回 new Promise().resolve() ,则后面的console.log('async1 end') 相当于在Promise.then()中
// console.log('async1 end')加入微任务队列中
```

例题：

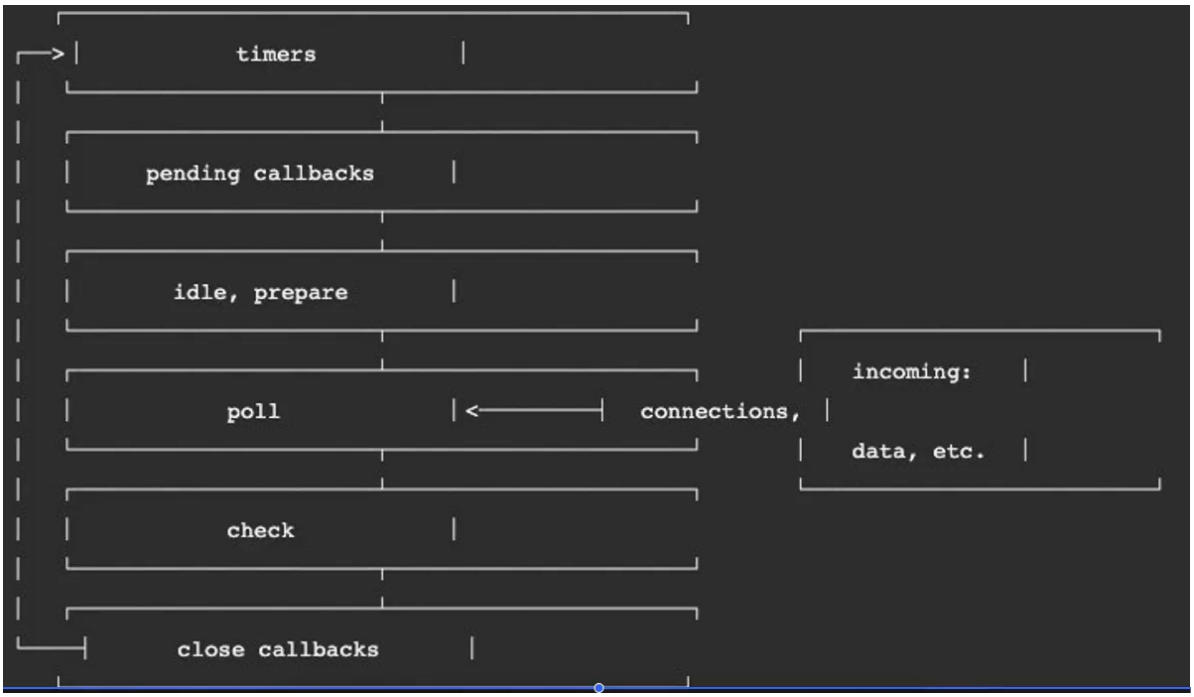
```
async function async1 () {
    console.log('async1 start')
    await async2()
    console.log('async1 end')
}
async function async2 () {
    console.log("async2")
}
console.log('script start')
setTimeout(() => {
    console.log('setTimeout')
}, 0)
async1()
new Promise(resolve => {
    console.log("promise1")
    resolve()
}).then(() => {
    console.log("promise2")
})
console.log('script end')


/*
宏任务：setTimeout
微任务：async1 end  promise2
执行顺序：script start
        async1 start
```

```
        async2
        promise1
        script end
        async1 end
        promise2
        setTimeout
*/
```

二、 Node中的事件循环机制

Node的一次Tick分为以下阶段：

- **定时器(timer)**: 在这个阶段执行 setTimeout、setInterval的回调函数
- **待定回调(pending callbacks)**：某些系统操作（如 TCP 错误类型）执行回调
- **idle,prepare** 仅在内部使用
- **轮询：poll** 检索新的 I/O 事件；执行 I/O 相关的回调（几乎所有相关的回调，关闭回调）
- **检测：check** setImmediate() 会在此阶段调用
- **关闭的回调函数** 关闭回调，例如： socket.on('close', ...)



宏任务：

setTimeout、setInterval、IO事件 、 setImmediate 、 close 事件

微任务：

Promise的then回调 、process.nextTick、queueMicrotask

node中的任务队列： （有顺序）

- 微任务队列
    1. next tick queue : process.nextTick
    2. other queue： Promise.then() 、 queueMicrotask
- 宏任务队列
    1. timer queue： setTimeout 、 setTnterval
    2. poll queue :IO事件
    3. check queue : setImmediate
    4. close queue； close事件

ajax的使用

```
function myajax (method, url, data, callback) {
    const xhr
    if (window.XMLHttpRequest) {   // Mozilla, Safari...
       xhr = new XMLHttpRequest();
    } else if (window.ActiveXObject) { // IE
       try {
         xhr = new ActiveXObject('Msxml2.XMLHTTP');
       } catch (e) {
         try {
           xhr = new ActiveXObject('Microsoft.XMLHTTP');
         } catch (e) {}
       }
    }
    xhr.onreadystatechange = function () {
        if (xhr.readyState === 4) {
            if (xhr.status >= 200 && xhr.status < 400) {
                callback(xhr.response)
            }
        }
        callback(xhr.status)
    }
    xhr.open(method, url)
    xhr.setRequestHeader('Content-Type', 'application/json') //设置请求头
    xhr.send(data)
}
```

readyState的状态从0到4变化

- 0：请求未初始化，xmlhttprequest对象还没有完成初始化
- 1：服务器连接已建立，xmlhttprequest对象开始发送请求
- 2：请求已接收，xmlhttprequest对象的请求发送完成
- 3：请求处理中，xmlhttprequest对象开始读取服务器的响应
- 4：请求已完成且响应就绪，xmlhttprequest对象读取服务器响应结束

Promise 封装ajax

```
function myajax (method, url, data) {
    return new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest()
        xhr.onreadystatechange = function () {
            if (xhr.readyState === 4) {
                if (xhr.status >= 200 && xhr.status < 400) {
                    resolve(xhr.response)
                } else {
                    reject(xhr.status)
                }
            } else {
                retrun
            }

        }
        xhr.open(method, url)
        xhr.setRequestHeader('Content-Type', 'application/json') //设置请求头
        xhr.send(data)
```

```
    })
}
```

Fetch

fetch的返回值是一个Promise

```
fetch(url, options).then(function(response) {
// handle HTTP response
}, function(error) {
 // handle network error
})
```

Esmodule，commonjs，UMD，AMD,CMD

一、CommonJS

```
   const express = require('express')       //导入
    module.exports = function(){                //导出

    }
    exports.obj = {
       age:18
    }
```

- commenjs 是 **同步**导入
- 模块是在**运行时进行加载**，模块可以多次加载，但是只会在第一次加载时运行一次，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果。要想让模块再次运行，必须清除缓存。
- CommonJS模块的加载机制是，输入的是被输出的值的**拷贝**，模块内部的变化也不会影响这个值
- 需要在浏览器中运行时，需要进行转换和打包
- this指向当前模块

二、ES Module

```
    import {ref,reactive} from 'vue'

    export defalut {
        name:'hello'
    }
```

- 可以直接在浏览器中使用，即使用`<script type = 'module' src = ''></script>`
- import 是 **异步加载**
- 模块输出是 **值的引用**，即被输出模块的内部的改变会影响引用的改变；
- 模块实在**编译时进行加载**
- this指向undefined

ES6 Module的特点(对比CommonJS)

- CommonJS模块是运行时加载，ES6 Module是编译时输出接口；
- CommonJS加载的是整个模块，将所有的接口全部加载进来，ES6 Module可以单独加载其中的某个接口；
- CommonJS输出是值的拷贝，ES6 Module输出的是值的引用，被输出模块的内部的改变会影响引用的改变；
- CommonJS this指向当前模块，ES6 Module this指向undefined;

三、AMD （RequireJS）

通过define来定义模块，require来导入模块

```
//define可以传入三个参数，分别是字符串-模块名、数组-依赖模块、函数-回调函数
//a.js
define(function (){
    console.log(1)
})

//数组中声明需要加载的模块，可以是模块名、js文件路径
//b.js
require(['a.js'], function(a){
    console.log(a);// 1
});
```

- 在define方法里传入的依赖模块(数组)，会在一开始就下载并执行。
- 采用异步方式加载模块，模块的加载不影响它后面语句的运行.

四、CMD (SeaJS)

```
 // 定义模块  module.js
define(function(require, exports, module) {
  var $ = require('jquery.min.js')
  $('div').addClass('active');
});

// 加载模块
seajs.use(['module.js'], function(my){
});
```

- 对于依赖的模块，CMD推崇依赖就近，延迟执行。也就是说，只有到require时依赖模块才执行。
- 异步加载

五、UMD

它就是将AMD和Commonjs整合起来,使得代码在不同环境都可以正常运行.

```
(function(window, factory){
   if (typeof exports === 'object') {
      module.exports = factory()
   } else if (typeof define === 'function' && define.amd) {
      define(factory)
   } else {
     window.eventUtil = factory()
  }
})(this,function () {
    //    do something
})
```