js相关基础：

什么是主流浏览器：具备独立内核的浏览器

IE：trdient内核

Chrome:webkit/blink内核

Firefox:Gecko内核

Opera:presto 内核

safari:webkit内核

js 纯函数：

1.一个函数的返回值只依赖于它的参数

2.函数执行过程中没有副作用（一个函数执行过程产生了外部可观察的变化那么就说这个函数是有副作用的，除了修改外部的变量，一个函数在执行过程中还有很多方式产生外部可观察的变化，比如说调用 DOM API 修改页面，或者你发送了 Ajax 请求，还有调用 window.reload 刷新浏览器，甚至是 console.log 往控制台打印数据，更改了入参也是副作用。）

当我们在浏览器输入网址并回车后发生了什么?

事件的定义

事件：元素天生具备的行为方式（和写不写JS代码没有关系）【onclick不是事件，click才是事件，浏览器会把一些常用事件挂载到元素对象的私有属性上，让我们可以实现DOM0事件绑定】，当我们去操作元素的时候会触发元素的很多事件

事件绑定：给当前元素的某个事件绑定方法，目的是让当前元素某个事件被触发时，做出一些反应

事件绑定的两种方法

- DOM0级事件绑定： curEle.onclick=function(){};
- DOM2级事件绑定

标准浏览器： curEle.addEventListener('click',function(){},false) //false=>让事件在冒泡传播时执行 //true=>让事件在捕获阶段执行

IE6-8:curEle.attachEvent('onclick',function(){}) //*绑定的方法都是在冒泡传播阶段执行*

DOM0级事件绑定的原理

给当前元素的某一私有属性（onXXX）赋值的过程；（之前属性默认值是null，如果我们赋值了一个函数，就相当于绑定了一个方法）

当我们赋值成功（赋值一个函数），此时浏览器会把DOM元素和赋值的的函数建立关联，以及建立DOM元素的行为监听，当某一行为被用户触发，浏览器会把赋值的函数执行；

DOM0级事件绑定的特点

只有DOM元素天生拥有这个私有属性（onxxx事件私有属性），我们赋值的方法才叫事件绑定，否则属于设置自定义属性

移除事件绑定的时候，我们只需要赋值为null；

在DOM0事件绑定中，只能给当前元素的某一个事件行为绑定一个方法，绑定多个方法，最后一次的绑定的会覆盖前面绑定的

DOM2级事件绑定的原理

DOM2事件绑定使用的 addEventListener/attachEvent方法都是在eventTarget这个内置类的原型上定义的，我们调用的时候，首先要通过原型链找到这个方法，然后执行完成事件绑定的效果

浏览器会给当前元素的某个事件行为开辟一个事件池（事件队列）【浏览器有一个统一的事件池，每个元素绑定的行为都放在这里，通过相关标志区分】，当我们通过addEventListener/attachEvent进行事件绑定的时候，会把绑定的方法放在事件池中；

当元素的某一行为被触发，浏览器回到对应事件池中，把当前放在事件池的所有方法按序依次执行

特点

所有DOM0支持的行为，DOM2都可以用，DOM2还支持DOM0没有的事件行为（这样说比较笼统）

（核心）【浏览器会把一些常用事件挂载到元素对象的私有属性上，让我们可以实现DOM0事件绑定，DOM2：凡是浏览器给元素天生设置的事件在DOM2中都可以使用】

例如：onDOMContentLoaded（所有的DOM0和IE6-8的DOM2都不支持）

onDOMContentLoaded//当前浏览器中的DOM结构加载完成，就会触发这个事件

DOM2中可以给当前元素的某一事件行为绑定多个不同方法（因为绑定的所有方法都放在事件池中）；

事件的移除:事件类型、绑定的方法、传播阶段三个完全一致，才可以完成移除(因此在绑定方法时，尽量不要用匿名函数，否则不好移除)

```
//=>ON:给当前元素的某个事件绑定某个方法
 var on = function (curEle, type, fn) {
 if (document.addEventListener) {
 //=>标准浏览器
 curEle.addEventListener(type, fn, false);
 return;
 }
 //=>IE6~8
 curEle.attachEvent('on' + type, fn);
};
//=>OFF:移除当前元素某个事件绑定的某个方法
.var off = function (curEle, type, fn) {
 if (document.removeEventListener) {
 curEle.removeEventListener(type, fn, false);
 return;
 }
 //=>IE6~8
 curEle.detachEvent('on' + type, fn);
};
```

DOM0和DOM2绑定的方法是毫无联系的（因为是两套完全不同的机制），即使绑定的方法相同，也是执行两次，谁先绑定，就先执行谁

事件对象

> 对以下事件绑定的严谨版描述：基于DOM0级事件的绑定方式，给box的click事件绑定了一个方法。当手动触发click事件时，会把绑定的方法执行
> box.onclick=function(e){
> arguments[0]===e;//=>e就是事件对象
> e=e||window.event;(兼容版获取事件对象的写法)
> };

> 当元素的某个事件行为被触发，不仅会把之前绑定的方法执行，还会给绑定的方法传递一个值（浏览器默认传递的），我们把传递的这个值称为事件对象：（标准浏览器）
> - 这个值是个对象类型的值，里面存储了很多的属性和方法，
> - 这些存储的值都是当前本次操作的基本信息，例如：鼠标位置，触发行为类型。。。。【只和本次操作有关,一个页面中只有一个事件对象】
>
> IE6-8下关于事件对象的机制
> - 方法被触发执行时，浏览器并没有把事件对象当做值传递给函数；(e是undefined)
> - IE6-8下的事件对象需要我们通过window.event来获取

实例的_proto_属性等于构造函数的Prototype属性（实例都有一个属性叫做_proto_，它是个指针，指向原型对象。）

构造函数、原型与实例之间的关系

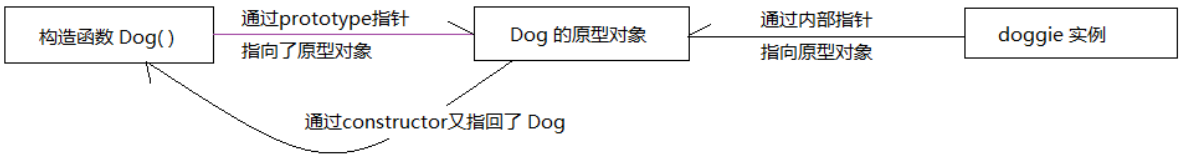
每创建一个函数，该函数就会自动带有一个 **prototype** 属性。**该属性是个指针，指向了一个对象**，我们称之为 **原型对象**。什么是指针？指针就好比学生的学号，原型对象则是那个学生。我们通过学号找到唯一的那个学生。假设突然，指针设置 null，学号重置空了，不要慌，对象还存在，学生也没消失。只是不好找了。

原型对象上默认有一个属性 **constructor**，该属性也是一个指针，指向其相关联的构造函数。

通过调用构造函数产生的实例，都有一个**内部属性**，指向了原型对象。所以实例能够访问原型对象上的所有属性和方法。

所以三者的关系是，每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个指向原型对象的内部指针。通俗点说就是，实例通过内部指针可以访问到原型对象，原型对象通过constructor指针，又可以找到构造函数。

> ”每个构造函数都有一个原型对象，eg:Function.prototype
>
> 原型对象都包含一个指向构造函数的指针,eg:Function.prototype.constructor === Function
>
> 实例都包含一个指向原型对象的内部指针。”eg:f._proto_ === Function.prototype
>
> ——此段话摘自《JavaScript高级程序设计》。



ES6相关：

es6数组map方法：返回一个新数组，如果迭代器函数没有return，默认返回undefined;
es6数组find方法：找到目标后（return true）自动跳出循环（break），不会再遍历后面的内容;

every方法:（&&） 一假即假，找到第一个为假的值，后面的不会再遍历 返回布尔值
some方法： （||） 一真即真 找到第一个为真的值，后面的不会再遍历 返回布尔值

let 在同一作用域下不能重复声明

向 Set 加入值的时候，不会发生类型转换，所以`5`和`"5"`是两个不同的值。Set 内部判断两个值是否不同，使用的算法叫做“Same-value-zero equality”，它类似于精确相等运算符（`===`），主要的区别是向 Set 加入值时认为`NaN`等于自身，而精确相等运算符认为`NaN`不等于自身。两个空对象不相等，所以它们被视为两个值。

```
let set = new Set();
set.add({});
set.size // 1;
set.add({});
set.size //2
```

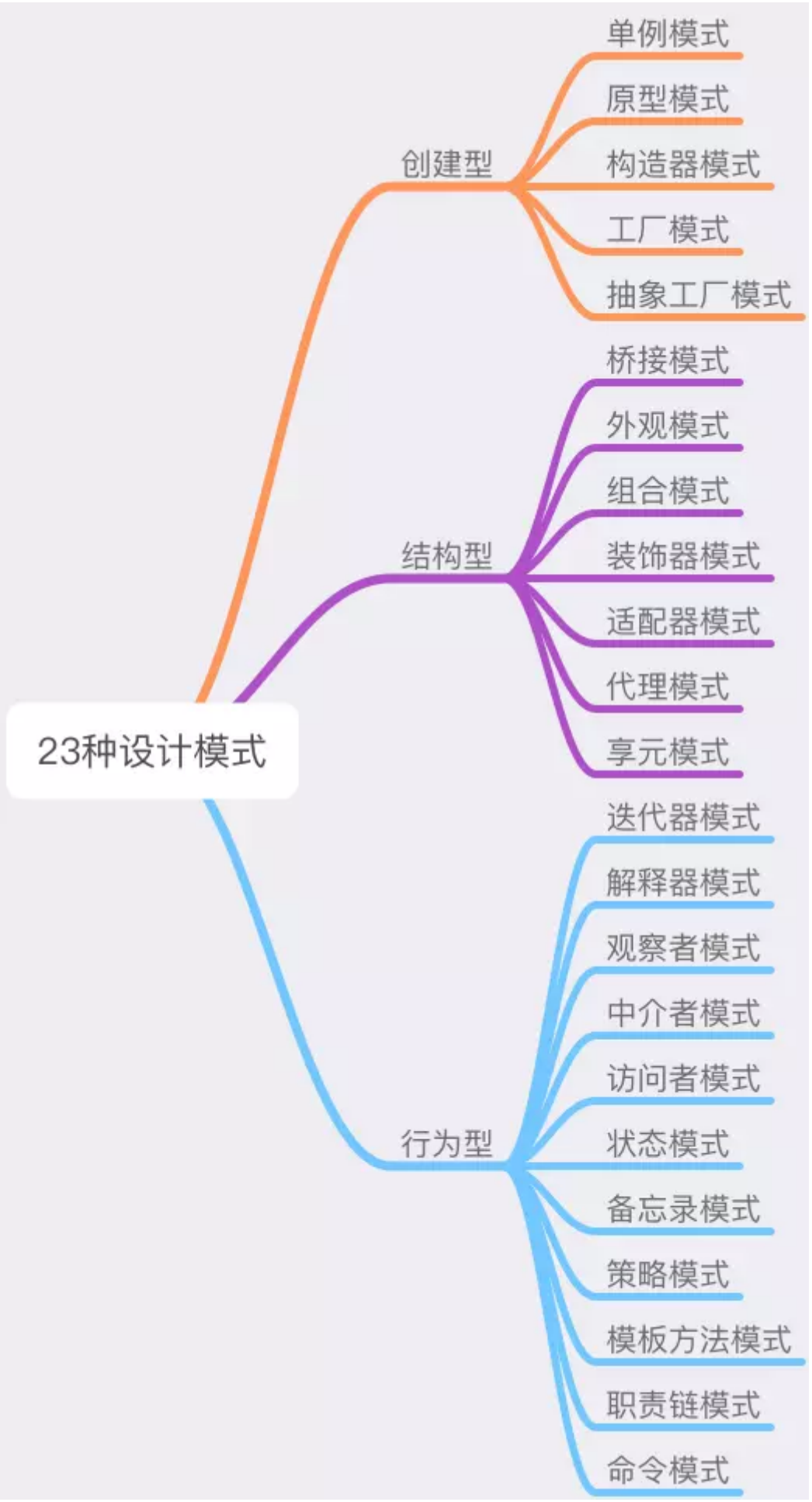
`Set`的遍历顺序就是插入顺序,这个特性有时非常有用，比如使用 Set 保存一个回调函数列表，调用时就能保证按照添加顺序调用。

js设计模式相关：

烹饪有菜谱，游戏有攻略，每个领域都存在一些能够让我们又好又快地达成目标的“套路”。在程序世界，编程的“套路”就是设计模式。---解决不同问题的核心方案
像做题一样从“映射到默写”；
设计原则是设计模式的指导理论，它可以帮助我们规避不良的软件设计。SOLID 指代的五个基本原则分别是：

- 单一功能原则（Single Responsibility Principle）
- 开放封闭原则（Opened Closed Principle）
- 里式替换原则（Liskov Substitution Principle）
- 接口隔离原则（Interface Segregation Principle）
- 依赖反转原则（Dependency Inversion Principle）

封装变化：**将变与不变分离，确保变化的部分灵活，不变的部分稳定**;这样的代码，就是我们所谓的“**健壮**”的代码，它可以经得起变化的考验。而设计模式出现的意义，就是帮我们写出这样的代码。


23种设计模式
创建型
单例模式
原型模式
构造器模式
工厂模式
抽象工厂模式
结构型
桥接模式
外观模式
组合模式
装饰器模式
适配器模式
代理模式
享元模式
行为型
迭代器模式
解释器模式
观察者模式
中介者模式
访问者模式
状态模式
备忘录模式
策略模式
模板方法模式
职责链模式
命令模式

开放封闭原则：**对拓展开放，对修改封闭**；
抽象工厂类：抽取共性，不做具体实现
具体工厂类：继承抽象工厂类，对其中的方法具体实现；
抽象产品类：抽取共性，不做具体实现；
具体产品类：继承抽象产品类，对其中的方法具体实现；

保证一个类仅有一个实例，并提供一个访问它的全局访问点，这样的模式就叫做单例模式。

单例模式想要做到的是，**不管我们尝试去创建多少次，它都只给你返回第一次所创建的那唯一的一个实例。**
要做到这一点，就需要构造函数**具备判断自己是否已经创建过一个实例**的能力。我们现在把这段判断逻辑写成一个静态方法(其实也可以直接写入构造函数的函数体里)

实例1：实现Storage，使得该对象为单例，基于 localStorage 进行封装。实现方法 setItem(key,value) 和 getItem(key)。

1.静态方法版

```
// 定义Storage
class Storage {
 static getInstance() {
 // 判断是否已经new过1个实例
 if (!Storage.instance) {
 // 若这个唯一的实例不存在，那么先创建它
 Storage.instance = new Storage()
 }
 // 如果这个唯一的实例已经存在，则直接返回
  return Storage.instance
 }
 getItem (key) {
 return localStorage.getItem(key)
 }
 setItem (key, value) {
 return localStorage.setItem(key, value)
 }
```

```
}

const storage1 = Storage.getInstance()
const storage2 = Storage.getInstance()

storage1.setItem('name', '李雷')
// 李雷
storage1.getItem('name')
// 也是李雷
storage2.getItem('name')

// 返回true
storage1 === storage2
```

2.闭包版

```
// 先实现一个基础的StorageBase类，把getItem和setItem方法放在它的原型链上
function StorageBase () {}
StorageBase.prototype.getItem = function (key){
 return localStorage.getItem(key)
}
StorageBase.prototype.setItem = function (key, value) {
 return localStorage.setItem(key, value)
}

// 以闭包的形式创建一个引用自由变量的构造函数
const Storage = (function(){
 let instance = null
 return function(){
 // 判断自由变量是否为null
 if(!instance) {
 // 如果为null则new出唯一实例
 instance = new StorageBase()
 }
 return instance
 }
})()

const storage1 = new Storage()
const storage2 = new Storage()

```

```
storage1.setItem('name', '李雷')
// 李雷
storage1.getItem('name')
// 也是李雷
storage2.getItem('name')

// 返回true
storage1 === storage2
```

实例1：实现全局唯一弹框

原型

在 JavaScript 中，每个构造函数都拥有一个`prototype`属性，它指向构造函数的原型对象，这个原型对象中有一个 construtor 属性指回构造函数；每个实例都有一个`__proto__`属性，当我们使用构造函数去创建实例时，实例的`__proto__`属性就会指向构造函数的原型对象。

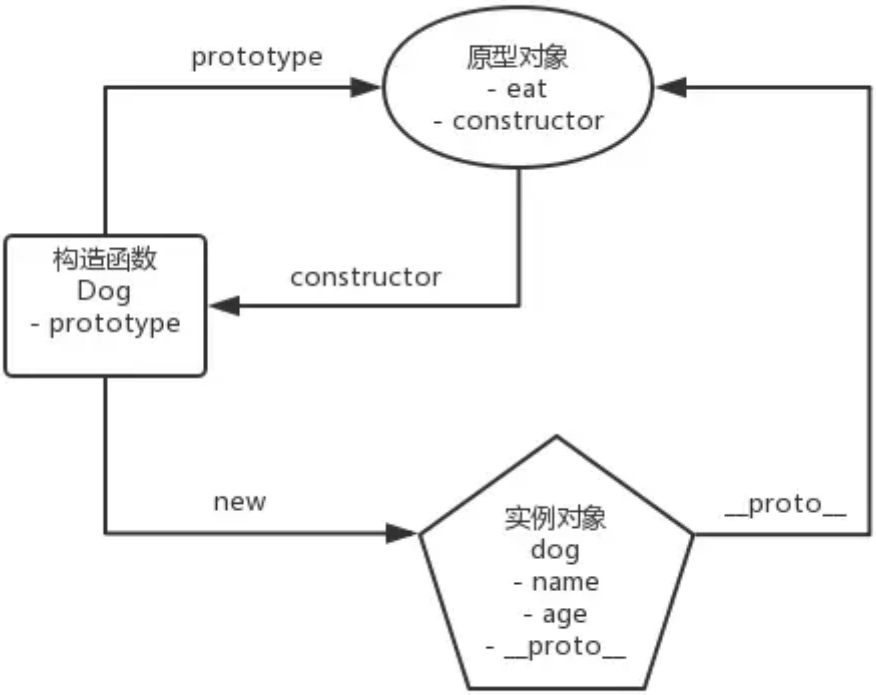
具体来说，当我们这样使用构造函数创建一个对象时：

```
// 创建一个Dog构造函数
function Dog(name, age) {
  this.name = name
  this.age = age
}

Dog.prototype.eat = function() {
  console.log('肉骨头真好吃')
}

// 使用Dog构造函数创建dog实例
const dog = new Dog('旺财', 3)
```

这段代码里的几个实体之间就存在着这样的关系：



深拷贝没有完美方案，每一种方案都有它的边界 case。而面试官向你发问也并非是要求你破解人类未解之谜，多数情况下，他只是希望考查你对**递归**的熟练程度。所以递归实现深拷贝的核心思路，大家需要重点掌握（解析在注释里）：(这个示例也存在一定问题，爆栈，循环引用等问题)

```
function deepClone(obj) {
 // 如果是值类型，则直接return
 if(typeof obj !== 'object') {
 return obj
 }

 // 定义结果对象
 let copy = {}

 // 如果对象是数组，则定义结果数组
 if(obj.constructor === Array) {
 copy = []
 }

 // 遍历对象的key
 for(let key in obj) {
 // 如果key是对象的自有属性
```

```
18  if(obj.hasOwnProperty(key)) {
19  //判断obj子元素是否为对象或数组，如果是，递归复制
20            if(obj[key]&&typeof obj[key] ==="object"){
21               // 递归调用深拷贝方法
22  copy[key] = deepClone(obj[key])
23            }else{
24              //如果不是，简单复制
25              copy[key] = obj[key];
26            }
27
28  }
29  }
30
31  return copy
32 }
```

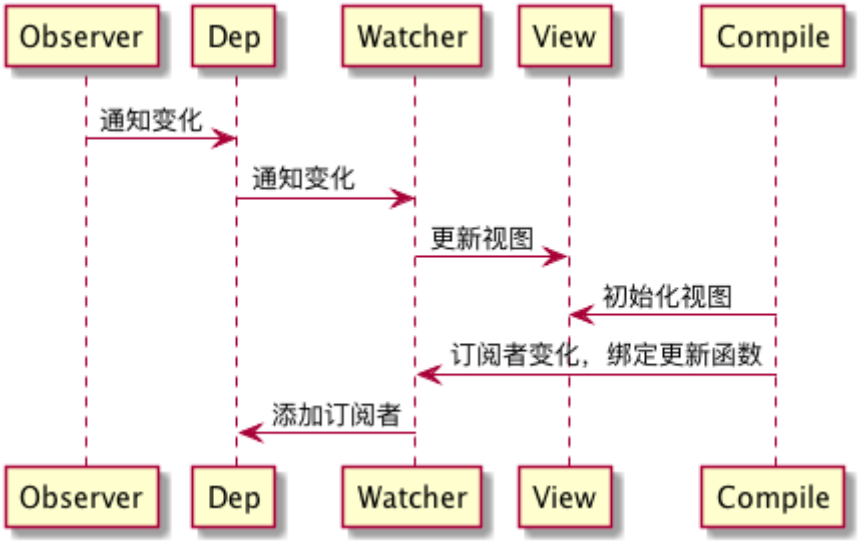
调用深拷贝方法，若属性为值类型，则直接返回；若属性为引用类型，则递归遍历。这就是我们在解这一类题时的核心的方法。

vue的响应式原理理解：

在Vue数据双向绑定的实现逻辑里，有这样三个关键角色：

- observer（监听器）：注意，此 observer 非彼 observer。在我们上节的解析中，observer 作为设计模式中的一个角色，代表“订阅者”。但在Vue数据双向绑定的角色结构里，所谓的 observer 不仅是一个数据监听器，它还需要对监听到的数据进行**转发**——也就是说它**同时还是一个发布者**。
- watcher（订阅者）：observer 把数据转发给了**真正的订阅者**——watcher对象。watchr 接收到新的数据后，会去更新视图。
- compile（编译器）：MVVM 框架特有的角色，负责对每个节点元素指令进行扫描和解析，指令的数据初始化、订阅者的创建这些“杂活”也归它管~

这三者的配合过程如图所示：



核心代码

首先我们需要实现一个方法，这个方法会对需要监听的数据对象进行遍历、给它的属性加上定制的 getter 和 setter 函数。这样但凡这个对象的某个属性发生了改变，就会触发 setter 函数，进而通知到订阅者。这个 setter 函数，就是我们的监听器：

```
// observe方法遍历并包装对象属性
function observe(target) {
 // 若target是一个对象，则遍历它
 if(target && typeof target === 'object'){
 object.keys(target).forEach((key)=> {
 // defineReactive方法会给目标属性装上“监听器”
 defineReactive(target, key, target[key])
 })
 }
}

// 定义defineReactive方法
function defineReactive(target, key, val) {
 // 属性值也可能是object类型，这种情况下需要调用observe进行递归遍历
 observe(val)
 // 为当前属性安装监听器
 Object.defineProperty(target, key, {
 // 可枚举
 enumerable: true,
 // 不可配置
 configurable: false,
```

```
  get: function () {
  return val;
  },
  // 监听器函数
  set: function (value) {
  console.log(`${target}属性的${key}属性从${val}值变成了了${value}`)
  val = value
  }
  });
 }
```

下面实现订阅者 Dep：

```
// 定义订阅者类Dep
class Dep {
 constructor() {
 // 初始化订阅队列
 this.subs = []
 }

 // 增加订阅者
 addSub(sub) {
  this.subs.push(sub)
 }

 // 通知订阅者（是不是所有的代码都似曾相识？）
 notify() {
 this.subs.forEach((sub)=>{
 sub.update()
 })
 }
}
```

现在我们可以改写 defineReactive 中的 setter 方法，在监听器里去通知订阅者了：

```
function defineReactive(target, key, val) {
 const dep = new Dep()
 // 监听当前属性
 observe(val)
 Object.defineProperty(target, key, {
 set: (value) => {
 // 通知所有订阅者
 dep.notify()
```

```
  }
  })
}
```

一起来实现一个Event Bus（注意看注释里的解析）：

```
class EventEmitter {
 constructor() {
 // handlers是一个map，用于存储事件与回调之间的对应关系
 this.handlers = {}
 }

 // on方法用于安装事件监听器，它接受目标事件名和回调函数作为参数
 on(eventName, cb) {
 // 先检查一下目标事件名有没有对应的监听函数队列
  if (!this.handlers[eventName]) {
  // 如果没有，那么首先初始化一个监听函数队列
  this.handlers[eventName] = []
  }

  // 把回调函数推入目标事件的监听函数队列里去
  this.handlers[eventName].push(cb)
  }

  // emit方法用于触发目标事件，它接受事件名和监听函数入参作为参数
  emit(eventName, ...args) {
  // 检查目标事件是否有监听函数队列
  if (this.handlers[eventName]) {
  // 如果有，则逐个调用队列里的回调函数
  this.handlers[eventName].forEach((callback) => {
  callback(...args)
  })
  }
  }

  // 移除某个事件回调队列里的指定回调函数
  off(eventName, cb) {
  const callbacks = this.handlers[eventName]
  const index = callbacks.indexOf(cb)
  if (index !== -1) {
  callbacks.splice(index, 1)
```

```
  }
  }

  // 为事件注册单次监听器
  once(eventName, cb) {
  // 对回调函数进行包装，使其执行完毕自动被移除
  const wrapper = (...args) => {
  cb.apply(...args)
  this.off(eventName, cb)
  }
  this.on(eventName, wrapper)
  }
}
```

观察者模式与发布-订阅模式的区别是什么?

发布者直接触及到订阅者的操作，叫观察者模式。发布者不直接触及到订阅者、而是由统一的第三方来完成实际的通信的操作，叫做发布-订阅模式。

行为型----迭代器模式

ES6实现一个迭代器生成函数，使用内置的**生成器**（Generator）实现

```
// 编写一个迭代器生成函数
function *iteratorGenerator() {
 yield '1号选手'
 yield '2号选手'
 yield '3号选手'
}

const iterator = iteratorGenerator()

iterator.next()
iterator.next()
iterator.next()
```

ES5去写一个能够生成迭代器对象的迭代器生成函数（解析在注释里）：

```
// 定义生成器函数，入参是任意集合
function iteratorGenerator(list) {
 // idx记录当前访问的索引
 var idx = 0
 // len记录传入集合的长度
 var len = list.length
```

```
    return {
    // 自定义next方法
    next: function() {
     // 如果索引还没有超出集合长度，done为false
     var done = idx >= len
     // 如果done为false，则可以继续取值
     var value = !done ? list[idx++] : undefined

     // 将当前值与遍历是否完毕（done）返回
     return {
     done: done,
     value: value
     }
     }
     }
}

var iterator = iteratorGenerator(['1号选手', '2号选手', '3号选手'])
iterator.next()
iterator.next()
iterator.next()
```

此处为了记录每次遍历的位置，我们实现了一个闭包，借助自由变量来做我们的迭代过程中的“游标”。

算法和数据结构相关：

算法：计算或解决问题的步骤，与食谱的区别：精确

仅当列表是有序的，二分查找才有效

随机读取数据时：数组的效率比链表效率高

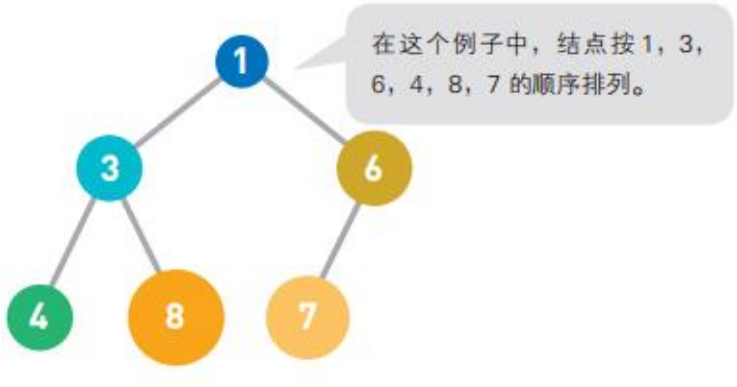
全部读取数据时：链表的效率较高

链表中的元素可存储在内存中的任何地方，链表中的每个元素都存储了下一个元素的地址，从而使一系列随机的内存地址联系起来

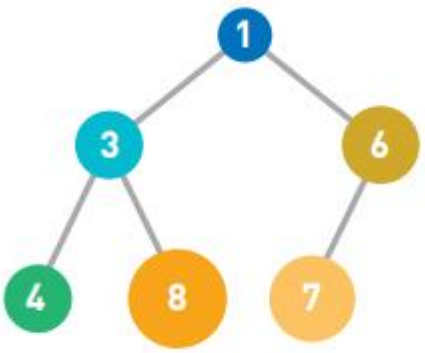
	数组	链表
读取	O(1)	O(n)
插入	O(n)	O(1)
删除	O(n)	O(1)

O(1)：常量时间　O(n)：线性时间

堆是一种图的树形结构，被用于实现“优先队列”（priority queues）。优先队列是一种数据结构，可以自由添加数据，但取出数据时要从最小值开始按顺序取出。在堆的树形结构中，各个顶点被称为“结点”（node），数据就存储在这些结点中。



这就是堆的示例。结点内的数字就是存储的数据。堆中的每个结点最多有两个子结点。树的形状取决于数据的个数。另外，结点的排列顺序为从上到下，同一行里则为从左到右。



在堆中存储数据时必须遵守这样一条规则：子结点必定大于父结点。因此，最小值被存储在顶端的根结点中。往堆中添加数据时，为了遵守这条规则，一般会把新数据放在最下面一行靠左的位置。当最下面一行里没有多余空间时，就再往下另起一行，把数据加在这一行的最左端。

堆中最顶端的数据始终最小，所以无论数据量有多少，取出最小值的时间复杂度都为 O(1)。

另外，因为取出数据后需要将最后的数据移到最顶端，然后一边比较它与子结点数据的大小，一边往下移动，所以取出数据需要的运行时间和树的高度成正比。假设数据量为 n，根据堆的形状特点可知树的高度为 log2n ，那么重构树的时间复杂度便为 O(logn)。

添加数据也一样。在堆的最后添加数据后，数据会一边比较它与父结点数据的大小，一边往上移动，直到满足堆的条件为止，所以添加数据需要的运行时间与树的高度成正比，也是 O(logn)

二叉查找树（又叫作二叉搜索树或二叉排序树）是一种数据结构，采用了图的树形结构。数据存储于二叉查找树的各个结点中。二叉查找树有两个性质。第一个是**每个结点的值均大于其左子树上任意一个结点的值（图1）;第二个是每个结点的值均小于其右子树上任意一个结点的值（图2）。**

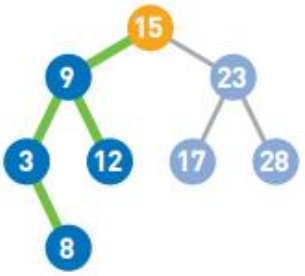


图1

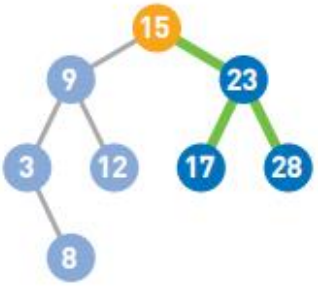


图2