Set，Map 详解

为什么要使用 Set，以及 Map？

在说 Set，Map 之前，我们先来了解一个概念，叫做集合。Map 和 Set 的产生原因来自于集合。那什么是集合呢？

集合的定义：

计算机科学中，集合是一组可变数量的数据项(也可能是 0 个)的组合，这些数据项可能共享某些特征，需要以某种操作方式一起进行操作。一般来讲，这些数据项的类型是相同的，或是满足相同的条件，或者遵循相同的规则。

集合的特点： 长度可变。

正常来说，传统的数组（C，JAVA 中）是不属于集合的。在 JAVA 或者 C 中，数组是静态的，一个数组实例具有固定的大小，一旦创建了就无法改变容量了。而集合大小不固定，是可以动态扩展容量的，可以根据需要动态改变大小，集合提供更多的成员方法，能满足更多的需求。若程序时不知道究竟需要多少对象，需要在空间不足时自动扩增容量，则需要使用集合，数组不适用。

但是在我们的 JavaScript 当中数组发生的巨大的变化，更灵活，更方便，与传统意义上的数组只是“形似“，包括在 V8 引擎反编译之后对应的不是 C 语言中的真实数组，在加上 JavaScript 是解释型语言，在加上反编译，执行效率自然就会偏低一些。关于 JS 的 Array 在 V8 引擎当中到底怎么实现的，咱们在这里不过多的说明，如果想深入的了解。可以持续关注我们的课程，后面也会出一些相关的内容。

现在有了一个集合的概念，很大很广泛。根据这样的集合大概念，在加上不同的共享特征，或者是统一的条件，就可以抽象出来称为一个新的集合。先从 JS 的 Array 说起。

JS 数组是一个集合，这个集合的特点是，存储一些有序的内容（通过下标访问），长度可变，类型不限制。但是基于前车之鉴，在 JS 数组封装的同时添加多种的遍历方式(forEach, map，等等)，支持栈操作（push,pop），队列操作(shift, unshift)，包括更灵活的在数组中插值，删值等功能。

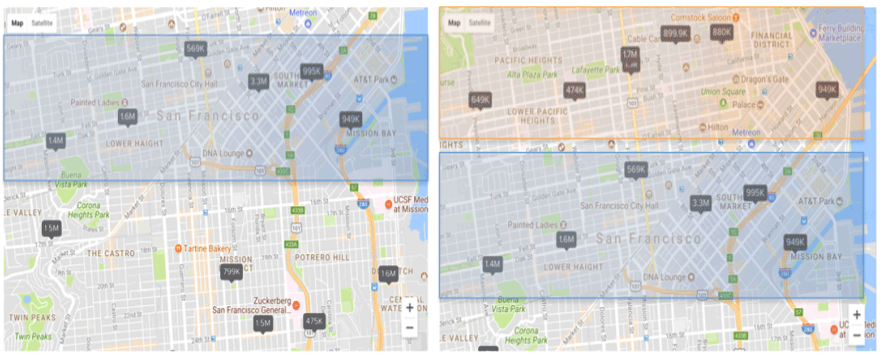
我们需要这种线性的结构用于存储数据，在 js 定制初期的时候，就把数组引入进来，后续的不断发展中得到我们现在的数组。

对于 JS 对象是一种满哈希表结构（散列表），至于什么是哈希表结构，在后面会说明。

在我们已有数组这种结构的基础之上，我们会有一个很常用的需求，数组去重。去重是开发中经常会碰到的一个热点问题，不过目前项目中碰到的情况都是后台接口使用 SQL 去重，简单高效，基本不会让前端处理去重。那么前端处理去重该怎么办呢？

在我们前端当中会有这样的应用场景：

对于地图的范围选择，我们会对地图进行不同区域的选择。



地图往下拖的时候要更新地图上的房源标签数据，上图蓝框表示不变的标签，而黄框表示新加的房源。

后端每次都会把当前地图可见区域的房源返回给我，当用户拖动的时候需要知道哪些是原先已经有的房源，哪些是新加的。把新加的房源画上，已有的房源保持不动。因此需要对比当前房源和新的结果哪些是重复的。我们只需要不重复一组数据。

把这个问题抽象一下就变成：给两个数组，进行合并然后去重。这个也就是我们在项目中存在的问题，通过数组去重去解决。关于数组去重的方式，有非常多的方法可以实现。大家一般来说会按照下面的逻辑进行去重：

```
Array.prototype.unique = function () {
    var newArr = []
    var len = this.length
    for(var i=0; i<len; i++) {
       if(newArr.indexOf(this[i]) == -1) {
          newArr.push(this[i])
       }
    }
    return newArr
}
```

这种方法是最最直接的一种方法进行去重。虽说功能我们可以做到，当数据量小的时候完全可以采用这样方式，当数据量大时，数组的 indexOf 查

找性能还是非常差的。很多人都会使用快速查找算法进行封装一个去重的集合。至于什么算法快，下面会讲给大家
上面的数组去重的重要应用，**我们需要一个集合用于存储一组数据，要求是集合中的每一位数据唯一**，意味着里面的值，每个值有且只有一个，最终得到是一个无重复数据的集合。

我们需要具有类似去重的思想的集合，介于在 JAVA 以及其他语言中出现了 set 结构，JS 中吸取相应的经验在 ES6 新的语法规范中引入了 Set 数据结构，用于返回一组不重复的数据。
接下来介绍的是 Set 的基本使用

Set 对象是值的集合，你可以按照插入的顺序迭代它的元素。 Set 中的元素只会出现一次，即 Set 中的元素是唯一的。
它的声明
new Set([iterable]);
其中 iterable 是一个可迭代对象，其中的所有元素都会被加入到 Set 中。null 被视作 undefined。也可以不传入[iterable]，通过其 add 方法来添加元素。

对于 Java 或者是 python 比较熟悉的同学可能会比较了解 set。它是 ES6 新增的有序列表集合，它不会包含重复项。

Set 的属性
Set.prototype.size:返回 Set 实例的成员数量。
Set.prototype.constructor:默认的构造 Set 函数。
Set 方法
add(value):添加某个值，返回 Set 结构本身。
delete(value):删除某个值，返回一个布尔值，表示删除成功。
has(value):返回一个布尔值，表示参数是否为 Set 的成员。
clear():清除所有成员，没有返回值。
keys() :返回一个键名的遍历器
values() :返回一个值的遍历器
entries() :返回一个键值对的遍历器
forEach():使用回调函数遍历每个成员

对于去重于是现在还能这么写

```
let arr = [1,2,3,2,'1',4,5,4]
let set = new Set(arr)
let arrUnique = [...set]
//arrUnique    [1,2,3,'1',4,5]
```

除了 Array.from，我们也可以通过扩展运算符来实现数组转化
而利用 Array 与 Set 的相互转化，还可以很容易地实现并集（Union）和交集（Intersect）

```
let a = new Set([1,2,4])
let b = new Set([4,3,2])
let uniqueArr = new Set([...a, ...b])
//    uniqueArr[1,2,3,4]
```

Set 结构与 Array 相比：
Set 中存储的元素是唯一的，而 Array 中可以存储重复的元素。
Set 中遍历元素的方式：Set 通过 for…of…，Array 可以使用 for…of,还可以使用 forEach，forin 等
Set 是集合，不能像数组用下标取值。

接下来说一说 map 结构。
在我们读书的时候，都会有目录，也就是索引，这个索引的目的是为了让我们更快度的查找到我们想要的内容。这不仅仅用于书中，我们手机通讯录也是有这样的索引。同样对于我们的程序来说也需要通过某种结构快速的查找到我们想要的内容。比如说，在我们前端当中，我们通过一个 id 就可以找到一个产品，以及对应的产品信息。或者我们通过电影标题就可以找到对应电影的信息，包括我们希望通过某些字段进行查询，用来查找我们想要的内容。这种需求在我们程序中用的地方有很多很多。
介于上面的例子，我们能看的出来查询方式为：

页码 ==》 内容
Id ==》 商品信息
电影标题 ==》 电影信息

是一种 “a ==》b” 的这一种形式，我们可以通过 A 去查找到 B。映射到我们计算机中来说是 key => value 结构，我们第一反应会想到 JavaScript 中的对象，对象里面都是这样的结构，由若干键值对构成。在我们的程序中想通过这种方式去进行查找，都会通过利用对象来完成。
js 中的对象是基于哈希表结构的，而哈希表的查找时间复杂度为 O(1)，所以很多人喜欢用对象来做映射，减少遍历循环。

比如常见的数组去重：

```
Array.prototype.unique = function () {
    var newArr = []
    var len = this.length
    var temp = {}
    for(var i=0; i<len; i++) {
      if(!temp[this[i]]) {
        newArr.push(this[i])
        temp[this[i]] = true
      }
    }
    return newArr
  }
```

这里使用了一个 temp 对象来保存出现过的元素，在循环中每次只需要判断当前元素是否在 temp 对象内即可判断出该元素是否已经出现过。

上面的代码看起来 OK，但有点经验的同学可能会说了，假如目标数组是[1,'1'], 这是 2 个不同类型元素，所以我们的期望值应该是原样输出的。但不幸的是，函数输出结果却是[1]。同理的还有 true、null 等，也就是说对象中的 key 在 obj[key]操作时都被自动转成了字符串类型。

所以，如果要区分出不同的类型的话，temp 里面的属性值就不能是一个简单的 true 了，而是要包含几种数据类型。比如可以是：

```
temp[this[i]]={};
temp[this[i]][(typeof temp[this[i]])] = 1;
temp[typeof this[i] + this[i]] = true
```

在判断的时候除了要判断键是否存在之外，也要判断对应的数据类型计数是否大于 1，以此来判断元素是否重复。这里可以设置为 true，但用数字的话还可以统计重复数量，扩展性更好。

另外，上面的代码语法也有点问题。

我们造的这个 temp 对象并不是完全空白，他是基于 Object 原型链继承而来的，所以自带了一个__proto__属性，如果你的目标数组里面恰好有"__proto__"这个值，返回的结果就有问题了，具体结果可以自己测试确认。

解决方法有两种：

1） 想办法去掉这个磨人的__proto__。显然，我们需要去掉原型链，那么可以使用`Object.create(null)`的方式来创建一个完全空白、无原型的空对

象。
2） 针对__proto__做特殊处理，专门统计它。

在真实世界中，我们描述一个事物最常用的方式是使用`属性`-`值`（`key`-`value`）这样的键值对数据，js 中的对象都是这种模式，这也是通常所说的字典结构。比如我们描述一个人是这样的：

Name: 丁丁
Age: 18
No: 666666
City: Harbin,
Weight: 100,
Height: 180

但有得时候我们的 key 能淡出是字符串，数字类型也是可以的，但是对于我们的 js 对象来说是不可以的。我们想要这种结构，自己去封装去应用，也是不断在期望有 js 中有这种规范独立对于对象，使得我们的 key 不限于字符串的拘束。再结合之前一些已有的语言，Java 中的 map 结构，同样在 ECMAScript6 中也提出了 map 结构，目的是为了我们存储更加方便更加快捷。而且对于我们存储的 key，并没有类型要求的这样的一个集合。
下面说一下 map 结构的基本使用
.ES6 提供了 Map 数据结构，它类似于对象，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，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适；

Map 常见的操作方法有：

set(key，val)：添加某个值，返回 Map 结构本身。
get(key)： 读取某个键，如果该键未知，则返回 undefined
delete(key)： 删除某个键，返回一个布尔值，表示删除是否成功。
has(key)： 返回一个布尔值，表示该值是否为 Map 的键。
clear() : 清除所有成员，没有返回值。

```
const m = new Map();
const o = { str : 'Hello World'};
m.set(o, 'content')
m.get(o) // "content"
m.has(o) // true
m.delete(o) // true
```

```
m.has(o) // false
```

只有对同一个对象的引用，Map 结构才将其视为同一个键

```
const map = new Map();

const k1 = ['a'];
const k2 = ['a'];

map.set(k1, 111).set(k2, 222);

map.get(k1) // 111
map.get(k2) // 222
```

上面例子表明，Map 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键，因为 k1 和 k2 是两个不同的对象，放在不同的内存地址中，所以 Map 视为不同的键

Map 结构原生提供三个遍历器生成函数和一个遍历方法。

keys()：返回键名的遍历器。
values()：返回键值的遍历器。
entries()：返回所有成员的遍历器。
forEach()：遍历 Map 的所有成员。
ps：Map 的遍历顺序就是插入顺序，这里就不示例了，大家自己动手实践一下。

可以使用扩展运算符（...）将 Map 转换为数组，反过来，将数组传入 Map 构造函数，就可以转为 Map 了

```
//Map 转数组
const map = new Map();
map.set('name' , 'hello').set({},'world');

[...map] //[["name","hello"],[{},"world"]]

//数组转 Map
const map = new Map([["name","hello"],[{},"world"]]);
```

map // {"name" => "hello", Object {} => "world"}

接下来我们来探究一下底层是怎么实现的，数据是如何存储的。

对于 Map 结构在计算机中怎么保存这样的数据呢？
计算机存储空间有两个属性：`存储地址`和所存储的`值`，机器可以根据给定的`存储地址`去读写该地址下的`值`。根据这种结构，假如我们将一块存储空间分成一个一个的格子，然后将这些数据依次塞到每个格子里，接下来我们就可以根据格子编号直接访问格子的内容了。这种方式就是数组（也叫线性连续表）：数组头保存整个数组储存空间的起始地址，不同下标代表不同的储存地址的偏移量，访问不同下标所对应的地址就能实现数组元素的读写。所以，很自然就会想到将上述的键值对数据的`key`映射成数组下标，接着读写数组就变成了读写`value`值。将`key`的字符串转换成代表下标数值比较简单，可以用特定的码表（如 ASCII）进行转换。
我们的 map 结构是

```
▶ 0: {"name" => "DD"}
▶ 1: {"age" => 18}
▶ 2: {"No" => 10086}
▶ 3: {"weight" => 100}
▶ 4: {"height" => 180}
```

映射到我们的内存中

```
{"name"(120) => "DD"}
{"age"(15) => 18}
{"No"(7) => 10086}
{"weight"(9) => 100}
{"height"(14) => 180}
```

括号里面的是内存地址。

由于 key 的值不同长度不一，所以转换后的下标的值也相差巨大，另外 key 的个数不确定，也就意味着下标的个数也有很大的范围，甚至无穷多，就有了下面的问题：

怎么将一组值相差范围巨大，个数波动范围很大的下标放入特定的数组空间呢？

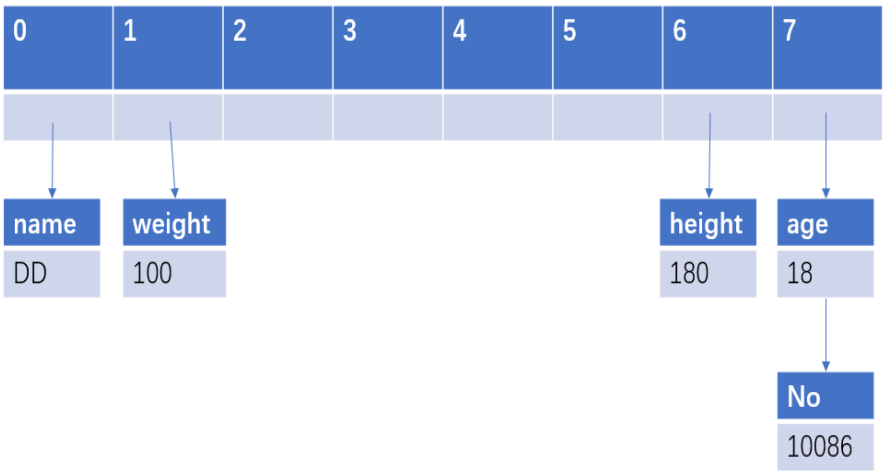
如果我们直接取下标值作为存储数组的下标，虽然简单，但是你会发现这个长度为 120 的数组只存了 5 个值，太浪费！如果我们想要缩短数组的长度，比如缩为 8 ，对于 hash 结构底层通常就为 2 的 n 次幂 一般去 8 或者 16，最简单的方式可以使用取模的方式来确定下标：120% 8、 = 0，15 % 8 = 7, 7%8 = 7 ……。这个取模就是哈希算法、也叫散列算法。

0 name	1 Weight	2	3	4	5	6 Height	7 Age
DD	100					180	18

但是这种方式很容易出现重复，age 和 No 对应的映射值分别为 15 跟 7，算出来的下标是 7 就重复了。也就是哈希算法中的冲突，也叫碰撞。好的哈希算法能极大减少冲突，但由于输入几乎是无穷的，而输出却要求在有限的空间内，所以冲突是不可避免的。

那如何处理冲突呢？

还是上面这个例子，29 和 69 发生了冲突，但是我们可以将他们组成一个链表，链表的头部放在数组中，得到。链表结构中，每个元素（除单向链表的尾部）都包含了相连元素的内存地址和本身的值，上文中的冲突放入一个链表中，可以得到这样的结构：



最终得到的这个数据结构，也就是我们常说哈希表了。这种将数组与链表结合生成哈希表的方法，叫拉链法，也叫邻接链表，数组中的每一位都是一个链表

哈希表数据的查找

比如想知道 name 属性，即 map.get('name')。流程是这样的：

1）根据字符映射关系得到映射值为 120

2）使用哈希算法得到下标 index=hash(120)=0

3）遍历数组中下标为 0 的链表，链表的第一个元素的 key 刚好就是我们要找的 name，所以返回 value 值 DD

哈希表中增删一个元素并不会影响到其他的元素，不像数组一样需要改变后面所有的元素下标。在拉链式的哈希表中，属性的增删就是链表的增删，非常方便。而在纯数组形式的哈希表中，对属性的删并不是真的删除，而是做一个空标志而已，所以不影响其他元素。

哈希算法（散列算法）

根据上面的例子得知，哈希算法的目的就是将不定的输入转换成特定范围的输出，并且要求输出尽量均匀分布。由于散列算法是应用在每一次数据定位中的，它的使用频率非常的高，这意味着我们必须要选择简单的算法。散列算法有很多，这里简单介绍几种。

1，除法散列法

最直观的一种，小茄上文使用的就是这种散列法，公式：

index = key % 16

2，平方散列法
求 index 是非常频繁的操作，而乘法的运算要比除法来得省时（对现在的 CPU 来说，估计我们感觉不出来），所以我们考虑把除法换成乘法和一个位移操作。公式：
index = (key * key) >> 28
如果数值分配比较均匀的话这种方法能得到不错的结果，另外 key 如果很大，key * key 会发生溢出。但我们这个乘法不关心溢出，因为我们根本不是为了获取相乘结果，而是为了获取 index。

3，斐波那契（Fibonacci）散列法
平方散列法的缺点是显而易见的，所以我们能不能找出一个理想的乘数，而不是拿 value 本身当作乘数呢？答案是肯定的。

1，对于 16 位整数而言，这个乘数是 40503
2，对于 32 位整数而言，这个乘数是 2654435769
3，对于 64 位整数而言，这个乘数是 11400714819323198485

这几个“理想乘数”是如何得出来的呢？这跟一个法则有关，叫黄金分割法则，而描述黄金分割法则的最经典表达式无疑就是著名的斐波那契数列，即如此形式的序列：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,233, 377, 610， 987, 1597, 2584, 4181, 6765, 10946，…。

对我们常见的 32 位整数而言，公式：
index = (key* 2654435769) >> 28

处理冲突的策略
上文介绍了拉链法来处理冲突，处理冲突的方法其实也有很多，下面简单介绍一下另外几种：

1）拉链法变种。由于链表的查找需要遍历，如果我们将链表换成树或者哈希表结构，那么就能大幅提高冲突元素的查找效率。不过这样做则会加大哈希表构造的复杂度，也就是构建哈希表时的效率会变差。

2）开放寻址： 当关键字 key 的哈希地址 p=H（key）出现冲突时，以 p 为基础，产生另一个哈希地址 p1，如果 p1 仍然冲突，再以 p 为基础，产

生另一个哈希地址 p2，…，直到找出一个不冲突的哈希地址 pi，将相应元素存入其中。这种方法有一个通用的函数形式：

Hi=（H（key）+di）% m i=1，2，…，n

根据 di 的不同，又可以分为线性的、平方的、随机数之类的。。。这里不再展开。

开发寻址的好处是存储空间更加紧凑，利用率高。但是这种方式让冲突元素之间产生了联系，在删除元素的时候，不能直接删除，否则就打乱了冲突元素的寻址链。

3）再哈希法

这种方法会预先定义一组哈希算法，发生冲突的时候，调用下一个哈希算法计算一直计算到不发生冲突的时候则插入元素，这种方法跟开放寻址的方法优缺点类似。函数表达式：

index=Hi（key） , i=1，2，…，n

接下来我们通过 js 去给大家利用 取模 hash 的方式去书写一个 map 结构

```
// 构造函数
function myMap() {
    this.init()
}
// 默认桶长
myMap.prototype.bucketLen = 8
// 设置初始化方法
myMap.prototype.init = function () {
    this.bucket = []
    for (let i = 0; i < this.bucketLen; i++) {
        this.bucket[i] = { type: `bucket_${i}`, next: null }
    }
```

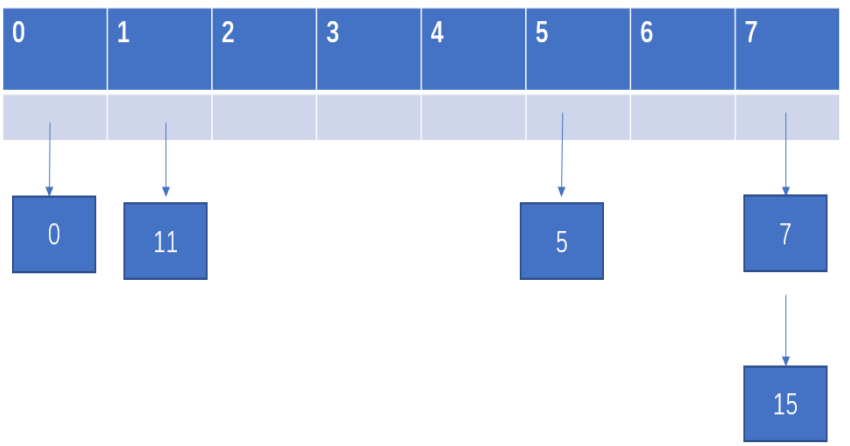
```
}
// 计算 hash
myMap.prototype.makeHash = function (str) {   // hash 分类算法，求和模 8（桶长）
    let hash = 0
    if (typeof str !== 'string') {   // 不是字符串，都转化成数字
        if (str == undefined || Object.is(str, NaN)) {
            hash = 0
        } else {
            hash = Number(str)
        }
    } else { // 是字符串，取前三位字符串取 Unicode 编码。
        for (let i = 0; i < 3 && i >= 0; i++) {
            hash += str[i] ? str[i].charCodeAt() : 0
        }
    }
    return hash % this.bucketLen //返回 hash 值（0-7）
}
// test：  myMap.prototype.makeHash('1')

myMap.prototype.set = function (key, value) {
    let len = this.makeHash(key) //算 hash
    let tempBucket = this.bucket[len] //取桶
    // 查找，插入
    while (tempBucket.next) {   // 判断当前链表的下一个位置有没有值
        // 如果有，判断当前节点 key 与你要插入的 key 时候相等
        if (tempBucket.next.key === key) {   // 相等
            tempBucket.next.value = value // 改 value
            return
        } else { // 不相等，向下找
            tempBucket = tempBucket.next
        }
    }
    //   没有直接后插
    tempBucket.next = { key, value, next: null }
}
myMap.prototype.get = function (key) {
```

```
    let len = this.makeHash(key) //算 hash
    let tempBucket = this.bucket[len] //取桶
    // 查找
    while (tempBucket) {
        if (tempBucket.key === key) {
            return tempBucket.value
        } else {
            tempBucket = tempBucket.next
        }
    }
    return undefined
}
myMap.prototype.has = function (key) {
    let len = this.makeHash(key) //算 hash
    let tempBucket = this.bucket[len] //取桶
    // 查找
    while (tempBucket) {
        if (tempBucket.key === key && (tempBucket.type === undefined)) {
            return true
        } else {
            tempBucket = tempBucket.next
        }
    }
    return false
}
myMap.prototype.delete = function (key) {
    let len = this.makeHash(key) //算 hash
    let tempBucket = this.bucket[len] //取桶
    // 查找，删除
    while (tempBucket.next) {   // 判断当前链表的下一个位置有没有值
        // 如果有，判断当前节点 key 与你要删除的 key 时候相等
        if (tempBucket.next.key === key) {   // 相等
            tempBucket.next = tempBucket.next.next // 把中间的对象干
掉
            return true
        } else { // 不相等，向下找
            tempBucket = tempBucket.next
```

```
            }
        }
        //    没有直接返回
        return false
}
myMap.prototype.clear = function () {
        this.init()
}
```

对于 set 结构与 map 结构类似，基本思想也是一样的。存储结构如下：




```
function mySet() {
        this.init()
}
// 设置桶长
mySet.prototype.bucketLen = 8
// 设置初始化方法
mySet.prototype.init = function () {
        this.bucket = []
        for (let i = 0; i < this.bucketLen; i++) {
                this.bucket[i] = { type: `bucket_${i}`, next: null }
        }
}
// 计算 hash
mySet.prototype.makeHash = function (str) {    // hash 分类算法，求和模 8（桶
```

```
长）
    let hash = 0
    if (typeof str !== 'string') {   // 不是字符串，都转化成数字
        if (str == undefined || Object.is(str, NaN)) {
            hash = 0
        } else {
            hash = Number(str)
        }
    } else { // 是字符串，取前三位字符串取 Unicode 编码。
        for (let i = 0; i < 3 && i >= 0; i++) {
            hash += str[i] ? str[i].charCodeAt() : 0
        }
    }
    return hash % this.bucketLen //返回 hash 值（0-7）
}
// test：  mySet.prototype.makeHash('1')

mySet.prototype.add = function (key) {
    let len = this.makeHash(key) //算 hash
    let tempBucket = this.bucket[len] //取桶
    // 查找，插入
    while (tempBucket.next) {   // 判断当前链表的下一个位置有没有值
        // 如果有，判断当前节点 key 与你要插入的 key 时候相等
        if (tempBucket.next.key === key) {   // 相等 什么都不做
            return
        } else { // 不相等，向下找
            tempBucket = tempBucket.next
        }
    }
    //   没有直接后插
    tempBucket.next = { key, next: null }
}

mySet.prototype.has = function (key) {
    let len = this.makeHash(key) //算 hash
    let tempBucket = this.bucket[len] //取桶
    // 查找
```

```
        while (tempBucket) {
            if (tempBucket.key === key && (tempBucket.type === undefined)) {
                return true
            } else {
                tempBucket = tempBucket.next
            }
        }
        return false
}
mySet.prototype.delete = function (key) {
        let len = this.makeHash(key) //算 hash
        let tempBucket = this.bucket[len] //取桶
        // 查找，删除
        while (tempBucket.next) {   // 判断当前链表的下一个位置有没有值
            // 如果有，判断当前节点 key 与你要删除的 key 时候相等
            if (tempBucket.next.key === key) {   // 相等
                tempBucket.next = tempBucket.next.next // 把中间的对象干
掉
                return true
            } else { // 不相等，向下找
                tempBucket = tempBucket.next
            }
        }
        //    没有直接返回
        return false
}
mySet.prototype.clear = function () {
        this.init()
}
```