# 15 | 如何通过哈希查找JS对象内存地址？

2022-10-22 石川 来自北京



《JavaScript进阶实战课》

课程介绍 >

讲述：石川

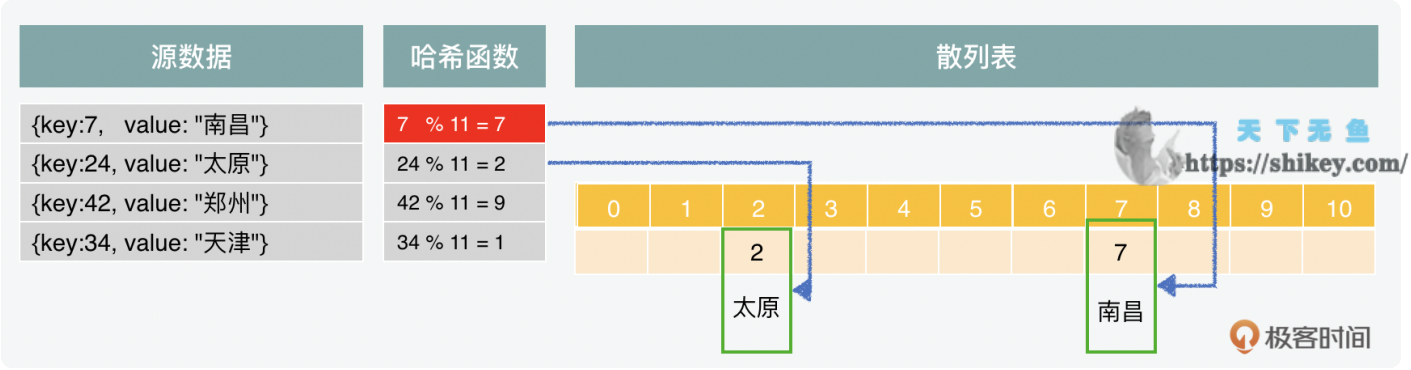
时长 11:29　大小 10.49M

你好，我是石川。

我们曾经讲过，在 Javascript 中，对象在调用栈中只做引用不做存储，实际的存储是在堆里面实现的。那么我们如何查找对象在堆里的实际存储地址呢？通过我们对字典的了解，这个问题就迎刃而解了。字典也被称作映射、符号表或关联数组，这么说可能比较抽象，所以我们先来说说字典的一种实现：散列表。

## 散列表：如何检查单词是否存在

如果你用过一些文档编辑的软件，应该很常用的一个功能就是拼写检查，这个检查是怎么做到的呢？从它的最底层逻辑来说，就是看一个单词存在与否。那么一个单词是否存在是如何判断的呢？这里就需要用到散列表。散列表的实现逻辑就是基于每个单词都生成一个唯一的哈希值，把这些值存放在一个数组中。当我们想查询一个词是否有效，就看这个词的哈希值在数组中是否存在即可。



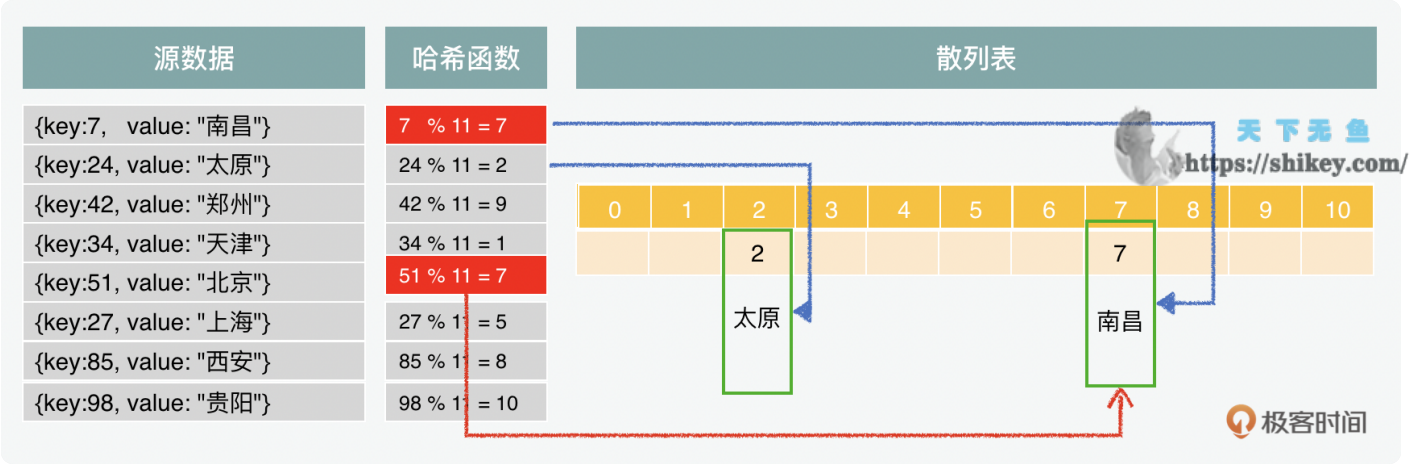

假设我们有上图中这样的一组城市的键值对组成的对象，我们可以看出，在哈希的过程中，一个城市的键名，通过一个哈希函数，生成一个对应的唯一的哈希值，这个值被放到数组中，形成一个哈希列表。下次，当我们想要访问其中数据的时候，就会通过对这个列表的遍历来查询相关的值。

这里我们可以看到，图中间位置的是哈希函数，我们需要一个哈希函数来生成哈希值，那么哈希值是怎么生成的呢？生成散列表中的哈希值有很多种方式，比如素数哈希、ASCII 哈希，还有 djb2 等方式。

在哈希算法当中，最基础的就是素数（prime number）哈希。这里我们把一个素数作为模数（modulus number），来给你举一个例子，在这个例子里，我们把 11 这个素数作为了模数，用下面的一组键值对中的键除以模数，所获得的余数，放到一个数组中。就形成了一个散列表。这样可以获得一个统一的索引。

```
{key:7, value: "南昌"}
{key:24, value: "太原"}
{key:42, value: "郑州"}
Prime number: 11
7 % 11  = 7 // 余数为7
24 % 11 = 2 // 余数为2
42 % 11 = 9 // 余数为9
```

这个方式看似可以用来生成哈希值，但是也存在一个问题。在将余数放入数组的过程中，我们会发现，如果处理的数据数量足够多，那么就会出现冲突的情况，比如下图中标红的两个对象的键除以素数 11 的余数是相同的，7 和 51 的余数都是 7，这样就会造成冲突。一个完美的哈希表是不应该存在冲突的，可是这样完美的哈希表其实在现实中并不存在，所以我们只能尽量减少这种情况。




为了尽量减少这种冲突，业界也在尝试其他办法，比如使用 ASCII code 和素数结合来生成哈希，但这种方式和上面的素数哈希一样，即使结合了 ASCII，哈希值也不能完全避免碰撞的产生，只能减少冲突。

复制代码

```
asciiHashCode(key) {
  if (typeof key === 'number') {
    return key;
  }
  const tableKey = this.toStrFn(key);
  let hash = 0;
  for (let i = 0; i < tableKey.length; i++) {
    hash += tableKey.charCodeAt(i);
  }
  return hash % 37;
}
```

除此之外，还有一种经典的 djb2 的算法，可以用来进一步减少这种问题的发生。它的做法是先用一个长质数 5381 作为哈希数，然后根据字符串长度循环，将哈希数乘以 33，再加上字符的 ASCII 码迭代。结果和模数 1013 的余数结果就是最后的哈希值。

这里你可能会问，33 和 5381 这两个数字是什么意思？这里乘以 33 呢，是因为更易于移位和加法计算。使用 33 可以复制累加器中的大多数输入位，然后将这些位分散开来。5 的移位和 32 是互素的，这有助于雪崩。ASCII 可以看做是 2 个 4 位字符类型选择器，比如说，数字的前四位都是 0x3。所以 2、4、8 位都可能导致相似的位之间交互，而 5 位可以让一个字符中许多的 4 个低位与 4 个高位强烈交互。所以这就是选择 33 的原因。那么至于原则 5381 作为质数呢，则更多是一种习惯，也可以由其它大的质数代替。

复制代码

```
djb2HashCode(key) {
  const tableKey = this.toStrFn(key);
  let hash = 5381;
  for (let i = 0; i < tableKey.length; i++) {
    hash = (hash * 33) + tableKey.charCodeAt(i);
  }
  return hash % 1013;
}
```



这几种方式只是给你一个概念，实际上哈希函数的实现方法还有很多，可能专门一本书都不一定能讲完，但是在这里，我们只是为了了解它的原理和概念。

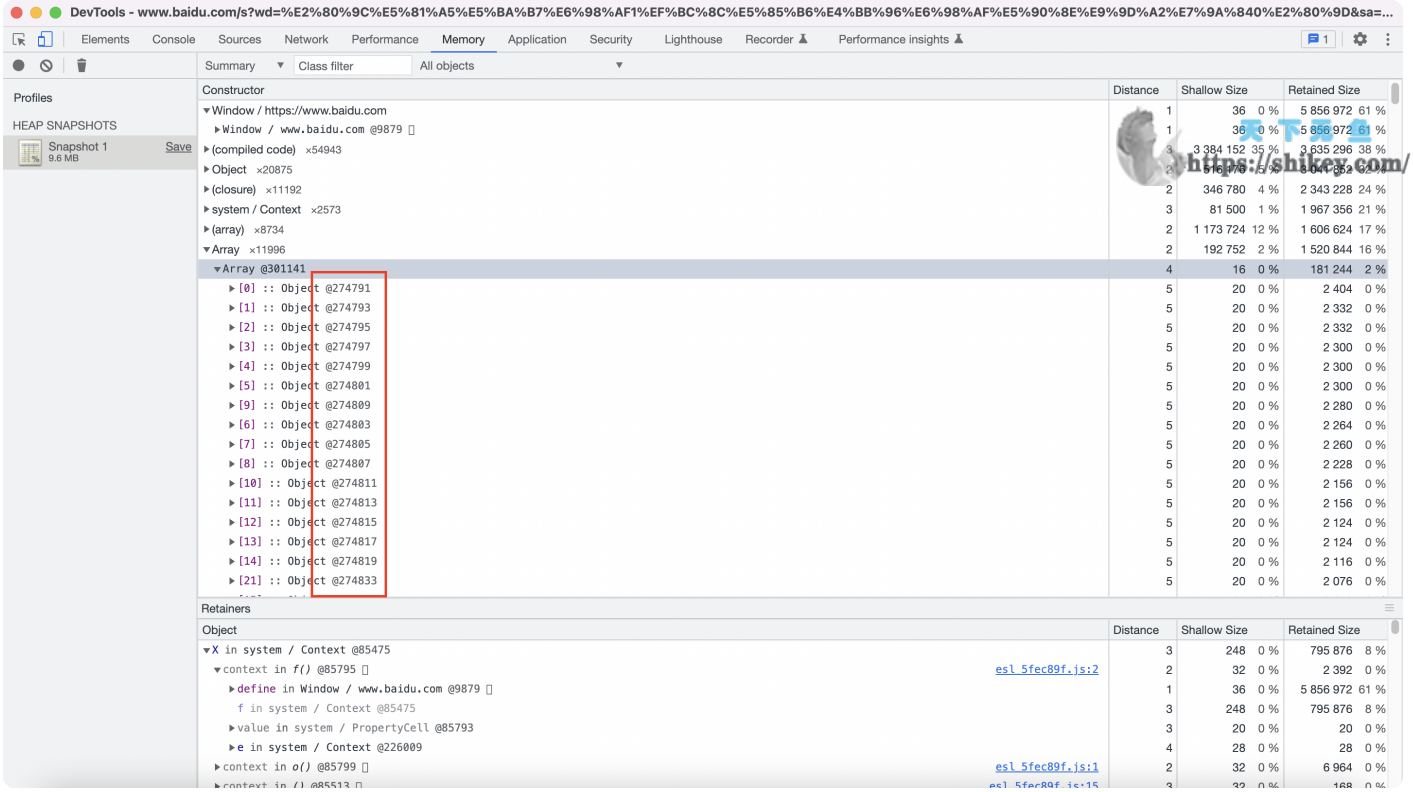
通过哈希函数，我们基本可以让一个单词在哈希表中找到自己的存在了。那么解决了存在的问题后，单词就该问“我的意义是什么？”，这个时候，我们就需要字典的出场了。

## 字典：如何查找对象的内存地址

散列表可以只有值，没有键，可以说是数组延伸出来的索引。说完散列表，我们再来看看字典（dictionary）。顾名思义，这种数据结构和我们平时用的字典类似，它和索引的主要的作用都是快速地查询和搜索。但是我们查字典的时候，关心的不光是有没有这个词，更重要的是，我们要知道这个单词对应的意思。所以我们需要通过一组的键值对来表明它们的关系。

我们设想一个很初级的字典，就是每个字都有一个哈希作为键名，它的意思作为键值。这样一条条地放到每一页，最后形成一个字典。所以通常字典是有键值对的。所以我们前面说过，字典作为一种数据结构，又叫做映射（map）、符号表（symbol table）或者关联数组（associative array）。在 JavaScript 中，我们其实可以把对象看做是一种可以用来构建字典的一种散列表，因为对象里就包含 key-value 的属性。

回到我们开篇的问题，在前端，最常见的字典就是我们使用的对象引用了。我们用的浏览器和 JS 引擎会在调用栈中引用对象，那么对象在堆中的实际位置如何寻找呢？这里，**对象在栈的引用和它在堆中的实际存储间的关联就是通过地址映射来实现的。这种映射关系就是通过字典来存储的。**我们可以用浏览器打开任何一个页面，然后打开开发者工具，在工具里，我们进入内存的标签页，然后选择获取一个堆的快照，之后我们便可以看到对象旁边的 @后面的一串数字，这串数字就是对象在内存的地址。所以这里的字典就是一部地址簿。




## Map 和 Set：各解决什么问题

在 ES6 之前，JavaScript 中只有数组和对象，并没有字典这种数据结构。在 ES6 之后才引进了 Map 和 Set 的概念。

JavaScript 中的 Map 就是字典的结构，它里面包含的就是键值对。那你可能会问，它和对象有什么区别？我们说过对象就是一个可以用来实现字典的支持键值对的散列表。Map 和对象最大的区别就是 Map 里的键可以是字符串以外的其它数据结构，比如对象也可以是一个键名。

JavaScript 中的 Set 就是集合的结构，它里面包含值，没有键。这里你也可能会问，那这种结构和数组有什么区别？它的区别主要在于 JS 中的集合属于无序集合，并且里面不能有相同的元素。

JavaScript 同时还提供了 WeakMap 或 WeakSet，用它们主要有 2 个原因。第一，它们都是弱类型，代表没有键的强引用。所以 JavaScript 在垃圾回收时可以清理掉整条记录。第二个原因，也是它的特点，在于既然 WeakMap 里没有键值的迭代，只能通过钥匙才能取到相关的值，所以保证了内部的封装私有属性。这也是为什么我们前面07 讲说到对象的私有属性的时候，可以用 WeakMap 来存储。

## 散列冲突：解决哈希碰撞的方式

其实解决哈希碰撞的几种方式也值得了解。上面我们已经介绍了几种通过哈希函数算法角度解决哈希碰撞的方式。下面，我们再来看看通过数据结构的方式是如何解决哈希冲突的。这里有几种基础方式，包含了线性探查法、平方探测法和二度哈希法。


### 线性探查法

我们先来说说线性探查法。用这种方式的话，当一个散列碰撞发生时，程序会继续往下去找下一个空位置，比如在之前例子中，7 被南昌占用了，北京就会顺移到 8。这样在存储的时候问题也许不大，但是在查找的时候会有一定的问题，比如当我们想要查找某个数据的时候，则需要在集群中迭代寻找。

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 98 | 34 | 24 | | | 27 | | 7 | 51 | 42 | 85 |
| 贵阳 | 天津 | 太原 | | | 上海 | | 南昌 | 北京 | 郑州 | 西安 |

### 平方探测法

另外一种方式就是平方探测法。平方探测法用平方值来代替线性探查法中的往后顺移一位的方式，这样就可以做到基于有效的指数做更平均的分布。

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 34 | 24 | | | 27 | | 7 | 51 | 42 | 98 | | | 85 |
| | 天津 | 太原 | | | 上海 | | 南昌 | 北京 | 郑州 | 贵阳 | | | 西安 |

### 二度哈希法

第三种方式是二度哈希（Rehashing/Double-Hashing），也就是在第一次的哈希的基础上再次哈希。在下面公式里，x 是第一次哈希的结果，R 小于哈希表。假设每次迭代序列号是 i，每次哈希碰撞通过 i * hash2(x) 来解决。

复制代码

```
hash2(x) = R - (x % R)
```

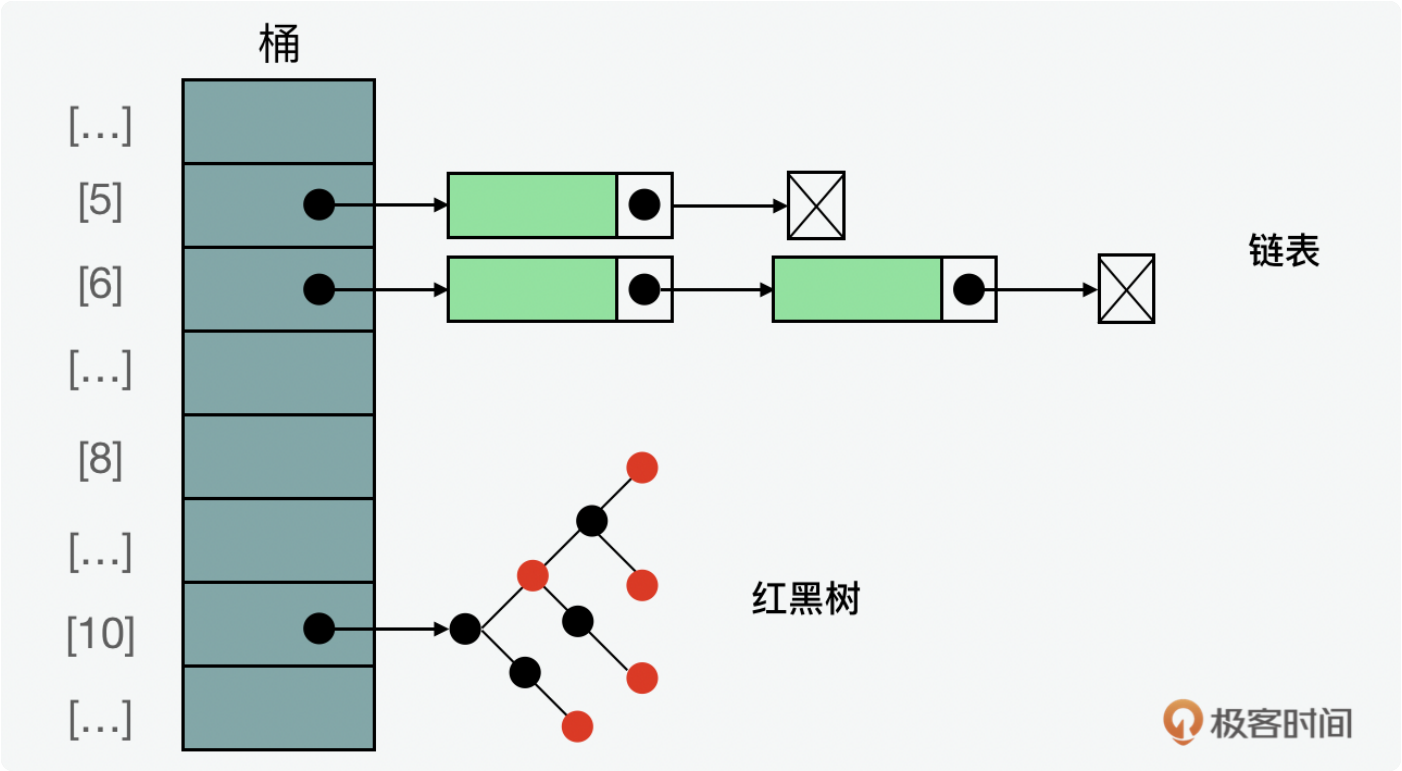
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 85 | 7 | 51 | 27 | | | 24 | 34 | 42 | 98 |
| | 西安 | 南昌 | 北京 | 上海 | | | 太原 | 天津 | 郑州 | 贵阳 |



## HashMap：Java 是如何解决散列冲突的？

先把 JS 放在一边，其实我们也可以通过 Java 语言里一些更高阶的链式数据结构，来更深入了解下哈希碰撞的解决方式。如果你学过 Java，可能有用到过 HashMap、LinkedHashMap 和 TreeMap。那么 Java 中的 HashMap 和 LinkedHashMap，那么 Java 中的这些数据结构有什么优势，分别是如何实现的？下面，我们可以来看看。

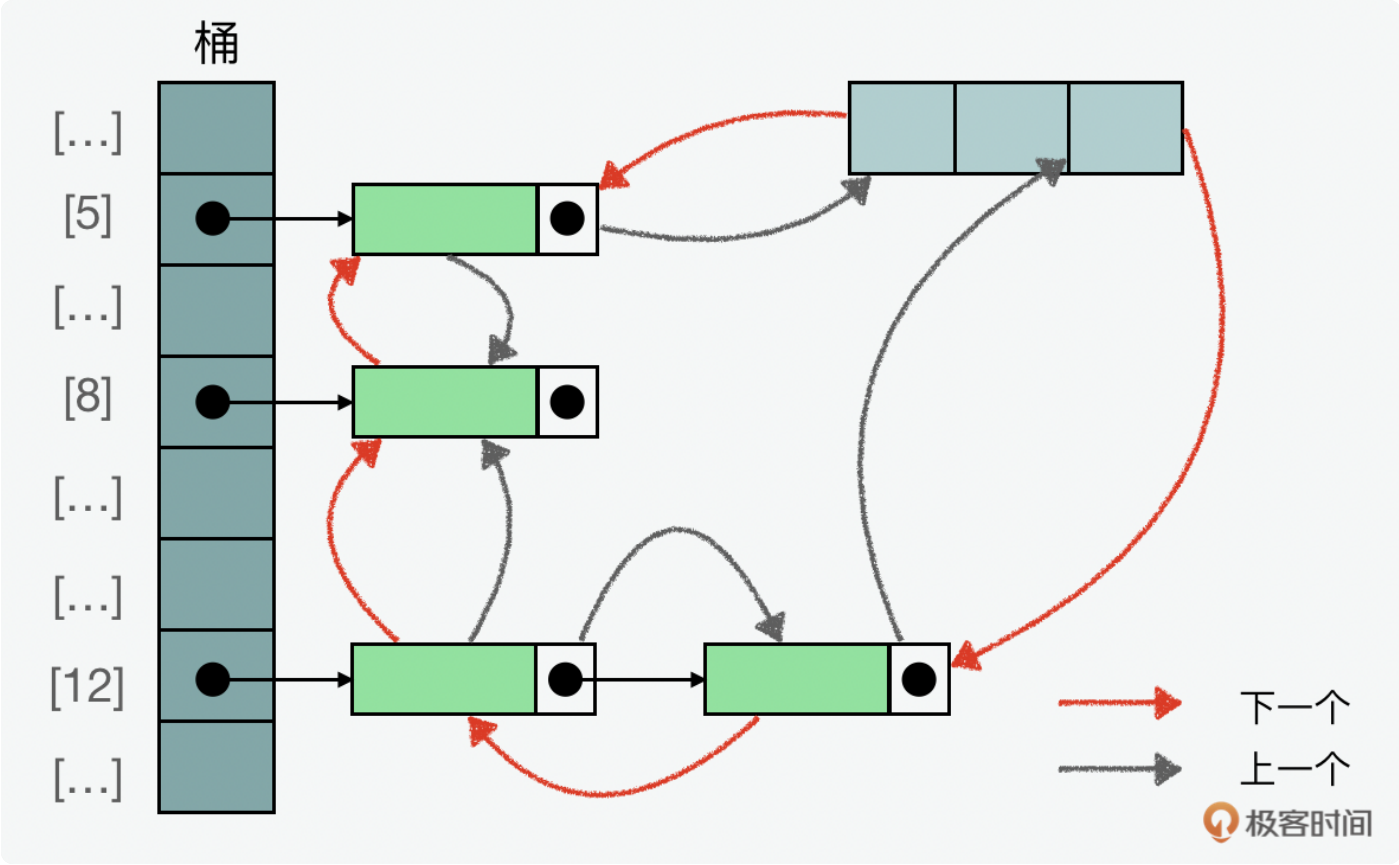
HashMap 的底层逻辑是通过链表和红黑树实现的。它最主要解决的问题就是哈希碰撞。我们先来说说链表。它的规则是，当哈希函数生成的哈希值有冲突的时候，就把有冲突的数据放到一个链表中，以此来解决哈希碰撞。那你可能会问，既然链表已经解决了这个问题，为什么还需要用到红黑树？这是因为当链表中元素的长度比较小的时候，链表性能还是可以的，但是当冲突的数据过多的时候，它就会产生性能上的问题，这个时候用增删改查的红黑树来代替会更合适。




## 散列加链表：基于双链表存值排序

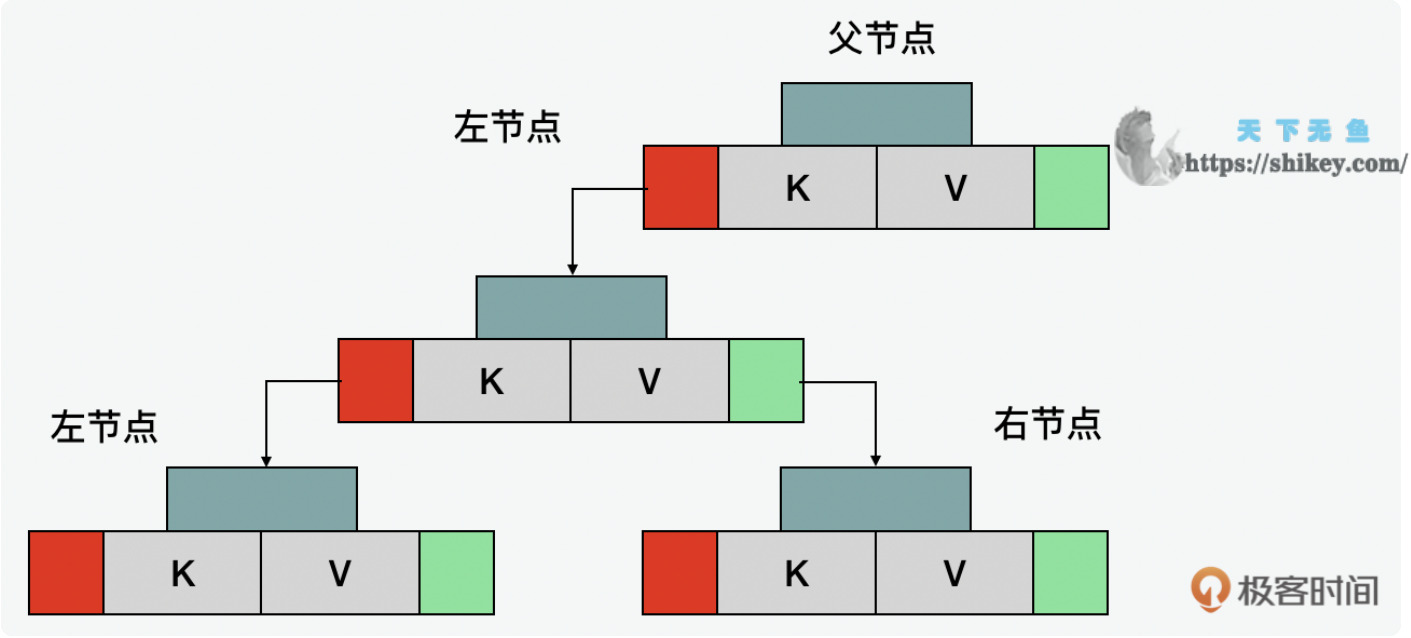
了解完 HashMap，再来看看 LinkedHashMap。LinkedHashMap 是在 HashMap 的基础上，内部维持了一个双向链表（Doubly Linked List），它利用了双向链表的性能特点，可以起到另外一个非常重要的作用，就是可以保持插入的数据元素的顺序。






## TreeMap：基于红黑树的键值排序

除了 HashMap 和 LinkedHashMap，TreeMap 也是 Java 一种基于红黑树实现的字典，但是它和 HashMap 有着本质的不同，它完全不是基于散列表的。而是基于红黑树来实现的。相比 HashMap 的无序和 LinkedHashMap 的存值有序，TreeMap 实现的是键值有序。它的查询效率不如 HashMap 和 LinkedHashMap，但是相比前两者，它是线程安全的。




## 总结

通过这节课，你应该了解了如何通过哈希查找 JS 对象的内存地址。不过，更重要的是希望通过今天的学习，你也能更好地理解哈希、散列表、字典，这些初学者都比较容易混淆的概念。最后我们再来总结下吧。我们说字典（dictionary）也被称为映射、符号表或关联数组，哈希表（hash table）是它的一种实现方式。在 ES6 之后，随着字典（Map）这种数据结构的引入，可以用来实现字典。集合（Set）和映射类似，但是区别是集合只保存值，不保存键。举个例子，这就好比一个只有单词，没有解释的字典。

## 思考题

今天的思考题是，我们知道 Map 是在 ES6 之后才引入的，在此之前，人们如果想实现类似字典的数据结构和功能会通过对象数据类型，那你能不能用对象手动实现一个字典的数据结构和相关的方法呢？来动手试试吧。

欢迎在留言区分享你的答案、交流学习心得或者提出问题，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。我们下节再见！

上一篇 14 | 通过SparkPlug深入了解调用栈

## 精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。