17 | 如何通过链表做LRU/LFU缓存？

2022-10-27 石川 来自北京

《JavaScript进阶实战课》

课程介绍 >

讲述：石川

时长 09:15　大小 8.45M

你好，我是石川。

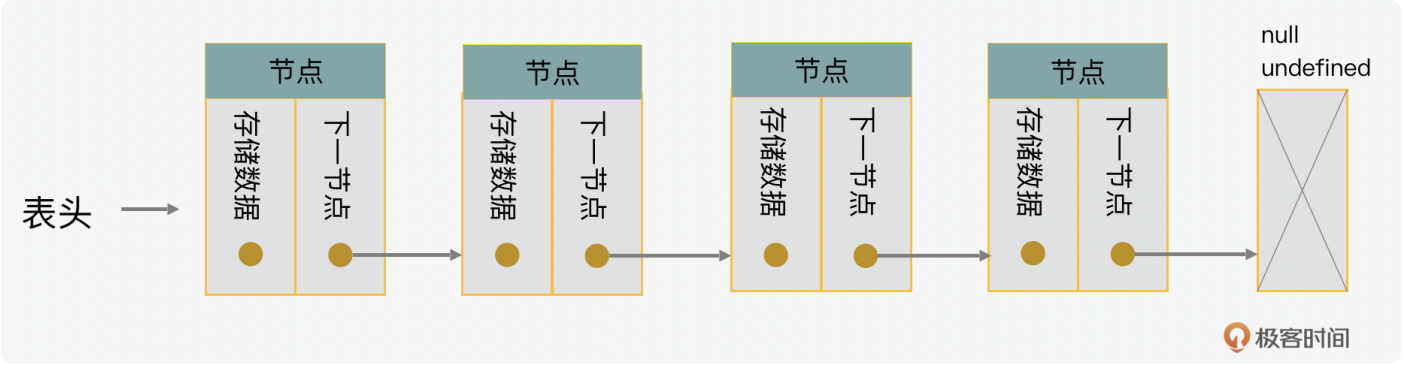
前面我们在第 10-13 讲讲过了数组这种数据类型，以及通过它衍生出的栈和队列的数据结构。之后，我们在讲到散列表的时候，曾经提到过一种链表的数据结构。今天，我们就来深入了解下链表和它所支持的相关的缓存算法。链表有很多使用场景，最火的例子当属目前热门的区块链了。它就是用了链表的思想，每一个区块儿之间都是通过链表的方式相连的。在链表结构里，每一个元素都是节点，每个节点有自己的内容和相连的下一个元素的地址参考。

既然我们讲的是 JavaScript，还是回到身边的例子，就是缓存。无论是我们的操作系统，还是浏览器，都离不开缓存。我们把链表和散列表结合起来，就可以应用于我们常说的缓存。那这种缓存是如何实现的呢？接下来就先从链表说起吧。

如何实现单双循环链表

单链表

下面我们先来看看一个单向链表是怎么实现的。链表就是把零散的节点（node）串联起来的一种数据结构。在每个节点里，会有两个核心元素，一个是数据，一个是下一个节点的地址，我们称之为后继指针（next）。在下面的例子里，我们先创建了一个节点（node），里面有节点的数据（data）和下一个节点的地址（next）。



在实现上，我们可以先创建一个节点的函数类，里面包含存储数据和下一节点两个属性。

复制代码

```
class Node {
  constructor(data){
    this.data = data;
    this.next = null;
  }
}
```

之后，我们再创建一个链表的函数类。在链表里，最常见的方法就是头尾的插入和删除，这几项操作的复杂度都是 $O(1)$。应用在缓存当中的时候呢，通常是头部的插入，尾部删除。但倘若你想从前往后遍历的话，或是在任意位置做删除或插入的操作，那么相应的复杂度就会是 $O(n)$。这里的重点不是实现，而是要了解链表里面的主要功能。因此我没有把所有代码都贴在这里，但是我们可以看到它核心的结构。

复制代码

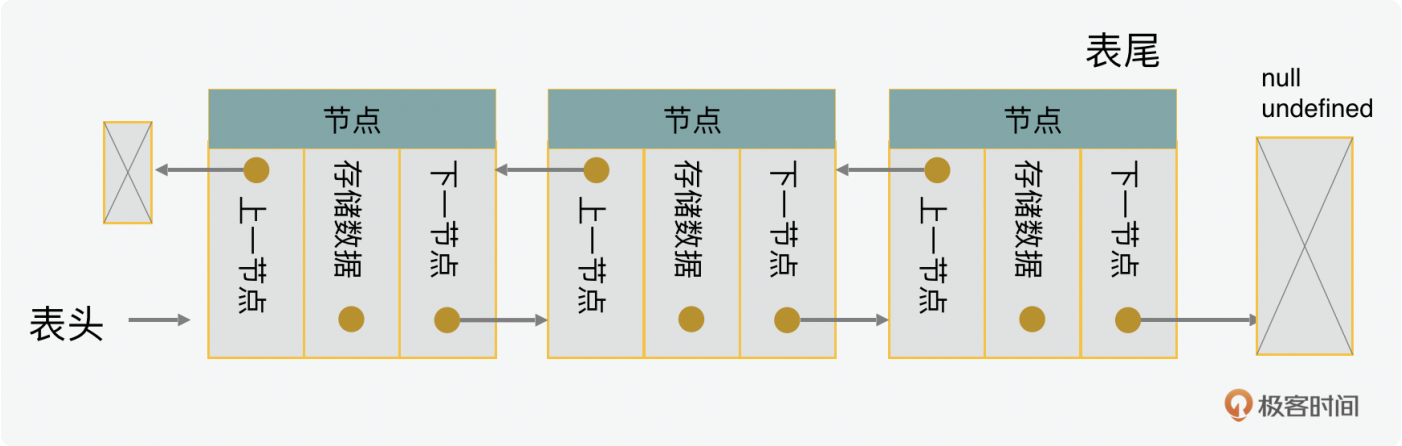
```
class LinkedList {
  constructor(){
    this.head = null;
    this.size = 0;
  }
  isEmpty(){ /*判断是否为空*/ }
  size() { /*获取链表大小*/ }
  getHead() { /*获取链表头节点*/ }
```

```
  indexOf(element) { /*获取某个元素的位置*/ }
  insertAtTail(element) { /*在表尾插入元素*/ }
  insertAt(element, index) { /*在指定位置插入*/ }
  remove(element) { /*删除某个元素*/ }
  removeAt(index) { /*在指定位置删除*/ }
  getElementAt(index) { /*根据某个位置获取元素*/ }
  removeAtHead(){ /*在表头位置删除元素*/ }
}
```


注意观察的你，可能会发现，最后一个节点是 null 或 undefined 的，那它是做什么的呢？这个是一个哨兵节点，它的目的啊，是对插入和删除性能的优化。因为在这种情况下，插入和删除首尾节点的操作就可以和处理中间节点用同样的代码来实现。在小规模的数据中，这种区别可能显得微不足道。可是在处理缓存等需要大量频繁的增删改查的操作中，就会有很大的影响。

双链表

说完单链表，我们再来看看双链表。在前面单链表的例子里，我们可以看到它是单向的，也就是每一个节点都有下一个节点的地址，但是没有上一个的节点的指针。双链表的意思就是双向的，也就是说我们也可以顺着最后一个节点中关于上一个节点的地址，从后往前找到第一个节点。



所以在双链表节点的实现上，我们可以在单链表基础上增加一个上一节点指针的属性。同样的，双链表也可以基于单链表扩展，在里面加一个表尾节点。对于从后往前的遍历，和从前往后的遍历一样，复杂度都是 $O(n)$。

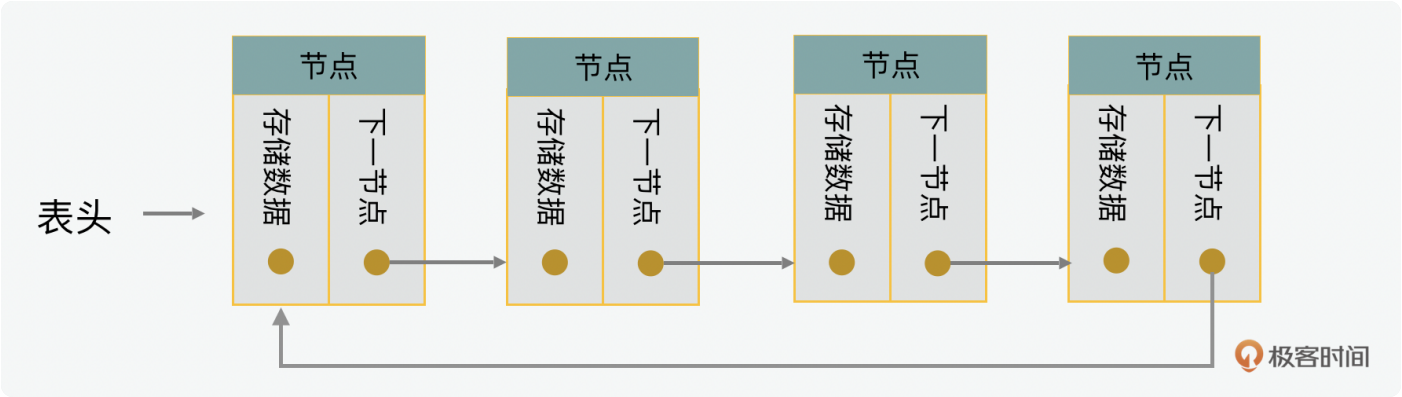
复制代码

```
class DoublyNode extends Node {
  constructor(data, next, prev) {
    super(data, next);
    this.prev = prev;
  }
```

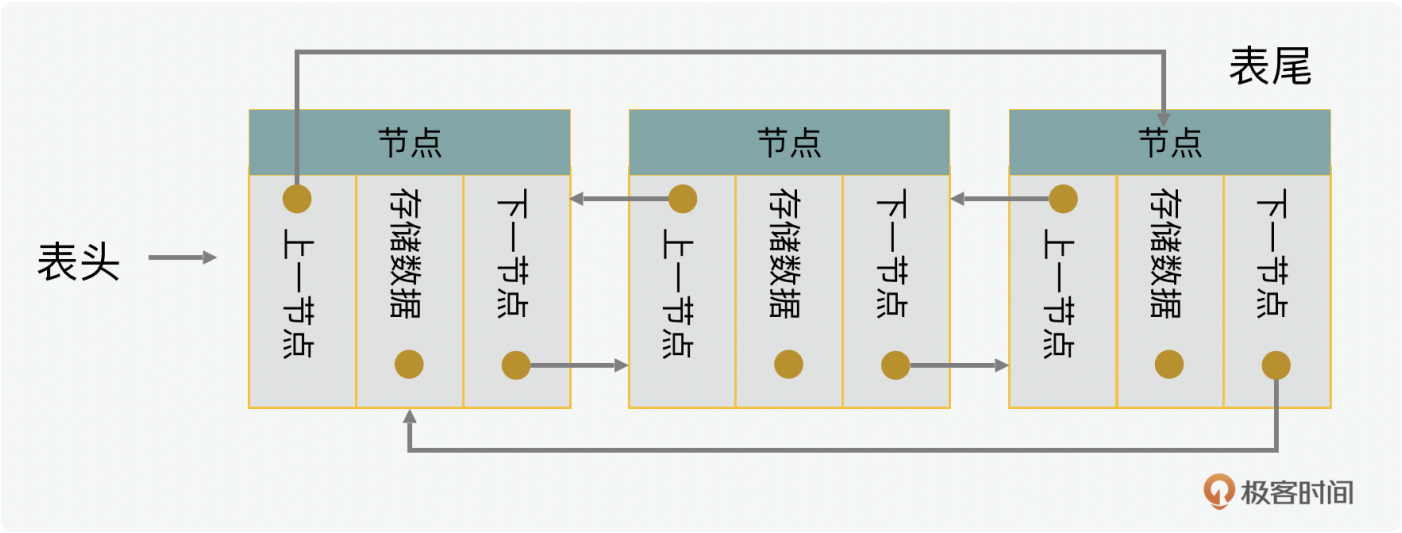
```
}

class DoublyLinkedList extends LinkedList {
  constructor() {
    this.tail = undefined;
  }
}
```


循环链表

我们接下来再来看看循环链表（circular list）。循环链表，顾名思义，就是一个头尾相连的链表。



如果是双向循环列表的话，就是除了顺时针的头尾相接以外，从逆时针也可以循环的双循环链表。



如何通过链表来做缓存

了解了单双循环链表后，现在我们回到题目，那我们如何能通过链表来做缓存呢？我们先了解下缓存的目的。缓存主要做的就是把数据放到临时的内存空间，便于再次访问。比如数据库会

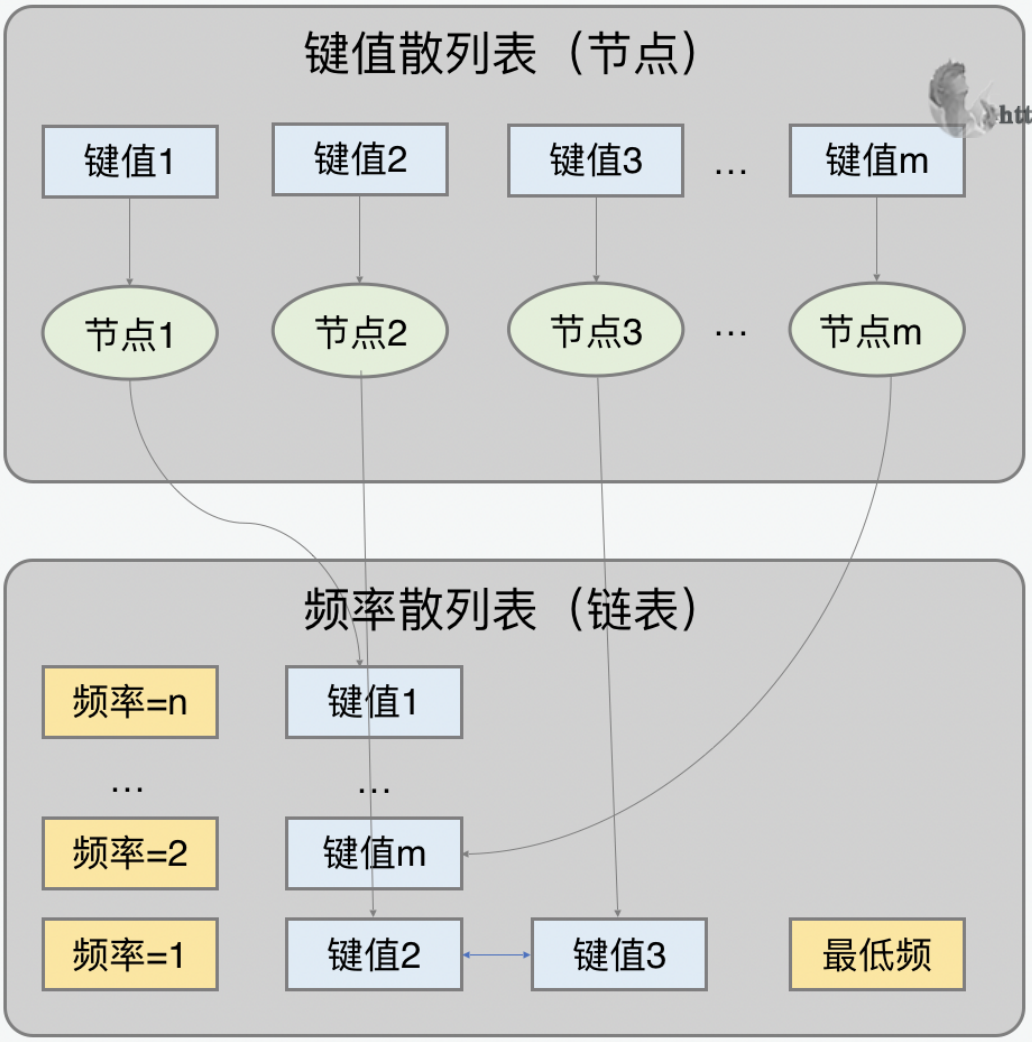
有缓存，目的是减少对硬盘的访问，我们的浏览器也有缓存，目的是再次访问页面的时候不用重新下载相关的文本、图片或其它素材。通过链表，我们可以做到缓存。下面，我们会看两种缓存的算法，他们是最近最少使用（LRU，least recently used）和最不经常使用（LFU，least frequently used），简称 LRU 缓存和 LFU 缓存。

一个最优的缓存方案应该把最不可能在未来访问的内容换成最有可能被访问的新元素。但是在实际情况中，这是不可能的，因为没人能预测未来。所以作为一个次优解，在缓存当中，通常会考虑两个因素，一个是时间局部性（Temporal Locality），我们认为一个最近被访问的内存位有可能被再次访问；另外一个因素是空间局部性 (Spatial Locality)，空间局部性考虑的是一个最近被访问的内存位相近的位置也有可能被再次访问。

在实际的使用中，LRU 缓存要比 LFU 的使用要多，你可能要问为什么？这里我先卖个关子，咱们先看看这两种方法的核心机制和实现逻辑。之后我们再回到这个问题。

LFU 缓存

我们先来看看 LFU 缓存。一个经典的 LFU 缓存中，有两个散列表，一个是键值散列表，里面存放的是节点。还有一个是频率散列表，里面根据每个访问频率，会有一个双链表，如下图中所示，如果键值为 2 和 3 的两个节点内容都是被访问一次，那么他们在频率 1 下面会按照访问顺序被加到链表中。



在这里，我们看一个 LFU 关键的代码实现部分的说明。LFU 双链表节点，可以通过继承双链表的节点类，在里面增加需要用到的键值，除此之外还有一个计数器来计算一个元素被获取和设置的频率。

复制代码

```
class LFUNode extends DoublyNode {
  constructor(key) {
    this.key = key;
    this.freqCount = 1;
  }
}

class LFUDoublyLinkedList extends LinkedList {
  constructor() {/* LFU 双链表 */ }
}
```

前面说到一个 LFU 缓存里面有两个散列表，一个是键值散列表，一个是频率散列表。这两个散列表都可以通过对象来创建。键值散列表保存着每个节点的实例。频率散列表里有从 1 到 n 的频率的键名，被访问和设置次数最多的内容，就是 n。频率散列表里每一个键值都是一个双向链表。

那围绕它呢，这里有 3 个比较核心的操作场景，分别是插入、更新和读取。

对于插入的场景，也就是**插入新的节点**，这里我们要看缓存是否已经满了，如果是，那么在插入时它的频率是 1。如果没有满的话，那么新的元素在插入的同时，尾部的元素就会被删除。

对于更新场景，也就是**更新旧的节点**，这时，这个元素会被移动到表头。同时，为了计算下一个被删除的元素，最小的频率 minFreq 会减 1。

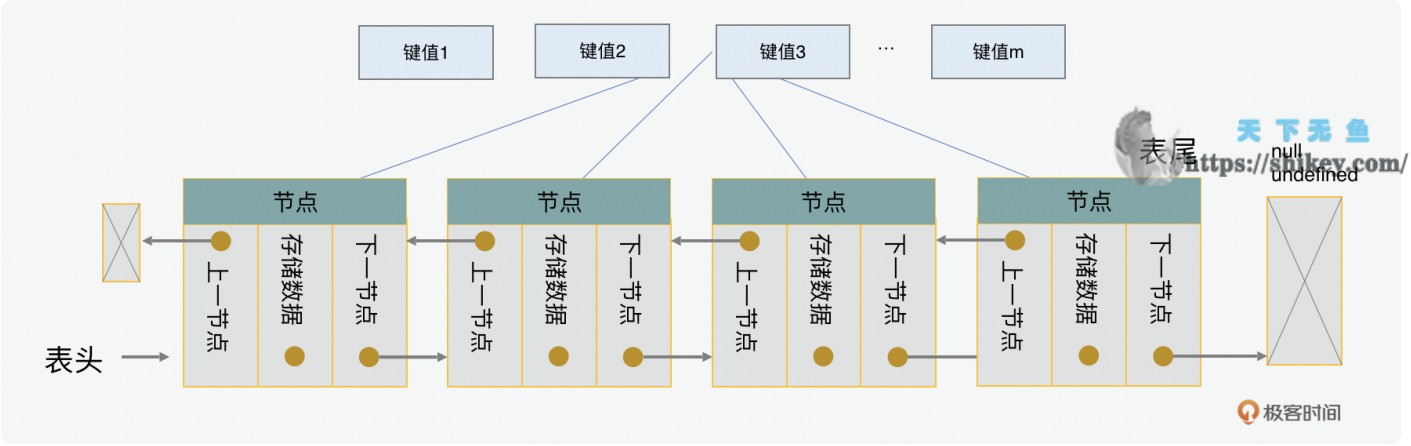
对于读取场景，也就是**获取节点的动作**，缓存可以返回节点，并且增加它的调用频率的计数，同时将这个元素移动到双链表的头部。同样与插入场景类似，最后一步，最低频率 minFreq 的值会被调整，用来计算下次操作中会被取代的元素。

复制代码

```
class LFUCache {
  constructor() {
    this.keys = {}; // 用来存储LFU节点的键值散列表
    this.freq = {}; // 用来存储LFU链表的频率散列表
    this.capacity = capacity; // 用来定义缓存的承载量
    this.minFreq = 0; // 把最低访问频率初始设置为0
    this.size = 0; // 把缓存大小初始设置为0
  }
  set() {/* 插入一个节点 */}
  get() {/* 读取一个节点 */}
}
```

LRU 缓存

以上是 LFU 的机制和核心实现逻辑，下面我们再来看看最近最少使用（LRU least recently used）。LRU 缓存的目的是在有限的内存空间中，当有新的内容需要缓存的时候，清除最老的内容缓存。当一个缓存中的元素被访问了，这个最新的元素就会被放到列表的最后。当一个没有在缓存中的内容被访问的时候，最前面的也就是最老的内容就会被删除。



在 LRU 缓存的实现中，最需要注意的是要追踪哪个节点在什么时间被访问了。为了达到这个目的，我们同样要用到链表和散列表。之所以要用到双向链表，是因为需要追踪在头部需要删除的最老的内容，和在尾部插入最新被访问的内容。

从实现的角度看，LRU 当中的节点和 LFU 没有特别大的差别，也需要一个键值散列表，但是不需要频率散列表。LRU 缓存可以通过传入承载量 capacity 参数来定义可以允许缓存的量。同样和 LFU 类似的，LRU 也需要在链表头部删除节点和链表尾部增加节点的功能。在此基础之上，有着获取和设置的两个对外使用的方法。所以总体看来，LRU 的实现和 LFU 相比是简化了的。

复制代码

```
class LRUNode extends DoublyNode {
  constructor(key) {
    this.key = key;
  }
}

class LRUCache {
  constructor() {
    this.keys = {}; // 用来存储LFU节点的键值散列表
    this.capacity = capacity; // 用来定义缓存的承载量
    this.size = 0; // 把缓存大小初始设置为0
  }
  set() {/* 插入一个节点 */}
  get() {/* 读取一个节点 */}
}
```

总结

如果从直觉上来看，你可能会觉得对比 LRU，似乎 LFU 是更合理的一种缓存方法，它根据访问的频率来决定对内容进行缓存或清除。回到上面的问题，为什么在实际应用中，更多被使用

的是 LRU 呢？这是因为一是短时间高频访问本身不能证明缓存有长期价值，二是它可能把一些除了短时间高频访问外的内容挤掉，第三就是刚被访问的内容也很可能不会被频繁访问，但是仍然可能被重复访问，这里也增加了它们被意外删除的可能性。

所以总的来说，LFU 没有考虑时间局部性，也就是“最近被访问的可能再次被访问”这个问题。所以在实际的应用中，使用 LRU 的概率要大于 LFU。希望通过这节课，你能对链表和散列表的使用有更多的了解，更主要的是，当你的应用需要写自己的缓存的时候，也可以把这些因素考虑进去。

思考题

下面到了思考题时间，虽然说 LFU 的应用场景比较少，但是它在某些特定场合还是很有用处的，你能想到一些相关的场景和实现吗？

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。

上一篇 16 | 为什么环形队列适合做Node数据流缓存？

下一篇 18 | TurboFan如何用图做JS编译优化？

精选留言 (1)

写留言

天择

2022-10-27 来自北京

LRU具备时间“局部”性，而LFU恰恰一种“全局”性。在需要考虑全局频率的场景里面，LFU是有用的。最近有大量访问，不代表始终有大量访问；比如电商的热门商品，可能在几个月的维度上做了统计，如果采用LRU策略，短时促销活动可能会把这些长期热门商品冲出缓存，导致过了促销，原来的热门商品缓存无法命中。

作者回复: 有道理

1