前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

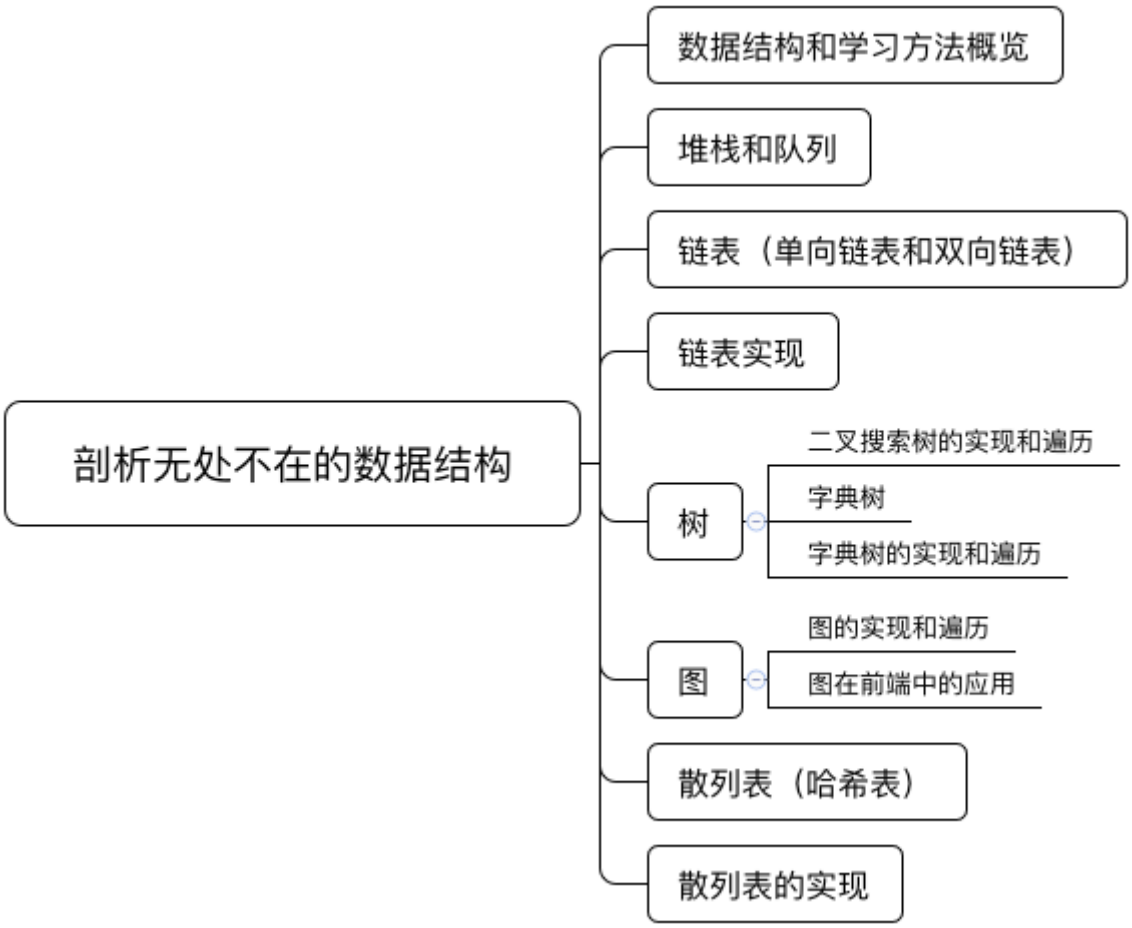
查看详情 >

剖析无处不在的数据结构

数据结构是计算机中组织和存储数据的特定方式，它的目的是方便且高效地对数据进行访问和修改。数据结构表述了数据之间的关系，以及操作数据的一系列方法。数据又是程序的基本单元，因此无论是哪种语言、哪种领域，都离不开数据结构；另一方面，数据结构是算法的基础，其本身也包含了算法的部分内容。也就是说，想要掌握算法，先有一个巩固的数据结构基础是必要条件。

前端领域也到处体现着数据结构的应用，尤其是随着需求的复杂度上升，前端工程师越来越离不开数据结构。React、Vue 这些设计精巧的框架，在线文档编辑系统、大型管理系统，甚至一个简单的检索需求，都离不开数据结构的支持。是否能够掌握这个难点内容，将是进阶的重要考量。我们应该如何学习数据结构呢?

下图是本讲内容的提纲。



数据结构和学习方法概览

我通常将数据结构分为八大类：

数组：Array

堆栈：Stack

队列：Queue

链表：Linked Lists

树：Trees

图：Graphs

字典树：Trie

散列表（哈希表）：Hash Tables

这么多的类型，这节课该如何介绍呢？我认为，按部就班地只是实现各种数据结构的意义不大，这些内容读者都可以从算法书籍中找到。更重要地是应用，也只有在应用中，才能真正地记住并掌握特定的数据结构，才能在下次有类似场景时，能够想起来相关的数据结构实现。因此，这节课程我将从前端出发，从前端类库或者典型场景入手，结合数据结构来剖析其实现和应用。这需要读者首先对每种数据结构有一个大概认知，我们可以先来细化感知一下：

栈和队列是类似数组的结构，非常多的初级题目要求用数组实现栈和队列，他们在插入和删除的方式上和数组有所差异，但是实现还是非常简单的；

链表、树和图这种数据结构的特点是，其节点需要引用到其他节点，因此在增删时，需要注意对相关前驱和后继节点的影响；

可以从堆栈和队列出发，构建出链表；

树和图最为复杂，因为他们本质上是扩展了链表的概念；

散列表的关键是理解散列函数，明白依赖散列函数实现保存和定位数据的过程；

直观上认为，链表适合记录和存储数据；哈希表和字典树在检索数据以及搜索方面有更大的应用场景。

以上这些「直观感性」的认知并不是「恒等式」，**我们将在下面的学习中去印证这些「认知」，你将会看到熟悉的 React、Vue 框架的部分实现，将会看到典型的算法场景，也请读者做好基础知识的储备。**

堆栈和队列

栈和队列是一种操作受限的线性结构，它们非常简单，虽然 JavaScript 并没有原生内置这样的数据结构，但是我们可以轻松地模拟出来。

栈的实现，后进先出 LIFO（Last in、First out）：

```
class Stack {
 constructor(...args) {
   this.stack = [...args]
 }

 // Modifiers
 push(...items) {
   return this.stack.push(... items)
 }

 pop() {
   return this.stack.pop()
 }

 // Element access
 peek() {
   return this.isEmpty()
       ? undefined
       : this.stack[this.size() - 1]
 }

 // Capacity
 isEmpty() {
   return this.size() == 0
 }

 size() {
   return this.stack.length
 }
}
```

队列的实现，先进先出 FIFO（First in、First out）：

```
class Queue {
 constructor(...args) {
   this.queue = [...args]
 }
```

```
  // Modifiers
  enqueue(...items) {
    return this.queue.push(... items)
  }

  dequeue() {
    return this.queue.shift()
  }

  // Element access
  front() {
    return this.isEmpty()
        ? undefined
        : this.queue[0]
  }

  back() {
    return this.isEmpty()
        ? undefined
        : this.queue[this.size() - 1]
  }

  // Capacity
  isEmpty() {
    return this.size() == 0
  }

  size() {
    return this.queue.length
  }
}
```

关于栈和队列的实际应用比比皆是：

浏览器的历史记录，因为回退总是回退「上一个」最近的页面，它需要遵循栈的原则；

类似浏览器的历史记录，任何 undo/redo 都是一个栈的实现；

在代码中，广泛应用的递归产生的调用栈，同样也是栈思想的体现，想想我们常说的「栈溢出」就是这个道理；

同上，浏览器在抛出异常时，常规都会抛出调用栈信息；

在计算机科学领域应用广泛，如进制转换、括号匹配、栈混洗、表达式求值等；

队列的应用更为直观，我们常说的宏任务 / 微任务都是队列，不管是什么类型的任务，都是先进先执行；

后端也应用广泛，如消息队列、RabbitMQ、ActiveMQ 等，能起到延迟缓冲的功效。

我们看到不管是栈还是队列，都是用数组来模拟的。数组是最基本的数据结构，但是它的价值是惊人的，这里稍微提一下 React hooks 的本质就是数组。给大家推荐文章：React hooks: not magic, just arrays

另外，与性能后话相关，HTTP 1.1 有一个队头阻塞问题，这个原因就在于队列这样的数据结构：我们先看 HTTP 1.0，对于同一个 tcp 连接，HTTP 1.0 是将所有请求都放入队列当中，这么一来，在客户端，「先进先出」，只有前一个请求得到了响应，下一个请求才会发出。在 HTTP 1.1 中，这样的情况得到了改观，每一个链接都默认是长链接，因此对于同一个 tcp 链接，不必等到前一个响应回来。但是这只是解决了客户端的队头阻塞问题，事实上，HTTP 1.1 规定：服务端的响应返回顺序需要遵循其接收到相应的顺序，这样的问题是：如果第一个请求处理需要较长时间，响应较慢，也都会「拖累」其他后续请求的响应，这仍然是一种队头阻塞。

HTTP 2 采用了二进制分帧和多路复用等方法， 同域名下的通信都是在同一个连接上完成，并且这种链接是双向的，在这个链接上可以并行请求和响应而互不干扰。

这里延伸的有点多了，主要是读者需要明白队列和栈这种数据结构的应用，以及利弊。

链表（单向链表和双向链表）

堆栈和队列都可以用数组实现，链表同样和数组一样，都实现了**按照一定的顺序**存储元素，不同的地方在于链表不能像数组一样通过下标访问，而是每一个元素指向下一个元素。我们不再过多介绍链表方面的基础知识，对于链表仍不理解的读者可以先自行学习。

直观上我们就可以得出结论：链表不需要一段连续的存储空间，「指向下一个元素」的方式能够更大限度地利用内存。

根据上面结论可以继续总结，链表的优点在于：

> 链表的插入和删除操作的时间复杂度是常数级的，我们只需要改变相关节点的指针指向即可；
>
> 链表可以像数组一样顺序访问，查找元素的时间复杂度是线性的。

我们来看看链表的应用场景：

> React 的核心算法 Fiber 的实现就是链表

关于此我们可以稍作展开。React 最早开始使用大名鼎鼎的 Stack reconciler 调度算法，关于此在之前的课程中已经有所涉及。Stack reconciler 调度算法最大的问题在于：**它是像函数调用栈一样，递归地、自顶向下进行 diff 和 render 相关操作的**，在 Stack reconciler 执行的过程当中，该调度算法始终会占据浏览器主线程。也就是说在此期间，用户的交互所触发的布局行为、动画执行任务都不会被立即响应，从而影响用户体验。

因此 React Fiber 将渲染和更新过程进行了拆解，简单来说，就是每次检查虚拟 DOM 的一小部分，在检查间隙会检查「是否还有时间继续执行下一个虚拟 DOM 树上某个分支任务」，同时观察是否有更优先的任务需要响应，如果「没有时间执行下一个虚拟 DOM 树上某个分支任务」，且有更高优先级，React 就会让出主线程，直到主线程「不忙」的时候继续执行任务。

React Fiber 因此也很简单，它是将 Stack reconciler 过程分成块，一次执行一块，执行完一块需要将结果保存起来，根据是否还有空闲的响应时间

（requestIdleCallback）来决定下一步策略。当所有的块都已经执行完，就进入提交阶段，这个阶段需要更新 DOM，它是一口气完成的。

以上是比较主观地介绍，我们来看更具体的实现。

为了达到「随意中断调用栈并手动操作调用栈」，React Fiber 就是专门用于 React 组件堆栈调用的重新实现，也就是说一个 Fiber 就是一个虚拟堆栈帧，一个 Fiber 的结构类似：

```
function FiberNode(
 tag: WorkTag,
 pendingProps: mixed,
 key: null | string,
 mode: TypeOfMode,
) {
 // Instance
 // ...
 this.tag = tag;

 // Fiber
 this.return = null;
 this.child = null;
 this.sibling = null;
 this.index = 0;

 this.ref = null;

 this.pendingProps = pendingProps;
 this.memoizedProps = null;
 this.updateQueue = null;
 this.memoizedState = null;
 this.dependencies = null;

 // Effects
 // ...
 this.alternate = null;
}
```

这么看 Fiber 就是一个对象，通过 parent、children、sibling 维护一个树形关系，同时 parent、children、sibling 也都是一个 Fiber 结构，FiberNode.alternate 这个属性来存储上一次渲染过的结果，事实上整个 Fiber 模式就是一个链表。React 也借此，从依赖于内置堆栈的同步递归模型，变为具有链表和指针的异步模型了。

具体的渲染过程：

```
function renderNode(node) {
  // 判断是否需要渲染该节点，如果 props 发生变化，则调用 render
  if (node.memoizedProps !== node.pendingProps) {
     render(node)
  }

  // 是否有子节点，进行子节点渲染
  if (node.child !== null) {
     return node.child
  // 是否有兄弟节点，进行兄弟点渲染
  } else if (node.sibling !== null){
     return node.sibling
  // 没有子节点和兄弟节点
  } else if (node.return !== null){
     return node.return
  } else {
     return null
  }
}

function workloop(root) {
  nextNode = root
  while (nextNode !== null && (no other high priority
task)) {
     nextNode = renderNode(nextNode)
  }
}
```

注意在 workloop 当中，while 条件 nextNode !== null && (no other high priority task)，这是描述 Fiber 工作原理的关键伪代码。

当然这里是为了说明链表的数据结构，伪代码较为简略，也没有深入：

> requestAnimationFrame(callback)
>
> requestIdleCallback(callback)

的实现和应用。React Fiber 的介绍我们先到此为止，重点是体会链表数据结构的思想。

链表实现

实现链表，我们需要先对链表进行分类，常见的有：

> 单链表：单链表是维护一系列节点的数据结构，其特点是：每个节点包含了数据，同时包含指向链表中下一个节点的指针。
>
> 双向链表：不同于单链表，双向链表特点：每个节点分除了包含其数据以外，还包含了分别指向其前驱和后继节点的指针。

由于篇幅有原因，我们挑选更加复杂的双向链表进行实现，实现思路如下。

首先，根据双向链表的特点，我们实现一个节点构造函数（节点类）：

```
class Node {
   constructor(data) {
       // data 为当前节点所储存的数据
       this.data = data
       // next 指向下一个节点
       this.next = null
       // prev 指向前一个节点
       this.prev = null
   }
}
```

有了节点类，我们来初步实现双向链表类：

```
class DoublyLinkedList {
   constructor() {
       // 双向链表开头
       this.head = null
       // 双向链表结尾
     this.tail = null
   }

   // ...
}
```

接下来，需要实现双向链表原型上的一些方法，这些方法包括

add：在链表尾部添加一个新的节点

addAt：在链表指定位置添加一个新的节点

remove：删除链表指定数据项节点

removeAt：删除链表指定位置节点

reverse：翻转链表

swap：交换两个节点数据

isEmpty：查询链表是否为空

length：查询链表长度

traverse：遍历链表

find：查找某个节点的索引

add 方法：

```
add(item) {
 // 实例化一个节点
 let node = new Node(item)

 // 如果当前链表还没有头
 if(!this.head) {
   this.head = node
   this.tail = node
 }
 // 如果当前链表已经有了头，只需要在尾部加上该节点
 else {
   node.prev = this.tail
   this.tail.next = node
   this.tail = node
 }
}
```

addAt 方法：

```
addAt(index, item) {
  let current = this.head
  // 维护查找时当前节点的索引
  let counter = 1
  let node = new Node(item)

  // 如果在头部插入
  if (index === 0) {
    this.head.prev = node
    node.next = this.head
    this.head = node
  }
  // 非头部插入，需要从头开始，找寻插入位置
  else {
    while(current) {
     current = current.next
     if( counter === index) {
       node.prev = current.prev
```

```
        current.prev.next = node
        node.next = current
        current.prev = node
      }
      counter++
    }
  }
}
```

remove 方法：

```
remove(item) {
 let current = this.head

 while (current) {
      // 找到了目标节点
   if (current.data === item ) {
     // 目标链表只有当前目标项，即目标节点即是链表头又是链表尾
     if (current == this.head && current == this.tail) {
       this.head = null
       this.tail = null
     }
     // 目标节点为链表头
     else if (current == this.head ) {
       this.head = this.head.next
       this.head.prev = null
     }
     // 目标节点为链表尾部
     else if (current == this.tail ) {
       this.tail = this.tail.prev;
       this.tail.next = null;
     }
     // 目标节点在链表收尾之间，中部
     else {
       current.prev.next = current.next;
       current.next.prev = current.prev;
     }
```

```
  }
  current = current.next
 }
}
```

removeAt 方法：

```
removeAt(index) {
 // 都是从「头」开始遍历
 let current = this.head
 let counter = 1

 // 删除链表头部
 if (index === 0 ) {
  this.head = this.head.next
  this.head.prev = null
 }
 else {
  while(current) {
   current = current.next
   // 如果目标节点在链表尾
   if (current == this.tail) {
    this.tail = this.tail.prev
    this.tail.next = null
   }
   else if (counter === index) {
    current.prev.next = current.next
    current.next.prev = current.prev
    break
   }
   counter++
  }
 }
}
```

reverse 方法：

```
reverse() {
 let current = this.head
 let prev = null

 while (current) {
  let next = current.next

  // 前后倒置
  current.next = prev
  current.prev = next

  prev = current
  current = next
 }

 this.tail = this.head
 this.head = prev
}
```

swap 方法，交换两个节点数据值：

```
swap(index1, index2) {
 // 使 index1 始终小于 index2，方便后面查找交换
 if (index1 > index2) {
   return this.swap(index2, index1)
 }

 let current = this.head
 let counter = 0
 let firstNode

 while(current !== null) {
   // 找到第一个节点，先存起来
   if (counter === index1 ){
       firstNode = current
   }
```

```
    // 找到第二个节点，进行数据交换
    else if (counter === index2) {
      // ES 提供了更简洁交换数据的方法，这里我们用传统方式实现，更为直观
      let temp = current.data
      current.data = firstNode.data
      firstNode.data = temp
    }

    current = current.next
    counter++
  }
  return true
}
```

isEmpty 方法：

```
isEmpty() {
  return this.length() < 1
}
```

这里通过 DoublyLinkedList 类 length 的方法实现。马上看一下 length 方法：

```
length() {
  let current = this.head
  let counter = 0
  while(current !== null) {
    counter++
    current = current.next
  }
  return counter
}
```

length 方法通过遍历链表，返回链表的长度。

traverse 方法：

```
traverse(fn) {
 let current = this.head
 while(current !== null) {
   fn(current)
   current = current.next
 }
 return true
}
```

有了上面 length 方法的遍历实现，traverse 也就不难理解了，它接受一个遍历执行函数，在 while 循环中进行调用。

最后一个 search 方法：

```
search(item) {
 let current = this.head
 let counter = 0

 while( current ) {
   if( current.data == item ) {
     return counter
   }
   current = current.next
   counter++
 }
 return false
}
```

到此，我们就实现了所有 DoublyLinkedList 类双向链表的方法。仔细分析整个实现过程，可以发现：双向链表的实现并不复杂，在手写过程当中，需要开发者做到心中有表，考虑到当前节点的 next 和 prev 取值，逻辑上还是很简单的。

掌握了这些内容，在回想一下链表的应用，回想 React Fiber 的设计和实现，也许一切都变的不再神秘。

树

前端开发者应该对树这个数据结构丝毫不陌生，不同于之前介绍的所有数据结构，树是非线性的。因为树决定了其存储的数据直接有明确的层级关系，因此对于维护具有层级特性的数据，树是一个天然良好的选择。

在前面总领中，我们看到树有很多种分类，但是他们都具有以下特性：

除了根节点以外，所有的节点都有一个父节点

每一个节点都**可以有**若干子节点，如果没有子节点，那么称此节点为叶子节点

一个节点所拥有的叶子节点的个数，称之为该节点的度，因此叶子节点的度为 0

所有节点中，最大的度为整棵树的度

树的最大层次称为树的深度

从应用上来看，我们前端开发离不开的 DOM 就是一个树状结构；同理，不管是 React 还是 Vue 的虚拟 DOM 也都是树。

我们从最基本的二叉树入手，来慢慢深入。

二叉搜索树的实现和遍历

说二叉树最为基本，因为他的结构最简单，每个节点至多包含两个子节点。二叉树又非常有用：因为根据二叉树，我们可以延伸出二叉搜索树（BST）、平衡二叉搜索树（AVL）、红黑树（R/B Tree）等。

二叉搜索树有以下特性：

左子树上所有结点的值均小于或等于它的根结点的值

右子树上所有结点的值均大于或等于它的根结点的值

左、右子树也分别为二叉搜索树

根据其特性，我们实现二叉搜索树还是应该先构造一个节点类：

```
class Node {
 constructor(data) {
   this.left = null
   this.right = null
   this.value = data
 }
}
```

接着按照惯例，我们实现二叉搜索树的以下方法：

- insertNode：根据一个父节点，插入一个子节点
- insert：插入一个新节点
- removeNode：根据一个父节点，移除一个子节点
- remove：移除一个节点
- findMinNode：获取子节点的最小值
- searchNode：根据一个父节点，查找子节点
- search：查找节点
- preOrder：前序遍历
- InOrder：中序遍历
- PostOrder：后续遍历

```
insertNode(root, newNode) {
 if (newNode.value < root.value) {
   (!root.left) ? root.left = newNode :
this.insertNode(root.left, newNode)
```

```
 } else {
   (!root.right) ? root.right = newNode :
this.insertNode(root.right, newNode)
 }
}

insert(value) {
   let newNode = new Node(value)
   if (!this.root) {
     this.root = newNode
   } else {
     this.insertNode(this.root, newNode)
   }
}
```

理解这两个方法是理解二叉搜索树的关键，下面的其他方法也就「不在话下」。我们看，insertNode 方法先判断目标父节点和插入节点的值，如果插入节点的值更小，则考虑放到父节点的左边，接着递归调用 his.insertNode(root.left, newNode)；如果插入节点的值更大，以此类推即可。

insert 方法只是多了一步构造 Node 节点实例，接下来区分有无父节点的情况，调用 this.insertNode 方法即可。

```
 removeNode(root, value) {
   if (!root) {
     return null
   }

   if (value < root.value) {
     root.left = this.removeNode(root.left, value)
     return root
   } else if (value > root.value) {
     root.right = tis.removeNode(root.right, value)
     return root
   } else {
     // 找到了需要删除的节点
     // 如果当前 root 节点无左右子节点
```

```
      if (!root.left && !root.right) {
        root = null
        return root
      }

      // 只有左节点
      if (root.left && !root.right) {
        root = root.left
        return root
      }
      // 只有右节点
      else if (root.right) {
        root = root.right
        return root
      }

      // 有左右两个子节点
      let minRight = this.findMinNode(root.right)
      root.value = minRight.value
      root.right = this.removeNode(root.right,
minRight.value)
      return root
    }
  }

remove(value) {
    if (this.root) {
      this.removeNode(this.root, value)
    }
}
```

上述代码不难理解，可能最需要读者思考的就是：

```
// 有左右两个子节点
let minRight = this.findMinNode(root.right)
root.value = minRight.value
```

```
root.right = this.removeNode(root.right, minRight.value)
return root
```

我来特殊说明一下：当需要删除的节点含有左右两个子节点时，因为我们要把当前节点删除，就需要找到合适的「补位」节点，这个「补位」节点一定在该目标节点的右侧树当中，因为这样才能保证「补位」节点的值一定大于该目标节点的左侧树所有节点，而该目标节点的左侧树不需要调整；同时为了保证「补位」节点的值一定要小于该目标节点的右侧树值，因此要找的「补位」节点其实就是该目标节点的右侧树当中最小的那个节点。

这个过程我们借助 this.findMinNode 方法实现：

```
findMinNode(root) {
   if (!root.left) {
     return root
   } else {
     return this.findMinNode(root.left)
   }
}
```

该方法不断递归，直到找到最左叶子节点即可。

查找方法：

```
searchNode(root, value) {
   if (!root) {
     return null
   }

   if (value < root.value) {
     return this.searchNode(root.left, value)
   } else if (value > root.value) {
     return this.searchNode(root.right, value)
   }

   return root
```

```
}

search(value) {
   if (!this.root) {
     return false
   }
   return Boolean(this.searchNode(this.root, value))
}
```

这也比较简单，其实就是对递归的运用。最能体现递归简便优势的其实是对于树的遍历：

前序遍历：

```
preOrder(root) {
   if (root) {
     console.log(root.value)
     this.preOrder(root.left)
     this.preOrder(root.right)
   }
}
```

中序遍历：

```
inOrder(root) {
   if (root) {
     this.inOrder(root.left)
     console.log(root.value)
     this.inOrder(root.right)
   }
}
```

后序遍历：

```
postOrder(root) {
   if (root) {
```

```
    this.postOrder(root.left)
    this.postOrder(root.right)
    console.log(root.value)
  }
}
```

前后中序遍历其实就在于 console.log(root.value) 方法执行的位置。

字典树

字典树（Trie）是针对特定类型的搜索而优化的树数据结构。典型的例子是 autoComplete，也就是说它适合实现：通过部分值得到完整值的场景。字典树因此也是一种搜索树，我们有时候也叫做前缀树，因为任意一个节点的后代都存在共同的前缀。更多基础概念请读者先做了解。我们总结一下它的特点：

字典树能做到高效查询和插入，时间复杂度为 O(k)，k 为字符串长度

但是如果大量字符串没有共同前缀，那就很耗内存，读者可以想象一下最极端的情况，所有单词都没有共同前缀时，这颗字典树是什么样子

字典树的核心就是减少没必要的字符比较，使查询高效率，也就是说用空间换时间，再利用共同前缀来提高查询效率

除了我们刚刚提到的 autoComplete 自动填充的情况，字典树还有很多其他应用场景：

搜索

输入法选项

分类

IP 地址检索

电话号码检索

字典树的实现和遍历

字典树的实现也不复杂，我们慢慢一步步来，首先实现一个字典树上的节点：

```
class PrefixTreeNode {
 constructor(value) {
   // 存储子节点
   this.children = {}
   this.isEnd = null
   this.value = value
 }
}
```

一个字典树继承 PrefixTreeNode 类：

```
class PrefixTree extends PrefixTreeNode {
 constructor() {
   super(null)
 }
}
```

我们实现方法：

addWord：创建一个字典树节点

predictWord：给定一个字符串，返回字典树中以该字符串开头的所有单词

addWord 实现：

```
addWord(str) {
   const addWordHelper = (node, str) => {
         // 当前 node 不含当前 str 开头的目标
       if (!node.children[str[0]]) {
           // 以当前 str 开头的第一个字母，创建一个
PrefixTreeNode 实例
           node.children[str[0]] = new
```

```
PrefixTreeNode(str[0])
            if (str.length === 1) {
                node.children[str[0]].isEnd = true
            }
            else if (str.length > 1) {
                addWordHelper(node.children[str[0]],
str.slice(1))
            }
        }
    }
    addWordHelper(this, str)
}
```

predictWord 实现：

```
predictWord(str) {
    let getRemainingTree = function(str, tree) {
      let node = tree
      while (str) {
        node = node.children[str[0]]
        str = str.substr(1)
      }
      return node
    }

    // 该数组维护所有以 str 开头的单词
    let allWords = []

    let allWordsHelper = function(stringSoFar, tree) {
      for (let k in tree.children) {
        const child = tree.children[k]
        let newString = stringSoFar + child.value
        if (child.endWord) {
          allWords.push(newString)
        }
        allWordsHelper(newString, child)
      }
```

```
    }

    let remainingTree = getRemainingTree(str, this)

    if (remainingTree) {
      allWordsHelper(str, remainingTree)
    }

    return allWords
}
```

图

图是由具有边的节点集合组成的数据结构，图可以是定向的或不定向的。因此图可以分为好多种类，这里不一一讲解，主要看图的应用场景：

LBS 地图服务以及 GPS 系统

社交媒体网站的用户关系图

前端工程化中的开发依赖图

搜索算法使用图，保证搜索结果的相关性

寻找降低运输和交付货物和服务成本的最佳途径

图也是应用最广泛的数据结构之一，真实场景中处处有图。更多概念还是需要读者先进行了解，尤其是图的几种基本元素：

节点 Node

边 Edge

|V| 图中顶点（节点）的总数

|E| 图中的连接总数（边）

图的实现和遍历

这里我们主要实现一个有向图，Graph 类：

```
class Graph {
 constructor() {
   this.AdjList = new Map()
 }
}
```

使用 Map 数据结构表述图中顶点关系。

实现方法：

- 添加顶点：addVertex
- 添加边：addEdge
- 打印图：print
- 广度优先算法遍历
- 深度优先算法

addVertex 方法：

```
addVertex(vertex) {
 if (!this.AdjList.has(vertex)) {
   this.AdjList.set(vertex, [])
 } else {
   throw 'vertex already exist!'
 }
}
```

创建顶点：

```
let graph = new Graph();
graph.addVertex('A')
graph.addVertex('B')
graph.addVertex('C')
graph.addVertex('D')
```

其中 A、B、C、D 顶点都对应一个数组：

```
  'A' => [],
 'B' => [],
 'C' => [],
 'D' => []
```

该数组将用来存储边。我们设计图预计得到如下关系：

```
Map {
 'A' => ['B', 'C', 'D'],
 'B' => [],
 'C' => ['B'],
 'D' => ['C']
}
```

根据此描述，其实已经可以把图画出来了。addEdge 因此需要两个参数：一个是顶点，一个是连接对象 Node：

```
 addEdge(vertex, node) {
   if (this.AdjList.has(vertex)) {
     if (this.AdjList.has(node)){
       let arr = this.AdjList.get(vertex)
       if(!arr.includes(node)){
         arr.push(node)
       }
     }else {
       throw `Can't add non-existing vertex ->'${node}'`
     }
   } else {
```

```
      throw `You should add '${vertex}' first`
    }
}
```

理清楚数据关系，我们就可以打印图了，其实就是一个很简单的 for...of 循环：

```
print() {
 for (let [key, value] of this.AdjList) {
   console.log(key, value)
 }
}
```

剩下的内容就是遍历图了。

广度优先算法（BFS），是一种利用队列实现的搜索算法。对于图，其搜索过程和 「湖面丢进一块石头激起层层涟漪」 类似。换成算法语言，就是从起点出发，对于每次出队列的点，都要遍历其四周的点。

因此 BFS 的实现步骤：

- 起始节点作为起始，并初始化一个空对象：visited
- 初始化一个空数组，该数组将模拟一个队列
- 将起始节点标记为已访问
- 将起始节点放入队列中
- 循环直到队列为空

实现：

```
createVisitedObject() {
 let map = {}
 for(let key of this.AdjList.keys()) {
   arr[key] = false
```

```
 }
 return map
}

bfs(initialNode) {
 // 创建一个已访问节点的 map
 let visited = this.createVisitedObject()
 // 模拟一个队列
 let queue = []

 // 第一个节点已访问
 visited[initialNode] = true
 // 第一个节点入队列
 queue.push(initialNode)

 while(queue.length) {
   let current = queue.shift()
   console.log(current)

    // 获得该节点的其他节点关系
   let arr = this.AdjList.get(current)

   for (let elem of arr) {
     // 如果当前节点没有访问过
     if (!visited[elem]) {
       visited[elem] = true
       q.push(elem)
     }
   }
 }
}
```

那么对于深度优先搜索算法（DFS），我把它总结为：「不撞南墙不回头」，从起点出发，先把一个方向的点都遍历完才会改变方向。换成程序语言就是：「DFS 是利用递归实现的搜索算法」。

因此 DFS 过程：

起始节点作为起始，创建访问对象

调用辅助函数递归起始节点

实现代码：

```
createVisitedObject() {
 let map = {}
 for (let key of this.AdjList.keys()) {
   arr[key] = false
 }
 return map
}

dfs(initialNode) {
   let visited = this.createVisitedObject()
   this.dfsHelper(initialNode, visited)
 }

 dfsHelper(node, visited) {
   visited[node] = true
   console.log(node)

   let arr = this.AdjList.get(node)

   for (let elem of arr) {
     if (!visited[elem]) {
       this.dfsHelper(elem, visited)
     }
   }
 }
}
```

BFS 的重点在于队列，而 DFS 的重点在于递归，这是它们的本质区别。

图在前端中的应用

图其实在前端中应用不算特别多，但绝对还是不容忽视的一部分。这里我举一个我现实中应用的例子——循环图。

在前端工程化发展的今天，理清项目中的依赖关系：比如查找项目中的循环依赖，可视化依赖都是图的应用，有助于开发者在宏观上把控工程化项目。在我们的项目中，我借助 mermaidj 画图工具，实现了项目依赖的完全可视化。并借助 npm script 来生成图片结果，相关 script 脚本：

```
yarn graph
```

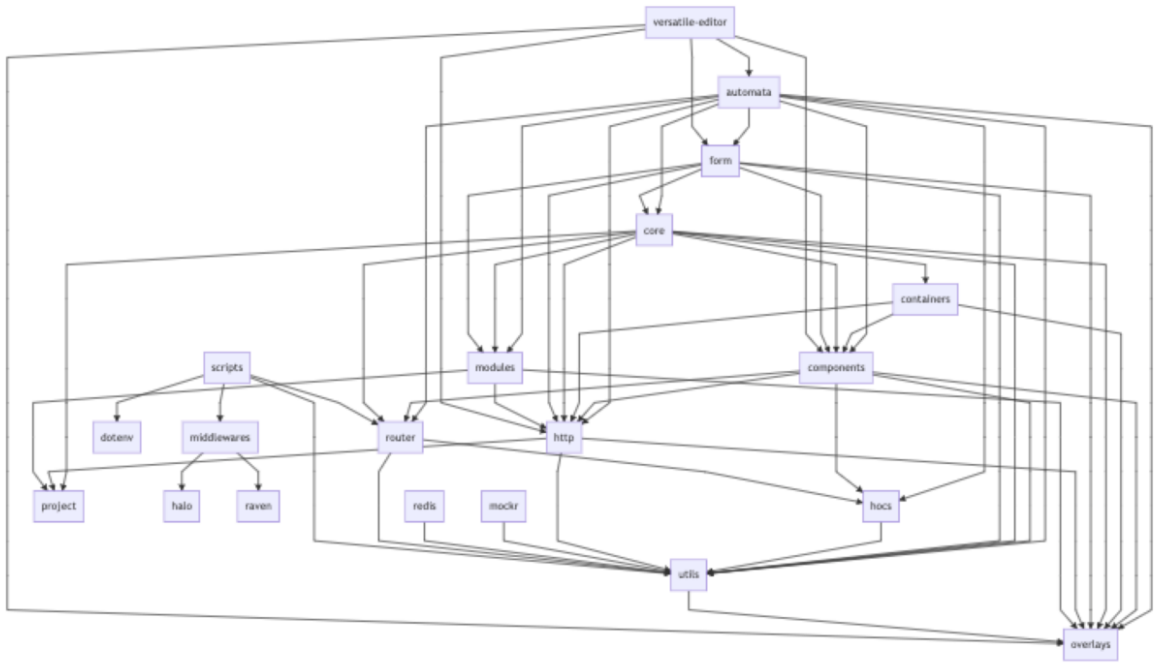
脚本：

```
import glob from 'glob'
import readJSON from 'XXX/utils/readJSON'

const pkgs =
glob.sync('packages/*/package.json').map(readJSON)

const deps = {}

for (const pkg of pkgs) {
 deps[pkg.name] = Object.keys(pkg.dependencies ||
[]).filter(dep =>
   // ...
 )
}

const graph = { code: '', mermaid: { theme: 'default' } }

graph.code += 'graph TD;'
for (const name in deps) {
 for (const dep of deps[name]) {
   graph.code += `${name}-->${dep};`
 }
}
```

```
const base64 =
Buffer.from(JSON.stringify(graph)).toString('base64')

/* eslint-disable-next-line */
console.log(
 `Open in browser: https://mermaidjs.github.io/mermaid-
live-editor/#/edit/${base64}`
)
```

上述代码，我首先获取到 packages/*/package.json 中声明的所有依赖，然后对依赖进行必要性过滤之后，维护到 deps 对象当中，按照 mermaid 需求，将 monorepo 项目中的每一个子项目名和依赖按照 → 的间隔维护为 graph.code，最后通过生成 base64 交给 mermaid 进行绘图，绘图过程会根据约定（→ 的标记），成生可视化的依赖图。

最终效果：



那么 mermaid 是如何对图进行绘制的呢？了解了课程前面实现图的代码，我们再看 mermaid 绘制图的部分源码实现：

```
export const addVertices = function (vert, g, svgId) {
 const svg = d3.select(`[id="${svgId}"]`)
```

```
 const keys = Object.keys(vert)

 const styleFromStyleArr = function (styleStr, arr) {
   // Create a compound style definition from the style
definitions found for the node in the graph definition
   for (let i = 0; i < arr.length; i++) {
     if (typeof arr[i] !== 'undefined') {
       styleStr = styleStr + arr[i] + ';'
     }
   }

   return styleStr
 }

 // Iterate through each item in the vertex object
(containing all the vertices found) in the graph
definition
 keys.forEach(function (id) {
   const vertex = vert[id]

   /**
    * Variable for storing the classes for the vertex
    * @type {string}
    */
   let classStr = ''
   if (vertex.classes.length > 0) {
     classStr = vertex.classes.join(' ')
   }

   /**
    * Variable for storing the extracted style for the
vertex
    * @type {string}
    */
   let style = ''
   // Create a compound style definition from the style
definitions found for the node in the graph definition
```

```
    style = styleFromStyleArr(style, vertex.styles)

    // Use vertex id as text in the box if no text is
provided by the graph definition
    let vertexText = vertex.text !== undefined ?
vertex.text : vertex.id

    // We create a SVG label, either by delegating to
addHtmlLabel or manually
    let vertexNode
    if (conf.htmlLabels) {
      // TODO: addHtmlLabel accepts a labelStyle. Do we
possibly have that?
      const node = { label:
vertexText.replace(/fa[lrsb]?:fa-[\w-]+/g, s => ``) }
      vertexNode = addHtmlLabel(svg, node).node()
      vertexNode.parentNode.removeChild(vertexNode)
    } else {
      const svgLabel =
document.createElementNS('http://www.w3.org/2000/svg',
'text')

      const rows = vertexText.split(//)

      for (let j = 0; j < rows.length; j++) {
        const tspan =
document.createElementNS('http://www.w3.org/2000/svg',
'tspan')
        tspan.setAttributeNS('http://www.w3.org/XML/1998/na
mespace', 'xml:space', 'preserve')
        tspan.setAttribute('dy', '1em')
        tspan.setAttribute('x', '1')
        tspan.textContent = rows[j]
        svgLabel.appendChild(tspan)
      }
      vertexNode = svgLabel
    }
```

```
    // If the node has a link, we wrap it in a SVG link
    if (vertex.link) {
      const link =
document.createElementNS('http://www.w3.org/2000/svg',
'a')
      link.setAttributeNS('http://www.w3.org/2000/svg',
'href', vertex.link)
      link.setAttributeNS('http://www.w3.org/2000/svg',
'rel', 'noopener')
      link.appendChild(vertexNode)
      vertexNode = link
    }

    let radious = 0
    let _shape = ''
    // Set the shape based parameters
    switch (vertex.type) {
      case 'round':
        radious = 5
        _shape = 'rect'
        break
      case 'square':
        _shape = 'rect'
        break
      case 'diamond':
        _shape = 'question'
        break
      case 'odd':
        _shape = 'rect_left_inv_arrow'
        break
      case 'lean_right':
        _shape = 'lean_right'
        break
      case 'lean_left':
        _shape = 'lean_left'
        break
```

```
      case 'trapezoid':
        _shape = 'trapezoid'
        break
      case 'inv_trapezoid':
        _shape = 'inv_trapezoid'
        break
      case 'odd_right':
        _shape = 'rect_left_inv_arrow'
        break
      case 'circle':
        _shape = 'circle'
        break
      case 'ellipse':
        _shape = 'ellipse'
        break
      case 'group':
        _shape = 'rect'
        break
      default:
        _shape = 'rect'
    }
    // Add the node
    g.setNode(vertex.id, { labelType: 'svg', shape: _shape,
label: vertexNode, rx: radious, ry: radious, 'class':
classStr, style: style, id: vertex.id })
  })
}

/**
* Add edges to graph based on parsed graph defninition
* @param {Object} edges The edges to add to the graph
* @param {Object} g The graph object
*/
export const addEdges = function (edges, g) {
  let cnt = 0

  let defaultStyle
```

```
 if (typeof edges.defaultStyle !== 'undefined') {
   defaultStyle =
edges.defaultStyle.toString().replace(/,/g, ';')
 }

 edges.forEach(function (edge) {
   cnt++
   const edgeData = {}

   // Set link type for rendering
   if (edge.type === 'arrow_open') {
     edgeData.arrowhead = 'none'
   } else {
     edgeData.arrowhead = 'normal'
   }

   let style = ''
   if (typeof edge.style !== 'undefined') {
     edge.style.forEach(function (s) {
       style = style + s + ';'
     })
   } else {
     switch (edge.stroke) {
       case 'normal':
         style = 'fill:none'
         if (typeof defaultStyle !== 'undefined') {
           style = defaultStyle
         }
         break
       case 'dotted':
         style = 'stroke: #333; fill:none;stroke-
width:2px;stroke-dasharray:3;'
         break
       case 'thick':
         style = 'stroke: #333; stroke-width:
3.5px;fill:none'
         break
```

```
      }
    }
    edgeData.style = style

    if (typeof edge.interpolate !== 'undefined') {
      edgeData.curve = interpolateToCurve(edge.interpolate,
d3.curveLinear)
    } else if (typeof edges.defaultInterpolate !==
'undefined') {
      edgeData.curve =
interpolateToCurve(edges.defaultInterpolate,
d3.curveLinear)
    } else {
      edgeData.curve = interpolateToCurve(conf.curve,
d3.curveLinear)
    }

    if (typeof edge.text === 'undefined') {
      if (typeof edge.style !== 'undefined') {
        edgeData.arrowheadStyle = 'fill: #333'
      }
    } else {
      edgeData.arrowheadStyle = 'fill: #333'
      if (typeof edge.style === 'undefined') {
        edgeData.labelpos = 'c'
        if (conf.htmlLabels) {
          edgeData.labelType = 'html'
          edgeData.label = '' + edge.text + ''
        } else {
          edgeData.labelType = 'text'
          edgeData.style = edgeData.style || 'stroke: #333;
stroke-width: 1.5px;fill:none'
          edgeData.label = edge.text.replace(/
/g, '\n')
        }
      } else {
        edgeData.label = edge.text.replace(/
```

```
/g, '\n')
      }
    }
    // Add the edge to the graph
    g.setEdge(edge.start, edge.end, edgeData, cnt)
  })
}
```

那么根据我的脚本，用 → 表现的依赖关系，除了可视化以外，还有其他用处吗？ 其实肯定是有的，除了「花架子」，这个依赖图对于项目的部署构建也有非常重要的作用。比如在对 monorepo 项目进行构建时，因为子项目过多，导致构建时间过长。为此，我给出的方案是增量构建，如果这次改动只设计项目 A、项目 B，以及公共依赖 C，那么项目 C，项目 D 等其他项目在构建时只需要读取缓存构建结果即可。思路是很简单，但是一个直接问题是，如果检测说真正需要构建的项目呢?

举个例子，项目 A 依赖公共依赖 C，那么及时通过 git hook 拿到的 diff 表明项目 A 并没有代码变动，但是可能因为 C 变了，我们还需要重新构建项目 A（因为 A 依赖 C）。按照正常的思路，需要遍历整个项目，这样带来的问题是增加了回溯构建的可能：构建时先遍历到 A，读取缓存，再遍历到 C 时，不得不回退到 A，重新构建。解决思路就是使用一个拓扑图，根据拓扑图，按照一定的顺序进行遍历和编译构建即可。

这是我近期一个使用到拓扑图数据结构的经典场景。具体实施过程因为机密性，我不在贴代码了，对于读者来说，更重要地是体会思想，相信自己动手实现也不会困难。

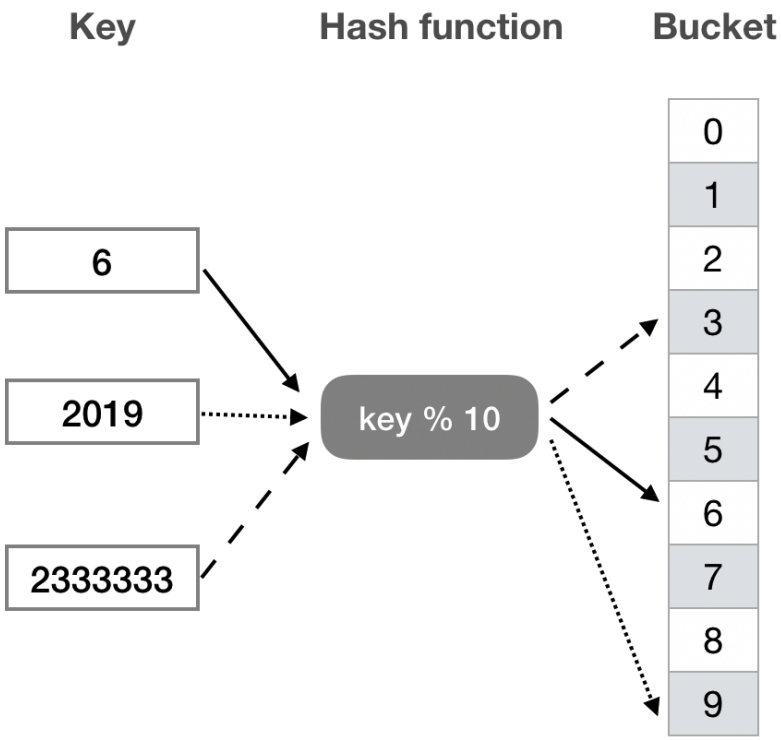
散列表（哈希表）

散列表是一种以 key-value 形式存储数据的数据结构，可以把散列表理解为一种高级的数组，这种数组的下标可以是很大的整数，浮点数，字符串甚至结构体。这种数据结构非常有用，js 里的 Map/Set/WeakMap/WeakSet 在 v8 里都是通过散列表来实现的，再比如 LRU Cache、数据库索引等非常多的场景也都能看到散列表的身影。

散列并不仅仅是一种技术，从某种意义上讲，它甚至是一种思想。接下来让我们一起揭开散列表神秘的面纱。

假如，我们要存储 key 为 6、2019、2333333 的三组数据，如果用数组来存，至少需要一个长度为 2333333 的数组来做这件事情，显然这种做法存在大量的空间浪费。

我们也可以像下图一样，准备一个长度为 10 的数组（bucket array），将每一个 key 通过一个散列函数（hash function），映射到桶数组中的一位，将 key 相应的值直接存入即可。可以看到这种方式只需使用一个长度为 10 的数组，同时查找和插入的时间复杂度都是 O(1)。这就是散列表的核心思想。



散列表中的几个概念：

桶（bucket），用来直接存放或间接指向一个数据

桶数组（bucket array）由桶组成的数组

散列函数（hash function）将 key 转换为桶数组下标的函数

上面的例子比较简单，如果我们继续在之前的基础上再存储一个 key 为 9 的数据，通过 9 % 10 计算得出的也是落在下标为 9 的 bucket 上，此时有两个不同的 key 落在了同一个 bucket 上，这一现象被称为散列冲突。

散列冲突理论上是不可避免的，我们能做的优化主要从以下两个方面入手：

精心设计桶数组长度及散列函数，尽可能降低冲突的概率

发生冲突时，能对冲突进行排解

假设不用散列表直接用数组来存储需要的数组长度为 R，用散列表存储需要的桶数组长度为 M，需要存储的元素个数为 N，则一定存在以下关系 `N < M << R`，只有这样散列表才能既保持操作的高效同时起到节省空间的效果。

其中，N / M 称为散列表的装载因子，当装载因子超过一定的阈值时，需要对桶数组扩容并 rehash。

理想的散列函数遵循以下的设计原则：

确定：同一 key 总是被映射至同一地址

高效：插入/查找/删除 excepted-O(1) 时间复杂度

满射：尽可能充分地覆盖整个桶数组空间

均匀：key 映射到桶数组各位置的概率尽量接近

常用的散列函数如下。

除余法

`hash(key) = key % M`，直接对 key 按桶数组的长度取余，这种方法非常简单，但存在以下缺陷。

存在不动点：无论桶数组长度 M 取何值，总有 `hash(0) = 0`，这与任何元素都有均等的概率被映射到任何位置的原则相违背。

零阶均匀：`[0, R)` 的关键码，平均分配至 M 个桶；但相邻关键码的散列地址也必相邻。

MAD 法 multiply-add-divide

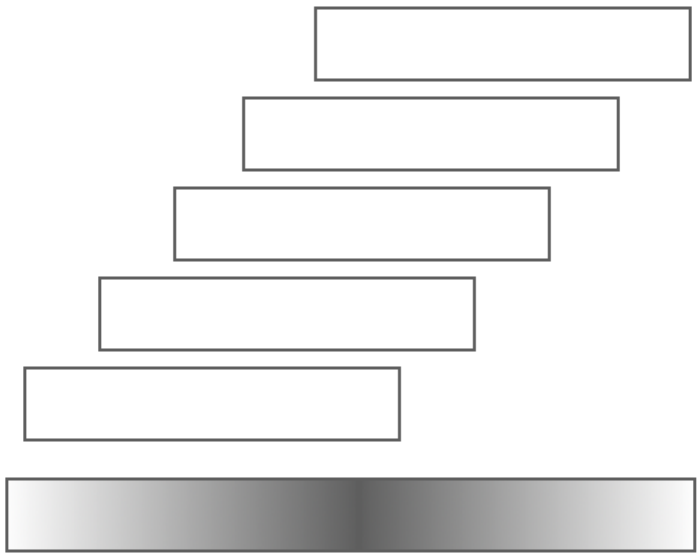
`hash(key) = (a x key + b) % M`，跟除余法相比，引入的变量 b 可以视作偏移量，可有效的消除不动点，另一个变量 a 扮演着步长的角色，也就是说原本相邻的关键码在经过散列后步长为 a，从而不再继续相邻。

平方取中 mid-square

取 `key^2` 的中间若干位，构成地址：

```
hash(123) = 512 // 保留 key^2 = 123^2 = 15219 的中间 3 位

hash(1234567) = 556 // 1234567^2 = 1524155677489
```

我们可以将一个数的平方运算，分解为一系列的左移操作以及若干次加法，从下图中不难看出，每一个数位都是由原关键码中的若干数位经求和得到的，因此两侧的数位由更少的原数位求和而得，越是居中的数位，则是由更多的原数位积累而得，因此截取居中的若干位，可使得原关键码的各数位都能对最终结果产生影响，从而实现更好的均匀性



多项式法

在实际应用中，我们的 key 不一定都是整数形式，因此往往需要一个预处理将其转换为散列码(hashcode)，然后才可以对其进一步处理为桶数组的下标地址。整个过程可以描述为 key → hashcode → bucket addr，多项式法就是一种有效的将字符串 key 转换为 hashcode 的方法 对于一个长度为 n 的字符串，其计算过程如下：

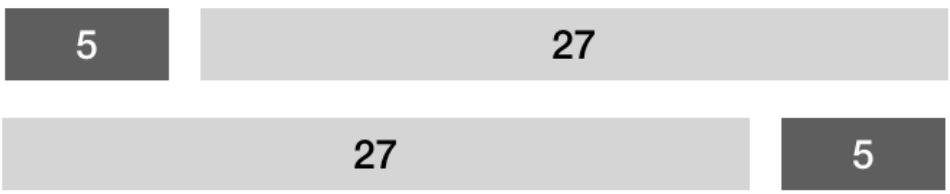
```
hash(x0 x1 ... xn-1) = x0 * a^(n-1) + x1 * a^(n-2) ... +
xn-2 * a + xn-1
// 如果上面的不是很理解，它其实等价于下面这样
(...((x0 * a + x1) * a + x2) * a + ... xn-2) * a + xn-1)
```

这个多项式可以在 O(n) 而不是 O(n2) 的时间复杂度内计算出结果，具体证明的过程这里就不详细展开了。

在实际的工程中会采用如下这种近似多项式，但更快捷的做法：

```
function hash(key) {
   let h = 0
   for (let n = key.length, i = 0; i != n; i++) {
       h = (h << 5 | h >> 27)
       h += key[i].charCodeAt()
   }
   return h >>> 0
}
```

通过一个循环依次处理字符串的每一个字符，对于每一个字符将它转换为整数后累加，在累加之前对原有的累积值，都按照 h << 5 | h >> 27 这样的规则做一个数位变换



这一不断调整累加的过程，实际上可以是作为是对以上多项式计算的近似，只不过这里消除掉了相对耗时的乘法运算，至于如何理解和解释这种近似的效果，可以作为本文课后的一项作业。

除了上文讲到的方法外，还有非常多的散列函数的方法，如折叠法、位异或法、（伪）随机数法，此类方法林林总总，每种方法都有各自的特点及应用的场景，由于篇幅原因这里就不再展开了，感兴趣的读者可以在读者群中继续研究和探讨。

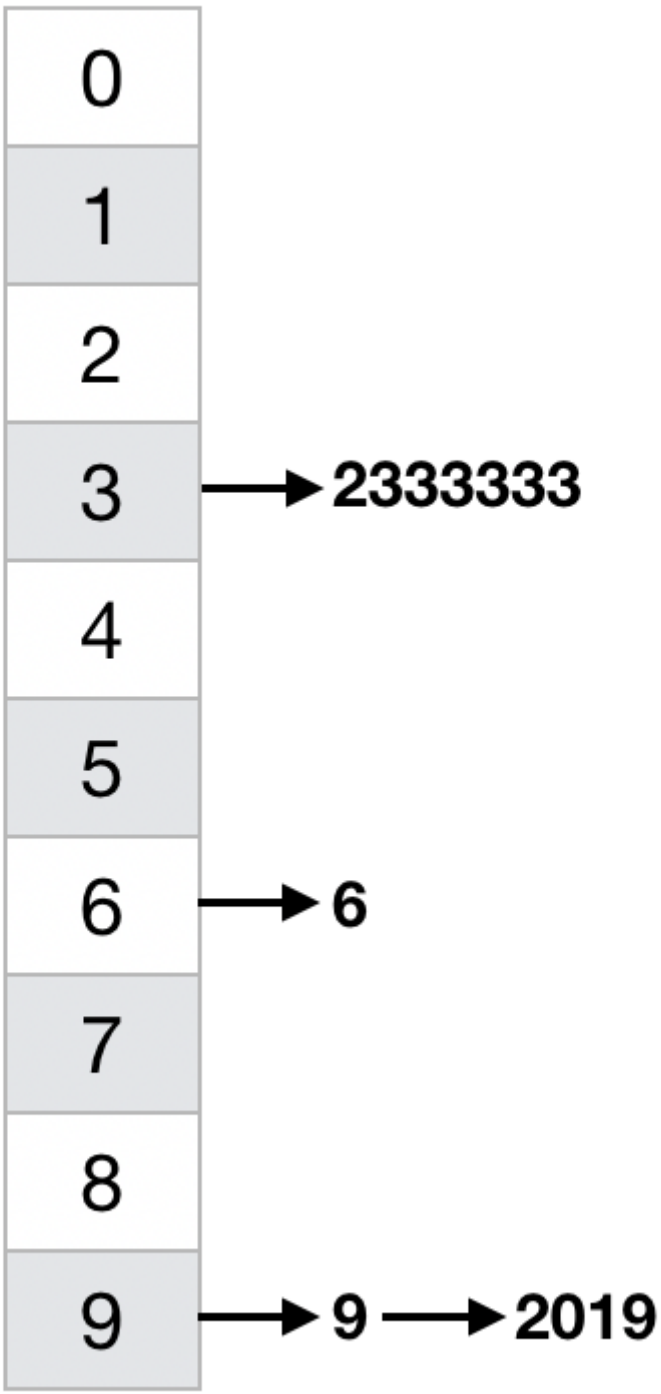
总之散列函数产生的关键码越是随机，越是没有规律就越好。

冲突解决方法

主要的处理散列表冲突的方法有开链法和探测法这两类。

开链法（linked-list chaining / seperate chaining）

每个桶存放一个指针，将冲突的 key 以链表的形式组织起来，这种处理方式最大的优点是能解决任意次数的冲突，但缺点也很明显，最极端的情况所有的 key 数据都落在一个桶上时，散列表将退化为一个链表，查找插入删除的复杂度都将变成 O(n)。



探测法（open addressing / closed hashing）

探测法所有的冲突都在这块连续的空间中加以排解，而不用像开链法那样申请额外的空间。当存入一个 key 时，所有的桶都按照某种优先级关系排成一个序列，从本该属于该 key 的桶出发，顺次查看每一个桶直到找到可用的桶。每个 key 对应的这样的一个序列，称为试探序列或者查找链，在查找 key 时，沿查找链查找有两种结果，在桶中找到了查询的 key 也就是查找成功，还有的一种可能是找到一个空桶，则说明查找失败，没有这个 key。

最简单的试探序列的生成方法叫做线性试探（Linear probing），具体做法是一旦发生冲突，则试探后一个紧邻的桶单元，直到成功或失败。这种做法的优点是无需附加的（指针、链表等）空间，缺点也很明显，以往的冲突会导致后续的冲突。

```
[hash(key) + 1] % M
[hash(key) + 2] % M
[hash(key) + 3] % M
...
```

线性试探的问题根源在于大部分的试探位置都集中在某一个相对较小的局部，因此优化线性试探的方式就是适当的拉开各次探测的间距，平方试探（Quadratic Probing）就是基于这一优化思路的具体实现方式，所谓平方试探顾名思义就是以平方数为距离，确定下一试探桶单元。

```
[hash(key) + 1^2] % M
[hash(key) + 2^2] % M
[hash(key) + 3^2] % M
...
```

相对于线性试探，平方探测的确可以在很大程度上缓解数据聚集的现象，查找链上，各桶间距线性递增，一旦冲突，可从没地跳是非之地。

散列表的实现

最后用 JavaScript 来模拟实现一下 hashtable，这里我们采用开链法来解决散列的冲突。

```
// 单向链表节点
class ForwardListNode {
   constructor(key, value) {
       this.key = key
       this.value = value
       this.next = null
   }
}
```

```
class Hashtable {
 constructor(bucketSize = 97) {
   this._bucketSize = bucketSize
   this._size = 0
   this._buckets = new Array(this._bucketSize)
 }

 hash(key) {
   let h = 0
   for (let n = key.length, i = 0; i != n; i++) {
     h = (h << 5 | h >> 27)
     h += key[i].charCodeAt()
   }
   return (h >>> 0) % this._bucketSize
 }

 // Modifiers
 put(key, value){
   let index = this.hash(key);
   let node = new ForwardListNode(key, value)

   if (!this._buckets[index]) {
     // 如果桶是空的，则直接把新节点放入桶中即可
     this._buckets[index] = node
   } else {
     // 如果桶不为空，则在链表头插入新节点
     node.next = this._buckets[index]
     this._buckets[index] = node
   }
   this._size++
   return index
 }

 delete(key) {
   let index = this.hash(key)
   if (!this._buckets[index]) {
```

```
    return false
  }

  // 添加一个虚拟头节点，方便后面的删除操作
  let dummy = new ForwardListNode(null, null)
  dummy.next = this._buckets[index]
  let cur = dummy.next, pre = dummy
  while (cur) {
    if (cur.key === key) {
      // 从链表删除该节点
      pre.next = cur.next
      cur = pre.next
      this._size--
    } else {
      pre =  cur
      cur = cur.next
    }
  }
  this._buckets[index] = dummy.next
  return true
}

// Lookup
find(key){
  let index = this.hash(key);
  // 如果对应的 bucket 为空，说明不存在此 key
  if (!this._buckets[index]) {
    return null
  }

  // 遍历对应桶的链表
  let p = this._buckets[index]
  while (p) {
    // 找到 key
    if (p.key === key) {
      return p.value
    }
```

```
      p = p.next
    }
    return null
  }

  // Capacity
  size() {
      return this._size
  }

  isEmpty() {
      return this._size == 0
  }
}
```

总结

这一节课我们介绍了和前端最为贴合的几种数据结构，虽然篇幅较长，但是内容算不上太难。一些基本概念并没有深入讲解， 因为数据结构更重要的是应用，我希望读者能够做到的是：在需要的场景，能够想到最为适合的数据结构处理问题。请读者务必掌握好这些内容，接下来的算法章节需要对数据结构有一个较为熟练地掌握和了解。

点击查看下一节

古老又新潮的函数式