javascript 中的链表结构

1.定义

很多编程语言中数组的长度是固定的，就是定义数组的时候需要定义数组的长度，所以当数组已经被数据填满的时候，需要再加入新的元素就很困难。只能说在部分变成语言中会有这种情况，在 javascript 中和 php 中数组的长度是可以任意增加的。在数组中添加和删除元素也是比较麻烦，因为要将数组中其他元素向前或者向后平移，这个在 javascript 中也不是问题，javascript 中有一个很方便的方法 splice()方法很方便的就可以添加或删除元素。

但是凡是都是相对的，javascript 中的数组也有自己的问题，他们被是成了对象，与其他语言（比如 c++和 java）相比它的效率很低。

如果在实际的使用中发现数组的效率很慢，就可以考虑使用链表来代替。数组还有个优势是可以根据键值很方便的访问数组的值，除此之外，链表在任何场合都可以代替数组。如果需要随机地访问元素，数组仍然是更好的选择。

链表是由一组节点组成的集合。每一个节点都使用一个对象的引用指向它的后续借点。指向另外一个借点的引用叫做链。

数组元素靠它们的位置进行引用，链表元素则是靠相互之间的关系进行引用。在数组中会说这个元素是数组中的第几个元素，但是在链表中就说这个元素是某个元素的后面一个元素。遍历链表就是跟着链表从链表的头元素（head）一直走到尾元素（但是不包含链表的头借点，头通常用来作为链表的接入点）。还有一个问题，链表的尾元素指向一个 null 节点。如下图 1

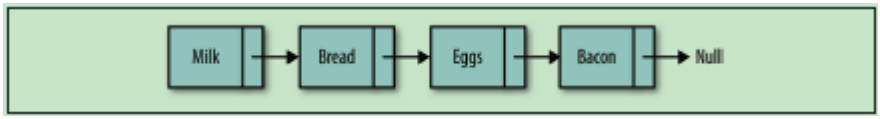

图 1

许多链表的实现都在链表前面有一个特殊的节点，叫做头节点。最后一个节点指向 null，所有最后再加上一个 null 节点。如下图 2

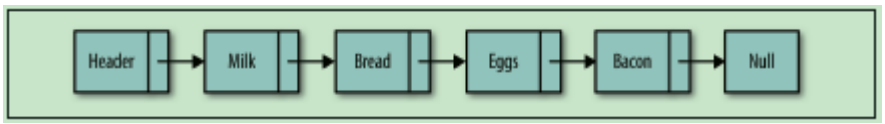

图 2

在链表中插入一个节点的效率很高。向链表中插入一个节点，需要修改它前面的节点，使其指向新加入的节点，而新加入的节点则指向前面指向的节点。如下图展示的是在 eggs 后面加上 cookies 节点。如下图 3.

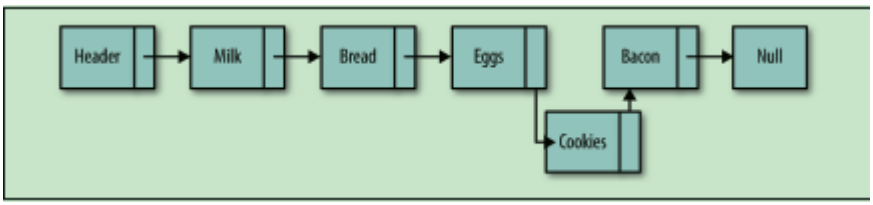

图 3

从链表中删除一个节点也很简单，将待删除的元素的前驱节点指向待删除的后续节点，同时将待删除元素指向 null 来释放。下图是一个巧合删除是的 null 元素前面的一个元素。如下图 4

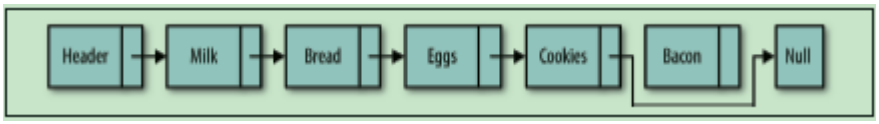

图 4

2.代码实现

在下面的链表实现中有两个类。node 类用来标识节点，LinkedList 类提供插入节点，删除节点，显示链表节点元素的方法，以及一些其他的辅助方法。

Node 类包含两个属性，element 用来保存节点上的数据，next 用来保存指向下一个节点的链接。我们使用一个构造函数来创建节点，改构造函数设置了这两个属性的值。如下：

```
function Node(element){
    this.element = element;
    this.next = null;
}
```

LinkedList 类提供了对链表进行操作的方法，该类的功能包含插入节点，在链表中查找给定的节点。该类也有一个构造函数，链表只有一个属性，那就是使用一个 node 对象来保存该链表的头结点，代码如下：

```
function LList(){
    this.head = new Node('head');
    this.find = find;
    this.insert = insert;
    //this.remove = remove;
    this.display = display;
}
```

head 节点的 next 属性被初始华为 null，当有新元素插入时，next 会指向新的元素。

插入节点的方法是 insert，该方法向链表中插入新节点的时候，需要明确指出在那个节点的前面或者后面插入。这里先讨论在一个已知节点的后面插入元素。在元素后面插入元素的时候不，需要先找到"后面"的节点。为此创建一个辅助方法 find()，该方法遍历链表，查找指定的数据，如果找到该数据，就返回保存该数据的节点，find()方法的实现的代码如下：

```
function find(item){
    var currNode = this.head;
    while (currNode.element != item){
        currNode = currNode.next;
    }
    return currNode;
}
```

find()方法演示了如何在链表上移动。首先创建一个新节点，并将链表的头节点赋给这个新创建的节点。然后再链表上进行循环，如果当前节点的 element 属性和我们要找的信息不符合，就从当前节点移动到下一个节点。如果查找成功个，该方法返回包含该数据的节点，否则返回 null。

一旦找到"后面"的节点，就可以将新节点插入链表了。首先，将新节点的 next 属性设置为"后面"节点对应的值，然后设置"后面"节点的 next 属性指向新的节点，insert()定义如下：

```
//插入一个元素
function insert(newElement, item){
    var newNode = new Node(newElement);
    var current = this.find(item);
    newNode.next = current.next;
    current.next = newNode;
}
```

最后，我们定义一个打印链表元素的方法，如下：

```
function display(){
    var currNode = this.head;
    while (!(currNode.next == null)){
        document.write(currNode.next.element + ' ');
        currNode = currNode.next;
    }
}
```

这个方法首先将链表的头赋给一个变量，然后循环遍历链表，当前节点的 next 属性为 null 的时候循环结束。为了只显示包含数据的节点，我们使用 currNode.next.element 表达式来访问节点中的数据。

最后我们用下面的代码来测试链表。在链表中保存几个美国城市"Conway"，"Russellville"，"Alma"并将他们打印出来，完整代码如下：

```
function Node(element){
    this.element = element;
    this.next = null;
}

function LList(){
    this.head = new Node('head');
    this.find = find;
    this.insert = insert;
    //this.remove = remove;
    this.display = display;
}
```

```
function find(item){
    var currNode = this.head;
    while (currNode.element != item){
        currNode = currNode.next;
    }
    return currNode;
}

//插入一个元素
function insert(newElement, item){
    var newNode = new Node(newElement);
    var current = this.find(item);
    newNode.next = current.next;
    current.next = newNode;
}

function display(){
    var currNode = this.head;
    while (!(currNode.next == null)){
        document.write(currNode.next.element + ' ');
        currNode = currNode.next;
    }
}


//测试程序
var cities = new LList();
cities.insert("Conway", "head");
cities.insert("Russellville", "Conway");
cities.insert("Alma", "Russellville");
cities.display();
```