链表的实现

基本实现

链表元素：对于单链表中的元素，由两部分组成
- 自身存储的数据，数据可以是任意类型
- 指向下一个链表元素的指针

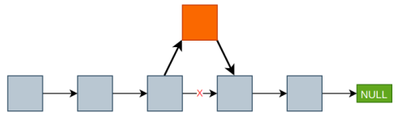
```
class Element(object):
    def __init__(self, data, next):
        self.data = data
        self.next = next
```

链表：不管是单向链表还是双向链表，链表本身的属性都是一样的
- 头结点与尾结点
- 链表大小

```
class LinkedList(object):
    def __init__(self):
        self._head = None
        self._size = 0
```

并且这些属性都应该受到保护，不应由外界进行修改

向链表插入元素



基本原理：向链表中添加元素时，只需要进行指针的操作即可。但是若在链表的头部插入，则需要进行特殊的处理，因为头结点没有前驱结点

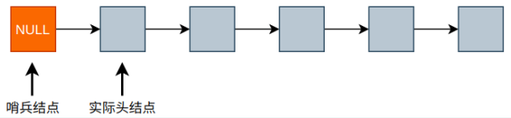
向任意位置插入元素：

```
def insert(self, index, element):
    if index < 0 or index > self._size:
        raise ValueError()

    if index == 0:
        self._head = Element(element, self._head)
    else:
        previous = self._head
        # 找到待插入索引的前一个元素
        for i in range(index - 1):
            previous = previous.next

        previous.next = Element(element, previous.next)

    self._size += 1
```

在实现中可以看到，因为头结点和其它结点的不同，使得在代码中需要对头结点进行特殊处理

哨兵(虚拟头结点)

在数据结构中，为了省去对特殊结点的处理，通常会在其之前或者之后添加一个哨兵结点，也称之为虚拟结点。哨兵结点本身不存储数据，其目的就是将特殊结点一般化



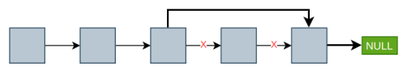
在链表实际的头结点之前添加一个哨兵节点，该节点用户无感知，只是为了编码更加方便

```
class LinkedList(object):
    def __init__(self):
        self._sentinel_head = Element(None, None)
        self._size = 0

    def insert(self, index, element):
        # 省略对index的校验
        previous = self._sentinel_head
        for i in range(index):
            previous = previous.next

        previous.next = Element(element, previous.next)

        self._size += 1
```

使用哨兵节点对整个链表类进行重构

相较于没有哨兵节点的链表，具有哨兵节点的链表在编码时将会更加的简洁

删除链表元素



基本原理：简单来说只需要两行代码即可实现链表元素的删除，假设待删除结点前一个结点为previous

```
delete_node = previous.next
previous.next = delete_node.next
delete_node.next = None
```

不好意思用了三行 ：）

```
def remove(self, index):
    # 省略对index的校验
    previous = self._sentinel_head
    for i in range(index):
        previous = previous.next

    delete_node = previous.next
    previous.next = delete_node.next
    delete_node.next = None

    self._size -= 1

    return delete_node.data
```

同样地，在有了哨兵节点以后，根本不用在编码上考虑删除的index是否为0，即删除的是否为头结点