补充阅读材料：

分词算法实现

(Python)

正向最大匹配算法

- 基本思想：
  - 1. 设自动分词词典中最长词条所含汉字个数为I
  - 2. 取被处理材料当前字符串序数中的I个字作为匹配字段，查找分词词典。若词典中有这样的一个I字词，则匹配成功，匹配字段作为一个词被切分出来，转6
  - 3. 如果词典中找不到这样的一个I字词，则匹配失败
  - 4. 匹配字段去掉最后一个汉字，I--
  - 5. 重复2-4，直至切分成功为止
  - 6. I重新赋初值，转2，直到切分出所有词为止

正向最大匹配算法(python)

- 1.初始化词典

```
# 初始化词典
# 读取词典文件
file = open('dic.txt', 'r', encoding='utf-8')
try:
    b = file.read()
finally:
    file.close()
# 将词典转化为list
dic = b.split('\n')
# 计算词典中最长词的长度
maxLen = 0
for word in dic:
    if len(word) > maxLen:
        maxLen = len(word)
```

Dic.txt

下载
手机
互联网
在线
博客
点击
免费版
搜狐
电子商务
中华人民共和国
诺基亚
链接
本站
工作人员
……
……

正向最大匹配算法

- 2.开始分词

```
# 待分词字符串
text = '自然语言处理是一门非常棒的课程'

# 用于存储切分好的词的列表
segList = []
while len(text) > 0:
    length = maxLen
    # 如果最大分词长度大于待切分字符串长度，则切分长度设置为待切分字符串长度
    if len(text) < maxLen:
        length = len(text)
    # 正向取字符串中长度为length的子串
    tryWord = text[0:length]
    while tryWord not in dic:
        # 若子串长度为1，跳出循环
        if len(tryWord) == 1:
            break
        # 截掉子串尾部一个字，用剩余部分到字典中匹配
        tryWord = tryWord[0:len(tryWord)-1]
    # 将匹配成功的词加入到分词列表中
    segList.append(tryWord)
    # 将匹配成功的词从待分词字符串中去除，继续循环，直到分词完成
    text = text[len(tryWord):]
```

逆向最大匹配算法

- 1.初始化词典

```
# 初始化词典
# 读取词典文件
file = open('dic.txt', 'r', encoding='utf-8')
try:
    b = file.read()
finally:
    file.close()
# 将词典转化为list
dic = b.split('\n')
# 计算词典中最长词的长度
maxLen = 0
for word in dic:
    if len(word) > maxLen:
        maxLen = len(word)
```

逆向最大匹配算法

- 2.开始分词

```
# 待分词字符串
text = "自然语言处理是一门非常棒的课程"
# 用于存储切分好的词的列表
segList = []

while len(text) > 0:
    length = maxLen
    # 如果最大分词长度大于待切分字符串长度，则切分长度设置为待切分字符串长度
    if len(text) < maxLen:
        length = len(text)
    # 逆向取字符串中长度为length的子串
    tryWord = text[len(text)-length:]
    while tryWord not in dic:
        # 若子串长度为1，跳出循环
        if len(tryWord) == 1:
            break
        # 截掉子串头部一个字，用剩余部分到字典中匹配
        tryWord = tryWord[1:]
    # 将匹配成功的词插入到分词列表的头部
    segList.insert(0, tryWord)
    # 将匹配成功的词从待分词字符串中去除，继续循环，直到分词完成
    text = text[:len(text)-len(tryWord)]
```

算法分析

- 词典
  - dic.txt 4.6M
  - 427452个词
- 之前代码中使用python中list对字典进行存储
  - 例：[中国, 自然, 语言, 处理]
- 限制算法性能之处便是对字典的查找，尝试其他的数据结构对字典进行存储和查找
  - 集合set
  - 字典dict

算法分析

- list, set, dict查找速度比较

数据结构	查询次数	消耗时间(ms)
list	1000	466873
set	1000	472
set	10000	4781
set	100000	41444
dict	100000	46741

- 消耗时间是运行十次取平均值后的结果

算法分析

- list, set, dict内存占用比较

数据结构	内存占用(bytes)
list	3617184
set	16777440
dict	25165920

- set占用内存约是list的4.6倍
- dict占用内存约是list的7倍
- set和dict采用空间换时间策略

算法改进

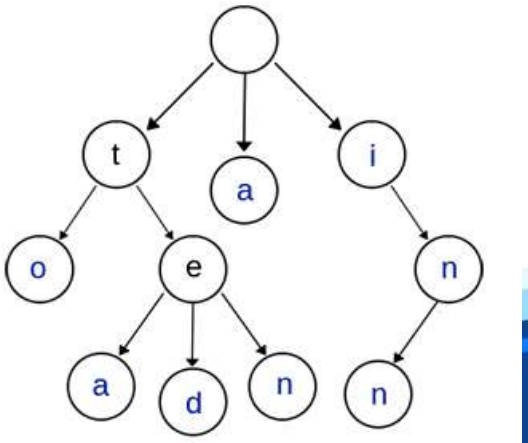
- 由上述分析得，可将使用set代替list对词典进行存储，查询效率提升三个数量级
- set和dict的实现均是基于哈希表（hash table），而哈希表查找的时间复杂度为O(1)

```
dic = b.split('\n')
```

```
dic = set(b.split('\n'))
```

Trie树

- Trie树，又叫字典树、前缀树（Prefix Tree），是一种树型结构
- 如右图Trie树，包含了{a, to, tea, ted, ten, I, in, inn}
- 三点基本性质
  - 根节点不包含字符，除根节点外的每一个子节点都包含一个字符
  - 从根节点到**某一个节点**，路径上经过的字符连接起来，为该节点对应的字符串。
  - 每个节点的所有子节点包含的字符互不相同
- 通常会在节点中设置一个标志，表示到该节点处是否组成一个完整的词



Trie树实现

```python
class TrieNode:
    # 构造函数
    def __init__(self, character, terminal, children):
        self.character = character
        self.terminal = terminal
        self.children = children

    # 返回当前节点Terminal的值
    def is_terminal(self):
        return self.terminal

    # 设置当前节点Terminal的值
    def set_terminal(self, terminal):
        self.terminal = terminal

    # 获得当前节点的字符
    def get_character(self):
        return self.character

    # 设置当前节点的字符
    def set_character(self, character):
        self.character = character
```

Trie树实现

```
# 获取当前节点的子节点
def get_children(self):
    return self.children

# 获取指定的一个子节点
def get_child(self, character):
    if character not in self.children:
        return None
    return self.children[character]

# 获取子节点，若不存在则创建一个
def get_child_if_not_exist_then_creat(self, character):
    child = self.get_child(character)
    if not child:
        child = TrieNode(character, False, {})
        self.add_child(child)
    return child

# 添加子节点
def add_child(self, child):
    self.children[child.character] = child

# 移除子节点
def remove_child(self, child):
    self.children[child.character] = None
```

词典构建

```
# 在trie树中查找
def contain(astr):
    # 去除空格
    astr = astr.replace(' ', '')
    # 若长度小于1，则查找失败
    if len(astr) < 1:
        return False
    # 从根节点开始
    node = ROOT_NODE
    # 逐个字符查找
    for i in astr:
        child = node.get_child(i)
        if not child:
            return False
        else:
            # 切换当前节点
            node = child
    # 在trie树中存在字符串，若是完整字符串则查找成功，否则查找失败
    return node.is_terminal()
```

词典构建

```
# 添加所有词
def add_all(word_list):
    for word in word_list:
        add(word)

# 向trie中添加词
def add(word):
    # 去除空格
    word = word.replace(' ', '')

    # 若长度小于1，则不添加
    if len(word) < 1:
        return

    # 从根节点开始
    node = ROOT_NODE

    # 逐个字符查找，遇到不存在的字符则创建
    for i in word:
        child = node.get_child_if_not_exist_then_creat(i)
        # 切换当前节点
        node = child

    # 添加完成，设置最后一个节点为合法
    node.set_terminal(True)
```

实验结果

- 经实验测试，对于含有427452个词的词典构建Trie树共需427140ms
- Trie树查找性能：

数据结构	查询次数	消耗时间(ms)
list	1000	466873
set	1000	472
set	10000	4781
set	100000	41444
dict	100000	46741
trie	1000	2705
trie	10000	27435
trie	100000	258671

实验结果

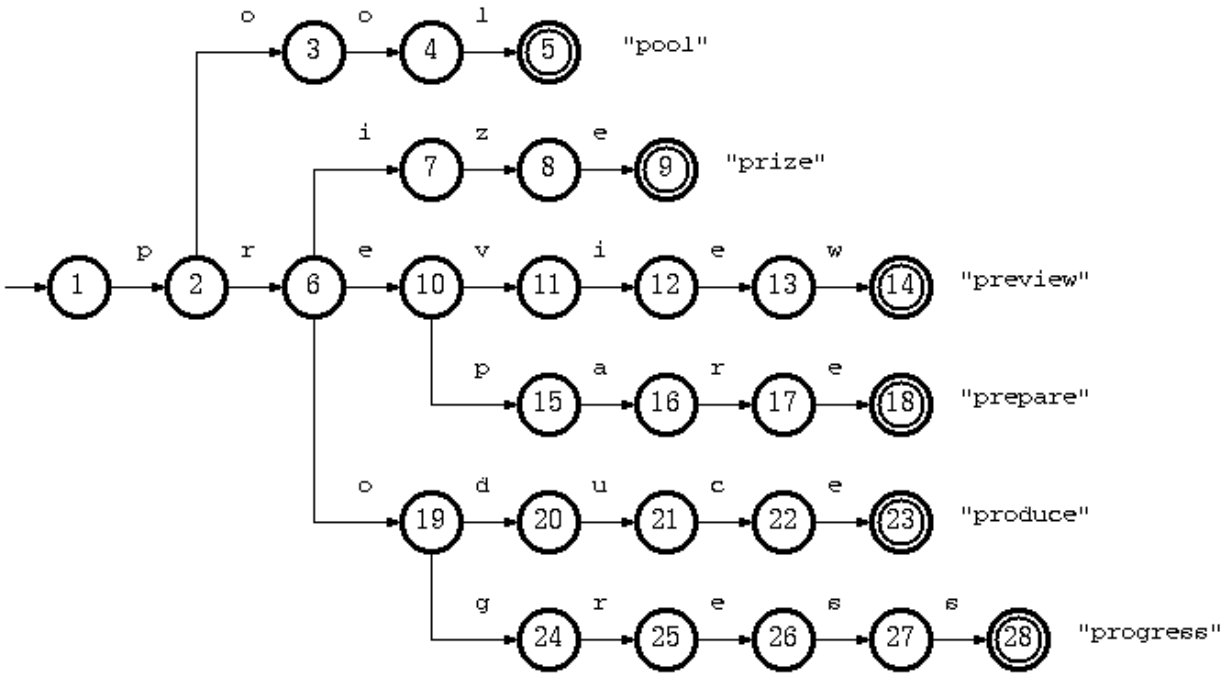
- 参考上表可发现，使用trie树进行查询的速率比set和dict慢了近6倍
- 而trie树占用内存如下表所示：

数据结构	内存占用(bytes)
list	3617184
set	16777440
dict	25165920
trie	786528

- 可以看出，trie树在占用内存方面要比set少20倍，通过处理相同前缀的词，trie有效的节省了内存开支

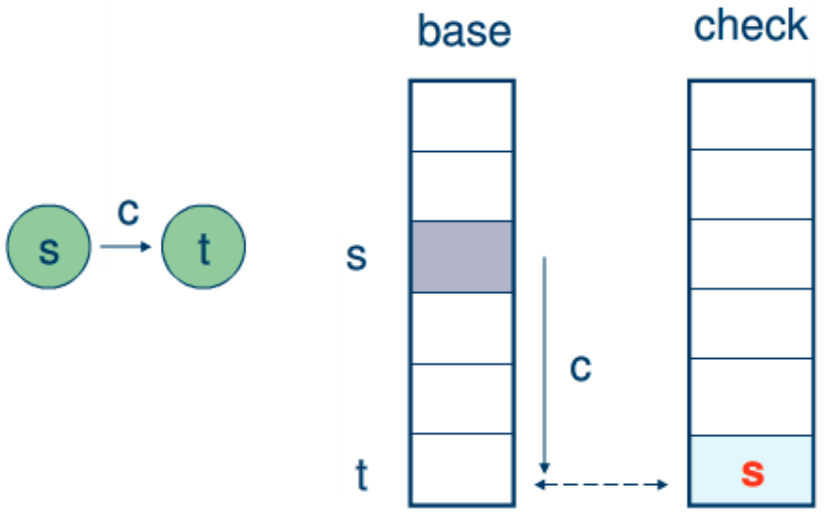
双数组Trie树

- 将Trie看做一个确定的有限状态自动机（DFA），每个节点表示自动机的一个状态，接受不同的字符，转移到不同的状态



双数组Trie树

- 两个数组：base[]、 check[]
  - base: 用于存储每个状态的转移基数
  - check: 用于存储当前状态的前一状态
- 例：当前状态为s，接受字符c，转移到状态t，需要满足：
  - base[s] + c.code = t
  - check[t] = s
  - c.code是字符c的编号



双数组Trie树

- 基本思想
  - 对于词典中所有的字符进行编号（中文对汉字编号，英文对字母编号），假设首字为i的词有n个，它们的第二个字的编号分别为a1，a2,…,an，则初始状态s接受字符i跳转到状态t，状态t分别接受这n个第二个字，跳转到状态base[t]+a1, base[t]+a2,…, base[t]+an。
  - 若base和check的值同时为空，说明该位置为空。如果base为负，说明到该状态为止，之前的字符组成一个合法的词。
  - 在查询时，仅需将词转化为词的编号，做状态转移，经过数次加法后可迅速返回查询结果，速度非常快

参考文献

- 中文分词算法 之 基于词典的正向最大匹配算法
  http://yangshangchuan.iteye.com/blog/2031813
- Trie树（ Prefix Tree ）介绍
  https://blog.csdn.net/lisonglisonglisong/article/details/45584721
- Trie树详解 https://www.jianshu.com/p/6f81da81bd02
- Trie树优化算法：Double Array Trie 双数组Trie
  https://blog.csdn.net/heiyeshuwu/article/details/42526461
- Java实现双数组Trie树(DoubleArrayTrie,DAT)
  https://blog.csdn.net/dingyaguang117/article/details/7608568

高级要求：
研习list，set，dict等数据结构的源码