基本解法

Java代码

```
class Solution {
    public List<String> topKFrequent(String[] words, int k) {
        //初始化一个map，用于存放每个单词出现频率
        Map<String, Integer> countFreq = new HashMap();
        //遍历整个字符串数组，如果某个单词出现，频率加1
        for (String word: words) {
            countFreq.put(word, countFreq.getOrDefault(word, 0) + 1);
        }
        //初始化字符串数组用于存储排序后的单词
        List<String> result = new ArrayList(countFreq.keySet());
        //按照频率排序，如果频率相同，则选择字母顺序靠前的单词，否则选择出现频率高的单词
        Collections.sort(result, (word1, word2) -
> countFreq.get(word1).equals(countFreq.get(w2)) ?
                word1.compareTo(word2) : countFreq.get(word2) - countFreq.get(w
ord1));
        //返回排序后前k个单词
        return result.subList(0, k);
    }
}
```

优化解法

Java代码

```
class Solution {
    public List<String> topKFrequent(String[] words, int k) {
        //初始化一个map，用于存放每个单词的出现频率
        Map<String, Integer> countFreq = new HashMap();
        //遍历整个字符串数组，如果某个单词出现，频率加1
        for (String word: words) {
            countFreq.put(word, countFreq.getOrDefault(word, 0) + 1);
        }
        //初始化一个优先级队列，排序方式为优先按频率从高到低排序，如果频率相同，则按字母顺序
        PriorityQueue<String> heapSort = new PriorityQueue<String>(
                (word1, word2) -
> countFreq.get(word1).equals(countFreq.get(word2)) ?
                word2.compareTo(word1) : countFreq.get(word1) - countFreq.get(w
ord2) );
        //对于每一个单词，入队，如果队列大小超过k，则弹出队首，维持队列大小为k
        for (String word: countFreq.keySet()) {
```

```
            heapSort.offer(word);
            if (heapSort.size() > k) heapSort.poll();
        }
        //最后队列中存储的k个单词为出现频率最高的单词，逆序输出
        List<String> ans = new ArrayList();
        while (!heapSort.isEmpty()) ans.add(heapSort.poll());
        Collections.reverse(ans);
        return ans;
    }
}
```

Python代码

```
class Solution(object):
    def topKFrequent(self, words, k):
        count = collections.Counter(words)
        heap = [(-freq, word) for word, freq in count.items()]
        heapq.heapify(heap)
        return [heapq.heappop(heap)[1] for _ in xrange(k)]
```

最优解法

Java代码

```
class Solution {
    public List<String> topKFrequent(String[] words, int k) {
        Map<String, Integer> map = new HashMap<>();
        for(String word:words){
            map.put(word, map.getOrDefault(word, 0)+1);
        }
        Trie[] buckets = new Trie[words.length];
        for(Map.Entry<String, Integer> e:map.entrySet()){
            //遍历每个单词，将其加入它所在桶的trie树
            String word = e.getKey();
            int freq = e.getValue();
            if(buckets[freq]==null){
                buckets[freq] = new Trie();
            }
            buckets[freq].addWord(word);
        }
        List<String> ans = new LinkedList<>();
        for(int i = buckets.length-1;i>=0;i--){
      //对于每个桶中的trie树，将单词按照字典序排序，和k比较
            if(buckets[i]!=null){
                List<String> l = new LinkedList<>
();
```

```
                buckets[i].getWords(buckets[i].root, l);
                if(l.size()<k){
                    ans.addAll(l);
                    k = k - l.size();
                }
                else {
                   for(int j = 0;j<=k-1;j++){
                       ans.add(l.get(j));
                   }
                    break;
                }
            }
        }
        return ans;
    }
}
//构建trie树
class Trie {
    TrieNode root = new TrieNode();
    public void addWord(String word){
        TrieNode cur = root;
        for(char c:word.toCharArray()){
            if(cur.children[c-'a']==null){
                cur.children[c-'a'] = new TrieNode();
            }
            cur = cur.children[c-'a'];
        }
        cur.word = word;
    }
   //对于每个桶中同样频率的单词，用DFS得到字典序顺序
    public void getWords(TrieNode node, List<String> ans){
        if(node==null){
            return;
        }
        if(node.word!=null){
            ans.add(node.word);
        }
        for(int i = 0;i<=25;i++){
            if(node.children[i]!=null){
                getWords(node.children[i], ans);
            }
        }

    }
    //定义TrieNode
    class TrieNode {
        TrieNode[] children;
        String  word;
        TrieNode() {
```

```
            this. children  = new TrieNode[26];
            this.word = null;
     }

}
```

C++代码

```
#define new_pair pair<string, int>

struct comp{
    bool operator()(const new_pair& x, const new_pair& y){
        if(x.second == y.second)
            return x.first > y.first;
        return x.second < y.second;
    };
};

class TrieNode{
public:
    unordered_map<char, TrieNode*> children;
    char val;
    int count = 0;
    string word;

    TrieNode(){}

    TrieNode(char v){
        this->val = v;
    }
};

//构建tried树
class Trie{
private:
    TrieNode* root;

public:
    Trie(){
        this->root = new TrieNode();
    }

    TrieNode* get_node(){
        return root;
    }
    //对于传入的单词，构建trie树
    void insert(string word){
        TrieNode* temp = root;
```

```
        for(char ch : word){
            if(temp->children.find(ch) == temp->children.end()){
                temp->children[ch] = new TrieNode(ch);
            }
            temp = temp->children[ch];
        }

        temp->count++; temp->word = word;
    }
    //优先队列统计每个单词出现次数，按频率排序
    void preorder(TrieNode* curr, priority_queue<new_pair, vector<new_pair>,
comp>& que){
        if (curr == NULL) return;

        for(auto it : curr->children)
        {
            if (it.second->count)
            que.push({ it.second->word, it.second->count });

            preorder(it.second, que);
        }
    }
};

class Solution {
public:
    vector<string> topKFrequent(vector<string>& words, int k) {
        Trie* root = new Trie();
        vector<string> result;

        for(string word : words){
            root->insert(word);
        }

        priority_queue<new_pair, vector<new_pair>, comp> que;

        root->preorder(root->get_node(), que);

        while(!que.empty()){
            if(result.size() == k) break;
            result.push_back(que.top().first);
            que.pop();
        }

        return result;
    }
};
```

Python代码

```
class Node:
    def __init__(self, end=False):
        self.children = dict()
        self.end = end
        self.word = ''
#构建trie树
class Trie:
    def __init__(self):
        self.head = Node("")
    def addWord(self, word):
        node = self.head
        for c in word:
            if c not in node.children:
                node.children[c] = Node()
            node = node.children[c]
        node.end = True
        node.word = word
    #对于每个桶中同样频率的单词，用DFS得到字典序顺序
    def getSortedWord(self):
        lis = []
        def dfs(node):
            if node.end == True:
                lis.append(node.word)
            for i in range(26):
                c = chr(i+97)
                if c in node.children:
                    dfs(node.children[c])
        dfs(self.head)
        return lis



class Solution:
    def topKFrequent(self, words: List[str], k: int) -> List[str]:
        frequents = collections.defaultdict(set)
        frequents_reverse = dict()

        max_frequent = 0
        for word in words:
            if word not in frequents_reverse:
                frequents_reverse[word] = 0
                frequents[0].add(word)
            cnt = frequents_reverse[word]
            frequents[cnt].remove(word)
            frequents[cnt+1].add(word)
            frequents_reverse[word] = cnt + 1
```

```
            max_frequent = max(max_frequent, cnt+1)

        ans = []

        frequents_lis = []
        l = 0
        for i in range(max_frequent+1)[::-1]:
            trie = Trie()
            for word in frequents[i]:
                trie.addWord(word)
            ll = trie.getSortedWord()
            if len(ll) < k:
                ans = ans + ll
                k -= len(ll)
            else:
                ans = ans + ll[:k]
                k=0
                return ans
        return ans
```