题目描述（中等难度）

139. Word Break

Medium 2693 144 Favorite Share

Given a **non-empty** string *s* and a dictionary *wordDict* containing a list of **non-empty** words, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.

Note:

- The same word in the dictionary may be reused multiple times in the segmentation.
- You may assume the dictionary does not contain duplicate words.

Example 1:

```
Input: s = "leetcode", wordDict = ["leet", "code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
```

Example 2:

```
Input: s = "applepenapple", wordDict = ["apple", "pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
             Note that you are allowed to reuse a dictionary word.
```

Example 3:

```
Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
Output: false
```

给一个字符串，和一些单词，问字符串能不能由这些单词构成。每个单词可以用多次，也可以不用。

解法一 回溯

来一个简单粗暴的方法，利用回溯法，用 `wordDict` 去生成所有可能的字符串。期间如果出现了目标字符串 `s` ，就返回 `true` 。

```java
public boolean wordBreak(String s, List<String> wordDict) {
    return wordBreakHelper(s,wordDict,"");
}
//temp 是当前生成的字符串
private boolean wordBreakHelper(String s, List<String> wordDict, String temp) {
    //如果此时生成的字符串长度够了，就判断和目标字符日是否相等
    if(temp.length() == s.length()){
        if(temp.equals(s)){
            return true;
        }else{
            return false;
        }
    }
    //长度超了，就返回 false
    if(temp.length() > s.length()){
        return false;
    }
    //考虑每个单词
    for(int i = 0;i < wordDict.size(); i++){
        if(wordBreakHelper(s,wordDict,temp + wordDict.get(i))){
            return true;
        }
    }
    return false;
}
```

意料之中，超时了

Submission Detail

26 / 36 test cases passed.

Status: Time Limit Exceeded

Submitted: 1 day, 4 hours ago

```
Last executed input:  "bccdbacdbdacddabbaaaadababadad"
["cbc","bcda","adb","ddca","bad","bbb","dad","dac","ba","aa","bd","abab","bb","dbda","cb","caccc","d","dd","aad
b","cc","b","bcc","bcd","cd","cbca","bbd","ddd","dabb","ab","acd","a","bbcc","cdcbd","cada","dbca","ac","abac
d","cba","cdb","dbac","aada","cdcda","cdc","dbc","dbcb","bdb","ddbdd","cadaa","ddbc","babb"]
```

让我们考虑优化的方法。

在递归出口的地方优化一下。

之前是在长度相等的时候，开始判断字符串是否相等。

很明显，字符串长度相等之前我们其实就可以判断当前是不是符合了。

例如 `temp = "abc"` ，如果 `s = "dddefg"` ，虽然此时 `temp` 和 `s` 的长度不相等。但因为前缀已经不同，所以后边无论是什么都不可以了。此时就可以返回 `false` 了。

所以递归出口可以从头判断每个字符是否相等，不相等就直接返回 `false` 。

```java
for (int i = 0; i < temp.length(); i++) {
    if (s.charAt(i) != temp.charAt(i)) {
        return false;
    }
}
```

然后代码就是下边的样子。

```java
public boolean wordBreak(String s, List<String> wordDict) {
    return wordBreakHelper(s, wordDict, "");
}

private boolean wordBreakHelper(String s, List<String> wordDict, String temp) {
    if (temp.length() > s.length()) {
        return false;
    }
    //判断此时对应的字符是否全部相等
    for (int i = 0; i < temp.length(); i++) {
        if (s.charAt(i) != temp.charAt(i)) {
            return false;
        }
    }
    if (s.length() == temp.length()) {
        return true;
    }
    for (int i = 0; i < wordDict.size(); i++) {
        if (wordBreakHelper(s, wordDict, temp + wordDict.get(i))) {
            return true;
        }
    }
    return false;
}
```

遗憾的是，依旧是超时

Submission Detail

29 / 36 test cases passed.

Status: Time Limit Exceeded

Submitted: 23 hours, 23 minutes ago

```
Last executed input:  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab"
["a","aa","aaa","aaaa","aaaaa","aaaaaa","aaaaaaa","aaaaaaaa","aaaaaaaaa","aaaaaaaaaa"]
```

发现上边的例子答案很明显是 `false` ，因为 `s` 中的 `b` 字母在 `wordDict` 中并没有出现。

所以我们可以先遍历一遍 `s` 和 `wordDict` ，从而确定 `s` 中的字符是否在 `wordDict` 中存在，如果不存在可以提前返回 `false` 。

所以代码可以继续优化。

```java
public boolean wordBreak(String s, List<String> wordDict) {
    HashSet<Character> set = new HashSet<>();
    //将 wordDict 的每个字母放到 set 中
    for (int i = 0; i < wordDict.size(); i++) {
        String t = wordDict.get(i);
        for (int j = 0; j < t.length(); j++) {
            set.add(t.charAt(j));
        }
    }
    //判断 s 的每个字母在 set 中是否存在
    for (int i = 0; i < s.length(); i++) {
        if (!set.contains(s.charAt(i))) {
            return false;
        }
    }
    return wordBreakHelper(s, wordDict, "");
}

private boolean wordBreakHelper(String s, List<String> wordDict, String temp) {
    if (temp.length() > s.length()) {
        return false;
    }
    for (int i = 0; i < temp.length(); i++) {
        if (s.charAt(i) != temp.charAt(i)) {
            return false;
        }
    }
    if (s.length() == temp.length()) {
        return true;
    }
    for (int i = 0; i < wordDict.size(); i++) {
        if (wordBreakHelper(s, wordDict, temp + wordDict.get(i))) {
            return true;
        }
    }
    return false;
}
```

令人悲伤的是

Submission Detail

32 / 36 test cases passed.

Status: Time Limit Exceeded

Submitted: 4 hours, 26 minutes ago

```
Last executed input:  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabaabaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
["aa","aaa","aaaa","aaaaa","aaaaaa","aaaaaaa","aaaaaaaa","aaaaaaaaa","aaaaaaaaaa","ba"]
```

还有 `5` 个 `test` 没有通过。还有什么可以优化的地方呢?

是时候拿出绝招了，在前边的题已经用过很多很多次， `memoization` 技术。思想就是把回溯中已经考虑过的解存起来，第二次回溯过来的时候可以直接使用。

这里的话，我们可以用一个 `HashMap` ， `key` 的话就存 `temp` ， `value` 的话就代表以当前 `temp` 开始的字符串，经过后边的尝试是否能达到目标字符串 `s` 。

```java
public boolean wordBreak(String s, List<String> wordDict) {
    HashSet<Character> set = new HashSet<>();
    for (int i = 0; i < wordDict.size(); i++) {
        String t = wordDict.get(i);
        for (int j = 0; j < t.length(); j++) {
            set.add(t.charAt(j));
        }
    }
    for (int i = 0; i < s.length(); i++) {
        if (!set.contains(s.charAt(i))) {
            return false;
        }
    }
    return wordBreakHelper(s, wordDict, "", new HashMap<String,Boolean>());
}

private boolean wordBreakHelper(String s, List<String> wordDict, String temp, HashMap<String, Boolean> hashMap) {
    if (temp.length() > s.length()) {
        return false;
    }
    //之前是否存过
    if(hashMap.containsKey(temp)){
        return hashMap.get(temp);
    }
    for (int i = 0; i < temp.length(); i++) {
        if (s.charAt(i) != temp.charAt(i)) {
            return false;
        }
    }
    if (s.length() == temp.length()) {
        return true;
    }
    for (int i = 0; i < wordDict.size(); i++) {
        if (wordBreakHelper(s, wordDict, temp + wordDict.get(i), hashMap)) {
            //结果放入 hashMap
            hashMap.put(temp, true);
            return true;
        }
    }
    //结果放入 hashMap
    hashMap.put(temp, false);
    return false;
}
```

这次就成功通过了。

解法二 分治

换一种思想，分治，也就是大问题转换为小问题，通过小问题来解决。

这个想法前边已经做过很多很多题了，大家可以参考 97 题 、115 题 等等。

我们现在要判断目标串 `s` 是否能由 `wordDict` 构成。

我们用 `dp[i,j)` ，表示从 `s` 的第 `i` 个字符开始，到第 `j` 个字符的前一个结束的字符串是否能由 `wordDict` 构成。

假如我们知道了 `dp[0,1) dp[0,2) dp[0,3)...dp[0,len - 1)` ，也就是除 `s` 本身的所有子串是否能由 `wordDict` 构成。

那么我们就可以知道

```
dp[0,len) =  dp[0,1) && wordDict.contains(s[i,len))
             || dp[0,2) && wordDict.contains(s[2,len))
             || dp[0,3) && wordDict.contains(s[3,len))
             ...
             || dp[0,len - 1) && wordDict.contains(s[len - 1,len))
```

`dp[0,len)` 就代表着 `s` 是否能由 `wordDict` 构成。有了上边的转移方程，就可以用递归写出来了。

```java
public boolean wordBreak(String s, List<String> wordDict) {
    HashSet<String> set = new HashSet<>();
    for (int i = 0; i < wordDict.size(); i++) {
        set.add(wordDict.get(i));
    }
    return wordBreakHelper(s, set);
}

private boolean wordBreakHelper(String s, HashSet<String> set) {
    if (s.length() == 0) {
        return true;
    }
    for (int i = 0; i < s.length(); i++) {
        if (set.contains(s.substring(i, s.length())) && wordBreakHelper(s.substring(0, i), set)) {
            return true;
        }
    }
    return false;
}
```

如果不做任何处理，依旧会得到超时。

Submission Detail

31 / 36 test cases passed.

Status: Time Limit Exceeded

Submitted: 5 minutes ago

```
Last executed input:  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
["a","aa","aaa","aaaa","aaaaa","aaaaaa","aaaaaaa","aaaaaaaa","aaaaaaaaa","aaaaaaaaaa"]
```

所有， `memoization` 又来了，和之前一样将中间结果存储起来。

```java
public boolean wordBreak(String s, List<String> wordDict) {
    HashSet<String> set = new HashSet<>();
    for (int i = 0; i < wordDict.size(); i++) {
        set.add(wordDict.get(i));
    }
    return wordBreakHelper(s, set, new HashMap<String, Boolean>());
}

private boolean wordBreakHelper(String s, HashSet<String> set, HashMap<String, Boolean> map) {
    if (s.length() == 0) {
        return true;
    }
    if (map.containsKey(s)) {
        return map.get(s);
    }
    for (int i = 0; i < s.length(); i++) {
        if (set.contains(s.substring(i, s.length())) && wordBreakHelper(s.substring(0, i), set, map)) {
            map.put(s, true);
            return true;
        }
    }
    map.put(s, false);
    return false;
}
```

当然除了递归中存储，我们也可以直接用动态规划的思想，求一个结果就保存一个结果。

用 `dp[i]` 表示字符串 `s[0,i)` 能否由 `wordDict` 构成。

```java
public boolean wordBreak(String s, List<String> wordDict) {
    HashSet<String> set = new HashSet<>();
    for (int i = 0; i < wordDict.size(); i++) {
        set.add(wordDict.get(i));
    }
    boolean[] dp = new boolean[s.length() + 1];
    dp[0] = true;
    for (int i = 1; i <= s.length(); i++) {
        for (int j = 0; j < i; j++) {
            dp[i] = dp[j] && set.contains(s.substring(j, i));
            if (dp[i]) {
                break;
            }
        }
    }
    return dp[s.length()];
}
```

总

解法一的回溯优化主要就是剪枝，让一些提前知道结果的解直接结束，不进入递归。解法二的想法，就太常用了，从递归到 `memoization` 再到动态规划，其实本质都是一样的。