XDictionary 词典类通用工作流程

By Wensong He

提纲：

- *词典类简介*
- *词典类做了哪些事？*
- *词典类的剥离*
- *设计*

1. 词典类简介

词典类的功能是用于“查找”，输入是键盘符上的字符串，输出是查找到的一列候选词句。

针对不同的“查找”需求，词典类可以分为许多种，例如：

系统词典类 — 提供系统词库的“查找”需求
用户词典类 — 提供用户词库的“查找”需求
英汉词典类 — 提供英汉词库的“查找”需求
人名词典类 — 提供人名词库的“查找”需求
整句词典类 — 提供整句、长句的“查找”需求

等等。

可以看出以上加了引号的**“查找”**不再是一般意义的**“匹配”**了，也可以是**动态的去识别、构造**，人名、整句等就属于可以动态构造的情况，其他人也可以自己构造一个词典类，比如：地名词典类，当然具体实现方式可以采用静态的词表也可以是动态生成的。

因为对于词典类的使用者来说，仅仅是输入字符串，便能得到一列候选词句，所以从这个角度理解便相当于“查找”一个词典了。

2. 词典类做了哪些事？

因为可以衍生出无数个词典类，所以我们希望从中寻找一些共同点，以便能够尽可能通用、简单、快速的去构造一个自己的词典类，并将其加入到整个输入体系中，使其立即生效。

我们先来看下一个词典类需要做哪些事情：

1）读取配置

词典类需要了解当前输入体系的配置，以便能够有针对性的反馈结果，例如：

输入串是拼音还是英文？是双拼还是全拼？或者是五笔？根据不同的字符串采取不同的切分处理方案。

2）数据加载

数据通常是词表或者一些词的频次关系，以便在“查找”的时候使用。

3）切分输入串

因为词典类如何智能也无法一次性的直接将输入串翻译到想要的文字，所以需要先对输入串进行切分，然后对逐个子串进行“查找”或者称为“翻译”。

切分的结果可能有很多种，例如全拼串：

xianshi

可以是 xian’shi，也可以是 xi’an’shi。

所以会得到一列候选切分结果。

4）“查找”词典

有了切分结果后，便可以开始“查找”的过程了，从最简单的一一对应的方式

去“匹配”，如：

xian’shi - 现实

xi’an’shi – 西安市

再到复杂的有上下文参与计算的方式去“查找”，例如：整句、人名等动态计算。

5）结果排序并返回

如果需要在返回结果列表之前还需要先按一定的规则进性排序，也可以直接返回。

3. 词典类的剥离

根据以上流程我们知道词典类大概要做哪些事情了，日后也许会有许多个词典类，并且由不同的开发者来实现，让每个开发者都去实现一遍所有的功能是最不易于扩展的方式！

因此，站在一位新的开发者的角度，希望有如下可扩展的能力：

- ***可重用的数据加载方式***

  当第一位开发者写了某种加载方式来载入词表时，是否能留有该加载方式的副本？包括数据结构及相应的功能函数…

  也许你会想要 trie 树的加载方式（便于查找相同字符前缀的结果），hash 的加载方式（快速命中）等等。

  动态识别还需要加载“统计数据”，应该也留有一个副本，也许某位开发者想要利用统计数据，并尝试不同的算法来制作一个词典类就会用到该副本了。

- ***可重用的切分方式***

  类似的还需要一些可重用的切分方式。

- ***可重用的查找方式***

  当然也会有可重用的查找方式。

  注意：这里的查找没有引号，即只表示**“匹配”**，不包括逻辑算法。

以上可重用的部分都希望能独立出来，以便不同的词典类使用，这些部分便成了词典类的“可配置组件”。

目前看来，有些东西是很难抽离的：

逻辑算法

不同的词典类可能会有不同的逻辑算法，得出的结果自然也会不同。

当然开发者也可以完全不依赖以上这些：

采用新的数据结构和加载方式

采用新的切分方式

采用新的查找方式

以及新的逻辑算法

若是有办法直接返回结果，甚至无需以上任何流程，只需要拥有同样的输入及输出方式即可，以下有一个***异想天开的方案***:

有趣的对话词典类

输入是一串字符，也许是“How are you?”。

输出则是对该字符串的逻辑回应，对应以上则可以是“Fine, and you?”。

所以，不必拘泥于任何形式，大可以将其想象为一个神奇的智能类，你指定输入，它便给你最可能想要的结果。

4. 设计

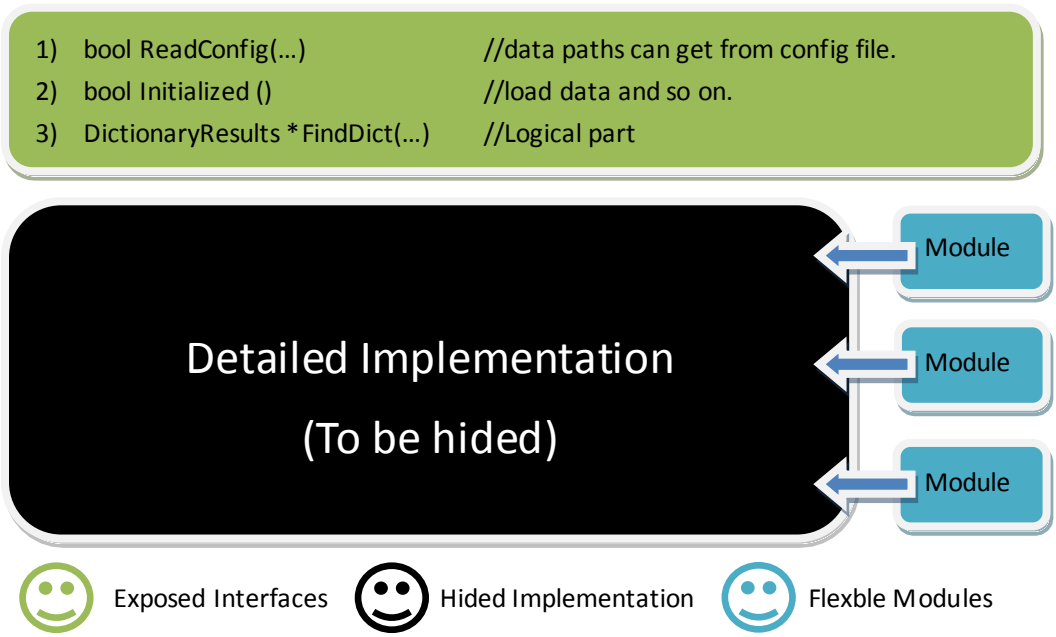

图 **1** XDictionary 结构图

Pseudocode Demo:

Definition:

```
DemoDictionary : BasicDictionary{
    Public:
        Bool ReadConfig(…);
        Bool Initialized(…);
        DictionaryResults *FindDict(…);
};
```

Implementation:

```
Bool DemoDictionary::ReadConfig(…){
    //omitted…
}

Bool DemoDictionary::Initialized(…){
    //omitted…
}

DictionaryResults * DemoDictionary::FindDict(…){
    //do some things
    …
    //maybe load a module
    …
    //some algorithm
```

```
        …
        //sort candidates
        …
    }
```

Usage:

```
BasicDictionary *basic_dic=( BasicDictionary *)new DemoDictionary();
basic_dict-> ReadConfig(…);
basic_dict-> Initialized();
//give an input string, then get results
DictionaryResults *dict_results=basic_dict->FindDict(str);
```

Obvious Advantages of the Design:

1. Interface users don't need to know the implementation of XDictionary.
2. If the internal parts of XDictionary have been changed, the modifier almost not need to notify the interface users.
3. The last is easy to expand.