**0. 词表**

为了在SunPinyin2中支持不同的拼音方案，甚至包括注音、粤拼等不同的系统，我们改变了以前使用拼音字符作为trie树索引key的方式，而使用音节编码作为trie树的key。

我们定义了一个TSyllable的数据结构，来表示一个音节：

struct TSyllable {  
    unsigned other    : 12;  
    unsigned initial  : 8;  
    unsigned final    : 8;  
    unsigned tone     : 4;  
}

这里只显示了big-endian时的结构布局，other是padding用的，initial表示的声母，final表示的是韵母，tone自然就是声调了。

相应地，我们修改了pytrie\_gen.{h|cpp}，主要集中在insertFullPinyinPair, threadNonCompletePinyin, 并增加了parseFullPinyin, combineInitialTrans, addCombinedTransfers, expandCombinedNode等辅助方法；CPinyinTrieMaker类中其他的方法基本没有什么修改。

让我们看看代码：

parseFullPinyin (pinyin, ret):

解析传入的拼音字符串（pinyin），调用CPinyinData的encodeSyllable静态方法，将得到的音节编码（TSyllable），并顺序保存到ret中，最后返回。

CPinyinTrieMaker::insertFullPinyinPair (pinyin, wid):

调用parseFullPinyin，将拼音字符串转换为音节编码（TSyllable）的序列，然后调用insertTransfer将这个音节的分支加入到trie树中，并调用insertWordId更新叶子节点上的word ID列表。

CPinyinTrieMaker::combineInitialTrans (pnode):

遍历trie树节点pnode上所有的自节点（m\_Trans），将具有相同声母（initial）的子节点，收集到一起。然后调用addCombinedTransfers方法，将某一个声母对应的若干子节点，作为一个特殊的节点（尚未扩展的复合节点），加入到原来的trie树中。例如，

(N0)--->shang--->(N1)  
  | \  
  |  +->sheng--->(N2)  
  |  
  +---->sh------>(N3\_{N1,N2})

CPinyinTrieMaker::addCombinedTransfers (pnode, s, nodes):

如果节点集（nodes）的size为1，则直接在pnode上将s的转发分支指向那个节点。举例来说，N0只有一个shang的子路径，但是我们依然希望为N0加入一个sh的转发路径。不过我们不需要创建新的符合节点，直接把这个转发指向N1就好了。

如果集合size>1，就创建一个新的节点（m\_bExpanded缺省是false），其m\_cmbNodes成员赋值为nodes，然后在m\_StateMap中记录一下，已经为这个节点集创建过新节点了（即p），最后把nodes上所有的候选词IDs，merge到新的子节点上。最后，将pnode上的s转发分支指向新的复合节点。

CPinyinTrieMaker::expandCombinedNode (pnode):

遍历pnode的复合节点，整合它们的子转发路径，并创建新的符合节点，插入到trie树中。

例如，扩展N3节点，将N1和N2的diao转发合并，创建一个N7\_{N4,N6}新的复合节点，作为N3节点“diao”的转发路径。因为只有N1有“hai”的转发路径，直接将N3节点上“hai”的转发路径指向N5节点。

(N0)---->shang--->(N1)---diao--->(N4)  
  |\               \  
  | \               +----hai---->(N5)<--------------+  
  |  \                                              ^  
  |   +->sheng--->(N2)---diao--->(N6)               |  
  |                                                 |  
  +----->sh------>(N3\_{N1,N2})                      |  
                   |  \                             |  
                   |   +----diao--->(N7\_{N4,N6})    |  
                   |                                |  
                   +--------------hai-------------->+

CPinyinTrieMaker::threadNonCompletePinyin:

遍历m\_AllNodes，如果该节点的m\_bExpanded为true，则调用combineInitialTrans，对该节点进行声母合并的操作，其间创建的新节点会append到m\_AllNodes。如果m\_bExpanded为false，则是一个复合节点，调用expandCombinedNode，将复合节点的子节点进行合并操作，其间创建的新节点也会append到m\_AllNodes。最后，这棵支持不完整拼音的trie树就构建完成了。

另外，对CPinyinTrie这个类也进行了相应的修改，使其按照TSyllable来进行索引。

**1. 拼音切分**

在原先SunPinyin的ime部分中，拼音切分是借助CPinyinTrie来完成的，同时紧密地耦合在CIMIContext类中（注：CIMIContext类是输入引擎的核心，主要负责根据拼音字符串建立一个search lattice并进行搜索，并处理用户的user selection，以及获取某个位置开始的candidates，参见sunpinyin代码导读（九））。

这为我们添加新的拼音方案（例如双拼）带来了极大的不便，所以在2.0中，我们将拼音切分从CIMIContext中剥离了出来，定义了一个统一的接口——IPySegmentor。要为SunPinyin2添加一个新的拼音方案，就是要实现一个IPySegmentor的接口（当然，如果该方案无法和全拼一一对应，例如粤拼，还需要自行定义自己的lexicon，参见上篇）。

让我们来看看这个IPySegmentor接口：

struct IPySegmentor  
{  
    enum ESegmentType  
        {SYLLABLE, SYLLABLE\_SEP, INVALID, STRING,};  
  
    struct TSegment {  
        std::vector<unsigned>   m\_syllables;  
        unsigned                m\_start        : 16;  
        unsigned                m\_len          : 8;  
        ESegmentType            m\_type         : 8;  
    };  
  
    typedef std::vector<TSegment>  TSegmentVec;  
  
    virtual TSegmentVec& getSegments () = 0;  
    virtual wstring& getInputBuffer () = 0;  
    virtual const char\* getSylSeps () = 0;  
  
    virtual unsigned push (unsigned ch) = 0;  
    virtual unsigned pop () = 0;  
    virtual unsigned insertAt (unsigned idx, unsigned ch) = 0;  
    virtual unsigned deleteAt (unsigned idx, bool backward=true) = 0;  
    virtual unsigned clear (unsigned from=0) = 0;  
  
    virtual unsigned updatedFrom () = 0;  
    virtual void locateSegment (unsigned idx, unsigned &strIdx, unsigned &segIdx) = 0;  
};

拼音切分器的任务是，将用户的输入切分成TSegment(s)，例如用户的输入是“xi’anshiAi”，那么得到的segments是，[(xi), s:0, l:2, SYLLABLE], [(), s:2, l:1, SYLLABLE\_SEP], [(an), s:3, l:2, SYLLABLE], [(shi), s:5, l:3, SYLLABLE], [('A'), s:8, l:1, STRING], [('i'), s:9, l:1, INVALID]。之所以区分INVALID和STRING，是为了preedit可以考虑用不同的颜色来显示它们。

如果切分器支持易混淆音（例如z-zh, c-ch, s-sh），一个segment可能会有多个syllables。例如，对上例中的shi，对应的segment为[(shi, si), s:3, l:3, SYLLABLE]。注意，CIMIContext会将m\_syllables的第二个及之后的音节视为是易混淆音，而在搜索的宽度和句子评分方面有所降低。

细心的读者可能发现了，在上面的例子中，start的信息好像是冗余的。但如果切分器支持模糊切分，例如fangan -> fang’an, fan’gan，那么TSegment的m\_start就有用武之地了。并且隐含要求，TSegmentVec中的segments是按照m\_start从小到大排序的。不过，现有的CIMIContext还需要一些改进，需要根据搜索的最优路径来返回正确的切分路径。

下面我们来看一看IPySegmentor的主要接口。因为用户输入、删除或编辑，是以单个字符为单位进行的，我们的接口也是按照这种风格设计的。例如push，是指用户在尾部添加一个字符，这个方法的返回值是，指示segments从拼音字符串的什么位置发生了更新。这个更新值非常重要，考虑下面的示例，

Input | Segments | UpdatedFrom  
------+----------+------------  
z     | z        | 0  
zh    | zh       | 0  
zho   | zh, o    | 2  
zhon  | zh, o, n | 3  
zhong | zhong    | 0

可以看到，在用户输入g时，原本3个segments被合并成了一个，且返回值为0，表示segments从拼音字符串0这个位置发生了更新。这个返回值，和updatedFrom方法返回的是一致的，只是为了client调用方便而添加的。之所以需要这个updatedFrom值，是出于性能的要求：CIMIContext希望在用户输入/编辑之后，只重新build产生更新部分拼音字符串对应的lattice，而不需要从新build整个lattice。

其他的几个online editing方法，也是类似的。pop指从尾部删除一个字符，insertAt/deleteAt分别表示从中间位置插入/删除一个字符。之所以区分push/insertAt，或pop/deleteAt，是因为push/pop可能有很多优化的空间。locateSegment返回拼音字符串的某个位置（idx），其所在音节的起始位置（strIdx），和segments vector中的下标（segIdx）。getSegments返回一个TSegmentVec的引用，即切分的结果。getInputBuffer返回用户输入的原始数据，getSylSeps返回这个切分器支持的音节切分符。

目前SunPinyin2中实现了两个切分器，分别为CQuanpinSegmentor、CShuangpinSegmentor。其中CQuanpinSegmentor支持易混淆音、自动纠错等功能，且这些功能都是可配置的，但尚不支持模糊切分。我们将在下一篇介绍CQuanpinSegmentor的实现。

**2. 全拼切分器**

如在上篇介绍的那样，SunPinyin原先是借助CPinyinTrie来进行拼音切分的，因为CPinyinTrie是一棵前缀树，因此切分是按最大前向匹配的方式进行的。还有一些额外的处理，例如fangan会被切分为fan’gan，但是dier是被切分为die’r的（我们也因此收到了很多用户的反馈）。在SunPinyin2中，我们依然借助一个trie树来进行音节切分。不过这个tire树是由所有有效音节组成的一棵后缀树。因此，我们的切分原则是以最大后向匹配为基础的。

简单的后向匹配还是有一些问题，例如zhonge（中俄），会被切分成zh’o'n’ge，如果你留意FIT或者ibus-pinyin，就会发现他们都有上述的问题。那这个问题在SunPinyin2中的全拼切分器是如何解决的呢，用户在输入g的当下，切分结果是zh’o'n，加入g之后，segments发生了向前的合并，那么我们就把’g'换成’G'（实际的处理是高位置1）放到内部匹配的buffer中。那么下次用户在输入e的时候，最大后向匹配zhonGe的结果，就会产生正确的结果了。还有一种可能性要处理，例如feng，fe不是一个合法的音节，所以在输入n的当下，切分的结果是f’e，在输入n之后，音节发生了向前的合并，因此N被加入到内部匹配的buffer中，变为feN，但是再输入g时，最大后向匹配却无法返回正确的切分结果了。所以，所以我们还要尝试在某些情况下，将N临时改为n再match一次，如果得到的segment长度是原先的长度+1，就接受这个新的结果，否则再把n改回到N。

我们前面提到了，SunPinyin2使用一棵后缀trie树进行拼音切分的。trie树可以通过逐级二分来实现（如果是内存中的，可以是hash或map）。为了尽可能地保障切分器的效率和性能，我们使用了Double Array Trie（DAT）的方式。DAT的优势在于，可以直接load/mmap到内存中，并且查询子节点的复杂度是O(1)的，缺点是后续的插入和删除比较复杂。因此十分适合作为构造静态词典的方法。在SunPinyin2中，DAT的构造是用python实现的。我们定义了一个C++的模板类CDATrie来访问DAT，其中match\_longest (first, last, length)是最核心的方法。读者有兴趣可以去查看相关的代码和资料，这里就不赘述了。

接下来我们看看CQuanpinSegmentor的实现细节。CQuanpinSegmentor有两个助手类，CGetFuzzySyllablesOp和CGetCorrectionPairOp。这两个助手类分别用来得到易混淆音（例如zhan -> zan/zang/zhang），和纠错结果（例如ign -> ing）。这两个类都是可配置的。并且所有的CQuanpinSegmentor实例，应该共享相同的助手类的实例。这种可配置项，我们称为shared & global的，在用户修改用户配置的时候，不会影响已经创建的那些会话（sessions）。我们后面还会看到，有些配置项是non-shared but global的，对这种配置项的更改，需要重新创建已存的那些会话。这是后话，暂且不表。

这两个助手类比较直观，这里也就不赘述了，读者可以参考相关的代码。

CQuanpinSegmentor的核心是*push方法。目前的实现中，deleteAt/insertAt都是用*push方法来实现的，也就是把修改位置后面的字符串重新push了一遍。这个实在是比较粗笨，应该还有一定的改进空间。不过因为用户输入的拼音字符串长度有限（缺省的设置是512个字符），性能目前还是可以接受的。

*push方法首先将传入的字符ch，push到m\_pystr中，然后调用m\_pytrie的match\_longest (m\_pystr.rbegin(), m\_pystr.rend(), l)方法进行后向匹配，匹配的结果主要有以下几种情况：*

* l == 0，表明ch不是一个有效的音节，根据字符的特征选定一个seg\_type，创建一个长度为1的segment并push到m\_segs中。
* l == 1，可能是一个新的segment，但是如果是上面类似feNg这样的情况，就修改上一个segment的属性。否则，创建一个新的长度为1的segment，并push到m\_segs中。
* l == m\_segs.back().m\_len + 1，也就是说ch可以扩展上一个segment，那么修改上一个segment的属性。
* 其他情况，一种是 l > m\_segs.back().m\_len + 1，例如zhon+g，l = 5，但是上一个segment（n）的长度是1。另一种情况是l < m\_segs.back().m\_len + 1，例如die+r，l=2，但是上一个segment（die）的长度是3。这两种情况都会涉及合并或拆分现有的segments。并且在此过程中，可能会引起进一步的合并和拆分动作。因此我们用一个while循环来追赶，直到达到一个既有的音节边界为止。

最后，所有的分支都会跳到RETURN，进行post-editing的操作，这里我们只有一个post-editing的action，就是处理易混淆音。