# PRTPinyin-自由输入的时代

# 描述：

PRTPinyin输入法是一个提供了快速输入，智能学习与联想等功能为一体

的完整拼音输入法设计。

# 功能：

拼音序列->候选词结果；---输入法核心

字符序列->候选词结果；---联想

拼音，词组->智能学习新词。---智能学习

# 达成：

依赖：C++，脱离Qt库

功能完整性：95%(纠错，组合词)

效率：中文长度为60的搜索（搜索次数达几百至上千）耗时32-64ms。

易用性：组合词定制、词频与新词学习。

移植性：demo完美移植G4

# 数据定制：

1. 普通字典（3个）

描述：针对单个拼音或者汉字。建立拼音或拼音与汉字的连接关系，说到底就是，整个设计的字符处理的基础；

特点：占用小，可直接入内存；

编码：UTF-8（注意忽略掉bom，前三个字节EF BB BF）。

1. PRTPinyinBasic.ly：看它的名字，就知道是拼音字符串处理的基础，使用它来建立声母韵母的映射。

表形式：f:a,u,o,an,ou,ei,en

主要使用点：判断是否是声母(ch、w)、韵母(en、ing)或整体认读(ai、ang)；是否是完整的拼音（chi）；是否是可填充拼音(jion)；以及可能成为的拼音(ji->jin、jing、jiong、jiang…)

1. PRTPinyinSingle.ly：建立一个完整拼音对应的汉字映射关系。

表形式：a:啊阿呵吖嗄腌锕

主要使用点：获取拼音对应的汉字；获取汉字的拼音（可能是多音字）

1. PRTPinyinDYZ.ly：建立多音字与拼音的映射关系。

表形式：曝:bao,pu

主要使用点：是否是多音字；获取多音字对应的拼音；获取多音字默认的拼音。

三个字典都是针对单个的拼音或者汉字。是字符处理的基础。

1. 词库

心酸的历程… …

关于词库的制作，这里涉及到几个点：

1. 词库的容器是什么？也就是我们选择什么作为数据的载体？

这个我在2012年下半年设计输入法的时候就开始想过这个问题。那时候竟然想过用文本来存储，现在想起来好天真。既然这样，那很多人应该也想到了数据库了。其实HP曾经跟我说过自己设计数据存储结构，但我想想，自己几斤几两，何能设计出能和现今几大数据库匹敌的数据结构来。

既然决定使用数据库，接下来就是数据库的选择问题了。

说说几个现在流行的数据库的特点以及我们在选择数据库时应该注意的事项。

现在流行的数据库大概有几个：Oracle、db2、Sybase、sql server 、mysql、sqlite 、access。

1. 关于数据量：大数据Oracle、db2、Sybase；中数据sql server 、mysql、sqlite；小数据access…
2. 关于跨平台：可跨平台的数据库Oracle、db2、Sybase、mysql、sqlite；

sql server 、access不支持跨平台。

1. 关于效率：如果是百万级以上的数据量，大型数据库比如Oracle、db2肯定是占有优势，但是在中小型的数据量相差不是很多。
2. 关于价格：Oracle、db2、Sybase、sql server、access都是要付费的；mysql、sqlite开源免费！
3. 文件型数据库？oracle（可服务器可文件）、sqlite、access
4. 综合考虑的结果：针对词库总结了需求，数据量中（大概几十万，即使多个数据库联合也就100W左右）；跨平台（需要）；效率（查询效率高，写效率不做要求）；价格（不必花钱）；文件型数据库（使用方便）。

鉴于以上，选择sqlite是比较机智的（我说的是明智）。

1. 数据来源是什么，也就是我们去哪里拿数据来填充数据库？

目前制作的系统词库词条数17.7W多，其中二字词组6.5W左右，三字词组3.3W左右，四字词组6.9W左右，余下的1W多为五字到十四字的词组。词条涵盖普通词语、常用语、成语、古诗词、歇后语等。这些词语花费了我大概半年的时间，包括收集、筛选、以及纠错。每个过程都是很耗时且枯燥的，基本都要人工审核，而且要写一堆程序去协助做这些事情。这里我想说的是，枯燥且无趣！但是一定要做。

1. 我需要拿数据库来做什么？并且我怎么样设计它来更好的为我的整体设计服务？

好了，数据库选择好了，又有了数据，我们需要设计数据库的结构。要考虑的也就两点：表结构以及每个表的列结构。数据库的设计经历了以下三个阶段：

刚开始是只有一个表的，表内2个字段分别为pinyin，cizu，也就是所有的词条都放在一个表里面。后来发现搜索效率不算高，因为在搜索的时候会使用到like关键字，这就是说要检测表内的所有数据，效率肯定低下。

第二阶段发展成23\*23=529个表，也就是ac-zz（中间的i，u，v组合忽略），然后依然2个字段pinyin，cizu。

后来发现还有可优化的空间，于是发展成了现在的结构。23\*23=529个表，4个字段size，pinyin，cizu，frequency，添加size是为了某些查询时候的判断，frequency是为了排序。pinyin，cizu字段建立索引并且设定唯一性。

三个阶段的结构变化如下：

|  |  |
| --- | --- |
| pinyin | cizu |
| de’sai’xi’wei | 德赛西威 |

↓

|  |  |  |  |
| --- | --- | --- | --- |
| size | pinyin | cizu | frequency |
| 4 | de’sai’xi’wei | 德赛西威 | 100 |

1. 词库的维护

我们需要一个针对此结构的词库提供一个管理工具，来管理数据的插入、删除、修改以及数据导入和备份功能。另外，也可以用来制作一个用户级词库。

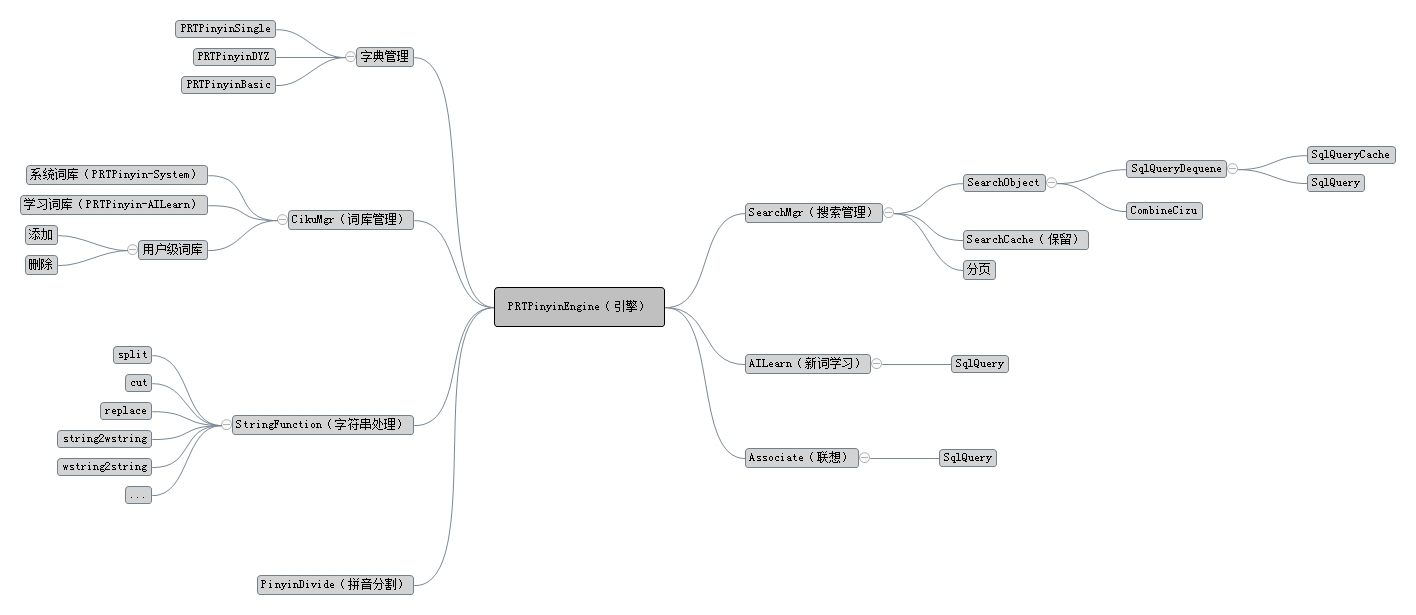
（展示词库管理工具）

# 结构设计

* **流程图**



**思维导图**



几个设计细节：

* 初始化：三个普通字典文件都是utf8编码，并且放入内存中map进行查询。

词库管理类管理所有词库，包括系统词库、学习词库和用户级词库，并允许外部添加或删除用户级的词库。

* 拼音字串合法性检测：26小写字母和分隔符’组成的序列

1、只包含小写字母和分隔符（可以不含分隔符）；

2、以字母开头；

3、不允许两个连续的分隔符

如：wo'ai'ni；woaini；wo'aini等合法的字串都是合法的拼音

如：以下是非法的，如：'ni'de；wo\_\_ni+df'de；

* 拼音分割：

\*主要依赖于PinyinBasic来作为分割的依据

\*分割后用词库数据库模糊查询的输入

\*输出为一个或多个分割结果（如果有特殊点的话）

\*由于拼音分割使用频繁，所以算法需要注意，另外复制开销尽量避免，尽量减少stl容器的使用，特别是复杂容器（频繁new 和delete的开销还是很大的）。

\*分割步骤：①初步分割->②去掉可前后匹配的特殊n、g点，

同时替换只可后匹配的特殊n、g点->③可前后匹配

的特殊n、g点排列成多种组合可能->得到的组合与原来的②结果拼起来。

上面讲到了特殊点（它的判定条件有一定的算法，也就是不是所有的n或g都是特殊点），概念---n或者g，他们在拼音中的位置有三种情况：

只可前匹配比如jionga的g只能和jion匹配所以成了jiong’a；

只可后匹配比如mani的n只能和i匹配所以成了ma’ni；

即可前匹配又可以后匹配比如ganga的g，所以成了gan’ga，gang’a。

一个例子：wojuedehenganga

初步分割---wo’jue’de’hen’gang’a

存在特殊点g(倒数第二个字符，可前后匹配)

最后输出wo’jue’de’hen’gang’a; wo’jue’de’hen’gan’ga

* 搜索并确立候选词：

全部候选词的结构是：组合词+搜索队列+第一个拼音的汉字序列。

假如输入为wo’zai’zhe’li’ni’zai’na’li。

组合词：以可能的最长长度的拼音串搜索的结果组合，比如最先搜索wo’zai’zhe’li’ni’zai’na’li，无结果后搜索wo’zai’zhe’li’ni’zai’na，以此类推，直到有结果后比如wo’zai’zhe’li搜到结果，剩下的ni’zai’na’li按同样方式进行下去，直到组合完成。

建立搜索队列：wo’zai’zhe’li’ni’zai’na’li

wo’zai’zhe’li’ni’zai’na

…

…

wo’zai

也就是搜索长度为2到n-2的拼音序列。这些都是sql查询并排序后得到的候选词。

查询语句使用sql标准语法，使用union联合各数据库进行查询。保证一个拼音串只需一次查询语句就搞定，提高效率。

第一个拼音的汉字序列：wo的汉字序列结果。

以上，所有的候选结果就出来了。

* 缓存机制：

建立sql对象缓存机制的必要性在于，按照输入的习惯，我们的输入字符都是在原有的字符上递增的，那么之前搜索过的很多字符都要重新进行搜索。建立缓存机制，如果是之前搜索过的字符串，直接就可以拿取结果，而不是建立搜索对象并重新搜索。效率大大提高。

比如，我们搜索一个词组长度为N拼音串，那么最终需要进行的搜索次数为：(n-1)+（1到(n-1 + 1) \* (n-1)/2）。那么如果n=60,搜索次数就会是60次到1700多次。如果使用缓存机制，那么执行那么多次需要的时间大概是500ms，使用缓存机制后效率提高了10倍，大概消耗为32-64ms，已经是可接受的范围了。

* 编码（程序员永远的痛）

设计的全部字符源都使用utf-8，在涉及到汉字处理的部分先转到unicode宽字符，然后在转回utf-8。这样做的原因是sqlite内部使用的是utf-8的编码，而主要工作都是在与数据库打交道，所以我们的数据源都规定为utf-8，最后的候选字输出也同一为utf-8。

注意：编码转换最好不要依赖于系统api！在 linux下，x86版本可以使mbstowcs或icon函数族是c的标准，但是linux arm下使用会有问题，我估计是因为这两种方式都依赖于系统的local（本地语言编码），linux arm上很可能只有C，而没有zh\_utf8等中文的支持。最后我用针对每个字节的转换的方式来实现宽字符的转换。

# 接口使用与定义：

virtual bool SetSourceFilePath(const std::string &sPinyinBasic, const std::string &sPinyinDYZ,

const std::string &sPinyinSingle, const std::string &sDBSystem, const std::string &sDBAILearn) = 0;

说明：设置字典资源的路径（不检查路径的合法性）

参数：分别为PinyinBasic.ly、PinyinSingle.ly、PinyinDYZ.ly、PRTPinyin-system.db和PRTPinyin-AILearn.db的路径。

返回值：true

virtual bool InitPinyinEngine() = 0;

说明：初始化拼音引擎，根据SetSourceFilePath路径加载字典资源。

参数：

返回值：成功返回true，失败返回false

virtual bool ReleasePinyinEngine() = 0;

说明：释放拼音引擎

参数：

返回值：成功返回true，失败返回false

virtual bool AddUserCikuDataBase(const std::string &sPath) = 0;

说明：添加一个外部词库

参数：词库路径

返回值：成功返回true，失败返回false。

virtual bool RemoteUserCikuDataBase(const std::string &sPath) = 0;

说明：移除一个外部词库

参数：词库路径

返回值：成功返回true，失败返回false。

virtual bool RemoteAllUserCikuDataBase() = 0;

说明：移除所有添加的外部词库

参数：

返回值：成功返回true，失败返回false。

virtual bool GetAllUserCikuDataBase(std::list<std::string> &listCiku) = 0;

说明：获取所有添加词库的路径

参数：返回结果的list

返回值：true

virtual bool SetCandidatePerPageCount(int nPerPageCount) = 0;

说明：设置每页候选词的的个数

参数：

返回值：成功返回true，失败返回false。

virtual bool GetCandidatePerPageCount(int &nPerPageCount) = 0;

说明：获取每页候选词个数  
参数：引用

返回值：true

virtual int GetCandidatePerPageCount() = 0;

说明：GetCandidatePerPageCount重载

参数：

返回值：结果。

virtual bool SearchPinyin(const std::string &sInput) = 0;

说明：搜索  
参数：sInput乱序拼音

返回值：成功返回true，失败返回false

virtual bool GetSearchingPinyin(std::string &sSearchingPinyin) = 0;

说明：获取搜索的乱序拼音

参数：引用

返回值：返回true

virtual std::string GetSearchingPinyin() = 0;

说明：sSearchingPinyin重载

参数：

返回值：结果

virtual bool GetCandidatePageCount(int &nPageCount) = 0;

说明：获取搜索到的候选词页数  
参数：引用

返回值：true

virtual int GetCandidatePageCount() = 0;

说明：GetCandidatePageCount重载

参数：  
返回值：结果

virtual bool GetPageCandidate(int nPageIndex, std::vector<std::string> &vtPageCandidate) = 0;

说明：获取指定页码的候选词

参数：nPageIndex页码下标，vtPageCandidate结果容器

返回值：成功返回true，失败返回false

virtual bool GetCandidateCount(int &nCandidateCount) = 0;

说明：获取搜索结果的候选词个数。  
参数：引用  
返回值：true

virtual int GetCandidateCount() = 0;

说明：GetCandidateCount重载

参数：

返回值：结果

virtual bool GetCandidate(int nIndex, int nCount, std::vector<std::string> &vtCandidate) = 0;

说明：获取范围内的候选词  
参数：nIndex起始下标，nCount个数，vtCandidate结果容器  
返回值：成功返回true，失败返回false

virtual bool GetAllCandidate(std::vector<std::string> &vtAllCandidate) = 0;

说明：获取所有搜索结果候选词

参数：vtAllCandidate结果容器  
返回值：成功返回true，失败返回false

virtual CandidateInfo PickOneCandidate(int nPage, int nIndex) = 0;

说明：选取一个候选词  
参数：nPage页码，nIndex当页下标  
返回值：选择候选词的信息

virtual bool PickOneCandidate(int nPage, int nIndex, CandidateInfo &PickCandidateInfo) = 0;

说明：重载  
参数：nPage页码，nIndex当页下标，PickCandidateInfo结果候选词信息  
返回值：成功返回true，失败返回false

virtual CandidateInfo PickOneCandidate(int nIndex) = 0;

说明：重载  
参数：下标  
返回值：选择候选词的信息

virtual bool PickOneCandidate(int nIndex, CandidateInfo &PickCandidateInfo) = 0;

说明：重载  
参数：nIndex下标，PickCandidateInfo结果候选词信息

返回值：成功返回true，失败返回false

virtual bool GetDividedPinyinString(std::string &sPinyinDivided) = 0;

说明：获取正在搜索的拼音的分割字符串  
参数：sPinyinDivided结果  
返回值：成功返回true，失败返回false

virtual std::string GetDividedPinyinString() = 0;

说明：重载  
参数：

返回值：结果

virtual bool LearnWord(const std::string &sPinyin, const std::string &sCizu) = 0;

说明：学习一个新词  
参数：sPinyin拼音，sCizu词组

返回值：成功返回true，失败返回false

virtual bool ForgetWord(const std::string &sPinyin, const std::string &sCizu) = 0;

说明：忘记一个词

参数：sPinyin拼音，sCizu词组

返回值：成功返回true，失败返回false

# 总结

涉及到的点：结构设计、编码处理、sql使用等。