## 一.开发环境

Linux: centos 7.0 64bite
G++: version 4.8

## 二.系统目录结构

include :存放系统的头文件
bin: 存放系统的可执行文件
conf: 系统的配置文件
src： 系统的源码目录
data:系统的词典，磁盘缓存，系统日志

## 三.系统执行过程图


## 四.系统结构图

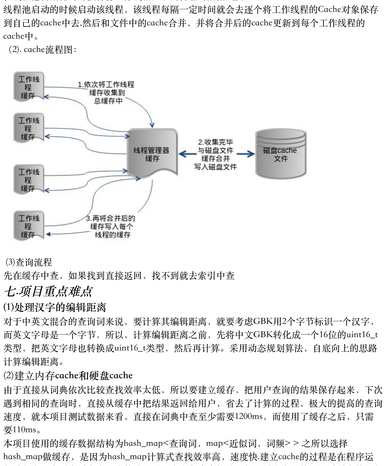



## 五.相关类与其说明

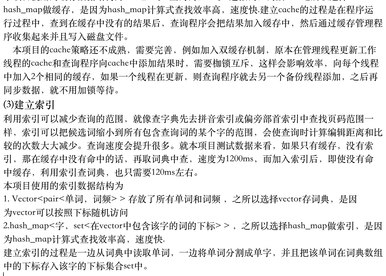
1)前期准备，自己生成英文词库和中文词库，由于中文词库大多是GBK的编码，因此需要转码gbktoutf.cpp就是用来转码，使用cut.py来利用结巴分词对中文语料库进行分词，最后使用make_dic_hash_ch.cpp进行中文分词，make_dic_hash_en.cpp进行英文分词
2)class CMyconf 用来读取配置文件，将用户词典，磁盘缓存文件，日志文件的目录传入系统
3)struct Edit距离编辑算法，主要用来计算编辑距离的
4)struct CMyhash主要用来声明一个hash_map，来重命名两个hash_map 模版，strhash用来建立倒排索引，hash_record用来建立磁盘缓存文件
5)class CCond,CMutex,主要用来同步线程池访问独占资源任务队列
6)class CTask用来表示具体执行的任务，CPtask用来表示具体执行任务的指针,class CQueue用来队列中生产任务和消费任务
7)class CThread用来启动线程，同时将查询记录缓存和日志类指针传入线程池
8)struct Log_system用来记录系统日志，将日志信息写入系统日志文件中
9)class CThread_pool用来初始化线程池，并启动线程池
10)class Query_handle生产查询任务，并将查询任务放入线程池队列中执行
11)class CWord_correct 是整个系统最核心的类，用来接收查询词，将查询词分词，建立查询词的倒排索引.返回查询结果，同时建立查询记录缓存
12)class My_socket类用来将查询结果和查询客户端的套接字一起打包传给具体执行查询任务的线程，然后线程才得以将查询结果发送给客户端
13)class My_epoll用来建立服务器查询系统，采用epoll模式设计，非阻塞监听套接字，通过My_epoll来建立和启动整个服务器查询系统
14)class My_client客户端测试类，用来测试服务器系统的性能和准确率

## 六.主要功能说明

1. 主线程：
(1).初始化配置类，日志类，词典、线程池、索引、读取cache文件到线程的cache
(2).读取配置并启动udp服务器。
(3).循环监听并接受客户端发来的查询。
(4).只要一有请求，就根据请求初始化一个任务对象task，同时线程池把该任务放到任务队列中去 。
(5).当线程处理完后请求，直接根据task中保存的客户端信息，把结果发给客户端
2.主线程流程图




3 .工作线程工作流程
(1)初始化自己的内存cache对象。
(2)循环不停地去任务队列里取任务 。
(3) 执行任务。
调用成员函数excute（）获得查询结果。
在excute（）中，调用Query类程序先检查客户端发来的请求是否在cache中，如果在则直接从cache中把结果返回。如果不在cache中，根据先前建立的索引遍历存放词典的vector容器。在遍历的过程中逐次比较编辑距离并将结果存储到优先级队列。当遍历完map后将最优结果返回给客户端并将结果存储到cache中去 。
4.客户端查询任务流程图




5.cahce管理线程的工作流程
（1).说明：
在线程池初始化的时候将各工作线程关联到自己的数据成员。
线程池启动的时候启动该线程，该线程每隔一定时间就会去遂个将工作线程的Cache对象保存到自己的cache中去.然后和文件中的cache合并，并将合并后的cache更新到每个工作线程的cache中。
(2). cache流程图：




(3)查询流程
先在缓存中查，如果找到直接返回，找不到就去索引中查

## 七.项目难点难点

(1)处理汉字的编辑距离
对于中英文混合的查询词来说，要计算其编辑距离，就要考虑GBK用2个字节标识一个汉字，而英文字母是一个字节，所以，计算编辑距离之前，先将中文GBK转化成一个16位的uint16_t类型，把英文字母也转换成uint16_t类型，然后再计算。采用动态规划算法，自底向上的思路计算编辑距离。
(2)建立内存cache和硬盘cache
由于直接从词典依次比较查找效率太低，所以要建立缓存，把用户查询的结果保存起来，下次遇到相同的查询时，直接从缓存中把结果返回给用户，省去了计算的过程，极大的提高的查询速度，就本项目测试数据来看，直接在词典中查至少需要1200ms，而使用了缓存之后，只需要110ms。
本项目使用的缓存数据结构为hash_map<查询词，map<近似词，词频> > 之所以选择hash_map做缓存，是因为hash_map计算式查找效率高，速度快.建立cache的过程是在程序运行过程中，查询在缓存中没有的结果后，查询程序会把结果加入缓存中，然后通过缓存管理程序收集起来并且写入磁盘文件。
本项目的cache策略还不成熟，需要完善，例如加入双缓存机制，原本在管理线程更新工作线程的cache和查询程序向cache中添加结果时，需要细锁互斥，这样会影响效率，向每个线程中加入2个相同的缓存，如果一个线程在更新，则查询程序就去另一个备份线程添加，之后再同步数据，就不用加锁等待。
(3)建立索引
利用索引可以减少查询的范围，就像查字典先去拼音索引或偏旁部首索引中查找页码范围一样，索引可以把候选词缩小到所有包含查询词的某个字的范围，会使查询时计算编辑距离和比较的次数大大减少。查询速度会提升很多。就本项目测试数据来看。如果只有缓存，没有索引，那在缓存中没有命中的话，再取词典中查，速度为1200ms，而加入索引后，即使没有命中缓存，利用索引查词典，也只需要120ms左右。
本项目使用的索引数据结构为
1. Vector<pair<单词，词频> > 存放了所有单词和词频 ，之所以选择vector存词典，是因为vector可以按照下标随机访问
2.hash_map<字，set<在vector中包含该字的词的下标> > ，之所以选择hash_map做索引，是因为hash_map计算式查找效率高，速度快.
建立索引的过程是一边从词典中读取单词，一边将单词分割成单字，并且把该单词在词典数组中的下标存入该字的下标集合set中。
|