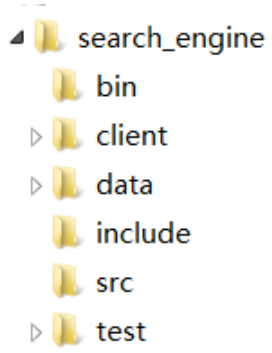
搜索引擎文档说明书

1. 开发环境：

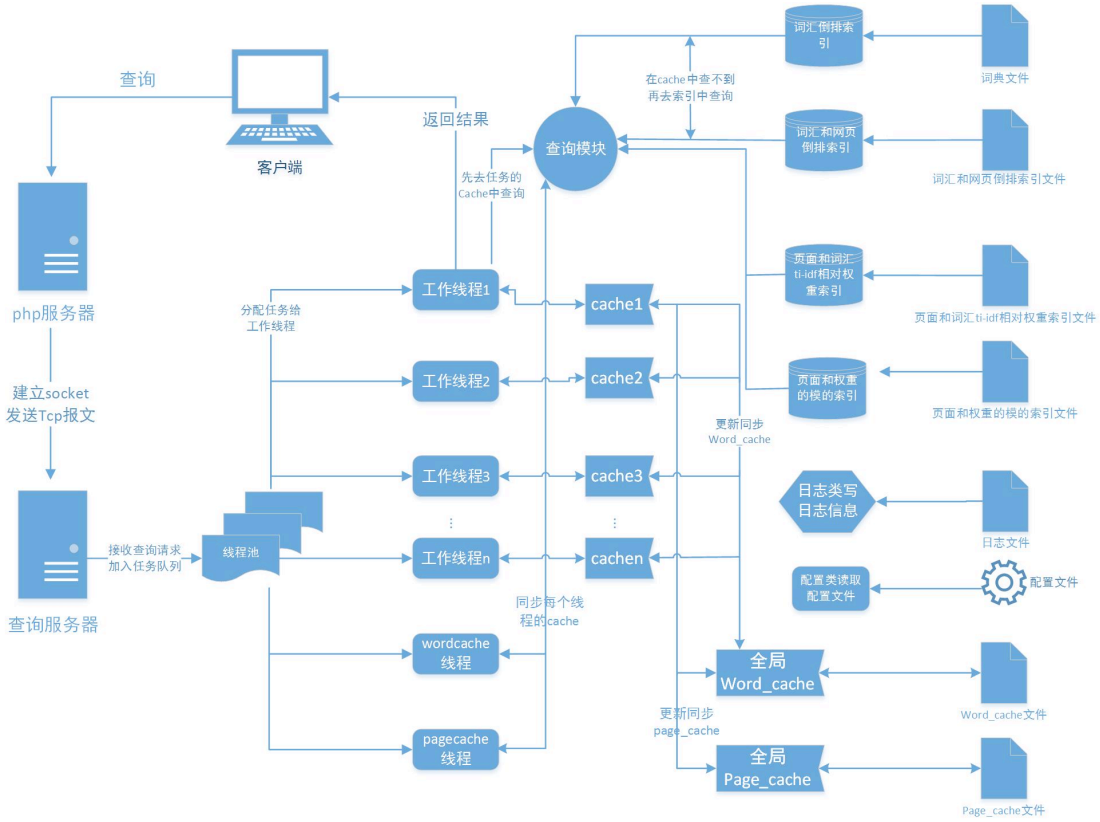
Linux: centos 7.0
G++: version 4.8

2. 系统目录结构



src ：存放系统的源文件（.cpp） 。
include：存放系统的头文件（.h） 。
bin: 存放系统的可执行程序
client : 客户端程序
data/my.conf：存放系统程序中所需的相关配置信息
data/page_lib： 存放系统程序中所使用的库文件
data/dic.t： 存放系统程序所需的字典
data/cache.dat 文本纠错缓存
data/page_cache.dat 网页查询缓存
data/content_index 网页内容索引
data/word_page_rindex 查询网页结果倒排索引
data/page_wright_rindex 页面查询词权重倒排索引
data/cppjieba 结巴分词库

3. 系统运行过程图



4. 相关类及其说明

Class CCond(条件变量类)

数据成员：

CMutex& m_mutex	词典
pthread_cond_t m_cond	本类实例

函数成员：

void wait()	等待
void notify()	唤醒
void notifyall()	唤醒全部

Class CMutex（单例模式，互斥变量类）

数据成员：

pthread_mutex_t m_mutex;	互斥变量

函数成员：

void lock()	锁函数
void unlock()&cache);	解锁函数

Class CTask(抽象任务类)

主要函数成员：

virtual void excute(hash_record* cache,hash_record*	任务执行函数

m_page_cache,hash_record* title_cache,LOG_SYSTEM::Log_system* log)=0;	
bool consume(CPtask* task)	消费任务
bool produce(CPtask task);	生产任务

Struct Log_system(日志类)

数据成员：

log4cpp::PatternLayout layout;	日志布局
log4cpp::Appender* file_app;	日志输出路径
log4cpp::Category& my_cat;	日志对象

函数成员：

void* handle(void* arg);	日志处理
void init();	日志系统初始化

Class CMyconf(配置文件)

数据成员：

map<string,string> config	配置文件路径信息

主要的函数成员：

CMyconf(const std::string& conf_path);	构造函数

Class Query_handle(多线程处理文本纠错)

数据成员：

CWord_correct m_correct;	文本纠错类
CThread_pool m_pool;	线程池
CRun m_run	执行任务线程类
Cache m_cache_run	执行文本纠错结果的缓存线程类
Page_cache m_page_cache_run	执行网页摘要标题缓存类
Page_rank* m_rank	执行搜索的核心类

函数成员：

Query_handle(CMyconf& conf,Log_system* log)	构造函数
void operator()(const int& fd_client)	将任务放入线程池

Class CWord_correct(处理文本纠错的核心类)

数据成员：

const char* dic_path	字典的路径
hash_map<string,set<pair<string,int>> r_index	字典的倒排索引

函数成员：

void make_index(strhash& r_index,const char* dic_path)	建立字典的倒排索引
void make_cache(word_cache& cache,strhash& r_index, const sting& search_word)	建立查询词的倒排索引
void get_correct(word_cache& cache,vector<CMypair>& re_vec, const string& search_word);	获取文本纠错结果
void operator()(const string& search_word,string& search_result);	文本纠错处理函数

Struct Edit(距离编辑算法)

函数成员

int operator()(const string& src,const string& dest)	函数对象计算编辑距离
int triple_min(int a,int b,int c)	取编辑距离最小值
int length(const string& str)	计算词汇的长度

Class Page_rank(单例模式)(搜索引擎的核心类)

主要数据成员：

CMyconf& conf	配置文件对象
hash_tf query_tf_idf	查询词的 tf-idf 权重
priority_queue<Weight>	权重优先级队列
page_weight_rindex　p_w_rindex	页面 ID，单词，tf-idf 权重索引
word_page_rindex word_tf_idf	单词，页面 ID，tf-idf 权重倒排索引
MixSegment seg	结巴分词对象
hash_page_index page_index	页面索引

主要函数成员：

void make_word_page_rindex()	建立词汇文档权重倒排索引
void make_page_weight_rindex()	建立文档词汇权重倒排索引
void make_page_index()	建立文档标题摘要内容索引
void make_page_module_index()	建立文档词汇模的索引

Class CThread(线程类)

数据成员

pthread_t m_tid;	线程 ID
hash_record m_cache;	词汇纠错缓存
hash_record m_page_cache;	网页查询结果缓存
hash_record m_title_cache;	网页标题内容缓存
LOG_SYSTEM::Log_system* m_log;	日志系统内容指针

函数成员

void start(CThread_RUN* arg);	创建并且启动线程
hash_record* get_record();	获取线程的词汇纠错缓存
hash_record* get_page_cache();	获取线程的网页标题摘要缓存
hash_record* get_title_cache();	获取线程的网页摘要内容缓存
void set_log(LOG_SYSTEM::Log_system* log)	设置线程的日志类指针

Class　CThread_RUN(抽象类)

主要函数成员：

virtual void run()=0;	执行任务
virtual void set_cache(hash_record* m_cache)=0;	设置词汇纠错缓存
virtual void set_page_cache(hash_record* m_page_cache)=0;	设置网页标题摘要缓存
virtual void set_title_cache(hash_record* m_title_cache)=0;	设置网页标题内容缓存
virtual void set_log(LOG_SYSTEM::Log_system* log)=0;	设置日志系统类指针

Class CRun(CThread_RUN 派生类,工作线程类)

数据成员：

CQueue* m_pq	任务队列指针
hash_record* m_cache	词汇纠错结果缓存
hash_record* m_page_cache	页面标题摘要缓存
hash_record* m_title_cache	标题内容缓存
Log_system* m_log	日志系统类指针

函数成员：

void run()	执行工作线程
void set_log(Log_system* log)	设置线程日志系统指针
void set_cache(hash_record* cache)	设置词汇纠错结果缓存
void set_page_cache(hash_record* page_cache)	设置页面摘要缓存
void set_title_cache(hash_record* titlecache)	设置页面标题内容缓存

Class Page_cache(CThread_RUN 派生类,page_cache 缓存管理类)

数据成员：

CMyconf& m_conf	配置文件类的引用
hash_record* m_page_cache	页面标题摘要缓存
hash_record* m_title_cache	页面标题内容缓存
vector<CThread>& m_thread_pool	线程池向量的引用
Log_system* m_log	日志系统类指针

函数成员：

void run()	执行工作线程
void set_log(Log_system* log)	设置线程日志系统指针
void set_cache(hash_record* cache)	设置词汇纠错结果缓存
void set_page_cache(hash_record* page_cache)	设置页面摘要缓存
void set_title_cache(hash_record* titlecache)	设置页面标题内容缓存
void read_disk_page_cache()	读取磁盘网页查询结果缓存
void write_disk_page_cache()	将同步后的缓存重新写入磁盘

Class Cache(CThread_RUN 派生类，cache 缓存管理类)

数据成员：

CMyconf& m_conf	配置文件类的引用
hash_record* m_cache	词汇纠错结果缓存
vector<CThread>& m_thread_pool	线程池向量的引用
Log_system* m_log	日志系统类指针

函数成员：

void run()	执行工作线程
void set_log(Log_system* log)	设置线程日志系统指针
void set_cache(hash_record* cache)	设置词汇纠错结果缓存

void set_page_cache(hash_record* page_cache)	设置页面摘要缓存
void set_title_cache(hash_record* titlecache)	设置页面标题内容缓存
void read_disk_cache()	读取磁盘词汇纠错缓存
void write_disk_cache()	将同步后的缓存重新写入磁盘

Class CTask_excute（用来存储任务信息的）

数据成员：

CWord_correct& m_correct	词汇纠错类的对象
Page_rank* m_rank	处理网页查询
int& m_fd_client	客户端的文件描述符
char message[MAXSIZE]	客户端的查询信息

成员函数

int word_correct(hash_record* m_cache, string word_result,Log_system* m_log, string m_search)	处理客户端的查询词的纠错，并返回纠错结果
void handle_title_content(hash_record* m_cache,Los_system* m_log)	返回对应查询标题的内容
int hand_title(hash_record* m_cache, string word_result	用于保存用户端地址和端口号
void excute(hash_record* m_cache, hash_record* m_page_cache, hash_record*m_title_cache, Log_system* m_log)	处理客户端具体的查询
bool send_msg(const char* msg,Log_system* m_log)	发送消息
bool recv_msg(Log_system* m_log)	接收消息

Class CThread_pool(线程池)

数据成员

vector<THREAD::CThread> m_thread_pool	包含工作线程的数组容器
THREAD::CThread cache_thread	查询词纠错缓存线程
THREAD::CThread page_cache_thread	页面结果缓存线程
bool flag	线程池开关

函数成员：

ThreadPool(CMyconf& conf,Log_system* log)	构造函数（参数是配置类对象和日志类指针）
void on(CThread_RUN* thread_run,CThread_RUN* thread_cache,CThread_RUN* thread_page_cache)	启动线程池
vector<CThread> get_thread_vector()	获取线程池的引用

Class My_epoll(服务器类)

数据成员：

CMyconf　m_conf	配置文件类对象
int server_sock	服务器端 socket 描述符
struct sockaddr_in server_addr	服务器端 socket 地址
string m_ip	服务器端 IP 地址
int m_port	服务器端 port
Log_system m_log	日志系统类对象

函数成员：

My_epoll(string conf_path,string port);	构造函数
void socket_init()	初始化服务器端套接字
void handle_query()	处理查询
void epoll_add(int epollfd,int sockfd,int state)	将描述符放入监听集
void socket_error(string message,int value)	处理错误
void set_nonblock(int& sockfd)	将 socket 描述符号设为非阻塞

5. 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，page_cache 文件到线程的 cache，page_cache。

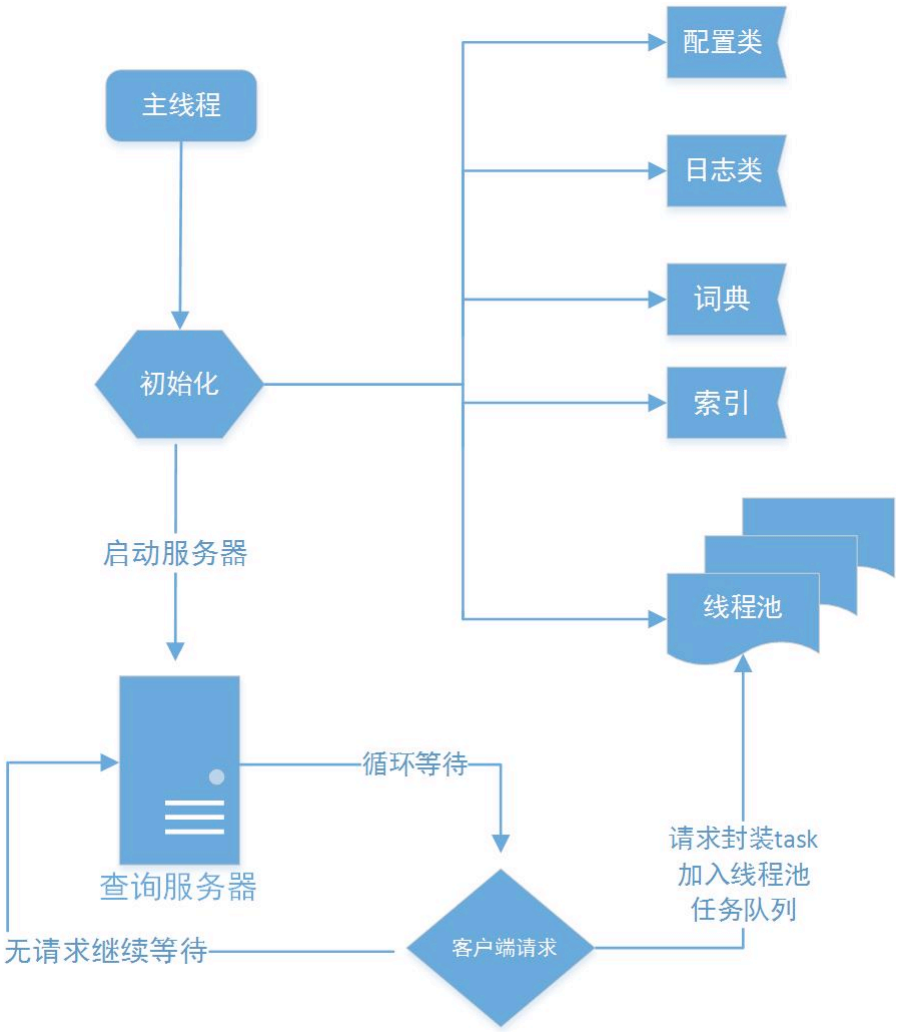
(2).读取配置并启动 TCP 服务器。

(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 socket 文件描述符，把结果发给客户端

2. 主线程的流程图如下：



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 。

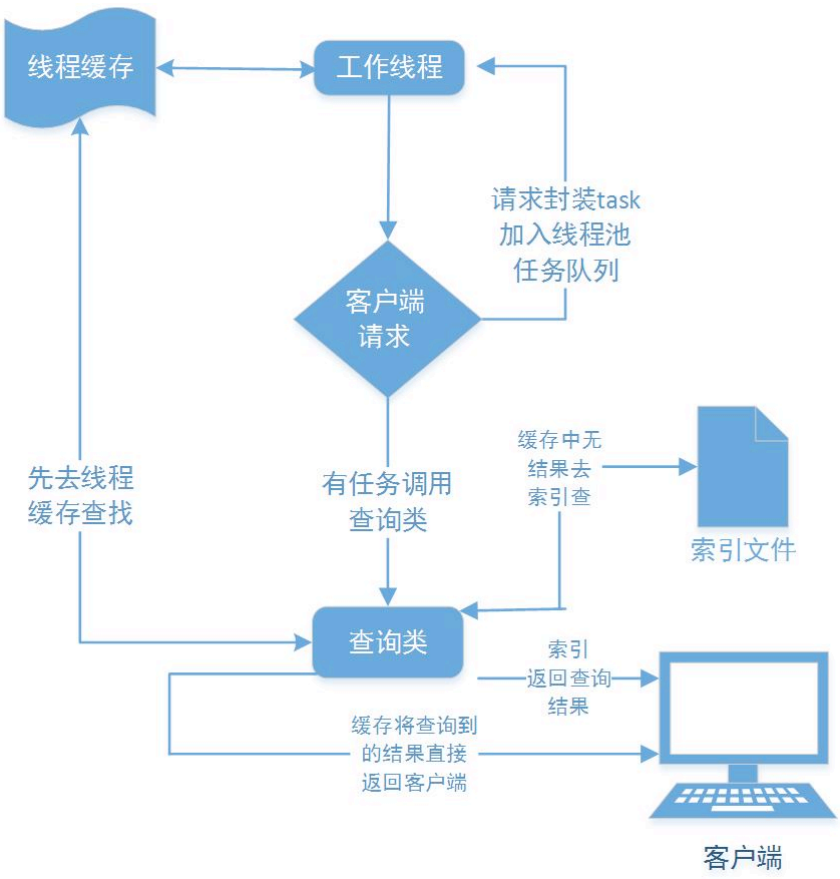
(3)执行任务。

(4)调用成员函数 excute()获得查询结果。在 excute()中，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 chache 中，根据先前建立的索引遍历存放词典的 hash_map 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 。

(5)如果文本纠错结果确认客户端发送的消息无误，就先检查该查询词对应的标题摘要结果是否在 page_cache 中，再就直接返回结果，如果没有的话，就调用 page_rank()的算法来返回结果，并且将返回的 title_cache,page_cache 插入两个缓存中

(6)根据用户的标题请求查询 title_cache，如果有直接返回结果，如果没有就通过遍历完 hash_map<string,string> page_content 获取结果，并且将查询标题和对应结果插入缓存 m_title_cache.

(7)流程图



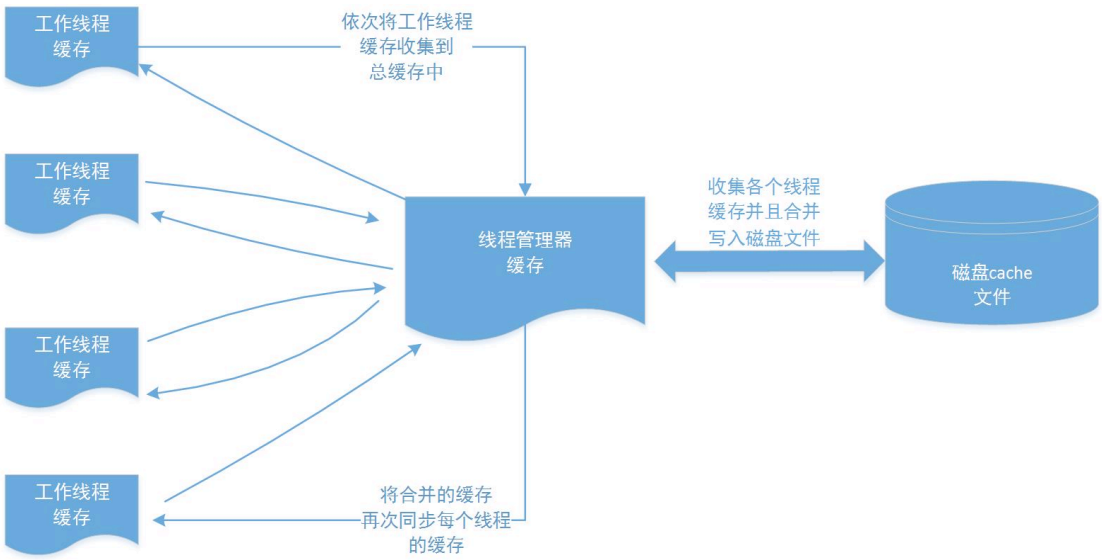
4.page_cache,cache 管理线程的工作流程

(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逐个将工作线程的 Cache,Page_cache 对象保存到自己的 cache 和 page_cache 中去,然后和文件中的 cache,page_cache 合并，并将合并后的 cache,page_cahe 更新到每个工作线程的 cache,page_cache 中。

(2)流程图：



5.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

6. 项目难点重点

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 UTF-8 用 2,3,4 个字节三种方式标识一个汉字，而英文字母是一个字节，所以，计算编辑距离之前，先将中文 UTF-8 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。（详情看代码/include/dis.h）

建立内存 cache，page_cache 和硬盘 cache,page_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.

map hash_map　效果对比表

	map	hash_map
文本纠错	100ms	90ms
网页查询	90ms	80ms

使用的网页查询结果缓存也是 hash_map<查询词,页面标题摘要>,hash_map<标题，页面内容>两种数据结构，页面标题摘要为 JSON::Value 建立 cache 的过程是在程序运行过程中，查到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。格式的字符串，通过反向解析就可以获得相应的结果。

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，需要枷锁互斥，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。

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。而网页查询处理的话，如果缓存没中的话，利用网页索引 hash_map<int,pair<int,int>>在网页库中查询的话也只需要 1500ms, 加入网页索引 hash_map<int,pair<int,int>>的后，只要 120ms,

本项目使用的索引数据结构列表

hash_map<string,map<int,double>	查询词，网页 ID，tf-idf 权重倒排索引
hash_map<int,map<string,double>	网页 ID，查询词，tf-idf 权重倒排索引
hash_map<int,pair<int,int>>	网页 ID,在网页库中的偏移量,网页内容长度索引
hash_map<int,double>	网页 ID，网页词汇的权重的模索引
hash_map<string,set<pair<string,int>>>	单字，包含单字的单词，词频的倒排索引
hash_map<int,pair<string,string>>	网页 ID，标题，摘要索引

使用索引和缓存机制处理效率对比表

	缓存机制	索引	平均执行时间
文本纠错	×	×	1500ms
文本纠错	√	×	1200ms
文本纠错	×	√	120ms
文本纠错	√	√	90ms
网页查询	×	×	1800ms
网页查询	√	×	1100ms
网页查询	×	√	100ms
网页查询	√	√	80ms

建立网页库

由于从语料库中获取的文本数据都是 gbk 编码的，而 linux 系统是使用 UTF-8 编码的，所以会引起乱码问题，需要编写转码函数来建立 UTF-8 编码的语料库，然后就是要进行网页去重，因为不同预料库中的标题内容不同，但是正文内容可能是相似的，本项目采用冒泡排序来进行网页去重。具体可参见 make_lib 目录下 Trim_redunt 类中的去重函数。

实现网页搜索查询

(1)首先建立如下的数据索引

hash_map<string,map<int,double>	查询词，网页 ID，tf-idf 权重倒排索引
hash_map<int,map<string,double>	网页 ID，查询词，tf-idf 权重倒排索引
hash_map<int,pair<int,int>>	网页 ID,在网页库中的偏移量,网页内容长度索引
hash_map<int,double>	网页 ID，网页词汇的权重的模索引
hash_map<string,set<pair<string,int>>>	单字，包含单字的单词，词频的倒排索引
hash_map<int,pair<string,string>>	网页 ID，标题，摘要索引

(2)确认查询词纠错结果后通过 get_query_tf_idf()方法获取查询词的 tf_idf 权重,然后通过 get_query_module()获取查询词的 tf_idf 权重的模.

(3)获取包含所有查询关键词的网页的查询词，网页 ID，tf-idf 权重倒排索引的集合，然后通过计算每个网页与查询之间的文档相似度获取相应的排序结果，然后将结果放入优先级队列，最后按照相似度高低进行排序，从高到低输出结果。