SpellCorrect 项目文档

王道第三期 一组 王维

一、简介

本程序是在 Linux 下使用 C++编写的一个单词纠错程序(我称之为 miniSearch)。这个程序需要达到的效果是：

客户端输入一个单词，服务器接收后在词库中寻找与之最相近的一个词并输出，并返回客户端。

二、运行演示（效果）

开启服务器

```
➜  0504_miniSearch  ./bin/runServer.sh start
start!
```

开启客户端，输入单词，并接收显示服务器返回的单词

```
➜  0504_client  ./client
192.168.1.13 8888
你好
你好
哈喽
哈
你妹
你
哟西
东西
iphone
phone
```

三、程序目录结构

```
➜  0504_miniSearch  ls
bin  conf  data  include  lib  log  Makefile  src
```

bin	//存放可执行文件和执行脚本
conf	//配置文件，包括词库文件路径、磁盘 cache 路径、服务器地址等
data	//存放词库文件和 cache 文件
include	//存放头文件
lib	//（暂无）
log	//存放系统日志文件
src	//存放 cpp 代码文件
Makefile	//Makefile

遍历程序主目录得到如下文件和目录信息：

```
→ 0507_dir_scan  ./dir_scan ../0504_miniSearch
Directory scan of ../0504_miniSearch
include/
    EditDistance.h
    CacheThread.h
    Condition.h
    ThreadPool.h
    Thread.h
    WorkThread.h
    Socket.h
    MutexLock.h
Makefile
lib/
src/
    MutexLock.cpp
    CacheThread.cpp
    Condition.cpp
    Socket.cpp
    server_main.cpp
    WorkThread.cpp
    ThreadPool.cpp
    EditDistance.cpp
    Thread.cpp
conf/
    config.txt
log/
    log.txt
data/
    cache.dat
    word_frequence_en.dat
    word_frequence_ch.dat
    frequence.dat
bin/
    cp.sh
    runServer.sh
    miniSearchServer
done.
```

四、程序用到的类

Socket

封装了服务器进行 UDP 通信的所有函数，主函数直接调用该类来获取客户端地址及其发来的信息。

MutexLock

封装了一个互斥锁，在需要互斥访问某资源时使用这个对象来对资源进行加锁和解锁。

Condition

封装了一个条件变量，初始化该条件变量时需要一个 MutexLock 对象的参数。用于同线程间的同步。

Thread

线程抽象类，封装了线程的创建、销毁等过程。成员函数中有一个纯虚函数:

```
virtual void run() = 0;
```

其子类继承 Thread 类后只需要实现 run()函数即可。

WorkThread

工作线程，继承自 Thread。成员函数 run()中实现了所有处理过程。类中有一个指向 ThreadPool 的指针成员变量，并有一个注册到 ThreadPool 的成员函数调用它，以此来实现工作线程与线程池的关联。

该类中有一个 cache 缓存，其类型为

std::unordered_map<std::string, std::string> (在 c++11 的 std 命名空间中)

存储了输入的查询记录单词，和输出的在词库中计算出的最匹配单词。

ThreadPool

线程池类，控制 cache 线程和工作线程的运行，终止，获取任务等行为，以及运行时的同步和互斥关系。该类中也一个 cache 缓存。

CacheThread

cache 控制线程，该类中有一个 cache 缓存。用于在服务器启动时同步磁盘 cache 和线程池 cache，并发出信号让线程池 cache 去同步工作线程 cache。然后进行循环计时，并在每轮计时结束时再进行新一轮同步。

EditDistance

该类用于计算编辑距离。在初始化该类的一个对象时，会从词库文件中读取单词和词频，存入索引中，索引采用如下数据类型

std::unordered_map<uint32_t, std::map<std::string, int> >

key 值是一个 32 位整型数，这个数表示词库中每个单词中的一个字（英文字母或中文字）。value 值是一个 map，存储了包含 key 关键字的所有单词(std::string)及其词频(int)。

由于 unodered_map 支持下标访问，使得我们在计算编辑距离时能很快的确定比较范围，极大缩短计算时间。

将与词库的比较结果放在优先级队列中，并按照编辑距离最小（编辑距离相等时取词频最高）的优先级顺序存放，并输出优先级最高的一个单词。

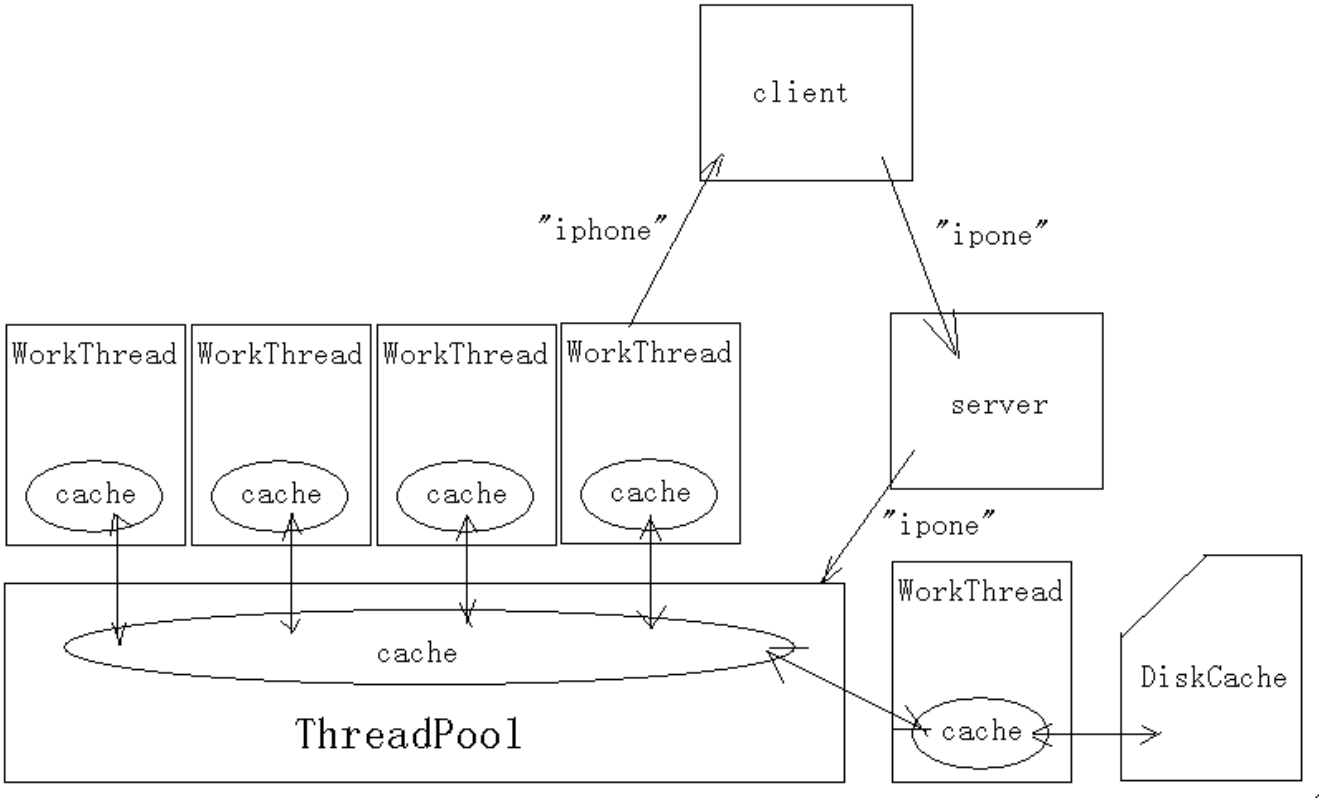
五、程序结构及处理过程

服务器接收到客户端发来的查询单词后，将单词放入线程池的任务队列中，工作线程取出任务后在词库中进行编辑距离的计算，然后得到一个与输入词编辑距离最短的单词，直接通过 UDP 通信发回给客户端。

在计算编辑距离的算法中，为兼容 UTF8 中文字符，将单词字符串转换为 32 位整型数数组，再对两个数组进行比较计算编辑距离。为了提高编辑距离计算的效率，采用了 unordered_map 的数据结构对词库建立索引。

为进一步提高程序响应速度，采用了内存 cache 和磁盘 cache 来提高对已搜索单词的处理速度。在我设计的 cache 系统中，每个 WorkThread 有一个内存 cache，另有一个注册到线程池的 cache 控制线程，每 100 秒通过线程池指针调用线程池同步函数，进行线程池和控制线程的 cache 同步，然后发出一个同步信号，让每个 WorkThread 去和线程池同步 cache。

处理流程如下图所示：



六、遇到的问题及解决方法

1. 问题描述：

在编辑距离的计算中，需要加入对中文的支持。最初采用的是对中文 GBK 编码的解析和计算，而客户端输入的中文是 UTF8 编码，词库的单词却是 GBK 编码，这样在计算编辑距离时就必须进行 GBK 和 UTF8 编码的转换，但是我在实际操作中发现，少量数据的编码转换处理正常，但是对词库进行大量的编码转换比较编辑距离时每次都会出现段错误，最后调试 core 发现是编码转换的函数中间出了问题。

解决办法：

直接采用 UTF8 的词库，并在编辑距离的计算函数中针对 UTF8 中文编码进行解析，所有的中文都采用 UTF8 编码进行处理，这样统一了文字编码，避开了编码转换的问题。

2. 问题描述：

在工作线程调用 EditDistance 类时，初始化 EditDistance 的对象写在 WorkThread 线程类的 run() 函数的 while 循环中，这样就导致了工作线程每次 while 循环取任务时都初始化了一个临时的 EditDistance 对象，并且进行了读词库文件并建立索引的过程，这就导致了系统 IO 开销过大，并且可能导致搜词时间消耗特别久。

解决办法：

将初始化 EditDistance 的对象的代码写在 while 循环之前，然后将该对象的引用作为形参传入 handle_task(Task &task, EditDistance &ed)中，这样就保证了只在线程开启阶段进行

一 次读词库建立索引，解决了系统开销过大和搜词耗时过长的问题。

产生了新的问题：

搜索过程中，搜索一个单词，有时会输出上一次的搜索结果。

解决办法：

调试发现，保存搜索结果的优先级队列是 EditDistance 类中的一个成员变量，当一直调用这同一个对象进行编辑距离的计算时，优先级队列里存放的搜索结果没有被清空，也就是仍然保留了上一次搜索的结果。在 EditDistance 类中，计算编辑距离返回输出结果之前将优先级队列清空，问题得到解决。

3. 问题描述：

关于同步 cache，我采用的方法是，建立一个新的线程来计时，计时结束时与线程池内的 cache 同步，然后线程池将同步开关打开，由于不知道线程同步什么时候结束，此时设置了等待几秒后关闭同步开关。在同步开关打开期间线程池与每个工作线程进行同步 cache。问题在与同步开关打开期间工作线程是阻塞的，只能去进行同步操作，而不能取任务。

解决办法：

在线程池中进行已同步线程的计数，每当一个工作线程与线程池同步结束后，将计数器+1，当计数器到最大值（工作线程数量）时，将计数器置零，此时关闭同步开关。这样就解决了同步时阻塞时间过长的问题。

七、log 日志、配置文件及 cache 测试

log 日志：

在通过 shell 脚本运行服务器时，直接将输出流重定向至 log.txt，代码如下

```
./bin/miniSearchServer >> ./log/log.txt
```

这样在代码中所有的 cout 打印信息就全部输入 log 日志了

配置文件：

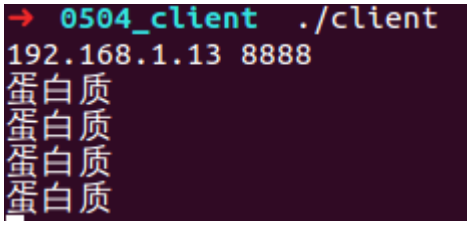
./conf/config.txt

```
address: 192.168.1.13 8888     //服务器地址
word_lib: ./data/frequence.dat       //选用的词典文件
cache: ./data/cache.dat            //磁盘 cache 文件
```

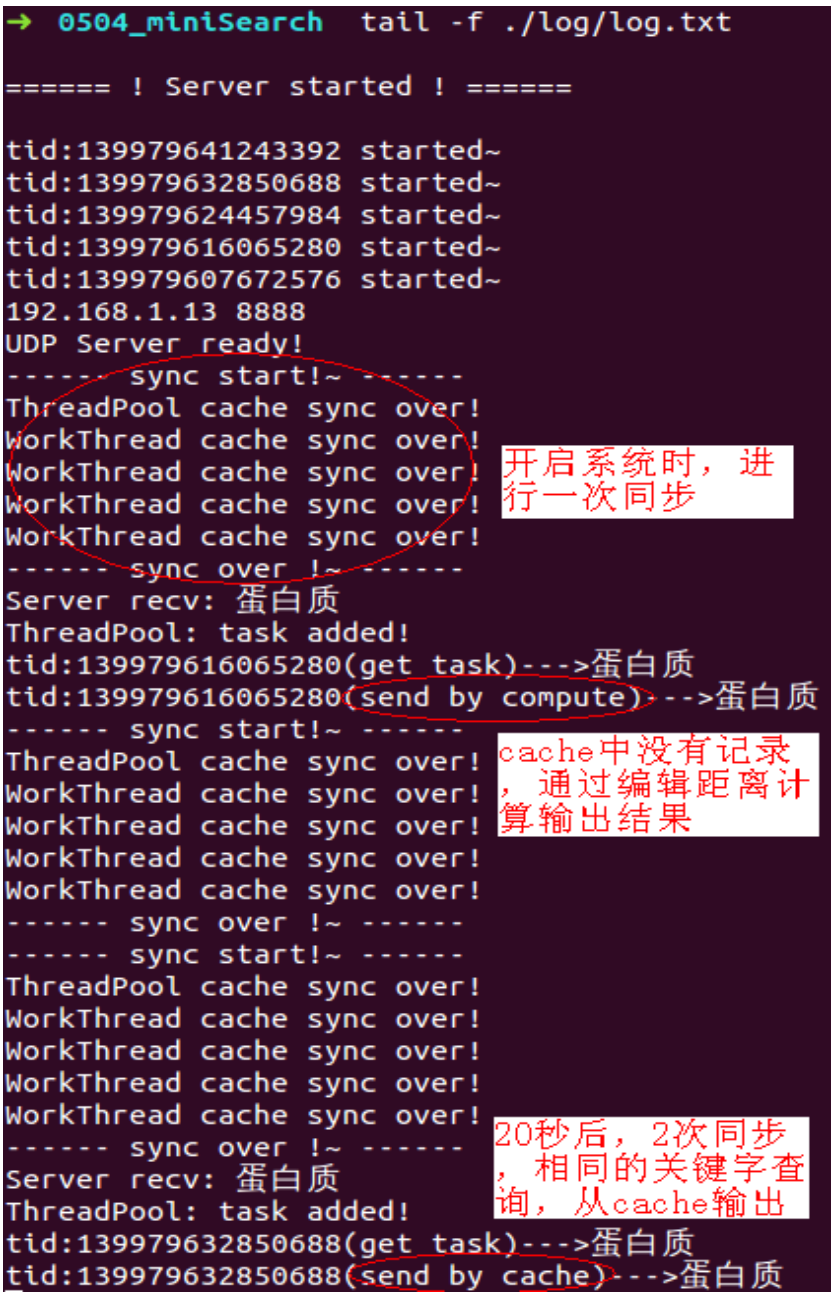
所有的配置都在配置文件中，方便修改，不需要重新编译服务器。

cache 测试：

为方便测试，将服务器 cache 更新周期设置为 10 秒，期间客户端两次输入同样的信息：



查看系统日志 log.txt 显示如下信息：



第二次返回的结果显示，是通过 cache 输出，证明 cache 起作用了。两次处理任务的线程号不相同，证明 10 秒一次的 cache 同步成功。

八、如何测试运行本程序

命令行切换到程序主目录下：make　　（需要支持 c++11）
更改 conf/config.txt 中的 ip 和 port 为你的网络地址
更改./bin/runServer.sh 第 5 行为程序主目录的绝对路径
运行./bin/runServer.sh start 开启服务器
运行./bin/runServer.sh stop 关闭服务器