@Dubyoo 2014-06-13 19:26 字数 12071 阅读 143

2014-05-14 项目二：miniSearch项目文档

项目文档

说明

1. 开发环境：Linux、 c++、 PHP、 TCP、 多进程
2. 程序框架：客户端通过PHP页面用TCP协议发送搜索词到服务器，服务器接收后交予子进程处理，子进程进行切词，计算权重并与网页库进行相似度计算，逆序输出相似度最高的N篇文章。

第一阶段：搭建框架 0514

1.目录结构

```
bin         //存放可执行文件和执行脚本
conf        //配置文件，包括服务器地址，网页库、网页库索引、单词索引、停用词表的路径等
data        //存放网页库、网页库索引、单词索引、停用词表
include     //存放头文件
log         //存放系统日志文件
src         //存放 cpp 代码文件
Makefile    //Makefile
```

2.将服务器设置为守护进程

```
static void Daemon() {
    const int MAXFD = 64;
    if (fork() != 0)     //父进程退出
        exit(0);
    setsid();         //成为新进程组组长和新会话领导，脱离控制终端
    //chdir("/");     //此时目录不能改变，因为代码内文件路径是相对路径
    umask(0);         //重设文件访问权限掩码
    for (int i = 0; i < MAXFD; i++) //尽可能关闭所有从父进程继承来的文件
            {
        if (i == 1) //关闭所有文件描述符，除了1，以便通过脚本运行时将输出流重定向至./log/log.txt
                {
            continue;
        }
        close(i);
    }
}
```

3.封装tcp类

4.搭建多进程框架

每当检测到客户端连接时，利用 `fork()` 生成一个子进程，让子进程去处理查询任务，并将查询结果发回客户端。

这里涉及到子进程的回收问题。若利用父进程的 `waitpid()` 回收，有可能造成父进程阻塞，不能及时处理新的访问，或者有可能产生僵尸进程，造成资源浪费。解决方法是：在 `fork()` 函数前设置 `signal(SIGCHLD, wait_child);` 或 `signal(SIGCHLD, SIG_IGN);`， 让系统接管子进程的回收。

```
static void wait_child(int pid) {
    if ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        cout << "pid: " << pid << " exit!" << endl;
    }
}
int main(int argc, char **argv) {

    Daemon(); //守护进程

    std::ifstream ifs("./conf/config.txt");
    std::string line, conf, ip, port;
    while (getline(ifs, line)) {
        istringstream iss(line);
        iss >> conf;
        if (conf == "address:") {
            iss >> ip >> port;
            break;
        }
    }
#ifndef DEBUG   //输出ip和端口
    std::cout << ip << " " << port << std::endl;
#endif

    signal(SIGCHLD, wait_child); //通过wait_child(int)函数来等待回收子进程退出后回收子进程的资源(也可使用下面这句语句回收子进程资源)

    /* 或单独使用下面的函数 */
    //signal(SIGCHLD, SIG_IGN); //父进程忽略子进程退出时传回的信号，让子进程结束的信号由系统接收，由1号进程负责回收子进程的资源

    Task task(50,10);   //Task(最大显示的篇数，每篇显示的行数)
    TcpSocket mysocket(ip, port);
#ifndef DEBUG   //打印标志信息
    cout << "=== ready to recv message ===" << endl;
#endif
```

```
        while (int client_fd = mysocket.accept_connection()) {
            //当检测到新的连接时，创建子进程来处理任务
            struct sockaddr_in client_addr = mysocket.get_client_addr();
            int pid = fork();
            if (pid == 0) {
                while (true) {

                    //recv
                    std::string recv_buf;
                    mysocket.recv_message(client_fd, recv_buf);
                    cout << "recv:" << recv_buf << endl;
                    if (recv_buf == "exit") {
                        cout << "client " << "ip: "
                                << inet_ntoa(client_addr.sin_addr) << " "
                                << ntohs(client_addr.sin_port) << " exit" << endl;
                        close(client_fd);
                        exit(0);
                    }

                    //send
                    std::string send_buf;
                    send_buf = task.search(std::string(recv_buf));
                    mysocket.send_message(client_fd, send_buf);
                }
            } else {
                //close(client_fd);
                //waitpid(pid, NULL, WNOHANG);  //signal()之后不需要wait()
            }
        }
        return 0;
    }
```

第二阶段：网页库构建 0515

遍历目录读库文件，拼接成标准格式，然后写入文件，并同时建立库索引

```
    /*
     * 网页库文件格式：
     * <doc><docid>1</docid>
     * <url>http://baidu.com/</url>
     * <title>标题</title>
     * <content>标题 + 内容</content></doc>
     */
    doc = "<doc><docid>" + string(id)  + "</docid><url>" +
        string(entry->d_name) + "</url>" + "<title>" + title +
        "</title><content>" + content  + "</content></doc>\n";
    lib_vec.pushback(doc);  //每篇doc存入vector
```

```
    /*
     * 写入网页库文件的同时，
     * 记录每篇doc的文件偏移量，和该篇doc的长度，
     * 写入库索引文件
     */
    for(vector<string>::iterator iter = lib_vec.begin(); iter != lib_vec.end(); ++iter)
    {
        ofs_index << i << " " << ofs_lib.tellp();
            //向index写入索引 docid，start_pos
        ofs_lib << *iter;
            //向lib写入doc内容
        ofs_index << " " << (*iter).size() << endl;
            //向index写入索引 size
        i++;
    }
```

编码的转换

由于语料库是gbk编码，在Ubuntu Terminal显示是乱码，并且由于网页输入是utf8编码，而编码转换效率较低而且容易出现段错误，在构建网页库之前，为便于测试，将所有语料库批量装换为utf8编码。转换时，会出现类似"^M"的乱码，这是由于windows下的回车是"\r\n"，在Ubuntu下显示的问题。
必须去除该字符，因为者会造成某些语句被下一行语句覆盖！！！
因为\r是回到行首，而\n才是换行。

第三阶段：网页去重 0516

遍历网页库文件，分别取出每一篇doc，切词，去停用词后统计每篇doc的词频，放入优先级队列。从优先级队列中取出k个词频最高的单词，存入

```
    map<int, map<string, int> > doc_feature;
     //docid,      word,  freq
```

它可以代表每篇doc的特征。
然后开始去重：

```
    //去重
    int * arr = new int [doc_feature.size()];    //构造一个数组
    for(int i = 0; i != (int)doc_feature.size(); ++i)
    {
        arr[i] = i+1;    //给数组赋初始值
    }
    int ix = 0, iy = 0;
```

```
    map<int, map<string, int> >::iterator iter_end = doc_feature.end();
    iter_end--;
    for(map<int, map<string, int> >::iterator iter_x = doc_feature.begin(); iter_x != iter_end; ++iter_x) //待比较文章x
    {
        if(arr[ix] == 0) //当前文章x已经被去除，跳过
        {
            iter_x ++;
            ix ++;
            continue;
        }
        map<int, map<string, int> >::iterator iter_y = iter_x;
        iter_y++;
        iy = ix; iy++;
        for(; iter_y != doc_feature.end(); ++iter_y) //待比较文章y
        {
            if(arr[iy] == 0) //当前文章y已经被去除，跳过
            {
                iter_y ++;
                iy ++;
                continue;
            }
            if(compare_two_doc(iter_x->second, iter_y->second) >= 6) //重复
            {
    #ifndef NDEBUG  //有重复时输出显示发生重复的两个docid和去除的docid
                cout << "repeat! " << iter_x->first << " " << iter_y->first << "  trim " << arr[iy] << "  total(" << doc_feature.size
    #endif
                arr[iy] = 0;    //直接去除docid大的一篇
            }
            iy++;
        }
        ix++;
    }

    //将去重数组写入文件
    string outfile = argv[1] + string("_dup_arr.dat");
    std::ofstream dup_arr_file(outfile.c_str());
    for(int i = 0; i != (int)doc_feature.size(); ++i)
    {
        dup_arr_file << arr[i] << " ";
    }
    dup_arr_file.close();
    delete [] arr;
```

比较判断两篇 doc 是否重复的 top k 策略：

```
    /*
     * 比较两篇doc的词频最高的10个单词
     * 返回相似度
     * 即相同单词的个数
     *
     * 其他方法：
     *     将一个map插入另一个map
     *     每插入一个元素时
     *         如果相同则插入后size不变
     *         如果不同则size +1
     */
    int compare_two_doc(const map<string, int> &mp1, const map<string, int> &mp2)
    {
        int count_dup = 0;
        for(map<string, int>::const_iterator iter1 = mp1.begin(); iter1 != mp1.end(); ++iter1)
        {
            for(map<string, int>::const_iterator iter2 = mp2.begin(); iter2 != mp2.end(); ++iter2)
            {
                if(iter1->first == iter2->first)
                    count_dup++;
            }
        }
        return count_dup;
    }
```

根据以上去重方法得到的去重数组、和原来的网页库、网页库索引，建立新的网页库和网页库索引。重做新网页库的docid。

第四阶段：建索引 0517

目标：

```
    //单词  docid   归一化之后的权重
    word1    docid   normalized_power    docid   normalized_power
    word2   docid   normalized_power    ...
    ...
```

第一步，根据公式求权重：

$$power_{word} = tf_{doc} * log(\frac{N}{df_{word}})$$

```
    power = tf * log(N/df);
    //power 该词在某一篇doc中的权重
    //tf    该词在某一篇doc中的词频
    //df    该词的文档频率
    //N     文档总数
```

```
/*
 * 计算权重
 */
void compute_power(map<string, map<int, int> > &word_docid_freq, map<string, map<int, double> > &word_power)
{
    cout << "!!!!!!compute_power!!!!!!" << endl;
    for(map<string, map<int, int> >::iterator iter = word_docid_freq.begin(); iter != word_docid_freq.end(); ++iter)
        //遍历所有单词
    {
        string word = iter->first;
#ifndef DEBUG   //测试输出单词的权值
        //cout << word << ": " << endl;
#endif
        for(map<int, int>::iterator it = word_docid_freq[word].begin(); it != word_docid_freq[word].end(); ++it)
            //遍历含有该单词的docid
        {
            int docid = it->first;
            int freq = it->second;
            word_power[word][docid] = freq*log(0.05 + (double)word_docid_freq.size()/(double)word_docid_freq[word].size());

#ifndef DEBUG   //测试输出单词的权值
            cout << "docid: " << docid << " power: " << word_power[word][docid] << endl;
#endif
        }
    }
    cout << "compute over!!!" << endl;
}
```

第二步，根据公式，把权重归一化：

$$normalized\_power_{w1} = \frac{power_{w1}}{\sqrt{\sum_{i=1}^{n}(power_{wi})^2}}$$

```
//例如对于文档 d1,可以得到一个所有单词的权重序列
W1, w2, w3 ..... wn
//然后需要归一化,即
w1 = w1 / sqrt(pow(w1) + pow(w2) + ..... pow(wn))
```

```
/*
 * 权值归一化
 */
void get_normalized_power(map<string, map<int, double> > &word_power, map<int, map<string, int> > &doc_feature, map<string, map<in
{
    cout << "!!!!compute normalized_power!!!!" << endl;
    map<int, double> power_2;   //docid权重的平方和开根号
    cout << "!!!compute power^2!!!" << endl;
    for(map<int, map<string, int> >::iterator iter_doc = doc_feature.begin(); iter_doc != doc_feature.end(); ++iter_doc)
        //遍历docid
    {
        int docid = iter_doc->first;
        cout << "docid: " << docid << endl;
        double base = 0;
        for(map<string, int>::iterator iter_word = iter_doc->second.begin(); iter_word != iter_doc->second.end(); ++iter_word)
            //计算分母 sqrt(pow(w1) + pow(w2) + ... + pow(wn))
        {
            double power = word_power[iter_word->first][docid];
            base += (power*power);
        }
        power_2[docid] = sqrt(base);
    }
    cout << "compute sqrt(power^2) over" << endl;

    cout << "!!!!compute normalized_power!!!!" << endl;
    for(map<int, map<string, int> >::iterator iter_doc = doc_feature.begin(); iter_doc != doc_feature.end(); ++iter_doc)
        //遍历docid
    {
        int docid = iter_doc->first;
        for(map<string, int>::iterator iter_word = iter_doc->second.begin(); iter_word != iter_doc->second.end(); ++iter_word)
        {
            normalized_power[iter_word->first][docid] = word_power[iter_word->first][docid]/power_2[docid];
                //取出分子，与分母相除，计算归一化权重
        }
        cout << "compute over!!!" << endl;
    }
```

第三步，单词索引写入文件：

```
ofstream ofs_word_index("words.index");
    //写单词索引文件
for(map<string, map<int, double> >::iterator iter = normalized_power.begin(); iter != normalized_power.end(); ++iter)
{
    ofs_word_index << iter->first << " ";
    for(map<int, double>::iterator it = iter->second.begin(); it != iter->second.end(); ++it)
    {
        ofs_word_index << it->first  << " " << it->second << " ";
    }
    ofs_word_index << endl;
}
```

第五阶段：计算文本相似度 0518

根据公式，计算两篇doc之间的相似度：

$$sim(doc_1, doc_2) = \sum_{i \in doc_1 \cap doc_2} (n\_power\_doc1_{wi} * n\_power\_doc2_{wi})$$

```
//计算 doc_x, doc_y 之间的相似度
//两篇doc之间的公共单词是 word_1, word_2 ... word_n
sim(doc_x, doc_y) = (wx_1*wy_1 + wx_2*wy_2 + ... + wx_n*wy_n);
```

第六阶段：查询模块 0518

对输入的查询query进行切词（去停用词），计算出同时包含所有查询词的所有docid：

```
/*
 * 求出同时包含不同搜索词的docid,
 * 放入 set<int> common_docid 中
 */
static void get_common_docid(map<string, map<int, double> > &m_word_index, vector<string> &words, set<int> &common_docid)
{
    if(words.size() >= 1)
    {
        for(map<int, double>::iterator iter = m_word_index[words[0]].begin(); iter != m_word_index[words[0]].end(); ++iter)
        {
            common_docid.insert(iter->first);
                //将包含第一个单词的docid全部输入到set<int>中
        }
    }
    if(words.size() > 1)
    {
        for(vector<string>::size_type ix = 1; ix != words.size(); ++ix)
            //从第二个单词开始遍历，查找同时含有搜索关键词的docid
        {
            for(set<int>::iterator iter = common_docid.begin(); iter != common_docid.end(); )
                //遍历set<int>查找该docid是否存包含当前搜索词
                //这里涉及到遍历set并用erase删除元素
                //***需要判断边界！！！（重要）***//
            {
                set<int>::iterator it_back = iter;  //备份迭代器
                bool is_begin = false;
                if(it_back == common_docid.begin())
                {
                    is_begin = true;
                }
                else
                {
                    it_back --; //备份迭代器
                }

                if(!m_word_index[words[ix]].count(*iter))
                    //set<int>的docid不包含当前搜索词
                {
                    //cout << "not common " << *iter << endl;
                    common_docid.erase(iter);
                        //删除元素（docid）
                    if(is_begin)
                        //如果删除的是begin元素，重置迭代器
                    {
                        iter = common_docid.begin();
                    }
                    else
                    {
                        iter = ++ it_back;
                        //删除元素后重新设置迭代器
                    }
                }
                else
                {
                    iter++;
                }
            }
        }
    }
#ifndef DEBUG   //测试输出包含搜索词的docid
    for(set<int>::iterator iter = common_docid.begin(); iter != common_docid.end(); ++iter)
    {
        cout << "common_docid: " << *iter << endl;;
    }
#endif
}
```

然后将query当做一篇doc，计算单词的权重，以及归一化权重。
根据公式，计算两篇doc（query和doc）之间的相似度：

```
/*
 * 计算相似度
 * 求doc与搜索关键字的相似度
 * 结果存入优先级队列
 */
void compute_similarity(set<int> &common_docid, map<string, double> &search_word_normalized_power, map<string, map<int, double> >
{
        //求相似度
        for(set<int>::iterator iter = common_docid.begin(); iter != common_docid.end(); ++iter)
        {
            int docid = *iter;
            double similarity = 0;
```

```
                for(map<string, double>::iterator it = search_word_normalized_power.begin(); it != search_word_normalized_power.end();
                {
                    similarity += it->second * word_index[it->first][docid];
                }
                Similarity sim;
                sim._docid = docid;
                sim._similarity = similarity;
                q.push(sim);
            }
    }
```

从计算得到的docid取出整篇doc，分别取出标题和内容:

```
    std::string Task::get_title(const std::string &doc) //取标题
    {
        int start = doc.find("<title>") + 7;
        int end = doc.find("</title>");
        string title(doc, start, end - start);
        if(title[0] == '\n')
        {
            title.erase(0, 1);
        }
        return title;
    }
```

```
    std::string Task::get_content(const std::string &doc)    //取内容
    {
        int start = doc.find("<content>") + 9;
        int end = doc.find("</content>");
        string content(doc, start, end - start);
        if(content[0] == '\n')
        {
            content.erase(0, 1);
        }
        string line;
        istringstream iss(content);
        int count = m_out_line;
        string ret;
        while(getline(iss, line) && count > 0)
        {
            ret += line + "<br>";    //html的换行模式
            count--;
        }
        ret += "......";
        return ret;
    }
```

将查询结果制作成 Jason 字符串，发回客户端（前台页面）：

```
    /*
     * 将一个vector<pair<string,string> >做成json字符串
     * pair中存放两个string，分别是title和content
     */
    static std::string json_string(vector<pair<string, string> > &result_pair)
    {
        Json::Value root ;
        Json::Value arr ;
        for(vector<pair<string, string> >::iterator iter = result_pair.begin(); iter != result_pair.end(); ++iter)
        {
            Json::Value elem ;
            elem["title"] = iter->first ;
            elem["summary"] = iter->second ;
            arr.append(elem);
        }
        root["files"]=arr ;
        Json::FastWriter writer ;
        Json::StyledWriter stlwriter ;
        return stlwriter.write(root);
    }
```

第七阶段：前台页面

index.html 使用 javascript 通过 post 方法向 php 写的 tcp_client 发送查询 query，并将接收到的查询结果（Jason字符串）解析出来后显示在页面上:

```
    <script>
    //点击search按钮，执行其中的事件，注意js代码的注释与html代码的注释的区别
    $("#submitButton").click(function(){

        //取输入框的值
        var myWords=$("#txtSearch").val();
        //ajax请求，方法为post,php客户端返回的数据（echo）存在data变量中
     $.post("tcp_client.php",{content:myWords},function(data,status){
        if(status=="success")//post请求状态成功
        {
        //将收到的json字符串（data）转化为json对象，注意json字符串与json对象的区别
            var obj = eval("(" + data + ")");
           $("#result").html("");//清空result内容,用的是jquery的html()函数
         $.each(obj.files, function(i, item) {//遍历json对象，用的是jquery的each()方法，该json对象的格式近似于：{"files":[{"title":title_1,"s
                        $("#result").append(//将遍历到的数据显示在id为result这个div里面
                                //根据json对象的每一个子集的键显示相应的值，哟给你的是json的语法
                                "<div>" + item.title  + "</div>" +
                                "<div>" + item.summary+ "</div><hr/>");
                    });

        }
```

```
        else //post failure
        {

            alert(error);
        }

     });  //end post

        });

    </script><!--  javascript 结束 -->
```

php 客户端，主要功能是，当用户提交查询时，使用 tcp 协议向服务器发送查询词，并接收服务器发回的查询结果：

```
    <?php
    $buff=$_REQUEST["content"];//采用$_REQUEST超全局数组来接收index.html页面post请求传递过来的数据
    //tcp  client
    $server_Ip="127.0.0.1";//服务端ip地址，如果你的客户端与服务端不在同一台电脑，请修改该ip地址
    $server_Port=5080;//通信端口号

    //设置超时时间
    set_time_limit(0);
    //创建套接字
     $sock= socket_create(AF_INET,SOCK_STREAM,SOL_TCP);
    if(!$sock)
    {
        echo "creat sock failed";
        exit();//创建套接字失败，结束程序
    }

    socket_connect($sock,$server_Ip,$server_Port);

    //发送数据到tcp的服务端（C语言写的）
    socket_send($sock,$buff,strlen($buff),0);
    $buff="";//清空缓冲区
    socket_recv($sock,$buff,1024000,0);//接收tcp_server传递过来json字符串，存在变量$buff中

    echo trim($buff)."\n";//去掉接受到的字符串的首尾空格，返回给post请求的data
    //关闭套接字
    socket_close($sock);
    ?>
```