项目中几个第三方软件的使用

一、cppjieba的使用

cppjieba路径与安装

```
https://github.com/yanyiwu/cppjieba

//下载和编译
git clone --depth=10 --branch=master git://github.com/yanyiwu/cppjieba.git
cd cppjieba
mkdir build
cd build
cmake ..
make
```

将include下的整个文件夹cppjieba，放在要使用的文件中，然后将与include同目录的deps下面的limonp文件夹放在cppjieba中，然后将与include同目录的dict文件夹放在cppjieba中即可。

重点就是Cut函数以及CutXXX函数。创建jieba对象，然后使用Cut函数进行切割。（Cut的变种很多）

```
void test()
{
    cppjieba::Jieba jieba(DICT_PATH,
                          HMM_PATH,
                          USER_DICT_PATH,
                          IDF_PATH,
                          STOP_WORD_PATH);
    vector<string> words;
    vector<cppjieba::Word> jiebawords;
    string s;
    string result;

    s = "他来到了网易杭研大厦";
    cout << "[demo] Cut With HMM" << endl;
    jieba.Cut(s, words, true);
}
```

WordSegmentation.h

```
#ifndef _WORDSEGMENTATION_H_
#define _WORDSEGMENTATION_H_

#include "cppjieba/Jieba.hpp"

#include <iostream>
#include <string>
#include <vector>

using std::cout;
using std::endl;
using std::string;
using std::vector;
```

```cpp

const char * const DICT_PATH = "./cppjieba/dict/jieba.dict.utf8";//最大概率法(MPSegment: Max Probability)分词所使用的词典路径
const char * const HMM_PATH = "./cppjieba/dict/hmm_model.utf8";//隐式马尔科夫模型(HMMSegment: Hidden Markov Model)分词所使用的词典路径
const char * const USER_DICT_PATH = "./cppjieba/dict/user.dict.utf8";//用户自定义词典路径
const char* const IDF_PATH = "./cppjieba/dict/idf.utf8";//IDF路径
const char* const STOP_WORD_PATH = "./cppjieba/dict/stop_words.utf8";//停用词路径

class WordSegmentation//使用结巴分词库进行分词
{
public:
    WordSegmentation()
    : _jieba(DICT_PATH, HMM_PATH, USER_DICT_PATH,IDF_PATH,STOP_WORD_PATH)//初始化Jieba类对象
    {
        cout << "cppjieba init!" << endl;
    }

    vector<string> operator()(const string str)//返回str的分词结果
    {
        vector<string> words;
        _jieba.CutAll(str, words);//FullSegment
        return words;
    }
private:
    cppjieba::Jieba _jieba;
};

#endif
```

testJieba.cc

```cpp
#include "WordSegmentation.h"
#include <iostream>
#include <string>
#include <vector>

using std::cout;
using std::endl;
using std::string;
using std::vector;

int main()
{
    string str = "结巴分词库的下载和应用";
    WordSegmentation wordSeg;
    vector<string> results = wordSeg(str);
    cout << "分词结果如下:" << endl;

    for(auto it = results.begin(); it != results.end(); ++it)
    {
        cout << *it <<" ";
    }
    cout << endl;
    return 0;
}
```

```
cppjieba init!
分词结果如下:
结巴 分词 词库 的 下载 和 应用
```

二、Simhash算法

simhash算法的五个步骤：分词、哈希、加权、合并、降维。

在项目中的使用，可以看看example下的demo.cpp。三个接口：

```cpp
void test()
{
    Simhasher simhasher("../dict/jieba.dict.utf8",
                        "../dict/hmm_model.utf8",
                        "../dict/idf.utf8",
                        "../dict/stop_words.utf8");
    string s("我是蓝翔技工拖拉机学院手扶拖拉机专业的。不用多久，我就会升职加薪，当上总经理，出任CEO，走上人生巅峰。");
    size_t topN = 5;
    uint64_t u64 = 0;
    vector<pair<string ,double> > res;
    simhasher.extract(s, res, topN); //提取关键词与权重
    simhasher.make(s, topN, u64); //
    cout<< "文本：\"" << s << "\"" << endl;
    cout << "关键词序列是: " << res << endl;
    cout<< "simhash值是: " << u64 <<endl;

    const char * bin1 = "100010110110";
    const char * bin2 = "110001110011";
    uint64_t u1, u2;
    u1 = Simhasher::binaryStringToUint64(bin1);
    u2 = Simhasher::binaryStringToUint64(bin2);

    cout << bin1 << "和" << bin2 << " simhash值的相等判断如下："<<endl;
    cout << "海明距离阈值默认设置为3，则isEqual结果为：" 
        << (Simhasher::isEqual(u1, u2)) << endl;
    cout << "海明距离阈值默认设置为5，则isEqual结果为：" 
        << (Simhasher::isEqual(u1, u2, 5)) << endl;
    return EXIT_SUCCESS;
}
```

三、nlohmann/json的使用

环境安装与配置

```
https://github.com/nlohmann/json

//编译安装
mkdir build
cd build
cmake ..
make
sudo make install
```

类似于log4cpp的安装与使用，会在/usr/local/include下有nlohmann文件夹。

代码中的配置

引入头文件

```
#include <nlohmann/json.hpp>
using json = nlohmann::json;
```

```
#include <nlohmann/json.hpp>
/* #include "../include/nlohmann/json.hpp" */
#include <iostream>
#include <string>
#include <vector>

using std::cout;
using std::endl;
using std::string;
using std::vector;

using json = nlohmann::json;

struct Player
{
    string name;
    int credits;
    int ranking;
};
#if 0
void to_json(json &j, const Player &p)
{
    j= json{ {"name", p.name},
             {"credits", p.credits},
             {"ranking", p.ranking} };
}
#endif
#if 1
void from_json(const json &j, Player &p)
{
    j.at("name").get_to(p.name);
    j.at("credits").get_to(p.credits);
    j.at("ranking").get_to(p.ranking);
}
#endif
void test()
```

```
{
    auto j = R"([
    {"name": "Judd Trump","credits": 1754500,"ranking": 1},
    {"name": "Neil Robertson","credits": 1040500,"ranking": 2},
    {"name": "Ronnie O'Sullivan","credits": 954500,"ranking": 3}
    ])"_json;

    vector<Player> players = j.get<vector<Player>>();
    /* vector<Player> players; */
    /* j.get_to(players); */
    cout<< "name:" << players[2].name << endl;
    cout<< "credits:" << players[2].credits << endl;
    cout<< "ranking:" << players[2].ranking << endl;
}

void test3()
{
    auto j3 = json::parse(R"({"happy": true, "pi": 3.141})");
    cout << "j3 = " << j3 << endl;

    json j_string = "this is a string";
    auto cpp_string = j_string.get<std::string>();
    cout << "cpp_string = " << cpp_string << endl;
}

void test4()
{
    json j_string = "this is a string";

    auto cpp_string = j_string.get<std::string>();
    std::string cpp_string2;
    j_string.get_to(cpp_string2);

    std::string serialized_string = j_string.dump();

    std::cout << cpp_string
        << " == " << cpp_string2
        << " == " << j_string.get<std::string>() << '\n';
    std::cout << j_string
        << " == " << serialized_string << std::endl;
}

int main()
{
    test();

    return 0;
}
```

四、Redis的使用

1、hiredis相关

1.1、安装与编译

```
https://github.com/redis/hiredis
git clone https://github.com/redis/hiredis.git
tar -xzvf hiredis.tar.gz
cd hiredis
make
sudo make install //将可执行程序赋值到/usr/local/bin目录中，当执行程序中就不要输入完整的路径
sudo ldconfig (更新动态库配置文件) /usr/local/lib

make test   //测试redis是否编译正确

//编译时需要加上的后缀
g++ xxx.c -o xxx -I /usr/local/include/hiredis -lhiredis
或者直接g++ xxx.cc -lhiredis
```

安装完成后在需要使用的文件中加入如下代码

```
#include <hiredis/hiredis.h>
```

1.2、Redis的重要API

1.2.1、连接redis数据库

```
redisContext* redisConnect(const char *ip, int port)
redisContext* redisConnectWithTimeout(const char *ip, int port, timeval tv)
```

```
//redisContext不是线程安全的
typedef struct redisContext
{
    int err; /*错误标志，正确连接标志为0，出错时设置为非零常量*/
    char errstr[128]; /*存放错误信息的字符串*/
    int fd;
    int flags;
    char *obuf; /* Write buffer */
    redisReader *reader; /* Protocol reader */
} redisContext;
```

示例

```
redisContext *c = redisConnect("127.0.0.1", 6379);
if (c == NULL || c->err) {
    if (c) {
        printf("Error: %s\n", c->errstr);
        // handle error
    } else {
        printf("Can't allocate redis context\n");
    }
}
```

1.2.2、发送请求命令

第一个参数为连接数据库返回的值，剩余的是可变参数，类似printf。此函数的返回值是void *，但是一般会强制转换为redisReply类型，便于做进一步处理。

```
void *redisCommand(redisContext *c, const char *format...)
```

如果命令执行错误，返回值为NULL，`redisContext`的err字段被设置为**非零常量**。如果，错误发生，原先的redisContext就不能重复使用，需要重新建立一个新的连接。如果成功执行命令，则标准返回一个`redisReply`类型，该类型结构如下：

```
typedef struct redisReply
{
    int type; /* 测试收到什么样的回返回 REDIS_REPLY_* */
    long long integer; /* type 是 REDIS_REPLY_INTEGER 类型， integer保存返回的值*/
    int len; /* 保存str类型的长度 */
    char *str; /* type 是 REDIS_REPLY_ERROR 和 REDIS_REPLY_STRING，str保存返回的值 */
    size_t elements; /* type 是 REDIS_REPLY_ARRAY，保存返回多个元素的数量 */
    struct redisReply **element; /* 返回多个元素以redisReply对象的形式存放 */
} redisReply;

//type还可以是REDIS_REPLY_NIL，表示返回了一个零对象，没有数据可以访问。
```

通过redisReply结构体中的type变量可以确定命令执行的情况.

```
#define REDIS_REPLY_STRING 1    //字符串
#define REDIS_REPLY_ARRAY 2     //数组，例如mget返回值
#define REDIS_REPLY_INTEGER 3   //数字类型
#define REDIS_REPLY_NIL 4       //空
#define REDIS_REPLY_STATUS 5    //状态，例如set成功返回的‘OK’
#define REDIS_REPLY_ERROR 6     //执行失败
```

- REDIS_REPLY_STATUS：

返回执行结果为状态的命令。比如set命令的返回值的类型是REDIS_REPLY_STATUS，然后只有当返回信息是"OK"时，才表示该命令执行成功。可以通过reply->str得到文字信息，通过reply->len得到信息长度。

- REDIS_REPLY_ERROR：

返回错误。错误信息可以通过reply->str得到文字信息，通过reply->len得到信息长度。

- REDIS_REPLY_INTEGER：

返回整型标识。可以通过reply->integer变量得到类型为long long的值。

- REDIS_REPLY_NIL:

返回nil对象，说明不存在要访问的数据。

- REDIS_REPLY_STRING:

返回字符串标识。可以通过reply->str得到具体值，通过reply->len得到信息长度。

- REDIS_REPLY_ARRAY:

返回数据集标识。数据集中元素的数目可以通过reply->elements获得，每个元素是个redisReply对象，元素值可以通过reply->element[..index..].*形式获得，用在获取多个数据结果的操作。

1.2.3、释放资源

释放redisCommand执行后返回的的redisReply所占用的内存。

```
void freeReplyObject(void *reply)
```

释放redisConnect()所产生的连接

```
void redisFree(redisContext *c)
```

1.3、使用C语言测试

```
#include <hiredis/hiredis.h>
#include <stdio.h>


int main(int argc, char **argv)
{
    redisContext *pConnect = redisConnect("127.0.0.1", 6379);
    if(pConnect == NULL || pConnect->err)
    {
        if(pConnect)
        {
            printf("error: %s\n", pConnect->errstr);
            redisFree(pConnect);
            return -1;
        }
        else
        {
            printf("can not allocate redis context\n");
            return -1;
        }
    }

    printf("connect success\n");

    redisReply *pReply = (redisReply *)redisCommand(pConnect, "set %s %d", "boldness", 30);

    if(pReply == NULL)
    {
        printf("command error\n");
        redisFree(pConnect);
        return -1;
    }

    if(pReply->type == REDIS_REPLY_NIL)
    {
        printf("command error\n");
    }

    pReply = (redisReply *)redisCommand(pConnect, "get %s ", "boldness");
    if(pReply->type == REDIS_REPLY_NIL)
    {
        printf("get nil\n");
    }
    else if(pReply->type == REDIS_REPLY_STRING)
    {
        printf("get value: %s\n", pReply->str);
    }

    freeReplyObject(pReply);
    redisFree(pConnect);

    return 0;
}
```

1.4、使用C++进行封装

MyRdis.h

```
#ifndef __TESTREDIS_H__
#define __TESTREDIS_H__

#include <string.h>
#include <stdio.h>
#include <hiredis/hiredis.h>
#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::string;

class MyRedis
{
public:
    MyRedis()
    {
    }

    ~MyRedis()
    {
        _pConnect = nullptr;
        _pReply = nullptr;
    }

    bool connect(string host, int port)
    {
        _pConnect = redisConnect(host.c_str(), port);
        if(_pConnect != nullptr && _pConnect->err)
        {
            cout << "connect error : " << _pConnect->errstr << endl;
            return false;
        }
        return true;
    }

    string get(string key)
    {
        _pReply = (redisReply*)redisCommand(_pConnect, "GET %s",
key.c_str());
        string str = _pReply->str;

        freeReplyObject(_pReply);
        _pReply = nullptr;

        return str;
    }

    void set(string key, string value)
    {
        _pReply = (redisReply*)redisCommand(_pConnect, "SET %s %s",
key.c_str(), value.c_str());

        freeReplyObject(_pReply);
        _pReply = nullptr;
    }

private:
    redisContext* _pConnect;
    redisReply* _pReply;
};

#endif
```

TestRedis.cc

```
#include "MyRedis.h"

int main(int argc, char *argv[])
{
    MyRedis *pRedis = new MyRedis();
    if(!pRedis->connect("127.0.0.1", 6379))
    {
        cout << "connect error!" << endl;
        return 0;
    }

    pRedis->set("name", "Andy");
    cout << "Get the name is " << pRedis->get("name").c_str() << endl;

    delete pRedis;

    return 0;
}
```

2、redis-plus-plus相关

2.1、安装与编译

如果之前没有安装hiredis，在使用redis-plus-plus的时候需要先配置hiredis环境

```
git clone https://github.com/redis/hiredis.git
cd hiredis
make
sudo make install
sudo ldconfig
```

然后再安装redis-plus-plus

```
git clone https://github.com/sewenew/redis-plus-plus.git

cd redis-plus-plus
mkdir build
cd build
cmake -DREDIS_PLUS_PLUS_CXX_STANDARD=17 ..  #使用C++17版本，C++11/14则修改为
11/14
make
sudo make install
```

安装完成之后会在/usr/local/include下面生成类似log4cpp的文件夹hiredis、sw/redis++，相应的库文件在/usr/local/lib下面。

静态编译方式

```
g++ -std=c++17 -o app main.cpp /path/to/your/libredis++.a
/path/to/your/libhiredis.a -pthread

g++ -std=c++17 -o app main.cpp /usr/local/lib/libredis++.a
/usr/local/lib/libhiredis.a -pthread
```

动态编译方式（**首选**）

```
g++ redisCpp.cc -o main -lredis++ -lhiredis
g++ -std=c++17 -o app redisCpp.cc -lredis++ -lhiredis -pthread
```

注意动态链接时候，需要在"~/.bashrc"末尾为LD_LIBRARY_PATH添加"/usr/local/lib"路径即可：

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

头文件中需要加入如下头文件以及实体。

```
#include <sw/redis++/redis++.h>
using namespace sw::redis;
```

2.2、C++的使用

```
#include <sw/redis++/redis++.h>
#include <iostream>
#include <unordered_set>
#include <algorithm>

using namespace std;
using namespace sw::redis;
```

```cpp

// cout << vector
template <typename T>
std::ostream &operator<<(std::ostream &os, const std::vector<T> &v)
{
    if (!v.empty())
    {
        os << '[';
        std::copy(v.begin(), v.end(), std::ostream_iterator<T>(os, ", "));
        os << "\b\b]"; // 删除末尾的", "
    }
    return os;
}

// cout << unordered_map
template <typename T, typename U>
std::ostream &operator<<(std::ostream &os, const std::unordered_map<T, U> &umap)
{
    os << '[';
    for (auto item : umap)
    {
        os << "(" << item.first << "," << item.second << "),";
    }
    os << "\b]"; // 删除末尾的","

    return os;
}

// cout << unorderd_set
template <typename T>
std::ostream &operator<<(std::ostream &os, const std::unordered_set<T> &uset)
{
    os << '(';
    for (auto item : uset)
    {
        os << item << ",";
    }
    os << "\b)"; // 删除末尾的","

    return os;
}

int main()
{
    try
    {
        // Create an Redis object, which is movable but NOT copyable.
        auto redis = Redis("tcp://127.0.0.1:6379");

        /// ***** STRING commands *****
        redis.set("key", "val");
        // val is of type OptionalString. See 'API Reference' section for details.
        auto val = redis.get("key");
        if (val)
        {
            // Dereference val to get the returned value of std::string type.
            std::cout << *val << std::endl;
        } // else key doesn't exist.

        /// ***** LIST commands *****
        // std::vector<std::string> to Redis LIST.
        std::vector<std::string> vec = {"a", "b", "c"};
        redis.rpush("list", vec.begin(), vec.end());

        // std::initializer_list to Redis LIST.
        redis.rpush("list", {"a", "b", "c"});

        // Redis LIST to std::vector<std::string>.
        vec.clear();
        redis.lrange("list", 0, -1, std::back_inserter(vec));

        cout << "list: " << vec << endl;

        /// ***** HASH commands *****
        redis.hset("hash", "field", "val");

        // Another way to do the same job.
        redis.hset("hash", std::make_pair("field", "val"));

        // std::unordered_map<std::string, std::string> to Redis HASH.
        std::unordered_map<std::string, std::string> m =
        {
            {"field1", "val1"},
            {"field2", "val2"}
        };
        redis.hmset("hash", m.begin(), m.end());

        // Redis HASH to std::unordered_map<std::string, std::string>.
        m.clear();
        redis.hgetall("hash", std::inserter(m, m.begin()));

        cout << "hash:" << m << endl;

        // Get value only.
        // NOTE: since field might NOT exist, so we need to parse it to OptionalString.
        std::vector<OptionalString> vals;
        redis.hmget("hash", {"field1", "field2"}, std::back_inserter(vals));

        /// ***** SET commands *****
        redis.sadd("set", "m1");

        // std::unordered_set<std::string> to Redis SET.
        std::unordered_set<std::string> set = {"m2", "m3"};
        redis.sadd("set", set.begin(), set.end());

        // std::initializer_list to Redis SET.
        redis.sadd("set", {"m2", "m3"});

        // Redis SET to std::unordered_set<std::string>.
        set.clear();
```

```cpp
        redis.smembers("set", std::inserter(set, set.begin()));

        cout << "set:" << set << endl;

        if (redis.sismember("set", "m1"))
        {
            std::cout << "m1 exists" << std::endl;
        } // else NOT exist.

        /// ***** SORTED SET commands *****
        redis.zadd("sorted_set", "m1", 1.3);

        // std::unordered_map<std::string, double> to Redis SORTED SET.
        std::unordered_map<std::string, double> scores =
        {
            {"m2", 2.3},
            {"m3", 4.5}
        };
        redis.zadd("sorted_set", scores.begin(), scores.end());

        // Redis SORTED SET to std::vector<std::pair<std::string, double>>.
        // NOTE: The return results of zrangebyscore are ordered, if you save the results
        // in to `std::unordered_map<std::string, double>`, you'll lose the order.
        std::vector<std::pair<std::string, double>> zset_result;
        redis.zrangebyscore("sorted_set",
                            UnboundedInterval<double>{}, // (-inf, +inf)
                            std::back_inserter(zset_result));

        // Only get member names:
        // pass an inserter of std::vector<std::string> type as output parameter.
        std::vector<std::string> without_score;
        redis.zrangebyscore("sorted_set",
                            BoundedInterval<double>(1.5, 3.4, BoundType::CLOSED),
                            // [1.5, 3.4]
                            std::back_inserter(without_score));

        // Get both member names and scores:
        // pass an back_inserter of std::vector<std::pair<std::string, double>> as output parameter.
        std::vector<std::pair<std::string, double>> with_score;
        redis.zrangebyscore("sorted_set",
                            BoundedInterval<double>(1.5, 3.4, BoundType::LEFT_OPEN),
                            // (1.5, 3.4]
                            std::back_inserter(with_score));

        /// ***** SCRIPTING commands *****
        // Script returns a single element.
        auto num = redis.eval<long long>("return 1", {}, {});

        // Script returns an array of elements.
        std::vector<std::string> nums;
        redis.eval("return {ARGV[1], ARGV[2]}", {},
                   {"1", "2"}, std::back_inserter(nums));

        // mset with TTL
        auto mset_with_ttl_script = R"(
        local len = #KEYS
        if (len == 0 or len + 1 ~= #ARGV) then return 0 end
        local ttl = tonumber(ARGV[len + 1])
        if (not ttl or ttl <= 0) then return 0 end
        for i = 1, len do redis.call("SET", KEYS[i], ARGV[i], "EX", ttl) end
        return 1
        )";

        // Set multiple key-value pairs with TTL of 60 seconds.
        auto keys = {"key1", "key2", "key3"};
        std::vector<std::string> args = {"val1", "val2", "val3", "60"};
        redis.eval<long long>(mset_with_ttl_script, keys.begin(), keys.end(),
                              args.begin(), args.end());

        /// ***** Pipeline *****
        // Create a pipeline.
        auto pipe = redis.pipeline();

        // Send mulitple commands and get all replies.
        auto pipe_replies = pipe.set("key", "value").get("key")
                                .rename("key", "new-key")
                                .rpush("list", {"a", "b", "c"})
                                .lrange("list", 0, -1)
                                .exec();

        // Parse reply with reply type and index.
        auto set_cmd_result = pipe_replies.get<bool>(0);

        auto get_cmd_result = pipe_replies.get<OptionalString>(1);

        // rename command result
        pipe_replies.get<void>(2);

        auto rpush_cmd_result = pipe_replies.get<long long>(3);

        std::vector<std::string> lrange_cmd_result;
        pipe_replies.get(4, back_inserter(lrange_cmd_result));

        /// ***** Transaction *****
        // Create a transaction.
        auto tx = redis.transaction();

        // Run multiple commands in a transaction, and get all replies.
        auto tx_replies = tx.incr("num0")
                              .incr("num1")
                              .mget({"num0", "num1"})
                              .exec();

        // Parse reply with reply type and index.
        auto incr_result0 = tx_replies.get<long long>(0);

        auto incr_result1 = tx_replies.get<long long>(1);

        std::vector<OptionalString> mget_cmd_result;
        tx_replies.get(2, back_inserter(mget_cmd_result));

        /// ***** Generic Command Interface *****
        // There's no *Redis::client_getname* interface.
        // But you can use *Redis::command* to get the client name.
        val = redis.command<OptionalString>("client", "getname");
        if (val)
        {
            std::cout << *val << std::endl;
        }

        // Same as above.
        auto getname_cmd_str = {"client", "getname"};
        val = redis.command<OptionalString>(getname_cmd_str.begin(),
                                            getname_cmd_str.end());

        // There's no *Redis::sort* interface.
        // But you can use *Redis::command* to send sort the list.
        std::vector<std::string> sorted_list;
        redis.command("sort", "list", "ALPHA",
std::back_inserter(sorted_list));

        // Another *Redis::command* to do the same work.
        auto sort_cmd_str = {"sort", "list", "ALPHA"};
        redis.command(sort_cmd_str.begin(), sort_cmd_str.end(),
                      std::back_inserter(sorted_list));

        /// ***** Redis Cluster *****
        // Create a RedisCluster object, which is movable but NOT copyable.
        auto redis_cluster = RedisCluster("tcp://127.0.0.1:7000");

        // RedisCluster has similar interfaces as Redis.
        redis_cluster.set("key", "value");
        val = redis_cluster.get("key");
        if (val)
        {
            std::cout << *val << std::endl;
        } // else key doesn't exist.

        // Keys with hash-tag.
        redis_cluster.set("key{tag}1", "val1");
        redis_cluster.set("key{tag}2", "val2");
        redis_cluster.set("key{tag}3", "val3");

        std::vector<OptionalString> hash_tag_res;
        redis_cluster.mget({"key{tag}1", "key{tag}2", "key{tag}3"},
                           std::back_inserter(hash_tag_res));
    }
    catch (const Error &e)
    {
        // Error handling.
    }

    return 0;
}
```