boost 备忘录

时间:: 2019年

---

tokenizer:分词

```
#include<boost/tokenizer.hpp>
using  namespace boost;
bool tokenizer_test()
{
    string s = "This is a test";
    tokenizer<> token1(s);
    for(auto item:token1){
        cout<<item<<endl;//空格划分
    }
  //第二种:
    string str = ";;I|love||-the--one;is;you|";
    char_separator<char> sep("-;|");
    tokenizer<char_separator<char>> token2(str, sep);
            //escaped_list_separator<char>
            //offset_separator
    for(auto item:token2){
        cout<<item<<" ";//I love the one is you
    }
    cout<<endl;
  //第三种:
    string s1 = "12252001";
    int offsets[] = {2,2,4};
    offset_separator f(offsets, offsets+3);
    tokenizer<offset_separator> tok(s1,f);
    for(auto item:tok){
        cout<<item <<" "; //12 25 2001
    }
    cout<<endl;
    return true;
}
```

内存池:poo1

```
#include<boost/pool/pool.hpp>
using namespace boost;
//
//内存池操作:head-only library
//std:vector使用:pool_allocator
//std:list使用:fast_pool_allocator
//
#include<boost/pool/singleton_pool.hpp>
#include<boost/pool/object_pool.hpp>
#include<boost/pool/pool_alloc.hpp>
#include<boost/pool/poolfwd.hpp>
#include<boost/pool/simple_segregated_storage.hpp>
struct X{int i;};
bool pool_test()
{
    //形式1
    boost::pool<> p(sizeof(int));
    for (int i = 0; i < 10000; ++i)
    {
        void * const t = p.malloc();
    }
    //形式2
    boost::object_pool<X> p1;
    for (int i = 0; i < 10000; ++i){X * const t = p1.malloc();}

    //形式3
    typedef boost::singleton_pool<X, sizeof(int)> my_pool;
    for (int i = 0; i < 10000; ++i){
        void * const t = my_pool::malloc();
    }
    my_pool::purge_memory();

    return true;
}
```

安全数值库

```
//  The Safe Numerics library i
//C++14,依赖较多;溢出报错
//

#include <boost/safe_numerics/safe_integer.hpp>
using namespace boost::safe_numerics;
safe<int> func(safe<int> x, safe<int> y){
  return x + y;
}
```

文件和目录操作

```
-lboost_filesystem
#include<boost/filesystem.hpp>
using namespace boost;
//前面必须有#inlcude<boost/*>
//否则:error: ‘boost’ is not a namespace-name


//文件系统
// absolute   canonical
// copy  copy_directory copy_file  copy_symlink
// create_directories   create_directory   create_hard_link     create_symlink
// exists[存在性]  equivalent   hard_link_count
// initial_path. is_directory, is_empty
// is_other is_regular_file  is_symlink
// last_write_time    permissions    read_symlink    relative
// remove  remove_all  rename  resize_file   space
// status  status_known   symlink_status   system_complete
bool fs_test()
{
    auto filename="log.txt";
    filesystem::path p1{"/usr/bin"};
    cout<<"文件大小为:"<<filesystem::file_size(filename)<<"字节"<<endl;
    cout<<"常规文件:"<<filesystem::is_regular_file(filename)<<endl;
    cout<<"是目录:"<<filesystem::is_directory(p1)<<endl;
    int i=0;
    for (filesystem::directory_entry& item : filesystem::directory_iterator(p1)){
        //cout<<item.path()<<endl;  //文件的路径
        cout<<item.path().filename()<<endl; //只有文件名
        i++;
        if(i>10)break;
    }
    cout<<"根路径:"<< p1.root_path() <<endl;
    cout<<"名字"<<p1.stem()<<endl;
    cout<<"后缀"<<p1.extension()<<endl;
    cout<<"为空:"<<p1.empty()<<endl;
    cout<<"绝对路径:"<<p1.is_absolute()<<endl;
    cout<<"有root_name: "<<(p1.has_root_name()?"true":"false")<<endl;
    cout<<"有根目录:"<<p1.has_root_directory()<<endl;
    cout<<"有根路径:"<<p1.has_root_path()<<endl;
    cout<<"有相对路径:"<<p1.has_relative_path()<<endl;
    cout<<"有父目录:"<<p1.has_parent_path()<<endl;
    cout<<"有文件名:"<<p1.has_filename()<<endl;
    cout<<"有stem;"<<p1.has_stem()<<endl;
    cout<<"有后缀;"<<p1.has_extension()<<endl;
    cout<<"string表示:"<<p1.string()<<endl;
    cout<<"通用string表示:"<<p1.generic_string()<<endl;
    auto p2=filesystem::current_path();
    cout<<"当前目录为:"<<p2.string()<<endl;
    cout<<"状态:"<<status(p2).type()<<endl;
    cout<<"状态:"<<status(p2).permissions()<<endl;
    cout<<"最后修改时间:"<<last_write_time(p2)<<endl;
    filesystem::rename(filename,"log_record.txt");
    remove("log_record.txt");
    return true;
}
```

大数: multiprecision

```
//
//uint128_t,uint256_t,uint512_t,uint1024_t;
//int128_t      ...
//cpp_rational
//
#include<boost/multiprecision/cpp_int.hpp> //慢,gmp_int快
#include<boost/multiprecision/cpp_bin_float.hpp>
#include<boost/multiprecision/cpp_dec_float.hpp>
//#include<boost/multiprecision/gmp.hpp>  //mpf_float,mpf_float_50,mpf_float_500,mpf_float_1000
//#include<boost/multiprecision/mpfr.hpp> //mpfr_float,mpfr_float_100,mpfr_float_1000
#include<boost/multiprecision/float128.hpp>
#include<boost/multiprecision/cpp_complex.hpp>


bool multiprecision_func()
{
    multiprecision::cpp_int big_int=1234254;//任意精度int
    big_int=big_int*big_int*big_int*big_int*big_int;
    cout<<"int大数为:"<<big_int<<endl;

    multiprecision::cpp_bin_float_100   big_float=0.234345456;
    big_float*=big_float;
    cout<<"float大数为:"<<big_float<<endl;

    multiprecision::cpp_complex_100 complex1={123,234};
    cout<<"实部"<<complex1.real()<<endl;
    cout<<"虚部"<<imag(complex1)<<endl;

    multiprecision::cpp_rational rat_val=123;
    for(unsigned i=1;i<1000;i++ ){
        rat_val*=i;
    }
    cout<<std::cout.precision(10)<<endl;//设置小数精度
    cout<<"有理数:"<<rat_val<<endl;
    return true;
}
```

运行时间测量:timer

```
//      运行时间测量函数
//  -lboost_timer -lboost_chrono
//  还有timer类:用于定时等
#include <boost/timer/timer.hpp>
bool sub_func()
{
    timer::auto_cpu_timer sub_t;
    cout<<"子函数调用时间"<<endl;
    return true;
}
bool timer_test()
{
    timer::auto_cpu_timer t;//定义时开始计时
    cout<<"计时间过程"<<endl;
    vector<string> asd{"qweasd","asdxcv","dfgyert","tyugvhbn"};
    for(auto i:asd){
        cout<<i<<endl;
    }
    sub_func();
    return true;
}
```

随机数 boost::random

```
//
// 各种分布:仍待探索
//
#include<boost/random/bernoulli_distribution.hpp>
#include<boost/random/beta_distribution.hpp>
#include<boost/random/binomial_distribution.hpp>
#include<boost/random/cauchy_distribution.hpp>
#include<boost/random/chi_squared_distribution.hpp>
#include<boost/random/discrete_distribution.hpp>
#include<boost/random/exponential_distribution.hpp>
#include<boost/random/extreme_value_distribution.hpp>
#include<boost/random/gamma_distribution.hpp>
#include<boost/random/fisher_f_distribution.hpp>
#include<boost/random/generate_canonical.hpp>
#include<boost/random/geometric_distribution.hpp>
#include<boost/random/hyperexponential_distribution.hpp>
#include<boost/random/laplace_distribution.hpp>
#include<boost/random/lognormal_distribution.hpp>
#include<boost/random/negative_binomial_distribution.hpp>
#include<boost/random/non_central_chi_squared_distribution.hpp>
#include<boost/random/normal_distribution.hpp>
#include<boost/random/piecewise_constant_distribution.hpp>
#include<boost/random/piecewise_linear_distribution.hpp>
#include<boost/random/poisson_distribution.hpp>
#include<boost/random/student_t_distribution.hpp>
#include<boost/random/triangle_distribution.hpp>
#include<boost/random/uniform_int_distribution.hpp>
#include<boost/random/uniform_real_distribution.hpp>
#include<boost/random/weibull_distribution.hpp>

#include<boost/random/uniform_01.hpp>
#include<boost/random/uniform_smallint.hpp>
```

```
#include<boost/random/uniform_on_sphere.hpp>


//
//  生成器
//
#include<boost/random/linear_congruential.hpp> //minstd_rand0  minstd_rand rand48
#include<boost/random/additive_combine.hpp>    //ecuyer1988
#include<boost/random/shuffle_order.hpp>       //knuth_b, kreutzer1986
#include<boost/random/taus88.hpp>              //taus88
#include<boost/random/inversive_congruential.hpp>//hellekalek1995
#include<boost/random/mersenne_twister.hpp>    //mt11213b, mt19937
#include<boost/random/lagged_fibonacci.hpp>
//lagged_fibonacci607, lagged_fibonacci1279, lagged_fibonacci2281
//lagged_fibonacci3217, lagged_fibonacci4423, lagged_fibonacci9689
//lagged_fibonacci19937, lagged_fibonacci23209, lagged_fibonacci44497
#include<boost/random/ranlux.hpp>
//ranlux3, ranlux4, ranlux64_3,ranlux64_4, ranlux3_01, ranlux4_01
//ranlux64_3_01, ranlux64_4_01, ranlux24, ranlux48

///
///其他
///
#include<boost/random/seed_seq.hpp>
#include<boost/random/random_number_generator.hpp>
#include<boost/random/generate_canonical.hpp>


random::mt19937 rng;
int rand_one(int start=1,int end=600){
    //random::uniform_int_distribution<> diset(start,end);
    //random::student_t_distribution<> diset;
    random::normal_distribution<> diset(start,end);

    return diset(rng);
}
int gailv(){
    double probabilities[] = {0.1, 0.2, 0.1, 0.3, 0.1, 0.2};
    random::discrete_distribution<> dist(probabilities);
    return dist(rng)+1;
}
#include<boost/random/random_device.hpp>
random::random_device rng1;
bool mima(){
//该函数需要:-lboost_random
    string table="abcdefghijklmnopqrstuvwxyz1234567890";

    random::uniform_int_distribution<> index_dist(0, 35);
    for(int i=0;i<10;i++){
        cout<<table[index_dist(rng1)]<<",";
    }
    cout<<"mima"<<endl;
    return true;
}
bool rand_test()//再试试其他的分布和生成器
{
    int s=0,s1=0,m=0,e1=0,e=0;
    int start=1;
    int end=600;
```

```
    for(int i=0;i<1000;i++){
        int tmp=rand_one();
        //int tmp=gailv();
        if(tmp<= (end/5))s++;
        else if(tmp<= (end*0.4))s1++;
        else if(tmp<= (end*0.6))m++;
        else if(tmp<= (end*0.8))e1++;
        else e++;
    }
    cout<<"从小到达的数量:"
        <<s<<"-"
        <<s1<<"-"
        <<m<<"-"
        <<e1<<"-"
        <<e<<endl;

    //mima();

    return true;
}
```

协程 coroutine2

```
//
// coroutine2 :provides asymmetric coroutines.
//   协程: 相当于python的yield 程序在任一位置中断或恢复执行
//       关键 就是执行的流程控制
//       The implementation uses Boost.Context for context switching
//   适合事件驱动模型
//   实现: fcontext_t:默认实现,基于汇编,有平台依赖,性能最好
//        ucontext_t:跨平台,性能稍差
//   对称协程:显示yield;非对称协程:隐式转移控制权[本库实现方式]
//   编译选项:-lboost_context
//
#include <boost/coroutine2/all.hpp>
bool called_func(coroutines2::coroutine<void>::push_type & qwe)//无值传递
{
    cout << "一 ";
    qwe();
    cout << "二 ";
    qwe();
    cout << "三 ";
    return true;
}
bool call_func()
{
    //push_type 与 pull_type 的位置可互换 从push开始向下执行
    coroutines2::coroutine<void>::pull_type goto_qwe(called_func);
    cout << "1 ";
    goto_qwe();
    cout << "2 ";
    goto_qwe();
    cout << "3 "<<endl;
    return true;
}
bool called_func1(coroutines2::coroutine<unsigned int>::push_type& qwe)//有值传递
{
```

```
    for(int i=0;i<10;i++){
        cout <<i<< " ";
        qwe(++i);
    }
    return true;
}
bool call_func1()
{
    coroutines2::coroutine<unsigned int>::pull_type goto_qwe(called_func1);
    /*
    //第一种方法:
    unsigned int  i=1;
    for(;i<10;i++){
        cout <<goto_qwe.get()<<" ";
        goto_qwe();
    }
    //第二种方法
    using iter = coroutines2::coroutine<unsigned int>::pull_type::iterator;
    for (iter start = begin(goto_qwe); start != end(goto_qwe); ++start) {
    cout<< *start << " ";
    }
    */
    //第三种方法:
    for(auto val:goto_qwe){
        cout<<val<<" ";
    }

    cout <<endl;
    return true;
}
```

串算法库 string algo

```
#include<iostream>
#include<string>
#include<boost/algorithm/string_regex.hpp>
#include<boost/algorithm/string.hpp> // split join replace
#include <boost/lexical_cast.hpp>
#include <boost/convert.hpp>
#include <boost/regex.hpp>

using namespace boost;

void show_str(string one_str){cout<<one_str+";"<<endl;}
bool str_opt()
{
    string one_str="   啊撒了123看到    年份,12314s gdfg1bf. SDFG345SDFUK    ";
    cout<<"1、size:"<<one_str.size()<<endl;
    cout<<"2、length:"<<one_str.length()<<endl;
    //to_lower(),to_upper(one_str); show_str(one_str);
    cout<<"3、转为小写"<<to_lower_copy(one_str) <<endl;

    // 去掉开头结尾空格trim(),trim_left,trim_right,trim_left_copy
    //trim_copy_if(str, boost::is_alnum());
    cout<<"4、去开头空格: "<<trim_left_copy(one_str)<<endl;

    cout<<"5、starts_with:"<<starts_with(" BBCSDRFG","BBC")<<endl;
```

```
cout<<"6、ends_with: "<<ends_with("踩踩julia","julia") <<endl;

cout<<"7、contains(包含 in): "<<contains("123zhong456","zhong") <<endl;
cout<<"8、equals（串相等）"<<equals("julia123","julia123") <<endl;
cout<<"9、字典序比较: "<< lexicographical_compare("abvfg","bwerf")<<endl;
//cout<<"10、all(元素相同):"<<all("aaaaa",[](char i){return i=='a'?true:false;}) <<endl;
cout<<"10、all(元素相同):"<<all("aaaaa",is_any_of("a") ) <<endl;

//find_last,find_nth,
cout<<"11、find_first:"<<find_first(one_str,"123") <<endl;  //怎么用?返回123不是索引
    //if(find_first(one_str,"看到")){cout<<"找到了"<<endl;}else{cout<<"未找到"<<endl;}
cout<<"12、检索头"<<find_head(one_str,6) <<endl;  //find_tail
iterator_range<string::iterator> iter1=find_token(one_str,is_any_of("312"));
cout<<"13、find_token:"<<string(iter1.begin(),iter1.end()) <<endl;  //怎么用???

regex re1{"[0-9]+"};
iterator_range<string::iterator> iter=find_regex(one_str,re1);
cout<<"14、find_regex:"<<string(iter.begin(),iter.end()) <<endl;
//cout<<"15、find: "<<find(one_str,first_finder("查找的串",is_iequal()) ) <<endl;

//{replace | erase}_{first | last | all | nth | head | tail | regex}_{copy|""}
// 28中组合
replace_first(one_str,"啊撒了","Julia");show_str(one_str);
replace_last(one_str,"年份","天狼星");show_str(one_str);
replace_all(one_str,"gdf","aab");show_str(one_str);
erase_all(one_str,"1");show_str(one_str);  //删除所有的1
//replace_all_regex_copy()

//find_all,find_all_regex,iter_find,iter_split
vector<string> neirong;
find_all_regex(neirong,one_str,re1);
cout<<"16、find_all_regex 个数"<<neirong.size()<<" "<<neirong[1]<<endl;  //****常用****
find_all(neirong,one_str,"23");
cout<<"17、find_all: "<<neirong[1]<<endl;

vector<string> str_vec;
regex e{"\\s+"};
//split(str_vec,one_str,is_any_of(" "));
split_regex(str_vec,one_str,e);
for(auto val:str_vec){cout<<val+"+"<<endl;}

//join,join_if（比join多个谓词）
string single_str=join(str_vec,"+");
cout<<"18、join连接: "<<single_str<<endl;
unsigned int y=142;
cout<<"19、字面量转换 : "<<lexical_cast<string>(y)<<endl; //类型转换142->"142"
//string s2 = convert<string>(100).value();
//cout<<"20.convert: "<< s2<<endl;


//finder
//{first,last,nth,head,tail,token,range,regex}_finder

//Formatters 格式化器
//const_formatter,identity_formatter,empty_formatter,regex_formatter

//迭代器
//find_iterator,split_iterator
```

```
    return true;
}
```

Circular Buffer{循环缓冲,vector}

```
#include <boost/circular_buffer.hpp>
bool buffer()
{
    circular_buffer<string> buffer5(5);
    buffer5.push_back("1.sdf");
    buffer5.push_back("2.啊撒");
    buffer5.push_back("3.撒地方了");
    buffer5.push_back("4.333");
    buffer5.push_back("5.让他鱼");
    string example=buffer5[3];
    buffer5[3]="4.电饭锅";
    buffer5.pop_back();
    buffer5.pop_front();
    for(auto i : buffer5){
        cout<<"["<<i<<endl;
    }
    return true;
}
```

序列化 Serialization

```
//  ===     ========================
//
//   1. 非自定义对象:写个函数(文件名,对象,存储方式)直接存储
//   2. 自定义对象: 在类中添加方法
//
//   编译时添加lib库 : -lboost_serialization
//
//  ===     ========================
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/archive/text_woarchive.hpp> //宽字符 utf8
#include <boost/archive/text_wiarchive.hpp>

#include <boost/archive/xml_oarchive.hpp>
#include <boost/archive/xml_iarchive.hpp>
#include <boost/archive/xml_woarchive.hpp>
#include <boost/archive/xml_wiarchive.hpp>

#include <boost/archive/binary_oarchive.hpp>
#include <boost/archive/binary_iarchive.hpp>

#include <boost/serialization/vector.hpp>
#include <boost/serialization/map.hpp>
#include <boost/serialization/set.hpp>
//
//  自定义类 的序列化
//
class self_def
```

```
{
public:
    //类内数据定义
    vector<float> aaa;
    set<string> bbb;
    string ccc;

    friend class boost::serialization::access;

    template<class Archive>
    void serialize(Archive &ar,unsigned int version)
    {
        ar & aaa & bbb & ccc ;//读取,写入均可
    }

    self_def(vector<float>& val1,set<string>& val2,string val3){
        this->aaa=val1;
        this->bbb=val2;
        this->ccc=val3;
    }
    self_def(){};

};
//
//  函数对象 实现序列化与反序列化
//
template<class object>
class Arch_out
{
public:
    bool operator()(string filename,object& obj,string type="xml")
    {
        ofstream out_stream(filename);

        if(type=="text"){
            archive::text_oarchive out_ar(out_stream);
            out_ar<<obj;
        }
        else if(type=="xml"){
            archive::xml_oarchive out_ar(out_stream);
            out_ar<<BOOST_SERIALIZATION_NVP(obj);
        }
        else{}

        // out_ar && obj;  //相同
        return true;
    }
    friend class boost::serialization::access;
};
template<class object>
class Arch_in  //该class代码未测试
{
public:
    bool operator()(string filename,object& obj,string type="xml")
    {
        ifstream in_stream(filename);

        if(type=="text"){
            archive::text_iarchive in_ar(in_stream);
            in_ar>>obj;
```

```
        }
        else if(type=="xml"){
            archive::xml_iarchive in_ar(in_stream);
            in_ar>>BOOST_SERIALIZATION_NVP(obj);
        }
        else{}
        return true;
    }
    friend class boost::serialization::access;
};

//
//  泛型 实现序列化与反序列化
//
template<class object>
bool arch_out(string filename,object& obj,string type="xml")
{
    ofstream out_stream(filename);
    //多试几种存储方式
    if(type=="txt"){
        archive::text_oarchive out_ar(out_stream);
        out_ar<<obj;
    }
    else if(type=="xml"){
        archive::xml_oarchive out_ar(out_stream);
        out_ar<<BOOST_SERIALIZATION_NVP(obj);
    }
    else if(type=="bin"){
        archive::binary_oarchive out_ar(out_stream);
        out_ar<<obj;
    }
    /*对‘vtable for boost::archive::codecvt_null<wchar_t>’未定义的引用
    //      -lboost_wserialization 加上无效,难道依赖C++20的u8string

    if(type=="wtxt"){
        wofstream out_stream(filename);
        archive::text_woarchive out_ar(out_stream);
        out_ar<<obj;
    }
    else if(type=="wxml"){
        wofstream out_stream(filename);
        archive::xml_woarchive out_ar(out_stream);
        out_ar<<BOOST_SERIALIZATION_NVP(obj);
    } */
    else{
        //type参数的内容:未知
        cerr<<"type 参数错误"<<endl;
    }
    out_stream.close();
    return true;
}
template<class object>
bool arch_in(string filename,object& obj,string type="xml")
{
    ifstream in_stream(filename);//stringstream is ok

    if(type=="txt"){
        archive::text_iarchive in_ar(in_stream);
        in_ar>>obj;
    }
```

```
    else if(type=="xml"){
        archive::xml_iarchive in_ar(in_stream);
        in_ar>>BOOST_SERIALIZATION_NVP(obj);
    }
    else if(type=="bin"){//binary
        archive::binary_iarchive in_ar(in_stream);
        in_ar>>obj;
    }
    /*
    else if(type=="wtxt"){
        wifstream in_stream(filename);
        archive::text_wiarchive in_ar(in_stream);
        in_ar>>obj;
    }
    else if(type=="wxml"){
        wifstream in_stream(filename);
        archive::xml_wiarchive in_ar(in_stream);
        in_ar>>BOOST_SERIALIZATION_NVP(obj);
    } */
    else{
    //wait to fill
    }
    in_stream.close();
    return true;
}
//
//  序列化测试函数
//
#include<map>
#include<set>
bool Serial_test()
{
    //1. 多试几种类型 map set vector<int string long>
    vector<long> example_obj{234,456,2342456,6784523,7897435,123457};
    map<string,long> map_tmp{{"C++",1},{"julia",2},{"python",3},{"Perl",4}};
    set<string> set_tmp{"喀什的愤怒","速度快了女","sdlvn","所得率几年","送到了房间"};

    set<string> tmp;


    //2. 多次类型
    auto filename="archive";
    //方法1 函数对象
    //Arch_out<vector<long>> CC;
    //CC(filename,example_obj);
    //方法2 泛型
    //*
    vector<string> type{"txt","bin","xml"};
    for(auto ll:type){
        arch_out(filename,set_tmp,ll);
        arch_in(filename,tmp,ll);
        //for(auto i:tmp){cout<<i.first<<"="<<i.second<<" ";} //map
        for(auto i:tmp){cout<<i<<" ";}
        cout<<ll<<"_tested"<<endl;
    }//
    //方法3 自定义类的序列化
    vector<float> val1={3.12443,5.2347,2345.3456,2345.789};
    set<string> val2={"喀什的愤怒","速度快了女","sdlvn","所得率几年","送到了房间"};
    string val3="撒的开ak sdnf发那可asdf怜检索2354;多次";
```

```
    self_def item1{val1,val2,val3};
    self_def item2;

    ofstream out_stream(filename);
    archive::text_oarchive out_ar(out_stream);
    out_ar<<item1;
    out_stream.close();

    ifstream in_stream(filename);
    archive::text_iarchive in_ar(in_stream);
    in_ar>>item2;
    /*
    arch_out<self_def>(filename,item1,"txt");  //该部分出现的错误,尚未解决
    arch_in <self_def>(filename,item2,"txt");   //暂时使用上面七行代码解决
    */
    cout<<item2.aaa[0]<<endl;;
    cout<<item2.bbb.size()<<endl;
    cout<<item2.ccc<<endl;

    return true;
}
```

-lboost_serialization 编译时需要添加的命令后缀

其他内容

日志功能的简单实现

```
using namespace std;
#include<vector>
#include<fstream>
bool writelines(vector<string> log_record)
{
    ofstream log_file("log.txt",ios::app);
    for(auto item:log_record){
        log_file<<item<<endl;
    }
    return true;
}
vector<string> log_record{"项目执行日志\n","时间:2019- \n"};
bool log_add(string log_info,bool write2file=false)
{
    log_record.push_back(log_info);
    if(write2file==true){
        writelines(log_record);
        log_record.clear();
    }
    return true;
}
//调用方式:
//  log_add("项目执行结束.",true);
//将上述代码放到main函数前
```

C++实现print

```
//
// this code from boost doc
//  print("为恶哦","asdf","啊撒地方",123);
//
void print(){}
template<class T, class... Ts>
void print(const T& x, const Ts&... xs)
{
    cout << x;
    print(xs...);
}
```

其他未详细看的内容

```
//=======================================
//      boost 的 相关库
//
//  Range:增强stl 的可读性
//  core:核心工具,无依赖,元编程的小函数,
//  tribool: <boost/logic/tribool.hpp>: 三值:true,false,indeterminate
//  beast:网络编程:request,repose
//  Graph:图论中的图,
//  icl:区间,interval_set,interval_map
//
//
//
//
//==================================

//
//Higher-order functions
//
//1.函数指针:&函数名 2.函数对象
//包含:BOOST_HOF_STATIC_FUNCTION,BOOST_HOF_STATIC_LAMBDA
//      BOOST_HOF_LIFT(函数名)
//感觉没啥用啊
//
//
constexpr auto pi=3.141592657;
constexpr auto two_pi=2*pi;//常量表达式:编译时求值
cout<<"常量表达式"<<two_pi<<endl;
```

笔记

- //proto:领域专用语言;嵌入式;表达式模板
- //statechart:transform a UML statechart into executable C++ code
- //DLL:动态链接库的使用
- //Boost.System:扩展的错误报告
- //process:子进程,使用pipe通信
- //Interprocess:进程间通信
- //fiber: 可以在线程间转移???

- //lockfree: 生产者消费者数据结构
- //xpressive:正则表达式模板库
- // 1.regex_match() 2.regex_search() 3.regex_replace ()
- // 4. regex_iterator<> 5.regex_token_iterator<>
- //Signals2 :信号&&槽(未看)
- //minmax:比min和max比较次数少

//什么时候用多线程?
// 1.大规模网络爬虫
// 2.大规模网络请求
// 3.密集的计算任务
//noexcept: 该关键字告诉编译器, 函数中不会发生异常
// 确定函数不发生异常的根据,或判断标准是什么?
//token : an individual instance of a type of symbol

regex:有时间再看!!!!

完