Module05-03
C++ Boost：正则表达式

C++ Boost - 正则表达式

- 容器相关
- 字符串和文字处理
- ➔ 正则表达式
- 智能指针
- 函数对象相关
- 序列化
- 日期与时间
- 多线程
- 网络

正则表达式 - 语法

- 正则表达式：
  - 语法
  - 类 regex
  - 类 match_results
  - 类 sub_match
  - 算法 regex_match()、 regex_search()、 regex_replace()
  - 迭代器 regex_iterator、 regex_token_iterator
  - 示例

正则表达式 - 语法

- 正则表达式语法
  - boost.regex 的语法默认基于 Perl 语言的正则表达式语法
  - 语法包括匹配 (搜索) 和替代两个部分
  - 具体语法描述见 Boost Regex Doc - Perl Regular Expression Syntax 一节
  - (参考语法测试示例： regex_syntax.txt)

正则表达式 - 类 regex

- 正则表达式：
  - 语法
  - 类 regex
  - 类 match_results
  - 类 sub_match
  - 算法 regex_match()、 regex_search()、 regex_replace()
  - 迭代器 regex_iterator、 regex_token_iterator
  - 示例

正则表达式 - 类 regex

- 关于 class regex
  - class regex 用一系列的字符构建一个合法的正则表达式，本质上讲就是一个 string(字符串对象)
  - 如果给定的字符序列无法构建一个合法的正则表达式，将抛出一个异常，以示构造失败

正则表达式 - 类 regex

- 构造、复制

```
template<class charT, class traits = regex_traits<charT> >
class basic_regex {
public:
    // construct/copy/destroy:
    explicit basic_regex();
    explicit basic_regex(const charT* p,
            flag_type f = regex_constants::normal);
    basic_regex(const charT* p1, const charT* p2,
            flag_type f = regex_constants::normal);
    basic_regex(const charT* p, size_type len, flag_type f);
    basic_regex(const basic_regex&);
    template<class ST, class SA>
    explicit basic_regex(const basic_string<charT, ST, SA>& p,
            flag_type f = regex_constants::normal);
    template<class InputIterator>
    basic_regex(InputIterator first, InputIterator last,
            flag_type f = regex_constants::normal);
    ~basic_regex();
```

正则表达式 - 类 regex

- 赋值

```
    basic_regex& operator=(const basic_regex&);
    basic_regex& operator=(const charT* ptr);

    template<class ST, class SA>
    basic_regex& operator=(const basic_string<charT, ST, SA>&
p);
    basic_regex& assign(const basic_regex& that);
    basic_regex& assign(const charT* ptr,
            flag_type f = regex_constants::normal);
    basic_regex& assign(const charT* ptr,
            unsigned int len, flag_type f);
    template<class string_traits, class A>
    basic_regex& assign(
        const basic_string<charT, string_traits, A>& s,
        flag_type f = regex_constants::normal);
    template<class InputIterator>
    basic_regex& assign(InputIterator first,
        InputIterator last,
        flag_type f = regex_constants::normal);
```

正则表达式 - 类 regex

- 常用操作

```
// iterators:
std::pair<const_iterator, const_iterator>
subexpression(size_type n) const;

const_iterator begin() const;
const_iterator end() const;

// capacity:
size_type size() const;
size_type max_size() const;
bool empty() const;
unsigned mark_count() const;

string str() const;
```

正则表达式 - 类 regex

- 使用 regex
  - 包含头文件 <boost/regex.hpp>

```
using boost::regex;
regex e1;
e1 = "^[[:xdigit:]]*$";
cout << e1.str() << endl;
cout << e1.mark_count() << endl; // 1

regex e2("\\b\\w+(?=ing)\\b.{2,}?([[:alpha:]]*)$",
        regex::perl | regex::icase |
        regex::save_subexpression_location);
cout << e2.str() << endl;
cout << e2.mark_count() << endl;  // 2

// 如果regex e2 未设save_subexpression_location
// subexpression()将会抛出异常
pair<regex::const_iterator, regex::const_iterator> sub1 =
        e2.subexpression(1);
string sub1Str(sub1.first, ++sub1.second);
cout << sub1Str << endl;
```

正则表达式 - 类 match_results

- 正则表达式：
  - 语法
  - 类 regex
  - 类 match_results
  - 类 sub_match
  - 算法 regex_match()、 regex_search()、 regex_replace()
  - 迭代器 regex_iterator、 regex_token_iterator
  - 示例

正则表达式 - 类 match_results

- 关于 class match_results
  - class match_results 是记录一个正则表达式匹配过程中，每个子表达式的（或整个表达式）的匹配结果，如：
    - 表达式：'\b\w+\b ([0-6]*)' 匹配字符串：'abcd 12 a_n 005'
      - 整个表达式命中 2 次：'abcd 12' 和 'a_n 005'
      - 第 1 个子表达式 '([0-6]*)' 也命中 2 次：'12' 和 '005'
  - match_results 中通过一个或多个 sub_match 对象记录这些命中的字符串的起止位置（而不是真正意义上的字符串内容，只记录代表位置的迭代器）
  - match_results 的内部拥有一个容纳 sub_match 对象的 vector（或其它序列容器）
  - 注意：如果有匹配，则整个表达式匹配的结果作为第一个 sub_match 放在 match_results 中（下标为 0 ）

正则表达式 - 类 match_results

- 构造、复制、析构

```
template <class BidirectionalIterator,
class Allocator = std::allocator<sub_match<BidirectionalIterator> >
class match_results {
public:
    // construct/copy/destroy:
    explicit match_results(const Allocator& a = Allocator());
    match_results(const match_results& m);
    match_results& operator=(const match_results& m);
    ~match_results();

    // size:
    size_type size() const;
    size_type max_size() const;
    bool empty() const;

    // ...
};
```

正则表达式 - 类 match_results

- 一些内部定义的类型：

```
template <class BidirectionalIterator,
class Allocator =
std::allocator<sub_match<BidirectionalIterator> >
class match_results {
public:
    // ...
    typedef sub_match<BidirectionalIterator> value_type;
    typedef const value_type& const_reference;
    typedef const_reference reference;
    // ...
};
```

正则表达式 - 类 match_results

- 常用操作

```
    // element access:
    difference_type length(int sub = 0) const;
    difference_type length(const char_type* sub) const;
    template <class charT>
    difference_type length(const charT* sub) const;
    template <class charT, class Traits, class A>
    difference_type length(const std::basic_string<charT,
Traits, A>& sub) const;

    difference_type position(unsigned int sub = 0) const;
    difference_type position(const char_type* sub) const;
    template <class charT>
    difference_type position(const charT* sub) const;
    template <class charT, class Traits, class A>
    difference_type position(const std::basic_string<charT,
Traits, A>& sub) const;
```

正则表达式 - 类 match_results

- 常用操作（续 1 ）

```
    string_type str(int sub = 0) const;
    string_type str(const char_type* sub)const;
    template <class Traits, class A>
    string_type str(const std::basic_string<char_type, Traits,
A>& sub)const;
    template <class charT>
    string_type str(const charT* sub)const;
    template <class charT, class Traits, class A>
    string_type str(const std::basic_string<charT, Traits, A>&
sub)const;
    const_reference operator[](int n) const;
    const_reference operator[](const char_type* n) const;
    template <class Traits, class A>
    const_reference operator[](const
std::basic_string<char_type, Traits, A>& n) const;
    template <class charT>
    const_reference operator[](const charT* n) const;
    template <class charT, class Traits, class A>
    const_reference operator[](const std::basic_string<charT,
Traits, A>& n) const;
```

正则表达式 - 类 match_results

- 常用操作（续2）

```
const_reference prefix() const;

const_reference suffix() const;
const_iterator begin() const;
const_iterator end() const;
// format:
template <class OutputIterator, class Formatter>
OutputIterator format(OutputIterator out,
        Formatter fmt,
        match_flag_type flags = format_default) const;
template <class Formatter>
string_type format(Formatter fmt,
        match_flag_type flags = format_default) const;

allocator_type get_allocator() const;
void swap(match_results& that);
```

正则表达式 - 类 match_results

- typedef of match_results

```
using namespace std;
template <class BidirectionalIterator,
          class Allocator =
std::allocator<sub_match<BidirectionalIterator> >
class match_results;

typedef match_results<const char*>                cmatch;
typedef match_results<const wchar_t*>             wcmatch;
typedef match_results<string::const_iterator>     smatch;
typedef match_results<wstring::const_iterator>    wsmatch;
```

正则表达式 - 类 match_results

- 使用 match_results
  - 使用类 match_results 须包含头文件： <boost/regex.hpp>

```
using boost::regex;
regex e1("\\bT\\w+\\b ([[:xdigit:]]+)");

string s("Time ef09,Todo 001");
boost::smatch m;

bool b = boost::regex_search(s, m, e1, boost::match_all);
cout << b << endl;

const int n = m.size();
for (int i = 0; i < n; ++i) {
    cout << "matched " << i << " position: "
         << m.position(i) << ", ";
    cout << "length: " << m.length(i) << ", str: ";
    cout << m.str(i) << endl;
}
```

正则表达式 - 类 sub_match

- 正则表达式：
  - 语法
  - 类 regex
  - 类 match_results
  - 类 sub_match
  - 算法 regex_match()、 regex_search()、 regex_replace()
  - 迭代器 regex_iterator、 regex_token_iterator
  - 示例

正则表达式 - 类 sub_match

- 关于 class sub_match
  - 一个 sub_match 就是一个 pair<Iter, Iter> 类型，用来保存某个子表达式匹配结果的起止位置（指向源字符串的迭代器），如：
    - 表达式 e： '(ab)[df2]' 匹配字符串 s： 'kkyabfm'
      - 子表达式 '(ab)' 匹配目标字符串 s 中的 ab，范围是：
        s.begin()+3 到 s.begin()+5，
        则 sub_match 对象 sm 的
        sm.first = s.begin()+3,
        sub_match.second = s.begin()+5
  - sub_match 一般配合 match_results 使用

正则表达式 - 类 sub_match

- class sub_match 接口

```
template<class BIter>
class sub_match: public std::pair<BIter, BIter> {
public:
    typedef typename iterator_traits<BIter>::value_type
value_type;
    typedef typename iterator_traits<BIter>::difference_type
difference_type;
    typedef BIter iterator;

    bool matched;

    difference_type length() const;
    operator basic_string<value_type>() const;
    basic_string<value_type> str() const;

    int compare(const sub_match& s) const;
    int compare(const basic_string<value_type>& s) const;
    int compare(const value_type* s) const;
};
```

正则表达式 - 算法函数

- 正则表达式：
  - 语法
  - 类 regex
  - 类 match_results
  - 类 sub_match
  - 算法 regex_match()、 regex_search()、 regex_replace()
  - 迭代器 regex_iterator、 regex_token_iterator
  - 示例

正则表达式 - 算法函数 regex_match

- 关于 regex_match 函数
  - regex_match 用于一个表达式匹配整个目标字符串的情形，该函数通常用于输入验证，如：
    - 表达式 e： '([a-d]+)mk' 匹配字符串 s： '#aabmkT'
      - 尽管表达式 e 可以匹配 s 的 aabmk，但函数 regex_match() 还是返回 false，因为表达式 e 没有完整的匹配完 s 整个字符串
  - 如果要查询某个字符串中是否含有指定样式的串，应该使用 regex_search()
  - regex_match() 函数有 6 种形式：
    - 三种类型输入源（一对迭代器指代一个字符序列、 string 类型、 char* 类型）
    - 上述 3 中形式各有保存匹配结果和不保存匹配结果 2 中形式

正则表达式 - 算法函数 regex_match

- regex_match 函数接口

```
template<class BIter, class Allocator,
class charT, class traits>
bool regex_match(BIter first, BIter last,
        match_results& m,  // 保存匹配结果
        const regex& e,
        match_flag_type flags = match_default);

template<class BIter, class charT, class traits>
bool regex_match(BIter first, BIter last,
        const regex& e,
        match_flag_type flags = match_default);

template<class charT, class Allocator, class traits>
bool regex_match(const charT* str,
        match_results& m,  // 保存匹配结果
        const regex& e,
        match_flag_type flags = match_default);
```

正则表达式 - 算法函数 regex_match

- regex_match 函数接口（续）

```
template<class charT, class traits>
bool regex_match(const charT* str,
        const regex<charT, traits>& e,
        match_flag_type flags = match_default);

template<class ST, class SA, class Allocator,
class charT, class traits>
bool regex_match(const string& s,
        match_results& m,  // 保存匹配结果
        const regex& e,
        match_flag_type flags = match_default);

template<class ST, class SA, class charT, class traits>
bool regex_match(const string& s,
        const regex& e,
        match_flag_type flags = match_default);
```

正则表达式 - 算法函数 regex_match

- regex_match 示例

```
using boost::regex;
regex e("\\w{6,}");

string line;
while (getline(cin, line)) {
    if (line == "quit")
        break;

    if (regex_match(line, e, boost::match_default))
        cout << "Matched" << endl;
    else
        cout << "No Match" << endl;
}
```

正则表达式 - 算法函数 regex_search

- 关于 regex_search 函数
  - regex_search 的接口同 regex_match 类似，与 regex_match 不同的是， regex_search 在表达式匹配源字符串的部分或全部时返回 true ， 如：
    - 表达式 e： '([a-d]+)mk' 匹配字符串 s： '#aabmkT'
      - 表达式 e 可以匹配 s 的 aabmk ， 所以函数 regex_search() 返回 true
  - regex_search() 函数有 6 种形式：
    - 三种类型输入源（一对迭代器指代一个字符序列、 string 类型、 char* 类型）
    - 上述 3 中形式各有保存匹配结果和不保存匹配结果 2 中形式

正则表达式 - 算法函数 regex_search

- regex_search 函数接口

```
template<class BIter, class Allocator,
class charT, class traits>
bool regex_search(BIter first, BIter last,
        match_results& m,  // 保存匹配结果
        const regex& e,
        match_flag_type flags = match_default);

template<class BIter, class charT, class traits>
bool regex_search(BIter first, BIter last,
        const regex& e,
        match_flag_type flags = match_default);

template<class charT, class Allocator, class traits>
bool regex_search(const charT* str,
        match_results& m,  // 保存匹配结果
        const regex& e,
        match_flag_type flags = match_default);
```

正则表达式 - 算法函数 regex_search

- regex_search 函数接口（续）

```
template<class charT, class traits>
bool regex_search(const charT* str,
        const regex<charT, traits>& e,
        match_flag_type flags = match_default);

template<class ST, class SA, class Allocator,
class charT, class traits>
bool regex_search(const string& s,
        match_results& m,  // 保存匹配结果
        const regex& e,
        match_flag_type flags = match_default);

template<class ST, class SA, class charT, class traits>
bool regex_search(const string& s,
        const regex& e,
        match_flag_type flags = match_default);
```

正则表达式 - 算法函数 regex_search

- regex_search 示例

```
using boost::regex;
regex e1("\\bT\\w+\\b ([[:xdigit:]]+)");

string s("Time ef09,Todo 001");
boost::smatch m;

bool b = boost::regex_search(s, m,
            e1, boost::match_default);
cout << *(m[1].first) << ' ' << *(m[1].second) << endl;
cout << "----------------------" << endl;
const int n = m.size();
for (int i = 0; i < n; ++i) {
    cout << "matched " << i << " position: "
         << m.position(i) << ", ";
    cout << "length: " << m.length(i) << ", str: ";
    cout << m.str(i) << endl;
}
```

正则表达式 - 算法函数 regex_replace

- 关于 regex_replace 函数
  - regex_replace 在匹配的同时作替换动作
  - 接口：

```
template<class OIter, class BIter,
    class traits, class Formatter>
OIter regex_replace(OIter out, BIter first, BIter last,
        const regex& e,
        Formatter fmt,
        match_flag_type flags = match_default);

template<class traits, class Formatter>
string regex_replace(const string& s,
        const regex& e, Formatter fmt,
        match_flag_type flags = match_default);
```

正则表达式 - 算法函数 regex_replace

- regex_replace 函数示例

```
using boost::regex;
regex e1("([TQV])|(\\*)|(@)");
string replaceFmt("(\\L?1$&)(?2+)(?3#)");
string src("guTdQhV@@g*b*");
cout << "before replaced: " << src << endl;
// 注意: format_all
string newStr1 = regex_replace(src, e1, replaceFmt,
        boost::match_default | boost::format_all);
cout << "after replaced: " << newStr1 << endl;

// 注意: format_default
string newStr2 = regex_replace(src, e1, replaceFmt,
        boost::match_default | boost::format_default);
cout << "after replaced: " << newStr2 << endl;

// 另一种形式的调用
ostream_iterator<char> oi(cout);
regex_replace(oi, src.begin(), src.end(), e1, replaceFmt,
        boost::match_default | boost::format_all);
```

正则表达式 - 迭代器

- 正则表达式：
  - 语法
  - 类 regex
  - 类 match_results
  - 类 sub_match
  - 算法 regex_match()、 regex_search()、 regex_replace()
  - 迭代器 regex_iterator、 regex_token_iterator
  - 示例

正则表达式 - 迭代器 regex_iterator

- 关于迭代器 regex_iterator
  - regex_iterator 可以迭代一次匹配过程中匹配的结果，该迭代器实际上指向的是 match_results 对象

正则表达式 - 迭代器 regex_iterator

- 迭代器 regex_iterator 接口

```
template<class BIter, class charT =
iterator_traits<BIter>::value_type,
         class traits = regex_traits<charT> >
class regex_iterator {
public:
    typedef basic_regex<charT, traits> regex_type;
    typedef match_results<BIter> value_type;
    typedef typename iterator_traits<BIter>::difference_type
difference_type;
    typedef const value_type* pointer;
    typedef const value_type& reference;
    typedef std::forward_iterator_tag iterator_category;

    regex_iterator();
    regex_iterator(BIter a, BIter b, const regex_type& re,
            match_flag_type m = match_default);
    regex_iterator(const regex_iterator&);
```

正则表达式 - 迭代器 regex_iterator

- 迭代器 regex_iterator 接口（续）

```
    regex_iterator& operator=(const regex_iterator&);
    bool operator==(const regex_iterator&) const;
    bool operator!=(const regex_iterator&) const;
    const value_type& operator*() const;
    const value_type* operator->() const;
    regex_iterator& operator++();
    regex_iterator operator++(int);
};
```

```
typedef regex_iterator<const char*> cregex_iterator;
typedef regex_iterator<std::string::const_iterator>
sregex_iterator;

#ifndef  BOOST_NO_WREGEX
typedef regex_iterator<const wchar_t*> wcregex_iterator;
typedef regex_iterator<std::wstring::const_iterator>
wsregex_iterator;
#endif
```

正则表达式 - 迭代器 regex_iterator

- 迭代器 regex_iterator 相关的辅助函数

```
template<class charT, class traits>
regex_iterator<const charT*, charT, traits>
make_regex_iterator(const charT* p, const regex& e,
        regex_constants::match_flag_type m =
regex_constants::match_default);

template<class charT, class traits, class ST, class SA>
regex_iterator<typename string::const_iterator, charT, traits>
make_regex_iterator(const string& p,
        const regex<charT, traits>& e,
        regex_constants::match_flag_type m =
regex_constants::match_default);
```

正则表达式 - 迭代器 regex_iterator

- 迭代器 regex_iterator 示例

```
using boost::regex;
regex e("(a+).+?", regex::icase);

string s("ann abb aaat");

boost::sregex_iterator it1(s.begin(), s.end(), e);
boost::sregex_iterator it2; // 结束迭代器

for (;it1 != it2; ++it1) {
    boost::smatch m = *it1;
    cout << m << endl;
}
```

正则表达式 - 迭代器 regex_token_iterator

- 关于迭代器 regex_token_iterator
  - 模板类 regex_token_iterator 是迭代器适配器； 它以新的方式 (遍历序列中正则表达式的所有出现，并以一个或多个字符序列表示每个匹配) 表示一个已经存在的迭代器序列。 迭代器遍历的每个位置都是一个 sub_match 对象，表示正则表达式中的特定子表达式匹配。

    当类 regex_token_iterator 用序号 -1 来遍历单个子表达式时，迭代器执行区域分割： 也就是说，遍历所有未被表达式匹配的文本序列。

正则表达式 - 迭代器 regex_token_iterator

- 迭代器 regex_token_iterator 接口

```
template<class BIter, class charT =
iterator_traits<BIter>::value_type,
        class traits = regex_traits<charT> >
class regex_token_iterator {
public:
    typedef basic_regex<charT, traits> regex_type;
    typedef sub_match<BIter> value_type;
    // ...

    regex_token_iterator();
    regex_token_iterator(BIter a, BIter b, const regex_type&
re, int submatch = 0, match_flag_type m = match_default);
    regex_token_iterator(BIter a, BIter b,
        const regex_type& re,
        const std::vector<int>& submatches,
        match_flag_type m = match_default);
```

正则表达式 - 迭代器 regex_token_iterator

- 迭代器 regex_token_iterator 接口（续）

```
    template<std::size_t N>
    regex_token_iterator(BIter a, BIter b,
        const regex_type& re, const int(&submatches)[N],
        match_flag_type m = match_default);
    regex_token_iterator(const regex_token_iterator&);
    regex_token_iterator& operator=(const
regex_token_iterator&);

    bool operator==(const regex_token_iterator&) const;
    bool operator!=(const regex_token_iterator&) const;
    const value_type& operator*() const;
    const value_type* operator->() const;
    regex_token_iterator& operator++();
    regex_token_iterator operator++(int);
};
```

正则表达式 - 迭代器 regex_token_iterator

- typedef of regex_token_iterator

```
typedef regex_token_iterator<const char*>
cregex_token_iterator;

typedef regex_token_iterator<std::string::const_iterator>
sregex_token_iterator;

#ifndef BOOST_NO_WREGEX
typedef regex_token_iterator<const wchar_t*>
wcregex_token_iterator;

typedef regex_token_iterator<<std::wstring::const_iterator>
wsregex_token_iterator;
#endif
```

正则表达式 - 迭代器 regex_token_iterator

- 迭代器 regex_token_iterator 辅助函数

```
template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m =
regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<typename string::const_iterator, charT,
traits>
   make_regex_token_iterator(
         const string& p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m =
regex_constants::match_default);
```

正则表达式 - 迭代器 regex_token_iterator

- 迭代器 regex_token_iterator 辅助函数（续 1 ）

```
template <class charT, class traits, std::size_t N>
regex_token_iterator<const charT*, charT, traits>
make_regex_token_iterator(
          const charT* p,
          const basic_regex<charT, traits>& e,
          const int (&submatch)[N],
          regex_constants::match_flag_type m =
regex_constants::match_default);

template <class charT, class traits,
           class ST, class SA, std::size_t N>
regex_token_iterator<
      typename string::const_iterator, charT, traits>
   make_regex_token_iterator(
          const std::basic_string<charT, ST, SA>& p,
          const basic_regex<charT, traits>& e,
          const int (&submatch)[N],
          regex_constants::match_flag_type m =
regex_constants::match_default);
```

正则表达式 - 迭代器 regex_token_iterator

- 迭代器 regex_token_iterator 辅助函数（续 2 ）

```
template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m =
regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<
      typename string::const_iterator, charT, traits>
   make_regex_token_iterator(
         const string& p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m =
regex_constants::match_default);
```

正则表达式 - 迭代器 regex_token_iterator

- 迭代器 regex_token_iterator 示例

```
    string s("This is a string of tokens");
    boost::regex re("\\s+");
    boost::sregex_token_iterator i(s.begin(), s.end(), re,
-1);
    boost::sregex_token_iterator j;

    unsigned count = 0;
    while (i != j) {
        cout << *i++ << endl;
        count++;
    }
    cout << "There were " << count << " tokens found." <<
endl;
```

正则表达式 - 示例

- 正则表达式：
  - 语法
  - 类 regex
  - 类 match_results
  - 类 sub_match
  - 算法 regex_match()、 regex_search()、 regex_replace()
  - 迭代器 regex_iterator、 regex_token_iterator
  - 示例

正则表达式 - 示例

- 示例
  - (DEMO Using Boost Regex Examples)

正则表达式 - More Stuffs...

- boost.regex 正则表达式提供了一个非常简约并且方便的接口集，所以调用该库的接口十分轻松。
- boost.regex 中对于匹配的字符串的记录，不是将其 copy 并储存，而是利用 sub_match 来保存匹配字符串序列的起止位置（一对迭代器），所以使用的 boost.regex 的开销很小
- 相对于 boost.regex 的简约接口，其所能支持的正则表达式的语法则非常完备和丰富，要充分发挥 boost.regex 这一利器的威力，理解正则表达式的语法是关键，下面是一些资源：
  - Mastering Regular Expressions, 3rd Edition (Book)
  - http://www.regular-expressions.info/examples.html
- boost.xpressive 是另一个优秀的正则表达式库