# 【正则表达式】regular expression

【简写】

在代码中常简写为 regex , regexp 或 RE

正则表达式又常被称为模式（pattern）

【解决问题】

问题场景不外乎两种情况——

①搜索

②替换

【注意】

语法是正则表达式里最容易掌握的部分，真正的挑战是学会运用这些语法把实际问题分解为一系列正则表达式，并最终解决。

编写正则表达式的时候，并非只有对错之分，解决问题的方案往往不止一种。

正则表达式的不同实现往往会有所差异，有些差异和不兼容是无法回避的。

# 【C++中的regex】

#include <regex>

using namespace std;

【C++ regex函数有3个】

|  |  |
| --- | --- |
| regex\_match | 全文匹配，要求整个字符串符合匹配规则 |
| regex\_search | 搜索字符串中存在符合规则的子字符串 |
| regex\_replace | 将符合匹配规则的子字符串替换为其他字符串 |

正则表达式的匹配结果是一样的，只是函数的行为不同？

## 【smatch】匹配结果

std::smatch sm;

sm.prefix();//匹配的子字符串的前缀

sm.str();//匹配的子字符串

sm.suffix();//匹配的子字符串的后缀

sm[0];//匹配的子字符串

sm[1];//匹配的子字符串中，子正则表达式匹配的信息

//2、3、4......

## 【regex\_match】

整个目标序列必须与此函数的正则表达式匹配才能返回true（即，在匹配之前或之后没有任何其他字符）

//字面量

bool flag = regex\_match("subject", regex("(sub)(.\*)"));

//string对象

string str("subject");

regex re("(sub)(.\*)");

regex\_match(str, re);

//范围匹配

regex\_match(str.begin(), str.end(), re);

//结果是C风格字符串列表？

cmatch cm; // same as std::match\_results<const char\*> cm;

regex\_match("subject", cm, re);

cout << "string literal with " << cm.size() << " matches\n";

//结果是string对象列表？

smatch sm; // same as std::match\_results<string::const\_iterator> sm;

regex\_match(str, sm, re);

cout << "string object with " << sm.size() << " matches\n";

regex\_match(str.cbegin(), str.cend(), sm, re);

cout << "range with " << sm.size() << " matches\n";

// using explicit flags:

regex\_match("subject", cm, re, regex\_constants::match\_default);

std::cout << "the matches were: ";

for (unsigned i = 0; i<sm.size(); ++i) {

std::cout << "[" << sm[i] << "] ";

}

## 【regex\_search】

搜索字符串中存在符合规则的子字符串，在查找到第一个匹配的子串后，就会停止查找。想在剩余的部分中继续搜索，就是继续搜索sm.prefix()的内容。

#include <iostream>

#include <string>

#include <regex>

int main()

{

std::string text("Roses are #ff0000, violets are #0000ff, all of my base are belong to you");

std::regex re("#([a-f0-9]{2})"

"([a-f0-9]{2})"

"([a-f0-9]{2})");

//【简单匹配】（只获知是否匹配）

std::cout << "简单匹配\n";

std::cout << text << "\n";

std::cout << std::boolalpha << std::regex\_search(text, re) << '\n';

//std::boolalpha让流把布尔值解释为单词

//【重复搜索】（参阅 std::regex\_iterator ）

std::cout << "重复搜索\n";

std::smatch sm;

while (std::regex\_search(text, sm, re))

{

std::cout << "Prefix: " << sm.prefix() << "\n";//前缀

std::cout << "Match: " << sm.str() << '\n';//sm[0]

std::cout << "Suffix: " << sm.suffix() << "\n";//后缀

std::cout << "匹配内容 与 子表达式的匹配内容：\n";

for (int i = 0; i < sm.size(); i++)

std::cout << sm[i] << std::endl;

//继续对剩余后缀进行匹配

text = sm.suffix();//有可能sm是对text的引用，因为这一句提前将无法操作sm[i]

}

//【C风格字符串演示】

std::cout << "C 风格字符串演示\n";

std::cmatch cm;

if (std::regex\_search("this is a test", cm, std::regex("test")))

std::cout << "\nFound " << cm[0] << " at position " << cm.prefix().length();

system("pause");

return 0;

}

## 【raw string】待学习

未加工的字符串，R"( )"把字符串放在括号中，\n将会被当成两个字符

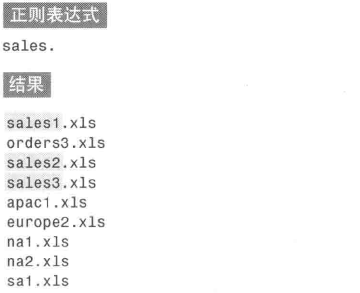
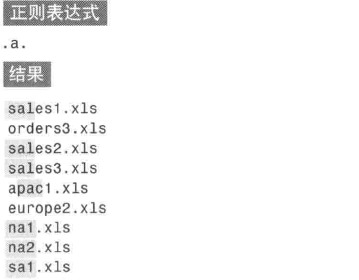
# 【匹配单个字符】

## 【匹配纯文本】【文本】



## 【匹配单个任意字符】【.】

应该说是能够匹配除了换行符以外的任意一个字符。

## 【转义字符】【\】

把【.】转义为普通字符，不再是代表“匹配单个任意字符”的元字符（metacharacter）



【C++中的转义与正则表达式中的转义】

如果要匹配windows路径中的【\】，按照正则表达式的转义语法：

【\\】

然后按照C++中string的转义语法，两个【\】都需要转义：

std::regex re("\\\\");

此时如果使用raw string就会简单许多：

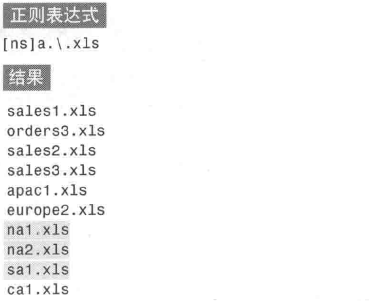
std::regex re(R"(\\)");

# 【匹配一组字符】

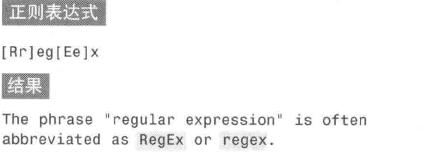
## 【匹配一组字符】【[ ]】

元字符[ ]中的字符组成了字符集合，能够匹配集合中的任意给一个成员。

第一个例子中的集合能够匹配n或者s。



在匹配一个字母大小写的时候也很有用：



## 【连字符】【-】

使用正则表达式时会频繁用到一些字符区间，如0-9、A-Z、a-z等，为了简化区间的定义可以使用【-】。

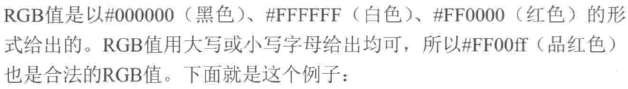
模式[0-9]的功能与[0123456789]完全相同

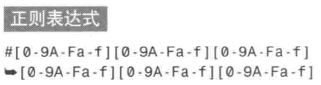
字符区间的首位可以是ASCII字符集里的任意字符，但实际中常用的还是数字区间和字母区间。

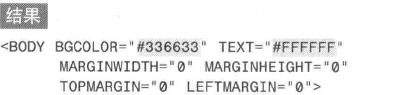
字符【-】在【[ ]】以外的地方是普通字符，不需要转义。

在同一个字符集合里可以有多个字符区间：

[A-Za-z0-9]将匹配所有字母和数字

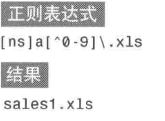






## 【取非匹配】【^】

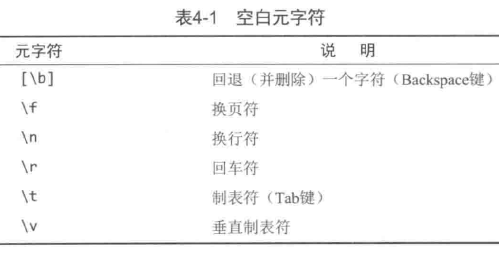
把元字符【^】放入【[ ]】中，可以对一个字符集合进行取非匹配。





# 【使用元字符】

## 【空白元字符】

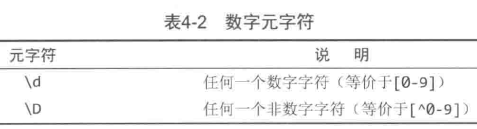


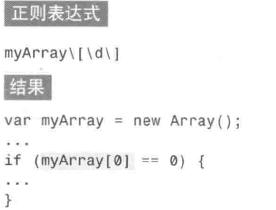
\r\n匹配一个“回车+换行”组合，windows系统的把这个组合作为文本的行结束标签

## 【匹配特定的字符类别】

专业术语是字符类。

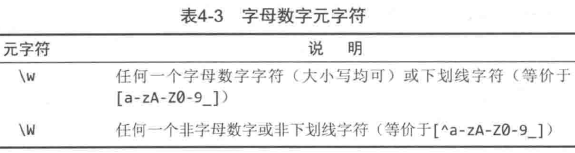
### 【匹配数字】【\d】

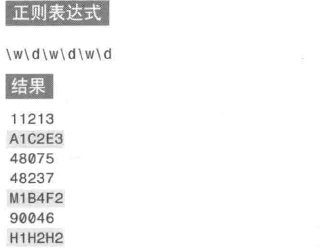




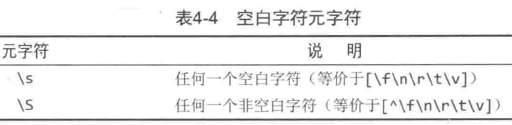
### 【匹配字母和数字】【\w】

包括下划线\_



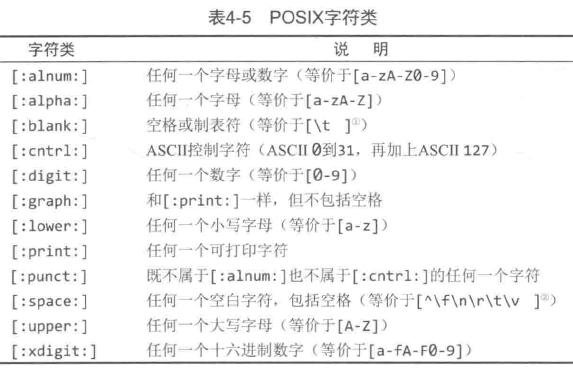
注意是匹配6个字符

### 【匹配空白字符】【\s】



## 【POSIX字符类】

不一定支持这种写法。





# 【重复匹配】

【匹配一个或多个字符】【+】

想要匹配一个字符（或字符集合），只要给该字符（或字符集合）加上【+】作为后缀。



