

Free.L

思想比知识更重要，成长比成功更重要

[主页](#) [博客](#) [相册](#) [个人档案](#) [好友](#)

[查看文章](#)

语 法 解 析 器 **Boost Spirit** 入门

2009年09月14日 星期一 11:52

Boost.Spirit能使我们轻松地编写出一个简单脚本的语 法 解 析 器，它巧妙利用了元编程并重载了大量的C++操作符使得我们能够在C++里直接使用类似EBNF的语 法 构造出一个完整的语 法 解析器(同时也把C++弄得面目全非-_-)。

关于EBNF的内容大家可以到网上或书店里找：

EBNF基本形式

<符号> ::= <表达式> 或 <符号> = <表达式>

表达式里常用的操作符有：

- | 分隔符，表示由它分隔的某一个子表达式都可供选择
- * 重复，和正则表达式里的*类似，表示它之前的子表达式可重复多次
- 排除，不允许出现跟在它后面的那个子表达式
- , 串接，连接左右子表达式
- ； 终止符，一条规则定义结束
- " 字符串
- "" 字符串
- (...) 分组，就是平时括号的功能啦，改变优先级用的。
- (*...*) 注释
- [...] 可选，综括号内的子表达式允许出现或不出现
- {...} 重复，大括号内的子表达式可以多次出现
- ?...? 特殊字符，由ISO定义的一些特殊字例如：

只允许赋值的简单编程语言可以用 EBNF 定义为：

```
(* a simple program in EBNF ? Wikipedia *)
program = 'PROGRAM' , white space , identifier , white space ,
         'BEGIN' , white space ,
         { assignment , ":", white space } ,
         'END.' ;

identifier = alphabetic character , [ { alphabetic character | digit } ] ;
number = [ "-" ] , digit , [ { digit } ] ;
string = "" , { all characters ? "" } , "" ;
assignment = identifier , ":", ( number | identifier | string ) ;
alphabetic character = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z" ;
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" ;
white space = ? white space characters ? ;
all characters = ? all visible characters ? ;
一个语法上正确的程序:
PROGRAM DEMO1
BEGIN
A0:=3;
B:=45;
H:=-100023;
C:=A;
D123:=B34A;
BABOON:=GIRAFFE;
TEXT:="Hello world!";
END.这个语言可以轻易的扩展上控制流，算术表达式和输入/输出指令。就可以开发出一个小的、可用的编程语言了。
```

由于C++语法规则的限制，Spirit改变了EBNF中的一部分操作符的使用方式，如：

星号重复符(*)由原来的后置改为前置

逗号串接符(,)由>>或&&代替

中括号可选功能(!表达式))改为(!表达式)

大括号重复功能({ 表达式}))由重复符(*表达式)替代

取消注释功能

取消特殊字符功能

同时Spirit又提供了大量的预置解析器加强了它的表达能力，因此可以把Spirit的语法看成是一种EBNF的变种。

入门

头文件：

```
#include <boost/spirit.hpp>
```

例一，解析一个浮点数

首先，要弄一个关于浮点数的EBNF规则

假设我们的浮点数形式是：[±]xxxx[.xxxx][Ex]，其中正负号可有可无，后面的幂可有可无，允许不带小数点则对应的EBNF规则是：

```
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
real = ["+"|"-"].digit, [{digit}], [".", digit, [{digit}]], ["E"|"e"], ["+"|"-"], digit, {digit}]]
那么对应在Spirit里的是什么样的呢？
!(ch_p('+')(ch_p('－'))>>+digit_p>>!(ch_p('.')>>+digit_p)>>!(ch_p('e')(ch_p('E'))>>!(ch_p('+')(ch_p('－'))>>+digit_p)
在Spirit中，用于匹配表达式的对象叫解析器，如这里的ch_p, digit_p以及由它们和操作符组成的整个或部分都可以称为解析器。
!符号代表其后的表达式是可选的，它代替了EBNF里的中括号功能。
ch_p()是一个Spirit预置的解析器生成函数，这个解析器用于匹配单个字符。
>>用于代替逗号顺序连接后面的解析器
+符号代表1次或多次重复
digit_p也是一个Spirit预置的解析器，它匹配数字字符。
这样，再看上面就好理解了：可选的+-号，接着是数字，再跟着是可选的小数点和数字，最后是可选的E跟一个可接+-号的数字
```

代码：

```
#include <iostream>
#include <boost/spirit.hpp>
using namespace std;
using namespace boost::spirit;
int main()
{
    parse_info<> r = parse("-12.33E-10",
        !(ch_p('+')(ch_p('－'))>>+digit_p>>!(ch_p('.')>>+digit_p)>>!(ch_p('e')(ch_p('E'))>>!(ch_p('+')(ch_p('－'))>>+digit_p)
    );
    cout << "parsed " << (r.full?"successful":"failed") << endl;
    return 0;
}
```

这就是Spirit,这个变种的EBNF语法直接就写在C++代码里就可以了，实际上它们是由一系列的简单解析器对象通过重载操作符后组合而成的复杂解析器。解析器重载的操作符也可以帮我们自动作一些转换工作，如上面的式子中ch_p('+')(ch_p('－'))就可以改成ch_p('+')|'-',只要左边或右边的数值其中之一是解析器，它就能自动和另一边的数值组合。

简化后如下：!(ch_p('+')|'-')>>+digit_p>>!(('.'>>+digit_p)>>(((ch_p('e')|'E')>>!(ch_p('+')|'-')>>+digit_p)

parse函数调用解析器来解析指定的字符串，它的原型是：

```
parse_info<charT const*> parse(字符串, 解析器);
parse_info<charT const*> parse(字符串, 解析器1, 解析器2);
```

第二个版本中的解析器2指出解析时可以忽略的一些字符，比如语句中的空格之类的。

另外，parse还有迭代器的版本

```
parse_info<IteratorT> parse(IteratorT first, IteratorT last, 解析器);
parse_info<IteratorT> parse(IteratorT first, IteratorT last, 解析器1, 解析器2);
IteratorT可以是任何迭代器类包括字符串指针，前面的这个两个版本其实只是简单地包装了一下这两个函数。
返回的parse_info<IteratorT>类（其中的IteratorT模板默认为char const*）包含了解析结果信息，里面的成员有：
IteratorT stop; //最后解析的位置
bool hit; //是否与整个解析器匹配
bool full; //是否与整个字符串匹配
std::size_t length; //解析器解析了多少个字符，注意，first+length不一定与stop相同
其实，Spirit已经帮我们准备好了很多解析器，比如上面我们写得要死的浮点数匹配，只要一个real_p就行了
parse_info<> r = parse("-12.33E-10",real_p);
```

Spirit预置的一些原始解析器，它们的名字都是以"xxxx_p"的形式出现。

字符解析器

- ch_p('X') 返回单字符解析器
 - range_p('a','z') 返回一个字符范围解析器，本例中匹配'a'..'z'
 - str_p("Hello World") 返回一个字符串解析器
 - chseq_p("ABCDEFGF") 返回一个字符序列解析器，它可以匹配"ABCDEFGF","A B C D E F G","AB CD EFG"等
 - anychar_p 匹配任何字符（包括'\0'）
 - alnum_p 匹配A-Z,a-z,0-9
 - alpha_p 匹配字母
 - blank_p 匹配空格和TAB
 - cntrl_p 匹配控制字符
 - digit_p 匹配数字字符
 - graph_p 匹配可显示字符(除空格，回车，TAB等)
 - lower_p 匹配小写字符
 - print_p 匹配可打印字符
 - punct_p 匹配标点符号
 - space_p 匹配空格，回车，换行，TAB
 - upper_p 匹配大写字符
 - xdigit_p 匹配十六进制数字字符串
 - eol_p 匹配行尾
 - nothing_p 不匹配任何字符，总是返回Fail(不匹配)
 - end_p 匹配结尾
- 字符解析器支持的操作符
- ~a 排除操作，如~ch_p('x')表示排除'x'字符
 - a|b 二选一操作，或称为联合，匹配a or b
 - a&b 交集，同时匹配a和b
 - a-b 差，匹配a但不匹配b
 - a^b 异或，匹配a 或 匹配b，但不能两者同时匹配
 - a>>b 序列连接，按顺序先匹配a，接下来的字符再匹配b
 - a&&b 同上(象C语言一样，有短路效果，若a不匹配，则b不会被执行)
 - a|b 连续或，按顺序先匹配a，接下来的字符匹配b(象C语言一样，有短路效果，若a已匹配，则b不会被执行)
 - *a 匹配0次或多次
 - +a 匹配1次或多次
 - !a 可选，匹配0或1次
 - a%b 列表，匹配a b a b a b a...，效果与 a >> *(b >> a)相同

整数解析器

Spirit给我们准备了两个整数解析器类，对应于有符号数和无符号数int_parser和uint_parser 它们都是模板类，定义如下：

```
template <
    typename T = int,
    int Radix = 10,
    unsigned MinDigits = 1,
    int MaxDigits = -1>
struct int_parser;
template <
    typename T = unsigned,
    int Radix = 10,
    unsigned MinDigits = 1,
    int MaxDigits = -1>
struct uint_parser;模板参数用法：
```

T为数字类型
Radix为进制形式
MinDigits为最小长度
MaxDigits为最大长度，如果是-1表示 unlimited

比如下面这个例子可以匹配象 1,234,567,890 这种形式的数字

```
uint_parser<unsigned, 10, 1, 3> uint3_p;    // 1..3 digits
uint_parser<unsigned, 10, 3, 3> uint3_3_p;    // exactly 3 digits
ts_num_p = (uint3_p >> *(',' >> uint3_3_p)); // our thousand separated number parserSpirit已预置的几个int_parser/uint_parser的特化版本：
```

```
int_p int_parser<int, 10, 1, -1> const
bin_p uint_parser<unsigned, 2, 1, -1> const
oct_p uint_parser<unsigned, 8, 1, -1> const
uint_p uint_parser<unsigned, 10, 1, -1> const
hex_p uint_parser<unsigned, 16, 1, -1> const
```

实数解析器

Spirit当然也会给我们准备实数解析器，定义如下：

```
template<
    typename T = double,
    typename RealPoliciesT = ureal_parser_policies<T> >
struct real_parser;
```

模板参数用法：
T表示实数类型
RealPoliciesT是一个策略类，目前不用深究，只要知道它决定了实数解析器的行为就行了。
已预置的实数解析器的特化版本：

```
ureal_p real_parser<double, ureal_parser_policies<double> > const
real_p real_parser<double, real_parser_policies<double> > const
strict_ureal_p real_parser<double, strict_ureal_parser_policies<double> > const
strict_real_p real_parser<double, strict_real_parser_policies<double> > const    real_p前面实例里已经见过，ureal_p是它的unsigned版本。strict_*则更严格地匹配实数（它不匹配整数
```

例二，解析实数序列

有了上面的知识，我们可以试试解析以逗号分隔的实数序列
字符串形式为"real,real,real,...real"
参考上面的一堆预置解析器，我们可以这样组合：

```
real_p >> *(',' >> real_p);
更简单点，我们可以使用%操作符 real_p%'
real_p%';
```

于是很简单地写下这样的代码：

```
{
    //用于解析的字符串
    const char *szNumberList = "12.4,1000,-1928,33,30";
    parse_info<> r = parse( szNumberList, real_p % ',' );
    cout << "parsed " << (r.full?"successful":"failed") << endl;
    cout << szNumberList << endl;
    //使用parse_info::stop确定最后解析的位置便于查错
    cout << string(r.stop - szNumberList, ' ') << '^' << endl;
}
```

解析成功！接下来我们就把里面的数字取出来，解析器重载了[]操作符，在这里可以放入函数或函数对象，放在这里面的函数或函数对象在Spirit里称之为Actor

对于real_p，它要求形式为void func(double v)的函数或函数对象，下面我们就来取出这些数字：

```
#include <iostream>
#include <boost/spirit.hpp>
```

```
using namespace std;
using namespace boost::spirit;
```

```
//定义函数作为解析器的Actor
void showreal(double v)
{
    cout << v << endl;
}
```

```
int main()
{
    //用于解析的字符串
```

```
const char *szNumberList = "12.4,1000,-1928,33,30";
//加入函数
parse_info<> r = parse( szNumberList, real_p[&showreal] % ' ');
cout << "parsed " << (r.full?"successful":"failed") << endl;
cout << szNumberList << endl;
//使用parse_info::stop确定最后解析的位置便于查错
cout << string(r.stop - szNumberList, ' ') << '^' << endl;
return 0;
}
再次运行，显示了一列数字了吧:)
再写一个函数对象版本的，这次把这列数字写到vector里
#include <iostream>
#include <vector>
#include <boost/spirit.hpp>
using namespace std;
using namespace boost::spirit;

int main()
{
    // pushreal函数对象，把数字放入vector中
    struct pushreal
    {
        void operator()(double v) const
        {
            m_vec.push_back(v);
        }
        pushreal(vector<double> &vec)
            :m_vec(vec){}
    private:
        vector<double> &m_vec;
    };

    vector<double> reallist;
    //用于解析的字符串
    const char *szNumberList = "12.4,1000,-1928,33,30";
    //这次用pushreal对象作为Actor
    parse_info<> r = parse( szNumberList, real_p[pushreal(reallist)] % ' ');
    cout << "parsed " << (r.full?"successful":"failed") << endl;
    cout << szNumberList << endl;
    //使用parse_info::stop确定最后解析的位置便于查错
    cout << string(r.stop - szNumberList, ' ') << '^' << endl;
    //显示结果
    copy(reallist.begin(),reallist.end(),ostream_iterator<double>(cout," "));
    return 0;
}
Spirit也提供了比偶的这个pushreal强得多的函数对象push_back_a
#include <iostream>
#include <vector>
#include <boost/spirit.hpp>
using namespace std;
using namespace boost::spirit;
int main()
{
    vector<double> reallist;
    //用于解析的字符串
    const char *szNumberList = "12.4,1000,-1928,33,30";
    //使用自带的push_back_a
    parse_info<> r = parse( szNumberList, real_p[push_back_a(reallist)] % ' ');
    cout << "parsed " << (r.full?"successful":"failed") << endl;
    cout << szNumberList << endl;
    //使用parse_info::stop确定最后解析的位置便于查错
    cout << string(r.stop - szNumberList, ' ') << '^' << endl;
    //显示结果
    copy(reallist.begin(),reallist.end(),ostream_iterator<double>(cout," "));
    return 0;
}
在上面的实数序列中，如果中间含有空格或TAB，这个解析就不能成功，这时可以使用parse函数的另一个重载版本：
parse_info<charT const*> parse(字符串, 解析器1, 解析器2);
//或
parse_info<IteratorT> parse(IteratorT first, IteratorT last, 解析器1, 解析器2);
其中的解析器2用于跳过其匹配的字符，我们要跳过空格，所以解析器2可以使用space_p:
parse_info<> r = parse( szNumberList,
    real_p[push_back_a(reallist)] % ',',
    space_p);
如果更进一步，我们甚至可以连逗号也跳过，直接取得一系列数字：
parse_info<> r = parse( szNumberList,
    *real_p[push_back_a(reallist)],
    space_p[ch_p(',')]);
除push_back_a外，Spirit还提供了不少有用的Actor（就是函数对象啦），如下
注：这里的ref是外部数据，就象上例中的reallist，value_ref是外部数值，value是解析出的数值
```

```
increment_a(ref)  自增 ++ref
decrement_a(ref) 自减 --ref
赋值操作
assign_a(ref)     赋值 ref = value
assign_a(ref, value_ref)  常量赋值 ref = value_ref
容器操作
push_back_a(ref)   ref.push_back(value)
push_back_a(ref, value_ref)  ref.push_back(value_ref)
push_front_a(ref)  ref.push_front(value)
push_front_a(ref, value_ref)  ref.push_front(value_ref)
clear_a(ref)       ref.clear()
关联容器操作(vt类型是typeof(ref)::value_type)
insert_key_a(ref, value_ref)  ref.insert(vt(value, value_ref))
insert_at_a(ref, key_ref, value_ref)  ref.insert(vt(key_ref,value_ref))
insert_at_a(ref, key_ref)    ref.insert(vt(key_ref,value))
assign_key_a(ref, value_ref)  ref[value] = value_ref
erase_a(ref)                 ref.erase(ref,value)
erase_a(ref, key_ref)        ref.erase(ref,key_ref)
其它操作
swap_a(a_ref, b_ref)  交换a_ref和b_ref
```

例三，四则运算

解析四则运算表达式，同样先要把EBNF规则写出来：

```
//实数或者是括号包围的子表达式
因子 = 实数 | '(' , 表达式 , ')';
//因子*因子或因子/因子，可连续乘除也可只是一个因子
乘除计算 = 因子,{('*',因子)}('/',因子));
//加减计算，与上面类似
表达式 = 乘除计算,{('+',乘除计算)}('-',乘除计算)); 这个定义已经隐含了优先级:
```

要计算表达式(加减计算),必然要先计算乘除计算;
要计算乘除计算,就要先计算因子;
要计算因子,要么得到一个数字,要么就要计算括号内的子表达式。
转成Spirit解析器组合:

```
rule<phrase_scanner_t> factor, term, exp;
factor = real_p | ('(' >> exp >> ')');
term  = factor >> *((('*' >> factor) | ('/' >> factor));
exp   = term >> *(('+' >> term) | ('-' >> term));
这里的rule是一个规则类，它可以作为所有解析器的占位符，定义如下：
```

```
template<
    typename ScannerT = scanner<>,
    typename ContextT = parser_context<>,
    typename TagT = parser_address_tag>
class rule;
其中的模板参数作用是：
ScannerT 扫描器策略类
它有两类工作模式，一种是字符模式，一种是语法模式，默认的scanner<>是工作于字符模式的。
ContextT 内容策略类
它决定了rule里的成员变量以及Actor的类型，稍后会有利用这个模板参数来加入自定义的成员变量的例子
TagT 标识策略类
每个rule都有一个id()方法，用于识别不同的rule，TagT就用于决定id()返回的数据（后面会讲到）。
这三个策略类可以不按顺序地输入，如
rule<parser_address_tag,parser_context<>,scanner<> >;
rule<parser_context<> >;
rule<scanner<>,parser_address_tag >;
是同一个类。
```

值得注意的是ScannerT,我们上面没有使用默认的scanner<>,而是使用了phrase_scanner_t,因为工作于字符模式的扫描器无法与parse的解析器2参数（跳过匹配字符，见上）一同工作，这样就无法解析含有空格的表达式，这可不完美，所以我们使用的工作于语法模式的phrase_scanner_t。

```
#include <iostream>
#include <vector>
#include <boost/spirit.hpp>
using namespace std;
using namespace boost::spirit;
int main()
{
    rule<phrase_scanner_t> factor, term, exp;
    factor = real_p | ('(' >> exp >> ')');
    term  = factor >> *((('*' >> factor) | ('/' >> factor));
    exp   = term >> *(('+' >> term) | ('-' >> term));

    const char *szExp = "1 + (2 * (3 / (4 + 5)))";
    parse_info<> r = parse( szExp , exp, space_p);
    cout << "parsed " << (r.full?"successful":"failed") << endl;

    return 0;
}
```

接下来，要得到这个四则表达式的计算结果，这才是我们要的，于是Spirit自带的lambda支持：phoenix登场！

头文件：

```
#include <boost/spirit/phoenix.hpp> phoenix提供和与Boost.Lambda类似的功能，它可以直接就地生成匿名函数对象，phoenix使用arg1,arg2,arg3...作为占位
```

符，Boost.Lambda则使用_1,_2,_3...，使用举例：

```
#include <iostream>
#include <vector>
#include <boost/spirit.hpp>
#include <boost/spirit/phoenix.hpp>
using namespace std;
using namespace boost::spirit;
using namespace phoenix;
int main()
{
    vector<int> vec(10);
    int i=0;
    //arg1 = var(i)++ 把i++赋值给vec里各单元
    for_each(vec.begin(),vec.end(),arg1 = var(i)++);
    //cout<<arg1<<endl 把vec各单元输出至cout
    for_each(vec.begin(),vec.end(),cout << arg1 << endl);
    return 0;
}
```

这样我们就可以利用phoenix提供的匿名函数对象作为Actor，同时利用Spirit提供的closure类为rule添加一个val成员变量存储计算结果（还记得rule的ContextT策略吗？）

```
#include <iostream>
#include <vector>
#include <boost/spirit.hpp>
#include <boost/spirit/phoenix.hpp>
using namespace std;
using namespace boost::spirit;
using namespace phoenix;
int main()
{
    //为rule准备一个val变量，类型为double
    //准确地说：是一个phoenix类(这里的member1)，它和其它phoenix类组成lambda表达式，在lambda中可以把它看成是一个double。
    struct calc_closure : boost::spirit::closure<calc_closure, double>
    {
        member1 val;
    };

    //定义ContextT策略为calc_closure::context_t
    rule<phrase_scanner_t, calc_closure::context_t> factor, term, exp;

    //直接使用phoenix的lambda表达式作为Actor
    factor = real_p[factor.val = arg1] | ('<' >> exp[factor.val = arg1] >> ')';
    term  = factor[term.val = arg1] >> *('<*' >> factor[term.val *= arg1]) | ('/' >> factor[term.val /= arg1]);
    exp  = term[exp.val = arg1] >> *('<+' >> term[exp.val += arg1]) | ('-' >> term[exp.val -= arg1]);

    const char *szExp = "1 + (2 * (3 / (4 + 5)))";
    double result;
    parse_info<> r = parse( szExp , exp[assign_a(result)], space_p);

    cout << szExp;
    if(r.full)
    {
        //成功，得到结果
        cout << " = " << result << endl;
    }
    else
    {
        //失败，显示错误位置
        cout << endl << string(r.stop - szExp, ' ') << '^' << endl;
    }
    return 0;
}
```

boost::spirit::closure的作用，它的使用方法是：

struct name : spirit::closure<name, type1, type2, type3,... typeN>

```
{
    member1 m_name1;
    member2 m_name2;
    member3 m_name3;
    ...
    memberN m_nameN;
};
```


一种类型对应一个member,使用name::context_t作为ContextT策略的rule就会含有N个相应的变量，而且这个rule的Actor将会接收到member1对应的数据。也可以用于语法类，如grammar<T, name::context_t>，关于语法类，后面章节将会提到。


注：默认最多到member3，要想使用更多数据，在包含Spirit头文件前预定义PHOENIX_LIMIT和BOOST_SPIRIT_CLOSURE_LIMIT，如


```
#define PHOENIX_LIMIT 10
#define BOOST_SPIRIT_CLOSURE_LIMIT 10
```


相关文章：
[? boost spirit库介绍...](#)


最近读者：


[登录后，您就出现在这里。](#)


[Sevancr7](#)


[sqk08](#)


[yejean90](#)

[一叶苇草](#)

[woodcock1017](#)

[五星国足](#)

[chungqing286](#)

[answer_or](#)

- 网友评论：
- 1

[匿名网友](#)

2009年09月30日 星期三 08:03 | [回复](#)

学习了，很好的文章，容易理解。Spirit真棒。多谢。

发表评论：

内 容：

Free.L

思想比知识更重要，成长比成功更重要

[主页](#) [博客](#) [相册](#) [个人档案](#) [好友](#)

查看文章

语法解析器 **Boost Spirit** 提高

2009年09月14日 星期一 12:36

提高

例四，使用自定义语法类框架

把所有的规则堆在一起不仅恶心，而且难以维护，Spirit提供了语法类grammar来集中管理。

grammar的定义如下：

```
template<
    typename DerivedT,
    typename ContextT = parser_context<> >
struct grammar;
```

DerivedT参数是反引用自身类型，如果用过WTL库的可能对这个比较熟悉，使用这种技术可以保持多态性的同时消除虚函数带来的性能开销。

ContextT参数就是内容策略类，在例三中提到过。

编写一个语法类框架的基本形式如下：

```
struct my_grammar : public grammar<my_grammar>
{
    template <typename ScannerT>
    struct definition
    {
        rule<ScannerT> r;
        definition(my_grammar const& self) { r = /*..define here..*/; }
        rule<ScannerT> const& start() const { return r; }
    };
};
```

它继承自grammar,模板参数DerivedT就是自身类型，ContextT可以使用默认的parser_context<>或者自己定义一个(比如例三中的closure)。

这个类内部必须要有一个definition类的定义，这个definition类的模板参数ScannerT由框架使用环境决定。它由两个重要方法：

start() const函数：它返回一个rule。使用my_grammar解析时，就从这个rule开始。

definition构造函数：这里是初始化rule的最好场所。它的self参数是整个my_grammar的实例引用，接下去你会发现这可是个很有用的东西。同时，很重要的一点：语法类本身也是一个解析器，它也能与其它解析器组合。

下面，我们把例三中的四则运算解析功能放到一个语法类中，然后再用这个语法类与其它解析器合作弄一个简单的赋值操作出来：

```
#include <iostream>
#include <boost/spirit.hpp>
#include <boost/spirit/phoenix.hpp>
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

//closure，为解析器提供存储策略，见例三
struct calc_closure : boost::spirit::closure<calc_closure, double>
{
    member1 val;
};
//四则运算语法类，它也使用了closure的内容策略
struct calculator : public grammar<calculator, calc_closure::context_t>
{
    //语法类重要成员:struct definition
    template <typename ScannerT>
    struct definition
    {
        // factor, term, exp的rule类型，同例三(ScannerT模板在使用时决定)
        typedef rule<ScannerT, calc_closure::context_t> rule_type;
        rule_type factor, term, exp;

        // 启动rule，在这个例子中，它也是递归的最顶层，负责把exp的最终结果赋值给框架本身。
        rule<ScannerT> rlStart;
        const rule<ScannerT>& start() const { return rlStart; }

        //definition的构造函数，self参数引用的是calculator类的实例
        definition(calculator const& self)
        {
            // 四则运算规则定义与例三相同
            factor = real_p[ factor.val = arg1 ] |
                ('(' >> exp[ factor.val = arg1 ] >> ')');
```



```
term = factor[term.val = arg1] >>
    *((('*' >> factor[term.val *= arg1]) |
      ('/' >> factor[term.val /= arg1]));
exp  = term[exp.val = arg1] >>
    *((('+' >> term[exp.val += arg1]) |
      ('-' >> term[exp.val -= arg1]));
//self.val=arg1也是phoenix的匿名函数：把exp的结果赋值给框架本身（self的作用）
rlStart = exp[self.val = arg1];
}
};
};
```

```
int main()
{
    string strVar; //变量名
    double result; //结果

    calculator calc;
    // 赋值语法：变量名 = 表达式
    rule<phrase_scanner_t> rEqu = (+alpha_p)[assign(strVar)] >> '=' >> calc[assign_a(result)];
    const char *szEqu = "value = 1 + (2 * (3 / (4 + 5)))";

    parse_info<> r = parse( szEqu , rEqu, space_p);

    if(r.full) //成功，得到结果
        cout << strVar << " = " << result << endl;
    else //失败，显示错误位置
        cout << endl << string(r.stop - szEqu, ' ') << '^' << endl;

    return 0;
}
// 如果没拼写出错的话，应该会显示出"value = 1.66667"。
```

例五，在四则运算表达式中使用变量

在例四中，我们可以解析"变量名 = 表达式"这种形式的语句。现在，我们再进一步，允许在表达式中使用变量，如value = value * pi + 5
那么，还是先从规则动手。
这里我把变量名的规则放松了一点，例四里变量名只能用字母，这里除了第一位是字母后面允许使用数字。于是变量名规则写成(alpha_p >> *(alnum_p))
变量代表的是一个数值，它和实数应该属于同一级别，所以我们把变量规则加入到factor规则里：

```
factor = real_p[factor.val = arg1] |
    // 在表达式中使用变量
    (alpha_p >> *(alnum_p))[/*这里写什么呢*/]
    ('' >> exp[factor.val = arg1] >> ');
那么，变量名对应的Actor写什么呢？具体地说是"factor.val = 什么"呢？
只要把变量名和它的数值一一对应起来，那么这里只要把此变量名对应的数值送给factor.val就行了，标准库里的map在这里用是再适合不过了。
为了把变量和它的数值放到map里，main里的rEqu规则我们也要小修改：
rule<phrase_scanner_t> rEqu =
    ((alpha_p >> *(alnum_p))[assign(strVar)] >>
    '=' >> calc[assign_a(result)]) [ insert_at_a(mapVar,strVar,result) ];
后面又加了一个Actor，把strVar及result放到map类型的mapVar中。
```

回到factor规则，试着把变量名规则的Actor写成[factor.val = getvalue(arg1, arg2)]，注意所有字符串规则的Actor都会有两个参数，它们是两个迭代器，分别指向起始位置和结束位置。所以这里使用了phoenix的arg1和arg2占位符。
这个getvalue我们把它写成一个函数，它从map中取出变量名对应的数值。

```
double getvalue(const char*first, const char*last)
{
    return mapVar[string(first,last)];
}
```

编译，出现错误，查phoenix手册，手册说想要在phoenix的表达式中使用函数，就得按它说的去做
它的要求是这样地：
1.先按如下形式做一个函数对象

```
struct func_impl
{
    //Param1等对就的是各个输入参数的类型
    template<typename Param1,typename Param2,...,typename ParamN>
        struct result{
            //定义输出参数的类型
            typedef returntype type;
        };
    //在这里该干啥干啥
    template<typename Param1,typename Param2,...,typename ParamN>
        returntype operator()(...)
        {
            ...
        }
};
// 2.使用phoenix::function类来包装第一步做的函数对象，这样才能和phoenix配合呢
```

另外，也可以直接用phoenix::bind把简单函数包装起来使用，不过这样虽然简单很多，在我们这个例子中却不便于封装于是作罢。
//适配phoenix的函数对象
struct getvalue_impl

```
{
template <typename ParamA,typename ParamB> //输入参数类型
struct result{
    typedef double type; //返回类型
};

//函数主体,其实这里的ParamA和ParamB都是char*
template <typename ParamA,typename ParamB>
double operator()(ParamA const& start,ParamB const& end) const
{
    //返回变量名对应的数值
    return m_mapVar[string(start,end)];
}

getvalue_impl(map<string,double> &mapVar)
:m_mapVar(mapVar){;}
private:
map<string,double> &m_mapVar;
};

// phoenix表达式中能接受的仿函数类型
const function<getvalue_impl> getValue = getValue_impl();
//适配phoenix的函数对象

#include <iostream>
#include <map>
#include <boost/spirit.hpp>
#include <boost/spirit/phoenix.hpp>
#include <boost/spirit/actor.hpp> // insert_at_a需要

using namespace std;
using namespace boost::spirit;
using namespace phoenix;

struct calc_closure : boost::spirit::closure<calc_closure, double>
{
    member1 val;
};

struct calculator : public grammar<calculator, calc_closure::context_t>
{
    template <typename ScannerT>
    struct definition
    {
        typedef rule<ScannerT, calc_closure::context_t> rule_type;
        rule_type factor, term, exp;

        rule<ScannerT> r1Start;
        const rule<ScannerT>& start() const { return r1Start; }

        definition(calculator const& self)
        {
            factor = real_p[factor.val = arg1] |
                // 允许在表达式中使用变量, 结果用calculator::m_getValue从map中取
                (alpha_p >> *(alnum_p)[ factor.val = self.m_getValue(arg1, arg2) ] |
                (' ' >> exp[factor.val = arg1] >> '));
            term = factor[term.val = arg1] >>
                * (('*' >> factor[term.val *= arg1]) |
                ('/' >> factor[term.val /= arg1]));
            exp = term[exp.val = arg1] >>
                * (('+' >> term[exp.val += arg1]) |
                ('-' >> term[exp.val -= arg1]));
            r1Start = exp[self.val = arg1];
        }
    };

    calculator(map<string,double> &mapVar)
    :m_getValue( getValue_impl(mapVar) ) //初始化, 把map传给m_getValue
    {}

    //适配phoenix的函数对象
    struct getValue_impl
    {
        template <typename ParamA,typename ParamB> //输入参数类型
        struct result{
            typedef double type; //返回类型
        };

        //函数主体,其实这里的ParamA和ParamB都是char*
        template <typename ParamA,typename ParamB>
```

```
double operator()(ParamA const& start,ParamB const& end) const
{
    //返回变量名对应的数值
    return m_mapVar[string(start,end)];
}

getvalue_impl(map<string,double> &mapVar)
:m_mapVar(mapVar){;}
private:
map<string,double> &m_mapVar;
};

// phoenix表达式中能接受的仿函数类型
const function<getvalue_impl> m_getValue;
};

//用来显示map中变量的值
void showPair(const pair<string,double> &val)
{
    cout << val.first << " = " << val.second << endl;
}

int main()
{
    string strVar;
    double result;
    //用来保存变量和对应的数值
    map<string,double> mapVar;

    //把map传给语法类，让解析器知道变量的值
    calculator calc(mapVar);

    // 变量名规则(alpha_p >> +(alnum_p)), 除第一位外后面可以跟数字。
    // 整个等式末尾加入insert_at_a的actor，匹配成功后把变量和数值存到map中。
    rule<phrase_scanner_t> rEqu =
    (
        (alpha_p >> *(alnum_p))[assign(strVar)] >>
        '=' >> calc[assign_a(result)] ) [ insert_at_a(mapVar,strVar,result) ];

    // 多行赋值语句，表达式用使用变量
    const char *szEqus[3] = {
        "PI = 3.1415926",
        "Rad = PI*2.0/3.0",
        "Deg = Rad*180/PI"};

    // 逐句解析
    for(int i=0; i<3; i++)    parse(szEqus[i], rEqu, space_p);

    // 显示每个变量的数值
    for_each(mapVar.begin(), mapVar.end(), showPair );

    return 0;
}
```

显示结果:

```
Deg = 120
PI = 2.14159
Rad = 2.0944
```

持续改进...

例五的代码有个小问题，因为设置了跳过空格，这对于语句来说是必须的，但却带来了一个副作用。试试把szEqus里的变量名中间加个空格，比如改成"R ad = P I*2.0/3.0"，这样的语句居然也能正确解析。那么怎样才能解析变量名时不许跳过空格，而解析语句的又允许跳过呢(搞双重标准)? 下面介绍的命令就可以帮上忙了。把所有的变量名规则（factor规则定义里有一个，rEqu规则定义里有一个）用lexeme_d包裹起来：

```
lexeme_d[(alpha_p >> *(alnum_p))]
现在不允许出现含有空格的变量名了。
```

下面介绍各种预置命令

使用形式: 命令[解析器表达式]

lexeme_d

不跳过空白字符，当工作于语法级时，解析器会忽略空白字符，lexeme_d使其临时工作于字符级
如整数定义应该是: integer = lexeme_d[!(ch_p('+') | '-') >> +digit], 这样可以防止"1 2 345"被解析为"12345"

as_lower_d

忽略大小写，解析器默认是大小写敏感的，如果要解析象PASCAL一样的大小写不敏感的语法，使用r = as_lower_d["begin"];(注，里面的参数都得小写)

no_actions_d

停止触发Actor

longest_d

尝试最长匹配

如number = integer | real;用它匹配123.456时，integer会匹配123直到遇到小数点结束，使用number=longest_d[integer | real];可以避免这个问题。

```
shortest_d
与longest_d相反

limit_d
定义范围,用法limit_d(min, max)[expression]
如
uint_parser<int, 10, 2, 2> uint2_p;
r = lexeme_d
[
    limit_d(0u, 23u)[uint2_p] >> ':' // Hours 00..23
    >> limit_d(0u, 59u)[uint2_p] >> ':' // Minutes 00..59
    >> limit_d(0u, 59u)[uint2_p] // Seconds 00..59
];
min_limit_d/max_limit_d
定义最小/最大值, 用法:min_limit_d(min)[expression]
```

例七、牛叉型解析器

```
相对于Spirit预置的一些简单解析器， 它也提供了很多功能更强大的“牛叉型”解析器。现介绍如下：
f_ch_p
语法： f_ch_p(ChGenT chgen)
作用： 和ch_p类似， 它解析的字符由chgen的返回值决定， chgen是一个类型为"CharT func()"的函数（或函数对象）
例如： char X(){return 'X';} f_ch_p(&X);
```

```
f_range_p
语法： f_range_p(ChGenAT first, ChGenBT last)
作用： 和range_p类似， 它由first和last两个函数（或函数对象）的返回值决定解析的字符范围。
```

```
f_chseq_p
语法： f_chseq_p(IterGenAT first, IterGenBT last)
作用： 和chseq_p类似， 同样由first和last两个函数（或函数对象）的返回值决定起始和终止迭代器。
```

```
f_str_p
语法： f_str_p(IterGenAT first, IterGenBT last)
作用： 和str_p类似， 参数同f_chseq_p
```

```
if_p
语法： if_p(condition)[then-parser].else_p[else-parser]， 其中else_p可以不要
作用： 如果condition成立，就使用then-parser, 否则用else-parset
例如： if_p("0x")[hex_p].else_p[uint_p]
```

```
for_p
语法： for_p(init, condition, step)[body-parser]
作用： init和step是一个无参数的函数或函数对象， 各参数与for的作用类似（先init,再检查condition,有效则执行body-parser及step,再检查condition...）
例如： for_p(var(i)=0, var(i) < 10, ++var(i) ) [ int_p[ var(sum) += arg1 ] ]
```

```
while_p, do_p
语法： while_p(condition)[body-parser] 及 do_p[body-parser].while_p(condition)
作用： 条件循环，直接condition不成立为止。
```

```
select_p, select_fail_p
语法： select_p(parser_a , parser_b /* ... */ , parser_n);
作用： 从左到右按顺序测试各解析器， 并得到匹配的解析器的序号（0表示匹配parser_a,1匹配parser_b...）
例如： 见switch_p例
```

```
switch_p
语法： switch_p(value)[case_p<value_a>(parser_a),case_p<value_b>(parser_b),...,default_p(parser_def)]
作用： 按value的值选择解析器
例如： 下例中匹配的形式为:字符a后是整数， b后是个逗号,c后跟着"bcd"， d后什么也没有。
```

```
int choice = -1;
rule<> rule_select =
    select_fail_p('a', 'b', 'c', 'd')[assign_a(choice)]
    >> switch_p(var(choice))
    [
        case_p<0>(int_p),
        case_p<1>(ch_p(',')),
        case_p<2>(str_p("bcd")),
        default_p
    ];
```

```
c_escape_ch_p, lex_escape_ch_p
语法： c_escape_ch_p
作用： 和ch_p类似， 其牛叉的地方在于能解析C语言里的转义字符： \b, \t, \f, \, \", \', \xHH, \OOO
例如： confix_p("", *c_escape_ch_p, "")
```

```
repeat_p
语法、作用：
repeat_p (n) [p] 重复n次执行解析器p
repeat_p (n1, n2) [p] 重复n1到n2次解析器p
repeat_p (n, more) [p] 至少重复n次解析
```

例如：检验是否是有效的文件名

```
valid_fname_chars = /*.*;/;
filename = repeat_p(1, 255)[valid_fname_chars];
```

confix_p
语法：confix_p(open,expr,close)
作用：解析独立元素，如C语言里的字符串，注释等，相当于open >> (expr - close) >> close
例如：解析C注释confix_p("//*", *anychar_p, "**/")

comment_p,comment_nest_p
语法：comment_p(open,close),如果close不指定，默认为回车
作用：confix_p的辅助解析器，comment_p遇到第一个close时即返回，而comment_nest_p要open/close对匹配才返回。
例如：
comment_p("/"/) C++ 风格注释
comment_nest_p('{','}')comment_nest_p("(","") pascal风格注释

list_p
语法：list_p(paser,delimiter)
作用：匹配以delimiter作为分隔符的列表

regex_p
语法：regex_p("正则表达式")
作用：使用正则表达式来匹配字符串(强强联手啊~~啥也不说了)

symbols类
定义：template

```
<
    typename T = int,
    typename CharT = char,
    typename SetT = impl::tst<T, CharT>
>
class symbols;
    初始化方式:
symbols<> sym;
sym = "pineapple", "orange", "banana", "apple", "mango";
sym.add("hello", 1)("crazy", 2)("world", 3);
    作用：匹配字符串(CharT*)返回对应的整数(T)
    例如：
struct Show{
    void operator()( int n ) const
    {
        cout << n;
    }
};
symbols<> sym;
sym.add("零",0)
("一",1) ("二",2) ("三",3)
("四",4) ("五",5) ("六",6)
("七",7)("八",8)("九",9);
parse("二零零八",*(sym[Show()]));
```

functor_parser
作用：可以方便地用它来创建一个解析器
例如：见下例

演示怎样自己写一个解析器，解析一个整数

```
struct number_parser
{
    typedef int result_t; //定义解析器结果类型

    //参数是:扫描器，结果
    template <typename ScannerT>
    std::ptrdiff_t operator()(ScannerT const& scan, result_t& result) const
    {
        if (scan.at_end()) //如果结果或出错，返回-1
            return -1;

        char ch = *scan;
        if (ch < '0' || ch > '9')
            return -1;

        result = 0;
        std::ptrdiff_t len = 0;

        do //解析字符串，得到结果
        {
            result = result*10 + int(ch - '0');
            ++len;
            ++scan;
        } while (!scan.at_end() && (ch = *scan, ch >= '0' && ch <= '9'));
```

```
        return len; //返回解析的字符串长度
    }
};
//用functor_parser包装成解析器
functor_parser<number_parser> number_parser_p;
```

类别: [boost & stl](#) | [转贴](#) [+](#) | [添加到收藏](#) | [分享到i贴吧](#) | [浏览\(253\)](#) | [评论 \(0\)](#)
上一篇: 语法解析器 [Boost Spirit](#) 入门 下一篇: 语法解析器 [Boost Spirit](#) 提高(...

相关文章:
[? boost spirit库介绍...](#)

最近读者:

[登录后，您就出现在这里。](#)

[Sevancr7](#)

[yejean90](#)

[一叶苇草](#)

[woodcock1017](#)

[apacho](#)

[chunqing286](#)

[sqk08](#)

[476539734](#)

网友评论:

发表评论:

内 容:

Free.L

思想比知识更重要，成长比成功更重要

[主页](#) [博客](#) [相册](#) [个人档案](#) [好友](#)

[查看文章](#)

语法解析器 **Boost Spirit** 提高(续)

2009年09月14日 星期一 12:43

例八，抽象语法树(abstract syntax tree，简称AST)
上面的例子都是就地解析。在比较大型的语法解析中，一种更通用的方式是先产生抽象语法树再遍历它来做解析工作。比如著名的GCC，观察它的源代码就可以发现，它解析源代码时首先生成AST再开始编译。其它编译器因为看不到源码不好说，想来也该是如此吧。
Spirit也支持生成抽象语法树的功能（不过用它来解析C++代码可就不太合适了，Spirit针对的是轻量的小型脚本）

头文件

```
#include <boost/spirit/include/classic_ast.hpp>
使用AST和之前的解析步骤很相似，一个重要的区别是所有的子规则都应该是字符串形式的，也就是说real_p.int_p之类的帮不上忙了，我们得自力更生。
我们在例一中使用过的浮点数解析器这次可以派上用场了。
下面的例子参考了例四中的解析器规则：
```

```
#include <iostream>
#include <boost/spirit.hpp>
#include <boost/spirit/include/classic_ast.hpp>

using namespace std;
using namespace boost::spirit;

struct calculator : public grammar<calculator>
{
    template <typename ScannerT>
    struct definition
    {
        typedef rule<ScannerT> rule_type;
        rule_type factor, term, exp, str_real_p;

        const rule_type& start() const { return exp; }

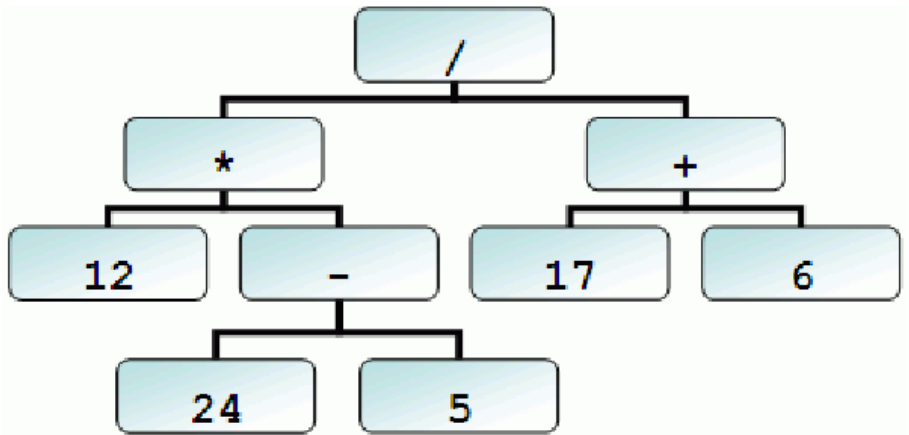
        definition(calculator const& self)
        {
            str_real_p = leaf_node_d[
                !(ch_p('+')|'-')>>+digit_p>>
                !('.'>>+digit_p)>>!((ch_p('e')|'E') >>
                    !(ch_p('+')|'-')>>+digit_p)
            ];
            factor = str_real_p | inner_node_d[('(' >> exp >> ')')];
            term = factor >> *((root_node_d[ch_p('*')] >> factor)
                | (root_node_d[ch_p('/')] >> factor));
            exp = term >> *((root_node_d[ch_p('+')] >> term)
                | (root_node_d[ch_p('-')] >> term));
        }
    };
};

//显示AST的结构.Indent是缩进宽度
typedef tree_match<char const*>::container_t container_t;
void showTree(const container_t& con, int Indent)
{
    for(container_t::const_iterator itr=con.begin(); itr!=con.end(); ++itr)
    {
        //tree_node<T>: value, children
        //显示当前值
        cout << string(Indent*4, ' ') << "|--(" <<
            string(itr->value.begin(), itr->value.end()) << ')' << endl;
        //显示子节点
        showTree(itr->children, Indent+1);
    }
}

int main()
{
    calculator calc;
```

```
const char *szExq = "12 * (24 - 15) / (17 + 6)";
tree_parse_info<> info = ast_parse(szExq, calc, space_p);
showTree(info.trees, 0);
return 0;
}
```

这个程序可以显示出整个AST的结构，比如例子中的“12 * (24 - 15) / (17 + 6)”，解析结果(用图片美化了一下):



这个代码和之前的代码主要区别是多了几个xxxx_node_d形式的命令，以及使用ast_parse函数来解析。

tree_parse_info类型
ast_parse的参数与parse相同，主要区别就在于它的返回值不是parse_info而是tree_parse_info。
tree_parse_info的成员有：

```
IteratorT stop;
bool match;
bool full;
std::size_t length;
typename tree_match<IteratorT, NodeFactoryT, T>::container_t trees;
```

前四个和parse_info相同，多出来的trees是一个含有tree_node<T>的容器（默认的容器是std::vector，如果预定义了BOOST_SPIRIT_USE_LIST_FOR_TREES，就会使用std::list）

tree_node<T>有两个重要的成员：
children: 子节点，与tree_parse_info里的trees类型相同：std::vector< tree_node<T> >(或std::list<...>)
value: 数据，类型为模板T,这个参数默认类型是node_val_data<IteratorT, ValueT>

整个AST就是由tree_node的数据value以及子节点children组成的。（参考例子中showTree的代码就可以看出）

node_val_data<IteratorT, ValueT>的模板参数IteratorT默认是const char*, ValueT是nil_t（空数据，定义为struct nil_t {}:）。

在这个类内部维护着一个vector(或list)，它保存着解析出来的脚本字符串，比如上面例子中的"12","*","24"等。node_val_data向外提供的重要方法有：

begin()/end(): 直接返回内部vector(或list)的begin()和end()
is_root()/is_root(bool): 取得/设置对应节点的root状态（由root_node_d命令设置）
value()/value(const ValueT&)取得/设置用户自定义数值（默认的nil_t没法带数据，必须通过指定NodeFactoryT来改变ValueT类型，马上会讲到）
id()/id(parser_id): 取得/设置解析此节点的解析器id(还记得rule的TagT策略吗，下面还会讲到)

它的value()方法可以设置和取得自定义数据，不过默认的nil_t却是个空结构，根本不能使用。这时我们可以通过指定“工厂类”来改变ValueT的类型，方法如下(假设使用double):

```
typedef node_val_data_factory<double> factory_t;
my_grammar gram;
my_skip_grammar skip;
tree_parse_info<iterator_t, factory_t> i =
    ast_parse<factory_t>(first, last, gram, skip);
```

rule有一个id()方法可以返回一个parser_id类型的标记,用它可以区分各个不同的rule,它返回什么值由TagT模板参数决定，默认的parser_address_tag返回的是rule的内存地址。

我们可以用其它参数代替它以实现更适用的标记，Spirit已准备好的TagT策略有：

parser_tag<N>,它接收一个整数，如

```
rule<parser_tag<123>> my_rule;
assign(rule.id().to_long() == 123);
```

dynamic_parser_tag, 它给rule加入了set_id(int)的能力，如：

```
rule<dynamic_parser_tag> my_dynrule;
my_dynrule.set_id(1234); // set my_dynrule's id to 1234
```

利用这些TagT策略再和node_val_data里的id()相比较就能知道这个数据是由哪个解析器解析的。

下面介绍Spirit为AST而引入的几个命令：

leaf_node_d
由leaf_node_d命令包裹的规则将被视为一个整体，它还由另一个名字token_node_d。
尝试把上例中的leaf_node_d命令去掉，再看解析结果：所有的数字都被折成了一个字节。

inner_node_d
这个命令会忽略第一个子规则和最后一个子规则，只取中间部分。
把上例中的inner_node_d去掉，那么括号也被参与解析。

`root_node_d`
这个命令对于AST至关重要，由`root_node_d`命令包裹的节点将成为同一规则中其它节点的父节点。它的工作方式如下：

假设A是前一节点
B是新产生的节点
如果B是根节点
 A成为B的第一个子节点
否则，如果A是根节点而B不是，那么
 B成为A的最后一个子节点
其它情况
 A和B处于同一级
比如这个例子中的“12 * (24 - 15) / (17 + 6)”
对于解析器解析顺序是：

```
exp = term
term = 12{factor} *{root} (24 - 15){exp} /{root} (17 + 6){exp}
...
```

首先解析12, 然后是*, 这时发现*是root, 于是12成为*的第一个子节点
接着解析(24 - 15)这个exp, 同理, 24成为-的第一个子节点, 然后是15, 它不是root, 而前一个是, 于是15成为-的最后一个子节点。
因为(24 - 15)这个exp不是root, 同样成为了*的最后一个子节点。
再解析/, 是root, 于是把前一节点 (是*哦, 因为其它的都成了*的子节点) 变成了它的首个子节点。
最后解析(17+6)这个exp, 最终成为了/的最后一个子节点。

`no_node_d`
忽略由它包裹的规则，比如例子中的：
`factor = str_real_p | inner_node_d[(' >> exp >> ')];`
也可以这样表示：
`factor = str_real_p | (no_node_d[ch_p('(')] >> exp >> no_node_d[ch_p(')')]);`

`infix_node_d`
这个命令会删除其规则匹配出的所有节点中偶数位置上的节点，比如：
`rule_t intlist = infix_node_d[integer >> *('(' >> integer)];`
这条规则将只产生整数数组，偶数位置上的逗号将被删除。

`discard_first_node_d/discard_last_node_d`
忽略第一个/最后一个子规则(半个`inner_node_d`功能)

我们的Spirit学习先到这里，这些只不过是Spirit里的冰山一角，要发挥Spirit的强大威力，还得继续前进...
在<boost>/libs/spirit/example里有不少“很好很强大”的例子，比如：小型的C语言解释器，XML解释器等，大家有兴趣可以去研究研究。

类别: [boost & stl](#) | [转贴](#) | [添加到收藏](#) | [分享到i贴吧](#) | [浏览\(360\)](#) | [评论 \(0\)](#)

[上一篇: 语法解析器 Boost Spirit 提高](#) [下一篇: 浅谈并行编程中的任务分解模式](#)

相关文章：
[? boost spirit库介绍...](#)

最近读者:



登录后，您就出现在这里。



Sevancr7



sqk08



攻成天下



yejean90



sgjianlin



一叶苇草



woodcock10



五星国足

17

网友评论:

发表评论:

内 容:

