

个人资料



laibach0304



访问：30372次  
积分：562  
等级：BLOG > 3  
排名：千里之外  
  
原创：21篇 转载：15篇  
译文：0篇 评论：11条

文章搜索

文章分类

- C# (3)
- C++ (23)
- C++0x (6)
- Other (2)
- Reading (3)

文章存档

- 2007年10月 (2)
- 2007年09月 (8)
- 2007年08月 (22)
- 2007年07月 (2)
- 2007年02月 (3)

阅读排行

- C++ Meta Programming (1467)
- boost.pool源码整理和使用 (1123)
- 《C++0x漫谈》系列之： (934)
- C++ Meta Programming (908)
- boost.shared\_ptr源码整理 (906)
- C++ Meta Programming (898)
- 智能指针的标准之争：Boost (872)

【活动】Python创意编程活动开始啦!!! CSDN日报20170428 ——《你的开发为何如此低效?》 深入浅出，带你学习 Unity

## C++ Meta Programming 和 Boost MPL(4)

标签：c++ vector struct iterator preprocessor lambda

2007-08-30 23:07 909人阅读 评论(0) 收藏 举报

分类：C++ (22)

目录(?)

本系列全部转载自kuibyshev.bokee.com

### 1. Boost中的MPL库分析

MPL是由David Abrahams和Aleksey Gurtovoy为方便模板元编程而开发的库，2003年被Boost吸纳为其中的一员，此后又历经一些大幅度修改，目前已经相当完善，其最新版本于2004年11月发布。MPL的出现是C++模板元编程发展中的一大创举，它提供了一个通用、高层次的编程框架，其中包括了序列（Sequence）、迭代器（Iterator）、算法（Algorithm）、元函数（Metafunction）等多种组件，具有高度的可重用性，不但提高了模板元编程的效率，而且使模板元编程的应用范围得到相当的扩展。

#### 1.1. MPL的组织架构

一个库的组织形式有时候甚至比它的功能还重要。MPL的作者聪明地借鉴了已经取得巨大成功的STL，在MPL中保留了许多STL的概念，对函数式的编程方式进行了精巧的包装，使得任何熟悉STL的程序员都可以轻易地理解MPL的使用方法。像STL一样，MPL有一个完整的概念体系，对组件作了精心的划分，组件之间相对独立，接口具有通用性，因此将组件之间的依存度和耦合性降低到最小的限度。

STL和MPL的组件概念对照如下：

STL概念	MPL对应概念
容器（Container）	序列（Sequence）
算法（Algorithm）	算法（Algorithm）
迭代器（Iterator）	迭代器（Iterator）
仿函数（Functor）	元函数类（Metafunction）
配接器（Adaptor）	有View、Inserter Iterator和相当于仿函数配接器的Binding元函数
配置器（Allocator）	无此概念
标准中没有定义	宏（Macro）

#### 1.2. MPL对其他库的依赖

MPL是一个高层次的库，它的地位和编译期执行的特殊性决定了它需要一些特殊的辅助设施，并对其他库有所依赖。

##### 1.2.1. Boost的Preprocessor库

Preprocessor库是一个基于宏的元编程库[7]。预处理器的作用发生在编译以前，所以它比MPL所处的地位还要高端，能够真正实现代码生成。它的典型功能是迭代或者枚举相似的代码段，减少重复而易写错的代码段。MPL中不少代码是近似的，比如在vector的原始代码中，就需要定义n个

vector { ... }

其中i从1迭代到n。为了减少重复劳动，MPL的源代码大量使用自定义和Preprocessor库的宏对重复或具有递推性的内容进行迭代。不过，这也导致源代码难以阅读。比如上面一段展开后的源代码首先是定义在vector/aux\_/numbered.cpp的：

- [一个STL风格的动态二维](#) (856)
- [C++ Meta Programming](#) (855)
- [C# - Ini文件类](#) (639)

- 评论排行
- [为什么C++](#) (2)
- [智能指针的标准之争：Boost vs. STL](#) (2)
- [一个STL风格的动态二维](#) (2)
- [你应当如何学习C++\(以及编程\)](#) (2)
- [boost.shared\\_ptr源码整理](#) (1)
- [我看国内的C++教育以及我的建议](#) (1)
- [C++ Meta Programming](#) (1)
- [《C++0x漫谈》系列之：智能指针](#) (0)
- [《C++0x漫谈》系列之：右尖括号](#) (0)
- [C#中const与readonly字段的区别](#) (0)

- 推荐文章
- [\\* CSDN日报20170429 ——《程序修行从“拔刀术”到“万剑诀”》](#)
- [\\* 抓取网易云音乐歌曲热门评论生成词云](#)
- [\\* Android NDK开发之从环境搭建到Demo级十步流](#)
- [\\* 个人的中小型项目前端架构浅谈](#)
- [\\* 基于卷积神经网络\(CNN\)的中文垃圾邮件检测](#)
- [\\* 四无年轻人如何逆袭](#)

- 最新评论
- [C++ Meta Programming 和 Boost code\\_pipeline: 大哥，这个代码错了 template<T> unsigned n &gt; struct fa...](#)
- [为什么C++ don: 总觉得刘某就是针对linus在说。](#)
- [为什么C++ don: C++肯定还是有用的，这点毫无疑问](#)
- [我看国内的C++教育以及我的建议 coolage31: 多态与虚拟是其核心，这也是所有面向对象的核心。你的心得总结真的写的很好~但这两者也真的很难吃透现...](#)
- [一个STL风格的动态二维数组 laibach0304: 恩，没错 完全原创](#)
- [一个STL风格的动态二维数组 jxlczjp77: 老兄这时你自己写的吗？看着恐怖，这么长。^\\_^](#)
- [智能指针的标准之争：Boost vs. laibach0304: 呵呵，欢迎光临。我也准备尝试新的东西了，现在还在观望中，但是C++我还是会坚持下去的。不提马光...](#)
- [智能指针的标准之争：Boost vs. guokeno1: 我来也！不错，老兄你研究的东西估计马光志抓狂也弄明白了..... 我现在不怎么用C++了，...](#)
- [你应当如何学习C++\(以及编程\) laibach0304: 呵呵，转载的文章，没仔细排版了，见谅](#)
- [你应当如何学习C++\(以及编程\) BillEasy: 看起来太累，字体大小格式应该整理下。](#)

```
// Preprocessor的宏，得到目前属于第几次迭代
#define i_ BOOST_PP_FRAME_ITERATION(1)
...
template<
    //Preprocessor的宏，枚举参数列表
    BOOST_PP_ENUM_PARAMS(i_, typename T)
>
// Preprocessor的宏，拼合vector和当前次数
struct BOOST_PP_CAT(vector,i_)
{ ... }
```

然后为了迭代n个上面的类模板，另一个文件则需要重复include这个文件，利用Preprocessor的文件迭代能力可以这样写：

```
// Preprocessor的宏，其中第一个参数3表示后面的参数组有3个
//元素，0和10表示迭代的范围是从0到10，最后一个参数是文件
//迭代的文件名
# define BOOST_PP_ITERATION_PARAMS_1 /
(3,(0, 10, ))
// Preprocessor的宏，要求按照上面的指定的参数进行递归
# include BOOST_PP_ITERATE()
```

尽管如此，宏还是必需的，它不但避免了重复编写递推式的代码（比如在上述的vector类模板中，n可达50之大，如果完全手写确实是浪费时间），而且还有效控制了代码的生成（比如只需要通过定义迭代次数，即可控制实际生成的类模板个数）。实际上，在使用vector（或其他组件）时，通常我们并不需要每次编译都把这些代码重新生成一次，MPL的作者已经充分考虑到编译效率的问题，所以在MPL的代码中，为每个流行的编译器都建立了一个Processed目录，里面存放着针对编译器特点展开了的代码。仅当定义了BOOST\_MPL\_CFG\_NO\_PREPROCESSED\_HEADERS时才会强制MPL重新用宏来生成代码。

MPL的作者指出，无论喜欢还是不喜欢，目前宏必须在MPL中扮演着这个不可替代的角色。

### 1.2.2. Boost的Type Traits库

Type Traits库[9]用于验证传递的参数或参数之间是否符合一定的条件，比如可以判定两个参数是否有继承关系、是否可转换等。

### 1.2.3. Boost的Static Assert库

Static Assert库[8]用于编译时断言，用法类似于C中常用的断言assert()。如果参数经编译时的静态计算为true，则代码能通过编译，不会有任何效果，反之，则会编译出错，并且在使编译信息里面包含有“STATIC\_ASSERTION\_FAILURE”的字样。Static Assert的底层是接受一个bool参数的模板STATIC\_ASSERTION\_FAILURE，它对true定义一个有成员的特化模板，对false的情况则只有一个特化的声明（无定义）。其接口是一个宏，它产生的代码是sizeof(STATIC\_ASSERTION\_FAILURE< ... >)，显然当参数的实际结果为false时，编译器无法判断STATIC\_ASSERTION\_FAILURE的长度，因为它尚未定义。因为MPL是只在编译时生效的库，用Static Assert来调试程序是非常合适的，它往往与Type Traits库搭配使用。

### 1.2.4. Boost的Config库

像STL一样，由于编译器对标准支持不同，为了使程序库具有移植性，最好是针对环境进行预先的设置。对于MPL这种先锋性的库来说，编译器问题更加让库作者相当头痛。借助于对环境的侦查，可以对预先发现的问题，比如模板的局部特化能力、已知的一些编译4)。

## 1.3. MPL中的序列

### 1.3.1. MPL序列概述

序列是MPL中的数据结构的统称，是MPL中处于中心地位的组件，其地位相当于STL中的容器。MPL对序列的性质进行了细致的划分：

性质	含义 / 主要模型
前向序列Forward Sequence	begin和end元函数能够界定其头尾范围的类型序列 / MPL中所有序列

关闭

其他人的blog

刘未鹏|C++的罗浮宫

马维达

荣耀

yacht

双向序列Bidirectional Sequence	迭代器属于双向迭代器的前向序列 / vector , range_c
随机访问序列Random Access Sequence	迭代器属于随机访问迭代器的双向序列 / vector , range_c
可扩展序列Extensible Sequence	允许插入和删除元素的序列 / vector , list
前可扩展序列Front Extensible Sequence	允许在前端插入和删除元素的可扩展序列 / vector , list
后可扩展序列Back Extensible Sequence	允许在后端插入和删除元素的可扩展序列 / vector , list
关联序列Associative Sequence	可以用key值来检索元素的前向序列 / set , map
可扩展关联序列Extensible Associative Sequence	允许插入和删除元素的关联序列 / set , map
整型序列包装器Integral Sequence Wrapper	存放一系列整型常量类 ( Integral Constant ) 的类模板 / vector_c , list_c , set_c
不定序列Variadic Sequence	可以用给定元素个数或用不指定元素个数的形式来定义的序列 / vector , list , map

部分概念在现阶段的MPL版本中其实存在着一些冗余，但这种以概念驱动的程序库却是很清晰的：每一种概念的背后都指明了它所支持的操作。

1.3.2. vector和deque

(1) 概述

MPL中最简单和最常用的序列就是vector。而deque在目前版本的MPL中相当于vector。vector的实质十分类似于前面示例的类型“数组”，逻辑上是连续线性的，由于它属于不定序列，使用时既可以指定长度，以vector<n>来定义，也可以直接用vector来定义。注意n不能超过宏BOOST\_MPL\_LIMIT\_VECTOR\_SIZE的定义，目前MPL的默认值是20。vector的特点是支持尾端常数时间的插入和删除操作以及中段和前端线性时间的插入和删除操作。

(2) 操作

vector支持的操作无论在命名还是逻辑上基本都与STL 一致，但有一个重大区别，STL的操作函数定义在类的内部，但是限于模板元编程的特殊性，MPL的这些元函数在容器外定义。下表列出它们的使用法：

begin::type	返回一个迭代器指向v的头部
end::type	返回一个迭代器指向v的尾部
size::type	返回一个v的大小
empty::type	当且仅当v为空时返回一个整型常量类，其值为true
front::type	返回v的第一个元素
back::type	返回v的最后一个元素
at::type	返回v的第n个元素
insert::type	返回一个新的vector使其定义为[begin::type, pos), x, [pos, end::type)
insert_range::type	返回一个新的vector使其定义为[begin::type, pos), [begin::type, end::type) [pos, end::type)
erase::type	返回一个新的vector使其定义为[begin::type, pos), [next::type, end::type)
erase::type	返回一个新的vector使其定义为[begin::type, pos), [last, end::type)
clear::type	返回一个空的vector
push_back::type	返回一个新的vector使其定义为[begin::type, end::type), x
pop_back::type	返回一个新的vector使其定义为[begin::type, prior< end::type >::type)

关闭

push_front::type	返回一个新的vector使其定义为[begin::type, end::type), x
pop_front::type	返回一个新的vector使其定义为[next< begin::type >::type, end::type)

### (3) 源代码分析

MPL的源代码有着比较复杂的脉络，主要原因是为了保持移植性，需要针对不同的编译器问题进行规避。比如vector的底层就有三个不同的版本，第一个专门针对不支持模板局部特化的编译器，第二个用于基于类型的序列，第三个是普通版本。在预处理时会根据情况确定使用哪一个版本。它们之间的差异是什么呢？vector0的实现代码中把它们放在了一起，正好可以说明其区别：

```
template< typename Dummy = na > struct vector0;
template<> struct vector0
{
    #if defined(BOOST_MPL_CFG_TYPEOF_BASED_SEQUENCES)
        typedef aux::vector_tag tag;
        typedef vector0 type;
        typedef long_<32768> lower_bound_;
        typedef lower_bound_ upper_bound_;
        typedef long_<0> size;
        static aux::type_wrapper item(...);
    #else
        typedef aux::vector_tag<0> tag;
        typedef vector0 type;
        typedef void_ item0;

        typedef v_iter<VECTOR0< />,0> begin;
        typedef v_iter<VECTOR0< />,0> end;
    #endif
};
```

定义的上半部分是基于类型的版本，下半部分则用于另外两个版本。MPL的参考手册没有说明vector的底层是实现的原理，看起来两种实现之间的差异比较大，其中最重要的差别是vector\_tag的用法。vector\_tag同样是一个底层的定义，作用应该是传递给各类算法，以区别不同的序列类型。tag的定义同样有两种：

```
#if defined(BOOST_MPL_CFG_TYPEOF_BASED_SEQUENCES)
    struct vector_tag;
#else
    template< long N > struct vector_tag;
#endif
```

大概基于类型的版本可以不必实例化一个vector\_tag，性能上更优越。从MPL的config配置情况来看，似乎默认只使用基于类型的序列，也就是序列会以v\_item作为基类。限于篇幅，这里仅分析基于类型的vector，下文有类似情况时也做同样的处理，不一一展开了。

前面已经指出，vector是一个不定序列，这类序列可以不必指定参数的个数直接使用。C++模板支持不定个数的参数表吗？当然不是。实际上不定序列的效果是通过模板的局部特化来实现的。而能够确定个数的vectorn则是vector的基础。因此首先要看看vectorn（n不等于0时）是怎样实现的：

```
template<
    typename T0
>
struct vector1
: v_item<
    T0
    , vector0< >
>
{
    typedef vector1 type;
};
template<
    typename T0, typename T1
>
struct vector2
```

[关闭](#)

```

        : v_item<
          T1
          , vector1
        >
    {
        typedef vector2 type;
    };
    .....

```

目前MPL对vector中元素个数的限制是20个以内，所以这段代码一直递推到vector20为止。其中的v\_item是一个最底层的结构，它包含的内容类似于上面vector0中的那些成员：

```

template<
    typename T
    , typename Base
    , int at_front = 0
>
struct v_item
: Base
{
    typedef typename Base::upper_bound_index_;
    typedef typename next::type upper_bound_;
    typedef typename next::type size;
    typedef Base base;
    //这个空的静态函数将在at元函数中有确定类型位置的作用
    static aux::type_wrapper item_(index_);
    //默认的继承方式是private，重新使之可见
    using Base::item_;
};

```

很容易联想到前一部分提到的Typelist的Head和Tail结构，但是这里并不像Typelist一样需要一个NullType作结束标记。

至于不定序列vector的定义，则这样给出：

```

template<
    typename T0 = na, typename T1 = na, typename T2 = na,
    typename T3 = na, typename T4 = na, typename T5 = na,
    typename T6 = na, typename T7 = na, typename T8 = na,
    typename T9 = na, typename T10 = na, typename T11 = na,
    typename T12 = na, typename T13 = na, typename T14 = na,
    typename T15 = na, typename T16 = na, typename T17 = na,
    typename T18 = na, typename T19 = na
>
struct vector;
template< >
struct vector<
    na, na, na, na, na, na, na, na, na, na,
    na, na, na, na, na, na, na, na, na, na,
    , na, na, na
>
: vector0< >
{
    typedef vector0< >::type type;
};
template<
    typename T0
>
struct vector<
    T0, na, na, na, na, na, na, na, na, na,
    na, na, na, na, na, na, na, na, na, na,
    , na, na, na
>
: vector1

```

[关闭](#)

```

{
    typedef typename vector1::type type;
};
.....

```

显然可以看出，参数个数之所以可以不定，只是一个特化后的假象而已，针对每一个n，vector都会继承vectorn来制造这种假象。上面代码中的na是一个特殊的类，专用于标明参数未使用。

顺道一提deque，其实现也是通过继承vectorn来实现的，比如：

```

template<
    typename T0
>
struct deque<
    T0, na, na, na, na, na, na, na, na,
    na, na, na, na, na, na, na, na,
    , na, na, na
>
    : vector1
{
    typedef typename vector1::type type;
};

```

所以说deque是与vector等价的一个概念。

### 1.3.3. list

#### (1) 概述

MPL的list的原理类似于STL中list，其特点是支持前端常数时间的插入和删除操作以及中段和尾端线性时间的插入和删除操作。

#### (2) 操作

list所支持的操作与vector完全一样，参见vector的操作列表。

#### (3) 源代码分析

MPL的list的原理上十分类似于上面提到Typelist，其底层的结构l\_item是这样定义的：

```

template<
    typename Size, typename T, typename Next
>
struct l_item
{
    typedef aux::list_tag tag;
    typedef l_item type;
    typedef Size size;
    typedef T item;
    typedef Next next;
};

```

另外还需要一个结束标记l\_end：

```

struct l_end
{
    typedef aux::list_tag tag;
    typedef l_end type;
    typedef long_<0> size;
};

```

注意上面代码中的long\_<int>是一个整型常量类，下文还会分析到它

与vector相类似，list也分为不确定参数个数和确定参数个数的用，`list0`，`list1`，`list2`，`list3`，`list4`，`list5`，`list6`，`list7`，`list8`，`list9`，`list10`，`list11`，`list12`，`list13`，`list14`，`list15`，`list16`，`list17`，`list18`，`list19`，`list20`。

包括了从0到20的listn实现，其中list0直接继承结束标志使用，定义如下：

```

template<> struct list0<na>
    : l_end
{
    typedef l_end type;
};

```

对于n为1到20的实现，其代码如下所示：

```

template<
    typename T0, ..., typename Tn-1

```

关闭

```
>
struct listn
: l_item<
    long_<n>, T0, listn-1<T1, ..., Tn-1>
>
{
    typedef listn type;
};
```

可以看到，l\_item接受三个参数，第一个参数表示list的长度，第二个参数相当于上文Typelist实现中的Head，第三个参数相当于Tail。由于递推式的继承关系，listn的终结标记总是为l\_end。

list作为不定序列，其实现方式与vector如出一辙，也是通过继承listn，比如对于一个参数的list：

```
template<
    typename T0
>
struct list<
    T0, na, na, na, na, na, na, na, na, na, na,
    na, na, na, na, na, na, na, na, na
>
: list1<T0>
{
    typedef typename list1<T0>::type type;
};
```

### 1.3.4. set

#### (1) 概述和操作

set保证了key值在序列中没有重复，对它的插入和删除操作都只需要常数时间。

较之于vector和list，set没有pop\_back、pop\_front、push\_front、push\_back、insert\_range、back等几个操作，但另有几个特殊的操作：

has_key<s,k>::type	如果s中包含一个类型key值为k，则返回一个整型常量类，其值为true。
count<s,k>::type	返回s中key值为k的元素的序号。
order<s,k>::type	返回s中key值为k的元素唯一的整型常量类，其值是一个无符号整数。
at<s,k>::type at<s,k,def>::type	返回s中含有key值为k的元素。
key_type<s,x>::type	返回类型等同于x。
value_type<s,x>::type	返回类型等同于x。
erase_key<s,k>::type	返回一个新的set，当中不包括key值k。

#### (2) 源代码分析

MPL的序列都有一个共同点，就是都从一个sequence0开始构造，并以x\_item作为存放类型的基础结构。set比起上面的两种序列都来得复杂，它的构造首先从set0开始：

```
template< typename Dummy = na > struct set0
{
    typedef aux::set_tag    tag;
    typedef void_          last_masked_;
    typedef void_          item_type_;
    typedef item_type_     type;
    typedef long_<0>       size;
    typedef long_<1>       order;

};
```

s\_item会起到一个底层的构筑作用：

```
template< typename T, typename Base >
struct s_item
: Base
{

```

[关闭](#)

```
typedef void_    last_masked_;
typedef Base     next_;
typedef T        item_type_;
typedef item_type_ type;
typedef Base     base;

typedef typename next
< typename Base::size >::type size;
typedef typename next
< typename Base::order >::type order;
```

这样set $n$ 就可以定义了：

```
template<
    typename T0, typename T1, ...typename T $n-1$ 
>
struct set $n$ 
: s_item<
    T $n-1$ 
    , set $n-1$ < T0,T1...,T $n-2$  >
>
{
    typedef set $n$  type;
};
```

从这个结构看来，set并没有任何特别之处，更不像STL中的关联容器需要一个树形结构，还是递推式的结构，做法与list一模一样。

### 1.3.5. map

#### (1) 概述和操作

与set很相近，map也保证了key值在序列中没有重复，对它的插入和删除操作都只需要常数时间，它们间的唯一差别在于map存放key值类型和值类型成对的序列，而set中只有值（也是key值）map支持的操作与set也是相近的，但有两点不一样：

key_type<s,x>::type	返回类型等同于x::first。
value_type<s,x>::type	返回类型等同于x::second。

#### (2) 源代码分析

map的代码组织依然与上面的序列一样，由于使用值对，map的底层结构多接受了一个类型参数，是这样构筑的：

```
template< typename Key, typename T, typename Base >
struct m_item
: Base
{
    typedef Key    key_;
    typedef pair<Key,T> item;
    typedef Base   base;
    typedef typename next
< typename Base::size >::type size;
    typedef typename next
< typename Base::order >::type order;
};
```

相应地，map $n$ 比起set $n$ 也有了改变，定义时必须使用MPL的另一个辅助工具pair，以同时存放key值类型和值类型，下面代码中的P即指pair：

```
template<
    typename P0, typename P1, ...typename P $n-1$ 
>
struct map $n$ 
: m_item<
    typename P $n-1$ ::first
    , typename P $n-1$ ::second
```

关闭



```

        , mapn-1< P0,P1...,Pn-2 >
    >
    {
    };
};

```

map也有不定序列的形式，实现仍然是与上面3个序列非常类似，这里就从略了。

### 1.3.6. 整型序列包装器

#### (1) 概述

MPL的序列不只是为了存放类型而存在的，它们也可以存放整型常量。这里先引入一个整型常量类的概念。所谓的整型常量类，是指MPL中的一类特殊的包装器，它们的存在是为了把整型常量转换为新的类型，以便可以用于上面的四种序列。整型常量类包括如下5种：bool\_、int\_、long\_、size\_t和integral\_c。从名字就可以看出，每一种整型常量类都对应一种整型（在MPL中，bool也作为整型的一员），比如int\_<5>就可以产生一种代表常量5的类型。

以int\_为例，展开宏定义以后，它的定义大致如下：

```

template<int N>
struct int_
{
    static const int value=N;
    typedef int_ type;
    typedef int value_type;
    typedef integral_c_tag tag;
    typedef int_<N-1> prior;
    typedef int_<N+1> next;
    operator int() const
    { return static_cast<int>(this->value); }
};

```

至于integral\_c则更为通用，它的用法则是指定一种整型T，把一个整数常量转换为对应于T的新的类。比如MPL并没有提供short\_整型常量类，但

```
typedef integral_c<short,8> eight;
```

就可以实现相同的效果。

使用这些整型常量类，可以很方便地把常量也放进序列中与其他类型共存了，比如可以这样用：

```
typedef list<int_<4>, float,double,long double> floats;
```

但是这样用仍然有不方便之处，如果要建立一个只包含整数的序列，就要重复写很多次int\_<>了。整型序列包装器正是为了这个方便的目的而建立的。MPL共提供了4种整型序列包装器，它们是：range\_c、vector\_c、list\_c、set\_c。以vector\_c为例，要建立一个存放整数的序列，现在可以简单写成：

```
vector_c<T,c1,c2,... cn>
vectorn_c<T,c1,c2,... cn>
```

上面的T是一种整数类型，而cn则可以直接指定一个整数常量了。

有一个序列是需要特别观察的，那就是参数表跟其他序列很不一样的range\_c，它的用法是这样的：

```
//定义一个序列，其元素的范围是[0, 10)
typedef range_c<int,0,10> range10;
```

可以看到这样使用是更加简便了。但这也使得range\_c的实现与其他整型序列包装器都有一定的区别。range\_c不是一个可扩展序列，它没有插入或者删除等操作。

#### (2) 源代码分析

很容易猜测，这些整型序列包装器只是为使用的方便作一个包装，底层应该还是使用各序列原有的设施。仍然以vectorn\_c为例，固定参数的形式是这样的：

```

template<
    typename T
    , T C0, T C1,..., T Cn-1
>
struct vectorn_c
: v_item<
    integral_c< T,Cn-1 >
    , vectorn_c< T,C0,C1..., Cn-2 >
>
{
    typedef vectorn_c type;
    typedef T value_type;
};

```

关闭

```
};
```

同样，不定参数的vector\_c也是直接继承vectorn\_c，这里就从略了。

而MPL序列中的异类range\_c的实现则有点不一样：

```
template<
    typename T
    , T Start
    , T Finish
>
struct range_c
{
    typedef aux::half_open_range_tag tag;
    typedef T value_type;
    typedef range_c type;

    typedef integral_c<T,Start> start;
    typedef integral_c<T,Finish> finish;

    typedef r_iter<start> begin;
    typedef r_iter<finish> end;
};
```

可见range\_c并没有投射到任何一个已有的序列里面，而只定义了头尾的整型常量类。这也说明了range\_c为什么不是一个可扩展序列。

## 1.4. MPL的迭代器

### 1.4.1. 迭代器的定义和分类

迭代器是一类指向某一个或一定范围序列元素的实体，在MPL中起着解耦算法和序列关系的作用，由于这个作用，MPL中的算法甚至可以用于任何编译时的类型，只要这种类型符合迭代器的要求就可以了。

MPL中的迭代器一共分为3种：前向迭代器（Forward Iterator）、双向迭代器（Bidirectional Iterator）和随机访问迭代器（Random Access Iterator）。按照定义，这3种迭代器按照从先到后的顺序访问能力逐步增强。前向迭代器拥有产生指向到下一个元素的迭代器的能力，双向迭代器比起前向迭代器增加了产生指向前一个元素的迭代器的能力，随机访问迭代器则在双向迭代器的基础上能够根据特定的整数值产生同一序列中相对位置的迭代器。序列中也有3种访问方式的分类（前向、双向、随机访问），实际上序列的访问方式正是根据它可以产生的迭代器能力大小来分类的。

在目前的MPL实现中，随机访问迭代器与双向迭代器实际上是等价的，因为有且只有vector和range\_c同时支持这两个迭代器。

### 1.4.2. 迭代器的操作

迭代器支持以下6个操作：

advance	把迭代器移动一个相对的位置N。双向迭代器中N可为负值。
distance	计算出两个迭代器之间的距离，并包装在整型常量类中返回。
next	返回序列下一个迭代器，取决于序列的定义。
prior	返回序列上一个迭代器，取决于序列的定义。
deref	解引用，也就是提取迭
iterator_category	返回一个迭代器分类标记类型。

关闭

### 1.4.3. 源代码分析

由于每一个迭代器的操作包括了好几个实现版本，本文将不逐一分析每个实现，仅以advance和prior为例分析迭代器的一些基本实现手段。首先观察vector中的prior的源代码：

```
template<
    typename Vector
    , long, n_
>
```

```
struct prior< v_iter<Vector,n_> >
{
    typedef v_iter<Vector,(n_ - 1)> type;
};
```

现在这个通用接口又把具体的实现交给了序列的迭代器v\_iter，这段代码充分体现了随机访问迭代器的特点。迭代器v\_iter中有这样定义：

```
typedef typename v_at<Vector,n_>::type type;
```

这个绣球抛到内部函数v\_at那里去了，那是实现类型检索的最核心部件。

```
template< typename Vector, long n_ >
struct v_at
: aux::wrapped_type< typename v_at_impl<Vector,n_>::type >
{
};
```

```
template< typename Vector, long n_ >
struct v_at_impl
{
    typedef long_< (Vector::lower_bound_::value + n_) > index_;
    typedef __typeof__( Vector::item_(index_()) ) type;
};
```

上文在分析v\_item时曾经提到过一个静态的成员函数item\_，其特别之处在于index\_是一个整型常量类，函数item\_在以v\_item为桥梁的连串的继承（从vector0继承到vectorn）中不断被重载，于是，对于vectorn中的某个索引值，只要调用item\_，编译器就可以知道被重载的相应的类型值。最为巧妙的是，函数item\_完全不需要定义，它仅仅在编译期用于快速索引vector中的一个类型。代码中似乎新建了一个index\_对象，但其实这个对象在运行期不会产生任何作用，因为\_\_typeof\_\_只运行在编译期。现在来分析advance的实现，它的底层需要以prior和next为基础以移动迭代器：

*//外层通用接口，参见下一节“标记分派元函数”的说明*

```
template<
    typename BOOST_MPL_AUX_NA_PARAM(Iterator)
    , typename BOOST_MPL_AUX_NA_PARAM(N)
>
struct advance
//根据tag标记来选择相应迭代器的实现版本
: advance_impl< typename tag<Iterator>::type >
    ::template apply<Iterator,N>
{
};
```

```
template< typename Tag >
struct advance_impl
{
    //Iterator为要执行操作的迭代器，N是偏移值，可能为负
    template< typename Iterator, typename N >
    struct apply
    {
        //如果N小于0，则需要向反方向移动迭代器，
        //less见下文分析
        typedef typename less
            < N,long_<0> >::type backward_;
        //保证偏移值为正值，if_的原理上文已分析过
        typedef typename if_
            < backward_, negate<N>, N >::type offset_;
        //根据是否反方向移动，执行辅助元函数，见下面的代码
        //此处代码已展开宏以方便阅读
        typedef typename if_<
            backward_
            , aux::advance_backward< offset_::value >
            , aux::advance_forward< offset_::value >
            >::type f_;
```

关闭

```

        typedef typename apply_wrap1<f_,iterator>::type type;
    };
};
apply_wrapn是一个元函数的包装器，只是为了用更短的写法调用一个元函数类。
template<
    typename F, typename T1
>
struct apply_wrap1

    : F::template apply<T1>
    {
    };
advance_backward利用prior逐位反向移动，直到递归到特化版本为止，当中利用了一种称为“解循环/
递归”的特殊方法，目的是用预定义的大块移动减少循环次数，提高执行效率。
//特化版本，包括了从0到4共五个
template< long N > struct advance_backward;
template<>
struct advance_backward<0>
{
    template< typename Iterator > struct apply
    {
        typedef Iterator iter0;
        typedef iter0 type;
    };
};

template<>
struct advance_backward<1>
{
    template< typename Iterator > struct apply
    {
        typedef Iterator iter0;
        typedef typename prior<iter0>::type iter1;
        typedef iter1 type;
    };
};
.....
//非特化版本，当N大于4时调用
template< long N >
struct advance_backward
{
    template< typename Iterator > struct apply
    {
        //利用上述特化的最大值进行大块（chunk）移动，
        //使之尽快接近目的地
        typedef typename apply_wrap1<
            advance_backward<4>
            , Iterator
            >::type chunk_result_;

        typedef typename apply_wrap1<
            advance_backward<
                (N - 4) < 0
                ? 0: N - 4>
            , chunk_result_
            >::type type;
    };
};
};

```

关闭

advance\_forward做法类似，在此不再列举了。

## 1.5. MPL序列的内部元函数

### 1.5.1. 元函数和元函数类

上文提到的所有元函数，本质上都是一个类模板。上文分析过，这种简单的做法实际上隐藏着一个问题，由于C++严格区分类和模板，迫使模板无法直接传入另一个模板作为类型参数，除非使用限制比较大的“类模板的模板参数”。MPL的作者巧妙地解决了这个问题，方法是使用元函数类（Metafunction Class）。一个元函数类总是用一个类来包装一个命名为apply的模板，而这个apply模板中必须包括一个返回值type，代码类似于：

```
struct function_name
{
    template <typename t1, typename t2,...>
    struct apply
    {
        //要执行的操作
        typedef ... type;
    };
};
```

元函数类的好处是有效地包装了元函数的功能，使得元函数可以像一个普通类一样使用，同时规定了::apply<>::type风格的统一接口，使得元函数类很容易进行组合，与STL的仿函数用法十分相近。

### 1.5.2. 序列内部元函数的机制

序列内部的元函数指的就是上文提及的序列支持的各类操作，这些操作在名称上完全仿照STL，但MPL与STL的差别在于，STL容器的操作通常是作为容器成员函数定义和使用的。在MPL中，这些操作定义在序列的外部，无论使用任何一个操作都必须包含一个声明这个操作的头文件，比如要使用at这个元函数，就必须包含at.hpp文件。

使用通用的接口必然要求对类型进行识别，不幸的是，类模板无法（联系到不定序列的组织形式，或者也可以说是不便）提供像C++类成员函数那样的重载和匹配参数的能力，为了使用通用的接口，唯一的方案就是用一些预定义的标记来识别，在MPL中，每个标记实际上都是一个类。利用这种机制实现的元函数称作标记分派元函数（Tag Dispatched Metafunction）。

每一个标记分派元函数总是包含3个部分：一个元函数（作为外部的通用接口）、一个相关联的产生标记的元函数（区分实体类型）和底层实现（以元函数类的形式出现）。通过标记的判断，外层结构选择相应的底层实现来执行具体的操作，达到一种重载的效果。

MPL中大多数与序列和迭代器打交道的元函数都遵循标记分派元函数的机制来实现。

### 1.5.3. 源代码分析

由于序列的元函数为数甚多，每一个元函数又包括了好几个实现版本，本文将不逐一分析每一个实现，仅挑选具有代表性的元函数进行分析，以了解它们的概貌和内在的关系。

序列的内部元函数与迭代器关系十分密切，常常相互作用，以vector为例，元函数begin的接口和实现代码分别是：

```
template<
    typename Sequence=na
>
struct begin
{
    typedef typename sequence_tag<Sequence>::type tag_;
    typedef typename begin_impl< tag_ >
        ::template apply< Sequence >::type type;

};
template<>
struct begin_impl< aux::vector_tag >
{
    template< typename Vector > struct apply
    {
        typedef v_iter<Vector,0> type;
    };
};
```

关闭

};

这段代码非常典型，首先它体现出标记分派机制的作用，模板的特化能力令编译器能够找到适合的begin\_impl版本；另一方面，它又是元函数类的一个典型例子。可以看到，begin返回了一个指向vector头部的迭代器，然而从上文的分析又可以知道，v\_iter内部实际上还是调用了at的底层实现。

对序列进行删除的操作时会使用到vector的另外一个底层结构v\_mask：

```
template<
    typename Base
    , int at_front
>
struct v_mask
: Base
{
    typedef typename prior<typename Base::upper_bound_>::type index_;
    typedef index_ upper_bound_;
    typedef typename prior<typename Base::size_>::type size;
    typedef Base base;
    static aux::type_wrapper<void_> item_(index_);
    using Base::item_;
};

template<
    typename Base
>
struct v_mask<Base,1>
: Base
{
    typedef typename Base::lower_bound_index_;
    typedef typename next<index_>::type lower_bound_;
    typedef typename prior<typename Base::size_>::type size;
    typedef Base base;

    static aux::type_wrapper<void_> item_(index_);
    using Base::item_;
};
```

其定义与v\_item非常相似，当需要用pop\_front进行头部的删除时，v\_mask便会被使用了：

```
template<>
struct pop_front_impl< aux::vector_tag >
{
    template< typename Vector > struct apply
    {
        typedef v_mask<Vector,1> type;
    };
};
```

pop\_front调用的是v\_mask的特化版本，当v\_mask的第二个模板参数被设定为1时，表示从头部删除，否则将表示从尾部删除。事实上v\_mask并没有真正删除一个类型，它所做的事情只是屏蔽了第一个或最后一个元素。类似地pop\_back也使用v\_mask，差异仅仅是第二个参数改为0而已。

push\_front和push\_back使用的仍然是vector的最基础设施v\_item，其桥梁作用使得在头尾插入异常简单：

```
template<>
struct push_front_impl< aux::vector_tag >
{
    template< typename Vector, typename T >
    struct apply
    {
        typedef v_item<T,Vector,1> type;
    };
};
```

相应地，push\_back的实现中需要把v\_item的第三个参数改为0。

## 1.6. MPL中的元函数

关闭

### 1.6.1. 元函数的分类

与上述的序列内部元函数不同，这里指的元函数是具有通用意义的一些独立于数据结构的元函数，更确切地，可以与STL中的仿函数相对应。在运行时的编程中，普通的函数与重载了()运算符的仿函数类是截然不同的，但在模板元编程中，尽管就接口和用法而言，有普通的元函数和元函数类的区别，但模板元编程的本质上来说，两者并无区别。MPL的作者把一些在STL仿函数中的对应物抽取出来分为一类，仍以“元函数（Metafunctions）”称呼；而服务于序列和迭代器元函数则称为“内部元函数（Intrinsic Metafunctions）”；另有一类通用元函数，称为“算法（Algorithms）”。这也就是名词上容易造成一些混淆的原因。

MPL中的元函数分为两大类：通用元函数和数值元函数。通用元函数的作用类似于STL的仿函数适配器（Function Adapter），包括了类型选择（以if\_为代表）以及元函数的调用、组合和绑定设施（以Lambda表达式为代表）。数值元函数则差不多与STL的仿函数相对应，包括了算术、比较、逻辑和位运算四类。

### 1.6.2. 元函数的机制

首先来看看数值元函数的源代码，以less<>元函数为例：

```
//下面代码已展开所有宏
template< typename N1=na, typename N2=na >
struct less
: less_impl<
    //less_tag是一个简单的类型包装，定义见下
    typename less_tag::type
    , typename less_tag::type
    >::template apply::type
{};
//less_tag用于取得参数的标记，起识别运算符的作用
template< typename T > struct less_tag
{
    typedef typename T::tag type;
};
//如果两个参数都由标记判断出是整型常量类，则使用以下特化版本
template<>
struct less_impl< integral_c_tag,integral_c_tag >
{
    template< typename N1, typename N2 > struct apply
    : bool_< (N2::value > N1::value ) >
    {
    };
};
```

less的原理很简单，核心部分只是判断两个参数的value成员的大小而已。其他的数值元函数原理也一样。

至于通用元函数的机理，其核心是Lambda表达式的调用和组合能力，集中在下一节进行叙述。

## 1.7. Lambda表达式

### 1.7.1. lambda演算

逻辑学家Alonzo Church于1957年发明了称为lambda演算的数学体系。这种体系可以作为函数式编程语言的模型使用，类似于图灵机可以用于命令式的编程语言模型。实际上，lambda演算作为计算的描述与图灵机是相当的。

关闭

lambda演算的基础是lambda抽象：

$$(\lambda x. +1\ x)$$

其含义是建立了一个未命名的函数，此函数接受一个x作为参数，要实现1+x的操作。在lambda演算中，总是使用前缀表达式。如果需要应用这个lambda抽象，则写作：

$$(\lambda x. +1\ x)\ 5$$

这表示把1+x应用在常量5上。lambda演算使用归约规则应用函数，并产生结果。在这里，归约规则将用5来替换lambda抽象中的x，然后去掉lambda符号，得到表达式(+1 5)即3。

### 1.7.2. MPL中的Lambda

模板元编程既然是函数式风格的编程，自然应当允许执行lambda演算。MPL就提供了这样的设施，称之为Lambda表达式。每一个Lambda表达式都是一个编译期可调用的实体，它包含两种形式：元函数类和占位符表达式（Placeholder Expression）。

MPL把占位符表达式定义为：一种placeholder类或者一个包含至少一个占位符表达式参数的类模板的特化。所有的placeholder类都以“\_”开头，例如\_、\_1、\_2、.....\_n。这种写法只是为了方便使用而已，其背后实际上是一个类模板arg<n>，定义如下：

```
//已展开所有宏，n的范围从1到BOOST_MPL_LIMIT_METAFUNCTION_ARITY
template<> struct arg<n>
{
    static const int value = n;
    typedef arg<n+1> next;
    typedef na tag;
    typedef na type;
    template<
        typename U1 = na, typename U2 = na, typename U3 = na
        , typename U4 = na, typename U5 = na
    >
    struct apply
    {
        typedef Un type;
        //如果type是na，则表明未传递足够的参数，
        //需要产生一个错误信息
        BOOST_MPL_AUX_ASSERT_NOT_NA(type);
    };
};
```

arg<n>::apply允许接受的参数个数是由BOOST\_MPL\_LIMIT\_METAFUNCTION\_ARITY来决定的，MPL默认的设置是5，即最多接受5个参数。MPL还另外特化了一个arg<-1>，这个类型代表无参数的含义。

占位符的常用形式由arg<n>而来：

```
typedef arg<-1> _;
typedef arg<1> _1;
typedef arg<2> _2;
.....
```

可见，占位符实质是元函数类，它们能够起到选择参数表中特定位置的参数的作用，并把决定参数的时间推迟到向占位符填充实际的类型的时候。比如\_2::applyn>::type即表示参数表中的第二个参数的类型值。有了这些占位符，lambda演算中的变量就可以用它做中介，在使用Lambda表达式时，为被组合的元函数选择参数了。这样，元函数就可以作为一类值优雅地组合在一起，请看下面这个例子：

```
typedef plus<_, int<2>> >expr;
typedef lambda::type func;
```

这段代码把一个元函数包装起来，生成一个接受一个参数的lambda表达式，作用是为一个常量包装类的值加上2（int<2>）。这中间的巧妙之处就是，plus元函数接受两个类型参数，普通的整型常量类是类，占位符同样是类，它们都作为实际参数传入，并没有任何差别。特别地，占位符在进行Lambda表达式绑定以后，能够自动地对参数表进行选择，把正确的参数取来作为“真正的”实际参数，当中的过程十分微妙。

lambda元函数的定义如下：

```
template<
    typename T = na
    , typename Tag = void_
    , typename Arity = int< aux::template_arity::value >
>
struct lambda;
```

它有许多不同的特化版本，为不同的参数进行特别处理。下面这段代码是为拥有一个参数的类模板而特化的，这说明普通的元函数也可以用于Lambda表达式。

```
template<
    template< typename P1 > class F
    , typename T1
    , typename Tag
>
struct lambda<F, Tag >
```

[关闭](#)



```

{
    typedef lambda< T1,Tag > l1;
    typedef typename l1::is_le is_le1;
    typedef typename aux::lambda_or<
        is_le1::value
        >::type is_le;
    typedef aux::le_result1<
        is_le, Tag, F, l1
        > le_result_;
    typedef typename le_result_::result_result_;
    typedef typename le_result_::type type;
};
.....

```

对于普通的元函数类，特化版本是这样的：

```

template<
    typename F, typename T1
    , typename Tag
    >
struct lambda<
    bind1< F,T1 >, Tag, int_<2>
    >
{
    typedef false_ is_le;
    typedef bind1 result_;
    typedef result_ type;
};

```

从上面的代码可以知道，lambda的转换操作最终都要使用到bind $n$ 元函数类，这个函数正是高阶函数式编程的关键。这里仍以bind1为例：

```

template<
    typename F, typename T1
    >
struct bind1
{
    template<
        typename U1 = na, typename U2 = na,
        typename U3 = na, typename U4 = na,
        typename U5 = na
        >
    struct apply
    {
    private:
        //遇到_或arg<-1>占位符，根据其出现的
        //位置来解析为相应的arg，这里只绑定一个参数，
        //所以占位符要解析为arg<1>
        typedef aux::replace_unnamed_arg
            < F, mpl::arg<1> > r0;
        typedef typename r0::type a0;
        typedef typename r0::next n1;
        //f_在这里实际上就是要绑定的元函数类F
        typedef typename aux::resolve_bind_arg
            < a0,U1,U2,U3,U4,U5 >::type f_;

        typedef aux::replace_unnamed_arg
            < T1,n1 > r1;
        typedef typename r1::type a1;
        typedef typename r1::next n2;
        //当resolve_bind_arg的第一个参数为arg时，
        //t1::type的运算结果将是Un，即选择传入的
        //参数列表中的第n个作为元函数类F的参数，

```

关闭

```

        //比如在bind1这里，结果将是U1。
        typedef aux::resolve_bind_arg
            < a1,U1,U2,U3,U4,U5 > t1;
    public:
        //apply_wrapn的作用其实是接受多个参数的元函数类f_
        //生成一个新的元函数类，它只接受一个类型参数
        //以vector的形式出现，其中an是
        //f_原来的参数。
        typedef typename apply_wrap1<
            f_
            , typename t1::type
            >::type type;
    };
};

```

上面的源代码逐段看起来还很令人疑惑，概括起来

```
typedef bindn g;
```

的含义相当于

```

struct g
{
    template<
        typename U1 = unspecified
        ...
        , typename Un = unspecified
    >
    struct apply
    : apply_wrapn<
        typename h0::type
        , typename h1::type
        ...
        , typename hn::type
    >
    {
    };
};

```

其中hk就相当于上面代码中的resolve\_bind\_arg，遇到元函数类时type就是该元函数类，遇到arg<n>时type就是参数Un。所以，元函数类bindn能够将元函数绑定在第n个占位符，从而生成接受n个参数的更高阶的元函数类。Lambda表达式也就是这样被构造出来的。

### 1.8. MPL中的算法

MPL的提供了相当丰富的算法，这些算法的概念基本上跟STL的algorithm头文件中定义的算法相对应，熟悉STL的程序员能够很轻易地掌握它们的用法。有的算法依赖于序列的类型，不同的序列会定义自己不同的实现方法。有的算法则用迭代器和MPL的一些元函数实现出来。

其中的iter\_fold元函数类，作用是把一个操作运用在序列中的每一个类型上。iter\_fold在MPL的算法实现中起着关键作用：

```

template<
    //欲作用的序列
    typename Sequence=na
    //开始状态，比如定义为序列的开端begin
    , typename State=na
    //对序列中的每个元素进行的操作
    , typename ForwardOp=na
>
struct iter_fold
{
    typedef typename aux::iter_fold_impl<
        ::boost::mpl::O1_size::value
        , typename begin::type
        , typename end::type
        , State
        , typename lambda::type
    >::type
    ;
};

```

关闭

```

        >::state type;
    };
    template<
        typename First
        , typename Last
        , typename State
        , typename ForwardOp
    >
    struct iter_fold_impl< n,First,Last,State,ForwardOp >
    {
        typedef First iter0;
        typedef State state0;
        typedef typename apply2< ForwardOp,state0,iter0 >::type state1;
        typedef typename mpl::next::type iter1;
        typedef typename apply2< ForwardOp,state1,iter1 >::type state2;
        typedef typename mpl::next::type iter2;
        .....
        //逐次迭代，直到把序列中的所有元素都执行了一次ForwardOp。
        typedef staten state;
        typedef itern iterator;
    };

```

比如当我们要构造一个返回序列numbers中最大元素的元函数类，就可以这样利用iter\_fold：

```

    typedef iter_fold<
        numbers
        , begin::type
        , if_< less< deref<_1>, deref<_2> >, _2, _1 >
        >::type max_element_iter;

```

由此也可见到其他算法对iter\_fold（包括其他的迭代算法比如fold、reverse\_fold、reverse\_iter\_fold等）的依赖。这种依赖是基础性的，为什么呢？对比起STL的迭代器，模板元编程本身实际上无法实现真正的迭代器访问，比如没有方便的“++”运算，同时在模板元编程中，由于无法定义变量，保存运算中的状态需要相当的笨拙的方法（如上面的iter\_fold）。如果没有这些基础的遍历算法，其他算法就会需要大量重复冗长的代码来完成功能了。

这里就不一一赘述其他的算法了，参考着STL的概念，是容易理解它们的实现的，不过如上分析，因为这是模板元编程的缘故，有许多代码不像普通程序实现得那么直接。

## 2. 结论

### 2.1. 为什么要使用模板元编程？

一项新的技术的引入，如果不是为了取代旧有的技术，那么必然是可以加强旧有的技术。C++的发展史中有无数例子阐述了这一点。模板元编程是一种编译时的计算，它不会也不可能取代运行时必须的动态处理技术。然而通过巧妙的模板元编程，一些传统上存在的技术矛盾得以缓解，程序能够以更优雅更自动的方式组织起来。

第一、一些原来要留到运行时才能确定的类型或数据可以提前到编译时确定，提高了程序的效率。

第二、程序有时可以当成元数据输入元函数处理，依赖模板元编程的代码生成能力，自动产生代码，在熟练掌握的情况下，编程效率得到提高，并且这些代码往往还带有某种规范性，容易管理。

第三、在C++中，模板元编程所用的语言是其子集，程序员无需额外学习别的元语言来操纵程序，他们可以用很自然的方式去理解和运用模板元编程。

### 2.2. 为什么要使用MPL？

在接触MPL之前，人们确实容易疑惑，像模板元编程这样应用面

关闭

立一个库呢？只有通过对MPL的了解和分析，MPL的强大威力才能展现在我们眼前。MPL的最大意义在于，它重新整合了先前零散发展起来的模板元编程技术，建立了一套相当完整的标准，使模板元编程从深奥难用的理论出发走向了实用化的道路。

概括起来，至少有四个原因使用MPL[6]：

1. 质量。MPL无疑是一个高质量的通用程序库，无论从架构到代码的实现，都体现出工业的强度，其代码的准确性和高效性更是完全值得信赖，用侯捷的话来说就是“无所不用其极”[24]。
2. 重用性。MPL的重用性突出表现在它拥有一个概念完整、耦合度低的组织架构，使模板元编程的一般使用者可以轻易摆脱一些复杂但与问题域无关的技术考虑，可以集中精力进行关于问题域的设计。
3. 可移植性。正如上文的分析，MPL为了跨越平台和编译器做了极多的基础工作，能够在各大主

流编译器上成功编译。单是这一点已经可以作为使用MPL的充分理由了。假如没有这个库的包装，很难想象要为那些没有模板局部特化能力的编译器额外写多少代码，也更难想象调试和优化在存在bug的编译器上是如何困难。

4. 乐趣！MPL能把程序员从大量重复性的烦杂劳动中解脱出来，要写出更健壮的程序成为了更加容易的事情。

MPL作为模板元编程发展史上的里程碑，必将为这个领域的发展起到引导作用。审慎乐观地估计，模板元编程终将在不久的将来走下金字塔的尖端，成为C++程序员日常编程工作的一部分。

顶 踩  
0 0

上一篇 C++ Meta Programming 和 Boost MPL(3)

下一篇 My reading schema for next few months

我的同类文章

C++ ( 22 )

- |                              |                    |                              |                   |
|------------------------------|--------------------|------------------------------|-------------------|
| • boost.tuple源码整理和使用...      | 2007-10-07 阅读 417  | • 泛型插入排序                     | 2007-09-18 阅读 350 |
| • 泛型归并排序                     | 2007-09-18 阅读 303  | • 为什么C++                     | 2007-09-18 阅读 438 |
| • C++ Meta Programming 和 ... | 2007-08-30 阅读 1467 | • C++ Meta Programming 和 ... | 2007-08-30 阅读 856 |
| • C++ Meta Programming 和 ... | 2007-08-30 阅读 898  | • 智能指针的标准之争：Boost...         | 2007-08-28 阅读 872 |
| • 我看国内的C++教育以及我...           | 2007-08-28 阅读 602  | • 泛型快速排序                     | 2007-08-28 阅读 448 |

更多文章



人工智能研究



开发一个app



清扫机



达内真假



卖车



笔记本cpu排



沉香手串价

猜你在找

- |                                   |   |
|-----------------------------------|---|
| 《C语言/C++学习指南》加解密密篇（安全相关算法）        | 基本语言细节--《The C++ Programming Language》--            |
| Swift与Objective-C\C\C++混合编程       | 《C++GUIProgrammingwithQt4》读书笔记Chapter               |
| 移动端海水技术实现                         | c++ GUI programming with QT4                        |
| 算法与游戏实战技术                         | C++ GUI Programming with Qt 4 - 103 实现自定义模型         |
| Android JNI专题高级实战 安卓项目整合C/C++网络通讯 | C++ GUI Programming With Qt4 - Prentice Hall 2006 1 |



智能超长待机



金立手机



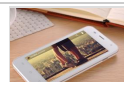
微型摄像头



五色草



智能...



...



...

关闭

查看评论

暂无评论

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

\* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目


[全部主题](#) [Hadoop](#) [AWS](#) [移动游戏](#) [Java](#) [Android](#) [iOS](#) [Swift](#) [智能硬件](#) [Docker](#) [OpenStack](#)

VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery  
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity  
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC  
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo  
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr  
Angular Cloud Foundry Redis Scala Django Bootstrap

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)

[网站客服](#) [杂志客服](#) [微博客服](#) [webmaster@csdn.net](mailto:webmaster@csdn.net) 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved 

关闭