C++ (/tags/#C++)　　基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B)

STL源码解析 (/tags/#STL%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90)

STL 源码剖笔记(一)

STL 源码剖析笔记(一)

Posted by 敬方 on July 6, 2019

> *2019-7-21 16:46:53*

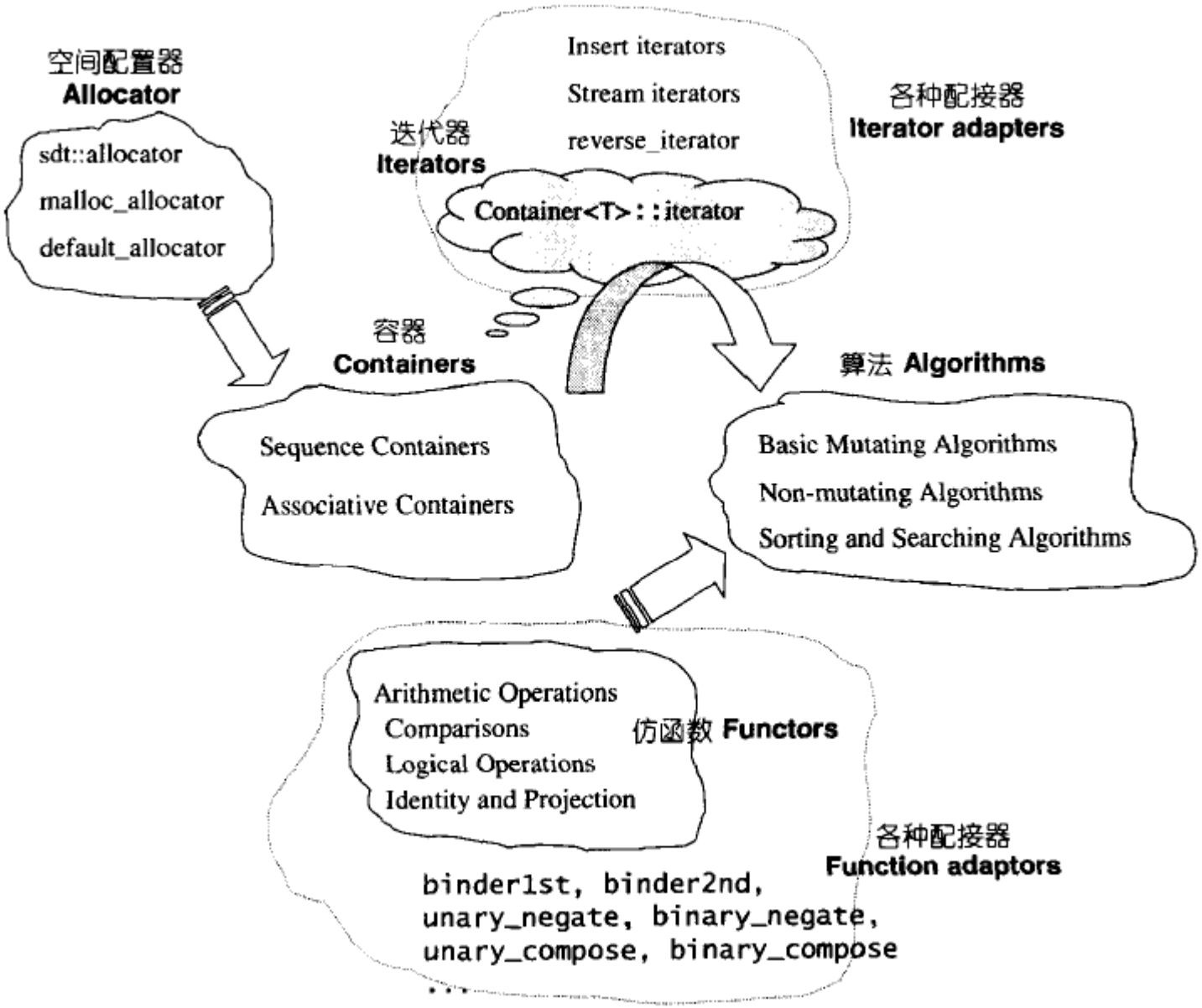
STL 源码剖析 –侯捷

STL概论与版本简介

STL六大组件

- 容器(containers):vector.list.deque,set,map等数据存放的类。
- 算法(algorithms):各种排序算法;sort、search、copy、erase..等
- 迭代器(iterators)：扮演容器与算法之间的胶合剂，即泛型指针

- 功能函数(functors)：行为类似函数，可以视为算法的某种策略
- 配接器(adapters)：一种用来修饰容器(containers)或仿函数(functors)或迭代器(iterators)借口的东西
- 配置器(allocators):负责空间配置与管理


空间配置器
Allocator
sdt::allocator
malloc_allocator
default_allocator
迭代器
Iterators
Insert iterators
Stream iterators
reverse_iterator
Container<T>::iterator
各种配接器
Iterator adapters
容器
Containers
Sequence Containers
Associative Containers
算法 Algorithms
Basic Mutating Algorithms
Non-mutating Algorithms
Sorting and Searching Algorithms
Arithmetic Operations
Comparisons
Logical Operations
Identity and Projection
仿函数 Functors
各种配接器
Function adaptors
binder1st, binder2nd,
unary_negate, binary_negate,
unary_compose, binary_compose
...

第二章 空间配置器

allocator是空间配置器，定义于头文件 std::allocator类模板是所有标准库容器所用的默认分配器 (Allocator)，若不提供用户指定的分配器。默认分配器无状态，即任何给定的 allocator实例可交换、比较相等，且能解分配同一`allocator`类型的任何其他实例所分配的内存。

成员类型

类型	定义
value_type	T
pointer (C++17 中弃用)(C++20 中移除)	T*
const_pointer (C++17 中弃用)(C++20 中移除)	const T*
reference (C++17 中弃用)(C++20 中移除)	T&
const_reference (C++17 中弃用)(C++20 中移除)	const T&
size_type	std::size_t
difference_type	std::ptrdiff_t
propagate_on_container_move_assignment(C++14)	std::true_type
rebind (C++17 中弃用)(C++20 中移除)	template< class U > struct rebind { typedef allocator other; };
is_always_equal(C++17)	std::true_type

成员函数

函数	定义
(构造函数)	创建新的 allocator 实例(公开成员函数)
(析构函数)	析构 allocator 实例(公开成员函数)
address(C++17 中弃用)(C++20 中移除)	获得对象的地址，即使重载了 operator&(公开成员函数)
allocate	分配未初始化的存储(公开成员函数)
deallocate	解分配存储(公开成员函数)
max_size(C++17 中弃用)(C++20 中移除)	返回最大的受支持分配大小(公开成员函数)
construct(C++17 中弃用)(C++20 中移除)	在分配的存储构造对象(公开成员函数)
destroy	(C++17 中弃用)(C++20 中移除)析构在已分配存储中的对象(公开成员函数)

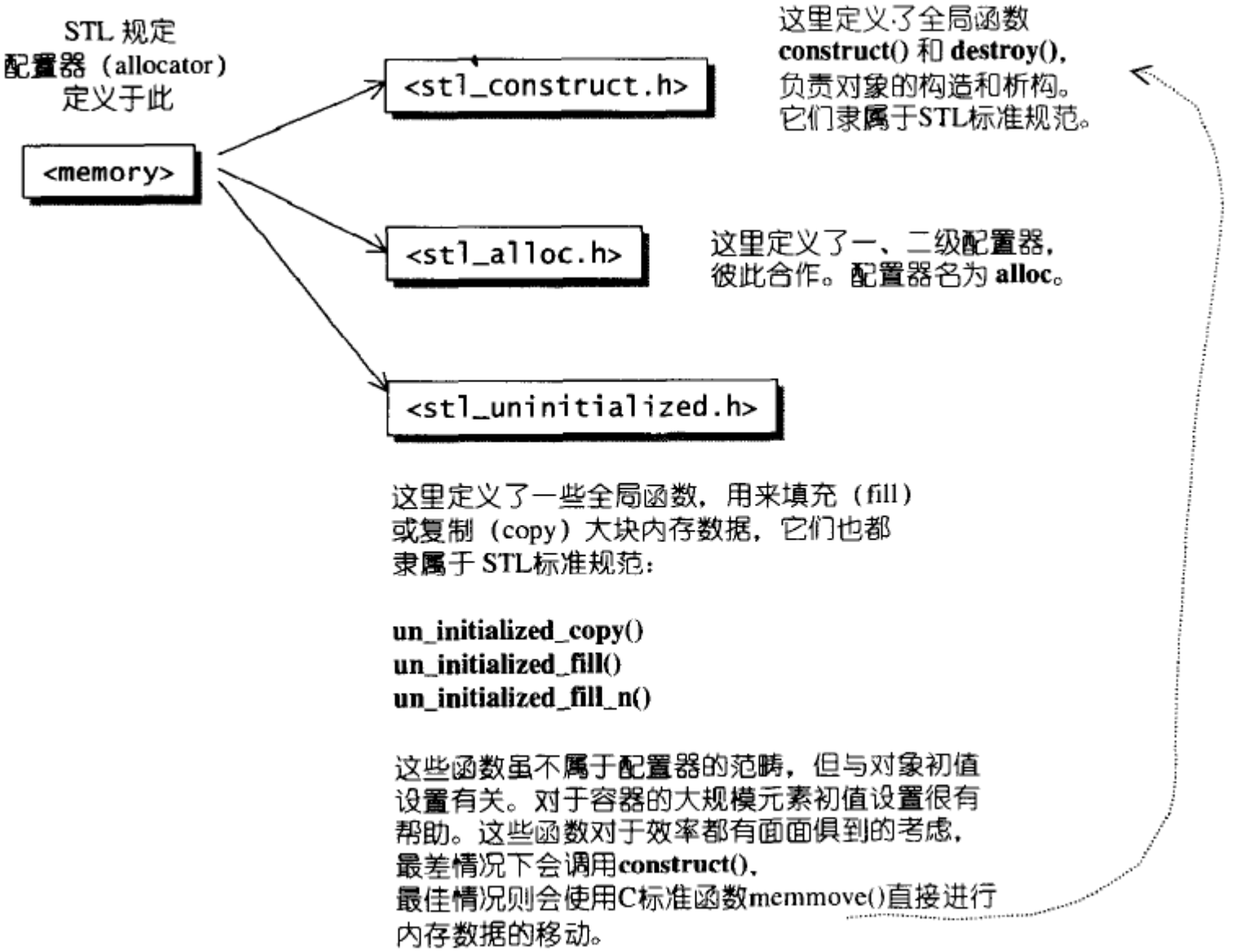
下面是一个 `std::allocator` 的简单实现 注意这里的 `size_t` 与操作系统相关，32位是4字节，64位时8字节； `ptrdiff_t` 是两个指针相减的结果类型，是一种有符号整数类型。减法运算的值为两个指针在内存中的距离。ptrdiff_t是signed类型，用于存放同一数组中两个指针之间的差距，它可以使负数，std::ptrdiff_t.同上，使用ptrdiff_t来得到独立于平台的地址差值.

```
#include <new.h> #include <stddef.h> #include <limits.h> #include <iostream.h> #include <algobase.h>
template <class T>
inline T* allocate(ptrdiff_t size,T*)
{
    //设置新句柄

    set_new_handler(0);
    T* tmp=(T*)(::operator new((size_t)(size* sizeof(T))));
    if(tmp==0)
    {
        cerr<<"out of memory"<<endl;
        exit(1);
    }
    return tmp;
}
template <class T>
inline void_deallocate(T* buffer)
{
    ::operator delete(buffer);
}
template <class T1,class T2>
inline void _construct(T1* p,const T2& value)
{
    new(p) T1(value);
}
inline void _destroy(T* ptr)
{
    ptr->~T();
}


//allocator类的基本实现

template <class T>
class allocator
{
public:
    typedef T  value_type;
    typedef T* pointer;
```

```
    typedef const T* const_pointer;
    typedef T&      reference;
    typedef const T& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t size_type;
    //连接 allocator和U

    template <class U>
    struct rebind
    {
        typedef allocator<U> other;
    };
    //获取左值指针

    pointer allocate(size_type n,const void* hint=0)
    {
        //分配内存并初始化为0

        return _allocate((difference_type)n,(pointer)0);
    }
    void deallocate(pointer p,size_type n){_deallocate(p);}

    void address(reference x){return (pointer)&x;}

    const_pointer const_address(const_reference x)
    {
        return (const_pointer)&x;
    }
    size_type max_size() const
    {
        return size_type(UINT_MAX/sizeof(T));
    }


};
```

上面只是简单的allocator实现，真实情况比这个要复杂的多。 SGI标准的空间配置器，是对::operator new和::operator delete做了一层薄薄的封装。

STL allocator将两阶段操作区分开来。内存配置操作由alloc:allocate()负责；内存释放操作由alloc::deallocate()负责；对象构造操作由::construct()负责，对象析构操作由::destroy()负责。



2.2.3 构造和析构基本工具： construct() 和 destory()

下面是c++1中 stl_construct.h的部分内容

```
#ifndef _STL_CONSTRUCT_H #define _STL_CONSTRUCT_H 1
#include <new> #include <bits/move.h> #include <ext/alloc_traits.h>
namespace std _GLIBCXX_VISIBILITY(default)
{
_GLIBCXX_BEGIN_NAMESPACE_VERSION

  /** * Constructs an object in existing memory by invoking an allocated * object's constructor with an initializer. */
//c++11 新添加Construct函数

#if __cplusplus >= 201103L   template<typename _T1, typename... _Args>
    inline void
    _Construct(_T1* __p, _Args&&... __args)
    { ::new(static_cast<void*>(__p)) _T1(std::forward<_Args>(__args)...); }
#else   template<typename _T1, typename _T2>
    inline void
    _Construct(_T1* __p, const _T2& __value)
    {
      // _GLIBCXX_RESOLVE_LIB_DEFECTS
      // 402. wrong new expression in [some_]allocator::construct
      ::new(static_cast<void*>(__p)) _T1(__value);
    }
#endif
  /** * Destroy the object pointed to by a pointer type. */
    //接受指针，并直接调用指针类的析构函数

  template<typename _Tp>
    inline void
    _Destroy(_Tp* __pointer)
    { __pointer->~_Tp(); }
//模板函数

  template<bool>
    struct _Destroy_aux
    {
      template<typename _ForwardIterator>
        static void
        __destroy(_ForwardIterator __first, _ForwardIterator __last)
    {
```

```
      for (; __first != __last; ++__first)
        std::_Destroy(std::__addressof(*__first));
    }
    };
//特例化函数判断对象是否含有默认构造函数，如果是则什么也不做，这样可以提高效率

  template<>
    struct _Destroy_aux<true>
    {
      template<typename _ForwardIterator>
        static void
        __destroy(_ForwardIterator, _ForwardIterator) { }
    };

  /** * Destroy a range of objects. If the value_type of the object has * a trivial destructor, the compiler should optimize all of

    //销毁一系列对象。如果对象的value_type是一个默认的析构函数，编译器应该优化所有这些，否则必须调用对象的析构函数。

  template<typename _ForwardIterator>
    inline void
    _Destroy(_ForwardIterator __first, _ForwardIterator __last)
    {
        //迭代器别名

      typedef typename iterator_traits<_ForwardIterator>::value_type
                       _Value_type;

      std::_Destroy_aux<__has_trivial_destructor(_Value_type)>::
    __destroy(__first, __last);
    }

  /** * Destroy a range of objects using the supplied allocator. For * nondefault allocators we do not optimize away invocation of *
    //使用支持的allocator销毁一系列对象，如果没有非默认的allocators不会进行销毁函数的初始化，即便，对象含有默认构造函数

  template<typename _ForwardIterator, typename _Allocator>
    void
    _Destroy(_ForwardIterator __first, _ForwardIterator __last,
         _Allocator& __alloc)
    {
```

```
      typedef __gnu_cxx::__alloc_traits<_Allocator> __traits;
      for (; __first != __last; ++__first)
    __traits::destroy(__alloc, std::__addressof(*__first));
    }

  template<typename _ForwardIterator, typename _Tp>
    inline void
    _Destroy(_ForwardIterator __first, _ForwardIterator __last,
         allocator<_Tp>&)
    {
      _Destroy(__first, __last);
    }

_GLIBCXX_END_NAMESPACE_VERSION
} // namespace std
#endif
```

上述代码中 _Destroy_aux 主要是用来检测，对象是否有自定义的析构函数，如果有就进行迭代调用。如果没有(使用默认析构函数)直接跳过，避免资源浪费。

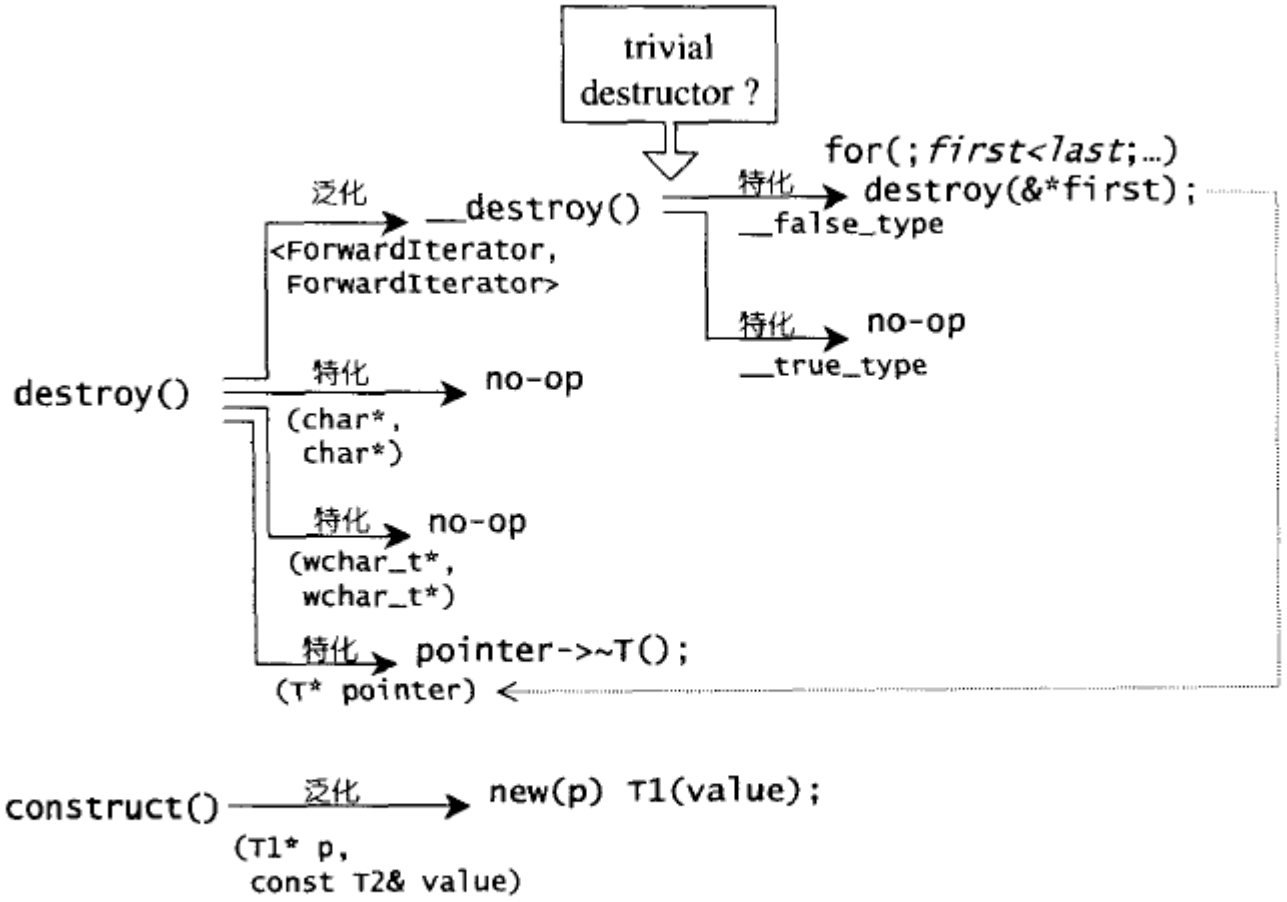


图 2-1 construct() 和 destroy() 示意

2.2.4 空间的配置与释放，std::alloc

构造空间的配置与释放，由负责；设计思路如下：

- 向 system heap 请求空间
- 考虑多线程状态
- 考虑内存不足时的应变措施
- 考虑过多“小行区块”可能造成的内存碎片(fragment)问题。

考虑到小型区块可能造成的内存破碎问题，SGI设计了双层级配置器,第一级配置器直接使用 malloc() 和 free()；第二级配置器则视情况采用不同的策略。当配置区块超过128byte时，视之为足够大，使用第一级配置器，当小于128byte时，视之为过小，采用复杂的memory pool整理方式。而不再求助于第一级适配器。使用哪一级适配器取决于 __USE_MALLOC 是否被定义。

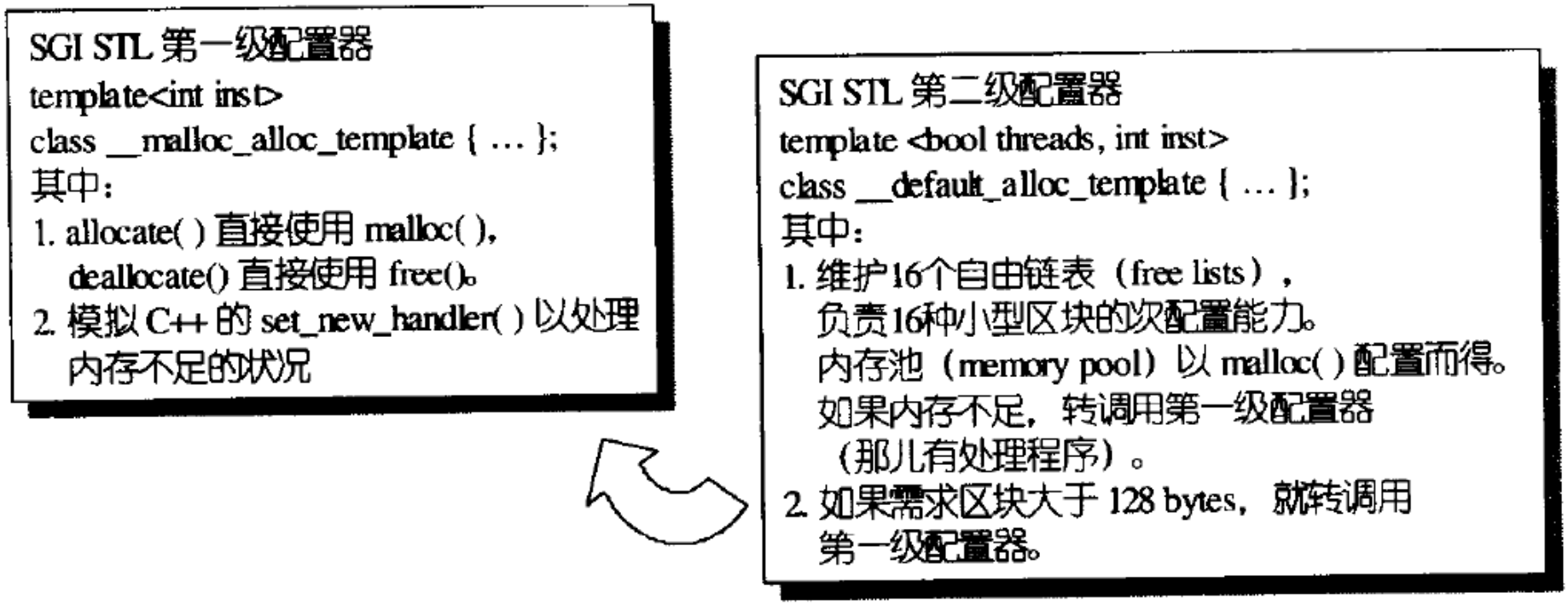


图 **2-2a** 第一级配置器与第二级配置器

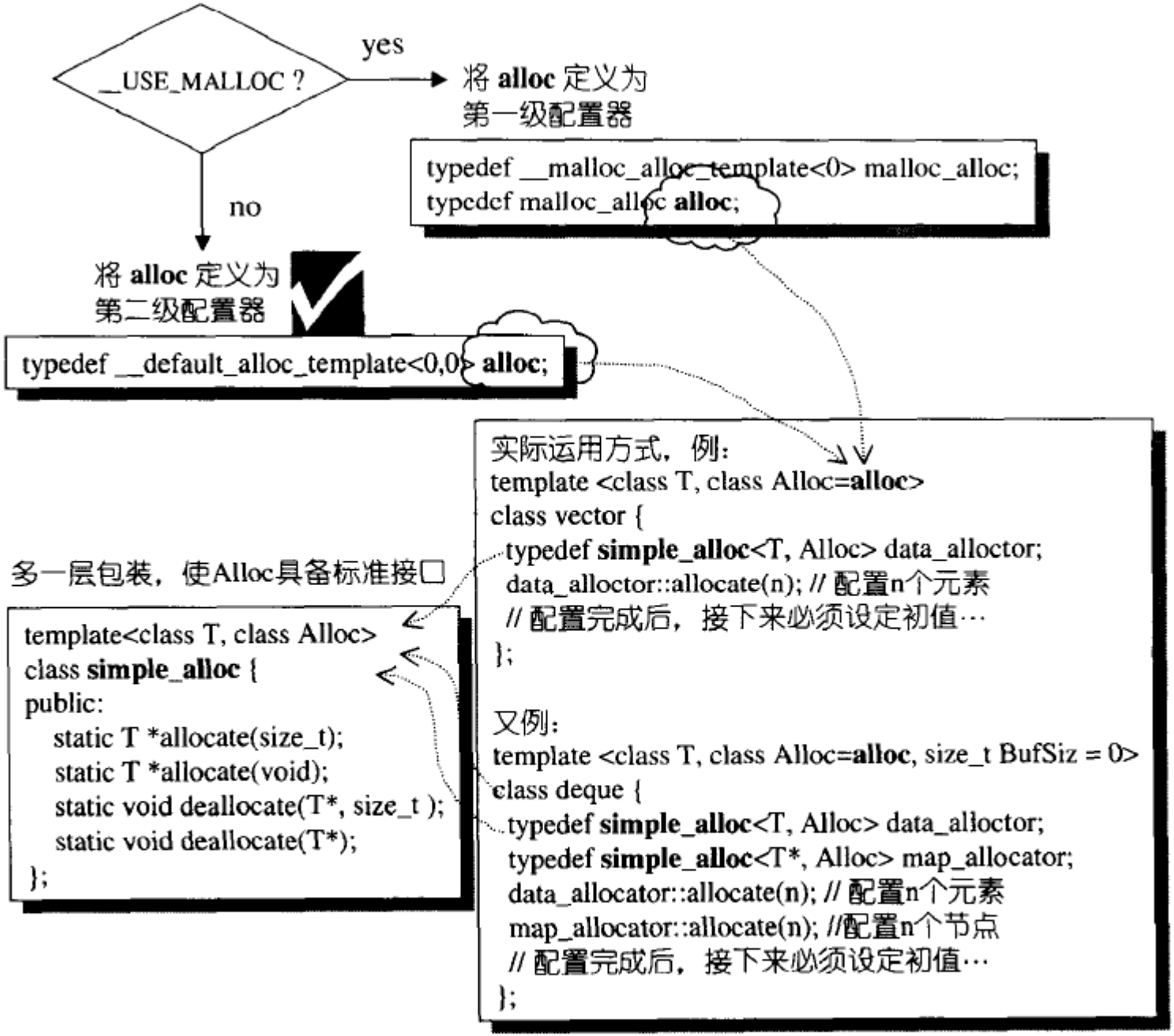


图 2-2b 第一级配置器与第二级配置器的包装接口和运用方式

图 2-20 第一级配置器与第二级配置器的总体设计结构与调用方式

2.2.5 一级适配器__malloc_alloc_template剖析

```
//一级适配器

template <int inst>

class __malloc_alloc_template
{
private:
    //函数指针，所代表的函数将用来处理内存不足的情况
    //oom: out of memory

    static void *oom_malloc(size_t);
    static void *oom_realloc(void*,size_t);
    static void (*__malloc_alloc_oom_handler)();

public:
    static void *allocate(size_t n)
    {
        //一级配置器必须使用malloc()

        void *result=malloc(n);
        //当无法满足需求时，改用oom_malloc()

        if(0==result) result=oom_malloc(n);
        return result;
    }
    static void deallocate(void *p,size_t)
    {
        //第一级配置器直接使用free()

        free(p);
    }
    //输入旧size和新size

    static void* reallocate(void *p,size_t,size_t new_sz)
    {
        void* result=realloc(p,new_sz);
        if(0==result) result=oom_realloc(p,new_sz);
        return result;
    }
```

```
    //指定自己的out-of-memory handler

    static void (* set_malloc_handler(void(*f)()))()
    {
        //获取函数指针

        void (* old)()=__malloc_alloc_oom_handler;
        __malloc_alloc_oom_handler=f;
        return(old);
    }


};
//下面是用户提供的malloc_alloc函数

template <int inst>
void (*__malloc_alloc_template<inst>::__malloc_alloc_oom_handler)()=0;

template <int inst>
void *__malloc_alloc_template<inst>::oom_malloc(size_t n)
{
    //获取内存分配句柄函数指针

    void (*my_malloc_handler)();
    void *result;
    for(;;)
    {
        //指针指向分配函数

        my_malloc_handler=__malloc_alloc_oom_handler;
        //分配失败抛出异常

        if(0==my_malloc_handler){__THROW_BAD_ALLOC;}
        //调用处理例程，企图释放内存

        (*my_malloc_handler)();
        //再次尝试分配内存

        result=malloc();
```

```
        if(result) return(result);
    }
}
//内存调用分配不成功时，进行二次调用

template <int inst>
void * __malloc_alloc_template<inst>::oom_realloc(void *p,size_t n)
{
    void (*my_malloc_handler)();
    void *result;
    for(;;)
    {
        my_malloc_handler=__malloc_alloc_oom_handler;
        if(0==my_malloc_handler){__THROW_BAD_ALLOC;}
        //尝试调用处理例程

        (*my_malloc_handler)();
        //尝试分配内存

        result=realloc(p,n);
        if(result) return(result);
    }
}
//注意,以下参数直接将inst指定为0

typedef __malloc_alloc_template<0> malloc_alloc;
```

2.2.6 第二级适配器 __default_alloc_template 剖析

因为对于操作系统而言，需要块分配一定的内存来存储块的位置信息，因此当内存过小时，单独开辟块反而是得不偿失的。因此在此时，C++使用内存池机制来对数据进行管理。由配置器负责内存的管理和回收，通常SGI配置其会将小额区块的内存需求量上调至8的倍数(例如30B-32B),并唯独16个free-list,各自管理大小分别为8,16,24,3，,40,48,56,64…128 bytes的小额区块。结构如下：

```
union obj{
    union obj* free_list_link;
    char client_data[1];
}
```

因为使用了union因此第一个指针直接指向了内存地址(c++ –> union介绍 (https://www.cnblogs.com/jeakeven/p/5113508.html))。

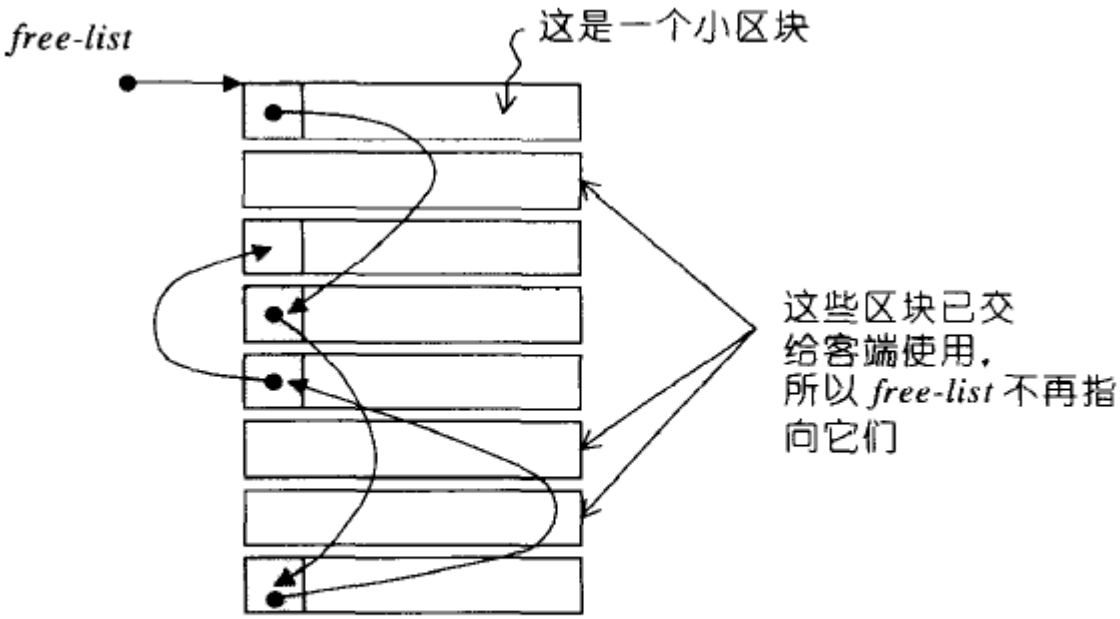


图 2-4 自由链表（*free-list*）的实现技巧

2.2.7 空间配置函数allocate()

__default_alloc_template拥有配置器的标准接口函数allocate()。此函数首先判断区块大小，大于128bytes就调用第一级配置器，小于128bytes就检查对应的free list。如果有可用区块就直接拿来用，没有就将区块的大小上调至8倍数边界，然后调用 `refill()` 准备为 `free list` 重新填充空间。

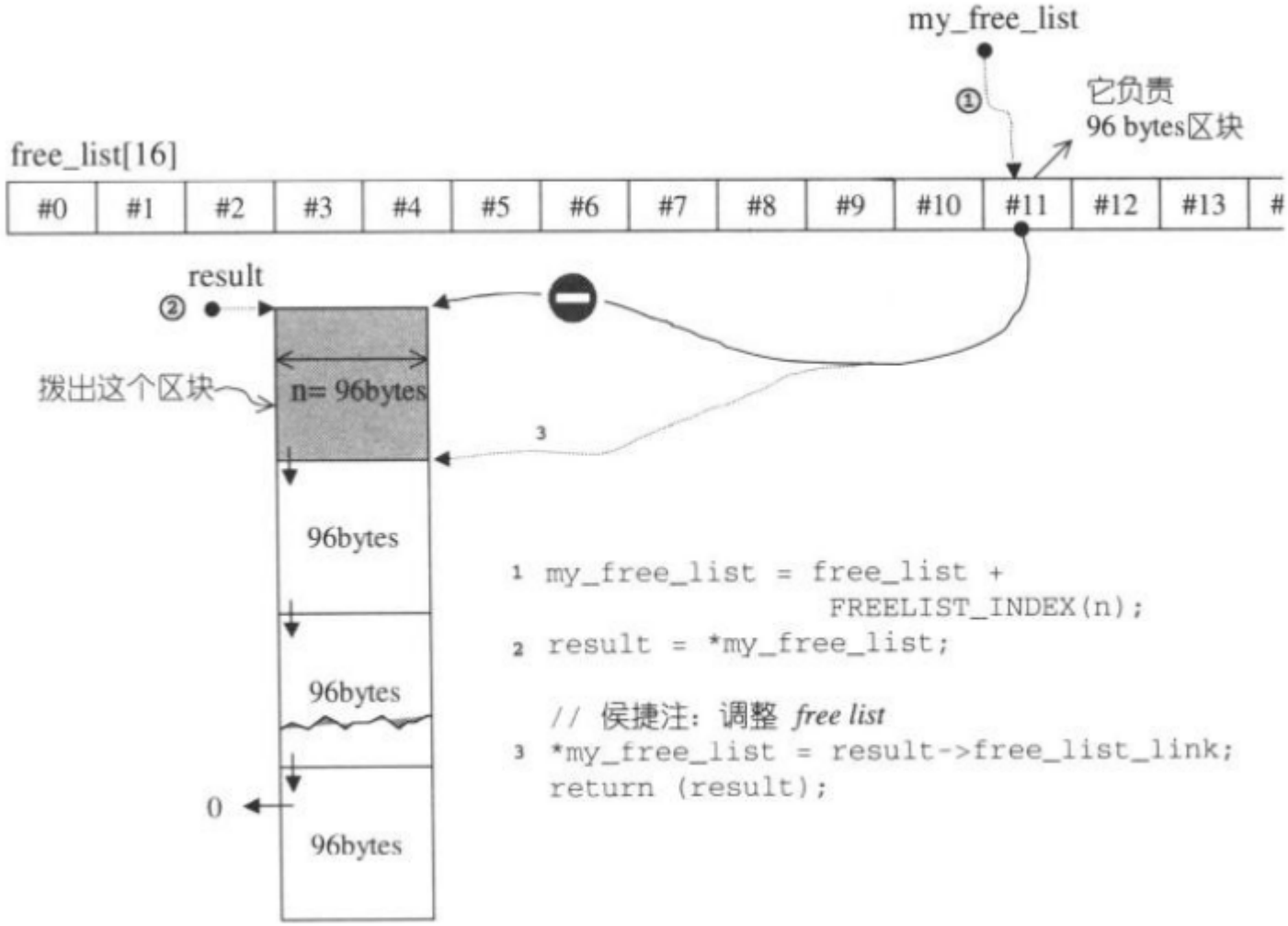


图 **2-5** 区块自 *free list* 拨出，阅读次序请循图中编号

2.2.8 空间释放函数 deallocate()

对于deallocate()函数首先判断区块大小，大于128bytes就调用第一级配置器，小于128bytes就找出对应的free list，将区块回收。

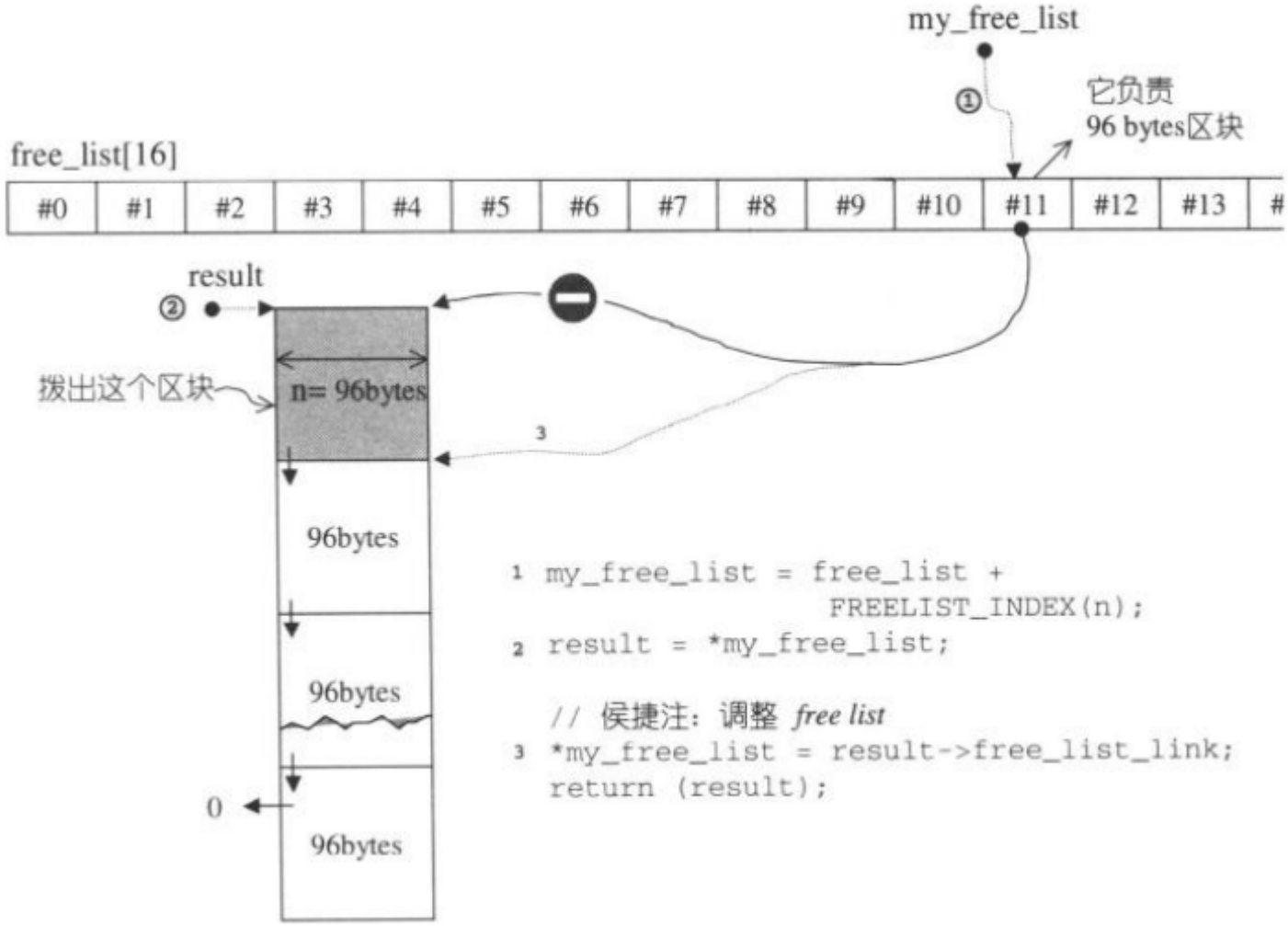


图 **2-5** 区块自 *free list* 拨出，阅读次序请循图中编号

2.2.9 重新填充 free-list

当内存池中没有可用区块的时候，就调用 `refill()` 准备为free-list重新分配空间。新的空间将取自内存池，缺省取得20个新节点(新区块)，但万一内存池空间不足，获得的节点数(区块数)可能小于20。

2.2.10 内存池(memory pool)

参考链接： C++实现内存池 (https://blog.csdn.net/u010183728/article/details/81531392);

上文中提及使用内存池中的内存，提供给free-list方便存取内存。内存池主要是和线程池类似，使用预先分配来实现内存的集中分配和同一管理

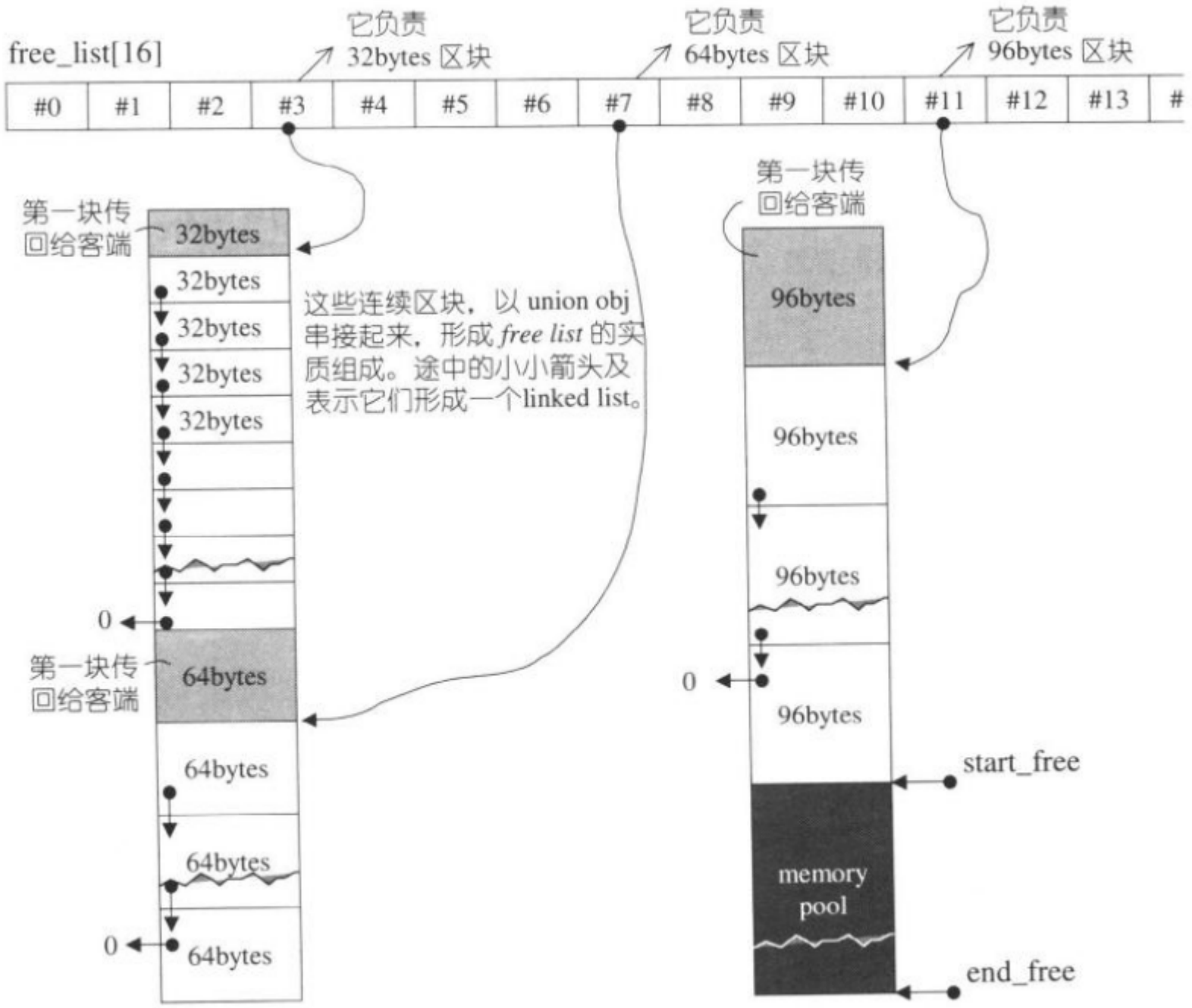
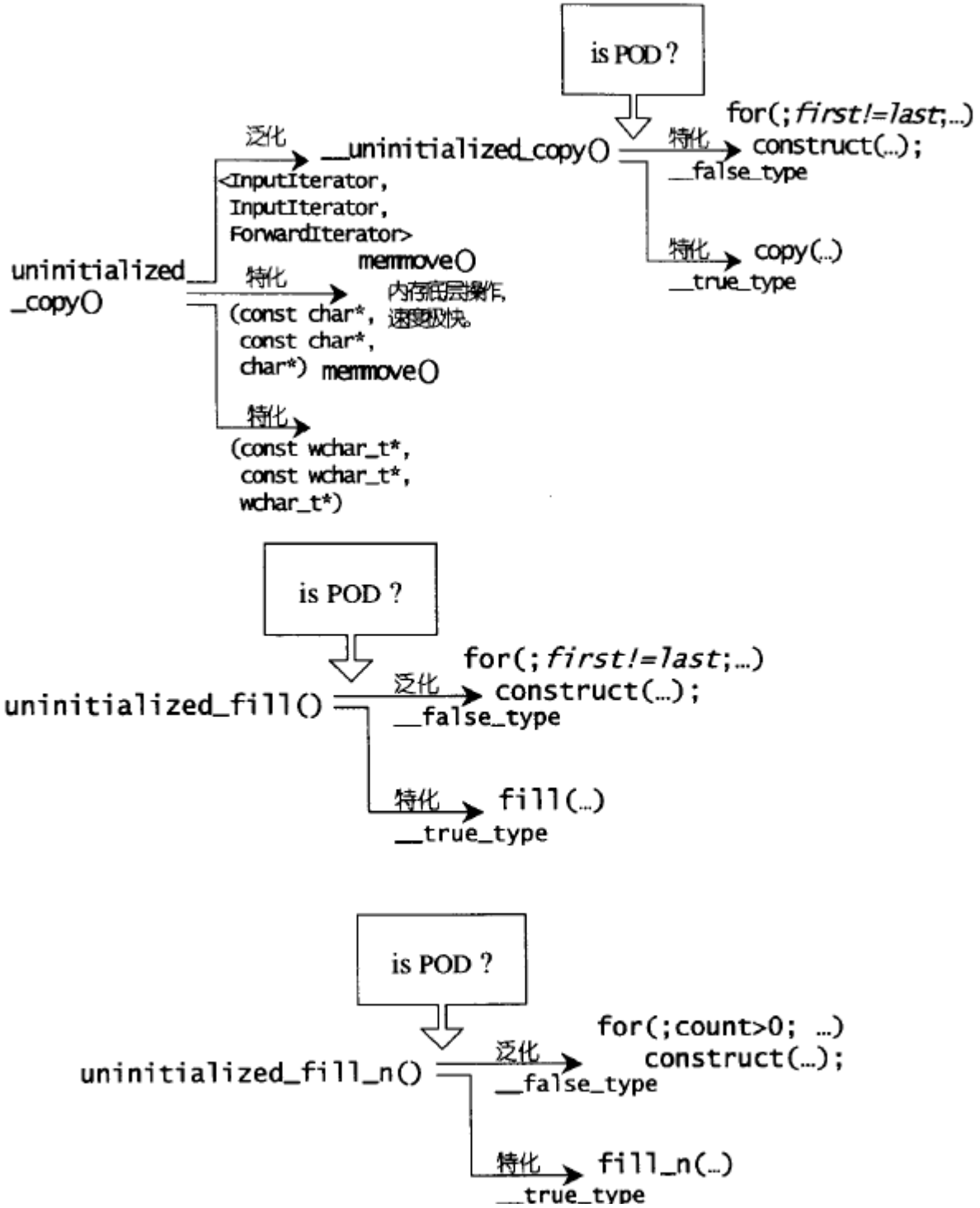


图 2-7　内存池（memory pool）实际操练结果

2.3 内存基本处理工具

STL中定义有5个全局函数，作用于未初始化空间之上。分别是 `construct()` 、 `destroy()` 、uninitailized_copy()、uninitailized_fill()、uninitialized_fill_n();分别对应于高层次函数copy()、fill()、fill_n();后面几个函数定义于

- uninitialized_fill_n((_ForwardIterator __first, _Size __n, const _Tp& __x))：它接受三个参数；并通过value_type来判断对象有没有构造函数，有就进行，没有直接跳过。
  - 迭代器 frist指向初始化空间的起始处
  - n表示分配内存的数量
  - x表示进行内存分配时候的初值
- uninitialized_copy(_InputIterator __first, _InputIterator __last,_ForwardIterator __result);输入参数列表如下，它调用了 `__uninitialized_copy()` 函数，并且通过value_type来判断是否为POD(传统基本数据类型含有拷贝构造等函数)，是则直接使用 `std::copy()` 进行构造。不是就使用遍历并使用 `td::_Construct` 调用类的构造函数进行构造。
  - frist 迭代器
  - last 迭代器
  - 表示最终结果
- uninitialized_fill(_ForwardIterator __first, _ForwardIterator __last,const _Tp& __x)和上面一样也是需要判断POD来决定使用构造方式和类型。
  - frist 迭代器
  - last 迭代器
  - 表示最终结果


is POD ?
for(;first!=last;...)
construct(...);
__false_type
copy(...)
__true_type
泛化
__uninitialized_copy()
特化
<InputIterator,
InputIterator,
ForwardIterator>
memmove()
uninitialized
_copy()
特化
(const char*,
const char*,
char*) memmove()
特化
(const wchar_t*,
const wchar_t*,
wchar_t*)
is POD ?
for(;first!=last;...)
construct(...);
__false_type
uninitialized_fill()
泛化
特化
fill(...)
__true_type
is POD ?
for(;count>0; ...)
construct(...);
__false_type
uninitialized_fill_n()
泛化
特化
fill_n(...)
__true_type

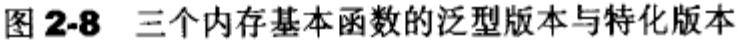


图 2-8 三个内存基本函数的泛型版本与特化版本

这一章主要讲述了STL中的alloctor通过源码解析，我们发现其实STL的迭代器使用了struct来进行构造，内部还是使用了指针运算符，而所谓的value_type也是指针相关的别名，感觉编译器厂商偷懒了，不过标准库的泛用性得到了保证。而且关于内存池确实是很好的亮点。慎用STL;

第三章 迭代器(iterators) 概念与traits编程技法

迭代器是一種抽象設計的概念，實現程序語言中並沒有直接對應的概念的實物。使用的23種設計模式中的迭代器模式：提供一種方法可以依次訪問某個聚合物所含的各個容器的元素。STL的中心思想在於：將數據容器和算法分開來。而迭代器就是扮演着兩者之間的膠合劑角色。

3.2 迭代器是一種 smart pointer

迭代器是一種行爲類似指針的對象，指针中最常见也最重要的行为便是间接引用(dereference)和成员访问(member access)，因此迭代器最重要的就是对operator*和operator->进行重载操作。c++STL中有一个auto_ptr（11中已经废弃），是用来包装一个原生指针(native pointer)的对象。可以解决各种内存漏洞。下面是auto_ptr的源码精要。

```
template<class T>
class auto_ptr
{
public:
    explicit auto_ptr(T *p=0):pointer(p){}
    template<class U>
    auto_ptr(auto_ptr<U>& rhs):pointee(rhs.release()){}
    ~auto_ptr(){delete pointee;}
    template<class U>
    auto_ptr<T>& operator=(auto_ptr<U>& rhs)
    {
        if(this!=&rhs) reset(rhs.release());
        return *this;
    }
    T& operator*() const {return *pointee;}
    T* operator->() const {return pointee;}
    T* get() const {return pointee;}
private:
    T *pointee;
}
```

下面是一个简单迭代器

```
template <class Item>
struct LisIter
 {
     Item* ptr;
     ListIter(Item* p=0):ptr(p){}
     //关键操作

     Item& operator*() const {return *ptr;}
     Item* operator->() const {return ptr;}
     ListIter& operator++()
     {
        ptr=ptr->next();
        return *this;
     }
     ListIter operator++(int)
     {
        ListIter tmp=*this;
        ++*this;
        retrun tmp;
     }
     bool operator==(const ListIter& i) const
     {
        return ptr!=i.ptr;
     }
 };
```

迭代器的另外一个非常重要的功能就是隐藏其中的细节；这一点标准库已经做的很好了。

3.3 迭代器相应型别(associated types)

迭代器相应型别是一种类似的封装行为，并不只是“迭代器所指对象的型别”

3.4 Traits编程技法–STL源码门钥

迭代器所指的型别，称为该迭代器的 value type。value用于函数的传回值就无法正常工作了。使用内嵌类型可以很好的避免这种问题。

```
template <class T>
struct MyIter
{
    //内嵌类型声明

    typedef T value_type;
    T* ptr;
    MyIter(T* p=0):ptr(p){}
    T& operator*() const {return *ptr;}
}

template <class I>
typename I::value_type func(I ite){return *ite;}

MyIter<int> ite(new int(8));
cout<<func(ite);
```

上面可以实现对类的特例化别名，但是对于基本数据类型却不可以，因此我们需要使用特例化模板 `template<>` 来对基本的数据类型进行封装;STL中提供了traits特性来对数据进行封装和改进。

traits意义在于，如果I定义有自己的value type,那么通过这个traits的作用，萃取出来的value_type就是I::value_type。这样traits就可以拥有特例化版本。

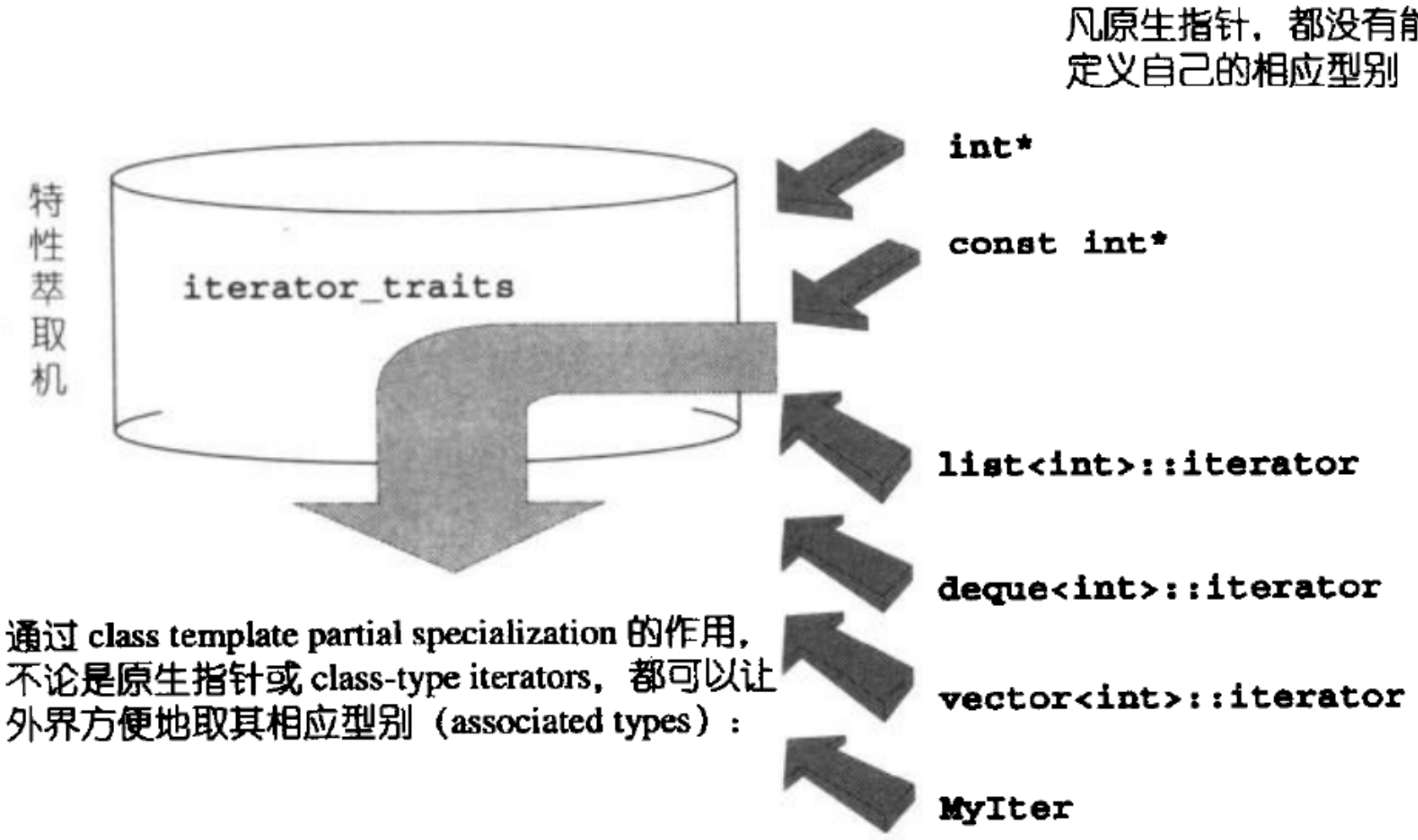


图 3-1 traits 就像一台“特性萃取机”，榨取各个迭代器的特性（相应型别）

常用的迭代器类型有5种：

- value_type: 迭代器所指对象的型别。
- difference_type:两个迭代器之间的距离，因此也可以用来表示一个容器的最大容量。表示头尾之间的距离参数。其原型是 `typedef ptrdiff_t difference_type` ;使用时可以用 `typename iterator_traits<I>::difference_type`
- reference_type: `typedef const T& reference type`
  - 迭代器分为两种 `const` 和非 `const` ； 不允许/允许改变所指内容的对象。
- pointer type 主要还是对象的指针类型。
- iterator_category: 主要用于大规模的迭代器

根据移动特性与施行操作，迭代器被分为5类

- input iterator:这种迭代器所指的对象，不允许外界改变。只读(read only)
- output Iterator:唯写(write only)
- Forward Iterator:允许“写入型”算法(如replace())在此种迭代器所形成的区间上进行读写操作。
- Bidirectional Iterator:可以双向移动。某些算法需要逆向走访某个迭代器区间(例如逆向拷贝某范围内的元素)，可以使用Bidirectional Iterators。
- Random Access Iterator: 前四中迭代器都只提供一部分指针算术能力，第五种则涵盖所有指针和算术能力，包括p+n,p-n,p[n],p1-p2,p1小于p2。

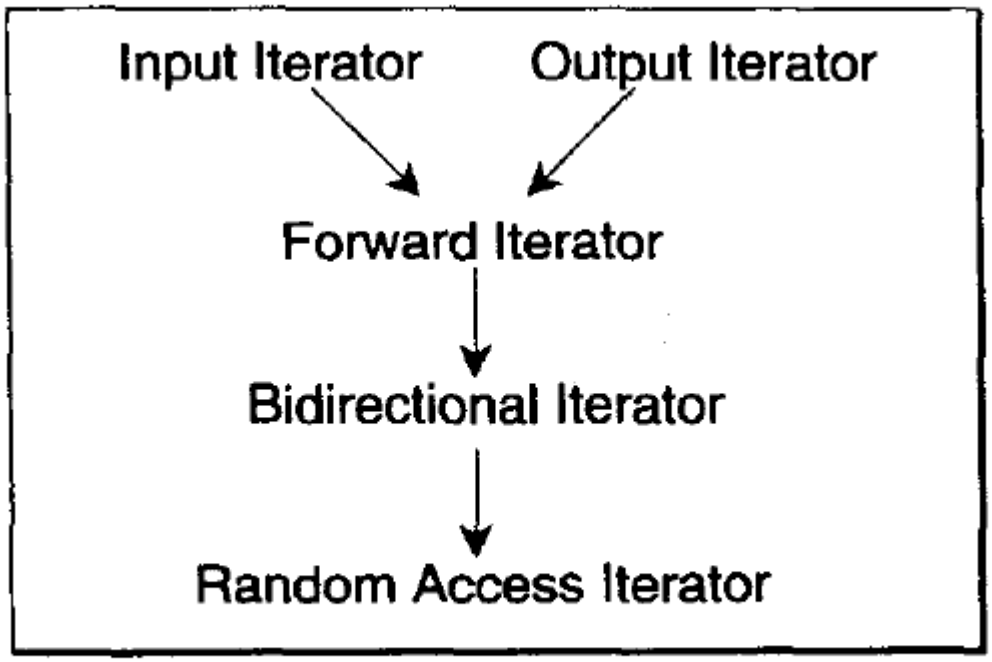


图 3-2 迭代器的分类与从属关系

;

3.5 std::iterator的保证

STL提供了一个iterator class 。每个新设计的迭代器都继承自它，可以保证STL所需之规范

```
template <
      class Category,
      class T,
      class Distance=ptrdiff_t,
      class Pointer=T*,
      class Reference=T&
          >
struct iterator
{
  typedef Category iterator_category;
  typedef T value_type;
  typedef Distance difference_type;
  typedef Pointer pointer;
  typedef Reference reference;

};

//使用

std::iterator<std::forward_iterator_tag,Item>
```

纯粹的接口类型类，因此没有额外的负担。

总结：traits编程技法大量用于STL实现品中，利用“内嵌型别”的编程技巧与编译器的template参数推导功能,增强了c++的类型推导能力。

3.6 iterator源码完整重列

```
//源自 <stl_iterator.h>

struct input_iterator_tag{};
struct output_iterator_tag{};
struct forward_iterator_tag:public input_iterator_tag{};
struct bidirectional_iterator_tag:public forward_iterator_tag{};
struct random_access_iterator_tag:public bidirectional_iterator_tag{};
//迭代器封装类

template <
      class Category,
      class T,
      class Distance=ptrdiff_t,
      class Pointer=T*,
      class Reference=T&
          >
struct iterator
{
  typedef Category iterator_category;
  typedef T value_type;
  typedef Distance difference_type;
  typedef Pointer pointer;
  typedef Reference reference;

};
//traits

template <class Iterator>
struct iterator_traits{
  typedef typename Iterator::iterator_category iterator_category;
  typedef typename Iterator::value_type value_type;
  typedef typename Iterator::difference_type difference_type;
  typedef typename Iterator::pointer pointer;
  typedef typename Iterator::reference reference;
}
//为原生指针而设计的traits偏特化版
```

```
template <class T>
struct iteraror_traits<T*>
{
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};
...
```

SGI STL的私房菜 __type_traits

iterator_traits负责萃取迭代器的特性，__type_traits则负责萃取型别(type)的特性。它提供了一种机制，允许针对不同的型别属性，在编译时期完成函数派送决定。

PREVIOUS
C++ 并发编程笔记(四)
(/2019/07/06/CPLUSPLUS_CONCURRENCY_IN_ACTION_04/)

NEXT
STL 源码剖笔记(二)
(/2019/07/06/CPLUSPLUS_ANNOTATED_STL_SOURCES_02/)

Be the first person to leave a comment!

FEATURED TAGS (/tags/)

C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B) C/C++ (/tags/#C/C++)
后台开发 (/tags/#%E5%90%8E%E5%8F%B0%E5%BC%80%E5%8F%91) C (/tags/#C) 网络编程 (/tags/#%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B)
STL源码解析 (/tags/#STL%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90) Linux (/tags/#Linux)
操作系统 (/tags/#%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F) 程序设计 (/tags/#%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1)
优化 (/tags/#%E4%BC%98%E5%8C%96) UML (/tags/#UML) UNIX (/tags/#UNIX) 学习笔记 (/tags/#%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0)
面试 (/tags/#%E9%9D%A2%E8%AF%95) Java (/tags/#Java) 读书笔记 (/tags/#%E8%AF%BB%E4%B9%A6%E7%AC%94%E8%AE%B0) go (/tags/#go)
阅读笔记 (/tags/#%E9%98%85%E8%AF%BB%E7%AC%94%E8%AE%B0)

FRIENDS

WY (http://zhengwuyang.com) 简书·JF (http://www.jianshu.com/u/e71990ada2fd) Apple (https://apple.com)
Apple Developer (https://developer.apple.com/)

● (https://www.facebook.com/wangpengcheng) ● (https://github.com/wangpengcheng)

Theme on GitHub (https://github.com/wangpengcheng/wangpengcheng.github.io.git) | Star 12