bitmap_allocator分析

说明	
来源	gcc version 11.3.0 (Ubuntu 11.3.0-1ubuntu1~22.04.1)
作者	libstdc++
类型	源代码

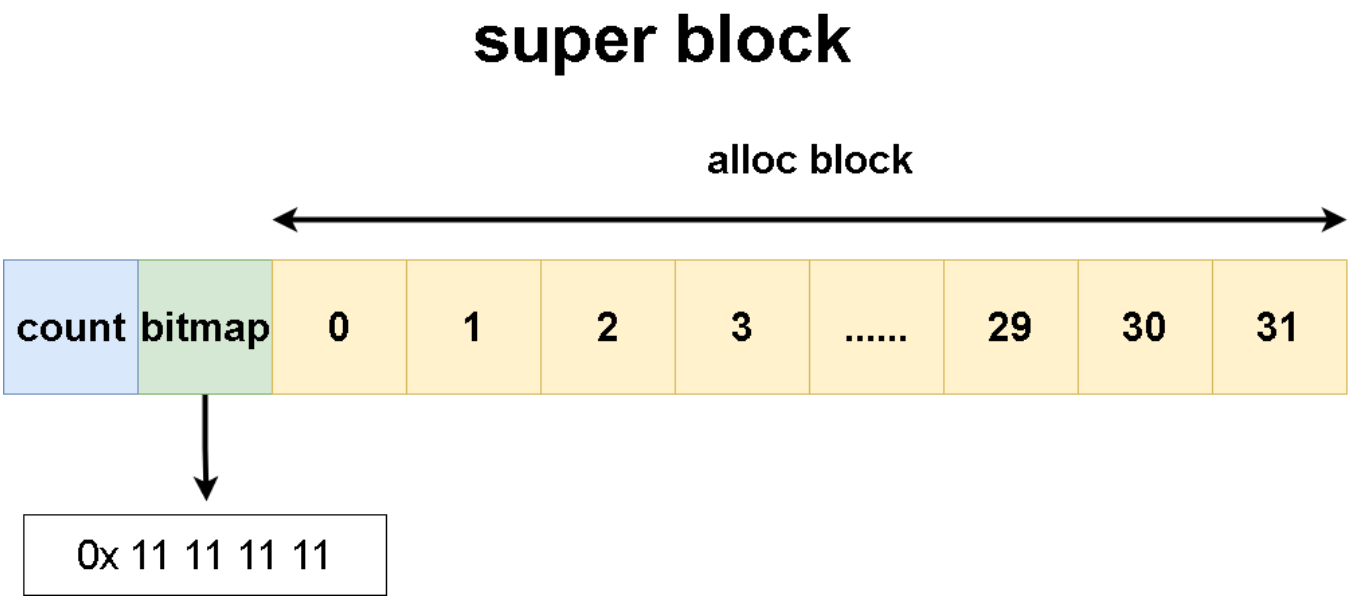
1 bitmap_allocator基本原理

bitmap内存分配器顾名思义是由bitmap管理内存的。主要有三个组件构成。分别是空闲链表（free list）、块对向量（block pair vector）以及超级块（super block）。其中super block是数据存储的主体，用于存放实际数据，block pair vector保存指向当前用户使用的超级块的指针。free list则维护了一个全局内存池，保存已从new/malloc分配，但未分配给用户的空内存块。

1.1 主要组件

super block超级块

super block是从free list获取的内存块，用于存储实际的数据。super block通过位图来管理内存的分配和释放，其具体构成为：计数器（count）、位图（bitmap）以及分配块（alloc block）。



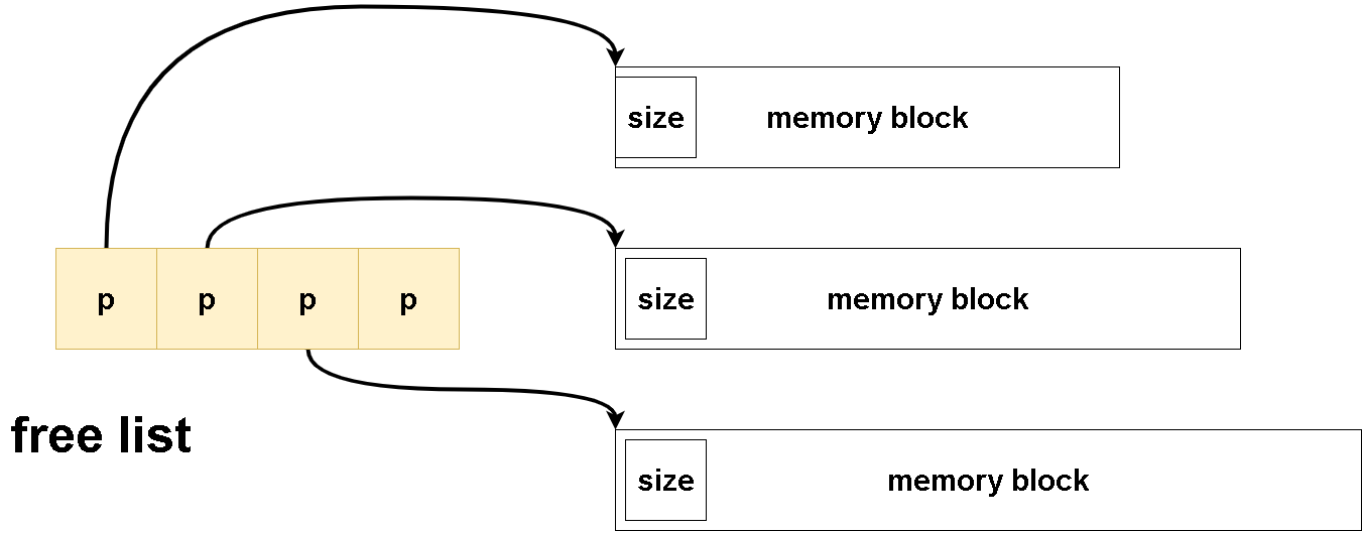
count存储了当前super block已经分配的block计数值。主要用于判断当前super block的所有block是否全部从用户回收，即count为0，如果全部回收则将这块super block归还free list。由free list决定将其暂存还是归还给操作系统（实际上应该是归还给malloc，因为malloc也会维护一个内存池）。

bitmap记录当前super block哪些块已经被分配。bitmap是一些std::size_t大小的整数，每个整数默认32位，因此一个bitmap block记录了32个block的使用情况。上图中共有32个block，因此只需要一个std::size_t大小的bitmap block。另外bitmap中记录的是逆序block使用情况，即最左边的block使用情况是由bitmap最右边的那一位表示的，其中1表示可分配，0表示未分配。

alloc blocks是实际数据存储单元。每个alloc block大小由bitmap_allocator模板参数的数据类型以及对其长度（_BALLOC_ALIGN_BYTES）决定。其中_BALLOC_ALIGN_BYTES默认8字节。每个super blocks中的alloc blocks数量不固定，初始super block的alloc blocks数是2个bitmap block对应数量（默认1个bitmap对应32个blocks）。后续每分配一个super block，就将新分配的alloc blocks数量翻倍，每归还一个super block就将新分配的alloc blocks数量减半。

block pair vector块对向量

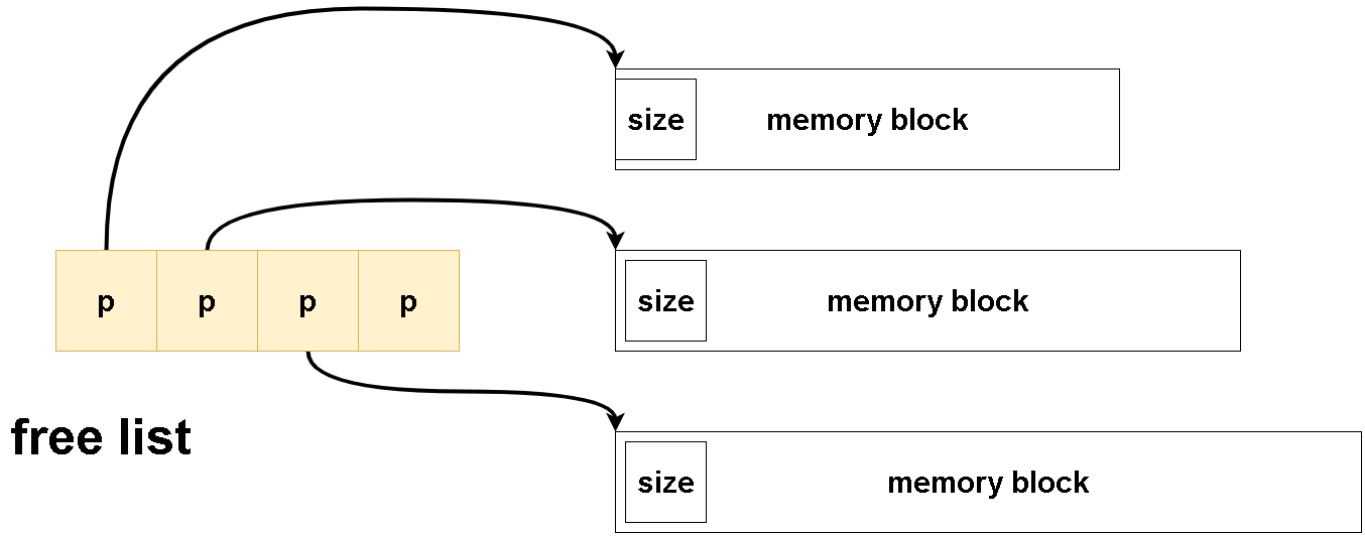
block pair vector记录了所有当前用户使用的super block的地址。顾名思义，block pair vector是一个vector，其中每一个元素是一个pair。pair由两个指向alloc block指针组成，分别指向一个super block的第一块alloc block和最后一块。



free list空闲链表

free list保存所有new/malloc分配的内存块。free list职责是作为bitmap allocator与底层内存分配的中介，当allocator需要一块super block时就会查找free list，向其索取合适的内存块。同时free list还会在适当时候将内存块归还给操作系统。

free list本质上也是一个vector，其中每一个元素是一个指针（类型为std::size_t*）指向内存块首地址。所有内存块按大小顺序进行排列，这里借用了所有内存块的前几个字节作为std::size_t类型，记录了内存块的大小。

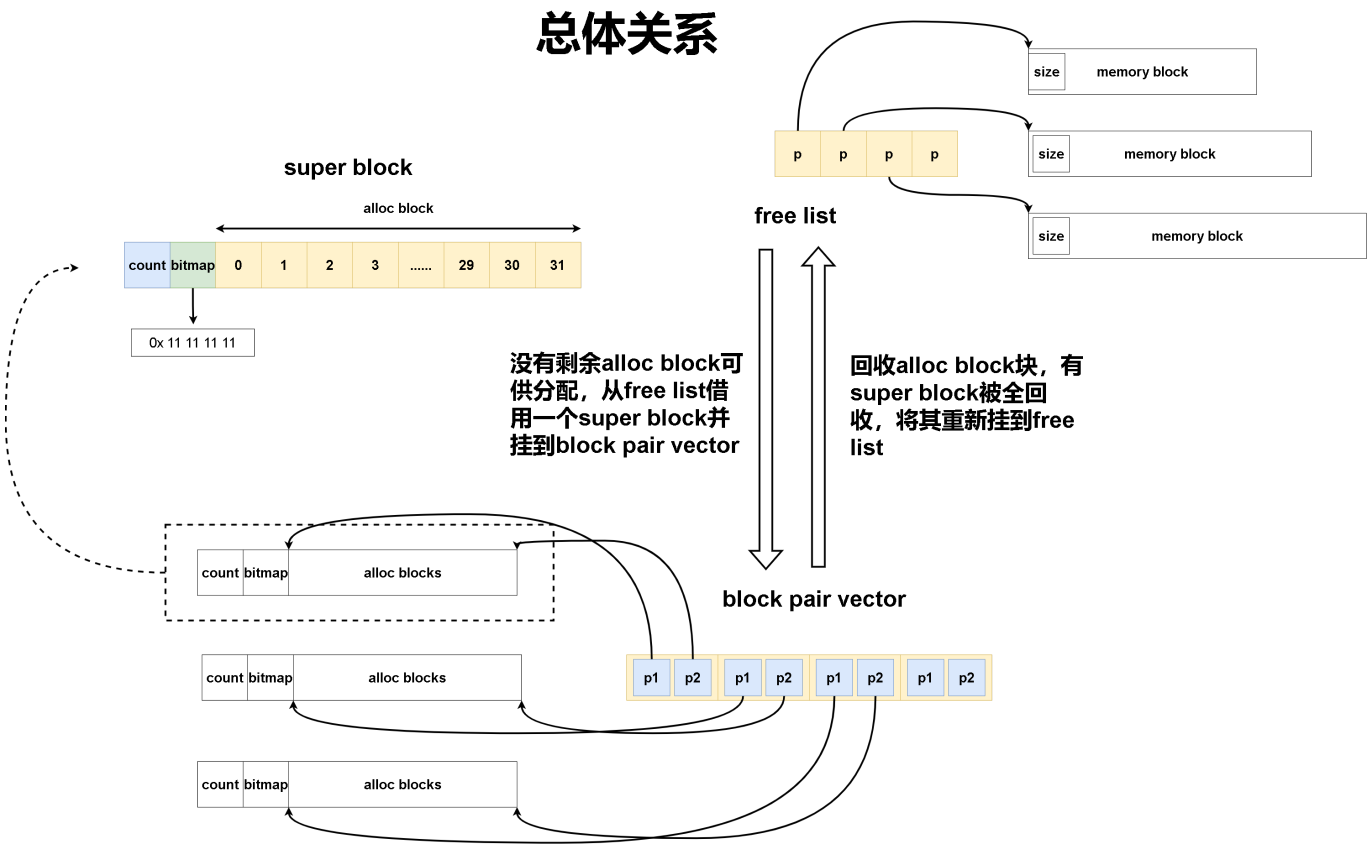


对于分配操作，因为free list中内存块有序，因此要获取required_size大小的内存块只需要找到block_size大小的内存块保证block_size > required_size即可。但是为了给每个分配的super block预留一定的空间，因此

block_size可以略大于required_size。最大block_size满足：(block_size - required_size) / block_size < max_wastage_percentage。其中max_wastage_percentage取值为固定值36。

对于释放操作规定free list最大长度max_size（固定为64），当free list超过最大长度时则需要将内存块归还操作系统。归还的内存块是free list中最大的那块。

上述几个组件的总体关系如下图所示：



1.2 示例

2 allocator接口

STL的allocator有统一接口标准。

std::allocator定义为：

```
template<class T>
struct allocator;
```

类型成员：

```
using value_type = T;
using pointer_type = T*;  //C++17弃用，C++20删除
using const_pointer = const T*;  //C++17弃用，C++20删除
using reference = T&;  //C++17弃用，C++20删除
using const_reference = const T&;  //C++17弃用，C++20删除
```

```
using size_type = std::size_t;
using difference_type = std::ptrdiff_t
using propagate_on_container_move_assignment = std::true_type;  //C++11

template< class U >
struct rebind
{
    typedef allocator<U> other;
};

using is_always_equal = std::true_type;  //C++11, C++23弃用
```

成员函数：

```
constructor  //创建新分配器实例
destructor  //销毁分配器实例
address  //获取对象的地址
allocate  //分配未初始化的内存
deallocate  //解除分配内存
max_size  //返回支持的最大分配大小
construct  //分配内存中构造对象
destroy  //销毁分配内存中的对象
```

非成员函数：

```
operator==  //比较两个分配器实例
operator!=  //C++20移除
```

3 详细设计

3.1 UML类图及方法说明

下面是libstdc++中bitmap_allocator主要class的接口及UML类图（其中为人熟知的STL统一接口用黄色标出）。这里只简要介绍每个class的职责及主要方法。

bitmap_allocator

位图分配器的主体class。包含STL allocator的统一接口。除了统一接口外，bitmap_allocator主要成员变量：

- _S_mem_blocks：这是上文所说的block pair vector，记录了所有当前用户使用的super block的地址。
- _S_block_size：上文提到super block中包含的alloc block块数不固定，_S_block_size记录了下一次新分配的super block中应该包含多少块alloc block。
- _S_last_request：Bitmap_counter类型（见下文）。用于记录上次请求分配的alloc block对应bitmap block位置，即对应哪个super block的哪个bitmap block。这是为了利用局部性原理，每次分配新的alloc block都会从上次分配的位置继续向后寻找。

- _S_last_deallocate_index：记录上次释放的alloc black所在的super block对应的block pair vector索引。同样是为了利用局部性原理。每次释放alloc block都会优先搜索其地址是否存在于上次释放alloc block所在的super block。

bitmap_allocator主要成员函数：

- _S_refill_pool()：分配新的super block并将其对应block pair加入block pair vector。这里分配主要调用了free_list::_M_get()方法（见下文）。
- _M_allocato_single_object()：分配从super block一个alloc block。如果没有可分配空间，调用_S_refill_pool()方法填充。
- _M_deallocate_single_object()：释放一个alloc block到super block。如果一块super block中所有alloc block都已经回收，就调用free_list::_M_insert()方法插入free list。同时更新block pair vector。

template<typename _Tp> bitmap_allocator
+ size_type: typeof std::size_t
+ difference_type: typeof std::ptrdiff_t
+ value_type: typeof _Tp
+ pointer: typeof _Tp*
+ const_pointer: typeof const _Tp*
+ reference: typeof _Tp&
+ const_reference: typeof const _Tp&
+ rebind: template<typename _Tp1> struct rebind {}
- aligned_size: template<std::size_t _BSize, std::size_t _AlignSize> struct aligned_size {} - _Alloc_block:: struct _Alloc_block { char __M_unused[]; } - _Block_pair:: typeof std::pair<_Alloc_block*, _Alloc_block*> - _BPVector:: typeof __mini_vector<_Block_pair> - _BPiter:: typeof _BPVector::iterator - _S_mem_blocks: _BPVector - _S_block_size: std::size_t - _S_last_request: _Bitmap_counter<_Alloc_block*> - _S_last_dealloc_index: _BPVector::size_type
- template<typename _Predicate> _S_find(_Predicate __P): _BPiter - _S_refill_pool(): void + _M_allocate_single_object(): pointer + _M_deallocate_single_object(pointer __p): void
+ allocate(size_type __n): pointer
+ allocate(size_type __n, typename bitmap_allocator<void>::const_pointer): pointer
+ deallocate(pointer __p, size_type __n): void
+ address(reference __r) const: pointer
+ address(const_reference __r) const: pointer
+ max_size() const: size_type
+ construct(pointer __p, const_reference __data): void
+ destroy(pointer __p): void
+ template<typename _Tp1, typename _Tp2> operator==(const bitmap_allocator<_Tp1>&, const bitmap_allocator<_Tp2>&): bool
+ bitmap_allocator()
+ ~bitmap_allocator()

free_list

空闲链表。free list实际上也是一个vector，注意free list没有显示的vector成员变量，但是提供了_M_get_free_list()成员函数获得一个静态vector变量_S_free_list。free list主要成员函数：

- _M_get_free_list()：获得静态vector变量_S_free_list，即空闲链表。

- _M_validate()：回收地址对应super block。同时根据当前链表长度及这个super block大小确定要释放哪个内存块给操作系统。
- _M_should_i_give()：根据请求块大小__required_size判断__block_size大小的内存块是否给出。
- _M_insert()：将地址对应super block插入free list，主要调用_M_validate()。
- _M_get()：返回可用内存块。会先调用_M_should_i_give()判断当前free list是否有可用内存块，否则调用new获取一块新的内存返回。

free_list
+ value_type: typeof std::size_t* + vector_type: typeof __minivector<value_type> + iterator: typeof vector_type::iterator - _LT_pointer_compare: struct _LT_pointer_compare { bool operator()() }
- _M_get_free_list(): vector_type& - _M_validate(std::size_t* __addr): void - _M_should_i_give(std::size_t __block_size, std::size_t __required_size): bool + _M_insert(std::size_t* __addr): void + _M_get(std::size_t __sz): std::size_t* + _M_clear(): void

_Bitmap_counter

_Bitmap_counter主要记录了bitmap allocator上次分配的alloc block对应bitmap block位置，并提供一些接口用于设置其指向位置。主要成员变量：

- _M_vbp：block pair vector的引用。
- _M_curr_index：指向上次分配alloc block对应的block pair vector中索引。
- _M_curr_bmap：指向上次分配alloc block对应bitmap block指针。
- _M_last_bmap_in_block：指向指向上次分配alloc block对应的super block中最后一块bitmap block指针。

主要成员函数：

- _M_reset()：将当前_M_curr_index指向设置为指定__index，并将_M_curr_bmap和_M_last_bmap_in_block设置为对应__index的super block的首尾bitmap block指针。
- _M_finished()：当前_M_curr_bmap是否为空。
- operator++()：向后移动_M_curr_bmap。
- _M_get()：获取_M_curr_bmap。
- _M_base()：获取_M_curr_index指向super block首个bitmap block指针。
- _M_offset()：获取当前_M_curr_bmap对应alloc block的偏移量。
- _M_where()：获取_M_curr_index。

template<typename _Tp> _Bitmap_counter
- _BPVector: typeof __mini_vector<typename std::pair<_Tp, _Tp>> - _Index_type: typeof _BPVector::size_type - pointer: typeof _Tp - _M_vbp: _BPVector& - _M_curr_index: _Index_type - _M_curr_bmap: std::size_t* - _M_last_bmap_in_block: std::size_t*
+ _M_reset(long __index = -1): void + _M_set_internal_bitmap(std::size_t* __new_internal_marker): void + _M_finished() const: bool + operator==(): _Bitmap_counter& + _M_get() const: std::size_t* + _M_base() const: pointer + _M_offset() const: _Index_type + _M_where() const: _Index_type + _Bitmap_counter(_BPVector& Rvbp, long __index = -1)

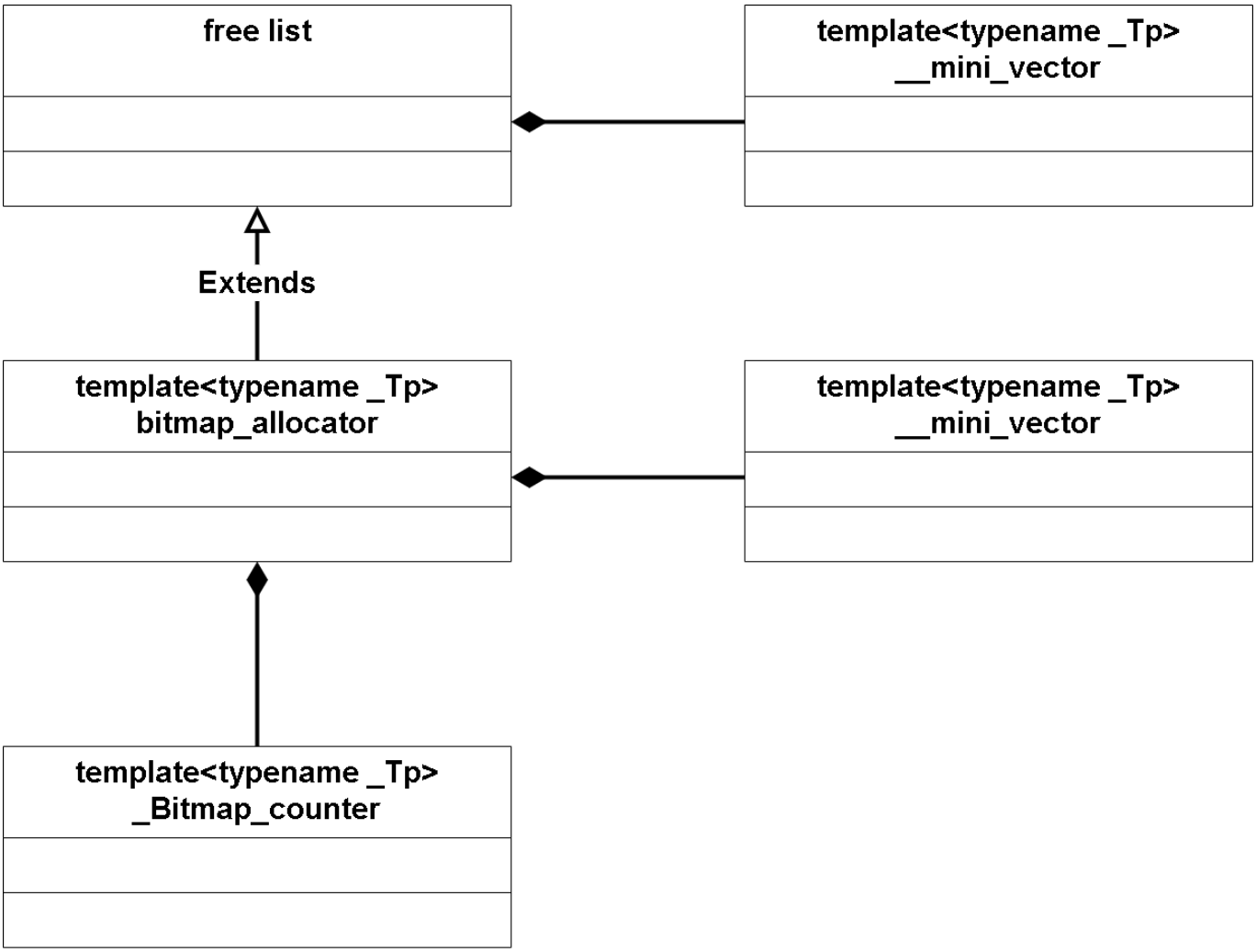
__mini_vector

__mini_vector是独立stl实现的vector，主要作为block pair vector和free list的底层数据结构。其基本接口与stl标准接口类似。唯一需要说明的是由于__mini_vector是脱离stl的，同时其本身就是用于实现allocator，因此其底层分配器使用了最简单的::operator new()和::operator delete()：

- allocate()：使用::operator new()分配空间。
- deallocate()：使用::operator delete()释放空间。

template<typename _Tp> __mini_vector
+ value_type: typeof _Tp + pointer: typeof _Tp* + reference: typeof _Tp& + const_reference: typeof const _Tp& + size_type: std::size_t + difference_type: std::ptrdiff_t + iterator: pointer
- _M_start: pointer - _M_finish: pointer - _M_end_of_storage: pointer
- _M_space_left() const: size_type - allocate(size_type __n): pointer - deallocate(pointer __p, size_type): void
+ size() const: size_type + begin() const: iterator + end() const: iterator + back() const: reference + operator[](const size_type __pos) const: reference + insert(iterator __pos, const_reference __x): void + push_back(const_reference __x): void + pop_back(): void + erase(iterator __pos): void + clear(): void + __mini_vector()

将这些主要class构成的类图如下图所示：



另外还有两个函数对象类型_Inclusive_between和_Ffit_finder。

_Inclusive_between用于判断给定block pair对应的alloc blocks中是否存在特定地址。被用于在_M_deallocate_single_object()函数中查找释放alloc block在哪个block pair对应的super block。

_Ffit_finder用于查找给定block pair中是否有非零的bitma block，即存在可用alloc block。被用于在_M_allocate_single_object()函数查找哪个block pair对应super block有可分配空间。

template<typename _Tp> _Ffit_finder
- _Block_pair: typeof std::pair<_Tp, _Tp> - _BPVector: typeof __mini_vector<typename std::pair<_Tp, _Tp>> - _Counter_type: typeof _BPVector::difference_type - _M_pbitmap: std::size_t* - _M_data_offset: _Counter_type
+ operator()(_Block_pair __bp): bool + _M_get() const: std::size_t* + _M_offset() const: _Counter_type + _Ffit_finder()

template<typename _Tp> _Inclusive_between
- pointer: typeof _Tp - _Mptr_value: pointer - _Block_pair: typeof std::pair<_Tp, _Tp>
+ operator()(_Block_pair __bp) const: bool + _Inclusive_between(pointer __ptr)

4 源码走读

上一章节提到的bitmap allocator四个主要class，其中__mini_vector由于和普通vector基本类似，就不做介绍。剩下三个class按照_Bitmap_counter、free_list和bitmap_allocator的顺序按自低向上的方式依次介绍。

4.1 一些重要定义

```
#define _BALLOC_ALIGN_BYTES 8

enum
{
    bits_per_byte = 8,
    bits_per_block = sizeof(std::size_t) * std::size_t(bits_per_byte)
};
```

bits_per_block在代码中出现频率较高，其定义了每一个bitmap block能管理的alloc block块数。通常为32块。

4.2 _Bitmap_counter

```
//_Bitmap_counter主要用于记录bitmap_allocator上次分配的位置用于下次分配从此位置开始
//_Bitmap_counter在bitmap_allocator中将被这样使用：_Bitmap_counter<_Alloc_block*>
_S_last_request
template<typename _Tp>
class _Bitmap_counter
{
    typedef typename
        __detail::__mini_vector<typename std::pair<_Tp, _Tp> > _BPVector;  //block
pair vector类型
```

```
    typedef typename _BPVector::size_type _Index_type;  //block pair vector索引类
型
    typedef _Tp pointer;  //指向_Alloc_block的指针类型

    _BPVector& _M_vbp;  //block pair vector，bitmap_allocator中block pair vector的
引用
    std::size_t* _M_curr_bmap;  //指向分配位置对应bitmap block指针
    std::size_t* _M_last_bmap_in_block;  //指向分配位置对应super block最后一块bitmap
block指针
    _Index_type _M_curr_index;  //指向分配位置对应block pair

public:

    _Bitmap_counter(_BPVector& Rvbp, long __index = -1) : _M_vbp(Rvbp)
    {
        this->_M_reset(__index);
    }

    //将当前block pair vector索引更新为__index。同时更新_M_curr_bmap和
_M_last_bmap_in_block为对应super block首尾bitmap block
    void
        _M_reset(long __index = -1) throw()
    {
        if (__index == -1)
        {
            _M_curr_bmap = 0;
            _M_curr_index = static_cast<_Index_type>(-1);
            return;
        }

        _M_curr_index = __index;  //注意：bitmap block记录顺序和alloc block顺序相
反，最后一位bitmap记录的是起始block的使用情况。
        _M_curr_bmap = reinterpret_cast<std::size_t*>
            (_M_vbp[_M_curr_index].first) - 1;

        _GLIBCXX_DEBUG_ASSERT(__index <= (long)_M_vbp.size() - 1);

        _M_last_bmap_in_block = _M_curr_bmap  //指向当前最后一个bitmap block下一位置
            - ((_M_vbp[_M_curr_index].second
                - _M_vbp[_M_curr_index].first + 1)
                / std::size_t(bits_per_block) - 1);
    }

    void _M_set_internal_bitmap(std::size_t* __new_internal_marker) throw()
    {
        _M_curr_bmap = __new_internal_marker;
    }

    //_M_curr_bmap是否指向空（表示没有可分配空间）
    bool _M_finished() const throw()
    {
        return(_M_curr_bmap == 0);
    }
```

```
    //累加_M_curr_bmap，如果到整个bitmap_allocator最后，将_M_curr_bmap置零
    _Bitmap_counter& operator++() throw()
    {
        if (_M_curr_bmap == _M_last_bmap_in_block)
        {
            if (++_M_curr_index == _M_vbp.size())
                _M_curr_bmap = 0;
            else
                this->_M_reset(_M_curr_index);
        }
        else
            --_M_curr_bmap;
        return *this;
    }

    std::size_t* _M_get() const throw()
    {
        return _M_curr_bmap;
    }

    //获取当前block pair对应区间的首个alloc block
    pointer _M_base() const throw()
    {
        return _M_vbp[_M_curr_index].first;
    }

    //获取当前bitmap block对应的alloc block偏移
    _Index_type _M_offset() const throw()
    {
        return std::size_t(bits_per_block)
            * ((reinterpret_cast<std::size_t*>(this->_M_base())
                - _M_curr_bmap) - 1);
    }

    _Index_type _M_where() const throw()
    {
        return _M_curr_index;
    }
};
```

4.3 free_list

```
class free_list
{
public:
    typedef std::size_t* value_type;  //元素类型，free list存储指向内存块的指针
    typedef __detail::__mini_vector<value_type> vector_type;
    typedef vector_type::iterator       iterator;

private:
    //_LT_pointer_compare函数对象
```

```
    struct _LT_pointer_compare
    {
        //比较__pui指针解引用与__cui大小比较结果
        //注意free list中内存区块借用起始位置（一个size_t类型）记录区块大小
        bool operator()(const std::size_t* __pui, const std::size_t __cui) const
throw()
        {
            return *__pui < __cui;
        }
    };

    //返回静态空闲链表对象引用
    //这种将静态对象搬到专属函数内的做法可以参考effective c++条款4
    vector_type& _M_get_free_list()
    {
        static vector_type _S_free_list;
        return _S_free_list;
    }

     //将__addr对应内存块并插入free list，如果free list长度超过64，则释放最大的块
    void _M_validate(std::size_t* __addr) throw()
    {
        vector_type& __free_list = _M_get_free_list();
        const vector_type::size_type __max_size = 64;
        if (__free_list.size() >= __max_size)  //到达自由链表最大长度64
        {
            //移除最大的块
            if (*__addr >= *__free_list.back())
            {
                ::operator delete(static_cast<void*>(__addr));
                return;
            }
            else
            {
                ::operator delete(static_cast<void*>(__free_list.back()));
                __free_list.pop_back();
            }
        }

        //按内存块大小顺序将__addr插入free list
        //__lower_bound函数使用二分查找在__free_list寻找满足条件的插入点
        iterator __temp = __detail::__lower_bound(__free_list.begin(),
__free_list.end(), *__addr, _LT_pointer_compare());
        __free_list.insert(__temp, __addr);
    }


     //判断请求__required_size大小的块时__block_size大小的块是否应该给出
    bool _M_should_i_give(std::size_t __block_size, std::size_t __required_size)
throw()
    {
        const std::size_t __max_wastage_percentage = 36;
        if (__block_size >= __required_size &&
            (((__block_size - __required_size) * 100 / __block_size)
```

```
                < __max_wastage_percentage))
            return true;
        else
            return false;
    }

public:

     //向free list插入空的super block
    inline void _M_insert(std::size_t* __addr) throw()
    {
        this->_M_validate(reinterpret_cast<std::size_t*>(__addr) - 1);
    }

    std::size_t* _M_get(std::size_t __sz) _GLIBCXX_THROW(std::bad_alloc);

    void _M_clear();
};


size_t* free_list::_M_get(size_t __sz) throw(std::bad_alloc)
{
    const vector_type& __free_list = _M_get_free_list();
    using __gnu_cxx::__detail::__lower_bound;
    //找到大于_sz最小区块，并返回其迭代器
    iterator __tmp = __lower_bound(__free_list.begin(), __free_list.end(), __sz,
_LT_pointer_compare());

    if (__tmp == __free_list.end() || !_M_should_i_give(**__tmp, __sz))  //如果没
有找到合适的区块
    {
        //__ctr为尝试次数，这里表示尝试两次请求内存
        int __ctr = 2;
        while (__ctr)
        {
            size_t* __ret = 0;
            --__ctr;
            __try
            {
                //使用::operator new请求内存（多请求一个size_t？？？）
                __ret = reinterpret_cast<size_t*> (::operator new(__sz +
sizeof(size_t)));
            }
            __catch(const std::bad_alloc&)
            {
                this->_M_clear();
            }
            if (!__ret)
                continue;
            *__ret = __sz;
            return __ret + 1;
        }

        //请求内存失败
```

```
            std::__throw_bad_alloc();
        }
        else  //找到的区块合适，从free list拿出并返回
        {
            size_t* __ret = *__tmp;
            _M_get_free_list().erase(__tmp);
            return __ret + 1;
        }
    }

    void free_list::_M_clear()
    {
        vector_type& __free_list = _M_get_free_list();
        iterator __iter = __free_list.begin();
        while (__iter != __free_list.end())
        {
            ::operator delete((void*)*__iter);
            ++__iter;
        }
        __free_list.clear();
    }
```

4.4 bitmap_allocator

```
    //bitmap_allocator主体，继承自free_list
    template<typename _Tp>
    class bitmap_allocator : private free_list
    {
    public:

        //STL allocator标准接口
        typedef std::size_t         size_type;
        typedef std::ptrdiff_t      difference_type;
        typedef _Tp* pointer;
        typedef const _Tp* const_pointer;
        typedef _Tp& reference;
        typedef const _Tp& const_reference;
        typedef _Tp                 value_type;

        template<typename _Tp1>
        struct rebind
        {
            typedef bitmap_allocator<_Tp1> other;
        };

    private:

        //_BSize被填充到_AlignSize倍数后的大小
        template<std::size_t _BSize, std::size_t _AlignSize>
        struct aligned_size
```

```
    {
        enum
        {
            modulus = _BSize % _AlignSize,
            value = _BSize + (modulus ? _AlignSize - (modulus) : 0)
        };
    };

    //alloc block (填充后, _BALLOC_ALIGN_BYTES默认8字节)
    struct _Alloc_block
    {
        char __M_unused[aligned_size<sizeof(value_type),
_BALLOC_ALIGN_BYTES>::value];
    };

    typedef typename std::pair<_Alloc_block*, _Alloc_block*> _Block_pair;
    typedef typename __detail::__mini_vector<_Block_pair> _BPVector;
    typedef typename _BPVector::iterator _BPiter;

    //从block pair vector中找到符合(!__p())条件的block pair并返回其迭代器
    template<typename _Predicate>
    static _BPiter _S_find(_Predicate __p)
    {
        _BPiter __first = _S_mem_blocks.begin();
        while (__first != _S_mem_blocks.end() && !__p(*__first))
            ++__first;
        return __first;
    }

     //分配一个super block
    void _S_refill_pool() _GLIBCXX_THROW(std::bad_alloc)
    {
        using std::size_t;

        //bitmap block数
        const size_t __num_bitmaps = (_S_block_size /
size_t(__detail::bits_per_block));
        //一个super block大小 (count+bitmap+alooc block三部分组成)
        const size_t __size_to_allocate = sizeof(size_t) + _S_block_size *
sizeof(_Alloc_block)    + __num_bitmaps * sizeof(size_t);

        //使用free_list::_M_get()获取一块内存
        size_t* __temp = reinterpret_cast<size_t*>(this-
>_M_get(__size_to_allocate));
        *__temp = 0;
        ++__temp;

        //设置新分配super block对应block pair
        _Block_pair __bp =  std::make_pair(reinterpret_cast<_Alloc_block*> (__temp
+ __num_bitmaps),
                reinterpret_cast<_Alloc_block*> (__temp + __num_bitmaps) +
_S_block_size - 1);

        //新block pair放入block pair vector
```

```
            _S_mem_blocks.push_back(__bp);

            //将bitmap block全部置1
            for (size_t __i = 0; __i < __num_bitmaps; ++__i)
                __temp[__i] = ~static_cast<size_t>(0); // 1 Indicates all Free.

            _S_block_size *= 2;
        }

        static _BPVector _S_mem_blocks;  //block pair vector
        static std::size_t _S_block_size;  //记录下次分配super block大小
        static __detail::_Bitmap_counter<_Alloc_block*> _S_last_request;  //上次分配
alloc block对应bitmap block位置
        static typename _BPVector::size_type _S_last_dealloc_index;  //上次释放alloc
block对应block pair索引

    public:

         //分配一个block
        pointer _M_allocate_single_object() _GLIBCXX_THROW(std::bad_alloc)
        {
            using std::size_t;

            //如果当前bitmap counter没有遍历完bitmap block，且当前bitmap block对应位全为
0。就移动bitmap block指针向后寻找
            while (_S_last_request._M_finished() == false && (*
(_S_last_request._M_get()) == 0))
                _S_last_request.operator++();

            //__builtin_expect的功能是分支预测，用于流水线加速。__builtin_expect(EXP, N)
表示EXP很可能为N
            //下面__builtin_expect含义是bitmap block分配完这个事件很可能为假
            if (__builtin_expect(_S_last_request._M_finished() == true, false))
            {
                // _FFF是函数对象，用于寻找block pair对应super block中第一个bitmap block
位非全0的块
                typedef typename __detail::_Ffit_finder<_Alloc_block*> _FFF;
                _FFF __fff;
                //_S_find从头开始遍历整个block pair向量找到第一个bitmap block非0的块，并
将__fff指向这个bitmap block
                _BPiter __bpi = _S_find(__detail::_Functor_Ref<_FFF>(__fff));

                if (__bpi != _S_mem_blocks.end())  //__bpi不指向block pair vector的末
尾，说明寻找成功
                {
                    //将对应可分配bitmap block置位（置为0）
                    size_t __nz_bit = _Bit_scan_forward(*__fff._M_get());
                    __detail::__bit_allocate(__fff._M_get(), __nz_bit);

                    _S_last_request._M_reset(__bpi - _S_mem_blocks.begin());

                    //__ret：获取分配block的地址
                    pointer __ret = reinterpret_cast<pointer> (__bpi->first +
__fff._M_offset() + __nz_bit);
```

```
                //__puse_count指向super block第一个size_t变量，记录super block分配的
块数
                size_t* __puse_count =  reinterpret_cast<size_t*> (__bpi->first) -
(__detail::__num_bitmaps(*__bpi) + 1);

                ++(*__puse_count);
                return __ret;
            }
            else
            {

                //分配新的super block并将其对应的block pair加入block pair vector
                _S_refill_pool();

                //指向新分配的super block
                _S_last_request._M_reset(_S_mem_blocks.size() - 1);
            }
        }

        //跟上面同样的分配操作
        size_t __nz_bit = _Bit_scan_forward(*_S_last_request._M_get());
        __detail::__bit_allocate(_S_last_request._M_get(), __nz_bit);

        pointer __ret = reinterpret_cast<pointer> (_S_last_request._M_base() +
_S_last_request._M_offset() + __nz_bit);

        size_t* __puse_count = reinterpret_cast<size_t*>
(_S_mem_blocks[_S_last_request._M_where()].first)
            - (__detail::__num_bitmaps(_S_mem_blocks[_S_last_request._M_where()])
+ 1);

        ++(*__puse_count);
        return __ret;
    }


     //释放一个block
    void _M_deallocate_single_object(pointer __p) throw()
    {
        using std::size_t;

        _Alloc_block* __real_p = reinterpret_cast<_Alloc_block*>(__p);

        typedef typename _BPVector::iterator _Iterator;
        typedef typename _BPVector::difference_type _Difference_type;

        _Difference_type __diff;  //__diff记录释放地址对应block pair在block pair
vector中偏移
        long __displacement;  //__displacement记录释放地址到对应super block第一个
block首地址的偏移量

        _GLIBCXX_DEBUG_ASSERT(_S_last_dealloc_index >= 0);

        //__ibt是一元谓词，用于判断传入的block pair是否包含__real_p
```

```
        __detail::_Inclusive_between<_Alloc_block*> __ibt(__real_p);
        if (__ibt(_S_mem_blocks[_S_last_dealloc_index]))  //先判断上次释放block所在
的block pair是否存在__real_p
        {
            _GLIBCXX_DEBUG_ASSERT(_S_last_dealloc_index <= _S_mem_blocks.size() -
1);

            __diff = _S_last_dealloc_index;
            __displacement = __real_p - _S_mem_blocks[__diff].first;
        }
        else
        {
            //_S_find遍历block pair vector找到符合__ibt条件的block pair
            _Iterator _iter = _S_find(__ibt);

            _GLIBCXX_DEBUG_ASSERT(_iter != _S_mem_blocks.end());

            __diff = _iter - _S_mem_blocks.begin();
            __displacement = __real_p - _S_mem_blocks[__diff].first;
            _S_last_dealloc_index = __diff;
        }


        //__rotate为释放block对应bitmap block中哪一位
        const size_t __rotate = (__displacement
            % size_t(__detail::bits_per_block));

        //__bitmapC为释放block对应哪个bitmap block
        size_t* __bitmapC = reinterpret_cast<size_t*>
(_S_mem_blocks[__diff].first) - 1;
        __bitmapC -= (__displacement / size_t(__detail::bits_per_block));

        __detail::__bit_free(__bitmapC, __rotate);
        //更新对应super block的分配数(*__puse_count)
        size_t* __puse_count = reinterpret_cast<size_t*>
(_S_mem_blocks[__diff].first) - (__detail::__num_bitmaps(_S_mem_blocks[__diff]) +
1);

        _GLIBCXX_DEBUG_ASSERT(*__puse_count != 0);

        --(*__puse_count);

        //如果super block对应分配块数归零，说明此super block已经全回收
        if (__builtin_expect(*__puse_count == 0, false))
        {
            _S_block_size /= 2;

            //将全回收super block插入free list
            this->_M_insert(__puse_count);
            //block pair vector中删除对应block pair
            _S_mem_blocks.erase(_S_mem_blocks.begin() + __diff);

            //这里需要更新_S_last_request，因为vector释放后，后面的block pair都要前移
            if ((_Difference_type)_S_last_request._M_where() >= __diff--)
```

```
                _S_last_request._M_reset(__diff);

            //同样的，当对应block pair释放后，需要将_S_last_dealloc_index重置
            if (_S_last_dealloc_index >= _S_mem_blocks.size())
            {
                _S_last_dealloc_index = (__diff != -1 ? __diff : 0);
                _GLIBCXX_DEBUG_ASSERT(_S_last_dealloc_index >= 0);
            }
        }
    }

public:

    //STL allocator标准接口
    bitmap_allocator() _GLIBCXX_USE_NOEXCEPT
    { }

    bitmap_allocator(const bitmap_allocator&) _GLIBCXX_USE_NOEXCEPT
    { }

    template<typename _Tp1>
    bitmap_allocator(const bitmap_allocator<_Tp1>&) _GLIBCXX_USE_NOEXCEPT
    { }

    ~bitmap_allocator() _GLIBCXX_USE_NOEXCEPT
    { }

    _GLIBCXX_NODISCARD pointer
        allocate(size_type __n)
    {
        if (__n > this->max_size())
            std::__throw_bad_alloc();

        if (__builtin_expect(__n == 1, true))
            return this->_M_allocate_single_object();
        else
        {
            const size_type __b = __n * sizeof(value_type);
            return reinterpret_cast<pointer>(::operator new(__b));
        }
    }

    _GLIBCXX_NODISCARD pointer
        allocate(size_type __n, typename bitmap_allocator<void>::const_pointer)
    {
        return allocate(__n);
    }

    void
        deallocate(pointer __p, size_type __n) throw()
    {
        if (__builtin_expect(__p != 0, true))
        {
```

```
            if (__builtin_expect(__n == 1, true))
                this->_M_deallocate_single_object(__p);
            else
                ::operator delete(__p);
        }
    }

    pointer
        address(reference __r) const _GLIBCXX_NOEXCEPT
    {
        return std::__addressof(__r);
    }

    const_pointer
        address(const_reference __r) const _GLIBCXX_NOEXCEPT
    {
        return std::__addressof(__r);
    }

    size_type
        max_size() const _GLIBCXX_USE_NOEXCEPT
    {
        return size_type(-1) / sizeof(value_type);
    }

    void
        construct(pointer __p, const_reference __data)
    {
        ::new((void*)__p) value_type(__data);
    }

    void
        destroy(pointer __p)
    {
        __p->~value_type();
    }

};
```

```
template<typename _Tp1, typename _Tp2>
bool operator==(const bitmap_allocator<_Tp1>&,
    const bitmap_allocator<_Tp2>&) throw()
{
    return true;
}

//静态成员定义
template<typename _Tp>
typename bitmap_allocator<_Tp>::_BPVector
bitmap_allocator<_Tp>::_S_mem_blocks;
```

```
template<typename _Tp>
std::size_t bitmap_allocator<_Tp>::_S_block_size
= 2 * std::size_t(__detail::bits_per_block);

template<typename _Tp>
typename bitmap_allocator<_Tp>::_BPVector::size_type
bitmap_allocator<_Tp>::_S_last_dealloc_index = 0;

template<typename _Tp>
__detail::_Bitmap_counter
<typename bitmap_allocator<_Tp>::_Alloc_block*>
bitmap_allocator<_Tp>::_S_last_request(_S_mem_blocks);
```

5 bitmap_allocator改进

5.1 bitmap_allocator改进分析

尽管已经看到了bitmap allocator的全貌，但是不应该把它当作一个整体看待。显然为了使分配器提供不同的特性，bitmap allocator中集成了不同的解决方法。下面进行详细分析：

bitmap allocator可以看作两个层次：上层是bitmap allocator主体，由block pair vector和super block组成。这里为了区分名称，将这两部分统一称为bitmap controller。底层是free list空闲链表，用于为上层bitmap controller分配或回收内存块，同时从下层操作系统索取或归还内存块。

free list从功能上分析，简单说其只做了一件事：deferring。即在bitmap controller和底层操作系统间形成缓冲。避免频繁的使用系统调用获取内存块。因此如果希望你的内存分配器拥有deferring功能，可以参考此设计。

free list从实现上分析，由于free list主要操作是在列表中内存块的删除与插入，同时需要维护内存块的有序，因此从这些特点来看vector并不是好的选择。二叉搜索树或者红黑树更为合适。但是从整体看，free list的性能对bitmap allocator影响占比不大。大部分alloc block操作还是由bitmap controller管理。

bitmap controller从功能上分析其提供了这么几个机制：首先将alloc block使用super block管理，同时使用counter记录super block中alloc block的分配情况实际上提供了可以将分配内存返还操作系统的机制，当一个super block所有alloc block都被回收，就可以将整个super block返还操作系统。如果希望你的内存分配器拥有返还操作系统内存的功能，可以参考此设计。其次使用bitmap来记录alloc block分配情况可以提供快速随机访问分配检查的功能，bitmap使用也使得删除一个alloc block和反转一个位一样简单（但是其中还涉及计算alloc block对应的bitmap block位置），如果希望你的分配器拥有随机访问分配检查，可以参考bitmap的设计。另外还有两个trick值得一提，第一个就是在连续分配的过程中block pair vector大小和super block的大小都会增长，尽管按几倍增长没有实际论证，但是直观上是合理的。第二个是记录分配和释放过程中的alloc block位置，同样从直观上看利用了局部性原理，会提高分配器的性能。

bitmap controller从实现上分析，首先应该明确bitmap实现在什么情况下有意义，bitmap是用位图记录不同对象状态。个人认为在两种情况下bitmap可以被很好的运用，一是当直接判断一个对象状态比较困难时，如当对象是一条链表，我需要记录链表中节点是否够32个，这时直接判断对象是否符合条件需要较大开销，因此可以直接使用bitmap记录对象当前状态。二是需要支持随机的对象状态判断，显然位图是连续的01序列，因此对于随机状态判断很方便。那么分配器需要这两种特性吗？显然并不是必须的。在基于链表的内存分配器中，每次内存分配只需要在链表头部取内存块返回用户即可，并不需要做随机访问分配检查。

最后bitmap allocator的分配单位是固定大小的，同时每次只能分配1单元的alloc block，显然bitmap allocator不适合vector或deque之类的需要大块连续内存的容器，更适合list、RB-Tree这样多量小块的内存分配需求。