容器和算法

参考网址

请你来说一下map和set有什么区别，分别又是怎么实现的?

map和set都是C++的关联容器，其底层实现都是红黑树（RB-Tree）。由于 map 和set所开放的各种操作接口，RB-tree 也都提供了，所以几乎所有的 map 和set的操作行为，都只是转调 RB-tree 的操作行为。

map和set区别在于：

（1）map中的元素是key-value（关键字—值）对：关键字起到索引的作用，值则表示与索引相关联的数据；Set与之相对就是关键字的简单集合，set中每个元素只包含一个关键字。

（2）set的迭代器是const的，不允许修改元素的值；map允许修改value，但不允许修改key。其原因是因为map和set是根据关键字排序来保证其有序性的，如果允许修改key的话，那么首先需要删除该键，然后调节平衡，再插入修改后的键值，调节平衡，如此一来，严重破坏了map和set的结构，导致iterator失效，不知道应该指向改变前的位置，还是指向改变后的位置。所以STL中将set的迭代器设置成const，不允许修改迭代器的值；而map的迭代器则不允许修改key值，允许修改value值。

（3）map支持下标操作，set不支持下标操作。map可以用key做下标，map的下标运算符[ ]将关键码作为下标去执行查找，如果关键码不存在，则插入一个具有该关键码和mapped_type类型默认值的元素至map中，因此下标运算符[ ]在map应用中需要慎用，const_map不能用，只希望确定某一个关键值是否存在而不希望插入元素时也不应该使用，mapped_type类型没有默认值也不应该使用。如果find能解决需要，尽可能用find。

Map和Set的简易实现

前面两篇博客我分别介绍了Map容器的使用以及Set容器的使用，我们了解到它的用法以及明白了它的底部实现其实就是红黑树. 这个时候可能有人就不

理解了.底层是红黑树Map为什么可以存储一个主键一个键值. 而Set只拥有一个主键.这里就是STL的强大之处. 利用巧妙地方法极大的提升了代码的

复用.当然在实现之前我也看了Map和Set的源码. 我这里实现的只是最简易的Map和Set，并没有很多复杂的功能. 可以插入，删除,迭代器的加减操作.好了那我们进入正题.

首先思考第一个问题. Map和Set的元素个数不同.当然如果你说我给Map设计一个红黑树容器，再给Set设计一个红黑树容器.这样没问题，但是你的代码

是不是但有点太长了.如果这里不是红黑树是一个很大的项目呢？ 所以需要使用同一个红黑树的底层容器来提升代码复用.当然你可以让Map在这里传入

一个pair<>，让Set在这里传入一个Key，这个思想没问题.但红黑树里面需要用到键值之间的比较，Set在这里可以直接使用Key进行比较.但是Map的

pair<K,V>需要使用其中的.first来进行比较？ 那么这里应该如何实现呢？ 这里需要用到仿函数的知识，以及巧妙的运用模板. 下面我贴出来代码

还有一张过程步骤图帮大家理解这个过程.

Map的封装代码:

```
#include"RBTree.h"

template<class K,class V>
class MakeMap
{

public:
    typedef pair<K, V> ValueType;

    struct KeyOfValue
    {
        const K& operator()(const ValueType& kv)
        {
            return kv.first;
        }
    };

    typedef typename RBTree<K, ValueType,KeyOfValue>::Iterator Iterator;

    pair<Iterator, bool> Insert(const ValueType& v)
    {
        return _Tree.Insert(v);
    }

    V& operator[](const K& key)
    {
        pair<Iterator,bool> ret = _Tree.Insert(make_pair(key, V()));
        //模板参数的V() 缺省值.
        return ret.first;
    }
    Iterator Begin()
    {
        return _Tree.Begin();
    }

    Iterator End()
    {
        return _Tree.End();
    }
private:
    RBTree<K, ValueType, KeyOfValue> _Tree;
};

void Test()
{
    MakeMap<string, string> dict;

    dict.Insert(make_pair("liangliang", "亮亮"));
    dict.Insert(make_pair("MT", "梦婷"));
    dict.Insert(make_pair("Steam", "蓝洞"));
    dict.Insert(make_pair("type", "字节"));
```

```
    MakeMap<string, string>::Iterator it = dict.Begin();

    while (it != dict.End())
    {
        cout << it->second << " ";
        ++it;
    }
}
```

Set的封装代码:

```
#include"RBTree.h"

template<class K>
class mySet
{
public:
    typedef K ValueType;
    struct KeyOfKey
    {
        const ValueType& operator()(const ValueType& key)
        {
            return key;
        }
    };

    typedef typename RBTree<K, K,KeyOfValue>::Iterator Iterator;
    //如果没有typename，编译器就会去RBTree里面去寻找Iterator.但是RBTree并没有实例化，所以会找不到
    //然后报错. 所以typename告诉编译器这个类型是一个模板的类型，现在先不要确定它的类型.

    pair<Iterator, bool>insert(const K& key)
    {
        return Tree.Insert(key);
    }

    Iterator Begin()
    {
        return Tree.Begin();
    }

    Iterator End()
    {
        return Tree.End();
    }




protected:

    RBTree<K, ValueType, KeyOfKey> Tree;
```

```
};


void Test()
{
    mySet<int> T;

    T.insert(1);
    T.insert(2);
    T.insert(3);
    T.insert(4);
    T.insert(5);
    T.insert(6);
    T.insert(7);

    mySet<int>::Iterator it = T.Begin();

    while (it != T.End())
    {
        cout << *it << " ";
        ++it;
    }

    cout << endl;

}
```

RBTree的实现代码:

```
#include<iostream>
#include<Windows.h>
#include<string>
#include<assert.h>
using namespace std;

enum colour
{
    RED,
    BLACK
};

template<class ValueType>
struct RBTreeNode
{
    ValueType _valueField;
    RBTreeNode<ValueType>* _left;
    RBTreeNode<ValueType>* _right;
    RBTreeNode<ValueType>* _parent;

    colour _col;

    RBTreeNode(const ValueType& v)
        :_valueField(v)
```

```
        , _left(NULL)
        , _right(NULL)
        , _parent(NULL)
        , _col(RED)
    {}
};

template<class ValueType>
struct __RBtreeIteartor
{
    typedef RBTreeNode<ValueType> Node;
    typedef __RBtreeIteartor<ValueType> self;

public:

    __RBtreeIteartor(Node* node)
        :_node(node)
    {}

    __RBtreeIteartor(const self& node)
        :_node(node._node)
    {}

    ValueType& operator*()
    {
        return _node->_valueField;
    }

    ValueType* operator->()
    {
        return &operator*();
    }

    self& operator=(const self& node)
    {
        _node = node._node;
    }

    self& operator++()
    {
        //1.如果右不为空，访问右树的最左节点
        //2.如果我的右为空，下一个访问的就是沿着这个路径往上找，第一个右树不是我的节点
        //然后访问该节点.
        if (_node->_right)
        {
            Node* subR = _node->_right;
            while (subR->_left)
            {
                subR = subR->_left;
            }
            _node = subR;
        }
        else
```

```
        {
            Node* cur = _node;
            Node* parent = cur->_parent;
            while (parent && cur == parent->_right)
            {
                cur = parent;
                parent = cur->_parent;
            }
            _node = parent;
        }
        return *this;
    }

    self& operator--()
    {
        if (_node->_left)
        {
            Node* subL = _node->_left;
            while (subL->_right)
            {
                subL = subL->_right;
            }
            _node = subleft;
        }
        else
        {
            Node* cur = _node;
            Node* parent = cur->_parent;
            while (parent && cur == parent->_left)
            {
                cur = parent;
                parent = cur->_parent;
            }
            _node = parent;
        }
        return *this;
    }

    bool operator==(const self& s)
    {
        return _node == s._node;
    }

    bool operator!=(const self& s)
    {
        return _node != s._node;
    }

private:
    Node* _node;

};
```

```
template<class K, class V,class KeyOfValue>
class RBTree
{
    typedef  V ValueType;
    typedef RBTreeNode<ValueType> Node;

public:

    typedef __RBtreeIteartor<ValueType> Iterator;

    RBTree()
        :_root(NULL)
    {}
    Iterator Begin()
    {
        Node* cur = _root;
        while (cur && cur->_left != NULL)
        {
            cur = cur->_left;
        }

        return Iterator(cur);
    }

    Iterator End()
    {
        return Iterator(NULL);
    }

    pair<Iterator,bool> Insert(const ValueType& v)
    {
        //_Insert(_root, x, y);
        if (_root == NULL)
        {
            _root = new Node(v);
            _root->_col = BLACK;
            return make_pair(Iterator(_root), true);
        }

        KeyOfValue keyofvalue;

        Node* cur = _root;
        Node* parent = cur;

        while (cur)
        {
            if (keyofvalue(cur->_valueField) > keyofvalue(v))
            {
                parent = cur;
                cur = cur->_left;
            }
            else if (keyofvalue(cur->_valueField) < keyofvalue(v))
            {
```

```
            parent = cur;
            cur = cur->_right;
        }
        else if (keyofvalue(cur->_valueField) == keyofvalue(v))
        {
            return make_pair(Iterator(cur), false);
        }
    }

    if (keyofvalue(parent->_valueField) > keyofvalue(v))
    {
        parent->_left = new Node(v);
        parent->_left->_parent = parent;
        cur = parent->_left;
    }
    else
    {
        parent->_right = new Node(v);
        parent->_right->_parent = parent;
        cur = parent->_right;
    }

    Node* newNode = cur;
    //目前父亲节点，插入节点，叔叔节点已经就绪.
    while (parent && parent->_col == RED)
    {
        Node* parentparent = parent->_parent;
        Node* uncle = NULL;

        if (parentparent->_left == parent)
            uncle = parentparent->_right;

        else
            uncle = parentparent->_left;

        if (uncle && uncle->_col == RED)
        {
            parentparent->_col = RED;
            parent->_col = BLACK;
            uncle->_col = BLACK;

            cur = parentparent;
            parent = cur->_parent;

        }
        else if (uncle == NULL || uncle->_col == BLACK)
        {
            if (parentparent->_left == parent)
            {
                if (parent->_left == cur)
                {
                    RotateR(parentparent);
                    parent->_col = BLACK;
```

```
                }
                else
                {
                    RotateLR(parentparent);
                    cur->_col = BLACK;
                }
            }
            else
            {
                if (parent->_right == cur)
                {
                    RotateL(parentparent);
                    parent->_col = BLACK;
                }
                else
                {
                    RotateRL(parentparent);
                    cur->_col = BLACK;
                }
            }
            parentparent->_col = RED;
            if (parentparent == _root)
            {
                _root = parent;
            }

        }
        else
        {
            assert(false);
        }
    }
    _root->_col = BLACK;
    return make_pair(Iterator(newNode), true);
    //担心经过旋转之后，找不到新增节点了,所以提前记录好.
}



Iterator Find(const K& key)
{
    Node* cur = _root;
    while (cur)
    {
        if (keyofvalue(cur->_valueField) > keyofvalue(key))
        {
            cur = cur->_right;
        }
        else if (keyofvalue(cur->_valueField) < keyofvalue(key))
        {
            cur = cur->_left;
        }
```

```
            else if (keyofvalue(cur->_valueField) == keyofvalue(key))
            {
                return Iterator(cur);
            }
        }
        return Iterator(NULL);
    }

protected:

    void RotateLR(Node*& parent)
    {

        RotateL(parent->_left);

        RotateR(parent);
    }

    void RotateRL(Node*& parent)
    {
        RotateR(parent->_right);
        RotateL(parent);
    }

    void RotateR(Node*& parent)
    {
        Node* subL = parent->_left;
        Node* subLR = subL->_right;

        parent->_left = subLR;
        if (subLR)
            subLR->_parent = parent;
        Node* ppNode = parent->_parent;
        subL->_right = parent;
        parent->_parent = subL;
        if (ppNode == NULL)
        {
            _root = subL;
            _root->_parent = NULL;
        }
        else
        {
            if (ppNode->_left == parent)
                ppNode->_left = subL;
            else
                ppNode->_right = subL;
            subL->_parent = ppNode;
        }
    }

    void RotateL(Node*& parent)
    {
        Node* subR = parent->_right;
```

```
        Node* subRL = subR->_left;

        parent->_right = subRL;
        if (subRL)
            subRL->_parent = parent;

        Node* ppNode = parent->_parent;
        subR->_left = parent;
        parent->_parent = subR;

        if (ppNode == NULL)
        {
            _root = subR;
            _root->_parent = NULL;
        }
        else
        {
            if (ppNode->_left == parent)
                ppNode->_left = subR;
            else
                ppNode->_right = subR;
            subR->_parent = ppNode;
        }
    }
private:
    Node* _root;
};
```

上面的代码我们可以看到. Set和Map其实就是一层马甲，对他们来说只是封装了底层的红黑树，只不过他们传入红黑树KeyOfValue的参数不同.而后面的KeyOfValue才是决定，RBTreeNode当中的*valueField与key比较时，返回的是Key还是pair<>.first. 我下面还有一幅图用来帮大家理解这整个复用代码的框架. 大家仔细看一定会明白这里的KeyOfValue模板参数，以及*ValueType的作用. 如果理解这些那么Map和Set的简易实现应该就差不多了.

```
template<class K,class V>
class Map
{
public:
	typedef pair<K, V> ValueType;
	struct KeyOfValue
	{
		const K& operator()(const ValueType& kv)
		{
			return kv.first;
		}
	};
private:
	RBTree<K, ValueType, KeyOfValue> _Tree;
```

这里的底层复用原理就是ValueType，的功劳．首先我们发现RBTree底层接受数据虽然接收了一个K一个V.但是呢，它里面实际运用其实只使用V．这里我们再看看map,set的封装．看他们都给V模板参数传递的是什么？

Set -> V -> K;
Map -> V -> piar<K,V>

这里你从他们底层构建的那个RBTree传参就可以清晰的看到．所以这里巧妙地解决了复用的问题．

```
template<class K>
class mySet
{
public:
	typedef K ValueType;
	struct KeyOfKey
	{
		const ValueType& operator()(const ValueType& key)
		{
			return key;
		}
	};
protected:
	RBTree<K, ValueType, KeyOfKey> Tree;
```

ValueType ->pair<K,V>

ValueType->K

```
template<class K, class V, class KeyOfValue>
	class RBTree
	{
		typedef  V ValueType;
		typedef RBTreeNode<ValueType> Node;

	public:

		typedef __RBtreeIteartor<ValueType> Iterator;

private:
	Node* _root;
```

你的RBTreeNode.里面只剩下_valueField但如果你是map，_valueField就是一个pair.而你是set，_valueField就是一个key．这里存在一个潜在问题．RBTree的原理其实需要用到大小比较，但是很显然当你传入一个key的时候，假如你是map类型的，那么你的pair<K,V>无法匹配的上人家的key,导致无法比较．所以这个时候我们开始介绍我们传进来的另外一个模板参数->KeyOfValue．它的作用就是利用仿函数来解决这种私人订制的问题．利用仿函数，让pair的_valueField返回一个key．让set的_valueField也返回一个key.

```
template<class ValueType>
struct RBTreeNode
{
	ValueType _valueField;
	RBTreeNode<ValueType>* _left;
	RBTreeNode<ValueType>* _right;
	RBTreeNode<ValueType>* _parent;

	colour _col;

	RBTreeNode(const ValueType& v)
		:_valueField(v)
		, _left(NULL)
		, _right(NULL)
		, _parent(NULL)
		, _col(RED)

template<class ValueType>
struct __RBtreeIteartor
{
	typedef RBTreeNode<ValueType> Node;
	typedef __RBtreeIteartor<ValueType> self;
private:
	Node*  node;

};
```

```
pair<Iterator,bool> Insert(const ValueType& v)
{
	...
	KeyOfValue keyofvalue;

	Node* cur = _root;
	Node* parent = cur;

	while (cur)
	{
		if (keyofvalue(cur->_valueField) > keyofvalue(v))
		{
			parent = cur;
			cur = cur->_left;
		}
		else if (keyofvalue(cur->_valueField) < keyofvalue(v))
		{
			parent = cur;
			cur = cur->_right;
		}
		else if (keyofvalue(cur->_valueField) == keyofvalue(v))
		{
			return make_pair(Iterator(cur), false);
		}
	}
}
```

在这里由传入的KeyOfValue来确定到底应该调用哪一个operator()

最后这里，如果传进来的KeyOfValue如果KeyOfKey那么keyofvalue(cur->_valueField)就是直接返回key．如果传入的是KeyOfValue那么keyofvalue(cur->_valueField)返回的值就是._valueField.first．所以这里利用仿函数完美解决掉比较的问题．到现在一个RBTree底层可以给两个不同类型容器使用的复用代码就完成了．注意里面的仿函数应用，以及ValueType这种方法.很实用的

接下来我着重解释一下红黑树的迭代器，因为上一篇博客红黑树当中只是简单构建出来一颗红黑树，并没有对它的迭代器进行实现，那么现在我们继续

了解它的迭代器:

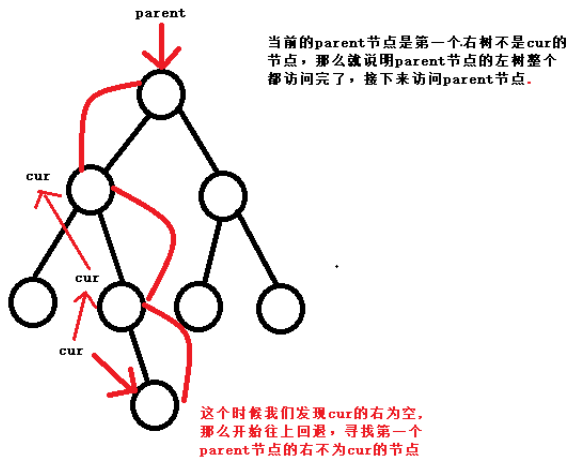
首先迭代器就是封装一层指针，让我们能够方便的遍历每一个容器.使用相同的方法. 也就是说我们不需要知道容器的底层实现. 就会遍历这个容器.

所以迭代器就是增加了封装性.给外边暴露一个接口，让你只会用就行了，不用知道为什么.红黑树的迭代器的operator*，operator->已经是老生常谈

了，所以我们今天的重点是明白迭代器的operator++()以为这是一个算法-> 因为红黑树遍历是中序遍历，所以我们只需要帮迭代器找到它下一个需要

访问的节点即可. 首先中序遍历的顺序是 左 中 右. 而我们这个算法就是找到中序遍历的下一个节点位置:

```
self& operator++()
	{
		//1.如果右不为空，访问右树的最左节点
		//2.如果我的右为空，下一个访问的就是沿着这个路径往上找，第一个右树不是我的节
		//然后访问该节点.
		if (_node->_right)
		{
			Node* subR = _node->_right;
			while (subR->_left)
			{
				subR = subR->_left;
			}
			_node = subR;
		}
		else
		{
			Node* cur = _node;
			Node* parent = cur->_parent;
			while (parent && cur == parent->_right)
			{
				cur = parent;
				parent = cur->_parent;
			}
			_node = parent;
		}
		return *this;
	}
```



当然operator--我们只需要颠倒一下就可以~ 这个算法需要自己自己走一走过程就会理解他为什么这么吊！！！！！更多红黑树的知识我们可以去看

一下我的上一个博客. 当然map和set肯定不仅仅只有红黑树版本的. 我们以后还会有hash版本的map和set.其实代码复用的原理都一样. 只不过后面

hash的模板结构嵌套更加复杂一点. 但是只要你理解了之后，自己也就会设计出来这些结构. Map和Set的结构我们只要好好地认识上面的图就可以好好

请你来介绍一下STL的allocaotr

STL的分配器用于封装STL容器在内存管理上的底层细节。在C++中，其内存配置和释放如下：

new运算分两个阶段：(1)调用::operator new配置内存;(2)调用对象构造函数构造对象内容

delete运算分两个阶段：(1)调用对象希构函数；(2)掉员工::operator delete释放内存

为了精密分工，STL allocator将两个阶段操作区分开来：内存配置有alloc::allocate()负责，内存释放由alloc::deallocate()负责；对象构造由::construct()负责，对象析构由::destroy()负责。

同时为了提升内存管理的效率，减少申请小内存造成的内存碎片问题，SGI STL采用了两级配置器，当分配的空间大小超过128B时，会使用第一级空间配置器；当分配的空间大小小于128B时，将使用第二级空间配置器。第一级空间配置器直接使用malloc()、realloc()、free()函数进行内存空间的分配和释放，而第二级空间配置器采用了内存池技术，通过空闲链表来管理内存。

空间配置器，由两级分配器构成，大于128字节，调用一级配置器，malloc/free，realloc；小于128字节，默认二级配置器，分配内存池。为了便于内存管理，减少内存碎片产生

为一些泛型容器分配内存 使用户不必担心内存问题 只要添加数据即可

简单来说，是为了各种泛型容器如vector，map等分配内存，使程序员不比为内存而担心，只需添加数据即可

参考网址 2 详细

一般而言，我们习惯的 C++ 内存配置操作和释放操作是这样的：

```
1 class FOO{};
2 FOO *pf = new FOO;
3 delete pf;
```

我们看其中第二行和第三行，虽然都是只有一句，当是都完成了两个动作。但你 new 一个对象的时候两个动作是：先调用::operator new 分配一个对象大小的内存，然后在这个内存上调用FOO::FOO()构造对象。同样，当你 delete 一个对象的时候两个动作是：先调用FOO::~FOO() 析构掉对象，再调用::operator delete将对象所处的内存释放。为了精密分工，STL 将allocator决定将这两个阶段分开。分别用 4 个函数来实现：

1.内存的配置：alloc::allocate();

2.对象的构造：::construct();

3.对象的析构：::destroy();

4.内存的释放：alloc::deallocate();

其中的 construct() 和 destroy()定义在 STL的库文件中，源代码如下：

```
template <class T>
inline void destroy(T* pointer) {
    pointer->~T();                                  //只是做了一层包装，将指针所指的对象析构---通过直接调用类的析构函数
}

template <class T1, class T2>
inline void construct(T1* p, const T2& value) {
  new (p) T1(value);                                //用placement new在 p 所指的对象上创建一个对象，value是初始化对象的值。
}

template <class ForwardIterator>                    //destory的泛化版，接受两个迭代器为参数
inline void destroy(ForwardIterator first, ForwardIterator last) {
  __destroy(first, last, value_type(first));        //调用内置的 __destory(),value_type()萃取迭代器所指元素的型别
```

```
}

template <class ForwardIterator, class T>
inline void __destroy(ForwardIterator first, ForwardIterator last, T*) {
  typedef typename __type_traits<T>::has_trivial_destructor trivial_destructor;
  __destroy_aux(first, last, trivial_destructor());        //trival_destructor()相当于用来判断迭代器所指型别是否有 trival destructor
}


template <class ForwardIterator>
inline void                                                //如果无 trival destructor ，那就要调用destroy()函数对两个迭代器之间的对象元素进行一个个析构
__destroy_aux(ForwardIterator first, ForwardIterator last, __false_type) {
  for ( ; first < last; ++first)
    destroy(&*first);
}

template <class ForwardIterator>                          //如果有 trival destructor ，则什么也不用做。这更省时间
inline void __destroy_aux(ForwardIterator, ForwardIterator, __true_type) {}

inline void destroy(char*, char*) {}          //针对 char * 的特化版
inline void destroy(wchar_t*, wchar_t*) {}    //针对 wchar_t*的特化版
```

看到上面这么多代码，大家肯定觉得 construct() 和 destroy() 函数很复杂。其实不然，我们看到construct()函数只有几行代码。而 destroy() 稍微多点。但是这么做都是为了提高销毁对象时的效率。为什么要判断迭代器所指型别是否有 trival destructor，然后分别调用不同的执行函数？因为当你要销毁的对象很多的时候，而这样对象的型别的 destructor 都是 trival 的。如果都是用**destroy_aux(ForwardIterator first, ForwardIterator last,** false_type)来进行销毁的话很费时间，因为没必要那样做。而当你对象的destructor 都是 non-trival 的时候，你又必须要用**destroy_aux(ForwardIterator first, ForwardIterator last,** false_type)来析构。所以，我们要判断出对象型别的 destructor 是否为 trival，然后调用不同的__destroy_aux。

说完 construct() 和 destory() ，我们来说说 alloc::allocate() 和 alloc::deallocate()，其源代码在 <stl_alloc.h> 中。stl_alloc.h中代码设计的原则如下：

1.向 system heap 要求空间

2.考虑多线程状态

3.考虑内存不足时的应变措施

4.考虑过多“小型区块”可能造成的内存碎片问题。

stl_alloc.h中的代码相当复杂，不过没关系。我们今天只看其中的allocate() 和 deallocate()。在讲这两个函数之前，我们还必须来了解一下SGI STL(SGI限定词是STL的一个版本，因为真正的STL有很多不同公司实现的版本，我们所讨论的都是SGI版本) 配置器的工作原理：

考虑到小型区块可能造成内存破碎问题(即形成内存碎片)，SGI STL 设计了双层级配置器。第一层配置器直接使用malloc() 和 free().第二层配置器则视情况采用不同的策略：但配置区块超过 128 bytes时，调用第一级配置器。当配置区块小于 128 bytes时，采用复杂的 memory pool 方式。下面我们分别简单的介绍一下第一级和第二级配置器：

第一级配置器 _ _malloc_alloc_template：

由于第一级配置器的配置方法比较简单，代码也容易理解，我在这里全部贴出：

```
//以下是第第一级配置器
template <int inst>
class __malloc_alloc_template {

private:

//以下函数用来处理内存不足的情况
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)
{
    void *result = malloc(n);                    //第一级配置器，直接使用malloc()
    //如果内存不足，则调用内存不足处理函数oom_alloc()来申请内存
    if (0 == result) result = oom_malloc(n);
    return result;
}

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

static void * reallocate(void *p, size_t /* old_sz */, size_t new_sz)
{
    void * result = realloc(p, new_sz);            //第一级配置器直接使用realloc()
    //当内存不足时，则调用内存不足处理函数oom_realloc()来申请内存
    if (0 == result) result = oom_realloc(p, new_sz);
    return result;
}

//设置自定义的out-of-memory handle就像set_new_handle()函数
static void (* set_malloc_handler(void (*f)()))()
{
    void (* old)() = __malloc_alloc_oom_handler;
    __malloc_alloc_oom_handler = f;
    return(old);
}
};

template <int inst>
void (* __malloc_alloc_template<inst>::__malloc_alloc_oom_handler)() = 0;    //内存处理函数指针为空，等待客户端赋值

template <int inst>
void * __malloc_alloc_template<inst>::oom_malloc(size_t n)
```

```
{
    void (* my_malloc_handler)();
    void *result;

    for (;;) {                                                  //死循环
        my_malloc_handler = __malloc_alloc_oom_handler;         //设定自己的oom(out of
memory)处理函数
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }      //如果没有设定自己的oom处理
函数，毫不客气的抛出异常
        (*my_malloc_handler)();                                 //设定了就调用oom处理函数
        result = malloc(n);                                     //再次尝试申请
        if (result) return(result);
    }
}

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; }    //如果自己没有定义oom处理函数，则
编译器毫不客气的抛出异常
        (*my_malloc_handler)();                                //执行自定义的oom处理函数
        result = realloc(p, n);                                //重新分配空间
        if (result) return(result);                            //如果分配到了，返回指向内存的指针
    }
}
```

上面代码看似繁杂，其实流程是这样的：

1.我们通过allocate()申请内存，通过deallocate()来释放内存，通过reallocate()重新分配内存。

2.当allocate()或reallocate()分配内存不足时会调用oom_malloc()或oom_remalloc()来处理。

3.当oom_malloc() 或 oom_remalloc()还是没能分配到申请的内存时，会转如下两步中的一步：

a).调用用户自定义的内存分配不足处理函数(这个函数通过set_malloc_handler() 来设定)，然后继续申请内存！

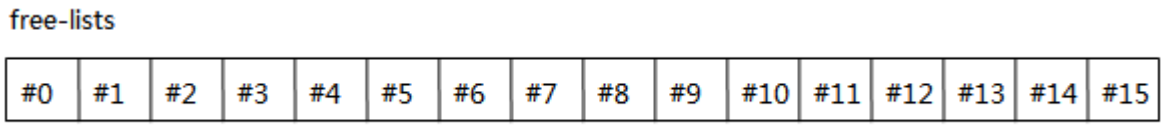
b).如果用户未定义内存分配不足处理函数，程序就会抛出bad_alloc异常或利用exit(1)终止程序。

看完这个流程，再看看上面的代码就会容易理解多了！

第二级配置器 _ _default_alloc_template：

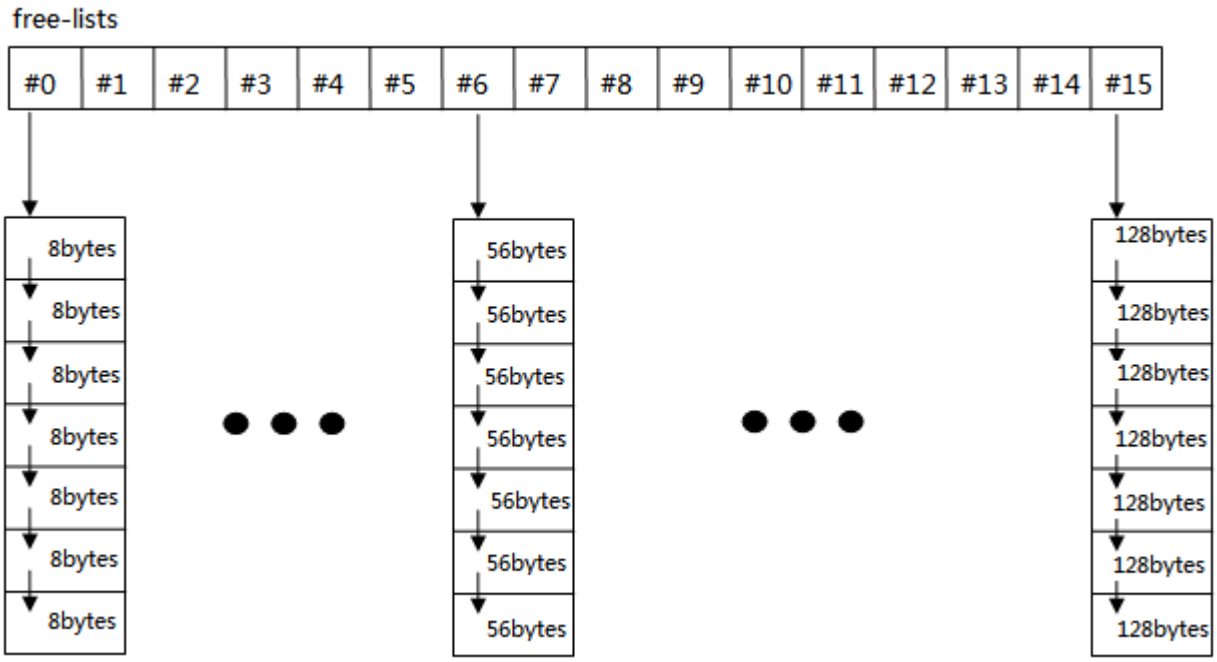
第二级配置器的代码很多，这里我们只贴出其中的 allocate() 和 dellocate()函数的实现和工作流程(参考侯捷先生的《STL源码剖析》)，而在看函数实现代码之前，我大致的描述一下第二层配置器配置内存的机制。

我们之前说过，当申请的内存大于 128 bytes时就调用第一层配置器。当申请的内存小于 128bytes时才会调用第二层配置器。第二层配置器如何维护128bytes一下内存的配置呢？ SGI 第二层配置器定义了一个 free-lists,这个free-list是一个数组，如下图：



这数组的元素都是指针，用来指向16个链表的表头。这16个链表上面挂的都是可以用的内存块。只是不同链表中元素的内存块大小不一样，16个链表上分别挂着大小为

8,16,24,32,40,48,56,64,72,80,88,96,104,112,120,128 bytes的小额区块，图如下：

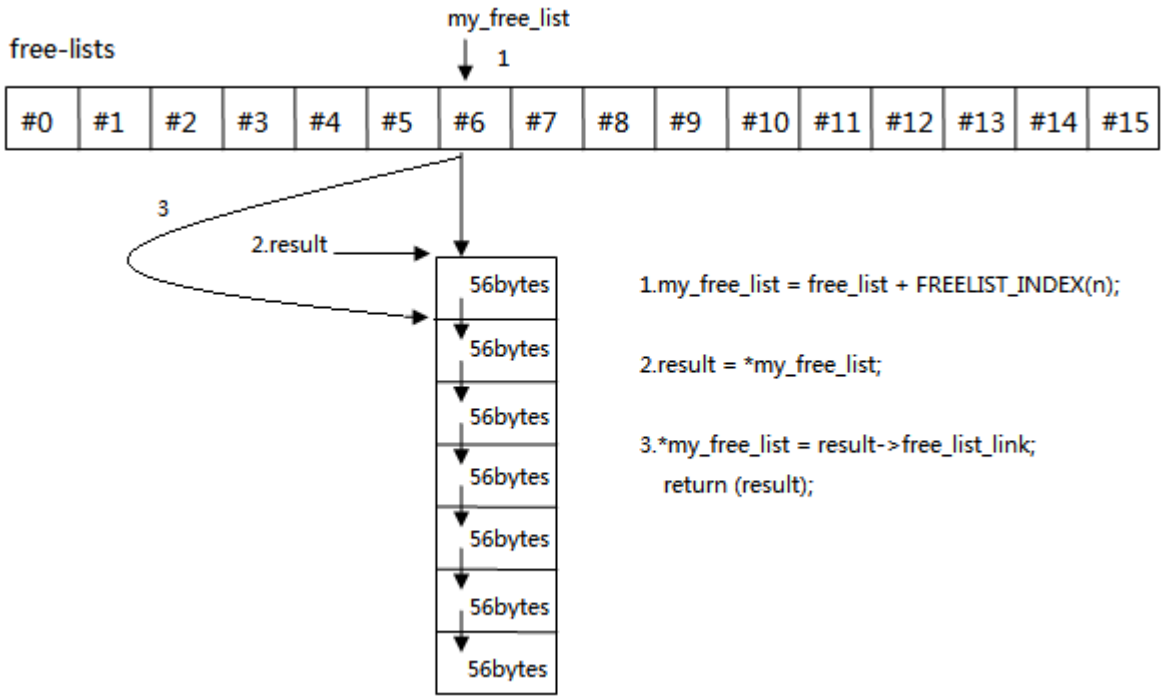


就是这样，现在我们来看allocate()代码：

```
static void * allocate(size_t n)
 {
   obj * __VOLATILE * my_free_list;
   obj * __RESTRICT result;

   //要申请的空间大于128bytes就调用第一级配置
   if (n > (size_t) __MAX_BYTES) {
       return(malloc_alloc::allocate(n));
   }
   //寻找 16 个free lists中恰当的一个
   my_free_list = free_list + FREELIST_INDEX(n);
   result = *my_free_list;
   if (result == 0) {
       //没找到可用的free list，准备新填充free list
       void *r = refill(ROUND_UP(n));
       return r;
   }
   *my_free_list = result -> free_list_link;
   return (result);
 };
```

其中有两个函数我来提一下，一个是ROUND_UP()，这个是将要申请的内存字节数上调为8的倍数。因为我们free-lists中挂的内存块大小都是8的倍数嘛，这样才知道应该去找哪一个链表。另一个就是refill()。这个是在没找到可用的free list的时候调用，准备填充free lists.意思是：参考上图，假设我现在要申请大小为 56bytes 的内存空间，那么就会到free lists 的第 7 个元素所指的链表上去找。如果此时 #7元素所指的链表为空怎么办？这个时候就要调用refill()函数向内存池申请N(一般为20个)个大小为56bytes的内存区块，然后挂到 #7 所指的链表上。这样，申请者就可以得到内存块了。当然，这里为了避免复杂，误导读者我就不讨论refill()函数了。allocate()过程图如下：

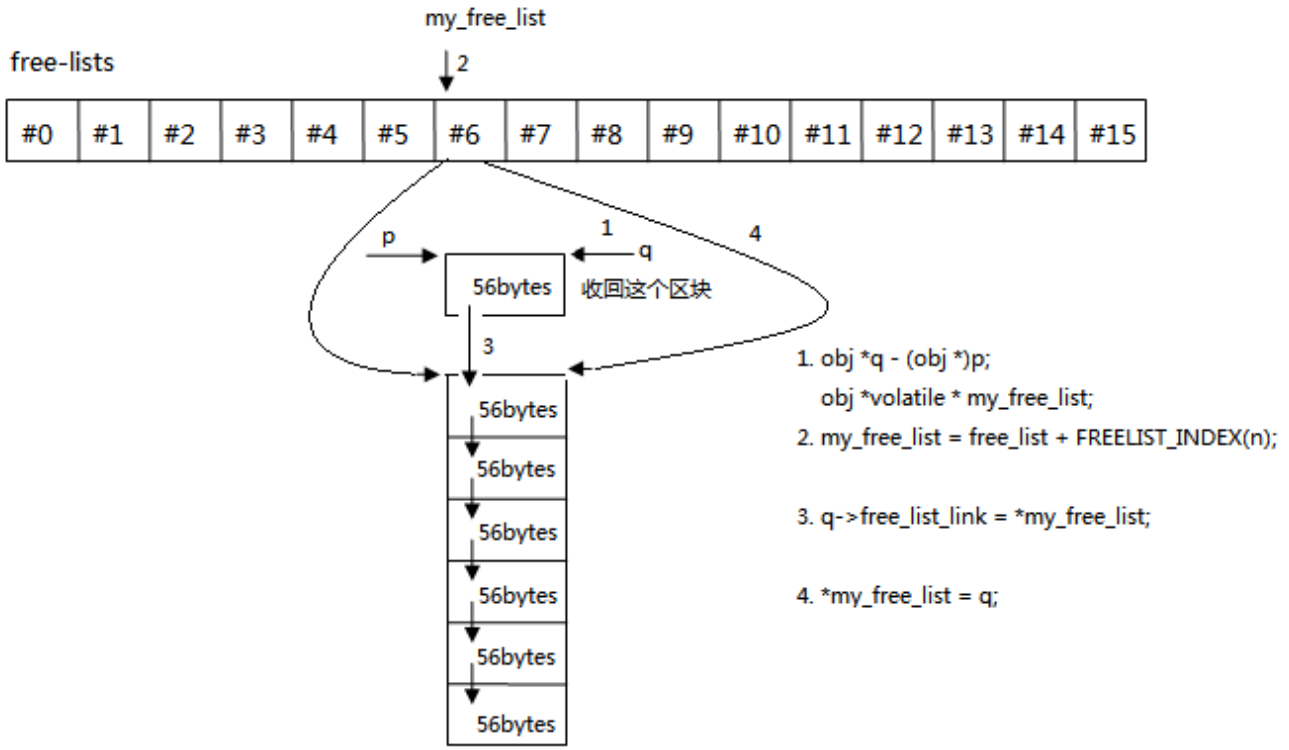


学过链表的操作的人不难理解上图，我就不再讲解。下面看deallocate()，代码如下：

```
static void deallocate(void *p, size_t n)
{
    obj *q = (obj *)p;
    obj * __VOLATILE * my_free_list;

    //如果要释放的字节数大于128，则调第一级配置器
    if (n > (size_t) __MAX_BYTES) {
        malloc_alloc::deallocate(p, n);
        return;
    }
    //寻找对应的位置
    my_free_list = free_list + FREELIST_INDEX(n);
    //以下两步将待释放的块加到链表上
    q -> free_list_link = *my_free_list;
    *my_free_list = q;
}
```

deallocate()函数释放内存的步骤如下图：



区块回收，纳入free-lists 阅读顺序请遵循图中编号

其实这就是一个链表的插入操作，也很简单。不再赘述！上面忘了给链表结点的结构体定义了，如下：

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

至此，SGI STL的对象的构造与析构、内存的分配与释放就介绍完毕了。

请你来说一说STL迭代器删除元素

这个主要考察的是迭代器失效的问题。

1.对于序列容器vector,deque来说，使用erase(itertor)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是erase会返回下一个有效的迭代器；

2.对于关联容器map set来说，使用了erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响到下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可。

3.对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator，因此上面两种正确的方法都可以使用。

参考网址

对于关联容器（如map，set，multimap，multiset），删除当前的iterator，仅仅会使当前的iterator失效，只要在erase时，递增当前的iterator即可。这是因为map之类的容器，使用了红黑树来实现，插入，删除一个结点不会对其他结点造成影响。使用方式如下例子：

```
set<int> valset = { 1,2,3,4,5,6 };
set<int>::iterator iter;
for (iter = valset.begin(); iter != valset.end(); )
{
     if (3 == *iter)
          valset.erase(iter++);
     else
          ++iter;
}
```

因为传给erase的是iter的一个副本，iter++是下一个有效的迭代器。

（2）对于序列式容器（如vector，deque，list等），删除当前的iterator会使后面所有元素的iterator都失效。这是因为vector，deque使用了连续分配的内存，删除一个元素导致后面所有的元素会向前移动一个位置。不过erase方法可以返回下一个有效的iterator。使用方式如下,例如：

```
vector<int> val = { 1,2,3,4,5,6 };
vector<int>::iterator iter;
for (iter = val.begin(); iter != val.end(); )
{
     if (3 == *iter)
          iter = val.erase(iter);     //返回下一个有效的迭代器，无需+1
     else
          ++iter;
}
```

请你说一说STL中MAP数据存放形式

红黑树。unordered map底层结构是哈希表

请你讲讲STL有什么基本组成

STL主要由：以下几部分组成： 容器，迭代器，仿函数，算法，分配器，配接器 他们之间的关系：分配器给容器分配存储空间，算法通过迭代器获取容器中的内容，仿函数可以协助算法完成各种操作，配接器用来套接适配仿函数

参考网址

STL有三大核心部分：容器（Container）、算法（Algorithms）、迭代器（Iterator），容器适配器（container adaptor），函数对象(functor，仿函数)，除此之外还有STL其他标准组件。通俗的讲：

容器：装东西的东西，装水的杯子，装咸水的大海，装人的教室.....STL里的容器是可容纳一些数据的模板类。

算法：就是往杯子里倒水，往大海里排污，从教室里撵人.....STL里的算法，就是处理容器里面数据的方法、操作。

迭代器：往杯子里倒水的水壶，排污的管道，撵人的那个物业管理人员……STL里的迭代器：遍历容器中数据的对象。对存储于容器中的数据进行处理时，迭代器能从一个成员移向另一个成员。他能按预先定义的顺序在某些容器中的成员间移动。对普通的一维数组、向量、双端队列和列表来说，迭代器是一种指针。

下面让我们来看看专家是怎么说的：

容器（container）：容器是数据在内存中组织的方法，例如，数组、堆栈、队列、链表或二叉树（不过这些都不是STL标准容器）。STL中的容器是一种存储T（Template）类型值的有限集合的数据结构,容器的内部实现一般是类。这些值可以是对象本身，如果数据类型T代表的是Class的话。

算法（algorithm）：算法是应用在容器上以各种方法处理其内容的行为或功能。例如，有对容器内容排序、复制、检索和合并的算法。在STL中，算法是由模板函数表现的。这些函数不是容器类的成员函数。相反，它们是独立的函数。令人吃惊的特点之一就是其算法如此通用。不仅可以将其用于STL容器，而且可以用于普通的C＋＋数组或任何其他应用程序指定的容器。

迭代器(iterator)：一旦选定一种容器类型和数据行为(算法)，那么剩下唯一要他做的就是用迭代器使其相互作用。可以把达代器看作一个指向容器中元素的普通指针。可以如递增一个指针那样递增迭代器，使其依次指向容器中每一个后继的元素。迭代器是STL的一个关键部分，因为它将算法和容器连在一起。

1. 容器

STL中的容器有队列容器和关联容器，容器适配器（congtainer adapters：stack,queue，priority queue），位集（bit_set），串包(string_package)等等。

2. 算法（algorithm）：

```
#inlcude <algorithm>
```

STL中算法的大部分都不作为某些特定容器类的成员函数，他们是泛型的，每个算法都有处理大量不同容器类中数据的使用。值得注意的是，STL中的算法大多有多种版本，用户可以依照具体的情况选择合适版本。中在STL的泛型算法中有4类基本的算法：

变序型队列算法：可以改变容器内的数据；

3. 迭代器（itertor）：

```
#include<iterator>
```

迭代器实际上是一种泛化指针，如果一个迭代器指向了容器中的某一成员，那么迭代器将可以通过自增自减来遍历容器中的所有成员。迭代器是联系容器和算法的媒介，是算法操作容器的接口。在运用算法操作容器的时候，我们常常在不知不觉中已经使用了迭代器。 STL中定义了6种迭代器：

输入迭代器，在容器的连续区间内向前移动，可以读取容器内任意值；

输出迭代器，把值写进它所指向的队列成员中；

前向迭代器，读取队列中的值，并可以向前移动到下一位置（++p,p++）；

双向迭代器，读取队列中的值，并可以向前向后遍历容器；

随机访问迭代器, vector::iterator，list::iterator等都是这种迭代器 ；

流迭代器，可以直接输出、输入流中的值；

4. STL的其他标准组件

函数对象（functor或者funtion objects）

```
#include<functional>
```

函数对象又称之为仿函数。函数对象将函数封装在一个对象中，使得它可作为参数传递给合适的STL算法，从而使算法的功能得以扩展。可以把它当作函数来使用。用户也可以定义自己的函数对象。下面让我们来定义一个自己的函数对象.

请你说说STL中map与unordered_map

1、Map

映射，map 的所有元素都是 pair，同时拥有实值（value）和键值（key）。pair 的第一元素被视为键值，第二元素被视为实值。所有元素都会根据元素的键值自动被排序。不允许键值重复。

底层实现：红黑树

适用场景：有序键值对不重复映射

2、Multimap

多重映射。multimap 的所有元素都是 pair，同时拥有实值（value）和键值（key）。pair 的第一元素被视为键值，第二元素被视为实值。所有元素都会根据元素的键值自动被排序。允许键值重复。

底层实现：红黑树

适用场景：有序键值对可重复映射

1. map存储结构是红黑树，所以需要定义比较函数（less），查找效率为O(logN).
2. unordered_map存储结构是数组，需要定义hash函数（计算key）和比较函数(equal)，查找效率为O(1).
3. unordered_map就是hash_map.
4. insert、find、[]等方法形式上一致。

map： map内部实现了一个红黑树（红黑树是非严格平衡二叉搜索树，而AVL是严格平衡二叉搜索树），红黑树具有自动排序的功能，因此map内部的所有元素都是有序的，红黑树的每一个节点都代表着map的一个元素。因此，对于map进行的查找，删除，添加等一系列的操作都相当于是对红黑树进行的操作。map中的元素是按照二叉搜索树（又名二叉查找树、二叉排序树，特点就是左子树上所有节点的键值都小于根节点的键值，右子树所有节点的键值都大于根节点的键值）存储的，使用中序遍历可将键值按照从小到大遍历出来。

unordered_map: unordered_map内部实现了一个哈希表（也叫散列表，通过把关键码值映射到Hash表中一个位置来访问记录，查找的时间复杂度可达到O(1)，其在海量数据处理中有着广泛应用）。因此，其元素的排列顺序是无序的。

请你说一说vector和list的区别，应用，越详细越好

1、概念：

1）Vector

连续存储的容器，动态数组，在堆上分配空间

底层实现：数组

两倍容量增长：

vector 增加（插入）新元素时，如果未超过当时的容量，则还有剩余空间，那么直接添加到最后（插入指定位置），然后调整迭代器。

如果没有剩余空间了，则会重新配置原有元素个数的两倍空间，然后将原空间元素通过复制的方式初始化新空间，再向新空间增加元素，最后析构并释放原空间，之前的迭代器会失效。

性能：

访问：O(1)

插入：在最后插入（空间够）：很快

在最后插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

在中间插入（空间够）：内存拷贝

在中间插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

删除：在最后删除：很快

在中间删除：内存拷贝

适用场景：经常随机访问，且不经常对非尾节点进行插入删除。

2、List

动态链表，在堆上分配空间，每插入一个元数都会分配空间，每删除一个元素都会释放空间。

底层：双向链表

性能：

访问：随机访问性能很差，只能快速访问头尾节点。

插入：很快，一般是常数开销

删除：很快，一般是常数开销

适用场景：经常插入删除大量数据

2、区别：

1）vector底层实现是数组；list是双向 链表。

2）vector支持随机访问，list不支持。

3）vector是顺序内存，list不是。

4）vector在中间节点进行插入删除会导致内存拷贝，list不会。

5）vector一次性分配好内存，不够时才进行2倍扩容；list每次插入新节点都会进行内存申请。

6）vector随机访问性能好，插入删除性能差；list随机访问性能差，插入删除性能好。

3、应用

vector拥有一段连续的内存空间，因此支持随机访问，如果需要高效的随即访问，而不在乎插入和删除的效率，使用vector。

list拥有一段不连续的内存空间，如果需要高效的插入和删除，而不关心随机访问，则应使用list。

请你来说一下STL中迭代器的作用，有指针为何还要迭代器

1、迭代器

Iterator（迭代器）模式又称Cursor（游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。或者这样说可能更容易理解：Iterator模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素。

由于Iterator模式的以上特性：与聚合对象耦合，在一定程度上限制了它的广泛运用，一般仅用于底层聚合支持类，如STL的list、vector、stack等容器类及ostream_iterator等扩展iterator。

2、迭代器和指针的区别

迭代器不是指针，是类模板，表现的像指针。他只是模拟了指针的一些功能，通过重载了指针的一些操作符，->、*、++、--等。迭代器封装了指针，是一个“可遍历STL（ Standard Template Library）容器内全部或部分元素”的对象， 本质是封装了原生指针，是指针概念的一种提升（lift），提供了比指针更高级的行为，相当于一种智能指针，他可以根据不同类型的数据结构来实现不同的++，--等操作。

迭代器返回的是对象引用而不是对象的值，所以cout只能输出迭代器使用*取值后的值而不能直接输出其自身。

3、迭代器产生原因

Iterator类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构而达到循环遍历集合的效果。

参考网址

迭代器实际上是对“遍历容器”这一操作进行了封装。 在编程中我们往往会用到各种各样的容器，但由于这些容器的底层实现各不相同，所以对他们进行遍历的方法也是不同的。例如，数组使用指针算数就可以遍历，但链表就要在不同节点直接进行跳转。 这是非常不利于代码重用的。例如你有一个简单的查找容器中最小值的函数findMin，如果没有迭代器，那么你就必须定义适用于数组版本的findMin和适用于链表版本的findMin，如果以后有更多容器需要使用findMin，那就只好继续添加重载……而如果每个容器又需要更多的函数例如findMax，sort，那简直就是重载地狱…… 我们的救星就是迭代器啦！如果我们将这些遍历容器的操作都封装成迭代器，那么诸如findMin一类的算法就都可以针对迭代器编程而不是针对具体容器编程，工作量一下子就少了很多！ 至于指针，由于指针也可以用来遍历容器(数组)，所以指针也可是算是迭代器的一种。但是指针还有其他功能，并不只局限于遍历数组。因为使用指针变量数组的操作太深入人心，c++stl中的迭代器就是刻意仿照指针来设计接口的

请你说一说epoll原理

O/I多路复用解释：https://www.zhihu.com/question/28594409

作者：晨随

链接：https://www.zhihu.com/question/28594409/answer/52763082

来源：知乎

关于I/O多路复用(又被称为“事件驱动”)，首先要理解的是，操作系统为你提供了一个功能，当你的某个socket可读或者可写的时候，它可以给你一个通知。这样当配合非阻塞的socket使用时，只有当系统通知我哪个描述符可读了，我才去执行read操作，可以保证每次read都能读到有效数据而不做纯返回-1和EAGAIN的无用功。写操作类似。操作系统的这个功能通过select/poll/epoll/kqueue之类的系统调用函数来使用，这些函数都可以同时监视多个描述符的读写就绪状况，这样，多个描述符的I/O操作都能在一个线程内并发交替地顺序完成，这就叫I/O多路复用，这里的“复用”指的是复用同一个线程。

以select和tcp socket为例，所谓可读事件，具体的说是指以下事件： 1 socket内核接收缓冲区中的可用字节数大于或等于其低水位SO_RCVLOWAT; 2 socket通信的对方关闭了连接，这个时候在缓冲区里有个文件结束符EOF，此时读操作将返回0； 3 监听socket的backlog队列有已经完成三次握手的连接请求，可以调用accept； 4 socket上有未处理的错误，此时可以用getsockopt来读取和清除该错误。

所谓可写事件，则是指： 1 socket的内核发送缓冲区的可用字节数大于或等于其低水位SO_SNDLOWAIT； 2 socket的写端被关闭，继续写会收到SIGPIPE信号； 3 非阻塞模式下，connect返回之后，发起连接成功或失败； 4 socket上有未处理的错误，此时可以用getsockopt来读取和清除该错误。

int epoll_create(int size);

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

int epoll_wait(int epfd, struct epoll_event *events,int maxevents, int timeout);

首先创建一个epoll对象，然后使用epoll_ctl对这个对象进行操作，把需要监控的描述添加进去，这些描述如将会以epoll_event结构体的形式组成一颗红黑树，接着阻塞在epoll_wait，进入大循环，当某个fd上有事件发生时，内核将会把其对应的结构体放入到一个链表中，返回有事件发生的链表。

参考网址-select与epoll介绍

为什么epoll这么快：

epoll是多路复用IO(I/O Multiplexing)中的一种方式,但是仅用于linux2.6以上内核,在开始讨论这个问题之前,先来解释一下为什么需要多路复用IO.

以一个生活中的例子来解释.

假设你在大学中读书,要等待一个朋友来访,而这个朋友只知道你在A号楼,但是不知道你具体住在哪里,于是你们约好了在A号楼门口见面. 如果你使用的阻塞IO模型来处理这个问题,那么你就只能一直守候在A号楼门口等待朋友的到来,在这段时间里你不能做别的事情,不难知道,这种方式的效率是低下的. 现在时代变化了,开始使用多路复用IO模型来处理这个问题.你告诉你的朋友来了A号楼找楼管大妈,让她告诉你该怎么走.这里的楼管大妈扮演的就是多路复用IO的角色. 进一步解释select和epoll模型的差异. select版大妈做的是如下的事情:比如同学甲的朋友来了,select版大妈比较笨,她带着朋友挨个房间进行查询谁是同学甲,你等的朋友来了,于是在实际的代码中,select版大妈做的是以下的事情:

```
int n = select(&readset,NULL,NULL,100);
for (int i = 0; n > 0; ++i)
{
    if (FD_ISSET(fdarray[i], &readset))
    {
        do_something(fdarray[i]);
        --n;
    }
}
```

epoll版大妈就比较先进了,她记下了同学甲的信息,比如说他的房间号,那么等同学甲的朋友到来时,只需要告诉该朋友同学甲在哪个房间即可,不用自己亲自带着人满大楼的找人了.于是epoll版大妈做的事情可以用如下的代码表示:

```
n = epoll_wait(epfd,events,20,500);
for(i=0;i<n;++i)
{
    do_something(events[n]);
}
```

在epoll中,关键的数据结构epoll_event定义如下:

```
typedef union epoll_data
{
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;
struct epoll_event {
    __uint32_t events;      /* Epoll events */
    epoll_data_t data;      /* User data variable */
};
```

可以看到,epoll_data是一个union结构体,它就是epoll版大妈用于保存同学信息的结构体,它可以保存很多类型的信息:fd,指针,等等.有了这个结构体,epoll大妈可以不用吹灰之力就可以定位到同学甲.

别小看了这些效率的提高,在一个大规模并发的服务器中,轮询IO是最耗时间的操作之一.再回到那个例子中,如果每到来一个朋友楼管大妈都要全楼的查询同学,那么处理的效率必然就低下了,过不久楼底就有不少的人了.

对比最早给出的阻塞IO的处理模型, 可以看到采用了多路复用IO之后, 程序可以自由的进行自己除了IO操作之外的工作, 只有到IO状态发生变化的时候由多路复用IO进行通知, 然后再采取相应的操作, 而不用一直阻塞等待IO状态发生变化了.

从上面的分析也可以看出,epoll比select的提高实际上是一个用空间换时间思想的具体应用.

二、深入理解epoll的实现原理:

开发高性能网络程序时，windows开发者们言必称iocp，linux开发者们则言必称epoll。大家都明白epoll是一种IO多路复用技术，可以非常高效的处理数以百万计的socket句柄，比起以前的select和poll效率高大发了。我们用起epoll来都感觉挺爽，确实快，那么，它到底为什么可以高速处理这么多并发连接呢?

先简单回顾下如何使用C库封装的3个epoll系统调用吧。

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,int maxevents, int timeout);
```

使用起来很清晰，首先要调用epoll_create建立一个epoll对象。参数size是内核保证能够正确处理的最大句柄数，多于这个最大数时内核可不保证效果。 epoll_ctl可以操作上面建立的epoll，例如，将刚建立的socket加入到epoll中让其监控，或者把 epoll正在监控的某个socket句柄移出epoll，不再监控它等等。 epoll_wait在调用时，在给定的timeout时间内，当在监控的所有句柄中有事件发生时，就返回用户态的进程。

从上面的调用方式就可以看到epoll比select/poll的优越之处：因为后者每次调用时都要传递你所要监控的所有socket给select/poll系统调用，这意味着需要将用户态的socket列表copy到内核态，如果以万计的句柄会导致每次都要copy几十几百KB的内存到内核态，非常低效。而我们调用epoll_wait时就相当于以往调用select/poll，但是这时却不用传递socket句柄给内核，因为内核已经在epoll_ctl中拿到了要监控的句柄列表。

所以，实际上在你调用epoll_create后，内核就已经在内核态开始准备帮你存储要监控的句柄了，每次调用epoll_ctl只是在往内核的数据结构里塞入新的socket句柄。 在内核里，一切皆文件。所以，epoll向内核注册了一个文件系统，用于存储上述的被监控socket。当你调用epoll_create时，就会在这个虚拟的epoll文件系统里创建一个file结点。当然这个file不是普通文件，它只服务于epoll。

epoll在被内核初始化时（操作系统启动），同时会开辟出epoll自己的内核高速cache区，用于安置每一个我们想监控的socket，这些socket会以红黑树的形式保存在内核cache里，以支持快速的查找、插入、删除。这个内核高速cache区，就是建立连续的物理内存页，然后在之上建立slab层，简单的说，就是物理上分配好你想要的size的内存对象，每次使用时都是使用空闲的已分配好的对象。

```
static int __init eventpoll_init(void)
{
    ... ...
    /* Allocates slab cache used to allocate "struct epitem" items */
    epi_cache = kmem_cache_create("eventpoll_epi", sizeof(struct epitem),
            0, SLAB_HWCACHE_ALIGN|EPI_SLAB_DEBUG|SLAB_PANIC,
            NULL, NULL);
    /* Allocates slab cache used to allocate "struct eppoll_entry" */
    pwq_cache = kmem_cache_create("eventpoll_pwq",
            sizeof(struct eppoll_entry), 0,
            EPI_SLAB_DEBUG|SLAB_PANIC, NULL, NULL);
 ... ...
```

epoll的高效就在于，当我们调用epoll_ctl往里塞入百万个句柄时，epoll_wait仍然可以飞快的返回，并有效的将发生事件的句柄给我们用户。这是由于我们在调用epoll_create时，内核除了帮我们在epoll文件系统里建了个file结点，在内核cache里建了个红黑树用于存储以后epoll_ctl传来的socket外，还会再建立一个list链表，用于存储准备就绪的事件，当epoll_wait调用时，仅仅观察这个list链表里有没有数据即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。所以，epoll_wait非常高效。

而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll_wait仅需要从内核态copy少量的句柄到用户态而已，如何能不高效？！

那么，这个准备就绪list链表是怎么维护的呢？当我们执行epoll_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。

如此，一颗红黑树，一张准备就绪句柄链表，少量的内核cache，就帮我们解决了大并发下的socket处理问题。执行epoll_create时，创建了红黑树和就绪链表，执行epoll_ctl时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据。执行epoll_wait时立刻返回准备就绪链表里的数据即可。

最后看看epoll独有的两种模式LT和ET。无论是LT和ET模式，都适用于以上所说的流程。区别是，LT模式下，只要一个句柄上的事件一次没有处理完，会在以后调用epoll_wait时次次返回这个句柄，而ET模式仅在第一次返回。

这件事怎么做到的呢？当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll_wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表，最后，epoll_wait干了件事，就是检查这些socket，如果不是ET模式（就是LT模式的句柄了），并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表了。所以，非ET的句柄，只要它上面还有事件，epoll_wait每次都会返回。而ET模式的句柄，除非有新中断到，即使socket上的事件没有处理完，也是不会次次从epoll_wait返回的。

三、扩展阅读（epoll与之前其他相关技术的比较）：

Linux提供了select、poll、epoll接口来实现IO复用，三者的原型如下所示，本文从参数、实现、性能等方面对三者进行对比。

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout); int poll(struct pollfd *fds, nfds_t nfds, int timeout); int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);

select、poll、epoll_wait参数及实现对比 1． select的第一个参数nfds为fdset集合中最大描述符值加1，fdset是一个位数组，其大小限制为__FD_SETSIZE（1024），位数组的每一位代表其对应的描述符是否需要被检查。

select的第二三四个参数表示需要关注读、写、错误事件的文件描述符位数组，这些参数既是输入参数也是输出参数，可能会被内核修改用于标示哪些描述符上发生了关注的事件。所以每次调用select前都需要重新初始化fdset。

timeout参数为超时时间，该结构会被内核修改，其值为超时剩余的时间。

select对应于内核中的sys_select调用，sys_select首先将第二三四个参数指向的fd_set拷贝到内核，然后对每个被SET的描述符调用进行poll，并记录在临时结果中（fdset），如果有事件发生，select会将临时结果写到用户空间并返回；当轮询一遍后没有任何事件发生时，如果指定了超时时间，则select会睡眠到超时，睡眠结束后再进行一次轮询，并将临时结果写到用户空间，然后返回。

select返回后，需要逐一检查关注的描述符是否被SET（事件是否发生）。

2． poll与select不同，通过一个pollfd数组向内核传递需要关注的事件，故没有描述符个数的限制，pollfd中的events字段和revents分别用于标示关注的事件和发生的事件，故pollfd数组只需要被初始化一次。

poll的实现机制与select类似，其对应内核中的sys_poll，只不过poll向内核传递pollfd数组，然后对pollfd中的每个描述符进行poll，相比处理fdset来说，poll效率更高。

poll返回后，需要对pollfd中的每个元素检查其revents值，来得指事件是否发生。

3． epoll通过epoll_create创建一个用于epoll轮询的描述符，通过epoll_ctl添加/修改/删除事件，通过epoll_wait检查事件，epoll_wait的第二个参数用于存放结果。

epoll与select、poll不同，首先，其不用每次调用都向内核拷贝事件描述信息，在第一次调用后，事件信息就会与对应的epoll描述符关联起来。另外epoll不是通过轮询，而是通过在等待的描述符上注册回调函数，当事件发生时，回调函数负责把发生的事件存储在就绪事件链表中，最后写到用户空间。

epoll返回后，该参数指向的缓冲区中即为发生的事件，对缓冲区中每个元素进行处理即可，而不需要像poll、select那样进行轮询检查。

更细节建议观看

所有epoll处理框架

```
for( ; ; )
    {
        nfds = epoll_wait(epfd,events,20,500);
        for(i=0;i<nfds;++i)
        {
            if(events[i].data.fd==listenfd) //有新的连接
            {
                connfd = accept(listenfd,(sockaddr *)&clientaddr, &clilen); //accept这个连接
                ev.data.fd=connfd;
                ev.events=EPOLLIN|EPOLLET;
                epoll_ctl(epfd,EPOLL_CTL_ADD,connfd,&ev); //将新的fd添加到epoll的监听队列中
            }

            else if( events[i].events&EPOLLIN ) //接收到数据，读socket
            {
                n = read(sockfd, line, MAXLINE)) < 0    //读
                ev.data.ptr = md;     //md为自定义类型，添加数据
                ev.events=EPOLLOUT|EPOLLET;
                epoll_ctl(epfd,EPOLL_CTL_MOD,sockfd,&ev);//修改标识符，等待下一个循环时发送数据，
异步处理的精髓
            }
            else if(events[i].events&EPOLLOUT) //有数据待发送，写socket
            {
                struct myepoll_data* md = (myepoll_data*)events[i].data.ptr;    //取数据
                sockfd = md->fd;
                send( sockfd, md->ptr, strlen((char*)md->ptr), 0 );        //发送数据
                ev.data.fd=sockfd;
                ev.events=EPOLLIN|EPOLLET;
                epoll_ctl(epfd,EPOLL_CTL_MOD,sockfd,&ev); //修改标识符，等待下一个循环时接收数据
            }
            else
            {
                //其他的处理
            }
        }
 }
```

比较解释

select：是最初解决IO阻塞问题的方法。用结构体fd_set来告诉内核监听多个文件描述符，该结构体被称为描述符集。由数组来维持哪些描述符被置位了。对结构体的操作封装在三个宏定义中。通过轮寻来查找是否有描述符要被处理。

存在的问题：

1. 内置数组的形式使得select的最大文件数受限与FD_SIZE；
2. 每次调用select前都要重新初始化描述符集，将fd从用户态拷贝到内核态，每次调用select后，都需要将fd从内核态拷贝到用户态；
3. 轮寻排查当文件描述符个数很多时，效率很低；

3、poll

poll：通过一个可变长度的数组解决了select文件描述符受限的问题。数组中元素是结构体，该结构体保存描述符的信息，每增加一个文件描述符就向数组中加入一个结构体，结构体只需要拷贝一次到内核态。poll解决了select重复初始化的问题。轮寻排查的问题未解决。

4、epoll

epoll：轮寻排查所有文件描述符的效率不高，使服务器并发能力受限。因此，epoll采用只返回状态发生变化的文件描述符，便解决了轮寻的瓶颈。

epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式

1. LT模式

LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的。

2. ET模式

ET(edge-triggered)是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。但是请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once)

ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

3、LT模式与ET模式的区别如下： LT模式：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll_wait时，会再次响应应用程序并通知此事件。 ET模式：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll_wait时，不会再次响应应用程序并通知此事件。

n个整数的无序数组，找到每个元素后面比它大的第一个数，要求时间复杂度为O(N)

单调栈,单调递减栈

请你回答一下STL里resize和reserve的区别

resize 调整的是vector的大小，改变的是vector的size；如果resize的调整值小于vector的size，则会删除多余的元素；如果大于size，则增加元素。

reserve 调整的是vector的预分配的内存，改变的是vector的capacity。如果reserve的调整值大于vector的capacity，则会增加预分配的内存；如果小于capacity，则不做任何改变。