

**TinySTL 说明文档**

Author：HLhuanglang

Email:1282424466qq.com

1. 该文档目标为解释相关内容的原理以及API介绍

# 第一章 预备知识

## 1.1 前期准备

TinySTL依照的C++11标准，流畅的阅读并把握整个项目需要掌握的预备知识有如下：

* C++基础语法(推荐阅读C++ prime,可大致阅读，有使用到再细读)；
* C++模板相关知识；
* C++11标准(推荐阅读深入理解C++11)；
* 常见的数据结构与算法。

如果你需要相关电子书籍资料，可以通过邮箱联系我，备注来意即可，邮箱为：1282424466@qq.com

## 1.2 STL介绍

### 1.2.1 STL

* 迭代器、算法、容器、仿函数、内存配置器、配接器 六部分组成；
* STL所有组件都是由模板构成，其元素可以是任意的；
* STL主要组件包括“容器、迭代器、算法和仿函数”

# 第二章 代码规范定义

## 2.1 Git commit与gitflow

### 2.1.1 git commit

commit信息提交格式如表2.1所示。

表2.1 Commit信息格式

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 影响模块 | 描述信息 | issue号 |

提交类型具体如下：

1. add：增加新功能；
2. update：更新功能；
3. refactor：重写代码；
4. remove：删除代码。

一次提交的实例：<add>(deque.h)创建deque容器。

### 2.1.2 gitflow

整个工作流程使用三条分支。master为最稳定的主线分支；xxx\_dev是开发某一个具体的功能时使用的分支，当完成大功能模块时，再开启一条测试分支test，测试无误后继续开发，当整个功能模块完成后，再进行一次综合测试，测试无误后并入master。

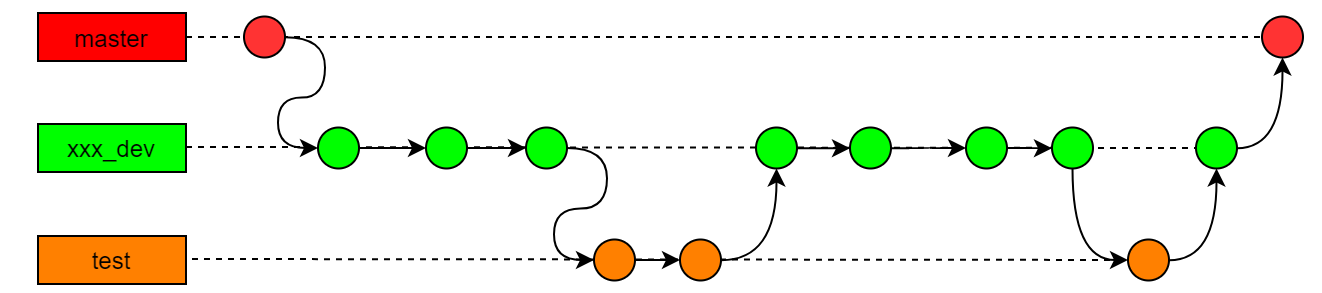


图2-1 gitflow

## 2.2 变量符号说明

本次项目中使用的变量名称说明如表2.2所示。

表2.2 变量缩写

|  |  |  |
| --- | --- | --- |
| **意义** | **变量** | **缩写** |
| 输入迭代器 | InputIterator | \_InIt |
| 输出迭代器 | OutputIterator | \_OutIt |
| 前进迭代器 | ForwardIterator | \_FwdIt |
| 任意(方向)迭代器 | RandowaccessIterator | \_RanIt |
| 操作符重载对象参数 | \_Left、\_Right |  |
| 函数的对象参数 | \_VectorX(x=1、2)  \_ListX(x=1、2)诸如此类 |  |
| 函数的值参数 | \_Val |  |
| 模板类型 | \_TyX(x=1、2) |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## 2.3 备注形式说明

### 2.3.1 文件说明

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

文件名 :

相关文件 :

文件实现功能 :

作者 :

版本 :

---------------------------------------------------------------------

多线程安全性 : <是/否>[说明]

异常时安全性 : <是/否>[说明]

---------------------------------------------------------------------

备注 :

---------------------------------------------------------------------

修改记录 :

日 期 版本 修改人 修改内容

2020/05/2 1.0 HLhuanglang 创建

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 2.3.2 函数备注说明

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

函数名 : uninitialized\_fill

功能 : 对范围(\_First, \_Last]进行对象填充，对象值为\_val

参数 : [IN\OUT\DUMY]

[IN] \_First : 给定范围的起始迭代器

[IN] \_Last : 给定范围的结束迭代器

[IN] \_Val : 填充对象的值

返回值 : 无

抛出异常 : 无

作者 : HLhuanglnag

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 第三章 开发期间小测试

在开发过程中，可能时不时的需要进行小测试，如下是个小例子。

|  |
| --- |
| #include<iostream>  #include"allocator.h"  #include"uninitialized.h"  using std::ostream;  using std::cout;  using std::endl;  class test {  public:  test() : a(1), b('a'){}  test(int a) : a(a), b('A') {}  friend ostream &operator<<(ostream &out, test& item);  private:  int a;  char b;  };  ostream &operator<<(ostream &out, test& item) {  out << item.a << " " << item.b;  return out;  }  int main() {  for (int i = 1; i <= 20; ++i) {  auto p = allocator<test>::allocate(5); //申请5个元素大小的空间  auto q = allocator<test>::allocate(5);  TinySTL::uninitialized\_fill(p, p+5, test(i));  TinySTL::uninitialized\_copy(p, p + 5, q);  for(int j=0; j<5; ++j)  cout << \*(p+j) <<" "<<\*(q+j)<< endl;  cout << "------"<<endl;  }  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;  int array[20];  TinySTL::uninitialized\_fill\_n(array, 20, 22); //对array填充20个22  for (auto n : array)  cout << n << endl;  system("pause");  return 0;  } |

# 第四章 基础模块

就像建房子需要先挖地基，任何事情都有最底层的支撑模块，TinySTL也不例外，在本项目中，我将空间配置器、迭代器、函数对象等为容器和算法提供服务的模块划分为基础模块。

## 4.1 空间配置器

空间配置器整体调用结构如图4-1所示。

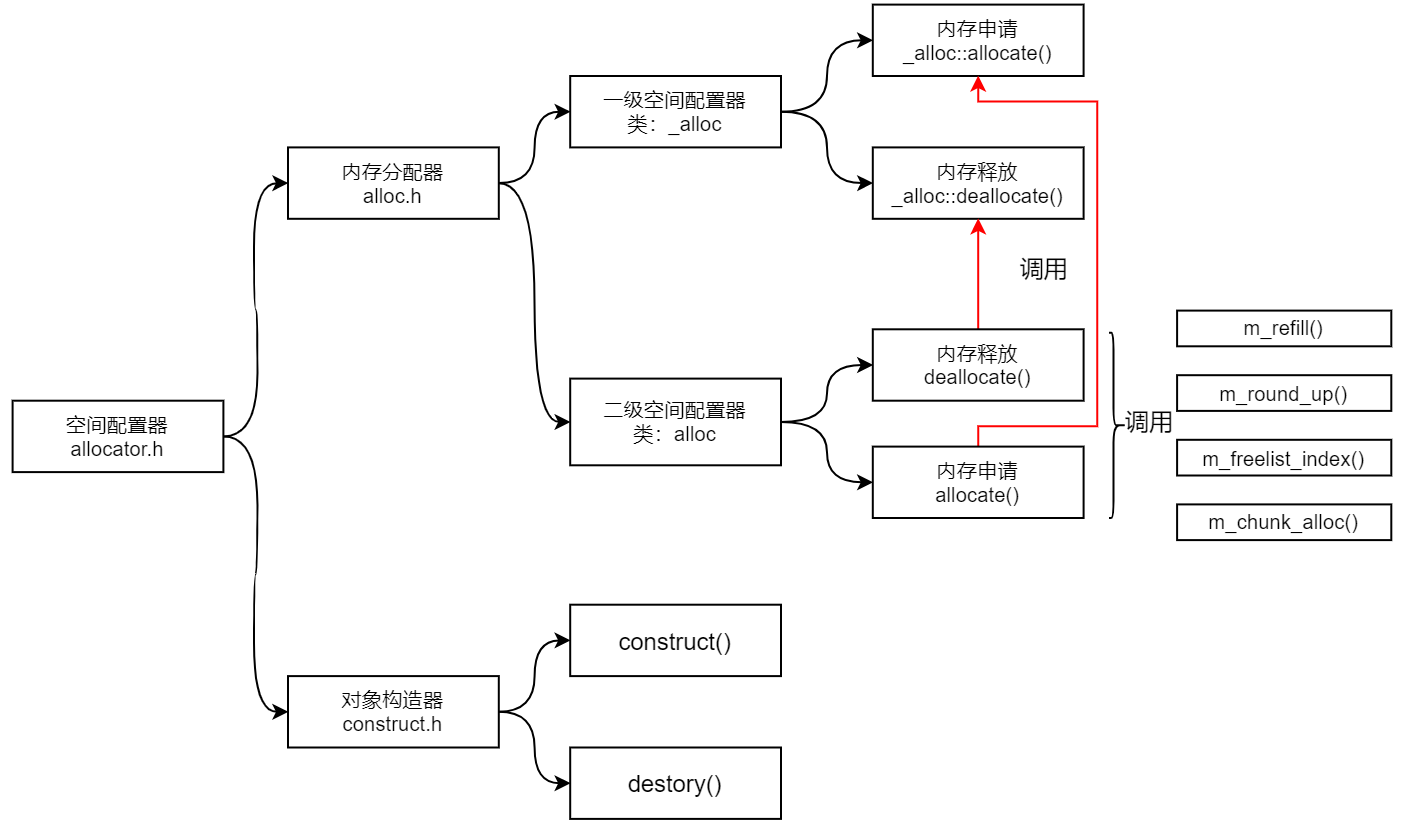


图4-1 空间配置器调用结构图

### 4.1.1 内存分配与释放

使用malloc()函数申请内存，malloc()不管申请内存多大都会执行，malloc()是内部是通过系统调用来申请内存的，因此频繁的调用malloc()来申请小内存如1byte时，会影响性能。为了解决小内存的申请问题，采用内存池+free\_list的内存管理方式来解决。内存分配规则为：

1. need\_bytes>128bytes，通过malloc函数申请，这称为一级内存分配器；
2. need\_bytes<=128bytes，通过内存池获取，这称为二级内存分配器。

关键静态变量：memory\_pool\_start 和memory\_pool\_end。用这两个变量表示一个内存池，当需要从内存池获取内存时，就是读取memory\_pool\_start的地址，然后更新地址memory\_pool\_start+=need\_bytes。由于这两个变量在程序运行时要一直存在且独有一份，所以定义为static，所以相关函数也就都定义成了static。

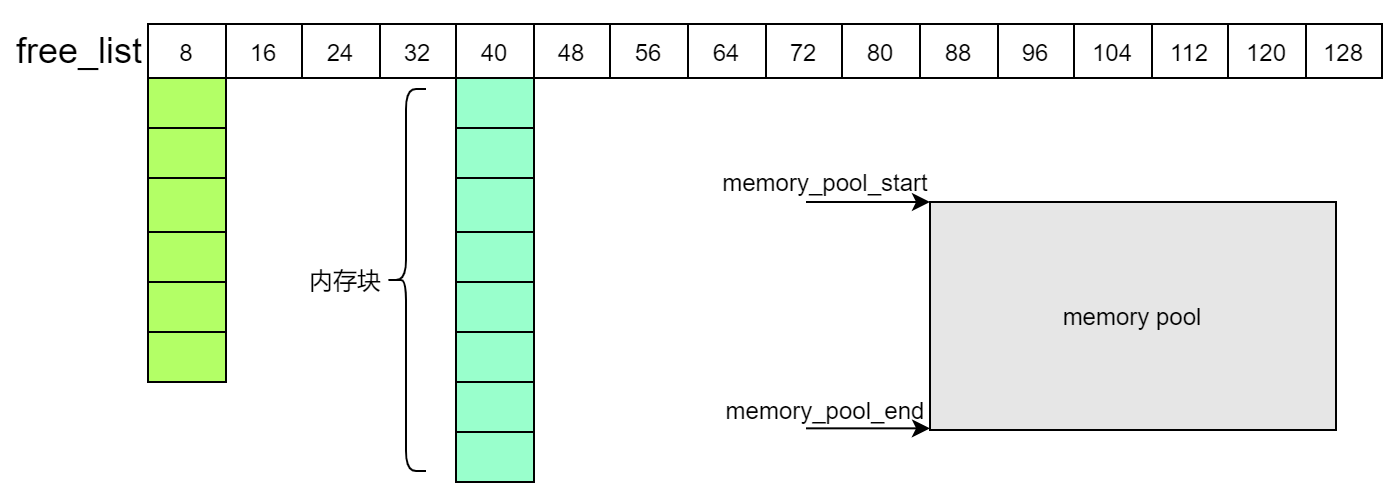


图4-2 内存池与free\_list

以图4-2为参考，进行一次内存申请过程的分析，申请逻辑如图4-3所示，使用到的具体函数如内部API所示。

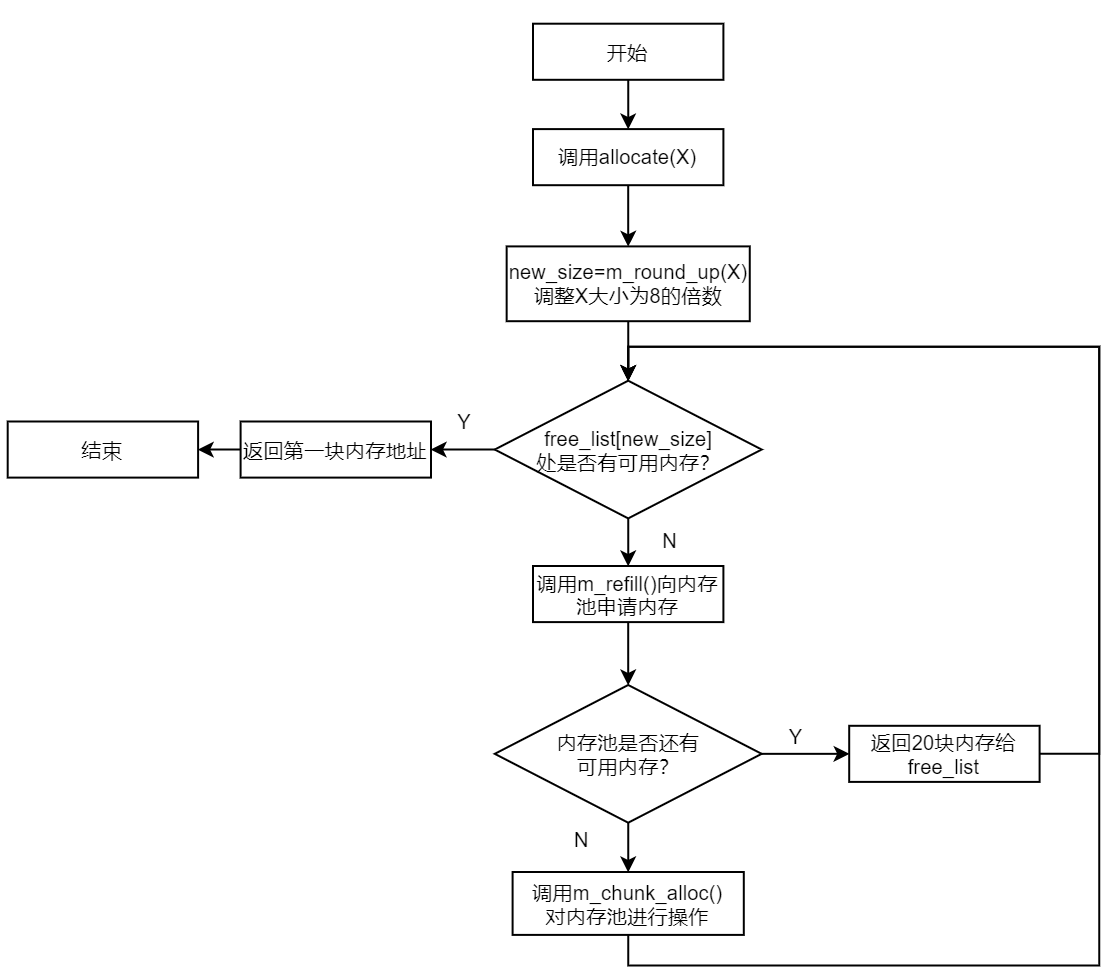


图4-2 内存池与free\_list

调用allocate(9)，首先可知申请内存的大小为9bytes，然后将该值上调为8的整数16bytes，然后查询free\_list上16bytes大小处是否有可用的内存块，如果有，则返回第一块内存块的地址，然后free\_list[16]处指针指向下一个内存块，如果没有，则向内存池申请20块大小为16bytes的内存块。

内存池收到申请后，首先自查是否有可用的内存，如果没有，则先调用malloc()把自身填满再来响应，如果有，则计算具体能够提供多少块，然后提供给free\_list。

**开放API**

|  |  |
| --- | --- |
| 函数名 | **static void\* allocate(size\_t \_Size)** |
| 作用 | 申请内存 |
| 参数 | \_Size为要申请内存的具体大小，如20byte |
| 说明 | 无 |

|  |  |
| --- | --- |
| 函数名 | **static void deallocate(void \*\_Ptr, size\_t \_Size)** |
| 作用 | 释放\_Ptr指向的内存 |
| 参数 | \_Ptr 为指针，指向要释放的内存  \_Size为要释放内存的具体大小 |
| 说明 | 当\_Size小于128时，内存会重新分配到freelist上，大于128bytes才会直接释放 |

|  |  |
| --- | --- |
| 函数名 | **static void\* reallocate(void \*\_Ptr, size\_t old\_size, size\_t new\_size)** |
| 作用 | 重新分配\_Ptr指向的内存大小 |
| 参数 | \_Ptr 为指针，指向要重新分配内存的地址  old\_size 旧内存的大小  new\_size 新的内存大小 |
| 说明 | 无 |

**内部API**

|  |  |
| --- | --- |
| 函数名 | **static size\_t M\_round\_up(size\_t bytes)** |
| 作用 | 将内存大小调整至8的倍数。如5调整至8 |
| 参数 | bytes 所申请内存的原始大小 |
| 说明 | 无 |

|  |  |
| --- | --- |
| 函数名 | **static size\_t M\_freelist\_index(size\_t bytes)** |
| 作用 | 计算出对应内存大小在freelist上的位置 |
| 参数 | bytes调整过的内存大小 |
| 说明 | 无 |

|  |  |
| --- | --- |
| 函数名 | **static void\* M\_refill(size\_t \_Size)** |
| 作用 | 当对应freelist位置上无可用内存时，向内存池申请内存 |
| 参数 | \_Size 向内存池申请内存的大小 |
| 说明 |  |

|  |  |
| --- | --- |
| 函数名 | **static char\* M\_chunk\_alloc(size\_t \_Size, size\_t& \_Count)** |
| 作用 | 从内存池分配内存给freelist |
| 参数 | \_Size 为请求分配内存的大小  \_Count为实际能够分配\_Size大小的内存有多少个 |
| 说明 | 该函数实际操作内存池 |

### 4.1.2 对象构造与析构

对象构造主要涉及到operator new和placement new的相关知识，自行查阅。

**开放API**

|  |  |
| --- | --- |
| 函数名 | **template<typename \_Ty> inline void construct(\_Ty\* \_Ptr)** |
| 作用 | 在\_Ptr指向的内存上构造对象，对象值为默认 |
| 参数 | \_Ptr 指向要构造对象的地址 |
| 说明 | 无 |

|  |  |
| --- | --- |
| 函数名 | **template<typename \_Ty1, typename \_Ty2>**  **inline void construct(\_Ty1 \*\_Ptr, const \_Ty2& \_Val)** |
| 作用 | 在\_Ptr指向的内存上构造对象，对象值为\_Val |
| 参数 | \_Ptr指向要构造对象处的地址  \_Val为对象的值 |
| 说明 | 无 |

|  |  |
| --- | --- |
| 函数名 | **template<typename \_Ty> inline void destory(\_Ty \*\_Ptr)** |
| 作用 | 析构 \_Ptr指向的对象 |
| 参数 | \_Ptr指向要析构对象处的地址 |
| 说明 | 无 |

|  |  |
| --- | --- |
| 函数名 | **template<typename \_InIt>**  **inline void destory(\_InIt \_First, \_InIt \_Last)** |
| 作用 | 析构[\_First, \_Last]范围内的对象 |
| 参数 | \_First区间起始地址处迭代器  \_Last区间结束地址处迭代器 |
| 说明 | 无 |

**内部API**

|  |  |
| --- | --- |
| 函数名 | **template<typename \_InIt> inline**  **void M\_destory(\_InIt \_First, \_InIt \_Last, \_\_true\_type \_True)** |
| 作用 | 供范围析构destory函数使用 |
| 参数 | \_True，\_true\_type类型结构体对象形参，在重载中起区分的作用 |
| 说明 |  |

|  |  |
| --- | --- |
| 函数名 | **template<typename \_InIt> inline**  **void M\_destory(\_InIt \_First, \_InIt \_Last, \_\_false\_type \_False)** |
| 作用 | 供范围析构destory函数使用 |
| 参数 | \_False，\_false\_type类型结构体对象形参，在重载中起区分的作用 |
| 说明 |  |

### 4.1.3 内存基本处理工具

|  |  |
| --- | --- |
| 函数名 | template<typename \_InIt, typename \_FwdIt>  \_FwdIt uninitialized\_copy(\_InIt \_First, \_InIt \_Last, \_FwdIt \_Result) |
| 作用 | 将区间[\_First, \_Last]的值复制到[\_Result, \_Result+(\_Last-\_First) ) |
| 参数 |  |
| 说明 |  |

|  |  |
| --- | --- |
| 函数名 |  |
| 作用 |  |
| 参数 |  |
| 说明 |  |

## 4.2 type\_traists

## 4.3 迭代器与反向迭代器

### 4.3.1 迭代器

### 4.3.2 反向迭代器

反向迭代器是一种适配器 adapter，是对迭代器进行改造。

## 4.4 函数对象

# 第五章 容器模块

## Vector

底层数据结构为**数组**，vector维护的是一个连续线性空间，所以不论元素型别为何，普通指针都可以作为vector的迭代器而满足所有必要条件，因为vector迭代器所需要的操作行为，如\*、->、++、--这些，普通指针天生就具备。vector支持随机存取，而普通指针正有着这样的能力。所以vector提供的是Random Access Iterators。

vector容器由 **Iter\_first**、**Iter\_last**、**Iter\_end**三个迭代器来表示，而指针能够满足vector迭代器的所有需求，所以vector容器的迭代器直接用指针。

Iter\_first Iter\_last Iter\_end

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| elem1 | elem2 | elem3 | … | … | … |  |  |  |  |

因为vector容器要实现动态扩容，为了提高元素插入的效率，创建vector容器时所申请的实际内存大于请求的内存。Iter\_first、Iter\_last对应的就是已经使用了的内存，当Iter\_last移动到了Iter\_end的位置时，这就表明需要重新申请一块更大的内存了，此时会造成迭代器失效。

### 对象创建与析构

|  |  |
| --- | --- |
| Vector<int> \_Vec1; | 产生一个空的容器对象，无任何元素 |
| Vector<int> \_Vec2(10); | 产生一个拥有10个元素的对象，其元素值都为0（int类型默认） |
| Vector<int> \_Vec3(10,1); | 产生一个拥有10个元素的对象，其元素值为1 |
| Vector<int> \_Vec4(\_Vec2); | 产生一个\_Vec2的副本（所有元素被拷贝） |
| Vector<int>\_Vec5  (\_Vec3.begin(), \_Vec3.end()) | 产生一个vector，以\_Vec3的区间[beg,end)的元素为初值 |
| ~vector<int>(); | 销毁所有元素，并释放内存 |

### 成员函数

所有函数以TinySTL::vector<int> Vec为例来解释

|  |  |
| --- | --- |
| 函数名 | **size\_type Size()** |
| 作用 | 返回当前容器内的元素个数 |
| 参数 | 无 |
| 使用 | int \_size = Vec.size(); |
| 说明 |  |

|  |  |
| --- | --- |
| 函数名 | **bool empty()** |
| 作用 | 判断当前容器内元素个数是否为0 |
| 参数 | 无 |
| 使用 | if(Vec.empty()) 或 bool is\_empty = Vec.empty(); |
| 说明 | empty()一般直接用在if的判断条件中 |

|  |  |
| --- | --- |
| 函数名 | **size\_type capacity()** |
| 作用 | 返回重新分配空间之前，容器所能容纳的最大元素数量 |
| 参数 | 无 |
| 使用 | int \_Cap = Vec.capacity(); |
| 说明 |  |

|  |  |
| --- | --- |
| 函数名 | **void reserve(\_NewCapacity)** |
| 作用 | 如果容量不足，则扩大 |
| 参数 | \_NewCapacity，扩容的目标大小 |
| 使用 | Vec.reverse(100); 从原来的大小扩大到能容纳100个元素 |
| 说明 | reverse函数只有在\_NewCapacity>capacity()时才工作！否则就算使用了也无效 |

|  |  |
| --- | --- |
| 函数名 | **void assign(\_Count, \_Elem)** |
| 作用 | 将\_Count个\_Elem赋值给容器 |
| 参数 |  |
| 使用 | Vec.assign(10,1); |
| 说明 |  |

|  |  |
| --- | --- |
| 函数名 | **reference front()** |
| 作用 | 返回第一个元素 |
| 参数 | 无 |
| 使用 | Vec.front() = 1;  或 cout<<Vec.front(); |
| 说明 | front不会检测第一个元素是否存在，且返回的是引用 |

|  |  |
| --- | --- |
| 函数名 | **void push\_back(const value\_type& \_Val)** |
| 作用 | 在容器尾部插入新元素，即在Iter\_last处插入新元素 |
| 参数 | 无 |
| 使用 | Vec.push\_back() |
| 说明 |  |

|  |  |
| --- | --- |
| 函数名 | **iterator erase(const\_iterator& \_Where)** |
| 作用 | 清除某个位置上的元素 |
| 参数 | \_Where 需要清除的位置 |
| 使用 | Vec.erase(Vec.begin()); |
| 说明 | 通过将\_Where后面的元素全部往前移动一位 |

|  |  |
| --- | --- |
| 函数名 | **iterator erase(const\_iterator& \_First, const\_iterator& \_Last)** |
| 作用 | 清除区间[\_First,\_Last)内的元素 |
| 参数 |  |
| 使用 | Vec.erase(Vec.begin(), Vec.end()); |
| 说明 |  |

## List

list容器是双向链表，有一个首元节点node，该节点的存在使得左闭右开原则容易实现。

node->pre就是链表尾部，node->next是链表的头部。

# 第六章 算法

# 第七章 测试框架