

**TinySTL 说明文档**

Author：派大星Pstar

CSDN：派大星Pstar

Email:1282424466qq.com

1. 该文档目标为解释相关内容的原理以及API介绍
2. 带有Pstar\_ 开头的函数为内部函数，不是通用API，仅为代码复用而设计

文章结构(以vector容器为例)：

简述vector容器相关原理以及设计思路

Vector容器相关通用API以及Pstar\_XXX等helper function介绍

常见问题以及解答(针对面试)

# 预备知识

## 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：

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

相关的资料都放在百度云了，如果链接失效了请通过邮件联系我，说明来意，我会回复的。百度云链接：<https://pan.baidu.com/s/1E9107zw3xDhHkPnmhfVTtg>提取码：wrs3

## TinySTL介绍

1.1 STL：

①迭代器、算法、容器、仿函数、内存配置器、配接器 六部分组成

②STL所有组件都是由模板构成，其元素可以是任意的。

③STL主要组件包括“容器、迭代器、算法和仿函数”

1.2容器：

用来存储并管理某类对象的集合。我们可以把容器看成是一个专柜，比如专柜A是专门用来存放蛋糕的，一个格子只能存放一个蛋糕。我们可以这么定义class Cake{....}; vector<Cake> myCake;

1.3迭代器：

我们设法不暴露太多容器内部实现细节给用户，因为会增加使用的复杂性，所以对内部构造进行封装。为了实现内外交互，就提供了迭代器这样一种“接口”，用户操作迭代器，迭代器对容器内部进行操作。每种容器都提供了各自的迭代器。迭代器了解该容器的内部结构，所以能够正确执行相关操作。

1.4算法：

算法用来处理群集内的元素。 所有的容器都有迭代器迭代器，通过迭代器的协助，算法程序可以用于任何容器。STL的一个特性是将数据和操作分离。数据由容器类别加以管理，操作由可定制的算法定义。迭代器在两者之间充当粘合剂。另外一个特性即“组件可以针对任意型运作”

2.1序列容器：

【2.1.1 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。

【2.1.2 list ------------------------------底层数据结构为：双向链表】

list不能够再像vector一样以普通指针作为迭代器，因为其结点不保证在存储空间中连续存在。list迭代器必须有能力指向list的结点，并且有能力进行正确的递增、递减、取值、成员存取等操作。由于list是一个双向链表，迭代器必须具备前移、后移能力，所以list提供Bidirectional Iterators.

【2.1.3 deque --------------------------底层数据结构为：中央控制器和多个缓冲区】

deque是一种双向开口的连续线性空间，所谓双向开口，也就是可以在头尾两端分别做元素的插入和删除操作。虽然vector也可以，但是在头部操作，效率极差。deque与vector最大的差异，在于deque没有所谓容量的概念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并连接起来。deque也提供Random Access Iterator，但它的迭代器并不是普通的指针。

2.2容器适配器：

stack、queue、priority\_queue，这种“修改某物接口，形成另一种风貌”之性质者，称为适配器

【2.2.1 stack -----------------------------底层容器为：list或deque】

stack所有元素的进出都必须符合”先进后出”的条件，只有stack顶端的元素，才有机会被外界取用。stack不提供走访功能，也不提供迭代器。

【2.2.2 queue ---------------------------底层容器为：list或deque】

queue是一种先进先出的数据结构。只有queue顶端的元素，才有机会被外界取用。queue不提供走访功能，也不提供迭代器。

【2.2.3 priority\_queue -------------底层容器为：vector,heap为处理规则来管理底层容器】

priority\_queue完全以底部容器为根据，再加上heap处理规则，所以其实现非常简单。缺省情况下是以vector为底层容器。priority\_queue的所有元素，进出都有一定的规则，只有queue顶端的元素（权值最高者），才有机会被外界取用。priority\_queue不提供遍历功能，也不提供迭代器。

2.3关联容器：

标准的STL关联容器分为set（集合）与map（映射表）两大类，以及以这两大类的衍生体multiset（多建集合）与multimap（多建映射表）。这些容器的底层机制均以RB-tree完成。RB-tree也是一个独立容器，但是不对外开放。

SGI STL还提供了一个不在标准规格之列的关联式容器：hash table（散列表），以及以hash table为底层机制而完成的hash\_set、hash\_map、hash\_multiset、hash\_multimap

# 代码规范定义

## Git commit与gitflow

1. **规范了git提交信息格式**：

<类型>(影响模块) 具体描述信息 (#issue？)

**类型**: **add(增加新内容)**、**fix(修复bug)**、**refactor(重构)**、

**update(更新功能)**、 **config(配置文件)**

**影响模块**：例如list.h这样写

**具体描述信息**：这次提交做了什么事情

**#issue？**：对应具体的issue号

具体例子如

<add>(deque.h)创建deque容器

1. **规范一套自己的 gitflow**

新开一个功能模块，就开一条分支，当完成一定量的设计时，再开一条test分支，用来检查之前的工作，大概没问题以后进行合并。

master

↘deque-------------- ---------

↘test--------↗

## 算法格式

算法模板：

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

函 数 名：xxxxxx

函数作用：xxxxxx

返 回 值：有则说明

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

Type func(xxx, xxx,....)

{

/\*注释：只解释关键疑难点\*/

}

## 变量缩写约定

InputIterator \_InIt

ForwardIterator \_FwdIt

OutputIterator \_OutIt

difference\_type \_Diff

value\_type \_Ty

randow\_access\_iterator \_RanIt

...

变量一下 下划线开头，第一个字母大写： \_Tmp

# 项目开发期测试

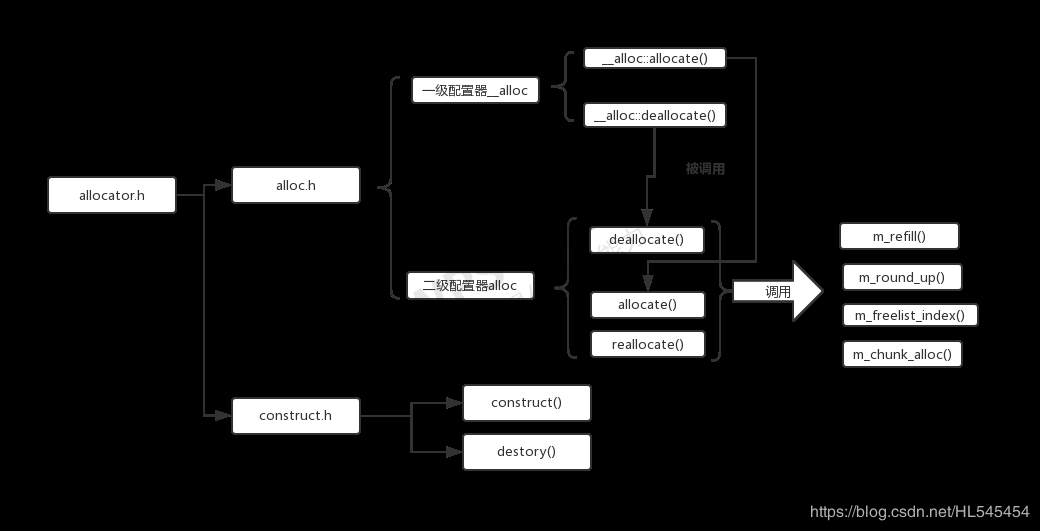
在项目编写的过程中，需要检测每一部分内容的正确性，例如如下代码，就是测试内存分配和常用内存算法函数的代码是否能正常工作的，确认正常无误后再开始后续工作。

|  |
| --- |
| #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;  } |

# 基础模块

## 空间配置器

空间配置器整体调用结构如图所示。



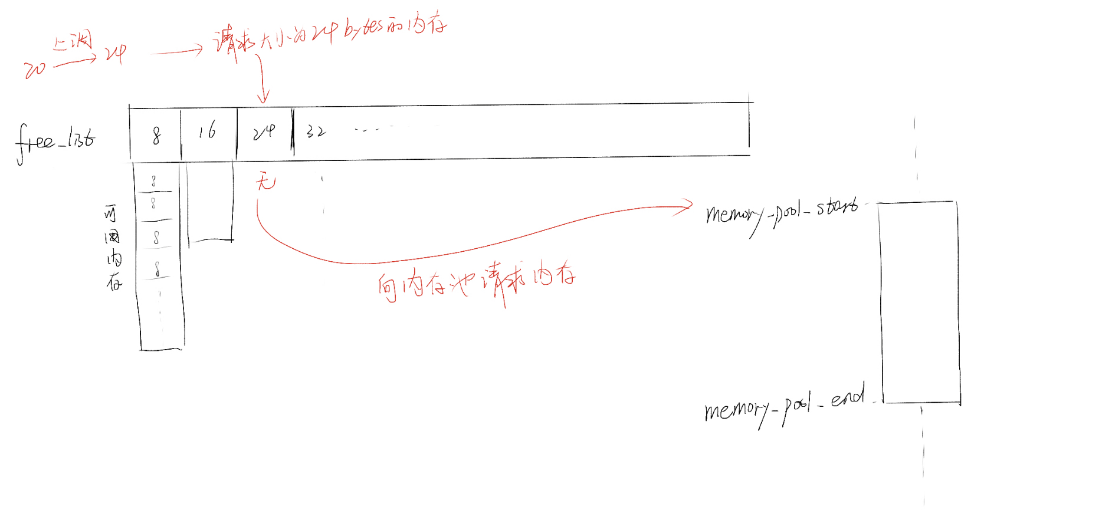
### 内存分配与释放

使用malloc()函数申请内存，malloc()不管申请内存多大都会响应，而且需要通过系统调用，因此频繁的调用malloc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，内存分配规则为：

need\_bytes>128bytes，通过malloc函数申请。这称为一级内存分配器

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。



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 Pstar\_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 Pstar\_freelist\_index(size\_t bytes) |
| 作用 | 计算出对应内存大小在freelisy上的位置 |
| 参数 | bytes调整过的内存大小 |
| 说明 | 无 |

|  |  |
| --- | --- |
| 函数名 | static void\* Pstar\_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\* Pstar\_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 |

### 对象构造与析构

开放API

|  |  |
| --- | --- |
| 函数名 | template<typename T1>  inline void construct(T1\* \_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 T1, typename T2>  inline void construct(T1 \*\_Ptr, const T2& \_Val) |
| 作用 | 在\_Ptr指向的内存上构造对象，对象值为\_Val |
| 参数 | \_Ptr指向要构造对象的地址  \_Val为对象的值 |
| 说明 | 无 |

|  |  |
| --- | --- |
| 函数名 | template<typename T>  inline void destory(T \*\_Ptr) |
| 作用 | 析构 \_Ptr指向的对象 |
| 参数 |  |
| 说明 | 无 |

|  |  |
| --- | --- |
| 函数名 | template<typename InputIterator>  inline void destory(InputIterator \_First, InputIterator \_Last) |
| 作用 | 析构[\_First, \_Last]范围内的对象 |
| 参数 |  |
| 说明 | 无 |

**内部API**

|  |  |
| --- | --- |
| 函数名 | template<typename InputIterator>  inline void Pstar\_destory(InputIterator \_First, InputIterator \_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，在重载中起到区分的作用 |
| 说明 |  |

|  |  |
| --- | --- |
| 函数名 | template<typename InputIterator>  inline void Pstar\_destory(InputIterator \_First, InputIterator \_Last, \_\_false\_type \_False) |
| 作用 | 供范围析构destory函数使用 |
| 参数 |  |
| 说明 |  |

### 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 |  |

## type\_traists

明白using(作用和typedef一样，只是typedef是C99标准)和typename关键字的作用即可容易理解

## 迭代器与反向迭代器

### 迭代器

### 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。

# 容器模块

带有Pstar\_的是内部API不对外提供，Pstar\_系列的API主要是为了提高复用，将一些常用的功能进行封装。

## Vector

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。

# 算法模块

# 测试框架