MengX
感谢努力的自己
博客园 首页 新随笔 联系 订阅 管理

C++《STL源码剖析》vector学习

写在前面：

以前竞赛只是会用vector的接口函数，这次深入了解下

参考博客：https://www.cnblogs.com/IamTing/p/4605820.html

vector源码摘录：

无空间配置器部分

```
#include<iostream>
using namespace std;
#include<memory.h>
// alloc是SGI STL的空间配置器
template <class T, class Alloc = alloc>
class vector
{
public:
    // vector的嵌套类型定义,typedefs用于提供iterator_traits<I>支持
    typedef T value_type;
    typedef value_type* pointer;
    typedef value_type* iterator;
    typedef value_type& reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
protected:
    // 这个提供STL标准的allocator接口
    typedef simple_alloc <value_type, Alloc> data_allocator;

    iterator start;              // 表示目前使用空间的头
    iterator finish;             // 表示目前使用空间的尾
    iterator end_of_storage;     // 表示实际分配内存空间的尾

    void insert_aux(iterator position, const T& x);

    // 释放分配的内存空间
    void deallocate()
    {
        // 由于使用的是data_allocator进行内存空间的分配，
        // 所以需要同样使用data_allocator::deallocate()进行释放
        // 如果直接释放，对于data_allocator内部使用内存池的版本
        // 就会发生错误
        if (start)
            data_allocator::deallocate(start, end_of_storage - start);
    }

    void fill_initialize(size_type n, const T& value)
    {
        start = allocate_and_fill(n, value);
        finish = start + n;                          // 设置当前使用内存空间的结束点
        // 构造阶段，此实作不多分配内存，
        // 所以要设置内存空间结束点和，已经使用的内存空间结束点相同
        end_of_storage = finish;
    }

public:
```

公告

昵称： MengX
园龄： 2年11个月
粉丝： 21
关注： 16
+加关注

< 2021年1月 >

日	一	二	三	四	五	六
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

解题报告(61)
训练总结(14)
模板(9)
题集(3)
暑假集训(1)

随笔分类

2019 CCPC-Wannafly Winter Camp Div2(2)
BFS(2)
cdq(2)
DFS(1)
DP(2)
LCA(1)
RMQ(1)
Tarjan(2)
并查集(2)
尺取(1)
二分/三分(1)
二分图(1)
分块(1)
计算机网络(1)

```
// 获取几种迭代器
iterator begin() { return start; }
iterator end() { return finish; }

// 返回当前对象个数
size_type size() const { return size_type(end() - begin()); }
size_type max_size() const { return size_type(-1) / sizeof(T); }
// 返回重新分配内存前最多能存储的对象个数
size_type capacity() const { return size_type(end_of_storage - begin()); }
bool empty() const { return begin() == end(); }
reference operator[](size_type n) { return *(begin() + n); }

// 本实作中默认构造出的vector不分配内存空间
vector() : start(0), finish(0), end_of_storage(0) {}


vector(size_type n, const T& value) { fill_initialize(n, value); }
vector(int n, const T& value) { fill_initialize(n, value); }
vector(long n, const T& value) { fill_initialize(n, value); }

// 需要对象提供默认构造函数
explicit vector(size_type n) { fill_initialize(n, T()); }

vector(const vector<T, Alloc>& x)
{
    start = allocate_and_copy(x.end() - x.begin(), x.begin(), x.end());
    finish = start + (x.end() - x.begin());
    end_of_storage = finish;
}

~vector()
{
    // 析构对象
    destroy(start, finish);
    // 释放内存
    deallocate();
}

vector<T, Alloc>& operator=(const vector<T, Alloc>& x);

// 提供访问函数
reference front() { return *begin(); }
reference back() { return *(end() - 1); }

void push_back(const T& x)
{
    // 内存满足条件则直接追加元素，否则需要重新分配内存空间
    if (finish != end_of_storage)
    {
        construct(finish, x);
        ++finish;
    }
    else
        insert_aux(end(), x);
}

iterator insert(iterator position, const T& x)
{
    size_type n = position - begin();
    if (finish != end_of_storage && position == end())
    {
        construct(finish, x);
        ++finish;
    }
    else
        insert_aux(position, x);
    return begin() + n;
}

iterator insert(iterator position) { return insert(position, T()); }
```

离线乱搞~莫队(4)

更多

随笔档案

2020年2月(8)

2019年11月(1)

2019年9月(1)

2019年8月(25)

2019年7月(5)

2019年5月(5)

2019年4月(5)

2019年3月(1)

2019年2月(3)

2019年1月(2)

2018年12月(2)

2018年10月(2)

2018年9月(2)

2018年8月(8)

2018年7月(9)

更多

最新评论

1. Re:逆序数&&线段树

膜

--L19

阅读排行榜

1. HDU-6315 Naive Operations 线段树(1251)
2. C++ 《STL源码剖析》vector学习(896)
3. C++ 面经常见题(738)
4. Codeforces - 449B. Jzzhu and Cities(最短路(480)
5. HDU-6273 Master of GCD(410)

评论排行榜

1. 逆序数&&线段树(1)

推荐排行榜

1. C++ 《STL源码剖析》 List学习(1)
2. 逆序数&&线段树(1)

```
void pop_back()
{
    --finish;
    destroy(finish);
}

iterator erase(iterator position)
{
    if (position + 1 != end())
        copy(position + 1, finish, position);
    --finish;
    destroy(finish);
    return position;
}


iterator erase(iterator first, iterator last)
{
    iterator i = copy(last, finish, first);
    // 析构掉需要析构的元素
    destroy(i, finish);
    finish = finish - (last - first);
    return first;
}

// 调整size, 但是并不会重新分配内存空间
void resize(size_type new_size, const T& x)
{
    if (new_size < size())
        erase(begin() + new_size, end());
    else
        insert(end(), new_size - size(), x);
}
void resize(size_type new_size) { resize(new_size, T()); }

void clear() { erase(begin(), end()); }

protected:
    // 分配空间, 并且复制对象到分配的空间处
    iterator allocate_and_fill(size_type n, const T& x)
    {
        iterator result = data_allocator::allocate(n);
        uninitialized_fill_n(result, n, x);
        return result;
    }

    template <class T, class Alloc>
    void insert_aux(iterator position, const T& x)
    {
        if (finish != end_of_storage)    // 还有备用空间
        {
            // 在备用空间起始处构造一个元素，并以vector最后一个元素值为其初值
            construct(finish, *(finish - 1));
            ++finish;
            T x_copy = x;
            copy_backward(position, finish - 2, finish - 1);
            *position = x_copy;
        }
        else   // 已无备用空间
        {
            const size_type old_size = size();
            const size_type len = old_size != 0 ? 2 * old_size : 1;
            // 以上配置元素：如果大小为0，则配置1（个元素大小）
            // 如果大小不为0，则配置原来大小的两倍
            // 前半段用来放置原数据，后半段准备用来放置新数据

            iterator new_start = data_allocator::allocate(len);  // 实际配置
            iterator new_finish = new_start;
            // 将内存重新配置
            try
            {
```

```
            // 将原vector的安插点以前的内容拷贝到新vector
            new_finish = uninitialized_copy(start, position, new_start);
            // 为新元素设定初值 x
            construct(new_finish, x);
            // 调整水位
            ++new_finish;
            // 将安插点以后的原内容也拷贝过来
            new_finish = uninitialized_copy(position, finish, new_finish);
        }
        catch(...)
        {
            // 回滚操作
            destroy(new_start, new_finish);
            data_allocator::deallocate(new_start, len);
            throw;
        }
        // 析构并释放原vector
        destroy(begin(), end());
        deallocate();

        // 调整迭代器，指向新vector
        start = new_start;
        finish = new_finish;
        end_of_storage = new_start + len;
    }
}

template <class T, class Alloc>
void insert(iterator position, size_type n, const T& x)
{
    // 如果n为0则不进行任何操作
    if (n != 0)
    {
        if (size_type(end_of_storage - finish) >= n)
        {       // 剩下的备用空间大于等于"新增元素的个数"
            T x_copy = x;
            // 以下计算插入点之后的现有元素个数
            const size_type elems_after = finish - position;
            iterator old_finish = finish;
            if (elems_after > n)
            {
                // 插入点之后的现有元素个数 大于 新增元素个数
                uninitialized_copy(finish - n, finish, finish);
                finish += n;    // 将vector 尾端标记后移
                copy_backward(position, old_finish - n, old_finish);
                fill(position, position + n, x_copy); // 从插入点开始填入新值
            }
            else
            {
                // 插入点之后的现有元素个数 小于等于 新增元素个数
                uninitialized_fill_n(finish, n - elems_after, x_copy);
                finish += n - elems_after;
                uninitialized_copy(position, old_finish, finish);
                finish += elems_after;
                fill(position, old_finish, x_copy);
            }
        }
        else
        {   // 剩下的备用空间小于"新增元素个数"（那就必须配置额外的内存）
            // 首先决定新长度：就长度的两倍 ， 或旧长度+新增元素个数
            const size_type old_size = size();
            const size_type len = old_size + max(old_size, n);
            // 以下配置新的vector空间
            iterator new_start = data_allocator::allocate(len);
            iterator new_finish = new_start;
            __STL_TRY
            {
                // 以下首先将旧的vector的插入点之前的元素复制到新空间
                new_finish = uninitialized_copy(start, position, new_start);
                // 以下再将新增元素（初值皆为n）填入新空间
                new_finish = uninitialized_fill_n(new_finish, n, x);
```

```
                    // 以下再将旧vector的插入点之后的元素复制到新空间
                    new_finish = uninitialized_copy(position, finish, new_finish);
                }
#           ifdef  __STL_USE_EXCEPTIONS
                catch(...)
                {
                    destroy(new_start, new_finish);
                    data_allocator::deallocate(new_start, len);
                    throw;
                }
#           endif /* __STL_USE_EXCEPTIONS */
                destroy(start, finish);
                deallocate();
                start = new_start;
                finish = new_finish;
                end_of_storage = new_start + len;
            }
        }
    }
};
```

本文章是笔者学习《STL源码剖析》的学习笔记，记录的是笔者的个人理解，因为个人的水平有限，难免会有理解不当的地方，而且该书出版的时间比较久，难免会有些不一样。如有不当，欢迎指出。

vector是c++中经常用到的数据结构，而且在面试时也会有提及，因此了解vector很重要。

一说到vector，我们就很容易想到另外一个与它十分相似的数据结构，关于它们之间显著的差别，我觉得是在于空间运用的灵活性上。数组是静态的，在声明的时候就要指明其具体的空间大小，而vector是动态的，随着元素的增加，它内部机制会自行扩充以容纳新元素。

这里提及一个问题。

```
#include <iostream>

int main() {
    int len;
    std::cin >> len;
    int arr[len];
    return 0;
}
```

如上的一小段代码，在VS中编译会报错，而在g++编译器中却能顺利通过，这里个人不是很理解，或许是跟编译器内部的编译规则有关系。

平时，我们要使用vector的时候，声明如下

```
std::vector<int> vec0;
std::vector<int> vec1(10);
std::vector<int> vec2(10, 0);
```

这短短的一句代码中，到底是做了些什么呢？

我们还是先看看书中给出的部分代码吧。

```
template <class T, class Alloc = alloc>
class vector {

...

protected:
  typedef simple_alloc<value_type, Alloc> data_allocator;
  iterator start;                                  //表示目前使用空间的头
  iterator finish;                                 //表示目前使用空间的尾
  iterator end_of_storage;                         //表示目前可用空间的尾
```

```
11
12   void fill_initialize(size_type n, const T& value) {
13         start = allocate_and_fill(n, value);
14         finish = start + n;
15         end_of_storage = finish;
16   }
17
18    // 分配空间并填满内容
19    iterator allocate_and_fill(size_type n, const T& x) {
20         iterator result = data_allocator::allocate(n);
21         uninitialized_fill_n(result, n, x);
22         return result;
23     }
24
25 ...
26
27 public:
28     vector() : start(0), finish(0), end_of_storage(0) {}
29     vector(size_type n, const T& value) {fill_initialize(n, value);}
30     vector(int n, const T& value) {fill_initialize(n, value);}
31     vector(long n, const T& value) {fill_initialize(n, value);}
32     explicit vector(size_type n) {fill_initialize(n, T());}
33 ...
34 }
```

上述中可以看到vector部分构造函数，其中默认构造函数只是把所有的迭代器都初始化为 0 ，这是最简单的了，但注意它并没有申请内存空间。另外 4 个大同小异，都向堆申请了大小为n的内存空间，只是初始化这些空间的时候进行的操作不一样而已： 3 个用传入的形参来进行初始化， 1 个用T类型的值初始化T()来进行初始化。这 4 个构造函数都用同一个函数fill_initialize()来进行堆空间的申请并且初始化。现在让我们看看这个函数到底干了些什么。

```
void fill_initialize(size_type n, const T& value) {
        start = allocate_and_fill(n, value);
        finish = start + n;
        end_of_storage = finish;
}
```

这个函数不难理解，它要做的工作主要是初始化迭代器。它接受两个参数n和value，n指明了要申请的堆空间大小，value指明了要初始化这些堆空间的内容，并把它们传给另外一个函数allocate_and_fill() ,该函数才是真正的申请堆空间和初始化。接着fill_initialize()就根据allocate_and_fill() 返回的迭代器来初始化start迭代器，并且初始化finish和end_of_storage迭代器。

以下再让我们看看allocate_and_fill() 函数。

```
  // 分配空间并填满内容
iterator allocate_and_fill(size_type n, const T& x) {
        iterator result = data_allocator::allocate(n);
        uninitialized_fill_n(result, n, x);
        return result;
 }
```

从函数的名字我们都可以大概知道它是要干什么的了。它接受来自fill_initialize()的两个参数，该两个参数的含义在前面已提及。然后申请堆空间并初始化。data_allocator实质上就是simple_alloc<value_type, Alloc> ，simple_alloc是SGI STL的空间配置器，SGI STL的空间配置器的简单介绍请点这里或自行谷歌。若果是SGI STL第一级配置器那么data_allocator::allocate()实质上就是直接调用c语言中的malloc()来申请堆空间；若是第二级配置器，就先考察申请区块是否大于128bytes，若是大于则转调用第一级配置器，否则就以内存池来管理，目的是为了避免太多小额区块造成内存碎片化。

在讲完data_allocator::allocate()后，我们不妨看看uninitialized_fill_n()，它定义在<stl_uninitialized.h>中。<stl_uninitialized.h>定义了一些全局函数，如uninitialized_fill_n()，用来填充或复制大块内存数据，以便提高效率。

```
template<class ForwardIterator, class SIze, class T>
ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

迭代器first指向将要初始化空间的起始处；n表示将要初始化空间的的大小；x表示初始化的值。如果[first，first+n)范围内的每一个迭代器都指向未初始化的内存，那么uninitialized_fill_n()会调用copy constructor，在该范围内产生x的副本，但注意一点是在此过程中若任何一个copy constructor丢出异常，uninitialized_fill_n()必须析构已产生的所有元素。

说完vector的构造，那么就先看看析构函数吧。

```
... ...

~vector() {
    destroy(start, finish);
    dellocate();
}

... ...
```

析构函数很简单，就调用两个函数:destroy()和dellocate()。destroy()负责对象的析构，这个下面将要讲述。dellocate()负责释放申请的堆空间。这里释放的方式又与空间配置器相关。若果是第一级配置器，就直接调用c语言中free()函数，这正如申请时的简便。但若果是第二级配置器，则比较复杂一些，具体请参考这里或自行谷歌。

上面的内容就是关于vector申请和释放堆空间的大概过程，但仅仅是申请和释放堆空间而已。

为了能继续下去，我们来看看下面的代码。

```
class Foo {..};
Foo* pf = new Foo;              // 分配内存，然后构造对象
delete pf;                      // 将对象析构，然后释放内存
```

上述的new操作符做了两件事:(1)分配内存(2)调用Foo::Foo()构造对象。

同样，delete操作符也做了两件事:(1)调用Foo::~Foo()将对象析构(2)释放内存

但是，在STL 配置器中为了精密分工，把这些操作都细分开来。具体来说就是

::construct()负责对象的构造

::destroy()负责对象的析构

alloc::allocate()负责内存分配

alloc::deallocate()负责内存释放

上面vector的内存操作所用到就是alloc::allocate()，alloc::deallocate()，这里就不在讲述了。

construct()和destroy()都包含在<stl_construct.h>内。

```
// 这是construct()函数
template <class T1, class T2>
inline void construct(T1* p, const T2& value) {
    new (P) T1(value);    // placement new;调用T1::T1(value)
}
```

上述construct()接受一个指针p和一个初值value，该函数的用途就是将初值拷贝一份到指针所指的空间上，这个还是简单明了的。

接着是destroy()。这个函数有两个版本，先看第一个版本。

```
// 第一个版本
template <class T>
inline void destroy(T* pointer) {
    pointer->~T();
}
```

第一个版本接受一个指针，调用析构函数将该指针所指的对象析构。

```
// 第二个版本
template<class ForwardIterator, class T>
inline void destroy(ForwardIterator first, ForwardIterator last, T*) {
    __destroy(first, last, value_type(first));
}
```

第二版本接受first和last两个迭代器，将[first, last)范围内的所有对象析构掉。这里还要考虑效率问题，但代码上不再展开。

好了，对对象的构建和析构的过程有了基本的认识后，再看看vector的push_back()函数的实现吧

```
1 void push_back(const T& x) {
2     if (finish != end_of_storage) {
3         construct(finish, x);
4         ++finish;
5     } else {
6         insert_aux(end(), x);
7      }
8 }
```

当我们把元素push_back到vector的尾端后，函数首先检查是否还有备用的空间，如果有的话就调用construct()函数，在finish迭代器指定的位置上构建x对象，同时改变finish迭代器，使其自增１。

然而若果没有备用空间，就需要扩充空间了，这就是insert_aux()函数所要做的。

```
 1 template <class T, class Alloc>
 2     void insert_aux(iterator position, const T& x)
 3     {
 4         if (finish != end_of_storage)    // 还有备用空间
 5         {
 6             // 在备用空间起始处构造一个元素，并以vector最后一个元素值为其初值
 7             construct(finish, *(finish - 1));
 8             ++finish;
 9             T x_copy = x;
10             copy_backward(position, finish - 2, finish - 1);
11             *position = x_copy;
12         }
13         else   // 已无备用空间
14         {
15             const size_type old_size = size();
16             const size_type len = old_size != 0 ? 2 * old_size : 1;
17             // 以上配置元素：如果大小为0，则配置1（个元素大小）
18             // 如果大小不为0，则配置原来大小的两倍
19             // 前半段用来放置原数据，后半段准备用来放置新数据
20 
21             iterator new_start = data_allocator::allocate(len);  // 实际配置
22             iterator new_finish = new_start;
23             // 将内存重新配置
24             try
25             {
26                 // uninitialized_copy()的第一个参数指向输入端的起始位置
27                 // 第二个参数指向输入端的结束位置（前闭后开的区间）
28                 // 第三个参数指向输出端(欲初始化空间)的起始处
29 
30                 // 将原vector的安插点以前的内容拷贝到新vector
31                 new_finish = uninitialized_copy(start, position, new_start);
32                 // 为新元素设定初值 x
33                 construct(new_finish, x);
34                 // 调整已使用迭代器的位置
35                 ++new_finish;
36                 // 将安插点以后的原内容也拷贝过来
37                 new_finish = uninitialized_copy(position, finish, new_finish);
38             }
39             catch(...)
40             {
41                 // 回滚操作
42                 destroy(new_start, new_finish);
43                 data_allocator::deallocate(new_start, len);
44                 throw;
45             }
46             // 析构并释放原vector
47             destroy(begin(), end());
48             deallocate();
49 
50             // 调整迭代器，指向新vector
51             start = new_start;
52             finish = new_finish;
```

```
53              end_of_storage = new_start + len;
54          }
55      }
```

从代码中可以知道，当备用空间不足时，vector做了以下的工作：

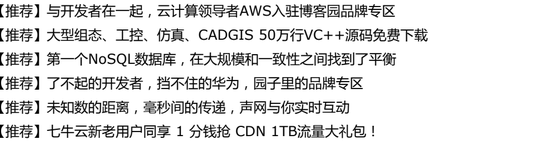
1. 重新分配空间(15~22行)：若原来的空间大小为0，则扩充空间为 1 ，否则扩充为原来的两倍。

2. 移动数据（31～37行）

3. 释放原空间（47～48行）

4. 更新迭代器（51～53行)

当调用默认构造函数构造vector时，其空间大小为0，但当我们push_back一个元素到vector尾端时，vector就进行空间扩展，大小为 1 ，以后每当备用空间用完了，就将空间大小扩展为原来的两倍。



注意的是，所谓动态增加大小，并不是在原空间之后接续新空间，（因为无法保证原空间之后上有可供分配的空间），而是以原大小的两倍来另外分配一块较大空间，因此，一旦空间重新分配，指向原vector的所有迭代器就会失效，这里要特别注意。

总结：

vector的各种操作其实离不开四个操作 堆内存的申请和释放 对象的创建和摧毁

我们在使用vector的时候 应该处理好初始大小问题 vector虽然为动态数组 但是它调整大小的代价很大

所谓动态调整大小，并不是在原vector后面添加新空间，而是申请两倍长度的vector新空间 然后将原本vector数据复制到新vector上。

对vector的任何操作 如果引起了新的空间配置 那么指向vector的所有迭代器都会失效了

MengX
关注 - 16
粉丝 - 21
+加关注

0 0

« 上一篇： C++ 专项训练错题
» 下一篇： C++ 面经常见题

posted @ 2020-02-17 17:40 MengX 阅读(896) 评论(0) 编辑 收藏

梦想不是空想