2 vector

vector概述

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。

vector的实现技术，关键在于其对大小的控制以及重新配置时的数据移动效率，即“配置新空间/数据移动/释放旧空间”的这个过程。

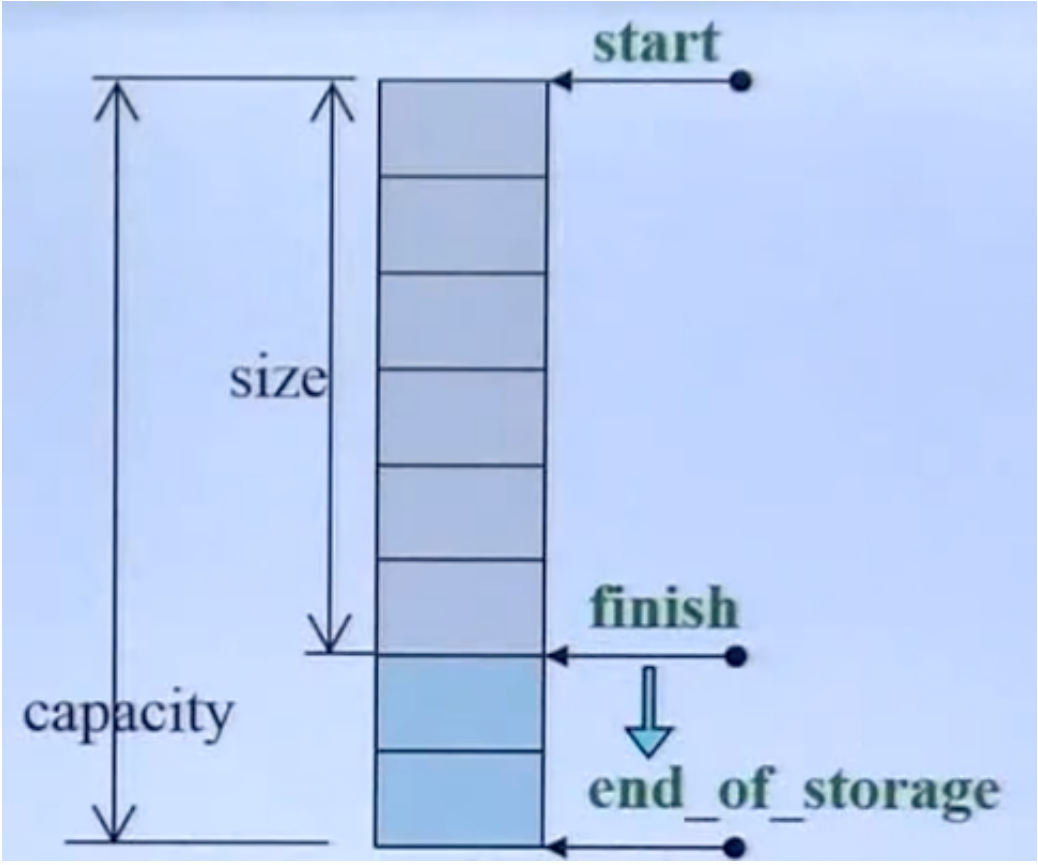
vector的主要定义如下：

```
// alloc 是SGI STL的空间配置器
template<class T,class Alloc=alloc>
class vector{
   public:
     //vector的嵌套型别定义
     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:
     //simple_alloc 是SGI STL的空间配置器
     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

     void insert_aux(iterator position,const T& x);
     void deallocate(){
         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;
         end_of_storage=finsih;
     }

  public:
     iterator begin(){return start;}
```

```
    iterator end(){return finish;}
    size_type size() const {return size_type(end()-begin());}
    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);}

    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);}
    explicit  vector(size_type n){fill_initialize(n,T());}

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

    reference front(){return *begin();}//第一个元素
    reference back() {return *(end()-1);}//最后一个元素
    void push_back(const T& x){//将元素插入至最尾端
        if(finish!=end_of_storage){
            construct(finish,x);
            ++finish;
        }
        else
           insert_aux(end(),x);
    }

    void pop_back(){//将最尾端元素取出
        --finish;
        destroy(finish);//全局函数
    }

    iterator erase(iterator position){//清除某位置上的元素
        if(position+1 !=end)
        {
           copy(position+1,finish,position);//后续元素往前移动
        }
        --finish;
        destroy(finish);
        return position;
    }

    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());}
```

```
86
87  protected:
88      //配置空间并填满内容
89      iterator allocate_and_fill(size_type n,const T& x)
90      {
91          iterator result=data_allocator::allocate(n);
92          uninitialized_fill_n(result,n,x);
93          return result;
94      }
95  };
```

这么大一串代码看着头有点晕，那就上个图来表示吧：



vector的迭代器

vector维护的是一个连续的线性空间,由于是连续线性空间,所以其迭代器所要进行的一些操作比如:operator*,operator->,operator+,operator-,operator++,operator--等等普通的指针都可以满足所以vector的迭代器就是普通指针。通过普通指针也可让vector随机存取(所以vector的迭代器是Random Access Iterator).

迭代器的类型有5种：

```
1  输入迭代器input_iterator：  只读，且只能一次读操作，支持操作：++p,p++,!=,==,=*p,p->;
2  输出迭代器output_iterator：  只写，且只能一次写操作，支持操作：++p,p++;
3  正向迭代器forward_iterator：  可多次读写，支持输入输出迭代器的所有操作；
4  双向迭代器bidirectional_iterator：  支持正向迭代器的所有操作，且支持操作：--p,--p;
5  随机访问迭代器random_access_iterator：  除了支持双向迭代器操作外，还支持：p[n],p+n,
6  n+p,p-n,p+=n,p-=n,p1-p2,p1<p2,p1>p2,p1>=p2,p1<=p2;
```

vector源码中迭代器的定义如下：

```
template<class T,class Alloc=alloc>
class vector{
   public:
       typedef     T             value_type;
       typedef     value_type*   iterator;//vector的迭代器是普通指针
};
```

其实，iterator就是一个指针，即有以下代码。

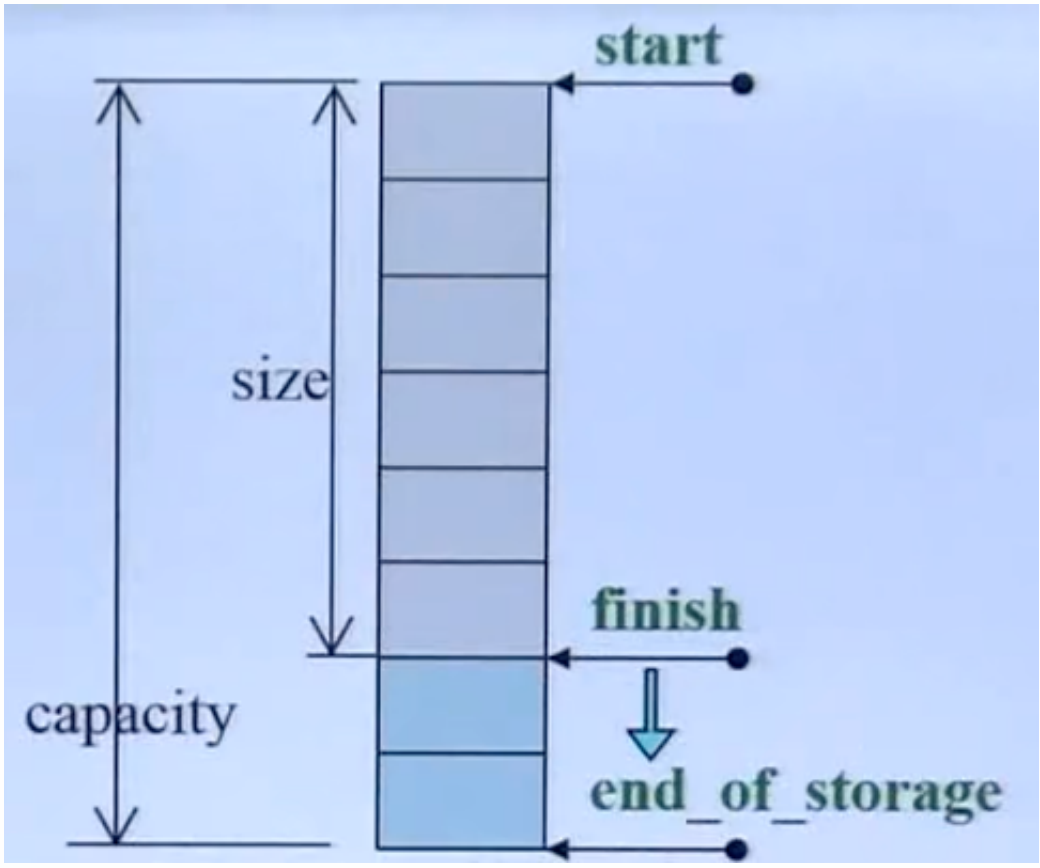
```
vector<int>::iterator ivite; // int*
vector<Shape>::iterator svite; //Shape*
```

vector的数据结构

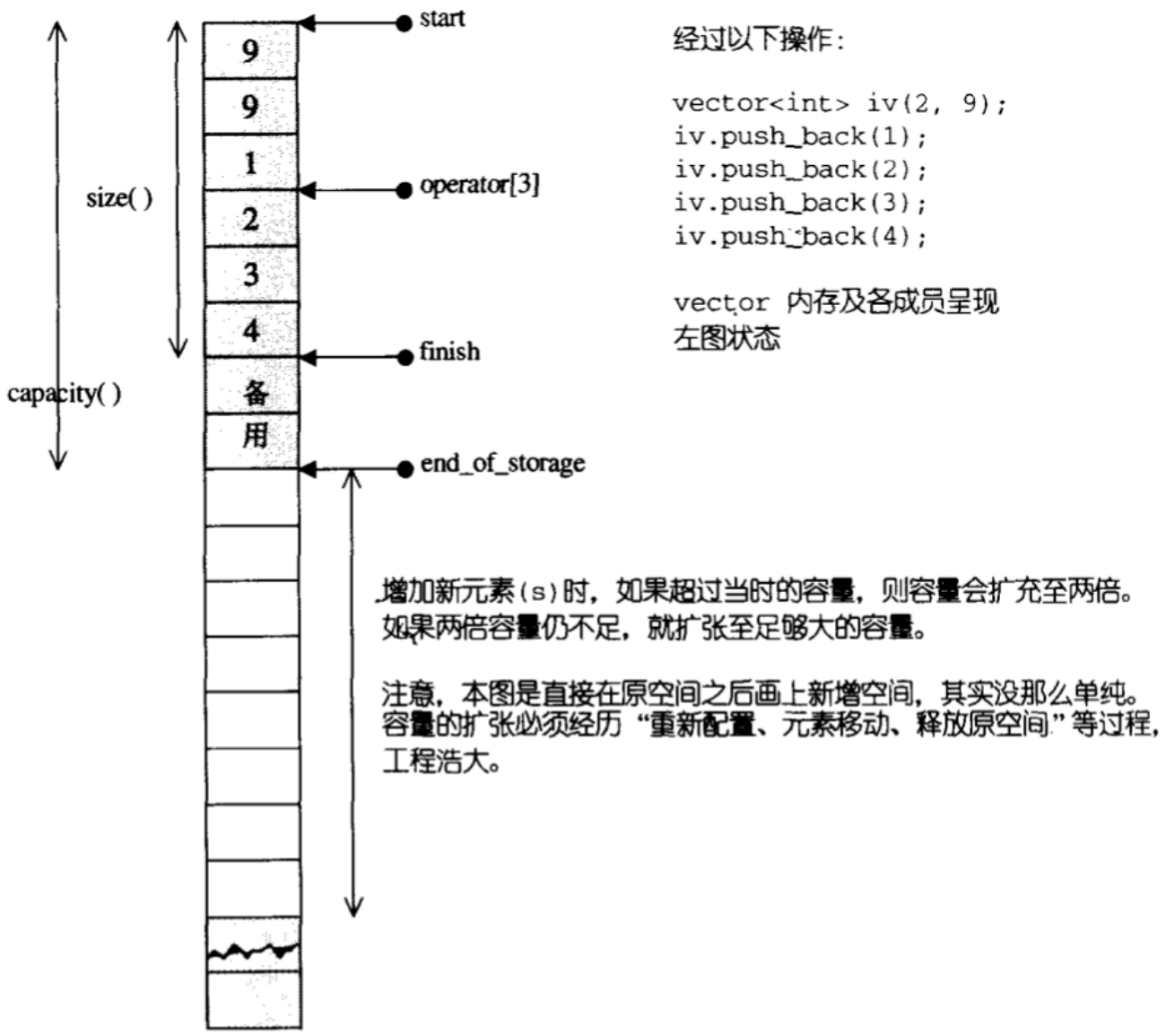
正如上面看到的，vector的数据结构如下代码，就是三根指针。vector所表示的是一片线形的连续空间，就相当于一个array，它以start和finish分别指向头和尾（左闭右开），表示连续区间内目前已经使用的范围，并以迭代器end_of_storage指向整块连续空间的尾端。

```
template<class T,class Alloc=alloc>
class vector{
...
protected:
    iterator start;
    iterator finish;
    iterator end_of_storage;
};
```

还是上面那张图：



结合图像，我们可以看到，[start,finish)这块区域，就是已经放了元素的区域;而[finish,end_of_storage)这块区域则是没有放元素的备用空间。此时vector的大小是size，也就是finish-start，end_of_storage-start则是vector目前总共能放的元素个数。如果finish==end_of_storage，则说明现在这个vector是满了的，我们需要对其扩充空间。即：“配置新空间->数据移动->释放旧空间”这个动作，参考下图。



通过这三个迭代器,就可以实现很多操作,比如提供首尾标示,大小,容量,空容器判断,[ ]运算符,最前端元素,最后端元素等。

```
template <class T,class Alloc=alloc>
class vector{
...
public:
    iterator begin(){return start;}
    iterator end(){return finish;}
    size_type size() const {return size_type(end()-begin());}
    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);}
    reference front(){return *begin();}
    reference back() {return *(end()-1);}
...
};
```

vector的构造与析构

vector提供了许多构造函数：

```
//默认构造函数
vector():start(0),finish(0),end_of_storage(0){}
//指定大小和初值
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);}
explicit  vector(size_type n){fill_initialize(n,T());}

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

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

对于指定大小和初值的构造方式，其是调用了fill_initialize()，fill_initialize()又调用了uninitialized_fill_n()，uninitialized_fill_n()会根据第一个参数的类别来判断是采用fill_n()还是反复调用construct()来完成vector的构造。

析构函数为：实现了调用析构函数和释放内存空间两个步骤。

```
~vector(){
    destroy(start,finish); //全局函数，如果不是trival destructor，则一个个调用析构函数
    deallocate(); // deallocate() is a member function of vector class
}
```

vector的扩容机制

”配置新空间->数据移动->释放旧空间“这个动作一般发生于push_back()元素时，push_back()函数的作用为：将新元素插入到vector的尾端。该函数首先检查是否还有备用空间，如果有，则直接在备用空间上构造，并调整迭代器finish，使vector增大；如果没有备用空间了，则去扩充空间（配置新空间->数据移动->释放旧空间）。

push_back()的源码如下（此为GNU）：

```
void push_back(const T& x) {
```

```
    if (finish != end_of_storage) { //若当前还有备用空间
      construct(finish, x); //将当前水位的值设为x
      ++finish; //提升水位
    }
    else
      insert_aux(end(), x);
}

template <class T, class Alloc>
void vector<T, Alloc>::insert_aux(iterator position, const T& x) {
  if (finish != end_of_storage) {
    construct(finish, *(finish - 1));
    ++finish;
    T x_copy = x;
    copy_backward(position, finish - 2, finish - 1);
    *position = x_copy;
  }
  else {
    const size_type old_size = size(); //获取之前数组的大小
    //以上原则，如果原大小为0，则配置1（个元素）
    //如果原大小不为0，则配置原大小的两倍
    //前半段用来放置原数据，后半段用来放置新数据
    const size_type len = old_size != 0 ? 2 * old_size : 1;
    iterator new_start = data_allocator::allocate(len); //重新分配新数组的起始迭代器
    iterator new_finish = new_start;
    __STL_TRY {
      new_finish = uninitialized_copy(start, position, new_start); //将旧数组的值重新分配给当前的新数组
      construct(new_finish, x); //将当前数组的水位的值设为x
      ++new_finish; //提升新数组的水位
      new_finish = uninitialized_copy(position, finish, new_finish); //这语句感觉可有可无，因为它根本就不会执行，position即last，而finish也是last
    }

#       ifdef  __STL_USE_EXCEPTIONS
    catch(...) { //如果重新构造的新数组出现异常，则销毁当前新创建的数组，并释放内存空间
      destroy(new_start, new_finish);
      data_allocator::deallocate(new_start, len);
      throw;
    }
#       endif /* __STL_USE_EXCEPTIONS */
    destroy(begin(), end()); //将旧数组的空间释放掉
    deallocate();
    start = new_start; //new_start记录新数组的起始位置
    finish = new_finish; //重新设置当前水位的指针
    end_of_storage = new_start + len; //设置新数组的容量
  }
}
```

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尚有可供配置的空间），而是以原大小的两倍另外配置一块较大空间，然后将原内容拷贝过来，然后才开始在原内容之后构造新元素，并释放原空间。

因此，对vector的任何操作，一旦引起空间重新配置，指向原vector的所有迭代器就都失效了。

需要注意的是：不同的编译器实现的扩容方式不一样，VS2015中以1.5倍扩容，GCC以2倍扩容。

vector的一些其他操作：

pop_back,erase,clear,insert

pop_back()源码:即finish回退一步，然后调用析构函数即可。

```
//将尾端元素拿掉，并调整大小
void pop_back(){
    --finish;//将尾端标记往前移动一格，表示将放弃尾端元素
    destroy(finish);
}
```

erase()与clear()源码：要注意这里第一个erase的last并不会删除，制定first和last，删除的区间为[first,last)，因为stl的迭代器都是左闭右开的。

```
//清除[first,last)中的所有元素
iterator erase(iterator first,iterator last){
    //将[last,finish)前移动到[first,finish-(last-first))
    iterator i=copy(last,finish,first);
    //析构之
    destroy(i,finish);
    //调整水位
    finish=finish-(last-first);
    return first;
}

//清除某位置上的元素
iterator erase(iterator position){
    if(position+1 !=end){
        copy(position+1,finish,position);//后续元素往前移动
    }
    --finish;
    destroy(finish);
    return position;
}

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

erase()左闭右开测试程序，由此例可得出ed所指向内容不会被erase()，也可以这么想：如果传入end()为last，end()是不可以被析构的，所以last也不会被析构。

```cpp
int main(){
    vector<int> a = {0,1,2,3,4,5,6};
    auto bg = ++a.begin(); // bg指向1
    auto ed = --a.end();
    --ed; // ed指向5
    a.erase(bg,ed);
    for(auto &i : a){ // ans : 0 5 6
        cout << i << " ";
    }
}
```

insert()源码：和erase()类似，就是中间断开增加元素，需要拷贝很多值，效率较低。

```cpp
//下面是vector::insert()实现内容
//从position开始，插入n个元素，元素初值为x
template<class T,class Alloc>
void vector<T,Alloc>::insert(iterator position,size_type n,const T& x)
{
    if(n!=0)
    {   //当n! =0才进行以下操作
        if(size_type(end_of_storage-finish)>=n)
        {
           //备用空间大于等于“新增元素个数”
           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-eles_after,x_copy);
               finish+=n-elems_after;
               uninitialized_copy(position,old_finish,finish);
               finish+=elems_after;
               fill(position,old_finish,x_copy);
           }
        }
        else{
            //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：旧长度的两倍，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);
            //配置新的vector空间
            iterator new_start=data_allocator::allocate(n);
```

```
            iterator new_finish=new_start;
            __STL_TRY{
                //以下首先将旧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(...){
                    //如有异常发生，实现“commit or rollback” semantics
                    destroy(new_start,new_finish);
                    data_allocator::deallocate(new_start,len);
                    throw;
                }
            #endif /*__STL_USE_EXCEPTIONS*/
            //以下清除并释放旧的vector
            destroy(start,finish);
            deallocate();
            //以下调整水位标记
            start=new_start;
            finish=new_finish;
            end_of_storage=new_start+len;
        }
    }
}
```

vector常见question：

1. 为什么要成倍的扩容而不是一次增加一个固定大小的容量呢?

   可参考：https://www.drdobbs.com/c-made-easier-how-vectors-grow/184401375

```
以成倍方式增长
假定有 n 个元素,倍增因子为 m;
完成这 n 个元素往一个 vector 中的 push_back 操作，需要重新分配内存的次数大约为 logm(n);
第 i 次重新分配将会导致复制 m^(i) (也就是当前的vector.size() 大小)个旧空间中元素;
n 次 push_back 操作所花费的时间复制度为O(n):
m / (m – 1)，这是一个常量，均摊分析的方法可知，vector 中 push_back 操作的时间复杂度为常量时间.

一次增加固定值大小
假定有 n 个元素,每次增加k个;
第i次增加复制的数量为为：100i
n 次 push_back 操作所花费的时间复杂度为O(n^2):
均摊下来每次push_back 操作的时间复杂度为O(n);
```

```
14  总结：对比可以发现采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到O(n)的时间复杂度，因此，使用成倍的方式扩容
```

2. 为什么是以两倍的方式扩容而不是三倍四倍，或者其他方式呢?

   可参考：https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md

   其实VC的1.5倍扩容是更佳的。

> It is well known that `std::vector` grows exponentially (at a constant factor) in order to avoid quadratic growth performance. The trick is choosing a good factor. Any factor greater than 1 ensures O(1) amortized append complexity towards infinity. But a factor that's too small (say, 1.1) causes frequent vector reallocation, and one that's too large (say, 3 or 4) forces the vector to consume much more memory than needed.
>
> The initial HP implementation by Stepanov used a growth factor of 2; i.e., whenever you'd `push_back` into a vector without there being room, it would double the current capacity. This was not a good choice: it can be mathematically proven that a growth factor of 2 is rigorously the *worst* possible because it never allows the vector to reuse any of its previously-allocated memory. Despite other compilers reducing the growth factor to 1.5, gcc has staunchly maintained its factor of 2. This makes `std::vector` cache- unfriendly and memory manager unfriendly.

当使用2作为倍数增长时，每次扩展的尺寸的刚好大于之前所分配的总和。换而言之，之前分配的内存空间不可以被使用，对缓存不友好。

$$c\sum_{i=0}^{n}2^i = c(2^{n+1}-1) < c2^{n+1}$$

1.5倍增长和2倍增长对比：

```
k = 2, c = 4
0123
    01234567
            012345789ABCDEF
                        0123456789ABCDEF0123456789ABCDEF
                                                        012345...

k = 1.5, c = 4
0123
    012345
          012345678
                  0123456789ABCD
                              0123456789ABCDEF0123
0123456789ABCDEF0123456789ABCD
                            0123456789ABCDEF0123456789ABCDEF...
```

3. vector与list的区别与应用?

   区别:

对比项	vector	list
数据结构	顺序表	双向链表
随机访问	支持	不支持
插入删除	有内存拷贝	较快
迭代器类型	random_access_iterator	bidirectional_iterator

   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。

4. resize和reserve区别?

参考:https://www.cnblogs.com/cxl-/p/14482639.html

3. vector?

vector是vector的一个特化版本,之前一个byte存一个bool,在vector的设计中一个byte存8个bool,用一个bit来表示true和false。

详细的以后再说。

参考文献:

[1] https://blog.csdn.net/vjhghjghj/article/details/88713401

[2] https://blog.csdn.net/sinat_33442459/article/details/75142672

[3] https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md

[4] https://www.drdobbs.com/c-made-easier-how-vectors-grow/184401375

[5] 侯捷.STL源码剖析[M].武汉：华中科技大学出版社，2002.6：115-128.