问题

vector 与 C 中的数组不同，是一个数组大小可以动态变化的容器。那么这个变化过程具体是怎么变的呢，怎样才可以提高 vector 的使用效率呢？

原理概述

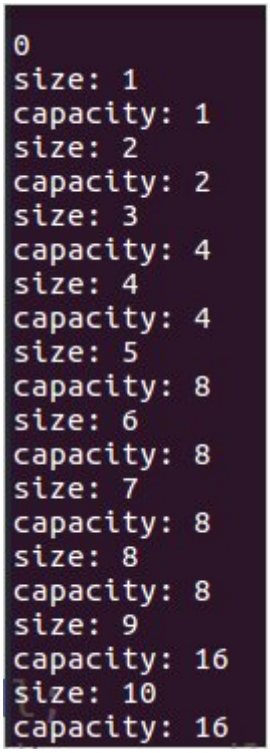
1. vector 存储的空间在内存中是连续的，如果 vector 现有空间已存满元素，在 push_back 新增数据的时候就要分配一块更大的内存，将原来的数据 copy 过来，接着释放之前的内存，再在新的内存空间中存入新增的元素。
2. 不同编译器对 vector 的扩容方式实现不一样，在 vs 中以 1.5 倍扩容，而在 gcc 中以 2 倍扩容，后面我们会看到以 1.5 倍扩容的方式效果更好。
3. vector 的初始的扩容方式代价太大，初始扩容效率低，需要频繁增长，不仅操作效率比较低，而且频繁的向操作系统申请内存容易造成过多的内存碎片，所以这个时候需要合理使用`resize()`和`reserve()`方法提高效率减少内存碎片的。
4. 对 vector 的任何操作，一旦引起空间重新配置，指向原 vector 的所有迭代器就都会失效。

gcc 中 vector 的扩容过程

我电脑系统是ubuntu，所以下面用一段代码来直观地体现在 gcc 中 vector 的扩容过程，需要说明的是 size() 表示的是 vector 当前的元素个数，而 capacity() 表示的是 vector 当前的容量大小，也就是当前给该 vector 分配的内存空间大小，当元素个数超过 capacity() 时，就需要扩容了。

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> vec;
    cout << vec.capacity() << endl;
    for (int i = 0; i<10; ++i)
    {
        vec.push_back(i);
        cout << "size: " << vec.size() << endl;
        cout << "capacity: " << vec.capacity() << endl;
    }
    system("pause");
    return 0;
}
```

输出如下：



为什么以成倍的方式扩容而不是一次增加一个固定大小的容量

假设我们需要往 vector 中存储 n 个元素，成倍方式增长存储空间的话，那么均摊到每一次的 push_back 操作的时间复杂度为 O(1) ，而一次增加固定值空间的方式均摊到每一次的 push_back 操作的时间复杂度为 O(n) 的时间。（具体就不展开了，相信算法面试官也不会问得这么细，又不是做底层开发的，问这么细的意义不大）

那为什么是以 2 倍或者 1.5 倍进行扩容呢

我们用个例子算算看：

两倍扩容

- 假设我们一开始申请了 16Byte 的空间。
- 当需要更多空间的时候，将首先申请 32Byte，然后释放掉之前的 16Byte。这释放掉的16Byte 的空间就闲置在了内存中。
- 当还需要更多空间的时候，你将首先申请 64Byte，然后释放掉之前的 32Byte。这将在内存中留下一个48Byte 的闲置空间（假定之前的 16Byte 和此时释放的32Byte 合并）
- 当还需要更多空间的时候，你将首先申请128Byte，然后释放掉之前的 64 Byte。这将在内存中留下一个112Byte 的闲置空间（假定所有之前释放的空间都合并成了一个块）

这时你就会发现一个问题，要申请的空间都比之前释放的空间合并起来的都大，因此无法循环利用之前释放的空间。

1.5 倍扩容

- 假设我们一开始申请了 16Byte 的空间。
- 当需要更多空间的时候，将申请 24 Byte ，然后释放掉 16 ，在内存中留下 16Byte 的空闲空间。
- 当需要更多空间的时候，将申请 36 Byte，然后释放掉 24，在内存中留下 40Byte (16 + 24)的空闲空间。
- 当需要更多空间的时候，将申请 54 Byte，然后释放 36，在内存中留下 76Byte。
- 当需要更多空间的时候，将申请 81 Byte，然后释放 54， 在内存中留下 130Byte。

- 当需要更多空间的时候，将申请 122 Byte 的空间（复用内存中闲置的 130Byte）

从上面你可以看到当要申请122 Bybes 空间的时候，之前释放的空间合并起来已经比要申请的空间大了，这时我们就可以利用之前释放的空间来充当新空间，从而减少了内存碎片风险，重复利用高更高。

所以说明了 1.5 倍扩容方法比 2 倍的好，也解释了为什么不采用更高倍的扩容方法而一般都采用 2 倍或者 1.5 倍的问题。这是空间和时间的权衡，空间分配地越多，平摊时间复杂度越低，但浪费空间也多。最好把增长因子设在 (1,2) 之间。

resize() 和 reserve() 区别

首先介绍下 vector 中与空间大小相关的四个函数含义：

- size()：返回vector中的元素个数
- capacity()：返回vector能存储元素的总数
- resize()操作：创建指定数量的的元素并指定vector的存储空间
- reserve()操作：指定vector的元素总数

`size()`函数返回的是已用空间大小，`capacity()`返回的是总空间大小，`capacity()-size()`则是剩余的可用空间大小。当`size()`和`capacity()`相等，说明 vector 目前的空间已被用完，如果再添加新元素，则会引起 vector 空间的动态增长。

由于动态增长会引起重新分配内存空间、拷贝原空间、释放原空间，这些过程会降低程序效率。因此，可以使用 `reserve(n)` 预先分配一块较大的指定大小的内存空间，这样当指定大小的内存空间未使用完时，是不会重新分配内存空间的，这样便提升了效率。只有当`n>capacity()`时，调用`reserve(n)`才会改变vector容量，不然保持 capacity() 不变。

1. `resize()` 既修改 capacity 的大小，也修改 size 的大小，即分配了空间的同时也创建了对象。而 `reserve()` 只修改 capacity 的大小，不修改 size 大小。
2. 使用`reserve()`预先分配一块内存后，在空间未满的情况下，不会引起重新分配，从而提升了效率。
3. 当`reserve()`分配的空间比原空间小时，是不会引起重新分配的。
4. 用`reserve(size_type)`只是扩大 capacity 值，这些内存空间可能还是“野”的，如果此时使用 “[ ]” 来访问，则可能会越界。而`resize(size_type new_size)`会真正使容器具有 new_size 个对象，这时可以使用 '[ ]' 来访问。
5. 下面有一段代码说明一下关键的地方:

```
vector<int> v1;
v1.resize(100); // size() == 100, capacity()== 100，且这 100 个元素都默认为 0
v1.push_back(3); // 这时元素 3 所在的索引为100，size() 变成 101，而 capacity() 变为 200
vector<int> v2;
v2.reserve(100); // 这时的 size() == 0, capacity()== 100
v2.push_back(3); // 这时元素 3 所在的索引为0，size() 变成 1，而 capacity() 依然是100
```

参考资料

vector扩容原理说明
c++STL vector扩容过程
C++ vector中的resize，reserve，size和capacity函数讲解