Vector类中有关Spare Capacity的分析、测试和结果说明

October 17, 2021

1. SPARE_CAPACITY在类中的作用的分析：

    1. SPARE_CAPACITY的含义和作用：根据Vector.h头文件程序，在构造Vector时，内存的容量并不是用户输入的initSize，而是initSize + SPARE_CAPACITY。SPARE_CAPACITY给Vector保留了适当容量，这是因为：

        I 当容器实际的大小超过theCapacity的大小时，Vector将调用reserve()函数分配一段大小为(2*theCapacity + 1)的新的内存，并将原内容移动过去，而这是一个很耗费时间、且占用不少内存空间的过程。

        使用SPARE_CAPACITY为容器分配一定的保留容量，可以避免程序频繁地进行扩容以致消耗大量时间，在一定程度上解决这个问题。

    2. SPARE_CAPACITY对操作效率的影响：在书第90页中，分析代码可得，SPARE_CAPACITY对操作效率的影响主要体现在push_back()操作上。如果执行push_back()时检测到已达最大容量，那么程序就分配一段新的内存。当SPARE_CAPACITY偏小时，我们不断执行push_back()操作时，系统就会频繁的分配内存，由此消耗大量的时间；当SPARE_CAPACITY偏大时，每次构造Vector的实例时，都会构造一个过大的容器，这将造成空间的浪费，而申请大容量的内存也会消耗更多的时间。因此，SPARE_CAPACITY过大或者过小都会造成操作效率的降低，它应该存在一个最优的取值。

2. SPARE_CAPACITY的最优取值的测试：

    1. 理论分析：

        I 根据题目条件，已知用户**初始输入**的Vector的大小是从1-N均匀分布的。为了得到SPARE_CAPACITY的最优取值，我们首先需要构建其评价体系。根据前文的分析，SPARE_CAPACITY主要会对程序的时空效率产生影响，因此我将从时间和空间两方面进行分析。

II 时间方面，根据前文的分析，push_back()操作的次数会对程序运行的时间产生影响。为了对push_back()操作次数有一个合理的估计，将有如下假设：

A．**一系列初始大小为x的Vector，其最终平均增加的容量是一个关于x的函数f(x)=[k*x]，其中k是一个与Vector实际性质有关的常数，[]表示上取整。**

对此的合理性解释是，相同性质（类型）的Vector，容量越大可能的改动就越多，因此可假设其呈正比关系。对于k，如果vector类用于记录班级成员，那么k的值会很小(如0.1)；如果是用于记录消费清单，那么k的值可能会很大(如10)。基于这个假设，当SPARE_CAPACITY逐渐增大，大小为i的Vector通过push_back()操作增大为f(i)时，一方面调用reserve()的次数减少了时间消耗，另一方面每次分配的内存容量变大又增加了时间消耗。**具体哪个因素起主导作用需要程序的验证。**

III 空间方面，显然有，当SPARE_CAPACITY增加时，一方面分配的内存空间增加；另一方面对于相同大小的Vector，当其达到最大容量时，在绝大多数情况下空闲的内存空间也在增加。因此可以说，SPARE_CAPACITY越大，程序的空间效率越低。

IV 对于SPARE_CAPACITY的最优值，我们希望它是一个时空效率都较优的解，而不是在某一方面拥有极高的效率而在另一方面效率很低的解。由于网络上并没有足够的已知数据来给时间和空间效率赋予合适的权重，对于不同的SPARE_CAPACITY,我将分别对时间效率和空间效率进行排序，然后将其时间和空间效率的位次**相乘**(这样可以保证解的时空效率均衡且较优)，作为该SPARE_CAPACITY的时空效率的综合值，并以此得出最优解。(**这也就是下文程序的测试思路.**)

V 对于一系列大小为1-N均匀分布的Vector，其增加的容量的最大值为f(N)=[k*N]。对于大于f(N)的SPARE_CAPACITY，它仅仅是占用了更多的内存空间而已。因此，仅需考虑范围在1-f(N)之间的SPARE_CAPACITY。而根据前文算法得出的SPARE_CAPACITY的最优值，我们期待它是一个关于f(N)的正比例函数optimal=p*f(N)。这样的话，实际应用时我们仅需知道k和N,即可得出最优值。另一方面，根据前文对于时空效率的理论分析，可以大致得出SPARE_CAPACITY的最优值不会是一个很小或者很大的数，即p不会是一个很接近0或者1的值。较为精确的p的值需要运行程序得出(因为时间效率方面哪一个因素起主导作用是有待程序确认的)。

2．测试说明：

I 测试输入：

A. 运行 make 命令，自动编译 main.cpp 并生成可执行文件 test；

B. 运行 bash run 命令后，**在同一行后输入三个正整数(用空格间隔)，它们的含义如下。** 第一个参数：程序 main.cpp 运行的次数； 第二个参数：N的值；第三个参数：k的值。

C. 注意事项：一方面k*N的值不能过小，否则会产生偶然误差；另一方面由于算法的时间复杂度是O(N^3),如果k*N的值过大会使程序运行缓慢。k*N的值**建议在50-200之间。**

II 测试输出：

A. 每一次运行 main.cpp，都将输出最优解的大小以及p的值(p即最优解的大小和k*N的比值)。程序已经给输出做出了清晰的解释。

3. 测试结果和最终结论：通过不断改变k和N的值，可以发现p的值在某一个较小的范围内波动。大量测试后，发现p存在一个大致范围：一个以0.25为中心的小邻域。由此我们可以得出结论，当k已知的vector的大小为1-N均匀分布时，SPARE_CAPACITY的最优值约为0.25*f(N)。一种可能的解释是，当SPARE_CAPACITY增加时，由于时间效率方面的内存配分次数减少和容量增加两个因素的共同作用，时间效率在某一小段是减小的（内存分配容量增加占主导因素），此后时间效率增加（内存分配次数减少占主导因素）；另一方面，在SPARE_CAPACITY较大时，其空间效率降低的幅度大于时间效率增加的幅度。也就是说，比起程序运行时间的增加，浪费的内存增加得更为明显。此外，p=0.25也较为符合实际应用中给容器留的余量大小。综上，可以说程序测试的结果是合理的。