D3文档

1.任务概要：

编写代码测试所有可能的遍历三维数组的方式下，不同规模的数组的遍历时间并分析。

2.代码分析：

略，详见源代码

3.结果比较：

根据分析，一共有六种不同的遍历方法。为了方便起见，用i,j,k分别指代最内层，次外层和最外层。并将输出的时间信息储存在runtime.txt文件中。

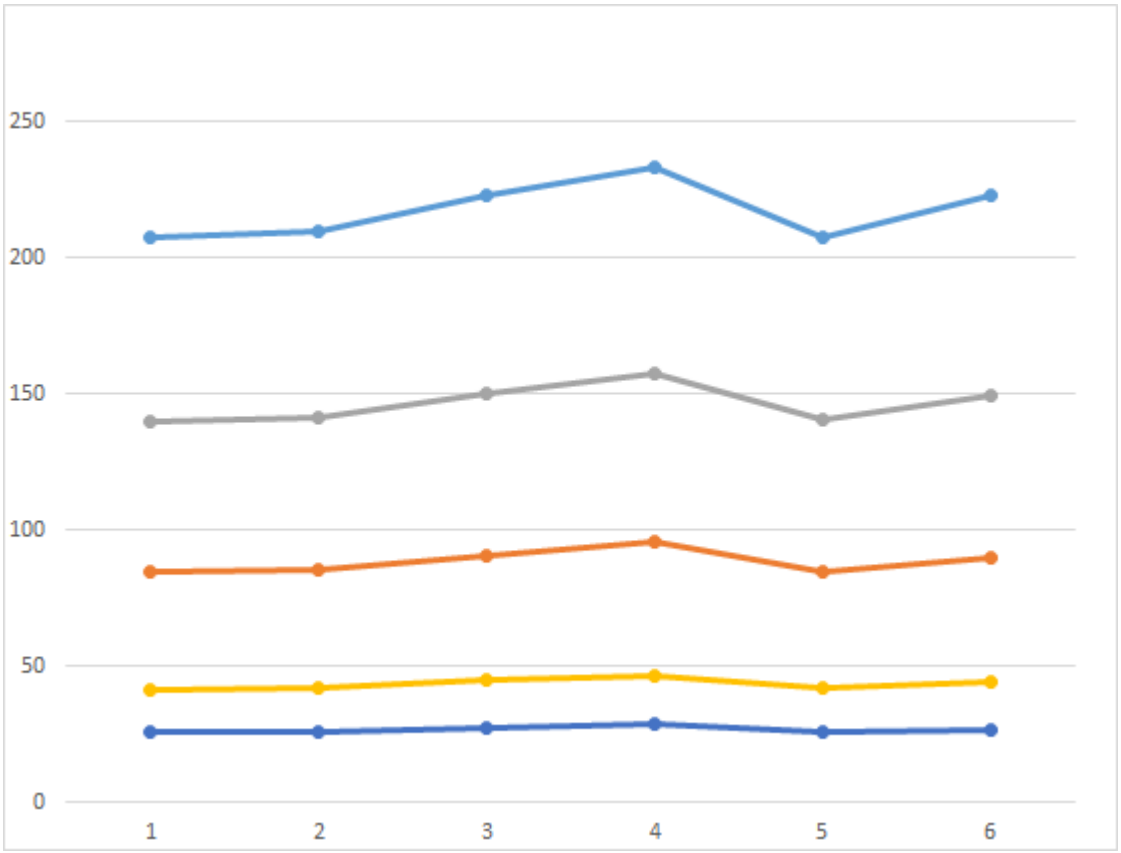
> 一共有六种访问顺序（i,j,k）(i,k,j) （k,i,j） （k,j,i） （j,i,k） （j,k,i),分别编号为1-6，生成的时间统计数据表格如下：

N\次序	1(i,j,k)	2(i,k,j)	3(k,i,j)	4(k,j,i)	5(j,i,k)	6(j,k,i)
16	0.008	0.007	0.01	0.007	0.006	0.006
32	0.055	0.056	0.064	0.056	0.054	0.058
64	0.427	0.415	0.42	0.422	0.415	0.421
100	1.581	1.598	1.7	1.684	1.566	1.565
128	3.237	3.258	3.448	3.513	3.259	3.282
180	9.006	9.061	9.622	9.861	9.052	9.151
256	26.017	26.203	27.64	28.787	25.915	26.933
300	41.698	42.052	44.722	46.609	41.741	44.021
380	84.926	85.601	90.892	95.569	84.701	89.877
450	139.927	141.342	150.531	157.327	140.453	149.452
512	207.716	210.136	223.224	233.097	207.607	222.689

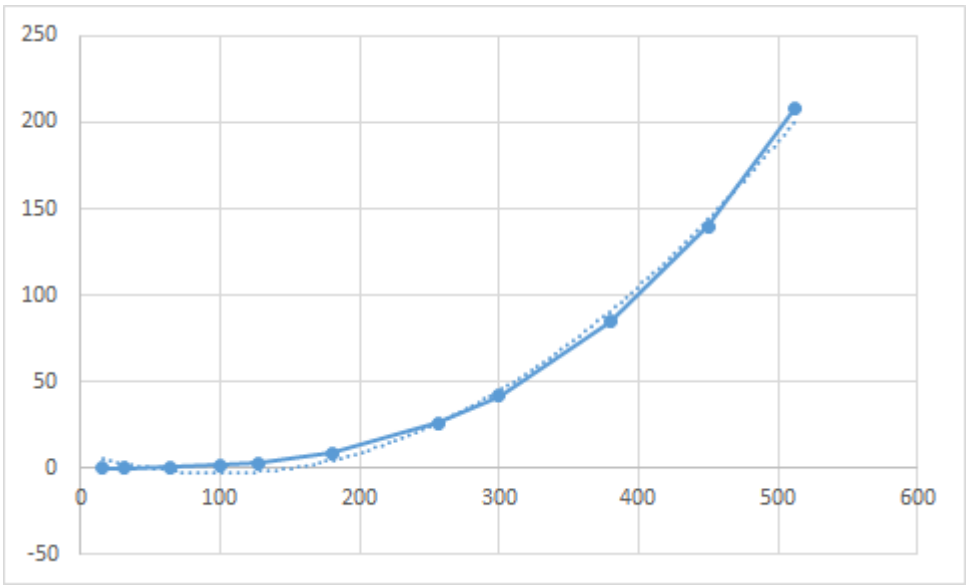
由于比例标度的问题，在此我们选取部分规模的的数据进行图表化分析：

如下图，从下到上分别代表N=512,N=450,...,N=256的数据规模，纵坐标表示遍历总时间，单位为秒

从图表和表格可见，对于不同规模的三维数组，总体上的访问顺序是1<=5<2<=6<4,（i,j,k）（内层->中层->外层）和（j,i,k）（内层->外层->中层）的访问顺序最快，而（k,j,i）（外层->中层->内层）的顺序最慢。这种差距在规模较小时不太明显，在规模较大时较为明显。



而对于同样的访问顺序，数组访问时间随着规模的变化图如下（选取（i,j,k）的顺序作为代表）



总体上说访问时间随规模N的变化成三次方的多项式增长，这是因为规模为N，总元素个数为N^3,呈立方分布

4.原因分析：

以下主要分析不同访问方式对于时间的影响。

对于一个二维数组，实际上可以理解为一维数组的数组，例如一个二维数组`a[2][3]={0},可以将其理解为一个具有两个元素的，长度为3的一维数组的数组，即` a={{0,0,0},{0,0,0}};一维数组的相邻元素物理地址相邻。由于在读取当前内存的时候，cache可能会将当前内存附近的内存进行预读取的缓存，而cache的访问速度远远高于内存，因此当横向访问即先访问内层时相邻的元素在物理上是相邻的，从缓存区命中率更高，因此比从纵向访问更快。

而对于三维数组，可以理解为二维数组的数组，如`a[2][3][2]={0}`等效为

a={{{0,0},{0,0},{0,0}},{{0,0},{0,0},{0,0}},{{0,0},{0,0},{0,0}}}。同样按照物理相邻的顺序访问数组元素会因为cache机制加快访问。而1(i,j,k)和5(j,i,k)的访问方法都是最先访问最内层，按照物理的顺序读取，因此速度最快。由于cache很小，因此一次只能加载小部分内存，所以i,j的访问顺序影响并不大，只在每次读完一个最内层的时候可能有少许影响。而对于方法4（k,j,i）则完全和1相反，连续读取的两个元素在物理上并不相邻，例如`a[1][20][30]`和`a[2][20][30]`在物理地址上差的很多，几乎无法缓存预载入加速读取，因此访问顺序最慢。而对于其他访问方式的时间差异，原理基本相同。