

Computing the dot product of two vectors

name: 吕梓翀

student id: 12313605

综述:

本次项目分别使用 java 和 C 语言，实现了计算两个向量点积的程序。并设法比较他们的运行效率，思考其原因。本项目是在英特尔 i7 处理器下，在 linux 系统下运行并检测时间的。为了方便进行检测，我还设计了一个 java 程序来生成随机的测试数据，并保存在 txt 文件内部。通过这样的方式，我可以通过调整计时的位置，来比较 C 语言与 java 语言在多种维度上的不同。在本次项目中，我重新安装了 ubuntu 的系统，这样可以更好的控制 cpu 的工作，在检测项目运行时间时，设计优先级，以及隔离 cpu 来隔绝系统服务以及电脑其他工作的干扰。另外，linux 有精度更加高的计时函数，“clock_gettime()”，来进行计时。

简要描述项目代码:

输入:

输入的 txt 文件形如:

```
int
4
-188, -151, -68, -185
-177, 141, 78, -21
```

第一行代表数据类型，第二行代表向量的维度数，后面两行是向量，我将他们用逗号隔开。如果是 signed char 类型，会用单引号包裹字符，便于处理。

以 C 语言版本为例:

```
typedef enum {
    INT,
    FLOAT,
    DOUBLE,
    SHORT,
    SIGNEDCHAR,
    UNKNOWN
} DataType;
```

首先，我设置了一个枚举型，来记录下可能出现的数据类型，为后续的处理作准备。此外设置了 UNKNOWN 来处理异常的输入。

```
FILE *file = fopen(filename, "r");
if (!file) {
    printf("Error opening file!\n");
    return 1;
}
```

以只读方式打开文件

```

// 读取数据类型
char datatype[50];
if (!fgets(datatype, sizeof(datatype), file)) {
    printf("Error reading datatype!\n");
    fclose(file);
    return 1;
}
// 去掉换行符
datatype[strcspn(datatype, "\n")] = '\0';

// 读取向量大小
int size;
if (fscanf(file, "%d\n", &size) != 1) {
    printf("Error reading vector size!\n");
    fclose(file);
    return 1;
}

// 读取两行向量
char line1[5000], line2[5000];
if (!fgets(line1, sizeof(line1), file) || !fgets(line2, sizeof(line2), file)) {
    printf("Error reading vectors!\n");
    fclose(file);
    return 1;
}
fclose(file);

```

诸行读取目标内容，并储存起来。这里我暂时规定读取向量行的最大长度是 5000,这意味着即使是读取 double 类型的数据，也可以进行 1000 维度的测试，这是一个比较大的数据了，后续测试需要时可以再更改。在读取完所有信息后，关闭 file。

下面解释点积的运算：

不同数据类型的情况比较雷同，所以这里以 short 和 int 的处理为例：

```

case INT: {
    int *vector1 = (int *)malloc(size * sizeof(int));
    int *vector2 = (int *)malloc(size * sizeof(int));
    if (!vector1 || !vector2) {
        printf("Memory allocation failed!\n");
        free(vector1);
        free(vector2);
        return 1;
    }

    char *token;
    token = strtok(line1, ",");
    for (int i = 0; i < size && token; i++) {
        vector1[i] = atoi(token);
        token = strtok(NULL, ",");
    }
    token = strtok(line2, ",");
    for (int i = 0; i < size && token; i++) {
        vector2[i] = atoi(token);
        token = strtok(NULL, ",");
    }

    int dot_product = 0;
    for (int i = 0; i < size; i++) {
        dot_product += vector1[i] * vector2[i];
    }
    printf("Dot product: %d\n", dot_product);

    free(vector1);
    free(vector2);
    break;
}

```

首先我使用的是 switch，用来将不同的数据类型进行不同的处理。然后，我为两个向量申请了一些内存。因为本项目需要考虑的是原生数据类型，所以我不将类型进行转换，如把 int 转为 long 存其来。这样可能带来溢出的情况，所以我在生成数据时调小了生成数据的上界。如果申请内存失败，我也做了相应的判断，来处理异常情况的发生。

关于解析读如的向量，我首先使用 `strtok` 将字符串按照 “，” 进行分割，然后通过循环，利用 `atoi` 将字符串转换为 `int` 类型的整数进行处理。

```
// 计算点积
long dot_product = 0;
for (int i = 0; i < size; i++) {
    dot_product += vector1[i] * vector2[i];
}
printf("Dot product: %ld\n", dot_product);

free(vector1);
free(vector2);
break;
```

然后就是对点积进行的计算，并在计算后释放内存空间。

Java 版本：

Java 中的 `switch` 可以直接输入字符串，可以省去枚举型，但是为了保证实验中的公平性，尽量保证两个版本的代码一致，我也在 `java` 中使用了枚举型。其余的部分思路与实现与 C 语言版本的几乎相同，由于 `java` 中不存在指针与内存分配的部分，全部用数组简单代替即可。这里就不过多赘述，因为本实验的核心点在于比较两个版本的程序。

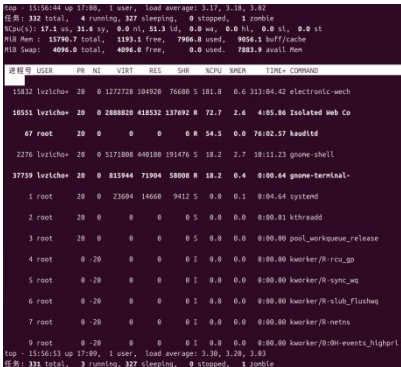
效率的比较实验：

注：本版块展示的可视化图片均由我编写的 `python` 程序生成

这一部分是本次实验的重点内容，所以下面我将要优先阐述一下我的实验思路：

首先，在 `window` 系统下简单检测了我的程序后，我意外的发现，即使是相同的程序，运行时间也出现了较大的波动，我将他们统计下来，计算了平均值和标准差后发现，他们确实波动情况很大，无法作为直接的实验数据。此外为 `window` 系统的时间测量得到的结果在 C 语言中是以秒为单位的，我认为这很不准确。在网上搜索信息后，我得出了导致程序运行波动的几个原因，下面我将仔细的阐述他们，并且附上我的解决办法：

1. 在运行程序的时候，`cpu` 不仅运行了测试的程序，还在进行一些系统服务的运行，这一点通过任务管理器就可以看出来。

The image shows a screenshot of Windows Task Manager. The 'Performance' tab is selected, displaying system metrics: CPU (1.0%), Memory (1.0%), Disk (0.0%), and Network (0.0%). Below this, the 'Processes' tab is active, showing a list of running tasks. The 'CPU' column is highlighted, showing that the 'System' process is using 1.0% of the CPU. Other processes like 'System Idle Time', 'smss.exe', 'cmd.exe', and 'java.exe' are also listed with their respective CPU usage percentages.

而这个系统服务对于 `cpu` 资源的占用并不是固定的，在不同的时间，系统服务对 `cpu` 的使用情况也不同。

对于这个问题，我的解决方法是，更换 `window` 系统为 `linux` 系统，将 `cpu` 隔离，然后将程序绑定到固定的 `cpu` 上并设置运行程序的优先级为 -20，这样一来就可以让程序的运行时间比较稳定了。

```
lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2$ java dot
Enter filename: tf500
Dot product: 155949.94
程序运行时间：3287653纳秒

lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2$ java dot
Enter filename: tf500
Dot product: 155949.94
程序运行时间：4128127纳秒
```

```
lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2$ sudo nice --20 taskset -c 1 java dot
Enter filename: tf500
Dot product: 155949.94
程序运行时间: 7933662纳秒
lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2$ sudo nice --20 taskset -c 1 java dot
Enter filename: tf500
Dot product: 155949.94
程序运行时间: 8143597纳秒
```

这是我在 java 程序运行下对于这个改动的检测。数据证明这样的改动效果不错。

下面是设置的命令行语句:

CPU 隔离:

```
sudo nano /etc/default/grub
```

```
GRUB_CMDLINE_LINUX="quiet splash isolcpus=1,2"
```

```
sudo update-grub
```

设置 cpu 亲和性:

```
taskset -c 1 ./a.out
```

设置优先级:

```
nice -n 5 ./a.out
```

2. 对于 java 来说,在我更换了系统之后,仍然出现了运行时间波动很大的情况的出现,为了解决这个问题,我在网上查阅到了一篇文章:

<https://www.cnblogs.com/niejunlei/p/14435438.html>

在该文章中,解释了 java 性能不稳定可能是因为 JVM 在程序运行的初期,可能会进行 GC 机制自动清理内存,以及简单的程序运行优化,这些会导致时间波动。于是我增加了一个固定的简单的点积运算程序,来预热 JVM,并且在启动程序后等待十秒钟,再进行测试。

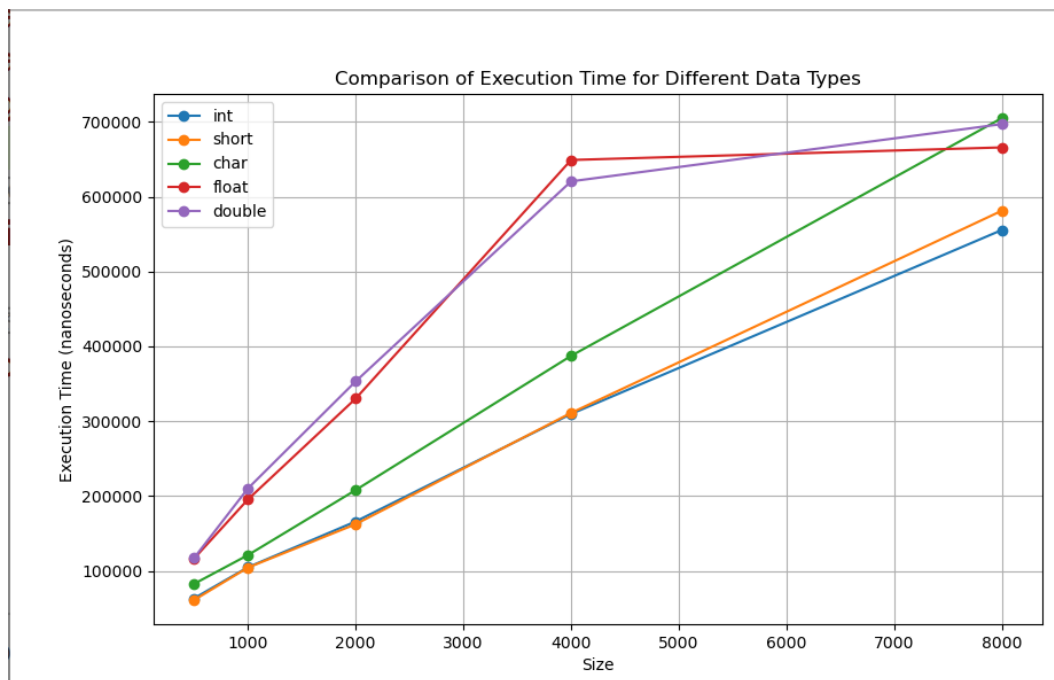
3. 另外,现在的电脑比较先进,cpu 会根据目前的任务大小来自动调整频率,这也是一个影响的地方。所以为使用 acpi 将 cpu 设置为 performance 模式,禁止其自动调频。

```
lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2$ sudo acpi -s performance
```

测量不同数据类型在 C 与 java 下的运行速度:

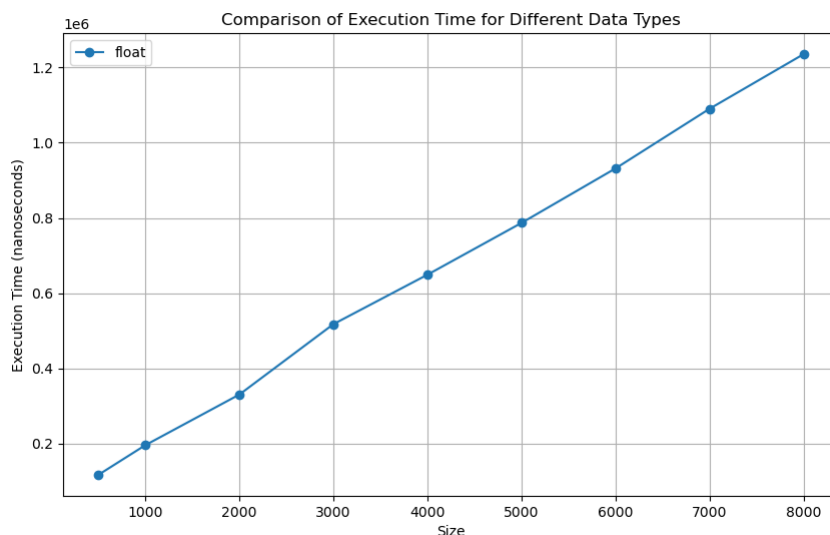
首先是数据规模的选用方面,我没有使用像 5,10,20 这样的小数据类型,而是使用 500, 1000,2000,这是因为处理器的运行速度比较快,小数据类型的增加速度缓慢且本身处理时间极短,在程序还需要处理文件读写的情况下,这种变化就会不明显。

C 语言:



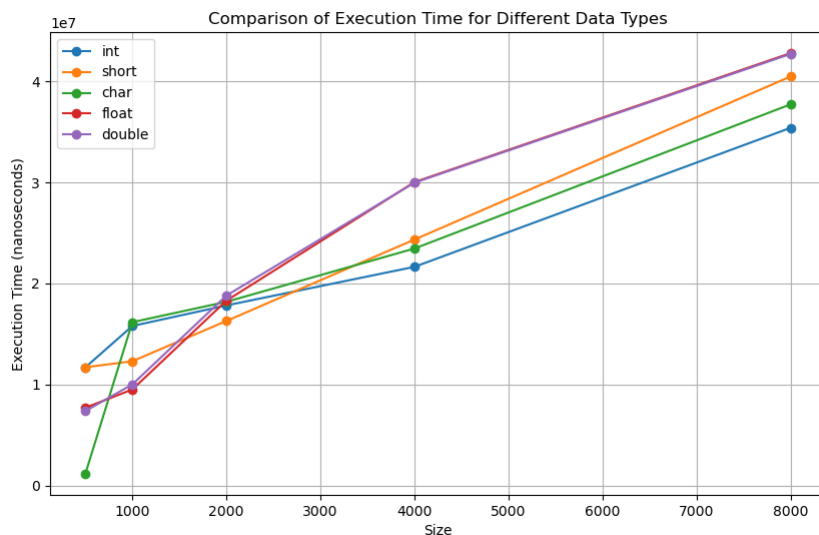
对于图片表达信息的解释：我们可以看到，对于这三个整数类型的，8000 以内的数据表现都比较的正常，整体呈现出一个线性关系，根据我们对程序时间复杂度的分析，这是正确的。但是对于 float 和 double 明显出现了一个增长的平台期。这个问题我很难分析出来，所以我将 float 和 double 类型的输入细分后继续测试：

测试后发现，数据在 7000 后出现了下跌。这让我非常的疑惑，在互联网上搜索了很久但是得不到任何信息。所以我去检查了我的代码。发现应该是文件读入时，数组大小分配不够，导致没有完全得到正确的结果。我修改了之后重新测试了数据，就得到了正确的结果。这时的浮点数也呈现出了一种线性的关系，比整数的增加还要陡峭。这是因为浮点类型的数据运算要比整数类型的运算更加复杂，内存需求也更多，所以增长也更加陡峭。



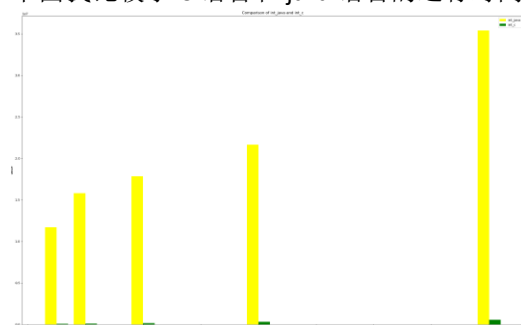
这作为我在进行 project 中的一个小曲折，我决定将他保留下来。

Java 语言：



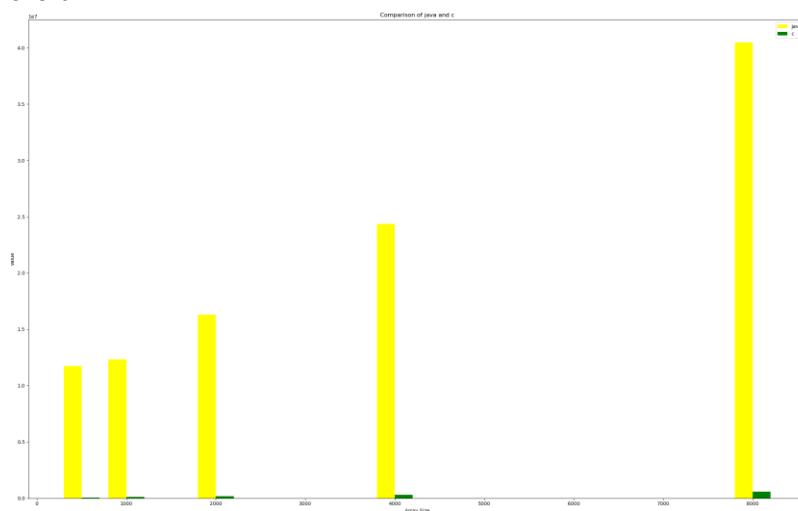
java 语言的结果比较曲折，但是整体上仍然呈现出了，正比例增长的趋势。`char` 类型的是增长最奇怪的，这可能是由于 `java` 中的字符都是没有符号的，而且多为 UTF-8 编码。这在处理时可能有些复杂。

下面我比较了 C 语言和 java 语言的运行时间差距：

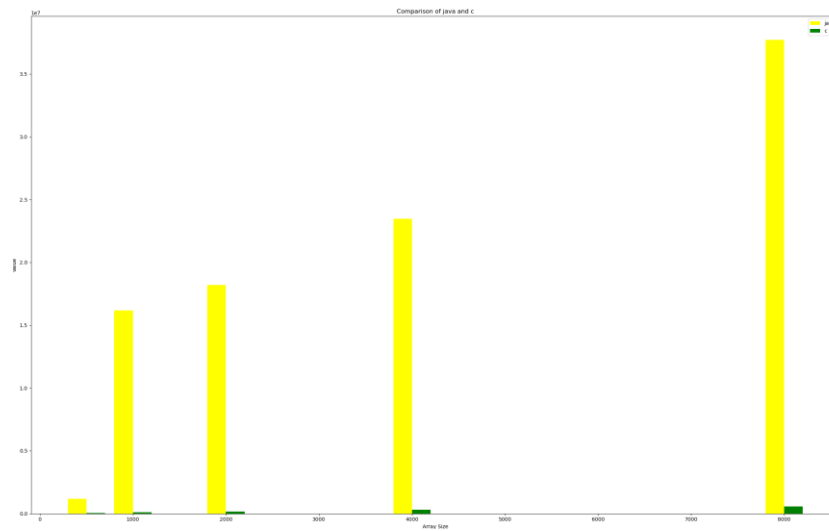


int 类型：在这个图中，黄色代表 java 绿色代表 C 语言，我们可以很直观的发现。绿色几乎看不到，这是因为 `java` 的运行时间几乎是 C 语言的数十倍。

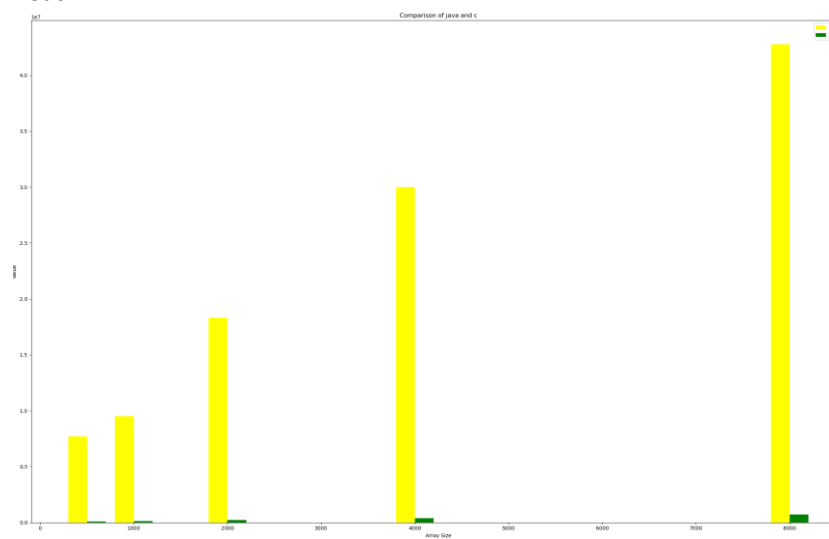
short:



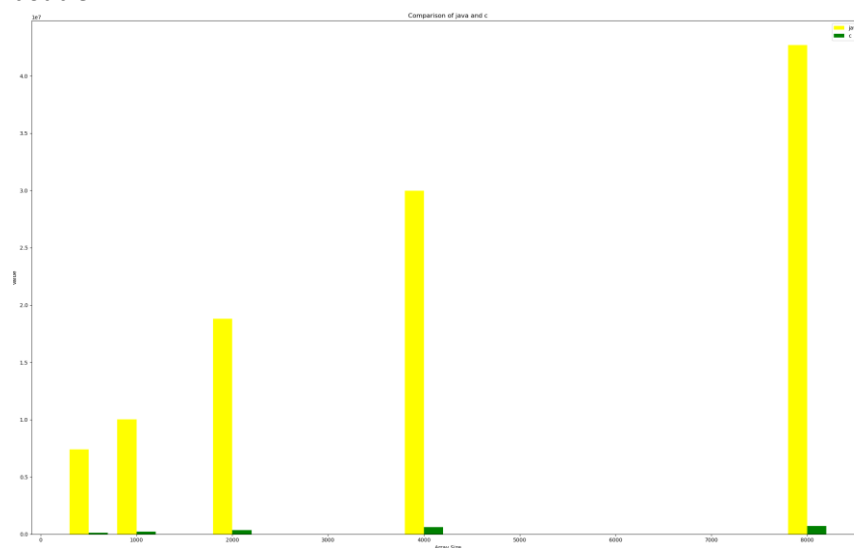
char:



float:



double:



从上面的图中可以非常形象的看出来，java 的运行速度相比 C 语言非常的慢。

下面我将从几个方面来分析一下：

首先，C 语言程序会在运行前被编译成为一个机器码文件，可以有计算机直接执行。而 java 语言不会这样做，java 语言程序会被编译成字节码，由 JVM 进行解释执行。这种额外的步骤会导致一些开销。

在内存管理方面，C 语言的内存管理需要由程序编写者来完成，申请空间，释放内存，都需要程序编写者亲历亲为。但是 Java 语言有他独特的 GC 内存管理机制，这种机制会自动进行内存的分配与垃圾的回收清理。但是这也带来了许多额外的开销，让程序的进行变得缓慢很多。

Java 语言中的对象内存分配都是堆上进行，只有方法中的局部变量才在栈上分配。而 C/C++ 的对象则有多种内存分配方式，既可能在堆上分配，也可能在栈上分配。这样也让程序速度变缓慢。

然后，java 语言的风格和 C 语言的风格也存在差异。C 语言更加接近硬件语言，而 Java 语言是更加抽象的，这在带来程序可读性优化的同时，带来了性能上的负担。

java 语言还会检查数组越界，这个也非常消耗时间，每一次对数组内容的调用都会引发一次检查。

读写速度：

下面我想探究一下 C 语言与 java 语言的读写速度差距：

去除掉读写部分

```
● lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2$ java read
Enter filename: test
程序运行时间：2399858纳秒
● lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2$ gcc read.c
● lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2$ ./a.out
test
程序运行时间：23209.0000000000 纳秒
```

从实验结果来看，java 的 i/o 的速度要比 C 语言慢很多很多。

原因如下：

C 语言有抽象程度更低的读写语句，如 `fgetc`，`fopen`。开销更低。C 的文件 I/O 函数通常不使用缓冲，或者只使用操作系统提供的简单缓冲机制。Java 的文件 I/O 操作默认使用缓冲（如 `BufferedReader` 和 `BufferedWriter`），这可以提高文件读写的效率，但如果关闭和刷新缓冲区的操作不够及时，可能会造成延迟。C 不支持异常处理机制，文件操作出错通常通过返回值来表示，这减少了运行时的额外开销。Java 使用异常处理机制，文件操作出错会抛出异常，需要捕获和处理，这增加了额外的开销。

正是因为这些原因，所以在与 java 相关的课程上，才需要引入快读快写，来提高 java 的读取和写的速度。

内存管理的分析：

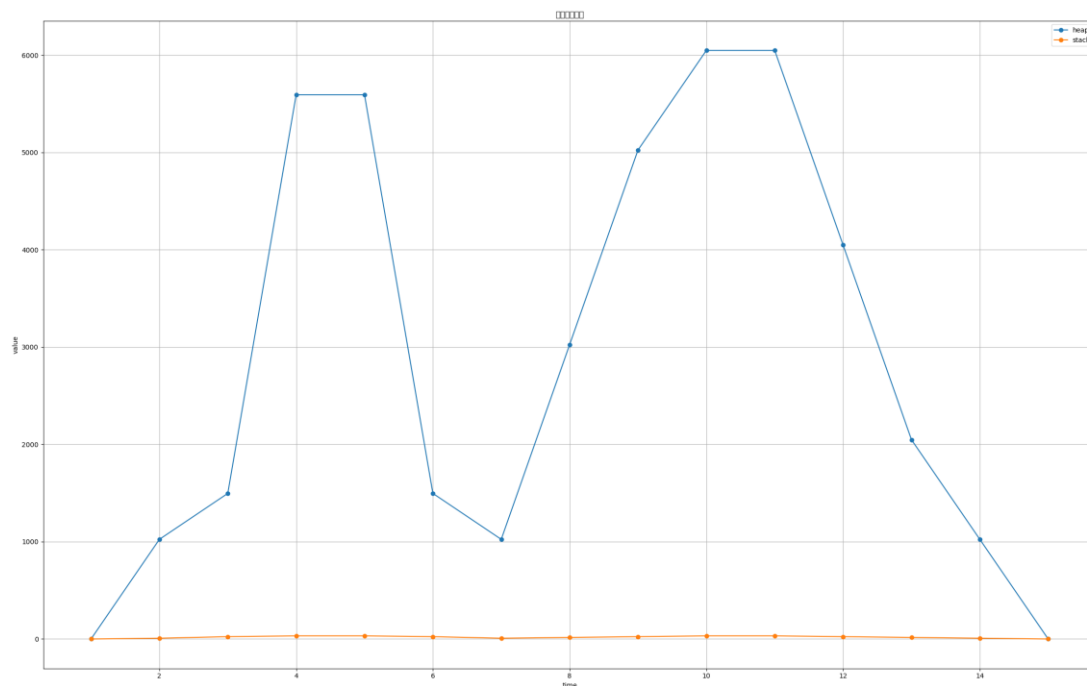
我连续对 java 程序运行两次，并分别使用 `jstat` 查看内存的情况：

● lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2\$ jstat -gc 4454												
S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC		
CCSU	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT					
2560.0		3584.0	2112.0		0.0	8704.0	5192.3	111104.0	89309.1	94976.0	92918.6	10880.0
9966.4	4057	6.187	158	9.600	-	-	15.788					
● lvzichong@lvzichong-MACHD-WXX9:~/桌面/C and Cpp/project_2\$ jstat -gc 4454												
S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC		
CCSU	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT					
1024.0		512.0	0.0	384.0		4096.0	835.7	110080.0	79925.5	95040.0	92926.7	10880.0
9966.4	4113	6.280	161	9.812	-	-	16.092					

两次程序运行前后内存出现了一些变化，我让 `deeepseek` 帮我分析了一下结果：

他向我分析了这个变化，并向我解释了 GC 机制，jvav 的堆内存分为新生代，老年代，永久代。当某一部分快要满时，就会自动触发一次 GC 垃圾回收。利用计数法或可达性分析，来判断要将什么内容回收删除。此外，GC 机制进行垃圾回收时，会将程序暂停，这也会让程序变慢，并在一定程度上引起程序运行时间的波动。

我对 C 语言程序的分析使用了 `valgrind` 的 `massif` 生成了一个程序的内存报告。（多达 150 行，不在此处展示）。很明显我不能快速高效的阅读这个报告，于是我仍然把他复制给了 `deeepseek`，责令他立刻马上仔细的给我逐行分析。通过 13 个时间点的内存快照，我做出了一个内存与时间的折线图。



通过这个折线图，我可以直观的看出来，C 语言程序在开始和结束时，对于内存空间的占用都是 0。所以就不需要 GC 垃圾回收机制在程序运行过程中的介入与检查，这让程序的运行变得更加的快速。在程序运行的过程中，内存始终是动态的增加或减少的。而 java 中，对于堆内存的使用是一直增加的。会一直增加到 GC 检查的条件，然后再进行内存释放，这让编程变得简单，但同时也让他的运行速度变得更慢了。在这个图中，栈的使用很少，因为我并没有在程序中使用函数的调用等需要大量栈的操作。所以这个实验结果也是符合理论分析的。

以上内容就是我本次项目的全部内容。在仔细分析的过程中，我学习到了很多我本来完全不知道的知识。对于内存的理解也更加的深入了。