多线程协作写文件

作者：宋超超

0. 题目解读

0.1 原题

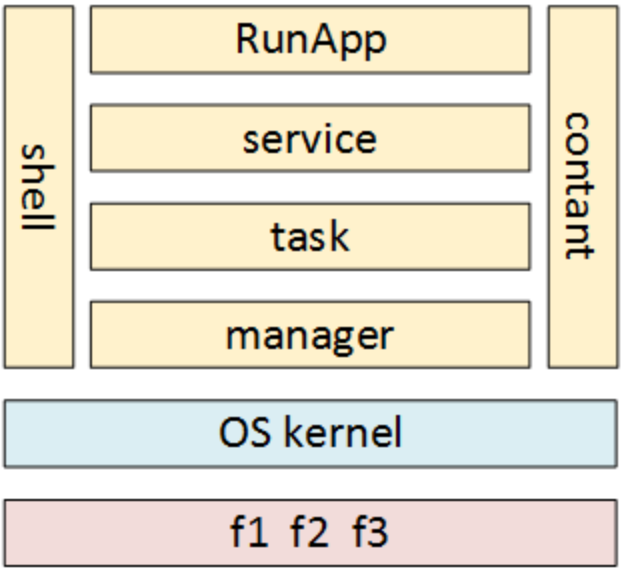
构造5个线程t1到t5,t1可以往任意文件里面写入1，t2可以写入2，依次类推。现在有3个文件f1到f3，5个线程需要合作往这三个文件内写入内容。每个文件的要求如下：
f1:123451234512345...
f2:543215432154321...
f3:134251342513425...
请设计一个让5个线程合作的好办法，让文件写的越快越好。

0.2 解读

题目中主要的硬性要求是线程必须按照一定的顺序写字符到文件（**同步**）。在实现同步的过程中，也需要通过锁、信号量等机制保证**互斥**。为了提高文件写入效率，最好不要让线程阻塞。
另外，由于磁盘的读写机制（最小读写单位为页）和硬件限制（磁盘速度相对慢），造成磁盘的读写速率远低于内存。故为了提高写文件的效率，可以参考LSM树的缓存+刷盘的方案进行IO优化。

1. 代码说明

为了提高代码的可复用性和拓展性，我将代码分成了几个层次：主函数RunApp、常量constant、服务service、任务task、文件管理manager、脚本shell。这几个层次的包或者文件均在代码包中均在。多线程协同（同步互斥）对比实验部分主要在文件管理部分实现，IO优化对比实验主要在任务部分实现。



1.1 主函数RunApp

主函数可以输入2个参数，第一个参数是方案类名称，比如WriterPlan1；第二个参数是文件路径，需要先创建3个空文件，即f1-f3。主函数会根据方案类名称来创建对应的实例进行实验，且写入到指定文件路径的3个文件中。以下是一个实验运行举例：

```
java -jar /path/to/jar/writer-1.0-SNAPSHOT.jar WriterPlan1 /path/to/files/
```

注：程序包的jar文件夹内已经打包了1个jar包，可以直接使用以上方法运行。需要注意的是，/path/to/files/文件夹内需要先创建空文件f1、f2、f3。

1.2 常量constant

常量在类Constant中，包含了运行所需的一些配置。

参数名称	参数作用	默认值
FILE_BASE_PATH	文件所在文件夹	/files/
PLAN	实验方案	WriterPlan1
FILE1_NAME	文件1名称	f1
FILE2_NAME	文件2名称	f2
FILE3_NAME	文件3名称	f3
FILE1_SEQ	文件1的字符顺序	{"1", "2", "3", "4", "5"}
FILE2_SEQ	文件2的字符顺序	{"5", "4", "3", "2", "1"}
FILE3_SEQ	文件3的字符顺序	{"1", "3", "4", "2", "5"}
THREAD_NUM	线程个数	5
FILE_NUM	文件个数	3
PAGE_SIZE	物理页的字节大小	4096
START_AFTER	创建实例后开始运行的时间（毫秒）	1000
END_AFTER	创建实例后结束运行的时间（毫秒）	10 * 1000

1.3 服务service

服务中包括了所有的方案WriterPlan1-WriterPlan9，总共9个方案。9个方案可以由主函数调用来进行实验。每个方案都是不同的任务类和文件管理类的排列组合。WritePlan1、WriterPlan2对比，观察任务

WriterFlushTask和任务WriterCheckFlushTask的写数据差异。WrtiePlan1、WrtiePlan4、WrtiePlan5、WrtiePlan6对比，观察不同的文件管理方案的写数据差异。WrtiePlan1、WrtiePlan3对比，观察加入一级缓存队列（实际上是StringBuilder）后是否会有性能提升。WriterPlan7和WriterPlan8是为了测试是否效果较好的ReentrantLockFile0、SemaphoreFile0和WriterCheckFlushTask结合可以得到更好的效果。WriterPlan9是一个无锁化的文件写入方案，可以达到最快的速度。

（1）WriterPlan1

基于Synchronized的写文件方案SyncFile0（具体见下方，下同），并且采用任务WriterFlushTask（具体见下方，下同）。

（2）WriterPlan2

基于Synchronized的写数据方案SyncFile0，并且采用任务WriterCheckFlushTask。

（3）WriterPlan3

基于Synchronized和队列的写数据方案QueuedSyncFile0Adapter，并且采用任务WriterFlushTask。

（4）WriterPlan4

基于ReentrantLock和tryLock的写数据方案ReentrantLockTryFile0，并且采用任务WriterFlushTask。

（5）WriterPlan5

基于ReentrantLock和Lock的写数据方案ReentrantLockFile0，并且采用任务WriterFlushTask。

（6）WriterPlan6

基于Semaphore的写数据方案SemaphoreFile0，并且采用任务WriterFlushTask。

（7）WriterPlan7

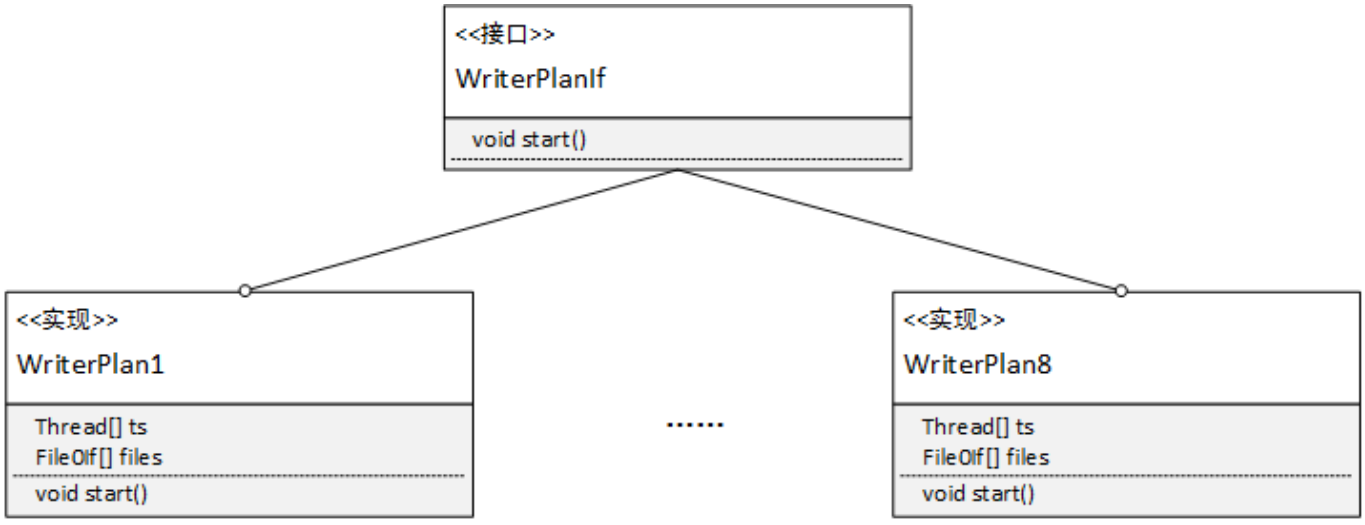
基于Semaphore的写数据方案SemaphoreFile0，并且采用任务WriterCheckFlushTask。

（8）WriterPlan8

基于ReentrantLock和tryLock的写数据方案ReentrantLockFile0，并且采用任务WriterCheckFlushTask。

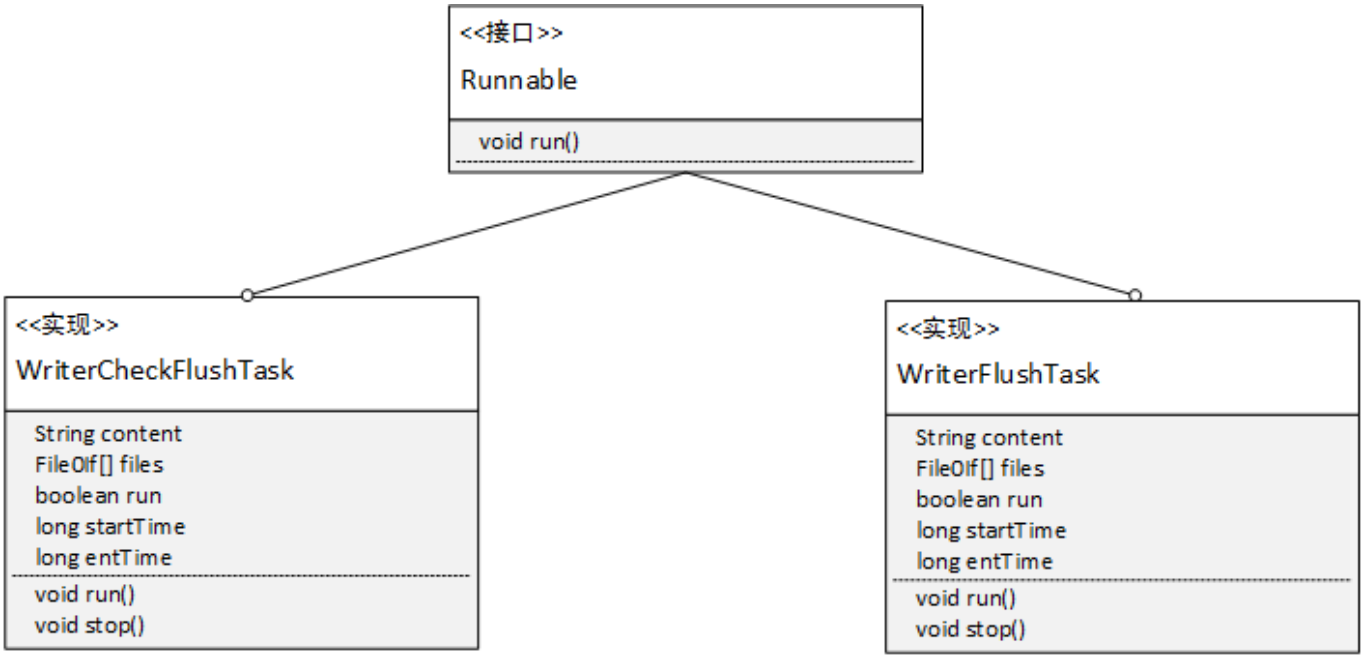
（9）WriterPlan9

基于无锁化的写数据方案NonLockFile0，并且采用任务WriterCheckFlushTask。



1.4 任务task

任务包括WriterCheckFlushTask和WriterFlushTask。WriterCheckFlushTask任务写入到缓存后不会马上刷盘flush，而是会检查是否到达阈值，达到阈值（测试采用的是1个物理页的大小）才会flush。WriterFlushTask任务写入缓存后，每次都会刷盘flush。



1.5 文件管理manager

文件管理方案的关键在于同步互斥，也就是多线程协作核心。**文件管理类通过一个指针和一个序列数组来确定下一个字符，也就保证了字符按照要求的顺序写入，即同步。这里的同步不会阻塞，而是线程判断不应该由自己写入时，直接跳过此部分代码。**

文件管理方案包括：

（1）SyncFile0类

在写方法上加synchronized进行互斥操作。synchronized会导致不同线程调用该方法时阻塞。

（2）QueuedSyncFile0Adapter类

SyncFile0的基础之上加入了一级缓存队列，也就是一个StringBuilder，在达到一定阈值后再写缓存write和刷盘flush。

（3）ReentrantLockTryFile0类

在写方法内通过ReentrantLock来进行互斥操作，并且使用tryLock函数获取锁，不会阻塞。

（4）ReentrantLockFile0类

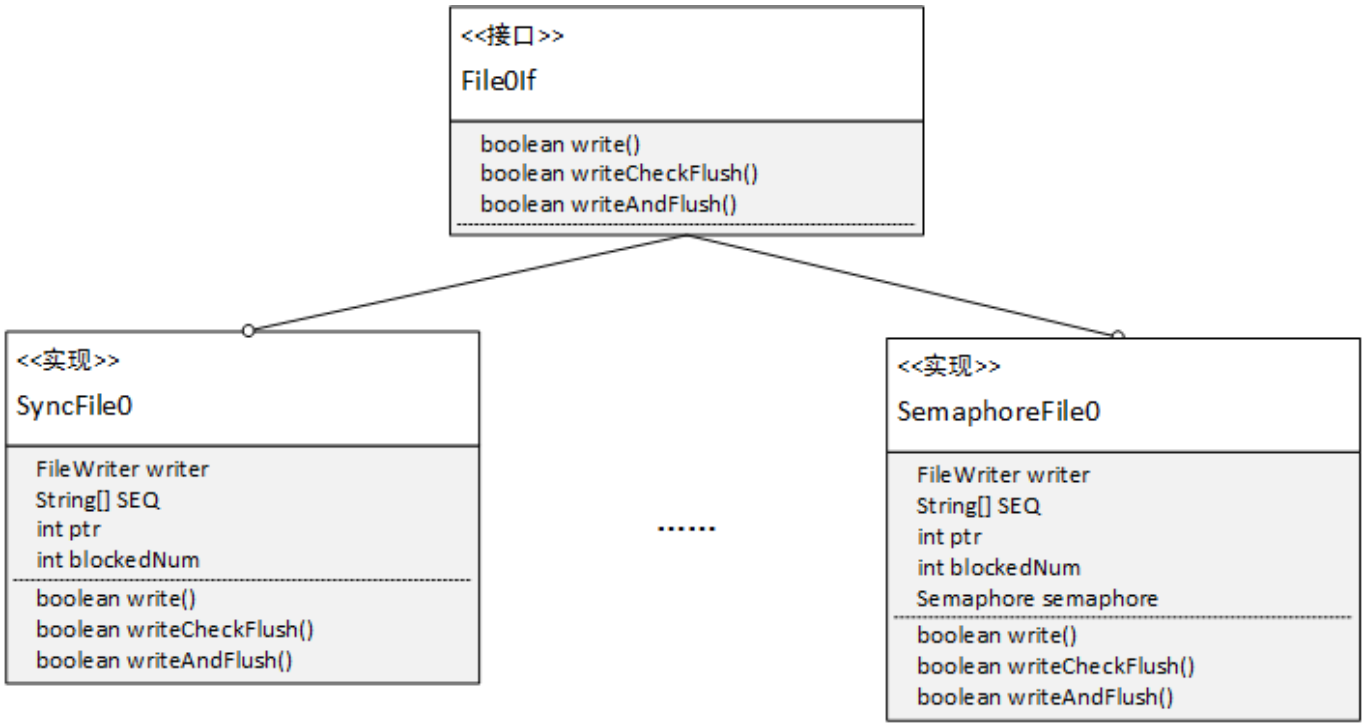
在写方法内通过ReentrantLock来进行互斥操作，并且使用Lock函数获取锁，会阻塞。

（5）SemaphoreFile0类

在写方法内通过信号量Semaphore来进行互斥操作，并且使用tryLock函数获取锁，不会阻塞。

（6）NonLockFile0类

基于无锁化的写数据方案。实现方法是通过通过一个指针和一个序列数组来确定下一个字符，也就保证了字符按照要求的顺序写入。需要注意的是代码的指针需要在写入数据后再加1，不能先加1再写入数据。



1.6 脚本

写入结果文件夹内（files/）包含多个Shell，用于实现快速文件清空、字符计算等功能。

（1）clear.sh

清除当前目录下的f1–f3的内容。

（2）count–char.sh

统计文件的字符个数，需要输入文件地址。

（3）preview.sh

预览文件的前几个字符，以核对是否按照正确排序写入。

1.7 其他说明

（1）如何保证线程同时在某一段时间内工作?

每个任务都会有开始时间和结束时间。创建5个线程前获取当前时间T0，并设置这5个线程的开始时间都是从当前时间T0后的1秒，结束时间都是从当前时间T0后的10秒，即工作9秒钟。线程举例：

```
    public void run () {
        // 等待开始
        while (System.currentTimeMillis() < startTime) {
        }
        while (System.currentTimeMillis() < endTime && run) {
            // 写文件
        }
    }
```

（2）为何要为方案类和文件管理类设计接口（WriterPlanIf和File0If）？

1. WriterPlanIf实现主函数RunApp调用的时候，不需要更改太多内容，只需要在创建的时候对应相应的类即可；
2. 写入文件的任务（WriterCheckFlushTask和WriterFlushTask）可以不关注哪种文件管理类，只需要调用接口即可。

2. 实验结果

2.1 结果统计

以下是给出的几种方案结果（5个线程运行9秒），具体的方案说明见上方：

实验名称	f1字符个数	f2字符个数	f3字符个数	平均数	字符/(秒*线程)
WriterPlan1	658618	658303	669616	662179	44145
WriterPlan2	4019157	4010963	4031448	4020523	268035
WriterPlan3	749928	758375	765438	757914	50528
WriterPlan4	1899822	1921263	1999504	1940196	129346
WriterPlan5	660635	776606	648135	695125	46342
WriterPlan6	2145540	2168024	2211616	2175060	145004
WriterPlan7	7774208	8474624	8720384	8323072	554871
WriterPlan8	5424428	6891154	6813311	6376298	425086
WriterPlan9	22605317	21052231	23109502	22255683	1483712

注：由于不是同时运行，受到其他干扰因素影响较多，有出现波动。实验结果取具有较代表性的实验，能够较为真实地体现实验结果。

2.2 结果说明

表格中的红色表示效果较好。

2.2.1 IO方案对比

实验名称	f1字符个数	f2字符个数	f3字符个数	平均数	字符/(秒*线程)
WriterPlan1	658618	658303	669616	662179	44145
WriterPlan2	4019157	4010963	4031448	4020523	268035

WritePlan1（WriterFlushTask）、WriterPlan2（WriterCheckFlushTask）对比，可以观察任务WriterFlushTask和任务WriterCheckFlushTask的写数据差异：WriterPlan2是WriterPlan1的7倍速度。也就是说**一边写缓存一边刷盘的方案明显逊色于写缓存到达阈值后再刷盘的方案。**

原因：我们知道页是最小的刷新单位，对页的操作，首先要读取页，然后修改页，再刷入磁盘。如果频繁的刷盘会造成大量的无效工作。这也是为什么许多数据库使用B+树和LSM树的原因：B+树的节点保存1页数据，正好不浪费；LSM树会先将数据缓存，然后一次性刷盘。

2.2.2 多线程协同方案对比

实验名称	f1字符个数	f2字符个数	f3字符个数	平均数	字符/(秒*线程)
WriterPlan1	658618	658303	669616	662179	44145
WriterPlan4	1899822	1921263	1999504	1940196	129346
WriterPlan5	660635	776606	648135	695125	46342
WriterPlan6	2145540	2168024	2211616	2175060	145004

（1）4种方案对比

WrtiePlan1、WrtiePlan4、WrtiePlan5、WrtiePlan6对比，观察不同的文件管理方案的写数据差异。并且控制任务变量都为WriterFlushTask。从结果来看，写入速率：

synchronized ~ ReentrantLockFile0 < ReentrantLockTryFile0 ~ SemaphoreFile0

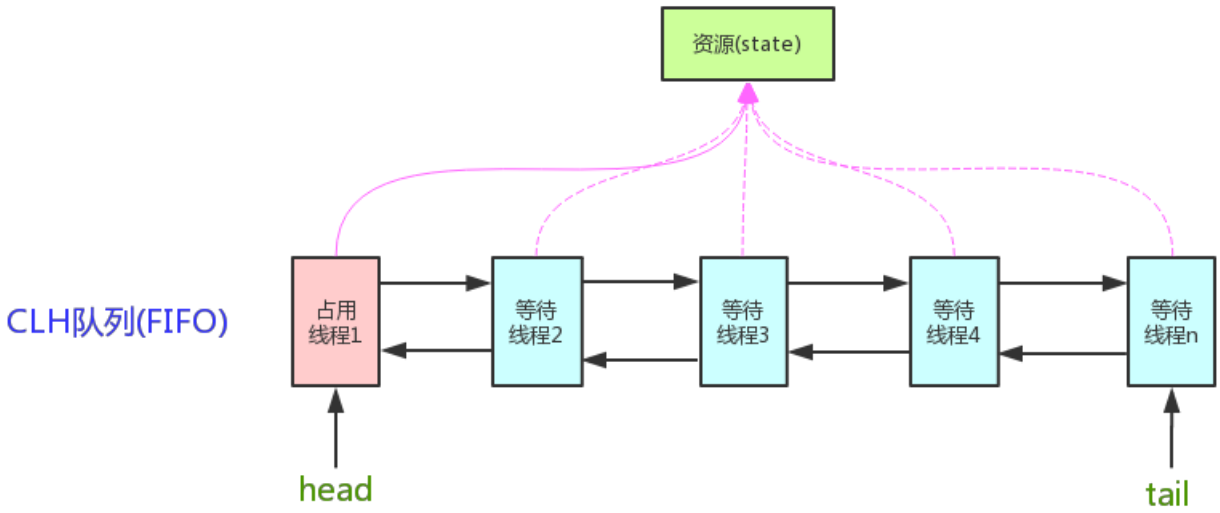
WriterPlan1 WriterPlan5 WriterPlan4 WriterPlan6

原因：ReentrantLockTryFile0和SemaphoreFile0都采用了try函数（tryLock和tryAcquire），也就是不会阻塞进程；而synchronized和ReentrantLockFile0如果无法获取锁，则会阻塞进程，造成性能下降。

（2）信号量和可重入锁对比

除此之外，信号量方案SemaphoreFile0比可重入锁ReentranLockTryFile0方案略好（后边的最优IO方案+最优多线程协同方案中也有体现）。

原因：Semaphore和ReentrantLock都是基于抽象队列同步器AQS实现的。线程共享资源state，申请使用资源时，如果被其他线程占用则需要在等待队列中等待。当然，ReentrantLockTryFile0和SemaphoreFile0都采用了try方法，不会进入到队列中。



Semaphore比ReentrantLock更加轻量：Semaphore只需要维护一个资源的state，不需要考虑重入等问题，相对较为轻量。而ReentrantLock需要考虑重入问题，较重。

2.2.3 最优IO方案+最优多线程协同方案

实验名称	f1字符个数	f2字符个数	f3字符个数	平均数	字符/(秒*线程)
WriterPlan2	4019157	4010963	4031448	4020523	268035
WriterPlan4	1899822	1921263	1999504	1940196	129346
WriterPlan6	2145540	2168024	2211616	2175060	145004
WriterPlan7	7774208	8474624	8720384	8323072	554871
WriterPlan8	5424428	6891154	6813311	6376298	425086

WriterPlan7（WriteCheckFlushTask+SemaphoreFile0）和WriterPlan8（WriteCheckFlushTask+ReentrantLockTryFile0）使用“写缓存到达阈值后再刷盘的方案”+“非阻塞方案”，效果较好。验证了**最优IO方案+最优多线程协同方案的组合可以得到更好写文件速度**：

（1）比“写缓存到达阈值后再刷盘的方案”+“synchronized方案”（WriterPlan2）将文件写入速度提高了较多。**原因**：tryLock非阻塞，比synchronized这种阻塞方式快。

（2）WriterPlan8比“一边写缓存一边刷盘”+“ReentrantLockTryFile0”方案（WriterPlan4）快，WriterPlan7比“一边写缓存一边刷盘”+“SemaphoreFile0”方案（WriterPlan4）快。**原因**：页是最小的刷新单位，对页的操作，首先要读取页，然后修改页，再刷入磁盘。如果频繁的刷盘会造成大量的无效工作。

2.2.4 最优IO方案+无锁化方案

实验名称	f1字符个数	f2字符个数	f3字符个数	平均数	字符/(秒*线程)
WriterPlan1	658618	658303	669616	662179	44145
WriterPlan2	4019157	4010963	4031448	4020523	268035
WriterPlan3	749928	758375	765438	757914	50528
WriterPlan4	1899822	1921263	1999504	1940196	129346
WriterPlan5	660635	776606	648135	695125	46342
WriterPlan6	2145540	2168024	2211616	2175060	145004
WriterPlan7	7774208	8474624	8720384	8323072	554871
WriterPlan8	5424428	6891154	6813311	6376298	425086
WriterPlan9	22605317	21052231	23109502	22255683	1483712

WriterPlan9采用无锁化写文件方案，并且通过文件管理类通过一个指针和一个序列数组来确定下一个字符，也就保证了字符按照要求的顺序写入。

原因：无锁化可以完全杜绝锁的竞争造成的资源损失。