

**2019年春季学期  
计算机学院《软件构造》课程**

**Lab 5实验报告**

|  |  |
| --- | --- |
| 姓名 | 郭子阳 |
| 学号 | 1170300520 |
| 班号 | 1703005 |
| 电子邮件 | guoziyang0033gmail.com |
| 手机号码 | 18800420598 |

**目录**

[1 实验目标概述 1](#_Toc9764347)

[2 实验环境配置 1](#_Toc9764348)

[3 实验过程 2](#_Toc9764349)

[3.1 Static Program Analysis 2](#_Toc9764350)

[3.1.1 人工代码走查（walkthrough） 2](#_Toc9764351)

[3.1.2 使用CheckStyle和SpotBugs进行静态代码分析 2](#_Toc9764352)

[3.2 Java I/O Optimization 3](#_Toc9764353)

[3.2.1 多种I/O实现方式 3](#_Toc9764354)

[3.2.2 多种I/O实现方式的效率对比分析 4](#_Toc9764355)

[3.3 Java Memory Management and Garbage Collection (GC) 6](#_Toc9764356)

[3.3.1 使用-verbose:gc参数 6](#_Toc9764357)

[3.3.2 用jstat命令行工具的-gc和-gcutil参数 7](#_Toc9764358)

[3.3.3 使用jmap -heap命令行工具 8](#_Toc9764359)

[3.3.4 使用jmap -histo命令行工具 8](#_Toc9764360)

[3.3.5 使用jmap -clstats命令行工具 8](#_Toc9764361)

[3.3.6 使用JMC/JFR、jconsole或VisualVM工具 9](#_Toc9764362)

[3.3.7 分析垃圾回收过程 9](#_Toc9764363)

[3.3.8 配置JVM参数并发现优化的参数配置 10](#_Toc9764364)

[3.4 Dynamic Program Profiling 10](#_Toc9764365)

[3.4.1 使用JMC或VisualVM进行CPU Profiling 10](#_Toc9764366)

[3.4.2 使用VisualVM进行Memory profiling 11](#_Toc9764367)

[3.5 Memory Dump Analysis and Performance Optimization 11](#_Toc9764368)

[3.5.1 内存导出 11](#_Toc9764369)

[3.5.2 使用MAT分析内存导出文件 11](#_Toc9764370)

[3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 12](#_Toc9764371)

[3.5.4 在MAT内使用OQL查询内存导出 12](#_Toc9764372)

[3.5.5 观察jstack/jcmd导出程序运行时的调用栈 13](#_Toc9764373)

[3.5.6 使用设计模式进行代码性能优化 14](#_Toc9764374)

[3.6 Git仓库结构 14](#_Toc9764375)

[4 实验进度记录 15](#_Toc9764376)

[5 实验过程中遇到的困难与解决途径 15](#_Toc9764377)

[6 实验过程中收获的经验、教训、感想 15](#_Toc9764378)

[6.1 实验过程中收获的经验和教训 15](#_Toc9764379)

[6.2 针对以下方面的感受 15](#_Toc9764380)

# 实验目标概述

本次实验通过对Lab4的代码进行静态和动态分析，发现代码中存在的不符合代码规范的地方、具有潜在bug 的地方、性能存在缺陷的地方（执行时间热点、内存消耗大的语句、函数、类），进而使用第 4、7、8章所学的知识对这些问题加以改进，掌握代码持续优化的方法，让代码既“看起来很美”，又“运行起来很美”。

具体训练的技术包括：

* 静态代码分析（CheckStyle和SpotBugs）
* 动态代码分析（Java命令行工具jstat、jmap、jcmd、VisualVM、JMC、JConsole等）
* JVM 内存管理与垃圾回收（GC）的优化配置
* 运行时内存导出(memory dump)及其分析（Java 命令行工具jhat、MAT）
* 运行时调用栈及其分析（Java命令行工具jstack）；
* 高性能 I/O
* 基于设计模式的代码调优
* 代码重构

# 实验环境配置

实验环境：

系统：macOS Mojave 10.14.3，Oracle JDK 1.8.0\_211

ide：Eclipse IDE for Eclipse Committers 2018-12 (4.10.0), IntelliJ IDEA ULTIMATE 2019.1

编辑器：Visual Studio Code 1.31.1

构建工具：Gradle，Travis CI，Drone CI

版本管理：git

代码托管：Github, Gogs

配置过程：

本次实验基于Lab4，实验环境无需配置，安装少量插件即可

GitHub Lab5仓库的URL地址

https://github.com/ComputerScienceHIT/Lab5-1170300520

# 实验过程

## Static Program Analysis

### 人工代码走查（walkthrough）

人工代码走查遵照Alibaba Java开发规范进行。

1. 抽象类命名使用Abstract或Base开头。

举例：Position -> AbstractPosition

1. 线程池不允许使用 Executors 去创建,而是通过 ThreadPoolExecutor 的方式。

举例：建立GuiThreadPool线程池类管理Gui绘制线程

1. 常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。

举例：defaultMaxSpeed -> DEFAULT\_MAX\_SPEED

1. 在一个 switch块内，每个 case要么通过 break/return来终止，要么注释说明程序将继续执行到哪一个 case为止；在一个 switch块内，都必须包含一个 default语句并且放在最后，即使它什么代码也没有。

举例：switch嵌套时丢失break

1. 不允许出现任何魔法值（即未经定义的常量）直接出现在代码中。

举例：private static Pattern periodPattern = Pattern.compile("(?<=Period ::= )((Hour)|(Day)|(Week)|(Month))$");

1. 在 if/else/for/while/do语句中必须使用大括号，即使只有一行代码

举例：

if (!inBlock) {

newline = new StringBuilder();

}

1. Map/Set的 key为自定义对象时，必须重写 hashCode和 equals。

举例：重写TimeLog等类等hashCode和equals方法。

### 使用CheckStyle和SpotBugs进行静态代码分析

CheckStyle使用Google规范：

1. 每次打开新的块或块状构造时，缩进增加两个空格。

举例：将四空格缩进改为两个空格

1. 包名称都是小写，连续的单词连接在一起（无下划线）。

举例：APIs -> apis

1. 类名写在UpperCamelCase（驼峰命名）中。

CircularOrbitAPIs -> CircularOrbitApis

1. 源文件名由其包含的顶级类（其中包含正好一个 ）的区分大小写的名称。

举例：为Coordinate与GuiThreadPool建立新的文件

1. 不使用静态或其他方式的通配符导入 。

举例：不使用类似于import java.util.\*;

1. 在每个块中，导入的名称以ASCII排序顺序显示。

将import语句按照ascii顺序排列

## Java I/O Optimization

### 多种I/O实现方式

使用了

1. Reader/Writer，带缓冲区的字符流
2. NIO的FileChannel
3. NIO的MappedByteBuffer，使用内存映射的NIO
4. Reader/Writer，使用循环reader.readLine()直接逐行进行读取，对读进来的每一行进行解析建立系统即可。写入时使用BufferedWriter，使用writer.write()方法对文件写入，然而该方法只是将内容写入缓冲区中，需要调用writer.flush()或者在关闭流writer.close()时才会将缓冲区的内容写入到文件中。（最常使用）
5. FileChannel，为NIO中用于代替标准IO读取文件的类。fileChannel.read(byteBuffer)，一次读取ByteBuffer大小个字节到ByteBuffer中。由于需要按行读取后进行解析，需要对ByteBuffer逐字节进行判断到换行符，再一次返回一个String。故写了一个readLine的接口对操作进行封装，一旦调用，就会循环读取到下一个换行符并返回当前行的string。写入时调用fileChannel.write()即可。
6. MappedByteBuffer，是NIO引入的文件内存映射方案，将待读取或待写入的文件直接映射到内存中，当作一个ByteBuffer，并直接读取ByteBuffer中的字节。读取和写入需要调用mappedByteBuffer.read()和mappedByteBuffer.write()方法。由于这种方法需要映射磁盘文件到内存，故需要指定映射文件的大小，读文件时速度很快，但是在动态写文件时由于无法确认文件大小，需要频繁地映射，效率极低。

### 多种I/O实现方式的效率对比分析

在读文件前后使用Calendar.getInstance().getTimeInMillis()获取当前的时间戳（ms），相减即获得所需的毫秒。

复制文件测试（读取的同时写入，31.6 MB）

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | Reader/Writer | FileChannel | MappedByteBuffer |
| 耗时（ms） | 840 | 82 | 666 |

可以看出，由于写文件的效率低下，MappedByteBuffer慢于FileChannel。

读文件测试（有解析过程，速度可能较慢，19.8 MB）

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | Reader/Writer | FileChannel | MappedByteBuffer |
| 耗时（s） | 5.956 | 0.447 | 0.285 |

写文件测试（写入30000行“All work and no play makes Jack a dull boy”（致敬《闪灵》），动态写入）

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | Reader/Writer | FileChannel | MappedByteBuffer |
| 耗时（s） | 0.061 | 0.408 | 75.53 |

在动态写入文件时，由于MappedByteBuffer不知道文件的大小，频繁地开关流导致效率低下。

写文件测试（写入30000行“All work and no play makes Jack a dull boy” （致敬《闪灵》），静态写入，指定大小）

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | Reader/Writer | FileChannel | MappedByteBuffer |
| 耗时（ms） | 48 | 305 | 10 |

在指定文件大小后，MappedByteBuffer效率极高。

图形对比

## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数

启动后构建PersonalAppEcosystem，选用内存映射NIO的方式进行读取配置文件构建系统，并运行了可视化、计算熵、日志查询和输出到文件。

日志中的GC若无修饰则为Minor GC。

期间共发生11次Minor GC和一次Full GC。大多数Minor GC发生在读取配置文件构建系统时，一次Full GC发生在日志查询时（读入大量日志）。

Minor GC耗时从0.0085758 secs到0.2170912 secs不等，而且随着时间推后，耗时有所增加，一次Full GC耗时0.2134761 secs。

Minor GC即新生代垃圾回收，Full GC针对整个堆进行垃圾回收。根据Full GC的日志信息：[Full GC (Ergonomics) [PSYoungGen: 16601K->0K(437248K)] [ParOldGen: 140756K->139602K(217088K)] 157358K->139602K(654336K), [Metaspace: 17889K->17889K(1064960K)], 0.2134761 secs] [Times: user=0.59 sys=0.04, real=0.21 secs]看来，在进行Full GC时会同时使用PSYoungGen、ParOldGen和Metaspace（这个本次看起来没起作用）进行垃圾回收，同时释放新生代、年老代和元空间的内容。

GC或Full GC后面的括号内容就是本次GC产生的原因，可以看出，所有的Minor GC产生的原因都是Allocation Failure（新生代中没有足够的空间），而Full GC的原因是Ergonomics（HotSpot自动选择和调优引发的FullGC）。

由日志可以看出，由于需要产生大量新的对象，导致内存的新生代区域不足，发生了频繁的Minor GC，尤其在读取文件（配置和日志）时最为明显。

### 用jstat命令行工具的-gc和-gcutil参数

jstat [-命令选项] [vmid] [间隔时间/毫秒] [查询次数]

vmid可用jps命令查询到。可将结果输出到文件便于分析。

单条信息如下：

S0 S1 E O M CCS YGC YGCT FGC FGCT GCT

0.00 91.29 12.46 0.01 97.99 90.46 1 0.008 0 0.000 0.008

S0：幸存0区当前使用比例，S1：幸存1区当前使用比例，E：伊甸园区使用比例，O：年老代使用比例，M：元数据区使用比例，CCS：压缩使用比例，YGC：新生代垃圾回收次数，YGCT：新生代垃圾回收耗时，FGC：年老代垃圾回收次数，FGCT：年老代垃圾回收消耗时间，GCT：垃圾回收消耗总时间。

新生代的大小就是伊甸园区+两个幸存区的大小。可以看出两个幸存区的占用不停的交换（0，占用，0，……），每一次交换都是由于一次Minor GC。而在一次Full GC后，两个区域的占用都归为0。

新生代垃圾回收的机制为：最初，所有的对象都在伊甸园区和“From”区（某一个幸存区），垃圾回收时会将伊甸园区的所有存活的对象复制到“To”区（另一个幸存区），而在“From”区，年龄达到一定的阈值的对象会被复制到年老区，没有达到的会被复制到“To”区。完成后“From”和“To”的关系互换。此时原“From”区被清空。

由此即可解释两个幸存区的占用不停互换的情况。

### 使用jmap -heap命令行工具

该命令用于展示垃圾回收机制以及堆的各个部分的占用情况。

貌似在MacOS上，JDK8版本下无法运行这个命令，会出现Can't attach symbolicator to the process的错误，这个错误高于9的JDK版本中被修复了。在Linux下，由于ptrace-scope机制，防止用户访问当前正在运行的进程的内存和状态，事先执行echo 0 > /proc/sys/kernel/yama/ptrace\_scope命令即可。

在jdk9及以上，jmap工具被jhsdb jmap代替，运行命令为

jhsdb jmap --heap --pid [vmpid]

可以看到，jvm使用Garbage-First(G1) GC进行垃圾收集，即垃圾优先策略。

伊甸园区总大小为84.0MB，使用率73.8%；幸存者区总大小为6.0MB，使用率100%；年老带总大小为140.0MB，使用率60.9%。

### 使用jmap -histo命令行工具

该命令用于查看被加载进内存的各个类的实例数量以及内存占用。

可以看出，占用内存最多的类为HashMap，内存中一共加载了1006289个HashMap实例，占用内存32201248字节的内存。

### 使用jmap -clstats命令行工具

该命令执行速度较慢。主要用来查看类加载器以及各个类加载器的加载情况和占用等。

共有8个类加载器，一共加载了3729个类，占用字节10261783个，其中4个已经dead，剩余4个alive。

在这8个classloader中，有一个为bootstrap classloader，用于加载核心类库，共加载了3041个类，且仍然存活（alive）。

剩下的7个classloader中，有4个为DelegatingClassLoader类，3个为ClassLoader类。DelegatingClassLoader为反射加载器，可能是由于我使用了Class.forName("applications.LogReader")来加载日志写入。所以在加载完成后，三个反射加载器都已dead。

貌似没法看到method的信息……

### 使用JMC/JFR、jconsole或VisualVM工具

使用VisualVM，JDK自带，使用命令jvisualvm即可进入。

运行时的流程时的监控情况如图。



其中，CPU的四个峰值分别为读入配置文件并建立系统、可视化、读取日志文件、输出日志文件。

可以看出，在读入配置文件并建立系统时，或许是由于堆堆大小是动态扩展的，在堆的占用提升后，堆的大小也相应地扩大了，但是装入的类的总数却没有明显的提升（有少量提升）。

由于在可视化时使用了大量的HashMap，在进行可视化展示的时候，装入的类的总数有了明显的提升，正好印证了上面jmap -histo命令行工具展示的装载入内存的实例最多的类是HashMap，同时，由于可视化涉及多线程操作，实时的线程也增加了。

### 分析垃圾回收过程

在程序运行的过程中，由于有大量类的装入，导致新生代区域内存不足，引发频繁的Minor GC。同时引发了一次Full GC，通过日志可以看出，唯一的一次Full GC是JVM自动调优引起的，可以忽略。

在新生代进行垃圾回收时，对象不停地在两个幸存区间复制，从“From”区到“To”区。通常，Minor GC发生在伊甸园区或者“From”区占用过高时。如在读入配置文件或可视化时，大量的对象装入内存，就容易导致伊甸园区和“From”区的高占用，从而引发多次Minor GC。

### 配置JVM参数并发现优化的参数配置

经过分析，主要的问题在于，由于堆空间不足引发较多次数的Minor GC，于是可以适当地调大堆内存来增加新生代区域的内存，减少Minor GC的次数。但是，增加堆内存必然会导致Minor GC的时间增加，故不能无限制地增加内存。

调优后的参数为

-Xms3G -Xmx3G -Xss256k -XX:+UseParallelOldGC -XX:NewSize=1G -XX:MaxNewSize=1G

-Xms3G -Xmx3G将最小与最大的堆内存固定为3G。

-Xss256k将每个线程的堆栈大小设置为256k（默认为1m）。

-XX:+UseParallelOldGC使用并行的年老代垃圾回收机制，提升Full GC的速度。

-XX:NewSize=1G -XX:MaxNewSize=1G设置新生代区域的大小为1G，提升了新生代区域的占用，用于减少Minor GC的发生次数。

使用该参数后，同样的流程，共发生Minor GC两次，Full GC零次，Minor GC的耗时分别为0.106s和0.273s。可以看出，调优有效，大大减少了Minor GC的次数，相应的，由于新生代区域内存增大，每次Minor GC的耗时增加，但还在可接受范围内。

## Dynamic Program Profiling

### 使用JMC或VisualVM进行CPU Profiling

使用VisualVM。

可能出现redefinition failed with error 62的问题，只需要在JVM参数后添加-Xverify:none即可解决。

启动后分析CPU的热点情况，执行后生成分析快照。

绝大多数的时间花在了menu()方法上，其次是readLog()方法。menu()函数可以理解，等待输出的时间都记在menu()方法上，而readLog()方法，则是主要的时间占用函数。readLog()方法用于从日志文件的上一次读取结束的位置开始，读取到文件末尾，由于日志文件较大，读取时间较长比较正常。再其次是searchLog()方法，用于对读入的日志进行筛选过滤，由于日志较多，执行时间长也比较正常。

### 使用VisualVM进行Memory profiling

占用内存最多的对象为int[]，其次分别为HashMap和Calendar。int[]较多的原因，通过查询得知，由于出现了大量的Calendar对象，Calendar类内部使用了较多的int[]，导致int[]实例较多。HashMap占用主要是由于在构建系统及可视化等操作中使用了大量的HashMap存储临时信息，导致占用过多。Calendar对象的创建主要是由于构建的系统是PersonalAppEcosystem，其中日期的存储使用的都是Calendar类。

## Memory Dump Analysis and Performance Optimization

### 内存导出

使用VisualVM的heap dump对构建完成系统后的程序导出内存，保存文件的格式为hprof。由于读取的是十万行级别的文件，建系统过程可能较慢，需要对其进行优化才可顺利建立。优化后建立系统的过程在2-10s左右即可建立系统。

### 使用MAT分析内存导出文件

MAT可以直接在eclipse的marketplace中搜索安装，安装Memory Analyzer即可。之后直接open file导出的hprof文件即可进行分析。

histogram视图显示的是内存中存储的各类型的实例数量以及所占用内存的情况。该视图显示的结果与使用VisualVM进行memory profiling的结果基本一致。占用内存最多的对象为int []，之后依次为HashMap与Calendar。HashMap在优化速度方面有着无与伦比的左右，在HashMap中寻找Key 的理论时间复杂度为O(1)，避免了大量的迭代。

dominator tree视图展示了各实例之间的引用关系，以及与root之间的引用路径。其中，Shallow Size表示对象自身占用的内存大小，不包括它引用的对象，而Retained Size表示当前对象大小+当前对象可直接或间接引用到的对象的大小总和。Retained Heap占用最大的类为AppClassLoader，为一个类加载器，占用99.56%，它间接了引用了几乎所有的构建系统所需要的类，如PersonalAppEcosystem、App、AbstractCircularOrbit等。大概是由于内存dump的时候系统刚刚建立完成，许多中间的类还没有来得及被回收，仍然处于被AppClassLoader引用的状态，导致占用过高。

top consumers视图展示了当前内存的占用热点，同上面一样，不出所料AppClassLoader是占用最多的类，原因上面已经解释过了。

leak suspects视图展示了可能的内存泄漏位置以及原因，AppClassLoader与其中的HashMap被认为可能产生了内存泄漏，需要改进。一个HashMap对象就占用了79.13%的内存……

### 发现热点/瓶颈并改进、改进前后的性能对比分析

主要改进的热点在于对上文提到的AppClassLoader与HashMap进行改进，减少其内存占用。

在内存泄漏视图下的Accumulated Objects in Dominator Tree选项中，可以看出造成大量内存占用的HashMap是属于一个HashSet的（HashSet的实现是使用HashMap），而该HashSet又是属于一个AppConfiguration的。点击该HashMap选择show objects by class -> by outgoing referneces，可以查看HashMap的内部引用情况，该HashMap的内部引用了大量的UsageLog对象，用于临时存储。

于是确定，造成内存泄漏的是构建系统时的一个临时存储配置信息的HashMap，属于一个临时的AppConfiguration对象，而该对象在构建完成系统之后便不再需要。于是，在构建完成系统后，使用appConfiguration = null将对象引用置为空，使没有任何引用指向该对象，并且手动请求一次垃圾回收System.gc()。

优化过后，再次dump memory后，可以看出，该对象已被完全释放，jvm总的内存占用由498.4MB下降到了87.3MB。

### 在MAT内使用OQL查询内存导出

Eclipse MAT可以对内存dump进行OQL语句查询，OQL的格式类似于SQL，是一种对象查询语句。

1. CircularOrbit的所有对象实例

CircularOrbit为一接口，在OpenJDK的BUG提交处（https://bugs.openjdk.java.net/browse/JDK-6543691）可找到回答：“Information about interfaces is \*not\* available in the current heap dump format”，即在当前的堆dump中是无法获取对象的接口信息的。于是只能查询第一层实现CircularOrbit接口的抽象类AbstractCircularOrbit（所有实现CircularOrbit接口的类必然继承自AbstractCircularOrbit），查询语句如下：

SELECT \* FROM instanceof circularorbit.AbstractCircularOrbit

1. 大于特定长度 n 的 String 对象

使用WHERE限定条件即可

查询语句如下：

SELECT \* FROM java.lang.String s WHERE s.value.@length > n

最后的n修改为需要查询的长度即可

1. 大于特定大小的任意类型对象实例

WHERE中的修饰改为继承自java.lang.Object即可限定为所有的类型。使用@可以查询到对象的属性，其中一条属性为@usedHeapSize，即为该对象占用大小

如下：

SELECT \* FROM instanceof java.lang.Object s WHERE s.@usedHeapSize > 1024

该语句可查询到所有大于1024字节的对象

1. PhysicalObject（及其子类）的对象实例的数量和总占用内存大小；

类似第一个要求，查询语句如下

SELECT \* FROM instanceof physicalobject.AbstractPhysicalObject

### 观察jstack/jcmd导出程序运行时的调用栈

使用jstack <pid>可输出程序运行的调用堆栈，可将结果重定向到文件中以便于分析。

输出结果中会列出全部线程的调用信息，我们基本只需要关注main线程的情况即可。

输出的堆栈信息显示的主要是某一个方法（单一线程某一时刻只运行一个方法），从下往上依次调用，最顶端即为当前方法，该方法由下层的方法调用，以此类推。

如下为等待输入时的调用栈：

java.lang.Thread.State: RUNNABLE

at java.io.FileInputStream.readBytes(Native Method)

at java.io.FileInputStream.read(FileInputStream.java:255)

at java.io.BufferedInputStream.read1(BufferedInputStream.java:284)

…

RUNNABLE表示线程正在运行，当前正在运行的方法为FileInputStream.readBytes()方法，该方法由FileInputStream.read()方法调用，而read()方法又由BufferedInputStream.read1()方法调用，等等。

### 使用设计模式进行代码性能优化

在创建配置文件的Reader和Parser时，使用了装饰器模式，可以动态地组合各种读取策略读取不同类型的配置文件。

创建GUI动态更新线程时，使用了object pool，自定义了线程池，提升了线程的使用效率。

同时，使用MappedByteBuffer来进行日志文件的读取，提升读取速度。

在构建系统时，多使用HashSet、HashMap而不是List，由于哈希函数，在这两个结构中寻找某个对象的理论时间复杂度为O(1)，效率极高。

## Git仓库结构



# 实验进度记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2019.5.18 | 14:00 – 16:30 | 人工走查 | 按期完成 |
| 2019.5.20 | 9:00 – 11:30 | 按照Google规约修改 | 按期完成 |
| 2019.5.21 | 18:30 - 20:43 | 完成配置文件写入 | 按期完成 |
| 2019.5.22 | 9:30 – 10:20 | 完成channel读取类 | 按期完成 |
| 2019.5.23 | 14:28 – 16:31 | 完成MappedByteBuffer读取 | 按期完成 |
| 2019.5.24 | 19:20 – 1:30 | 监控性能并优化 | 按期完成 |
| 2019.5.25 | 18:00 – 20:00 | 内存泄漏排查 | 按期完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 对于垃圾回收机制的不理解 | 查看课件，Google |
| NIO的Channel与MappedByteBuffer | Google |
| jmap、jstack、VisualVM等工具 | Google、stackoverflow |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

学习了静态代码分析工具和动态代码分析工具的使用，理解了JVM垃圾回收的原理，学习了多种IO的实现方式以及性能对比，学会了使用内存导出对某一时刻的内存进行分析，找出热点后优化。

## 针对以下方面的感受

“看起来很美”指的是代码符合规范，“运行起来很美”指的是程序运行速度较快，性能较优。二者没有必然联系，也没有谁比谁重要一说。程序性能较优则用户体验良好，而代码符合规范则便于后期维护升级。

Java I/O在jdk1.4之前，只有同步阻塞的IO（BIO），提供了普通的字符流和字节流，效率低下。在jdk1.4时提供了同步非阻塞的IO（NIO），并且提供了Channel、Buffer等抽象，在jdk1.7时，Java实现了异步非阻塞的IO（AIO）。

自动垃圾回收机制固然优秀，却无法由程序员掌控，程序员无法指定回收某个对象。

本次实验两周时间，代码量不大，分析内容较多。