链滴

JVM 八股文学习

作者：Gakkiyomi2019

原文链接：https://ld246.com/article/1618650582546

来源网站：链滴

JVM八股文学习

之前零零散散的看过一些jvm之类的知识，但都没有整理，导致每次想不起来都要去翻书或者google这里做一次完整的记录，可被方便的索引到。

内存分配和回收策略

Minor GC/Major GC /Full GC

● Minor GC：回收新生代（包括 Eden 和 Survivor 区域），因为 Java 对象大多都具备朝生夕灭的性，所以 Minor GC 非常频繁，一般回收速度也比较快。

● Major GC / Full GC: 回收老年代，出现了 Major GC，经常会伴随至少一次的 Minor GC，但这并绝对。Major GC 的速度一般会比 Minor GC 慢 10 倍 以上。

● 在 JVM 规范中，Major GC 和 Full GC 都没有一个正式的定义，所以有人也简单地认为 Major GC 清理老年代，而 Full GC 清理整个内存堆。

●

对象优先在eden区分配

大多数情况下，对象在新生代 Eden 区中分配。当 Eden 区没有足够空间进行分配时，虚拟机将发起次 Minor GC。

大对象直接进入老年代

大对象是指需要大量连续内存空间的 Java 对象，如byte[1024 12041024]。

一个大对象能够存入 Eden 区的概率比较小，发生分配担保的概率比较大，而分配担保需要涉及大量

复制，就会造成效率低下。

虚拟机提供了一个 -XX:PretenureSizeThreshold参数，令大于这个设置值的对象直接在老年代分配这样做的目的是避免在 Eden 区及两个 Survivor 区之间发生大量的内存复制。

长期存活的对象进入老年代

既然jvm采用了分代收集的思想，那必然就会为对象标注当前属于那个时代，所以JVM 给每个对象定了一个对象年龄计数器。如果对象在新生代发生一次 Minor GC 后仍然能存活，那么就会被移动到survivor区，并且对象年龄设为1，之后对象每熬过一次Minor GC，对象年龄加1，当年龄增长到一定程后(默认15)，就会晋升到老年代中。设置阈值：-XXMaxTenuringThreshold=15

使用 -XXMaxTenuringThreshold设置新生代的最大年龄，只要超过该参数的新生代对象都会被转移老年代中去。

动态对象年龄判定

如果当前新生代的 Survivor 中，相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄 >= 年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold 中要求的年龄。

空间分配担保

JDK 6 Update 24 之前的规则是这样的：

在发生 Minor GC 之前，虚拟机会先检查**老年代最大可用的连续空间是否大于新生代所有对象总空间** 如果这个条件成立，Minor GC 可以确保是安全的； 如果不成立，则虚拟机会查看HandlePromotioFailure值是否设置为允许担保失败， 如果是，那么会继续检查老年代最大可用的连续空间是否大于次晋升到老年代对象的平均大小， 如果大于，将尝试进行一次 Minor GC,尽管这次 Minor GC 是有险的； 如果小于，或者 HandlePromotionFailure 设置不允许冒险，那此时也要改为进行一次 Full G 。

JDK 6 Update 24 之后的规则变为：

只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小，就会进行 Minor GC，否则进行 Full GC。

通过清除老年代中废弃数据来扩大老年代空闲空间，以便给新生代作担保。HandlePromotionFailure个参数不会在影响空间分配担保策略了。

以上这个过程就是分配担保。

JVM触发Full GC 的情况

1. 我们手动调用 system.gc()方法，但这也不一定会执行Full GC，只会增加Full GC的执行频率我可以通过-XX:DisableExpliitGC来禁止调用Full GC

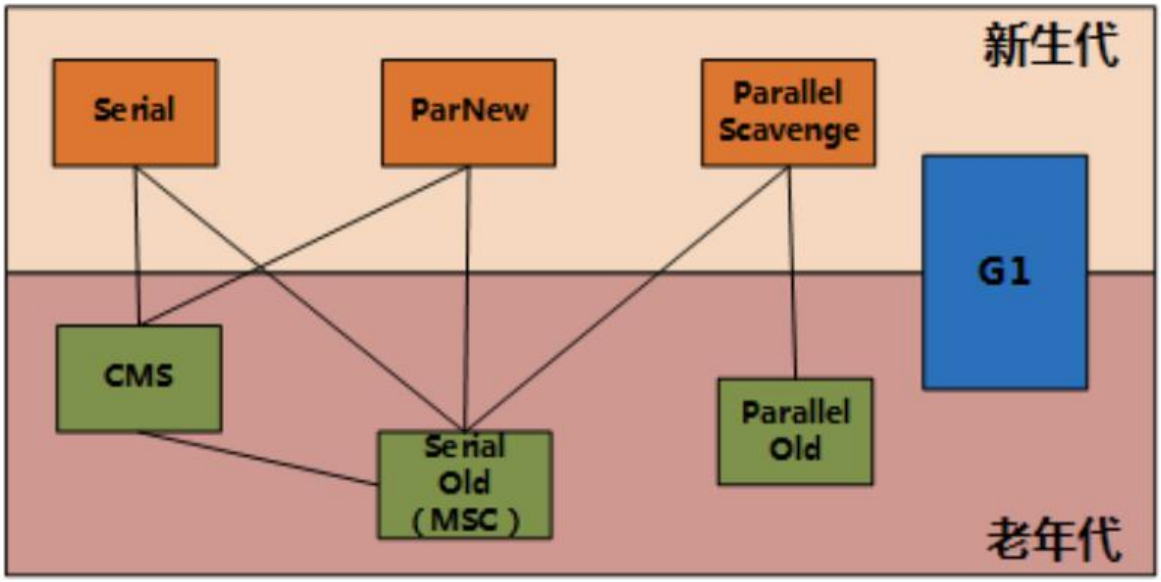
2. 老年代空间不足，当老年代空间不足后会触发Full GC,若触发后仍然不足，则会抛出 java.lang.OuOfMemoryError:Java heap space

3. 方法区(永久代)空间不足，方法区存放一些类信息，常量，和静态变量等数据，若系统加载的类，射的类和调用的方法较多的时候，永久代可能被占满，则触发Full GC，若经过Full GC仍然回收不了则会抛出 java.lang.OutOfMemoryError:PemGen space

4. 统计得到的Minor GC 晋升到旧生代的平均大小大于老年代的空间，则会触发Full GC

5. CMS GC 时出现 promotion failed 和 concurrent mode failure和promotion failed，就是上文说的担保失败，而 concurrent mode failure 是在执行 CMS GC 的过程中同时有对象要放入老年代而此时老年代空间不足造成的。

垃圾收集器



（A）、图中展示了7种不同分代的收集器：

Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1；

（B）、而它们所处区域，则表明其是属于新生代收集器还是老年代收集器：

新生代收集器：Serial、ParNew、Parallel Scavenge；

老年代收集器：Serial Old、Parallel Old、CMS；

整堆收集器：G1；

（C）、两个收集器间有连线，表明它们可以搭配使用：

表 3-2　垃圾收集相关的常用参数

参　数	描　述
UseSerialGC	虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial + Serial Old 的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用 ParNew + Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用 ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为 CMS 收集器出现 Concurrent Mode Failure 失败后的后备收集器使用
UseParallelGC	虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old（PS MarkSweep）的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收
SurvivorRatio	新生代中 Eden 区域与 Survivor 区域的容量比值，默认为 8，代表 Eden：Survivor=8：1
PretenureSizeThreshold	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	晋升到老年代的对象年龄。每个对象在坚持过一次 Minor GC 之后，年龄就增加 1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整 Java 堆中各个区域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况
ParallelGCThreads	设置并行 GC 时进行内存回收的线程数

（续）

参　数	描　述
GCTimeRatio	GC 时间占总时间的比率，默认值为 99，即允许 1% 的 GC 时间。仅在使用 Parallel Scavenge 收集器时生效
MaxGCPauseMillis	设置 GC 的最大停顿时间。仅在使用 Parallel Scavenge 收集器时生效
CMSInitiatingOccupancyFraction	设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集。默认值为 68%，仅在使用 CMS 收集器时生效
UseCMSCompactAtFullCollection	设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用 CMS 收集器时生效
CMSFullGCsBeforeCompaction	设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用 CMS 收集器时生效

class类文件结构

class 文件是一组8位字节为基础的二进制流，各个数据项目严格按照顺序紧凑的排列在Class文件中中间没有任何空格分隔符，这使得整个claas文件全部都是程序运行时必要的数据，没有一丁点的空隙在。当遇到需要占用8个字节以上空间的数据项时，则会按照高位在前的方式分割成若干个8字节进行储。

无符号数属于基本的数据类型，以 u1、u2、u4、u8 来分别代表 1 个字节、2 个字节、4 个字节和 8 个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照 UTF-8 编码构成字符串值。

表是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯性地以“_info”结尾。表用于描述有层次关系的复合结构的数据，整个 Class 文件本质上就是一张表，它由表 6-1 所示的数据项构成。

表 6-1　Class 文件格式

类　型	名　称	数　量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count-1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

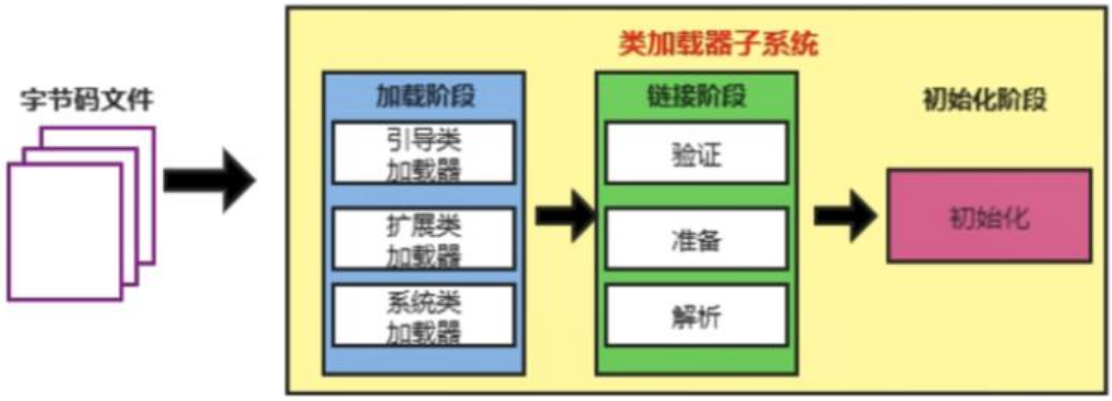
这张表里的名称含义大多都见名知意。值得一提的是，constant_pool 常量池

常量池中主要存放两大常量，字面量(Literal)和符号引用(Symbolic References)。字面量比较接近于java语言层面的常量含义，如用fianl声明的常量值等。而符号引用则属于编译原理方面的概念，包括了面3类常量：

- 类和接口的全限定名
- 字段的名称和描述符
- 方法的名称和描述符

类加载

类加载过程



- **第一步：Loading加载**

通过类的全限定名（包名 + 类名），获取到该类的二进制字节流

将二进制字节流所代表的静态存储结构，转化为方法区运行时的数据结构

在内存中生成一个代表该类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口

- **第二步：Linking链接**

链接是指将上面创建好的class类合并至Java虚拟机中，使之能够执行的过程，可分为验证、准备、解析个阶段。

① 验证（Verify）

确保class文件中的字节流包含的信息符合当前虚拟机的要求，保证这个被加载的class类的正确性，会危害到虚拟机的安全。

- 文件格式验证

首先要验证字节流是否符合class文件的格式规范，并且能被当前版本的虚拟机处理。这一阶段包括：

  - 是否已魔数　0xCAFEBABE开头
  - 主，次版本号是否在当前虚拟机的处理范围之内
  - 常量池中的常量是否存在不被支持的常量类型(检查常量flag标志)
  - 指向常量池的各种索引值是否有指向不存在的常量或不符合类型的常量
  - CONSTANT_Utf8_info 型常量是否有不符合utf8编码的数据
  - Class文件中的各个部分以及文件本身是否有被删除的或者附加的其他信息
  - 。。。。　（太多了，不列举了）

- 元数据验证

对字节码描述的信息做语义分析，对类的元数据信息做语义校验，保证不存在不符合java语言规范的数据信息。

- 字节码验证

通过数据流和控制流分析，确保程序语义是合法的，符合逻辑的，这个阶段会对方法体进行校验分析

保证这个方法在运行时不会做出危害虚拟机的安全事件

- 符号引用验证

最后一个阶段的校验发生在虚拟机将符号引用转换成直接引用的时候，这个转换动作将在连接的第三段--解析阶段中发生。此阶段可以看做是对类自身以外(常量池中的各种符号引用)的信息进行匹配行验。

② 准备（Prepare）

为类中的静态字段分配内存，并设置默认的初始值，比如int类型初始值是0。被final修饰的static字不会设置，因为final在编译的时候就分配了

③ 解析（Resolve）

解析阶段的目的，是将常量池内的符号引用转换为直接引用的过程（将常量池内的符号引用解析成为际引用）。如果符号引用指向一个未被加载的类，或者未被加载类的字段或方法，那么解析将触发这类的加载（但未必触发这个类的链接以及初始化。）

事实上，解析器操作往往会伴随着 JVM 在执行完初始化之后再执行。 符号引用就是一组符号来描述引用的目标。符号引用的字面量形式明确定义在《Java 虚拟机规范》的Class文件格式中。直接引用是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

解析动作主要针对类、接口、字段、类方法、接口方法、方法类型等。对应常量池中的 CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info等。

- **第三步：initialization初始化**

这个阶段主要是对**类变量**初始化，是执行类构造器的过程。

换句话说，只对static修饰的变量或语句进行初始化。

如果初始化一个类的时候，其父类尚未初始化，则优先初始化其父类。

如果同时包含多个静态变量和静态代码块，则按照自上而下的顺序依次执行。

类加载机制

类加载器

1. 启动类加载器（Bootstrap ClassLoader）：由C++语言实现。负责加载JAVA_HOME\lib目录中且能被虚拟机识别的类库到JVM内存中，如果名称不符合的类库即使放在lib目录中也不会被加载。该加载器无法被Java程序直接引用。

2. 扩展类加载器（Extension ClassLoader）：该加载器主要是负责加载JAVA_HOME\lib\ext，该加器可以被开发者直接使用。

3. 应用程序类加载器（Application ClassLoader）：该类加载器也称为系统类加载器，它负责加载户类路径（Classpath）上所指定的类库，开发者可以直接使用该类加载器，如果应用程序中没有自义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

双亲委派机制

工作原理：如果一个类加载器收到了类加载的请求，那么它首先不会尝试去加载该类，而是委托它的

类加载器去加载，每一层的类加载器都是如此，所以所有的加载请求都会传送到启动类加载器中。只父类无法完成加载请求时，子加载器才会尝试自己去加载。

代码清单 7-10　双亲委派模型的实现

```
    protected synchronized Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException
    {
        //首先，检查请求的类是否已经被加载过了
        Class c = findLoadedClass(name);
        if (c == null) {
            try {
            if (parent != null) {
                c = parent.loadClass(name, false);
            } else {
                c = findBootstrapClassOrNull(name);
            }
            } catch (ClassNotFoundException e) {
                //如果父类加载器抛出 ClassNotFoundException
                //说明父类加载器无法完成加载请求
            }
            if (c == null) {
                //在父类加载器无法加载的时候
                //再调用本身的 findClass 方法来进行类加载
                c = findClass(name);
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
```

破坏双亲委派

《深入理解Java虚拟机》这本书，读到破坏双亲委派机制这一小节，其中有一段话，如下

双亲委派模型的第二次“被破坏”是由这个模型自身的缺陷所导致的，双亲委派很好地解决了各个类载器的基础类的统一问题（越基础的类由越上层的加载器进行加载），基础类之所以称为“基础”，因为它们总是作为被用户代码调用的API，但世事往往没有绝对的完美，如果基础类又要调用回用户代码，那该怎么办?

这并非是不可能的事情，一个典型的例子便是JNDI服务，JNDI现在已经是Java的标准服务，它的代由启动类加载器去加载（在JDK 1.3时放进去的rt.jar），但JNDI的目的就是对资源进行集中管理和查，它需要调用由独立厂商实现并部署在应用程序的ClassPath下的JNDI接口提供者（SPI,Service Provier Interface）的代码，但启动类加载器不可能“认识”这些代码啊！那该怎么办?

为了解决这个问题，Java设计团队只好引入了一个不太优雅的设计：线程上下文类加载器（Thread Cntext ClassLoader）。这个类加载器可以通过java.lang.Thread类的setContextClassLoaser（）方进行设置，如果创建线程时还未设置，它将会从父线程中继承一个，如果在应用程序的全局范围内都有设置过的话，那这个类加载器默认就是应用程序类加载器。

有了线程上下文类加载器，就可以做一些“舞弊”的事情了，JNDI服务使用这个线程上下文类加载器加载所需要的SPI代码，也就是父类加载器请求子类加载器去完成类加载的动作，这种行为实际上就打通了双亲委派模型的层次结构来逆向使用类加载器，实际上已经违背了双亲委派模型的一般性原则但这也是无可奈何的事情。Java中所有涉及SPI的加载动作基本上都采用这种方式，例如JNDI、JDBC JCE、JAXB和JBI等。

JDBC中的逆双亲委派机制源码分析

首先，我们看一下JDBC连接数据库中加载驱动并获取连接的代码。

```
    try{
        //加载MySql的驱动类
        Class.forName("com.mysql.jdbc.Driver") ;
    }catch(ClassNotFoundException e){
        System.out.println("找不到驱动程序类 ， 加载驱动失败！ ");
        e.printStackTrace() ;
    }
    String url = "jdbc:mysql://localhost:3306/test" ;
    String username = "root" ;
    String password = "root" ;
    try{
        Connection con =  DriverManager.getConnection(url , username , password ) ;
    }catch(SQLException se){
        System.out.println("数据库连接失败！ ");
        se.printStackTrace() ;
    }
```

以上就是JDBC连接数据并获取连接的代码，那调用这些的方法到底做了些什么呢?

首先，我们用*Class.forName("com.mysql.jdbc.Driver")*加载了驱动，Driver的源码很简单，如下

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    public Driver() throws SQLException {
    }

    static {
        try {
            DriverManager.registerDriver(new Driver());
        } catch (SQLException var1) {
            throw new RuntimeException("Can't register driver!");
        }
    }
}
```

我们用系统类加载器加载了com.mysql.jdbc.Driver，类初始化的时候执行静态代码块，静态代码块中new了一个Driver实例并将他注册到DriverManager中。*注意，这里的Driver实例的类加载器是系统加载器。*接下来，我们调用了DriverManager.getConnection(String url,String user, String passwrd)，其源码如下

```
  @CallerSensitive
    public static Connection getConnection(String url,
        String user, String password) throws SQLException {
        java.util.Properties info = new java.util.Properties();
```

```
        if (user != null) {
            info.put("user", user);
        }
        if (password != null) {
            info.put("password", password);
        }

        return (getConnection(url, info, Reflection.getCallerClass()));
    }
```

其调用了另一段关键代码，如下

```
private static Connection getConnection(
        String url, java.util.Properties info, Class<?> caller) throws SQLException {
        /*
         * 这里要确保类加载不能是BootstrapClassLoader,
         * 因为BootstrapClassLoader不能加载到用户类库(JDBC驱动为用户类库)
         */
        ClassLoader callerCL = caller != null ? caller.getClassLoader() : null;
        synchronized(DriverManager.class) {
            // synchronize loading of the correct classloader.
            if (callerCL == null) {
                //获取系统类加载器
                callerCL = Thread.currentThread().getContextClassLoader();
            }
        }

        if(url == null) {
            throw new SQLException("The url cannot be null", "08001");
        }

        println("DriverManager.getConnection(\"" + url + "\")");

        // Walk through the loaded registeredDrivers attempting to make a connection.
        // Remember the first exception that gets raised so we can reraise it.
        SQLException reason = null;

        for(DriverInfo aDriver : registeredDrivers) {
            // If the caller does not have permission to load the driver then
            // skip it.
            if(isDriverAllowed(aDriver.driver, callerCL)) {
                try {
                    println("    trying " + aDriver.driver.getClass().getName());
                    Connection con = aDriver.driver.connect(url, info);
                    if (con != null) {
                        // Success!
                        println("getConnection returning " + aDriver.driver.getClass().getName());
                        return (con);
                    }
                } catch (SQLException ex) {
                    if (reason == null) {
                        reason = ex;
                    }
                }
```

```
            } else {
                println("    skipping: " + aDriver.getClass().getName());
            }

        }

        // if we got here nobody could connect.
        if (reason != null)    {
            println("getConnection failed: " + reason);
            throw reason;
        }

        println("getConnection: no suitable driver found for "+ url);
        throw new SQLException("No suitable driver found for "+ url, "08001");
    }
```

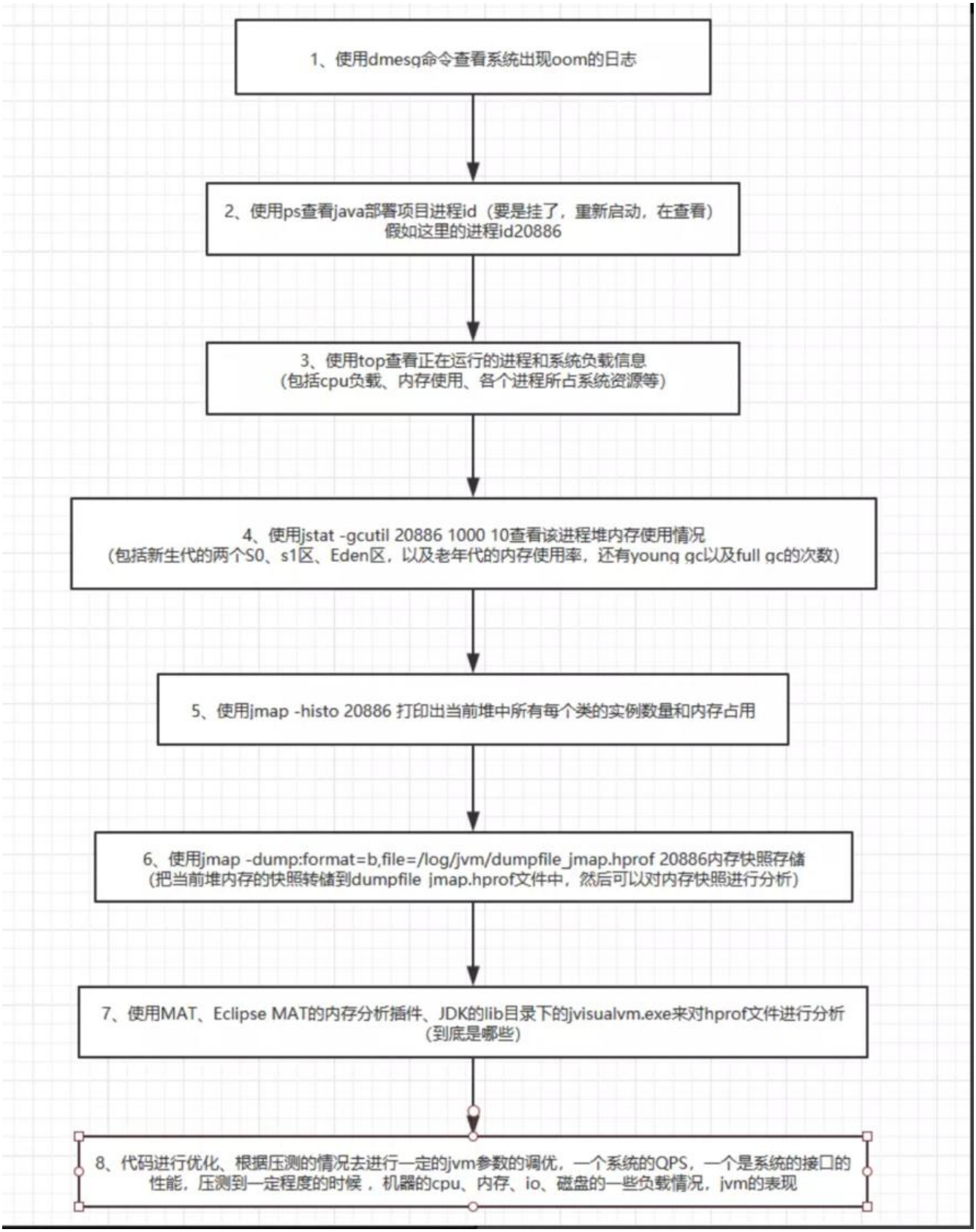
这段代码并没有使用ClassLoader

```
    private static boolean isDriverAllowed(Driver driver, ClassLoader classLoader) {
        boolean result = false;
        if(driver != null) {
            Class<?> aClass = null;
            try {
                aClass =  Class.forName(driver.getClass().getName(), true, classLoader);
            } catch (Exception ex) {
                result = false;
            }
            //类加载器与类相同才能确定==
             result = ( aClass == driver.getClass() ) ? true : false;
        }

        return result;
    }
```

小结，此处线程上下文类加载器的作用主要用于校验存放的driver是否和调用时的一致由此判断是否权限获取连接。

排查OOM



一般常见的OOM，要么是短时间内涌入大量的对象，导致你的系统根本支持不住，此时你可以考虑化代码，或者是加机器；要么是长时间来看，你的很多对象不用了但是还被引用，就是内存泄露了，也是优化代码就好了；这就会导致大量的对象不断进入老年代，然后频繁full gc之后始终没法回收，撑爆了

要么是加载的类过多，导致class在永久代理保存的过多，始终无法释放，就会撑爆