链滴

《深入理解 Java 虚拟机》笔记整理

作者：jingqueyimu

原文链接：https://ld246.com/article/1587048073600

来源网站：链滴

正文

一、Java 内存区域与内存溢出异常

1、运行时数据区域

- **程序计数器**：当前线程所执行的字节码的行号指示器。线程私有。
- **Java 虚拟机栈**：Java 方法执行的内存模型。线程私有。
- **本地方法栈**：Native 方法执行的内存模型。线程私有。
- **Java 堆**：存放对象实例。分为新生代（Eden 空间、From Survivor 空间、To Survivor 空间）和年代。线程共享。
- **方法区**：存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。也称“永久代”。线程共享。
- **运行时常量池**：方法区的一部分，用于存放编译期生成的各种字面量和符号引用。 线程共享。
- **直接内存。**

2、对象的创建

类加载检查 -> 分配内存 -> 初始化零值 -> 设置对象头 -> 执行 init 方法。

- **类加载检查**：检查 new 指令的参数能否在常量池中定位到一个类的符号引用，以及这个符号引用表的类是否已被加载、解析和初始化过。
- **分配内存**：把一块确定大小的内存从 Java 堆中划分出来。
- **初始化零值**：将分配到的内存空间初始化为零值（不包括对象头）。
- **设置对象头**：虚拟机需要对对象进行必要的设置，这些信息存放在对象的对象头中。
- **执行 init 方法**：把对象按照程序员的意愿进行初始化。

3、对象的内存布局

- **对象头**：
  - Mark Word：存储对象自身的运行时数据。
  - 类型指针：存储对象的类元数据的指针。
- **实例数据**：对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。
- **对齐填充**：仅仅起着占位符的作用。

4、对象的访问定位

- **句柄**：引用中存储的是对象的句柄地址。Java 堆中划分出一块内存作为句柄池，句柄中包含了对象例数据、类型数据两者的具体地址信息。
- **直接指针**：引用中存储的直接就是对象的地址。

5、OutOfMemoryError 异常

- Java 堆溢出。
- 虚拟机栈和本地方法栈溢出。
- 方法区和运行时常量池溢出。
- 本机直接内存溢出。

二、垃圾收集器与内存分配策略

1、判断对象是否可用

- **引用计数算法**：给对象添加一个引用计数器，每当有一个地方引用它时，计数器值加 1；当引用失时，计数器值减 1；任何时刻计数器为 0 的对象就是不可能再被使用的。
- **可达性分析算法**：通过一系列被称为“GC Roots”的对象作为起点，从这些节点开始向下搜索，索所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连时，则此对象不可用。

2、四种引用

- **强引用**：类似“Object obj = new Object()”的引用。只要强引用还存在，对象就永远不会回收。
- **软引用**：用来描述一些还有用但并非必需的对象。内存不足时，对象有可能被回收。
- **弱引用**：用来描述非必需的对象，但强度比软引用弱。GC时，无论内存是否足够，对象都会被回收。
- **虚引用**：也称幽灵引用或幻影引用，虚引用不会对对象的生存时间构成影响。虚引用的唯一作用就能在对象被回收时收到一个系统通知。

3、垃圾收集算法

- **标记-清除算法**：分为“标记”和“清除”两个阶段。首先标记出所有需要回收的对象，然后再统回收所有被标记的对象。会产生大量不连续的内存碎片。
- **复制算法**：将可用内存按容量划分为大小相等的两块，每次只使用其中一块。当一块内存用完时，将还存活的对象复制到另一块，然后再把已使用过的内存空间一次清理掉。
- **标记-整理算法**：首先标记出所有需要回收的对象，然后将所有存活对象向一端移动，最后直接清掉端边界以外的内存。
- **分代收集算法**：根据对象存活周期的不同，将 Java 堆划分为新生代和老年代，然后根据各个年代特点采用最适当的收集算法。
    - 新生代：采用复制算法。
    - 老年代：采用“标记-清除”或“标记-整理”算法。

4、垃圾收集器

- Serial 收集器：单线程。新生代收集器。
- ParNew 收集器：Serial 收集器的多线程版本。新生代收集器。
- Parallel Scavenge 收集器：多线程。新生代收集器。关注吞吐量。

- Serial Old 收集器：Serial 收集器的老年代版本。单线程。使用“标记-整理”算法。
- Parallel Old 收集器：Parallel Scavenge 收集器的老年代版本。多线程。使用“标记-整理”算法。
- CMS 收集器：并发收集器。使用“标记-清除”算法。关注点是如何缩短垃圾收集时用户线程的停时间。
- G1 收集器：面向服务端应用。并行与并发、分代收集、空间整合、可预测停顿时间。

5、内存分配与回收策略

- 对象优先在 Eden 分配。
- 大对象直接进入老年代。
- 长期存活的对象进入老年代。
- 动态对象年龄判定。
- 空间分配担保。

三、虚拟机性能监控与故障处理工具

1、JDK 的命令行工具

- **jps**：显示正在运行的虚拟机进程。常用命令：jps -l。
- **jstat**：监视虚拟机各种运行状态信息。常用命令：jstat -gcutil <pid>。
- **jinfo**：显示虚拟机配置信息。常用命令：jinfo -flags <pid>。
- **jmap**：主要用于生成堆转储快照。常用命令：jmap -dump:format=b,file=<filename> <pid>。
- **jhat**：分析 jmap 生成的堆转储快照。常用命令：jhat <filename>。
- **jstack**：显示虚拟机当前时刻的线程堆栈信息。常用命令：jstack -l <pid>。

2、JDK 的可视化工具

- JConsole：Java 监视与管理控制台。
- VisualVM：多合一故障处理工具。

四、类文件结构

1、无关性的基石

- 各种不同平台的 **虚拟机**。
- 所有平台都统一使用的 **字节码存储格式**。

2、Class 类文件的结构

（1）Class 文件的数据类型

● **无符号数**：基本数据类型，以 u1、u2、u4、u8 来分别代表 1 个字节、2 个字节、4 个字节和 8 字节的无符号数。用于描述数字、索引引用、数量值或按照 UTF-8 编码构成字符串值。

● **表**：由多个无符号数或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”结尾用于描述有层次关系的复合结构数据，整个 Class 文件本质上就是一张表。

（2）Class 文件格式

类型	名称	数量
u4	magic（魔数）	1
u2	minor_version（次版本号）	1
u2	major_version（主版本号）	1
u2	constant_pool_count（常量池容量计数器）	
cp_info	constant_pool（常量池）	onstant_pool_count - 1
u2	access_flags（访问标志）	1
u2	this_class（类索引）	1
u2	super_class（父类索引）	1
u2	interfaces_count（接口计数器）	1
u2	interfaces（接口索引集合）	terfaces_count
u2	fields_count（字段表计数器）	1
field_info	fields（字段表集合）	elds_count
u2	methods_count（方法表计数器）	1
method_info	methods（方法表集合）	ethods_count
u2	attributes_count（属性表计数器）	1
attribute_info	attributes（属性表集合）	ttributes_count

● 魔数：Class 文件的头 4 个字节，用于确定该文件是否为 Class 文件。其值为：0xCAFEBABE（咖宝贝？）。

● Class 文件的版本：第 5、6 个字节是次版本号，第 7、8 个字节是主版本号。

● 常量池：可以理解为 Class 文件中的资源仓库。主要存放字面量和符号引用。每一项常量都是一个。

● 访问标志：用于识别一些类或接口层次的访问信息，包括：这个 Class 是类还是接口、是否定义为 ublic、是否定义为 abstract、是否声明为 final（只有类可设置）等。

● 类索引、父类索引与接口索引集合：Class 文件由这三项数据确定这个类的继承关系。

● 字段表集合：用于描述接口或类中声明的变量。包括类变量和实例变量，但不包括在方法内部声明局部变量。

● 方法表集合：用于描述接口或类中声明的方法。

● 属性表集合：在 Class 文件、字段表、方法表都可以携带自己的属性表集合，以用于描述某些场景

有的信息。

3、字节码指令简介

- 加载和存储指令：用于将数据在栈帧中的局部变量表和操作数栈之间来回传输。
- 运算指令：用于对两个操作数以上的值进行某种特定运算，并把结果重新存入到操作数栈顶。
- 类型转换指令：将两种不同的数值类型进行相互转换。
- 对象创建与访问指令。
- 操作数栈管理指令：用于直接操作操作数栈。
- 控制转移指令：让 Java 虚拟机有条件或无条件地从指定位置的指令继续执行程序，而不是从控制移指令的下一条指令继续执行程序。可认为控制转移指令就是在有条件或无条件地修改 PC 寄存器的。
- 方法调用和返回指令。
- 异常处理指令。
- 同步指令：支持方法级的同步和方法内部一段指令序列的同步。

五、虚拟机类加载机制

1、类加载的过程

加载 -> 连接（验证、准备、解析） -> 初始化。

- **加载**：获取二进制字节流，并在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区个类的各种数据的访问入口。
- **验证**：确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的全。
    - 文件格式验证：验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理。
    - 元数据验证：对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范的要。
    - 字节码验证：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。
    - 符号引用验证：对符合引用进行匹配性校验，确保解析动作能正常执行。
- **准备**：为类变量分配内存并设置初始值。
- **解析**：将常量池内的符号引用替换为直接引用。
- **初始化**：根据程序员的主观计划去初始化类变量和其他资源。

2、类加载器

- **启动类加载器**：负责将存放在 <JAVA_HOME>\lib 目录的，或者 -Xbootclasspath 参数所指定路径中的，能被虚拟机识别的类库加载到虚拟机内存。
- **扩展类加载器**：负责加载 <JAVA_HOME>\lib\ext 目录中的，或者 java.ext.dirs 系统变量所指定路径中的所有类库。

- **应用程序类加载器**：负责加载用户类路径上所指定的类库。

3、双亲委派模型

如果一个类加载器收到类加载的请求，它会先把这个请求委派给父加载器去完成，而不会自己去尝试载这个类。只有父加载器无法完成这个加载请求时，子加载器才会尝试自己去加载。

六、虚拟机字节码执行引擎

1、运行时栈帧结构

栈帧是用于支持虚拟机进行方法调用和方法执行的数据结构。栈帧存储了方法的局部变量表、操作数、动态连接、方法返回地址和一些额外的附加信息。每一个方法从调用开始至执行完成的过程，都对着一个栈帧在虚拟机里面从入栈到出栈的过程。

- **局部变量表**：是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。
- **操作数栈**：也称为操作栈，它是一个后入先出的栈。操作数栈的每一个元素可以是任意的 Java 数类型。
- **动态连接**：每个栈帧都包含一个指向运行时常量池中，该栈帧所属方法的引用，持有这个引用是为支持方法调用过程中的动态连接。
- **方法返回地址**：方法退出后需要返回到方法被调用的位置，程序才能继续执行。
- **附加信息**：虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧中，例如与调相关的信息。

2、方法调用

方法调用并不等于方法执行，方法调用阶段唯一的任务就是确定被调用方法的版本（即调用哪一个方）。此时，在 Class 文件里存储的只是符号引用，而不是直接引用，只有在类加载期间，甚至是运行间才能确定目标方法的直接引用。

- **解析**：在类加载的解析阶段，将方法的符号引用转化为直接引用，这类方法调用称为解析。这种解能成立的前提是：方法在程序执行之前有一个可确定的调用版本，并且这个方法的调用版本在运行期可改变，即“编译期可知，运行期不可变”。
- **分派**：
    - 静态分派：在编译期依赖静态类型（又称外观类型）来定位方法执行版本的分派动作，称为静分派。静态分派的典型应用是方法重载。
    - 动态分派：在运行期根据实际类型确定方法执行版本的分派过程，称为动态分派。动态分派的型应用是方法重写。

七、早期（编译期）优化

1、Javac 编译过程

（1）解析与填充符号表

● **词法分析**：将源代码的字符流转变为标记（Token）集合，标记是编译过程的最小元素，关键字、量名、字面量、运算符都可以成为标记。

● **语法分析**：根据 Token 序列构造抽象语法树。

● **填充符号表**：符号表是由一组符号地址和符号信息构成的表格，可以把它想象成哈希表中 K-V 值对形式。

（2）注解处理

在编译期间对注解进行处理。可以读取、修改、添加抽象语法树中的任何元素。

（3）语义分析与字节码生成

● **语义分析**：对结构上正确的源程序进行上下文逻辑审查。

  ● 标注检查：包括变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配等。

  ● 数据及控制流分析：对程序上下文逻辑进行更进一步的验证，包括局部变量在使用前是否有赋、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理等。

● **解语法糖**：虚拟机运行时并不支持语法糖的语法，因此，需要在编译阶段还原回简单的基础语法结。

● **字节码生成**：把前面各个步骤所生成的信息（语法树、符号表）转化成字节码写到磁盘中，同时还行了少量的代码添加和转换工作。

2、Java 语法糖

● 泛型与类型擦除：泛型的本质是参数化类型的应用，即将所操作的数据类型指定为一个参数。

● 自动装箱与拆箱、遍历循环、变长参数。

● 条件编译：编译器在编译时只对满足条件的代码进行编译，而将不满足条件的代码舍弃。Java 语言以使用条件为布尔常量值的 if 语句进行条件编译。

八、晚期（运行期）优化

1、HotSpot 虚拟机内的即时编译器

（1）解释器与编译器

● 当程序需要迅速启动和执行时，解释器可以首先发挥作用，省去编译的时间，立即执行。

● 在程序运行后，随着时间的推移，编译器把越来越多的代码编译成本地代码后，可以获取更高的执效率。

（2）C1、C2 编译器

● C1 编译器（Client Compiler）：运行在 Client 模式。

● C2 编译器（Server Compiler）：运行在 Server 模式。

（3）混合模式、解释模式与编译模式

- 混合模式：解释器与编译器搭配使用的方式。
- 解释模式：全部代码都使用解释方式执行，编译器完全不介入工作。
- 编译模式：优先采用编译方式执行，但是解释器仍会在编译无法进行时介入执行过程。

（4）分层编译

分层编译根据编译器编译、优化的规模与耗时，划分出不同的编译层次。

- 第 0 层：程序解释执行，解释器不开启性能监控功能，可触发第 1 层编译。
- 第 1 层：也称 C1 编译，将字节码编译为本地代码，进行简单、可靠的优化，必要时加入性能监控逻辑。
- 第 2 层（或 2 层以上）：也称 C2 编译，也是将字节码编译为本地代码，但会启用一些编译耗时较的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。

2、即时编译触发条件

（1）热点代码

- 被多次调用的方法。
- 被多次执行的循环体。

（2）热点探测

判断一段代码是不是热点代码，是不是需要触发即时编译，这样的行为称为热点探测。

- **基于采样的热点探测**：虚拟机周期性地检查各个线程的栈顶，如果发现某个方法经常出现在栈顶，这个方法就是“热点代码”。
- **基于计数器的热点探测**：虚拟机为每个方法（甚至是代码块）建立计数器，统计方法的执行次数，果执行次数超过一定阈值就认为它是“热点代码”。

HotSpot 虚拟机使用的是基于计数器的热点探测方法，它为每个方法准备了两类计数器。

- 方法调用计数器：统计方法被调用的次数。
- 回边计数器：统计一个方法中循环体代码执行的次数。

3、编译优化技术

- 公共子表达式消除：如果一个表达式 E 已经计算过了，并且从先前计算到现在 E 中所有变量的值都有变化，那么 E 的这次出现就成了公共子表达式。对于这种表达式，没有必要再次进行计算，直接用面计算过的表达式结果代替 E 即可。
- 数组边界检查消除：编译器通过数据流分析判定数组下标是否会越界，如果分析后确定不会越界，么可以把数组的上下界检查消除。
- 方法内联：把目标方法的代码“复制”到发起调用的方法之中，避免发生真实的方法调用。

● 逃逸分析：当一个对象在方法中定义后，如果它被外部方法所引用或被外部线程访问到，那么就说个对象发生了逃逸。如果一个对象不会逃逸到方法或线程之外，那么可以为这个变量进行一些高效的化，比如栈上分配、同步消除、标量替换等。

九、Java 内存模型与线程

1、Java 内存模型

（1）主内存与工作内存

● 所有的变量都存储在主内存中。每条线程有自己的工作内存，工作内存中保存了被该线程使用到的量的主内存副本拷贝。

● 线程对变量的操作必须在工作内存中进行，而不能直接读写主内存中的变量。

● 不同的线程之间无法直接访问对方工作内存中的变量，线程间变量值的传递需要通过主内存来完成。

（2）内存间交互操作

● **lock（锁定）**：把一个主内存变量标识为一条线程独占的状态。

● **unlock（解锁）**：把一个处于锁定状态的主内存变量释放出来。

● **read（读取）**：把一个变量的值从主内存传输到线程的工作内存中，以便随后的 load 动作使用。

● **load（载入）**：把 read 操作从主内存中得到的变量值放入工作内存的变量副本中。

● **use（使用）**：把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量值的字节码指令时将会执行这个操作。

● **assign（赋值）**：把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量值的字节码指令时执行这个操作。

● **store（存储）**：把工作内存中一个变量的值传送到主内存中，以便随后的 write 操作使用。

● **write（写入）**：把 store 操作从工作内存中得到的变量的值放入主内存的变量中。

（3）volatile 的作用

● 保证变量对所有线程的可见性。

● 禁止指令重排序优化。

（4）原子性、可见性与有序性

● **原子性**：

  ● 基本数据类型的访问读写具备原子性： Java 内存模型直接保证了 read、load、assign、use、store 和 write 操作的原子性。

  ● synchronized 代码块之间的操作具备原子性：底层通过 lock 和 unlock 操作实现。

● **可见性**：当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。Java 内存模型通过在量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方来实现可见性。

● **有序性**：如果在本线程内观察，所有的操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。前半句是指“线程内表现为串行的语义”，后半句是指“指令重排序”现象和“工内存与主内存同步延迟”现象。

（5）先行发生原则

● **程序次序规则**：在一个线程内，按照程序代码顺序，书写在前的操作先行发生于书写在后的操作。确地说，是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。

● **管程锁定规则**：一个 unlock 操作先行发生于后面（时间上的先后顺序）对同一个锁的 lock 操作。

● **volatile 变量规则**：对一个 volatile 变量的写操作先行发生于后面（时间上的先后顺序）对这个变的读操作。

● **线程启动规则**：Thread 对象的 start() 方法先行发生于此线程的每一个动作。

● **线程终止规则**：线程中的所有操作都先行发生于对此线程的终止检测。

● **线程中断规则**：对线程 interrupt() 方法的调用先行发生于被中断线程检测到中断事件的发生。

● **对象终结规则**：一个对象的初始化完成（构造函数执行结束）先行发生于它的 finalize() 方法的开。

● **传递性**：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么可以得出操作 A 先行发于操作 C。

2、Java 与线程

（1）线程的实现

● 使用内核线程实现：内核线程就是直接由操作系统内核支持的线程。

● 使用用户线程实现：用户线程完全建立在用户空间的线程库上，系统内核不能感知线程的存在。

● 使用用户线程加轻量级进程混合实现：用户线程还是完全建立在用户空间中，而操作系统提供支持轻量级进程则作为用户线程和内核线程之间的桥梁。

（2）Java 线程调度

● **协同式线程调度**：线程的执行时间由线程本身来控制，线程执行完之后，主动通知系统切换到另外个线程上。

● **抢占式线程调度**：每个线程由系统来分配执行时间，线程的切换不由线程本身来决定。

Java 使用的线程调度方式就是抢占式调度。

（3）线程状态

● **新建**（New）：线程创建后尚未启动。

● **运行**（Runable）：包括了操作系统线程状态中的 Running 和 Ready，处于此状态的线程有可能在执行，也有可能正在等待着 CPU 为它分配执行时间。

● **无限期等待**（Waiting）：不会被分配 CPU 执行时间，等待着被其他线程显式地唤醒。

● **限期等待**（Timed Waiting）：不会被分配 CPU 执行时间，无须等待被其他线程显式地唤醒，在定时间之后会由系统自动唤醒。

● **阻塞**（Blocked）：线程被阻塞了，在等待着获取到一个排他锁。在程序等待进入同步区域的时候线程将进入这种状态。

● **结束**（Terminated）：已终止线程的线程状态，线程已经结束执行。

十、线程安全与锁优化

1、Java 语言中的线程安全

按线程安全的“安全程度”由强至弱排序，可以将多个线程的共享数据分为 5 类：不可变、绝对线程全、相对线程安全、线程兼容和线程对立。

● **不可变**：不可变的对象一定是线程安全的，无论是对象的方法实现还是方法的调用者，都不需要再取任何的线程安全保障措施。

● **绝对线程安全**：必须满足“不管运行时环境如何，调用者都不需要任何额外的同步措施”。

● **相对线程安全**：就是我们通常意义上所讲的线程安全，它需要保证对一个对象单独的操作是线程安的，但是对于一些特定顺序的连续调用，则需要在调用端使用额外的同步手段来保证调用的正确性。

● **线程兼容**：对象本身并不是线程安全的，但可以通过在调用端正确地使用同步手段来保证对象在并环境中可以安全地使用。

● **线程对立**：无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。

2、线程安全的实现方法

● **互斥同步（阻塞同步）**：同步是指在多个线程并发访问共享数据时，保证共享数据在同一个时刻只一个线程使用，而互斥是实现同步的一种手段。

● **非阻塞同步**：在进行同步操作时，不需要把线程挂起，而是先进行操作，如果没有其他线程争用共数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就采取其他的补偿措施。

● **无同步方案**：

● 可重入代码（纯代码）：如果一个方法的返回结果是可以预测的，只要输入了相同的数据，就能返回相同的结果，那它就满足可重入性的要求，当然也就是线程安全的。

● 线程本地存储：如果能保证使用共享数据的代码在同一个线程中执行，那么就可以把共享数据可见范围限制在同一个线程之内。这样，无须同步也能保证线程之间不出现数据争用的问题。

3、锁优化

● **自旋锁**：如果物理机有多个处理器，能让多个线程同时并行执行，那么可以让后面请求锁的线程“等一下”，但不放弃处理器的执行时间，然后看看持有锁的线程是否很快就会释放锁。为了让线程等，只需让线程执行一个忙循环（自旋），这就是所谓的自旋锁。

● **锁消除**：锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存共享数据竞争的锁进行消除。

● **锁粗化**：如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体中，那么虚拟机将会把加锁同步的范围扩展（粗化）到整个操作序列的外部，这样只需要加锁一次就可了。

● **轻量级锁**：轻量级锁并不是用来代替重量级锁的，而是在没有多线程竞争的前提下，减少传统的重级锁使用操作系统互斥量产生的性能消耗。对象头的 Mark Word 有个锁标志位，用于标识同步对象锁状态。

● **偏向锁**：偏向锁是指这个锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁没有其他线程获取，则持有偏向锁的线程将永远不需要再进行同步。

相关文章

《深入理解 Java 虚拟机》读书笔记：Java 内存区域与内存溢出异常

《深入理解 Java 虚拟机》读书笔记：垃圾收集器与内存分配策略

《深入理解 Java 虚拟机》读书笔记：虚拟机性能监控与故障处理工具

《深入理解 Java 虚拟机》读书笔记：类文件结构

《深入理解 Java 虚拟机》读书笔记：虚拟机类加载机制

《深入理解 Java 虚拟机》读书笔记：虚拟机字节码执行引擎

《深入理解 Java 虚拟机》读书笔记：早期（编译期）优化

《深入理解 Java 虚拟机》读书笔记：晚期（运行期）优化

《深入理解 Java 虚拟机》读书笔记：Java 内存模型与线程

《深入理解 Java 虚拟机》读书笔记：线程安全与锁优化

交流区

```
<p align="center">
   <img src="https://b3logfile.com/file/2020/04/qrcode-8e461b12.jpg" height="250px" widt
="250px"><br/>
   微信公众号：惊却一目<br/>
   个人博客：<a href="https://www.jingqueyimu.com">惊却一目</a>
</p>
```