JVM相关

笔记本:	工作笔记		
创建时间:	2020/11/4 19:30	**更新时间:**	2020/11/8 11:21
作者:	438842220@qq.com		
URL:	about:blank		

`JMM/类加载器/垃圾回收/java内存区域/调优`

JMM:

https://www.jianshu.com/p/8a58d8335270

现代计算机的内存模型:

为了平衡CPU和内存之间速度的矛盾引入了缓存。从而引发了缓存一致性问题，当多个线程共享同一块内存时这个问题就会出现。为了解决这个问题，就需要缓存一致性协议。

为了平衡写的速度，引入了写缓冲区。但写缓冲区只对自己可见。因此可能导致读写的顺序不一致。

java内存模型（JMM）:

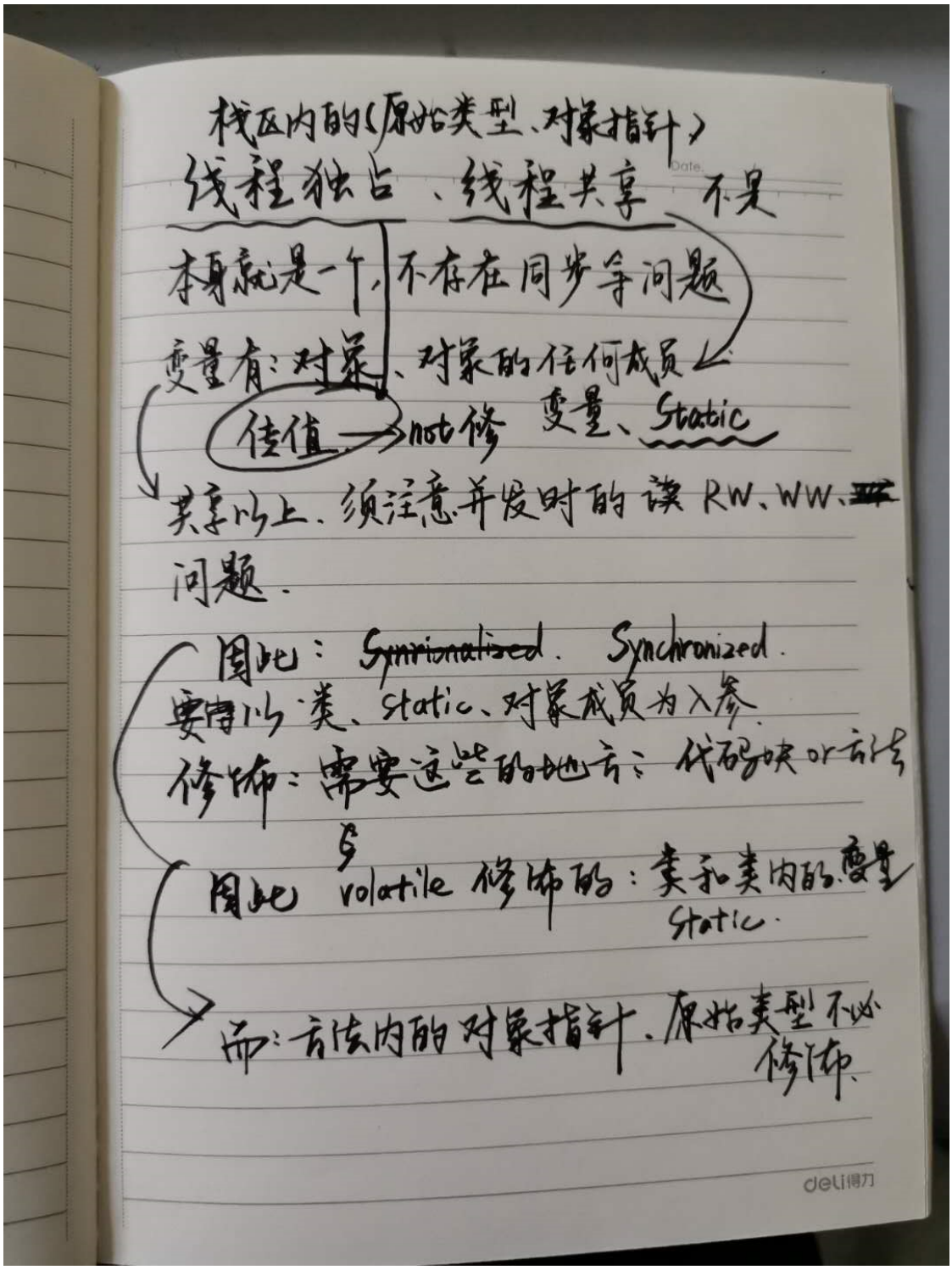
JMM是隶属于JVM的，定义了JVM在内存中的工作方式。

JAVA线程 ------工作内存-----save/load操作----主内存

JAVA线程 ------工作内存------↑

JAVA线程 ------工作内存------↑

JAVA线程 ------工作内存------↑



类加载器：

我们需要了解JAVA字节码定义，以及有了字节码之后的类加载流程。

然后再去了解类加载器的树形结构，这里面还有个双亲委托。最后自己定义类加载器。这样就基本完成了。

JAVA字节码，也就是.class类型的文件。它是跑在JVM上的中间代码。

包括的内容主要有：魔数/版本/常量池/访问标志（public/private等）/类索引，父类索引，接口索引集合/字段表集合/方法表集合/属性表集合

类生命周期：加载/验证/准备/解析/初始化/使用/卸载

其中234步统称连接。其中解析可能在初始化后，这个和运行时绑定有关。

在这之中，并不是所有引用类的时候都会触发初始化。这里面分主动引用和被动引用。

类初始化要求父类初始化，但接口，除非实际用到，否则不要求父类初始化。

一句话解释类加载的过程：

加载：把.class文件（这个二进制字节流）读取，转化为运行时数据结构，并生成代表这个类的java.lang.Class

验证：文件格式验证/元数据验证（父类相关）/字节码验证（确定语义）/符号引用验证（是否能访问类和字段）

准备：正式给静态变量分配内存

解析：将符号引用替换为直接引用（确定的指针）

初始化：将类中的所有变量赋值操作和静态语句块生成一个<clinit>()，当然如果没有，可以不生成

类加载器不是JVM内部实现，而是放在外部由应用程序实现。这个机制带来了许多好处（OSGI热部署/代码加密）（比如你在加载器里读加密的类，再解密，这样就实现了代码加密的过程）

类加载器的结构：

虚拟机角度：启动类加载器（C++实现，虚拟机自身部分）/其他类加载器（JAVA实现，独立于虚拟机外部，都是继承同一抽象类）

程序员角度：启动类加载器（JAVA_HOME/lib）---->扩展类加载器（JAVA_HOME/lib/ext）---->应用程序类加载器（类库/开发者代码）

双亲委派：有问题先找双亲，双亲继续如此，找不到在向下传递。（它是一种保证不重名的巧妙机制）

也有出于其他目的，打破双亲委派机制的情况。（OSGI热替换的实现）

自己定义类加载器，参考：
https://blog.csdn.net/qq_24451605/article/details/51202064?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.edu_weight&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.edu_weight

另外深入理解JVM第9章也有相关参考。

垃圾回收：

三个问题：何为垃圾？何时回收？如何回收垃圾？

每个线程内的（程序计数器/虚拟机栈/本地方法栈）随线程而生死，不必考虑。但JAVA堆区和方法区则不一样。

对象：引用计数（循环引用）/可达性分析

这里面的引用也分为几种：强引用/软引用（内存溢出前回收）/弱引用（GC时回收）/虚引用（不影响生命周期，用来做通知）

方法区：常量（有无引用）/类（条件比较多，实例/类加载器/对应的java.lang.class无引用）而且确认了之后也不是必然回收的

以上我们确认了什么是垃圾。

垃圾收集方法：普遍基于分代回收的原则。

标记清除（碎片）

标记复制（分两片，互相复制）

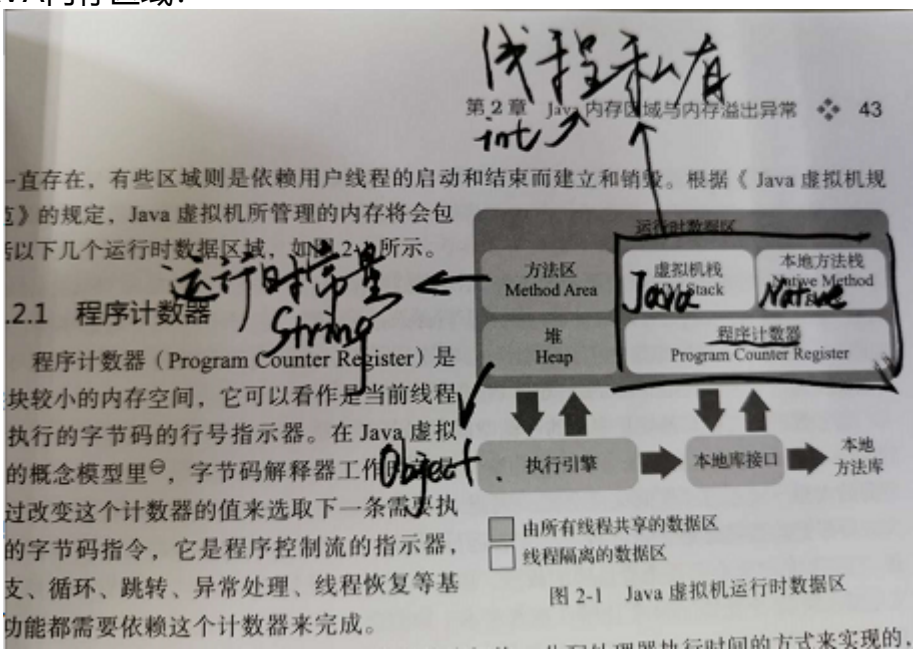
标记整理（后边的挪到前边去）

几款垃圾收集器：

serial　　单线程　　新生代复制　　老生代标记整理

parnew　　多线程
parallel scavenge　　吞吐量大（就是停顿比低）
serial old　　以上的老年代版本
parallel old　　以上的老年代版本
cms　　最短目标停顿时间/吞吐量自然低了/浮动垃圾收集不了/标记清除会有碎片　　（初始标记(stw)/并发标记/重新标记/标记清除(stw)）
G1　　将堆化为region，每次回收最有价值的region。每个region代表的年代角色不同。　目前小内存cms/大内存G1，但优化在向G1靠拢。

JAVA内存区域：



调优：