### ##### JAVA程序运行流程#####

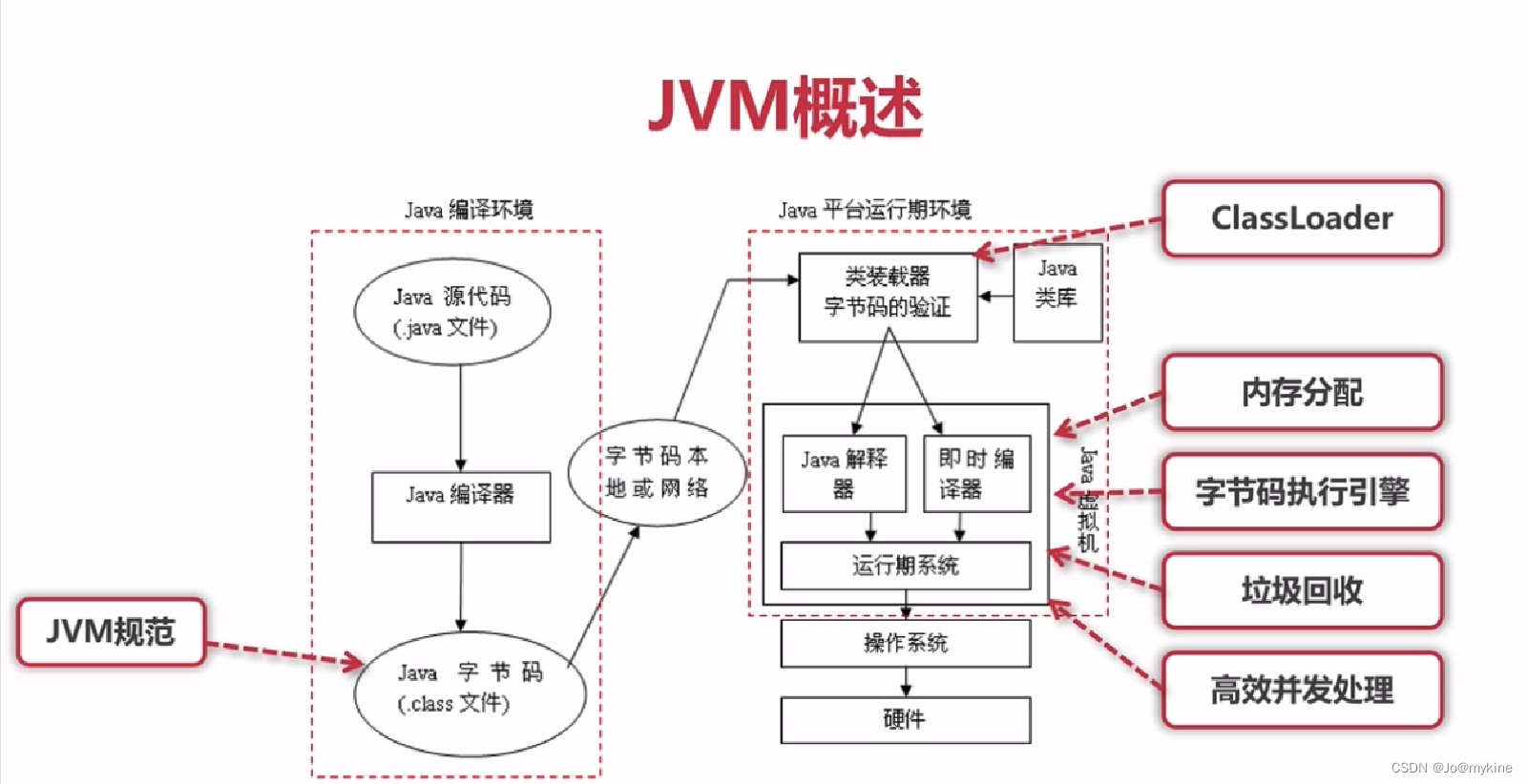
首先，java源代码被编译成.class文件，即字节码文件，

然后，.class文件被类加载器加载到内存，

然后，字节码引擎解析内存中的字节码，翻译成机器可执行的指令，运行程序，

然后，程序运行期间会产生并发安全问题，进行并发安全处理，

最后，程序运行期间还会产生内存垃圾，要进行内存回收，也就是GC



### ##### JVM运行时数据区的内存结构#####

堆：存储对象、JAVA7后开始存静态变量、字符串常量池

虚拟机栈：存储线程执行JAVA方法时的本地变量

程序计数器：存放下一条要执行的字节码指令

方法区：存放常量、类的元信息

本地方法栈：存放执行C/C++代码的变量

### ##### JAVA内型模型JMM#####

JAVA内存模型也叫JMM,是JAVA用于解决多线程并发访问主内存中共享变量的操作规范，

规定了一个线程如何、何时看到其他线程修改后的共享变量值以及在何时同步共享变量值。在该规范下实现并发操作的**原子性、可见性、有序性**这三大要素，也就保证了并发的安全性。

这背后的技术背景涉及到CPU和内存硬件的物理架构特性，CPU计算速度远大于内存，为了最大发挥CPU的性能，在CPU与内存架构之间中添了高速缓存寄存器，将内存的数据先与高速缓存进行交互，再由CPU对高速缓存进行操作，就解决了这种硬件之间速率不匹配的问题，但是这样会带来**缓存数据不一致**的问题。多个CPU同时将主内存中共享变量加载到各自的高速缓存中操作最后将高速缓存中的结果都写回主内存，有可能会导致数据不一致，**缓存不一致**问题就会对应了并发特性的**可见性问题**。

此外，为了提高CPU执行程序的效率，编译器或CPU会对程序命令进行**指令重排**，这会导致多线程执行同一批程序时，执行的指令顺序可能不一样，出现线程安全问题。

**指令重排**就对应了并发特性的**有序性问题**。

**如何解决CPU通过高速缓存与主内存交互带来的缓存一致性问题：**

**让CPU操作高速缓存和主内存时遵循一定的协议，比如常见的MESI-通用缓存一致性协议**，该协议是基于**总线嗅探的缓存锁定**技术来实现总线事务，CPU执行指令操作数据是通过系统总线和内存总线来和内存交互的，当CPU执行一个线程任务进行写数据更新时，这个写事务会被总线上的窥探器嗅探到，窥探器会检查该变量在其他线程任务对应的工作内存中是否也存在副本，存在就让这些副本置为都失效状态，当这些线程读到自己工作内存的副本变量为失效状态时，会等待那个写事务完成后，重新从主内存读取最新的值更新到工作内存，才能用该副本的值进行自己的下一步操作，这样就保证了缓存的一致性，这也是java关键字volatile能实现共享变量对所有线程可见的底层原理。

**JMM如何保证原子性：**

规定了数据同步在主内存与工作内存之间的8大原子操作：

1. lock(锁定)：作用于主内存的变量，把一个变量标记成线程独占状态
2. unlock(解锁)：作用于主内存变量，释放变量锁状态
3. read(读取)：作用于主内存变量，将主内存变量值读取到线程的工作内存中
4. load(载入)：作用于工作内存变量，将read到的主内存变量值复制到工作内存变量
5. use(使用)：作用于工作内存变量，把工作内存的变量值传递给执行引擎去执行
6. assign(赋值)：作用于工作内存变量，将执行引擎执行的结果值写回工作内存变量
7. store(存储)：作用于工作内存变量，将工作内存变量值传递到主内存
8. Write(写入)：作用于工作内存变量，将store操作传进来的值写入主内存变量中

JMM保证线程对一个变量在内存的操作遵循以上8个原子操作顺序。

JAVA代码可以通过CAS、Synchronized、Lock等各种锁实现同步代码块内的程序操作的原子性。

**JMM如何保证可见性和有序性：**

使用**volatile**关键字修饰变量：

volatile关键字**能让被修饰的变量对所有线程可见**，还**能禁止指令重排。**

volatile关键字底层原理：使用了**cpu的lock前缀**指令，

**lock前缀**指令具有**总线嗅探、MESI-通用缓存一致性协议功能，**能保证多核缓存

一致性问题，所以volatile关键字能保证可见性；

**lock前缀**指令还具有**内存屏障**功能，通过对变量的读写操作前后加入内存屏障指令，告诉编译器和CPU，屏障指令前后的操作不能进行重排序，所以volatile关键字能禁止指令重排。

X86架构的CPU内存屏障有4种：

**lfence**：是一种Load Barrier 读屏障

**sfence**：是一种Store Barrier 写屏障

**mfence**：是一种全能型的屏障，具备ifence和sfence的能力

**lock前缀**：Lock不是一种内存屏障，但是它能完成类似内存屏障的功能。Lock会对CPU总线和高速缓存加锁，可以理解为CPU指令级的一种锁。它后面可以跟ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG等指令。

JVM提供了4种类型的内存屏障：

1. LoadLoad：在load2读取之前，保证load1读取数据全部完毕，
2. StoreStore：在store2写入之前，保证store1写操作已完成，
3. LoadStore：在store2写入之前，保证load1读取数据全部完毕，
4. StoreLoad：在load2读取之前，保证store1写操作已完成,是**万能屏障,**包含以上3个屏障的功能。

**JMM的happens-before原则：**

从JDK1.5开始，JMM使用happens-before概念来阐述多线程之间的内存可见性。

happens-before（先行发生）原则的定义：

1. **程序执行次序规则**：在一个线程内，执行一段代码指令，前面的代码指令优先执行于后面的代码指令。
2. **锁定规则**：一个unlock操作先行发生于后面对这个锁的lock操作。
3. **volatile变量规则**：对被volatile修改的变量的写操作优先发生于对这个变量的读操作。
4. **线程启动规则**：thread的start()方法优先发生于此线程的每一个动作。
5. **线程终止规则：**线程的所有操作都优先发生于对该线程的终止操作。
6. **线程中断规则：**线程的interrupt()方法优先发生于该线程中判断中断事件代码的执行。
7. **对象终结规则**：对于同一个对象，他的构造方法执行结束优先发生于它的finalize()方法的开始。
8. **传递规则**：如果操作A线性发生于操作B，操作B先行发生于操作C，则A先行于C。

### ##### GC收集器和算法#####

**1.GC中判断对象存活的方法--根搜索算法：**

将常量、静态变量的引用作为根集中的根，从根开始顺藤摸瓜搜索对象，能搜索到的对象都是存活的，否则就是要被回收的。

1. **常见的GC收集器和使用的回收算法**
2. **串行类收集器**（GC单线程进行内存回收，会暂停所有用户线程）：

**Serial收集器**，作用于新年代，采用复制算法；

**SerialOld收集器**，作用域老年代，采用双色标记整理算法；

优点：简单，对单核CPU更高效，适合桌面客户端应用

缺点：无法实现并发GC，GC时用户线程要全部停止

1. **并行类收集器**（多个GC线程并发工作，但是用户线程仍然会暂停工作）：

**ParNew收集器**：作用于新时代，采用复制算法；

**ParOld收集器**：作用于老年代，采用标记整理算法

优点：相对串行收集器，GC线程并发执行，GC效率高

缺点：GC时仍然要完全停止所有用户线程，会出现较长时间的stop the world

1. **并发类收集器：**

这里的并发是指：GC线程和用户线程并发执行，不需要停止用户线程

**CMS收集器：**作用于老年代，采用并发清理算法

GC过程：初始标记、并发标记、重新标记、并发清除。

其中，初始标记和重新标记会触发STW

优点：GC时不用完全停止用户线程，GC线程和用户线程大部分时间内可以并发执 行，大幅度减少stop the world的时间。

缺点：采用的标记清除算法会产生大量的内存碎片，在分配大内存对象时可能触发FullGC。

**G1收集器：**作用于新生代和老年代，但是这里的新生代和老年代是逻辑上的，因 为G1收集器从物理上将堆内存划分成多个相同大小并且内存地址连续的区域单元， 叫做region，G1收集器将一部分region集合划分为逻辑上的新生代，一部分region 集合划分为逻辑上的老年代。G1收集器使用标记整理法+复制算法来进行GC，避 产生免内存碎片。

GC过程：初始标记、并发标记、最终标记、筛选回收。

其中，初始标记和最终标记会发生stop the world

优点：采用标记整理法和复制算法,不会产生内存碎片，

缺点：需要记录逻辑上新生代与老年代之间的引用关系，俗称记忆表，会对内存和 CPU有额外的消耗。

1. **CMS收集器和G1收集器的区别：**
2. CMS的堆内存划分遵循传统的新生代和老年代，G1收集器突破创新，将堆内 存划分为多块相同大小且地址连续的内存块，叫做region
3. CMS收集器只作用于老年代，需要与ParNew收集器一起使用，而G1收集器 直接作用于新生代和老年代。
4. CMS收集器采用标记清除算法，会产生内存碎片，G1采用标记整理算法+复制 算法，不会产生内存碎片。
5. G1收集器能预测GC的停顿时间，而CMS不能。

### 5.##### 讲述JVM中一次完整GC流程 #####

1.先讲下JAVA堆内存的划分：

堆内存分为 **新生代 和老年代**（默认比例1:2），新生代又分为 **Eden区和Servival区**（默认比例8：1）

2.对于大对象，比如长字符串，直接进入老年代

3.新生代Eden区存放new出来的新对象，当Eden区对象过多时，会触发MinorGC，然后将存活下来的对象从Eden区转移到Servivor区，然后清空Eden区。

4.当Servival区对象过多以及对象长期存活达到一定阈值时，会将Servivor区对象转移到老年代（但这时并不会发生GC）

5.当老年代对象太多了，会先发生MajorGC,对老年代对象进行垃圾回收

6.当整个对空间不够用时，会触发FullGC，对新生代和老年代都进行垃圾回收

**衍生问题：为啥要将堆内存进行分代？**

**答：**为了提高GC效率，因为不分代的话，每次GC都要对堆中所有的对象进行扫描，

而分代后，根据不同的场景找对应分代区的对象进行GC即可，效率明显更高，而且GC会伴随STW，GC效率高，意味着STW的时间就短，对业务程序的影响就越少。

### 6.##### JAVA对象在内存中的存储布局 #####

**对象头：**

Markword部分：具体存储了HashCode、GC分代年龄、锁状态标志。

类型指针部分：该指针指向了方法区中的这个类的元信息

**实例数据：**

在堆中存储的对象实例数据

**对齐填充：**

为了满足8字节整数倍的内存对齐而额外添加的占位符数据

### ##### 类加载过程/生命周期 #####

**加载、验证、准备、解析、初始化、使用、卸载**

**1.加载：**通过classLoader的双亲委派模型加载.class文件到内存。

**2.验证：**检查字节码文件是否符合JAVA规范。

**3.准备：**对类的静态变量分配内存并赋默认值。

**4.解析：**将常量池中的符号引用替换成可直接引用的内存地址。

**5.初始化：**不是实例化，而是对类的静态变量赋初始值、

执行类的static静态代码块，如果有父类则会先执行父类的初始化（父类的静态变 量、static块）。

**6.使用：**执行类的实例化，执行构造函数，如果有父类，会先执行父类的构造函数。

**7.卸载：**程序运行完毕后销毁内存中的对象。

### ##### 类加载器和双亲委派模型 #####

JVM自带的类加载器：**启动类加载器**、**扩展类加载器**、**应用程序类加载器，**

此外，还有用户的**自定义类加载器，这几个加载器依次是父子类关系。**

启动类加载器，加载jdk核心类；

扩展类加载器，加载扩展模块的类，

应用程序类加载器，加载应用程序配置的classPath路径下的类，

自定义类加载器，加载自定义的类。

**双亲委派模型**：是classloader类加载器加载类的默认方式，

是指：一个类加载器收到加载某个类的请求时，它首先不会自己去尝试加载这个类，而是将请求委派给父类加载器去完成，而父类加载器又会委托给父类的父类，直到没有父类为止，即到启动类加载器为止。当父类加载器没有找到需要加载的类时，会转头交给子类加载器去加载。

**为何需要双亲委派模型？**

避免同名类重复加载，防止核心类被篡改。

**如何打破双亲委派模型？**

继承ClassLoader类并重写loadClass和findClass方法。