JUC

1 如何创建线程

Java提供了多种方式来创建和管理线程，最常见的方式一共有四种

第一种是**通过继承Thread类并重写其run()方法**来创建线程。在run()方法中定义线程需要执行的任务逻辑。然后，创建该类的实例，调用start()方法启动线程。这种方法简单直观，但Java不支持多继承，因此限制了类的扩展性。

第二种是**实现Runnable接口并将其传递给Thread构造器**来创建线程。Runnable是一个函数式接口，其中的run()方法定义了任务逻辑。这种方式更加灵活，因为它不占用类的继承关系，同时可以更好地支持资源共享，可以让多个线程共享同一个Runnable实例。

第三种是**通过实现Callable接口来创建有返回值的线程。**Callable接口类似于Runnable，但它可以返回结果并抛出异常。Callable的call()方法需要通过FutureTask包装后传递给Thread构造器。通过Future对象可以获取线程执行结果或捕获异常。这种方式适合于需要获取线程执行结果的场景。

第四种是**通过Executor框架创建线程池来管理线程。**Executor框架提供了更高级的线程管理功能，例如线程复用、任务调度等。通过execute()或submit()方法提交任务，避免频繁创建和销毁线程的开销。广泛用于需要高效管理大量线程的场景。

2 创建线程的本质

在 Java 中，线程的启动本质上是通过调用 Thread 类的 start() 方法完成的。start() 方法会通知 JVM 启动一个新的线程，并在这个新线程中执行 run() 方法中的代码逻辑。因此，无论你使用哪种方式创建线程，最终都会归结到调用 new Thread().start() 来真正启动线程。

换句话说，new Thread().start() 是启动线程的唯一入口，而其他方式（如继承 Thread、实现 Runnable 或 Callable、使用 Executor 框架）都是对 Thread 类的封装或扩展，目的是为了提供更灵活的编程模型。

3 说说线程的生命周期

在java中，线程的生命周期可分为六个状态：新建状态、运行状态、阻塞状态、无限等待状态、超时等待状态和终止状态

线程在运行的整个生命周期的任意一个时刻，只可能处于其中一个状态。

第一个是**新建状态**，当一个线程对象被创建时，它处于新建状态。

第二个是**运行状态**，当调用线程的start()方法后进入运行状态，可能正在运行(Running)或者等待CPU调度(Ready)

第三个**阻塞状态**，当线程试图获取一个锁（例如进入同步代码块或方法），但该锁正在被其它线程占用时，会进入阻塞状态。

第四个是**无限等待状态**，当线程调用了某些特定的方法（如 Object.wait()、Thread.join() 或 LockSupport.park()），它会进入等待状态。在这种状态下，线程会无限期地等待，直到其他线程显式地唤醒它（例如通过 notify() 或 notifyAll() 方法）。等待状态通常用于线程间的协作。

第五个是**超时等待状态**，当线程调用了Thread.sleep()、Object.wait(timeout)等，会进入超时等待状态，在指定时间内自动恢复为运行状态

第六个是**终止状态**，当线程的 run() 方法执行完毕，或者由于未捕获的异常导致线程提前退出时，它进入终止状态。

4 什么是线程上下文切换

线程上下文切换是多线程编程中的一个概念，它直接影响程序的性能和效率。

首先什么是线程上下文切换，它是指当CPU从一个线程切换到另一个线程时，操作系统需要保存当前线程的执行状态，并加载下一个线程的执行状态，以便它们能够正确地继续运行。**执行状态主要包括：寄存器状态、程序计数器、栈信息、线程的优先级等。**

然后是线程上下文切换的发生时机，通常有四种情况会发生线程上下文切换。

第一种是**时间片耗尽**，操作系统为每个线程分配了一个时间片，当线程的时间片用完后，操作系统会强制切换到其它线程，这是为了保证多个线程能够公平地共享CPU资源

第二种是**线程主动让出CPU**，当线程调用了某些方法，如Thread.sleep()、Object.wait()等，会使线程主动让出CPU，导致上下文切换。

第三种是**调用了阻塞类型的系统中断**，比如：线程执行I/O操作时，由于I/O操作通常需要等待外部资源，线程会被挂起，会触发上下文切换。

第四种是**被终止或结束运行**

接着是线程上下文切换的过程，分为四步

第一步是**保存当前线程的上下文**，将当前线程的寄存器状态、程序计数器、栈信息保存到内存中

第二步是**根据线程调度算法**，如：时间片轮转、优先级调度等，选择下一个要运行的线程

第三步是**加载下一个线程的上下文**，从内存中恢复所选线程的寄存器状态、程序计数器和栈信息。

第四步是**CPU开始执行**被加载线程的代码

最后是线程上下文切换所带来的影响。线程上下文切换虽然能够实现多任务并发执行，但它也会带来 CPU 时间消耗、缓存失效以及资源竞争等问题。为了减少线程上下文切换带来的性能损失，可以采取减少线程数量、使用无锁数据结构等方式进行优化。

5 并发和并行的区别

并发和并行是多线程编程中的两个核心概念，它们描述了任务执行的不同方式。它们有着本质的区别。

首先说一下**什么是并发**，并发指的是多个任务在同一时间段内交替执行的能力。换句话说，并发并不一定要求任务同时进行，而是通过快速切换任务来实现“看起来同时运行”的效果。

然后再说一下**什么是并行**，并行指的是多个任务在同一时刻真正同时执行的能力。并行通常需要多核 CPU 的支持，每个核心独立处理一个任务，从而实现真正的并行计算。

6 同步和异步的区别

同步和异步是编程中两种常见的任务执行模式。

首先说一下**什么是同步**，同步指的是任务按照顺序依次执行的方式。在这种模式下，调用者会阻塞等待任务完成并返回结果后，才会继续执行后续的操作。

然后再**说一下什么是异步**，异步指的是任务无需等待立即返回，调用方可以继续执行其他操作，而任务的结果会在稍后通过如回调函数、事件通知或 Future 对象等机制传递给调用方。

7 线程池的七大参数

线程池是 Java 并发编程中的重要工具，它通过复用线程来减少线程创建和销毁的开销，从而提高系统的性能和稳定性。在 Java 中，线程池的核心实现类是 ThreadPoolExecutor，它提供了七个重要的参数来配置线程池的行为。

第一个是核心线程数，它是指线程池中始终保持存活的线程数量，即使这些线程处于空闲状态。

第二个是最大线程数，它是指线程池中允许的最大线程数量。当任务队列已满且当前线程数小于最大线程数时，线程池会继续创建新线程来处理任务。如果线程数已经达到最大值，则任务会被拒绝。

第三个是线程空闲时间，它是指非核心线程在空闲状态下保持存活的时间。

第四个是时间单位

第五个是任务队列，它是一个阻塞队列，用于存放等待执行的任务。当线程池中的线程数达到核心线程数时，新提交的任务会被放入任务队列中等待执行。常见的队列类型包括：

- ArrayBlockingQueue：有界队列，适用于控制资源使用。
- LinkedBlockingQueue：无界队列，适用于任务量较大的场景。
- SynchronousQueue：不存储任务的队列，适用于直接传递任务给线程的场景。

第六个是线程工厂（threadFactory），它用于创建线程池中的线程。通过自定义线程工厂，可以为线程设置名称、优先级或其他属性，便于调试和管理。例如，使用 Executors.defaultThreadFactory() 创建默认线程工厂。

第七个是拒绝策略（handler），它用于处理当线程池无法接受新任务时的情况（例如线程数达到最大值且任务队列已满）。常见的拒绝策略包括：

AbortPolicy：抛出异常，拒绝任务。

CallerRunsPolicy：由调用线程执行任务。

DiscardPolicy：直接丢弃任务。

DiscardOldestPolicy：丢弃队列中最旧的任务，并尝试重新提交新任务。

8 线程池四大拒绝策略

线程池是 Java 并发编程中用于管理线程的重要工具，而拒绝策略则是线程池在资源耗尽时处理新任务的一种机制。当线程池中的线程数达到最大值且任务队列已满时，线程池会根据配置的拒绝策略来决定如何处理无法接受的新任务。接下来我会详细讲述线程池的四大拒绝策略及其特点。

首先是 **AbortPolicy**（中止策略），它线程池的默认拒绝策略。当线程池无法接受新任务时，它会直接抛出 RejectedExecutionException 异常，终止任务的提交。这种策略适用于对任务执行有严格要求的场景，例如不允许任务丢失的情况。

然后是 **CallerRunsPolicy**（调用者运行策略），它会将被拒绝的任务回退给提交任务的线程执行。也就是说，任务不会被丢弃，而是由调用线程（通常是主线程）直接运行该任务。这种策略可以减缓任务提交的速度，从而缓解线程池的压力，但可能会导致调用线程阻塞。在这种情况下，主线程会承担部分任务的执行工作。

接下来是 **DiscardPolicy**（丢弃策略），它会直接丢弃无法处理的任务，并且不会抛出任何异常。这种策略适用于对任务执行要求不高的场景，例如允许部分任务丢失的情况。在这种情况下，被拒绝的任务会被静默丢弃，调用方不会收到任何通知。

最后是 **DiscardOldestPolicy**（丢弃最旧任务策略），它会丢弃任务队列中最旧的任务（即等待时间最长的任务），然后尝试重新提交当前任务。这种策略可以确保较新的任务有机会被执行，但可能会导致某些任务被重复提交或丢失。在这种情况下，队列中最旧的任务会被移除，为新任务腾出空间。

9 ConcurrentHashMap的原理

ConcurrentHashMap 是 Java 中常用的并发容器，它的实现从 JDK 1.7 到 JDK 1.8 发生了较大的变化，JDK 1.7 通过分段锁提高并发性能，但锁的粒度较粗，而 JDK 1.8 通过 CAS 和红黑树优化，实现了更高的并发性和查询效率，简化了实现逻辑。

首先说一下JDK1.7版本中的实现：

ConcurrentHashMap 的实现方式采用了 数组 + Segment + 分段锁 的方式。Segment 是一种特殊的分段锁，继承了 ReentrantLock，每个 Segment 里面包含一个 Entry 数组，Entry 数组中的元素以链表形式存储。

然后再说一下 **JDK 1.8**版本：

第一，JDK 1.8 摒弃了分段锁的实现方式，改用 synchronized + CAS，更加高效。

第二，它采用 CAS 操作（Compare-And-Swap）保证并发安全，必要时使用 Synchronized 来解决并发冲突。

第三，它采用了 数组 + 链表 + 红黑树 的数据结构。链表长度超过阈值（默认 8）时，会转化为红黑树，从而优化查询性能。

第四，它锁的颗粒度细化到桶（Node），并且 value 和 next 使用 volatile 修饰，保证并发的可见性。

第五，从查询时间复杂度来说，使用链表时为 O(n)，使用红黑树后降为 O(logN)。

第六，从并发性能来讲，并发粒度与数组长度相关，每个桶可以独立加锁，支持更高的并发度。

10 什么是线程死锁

线程死锁是指两个或多个线程在执行过程中，因为争夺资源而相互等待对方释放资源，从而导致所有相关线程都无法继续执行的情况。

死锁的产生条件主要有以下四个必要条件：

互斥

持有并等待

不可剥夺

循环等待

11 如何预防和避免线程死锁

如果我们想要**避免死锁的发生**，只要破坏其中任何一个死锁的产生条件。

第一种是破坏互斥条件，我们可以尽量减少对共享资源的独占性访问。 使用无锁数据结构来替代传统的同步机制，比如：ConcurrentHashMap、AtomicInteger 等。对于只读资源，可以通过复制或缓存的方式避免竞争。

第二种是破坏占有且等待条件，我们可以要求线程在开始执行前一次性获取所有需要的资源。如果无法获取所有资源，则释放已占有的资源并稍后重试。这种方法被称为“一次性申请所有资源”，但需要注意的是，它可能会增加资源的竞争压力。

第三种是破坏不可剥夺条件，我们可以允许系统强制剥夺线程占有的资源。这种方法通常用于操作系统层面，但在 Java 中并不常见，因为强制剥夺资源可能会导致数据不一致或复杂的恢复逻辑。

第四种是破坏循环等待条件，我们可以为资源分配一个全局的顺序编号，并要求线程按照固定的顺序申请资源。

12 Synchronized底层原理了解吗

synchronized 用于保证多线程环境下的数据一致性。接下来我会详细讲述 synchronized 的定义和底层实现。

首先说一下**什么是synchronized，**它是一种内置的锁机制，它可以作用于方法或代码块，用于控制多个线程对共享资源的访问。当一个线程进入 synchronized 保护的代码区域时，它会尝试获取锁；如果锁已被其他线程占用，则当前线程会被阻塞，直到锁被释放。锁的持有者在退出同步代码块或方法时会自动释放锁，从而允许其他线程继续执行。

JVM 通过 `monitorenter` 指令加锁，`monitorexit` 指令解锁。每个对象有一个 `Monitor`，其中包含锁的持有线程、等待队列等信息。线程执行 `monitorenter` 时，若 `Monitor` 未被占用，则获取锁并将进入计数 `count+1`，成为锁的所有者；若已被占用，则线程进入 `BLOCKED` 状态，等待锁释放。线程执行 `monitorexit` 时，计数 `count-1`，当 `count == 0` 时释放锁，其他等待线程可尝试获取 `Monitor`。

修饰方式	锁对象	适用场景
修饰实例方法	当前实例对象（this）	同步控制单个实例的方法调用。

修饰方式	锁对象	适用场景
修饰静态方法	类的 Class 对象（Counter.class）	同步控制所有实例共享的静态资源。
修饰代码块	显式指定的锁对象（如 this 或自定义对象）	灵活控制特定代码段的同步，减少锁的范围，提高性能。

13 Synchronized和ReentrantLock的区别

synchronized 和 ReentrantLock 是 Java 中实现线程同步的两种主要方式，它们都能保证多线程环境下的数据一致性，接下来我会详细讲述两者在基本概念、功能特性、性能表现以及锁的释放与异常处理上的区别。

第一个是**基本概念**上的区别，synchronized 是 Java 的内置关键字，它是隐式的，通过 JVM 提供的监视器锁机制实现同步，使用简单，无需手动管理锁的获取和释放；而 ReentrantLock 是 java.util.concurrent.locks 包中的一个类，它是是显式的，提供了更灵活的锁机制，需要开发者手动调用 lock() 和 unlock() 方法来控制锁的生命周期。

第二个是**功能特性**上的区别，ReentrantLock 提供了比 synchronized 更丰富的功能，比如：ReentrantLock 支持在等待锁的过程中响应中断，而 synchronized 不支持中断；还有ReentrantLock 提供了 tryLock() 方法，允许线程尝试获取锁并在指定时间内返回结果，而 synchronized 必须一直等待锁释放。

第三个是**性能**上的区别，synchronized 和 ReentrantLock 在不同场景下各有优势。

对于低竞争场景，由于synchronized 经过多次优化（如偏向锁、轻量级锁），一般与 ReentrantLock 相当甚至更好。

对于高竞争场景，ReentrantLock 提供了更多的灵活性（如公平锁、可中断锁等），更适合复杂需求。

第四个是**锁的释放与异常处理上**的区别，synchronized 在退出同步代码块时会自动释放锁，即使发生异常也不会导致死锁；而ReentrantLock 需要开发者手动调用 unlock() 方法释放锁，因此必须在 finally 块中确保锁的释放，否则可能导致死锁。

14 什么是乐观锁

乐观锁是一种并发控制机制

首先说一下**什么是乐观锁**，它是一种基于“无锁”思想的并发控制机制。它假设多线程操作之间很少发生冲突，因此在读取数据时不会加锁，而是通过某种机制（如版本号或时间戳）来检测数据是否被其他线程修改过。如果检测到数据未被修改，则提交更新；如果检测到数据已被修改，则根据策略进行处理（如重试或抛出异常）。

接下来说一下**乐观锁的实现方式**，乐观锁的实现通常依赖于以下两种机制：

一种是版本号机制：为数据添加一个版本号字段，每次更新时递增版本号，并在更新时验证版本号是否匹配。

另一种是CAS 操作：使用比较并交换（Compare-And-Swap）指令，直接在硬件层面实现无锁操作。CAS 操作包含内存位置（V）、预期值（A）和新值（B）这三个参数。只有当内存位置的值等于预期值时，才会将内存位置的值更新为新值。

然后再说一下**乐观锁的特点**，一共有三个。

第一个是无锁设计，乐观锁不依赖传统的锁机制，减少了线程阻塞和上下文切换的开销。

第二个是性能好，在低冲突场景下，乐观锁的性能优于悲观锁，因为它避免了锁的竞争。

第三个是支持冲突检测，乐观锁通过版本号或 CAS 操作检测冲突，但需要开发者显式处理冲突（如重试或回滚），这可能会增加代码的复杂性。

最后说一下**乐观锁的适用场景**，一般有三种场景比较适合。

第一种是读多写少的场景，例如缓存系统、统计计数器等，读操作远多于写操作，冲突概率较低。

第二种是在分布式环境中，乐观锁可以通过版本号或时间戳实现跨节点的数据一致性。

第三种是高并发环境，在高并发场景下，乐观锁可以减少锁的竞争，从而提高系统的吞吐量。

需要注意的是，乐观锁并不适合写操作频繁或冲突概率较高的场景，因为频繁的冲突会导致大量的重试操作，反而降低性能。

JVM

1 对象创建的过程了解吗

对象创建的过程是JVM中一个非常重要的环节，它主要分为五步：

第一步是**进行类加载检查**，当程序执行到new指令时，JVM会先检查对应的类是否已经被加载、解析和初始化过。如果类尚未加载，JVM会按照类加载机制（加载、验证、准备、解析、初始化）完成类的加载过程。这一步确保了类的元信息（如字段、方法等）已经准备好，为后续的对象创建奠定基础

第二步是**进行内存的分配**，JVM会为新对象分配内存空间。对象所需的内存大小在类加载完成后就可以确定，因此分配内存的过程就是从堆中划分一块连续的空间，主要有两种方式：

一种是通过指针碰撞，如果堆中的内存是规整的（已使用和空闲区域之间有明确分界），JVM 可以通过移动指针来分配内存。另一种是通过空闲列表，如果堆中的内存是碎片化的，JVM 会维护一个空闲列表，记录可用的内存块，并从中分配合适的区域。

此外，为了保证多线程环境下的安全性，JVM 还会采用两种策略避免内存分配冲突，一种是通过 CAS 操作尝试更新分配指针，如果失败则重试；另一种是每个线程在堆中预先分配一小块专属区域，避免线程间的竞争。

第三步是将**零值初始化**，JVM 会对分配的内存空间进行初始化，将其所有字段设置为零值（如 int 为 0，boolean 为 false，引用类型为 null）。这一步确保了对象的实例字段在未显式赋值前有一个默认值，从而避免未初始化的变量被访问。

第四步是**设置对象头**，其中包含Mark Word、Klass Pointer和数组长度。Mark Word 用于存储对象的哈希码、GC 分代年龄、锁状态标志等信息。Klass Pointer 指向对象所属类的元数据（即 Person.class 的地址）。

第五步是**执行构造方法**，用 方法完成对象的初始化。构造方法会根据代码逻辑对对象的字段进行赋值，并调用父类的构造方法完成继承链的初始化。这一步完成后，对象才真正可用。

2 类载入过程JVM会做什么

类的加载过程确保了类在运行时能够被正确地使用，可以分为五个阶段：加载、验证、准备、解析、初始化。

第一个是加载阶段，在这个阶段，JVM会完成三件事，

- 首先是用过类的全限定名获取定义此类的二进制字节流
- 然后是将字节流所代表的静态存储结构转化为方法区的运行时数据结构，即将类的元信息（如字段、方法、父类等）存储到方法区中。
- 最后是在堆中生成一个代表该类的 java.lang.Class 对象，这个对象作为程序访问该类的入口点，所有的反射操作都通过这个对象进行。

第二个是**验证**阶段，在这个阶段，JVM 会对加载的字节码进行校验，以确保其符合 Java 虚拟机规范，并且不会危害虚拟机的安全，一般会验证四样东西。

- 首先是文件格式，检查字节码文件是否符合 Class 文件格式规范。
- 然后是元数据，检查类的元信息是否符合语法规则，例如父类是否存在、是否继承了 final 类等。
- 其次是字节码，分析字节码指令，确保其不会执行非法操作（如类型转换错误、越界访问等）。
- 最后是符号引用，检查符号引用能否正确解析为直接引用，例如检查类、字段、方法是否存在并且可访问。

第三个是**准备**阶段，JVM 会为类的静态变量分配内存，并设置默认初始值（零值）。这个阶段并不会执行任何 Java 代码，也不会为实例变量分配内存（实例变量是在对象创建时分配的）。例如，如果类中有一个静态变量 static int value = 123;，在这个阶段，value 会被初始化为 0，而不是 123（赋值操作会在初始化阶段完成）。

第四个是**解析**阶段，JVM 会将类中的符号引用替换为直接引用。符号引用是以一组符号描述所引用的目标，例如类的全限定名、字段的名称和描述符等。直接引用是可以直接定位到目标的指针、句柄或偏移量。解析的对象包括类或接口、字段、方法、方法类型、方法句柄和调用点限定符等。

第五个是**初始化**阶段，在此阶段，JVM 会执行类的初始化代码，包括静态变量赋值和静态代码块的执行。这是类加载过程的最后一个阶段，也是唯一一个会执行用户代码的阶段。初始化的顺序遵循“父类优先”的原则，即先初始化父类，再初始化子类。

3 什么是双亲委派模型

首先说一下**什么是双亲委派模型**，它其实是一种类加载机制，其规定了当一个类加载器收到类加载请求时，不会立即尝试自己去加载这个类，而是先将请求委托给父类加载器完成。只有当父类加载器无法加载该类（例如在父类的搜索范围内找不到对应的类）时，子类加载器才会尝试自己加载。这种机制确保了类的加载过程具有层次性，并且优先使用高层级的类加载器来加载核心类库。

接下来说一下**类加载器的层次结构**，主要分为四层，

第一层是启动类加载器（Bootstrap ClassLoader），它负责加载 JVM 核心类库（如 rt.jar 中的类），位于最顶层，通常由本地代码实现。

第二层是扩展类加载器（Extension ClassLoader），它负责加载 $JAVA_HOME/lib/ext 目录下的扩展类库。

第三层是应用程序类加载器（Application ClassLoader），它负责加载用户类路径（ClassPath）上的类，也称为系统类加载器。

第四层是自定义类加载器，开发者可以通过继承 ClassLoader 类实现自己的类加载器，用于加载特定需求的类。

这些类加载器之间形成了一个树状的层次结构，每个类加载器都有一个父加载器。

最后说一下**双亲委派模型的工作流程**，主要分为四步，

第一步是检查缓存，当前类加载器会先检查是否已经加载过目标类，如果已加载，则直接返回对应的 Class 对象。

第二步是委派父加载器，如果没有加载过，当前类加载器会将加载请求委派给父加载器处理。

第三步是递归向上，父加载器继续将请求委派给它的父加载器，直到到达 Bootstrap ClassLoader。

第四步是尝试加载，如果父加载器无法加载目标类，则子加载器会尝试自己加载。

4 JVM的内存区域

JVM 的内存区域可以分为线程共享和线程私有两部分，每个部分都有明确的职责和作用，保障 Java 程序的高效运行。JDK1.7 和 1.8 时内存结构略有不同，接下来我先讲解一下 JDK1.7 时 JVM 的内存结构，然后再说一下 JDK1.8 时发生了哪些变动。

首先是**线程共享**的部分，一共有两个，

一个是堆（Heap），所有对象实例和数组都在这里分配内存，垃圾回收器（GC）会管理其中的对象回收。堆中还包含了字符串常量池（String Constant Pool），用于存储字符串字面量和常量。

另一个是方法区（Method Area）：用于存储类元信息、静态变量、常量、方法字节码等。其中运行时常量池（Runtime Constant Pool）是方法区的一部分，用于存储编译期生成的各种字面量和符号引用。

然后是**线程私有**的部分，一共有三个，

第一个是虚拟机栈（VM Stack），每个线程启动时都会创建一个虚拟机栈，它存储方法调用过程中产生的栈帧，包括局部变量、操作数栈、方法返回地址等，每个方法调用都会创建一个新的栈帧，方法执行结束后栈帧出栈。

第二个是本地方法栈（Native Method Stack），专门用于存储本地方法（Native Method）的调用信息，与虚拟机栈类似，但用于 JNI（Java Native Interface）调用。

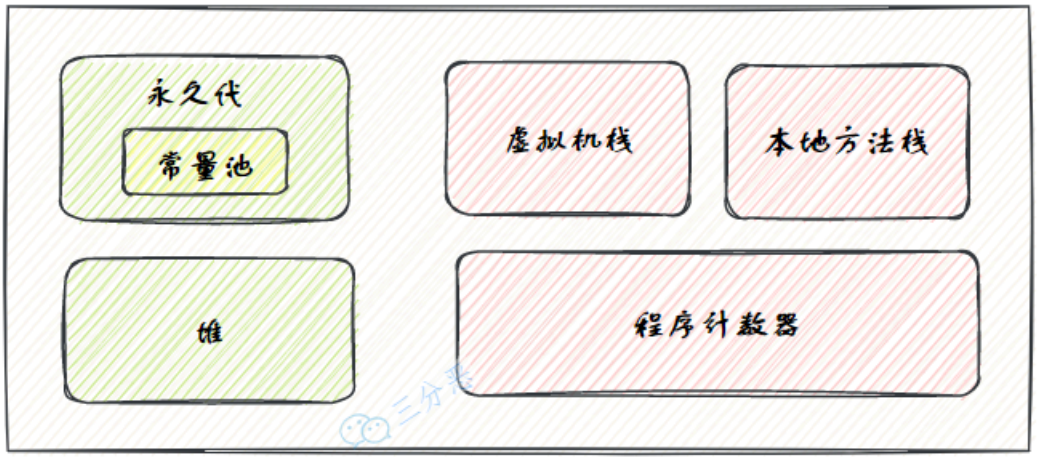
第三个是程序计数器（Program Counter Register），记录当前线程正在执行的字节码指令地址。它是 JVM 运行时最小的内存区域，每个线程都有一个独立的程序计数器。

最后是**本地内存**，

里面包含直接内存（Direct Memory），由 NIO（New Input/Output）直接分配，不受 JVM 堆的管理，通常用于高性能数据传输，如缓冲区（Buffer）。这个内存结构保证了 JVM 在执行 Java 代码时能够高效管理对象、执行方法调用，并支持多线程并发。

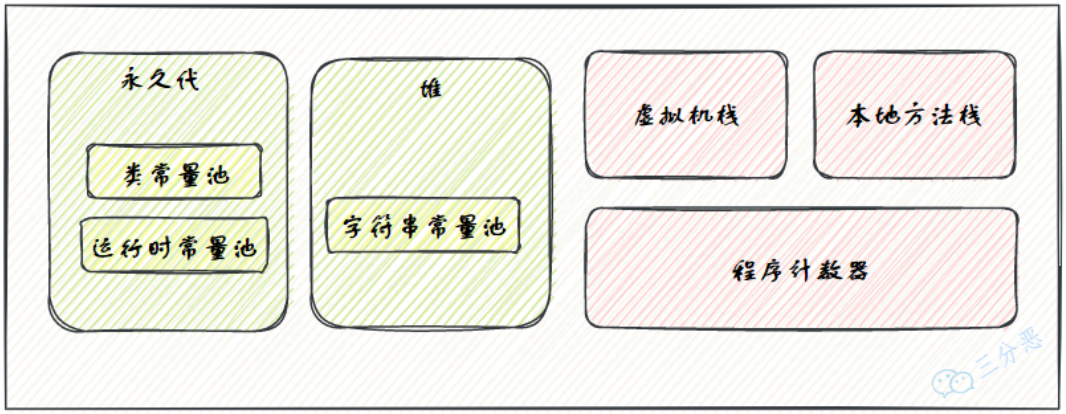
- JDK1.6使用永久代实现方法区

JDK1.6

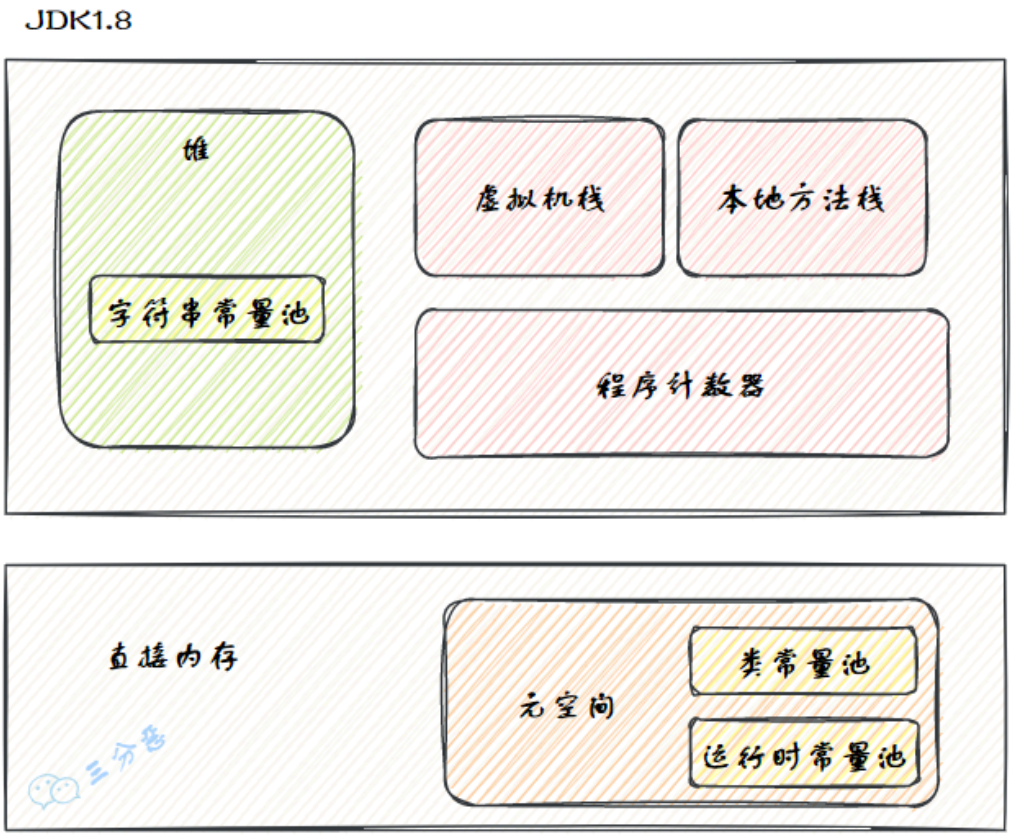


- JDK1.7将字符串常量池、静态变量存放在堆上

JDK1.7



- JDK1.8彻底干掉了永久代，在直接内存中划出一块区域作为**元空间**，运行时常量池、类常量池都移动到元空间



5 垃圾回收器

Java 的垃圾回收机制通过标记垃圾对象和回收无用内存，提升内存利用率，降低程序停顿时间。垃圾回收器是具体实现算法的工具，最常用的两种收集器是 CMS 和 G1 ，分别适用于不同的场景，接下来我会分别进行讲述。

第一个是 **CMS 收集器**，CMS（Concurrent Mark Sweep）是以最小化停顿时间为目标的垃圾收集器，适用于需要高响应的应用场景（如 Web 应用）。其基于**"标记-清除算法"**，回收流程包括以下阶段：

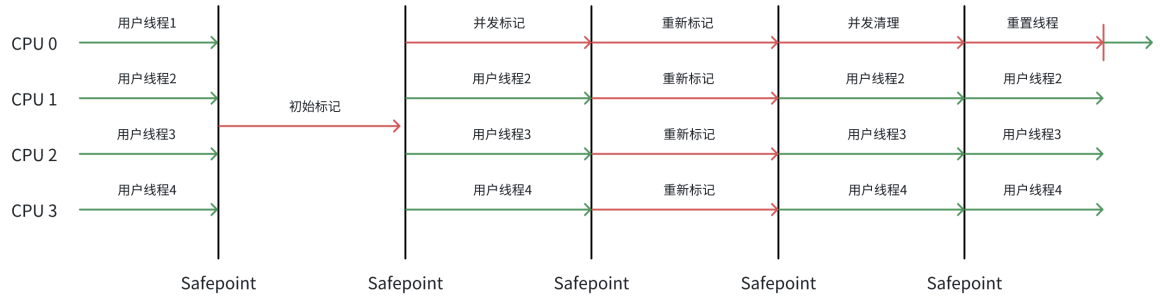
首先停止所有用户线程，启用一个GC线程进行**初始标记**（Stop The World），标记 GC Roots 能直接引用的对象，停顿时间短。

其次由用户线程和 GC 线程并发执行，进行**并发标记**，用户线程和 GC 线程并发执行，完成从 GC Roots 开始的对象引用分析。

然后，启动多个GC 线程进行**重新标记（Stop The World）**，修正并发标记期间用户线程对对象引用的变动，停顿时间稍长但可控。

最后，启动多个用户线程和一个GC 线程，进行**并发清除**，清理不可达对象，清理完成后把GC线程进行重置。

CMS 的优点是以响应时间优先，停顿时间短，但也有两个缺点，一个是由于CMS采用"标记-清除"，会导致内存碎片积累，另一个是由于在并发清理过程中仍有用户线程运行，可能生成新的垃圾对象，需在下次 GC 处理。



第二个是 **G1 收集器**，G1（Garbage-First）收集器以控制 GC 停顿时间为目标，兼具高吞吐量和低延迟性能，适用于大内存、多核环境。其基于**“标记-整理”和“标记-复制算法”**，整体上是**标记 + 整理**算法；两个区域之间**复制**算法，回收流程包括以下阶段：

首先，停止所有用户线程，启用一个GC线程进行**初始标记**（Stop The World），标记从 GC Roots 可达的对象，时间短。

其次，让用户线程和一个GC 线程并发工作，用GC 线程进行**并发标记**，分析整个堆中对象的存活情况。

然后，停止所有用户线程，让多个GC 线 程进行**最终标记**（Stop The World），修正并发标记阶段产生的引用变动，识别即将被回收的对象。

最后，让多个GC 线程进行筛选回收，根据收集时间预算，**优先回收回收价值最高的 Region**。回收完成后把GC线程进行重置。这是 G1 的核心优化，基于堆分区，将回收工作集中于垃圾最多的区域，避免全堆扫描。

G1 具有三个优点，

其一，将堆内存划分为多个 Region，可分别执行标记、回收，提升效率。

第二，采用“标记-整理”和“标记-复制”，实现内存紧凑化。

第三，方便控制停顿时间，通过后台维护的优先队列，动态选择高价值 Region，极大减少了全堆停顿的频率。

但G1缺点是：调优复杂，对硬件资源要求较高。