

破4! 《我想进大厂》之Java基础夺命连环16问

原创 科技缪缪 艾小仙 2020-11-12

收录于话题

#面试大全 29 #文章精选汇总 78

说好了面试系列已经完结了，结果发现还是真香，嗯，以为我发现我的Java基础都没写，所以这个就算作续集了，续集第一篇请各位收好。

等到你们收到这篇文章的时候，公众号读者数量就破4000了，可不是4万，就庆祝下，存稿都发出来了，下周又得肝了！

说说进程和线程的区别？

进程是程序的一次执行，是系统进行资源分配和调度的独立单位，他的作用是是程序能够并发执行提高资源利用率和吞吐率。

由于进程是资源分配和调度的基本单位，因为进程的创建、销毁、切换产生大量的时间和空间的开销，进程的数量不能太多，而线程是比进程更小的能独立运行的基本单位，他是进程的一个实体，可以减少程序并发执行时的时间和空间开销，使得操作系统具有更好的并发性。

线程基本不拥有系统资源，只有一些运行时必不可少的资源，比如程序计数器、寄存器和栈，进程则占有堆、栈。

知道synchronized原理吗？

synchronized是java提供的原子性内置锁，这种内置的并且使用者看不到的锁也被称为**监视器锁**，使用synchronized之后，会在编译之后在同步的代码块前后加上monitorenter和monitorexit字节码指令，他依赖操作系统底层互斥锁实现。他的作用主要就是实现原子性操作和解决共享变量的内存可见性问题。

执行monitorenter指令时会尝试获取对象锁，如果对象没有被锁定或者已经获得了锁，锁的计数器+1。此时其他竞争锁的线程则会进入等待队列中。

执行monitorexit指令时则会把计数器-1，当计数器值为0时，则锁释放，处于等待队列中的线程再继续竞争锁。

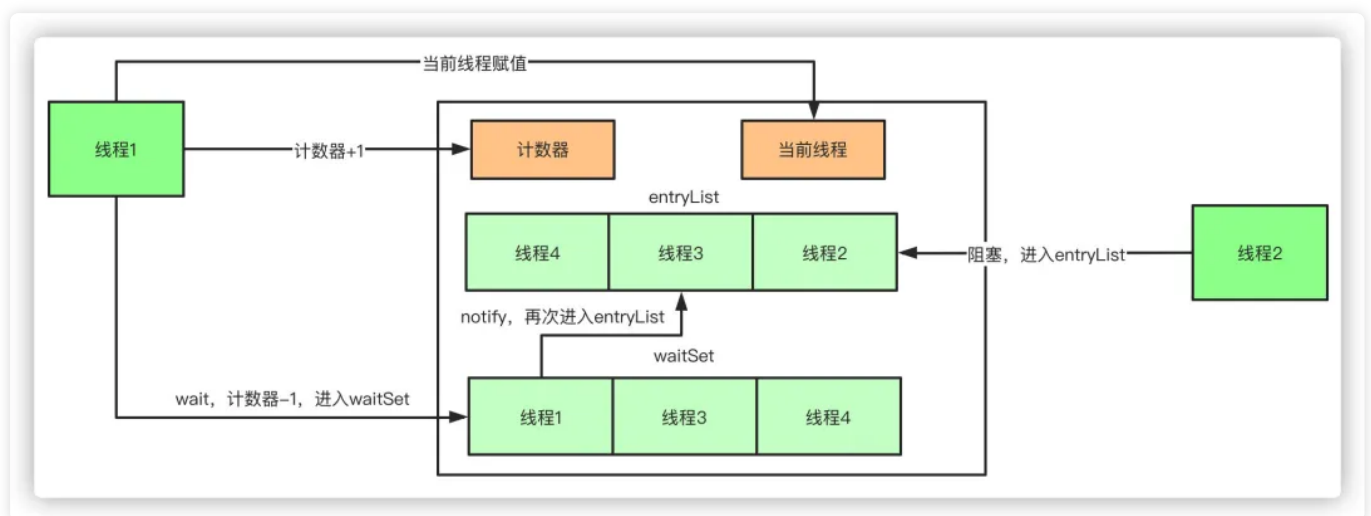
synchronized是排它锁，当一个线程获得锁之后，其他线程必须等待该线程释放锁后才能获得锁，而且由于Java中的线程和操作系统原生线程是一一对应的，线程被阻塞或者唤醒时会从用户态切换到内核态，这种转换非常消耗性能。

从内存语义来说，加锁的过程会清除工作内存中的共享变量，再从主内存读取，而释放锁的过程则是将工作内存中的共享变量写回主内存。

实际上大部分时候我认为说到monitorenter就行了，但是为了更清楚的描述，还是再具体一点。

如果再深入到源码来说，synchronized实际上有两个队列waitSet和entryList。

1. 当多个线程进入同步代码块时，首先进入entryList
2. 有一个线程获取到monitor锁后，就赋值给当前线程，并且计数器+1
3. 如果线程调用wait方法，将释放锁，当前线程置为null，计数器-1，同时进入waitSet等待被唤醒，调用notify或者notifyAll之后又会进入entryList竞争锁
4. 如果线程执行完毕，同样释放锁，计数器-1，当前线程置为null



那锁的优化机制了解吗？

从JDK1.6版本之后，synchronized本身也在不断优化锁的机制，有些情况下他并不会是一个很重量级的锁了。优化机制包括自适应锁、自旋锁、锁消除、锁粗化、轻量级锁和偏向锁。

锁的状态从低到高依次为**无锁->偏向锁->轻量级锁->重量级锁**，升级的过程就是从低到高，降级在一定条件也是有可能发生的。

自旋锁：由于大部分时候，锁被占用的时间很短，共享变量的锁定时间也很短，所有没有必要挂起线程，用户态和内核态的来回上下文切换严重影响性能。自旋的概念就是让线程执行一个忙循环，可以理解为就是啥也不干，防止从用户态转入内核态，自旋锁可以通过设置-XX:+UseSpining来开启，自旋的默认次数是10次，可以使用-XX:PreBlockSpin设置。

自适应锁：自适应锁就是自适应的自旋锁，自旋的时间不是固定时间，而是由前一次在同一个锁上的自旋时间和锁的持有者状态来决定。

锁消除：锁消除指的是JVM检测到一些同步的代码块，完全不存在数据竞争的场景，也就是不需要加锁，就会进行锁消除。

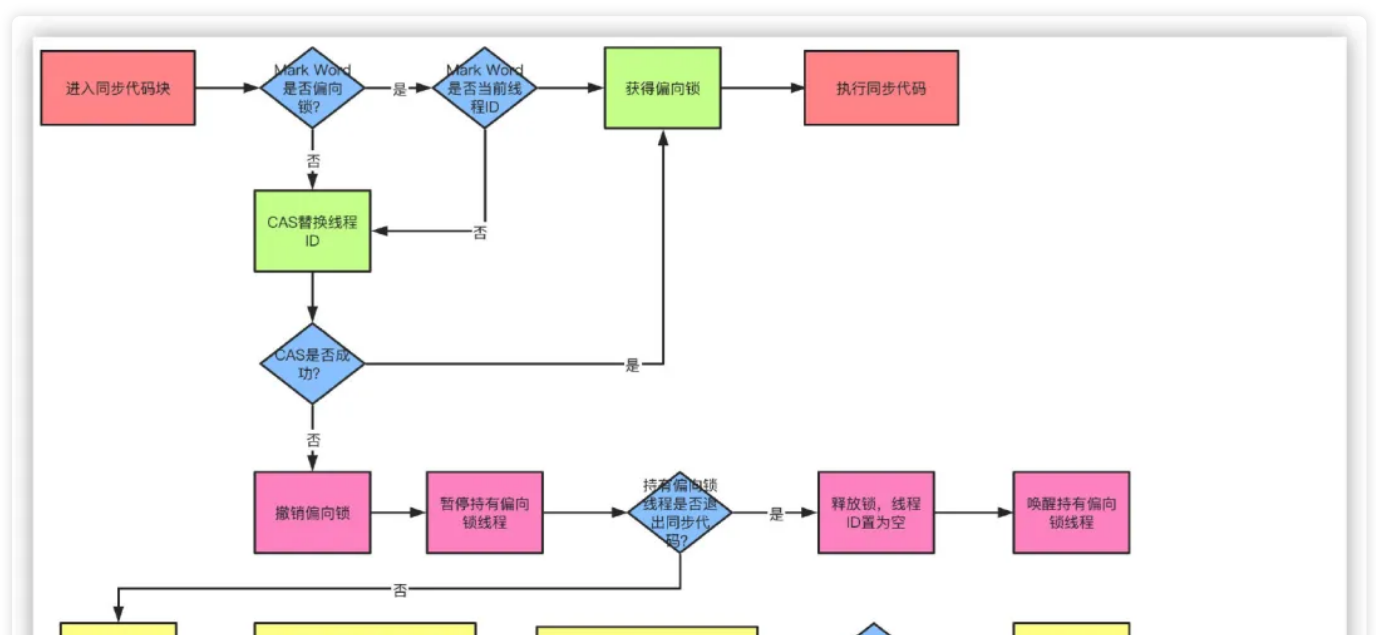
锁粗化：锁粗化指的是有很多操作都是对同一个对象进行加锁，就会把锁的同步范围扩展到整个操作序列之外。

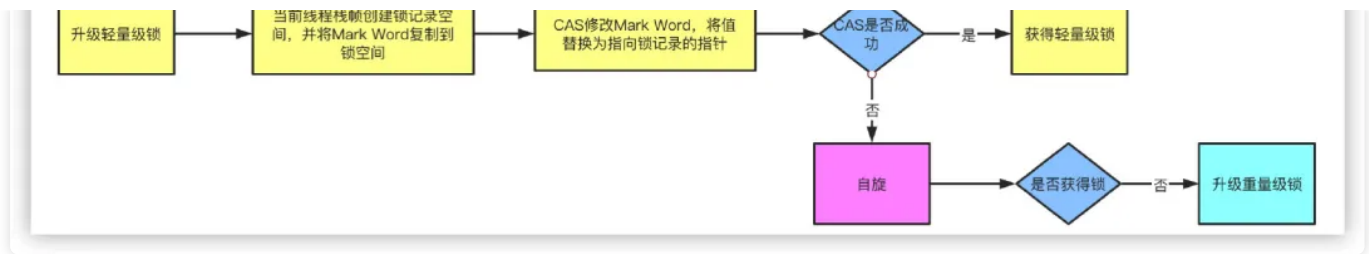
偏向锁：当线程访问同步块获取锁时，会在对象头和栈帧中的锁记录里存储偏向锁的线程ID，之后这个线程再次进入同步块时都不需要CAS来加锁和解锁了，偏向锁会永远偏向第一个获得锁的线程，如果后续没有其他线程获得过这个锁，持有锁的线程就永远不需要进行同步，反之，当有其他线程竞争偏向锁时，持有偏向锁的线程就会释放偏向锁。可以用过设置-XX:+UseBiasedLocking开启偏向锁。

轻量级锁：JVM的对象的对象头中包含有一些锁的标志位，代码进入同步块的时候，JVM将会使用CAS方式来尝试获取锁，如果更新成功则会把对象头中的状态位标记为轻量级锁，如果更新失败，当前线程就尝试自旋来获得锁。

整个锁升级的过程非常复杂，我尽力去除一些无用的环节，简单来描述整个升级的机制。

简单点说，偏向锁就是通过对象头的偏向线程ID来对比，甚至都不需要CAS了，而轻量级锁主要就是通过CAS修改对象头锁记录和自旋来实现，重量级锁则是除了拥有锁的线程其他全部阻塞。





那对象头具体都包含哪些内容？

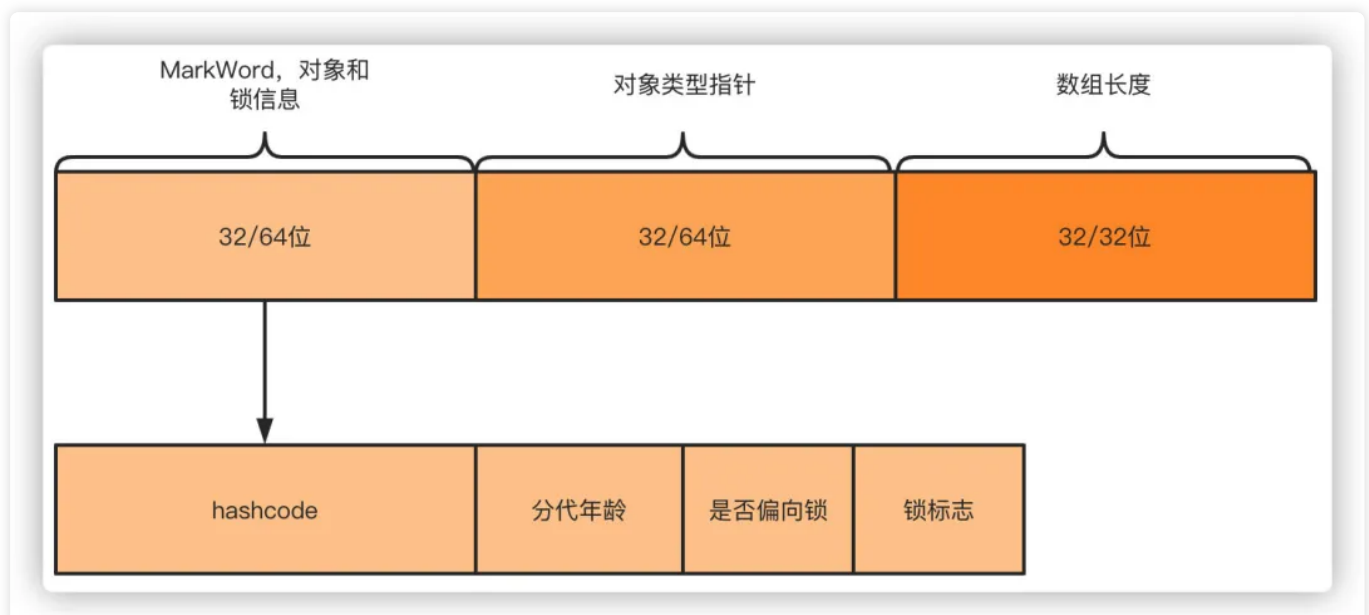
在我们常用的Hotspot虚拟机中，对象在内存中布局实际包含3个部分：

1. 对象头
2. 实例数据
3. 对齐填充

而对象头包含两部分内容，Mark Word中的内容会随着锁标志位而发生变化，所以只说存储结构就好了。

1. 对象自身运行时所需的数据，也被称为Mark Word，也就是用于轻量级锁和偏向锁的关键点。具体的内容包含对象的hashcode、分代年龄、轻量级锁指针、重量级锁指针、GC标记、偏向锁线程ID、偏向锁时间戳。
2. 存储类型指针，也就是指向类的元数据的指针，通过这个指针才能确定对象是属于哪个类的实例。

如果是数组的话，则还包含了数组的长度



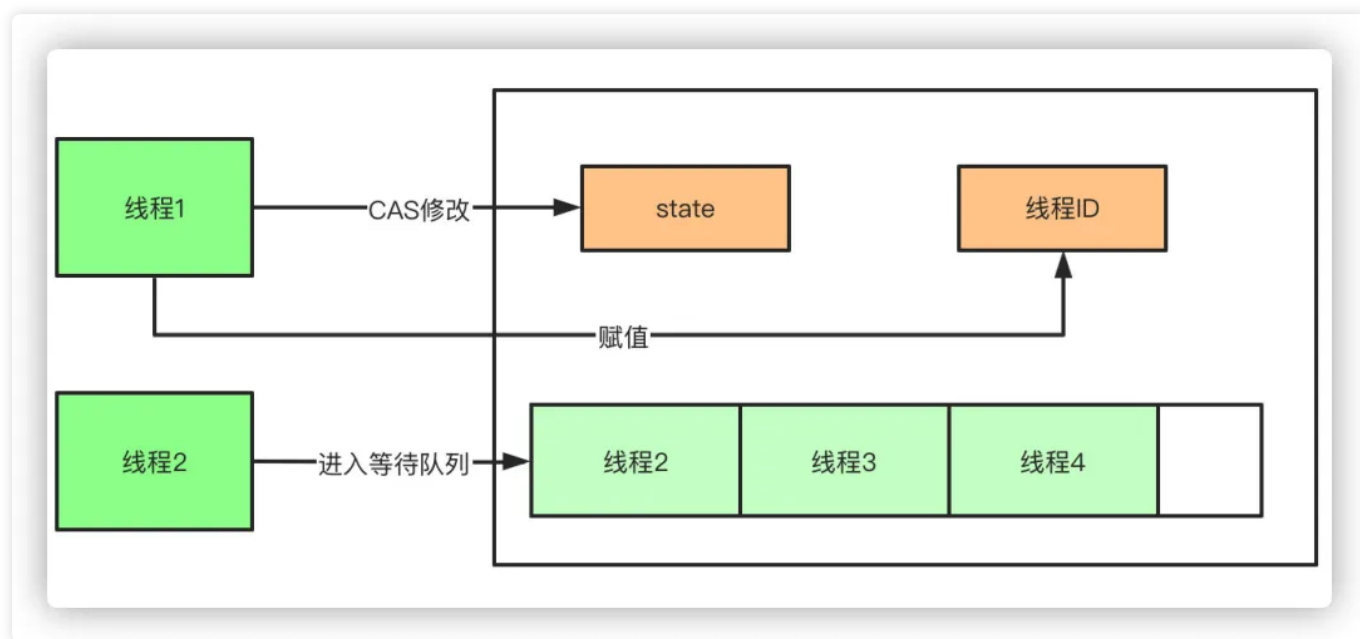
对于加锁，那再说下ReentrantLock原理？他和synchronized有什么区别？

相比于synchronized，ReentrantLock需要显式的获取锁和释放锁，相对现在基本都是用JDK7和JDK8的版本，ReentrantLock的效率和synchronized区别基本可以持平了。他们的主要区别有以下几点：

1. 等待可中断，当持有锁的线程长时间不释放锁的时候，等待中的线程可以选择放弃等待，转而处理其他的任务。
2. 公平锁：synchronized和ReentrantLock默认都是非公平锁，但是ReentrantLock可以通过构造函数传参改变。只不过使用公平锁的话会导致性能急剧下降。
3. 绑定多个条件：ReentrantLock可以同时绑定多个Condition条件对象。

ReentrantLock基于AQS(**AbstractQueuedSynchronizer 抽象队列同步器**)实现。别说了，我知道问题了，AQS原理我来讲。

AQS内部维护一个state状态位，尝试加锁的时候通过CAS(CompareAndSwap)修改值，如果成功设置为1，并且把当前线程ID赋值，则代表加锁成功，一旦获取到锁，其他的线程将会被阻塞进入阻塞队列自旋，获得锁的线程释放锁的时候将会唤醒阻塞队列中的线程，释放锁的时候则会把state重新置为0，同时当前线程ID置为空。



CAS的原理呢？

CAS叫做CompareAndSwap，比较并交换，主要是通过处理器的指令来保证操作的原子性，它包含三个操作数：

1. 变量内存地址，V表示
2. 旧的预期值，A表示
3. 准备设置的新值，B表示

当执行CAS指令时，只有当V等于A时，才会用B去更新V的值，否则就不会执行更新操作。

那么CAS有什么缺点吗？

CAS的缺点主要有3点：

ABA问题：ABA的问题指的是在CAS更新的过程中，当读取到的值是A，然后准备赋值的时候仍然是A，但是实际上有可能A的值被改成了B，然后又被改回了A，这个CAS更新的漏洞就叫做ABA。只是ABA的问题大部分场景下都不影响并发的最终效果。

Java中有AtomicStampedReference来解决这个问题，他加入了预期标志和更新后标志两个字段，更新时不光检查值，还要检查当前的标志是否等于预期标志，全部相等的话才会更新。

循环时间长开销大：自旋CAS的方式如果长时间不成功，会给CPU带来很大的开销。

只能保证一个共享变量的原子操作：只对一个共享变量操作可以保证原子性，但是多个则不行，多个可以通过AtomicReference来处理或者使用锁synchronized实现。

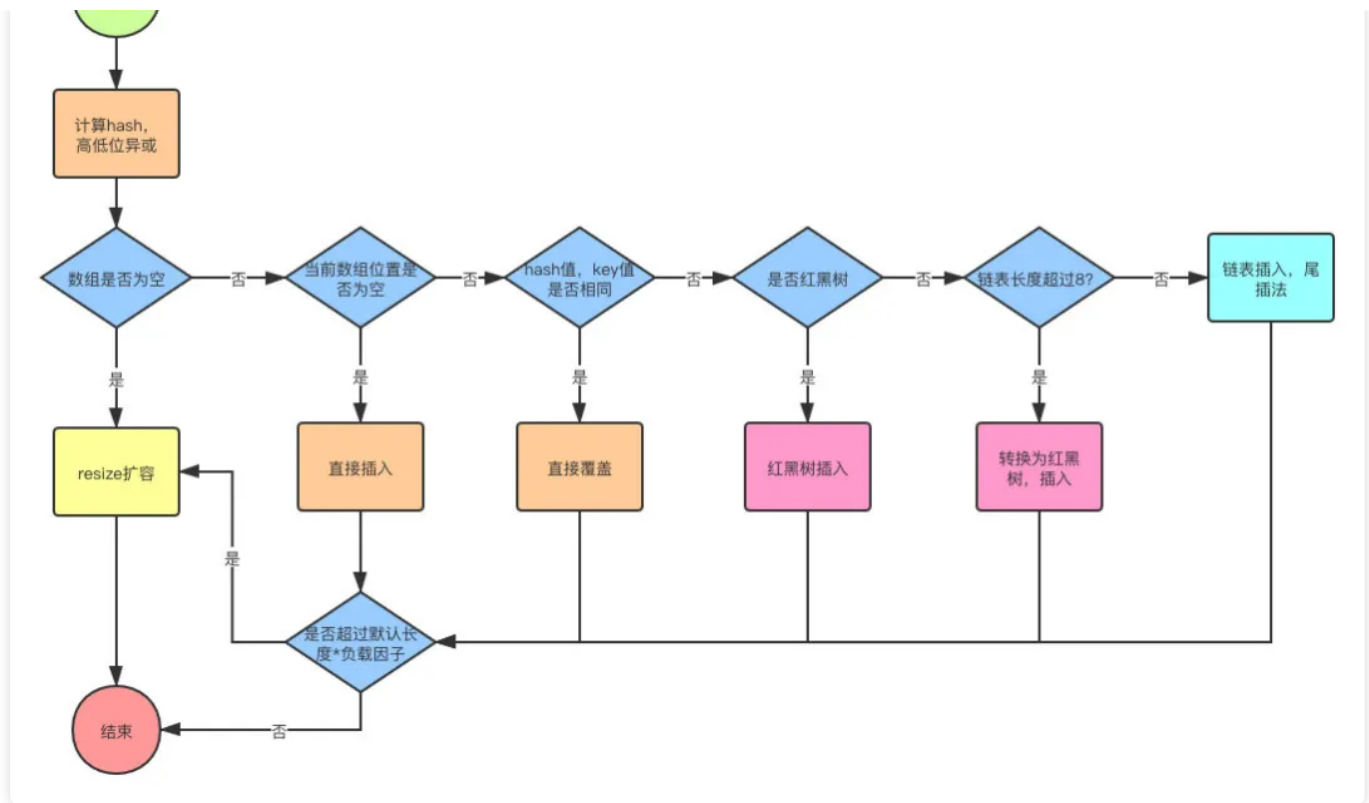
好，说说HashMap原理吧？

HashMap主要由数组和链表组成，他不是线程安全的。核心的点就是put插入数据的过程，get查询数据以及扩容的方式。JDK1.7和1.8的主要区别在于头插和尾插方式的修改，头插容易导致HashMap链表死循环，并且1.8之后加入红黑树对性能有提升。

put插入数据流程

往map插入元素的时候首先通过对key hash然后与数组长度-1进行与运算 $((n-1) \& \text{hash})$ ，都是2的次幂所以等同于取模，但是位运算的效率更高。找到数组中的位置之后，如果数组中没有元素直接存入，反之则判断key是否相同，key相同就覆盖，否则就会插入到链表的尾部，如果链表的长度超过8，则会转换成红黑树，最后判断数组长度是否超过默认的长度*负载因子也就是12，超过则进行扩容。





get查询数据

查询数据相对来说就比较简单了，首先计算出hash值，然后去数组查询，是红黑树就去红黑树查，链表就遍历链表查询就可以了。

resize扩容过程

扩容的过程就是对key重新计算hash，然后把数据拷贝到新的数组。

那多线程环境怎么使用Map呢？ConcurrentHashMap了解过吗？

多线程环境可以使用Collections.synchronizedMap同步加锁的方式，还可以使用HashTable，但是同步的方式显然性能不达标，而ConcurrentHashMap更适合高并发场景使用。

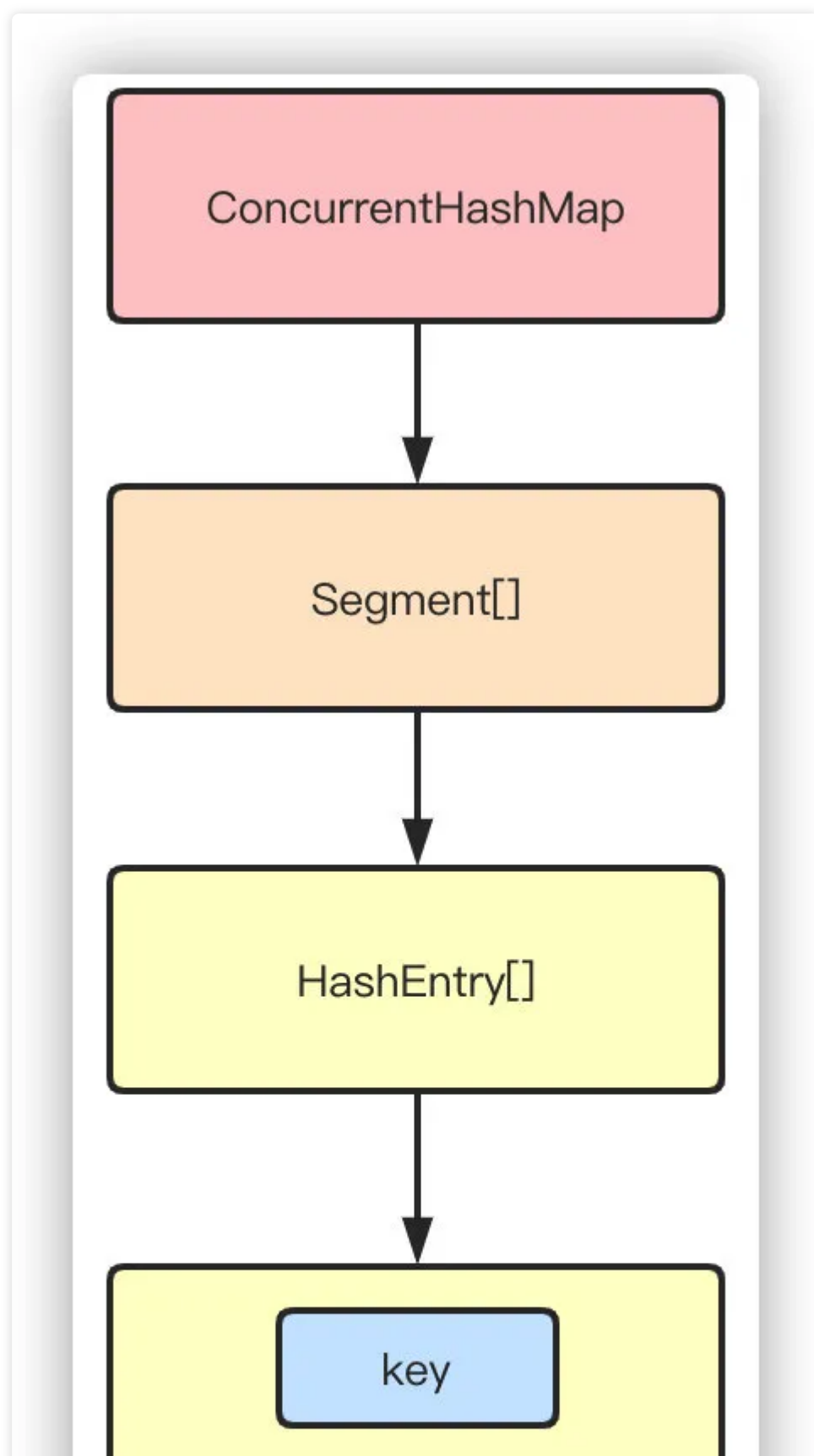
ConcurrentHashMap在JDK1.7和1.8的版本改动比较大，1.7使用Segment+HashEntry分段锁的方式实现，1.8则抛弃了Segment，改为使用CAS+synchronized+Node实现，同样也加入了红黑树，避免链表过长导致性能的问题。

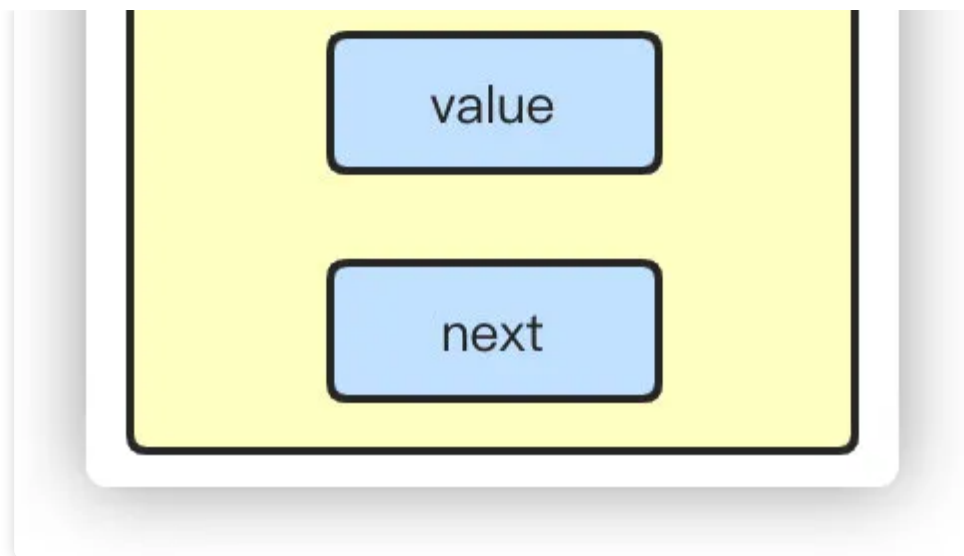
1.7分段锁

从结构上说，1.7版本的ConcurrentHashMap采用分段锁机制，里面包含一个Segment数组，Segment继承与ReentrantLock，Segment则包含HashEntry的数组，HashEntry本身就是一个链表的

结构，具有保存key、value的能力能指向下一个节点的指针。

实际上就是相当于每个Segment都是一个HashMap，默认的Segment长度是16，也就是支持16个线程的并发写，Segment之间相互不会受到影响。

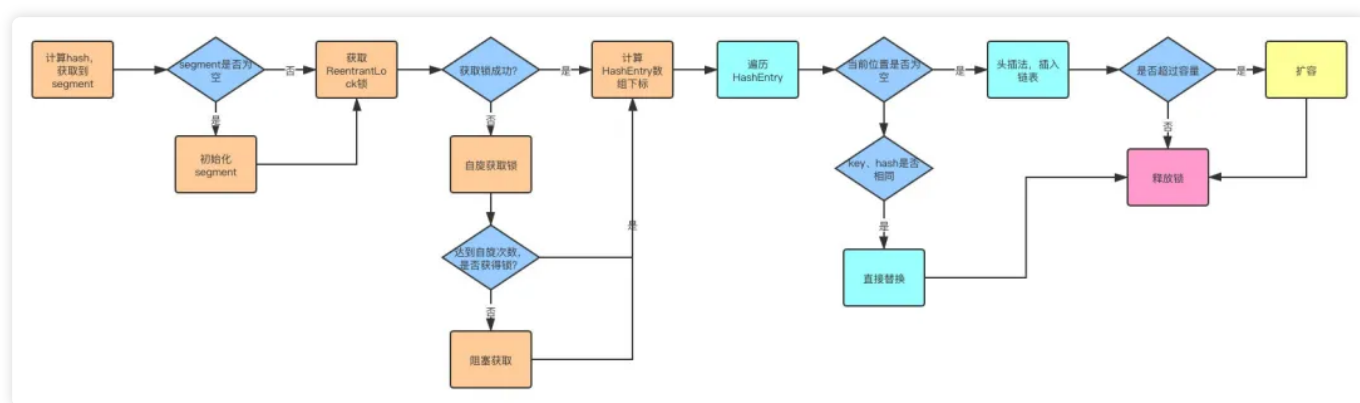




put流程

其实发现整个流程和HashMap非常类似，只不过是先定位到具体的Segment，然后通过ReentrantLock去操作而已，后面的流程我就简化了，因为和HashMap基本上是一样的。

1. 计算hash，定位到segment，segment如果是空就先初始化
2. 使用ReentrantLock加锁，如果获取锁失败则尝试自旋，自旋超过次数就阻塞获取，保证一定获取锁成功
3. 遍历HashEntry，就是和HashMap一样，数组中key和hash一样就直接替换，不存在就再插入链表，链表同样

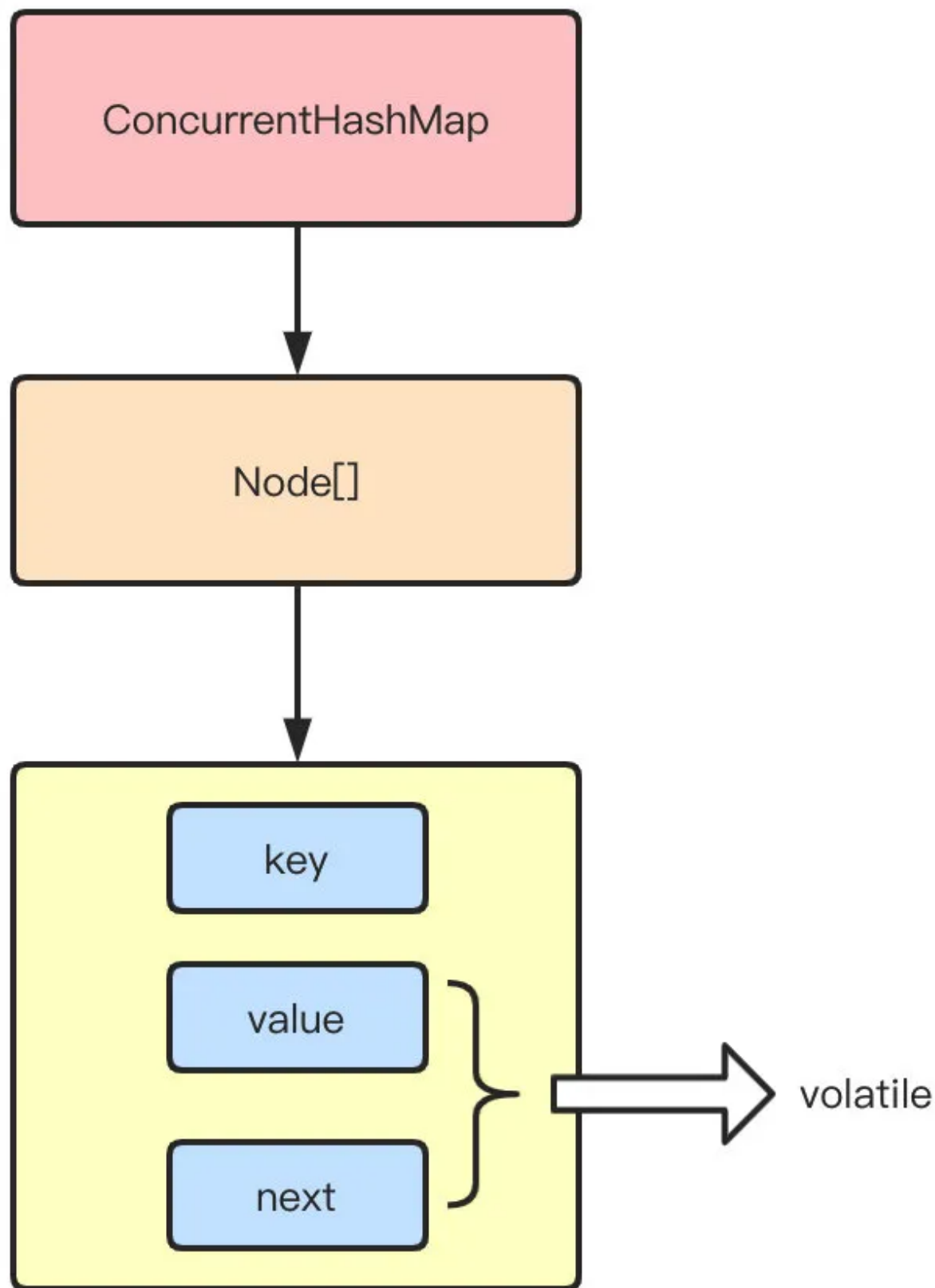


get流程

get也很简单，key通过hash定位到segment，再遍历链表定位到具体的元素上，需要注意的是value是volatile的，所以get是不需要加锁的。

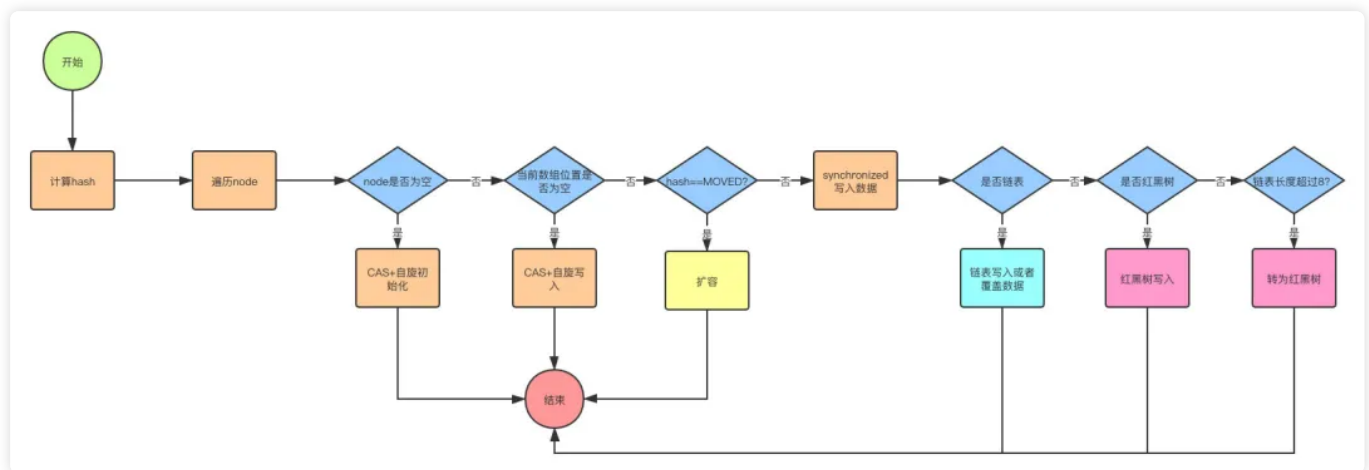
1.8CAS+synchronized

1.8抛弃分段锁，转为用CAS+synchronized来实现，同样HashEntry改为Node，也加入了红黑树的实现。主要还是看put的流程。



put流程

1. 首先计算hash，遍历node数组，如果node是空的话，就通过CAS+自旋的方式初始化
2. 如果当前数组位置是空则直接通过CAS自旋写入数据
3. 如果hash==MOVED，说明需要扩容，执行扩容
4. 如果都不满足，就使用synchronized写入数据，写入数据同样判断链表、红黑树，链表写入和HashMap的方式一样，key hash一样就覆盖，反之就尾插法，链表长度超过8就转换成红黑树



get查询

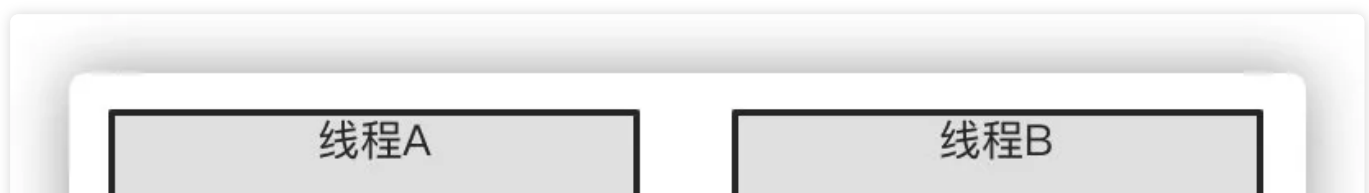
get很简单，通过key计算hash，如果key hash相同就返回，如果是红黑树按照红黑树获取，都不是就遍历链表获取。

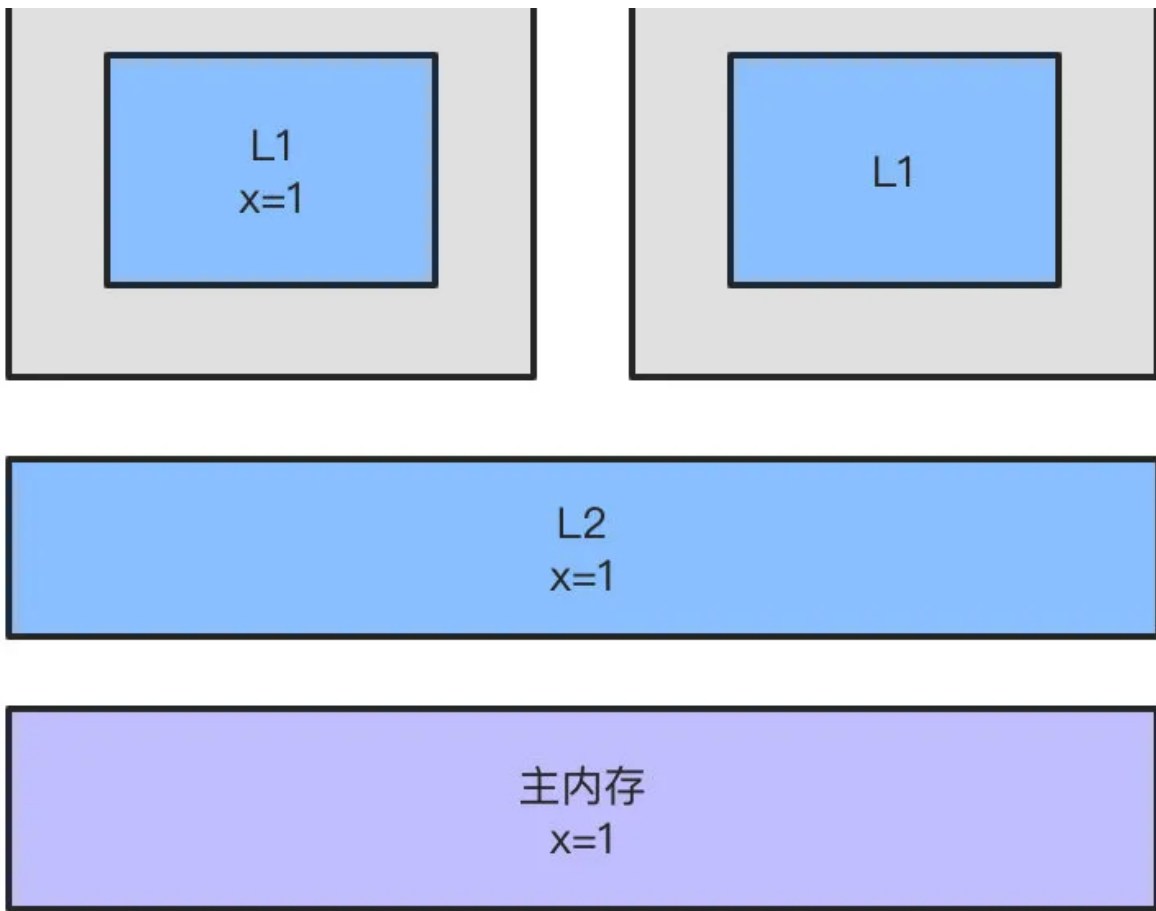
volatile原理知道吗？

相比synchronized的加锁方式来解决共享变量的内存可见性问题，volatile就是更轻量的选择，他没有上下文切换的额外开销成本。使用volatile声明的变量，可以确保值被更新的时候对其他线程立刻可见。volatile使用内存屏障来保证不会发生指令重排，解决了内存可见性的问题。

我们知道，线程都是从主内存中读取共享变量到工作内存来操作，完成之后再把结果写回主内存，但是这样就会带来可见性问题。举个例子，假设现在我们是两级缓存的双核CPU架构，包含L1、L2两级缓存。

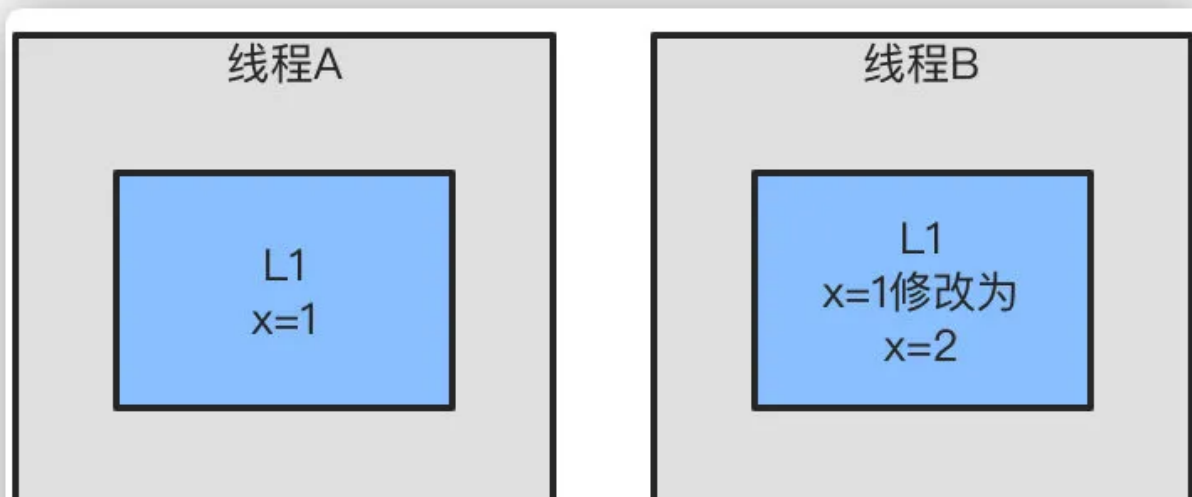
1. 线程A首先获取变量X的值，由于最初两级缓存都是空，所以直接从主内存中读取X，假设X初始值为0，线程A读取之后把X值都修改为1，同时写回主内存。这时候缓存和主内存的情况如下图。





2. 线程B也同样读取变量X的值，由于L2缓存已经有缓存X=1，所以直接从L2缓存读取，之后线程B把X修改为2，同时写回L2和主内存。这时候的X值如下图所示。

那么线程A如果再想获取变量X的值，因为L1缓存已经有x=1了，所以这时候变量内存不可见问题就产生了，B修改为2的值对A来说没有感知。



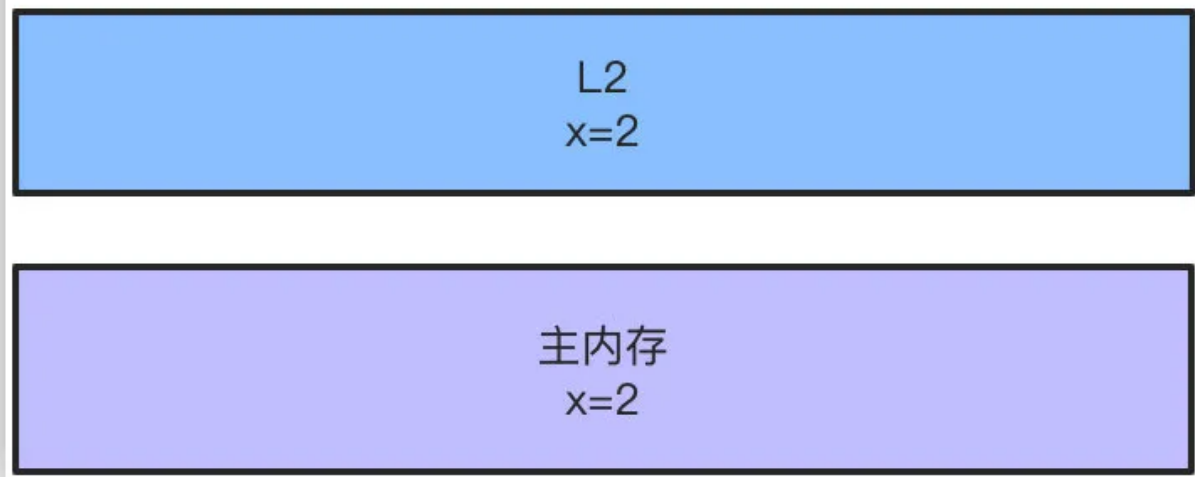
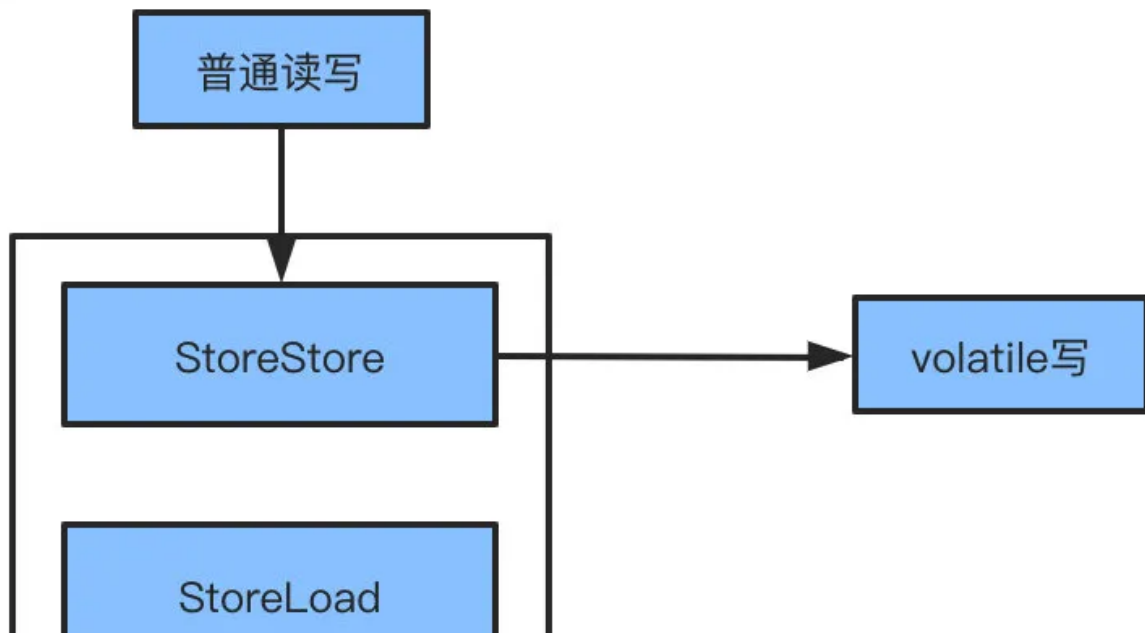


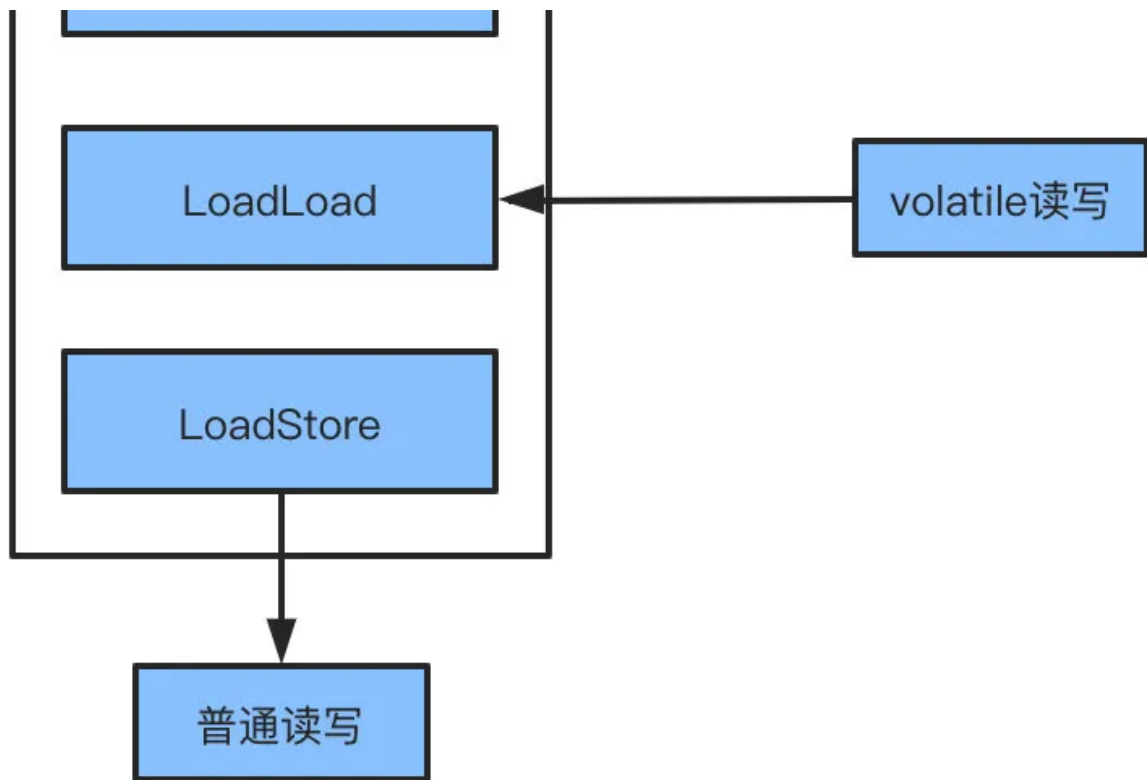
image-20201111171451466

那么，如果X变量用volatile修饰的话，当线程A再次读取变量X的话，CPU就会根据缓存一致性协议强制线程A重新从主内存加载最新的值到自己的工作内存，而不是直接用缓存中的值。

再来说内存屏障的问题，volatile修饰之后会加入不同的内存屏障来保证可见性的问题能正确执行。这里写的屏障基于书中提供的内容，但是实际上由于CPU架构不同，重排序的策略不同，提供的内存屏障也不一样，比如x86平台上，只有StoreLoad一种内存屏障。

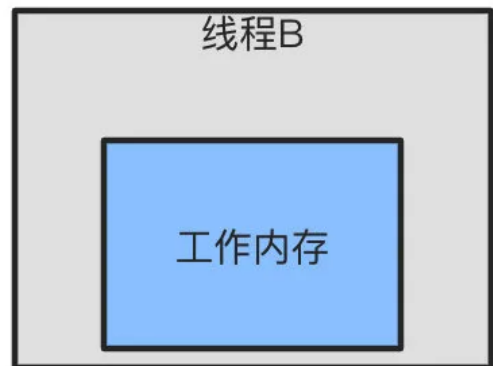
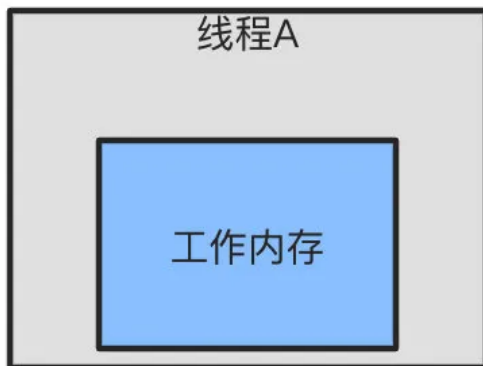
1. StoreStore屏障，保证上面的普通写不和volatile写发生重排序
2. StoreLoad屏障，保证volatile写与后面可能的volatile读写不发生重排序
3. LoadLoad屏障，禁止volatile读与后面的普通读重排序
4. LoadStore屏障，禁止volatile读和后面的普通写重排序






那么说说你对JMM内存模型的理解？为什么需要JMM？

本身随着CPU和内存的发展速度差异的问题，导致CPU的速度远快于内存，所以现在的CPU加入了高速缓存，高速缓存一般可以分为L1、L2、L3三级缓存。基于上面的例子我们知道了这导致了缓存一致性的问题，所以加入了缓存一致性协议，同时导致了内存可见性的问题，而编译器和CPU的重排序导致了原子性和有序性的问题，JMM内存模型正是对多线程操作下的一系列规范约束，因为不可能让陈雇员的代码去兼容所有的CPU，通过JMM我们才屏蔽了不同硬件和操作系统内存的访问差异，这样保证了Java程序在不同的平台下达到一致的内存访问效果，同时也是保证在高效并发的时候程序能够正确执行。





缓存一致性协议

主内存

原子性：Java内存模型通过read、load、assign、use、store、write来保证原子性操作，此外还有lock和unlock，直接对应着synchronized关键字的monitorenter和monitorexit字节码指令。

可见性：可见性的问题在上面的回答已经说过，Java保证可见性可以认为通过volatile、synchronized、final来实现。

有序性：由于处理器和编译器的重排序导致的有序性问题，Java通过volatile、synchronized来保证。

happen-before规则

虽然指令重排提高了并发的性能，但是Java虚拟机会对指令重排做出一些规则限制，并不能让所有的指令都随意的改变执行位置，主要有以下几点：

1. 单线程每个操作，happen-before于该线程中任意后续操作
2. volatile写happen-before与后续对这个变量的读
3. synchronized解锁happen-before后续对这个锁的加锁
4. final变量的写happen-before于final域对象的读，happen-before后续对final变量的读
5. 传递性规则，A先于B，B先于C，那么A一定先于C发生

说了半天，到底工作内存和主内存是什么？

主内存可以认为就是物理内存，Java内存模型中实际就是虚拟机内存的一部分。而工作内存就是CPU缓存，他有可能是寄存器也有可能是L1\L2\L3缓存，都是有可能的。

说说ThreadLocal原理？

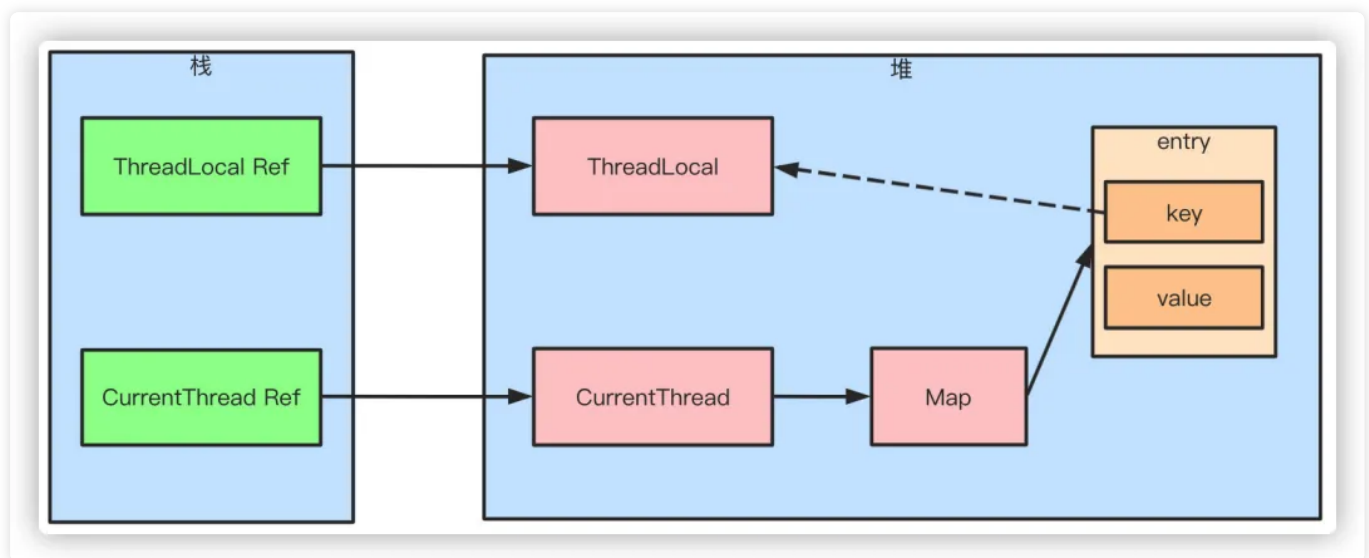
ThreadLocal可以理解为线程本地变量，他会在每个线程都创建一个副本，那么在线程之间访问内部副本变量就行了，做到了线程之间互相隔离，相比于synchronized的做法是用空间来换时间。

ThreadLocal有一个静态内部类ThreadLocalMap，ThreadLocalMap又包含了一个Entry数组，Entry本身是一个弱引用，他的key是指向ThreadLocal的弱引用，Entry具备了保存key value键值对的能力。

弱引用的目的是为了防止内存泄露，如果是强引用那么ThreadLocal对象除非线程结束否则始终无法被回收，弱引用则会在下一次GC的时候被回收。

但是这样还是会存在内存泄露的问题，假如key和ThreadLocal对象被回收之后，entry中就存在key为null，但是value有值的entry对象，但是永远没办法被访问到，同样除非线程结束运行。

但是只要ThreadLocal使用恰当，在使用完之后调用remove方法删除Entry对象，实际上是不会出现这个问题的。



那引用类型有哪些？有什么区别？

引用类型主要分为强软弱虚四种：

1. 强引用指的就是代码中普遍存在的赋值方式，比如A a = new A()这种。强引用关联的对象，永远不会被GC回收。
2. 软引用可以用SoftReference来描述，指的是那些有用但是不是必须需要的对象。系统在发生内存溢出前会对这类引用的对象进行回收。
3. 弱引用可以用WeakReference来描述，他的强度比软引用更低一点，弱引用的对象下一次GC的时候一定会被回收，而不管内存是否足够。

4. 虚引用也被称作幻影引用，是最弱的引用关系，可以用PhantomReference来描述，他必须和ReferenceQueue一起使用，同样的当发生GC的时候，虚引用也会被回收。可以用虚引用来管理堆外内存。

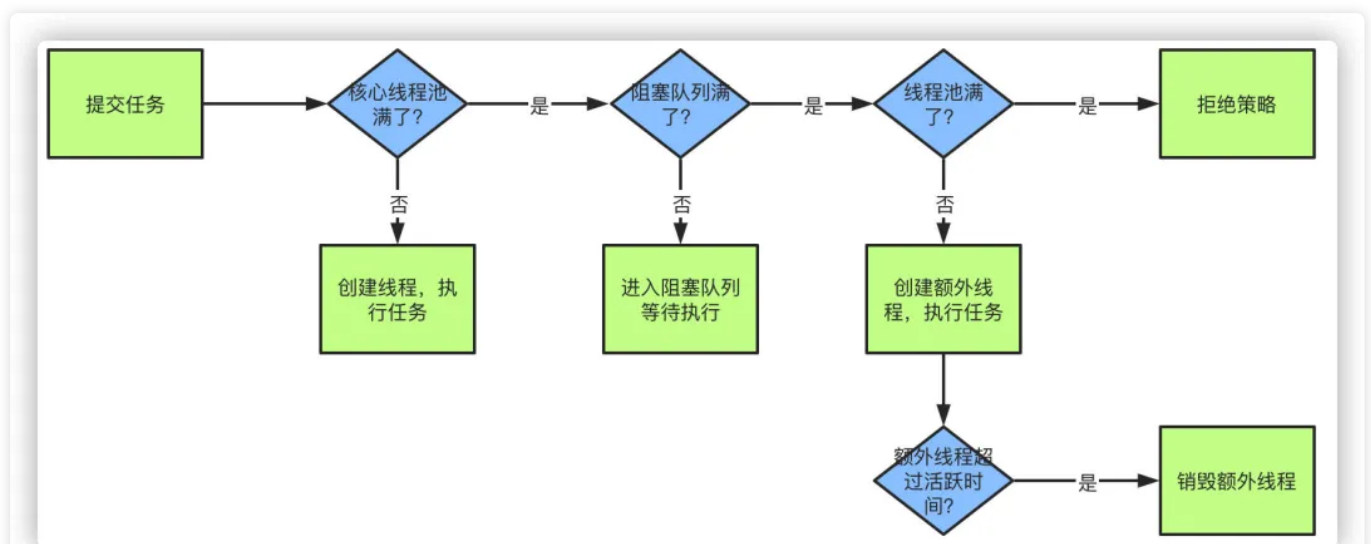
线程池原理知道吗？

首先线程池有几个核心的参数概念：

1. 最大线程数maximumPoolSize
2. 核心线程数corePoolSize
3. 活跃时间keepAliveTime
4. 阻塞队列workQueue
5. 拒绝策略RejectedExecutionHandler

当提交一个新任务到线程池时，具体的执行流程如下：

1. 当我们提交任务，线程池会根据corePoolSize大小创建若干任务数量线程执行任务
2. 当任务的数量超过corePoolSize数量，后续的任务将会进入阻塞队列阻塞排队
3. 当阻塞队列也满了之后，那么将会继续创建(maximumPoolSize-corePoolSize)个数量的线程来执行任务，如果任务处理完成，maximumPoolSize-corePoolSize额外创建的线程等待keepAliveTime之后被自动销毁
4. 如果达到maximumPoolSize，阻塞队列还是满的状态，那么将根据不同的拒绝策略对应处理



拒绝策略有哪些？

主要有4种拒绝策略：

1. AbortPolicy：直接丢弃任务，抛出异常，这是默认策略
2. CallerRunsPolicy：只用调用者所在的线程来处理任务
3. DiscardOldestPolicy：丢弃等待队列中最旧的任务，并执行当前任务
4. DiscardPolicy：直接丢弃任务，也不抛出异常

另外，读者群已经开通，讲真的，就算不说话，在群里也能学到不少东西呢。加我微信，备注进群，我拉你！



往期推荐

《我想进大厂》之JVM夺命连环10问

《我想进大厂》之Redis夺命连环11问

面试官：面对千万级、亿级流量怎么处理？

《我想进大厂》之MQ夺命连环11问



点击二维码识别关注

点在看，让更多看见。

收录于话题 #面试大全·29个

上一篇

《我想进大厂》之Spring夺命连环10问

下一篇

高频面试题：秒杀场景设计

文章已于2020/11/12修改

喜欢此内容的人还喜欢

毕业旅行

labuladong

聊聊spring事务失效的12种场景，太坑了

苏三说技术

RocketMQ 消息丢失场景分析及如何解决!

云时代架构