1. 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，它的作用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，而县城是比进程更小的能独立运行的基本单位，它是进程的一个试题，可以减少程序并发执行的时间和空间开销，使操作系统具有更好的并发性。

县城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

*2. synchronized*原理

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，它依赖操作系统底层互斥锁实现。它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。

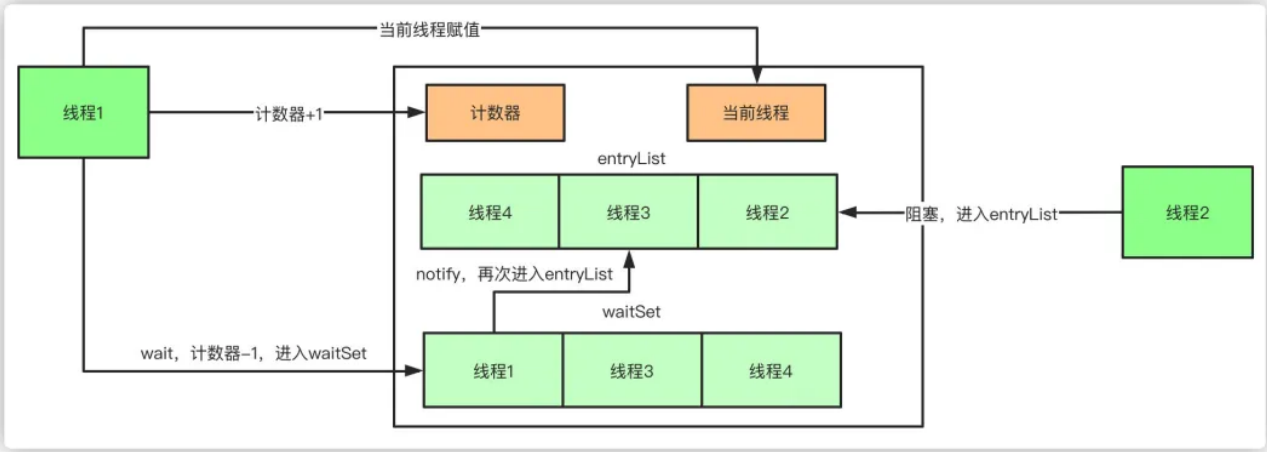
执行monitorexit指令时则会把计数器-1，当计数器值为0时，则锁释放，处于等待队列中的线程再继续竞争锁。

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之后才能获得锁，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，加锁的过程会清楚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。

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



互斥锁

通过原子汇编指令实现，比如x86的tsl指令，test and set， 用一条 无法继续分割的 汇编指令实现判断 变量值 并根据是否为0进行置位，具体这个指令实现原子性一般通过锁总线实现，也就是我执行这条指令时，其他核都不能访问这个地址了

用户态和内核态的切换

因为线程的调度是在内核态运行的，线程中的代码是在用户态运行的

3. 锁的优化机制

锁的状态从低到高依次为 无锁 -> 偏向锁 -> 轻量级锁 -> 重量级锁

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。

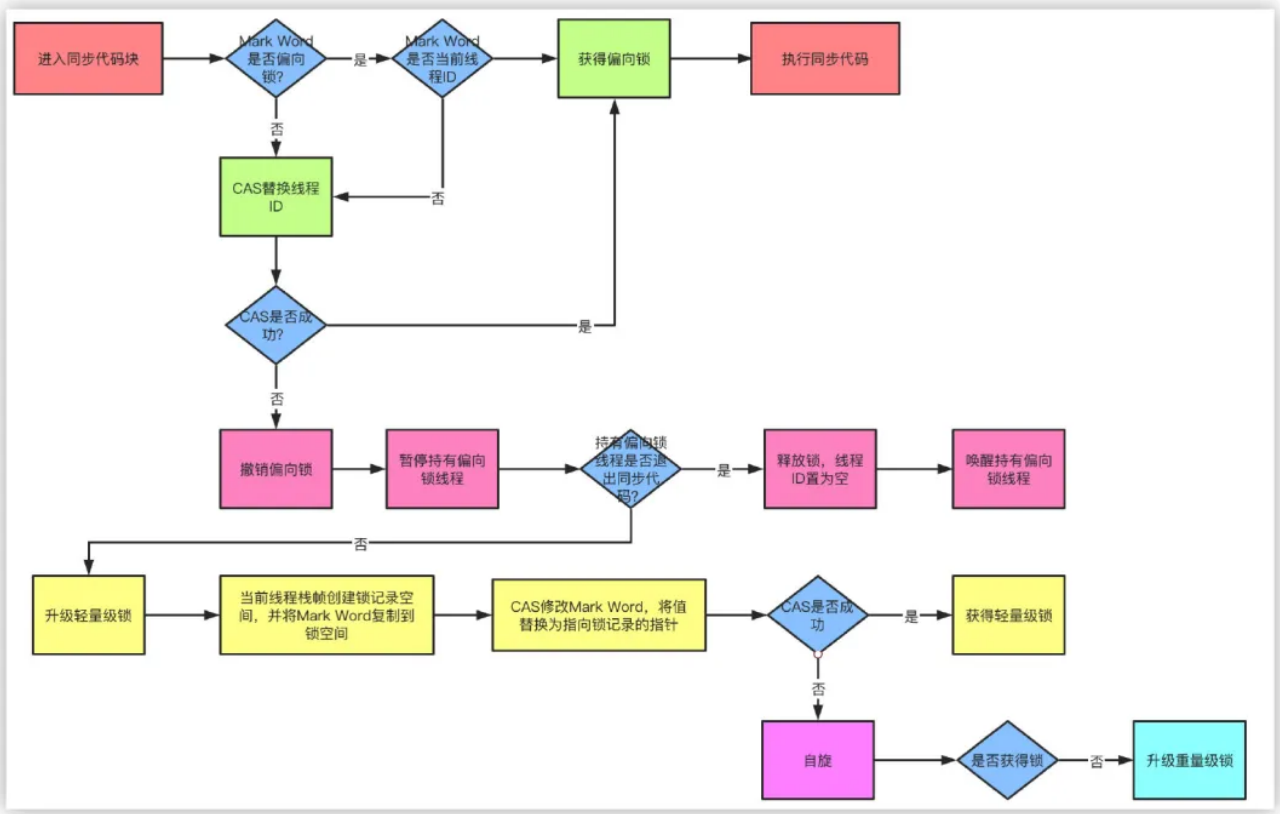
锁消除：锁消除指的是JVM检测到一些同步的代码块，完全不存在数据竞争的场景，也就是不需要加锁，就会进行锁消除。

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。

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



4. 对象头

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

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

5. ReentrantLock原理

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和synchronized的效率基本可以持平。

区别：

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，转而处理其他任务

2. 公平锁：synchronized和ReentrantLock默认都是非公平锁，但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 （AbstractQueueSynchronizer 抽象队列同步器）

AQS内部维护一个state状态位，尝试加锁的时候通过CAS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

6. CAS

原理：

比较并交换，通过处理器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，否则不会执行更新操作

缺点：

ABA问题：Java中有AtomicStampedReference来解决这个问题，它假如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synchronized实现

*7. HashMap*原理

HashMap主要由数组和链表组成，线程不安全。

JDK1.7和1.8的主要区别在于头插和尾插的修改，头插容易导致HashMap链表死循环，并且在1.8之后假如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) & hash)，都是2的次幂所以等同与取模，因为位运算的效率更高。

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，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长度 * 负载因子，超过则进行Kors。

get查询数据

首先计算出hash值，然后去数组查询、如果是红黑树就去红黑树查询，链表就遍历链表查询

resize扩容过程

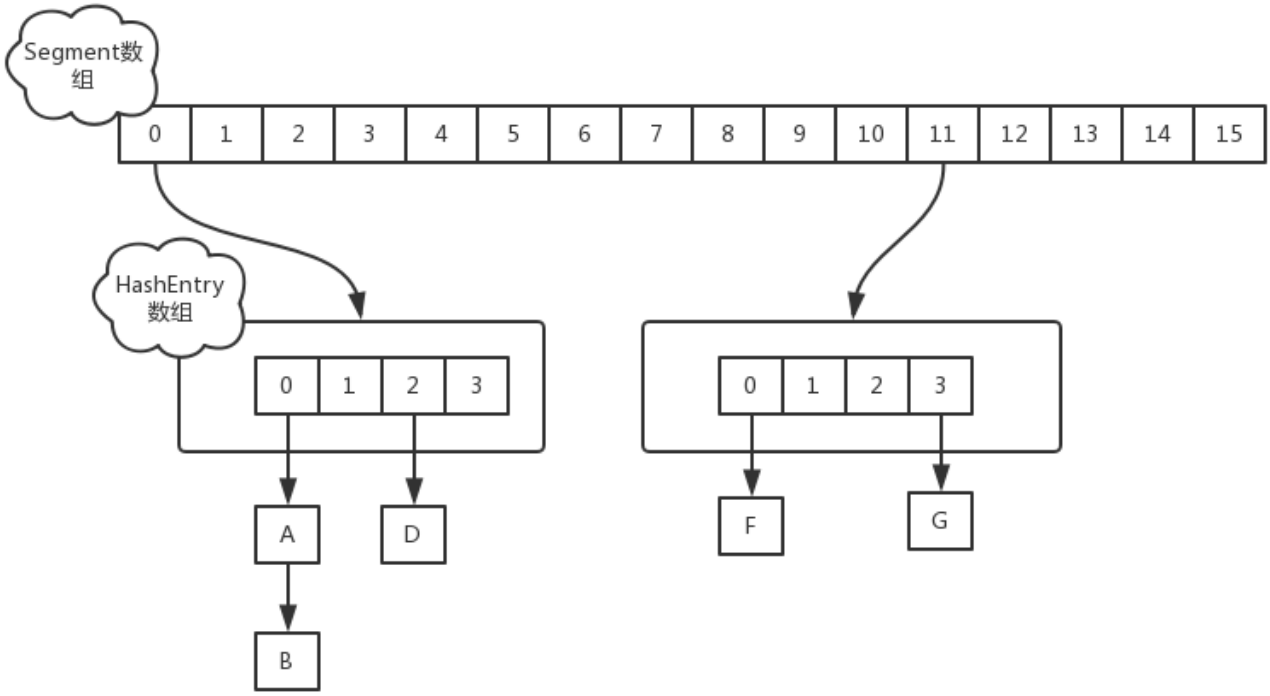
对ket重新计算hash，然后把数据拷贝到新的数组

8. 多线程环境怎么使用map

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

1.7使用Segment+HashEntry分段锁的方式实现



put流程

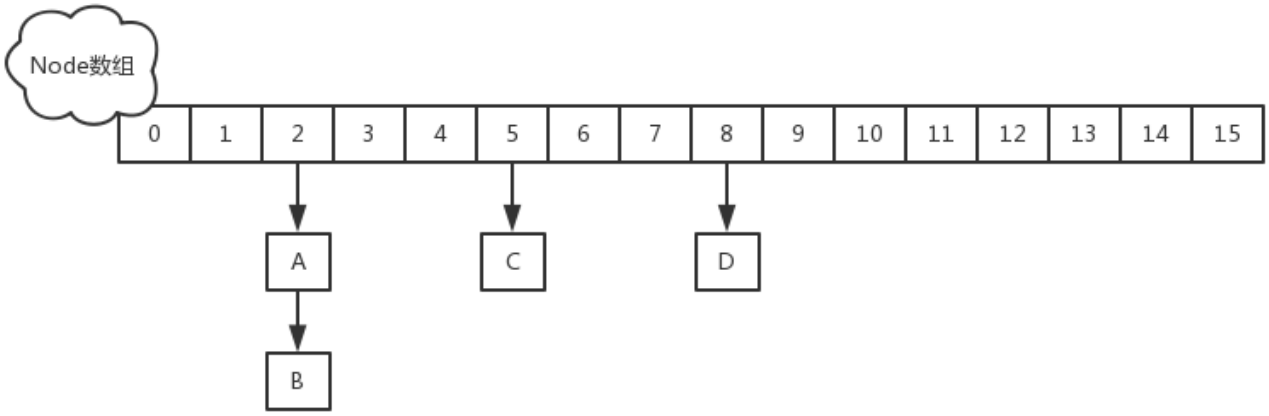
其实发现整个流程和HashMap非常类似，只不过是先定位到具体的Segment，然后通过ReentrantLock去操作而已，后面的流程我就简化了，因为和HashMap基本上是一样的。

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。

1.8改为使用CAS+synchronized+Node实现，同样假如了红黑树，避免链表过长导致的性能问题



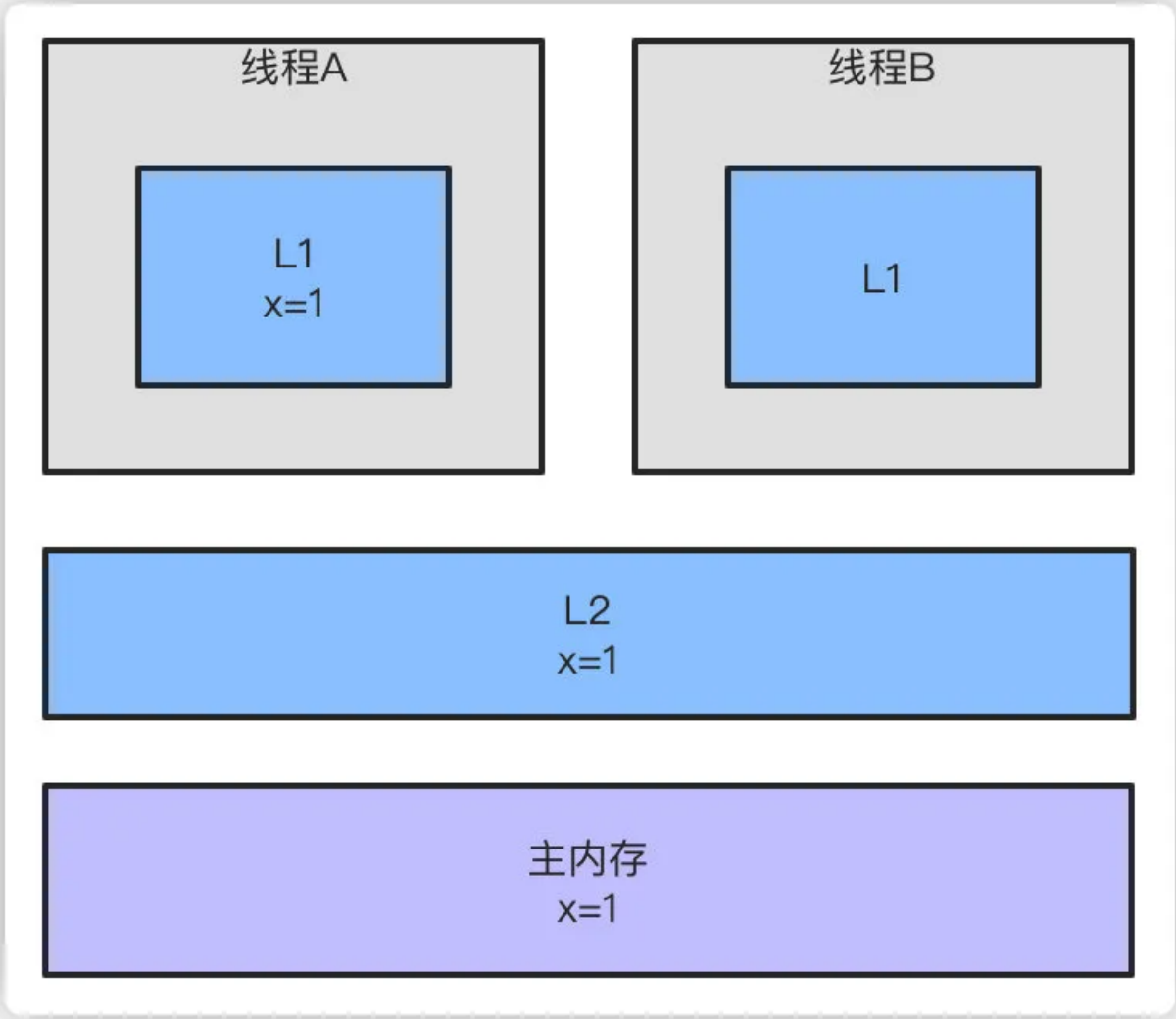
*9. volatile*原理

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会主内存，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。



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