**多线程的实现**

1. **进程和线程**

进程：程序运行资源分配的最小单位，进程内部有多个线程，会共享这个进程的资源。

线程：CPU调度的最小单位，必须依赖进程而存在。

1. 进程是资源分配的最小单位，线程是程序执行的最小单位。
2. 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵；线程是共享进程中的数据的，使用相同的地址空间，cpu切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。
3. 线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC）进行。
4. 多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间
5. **多线程三大特性**
6. 原子性：一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。
7. 可见性：当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看的到修改的值。
8. 有序性：程序执行的顺序按照代码的先后顺序执行。
9. **实现多线程的方法**

继承Thread类，重写run（）方法

实现Runnable接口，重写run（）方法

实现Callable接口，通过FutureTask包装器创建Thread线程

1. 继承Thread类为单继承，实现Runnable方法为多实现，所以灵活性上来讲，使用Runnable方法更灵活。
2. 通过实现Runnale接口的方式可以实现多线程内的资源共享，增加代码的健壮性，代码可以被多个线程共享，代码和数据独立。
3. 线程池只能放实现Runnable或Callable类的线程，不能直接放入继承Thread类的线程。
4. Thread和Runnable没有返回值，Callable可以有返回值。
5. 通过Thread实现线程时，一个线程只能启动一次，线程和线程所要执行的任务是捆绑在一起的。也就使得一个任务只能启动一个线程，不同的线程执行的任务是不相同的，所以没有必要，也不能让两个线程共享彼此任务中的资源；通过Runnable方式实现的线程，一个任务可以启动多个线程，实际是开辟一个线程，将任务传递进去，由此线程执行。可以实例化多个Thread对象，将同一个任务传递进去，也就是一个任务可以启动多个线程来执行它。这些线程执行的是同一个任务，所以他们资源是共享的。
6. **线程的状态**
7. 产生（start）：线程被创建，但未启动（执行new，未调用start）。
8. 就绪（runnable）：线程被启动，处于可执行状态，等待cpu的调度（调用start）。
9. 执行（running）：线程正常的运行状态（执行run方式里面的代码）。
10. 死亡（dead）：一个线程正常执行结束（run方法里面的代码执行完毕）。
11. 阻塞（blocking）：线程由于等待阻塞、同步阻塞、其他阻塞等原因进入阻塞状态。
12. 等待（waiting）：调用wait方法进入休眠，需要等待notify唤醒，进入就绪状态
13. **线程优先级**

线程的等级在1-10之间，小于1或者大于10将报异常。优先级越高，获取cpu时间片越多，获取的几率也越大。可以通过setPriority（int）方法设置线程的优先级。

1. **阻塞方法：**
2. Thread.yield（）方法：让出当前线程的cpu，使当前线程变成就绪态，和其他线程一起竞争cpu资源，但并没有终止当前线程的意思。
3. Thread.jion（）方法：在线程a中调用线程b的jion方法，会中断线程a的执行，等待线程b执行完才会在继续执行线程a。
4. Object.wait（）方法：该方法会导致当前线程等待，直到其他线程调用此对象的notify（）方法或者notifyAll（）方法，线程才会重新进入就绪状态，该方法会释放锁。
5. Thread.sleep（）方法：使当前休眠一段时间，该方法不会释放锁。
6. 关键字synchronized修饰的代码块或者方法，处于阻塞状态的时候，那么其他线程都无法访问这个方法，会一直阻塞下去。
7. **Dd**

**并发安全问题**

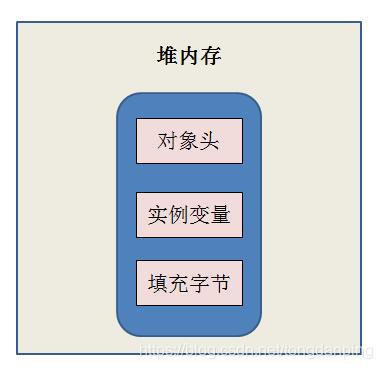
1. **定义：当多个线程同时执行时，多线程之间是相互抢占资源执行，并且抢占是发生在线程执行的每一步过程中，从而导致出现的非法数据的现象称之为多线程并发安全问题。**

**可以通过在并发地方加锁，解决并发安全的问题。（满足条件：一是数据共享<临界资源>，二是多线程同时访问并改变数据）**

1. **Synchronized：**
2. 使用场景：
3. Synchronized修饰普通同步方法：锁对象是当前实例对象
4. Synchronized修饰静态同步方法：锁对象是当前类Class对象
5. Synchronized修改同步代码块：锁对象是Synchronized后面括号里面配置的对象，这个对象可以是某个对象（xlock），也可以是某个类（Xlock.class）
6. 注意事项：
7. 使用synchronized修饰非静态方法或者使用synchronized修饰代码块时制定的为实例对象时，同一个类的不同对象拥有自己的锁，因此不会相互阻塞。
8. 使用synchronized修饰类和对象时，由于类对象和实例对象分别拥有自己的监视器锁，因此不会相互阻塞。
9. 线程A访问实例对象的非static synchronized方法时，线程B也可以同时访问实例对象的static synchronized方法，因为前者获取的时实例对象的监视器锁，而后者获取的是类对象的监视器锁，两者不存在互斥关系。
10. 使用synchronized修饰实例对象时，如果一个线程正在访问实例对象的一个synchronized方法时，其他线程不仅不能访问该synchronized方法，该对象的其他synchronized方法也不能访问，因为一个对象只有一个监视器锁对象，但是其他线程可以访问该对象的非synchronized方法。
11. Synchronized实现原理--java对象头：

首先，要知道对象在内存中的布局：已知对象是存放在堆内存中的，对象大致可以分为三部分，分别是对象头、实例变量和填充字节。

1. 对象头主要由Mark Word（标记词）和Klass Point（类型指针）组成，其中Klass Point是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，Mark Word用于存储对象自身的运行时数据。数组对象的对象头占用3个字节，非数组对象的对象头占用2个字节。（1 word=2 byte = 16 bit）。
2. 实例变量存储的是对象的属性信息，包括父类的属性信息，按照4字节对齐。
3. 填充字符，因为虚拟机要求对象字节必须是8字节的整数倍，填充字符就是用于凑齐这个整数倍的。

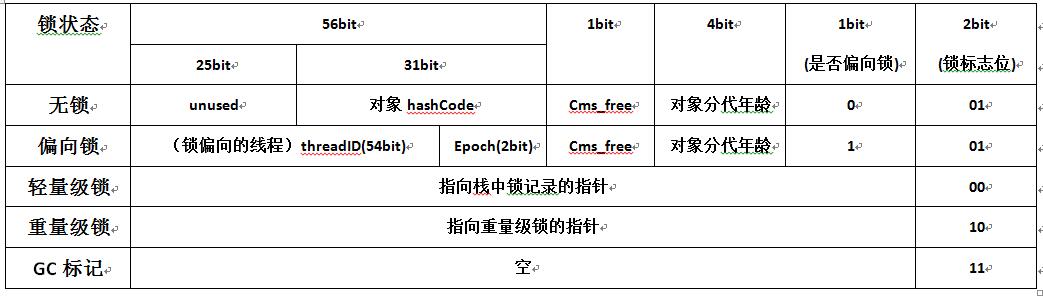


Synchronized不论是修饰方法还是代码块，都是通过持有修饰对象的锁来实现同步的，锁是存在锁对象的对象头的Mark Word中。Mark Word存储内容如下图，下图中的偏向锁和轻量级锁都是在java6以后对锁机制进行优化时引进的。

32位的虚拟机中：



64位的虚拟机中：



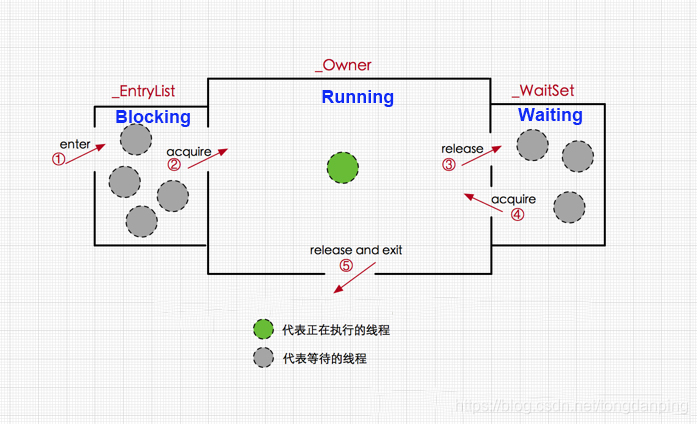
1. Synchronized实现原理—JVM

重量级锁对象的锁标志位是10，存储了指向重量级监视器锁的指针，在Hotspot（热点区）中，对象的监视器（monitor）锁对象由ObjectMonitor对象实现（C++），跟其同步相关的数据结构如下：



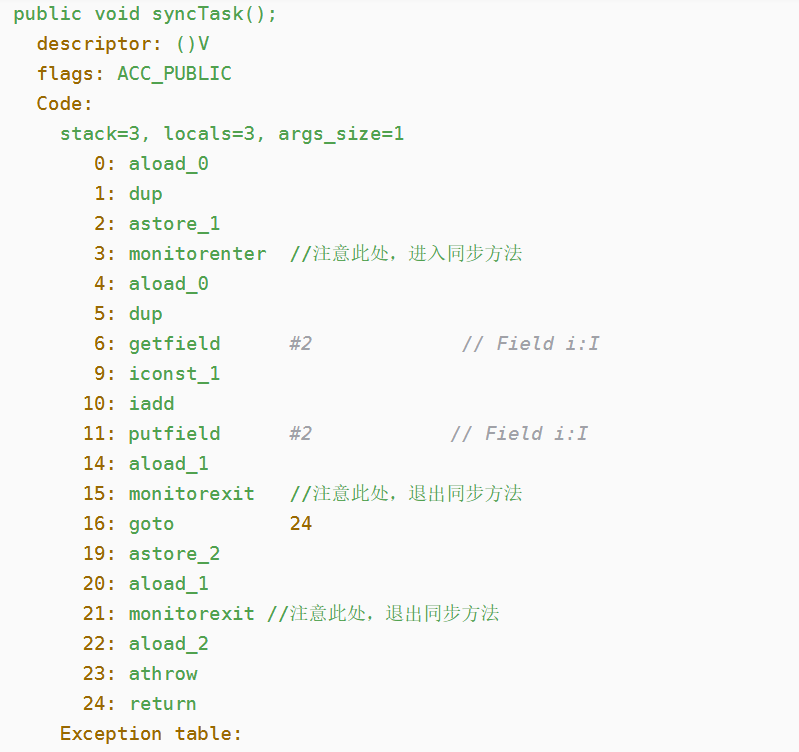
对于一个synchronized修饰的方法（代码块）来说：（线程的6种状态前文有介绍）

1. 当多个线程同时访问该方式，那么这些线程会先被放进\_EntryList队列，此时线程处于blocking状态。
2. 当一个线程获取到了实例对象的监视器（monitor）锁，那么就可以进入running状态，执行方法，ObjectMonitor对象的\_owner指向当前线程，\_count加1表示当前对象锁被一个线程获取。
3. 当running状态的线程调用wait（）方法，那么当前线程释放monitor对象，进入waiting状态，ObjectMonitor对象的\_owner变成null，\_count减1，同时线程进入\_WaitSet队列，直到有线程调用notify（）方法唤醒该线程。
4. 如果当前线程执行完毕，那么也会释放monitor对象，ObjectMonitor对象的\_owner变成null，\_count减1。



1. Synchronized不管是方法同步还是代码块同步都是基于进入和退出monitor对象来实现，然而二者在具体实现上又存在很大的区别。
2. Synchronized修饰代码块：在需要同步的代码块开始的位置插入monitorentry指令，在同步结束的位置或者异常出现的位置插入montorexit指令；JVM要保证monitorentry和monitorexit都是成对出现的，任何对象都有一个monitor与之对应，当这个对象的monitor被持有以后，它处于锁定状态。





1. Synchronized修饰方法：由方法调用指令来读取运行时常量池中的ACC\_SYNCHRONIZED标识隐式实现，如果被设置，那么线程在执行方法前会先区获取对象的monitor对象，如果获取成功则执行方法代码，执行完毕后释放minitor对象，如果monitor对象已经被其它线程获取，那么当前线程被阻塞。



1. 锁升级：无所状态、偏向锁状态、轻量级锁状态、重量级锁状态（级别从低到高）
2. 偏向锁引入原因：大多数时候是不存在锁竞争的，常常是一个线程多次获取同一个锁，因此如果每次都要竞争锁会增大很多没必要付出的代价，为了降低获取锁的代价，才引入了偏向锁。
3. 偏向锁设置：

是默认开启的，而且开始时间一般是比应用程序启动慢几秒，如果不想有这个延迟，那么可以JVM调优-XX:BiasedLockingStartUpDelay=0;

如果不想要偏向锁，也可以用JVM调优-XX:-UseBiasedLocking=false;

1. 偏向锁升级：

当线程1访问代码块并获取锁对象时，会在java对象头和栈帧中记录偏向锁的threadID，**因为偏向锁不会主动释放锁，因此以后线程1再次获取锁的时候，需要比较当前线程的threadID和java对象头中的threadID是否一致**，如果一致（还是线程1获取锁对象），则无需使用CAS来加锁、解锁；如果不一致（如线程2要竞争锁对象，而偏向锁不会主动释放因此还是存储的线程1的threadID），那么需要查看java对象中记录的线程1是否存活，如果没有存活，那么锁对象被重置为无锁状态，其他线程（线程2）可以竞争将其设置为偏向锁；如果存活，那么立刻查看该线程（线程1）的栈帧信息，如果还需要继续持有这个锁对象，那么暂停当前线程1，撤销偏向锁，升级为轻量级，如果线程1不再使用该锁对象，那么将锁对象状态设为无锁状态，重新偏向新的线程。

1. 轻量锁引入原因：

轻量级锁考虑的是竞争锁对象的线程不多，而且线程持有锁的时间也不长的情景。因为阻塞线程需要CPU从用户态转到内核态，代价较大，如果刚刚阻塞不久这个锁就被释放了，那这个代价就有点得不偿失，因此这个时候就干脆不阻塞这个线程，让它自旋这个等待锁释放。

1. 轻量锁升级：

线程1获取轻量级锁时会先吧锁对象的对象头Mark Word复制一份到线程1的栈帧中创建的用于存储锁记录的空间（DisplacedMarkWord），然后使用CAS把对象头中的内容替换为线程1存储的锁记录的地址（DisplacedMarkWord）；如果在线程1复制对象头的同时，线程2也准备获取锁，复制了对象头到线程2的锁记录空间中，但是在线程2做CAS的时候，发现线程1已经把对象头换了，线程2的CAS失败了，那么线程2就尝试使用自旋锁来等待线程1释放锁；自旋的次数是有限制的，如果自旋次数到了线程1还没有释放锁，或者线程还在执行，线程2还在自旋等待，这时又有一个线程3过来竞争这个锁对象，那么这个时候轻量级锁就会膨胀为重量级锁。重量级锁把除了拥有锁的线程都阻塞，防止CPU空转。

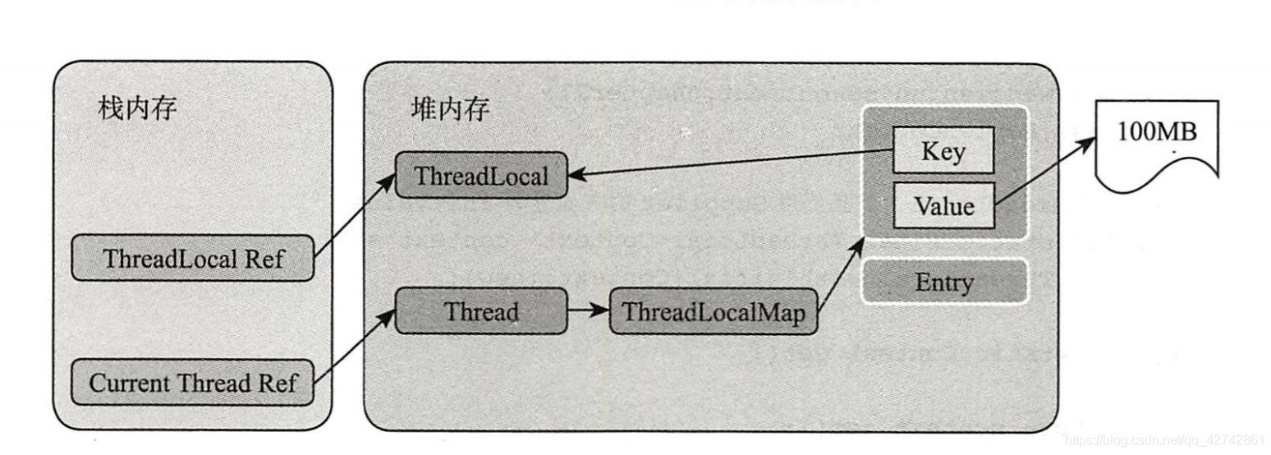
1. 轻量级锁一旦膨胀为重量级锁就不会再降级为轻量级锁； 偏向锁升级为轻量级锁也不能再降级为偏向锁。锁可以升级不能降级，但是偏向锁状态可以重置为无锁状态。

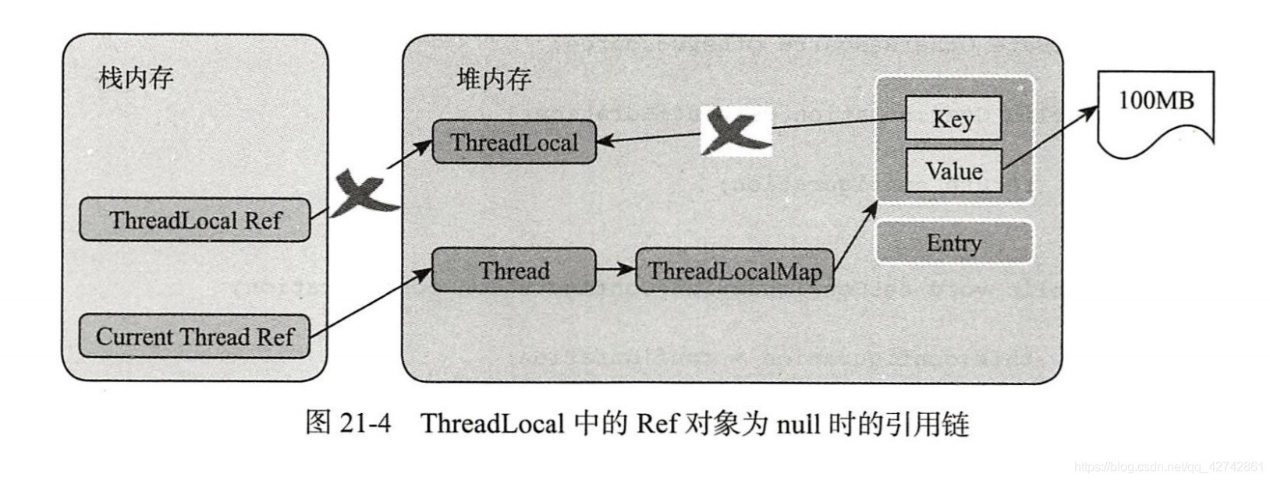


1. 锁粗化：按理来说，同步块的作用范围应该尽可能小，仅在共享数据的实际作用域中才进行同步，这样做的目的是为了使需要同步的操作数量尽可能缩小，缩短阻塞实际，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。但是加锁解锁也需要消耗资源，如果存在一系列的连续加锁解锁操作，可能会导致不必要的性能损耗。锁粗化就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁，避免频繁的加锁解锁操作。
2. 锁消除：JVM在JIT编译时（即使编译），通过对运行上下文的扫描，经过逃逸分析，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间。
3. **Lock和Synchronized比较**
4. Synchronized是java内置的关键字，当我们调用它时会在虚拟机指令层面加锁，关键字为monitorenter和monitorexit，使用后会自动释放锁；Lock是java.util.concurrent.Locks包下的一个接口，加锁的关键代码为lock和unLock，使用后必须手动释放锁，特别是发生异常时，需要在finally块中进行手动释放，否则会发生死锁行为。
5. Synchronized可对实例方法、静态方法和代码块加锁，相对应的，加锁前需要获得实例对象的锁或类对象的锁或指定对象的锁，说到底就是先获取对象的监视器（锁）然后才能进行相关的操作；Lock加锁并不是针对对象的，而是针对当前线程的，并却有一个原子类state来进行加锁次数的计数。
6. Lock可响应中断，并提供了更丰富的方法实现，而synchronized不能响应中断。
7. Synchronized是非公平锁，即不能保证等待锁线程的顺序；ReentrantLock可通过实例化构造参数实现公平锁和非公平锁，默认为非公平锁。
8. Synchronized无法判断释放获取锁的状态，Lock可以isLocked()判断是否获取到锁。
9. Lock锁适合大量的代码的同步问题，synchronized锁适合代码少量的同步问题。
10. Synchronized是一个悲观锁，lock是一个乐观锁（底层基于volatile和cas实现）。
11. **volatile**
12. 保证了有序性（防止指令重排序）和变量的内存可见性（每次都强制取主内存数据），每次取到volitile变量一定是最新的。
13. 可见性实现原理：修改volatile变量时会强制将修改后的值刷新到主内存；修改volatile变量后，会导致其他线程内存工作中对应的变量失效，在读取变量值时，要从主内存中读取。
14. 有序性实现原理：volatile采用“内存屏障”来实现禁止指令重排序。
15. 重排序：在不改变单线程语义的前提下，可以重新安排语句的执行顺序。
16. 重排序影响：指令重排序不会影响单线程，但会影响多线程的正确性。
17. **Atomic：通过CAS原理实现原子性**
18. **ThreadLocal类：对于多线程资源共享问题，同步机制采用的“以时间换空间”的方式，而ThreadLocal采用“以空间换时间”的方法。前者仅提供一份变量，让不同的线程排队访问，而后者为每个线程都提供一份变量，因此可以同时访问互不影响。**
19. 作用：保证线程安全；在线程级别传递变量
20. 注意事项：

A、线程复用会造成脏数据：由于线程池会复用Thread对象，因此Thread类的成员变量threadLocals也会被复用。如果在线程的run（）方法中不显示调用remove（）清理与线程相关的ThreadLocal信息，并且下一个线程不调用set（）设置初始值，就可能get（）到上个线程设置的值。(解决方法：每次使用完ThreadLocal后，及时调用remove（）方法清理即可)

B、内存泄漏：ThreadLocal在一定程度上保证不会发生内存泄漏，但是Thread类中有ThreadLocalMap的引用，导致对象的可达性，故不能回收，除非当前线程结束，Thread引用就会被垃圾回收器回收。（解决方法：在使用完ThreadLocal后，及时remove掉）





1. ThreadLocal用于线程间的数据隔离，而synchronized用于线程间的数据共享。
2. **Sss**
3. **Sss**
4. **Sssss**
5. **Sssssss**
6. **Sssssssss**