链滴

Java 并发编程之（一）常用概念 (转)

作者：lanjian

原文链接：https://ld246.com/article/1478594144294

来源网站：链滴

本文主要介绍在并发与多线程编程中常用的一些词汇以及简单解释，主要包括了可见性、原子性活跃性、线程安全等重要概念。
<h1>可见性</h1>
所谓的可见性，就是在多线程环境下，一个线程的改动能够被其他线程看见。通常在多线程环境，每个线程都有自己的线程空间，线程创建的时候，会将主线程变量拷贝一份到线程空间中，线程中行的更改会立即在线程空间中反映出来，但是对于其它线程并不能看见该线程做的改动。
<h2>失效数据</h2>
当一个线程对数据修改之后没有及时将修改同步到其他线程，这个数据就是失效的数据。
<h2>同步</h2>
此时需要进行同步，将线程空间中的改动同步到主线程中，因此为了实现可见性，同步是必不可的操作。
常用的具有同步功能的java关键字或者类：

synchronized代码块
继承了Lock的锁，例如ReentrantLock
volatile关键字
原子变量
同步原语，例如compareAndSwapObject

<h2>加锁与可见性</h2>
上节提到了同步的重要性，然而在多线程环境中，不加限制的同步会导致程序错误。
例如主线程有一个变量x=1；线程A对其进行加一操作之后x=2，同时线程B也进行了加一操作，理说此时x应该为3，然而由于线程B的加一操作是在线程A进行同步操作之前发生的，线程B返回的x也等于2，此时程序发生了错误。
为了避免这种情况的发生，为了保证线程的安全，需要采取一定的措施。普遍的做法有两种：</>
<h3>（1）加锁</h3>
加锁的意思是，当多个线程要修改同一个对象的内容时，要求同一个时间只能有一个线程对该对进行操作，当一个线程获得了锁，其他线程就需要阻塞。这样保证了数据修改的一致性，就不会发生误了。
<h3>（2）循环cas</h3>
循环cas是一种无锁（无阻塞）的线程安全方式，常用同步原语compareAndSwap来实现，具体在后面介绍无阻塞队列中讲解。
<h1>原子性</h1>
 原子性又可以称为不可分割性，指出一系列操作要么一起完成，要么都不完成，不会中结束。原子性无论在事务还是程序运行上都是一个很重要的特性。在单线程程序中一段代码编译之后执行顺序和程序员编码的顺序并不是一样的，这是编译器优化的结果。虽然顺序不一样，但优化的时还是会保证程序的顺序一致性，也就是不会出错。然而在多线程程序中，对共享变量的操作顺序很难证，如下代码所示，两个原子变量单独的操作是原子的，但是不能保证两个变量NumberA和Number的复合操作也是原子的。假设两个线程同时执行doSet方法，一个线程执行doSet(1),另一个线程执行dSet(2),那么最后的结果可以是：1、NumberA=1,NumberB=1;2、NumberA=2,NumberB=2;3、NmberA=1,NumberB=2;4、NumberA=2,NumberB=1;假如能够保证doSet方法的原子性，那么结只可能是：1、NumberA=1,NumberB=1;2、NumberA=2,NumberB=2;
<pre class="prettyprint prettyprinted">AtomicReference<Integer> NumberA =ne AtomicReference<Integer>();
AtmicReference<Integer<san class="pun">> NumberB=</pan>new AtomicReference<Integer>();
publicvoid doSet(int<span class="p

n"> i){
  NumberA.set(i);
  NumberB.set(i);
}</pre>
在很多场景下，原子性的要求是必要的，例如我们需要NumberA和NumberB同时变化。为了实现原子性，有两种方法：加锁、原语操作
加锁的方式来实现的原子性，同一时间只有一个线程在修改某个对象，那么就能够保证原子性。如synchronized代码块中的代码就具有原子性，编译器在编译优化的时候就不会将synchronized代码块内的代码和代码块外的代码交换顺序，这就保证了顺序一致性。
原语操作中包含某些简单的原子操作，例如incrementAndGet包含了读取-修改-写入操作。
<h1>活跃性</h1>
加锁能保证线程安全，但是不恰当的加锁方式会导致活跃性的降低甚至活跃性的僵死。
<h2>死锁</h2>
<h3>（1）顺序死锁</h3>
最简单的一种死锁形式是有两个锁A,B，线程1持有锁A等待锁B，线程2持有锁B等待锁A，这种况下两个线程会一直等待下去。
上述情况更加普遍化就是两个线程试图以不同的顺序来获得相同的锁，如果锁链形成了一个环状会出现死锁。数据库系统中有监测锁的依赖关系，监测到环状的锁链就会强制释放锁从而解除死锁。而JVM没有那么强大，需要我们在编码的时候避免顺序死锁。
<h3>（2）资源死锁</h3>
<h2>饥饿</h2>
饥饿是在线程无法访问它需要的资源时而不能继续执行的情况，引发饥饿最常见的资源就是CPU钟周期，例如Java应用程序对线程的优先级使用不当，或者持有锁的线程执行一些无法结束的结构。以通过Thread.sleep或者Thread.yield来克服优先级调整和响应性问题。
<h2>活锁</h2>
活锁是犹豫错误处理机制或者回滚机制导致的，典型的现就是一个任务不停地失败，不停地回滚，导致其他线程不能前进。还有一个典型的现象，例如两个非有礼貌的路人互相让路，结果都没前进，还有就是局域网拥塞。
解决方法就是在冲突重试中加入随机数。（或者参考tcp的重传机制）
<h1>另类的线程安全</h1>
<h2>不变性</h2>
不变的东西是不会存在线程安全问题的。良好的编程风格中有如下说法：
除非需要更高的可见性，否则应将所有的域都声明为私有欲，是一个良好的编程习惯。
除非需要某个域是可变的，否则应将其声明为final域，也是一个良好的编程习惯。
<h2>封闭性</h2>
当一个线程内的数据不会被其他线程共享时，也是线程安全的。例如无状态的Servlet,独立的Servet线程完成独立的工作，不需要和其他线程协作，此时就是线程安全的。
<h2>发布与逸出</h2>
 上面提到的封闭性，一个类中的成员变量如果不希望被其他线程或对象访问到，就应该置成private，甚至不应该在public函数中返回该变量的引用，一旦返回了该变量的引用就相当于将该象发布出去了，其他线程就可以修改该对象。
如果想要把一个对象发布出去，应当尽量按照如下方式：

在静态初始化函数中初始化一个对象的引用
将对象的引用保存到volatile类型的域或者AtomicReferance对象中
将对象的引用保存到某个正确构造对象的final类型域中
将对象的引用保存到一个由锁保护的域中
