**Deadlock Prevention**

* [Lock Ordering](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#ordering)
* [Lock Timeout](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#timeout)
* [Deadlock Detection](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#detection)

在某些情况下，可以防止死锁。我将在本文中描述三种技巧：

1. [**Lock Ordering**](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#ordering)
2. [**Lock Timeout**](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#timeout)
3. [**Deadlock Detection**](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#detection)

**Lock Ordering**

当多个线程需要相同的锁但以不同的顺序获得它们时会发生死锁。

如果确定所有锁始终被任一线程按相同顺序获取，则死锁不会发生。看看这个例子：

Thread 1:

lock A

lock B

Thread 2:

wait for A

lock C (when A locked)

Thread 3:

wait for A

wait for B

wait for C

如果一个线程，比如线程3，需要几个锁，它必须按定好的顺序获取它们。它不能获取在序列中靠后的锁，直到它获得了先前的锁。

例如，线程2或线程3都不能锁定C，直到它们锁定了第一个。由于线程1持有锁A，线程2和3必须首先等待直到锁定A被解锁。然后他们必须成功锁定A，然后才能锁定B或C。

锁排序是一种简单而有效的死锁预防机制。然而，它只能用在你在获取任一锁之前知道所有需要的锁。情况并非总是如此。

**Lock Timeout**

另一个死锁预防机制是对锁尝试设置超时，这意味着试图获得锁的线程在放弃之前只会尝试这么长时间。如果一个线程在给定的超时内没有成功获取所有必需的锁，那么它将进行备份、释放所获取的所有锁、等待随机的时间量然后重试。等待的随机时间量给尝试使用相同锁的其他线程一个获取所有锁的机会，从而让应用程序继续运行而不锁定。

下面是两个线程试图以不同顺序使用相同的两个锁的示例，其中线程备份并重试：

Thread 1 locks A

Thread 2 locks B

Thread 1 attempts to lock B but is blocked

Thread 2 attempts to lock A but is blocked

Thread 1's lock attempt on B times out

Thread 1 backs up and releases A as well

Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2's lock attempt on A times out

Thread 2 backs up and releases B as well

Thread 2 waits randomly (e.g. 43 millis) before retrying.

在上面的示例中，线程2将在线程1之前重试约200毫秒尝试获取锁，因此很可能成功获取两个锁。然后，线程1将等待已经尝试获取锁A。当线程2完成时，线程1将能够同时获取两个锁(除非线程2或其他线程在两者之间获取锁)。

要记住的一个问题是，仅仅因为锁超时并不意味着线程已经死锁。它还可能仅仅意味着持有锁(导致其他线程超时)的线程需要很长时间来完成其任务。

此外，如果足够多的线程竞争相同的资源，那么即使超时和备份，它们仍然会冒一次又一次尝试同时占用线程的风险。这可能不会发生在重试之前每个等待0-500毫秒的2个线程中，但是对于10-20个线程的情况是不同的。那样两个线程在重试之前等待相同时间(或足够接近以导致问题)的可能性要高得多。

锁定超时机制的一个问题是，无法在Java中设置进入同步块的超时。您必须创建自定义锁类，或者使用java.util.concurrency包中的Java 5并发结构之一。编写自定义锁并不困难，但它超出了本文的范围。后面的Java并发教程部分的文章将涉及自定义锁。

**Deadlock Detection**

死锁检测是一种较重的死锁预防机制，用于锁排序不可行、锁超时不可行的情况。

每次线程**拿到**锁时，它在线程和锁的数据结构(map、graph等)中被标记。此外，每当线程**请求**锁时，在这个数据结构中也会标记。

当线程请求锁但请求被拒绝时，线程可以遍历锁图来检查死锁。例如，如果线程A请求锁7，但是锁7由线程B持有，那么线程A可以检查线程B是否请求了线程A持有的任何锁(如果有的话)。如果线程B已经请求了，则出现死锁(线程A已取得锁1，请求锁定7，线程B已取得锁7，请求锁定1)。

当然，死锁场景可能比两个线程相互锁的复杂得多。线程A可以等待线程B，线程B等待线程C，线程C等待线程D，线程D等待线程A。为了让线程A检测死锁，它必须传递性地检查线程B请求的所有锁。线程A将从线程B的请求锁到达线程C，然后到达线程D，从中找到线程A本身持有的锁之一。然后它知道死锁已经发生了。

下面是由4个线程(A、B、C和D)所获取和请求的锁的图。这样的数据结构，可以用来检测死锁。

|  |
| --- |
| Deadlock Detection Data Structure |

那么，如果检测到死锁，线程会做什么？

一个可能的行动是释放所有锁，备份，等待一个随机的时间量，然后重试。这与简单的锁超时机制类似，只是线程在实际发生死锁时才备份。而不仅仅是因为他们的锁定请求超时。然而，如果许多线程争用相同的锁，那么即使它们备份并等待，它们也可能重复地以死锁结束。

一个更好的选择是确定或分配线程的优先级，以便只有一个(或几个)线程备份。其余线程继续占用它们所需的锁，就像没有发生死锁一样。如果分配给线程的优先级是固定的，那么相同的线程将总是被赋予更高的优先级。为了避免这种情况，可以在检测到死锁时随机分配优先级。