# 并发级别

由于临界区的存在，多线程之间的并发必须受到控制。根据控制并发的策略，我们可以把并发的级别分为**阻塞、无饥饿、无障碍、无锁、无等待**几种。

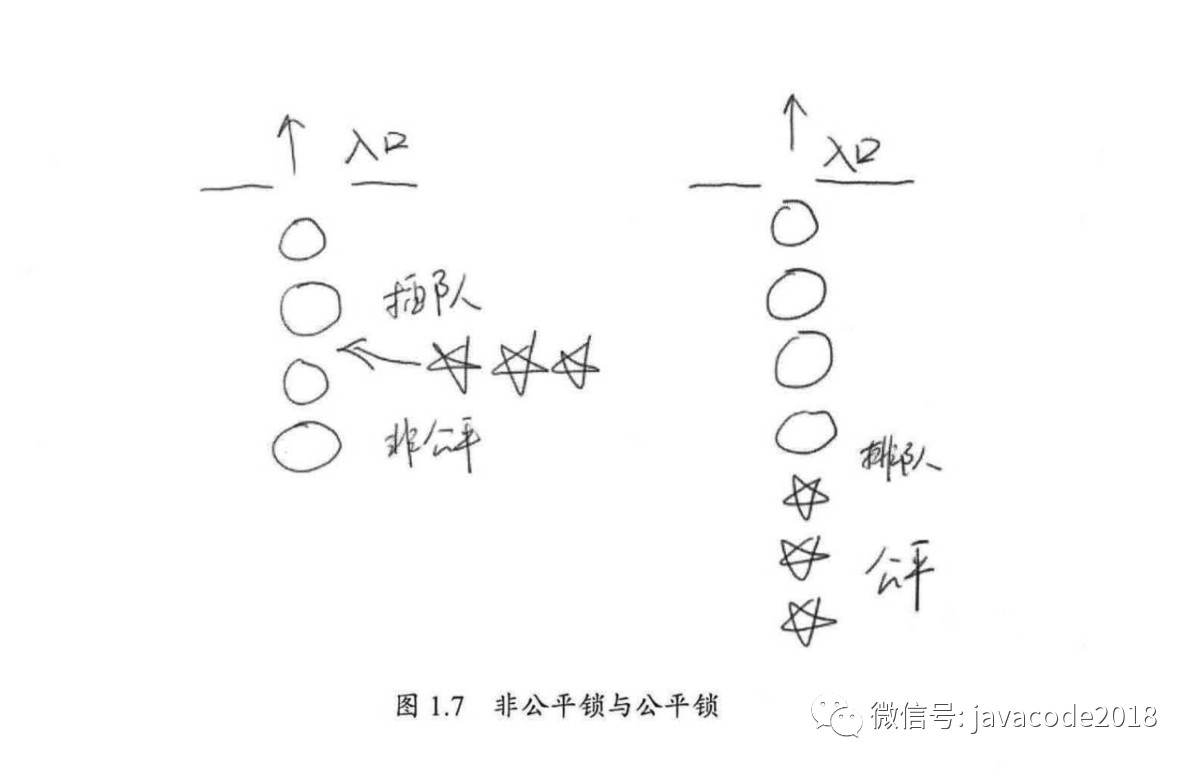
## 阻塞（Block）

一个线程是阻塞的，那么在其他线程释放资源之前，当前线程无法继续执行。当我们使用synchronized关键字或者重入锁时，我们得到的就是阻塞的线程。

synchronize关键字和重入锁都试图在执行后续代码前，得到临界区的锁，如果得不到，线程就会被挂起等待，直到占有了所需资源为止。

## 无饥饿(Starvation-Free)

如果线程之间是有优先级的，那么线程调度的时候总是会倾向于先满足高优先级的线程。也就是说，对于同一个资源的分配，是不公平的！下图中显示了非公平锁与公平锁两种情况(五角星表示高优先级线程)。对于非公平锁来说，系统允许高优先级的线程插队。这样有可能导致低优先级线程产生饥饿。但如果锁是公平的，按照先来后到的规则，那么饥饿就不会产生，不管新来的线程优先级多高，要想获得资源，就必须乖乖排队，这样所有的线程都有机会执行。



## 无障碍(Obstruction-Free)

无障碍是一种最弱的非阻塞调度。两个线程如果无障碍地执行，那么不会因为临界区的问题导致一方被挂起。换言之，大家都可以大摇大摆地进入临界区了。那么大家一起修改共享数据，把数据改坏了怎么办呢？对于无障碍的线程来说，一旦检测到这种情况，它就会立即对自己所做的修改进行回滚，确保数据安全。但如果没有数据竞争发生，那么线程就可以顺利完成自己的工作，走出临界区。

**如果说阻塞的控制方式是悲观策略，也就是说，系统认为两个线程之间很有可能发生不幸的冲突，因此以保护共享数据为第一优先级，相对来说，非阻塞的调度就是一种乐观的策略。它认为多个线程之间很有可能不会发生冲突，或者说这种概率不大。因此大家都应该无障碍地执行，但是一旦检测到冲突，就应该进行回滚。**

从这个策略中也可以看到，**无障碍的多线程程序并不一定能顺畅运行**。因为当临界区中存在严重的冲突时，所有的线程可能都会不断地回滚自己的操作，而没有一个线程可以走出临界区。这种情况会影响系统的正常执行。所以，我们可能会非常希望在这一堆线程中，至少可以有一个线程能够在有限的时间内完成自己的操作，而退出临界区。至少这样可以保证系统不会在临界区中进行无限的等待。

一种可行的无障碍实现可以依赖一个"一致性标记"来实现。线程在操作之前，先读取并保存这个标记，在操作完成后，再次读取，检查这个标记是否被更改过，如果两者是一致的，则说明资源访问没有冲突。如果不一致，则说明资源可能在操作过程中与其他线程冲突，需要重试操作。而任何对资源有修改操作的线程，在修改数据前，都需要更新这个一致性标记，表示数据不再安全。

数据库中乐观锁，应该比较熟悉，表中需要一个字段version(版本号)，每次更新数据version + 1，更新的时候将版本号作为条件进行更新，根据更新影响的行数判断更新是否成功，伪代码如下：

1.查询数据，此时版本号为w\_v

2.打开事务

3.做一些业务操作

4.update t set version = version+1 where id = 记录id and version = w\_v;//此行会返回影响的行数c

5.if(c>0){

//提交事务

}else{

//回滚事务

}

多个线程更新同一条数据的时候，数据库会对当前数据加锁，同一时刻只有一个线程可以执行更新语句。

## 无锁(Lock-Free)

**无锁的并行都是无障碍的**。在无锁的情况下，所有的线程都能尝试对临界区进行访问，但不同的是，无锁的并发保证必然有一个线程能够在有限步内完成操作离开临界区。

**在无锁的调用中，一个典型的特点是可能会包含一个无穷循环**。在这个循环中，线程会不断尝试修改共享变量。如果没有冲突，修改成功，那么程序退出，否则继续尝试修改。但无论如何，无锁的并行总能保证有一个线程是可以胜出的，不至于全军覆没。至于临界区中竞争失败的线程，他们必须不断重试，直到自己获胜。如果运气很不好，总是尝试不成功，则会出现类似饥饿的现象，线程会停止。

下面就是一段无锁的示意代码，如果修改不成功，那么循环永远不会停止。

while(!atomicVar.compareAndSet(localVar, localVar+1)){

localVal = atomicVar.get();

}

## 无等待

无锁只要求有一个线程可以在有限步内完成操作，而无等待则在无锁的基础上更进一步扩展。它要求所有线程都必须在有限步内完成，这样不会引起饥饿问题。如果限制这个步骤的上限，还可以进一步分解为有界无等待和线程数无关的无等待等几种，他们之间的区别只是对循环次数的限制不同。

一种典型的无等待结果就是RCU(Read Copy Update)。它的基本思想是，对数据的读可以不加控制。因此，所有的读线程都是无等待的，它们既不会被锁定等待也不会引起任何冲突。但在写数据的时候，先获取原始数据的副本，接着只修改副本数据(这就是为什么读可以不加控制)，修改完成后，在合适的时机回写数据。