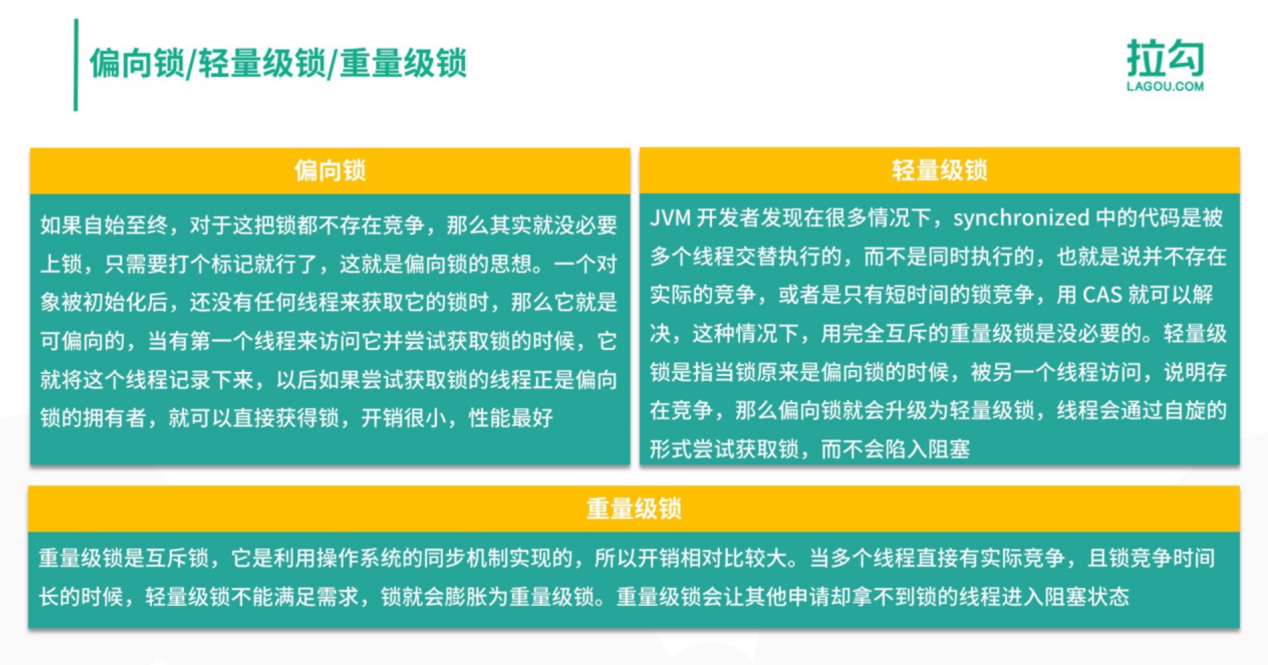
1. **锁的 7 大分类**

* 偏向锁/轻量级锁/重量级锁；



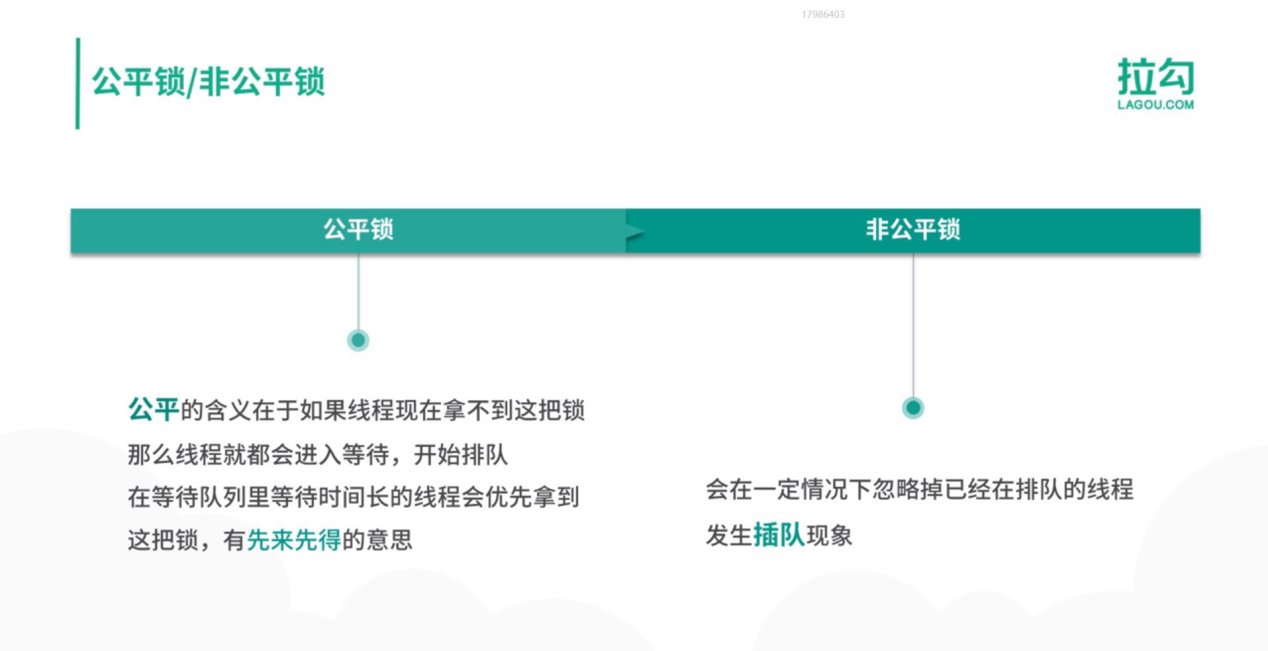
* 可重入锁/非可重入锁；



* 共享锁/独占锁；



* 公平锁/非公平锁；



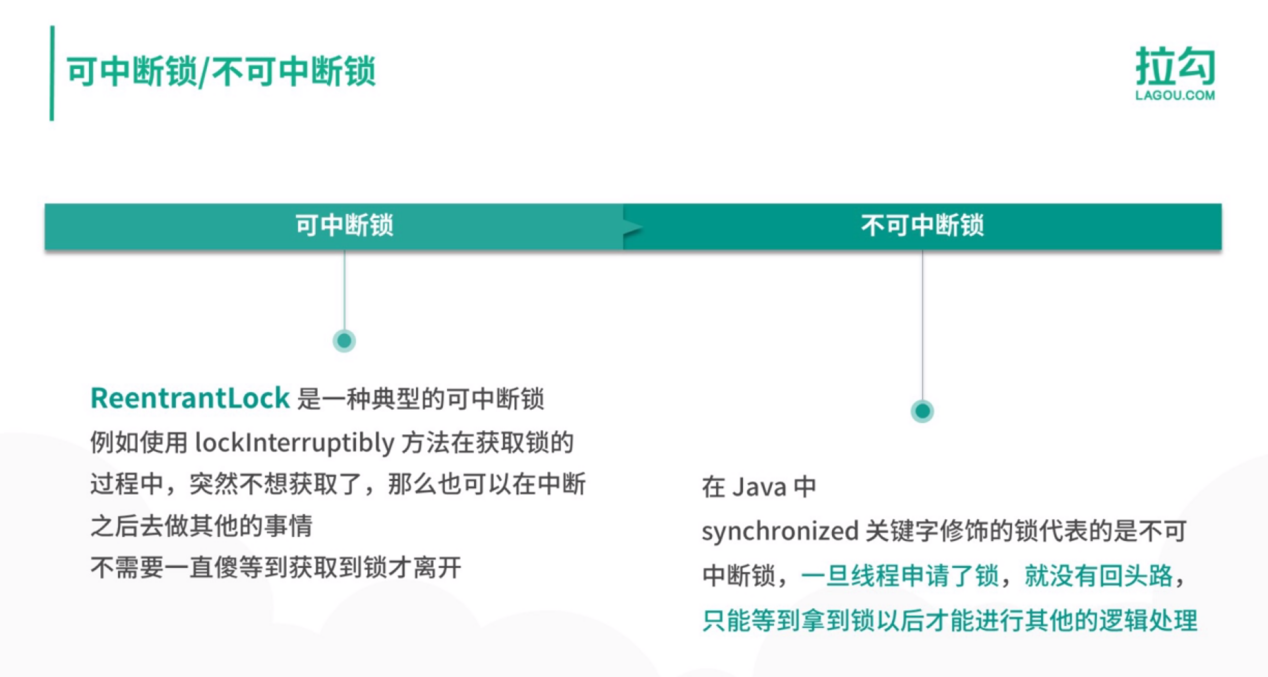
* 悲观锁/乐观锁；



* 自旋锁/非自旋锁；



* 可中断锁/不可中断锁。



1. **悲观锁和乐观锁的本质**

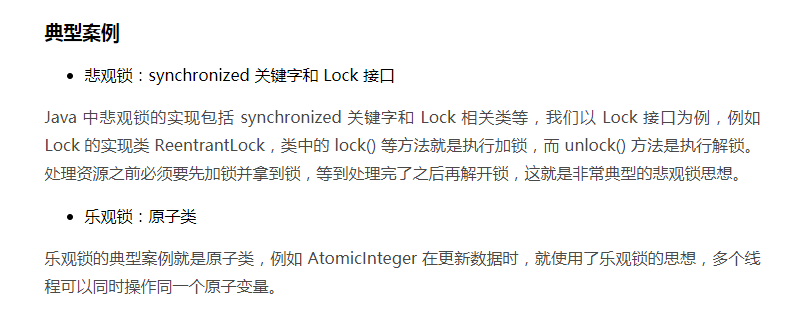
1、悲观锁

需要先获取锁资源，才能操作，悲观锁为了确保结果的正确性，会在每次获取并修改数据时，都把数据锁住，让其他线程无法访问该数据，这样就可以确保数据内容万无一失。

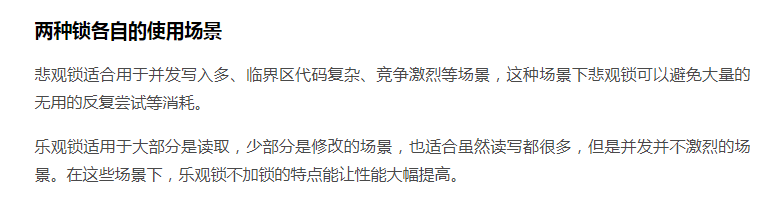
1. 乐观锁

乐观锁不会锁住被操作对象，不会不让别的线程来接触它，同时，为了确保数据正确性，在更新之前，会去对比在我修改数据期间，数据有没有被其他线程修改过：如果没被修改过，就说明真的只有我自己在操作，那我就可以正常的修改数据；如果发现数据和我一开始拿到的不一样了，说明其他线程在这段时间内修改过数据，那说明我迟了一步，所以我会放弃这次修改，并选择报错、重试等策略。

经典案例：



使用场景：



1. **synchronized 背后的“monitor 锁”**

synchronized 的背后正是利用 monitor 锁实现同步的

同步代码块例子：

public class SynTest {

    public void synBlock() {

        synchronized (this) {

            System.out.println("jy");

        }

    }

}

对应的class字节码：

  public void synBlock();

    descriptor: ()V

    flags: ACC\_PUBLIC

    Code:

      stack=2, locals=3, args\_size=1

         0: aload\_0

         1: dup

         2: astore\_1

         3: monitorenter

         4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;

         7: ldc           #3                      // String jy

         9: invokevirtual #4               // Method java/io/PrintStream.println:(Ljava/lang/String;)V

        12: aload\_1

        13: monitorexit

        14: goto          22

        17: astore\_2

        18: aload\_1

        19: monitorexit

        20: aload\_2

        21: athrow

        22: return

分析： synchronized 代码块实际上多了 monitorenter 和 monitorexit 指令，两个 monitorexit 指令的原因是：JVM 要保证每个 monitorenter 必须有与之对应的 monitorexit，monitorenter 指令被插入到同步代码块的开始位置，而 monitorexit 需要插入到方法正常结束处和异常处两个地方，这样就可以保证抛异常的情况下也能释放锁

同步方法例子：

public synchronized void synMethod() {

}

对应的class字节码：

  public synchronized void synMethod();

    descriptor: ()V

    flags: ACC\_PUBLIC, ACC\_SYNCHRONIZED

    Code:

      stack=0, locals=1, args\_size=1

         0: return

      LineNumberTable:

        line 16: 0

分析： synchronized 修饰的方法会有一个 ACC\_SYNCHRONIZED 标志。当某个线程要访问某个方法的时候，会首先检查方法是否有 ACC\_SYNCHRONIZED 标志，如果有则需要先获得 monitor 锁，然后才能开始执行方法，方法执行之后再释放 monitor 锁。其他方面， synchronized 方法和刚才的 synchronized 代码块是很类似的，例如这时如果其他线程来请求执行方法，也会因为无法获得 monitor 锁而被阻塞。

1. **synchronized 和 Lock 孰优孰劣，如何选择？**

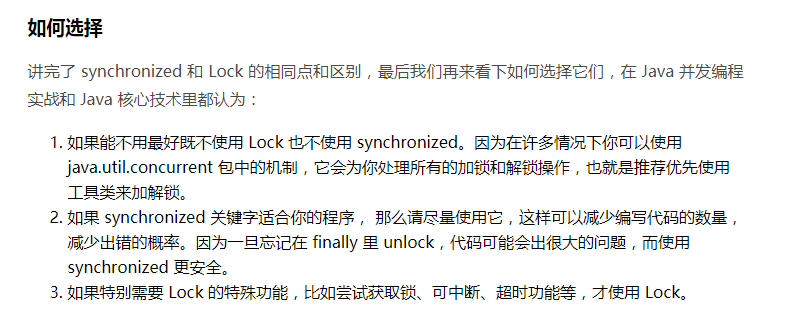
相同点：

* synchronized 和 Lock 都是用来保护资源线程安全的。
* 都可以保证可见性。
* synchronized 和 ReentrantLock 都拥有可重入的特点。

不同点：

* 用法区别
* 加解锁顺序不同
* synchronized 锁不够灵活
* synchronized 锁只能同时被一个线程拥有，但是 Lock 锁没有这个限制
* synchronized 是内置锁，由 JVM 实现获取锁和释放锁的原理，还分为偏向锁、轻量级锁、重量级锁。
* 是否可以设置公平/非公平

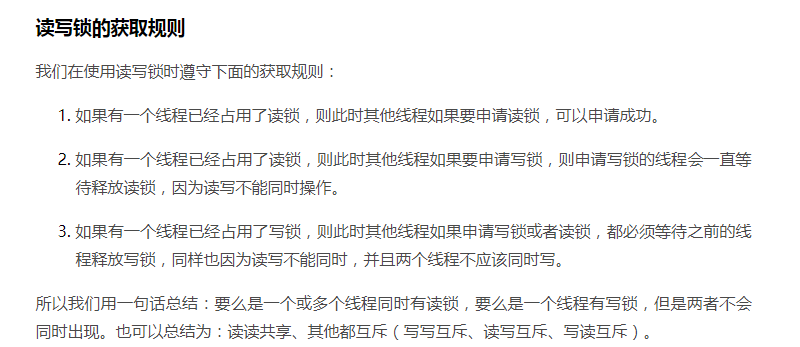
如何选择？



1. **公平和非公平锁**



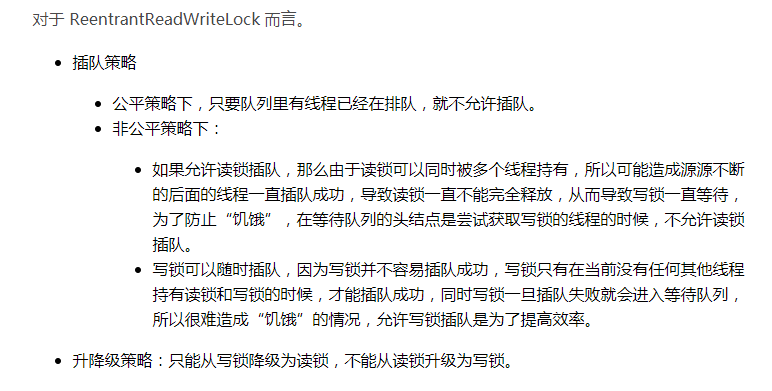
1. **读写锁 ReadWriteLock**
2. 规则



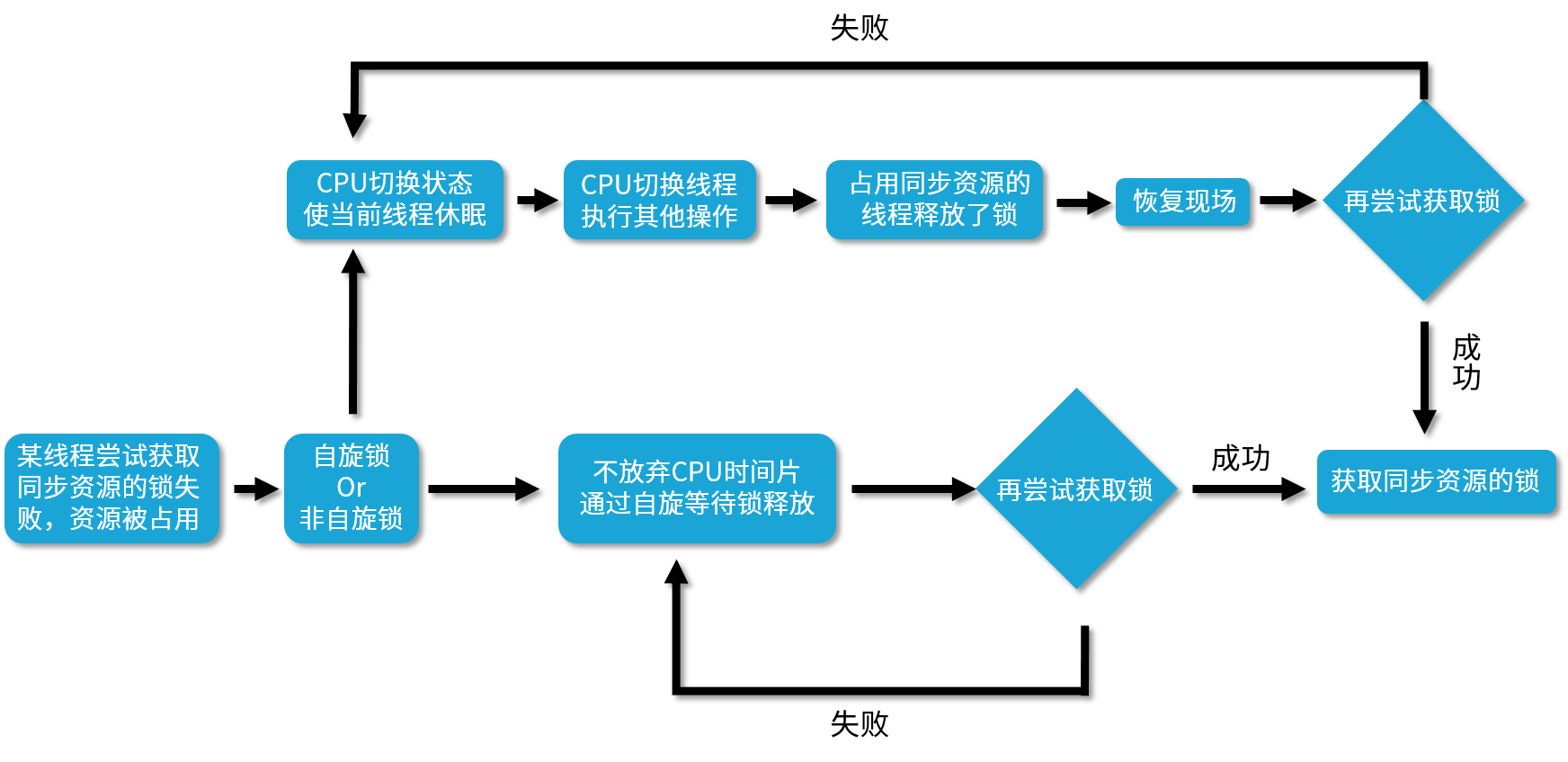
1. 锁降级



1. 策略



1. **自旋锁**
2. 自旋锁和非自旋锁的获取锁的过程



优点：自旋锁用循环去不停地尝试获取锁，让线程始终处于 Runnable 状态，节省了线程状态切换带来的开销。

缺点：虽然避免了线程切换的开销，但是它在避免线程切换开销的同时也带来了新的开销，因为它需要不停得去尝试获取锁。如果这把锁一直不能被释放，那么这种尝试只是无用的尝试，会白白浪费处理器资源。

1. **jvm对锁的优化**

从无锁到偏向锁，再到轻量级锁，最后到重量级锁。结合前面我们讲过的知识，偏向锁性能最好，避免了 CAS 操作。而轻量级锁利用自旋和 CAS 避免了重量级锁带来的线程阻塞和唤醒，性能中等。重量级锁则会把获取不到锁的线程阻塞，性能最差

