多线程合理运用的好处：

- 运用多核心 CPU 和超线程技术，实现多任务多线程并行执行
- 线程异步化执行的优势，提高处理性能和并发量

多线程带来的麻烦：

- 共享资源的 race condition 问题，线程安全问题
- 共享资源会被多个线程访问并改变，可能达不到预期的结果

锁可以将并行的多线程在共享资源处（临界区）串行化

- 锁就是处理并发的一种同步手段
- 互斥锁：一个线程拿到了锁才能进行共享资源操作（进入临界区），其他线程都只能排队等待（阻塞，等待）这个锁，等持有锁的线程释放了，才能再次拿到锁
- synchronized
  - 同步锁 / 互斥锁
  - 早期的 synchronized 是基于重量级锁（1.6 之前）
    - 获得锁时，如果存在线程竞争，线程会被挂起来
    - 后面获得锁的其他线程释放以后，再唤醒挂起来的线程
    - 为什么这是重量级锁?
      - 因为线程的阻塞和唤醒，需要进行内核态、用户态的切换，这种切换是需要操作系统支持的，性能消耗是非常大的
  - 1.6 以后做了优化，引入偏向锁、轻量级锁（自旋锁），可以锁膨胀、锁收缩、锁消除，大幅优化了锁的性能
    - 为什么要进行优化？既要保证数据安全性，又要保证性能
      - 除了控制 synchronized 的锁定粒度以外，1.6 之前没有其他办法，1.6 之后优化的最终目的就是无锁化
    - 数据安全性和性能是存在冲突的
  - 锁，相当于一个线程去访问，拿到唯一的钥匙开锁进去（然后锁关了），后面的线程来了拿不到钥匙开不了锁，只能在门口等待前面的那个线程出来释放锁还回钥匙

锁的设计和底层优化想要达到的目的：

- 数据的安全性
- 性能

synchronized 的基本使用方法：

- 理论上来说有三种，分别代表的锁的控制粒度不同，锁的粒度与锁的性能相关
  - 修饰实例方法
    - 代表当前的锁只锁定当前实例（不同的实例中运行不会冲突）
  - 修饰静态方法
    - 代表当前的锁会锁定当前类（不同的实例（同一个类）会冲突）
  - 修饰代码块
    - 任何对象传入都可以作为锁，可以是任何一个 new 的 Object ，或者当前类的 class
      - 如果传入的是一个 Object ，就是同一个 Object 的作用域范围下的锁
      - 如果是类的 class ，就是全局的，相当于对同一个 class 生效，其实在同一个 JVM 进程中就是全局锁（因为同一个 JVM 进程中这个 class 是唯一的，一般这个进程生命周期结束，这个 class 的生命周期才结束）
        - 所以这种锁的生命周期最大，锁的范围也就最大
- 粒度和作用范围越大，锁的效率越差
- 控制锁的范围是由 synchronized 锁定对象的生命周期决定的

锁应当存储在哪里?

- 唯一控制锁范围的，是 synchronized 锁定的对象
  - 不管是实例对象、当前对象，还是类对象，取决于对象的生命周期
  - 所以，锁应当存储在对象头里
- 对象在内存中是如何存储的？锁的存储一定跟对象在内存中的存储有关系
  - 对象在堆内存里的存储，有一个内存布局，参考：
    https://cloud.tencent.com/developer/article/1339010（如果原链接挂了，可看有道云笔记收藏：https://note.youdao.com/web/#/file/recent/note/wcp1587702313800 91/）
    - 对象头 Object Header（参考 hotspot/src/share/vm/oops/markOop.hpp 中的注释说明）
      - Mark Word： 主要用来存储对象自身的运行时数据（下面是无锁和偏向锁状态，lock 标志都是 01，偏向锁状态偏向锁标记为 1，可以认为偏向锁也是一种无锁化处理优化）
        - HashCode
        - GC 分代年龄 age（4 bit，最大 15）
        - 偏向锁标记
        - 锁标记
      - Klass Word （Class Metadata Address）： 存储到对象类型数据的指针
      - Array Length （如果是数组的话）： 数组的长度
    - 实例数据
    - 对齐填充

```
// Bit-format of an object header (most significant first, big endian layout below
//
//  32 bits:
//  --------
//             hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal object
//             JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object
//             size:32 ------------------------------------------>| (CMS free bloc
//             PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted
//
//  64 bits:
//  --------
//  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal objec
//  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased objec
//  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promoted
//  size:64 ----------------------------------------------------->| (CMS free blo
//
//  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && norm
//  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && bias
//  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CMS
//  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CMS
```

上面的源码注释用图表示如下（下图中是 32 位 JVM）：

锁状态	25 bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC标记	空				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01

无锁状态 ： 对象的HashCode + 对象分代年龄 + 状态位001

- Epoch ： 偏向时间戳（即以 0 表示 1970-01-01T00:00:00Z），精确到纳秒
  - Epoch 可以认为是一个时钟周期，通过 Epoch 这个时钟周期去保证锁对象的升级，老的 Epoch 就无法获得新的 Epoch 的偏向锁，就只能撤销偏向锁
  - JVM 用一个 epoch 表示一个偏向锁的时间戳（真实地生成一个时间戳代价还是蛮大的，因此这里应当理解为一种类似时间戳的 identifier），对 epoch ，官方是这么解释的： A similar mechanism, called bulk rebiasing, optimizes situations in which objects of a class are locked and unlocked by different threads but never concurrently. It invalidates the bias of all instances of a class without disabling biased locking. An epoch value in the class acts as a timestamp that indicates the validity of the bias. This value is copied into the header word upon object allocation. Bulk rebiasing can then efficiently be implemented as an increment of the epoch in the appropriate class. The next time an instance of this class is going to be locked, the code detects a different value in the header word and rebiases the object towards the current thread. 参考：https://cloud.tencent.com/developer/article/1339010
- 其他具体分析和字段说明可参考：https://www.jianshu.com/p/3d38cba67f8b

无锁状态，如下图，biased_lock 0 ， lock 01 ， 所以状态位为 001

biased_lock	lock	状态
0	01	无锁
1	01	偏向锁
0	00	轻量级锁
0	10	重量级锁
0	11	GC标记

什么时候去触发锁的记录的?

- 先看 JVM 源码中的 hotspot/src/share/vm/oops/instanceOop.hpp

```
// An instanceOop is an instance of a Java Class
// Evaluating "new HashTable()" will create an instanceOop.

class instanceOopDesc : public oopDesc {
    ......
}
```

- instanceOop 对应的是一个 Java 类的实例，继承自 oopDesc ，进入 hotspot/src/share/vm/oops/oop.hpp 中，可以看到

```
class oopDesc {
    friend class VMStructs;
  private:
    volatile markOop  _mark;
    union _metadata {
        Klass*      _klass;
        narrowKlass _compressed_klass;
    } _metadata;
    ……
}
```

- 其中，volatile markOop _mark 是对象头（就是前面说明的对象头），union _metadata {……} _metadata 是元数据

根据 synchronized 的使用与否和实际锁状态，可以划分出 4 种：

- 无锁
- 偏向锁
- 轻量级锁
- 重量级锁

划分以上 4 种状态的目的，就是在加锁保证安全性的同时，兼顾性能

- 我们可以通过控制 synchronized 锁对象来控制锁的粒度，比如 new 一个 Object lock 传入若干个实例对象，对它们进行加锁，而另外的实例对象可以用其他 lock 对象加锁，这样就可以按需要控制锁的粒度
- 这就相当于可以加锁加在单元门（new 的 Object lock 对象），就不用加锁加在小区大门（class 对象）

synchronized 在 JDK 1.6 以后做了优化

- 1.6 之前，基于重量级锁实现的
  - 重量级锁就是存在多线程竞争的情况下，去获得锁，没有获得锁的线程会被挂起来，阻塞
  - 挂起来以后，后面线程释放锁以后，会唤醒其他等待获取锁的线程
  - 因为线程的阻塞和唤醒需要进行内核态和用户态的切换，这个切换需要操作系统来支持，性能消耗非常大
    - 线程的阻塞和唤醒需要 CPU 从用户态转为核心态，频繁的阻塞和唤醒对 CPU 来说是一件负担很重的工作
    - Java SE 1.6 为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在 Java SE 1.6 里锁一共有四种状态
      - 无锁状态
      - 偏向锁状态
      - 轻量级锁状态
      - 重量级锁状态
    - 它会随着竞争情况逐渐升级
    - 锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁
      - 这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率
- 如何优化？既能保证数据安全，又能保证性能？
  - 除了基于 synchronized 的锁的粒度的控制之外，没有其他方法。
  - 能不能尽量不加锁（无锁化），也能保证数据安全性？
    - 所以引入了偏向锁、轻量级锁
  - 可以认为重量级锁才是真正意义上的加锁
    - 偏向锁、轻量级锁可以认为是无锁状态
  - 如何实现无锁编程达到安全性？所以引入了锁的升级（锁膨胀）

如果有一块同步代码块：

```
synchronized(lock) {
    // 同步代码块
}
// 有两个线程 ThreadA / ThreadB
```

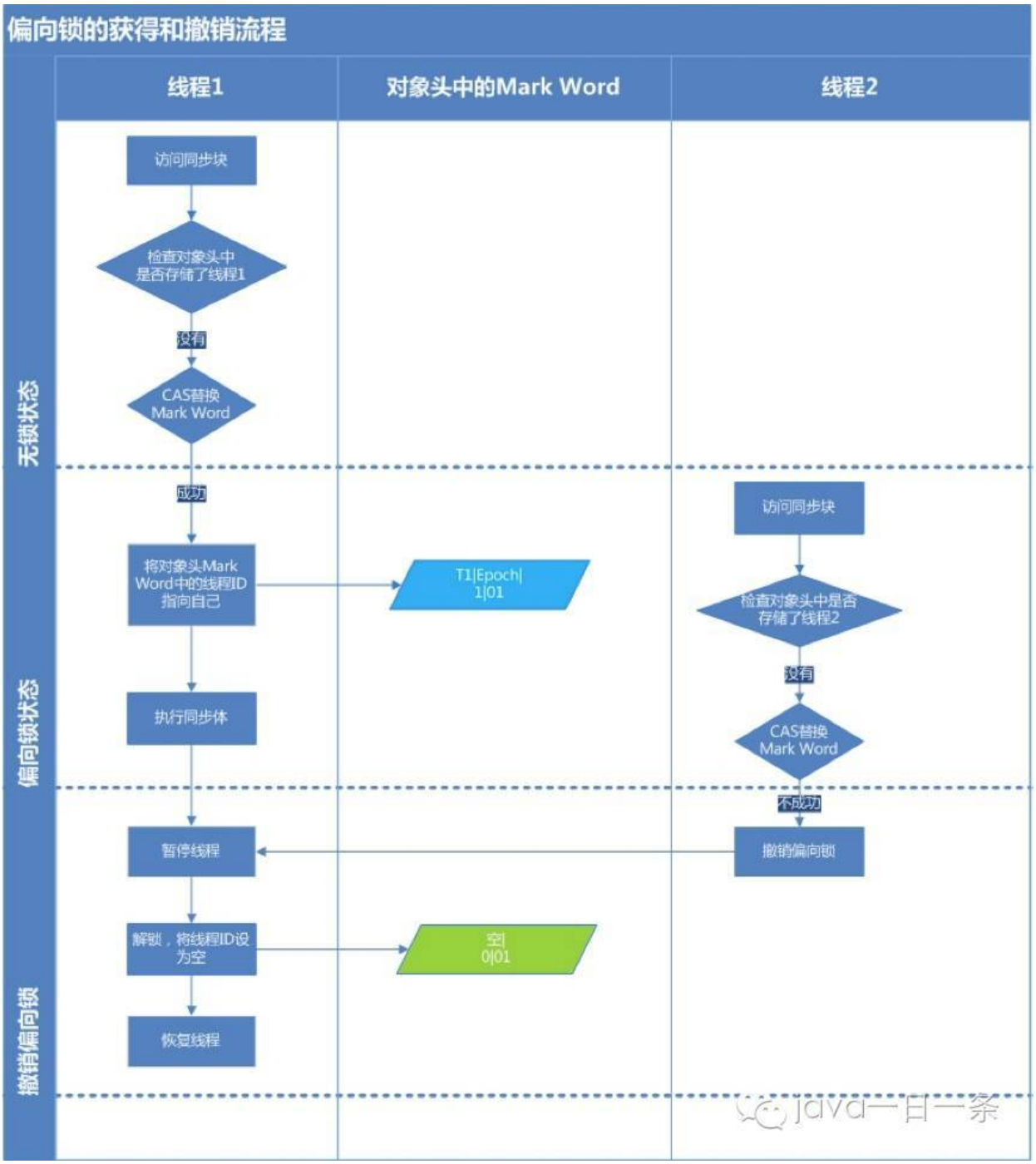
看以下几种情况：

- 只有 ThreadA 去访问
  - （绝大部分情况下是这种情况）所以引入了偏向锁，可参考理解：https://cloud.tencent.com/developer/article/1339010
  - 如果每一次访问都是 ThreadA 获取到这个锁，那么在 lock 对象的对象头 Mark Word 里，就存储 ThreadA 的偏向锁
    - 存储 ThreadA 的线程 ID （ThreadID）
    - 偏向锁标记 1
  - 下一次 ThreadA 再去访问时，就不需要再获取锁了
    - ThreadA 只需要比较 lock 对象头 Mark Word 里的 ThreadID 跟 ThreadA 的一致，并且当前偏向锁标记是 1
  - 我们写的代码里面如果没有当前这种情况，就可以关闭偏向锁，默认就走轻量级锁
- ThreadA 和 ThreadB 交替访问
  - ThreadB 来了，发现 ThreadA 获得了偏向锁，此时需要暂停 ThreadA ，释放偏向锁，重新成为无锁状态，然后加轻量级锁（自旋锁，尝试获取锁）
- 多个线程同时并行访问
  - 轻量级锁也解决不了多个线程激烈的竞争，就只有阻塞拿不到锁的线程，让它们挂起
    - 基于操作系统 mutex 挂起
  - 此时就升级为了重量级锁

上面是初步介绍锁的升级，可以体会到杀鸡焉用牛刀，能更高性能无锁解决线程安全问题，就不用更重的锁。

- 用相对合适的方式，解决加锁的性能问题，锁升级

下面首先看偏向锁，参考：https://cloud.tencent.com/developer/article/1339010


偏向锁的获得和撤销流程
线程1
对象头中的Mark Word
线程2
无锁状态
偏向锁状态
撤销偏向锁
访问同步块
检查对象头中是否存储了线程1
没有
CAS替换Mark Word
成功
将对象头Mark Word中的线程ID指向自己
T1|Epoch|1|01
执行同步体
访问同步块
检查对象头中是否存储了线程2
没有
CAS替换Mark Word
不成功
暂停线程
撤销偏向锁
解锁，将线程ID设为空
空|0|01
恢复线程
javd一日一条