《Java 性能优化实战 21 讲》

李国 前京东、陌陌高级架构师

— 拉勾教育出品 —

13｜案例分析：多线程锁的优化

案例分析：多线程锁的优化

上一课时

使用 ThreadLocal，来避免 SimpleDateFormat 在并发环境下引起的时间错乱问题

案例分析：多线程锁的优化

对 parse 方法进行加锁

```
public class ThreadSafeDateFormat {
    SimpleDateFormat format = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss");

    public static void main(String[] args) {
        ThreadSafeDateFormat threadSafeDateFormat = new ThreadSafeDateFormat();
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i = 0; i < 1000; i++) {
            executor.submit(() -> {
                try {
                    synchronized (threadSafeDateFormat) {
                        System.out.println(threadSafeDateFormat.format.parse( source: "2020-07-25 08:56:40")
                    }
                } catch (ParseException e) {
                    throw new IllegalStateException();
                }
            });
        }
        executor.shutdown();
    }
}
```

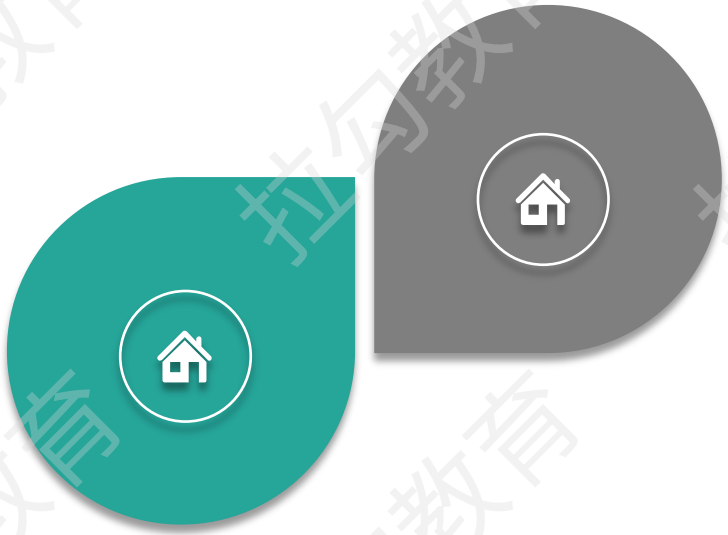
案例分析：多线程锁的优化

开启了 50 个线程，使用 ThreadLocal 和同步锁方式性能的一个对比

```
Benchmark                                  Mode  Cnt    Score   Error  Units
SynchronizedNormalBenchmark.sync           thrpt   10 2554.628 ± 5098.059  ops/ms
SynchronizedNormalBenchmark.threadLocal thrpt   10 3750.902 ±  103.528  ops/ms
========去掉业务影响========
Benchmark                                  Mode  Cnt     Score     Error  Units
SynchronizedNormalBenchmark.sync           thrpt   10  26905.514 ±  1688.600  ops/ms
SynchronizedNormalBenchmark.threadLocal thrpt   10 7041876.244 ± 355598.686  ops/ms
```

案例分析：多线程锁的优化

Java 中有两种加锁的方式



synchronized 关键字

concurrent 包里面的 Lock

synchronied

给普通方法加锁时
上锁的对象是 this

给静态方法加锁时
锁的是 class 对象

给代码块加锁
可以指定一个具体的对象作为锁

monitor 原理

monitor 原理

给方法加了一个 flag：ACC_SYNCHRONIZED

```
synchronized void syncMethod() {
    System.out.println("syncMethod");
}
======字节码=====
synchronized void syncMethod();
  descriptor: ()V
  flags: ACC_SYNCHRONIZED
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #4
       3: ldc           #5
       5: invokevirtual #6
       8: return
```

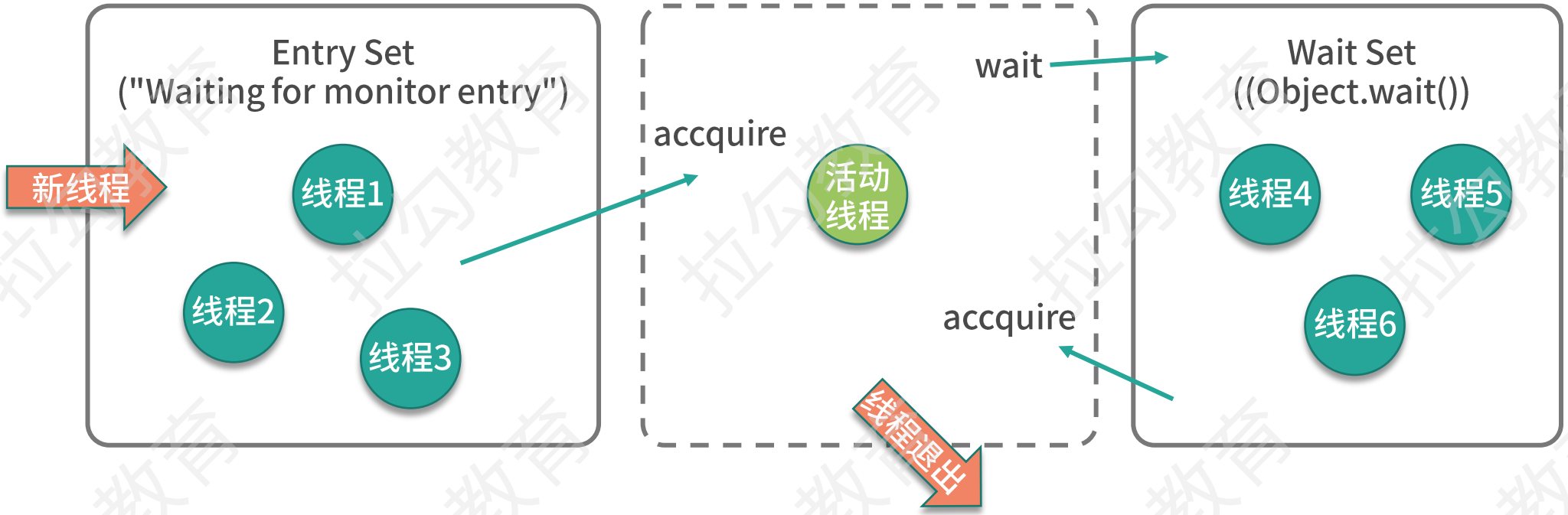
monitor 原理


```
void syncBlock(){
    synchronized (Test.class){
    }
}
=======字节码=======
void syncBlock();
  descriptor: ()V
  flags:
  Code:
    stack=2, locals=3, args_size=1
       0: ldc           #2
       2: dup
       3: astore_1
       4: monitorenter
       5: aload_1
       6: monitorexit
```

monitor 原理


```
 2: dup
 3: astore_1
 4: monitorenter
 5: aload_1
 6: monitorexit
 7: goto       15
10: astore_2
11: aload_1
12: monitorexit
13: aload_2
14: athrow
15: return
Exception table:
 from   to  target type
    5    7   10   any
   10   13   10   any
```

monitor 原理


Entry Set
("Waiting for monitor entry")
新线程
线程1
线程2
线程3
accquire
活动
线程
wait
accquire
线程退出
Wait Set
((Object.wait())
线程4
线程5
线程6

monitor 原理


```
"http-nio-8084-exec-120" #143 daemon prio=5 os_prio=31 cpu=122.86ms elapsed=317.88s
tid=0x00007fedd8381000 nid=0x1af03 waiting for monitor entry  [0x00007000150e1000]
   java.lang.Thread.State: BLOCKED (on object monitor)
    at java.io.BufferedInputStream.read(java.base@13.0.1/BufferedInputStream.java:263)
    - waiting to lock <0x0000000782e1b590> (a java.io.BufferedInputStream)
    at org.apache.commons.httpclient.HttpParser.readRawLine(HttpParser.java:78)
    at org.apache.commons.httpclient.HttpParser.readLine(HttpParser.java:106)
    at org.apache.commons.httpclient.HttpConnection.readLine(HttpConnection.java:1116)
    at org.apache.commons.httpclient.HttpMethodBase.readStatusLine(HttpMethodBase.java:1973)
    at org.apache.commons.httpclient.HttpMethodBase.readResponse(HttpMethodBase.java:1735)
```

monitor 原理


```
synchronized (lock){
  try {
      lock.wait();
  } catch (InterruptedException e){
      e.printStackTrace();
  }
}
```

monitor 原理


```
"wait-demo" #12 prio=5 os_prio=31 cpu=0.14ms elapsed=12.58s
tid=0x00007fb66609e000 nid=0x6103 in Object.wait()  [0x000070000f2bd000]
   java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(java.base@13.0.1/Native Method)
    - waiting on <0x0000000787b48300> (a java.lang.Object)
    at java.lang.Object.wait(java.base@13.0.1/Object.java:326)
    at WaitDemo.lambda$main$0(WaitDemo.java:7)
    - locked <0x0000000787b48300> (a java.lang.Object)
    at WaitDemo$$Lambda$14/0x0000000800b44840.run(Unknown Source)
    at java.lang.Thread.run(java.base@13.0.1/Thread.java:830)
```

monitor 原理

WaitSet 中的线程是如何再次被激活的呢？

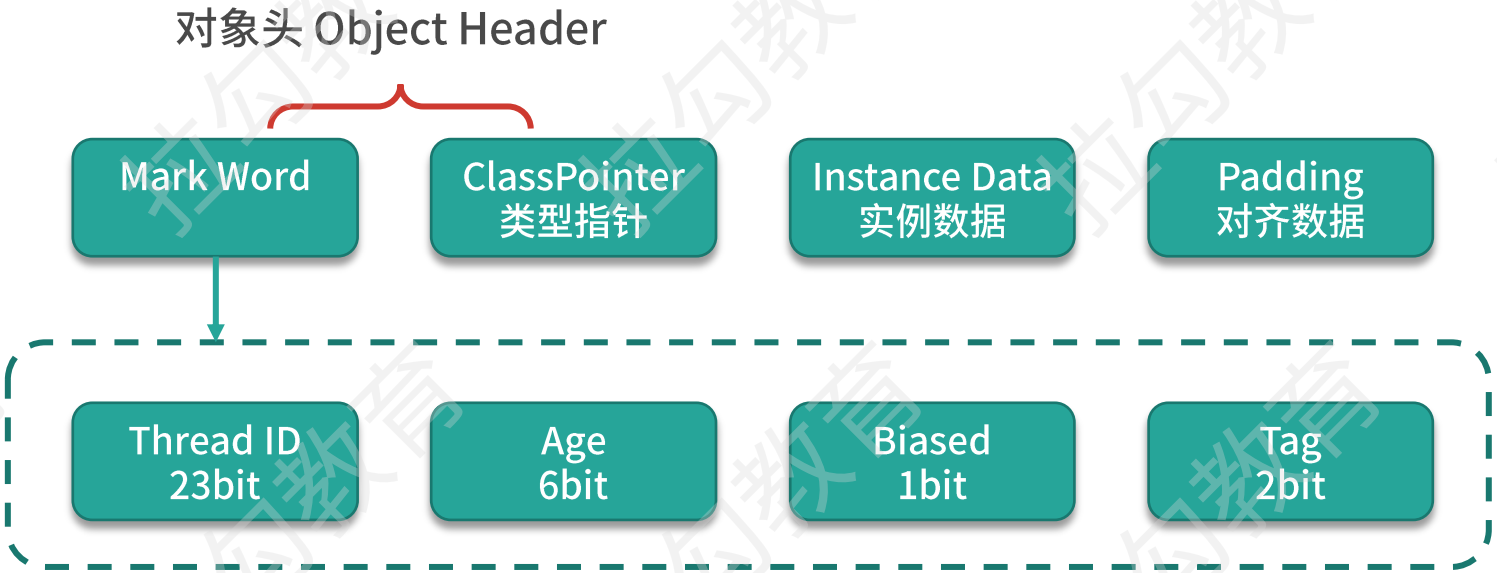
在某个地方，执行了锁的 notify 或者 notifyAll 命令

会造成 WaitSet 中的线程，转移到 EntrySet 中，重新进行锁的争夺

分级锁

JVM 会根据使用情况，对 synchronized 的锁，进行升级

偏向锁 — 轻量级锁 — 重量级锁



偏向锁

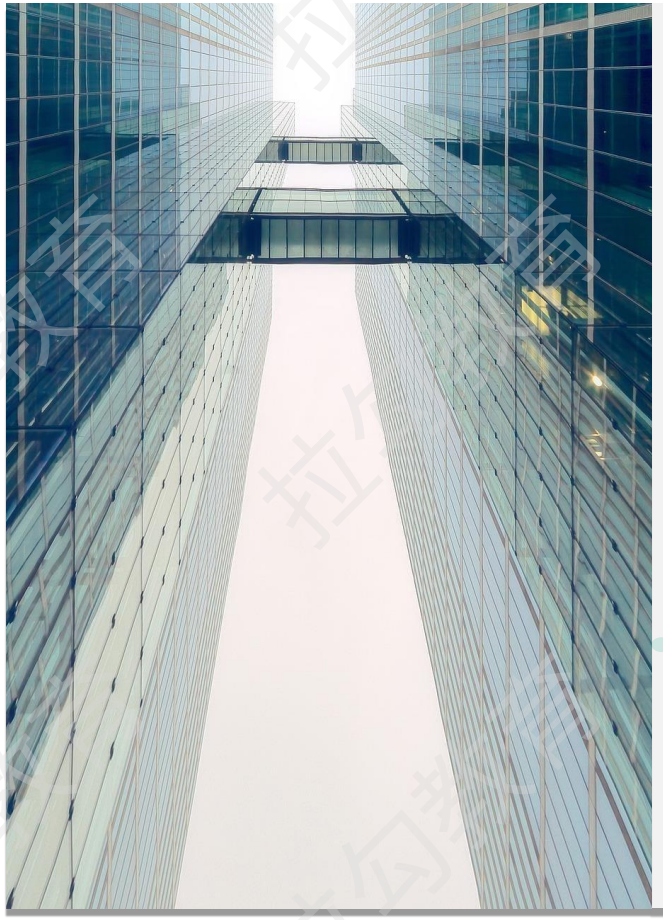
在只有一个线程使用了锁的情况下，偏向锁能够保证更高的效率

当第一个线程第一次访问同步块时，会先检测对象头 Mark Word 中的标志位 Tag 是否为 01

01 是锁默认的状态，线程一旦获取了这把锁，就会把自己的线程 ID 写到 MarkWord 中

判断 MarkWord 中保存的线程 ID 是否与这个线程 ID 相等
不相等，会立即撤销偏向锁，升级为轻量级锁

轻量级锁



轻量级锁

轻量级锁使用自旋方式获取

参与竞争的每个线程，会在线程栈中生成一个 LockRecord (LR)

每个线程通过 CAS（自旋）的方式

将锁对象头中的 MarkWord 设置为指向自己的 LR 的指针

当锁处于轻量级锁的状态时，每次对锁的获取，都需要通过自旋

自旋是面向不存在锁竞争的场景

重量级锁

重量级锁——对 synchronized 的直观认识

线程会挂起，进入到操作系统内核态，等待操作系统的调度，然后再映射回用户态

如果并发非常严重，可以通过参数 -XX:-UseBiasedLocking 禁用偏向锁

Lock

在 concurrent 包里，有 ReentrantLock 和 ReentrantReadWriteLock 两个类

可重入：一个线程运行时，可以多次获取同一个对象锁

```
public synchronized void a(){
    b();
}
public synchronized void b(){
    c();
}
public synchronized void c(){
}
```

主要方法

Lock 需要手动加锁，然后在 finally 中解锁

Lock

如果获取不到锁，都会被阻塞

tryLock

尝试获取锁，获取到锁就会返回 true

tryLock

在拿不到锁的情况下

会等待一段时间，直到超时

LockInterruptibly

可以锁等待，可以被中断

中断后返回 InterruptedException

读写锁

ReentrantReadWriteLock 允许多个读线程同时进行，但读和写、写和写是互斥的

```
ReentrantReadWriteLock lock = new
ReentrantReadWriteLock();
  Lock readLock = lock.readLock();
  Lock writeLock = lock.writeLock();

  public void put(K k, V v) {
    writeLock.lock();
    try {
      map.put(k, v);
    } finally {
      writeLock.unlock();
    }
  }
...
```

读写锁

除了 ReadWriteLock，还能有更快的读写分离模式吗？

JDK 1.8 加入了哪个 API？

公平锁与非公平锁

非公平锁

当持有锁的线程释放锁时，EntrySet 里的线程就会争抢这把锁

某个线程通过 setPriority 设置得比较低的优先级

这个抢不到锁的线程，就一直处于饥饿状态

公平锁与非公平锁

- 公平锁

在Lock 中可以通过构造参数设置成公平锁

```
public ReentrantReadWriteLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
    readerLock = new ReadLock(this);
    writerLock = new WriteLock(this);
}
```

公平锁与非公平锁

- 公平锁

Benchmark	Mode	Cnt	Score	Error	Units
FairVSNoFairBenchmark.fair	thrpt	10	186.144 ±	27.462	ops/ms
FairVSNoFairBenchmark.nofair	thrpt	10	35195.649 ±	6503.375	ops/ms

锁的优化技巧——死锁


```
public class DeadLockDemo {
  public static void main(String[] args) {
    Object object1 = new Object();
    Object object2 = new Object();
    Thread t1 = new Thread(() -> {
      synchronized (object1) {
        try {
          Thread.sleep(200);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        synchronized (object2) {
        }
      }
    }, "deadlock-demo-1");

    t1.start();
    Thread t2 = new Thread(() -> {
      synchronized (object2) {
        try {
```

锁的优化技巧——死锁

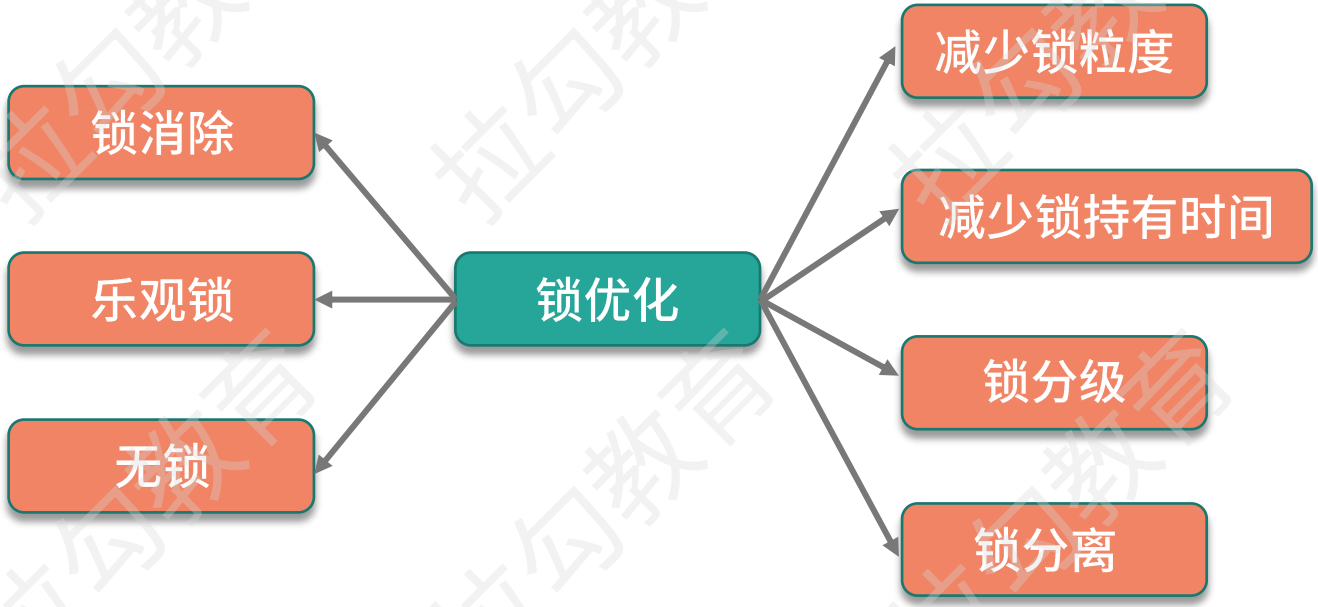

```
            synchronized (object2) {
            }
        }
    }, "deadlock-demo-1");

    t1.start();
    Thread t2 = new Thread(() -> {
        synchronized (object2) {
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (object1) {
            }
        }
    }, "deadlock-demo-2");
    t2.start();
  }
}
```

锁的优化技巧——优化技巧

锁的优化理论——减少锁的冲突

本质——为了避免多个线程同时获取同一把锁



减少锁粒度


```
public class LockLessDemo {
   List<String> list1 = new ArrayList<>();
   List<String> list2 = new ArrayList<>();
   public synchronized void addList1(String v){
      this.list1.add(v);
   }
   public synchronized void addList2(String v){
      this.list2.add(v);
   }
}
```

减少锁粒度


```
public class LockLessDemo {
    List<String> list1 = new ArrayList<>();
    List<String> list2 = new ArrayList<>();
    final Object lock1 = new Object();
    final Object lock2 = new Object();
    public void addList1(String v) {
        synchronized (lock1) {
            this.list1.add(v);
        }
    }
    public void addList2(String v) {
        synchronized (lock2) {
            this.list2.add(v);
        }
    }
}
```

减少锁持有时间


```
public class LockTimeDemo {
    List<String> list = new ArrayList<>();
    final Object lock = new Object();
    public void addList(String v) {
        synchronized (lock) {
            slowMethod();
            this.list.add(v);
        }
    }
    public void slowMethod(){
    }
}
```

锁的优化技巧——优化技巧

拉勾教育
互联网人实战大学

锁分级

锁分级，是Synchronied锁的锁升级，属于 JVM 的内部优化，它从偏向锁开始，逐渐升级为轻量级锁、重量级锁

锁分离

读操作一般是不会对资源产生影响的，可以并发执行；写操作和其他操作是互斥的，只能排队执行。读写锁适合读多写少的场景

锁消除

通过 JIT 编译器，JVM 可以消除某些对象的加锁操作

锁的优化技巧——优化技巧


```
String m1(){
    StringBuffer sb = new StringBuffer();
    sb.append("");
    return sb.toString();
}
```

小结

类别	Synchronized	Lock
实现方式	monitor	AQS
底层细节	JVM优化	Java API
分级锁	是	否
功能特性	单一	丰富
锁分离	无	读写锁
锁超时	无	带超时时间的 tryLock
可中断	否	lockInterruptibly

小结

如果只用最简单的锁互斥功能，建议直接使用 Synchronized

- Synchronized 的编程模型更加简单，更易于使用
- Synchronized 引入了偏向锁，轻量级锁等功能，能够从 JVM 层进行优化

  同时JIT 编译器也会对它执行一些锁消除动作

Next：14 |《案例分析：乐观锁和无锁》

拉勾教育

—互联网人实战大学—

关注拉勾「教育公众号」
获取更多课程信息

L / A / G / O / U