实战Java高并发程序设计第9周

DATAGURU专业数据分析社区

法律声明

【声明】本视频和幻灯片为炼数成金网络课程的教学资料，所有资料只能在课程内使用，不得在课程以外范围散播，违者将可能被追究法律和经济责任。

课程详情访问炼数成金培训网站

http://edu.dataguru.cn

炼数成金逆向收费式网络课程

- Dataguru（炼数成金）是专业数据分析网站，提供教育，媒体，内容，社区，出版，数据分析业务等服务。我们的课程采用新兴的互联网教育形式，独创地发展了逆向收费式网络培训课程模式。既继承传统教育重学习氛围，重竞争压力的特点，同时又发挥互联网的威力打破时空限制，把天南地北志同道合的朋友组织在一起交流学习，使到原先孤立的学习个体组合成有组织的探索力量。并且把原先动辄成千上万的学习成本，直线下降至百元范围，造福大众。我们的目标是：低成本传播高价值知识，构架中国第一的网上知识流转阵地。
- 关于逆向收费式网络的详情，请看我们的培训网站 http://edu.dataguru.cn

内容提要

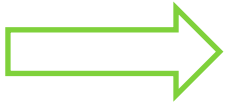
- 锁优化的思路和方法
- 虚拟机内的锁优化
- 一个错误使用锁的案例
- ThreadLocal及其源码分析

锁优化的思路和方法

- 减少锁持有时间
- 减小锁粒度
- 锁分离
- 锁粗化
- 锁消除

减少锁持有时间

```
public synchronized void syncMethod(){
        othercode1();
        mutextMethod();
        othercode2();
}
```



```
public void syncMethod2(){
        othercode1();
        synchronized(this){
                mutextMethod();
        }
        othercode2();
}
```

减小锁粒度

- 将大对象，拆成小对象，大大增加并行度，降低锁竞争
- 偏向锁，轻量级锁成功率提高
- ConcurrentHashMap
- HashMap的同步实现
  - Collections.synchronizedMap(Map<K,V> m)
  - 返回SynchronizedMap对象

```
 public V get(Object key) {
            synchronized (mutex) {return m.get(key);}
        }
public V put(K key, V value) {
            synchronized (mutex) {return m.put(key, value);}
}
```

减小锁粒度

- ConcurrentHashMap
  - 若干个Segment ：Segment<K,V>[] segments
  - Segment中维护HashEntry<K,V>
  - put操作时
    - 先定位到Segment，锁定一个Segment，执行put
- 在减小锁粒度后， ConcurrentHashMap允许若干个线程同时进入

锁分离

- 根据功能进行锁分离
- ReadWriteLock
- 读多写少的情况，可以提高性能

	读锁	写锁
读锁	可访问	不可访问
写锁	不可访问	不可访问

锁分离

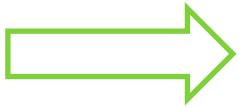
- 读写分离思想可以延伸，只要操作互不影响，锁就可以分离
- LinkedBlockingQueue
  - 队列
  - 链表

锁粗化

- 通常情况下，为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽量短，即在使用完公共资源后，应该立即释放锁。只有这样，等待在这个锁上的其他线程才能尽早的获得资源执行任务。但是，凡事都有一个度，如果对同一个锁不停的进行请求、同步和释放，其本身也会消耗系统宝贵的资源，反而不利于性能的优化

锁粗化

```
public void demoMethod(){
        synchronized(lock){
                //do sth.
        }
        //做其他不需要的同步的工作，但能很快执行完毕
        synchronized(lock){
                //do sth.
        }
}
```



```
public void demoMethod(){
                //整合成一次锁请求
        synchronized(lock){
                //do sth.
                //做其他不需要的同步的工作，但能很快执行完毕
        }
}
```

锁粗化

```
for(int i=0;i<CIRCLE;i++){
        synchronized(lock){

        }
}
```

```
synchronized(lock){
for(int i=0;i<CIRCLE;i++){

        }
}
```

锁消除

- 在即时编译器时，如果发现不可能被共享的对象，则可以消除这些对象的锁操作

```
public static void main(String args[]) throws InterruptedException {
        long start = System.currentTimeMillis();
        for (int i = 0; i < CIRCLE; i++) {
                craeteStringBuffer("JVM", "Diagnosis");
        }
        long bufferCost = System.currentTimeMillis() - start;
        System.out.println("craeteStringBuffer: " + bufferCost + " ms");
}

public static String craeteStringBuffer(String s1, String s2) {
        StringBuffer sb = new StringBuffer();
        sb.append(s1);
        sb.append(s2);
        return sb.toString();
}
```

同步操作

锁消除

CIRCLE= 2000000

-server -XX:+DoEscapeAnalysis -XX:+EliminateLocks

craeteStringBuffer: 187 ms

-server -XX:+DoEscapeAnalysis -XX:-EliminateLocks

craeteStringBuffer: 254 ms

虚拟机内的锁优化

- 偏向锁
- 轻量级锁
- 自旋锁

对象头Mark

- Mark Word，对象头的标记，32位
- 描述对象的hash、锁信息，垃圾回收标记，年龄
  - 指向锁记录的指针
  - 指向monitor的指针
  - GC标记
  - 偏向锁线程ID

偏向锁

- 大部分情况是没有竞争的，所以可以通过偏向来提高性能
- 所谓的偏向，就是偏心，即锁会偏向于当前已经占有锁的线程
- 将对象头Mark的标记设置为偏向，并将线程ID写入对象头Mark
- 只要没有竞争，获得偏向锁的线程，在将来进入同步块，不需要做同步
- 当其他线程请求相同的锁时，偏向模式结束
- -XX:+UseBiasedLocking
  - 默认启用
- 在竞争激烈的场合，偏向锁会增加系统负担

偏向锁

```
public static List<Integer> numberList =new Vector<Integer>();
public static void main(String[] args) throws InterruptedException {
        long begin=System.currentTimeMillis();
        int count=0;
        int startnum=0;
        while(count<10000000){
                numberList.add(startnum);
                startnum+=2;
                count++;
        }
        long end=System.currentTimeMillis();
        System.out.println(end-begin);
}
```

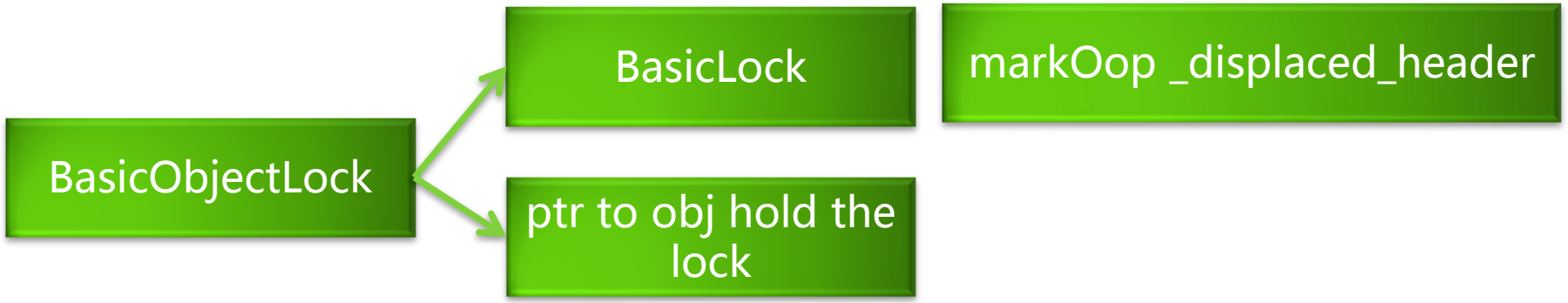
本例中，使用偏向锁，可以获得5%以上的性能提升

```
-XX:+UseBiasedLocking -XX:BiasedLockingStartupDelay=0
```

```
-XX:-UseBiasedLocking
```

轻量级锁

- BasicObjectLock
  - 嵌入在线程栈中的对象



轻量级锁

- 普通的锁处理性能不够理想，轻量级锁是一种快速的锁定方法。
- 如果对象没有被锁定
  - 将对象头的Mark指针保存到锁对象中
  - 将对象头设置为指向锁的指针（在线程栈空间中）

```
lock->set_displaced_header(mark);
 if (mark == (markOop) Atomic::cmpxchg_ptr(lock, obj()->mark_addr(), mark))
{
      TEVENT (slow_enter: release stacklock) ;
      return ;
}
```

lock位于线程栈中

轻量级锁

- 如果轻量级锁失败，表示存在竞争，升级为重量级锁（常规锁）
- 在没有锁竞争的前提下，减少传统锁使用OS互斥量产生的性能损耗
- 在竞争激烈时，轻量级锁会多做很多额外操作，导致性能下降

自旋锁

- 当竞争存在时，如果线程可以很快获得锁，那么可以不在OS层挂起线程，让线程做几个空操作（自旋）
- JDK1.6中-XX:+UseSpinning开启
- JDK1.7中，去掉此参数，改为内置实现
- 如果同步块很长，自旋失败，会降低系统性能
- 如果同步块很短，自旋成功，节省线程挂起切换时间，提升系统性能

偏向锁，轻量级锁，自旋锁总结

- 不是Java语言层面的锁优化方法
- 内置于JVM中的获取锁的优化方法和获取锁的步骤
  - 偏向锁可用会先尝试偏向锁
  - 轻量级锁可用会先尝试轻量级锁
  - 以上都失败，尝试自旋锁
  - 再失败，尝试普通锁，使用OS互斥量在操作系统层挂起

一个错误使用锁的案例

```
public class IntegerLock {
        static Integer i=0;
        public static class AddThread extends Thread{
                public void run(){
                        for(int k=0;k<100000;k++){
                                synchronized(i){
                                        i++;
                                }
                        }
                }
        }

        public static void main(String[] args) throws InterruptedException {
                AddThread t1=new AddThread();
                AddThread t2=new AddThread();
                t1.start();t2.start();
                t1.join();t2.join();
                System.out.println(i);
        }
}
```

ThreadLocal及其源码分析

```
private static final  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
public static class ParseDate implements Runnable{
        int i=0;
        public ParseDate(int i){this.i=i;}
        public void run() {
                try {
                        Date t=sdf.parse("2015-03-29 19:29:"+i%60);
                        System.out.println(i+":"+t);
                } catch (ParseException e) {
                        e.printStackTrace();
                }
        }
}
public static void main(String[] args) {
        ExecutorService es=Executors.newFixedThreadPool(10);
        for(int i=0;i<1000;i++){
                es.execute(new ParseDate(i));
        }
}
```

SimpleDateFormat被多线程访问

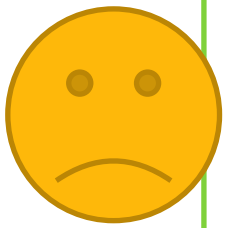
ThreadLocal及其源码分析

```
static ThreadLocal<SimpleDateFormat> tl=new ThreadLocal<SimpleDateFormat>();
public static class ParseDate implements Runnable{
        int i=0;
        public ParseDate(int i){this.i=i;}
        public void run() {
                try {
                        if(tl.get()==null){
                                tl.set(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
                        }
                        Date t=tl.get().parse("2015-03-29 19:29:"+i%60);
                        System.out.println(i+":"+t);
                } catch (ParseException e) {
                        e.printStackTrace();
                }
        }
}
public static void main(String[] args) {
        ExecutorService es=Executors.newFixedThreadPool(10);
        for(int i=0;i<1000;i++){
                es.execute(new ParseDate(i));
        }

}
```

为每一个线程分配一个实例

ThreadLocal及其源码分析

```
static ThreadLocal<SimpleDateFormat> tl=new ThreadLocal<SimpleDateFormat>();
private static final  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
public static class ParseDate implements Runnable{
        int i=0;
        public ParseDate(int i){this.i=i;}
        public void run() {
                try {
                        if(tl.get()==null){
                                tl.set(sdf);
                        }
                        Date t=tl.get().parse("2015-03-29 19:29:"+i%60);
                        System.out.println(i+":"+t);
                } catch (ParseException e) {
                        e.printStackTrace();
                }
        }
}
public static void main(String[] args) {
        ExecutorService es=Executors.newFixedThreadPool(10);
        for(int i=0;i<1000;i++){
                es.execute(new ParseDate(i));
        }

}
```



如果使用共享实例，起不到效果

ThreadLocal及其源码分析

- 源码参见JDK

Thanks

FAQ时间