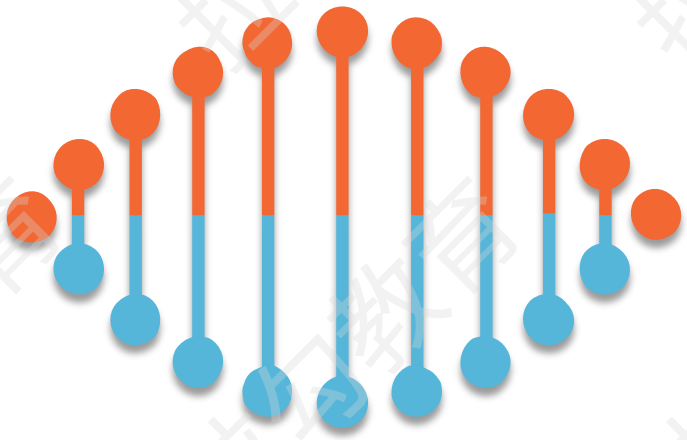
《Java性能优化与面试21讲》

李国

— 拉勾教育出品 —

14 | 案例分析：乐观锁和无锁

Lock 基于 AQS（AbstractQueuedSynchronizer）实现



前言

synchronized

RUNNABLE 状态

BLOCKED 状态

前言

synchronized

RUNNABLE 状态

BLOCKED 状态

前言

synchronized

RUNNABLE 状态

BLOCKED 状态

前言

synchronized

造成用户态和内核态的频繁切换，效率低

RUNNABLE 状态

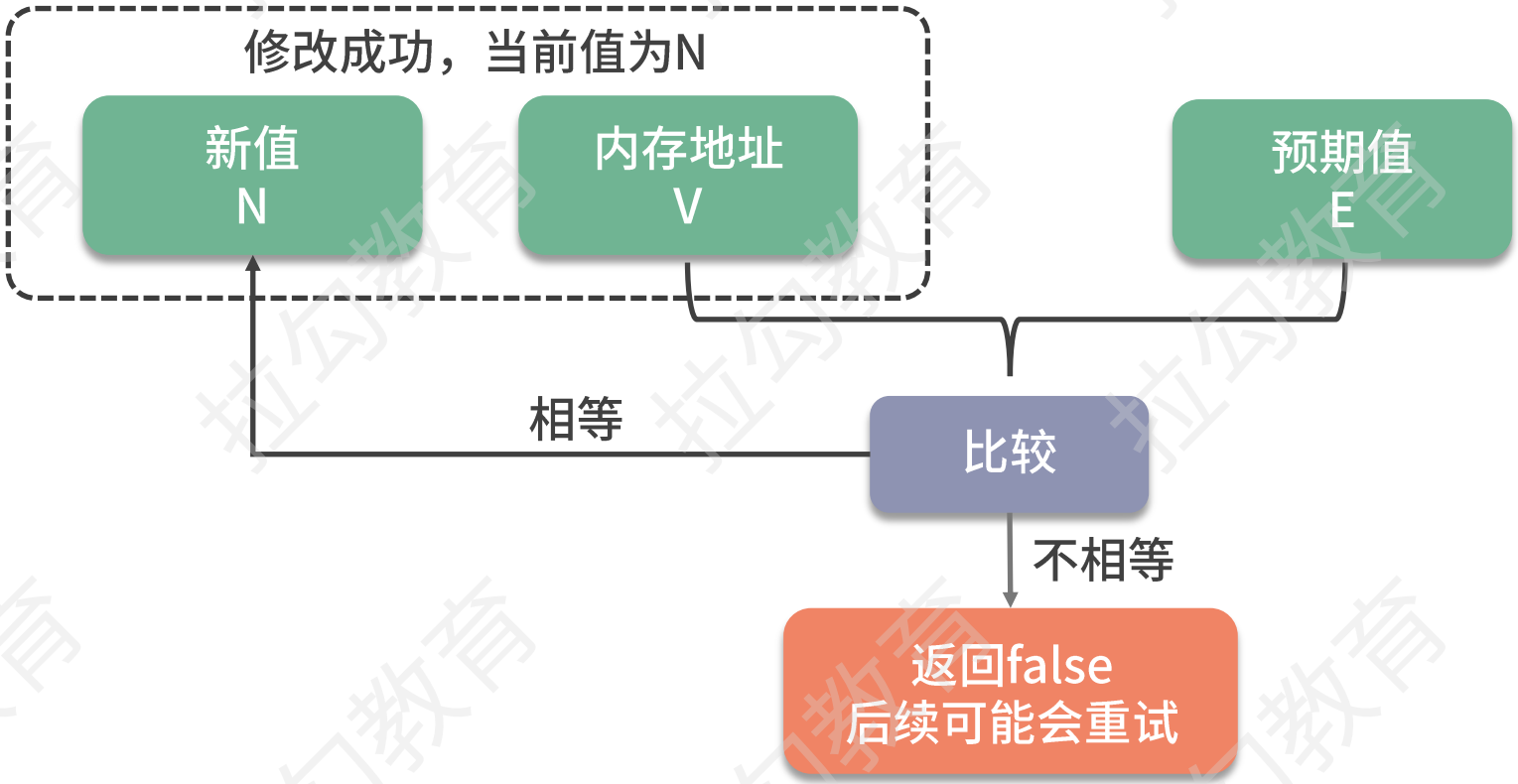
BLOCKED 状态

CAS

CAS（Compare And Swap），意思是**比较并替换**

CAS

CAS （Compare And Swap），意思是比较并替换



CAS 机制图

CAS


```
public final boolean compareAndSet(int expectedValue, int newValue) {
        return U.compareAndSetInt(this, VALUE, expectedValue, newValue);
}
```

CAS

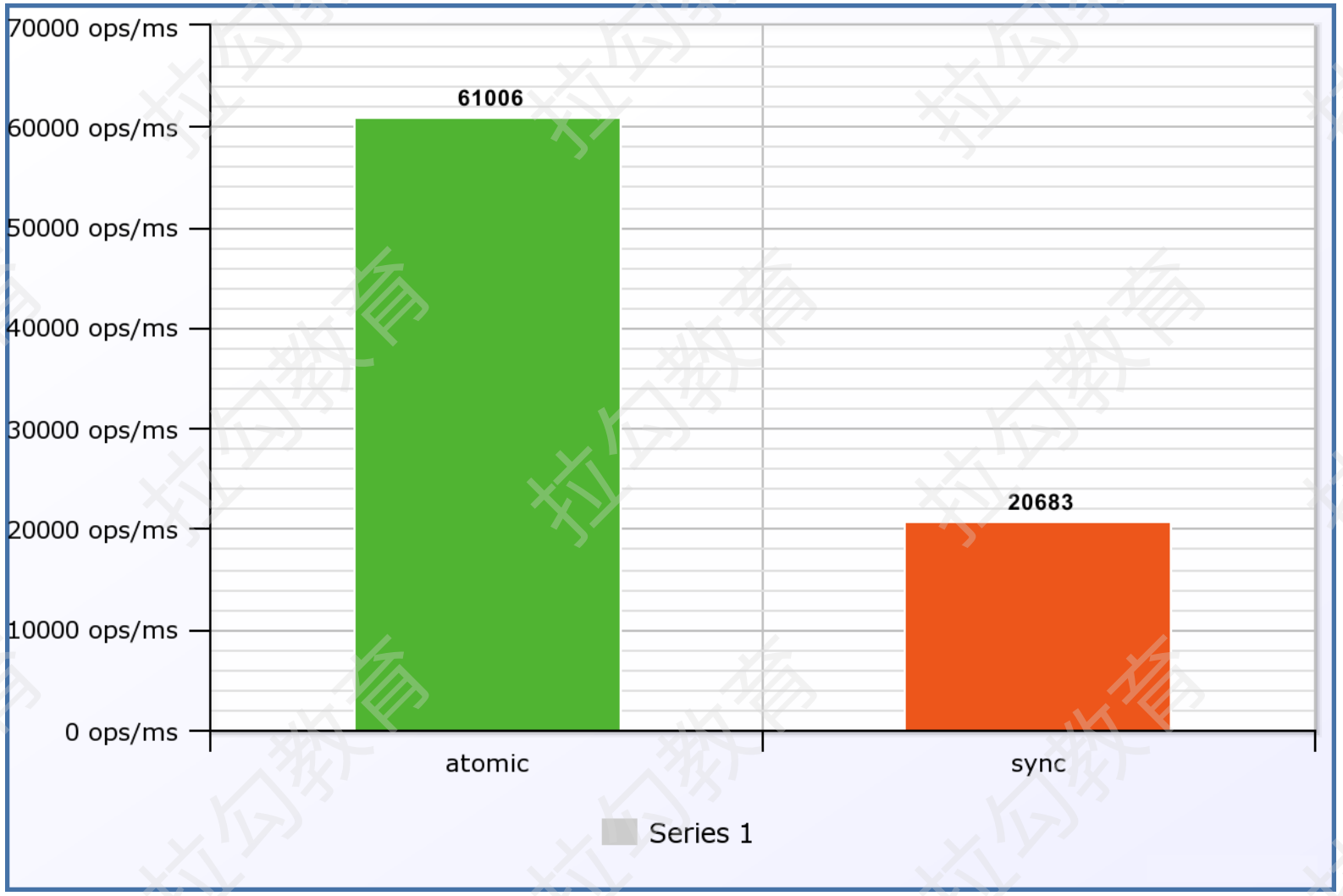

```
@HotSpotIntrinsicCandidate
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));
    return v;

}
```

CAS

```
template<>
template<typename T>
inline T Atomic::PlatformCmpxchg<4>::operator()(T exchange_value,
                                                T volatile* dest,
                                                T compare_value,
                                                atomic_memory_order /* order */) const {
  STATIC_ASSERT(4 == sizeof(T));
  __asm__ volatile ("lock cmpxchgl %1,(%3)"
                    : "=a" (exchange_value)
                    : "r" (exchange_value), "a" (compare_value), "r" (dest)
                    : "cc", "memory");
  return exchange_value;
}
```

CAS



CAS

你知道它是用来干什么的吗？

```
private volatile int value;
```

乐观锁

乐观锁

提供了一种检测冲突的机制

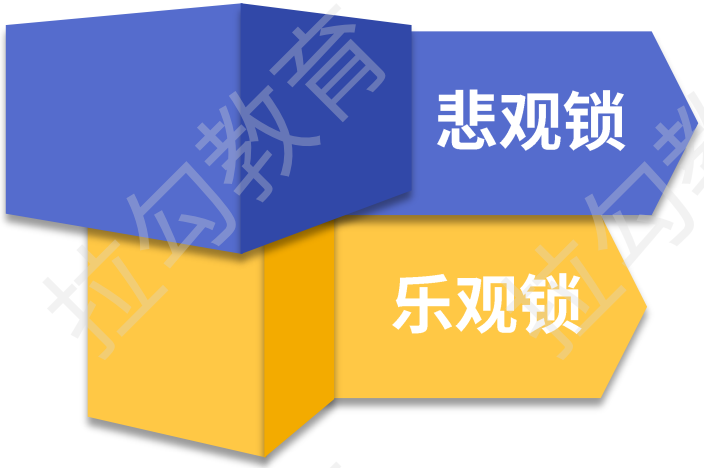
并在有冲突的时候，采取重试的方法完成某项操作

乐观锁

每次操作数据的时候，都会认为别人会修改

所以每次在操作数据的时候，都会加锁，除非释放掉锁

乐观锁



每次操作数据的时候，都会认为别人会修改

所以每次在操作数据的时候，都会加锁，除非释放掉锁

检测到冲突的时候，会有多次重试操作

资源冲突比较严重的场景，乐观锁会出现多次失败的情况，造成 CPU 的空转

乐观锁

悲观锁需要遵循下面三种模式

一锁、二读、三更新

即使在没有冲突的情况下，执行也会非常慢

乐观锁本质上不是锁，它只是一个判断逻辑

资源冲突少的情况下，它不会产生任何开销

乐观锁

- 在并发量比较高的情况下，有些线程可能会一直尝试修改某个资源

  但由于冲突比较严重，一直更新不成功，这时候，就会给 CPU 带来很大的压力

- CAS 操作的对象，只能是单个资源

  如果想要保证多个资源的原子性，最好使用synchronized 等经典加锁方式

- ABA 问题，意思是指在 CAS 操作时，有其他的线程现将变量的值由 A 变成了 B，然后又改成了 A

  当前线程在操作时，发现值仍然是 A，于是进行了交换操作

乐观锁实现余额更新

请求A：读取余额100

请求B：读取余额100

请求A：花掉5元，临时余额是95

请求B：花掉80元，临时余额是20

请求B：写入余额20成功

请求A：写入余额95成功

乐观锁实现余额更新


```
select * from user where userid={id} for update
```

乐观锁实现余额更新


```
select * from user where userid={id} for update
```

使用 select for update 其实在底层就加了三把锁，非常**昂贵**

乐观锁实现余额更新

比较好的办法是使用**乐观锁**

乐观锁实现余额更新


```
# old_balance获取
select balance from  user where userid={id}
# 更新动作
update user set balance = balance - 20
   where userid={id}
   and balance >= 20
   and balance = $old_balance
```

乐观锁实现余额更新

```
version,balance = dao.getBalance(userid)
balance = balance - cost
dao.exec("
    update user
    set balance = balance - 20
    version = version + 1
    where userid=id
    and balance >= 20
    and version = $old_version
")
```

Redis 分布式锁

Redis 分布式锁是互联网行业经常使用的方案

但 Redis 的分布式锁其实有**很多坑**

Redis 分布式锁

拉勾教育
互联网人实战大学

锁创建

SETNX [KEY] [VALUE] 原子操作意思是在指定的 KEY 不存在的时候，创建一个并返回 1，否则返回 0

通常使用参数更全的 set key value [EX seconds] [PX milliseconds] [NX|XX] 命令，同时对 KEY 设置一个超时时间

锁查询

GET KEY

通过简单地判断 KEY 是否存在

锁删除

DEL KEY

删掉相应的 KEY 即可

Redis 分布式锁


```
public void lock(String key, int timeOutSecond) {
    for (; ; ) {
        boolean exist = redisTemplate.opsForValue().setIfAbsent(key, "", timeOutSecond,
TimeUnit.SECONDS);
        if (exist) {
            break;
        }
    }
}
public void unlock(String key) {
    redisTemplate.delete(key);
}
```

Redis 分布式锁

- 请求A：获取了资源 x 的锁，锁的超时时间为 5 秒
- 请求A：由于业务执行时间比较长，业务阻塞等待，超过 5 秒
- 请求B： 第 6 秒发起请求，结果发现锁 x 已经失效，于是顺利获得锁
- 请求A： 第 7 秒，请求 A 执行完毕，然后执行锁释放动作
- 请求C： 请求 C 在锁刚释放的时候发起了请求，结果顺利拿到了锁资源

Redis 分布式锁


```
public String lock(String key, int timeOutSecond) {
    for (; ; ) {
        String stamp = String.valueOf(System.nanoTime());
        boolean exist = redisTemplate.opsForValue().setIfAbsent(key, stamp,
timeOutSecond, TimeUnit.SECONDS);
        if (exist) {
            return stamp;
        }
    }
}
public void unlock(String key, String stamp) {
    redisTemplate.execute(script, Arrays.asList(key), stamp);
}
```

Redis 分布式锁


```
local stamp = ARGV[1]
local key = KEYS[1]
local current = redis.call("GET",key)
if stamp == current then
   redis.call("DEL",key)
   return "OK"
end
```

Redis 分布式锁


```
String resourceKey = "goodgirl";
RLock lock = redisson.getLock(resourceKey);
try {
   lock.lock(5, TimeUnit.SECONDS);
   //真正的业务
   Thread.sleep(100);
} catch (Exception ex) {
   ex.printStackTrace();
} finally {
   if (lock.isLocked()) {
      lock.unlock();
   }
}
```

Redis 分布式锁


```
127.0.0.1:6379> monitor
OK
1596370270.974798 [0 127.0.0.1:57858] "EVAL" "if (redis.call('exists', KEYS[1]) == 0) then redis
edis.call('pexpire', KEYS[1], ARGV[1]); return nil; end; if (redis.call('hexists', KEYS[1], ARGV
YS[1], ARGV[2], 1); redis.call('pexpire', KEYS[1], ARGV[1]); return nil; end; return redis.call(
 "8a01d3f5-fe9b-498f-bf40-2b74fa0d5993:1"
1596370270.974915 [0 lua] "exists" "goodgirl"
1596370270.974926 [0 lua] "hincrby" "goodgirl" "8a01d3f5-fe9b-498f-bf40-2b74fa0d5993:1" "1"
1596370270.974942 [0 lua] "pexpire" "goodgirl" "5000"
1596370271.103941 [0 127.0.0.1:57859] "EXISTS" "goodgirl"
1596370271.115117 [0 127.0.0.1:57860] "EVAL" "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0)
is.call('hincrby', KEYS[1], ARGV[3], -1); if (counter > 0) then redis.call('pexpire', KEYS[1], A
, KEYS[1]); redis.call('publish', KEYS[2], ARGV[1]); return 1; end; return nil;" "2" "goodgirl"
"5000" "8a01d3f5-fe9b-498f-bf40-2b74fa0d5993:1"
1596370271.115212 [0 lua] "hexists" "goodgirl" "8a01d3f5-fe9b-498f-bf40-2b74fa0d5993:1"
1596370271.115231 [0 lua] "hincrby" "goodgirl" "8a01d3f5-fe9b-498f-bf40-2b74fa0d5993:1" "-1"
1596370271.115242 [0 lua] "del" "goodgirl"
1596370271.115248 [0 lua] "publish" "redisson_lock__channel:{goodgirl}" "0"
```

无锁（Lock-Free）

指的是在多线程环境下，在访问共享资源的时候，不会阻塞其他线程的执行

无锁

拉勾教育
互联网人实战大学

ConcurrentLinkedQueue

是一个非阻塞队列，性能很高，但不是很常用

Disruptor

是一个无锁、有界的队列框架，它的性能非常

小结

乐观锁严格来说并不是一种锁，提供了一种检测冲突的机制

并在有冲突的时候，采取重试的方法完成某项操作

假如没有重试操作，乐观锁就仅仅是一个判断逻辑而已

悲观锁每次操作数据的时候，都会认为别人会修改

所以每次在操作数据的时候，都会加锁，除非别人释放掉锁

思考题

一个接口的写操作，大约会花费 5 分钟左右的时间

它在开始写时，会把数据库里的一个字段值更新为 start，写入完成后，更新为 done

有另外一个用户也想写入一些数据，但需要等待状态为 done

于是开发人员在 WEB 端，使用轮询，每隔 5 秒，查询字段值是否为 done

当查询到正确的值，即可开始进行数据写入

开发人员的这个方法，属于乐观锁么？有哪些潜在问题？应该如何避免？

Next：第15讲《案例分析：从 BIO 到 NIO，再到 AIO》

拉勾教育

—互联网人实战大学—

下载「拉勾教育App」
获取更多内容

L / A / G / O / U