L06-线程安全-锁与原子类详解

JAVA并发编程

学习目标

- 掌握JUC-Lock-API
- 掌握原子类原理与使用

JUC-Lock-API详解

1 认识Lock

- 问题：有synchronized ,为什么还提供Lock?

- synchronized 的特点：

  - 非公平锁
  - 不可尝试加锁
  - 不可灵活指定等待的时长
  - 不可中断等待过程
  - 持有多个锁时，必须以相反的顺序释放锁，即锁的释放顺序是固定的。

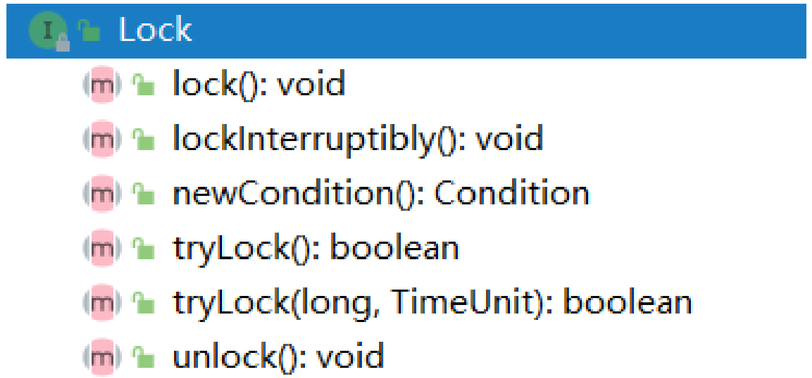
    如获取持有多个锁的顺序： ABCD

    释放顺序：DCBA

    不可以： 获取A-获取B-释放A-获取C-释放B-获取D
  - 只有一个条件来控制多个线程的wait/notify，不能多条件来控制等待不同条件的线程

1 认识Lock

- 通过Lock,JDK为我们提供了上述的遍历

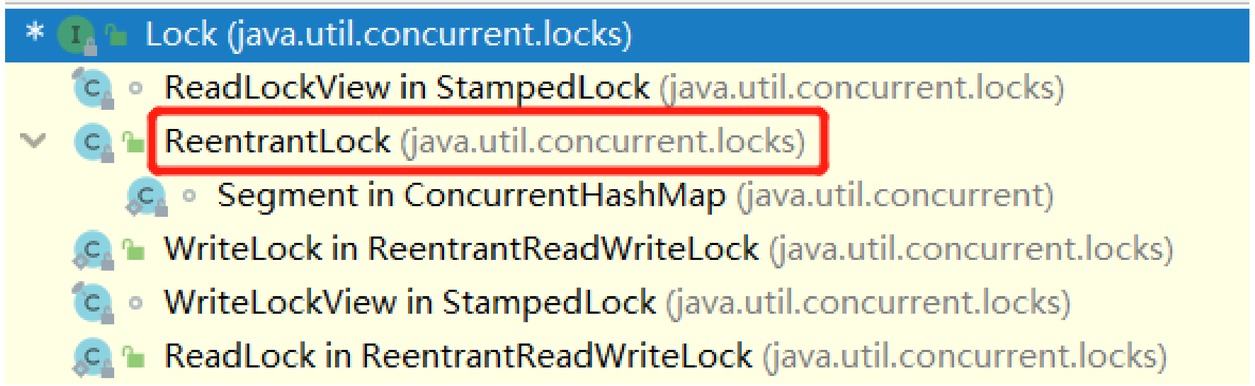


- Lock的使用方式

```
Lock l = ...;
l.lock();
try {
  // access the resource protected by this lock
} finally {
  l.unlock();
})
```

1 认识Lock

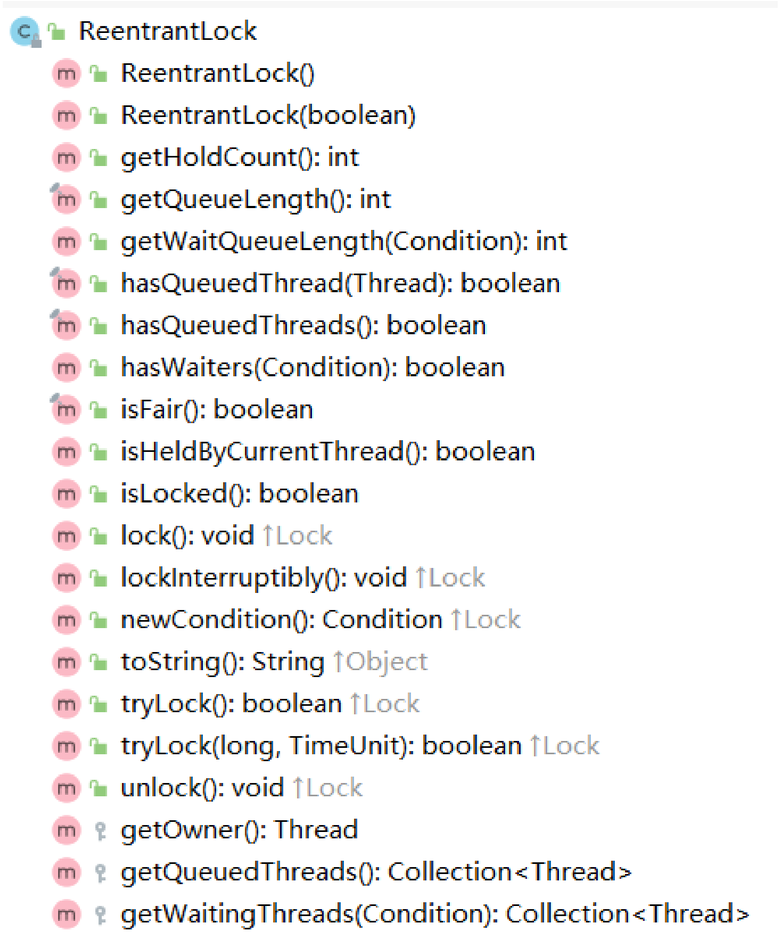
- JUC中Lock的实现



- Lock实现注意事项：
  - 必须实现和monitor lock(synchronized)同样的内存同步语义
    - 成功的lock操作要有与成功的Lock Action相同的内存同步效果
    - 成功的unlock操作要有与成功的unlock Action 相同的内存同步效果
    - 不成功的lock和unlock操作，及reentrant(重入)的lock/unlock操作，不需要任何内存同步效果

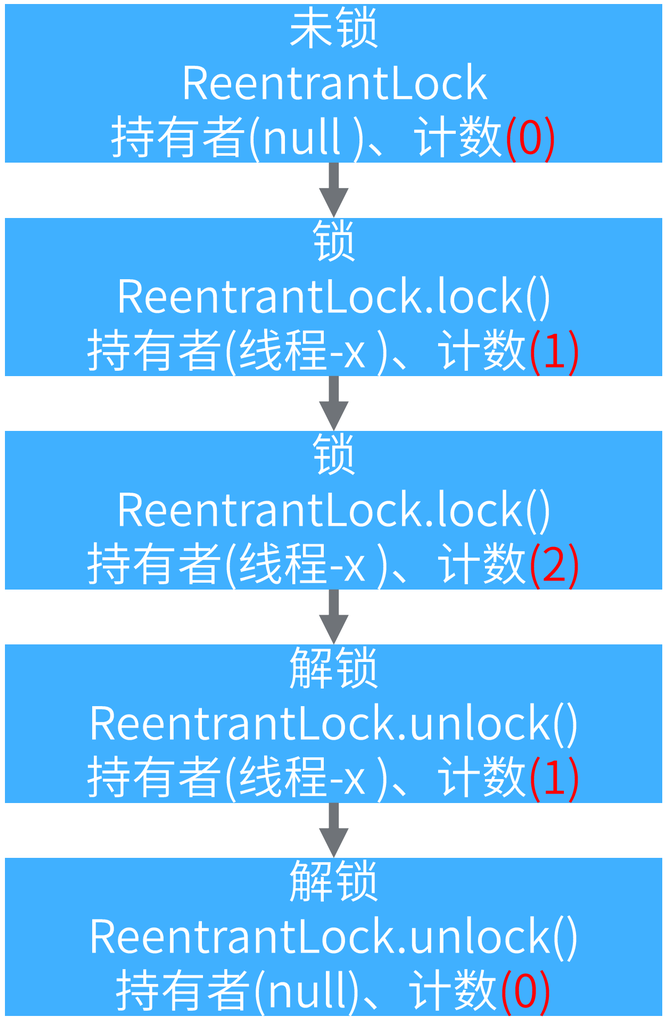
2 掌握ReentrantLock使用

- 可重入锁：独占锁（同synchronized），支持公平锁、非公平锁两种模式




```
class X {
  private final ReentrantLock lock = new
ReentrantLock();
  // ...

  public void m() {
    lock.lock();  // block until condition holds
    try {
      // ... method body
    } finally {
      lock.unlock()
    }
  }
}
```



3 掌握Condition

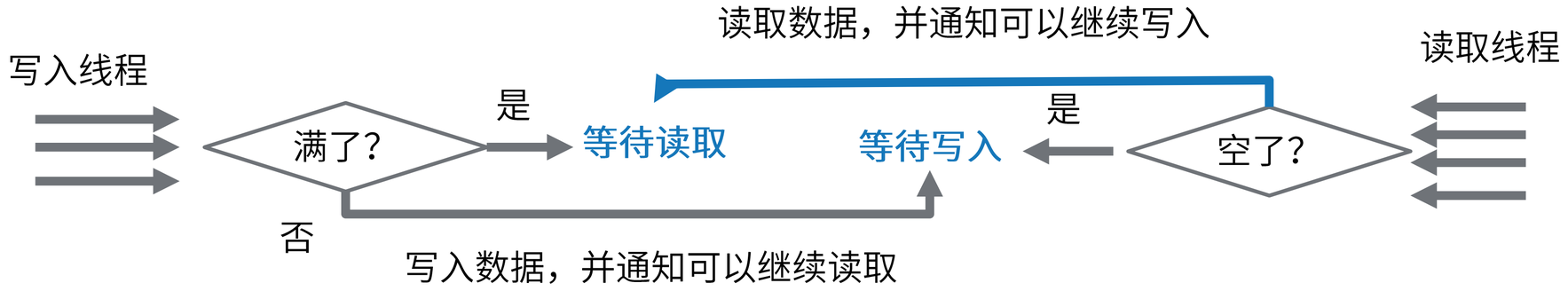
- 俗称：条件等待，条件变量，用于在Lock中提供synchronized的Object.wait/notify 能力

- Object中的wait(),notify(),notifyAll()和synchronized配合使用，可以唤醒一个或者全部(单个等待集)
- Condition与Lock配合使用，一个Lock实例可以创建多个Condition，一个条件一个等待集合，可根据条件精确控制等待线程

典型场景：JDK中的队列实现。

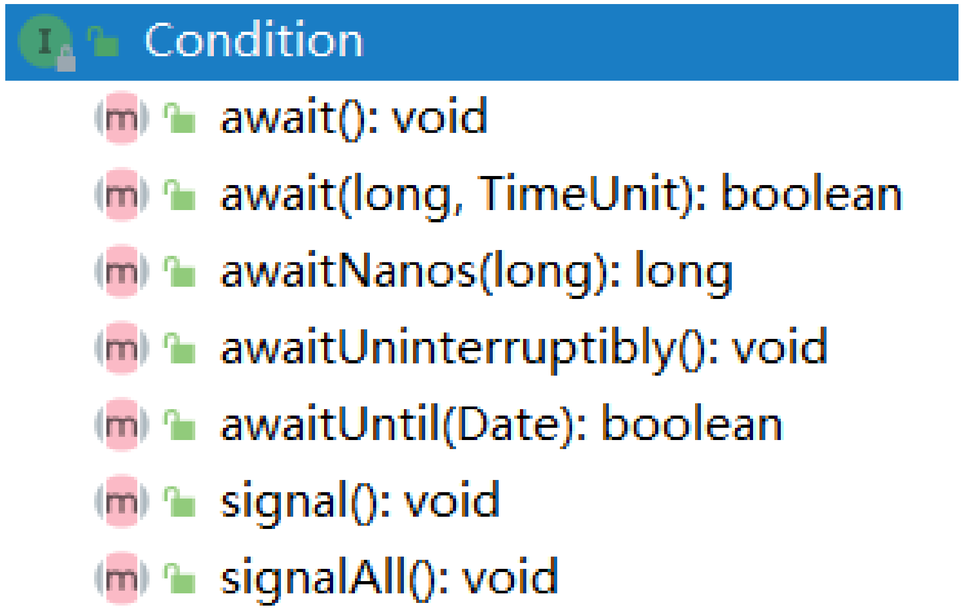
多线程读写队列，写入数据时，唤醒**读取线程**继续执行；读取数据后，通知**写线程**继续执行

Wait/notify/notifyAll无法精确控制唤醒读的线程、写的线程



3 掌握Condition

- Condition-api



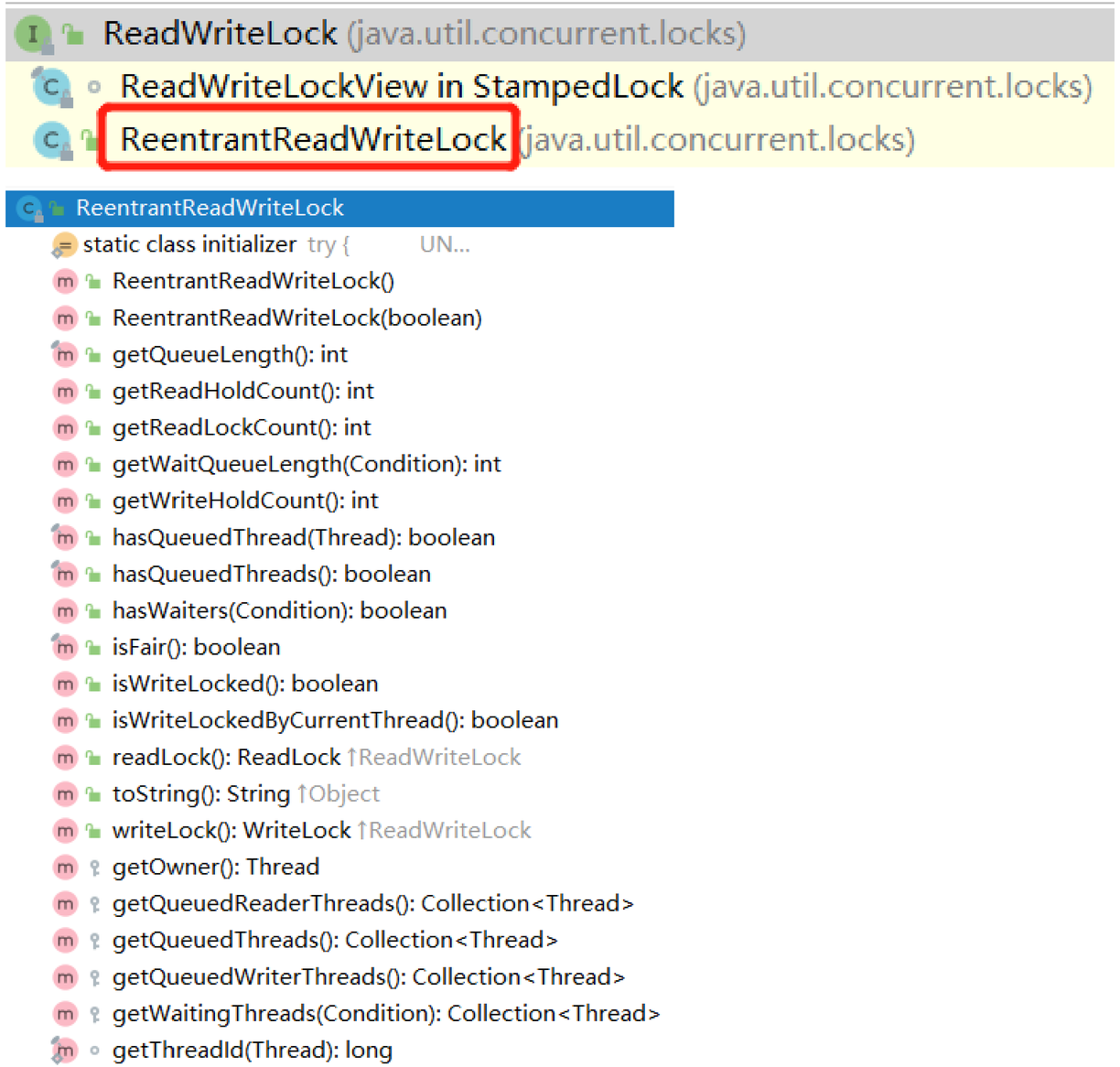
4 掌握ReadWriteLock

- 维护一对关联的读锁、写锁，读锁可以多个读线程共享，写锁排他。读读共享、读写互斥、写写互斥
- **适用场景**：适合读取线程比写入线程多的场景，改进互斥锁的性能，比如：集合的并发线程安全性改造、缓存组件。读写锁适用于并发读很频繁，写很少的场景;但当读大量并发，写线程会迟迟抢不到写锁，数据迟迟不能更新，总体的性能表现反而会很差。所以使用读写锁时，需特别小心
- **锁降级**：指的是写锁降级成为读锁。把持住当前拥有的写锁的同时，再获取到读锁，随后释放写锁的过程。写锁是线程独占，读锁是共享，所以写->读是降级。(读->写，是不能实现的)

4 掌握ReadWriteLock

```
public interface ReadWriteLock {
    /**
     * Returns the lock used for reading.
     *
     * @return the lock used for reading
     */
    @NotNull()
    Lock readLock();

    /**
     * Returns the lock used for writing.
     *
     * @return the lock used for writing
     */
    @NotNull()
    Lock writeLock();
}
```



4 掌握ReadWriteLock

- 使用示例

```
class RWDictionary {
  private final Map<String, Data> m = new TreeMap<String, Data>();
  private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
  private final Lock r = rwl.readLock();
  private final Lock w = rwl.writeLock();

  public Data get(String key) {
    r.lock();
    try { return m.get(key); }
    finally { r.unlock(); }
  }
  public String[] allKeys() {
    r.lock();
    try { return m.keySet().toArray(); }
    finally { r.unlock(); }
  }
  public Data put(String key, Data value) {
    w.lock();
    try { return m.put(key, value); }
    finally { w.unlock(); }
  }
  public void clear() {
    w.lock();
    try { m.clear(); }
    finally { w.unlock(); }
  }
}
```

```
class CachedData {
  Object data;
  volatile boolean cacheValid;
  final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

  void processCachedData() {
    rwl.readLock().lock();
    if (!cacheValid) {
      // Must release read lock before acquiring write lock
      rwl.readLock().unlock();
      rwl.writeLock().lock();
      try {
        // Recheck state because another thread might have
        // acquired write lock and changed state before we did.
        if (!cacheValid) {
          data = ...
          cacheValid = true;
        }
        // Downgrade by acquiring read lock before releasing write lock
        rwl.readLock().lock();
      } finally {
        rwl.writeLock().unlock(); // Unlock write, still hold read
      }
    }

    try {
      use(data);
    } finally {
      rwl.readLock().unlock();
    }
  }
}
```

锁降级使用示例

5 掌握StampedLock

- 邮戳锁：Java8中推出的对读写锁的缺点进行改进的邮戳锁，它推出了乐观读来改进大量并发读，少量写的情况的性能。
- StampedLock是不可重入的，使用时需特别注意。
- StampedLock有三种锁模式，读写可相互转换:
  - Writing　写模式
  - Reading　悲观读模式
  - Optimistic Reading 乐观读模式
- 一个StampedLock状态是由版本（票据）和模式两个部分组成，锁获取方法返回一个数字作为票据stamp，它用相应的锁状态表示并控制访问，数字0表示没有写锁被授权访问。在读锁上分为悲观锁和乐观锁。

5 掌握StampedLock-API

- 获得独占写锁：
  - long writeLock() 阻塞式获取写锁。
  - long tryWriteLock() 尝试获得写锁，如返回0表示没有获得锁。
  - long tryWriteLock(long time, TimeUnit unit) 尝试获得写锁，如返回0表示没有获得锁。
  - long writeLockInterruptibly() 可中断式获取写锁。
- 释放写锁：
  - void unlockWrite(long stamp)
  - boolean tryUnlockWrite() 尝试释放写锁，如果线程持有写锁，释放，返回true，否则返回false

5 掌握StampedLock-API

- 获取读锁：
  - long readLock() 阻塞式非独占获取写锁。
  - long tryReadLock()尝试获得写锁，如返回0表示没有获得锁。
  - long tryReadLock(long time, TimeUnit unit) 尝试获得写锁，如返回0表示没有获得锁。
  - long readLockInterruptibly() 可中断式获取读锁。
- 释放读锁：
  - void unlockRead(long stamp)
  - boolean tryUnlockRead() 尝试释放对读锁的一个持有；如果持有，则释放，返回true,否则返回false

5 掌握StampedLock-API

- 获取乐观读锁：
  - long tryOptimisticRead() 当被其他线程独占时，将返回0。
- 检查从获得乐观读锁开始，到目前锁是否被独占过，以判断数据是否被改变，决定是否重新获取：
  - boolean validate(long stamp) 参数为乐观锁票据。
- 释放锁，如果锁的状态匹配给入的stamp,释放锁的对应模式
  - void unlock(long stamp)
- 判断锁的当前状态（模式）
  - boolean isWriteLocked()
  - boolean isReadLocked()

5 掌握StampedLock-API

- 转换模式：
  - long tryConvertToOptimisticRead(long stamp)
  - long tryConvertToReadLock(long stamp)
  - long tryConvertToWriteLock(long stamp)
- 为优化替换老代码中的读写锁，StampedLock 提供了三个方法将自身转为读写锁：
  - Lock asReadLock()
  - Lock asWriteLock()
  - ReadWriteLock asReadWriteLock()

- 看示例代码了解应用

5 各种锁的性能测试

- 测试场景：读&增一个计数器到Integer.MAX_VALUE/4，在不同读写线程数量小的耗时（毫秒），硬件：4G + intel core i5

	5w5r	10r10w	16r4w	19r1w
sync:	46411	48753	50692	38905
lock:	60485	60114	59921	36843
rwlock:	435214	477336	437045	48491
stamped:	84361	92289	81420	42111
optimistic:	30765	27147	29250	20763

- 由此可知：同步关键字的性能表现稳定可靠；乐观读模式下的性能最佳；可重入读写锁的表现不佳，仅在读非常多、写非常少的情况下表现还可，但不如同步关键字。

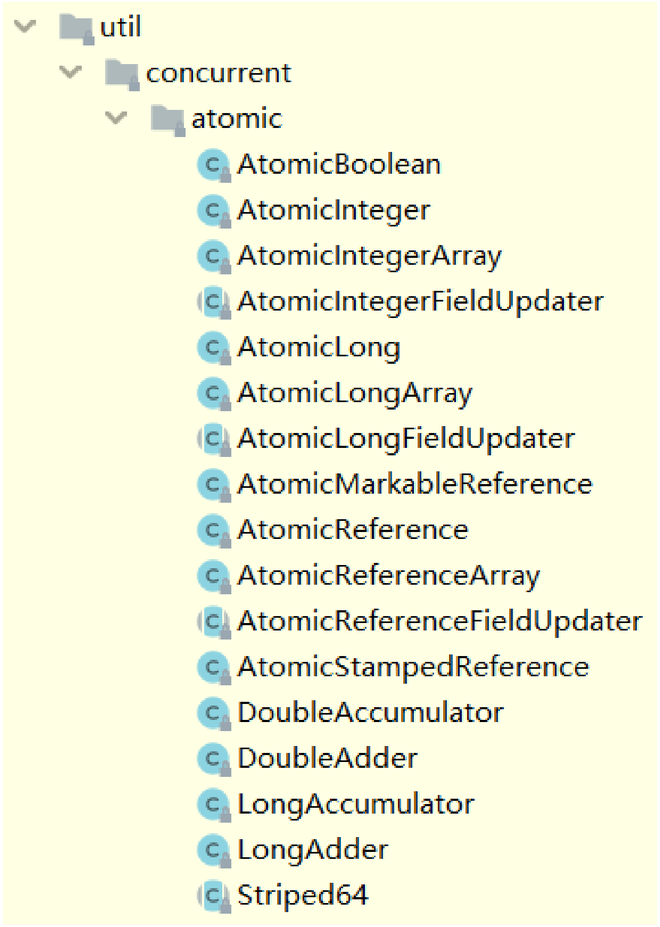
原子类详解

原子类-学习目标

- 了解何为原子类、用途
- 熟悉juc中提供了哪些原子类，能熟练使用
- 了解原子类的实现原理

1了解原子类

- 锁：能很好保证线程安全，但在高并发的情况下，可能满足不了对性能的需要
- 原子类：JUC包中提供的对**单个变量**能进行**无锁、线程安全**修改的工具类



➢AtomicInteger/AtomicLong/AtomicBoolean/AtomicReference
提供对变量的原子更新

➢AtomicIntegerArray/AtomicLongArray/AtomicReferenceArray
提供对数组的原子更新

➢AtomicIntegerFieldUpdater/AtomicLongFieldUpdater/Atomic
ReferenceFieldUpdater 对对象的volatile修饰的int/long/对象引
用属性的原子更新

- 核心API: compareAndSet + 简便的增减操作

2 掌握使用

AtomicInteger/AtomicLong/AtomicBoolean/AtomicReference

- AtomicInteger/AtomicLong 主要用途：对 int、long变量提供原子更新，典型应用场景：计数、计票等
- AtomicReference提供对引用类型变量的原子更新

AtomicIntegerArray/AtomicLongArray/AtomicReferenceArray

- 对对应类型数组的原子更新

2 掌握使用

- **AtomicIntegerFieldUpdater/AtomicLongFieldUpdater/AtomicReference FieldUpdater 基于反射的，对应类型的属性原子更新器。使用规则：**

➢字段必须是volatile类型的，在线程之间共享变量时保证立即可见

➢在执行更新的代码中一定要能直接访问到该字段（字段修饰符可能是public/protected/default/private，在执行原子更新的代码中要能直接放到该字段）。

➢对于父类的字段，子类是不能直接操作的，尽管子类可以访问父类的字段。

➢只能是可修改变量，不能使final变量，因为final的语义就是不可修改。

➢只能是实例变量，不能是类变量，也就是说不能加static关键字。

➢对于AtomicIntegerFieldUpdater和AtomicLongFieldUpdater只能修改int/long类型的字段，不能修改其包装类型（Integer/Long）。如果要修改包装类型就需要使用AtomicReferenceFieldUpdater。

2 掌握使用-ABA问题

- **AtomicStampedReference/AtomicMarkableReference，引入版本/标识来解决ABA问题**

➢ AtomicStampedReference 加入了int邮戳（版本号），记录了变更的次数，解决ABA

➢ AtomicMarkableReference 加入了一个boolean标识，标识值是否变更了，来解决ABA

3 原子类的实现原理

- Unsafe 的 CAS (compareAndSwap)

作业

- 用所学的unsafe、LockSupport等实现自己的ReentrantLock
  - 1、实现lock/unlock
  - 2、实现公平、非公平模式；
  - 3、实现可重入
  - 4、实现内存同步语义

动手实践，下节课见