多线程-高阶

本节目标

> - 围绕多线程的常见面试题给出解答
> - 重点学习 ConcurrentHashMap 的相关知识
> - 了解多线程的其他常见类

1. 常见的锁策略

```
面试题：

1. 你是怎么理解乐观锁和悲观锁的，具体怎么实现呢？

2. 有了解什么读写锁么？

3. 什么是自旋锁，为什么要使用自旋锁策略呢，缺点是什么？

4. synchronized 是可重入锁么？
```

1.1 乐观锁 vs 悲观锁

乐观锁：乐观锁假设认为数据一般情况下不会产生并发冲突，所以在数据进行提交更新的时候，才会正式对数据是否产生并发冲突进行检测，如果发现并发冲突了，则让返回用户错误的信息，让用户决定如何去做。

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。

悲观锁的问题：总是需要竞争锁，进而导致发生线程切换，挂起其他线程；所以性能不高。

乐观锁的问题：并不总是能处理所有问题，所以会引入一定的系统复杂度。

1.2 读写锁

多线程之间，数据的读取方之间不会产生线程安全问题，但数据的写入方互相之间以及和读者之间都需要进行互斥。如果两种场景下都用同一个锁，就会产生极大的性能损耗。所以读写锁因此而产生。

读写锁（readers-writer lock），看英文可以顾名思义，在执行加锁操作时需要额外表明读写意图，复数读者之间并不互斥，而写者则要求与任何人互斥。

1.3 自旋锁（Spin Lock）

按之间的方式处理下，线程在抢锁失败后进入阻塞状态，放弃 CPU，需要过很久才能再次被调度。但经过测算，实际的生活中，大部分情况下，虽然当前抢锁失败，但过不了很久，锁就会被释放。基于这个事实，自旋锁诞生了。

你可以简单的认为自旋锁就是下面的代码

```
while (抢锁(lock) == 失败) {}
```

只要没抢到锁，就死等。

自旋锁的缺点：

缺点其实非常明显，就是如果之前的假设（锁很快会被释放）没有满足，则线程其实是光在消耗 CPU 资源，长期在做无用功的。

1.4 可重入锁

可重入锁的字面意思是“可以重新进入的锁”，即**允许同一个线程多次获取同一把锁。**比如一个递归函数里有加锁操作，递归过程中这个锁会阻塞自己吗？如果不会，那么这个锁就是**可重入锁**（因为这个原因可重入锁也叫做**递归锁**）。

Java里只要以Reentrant开头命名的锁都是可重入锁，而且**JDK提供的所有现成的Lock实现类，包括synchronized关键字锁都是可重入的。**

2. CAS

面试题：

1．讲解下你自己理解的 CAS 机制

2．ABA问题怎么解决？

2.1 什么是 CAS

CAS: 全称Compare and swap，字面意思:”比较并交换“，一个 CAS 涉及到以下操作：

> 我们假设内存中的原数据V，旧的预期值A，需要修改的新值B。 1. 比较 A 与 V 是否相等。（比较） 2. 如果比较相等，将 B 写入 V。（交换） 3. 返回操作是否成功。

当多个线程同时对某个资源进行CAS操作，只能有一个线程操作成功，但是并不会阻塞其他线程,其他线程只会收到操作失败的信号。可见 CAS 其实是一个乐观锁。

2.2 CAS 是怎么实现的

针对不同的操作系统，JVM 用到了不同的 CAS 实现原理，简单来讲：

- java 的 CAS 利用的的是 unsafe 这个类提供的 CAS 操作；
- unsafe 的 CAS 依赖了的是 jvm 针对不同的操作系统实现的 Atomic::cmpxchg；
- Atomic::cmpxchg 的实现使用了汇编的 CAS 操作，并使用 cpu 硬件提供的 lock 机制保证其原子性。

简而言之，是因为硬件予以了支持，软件层面才能做到。

2.3 CAS 有哪些应用

可以用于实现自旋锁，演示代码：

```
public class SpinLock {
  private AtomicReference<Thread> sign =new AtomicReference<>();
```

```
  public void lock(){
      Thread current = Thread.currentThread();

      // 不放弃 CPU，一直在这里旋转判断
      while(!sign .compareAndSet(null, current)){
      }
  }

  public void unlock (){
      Thread current = Thread.currentThread();
      sign.compareAndSet(current, null);
  }
}
```

用于实现原子类，示例代码：

```
public class AtomicInteger {
    public final int incrementAndGet() {
        return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
    }
}

public class Unsafe {
    public final int getAndAddInt(Object var1, long var2, int var4) {
        int var5;
        do {
            var5 = this.getIntVolatile(var1, var2);
        } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

        return var5;
    }
}
```

2.4 ABA 问题如何处理

ABA 的问题，就是一个值从A变成了B又变成了A，而这个期间我们不清楚这个过程。

解决方法：加入版本信息，例如携带 AtomicStampedReference 之类的时间戳作为版本信息，保证不会出现老的值。

3. synchronized 背后的原理（重点）

```
面试题：

1. 什么是偏向锁?

2. java 的 synchronized 是怎么实现的，有了解过么?

3. synchronized 实现原理 是什么?
```

JVM 将 synchronized 锁分为 无锁、偏向锁、轻量级锁、重量级锁 状态。会根据情况，进行依次升级。

无锁：没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功，其他修改失败的线程会不断重试直到修改成功。

偏向锁：对象的代码一直被同一线程执行，不存在多个线程竞争，该线程在后续的执行中自动获取锁，降低获取锁带来的性能开销。偏向锁，指的就是偏向第一个加锁线程，该线程是不会主动释放偏向锁的，只有当其他线程尝试竞争偏向锁才会被释放。

偏向锁的撤销，需要在某个时间点上没有字节码正在执行时，先暂停拥有偏向锁的线程，然后判断锁对象是否处于被锁定状态。如果线程不处于活动状态，则将对象头设置成无锁状态，并撤销偏向锁；

如果线程处于活动状态，升级为轻量级锁的状态。

轻量级锁：轻量级锁是指当锁是偏向锁的时候，被第二个线程 B 所访问，此时偏向锁就会升级为轻量级锁，线程 B 会通过自旋的形式尝试获取锁，线程不会阻塞，从而提高性能。

当前只有一个等待线程，则该线程将通过自旋进行等待。但是当自旋超过一定的次数时，轻量级锁便会升级为重量级锁；当一个线程已持有锁，另一个线程在自旋，而此时又有第三个线程来访时，轻量级锁也会升级为重量级锁。

重量级锁：指当有一个线程获取锁之后，其余所有等待获取该锁的线程都会处于阻塞状态。

重量级锁通过对象内部的监视器（monitor）实现，而其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态切换到内核态，切换成本非常高。

4. Callable 的使用

```
面试题：

1. Callable 使用过么?

2. Furture 类听说过么?
```

封装成带返回结果的多线程任务

```
@FunctionalInterface
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
}
```

```
public class FutureTask<V> implements Runnable, Future<V> {
    ...
}
```

示例代码

```
class MyThread implements Callable<String> {
    private int ticket = 10 ; // 一共10张票
    @Override
    public String call() throws Exception {
        while(this.ticket>0){
            System.out.println("剩余票数: "+this.ticket -- );
        }
        return "票卖完了，下次吧。。。" ;
    }
}
```

```
public class TestDemo {
    public static void main(String[] args) throws Exception {
        FutureTask<String> task = new FutureTask<>(new MyThread()) ;
        new Thread(task).start();
        new Thread(task).start();
        System.out.println(task.get());
    }
}
```

5. java.util.concurrent 包下的常见类

面试题:

1. 线程同步的方式有哪些?

2. 你平时用过哪些线程同步的方式?

3. 为什么有了 synchronized 还需要 juc 下的 lock?

4. AtomicInteger 的实现原理是什么?

5. 信号量听说过么? 之前都用在过哪些场景下?

6. 说一下并发包下有哪些并发类，concurrentHashmap，unsafe,原子类，都分别讲一下怎么支持并发的?

5.1 Semaphore

一个计数信号量，主要用于控制多线程对共同资源库访问的限制。

```
public class SemaphoreTest {
    // 最多 5 个坑
    private static final Semaphore avialable = new Semaphore(5);
```

```
    public static void main(String[] args) {
        ExecutorService pool = Executors.newFixedThreadPool(10);
        Runnable r = new Runnable() {
            public void run(){
                try {
                    avialable.acquire();    //此方法阻塞
                    Thread.sleep(10 * 1000);
                    System.out.println(Thread.currentThread().getName());
                    avialable.release();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        for(int i=0;i<10;i++){
            pool.execute(r);
        }
        pool.shutdown();
    }
}
```

5.2 ReentrantLock

可重入互斥锁。使用上与synchronized关键字对比理解

```
synchronized(object){
    // working
}
```

```
ReentrantLock lock = new ReentrantLock();
-----------------------------------------

lock.lock();
try {
    // working
} finally {
    lock.unlock()
}
```

5.3 CountDownLatch

好像跑步比赛，10个选手依次就位，哨声响才同时出发；所有选手都通过终点，才能公布成绩。

```
public class Demo {
    public static void main(String[] args) throws Exception {
        CountDownLatch latch = new CountDownLatch(10);
        Runnable r = new Runable() {
            @Override
            public void run() {
```

```
                try {
                    Thread.sleep(Math.random() * 10000);
                    latch.countDown();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };
        for (int i = 0; i < 10; i++) {
            new Thread(r).start();
        }
        // 必须等到 10 人全部回来
        latch.await();
        System.out.println("比赛结束");
    }
}
```

5.4 原子类介绍

其实刚才我们以及看到了原子类的出现了，因为原子类内部用的是 CAS 实现，所以性能要比加锁实现 i++ 高很多。原子类有以下几个

AtomicBoolean/AtomicInteger/AtomicIntegerArray/AtomicLong/AtomicReference/AtomicStampedReference

拿 AtomicInteger 举例，常见方法有

```
addAndGet(int delta);   i += delta;
decrementAndGet();      --i;
getAndDecrement();      i--;
incrementAndGet();      ++i;
getAndIncrement();      i++;
```

6. ConcurrentHashMap（重点）

面试题：

1. ConcurrentHashMap的读是否要加锁，为什么?

2. ConcurrentHashMap的锁分段技术?

3. ConcurrentHashMap的迭代器是强一致性的迭代器还是弱一致性的迭代器?

4. Mashmap和ConcurrentHashMap怎么确定key的唯一性?

5. HashTable和HashMap、ConcurrentHashMap?

6. ConcurrentHashMap的原理使用?

7. ConcurrentHashMap在jdk1.8做了哪些优化?

8. ConcurrentHashMap如何实现线程安全?

6.1 Concurrent相关历史

JDK5 中添加了新的concurrent包，在线程安全的基础上提供了更好的写并发能力，但同时降低了对读一致性的要求。与 Vector和 Hashtable、Collections.synchronizedXxx()同步容器等相比，util.concurrent中引入的并发容器主要解决了两个问题：

1）根据具体场景进行设计，尽量避免synchronized，提供并发性。

2）定义了一些并发安全的复合操作，并且保证并发环境下的迭代操作不会出错。

java.util.concurrent中容器在迭代时，可以不封装在synchronized中，可以保证不抛异常，但是未必每次看到的都是"最新的、当前的"数据。

并发编程实践中，ConcurrentHashMap是一个经常被使用的数据结构，它的设计与实现非常精巧，大量的利用了volatile，final，CAS等lock-free技术来减少锁竞争对于性能的影响，无论对于Java并发编程的学习还是Java内存模型的理解，ConcurrentHashMap的设计以及源码都值得非常仔细的阅读与揣摩。

6.2 对比说明

HashTable 是一个线程安全的类，它使用synchronized来锁住整张Hash表来实现线程安全，即每次锁住整张表让线程独占，相当于所有线程进行读写时都去竞争一把锁，导致效率非常低下。

HashMap 则不是一个线程安全的类，完全不能用在多线程的场景下。

ConcurrentHashMap可以做到读取数据不加锁，并且其内部的结构可以让其在进行写操作的时候能够将锁的粒度保持地尽量地小，允许多个修改操作并发进行，其关键在于使用了锁分离技术。它使用了多个锁来控制对hash表的不同部分进行的修改。只要不争夺同一把锁，它们就可以并发进行。

6.3 大体实现原理

1.7 的时候，曾经使用过一种分段锁的机制，即把数组的看作好几部分，分别赋予不同的锁。但 1.8 之后，分段锁已经不再使用了。

1.8 的大概实现原理：

读取数据是不需要加锁，但不保证一定读到最新的数据。

写入时，不涉及扩容，则对链表的第一个结点进行加锁，即只有下标一样的线程才争抢同一把锁。

如果发生了扩容，则情况就非常复杂了；为了效率，它支持多个线程同时进行扩容，而且没有加锁。具体细节咱这里不做详细的讲解了，简单的讲下，就是它在扩容的时候，会保留两个 table，一个用于扩容，一个还在提供服务，做到真正的不停机。

除了这些大的机制，为了尽可能的减少锁的使用，内部实现中还使用了大量的 CAS、volatile、UnSafe 类等技术，代码中技巧性特别强，是要学习多线程登峰造极的极好参考代码。

参考资料：https://blog.csdn.net/u010723709/article/details/48007881

7. 其他常见面试题

```
面试题：

1. volatile关键字?
```

2. Java多线程是如何实现数据共享的?

3. Java创建线程池的接口是什么? 参数`LinkedBlockingQueue`的作用是什么?

4. Java线程共有几种状态? 状态之间怎么切换的?

5. synchronized和lock获取不到锁分别进入什么状态

6. 在多线程下，如果对一个数进行叠加，该怎么做?

7. Servlet是否是线程安全的?

8. Thread和Runnable的区别和联系?

9. 多次start一个线程会怎么样

10. 有synchronized两个方法，两个线程分别同时用这个方法，请问会发生什么?

11. 进程和线程的区别?

12. 死锁，如何避免，避免算法? 实际解决过没有?

13. java线程池的具体过程，及线程池参数 和线程池类型具体阻塞队列?

7.1 死锁

死锁是这样一种情形：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

死锁产生的四个必要条件：

- 1、互斥使用，即当资源被一个线程使用(占有)时，别的线程不能使用
- 2、不可抢占，资源请求者不能强制从资源占有者手中夺取资源，资源只能由资源占有者主动释放。
- 3、请求和保持，即当资源请求者在请求其他的资源的同时保持对原有资源的占有。
- 4、循环等待，即存在一个等待队列：P1占有P2的资源，P2占有P3的资源，P3占有P1的资源。这样就形成了一个等待环路。

当上述四个条件都成立的时候，便形成死锁。当然，死锁的情况下如果打破上述任何一个条件，便可让死锁消失。① 资源一次性分配（破坏请求与保持条件） ② 可剥夺资源：在线程满足条件时，释放掉已占有的资源 ③ 资源有序分配：系统为每类资源赋予一个编号，每个线程按照编号递 请求资源，释放则相反

7.2 银行家算法-操作系统分配资源

一个银行家如何将一定数目的资金安全地借给若干个客户，使这些客户既能借到钱完成要干的事，同时银行家又能收回全部资金而不至于破产。

银行家算法需要确保以下四点：

1、当一个顾客对资金的最大需求量不超过银行家现有的资金时就可接纳该顾客；

2、顾客可以分期贷款, 但贷款的总数不能超过最大需求量；

3、当银行家现有的资金不能满足顾客尚需的贷款数额时，对顾客的贷款可推迟支付，但总能使顾客在有限的时间里得到贷款；

4、当顾客得到所需的全部资金后，一定能在有限的时间里归还所有的资金。

8. 重点内容总结

1. 围绕面试题，理解多线程其他常见的知识体系
2. 重点掌握 synchronized 背后原理 及 ConcurrentHashMap 的知识