面试常见多线程应用案例

1. JAVA中对象锁的模型、wait()、notify()、notifyAll()原理

JAVA 中创建的每个对象都有一个关联的监视器Monitor（也就是互斥锁）。在任何给定时间，只有一个线程可以拥有该监视器.

JVM会为一个使用内部锁（synchronized）的对象维护两个集合：**Entry Set**和**Wait Set**.

Entry Set: 如果一个线程A已经持有了该对象的锁，此时如果有其他线程也想获得该对象的锁的话，它就只能进入到**Entry Set**，并且处于线程的**BLOKED状态**。（当然，这里是针对膨胀到重量级锁的情况）；

Wait Set: 如果线程A调用了wait()方法，那么线程A就会释放该对象的锁，进入到该对象的**Wait Set**中，并且处于线程的**AWAITING状态**。

一个线程B如果想获得对象的锁，一般情况下有两个先决条件：

1. 对象锁已经被释放了（如曾经持有该对象锁的前任线程A执行完了synchronized代码块或者调用了wait()方法）；
2. 线程B已处于RUNNABLE状态。

那么Entry Set和Wait Set两个集合中的线程什么时候会变成RUNNABLE状态呢?

- 对于**Entry Set**中的线程，当对象锁被释放的时候，JVM会唤醒处于Entry Set中的某一线程，这个线程的状态就会**从BLOCKED转变为RUNNABLE**.(这里涉及到**公平锁**和**非公平锁**。**公平锁**就是先来先得，后来的自己到后面排队去；**非公平锁**是“来得早不如来得巧”，当刚好释放了对象锁，就有一个新来的线程想获得对象锁，那么就把该锁分配给该线程，不用再从队列中唤醒一个线程了，节省了时间，正因此，非公平锁的效率高于公平锁)；
- 对于Wait Set中的线程，当对象的notify()方法被调用时，JVM会唤醒处于Wait Set中等待队列的第一个线程，这个线程的状态就会**从WAITING转变为RUNNABLE**；或则当notifyAll()方法被调用时，**Wait Set中的全部线程会转变为RUNNABLE状态**。所有Wait Set中被唤醒的线程会被转移到Entry Set中。

  然后，每个对象的锁释放后，哪些所有处于RUNNABLE状态的线程会共同去竞争获取对象的锁，最终会有一个线程（具体哪一个取决于JVM实现，队列里面的第一个？随机的一个？）真正获取到对象的锁，而其他竞争失败的线程会继续在Entry Set中等待下一次机会。

2. 两个线程交替打印自然数（类似，多个线程打印自然数）

通过Object类的wait()、notify()、notifyAll()等方法实现同步。

刚开始时，一个线程先获得锁并打印，其他线程若获得锁先进入该对象锁的等待队列，当当前线程打印完成后再唤醒等待队列里面的线程。若多个线程要满足线程的顺序，那么就只能调用notify()唤醒等待队列里面的第一个线程。

两个线程依次打印自然数：

两个线程依次打印，那么子能是一个线程在打印另一个线程在等待，当打印完成再唤醒另外的一个线程。在开始的时候，一个线程获得锁可以打印，其他的线程获得了锁只能进入到锁对象的等待队列。

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Demo {
    public static void main(String[] args) {
        Object obj = new Object();
        Thread thread1 = new Thread(new PrintNum(obj, true), "thread1");
        Thread thread2 = new Thread(new PrintNum(obj, false), "thread2");
        //Thread thread3 = new Thread(new PrintNum(obj, false), "thread3");

        thread2.start();
        //thread3.start();
        thread1.start();
    }
}

class PrintNum implements Runnable{
    private static int num;
    private Object obj;
    private boolean isFirstThread;
    public PrintNum(Object obj, boolean isFirstThread) {
        num = 0;
        this.obj = obj;
        this.isFirstThread = isFirstThread;
    }
    @Override
    public void run() {
        synchronized (obj) {
            while (true) {
                // 第一次进来可以直接打印，但是第二次进来后要等到被被唤醒才能打印
                if (isFirstThread) {
                    isFirstThread = false;
                } else {
                    try {
                        obj.wait();
```

```
                        TimeUnit.SECONDS.sleep(1);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
                System.out.println(Thread.currentThread().getName() + ": "
+ num++);

                // 通知其他线程
                obj.notifyAll();
            }
        }
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_211.jdk/Contents/Home/bin/java ...
thread1: 0
thread2: 1
thread1: 2
thread2: 3
thread1: 4
thread2: 5
thread1: 6
thread2: 7
thread1: 8
thread2: 9
thread1: 10
```

多个线程依次打印自然数：

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Demo {
    public static void main(String[] args) {
        Object obj = new Object();
        Thread thread1 = new Thread(new PrintNum(obj, true), "thread1");
        Thread thread2 = new Thread(new PrintNum(obj, false), "thread2");
        Thread thread3 = new Thread(new PrintNum(obj, false), "thread3");
        Thread thread4 = new Thread(new PrintNum(obj, false), "thread4");
        Thread thread5 = new Thread(new PrintNum(obj, false), "thread5");

        thread2.start();
        thread3.start();
        thread4.start();
        thread5.start();
```

```
        thread1.start();
    }
}

class PrintNum implements Runnable{
    private static int num;
    private Object obj;
    private boolean isFirstThread;
    public PrintNum(Object obj, boolean isFirstThread) {
        num = 0;
        this.obj = obj;
        this.isFirstThread = isFirstThread;
    }
    @Override
    public void run() {
        synchronized (obj) {
            while (true) {
                // 第一次进来可以直接打印，但是第二次进来后要等到被被唤醒才能打印
                if (isFirstThread) {
                    isFirstThread = false;
                } else {
                    try {
                        obj.wait();
                        TimeUnit.SECONDS.sleep(1);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
                System.out.println(Thread.currentThread().getName() + ": "
+ num++);

                // 通知其他线程
                obj.notify();
            }
        }
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_211.jdk/Contents/Home/bin/java ...
thread1: 0
thread2: 1
thread3: 2
thread4: 3
thread5: 4
thread1: 5
thread2: 6
thread3: 7
thread4: 8
thread5: 9
thread1: 10
thread2: 11
thread3: 12
thread4: 13
```

3. 生产者/消费者模型

通过wait()、notify()、notifyAll()来实现生产者、消费者模型

Buffer的size为10，3个消费者线程，3个生产者线程

```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class Something {
    private Buffer mBuf = new Buffer();

    public void produce() {
        synchronized (this) {
            while (mBuf.isFull()) {
                try {
                    this.wait();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            mBuf.add();
            this.notifyAll();
        }
    }

    public void consume() {
        synchronized (this) {
            while (mBuf.isEmpty()) {
```

```
                try {
                    this.wait();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            mBuf.remove();
            this.notifyAll();
        }
    }

    static class Buffer{
        private static final int MAX_CAPACITY = 10;
        private List<Object> innerList = new ArrayList<>();

        void add() {
            if (isFull()) {
                throw new IndexOutOfBoundsException();
            } else {
                innerList.add(new Object());
            }
            try {
                TimeUnit.MILLISECONDS.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "
produced a product.");
        }

        void remove() {
            if (isEmpty()) {
                throw new IndexOutOfBoundsException();
            } else {
                innerList.remove(innerList.size() - 1);
            }
            try {
                //TimeUnit.MILLISECONDS.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "
consume a product.");
        }

        boolean isEmpty() {
            return innerList.isEmpty();
        }

```

```
72          boolean isFull() {
73              return (innerList.size() == MAX_CAPACITY);
74          }
75      }
76
77      public static void main(String[] args) {
78          Something something = new Something();
79
80          Thread thread1 = new Thread(new Runnable() {
81              @Override
82              public void run() {
83                  while (true) {
84                      something.produce();
85                  }
86              }
87          }, "producer1");
88
89          Thread thread2 = new Thread(new Runnable() {
90              @Override
91              public void run() {
92                  while (true) {
93                      something.produce();
94                  }
95              }
96          }, "producer2");
97
98          Thread thread3 = new Thread(new Runnable() {
99              @Override
100             public void run() {
101                 while (true) {
102                     something.produce();
103                 }
104             }
105         }, "producer3");
106
107         Thread thread5 = new Thread(new Runnable() {
108             @Override
109             public void run() {
110                 while (true) {
111                     something.consume();
112                 }
113             }
114         }, "consumer1");
115
116         Thread thread6 = new Thread(new Runnable() {
117             @Override
118             public void run() {
119                 while (true) {
120                     something.consume();
```

```
                }
            }
        }, "consumer2");

        Thread thread7 = new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    something.consume();
                }
            }
        }, "consumer3");

        thread1.start();
        thread2.start();
        thread3.start();

        thread5.start();
        thread6.start();
        thread7.start();
    }
}
```

```
/opt/jdk/jdk-11.0.7.jdk/Contents/Home/bin/java ...
producer1 produced a product.
producer1 produced a product.
producer1 produced a product.
producer1 produced a product.
producer1 produced a product.
producer1 produced a product.
producer1 produced a product.
producer1 produced a product.
producer1 produced a product.
producer1 produced a product.
consumer3 consume a product.
consumer3 consume a product.
consumer3 consume a product.
consumer3 consume a product.
consumer3 consume a product.
consumer3 consume a product.
consumer3 consume a product.
consumer3 consume a product.
consumer3 consume a product.
consumer3 consume a product.
consumer3 consume a product
```

注意：为什么在判断buffer是否满时是用的while不是if

因为，wait()的线程永远不能确定其他线程会在什么状态下notify()，所以在被唤醒、抢占到锁并且从wait()方法推出的时候再次进行指定条件的判断，以决定是满足条件往下执行还是不满足条件再次wait().

notify()非常容易导致死锁，尽量使用notifyAll()，它每次都将Wait Setz中的线程唤醒进入到Entry Set中，不会出现死锁，但是时间开销就变大了。如果能确定不会出现死锁的情况，也可以使用notify().