第五周作业：并发编程问答题

题目01 - 请你说一说什么是线程和进程?

题目02 - 使用了多线程会带来什么问题呢?

题目03 - 什么是死锁？如何排查死锁?

题目04 - 请你说一说 synchronized 和 volatile 的原理与区别

题目05 - 为什么使用线程池？如何创建线程池?

题目06 - ThreadLocal 中 Map 的 key 为什么要使用弱引用?

题目01 – 请你说一说什么是线程和进程?

进程：进程就是一个应用程序，它是系统分配资源的最小单元

线程：线程是进程中的一个执行单元，它是系统调度的最小单元

两者的区别关系：

- 进程是系统资源分配的最小单元，线程是系统调度的最小单元
- 进程和进程之间是隔离的，都有其独立的空间；线程和线程之间，堆、方法区是共享的，java虚拟机栈、本地方法栈，程序计数器是私有的

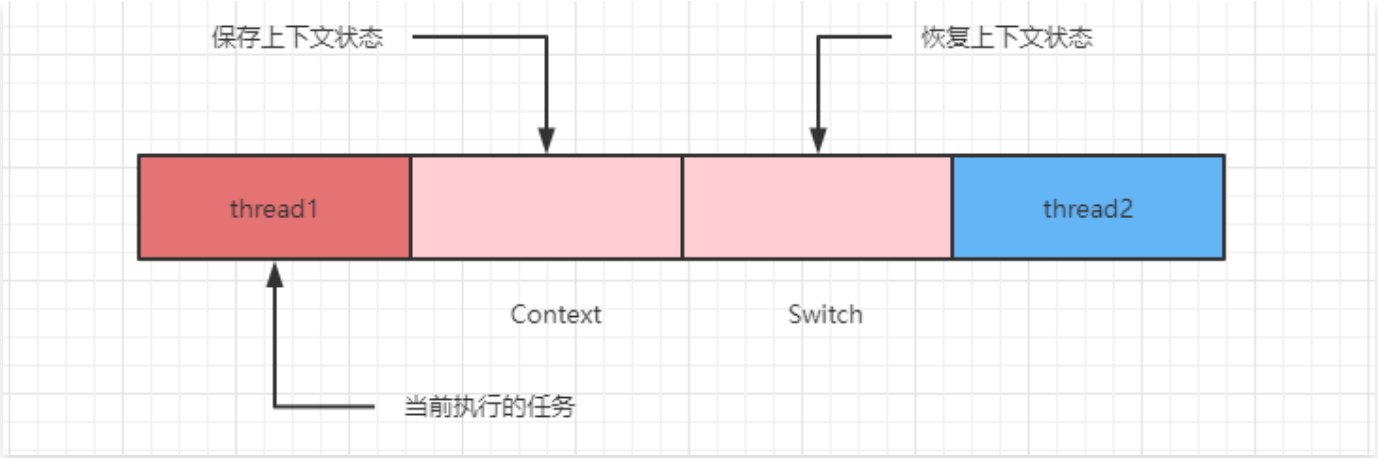
上下文切换：任务状态的保存以及在加载的过程。

上下文切换的过程如下：

（1）挂起当前任务，将任务在CPU中的状态保存

（2）恢复一个任务，从内存中检索下一个任务的上下文并恢复

（3）跳转到程序计数器的指定位置



并行：指在某个时间点，有多个任务同时执行。并行上线取决于CPU核数

并发：指在某个时间段内，有多个任务在执行。

题目02 – 使用了多线程会带来什么问题呢?

线程安全问题：在多线程环境中，当多线程同时访问一个代码的时候，如果每次获取的结果都是和单线程获取的结果一致，那么就没有线程安全的问题，反之则存在线程安全的问题。线程安全性问题都是由于多个线程访问了全局变量或者静态共享变量导致的。

解决线程安全问题的方式：

- 加锁使其线程同步
- JUC工具类
- volatile 关键字

并发有三个特点：

- 原子性：执行一组操作，要么全部执行，要么全部不执行
- 有序性：由于编译器和处理器为了提高运行效率，会对代码进行优化，优化过后的代码不一定与源代码顺序一致。因此有序性就是程序按照代码顺序进行执行
- 可见性：当多个线程访问同一个变量的时候，一个线程修改了这个变量的值，其他线程可以马上看到修改过后的值

题目03 – 什么是死锁？如何排查死锁?

在多线程编程中，为了防止多线程竞争共享资源而导致数据错乱，都会在操作共享资源之前加上互斥锁，只有成功获得到锁的线程，才能操作共享资源，获取不到锁的线程就只能等待，直到锁被释放。那么，当两个线程为了保护两个不同的共享资源而使用了两个互斥锁，那么这两个互斥锁应用不当的时候，可能会造成两个线程都在等待对方释放锁，在没有外力的作用下，这些线程会一直相互等待，就没办法继续运行，这种情况就是发生了死锁。

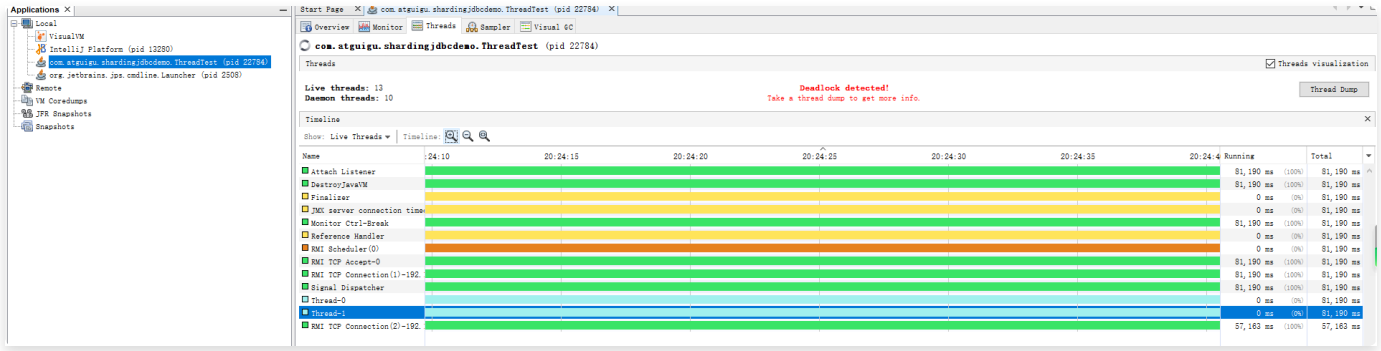
死锁产生的必要条件：

- 互斥条件
- 持有并等待条件
- 不可剥夺条件
- 循环等待条件

可以利用工具排查死锁问题，可以使用 jstack 工具或者一些可视化工具，jstack 是 jdk 自带的线程堆栈分析工具。

这里使用visual VM为例：

可视化工具中可以查看到出现死锁的提示！



```
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x0000000026abd2e8 (object 0x0000000716e2cc98, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x0000000026abd448 (object 0x0000000716e2cca8, a java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at com.atguigu.shardingjdbcdemo.ThreadTest$2.run(ThreadTest.java:29)
        - waiting to lock <0x0000000716e2cc98> (a java.lang.Object)
        - locked <0x0000000716e2cca8> (a java.lang.Object)
        at java.lang.Thread.run(Thread.java:748)
"Thread-0":
        at com.atguigu.shardingjdbcdemo.ThreadTest$1.run(ThreadTest.java:16)
        - waiting to lock <0x0000000716e2cca8> (a java.lang.Object)
        - locked <0x0000000716e2cc98> (a java.lang.Object)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

使用jstack 为例：

```
D:\IntelliJ_IDEA\project\                    >jps -l
13280
22784 com.atguigu.shardingjdbcdemo.ThreadTest
22832 sun.tools.jps.Jps
22952 org/netbeans/Main
2508 org.jetbrains.jps.cmdline.Launcher
```

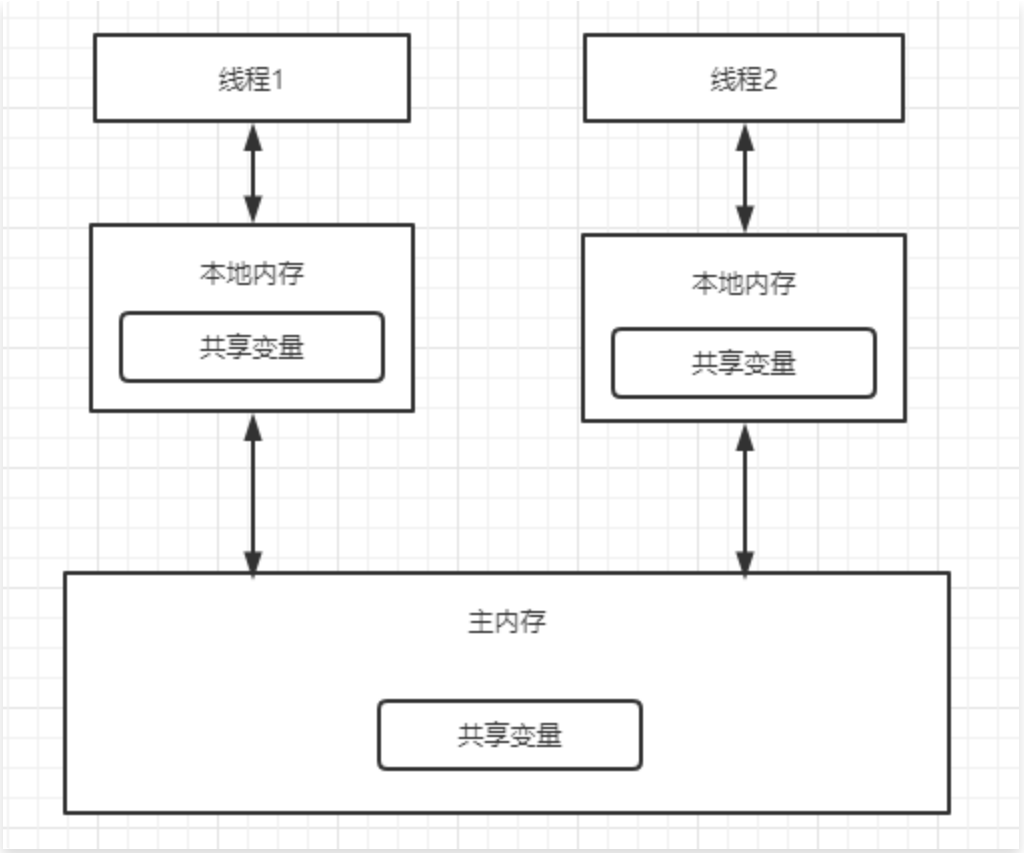
```
D:\IntelliJ_IDEA\project\                    >jstack 22784
2022-09-25 20:28:16
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.161-b12 mixed mode):

"RMI TCP Connection(2)-192.168.56.1" #20 daemon prio=5 os_prio=0 tid=0x000000002a8df800 nid=0x200 runnable [0x000000002bfee000]
   java.lang.Thread.State: RUNNABLE
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
        at java.net.SocketInputStream.read(SocketInputStream.java:171)
        at java.net.SocketInputStream.read(SocketInputStream.java:141)
        at java.io.BufferedInputStream.fill(BufferedInputStream.java:246)
        at java.io.BufferedInputStream.read(BufferedInputStream.java:265)
        - locked <0x0000000718ef71b8> (a java.io.BufferedInputStream)
        at java.io.FilterInputStream.read(FilterInputStream.java:83)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:550)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(TCPTransport.java:826)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.lambda$run$0(TCPTransport.java:683)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler$$Lambda$6/251624711.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:682)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:748)

"JMX server connection timeout 19" #19 daemon prio=5 os_prio=0 tid=0x000000002a376000 nid=0x4b70 in Object.wait() [0x000000002bcef000]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        at com.sun.jmx.remote.internal.ServerCommunicatorAdmin$Timeout.run(ServerCommunicatorAdmin.java:168)
        - locked <0x00000007182bf4b0> (a [I)
        at java.lang.Thread.run(Thread.java:748)
```

```
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x0000000026abd2e8 (object 0x0000000716e2cc98, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x0000000026abd448 (object 0x0000000716e2cca8, a java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at com.atguigu.shardingjdbcdemo.ThreadTest$2.run(ThreadTest.java:29)
        - waiting to lock <0x0000000716e2cc98> (a java.lang.Object)
        - locked <0x0000000716e2cca8> (a java.lang.Object)
        at java.lang.Thread.run(Thread.java:748)
"Thread-0":
        at com.atguigu.shardingjdbcdemo.ThreadTest$1.run(ThreadTest.java:16)
        - waiting to lock <0x0000000716e2cca8> (a java.lang.Object)
        - locked <0x0000000716e2cc98> (a java.lang.Object)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

题目04 – 请你说一说 synchronized 和 volatile 的原理与区别

JMM内存模型

JMM ，Java Memory Model（JMM），它不是真实存在的，它只是一种规范，这个规范规定了一个线程对共享变量写入时，对另一个线程是可见的。



JMM线程操作内存的基本规则：

（1）线程对共享变量的所有操作都必须在自己的本地内存中进行，不能直接从主内存中读写

（2）不同线程之间无法直接访问其他线程本地内存中的变量，线程间变量值的传递都需要经过主内存

什么是happens-before规则?

happens-before规则是JMM中的一种，用于保障内存可见性的方案。

- 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。
- 监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。
- volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。
- 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

synchronized 原理

synchronized 同步操作主要是使用 monitorenter 和 monitorexit 这两个jvm指令实现的：

字节码　异常表　杂项

```
 0 getstatic #3 <java/lang/System.out : Ljava/io/PrintStream;>
 3 ldc #4 <子线程执行。。。>
 5 invokevirtual #5 <java/io/PrintStream.println : (Ljava/lang/String;)V>
 8 aload_0
 9 getfield #2 <com/hero/multithreading/Demo04JmmSynchronized$JmmDemo.flag : Z>
12 ifeq 32 (+20)
15 aload_0
16 dup
17 astore_1
18 monitorenter
19 aload_1
20 monitorexit
21 goto 29 (+8)
24 astore_2
25 aload_1
26 monitorexit
27 aload_2
28 athrow
29 goto 8 (-21)
32 getstatic #3 <java/lang/System.out : Ljava/io/PrintStream;>
35 ldc #6 <子线程结束。。。>
37 invokevirtual #5 <java/io/PrintStream.println : (Ljava/lang/String;)V>
40 return
```

后面多的 monitorexit 命令是因为java中会有隐式的一套 try – catch –finally，其中的 finally 中会再次进行释放锁的操作

volatile 原理

volatile能够确保在多线程环境中共享变量的可见性和有序性。

- 可见性：保证此变量的修改对所有线程可见
- 有序性：禁止指令重排序优化，编译器和处理器在进行指令优化时，不能把在volatile变量操作后面的语句放到其前面执行，也不能将volatile变量操作前面的语句放在其后执行。遵循了JMM 的happens–before规则

volatile可见性实现原理：内存屏障

内存屏障（Memory Barrier）是一种CPU指令，用于控制特定条件下的重排序和内存可见性问题。Java 编译器也会根据内存屏障的规则禁止重排序 写操作时，通过在写操作指令后加入一条store屏障指令，让本地内存中变量的值能够刷新到主内 存中 读操作时，通过在读操作前加入一条load屏障指令，及时读取到变量在主内存的值 。

JMM 内存屏障插入策略：

- 在每个 volatile 写前，插入StoreStore 屏障
- 在每个 volatile 写后，插入StoreLoad 屏障
- 在每个 volatile 读后，插入LoadLoad 屏障
- 在每个 volatile 读后，插入LoadStore 屏障
-

synchronized 和 volatile 的区别

- volatile不需要加锁，比 synchronized 更轻便，不会阻塞线程
- synchronized既能保证可见性，又能保证原子性，而volatile只能保证可见性，无法保证原子性（防止重排序）
- volatile与synchronized相比是一种非常简单的同步机制，在某些情况下，性能比synchronized高

题目05 – 为什么使用线程池？如何创建线程池?

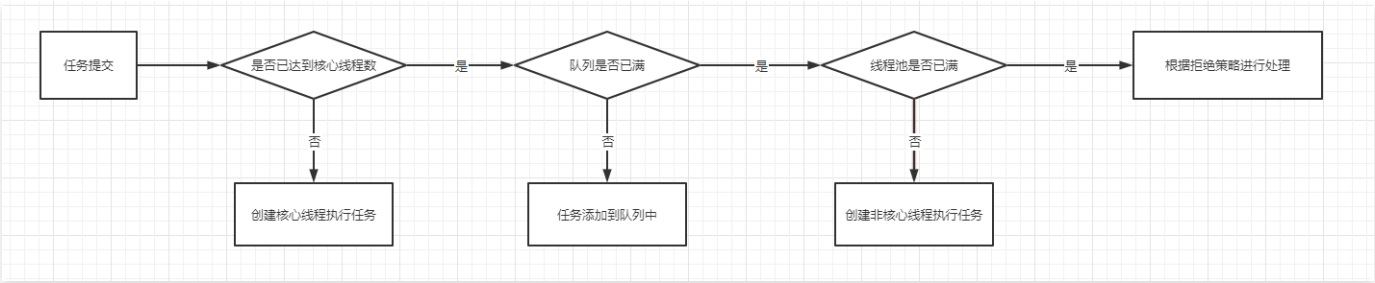
使用线程池的原因：每个任务在创建线程时，其消耗很大，因此使用线程池可以复用线程并且减少创建线程时产生的消耗

线程池有6个重要参数：

- corePoolSize 核心线程数
- maxPoolSize 最大线程数
- keepAliveTime 存活时间
- workQueue 工作队列
- threadFactory 创建线程的线程工厂
- Handler 拒绝策略

线程池原理：

- 如果线程数小于corePoolSize，即使其它工作线程处于空闲状态，也会创建一个新线程来运行新任 务。
- 如果线程数大于等于corePoolSize，但少于maximumPoolSize，将任务放入队列。
- 如果队列已满，并且线程数小于maxPoolSize，则创建一个新线程来运行任务。
- 如果队列已满，并且线程数大于或等于maxPoolSize，则拒绝该任务。



自动创建线程：

- newFixedThreadPool
  - 固定线程的线程池，核心线程数与最大线程数一致，使用无界队列
- newSingleThreadExecutor
  - 创建单线程，核心线程数与最大线程数都是1
- newCachedThreadPool
  - 可缓存线程池，核心线程数为0，最大线程数是整数的最大值可认为没有上限，使用直接交换队列
- newScheduledThreadPool
  - 创建可以定时执行的线程池

手动创建线程：

使用 ThreadPoolExecutor 创建自己的线程池，相关的参数在上面已经说明。

- CPU 密集型任务的并行执行的数量应当等于 CPU 的核心数
- I/O 密集型任务通常需要开 CPU 核心数两倍的线程

拒绝策略：

- AbortPolicy
- DiscardPolicy
- DiscardOldestPolicy
- CallerRunsPolicy

题目06 – ThreadLocal 中 Map 的 key 为什么要使用弱引用?

ThreadLocal是线程本地变量，可以在多线程情况下，保证ThreadLocal 中的变量在每个线程中都有各自的变量值。

ThreadLocal的使用场景：

- 线程隔离
- 跨函数传递数据

ThreadLocal底层原理：

首先每个线程Thread有一个ThreadLocal，每个ThreadLocal里面是一个Map，叫ThreadLocalMap，ThreadLocalMap的key为ThreadLocal实例，value为带保存的值。它和之前版本的ThreadLocal版本的优势在于key-value对的数量相对较少、Thread实例被销毁之后，其ThreadLocalMap也会随之销毁，在一定程度上减少了内存的消耗。

ThreadLocal中的ThreadLocalMap中是一个一个的Entry，也就是说Entry里面是key-value对，Entry的key使用的弱引用。这里**使用弱引用的原因是**：

①如果是强引用的话，当一个方法在结束时，对应的ThreadLocal中的ThreadLocalMap的Entry的key由于是ThreadLocal实例导致其依然指向ThreadLocal实例，强引用导致JVM垃圾回收时发现该引用不可回收，产生内存泄漏。

②如果是弱引用的话，在上面的情况下，垃圾回收时会将其进行回收。回收之后，Entry的key为null，这样在后续的操作中，ThreadLocalMap的内部代码会清理这些Entry中key为null 的值，这样就使得内存得以释放。