多线程&并发篇

1 、说说Java中实现多线程有几种方法

1. 继承 `Thread` 类
2. 实现 `Runnable` 接口
3. 实现 `Callable` 接口（JDK1.5>=）
4. 线程池方式创建。

PS: 阿里巴巴开发手册中规定 **线程资源必须通过线程池提供，不允许在应用中自行显式创建线程**，

> 通过继承Thread类或者实现Runnable接口、Callable接口都可以实现多线程，
> 实现Runnable接口与实现Callable接口的方式基本相同，只是Callable接口里定义的方法返回值，可以声明抛出异常而已。
> 因此将实现Runnable接口和实现Callable接口归为一种方式。这种方式与继承Thread方式之间的主要差别如下。
> 采用实现Runnable、Callable接口的方式创建线程的优缺点
> - 优点：线程类只是实现了Runnable或者Callable接口，还可以继承其他类。这种方式下，多个线程可以共享一个target对象，
> 所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好的体现了面向对象的思想。
> - 缺点：编程稍微复杂一些，如果需要访问当前线程，则必须使用Thread.currentThread()方法
> 采用继承Thread类的方式创建线程的优缺点
> - 优点：编写简单，如果需要访问当前线程，则无需使用Thread.currentThread()方法，直接使用this即可获取当前线程
> - 缺点：因为线程类已经继承了Thread类，Java语言是单继承的，所以就不能再继承其他父类了。

2 、如何停止一个正在运行的线程? run方法完成后程序终止/使用stop方法强行终止/使用interrupt方法中断线程

1. ~~使用退出标志，使线程正常退出，也就是当~~ **run方法完成后线程终止。**
2. **使用 stop() 方法强行终止**，但是不推荐这个方法，因为 `stop` 和`suspend` 及 `resume` 一样 都是被 `@Deprecated` 修饰，也就是过期作废的方法。
3. **使用 Thread对象调用 interrupt() 方法中断线程。**

```
class MyThread extends Thread {
    volatile boolean stop = false;

    public void run() {
        while (!stop) {
            System.out.println(getName() + " is running");
            try {
                sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("week up from blcok...");
                stop = true; // 在异常处理代码中修改共享变量的状态
            }
        }
        System.out.println(getName() + " is exiting...");
    }
}
```

```
class InterruptThreadDemo3 {
    public static void main(String[] args) throws InterruptedException {
        MyThread m1 = new MyThread();
        System.out.println("Starting thread...");
        m1.start();
        Thread.sleep(3000);
        System.out.println("Interrupt thread...: " + m1.getName());
        m1.stop = true; // 设置共享变量为true
        m1.interrupt(); // 阻塞时退出阻塞状态
        Thread.sleep(3000); // 主线程休眠 3 秒以便观察线程m1的中断情况
        System.out.println("Stopping application...");
    }
}
```

3 、notify()和notifyAll()有什么区别？ 是否会导致死锁/ 唤醒几个处于wait状态的线程/ notify需要正确使用

- `notify` 可能会导致死锁，而 `notifyAll` 则不会
- 任何时候只有一个线程可以获得锁，也就是说只有一个线程可以运行 synchronized 中的代码
- 使用 `notifyall`,可以唤醒 所有处于 `wait` 状态的线程，使其重新进入锁的争夺队列中，而 `notify` 只能唤醒一个。
- wait() 应配合while循环使用，不应使用if，务必在wait()调用前后都检查条件，如果不满足，必须调用notify()唤醒另外的线程来处理，自己继续wait()直至条件满足再往下执行。

`notify()` 是对 `notifyAll()` 的一个优化，但它有很精确的应用场景，并且要求正确使用。不然可能导致死锁。正确的场景应该是 WaitSet中等待的是相同的条件，
唤醒任一个都能正确处理接下来的事项，如果唤醒的线程无法正确处理，务必确保继续notify()下一个线程，并且自身需要重新回到 WaitSet中.

4 、sleep()和wait() 有什么区别？ sleep没有释放锁， sleep属于线程

答案一：javaGuide

- 两者最主要的区别在于：`sleep()` **方法没有释放锁，而** `wait()` **方法释放了锁** 。sllep()完成后线程会自动苏醒
- 两者都可以暂停线程的执行。
- `wait()` 通常被用于线程间交互/通信，`sleep()` 通常被用于暂停执行。
- `wait()` 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 `notify()` 或者 `notifyAll()` 方法。
  `sleep()` 方法执行完成后，线程会自动苏醒。
  或者可以使用 `wait(long timeout)` 超时后线程会自动苏醒。
- `sleep()` 方法是属于 `Thread` 类中的。而 `wait()` 方法则是属于 `Object` 类 中的。

```
/**
 * @see java.lang.Object#wait()
 * @see java.lang.Thread#sleep(long)
 */
```

答案二： 面试题.pdf

sleep()方法是属于`Thread`类中的。而wait()方法则是属于`Object`类 中的。

sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。
在调用sleep()方法的过程中，线程不会释放对象锁。

当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备，获取对象锁进入运行状态。

5 、volatile 是什么?可以保证有序性吗? `轻量级`的同步机制，修饰共享变量

`volatile` 是Java虚拟机提供的`轻量级`的同步机制,用来修饰共享变量（**类的成员变量、类的静态成员变量**），

它有三大特性：

- 保证可见性：一个线程修改了某个变量的值，新的值对其他线程来说是立即可见的,volatile关键字会强制将修改的值立即写入主存
- 不保证原子性：`原子性` 就是某个线程正在做某个具体业务时，中间不可以被其他线程加塞或者被分割。 volatile不保证原子性（解决办法 1.synchronized 2. JUC下的原子类）
- 禁止指令重排： 计算机在执行程序时，为了提高性能，编译器和处理器常常会对指令重排，一般分为以下三种：**源代码 -> 编译器优化的重排 -> 指令并行的重排 -> 内存系统的重排 -> 最终执行指令**

  `volatile`在日常的单线程环境是应用不到的
  使用volatile 一般用于 `状态标记量` 和 `多线程下单例模式的双检锁`。

你在哪些地方用到过volatile?

1. 多线程下单例模式的双端检锁机制 DCL Double Check Lock
2. 读写锁,手写一个缓存的时候
3. CAS底层源码分析时 JUC的源码包中大量用到 volatile

对volatile的具体详细理解

什么叫保证部分有序性?
当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

```
x = 2; //语句 1
y = 0; //语句 2
flag = true; //语句 3
x = 4; //语句 4
y = -1; //语句 5
```

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句 3 放到语句 1 、语句 2 前面，也不会讲语句 3 放到语句 4 、语句 5 后面。
但是要注意语句 1 和语句 2 的顺序、语句 4 和语句 5 的顺序是不作任何保证的。

6 、Thread 类中的start() 和 run() 方法有什么区别？ start()内部调用了run()方法。调用run()方法的时候，只会是在原来的线程中调用

start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。
当 **调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动**，start()方法才会启动新线程。

7 、为什么wait, notify 和 notifyAll这些方法不在 Thread 类里面？由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象

~~明显的原因是~~ **JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。**如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。
如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，**由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。**

8 、为什么wait和notify方法要在同步块中调用？ 只有在调用线程拥有某个对象的独占锁时，才能够调用该对象的wait(),notify()和notifyAll()方法

1. 只有在调用线程拥有某个对象的独占锁时，才能够调用该对象的wait(),notify()和notifyAll()方法。
2. 如果你不这么做，你的代码会抛出IllegalMonitorStateException异常。
3. 还有一个原因是为了避免wait和notify之间产生竞态条件。

wait()方法强制当前线程释放对象锁。这意味着在调用某对象的wait()方法之前，当前线程必须已经获得该对象的锁。因此线程必须在某个对象的同步方法或同步代码块中才能调用该对象的wait()方法。

在调用对象的notify()和notifyAll()方法之前，调用线程必须已经得到该对象的锁。因此，必须在某个对象的同步方法或同步代码块中才能调用该对象的notify()或notifyAll()方法。

调用wait()方法的原因通常是，**调用线程希望某个特殊的状态(或变量)被设置之后再继续执行。**调用notify()或notifyAll()方法的原因通常是，调用线程希望告诉其他等待中的线程:
"**特殊状态已经被设置**"。这个状态作为线程间通信的通道，它必须是一个可变的共享状态(或变量)。

9 、Java中 interrupted() 和 isInterrupted() 方法的区别？interrupted() 会将 线程的中断状态

interrupted() 和 isInterrupted() 的主要区别是 interrupted() 会将 `线程的中断状态（interrupted status of the thread）` 清除 而 isInterrupted() 不会。

Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。

当中断线程调用静态方法Thread.interrupted()来检查中断状态时，中断状态会被清零。
而非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。
无论如何，一个线程的中断状态有可能被其它线程调用中断来改变。

~~interrupted() 是线程类的静态方法 。isInterrupted() 是线程类的实例方法~~

```
/**
 * @see java.lang.Thread#interrupted()
 * @see java.lang.Thread#isInterrupted()
 */
```

10 、Java中 synchronized 和 ReentrantLock 有什么不同?

相似点:

这两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，
其他访问该同步块的线程都必须阻塞在同步块外面等待，而进行线程阻塞和唤醒的代价是比较高的.

区别:

这两种方式最大区别就是对于 `Synchronized` 来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现。
而 `ReentrantLock` 它是JDK 1.5之后提供的 API 层面的互斥锁，需要lock()和 unlock()方法配合 try/finally 语句块来完成。

`synchronized` 经过编译，会在同步块的前后分别形成 monitorenter 和 monitorexit 这个两个字节码指令。
在执行monitorenter指令时，首先要尝试获取对象锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，把锁的计算器加 1 ，
相应的，在执行monitorexit指令时会将锁计算器就减 1 ，当计算器为 0 时，锁就被释放了。

如果获取对象锁失败，那当前线程就要阻塞，直到对象锁被另一个线程释放为止。

由于 ReentrantLock 是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，
ReentrantLock 类提供了一些高级功能，主要有以下 3 项:

1. `ReentrantLock` 等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于 Synchronized 来说可以避免出现死锁的情况。

2.是否公平锁： 多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，
 可以通过参数true设为公平锁，但公平锁表现的性能不是很好。

3.锁绑定多个条件，一个 `ReentrantLock` 对象可以同时绑定多个对象。

- **公平锁**： 按照线程在队列中的排队顺序，先到者先拿到锁
- **非公平锁**： 当线程要获取锁时，先通过两次 CAS 操作去抢锁，如果没抢到，当前线程再加入到队列中等待唤醒。

11 、有三个线程T1,T2,T3,如何保证顺序执行？ Thread类的 实例方法`join()`

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的实例方法 `join()` 方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。
为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

实际上先启动三个线程中哪一个都行， 因为在每个线程的run方法中用join方法限定了三个线程的执行顺序。

```
```

```
public class JoinTest2 {
    // 1.现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行
    public static void main(String[] args) {
        final Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("t1");
            }
        });
        final Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
// 引用t1线程，等待t1线程执行完
                    t1.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("t2");
            }
        });
        Thread t3 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
// 引用t2线程，等待t2线程执行完
                    t2.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("t3");
            }

        });
        t3.start();//这里三个线程的启动顺序可以任意，大家可以试下！
        t2.start();
        t1.start();
    }
}
```

12 、SynchronizedMap 和 ConcurrentHashMap 有什么区别?

TODO： 理解 ConcurrentHashMap

```
/**
 * @see java.util.concurrent.ConcurrentHashMap
 */
```

SynchronizedMap 一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为 map。

ConcurrentHashMap 使用分段锁来保证在多线程下的性能。

ConcurrentHashMap 中则是一次锁住一个桶。ConcurrentHashMap 默认将hash 表分为 16 个桶，诸如 get,put,remove 等常用操作只锁当前需要用到的桶。

这样，原来只能一个线程进入，现在却能同时有 16 个写线程执行，并发性能的提升是显而易见的。

另外 ConcurrentHashMap 使用了一种不同的迭代方式。在这种迭代方式中，当iterator 被创建后集合再发生改变就不再是抛出 ConcurrentModificationException，

取而代之的是在改变时 new 新的数据从而不影响原有的数据 ， iterator 完成后再将头指针替换为新的数据 ， 这样 iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。

~~SynchronizedMap 和Hashtable一样，实现上在调用map所有方法时，都对整个map进行同步。~~
~~而ConcurrentHashMap的实现却更加精细，它对map中的所有桶加了锁。所以，只要有一个线程问map，其他线程就无法进入map，~~
~~而如果一个线程在访问ConcurrentHashMap某个桶时，其他线程，仍然可以对map执行某些操作。~~

所以，ConcurrentHashMap在性能以及安全性方面，明显比Collections.synchronizedMap()更加有优势。同时，同步操作精确控制到桶，
这样，即使在遍历map时，如果其他线程试图对map进行数据修改，也不会抛出ConcurrentModificationException。

13 、什么是线程安全？ 多线程访问同一段代码产生的结果是确定的

线程安全就是说多线程访问同一段代码，产生的结果是确定的。

~~又是一个理论的问题，各式各样的答案有很多，我给出一个个人认为解释地最好的：~~

如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。

这个问题有值得一提的地方，

就是线程安全也是有几个级别的：

1. 不可变
   像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用
2. 绝对线程安全
   不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，~~Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，~~ 比方说JUC包下的 CopyOnWriteArrayList、CopyOnWriteArraySet、ConcurrentHashMap
3. 相对线程安全
   相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，
   如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是fail-fast机制。
4. 线程非安全
   ArrayList、LinkedList、HashMap 等都是线程非安全的类

14 、Thread类中的 yield 方法有什么作用?

`yield()` 方法可以**暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。**它是一个静态方法而且只保证当前线程放弃CPU占用而不能保证使其它线程一定能占用CPU，
执行yield()的线程有可能在进入到暂停状态后马上又被执行。
yield 单词是 让步

15 、Java线程池中submit() 和 execute()方法有什么区别?

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有 `异步计算结果` 的 `Future` 对象，
它定义在 `ExecutorService` 接口中，它扩展了Executor接口，其它线程池类像 `ThreadPoolExecutor` 和 `ScheduledThreadPoolExecutor` 都有这些方法。

```
/**
 * @see java.util.concurrent.Executor#execute(java.lang.Runnable)
 * @see java.util.concurrent.ExecutorService#submit(java.lang.Runnable)
 * @see java.util.concurrent.ThreadPoolExecutor;
 * @see java.util.concurrent.ScheduledThreadPoolExecutor;
 */
```

16 、说一说自己对于 synchronized 关键字的了解

`synchronized` 关键字解决的是多个线程之间访问资源的同步性，`synchronized` 关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在 Java 早期版本中，`synchronized` 属于 **重量级锁**，效率低下。

为什么呢?

因为监视器锁（monitor）是依赖于底层的操作系统的 `Mutex Lock` 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，
而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高。

庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 `synchronized` 较大优化，所以现在的 `synchronized` 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化，
如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

~~所以，你会发现目前的话，不论是各种开源框架还是 JDK 源码都大量使用了 `synchronized` 关键字。~~

17 、说说自己是怎么使用 synchronized 关键字?

synchronized 关键字最主要的三种使用方式:

1.修饰实例方法: 作用于当前对象实例加锁，进入同步代码前要获得 **当前对象实例的锁**

```
synchronized void method() {
    //业务代码
}
```

2.修饰静态方法: 也就是给当前类加锁，会**作用于类的所有对象实例** ，进入同步代码前要获得 **当前 class 的锁**。因为静态成员不属于任何一个实例对象，
是类成员（ *static 表明这是该类的一个静态资源，不管 new 了多少个对象，只有一份*）。所以，如果一个线程 A 调用一个实例对象的非静态 `synchronized` 方法，
而线程 B 需要调用这个实例对象所属类的静态 `synchronized` 方法，是允许的，不会发生互斥现象，
因为访问静态 `synchronized` 方法占用的锁是当前类的锁，而访问非静态 `synchronized` 方法占用的锁是当前实例对象锁。

```
synchronized static void method() {
    //业务代码
}
```

3.修饰代码块：指定加锁对象，对给定对象/类加锁。`synchronized(this|object)` 表示进入同步代码库前要获得**给定对象的锁。**`synchronized(类.class)` 表示进入同步代码前要获得 **当前 class 的锁**

```
synchronized(this) {
    //业务代码
}
```

总结：

- `synchronized` 关键字加到 `static` 静态方法和 `synchronized(class)` 代码块上都是是给 Class 类上锁。
- `synchronized` 关键字加到实例方法上是给对象实例上锁。
- 尽量不要使用 `synchronized(String a)` 因为 JVM 中，字符串常量池具有缓存功能！

18 、什么是线程安全？Vector是一个线程安全类吗？

线程安全就是说多线程访问同一段代码，产生的结果是确定的。 一个线程安全的计数器类的同一个实例对象在被多个线程使用的情况下也不会出现计算失误。

集合类可以分成线程安全和非线程安全的集合类两组。
Vector 是一个线程安全类, 是用同步方法来实现线程安全的, 而和它相似的 ArrayList不是线程安全的。

19 、 volatile 关键字的作用？

重复

20 、常用的线程池有哪些？ Executors工具类的方法： 单线程线程池、固定大小线程池、缓存线程池、周期线程池

> Executors工具类的方法： 单线程线程池、固定大小线程池、缓存线程池、周期线程池

- newSingleThreadExecutor：创建一个单线程的线程池，此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
- newFixedThreadPool：创建固定大小的线程池，每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。
- newCachedThreadPool：创建一个可缓存的线程池，此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。
- newScheduledThreadPool：创建一个大小无限的线程池，此线程池支持定时以及周期性执行任务的需求。

```
/**
 * @see java.util.concurrent.ThreadPoolExecutor
 * @see java.util.concurrent.Executors#newSingleThreadExecutor() ()
 * @see java.util.concurrent.Executors#newFixedThreadPool(int) ()
 * @see java.util.concurrent.Executors#newCachedThreadPool() ()
 * @see java.util.concurrent.Executors#newScheduledThreadPool(int)
 */
```

/**

21 、简述一下你对线程池的理解。（说一下线程池如何用、线程池的好处、线程池的启动策略）

> **~~线程池、数据库连接池、Http 连接池等等都~~ 是对池化技术思想的应用。池化技术的思想主要是为了减少每次获取资源的消耗，提高对资源的利用率。**

合理利用线程池能够带来三个好处。

1. **降低资源消耗。**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
2. **提高响应速度。**当任务到达时，任务可以不需要等到线程创建就能立即执行。
3. **提高线程的可管理性。**线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进**行统一的分配，调优和监控。**

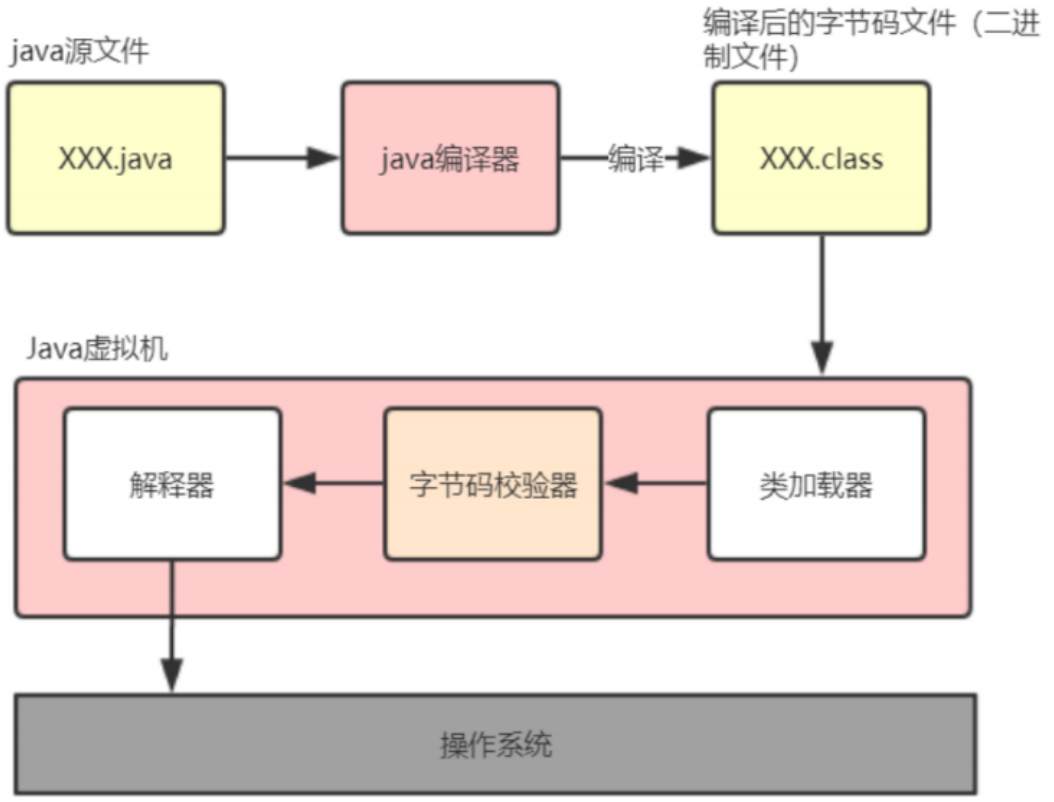
22 、Java程序是如何执行的

~~我们日常的工作中都使用开发工具（Intellij IDEA 或 Eclipse 等）可以很方便的调试程序，或者是通过打包工具把项目打包成 jar 包或者 war 包，~~
~~放入 Tomcat 等 Web 容器中就可以正常运行了，但你有没有想过 Java 程序内部是如何执行的？其实不论是在开发工具中运行还是在 Tomcat 中运行，~~
~~Java 程序的执行流程基本都是相同的，~~

Java程序的 执行流程如下：

- 先把 Java 代码编译成字节码，也就是把 `.java` 类型的文件编译成 `.class` 类型的文件。这个过程的大致执行流程：
  Java 源代码 -> 词法分析器 -> 语法分析器 -> 语义分析器 -> 字符码生成器 -> 最终生成字节码，其中任何一个节点执行失败就会造成编译失败；
- 把 class 文件放置到 Java 虚拟机，这个虚拟机通常指的是 Oracle 官方自带的 Hotspot JVM；
- Java 虚拟机使用类加载器（Class Loader）装载 class 文件；
- 类加载完成之后，会进行字节码效验，字节码效验通过之后 JVM 解释器会把 `字节码` 翻译成 `机器码` 交由操作系统执行。但不是所有代码都是解释执行的，
  JVM 对此做了优化，比如，以Hotspot 虚拟机来说，它本身提供了 JIT（Just In Time）也就是我们通常所说的动态编译器， 它能够在运行时将热点代码编译为机器码，

这个时候字节码就变成了编译执行。Java 程序执行流程图如下：



-

23 、锁的优化机制了解吗？ 锁的优化机制包括 自旋锁、自适应锁、锁消除、锁粗化、轻量级锁和偏向锁

> 根据进程访问资源的特点，我们可以把进程在系统上的运行分为两个级别：
> 1. 用户态(user mode) : 用户态运行的进程可以直接读取用户程序的数据。
> 2. 系统态(kernel mode):可以简单的理解系统态运行的进程或程序几乎可以访问计算机的任何资源，不受限制。

从JDK1.6版本之后，`synchronized` 本身也在不断优化锁的机制，有些情况下他并不会是一个很重量级的锁了。

锁的优化机制包括自旋锁、自适应锁、锁消除、锁粗化、轻量级锁和偏向锁。

锁的状态从低到高依次为无锁->偏向锁->轻量级锁->重量级锁，升级的过程就是从低到高，降级在一定条件也是有可能发生的。

自旋锁：**在互斥同步的时候，对性能影响最大的就是阻塞的实现，挂起线程，恢复线程等的操作都需要用户态转为内核态去完成。这些操作给性能带来了巨大的压力。**
由于大部分时候，锁被占用的时间很短，共享变量的锁定时间也很短，所有没有必要挂起线程，用户态和内核态的来回上下文切换严重影响性能。
自旋的概念就是让线程执行一个忙循环，可以理解为就是啥也不干，**防止从用户态转入内核态，避免用户态和内核态的来回上下文切换导致的影响性能**，自旋锁可以通过设置-XX:+UseSpining来开启，自旋的默认次数是 10 次，可以使用`-XX:PreBlockSpin`设置。
如果占有对象锁的线程在很短的时间内就执行完，然后释放锁，这样的话，自旋锁的效果就会非常好。

自适应锁：自适应锁就是自适应的自旋锁，自旋的时间不是固定时间，而是**由前一次在同一个锁上的自旋时间和锁的持有者状态来决定。**

锁消除：锁消除指的是**JVM检测到一些同步的代码块，完全不存在数据竞争的场景，也就是不需要加锁，就会进行锁消除。**

锁粗化：锁粗化指的是有很多操作都是对同一个对象进行加锁，就会把加锁同步的范围粗化到整个操作的最外层。

- 如果**一段代码中自始至终都锁的是同一个对象**，那么就会对这个对象进行重复的加锁，释放，加锁，释放。频繁的进行用户态和内核态的切换，效率居然变低了。通过锁粗话可以提高效率

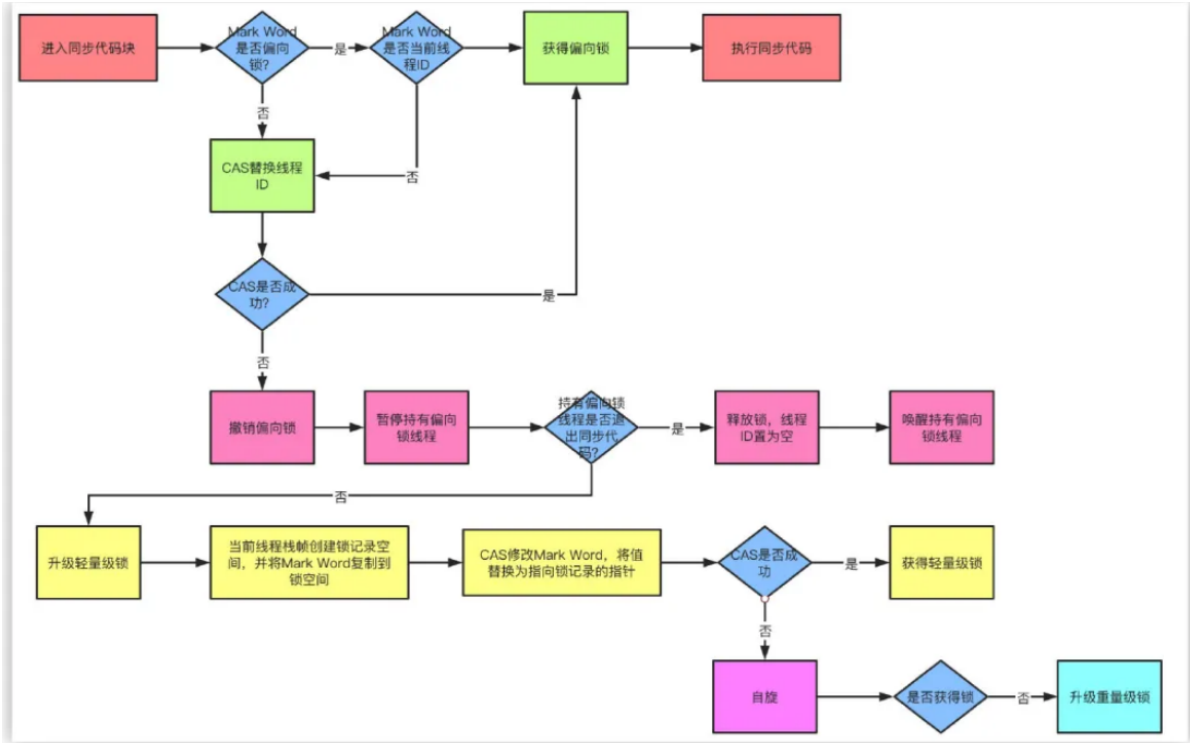
偏向锁：当线程访问同步块获取锁时，会在对象头和栈帧中的锁记录里存储偏向锁的线程ID，之后这个线程再次进入同步块时都不需要CAS来加锁和解锁了，偏向锁会永远偏向第一个获得锁的线程，
如果后续没有其他线程获得过这个锁，持有锁的线程就永远不需要进行同步，反之，当有其他线程竞争偏向锁时，持有偏向锁的线程就会释放偏向锁。可以用过设置`-XX:+UseBiasedLocking`开启偏向锁。

轻量级锁：JVM的对象的对象头中包含有一些锁的标志位，代码进入同步块的时候，JVM将会使用 CAS方式来尝试获取锁，如果更新成功则会把对象头中的状态位标记为轻量级锁， 如果更新失败，当前线程就尝试自旋来获得锁。

轻量级锁提升性能的依据是：绝大多数的锁在整个同步过程中都是不存在竞争的。这样的话，就通过CAS操作避免了使用操作系统中互斥量的开销。

整个锁升级的过程非常复杂，我尽力去除一些无用的环节，简单来描述整个升级的机制。

简单点说，偏向锁就是通过对象头的偏向线程ID来对比，甚至都不需要CAS了，而轻量级锁主要就是通过CAS修改对象头锁记录和自旋来实现，重量级锁则是除了拥有锁的线程其他全部阻塞。



[扩展-面试官：你了解Java中的锁优化吗？]

24 、说说进程和线程的区别？

1. 进程是一个“执行中的程序”，是系统进行资源分配和调度的一个独立单位。
2. 线程是进程的一个实体，一个进程中拥有多个线程，线程之间共享地址空间和其它资源（所以通信和同步等操作线程比进程更加容易）
3. 线程上下文的切换比进程上下文切换要快很多。

- 进程切换时，涉及到当前进程的CPU环境的保存和新被调度运行进程的CPU环境的设置。
- 线程切换仅需要保存和设置少量的寄存器内容，不涉及存储管理方面的操作。

25，产生死锁的四个必要条件？ 互斥条件/请求与保持/不剥夺条件/循环等待条件

1. **互斥条件**：一个资源每次只能被一个线程使用
2. **请求与保持条件**：一个线程因请求资源而阻塞时，对已获得的资源保持不放
3. **不剥夺条件**：进程已经获得的资源，在未使用完之前，不能强行剥夺
4. **循环等待条件**：若干线程之间形成一种头尾相接的循环等待资源关系

死锁demo

```
/**
 * @see com.atguigu.review.deadlock.DeadLock;
 */
```

如何预防死锁？ 破坏死锁的产生的必要条件即可：

1. **破坏请求与保持条件** ： 一次性申请所有的资源。
2. **破坏不剥夺条件** ： 占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
3. **破坏循环等待条件** ： 靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

26 、如何避免死锁？ 指定获取锁的顺序

线程死锁描述的是这样一种情况：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。

指定获取锁的顺序，举例如下：

1. 比如某个线程只有获得A锁和B锁才能对某资源进行操作，在多线程条件下，如何避免死锁？
2. 获得锁的顺序是一定的，比如规定，只有获得A锁的线程才有资格获取B锁，按顺序获取锁就可以避免死锁！！！

27，线程池核心线程数怎么设置呢？ CPU 核心数 + 1 / IO密集型：CPU核心数量*2

分为CPU密集型和IO密集型

CPU密集型

这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N（CPU 核心数）+1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，
或者其它原因导致的任务暂停而带来的影响。一旦 任务暂停，CPU 就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。

IO密集型

IO密集型corePoolSize主流的配置方案有两种：

这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交出给其它线程使用。
因此在 I/O 密集型任务的应用中，我们 可以多配置一些线程，具体的计算方法是：

- 第一种： 核心线程数= CPU核心数量*2。

- 第二种：CPU核数 / (1 - 阻塞系数)　阻塞系数在0.8 ~ 0.9左右**
  例如：8核CPU：8/ (1 - 0.9) = 80个线程数

maxPoolSize 最大线程数在生产环境上我们往往设置成corePoolSize一样，这样可以减少在处理过程中创建线程的开销

28， Java线程池中队列常用类型有哪些?

ArrayBlockingQueue 是一个基于`数组结构`的`有界阻塞队列`，此队列按 FIFO（先进先出）原则对元素进行排序。
LinkedBlockingQueue 一个基于 `单向链表结构` 的有界阻塞队列(实际上是Integer.MAX_VALUE可以看做无界)，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。
SynchronousQueue 一个不存储元素的`无界阻塞队列`。
PriorityBlockingQueue 一个`具有优先级的` `无限阻塞队列`。PriorityBlockingQueue也是基于最小二叉堆实现
DelayQueue

- 只有当其指定的延迟时间到了，才能够从队列中获取到该元素。
- DelayQueue是一个没有大小限制的队列，
- 因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

```
BlockingQueue阻塞队列是属于一个接口，底下有七个实现类

- **ArrayBlockQueue：由数组结构组成的有界阻塞队列**
- **LinkedBlockingQueue：由链表结构组成的有界（但是默认大小 Integer.MAX_VALUE）的阻塞队列**
  - 有界，但是界限非常大，相当于无界，可以当成无界
- PriorityBlockQueue：支持优先级排序的无界阻塞队列
- DelayQueue：使用优先级队列实现的延迟无界阻塞队列
- **SynchronousQueue：不存储元素的阻塞队列，也即单个元素的队列**  （专属个人定制版，超级奢饰品订单不下工厂不开工）生产一个，消费一个，不存储元素，不消费不生产
- LinkedTransferQueue：由链表结构组成的无界阻塞队列
- LinkedBlocking**Deque**：由链表结构组成的双向阻塞队列
```

29， 线程安全需要保证几个基本特征?

- 原子性，简单说就是相关操作不会中途被其他线程干扰，一般通过同步机制实现。
- 可见性，是一个线程修改了某个共享变量，其状态能够立即被其他线程知晓，通常被解释为将线程本地状态反映到主内存上，volatile 就是负责保证可见性的。
- 有序性，是保证线程内串行语义，避免指令重排等。

30， 说一下线程之间是如何通信的?

线程之间的通信有两种方式：`共享内存`和`消息传递`。

共享内存

在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信。典型的共享内存通信方式，就是通过共享对象进行通信。

例如上图线程 A 与 线程 B 之间如果要通信的话，那么就必须经历下面两个步骤：

1. 线程 A 把本地内存 A 更新过得共享变量刷新到主内存中去。

2. 线程 B 到主内存中去读取线程 A 之前更新过的共享变量。

消息传递

在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信。在 Java 中典型的消息传递方式，就是 wait() 和 notify() ，或者 BlockingQueue 。

31 、CAS的原理呢?

CAS并发原语体现在Java语言中就是`sun.misc.Unsafe`类的各个方法。调用`UnSafe`类中的CAS方法，JVM会帮我们实现出**CAS汇编指令**， 是一种**完全依赖于硬件**的功能，通过它实现了原子操作，
由于CAS是一种系统原语，原语属于操作系统用于范畴，是由若干条指令组成， 用于完成某个功能的一个过程，**并且原语的执行必须是连续的，在执行过程中不允许被中断(主席出行，交通管制，不允许其他车加塞)，**
也就是说CAS是一条CPU的原子指令，不会造成所谓的数据不一致的问题，也就是说CAS是线程安全的。

CAS是`compareAndSwap`，比较当前工作内存中的值和主物理内存中的值，如果相同则执行规定操作，否者继续比较直到主内存和工作内存的值一致为止

CAS叫做CompareAndSwap，比较并交换，主要是通过处理器的指令来保证操作的原子性，它包含三个操作数:

```
1. 变量内存地址，V表示
2. 旧的预期值，A表示
3. 准备设置的新值，B表示
```

CAS有3个操作数，内存值V，旧的预期值A，要修改的更新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否者什么都不做

32 、CAS有什么缺点吗?

CAS的缺点主要有 3 点:

- **ABA问题**：ABA的问题指的是在CAS更新的过程中，当读取到的值是A，然后准备赋值的时候仍然是 A，但是实际上有可能A的值被改成了B，然后又被改回了A，这个CAS更新的漏洞就叫做ABA。只是 ABA的问题大部分场景下都不影响并发的最终效果。

  Java中有`AtomicStampedReference`来解决这个问题，他加入了预期标志和更新后标志两个字段，更新时不光检查值，还要检查当前的标志是否等于预期标志，全部相等的话才会更新。
- 循环时间长开销大：自旋CAS的方式如果长时间不成功，会给CPU带来很大的开销。
- 只能保证一个共享变量的原子操作：只对一个共享变量操作可以保证原子性，但是多个则不行，多个可以通过`AtomicReference`来处理或者使用锁synchronized实现。

33 、引用类型有哪些？有什么区别?

引用类型主要分为强软弱虚四种:

- 强引用 :java中绝大部分都是强引用，就算报OOM，GC时也不回收
- 软引用 SoftReference：为了降低OOM发生概率。GC时，内存够就不回收，内存不够就回收
- 弱引用 WeakReference：只要发生GC就回收
- 虚引用：如果一个对象持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收，它不能单独使用也不能通过它访问对象，虚引用必须和引用队列 ReferenceQueue 联合使用

34 、说说 ThreadLocal 原理?

ThreadLocal 可以理解为 **线程本地变量**，他会在每个线程都创建一个副本，那么在线程之间访问内部副本变量就行了，做到了线程之间互相隔离，相比于 synchronized 的做法是**用空间来换时间。**

ThreadLocal 有一个静态内部类 `ThreadLocalMap` ，ThreadLocalMap 又包含了一个 Entry 数组，
Entry本身是一个弱引用，他的key是指向ThreadLocal的弱引用，
Entry具备了保存key value键值对的能力。

每个线程（`Thread`）中都具备一个自己的`ThreadLocalMap`对象，而`ThreadLocalMap`可以存储以`ThreadLocal`为 key，Object 对象为 value 的键值对。
比如我们在同一个线程中声明了两个 `ThreadLocal` 对象的话， `Thread`内部都是使用仅有的那个 `ThreadLocalMap` 存放数据的，
简而言之，存进ThreadLocal的数据，相当于存进了线程自己的变量中。因此 如果要**保证共享变量的隔离性**，就把他放进ThreadLocal（相当于在线程中拷贝了一份），要用时，从ThreadLocal中拿出来即可（拿出当前线程的副本）
注意：保证的是隔离性

`ThreadLocalMap`的 key 就是 `ThreadLocal`对象，value 就是 `ThreadLocal` 对象调用`set`方法设置的值。

弱引用的目的是为了防止内存泄露，如果是强引用那么ThreadLocal对象除非线程结束否则始终无法被回收，弱引用则会在下一次GC的时候被回收。

但是这样还是会存在内存泄露的问题，假如key和ThreadLocal对象被回收之后，entry中就存在key为null，但是value有值的entry对象，但是永远没办法被访问到，同样除非线程结束运行。

但是只要ThreadLocal使用恰当，在使用完之后调用remove方法删除Entry对象，实际上是不会出现这个问题的。

```
/**
 * @see java.lang.ThreadLocal;
 * @see java.lang.ThreadLocal.ThreadLocalMap;
 */
```

/**

35 、线程池原理知道吗？以及核心参数

```
/**
 * @see java.util.concurrent.ThreadPoolExecutor
 */
public ThreadPoolExecutor(int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable> workQueue,
        ThreadFactory threadFactory,
        RejectedExecutionHandler handler)
```

线程池在创建的时候，一共有7大参数

- **corePoolSize**：**核心线程数，线程池中的常驻核心线程数**

    - 在创建线程池后，当有请求任务来之后，就会安排池中的线程去执行请求任务，近似理解为今日当值线程
    - 当线程池中的线程数目达到 `corePoolSize` 后，就会把到达的队列放到缓存队列中
- **maximumPoolSize**：线程池能够容纳同时执行的最大线程数，此值必须大于等于1、
    - 相当有扩容后的线程数，这个线程池能容纳的最多线程数
- **keepAliveTime**：多余的空闲线程存活时间
    - 当线程池数量超过 `corePoolSize` 时，当空闲时间达到 `keepAliveTime` 值时，多余的空闲线程会被销毁，直到只剩下 `corePoolSize` 个线程为止
    - 默认情况下，只有当线程池中的线程数大于 `corePoolSize` 时，`keepAliveTime` 才会起作用
- **unit**：`keepAliveTime` 的单位
- **workQueue**：任务队列，被提交的但未被执行的任务（类似于银行里面的候客区）
    - LinkedBlockingQueue：链表阻塞队列
    - SynchronousBlockingQueue：同步阻塞队列
- **threadFactory**：表示生成线程池中工作线程的线程工厂，用于创建线程池 一般用默认即可
- **handler**：拒绝策略，表示 **当队列满了并且工作线程大于线程池的最大线程数**（maximumPoolSize3）时，如何来拒绝请求执行的Runnable的策略
  当提交一个新任务到线程池时，具体的执行流程如下：

线程池的工作原理：

1. 在创建了线程池后，等待提交过来的任务请求
2. 当调用 `execute()` 方法添加一个请求任务时，线程池会做出如下判断
    1. 如果正在运行的线程池数量小于 `corePoolSize` ，那么马上创建线程运行这个任务
    2. 如果正在运行的线程数量大于或等于 `corePoolSize` ，那么将这个任务放入队列
    3. 如果这时候队列满了，并且正在运行的线程数量还小于 `maximumPoolSize` ，那么还是创建非核心线程 来运行这个任务；
    4. 如果队列满了并且正在运行的线程数量大于或等于 `maximumPoolSize` ，那么线程池会启动饱和拒绝策略来执行
3. 当一个线程完成任务时，它会**从队列中取下一个任务来执行**
4. 当一个线程无事可做操作一定的时间(keepAliveTime)时，线程池会判断：
    1. 如果当前运行的线程数大于corePoolSize，那么这个线程就被停掉
    2. 所以线程池的所有任务完成后，它会最终收缩到corePoolSize的大小

以顾客去银行办理业务为例，谈谈线程池的底层工作原理

1. 最开始假设来了两个顾客，因为corePoolSize为2，因此这两个顾客直接能够去窗口办理
2. 后面又来了三个顾客，因为corePool已经被顾客占用了，因此只有去候客区，也就是阻塞队列中等待
3. 后面的人又陆陆续续来了，候客区可能不够用了，因此需要申请增加处理请求的窗口，这里的窗口指的是线程池中的线程数，以此来解决线程不够用的问题
4. 假设受理窗口已经达到最大数，并且请求数还是不断递增，此时候客区和线程池都已经满了，为了防止大量请求冲垮线程池，已经需要开启拒绝策略
5. 临时增加的线程会因为超过了最大存活时间，就会销毁，最后从最大数削减到核心数

36 、 线程池的拒绝策略有哪些?

主要有 4 种拒绝策略：

1. AbortPolicy：直接丢弃任务，抛出异常，这是默认策略
2. CallerRunsPolicy：只用调用者所在的线程来处理任务
3. DiscardOldestPolicy：丢弃等待队列中最旧的任务，并执行当前任务
4. DiscardPolicy：直接丢弃任务，也不抛出异常
5.

37 、说说你对JMM内存模型的理解？为什么需要JMM?

JMM是Java内存模型，也就是Java Memory Model，简称JMM，本身是一种抽象的概念，实际上**并不真实存在**，
它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式

JMM关于同步的规定：

- 线程解锁前，必须把共享变量的值刷新回主内存
- 线程加锁前，必须读取主内存的最新值，到自己的工作内存
- **加锁和解锁是同一把锁**

由于JVM运行程序的实体是线程，而每个线程创建时,JVM都会为其创建一个工作内存（有些地方称为栈空间），工作内存是每个线程的私有数据区域，
而Java内存模型中规定所有变量都存储在**主内存**，主内存是共享内存区域，所有线程都可以访问，
但线程对变量的操作（读取赋值等）必须在工作内存中进行，首先要将变量从主内存拷贝到自己的工作内存空间，然后对变量进行操作，操作完成后再将变量写会主内存，
不能直接操作主内存中的变量，各个线程中的工作内存中存储着主内存中的**变量副本拷贝**，因此不同的线程间无法访问对方的工作内存，线程间的通信（传值）必须
通过主内存来完成，

随着CPU和内存的发展速度差异的问题，导致CPU的速度远快于内存，所以现在的CPU加入了高速缓存，高速缓存一般可以分为L1、L2、L3三级缓存。基于上面的例子我们知道了这导致了缓存一致性的问题，
所以加入了缓存一致性协议，同时导致了内存可见性的问题，而编译器和CPU的重排序导致了原子性和有序性的问题，JMM内存模型正是对多线程操作下的一系列规范约束，因为不可能让程序员的代码去兼容所有的CPU，
通过JMM我们才屏蔽了不同硬件和操作系统内存的访问差异，这样保证了Java程序在不同的平台下达到一致的内存访问效果，同时也是保证在高效并发的时候程序能够正确执行。

- 原子性：Java内存模型通过read、load、assign、use、store、write来保证原子性操作，此外还有lock和unlock，直接对应着synchronized关键字的monitorenter和monitorexit字节码指令。
- 可见性：可见性的问题在上面的回答已经说过，Java保证可见性可以认为通过volatile、synchronized、final来实现。
- 有序性：由于处理器和编译器的重排序导致的有序性问题，Java通过volatile、synchronized来保证。

happen-before规则

虽然指令重排提高了并发的性能，但是Java虚拟机会对指令重排做出一些规则限制，并不能让所有的指令都随意的改变执行位置，主要有以下几点：

1. 单线程每个操作，happen-before于该线程中任意后续操作
2. volatile写happen-before与后续对这个变量的读

3. synchronized解锁happen-before后续对这个锁的加锁
4. final变量的写happen-before于final域对象的读，happen-before后续对final变量的读
5. 传递性规则，A先于B，B先于C，那么A一定先于C发生
6. 说了半天，到底工作内存和主内存是什么?

主内存可以认为就是物理内存，Java内存模型中实际就是虚拟机内存的一部分。而工作内存就是 CPU缓存，他有可能是寄存器也有可能是L1\L2\L3缓存，都是有可能的。

38 、多线程有什么用?

~~一个可能在很多人看来很扯淡的一个问题：我会用多线程就好了，还管它有什么用?在我看来，这个回答更扯淡。所谓"知其然知其所以然"，"会用"只是"知其然"，~~
~~"为什么用"才是"知其所以然"，只有达到"知其然知其所以然"的程度才可以说是把一个知识点运用自如。OK，下面说说我对这个问题的看法：~~

1. 发挥多核CPU的优势

> 随着工业的进步，现在的笔记本、台式机乃至商用的应用服务器至少也都是双核的， 4 核、 8 核甚至16核的也都不少见，如果是单线程的程序，那么在双核CPU上就浪费了50%，在 4 核CPU上就浪费了75%。
> 单核CPU上所谓的"多线程"那是假的多线程，同一时间处理器只会处理一段逻辑，只不过线程之间切换得比较快，看着像多个线程"同时"运行罢了。多核CPU上的多线程才是真正的多线程，它能让你的多段逻辑同时工作，
> 多线程，可以真正发挥出多核CPU的优势来，达到充分利用 CPU 的目的。

2. 防止阻塞

从程序运行效率的角度来看，单核CPU不但不会发挥出多线程的优势，反而会因为在单核CPU上运行多线程导致线程上下文的切换，而降低程序整体的效率。
但是单核CPU我们还是要应用多线程，就是为了防止阻塞。试想，如果单核CPU使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据吧，对端迟迟未返回又没有设置超时时间，
那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

3. 便于建模

这是另外一个没有这么明显的优点了。假设有一个大的任务A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务 D，分别建立程序模型，
并通过多线程分别运行这几个任务，那就简单很多了。

39 、说说 `CyclicBarrier` 和 `CountDownLatch` 的区别?

CyclicBarrier 循环使用的屏障(集齐龙珠) CountDownLatch 发射倒计时器（秦灭六国4）

两个看上去有点像的类，都在 java.util.concurrent 下，都可以用来表示代码运行到某个点上，二者的区别在于：

1. `CyclicBarrier` 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；
   CountDownLatch则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行
2. `CyclicBarrier` 只能唤起一个任务，CountDownLatch 可以唤起多个任务
3. `CyclicBarrier` 可重用，CountDownLatch 不可重用，计数值为 0 该 `CountDownLatch` 就不可再用了

40 、什么是AQS?

```
/**
 * @see java.util.concurrent.locks.AbstractQueuedSynchronizer;
 * @see java.util.concurrent.locks.AbstractQueuedSynchronizer#tryRelease(int)
 * @see java.util.concurrent.locks.AbstractQueuedSynchronizer#
 */
```

简单说一下AQS，AQS全称为 `AbstractQueuedSynchronizer` ，翻译过来应该是 `抽象队列同步器` 。
AQS 就是一个抽象类，主要用来构建锁和同步器。

如果说java.util.concurrent的基础是CAS的话，那么 `AQS` 就是整个 `Java并发包` （JUC:java.util.concurrent） 的核心了， ReentrantLock、CountDownLatch、Semaphore 等等都用到了它。
AQS实际上以双向队列的形式连接所有的Entry，比方说 `ReentrantLock` ，所有等待的线程都被放在一个Entry中并连成双向队列，前面一个线程使用ReentrantLock好了，
则双向队列实际上的第一个 Entry 开始运行。

AQS定义了对双向队列所有的操作，而只开放了 tryLock 和 tryRelease 方法给开发者使用，开发者可以根据自己的实现重写tryLock和tryRelease方法，以实现自己的并发功能。

AQS 简单介绍

AQS 的全称为 `AbstractQueuedSynchronizer` ，翻译过来的意思就是抽象队列同步器。这个类在 `java.util.concurrent.locks` 包下面。

AQS 就是一个抽象类，主要用来构建锁和同步器。

```
public abstract class AbstractQueuedSynchronizer extends
AbstractOwnableSynchronizer implements java.io.Serializable {
}
```

AQS 为构建锁和同步器提供了一些通用功能的是实现，因此，使用 **AQS 能简单且高效地构造出应用广泛的大量的同步器**，比如我们提到的 `ReentrantLock` ， `Semaphore` ，
其他的诸如 `ReentrantReadWriteLock` ， `SynchronousQueue` ， `FutureTask` (jdk1.7) 等等皆是基于 AQS 的。

AQS 原理

> 在面试中被问到并发知识的时候，大多都会被问到“请你说一下自己对于 AQS 原理的理解”。下面给大家一个示例供大家参考，面试不是背题，大家一定要加入自己的思想，即使加入不了自己的思想也要保证自己能够通俗的讲出来而不是背出来。

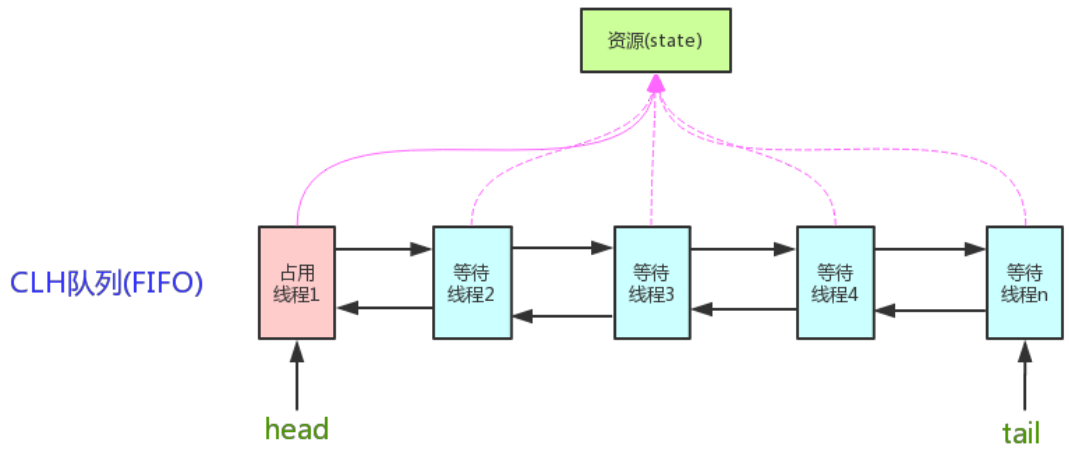
~~下面大部分内容其实在 AQS 类注释上已经给出了，不过是英语看着比较吃力一点，感兴趣的话可以看看源码。~~

AQS 原理概览

AQS 核心思想是，**如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制**，
这个机制 AQS 是用 **CLH 队列锁**实现的，即将暂时获取不到锁的线程加入到队列中。

> CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS 是将每条请求共享资源的线程封装成一个 CLH 锁队列的一个结点（Node）来实现锁的分配。

看个 AQS(`AbstractQueuedSynchronizer`)原理图：



AQS 使用一个 int 成员变量来表示同步状态，通过内置的 FIFO 队列来完成获取资源线程的排队工作。AQS 使用 CAS 对该同步状态进行原子操作实现对其值的修改。

```
private volatile int state;//共享变量，使用volatile修饰保证线程可见性
```

状态信息通过 `protected` 类型的`getState()`，`setState()`，`compareAndSetState()` 进行操作

```
//返回同步状态的当前值
protected final int getState() {
        return state;
}
 // 设置同步状态的值
protected final void setState(int newState) {
        state = newState;
}
//原子地（CAS操作）将同步状态值设置为给定值update如果当前同步状态的值等于expect（期望值）
protected final boolean compareAndSetState(int expect, int update) {
        return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
```

AQS 对资源的共享方式

AQS 定义两种资源共享方式: 独占和共享

1)Exclusive（独占）

只有一个线程能执行，如 `ReentrantLock`。又可分为公平锁和非公平锁，`ReentrantLock` 同时支持两种锁，下面以 `ReentrantLock` 对这两种锁的定义做介绍：

- **公平锁**：按照线程在队列中的排队顺序，先到者先拿到锁
- **非公平锁**：当线程要获取锁时，先通过两次 CAS 操作去抢锁，如果没抢到，当前线程再加入到队列中等待唤醒。

> 说明：下面这部分关于 `ReentrantLock` 源代码内容节选自：https://www.javadoop.com/post/AbstractQueuedSynchronizer-2 ，这是一篇很不错文章，推荐阅读。

下面来看 `ReentrantLock` 中相关的源代码：

`ReentrantLock` 默认采用非公平锁，因为考虑获得更好的性能，通过 `boolean` 来决定是否用公平锁（传入 true 用公平锁）。

```
/** Synchronizer providing all implementation mechanics */
private final Sync sync;
public ReentrantLock() {
    // 默认非公平锁
    sync = new NonfairSync();
}
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

`ReentrantLock` 中公平锁的 `lock` 方法

```
static final class FairSync extends Sync {
    final void lock() {
        acquire(1);
    }
    // AbstractQueuedSynchronizer.acquire(int arg)
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
    protected final boolean tryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            // 1. 和非公平锁相比，这里多了一个判断：是否有线程在等待
            if (!hasQueuedPredecessors() &&
                compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0)
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }
}
```

非公平锁的 `lock` 方法：

```
static final class NonfairSync extends Sync {
    final void lock() {
        // 2. 和公平锁相比，这里会直接先进行一次CAS，成功就返回了
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }
    // AbstractQueuedSynchronizer.acquire(int arg)
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
    protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
}
/**
 * Performs non-fair tryLock.  tryAcquire is implemented in
 * subclasses, but both need nonfair try for trylock method.
 */
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        // 这里没有对阻塞队列进行判断
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

总结：公平锁和非公平锁只有两处不同：

1. 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。
2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 `tryAcquire` 方法，在 `tryAcquire` 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

2)Share（共享）

多个线程可同时执行，如 `Semaphore/CountDownLatch`。`Semaphore`、`CountDownLatCh`、`CyclicBarrier`、`ReadWriteLock` 我们都会在后面讲到。

`ReentrantReadWriteLock` 可以看成是组合式，因为 `ReentrantReadWriteLock` 也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS 已经在上层已经帮我们实现好了。

AQS 底层使用了模板方法模式

同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

1. 使用者继承 `AbstractQueuedSynchronizer` 并重写指定的方法。（这些重写方法很简单，无非是对于共享资源 state 的获取和释放）
2. 将 AQS 组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用。

AQS 使用了模板方法模式，自定义同步器时需要重写下面几个 AQS 提供的钩子方法：

```
protected boolean tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。
protected boolean tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。
protected boolean tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
protected boolean tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。
protected boolean isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。
```

什么是钩子方法呢？ 钩子方法是一种被声明在抽象类中的方法，一般使用 `protected` 关键字修饰，它可以是空方法（由子类实现），也可以是默认实现的方法。模板设计模式通过钩子方法控制固定步骤的实现。

篇幅问题，这里就不详细介绍模板方法模式了，不太了解的小伙伴可以看看这篇文章：用Java8 改造后的模板方法模式真的是 yyds!。

除了上面提到的钩子方法之外，AQS 类中的其他方法都是 `final` ，所以无法被其他类重写。

以 `ReentrantLock` 为例，state 初始化为 0，表示未锁定状态。A 线程 `lock()` 时，会调用 `tryAcquire()` 独占该锁并将 `state+1` 。此后，其他线程再 `tryAcquire()` 时就会失败，直到 A 线程 `unlock()` 到 `state=0`（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A 线程自己是可以重复获取此锁的（state 会累加），这就是可重入的概念。但要注意，获取多少次就要释放多少次，这样才能保证 state 是能回到零态的。

再以 `CountDownLatch` 以例，任务分为 N 个子线程去执行，state 也初始化为 N（注意 N 要与线程个数一致）。这 N 个子线程是并行执行的，每个子线程执行完后`countDown()` 一次，state 会 CAS(Compare and Swap) 减 1。等到所有子线程都执行完后(即 `state=0`)，会 `unpark()` 主调用线程，然后主调用线程就会从 `await()` 函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现`tryAcquire-tryRelease`、`tryAcquireShared-tryReleaseShared`中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式，如`ReentrantReadWriteLock`。

推荐两篇 AQS 原理和相关源码分析的文章：

- Java并发之AQS详解
- Java并发包基石-AQS详解

41 、了解 Semaphore 吗?

Semaphore 就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore有一个构造函数， 可以传入一个int型整数n，表示某段代码最多只有n个线程可以访问，如果超出了n，那么请等待，
等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

Semaphore 关键方法：得到 `acquire()` 释放 `release()`
信号量主要用于两个目的

- 一个是用于共享资源的互斥使用
- 另一个用于并发线程数的控制
  比较经典的demo：抢车位

42 、什么是 Callable和 Future?

`Callable` 接口类似于 `Runnable`，是实现多线程的方式之一， 但是Runnable不会返回结果，并且无法抛出返回结果的异常，而Callable功能更强大一些，被线程执行后，可以返回值，
这个返回值可以被 Future 拿到，也就是说，**Future 可以拿到异步执行任务的返回值**。可以认为是带有回调的 Runnable。

Future 接口表示异步任务，是还没有完成的任务给出的未来结果。所以说Callable用于产生结果，Future 用于获取结果。

43 、什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型?

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。

这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

JDK7提供了 7 个阻塞队列。分别是：

- ArrayBlockingQueue ： 一个由数组结构组成的有界阻塞队列。
- LinkedBlockingQueue ： 一个由链表结构组成的有界阻塞队列。
- PriorityBlockingQueue ： 一个支持优先级排序的无界阻塞队列。
- DelayQueue：一个使用优先级队列实现的无界阻塞队列。

- SynchronousQueue：一个不存储元素的阻塞队列。
- LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
- LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

Java 5之前实现同步存取时，可以使用普通的一个集合，然后在使用线程的协作和线程同步可以实现生产者，消费者模式，主要的技术就是用好，wait ,notify,notifyAll,sychronized这些关键字。
而在java 5之后，可以使用阻塞队列来实现，此方式大大简少了代码量，使得多线程编程更加容易，安全方面也有保障。

BlockingQueue接口是Queue的子接口，它的主要用途并不是作为容器，而是作为线程同步的的工具，因此他具有一个很明显的特性，当生产者线程试图向BlockingQueue放入元素时，
如果队列已满，则线程被阻塞，当消费者线程试图从中取出一个元素时，如果队列为空，则该线程会被阻塞，正是因为它所具有这个特性，所以在程序中多个线程交替向BlockingQueue中放入元素，
取出元素，它可以很好的控制线程之间的通信。

阻塞队列使用最经典的场景就是socket客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。

44 、什么是多线程中的上下文切换?

在上下文切换过程中，CPU会停止处理当前运行的程序，并保存当前程序运行的具体位置以便之后继续运行。从这个角度来看，上下文切换有点像我们同时阅读几本书，在来回切换书本的同时我们需要记住每本书当前读到的页码。
在程序中，上下文切换过程中的“页码”信息是保存在**进程控制块** `PCB:Process Control Block` 中的。PCB还经常被称 作“切换桢”（switchframe）。“页码”信息会一直保存到CPU的内存中，直到他们被再次使用。

上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

45 、什么是 Daemon 线程？它有什么意义?

Daemon 线程 是后台线程，也叫守护线程，是指 **在程序运行的时候在后台提供一种通用服务的线程，并且这个线程并不属于程序中不可或缺的部分。**

因此，当所有的非后台线程结束时，程序也就终止了，同时会杀死进程中的所有后台线程。反过来说，只要有任何非后台线程还在运行，程序就不会终止。

必须在线程启动之前调用setDaemon()方法，才能把它设置为后台线程。注意：后台进程在不执行 finally 子句的情况下就会终止其run()方法。

比如：JVM的垃圾回收线程就是 Daemon线程，`Finalizer` 也是守护线程。

46 、乐观锁 和悲观锁的理解及如何实现，有哪些实现方式？ 悲观锁实现：关系型数据库，synchronized 乐观锁实现：CAS，版本号表示

- **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。

传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如Java里面的同步原语 `synchronized` 关键字的实现也是悲观锁。

- **乐观锁**：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，使用版本号等机制，每次更新的时

候同时维护一个version字段

乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write_condition机制，其实都是提供的乐观锁。

在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

乐观锁的实现方式：

1 、使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

2 、java中的Compare and Swap即CAS ，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。
CAS 操作中包含三个操作数 —— 需要读写的内存位置（V）、进行比较的预期原值（A）和 拟写入的新值(B)。如果内存位置V的值与预期原值A相匹配，那么处理器会自动将该位置值更新为新值B。否则处理器不做任何操作。

CAS缺点：

这时候线程one进行CAS操作发现内存中仍然是A，然后one操作成功。尽管线程one的CAS操作成功，但可能存在潜藏的问题。
从Java1.5开始JDK的atomic包里提供了一个类 `AtomicStampedReference` 来解决ABA问题。
2. 循环时间长开销大：对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。
3. 只能保证一个共享变量的原子操作：当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，
循环CAS就无法保证操作的原子性，这个时候就可以用锁。