线程Thread

一、进程和线程概念

进程(process):操作系统上一个正在进行中的程序，进程会拥有当前程序的所有资源数据。

线程(thread):进程内部的一个执行功能单元，每个线程都会有自己独立的内存空间，一个进程内部会有一到多个线程组成。进程只是拥有一个程序的资源，而线程是负责获取资源来执行具体任务的，进程内部的资源会被所有线程所共享。

每个进程内部有一个主线程，以及若干工作线程，主线程可以用于启动其他的工作线程。

进程是操作系统分配资源的基本单元。

线程是操作系统执行任务的基本单元。

二、多线程的执行方式

1、并发（concurrent）

微观：线程轮流切换执行 宏观上：同时执行

多线程默认就是并发执行的

操作系统会将时间片划分很多个时间点，在每个时间点上只允许一个线程获得cpu，并进入cpu中执行他的任务，执行一段时间后就退出cpu，下一个线程再获得cpu进入执行，依次类推。

因为时间被划分足够细，切换足够快，所有在宏观上看多线程是在同时进行的，而微观上是轮流切换执行的，这种现象就叫线程的并发，并发并不是真正意义上的同时。

2、并发（parlen）

并行只会出现在多核cpu上，一个cpu在一个时间点上只能干一件事，但是多个cpu就可以在一个时间点上干很多事情，并行是真正意义上的同时。

三、线程的创建方式

1、继承Thread类

1、写一个类继承Thread类，该类实例就代表一个线程

2、重写run()方法，代表线程所要执行的具体任务是什么

3、创建线程实例

4、调用start()方法启动线程

2、实现Runnable接口

1、写一个类实现Runnable接口，该类实例代表线程所要执行的具体任务对象

2、重写run()方法，代表具体的任务是什么

3、创建任务实例

4、再创建一个线程，将任务实例作为参数传给线程

5、调用start()方法启动线程

一般创建线程推荐使用实现Runnable接口

1)解决java中单继承带来的弊端

2)可以让多个线程来做同一个任务

3)将线程实例和任务实例解耦合

线程面试题：

1)线程的创建方式（4种）

2)启动线程调用start()还是调用run()

3)线程调用start()方法就会立即执行么?

4)什么是并发，并发和并行的区别

四、线程的主要方法

1)void run()

线程获得cpu之后，所执行的具体任务

2)void start()

启动线程，线程调用了start()方法之后不会立即获得cpu，而是处于一个就绪（准备）状态，等待操作系统为其分配时间片，线程只有获得时间片才会进入cpu中执行任务也就是执行run()方法中代码。

3)void setName()/String getName()

设置/获取线程的名称，如果没有设置系统会自动分配

主线程：main

工作线程：thread-0,thread-1...

4)long getId()

获取id，由系统自动分配

5)static Thread currentThread()

获取当前线程实例

6)static int activeCount()

获取当前线程所属的线程组中活动的线程数

7)boolean isAlive()

判断线程是否属于存活状态，

线程启动了，但是还没有执行完任务，呢么就属于存活状态。

如果线程还没启动，或者执行完了任务，线程就是死亡状态。

8)void setPriority()/int getPriority()

设置/获取线程的优先级

线程间的切换是由操作系统来控制的，无法通过代码改变

优先级只是用于改变线程获取cpu的概率

但是优先级高的线程不一定会先获得cpu执行

线程优先级可以设置1-10级

默认优先级为5，最小优先级1，最大优先级10

9)boolean isDaemon()/void setDaemon()

判断是否为守护线程/设置一个线程为守护线程。

如果将进程中一个线程设置为守护线程（用户线程，精灵线程，后台进程）

呢么其他线程就自动变成了前台线程，

当其他所有前台线程都结束后，守护线程也就结束了，

一个进程中最多只允许有一个守护线程，

必须要在线程启动前设置

10)static void sleep()

线程休眠

调用sleep()方法的线程会进入到阻塞状态，在此期间无法获得cpu执行任务，只有等待休眠结束才会结束阻塞状态，进入到就绪状态。休眠状态的线程可以在其他线程调用interrupt()方法提前唤醒。

线程休眠的应用场景：

1)执行延迟性任务

2)执行周期性任务

11)线程插队 void join()

插入式线程

1>不带参数插入

t1线程插入到t2线程前面去执行，在此期间，t2线程处于阻塞状态，一直等到t1线程执行完任务退出cpu之后，t2线程才会结束阻塞状态回到就绪状态。

2>带参数插入

带参数的为插入时间，指定时间结束后，结束插入，在此期间被插入线程相当于阻塞状态，相当于线程休眠了。

五、线程的生命周期（状态）

1、新建状态 (NEW)

一个线程刚被创建出来，还没有调用start()方法。

2、就绪状态(准备状态，可运行状态) (READY)

就绪状态线程随时等待cpu为其分配时间片，获得时间片后，就可以进入运行状态，执行任务。

1)新建线程调用了start()方法。

2)阻塞状态的线程阻塞结束了。

3、运行状态 (RUNNING)

就绪状态的线程获得了cpu为其分配的时间片，正在操作系统中执行它的任务，也就是调用run()方法中的代码。

4、阻塞状态(等待状态) （BLOCKED）

线程状态的线程会立即退出cpu，处于等待状态，在此期间是无法获得cpu。只有阻塞结束才会进入就绪状态。

1)线程sleep()休眠

2)线程被别的线程调用join()插队

3)io阻塞

4)wait()/await()

5、死亡状态 (DEAD)

运行状态的线程在cpu中执行完了它的任务，就会被操作系统所回收。

六、线程间的同步与协作

1、同步与异步的概念

1)异步：

一个进程中如果有多个线程，那么多线程之间默认是异步并发执行的，也就是轮流切换时间来执行任务，从宏观上看是同时进行的。

2)同步：

因为多线程默认是异步并发执行的，所以如果有多个线程在对同一资源（共享变量）进行操作时，就有可能会引发线程不安全问题，所以这个时候多线程就需要有先后顺序的来执行，这个就叫线程同步，线程同步也叫串发。

2、如何将多线程之间默认的异步操作改为同步操作

java中提供了一种锁机制来支持多线程同步，保证共享变量的安全

常用的一种锁为：

2.1>synchronized同步锁

1、同步代码块

```
synchronized(同步锁对象){
        //需要进行同步的代码
}
```

同步代码块

一个线程先抢到同步锁之后，就会进入同步代码块中执行代码，同时锁死这一块代码，在此期间其他线程是无法获得同步锁，因此其他没抢到锁的线程都会处于阻塞状态，抢到锁的线程只有在在同步代码块中执行完了它所有的任务后，才会释放同步锁对象。其他处于阻塞状态的线程会进入到就绪状态，从而再来抢占已被释放的同步锁，抢到锁的线程进入同步代码块中执行任务，其他未抢到锁的线程再次进入阻塞状态，以此类推。那么，多线程就会按顺序来执行这一块代码，实现多线程同步。

2、同步方法

如果一个方法多线程只能顺次序先后调用，那么可以在方法上面添加synchronized修饰符来进行加锁，先获得锁的线程来调用方法，其他线程阻塞，调用完方法后释放锁，其他阻塞的线程变为就绪状态，在轮流获得锁调用方法

3、同步锁对象的选择

1)在Java中任意引用类型的对象都可以当作同步锁对象，只需要多线程看到的是同一把锁对象即可

2)同步方法为实例方法，锁对象默认为this

3)同步方法为静态方法，锁对象默认为当前类.class

4、代码同步之后，安全性会提高，但是效率会下降。

多线程的默认异步并发操作，效率比较高，但是会造成安全性问题。

线程安全 线程不安全

Vector ArrayList

StringBuffer StringBuilder

Hashtable HashMap

ConcurrentHashMap(线程安全的HashMap)

5、可重入锁（可重入性、递归锁)

指的是，如果一个线程在外部拿到锁之后进入到线程内部可以直接再次获得该锁，而不会因为之前的锁没有释放，处于阻塞。

可重入锁在一定程度上可以避免线程死锁的问题。

2.2、ReentrantLock可重入锁

1>ReentrantLock可重入锁实现线程同步

主要方法：

1)lock()

获取锁，上锁，获取不到会一直阻塞。

2)unlock()

释放锁。

3)tryLock()

尝试获取锁，获取到了返回true，获取不到返回false，但是不会阻塞。

4)tryLock(long timeout,TimeUnit unit)

尝试获取锁，获取到了返回true，获取不到设置一个等待时间，在指定时间内获取到锁了返回true，获取不到返回false，结束阻塞，不再等待了。

5)lockInterruptibly()

和lock()方法相同，都是获取不到锁就会一直等待，但是lockInterruptibly()的等待是可以被打断的，而lock()的等待是不会被打断的。

为了保证无论如何锁都会释放

```
try{
    lock();//上锁
    //需要同步的代码
}finally{
    unlock();//释放锁
}
```

2>公平锁和非公平锁

多线程获取锁的过程：当有多线程获取同一个锁时，只有一个线程能获取到锁，其他未获取到锁的线程会被添加到等待队列中，先就绪的线程被添加到队列的头部，等上一个获得锁的线程，执行完任务释放锁后，再从队列的头部再取出一个线程来获得锁执行任务，以此类推。

注意：线程调用start()方法之后会处于就绪状态但是先调用start()的线程不一定先就绪。

1)非公平锁

在创建ReentrantLock时不带参数或者参数为false，那么创建的就是非公平锁如果持有锁的线程在释放锁的瞬间，有非队列中的线程过来获取锁，那么该线程就能立即获得锁资源

优点：这种方式性能会比较高，因为不需要频繁的在等待队列中执行插入线程和取出线程操作但是如果一直有非队列中的线程过来，那么原来队列中的线程就一直无法获得锁资源，就会造成饥饿等待。

2)公平锁

在创建ReentrantLock时不带参数或者参数为true，那么创建的就是公平锁，严格按照FIFO先进先出的原则从等待队列中取出线程来获取锁资源执行任务，即使上一个线程在执行完任务，释放锁的瞬间，有非队列中的线程过来，该线程也不会获得锁资源，依旧会被添加到等待队列的末尾。

优点:公平锁不会造成饥饿等待，所有线程都有相同的机会获得锁资源，但是因为需要频繁的往队列中添加和取出线程，所有性能就会比较低。

3>ReentrantLock实现锁的可重入性

synchronized和ReentrantLock都是可重入锁，都具有可重入性。

4>读写锁

多线程在对同一资源进行操作时，可以对资源添加读锁和写锁两把锁。

必须保证读锁和写锁是从同一个ReentrantReadWriteLock对象来获取的

ReentrantReadWriteLock lock = new ReentrantReadWriteLock

1>获取读锁

ReentrantReadWriteLock.ReadLock readLock = lock.readLock();

2>获取写锁

ReentrantReadWriteLock.WriteLock writeLock = lock.writeLock();

读写特点:

1)多线程间获取读锁是共享的(读共享)

2)多线程间获取写锁是互斥的（写互斥）

3)多线程间获取读锁和写锁是互斥的（读写互斥）

3、线程间协作（线程等待)

1>线程协作的概念

线程在同步了的基础之上，如果在操作同一资源时，必须按照一个排好的顺序来操作这个资源，这个就是线程间协作。

2>线程间协作方式

2.1>使用synchronized同步锁实现

借助Object类中的wait()/notify()/notifyAll()三个方法来实现的

1>wait()

调用该方法的线程会释放已经获取的锁，并退出cpu处于阻塞状态，必须等待其他具备相同锁的线程进行通知才能唤醒，唤醒后进入到就绪状态，从而重新尝试获取锁，获得了锁之后

继续往下执行，如果没有获得锁会继续进入到阻塞状态重新等待其他线程通知，所以wait()一般是卸载同步代码块最前面。

2>notify()

通知其他具备相同锁的线程不要再等待了，让其结束阻塞状态，从而进入就绪状态，但是该方法并不会释放锁，只能等同步代码块执行完成后才会释放锁，所以notify()一般都是卸载同步代码块的最后。

3>notifyAll()

功能和notigy()相同，用于唤醒多个处于阻塞等待的线程。

注意：

1)这三个方法必须在同步代码块或同步方法内部调用

2)这三个方法必须由同一个同步锁对象来进行调用

2.2>使用ReentrantLock可重入锁实现

通过ReentrantLock获取一个Condition对象，再借助Condition对象中await()/signal()/signalAll()三个方法来实现线程间协作。

1)await() 相当于Object类中的wait()方法

2)signal() 相当于Object类中的notift()

3)signalAll() 相当于Object类中的notifyAll()

注意：

1)这三个方法必须由同一个Condition对象来进行调用

2)必须在lock()和unlock()之间来进行调用

4、synchronized和ReentrantLock区别

1)都可以实现线程间同步，都是可重入锁，都是悲观锁

2)synchronized是java中一个关键字，上锁和解锁过程都是自动完成的，使用方便，Reentranlock是并发包下一个类，上锁和解锁过程需要手动控制，使用灵活。

七、线程池（ThreadPool）

1、什么是线程池

用于创建和管理多个线程的容器对象，当我们的程序需要线程执行任务时，可以直接从池中获取，当任务执行完成后，再将线程返还到池中，让线程得到复用，这样做的好处时，避免频繁的创建和销毁线程，节省系统开销。

2、线程池的创建

```
ThreadPoolExecutor pool =
        new ThreadPoolExecutor(3, 5, 5, TimeUnit.SECONDS,
        new ArrayBlockingQueue<>(20), r -> new Thread(r),
        new ThreadPoolExecutor.AbortPolicy());
```

参数解释

1)corePoolSize : 核心线程数

线程池创建的时候就需要创建的线程数，也是线程池中需要保证的最小数量。

2)int maximumPoolSize : 最大线程数

线程池中允许存在的最大线程数

3)long keepAliveTime : 保持活动时间 （超时时间、活跃时间、空闲时间）

非核心线程在指定时间内没有接到新的任务，那么就会被销毁，如果调用了allowCoreThreadTimeOut()方法，并且参数值设为true，那么在超过指定时间后，核心线程也会被销毁。

4)TimeUnit unit : 超时的时间单位

5)BlockingQueue<Runnable> workQueue : 任务队列

在线程池的编程模式下，所有需要执行的任务都会被提交到一个任务队列中然后从队列中按照FIFO原则取出一个任务，再从线程池中取出一个空闲线程来执行这个任务。

任务执行完成后，线程返还到池中，队列是一个阻塞式队列

6)ThreadFactory threadFactory : 线程工厂

用于创建线程池中的所有线程

7)RejectedExecutionHandler handler : 被拒接的执行处理器（线程池的拒接策略）

当线程池中所有线程都在执行任务，并且任务队列也是满负荷的情况下，

又有新的任务提交给线程池需要执行时，就会触发拒接策略。

拒接策略

1>AbortPolicy

拒接执行新的任务，并且抛出异常

2>DiscardPolicy

丢弃这个线程，拒接执行新的任务，但是不会抛异常，DoNothing

3>DiscardOldextPolicy

如果线程池没有关闭，将任务队列中存在最久的任务（队首元素）移除掉，然后将新的任务加入到队列中，如果线程池已经关闭了，则直接丢弃掉该任务不执行。

4>CallerRunsPolicy

如果线程池没有关闭，它不会使用线程池中线程执行任务，也不会加入到任务队列，而是直接再当前线程直接调用run()方法来执行任务，如果线程池已经关闭了，则直接丢弃掉该任务不执行。

主要方法:

1)allowCoreThreadTimeOut()

超过指定的超时时间后,非核心线程也会被销毁。

2)execute(Runnable task)

将任务提交给线程池线程池会自动将任务放入到队列中,并取出空闲线程来执行任务。

3)shutdown()

安全关闭,必须要等到线程池中所线程把所有任务都执行完成后再关闭线程池

4)shutdownNow()

立即关闭,即使还有未执行完的任务也会立即关闭线程池。

3、Java自带线程池

1)单线程的线程池

ExecutorService es = Executors.newSingleThreadExecutor();

2)固定数量线程的线程池

ExecutorService es = Executors.newFixedThreadPool(5);

3)可缓存线程池

ExecutorService es = Executors.newCachedThreadPool();

4)周期性线程池

ScheduledExecutorService es = Executors.newScheduledThreadPool(int n);

4.1)schedule()

执行延迟任务

4.2)scheduleAtFixedRate()

执行周期性任务

4.3)scheduleWithFixedDely()

执行周期性加延迟任务

进程 线程

公司领导 公司员工

虚拟化：就是虚拟内存