本站概述

1. 程序 进程 线程
2. java实现多线程
3. 线程的状态和方法
4. 线程的基本信息和优先级
5. 线程的同步和死锁问题
6. 生产者消费者者模式
7. 任务调度

一、 程序 进程 线程

1、程序：Program指令集，是一个静态概念

2、进程：Process，操作系统调度程序的动态概念

1）进程是程序的一次动态执行过程，占用特定的地址空间

2）每个进程都是独立的，有3部分组成cpu data code

3）缺点：内存的浪费，cpu的负担

3、线程：在进程内多条执行路径

Thread，是进程中一个单一的连续控制流程/执行路径 a single sequential flow of control

1. 线程又被称为轻量级进程 lightweight process
2. Threads run at the same time, independently of one another
3. 一个进程可拥有多个并行（concurrent）的线程
4. 一个进程中的线程共享相同的内存单元/内存地址空间，可以访问相同的变量和对象，而且它们从同一堆中分配对象，通信、数据交换、同步操作
5. 由于线程间的通信是在同一地址空间上进行的，所以不需要额外的通信机制，这就使得通信更简便而且信息传递的速度也更快

线程和进程的区别:

|  |  |  |
| --- | --- | --- |
| 区别 | 进程 | 线程 |
| 根本区别 | 作为资源分配的单位 | 调度和执行的单位 |
| 开销 | 每个进程都有独立的代码和数据空间（进程上下文），进程间切换会有较大的开销 | 线程可以看成是轻量级的进程，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器，线程切换的开销小 |
| 所处环境 | 在操作系统中能同时运行多个任务（程序） | 在同一程序中有多个顺序流同时执行 |
| 分配内存 | 系统在运行的时候会为每个进程分配不同的内存区域 | 除了CPU之外，不会为线程分配内存（线程所使用的资源是它所属的进程的资源），线程组只能共享资源 |
| 包含关系 | 没有线程的进程可以被看作单线程的，如果一个进程内拥有多个线程，则执行过程不是一条线，而是多条线程共同完成的 | 线程是进程的一部分，所以线程有的时候被称作轻权进程或轻量级进程 |

继承Thread类方式的缺点：

如果我们的类已经从一个类继承（如小程序必须继承Applet类），则无法再继承Thread类

通过Runnable接口实现多线程

优点：可以同时实现继承，实现Runnable接口方式要通用一些

1. 避免单继承
2. 方便共享资源，同一份资源，多个代理访问

通过Callable 接口实现多线程

优点：可以获取返回值

Callable和Future接口

Callable是类似于Runable的接口，实现Callable接口的类和实现Runable的类都是可被其它线程执行的任务

Callable和Runable有几点不同：

1. Callable规定的方法使call()；而Runable规定的方法使run()；
2. call()方法可抛出异常，而run()方法是不能抛出异常的。
3. Callable的任务执行后可返回值，运行Callable任务可拿到一个Future对象，而Runable接口的任务是不能返回值的

Future表示异步计算的结果，它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。

通过Future对象可了解任务执行情况，可取消任务执行，还可获取任务执行的结果

缺点：繁琐

思路：

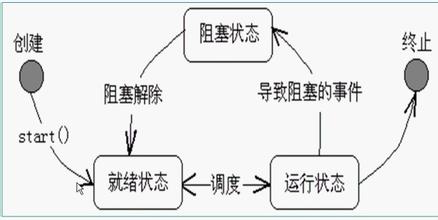
1. 创建Callable实现类+重写call()
2. 借助执行调度服务ExecutorService，获取Future对象

ExecutorService service = Executors.newFixedThreadPool(2);

Future<Integer> future = service.submit(race);

1. 获取返回值future.get()
2. 停止服务service.shutdownNow();

线程状态



1、新生状态：

用new关键字和Thread类或其子类建立以一个线程后，该线程对象就处于新生状态。处于新生状态的线程有自己的内存空间，通过调用start()方法进入就绪状态。

2、就绪状态：

处于就绪状态的线程已经具备了运行条件，但还没有分配到CPU，处于线程就绪队列，等待系统为其分配CPU，等待状态并不是执行状态，当系统选定一个等待执行的Thread对象以后，它就会从等待执行状态进入执行状态，系统挑选的动作称之为CPU调度。一旦获得CPU，线程就进入运行状态并自动调用自己的run()方法。

3、运行状态：

在运行状态的线程执行自己的run方法中的代码，直到调用其它方法而终止、或等待某资源而阻塞或完成任务而死亡。如果在给定的时间片内没有执行结束，就会被系统给换下来会带等待执行状态。

4、阻塞状态：

处于运行状态的线程在某些情况下，如执行了sleep（睡觉）方法，或等待IO设备等资源，将让出CPU并暂时停止自己的运行，进入阻塞状态。在阻塞状态下的线程不能进入就绪队列。只有当引起阻塞的原因消除时，如睡眠时间已到，或等待的IO设备空闲下来，线程便进入就绪状态，重新到就绪队列中排队等待，被系统选中后从原来停止的位置开始继续运行。

5、死亡状态

死亡状态是线程生命周期中的最后一个阶段。线程死亡的原因有两个。一个是正常运行的线程完成它的全部工作；另一个是线程被强制性终止。如通过执行stop或destroy方法来终止一个线程【不推荐使用这两个方法。前者会产生异常，后者是强制终止，不会释放锁。】

终止线程

1. 自然终止：线程体正常执行完毕
2. 外部干涉：
3. 线程类中定义线程体使用的标识
4. 线程体使用该标识
5. 提供对外的方法，改变该标识
6. 外部根据条件调用该方法即可
7. 阻塞
8. join 合并线程
9. yied 暂停线程 static
10. sleep 休眠，不释放锁
11. 与时间相关：倒计时
12. 模拟网络延时

线程基本信息

|  |  |
| --- | --- |
| 方法 | 功能 |
| isAlive() | 判断线程是否还活着，即线程是否还未终止 |
| getPriority() | 获得线程的优先级数值 |
| setPriority() | 设置线程的优先级数值 |
| getName() | 取得线程的名字 |
| setName() | 给线程一个名字 |
| currentThread() | 取得当前线程正在运行的线程对象，也就是取得自己本身 |

线程的优先级:

MIN\_PRIORITY 一个线程能拥有的最小优先级

NORM\_PRIORITY 一个线程被赋予的默认优先级

MAX\_PRIORITY 一个线程能拥有的最大优先级

priority的数值并不代表谁会先执行，而是代表被执行的概率。

同步：并发，多个线程访问同一份资源，确保资源安全 🡪线程安全

synchronize 🡪同步

1. 同步块

synchronized(引用类型|this|类.class){

}

1. 同步方法

synchronized

1. 死锁：过多的同步容易造成死锁

生产者消费者模式

生产者消费者问题（Producer-Consumer problem），也成为有限缓冲问题（Bounded buffer problem），是一个多线程同步问题的经典案例。该问题描述了两个共享固定大小缓冲区的线程—即所谓的“生产者”和“消费者”—在实际运行时会发生的问题。

生产者的主要作用是生成一定数量的数据放到缓冲区内，然后重读此过程。与此同时，消费者也在缓冲区小号这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会再缓冲区空时小号数据。

要解决该问题，就必须让生产者在缓冲区满时休眠（要么干脆放弃数据），等到下次消费者消耗缓冲区中的数据的时候，生产者才能被唤醒，开始往缓冲区添加数据。同样，可以让消费者在缓冲区空时进入休眠，等到生产者网缓冲区添加数据之后，再唤醒消费者。通常常用的方法有信号灯法、管程等。如果解决方法不够完善，则容易出现死锁。出现死锁时，两个线程都会出现死锁，等待对方唤醒自己。

wait():等待，释放锁 sleep() 不释放锁

notify()/notifyAll()：唤醒

生产者消费者模式信号灯法与synchronized一起使用

任务调度

了解

Timer定时器类

TimerTask任务类

通过TimerTask（Spring的任务调度就是通过他们来实现的）

在种种实现方式中，Timer类实现的是类似闹钟的功能，也就是定时或者每隔一定时间触发一次线程。其实，Timer类本身实现的就是一个线程，只是这个线程是用来实现调用其它线程的。而TimerTask类是一个抽象类，该类实现了Runable接口，所以按照前面的介绍，该类具备多线程的能力

在这种实现方式中，通过集成TimerTask，使该类活得多线程能力，将需要多线程执行的代码书写在run()方法内部，然后通过Timer类启动线程的执行。

在实际使用时，一个Timer类可以启动任意多个TimerTask实现的线程，但是多个线程之间存在阻塞。所以如果多个线程之间，如果需要完全独立运行的话，最好还是一个Timer启动一个TimerTask类。

schedule(TimerTask task,DateTime time);

schedule(TimerTask task,DateTime time,long period);

自学quartz