# 多线程

## 线程的状态

## C:\Users\Administrator\Pictures\1689841-383f7101e6588094.png

1.调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。

2.调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool )，释放同步锁使线程回到可运行状态（Runnable）

3.对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool ),同步锁被释放进入可运行状态(Runnable)。

## 实现方法

1. 继承Thread类。
2. 实现Runable接口
3. 实现ThreadFactory接口

工厂模式方法创建线程：

1、属性：count（线程对象的数量），name（名称），stats（统计数据？），

2、实现newThread方法

3、创建Task类并实现Runable接口

4、通过1,2创建的MyThreadFactory启动Task。

eg：

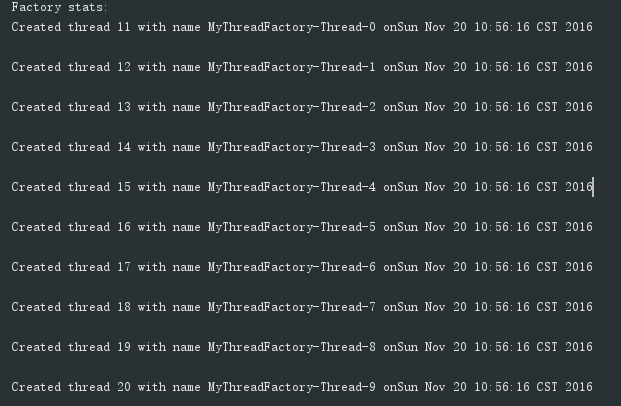
Task类

public class Task implements Runnable{  
 @Override  
 public void run() {  
 try{  
 TimeUnit.***SECONDS***.sleep(1);  
 }  
 catch (InterruptedException e){  
 e.printStackTrace();;  
 }  
 }  
}

MyThreadFactory类

public class MyThreadFactory implements ThreadFactory {  
  
 private int counter;  
 private String name;  
 private List<String> stats;  
  
 public MyThreadFactory(String name) {  
 counter = 0;  
 this.name = name;  
 stats = new ArrayList<String>();  
 }  
  
 @Override  
 public Thread newThread(Runnable run) {  
 Thread t = new Thread(run, name + "-Thread-" + counter);  
 counter++;  
 stats.add(String.format("Created thread %d with name %s on%s\n", t.getId(), t.getName(), new Date()));  
 return t;  
 }  
  
 public String getStas() {  
 StringBuffer buffer = new StringBuffer();  
 Iterator<String> it = stats.iterator();  
 while (it.hasNext()) {  
 buffer.append(it.next());  
 buffer.append("\n");  
 }  
 return buffer.toString();  
 }  
  
 public static void main(String[] args) {  
 MyThreadFactory factory = new MyThreadFactory("MyThreadFactory");  
 Task task = new Task();  
 Thread thread = null;  
 for (int i = 0; i < 10; i++) {  
 thread = factory.newThread(task);  
 thread.start();  
 }  
 System.***out***.printf("Factory stats:\n");  
 System.***out***.printf("%s\n", factory.getStas());  
 }  
}

结果：



## 线程同步

1. 使用synchronized
2. 在同步代码块使用条件

生产者，消费者实例

当缓冲区为空时，消费者不能读取；当缓冲区满的时候，生产者不能读取。

1. 使用锁

允许多个读取操作，但是只要一个写操作。比synchronized性能更好。

### 并发集合

（只是了解了下并发集合）

线程安全的集合。

1、并发List

Vector和CopyOnWriteArrayList是两个线程安全的List，Vector读写操作都用了同步，相对来说更适用于写多读少的场合，CopyOnWriteArrayList在写的时候会复制一个副本，对副本写，写完用副本替换原值，读的时候不需要同步，适用于写少读多的场合。

2、并发Set

CopyOnWriteArraySet基于CopyOnWriteArrayList来实现的，只是在不允许存在重复的对象这个特性上遍历处理了一下。

3、并发Map

ConcurrentHashMap是专用于高并发的Map实现，内部实现进行了锁分离，get操作是无锁的。

4、并发的Queue

在并发队列上JDK提供了两套实现，一个是以ConcurrentLinkedQueue为代表的高性能队列，一个是以BlockingQueue接口为代表的阻塞队列。ConcurrentLinkedQueue适用于高并发场景下的队列，通过无锁的方式实现，通常ConcurrentLinkedQueue的性能要优于BlockingQueue。BlockingQueue的典型应用场景是生产者-消费者模式中，如果生产快于消费，生产队列装满时会阻塞，等待消费。

5、并发的Dueue

Queue是一种双端队列，它允许在队列的头部和尾部进行出队和入队的操作。Dueue实现类有非线程安全的LinkedList、ArrayDueue和线程安全的LinkedBlockingDueue。LinkedBlockingDueue没有进行读写锁的分离，因此同一时间只能有一个线程对其操作，因此在高并发应用中，它的性能要远远低于LinkedBlockingQueue，更低于ConcurrentLinkedQueue。

6、并发锁重入锁ReentrantLock

ReentrantLock是一种互斥锁的实现，就是一次最多只能一个线程拿到锁；

7、读写锁ReadWriteLock

读写锁有读取和写入两种锁，读取锁允许多个读取的线程同时持有，而写入锁只能有一个线程持有。

8、条件Condition

调用Condition对象的相关方法，可以方便的挂起和唤醒线程。