Java 核心技术进阶

多线程和并发编程

简介

- 当前的操作系统都是多任务OS
- 每个独立执行的任务就是一个进程
- OS将时间划分为多个时间片（时间很短）
- 每个时间片内将CPU分配给某一个任务，时间片结束，CPU将自动回收，再分配给另外任务。从外部看，所有任务是同时在执行。但是在CPU上，任务是按照串行依次运行（单核CPU）。如果是多核，多个进程任务可以并行。但是单个核上，多进程只能串行执行。

- 多进程的优点
  - 可以同时运行多个任务
  - 程序因IO堵塞时，可以释放CPU，让CPU为其他程序服务
  - 当系统有多个CPU时，可以为多个程序同时服务
    - 我们的CPU不再提高频率，而是提高核数
    - 2005年Herb Sutter的文章 The free lunch is over，指明多核和并行程序才是提高程序性能的唯一办法
- 多进程的缺点
  - 太笨重，不好管理
  - 太笨重，不好切换

多线程概念

- 一个程序可以包括多个子任务，可串/并行
- 每个子任务可以称为一个线程
- 如果一个子任务阻塞，程序可以将CPU调度另外一个子任务进行工作。这样CPU还是保留在本程序中，而不是被调度到别的程序(进程)去。这样，提高本程序所获得CPU时间和利用率。

- 多进程 vs 多线程
  - 线程共享数据
  - 线程通讯更高效
  - 线程更轻量级，更容易切换
  - 多个线程更容易管理

多线程的实现

Java 多线程两种方式：

- java.lang.Thread
  - 线程继承Thread类，实现run方法
- java.lang.Runnable接口
  - 线程实现Runnable接口，实现run方法

```
public class Thread1 extends Thread{
    public void run()
    {
        System.out.println("hello");
    }
}
```

```
public class Thread2 implements Runnable{
    public void run()
    {
        System.out.println("hello");
    }
}
```

- 启动
  - start方法，会自动以新进程调用run方法
  - 直接调用run方法，将变成串行执行
  - 同一个线程，多次start会报错，只执行第一次start方法
  - 多个线程启动，其启动的先后顺序是随机的
  - 线程无需关闭，只要其run方法执行结束后，自动关闭
  - main函数(线程)可能早于新线程结束，整个程序并不终止
  - 整个程序终止是等所有的线程都终止(包括main函数线程)

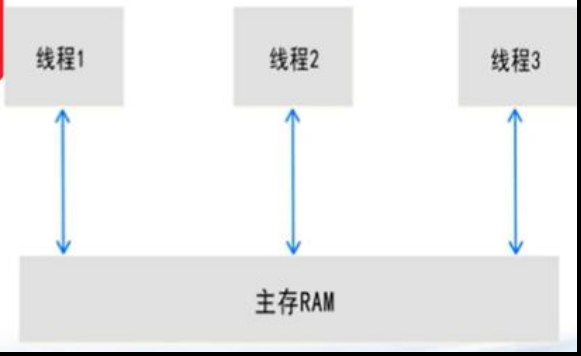
1. 实现Runnable的对象必须包装在Thread类里面，才可以启动；
2. 不能直接对Runnable的对象进行start方法。

- Thread vs Runnable
  - Thread占据了父类的名额，不如Runnable方便
  - Thread 类实现Runnable
  - Runnable启动时需要Thread类的支持
  - Runnable更容易实现多线程中资源共享
- 结论：建议实现Runnable接口来完成多线程

多线程信息共享

- 线程类
  - 通过继承Thread或实现Runnable
  - 通过start方法，调用run方法， run方法工作
  - 线程run结束后，线程退出
- 粗粒度：子线程与子线程之间、和main线程之间缺乏交流
- 细粒度：线程之间有信息交流通讯
  - 通过共享变量达到信息共享
  - JDK原生库暂不支持发送消息(类似MPI并行库直接发送消息)

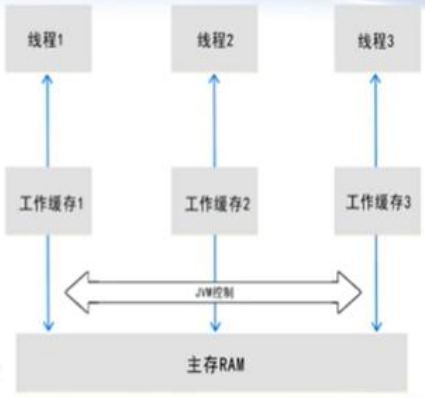
- 通过共享变量在多个线程中共享消息
  - static变量
  - 同一个Runnable类的成员变量
- 查看示例



1. Static 变量
2. Runnable 的类成员变量

```
12 class TestThread0 extends Thread
13 {
14     private int tickets=100;            //每个线程卖100张，没有共享
15     //private static int tickets=100;  //static变量是共享的，所有的线程共享
16     public void run()
```

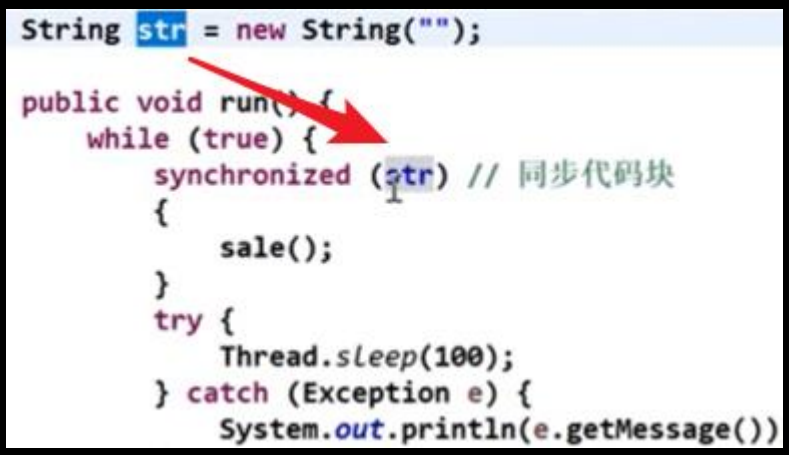
- 多线程信息共享问题
  - 工作缓存副本 自己的栈
  - 关键步骤缺乏加锁限制
- i++，并非原子性操作
  - 读取主存i(正本)到工作缓存(副本)中
  - 每个CPU执行(副本)i+1操作
  - CPU将结果写入到缓存(副本)中
  - 数据从工作缓存(副本)刷到主存(正本)中

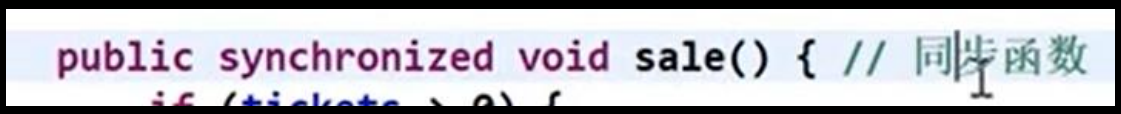


- 变量副本问题的解决方法
  - 采用volatile 关键字修饰变量
  - 保证不同线程对共享变量操作时的可见性

Volatile 关键字：变量在“工作缓存”中被修改时，其他程序能够察觉到。

```
3 class TestThread2 extends Thread
4 {
5     //boolean flag = true;    //子线程不会停止
6     volatile boolean flag = true;  //用volatile修饰的变量可以及时在各线程里面通知
7     public void run()
8     {
9         int i=0;
0         while(flag)
1         {
2             i++;
3         }
4         System.out.println("test thread3 is exiting");
5     }
```





代码块和函数都可以修饰

总结：

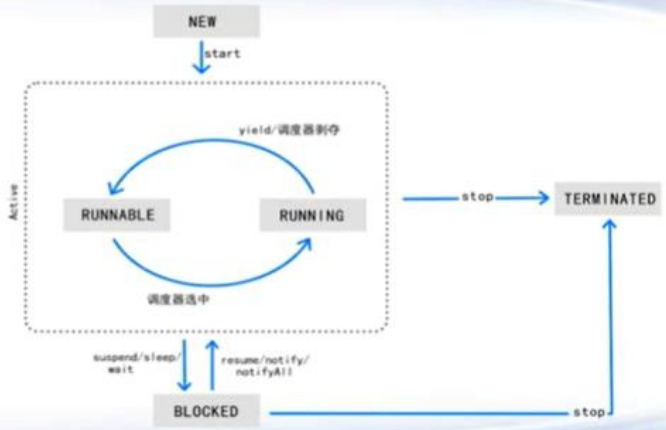
1. Volatile：工作缓存

2. Synchronized:互斥锁

多线程管理

线程状态

- NEW 刚创建(new)
- RUNNABLE 就绪态(start)
- RUNNING 运行中(run)
- BLOCK 阻塞(sleep)
- TERMINATED 结束



- Thread的部分API已经废弃
  - 暂停和恢复suspend/resume
  - 消亡 stop/destroy
- 线程阻塞/和唤醒
  - sleep，时间一到，自己会醒来
  - wait/notify/notifyAll，等待，需要别人来唤醒
  - join，等待另外一个线程结束
  - interrupt，向另外一个线程发送中断信号，该线程收到信号，会触发InterruptedException(可解除阻塞)，并进行下一步处理

- 线程被动地暂停和终止
  - 依靠别的线程来拯救自己 ☹☹☹
  - 没有及时释放资源
- 线程主动暂停和终止
  - 定期监测共享变量
  - 如果需要暂停或者终止，先释放资源，再主动动作☺☺☺
  - 暂停：Thread.sleep()，休眠
  - 终止：run方法结束，线程终止

Thread.sleep:自己睡眠

- 多线程死锁
  - 每个线程互相持有别人需要的锁(哲学家吃面问题)
  - 预防死锁，对资源进行等级排序
- 守护(后台)线程
  - 普通线程的结束，是run方法运行结束
  - 守护线程的结束，是run方法运行结束，或main函
  - 守护线程永远不要访问资源，如文件或数据库等
- 线程查看工具 jvisualvm

死锁：所有线程都在期待某种资源，但是这种资源却都不可用。

比如：A 需要 a， a 在 B 那里；B 需要 b，b 在 A 那里。于是 AB 相互等待。

死锁防止：对资源进行等级排序

守护线程：

```
TestThread4 t = new TestThread4();
t.setDaemon(true);
t.start();
Thread.sleep(2000);
System.out.println("main thread is exiting");
```

并发框架 Executor

- 业务：任务多，数据量大
- 串行 vs 并行
  - 串行编程简单，并行编程困难
  - 单个计算核频率下降，计算核数增多，整体性能变高
- 并行困难(任务分配和执行过程高度耦合)
  - 如何控制粒度，切割任务
  - 如何分配任务给线程，监督线程执行过程

- 并行模式
  - 主从模式 (Master-Slave)
  - Worker模式(Worker-Worker)
- Java并发编程
  - Thread/Runnable/Thread组管理
  - Executor(本节重点)
  - Fork-Join框架

- 线程组ThreadGroup
  - 线程的集合
  - 树形结构，大线程组可以包括小线程组
  - 可以通过enumerate方法遍历组内的线程，执行操作
  - 能够有效管理多个线程，但是管理效率低
  - 任务分配和执行过程高度耦合
  - 重复创建线程、关闭线程操作，无法重用线程

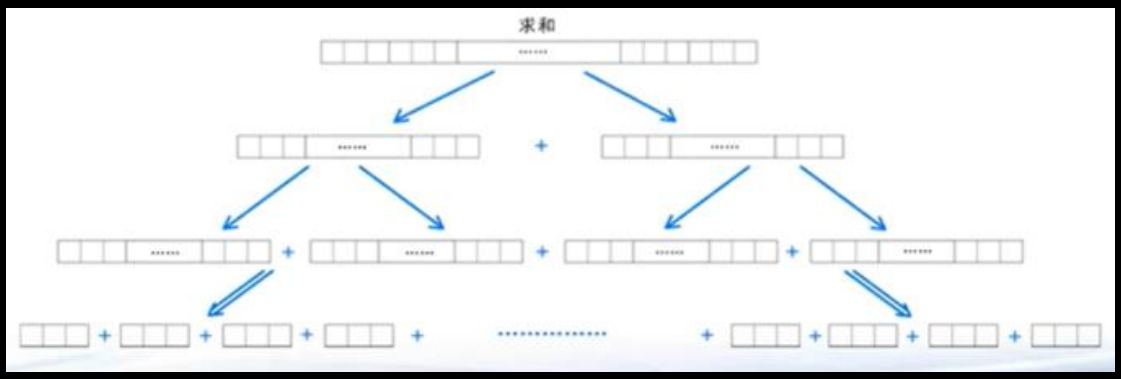
线程组


```
public class SumTask implements Callable<Integer> {
    //定义每个线程计算的区间
```

实现 Callable 也可以构建多线程

Executor 框架

并行框架 Fork-join



- 关键类
  - ForkJoinPool 任务池
  - RecursiveAction
  - RecursiveTask

2021/4/14