# Java并发编程实战学习笔记

## Maven jar包管理小知识：

### 1.1 Jar包的作用域

*<!--日志相关jar包-->*<**dependency**>  
 <**groupId**>org.slf4j</**groupId**>  
 <**artifactId**>slf4j-log4j12</**artifactId**>  
 <**version**>1.7.7</**version**>  
 *<!--maven项目的jar包管理，这作用域对于jar包的使用是有影响的  
 <比如这里，写着<scope>test</scope>,表示改jar包只在测试运行时期生效，这也一来，咱们写的代码编译器就无法使用到、  
 jar包中的资源了，这样是会有问题的，编译都过不去，程序就会报错,这时可以改变作用域，可以直接不写，去掉<scope>标签  
 这样会使用默认的作用域，而默认的作用域就是编译compile  
 -->* <**scope**>compile</**scope**>  
</**dependency**>

### 1.2 Jar包版本锁定

<**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 <**maven.compiler.source**>1.8</**maven.compiler.source**>  
 <**maven.compiler.target**>1.8</**maven.compiler.target**>  
 <**spring.version**>5.0.2.RELEASE</**spring.version**>  
</**properties**>

如上：如果需要锁定spring相关jar包的版本，可以在<properties标签中使用

<spring.version标签来锁定spring jar包的版本

然后再导入spring相关jar包的时候，通过el表达式，来指定spring相关jar包的版本

如下：

<**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-webmvc</**artifactId**>  
 <**version**>${spring.version}</**version**>  
</**dependency**>

## 进程/线程

### 2.1 进程

程序由指令和数据组成，但这些指令要运行，数据要读写，就必须将指令加载至CPU，数据加载至内存。在指令加载过程中还需要用到磁盘，网络等设备。进程就是用来加载指令、管理内存、管理IO的

当一个程序被运行，从磁盘加载这个程序的代码至内存，这时就开启了一个进程

进程就可以视为程序的一个实例，大部分程序同时可以运行多个实例进程（例如记事本，画图，浏览器等。）也有的程序只能启动一个实例进程（例如网易云音乐，360安全卫士）

### 2.2 线程

一个进程之内可以分为一到多个线程

一个线程就是一个指令流，将指令流中的一条条指令以一定的顺序交给CPU执行

Java中，线程作为最小调度单位，进程作为资源分配的最小单位，在windows中进程是不活动的，只是作为线程的容器

### 2.3 二者对比

进程基本上独立的，而线程存在进程内，是进程的一个子集

进程拥有共享的资源，如内存空间等，供其内部的线程共享，

进程间通信较为复杂：

同一台计算机的进程通信称为IPC

不同计算机之间的进程通信，需要通过网路，并遵守共同的协议，例如HTTP协议

线程通信相对简单，因为它们共享进程的内存，一个例子是多个线程可以反问同一个共享的的变量

线程更轻量，线程上下文切换成本一般要比进程上下文切换低

## 上下文切换：

### 3.1 概述

上下文切换指的是[内核](https://baike.baidu.com/item/%E5%86%85%E6%A0%B8/108410)（操作系统的核心）在[CPU](https://baike.baidu.com/item/CPU/120556)上对进程或者线程进行切换。上下文切换过程中的信息被保存在进程控制块（PCB-Process Control Block）中。PCB又被称作切换桢（SwitchFrame）。上下文切换的信息会一直被保存在CPU的内存中，直到被再次使用。

上下文切换 (context switch) , 其实际含义是任务切换, 或者CPU寄存器切换。当多任务内核决定运行另外的任务时, 它保存正在运行任务的当前状态, 也就是CPU寄存器中的全部内容。这些内容被保存在任务自己的堆栈中, 入栈工作完成后就把下一个将要运行的任务的当前状况从该任务的栈中重新装入CPU寄存器, 并开始下一个任务的运行, 这一过程就是context switch。

每个任务都是整个应用的一部分, 都被赋予一定的优先级, 有自己的一套CPU寄存器和栈空间, 如图所示

### 3.2 原理

上下文切换的基本原理就是当发生任务切换时, 保存当前任务的寄存器到内存中, 将下一个即将要切换过来的任务的寄存器状态恢复到当前CPU寄存器中, 使其继续执行, 同一时刻只允许一个任务独享寄存器。在任务切换的过程中是涉及任务上下文的保存和恢复操作, 而任务上下文切换操作的性能是衡量操作系统性能的一个重要指标。任务上下文切换指标可以反映出操作系统在多任务环境下的处理能力。

### 3.3 流程

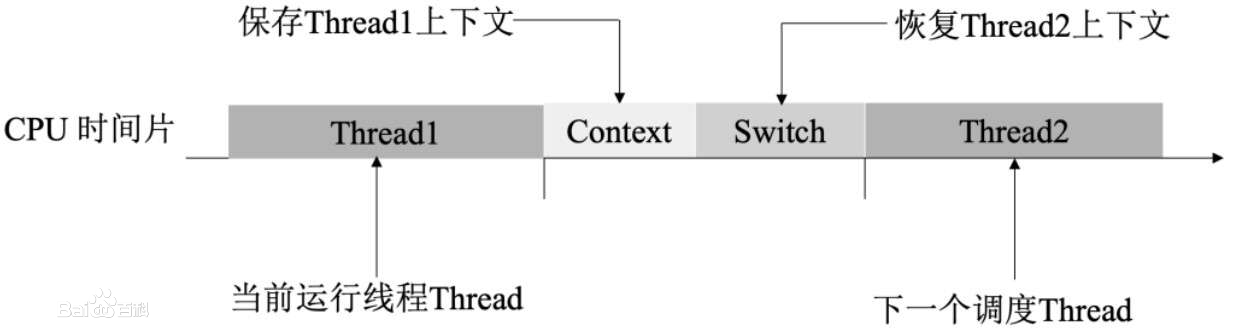
上下文的切换流程如下 [1] 

（1）挂起一个进程，将这个进程在CPU中的状态（上下文信息）存储于内存的PCB中。 [1]

（2）在PCB中检索下一个进程的上下文并将其在CPU的寄存器中恢复。 [1]

（3）跳转到程序计数器所指向的位置（即跳转到进程被中断时的代码行）并恢复该进程。 [1]

时间片轮转方式使多个任务在同一CPU上的执行有了可能，具体过程如图所示。 [1]



### 3.4 引起上下文切换的原因

引起线程上下文切换的原因如下 [1]

（1）当前正在执行的任务完成，系统的CPU正常调度下一个任务。

（2）当前正在执行的任务遇到I/O等阻塞操作，调度器挂起此任务，继续调度下一个任务。

（3）多个任务并发抢占锁资源，当前任务没有抢到锁资源，被调度器挂起，继续调度下一个任务。

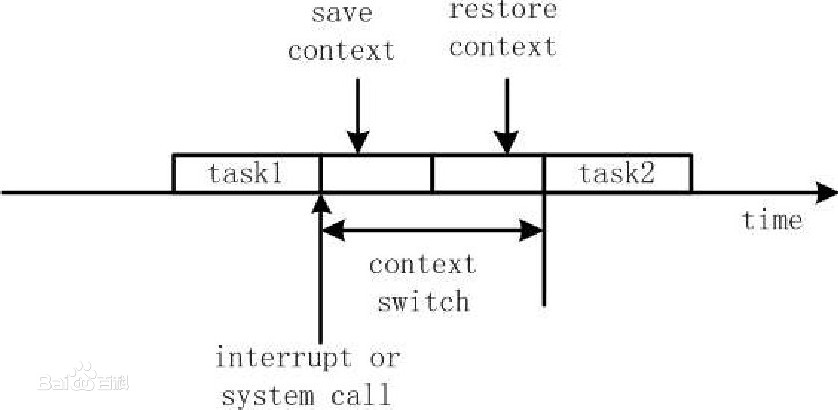
（4）用户的代码挂起当前任务，比如线程执行sleep方法，让出CPU。

（5）硬件中断。

### 3.5 上下文切换开销

上下文切换是[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/192)内核优化的一个关键参数指标。在任务间发生切换需要花费大量的时间用于处理诸如:保存和恢复[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8/187682)和内存页表、更新内核相关数据结构等操作。上下文切换通常是计算密集型的。也就是说, 它需要相当可观的处理器时间, 在每秒几十上百次的切换中, 每次切换都需要纳秒量级的时间。所以, 上下文切换对系统来说意味着消耗大量的CPU时间。 [4]

从Linux内核内部实现来看, 如图所示, 上下文切换所花费的延迟时间是从调度器选好要调度的任务 (任务1) 后到把任务上下文切换到另一个任务 (任务2) 所花费的时间。即context\_switch () 函数的开销



### 3.6 性能影响

上下文切换会对性能造成负面影响。一些上下文切换相对其他切换而言更加昂贵；其中一个更昂贵的上下文切换是跨核上下文切换（Cross-Core Context Switch）。一个线程可以运行在一个专用处理器上，也可以跨处理器。由单个处理器服务的线程都有处理器关联（Processor Affinity），这样会更加有效。在另一个处理器内核抢占和调度线程会引起缓存丢失，作为缓存丢失和过度上下文切换的结果要访问本地内存。总之，这称为“跨核上下文切换”

## 4 并行并发概念

并行是指两个或者多个事件在**同一时刻**发生；而并发是指两个或多个事件在同一时间间隔发生

并行是在不同实体上的多个事件，并发是在同一实体上的多个事件

在一台处理器上“同时”（这个同时实际上市交替“”）处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

### 4.1 普通解释：

#### 4.1.1并发：Concurrent

交替做不同事情的能力（同一时间应对（dealing with）多件事情的能力）

#### 4.1.2 并行：parallel

同时做不同事情的能力（同一时间动手做（doing）多件事情的能力）

### 4.2 专业术语：

#### 4.2.1 并发：

不同的代码块交替执行

#### 4.2.2 并行：

不同的代码块同时执行

### 4.3 总结

如果某个系统支持两个或者多个动作（Action）同时存在，那么这个系统就是一个并发系统。如果某个系统支持两个或者多个动作同时执行，那么这个系统就是一个并行系统。

　　并发系统与并行系统这两个定义之间的关键差异在于“存在”这个词。

　　在并发程序中可以同时拥有两个或者多个线程。这意味着，如果程序在单核处理器上运行，那么这两个线程将交替地换入或者换出内存。这些线程是同时“存在”的——每个线程都处于执行过程中的某个状态。

　　如果程序能够并行执行，那么就一定是运行在多核处理器上。此时，程序中的每个线程都将分配到一个独立的处理器核上，因此可以同时运行。

　　我相信你已经能够得出结论——“并行”概念是“并发”概念的一个子集。也就是说，你可以编写一个拥有多个线程或者进程的并发程序，但如果没有多核处理器来执行这个程序，那么就不能以并行方式来运行代码。因此，凡是在求解单个问题时涉及多个执行流程的编程模式或者执行行为，都属于并发编程的范畴。

**并发就是指代码逻辑上可以并行，有并行的潜力，但是不一定当前是真的以物理并行的方式运行。并发指的是代码的性质，并行指的是物理运行状态。**

顾名思义，并发强调的是一起出发，并行强调的是一起执行。**并发的反义是顺序，并行的反义是串行。并发并行并不是互斥概念，只不过并发强调任务的抽象调度，并行强调任务的实际执行。**

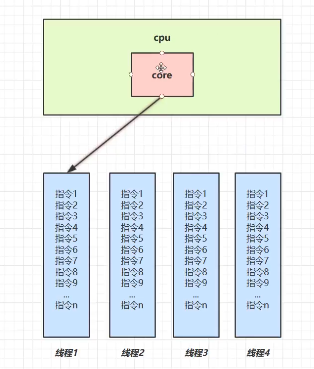


图. 并发Concurrent （单核CPU）

单核cpu下，线程实际上还是串行执行的，操作系统中有一个组件叫做任务调度器，将cpu的时间片（windows下时间片最小约为15毫秒）分给不同的线程使用，只是由于cpu在线程间（时间片很短）的切换非常快，人类感觉是同时运行的，总结一句话就是：微光串行，宏观并行，一般会将这种线程轮流使用cpu的做法成为并发





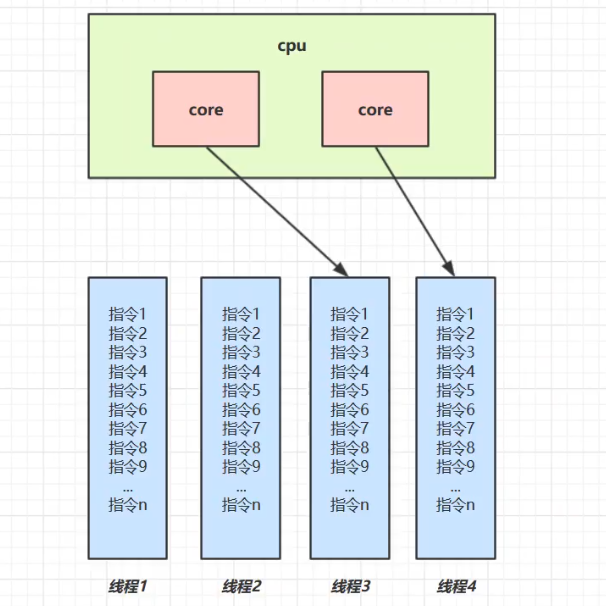
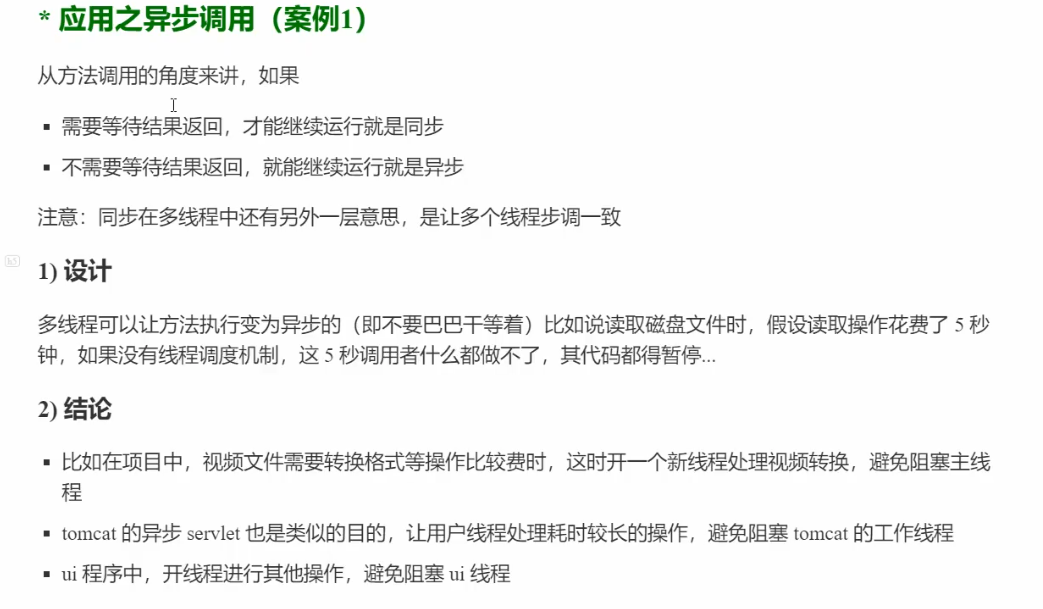


图. 并行parallel（多核心CPU）

## 5 应用

### 5.1应用之异步调用



### 5.2应用之提高效率

