# 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

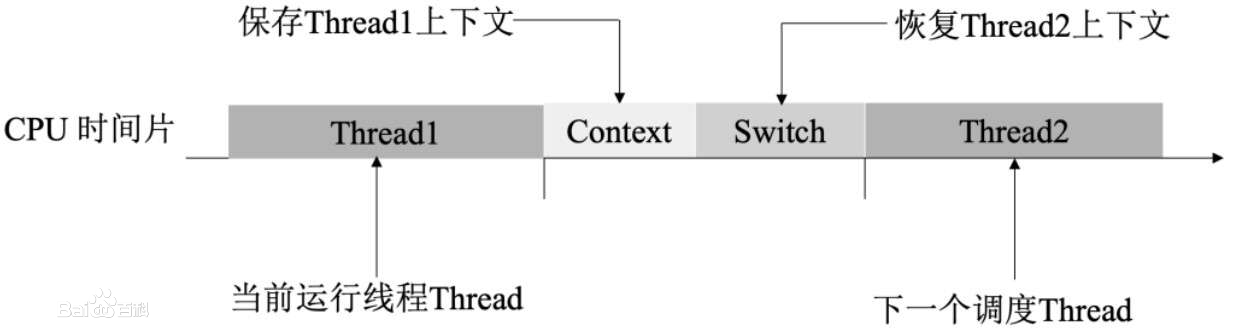
上下文的切换流程如下 [1] 

（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。

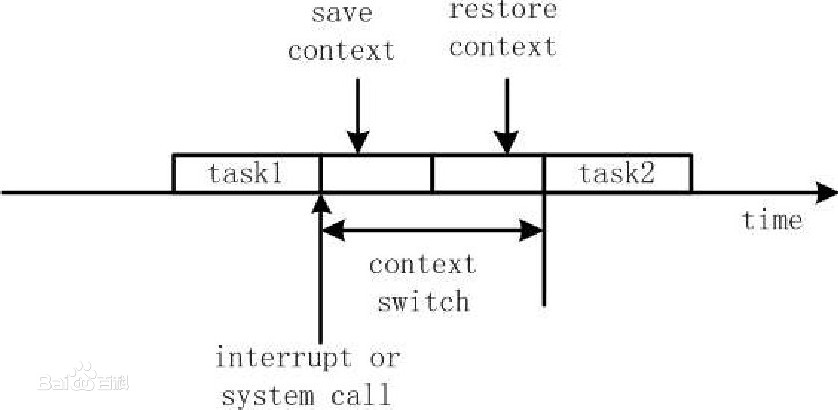
（4）用户的代码挂起当前任务，比如线程执行sleep方法，让出CPU。

（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。**

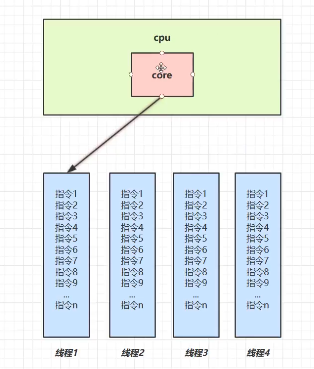


图. 并发Concurrent （单核CPU）

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





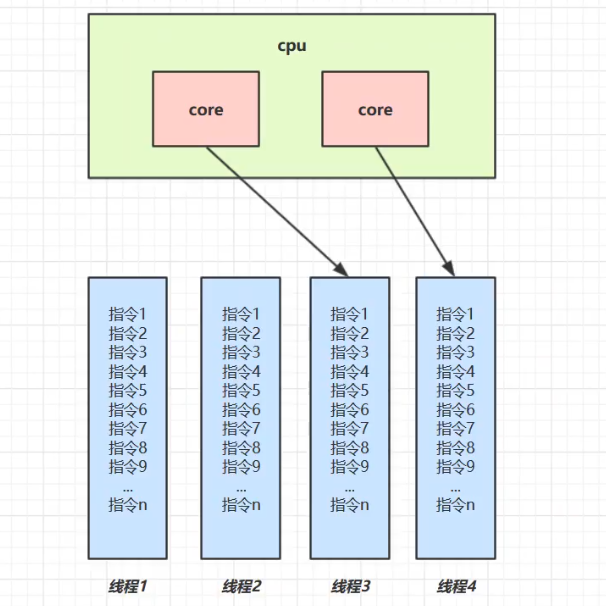
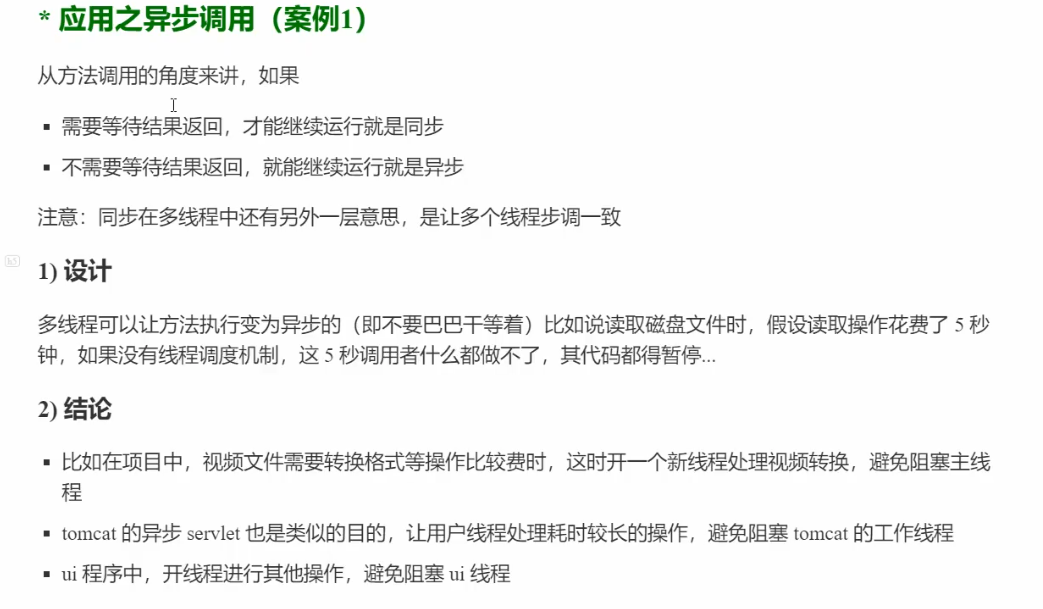


图. 并行parallel（多核心CPU）

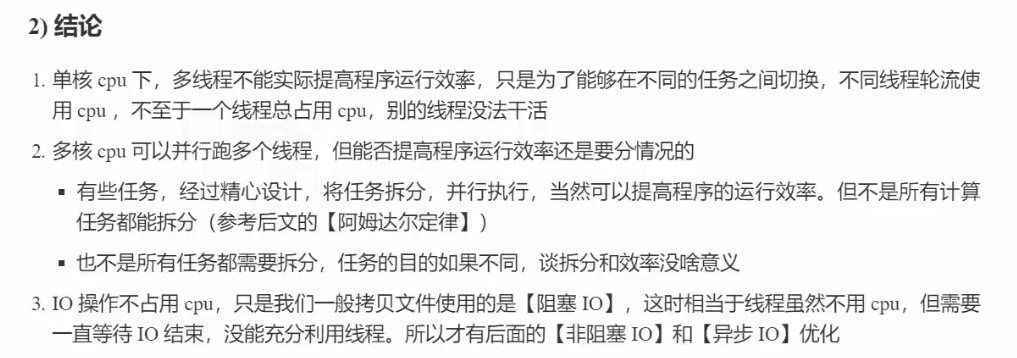
## 5 应用

### 5.1应用之异步调用



### 5.2应用之提高效率





### 5.3 Lambda表达式

Java中有一些接口，只有一个抽象方法，通常这样的接口会用@FunctionInterface注解标识，这样的接口比如Runnable接口，里面只有一个run方法，这时当我们需要常见接口类型的对象时，就可以使用Lamuda表达式来简化，说白了Lamuda表达式，就是一个匿名函数，既没有函数名的函数，Lamuda表达式可以使用闭包

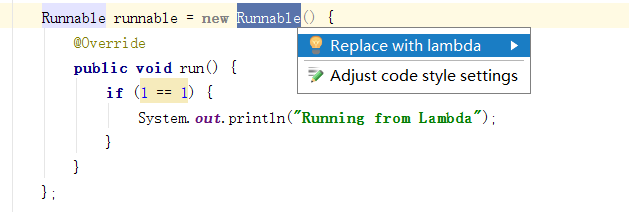
当开发者在编写Lambda表达式时，也会随之被编译成一个函数式接口,如果接口中有多个抽象方法，是无法使用Lambda表达式简化的

#### 5.3.1不采用Lambda的老方法：

Runnable runnable1=**new** Runnable(){  
 @Override  
 **public void** run(){  
 System.***out***.println(**"Running without Lambda"**);  
 }  
};

#### 5.3.2使用Lambda：

只有一行语句的时候，大括号也是可以省略的，也可以使用IDEA中的快捷键alt + Enter将代码简化为Lambda表达式的写法：



Runnable runnable2=()->System.***out***.println(**"Running from Lambda"**);  
  
Thread t = **new** Thread(runnable2);  
  
t.setName(**"test\_Lambda"**);  
t.start();

更简洁的代码如下：



#### 5.3.3原理之 Thread 与 Runnable 的关系

分析 Thread 的源码，理清它与 Runnable 的关系

小结

方法1 是把线程和任务合并在了一起，方法2 是把线程和任务分开了

用 Runnable 更容易与线程池等高级 API 配合

用 Runnable 让任务类脱离了 Thread 继承体系，更灵活

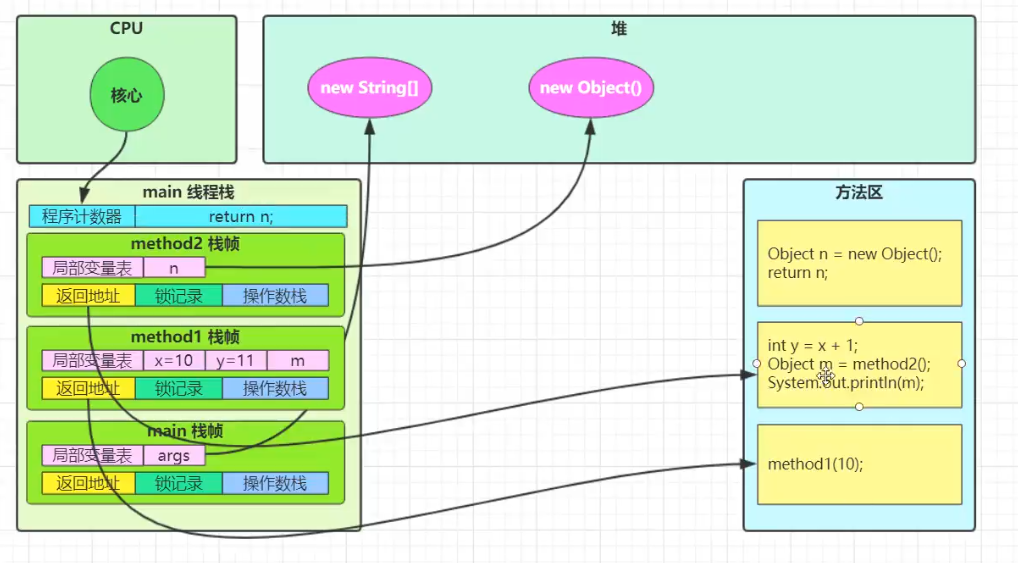
就是说在我们的Java里，关于继承和组合的选择，我们优先更偏向于选择组合，而Thread和Runnable就是属于组合这种形式，使得我们的任务和线程分开，编码和架构都将更加灵活

## 6. 线程运行原理

Java Virtual Machine Stacks （Java 虚拟机栈）

我们都知道 JVM 中由堆、栈、方法区所组成，其中栈内存是给谁用的呢？其实就是线程，每个线程启动后，虚拟机就会为其分配一块栈内存。

* 每个栈由多个栈帧（Frame）组成，对应着每次方法调用时所占用的内存
* 每个线程只能有一个活动栈帧，对应着当前正在执行的那个方法



线程的栈内存是相互独立的，每个线程都有自己的栈内存，栈内存里面有多个栈帧，它们互不干扰

#### 6.2线程上下文切换（Thread Context Switch）

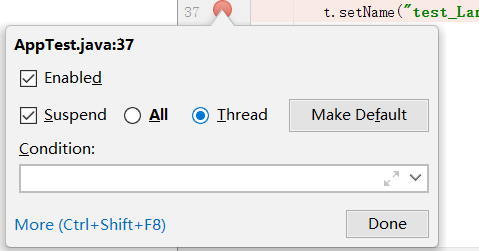
因为以下一些原因导致 cpu 不再执行当前的线程，转而执行另一个线程的代码，

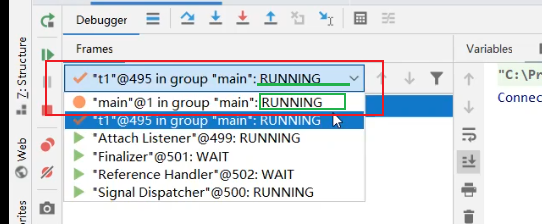
线程的上下文切换，就是从使用CPU到不适用CPU，比如：一个线程执行一定时间片后，CPU中的任务调度器，将CPU的使用权，分给了另外一个线程（或者说，另一个线程竞争，获取到了CPU的使用权）

* 线程的 cpu 时间片用完
* 垃圾回收:垃圾回收了，会暂停所有的工作线程，让垃圾回收的线程回收垃圾，线程的暂停就会导致线程上下文的切换，
* 有更高优先级的线程需要运行
* 线程自己调用了 sleep、yield、wait、join、park、synchronized、lock 等方法，
* 当 Context Switch 发生时，需要由操作系统保存当前线程的状态，并恢复另一个线程的状态，Java 中对应的概念就是程序计数器（Program Counter Register），它的作用是记住下一条 jvm 指令的执行地址，是线程私有的
* 状态包括程序计数器、虚拟机栈中每个栈帧的信息，如局部变量、操作数栈、返回地址等
* Context Switch 频繁发生会影响性能

#### 6.3多线程调试debug技巧

要选择debug模式为Thread，不要选All选了all就看不到并发运行的效果





## 7.设计模式-两阶段终止

Two Phase Termination：在一个线程T1如何“优雅终止线程T2”，这里的优雅指的是给T2一个料理后事的机会

### 7.1错误思路

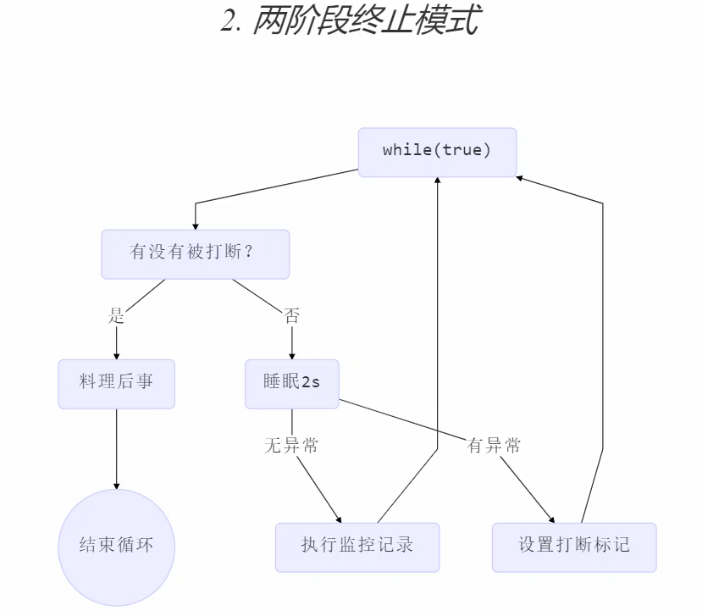
#### 7.1.1使用线程对象的stop方法停止线程

Stop方法会真正杀死线程，如果这时线程锁住了共享资源，那么当它被杀死后就再也没有机会释放锁，其它线程将永远无法获取锁

#### 7.1.2使用System.exit(int)方法停止线程

目的仅是停止一个线程，但这种做法会让整个程序都停止，整个进程都停下来了

### 7.2正确的做法



#### 7.2.1使用interrupt打断线程的方法来终止线程

#### 7.2.2代码示例

*/\*\*  
 \* author: ZhaoBO  
 \* description：两阶段终止模式练习  
 \* Date：2020/6/7 - 15:34  
 \*/*@Slf4j(topic = **"TwoPhaseTerminationTest"**)  
**public class** TwoPhaseTerminationTest {  
  
 *//下面这种问题，出现不止一次了，出现这种方法不能够调用的问题，就是你调用方法的代码本身没有写在一个方法里，我去奥，fuck  
 /\* TwoPhaseTermination twoPhaseTermination = new TwoPhaseTermination();  
 TwoPhaseTermination.start();\*/* **public static void** main(String[] args) **throws** InterruptedException {  
  
 TwoPhaseTermination monitorTask = **new** TwoPhaseTermination();  
 monitorTask.start();  
  
 *//主线程6秒后打断监控线程* Thread.*sleep*(6000);  
 monitorTask.stop();  
 }  
  
}  
  
@Slf4j(topic = **"TwoPhaseTermination"**)  
**class** TwoPhaseTermination {  
  
 **private** Thread **monitor**;  
  
 **public void** start() {  
 **monitor** = **new** Thread(() -> {  
  
 **while** (**true**) {  
 Thread currentThread = Thread.*currentThread*();  
 **if** (currentThread.isInterrupted()) {  
 ***log***.info(**"料理后事，退出循环，终止线程"**);  
 **break**;  
 }**else** {  
 **try** {  
 Thread.*sleep*(2000);  
 ***log***.info(**"执行监控记录"**);  
 } **catch** (InterruptedException e) {  
 */\*\*  
 \* 这里捕获异常，sleep方法会出现异常的情况，当程序调用interrupt方法的时候，sleep就会抛出InterruptedException异常  
 \*/* e.printStackTrace();  
 *//捕获异常后，重新设置打断标记，所谓的重新设置打断标记，也就是用线程对象，调用interrupt方法，  
 //执行到这里，属于在程序正常运行时打断线程，所以打断标记会被设置为true，而sleep时，线程处于阻塞状态，这时调用interrupt打断  
 //线程时，打断标记会被清空，为false* **monitor**.interrupt();  
 }  
 }  
 }  
 },**"monitor"**);  
  
 **monitor**.start();  
 }  
  
 **public void** stop() {  
 *//打断线程* **monitor**.interrupt();  
 }  
  
}

## 8.共享带来的问题

### 8.1线程安全

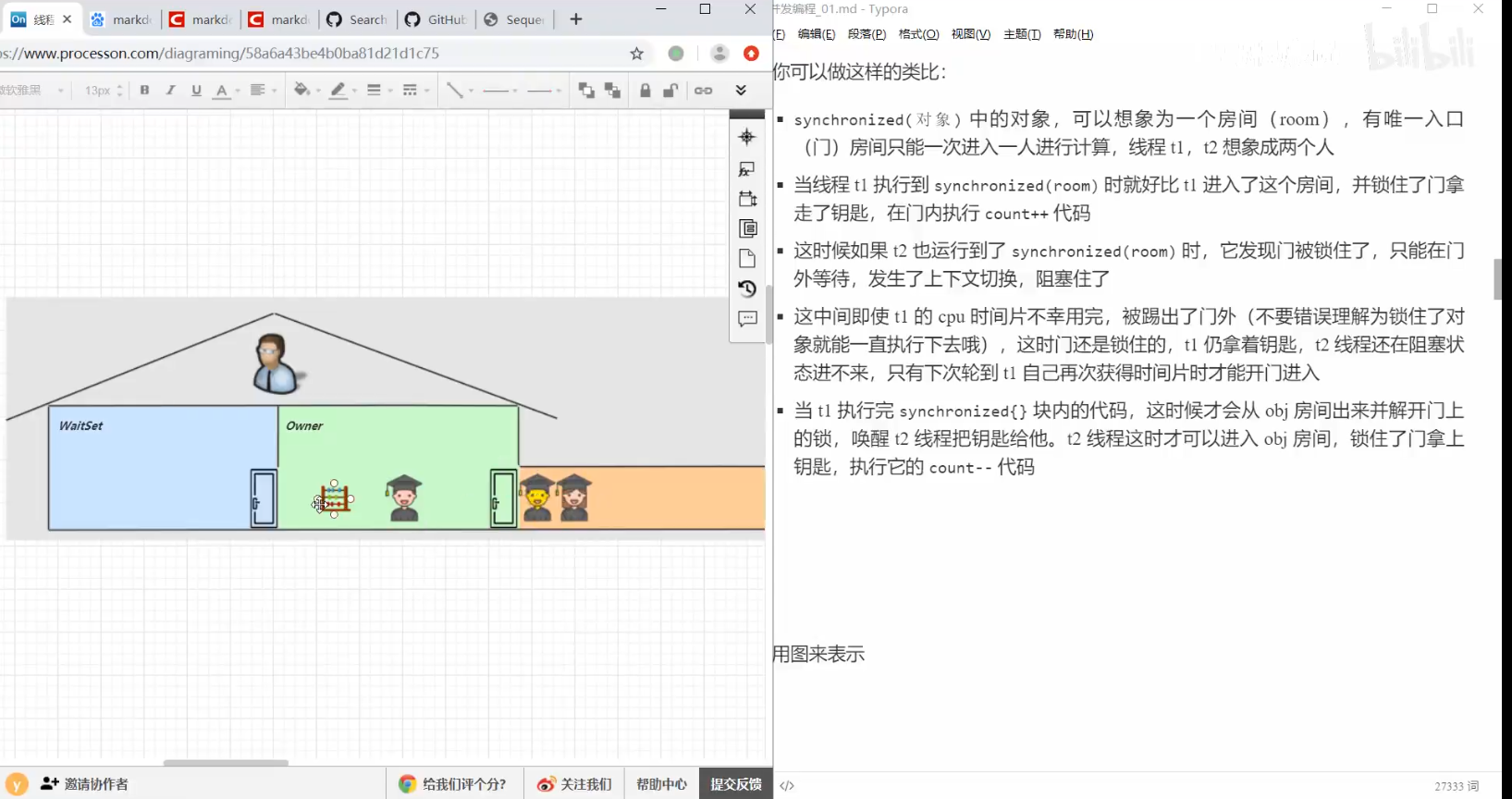
多线程下，访问共享资源，由于分时系统而产生的问题

也就是说操作系统是一个分时系统，任务调度器，会将CPU的时间片，分给不同的线程使用，一个线程在执行的过程中，可能出现阻塞的各种情况，让出CPU的时间片，任务调度器就会将CPU的时间片分给其它线程，这时候如果多个线程访问的是同一个共享资源，就会产生线程安全问题，举个栗子：

比如A线程和B线程都访问同一个变量c，在线程执行过程中都会改变变量c的值，比如A线程每次执行过程中会对变量c进行加一操作，线程B每次执行会对变量c减一操作，任务调度器首先将CPU的时间片分给了线程A，线程A执行到一半，可能出现了问题进入了阻塞状态（比如sleep，wait，阻塞IO），这时候线程A会记录下本次执行结果（也就是线程A执行到哪里了），然后让出CPU的时间片，线程A从运行状态进入了就绪状态；线程B获得CPU的时间片后开始执行，并对变量c进行了减一操作，比如c初始值未0，减一后c变为了-1，线程B执行完毕后进入了终止状态；这时候任务调度器又将CPU的时间片分给了A线程，A线程又从上次执行到的位置，开始执行了，可是A线程上次获取到的变量c的值还是初始值0，这时对变量c进行了+1操作并赋值给了变量c,

正常情况c初始值为0B线程对变量c减一，A线程对变量c加一，最终结果应该还是0，但是由于分时系统，多线程下，访问共享资源，产生了线程不安全问题

### 8.2Synchronized原理分析



### 8.3分析线程安全问题

要分析线程安全问题，就得考虑哪部分代码是处于临界区，找出的临界区，我们再想办法给它加锁

那么什么是临界区呢？（这个问题，曾今面试的时候，就被面试官问道过，当时被问的是一脸懵逼）：所谓的临界区就是，一段代码块内，如果存在对共享资源的多线程读写操作，这段代码块就是临界区

如下所示：

static int counter = 0;

static void increment()

// 临界区

{

counter++;

}

static void decrement()

// 临界区

{

counter--;

}

这里的counter就是一个共享资源，可能有多个线程同时对这个共享资源进行读写操作，比如线程A执行了increment方法，线程B执行decrement方法，都将对这个共享变量counter造成影响，而对counter造成影响的代码是counter++和counter—所以，这两段代码块就是属于临界区

### 8.4Volatile保证可见性、有序性



就拿双重检查的单例模式模式来说：

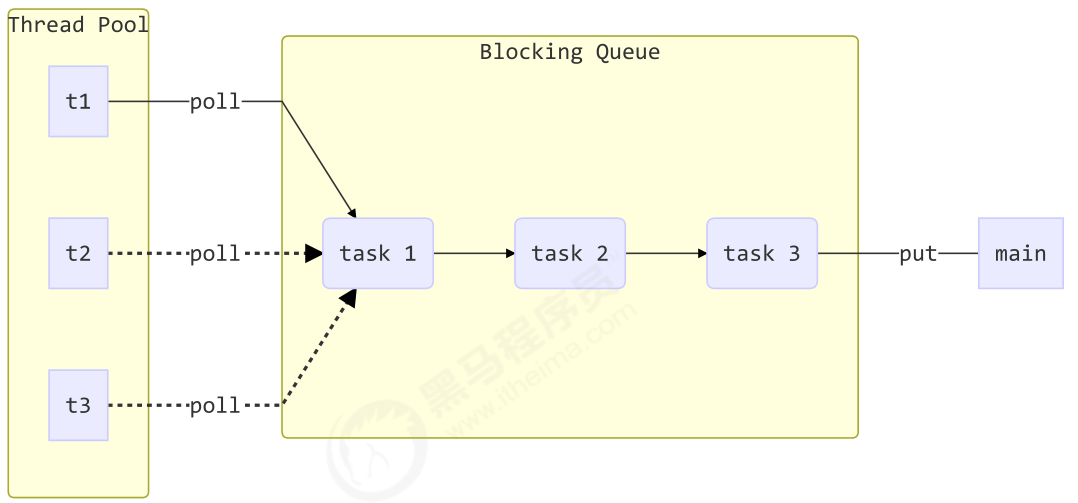
Synchronized能保证共享变量的可见性、原子性、有序性，但前提是，共享变量都要被Synchronized保护，也就是在Synchronized代码块内部，synchronized实际上并不能解决由于JVM自身对访问共享变量的优化，引起的指令交错问题,那么synchronized是怎么保证原子性，可见性了，这就是锁机制，线程互斥，来保证的。

拿如上代码来说，如果t1线程，首先进来了，判断INSTANCE实例为null，就会执行到synchronized代码块内部，但是由于JVM自身做出的优化，可能会出现指令指令重排，比如赋值，和构造方法的执行，发生了指令重排，可能先赋值，再执行的构造方法，但是虽然synchronized代码块内指令重排了，但是线程互斥，锁机制，即使发生了线程上下文的切换（任务调度器把CPU时间片分给了其它线程，如T2）,这时如果对共享变量的操作都被synchronized代码块包裹，T2线程由于获取不到锁，继续阻塞，只有T1线程执行完毕释放锁锁后，T2线程才能正常执行，这时T1线程在同步代码块内部对共享变量的整体操作（即使指令重排）对synchronized外部，或者其它线程来说，它是有序、原子、可见的

但是如上代码对共享变量的操作没有全部被synchronized保护，这时候synchronized内部代码的指令重排，对结果就会有影响，比如synchronized内部发生了指令重排，T1线程先赋值，再执行构造方法，但是T1执行完赋值操作后，还没有执行构造方法，这时来了另外一个线程T2，由于synchronized外部还有一个对共享变量的操作，T2线程不需要获取到锁就能执行到那段代码，T1线程已经赋值了，于是T2获取到了，判断不为null，于是拿着还未初始化完毕的实例去用了，着肯定就会存在问题，这时候就可以看出，synchronized不能解决指令重排问题，而使用volatile来修饰共享变量就能解决指令重排问题，以此来保证有序性性，通过读写屏障，被volatile修饰的共享变量，不能被指令重排序；在被volatile修饰的共享变量写操作的后面会产生写屏障，还有传导性，会使得对包括volatile修饰的共享变量及其之前的其它变量的写操作，都将数据写到主存中，在被volatile修饰的共享变量的读操作前会产生读屏障，会使得对包括volatile修饰的共享变量及其之后的其它变量的读操作都将到主存中读取数据，以此来保证可见性

如上，当共享变量被volatile修饰后，即使共享变量没有完全被synchronized保护，但是synchronized代码块内部的代码将不会发生指令重排，也就是不会出现先赋值再执行构造方法的情况，这时后如果T2线程来了，判断不为null，那获取到的一定是正确初始化后的实例，即使如果实例为null.如果T1线程没有执行完毕，T2获取不到锁阻塞，T1执行完毕，释放锁后，进入第二次判断（双重检查）,由于T1已经正常运行结束，T2获取到了判断不为null返回实例，这时候的实例就是正确初始化的实例，也就是只要有一个线程成功执行后，后面来的线程都将能获取到正确的实例对象，而且能保证在JVM内存中只有一个唯一的该实例对象

## 9线程池原理分析



线程是一种系统资源，每创建一个新的线程，都要占用一定的内存（需要分配栈内存），如果是高并发的情况下，一下来了很多的任务，这时如果为每个任务都创建一个新的线程，那可以想象这样对内存的占用是相当大的，甚至有可能出现OutOfMemoryError：

还有一个问题就是线程并不是创建的越多越好，比如有大量的任务来了，创建了很多很多的线程，那实际上我们都知道这样是不对的，因为从CPU的角度出发，CPU的核心数是有限的，每个人的电脑或者服务器还不一样，一下子来了这么多线程，那CPU其实也处理不过来，那它必然就让其中一部分线程（获取不到CPU时间片的那些线程）进入阻塞，这就会引起线程的上下文切换问题，需要将当前线程的运行状态先保存，下次轮到这个线程运行时，还要恢复线程当初的那些状态，也就是线程的上下文切换，这个线程上下文切换的越频繁，实际上对系统性能影响越大，尤其是在高并发的情况下，频繁的上下文切换，反而会导致系统性能降低，因此处于这样两个原因，我们应该不是说每次任务来了，都创建新的线程，而是应该充分利用已有线程的潜力，去处理任务，而不是每次都创建新的线程去处理任务，这也体现了享元模式的设计思想，所以就有了线程池的出现。类似的数据库连接池采用的也是这种享元模式的设计思想

线程池就是创建一批线程，让这些线程能够得到重复的利用，这样呢既可以减少内存的占用，也可以减少线程的数量，避免他们频繁的发生上下文切换。

### 自定义线程池

如上图可知，自定义线程池，实际上分为了几个组件，第一个组件就是ThreadPool线程池，线程池里面就是有一些可以被重用的线程。第二个组件是BlockQuune阻塞队列，它体现的是这种生产者、消费者模式下，平衡他们之间速度差异的一个组件。线程池中的线程相当于任务的消费者，它去不断的获取任务，并执行任务；任务的生产者线程可能源源不断的产生新的任务，在这种生产者、消费者模式下，它们两边的速率可能是不一致的，比如说生产者这边迟迟没有提交新的任务，线程池中的消费者线程就要等待，在哪等呢？这时就需要一个阻塞队列提供一个等待的位置；另外一种情况可能就是任务量一下子特别多，每个线程都忙不过来了，那多出来的那些任务放哪里呢？也得放在这个阻塞队列里，所以这个阻塞队列非常的重要，它是平衡消费者线程和生产者线程之间的一个桥梁