1、 从0开始深入理解并发、线程与等待通知机制

> 【有道云笔记】1、从0开始深入理解并发、线程与等待通知机制
> https://note.youdao.com/s/1E1RuIhk

为什么我们要学习并发编程?

最直白的原因，因为面试需要，我们来看看阿里和美团对Java岗位的JD：

从上面两大互联网公司的招聘需求可以看到，大厂的Java岗的并发编程能力属于标配。

而在非大厂的公司，并发编程能力也是面试的极大加分项，而工作时善用并发编程则可以极大提升程序员在公司的技术话语权。

为什么开发中需要并发编程?

从阿里的岗位JD其实就能看出来，并发编程和性能优化是密切相关的，使用并发编程可以做到:

(1)加快响应用户的时间

比如我们经常用的迅雷下载,都喜欢多开几个线程去下载,谁都不愿意用一个线程去下载,为什么呢?答案很简单,就是多个线程下载快啊。

我们在做程序开发的时候更应该如此,特别是我们做互联网项目,网页的响应时间若提升1s,如果流量大的话,就能增加不少转换量。做过高性能web前端调优的都知道,要将静态资源地址用两三个子域名去加载,为什么?因为每多一个子域名,浏览器在加载你的页面的时候就会多开几个线程去加载你的页面资源,提升网站的响应速度。

(2)使你的代码模块化,异步化,简单化

例如我们实现电商系统，下订单和给用户发送短信、邮件就可以进行拆分，将给用户发送短信、邮件这两个步骤独立为单独的模块，并交给其他线程去执行。这样既增加了异步的操作，提升了系统性能，又使程序模块化,清晰化和简单化。

多线程应用开发的好处还有很多,大家在日后的代码编写过程中可以慢慢体会它的魅力。

(3)充分利用CPU的资源

目前市面上没有CPU的内核不是多核的，比如这台机器

多核下如果还是使用单线程的技术做思路明显就out了，无法充分利用CPU的多核特点。如果设计一个多线程的程序的话,那它就可以同时在多个CPU的多个核的多个线程上跑，可以充分地利用CPU,减少CPU的空闲时间,发挥它的运算能力,提高并发量。

就像我们平时坐地铁一样,很多人坐长线地铁的时候都在认真看书,而不是为了坐地铁而坐地铁,到家了再去看书,这样你的时间就相当于有了两倍。这就是为什么有些人时间很充裕,而有些人老是说没时间的一个原因,工作也是这样,有的时候可以并发地去做几件事情,充分利用我们的时间,CPU也是一样，也要充分利用。

当然有同学会有疑问，单核CPU呢？单核CPU一样可以利用到并发编程的好处吗？当然可以，用我们平时常用的QQ之类的聊天程序来举例，当我们用QQ聊天时，其实程序要做好几件事，比如：接受我们的键盘输入，把输入的信息通过网络发给对方，接受对方通过网络发来的信息，把对方的信息显示在屏幕上，很多的时候，这些事情是可以同时发生的。如果程序不能利用并发编程同时处理，我们和对方的通话就只能一问一答的方式进行了。

我们怎么学并发编程?

> 深入浅出并发编程：https://www.processon.com/view/link/66582d1f411e091210866e71?cid=66582ce7783d400b4cd52506

课程章节安排如下:

1、从0开始深入理解并发、线程与等待通知机制

2、异步编程Future&CompletableFuture实战

3、导致JVM内存泄露的ThreadLocal详解

4、并发编程之CAS&Atomic原子操作详解

5、深入理解独占锁Synchronized底层原理

6、JUC并发工具类在大厂的应用场景详解

7、深入理解AQS之ReentrantLock源码分析

8、Semaphore&CountDownlatch&CyclicBarrier源码分析

9、并发容器（Map、List、Set）实战及其原理

10、阻塞队列BlockingQueue实战及其原理分析

11、线程池ThreadPoolExecutor实战及其原理分析

12、线程池ForkJoinPool工作原理分析

13、深入理解并发可见性、有序性、原子性与JMM内存模型

14、CPU缓存架构详解&高性能内存队列Disruptor实战

可以看到并发编程的课时其实是相当多的，反过来也说明并发编程在Java程序员的技能栈中重要地位。

对于没有或者很少接触并发编程的同学，建议主要掌握并发里的基础概念、基础用法和并发工具类、并发容器的用法，章节主要对应第1 ~ 5 章、第9、10、11、12。对于已经有较多并发编程经验建议全部学习。

注意：以上的章节和授课顺序是根据并发编程本身的知识结构和人类学习的认知机制设计安排的，和具体授课时的课程数、课程标题可能存在一定的不匹配情况，因此出现**一节课讲述多个章节和一个章节跨越多节课属于正常现象**。

基础概念

在正式学习Java的并发编程之前，还有几个并发编程的基础概念我们需要熟悉和学习。

进程和线程

进程

我们常听说的是应用程序，也就是app，由指令和数据组成。但是当我们不运行一个具体的app时，这些应用程序就是放在磁盘（也包括U盘、远程网络存储等等）上的一些二进制的代码。一旦我们运行这些应用程序，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。在指令运行过程中还需要用到磁盘、网络等设备，从这种角度来说，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。大部分程序可以同时运行多个实例进程（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 安全卫士等）。显然,程序是死的、静态的,进程是活的、动态的。进程可以分为系统进程和用户进程。凡是用于完成操作系统的各种功能的进程就是系统进程,它们就是处于运行状态下的操作系统本身,用户进程就是所有由你启动的进程。

站在操作系统的角度，进程是程序运行资源分配（以内存为主）的最小单位。

线程

一个机器中肯定会运行很多的程序，CPU又是有限的，怎么让有限的CPU运行这么多程序呢？就需要一种机制在程序之间进行协调，也就所谓CPU调度。线程则是CPU调度的最小单位。

线程必须依赖于进程而存在，线程是进程中的一个实体，是CPU调度和分派的基本单位，它是比进程更小的、能独立运行的基本单位。线程自己基本上不拥有系统资源，，只拥有在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。一个进程可以拥有多个线程，一个线程必须有一个父进程。线程，有时也被称为轻量级进程(Lightweight Process，LWP），早期Linux的线程实现几乎就是复用的进程，后来才独立出自己的API。

进程与线程的区别：

- 进程基本上相互独立的，而线程存在于进程内，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（Inter-process communication）
  - 不同计算机之间的进程通信，需要通过网络，并遵守共同的协议，例如 HTTP
- 线程通信相对简单，因为它们共享进程内的内存，一个例子是多个线程可以访问同一个共享变量
- 线程更轻量，线程上下文切换成本一般上要比进程上下文切换低

CPU核心数和线程数的关系

前面说过，目前主流CPU都是多核的，线程是CPU调度的最小单位。同一时刻，一个CPU核心只能运行一个线程，也就是CPU内核和同时运行的线程数是1:1的关系，也就是说8核CPU同时可以执行8个线程的代码。但 Intel引入超线程技术后，产生了逻辑处理器的概念，使核心数与线程数形成1:2的关系。在我们前面的Windows任务管理器贴图就能看出来，内核数是6而逻辑处理器数是12。

在Java中提供了Runtime.getRuntime().availableProcessors()，可以让我们获取当前的CPU核心数，注意这个核心数指的是逻辑处理器数。

获得当前的CPU核心数在并发编程中很重要，并发编程下的性能优化往往和CPU核心数密切相关。

上下文切换（Context switch）

既然操作系统要在多个进程（线程）之间进行调度，而每个线程在使用CPU时总是要使用CPU中的资源，比如CPU寄存器和程序计数器。这就意味着，操作系统要保证线程在调度前后的正常执行，所以，操作系统中就有上下文切换的概念，它是指CPU(中央处理单元)从一个进程或线程到另一个进程或线程的切换。

> 上下文是CPU寄存器和程序计数器在任何时间点的内容。
> 寄存器是CPU内部的一小部分非常快的内存(相对于CPU内部的缓存和CPU外部较慢的RAM主内存)，它通过提供对常用值的快速访问来加快计算机程序的执行。
> 程序计数器是一种专门的寄存器，它指示CPU在其指令序列中的位置，并保存着正在执行的指令的地址或下一条要执行的指令的地址，这取决于具体的系统。

上下文切换可以更详细地描述为内核(即操作系统的核心)对CPU上的进程(包括线程)执行以下活动:

1. 暂停一个进程的处理，并将该进程的CPU状态(即上下文)存储在内存中的某个地方
2. 从内存中获取下一个进程的上下文，并在CPU的寄存器中恢复它
3. 返回到程序计数器指示的位置(即返回到进程被中断的代码行)以恢复进程。

从数据来说，以程序员的角度来看， 是方法调用过程中的各种局部的变量与资源； 以线程的角度来看， 是方法的调用栈中存储的各类信息。

引发上下文切换的原因一般包括：线程、进程切换、系统调用等等。上下文切换通常是计算密集型的，因为涉及一系列数据在各种寄存器、 缓存中的来回拷贝。就CPU时间而言，一次上下文切换大概需要5000~20000个时钟周期，相对一个简单指令几个乃至十几个左右的执行时钟周期，可以看出这个成本的巨大。

并发和并行

我们举个例子,如果有条高速公路A上面并排有8条车道,那么最大的**并行**车辆就是8辆，此条高速公路A同时并排行走的车辆小于等于8辆的时候,车辆就可以并行运行。CPU也是这个原理,一个CPU相当于一个高速公路A,核心数或者线程数就相当于并排可以通行的车道;而多个CPU就相当于并排有多条高速公路,而每个高速公路并排有多个车道。

当谈论**并发**的时候一定要加个单位时间,也就是说单位时间内并发量是多少？离开了单位时间其实是没有意义的。

综合来说：

并发Concurrent:指应用能够交替执行不同的任务,比如单CPU核心下执行多线程并非是同时执行多个任务,如果你开两个线程执行,就是在你几乎不可能察觉到的速度不断去切换这两个任务,已达到"同时执行效果",其实并不是的,只是计算机的速度太快,我们无法察觉到而已。一般会将这种 线程轮流使用 CPU 的做法称为并发。总结为一句话就是： 微观串行，宏观并行。

并行Parallel:指应用能够同时执行不同的任务,例:吃饭的时候可以边吃饭边打电话,这两件事情可以同时执行。多核 cpu下，每个 核（core） 都可以调度运行线程，这时候线程可以是并行的。

两者区别:一个是交替执行,一个是同时执行。

认识Java里的线程

Java程序天生就是多线程的

一个Java程序从main()方法开始执行，然后按照既定的代码逻辑执行，看似没有其他线程参与，但实际上Java程序天生就是多线程程序，因为执行main()方法的是一个名称为main的线程。

示例代码

```
public class OnlyMain {
    public static void main(String[] args){
        //Java 虚拟机线程系统的管理接口
        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
        // 不需要获取同步的monitor和synchronizer信息，仅仅获取线程和线程堆栈信息
        ThreadInfo[] threadInfos =
                threadMXBean.dumpAllThreads(false, false);
        // 遍历线程信息，仅打印线程ID和线程名称信息
        for (ThreadInfo threadInfo : threadInfos) {
            System.out.println("[" + threadInfo.getThreadId() + "] "
                    + threadInfo.getThreadName());
        }

    }
}
```

而一个Java程序的运行就算是没有用户自己开启的线程，实际也有有很多JVM自行启动的线程，一般来说有：

[6] Monitor Ctrl-Break //监控Ctrl-Break中断信号的

[5] Attach Listener //内存dump，线程dump，类信息统计，获取系统属性等

[4] Signal Dispatcher // 分发处理发送给JVM信号的线程

[3] Finalizer // 调用对象finalize方法的线程

[2] Reference Handler//清除Reference的线程

[1] main //main线程，用户程序入口

尽管这些线程根据不同的JDK版本会有差异，但是依然证明了Java程序天生就是多线程的。

线程的创建和启动

刚刚看到的线程都是JVM启动的系统线程，我们学习并发编程希望的自己能操控线程，所以我们先来看看如何创建和启动线程。

创建和启动线程的方式有：

方式1：使用 Thread类或继承Thread类

```
// 创建线程对象
Thread t = new Thread() {
      public void run() {
      // 要执行的任务
      }
};
// 启动线程
t.start();
```

示例：

```
// 构造方法的参数是给线程指定名字，推荐
Thread t1 = new Thread("t1") {
      @Override
      // run 方法内实现了要执行的任务
      public void run() {
          log.debug("Hello Thread");
      }
};
t1.start();
```

方式2：实现 Runnable 接口配合Thread

把【线程】和【任务】（要执行的代码）分开

- Thread 代表线程
- Runnable 可运行的任务（线程要执行的代码）

```
Runnable runnable = new Runnable() {
     public void run(){
         // 要执行的任务
     }
};
// 创建线程对象
Thread t = new Thread( runnable );
// 启动线程
t.start();
```

示例：

```
// 创建任务对象
Runnable task2 = new Runnable() {
     @Override
     public void run() {
         log.debug("hello");
     }
};
// 参数1 是任务对象; 参数2 是线程名字，推荐
Thread t2 = new Thread(task2, "t2");
t2.start();
```

Java 8 以后可以使用 lambda 精简代码

```
// 创建任务对象
Runnable task2 = () -> log.debug("hello");
// 参数1 是任务对象; 参数2 是线程名字，推荐
Thread t2 = new Thread(task2, "t2");
t2.start();
```

小结

- Thread才是Java里对线程的唯一抽象，Runnable只是对任务（业务逻辑）的抽象。Thread可以接受任意一个Runnable的实例并执行。
- 方式1 是把线程和任务合并在了一起，方式2 是把线程和任务分开了。
- 用 Runnable 让任务类脱离了 Thread 继承体系，更灵活，更容易与线程池等高级 API 配合

方式3：使用FutureTask 配合 Thread

FutureTask 能够接收 Callable 类型的参数，用来处理有返回结果的情况。

```
// 创建任务对象
FutureTask<Integer> task3 = new FutureTask<>(() -> {
     log.debug("hello");
     return 100;
});
// 参数1 是任务对象; 参数2 是线程名字，推荐
new Thread(task3, "t3").start();
// 主线程阻塞，同步等待 task 执行完毕的结果
Integer result = task3.get();
log.debug("结果是:{}", result);
```

Runnable是一个接口，在它里面只声明了一个run()方法，由于run()方法返回值为void类型，所以在执行完任务之后无法返回任何结果。

Callable位于java.util.concurrent包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做call()，这是一个泛型接口，call()函数返回的类型就是传递进来的V类型。

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask。

FutureTask类实现了RunnableFuture接口，RunnableFuture继承了Runnable接口和Future接口，而FutureTask实现了RunnableFuture接口。所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

因此我们通过一个线程运行Callable，但是Thread不支持构造方法中传递Callable的实例，所以我们需要通过FutureTask把一个Callable包装成Runnable，然后再通过这个FutureTask拿到Callable运行后的返回值。

要new一个FutureTask的实例，有两种方法

面试题：创建线程有几种方式?

这个问题的答案其实众说纷纭，有2种，3种，4种等等答案，建议比较好的回答是：

按照Java源码中Thread上的注释：

官方说法是在Java中有两种方式创建一个线程用以执行，一种是派生自Thread类，另一种是实现Runnable接口。

当然本质上Java中实现线程只有一种方式，都是通过new Thread()创建线程对象，调用Thread#start启动线程。

至于基于callable接口的方式，因为最终是要把实现了callable接口的对象通过FutureTask包装成Runnable，再交给Thread去执行，所以这个其实可以和实现Runnable接口看成同一类。

而线程池的方式，本质上是池化技术，是资源的复用，和新启线程没什么关系。

所以，比较赞同官方的说法，有两种方式创建一个线程用以执行。

run和start

Thread类是Java里对线程概念的抽象，可以这样理解：我们通过new Thread()其实只是new出一个Thread的实例，还没有操作系统中真正的线程挂起钩来。只有执行了start()方法后，才实现了真正意义上的启动线程。

从Thread的源码可以看到，Thread的start方法中调用了start0()方法，而start0()是个native方法，这就说明Thread#start一定和操作系统是密切相关的。

start()方法让一个线程进入就绪队列等待分配cpu，分到cpu后才调用实现的run()方法，start()方法不能重复调用，如果重复调用会抛出异常（**注意，此处可能有面试题：多次调用一个线程的start方法会怎么样?**）。

而run方法是业务逻辑实现的地方，本质上和任意一个类的任意一个成员方法并没有任何区别，可以重复执行，也可以被单独调用。

深入学习Java的线程

线程的状态/生命周期

五种状态

这是从 操作系统 层面来描述的

- 【初始状态】仅是在语言层面创建了线程对象，还未与操作系统线程关联
- 【可运行状态】（就绪状态）指该线程已经被创建（与操作系统线程关联），可以由 CPU 调度执行
- 【运行状态】指获取了 CPU 时间片运行中的状态
  - 当 CPU 时间片用完，会从【运行状态】转换至【可运行状态】，会导致线程的上下文切换
- 【阻塞状态】
  - 如果调用了阻塞 API，如 BIO 读写文件，这时该线程实际不会用到 CPU，会导致线程上下文切换，进入【阻塞状态】
  - 等 BIO 操作完毕，会由操作系统唤醒阻塞的线程，转换至【可运行状态】
  - 与【可运行状态】的区别是，对【阻塞状态】的线程来说只要它们一直不唤醒，调度器就一直不会考虑调度它们
- 【终止状态】表示线程已经执行完毕，生命周期已经结束，不会再转换为其它状态

六种状态

这是从 Java API 层面来描述的。

根据 Thread.State 枚举，Java中线程的状态分为6种：

1. 初始(NEW)：新创建了一个线程对象，但还没有调用start()方法。
2. 运行(RUNNABLE)：Java线程中将就绪（ready）和运行中（running）两种状态笼统的称为“运行”。线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取CPU的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得CPU时间片后变为运行中状态（running）。
3. 阻塞(BLOCKED)：表示线程阻塞于锁。
4. 等待(WAITING)：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。
5. 超时等待(TIMED_WAITING)：该状态不同于WAITING，它可以在指定的时间后自行返回。
6. 终止(TERMINATED)：表示该线程已经执行完毕。

状态之间的变迁如下图所示：

掌握这些状态可以让我们在进行Java程序调优时可以提供很大的帮助。

线程常见方法

方法名	static	功能说明	注意
start()		启动一个新线程，在新的线程运行 run 方法中的代码	start 方法只是让线程进入就绪，里面代码不一定立刻运行（CPU 的时间片还没分给它）。每个线程对象的start方法只能调用一次，如果调用了多次会出现IllegalThreadStateException
run()		新线程启动后会调用的方法	如果在构造 Thread 对象时传递了 Runnable 参数，则线程启动后会调用 Runnable 中的 run 方法，否则默认不执行任何操作。但可以创建 Thread 的子类对象，来覆盖默认行为
join()		等待线程运行结束	
join(long n)		等待线程运行结束,最多等待 n 毫秒	
getId()		获取线程长整型的 id	id 唯一
getName()		获取线程名	
setName(String)		修改线程名	
getPriority()		获取线程优先级	
setPriority(int)		修改线程优先级	java中规定线程优先级是1~10 的整数，较大的优先级能提高该线程被 CPU 调度的机率
getState()		获取线程状态	Java 中线程状态是用 6 个 enum 表示，分别为NEW,RUNNABLE,BLOCKED,WAITING,TIMED_WAITING,TERMINATED
isInterrupted()		判断是否被中断	不会清除 中断标记
isAlive()		线程是否存活（还没有运行完毕）	

interrupt()		中断线程	如果被中断线程正在 sleep，wait，join 会导致被中断的线程抛出 InterruptedException，并清除 中断标记 ；如果中断的正在运行的线程，则会设置中断标记 ；park 的线程被中断，也会设置中断标记
interrupted()	static	判断当前线程是否被中断	会清除中断标记
currentThread()	static	获取当前正在执行的线程	
sleep(long n)	static	让当前执行的线程休眠n毫秒，休眠时让出 cpu 的时间片给其它线程	
yield()	static	提示线程调度器让出当前线程对CPU的使用	

还有一些不推荐使用的方法，这些方法已过时，容易破坏同步代码块，造成线程死锁。

方法名	**static**	**功能说明**
stop()		停止线程运行
suspend()		挂起（暂停）线程运行
resume()		恢复线程运行

sleep与 yield

sleep方法

- 调用 sleep 会让当前线程从 Running 进入 Timed Waiting 状态（阻塞），不会释放对象锁
- 其它线程可以使用 interrupt 方法打断正在睡眠的线程，这时 sleep 方法会抛出InterruptedException
- 睡眠结束后的线程未必会立刻得到执行
- 建议用 TimeUnit 的 sleep 代替 Thread 的 sleep 来获得更好的可读性
- sleep当传入参数为0时，和yield相同

```
Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        log.debug("执行完成");

    }
},"t1");
t1.start();

log.debug("线程t1的状态："+t1.getState());

try {
    Thread.sleep(500);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}

log.debug("线程t1的状态："+t1.getState());

//t1.interrupt();
```

在没有利用 cpu 来计算时，不要让 while(true) 空转浪费 cpu，这时可以使用 yield 或 sleep 来让出 cpu 的使用权给其他程序。

```
while(true) {
      try {
          Thread.sleep(50);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
}
```

- 可以用 wait 或 条件变量达到类似的效果
- 不同的是，后两种都需要加锁，并且需要相应的唤醒操作，一般适用于要进行同步的场景
- sleep 适用于无需锁同步的场景

yield方法

- yield会释放CPU资源，让当前线程从 Running 进入 Runnable状态，让优先级更高（至少是相同）的线程获得执行机会，不会释放对象锁；
- 假设当前进程只有main线程，当调用yield之后，main线程会继续运行，因为没有比它优先级更高的线程；
- 具体的实现依赖于操作系统的任务调度器

比如ConcurrentHashMap#initTable 方法中就使用了yield方法，

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt( o: this, SIZECTL, sc, i1: -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    /unchecked/
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
```

这是因为ConcurrentHashMap中可能被多个线程同时初始化table，但是其实这个时候只允许一个线程进行初始化操作，其他的线程就需要被阻塞或等待，但是初始化操作其实很快，这里Doug Lea大师为了避免阻塞或者等待这些操作引发的上下文切换等等开销，就让其他不执行初始化操作的线程干脆执行yield()方法，以让出CPU执行权，让执行初始化操作的线程可以更快的执行完成。

线程的优先级

线程优先级会提示调度器优先调度该线程，但它仅仅是一个提示，调度器可以忽略它。如果 cpu 比较忙，那么优先级高的线程会获得更多的时间片，但 cpu 闲时，优先级几乎没作用。

在Java线程中，通过一个整型成员变量priority来控制优先级，优先级的范围从1~10，在线程构建的时候可以通过setPriority(int)方法来修改优先级，默认优先级是5，优先级高的线程分配时间片的数量要多于优先级低的线程。

设置线程优先级时，针对频繁阻塞（休眠或者I/O操作）的线程需要设置较高优先级，而偏重计算（需要较多CPU时间或者偏运算）的线程则设置较低的优先级，确保处理器不会被独占。在不同的JVM以及操作系统上，线程规划会存在差异，有些操作系统甚至会忽略对线程优先级的设定。

```
Runnable task1 = () -> {
    int count = 0;
    for (;;) {
        System.out.println("t1---->" + count++);
    }
};
Runnable task2 = () -> {
    int count = 0;
    for (;;) {
        // Thread.yield();
        System.out.println("t2---->" + count++);
    }
};
Thread t1 = new Thread(task1, "t1");
Thread t2 = new Thread(task2, "t2");
// t1.setPriority(Thread.MIN_PRIORITY);
// t2.setPriority(Thread.MAX_PRIORITY);
t1.start();
t2.start();
```

join方法

等待调用join方法的线程结束之后，程序再继续执行，一般用于等待异步线程执行完结果之后才能继续运行的场景。

为什么需要 join?

下面的代码执行，打印 count 是什么?

```
private static int count = 0;

public static void main(String[] args) throws InterruptedException {
    log.debug("开始执行");

    Thread t1 = new Thread(() -> {
        log.debug("开始执行");
        SleepTools.second(1);
        count = 5;
        log.debug("执行完成");
    },"t1");
    t1.start();
    log.debug("结果为:{}", count);
    log.debug("执行完成");
}
```

输出

```
19:30:09.614 [main] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 开始执行
19:30:09.660 [t1] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 开始执行
19:30:09.660 [main] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 结果为:0
19:30:09.662 [main] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 执行完成
19:30:10.673 [t1] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 执行完成
```

分析

- 因为主线程和线程 t1 是并行执行的，t1 线程需要 1 秒之后才能算出 count=5
- 而主线程一开始就要打印 count的结果，所以只能打印出 count=0

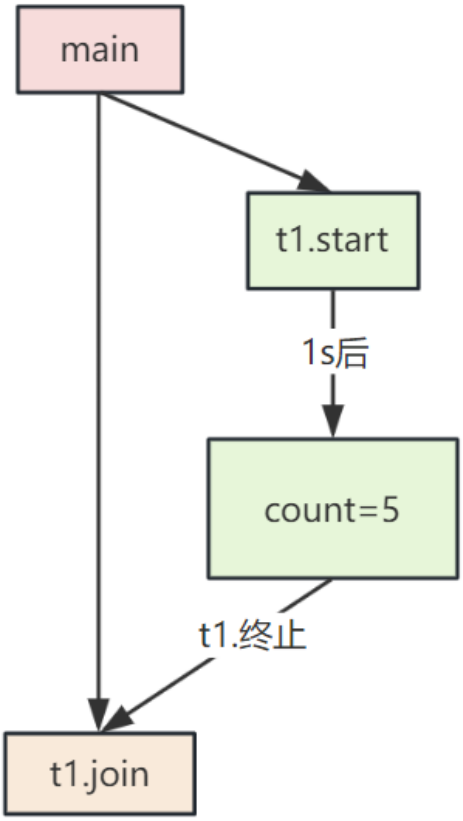
解决方法

- 用 sleep 行不行？为什么?
- 用 join，加在 t1.start() 之后即可

实现同步

以调用方角度来讲，如果：

- 需要等待结果返回，才能继续运行就是同步
- 不需要等待结果返回，就能继续运行就是异步


main
t1.start
1s后
count=5
t1.终止
t1.join

```
private static int count = 0;

public static void main(String[] args) throws InterruptedException {
    log.debug("开始执行");

    Thread t1 = new Thread(() -> {
        log.debug("开始执行");
        SleepTools.second(1);
        count = 5;
        log.debug("执行完成");
    },"t1");
    t1.start();

    //SleepTools.second(1);

    t1.join();

    log.debug("结果为:{}", count);
    log.debug("执行完成");
 }
```

输出

```
19:36:05.192 [main] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 开始执行
19:36:05.235 [t1] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 开始执行
19:36:06.239 [t1] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 执行完成
19:36:06.239 [main] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 结果为:5
19:36:06.240 [main] DEBUG com.tuling.learnjuc.base.ThreadJoinDemo - 执行完成
```

面试题

现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行?

可以利用join()方法实现，把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行。比如在线程T2中调用了线程T1的join()方法，直到线程T1执行完毕后，才会继续执行线程T2剩下的代码。

```
Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {
        log.debug("线程t1执行完成");
    }
},"t1");
Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            t1.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        log.debug("线程t2执行完成");
    }
},"t2");
Thread t3 = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            t2.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        log.debug("线程t3执行完成");
    }
},"t3");

t1.start();
t2.start();
t3.start();
```

守护线程

默认情况下，Java 进程需要等待所有线程都运行结束，才会结束。有一种特殊的线程叫做守护线程，只要其它非守护线程运行结束了，即使守护线程的代码没有执行完，也会强制结束。

```
log.debug("开始运行...");
Thread t1 = new Thread(() -> {
    log.debug("开始运行...");
    SleepTools.second(3);
    log.debug("运行结束...");
}, "t1");

// 设置t1线程为守护线程
t1.setDaemon(true);
t1.start();
SleepTools.second(1);
log.debug("运行结束...");
```

输出

```
21:21:58.104 [main] DEBUG com.tuling.learnjuc.base.DaemonThreadDemo - 开始运行...
21:21:58.143 [t1] DEBUG com.tuling.learnjuc.base.DaemonThreadDemo - 开始运行...
21:21:59.158 [main] DEBUG com.tuling.learnjuc.base.DaemonThreadDemo - 运行结束...
```

守护线程的应用场景

守护线程看起来好像没什么用？但是实际上它的作用很大。

- 比如在JVM中垃圾回收器就采用了守护线程，如果一个程序中没有任何用户线程，那么就不会产生垃圾，垃圾回收器也就不需要工作了。
- 在一些中间件的心跳检测、事件监听等涉及定时异步执行的场景中也可以使用守护线程，因为这些都是在后台不断执行的任务，当进程退出时，这些任务也不需要存在，而守护线程可以自动结束自己的生命周期。

从这些实际场景中可以看出，对于一些后台任务，当不希望阻止JVM进程结束时，可以采用守护线程。

线程的终止

线程自然终止

要么是run执行完成了，要么是抛出了一个未处理的异常导致线程提前结束。

面试题： 如何正确终止正在运行的线程?

stop（不要使用）

stop()方法已经被jdk废弃，调用stop方法无论run()中的逻辑是否执行完，都会释放CPU资源，释放锁资源。这会导致线程不安全，因为该方法会导致两个问题：

- 立即抛出ThreadDeath异常，在run()方法中任何一个执行指令都可能抛出ThreadDeath异常。
- 会释放当前线程所持有的所有的锁，这种锁的释放是不可控的。

比如：线程A的逻辑是转账（获得锁，1号账户减少100元，2号账户增加100元，释放锁），那线程A刚执行到1号账户减少100元就被调用了stop方法，释放了锁资源，释放了CPU资源。1号账户平白无故少了100元。一场撕逼大战开始了。

```
public class ThreadStopDemo {

    private static final Object lock = new Object();
    private static int account1 = 1000;
    private static int account2 = 0;

    public static void main(String[] args)  {
        Thread threadA = new Thread(new TransferTask(),"threadA");
        threadA.start();

        // 等待线程A开始执行
        SleepTools.ms(50);
        // 假设在转账过程中，我们强制停止了线程A
        threadA.stop();

        //验证锁是否释放
//        synchronized (lock){
//            System.out.println("主线程加锁成功");
//        }
    }

    static class TransferTask implements Runnable {
        @Override
        public void run() {
            synchronized (lock) {
                try{
                    System.out.println("开始转账...");
                    // 1号账户减少100元
                    account1 -= 100;
                    // 休眠100ms
                    SleepTools.ms(50);
                    // 假设在这里线程被stop了，那么2号账户将不会增加，且锁会被异常释放
                    System.out.println("1号账户余额: " + account1);
                    account2 += 100; // 2号账户增加100元
                    System.out.println("2号账户余额: " + account2);
                    System.out.println("转账结束...");
                }catch (Throwable t){
                    System.out.println("线程A结束执行");
```

```
39                      t.printStackTrace();
40                  }
41
42              }
43          }
44      }
45  }
```

因此，在实际应用中，一定不能使用stop()方法来终止线程，那么如何安全地实现线程的终止呢?

中断机制

安全的中止则是其他线程通过调用某个线程A的**interrupt()**方法对其进行中断操作, 中断好比其他线程对该线程打了个招呼，“A，你要中断了”，不代表线程A会立即停止自己的工作，同样的A线程完全可以不理会这种中断请求。线程通过检查自身的中断标志位是否被置为true来进行响应，

线程通过方法**isInterrupted()**来进行判断是否被中断，也可以调用静态方法**Thread.interrupted()**来进行判断当前线程是否被中断，不过Thread.interrupted()会同时将中断标识位改写为false。

如果一个线程处于了阻塞状态（如线程调用了thread.sleep、thread.join、obj.wait等），则在线程在检查中断标示时如果发现中断标示为true，则会在这些阻塞方法调用处抛出InterruptedException异常，并且在抛出异常后会立即将线程的中断标示位清除，即重新设置为false。

不建议自定义一个取消标志位来中止线程的运行。因为run方法里有阻塞调用时会无法很快检测到取消标志，线程必须从阻塞调用返回后，才会检查这个取消标志。这种情况下，使用中断会更好，因为，

1. 一般的阻塞方法，如sleep等本身就支持中断的检查，

2. 检查中断位的状态和检查取消标志位没什么区别，用中断位的状态还可以避免声明取消标志位，减少资源的消耗。

注意：处于死锁状态的线程无法被中断

中断正常运行的线程

中断正常运行的线程, 不会清空中断状态

```
Thread t1 = new Thread(()->{
    while(true) {
        Thread current = Thread.currentThread();
        boolean interrupted = current.isInterrupted();
        if(interrupted) {
            log.debug(" 中断状态: {}", interrupted);
            break;
        }
    }
}, "t1");
t1.start();

//中断线程t1
t1.interrupt();
log.debug("中断状态：{}",t1.isInterrupted());
```

输出

```
20:45:17.596 [t1] DEBUG com.tuling.learnjuc.base.ThreadInterruptDemo -  中断状态: true
20:45:17.596 [main] DEBUG com.tuling.learnjuc.base.ThreadInterruptDemo - 中断状态：true
```

中断 sleep，wait，join 的线程

这几个方法都会让线w程进入阻塞状态，中断线程会清空中断状态。以 sleep 为例：

```
Thread t1 = new Thread(()->{
    while(true) {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}, "t1");
t1.start();

Thread.sleep(100);

//中断线程t1
t1.interrupt();
log.debug("中断状态：{}",t1.isInterrupted());
```

线程的调度机制

线程调度是指系统为线程分配CPU使用权的过程，主要调度方式有两种：

- **协同式线程调度(Cooperative Threads-Scheduling)**
- **抢占式线程调度(Preemptive Threads-Scheduling)**

使用协同式线程调度的多线程系统，线程执行的时间由线程本身来控制，线程把自己的工作执行完之后，要主动通知系统切换到另外一个线程上。使用协同式线程调度的最大好处是实现简单，由于线程要把自己的事情做完后才会通知系统进行线程切换，所以没有线程同步的问题，但是坏处也很明显，如果一个线程出了问题，则程序就会一直阻塞。

使用抢占式线程调度的多线程系统，每个线程执行的时间以及是否切换都由系统决定。在这种情况下，线程的执行时间不可控，所以不会有「一个线程导致整个进程阻塞」的问题出现。

Java线程调度就是抢占式调度，为什么？后面会分析。

在Java中，Thread.yield()可以让出CPU执行时间，但是对于获取执行时间，线程本身是没有办法的。对于获取CPU执行时间，线程唯一可以使用的手段是设置线程优先级，Java设置了10个级别的程序优先级，当两个线程同时处于Ready状态时，优先级越高的线程越容易被系统选择执行。

Java线程模型

为什么Java线程调度是抢占式调度？这需要我们了解Java中线程的实现模式。

我们已经知道线程其实是操作系统层面的实体，Java中的线程怎么和操作系统层面对应起来呢？

任何语言实现线程主要有三种方式：使用内核线程实现（1:1实现），使用用户线程实现(1:N实现），使用用户线程加轻量级进程混合实现(N:M实现）。

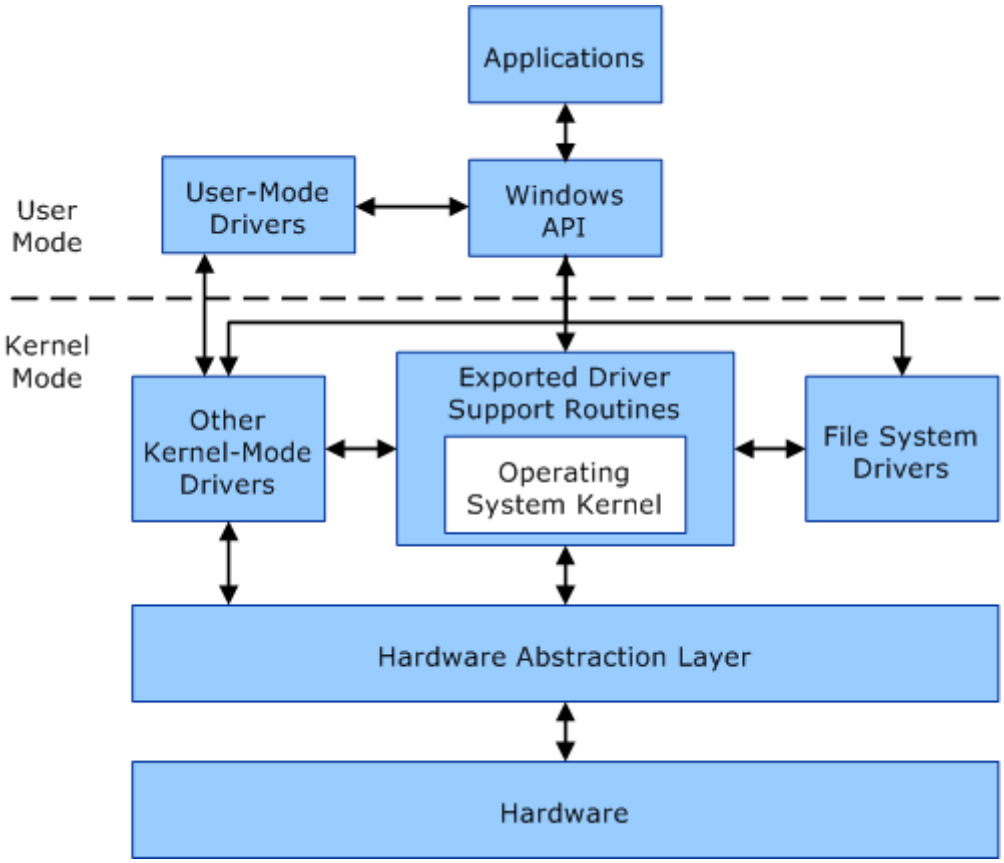
内核线程实现

使用内核线程实现的方式也被称为1： 1实现。 内核线程（Kernel-Level Thread， KLT） 就是直接由操作系统内核（Kernel， 下称内核） 支持的线程， 这种线程由内核来完成线程切换， 内核通过操纵调度器（Scheduler） 对线程进行调度， 并负责将线程的任务映射到各个处理器上。

由于内核线程的支持，每个线程都成为一个独立的调度单元，即使其中某一个在系统调用中被阻塞了，也不会影响整个进程继续工作，相关的调度工作也不需要额外考虑，已经由操作系统处理了。

局限性：首先，由于是基于内核线程实现的，所以各种线程操作，如创建、 析构及同步，都需要进行系统调用。而系统调用的代价相对较高， 需要在用户态（User Mode）和内核态（Kernel Mode）中来回切换。其次，每个语言层面的线程都需要有一个内核线程的支持，因此要消耗一定的内核资源（如内核线程的栈空间），因此一个系统支持的线程数量是有限的。

> 在现代操作系统中，CPU实际上都在两种截然不同的模式中花费时间:
>
> **Kernel Mode**
>
> 在内核模式下，执行代码可以完全且不受限制地访问底层硬件。它可以执行任何CPU指令和引用任何内存地址。内核模式通常为操作系统的最低级别、最受信任的功能保留。内核模式下的崩溃是灾难性的;他们会让整个电脑瘫痪。
>
> **User Mode**
>
> 在用户模式下，执行代码不能直接访问硬件或引用内存。在用户模式下运行的代码必须委托给系统api来访问硬件或内存。由于这种隔离提供的保护，用户模式下的崩溃总是可恢复的。在您的计算机上运行的大多数代码将在用户模式下执行。



Linux系统中线程实现方式

- LinuxThreads linux/glibc包在2.3.2之前只实现了LinuxThreads
- NPTL(Native POSIX Thread Library) 为POSIX(Portable Operating System Interface,可移植操作系统接口)标准线程库
  - POSIX 标准定义了一套线程操作相关的函数库pthread，用于让程序员更加方便地操作管理线程
  - pthread_create是类Unix操作系统（Unix、Linux、Mac OS X等）的创建线程的函数。

```
1 可以通过以下命令查看linux系统中使用哪种线程库实现
2 getconf GNU_LIBPTHREAD_VERSION
```

```
[root@node02 ~]# getconf GNU_LIBPTHREAD_VERSION
NPTL 2.17
```

用户线程实现

严格意义上的用户线程指的是完全建立在用户空间的线程库上，系统内核不能感知到用户线程的存在及如何实现的。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。 如果程序实现得当， 这种线程不需要切换到内核态， 因此操作可以是非常快速且低消耗的， 也能够支持规模更大的线程数量， 部分高性能数据库中的多线程就是由用户线程实现的。

用户线程的优势在于不需要系统内核支援，劣势也在于没有系统内核的支援， 所有的线程操作都需要由用户程序自己去处理。线程的创建、销毁、切换和调度都是用户必须考虑的问题，而且由于操作系统只把处理器资源分配到进程，那诸如“阻塞如何处理”“多处理器系统中如何将线程映射到其他处理器上”这类问题解决起来将会异常困难， 甚至有些是不可能实现的。 因为使用用户线程实现的程序通常都比较复杂，所以一般的应用程序都不倾向使用用户线程。Java语言曾经使用过用户线程，最终又放弃了。 但是近年来许多新的、以高并发为卖点的编程语言又普遍支持了用户线程，譬如Golang。

混合实现

线程除了依赖内核线程实现和完全由用户程序自己实现之外， 还有一种将内核线程与用户线程一起使用的实现方式， 被称为N：M实现。 在这种混合实现下， 既存在用户线程， 也存在内核线程。

用户线程还是完全建立在用户空间中， 因此用户线程的创建、 切换、 析构等操作依然廉价，并且可以支持大规模的用户线程并发。

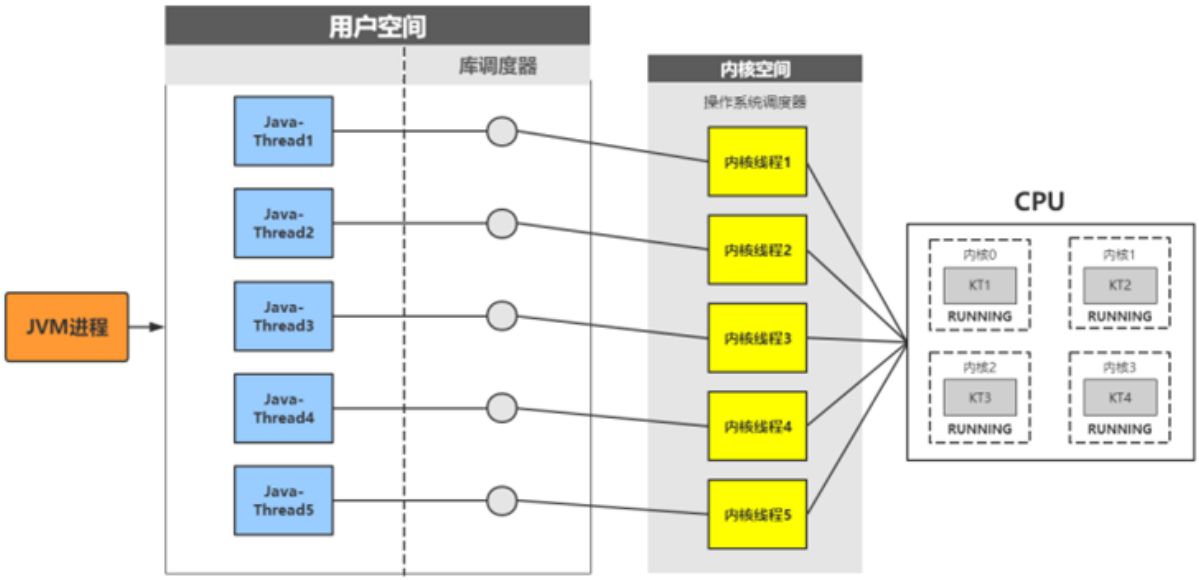
同样又可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过内核线程来完成。在这种混合模式中， 用户线程与轻量级进程的数量比是不定的，是N：M的关系。

Java线程的实现

Java线程在早期的Classic虚拟机上（JDK 1.2以前），是用户线程实现的， 但从JDK 1.3起，主流商用Java虚拟机的线程模型普遍都被替换为基于操作系统原生线程模型来实现，即采用1： 1的线程模型。

以HotSpot为例，它的每一个Java线程都是直接映射到一个操作系统原生线程来实现的，而且中间没有额外的间接结构， 所以HotSpot自己是不会去干涉线程调度的，全权交给底下的操作系统去处理。

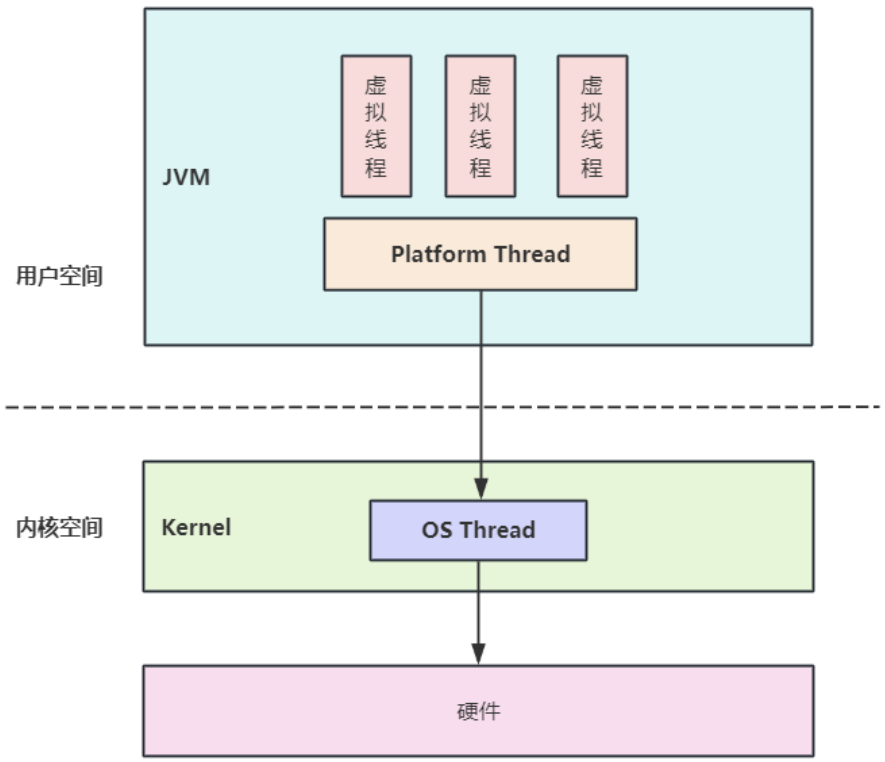
所以，这就是我们说Java线程调度是抢占式调度的原因。而且Java中的线程优先级是通过映射到操作系统的原生线程上实现的，所以线程的调度最终取决于操作系统，操作系统中线程的优先级有时并不能和Java中的一一对应，所以Java优先级并不是特别靠谱。



> **Thread#start()源码分析**
>
> https://www.processon.com/view/link/5f02ed9e6376891e81fec8d5

虚拟线程

在Java 21中，引入了虚拟线程（Virtual Threads），是一种用户级线程。虚拟线程是 Java 中的一种轻量级线程，它旨在解决传统线程模型中的一些限制，提供了更高效的并发处理能力，允许创建数千甚至数万个虚拟线程，而无需占用大量操作系统资源。



适用场景

- 虚拟线程适用于执行阻塞式任务，在阻塞期间，可以将CPU资源让渡给其他任务
- 虚拟线程不适合CPU密集计算或非阻塞任务，虚拟线程并不会运行的更快，而是增加了规模

- 虚拟线程是轻量级资源，用完即抛，不需要池化
- 通常我们不需要直接使用虚拟线程，像Tomcat、Jetty、 Netty、 Spring boot等都已支持虚拟线程

示例

```
public class VTDemo {

    public static void main(String[] args) throws InterruptedException {

        //平台线程
        Thread.ofPlatform().start(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread());
            }
        });

        //虚拟线程
        Thread vt = Thread.ofVirtual().start(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread());
            }
        });
        //等待虚拟线程打印完毕再退出主程序
        vt.join();

    }
}
```

输出

```
Thread[#22,Thread-0,5,main]
VirtualThread[#23]/runnable@ForkJoinPool-1-worker-1
```

线程间的通信

很多的时候，孤零零的一个线程工作并没有什么太多用处，更多的时候，我们是很多线程一起工作，而且是这些线程间进行通信，或者配合着完成某项工作，这就离不开线程间的通信和协调、协作。

管道输入输出流

Java的线程里有类似的管道机制，用于线程之间的数据传输，而传输的媒介为内存。

设想这么一个应用场景：通过 Java 应用生成文件，然后需要将文件上传到云端，我们一般的做法是，先将文件写入到本地磁盘，然后从文件磁盘读出来上传到云盘，但是通过Java中的管道输入输出流一步到位，则可以避免写入磁盘这一步。

Java中的管道输入/输出流主要包括了如下4种具体实现：PipedOutputStream、PipedInputStream、PipedReader和PipedWriter，前两种面向字节，而后两种面向字符。

```java
public class Piped {
    public static void main(String[] args) throws Exception {
        PipedWriter out = new PipedWriter();
        PipedReader in = new PipedReader();
        // 将输出流和输入流进行连接，否则在使用时会抛出IOException
        out.connect(in);

        Thread printThread = new Thread(new Print(in), "PrintThread");

        printThread.start();
        int receive = 0;
        try {
            while ((receive = System.in.read()) != -1) {
                out.write(receive);
            }
        } finally {
            out.close();
        }
    }

    static class Print implements Runnable {
        private PipedReader in;

        public Print(PipedReader in) {
            this.in = in;
        }

        @Override
        public void run() {
            int receive = 0;
            try {
                while ((receive = in.read()) != -1) {
                    System.out.print((char) receive);
                }
            } catch (IOException ex) {
            }
        }
    }
```

```
}
```

volatile，最轻量的通信/同步机制

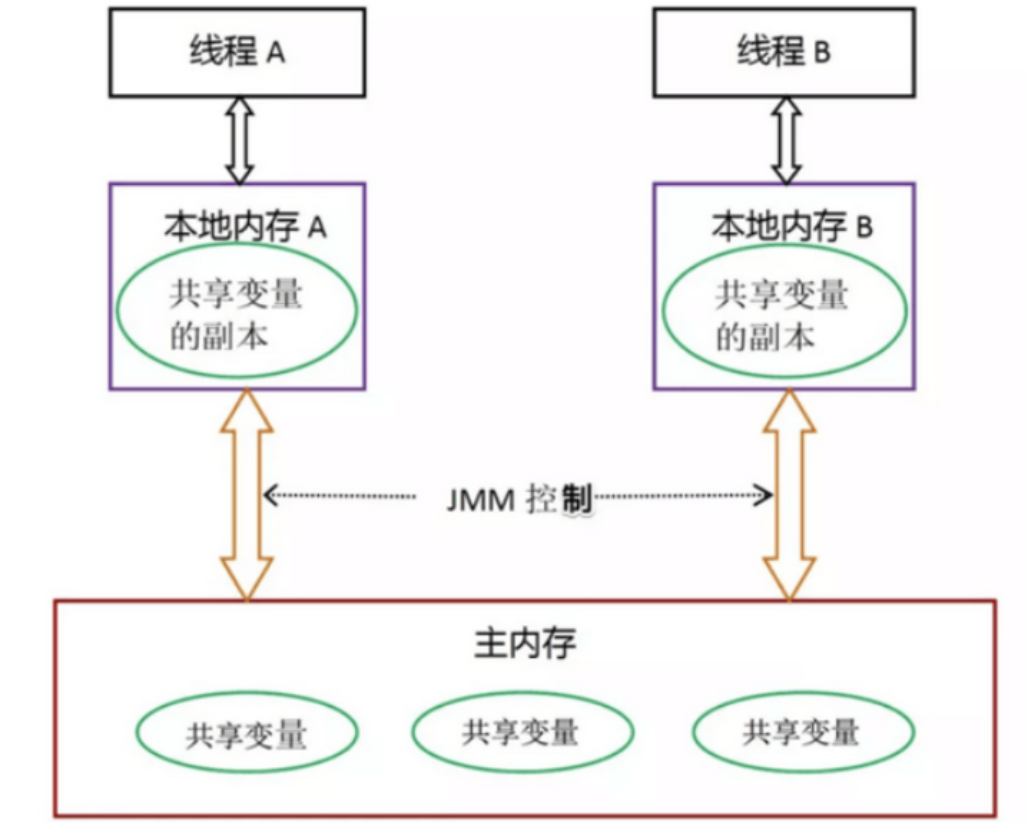
volatile保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

```
public class VolatileDemo {

    private static volatile  boolean stop = false;

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("线程t1开始执行");
                int i=0;
                while (!stop){
                    i++;
                }
                System.out.println("跳出循环");
            }
        },"t1");
        t1.start();

        Thread.sleep(1000);
        stop = true;
        System.out.println("主线程修改stop=true");

    }
}
```

不加volatile时，子线程无法感知主线程修改了stop的值，从而不会退出循环，而加了volatile后，子线程可以感知主线程修改了ready的值，迅速退出循环。

但是volatile不能保证数据在多个线程下同时写时的线程安全，volatile最适用的场景：一个线程写，多个线程读。

> Java线程之间的通信由Java内存模型（Java Memory Model，简称JMM）控制，JMM决定一个线程对共享变量的写入何时对另一个线程可见。根据JMM的规定，**线程对共享变量的所有操作都必须在自己的本地内存中进行，不能直接从主内存中读取。**JMM通过控制主内存与每个线程的本地内存之间的交互，来为Java程序提供内存可见性的保证。



Thread.join

join可以理解成是线程合并，当在一个线程调用另一个线程的join方法时，当前线程阻塞等待被调用join方法的线程执行完毕才能继续执行，所以join的好处能够保证线程的执行顺序，但是如果调用线程的join方法其实已经失去了并行的意义，虽然存在多个线程，但是本质上还是串行的，最后join的实现其实是基于等待通知机制的。

等待/通知机制

线程之间相互配合，完成某项工作，比如：一个线程修改了一个对象的值，而另一个线程感知到了变化，然后进行相应的操作，整个过程开始于一个线程，而最终执行又是另一个线程。前者是生产者，后者就是消费者，这种模式隔离了“做什么”（what）和“怎么做”（How），简单的办法是让消费者线程不断地循环检查变量是否符合预期在while循环中设置不满足的条件，如果条件满足则退出while循环，从而完成消费者的工作。却存在如下问题：

- 难以确保及时性。

- 难以降低开销。如果降低睡眠的时间，比如休眠1毫秒，这样消费者能更加迅速地发现条件变化，但是却可能消耗更多的处理器资源，造成了无端的浪费。

等待/通知机制则可以很好的避免上述的问题。

Object#wait/notify/notifyAll

等待通知机制可以基于对象的wait和notify方法来实现，在一个线程内调用该线程锁对象的wait方法，线程将进入等待队列进行等待直到被唤醒。

- notify()：通知一个在对象上等待的线程,使其从wait方法返回,而返回的前提是该线程获取到了对象的锁，没有获得锁的线程重新进入WAITING状态。
- notifyAll()：通知所有等待在该对象上的线程。尽可能用notifyAll()，谨慎使用notify()，因为notify()只会唤醒一个线程，我们无法确保被唤醒的这个线程一定就是我们需要唤醒的线程。
- wait(): 调用该方法的线程进入 WAITING状态,只有等待另外线程的通知或被中断才会返回.需要注意,调用wait()方法后,会释放对象的锁
- wait(long): 超时等待一段时间,这里的参数时间是毫秒,也就是等待长达n毫秒,如果没有通知就超时返回
- wait (long,int): 对于超时时间更细粒度的控制,可以达到纳秒

等待方遵循如下原则:

1）获取对象的锁。

2）如果条件不满足，那么调用对象的wait()方法，被通知后仍要检查条件。

3）条件满足则执行对应的逻辑。

```
synchronized(对象) {
    while (条件不满足) {
        对象.wait();
    }
    对应的逻辑;
}
```

通知方遵循如下原则。

1）获得对象的锁。

2）改变条件。

3）通知所有等待在对象上的线程。

```
synchronized(对象) {
    改变条件
    对象.notifyAll();
}
```

示例

```java
public class WaitDemo {

    public static void main(String[] args) throws InterruptedException {
        Object locker = new Object();

        Thread t1 = new Thread(() -> {
            try {
                System.out.println("wait开始");
                synchronized (locker) {
                    locker.wait();
                }
                System.out.println("wait结束");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        t1.start();

        //保证t1先启动，wait()先执行
        Thread.sleep(1000);

        Thread t2 = new Thread(() -> {
            synchronized (locker) {
                System.out.println("notify开始");
                locker.notifyAll();
                System.out.println("notify结束");
            }
        });

        t2.start();

    }
}
```

LockSupport#park/unpark

LockSupport是JDK中用来实现线程阻塞和唤醒的工具，线程调用park则等待“许可”，调用unpark则为指定线程提供“许可”。LockSupport很类似于二元信号量(只有1个许可证可供使用)，如果这个许可还没有被占用，当前线程获取许可并继续执行；如果许可已经被占用，当前线程阻塞，等待获取许可。使用它可以在任何场合使线程阻塞，可以指定任何线程进行唤醒，并且不用担心阻塞和唤醒操作的顺序，但要注意连续多次唤醒的效果和一次唤醒是一样的。

Java锁和同步器框架的核心AQS:AbstractQueuedSynchronizer，就是通过调用LockSupport.park()和LockSupport.unpark()实现线程的阻塞和唤醒的。

```
public class LockSupportDemo {

    public static void main(String[] args) throws InterruptedException {

        Thread parkThread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("ParkThread开始执行");
                // 当没有『许可』时，当前线程暂停运行；有『许可』时，用掉这个『许可』，当前线程恢复运行
                LockSupport.park();
                System.out.println("ParkThread执行完成");
            }
        });
        parkThread.start();

        Thread.sleep(1000);

        System.out.println("唤醒parkThread");
        // 给线程 parkThread 发放『许可』（多次连续调用 unpark 只会发放一个『许可』）
        LockSupport.unpark(parkThread);
    }

}
```

- LockSupport.park()和unpark()随时随地都可以调用。而wait和notify只能在synchronized代码段中调用
- LockSupport允许先调用unpark(Thread t)，后调用park()。如果thread1先调用unpark(thread2)，然后线程2后调用park()，线程2是不会阻塞的。如果线程1先调用notify，然后线程2再调用wait的话，线程2是会被阻塞的。