1. 线程基础知识
1.1 线程和进程
进程
线程
进程与线程的区别
进程间通信的方式
1.2 线程的同步互斥
1.3 上下文切换（Context switch）
通过命令查看CPU上下文切换情况
查看某一个线程\进程的上下文切换
1.4 操作系统层面线程生命周期
查看进程线程的方法
Linux系统中线程实现方式
2. Java线程详解
2.1 Java线程的实现方式
方式1：使用 Thread类或继承Thread类
方式2：实现 Runnable 接口配合Thread
方式3：使用有返回值的 Callable
方式4：使用 lambda
2.2 Java线程实现原理
Thread#start()源码分析
Java线程属于内核级线程

协程
2.3 Java线程的调度机制
协同式线程调度
抢占式线程调度
Java线程调度就是抢占式调度
2.4 Java线程的生命周期
2.5 Thread常用方法
sleep方法
yield方法
join方法
stop方法
2.5 Java线程的中断机制
API的使用
利用中断机制优雅的停止线程
sleep 期间能否感受到中断
2.6 Java线程间通信
volatile
等待唤醒机制
管道输入输出流
Thread.join

1. 线程基础知识

回顾：程序在计算机上是如何执行的?

1.1 线程和进程

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。在指令运行过程中

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 。

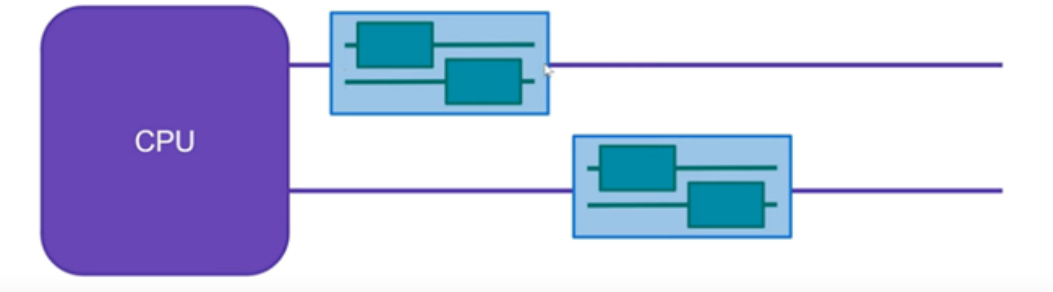
- 当一个程序被运行，从磁盘加载这个程序的代码至内存，这时就开启了一个进程。
- 进程就可以视为程序的一个实例。大部分程序可以同时运行多个实例进程（例如记事本、画图、浏览器 等），也有的程序只

能启动一个实例进程（例如网易云音乐、360 安全卫士等）。

  - 操作系统会以进程为单位，分配系统资源（CPU时间片、内存等资源），进程是资源分配的最小单位。

线程

- 线程是进程中的实体，一个进程可以拥有多个线程，一个线程必须有一个父进程。
- 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 。
- 线程，有时被称为轻量级进程(Lightweight Process，LWP），是操作系统调度（CPU调度）执行的最小单位。



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

进程间通信的方式

1. 管道（pipe）及有名管道（named pipe）：管道可用于具有亲缘关系的父子进程间的通信，有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。

2. 信号（signal）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一致的。

3. 消息队列（message queue）：消息队列是消息的链接表，它克服了上两种通信方式中信号量有限的缺点，具有写权限得进程可以按照一定得规则向消息队列中添加新信息；对消息队列有读权限得进程则可以从消息队列中读取信息。

4. 共享内存（shared memory）：可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。

5. 信号量（semaphore）：主要作为进程之间及同一种进程的不同线程之间得同步和互斥手段。
6. 套接字（socket）：这是一种更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

1.2 线程的同步互斥

线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。

线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

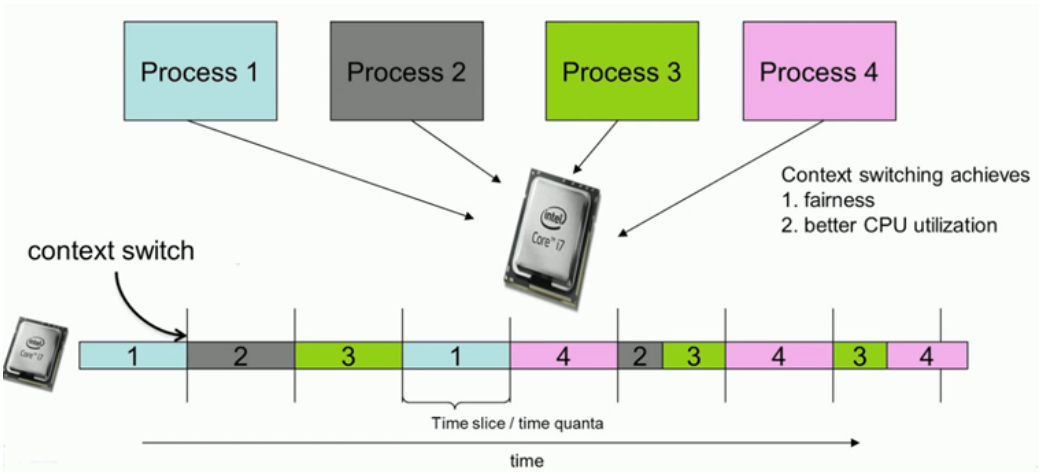
四种线程同步互斥的控制方法

- **临界区**:通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。（在一段时间内只允许一个线程访问的资源就称为临界资源）。
- **互斥量**:为协调共同对一个共享资源的单独访问而设计的。
- **信号量**:为控制一个具有有限数量用户资源而设计。
- **事件**:用来通知线程有一些事件已发生，从而启动后继任务的开始。

1.3 上下文切换（Context switch）

上下文切换是指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。

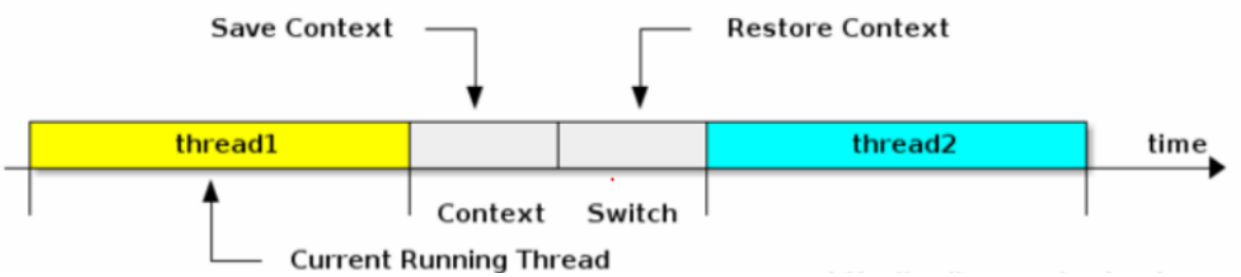
> 进程是程序的一个执行实例。在Linux中，线程是轻量级进程，可以并行运行，并与父进程(即创建线程的进程)共享一个地址空间和其他资源。
> 上下文是CPU寄存器和程序计数器在任何时间点的内容。
> 寄存器是CPU内部的一小部分非常快的内存(相对于CPU外部较慢的RAM主内存)，它通过提供对常用值的快速访问来加快计算机程序的执行。
> 程序计数器是一种专门的寄存器，它指示CPU在其指令序列中的位置，并保存着正在执行的指令的地址或下一条要执行的指令的地址，这取决于具体的系统。



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。



上下文切换只能在内核模式下发生。内核模式是CPU的特权模式，其中只有内核运行，并提供对所有内存位置和所有其他系统资源的访问。其他程序(包括应用程序)最初在用户模式下运行，但它们可以通过系统调用运行部分内核代码。

> 内核模式（Kernel Mode）vs 用户模式（User Mode）

上下文切换是多任务操作系统的一个基本特性。在多任务操作系统中，多个进程似乎同时在一个CPU上执行，彼此之间互不干扰。这种并发的错觉是通过快速连续发生的上下文切换(每秒数十次或数百次)来实现的。这些上下文切换发生的原因是进程自愿放弃它们在CPU中的时间，或者是调度器在进程耗尽其CPU时间片时进行切换的结果。

上下文切换通常是计算密集型的。就CPU时间而言，上下文切换对系统来说是一个巨大的成本，实际上，它可能是操作系统上成本最高的操作。因此，操作系统设计中的一个主要焦点是尽可能地避免不必要的上下文切换。与其他操作系统(包括一些其他类unix系统)相比，Linux的众多优势之一是它的上下文切换和模式切换成本极低。

通过命令查看CPU上下文切换情况

linux系统可以通过命令统计CPU上下文切换数据

```
1  #可以看到整个操作系统每1秒CPU上下文切换的统计
```

```
[root@node02 ~]# vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 8839868   1068 6579888    0    0     0     1    1    0  2  1 97  0  0
 1  0      0 8833396   1068 6579888    0    0     0    40 3232 5896  1  1 98  0  0
 0  0      0 8839356   1068 6579888    0    0     0     4 3085 5791  1  1 99  0  0
 0  0      0 8839356   1068 6579888    0    0     0     0 3094 5857  1  1 99  0  0
 0  0      0 8839480   1068 6579888    0    0     0     4 3334 5994  2  1 98  0  0
 0  0      0 8839356   1068 6579888    0    0     0     0 2989 5689  1  1 99  0  0
 0  0      0 8839232   1068 6579888    0    0     0    12 3273 5925  2  1 98  0  0
 0  0      0 8839356   1068 6579888    0    0     0    14 3118 5861  1  1 99  0  0
 0  0      0 8838736   1068 6579888    0    0     0     4 3311 5954  2  1 98  0  0
 0  0      0 8839480   1068 6579888    0    0     0     0 3154 5934  1  1 99  0  0
 0  0      0 8839388   1068 6579864    0    0     0     4 5430 7708  4  2 95  0  0
 0  0      0 8839140   1068 6579864    0    0     0     8 3038 5791  0  0 99  0  0
 0  0      0 8838768   1068 6579864    0    0     0     4 3183 5796  1  1 98  0  0
 0  0      0 8838768   1068 6579872    0    0     0     0 3353 6156  1  1 99  0  0
 0  0      0 8838668   1068 6579872    0    0     0     4 3321 5957  2  1 97  0  0
 0  0      0 8839016   1068 6579872    0    0     0     0 3020 5739  1  0 99  0  0
 0  0      0 8837976   1068 6579872    0    0     0    28 3255 5908  2  1 98  0  0
 0  0      0 8838504   1068 6579872    0    0     0     0 3612 6084  4  1 95  0  0
 0  0      0 8838580   1068 6579872    0    0     0     4 3310 5902  2  1 97  0  0
```

其中cs列就是CPU上下文切换的统计。当然，CPU上下文切换不等价于线程切换，很多操作会造成CPU上下文切换：

- 线程、进程切换
- 系统调用
- 中断

查看某一个线程\进程的上下文切换

- **使用pidstat命令**

> 常用的参数：
> -u 默认参数，显示各个进程的 CPU 统计信息
> -r 显示各个进程的内存使用情况
> -d 显示各个进程的 IO 使用
> -w 显示各个进程的上下文切换
> -p PID 指定 PID

```
1  # 显示进程5598每一秒的切换情况
```

```
05:01:04 PM   UID       PID   cswch/s nvcswch/s  Command
05:01:05 PM  1001      5598    484.00      0.00  java
05:01:06 PM  1001      5598    485.00      0.00  java
05:01:07 PM  1001      5598    484.00      0.00  java
05:01:08 PM  1001      5598    484.00      0.00  java
05:01:09 PM  1001      5598    484.00      0.00  java
05:01:10 PM  1001      5598    484.00      0.00  java
05:01:11 PM  1001      5598    485.00      0.00  java
05:01:12 PM  1001      5598    484.00      0.00  java
05:01:13 PM  1001      5598    484.00      0.00  java
05:01:14 PM  1001      5598    484.00      0.00  java
05:01:15 PM  1001      5598    485.00      0.00  java
```

其中cswch表示主动切换，nvcswch表示被动切换。从统计数据中看到，该进程每秒主动切换次数达到将近500次，因此代码中存在大量的 睡眠\唤醒 操作。

- **从进程的状态信息中查看**

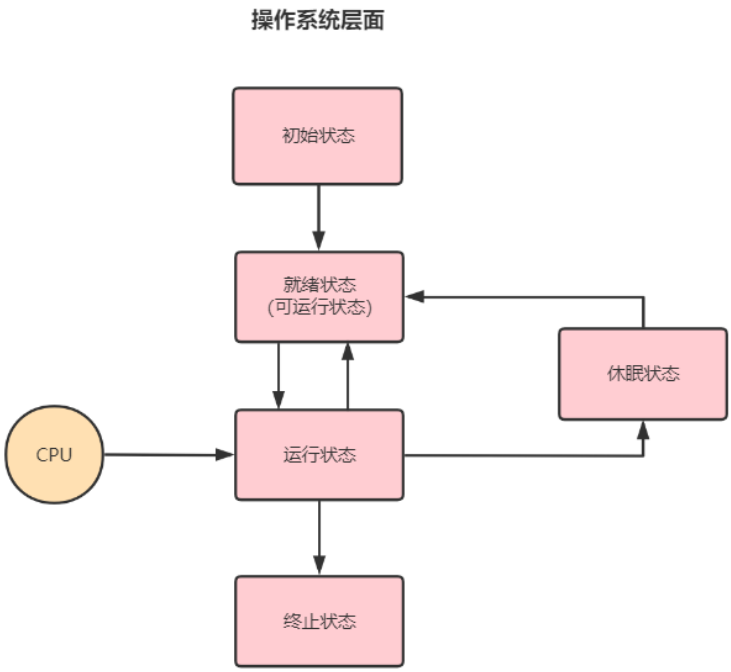
通过命令 cat /proc/5598/status 查看进程的状态信息

```
voluntary_ctxt_switches: 40469351
nonvoluntary_ctxt_switches: 2268
```

这2项就是该进程从启动到当前总的上下文切换情况。

1.4 操作系统层面线程生命周期

操作系统层面的线程生命周期基本上可以用下图这个“五态模型”来描述。这五态分别是：初始状态、可运行状态、运行状态、休眠状态和终止状态。



1. 初始状态，指的是线程已经被创建，但是还不允许分配 CPU 执行。这个状态属于编程语言特有的，不过这里所谓的被创建，仅仅是在编程语言层面被创建，而在操作系统层面，真正的线程还没有创建。
2. 可运行状态，指的是线程可以分配 CPU 执行。在这种状态下，真正的操作系统线程已经被成功创建了，所以可以分配 CPU 执行。
3. 当有空闲的 CPU 时，操作系统会将其分配给一个处于可运行状态的线程，被分配到 CPU 的线程的状态就转换成了运行状态。
4. 运行状态的线程如果调用一个阻塞的 API（例如以阻塞方式读文件）或者等待某个事件（例如条件变量），那么线程的状态就会转换到休眠状态，同时释放 CPU 使用权，休眠状态的线程永远没有机会获得 CPU 使用权。当等待的事件出现了，线程就会从休眠状态转换到可运行状态。
5. 线程执行完或者出现异常就会进入终止状态，终止状态的线程不会切换到其他任何状态，进入终止状态也就意味着线程的生命周期结束了。

这五种状态在不同编程语言里会有简化合并。例如，C 语言的 POSIX Threads 规范，就把初始状态和可运行状态合并了；Java 语言里则把可运行状态和运行状态合并了，这两个状态在操作系统调度层面有用，而 JVM 层面不关心这两个状态，因为 JVM 把线程调度交给操作系统处理了。

查看进程线程的方法

windows

- 任务管理器可以查看进程和线程数，也可以用来杀死进程
- tasklist 查看进程
- taskkill 杀死进程

linux

- ps -fe 查看所有进程
- ps -fT -p <PID> 查看某个进程（PID）的所有线程
- kill 杀死进程
- top 按大写 H 切换是否显示线程
- top -H -p <PID> 查看某个进程（PID）的所有线程

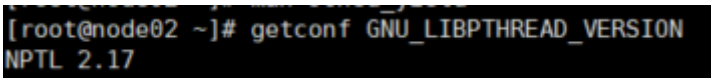
Java

- jps 命令查看所有 Java 进程
- jstack <PID> 查看某个 Java 进程（PID）的所有线程状态
- jconsole 来查看某个 Java 进程中线程的运行情况（图形界面）

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)

```
1 可以通过以下命令查看系统是使用哪种线程实现
```



2. Java线程详解

2.1 Java线程的实现方式

思考：Java中实现线程有几种方式?

方式1：使用 Thread类或继承Thread类

```
1 // 创建线程对象
2 Thread t = new Thread() {
3     public void run() {
4     // 要执行的任务
5     }
6 };
7 // 启动线程
```

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
```

方式3：使用有返回值的 Callable

```
class CallableTask implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        return new Random().nextInt();
    }
}
//创建线程池
ExecutorService service = Executors.newFixedThreadPool(10);
//提交任务，并用 Future提交返回结果
```

方式4：使用 lambda

```
new Thread(() -> System.out.println(Thread.currentThread().getName())).start();
```

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，调用Thread#start启动线程最终都会调用Thread#run方法

2.2 Java线程实现原理

思考：Java线程执行为什么不能直接调用run()方法,而要调用start()方法?

Thread#start()源码分析

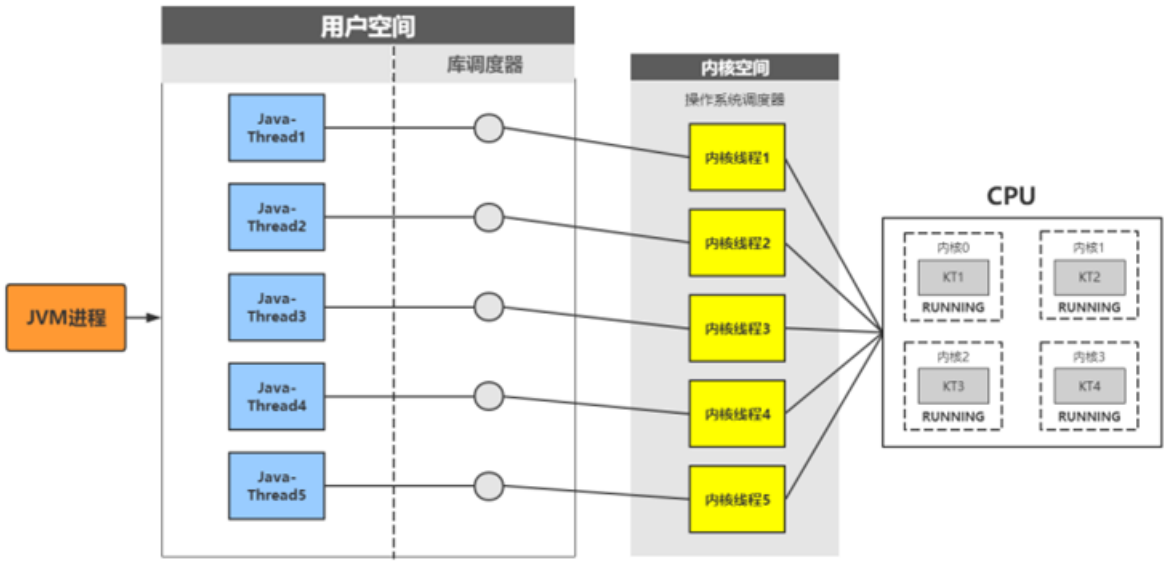
https://www.processon.com/view/link/5f02ed9e6376891e81fec8d5

Java线程属于内核级线程

JDK1.2——基于操作系统原生线程模型来实现。Sun JDK,它的Windows版本和Linux版本都使用一对一的线程模型实现，一条Java线程就映射到一条轻量级进程之中。

内核级线程（Kernel Level Thread，KLT）：它们是依赖于内核的，即无论是用户进程中的线程，还是系统进程中的线程，它们的创建、撤消、切换都由内核实现。

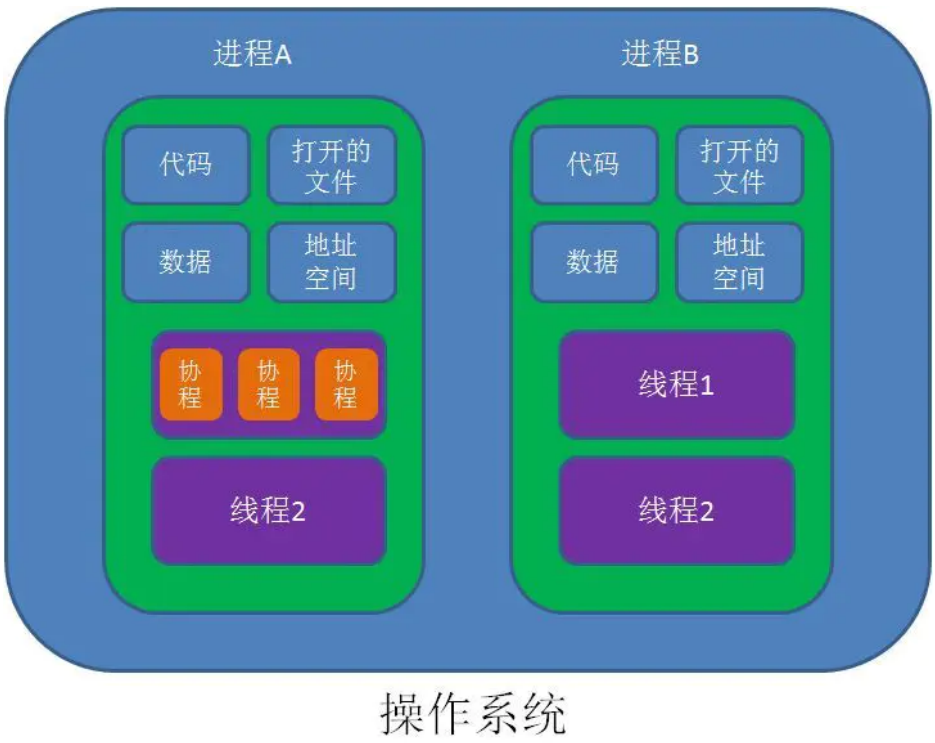
用户级线程（User Level Thread，ULT）：操作系统内核不知道应用线程的存在。



> java中是否存在协程？
> java中协程框架： kilim quasar

协程

协程，英文Coroutines, 是一种基于线程之上，但又比线程更加轻量级的存在，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行），具有对内核来说不可见的特性。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。



子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。而协程的调用和子程序不同。协程在子程序内部是可中断的，然后转而执行别的子程序，在适当的时候再返回来接着执行。

```
def A():
    print '1'
    print '2'
    print '3'
def B():
    print 'x'
    print 'y'
```

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：1 2 x y 3 z。

协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

- 线程的切换由操作系统调度，协程由用户自己进行调度，因此减少了上下文切换，提高了效率。
- 线程的默认stack大小是1M，而协程更轻量，接近1k。因此可以在相同的内存中开启更多的协程。
- 不需要多线程的锁机制：因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

注意： 协程适用于被阻塞的，且需要大量并发的场景（网络io）。不适合大量计算的场景。

2.3 Java线程的调度机制

线程调度是指系统为线程分配处理器使用权的过程，主要调度方式分两种，分别是协同式线程调度和抢占式线程调度

协同式线程调度

线程执行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。最大好处是实现简单，且切换操作对线程自己是可知的，没啥线程同步问题。坏处是线程执行时间不可控制，如果一个线程有问题，可能一直阻塞在那里。

抢占式线程调度

每个线程将由系统来分配执行时间，线程的切换不由线程本身来决定（Java中，Thread.yield()可以让出执行时间，但无法获取执行时间）。线程执行时间系统可控，也不会有一个线程导致整个进程阻塞。

Java线程调度就是抢占式调度

希望系统能给某些线程多分配一些时间，给一些线程少分配一些时间，可以通过设置线程优先级来完成。Java语言一共10个级别的线程优先级（Thread.MIN_PRIORITY至Thread.MAX_PRIORITY），在两线程同时处于ready状态时，优先级越高的线程越容易被系统选择执行。但优先级并不是很靠谱，因为Java线程是通过映射到系统的原生线程上来实现的，所以线程调度最终还是取决于操作系统。

```
public class SellTicketDemo implements Runnable {
    /**
     * 车票
```

```
     */
    private int ticket;

    public SellTicketDemo() {
        this.ticket = 1000;
    }

    @Override
    public void run() {
        while (ticket > 0) {
            synchronized (this) {
                if (ticket > 0) {
                    try {
                        // 线程进入暂时的休眠
                        Thread.sleep(2);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    // 获取到当前正在执行的程序的名称，打印余票
                    System.out.println(Thread.currentThread().getName()
                            + ":正在执行操作，余票:" + ticket--);
                }
            }
            Thread.yield();
        }
    }

    public static void main(String[] args) {
        SellTicketDemo demo = new SellTicketDemo();

        Thread thread1 = new Thread(demo,"thread1");
        Thread thread2 = new Thread(demo,"thread2");
        Thread thread3 = new Thread(demo,"thread3");
        Thread thread4 = new Thread(demo,"thread4");
        //priority优先级默认是5，最低1，最高10
        thread1.setPriority(Thread.MAX_PRIORITY);
        thread2.setPriority(Thread.MAX_PRIORITY);
        thread3.setPriority(Thread.MIN_PRIORITY);
        thread4.setPriority(Thread.MIN_PRIORITY);
        thread1.start();
```

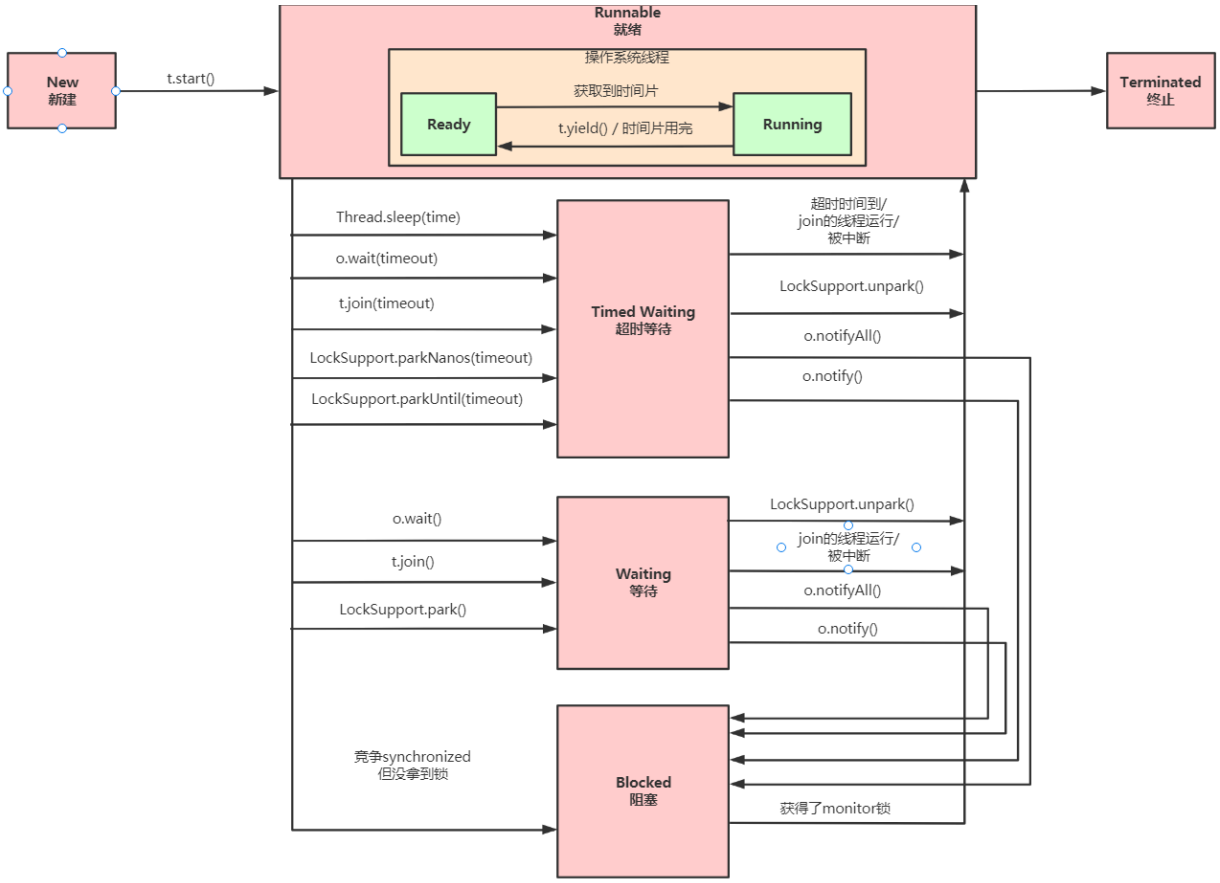
```
44          thread2.start();
45          thread3.start();
46          thread4.start();
47      }
```

2.4 Java线程的生命周期

Java 语言中线程共有六种状态，分别是：

1. NEW（初始化状态）
2. RUNNABLE（可运行状态+运行状态）
3. BLOCKED（阻塞状态）
4. WAITING（无时限等待）
5. TIMED_WAITING（有时限等待）
6. TERMINATED（终止状态）

在操作系统层面，Java 线程中的 BLOCKED、WAITING、TIMED_WAITING 是一种状态，即前面我们提到的休眠状态。也就是说只要 Java 线程处于这三种状态之一，那么这个线程就永远没有 CPU 的使用权。



从JavaThread的角度，JVM定义了一些针对Java Thread对象的状态（jvm.h）

```
/*
 * Java thread state support
 */
enum {
    JAVA_THREAD_STATE_NEW               = 0,
    JAVA_THREAD_STATE_RUNNABLE          = 1,
    JAVA_THREAD_STATE_BLOCKED           = 2,
    JAVA_THREAD_STATE_WAITING           = 3,
    JAVA_THREAD_STATE_TIMED_WAITING     = 4,
    JAVA_THREAD_STATE_TERMINATED        = 5,
    JAVA_THREAD_STATE_COUNT             = 6
};
```

从OSThread的角度，JVM还定义了一些线程状态给外部使用，比如用jstack输出的线程堆栈信息中线程的状态（osThread.hpp）

```
enum ThreadState {
  ALLOCATED,                    // Memory has been allocated but not initialized
  INITIALIZED,                  // The thread has been initialized but yet started
  RUNNABLE,                     // Has been started and is runnable, but not necessarily running
  MONITOR_WAIT,                 // Waiting on a contended monitor lock
  CONDVAR_WAIT,                 // Waiting on a condition variable
  OBJECT_WAIT,                  // Waiting on an Object.wait() call
  BREAKPOINTED,                 // Suspended at breakpoint
  SLEEPING,                     // Thread.sleep()
  ZOMBIE                        // All done, but not reclaimed yet
};
```

2.5 Thread常用方法

sleep方法

- 调用 sleep 会让当前线程从 *Running* 进入TIMED_WAITING状态，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，并且会清除中断标志
- 睡眠结束后的线程未必会立刻得到执行
- sleep当传入参数为0时，和yield相同

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

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。

```
public class ThreadJoinDemo {

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

        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("t begin");
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("t finished");
            }
        });
        long start = System.currentTimeMillis();
        t.start();
        //主线程等待线程t执行完成
        t.join();

        System.out.println("执行时间:" + (System.currentTimeMillis() - start));
        System.out.println("Main finished");
    }
```

思考：如何正确优雅的停止线程?

stop方法

stop()方法已经被jdk废弃，原因就是stop()方法太过于暴力，强行把执行到一半的线程终止。

```
public class ThreadStopDemo {

    private static Object lock = new Object();

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

        Thread thread = new Thread(new Runnable() {
```

```
8             @Override
9             public void run() {
10                synchronized (lock) {
11                    System.out.println(Thread.currentThread().getName() + "获取锁");
12                    try {
13                        Thread.sleep(60000);
14                    } catch (InterruptedException e) {
15                        e.printStackTrace();
16                    }
17                }
18                System.out.println(Thread.currentThread().getName() + "执行完成");
19            }
20        });
21        thread.start();
22        Thread.sleep(2000);
23        // 停止thread，并释放锁
24        thread.stop();
25
26        new Thread(new Runnable() {
27            @Override
28            public void run() {
29                System.out.println(Thread.currentThread().getName() + "等待获取锁");
30                synchronized (lock) {
31                    System.out.println(Thread.currentThread().getName() + "获取锁");
32                }
33            }
34        }).start();
35
36    }
```

stop会释放对象锁，可能会造成数据不一致。

2.5 Java线程的中断机制

Java没有提供一种安全、直接的方法来停止某个线程，而是提供了中断机制。中断机制是一种协作机制，也就是说通过中断并不能直接终止另一个线程，而需要被中断的线程自己处理。被中断的线程拥有完全的自主权，它既可以选择立即停止，也可以选择一段时间后停止，也可以选择压根不停止。

API的使用

- interrupt()： 将线程的中断标志位设置为true，不会停止线程

- isInterrupted(): 判断当前线程的中断标志位是否为true，不会清除中断标志位
- Thread.interrupted()：判断当前线程的中断标志位是否为true，并清除中断标志位，重置为fasle

```
public class ThreadInterruptTest {

    static int i = 0;

    public static void main(String[] args)  {
        System.out.println("begin");
        Thread t1 = new Thread(new Runnable() {
            @Override
            public  void run() {
                while (true) {
                    i++;
                    System.out.println(i);
                    //Thread.interrupted()  清除中断标志位
                    //Thread.currentThread().isInterrupted() 不会清除中断标志位
                    if (Thread.currentThread().isInterrupted() ) {
                        System.out.println("=========");
                    }
                    if(i==10){
                        break;
                    }

                }
            }
        });

        t1.start();
        //不会停止线程t1,只会设置一个中断标志位 flag=true
        t1.interrupt();
    }
```

利用中断机制优雅的停止线程

```
while (!Thread.currentThread().isInterrupted() && more work to do) {
    do more work
```

```
public class StopThread implements Runnable {
```

```
2
3       @Override
4       public void run() {
5           int count = 0;
6           while (!Thread.currentThread().isInterrupted() && count < 1000) {
7               System.out.println("count = " + count++);
8           }
9           System.out.println("线程停止:  stop thread");
10      }
11
12      public static void main(String[] args) throws InterruptedException {
13          Thread thread = new Thread(new StopThread());
14          thread.start();
15          Thread.sleep(5);
16          thread.interrupt();
17      }
```

注意：使用中断机制时一定要注意是否存在中断标志位被清除的情况

sleep 期间能否感受到中断

修改上面的代码，线程执行任务期间有休眠需求

```
1  @Override
2  public void run() {
3      int count = 0;
4      while (!Thread.currentThread().isInterrupted() && count < 1000) {
5          System.out.println("count = " + count++);
6
7          try {
8              Thread.sleep(1);
9          } catch (InterruptedException e) {
10             e.printStackTrace();
11         }
12     }
13     System.out.println("线程停止:  stop thread");
```

```
count = 998
count = 999
线程停止:  stop thread
```

处于休眠中的线程被中断，线程是可以感受到中断信号的，并且会抛出一个 InterruptedException 异常，同时清除中断信号，将中断标记位设置成 false。这样就会导致while条件Thread.currentThread().isInterrupted()为false，程序会在不满足count < 1000这个条件时退出。如果不在catch中重新手动添加中断信号，不做任何处理，就会屏蔽中断请求，有可能导致线程无法正确停止。

```
try {
    Thread.sleep(1);
} catch (InterruptedException e) {
    e.printStackTrace();
    //重新设置线程中断状态为true
    Thread.currentThread().interrupt();
```

sleep可以被中断 抛出中断异常：sleep interrupted， 清除中断标志位
wait可以被中断 抛出中断异常：InterruptedException， 清除中断标志位

2.6 Java线程间通信

volatile

volatile有两大特性，一是可见性，二是有序性，禁止指令重排序，其中可见性就是可以让线程之间进行通信。

```
public class VolatileDemo {

    private static volatile boolean flag = true;

    public static void main(String[] args) {

        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true){
                    if (flag){
                        System.out.println("trun on");
                        flag = false;
                    }
                }
            }
        }).start();
```

```
18
19          new Thread(new Runnable() {
20              @Override
21              public void run() {
22                  while (true){
23                      if (!flag){
24                          System.out.println("trun off");
25                          flag = true;
26                      }
27                  }
28              }
29          }).start();
30      }
```

等待唤醒(等待通知)机制

等待唤醒机制可以基于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。

```
1  public class WaitDemo {
2      private static Object lock = new Object();
3      private static  boolean flag = true;
4
5      public static void main(String[] args) {
6          new Thread(new Runnable() {
7              @Override
8              public void run() {
9                  synchronized (lock){
10                     while (flag){
11                         try {
12                             System.out.println("wait start .......");
13                             lock.wait();
14                         } catch (InterruptedException e) {
15                             e.printStackTrace();
16                         }
17                     }
18
19                     System.out.println("wait end ....... ");
20                 }
21             }
```

```
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                if (flag){
                    synchronized (lock){
                        if (flag){
                            lock.notify();
                            System.out.println("notify .......");
                            flag = false;
                        }

                    }
                }
            }
        }).start();
    }
```

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”。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。

```
public class LockSupportTest {

    public static void main(String[] args) {
        Thread parkThread = new Thread(new ParkThread());
        parkThread.start();

        System.out.println("唤醒parkThread");
        LockSupport.unpark(parkThread);
    }

    static class ParkThread implements Runnable{

        @Override
        public void run() {
            System.out.println("ParkThread开始执行");
```

```
            LockSupport.park();
            System.out.println("ParkThread执行完成");
        }
    }
```

管道输入输出流

管道输入/输出流和普通的文件输入/输出流或者网络输入/输出流不同之处在于，它主要用于线程之间的数据传输，而传输的媒介为内存。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。

```
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;
        }
```

```
27
28          @Override
29          public void run() {
30              int receive = 0;
31              try {
32                  while ((receive = in.read()) != -1) {
33                      System.out.print((char) receive);
34                  }
35              } catch (IOException ex) {
36              }
37          }
38      }
```

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。