# 多线程基础

# 第一章：进程和线程

### 【掌握】进程的介绍

是一个程序的运行状态和资源（内存、CPU）占用的描述

它是一个动态过程，它指的是从代码加载到执行完毕的一个完成的过程

1. 独立性：不同的进程之间独立，相互之间资源是不共享的
2. 动态性：进程在系统之中不是静止的，而是在系统之中一直处于活动状态
3. 并发性：多个进程在单个处理器上同时进行，并且不影响。

### 1.2【掌握】线程的介绍

线程是进程的一部分，一个进程可以有多个线程，每个线程去执行自己的一个特定的任务。

线程它抢占式，多个线程同时进行的时候，都去抢这个CPU的执行时间，抢不到CPU时间的线程就挂起。

### 1.3【掌握】进程和线程关系和区别

1. 一个程序运行后至少有一个进程
2. 一个进程是可以包含多个线程的，但是至少需要一个线程，否则是没有任何意义的。
3. 进程之间的资源不共享，但是线程之间资源**是共享**的。
4. 系统创建进程的时候，是需要给这个进程分配资源，但是创建线程的时候就容易得多。

## 第二章：多线程的实现（三种）

### 【掌握】2.1继承Thread类，覆写run方法，并且调用start方法，start方法千万不能忘记，否则这个线程相当于没启动。

|  |
| --- |
| **public class** NzThread **extends** Thread {  @Override  **public void** run() {  System.***err***.println(**"我是逆战的线程，开始跑咯"**);  } } |
| **public class** Test {  **public static void** main(String[] args) {  NzThread nzThread = **new** NzThread();*// 创建自定义的一个线程* nzThread.start();*// 调用start方法，让线程就绪* } } |

### 【掌握】2.2 实现Runnable接口，将Runnable接口传入Thread的构造方法，复写run方法，并且调用Thread的start方法。

|  |
| --- |
| **public static void** main(String[] args) {  Runnable runnable = **new** Runnable() {*// 创建一个Runnable的匿名实现类* @Override  **public void** run() {*// 复写Run方法* System.***err***.println(**"我是Runnable创建的线程"**);  }  };  Thread thread = **new** Thread(runnable);*// 把Runnable实例传入Thread的构造函数* thread.start();*// 调用start方法，让线程处于就绪状态，等待CPU调用。* }  *// 简洁版* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  System.***err***.println(**"简洁版Runnable实现多线程！"**);  } }).start(); |

### 【掌握】2.3 创建一个FutureTask,传入Callable接口，覆写call方法，并且调用FutureTask的run方法。最大的区别就是这种方式有返回值，可以知晓线程的运行状态，还有可以取消线程

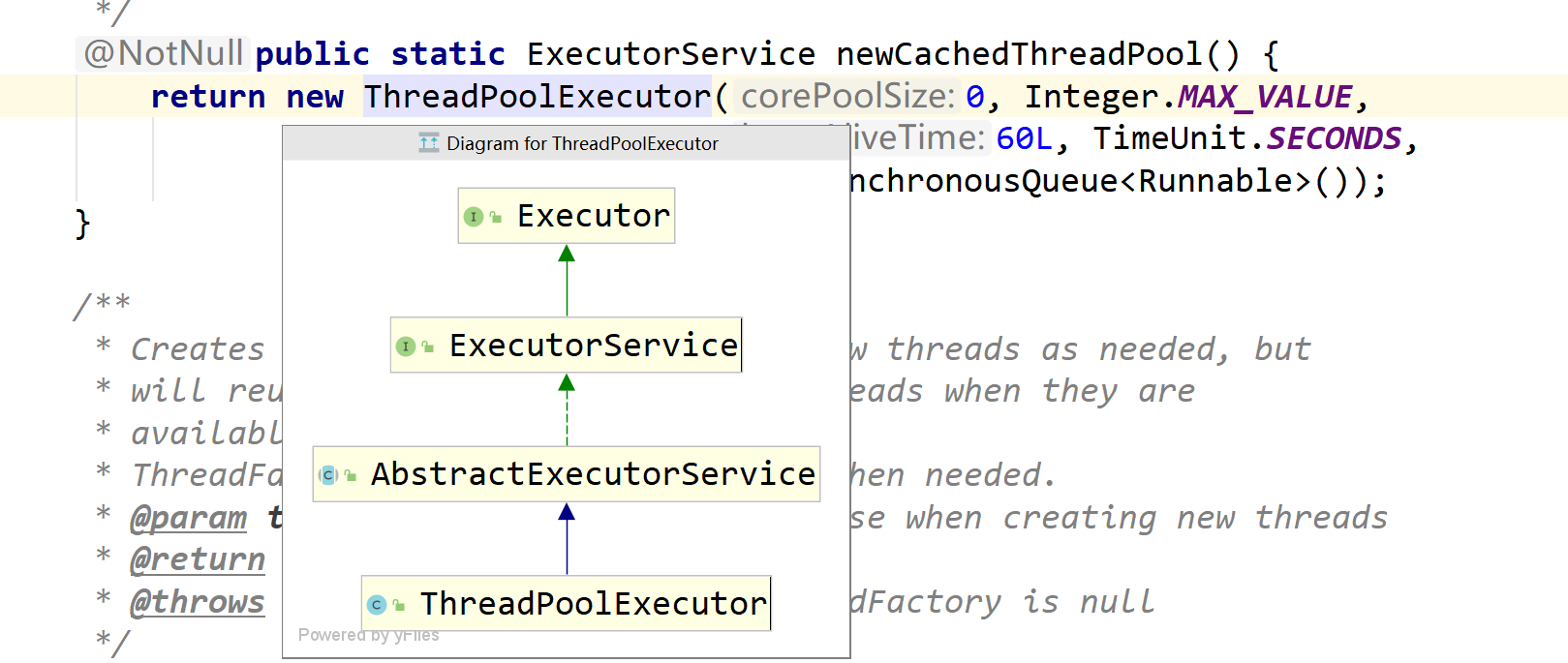
|  |
| --- |
| **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  FutureTask<String> futureTask = **new** FutureTask(**new** Callable() {  @Override  **public** String call() **throws** Exception {  System.***err***.println(**"这个线程是由实现Callable实现的！"**);  **return "call线程已经运行完毕，请老大发号司令！"**;*// call方法运行完毕之后，返回值* }  });  futureTask.run();*// 开启线程，让线程处于就绪状态！* String res = futureTask.get();*// 获取线程执行完毕之后的返回值* System.***err***.println(**"线程运行完毕之后的返回值是："**+res);*// 也就是说FutureTask是包含了线程运行完毕的一个状态* } |

**以上三个方法是基础入门版。实际工作之中，基本很少用到。**

**中级版：线程池创建方法**

|  |
| --- |
| **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  ExecutorService executorService = Executors.*newCachedThreadPool*();*// 创建一个缓存线程池，自动根据需要来扩充线程大小 // Executors.newFixedThreadPool(5);// 创建一个固定线程池大小的线程池 // Executors.newScheduledThreadPool(100);// 创建一个 包含100个线程的定时任务线程池 // Executors.newSingleThreadExecutor();// 创建单个线程的线程池 // Executors.newWorkStealingPool();// 根据工作状态，系统创建线程池* executorService.execute(**new** Runnable() {*// 执行一个线程，不用start方法，并且没有返回值* @Override  **public void** run() {  System.***err***.println(**"我是Executors创建出来的线程！"**);  }  });  Future<Object> submit = executorService.submit(**new** Callable<Object>() {  @Override  **public** String call() **throws** Exception {  System.***err***.println(**"我是submit方法执行的线程的内容"**);  **return "call线程运行完毕"**;  }  });  Object o = submit.get();*// 得到线程运行完毕的结果* System.***err***.println(o);*// 输出结果* } |

**高级版：以上操作，在阿里巴巴是禁止的！需要自定义线程池实现，自己创建ThreadPoolExecutor。**



**【面试题】:Runnable和Callable实现多线程的区别？**

1.Runnable没有返回值；Callable可以返回执行结果，是个泛型，和Future、FutureTask配合可以用来获取异步执行的结果

2.Callable接口的call()方法允许抛出异常；Runnable的run()方法异常只能在内部消化，不能往上继续抛

注：Callalbe接口支持返回执行结果，需要调用FutureTask.get()得到，此方法会阻塞主进程的继续往下执行，如果不调用不会阻塞。

## 第三章：【了解】线程常用方法

### 3.1设置线程的名称，通过setName方法实现

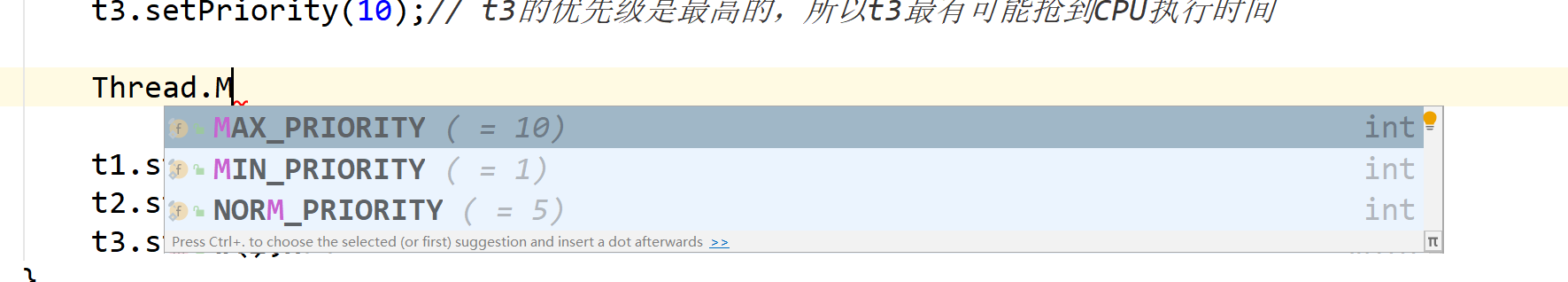
|  |
| --- |
| **public static void** main(String[] args) {*// main方法是主线程* NzThread nzThread = **new** NzThread();*// 创建自定义的一个线程* nzThread.setName(**"逆战线程"**);*// 设置线程的名称* nzThread.start();*// 调用start方法，让线程就绪* System.***err***.println(nzThread.getName());  System.***err***.println(**"主线程的名称："** + Thread.*currentThread*().getName()); } |
| Thread thread = **new** Thread(runnable, **"线程A"**);*// 把Runnable实例传入Thread的构造函数，把线程名称传入* thread.start();*// 调用start方法，让线程处于就绪状态，等待CPU调用。* System.***err***.println(thread.getName());*// 输出线程的名称* |

### 3.2 线程的休眠

|  |
| --- |
| **public static void** main(String[] args) {  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** x = 0; x < 100; x++) {  System.***err***.println(**"当前的线程是:"** + Thread.*currentThread*().getName() + **",x="** + x);  **try** {  Thread.*sleep*(1000);*// 让这个线程昏睡1秒* } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }, **"线程A"**).start();*// 开启线程* } |

### 3.3设置线程的优先级

多个线程一起去抢CPU执行时间的时候，是要看线程的优先级的。

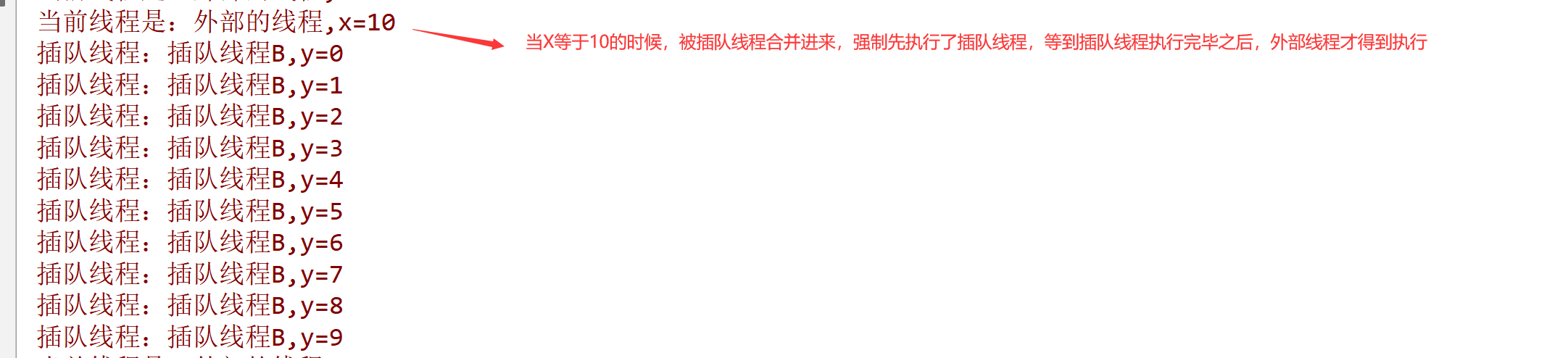


|  |
| --- |
| **public static void** main(String[] args) {  Runnable runnable = **new** Runnable() {  @Override  **public void** run() {  **for** (**int** x = 0; x < 100; x++) {  System.***err***.println(**"当前的线程是:"** + Thread.*currentThread*().getName() + **",x="** + x);  **try** {  Thread.*sleep*(1000);*// 让这个线程昏睡1秒* } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  };  Thread t1 = **new** Thread(runnable, **"线程A"**);*// 创建一个A线程* Thread t2 = **new** Thread(runnable, **"线程B"**);*// 创建一个B线程* Thread t3 = **new** Thread(runnable, **"线程C"**);*// 创建一个C线程* t1.setPriority(1);*// 线程的优先级是 （1-10）,线程优先级高的，并不是就一定先执行。* t2.setPriority(5);*// 优先级低的线程，还是始终会运行的，因为优先级高的线程运行完毕之后，就没线程跟它抢* t3.setPriority(10);*// t3的优先级是最高的，所以t3最有可能抢到CPU执行时间* t1.start();  t2.start();  t3.start();  System.***err***.println(**"主线程的优先级："** + Thread.*currentThread*().getPriority()); } |

### 3.4合并线程

有多个线程运行的时候，在线程中，把另一个线程Join进入到当前线程，那么加进来的那个线程就会强制先执行，被加入的那个线程就会等加进来的线程执行完毕之后，自己再执行。

|  |
| --- |
| **public static void** main(String[] args) {  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** x = 0; x < 100; x++) {  System.***err***.println(**"当前线程是："** + Thread.*currentThread*().getName() + **",x="** + x);  **if** (x == 10) {*// 当x等10的时候，我加入一个新的线程进来* Thread joinThread = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** y = 0; y < 10; y++) {  System.***err***.println(**"插队线程："** + Thread.*currentThread*().getName() + **",y="** + y);  }  }  }, **"插队线程B"**);  **try** {  joinThread.start();*// 开启强制线程* joinThread.join(); *// 强制加入* } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  }, **"外部的线程"**).start(); } |



### 3.5守护线程

有一种线程偷偷在后台进行着服务，就叫守护线程（Daemon Thread）,JVM垃圾回收就是这种线程.当前台线程（用户线程）全部都死亡（运行完毕），这个时候守护线程也就自动死亡。

|  |
| --- |
| **public static void** main(String[] args) {  Thread daemonThread = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** x = 0; x < 100; x++) {  System.***err***.println(**"当前线程是："** + Thread.*currentThread*().getName() + **",x="** + x);  **try** {  Thread.*sleep*(1000);*// 让线程昏睡1秒* } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }, **"守护线程"**);  daemonThread.setDaemon(**true**);*// 把这个线程设置为守护状态，也就是说把这个线程设置成守护线程* daemonThread.start();*// 跑起来这个线程* **for** (**int** x = 0; x < 100; x++) {  System.***err***.println(**"当前线程是："** + Thread.*currentThread*().getName() + **",x="** + x);  } } |

### 3.6线程的礼让

男女同学去食堂打饭，来了个美女，那么男同学就会主动让美女先去打饭，这种就叫礼让。

|  |
| --- |
| **public static void** main(String[] args) {  Runnable runnable = **new** Runnable() {  @Override  **public void** run() {  **for** (**int** x = 0; x < 100; x++) {  System.***err***.println(**"男同学线程是："** + Thread.*currentThread*().getName() + **",x="** + x);  **if** (x % 3 == 0) {*// 当x是3的倍数的时候，就做出让步* Thread.*yield*();*// 做出让步，其实就是让当前线程暂停一下，CPU重新进行选择线程执行。  // 让步不能等于阻塞,如果这个线程让步的话，它会立刻进入就绪状态，等待CPU执行时间的到来。* }  }  }  };  Thread boy = **new** Thread(runnable, **"男同学线程"**);   Thread girl = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** x = 0; x < 100; x++) {  System.***err***.println(**"女同学线程是："** + Thread.*currentThread*().getName() + **",x="** + x);  }  }  }, **"女同学的线程"**);  boy.start(); *// 开启男生的线程* girl.start();*// 开启女生的线程* } |

线程的礼让是暂停一下，下次CPU还是会有机会选到这个线程执行，线程此时是处于**就绪状态**的。

## 第四章：线程的生命周期

4.1【掌握】线程的生命周期

（新生）当线程new出来的时候，叫新生，线程被实例化，但是还没有开始执行。

（就绪）当条用start方法之后，那么线程就处于**就绪状态，**等待CPU执行时间，此时这个线程还是没有跑起来的。

（运行）当线程等啊等，等到CPU执行时间了，那么此时线程就跑起来了，CPU就开始处理这个线程中任务。

（阻塞）当CPU时间执行完了，CPU撤走，那么此时这个线程就**阻塞，**此时别的线程就有获取执行的机会了，被阻塞的线在合适的时机又会进入就绪状态。

（死亡）当所有线程都正常运行完毕，线程就终止了，也叫死亡了，还有另一个办法，也可以让线程死亡或者终止，stop方法。

流程图：