# 一、多线程

## 1. 进程与线程的基本概念

* 程序：用某种计算机语言编写的可执行代码。
* 进程：进程就是**应用程序在内存中分配的空间，也就是正在运行的程序**。

**进程让操作系统的并发性 成为了可能，⽽线程让进程的内部并发成为了可能。**

### 1.1 使用多进程或多线程

多进程的⽅式也可以实现并发，为什么我们要使⽤多线程？

* 进程间的通信⽐较复杂，⽽线程间的通信⽐较简单，通常情况下，我们需要使⽤共享资源，这些资源在线程间的通信⽐较容易。
* 进程是重量级的，⽽线程是轻量级的，故多线程⽅式的系统开销更⼩。

### 1.2 进程和线程的区别

本质的 区别是**是否单独占有内存地址空间及其它系统资源**

* 进程占有一定的内存空间，进程间内存隔离，数据共享比较复杂，数据同步简单；线程共享所属进程的内存资源，数据共享简单，数据同步复杂；
* 进程有内存隔离，相对独立，一个进程出现问题不影响其他进程，可靠性高；线程共享内存，线程间耦合度高，一个线程崩溃可能影响整个进程的运行，可靠性低；
* 进程占有内存空间，进程的创建和销毁需要保存**寄存器、栈信息**，还需要**资源分配和调度**，开销大；线程的创建只需要保存寄存器和栈信息。

**进程是操作系统进行资源分配的基本单位，而线程是操作系统进行调度的基本单位。**

### 1.3 上下文切换

上下文是指**某一时间点CPU寄存器和程序计数器的内容**。

CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后就会切换到下一任务，在切换前，会保存上一个任务的 状态，以便切换回时，加载这个任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。

上下⽂切换通常是计算密集型的，意味着此操作会消耗⼤量的 CPU 时间，**故线程 也不是越多越好**。

## 2. Java多线程类和接口

### 2.1 Thread类和Runnable接口

创建线程的方法：

* 继承Thread类，并重写run方法；
* 实现Runnable接口的run方法；

#### 2.1.1 继承Thread类

public class Demo {  
 public static class MyThread extends Thread {  
 @Override  
 public void run() {  
 System.out.println("MyThread");  
 }  
 }  
 public static void main(String[] args) {  
 Thread myThread = new MyThread();  
 myThread.start();  
 }  
}

要调用start方法，线程才算启动。

我们在程序⾥⾯调⽤了start()⽅法后，**虚拟机会先为我们创建⼀个线程，然 后等到这个线程第⼀次得到时间⽚时再调⽤run()⽅法**。 注意不可多次调⽤start()⽅法。在第⼀次调⽤start()⽅法后，再次调⽤start()⽅法会抛出异常。

#### 2.1.2 实现Runnable接口

可以看到 Runnable 是⼀个函数式接⼝，这意味着我们可以使⽤**Java 8**的函数式编程来简化代码。

public class Demo {  
 public static class MyThread implements Runnable {  
 @Override  
 public void run() {  
 System.out.println("MyThread");  
 }  
 }  
 public static void main(String[] args) {  
 new MyThread().start();  
 // Java 8 函数式编程，可以省略MyThread类  
 new Thread(() -> {  
 System.out.println("Java 8 匿名内部类");  
 }).start();  
 }  
}

#### **2.1.3 Thread**类构造⽅法

Thread类是通过在构造方法中调用一个私有的**init**方法实现初始化的。

// ⽚段1 - init⽅法  
private void init(ThreadGroup g, Runnable target, String name,  
 long stackSize, AccessControlContext acc,boolean inheritThreadLocals)  
   
// ⽚段2 - 构造函数调⽤init⽅法  
public Thread(Runnable target) {  
 init(null, target, "Thread-" + nextThreadNum(), 0);  
}  
  
// ⽚段3 - 使⽤在init⽅法⾥初始化AccessControlContext类型的私有属性  
this.inheritedAccessControlContext =   
 acc != null ? acc : AccessController.getContext();  
  
// ⽚段4 - 两个对⽤于⽀持ThreadLocal的私有属性  
ThreadLocal.ThreadLocalMap threadLocals = null;  
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;

* g：线程组，指定该线程在哪个组下；
* target：指定要执行的任务；
* name：线程的名字；(多个线程的名字是可以重复的)
* acc：⽤于初始化私有变量 inheritedAccessControlContext
* inheritThreadLocals：可继承的 ThreadLocal ，Thread 类⾥⾯有两
* 个私有属性来⽀持 ThreadLocal ，

#### 2.1.4 Thread类的常⽤⽅法

* currentThread()：静态⽅法，返回对当前正在执⾏的线程对象的引⽤；
* start()：开始执⾏线程的⽅法，java虚拟机会调⽤线程内的run()⽅法；
* yield()：指的是当前线程愿意让出对当前处理器的占⽤。就算当前线程调⽤了yield()⽅法，程序在调度的时候，也还有可能继续运⾏这个线程；
* sleep()：静态⽅法，使当前线程睡眠⼀段时间；
* join()：使当前线程等待另⼀个线程执⾏完毕之后再继续执⾏，内部调⽤的是Object类的wait⽅法实现的；

#### 2.1.5 Thread类与Runnable接⼝的⽐较

* 由于Java“单继承，多实现”的特性，Runnable接⼝使⽤起来⽐Thread更灵活。
* Runnable接⼝出现更符合⾯向对象，将线程单独进⾏对象的封装。
* Runnable接⼝出现，降低了线程对象和线程任务的耦合性。
* 如果使⽤线程时不需要使⽤Thread类的诸多⽅法，显然使⽤Runnable接⼝更为轻量。

### 2.2 Callable、Future与FutureTask

#### 2.2.1 Callable接⼝

Callable 与 Runnable 类似，同样是只有⼀个抽象⽅法的函数式接⼝。不同的 是， Callable 提供的⽅法是有返回值的，⽽且⽀持泛型。

@FunctionalInterface  
public interface Callable<V> {  
 V call() throws Exception;   
}

ExecutorService 可以使⽤ submit ⽅法来让⼀个 Callable 接⼝执⾏。它会返回⼀个 Future ，我们后续的程序可以通过这个 Future 的 get ⽅法得到结果。

例子：

// ⾃定义Callable  
class Task implements Callable<Integer>{  
 @Override  
 public Integer call() throws Exception {  
 // 模拟计算需要⼀秒  
 Thread.sleep(1000);  
 return 2;  
 }  
 public static void main(String args[]){  
 // 使⽤  
 ExecutorService executor = Executors.newCachedThreadPool();  
 Task task = new Task();  
 Future<Integer> result = executor.submit(task);  
 // 注意调⽤get⽅法会阻塞当前线程，直到得到结果。  
 // 所以实际编码中建议使⽤可以设置超时时间的重载get⽅法。  
 System.out.println(result.get());   
 }  
}

#### 2.2.2 Future接⼝

public abstract interface Future<V> {  
 public abstract boolean cancel(boolean paramBoolean);  
 public abstract boolean isCancelled();  
 public abstract boolean isDone();  
 public abstract V get() throws InterruptedException, ExecutionException;  
 public abstract V get(long paramLong, TimeUnit paramTimeUnit)  
 throws InterruptedException, ExecutionException, TimeoutException;   
}

cancel ⽅法是试图取消⼀个线程的执⾏。 注意是试图取消，并不⼀定能取消成功。因为任务可能已完成、已取消、或者⼀些 其它因素不能取消，存在取消失败的可能。 boolean 类型的返回值是“是否取消成 功”的意思。参数 paramBoolean 表示是否采⽤中断的⽅式取消线程执⾏。为了让任务有能够取消的功能，就使⽤ Callable 来代替 Runnable 。

#### 2.2.3 FutureTask类

FutureTask是RunnableFuture接口的实现类，RunnableFuture接口同时继承了Runnable接口和Future接口。

public interface RunnableFuture<V> extends Runnable, Future<V> {  
 void run();  
}

#### 2.2.4 FutureTask的⼏个状态

/\*\*  
 \* state可能的状态转变路径如下：  
 \* NEW -> COMPLETING -> NORMAL  
 \* NEW -> COMPLETING -> EXCEPTIONAL  
 \* NEW -> CANCELLED  
 \* NEW -> INTERRUPTING -> INTERRUPTED  
 \*/  
 private volatile int state;  
 private static final int NEW = 0;  
 private static final int COMPLETING = 1;  
 private static final int NORMAL = 2;  
 private static final int EXCEPTIONAL = 3;  
 private static final int CANCELLED = 4;  
 private static final int INTERRUPTING = 5;  
 private static final int INTERRUPTED = 6;

state表示任务的运⾏状态，初始状态为NEW。运⾏状态只会在set、  
setException、cancel⽅法中终⽌。COMPLETING、INTERRUPTING是任 务完成后的瞬时状态。

## 3. 线程组和线程优先级

### 3.1 线程组(ThreadGroup)

每个Thread必 然存在于⼀个ThreadGroup中，Thread不能独⽴于ThreadGroup存在。如果在new Thread时没有显式指定，那么默认将⽗线程 （当前执⾏new Thread的线程）线程组设置为⾃⼰的线程组。

ThreadGroup是⼀个标准的向下引⽤的树状 结构，这样设计的原因是**防⽌"上级"线程被"下级"线程引⽤⽽⽆法有效地被GC回收**。

### 3.2 线程的优先级

* Java默认的线程优先级为5，线程的执⾏顺序由调度程序来决定，线程的优先级会 在线程被调⽤之前设定。
* Java程序中对线程所设置的优先级只是给 操作系统⼀个建议，操作系统不⼀定会采纳。⽽真正的调⽤顺序，是由操作系统的 线程调度算法决定的。
* 如果某线程是守护线程，那如果所有的⾮守护线程结束，这个守护线程也会⾃动结束。⼀个线程默认是⾮守护线程，可以通过Thread类的setDaemon(boolean on)来设置。
* 如果某个线程优先级⼤于线程所在线程组的最⼤优先级，那么该线程的优先 级将会失效，取⽽代之的是线程组的最⼤优先级。

### 3.3 线程组的常⽤⽅法及数据结构

* 获取当前的线程组名字：
* Thread.currentThread().getThreadGroup().getName()
* 复制线程组
* Thread[] threads = new Thread[threadGroup.activeCount()];  
  TheadGroup threadGroup = new ThreadGroup();  
  threadGroup.enumerate(threads);
* 线程组统⼀异常处理
* package com.func.axc.threadgroup;  
  public class ThreadGroupDemo {  
   public static void main(String[] args) {  
   ThreadGroup threadGroup1 = new ThreadGroup("group1") {  
   // 继承ThreadGroup并重新定义以下⽅法  
   // 在线程成员抛出unchecked exception  
   // 会执⾏此⽅法  
   public void uncaughtException(Thread t, Throwable e) {  
   System.out.println(t.getName() + ": " + e.getMessage());  
   }  
   };  
   // 这个线程是threadGroup1的⼀员  
   Thread thread1 = new Thread(threadGroup1, new Runnable() {  
   public void run() {  
   // 抛出unchecked异常  
   throw new RuntimeException("测试异常");  
   }  
   });  
   thread1.start();  
   }  
  }

#### 线程组的数据结构

线程组还可以包含其他的线程组，不仅仅是线程。

ThreadGroup 源码：

public class ThreadGroup implements Thread.UncaughtExceptionHandler {  
 private final ThreadGroup parent; // ⽗亲ThreadGroup  
 String name; // ThreadGroupr 的名称  
 int maxPriority; // 线程最⼤优先级  
 boolean destroyed; // 是否被销毁  
 boolean daemon; // 是否守护线程  
 boolean vmAllowSuspension; // 是否可以中断  
 int nUnstartedThreads = 0; // 还未启动的线程  
 int nthreads; // ThreadGroup中线程数⽬  
 Thread threads[]; // ThreadGroup中的线程  
 int ngroups; // 线程组数⽬  
 ThreadGroup groups[]; // 线程组数组   
}

// 私有构造函数  
private ThreadGroup() {   
 this.name = "system";  
 this.maxPriority = Thread.MAX\_PRIORITY;  
 this.parent = null;   
}  
// 默认是以当前ThreadGroup传⼊作为parent ThreadGroup，新线程组的⽗线程组是⽬前正在运⾏线  
public ThreadGroup(String name) {  
 this(Thread.currentThread().getThreadGroup(), name);  
}  
// 构造函数  
public ThreadGroup(ThreadGroup parent, String name) {  
 this(checkParentAccess(parent), parent, name);  
}  
// 私有构造函数，主要的构造函数  
private ThreadGroup(Void unused, ThreadGroup parent, String name) {  
 this.name = name;  
 this.maxPriority = parent.maxPriority;  
 this.daemon = parent.daemon;  
 this.vmAllowSuspension = parent.vmAllowSuspension;  
 this.parent = parent;  
 parent.add(this);  
}

第三个构造函数⾥调⽤了 checkParentAccess ⽅法，这⾥看看这个⽅法的源码：  
// 检查parent ThreadGroup  
private static Void checkParentAccess(ThreadGroup parent) {  
 parent.checkAccess();  
 return null;   
}  
// 判断当前运⾏的线程是否具有修改线程组的权限  
public final void checkAccess() {  
 SecurityManager security = System.getSecurityManager();  
 if (security != null) {  
 security.checkAccess(this);  
 }  
 }

这⾥涉及到 SecurityManager 这个类，它是Java的安全管理器，它允许应⽤ 程序在执⾏⼀个可能不安全或敏感的操作前确定该操作是什么，以及是否是 在允许执⾏该操作的安全上下⽂中执⾏它。应⽤程序可以允许或不允许该操 作。⽐如引⼊了第三⽅类库，但是并不能保证它的安全性。 其实Thread类也有⼀个checkAccess()⽅法，不过是⽤来当前运⾏的线程是 否有权限修改被调⽤的这个线程实例。（Determines if the currently running thread has permission to modify this thread.）

## 4. Java线程状态及主要转化⽅法

操作系统线程主要有三种状态：

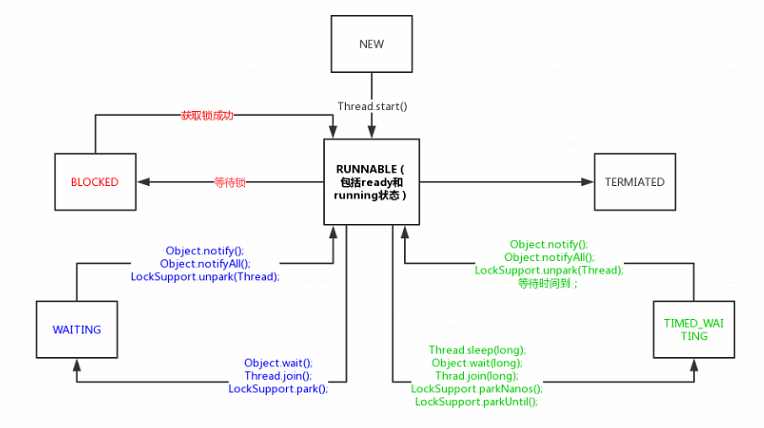
* 就绪状态：线程正在等待使⽤CPU，经调度程序调⽤之后可进⼊running状态。
* 执⾏状态(running)：线程正在使⽤CPU。
* 等待状态(waiting): 线程经过等待事件的调⽤或者正在等待其他资源（如I/O）。

### 4.1 Java线程的6个状态

// Thread.State 源码  
public enum State {  
 NEW,  
 RUNNABLE,  
 BLOCKED,  
 WAITING,  
 TIMED\_WAITING,  
 TERMINATED;  
}

* new：创建态；线程已经创建，但还没有调用start()方法，未启动。
  + 不能重复调用同一个线程的start()方法
  + 线程执行完毕（处于TERMINATED状态），不能调用start()方法
* runable：运行态；当前线程正在运行中，出于此状态的线程在JVM中运行，也可能在等待系统资源。Java线程的**RUNNABLE**状态其实是包括了传统操作系统线程的**ready**和**running**两个状态的。
* blocked：阻塞态；处于BLOCKED状态的线程正等待锁的释放以进⼊同步区。
* waiting：等待态；处于等待状态的线程变成RUNNABLE状态需要其他线程唤醒。
* 调⽤如下3个⽅法会使线程进⼊等待状态：  
  Object.wait()：使当前线程处于等待状态直到另⼀个线程唤醒它；  
  Thread.join()：等待线程执⾏完毕，底层调⽤的是Object实例的wait⽅法；  
  LockSupport.park()：除⾮获得调⽤许可，否则禁⽤当前线程进⾏线程调度。
* time\_waiting：超时等待态；线程等待一段具体时间，时间到后会被自动唤醒。
* **调⽤如下⽅法会使线程进⼊超时等待状态**：  
  Thread.sleep(long millis)：使当前线程睡眠指定时间；  
  Object.wait(long timeout)：线程休眠指定时间，等待期间可以通过notify()/notifyAll()唤醒；  
  Thread.join(long millis)：等待当前线程最多执⾏millis毫秒，如果millis为0，则 会⼀直执⾏；  
  LockSupport.parkNanos(long nanos)： 除⾮获得调⽤许可，否则禁⽤当前线 程进⾏线程调度指定时间；  
  LockSupport.parkUntil(long deadline)：同上，也是禁⽌线程进⾏调度指定时间
* terminated：终止态；此时线程已经执行完毕。

### 4.2 线程状态的转换



#### waiting状态与runnable状态的转换

有3个⽅法可以使线程从RUNNABLE状态转为WAITING状态：**Object.wait()**和**Thread.join()**。 **Object.wait()**。

* 调用waiting方法前线程必须持有对象的锁；调用waiting方法会释放当前的锁，直到有其他线程调用notify()或notifyAll()方法唤醒等待锁的线程。
* 需要注意的是，其他线程调⽤notify()⽅法只会唤醒单个等待锁的线程，如有 有多个线程都在等待这个锁的话不⼀定会唤醒到之前调⽤wait()⽅法的线程。
* 同样，调⽤notifyAll()⽅法唤醒所有等待锁的线程之后，也不⼀定会⻢上把时间⽚分给刚才放弃锁的那个线程，具体要看系统的调度。
* 调⽤join()⽅法不会释放锁，会⼀直等待当前线程执⾏完毕（转换为TERMINATED状态）。

#### **time\_waiting**与**runnable**状态转换

* Thread.sleep(long)：使当前线程睡眠指定的时间。只会使线程睡眠，不会释放锁；时间到后会重新进入runnable状态。
* Object.wait(long)：使线程进入time\_waiting状态；wait(long)⽅法与 ⽆参⽅法wait()相同的地⽅是，都可以通过其他线程调⽤notify()或notifyAll()⽅法来唤醒。不同的地⽅是，有参⽅法wait(long)就算其他线程不来唤醒它，经过指定时间long之后它会⾃动唤醒，拥有去争夺锁的资格。
* Thread.join(long)：是当前线程执行指定时间后，使线程进入time\_waiting状态。

#### 线程中断

* Thread.interrupt()：中断线程。这⾥的中断线程并不会⽴即停⽌线程，⽽是设置线程的中断状态为true（默认是flase）；
* Thread.interrupted()：测试当前线程是否被中断。线程的中断状态受这个⽅法 的影响，意思是调⽤⼀次使线程中断状态设置为true，连续调⽤两次会使得这 个线程的中断状态重新转为false；
* Thread.isInterrupted()：测试当前线程是否被中断。与上⾯⽅法不同的是调⽤ 这个⽅法并不会影响线程的中断状态。
* 在线程中断机制⾥，当其他线程通知需要被中断的线程后，线程中断的状态  
  被设置为true，但是具体被要求中断的线程要怎么处理，完全由被中断线程 ⾃⼰⽽定，可以在合适的实际处理中断请求，也可以完全不处理继续执⾏下 去。

## 5. Java线程间的通信

### 5.1 锁与同步

锁的概念都是基于对象的，所以我们⼜经常称它为对象锁。

线程同步是线程之间按照⼀定的顺序执⾏。

### 5.2 等待/通知机制

Java多线程的等待/通知机制是基于 Object 类的 wait() ⽅法和 notify() ,notifyAll() ⽅法来实现的。

notify()⽅法会随机叫醒⼀个正在等待的线程，⽽notifyAll()会叫醒所有正在等 待的线程。

⽽假如线程A现在持有了⼀ 个锁 lock 并开始执⾏，它可以使⽤ lock.wait() 让⾃⼰进⼊等待状态。这个时 候， lock 这个锁是被释放了的。 这时，线程B获得了 lock 这个锁并开始执⾏，它可以在某⼀时刻，使 ⽤ lock.notify() ，通知之前持有 lock 锁并进⼊等待状态的线程A，说“线程A你不 ⽤等了，可以往下执⾏了”。

需要注意的是，这个时候线程B并没有释放锁 lock ，除⾮线程B这个时候使⽤ lock.wait() 释放锁，或者线程B执⾏结束⾃⾏释放锁，线程A才能得 到 lock 锁。

需要注意的是等待/通知机制使⽤的是使⽤同⼀个对象锁，如果你两个线程使 ⽤的是不同的对象锁，那它们之间是不能⽤等待/通知机制通信的。

### 5.3 信号量

volitile关键字能够保证内存的可⻅性，如果⽤volitile关键字声明了⼀个变 量，在⼀个线程⾥⾯改变了这个变量的值，那其它线程是⽴⻢可⻅更改后的 值的。