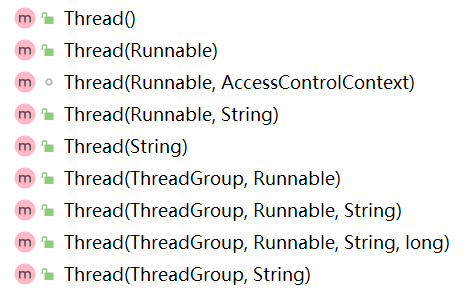
# 多线程并发编程

线程是程序中执行的线程。Java虚拟机允许应用程序拥有并发运行的多个执行线程。

多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。﻿

## 线程创建Thread Creation

* 声明一个类继承Thread类
* 声明一个类实现Runnable接口，然后创建一个此类的对象，作为参数来new一个Thread对象。- **策略模式, 职责分离，**Thread负责线程本身的职责和控制，Runnable是一个执行的任务单元。



调用线程对象的start()方法启动线程，启动后的线程如果**再次调用start()方法**，会抛出**IllegalThreadStateException**异常。

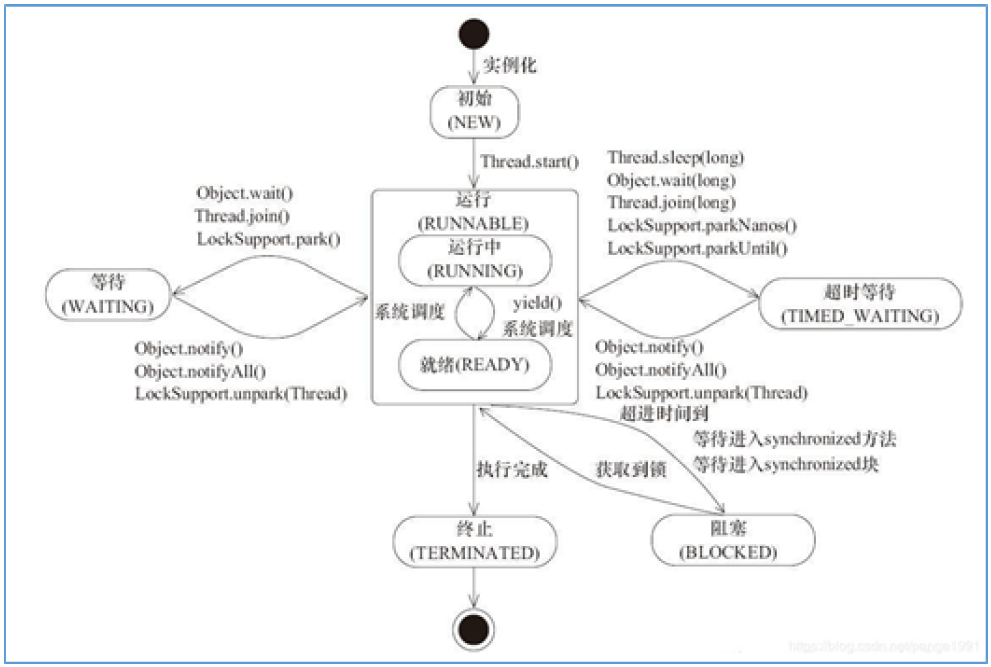
已经结束的线程(TERMINATED)同样不能再次调用start();

可以通过Java提供的**jconsole**等工具查看线程

**线程的父子关系：**main方法所在的线程为主线程，由JVM创建，程序中创建的其他线程是它的子线程。

**线程的创建和线程的调度都需要开销的，所以不是线程越多越好。**

## 线程状态Thread State



|  |  |
| --- | --- |
| **New** | NEW创建一个线程对象时，还没有调用start()方法执行线程。 |
| **RUNNABLE** | 调用线程对象的start()方法，线程进入可运行RUNNABLE状态，是否立即执行取决于CPU的调度。  RUNNABLE状态的线程还没有执行，可以转换为RUNNING状态，但不会转化为其他状态。 |
| **RUNNING** | CPU通过轮训或其他方式从可执行的队列中选中了线程，线程进入RUNNING状态。  RUNNING状态可以转换为WAITING, TIMED\_WAITING, BLOCKED or TERMINATED状态。 |
| **WAITING** | 无限期等待另一个线程执行特定操作的线程。 |
| **TIMED\_WAITING** | 在指定的等待时间内等待另一个线程执行某个操作的线程。 |
| **BLOCKED** | 在等待监视器**锁**时被阻塞的线程处于此状态。 |
| **TERMINATED** | 这是线程最终的状态的，在该状态的线程不会转换为其他状态。  以下情况线程进入TERMINATED状态：   * 线程运行正常结束 * 线程运行出错意外结束 * JVM Crash |

## 线程中断Thread Interception

当线程在**WAITING**, **TIMED\_WAITING** and **BLOCKED** 状态的时候，调用下面的intercept()方法可以中断线程。

interrupt()并不会终止处于“运行状态”的线程，它会将线程的中断标记设为true

与线程中断相关的方法和类有：

* public void intercept();
* public static boolean intercepted();
* public boolean isIntercepted()
* InterruptedException

当线程被中断时，会抛出**InterruptedException**, 通过捕获此异常来进行业务处理。当线程抛出InterruptedException时，线程的**intercept flag**将被清空，也就是**isIntercepted()**返回**false**。

**可中断方法**

* Object的wait方法
* Thread的sleep方法
* Thread的join方法
* 。。。

## 线程属性

* 守护线程(Demon Thread): JVM后台线程，如垃圾回收线程。
* 线程组
* 父子线程
* 线程优先级
* 线程ID
* 。。。

## 线程的异常处理

线程在执行单元中run()不能抛出cheched**异常**的，Java为线程提供了**UncaughtExceptionHandler**回调接口来处理异常。

|  |
| --- |
| **public class** CatchThreadException {  **public static void** main(String[] args) {  Thread thread = **new** Thread(()-> System.***out***.println(1/0));  thread.setUncaughtExceptionHandler(**new** Thread.UncaughtExceptionHandler() {  @Override  **public void** uncaughtException(Thread t, Throwable e) {  System.***out***.println(**"error: "** + e.getMessage());  }  });  thread.start();  } } |

类似于lambda 表达式，由于function interface中的抽象方法没有声明异常(throws 异常)， 所以lambda 表达式不能throw异常，只能通过try-catch语句将checked exception转化为unchecked exception.

## 共享数据与线程安全

一个对象的状态是可变的，并被多线程共享，就可能出现线程安全问题。

**线程安全性**首先是正确性，当多个线程访问某个类时，这个类始终能表现出正确的行为。

需要借助**原子性**和**封装性**解决线程安全性问题。

产生线程不安全的原因有：

* **Race condition(竞态条件)**，并发编程中因为不恰当的执行时序而出现不正确的结果的情况，比如：
* 先检查后执行

|  |
| --- |
| **//非线程安全的单例模式**  **public class** Singleton {  **private** Singleton **instance** = **null**;  **private** Singleton () {  }  **public** Singleton getInstance() {  *//惰性初始化，race condition 先检查后执行* **if**(**instance** == **null**)  **instance** = **new** Singleton();  **return instance**;  } } |

* 复合操作，比如：++i (读取-修改-写入)

|  |
| --- |
| x=10; //原子操作，执行线程在工作内存中将10写入x, 然后写入主内存。  x=y; //非原子操作，第一步执行线程从主内存中读取y的值，在工作内存中写入x；第二步再将修改后的x的值写入到主内存。  x++; //非原子的复合操作，读取-修改-写入。 |

* **不变性条件**

一个类有多个属性如min, max, 它们需要满足**min<=max**的不变性条件。

### 线程安全的类

* 无状态的类。
* 状态不可变的类。
* 使用同步机制对类或对象的可变状态的进行访问，如synchronized, locked, volatile。
* 使用ThreadLocal
* 组合对象将线程程安全性**委托**给**线程安全的状态变量**如Vector, Hashtable, ConcurrentHaspMap, StringBuffer, 原子封装类等。

|  |
| --- |
| @ThreadSafe **public class** Counting {  *//将类的状态委托给线程安全的类AtomicInteger* **private final** AtomicInteger **count** = **new** AtomicInteger(0);  **public void** increment() {  **count**.incrementAndGet(); //原子操作  }  **public int** getCount() {  **return count**.get();  } } |

但一个类有多个状态，即便每个状态都是线程安全的，也不能保证类是线程安全的。

|  |
| --- |
| */\*\*  \* 这个类不是线程安全的，虽然类的每个状态分别是线程安全的，但是两个状态之间有不变性条件的约束。  \* 必须通过加锁机制保证这些复合操作是原子的，才能实现线程安全。  \*\*/* @NotThreadSafe **public class** NumberRange {  *//多个变量之间的不变性条件 lower<=upper* **private final** AtomicInteger **lower** = **new** AtomicInteger(0);  **private final** AtomicInteger **upper** = **new** AtomicInteger(0);  **public void** setLower(**int** i) {  *//先检查后执行（race condition），不是原子操作，不安全* **if** (i > **upper**.get()) {  **throw new** IllegalArgumentException(**"can't set lower to "** + i + **" > upper"**);  }  **lower**.set(i);  }  **public void** setUpper(**int** i) {  *//先检查后执行（race condition），不是原子操作，不安全* **if** (i < **lower**.get()) {  **throw new** IllegalArgumentException(**"can't set upper to "** + i + **" < lower"**);  }  **upper**.set(i);  }  **public boolean** isInRange(**int** i) {  **return** (i >= **lower**.get() && i <= **upper**.get());  } } |

### 并发编程的三大特性

* **原子性**

一个或多个操作要么多成功，要么都失败。

**保证原子性：**

可以通过加锁(synchronized, lock), 或原子封装类型(java.util.concurrent.atomic.\*)

* **可见性**

当一个线程对共享数据的修改，其他线程可以立即看到修改后的新值。也就是执行线程将修改后的值更新到主内存，这样修改后的值对其他线程才可见。

**保证可见性：**

可以通过加锁(synchronized, lock), 或volatile关键字。

一个线程对volatile变量的修改，其他线程立即能看到。

* **有序性**

在Java的内存模型中，允许编译器和处理器对指令进行重新排序，在多线程的情况下，重新排序影响程序的正确执行。

保证有序性

使用volatile关键字保证,

使用synchronized 或Lock

|  |
| --- |
| int x = 0;  int y = 1;  volatile int z = 20; // volatile所在行之前的代码一定保证在其之后的代码之前执行。  x++;  y--; |

## synchronized & Locks

### synchronized

1. Java关键字，内置锁。
2. 获取锁的线程执行完同步代码后自动释放锁，或线程执行发生异常自动释放锁。
3. 获取锁时候需要**一直等待**其他线程释放锁。
4. **可重入**、不可中断、非公平
5. 不可以判断锁的状态

### Lock & ReentrantLock

1. 获取锁的线程执行完同步代码后不能自动释放，应该在finally子句中手工调用unlock()方法释放锁。
2. 获取锁的时候可以指定时间tryLock(long timeout, TimeUnit unit)，不需要一直等待。
3. **可重入**，可中断，可公平
4. 可以判断锁的状态

**ReadWriteLock & ReentrantReadWriteLock**

ReadWriteLock维护一对相关联的锁，一个用于只读操作，一个用于写入操作。读锁可以由多个读线程同时持有；写锁是排他的。

### synchronized的优化

synchronized关键字在 JDK1.6 版本之前，是通过操作系统的 Mutex Lock 来实现同步的，属于重量锁。

在 JDK1.6 版本中，HotSpot 虚拟机开发团队花了很大的精力去实现各种锁优化技术，如：适应性自旋、锁消除、锁粗话、偏向锁、轻量级锁等。其中最重要的是：**自旋锁、轻量级锁、偏向锁**

|  |  |  |  |
| --- | --- | --- | --- |
| **现状** | **锁名称** | **收益** | **使用场景** |
| 大多数情况下，等待锁的时间比操作系统 mutex 短得多 | 自旋锁 | 减少内核态与用户态切换的开销 | 线程获取锁时间较短的情况 |
| 大多数情况下，锁同步期间没有线程竞争 | 轻量级锁 | 与自旋锁相比，减少了自旋时间 | 没有线程竞争锁 |
| 大多数情况下，锁同步期间没有线程竞争 | 偏向锁 | 与轻量级锁相比，减少了多余的对象复制操作 | 没有线程竞争 |

## wait和sleep方法区别

* **相同点：**

1. 可以使当前线程进入WAITING or TIMED\_WAITING状态
2. 都是可中断方法，被中断后收到InterruptedException异常

* **不同点**

1. wait是Object方法，sleep是Thread的静态方法
2. 线程在同步方法中执行sleep方法时，并不会释放monitor的锁，而wait方法会释放monitor锁。
3. wait方法的执行必须在同步方法中进行，而sleep不需要。
4. sleep方法休眠之后会退出WAITING or TIMED\_WAITING状态，而wait方法则需要被其他线程唤醒（notify, notifyAll方法）或被其他线程中断才能退出WAITING or TIMED\_WAITING状态。

Thread的join方法，等待此线程结束

## 线程间通信

* 线程间通信方法
  + Object类的 wait/notify 方法
  + condition 的 await/signal 方法
* 生产者和消费者模式

## Runnable & Callable & Future & FutureTask

* **Runnable:** 无返回结果，没有声明抛出异常。配合Thread或 ThreadPoolExecutor使用。
* **Callable:** 返回结果并可能抛出异常的任务。只能配合**ThreadPoolExecutor**使用。
* **Future:** 表示异步计算的结果。只能配合**ThreadPoolExecutor**使用。提供了用于检查计算是否完成、等待计算完成以及检索计算结果的方法。只能在计算完成时使用get方法检索结果，必要时阻塞直到它准备好。通过cancel方法执行取消。提供了其他方法来确定任务是正常完成还是被取消。一旦计算完成，就不能取消计算。如果您希望使用Future来实现可取消性，但不提供可用的结果，则可以声明Future<?>并返回null作为底层任务的结果。
* **FutureTask:** 一个可取消的异步计算。可以作为**Runnable**被线程执行，又可以有**Future**的那些操作。这个类提供了Future的基本实现，具有启动和取消计算、查询计算是否完成以及检索计算结果的方法。只有在计算完成时才能检索结果;如果计算尚未完成，get方法将阻塞。一旦计算完成，就不能重新启动或取消计算。

|  |
| --- |
| **public class** FutureTaskDemo {  **public static void** main(String[] args) {  *//1. FutureTask & Thread* FutureTask<String> futureTask = **new** FutureTask<>(() -> **"future task"**);  **new** Thread(futureTask).start();  **try** {  System.***out***.println(futureTask.get());  } **catch** (InterruptedException e) {  //TODO  } **catch** (ExecutionException e) {  //TODO  }  *//2. FutureTask and ThreadPool* ExecutorService executorService = Executors.*newFixedThreadPool*(1);  executorService.submit(futureTask);  **try** {  System.***out***.println(futureTask.get());  } **catch** (InterruptedException e) {  //TODO  } **catch** (ExecutionException e) {  //TODO  } **finally** {  executorService.shutdownNow(); // ExecutorService 启动多线程执行完毕之后,应该关闭  }  } } |

\*Java 1.8 增加了 **CompletableFuture**。

## 执行器(Executors)与线程池

**执行器： Executors/ Executor**

﻿线程池的好处

1. 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
2. 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
3. 提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。

### 一个线程池四个基本组成部分：

1. 线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；
2. 工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；
3. 任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；
4. 任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

### 线程池工作流程

1）当提交一个新任务到线程池时，线程池判断corePoolSize线程是否都在执行任务，如果没有空闲线

程，则创建一个新的工作线程来执行任务，直到当前线程数等于corePoolSize；

2）如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

3）如果阻塞队列满了，那就创建新的线程执行当前任务，直到线程池中的线程数达到maxPoolSize，

这时再有任务来，由饱和策略来处理提交的任务。

4）如果超过corePoolSize的线程的空闲时间达到keepAliveTime,多余的线程会被销毁直到剩下corePoolSize个线程为止。

### 线程池参数

|  |
| --- |
| **public** ThreadPoolExecutor(**int** corePoolSize,  **int** maximumPoolSize,  **long** keepAliveTime,  TimeUnit unit,  BlockingQueue<Runnable> workQueue,  ThreadFactory threadFactory,  RejectedExecutionHandler handler) |

### 四种饱和策略

AbortPolicy：不处理，直接抛出异常。

CallerRunsPolicy：若线程池还没关闭，调用当前所在线程来运行任务，r.run()执行。

DiscardOldestPolicy：LRU策略，丢弃队列里最近最久不使用的一个任务，并执行当前任务。

DiscardPolicy：不处理，丢弃掉，不抛出异常。

### 线程池关闭和线程池状态

### 如何设置线程池大小？

CPU计算密集型：cpu数量

I/O读写密集型：

## 同步组件

* CountDownLatch
* CyclicBarrier
* Semaphore
* Exchanger
* Fork/Join (Java 7）ForkJoinPool 是ExecutorService的一个实现将大的任务分解成一个个小任务, 然后汇总。

**Work-stealing**

工作窃取算法的优点：

利用了线程进行并行计算，减少了线程间的竞争。

**工作窃取算法的缺点：**

1、如果双端队列中只有一个任务时，线程间会存在竞争。

2、窃取算法消耗了更多的系统资源，如会创建多个线程和多个双端队列。

**Fork/Join中两个重要的类：**

ForkJoinTask：使用该框架，需要创建一个ForkJoin任务，它提供在任务中执行fork和join操作的机制。一般情况

下，我们并不需要直接继承ForkJoinTask类，只需要继承它的子类，它的子类有两个：

RecursiveAction:用于没有返回结果的任务。

RecursiveTask:用于有返回结果的任务。

ForkJoinPool：任务ForkJoinTask需要通过ForkJoinPool来执行

## 线程的优点和风险

### 优点

多线程可以提供单核或多核CPU资源的利用率和吞吐量；异步处理事件，提高用户界面的响应度。

### 风险

* **安全性：**对于共享数据的修改，由于多线程的交替操作导致不可预测的结果。
* **死锁：**线程 1 拥有了锁 A 的情况下试图获取锁 B，而线程 2 又在拥有了锁 B 的情况下试图获取锁 A，这样双方就进入相互阻塞等待的情况，也就是锁的循环依赖。如果必须获得**多个锁**，多个线程以相同的**顺序**获得锁，就不会死锁。通过定时的锁避免死锁。通过线程转储信息来分析死锁，Windows 平台中按 Ctrl+C。
* **活锁：**当多个相互协作的线程都对彼此进行响应而修改各自的状态，并使得任何一个线程都无法继续进行。通过让线程等待随机的时间来避免活锁。
* **饥饿：**但线程无法访问它所需要的资源而不能执行是就发生了”饥饿**”**，通过修改线程得优先级未必能解决问题，避免使用线程优先级，使用默认的线程优先级。
* **性能：**线程创建、多线程切换、同步机制都有额外的性能开销。

## 多线程设计模式