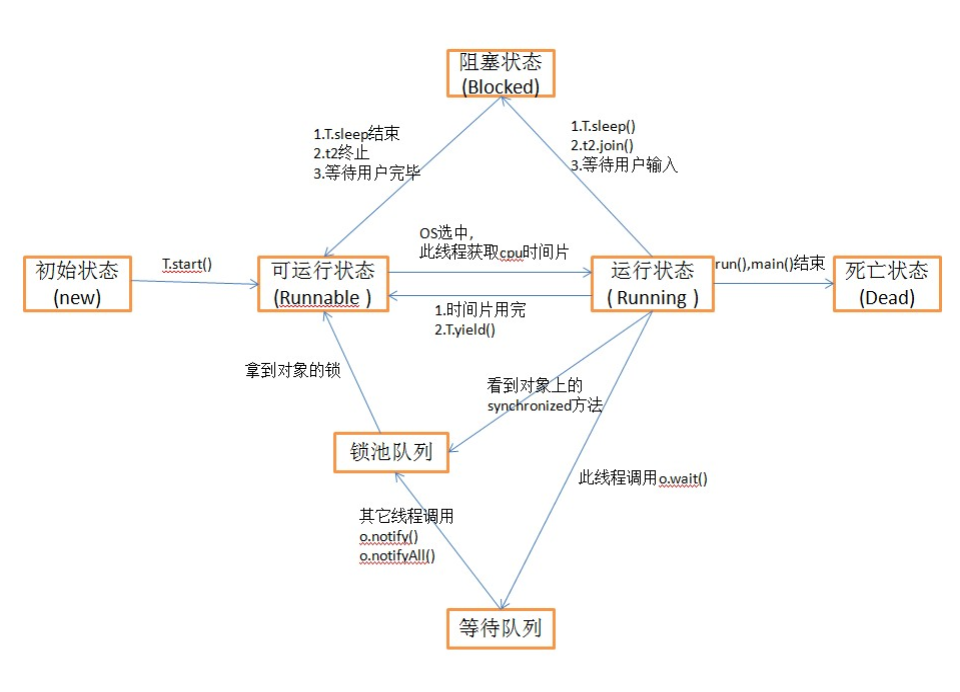
1. **线程的状态转换**



**（一）新建状态（New）：**

用new语句创建的线程处于新建状态，此时它和其他Java对象一样，仅仅在堆区中被分配了内存。

**（二）就绪状态（Runnable）**：

当一个线程对象创建后，其他线程调用它的start()方法，该线程就进入就绪状态，Java虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得CPU的使用权。

**（三）运行状态（Running）：**

处于这个状态的线程占用CPU，执行程序代码。只有处于就绪状态的线程才有机会转到运行状态。

**（四）阻塞状态（Blocked）：**

阻塞状态是指线程因为某些原因放弃CPU，暂时停止运行。当线程处于阻塞状态时，Java虚拟机不会给线程分配CPU。直到线程重新进入就绪状态，它才有机会转到运行状态。

阻塞状态可分为以下3种：

**1.位于对象等待池中的阻塞状态（Blocked in object’s wait pool）：**

当线程处于运行状态时，如果执行了某个对象的wait()方法，Java虚拟机就会把线程放到这个对象的等待池中，这涉及到“线程通信”的内容。

**2.位于对象锁池中的阻塞状态（Blocked in object’s lock pool）：**

当线程处于运行状态时，试图获得某个对象的同步锁时，如果该对象的同步锁已经被其他线程占用，Java虚拟机就会把这个线程放到这个对象的锁池中，这涉及到“线程同步”的内容。

**3.其他阻塞状态（Otherwise Blocked）：**

当前线程执行了sleep()方法，或者调用了其他线程的join()方法，或者发出了I/O请求时，就会进入这个状态。

**（五）死亡状态（Dead）：**

当线程退出run()方法时，就进入死亡状态，该线程结束生命周期。

1. **使用线程**

三种创建线程的方法

1. 实现 Runnable 接口；
2. 实现 Callable 接口；
3. 继承 Thread 类。

注：实现 Runnable 和 Callable 接口的类只能当做一个可以在线程中运行的任务，不是真正意义上的线程，因此最后还需要通过 Thread 来调用。可以说任务是通过线程驱动从而执行的

**（一）实现 Runnable 接口**

需要实现 run() 方法。

通过 Thread 调用 start() 方法来启动线程。

public class MyRunnable implements Runnable {

public void run() { }

}

public static void main(String[] args) {

MyRunnable instance = new MyRunnable();

Thread thread = new Thread(instance);

thread.start();

}

**（二）实现 Callable 接口**

与 Runnable 相比，Callable 可以有返回值，返回值通过 FutureTask 进行封装。

public class MyCallable implements Callable<Integer> {

public Integer call() {

return 123;

}

}

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

MyCallable mc = new MyCallable();

FutureTask<Integer> ft = new FutureTask<>(mc);

Thread thread = new Thread(ft);

thread.start();

System.out.println(ft.get());

}

**（三）继承 Thread 类**

同样也是需要实现 run() 方法，因为 Thread 类也实现了 Runable 接口。

当调用 start() 方法启动一个线程时，虚拟机会将该线程放入就绪队列中等待被调度，当一个线程被调度时会执行该

线程的 run() 方法。

public class MyThread extends Thread {

public void run() { }

}

public static void main(String[] args) {

MyThread mt = new MyThread();

mt.start();

}

**（四）实现接口 VS 继承 Thread**

实现接口会更好一些，因为：Java 不支持多重继承，因此继承了 Thread 类就无法继承其它类，但是可以实现多个接口；

类可能只要求可执行就行，继承整个 Thread 类开销过大。

**三、基础线程机制**

Executor

Executor 管理多个异步任务的执行，而无需程序员显式地管理线程的生命周期。这里的异步是指多个任务的执行互不干扰，不需要进行同步操作。

主要有三种 Executor：

CachedThreadPool：一个任务创建一个线程；

FixedThreadPool：所有任务只能使用固定大小的线程；

SingleThreadExecutor：相当于大小为 1 的 FixedThreadPool。

public static void main(String[] args) {

ExecutorService executorService = Executors.newCachedThreadPool();

for (int i = 0; i < 5; i++) {

executorService.execute(new MyRunnable());

}

executorService.shutdown();

}

Daemon

守护线程是程序运行时在后台提供服务的线程，不属于程序中不可或缺的部分。

当所有非守护线程结束时，程序也就终止，同时会杀死所有守护线程。

main() 属于非守护线程。

使用 setDaemon() 方法将一个线程设置为守护线程。

public static void main(String[] args) {

Thread thread = new Thread(new MyRunnable());

thread.setDaemon(true);

}

sleep()

Thread.sleep(millisec) 方法会休眠当前正在执行的线程，millisec 单位为毫秒。

sleep() 可能会抛出 InterruptedException，因为异常不能跨线程传播回 main() 中，因此必须在本地进行处理。线程中抛出的其它异常也同样需要在本地进行处理。

public void run() {

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

yield()

对静态方法 Thread.yield() 的调用声明了当前线程已经完成了生命周期中最重要的部分，可以切换给其它线程来执行。该方法只是对线程调度器的一个建议，而且也只是建议具有相同优先级的其它线程可以运行。

public void run() {

Thread.yield();

}

**四、中断**

一个线程执行完毕之后会自动结束，如果在运行过程中发生异常也会提前结束。

InterruptedException

通过调用一个线程的 interrupt() 来中断该线程，如果该线程处于阻塞、限期等待或者无限期等待状态，那么就会抛

出 InterruptedException，从而提前结束该线程。但是不能中断 I/O 阻塞和 synchronized 锁阻塞。

对于以下代码，在 main() 中启动一个线程之后再中断它，由于线程中调用了 Thread.sleep() 方法，因此会抛出一个

InterruptedException，从而提前结束线程，不执行之后的语句。

public class InterruptExample {

private static class MyThread1 extends Thread {

@Override

public void run() {

try {

Thread.sleep(2000);

System.out.println("Thread run");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

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

Thread thread1 = new MyThread1();

thread1.start();

thread1.interrupt();

System.out.println("Main run");

}

Main run

java.lang.InterruptedException: sleep interrupted

at java.lang.Thread.sleep(Native Method)

at InterruptExample.lambda$main$0(InterruptExample.java:5)

at InterruptExample$$Lambda$1/713338599.run(Unknown Source)

at java.lang.Thread.run(Thread.java:745)

interrupted()

如果一个线程的 run() 方法执行一个无限循环，并且没有执行 sleep() 等会抛出 InterruptedException 的操作，那么

调用线程的 interrupt() 方法就无法使线程提前结束。

但是调用 interrupt() 方法会设置线程的中断标记，此时调用 interrupted() 方法会返回 true。因此可以在循环体中使

用 interrupted() 方法来判断线程是否处于中断状态，从而提前结束线程。

public class InterruptExample {

private static class MyThread2 extends Thread {

@Override

public void run() {

while (!interrupted()) {

}

System.out.println("Thread end");

}

}

}

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

Thread thread2 = new MyThread2();

thread2.start();

thread2.interrupt();

}

Thread end

Executor 的中断操作

调用 Executor 的 shutdown() 方法会等待线程都执行完毕之后再关闭，但是如果调用的是 shutdownNow() 方法，

则相当于调用每个线程的 interrupt() 方法。

以下使用 Lambda 创建线程，相当于创建了一个匿名内部线程。

public static void main(String[] args) {

ExecutorService executorService = Executors.newCachedThreadPool();

executorService.execute(() -> {

try {

Thread.sleep(2000);

System.out.println("Thread run");

} catch (InterruptedException e) {

e.printStackTrace();

}

});

executorService.shutdownNow();

System.out.println("Main run");

}

Main run

java.lang.InterruptedException: sleep interrupted

at java.lang.Thread.sleep(Native Method)

at ExecutorInterruptExample.lambda$main$0(ExecutorInterruptExample.java:9)

at ExecutorInterruptExample$$Lambda$1/1160460865.run(Unknown Source)

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)

at java.lang.Thread.run(Thread.java:745)

如果只想中断 Executor 中的一个线程，可以通过使用 submit() 方法来提交一个线程，它会返回一个 Future<?> 对

象，通过调用该对象的 cancel(true) 方法就可以中断线程。

Future<?> future = executorService.submit(() -> {

// ..

});

future.cancel(true);

**五、互斥同步**

**synchronized**

**ReentrantLock**

ReentrantLock 是 java.util.concurrent（J.U.C）包中的锁。

Executors工具类可以创建各种类型的线程池，如下为常见的四种：

1. newCachedThreadPool ：大小不受限，当线程释放时，可重用该线程；
2. newFixedThreadPool ：大小固定，无可用线程时，任务需等待，直到有可用线程；
3. newSingleThreadExecutor ：创建一个单线程，任务会按顺序依次执行；
4. newScheduledThreadPool：创建一个定长线程池，支持定时及周期性任务执行

**线程之间的协作**

当多个线程可以一起工作去解决某个问题时，如果某些部分必须在其它部分之前完成，那么就需要对线程进行协调。

join()

在线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束。

对于以下代码，虽然 b 线程先启动，但是因为在 b 线程中调用了 a 线程的 join() 方法，b 线程会等待 a 线程结束才继续执行，因此最后能够保证 a 线程的输出先于 b 线程的输出。

wait() notify() notifyAll()

调用 wait() 使得线程等待某个条件满足，线程在等待时会被挂起，当其他线程的运行使得这个条件满足时，其它线程

会调用 notify() 或者 notifyAll() 来唤醒挂起的线程。

它们都属于 Object 的一部分，而不属于 Thread。

只能用在同步方法或者同步控制块中使用，否则会在运行时抛出 IllegalMonitorStateException。

使用 wait() 挂起期间，线程会释放锁。这是因为，如果没有释放锁，那么其它线程就无法进入对象的同步方法或者同步控制块中，那么就无法执行 notify() 或者 notifyAll() 来唤醒挂起的线程，造成死锁。

**wait() 和 sleep() 的区别**

wait() 是 Object 的方法，而 sleep() 是 Thread 的静态方法；

wait() 会释放锁，sleep() 不会。

**await() signal() signalAll()**

java.util.concurrent 类库中提供了 Condition 类来实现线程之间的协调，可以在 Condition 上调用 await() 方法使线程等待，其它线程调用 signal() 或 signalAll() 方法唤醒等待的线程。

相比于 wait() 这种等待方式，await() 可以指定等待的条件，因此更加灵活。

使用 Lock 来获取一个 Condition 对象。

**J.U.C – AQS**

java.util.concurrent（J.U.C）大大提高了并发性能，AQS 被认为是 J.U.C 的核心。

**CountDownLatch**

用来控制一个线程等待多个线程。

维护了一个计数器 cnt，每次调用 countDown() 方法会让计数器的值减 1，减到 0 的时候，那些因为调用 await() 方法而在等待的线程就会被唤醒。

**CyclicBarrier**

用来控制多个线程互相等待，只有当多个线程都到达时，这些线程才会继续执行。

和 CountdownLatch 相似，都是通过维护计数器来实现的。线程执行 await() 方法之后计数器会减 1，并进行等待，直到计数器为 0，所有调用 await() 方法而在等待的线程才能继续执行。

CyclicBarrier 和 CountdownLatch 的一个区别是，CyclicBarrier 的计数器通过调用 reset() 方法可以循环使用，所以它才叫做循环屏障。

CyclicBarrier 有两个构造函数，其中 parties 指示计数器的初始值，barrierAction 在所有线程都到达屏障的时候会执行一次。

**Semaphore**

Semaphore 类似于操作系统中的信号量，可以控制对互斥资源的访问线程数。

以下代码模拟了对某个服务的并发请求，每次只能有 3 个客户端同时访问，请求总数为 10。

**J.U.C - 其它组件**

**FutureTask**

在介绍 Callable 时我们知道它可以有返回值，返回值通过 Future 进行封装。FutureTask 实现了 RunnableFuture 接口，该接口继承自 Runnable 和 Future 接口，这使得 FutureTask 既可以当做一个任务执行，也可以有返回值。

FutureTask 可用于异步获取执行结果或取消执行任务的场景。当一个计算任务需要执行很长时间，那么就可以用

FutureTask 来封装这个任务，主线程在完成自己的任务之后再去获取结果。

**BlockingQueue**

java.util.concurrent.BlockingQueue 接口有以下阻塞队列的实现：

FIFO 队列 ：LinkedBlockingQueue、ArrayBlockingQueue（固定长度）

优先级队列 ：PriorityBlockingQueue

提供了阻塞的 take() 和 put() 方法：如果队列为空 take() 将阻塞，直到队列中有内容；如果队列为满 put() 将阻塞，直到队列有空闲位置。

**ForkJoin**

主要用于并行计算中，和 MapReduce 原理类似，都是把大的计算任务拆分成多个小任务并行计算。

ForkJoin 使用 ForkJoinPool 来启动，它是一个特殊的线程池，线程数量取决于 CPU 核数。

ForkJoinPool 实现了工作窃取算法来提高 CPU 的利用率。每个线程都维护了一个双端队列，用来存储需要执行的任

务。工作窃取算法允许空闲的线程从其它线程的双端队列中窃取一个任务来执行。窃取的任务必须是最晚的任务，避

免和队列所属线程发生竞争。