线程：

什么是线程&多线程

线程：

线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源(如程序计数器，一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

多线程：

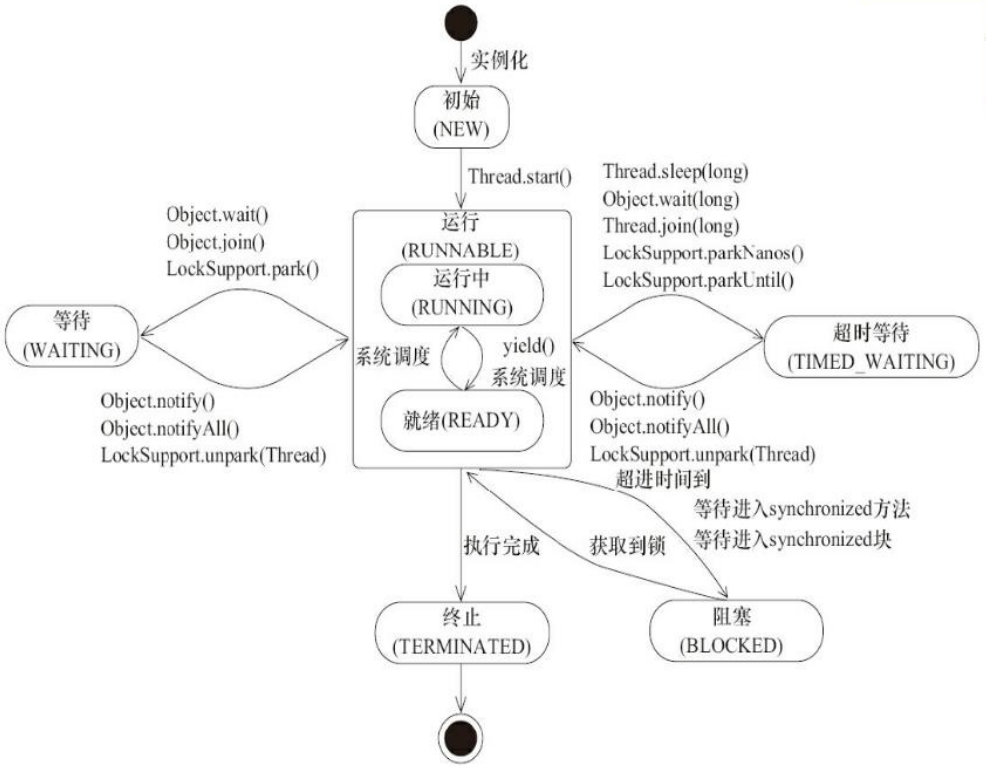
多线程指在单个程序中可以同时运行多个不同的线程执行不同的任务。

多线程编程的目的，就是“最大限度地利用 cpu 资源”，当某一线程的处理不需要占用 cpu 而只和 io 等资源打交道时，让需要占用 Cpu 的其他线程有其他机会获得 cpu 资源。从根本上说，这就是多线程编程的最终目的。

线程实现的方式：

Runnable、Thread、Callable

线程的生命周期&状态



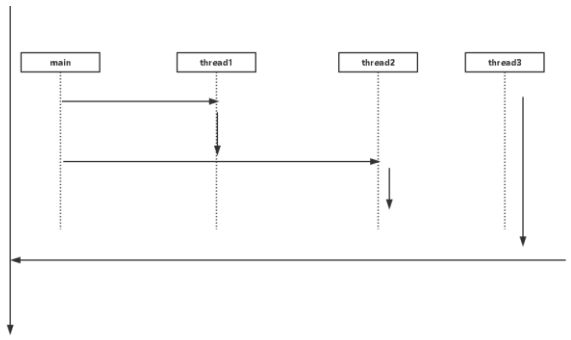
状态名称	说　　明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

线程的执行顺序：

```
public class ThreadSort {

  public static void main(String[] args) throws InterruptedException {
    Thread thread1 = new Thread(()->{
      System.out.println("thread1");

    });
    Thread thread2 = new Thread(()->{
      System.out.println("thread2");

    });
    Thread thread3 = new Thread(()->{
      System.out.println("thread3");

    });
    thread1.start();
    thread1.join();
    thread2.start();
    thread2.join();
    thread3.start();
    thread3.join();
```

Join 线程的运行顺序

原理：



Object wait

Callable&Future：

线程越多效率越高 1 0

Set get 方法（判断线程的状态来决定的）

总结

最后再来看看它们三个之间的总结。

实现 Runnable 接口相比继承 Thread 类有如下优势

1）可以避免由于 Java 的单继承特性而带来的局限
2）增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的
3）线程池只能放入实现 Runable 或 Callable 类线程，不能直接放入继承 Thread 的类

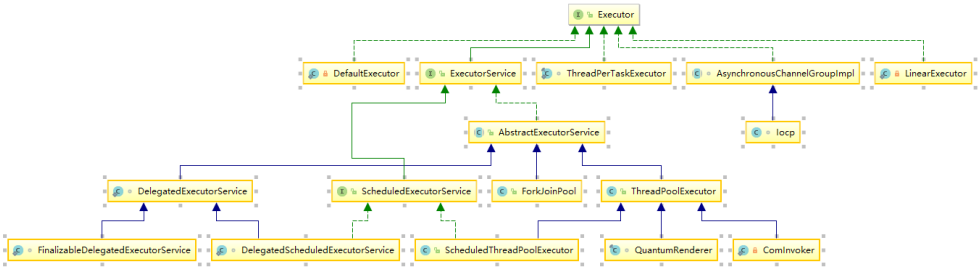
实现 Runnable 接口和实现 Callable 接口的区别

1）Runnable 是自从 java1.1 就有了，而 Callable 是 1.5 之后才加上去的
2）实现 Callable 接口的任务线程能返回执行结果，而实现 Runnable 接口的任务线程不能返回结果
3）Callable 接口的 call()方法允许抛出异常，而 Runnable 接口的 run()方法的异常只能在内部消化，不能继续上抛
4）加入线程池运行，Runnable 使用 ExecutorService 的 execute 方法，Callable 使用 submit 方法
注：Callable 接口支持返回执行结果，此时需要调用 FutureTask.get()方法实现，此方法会阻塞主线程直到获取返回结果，当不调用此方法时，主线程不会阻塞

线程与线程池对比

线程池

体系介绍：



Java 中有几种线程池？

普通线程池 ThreadPoolExecutor

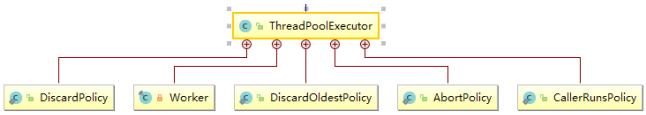
定时线程池 ScheduledThreadPoolExecutor

层级	名称	方法	说明	类型
1	java.util.concurrent.Executor	java.util.concurrent.Executor#execute	执行接口	接口
2	java.util.concurrent.ExecutorService	java.util.concurrent.ExecutorService#submit(java.util.concurrent.Callable<T>)	提交接口	接口
3	java.util.concurrent.AbstractExecutorService	java.util.concurrent.AbstractExecutorService#submit(java.util.concur	把执行和提交接口进行合并 区别:有返回值和无返回值	抽象类

		rent.Callable<T>)		
4	java.util.concurrent.ThreadPoolExecutor	java.util.concurrent.ThreadPoolExecutor#execute	调用 addwork（offer>task 放队列） Run 方法调用 runwork 方法 getTask（从队列拿数据）	实现类
5	java.util.concurrent.ScheduledExecutorService	Schedule 、scheduleAtFixedRat 、scheduleWithFixedDelay	定义方法 定义接口	接口 接口
6	java.util.concurrent.ScheduledThreadPoolExecutor	delayedExecute	具体实现 add>task>addWork	实现类

使用&源码分析：

java.util.concurrent.ThreadPoolExecutor



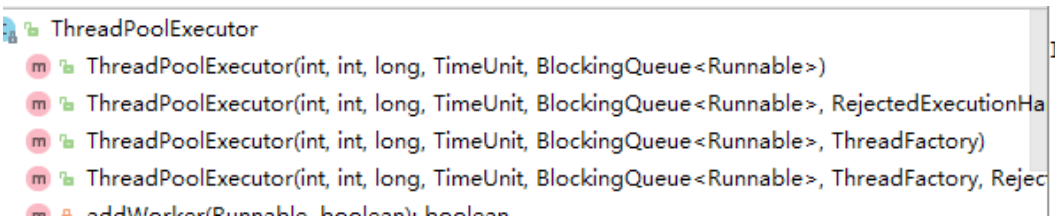
•

5 个内部类

分两种类型：policy（策略） worker（工作）

内部工作原理（构造方法赋值）

- corePoolSize ：池中所保存的线程数，包括空闲线程
- maximumPoolSize：池中允许的最大线程数
- keepAliveTime： 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间
- unit：keepAliveTime 参数的时间单位
- workQueue ：执行前用于保持任务的队列。此队列仅保持由 execute 方法提交的 Runnable 任务
- threadFactory：执行程序创建新线程时使用的工厂
- handler ：由于超出线程范围和队列容量而使执行



线程池运行思路

如果当前池大小 poolSize 小于 corePoolSize ，则创建新线程执行任务

如果当前池大小 poolSize 大于 corePoolSize ，且等待队列未满，则进入等待队列

如果当前池大小 poolSize 大于 corePoolSize 且小于 maximumPoolSize ，且等待队列已满，则创建新线程执行任务

如果当前池大小 poolSize 大于 corePoolSize 且大于 maximumPoolSize ，且等待队列已满，则调用拒绝策略来处理该任务

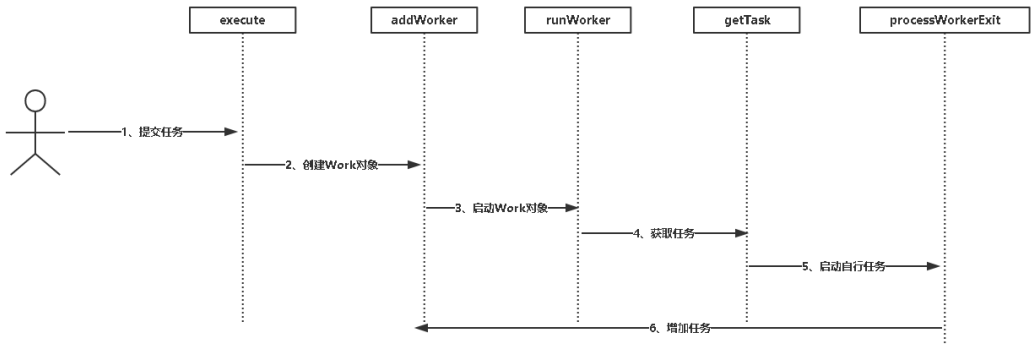
线程池里的每个线程执行完任务后不会立刻退出，而是会去检查下等待队列里是否还有线程任务需要执行，如果在 keepAliveTime 里等不到新的任务了，那么线程就会退出

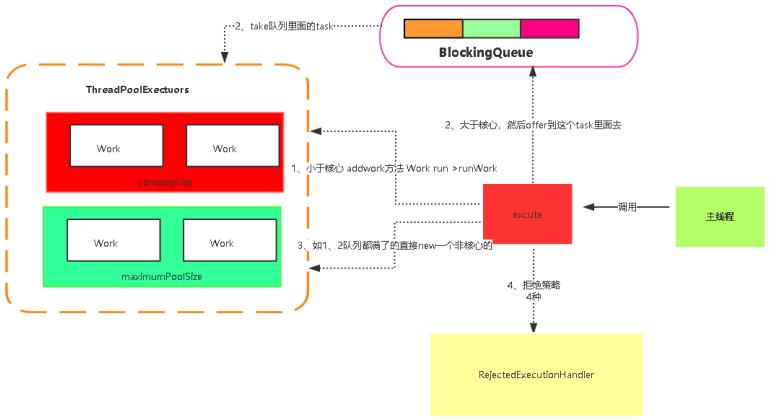
我们自己的任务，还是他线程

Submit 和 execute 方法

1、有返回值和无返回值

2、Task 不一样 futuretask 一个 task 本身





拒绝策略

线程池有四种拒绝策略：

AbortPolicy：抛出异常，**默认**

CallerRunsPolicy：不使用线程池执行

DiscardPolicy：直接丢弃任务

DiscardOldestPolicy：丢弃队列中最旧的任务

对于线程池选择的拒绝策略可以通过 RejectedExecutionHandler handler = new ThreadPoolExecutor.CallerRunsPolicy();来设置。

Jps>jstack

jstack 是排查线程死锁