ThreadPoolExecutor

一、线程池架构

1、Executor接口

2、ExecutorService接口

3、ScheduledExecutorService接口

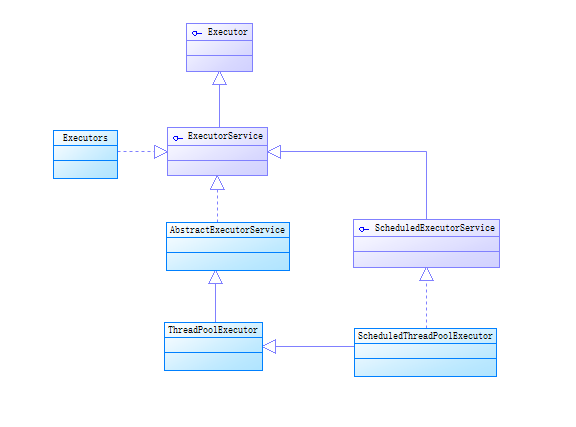
二、ThreadPoolExecutor

1、ThreadPoolExecutor构造参数

2、ThreadPoolExecutor线程池执行流程

三、Executors静态工厂创建几种常用线程池

一、线程池架构



**1、Executor 最基础的执行接口；**

ExecutorService接口继承了Executor，在其上做了一些shutdown()、submit()的扩展，可以说是真正的线程池接口；

抽象类实现了ExecutorService接口中的大部分方法；

TheadPoolExecutor继承了AbstractExecutorService，是线程池的具体实现；

ScheduledExecutorService接口继承了ExecutorService接口，提供了带"周期执行"功能ExecutorService；

ScheduledThreadPoolExecutor既继承了TheadPoolExecutor线程池，也实现了ScheduledExecutorService接口，是带"周期执行"功能的线程池；

Executors是线程池的静态工厂，其提供了快捷创建线程池的静态方法。

Executor只提供一个方法：

void execute(Runnable command);

提供了将任务提交与任务执行解耦的方法

**2、ExecutorService接口：**

“执行者服务”接口，可以说是真正的线程池接口，在Executor接口的基础上做了一些扩展，主要是

(A) 管理任务如何终止的 shutdown相关方法

/\*\*

 \* 启动一次有序的关闭，之前提交的任务执行，但不接受新任务

 \* 这个方法不会等待之前提交的任务执行完毕

 \*/

void shutdown();

/\*\*

 \* 试图停止所有正在执行的任务，暂停处理正在等待的任务，返回一个等待执行的任务列表

 \* 这个方法不会等待正在执行的任务终止

 \*/

List<Runnable> shutdownNow();

/\*\*

 \* 如果已经被shutdown，返回true

 \*/

boolean isShutdown();

/\*\*

 \* 如果所有任务都已经被终止，返回true

 \* 是否为终止状态

 \*/

boolean isTerminated();

/\*\*

 \* 在一个shutdown请求后，阻塞的等待所有任务执行完毕

 \* 或者到达超时时间，或者当前线程被中断

 \*/

boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;

(B) 可以生成用于追踪一个或多个异步任务执行结果的Future对象的 submit()相关方法

/\*\*

 \* 提交一个可执行的任务，返回一个Future代表这个任务

 \* 等到任务成功执行，Future#get()方法会返回null

 \*/

Future<?> submit(Runnable task);

/\*\*

 \* 提交一个可以执行的任务，返回一个Future代表这个任务

 \* 等到任务执行结束，Future#get()方法会返回这个给定的result

 \*/

<T> Future<T> submit(Runnable task, T result);

/\*\*

 \* 提交一个有返回值的任务，并返回一个Future代表等待的任务执行的结果

 \* 等到任务成功执行，Future#get()方法会返回任务执行的结果

 \*/

<T> Future<T> submit(Callable<T> task);

**3、ScheduledExecutorService接口**

/\*\*

 \* 在给定延时后，创建并执行一个一次性的Runnable任务

 \* 任务执行完毕后，ScheduledFuture#get()方法会返回null

 \*/

public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);

/\*\*

 \* 在给定延时后，创建并执行一个ScheduledFutureTask

 \* ScheduledFuture 可以获取结果或取消任务

 \*/

public <V> ScheduledFuture<V> schedule(Callable<V> callable, ong delay, TimeUnit unit);

/\*\*

 \* 创建并执行一个在给定初始延迟后首次启用的定期操作，后续操作具有给定的周期

 \* 也就是将在 initialDelay 后开始执行，然后在 initialDelay+period 后执行，接着在 initialDelay + 2 \* period 后执行，依此类推

 \* 如果执行任务发生异常，随后的任务将被禁止，否则任务只会在被取消或者Executor被终止后停止

 \* 如果任何执行的任务超过了周期，随后的执行会延时，不会并发执行

 \*/

public ScheduledFuture<?> scheduleAtFixedRate(

Runnable command, long initialDelay,long period,TimeUnit unit);

/\*\*

 \* 创建并执行一个在给定初始延迟后首次启用的定期操作，随后，在每一次执行终止和下一次执行开始之间都存在给定的延迟

 \* 如果执行任务发生异常，随后的任务将被禁止，否则任务只会在被取消或者Executor被终止后停止

 \*/

public ScheduledFuture<?> scheduleWithFixedDelay(

Runnable command, long initialDelay, long delay, TimeUnit unit);

二、ThreadPoolExecutor

1：ThreadPoolExecutor构造参数

public ThreadPoolExecutor(int corePoolSize,

          int maximumPoolSize,

          long keepAliveTime,

          TimeUnit unit,

          BlockingQueue<Runnable> workQueue,

          ThreadFactory threadFactory,

          RejectedExecutionHandler handler)

**corePoolSize**

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；

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；

如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

**maximumPoolSize**

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize

**keepAliveTime**

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间。默认情况下，该参数只在线程数大于corePoolSize时才有用

**workQueue**

workQueue必须是BlockingQueue阻塞队列。当线程池中的线程数超过它的corePoolSize的时候，线程会进入阻塞队列进行阻塞等待。通过workQueue，线程池实现了阻塞功能

**几种排队的策略：**

1. 不排队，直接提交  
   将任务直接交给线程处理而不保持它们，可使用SynchronousQueue  
   如果不存在可用于立即运行任务的线程（即线程池中的线程都在工作），则试图把任务加入缓冲队列将会失败，因此会构造一个新的线程来处理新添加的任务，并将其加入到线程池中（corePoolSize-->maximumPoolSize扩容）  
   Executors.newCachedThreadPool()采用的便是这种策略
2. 无界队列

可以使用LinkedBlockingQueue（基于链表的有界队列，FIFO），理论上是该队列可以对无限多的任务排队

将导致在所有corePoolSize线程都工作的情况下将新任务加入到队列中。这样，创建的线程就不会超过corePoolSize，也因此，maximumPoolSize的值也就无效了

1. 有界队列

可以使用ArrayBlockingQueue（基于数组结构的有界队列，FIFO），并指定队列的最大长度

使用有界队列可以防止资源耗尽，但也会造成超过队列大小和maximumPoolSize后，提交的任务被拒绝的问题，比较难调整和控制。

**threadFactory**

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名

**Static class DefaultThreadFactory implements ThreadFactory{**

**Private static final AtomicInteger poolNum = new AtomicInteger(1);**

**Private static ThreadGroup group;**

**Private static AtomicInteger threadNum = new AtomicInteger(1);**

**Private final String namePrex;**

**DefaultThreadFacotry{**

**SecutryManager s = System.getSecutryManager();**

**group = (s!=null)?s.getThreadGroup:Thread.currentThread().getThreadGroup();**

**namePrex = “pool-”+poolNum.getAndIncreament();+”-thread-”;**

**}**

**Public Thread newThread(Runnable r){**

**Thread t = new Thread(group,r,namePrex+threadNum.getAndIncrement(),0);**

**if(t.isDamon()){**

**t.setDamon(false);**

**}**

**if(t.getPriority()!=Thread.NORM\_PRIORITY){**

**t.setPriority(Thread.NORM\_PRIORITY);**

**}**

**return t;**

**}**

**}**

Executors静态工厂里默认的threadFactory

**RejectedExecutionHandler（饱和策略）**

**当阻塞队列满了，且没有空闲的工作线程的时候，如果继续提交任务，必须采取该任务**

**1：AbortPolicy。直接抛出异常，默认策略**

**2：CallerRunsPolicy 用调用者的线程来执行**

**3：DiscardOldestPolicy 丢弃阻塞队列中最前的任务，执行当前任务**

**4：DiscardPolicy 直接丢弃任务**

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

**2：ThreadPoolExecutor线程池执行流程**

**当试图通过execute方法将runnable加入到线程池中的时候**