目录

[Semaphore 信号量,信号系统 2](#_Toc29192)

[Exchanger<V> 泛型V数据类型 4](#_Toc25519)

[CountDownLatch 5](#_Toc24123)

[CyclicBarrier 6](#_Toc19454)

[Phaser 7](#_Toc18090)

[Executor (线程执行器) 8](#_Toc13883)

[ThreadPoolExecutor 11](#_Toc1661)

[Callable 和Future 15](#_Toc16327)

[Future: 15](#_Toc16614)

[CompletionService 17](#_Toc29955)

[ExecutorService 18](#_Toc14064)

[ScheduledExecutorService 19](#_Toc2126)

**Java并发编程核心方法与框架**

读高洪岩<Java并发编程核心方法与框架> 一书 所作总结

并发编程的类来自 java.util.concurrent 包

# Semaphore 信号量,信号系统

此类作用:实现限制线程并发数

Semaphore semaphore =new Semaphore(int permits)

这个构造方法的作用:同一时间内 最多允许 都少个线程 同时执行acquire和release 两方法之间的代码

semaphore.acquire();

.............

semaphore.release();

方法:

acquire(int permits): 每调用1次该方法,就使用 permits个许可

availablePermits():返回此Semaphore对象当前可用的许可数 许可数 实时在改变

acquireUninterruptibly(): 使等待进入acquire方法的线程,不允许被中断.

availablePermits(): 返回Semaphore对象中当前可用的许可数

drainPermits(): 获取并返回立即可用的所有许可个数,并且将可用许可置0

getQueueLength(): 获取等待许可的线程个数

hasQueuedThreads(): 判断有没线程在等待这个许可

公平和非公平信号量

公平信号量：获得锁的顺序与线程启动顺序有关（线程先启动，先获得许可）

非公平信号量：线程先启动不代表 先获得许可

Semaphore semaphore =new Semaphore(1,true); true: 公平 false: 非公平

tryAcquire 尝试获得一个许可，获取不到返回false

tryAcquire(int permits) 尝试地获得x个许可,获取不到 返回false

tryAcquire(long timeout,TimeUnit unit) 在指定的时间内,尝试获得一个许可,获取不到 返回false

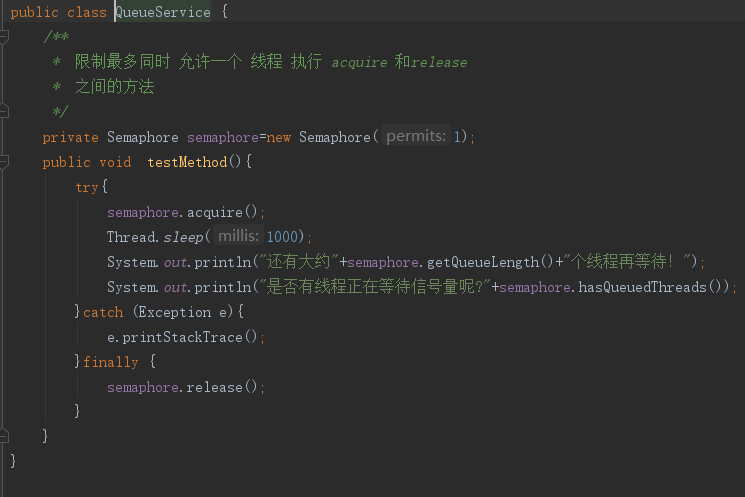
tryAcquire(int permits,long timeout,TimeUnit unit) 在指定的时间内,尝试获得X个许可,获取不到 返回false

多进路-多处理-多出路:允许多个线程同时处理任务(每个线程都在处理自己的任务)

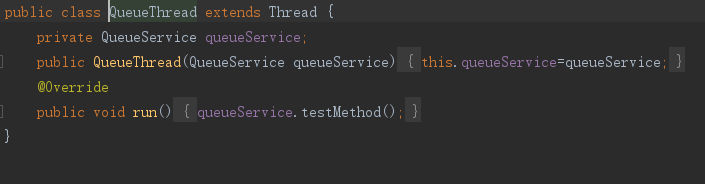
多进路-单处理-多出路: 允许多个线程同时处理任务,但执行任务的顺序是同步的(阻塞的)

代码示例

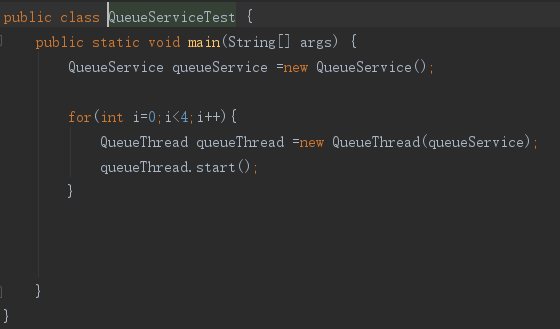
Service



任务



测试



# Exchanger<V> 泛型V数据类型

**功能 : 使2个线程之间传递任意类型的数据**

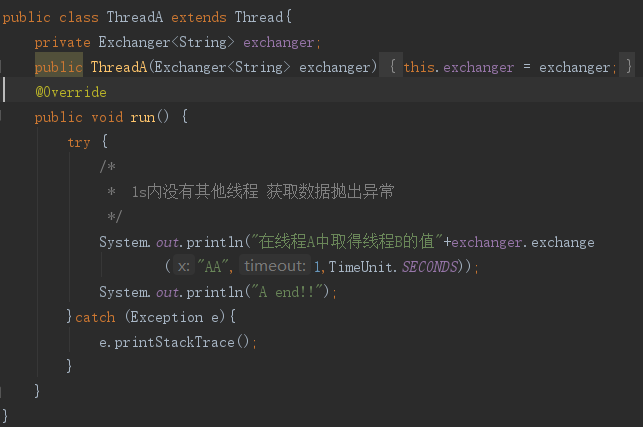
方法:

exchange() 方法,具有阻塞特性,此方法调用后,等待其他线程来取数据,若没有其他线程取得数据,则一直阻塞等待

exchange(V x,long timeout, TimeUnit unit) 在指定的时间内没有其他线程获取数据,抛出超时异常

V :数据 timeout :指定时间 TimeUnit: 时间单位

代码示例







# CountDownLatch

可以实现：一个或者多个线程，等待另外一个线程或多个线程完成某件事之后才能继续执行

控制线程执行任务的时机,使线程以”组团”的方式一起执行任务

提供的功能,判断count计数不为0时,当前线程呈wait状态

方法:

await() 调用此方法时 判断计数是否为0 不为0 当前线程陷入等待 当计数器为0 陷入等待的线程继续允许运行

countDown(): 其他线程调用该方法时,技术器 -1

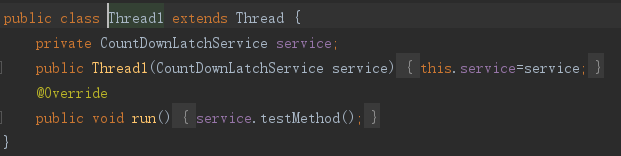
getCount() : 获得当前的计数个数

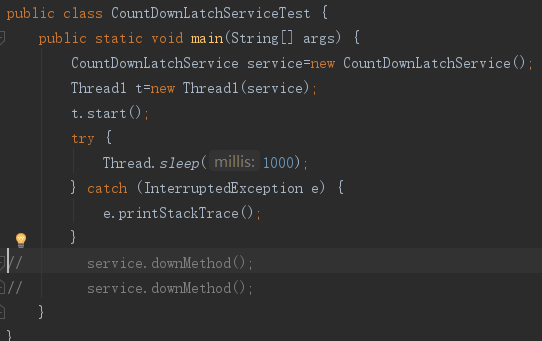
await(long timeout, TimeUnit unit) :在达到指定时间时，计数器即使不为 0 也会唤醒该线程

计数器 只要 为0 无论 达没达到 指定时间 都会唤醒 等待线程

代码示例







# CyclicBarrier

CyclicBarrier： 多个线程之间相互等待，

构造方法:CyclicBarrier(int parties, Runnable barrierAction), CyclicBarrier(int parties)

parties： 同行者

最大设置 n个同行者，也就是n个线程执行了 CyclicBarrier对象的 await方法 后程序才能继续向下运行

大于 n 个线程的话 时分批处理的，也就是 n 个线程 通过后 接着的线程要重新组队，达到n个后 才可以 继续

await() 调用该方法的线程数量达到 n (同行者) 时 才能继续往下运行 (屏障)

await(long timeout, TimeUnit unit) : 在指定时间内达到 parties数量，程序继续运行，否则抛出TimeoutException

getNumberWaiting(): 获得有几个线程 已经达到屏障点(也就是几个线程正在陷入阻塞) 当有 n个线程（n 为设置的同行者的个数）

getParties(): 获取 parties个数

reset() 重置屏障

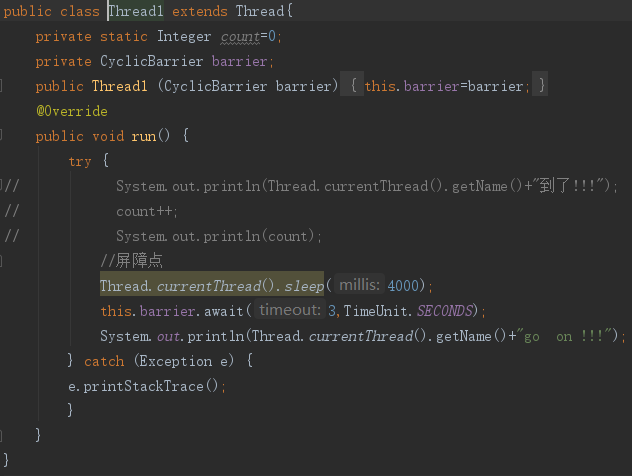
isBroken() : 查询此屏障点 是否处于损坏状态

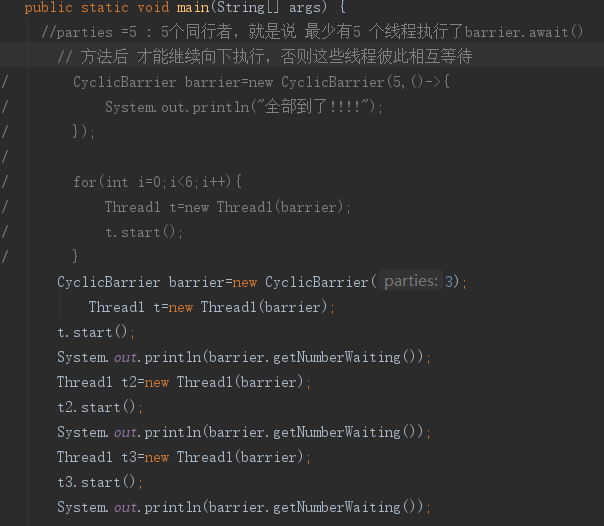
Thread 的 interrupte 方法，使线程处于中断状态，

如果一个线程 由于中断或者超时提前离开屏障点，其他所有在屏障点等待的线程会抛出

BrokenBarrierException 或者 InterruptedException 异常

代码示例





# Phaser

Phaser: 可以设置多个屏障

arriveAndAwaitAdvance() : 作用:当前线程已经达到屏障,在此等待一段时间,条件满足后向下一屏障继续执行

arriveAndDeregister(): 撤销当前线程,并且使parties -1

getPhase(): 返回已经到达第几个屏障

onAdvance(): 通过新的屏障时被调用 该方法返回true,线程不在等待,Pharse 呈无效状态

返回false 则 Pharse 继续工作

getRegisteredParties(): 获取注册的patties数量

register(): 每次执行一次该方法,就动态添加一个 parties

bulkRegister(int parties): 批量增加 parties 数量

getArrivedParties(): 获得已被使用的parties个数

getUnarrivedParties(): 获得未被使用的parties个数

arrive(): 使parties 值加1,并且不在屏障处等待,接着运行,并且Phaser类有计数重置的功能

awaitAdvance(int phase): 如果传入参数 phase的值和 当前getPhase() 返回值一样,则在屏障出等待,否则继续向下运行

awaitAdvanceInterruptibly(int phase):当线程 跨过的屏障数 不等于 phase 参数 继续向下执行,否则陷入等待

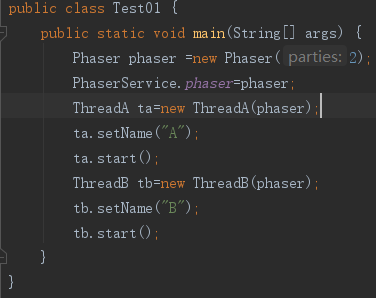
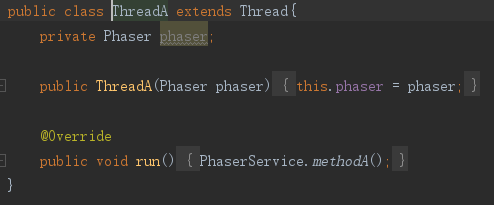
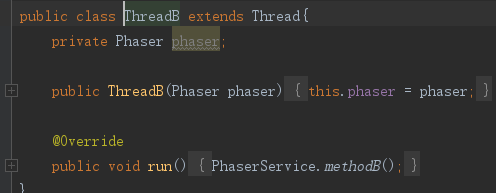
awaitAdvanceInterruptibly(int phase,long timeout, TimeUnit unit):

在指定的屏障数,在指定的时间内,如果屏障数未变 抛出异常

forceTermination(): 时Phaser 对象的屏障功能失效

isTerminated(): 判断Phaser对象是否已经呈销毁状态





# Executor (线程执行器)

与 线程池 有关的大部分类都是实现此接口的, 官方建议使用Executors 工厂类来创建线程池对象

使用 newCachedThreadPool 方法创建无界线程池,可以进行线程自动回收.

池中存放线程个数是理论上的Integer.MAX\_VALUE最大值

线程池中的线程对象可以复用

execute(); 调用Runnable(任务)的run方法

ExecutorService service = Executors.newCachedThreadPool ();

service.execute(Runnable r1)

service.execute(Runnable r1)

.....

可以 执行多个线程,线程之间是异步运行的 (执行效率高)

newFixedThreadPool(int nThreads) 方法创建有界线程池

使用有界线程池 后线程池中最多线程个数是可控的

ExecutorService service = Executors.newFixedThreadPool(int nThreads);

同一时间段,允许有 nThreads 个线程在池中被运行

/\*\*

\* 无界线程池中的Thread类还可以由程序员自己定制

\* newCachedThreadPool(factory); 就是解决此问题 (该方法定制线程工厂)

\* 通过自定义的 ThreadFactory接口实现类,实现了线程对象的定制

\*/

public class MyThreadFactory implements ThreadFactory {

@Override

public Thread newThread(Runnable r) {

Thread thread=new Thread(r);

thread.setName("thread定制");

// 线程池中出现异常,自定义处理

thread.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {

@Override

public void uncaughtException(Thread t, Throwable e) {

System.out.println("自定义异常:"+t.getName()+":"+e.getMessage());

e.printStackTrace();

}

});

return thread;

}

}

//采用ThradFactory 实现Thrad的定制

ExecutorService service = Executors.newFixedThreadPool(2, factory);

创建单一线程池,单一线程池可以实现以队列的方式执行任务

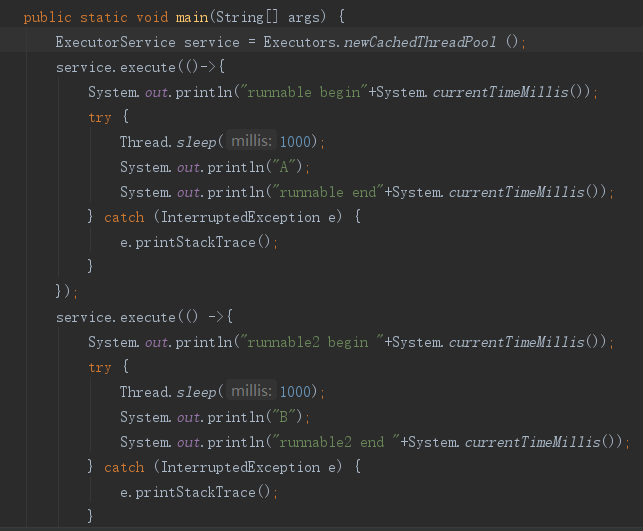
ExecutorService service = Executors.newSingleThreadExecutor()

ExecutorService service = Executors.newSingleThreadExecutor(factory)

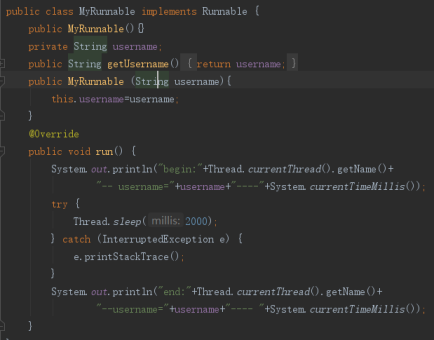
单一先程池,线程之间是 同步运行的,先执行的任务 运行完 才会执行下一个任

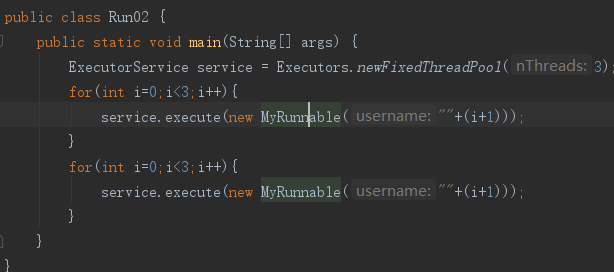
务

代码示例 newCachedThreadPool

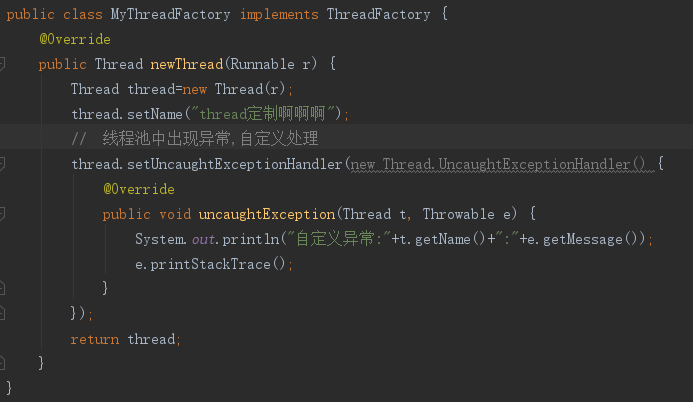


代码示例 newFixedThreadPool

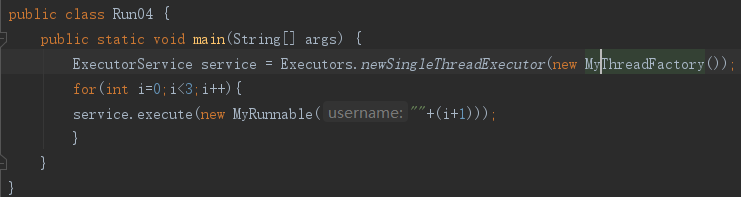




代码示例 工厂模式







# ThreadPoolExecutor

ThreadPoolExecutor :可以非常方便的创建线程池对象,而不需要程序员设计大量的new 实例化Thread 相关代码

其实Executors工厂类newXXXThreadPool 内部其实是实例化了一个ThreadPoolExecutor实例

最常用的 构造方法 (int corePoolSize,int maximumPoolSize, long keepAliveTime,

TimeUnit unit,BlockingQueue<Runnable> workQueue )

corePoolSize: 池中保存的线程数,包含空闲线程,也就是核心池的大小

maximumPoolSize: 池中允许的最大线程数

keepAliveTime: 当线程数量大于corePoolSize时,在没有超过指定时间内

不从线程池中将空闲线程删除的,如果超过此时间单位,则删除

unit: 参数的时间单位

workQueue:当执行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

prestartAllCoreThreads(): 启动全部核心线程,返回 启动核心线程的数量

prestartCoreThread(): 每调用一次就创建一个核心线程,返回值 boolean,是否启动了

超过 corePoolSize则 返回false ,也不创建线程

方法

submit(Runnable): 提交任务,其提交的任务 未被执行时,remove方法不能删除任务

remove(Runnable): 可以删除未被执行的 Runnable任务,这在执行的 Runnable任务不能删除

getActiveCount(): 获取 正在执行任务的线程数

getCompletedTaskCount(): 获取有多少个线程已经执行完任务了

//车中可载人的标准人数

getCorePoolSize(): 获取构造方法 传入的corePoolSize参数值

//车中可载人的最大人数

getMaximumPoolSize(): 获取构造方法传入的maximumPoolSize值

//车中正在载的人数

getPoolSize(): 获取池中有多少个线程

//扩展车中正在载的人数

getQueue().size()

getTaskCount(): 获取 有多少个任务发送给了线程池

接口Runnable在ThreadPoolExecutor 的队列中是按顺序取出,执行却是乱序的

ThreadPoolExecutor executor=new ThreadPoolExecutor

(7,8, 5,TimeUnit.SECONDS,new LinkedBlockingDeque<Runnable>());

队列使用 LinkedBlockingDeque 类

线程数量<=corePoolSize ,所以当超过指定时间时, 线程池中的空闲线程是不删除的

个人理解 大于 corePoolSize时 的多余线程为空闲线程

当线程数量>corePoolSize时,将其余的任务放入队列中

同一时间 最多有 corePoolSize个 线程运行

使用 LinkedBlockingDeque 类则 maximumPoolSize 参数作用将忽略

shutdown() : 作用是使当前未执行完的线程继续执行,而不再添加新的任务,该方法不会阻塞

调用之后主线程main 就马上停止,而线程池会继续运行直到所有任务执行完才会停止.

调用此方法后 线程池的状态 变为shutdown 状态 不能向线程池中添加任何任务

shutdownNow() 中断所有任务,并抛出异常

此方法使得线程池的状态立即变为stop状态,并且试图停止所有正在执行的线程

isShutdown(): 判断线程池是否已经关闭 调用了shutdown方法 isShutdown 就返回true

isTerminating() isTerminated()

做个比喻: shutdown() 或者shutdownNow() 发出一个关闭大门的命令,

isShutdown() 判断这个关闭大门的命令发出或未发出.

isTerminating() 代表大门是否正在关闭进行中,而 isTerminated()方法 判断大门是否已经关闭了

awaitTermination(long timeout,TimeUnit unit): 这个方法 不太懂

查看在指定时间之间,线程池是否已经终止工作,也就是最多等待多少时间后去判断线程池是否已经终止工作

队列使用 SynchronousQueue 类

线程数量<=corePoolSize ,所以当超过指定时间时, 线程池中的空闲线程是不删除的

当线程数量>corePoolSize时,将其余的任务也放入池中,目前总数为8

由于运行的线程数量为8 数量上>corePoolSize 7

所以超过指定时间(5s) 时清空空闲线程(超过 corePoolSize的线程)

使用 SynchronousQueue 类则 maximumPoolSize 参数作用将有效

队列使用 SynchronousQueue 线程数量>corePoolSize 并且大于 maximumPoolSize 则会抛出异常 (运行了多余的任务)

如果将 keepAliveTime 设置 为0

则任务执行完毕之后立即将从队列中删除 并且把 空闲线程 从线程池中删除

自定义 拒绝策略:

RejectedExecutionHandler : 当线程池关闭(executor.shutdown())后 依然有任务要执行时,可以实现一些处理

public class MyRejectedExecutionHandler implements RejectedExecutionHandler {

@Override

public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {

System.out.println(r+"被拒绝了!");

}

}

executor.setRejectedExecutionHandler(new MyRejectedExecutionHandler());

工厂 ThreadFactory +execute() + UncaughtExceptionHandler 处理异常

ThreadPoolExecutor pool= new ThreadPoolExecutor

(2,9999,9999L,TimeUnit.SECONDS,new LinkedBlockingDeque<Runnable>(),factory);

ThreadPoolExecutor pool=new ThreadPoolExecutor(2,3,

3,TimeUnit.SECONDS,new SynchronousQueue<>());

allowCoreThreadTimeOut(boolen) 这只配置核心线程是否 可以超时

true的话 超时就 清除 corePoolSize的任务

allowsCoreThreadTimeOut : 获取是否允许 超时

pool.setRejectedExecutionHandler((r,e)->{

System.out.println(((MyRunnable)r).getUsername()+" 被拒绝运行!");

});

当运行的任务 大于 maximumPoolSize时 拒绝运行多余的任务 抛出异常

set/getRejectedExecutionHandler:作用 处理任务被拒绝执行的 行为

常见 3 种队列,结合max值的效果 (实验环境 欲执行的任务数量大于线程池的 maximumPoolSize)

ArrayBlockingQueue LinkedBlockingDeque SynchronousQueue

LinkedBlockingDeque: 构造(int capacity) 使用 有参构造,maximumPoolSize值参考

使用无参构造, maximumPoolSize值参考被忽略.(最多执行corePoolSize任务)

capacity<=0 时抛出异常 (capacity :队列能容纳的任务数量)

分配策略: 假设 capacity=2,corePoolSize=2,maximumPoolSize=3

欲执行的 任务 为 4 则 分配 执行任务2, 放入队列任务2

5 3 2

6 3 2

(6,3,2)队列容量不够,有一个任务出现异常

ArrayBlockingQueue 和LinkedBlockingDeque 效果差不多

public class MyThreadPoolExecutor extends ThreadPoolExecutor {

public MyThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue){

super(corePoolSize,maximumPoolSize,keepAliveTime,unit,workQueue);

}

@Override

protected void beforeExecute(Thread t, Runnable r) {

super.beforeExecute(t, r);

System.out.println("准备执行: "+ ((MyRunnable) r).getUsername());

}

@Override

protected void afterExecute(Runnable r, Throwable t) {

super.afterExecute(r, t);

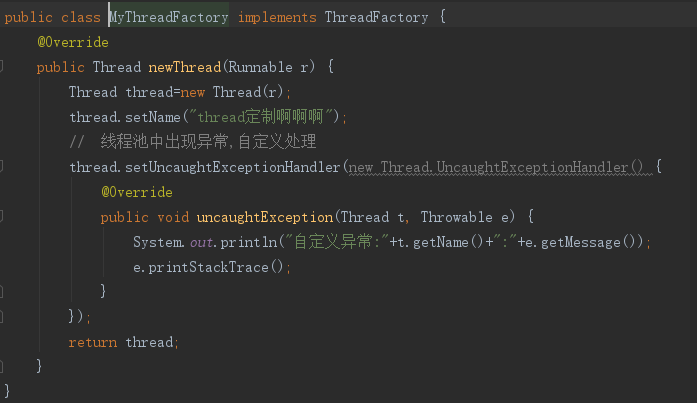
System.out.println( ((MyRunnable) r).getUsername()+" 执行完了");

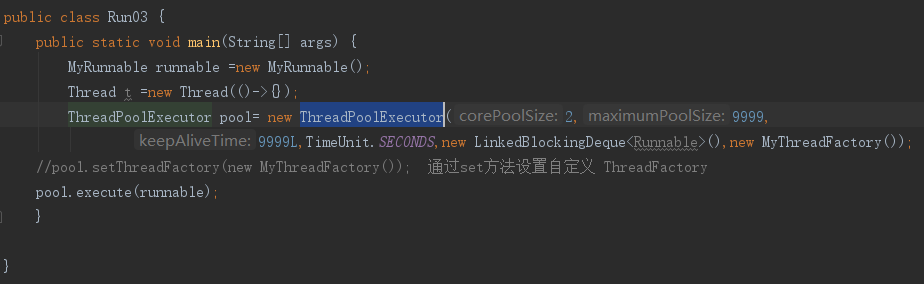
}

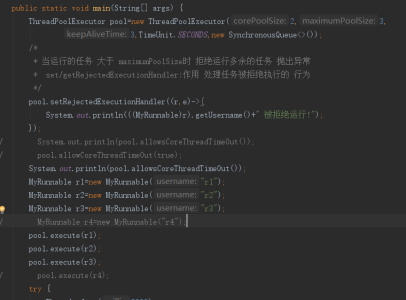
}

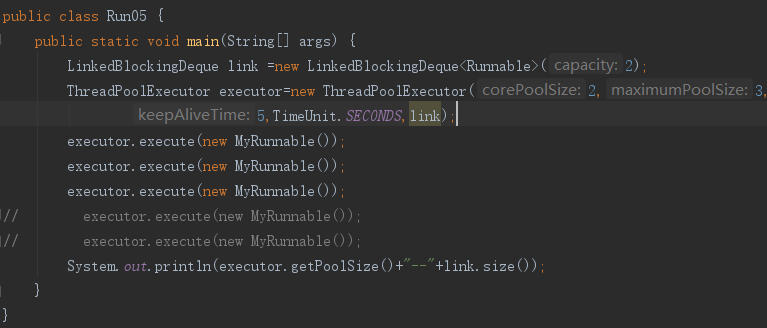
重写 beforeExecute和 afterExecute 方法可以对 线程池中执行的线程对象进行监控

代码示例 工厂模式









# Callable 和Future

Callable 类似与 Runnable 是一个任务线程

在默认情况下 Thread 对象 Runnable 的 run方法 不具有返回值的功能

用Future 和 Callable 来使线程具有返回值功能

接口 Callable与线程功能密不可分 和Runnable的区别

Callable的 call() 方法可以有返回值 Runnable的 run()方法没有返回值

call() 方法可以声明抛出异常 run()方法不可以

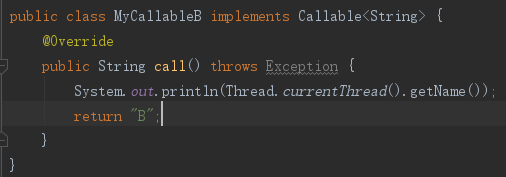
执行完Callable 接口中的任务 后 ,返回值 通过 Future 接口进行获得

ThreadPoolExecutor 的 submit 方法

可以 提交(执行) Runnable任务,也可以提交 Callable任务.

submit() 方法的返回值是 Future接口 的实例 FutureTask

Callable 的定义



## Future:

get() : 获取 执行任务的 返回结果,该方法是阻塞的(任务的 call() 方法阻塞,导致 该方法阻塞) 换句话说,如果 执行了 get方法,代表线程任务已经运行完毕

cancel(boolean ):该方法是不阻塞的.

参数作用:如果线程正在运行,则是否中断正在运行的线程

(参数为true,则发送命令请求中断线程 参数为 false,则发送命令不中断线程)

在代码中 需要 使用 if(Thread.currentThread().isInterrupted()) 进行配合

方法的返回值 代表 发送的取消命令是否成功

isCancelled() : 如果 命令执行成功返回 true

get(long timeout, TimeUnit unit):在指定最大时间内,等待获得返回值 ,超时则 抛出异常

ExecutorService 的 execute和 submit方法的区别

execute 没有返回值 .submit 有返回值

execute 在默认情况下异常直接抛出,不能捕获,但可以通过自定义的ThreadFactory 的方式进行捕获

submit 在默认情况下,可以 try catch (catch 捕获异常)

Future的缺点:其 get() 方法取得处理结果时,是阻塞性的,

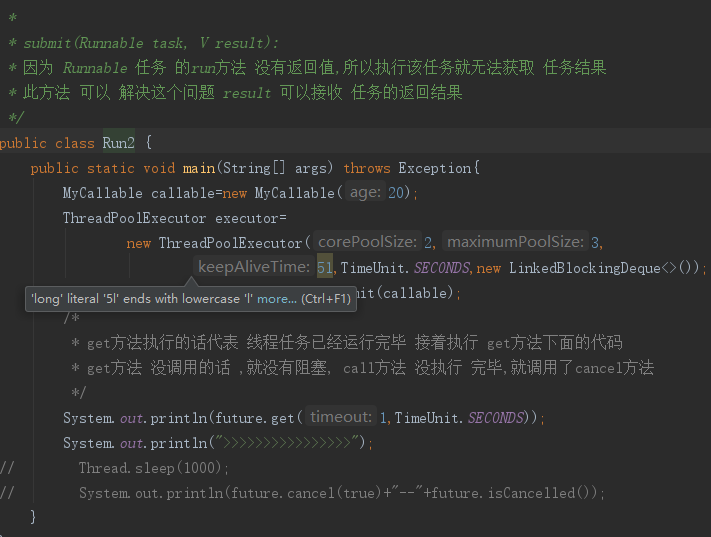
就是说,调用get时,任务尚未完成,则调用get方法时一直阻塞到任务完成为止.

若果前面的任务耗时很多.后面的任务 调用 get方法 就呈阻塞状态,排队进行等待,大大影响运行效率

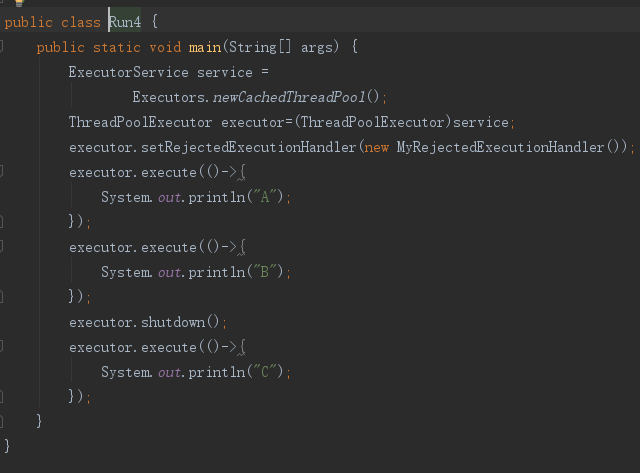
CompletionService 接口解决这个问题

代码示例









## CompletionService

ExecutorService service = Executors.newSingleThreadExecutor();

CompletionService cs =new ExecutorCompletionService(service);

CompletionService:以异步的方式一边生产新任务,一边处理已完成任务的结果

这样可以将执行任务与处理任务分离开来进行处理.

使用submit执行任务,使用take 取得已经完成的任务(Future)

CompletionService 仅有一个实现类 ExecutorCompletionService

从构造方法 可知 其依赖 Executor对象

CompletionService解决了 Future get()方法 的阻塞特性

哪个任务先完成了,哪个任务的返回值就先获取,

如果当前没有任务先执行完,则 take().get() 方法还是呈阻塞特性

take() 方法 获取最先完成任务的Future 对象,谁执行时间最短,谁最先返回

poll():获取并移除 表示下一个已完成任务的Future,如果不存在这样的任务,则返回null,如果 调用 poll 方法时 没有 任务 完成 ,则返回null

poll(long timeout, TimeUnit unit):

在指定时间之内,获取到 已完成的任务 立即向下运行,超时也立即向下运行

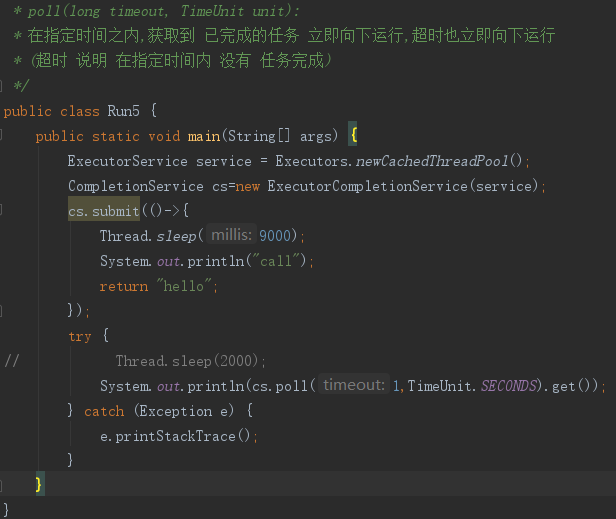
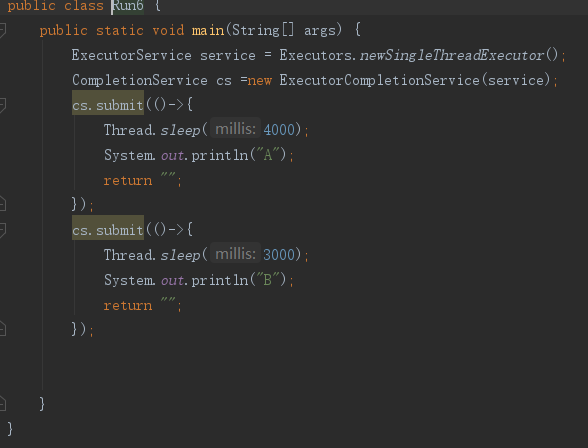
(超时 说明 在指定时间内 没有 任务完成)

submit(Callable<V> task):执行 Callable任务

submit(Runnable task,V result): 执行 Runnable任务,

submit(Runnable task, V result):

因为 Runnable 任务 的run方法 没有返回值,所以执行该任务就无法获取 任务结果.此方法 可以 解决这个问题 result 可以接收 任务的返回结果



# ExecutorService

invokeAny(): 方法具有阻塞特性,该方法取得第一个完成任务的结果值,当第一个任务执行完成后,

会调用 interrupt() 方法将其他任务中断.所以在这些任务中可以结合

if(Thread.currentThread().isInterrupted()==true) 代码决定任务是否继续运行

invokeAny(Collection tasks) :Collection集合 中盛放的是任务,该方法的执行具有不确定性

invokeAny() 与执行慢的任务异常,默认情况下,执行慢的任务出现异常时,不会在控制台输出异常信息,

如果显示使用 try catch 语句块,则可以自定义捕获异常,通过 e.printStackTrace():

显示的打印异常信息

invokeAny() 与执行快的任务异常:执行快的任务出现异常时,默认情况下不在控制台输出异常信息.

除非显示的使用try catch,等待执行慢的任务返回的结果值.

执行快的 任务 出现异常时,要在catch 代码中重新 throw 异常,这样主线程才能获取执行慢的线程的返回结果

当任务都出现异常时,返回最后一个异常并输出

invokeAny(Collection tasks,timeout ,unit): 在指定时间内取得第一个先执行完任务的结果值

在出现超时异常时,可以if(Thread.currentThread().isInterrupted()==true) 判断和 throw 异常,

来时 线程中断执行

若一个任务,即超时,又出现异常,则 TimeoutException 和 另外一个异常同时捕获

invokeAll() :等待全部线程任务执行完毕后,取得全部任务的结果值

该方法 是阻塞的. 该方法是建立在Future 接口上的

invokeAll(Collection tasks) :

该方法是阻塞的, 只有把所有任务的结果都取回时,才继续向下运行

invokeAll() 方法对Callable跑出去的异常是可以处理的

如果使用invokeAll 方法而某一个任务正确的返回值时,则其他Callable抛出去

的异常 在main方法 中是不被处理的.

如果都没有正确的返回值时,则最后Callable抛出去的异常是被 main 处理的

invokeAll(Collection tasks,timeout,unit):

如果全部任务在指定的时间内没有完成,则出现异常

# ScheduledExecutorService

计划任务

类ScheduledExecutorService的主要作用：可以将定时任务与线程池功能结合使用

\* ScheduledThreadPoolExecutor 是 ScheduledExecutorService 的实现类

\* ScheduledThreadPoolExecutor 的父类是 ThreadPoolExecutor

\* schedule(Callable<V> c,long delay,TimeUnit unit): 延迟运行 任务

\* delay 参数,在多个任务中同时消耗时间,(假如2 个任务 的delay 都是 4 ,则 4秒后,两个任务可以同时运行)

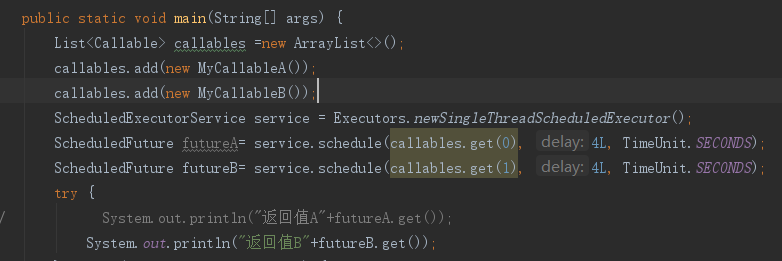
\* newSingleThreadScheduledExecutor() 一次只能执行一个任务

\* newScheduledThreadPool(int poolSize) :创建 能同时执行多个任务的线程池 ,参数为1 时,任务池还是单任务的

\* 也就是 能异步运行 指定 个数的任务

\* schedule(Runnable command, long delay, TimeUnit unit): 延迟运行 Runnable任务

代码示例



scheduleAtFixedRate(Runnable command, long initialDelay,

long period, TimeUnit unit);

该方法 实现周期性执行任务, period : 预定的周期时间 (每隔 这么长时间执行一次任务)

当执行任务的时间> period 时,产生超时效果

该方法不具有获得返回值的功能

scheduleWithFixedDelay(Runnable command,long initialDelay,

long delay, TimeUnit unit)

此方法同样实现 任务的周期性运行,但是没有超时与非超时的情况

initialDelay: 首次执行任务 所等待的时间

delay : 上一个任务的结束时间 和 下一个 任务的开始的时间间隔

getQueue(): 获取队列中的任务,这些任务时未来将要运行的,正在运行的任务不在此队列中

使用 scheduleAtFixedRate 和 scheduleWithFixedDelay 周期性执行任务时,未来要执行的任务都是要放入此队列中的

ScheduledThreadPoolExecutor :的remove() 方法,删除队列中的任务

ScheduledThreadPoolExecutor :执行 shutdown() 方法时,默认返回true,任务还是继续运行的

setExecuteExistingDelayedTasksAfterShutdownPolicy( false) : 任务不再运行

setContinueExistingPeriodicTasksAfterShutdownPolicy() :

传入true 时,当使用 scheduleAtFixedRate 和 scheduleWithFixedDelay 方法 时,如果调用ScheduledThreadPoolExecutor

对象的 shutdown 方法,任务还会继续运行.传入 false 任务不再运行,进程销毁

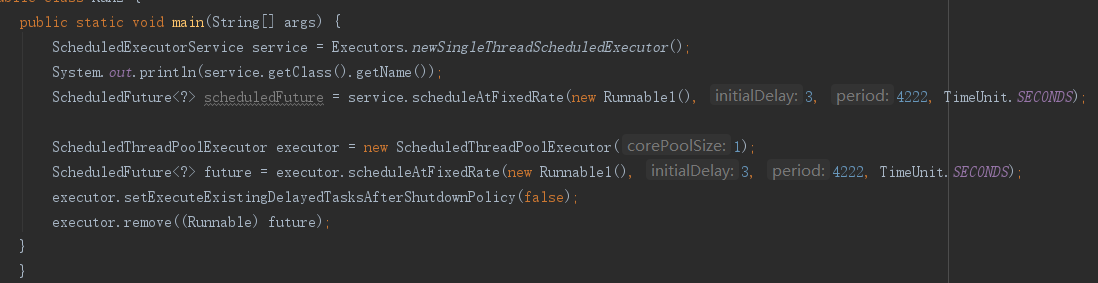
当 使用scheduleAtFixedRate 和 scheduleWithFixedDelay 方法 时,如果调用 shutdown,进程销毁

ScheduledFuture 的 cancel() 方法 传入参数为true时, 取消再任务队列中 相应任务的执行,但是 任务 还保留再任务队列中

cancel 也可以 终止 正在执行的任务,但要结合 if(Thread.currentThread().isInterrupted()==true) 判断

ScheduledThreadPoolExecutor 的 serRemoveOnCancelPolicy() 传入参数为 true时,将 取消的任务(cancel) 从队列中删除

代码示例



# 

# 