一、异步任务执行服务

1. 基本介绍

线程Thread既表示要执行的任务，又表示执行的机制，而这套框架引入了一个"执行服务"的概念，它将"任务的提交"和"任务的执行"相分离，"执行服务"封装了任务执行的细节，对于任务提交者而言，它可以关注于任务本身，如提交任务、获取结果、取消任务，而不需要关注任务执行的细节，如线程创建、任务调度、线程关闭等。

2. 基本接口

2.1 任务执行服务涉及的接口

(1) Runnable和Callable： 表示要执行的异步任务

Executor和ExecutorService： 表示执行服务

Future： 表示异步任务的结果

2.2 Runnable和Callable

两者都表示任务，Runnable没有返回结果，而Callable有，Runnable不会抛出异常，而Callable会。

2.3 Executor和ExecutorService

(1) Executor -- 表示最简单的执行服务

public interface Executor {

void execute(Runnable command);

}

可以执行一个Runnable，没有返回结果。接口没有限定任务如何执行，可能是创建一个新线程，可能是复用线程池中的某个线程，也可能是在调用者线程中执行。

(2) ExecutorService -- 扩展了Executor，定义了更多服务

public interface ExecutorService extends Executor {

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

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

Future<?> submit(Runnable task);

//... 其他方法

}

① 含义：三个submit都表示提交一个任务，返回值类型都是Future，返回后，只是表示任务已提交，不代表已执行，通过Future可以查询异步任务的状态、获取最终结果、取消任务等。

② 对于Callable，任务最终有个返回值，而对于Runnable是没有返回值的，第二个提交Runnable的方法可以同时提供一个结果，在异步任务结束时返回，而对于第三个方法，异步任务的最终返回值为null。

2.4 Future

(1) 接口定义

public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

boolean isDone();

V get() throws InterruptedException, ExecutionException;

V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException,

TimeoutException;

}

① get用于返回异步任务最终的结果，如果任务还未执行完成，会阻塞等待，另一个get方法可以限定阻塞等待的时间，如果超时任务还未结束，会抛出TimeoutException。

② cancel用于取消异步任务，如果任务已完成、或已经取消、或由于某种原因不能取消，cancel返回false，否则返回true。如果任务还未开始，则不再运行。但如果任务已经在运行，则不一定能取消，参数mayInterruptIfRunning表示，如果任务正在执行，是否调用interrupt方法中断线程，如果为false就不会，如果为true，就会尝试中断线程。

③ isDone和isCancelled用于查询任务状态。isCancelled表示任务是否被取消，只要cancel方法返回了true，随后的isCancelled方法都会返回true，即使执行任务的线程还未真正结束。isDone表示任务是否结束，不管什么原因都算，可能是任务正常结束、可能是任务抛出了异常、也可能是任务被取消。

④ get方法任务最终大概有三个结果：

a. 正常完成，get方法会返回其执行结果，如果任务是Runnable且没有提供结果，返回null。

b. 任务执行抛出了异常，get方法会将异常包装为ExecutionException重新抛出，通过异常的getCause方法可以获取原异常

c. 任务被取消了，get方法会抛出异常CancellationException

d. 如果调用get方法的线程被中断了，get方法会抛出InterruptedException。

(2) Future是一个重要的概念，是实现"任务的提交"与"任务的执行"相分离的关键，是其中的"纽带"，任务提交者和任务执行服务通过它隔离各自的关注点，同时进行协作。

3. 基本用法

3.1 实例代码

public class BasicDemo {

//内部类实现任务接口Callable

static class Task implements Callable<Integer> {

@Override

public Integer call() throws Exception {

int sleepSeconds = new Random().nextInt(1000);

Thread.sleep(sleepSeconds);

return sleepSeconds;

}

}

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

//创建任务执行服务ExecutorService，表示使用一个线程执行所有服务

// Executors，注意与Executor相区别，后者是单数，是接口。

ExecutorService executor = Executors.newSingleThreadExecutor();

//任务执行结果Future

Future<Integer> future = executor.submit(new Task());

// 模拟执行其他任务，当前线程睡眠

Thread.sleep(100);

try {

//打印任务执行结果

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

} catch (ExecutionException e) {

e.printStackTrace();

}

//关闭任务执行服务

executor.shutdown();

}

}

3.2 任务执行服务类ExecutorService

(1) 接口定义

public interface ExecutorService extends Executor {

//关闭方法

void shutdown();

//关闭方法

List<Runnable> shutdownNow();

注意：有两个关闭方法，shutdown和shutdownNow，区别是，shutdown表示不再接受新任务，但已提交的任务会继续执行，即使任务还未开始执行，shutdownNow不仅不接受新任务，已提交但尚未执行的任务会被终止，对于正在执行的任务，一般会调用线程的interrupt方法尝试中断，不过，线程可能不响应中断，shutdownNow会返回已提交但尚未执行的任务列表。

// shutdown和shutdownNow不会阻塞等待，它们返回后不代表所有任务都已结束，不过isShutdown方法会返回true。

boolean isShutdown();

//调用者可以通过awaitTermination等待所有任务结束，它可以限定等待的时间，如果超时前所有任务都结束了，即isTerminated方法返回true，则返回true，否则返回false。

boolean isTerminated();

//等待所有任务结束

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

//批量提交任务方法invokeAll

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)

throws InterruptedException;

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,

long timeout, TimeUnit unit)

throws InterruptedException;

//批量提交任务方法invokeAny

<T> T invokeAny(Collection<? extends Callable<T>> tasks)

throws InterruptedException, ExecutionException;

<T> T invokeAny(Collection<? extends Callable<T>> tasks,

long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException;

}

① invokeAll等待所有任务完成，返回的Future列表中，每个Future的isDone方法都返回true，不过isDone为true不代表任务就执行成功了，可能是被取消了，invokeAll可以指定等待时间，如果超时后有的任务没完成，就会被取消。

② 而对于invokeAny，只要有一个任务在限时内成功返回了，它就会返回该任务的结果，其他任务会被取消，如果没有任务能在限时内成功返回，抛出TimeoutException，如果限时内所有任务都结束了，但都发生了异常，抛出ExecutionException。

(2) 批量提交任务invokeAll实例

public class InvokeAllDemo {

//任务类UrlTitleParser，实现Callable接口

static class UrlTitleParser implements Callable<String> {

private String url;

public UrlTitleParser(String url) {

this.url = url;

}

@Override

public String call() throws Exception {

Document doc = Jsoup.connect(url).get();

Elements elements = doc.select("head title");

if (elements.size() > 0) {

return elements.get(0).text();

}

return null;

}

}

public static void main(String[] args) {

//创建一个线程池

ExecutorService executor = Executors.newFixedThreadPool(10);

String url1 = "http://www.cnblogs.com/swiftma/p/5396551.html";

String url2 = "http://www.cnblogs.com/swiftma/p/5399315.html";

Collection<UrlTitleParser> tasks = Arrays.asList(new UrlTitleParser[] {

new UrlTitleParser(url1), new UrlTitleParser(url2) });

try {

List<Future<String>> results = executor.invokeAll(tasks, 10,

TimeUnit.SECONDS);

//打印任务执行结果

for (Future<String> result : results) {

try {

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

} catch (ExecutionException e) {

e.printStackTrace();

}

}

} catch (InterruptedException e) {

e.printStackTrace();

}

//关闭任务执行服务

executor.shutdown();

}

}

① 注意：使用ExecutorService，编写并发异步任务的代码就像写顺序程序一样，不用关心线程的创建和协调，只需要提交任务、处理结果就可以了，大大简化了开发工作。

二、线程池

1. 基本介绍

(1) 线程池，顾名思义，就是一个线程的池子，里面有若干线程，它们的目的就是执行提交给线程池的任务，执行完一个任务后不会退出，而是继续等待或执行新任务。线程池主要由两个概念组成，一个是任务队列，另一个是工作者线程，工作者线程主体就是一个循环，循环从队列中接受任务并执行，任务队列保存待执行的任务。

(2) 优点：

① 可以重用线程，避免线程创建的开销

② 在任务过多时，通过排队避免创建过多线程，减少系统资源消耗和竞争，确保任务有序完成。

(3) Java并发包中线程池的实现类是ThreadPoolExecutor，它继承自AbstractExecutorService，实现了ExecutorService。

2. 线程池实现类 java.util.concurrent.ThreadPoolExecutor

2.1 构造方法

(1) public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,

TimeUnit unit, BlockingQueue<Runnable> workQueue)

① corePoolSize - 即使空闲时仍保留在池中的线程数，除非设置

allowCoreThreadTimeOut

② maximumPoolSize - 池中允许的最大线程数

③ keepAliveTime - 当线程数大于核心时，这是多余的空闲线程在终止之前等待新任

务的最大时间。

④ unit – keepAliveTime 参数的时间单位

⑤ workQueue - 在执行任务之前用于保存任务的队列。 该队列将仅保存execute

方法提交的Runnable任务。

(2) public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,

TimeUnit unit, BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

threadFactory - 执行程序创建新线程时使用的工厂

handler - 执行被阻止时使用的处理程序，因为达到线程限制和队列容量

2.2 线程池大小

(1) 线程池的大小主要与四个参数有关：

corePoolSize：核心线程个数

maximumPoolSize：最大线程个数

keepAliveTime和unit：空闲线程存活时间

① maximumPoolSize表示线程池中的最多线程数，线程的个数会动态变化，但这是最大值，不管有多少任务，都不会创建比这个值大的线程个数。

② corePoolSize表示线程池中的核心线程个数，不过，这并不是说，一开始就创建这么多线程，刚创建一个线程池后，实际上并不会创建任何线程。

③ 一般情况下，有新任务到来的时候，如果当前线程个数小于corePoolSiz，就会创建一个新线程来执行该任务，需要说明的是，即使其他线程现在也是空闲的，也会创建新线程。

不过，如果线程个数大于等于corePoolSiz，那就不会立即创建新线程了，它会先尝试排队，需要强调的是，它是"尝试"排队，而不是"阻塞等待"入队，如果队列满了或其他原因不能立即入队，它就不会排队，而是检查线程个数是否达到了maximumPoolSize，如果没有，就会继续创建线程，直到线程数达到maximumPoolSize。、

④ keepAliveTime的目的是为了释放多余的线程资源，它表示，当线程池中的线程个数大于corePoolSize时，额外空闲线程的存活时间，也就是说，一个非核心线程，在空闲等待新任务时，会有一个最长等待时间，即keepAliveTime，如果到了时间还是没有新任务，就会被终止。如果该值为0，表示所有线程都不会超时终止。

(2) 参数设定方法

public void setCorePoolSize(int corePoolSize)

public int getCorePoolSize()

public int getMaximumPoolSize()

public void setMaximumPoolSize(int maximumPoolSize)

public long getKeepAliveTime(TimeUnit unit)

public void setKeepAliveTime(long time, TimeUnit unit)

(3) 其他静态参数

//返回当前线程个数

public int getPoolSize()

//返回线程池曾经达到过的最大线程个数

public int getLargestPoolSize()

//返回线程池自创建以来所有已完成的任务数

public long getCompletedTaskCount()

//返回所有任务数，包括所有已完成的加上所有排队待执行的

public long getTaskCount()

2.2 队列

(1) ThreadPoolExecutor要求的队列类型是阻塞队列BlockingQueue，

LinkedBlockingQueue：基于链表的阻塞队列，可以指定最大长度，但默认是无界的。

ArrayBlockingQueue：基于数组的有界阻塞队列

PriorityBlockingQueue：基于堆的无界阻塞优先级队列

SynchronousQueue：没有实际存储空间的同步阻塞队列

(2) 如果用的是无界队列，需要强调的是，线程个数最多只能达到corePoolSize，到达corePoolSize后，新的任务总会排队，参数maximumPoolSize也就没有意义了。

(3) 对于SynchronousQueue，我们知道，它没有实际存储元素的空间，当尝试排队时，只有正好有空闲线程在等待接受任务时，才会入队成功，否则，总是会创建新线程，直到达到maximumPoolSize。

2.3 任务拒绝策略

(1) 背景：

如果队列有界，且maximumPoolSize有限，则当队列排满，线程个数也达到了maximumPoolSize，这时，新任务来了，如何处理呢？此时，会触发线程池的任务拒绝策略。

默认情况下，提交任务的方法如execute/submit/invokeAll等会抛出异常，类型为RejectedExecutionException。

(2) 自定义处理

拒绝策略是可以自定义的，ThreadPoolExecutor实现了四种处理方式：

ThreadPoolExecutor.AbortPolicy：这就是默认的方式，抛出异常

ThreadPoolExecutor.DiscardPolicy：静默处理，忽略新任务，不抛异常，也不执行

ThreadPoolExecutor.DiscardOldestPolicy：将等待时间最长的任务扔掉，然后自己排队

ThreadPoolExecutor.CallerRunsPolicy：在任务提交者线程中执行任务，而不是交给线程池

中的线程执行

① 都是ThreadPoolExecutor的public静态内部类，都实现了RejectedExecutionHandler接口，这个接口的定义为：

public interface RejectedExecutionHandler {

void rejectedExecution(Runnable r, ThreadPoolExecutor executor);

}

当线程池不能接受任务时，调用其拒绝策略的rejectedExecution方法。

② 拒绝策略可以在构造方法中进行指定，也可以通过如下方法进行指定：

public void setRejectedExecutionHandler(RejectedExecutionHandler handler)

(3) 拒绝策略的意义

我们需要强调下，拒绝策略只有在队列有界，且maximumPoolSize有限的情况下才会触发。如果队列无界，服务不了的任务总是会排队，但这不见得是期望的，因为请求处理队列可能会消耗非常大的内存，甚至引发内存不够的异常。如果队列有界但maximumPoolSize无限，可能会创建过多的线程，占满CPU和内存，使得任何任务都难以完成。所以，在任务量非常大的场景中，让拒绝策略有机会执行是保证系统稳定运行很重要的方面。

2.4 线程工厂

(1) 线程池还可以接受一个参数，ThreadFactory，它是一个接口，定义为：

public interface ThreadFactory {

Thread newThread(Runnable r);

}

这个接口根据Runnable创建一个Thread，ThreadPoolExecutor的默认实现是Executors类中的静态内部类DefaultThreadFactory，主要就是创建一个线程，给线程设置一个名称，设置daemon属性为false，设置线程优先级为标准默认优先级，线程名称的格式为： pool-<线程池编号>-thread-<线程编号>。

① 一般是默认调用默认实现的线程工厂类

2.5 关于核心线程的特殊配置

(1) 线程个数小于等于corePoolSize时，我们称这些线程为核心线程，默认情况下：

① 核心线程不会预先创建，只有当有任务时才会创建

② 核心线程不会因为空闲而被终止，keepAliveTime参数不适用于它

(2) 基本方法

//预先创建所有的核心线程

public int prestartAllCoreThreads()

//创建一个核心线程，如果所有核心线程都已创建，返回false

public boolean prestartCoreThread()

//如果参数为true，则keepAliveTime参数也适用于核心线程

public void allowCoreThreadTimeOut(boolean value)

2.6 工厂类Executors

(1) 类Executors提供了一些静态工厂方法，可以方便的创建一些预配置的线程池，主要方法有：

//创建单个线程执行器

public static ExecutorService newSingleThreadExecutor()

//创建线程池执行器

public static ExecutorService newFixedThreadPool(int nThreads)

public static ExecutorService newCachedThreadPool()

(2) newSingleThreadExecutor基本相当于调用：

public static ExecutorService newSingleThreadExecutor() {

return new ThreadPoolExecutor(1, 1,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

① 原理：只使用一个线程，使用无界队列LinkedBlockingQueue，线程创建后不会超时终止，该线程顺序执行所有任务。该线程池适用于需要确保所有任务被顺序执行的场合。

(3) newFixedThreadPool的调用代码：

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

① 原理：使用固定数目的n个线程，使用无界队列LinkedBlockingQueue，线程创建后不会超时终止。和newSingleThreadExecutor一样，由于是无界队列，如果排队任务过多，可能会消耗非常大的内存。

(4) newCachedThreadPool的代码为：

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,

60L, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>());

}

① 作用：当新任务到来时，如果正好有空闲线程在等待任务，则其中一个空闲线程接受该任务，否则就总是创建一个新线程，创建的总线程个数不受限制，对任一空闲线程，如果60秒内没有新任务，就终止。

(5) newFixedThreadPool和newCachedThreadPool区别：

① 在系统负载很高的情况下，newFixedThreadPool可以通过队列对新任务排队，保证有足够的资源处理实际的任务，而newCachedThreadPool会为每个任务创建一个线程，导致创建过多的线程竞争CPU和内存资源，使得任何实际任务都难以完成，这时，newFixedThreadPool更为适用。

② 不过，如果系统负载不太高，单个任务的执行时间也比较短，newCachedThreadPool的效率可能更高，因为任务可以不经排队，直接交给某一个空闲线程。

③ 在系统负载可能极高的情况下，两者都不是好的选择，newFixedThreadPool的问题是队列过长，而newCachedThreadPool的问题是线程过多，这时，应根据具体情况自定义ThreadPoolExecutor，传递合适的参数。

2.6 线程池的死锁

(1) 背景：

关于提交给线程池的任务，我们需要特别注意一种情况，就是任务之间有依赖，这种情况可能会出现死锁。比如任务A，在它的执行过程中，它给同样的任务执行服务提交了一个任务B，但需要等待任务B结束。如果任务A是提交给了一个单线程线程池，就会出现死锁，A在等待B的结果，而B在队列中等待被调度。

(2) 解决死锁的方法：

① 替换newFixedThreadPool为newCachedThreadPool，让创建线程不再受限，这个问题就没有了。

② 另一个解决方法，是使用SynchronousQueue，它可以避免死锁，怎么做到的呢？对于普通队列，入队只是把任务放到了队列中，而对于SynchronousQueue来说，入队成功就意味着已有线程接受处理，如果入队失败，可以创建更多线程直到maximumPoolSize，如果达到了maximumPoolSize，会触发拒绝机制，不管怎么样，都不会死锁。

三、CompletionService的用法

1. 应用场景：

在异步任务程序中，一种常见的场景是，主线程提交多个异步任务，然后希望有任务完成就处理结果，并且按任务完成顺序逐个处理，对于这种场景，Java并发包提供了一个方便的方法，使用CompletionService，这是一个接口，它的实现类是ExecutorCompletionService。

2. 基本用法

2.1 接口和类定义 -- （CompletionService 类似 ExecutorService）

public interface CompletionService<V> {

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

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

Future<V> take() throws InterruptedException;

Future<V> poll();

Future<V> poll(long timeout, TimeUnit unit) throws InterruptedException;

}

其submit方法与ExecutorService是一样的，多了take和poll方法，它们都是获取下一个完成任务的结果，take()会阻塞等待，poll()会立即返回，如果没有已完成的任务，返回null，带时间参数的poll方法会最多等待限定的时间。

(1) 构造方法

CompletionService的主要实现类是ExecutorCompletionService，它依赖于一个Executor完成实际的任务提交，而自己主要负责结果的排队和处理，它的构造方法有两个：

public ExecutorCompletionService(Executor executor)

public ExecutorCompletionService(Executor executor, BlockingQueue<Future<V>>

completionQueue)

① 至少需要一个Executor参数，可以提供一个BlockingQueue参数，用作完成任务的队列，没有提供的话，ExecutorCompletionService内部会创建一个LinkedBlockingQueue。

2.2 实例代码

public static void parse(List<String> urls) throws InterruptedException {

//新建一个线程执行服务

ExecutorService executor = Executors.newFixedThreadPool(10);

try {

//新建CompletionService

CompletionService<String> completionService = new

ExecutorCompletionService<>(executor);

for (String url : urls) {

//提交任务

completionService.submit(new UrlTitleParser(url));

}

for (int i = 0; i < urls.size(); i++) {

//取出第一个执行结果

Future<String> result = completionService.take();

try {

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

} catch (ExecutionException e) {

e.printStackTrace();

}

}

} finally {

//关闭

executor.shutdown();

}

}

注意：CompletionService可以理解为封装了ExecutorService。

3. 基本原理

(1) ExecutorCompletionService有一个额外的队列，每个任务完成之后，都会将代表结果的Future入队。

4. 实现invokeAny

(1) AbstractExecutorService的invokeAny的实现，就利用了ExecutorCompletionService，它的基本思路是，提交任务后，通过take方法获取结果，获取到第一个有效结果后，取消所有其他任务，不过，它的具体实现有一些优化，比较复杂。