链滴

java 多线程中的中断

作者：wuhongxu
原文链接：https://ld246.com/article/1537516590004
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

概要

在使用java进行多线程开发时,任务和线程的启动很容易.大多数时候我们都会让它们运行直到结束,或让它们自行停止.然而,有时候我们希望能提前结束任务或线程,或许时因为用户取消了操作,或者应用程需要被快速关闭.

要使任务和线程能够安全/快速/可靠的停止下来,并不是意见容易的事.java没有提供任何机制来安全的止线程

java提供了Thread.stop和Thread.suspend等方法提供了终止机制,但是存在依稀额严重的缺陷,因此该避免使用.另:这些方法已经被废弃.原因参见javadoc文档.

Thread.stop天生不安全.停止一个线程会导致它解锁它所锁定的所有monitor(当一个ThreadDeath Exception沿着栈向上传播时会解锁monitor),如果这些被释放的锁所保护的objects由任何一个进入一个一致的状态,其他将要访问该object的线程也会以一种不一致的状态来访问这些object.这种object成为被损坏了".当线程对被损坏的object上操作时,可能会产生意想不到的结果,这些行为可能是很严重的,且难以探测到.不像其他uncheck exception,ThreadDeath Exception静默地杀死进程,因此,用户不会警告他的程序会崩溃,这会在"损坏"之后的任何时候发生,甚至几小时或者几天之后.当然用户可以选择cach这个异常,但那是这会极大地简爱嗯多线程代码编写复杂化,以下两个原因,让这个公司做变得几乎不能:

1:一个线程会在几乎任何地方抛出ThreadDeathException.考虑到这一点,所有同步方法和代码块都必进行仔细的考察

2:线程可能在处理第一个异常的时候,抛出第二个异常,处理语句必须不得不重新开始反复如此直到成功使得编码非常复杂

但是java提供了__中断__(interruption),这是一种协作机制,能够使一个线程终止另一个线程的当前工作.

这种协作式的方法时必要的,我们很少希望某个任务/线程或服务立即停止,伊娃内这种立即停止会使共的数据结构处于不一致的状态.相反,在编写任务和服务时,可以使用一种协作的方式:当需要停止时,它们清除当前正在执行的工作,然后再结束.这提供了更好的灵活性,因为任务本身的代码比发出取消请求的码更清除如何执行清除工作.

以上书面说明摘选自<< java并发编程实战 >>

做一个简单的总结:

1. java不提供(或者说摒弃)线程强行中断的方式,而是采取一种协作的方式处理线程取消的行为.

2. 直接让线程停止是被java摒弃的,虽然由这个方法(已经标记了废弃,但是源码仍然有提供),千万不要用原因:**不安全**.

3. 我们在编写多线程代码时,如果由任务取消的场景千万不要直接在外部调用类似 stop的方法来让程停止运行,而是应该由内到外,内部封装提供一种__协作__机制,向外暴露取消的方法,内部来进行取消理.

总之,我们在编写多线程代码时,对于取消此次任务这类机制,一定要牢记这三点

场景

简要说明就是,这个操作如果能够被提前终止,那么这个操作就可以叫做__可取消的__(cancellable).

- **用户请求取消**.比如用户点击图形用户界面程序的"取消"按钮.

- **有时间限制的操作**.比如,有限时间内请求api接口,当请求超时则放弃请求.
- **应用程序事件**.比如通过多线程寻找最快的排序方式,通过多线程同时运行多个排序算法,当找到其中个最快的方式成功得出正确答案之后,其他也就不用再继续下去了.
- **错误**.比如当爬虫程序搜索相关页面时,当一个爬虫任务发生了一个足以影响整个程序的错误(比如磁满了),所有的搜索任务都会取消(需要注意的是,我们可以通过协作机制,在取消前,尝试解决问题或者保状态).
- **关闭**.比如当应用程序关闭,程序需要保存当前状态以便下次运行时能够继续下去.

仍然需要强调的是,java中没有一种__安全的__抢占式方法来停止线程,只有__协作式的__机制来使任务代码遵循协商好的协议来停止任务.

协作机制

协作机制大致有两种方式:

- 线程中设置取消标志,任务定期查看标志,如果被标记成取消,则任务提前结束,否则任务一直运行下去.
- 使用java提供的中断(interrupt).

取消标志

这是一种很简单也很容易被大家所能理解的方式,我甚至不太想单独拎出来说,但是为了尽可能详细的解问题(也可能是为了保持格式的强迫症?)还是说一说.

这种方式的中心思想其实很简单,说白了,就是在线程执行任务时定时判断以下标志,如果判断出来需要消了,那么就取消任务.参考以下代码(节选自<< java并发编程实战 >>):

```
public class PrimeGenerator implements Runable {
  private final List<BigInteger> primes = new ArrayList<>();
  //注意这里使用volatile标志,使得变量能够在多线程环境下安全的实时共享
  private volatile boolean cancelled;

  @Override
    public void run(){
      BigInteger p = BigInteger.ONE;
      while(!cancelled){
        p = p.nextProbablePrime();
        synchronized (this) {
          primes.add(p);
        }
      }
    }

  public void cancel(){
    cancelled = true;
  }

  public synchronized List<BigInteger> get(){
    //使用复制的方式安全的共享数据
    return new ArrayList<BigInteger>(primes);
  }
```

}

这是一个枚举素数的任务,外部代码能够调用cancel方法来使任务安全的结束.当然,**在外部调用cacel方时,可能存在延迟**,也很好理解.另外如果cancel没有被调用,那么搜索素数的线程将永远运行吸取,不断消cpu的时钟周期,并使得jvm不能正常的退出.

中断(注意!重要!)

上面的代码使用这种取消标志的机制能够使得任务退出.但是如果nextProbablePrime()这一行中,这个法是一个阻塞方法呢?比如BlockingQueue.put().当BlockingQueue队列已满并且一直没有被消耗,那这个方法调用将会一直被阻塞.这非常严重,因为如果是这样,这个任务将一直不能被取消,因为虽然canceled标记已经被标记为取消,但是代码中无法检测这个标记,因为方法一直被阻塞着,不能正常的向下执行去,也因此永远无法结束.

此时怎么办呢,java提供的__线程中断__机制闪亮登场了.线程可以通过这种机制来__通知__另一个线程,诉它在合适的或者可能的情况下停止当前工作,并转而执行其他的工作.

值得注意的是,__线程中断__仍然时一种协作机制,java的api或者语言规范中,并没有将中断与任何取消义关联起来,你可以把这种机制用在__取消__以外任何地方,当然并不推荐这样做. 在<< java并发编程战 >>中给出了这样的评价:**不合适,难以支撑起更大的应用**.

个人理解看来,其实中断机制就是取消标志java内部的官方实现.试想每个人都有不同的中断标志,采用的?当然还是java官方的啦,因为这是官方实现,所以几乎所有jdk里的源码以及第三方库,都会基于此来行线程中断的实现.所以我们就能够统一的进行使用

每个线程都有一个boolean类型的中断状态.当线程中断时,这个线程的中断状态将被设置为true.在Thred中包含了中断线程以及查询线程中断状态的方法.interrupt方法能够中断目标线程,而isInterrupted法能够返回目标线程的中断状态.静态的interrupted方法能够清除当前线程的中断状态,并返回它的值.也是清除中断状态的唯一方法.

java中基于所有阻塞库方法,例如Thread.sleap和Object.wait都会检测线程何时中断,并且在发现中断提前返回.它们会在相应中断时执行的操作由:**清除中断状态,抛出InterruptedException(表示阻塞方因为中断而提前结束)**.

<< java并发编程实战 >>中提到:"当线程在非阻塞的状态下中断时,它的中断状态将被设置,然后根据被取消的操作来检测中断状态以判断发生了中断,通过这样的方法,中断操作将变得有黏性,如果不触发InerruptedException,那么中断状态将一直保持,直到明确地清除中断状态".

我觉得过于书面化了,其实我们根据实际情况来简单模拟以下,非常容易理解.很简单,我们就想象成一个断标志(sdk中中断方法实际执行的方法是native的,我没有看到源码,所以只能想象了),然后对照上面素生成器的代码,如果不去cancelled变量被设置了,但是我们没有取检测,那么这个中断其实没有任何意义因为线程执行根本就没取看它,你只是一个标志,我不遵守你还能打我?

至于黏性更容易理解了,这个cancelled状态一直在那里,没人动过,它始终标记的时返回状态,没人动它,又不能坐上来自己动不是? 所以因此我们也能理解,不是说一调用中断方法线程就能马上返回的,还是要线程自己怎么处理,毕竟__协作机制__,你在线程里面要流氓,不遵守规范,也没人能管得着.

通常,中断是实现取消的最合理方式 --<< java并发编程实战 >>

中断响应处理策略

对于线程的可取消性,优先推荐使用中断处理,毕竟这是java的官方实现,更安全更可靠更易用

在多线程变成下,经常会有的编码场景是使用一个死循环,不停的进行某个任务,如前面的素数生成器,我使用中断策略来进行优化:

```
public class PrimeGenerator implements Runable {
  private final List<BigInteger> primes = new ArrayList<>();

  @Override
    public void run(){
      BigInteger p = BigInteger.ONE;
      while(!Thread.currentThread().isInterrupted()){
        p = p.nextProbablePrime();
        synchronized (this) {
          primes.add(p);
        }
      }
    }

  public void cancel(){
    interrupt();
  }

  public synchronized List<BigInteger> get(){
    //使用复制的方式安全的共享数据
    return new ArrayList<BigInteger>(primes);
  }
}
```

在循环时使用官方的中断标志进行中断检测.同时这种编码方式我们可以更加灵活的使用来增强对于中的响应及时性.例如在多个操作中穿插线程中断检测:

```
public void run(){
  while(Thread.currentThread().interrupted()){
    ...do long time Something
      if(Thread.currentThread().interrupted()){
        break;
      }
    ...do long time Something
      if(Thread.currentThread().interrupted()){
        break;
      }
    ...doSomething
  }
}
```

响应中断(处理InterrupedException)的方式:

- 传递异常(直接throws),从而使你的方法也成为可中断的阻塞方法
- catch异常,换用其他的方式来保存中断状态(不推荐).
- 恢复中断状态(调用静态的interruped方法),从而使上层代码能够对其进行处理

只有实现了中断策略的代码才可以屏蔽中断请求,否则常规的任务和库代码都不应该屏蔽中断请求

中断处理,我梳理出来几种需要铭记的准则:

1. **最合理的中断策略是某种形式的线程及(Thread-Level)取消操作或者服务级(Service-Level)取消作**:尽快退出,在必要时进行清理,通知某个线程所有者该线程已经退出.此外还可以建立其他中断策略,例__暂停服务__或__重新开始服务__.

2. 对于非线程所有者的代码来说(例如:对于线程池而言,任何在线程池实现以外的代码)中都应该__小心保存中断状态__.这样拥有线程的代码才能对中断作出响应,即使非所有者代码也可以作出相应.这也是什么大多数可阻塞的库函数都只是抛出InterruptedException来作为中断响应.它们永远不会在某个由己拥有的线程中执行,因此它们为任务或库代码实现了最合理的取消策略:**尽快退出线程**,并把中断信息递给调用者,从而使上层代码可以采取进一步的操作.

3. 当检测到中断操作时,任务不需要放弃所有的操作,可以推迟处理中断请求,直到某个更合适的时刻.因需要记住中断请求,在完成任务后抛出InterruptedException或者其他一个能够表示已收到中断的返回的标志,来确保发生中断时上层调用者能够知道线程被中断了.从而继续进行相应的处理.

4. 如果非得对 InterruptedException进行捕获(不推荐),那么一定要将中断状态进行恢复调用interrupted.以便后续不会再检测到中断状态,保证线程的正常执行.

总结:线程的中断处理一定要小心,时刻铭记InterruptedException与中断状态密切相关.遇到时一定要中断状态小心保存传递给上一层或者及时停止.从而使你的代码更加的健壮

超时

基于以上的关于中断的一系列说明之后,我们迎来一个新的问题,如何处理超时的情况,试想这样一种情况一个线程的执行如果没有在指定时间内得到结果,我们不再等待.比如向服务端请求数据,如果长时间得到响应,我们就视为服务器不可用,直接判定为失败.

那么如何进行这样的超时判定呢,以下做一个简单的实现:

```
public class TimeoutExecutor{
  private static final ScheduledExecutorService cancelExec = Executors.newFixedThreadPool(1);
  private static void timedRun(Runable r,long timeout,TimeUnit unit){
    final Thread taskThread = Thread.currentThread();
    //使用一个外部线程来执行指定时间后中断任务
    cancelExec.schedule(new Runnable(){
        public void run(){
          taskThread.interrupt();
        }
    },timeout,unit);
    //实际上直接执行,没有采用多线程
    r.run();
  }
  public static void main(String[] args){
    Executor.timeRun(new Runnable(){
        public void run(){
          ...doSomething
        }
        //一秒超时
    },1,TimeUnit.Second);
  }
}
```

上面的代码非常简单,并且基本能够达成超时判定的目的,但是却破坏了以下规则:在中断线程之前,应该解它的中断策略.由于timeRun可以从任意一个线程中调用,因此它无法知道这个调用线程的中断策略.果任务在超时之前完成,那么中断timeRun所在线程的取消任务将在timeRun返回到调用者之后启动,如这个示例中,我们的main线程之后可能会运行其他代码.我们不知道在这种情况下将执行什么代码,但果一定是不好的,因为线程已经被标记为阻塞了.而且如果我们在执行代码中不响应中断(catch execptio),那么任务还是会直到结束才会被返回,此时也不直到时限是否已经超过.如果main方法没有在指定的间内返回,那么给调用者带来的使更多的麻烦.

基于以上问题,我们再尝试做出改进:

```
public class TimeoutExecutor{
  private static final ScheduledExecutorService cancelExec = Executors.newFixedThreadPool(1);
  public static void timeRun(final Runnable r,long timeout,TimeUnit unit)throws InterruptedEx
eption{
    class RethrowableTask implements Runable{
      //使用volatile标记的异常,并且将这个异常留存以让它能够被共享
      private volatile Throwable t;
      public void run(){
        try {
          r.run()
        }catch(Throwable t) {
          this.t = t;
        }
      }
      void rethrow(){
        if(t != null) {
          throw t;
        }
      }
    }

    RethrowableTask task = new RethrowableTask();
    //开一个线程去执行目标方法
    final Thread taskThread = new Thread(task);
    taskThread.start();
    //开一个线程去进行超时打断操作
    cancelExec.schedule(new Runable() {
        public void run(){
          taskThread.interrupt();
        }
    },timeout,unit);
    //使用join的方式让调用这线程等待超时时间内执行完成,如果超时则不再等待
    taskThread.join(unit.toMillis(timeout));
    //尝试将一场抛出去,没有就不处理
    task.rethrow();
  }
}
```

这份代码就解决了前面提出的问题,但是由于它依赖与一个限时的join,因此存在join的不足:无法知道执控制是因为线程正常退出而返回还是因为join超时而返回的,当然,这是join方法本身的不足.

使用Future实现取消

java内置了Future,它比自行编写更好,因此我们可以使用Future和人物之性框架来构建timedRun.

ExecutorService.submit方法返回一个Future来描述任务,Future拥有一个cancel方法,该方法带有一个oolean参数的参数,表示取消操作是否成功(这只是表示任务是否能够接收中断,而不是表示任务是否能测并处理中断).

通过查看源码中的注释,这个参数名为:mayInterruptIfRunning.看到变量命名就很容易理解这个变量作用,是指是否在执行中发送中断消息.一般而言,除非你非常清除线程的中断策略,否则不要传入true,因通过前文我们了解到,这是一种协作机制,如果你不了解实现者是否时遵守标准的中断策略,那么最好不轻易使用,因为他可能不是按照标准策略来编写的,依照java的严谨性来说,你最不应该做的就是__猜测个功能的实现__.

另外还有一般而言,当不了解内部实现的情况下取消任务时一定不要尝试中断线程池,根据前文了解到,发送中断请求时,一定会有延迟(只是多与少的问题,因为中断信号时通过检测来执行的).然而如果不了具体实现,那么可能当前任务并没有响应中断,有可能你想要结束的任务所在的线程已经在执行其他任了,此时会影响到其他任务的运行.

通过Future,就能够轻松实现超时策略了.看看代码:

```
public static void timedRun(Runnable r,long timeout,TimeUnit unit){
    Future<?> task = taskExec.submit(r);
    try{
      task.get(timeout,unit);
    }catch(TimeoutException e){
      //取消任务
    }catch(ExecutionException e){
      //ExecutionException是内部执行代码的异常经过封装统一抛出来的,我们可以通过getCause方
获取到原异常来重新抛出
      throw e.getCause();
    } finally{
      //无论如何都取消,正常结束取消没有任何影响
      //当有异常产生时也不需要结果,直接取消并由前面的代码抛出异常
      task.cancel(true);
    }
}
```

总结

- 多线程的中断是一种协作机制
- 中断策略应当以保存/传递中断信号为主,其次再考虑恢复中断信号.
- java变成不宜作出猜测,特别是多线程情况下,代码会变得特别复杂,谨慎做好处理才能在多线程编码如鱼得水

最后的最后不得不说:

多线程的复杂性是真的高,一个简简单单的取消操作,都涉及到了很多的东西,需要考虑的情况也是非常的.本文关于取消操作并没有写完,只能说够用,还有很多情况并未说明,例如不可中断的阻塞方法/非标实现的取消策略的处理方式/线程池随服务停止/非正常线程终止甚至还有jvm关闭的处理等等.看情况向写不写下一篇吧.

本人也是因为在编写一些多线程相关的代码,遇到了很多的问题,以前自以为了解了解线程池/同步等等,能做好多线程了,想想还是图样图森破啊.不过多线程的这些复杂性反而激发了我的兴趣,不用玩curd是爽啊~2333,接下来应该还会继续对多线程相关的东西继续学习,给自己加个油!

待续...

参考

本文内容主要参考自 << java并发编程实战 >>.