线程中断、超时与降级——《亿级流量》内容补充

原创 2017-06-02 张开涛 开涛的博客

最近一位朋友在公众号留言问一个关于熔断的问题：

> 使用hystrix进行httpclient超时熔断错误，我是顺序操作的（没有并发），发现hystrix会超时断开，但是会导致hystrix线程池不断增多，直到后面因线程池装不下拒绝？

而该问题跟线程中断、超时与降级等有关，因此本文将详细介绍导致这个问题背后的原因。

需要提前了解的知识：
你的Java代码可中断吗（1）
你的Java代码可中断吗（2）

当我们在线程中执行如用户请求任务时，比如HTTP处理线程，最担心的什么？
1、线程数无限增长；
2、线程执行时间长；
3、线程不可中断。

对于线程数无限增长，我们可以通过使用线程池来控制线程数量，控制线程不是无限增长的。

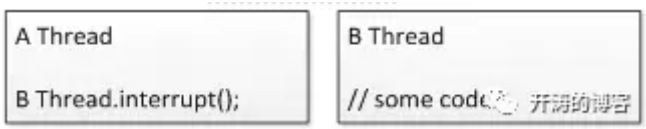
对于线程执行时间长，我们应设置合理的超时时间来保障线程执行时间可控，当超时时要么返回给用户错误页面，要么可以返回降级页面。

对于线程不可中断，我们应想办法将线程设计的可中断，从而在遇到问题可中断线程并降级处理。

线程池可以参考我新书《亿级流量》中的“第12章连接池线程池详解”。超时时间可以参考我新书《亿级流量》中的“第6章 超时与重试机制”。

接下来的部分将主要讲解线程中断。

线程中断是通过Thread.interrupt()方法来做，一般是在A线程来中断B线程。



首先我们来看下该方法的一些Javadoc描述：

1. 如果线程被Object的wait()、wait(long)、wait(long, int) 或者Thread的join()、join(long)、join(long, int)、sleep(long)、sleep(long, int)方法阻塞，执行线程中断，且抛出InterruptedException，但中断状态并清空重置，即Thread. isInterrupted()返回false；
2. 如果线程被java.nio.channels.InterruptibleChannel上的一个I/O操作阻塞，执行线程中断，且该InterruptibleChannel将被关闭，抛出java.nio.channels.ClosedByInterruptException，线程

中断状态会设置，即Thread. isInterrupted()返回true；

3. 如果线程被java.nio.channels.Selector阻塞，执行线程中断，该Selector#select()方法将立即返回，相当于调用了java.nio.channels.Selector#wakeup()，不会抛出异常，但会设置中断状态，即Thread. isInterrupted()返回true；
4. 如果不满足以上条件的，那么执行线程中断不会抛出异常，仅设置中断状态，即Thread. isInterrupted()返回true。也就是说我们代码要根据该状态来决定下一步怎么做。

从如上描述可以看出，如果方法异常描述上有抛出InterruptedException、ClosedByInterruptException异常的，说明该方法可以中断，如“public final native void wait(longtimeout) throws InterruptedException”，但是中断状态会被会被重置要看其Javadoc描述。其他情况基本都是设置中断状态而不会中断掉操作。

BIO（Blocking I/O）操作不可中断

如java.net.Socket读写网络I/O时是阻塞的，除了设置超时时间外，还应该考虑让它可中断或者尽早中断。可以参考《你的Java代码可中断吗》。还有如JDBC驱动mysql-connector-java、HttpClient等大部分都是使用BIO，它们也是不可中断的。

NIO（New I/O）操作可中断

NIO涉及到两部分：java.nio.channels.Selector和java.nio.channels.InterruptibleChannel，它们是可中断的。如java.nio.channels#SocketChannel实现了InterruptibleChannel，如下方法都是可中断的，并会抛出ClosedByInterruptException异常：

- connect(SocketAddress remote)
- read(ByteBuffer[] dsts, int offset, int length)
- read(ByteBuffer[] dsts)
- write(ByteBuffer src)

线程、BIO与中断

我们使用BIO实现的HttpClient来做个实验，如下代码所示：

```
public class BlockingIOTest {
  public static void main(String[] args) throws Exception {
    Thread threadA = new Thread(()-> {
      try {
       //该阻塞5s
      String url = "http://localhost:9090/ajax";
      //HttpClient是BIO，不可中断
      HttpResponse response = HttpClientUtils.getHttpClient().execute(new HttpGet(url));
      System.out.println("http status code : " + response.getStatusLine().getStatusCode());
      //虽然在threadB执行了threadA线程中断
      //但是仅仅是设置了中断状态为true
      //并没有中断线程A的执行，该线程还是正常的执行完成了
      System.out.println("threadA is interrupted: " + Thread.currentThread().isInterrupted());
```

```
        } catch (Exception e) {
          e.printStackTrace();
        }
      });
      Thread threadB = new Thread(()-> {
        try {
          Thread.sleep(2000L);
        //休眠2s后，中断线程A
          threadA.interrupt();
        } catch (Exception e) {
        }
      });
      threadA.start();
      threadB.start();
      Thread.sleep(15000L);
    }
}
```

如上代码的输出结果为：
http status code : 200
threadA is interrupted: true

如上代码的执行流程是这样的：

1. 线程A通过BIO实现HttpClient远程调用http://localhost:9090/ajax获取数据，而该服务需要5s才能响应；
2. 线程B在线程A执行2s后进行了中断处理，但是线程A调用的HttpClient是阻塞且不可中断的操作，仅仅是设置了线程A的中断状态为true，因此其一直等待网络I/O完成；
3. 当线程A从远程获取到结果后继续执行，Thread.currentThread().isInterrupted()将输出true，表示线程A被设置了中断状态。

从而需要注意设置了中断状态与中断执行不是一回事。因此对于使用BIO，一定要设置好连接和读写的超时时间，另外可以参考《你的Java代码可中断吗》进行可中断设计。

线程池、Future与中断

我们往线程池提交一个HttpClient任务，并通过Future来等待执行结果，如下代码所示：

```
public class ThreadPoolTest {
  private static ExecutorService executorService = Executors.newFixedThreadPool(5);
```

```
    public static void main(String[] args) throws Exception {
     Future<Integer> futureA = executorService.submit((Callable) () -> {
      //该url会阻塞5s
     String url = "http://localhost:9090/ajax";
     //HttpClient是BIO，不可中断
     HttpResponse response = HttpClientUtils.getHttpClient().execute(new HttpGet(url));
     Integer result = response.getStatusLine().getStatusCode();
     System.out.println("thread a result : " + result);
     return response.getStatusLine().getStatusCode();
     });
     Future<Integer> futureB = executorService.submit((Callable) () -> {
       //该url会阻塞5s
      String url = "http://localhost:9090/ajax";
      //HttpClient是BIO，不可中断
      HttpResponse response = HttpClientUtils.getHttpClient().execute(new HttpGet(url));
      Integer result = response.getStatusLine().getStatusCode();
      System.out.println("thread b result : " + result);
      return result;
      });
      try {
     Integer resultA = futureA.get(100, TimeUnit.MILLISECONDS);
     } catch (TimeoutException e) {
      System.out.println("future a timeout");
    }
    try {
      Integer resultB = futureB.get(100, TimeUnit.MILLISECONDS);
    } catch (TimeoutException e) {
      System.out.println("future b timeout");
     }
    executorService.awaitTermination(10000L, TimeUnit.MILLISECONDS);
    }
}
```

如上代码的输出结果为：

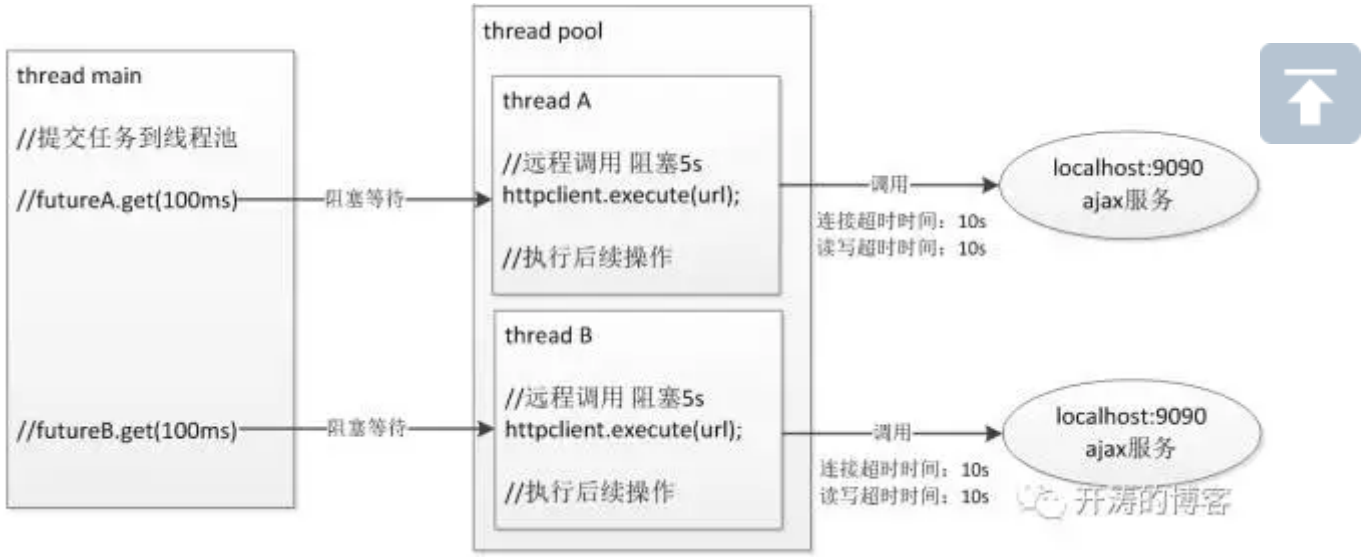
future a timeout

future b timeout

thread a result : 200

thread b result : 200

如上代码的执行流程是这样的：



1. 主线程往线程池提交了两个HttpClient阻塞调用任务，该任务响应时间为5s；
2. 主线程阻塞在两个带超时的Future等待上，Future在等待线程池任务执行结束，Future的超时时间设置为100ms，所以很快就超时并返回了，主线程继续执行，在《亿级流量》中我们用到了很多这种方法进行并发获取数据和降级或熔断处理；
3. 线程池中的两个任务其实并没有被中断，还是占用着线程池中的线程，在后台继续执行，直到完成。

从如上可以看出，使用Future时只是在主线程解除了阻塞，并没有连带把线程池任务取消掉，还是占用着线程并阻塞执行。

之前有位同学在公众号后台留言咨询：

> 使用hystrix进行httpclient超时熔断错误，我是顺序操作的（没有并发），发现hystrix会超时断开，但是会导致hystrix线程池不断增多，直到后面因线程池装不下拒绝？

看完如上示例，应该能解决该读者的疑惑。虽然熔断了，但是线程中的操作并没有真正的中断，而是还占着线程资源。

接下来我们可以简单看下Future其中的一个实现FutureTask：

超时等待方法get(long timeout, TimeUnit unit)伪代码：

```
while(true) {
 if (Thread.interrupted()) {//如果当前线程中断了，处理现场，并抛出中断异常
  //some code
  throw new InterruptedException();
 }
 //判断剩余休眠时间
 nanos = deadline - System.nanoTime();
 if (nanos <= 0L) {//如果没有休眠时间了，则处理线程，并终止执行
  //some code
  return state;
 }
 //休眠一段时间，内部实现为UNSAFE.park(false, nanos)
```

```
  LockSupport.parkNanos(this, nanos);
}
```

取消方法cancel(boolean mayInterruptIfRunning)伪代码：

```
if (mayInterruptIfRunning) {//中断当前线程
  Thread t = runner;
  if (t != null)
    t.interrupt();
}
//执行UNSAFE.unpark(thread)唤醒休眠的当前线程
LockSupport.unpark(t);
```

即当我们调用Future#cancel时，是通过唤醒Future所在线程实现，当然实际是比这个要复杂的。

回填结果方法set(V v)伪代码：

```
//修改Future状态为完成
//保持v的值，从而Future#get能获取到
//通过LockSupport.unpark(t)唤醒休眠的线程
```

当线程池中的线程执行完成后，是通过Future#set把值设置回Future，从而唤醒休眠的线程，即阻塞在Future#get的等待，然后获取到该结果。

锁与中断

synchronized和ReentrantLock#lock()在获取锁的过程中是不可中断的，假设出现了死锁将一直僵持在那，无法终止阻塞。但我们可以使用可中断的ReentrantLock#lockInterruptibly()方法或者ReentrantLock#tryLock(long timeout, TimeUnit unit)实现可中断。

总结

在设计高可用系统时，尽量使用线程池，而不是通过每个请求创建一个线程来实现，通过线程池的拒绝策略来优雅的拒绝无法处理的请求。

检查整个请求链路设置合理的超时时间，跟调用方协商合理的SLA、降级限流方案。更长的超时时间意味着出现问题时请求堆积的越多，越可能产生雪崩。

明确知道自己的服务是否可中断，如果不可中断，应该使用线程池和Future实现伪可中断，通过Future配置合理的超时时间，当超时时执行相应的降级策略。也要清楚的知道通过Future只是伪中断，线程池中的任务还是在后台执行，当Future超时后进行重试时，会对调用的服务产生更多的请求，从而造成一种DDos，一定要注意相应的处理策略。

池大小、超时时间和中断没有最优的配置策略，要根据自己的场景来动态调整，在系统遇到高并发或者异常时，我们要保护什么，放弃什么，要有权衡。

==========

在531京东图书音像超级品类日终结战报中，《亿级流量》进入科技TOP5，感谢大家的支持，书中的不足，我会持续更新补充内容到公众号，真正形成一个体系。