Java 编程中网络超时的简单处理

Disposal of the Network Overtime in Java Programming

刘 冬 (广州 暨南大学 计算机系 510632)

摘要:本文分别就连接后和连接时两种情况,讨论了基于套接字的 Java 语言网络编程中网络超时的简单处理,提出了让 Java 应用程序具有非阻塞模式 I/O 操作功能的方法,并且给出了一个简单的应用实例。

关键词:Java 套接字 超时 阻塞 非阻塞

1 前言

近些年来,基于互联网的应用与服务发展十分迅速,各个企业纷纷推出各式各样的网络服务。在理想的情况下,这些网络服务应该始终如一、不间断的为网络客户提供优质服务,不会因为数据传输的中断而使得客户服务的停滞甚至中止。但在现实条件下,由于网络传输等各种原因,网络服务远不能达到这种理想情况。因此,网络应用程序就必须较好的处理网络超时问题,而不能让用户在停滞的应用程序前感到不知所措。Java 语言是伴随着互联网大潮而诞生的语言,Java 语言天生具有了适于网络编程的许多优点,因此它也是网络编程中使用的较多的语言。本文就主要讨论了基于套接字的 Java 语言网络编程中网络超时的简单处理,使得用户的 Java 应用程序具有非阻塞模式 I/O 操作的功能。

2 问题的提出

基于文件的 I/O 操作在与 I/O 设备进行输入输出操作时,其操作可分为阻塞模式操作和非阻塞模式操作。阻塞模式操作是指在 I/O 设备准备好之前,读操作或写操作都将被阻塞即等待。而非阻塞模式操作是指在发出读写操作之后,如果 I/O 设备尚未准备好,则读写操作将被中断而不是等待。而基于套接字的网络 I/O 操作在进行网络 I/O 操作时,如同与 I/O 设备进行输入输出操作一样,也可以分为两类操作:阻塞模式操作和非阻塞模式操作。

在缺省的情况下,用 Java 语言编写的网络程序所采用的操作方式是阻塞模式操作。也就是当 Java 网络应用程序从一个套接字连接中读取数据时,如果没有立即的响应它将会等待。如果没有可以得到的数据,程序则会一直等待,而不会去做其他的工作。解决这一问题的常用办法之一就是在程序中用一个线程去执行此操作,这样,当此线程被阻塞时而应用程序依然能够对用户的指令做出反应,甚至可以中断被阻塞的线程。虽然这种方法很常用,但是它在一定程度上会增加应用程序的复杂性。

其实,有一个更为简单的解决办法。因为 Java 语言也支持非阻塞模式的网络 I/O 操作。Socket 类、ServerSocket 类和 DatagramSocket 类都可以被设置为非阻塞模式的网络 I/O 操作。我们可以指定读写操作所延迟等待的最长时间,超过此时间应用程序将重新获得控制权。这种方法对于客户端软件最为简单,而且所需的代码简单而易于处理。

但是 Java 环境下非阻塞模式网络 I/O 操作的唯一缺点就是它需要一个已经存在的套接字。也就是说,上面的方法虽然对于连接后(套接字已被创建)的各种网络 I/O 操作工作正常,但因为没有现成的方法给连接操作(此时套接字尚未被创建)设置最大延迟时间,所以连接操作有可能会延迟很长的时间。出于各方面的考虑,许多应用程序都需要连接操作也能够设置最大延迟时间。下文将分别对连接后和连接时两种情况的超时处理进行讨论。

3 连接后的超时处理

连接后的超时处理比较简单,我们就从连接后的超时处理开始讨论。

虽然有时有用阻塞模式网络 I/O 的必要,但大多数情况下非阻塞模式网络 I/O 是更清楚、更好的方式。因此自从 Java1.1 开始,Java API 允许程序员设置套接字的选项。这些选项让程序员可以更多的控制套接字通信。其中有一个很有用的选项:SO_TIMEOUT,它允许程序员设置读操作所阻塞的时间。我们可以设置一个短时间的延迟,使我们的网络程序实现非阻塞模式的 I/O 操作。

让我们先看一看这是怎么工作的。Java 中的 Socket 类、DatagramSocket 类和 ServerSocket 类都有 setSoTimeout(int)方法,这个方法允许我们以毫秒方式设置最大的延迟时间。在设置之前,以下的网络操作都是阻塞模式的:ServerSocket. accept ( ), SocketInputStream. read(), DatagramSocket.receive()。

而设置了最大的延迟时间后,当这些方法被调用时,程序开始计时,实现非阻塞模式。如果操作没有被阻塞,程序

会重置计时，这些方法下次被调用时程序将重新计时。下面介绍的例子演示了不采用多线程方式处理超时的简单方法。

```
//在 2000 端口创建一个 datagramsocket 来监听传来的 UDP 包
DatagramSocket dgSocket=new DatagramSocket(2000);
//设置 6 秒钟的最大延迟，使缺省的阻塞模式操作失效
dgSocket.setSoTimeout(6000);
```

指定最大延迟时间可防止我们的网络操作陷入无休止的阻塞中。当网络操作超过我们指定的最大延迟时间时，程序就会抛出 java.io.InterruptedIOException 异常。下面的程序片段演示了当从 TCP 套接字中进行读操作时是如何处理的。

```
//设置 10 秒钟的最大延迟
socket.setSoTimeout(10000);
try {
  //创建 BufferedReader 来从套接字中读数据
   BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    //读数据
    for(...;...;...)
    {
      String line=in.readLine();
      If (line ! = null)
        System.out.println(line);
      Else
      Break;
  }
}
//网络超时发生时抛出异常
catch (IOException ioe)
{
  System.err.println("网络 I/O 错误:"+ioe);
}
```

从上可见连接建立后处理网络超时的方法十分简单，仅需要在 try{} catch 块中加入少量代码，应用程序便可对超时做出反应而不会阻塞自己。例如，程序可以提示用户网络超时了，也可以试着建立新的连接。当使用数据报套接字时，程序可以以重发数据报的形式处理网络超时。事实上，非阻塞模式 I/O 在大多数情况下都是最佳的选择，只有少数情况例外。

4 连接操作时的超时处理

有时应用程序需要达到完全的超时监测和处理，那么仅有上面的连接后的超时处理是远远不够的，还需要考虑连接时操作的超时处理。当新创建 java.net.Socket 的一个实例时，程序便尝试建立一个连接。如果 java.net.Socket 构造器中所指定的目的主机已经监测到，而目的主机上没有服务运行在 java.net.Socket 构造器所指定的端口，则 ConnectionException 异常被抛出，程序重新获得控制权。然而，如果目的主机并未运行，或者没有到达目的主机的路由，则连接操作会盲目的等待，同时程序也会失去响应。而且，没有方法修改超时的值。尽管对套接字构造器的调用最终会返回程序，但这会经过相当长一段时间的延迟。处理这个问题的一种方法就是采用辅助线程：由辅助线程执行可能被阻塞的连接操作，并不断的轮询这个线程，以监测连接是否已经建立。

然而，这并不是一个很好的解决方法。我们可以按照上面的方法把客户端程序改为多线程程序，但会发现这样做所花的代价常常是难以接受的。它将使代码更复杂，而且当所写的仅仅只是一个简单的网络应用程序时，更是如此。如果你编写了许多的网络程序，会发现你经常做许多的重复的工作。

这里有一个更简单的解决方法。可以编写一个简单的可重用的类。这个类生成一个 TCP 套接字连接而不会阻塞很长的时间。我们可以简单的调用这个类的 getSocket 方法（要指定主机名、端口号、最大延迟时间）来获得一个套接字。以下的例子演示了一个连接请求：

```
//通过主机名连接远程服务器，并设置最大延迟为 4 秒钟
Socket connection = TimedSocket.getSocket("server.network.net",23,4000);
```

如果一切正常，它将返回一个套接字，和标准的 java.net.Socket 构造器所返回的一样。如果在指定时间内连接没有建立，这个方法会停止并抛出 java.io.InterruptedIOException 异常，就像上文所说的设定了最大延迟的读操作发生超时时的情况一样。

4.1 将多线程的网络连接代码封装入一个类中

下面就详细介绍一下我们要编写的处理连接操作超时的类——TimedSocket 类。

TimedSocket 类不仅本身很有用，而且它也非常有助于理解怎么处理阻塞模式 I/O。当一个单线程程序执行一个阻塞模式 I/O 时，程序会被阻塞或长或短的不确定的时间。如果是多线程程序，则只有一个线程被阻塞，其他线程可以继续执行。先让我们看一下 TimedSocket 类是如何工作的。

如图 1 所示，程序需要连接远程服务器时，就调用 TimedSocket.getSocket()方法并指定远程主机和端口号。getSocket()方法是重载的，既可输入字符串类型的主

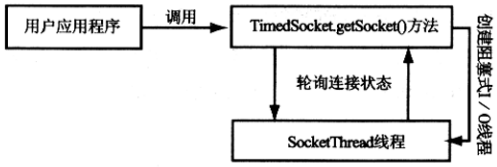

图 1　TimedSocket 类工作示意图

机名,也可输入网络地址类型的主机地址,这已能满足大多数套接字操作的要求,如果有特殊的要求,还可以编写特殊的重载方法。在 getSocket()方法中,创建了一个辅助线程(可以假定此线程叫 SocketThread),由它来创建 java.net.Socket 的实例。辅助线程提供检测连接是否建立和检测错误是否发生的方法(例如:在连接期间抛出 java.net.SocketException 异常)。在连接期间,主线程将等待错误的返回或者是网络的超时。每隔几百毫秒,主线程会检查辅助线程是否已经建立了连接。如果检测失败,就必须进一步检测是否发生错误或者是否网络超时。如果都没有就表明连接还在进行,主线程就会下次再轮询它。

在 getSocket()方法中,充分利用了异常处理。如果有错误发生,可以从 SocketThread 实例获取异常然后再抛出。如果发生网络超时,getSocket()方法则抛出 java.io.InterruptedIOException 异常。

以下的代码段演示了轮询和错误处理机制。

```
for (.....;...;...;)
{
  //检测连接是否已建立
  if (st.isConnected()) //st 为 SocketThread 的实例
  {
    //将 socket 赋给变量,跳出循环
    sock=st.getSocket();
    break;
  }
  else
  {
  //检测是否有错误发生
  if (st.isError())
  {
    //没有建立连接
    throw (st.getException());
  }
  try
  {
    //线程睡眠一段时间
    Thread.sleep (POLL_DELAY);
  }
  catch (InterruptedException ie) {}
  //计时增加
  timer+ =POLL_DELAY;
  //检查是否超过最大延迟
  if (timer > delay)
  {
    //连不上服务器
    throw new InterruptedIOException("在" + delay + "毫秒中未能连接上服务器");
  }
  }
}
```

4.2 辅助线程的分析

辅助线程(SocketThread 线程)尝试创建 java.net.Socket 的一个新实例时,连接有规律的被主线程轮询。辅助线程提供了检查连接状态的方法,也提供了方法来取得已建立的连接。SocketThread.isConnected()方法返回一个布尔值以表示连接是否建立,SocketThread.getSocket()方法返回一个已建立的连接。另外,还提供了相似的方法以确定是否发生错误和访问已捕捉到的异常。

所有的这些方法提供了控制 SocketThread 实例的接口,而不允许外部直接对私有成员变量的访问和修改。接下来的代码段演示了线程的 run()方法。如果套接字构造器成功返回套接字,它将被赋给一个私有成员变量,并提供了访问它的方法。下一次用 SocketThread.isConnected()方法询问连接状态时,套接字就可以被利用了。同样的技术被用来检测错误,如果 java.io.IOException 被捕捉到,它也被存储在私有成员变量中,可以通过 isError()和 getException()方法访问它。

```
public void run()
{
  //用作连接的 socket
  Socket sock=null;
  try {
    //判断指定的是服务器名还是服务器地址
    if (m_host ! = null)
    {
      //连接远程服务器(此时用的是阻塞模式)
      sock=new Socket(m_host,m_port);
    }
    else
    {
      //连接远程服务器(此时用的是阻塞模式)
      sock=new Socket(m_inet,m_port);
```

```
    }
  }
  catch (IOException ioe)
  {
    //给成员变量赋值
    m_exception=ioe;
    return;
  }
  //连接成功,赋予成员变量
  m_connection=sock;
}
```

4.3 在应用程序中应用 TimedSocket 类实例

设计 TimedSocket 类主要是用来使网络超时的处理变得更为简单,并提供与非阻塞 I/O 操作一样的超时处理机制。TimedSocket 类和用 setSoTimeout 方法控制的读操作抛出一样的异常,所以它们处理超时的代码量差不多相同。为了演示 TimedSocket 类在实际中的用法,我编写了一个简单的程序获取互联网网页,其中应用了上面所提到的两种超时处理技术。虽然 URL 类和 URLConnection 类都支持 HTTP 协议,但它们却不允许设置超时的时间。因此,应用套接字实现 HTTP 协议可能是开发人员所要面对的问题,也是很有价值的练习。对于必须经常建立连接的系统来说,支持超时处理尤为重要,例如浏览器等等。

先让我们看一下 HTTP 应用程序。所有的代码都包含在 try{}catch 块中,而且有几种不同的异常可以被捕捉到。其中在我们看来最重要的是 java.io.InterruptedIOException 异常。TimedSocket 类被用来建立套接字连接,读操作也被设置了超时标准。有几种原因可引起异常:连接操作的超时,或者读操作的超时。如果你需要分别处理这两种超时,也可以用两个不同的 try{}catch 块来处理。虽然,用不同 try{}catch 块捕捉不同的异常是一种好方法,但为了简单,我们也可以将所有的 try{}catch 块合并到一块。

```
try
{
  //解析 URL
  dataURL = new URL(args[0]);
  remote_address = dataURL.getHost();
  remote_path = dataURL.getFile();
  remote_port = dataURL.getPort();
  //检查端口,缺省是 80
  if (remote_port == -1) remote_port =80;
  //连接远程服务
  Socket connection = TimedSocket.getSocket
(remote_address,remote_port,4000);
    //设置最大延迟为 10 秒
    connection.setSoTimeout (10000);
    ......}
  catch (InterruptedIOException iioe)
  {
    System.err.println("Remote host timed out");
  }
  catch (MalformedURLException mue)
  {
    System.err.println ("Invalid URL");
  }
  catch (IOException ioe)
  {
    System.err.println ("Network I/O error -" +
ioe);
  }
```

运行这个测试程序的方式非常简单,只需传一个 URL 给测试程序即可。例如:

```
java testTimedSocket http://129.1.1.2
```

然而,我们真正想要测试的是它对网络超时的处理,所以,我们要传一个可以引起超时的 URL 给程序。例如:在局域网中,可以传一个已关机的机器的 URL 给程序。测试的结果与设计的结果一样。

5 总结

由于用户的网络质量有好有坏,所以,在应用程序开发时对网络超时的处理也很重要。一般的处理方法有较大的弊端,而本文所提出的方法,较好的解决对超时的处理。它将处理的细节都封装在一个类中,不仅使开发人员可以很方便的调用,而且可以在各个项目中重用,这将会节省开发人员大量的时间。相信这种方法会对广大 Java 开发人员有很好的借鉴之处。

参考文献

1 Java(tm) 2 Platform SE v1.3 API Specification. Sun Microsystems Inc. http://java.sun.com/j2se。

2 Bruce Eckel. Thinking in Java. Second Edition。

3 Elliotte Rusty Harold 著,刘东华等译,Java 网络编程(第二版),中国电力出版社。

4 Harvey M. Deitel 等著,袁兆山等译,Java 程序设计教程,机械工业出版社。