主要内容

# Socket通道

新的socket通道类可以非阻塞模式并且是可选择的。这两个性能可以激活大程序（如网络服务区和中间件组件）巨大的可伸缩性和灵活性。本节中我们会看到，再也没有为两个socket连接使用一个线程的必要了，也避免了管理大量线程所需的上下文交换总开销。借助新的NIO类，一个或几个线程就可以管理成百上千的活动。socket连接了并且只有很少甚至可能没有性能损失。

全部socket通道类（DatagramChannel、SocketChannel和ServerSocketChannel）在被实例化时都会创建一个对等socket对象。这些事我们所熟悉的来自java.net的类（Socket、ServerSocket和DatagramSocket），它们已经被更新以识别通道。对等socket可以通过调用socket()方法从一个通道上获取。此外，这三个java.net类现在都有getChannel()方法。

虽然每个socket通道（在java.nio.channels包中）都有一个关联的java.net socket对象，却并非所有的socket都有一个关联的通道。如果您用传统方式（直接实例化）创建了一个Socket对象，它就不会有关联的SocketChannel并且它的getChannel()方法将总是返回null。

**非阻塞模式**

Socket通道可以在非阻塞模式下运行。这个说法虽然简单却有着深远的含义。传统Java Socket的阻塞性质曾经是Java程序可伸缩性的重要制约之一。非阻塞I/O是许多复杂的、高性能的程序构建的基础。

设置或重新设置一个通道的阻塞模式是很简单的，只要调用configureBlocking()方法即可，传递参数值为true则设为阻塞模式，参数值为false值设为非阻塞模式。真的，就这么简答！您可以通过调用isBlocking()方法来判断某个socket通道当前处于哪种模式。

服务器端的使用经常会考虑到非阻塞socket通道，因为他们使同时管理很多socket通道变得更容易。但是在客户端使用一个或几个非阻塞式的socket通道也是有益处的，例如，借助非阻塞socket通道，GUI程序可以专注于用户请求并且同时维护与一个或多个服务器的会话。在很多程序上，非阻塞模式都是有用的。

**ServerSocketChannel**

它是一个基于通道的socket监听器。它同我们所熟悉的java.net.ServerSocket执行相同的基本任务，不过它增加了通道语义，因此能够在非阻塞模式下运行。用静态的open()工厂方法创建一个新的ServerSocketChannel对象，将会返回同一个未绑定的java.net.ServerSocket关联的通道。该对等ServerSocket可以通过在返回的ServerSocketChannel上调用socket()方法来获取。作为ServerSocketChannel的对等体被创建的SerSocket对象依赖通道实现。这些socket关联的SocketImpl能识别通道。

通道不能被封装在随意的socket对象外面。由于ServerSocketChannel没有bind()方法，因此有必要取出对等的socket并使用它来绑定到一个端口以开始监听连接。我们也是使用对等ServerSocket的API来根据需要设置其他的socket选项。

和java.net.ServerSocket一样，ServerSocketChannel也有accept()方法。一旦您创建了一个ServerSocketChannel并用对等socket绑定了它，然后您就可以在其中一个上调用accept()。如果您选择在ServerSocket上调用accept()方法，那么它会同任何其他的ServerSocket表现一样的行为：

总是阻塞并返回一个java.net.Socket对象。如果您选择在ServerSocketChannel上调用accept()方法则会返回SocketChannel类型对象，返回的对象能够在非阻塞模式下运行。假设系统已经有一个安全管理器（security manager），两种形式的方法调用都执行相同的安全检查。

如果以非阻塞模式被调用，当没有传入连接在等待时，ServerSocketChannel.accept()会立即返回null。正式这种检查连接而不阻塞的能力实现了可伸缩性并降低了复杂性。可选择性也因此得到了实现。我们可以使用一个选择器实例来注册一个ServerSocketChannel对象以实现新连接达到时自动通知的功能。

**SocketChannel**

Socket和SocketChannel类封装点对点、有序的网络连接，类似于我们所熟知并喜爱的TCP/IP网络连接。SocketChannel扮演客户端发起同一个监听服务器的连接。直到连接成功，它才能收到数据并且只会连接到的地址接收。

每个SocketChannel对象创建时都是同一个对等的java.net.Socket对象串联的。静态的open()方法可以创建一个新的SocketChannel对象，而在新创建的SocketChannel上调用socket()方法能返回它对等的Socket对象；在该Socket上调用getChannel()方法则能返回最初的那个SocketChannel。

新创建的SocketChannel虽已打开却是未连接的。在一个未连接的SocketChannel对象上尝试一个I/O操作会导致NotYetConnectedException异常。我们可以通过在通道上直接调用connect()方法或在通道关联的Socket对象上调用connect()来讲socket通道连接。一旦一个socket通道被连接，它将保持连接状态知道被关闭。您可以通过调用布尔类型的isConnected()方法来测试某个SocketChannel当前是否已经连接。

如果您选择使用传统方式进行连接——通过在对等Socket对象上调用connect()方法，那么传统的连接语义将适用于此。线程在连接建立好或超时过期之前都将保持阻塞。如果您选择通过在通道上直接调用connect()方法来建立连接并且通道处于阻塞模式（默认模式），那么连接过程实际上是一样的。

在SocketChannel上并没有一种connect()方法可以让您指定超时（timeout）值，当connect()方法在非阻塞模式下被调用时SocketChannel提供并发连接；它发起队请求地址的连接并且立即返回值。如果返回值是true，说明连接立即建立了（这可能是本地还回连接）；如果连接不能立即建立，connect()方法会返回false且并发地继续连接建立过程。

面向流的socket建立连接状态需要一定的时间，因为两个待连接系统之间必须进行包对话以建立维护流socket所需的状态信息。跨越开放互联网连接到远程系统会特别耗时。假如某个SocketChannel上当前正由一个并发连接，iSConnectPending()方法就会返回true值。

调用finishConnect()方法来完成连接过程，该方法任何时候都可以安全地进行调用。假如在一个非阻塞模式的SocketChannel对象上调用finishConnect()方法，将可能出现下列情形之一：

1、connect()方法尚未被调用。那么将产生NoConnectionPendingException异常。

2、连接建立过程正在进行，尚未完成。那么什么都不会发生，finishConnect()方法会立即返回false值。

3、在非阻塞模式下调用connect()方法之后，SocketChannel又被切换回了阻塞模式。那么如果有必要的话，调用线程阻塞直到连接建立完finishConnect()方法接着就会返回true值。

4、在初次调用connect()或最后一次调用finishConnect()之后，连接建立过程已经完成。那么SocketChannel对象的内部状态将被更新到已连接状态，finishConnect()方法会返回true值，然后SocketChannel对象就可以被用来传输数据了。

5、连接已经建立。那么什么都不会发生，finishConnect()方法会返回true值。