主要内容

# 通道基础

通道（Channel）是java.nio的第二个主要创新。它们既不是一个扩展也不是一项增强，而是全新、极好的Java I/O示例，提供与I/O服务的直接连接。Channel用于在字节缓冲区和位于通道另一侧的实体（通常是一个文件或套接字）之间有效地传输数据。

通道可以形象地比喻为银行出纳窗口使用的启动导管。您的薪水支票就是您要传送的信息，载体（Carrier）就好比一个缓冲区。您先填充缓冲区（将您的支票放到载体上），接着将缓冲“写”到通道中（将载体丢进导管中），然后信息负载就被传送到通道另一侧的I/O服务（银行出纳员）。

通道是一种途径，借助该途径，可以用最小的开销来访问操作系统本身的I/O服务。缓冲区则是通道内部用来发送和接收数据的端点。

与缓冲区不同，通道API主要由接口指定。不同的操作系统上通道实现（ChannelImplementation）会有根本性的差异，所以通道API仅仅描述了可以做什么。因此很自然地，通道实现经常使用操作系统的本地代码。通道接口允许您以一种受控且可移植的方式来访问底层的I/O服务。

您可以从顶层的Channel接口看到，对所有通道来说只有两种共同的操作：检查一个通道是否打开（isOpen()）和关闭一个打开的通道（close()）。所有有趣的东西都是那些Channel接口以及它的子接口的类。

通道是访问I/O服务的导管。I/O可以分为广义的两大类别：File I/O和Stream I/O。那么相应地有两种类型的通道也就不足为怪了，它们是文件（file）通道和套接字（socket）通道。你会发现有一个FileChannel类和三个socket通道类：SocketChannel、ServerSocketChannel和DatagramChannel。

通道可以以多种方式创建。Socket通道有可以直接创建新socket通道的工厂方法。但是一个FileChannel对象却只能通过在一个打开的RandomAccessFile、FileInputStream或FileOutputStream对象上调用getChannel()方法来获取。您不能直接创建一个FileChannel对象。File和socket通道会在后面的章节中予以详细讨论。

**使用通道**

通道可以是单向（unidirectional）或者双向（bidirectional）。一个channel类可能实现定义read()方法的ReadableByteChannel接口，而另一个channel类也许实现WritableByteChannel接口以提供write()方法。实现这两种接口其中之一的类都是单向的，只能在一个方向上传输数据。如果一个类**同时实现这两个接口**，那么它是双向的，可以双向传输数据。

通道会连接一个特定I/O服务且通道实例（channel instance）的性能受它所连接的I/O服务的特征限制，记住这很重要。一个连接到只读文件Channel实例不能进行写操作，即使该实例所属的类可能有write()方法。基于此，程序员需要知道通道是如何打开的，避免试图尝试一个底层I/O服务不允许的操作。

通道可以以阻塞（blocking）或非阻塞（nonblocking）模式运行。非阻塞模式的通道永远不会让调用的线程休眠。请求的操作要么立即完成，要么返回一个结果表明未进行任何操作。只有面向流的（stream-oriented）的通道，如sockets和pipes才能使用非阻塞模式。

**关闭通道**

通过调用通道的close方法进行关闭，但是可能会导致关闭底层I/O服务时发生阻塞（非阻塞模式和阻塞模式都一样）

通过isOpen方法来测试通道的开放状态，如果返回true，那么说明通道可以使用。反之，说明通道可以使用。反之，说明通道已经关闭，不能使用。

# 文件通道

FileChannel类

文件通道总是非阻塞式的，因此不能被置于阻塞模式。现代操作系统都有复杂的缓存的**预取机制**，使得本地磁盘I/O操作延迟很少。网络文件系统一般而言延迟会多些，不过却也因该优化而受益。面向流的I/O的非阻塞范例对于面向文件的操作无多大意义，这是由文件I/O本质上的不同性质造成的。对于文件I/O，最强大之处在于异步I/O（asynchronous I/O），它允许一个进程可以从操作系统请求一个或多个I/O操作而不必等待这些操作的完成。发起请求的进程之后会收到它请求的I/O操作已完成的通知。异步I/O是一种高级性能，当前的很多操作系统都不具备。以后的NIO增强也会把异步I/O纳入考虑范围。

FileChannel对象不能直接创建。一个FileChannel实例只能通过在一个打开的file对象（RandomAccessFile， FileInputStream或FileOutputStream）上调用getChannel()方法获取。调用getChannel()方法会返回一个连接到相同文件的FileChannel对象且该FileChannel对象具有与file对象相同的范文权限，然后您就可以使用该通道对象来利用强大的FileChannel API了：

FileChannel对象是**线程安全（thread-safe）**的。多个进程可以在同一个实例上并发调用方法而不会引起任何问题，不过并非所有的操作都是多线程的（multithreaded）。影响通道位置或者影响文件大小的操作都是单线程的（single-threaded）。如果有一个线程已经在执行会影响通道位置或文件大小的操作，那么其他尝试进行此类操作之一的线程必须等待。并发行为也会受到底层的操作系统或文件系统影响。

同大多数I/O相关的类一样，FileChannel是一个反映Java虚拟机外部一个具体对象的抽象。FileChannel类保证同一个Java虚拟机的所有实例看到的某个文件的视图均是一致的，但是Java虚拟机却不能对超出它控制范围的因素提供担保。通过一个FileChannel实例看到的某个文件的视图同通过一个外部的非Java进程看到的该文件的视图可能一致，也可能不一致。多个进程发起的并发文件访问的语义高度取决于底层的操作系统和（或）文件系统。一般而言，由运行在不同Java虚拟机上的FileChannel对象发起的对某个文件的并发访问和由非Java进程发起的对该文件的并发访问是一致的。

**访问文件**

在通道这块我们可以使用FileChannel的read和write方法进行文件的访问，以及配合position()进行文件的操作。

FIleChannel位置（position）是从底层的文件描述符获得的，该position同时被作为通道引用获取来源的文件对象共享。这也就意味着一个对象对该position的更新可以被另一个对象看到。

position能够决定文件中哪一处的数据接下来将被读或写。类似于缓冲区的get()和put()方法，当字节被read()或write()方法传输时，文件position会自动更新。如果position值达到了文件大小的值（文件大小的值可以通过size()方法返回），read()方法会返回一个文件尾的条件值（-1）。可是不同于缓冲区的是，如果实现write()方法时position前进到超过文件大小的值，该文件会扩展，以容纳新写入的字节。