**JAVA NIO**

La libreria NIO (New Input Output) fue introducida con la versión 1.4 de la JDK. Nio provee mecanismos de I/O de alta velocidad, orientados a bloques en el código Java Standard. Definiendo clases para contener la información y procesando la misma en bloques, NIO toma ventaja de optimizaciones a bajo nivel de una forma en la que el paquete original de IO no lo puede hacer.

**NIO vs. IO**

I/O (input/output) se refiere a la interface entre una computadora y el resto del mundo, o entre un programa y el resto de la computadora. I/O se basa en la metáfora de Streams (río o chorro en castellano), donde la información se representa en movimientos de bytes de uno a la vez a través de un objeto llamado Stream. Los Streams son utilizados para conectarse con el mundo exterior.

NIO fue creado para permitir a los programadores Java implementar funciones de alta velocidad de I/O sin la necesidad de tener que hacerlo escribiendo código nativo. NIO mueve las actividades que más tiempo consumen (llenado y drenado de buffers) nuevamente al sistema operativo, permitiendo de esta forma obtener una gran mejora en la velocidad.

**Diferencias principales**

**IO streams, NIO blocks**

La diferencia más importante entre las dos librerías es la forma en que la información es empaquetada y transmitida. Como lo mencionamos anteriormente, I/O utiliza Streams mientras que NIO utiliza bloques.

Un sistema de I/O orientado a Streams trabaja con uno o más bytes a la vez. Un Stream de entrada produce de a un byte de información y un byte de salida consume de un byte de información, el punto es que los bytes no son cacheados en ningún lado, no se puede mover atrás o adelante en un Stream.

En un sistema de I/O orientado a bloques, se trabaja con la información en forma de bloques. Cada operación produce o consume un bloque de información en un solo paso, de esta forma se puede procesar la información mucho más rápido que un sistema orientado a Stream.

**Bloqueante vs. No bloqueante IO**

Los Streams de Java IO son bloqueantes. Esto significa que cuando un thread invoca los métodos read() o write(), ese thread queda bloqueado hasta que haya información para leer o la información haya sido completamente escrita.

Con la capacidad “no bloqueante” de NIO (también llamada asincrónica), un thread puede solicitar que se escriba algo de información en un canal sin necesidad de esperar a que este completamente escrito. El thread puede continuar realizando actividades mientras esto sucede, de esta forma un solo thread puede manejar múltiples canales de entrada o salida.

**Diferencias en la API**

En las clases anteriores vimos como leer un archivo con las clases de IO a través de Streams, veamos a continuación un ejemplo de cómo hacerlo con NIO.

import java.io.IOException;

import java.io.RandomAccessFile;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

public class FileReader

{

    public static void main(String[] args) throws IOException

    {

        RandomAccessFile archivo = new RandomAccessFile("ejemplo.txt", "r");

        FileChannel in = archivo.getChannel();

        ByteBuffer buffer = ByteBuffer.allocate(1024);

        while(in.read(buffer) > 0)

        {

            buffer.flip();

            for (int i = 0; i < buffer.limit(); i++)

            {

                System.out.print((char) buffer.get(i));

            }

            buffer.clear();

        }

        in.close();

        archivo.close();

    }

}

**Conclusión**

NIO permite manejar múltiples canales utilizando solo un (o algunos) threads, pero el costo es que parsear la información puede ser de cierta forma más complicado que cuando se lee la información desde un Stream bloqueante utilizando las clases standard de IO.

Si es necesario manejar miles de conexiones abiertas simultáneamente, cada una enviando pequeñas cantidades de información, como por ejemplo un servidor de chat, en ese caso NIO es seguramente la mejor elección. Por el contrario, si se tienen pocas conexiones con un gran ancho de banda, enviando grandes cantidades de información, la mejor opción es implementar un servidor con las clases de IO.

**La clase Files**

El paquete NIO de Java al igual que el paquete IO nos provee clases para manejar todo tipo de entradas y salidas, ya sean flujos de información que provengan o sean enviados a través de Sockets, URLs o archivos, particularmente en lo que respecta al manejo de archivos la clase Files de Java NIO (java.nio.file.Files) contiene varios métodos para manipular archivos en el file system, como por ejemplo:

* Files.exists: Se utiliza para saber sí un archivo o directorio existe o no.
* Files.createDirectory: Se utiliza para crear un directorio.
* Files.copy: se utiliza para copiar archivos.
* Files.move: se utiliza para mover archivos.
* Files.delete: se utiliza para borrar archivos.

**Files.walkFileTree**

El método Files.walkFileTree() se utiliza para recorrer un árbol de directorios recursivamente.

El método recibe como parámetros una instancia de Path y un FileVisitor.

La instancia de Path apunta al directorio a recorrer.

El FileVisitor es llamado durante el recorrido.

FileVisitor es una interface, por lo cual nosotros debemos codear nuestra propia implementación.

La interface FileVisitor contiene los siguientes métodos:

public interface FileVisitor {

// Se ejecuta antes de visitar un directorio  
public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws IOException;

// Se ejecuta por cada archivo que es visitado  
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException;

// Se llama sí la visita a un archivo falla, por ejemplo por falta de permisos sobre el archivo  
public FileVisitResult visitFileFailed(Path file, IOException exc) throws IOException;

    // Se ejecuta despues de visitar un directorio  
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException;  
  
}

Como observarán cada uno de los métodos devuelve un objeto del tipo FileVisitResult.

FileVisitResult es una enum con los siguientes valores posibles:

* CONTINUE
* TERMINATE
* SKIP\_SIBLINGS
* SKIP\_SUBTREE

A través de estos valores el método llamado puede decidir como el recorrido debe continuar:

* CONTINUE el recorrido debe continuar normalmente.
* TERMINATE el recorrido debe terminar.
* SKIP\_SIBLINGS el recorrido debe continuar, pero los directorios o archivos que se encuentran en el mismo nivel no deben ser recorrido.
* SKIP\_SUBTREE el recorrido debe continuar, pero sin visitar las entradas en este directorio. Este valor puede ser solo devuelto desde preVisitDirectory() de cualquier otra parte será interpretado como un CONTINUE.

En caso de no necesitar todos los métodos expuestos por la interfaz FileVisitor, se puede utilizar la interfaz SimpleFileVisitor que contiene implementaciones por default para todos los métodos de FileVisitor.

**COMPARABLE Y COMPARATOR**

**Comparable**

Los elementos en una colección en Java se pueden ordenar de la siguiente forma:

List<String> nombres= Arrays.asList("Cesar","Amanda","Dario","Benjamin");

System.out.println("lista original:"+nombres);

Collections.sort(nombres);

System.out.println("lista ordenada:"+nombres);

El método Collections.sort() ordena cualquier lista.

Veamos también el siguiente ejemplo, donde se agregan elementos a un conjunto (java.util.Set). Recordemos que la clase TreeSet mantiene sus elementos ordenados:

       Set nombres = new TreeSet();

       nombres.add("Mario");

       nombres.add("Fernando");

       nombres.add("Omar");

       nombres.add("Juana");

       System.out.println("conjunto ordenado:" + nombres);

Esto funciona correctamente, porque los elementos de las colecciones son comparables entre sí.  Para que un objeto sea comparable, su clase debe implementar la interfaz java.lang.Comparable.

De esta forma, si queremos que una lista de objetos (o un conjunto) tenga sus elementos ordenados, y esos objetos son de una clase que hemos programado, es necesario que nuestra clase implemente la interfaz java.lang.Comparable.

Veamos el siguiente ejemplo, una clase Persona, que tiene algunas propiedades:

class Persona {

   private int id;

   private String nombre;

   private java.util.Date fechaNacimiento;

   public Persona() {

   }

   public Persona(int id, String nombre) {

       this.id = id;

       this.nombre = nombre;

   }

   public int getId() {

       return id;

   }

   public void setId(int id) {

       this.id = id;

   }

   public String getNombre() {

       return nombre;

   }

   public void setNombre(String nombre) {

       this.nombre = nombre;

   }

   public java.util.Date getFechaNacimiento() {

       return fechaNacimiento;

   }

   public void setFechaNacimiento(java.util.Date fechaNacimiento) {

       this.fechaNacimiento = fechaNacimiento;

   }

   @Override

   public String toString() {

       return String.format("persona{id:%1s,nombre:%2s}", id, nombre);

   }

}

Para que los objetos al guardarse en una colección (Set o List) estén ordenados por el campo nombre, debemos implementar la interfaz Comparable.

class Persona implements Comparable{....

Y además implementar su método de comparación, aquí es donde definimos qué campo vamos a utilizar para ordenar:

class Persona implements Comparable{

//....

   public int compareTo(Persona o) {

       return this.nombre.compareTo(o.nombre);

//...    }

De esta forma, al utilizar una colección para ordenar, el ordenamiento será automático:  
      Set personas = new TreeSet();

personas.add(new Persona(1, "Mario"));

       personas.add(new Persona(2, "Fernando"));

       personas.add(new Persona(3, "Omar"));

       personas.add(new Persona(4, "Juana"));

       System.out.println("conjunto ordenado de personas: "+personas);

El ordenamiento natural será a través del campo nombre de Persona.

**Comparator**

Si deseamos ordenar una colección por un criterio diferente al natural debemos utilizar un comparador de elementos. Esto se logra implementando la interfaz java.util.Comparator

Para continuar con nuestro ejemplo, creemos la siguiente clase que implemente la interfaz Comparator:

class OrdenarPersonaPorId implements Comparator {

   public int compare(Persona o1, Persona o2) {

       return o1.getId() - o2.getId();

   }

}

El método compare() debe devolver lo siguiente:

Sí o1 < o2 debe devolver un número menor a cero.

Sí o1 == o2 debe devolver cero.

Sí o1 > o2 debe devolver un número mayor a cero.

Para utilizar este comparador, debemos usar el parámetro adicional de Collections.sort()

List otrasPersonas = Arrays.asList(new Persona(4, "Juana"),

            new Persona(2, "Fernando"),

            new Persona(1, "Mario"),

            new Persona(3, "Omar"));

            Collections.sort(otrasPersonas, new OrdenarPersonaPorId());

            System.out.println("lista de personas ordenadas por ID:" + otrasPersonas);

Sí queremos utilizar un TreeSet podemos pasar como parámetro el comparador en el constructor de java.util.TreeSet

Set conjuntoPersonas = new TreeSet(new OrdenarPersonaPorId());

conjuntoPersonas.add(new Persona(3, "Omar"));

conjuntoPersonas.add(new Persona(4, "Juana"));

conjuntoPersonas.add(new Persona(2, "Fernando"));

conjuntoPersonas.add(new Persona(1, "Mario"));

System.out.println("conjunto de personas ordenadas por ID:" + conjuntoPersonas);