Entrada/Salida con ficheros en Java

Alberto Cortés <alcortes@it.uc3m.es>

31 de marzo de 2013

Índice general

Capítulo 1

Introducción

1.1. Propósito de este documento

Este documento es una breve introducción a cómo funciona la entrada/salida con ficheros en Java.

Está dirigido a alumnos de primero de carrera sin ninguna experiencia en Java, más allá de los conocimientos mínimos sobre la sintaxis del lenguaje, sus estructuras básicas y los conocimientos de orientación a objetos típicos de los primeros capítulos de cualquier libro de introducción a la programación en Java.

Haré un repaso sucinto del API y las clases involucradas y algunos de los conceptos de sistemas operativos y sistemas de ficheros necesarios. Acompaño las explicaciones de ejemplos.

En este documento no se explican aspectos avanzados del manejo de ficheros como los canales, cerrojos, *pattern-matching*, mapeado directo a memoria de ficheros, multiplexado, *selectors*, acceso aleatorio, operaciones no bloqueantes, *multithreading*, ni técnicas de acceso asíncronas.

La versión de Java utilizada en este documento es la 1.7.0_17, por lo que la información que aquí se incluye trata sobre NIO.2 (*New I/O version 2*), el nuevo sistema de entrada/salida con ficheros introducido en la versión 1.7 de Java en 2011-06-07.

La mayor parte de este documento está basado en The Java Tutorials, para la versión SE 7 de Java.

1.2. Contexto general

La mayoría del soporte de entrada/salida a ficheros en Java se incluye en el paquete `java.nio.file`. Aunque dicho API comprende numerosas clases, solo existen unas pocas de ellas que sirven de puntos de entrada al API, lo que simplifica considerablemente su manejo.

1.3. Sistemas de ficheros

Un fichero es una abstracción del sistema operativo para el almacenamiento genérico de datos. La parte del sistema operativo encargada del manejo de ficheros se denomina "sistema de ficheros".

Los sistemas de ficheros tradicionales se encargan de organizar, almacenar y nombrar (identificar mediante un nombre) los ficheros almacenados en dispositivos de almacenamiento permanente, como los discos duros, las memorias USB de estádo sólido, los DVDs...

Hoy en día, los sistemas de ficheros más conocidos son "ext3" para sistemas operativos Linux, "NTFS" para sistemas basados en Windows NT, como Windows XP o Windows 7, e "ISO9660" y "UDF" para dispositivos ópticos como CDs y DVDs.

Aunque cada sistema de ficheros ofrece su propia visión de los datos almacenados en un disco y los gestiona y ordena a su manera, todos ellos comparten algunos aspectos generales:

- Los ficheros suelen organizarse en estructuras jerárquicas de directorios. Estos directorios son contenedores de ficheros (y de otros directorios) que permiten organizar los datos del disco.
- El nombre de un fichero está relacionado con su posición en el árbol de directorios que lo contiene, lo que permite no sólo identificar unívocamente cada fichero, sino encontrarlo en el disco a partir de su nombre.
- Los ficheros suelen tener una serie de metadatos asociados, como pueden ser su fecha de creación, la fecha de última modificación, su propietario o los permisos que tienen diferentes usuarios sobre ellos (lectura, escritura...). Esto convierte al sistema de ficheros en una base de datos de los contenidos almacenados en el disco que puede ser consultada según múltiples criterios de búsqueda.

Capítulo 2

Paths: nombrado de ficheros

El nombre de un fichero se conoce como su *path* (la traducción más aproximada en castellano sería *ruta*, pero incluso en castellano utilizamos su denominación inglesa).

Por ejemplo, el nombre completo del fichero (su path) donde estoy escribiendo este documento es

```
/home/alcortes/room/current/Java-File-IO/tex/informe.tex
```

En mi ordenador, no puede existir otro fichero con ese mismo nombre, aunque seguramente existen muchos ficheros con el *nombre corto* "informe.tex".

Desde luego el path de un fichero cambia dependiendo del sistema de ficheros utilizado, por ejemplo, en sistemas operativos Linux, todos los ficheros están contenidos en un directorio raiz o en directorios que cuelgan de él.

Sin embargo en sistemas de ficheros tipo NTFS, los ficheros están almacenados en una "unidad" (identificada por una letra del alfabeto), esta unidad puede contener a su vez ficheros o directorios. Tradicionalmente, además, en el mundo Windows, los nombres de ficheros suelen tener una extensión (típicamente un "." y tres letras) que ayuda a identificar el tipo de contenido almacenado en ese fichero.

En ambos sistemas podemos localizar un fichero a partir de su nombre (su path), veamos un par de ejemplos:

- Linux: `/home/alcortes/mi_fichero`: Bajo el directorio raiz (`/`), existe un directorio `/home/`, dentro del cual existe un directorio `/home/alcortes/` (de nombre corto "alcortes"), dentro del cual existe un fichero `/home/alcortes/mi_fichero`, de nombre corto "mi_fichero".

- Windows 7: `C:\Mis documentos\fichero.txt`: Bajo la unidad `C`, existe un directorio `C:\Mis documentos\`, dentro del cual existe un fichero `C:\Mis documentos\fichero.txt`, de nombre corto "fichero.txt", que tiene una extensión típica de los antiguos sistemas Windows de tres letras ("txt").

Como se ve en estos ejemplos, los paths tienen un aspecto bien diferente según el sistema operativo, en Linux todo cuelga del directorio raiz y se utiliza el carácter `/` para separar directorios y ficheros entre si. En Windows en cambio, se parte de una letra de unidad y se separan los directorios con el carácter `\`.

También es importante comentar que cada sistema operativo tiene sus propias restricciones sobre otros aspectos del path, por ejemplo, en el sistema de ficheros FAT (utilizado en sistemas MSDOS) no se distinguían entre mayúsculas y minúsculas, por lo que el fichero `C:\FiChEro.TXT` era el mismo que `C:\fichero.txt`. Tampoco soportaba nombres cortos de fichero de más de 8 caracteres, ni caracteres fuera del estándar ASCII (como á, ñ, Å, ¿, $\alpha \ldots$). La mayoría de estas limitaciones se han ido resolviendo en los nuevos sistemas de ficheros.

2.1. Paths relativos y absolutos

Hasta ahora solo hemos visto ejemplos de path absolutos, que son aquellos que permiten identificar unívocamente un fichero y su localización sin necesidad de más datos.

Un path relativo especifica la ruta a un fichero a partir de un directorio de referencia, que se conoce como el *directorio de trabajo*.

Veamos algunos ejemplos:

- Si el directorio de trabajo es `/home/alcortes/`, el path relativo `mi_fichero` identifica al fichero `/home/alcortes/mi_fichero`.
- Si el directorio de trabajo es `/home/alcortes/`, el path relativo `mi_proyecto/mi_fichero` identifica al fichero `/home/alcortes/mi_proyecto/mi_fichero`.
- Si el directorio de trabajo es `/home/alcortes/`, el path relativo `../mcfp/mi_fichero` identifica al fichero `/home/mcfp/mi_fichero`. El directorio `../` es una alias al directorio padre de aquel al que sucede.
- Si el directorio de trabajo es `/home/alcortes/`, el path relativo `./mi_proyecto/mi_fichero` identifica al fichero `/home/alcortes/mi_proyecto/mi_fichero`. El directorio `./` es una alias del directorio de trabajo.

Para distinguir un path relativo de uno absoluto, la clave está en darse cuenta que un path relativo no viene precedido de la raiz del sistema de ficheros (`/` en Linux) o por una letra de unidad en Windows (`C:\` por ejemplo).

2.2. Paths en Java

La interfaz `java.nio.file.Path` representa un path y las clases que implementen esta interfaz puede utilizarse para localizar ficheros en el sistema de ficheros.

La forma mas sencilla de construir un objeto que cumpla la interfaz `Path` es a partir de la clase `java.nio.file.Paths`, que tiene métodos estáticos que retornan objetos `Path` a partir de una representación tipo `String` del path deseado, por ejemplo:

```
Path p = Paths.get("/home/alcortes/mi_fichero");
```

Por supuesto, no es necesario que los ficheros existan de verdad en el disco duro para que se puedan crear los objetos `Path` correspondientes: La representación y manejo de paths en Java no está restringida por la existencia de esos ficheros o directorios en el sistema de ficheros.

El interfaz `Path` declara numerosos métodos que resultan muy útiles para el manejo de paths, como por ejemplo, obtener el nombre corto de un fichero, obtener el directorio que lo contiene, resolver paths relativos, etc.

Nótese que trabajar con paths no tiene nada que ver con trabajar con el contenido de los ficheros que representan, por ejemplo, modificar el contenido de un fichero es una operación que poco tiene que ver con su nombre o su localización en el sistema de ficheros.

Una instancia de tipo `Path` refleja el sistema de nombrado del sistema operativo subyacente, por lo que objetos path de diferentes sistemas operativos no pueden ser comparados fácilmente entre si.

2.3. Ejemplo de uso

PathExample.java

```
1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3
4  class PathExample {
5    public static void main(String args[]) {
6      Path path = Paths.get("/home/alcortes/my_file");
7      System.out.println("path = " + path);
8      System.out.println("is absoute? = " + path.isAbsolute());
9      System.out.println("file short name = " + path.getFileName());
10     System.out.println("parent = " + path.getParent());
11     System.out.println("uri = " + path.toUri());
12   }
13 }
```

```
1  ; javac PathExample.java
2  ; java PathExample
3  path = /home/alcortes/my_file
4  is absoute? = true
5  file short name = my_file
6  parent = /home/alcortes
7  uri = file:///home/alcortes/my_file
```

Capítulo 3

Ficheros

La clase `java.nio.file.Files` es el otro punto de entrada a la librería de ficheros de Java. Es la que nos permite manejar ficheros *reales* del disco desde Java.

Esta clase tiene métodos estáticos para el manejo de ficheros, lo que permite crear y borrar ficheros y directorios, comprobar su existencia en el disco, comprobar sus permisos, moverlos de un directorio a otro y lo más importante, leer y escribir el contenido de dichos ficheros.

Veamos cómo se realizan algunas de estas operaciones.

3.1. Existencia y comprobación de permisos

ExistanceChecks.java

```
1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3  import java.nio.file.Files;
4
5  class ExistanceChecks {
6    public static void main(String args[]) {
7      Path path = Paths.get("./ExistanceChecks.java");
8      System.out.println("path = " + path);
9      System.out.println("  exists = " + Files.exists(path));
10     System.out.println("  readable = " + Files.isReadable(path));
11     System.out.println("  writeable = " + Files.isWritable(path));
12     System.out.println("  executeable = " + Files.isExecutable(path));
13
14     path = Paths.get("/this_file_doesn't_exist");
15     System.out.println("path = " + path);
16     System.out.println("  exists = " + Files.exists(path));
17     System.out.println("  readable = " + Files.isReadable(path));
18     System.out.println("  writeable = " + Files.isWritable(path));
19     System.out.println("  executeable = " + Files.isExecutable(path));
20   }
21 }
```

```
1 path = ./ExistanceChecks.java
2   exists = true
```

```
  readable = true
  writeable = true
  executeable = false
path = /this_file_doesn't_exist
  exists = false
  readable = false
  writeable = false
  executeable = false
```

3.2. Creación y borrado

CreateOrDelete.java

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;

// Creates a new file or delete it, if it already exists
class CreateOrDelete {
  private static void usage() {
    System.err.println("java CreateOrDelete <file>");
    System.err.println("  The <file> argument is required.");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 1)
      usage();

    Path path = Paths.get(args[0]);
    try {
      if (Files.exists(path))
        Files.delete(path);
      else
        Files.createFile(path);
    } catch (IOException e) {
      System.err.println(e);
      System.exit(1);
    }
  }
}
```

```
; java CreateOrDelete
java CreateOrDelete <file>
  The <file> argument is required.
; ls
CreateOrDelete.class  CreateOrDelete.java
; java CreateOrDelete bla
; ls
bla  CreateOrDelete.class  CreateOrDelete.java
; java CreateOrDelete bla
; ls
CreateOrDelete.class  CreateOrDelete.java
; java CreateOrDelete /root/bla
java.nio.file.AccessDeniedException: /root/bla
```

Capítulo 4

Lectura de ficheros

La lectura de ficheros en Java puede realizarse de varias maneras.

Para ficheros pequeños resulta cómodo meter todo el contenido del fichero en un array de bytes y procesarlo de la forma habitual cuando se manejan arrays.

Para ficheros más grandes, un array resulta incómodo e ineficiente, por lo que se opta por utilizar *buffers* de acceso secuencial que permiten un acceso cómodo y eficiente al contenido del fichero.

4.1. Tablas de caracteres (charsets)

Una tabla de caracteres es una asociación entre números y letras del alfabeto. Su propósito es asignar un número a cada letra, de forma que se puedan almacenar letras como números y posteriormente identificar dichos números y traducirlos a letras.

Las tablas de caracteres son necesarias ya que en disco solo pueden guardarse números (binarios). Por lo tanto si queremos almacenar un fichero textual en disco, tendremos que almacenar los números correspondiente a cada una de sus letras.

Tradicionalmente, la tabla de caracteres más utilizada ha sido la tabla ASCII, que reproduzco a continuación.

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL	32	SPACE	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL
Dec	Char	Dec	Char	Dec	Char	Dec	Char

Esto quiere decir que para almacenar una ’a’ en disco, si usamos la tabla ASCII, guardaremos el número 97. De la misma forma, cuando interpretamos un fichero textual de disco, como ASCII, si nos encontramos un 97, mostramos por pantalla una ’a’.

Nótese que la tabla ASCII no soporta caracteres típicos del castellano como las letras acentuadas o la ’ñ’.

Existen muchas otras tablas de caracteres, por ejemplo, hasta hace poco para castellano se ha utilizado la tabla ISO-8859-1 (latin-1) y más recientemente casi todos los sistemas operativos se han pasado a UTF-8.

Cuando un fichero textual se guarda en disco, no se almacena qué tipo de tabla de caracteres se ha utilizado para traducir de texto a números, por lo que al

volver a leerlo no sabremos que tabla de caracteres utilizar y si nos equivocamos, estaremos viendo un texto diferente al que se escribió originalmente.

Por eso, es habitual utilizar siempre la tabla de caracteres por defecto de la máquina virtual Java, de forma que quien los lea, pueda volver a utilizarla para descodificarlos.

La clase abstracta `java.nio.charset.Charset` se utiliza en Java para controlar las tablas de caracteres que queremos utilizar. Por ser una clase abstracta no puede instanciarse directamente, pero allá donde queramos utilizar una tabla concreta podremos nombrarla a partir de una descripción textual de su nombre. Por ejemplo, el siguiente programa imprime por pantalla la tabla de caracteres por defecto de la JVM y la configura para usar Latin1 para la entrada/salida a fichero.

CharsetExample.java

```
1  import java.nio.charset.Charset;
2
3  class CharsetExample {
4    public static void main(String args[]) {
5      // find default charset
6      System.out.println("Default Charset = " + Charset.defaultCharset());
7
8      // Use Latin1 for file i/o instead of the default charset
9      System.setProperty("file.encoding", "ISO-8859-1");
10     System.out.println("file.encoding = " + System.getProperty("file.
           encoding"));
11
12     // Example of directly using charset objects
13     Charset ascii = Charset.forName("US-ASCII");
14     System.out.println("Standard charset in old systems = " + ascii);
15   }
16 }
```

```
1 ; javac CharsetExample.java
2 ; java CharsetExample
3 Default Charset = UTF-8
4 file.encoding = ISO-8859-1
5 Standard charset in old systems = US-ASCII
```

4.2. Lectura a un array de bytes

Este es el método más arcaico de lectura de ficheros: lee el fichero en su totalidad y lo guarda en un array de bytes, por lo que resulta muy ineficiente en términos de memoria (raramente quieres tener acceso a todo el fichero a la vez, mejor sería ir cargándolo por partes e ir trabajando con cada una de ellas por separado).

Resulta complicado procesar los datos del fichero de esta manera, por ejemplo, para un procesado textual, habrá que ir construyendo `String`s con las porciones del array que representan palabras o líneas de texto, lo cual hay que hacer a mano, a base de buscar espacios o caracteres de final de línea en el array.

Aunque hay maneras de hacer estos procesados automáticamente, este método es inherentemente incómodo al tener que acceder a cada uno de los bytes del contenido del fichero uno por uno.

En general este método no se utiliza mas que para el acceso a ficheros muy pequeños (un par de frases de contenido) o ficheros binarios pequeños, donde el acceso byte a byte tiene sentido y utilidad.

El siguiente ejemplo muestra un programa Java que escribe a la salida estándar el contenido de un fichero cuyo nombre se le pasa como argumento.

```
; javac Cat1.java
; java Cat1 /tmp/bla
ERROR: java.nio.file.NoSuchFileException: /tmp/bla
; java Cat1 Cat1.java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;

// prints the contents of a file using an array of bytes
class Cat1 {
  private static void usage() {
    System.err.println("java Cat1 <file>");
    System.err.println("  A <file> argument is mandatory");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 1)
      usage();

    Path path = Paths.get(args[0]);
    try {
      byte[] content = Files.readAllBytes(path);
      for (int i=0; i<content.length; i++)
        System.out.print((char) content[i]);
      } catch (IOException e) {
        System.err.println("ERROR: " + e);
        System.exit(1);
    }
  }
}
```

El siguiente ejemplo muestra un programa Java que cuenta el número de líneas que tiene un fichero cuyo nombre se le pasa como argumento.

CountLines1.java

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;

// Count (UNIX) lines in a file
class CountLines1 {
  private final static char UNIX_NEWLINE = '\n';

  private static void usage() {
    System.err.println("java CountLines <file>");
    System.err.println("  The <file> argument is mandatory");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 1)
      usage();

    Path path = Paths.get(args[0]);
    long count = 0;
```

```
22      try {
23        byte[] content = Files.readAllBytes(path);
24        for (int i=0; i<content.length; i++)
25          if ((char) content[i] == UNIX_NEWLINE)
26            count++;
27      } catch (IOException e) {
28        System.err.println("ERROR: " + e);
29        System.exit(1);
30      }
31      System.out.println(count);
32    }
33  }
```

```
; javac CountLines1.java
; java CountLines1 CountLines1.java
33
```

Lectura a una lista enlazada de `Strings`

El método `java.nio.file.Files.readAllLines()` puede ser particularmente útil para determinados procesados de texto. Al igual que la lectura directa a un array, tiene el inconveniente de leer todo el fichero de golpe y almacenarlo en memoria. Sin embargo, tiene la ventaja de retornar una lista enlazada de `Strings`, lo que resulta mucho más cómodo que un array de bytes a al hora de procesar ficheros textuales.

4.3. Buffers

Un buffer es una estructura de datos que permite el acceso por trozos a una colección de datos.

Los buffers son útiles para evitar almacenar en memoria grandes cantidades de datos, en lugar de ello, se va pidiendo al buffer porciones pequeñas de los datos que se van procesando por separado.

También resultan muy útiles para que la aplicación pueda ignorar los detalles concretos de eficiencia de hardware subyacente, la aplicación puede escribir al buffer cuando quiera, que ya se encargará el buffer de escribir a disco siguiendo los ritmos más adecuados y eficientes.

4.4. Lectura a un buffer

La clase `java.io.BufferedReader` resulta ideal para leer ficheros de texto y procesarlos. Permite leer eficientemente caracteres aislados, arrays o líneas completas como `String`s.

Cada lectura a un `BufferedReader` provoca una lectura en el fichero correspondiente al que está asociado. Es el propio `BufferedReader` el que se va encargando de recordar la última posición del fichero leído, de forma que posteriores lecturas van accediendo a posiciones consecutivas del fichero.

El método `readLine()` lee una línea del fichero y la retorna en forma de `String`.

El siguiente ejemplo muestra un programa Java que escribe a la salida estándar el contenido del un fichero cuyo nombre se le pasa como argumento.

```
; javac Cat2.java
; java Cat2 Cat2.java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;

// prints the contents of a file using a BufferedReader
class Cat2 {
  private static void usage() {
    System.err.println("java Cat2 <file>");
    System.err.println("  A <file> argument is mandatory");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 1)
      usage();

    Path path = Paths.get(args[0]);
    try {
      BufferedReader reader =
        Files.newBufferedReader(path, Charset.defaultCharset());
      String line;
      while ( (line = reader.readLine()) != null )
        System.out.println(line);
      reader.close();
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
  }
}
```

La diferencia de eficiencia con el programa *Cat1*, que hacía lo mismo con un array de bytes, es notable incluso con ficheros pequeños:

```
; time java Cat1 Cat2.java > /dev/null

real  0m0.101s
user  0m0.088s
sys 0m0.012s
; time java Cat2 Cat2.java > /dev/null

real  0m0.086s
user  0m0.064s
sys 0m0.012s
```

El siguiente programa Java cuenta las líneas de un fichero utilizando lectura desde buffer:

CountLines2.java

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;

```

```
8  // Count (UNIX) lines in a file
9  class CountLines2 {
10   private static void usage() {
11     System.err.println("java CountLines2 <file>");
12     System.err.println("  A <file> argument is mandatory");
13     System.exit(1);
14   }
15
16   public static void main(String args[]) {
17     if (args.length != 1)
18       usage();
19
20     Path path = Paths.get(args[0]);
21     long count = 0;
22     try {
23       BufferedReader reader =
24         Files.newBufferedReader(path, Charset.defaultCharset());
25       while ( reader.readLine() != null )
26         count++;
27       reader.close();
28     } catch (IOException e) {
29       System.err.println("ERROR: " + e);
30       System.exit(1);
31     }
32     System.out.println(count);
33   }
34 }
```

```
; javac CountLines2.java
; java CountLines2 CountLines2.java
34
```

Capítulo 5

Escritura de ficheros

5.1. Modos de acceso, el parámetro `OpenOptions`

A la hora de utilizar un fichero en Java se puede restringir el acceso que tenemos al mismo desde el propio lenguaje, haciendo más estrictos los permisos de acceso que dicho fichero ya tiene en el sistema de ficheros.

Por ejemplo, si el usuario tiene permisos de lectura y escritura sobre un fichero, un programa Java que solo quiera leerlo puede abrir el fichero solo en modo lectura, lo que ayudará a evitar bugs desde el propio lenguaje.

A tal efecto, en java se definen una serie de *modos* de acceso a un fichero a través del parámetro `OpenOptions`. La forma más cómoda de utilizar este parámetro es a través del `enum StandardOpenOptions` que puede tomar los siguientes valores (hay más):

- `WRITE`: habilita la escritura en el fichero
- `APPEND`: todo lo escrito al fichero se hará al final del mismo
- `CREATE_NEW`: crea un fichero nuevo y lanza una excepción si ya existía
- `CREATE`: crea el fichero si no existe y simplemente lo abre si ya existía
- `TRUNCATE_EXISTING`: si el fichero existe, y tiene contenido, se ignora su contenido para sobreescribirlo desde el principio.

Los métodos que se muestran en las siguientes secciones utilizan este parámetro, en la descripción de cada método en el API se explica cual es el comportamiento por defecto en caso de no utilizarse este parámetro.

5.2. Escritura desde arrays de bytes

La escritura a ficheros mediante arrays es la forma más sencilla (y limitada) de escritura de ficheros, y se realiza mediante el método `java.nio.file.Files.write()`.

Cp1.java

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.file.StandardOpenOption;

// Copy a file
class Cp1 {
  private static void usage() {
    System.err.println("java Cp1 <input file> <output file>");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 2)
      usage();

    Path inputFile = Paths.get(args[0]);
    Path outputFile = Paths.get(args[1]);

    try {
      byte[] contents = Files.readAllBytes(inputFile);
      Files.write(outputFile, contents,
                  StandardOpenOption.WRITE,
                  StandardOpenOption.CREATE,
                  StandardOpenOption.TRUNCATE_EXISTING);
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
  }
}
```

```
; javac Cp1.java
; ls
Cp1.class  Cp1.java
; java Cp1 Cp1.class bla
; diff -sq Cp1.class bla
Files Cp1.class and bla are identical
```

5.3. Escritura desde buffers

Al igual que en el caso de la lectura, la escritura desde buffers resulta mucho más eficiente que utilizando arrays de bytes para ficheros grandes.

El siguiente programa Java copia ficheros, accediendo al fichero original una vez por línea y escribiendo en el fichero destino una línea cada vez:

Cp2.java

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.nio.file.StandardOpenOption;

// Copy a file
```

```
11 class Cp2 {
12   private static void usage() {
13     System.err.println("java Cp2 <input file> <output file>");
14     System.exit(1);
15   }
16
17   public static void main(String args[]) {
18     if (args.length != 2)
19       usage();
20
21     Path input = Paths.get(args[0]);
22     Path output = Paths.get(args[1]);
23
24     try {
25       BufferedReader inputReader =
26         Files.newBufferedReader(input, Charset.defaultCharset());
27       BufferedWriter outputWriter =
28         Files.newBufferedWriter(output, Charset.defaultCharset(),
29                                 StandardOpenOption.WRITE,
30                                 StandardOpenOption.CREATE,
31                                 StandardOpenOption.TRUNCATE_EXISTING);
32
33       String line;
34       while ( (line = inputReader.readLine()) != null ) {
35         outputWriter.write(line, 0, line.length());
36         outputWriter.newLine();
37       }
38
39       inputReader.close();
40       outputWriter.close();
41     } catch (IOException e) {
42       System.err.println("ERROR: " + e);
43       System.exit(1);
44     }
45   }
46 }
```

```
; javac Cp2.java
; java Cp2 Cp2.class bla
ERROR: java.nio.charset.MalformedInputException: Input length = 1
; java Cp2 Cp2.java bla
; diff -sq Cp2.java bla
Files Cp2.java and bla are identical
```

5.4. Ejemplos más avanzados

El siguiente programa Java, lee un fichero, ignora aquellas líneas que no están en mayúsculas y el resto las guarda en un segundo fichero:

CopyUpperCase.java

```
1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3  import java.nio.file.Files;
4  import java.io.IOException;
5  import java.nio.charset.Charset;
6  import java.io.BufferedReader;
7  import java.io.BufferedWriter;
8  import java.nio.file.StandardOpenOption;
9
10 // Copy uppercase lines of file
11 class CopyUpperCase {
```

```
  private static void usage() {
    System.err.println("java CopyUpperCase <input file> <output file>");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 2)
      usage();

    Path input = Paths.get(args[0]);
    Path output = Paths.get(args[1]);

    try {
      BufferedReader inputReader =
        Files.newBufferedReader(input, Charset.defaultCharset());
      BufferedWriter outputWriter =
        Files.newBufferedWriter(output, Charset.defaultCharset(),
                                StandardOpenOption.WRITE,
                                StandardOpenOption.CREATE,
                                StandardOpenOption.TRUNCATE_EXISTING);

      String line;
      while ( (line = inputReader.readLine()) != null ) {
        if (line.equals(line.toUpperCase())) {
          outputWriter.write(line, 0, line.length());
          outputWriter.newLine();
        }
      }

      inputReader.close();
      outputWriter.close();
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
  }
}
```

```
; javac CopyUpperCase.java
; cat test
This line has lowercase letters.
THIS LINE IS ALL IN UPPERCASE.
THIS LINE TOO.
this line is not in uppercase.
this LINE is NOT in uppercase.
THIS LAST LINE IS ALL IN UPPERCASE.
; java CopyUpperCase test output
; cat output
THIS LINE IS ALL IN UPPERCASE.
THIS LINE TOO.
THIS LAST LINE IS ALL IN UPPERCASE.
```

El siguiente programa Java, lee un fichero, busca aquellas lineas que mencionan un texto pasado como argumento, y las imprime por pantalla:

Grep.java

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;

// Search for a text in a file
class Grep {
```

```
10   private static void usage() {
11     System.err.println("java Grep <input file> <pattern>");
12     System.exit(1);
13   }
14
15   public static void main(String args[]) {
16     if (args.length != 2)
17       usage();
18
19     Path input = Paths.get(args[0]);
20
21     try {
22       BufferedReader inputReader =
23         Files.newBufferedReader(input, Charset.defaultCharset());
24
25       String line;
26       long lineNumber = 1;
27       while ( (line = inputReader.readLine()) != null ) {
28         if (line.contains(args[1]))
29           System.out.println(lineNumber + ": " + line);
30         lineNumber++;
31       }
32
33       inputReader.close();
34     } catch (IOException e) {
35       System.err.println("ERROR: " + e);
36       System.exit(1);
37     }
38   }
39 }
```

```
; javac Grep.java
; java Grep Grep.java import
1: import java.nio.file.Path;
2: import java.nio.file.Paths;
3: import java.nio.file.Files;
4: import java.io.IOException;
5: import java.nio.charset.Charset;
6: import java.io.BufferedReader;
; java Grep Grep.java line
25:       String line;
26:       long lineNumber = 1;
27:       while ( (line = inputReader.readLine()) != null ) {
28:         if (line.contains(args[1]))
29:           System.out.println(lineNumber + ": " + line);
30:         lineNumber++;
; java Grep Grep.java (
bash: syntax error near unexpected token `('
; java Grep Grep.java "("
10:   private static void usage() {
11:     System.err.println("java Grep <input file> <pattern>");
12:     System.exit(1);
15:   public static void main(String args[]) {
16:     if (args.length != 2)
17:       usage();
19:     Path input = Paths.get(args[0]);
23:         Files.newBufferedReader(input, Charset.defaultCharset());
27:       while ( (line = inputReader.readLine()) != null ) {
28:         if (line.contains(args[1]))
29:           System.out.println(lineNumber + ": " + line);
33:       inputReader.close();
34:     } catch (IOException e) {
35:       System.err.println("ERROR: " + e);
36:       System.exit(1);
```

Capítulo 6

Ficheros binarios

La lectura de ficheros binarios pequeños puede resolverse mediante la lectura de todo el fichero a un array de bytes (4.2). Sin embargo, para ficheros binarios grandes, este método resulta poco eficiente en términos de uso de memoria.

Lo ideal sería usar un buffer de lectura, al estilo de `BufferedReader` (4.4), sin embargo esta clase está pensada para leer ficheros de texto, por lo que no resulta fácil transformar las `String`s y caracteres leídos en bytes.

Lo que necesitamos es una acceso mediante buffer pero sin la funcionalidad añadida de la traducción de los bytes a texto. Para esto, lo mejor es usar `java.io.BufferedInputStream`, cuyo método `read(byte[] b, int off, int len)` permite leer de forma eficiente la cantidad deseada de bytes desde cualquier posición del fichero.

La escritura de ficheros binarios tiene el mismo problema, y podemos utilizar la clase `java.io.BufferedOutputStream` para solucionarlo, de forma que podamos invocar escrituras al buffer mediante `write(byte[] b, int off, int len)` delegando en la JVM y al sistema operativo la complicada tarea de cómo escribir esos bytes a disco de forma eficiente.

El siguiente programa copia ficheros binarios (o de cualquier otro tipo) leyendo y escribiendo un número configurable de bytes cada vez.

Dd.java

```
1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3  import java.nio.file.Files;
4  import java.io.IOException;
5  import java.nio.charset.Charset;
6  import java.io.BufferedInputStream;
7  import java.io.BufferedOutputStream;
8  import java.nio.file.StandardOpenOption;
9
10 // Copy files using binary buffers
11 class Dd {
12   private static void usage() {
13     System.err.println("java Dd <input file> <output file> <buffer size>");
14     System.exit(1);
15   }
16
```

```
17   public static void main(String args[]) {
18     if (args.length != 3)
19       usage();
20
21     Path inputPath = Paths.get(args[0]);
22     Path outputPath = Paths.get(args[1]);
23
24     try {
25       int bufferSize =  Integer.parseInt(args[2]);
26       if (bufferSize <= 0)
27           throw new NumberFormatException(args[2] + " is not positive");
28
29       BufferedInputStream input;
30       BufferedOutputStream output;
31       input  = new BufferedInputStream(
32                    Files.newInputStream(inputPath,
33                      StandardOpenOption.READ));
34       output = new BufferedOutputStream(
35                    Files.newOutputStream(outputPath,
36                      StandardOpenOption.WRITE,
37                      StandardOpenOption.CREATE,
38                      StandardOpenOption.TRUNCATE_EXISTING));
39
40       byte[] buffer = new byte[bufferSize];
41       int bytesRead = input.read(buffer, 0, bufferSize);
42       while ( bytesRead >= 0 ) {
43         output.write(buffer, 0, bytesRead);
44         bytesRead = input.read(buffer, 0, bufferSize);
45       }
46
47       input.close();
48       output.close();
49     } catch (IOException e) {
50       System.err.println("ERROR: " + e);
51       System.exit(1);
52     } catch (NumberFormatException e) {
53       System.err.println("ERROR: Bad number format: " + e);
54       System.exit(1);
55     }
56   }
57 }
```

```
; # create a 10 MB file with random contents
; dd if=/dev/urandom of=/tmp/input bs=100000 count=100
100+0 records in
100+0 records out
10000000 bytes (10 MB) copied, 1.35976 s, 7.4 MB/s
;
;
; # copy the file using a 1024 bytes buffer
; java Dd /tmp/input /tmp/output 1024
; diff -sq /tmp/input /tmp/output
Files /tmp/input and /tmp/output are identical
;
;
; # copy the file one byte at a time, this is slow
; # even if we use buffers!!
; time java Dd /tmp/input /tmp/output 1

real  0m1.168s
user  0m1.100s
sys 0m0.060s
;
;
; # copy the file using a 1024 byte buffer, this is much faster
; # in user time, but can still be slow on real time
; time java Dd /tmp/input /tmp/output 1024
```

```
26
27 real  0m0.168s
28 user  0m0.120s
29 sys 0m0.032s
30 ;
31 ;
32 ; # copy the file using a 1MBi byte buffer, this is waaay faster
33 ; time java Dd /tmp/input /tmp/output 1048576
34
35 real  0m0.114s
36 user  0m0.068s
37 sys 0m0.044s
```