Programación concurrente – Gestión de procesos en Java

Ahora que sabemos qué es un proceso, veamos cómo los maneja Java. De hecho, sólo hay unas pocas clases y métodos que necesitamos conocer, ya que Java se centra en subprocesos, no en procesos (cada aplicación principal de Java es, de hecho, un subproceso). Sin embargo, se añaden algunas funcionalidades que nos permiten llamar a programas externos o crear procesos desde una aplicación Java.

1. Creando procesos

Para crear un proceso en Java, necesitamos obtener un objeto `Process`. Recuerda que esto se puede conseguir de dos formas diferentes:

- Usando la clase `ProcessBuilder`. Tenemos que crear un array de Strings con el nombre del programa o comando a ejecutar y sus argumentos, y luego llamar al método `start` .

```
String[] cmd = {"ls", "-l"};
ProcessBuilder pb = new ProcessBuilder(cmd);
Process p = pb.start();
```

- Usando la clase estática `Runtime` . En este caso también tendremos que crear un array de Strings con el nombre y sus argumentos, pero luego llamaremos al método `exec` con el array creado como parámetro.

```
String[] cmd = {"notepad.exe"};
Runtime rt = Runtime.getRuntime();
Process p = rt.exec(cmd);
```

En ambos casos, estamos ejecutando un comando o programa existente en el sistema operativo donde se está ejecutando Java actualmente. Puede ser un shellscript de Linux, un archivo exe de Windows o incluso otra aplicación Java mediante el comando *java*. Si no se puede encontrar el programa, o no tenemos permiso para ejecutarlo, se generará una excepción cuando intentemos llamar a los métodos `start` o `exec` desde las clases `ProcessBuilder` o `Runtime`, respectivamente. Esta excepción será un subtipo de `IOException`.

```
try
{
    Process p = pb.start();
    ...
} catch (IOException e) {
    System.err.println("Exception: " + e.getMessage());
    System.exit(-1);
}
```

1.1. Diferencias entre *ProcessBuilder* y *Runtime*

Quizás te preguntes... ¿por qué existen 2 formas de hacer lo mismo? Bueno, la clase `Runtime` pertenece al núcleo de Java desde su primera versión, mientras que `ProcessBuilder` se agregó en Java 5. Con `ProcessBuilder` puedes añadir variables de entorno y cambiar el directorio de trabajo actual para que se inicie el proceso. Estas funciones no están disponibles para la clase `Runtime`. Además, existen algunas diferencias sutiles entre estas dos clases. Por ejemplo, la clase `Runtime` nos permite ejecutar un comando pasando la cadena completa como argumento, sin dividirla en argumentos separados en un array:

```
Process p = Runtime.getRuntime.exec("ls -l");
```

2. Sincronización de procesos

Acabamos de aprender cómo crear e iniciar un proceso en Java. Después de llamar al método `start` o `exec`, nuestro programa Java continúa y ejecuta la siguiente instrucción. Si queremos que se detenga hasta que el subproceso finalice su tarea, podemos llamar al método `waitFor` desde el objeto `Process` que creamos. Esto hace que el programa principal se detenga hasta que se complete este proceso.

Llamar al método `waitFor` puede generar una `InterruptedException` si el subproceso se interrumpe inesperadamente. Si todo sale bien, el control vuelve a la aplicación principal de Java tan pronto como finaliza el subproceso.

```
try
{
    Process p = pb.start();
    p.waitFor();
    ...
} catch (IOException e) {
    System.err.println("Exception: " + e.getMessage());
    System.exit(-1);
} catch (InterruptedException e) {
    System.err.println("Interrupted: " + e.getMessage());
}
```

El método `waitFor` devuelve un valor entero. Este valor suele ser 0 cuando el proceso ha finalizado correctamente, y cualquier otro número si finalizó irregularmente. Podemos verificar el estado final de un proceso comprobando su valor de retorno:

```
int value = p.waitFor();
if (value != 0)
    System.out.println("The task finished unexpectedly");
```

3. Finalizando procesos

Podemos finalizar un proceso que creamos previamente en nuestro programa llamando al método `destroy()`. Al hacer esto, el recolector de basura de Java liberará todos los recursos asociados a ese proceso.

```
ProcessBuilder pb = new ProcessBuilder(...)
Process p = pb.start();
...
p.destroy();
```

4. Comunicación con los procesos

Por lo general, un proceso necesita obtener cierta información (del usuario o de un archivo, por ejemplo) y generar algunos resultados (en un archivo, en una pantalla...). En muchos sistemas operativos, cuando un proceso utiliza una entrada/salida determinada, sus hijos utilizan la misma entrada/salida. En otras palabras, si un proceso lee datos de un archivo como entrada estándar y crea un subproceso, este subproceso también tendrá el mismo archivo como entrada predeterminada.

Sin embargo, **Java no** tiene ese comportamiento. Cuando un proceso se crea en Java a partir de otro proceso (padre), tiene su propia interfaz de comunicación. Si queremos comunicarnos con este subproceso, tenemos que obtener sus flujos de entrada y salida. Al hacer esto, podremos enviar datos a ese subproceso desde su proceso principal y obtener sus resultados también de su proceso principal, estableciendo un camino entre ambos.

El siguiente ejemplo obtiene el resultado del subproceso y lo muestra en pantalla:

```
Process p = pb.start();
BufferedReader br = new BufferedReader(new
InputStreamReader(p.getInputStream()));
String line = "";

System.out.println("Process output:");
while ((line = br.readLine()) != null)
{
    System.out.println(line);
}
```

Hay algo que debes saber cuándo manejas los datos de un proceso. Algunos sistemas operativos (como Linux, Android, Mac OS X...) utilizan UTF-8 como formato de codificación, mientras que otros sistemas (Windows) utilizan su propio formato de codificación. Esto puede ser un problema si por ejemplo guardamos un archivo de texto en Linux y lo leemos en Windows. Para evitar estos problemas, podemos usar un segundo argumento al crear el objeto `InputStreamReader`, para indicarle a la JVM cuál es el formato de codificación esperado para la entrada:

```
BufferedReader br = new BufferedReader(
new InputStreamReader(p.getInputStream(), "UTF-8"));
```

5. Ejemplo

Este ejemplo crea un proceso para llamar al comando "`ls`" (se espera que se ejecute en Linux o Mac OS X), con la opción "`-l`" para tener una lista detallada de archivos y carpetas del directorio actual. Luego, captura la salida y la imprime en la consola (o salida estándar):

```
import java.io.*;

public class FolderListing
{
    public static void main(String[] args)
    {
        String[] cmd = {"ls", "-l"};
        String line = "";
        ProcessBuilder pb = new ProcessBuilder(cmd);

        try
        {
            Process p = pb.start();
            BufferedReader br = new BufferedReader(
                new InputStreamReader(p.getInputStream()));
            System.out.println("Process output:");

            while ((line = br.readLine()) != null)
            {
                System.out.println(line);
            }
        } catch (Exception e) {
            System.err.println("Exception:" + e.getMessage());
        }
    }
}
```

Ejercicio 1:

Crea un proyecto llamado ProcessListPNG con un programa que le pida al usuario que introduzca una ruta (por ejemplo, /micarpeta/fotos) y luego inicie un proceso que muestre una lista de todas las imágenes PNG encontradas en esta ruta. Intenta hacerlo de forma recursiva (ya sea con un comando del sistema operativo o con tu propio script).

Ejercicio 2:

Crea un proyecto llamado ProcessKillNotepad con un programa que arranque el bloc de notas o cualquier editor de texto similar desde tu sistema operativo. Luego, el programa esperará 10 segundos a que "finalice" el subproceso y, transcurrido ese periodo, será destruido. Para dormir 10 segundos, utilice estas instrucciones:

```
Thread.sleep(milliseconds);
```

Prueba a cerrar manualmente el bloc de notas antes de los 10 segundos, ¿qué sucede?