17 de Junio de 2011

Java Reflection (parte 3)

En las anteriores entradas Java Reflection parte 1 y Java Reflection parte 2) hemos comentado conceptos básicos sobre cómo obtener en tiempo de ejecución los tipos, atributos y métodos de un objeto, y las manipulaciones básicas sobre los mismos (acceder y cambiar sus valores). También hemos explicado algunas manipulaciones más complejas (obtención de instancias a partir de constructores específicos, acceder y modificar elementos privados, ...).

En esta tercera parte nos dedicaremos a explicar cómo trabajar con reflexión con dos elementos un poco más avanzados de Java: arrays y tipos genéricos (Java Generics). Aunque quizás puedan ser considerados elementos básicos del lenguaje de programación (los arrays deben ser el primer tipo complejo que se enseña en las clases de programación), la manipulación vía Java Reflection de los mismos es un poco más compleja que los elementos vistos hasta el momento, aunque descubriremos que una vez aprendidos los conceptos básicos tal complejidad es sólo aparente.

Arrays

La manipulación de arrays mediante Java Reflection se efectúa usando la clase `java.lang.reflect.Array`. Para crear un array debemos usar el método `Array.newInstance(Class arrayClass, int size)`, siendo el primer parámetro la clase de los elementos del array y el segundo el tamaño del array. De esta forma, para crear un array de 3 cadenas:

```
String[] names = (String[]) Array.newInstance(String.class, 3);
```

Una vez tenemos nuestro array creado, podremos acceder y asignar sus elementos mediante los respectivos métodos `Array.get(Object array, int index)` y `Array.set(Object array, int index, Object value)`, donde los parámetros son: `array` = el array en cuestión, `index` = el elemento dentro del array (notación estándar de arrays, el primer elemento tendrá siempre el índice 0), `value` = el valor a asignar.

Además de estos dos métodos genéricos, la clase `Array` contiene unos métodos `get` y `set` específicos para los tipos primitivos de Java. De esta forma tenemos `Array.getBoolean()` y `Array.setBoolean()` para arrays de tipo `boolean`, `Array.getLong()` y `Array.setLong()` para arrays de tipo `long`, ... Los parámetros para estos métodos son los mismos que para los métodos genéricos, con la única salvedad de que el último parámetro del `set()` (`value`, el que contiene el valor a asignar) será del tipo primitivo en cuestión en vez de tipo genérico `Object`.

Como ejemplo, supongamos que tenemos una cadena almacenada en una variable `storedName`, y queremos guardarla en el segundo elemento (índice 1) de nuestro array anterior y almacenar en ella el siguiente valor del array:

```
String storedName;

/* ... */

Array.set(names, 1, storedName);
storedName = (String) Array.get(names, 2);
```

Obtener la clase de un array es ligeramente más complejo que con el resto de objetos, puesto que hay que utilizar una notación un tanto especial. Para obtener la clase de un array mediante el conocido método `Class.forName(String className)`, debemos saber que para indicar un array

debemos iniciar la cadena del parámetro `className` con el carácter `'['` seguido del tipo de clase de los elementos del array. Para tipos primitivos se usará una letra de la siguiente lista:

- `boolean`: `Z`
- `byte`: `B`
- `short`: `S`
- `int`: `I`
- `long`: `J`
- `char`: `C`
- `float`: `F`
- `double`: `D`

En cambio, para elementos de tipo "objetual" se debe usar la letra `L`, seguida del nombre calificado completo de la clase y finalizado con el carácter `';'`.

Así, si queremos obtener la clase de un array de enteros y un array de cadenas, usaremos:

```
Class intArrayClass = Class.forName("[I");
Class stringArrayClass = Class.forName("[Ljava.lang.String;");
```

Una vez tenemos la clase del array, podemos obtener el tipo del componente (es decir, el tipo de los elementos del array) mediante el método `Class.getComponentType()`. Por ejemplo, para nuestro array anterior:

```
Class namesClass = names.getClass();
Class componentClass = namesClass.getComponentType();
```

En este ejemplo, la variable `componentClass` contendrá la clase `java.lang.String` .

Generics

Mediante reflexión también seremos capaces de obtener en tiempo de ejecución información sobre tipos genéricos, aunque el sistema es un poco más complejo y no tan inmediato como cuando tratábamos con tipos "normales".

Crearemos la siguiente clase para nuestros ejemplos, extensión de la clase `User` que definimos en nuestro artículo Java Reflection parte 2:

```
package com.test.model;

public class SocialUser extends User
{
        private List<User> friends;

        public List<User> getFriends()
        {
                return friends;
        }

        public void setFriends(List<User> friends)
        {
                this.friends = friends;
        }
}
```

Como podemos ver, en nuestro ejemplo estamos usando Java Generics para un caso típico: parametrizar los elementos de una lista `Java.util.List` (tanto en la definición del atributo como los métodos que manipulan el mismo).

Primeramente vamos a ver cómo podemos obtener el tipo genérico del atributo `friends` mediante el método `Field.getGenericType()`. Quedará mejor explicado viéndolo directamente con un ejemplo:

```
Field friendsField = SocialUser.class.getDeclaredField("friends");

Type friendsGenericType = friendsField.getGenericType();
ParameterizedType friendsParameterizedType = (ParameterizedType) friendsGenericType;
Type[] friendsType = friendsParameterizedType.getActualTypeArguments();
Class userClass = (Class) friendsType[0];
```

El primer paso es obtener el objeto `Field` asociado al atributo como aprendimos en el primer artículo sobre Java Reflection. Posteriormente obtenemos el tipo genérico en la variable `friendsGenericType`, que como vemos es del tipo `Type` (un interfaz genérico para todos los tipos de Java). Pero el tipo contenido ahí es el genérico (`java.util.List`), mientras que nosotros queremos saber el tipo específico de los elementos de la lista (`User`). Para poder llegar hasta él, necesitamos hacer un cast de nuestro objeto `Type` a `ParameterizedType` (un subinterfaz específico de los tipos parametrizados en Java, recordemos que los tipos genéricos se llaman también tipos parametrizados). Finalmente obtenemos el array de los tipos específicos mediante el método `getActualTypeArguments()` (recordemos que en una declaración de genéricos, se pueden especificar varios tipos específicos). Puesto que esto es un ejemplo, sabemos que sólo tiene un tipo específico y está en el índice 0 del array resultante (`friendsType`). Si queremos utilizarlo como un objeto `Class`, sólo tenemos que asignarlo a una variable con un simple cast de `Type` a `Class`.

> En nuestro ejemplo se simplificó el código para facilitar la comprensión (y porque sabíamos desde el principio cuales iban a ser los tipos devueltos), pero en condiciones normales es una buena práctica asegurarse de la corrección de los tipos devueltos (por ejemplo haciendo un `if (friendsGenericType instanceof ParameterizedType)` antes del cast de `Type` a `ParameterizedType` y no accediendo a ningún elemento del array `friendsType` sin comprobar que ese índice existe).

Ahora que sabemos cómo obtener y manipular tipos genéricos (o parametrizados), obtener el tipo genérico de retorno de una función es tan simple como usar el método `Method.getGenericReturnType()`:

```
Method getFriendsMethod = SocialUser.class.getMethod("getFriends", null);
Type returnType = getFriendsMethod.getGenericReturnType();
```

Con este código de ejemplo, hemos almacenado en la variable `returnType` el objeto que contiene el tipo genérico de retorno de la función `getFriends()`, y cuyo procesamiento es totalmente idéntico al ejemplo anterior.

Finalmente, podemos obtener también en tiempo de ejecución los tipos genéricos de los parámetros de una función, usando `Method.getGenericParameterTypes()`:

```
Method setFriendsMethod = SocialUser.class.getMethod("setFriends", List.class);
Type[] parameterTypes = setFriendsMethod.getGenericParameterTypes();
```

La única diferencia con los dos ejemplos vistos anteriormente es que la variable `parameterTypes` ahora se trata de un array de objetos `Type`, donde cada elemento de dicho array se corresponde con el tipo genérico del parámetro correspondiente en la función `setFriends()` (en nuestro ejemplo un único elemento, pues nuestra función sólo tiene un parámetro).

Creado por Santi | Artículos para programadores, Java, Programación