Servidor HTTP con ejecución dinámica de código

Juan Gutiérrez

5 de noviembre de 2013

Índice

1. Servidor dinámico que puede ejecutar código de "terceros"

El servidor anterior era capaz de ejecutar un proceso ante una petición HTTP, obtener la salida de ese proceso y devolver esa salida como respuesta al cliente.

En este caso lo que queremos es que el código que se va a ejecutar ante una petición HTTP esté definido en un método de una clase no conocida al desarrollar el servidor.

1.1. Objetivo.

Desarrollar un framework tal que:

- El servidor pueda ejecutar código desarrollados por otros. Tendremos que ofrecer un mecanismo para que el servidor sepa qué fichero jar contiene el código y cual es el nombre de la clase que tiene el código a ejecutar.
- Facilitar el desarrollo de código dinámico (que pueda ser ejecutado por el servidor).

1.2. Punto de vista de quien desarrolla el framework

Un framework lo constituyen una serie de tipos (clases o interfaces) que implementa la parte que es general para un determinado tipo de aplicación. Quien quiera usar el framework deberá ímplementar (centrarse en) la parte de código que es dependiente de su problema.

Al desarrollar un framework lo más habitual es tener alguno de estos patrones:

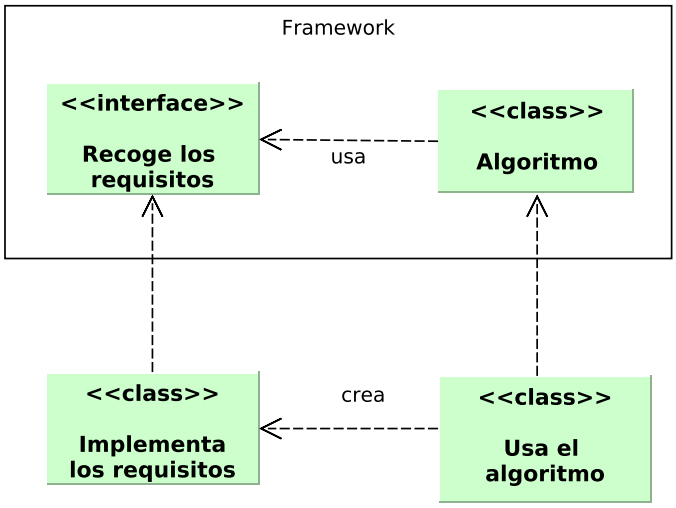

Figura 1: Framework con interfaces

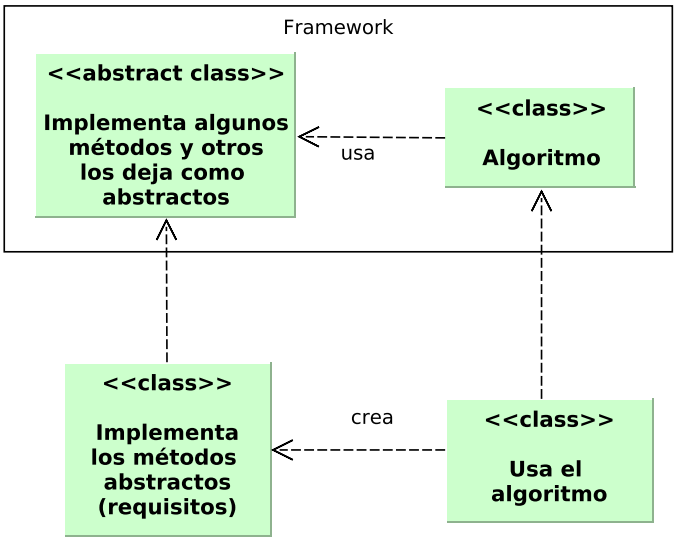

Figura 2: Framework con clases abstractas

Además, al igual que se hizo en el caso del CGI, tendremos que ofrecer algún mecanismo para definir el mapeo de URLs y código dinámico.

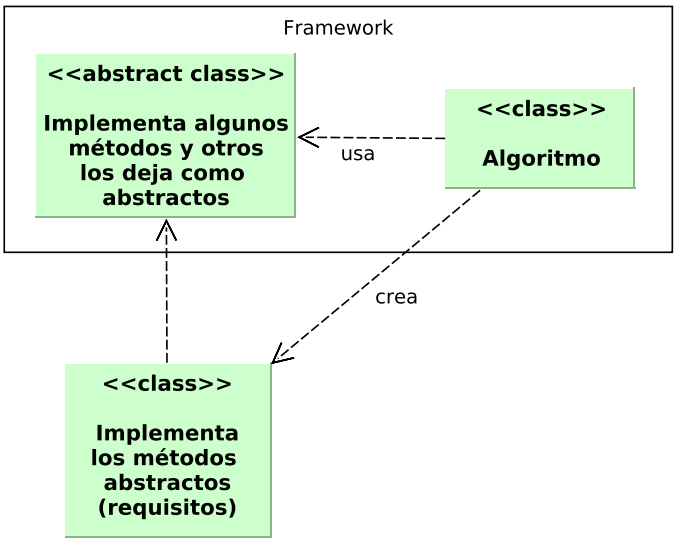

Figura 3: Framework con clases abstractas (2)

1.3. Punto de vista de quien desarrolla el código dinámico

Desde el punto de vista del desarrollo de código dinámico es poder realizar una clase lo más simple posible y que de la creación del objeto y de la llamada a sus métodos se encargue el framework. Algo así como:

```
class CodigoDinamico extends ClaseQueProporcionaElFramework{
   public void ifGet(ClaseDelFrameworkConElementosDeLaPeticion requestThings,
                     ClaseDelFrameworkConElementosDeLaRespuesta responseThings){
      // ¿Qué elementos debería poder obtener de requestThings?

      // ¿Qué elementos debería poder obtener de responseThings?

      // Código para escribir la respuesta
   }
   ...
}
```

Además, el servidor gestionará el ciclo de vida de este objeto y llamará a los métodos de esta clase. Por tanto será responsable pasar los objetos que necesiten.

Supongamos que compilamos esta clase y la almacenamos en un fichero jar (por ejemplo `aplicacion.jar`). Entonces el fichero con los mappings podría contener la siguiente línea:

```
aplicacion;/opt/web/dynamic/aplicacion.jar CodigoDinamico
```

El primer elemento es la URL, el segundo elemento contiene a su vez dos elementos: la localización del fichero jar (con el código compilado) y el nombre de la clase que extiende a la clase que proporciona el framework.

Supongamos que el servidor lee este fichero y obtiene el último de los elementos (el nombre de la clase). Esto es de tipo `String`. ¿Cómo podemos crear un objeto de este tipo si lo que tenemos es su nombre?.

Para esto se puede usar el API *reflection*.

1.4. Reflection

Ver las transparencias proporcionadas.

1.5. Clases del framework que almacenan los elementos de la petición y de la respuesta

```
public class ThingsAboutRequest {
   private HashMap<String, String> params;
   private HashMap<String, String> headers;
   private InputStream in;

   public ThingsAboutRequest(HashMap<String, String> h,
         HashMap<String, String> p, InputStream in) {
      params = p;
      headers = h;
      this.in = in;
   }

   public String getParam(String c) {
      return params.get(c);
   }

   public String getHeader(String c) {
      return headers.get(c);
   }

   public Set<String> getParamNames() {
      return params.keySet();
   }

   public Set<String> getPresentHeaders() {
      return headers.keySet();
   }

   InputStream getInputStream() {
      return in;
   }

}
```

```
public class ThingsAboutResponse {
   private OutputStream out;
   private PrintWriter pw;
   private HashMap<String, String> headers;

   public ThingsAboutResponse(OutputStream o) {
      out = o;
      pw = new PrintWriter(out);
      headers = new HashMap<String, String>();
   }

   public OutputStream getOutputStream() {
      return out;
   }

   public void flushResponseHeaders() {
      for (String h : headers.keySet()) {
         pw.print(h);
         pw.print(": ");
         pw.println(headers.get(h));
      }
      pw.println("");
      pw.flush();
   }
```

```
    public void setResponseHeader(String h, String v) {
        headers.put(h, v);
    }
}
```

1.6. Clase del framework que tendrá que extender quien desarrolle código dinámico

```
public class ResponseClass {
    private Socket canal;
    private String method;
    private String resource;

    void setSocket(Socket s) {
        canal = s;
    }

    void setMethod(String m) {
        method = m;
    }

    void setResource(String r) {
        resource = r;
    }

    public void ifGet(ThingsAboutRequest req, ThingsAboutResponse resp)
            throws Exception {
    }

    public void ifPost(ThingsAboutRequest req, ThingsAboutResponse resp)
            throws Exception {
    }

    public void dealWithCall() throws Exception {

        InputStream in = canal.getInputStream();
        OutputStream out = canal.getOutputStream();

        HashMap<String, String> headers = UtilsHTTP.getHeaders(in);
        HashMap<String, String> params;

        if (method.equals("GET"))
            params = UtilsHTTP.getParamsGet(resource);
        else
            params = UtilsHTTP.parseBody(UtilsHTTP.getBody(headers, in));

        ThingsAboutRequest req = new ThingsAboutRequest(headers, params, in);
        ThingsAboutResponse resp = new ThingsAboutResponse(out);

        PrintWriter pw = new PrintWriter(out);
        UtilsHTTP.writeResponseLineOK(pw);

        if (method != null) {
            if (method.equals("GET")) {
                ifGet(req, resp);
            } else if (method.equals("POST")) {
                ifPost(req, resp);
            }
        }

    }
```

```
}
```

1.7. Clase del framework que usa a la clase anterior

Esta clase representa el algoritmo en las figuras 1, 2 y 3.

```
public class ThreadDynamic implements Runnable {
   private Socket canal;
   private String clase;
   private String request;

   public ThreadDynamic(Socket s, String cl, String req) {
      canal = s;
      clase = cl;
      request = req;
   }

   public void run() {
      try {
         Class<?> c = Class.forName(clase);
         Constructor<?> con = c.getConstructor(new Class<?>[] {});
         ResponseClass rc = (ResponseClass) con.newInstance(new Object[] {});
         rc.setMethod(UtilsHTTP.getMethod(request));
         rc.setResource(UtilsHTTP.getResource(request));
         rc.setSocket(canal);
         rc.dealWithCall();
         canal.close();
         rc = null;
      } catch (Exception ex) {
         ex.printStackTrace();
         try {
            UtilsHTTP.writeResponseServerError(new PrintWriter(canal
                  .getOutputStream()));
         } catch (Exception ex2) {
         }
      }
   }
}
```

1.8. Código para añadir en tiempo de ejecución los fichero jar en el classpath del servidor

La clase `JarPathUtils` tiene métodos para añadir al classpath un fichero jar.

2. Código cliente

El código cliente consiste en una clase que extiende a la clase `ResponseClass`, proporcionada por el *framework*. En esta clase se implementan los métodos `ifGet` o `ifPost` o ambos.

```
//import ...

public class Response1 extends ResponseClass {
   public void ifGet(ThingsAboutRequest req, ThingsAboutResponse resp)
         throws Exception {
      OutputStream out = resp.getOutputStream();
```

```
        PrintWriter pw = new PrintWriter(out);

        resp.setResponseHeader("Content-Type", "text/html; charset=utf-8");
        resp.flushResponseHeaders();

        pw.println("<html>");
        pw.println("<body>");
        pw.println("<h1> Respuesta </h1>");

        Set<String> hdrs = req.getPresentHeaders();

        pw.println("<h2> Campos de cabecera de la peticion </h2>");

        for (String s : hdrs)
           pw.println(s + ": " + req.getHeader(s) + "</br>");

        pw.println("</body>");
        pw.println("</html>");
        pw.flush();
        pw.close();

    }
}
```

Una vez realizado este código de debe generar un fichero jar, y se debe añadir una línea al fichero de los mappings indicando la URL, el fichero jar y el nombre de la clase que extiende a `ResponseClass`.

3. Posibles mejoras al servidor:

1- Implementar el patrón *singleton*. En el código proporcionado se crea un objeto por cada petición. Si hay muchas peticiones estamos generando muchos objetos que provocarán que se lance el recolector de basura en el servidor y esto provocará que las peticiones que lleguen en ese momento tarden más en ser atendidas. En lugar de crear un objeto por cada petición podríamos tener un objeto por cada mapping y reutilizarlo. **Esto se deja como tarea**.

2- En lugar de añadir líneas al fichero dynamic_mappings.txt se podría pensar en que el propio fichero jar generado llevara esta información. Por ejemplo se podría incluir un fichero en formato XML con esa información. O mejor todavía, se podrían definir anotaciones que permitieran definir el *mapping* en el propio código fuente.

3- Se podría proporcionar un objeto común a todas las aplicaciones dinámicas donde colocar servicios que gestiona el servidor. Esos servicios podrían ser: conexiones a bases de datos, conexiones a colas de mensajes, etc. De este modo desde el código cliente obtendríamos una referencia a ese objeto y podríamos solicitar estos servicios.

4- Realizar una aplicación cliente/servidor que permitiera desplegar una aplicación en un servidor remoto.

5- Que el fichero jar además de contener código dinámico también pudiera contener contenido estático (páginas HTML, código JavaScript, etc).