ImplementingTransformers

Implementing transformers

This page describes how to implement your own Cocoon transformer. See WritingPipelineComponents for a general introduction on SAX pipelines.

The diagram below shows the class and interface hierarchy for transformers.

Transformer class diagram

Note that many interfaces shown in this diagram don't specify new methods but just combine other interfaces. As you can see, a Cocoon Transformer is an XMLPipe. An XMLPipe in itself an XMLConsumer and an XMLProducer. An XMLConsumer is basically a SAX ContentHandler and LexicalHandler combined. XMLProducer itself only defines one method, setConsumer, which will be called by Cocoon when it sets up the pipeline. A Transformer is also a SitemapModelComponent. This interface defines one method, setup, which provides the transformer with access to the current runtime environment (sitemap-defined parameters; a so called "object model" containing among others the request and response objects; a sourceresolver for resolving URI's; and the value specified in the 'src' attribute in the sitemap).

Since the ContentHandler and the LexicalHandler interfaces contain quite some methods, there is an AbstractTransformer which has already implemented a default behaviour for all these methods. The default behaviour is to simply pass on the received SAX-events to the consumer, thus effectively implementing an identity transformation. The only method which is really unimplemented is the setup method.

Suppose now that we would like to create a transformer which changes all elements called "foo" to elements called "bar". The code below shows just such a transformer.

```
package yourpackage;

import org.apache.cocoon.transformation.AbstractTransformer;
import org.apache.cocoon.environment.SourceResolver;
import org.apache.cocoon.ProcessingException;
import org.apache.avalon.framework.parameters.Parameters;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;

import java.util.Map;
import java.io.IOException;

public class FooToBarTransformer extends AbstractTransformer
{
    public void setup(SourceResolver resolver, Map objectModel, String src, Parameters par)
            throws ProcessingException, SAXException, IOException
    {
    }

    public void startElement(String namespaceURI, String localName, String qName,
            Attributes attributes) throws SAXException
    {
        if (namespaceURI.equals("") && localName.equals("foo"))
            super.startElement("", "bar", "bar", attributes);
        else
            super.startElement(namespaceURI, localName, qName, attributes);
    }

    public void endElement(String namespaceURI, String localName, String qName)
            throws SAXException
    {
        if (namespaceURI.equals("") && localName.equals("foo"))
            super.endElement("", "bar", "bar");
        else
            super.endElement(namespaceURI, localName, qName);
    }
}
```

Make sure you understand the code above. Note that it is recommended to always write your transformers in a namespace-safe manner. In this example above only elements which are in no namespace will be changed, elements called "foo" in any other namespace will stay untouched.

To try out the transformer, you'll need to compile it. For this to succeed, you will need to put the Cocoon-jars in your classpath. The resulting class file should then be placed in a jar file, and this jar file should be placed in the WEB-INF/lib directory of your deployed Cocoon.

To use the new transformer in the sitemap, declare it as follows in the map:transformers section of the sitemap:

```
<map:transformer name="foo2bar" src="yourpackage.FooToBarTransformer"/>
```

and specify a simple pipeline such as:

```
<map:match pattern="foo2bar">
  <map:generate src="foo.xml"/>
  <map:transform type="foo2bar"/>
  <map:serialize type="xml"/>
</map:match>
```

For this to work, you will of course need to create a foo.xml file and place some foo-tags in it. And now you can try it out.

Further details on transformers

Each transformer is actually an Avalon component. Avalon is a component framework from the Apache Jakarta project, on which Cocoon is heavely based. Basically, it boils down to it that all components, and thus all transformers, are managed by a container (the "component manager"). By implementing certain interfaces, a component can specify how it should be handled by the container, and can get access to certain external resources such as a logger or other components (datasources, ...). If you're really serious about developing Cocoon-components, you should learn these Avalon-interfaces (contracts), which is described here.

One important thing to note is that if you extend your transformer from AbstractTransformer, it also implements the Recyclable interface, which in itself extends the Poolable interface. When a component implements this interface, the instances of this component will be pooled and reused. The important thing about this is that you will most probably need to reset the values of the instance variables of the transformer object between usages (if any). This is usualy done in the setup method.

One question I have is why is a simple transformer **so** much faster than a simple xslt script? I notice that the identity transform in xslt (Xalan) takes about 10 times longer than the same transformer in Java. Shouldn't it be I/O bound? Has anyone else noticed this?

StephenNg

Good question. The reason is simple: an XSLT transformer (at least Xalan and Saxon) will **always** build a complete data structure containing the contents of the incoming SAX-events. This not only consumes a lot of memory, but also takes a lot of time. This also happens when using Xalan in streaming mode (which only advantage is that the output could get faster down the pipeline, but is slower in the end since it creates an additional thread). So it is a good practice to always keep the number of XSLT transforms in a pipeline as low as possible, especially in situations where caching is not an option. – BrunoDumon

XSLT also performs a lot of XPath expression evaluations which require traversing the input document tree, which adds some CPU cost. Simple transformers like the one above do their job "on the fly", but cannot restructure the document like XSLT can. An alternative to manual coding and XSLT could be STX, *Streaming Transformations for XML* that is maturing quickly and its Joost Java implementation (OpenSource, MPL). – SylvainWallez

For those who were looking (I know I was) for information about writing a DOM-based transformer, here are some quick notes to get started: The class to extend is AbstractDOMTransformer. A DOM document will be built for you and the notify() function will be invoked upon completion. Afterward, the transform() function will be invoked. This function is the location to begin implementing your transformer. The DOM document built from the SAX data is passed in as an argument, and the final DOM document to be passed down the pipeline should be returned. Any corrections or "best practice" comments would be appreciated! – WillDrewry