InterpretedSitemapInternals

This document explains the internals of the intepreted sitemap. This is of real interest only to very advanced core developers, and absolutely not needed to use Cocoon in an efficient way.

The reasons for a new engine

SylvainWallez started the TreeProcessor for two reasons.

The first reason was that the sitemap engine at that time was compiled into a Java class like XSP. But the sitemap logicsheet was very complex and recompiling a large sitemap took ages (more than 20 seconds on the samples sitemap), leading to painful try/fail cycles. We needed something faster.

The second reason was that at that time (autumn 2001), a number of RTs were written related to what we called "flowmaps" and later led to flowscript. These RTs were describing new ways to build a pipeline to take flow into account, but no real code was written to test these ideas, because deeply changing the way the sitemap code was generated was very painful: finding its way into the 2000-lines XSLT was not easy.

So I decided to consider another approach, based on an evaluation tree (hence TreeProcessor), each node in the tree corresponding to a xxxmap instruction (sitemap or flowmap).

An additional motivation for me was that it would require me to heavily use the Avalon concepts and therefore increase my knowledge in this area. This was mostly written at home, and my wife deserves many thanks, because this thing took my brain day and night for more than 2 months 😉

Major principles

The main idea of the TreeProcessor is that each kind of instruction (e.g. <map:act>, <map:generate>, etc) is described by two classes :

- a ProcessingNode, the runtime object that will execute the instruction,
- a ProcessingNodeBuilder, responsible for creating the ProcessingNode with the appropriate data and/or childnodes, extracted from attributes, child elements, etc.

Implementing the sitemap language then translates into writing the appropriate ProcessingNodeBuilder classes for all statements of the language. But since we were discussing flowmaps and other pipeline construction approaches, I wanted this to be easily extensible, and even allow the simultaneous use of different languages in the system (sitemap/flowmap). This is why <map:mount> supports an additional undocumented and never used "language" attribute (see MountNodeBuilder)

So the TreeProcessor configuration contains the definition of TreeBuilder implementations for various "languages", the sitemap being the only one we have today. The whole configuration document is actually a ComponentSelector for TreeBuilder implementations. The SitemapLanguage class is the implementation of TreeBuilder for the sitemap language. A TreeBuilder builds a processing node tree based on a file (e.g. sitemap.xmap) that is read in an Avalon configuration (this was chosen for its ease of use compared to raw DOM).

Roles, selectors and <map:components>

The <map:components> section of a sitemap is used to configure a ComponentManager (child of either the parent sitemap's manager or the main manager), and the <roles> section of the TreeProcessor configuration defines a RoleSelector that is used by this manager. For the sitemap, it defines the shorthands that will map <map:generators>, <map:selectors>, etc., to a special "ComponentsSelector" (yeah, the name could be better).

This ComponentsSelector handles the <map:components> syntax ("src" and not "class", etc.), and holds the "default" attribute, view labels and mime types for each hint (these are not know by the components themselves).

Building the processing tree

The second section in a language configuration, <nodes>, defines a ComponentSelector for ProcessingNodeBuilders. For each element encountered in the sitemap source file, the corresponding node builder is fetched from this selector with the local name of the element as the selection hint, i.e. <map:act> will lead to selector.select("act").

The contents of each <node> element is the specific Avalon configuration of the corresponding ProcessingNodeBuilder and mostly define the allowed child statements.

Now a sitemap is not a tree, but a graph because of resources and views that can be called from any point in the sitemap. To handle this, building the processing tree follows two phases:

- the whole node tree is built, and nodes that other nodes can link (or jump) to are registered in the common TreeBuilder by their respective node builders (see TreeBuilder.registerNode()).
- then those node builders that implement LikedProcessingNodeBuilder are asked link their node, which they do by fetching the appropriate node registered in the first phase.

We then obtain an evaluation tree (in reality a graph) that is ready for use. All build-time related components are then released.

It is to be noted also, that a ProcessingNode is considered as a "non-managed component": with the help of the LifecycleHelper class, the TreeBuilder honours any of the Avalon lifecycle interfaces that a node implements. This is required as many nodes require access to the component selectors defined by <map:components>. Disposable nodes are collected in a list that the TreeProcessor traverses when needed (sitemap change or system disposal).

Great care has been taken to cleanly separate build-time and run-time code and data, to ensure the smallest memory occupation and the fastest possible execution. This led this intepreted engine to be a bit faster at runtime than the compiled one (build time is more than 20 times faster).

Building a pipeline

When a request has to be processed, the TreeProcessor calls invoke() on the root node of the evaluation tree. This method has two parameters: the environment defining the request, and an InvokeContext that mainly holds the pipeline that is being built and the stack of sitemap variables.

The invoke method executes all processing nodes (depth first) until one of them returns "true", meaning that a pipeline was successfully built. Examples of nodes that return true are serializers, readers and redirect.

If the environment is external, the pipeline is executed as soon as it is ended (i.e. in the reader or serializer node). But if the environment is internal (i.e. a "cocoon:" source), it is not, meaning the pipeline is returned to the SitemapSource, ready for later execution if requested so (e.g. by a Source. getInputStream()).