Custom Database Migration

Custom migration is useful when your types have changed in ways that make automatic migration difficult or impossible. It is also convenient to use custom migration when you have complex initialization requirements for new types or new data members, because custom migration enables you to perform many of the same tasks that would otherwise require you to write a throwaway program.

Custom migration operates in conjunction with automatic migration, allowing you to inject your own transformation rules at well-defined intercept points in the automatic migration process. These rules are called *transformation descriptors*, and are written in XML.

On this page:

Simple Example of Custom Migration

We can use a simple example to demonstrate the utility of custom migration. Suppose our application uses a Freeze map whose key type is `string` and whose value type is an enumeration, defined as follows:

Slice

```
enum BigThree { Ford, DaimlerChrysler, GeneralMotors };
```

We now wish to rename the enumerator `DaimlerChrysler`, as shown in our new definition:

Slice

```
enum BigThree { Ford, Daimler, GeneralMotors };
```

According to the rules for default transformations, all occurrences of the `DaimlerChrysler` enumerator would be transformed into `Ford`, because `Chrysler` no longer exists in the new definition and therefore the default value `Ford` is used instead.

To remedy this situation, we use the following transformation descriptors:

XML

```
<transformdb>
    <database key="string" value="::BigThree">
        <record>
            <if test="oldvalue == ::Old::DaimlerChrysler">
                <set target="newvalue" value="::New::Daimler"/>
            </if>
        </record>
    </database>
</transformdb>
```

When executed, these descriptors convert occurrences of `DaimlerChrysler` in the old type system into `Daimler` in the transformed database's new type system.

Overview of Transformation Descriptors

As we saw in the previous example, FreezeScript transformation descriptors are written in XML.

A transformation descriptor file has a well-defined structure. The top-level descriptor in the file is `<transformdb>`. A `<database>` descriptor must be present within `<transformdb>` to define the key and value types used by the database. Inside `<database>`, the `<record>` descriptor triggers the transformation process.

Copyright © 2017, ZeroC, Inc.

During transformation, type-specific actions are supported by the <transform> and <init> descriptors, both of which are children of <transformdb>. One <transform> descriptor and one <init> descriptor may be defined for each type in the new Slice definitions. Each time transformdb creates a new instance of a type, it executes the <init> descriptor for that type, if one is defined. Similarly, each time transformdb transforms an instance of an old type into a new type, the <transform> descriptor for the new type is executed.

The <database>, <record>, <transform>, and <init> descriptors may contain general-purpose action descriptors such as <if>, <set>, and <echo>. These actions resemble statements in programming languages like C++ and Java, in that they are executed in the order of definition and their effects are cumulative. Actions can make use of the expression language that should look familiar to C++ and Java programmers.

Transformation Flow of Execution

The transformation descriptors are executed as follows:

- <database> is executed first. Each child descriptor of <database> is executed in the order of definition. If a <record> descriptor is present, database transformation occurs at that point. Any child descriptors of <database> that follow <record> are not executed until transformation completes.
- During transformation of each record, transformdb creates instances of the new key and value types, which includes the execution of the <init> descriptors for those types. Next, the old key and value are transformed into the new key and value, in the following manner:
    1. Locate the <transform> descriptor for the type.
    2. If no descriptor is found, or the descriptor exists and it does not preclude default transformation, then transform the data as in automatic database migration.
    3. If the <transform> descriptor exists, execute it.
    4. Finally, execute the child descriptors of <record>.

Transformation Descriptor Scopes

The <database> descriptor creates a global scope, allowing child descriptors of <database> to define symbols that are accessible in any descriptor.

> In order for a global symbol to be available to a <transform> or <init> descriptor, the symbol must be defined before the <record> descriptor is executed.

Furthermore, certain other descriptors create local scopes that exist only for the duration of the descriptor's execution. For example, the <transform> descriptor creates a local scope and defines the symbols old and new to represent a value in its old and new forms. Child descriptors of <transform> can also define new symbols in the local scope, as long as those symbols do not clash with an existing symbol in that scope. It is legal to add a new symbol with the same name as a symbol in an outer scope, but the outer symbol will not be accessible during the descriptor's execution.

The global scope is useful in many situations. For example, suppose you want to track the number of times a certain value was encountered during transformation. This can be accomplished as shown below:

XML

```
<transformdb>
    <database key="string" value="::Ice::Identity">
        <define name="categoryCount" type="int" value="0"/>
        <record/>
        <echo message="categoryCount = " value="categoryCount"/>
    </database>
    <transform type="::Ice::Identity">
        <if test="new.category == 'Accounting'">
            <set target="categoryCount" value="categoryCount + 1"/>
        </if>
    </transform>
</transformdb>
```

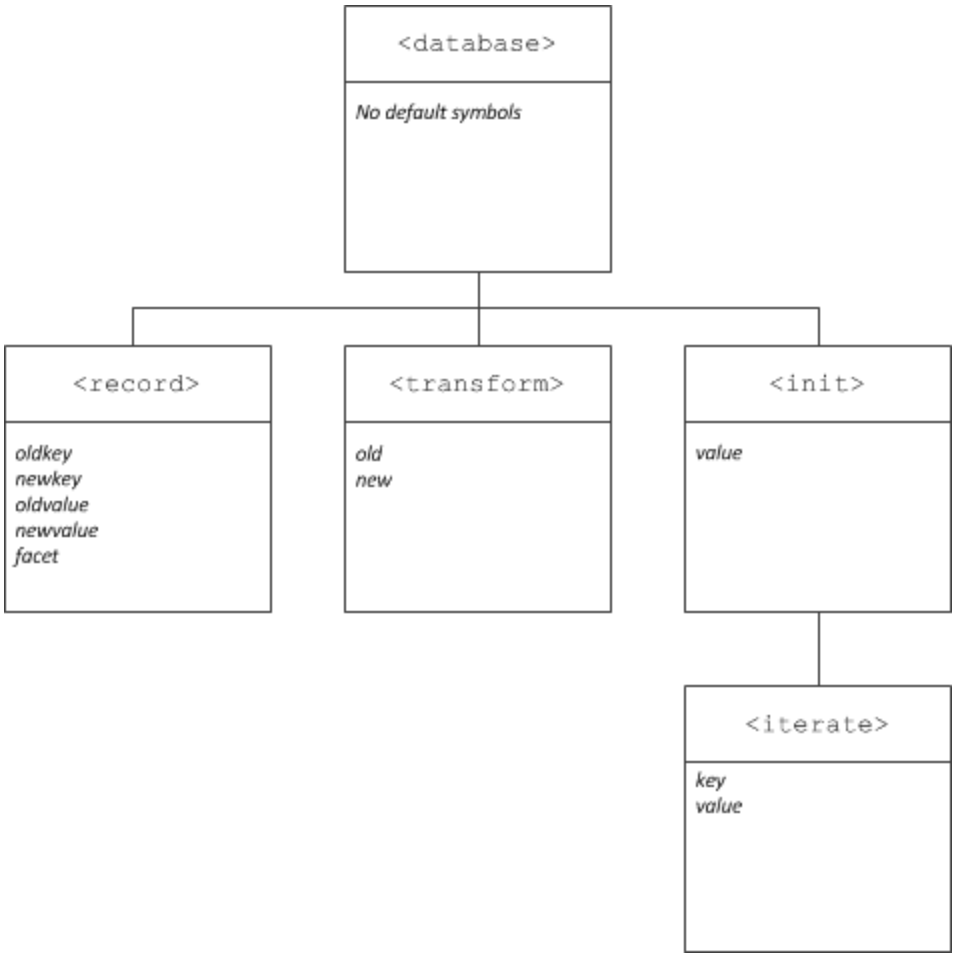
In this example, the <define> descriptor introduces the symbol categoryCount into the global scope, defining it as type int with an initial value of zero. Next, the <record> descriptor causes transformation to proceed. Each occurrence of the type Ice::Identity causes its <transform> descriptor to be executed, which examines the category member and increases categoryCount if necessary. Finally, after transformation completes, the <echo> descriptor displays the final value of categoryCount.

To reinforce the relationships between descriptors and scopes, consider the following diagram. Several descriptors are shown, including the symbols they define in their local scopes. In this example, the <iterate> descriptor has a dictionary target and therefore the default symbol for the element value, value, hides the symbol of the same name in the parent <init> descriptor's scope.

Copyright © 2017, ZeroC, Inc.

This situation can be avoided by assigning a different symbol name to the element value.

In addition to symbols in the <iterate> scope, child descriptors of <iterate> can also refer to symbols from the <init> and <database> scopes.



Relationship between descriptors and scopes.

Guidelines for Transformation Descriptors

There are three points at which you can intercept the transformation process: when transforming a record (<record>), when transforming an instance of a type (<transform>), and when creating an instance of a type (<init>).

In general, <record> is used when your modifications require access to both the key and value of the record. For example, if the database key is needed as a factor in an equation, or to identify an element in a dictionary, then <record> is the only descriptor in which this type of modification is possible. The <record> descriptor is also convenient to use when the number of changes to be made is small, and does not warrant the effort of writing separate <transform> or <init> descriptors.

The <transform> descriptor has a more limited scope than <record>. It is used when changes must potentially be made to all instances of a type (regardless of the context in which that type is used) and access to the old value is necessary. The <transform> descriptor does not have access to the database key and value, therefore decisions can only be made based on the old and new instances of the type in question.

Finally, the <init> descriptor is useful when access to the old instance is not required in order to properly initialize a type. In most cases, this activity could also be performed by a <transform> descriptor that simply ignored the old instance, so <init> may seem redundant. However, there is one situation where <init> is required: when it is necessary to initialize an instance of a type that is introduced by the new Slice definitions. Since there are no instances of this type in the current database, a <transform> descriptor for that type would never be executed.

See Also

Copyright © 2017, ZeroC, Inc.

Copyright © 2017, ZeroC, Inc.