Using dumpdb

This page describes dumpdb and provides advice on how to best use it.

On this page:

Overview of Inspection Descriptors

dumpdb can read descriptors from an XML file. A dumpdb descriptor file has a well-defined structure. The top-level descriptor in the file is <dumpdb>. A <database> descriptor must be present within <dumpdb> to define the key and value types used by the database. Inside <database>, the <record> descriptor triggers database traversal. Shown below is an example that demonstrates the structure of a minimal descriptor file:

XML

```
<dumpdb>
    <database key="string" value="::Employee">
        <record>
            <echo message="Key: " value="key"/>
            <echo message="Value: " value="value"/>
        </record>
    </database>
</dumpdb>
```

During traversal, type-specific actions are supported by the <dump> descriptor, which is a child of <dumpdb>. One <dump> descriptor may be defined for each type in the Slice definitions. Each time dumpdb encounters an instance of a type, the <dump> descriptor for that type is executed.

The <database>, <record>, and <dump> descriptors may contain general-purpose action descriptors such as <if> 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 FreezeScript expression language.

Although dumpdb descriptors are not allowed to modify the database, they can still define local symbols for scripting purposes. Once a symbol is defined by the <define> descriptor, other descriptors such as <set>, <add>, and <remove> can be used to manipulate the symbol's value.

Inspection Flow of Execution

The 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 traversal occurs at that point. Any child descriptors of <database> that follow <record> are not executed until traversal completes.
- For each record, dumpdb interprets the key and value, invoking <dump> descriptors for each type it encounters. For example, if the value type of the database is a struct, then dumpdb first attempts to invoke a <dump> descriptor for the structure type, and then recursively interprets the structure's members in the same fashion.

Inspection 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 <dump> 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 <dump> descriptor creates a local scope and defines the symbol `value` to represent a value of the specified type. Child descriptors of <dump> 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 database traversal. This can be accomplished as shown below:

XML

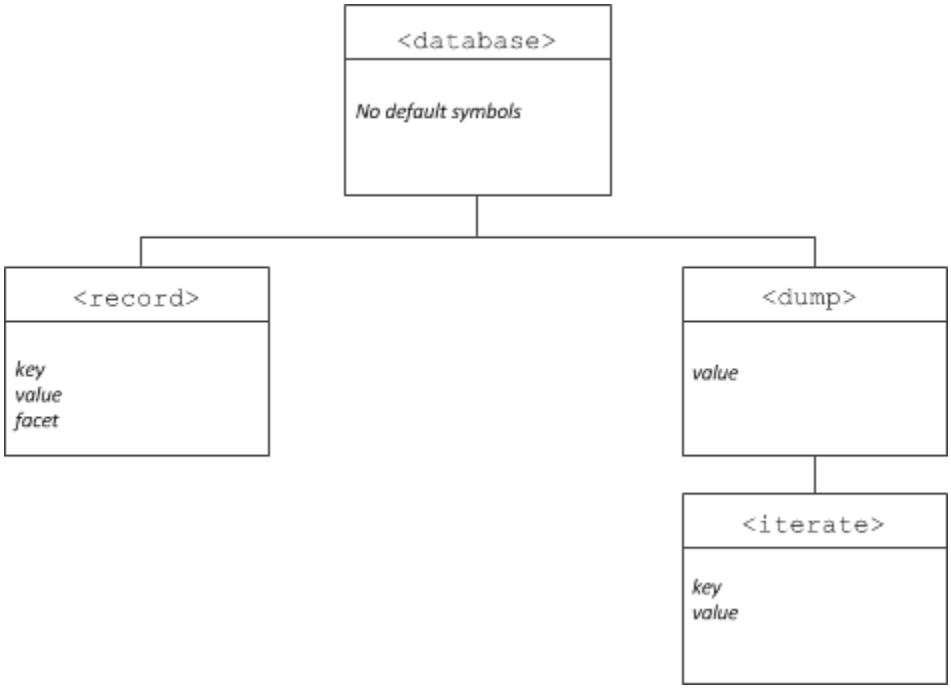
```
<dumpdb>
    <database key="string" value="::Ice::Identity">
        <define name="categoryCount" type="int" value="0"/>
        <record/>
        <echo message="categoryCount = " value="categoryCount"/>
    </database>
    <dump type="::Ice::Identity">
        <if test="value.category == `Accounting'">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

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 traversal to proceed. Each occurrence of the type `Ice::Identity` causes its <dump> descriptor to be executed, which examines the `category` member and increases `categoryCount` if necessary. Finally, after traversal completes, the <echo> descriptor displays the final value of `categoryCount`.

To reinforce the relationships between descriptors and scopes, consider the diagram in the figure below. 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 <dump> descriptor's scope.

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 <dump> and <database> scopes.



Relationship between descriptors and scopes.

Command Line Options for `dumpdb`

The tool supports the standard command-line options common to all Slice processors listed. The options specific to `dumpdb` are described below:

- `--load` ***`SLICE`***
  Loads the Slice definitions contained in the file ***`SLICE`***. This option may be specified multiple times if several files must be loaded. However, it is the user's responsibility to ensure that duplicate definitions do not occur (which is possible when two files are loaded that share a common include file). One strategy for avoiding duplicate definitions is to load a single Slice file that contains only `#include` statements for each of the Slice files to be loaded. No duplication is possible in this case if the included files use include guards correctly.

- `--key` ***`TYPE`***
  `--value` ***`TYPE`***
  Specifies the Slice type of the database key and value. If these options are not specified, and the `-e` option is not used, `dumpdb` obtains type information from the Freeze catalog.

- `-e`
  Indicates that a Freeze evictor database is being examined. As a convenience, this option automatically sets the database key and value types to those appropriate for the Freeze evictor, and therefore the `--key` and `--value` options are not necessary. Specifically, the key type of a Freeze evictor database is `Ice::Identity`, and the value type is `Freeze::ObjectRecord`. The latter is defined in the Slice file `Freeze/EvictorStorage.ice`, however this file does not need to be explicitly loaded. If this option is not specified, and the `--key` and `--value` options are not used, `dumpdb` obtains type information from the Freeze catalog.

- `-o` ***`FILE`***
  Create a file named ***`FILE`*** containing sample descriptors for the loaded Slice definitions. If type information is not specified, `dumpdb` obtains it from the Freeze catalog. If the `--select` option is used, its expression is included in the sample descriptors. Database traversal does not occur when the `-o` option is used.

- `-f` ***`FILE`***
  Execute the descriptors in the file named ***`FILE`***. The file's `<database>` descriptor specifies the key and value types; therefore it is not necessary to supply type information.

- `--select` ***`EXPR`***
  Only display those records for which the expression ***`EXPR`*** is true. The expression can refer to the symbols `key` and `value`.

- `-c, --catalog`
  Display information about the databases in an environment, or about a particular database. This option presents the type information contained in the Freeze catalog.

Database Arguments for dumpdb

If dumpdb is invoked to examine a database, it requires two arguments:

- dbenv
  The pathname of the database environment directory.
- db
  The name of the database file. dumpdb opens this database as read-only, and traversal occurs within a transaction.

To display catalog information using the -c option, the database environment directory dbenv is required. If the database file argument db is omitted, dumpdb displays information about every database in the catalog.

dumpdb Use Cases

The command line options support several modes of operation:

- Dump an entire database.
- Dump selected records of a database.
- Emit a sample descriptor file.
- Execute a descriptor file.
- Examine the catalog.

These use cases are described in the following sections.

Dump an Entire Database

The simplest way to examine a database with dumpdb is to dump its entire contents. You must specify the database key and value types, load the necessary Slice definitions, and supply the names of the database environment directory and database file. For example, this command dumps a Freeze map database whose key type is string and value type is Employee:

```
$ dumpdb --key string --value ::Employee --load Employee.ice db emp.db
```

As a convenience, you may omit the key and value types, in which case dumpdb obtains them from the catalog:

```
$ dumpdb --load Employee.ice db emp.db
```

Dump Selected Records

If only certain records are of interest to you, the --select option provides a convenient way to filter the output of dumpdb using an expression. In the following example, we select employees from the accounting department:

```
$ dumpdb --load Employee.ice --select "value.dept == 'Accounting'" db emp.db
```

In cases where the database records contain polymorphic class instances, you must be careful to specify an expression that can be successfully evaluated against all records. For example, dumpdb fails immediately if the expression refers to a data member that does not exist in the class instance. The safest way to write an expression in this case is to check the type of the class instance before referring to any of its data members.

In the example below, we assume that a Freeze evictor database contains instances of various classes in a class hierarchy, and we are only interested in instances of Manager whose employee count is greater than 10:

```
$ dumpdb -e --load Employee.ice \
--select "value.servant.ice_id == '::Manager' and value.servant.group.length > 10" \
db emp.db
```

Alternatively, if Manager has derived classes, then the expression can be written in a different way so that instances of Manager and any of its derived classes are considered:

```
$ dumpdb -e --load Employee.ice \
--select "value.servant.ice_isA('::Manager') and value.servant.group.length > 10" \
db emp.db
```

Creating a Sample Descriptor File

If you require more sophisticated filtering or scripting capabilities, then you must use a descriptor file. The easiest way to get started with a descriptor file is to generate a template using `dumpdb`:

```
$ dumpdb --key string --value ::Employee --load Employee.ice -o dump.xml
```

The output file `dump.xml` is complete and can be executed immediately if desired, but typically the file is used as a starting point for further customization. Again, you may omit the key and value types by specifying the database instead:

```
$ dumpdb --load Employee.ice -o dump.xml db emp.db
```

If the `--select` option is specified, its expression is included in the generated `<record>` descriptor as the value of the `test` attribute in an `<if>` descriptor.

`dumpdb` terminates immediately after generating the output file.

Executing a Descriptor File

Use the `-f` option when you are ready to execute a descriptor file. For example, we can execute the descriptor we generated in the previous section using this command:

```
$ dumpdb -f dump.xml --load Employee.ice db emp.db
```

Examine the Catalog

The `-c` option displays the contents of the database environment's catalog:

```
$ dumpdb -c db
```

The output indicates whether each database in the environment is associated with an evictor or a map. For maps, the output includes the key and value types.

If you specify the name of a database, `dumpdb` only displays the type information for that database:

```
$ dumpdb -c db emp.db
```

Using `dumpdb` on an Open Environment

It is possible to use `dumpdb` to migrate databases in an environment that is currently open by another process, but if you are not careful you can easily corrupt the environment and cause the other process to fail. To avoid such problems, you must configure both `dumpdb` and the other process to set `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`. This property has a default value of one, therefore you must explicitly set it to zero.

If you run `dumpdb` on an open environment but neglect to set `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`, you can expect `dumpdb` to terminate immediately with an error message stating that the database environment is locked. Before running `dumpdb` on an open environment, we strongly recommend that you first verify that the other process was also configured with `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`.

See Also

- Using the Slice Compilers
- Freeze Catalogs