# Code Generation in Ruby

The Ruby mapping supports two forms of code generation: dynamic and static.

On this page:

## Dynamic Code Generation in Ruby

Using dynamic code generation, Slice files are "loaded" at run time and dynamically translated into Ruby code, which is immediately compiled and available for use by the application. This is accomplished using the `Ice::loadSlice` method, as shown in the following example:

**Ruby**

```
Ice::loadSlice("Color.ice")
puts "My favorite color is #{M::Color.blue.to_s}"
```

For this example, we assume that `Color.ice` contains the following definitions:

**Slice**

```
module M {
    enum Color { red, green, blue };
};
```

### `Ice::loadSlice` Options in Ruby

The `Ice::loadSlice` method behaves like a Slice compiler in that it accepts command-line arguments for specifying preprocessor options and controlling code generation. The arguments must include at least one Slice file.

The function has the following Ruby definition:

**Ruby**

```
def loadSlice(cmd, args=[])
```

The command-line arguments can be specified entirely in the first argument, `cmd`, which must be a string. The optional second argument can be used to pass additional command-line arguments as a list; this is useful when the caller already has the arguments in list form. The function always returns `nil`.

For example, the following calls to `Ice::loadSlice` are functionally equivalent:

**Ruby**

```
Ice::loadSlice("-I/opt/IceRuby/slice Color.ice")
Ice::loadSlice("-I/opt/IceRuby/slice", ["Color.ice"])
Ice::loadSlice("", ["-I/opt/IceRuby/slice", "Color.ice"])
```

In addition to the standard compiler options, `Ice::loadSlice` also supports the following command-line options:

- `--all`
  Generate code for all Slice definitions, including those from included files.
- `--checksum`
  Generate checksums for Slice definitions.

## Locating Slice Files in Ruby

If your Slice files depend on Ice types, you can avoid hard-coding the path name of your Ice installation directory by calling the `Ice::getSliceDir` function:

**Ruby**

```
Ice::loadSlice("-I" + Ice::getSliceDir() + " Color.ice")
```

This function attempts to locate the `slice` subdirectory of your Ice installation using an algorithm that succeeds for the following scenarios:

- Installation of a binary Ice archive
- Installation of an Ice source distribution using `make install`
- Installation via a Windows installer
- RPM installation on Linux
- Execution inside a compiled Ice source distribution

If the `slice` subdirectory can be found, `getSliceDir` returns its absolute path name, otherwise the function returns `nil`.

## Loading Multiple Slice Files in Ruby

You can specify as many Slice files as necessary in a single invocation of `Ice::loadSlice`, as shown below:

**Ruby**

```
Ice::loadSlice("Syscall.ice Process.ice")
```

Alternatively, you can call `Ice::loadSlice` several times:

**Ruby**

```
Ice::loadSlice("Syscall.ice")
Ice::loadSlice("Process.ice")
```

If a Slice file includes another file, the default behavior of `Ice::loadSlice` generates Ruby code only for the named file. For example, suppose `Syscall.ice` includes `Process.ice` as follows:

**Slice**

```
// Syscall.ice
#include <Process.ice>
...
```

If you call `Ice::loadSlice("-I. Syscall.ice")`, Ruby code is not generated for the Slice definitions in `Process.ice` or for any definitions that may be included by `Process.ice`. If you also need code to be generated for included files, one solution is to load them individually in subsequent calls to `Ice::loadSlice`. However, it is much simpler, not to mention more efficient, to use the `--all` option instead:

**Ruby**

```
Ice::loadSlice("--all -I. Syscall.ice")
```

When you specify `--all`, `Ice::loadSlice` generates Ruby code for all Slice definitions included directly or indirectly from the named Slice files.

There is no harm in loading a Slice file multiple times, aside from the additional overhead associated with code generation. For example, this situation could arise when you need to load multiple top-level Slice files that happen to include a common subset of nested files. Suppose that we need to load both `Syscall.ice` and `Kernel.ice`, both of which include `Process.ice`. The simplest way to load both files is with a single call to `Ice::loadSlice`:

**Ruby**

```
Ice::loadSlice("--all -I. Syscall.ice Kernel.ice")
```

Although this invocation causes the Ice extension to generate code twice for `Process.ice`, the generated code is structured so that the interpreter ignores duplicate definitions. We could have avoided generating unnecessary code with the following sequence of steps:

**Ruby**

```
Ice::loadSlice("--all -I. Syscall.ice")
Ice::loadSlice("-I. Kernel.ice")
```

In more complex cases, however, it can be difficult or impossible to completely avoid this situation, and the overhead of code generation is usually not significant enough to justify such an effort.

### Limitations of Dynamic Code Generation in Ruby

The `Ice::loadSlice` method must be called outside of any module scope. For example, the following code is incorrect:

**Ruby**

```
# WRONG
module M
    Ice::loadSlice("--all -I. Syscall.ice Kernel.ice")
    ...
end
```

## Static Code Generation in Ruby

You should be familiar with static code generation if you have used other Slice language mappings, such as C++ or Java. Using static code generation, the Slice compiler `slice2rb` generates Ruby code from your Slice definitions.

### Compiler Output in Ruby

For each Slice file `X.ice`, `slice2rb` generates Ruby code into a file named `X.rb` in the output directory. The default output directory is the current working directory, but a different directory can be specified using the `--output-dir` option.

### Include Files in Ruby

It is important to understand how `slice2rb` handles include files. In the absence of the `--all` option, the compiler does not generate Ruby code for Slice definitions in included files. Rather, the compiler translates Slice `#include` statements into Ruby `require` statements in the following manner:

1. Determine the full pathname of the included file.
2. Create the shortest possible relative pathname for the included file by iterating over each of the include directories (specified using the `-I` option) and removing the leading directory from the included file if possible.
   For example, if the full pathname of an included file is `/opt/App/slice/OS/Process.ice`, and we specified the options `-I/opt/App` and `-I/opt/App/slice`, then the shortest relative pathname is `OS/Process.ice` after removing `/opt/App/slice`.
3. Replace the `.ice` extension with `.rb`. Continuing our example from the previous step, the translated `require` statement becomes

   ```
   require "OS/Process.rb"
   ```

As a result, you can use `-I` options to tailor the `require` statements generated by the compiler in order to avoid absolute pathnames and match the organizational structure of your application's source files.

## Static Versus Dynamic Code Generation in Ruby

There are several issues to consider when evaluating your requirements for code generation.

### Application Considerations for Code Generation in Ruby

The requirements of your application generally dictate whether you should use dynamic or static code generation. Dynamic code generation is convenient for a number of reasons:

- It avoids the intermediate compilation step required by static code generation.
- It makes the application more compact because the application requires only the Slice files, not the additional files produced by static code generation.
- It reduces complexity, which is especially helpful during testing, or when writing short or transient programs.

Static code generation, on the other hand, is appropriate in many situations:

- when an application uses a large number of Slice definitions and the startup delay must be minimized
- when it is not feasible to deploy Slice files with the application
- when a number of applications share the same Slice files
- when Ruby code is required in order to utilize third-party Ruby tools.

### Mixing Static and Dynamic Generation in Ruby

You can safely use a combination of static and dynamic translation in an application. For it to work properly, you must correctly manage the include paths for Slice translation and the Ruby interpreter so that the statically-generated code can be imported properly by `require`.

For example, suppose you want to dynamically load the following Slice definitions:

**Slice**

```
#include <Glacier2/Session.ice>

module MyApp {
    interface MySession extends Glacier2::Session {
        // ...
    };
};
```

Whether the included file `Glacier2/Session.ice` is loaded dynamically or statically is determined by the presence of the `--all` option:

**Ruby**

```
sliceDir = "-I#{ENV['ICE_HOME']}/slice"

# Load Glacier2/Session.ice dynamically:
Ice::loadSlice(sliceDir + " --all MySession.ice")

# Load Glacier2/Session.ice statically:
Ice::loadSlice(sliceDir + " MySession.ice")
```

In this example, the first invocation of `loadSlice` uses the `--all` option so that code is generated dynamically for all included files. The second invocation omits `--all`, therefore the Ruby interpreter executes the equivalent of the following statement:

```
require "Glacier2/Session.rb"
```

As a result, before we can call `loadSlice` we must first ensure that the interpreter can locate the statically-generated file `Glacier2/Session.rb`. We can do this in a number of ways, including:

- adding the parent directory (e.g., `/opt/IceRuby/ruby`) to the `RUBYLIB` environment variable
- specifying the `-I` option when starting the interpreter
- modifying the search path at run time, as shown below:
  ```
  $:.unshift("/opt/IceRuby/ruby")
  ```

## `slice2rb` Command-Line Options

The Slice-to-Ruby compiler, `slice2rb`, offers the following command-line options in addition to the standard options:

- `--all`
  Generate code for all Slice definitions, including those from included files.

- `--checksum`
  Generate checksums for Slice definitions.

### See Also

- Using the Slice Compilers
- Using Slice Checksums in Ruby