Compiler

Compiler 4.7.3
August 11 2011

August 11 2011

1 Reference Manual

The *Compiler* application compiles Erlang code to byte-code. The highly compact byte-code is executed by the Erlang emulator.

compile

Erlang module

This module provides an interface to the standard Erlang compiler. It can generate either a new file which contains the object code, or return a binary which can be loaded directly.

Exports

`file(File)`

Is the same as `file(File, [verbose,report_errors,report_warnings])`.

`file(File, Options) -> CompRet`

Types:

CompRet = ModRet | BinRet | ErrRet

ModRet = {ok,ModuleName} | {ok,ModuleName,Warnings}

BinRet = {ok,ModuleName,Binary} | {ok,ModuleName,Binary,Warnings}

ErrRet = error | {error,Errors,Warnings}

Compiles the code in the file `File`, which is an Erlang source code file without the `.erl` extension. `Options` determine the behavior of the compiler.

Returns `{ok,ModuleName}` if successful, or `error` if there are errors. An object code file is created if the compilation succeeds with no errors. It is considered to be an error if the module name in the source code is not the same as the basename of the output file.

Here follows first all elements of `Options` that in some way control the behavior of the compiler.

`basic_validation`

This option is fast way to test whether a module will compile successfully (mainly useful for code generators that want to verify the code they emit). No code will generated. If warnings are enabled, warnings generated by the `erl_lint` module (such as warnings for unused variables and functions) will be returned too.

Use the `strong_validation` option to generate all warnings that the compiler would generate.

`strong_validation`

Similar to the `basic_validation` option, no code will be generated, but more compiler passes will be run to ensure also warnings generated by the optimization passes are generated (such as clauses that will not match or expressions that are guaranteed to fail with an exception at run-time).

`binary`

Causes the compiler to return the object code in a binary instead of creating an object file. If successful, the compiler returns `{ok,ModuleName,Binary}`.

`bin_opt_info`

The compiler will emit informational warnings about binary matching optimizations (both successful and unsuccessful). See the *Efficiency Guide* for further information.

`compressed`

The compiler will compress the generated object code, which can be useful for embedded systems.

`debug_info`

Include debug information in the form of abstract code (see *The Abstract Format* in ERTS User's Guide) in the compiled beam module. Tools such as Debugger, Xref and Cover require the debug information to be included.

Warning: Source code can be reconstructed from the debug information. Use encrypted debug information (see below) to prevent this.

See *beam_lib(3)* for details.

`{debug_info_key,KeyString}`
`{debug_info_key,{Mode,KeyString}}`

Include debug information, but encrypt it, so that it cannot be accessed without supplying the key. (To give the `debug_info` option as well is allowed, but is not necessary.) Using this option is a good way to always have the debug information available during testing, yet protect the source code.

`Mode` is the type of crypto algorithm to be used for encrypting the debug information. The default type -- and currently the only type -- is `des3_cbc`.

See *beam_lib(3)* for details.

`encrypt_debug_info`

Like the `debug_info_key` option above, except that the key will be read from an `.erlang.crypt` file.

See *beam_lib(3)* for details.

`makedep`

Produce a Makefile rule to track headers dependencies. No object file is produced.

By default, this rule is written to `<File>.Pbeam`. However, if the option `binary` is set, nothing is written and the rule is returned in `Binary`.

For instance, if one has the following module:

```
-module(module).

-include_lib("eunit/include/eunit.hrl").
-include("header.hrl").
```

Here is the Makefile rule generated by this option:

```
module.beam: module.erl \
  /usr/local/lib/erlang/lib/eunit/include/eunit.hrl \
  header.hrl
```

`{makedep_output, Output}`

Write generated rule(s) to `Output` instead of the default `<File>.Pbeam`. `Output` can be a filename or an `io_device()`. To write to stdout, use `standard_io`. However if `binary` is set, nothing is written to `Output` and the result is returned to the caller with `{ok, ModuleName, Binary}`.

`{makedep_target, Target}`

Change the name of the rule emitted to `Target`.

`makedep_quote_target`

Characters in `Target` special to make(1) are quoted.

`makedep_add_missing`

Consider missing headers as generated files and add them to the dependencies.

`makedep_phony`

Add a phony target for each dependency.

`'P'`

Produces a listing of the parsed code after preprocessing and parse transforms, in the file `<File>.P`. No object file is produced.

`'E'`

Produces a listing of the code after all source code transformations have been performed, in the file `<File>.E`. No object file is produced.

`'S'`

Produces a listing of the assembler code in the file `<File>.S`. No object file is produced.

`report_errors/report_warnings`

Causes errors/warnings to be printed as they occur.

`report`

This is a short form for both `report_errors` and `report_warnings`.

`return_errors`

If this flag is set, then `{error,ErrorList,WarningList}` is returned when there are errors.

`return_warnings`

If this flag is set, then an extra field containing `WarningList` is added to the tuples returned on success.

`warnings_as_errors`

Causes warnings to be treated as errors. This option is supported since R13B04.

`return`

This is a short form for both `return_errors` and `return_warnings`.

`verbose`

Causes more verbose information from the compiler describing what it is doing.

`{outdir,Dir}`

Sets a new directory for the object code. The current directory is used for output, except when a directory has been specified with this option.

`export_all`

Causes all functions in the module to be exported.

`{i,Dir}`

Add `Dir` to the list of directories to be searched when including a file. When encountering an `-include` or `-include_dir` directive, the compiler searches for header files in the following directories:

- `"."`, the current working directory of the file server;
- the base name of the compiled file;

- the directories specified using the `i` option. The directory specified last is searched first.

`{d,Macro}`
`{d,Macro,Value}`

Defines a macro `Macro` to have the value `Value`. The default is `true`).

`{parse_transform,Module}`

Causes the parse transformation function `Module:parse_transform/2` to be applied to the parsed code before the code is checked for errors.

`asm`

The input file is expected to be assembler code (default file suffix ".S"). Note that the format of assembler files is not documented, and may change between releases - this option is primarily for internal debugging use.

`no_strict_record_tests`

This option is not recommended.

By default, the generated code for the `Record#record_tag.field` operation verifies that the tuple `Record` is of the correct size for the record and that the first element is the tag `record_tag`. Use this option to omit the verification code.

`no_error_module_mismatch`

Normally the compiler verifies that the module name given in the source code is the same as the base name of the output file and refuses to generate an output file if there is a mismatch. If you have a good reason (or other reason) for having a module name unrelated to the name of the output file, this option disables that verification (there will not even be a warning if there is a mismatch).

`{no_auto_import,[{F,A}, ...]}`

Makes the function `F/A` no longer being auto-imported from the module `erlang`, which resolves BIF name clashes. This option has to be used to resolve name clashes with BIFs auto-imported before R14A, if one wants to call the local function with the same name as an auto-imported BIF without module prefix.

> **Note:**
>
> From R14A and forward, the compiler resolves calls without module prefix to local or imported functions before trying auto-imported BIFs. If the BIF is to be called, use the `erlang` module prefix in the call, not `{ no_auto_import,[{F,A}, ...]}`

If this option is written in the source code, as a `-compile` directive, the syntax `F/A` can be used instead of `{F,A}`. Example:

```
-compile({no_auto_import,[error/1]}).
```

If warnings are turned on (the `report_warnings` option described above), the following options control what type of warnings that will be generated. With the exception of `{warn_format,Verbosity}` all options below have two forms; one `warn_xxx` form to turn on the warning and one `nowarn_xxx` form to turn off the warning. In the description that follows, the form that is used to change the default value is listed.

`{warn_format, Verbosity}`

Causes warnings to be emitted for malformed format strings as arguments to `io:format` and similar functions. `Verbosity` selects the amount of warnings: 0 = no warnings; 1 = warnings for invalid format strings and incorrect number of arguments; 2 = warnings also when the validity could not be checked (for example, when the

format string argument is a variable). The default verbosity is 1. Verbosity 0 can also be selected by the option `nowarn_format`.

`nowarn_bif_clash`

This option is removed, it will generate a fatal error if used.

> ### Warning:
>
> Beginning with R14A, the compiler no longer calls the auto-imported BIF if the name clashes with a local or explicitly imported function and a call without explicit module name is issued. Instead the local or imported function is called. Still accepting `nowarn_bif_clash` would makes a module calling functions clashing with autoimported BIFs compile with both the old and new compilers, but with completely different semantics, why the option was removed.
>
> The use of this option has always been strongly discouraged. From OTP R14A and forward it's an error to use it.
>
> To resolve BIF clashes, use explicit module names or the `{no_auto_import,[F/A]}` compiler directive.

`{nowarn_bif_clash, FAs}`

This option is removed, it will generate a fatal error if used.

> ### Warning:
>
> The use of this option has always been strongly discouraged. From OTP R14A and forward it's an error to use it.
>
> To resolve BIF clashes, use explicit module names or the `{no_auto_import,[F/A]}` compiler directive.

`warn_export_all`

Causes a warning to be emitted if the `export_all` option has also been given.

`warn_export_vars`

Causes warnings to be emitted for all implicitly exported variables referred to after the primitives where they were first defined. No warnings for exported variables unless they are referred to in some pattern, which is the default, can be selected by the option `nowarn_export_vars`.

`warn_shadow_vars`

Causes warnings to be emitted for "fresh" variables in functional objects or list comprehensions with the same name as some already defined variable. The default is to warn for such variables. No warnings for shadowed variables can be selected by the option `nowarn_shadow_vars`.

`nowarn_unused_function`

Turns off warnings for unused local functions. By default (`warn_unused_function`), warnings are emitted for all local functions that are not called directly or indirectly by an exported function. The compiler does not include unused local functions in the generated beam file, but the warning is still useful to keep the source code cleaner.

`{nowarn_unused_function, FAs}`

Turns off warnings for unused local functions as `nowarn_unused_function` but only for the mentioned local functions. `FAs` is a tuple `{Name,Arity}` or a list of such tuples.

`nowarn_deprecated_function`

Turns off warnings for calls to deprecated functions. By default (`warn_deprecated_function`), warnings are emitted for every call to a function known by the compiler to be deprecated. Note that the compiler does not know about the `-deprecated()` attribute but uses an assembled list of deprecated functions in Erlang/OTP. To do a more general check the `Xref` tool can be used. See also *xref(3)* and the function *xref:m/1* also accessible through the *c:xm/1* function.

`{nowarn_deprecated_function, MFAs}`

Turns off warnings for calls to deprecated functions as `nowarn_deprecated_function` but only for the mentioned functions. `MFAs` is a tuple `{Module,Name,Arity}` or a list of such tuples.

`warn_obsolete_guard`

Causes warnings to be emitted for calls to old type testing BIFs such as `pid/1` and `list/1`. See the *Erlang Reference Manual* for a complete list of type testing BIFs and their old equivalents. No warnings for calls to old type testing BIFs, which is the default, can be selected by the option `nowarn_obsolete_guard`.

`warn_unused_import`

Causes warnings to be emitted for unused imported functions. No warnings for unused imported functions, which is the default, can be selected by the option `nowarn_unused_import`.

`nowarn_unused_vars`

By default, warnings are emitted for variables which are not used, with the exception of variables beginning with an underscore ("Prolog style warnings"). Use this option to turn off this kind of warnings.

`nowarn_unused_record`

Turns off warnings for unused record types. By default (`warn_unused_records`), warnings are emitted for unused locally defined record types.

Another class of warnings is generated by the compiler during optimization and code generation. They warn about patterns that will never match (such as `a=b`), guards that will always evaluate to false, and expressions that will always fail (such as `atom+42`).

Note that the compiler does not warn for expressions that it does not attempt to optimize. For instance, the compiler tries to evaluate `1/0`, notices that it will cause an exception and emits a warning. On the other hand, the compiler is silent about the similar expression `X/0`; because of the variable in it, the compiler does not even try to evaluate and therefore it emits no warnings.

Currently, those warnings cannot be disabled (except by disabling all warnings).

> ### Warning:
>
> Obviously, the absence of warnings does not mean that there are no remaining errors in the code.

Note that all the options except the include path (`{i,Dir}`) can also be given in the file with a `-compile([Option,...])`. attribute. The `-compile()` attribute is allowed after function definitions.

Note also that the `{nowarn_unused_function, FAs}`, `{nowarn_bif_clash, FAs}`, and `{nowarn_deprecated_function, MFAs}` options are only recognized when given in files. They are not affected by the `warn_unused_function`, `warn_bif_clash`, or `warn_deprecated_function` options.

For debugging of the compiler, or for pure curiosity, the intermediate code generated by each compiler pass can be inspected. A complete list of the options to produce list files can be printed by typing `compile:options()` at the Erlang shell prompt. The options will be printed in order that the passes are executed. If more than one listing option is used, the one representing the earliest pass takes effect.

Unrecognized options are ignored.

Both `WarningList` and `ErrorList` have the following format:

```
[{FileName,[ErrorInfo]}].
```

`ErrorInfo` is described below. The file name has been included here as the compiler uses the Erlang pre-processor `epp`, which allows the code to be included in other files. For this reason, it is important to know to *which* file an error or warning line number refers.

`forms(Forms)`

Is the same as `forms(File, [verbose,report_errors,report_warnings])`.

`forms(Forms, Options) -> CompRet`

Types:

Forms = [Form]

CompRet = BinRet | ErrRet

BinRet = {ok,ModuleName,BinaryOrCode} | {ok,ModuleName,BinaryOrCode,Warnings}

BinaryOrCode = binary() | term()

ErrRet = error | {error,Errors,Warnings}

Analogous to `file/1`, but takes a list of forms (in the Erlang abstract format representation) as first argument. The option `binary` is implicit; i.e., no object code file is produced. Options that would ordinarily produce a listing file, such as 'E', will instead cause the internal format for that compiler pass (an Erlang term; usually not a binary) to be returned instead of a binary.

`format_error(ErrorDescriptor) -> chars()`

Types:

ErrorDescriptor = errordesc()

Uses an `ErrorDescriptor` and returns a deep list of characters which describes the error. This function is usually called implicitly when an `ErrorInfo` structure is processed. See below.

`output_generated(Options) -> true | false`

Types:

Options = [term()]

Determines whether the compiler would generate a `beam` file with the given options. `true` means that a `beam` file would be generated; `false` means that the compiler would generate some listing file, return a binary, or merely check the syntax of the source code.

`noenv_file(File, Options) -> CompRet`

Works exactly like *file/2*, except that the environment variable `ERL_COMPILER_OPTIONS` is not consulted.

noenv_forms(Forms, Options) -> CompRet

Works exactly like *forms/2*, except that the environment variable `ERL_COMPILER_OPTIONS` is not consulted.

noenv_output_generated(Options) -> true | false

Types:

Options = [term()]

Works exactly like *output_generated/1*, except that the environment variable `ERL_COMPILER_OPTIONS` is not consulted.

Default compiler options

The (host operating system) environment variable `ERL_COMPILER_OPTIONS` can be used to give default compiler options. Its value must be a valid Erlang term. If the value is a list, it will be used as is. If it is not a list, it will be put into a list.

The list will be appended to any options given to *file/2*, *forms/2*, and *output_generated/2*. Use the alternative functions *noenv_file/2*, *noenv_forms/2*, or *noenv_output_generated/2* if you don't want the environment variable to be consulted (for instance, if you are calling the compiler recursively from inside a parse transform).

Inlining

The compiler can do function inlining within an Erlang module. Inlining means that a call to a function is replaced with the function body with the arguments replaced with the actual values. The semantics are preserved, except if exceptions are generated in the inlined code. Exceptions will be reported as occurring in the function the body was inlined into. Also, `function_clause` exceptions will be converted to similar `case_clause` exceptions.

When a function is inlined, the original function will be kept if it is exported (either by an explicit export or if the `export_all` option was given) or if not all calls to the function were inlined.

Inlining does not necessarily improve running time. For instance, inlining may increase Beam stack usage which will probably be detrimental to performance for recursive functions.

Inlining is never default; it must be explicitly enabled with a compiler option or a `-compile()` attribute in the source module.

To enable inlining, either use the `inline` option to let the compiler decide which functions to inline or `{inline, [{Name,Arity},...]}` to have the compiler inline all calls to the given functions. If the option is given inside a `compile` directive in an Erlang module, `{Name,Arity}` may be written as `Name/Arity`.

Example of explicit inlining:

```
-compile({inline,[pi/0]}).

pi() -> 3.1416.
```

Example of implicit inlining:

```
-compile(inline).
```

The `{inline_size,Size}` option controls how large functions that are allowed to be inlined. Default is `24`, which will keep the size of the inlined code roughly the same as the un-inlined version (only relatively small functions will be inlined).

Example:

```
%% Aggressive inlining - will increase code size.
-compile(inline).
-compile({inline_size,100}).
```

Parse Transformations

Parse transformations are used when a programmer wants to use Erlang syntax but with different semantics. The original Erlang code is then transformed into other Erlang code.

Error Information

The `ErrorInfo` mentioned above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the following format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string describing the error is obtained with the following call:

```
Module:format_error(ErrorDescriptor)
```

See Also

epp(3), erl_id_trans(3), erl_lint(3), beam_lib(3)