NAME

code - Erlang code server.

DESCRIPTION

This module contains the interface to the Erlang *code server*, which deals with the loading of compiled code into a running Erlang runtime system.

The runtime system can be started in *interactive* or *embedded* mode. Which one is decided by the command-line flag *-mode*:

```
% erl -mode interactive
```

The modes are as follows:

* In interactive mode, which is default, only some code is loaded during system startup, basically the modules needed by the runtime system. Other code is dynamically loaded when first referenced. When a call to a function in a certain module is made, and the module is not loaded, the code server searches for and tries to load the module.

* In embedded mode, modules are not auto loaded. Trying to use a module that has not been loaded results in an error. This mode is recommended when the boot script loads all modules, as it is typically done in OTP releases. (Code can still be loaded later by explicitly ordering the code server to do so).

To prevent accidentally reloading of modules affecting the Erlang runtime system, directories *kernel*, *stdlib*, and *compiler* are considered *sticky*. This means that the system issues a warning and rejects the request if a user tries to reload a module residing in any of them. The feature can be disabled by using command-line flag *-nostick*.

CODE PATH

In interactive mode, the code server maintains a search path, usually called the *code path*, consisting of a list of directories, which it searches sequentially when trying to load a module.

Initially, the code path consists of the current working directory and all Erlang object code directories under library directory *$OTPROOT/lib*, where *$OTPROOT* is the installation directory of Erlang/OTP, *code:root_dir()*. Directories can be named *Name[-Vsn]* and the code server, by default, chooses the directory with the highest version number among those having the same *Name*. Suffix *-Vsn* is optional. If an *ebin* directory exists under *Name[-Vsn]*, this directory is added to the code path.

Environment variable *ERL_LIBS* (defined in the operating system) can be used to define more library directories to be handled in the same way as the standard OTP library directory described above, except that directories without an *ebin* directory are ignored.

All application directories found in the additional directories appear before the standard OTP applications, except for the Kernel and STDLIB applications, which are placed before any additional applications. In other words, modules found in any of the additional library directories override modules with the same name in OTP, except for modules in Kernel and STDLIB.

Environment variable *ERL_LIBS* (if defined) is to contain a colon-separated (for Unix-like systems) or semicolon-separated (for Windows) list of additional libraries.

Example:

On a Unix-like system, *ERL_LIBS* can be set to the following

```
/usr/local/jungerl:/home/some_user/my_erlang_lib
```

On Windows, use semi-colon as separator.

LOADING OF CODE FROM ARCHIVE FILES

Warning:

The support for loading code from archive files is experimental. The purpose of releasing it before it is ready is to obtain early feedback. The file format, semantics, interfaces, and so on, can be changed in a future release. The function *lib_dir/2* and flag *-code_path_choice* are also experimental.

The Erlang archives are *ZIP* files with extension *.ez*. Erlang archives can also be enclosed in *escript* files whose file extension is arbitrary.

Erlang archive files can contain entire Erlang applications or parts of applications. The structure in an archive file is the same as the directory structure for an application. If you, for example, create an archive of *mnesia-4.4.7*, the archive file must be named *mnesia-4.4.7.ez* and it must contain a top directory named *mnesia-4.4.7*. If the version part of the name is omitted, it must also be omitted in the archive. That is, a *mnesia.ez* archive must contain a *mnesia* top directory.

An archive file for an application can, for example, be created like this:

```
zip:create("mnesia-4.4.7.ez",
           ["mnesia-4.4.7"],
           [{cwd, code:lib_dir()},
            {compress, all},
            {uncompress,[".beam",".app"]}]).
```

Any file in the archive can be compressed, but to speed up the access of frequently read files, it can be a good idea to store *beam* and *app* files uncompressed in the archive.

Normally the top directory of an application is located in library directory *$OTPROOT/lib* or in a directory referred to by environment variable *ERL_LIBS*. At startup, when the initial code path is computed, the code server also looks for archive files in these directories and possibly adds *ebin* directories in archives to the code path. The code path then contains paths to directories that look like *$OTPROOT/lib/mnesia.ez/mnesia/ebin* or *$OTPROOT/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin*.

The code server uses module *erl_prim_loader* in ERTS (possibly through *erl_boot_server*) to read code files from archives. However, the functions in *erl_prim_loader* can also be used by other applications to read files from archives. For example, the call *erl_prim_loader:list_dir("/otp/root/lib/mnesia-4.4.7.ez/mnesia-4.4.7/examples/bench)"* would list the contents of a directory inside an archive. See *erl_prim_loader(3)*.

An application archive file and a regular application directory can coexist. This can be useful when it is needed to have parts of the application as regular files. A typical case is the *priv* directory, which must reside as a regular directory to link in drivers dynamically and start port programs. For other applications that do not need this, directory *priv* can reside in the archive and the files under the directory *priv* can be read through *erl_prim_loader*.

When a directory is added to the code path and when the entire code path is (re)set, the code server decides which subdirectories in an application that are to be read from the archive and which that are to be read as regular files. If directories are added or removed afterwards, the file access can fail if the code path is not updated (possibly to the same path as before, to trigger the directory resolution update).

For each directory on the second level in the application archive (*ebin*, *priv*, *src*, and so on), the code server first chooses the regular directory if it exists and second from the archive. Function *code:lib_dir/2* returns the path to the subdirectory. For example, *code:lib_dir(megaco,ebin)* can return */otp/root/lib/megaco-3.9.1.1.ez/megaco-3.9.1.1/ebin* while *code:lib_dir(megaco,priv)* can return */otp/root/lib/megaco-3.9.1.1/priv*.

When an *escript* file contains an archive, there are no restrictions on the name of the *escript* and no

restrictions on how many applications that can be stored in the embedded archive. Single Beam files can also reside on the top level in the archive. At startup, the top directory in the embedded archive and all (second level) *ebin* directories in the embedded archive are added to the code path. See *erts:escript(1)*.

When the choice of directories in the code path is *strict*, the directory that ends up in the code path is exactly the stated one. This means that if, for example, the directory *$OTPROOT/lib/mnesia-4.4.7/ebin* is explicitly added to the code path, the code server does not load files from *$OTPROOT/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin*.

This behavior can be controlled through command-line flag *-code_path_choice Choice*. If the flag is set to *relaxed*, the code server instead chooses a suitable directory depending on the actual file structure. If a regular application *ebin* directory exists, it is chosen. Otherwise, the directory *ebin* in the archive is chosen if it exists. If neither of them exists, the original directory is chosen.

Command-line flag *-code_path_choice Choice* also affects how module *init* interprets the *boot script*. The interpretation of the explicit code paths in the *boot script* can be *strict* or *relaxed*. It is particularly useful to set the flag to *relaxed* when elaborating with code loading from archives without editing the *boot script*. The default is *relaxed*. See *erts:init(3)*.

CURRENT AND OLD CODE

The code for a module can exist in two variants in a system: *current code* and *old code*. When a module is loaded into the system for the first time, the module code becomes ’current’ and the global *export table* is updated with references to all functions exported from the module.

If then a new instance of the module is loaded (for example, because of error correction), the code of the previous instance becomes ’old’, and all export entries referring to the previous instance are removed. After that, the new instance is loaded as for the first time, and becomes ’current’.

Both old and current code for a module are valid, and can even be evaluated concurrently. The difference is that exported functions in old code are unavailable. Hence, a global call cannot be made to an exported function in old code, but old code can still be evaluated because of processes lingering in it.

If a third instance of the module is loaded, the code server removes (purges) the old code and any processes lingering in it are terminated. Then the third instance becomes ’current’ and the previously current code becomes ’old’.

For more information about old and current code, and how to make a process switch from old to current code, see section Compilation and Code Loading in the Erlang Reference Manual.

ARGUMENT TYPES AND INVALID ARGUMENTS

Module and application names are atoms, while file and directory names are strings. For backward compatibility reasons, some functions accept both strings and atoms, but a future release will probably only allow the arguments that are documented.

Functions in this module generally fail with an exception if they are passed an incorrect type (for example, an integer or a tuple where an atom is expected). An error tuple is returned if the argument type is correct, but there are some other errors (for example, a non-existing directory is specified to *set_path/1*).

ERROR REASONS FOR CODE-LOADING FUNCTIONS

Functions that load code (such as *load_file/1*) will return *{error,Reason}* if the load operation fails. Here follows a description of the common reasons.

badfile:
The object code has an incorrect format or the module name in the object code is not the expected module name.

nofile:
No file with object code was found.

not_purged:
The object code could not be loaded because an old version of the code already existed.

on_load_failure:
The module has an -on_load function that failed when it was called.

sticky_directory:
The object code resides in a sticky directory.

DATA TYPES

load_ret() =
{error, What :: load_error_rsn()} |
{module, Module :: module()}

load_error_rsn() =
badfile | nofile | not_purged | on_load_failure |
sticky_directory

module_status() = not_loaded | loaded | modified | removed

prepared_code()

An opaque term holding prepared code.

EXPORTS

set_path(Path) -> true | {error, What}

Types:

Path = [Dir :: file:filename()]
What = bad_directory

Sets the code path to the list of directories *Path*.

Returns:

true:
If successful

{error, bad_directory}:
If any *Dir* is not a directory name

get_path() -> Path

Types:

Path = [Dir :: file:filename()]

Returns the code path.

add_path(Dir) -> add_path_ret()

add_pathz(Dir) -> add_path_ret()

Types:

Dir = file:filename()
add_path_ret() = true | {error, bad_directory}

Adds *Dir* to the code path. The directory is added as the last directory in the new path. If *Dir* already exists in the path, it is not added.

Returns *true* if successful, or *{error, bad_directory}* if *Dir* is not the name of a directory.

add_patha(Dir) -> add_path_ret()

Types:

Dir = file:filename()
add_path_ret() = true | {error, bad_directory}

Adds *Dir* to the beginning of the code path. If *Dir* exists, it is removed from the old position in the code path.

Returns *true* if successful, or *{error, bad_directory}* if *Dir* is not the name of a directory.

add_paths(Dirs) -> ok

add_pathsz(Dirs) -> ok

Types:

Dirs = [Dir :: file:filename()]

Adds the directories in *Dirs* to the end of the code path. If a *Dir* exists, it is not added.

Always returns *ok*, regardless of the validity of each individual *Dir*.

add_pathsa(Dirs) -> ok

Types:

Dirs = [Dir :: file:filename()]

Traverses *Dirs* and adds each *Dir* to the beginning of the code path. This means that the order of *Dirs* is reversed in the resulting code path. For example, if you add *[Dir1,Dir2]*, the resulting path will be *[Dir2,Dir1|OldCodePath]*.

If a *Dir* already exists in the code path, it is removed from the old position.

Always returns *ok*, regardless of the validity of each individual *Dir*.

del_path(NameOrDir) -> boolean() | {error, What}

Types:

NameOrDir = Name | Dir
Name = atom()
Dir = file:filename()
What = bad_name

Deletes a directory from the code path. The argument can be an atom *Name*, in which case the directory with the name *.../Name[-Vsn][/ebin]* is deleted from the code path. Also, the complete directory name *Dir* can be specified as argument.

Returns:

true:
If successful

false:
If the directory is not found

{error, bad_name}:
If the argument is invalid

replace_path(Name, Dir) -> true | {error, What}

Types:

Name = atom()
Dir = file:filename()

What = bad_directory | bad_name | {badarg, term()}

Replaces an old occurrence of a directory named *.../Name[-Vsn][/ebin]* in the code path, with *Dir*. If *Name* does not exist, it adds the new directory *Dir* last in the code path. The new directory must also be named *.../Name[-Vsn][/ebin]*. This function is to be used if a new version of the directory (library) is added to a running system.

Returns:

true:
If successful

{error, bad_name}:
If *Name* is not found

{error, bad_directory}:
If *Dir* does not exist

{error, {badarg, [Name, Dir]}}:
If *Name* or *Dir* is invalid

load_file(Module) -> load_ret()

Types:

Module = module()
load_ret() =
{error, What :: load_error_rsn()} |
{module, Module :: module()}

Tries to load the Erlang module *Module*, using the code path. It looks for the object code file with an extension corresponding to the Erlang machine used, for example, *Module.beam*. The loading fails if the module name found in the object code differs from the name *Module*. *load_binary/3* must be used to load object code with a module name that is different from the file name.

Returns *{module, Module}* if successful, or *{error, Reason}* if loading fails. See Error Reasons for Code-Loading Functions for a description of the possible error reasons.

load_abs(Filename) -> load_ret()

Types:

Filename = file:filename()
load_ret() =
{error, What :: load_error_rsn()} |
{module, Module :: module()}
loaded_filename() =
(Filename :: file:filename()) | loaded_ret_atoms()
loaded_ret_atoms() = cover_compiled | preloaded

Same as *load_file(Module)*, but *Filename* is an absolute or relative filename. The code path is not searched. It returns a value in the same way as *load_file/1*. Notice that *Filename* must not contain the extension (for example, *.beam*) because *load_abs/1* adds the correct extension.

ensure_loaded(Module) -> {module, Module} | {error, What}

Types:

Module = module()
What = embedded | badfile | nofile | on_load_failure

Tries to load a module in the same way as *load_file/1*, unless the module is already loaded. However, in embedded mode it does not load a module that is not already loaded, but returns *{error, embedded}* instead. See Error Reasons for Code-Loading Functions for a description of other possible error reasons.

load_binary(Module, Filename, Binary) ->
{module, Module} | {error, What}

Types:

Module = module()
Filename = loaded_filename()
Binary = binary()
What = badarg | load_error_rsn()
loaded_filename() =

(Filename :: file:filename()) | loaded_ret_atoms()
loaded_ret_atoms() = cover_compiled | preloaded

This function can be used to load object code on remote Erlang nodes. Argument *Binary* must contain object code for *Module*. *Filename* is only used by the code server to keep a record of from which file the object code for *Module* comes. Thus, *Filename* is not opened and read by the code server.

Returns *{module, Module}* if successful, or *{error, Reason}* if loading fails. See Error Reasons for Code-Loading Functions for a description of the possible error reasons.

atomic_load(Modules) -> ok | {error, [{Module, What}]}

Types:

Modules = [Module | {Module, Filename, Binary}]
Module = module()
Filename = file:filename()
Binary = binary()
What =
badfile | nofile | on_load_not_allowed | duplicated |
not_purged | sticky_directory | pending_on_load

Tries to load all of the modules in the list *Modules* atomically. That means that either all modules are loaded at the same time, or none of the modules are loaded if there is a problem with any of the modules.

Loading can fail for one the following reasons:

badfile:
The object code has an incorrect format or the module name in the object code is not the expected module name.

nofile:
No file with object code exists.

on_load_not_allowed:
A module contains an -on_load function.

duplicated:
A module is included more than once in *Modules*.

not_purged:
The object code cannot be loaded because an old version of the code already exists.

sticky_directory:
The object code resides in a sticky directory.

pending_on_load:
A previously loaded module contains an *-on_load* function that never finished.

If it is important to minimize the time that an application is inactive while changing code, use prepare_loading/1 and finish_loading/1 instead of *atomic_load/1*. Here is an example:

```
{ok,Prepared} = code:prepare_loading(Modules),
%% Put the application into an inactive state or do any
%% other preparation needed before changing the code.
ok = code:finish_loading(Prepared),
%% Resume the application.
```

prepare_loading(Modules) ->
{ok, Prepared} | {error, [{Module, What}]}

Types:

```
Modules = [Module | {Module, Filename, Binary}]
Module = module()
Filename = file:filename()
Binary = binary()
Prepared = prepared_code()
What = badfile | nofile | on_load_not_allowed | duplicated
```

Prepares to load the modules in the list *Modules*. Finish the loading by calling finish_loading(Prepared).

This function can fail with one of the following error reasons:

badfile:
The object code has an incorrect format or the module name in the object code is not the expected module name.

nofile:
No file with object code exists.

on_load_not_allowed:
A module contains an -on_load function.

duplicated:
A module is included more than once in *Modules*.

finish_loading(Prepared) -> ok | {error, [{Module, What}]}

Types:

Prepared = prepared_code()
Module = module()
What = not_purged | sticky_directory | pending_on_load

Tries to load code for all modules that have been previously prepared by prepare_loading/1. The loading occurs atomically, meaning that either all modules are loaded at the same time, or none of the modules are loaded.

This function can fail with one of the following error reasons:

not_purged:
The object code cannot be loaded because an old version of the code already exists.

sticky_directory:
The object code resides in a sticky directory.

pending_on_load:
A previously loaded module contains an *-on_load* function that never finished.

ensure_modules_loaded(Modules :: [Module]) ->
ok | {error, [{Module, What}]}

Types:

Module = module()
What = badfile | nofile | on_load_failure

Tries to load any modules not already loaded in the list *Modules* in the same way as load_file/1.

Returns *ok* if successful, or *{error,[{Module,Reason}]}* if loading of some modules fails. See Error Reasons for Code-Loading Functions for a description of other possible error reasons.

delete(Module) -> boolean()

Types:

Module = module()

Removes the current code for *Module*, that is, the current code for *Module* is made old. This means that processes can continue to execute the code in the module, but no external function calls can be made to it.

Returns *true* if successful, or *false* if there is old code for *Module* that must be purged first, or if *Module* is not a (loaded) module.

purge(Module) -> boolean()

Types:

Module = module()

Purges the code for *Module*, that is, removes code marked as old. If some processes still linger in the old code, these processes are killed before the code is removed.

Note:

As of ERTS version 9.0, a process is only considered to be lingering in the code if it has direct references to the code. For more information see documentation of *erlang:check_process_code/3*, which is used in order to determine this.

Returns *true* if successful and any process is needed to be killed, otherwise *false*.

soft_purge(Module) -> boolean()

Types:

Module = module()

Purges the code for *Module*, that is, removes code marked as old, but only if no processes linger in it.

Note:

As of ERTS version 9.0, a process is only considered to be lingering in the code if it has direct references to the code. For more information see documentation of *erlang:check_process_code/3*, which is used in order to determine this.

Returns *false* if the module cannot be purged because of processes lingering in old code, otherwise *true*.

is_loaded(Module) -> {file, Loaded} | false

Types:

Module = module()
Loaded = loaded_filename()
loaded_filename() =
(Filename :: file:filename()) | loaded_ret_atoms()
Filename is an absolute filename.
loaded_ret_atoms() = cover_compiled | preloaded

Checks if *Module* is loaded. If it is, *{file, Loaded}* is returned, otherwise *false*.

Normally, *Loaded* is the absolute filename *Filename* from which the code is obtained. If the module is preloaded (see *script(4)*), *Loaded==preloaded*. If the module is Cover-compiled (see *cover(3)*), *Loaded==cover_compiled*.

all_available() -> [{Module, Filename, Loaded}]

Types:

Module = string()
Filename = loaded_filename()
Loaded = boolean()
loaded_filename() =
(Filename :: file:filename()) | loaded_ret_atoms()
Filename is an absolute filename.
loaded_ret_atoms() = cover_compiled | preloaded

Returns a list of tuples *{Module, Filename, Loaded}* for all available modules. A module is considered to be available if it either is loaded or would be loaded if called. *Filename* is normally the absolute filename, as described for *is_loaded/1*.

all_loaded() -> [{Module, Loaded}]

Types:

Module = module()
Loaded = loaded_filename()
loaded_filename() =
(Filename :: file:filename()) | loaded_ret_atoms()
Filename is an absolute filename.
loaded_ret_atoms() = cover_compiled | preloaded

Returns a list of tuples *{Module, Loaded}* for all loaded modules. *Loaded* is normally the absolute filename, as described for *is_loaded/1*.

which(Module) -> Which

Types:

Module = module()
Which = loaded_filename() | non_existing
loaded_filename() =
(Filename :: file:filename()) | loaded_ret_atoms()
loaded_ret_atoms() = cover_compiled | preloaded

If the module is not loaded, this function searches the code path for the first file containing object code for *Module* and returns the absolute filename.

If the module is loaded, it returns the name of the file containing the loaded object code.

If the module is preloaded, *preloaded* is returned.

If the module is Cover-compiled, *cover_compiled* is returned.

If the module cannot be found, *non_existing* is returned.

get_object_code(Module) -> {Module, Binary, Filename} | error

Types:

Module = module()
Binary = binary()
Filename = file:filename()

Searches the code path for the object code of module *Module*. Returns *{Module, Binary, Filename}* if successful, otherwise *error*. *Binary* is a binary data object, which contains the object code for the module. This can be useful if code is to be loaded on a remote node in a distributed system. For example, loading module *Module* on a node *Node* is done as follows:

```
{_Module, Binary, Filename} = code:get_object_code(Module),
rpc:call(Node, code, load_binary, [Module, Filename, Binary]),
```

get_doc(Mod) -> {ok, Res} | {error, Reason}

Types:

Mod = module()
Res = #docs_v1{}
Reason = non_existing | missing | file:posix()

Searches the code path for EEP-48 style documentation and returns it if available. If no documentation can be found the function tries to generate documentation from the debug information in the module. If no debug information is available, this function will return *{error,missing}*.

For more information about the documentation chunk see Documentation Storage and Format in Kernel’s User’s Guide.

root_dir() -> file:filename()

Returns the root directory of Erlang/OTP, which is the directory where it is installed.

Example:

```
> code:root_dir().
"/usr/local/otp"
```

lib_dir() -> file:filename()

Returns the library directory, *$OTPROOT/lib*, where *$OTPROOT* is the root directory of Erlang/OTP.

Example:

```
> code:lib_dir().
"/usr/local/otp/lib"
```

lib_dir(Name) -> file:filename() | {error, bad_name}

Types:

Name = atom()

Returns the path for the "library directory", the top directory, for an application *Name* located under *$OTPROOT/lib* or on a directory referred to with environment variable *ERL_LIBS*.

If a regular directory called *Name* or *Name-Vsn* exists in the code path with an *ebin* subdirectory, the path to this directory is returned (not the *ebin* directory).

If the directory refers to a directory in an archive, the archive name is stripped away before the

path is returned. For example, if directory */usr/local/otp/lib/mnesia-4.2.2.ez/mnesia-4.2.2/ebin* is in the path, */usr/local/otp/lib/mnesia-4.2.2/ebin* is returned. This means that the library directory for an application is the same, regardless if the application resides in an archive or not.

Example:

```
> code:lib_dir(mnesia).
"/usr/local/otp/lib/mnesia-4.2.2"
```

Returns *{error, bad_name}* if *Name* is not the name of an application under *$OTPROOT/lib* or on a directory referred to through environment variable *ERL_LIBS*. Fails with an exception if *Name* has the wrong type.

Warning:

For backward compatibility, *Name* is also allowed to be a string. That will probably change in a future release.

lib_dir(Name, SubDir) -> file:filename() | {error, bad_name}

Types:

Name = SubDir = atom()

Returns the path to a subdirectory directly under the top directory of an application. Normally the subdirectories reside under the top directory for the application, but when applications at least partly reside in an archive, the situation is different. Some of the subdirectories can reside as regular directories while others reside in an archive file. It is not checked whether this directory exists.

Example:

```
> code:lib_dir(megaco, priv).
"/usr/local/otp/lib/megaco-3.9.1.1/priv"
```

Fails with an exception if *Name* or *SubDir* has the wrong type.

compiler_dir() -> file:filename()

Returns the compiler library directory. Equivalent to *code:lib_dir(compiler)*.

priv_dir(Name) -> file:filename() | {error, bad_name}

Types:

Name = atom()

Returns the path to the *priv* directory in an application. Equivalent to *code:lib_dir(Name, priv)*.

Warning:

For backward compatibility, *Name* is also allowed to be a string. That will probably change in a future release.

objfile_extension() -> nonempty_string()

Returns the object code file extension corresponding to the Erlang machine used, namely *.beam*.

stick_dir(Dir) -> ok | error

Types:

Dir = file:filename()

Marks *Dir* as sticky.

Returns *ok* if successful, otherwise *error*.

unstick_dir(Dir) -> ok | error

Types:

Dir = file:filename()

Unsticks a directory that is marked as sticky.

Returns *ok* if successful, otherwise *error*.

is_sticky(Module) -> boolean()

Types:

Module = module()

Returns *true* if *Module* is the name of a module that has been loaded from a sticky directory (in other words: an attempt to reload the module will fail), or *false* if *Module* is not a loaded module or is not sticky.

where_is_file(Filename) -> non_existing | Absname

Types:

Filename = Absname = file:filename()

Searches the code path for *Filename*, a file of arbitrary type. If found, the full name is returned. *non_existing* is returned if the file cannot be found. The function can be useful, for example, to locate application resource files.

clash() -> ok

Searches all directories in the code path for module names with identical names and writes a report to *stdout*.

module_status() -> [{module(), module_status()}]

Types:

module_status() = not_loaded | loaded | modified | removed

See *module_status/1* and *all_loaded/0* for details.

module_status(Module :: module() | [module()]) ->
module_status() | [{module(), module_status()}]

Types:

module_status() = not_loaded | loaded | modified | removed

The status of a module can be one of:

not_loaded:
If *Module* is not currently loaded.

loaded:
If *Module* is loaded and the object file exists and contains the same code.

removed:
If *Module* is loaded but no corresponding object file can be found in the code path.

modified:
If *Module* is loaded but the object file contains code with a different MD5 checksum.

Preloaded modules are always reported as *loaded*, without inspecting the contents on disk. Cover compiled modules will always be reported as *modified* if an object file exists, or as *removed* otherwise. Modules whose load path is an empty string (which is the convention for auto-generated code) will only be reported as *loaded* or *not_loaded*.

See also *modified_modules/0*.

modified_modules() -> [module()]

Returns the list of all currently loaded modules for which *module_status/1* returns *modified*. See also *all_loaded/0*.

is_module_native(Module) -> true | false | undefined

Types:

Module = module()

Returns *false* if the given *Module* is loaded, and *undefined* if it is not.

Warning:

This function is deprecated and will be removed in a future release.

get_mode() -> embedded | interactive

Returns an atom describing the mode of the code server: *interactive* or *embedded*.

This information is useful when an external entity (for example, an IDE) provides additional code for a running node. If the code server is in interactive mode, it only has to add the path to the code. If the code server is in embedded mode, the code must be loaded with *load_binary/3*.