tm manual page - Tcl Built-In Commands

tcl.tk/man/tcl/TclCmd/tm.htm

NAME

tm — Facilities for locating and loading of Tcl Modules

SYNOPSIS

::tcl::tm::path add ?*path*...?
::tcl::tm::path remove ?*path*...?
::tcl::tm::path list
::tcl::tm::roots *paths*

DESCRIPTION

This document describes the facilities for locating and loading Tcl Modules (see **MODULE DEFINITION** for the definition of a Tcl Module). The following commands are supported:

::tcl::tm::path add ?*path*...?
The paths are added at the head to the list of module paths, in order of appearance. This means that the last argument ends up as the new head of the list.
The command enforces the restriction that no path may be an ancestor directory of any other path on the list. If any of the new paths violates this restriction an error will be raised, before any of the paths have been added. In other words, if only one path argument violates the restriction then none will be added.

If a path is already present as is, no error will be raised and no action will be taken.

Paths are searched later in the order of their appearance in the list. As they are added to the front of the list they are searched in reverse order of addition. In other words, the paths added last are looked at first.

::tcl::tm::path remove ?*path*...?
Removes the paths from the list of module paths. The command silently ignores all paths which are not on the list.

::tcl::tm::path list
Returns a list containing all registered module paths, in the order that they are searched for modules.

::tcl::tm::roots *paths*
Similar to **path add**, and layered on top of it. This command takes a single argument containing a list of paths, extends each with "**tcl***X*/**site-tcl**", and "**tcl***X*/*X*.*y*", for major version *X* of the Tcl interpreter and minor version *y* less than or equal to the minor version of the interpreter, and adds the resulting set of paths to the list of paths to search.
This command is used internally by the system to set up the system-specific default paths.

The command has been exposed to allow a build system to define additional root paths beyond those described by this document.

MODULE DEFINITION

A Tcl Module is a Tcl Package contained in a single file, and no other files required by it. This file has to be **source**able. In other words, a Tcl Module is always imported via:

```
source module_file
```

The **load** command is not directly used. This restriction is not an actual limitation, as some may believe. Ever since 8.4 the Tcl **source** command reads only until the first ^Z character. This allows us to combine an arbitrary Tcl script with arbitrary binary data into one file, where the script processes the attached data in any it chooses to fully import and activate the package.

The name of a module file has to match the regular expression:

```
([_[:alpha:]][:_[:alnum:]]*)-([[:digit:]].*)\.tm
```

The first capturing parentheses provides the name of the package, the second clause its version. In addition to matching the pattern, the extracted version number must not raise an error when used in the command:

```
package vcompare $version 0
```

FINDING MODULES

The directory tree for storing Tcl modules is separate from other parts of the filesystem and independent of **auto_path**.
Tcl Modules are searched for in all directories listed in the result of the command **::tcl::tm::path list**. This is called the *Module path*. Neither the **auto_path** nor the **tcl_pkgPath** variables are used. All directories on the module path have to obey one restriction:

For any two directories, neither is an ancestor directory of the other.

This is required to avoid ambiguities in package naming. If for example the two directories "*foo/*" and "*foo/cool*" were on the path a package named **cool::ice** could be found via the names **cool::ice** or **ice**, the latter potentially obscuring a package named **ice**, unqualified.

Before the search is started, the name of the requested package is translated into a partial path, using the following algorithm:

All occurrences of "**::**" in the package name are replaced by the appropriate directory separator character for the platform we are on. On Unix, for example, this is "*/*".

Example:

The requested package is **encoding::base64**. The generated partial path is "*encoding/base64*".

After this translation the package is looked for in all module paths, by combining them one-by-one, first to last with the partial path to form a complete search pattern. Note that the search algorithm rejects all files where the filename does not match the regular expression given in the section **MODULE DEFINITION**. For the remaining files *provide scripts* are generated and added to the package ifneeded database.

The algorithm falls back to the previous unknown handler when none of the found module files satisfy the request. If the request was satisfied the fall-back is ignored.

Note that packages in module form have *no* control over the *index* and *provide script*s entered into the package database for them. For a module file **MF** the *index script* is always:

```
package ifneeded PNAME PVERSION [list source MF]
```

and the *provide script* embedded in the above is:

```
source MF
```

Both package name **PNAME** and package version **PVERSION** are extracted from the filename **MF** according to the definition below:

```
MF = /module_path/PNAME'-PVERSION.tm
```

Where **PNAME′** is the partial path of the module as defined in section **FINDING MODULES**, and translated into **PNAME** by changing all directory separators to "**::**", and **module_path** is the path (from the list of paths to search) that we found the module file under.

Note also that we are here creating a connection between package names and paths. Tcl is case-sensitive when it comes to comparing package names, but there are filesystems which are not, like NTFS. Luckily these filesystems do store the case of the name, despite not using the information when comparing.

Given the above we allow the names for packages in Tcl modules to have mixed-case, but also require that there are no collisions when comparing names in a case-insensitive manner. In other words, if a package **Foo** is deployed in the form of a Tcl Module, packages like **foo**, **fOo**, etc. are not allowed anymore.

DEFAULT PATHS

The default list of paths on the module path is computed by a **tclsh** as follows, where *X* is the major version of the Tcl interpreter and *y* is less than or equal to the minor version of the Tcl interpreter.

All the default paths are added to the module path, even those paths which do not exist. Non-existent paths are filtered out during actual searches. This enables a user to create one of the paths searched when needed and all running applications will automatically pick up any modules placed in them.

The paths are added in the order as they are listed below, and for lists of paths defined by an environment variable in the order they are found in the variable.

SYSTEM SPECIFIC PATHS

file normalize [info library]/../tcl*X*/*X*.*y*

In other words, the interpreter will look into a directory specified by its major version and whose minor versions are less than or equal to the minor version of the interpreter.
For example for Tcl 8.4 the paths searched are:

```
[info library]/../tcl8/8.4
[info library]/../tcl8/8.3
[info library]/../tcl8/8.2
[info library]/../tcl8/8.1
[info library]/../tcl8/8.0
```

This definition assumes that a package defined for Tcl *X*.*y* can also be used by all interpreters which have the same major number *X* and a minor number greater than *y*.

file normalize EXEC/tcl*X*/*X*.*y*

Where **EXEC** is **file normalize [info nameofexecutable]/../lib** or **file normalize [::tcl::pkgconfig get libdir,runtime]**
This sets of paths is handled equivalently to the set coming before, except that it is anchored in **EXEC_PREFIX**. For a build with **PREFIX** = **EXEC_PREFIX** the two sets are identical.

SITE SPECIFIC PATHS

file normalize [info library]/../tcl*X*/site-tcl

Note that this is always a single entry because *X* is always a specific value (the current major version of Tcl).

USER SPECIFIC PATHS

$::env(TCL*X*_*y*_TM_PATH)

A list of paths, separated by either **:** (Unix) or **;** (Windows). This is user and site specific as this environment variable can be set not only by the user's profile, but by system configuration scripts as well.

$::env(TCL*X*.*y*_TM_PATH)

Same meaning and content as the previous variable. However the use of dot '.' to separate major and minor version number makes this name less to non-portable and its use is discouraged. Support of this variable has been kept only for backward compatibility with the original specification, i.e. TIP 189.

These paths are seen and therefore shared by all Tcl shells in the **$::env(PATH)** of the user.

Note that *X* and *y* follow the general rules set out above. In other words, Tcl 8.4, for example, will look at these 10 environment variables:

```
$::env(TCL8.4_TM_PATH)  $::env(TCL8_4_TM_PATH)
$::env(TCL8.3_TM_PATH)  $::env(TCL8_3_TM_PATH)
$::env(TCL8.2_TM_PATH)  $::env(TCL8_2_TM_PATH)
$::env(TCL8.1_TM_PATH)  $::env(TCL8_1_TM_PATH)
$::env(TCL8.0_TM_PATH)  $::env(TCL8_0_TM_PATH)
```

SEE ALSO

package, **Tcl Improvement Proposal #189 "*Tcl Modules*" (online at https://tip.tcl-lang.org/189.html)**, **Tcl Improvement Proposal #190 "*Implementation Choices for Tcl Modules*" (online at https://tip.tcl-lang.org/190.html)**

TIP 189: Tcl Modules

core.tcl-lang.org/tips/doc/trunk/tip/189.md

Abstract

This document describes a new mechanism for the handling of packages by the Tcl Core which differs from the existing system in important details and makes different trade-offs with regard to flexibility of package declarations and to access to the filesystem. This mechanism is called "Tcl Modules".

Background and Motivation

The current mechanism for locating and loading packages employed by the Tcl core is very flexible, but suffers from a number of drawbacks as well. These are at least partially the result of the flexibility, and thus not easily solved without giving up something.

One problem with the current mechanism is that it extensively searches the filesystem for packages, and that it has to actually read a file (*pkgIndex.tcl*) to get the full information for a prospective package. All of these operations take time. The fact that "index scripts" are able to extend the list of paths searched tends to heighten this cost as it forces rescans of the filesystem. Installations where directories in the *auto_path* are large or mounted from remote hosts are hit especially hard by this (network delays). All of this together causes a slow startup of tclsh and Tcl-based applications.

"**Tcl Modules**" on the other hand is designed with less flexibility in mind and to allow implementations to glean as much information as possible without having to perform lots of accesses to the filesystem.

Additional benefits of the proposed design are a simplified deployment of packages, akin to the way starkits made application deployment simple, and from that an easier implementation and management of repositories.

It does not come without penalties however.

- The simplified design has no "index scripts". While this does away with extending the list of paths for searching, it also does away with the ability of packages to check preconditions, like the version of the currently executing Tcl interpreter. Dependencies of packages (in module form) on particular versions of Tcl have to be managed differently and outside of them.

- "Tcl Modules" is defined to be an extension of the existing package mechanism and does *not* replace it. This means that any failure to find a package as a module *has to* cause a fall back to the regular package mechanism. It also sets a limit on how much of our goals we can reach: searching for packages which are not installed will stay relatively slow, and dominated by the filesystem scan of the regular search. This implies that "Tcl Modules" will be best suited in installations where the number of regular packages is low, and contained in a small part of the overall filesystem.

    > On the gripping hand, the only regular packages required will be packages supporting the virtual filesystems employed by modules (more on that later), so a transformation of a installation based on a set of regular packages to the form above is quite feasible.

Specification

Introduction

Modules are regular Tcl Packages, in a different guise. To ease explanations, first a summary of the existing mechanism:

> Packages are identified through "*pkgIndex.tcl*" files and the "index script" they contain. These files are read and define the "provide script", which tells Tcl how to actually load the package. In other words, the provide script tells whether to use the "**source**" or "**load**" command, which file to specify as an argument to that command, etc. However as "*pkgIndex.tcl*" contains a regular tcl script, it can do more than that and actually influence the environment, i.e., the package search itself, in several ways:
>
> > * It may choose to not register the package if conditions for the package are not met, like being run by a too old version of Tcl.
> >
> > * It may extend the list of paths used to search for packages. This implies that a package is able to modify the behaviour of the package search (usually extending the search) even before it is loaded, and even if it will not be loaded at all.

The above is very flexible, but comes at a price. The filesystem is not only searched, but files have to be read as well to build up the in-memory index of packages. And this is iterated if index files change/extend the list of paths to search.

Tcl Modules simplifies the above considerably, by cutting down on the number of indirections involved. It only searches for module files and records their location, but does not read them. The search is only performed when required, on a limited part of the filesystem. This makes locating and importing packages in module form easier and faster.

The price is that packages in module form cannot prevent registration in an interpreter not of their choice, nor can they influence the package search itself before they are actually used.

The remainder of this document will cover the following topics

- What constitutes a Tcl Module ?
- How are they found ?
- How are they indexed, i.e. entered into the package database ?

Module Definition

A Tcl Module is a Tcl Package contained in a *single* file, and no other files required by it. This file has to be **source**able. In other words, a Tcl Module is always imported via:

```
source module_file
```

The "load" command is not directly used. This restriction is not an actual limitation, as we may believe. Ever since 8.4 the Tcl **source** command reads only until the first ^Z character. This allows us to combine an arbitrary Tcl script with arbitrary binary data into one file, where the script processes the attached data in any it chooses to fully import and activate the package. Please read [190] "Implementation Choices for Tcl Modules" for more explanations of the various choices which are possible.

The name of a module file has to match the regular expression

```
([[:alpha:]_][:[:alnum:]_]*)-([[:digit:]].*)\.tm
```

The first capturing parentheses provides the name of the package, the second clause its version. In addition to matching the pattern, the extracted version number must not raise an error when used in the command

```
package vcompare $version 0
```

This additional check has several benefits. The regular expression pattern is a bit simpler, and the full version check is based on the official definition of version numbers used by the Tcl core itself.

Finding Modules

Remember the check for a valid module in last section, and notice that any filename matching this name pattern is going to be treated by the TM system as if it's a Tcl module, whether it really is or not. This means it's a bad idea for any non-Tcl module files that might match that pattern to end up in a directory where TM will be scanning. This suggests that the directory tree for storing Tcl modules ought to be something separate from other parts of the filesystem. This further implies that a new search path over just these separate storage areas would be better than Yet Another Use of *$::auto_path*.

Therefore: Modules are searched for in all directories listed in the result of the command "**::tcl::tm::path list**" (See also section 'API to "Tcl Modules"'). This is called the "Module path". Neither "*auto_path*" nor "*tcl_pkgPath*" are used.

All directories on the module path have to obey one restriction:

> For any two directories, neither is an ancestor directory of the other.

This is required to avoid ambiguities in package naming. If for example the two directories

```
foo/
foo/cool
```

were on the path a package named 'cool::ice' could be found via the names 'cool::ice' or 'ice', the latter potentially obscuring a package named 'ice', unqualified.

Before the search is started, the name of the requested package is translated into a partial path, using the following algorithm:

> All occurrences of '::' in the package name are replaced by the appropriate directory separator character for the platform we are on. On Unix, for example, this is '/'.

Example:

> The requested package is *encoding::base64*. The generated partial path is
>
> ```
> encoding/base64
> ```

After this translation the package is looked for in all module paths, by combining them one-by-one, first to last with the partial path to form a complete search pattern. The exact pattern and mechanism is left unspecified, giving the implementation freedom of choice as to what glob searches to perform, how much of them, and when.

Independent of that, the implemented algorithm has to reject all files where the filename does not match the regular expression given in the previous section. For the remaining files "provide scripts" are generated and added to the **package ifneeded** database.

The algorithm has to fall back to the previous unknown handler when none of the found module files satisfy the request. If the request was satisfied no fall-back is required.

Provide and Index Scripts

Packages in module form have no control over the "index" and "provide script"s entered into the package database for them. For a module file *MF* the "index script" is

```
package ifneeded PNAME PVERSION [list source MF]
```

and the "provide script" embedded in the above is

```
source MF
```

Both package name **PNAME** and package version **PVERSION** are extracted from the filename **MF** according to the definition below:

```
MF = /module_path/PNAME'-PVERSION.tm
```

Where **PNAME' **is the partial path of the module as defined in section 'Finding Modules' before, and translated into **PNAME** by changing all directory separators to '::', and **module_path** is the path (from the list of paths to search) that we found the module file under.

Note that we are here creating a connection between package names and paths. Tcl is case-sensitive when it comes to comparing package names, but there are filesystems which are not, like NTFS. Luckily these filesystems do store the case of the name, despite not using the information when comparing.

Given the above we allow the names for packages in Tcl modules to have mixed-case, but also require that there are no collisions when comparing names in a case-insensitive manner. In other words, if a package 'Foo' is deployed in the form of a Tcl Module, packages like 'foo', 'fOo', etc. are not allowed anymore.

Regular packages have no problem with the names of their files, as their entry point has a standard name ("*pkgIndex.tcl*") and its contents can be adjusted according to the filesystem they are stored in.

API to "Tcl Modules"

"Tcl Modules" is implemented in Tcl, as a new handler command for **package unknown**. This command calls the previously installed handler when its own search fails, thereby ensuring proper fall-back to the regular package search.

All code and data structures implementing "Tcl Modules" reside in the namespace "*::tcl::tm*".

A namespace variable holds the list of paths to search for modules, but is not officially exported. All access to this variable is done through the following public commands:

- **::tcl::tm::path add** *PATH*

  > The path is added at the head to the list of module paths.
  >
  > The command enforces the restriction that no path may be an ancestor directory of any other path on the list. If the new path violates this restriction an error will be raised.
  >
  > If the path is already present as is, no error will be raised and no action will be taken.
  >
  > Paths are searched in the order of their appearance in the list. As they are added to the front of the list they are searched in reverse order of addition. In other words, the paths added last are looked at first.

- **::tcl::tm::path remove** *PATH*

  > Removes the path from the list of module paths. The command is silently ignored if the path is not on the list.

- **::tcl::tm::path list**

  > Returns a list containing all registered module paths, in the order that they are searched for modules.

- **::tcl::tm::roots** *PATH_LIST*

  > Similar to *path add*, and layered on top of it. This command takes a list of paths, extends each with *tclX/site-tcl*, and *tclX/X.y*, for major version X of the tcl interpreter and minor version y less than or equal to the minor version of the interpreter, and adds the resulting set of paths to the list of paths to search.
  >
  > This command is used internally by the system to set up the system-specific default paths. See section *Defaults* for their definition, and that their structure matches what this command does.
  >
  > The command has been exposed to allow a buildsystem to define additional root paths beyond those defined by this document.

We do *not* provide APIs for rescanning directories, clearing internal state and such. The official interface to this functionality is "package forget" and special interfaces are neither required nor desirable.

Discussion

Restriction to "source"

This has already been discussed in the specification above.

For more discussion I again refer to [190] "Implementation Choices for Tcl Modules" which explains the various implementation choices in much more detail.

Preconditions

It has already been mentioned in section 'Background and Motivation' that preconditions in "index scripts" are lost, one of the penalties of the simplified scheme specified here.

Their existence was most important to installations with multiple versions of Tcl coexisting with each other as they could share the directory hierarchy containing packages between the various Tcl cores. This is not possible anymore, at least not in a simple manner.

For the majority of installations however, i.e. those without only one version of Tcl installed, or controlled environments like the inside of starkits and starpacks, this loss is irrelevant and of no consequence.

For more discussion please see [191] "Managing Tcl Package and Modules in a Multi-Version Environment" which explains the various choices a sysadmin has in much more detail.

Package Metadata

An area possibly made harder by Tcl Modules is the storage and query of package metadata. [59] was one way of handling such information, by storing them in the binary library of packages which have such. Another approach was to store them in the package index script, using a hypothetical **package about** command.

The latter approach has the definite advantage that it was possible to query the database of metadata for a particular package without having to actually load said package, as a load may fail if the Tcl shell used to query the database does not fulfil the preconditions for that package.

Both approaches listed above assume that it makes sense to query the database of metadata for all installed packages from a plain Tcl shell. In other words, to use the standard Tcl shell also as the tool to directly manage an installation.

It is possible to extend the proposal made in this document to handle metadata as well. We already reserved the namespace **::tcl::tm** for use by us, so it is no problem to extend the public API with commands to locate all installed packages, their metadata, and to perform queries based on this. This will require an additional specification as to how metadata is stored in/by Tcl Modules, and it will have to be understood that these extended management operations can take considerably more time than a **package require**, as they will have to scan all defined search paths and all their sub directories for Tcl Modules, and have to extract the metadata itself as well.

Deployment

The fact that a Tcl Module consists only of a single file makes its deployment quite easy. We only have to ensure correct placement in one of the searched directories when installing it locally, but nothing more.

Regarding the usage of Tcl Modules in a wrapped application, please see [190] "Implementation Choices for Tcl Modules". This is highly dependent on the implementation chosen for a specific Tcl Module and thus not discussed here, but in the referred document.

Package Repositories

At a very basic level, the physical storage, any directory tree containing properly placed files for a number of modules can serve as a package repository for the modules in it. In other words, from that point of view an installation is virtually indistinguishable from a repository, and their creation and maintenance is very easy

Note however that the higher levels of a repository, like indexing package metadata in general, or dependence tracking in particular, licensing, documentation, etc. are not addressed here and by this.

This requires standards for package metadata, format and content, topics with which this document will not deal.

Defaults

The default list of paths on the module path is computed by a tclsh as follows, where *X* is the major version of the Tcl interpreter and *y* is less than or equal to the minor version of the Tcl interpreter.

- System specific paths

> * **file normalize** [**info library**]/../tcl_X_/*X.y*
>
> > In other words, the interpreter will look into a directory specified by its major version and whose minor versions are less than or equal to the minor version of the interpreter.
> >
> > Example: For Tcl 8.4 the paths searched are
> >
> > * [**info library**]/../tcl8/8.4
> >
> > * [**info library**]/../tcl8/8.3
> >
> > * [**info library**]/../tcl8/8.2
> >
> > * [**info library**]/../tcl8/8.1
> >
> > * [**info library**]/../tcl8/8.0
> >
> > This definition assumes that a package defined for Tcl *X.y* can also be used by all interpreters which have the same major number *X* and a minor number greater than *y*.
>
> * **file normalize** *EXEC*/tcl_X_/*X.y*
>
> > Where *EXEC* is [**file normalize** [**info nameofexecutable**]/../lib] or [**file normalize** [**::tcl::pkgconfig get** libdir,runtime]]
> >
> > This sets of paths is handled equivalently to the set coming before, except that it is anchored in *EXEC_PREFIX*. For a build with *PREFIX* = *EXEC_PREFIX* the two sets are identical.

- Site specific paths.

> * **file normalize** [**info library**]/../tcl_X_/site-tcl

- User specific paths.

> * **$::env**(TCL_X_.*y*_TM_PATH)
>
> > A list of paths, separated by either **:** (Unix) or **;** (Windows). This is user and site specific as this environment variable can be set not only by the user's profile, but by system configuration scripts as well.
> >
> > These paths are seen and therefore shared by all Tcl shells in the **$::env**(PATH) of the user.
> >
> > Note that *X* and *y* follow the general rules set out above. In other words, Tcl 8.4, for example, will look at these 5 environment variables
> >
> > * **$::env**(TCL8.4_TM_PATH)
> >
> > * **$::env**(TCL8.3_TM_PATH)
> >
> > * **$::env**(TCL8.2_TM_PATH)
> >
> > * **$::env**(TCL8.1_TM_PATH)
> >
> > * **$::env**(TCL8.0_TM_PATH)

All the default paths are added to the module path, even those paths which do not exist. Non-existent paths are filtered out during actual searches. This enables a user to create one of the paths searched when needed and all running applications will automatically pick up any modules placed in them.

The paths are added in the order as they are listed above, and for lists of paths defined by an environment variable in the order they are found in the variable.

Installation

The installation of a Tcl module for a particular interpreter is basically done like this:

```
#! /path/to/chosen/tclsh
# First argument is the name of the module.
# Second argument is the base filename
set mpaths [::tcl::tm::path list]
... remove all paths the user has no write permissions for.
... throw an error if there are no paths left.
... provide the user with some UI if more than one path is left
... so that she can select the path to use.
set selmpath [ui_select $mpaths]
file copy [lindex $argv 1] \
    [file join $selmpath \
    [file dirname [string map {:: /} \
    [lindex $argv 0]]]]
```

Glossary

The following terms and definitions are used throughout the document

- *index script*

  > A script used to index a package, or not. Usually contained in a file named "*pkgIndex.tcl*". Can check preconditions for a package and contains package specific code for setting up the package specific *provide script*.

- *provide script*

  > This is a package specific script and tells Tcl exactly how to import it. In the existing package system it is generated and registered by the *index script*. Tcl Modules on the other hand generates it based on information gleaned from filenames.

Reference Implementation

A reference implementation is available in Patch 942881 http://sf.net/tracker/?func=detail&aid=942881&group_id=10894&atid=310894

TIP 190: Implementation Choices for Tcl Modules

core.tcl-lang.org/tips/doc/trunk/tip/190.md

Abstract

This document is an informational adjunct to [189] "Tcl Modules", describing a number of choices for the implementation of Tcl Modules, pure-Tcl, binary, or mixed. It lists these choices and then discusses their relative merits and problems, especially their interaction with wrapping, i.e. when used in a wrapped application. The main point of the document is to dispel the illusion that the restriction to the "source" command for the loading Tcl Modules is an actual limitation. A secondary point is to make recommendations regarding preferred implementations, based the merits and weaknesses of the various possibilities.

Implementation Choices

A small recap first: Tcl Modules are Packages in a single file, and only **source** is used to import them into the running interpreter. These restrictions are the backdrop to all implementations discussed here.

Packages Written in Tcl

These are easy.

- A package which is implemented in a single file is already in the form required for a Tcl Module and nothing has to be done at all.

  > Most packages in Tcllibhttp://tcllib.sf.net can be of this form.

- In the case of a package whose implementation is spread over multiple .tcl files the solution is equally simple. Just concatenate all the files into one file when generating the distribution. This is a trivial operation.

  > Some packages in Tcllibhttp://tcllib.sf.net can be of this form, or rewritten into it.

- The usage of an compiler/obfuscater like TclPro/Tcl Dev Kit on such Tcl Modules is also no problem. While the result of these compilers contains binary, it is in encoded form, and the file is still a proper Tcl script which can be handled by **source**. The encoded binary is decoded by an adjunct package, **tbcload**, whose import is the first action done by the script.

  > This also points us already to the general solution for binary packages, i.e. usage of supporting packages to handle arbitrary data embedded or attached in some way in/to an initialization script.

The usage of pure-Tcl Modules within wrapped applications poses no problems at all.

Also note that all of the choices available to binary packages, as explained in the next section, are available to pure-Tcl packages as well.

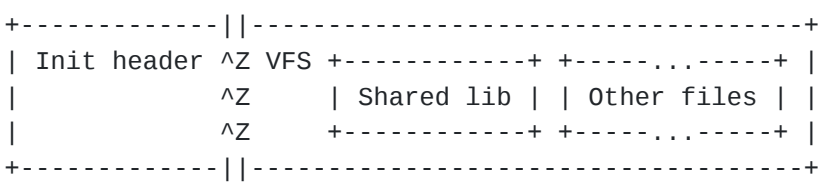
Binary Packages

A binary package consists of a shared library, possibly with adjunct Tcl and data files. These have to be bundled into a single file to be a Tcl Module.

The general approach to this is to combine an init script written in Tcl with binary data attached to it, both sections separated by a ^Z character. This is possible since 8.4, where **source** was changed to read only up to the first ^Z and ignore the remainder of the file, whereas other **file** and channel operations will see it.

Embedding a Virtual Filesystem

The most obvious way of doing this is the any-kit approach: a small initialization script in front which loads all required supporting packages and then uses them to mount an attached virtual filesystem containing all the other files. After the mount any package specific initialization can be performed, either in the initialization script itself, or in a separate script file stored in filesystem. The latter is the recommended form as it keeps the main initialization script small and package neutral i.e. it will be only filesystem specific, and not package specific. These two tasks are kept separate, which is good design in general, and becomes more important later on as well.



A concrete example of this are starkits, except that they use this technique to wrap an application into a single file, and not a package.

When interacting with wrapping this approach runs into problems. It is not possible to simply copy the module file into the wrapped application and then use it. The problem is a limitation in most implementations of alternate filesystem: they are not able to mount a

virtual file again as a directory, i.e. *nested* mounting. This however is required when a Tcl Module using the any-kit approach is placed into a wrapped application. It was thought for a while that this could be a limitation in the VFS core of Tcl itself, but further investigation proved this to be wrong. This proof came in the form of TROFS, the *Tcl Read Only Filesystem*, by Don Porter. This filesystem supports nesting and thus shows that the Tcl core is strong enough for this as well. It is the implementation of a filesystem which determines if nesting is possible or not, and most do not support this.

See also SF bug report *[941872] path<->FS function limitations* http://sf.net/tracker/?func=detail&aid=941872&group_id=10894&atid=110894 for more details on this and other problems.

There are two ways to work around this limitation, while it exists. These are explained below. However note that even if the limitation is removed we may still run into performance problems because of a file accessed through several layers of file systems, each with its own overhead. The workarounds we are about to discuss will help with this as well by removing layers of indirection and are therefore of general importance.

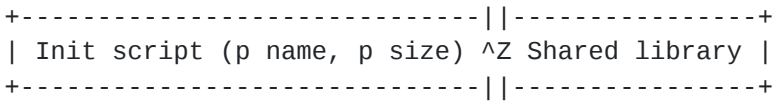
- The standard initialization script of the module is given code to recognize that it is stored in a virtual filesystem, and will copy the whole file to a temporary location and perform the mount on that. This workaround has to be done by every package.
- The wrapper application used to create the wrapped application is extended with code which works around the problem. It would basically convert the Tcl Module into a regular package by copying the virtual filesystem in the module as a directory, and adding all the necessary scripts, like "pkgIndex.tcl". Here the separation of filesystem specific from package specific initialization comes into play as well as it makes the unbundling much easier. The generated package index file can simply refer to the same package initialization script as the filesystem specific header of the bundled module.

    > It should be noted that unbundling is limited to the filesystems which are recognized by the wrapper application. Because of this a combination of this and the previous approach might be best, as it allows the module to function even if the wrapper application was not able to unbundle it.

More a problem of taste might be that Tcl Modules in this form require additional packages which implement the filesystem they use. This can be remedied in the future by adding additional reflection capabilities to the Tcl core which would allow the implementation of channel drivers, channels transformations, and filesystems in pure Tcl, and then implementing simple filesystems based on that. This would also allow the Tcl core itself to make use of filesystems attached to its shared libraries and executables.

Appending a Shared Library

Should the binary package consist of only one shared library we can forgo the use of a full-blown virtual filesystem and simply attach the shared library to the init script as is. Instead of mounting anything the init script just has to copy the library to a temporary place and then "load" it.



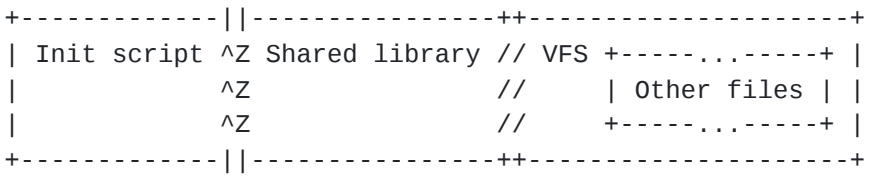
Tcl Modules implemented in this way will have no problems when used in a wrapped application as they will always copy their relevant file to the native filesystem before using it.

The disadvantage is that this is not a very general scheme. There are not that much packages which consist of only one shared library and nothing else.

Note: Should we ever get loading of a shared library directly from memory or from a location in another file, then copying the library to the filesystem won't be necessary anymore either.

Appending a Library and a VFS

An extension of the last approach is to attach the virtual file system not to the init script, but the shared library.



This approach has the same advantages as the last with regard to its interaction with wrapping, i.e no problems, and additionally handles additional files coming with the shared library. The initialization of the VFS happens in the init script, but after the shared library has been loaded.

I have to admit that I am not sure if this will truly work. In essence the library will have to be told about the directory for its files after its C level initialization has been run.

If the VFS is required during the C level initialization then the VFS has to be initialized and mounted from within the shared library, i.e. at the C level. This is not very convenient as we need an embedded Tcl script for this, and that makes the code of the library more complicated than required.

Recommendations

We currently recommend usage of the any-kit approach for binary packages, despite its problem with nested mounting. This approach has an existing implementation in the metakit-based starkits and is thus well tested in general. The other two approaches are

currently purely theoretical, with neither any implementation, nor testing.

Regarding Tcl packages no recommendation is necessary as we have in essence only one possibility for the more complex case, the simple concatenation of multiple files into a single one.

TIP 191: Managing Tcl Packages and Modules in a Multi-Version Environment

core.tcl-lang.org/tips/doc/trunk/tip/191.md

Abstract

This document is an informational adjunct to [189] "Tcl Modules", describing a number of choices for the management of Tcl Modules in environments with more than one version of the Tcl core installed. It lists these choices and then discusses their relative merits and problems.

Background and Motivation

A regular package can perform checks in its "pkgIndex.tcl" file regarding the environment the package would be loaded into should it be requested, and make the creation of its "provide script" dependent on the result. In other words, it is able to prevent its registration, making it invisible to the Tcl interpreter in question if the environment is not right (for example, if the interpreter is too old a version of Tcl).

A Tcl module cannot do this as its "provide script" is generated by the module system.

In a controlled environment, like wrapped applications of any form this is a complete non-issue as we can assume that only those modules are installed which are not only required, but needed.

This is no problem either for installations with only one version of the Tcl core. It is believed that this is currently the majority of cases.

The change breaks only environments with several coexisting Tcl installations which share package directories among them and rely on the index scripts to prevent the registration of packages in unsuitable interpreters.

Another situation where the change can break things is an environment with a single version of the interpreter, and the version of that interpreter is changed, upgraded, or downgraded. Packages for one version may not work anymore with the new version, or a different version of the package has to be selected from among the installed versions. This situation can be viewed as having multiple version of Tcl, however over time instead of space.

For the environments with multiple versions of Tcl in space a number of possible solutions are explained in the next section.

Choices

All solutions are done outside of the Tcl interpreter, in the filesystem.

- Each interpreter has its own part of the filesystem. Modules required in several of them are copied around. Modules not required are not copied. This is easy. It requires more disk space; however that is cheap.
- Same as above, but use hard- and/or soft-links instead of copying. Modules not eligible somewhere are not linked.

    > This schema can also be used to maintain a central repository, which is just a directory tree containing all module files in their proper locations. Then link the packages which should be visible to an interpreter into their respective directory trees.
    >
    > This makes the creation of test environments with a known set of packages very easy as well.

- Keep the modules in several directory trees as wanted and/or needed by sharing requirements and then set the list of search paths used by an interpreter to exactly those trees which have the modules required/usable by it.

Changes over Time

Note that the default paths set down in [189] ease the management, as each Tcl shell will not only have its own space, but also access to extensions for all minor versions which came before it. This means that placing an extension into the directory for the smallest version of Tcl supporting it will make this extension available to this minor version and all the versions which come after and share the major version. This is the right thing almost all of the time.

Only extensions using internal interfaces will have to be dealt with separately.