Source Code Navigation

Finding your way

As a projects code-base grows in size to the point where it gets too complex for a developer to hold in memory, easy navigation becomes increasingly important. BY DEAN WILSON

While high level approaches such as class hierarchy diagrams or comprehensive documentation can help to cut down on the time spent searching source files, a more important factor is the effort of context switching.

When you are already in your editor writing code, you should not have to break the flow and leave it to look up a method or function.

While most good editors will let you shell out and run a *grep* or a *find* command over the file system, we are going to explore a tool that is more integrated with our editor, assuming you use one that is listed at *http://ctags.sourceforge.net/tools.html*, which includes both Emacs and Vim.

This tool, called *ctags*, builds up a tag (or index) file of language objects that can be reached from the comfort of your editor with no more effort than a couple of keystrokes.

As of version 5.4 ctags is aware of 28 different languages. So while we use code samples written in Java, the same principles can be applied to your own project no matter which language it is in. The exact objects available varies for each language, typically these include equivalents to classes, methods, functions and packages.

Getting Ctags

Ctags is available from *http://ctags.sourceforge.net/* under the GPL and follows the standard install process (see Listing 1).

Once you have compiled and installed the application, you can run the command *ctags --version* to ensure that it has installed correctly, followed by the command *ctags -V*. This second command will display a list of file extension to language mappings and any configuration files ctags has found and will use when run.

Due to Unix paying little heed to the extension of a file, ctags runs through a number of methods to detect which language a source-file is written in:

- File extension (.pl, .java)
- Shebang line (If the file is executable)

If neither of these returns a language ctags recognizes then the file will be ignored. ctags behaviour as it runs through a selection of files can be viewed by running a *ctags -V*, it will display the name of each file encoun-

Listing 1: Standard Install

```
# where <version> is a string
# such as 5.4

tar -zxvf ctags-<version>.tar.gz
cd ctags-<version>
./configure

#start the actual build
make

#this requires root privileges
make install
```

tered and either list the language matched or show it as skipped. If you have a number of script files without file extensions then they are required to have executable permissions set, otherwise ctags will not parse the shebang line (The first line of the script, #!/bin/perl is an example of a shebang line.) This is not documented in the man pages and can cause some puzzlement when you first begin using the tool.

Now we have installed ctags and have some diagnostic capabilities under our belt we can look at the benefits that the tool provides. Shown in Listing 2 are two small snippets of Java source code, a class and an application that uses it. While the examples are simplistic and very stripped down they will serve to illustrate the principles of the tool.

Listing 2: Java source

```
//simple counter class, Tally.java
public class Tally {
  int tally;

  public Tally() {
    tally = 0;
  }

  public Tally(int num) {
    //start tally at 'num'
    tally = num;
  }

  public void incrTally(int num) {
    tally += num;
  }

  public void decrTally(int num) {
    tally -= num;
  }
}
```

The *Tally.java* class found in Listing 2, should be fairly self explanatory even to non-Java coders, an instance of this class serves as a counter that can either be increased or decreased and has multiple constructors, the first of which creates the instance with a value of zero while the second allows a starting value to be defined.

The *usetally.java* source which is shown in Listing 3, does very little, it creates and then modifies two instances of the Tally object before exiting silently.

If we now run *ctags* * in the directory containing these two files a new file called *tags* is created. If we open this file in our editor we can then see how simple the tag file format is.

We first have a number of comments detailing the version of ctags being used, these can be recognized by the *!* at the start of the line. Following these lines is the actual information we are interested in (wrapped to fit, in the tag file it is all one line):

```
incrTally   Tally.java  /^  ↵
public void incrTally(intnum)↵
 {$/;"    m class:Tally
```

Working through the tab delimited fields the first field is the name of the object found, in this case its one of our methods, *incrTally*.

The second field is the file name the object was found in while the third field is the regular expression used by ctags to locate the object in the file. The fourth field is a short description of the objects type, possible values vary by language, in this case an m in a Java file means a method.

The full list of languages and the objects supported under them is available in the ctags manual at sourceforge (*http://ctags.sourceforge.net/ctags.html*)

Listing 3: usetally.java

```
// usetally.java
public class usetally {
  public static void main(String[] args) {

  Tally delivered = new Tally();
  Tally dispatched = new Tally(10);

  delivered.incrTally(5);
  delivered.incrTally(5);
  dispatched.decrTally(5);
  }
}
```

Using ctags with Vim

Now we have a tag file let us begin navigating by using it in conjunction with vim (A short introduction to ctags and Emacs can be found in the sidebar.) All Java applications have a *main()* method that serves as an entry point to the program and is executed first. While we only have two source files to worry about unless you know which file this method is in you may end up having to search through both of them, using vim we can pass the problem to ctags:

```
$ vim -t main
```

By default vim looks for a file called (not surprisingly) tags for the tag definitions it should be aware of. You may have in the past invoked vim with either *vim +12 filename* to go directly to line 12 of *filename* or even a *vim +/word filename* to jump directly to the first occurrence of a word in the document, but in this case you are telling vim which tagged object you would like. Vim will then look-up the name of the file containing the object and the pattern that matches it from the tag-file and with this information take you to that object.

Now that we are in the *usetally.java* file, if we find a method call we want to investigate (such as *decrTally*) then we can move directly to its definition by entering *:tag decrTally* in Vim's command mode.

To return to our starting position in the *usetally.java* file you can enter *Ctrl t* and we will be whisked back. If you move through a chain of tags one after the other you can use the command *:tags* to

display the path you have taken. You can navigate through this list using *:[num] tags* to move forward the specified number of tags and *[num] Ctrl t* to move back. If you wish to jump back to the tag at the top of the list you can use the quick command *:tag* to save going back multiple levels.

While it is useful to be able to jump to another tag by name regardless of position it is much more common in day to day editing to need a refresher regarding a piece of code that is on screen. In an attempt to save the number of keystrokes needed to type in the full-name of the object we are interested in, we have a shortcut available.

To see this in action move down to the line containing *dispatched.decrTally(5);* and move the cursor over *decrTally*, you can now type *Ctrl]* and you will be whisked away to the method definition without needing to know which file or even which directory the method was defined in.

To jump back to where you were editing once you have browsed through the method you can enter *Ctrl t* and be back in *usetally.java*.

If you are unsure of the full name of an object that you want to display or are just a lazy typist, you can use vim to auto-complete the potential options by typing *:tag Tall* while in vim's command mode and then pressing tab.

If any tags are available that begin with *Tall* then the tag name will be auto-completed. In the event of more than one tag being a possible match, pressing tab again will move to the next tag; eventually this will cycle through all the possible tags and return to the first match. Once you find the tag you wish to move to, you just need to press return.

In a similar vein if you are unsure of the tags name then you can do a wild-card search with *:tag /tally* and then use the tab key to iterate through all the tags containing the string tally (This method is case insensitive) until you find the desired match.

Listing 4: getTally method

```
// add to Tally.java within the
// outermost curly braces
  public int getTally() {
    return(tally);
  }
```

Listing 5: getTally calls

```
//this line already present
dispatched.decrTally(5);
System.out.println("Num delivered: " + delivered.getTally());
System.out.println("Num dispatched: " + dispatched.getTally());
```

You may have noticed that when we displayed an object using a tag, the screen updated to show us only the code at our destination. To keep both on screen simultaneously we can either enter *:stag tagname* instead of *:tag tagname* to open an arbitrary tag in another window or we can use *Ctrl W]* to do the same with the object under the cursor. If the window created using the latter is not large enough then you can specify a number of lines before the command, so to open the object under the cursor in a 20 line window you would type *20 Ctrl W]* in command mode.

Advanced Tag Files

You may have noticed that we said you wouldn't need to know even the directory containing the method, if your project has a simple structure with a single base point then you can run:

```
$ ctags -R *
```

Which will recursively walk through each directory building up a single tag file containing the objects for all the source files it finds. Some important details to note when using this approach to create a project wide tag file is that by default ctags builds its tag-file using relative paths; a tag file that is moved from one directory to another is unlikely to continue working.

A second issue that crops up when using ctags like this is that of projects with different base directories. The source files used in your project may come from completely different parts of the file system.

One of the simplest ways of working around this is to run ctags once for each base path, but with the *-a* option so that the tag file is appended to rather than overwritten. This will also sort the tag-file for efficient searching.

While we have workarounds to both of the annoyances presented above it is possible to use a little knowledge of ctags, shell and the find command to tailor the contents of the tag-file to meet our needs. If ctags is given an absolute file name to search through then the objects listed in the tag-file will retain the absolute path, while this may seem like trivia it allows us to generate a tag file full of absolute paths with a find command:

```
$ ctags `find /usr/src/java
project/ -name "*java" -print`
```

In this example the back-ticks surrounding both the find command and its arguments alter the order of command execution so that find is run before ctags. When it has finished running its output is passed back into ctags providing the arguments ctags needs. This also allows us to run ctags over a number of different source trees at once by specifying multiple paths to *find*.

Combining ctags with other commands allows its functionality to be extended to meet the specific needs of your project. Most languages have the ability to reuse external code so with a little ingenuity you could even use ctags to dynamically generate comprehensive tag files based upon the projects current code base without even needing to know the libraries used, they could be determined from the code itself.

Now we have covered how to build both a tag-file and navigate within vim using it we can look at some less obvious details. Whenever we have mentioned the tag-file we have said that it uses patterns rather than line numbers to match objects. This design has both strengths and weaknesses, lets add some code to each file and then look at them.

In the *Tally.java* class add the getTally method to the body of the class, its exact position is irrelevant (see Listing 4). We can then add some calls to *getTally* in

usetall.java under the *dispatched. decrTally(5);* line (see Listing 5).

We now run *ctags -a ** to append any new objects to the tag-file. Running ctags in append mode will not add all the functions twice, it will only add those not present without needing to adjust those already in the file. This added to the fact that changing the code around the tagged objects does not break the tags are the two main reasons for choosing this implementation.

However, like most benefits, this tactic brings problems of its own into play. If we go back and look at the *Tally.java* class then it becomes clear that it has two constructors.

Open the *usetally.java* file and move the cursor to the *Tally()* section of the *Tally delivered = new Tally();* line. If you then jump to the constructor using *Ctrl]* you will be taken to the first constructor, the one that takes no arguments, as expected.

Now navigate back and then repeat the process with the line beneath. In this line the Tally object is created with an argument passed in. Notice how you are, incorrectly this time, taken to the same constructor as before?

Tagging Emacs

Although we use vim to illustrate the uses of ctags in the body of this article it is also possible to use tags from within Emacs, allowing us all of the same benefits. By default Emacs uses an external application called etags to generate its tag-files (which are named *TAGS* and have a different internal format from those generated by ctags.)

Fortunately when you install ctags a symbolic link is created called etags that invokes the ctags binary but in etags emulation mode. When invoked as etags its output is in the TAGS file format.

Emacs allows the full set of tag operations that vim does and this is a brief introduction to some of them. For more details about the when and why you would use them you should consult the main body of the text. Emacs is a complex and powerful editor, unless you are already an Emacs user familiar with principles such as the META key, this brief explanations given here will not be enough to get you started.

To read in a TAG file enter *M-x* and type *visit-tags-table* followed by return. You will then be prompted for the location of the TAGS file you wish to use. Select this and press return. To jump to a tag by name type *M-.* and then you can either type the full tag-name or enter a partial name and use tab for auto-completion.

If you want to jump to the tag under the cursor then you enter the same command (*M-.*) but press return to accept the default value, which is the currently selected tag. To return to your base point you enter *M-**.

If there are multiple possible matches for the destination tag once you have visited the first match you can always iterate through them with *C-u M*.

Full details of tags within Emacs can be found within the Emacs tag info page, a document that excels in coverage if not in being user-friendly.

Working

When ctags scans through the tag-file the first object that matches your destination is returned, in this case it is the first constructor every time as ctags does not look at the argument list of an object, in the eyes of ctags all constructors are created equal.

At first glance this seems like a show stopping bug but in line with the "Keep It Simple Stupid" (KISS) philosophy ctags delegates the resolution of this to the editor itself, ctags gathers information about all available object, how to best resolve conflicts is left to the editor. In Vim we have the *tnext* and *tselect* commands to remove this issue.

If you jump to a match (Using either *:tag tagname* or *Ctrl]*) and do not see the code you are expecting you can issue a *:tnext* in command mode and you begin to cycle forward through the possible matches. In this case if you try and jump to the second constructor you will be taken to the first match, which is incorrect, you can then issue a *:tnext* to continue through to the next possible match.

When dealing with an object that has a number of constructors or a class hierarchy that has multiple classes with identically named methods, iterating though the potential matches can be as time consuming as looking up the documentation. Instead of going through one at a time you can enter *:tselect* and be presented with a list of all possible matches along with some meta-data about the match, such as the type of object and the name of the file containing it.

In addition to the basic *tnext* and *tselect* navigation commands shown vim cal also use additional commands (see Table 1). If not specified in the command then [num] defaults to one. As an aside the *:tselect* command can take a search string (such as *:tselect /Tally*) and it will return a list of everything in the tag file that matches, along with a small amount of meta-data that may make your choice a little easier.

Table 1: vim commands

command	moves to
:tfirst	first match
:[num]tprevious	[num] previous match
:[num]tnext	[num] next match
:tlast	last match

Closing Tag

Now that you have been introduced to the basic functionality of ctags when used in conjunction with a powerful editor, you will hopefully find navigating through a tangle of unfamiliar source code much less of a chore.

ctags is a paragon of the "each tool should do one thing and do it well" philosophy and once you start to include it in your toolbox, either running it by hand or combining it with your other build tools you will wonder how you ever got along without it. ■

THE AUTHOR

Dean Wilson is a Unix system admin and software developer at WebPerform Group Ltd.

He has encountered outdated documentation once to often for his own good.