

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

Introduction

Software is abstract and non-tactile by its very nature. It can be difficult to see what it is doing and why it may be misbehaving. To get a better view of software, we often use tools like gdb, leaks, lsof, and sc_usage, just to name a few. We even still use "caveman debugging" techniques like recompiling the code with additional print statements.

A few years back, Sun Microsystems developed DTrace: a new and innovative way to trace running software on live systems. DTrace enables developers and administrators to "see" what their code, and others' code, is doing in a flexible and dynamic way. With the release of Leopard, Apple has brought DTrace to Mac OS X.

This article will begin with a crash course in DTrace. If you're already a seasoned DTrace veteran, feel free to skip that section. We will then move on to some examples of how to use DTrace by exploring our Leopard systems and discovering what makes them purr.

A crash course in DTrace

DTrace is a software tracing facility that can dynamically instrument code by modifying a program after it gets loaded into memory. DTrace can be used on production systems with optimized binaries, without ever having to restart the application - let alone, recompile it! Moreover, DTrace is not limited to tracing user-space applications like most other tracing tools, such as ktrace, strace, and truss. Parts of the system, such as the kernel itself, that were previously off-limits to runtime inspection are now fair game. And, since DTrace instruments code dynamically at runtime, it has zero overhead when not in use.

That said, DTrace will not, and should not, replace all of your other tools. You'll still want to use Shark, Sampler, ObjectAlloc, leaks, Instruments, etc. where they make sense. And let's not forget about good ol' fashion thinking. DTrace is not magic; it's just another tool (albeit a powerful tool) in your toolbox.

DTrace is often used to help answer questions about software, such as "Is function foo ever being called, and if so, by whom?", and "How much time is my code spending in the pwrite system call?". However, you must know what to ask. If you are tracing your own software, you probably have a good understanding of how it's supposed to work, so coming up with the right questions might not be too difficult. However, if you're using DTrace to explore someone else's software, it may be more difficult to ask the right questions. But never fear; we'll see later that there are some very common questions that are generally good jumping off points. As in life, the answer to one question often begets another. Follow your nose! Using DTrace is very much like surfing the web: your questions about the system are hyperlinks that when clicked will take you to another page full of new links/questions.

You interact with DTrace by writing small programs in the D programming language. These D programs can be saved in text files and run like shell scripts, or they can be stretched out right on the command line for quick, ad-hoc use (or if you simply want to impress your friends). An example D script that totals all the system calls made for each process on the system is shown in Listing 1.

Listing 1: syscalls_per_proc.d

Totals up all the system calls made for each process

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

```
syscall:::entry
{
    @[execname] = count();
}
```

When run for about 5 seconds on my laptop I got the following output:

```
$ sudo dtrace -s syscalls_per_proc.d
dtrace: script 'syscalls_per_proc.d' matched 427 probes
^C
Quicksilver    1
Finder         2
Pages          2
DirectoryServic  3
fseventsd     3
mds           6
ntpd          21
WindowServer   22
mDNSResponder  24
dtrace        38
Terminal      85
```

We could have also specified the D script on the command line as follows:

```
$ sudo dtrace -n 'syscall:::entry { @[execname] = count() }'
```

Key Concepts and the D Programming Language

The D programming language has syntax very similar to C's and should be very easy for C programmers to use. The D program structure, however, is more akin to AWK's and is made up of one or more clauses of the following form.

```
probe descriptions
/ predicate /
{
    action statements
}
```

One of the key concepts in DTrace is that of a probe, which identifies a point (or points) of interest in the kernel or in a user process. Probes are identified by their probe description, which is either a unique integer ID, or more commonly, a 4-tuple written as provider:module:function:name.

Fields in a probe description may use shell-style globbing, and omitted fields are assumed to match everything. For example, the probe description `syscall::read:entry` identifies the beginning of the read system call by naming the syscall provider, any module, the read function, and the probe name entry. The probe description `syscall::*read:entry`, however, identifies both the read and pread system calls. You can see a list of many (but not all!) of the probes on your system by running `dtrace -l`.

```
$ sudo dtrace -l | wc -l
40808
```

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

Each probe may be associated with an optional block of action statements that will be evaluated when the probe fires. A probe may also have an optional predicate that must evaluate to true before the probe's action statements will be called. For example, the following will use the built-in variable `execname` and print the name of each process that calls `read`.

```
syscall::read:entry
{
    printf("%s\n", execname);
}
```

And the following will only print the names of processes that are reading from their standard input (file descriptor 0):

```
syscall::read:entry
/arg0 == 0/
{
    printf("%s\n", execname);
}
```

It is not always insightful to see a separate line of output each time a probe fires. Instead, we may be interested in looking at the data in aggregate form. For example, we could find the total number of bytes allocated by `malloc` by setting a probe at the entry to `malloc`, printing out the size argument (`arg0`), then post-processing the data to sum it up. That would work. But DTrace makes this much easier by using aggregating functions and data structures called aggregates. In this way, I could see that Safari mallocs about 2.4MB on my machine when loading my website's homepage.

```
$ sudo dtrace -n 'pid147::malloc:entry { @total = sum(arg0) }'
```

```
dtrace: description 'pid147::malloc:entry ' matched 2 probes
```

```
^C
```

```
2414468
```

Here's how that DTrace script (i.e., the argument to `-n`) breaks down:

- The full "probe description" is `pid147::malloc:entry`, which uses the `pid` provider to set a probe at the entry of the `malloc` function in process ID 147 (Safari)
- We do not specify a predicate, so the action statements are always executed when the probe fires
- The only action statement in this example is the use of the aggregating function `sum`, which totals the argument passed to `malloc`, and stores the result in the aggregate data structure named `@total`
- Nothing is displayed until we hit `ctrl-c`; at which point `@total` is displayed
- A large part of writing useful D scripts is knowing what built-in variables and functions are available. The following are a few of the more useful built-in D variables:
 - `arg0, ..., arg9` - The first 10 arguments to the function matching the probe
 - `execname` - The name of the currently executing process
 - `pid` - The process ID of the currently executing process
 - `ppid` - The parent process ID of the currently executing process

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

- `probefunc` - The function name of the current probe description

And here are some of the more frequently used data recording functions:

- `trace()` - Evaluates its argument and outputs the result
- `printf()` - Like `trace()`, but takes a C-style format string
- `stack()` - Records a kernel stack trace
- `ustack()` - Records a user stack trace

DTrace Architecture

Users generally interact with DTrace through the `dtrace(1)` command, which is a generic front-end to the DTrace facility (an alternative front-end is Instruments, which we will not cover in this article). D programs get compiled in user-space and sent to the DTrace virtual machine in the kernel for execution. In other words, the D scripts you write are sent into the kernel and run inside the kernel's address space. They do not run in the `dtrace` process, nor do they run in the target process you are trying to instrument. This is important, and it is the reason we need DTrace functions like `copyin` and `copyinstr`, which are functions that copy data from user space into the kernel's address space.

DTrace providers live in the kernel and can be thought of as plugins to the in-kernel DTrace framework. They are responsible for creating probes by instrumenting various parts of the system. The probes made available by a given provider can be listed using the `dtrace` command. For example, the command `dtrace -l -n proc:::` will list all the probes available from the `proc` provider. The following are some of the providers available on Leopard and what they instrument.

- `syscall` - System calls in the kernel
- `fbt` (Function Boundary Tracing) - Functions in the kernel
- `proc` - Functions related to the process life cycle
- `mach_trap` - Mach traps in the kernel
- `pid` - C functions (or individual instructions) in user space
- `objc` - Objective-C objects in user space

Figure 1 shows the relationship between some of the different components of DTrace.

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

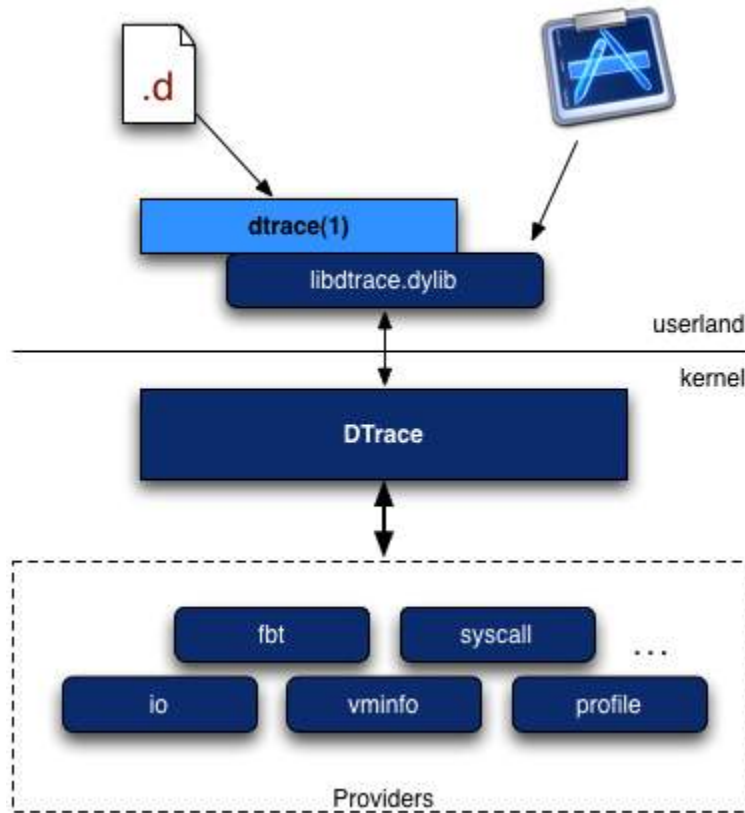


Figure 1. DTrace architecture

Exploring Leopard

Let us now look at a few examples of using DTrace to explore our Leopard system. Note that in order to minimize line wrapping we use /dev/stdin as the argument to `dtrace -s`, we type our D scripts right into the standard input, then close standard input by typing `ctrl-d (^D)`.

Tracing Objective-C messages

Tracing system calls and other low-level functions can be fun and insightful, but it may also be too low-level for some situations. Many of the great application frameworks on OS X are written in Objective-C, and it would be nice to trace them at that level. Apple apparently agreed, and they equipped DTrace with a provider for tracing Objective-C messages.

The objc provider is very much like the pid provider, in that the provider name must include the process ID of the target process. The objc provider exposes Objective-C class names as probe modules, and selector names as probe functions. For example, the probe description `objc123:NSView:-isFlipped:entry` would match the entry to the `isFlipped` instance method on the `NSView` class in process 123. Let's try this out by watching what Safari does when it loads `www.unixjunkie.net`:

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

```
$ sudo dtrace -q -s /dev/stdin
objc3447:::entry
{
    printf("%s %s\n", probemod, probefunc);
}
^D
dtrace: script '/dev/stdin' matched 66888 probes
(... TONS of output omitted ...)
NSConcreteNotification -recycle
NSObject -retainCount
NSCFString -release
NSObject -release
NSGarbageCollector +defaultCollector
NSLock -lock
NSThread +currentThread
NSObject -hash
NSCFArray -countByEnumeratingWithState:objects:count:
NSLock -unlock
(...)
^C
```

That's cool, but it's a TON of information. With a tool as powerful as DTrace, you can quickly find yourself with more data than you can grok. In this case, we could lessen the output by sharpening our probe description to match only what we're really interested in. For example, if we were only interested in methods dealing with URL handling, we could use the probe description `objc3447:NSURL*:::entry`. This quickly cuts the 66,000+ probes down to a manageable 500.

Another way to conquer this mountain of information is with an aggregating function. For example, let's say we're now interested in how long these Objective-C messages take to complete. We could figure this out using the following D script.

Listing 2: `objc_msg_times.d`

This is a D script to quantize the running time of Objective-C messages. We use the timestamp built-in D variable to record the entry time to the Objective-C method in a thread-local variable. Upon the method's return, we quantize the difference between the current timestamp and the start time. `$target` is a special D variable that evaluates to the PID of the process under inspection.

```
objc$target:::entry
{
    self->start = timestamp;
}
objc$target:::return
/self->start/
{
    @ = quantize(timestamp - self->start);
    self->start = 0;
}
$ sudo dtrace -s objc_msg_times.d -p 3447
dtrace: script 'objc_msg_times.d' matched 136042 probes
^C
      value  ----- Distribution ----- count
      1024 |                                     0
```

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

2048	@@@@	526082
4096	@@@@	188440
8192	@@@@	84940
16384	@	22815
32768		6999
65536		464
131072		82
262144		33
524288		26
value	----- Distribution -----	count
1048576		13
2097152		9
4194304		3
8388608		5
16777216		0

This script introduces a couple new things: thread-local variables and the quantize function. D allows you to save variables in thread-local storage by using the self-> syntax. These variables will be available in other clauses that fire on the same thread. We also use the built-in quantize aggregating function to build a power-of-two frequency distribution of the Objective-C messaging times; this can be an incredibly powerful way to interpret the data collected by DTrace. We see here that most of the Objective-C messages completed within 2048-4096 nanoseconds.

File activity

It can be enlightening to see which files are accessed on a system. For example, you may see that Foo.app is frequently writing to some file, or maybe that Bar.app is calling stat(2) on a log file every 10ms. This information can help you debug your own programs, or perhaps better understand the system in general. Below we use a small D script to print out the name of each file as it's opened.

```
$ sudo dtrace -s /dev/stdin
syscall::open*:entry
{
    printf("%s %s", execname, copyinstr(arg0));
}
^D
dtrace: script '/dev/stdin' matched 3 probes
CPU      ID      FUNCTION:NAME
  0  17584  open:entry Finder /.vol/234881026/562669
  0  17584  open:entry Finder /.vol/234881026/562669
  1  17584  open:entry iChatAgent /Users/jgm/Library/Caches/...
  0  17584  open:entry iChatAgent /Users/jgm/Library/Caches/...
  1  17584  open:entry iChat /System/Library/PrivateFrameworks/...
^C
```

This script sets a probe at the entry to all system calls having names beginning with "open". DTrace tells us that our probe description matched three probes. They are: open, open_extended, and open_nocancel. Our action statement prints out the name of the process (execname) that caused the probe to fire, and the first argument (arg0) to the function that matched the probe. Notice that we need to use the copyinstr function here rather than just printing arg0 directly. This is because D scripts execute in the kernel's address space, but the pathname argument to open is stored in user space. We could also modify our D script so that it shows us which files are accessed most often, as follows.

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

```
$ sudo dtrace -s /dev/stdin
syscall::open*:entry
{
    @[copyinstr(arg0)] = count();
}
^D
dtrace: script '/dev/stdin' matched 3 probes
^C
/Library/Managed Preferences/com.apple.Terminal.plist    1
/Library/Preferences/com.apple.Terminal.plist            1
/Users/jgm/Library/Caches/com.apple.iChat/Pictures/...    1
.                                                          2
/.vol/234881026/562669                                    2
/Users/jgm/Library/Preferences/com.apple.Terminal.plist    3
```

I am a little surprised to see Finder opening files in the `/.vol` directory. Volfs is a separate file system that is used to support the Carbon File Manager atop the BSD file system. It was used on earlier versions of Mac OS X, but it was removed in Leopard.

```
$ sw_vers -productVersion
10.5
$ df -k
Filesystem      1024-blocks    Used Available Capacity  Mounted on
/dev/disk0s2    81732372 38929756  42546616    48%      /
devfs           112         112         0    100%     /dev
fdesc           1           1           0    100%     /dev
map -hosts      0           0           0    100%     /net
map auto_home   0           0           0    100%     /home
$ ls -al /.vol
total 0
drwxr-xr-x@  2 root  wheel   68 Jul 30 22:08 ./
drwxrwxr-t  31 root  admin  1122 Sep 16 10:22 ../
```

I do not have a volfs file system, but apparently, accessing volfs paths still works.

```
$ ls -l /.vol/234881026/562669
total 3512
(... output snipped for brevity ...)
-rw-r--r--  1 jgm  staff    0 Sep 17 11:50 I_am_on_your_desktop.txt
```

Apparently, `/.vol/234881026/562669` refers to my Desktop. We can see which functions are using these paths by looking at the user stack trace when they are opened.

```
$ sudo dtrace -s /dev/stdin
syscall::open*:entry
/copyinstr(arg0) == "/.vol/234881026/562669"/
{
    ustack();
}
^D
dtrace: script '/dev/stdin' matched 3 probes
CPU      ID          FUNCTION:NAME
  1  17584          open:entry
```

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

```
libSystem.B.dylib`open$UNIX2003+0xa
CarbonCore`PBOpenIteratorSync+0x203
CarbonCore`FSOpenIterator+0x1d
DesktopServicesPriv`THFSPlusIterator::First(THFSPlusRef&)+0x6f
DesktopServicesPriv`THFSPlusIterator::Next(THFSPlusRef&)+0x26
DesktopServicesPriv`THFSPlusSynchronizer::...+0xb5
DesktopServicesPriv`TNode::SynchronizeChildren(bool)+0x44
DesktopServicesPriv`TNode::ReconcileChildren(bool, bool)+0x63
DesktopServicesPriv`TNode::HandleSync(bool, bool, bool, bool)+0x1b1
DesktopServicesPriv`TNodeSyncTask::...+0xda
DesktopServicesPriv`TNodeSyncTask::...+0x11f
DesktopServicesPriv`TNodeSyncTask::SyncTaskProc(void*)+0x98
CarbonCore`PrivateMPEntryPoint+0x38
libSystem.B.dylib`_pthread_start+0x141
libSystem.B.dylib`thread_start+0x22
^C
```

Looking at this stack, we can see that it is indeed the Carbon File Manager APIs that are using these volfs paths. Just as we expected. So, even though volfs no longer exists as a file system in Leopard, its main functionality still exists in the kernel to support the Carbon File Manager. Thanks DTrace!

Hard linking directories

One addition in Leopard that has the potential to send shivers up spines, is the addition of directory hard linking. This functionality was added to enable Time Machine to backup large directory structures of unchanged data without wasting space. The canonical argument against hard linked directories is that they can cause cycles in the directory tree. However, Apple avoided this problem by placing some restrictions on directory hard linking.

The only problem I currently see with directory hard links is that I can't get them to work.

```
$ mkdir Dir1
$ ln Dir1 Dir2
ln: Dir1: Is a directory
$ sudo ln Dir1 Dir2
ln: Dir1: Is a directory
```

Perhaps we can use DTrace to figure out what's going on. Let's start like we normally do by looking at all the system calls made by ln. We will use dtrace's -c option to run and trace the command in question. As we've already seen, the PID of the command is made available to our D script through the \$target macro variable.

```
$ sudo dtrace -s /dev/stdin -c "ln Dir1 Dir2"
syscall:::entry
/pid == $target/
{}
^D
dtrace: script '/dev/stdin' matched 427 probes
ln: Dir1: Is a directory
dtrace: pid 4389 has exited
CPU      ID      FUNCTION:NAME
  1  17950      stat:entry
```

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

```
1 18368          write_nocancel:entry
1 18368          write_nocancel:entry
1 18368          write_nocancel:entry
1 18368          write_nocancel:entry
1 17576          exit:entry
```

This is interesting because we don't see any calls to `link`, which is the system call ultimately responsible for creating the hard link. Let's see if we can get a better view of what `ln` is doing, by using the `D` script in listing 3.

Listing 3: `ln.d`

Uses the `pid` provider to trace all function calls in `libSystem`. To help limit the output, we only look at user stacks with a depth less than 6.

```
pid$target:libSystem*::entry,
pid$target:libSystem*::return
/ustackdepth < 6/
{}
```

And we will run it like this:

```
$ sudo dtrace -F -s ln.d -c "ln Dir1 Dir2"
dtrace: script 'ln.d' matched 8679 probes
ln: Dir1: Is a directory
dtrace: pid 6171 has exited
CPU FUNCTION
1 <- __cxa_atexit
1 -> rindex
1 <- rindex
1 -> getopt$UNIX2003
1 <- getopt$UNIX2003
1 -> stat
1 -> _sysenter_trap
1 -> __error
1 <- __error
1 -> warn
1 <- __error
1 <- vwarnc
1 <- fprintf
1 <- warn
1 -> exit
1 -> __cxa_finalize
1 <- __cxa_finalize
1 -> _cleanup
1 <- _fwalk
1 <- _cleanup
1 <- exit
1 -> _exit
1 -> _sysenter_trap
```

This script traces the function calls in `libSystem`, and it uses the `-F` (flow indent) option to visually indicate when functions are entered and return. This output shows that `stat` is called, followed by

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

functions to print the error message. This indicates that `ln` itself is detecting that the first path is a directory and is giving us the error without ever calling `link`. To get past this stumbling block, we will write our own simple C program that calls `link` directly.

Listing 4: `hlink.c`

Given two file name arguments, creates a hard link from the first to the second.

```
#include <unistd.h>
#include <stdio.h>
int
main(int argc, char *argv[])
{
    if (argc != 3)
        return 1;
    int ret = link(argv[1], argv[2]);
    if (ret != 0)
        perror("link");
    return ret;
}
$ gcc -o hlink hlink.c -Wall
$ ./hlink Dir1 Dir2
link: Operation not permitted
$ sudo ./hlink Dir1 Dir2
link: Operation not permitted
```

OK, it still didn't work, but at least we know that `link` is definitely being called. We could use DTrace's `fbt` provider to dig further down in the kernel and perhaps see why we're getting this new error, but let's not forget what we learned above: DTrace does not replace all of our other tools. It takes time to write D scripts, so maybe spending a few more seconds just thinking about the problem is a better solution. In our case we know that hard links do indeed work because Time Machine uses them. So, perhaps the problem is that we're trying to create the hard link in the same directory as the original. Let's try creating the hard link to a directory with a different parent.

```
$ mkdir NewDir
$ ./hlink Dir1 NewDir/Dir2
$ ls -liD Dir1 NewDir/Dir2
1140587 drwxr-xr-x  2 jgm  staff  68 Sep 17 18:08 Dir1/
1140587 drwxr-xr-x  2 jgm  staff  68 Sep 17 18:08 NewDir/Dir2/
```

Cool! It worked! `Dir2` is a hard link to `Dir1` and I don't even feel dirty (nor did I need to be root). We can verify that the hard link worked by looking at the directories' inode numbers (the first column in our `ls` output).

Conclusion

As we've seen here, DTrace is a very powerful and flexible tool, but it is just that - a tool. And it's best used in the hands of a skilled craftsman. The OpenSolaris DTrace community provides a collection of useful D scripts, under the name DTraceToolkit. Thankfully, the code-smiths at Apple have ported many of these scripts to Leopard. You can get a list of the more than 40 available DTrace scripts by running `man -k dtrace`. There are also examples of their use in `/usr/share/examples/DTTk`. One of the

EXPLORING LEOPARD WITH DTRACE

HOW TO USE DTRACE FOR DEBUGGING AND EXPLORATION

by Greg Miller

more useful D scripts is called dtruss, which most users will embrace as a replacement for ktrace. Some of the DTrace examples in this article could have been replaced by simpler dtruss invocations, but then what fun is that?! We wanted to play with DTrace.

We've just barely scratched the surface of the tip of the DTrace iceberg. We didn't get to touch on the more exotic topics like DTrace's "destructive" actions, the fasttrap provider, instrumenting individual instructions, or accessing external variables using the scoping operator. The technology behind DTrace is impressive, and it is open source so you can see what is happening behind the scenes. Perhaps Apple will provide some DTrace documentation soon, but in the meantime you can find a lot of documentation on Sun's website