## Recon-Erlang线上系统诊断工具

源码在[这里](https://github.com/ferd/recon)。Recon, 它的文档非常清晰，见[这里](http://ferd.github.io/recon/index.html)。

Recon主要包括三个模块：

recon  
Main module, contains basic functionality to interact with the recon application. It includes functions to gather information about processes and the general state of the virtual machine, ports, and OTP behaviours running in the node. It also includes a few functions to facilitate RPC calls with distributed Erlang nodes.

recon\_lib  
Regroups useful functionality used by recon when dealing with data from the node. Would be an interesting place to look if you were looking to extend Recon’s functionality

recon\_alloc  
Regroups functions to deal with Erlang’s memory allocators, or particularly, to try to present the allocator data in a way that makes it simpler to discover the presence of possible problems.

和一系列脚本用于在发生crashdump的时候帮助用户分析到原因, 设计的时候充分考虑到了对系统的最小影响，在线上使用是很安全的。

其中最有价值的是 recon\_alloc， 基本上把内存分配器的细节和复杂都屏蔽起来，用户可以很好的看到内存工作的效率.

<https://github.com/zhongwencool/observer_cli>

**[Linux/Unix笔记本](http://www.cnblogs.com/ggjucheng/archive/2012/08/18/2645321.html)**

<http://www.cnblogs.com/ggjucheng/archive/2012/08/18/2645321.html>

关于send/recv/accept

gen\_tcp:send

<http://blog.yufeng.info/archives/1581>

<http://blog.yufeng.info/archives/336>

当我们调用gen\_tcp:send的时候, kernel模块会根据gen\_tcp socket的类型决定调用相应的模块. 这个模块要么是inet\_tcp, 要么是inet6\_tcp. 这个模块会把发送请求委托给prim\_inet模块. prim\_inet模块首先检查Socket是否合法, 如果合法然后调用erlang:port\_command把系统推到ERTS运行期.  
这个推的结果有2个: 1. 成功, 进程挂起等待运行期的反馈. 2. 失败,立即返回.什么情况下会失败呢?  
1. 驱动不支持soft\_busy, 但是我们用了force标志  
2. 驱动已经busy了, 但是我们不允许进程挂起.

inet\_drv内部每个socket都有个消息队列, 保持着上层推来的消息. 这个消息队列有上下水位线的. 当消息的字节数目超过了高水位线的时候, inet\_drv就把socket标志为busy. 这个busy要到队列的字节数少于低水位线的时候才解除.

如果一条消息成功的推到协议栈, 那么tcp 驱动会给调用者进程发送{inet\_reply,S,Status}消息, 反馈结果. 这时候调用者进程也就是tcp:send返回, 完成了整个流程.

总结起来就是，gen\_tcp数据的发送需要占用宿主进程的reds,这也可能造成宿主进程被挂起，在设的时候尽量避免一个进程拥有太多的port. 更好的做法是: 手工把gen\_tcp的2个步骤分开做:  
1. 不停的erlang:port\_command(S, Data, OptList)  最好加上force标志   
2. 被动等待{inet\_reply,S,Status} 消息.

3. 并且使用inet:setopts(Socket, Options)设置socket的{send\_timeout, Integer} {send\_timeout\_close, Boolean}选项

## 效率监控 Erlang监测系统CPU、内存、磁盘

 (2011-11-17 09:36:57)

[IMG_256](http://blog.sina.com.cn/s/javascript:;)转载▼

|  |  |
| --- | --- |
| 标签： [杂谈](http://search.sina.com.cn/?c=blog&q=%D4%D3%CC%B8&by=tag" \t "http://blog.sina.com.cn/s/_blank) | 分类： [erlang1](http://blog.sina.com.cn/s/articlelist_2528682324_1_1.html" \t "http://blog.sina.com.cn/s/_blank) |

## os\_mon

Erlang的os\_mon服务中提供了一些用于监测系统信息的服务  
cpu\_sup：监测CPU负载和使用率（Unix）  
disksup：监测磁盘（Unix、Windows）  
memsup：监测内存（Unix、Windows、VxWorks）  
os\_sup：监测系统日志（Solaris、Windows）

使用os\_mon进行监测先必须启动监测服务*application:start(os\_mon)* ，因为os\_mon服务依赖于sasl服务，先必须启

动sasl服务，*application:start(sasl)* ，否则会返回*{error,{not\_started,sasl}}* 错误。os\_mon提供的四种监测服

务中默认会启动三种服务：cpu\_sup、disksup和memsup，如果需要自己设置启动的监测服务，可以修改os\_mon.app

文件中的配置参数  
start\_cpu\_sup = bool()  
start\_disksup = bool()  
start\_memsup = bool()  
start\_os\_sup = bool()  
等于true时启动，等于false时不启动。os\_mon.app文件在erlang的安装目录下../erl5.8.3/lib/os\_mon-2.2.5

/ebin 。

## cpu\_sup

cpu监测在5.8.3版本中只能用于Solaris和Linux操作系统，负载值与Unix进程运行前在队列中的排队时间成正比，

因此值越大意味着负载越高，返回值除以256为rup和top命令中显示的值。*avg1/0,avg5/0* 和*avg15/0* 函数计算负载，

*util/0* 和*util/1* 函数计算CPU使用率。在Linux系统中，必须保证/proc文件目录能被cpu\_sup服务访问，如果不能监

测服务会停止

****模块中的函数列表****  
nprocs() -> UnixProcesses | {error, Reason}  
返回UNIX进程数  
avg1() -> SystemLoad | {error, Reason}  
返回最后1分钟系统的负载  
avg5() -> SystemLoad | {error, Reason}  
返回最后5分钟系统的负载  
avg15() -> SystemLoad | {error, Reason}  
返回最后15分钟系统的负载  
util() -> CpuUtil | {error, Reason}  
返回CPU使用率  
util(Opts) -> UtilSpec | {error, Reason}  
返回CPU使用率的详细信息

调用这些函数取CPU监测数据时，如果前后两次调用，数值没有变化时显示为0，有点奇怪

## disksup

disksup是一个用来监测磁盘空间的进程，适用于Unix和Windows系统。监测服务定期检查磁盘，对于每个磁盘或分

区，在它使用超过一定的可用空间量，通过*{{disk\_almost\_full，MountedOn}，[]}* 设置产生报警。在Unix下所有的

本地磁盘都会被监测，包括存在的交换分区。在WIN32下所有类型为“FIXED\_DISK”逻辑驱动器都会被检查。

****配置监控间隔时间和阀值****  
disk\_space\_check\_interval = int()>0  
监测间隔时间，单位为分钟，默认为30分钟。  
disk\_almost\_full\_threshold = float()  
监测阀值，磁盘使用率达到多少时产生告警，默认为80，单位是百分比。

****模块中的函数列表****  
get\_disk\_data() -> [DiskData]  
返回最后一次磁盘检查结果  
get\_check\_interval() -> MS  
获取监测间隔时间，单位是毫秒  
set\_check\_interval(Minutes) -> ok  
设置监测间隔时间，这个设置在下一次监测时生效，服务退出后，这个值会失效，重启服务后使用默认值  
get\_almost\_full\_threshold() -> Percent  
获取监测阀值，为磁盘使用率  
set\_almost\_full\_threshold(Float) -> ok  
设置监测阀值，服务重启后，设置失效，使用默认值

## memsup

memsup用来监控系统内存和各个进程内存的使用率，适用于Unix、Windows和VxWorks系统，定时监测内存，如果内

存使用超过系统分配的一定值，通过*{system\_memory\_high\_watermark, []}*设置产生告警。如果系统中任何Erlang

进程使用内存超过在总内存中的一定百分比，通过设置*{process\_memory\_high\_watermark,Pid}* 产生告警。

****配置监测间隔时间和阀值****  
memory\_check\_interval = int()>0  
以分钟为刻度，默认为1分钟  
system\_memory\_high\_watermark = float()  
内存使用阀值，默认为80，单位是百分比  
process\_memory\_high\_watermark = float()  
单个Erlang进程使用阀值，默认为5，单位是百分比  
memsup\_helper\_timeout = int()>0  
等待监测结果的超时时间，默认为30秒  
memsup\_system\_only = bool()  
设置是否只监控系统内存使用率还是同时监测Erlang进程内存使用率，默认为false

****模块中的函数列表****  
get\_memory\_data() -> {Total,Allocated,Worst}  
获取系统总内存，使用内存，每个Erlang进程的使用内存  
get\_system\_memory\_data() -> MemDataList  
获取系统内存使用的详细信息  
get\_os\_wordsize() -> Wordsize  
获取操作系统的位数  
get\_check\_interval() -> MS  
获取监测间隔时间，单位毫秒  
set\_check\_interval(Minutes) -> ok  
设置监测间隔时间，单位分钟  
get\_procmem\_high\_watermark() -> int()  
获取每一进程内存使用告警阀值  
set\_procmem\_high\_watermark(Float) -> ok  
设置每一进程内存告警阀值  
get\_sysmem\_high\_watermark() -> int()  
获取系统内存使用阀值  
set\_sysmem\_high\_watermark(Float) -> ok  
设置系统内存使用阀值  
get\_helper\_timeout() -> Seconds  
获取监测数据返回等待时间  
set\_helper\_timeout(Seconds) -> ok  
设置监测数据返回等待时间

# **Erlang项目内存泄漏分析方法**

随着项目越来越依赖Erlang，碰到的问题也随之增加。前段时间线上系统碰到内存高消耗问题，记录一下troubleshooting的分析过程。线上系统用的是Erlang R16B02版本。

**问题描述**

有几台线上系统，运行一段时间，内存飙升。系统模型很简单，有网络连接，pool中找新的process进行处理。top命令观察，发现内存都被Erlang进程给吃完了，netstat命令查看网络连接数，才区区几K。问题应该是Erlang内存泄漏了。

**分析方法**

Erlang系统有个好处，可以直接进入线上系统，在生产现场分析问题。我们系统是通过Rebar管理的，可以用不同方法进入线上系统。

**本机登录**

可以直接登录到线上机器，然后通过以下命令attach到Erlang系统里面

复制代码代码如下:

$ cd /path/to/project  
$ rel/xxx/bin/xxx attach  
(node@host)>

**通过remote shell**

**获取Erlang系统的cookie**

复制代码代码如下:

$ ps -ef |grep beam  %%找到参数 --setcookie

**新开一个shell，使用同样的cookie，不同的nodename**

复制代码代码如下:

$ erl --setcookie cookiename -name test@127.0.0.1

**用start remote shell进入系统**

复制代码代码如下:

Erlang R16B02 (erts-5.10.3) [source] [64-bit] [smp:2:2] [async-threads:10] [hipe] [kernel-poll:false]

Eshell V5.10.3  (abort with ^G)  
(test1@127.0.0.1)1> net\_adm:ping('node@127.0.0.1').  
pong  
(test1@127.0.0.1)2> nodes().  
['node@127.0.0.1']  
(test1@127.0.0.1)3>   
User switch command  
 --> h  
  c [nn]            - connect to job  
  i [nn]            - interrupt job  
  k [nn]            - kill job  
  j                 - list all jobs  
  s [shell]         - start local shell  
  r [node [shell]]  - start remote shell  
  q                 - quit erlang  
  ? | h             - this message  
 --> r 'node@127.0.0.1'  
 --> j  
   1  {shell,start,[init]}  
   2\* {'node@127.0.0.1',shell,start,[]}  
 --> c 2

**分析流程**

Erlang有很多工具，可以分析系统信息，比如[appmon](http://www.erlang.org/documentation/doc-5.6.1/pdf/appmon-2.1.9.pdf" \t "http://www.jb51.net/article/_blank)，[webtool](http://erlang.org/doc/man/webtool.html" \t "http://www.jb51.net/article/_blank)。但是系统内存严重不足，已经没有办法启动这些工具了，幸好还有Erlang shell。

Erlang shell自带了很多有用的[命令](http://www.erlang.org/doc/man/shell.html" \t "http://www.jb51.net/article/_blank)，可以用help()方法查看

复制代码代码如下:

> help().

**Erlang系统内存消耗情况**

top结果显示，是内存问题，所以第一步可以先看看Erlang的系统内存消耗情况

复制代码代码如下:

> erlang:memory().

[memory()](http://www.erlang.org/doc/man/erlang.html" \l "memory-0" \t "http://www.jb51.net/article/_blank)可以看到Erlang emulator分配的内存，有总的内存，atom消耗的内存，process消耗的内存等等。

**Erlang process创建数量**

线上系统发现主要内存消耗都在process上面，接下来要分析，是process内存泄漏了，还是process创建数量太多导致。

复制代码代码如下:

> erlang:system\_info(process\_limit).  %%查看系统最多能创建多少process  
> erlang:system\_info(process\_count).  %%当前系统创建了多少process

[system\_info()](http://www.erlang.org/doc/man/erlang.html" \l "system_info-1" \t "http://www.jb51.net/article/_blank)返回当前系统的一些信息，比如系统process，port的数量。执行上面命令，大吃一惊，只有2，3k的网络连接，结果Erlang process已经有10多w了。系统process创建了，但是因为代码或者其它原因，堆积没有释放。

**查看单个process的信息**

既然是因为process因为某种原因堆积了，只能从process里找原因了

先要获取堆积process的pid

复制代码代码如下:

> i().  %%返回system信息  
> i(0,61,886).  %% (0,61,886)是pid

看到有很多process hang在那里，查看具体pid信息，发现message\_queue有几条消息没有被处理。下面就用到强大的[erlang:process\_info()](http://erlang.org/doc/man/erlang.html" \l "process_info-2" \t "http://www.jb51.net/article/_blank)方法，它可以获取进程相当丰富的信息。

复制代码代码如下:

> erlang:process\_info(pid(0,61,886), current\_stacktrace).  
> rp(erlang:process\_info(pid(0,61,886), backtrace)).

查看进程的backtrace时，发现下面的信息

复制代码代码如下:

0x00007fbd6f18dbf8 Return addr 0x00007fbff201aa00 (gen\_event:rpc/2 + 96)  
y(0)     #Ref<0.0.2014.142287>  
y(1)     infinity  
y(2)     {sync\_notify,{log,{lager\_msg,[], ..........}}  
y(3)     <0.61.886>  
y(4)     <0.89.0>  
y(5)     []

process在处理Erlang第三方的日志库[lager](https://github.com/basho/lager" \t "http://www.jb51.net/article/_blank)时，hang住了。

**问题原因**

查看lager的文档，发现以下信息

复制代码代码如下:

Prior to lager 2.0, the gen\_event at the core of lager operated purely in synchronous mode. Asynchronous mode is faster, but has no protection against message queue overload. In lager 2.0, the gen\_event takes a hybrid approach. it polls its own mailbox size and toggles the messaging between synchronous and asynchronous depending on mailbox size.

{async\_threshold, 20}, {async\_threshold\_window, 5}

This will use async messaging until the mailbox exceeds 20 messages, at which point synchronous messaging will be used, and switch back to asynchronous, when size reduces to 20 - 5 = 15.

If you wish to disable this behaviour, simply set it to 'undefined'. It defaults to a low number to prevent the mailbox growing rapidly beyond the limit and causing problems. In general, lager should process messages as fast as they come in, so getting 20 behind should be relatively exceptional anyway.

原来lager有个配置项，配置message未处理的数量，如果message堆积数超出，则会用 同步 方式处理！

当前系统打开了debug log，洪水般的log把系统给冲垮了。

老外也碰到类似问题，这个[thread](https://groups.google.com/forum/" \l "!searchin/erlang-programming/waiting$20handle_info$20timeout/erlang-programming/JL8HVBjnWy0/nEoBDIhhMFUJ" \t "http://www.jb51.net/article/_blank)给我们的分析带来很多帮助，感谢一下。

# **[[Erlang 0033] 接入Erlang控制台的几种方法](http://www.cnblogs.com/me-sa/archive/2012/01/18/erlang0033.html)**

      在window中调试的时候我们可以通过启动多个cmd窗口运行Erlang节点,在生产环境中我们需要Erlang服务在Centos服务器上后台运行;这就需要在启动的时候添加启动参数detached来脱离终端:

**-detached** Starts the Erlang runtime system detached from the system console. Useful for running daemons and backgrounds processes. Implies -noinput.

      对于我们自己的服务,即使部署到了生产环境一定要做到"像魔术师的飞刀,出手但并没有脱手",还是需要一些方式进入到Erlang后台进程来做一些工作比如:查看某一个Erlang节点的运行时信息(内存,进程数等),让服务优雅的退出而不是kill进程,或者做一下热更新(参见:[[Erlang 0010] Erlang 热更新](http://www.cnblogs.com/me-sa/archive/2011/10/29/erlang0010.html) 当然热更新可以使用reloader.erl的方案来简化);一开始的时候服务器比较少,我们采用的是JCL的方式去处理的;

# **Erlang Shell JCL**

JCL是Erlang Shell的一种运行模式,即Job Control Mode (JCL, in which jobs can be started, killed, detached and connected).我们启动两个节点来完成这个操作;

2012-11-14新增备注:下面的实验是在Linux下完成的,Windows下JCL需要启动werl.exe

**Node\_1** 添加了-detached选项,启动之后直接在后台运行并没有启动Shell  
erl -setcookie abc -name node\_1@192.168.1.123 -detached   
**Node\_2** 使用了和Node\_1相同的cookie,启动之后进入Erlang Shell界面  
erl -setcookie abc -name node\_2@192.168.1.123  
下面我们开始在node\_2@192.168.1.123演练JCL:

[IMG_256](http://www.cnblogs.com/me-sa/archive/2012/01/18/javascript:void(0);)

Eshell V5.9 (abort with ^G)  
(node\_2@192.168.1.123)1> node(). %当前这是在node\_2  
'node\_2@192.168.1.123'  
(node\_2@192.168.1.123)2> %Ctrl + G 进入JCL模式  
User switch command  
--> h   
 c [nn] - connect to job  
 i [nn] - interrupt job  
 k [nn] - kill job  
 j - list all jobs  
 s [shell] - start local shell  
 r [node [shell]] - start remote shell  
 q - quit erlang  
 ? | h - this message  
--> r'node\_1@192.168.1.123' %尝试连接到node\_1@192.168.1.123   
--> j  
 1 {shell,start,[init]} %列出所有的Job  
 2\* {'node\_1@192.168.1.123',shell,start,[]}  
--> c 2 %这里2是job的编号,切换到job 2   
Eshell V5.9 (abort with ^G)  
(node\_1@192.168.1.123)1> node(). %注意提示符,现在已经是在node\_1  
'node\_1@192.168.1.123'  
(node\_1@192.168.1.123)2> erlang:now().  
{1326,801888,347570}  
(node\_1@192.168.1.123)3> %再一次Ctrl + G  
User switch command   
--> j   
 1 {shell,start,[init]}  
 2\* {'node\_1@192.168.1.123',shell,start,[]}  
--> c 1 %切换到job 1  
(node\_2@192.168.1.123)2> node(). %注意提示符,我们已经回到了node\_2  
'node\_2@192.168.1.123'  
(node\_2@192.168.1.123)3>

[IMG_257](http://www.cnblogs.com/me-sa/archive/2012/01/18/javascript:void(0);)

这样来来回回切换是不是有点盗梦空间的意思?是不是可以更简单一点,比如直接进入node\_1呢?借助**-remsh**参数就可以做到

看看-remsh的说明恰好是我们需要的:

If you want an Erlang node to have a remote job active from the start (rather than the default local job), you start Erlang with the -remsh flag. Example: erl -sname this\_node -remsh other\_node@other\_host

动手试一下:  
erl -setcookie abc -name node\_3@192.168.1.123 -remsh node\_1@192.168.1.123 %%这样就直接进入了node\_1节点  
**注意**:**直接进入到了node\_1,执行完操作了想要退出怎么办?** 你要是在这里执行一下q(). node\_1这个节点就直接死掉了;  
正确的方法还是Ctrl+G进入JCL模式然后执行q命令退出;使用ps aux|grep node查看一下进程是不是还在

ejabberd网站上提到了这个方法   
[1] Attach an Erlang Shell to an Already Running ejabberd Process http://www.ejabberd.im/tricks  
[2] Interconnecting Erlang Nodes http://www.ejabberd.im/interconnect-erl-nodes