erlang深度分析

作者: mryufeng http://mryufeng.javaeye.com

分析erlang的VM, 性能的定量分析，编码最佳实践,
工具介绍都在这里。

目 录

1. erlang

1.1 erlang的dist研究 10

1.2 erlang distributed application 11

1.3 减少linux swap对erlang程序的影响 12

1.4 erlang的beam模拟器 13

1.5 erlang dist protocol 格式 14

1.6 erts运行期内存使用情况查看 15

1.7 质疑 apache和yaws的性能比较(必看) 16

1.8 erl_call erlang cnode 功能强大 20

1.9 erlang inet:setopts 未公开选项 22

1.10 如何看erts内部的状态 24

1.11 erl CTRL+C do_break 的功能 25

1.12 方便调试erlang程序的脚本 28

1.13 erlang 模拟器工作原理分析 30

1.14 erlang的erl_crash.dump产生以及如何解读 31

1.15 erlang数据库 ETS 工作原理分析 32

1.16 Erlang: 当你net_adm:ping(Node)的时候发生了什么？流程复杂 但是对理解dist工作原理至关重要 34

1.17 Erlang 进程堆内存增长算法 47

1.18 erlang中类似netstat的命令 48

1.19 erlang热部署 49

1.20 查看erlang内存使用情况 50

1.21 开始规模研究global 和 global_group global_search的原理 62

1.22 erlang远程控制 63

1.23 erlang sctp支持 65

1.24 erlang inet_ssl_dist有BUG 导致节点通讯不能用SSL 66

1.25 erlang driver和热部署（重要） 67

1.26 Erlang heart 高可靠性的最后防线 68

1.27 erlang R12B预览版本出来了 69

1.28 why-do-you-like-erlang(转) 70

1.29 Joel Reymont号称要出erlang新书教你高级网络应用 73

1.30 erlang r12 新增加Percept并发性能调测模块 75

1.31 erlang函数调用新语法(用于代码hot replace) 76

1.32 R12B0的文档细致了很多 78

1.33 arbow发起 erlycachedb 项目 79

1.34 dets ram_file模式和 dets + ets的选择 80

1.35 how to call os:cmd("ls") from shell?（老掉牙问题） 81

1.36 erlang r12b1 改进内存分配效率,大幅提高smp的运行速度 87

1.37 erlsnoop erlang消息监听器 调试erlang网络程序利器 88

1.38 Parsing CSV in erlang(转) 90

1.39 file:read_file的注意细节 96

1.40 4月9号erlangR12B-2 release了 这次更新速度超快 100

1.41 erlang如何写port驱动与外面世界通讯 101

1.42 看erlang库实现 理解erlang函数编程 115

1.43 看例子写 testserver 和commontest 测试案例 116

1.44 Erlang/OTP R12B-3 released 117

1.45 gen_tcp 应对对端半关闭 118

1.46 ASN.1协议 适合我们用吗？ 119

1.47 Erlang Source via Git 终于可以即使获取代码了 120

1.48 CN Erlounge III - 发起 121

1.49 小小的参数设置耗费大量内存 122

1.50 R12B-5发布加入eunit 支持从archive读取代码 123

1.51 group_leader的设计和用途 124

1.52 What is the relation between async threads and SMP 126

1.53 Async线程pool及其作用 127

1.54 Rickard Green erlang smp的主要贡献者 129

1.55 Erlang分布的核心技术浅析 130

1.56 erlang内置的port相关的驱动程序 133

1.57 smp下async_driver的用途和匠心 134

1.58 erlang的IO高效不是传说 135

1.59 erlang的timer和实现机制 136

1.60 erlang的IO调度 138

1.61 erlang进程的优先级 141

1.62 最大文件句柄数对内存的消耗 143

1.63 erlang运行期的自省机制 145

1.64 Hunting Bugs 155

1.65 inside-beam-erlang-virtual-machine 158

1.66 erlang的process等同于lua的coroutine？ 160

1.67 erlang对port子进程退出的处理 161

1.68 erlang的进程调度器工作流程 172

1.69 erlang的atom实现 174

1.70 让erlang支持Huge TLB 175

1.71 叹！beam的hybrid还未支持 177

1.72 CN Erlounge III 候选讲师名单及议题 179

1.73 我的演讲主题：Inside the Erlang VM 180

1.74 erlang最小系统支持从远端加载beam 181

1.75 erlang进程退出时清扫的资源 183

1.76 The Top Ten Erlang News Stories of 2008 189

1.77 Erlang ERTS的Trap机制的设计及其用途 191

1.78 Tentative new functions in R13B 193

1.79 Erlang Programming for Multi-core 多核编程必读 194

1.80 Erlang Message Receive Fundamentals 195

1.81 webtool添加了好几个模块 196

1.82 R13开始支持binary unicode 197

1.83 A new Erlang book is on it's way 199

1.84 binary的这个bug R13A还没有修复 200

1.85 erlang shell commands 203
1.86 erlang节点间connect_all流程 205
1.87 Hidden Nodes的用处 207
1.88 Exit Reasons （备查） 208
1.89 dist_auto_connect的作用 209
1.90 Internal Representation of Records 210
1.91 .erlang文件 系统自动加载运行 211
1.92 5 tty - A command line interface 213
1.93 Erlang for Concurrent Programming by Jim Larson 215
1.94 erlang及其应用PPT 216
1.95 release_handler底层指令 热部署的实际执行者 217
1.96 程序语言的新星:走近Erlang的世界 220
1.97 借鉴erlang odbc架构写外部接口 225
1.98 R13A 新增Reltool模块 228
1.99 Lies, Damned Lies, and Benchmarks（R13A smp性能测试） 229
1.100 Dynamically sizing a fragmented mnesia store 232
1.101 Can one mnesia table fragment be further fragmente 238
1.102 Mnesia consumption 246
1.103 Mnesia upgrade policy (PA3) 252
1.104 What is the storage capacity of a Mnesia database? 263
1.105 How to Fix Erlang Crashes When Using Mnesia 264
1.106 On bulk loading data into Mnesia 268

1.107 scaling mnesia with local_content 272
1.108 Making a Mnesia Table SNMP Accessible 274
1.109 mnesia 分布协调的几个细节 280
1.110 Writing a Tsung plugin 285
1.111 tsung inside 289
1.112 erlang允许不同的节点有不同的cookie 293
1.113 读ejabberdctl学先进科技 295
1.114 另一种实用的接入erlang控制台的方法 314
1.115 xml解释器的选择 316
1.116 erlang编程的小技巧 (持续更新中...) 317
1.117 Unit Testing with Erlang’s Common Test Framework 326
1.118 Building a Non-blocking TCP server using OTP... 331
1.119 Linux下Erlang使用UnixODBC连接数据库 349
1.120 find out which line my Erlang program crash 351
1.121 抄书 Drivers in general 353
1.122 measure memory consumption in an Erlang system 356
1.123 User-Defined Behaviours 359
1.124 未公开的ram_file 内存文件 361
1.125 Garbage Collection in Erlang 363
1.126 per module constant pool 调查和使用 365
1.127 分析表达式警告的原因 369
1.128 ETS & SMP 373

1.129 变量不是不可变 375
1.130 新书 erlang programing 出炉 381
1.131 R13B SMP Migration logic 382
1.132 Next steps with SMP and Erlang 383
1.133 yacc&lex 的erlang版本 384
1.134 erlang的hipe支持(高级） 385
1.135 How fast can Erlang create processes? 387
1.136 How fast can Erlang send messages? 392
1.137 Adding my own BIF 397
1.138 进程字典到底有多快 399
1.139 ets 到底有多快 401
1.140 Debugging for the Erlang Programmer 404
1.141 Call for CN Erlounge IV ! 416
1.142 ets为什么要设计成 内容不参与GC 419
1.143 In a mnesia cluster, which node is queried? 421
1.144 avoiding overloading mnesia 423
1.145 How to Eliminate Mnesia Overload Events 426
1.146 erlang高级原理和应用PPT 428
1.147 系统标准库的hipe支持（高级） 429
1.148 研究Erlang 4000小时以后 433
1.149 erlang到底能够并发发起多少系统调用 434
1.150 转：CPU密集型计算 erlang和C 大比拼 438

1.151 答erlang静态数据查询方式 441

1.152 高强度的port(Pipe)的性能测試 445

1.1 erlang的dist研究

发表时间: 2007-07-25

erlang在dist上花了很多时间 dist.c->global_search->global_group->global->pg->pg2 2w行左右的c代码 上w的erl代码 才让系统在分步上有强壮的基础。erlang这个方面作的很优秀 但是也有一些硬伤 如auth global_group.

1.2 erlang distributed application

发表时间: 2007-07-25

application 的这个特性很好 就是文档里面提到的failover takeover 等 说白了就是application能够自动从失效的节点上迁移到可用的节点，最后再迁移回来。 源码里面的dist_ac为这个特性做的非常复杂。 dist_app + mnesia 就解决了大部分单点故障大的问题了。

1.3 减少linux swap对erlang程序的影响

发表时间: 2007-07-27 关键字: erlang swap swappingess

在作erlang压力测试的时候 我的机器内存是4G 在erlang程序用到2-3G内存的时候 swapd开始工作了 erlang程序的
但是这时候我实际上还有1G左右的物理内存。起先尝试用swapoff把swap关闭了 系统运行平滑 但是当物理内存用完的
挂了。这个不是很好。找了资料发现可以解决/proc/sys/vm/swappiness这个问题。 /proc/sys/vm/swappiness默
linux就不倾向使用swap 反之则用swap. 最后推荐设为10

1.4 erlang的beam模拟器

发表时间: 2007-07-27 关键字: beam smp hybrid

beam 模拟器有3种

1. beam 默认的
2. beam.smp 支持多处理器的
3. beam.hybrid 支持混合堆的

我们允许erl的时候 在linux下实际运行的是shell脚本

```
#!/bin/sh
ROOTDIR=/usr/local/lib/erlang
BINDIR=$ROOTDIR/erts-5.5.5/bin
EMU=beam
PROGNAME=`echo $0 | sed 's/.*\///'`
export EMU
export ROOTDIR
export BINDIR
export PROGNAME
exec $BINDIR/erlexec ${1+"$@"}
```

这个脚本给erlexec 设置写必须的环境变量 具体调用那个模拟器是在erlexec里面根据 参数区分 -smp -hybrid 来分别调用不同
的beam

Note: beam.smp主线程的的poll 是select 而不是我们想要的epoll, 是因为主线程的select实际上只是等待中断 没有其他的事情可做。

1.5 erlang dist protocol 格式

发表时间: 2007-07-31 关键字: erlang dist protocol

源码的 erts\emulator\internal_doc\erl_ext_dist.txt 描述了erlang ext term的格式, epmd通讯的流程协议和流程 同时还要node 间通讯的协议 相信对理解erlang的分布原理有帮助。

1.6 erts运行期内存使用情况查看

发表时间: 2007-07-31

翻erlang的代码发现erlang对memory的管理非常重视 内置了多种内存allocator:

- B: binary_alloc
- D: std_alloc
- E: ets_alloc
- F: fix_alloc
- H: eheap_alloc
- L: ll_alloc
- M: mseg_alloc
- S: sl_alloc
- T: temp_alloc
- Y: sys_alloc

多种分配策略：

1。 Best fit

2 。 Address order best fit

3. Good fit

4. A fit

还要一个erl_mtrace 模块专门收集erts允许期间的内存使用情况 然后通过tcp socket发送到 emem 这个程序中打印到 tty让人观察系统的情况。

emem在lib/tools/目录下 make install的时候默认没有安装。

启动 ./lib/tools/bin/i686-pc-linux-gnu/emem -p 1234

然后用erl -sname foo +Mit ip:1234就可以观察erlang的内存运作了.

注意erts的文档里面写：

+Mit X

<!----> Reserved for future use. Do **not** use this flag.

是属于未公开的 。

1.7 质疑 apache和yaws的性能比较(必看)

发表时间: 2007-07-31 关键字: apache yaws erlang web服务器 比较

http://www.sics.se/~joe/apachevsyaws.html 上比较yaws的性能 显示apache 4000就挂了 但是yaws 8w还在挺着。
见附件的图

但是你仔细看下他的测试方式

What do we measure and how?

We use a 16 node cluster running at SICS. We plot throughput vs. parallel load.

- Machine 1 has a server (Apache or Yaws).
- Machine 2 requests 20 KByte pages from machine 1. It does this in tight a loop requesting a new page as soon as it has received a page from the server. From this we derive a throughput figure, which is plotted in the horizontal scale on the graph. A typical value (800) means the throughput is 800 KBytes/sec.
- Machines 3 to 16 generate load.

  Each machine starts a large number of parallel sessions.

  Each session makes a very slow request to fetch a one byte file from machine 1. This is done by sending very slow HTTP GET requests (we break up the GET requests and send them character at a time, with about ten seconds between each character)

这个比较是非常不公平的

apache的链接处理机制是 开线程或者进程来处理请求 按它的测试方法 你非常慢速的8w请求 导致apache开大量的线程来处理。而能开多少线程取决于操作系统的能力 这还是其次 大量的线程处理活跃的链接导致大量的thread content switch。 apache 挂了不奇怪。 而erlang的线程相大于c语言的一个数据结构 erl_process你开多少取决于你的内存 大量的但是慢速的链接刚好适合poll事件dispatch， 以epoll的能力（俺测试过epoll30w）能够轻松处理。 这个测试与其说测试web服务器的性能 不如说 测试服务器的进程生成能力。

俺的测试是这样的：.
./yaws --conf yaws.conf --erlarg "+K true +P 1024000" #epoll 最多1024000个进程 内核都已经调优过

yaws.conf 的内容：

auth_log = false

max_num_cached_files = 8000
max_num_cached_bytes = 6000000

<server test_yaws=""></server>

大家都用 ab -c 1000 -n 1000000 http://192.168.0.98:8000/bomb.gif 来测
果然发现yaws的性能也是非常一般 大概也就是3K左右.

各位看下 strace 的结果就知道了：

accept(10, {sa_family=AF_INET, sin_port=htons(5644), sin_addr=inet_addr("192.168.0.97")}, [16]) = 11
fcntl64(11, F_GETFL) = 0x2 (flags O_RDWR)
fcntl64(11, F_SETFL, O_RDWR|O_NONBLOCK) = 0
getsockopt(10, SOL_TCP, TCP_NODELAY, [0], [4]) = 0
getsockopt(10, SOL_SOCKET, SO_KEEPALIVE, [0], [4]) = 0
getsockopt(10, SOL_SOCKET, SO_PRIORITY, [0], [4]) = 0
getsockopt(10, SOL_IP, IP_TOS, [0], [4]) = 0
getsockopt(11, SOL_SOCKET, SO_PRIORITY, [0], [4]) = 0
getsockopt(11, SOL_IP, IP_TOS, [0], [4]) = 0
setsockopt(11, SOL_IP, IP_TOS, [0], 4) = 0
setsockopt(11, SOL_SOCKET, SO_PRIORITY, [0], 4) = 0
getsockopt(11, SOL_SOCKET, SO_PRIORITY, [0], [4]) = 0
getsockopt(11, SOL_IP, IP_TOS, [0], [4]) = 0
setsockopt(11, SOL_SOCKET, SO_PRIORITY, [0], 4) = 0
getsockopt(11, SOL_SOCKET, SO_PRIORITY, [0], [4]) = 0
getsockopt(11, SOL_IP, IP_TOS, [0], [4]) = 0
setsockopt(11, SOL_SOCKET, SO_KEEPALIVE, [0], 4) = 0
setsockopt(11, SOL_IP, IP_TOS, [0], 4) = 0
setsockopt(11, SOL_SOCKET, SO_PRIORITY, [0], 4) = 0
getsockopt(11, SOL_SOCKET, SO_PRIORITY, [0], [4]) = 0
getsockopt(11, SOL_IP, IP_TOS, [0], [4]) = 0
setsockopt(11, SOL_TCP, TCP_NODELAY, [0], 4) = 0
setsockopt(11, SOL_SOCKET, SO_PRIORITY, [0], 4) = 0
recv(11, "GET /bomb.gif HTTP/1.0\r\nUser-Age"..., 8192, 0) = 100
getpeername(11, {sa_family=AF_INET, sin_port=htons(5644), sin_addr=inet_addr("192.168.0.97")}, [16]) = 0
clock_gettime(CLOCK_MONOTONIC, {110242, 326908594}) = 0
stat64("/var/www/html/bomb.gif", {st_mode=S_IFREG|0644, st_size=4096, ...}) = 0

```
access("/var/www/html/bomb.gif", R_OK)  = 0
access("/var/www/html/bomb.gif", W_OK)  = 0
clock_gettime(CLOCK_MONOTONIC, {110242, 327135982}) = 0
time(NULL)                              = 1185894828
clock_gettime(CLOCK_MONOTONIC, {110242, 327222643}) = 0
stat64("/etc/localtime", {st_mode=S_IFREG|0644, st_size=405, ...}) = 0
writev(11, [{NULL, 0}, {"HTTP/1.1 200 OK\r\nConnection: clo"..., 231},
{"\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...
, 4096}], 3) = 4327
close(11
```

这里面充斥着大量的无用的昂贵的系统调用（至少有20个*10us = 200us 的系统调用是无效的）

对文件的access 2 次 连文件的cache都没有 每次 打开文件 读文件 然后写到socket去 。

这个case是小文件(4k)的情况。 看下大文件(40k)的情况

```
open("/var/www/html/bomb.gif", O_RDONLY|O_LARGEFILE) = 19
read(19, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 10240) = 10240
writev(16, [{NULL, 0}, {"\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 10240}], 2) =
10240
read(19, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 10240) = 10240
writev(16, [{NULL, 0}, {"\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 10240}], 2) =
10240
read(19, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 10240) = 10240
writev(16, [{NULL, 0}, {"\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 10240}], 2) =
10240
read(19, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 10240) = 10240
writev(16, [{NULL, 0}, {"\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 10240}], 2) = 7240
read(19, "", 10240)                     = 0
close(19)                               = 0
clock_gettime(CLOCK_MONOTONIC, {110574, 856508319}) = 0
epoll_ctl(3, EPOLL_CTL_DEL, 11, {0, {u32=11, u64=581990243524149259}}) = 0
epoll_ctl(3, EPOLL_CTL_DEL, 12, {0, {u32=12, u64=581990243524149260}}) = 0
epoll_ctl(3, EPOLL_CTL_ADD, 16, {EPOLLOUT, {u32=16, u64=581990243524149264}}) = 0
epoll_wait(3, {}, 256, 0)               = 0
clock_gettime(CLOCK_MONOTONIC, {110574, 856677411}) = 0
clock_gettime(CLOCK_MONOTONIC, {110574, 856729274}) = 0
```

大量的epoll_ctl 调用 clock_gettime的调用 足够让系统的速度变的非常慢。

比对下lighttpd的性能。 lighttpd用到了cache，用到了aio，还是完全用c语言小心编写， 他处理小文件大概是并发1w. 而yaws这个的处理方式打个3折我看差不多。

所以请各位大佬介绍erlang的性能时候不要 再用这个apache vs yaws的例子了 误导太多人了.

1.8 erl_call erlang cnode 功能强大

发表时间: 2007-08-13 关键字: erlang cnode erl_call

otp_src_R11B-5\lib\erl_interface\src\prog\erl_call.c 是个不错的工具, 就是ei的前端能够通过cnode给erlang的后端发各种请求。

具体的见 主题: 如何把erlang应用在项目中？ http://www.javaeye.com/topic/100425

```
where: -a  apply(Mod,Fun,Args) (e.g -a 'erlang length [[a,b,c]]'
       -c  cookie string; by default read from ~/.erlang.cookie
       -d  direct Erlang output to ~/.erl_call.out.<nodename>
       -e  evaluate contents of standard input (e.g echo "X=1,Y=2,{X,Y}."|erl_call -e ...)
       -h  specify a name for the erl_call client node
       -m  read and compile Erlang module from stdin
       -n  name of Erlang node, same as -name
       -name  name of Erlang node, expanded to a fully qualified
       -sname name of Erlang node, short form will be used
       -q  halt the Erlang node (overrides the -s switch)
       -r  use a random name for the erl_call client node
       -s  start a new Erlang node if necessary
       -v  verbose mode, i.e print some information on stderr
       -x  use specified erl start script, default is erl
```

使用方法 :

1. erl -name xx@192.168.0.98 启动后端

2. export EI_TRACELEVEL=6

i. erl_call -v -d -n xx@erl98.3322.org -m 模块方式 （ -v -d 调试和verbose模式 ）

如 :

```
-module(a).
```

CTRL+D

II. erl_call -v -d -n xx@erl98.3322.org -e 表达式方式

1+2.

CTRL+D

III. erl_call -v -d -n xx@erl98.3322.org -a 'erlang length [[a,b,c]]'

这2中都是从stdin读 直到你按下CTRL+D, 然后你就可以看到结果。

这只程序内存有泄漏
* Note: We don't free any memory at all since we only
* live for a short while.
而且 erl_call.c有bug 812行 free(mbuf); /* Allocated in read_stdin() */
注释掉这行就可以了

这个程序默认是不安装带标准发布目录去的。

从这个程序 我们可以知道cnode 能作的事情受限于你的想像力。

注：
ping erl98.3322.org
PING erl98.3322.org (192.168.0.98) 56(84) bytes of data.
由于erl_call程序有点小问题 -n xx@erl98.3322.org 最好用域名 否者erl_call就抓狂了。</nodename>

1.9 erlang inet:setopts 未公开选项

发表时间: 2007-08-14 关键字: erlang setopts

inet:setopt有packet设置选项：

`{packet, PacketType}` **(TCP/IP sockets)**

Defines the type of packets to use for a socket. The following values are valid:

`raw | 0`

No packaging is done.

`1 | 2 | 4`

Packets consist of a header specifying the number of bytes in the packet, followed by that number of bytes. The length of header can be one, two, or four bytes; the order of the bytes is big-endian. Each send operation will generate the header, and the header will be stripped off on each receive operation.

`asn1 | cdr | sunrm | fcgi | tpkt | line`

These packet types only have effect on receiving. When sending a packet, it is the responsibility of the application to supply a correct header. On receiving, however, there will be one message sent to the controlling process for each complete packet received, and, similarly, each call to `gen_tcp:recv/2,3` returns one complete packet. The header is **not** stripped off.

The meanings of the packet types are as follows:

`asn1` - ASN.1 BER,

`sunrm` - Sun's RPC encoding,

`cdr` - CORBA (GIOP 1.1),

`fcgi` - Fast CGI,

`tpkt` - TPKT format [RFC1006],

`line` - Line mode, a packet is a line terminated with newline, lines longer than the receive buffer are truncated.

文档中写的就这么多了 其实还有2个选项：http， httph 用于解释http的请求

实现在otp_src_R11B-5\erts\emulator\drivers\common\inet_drv.c 里面

#ifdef USE_HTTP

。。。

#endif

/*

**** load http message:**

**** {http_eoh, S} - end of headers**

** {http_header, S, Key, Value} - Key = atom() | string()
** {http_request, S, Method,Url,Version}
** {http_response, S, Version, Status, Message}
** {http_error, S, Error-Line}
*/
消息以上面的方式发给process，用于gen_tcp

由于在driver层面实现的 所以效率就很高，对于编写http隧道之类的程序 很有帮助哦

1.10 如何看erts内部的状态

发表时间: 2007-08-15 关键字: erlang erts debug get_internal_state

经常在性能优化的时候 要看下erts内部的允许状态 erlang有未公开的函数

erts_debug:get_internal_state(XX)

XX为atom有以下几个

```
DECL_AM(node_and_dist_references);
DECL_AM(DbTable_words);
DECL_AM(next_pid);
DECL_AM(next_port);
DECL_AM(check_io_debug);
DECL_AM(available_internal_state);
DECL_AM(monitoring_nodes);
```

XX为list有以下几个

```
DECL_AM(link_list);
DECL_AM(monitor_list);
DECL_AM(channel_number);
DECL_AM(have_pending_exit);
```

可以看的很细节的运行期数据.

前提是先用

erts_debug:set_internal_state(available_internal_state, true).

否者调用get_internal_state会提示失败.

1.11 erl CTRL+C do_break 的功能

发表时间: 2007-08-16

在erl shell下按下CTRL+C的时候

erts_printf("\n" "BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded\n" " (v)ersion (k)ill (D)b-tables (d)istribution\n");

但是实际上可以有更多功能 看代码：

```
while (1) {
  if ((i = sys_get_key(0)) <= 0)
    erl_exit(0, "");
  switch (i) {
  case 'q':
  case 'a':
  case '*': /*
       * The asterisk is an read error on windows,
       * where sys_get_key isn't that great in console mode.
       * The usual reason for a read error is Ctrl-C. Treat this as
       * 'a' to avoid infinite loop.
       */
    erl_exit(0, "");
  case 'A':       /* Halt generating crash dump */
    erl_exit(1, "Crash dump requested by user");
  case 'c':
    return;
  case 'p':
    process_info(ERTS_PRINT_STDOUT, NULL);
    return;
  case 'm':
    return;
  case 'o':
    port_info(ERTS_PRINT_STDOUT, NULL);
    return;
  case 'i':
    info(ERTS_PRINT_STDOUT, NULL);
    return;
  case 'l':
```

```
        loaded(ERTS_PRINT_STDOUT, NULL);
        return;
    case 'v':
        erts_printf("Erlang (%s) emulator version "
                ERLANG_VERSION "\n",
                EMULATOR);
        erts_printf("Compiled on " ERLANG_COMPILE_DATE "\n");
        return;
    case 'd':
        distribution_info(ERTS_PRINT_STDOUT, NULL);
        return;
    case 'D':
        db_info(ERTS_PRINT_STDOUT, NULL, 1);
        return;
    case 'k':
        process_killer();
        return;
#ifdef OPPROF
    case 'X':
        dump_frequencies();
        return;
    case 'x':
        {
        int i;
        for (i = 0; i <= HIGHEST_OP; i++) {
            if (opc[i].name != NULL) {
            erts_printf("%-16s %8d\n", opc[i].name, opc[i].count);
            }
        }
        }
        return;
    case 'z':
        {
        int i;
        for (i = 0; i <= HIGHEST_OP; i++)
            opc[i].count = 0;
        }
        return;
```

```
#endif
#ifdef DEBUG
    case 't':
        p_slpq();
        return;
    case 'b':
        bin_check();
        return;
    case 'C':
        abort();
#endif
    case '\n':
        continue;
    default:
        erts_printf("Eh?\n\n");
    }
    }
```

好多调试用的功能 希望对大家有用。

1.12 方便调试erlang程序的脚本

发表时间: 2007-08-18 关键字: erlang debug ddd beam

经常的时候看大型工程的时候 碰到一二个地方实在不明白他是如何运作的 这时候最好的工具就是debugger 如gdb，的backtrace 可以得到完整的函数调用栈。在linux下推荐使用ddd, 俺的centos5 下标准版本没有安装ddd 顺手下载个安装就好了（标准版本却个motif-devel yum下就好）。ddd图形界面方便查看函数和变量，还有点击跳转功能。 附上几个调试erlang的脚本，希望能够方便大家。

1.
[root@test98 ~]# cat gdb_beam
#! /bin/bash

ddd -x gdb.init /usr/local/lib/erlang/erts-5.5.5/bin/beam

2.
[root@test98 ~]# cat gdb.init
set arg -- -root /usr/local/lib/erlang -progname erl -- -home /root

3.
[root@test98 ~]# tail .bash_profile -n 13

export PATH=$PATH:/usr/local/lib/erlang/erts-5.5.5/bin
ROOTDIR=/usr/local/lib/erlang
BINDIR=$ROOTDIR/erts-5.5.5/bin
EMU=beam
PROGNAME=`echo $0 | sed 's/.*\///'`
export EMU
export ROOTDIR
export BINDIR
export PROGNAME
export EDITOR=vim

export LANG=utf8

上面的脚本是针对beam 的。

如果你要调试beam.smp beam.hybrid 可以erl -smp true +K true -emu_args 得到参数

Executing: /usr/local/lib/erlang/erts-5.5.5/bin/beam.smp /usr/local/lib/erlang/erts-5.5.5/bin/beam.smp -K true -- -root /usr/local/lib/erlang -progname erl -- -home /root -smp true

把以上脚本改下就方便多了。

1.13 erlang 模拟器工作原理分析

发表时间: 2007-10-25 关键字: emulator implement erts

这个是erlang聚会的时候作的一个ppt 简单介绍了emulator的组成和运作原理，凑合着看把，详细的等时间沉淀再写。

附件下载:

- emulator_implementation.rar (42.5 KB)
- dl.javaeye.com/topics/download/934aa7ca-cc14-4ea4-a351-87044a3aff9d

1.14 erlang的erl_crash.dump产生以及如何解读

发表时间: 2007-10-25 关键字: erl_crash core dump

正常情况下 当erlang进程发生错误没有catch的时候 emulator就会自动产生erl_crash.dump，来提供crash的时候的emulator最详细的情况，类似于unix的core dump. 其中下边几个env变量控制dump产生的行为:

`ERL_CRASH_DUMP`

If the emulator needs to write a crash dump, the value of this variable will be the file name of the crash dump file. If the variable is not set, the name of the crash dump file will be `erl_crash.dump` in the current directory.

`ERL_CRASH_DUMP_NICE`

Unix systems: If the emulator needs to write a crash dump, it will use the value of this variable to set the nice value for the process, thus lowering its priority. The allowable range is 1 through 39 (higher values will be replaced with 39). The highest value, 39, will give the process the lowest priority.

`ERL_CRASH_DUMP_SECONDS`

Unix systems: This variable gives the number of seconds that the emulator will be allowed to spend writing a crash dump. When the given number of seconds have elapsed, the emulator will be terminated by a SIGALRM signal.

除了被动产生dump以外, 用户还可以主动产生dump 方法有2种：

1. erl控制台 CTRL+C 然后+A

2. kill -s SIGUSR1 erlpid

产生的erl_crash.dump是个纯文本， 可能非常大，特别是你有成千上万的process和port什么的，对于系统调优有非常大的意义。

察看方式参见文档 erl5.5.5/erts-5.5.5/doc/html/crash_dump.html 请注意内存使用有的是byte 有的是WORD.

更方便的是用工具 webtool 来察看 web界面 比较直观。

1.15 erlang数据库 ETS 工作原理分析

发表时间: 2007-08-18

ETS 是erlang term strorage 的意思 文档见erl5.5.5/lib/stdlib-1.14.5/doc/html/index.html。 这个是beam里面很核心的一个功能。ets, dets, mnesia 组成了erlang的数据库，注意mnesia本身没有存储机制 它的存储就是ets 和dets。

用ets:i().看下可以知道

```
11          code            set   254    11393    code_server
 12          code_names      set   48     5323     code_server
 13          shell_records   ordered_set 0      72       <0.25.0>
 ac_tab      ac_tab          set   6      853      application_controller
 file_io_servers file_io_servers   set   0      279      file_server_2
 global_locks    global_locks      set   0      279      global_name_server
 global_names    global_names      set   0      279      global_name_server
 global_names_ext global_names_ext  set   0      279      global_name_server
 global_pid_ids  global_pid_ids    bag   0      279      global_name_server
 global_pid_names global_pid_names  bag   0      279      global_name_server
 inet_cache      inet_cache        bag   0      279      inet_db
 inet_db         inet_db           set   21     528      inet_db
 inet_hosts      inet_hosts        set   1      310      inet_db
```

也就是说erlang的kernel 和stdlib库的实现都很依赖于这个ets.

文档里面一句话： This module is an interface to the Erlang built-in term storage BIFs. ets.erl本身只是一个封装的模块 用于检查参数等等 实际的工作都是bif作的,所以效率非常好。
看下otp_src_R11B-5\erts\emulator\beam\bif.tab

```
#
# Bifs in ets module.
#

bif ets:all/0
bif 'erl.lang.ets':all/0        ebif_ets_all_0
bif ets:new/2
bif 'erl.lang.ets':new/2        ebif_ets_new_2
...
```

bif 'erl.lang.ets':match/1 ebif_ets_match_1
bif ets:match/2
bif 'erl.lang.ets':match/2 ebif_ets_match_2
bif ets:match/3

在emulator里面和ets实现有关的 有erl_db.c（界面） erl_db_hash.c（hash实现） erl_db_tree.c（tree实现） erl_db_util.c(match虚拟机等）总代码有 将近有20,000行实现是很复杂的，据说下一版本会用jarray的算法来做效率更高。

ets的实现不是多线程安全的，数据不参加GC, 使用的时候要注意。

当我们要遍历ets的时候 可以用first/next来遍历 也可以用foldr foldl来看ets看成list来使用。但是这样使用的时候有效率问题 数据要从erts内部搬到process 当ets很大的时候就效率低。

这时候ets:select match MatchSpec来帮你了. ets内部实现了一个虚拟机把matchspec编译成opcode 然后eval的时候把需要的数据才拷贝到process去 大大减少了数据量. 这个方法类似于sqlite。

见db_match_set_compile 编译matchspec成opcode
db_prog_match 运算opcode 细节可以看下代码。

这还不够 ets 考虑到matchspec比较难写 又提供了一个功能 fun2ms 可以把标准的erlang fun转换成matchspec.请参考ms_transform.

有了这些功能的辅助 ets使用起来就很方便了。

1.16 Erlang: 当你net_adm:ping(Node)的时候发生了什么？流程复杂 但是对理解dist工作原理至关重要

发表时间: 2007-09-05 关键字: net_adm trap dist erlang send

当你net_adm:ping(Node)的时候发生了什么？ 这个涉及到很复杂的流程。让我为你解刨：
这个流程很长而且在erlang代码和c代码里面窜来窜去，重要的点 我用红字标注 请各位耐心。

1. net_adm.erl:

```
ping(Node) when is_atom(Node) ->
    case catch gen:call({net_kernel, Node},
            '$gen_call',
            {is_auth, node()},
            infinity) of
    {ok, yes} -> pong;
    _ ->
        erlang:disconnect_node(Node),
        pang
    end.
```

2. gen.erl:

```
%% Remote by name
call({_Name, Node}=Process, Label, Request, Timeout)
  when is_atom(Node), Timeout =:= infinity;
       is_atom(Node), is_integer(Timeout), Timeout >= 0 ->
    if
     node() =:= nonode@nohost ->
         exit({nodedown, Node});
     true ->
         do_call(Process, Label, Request, Timeout)
    end.

do_call(Process, Label, Request, Timeout) ->
    %% We trust the arguments to be correct, i.e
    %% Process is either a local or remote pid,
    %% or a {Name, Node} tuple (of atoms) and in this
    %% case this node (node()) _is_ distributed and Node =/= node().
    Node = case Process of
```

```
        {_S, N} ->
        N;
        _ when is_pid(Process) ->
        node(Process);
        _ ->
        node()
    end,
case catch erlang:monitor(process, Process) of
Mref when is_reference(Mref) ->
    receive
    {'DOWN', Mref, _, Pid1, noconnection} when is_pid(Pid1) ->
        exit({nodedown, node(Pid1)});
    {'DOWN', Mref, _, _, noconnection} ->
        exit({nodedown, Node});
    {'DOWN', Mref, _, _, _} ->
        exit(noproc)
    after 0 ->
        Process ! {Label, {self(), Mref}, Request},
        wait_resp_mon(Process, Mref, Timeout)
    end;
{'EXIT', _} ->
    %% Old node is not supporting the monitor.
    %% The other possible case -- this node is not distributed
    %% -- should have been handled earlier.
    %% Do the best possible with monitor_node/2.
    %% This code may hang indefinitely if the Process
    %% does not exist. It is only used for old remote nodes.
    monitor_node(Node, true),
    receive
    {nodedown, Node} ->
        monitor_node(Node, false),
        exit({nodedown, Node})
    after 0 ->
        Mref = make_ref(),
        Process ! {Label, {self(),Mref}, Request},
        Res = wait_resp(Node, Mref, Timeout),
        monitor_node(Node, false),
        Res
```

```
        end
    end.
```

3 . Process ! {Label, {self(),Mref}, Request}, 相当于erlang:send(Process, {Label, {self(),Mref}, Request});

4. bif.tab

```
 bif 'erl.lang.proc':send/2    ebif_send_2 send_2 bif erlang:send/2
```

5. bif.c

```
Eterm send_2(Process *p, Eterm to, Eterm msg) {
    Sint result = do_send(p, to, msg, !0);

    if (result > 0) {
    BUMP_REDS(p, result);
    BIF_RET(msg);
    } else switch (result) {
    case 0:
    BIF_RET(msg);
    break;
    case SEND_TRAP:
    BIF_TRAP2(dsend2_trap, p, to, msg);
    break;
    case SEND_RESCHEDULE:
    BIF_ERROR(p, RESCHEDULE);
    break;
    case SEND_BADARG:
    BIF_ERROR(p, BADARG);
    break;
    case SEND_USER_ERROR:
    BIF_ERROR(p, EXC_ERROR);
    break;
    default:
    ASSERT(! "Illegal send result");
    break;
    }
    ASSERT(! "Can not arrive here");
    BIF_ERROR(p, BADARG);
}
```

6. bif.c

```
 Sint
do_send(Process *p, Eterm to, Eterm msg, int suspend) {
    Eterm portid;
    Port *pt;
    Process* rp;
    DistEntry *dep;
    Eterm* tp;

    if (is_internal_pid(to)) {
    if (IS_TRACED(p))
        trace_send(p, to, msg);
    if (p->ct != NULL)
       save_calls(p, &exp_send);

    if (internal_pid_index(to) >= erts_max_processes)
        return SEND_BADARG;
    rp = erts_pid2proc(p, ERTS_PROC_LOCK_MAIN, to, ERTS_PROC_LOCKS_MSG_SEND);

    if (!rp) {
        ERTS_SMP_ASSERT_IS_NOT_EXITING(p);
        return 0;
    }
    } else if (is_external_pid(to)) {
    Sint res;
    dep = external_pid_dist_entry(to);
    if(dep == erts_this_dist_entry) {
        erts_dsprintf_buf_t *dsbufp = erts_create_logger_dsbuf();
        erts_dsprintf(dsbufp,
            "Discarding message %T from %T to %T in an old "
            "incarnation (%d) of this node (%d)\n",
            msg,
            p->id,
            to,
            external_pid_creation(to),
            erts_this_node->creation);
        erts_send_error_to_logger(p->group_leader, dsbufp);
```

```
    return 0;
}

erts_dist_op_prepare(dep, p, ERTS_PROC_LOCK_MAIN);

/* Send to remote process */
if (is_nil(dep->cid))
    res = SEND_TRAP;
else if (dist_send(p, ERTS_PROC_LOCK_MAIN, dep, to, msg) == 1) {

    if (is_internal_port(dep->cid)) {
    if (suspend) {
        erts_suspend(p, ERTS_PROC_LOCK_MAIN, dep->port);
        if (erts_system_monitor_flags.busy_dist_port) {
        monitor_generic(p, am_busy_dist_port, dep->cid);
        }
    }
    res = SEND_RESCHEDULE;
    }
    else {
    res = SEND_TRAP;
    }
}
else {
    res = 50;
    if (IS_TRACED(p))
    trace_send(p, to, msg);
    if (p->ct != NULL)
    save_calls(p, &exp_send);
}

erts_dist_op_finalize(dep);

return res;
} else if (is_atom(to)) {
erts_whereis_name(p, ERTS_PROC_LOCK_MAIN,
          to,
          &rp, ERTS_PROC_LOCKS_MSG_SEND, 0,
```

```
        &pt);

if (pt) {
    portid = pt->id;
    goto port_common;
}

if (IS_TRACED(p))
    trace_send(p, to, msg);
if (p->ct != NULL)
   save_calls(p, &exp_send);

if (!rp) {
    return SEND_BADARG;
}
} else if (is_external_port(to)
       && (external_port_dist_entry(to)
       == erts_this_dist_entry)) {
erts_dsprintf_buf_t *dsbufp = erts_create_logger_dsbuf();
erts_dsprintf(dsbufp,
          "Discarding message %T from %T to %T in an old "
          "incarnation (%d) of this node (%d)\n",
          msg,
          p->id,
          to,
          external_port_creation(to),
          erts_this_node->creation);
erts_send_error_to_logger(p->group_leader, dsbufp);
return 0;
} else if (is_internal_port(to)) {
portid = to;
pt = erts_id2port(to, p, ERTS_PROC_LOCK_MAIN);
  port_common:
ERTS_SMP_LC_ASSERT(!pt || erts_lc_is_port_locked(pt));
/* XXX let port_command handle the busy stuff !!! */
if (pt && (pt->status & ERTS_PORT_S_PORT_BUSY)) {
    if (suspend) {
    erts_suspend(p, ERTS_PROC_LOCK_MAIN, pt);
```

```
    if (erts_system_monitor_flags.busy_port) {
        monitor_generic(p, am_busy_port, portid);
    }
    }
    erts_port_release(pt);
    return SEND_RESCHEDULE;
}

if (IS_TRACED(p))     /* trace once only !! */
    trace_send(p, portid, msg);
if (p->ct != NULL)
   save_calls(p, &exp_send);

if (SEQ_TRACE_TOKEN(p) != NIL) {
    seq_trace_update_send(p);
    seq_trace_output(SEQ_TRACE_TOKEN(p), msg,
            SEQ_TRACE_SEND, portid, p);
}

/* XXX NO GC in port command */
erts_port_command(p, p->id, pt, msg);
if (pt)
    erts_port_release(pt);
if (ERTS_PROC_IS_EXITING(p)) {
    KILL_CATCHES(p); /* Must exit */
    return SEND_USER_ERROR;
}
return 0;
} else if (is_tuple(to)) { /* Remote send */
int ret;
tp = tuple_val(to);
if (*tp != make_arityval(2))
    return SEND_BADARG;
if (is_not_atom(tp[1]) || is_not_atom(tp[2]))
    return SEND_BADARG;

/* sysname_to_connected_dist_entry will return NULL if there
   is no dist_entry or the dist_entry has no port*/
```

```
if ((dep = erts_sysname_to_connected_dist_entry(tp[2])) == NULL) {
    return SEND_TRAP;
}

if (dep == erts_this_dist_entry) {
    erts_deref_dist_entry(dep);
    if (IS_TRACED(p))
    trace_send(p, to, msg);
    if (p->ct != NULL)
      save_calls(p, &exp_send);

    erts_whereis_name(p, ERTS_PROC_LOCK_MAIN,
            tp[1],
            &rp, ERTS_PROC_LOCKS_MSG_SEND, 0,
            &pt);
    if (pt) {
    portid = pt->id;
    goto port_common;
    }

    if (!rp) {
    return 0;
    }
    goto send_message;
}

erts_dist_op_prepare(dep, p, ERTS_PROC_LOCK_MAIN);
if (is_nil(dep->cid))
    ret = SEND_TRAP;
else if (dist_reg_send(p, ERTS_PROC_LOCK_MAIN, dep, tp[1], msg) == 1) {
    if (is_internal_port(dep->cid)) {
    if (suspend) {
        erts_suspend(p, ERTS_PROC_LOCK_MAIN, dep->port);
        if (erts_system_monitor_flags.busy_dist_port) {
        monitor_generic(p, am_busy_dist_port, dep->cid);
        }
    }
    ret = SEND_RESCHEDULE;
```

```
        }
        else {
        ret = SEND_TRAP;
        }

    }
    else {
        ret = 0;
        if (IS_TRACED(p))
        trace_send(p, to, msg);
        if (p->ct != NULL)
        save_calls(p, &exp_send);
    }

    erts_dist_op_finalize(dep);
    erts_deref_dist_entry(dep);
    return ret;
    } else {
    if (IS_TRACED(p)) /* XXX Is this really neccessary ??? */
        trace_send(p, to, msg);
    if (p->ct != NULL)
       save_calls(p, &exp_send);
    return SEND_BADARG;
    }

 send_message: {
    Uint32 rp_locks = ERTS_PROC_LOCKS_MSG_SEND;
    Sint res;
#ifdef ERTS_SMP
    if (p == rp)
        rp_locks |= ERTS_PROC_LOCK_MAIN;
#endif
    /* send to local process */
    erts_send_message(p, rp, &rp_locks, msg, 0);
#ifdef ERTS_SMP
    res = rp->msg_inq.len*4;
    if (ERTS_PROC_LOCK_MAIN & rp_locks)
        res += rp->msg.len*4;
```

```
#else
    res = rp->msg.len*4;
#endif
    erts_smp_proc_unlock(rp,
                         p == rp
                         ? (rp_locks & ~ERTS_PROC_LOCK_MAIN)
                         : rp_locks);
    return res;
    }
}
```

8. 如果能找到节点的话 就调用dist_send 否者发生SEND_TRAP

9. dist.c

```
void init_dist(void)
{
    init_alive();
    init_nodes_monitors();

    no_caches = 0;

    /* Lookup/Install all references to trap functions */
    dsend2_trap = trap_function(am_dsend,2);
    dsend3_trap = trap_function(am_dsend,3);
    /*    dsend_nosuspend_trap = trap_function(am_dsend_nosuspend,2);*/
    dlink_trap = trap_function(am_dlink,1);
    dunlink_trap = trap_function(am_dunlink,1);
    dmonitor_node_trap = trap_function(am_dmonitor_node,3);
    dgroup_leader_trap = trap_function(am_dgroup_leader,2);
    dexit_trap = trap_function(am_dexit, 2);
    dmonitor_p_trap = trap_function(am_dmonitor_p, 2);
}

static Export*
trap_function(Eterm func, int arity)
{
```

```
    return erts_export_put(am_erlang, func, arity);
}
```

也就是说dsend2_trap 就是erlang:dsend这个函数

当send 失败 的时候 参考5 的 send_2 将执行 BIF_TRAP2(dsend2_trap, p, to, msg);

10。 bif.h

```
#define BIF_TRAP2(Trap_, p, A0, A1) do {    \
     (p)->arity = 2;                \
     (p)->def_arg_reg[0] = (A0);        \
     (p)->def_arg_reg[1] = (A1);        \
     (p)->def_arg_reg[3] = (Eterm) (Trap_);    \
     (p)->freason = TRAP;            \
     return THE_NON_VALUE;            \
} while(0)
```

11. beam_emu.c

```
 OpCase(call_bif2_e):
    {
    Eterm (*bf)(Process*, Eterm, Eterm, Uint*) = GET_BIF_ADDRESS(Arg(0));
    Eterm result;
    Eterm* next;

    SWAPOUT;
    c_p->fcalls = FCALLS - 1;
    if (FCALLS <= 0) {
       save_calls(c_p, (Export *) Arg(0));
    }
    PreFetch(1, next);
    CHECK_TERM(r(0));
    CHECK_TERM(x(1));
    PROCESS_MAIN_CHK_LOCKS(c_p);
    ASSERT(!ERTS_PROC_IS_EXITING(c_p));
    result = (*bf)(c_p, r(0), x(1), I);
    ASSERT(!ERTS_PROC_IS_EXITING(c_p) || is_non_value(result));
    PROCESS_MAIN_CHK_LOCKS(c_p);
    ERTS_HOLE_CHECK(c_p);
```

```
    POST_BIF_GC_SWAPIN(c_p, result);
    FCALLS = c_p->fcalls;
    if (is_value(result)) {
        r(0) = result;
        CHECK_TERM(r(0));
        NextPF(1, next);
    } else if (c_p->freason == RESCHEDULE) {
        c_p->arity = 2;
        goto suspend_bif;
    } else if (c_p->freason == TRAP) {
        goto call_bif_trap3;
    }

    call_bif_trap3:
        SET_CP(c_p, I+2);
        SET_I(((Export *)(c_p->def_arg_reg[3]))->address);
        SWAPIN;
        r(0) = c_p->def_arg_reg[0];
        x(1) = c_p->def_arg_reg[1];
        x(2) = c_p->def_arg_reg[2];
        Dispatch();
```

也就是说这时候trap erlang:dsend函数被调用。

12. erlang.erl

```
dsend({Name, Node}, Msg, Opts) ->
    case net_kernel:connect(Node) of
    true -> erlang:send({Name,Node}, Msg, Opts);
    false -> ok;
    ignored -> ok               % Not distributed.
    end.
```

先链接对方节点 成功 发数据包 转8 实际上是调用dist_send.
整个流程完毕。

这里面trap技术用的很巧妙。

net_kernel:connect 也很复杂 另文分析，请期待。

1.17 Erlang 进程堆内存增长算法

发表时间: 2007-09-27 关键字: memory fib erlang heap process

```
/*
    * Heap sizes start growing in a Fibonacci sequence.
    *
    * Fib growth is not really ok for really large heaps, for
    * example is fib(35) == 14meg, whereas fib(36) == 24meg;
    * we really don't want that growth when the heaps are that big.
    */
```

erlang的process heap堆大小默认是233 等于fib(11), 用fib算法的目的是后续的heap增长会比较慢 避免内存的浪费。

大多数的系统都是以上为2倍数增长的。有人对std:string做过统计分析，最后得出一个结论是 1.5 是比较合适的数字。

erlang用这个算法个人感觉体现这个系统的成熟。

1.18 erlang中类似netstat的命令

发表时间: 2007-10-17 关键字: inet erlang netstat module i

inet:i() 这个函数可以列出网络的连接情况 很实用。

```
(y@erl98.3322.org)46> inet:i().
Port Module   Recv  Sent   Owner    Local Address      Foreign Address     State
7    inet_tcp 0     0      <0.20.0> *:13660            *:*                 ACCEPTING
9    inet_tcp 4     16     <0.18.0> localhost:48084    localhost:epmd      CONNECTED
97   inet_tcp 62399 182817 <0.38.0> 192.168.0.98:13660 192.168.0.243:19342 CONNECTED
Port Module Recv Sent Owner Local Address Foreign Address State
ok
```

其实erlang的好多module都有i 函数都是 取得该模块的信息的 很方便。

1.19 erlang热部署

发表时间: 2007-10-17 关键字: erlang hot deploy release handling

erlang的热部署， 相当吸引人，它的release handling 作的非常细， 有核心模块支持（emulator实现）， 有工具支持，支持远程部署， 非常适合工业级别的应用。

文档很详细， 感觉应用起来前途很光明。

有空专门写个文章分析下原理。

1.20 查看erlang内存使用情况

发表时间: 2007-10-17 关键字: erlang info

io:format("~s~n", [binary_to_list(erlang:info(info))]).

得到结果
=memory
total: 219521173
processes: 718806
processes_used: 713510
system: 218802367
atom: 347085
atom_used: 338851
binary: 13159
code: 2951013
ets: 182152
=hash_table:atom_tab
size: 6421
used: 4522
objs: 7848
depth: 6
=index_table:atom_tab
size: 8192
limit: 1048576
entries: 7848
=hash_table:module_code
size: 97
used: 73
objs: 114
depth: 4
=index_table:module_code
size: 1024
limit: 65536
entries: 114
=hash_table:export_list
size: 2411
used: 1874

objs: 3674
depth: 7
=index_table:export_list
size: 4096
limit: 65536
entries: 3674
=hash_table:secondary_export_table
size: 97
used: 0
objs: 0
depth: 0
=hash_table:process_reg
size: 23
used: 17
objs: 34
depth: 4
=hash_table:fun_table
size: 797
used: 436
objs: 658
depth: 5
=hash_table:node_table
size: 5
used: 1
objs: 1
depth: 1
=hash_table:dist_table
size: 5
used: 1
objs: 1
depth: 1
=allocated_areas
processes: 713510 718806
ets: 182152
sys_misc: 5330186
static: 209977344
atom_space: 98316 90370
binary: 13159

atom_table: 58533
module_table: 4568
export_table: 26112
register_table: 156
fun_table: 3250
module_refs: 1024
loaded_code: 2697335
dist_table: 147
node_table: 107
bits_bufs_size: 2
bif_timer: 40100
link_lh: 0
proc: 21156 17636
atom_entry: 190236 189948
export_entry: 177400 177112
module_entry: 4848 4608
reg_proc: 992 848
monitor_sh: 828 108
nlink_sh: 3896 2984
proc_list: 24 24
fun_entry: 37116 37004
db_tab: 3552 3376
driver_event_data_state: 24 24
driver_select_data_state: 188 60
=allocator:sys_alloc
option e: true
option m: libc
option tt: 131072
option tp: 0
=allocator:temp_alloc
versions: 0.9 2.1
option e: true
option sbct: 524288
option asbcst: 4145152
option rsbcst: 90
option rsbcmt: 80
option mmbcs: 131072
option mmsbc: 256

option mmmbc: 10
option lmbcs: 5242880
option smbcs: 1048576
option mbcgs: 10
option as: af
mbcs blocks: 0 9 9
mbcs blocks size: 0 65544 65544
mbcs carriers: 1 1 1
mbcs mseg carriers: 0
mbcs sys_alloc carriers: 1
mbcs carriers size: 131104 131104 131104
mbcs mseg carriers size: 0
mbcs sys_alloc carriers size: 131104
sbcs blocks: 0 0 0
sbcs blocks size: 0 0 0
sbcs carriers: 0 0 0
sbcs mseg carriers: 0
sbcs sys_alloc carriers: 0
sbcs carriers size: 0 0 0
sbcs mseg carriers size: 0
sbcs sys_alloc carriers size: 0
temp_alloc calls: 14136
temp_free calls: 14136
temp_realloc calls: 46
mseg_alloc calls: 0
mseg_dealloc calls: 0
mseg_realloc calls: 0
sys_alloc calls: 1
sys_free calls: 0
sys_realloc calls: 0
=allocator:sl_alloc
versions: 2.1 2.1
option e: true
option sbct: 524288
option asbcst: 4145152
option rsbcst: 80
option rsbcmt: 80
option mmbcs: 131072

option mmsbc: 256
option mmmbc: 10
option lmbcs: 5242880
option smbcs: 1048576
option mbcgs: 10
option mbsd: 3
option as: gf
mbcs blocks: 0 3 3
mbcs blocks size: 0 240 240
mbcs carriers: 1 1 1
mbcs mseg carriers: 0
mbcs sys_alloc carriers: 1
mbcs carriers size: 131104 131104 131104
mbcs mseg carriers size: 0
mbcs sys_alloc carriers size: 131104
sbcs blocks: 0 0 0
sbcs blocks size: 0 0 0
sbcs carriers: 0 0 0
sbcs mseg carriers: 0
sbcs sys_alloc carriers: 0
sbcs carriers size: 0 0 0
sbcs mseg carriers size: 0
sbcs sys_alloc carriers size: 0
sl_alloc calls: 37
sl_free calls: 37
sl_realloc calls: 0
mseg_alloc calls: 0
mseg_dealloc calls: 0
mseg_realloc calls: 0
sys_alloc calls: 1
sys_free calls: 0
sys_realloc calls: 0
=allocator:std_alloc
versions: 0.9 2.1
option e: true
option sbct: 524288
option asbcst: 4145152
option rsbcst: 20

option rsbcmt: 80
option mmbcs: 131072
option mmsbc: 256
option mmmbc: 10
option lmbcs: 5242880
option smbcs: 1048576
option mbcgs: 10
option as: bf
mbcs blocks: 298 298 298
mbcs blocks size: 15480 15480 15480
mbcs carriers: 1 1 1
mbcs mseg carriers: 0
mbcs sys_alloc carriers: 1
mbcs carriers size: 131104 131104 131104
mbcs mseg carriers size: 0
mbcs sys_alloc carriers size: 131104
sbcs blocks: 0 0 0
sbcs blocks size: 0 0 0
sbcs carriers: 0 0 0
sbcs mseg carriers: 0
sbcs sys_alloc carriers: 0
sbcs carriers size: 0 0 0
sbcs mseg carriers size: 0
sbcs sys_alloc carriers size: 0
std_alloc calls: 349
std_free calls: 51
std_realloc calls: 3
mseg_alloc calls: 0
mseg_dealloc calls: 0
mseg_realloc calls: 0
sys_alloc calls: 1
sys_free calls: 0
sys_realloc calls: 0
=allocator:ll_alloc
versions: 0.9 2.1
option e: true
option sbct: 4294967295
option asbcst: 0

option rsbcst: 0
option rsbcmt: 0
option mmbcs: 2097152
option mmsbc: 0
option mmmbc: 0
option lmbcs: 5242880
option smbcs: 1048576
option mbcgs: 10
option as: aobf
mbcs blocks: 796 796 796
mbcs blocks size: 218837752 218837752 218837752
mbcs carriers: 4 4 4
mbcs mseg carriers: 0
mbcs sys_alloc carriers: 4
mbcs carriers size: 219152416 219152416 219152416
mbcs mseg carriers size: 0
mbcs sys_alloc carriers size: 219152416
sbcs blocks: 0 0 0
sbcs blocks size: 0 0 0
sbcs carriers: 0 0 0
sbcs mseg carriers: 0
sbcs sys_alloc carriers: 0
sbcs carriers size: 0 0 0
sbcs mseg carriers size: 0
sbcs sys_alloc carriers size: 0
ll_alloc calls: 804
ll_free calls: 8
ll_realloc calls: 284
mseg_alloc calls: 0
mseg_dealloc calls: 0
mseg_realloc calls: 0
sys_alloc calls: 4
sys_free calls: 0
sys_realloc calls: 0
=allocator:eheap_alloc
versions: 2.1 2.1
option e: true
option sbct: 524288

option asbcst: 4145152
option rsbcst: 50
option rsbcmt: 80
option mmbcs: 524288
option mmsbc: 256
option mmmbc: 10
option lmbcs: 5242880
option smbcs: 1048576
option mbcgs: 10
option mbsd: 3
option as: gf
mbcs blocks: 87 104 104
mbcs blocks size: 561896 942664 942664
mbcs carriers: 2 2 2
mbcs mseg carriers: 1
mbcs sys_alloc carriers: 1
mbcs carriers size: 1572896 1572896 1572896
mbcs mseg carriers size: 1048576
mbcs sys_alloc carriers size: 524320
sbcs blocks: 0 0 0
sbcs blocks size: 0 0 0
sbcs carriers: 0 0 0
sbcs mseg carriers: 0
sbcs sys_alloc carriers: 0
sbcs carriers size: 0 0 0
sbcs mseg carriers size: 0
sbcs sys_alloc carriers size: 0
eheap_alloc calls: 5615
eheap_free calls: 5528
eheap_realloc calls: 1525
mseg_alloc calls: 4
mseg_dealloc calls: 3
mseg_realloc calls: 0
sys_alloc calls: 1
sys_free calls: 0
sys_realloc calls: 0
=allocator:binary_alloc
versions: 0.9 2.1

option e: true
option sbct: 524288
option asbcst: 4145152
option rsbcst: 20
option rsbcmt: 80
option mmbcs: 131072
option mmsbc: 256
option mmmbc: 10
option lmbcs: 5242880
option smbcs: 1048576
option mbcgs: 10
option as: bf
mbcs blocks: 3 35 35
mbcs blocks size: 13176 621224 621224
mbcs carriers: 1 2 2
mbcs mseg carriers: 0
mbcs sys_alloc carriers: 1
mbcs carriers size: 131104 1179680 1179680
mbcs mseg carriers size: 0
mbcs sys_alloc carriers size: 131104
sbcs blocks: 0 0 0
sbcs blocks size: 0 0 0
sbcs carriers: 0 0 0
sbcs mseg carriers: 0
sbcs sys_alloc carriers: 0
sbcs carriers size: 0 0 0
sbcs mseg carriers size: 0
sbcs sys_alloc carriers size: 0
binary_alloc calls: 2514
binary_free calls: 2511
binary_realloc calls: 15
mseg_alloc calls: 17
mseg_dealloc calls: 17
mseg_realloc calls: 0
sys_alloc calls: 1
sys_free calls: 0
sys_realloc calls: 0
=allocator:ets_alloc

versions: 0.9 2.1
option e: true
option sbct: 524288
option asbcst: 4145152
option rsbcst: 20
option rsbcmt: 80
option mmbcs: 131072
option mmsbc: 256
option mmmbc: 10
option lmbcs: 5242880
option smbcs: 1048576
option mbcgs: 10
option as: bf
mbcs blocks: 632 632 632
mbcs blocks size: 159016 159016 159016
mbcs carriers: 2 2 2
mbcs mseg carriers: 1
mbcs sys_alloc carriers: 1
mbcs carriers size: 1179680 1179680 1179680
mbcs mseg carriers size: 1048576
mbcs sys_alloc carriers size: 131104
sbcs blocks: 0 0 0
sbcs blocks size: 0 0 0
sbcs carriers: 0 0 0
sbcs mseg carriers: 0
sbcs sys_alloc carriers: 0
sbcs carriers size: 0 0 0
sbcs mseg carriers size: 0
sbcs sys_alloc carriers size: 0
ets_alloc calls: 4362
ets_free calls: 3730
ets_realloc calls: 298
mseg_alloc calls: 1
mseg_dealloc calls: 0
mseg_realloc calls: 0
sys_alloc calls: 1
sys_free calls: 0
sys_realloc calls: 0

=allocator:fix_alloc
option e: true
proc: 21156 17636
atom_entry: 190236 189948
export_entry: 177400 177112
module_entry: 4848 4608
reg_proc: 992 848
monitor_sh: 828 108
nlink_sh: 3896 2984
proc_list: 24 24
fun_entry: 37116 37004
db_tab: 3552 3376
driver_event_data_state: 24 24
driver_select_data_state: 188 60
=allocator:mseg_alloc
version: 0.9
option amcbf: 4194304
option rmcbf: 20
option mcs: 5
option cci: 1000
cached_segments: 0
cache_hits: 16
segments: 2
segments_watermark: 2
mseg_alloc calls: 22
mseg_dealloc calls: 20
mseg_realloc calls: 0
mseg_create calls: 6
mseg_destroy calls: 4
mseg_recreate calls: 0
mseg_clear_cache calls: 0
mseg_check_cache calls: 5
=allocator:alloc_util
option mmc: 1024
option ycs: 1048576
=allocator:instr
option m: false
option s: false

option t: false

ok

1.21 开始规模研究global 和 global_group global_search的原理

发表时间: 2007-10-18 关键字: global global_group global_search erlang dist

The ability to globally register names is a central concept in the programming of distributed Erlang systems.

分布式系统 包括流行的p2p系统 要实现的好 必须解决的一个问题是 global名称的问题。 erlang实现的很复杂 erl代码实现了好几千行 折合c语言是好几w行 所以这个是erlang和其他系统竞争的强大优势。

但是实现原理上erlang没有提供相应的设计文档 而且google上也找不出任何有价值的材料 只能自己从代码分析出工作原理。 分析明白了 对实现其他的系统都有很大的借鉴意义。

action, now! 请期待。

1.22 erlang远程控制

发表时间: 2007-10-20 关键字: erlang slave pool ssh rsh connect boot loader

1. MODULE

slave

MODULE SUMMARY

Functions to Starting and Controlling Slave Nodes

2. MODULE

pool

MODULE SUMMARY

Load Distribution Facility

3. Erlang Shell

```
Eshell V5.5.5  (abort with ^G)
1>
User switch command
 --> h
  c [nn]   - connect to job
  i [nn]   - interrupt job
  k [nn]   - kill job
  j        - list all jobs
  s        - start local shell
  r [node] - start remote shell
  q        - quit erlang
  ? | h    - this message
 -->
```

4. MODULE

erl_boot_server

MODULE SUMMARY

Boot Server for Other Erlang Machines

利用slave模块 通过rsh ssh之类的程序 启动远端机器上的erlang程序， 接受主节点的控制， 完成如自动部署 资料收集 远程控制等活动。
pool模块进行简单的负载均衡。

通过shell 我们可以connect到远端的erlang的终端 下达命令。

通过erl_boot_server 可以实现模块从远端自动加载 省去了erlang程序部署和更新的繁琐 你只要更新主节点。

有这4个模块 我们的life easy很多。

1.23 erlang sctp支持

发表时间: 2007-10-22 关键字: erlang sctp

erlang kernel 带了个gen_sctp 实现了sctp的 client和server操作， 在linux 2.6 上协议栈支持 lksctp。 网络如果有双线 那么sctp就非常有用 可以起到网络冗余的作用。遗憾的是, R11B5 没有inet_sctp_dist模块 否则erlang的rpc也能走sctp， 那多美呀！

sctp可以参考ibm linux china上的文章。

1.24 erlang inet_ssl_dist有BUG 导致节点通讯不能用SSL

发表时间: 2007-10-23 关键字: inet_ssl_dist ssl node message

Erlang R115B 的inet_ssl_dist模块有bug, 我已经定位到问题 提交到 bug mailist，erlang社区的人已经修了这个BUG,并且提交了patch．有了SSL 通讯更安全些。

具体请参看： http://avindev.javaeye.com/blog/103310

1.25 erlang driver和热部署（重要）

发表时间: 2007-10-23 关键字: driver erl_ddll 热部署 release_handling

erlang的热部署包括2个方便的 beam(.beam)级别的和driver(.dll .so)级别的. beam级别的就简单。但是driver级别的就相对复杂很多。

先看下erl_ddll的说明:

Loading and reloading for code replacement

This scenario occurs when the driver code might need replacement during operation of the Erlang emulator. Implementing driver code replacement is somewhat more tedious than beam code replacement, as one driver cannot be loaded as both "old" and "new" code. All users of a driver must have it closed (no open ports) before the old code can be unloaded and the new code can be loaded.

The actual unloading/loading is done as one atomic operation, blocking all processes in the system from using the driver concerned while in progress.

The preferred way to do driver code replacement is to let **one single process** keep track of the driver. When the process start, the driver is loaded. When replacement is required, the driver is reloaded. Unload is probably never done, or done when the process exits. If more than one user has a driver loaded when code replacement is demanded, the replacement cannot occur until the last "other" user has unloaded the driver.

Demanding reload when a reload is already in progress is always an error. Using the high level functions, it is also an error to demand reloading when more than one user has the driver loaded. To simplify driver replacement, avoid designing your system so that more than than one user has the driver loaded.

由于动态库不能在操作系统进程内存里面 有新旧之分 所以最好的方法就是让 唯一 的erlang进程 管理这个port 由这个process 代理所有的请求和响应。这种情况下 更换driver就很简单 因为系统中只有一个driver的使用者， 非常容易作load unload reload的操作。而且在process异常终止的时候 emulator自动会释放driver的资源。

这个回答了我许久以来困惑我的问题： 为什么erlang kernel里面的file->file_server, code->code_server这样的架构， 终于在这个时候真相大白。看来erlang比我们考虑的要周道。

有兴趣的同学可以参考下release_handling的实现，相信会对hot deployment 有很深的理解。

1.26 Erlang heart 高可靠性的最后防线

发表时间: 2007-10-24 关键字: heart watchdog

heart由2部份组成：
1. 外部程序 heart
2. erlang port模块 heart.erl。

当开启heart的时候（erl - heart ...) 外部程序heart被erlang模块heart.erl 启动起来，监视emulator的运作. heart.erl 每隔一定的时间向heart外部程序报告状态。当外部heart没有监测到心跳的时候就要采取行动 重新运行$HEART_COMMAND所指定的命令。

heart机制有2个用处：

1. erlang虽然内置了很多supervisor 可以保证process的高可靠性 但是假如emulator死亡了，那这一切都消失了， erlang只能靠heart 来重新启动。

2。热部署的时候 release_handling 需要重新启动emulator的时候也必须借助外部程序。 即heart来起作用。

所以在heart模式下 你的erlang程序是杀不掉的 除非你先kill掉heart进程。

1.27 erlang R12B预览版本出来了

发表时间: 2007-10-27 关键字: R12B release

erlang R12B预览版本出来了 可以在 http://erlang.org/download/snapshots/otp_src_R12B-0.tar.gz 下载, 正在比对 看作了什么改进。

1.28 why-do-you-like-erlang(转)

发表时间: 2007-11-01 关键字: network server thread core

原文Url: http://www.clickcaster.com/items/why-do-you-like-erlang
感觉写的非常好（激动中） 和我的经历几乎一样 得出的结论就是: erlang就是我要的 网络应用erlang足够强大了

After having turned into something of an Erlang zealot about two months ago, I've been asked this question quite frequently. I think the best way to explain this is as a corollary to Greenspun's Tenth Rule:

> Any sufficiently advanced concurrent C program contains an ad hoc, informally-specified, bug ridden, slow implementation of half of Erlang

I've always had a slight obsession with developing network servers which I intended to process high numbers of concurrent connections. For years and years I tried to do this in C, exploring many different methods of event monitoring, as well as exploring different ways of implementing network servers as described in such books as Unix Network Programming. Sadly I found many of these models and approaches out-of-date. When glibc came out for Linux, threading became the de-facto standard.

Years later, I would try to implement a massively concurrent network server which used a thread-per-request model. By then I had built up quite a home-brewed server framework, with abstract interfaces to many OS facilities, and had written a network server with over 20,000 lines of code.

This is where I began to hit the wall with threads. I had cross-platform threading abstractions between Win32 and pthreads, and had been testing on a number of platforms, and started noticing that on certain platforms I was running into deadlocks, but not others. As the data structures grew increasingly advanced I started to notice more deadlocks. The locking became increasingly more sophisticated, more difficult to debug, and the performance began to decrease as important shared structures needed to be synchronized across all threads.

I began reading more on the overhead of mutexes, which for a performance-obsessed C programmer was quite maddening. Locking a mutex on Win32, for example, required over 600 CPU instructions! I soon came to the realization that the best approach was really one or two threads per CPU core, and that some lighter weight concurrency mechanism which avoids the synchronization problems of threads was the better approach to addressing high numbers of concurrent connections.

I first discovered Knuth's Coroutines as described in The Art of Computer Programming. These provide for lightweight cooperative scheduling without the need of a system call to switch between them. The GNU Pth library implemented something quite similar. I found several similar approaches to userspace threading, including the idea of "microthreads" as implemented by Win32's Fibers and *IX's ucontexts.

The end goal of all of these approaches was separating behavior from function. Each native thread (provided you have more than one) has its behavior isolated into an inner event loop, and the function can be implemented as a microthread.

Provided you got this far, subsequent problems occur when you discover you need a way to send messages between microthreads in a manner which abstracts processing network I/O events, and if you're using threads also lets you send messages between them. I experimented for years with this problem, coming up with some pretty complex abstractions between a multitude of event sources and threading models.

In the end, after years of struggling to build a framework for concurrent network programming, I never reached a point where I was satisfied. I never felt I had truly abstracted away the problems I was trying to solve.

Eventually I realized shared-nothing was the way to go. I became quite enamored with Dan Bernstein's qmail application, which used a collection of shared-nothing OS processes which communicate over pipes in order to process network traffic. Sick of threads, sick of microthreads, sick of trying to synchronize shared information, this is where I ended up. The wisdom of the Unix philosophy: "Do one thing and do it well," finally manifested itself in my eyes.

Finally, I found Erlang. The more I read about it, the more I realized it was an absolutely brilliant implementation of the very model that I had, for years, tried to implement myself. Furthermore, it combined the models of multiple native threads per CPU core which in turn run multiple "microthreads" with a messaging system far more elegant than I ever mustered myself.

For those of you who have not read it, here is a description of Erlang message processing, excerpted from the recently released Programming Erlang:

1. When we enter a receive statement, we start a timer (but only if an after section is present in the expression).

2. Take the first message in the mailbox and try to match it against Pattern1, Pattern2, and so on. If the match succeeds, the message is removed from the mailbox, and the expressions following the pattern are evaluated.
3. If none of the patterns in the receive statement matches the first message in the mailbox, then the first message is removed from the mailbox and put into a "save queue." The second message in the mailbox is then tried. This procedure is repeated until a matching message is found or until all the messages in the mailbox have been examined.
4. If none of the messages in the mailbox matches, then the process is suspended and will be rescheduled for execution the next time a new message is put in the mailbox. Note that when a new message arrives, the messages in the save queue are not rematched; only the new message is matched.
5. As soon as a message has been matched, then all messages that have been put into the save queue are reentered into the mailbox in the order in which they arrived at the process. If a timer was set, it is cleared.
6. If the timer elapses when we are waiting for a message, then evaluate the expressions ExpressionsTimeout and put any saved messages back into the mailbox in the order in which they arrived at the process.

After reading this I realized this was what I was trying to invent all along. I showed this approach to Zed Shaw (whom you might recognize as the creator of the Mongrel web server) and he expressed that it was quite similar to the way he implemented message handling in his Myriad framework. I'm not going to speak for Zed in terms of Myriad being an ad hoc reimplementation of Erlang, but just point it out to show that when dealing with these sorts of problems long enough people come to the same sort of conclusions about how to solve them.

All that said, I felt naturally comfortable with the Erlang way of doing things. This is how I've been writing programs all along. It's feels so nice to finally find a language that abstracts away all the complex behaviors of concurrent network processing. Furthermore, distributed Erlang and the OTP framework are beyond my wildest dreams in terms of what you can build once the underlying problems are all abstracted away.

Now I can finally stop trying to build an ad hoc, informally-specified, bug-ridden framework, and start focusing on what I've actually cared about all along: the function.

1.29 Joel Reymont号称要出erlang新书教你高级网络应用

发表时间: 2007-11-10 关键字: book game otp

注：Joel Reymont openpoker游戏的作者 都是非常高级的话题 涉及到实际项目的设计 编码 调试 部署 维护等话题 覆盖到大部分的otp应用 我们拭目以待。

Working title is Hardcore Erlang and it will be built around
OpenPoker. The emphasis of the book will be more on showing how to
write scalable Erlang servers and less on how to write a poker server.

Apart from the following topics, are there any other topics you would
like me to cover?

Thanks, Joel

- The architecture of a poker server from far above

- Thinking processes instead of objects

- Game logic
 - Stacking state machines
 - Swapping logic

- Storing data in Mnesia

- State machines (gen_fsm)

- OTP behaviours

- Poker bots
 - Simulating players
 - Scripting

- Designing a network protocol
 - Binary parsing

- Pickler combinators

- Automatic clustering

- Fault tolerance and fail-over

- Load balancing

- Testing a network server

- Debugging Erlang software

Thanks, Joel

1.30 erlang r12 新增加Percept并发性能调测模块

发表时间: 2007-11-19 关键字: Percept profile 并发

Percept is an application-level profiler with focus on parallelism.

Can help in finding:

– synchronization bottlenecks

– periods of few runnable processes

Makes use of new trace points in the virtual machine.

Collects data about when processes are runnable andwaiting.

Graphical interactive presentation of collected data.

1.31 erlang函数调用新语法(用于代码hot replace)

发表时间: 2008-01-18 关键字: 函数 调用

R12B0的文档里面写着：

6.6 Function Calls

ExprF(Expr1,...,ExprN)
ExprM:ExprF(Expr1,...,ExprN)

ExprM should evaluate to a module name and ExprF to a function name or a fun.

When including the module name, the function is said to be called by using the fully qualified function name. This is often referred to as a remote or external function call. Example:

lists:keysearch(Name, 1, List)

The module name can be omitted, if ExprF evaluates to the name of a local function, an imported function, or an auto-imported BIF. Then the function is said to be called by using the implicitly qualified function name. Examples:

handle(Msg, State)
spawn(m, init, [])

To avoid possible ambiguities, the fully qualified function name must be used when calling a function with the same name as a BIF, and the compiler does not allow defining a function with the same name as an imported function.

Note that when calling a local function, there is a difference between using the implicitly or fully qualified function name, as the latter always refer to the latest version of the module. See Compilation and Code Loading.

If ExprF evaluates to a fun, only the format ExprF(Expr1,...,ExprN) is correct. Example:

```
Fun1 = fun(X) -> X+1 end
Fun1(3)
=> 4

Fun2 = {lists,append}
Fun2([1,2],[3,4])
=> [1,2,3,4]
```

For code replacement of funs to work, the tuple syntax {Module,FunctionName} must be used to represent the fun.

1.32 R12B0的文档细致了很多

发表时间: 2008-01-18 关键字: 文档 patch

新文档重点的部分都用色彩重点标出 而且告诉你什么函数声明模块要废弃 要注意什么事项 真的很细致哦。赞个。另外R12B0的patch也出来了，在download页面里面有readme.

1.33 arbow发起 erlycachedb 项目

发表时间: 2008-01-21 关键字: erlycachedb

An memcachedb clone by Erlang 支持cache的分布和持久化。

http://code.google.com/p/erlycachedb/

年前解决issue 1,2 的问题。

1.34 dets ram_file模式和 dets + ets的选择

发表时间: 2008-01-22 关键字: dets ram_file

从mod_security_server.erl 中摘抄的:

```
%%
%% The storage model is a write-through model with both an ets and a dets
%% table. Writes are done to both the ets and then the dets table, but reads
%% are only done from the ets table.
%%
%% This approach also enables parallelism when using dets by returning the
%% same dets table identifier when opening several files with the same
%% physical location.
%%
%% NOTE: This could be implemented using a single dets table, as it is
%%       possible to open a dets file with the ram_file flag, but this
%%       would require periodical sync's to disk, and it would be hard
%%       to decide when such an operation should occur.
%%
```

这个选择比较有趣！

1.35 how to call os:cmd("ls") from shell?（老掉牙问题）

发表时间: 2008-01-25 关键字: run eval

之前erlang邮件列表的一个问题：

hi list,

I fellowed the otp document, but it failed.

erl -s os cmd ls

Erlang (BEAM) emulator version 5.5.5 [source] [async-threads:0] [hipe]
[kernel-poll:false] [lock-checking]

{"init terminating in
do_boot",{function_clause,[{os,validate1,[[ls]]},{os,cmd,1},{init,start_it,1},{init,start_em,1}]}}

Crash dump was written to: erl_crash.dump

Thx,
Jeremy

_____
erlang-questions mailing list
erlang-questions@erlang.org
http://www.erlang.org/mailman/listinfo/erlang-questions
Reply

Reply to all

Forward

yerl@club-internet.fr to erlanging, erlang-questio.

show details 7/11/07

Reply

echo 'os:cmd("ls").' | erl

cheers
Y.

----Message d'origine----
>Date: Wed, 11 Jul 2007 14:06:43 +0800
>De: "Jeremy Chow" <erlanging@gmail.com>
>A: erlang-questions@erlang.org
>Sujet: [erlang-questions] how to call os:cmd("ls") from shell?
- Show quoted text -
>
>hi list,
>
>I fellowed the otp document, but it failed.
>
># erl -s os cmd ls
>
>Erlang (BEAM) emulator version 5.5.5 [source] [async-threads:0] [hipe]
>[kernel-poll:false] [lock-checking]
>
>{"init terminating in
>do_boot",{function_clause,[{os,validate1,[[ls]]},{os,cmd,1},{init,start_it,1},{init,start_em,1}]}}
>
>Crash dump was written to: erl_crash.dump
>
>
>Thx,
>Jeremy
>_____
>erlang-questions mailing list
>erlang-questions@erlang.org
>http://www.erlang.org/mailman/listinfo/erlang-questions

>

_____

erlang-questions mailing list
erlang-questions@erlang.org
http://www.erlang.org/mailman/listinfo/erlang-questions
Reply

Reply to all

Forward

Bengt Kleberg to erlang-questio.

show details 7/11/07

Reply

On 2007-07-11 08:06, Jeremy Chow wrote:
> hi list,
>
> I fellowed the otp document, but it failed.
>
> # erl -s os cmd ls

this will not work.
1 erl -s will bundle the arguments (ls) into a list as atoms.
os:cmd([ls]).
2 os:cmd/1 wants a string. os:cmd("ls").

one might think that
erl -run os cmd ls
will work since -run keeps the arguments as strings, but -run also

bundles all arguments into a single string.

so you have to write an interface module that calls os:cmd/1 for you.

bengt
--
Those were the days...
EPO guidelines 1978: "If the contribution to the known art resides
solely in a computer program then the subject matter is not
patentable in whatever manner it may be presented in the claims."
- Show quoted text -

参见init 文档 :

-eval Expr
Scans, parses and evaluates an arbitrary expression Expr during system initialization. If any of these steps fail (syntax error, parse error or exception during evaluation), Erlang stops with an error message. Here is an example that seeds the random number generator:

```
% erl -eval '{X,Y,Z}' = now(), random:seed(X,Y,Z).'
```

This example uses Erlang as a hexadecimal calculator:

```
% erl -noshell -eval 'R = 16#1F+16#A0, io:format("~.16B~n", [R])' \
-s erlang halt
BF
```

If multiple -eval expressions are specified, they are evaluated sequentially in the order specified. -eval expressions are evaluated sequentially with -s and -run function calls (this also in the order specified). As with -s and -run, an evaluation that does not terminate, blocks the system initialization process.
-extra
Everything following -extra is considered plain arguments and can be retrieved using get_plain_arguments/0.
-run Mod [Func [Arg1, Arg2, ...]]
Evaluates the specified function call during system initialization. Func defaults to start. If no

arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list [Arg1,Arg2,...] as argument. All arguments are passed as strings. If an exception is raised, Erlang stops with an error message.

Example:

% erl -run foo -run foo bar -run foo bar baz 1 2

This starts the Erlang runtime system and evaluates the following functions:

foo:start()
foo:bar()
foo:bar(["baz", "1", "2"]).

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a -run call which does not return will block further processing; to avoid this, use some variant of spawn in such cases.

-s Mod [Func [Arg1, Arg2, ...]]

Evaluates the specified function call during system initialization. Func defaults to start. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list [Arg1,Arg2,...] as argument. All arguments are passed as atoms. If an exception is raised, Erlang stops with an error message.

Example:

% erl -s foo -s foo bar -s foo bar baz 1 2

This starts the Erlang runtime system and evaluates the following functions:

foo:start()
foo:bar()
foo:bar([baz, '1', '2']).

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a -s call which does not return will block further processing; to avoid this, use some variant of spawn in such cases.

Due to the limited length of atoms, it is recommended that -run be used instead.

还可以这样:

erl -noshell -eval 'os:cmd("ls").' -s init stop

但是因为没有shell所以目录的内容没有被列出来。

貌似没有最好的方案!!!

1.36 erlang r12b1 改进内存分配效率,大幅提高smp的运行速度

发表时间: 2008-02-23 关键字: r12b-1 内存 lksctp

release 说明： http://www.erlang.org/download/otp_src_R12B-1.readme

对比了下代码看来r12b1主要是在smp情况下改进了内存分配的速度(内存浪费更多 典型空间换时间) 还有就是lksctp的支持。
binary的也做了少许的优化。其他的大多数是bugfix。

1.37 erlsnoop erlang消息监听器 调试erlang网络程序利器

发表时间: 2008-03-04 关键字: erlsnoop 监听 调试

在erlang的邮件列表上看到：

Have you tried putting a snoop to see whether the delay is on the
sending/receiving side?

This might be useful: http://www.erlang.org/contrib/erlsnoop-1.0.tgz

去http://www.erlang.org/contrib/看了下 模块真不少 下载了erl_snoop

先安装lib-pcap yum install libpcap-devel
yum install libpcap

编译出错:
ip.c:77: error: label at end of compound statement
随便在default: 后面加个return 0;

utils.c: In function 'gmt2local':
utils.c:26: error: storage size of 'sgmt' isn't known
utils.c:28: warning: implicit declaration of function 'time'
utils.c:31: error: dereferencing pointer to incomplete type
utils.c:31: warning: implicit declaration of function 'gmtime'
utils.c:31: error: invalid type argument of 'unary *'
utils.c:32: warning: implicit declaration of function 'localtime'
utils.c:32: warning: assignment makes pointer from integer without a cast
utils.c:34: error: dereferencing pointer to incomplete type
utils.c:34: error: dereferencing pointer to incomplete type
utils.c:35: error: dereferencing pointer to incomplete type
utils.c:35: error: dereferencing pointer to incomplete type
utils.c:42: error: dereferencing pointer to incomplete type
utils.c:42: error: dereferencing pointer to incomplete type
utils.c:43: error: dereferencing pointer to incomplete type
utils.c:43: error: dereferencing pointer to incomplete type
utils.c:26: warning: unused variable 'sgmt'

make: *** [utils.o] Error 1

加个 #include <time.h>

搞定 运行

```
[root@test98 erlsnoop-1.0]# ./erlsnoop -hpnkt
Erlsnoop 1.0 (Mar  4 2008)
using interface eth0 (mtu=1500)
using filter "tcp"
option -? gets help
type ^C to quit

[ 192.168.0.98:44683 (x@192.168.0.98) > 192.168.0.243:30141 (y@192.168.0.243) ] 160
  REG_SEND from: #Pid<x@192.168.0.98.11.0.2> to: global_name_server

[ 192.168.0.98:44683 (x@192.168.0.98) > 192.168.0.243:30141 (y@192.168.0.243) ] 49
  MONITOR from: #Pid<x@192.168.0.98.36.0.2> to: net_kernel
   ref: #Ref<x@192.168.0.98.46.0.0.2>

[ 192.168.0.243:30141 (y@192.168.0.243) > 192.168.0.98:44683 (x@192.168.0.98) ] 161
  REG_SEND from: #Pid<y@192.168.0.243.11.0.2> to: global_name_server

[ 192.168.0.98:44683 (x@192.168.0.98) > 192.168.0.243:30141 (y@192.168.0.243) ] 84
  REG_SEND from: #Pid<x@192.168.0.98.36.0.2> to: net_kernel
```

效果不错的哦。可以看到erlang的交互信息， 相信的请看README.txt

1.38 Parsing CSV in erlang(转)

发表时间: 2008-03-05 关键字: parsing csv in erlang

引用地址（需要爬墙）http://ppolv.wordpress.com/2008/02/25/parsing-csv-in-erlang/

So I need to parse a CSV file in erlang.

Although files in CSV have a very simple structure, simply calling lists:tokens(Line,",") for each line in the file won't do the trick, as there could be quoted fields that spans more than one line and contains commas or escaped quotes.

A detailed discussion of string parsing in erlang can be found at the excellent Parsing text and binary files with Erlang article by Joel Reymont. And the very first example is parsing a CSV file!; but being the first example, it was written with simplicity rather than completeness in mind, so it didn't take quoted/multi-line fields into account.

Now, we will write a simple parser for RFC-4180 documents (witch is way cooler than parse plain old CSV files ;-)) . As the format is really simple, we won't use yecc nor leex, but parse the input file by hand using binaries,lists and lots of pattern matching.

Our goals are

* Recognize fields delimited by commas, records delimited by line breaks
* Recognize quoted fields
* Being able to parse quotes, commas and line breaks inside quoted fields
* Ensure that all records had the same number of fields
* Provide a fold-like callback interface, in addition to a return-all-records-in-file function

What the parser won't do:

* Unicode. We will treat the file as binary and consider each character as ASCII, 1 byte wide. To parse unicode files, you can use xmerl_ucs:from_utf8/1, and then process the resulting list instead of the raw binary

A quick lock suggest that the parser will pass through the following states:
cvs parsing states

* Field start

at the begin of each field. The whitespaces should be consider for unquoted fields, but any whitespace before a quoted field is discarded

* Normal

an unquoted field

* Quoted

inside a quoted field

* Post Quoted

after a quoted field. Whitespaces could appear between a quoted field and the next field/record, and should be discarded

Parsing state

While parsing, we will use the following record to keep track of the current state

```
-record(ecsv,{
  state = field_start,  %%field_start|normal|quoted|post_quoted
  cols = undefined, %%how many fields per record
  current_field = [],
  current_record = [],
  fold_state,
  fold_fun  %%user supplied fold function
  }).
```

API functions

```
parse_file(FileName,InitialState,Fun) ->
  {ok, Binary} = file:read_file(FileName),
   parse(Binary,InitialState,Fun).

parse_file(FileName)  ->
  {ok, Binary} = file:read_file(FileName),
   parse(Binary).
```

```
parse(X) ->
  R = parse(X,[],fun(Fold,Record) -> [Record|Fold] end),
  lists:reverse(R).

parse(X,InitialState,Fun) ->
  do_parse(X,#ecsv{fold_state=InitialState,fold_fun = Fun}).
```

The tree arguments functions provide the fold-like interface, while the single argument one returns a list with all the records in the file.
Parsing

Now the fun part!.
The transitions (State X Input -> NewState) are almost 1:1 derived from the diagram, with minor changes (like the handling of field and record delimiters, common to both the normal and post_quoted state).
Inside a quoted field, a double quote must be escaped by preceding it with another double quote. Its really easy to distinguish this case by matching against

```
<<$",$",_/binary>>
```

sort of "lookahead" in yacc's lexicon.

```
%% --------- Field_start state ---------------------
%%whitespace, loop in field_start state
do_parse(<<32,Rest/binary>>,S = #ecsv{state=field_start,current_field=Field})->
do_parse(Rest,S#ecsv{current_field=[32|Field]});

%%its a quoted field, discard previous whitespaces
do_parse(<<$",Rest/binary>>,S = #ecsv{state=field_start})->
do_parse(Rest,S#ecsv{state=quoted,current_field=[]});

%%anything else, is a unquoted field
do_parse(Bin,S = #ecsv{state=field_start})->
do_parse(Bin,S#ecsv{state=normal});

%% --------- Quoted state ---------------------
%%Escaped quote inside a quoted field
do_parse(<<$",$",Rest/binary>>,S = #ecsv{state=quoted,current_field=Field})->
```

```
do_parse(Rest,S#ecsv{current_field=[$"|Field]});

%%End of quoted field
do_parse(<<$",Rest/binary>>,S = #ecsv{state=quoted})->
do_parse(Rest,S#ecsv{state=post_quoted});

%%Anything else inside a quoted field
do_parse(<<X,Rest/binary>>,S = #ecsv{state=quoted,current_field=Field})->
do_parse(Rest,S#ecsv{current_field=[X|Field]});

do_parse(<<>>, #ecsv{state=quoted})->
throw({ecsv_exception,unclosed_quote});

%% --------- Post_quoted state ----------------------
%%consume whitespaces after a quoted field
do_parse(<<32,Rest/binary>>,S = #ecsv{state=post_quoted})->
do_parse(Rest,S);

%%---------Comma and New line handling. ------------------
%%---------Common code for post_quoted and normal state---

%%EOF in a new line, return the records
do_parse(<<>>, #ecsv{current_record=[],fold_state=State})->
State;
%%EOF in the last line, add the last record and continue
do_parse(<<>>,S)->
do_parse([],new_record(S));

%% skip carriage return (windows files uses CRLF)
do_parse(<<$\r,Rest/binary>>,S = #ecsv{})->
do_parse(Rest,S);

%% new record
do_parse(<<$\n,Rest/binary>>,S = #ecsv{}) ->
do_parse(Rest,new_record(S));

do_parse(<<$, ,Rest/binary>>,S = #ecsv{current_field=Field,current_record=Record})->
do_parse(Rest,S#ecsv{state=field_start,
```

```
 current_field=[],
 current_record=[lists:reverse(Field)|Record]});

%%A double quote in any other place than the already managed is an error
do_parse(<<$",_Rest/binary>>, #ecsv{})->
throw({ecsv_exception,bad_record});

%%Anything other than whitespace or line ends in post_quoted state is an error
do_parse(<<_X,_Rest/binary>>, #ecsv{state=post_quoted})->
throw({ecsv_exception,bad_record});

%%Accumulate Field value
do_parse(<<X,Rest/binary>>,S = #ecsv{state=normal,current_field=Field})->
do_parse(Rest,S#ecsv{current_field=[X|Field]}).
```

Record assembly and callback

Convert each record to a tuple, and check that it has the same number of fields than the previous records. Invoke the callback function with the new record and the previous state.

```
%%check the record size against the previous, and actualize state.
new_record(S=#ecsv{cols=Cols,current_field=Field,current_record=Record,fold_state=State,fold_fun=Fun})
->
NewRecord = list_to_tuple(lists:reverse([lists:reverse(Field)|Record])),
if
(tuple_size(NewRecord) =:= Cols) or (Cols =:= undefined) ->
NewState = Fun(State,NewRecord),
S#ecsv{state=field_start,cols=tuple_size(NewRecord),
current_record=[],current_field=[],fold_state=NewState};

(tuple_size(NewRecord) =/= Cols) ->
throw({ecsv_exception,bad_record_size})
end.
```

Final notes

We used a single function, do_parse/2, with many clauses to do the parsing. In a more complex scenario, you probably will use different functions for different sections of the grammar you are

parsing. Also you could first tokenize the input and then parse the resulting token stream, this could make your work simpler even if your aren't using a parser generator like yecc (this is the approach i'm using to parse ldap filters).

1.39 file:read_file的注意细节

发表时间: 2008-03-05 关键字: file:read_file file:open file:read

arbow发现的问题如下：

4> file:read_file("/proc/cpuinfo").
{ok,<<>>}

11> {ok, IoDevice} = file:open("/proc/cpuinfo", [binary]),file:read(IoDevice, 1024).
{ok,<<"processor\t: 0\nvendor_id\t: GenuineIntel\ncpu family\t: 6\nmodel\t\t: 15\nmodel name\t:
Intel(R) Core(TM)2 CPU "...>>}

同样的虚拟文件 不同的读法一个有内容 一个没有。

真实的文件就不存在这个问题。

erl -smp disable (禁止多smp的原因是容易找到pid)
strace -p PID

结果如下：

。。。。
stat("/proc/cpuinfo", {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
open("/proc/cpuinfo", O_RDONLY) = 7
close(7) = 0

stat("/proc/cpuinfo", {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
open("/proc/cpuinfo", O_RDONLY) = 7
poll([{fd=3, events=POLLIN|POLLRDNORM}, {fd=5, events=POLLIN|POLLRDNORM}, {fd=0,
events=POLLIN|POLLRDNORM}], 3, 0) = 0
clock_gettime(CLOCK_MONOTONIC, {457571, 853399847}) = 0
read(7, "processor\t: 0\nvendor_id\t: Genuin"..., 8192) = 1238

看下源码红色部分：
static void invoke_read_file(void *data)
{

```
    struct t_data *d = (struct t_data *) data;
    size_t read_size;
    int chop;

    if (! d->c.read_file.binp) {
int fd;
int size;
if (! (d->result_ok =
       efile_openfile(&d->errInfo, d->c.read_file.name,
      EFILE_MODE_READ, &fd, &d->c.read_file.size))) {
    goto done;
}
d->fd = fd;
size = (int) d->c.read_file.size;
if (size != d->c.read_file.size ||
    ! (d->c.read_file.binp = driver_alloc_binary(size))) {
    d->result_ok = 0;
    d->errInfo.posix_errno = ENOMEM;
    goto close;
}
d->c.read_file.offset = 0;
    }

    read_size = (size_t) (d->c.read_file.size - d->c.read_file.offset);
    if (! read_size) goto close;
    chop = d->again && read_size >= FILE_SEGMENT_READ*2;
    if (chop) read_size = FILE_SEGMENT_READ;
    d->result_ok =
efile_read(&d->errInfo,
   EFILE_MODE_READ,
   (int) d->fd,
   d->c.read_file.binp->orig_bytes + d->c.read_file.offset,
   read_size,
   &read_size);
    if (d->result_ok) {
d->c.read_file.offset += read_size;
if (chop) return; /* again */
    }
```

```
close:
    efile_closefile((int) d->fd);
done:
    d->again = 0;
}

static void invoke_read(void *data)
{
    struct t_data *d = (struct t_data *) data;
    int status, segment;
    size_t size, read_size;

    segment = d->again && d->c.read.bin_size >= 2*FILE_SEGMENT_READ;
    if (segment) {
size = FILE_SEGMENT_READ;
    } else {
size = d->c.read.bin_size;
    }
    read_size = size;
    if (d->flags & EFILE_COMPRESSED) {
read_size = erts_gzread((gzFile)d->fd,
d->c.read.binp->orig_bytes + d->c.read.bin_offset,
size);
status = (read_size != -1);
if (!status) {
    d->errInfo.posix_errno = EIO;
}
    } else {
status = efile_read(&d->errInfo, d->flags, (int) d->fd,
    d->c.read.binp->orig_bytes + d->c.read.bin_offset,
    size,
    &read_size);
    }
    if ( (d->result_ok = status)) {
ASSERT(read_size <= size);
d->c.read.bin_offset += read_size;
if (read_size < size || !segment) {
    d->c.read.bin_size = 0;
```

```
    d->again = 0;
} else {
    d->c.read.bin_size -= read_size;
}
    } else {
d->again = 0;
    }
}
```

也就是说read_file 在发现文件长度为0的时候 不进行进一步读取.所以使用的时候请注意。

1.40 4月9号erlangR12B-2 release了 这次更新速度超快

发表时间: 2008-04-09 关键字: erlang r12b

Bug fix release : otp_src_R12B-2
Build date : 2008-04-09

This is bug fix release 2 for the R12B release.

看看什么变化。

1.41 erlang如何写port驱动与外面世界通讯

发表时间: 2008-04-12 关键字: pg2 port supervision

鉴于很多同学在帖子里面询问erlang与外面世界交互的port如何写的问题， david king很早前就写了篇文章详细地解决了这个问题，而且很有深度。请参见http://www.ketralnis.com/roller/dking/entry/20070903

Supervision Trees and Ports in Erlang

08:31PM Sep 03, 2007 in category Erlang by David King

Erlang provides a whole lot of infrastructure for doing tasks commonly associated with building giant fault-tolerant systems. But what about when you have to talk to an external non-Erlang program? Can we make our communication to that program fault-tolerant using only native Erlang/OTP components?
The problem

I'm writing a blogging engine using Erlyweb and Mnesia, and one of my tasks is to pass the user's blog entry through a formatter. I use two: one for Markdown, and one to sanitise for XSS attacks. Since there's no Markdown or HTML-sanitiser implementation for Erlang, we need to communicate with the Perl implementations (markdown, HTML-TagFilter).

We'll want some fault-tolerance (in case one of our external programs crash, we want to re-launch it), and we don't want to introduce a bottleneck by serialising access to our external programs, so we'll launch several copies of it to spread the load and allow some concurrency. In addition, we'll want to leave room to move all of the processing to its own node in the event that our blog gets very popular.

We'll communicate using an Erlang Port. A Port is a connection to an external program, implemented using Unix pipes. So our external program just has to know how to read/write to stdout/stdin. We're going to keep this all pretty generic to the actual external program. You should be able to do the same for any other program that can take data in length-encoded packets (or can be wrapped in one that does, like our Perl wrapper) for which you want to launch mutliple workers.

To launch our workers, we'll use Erlang's gen_server interface, which is a way to write a generic server without having to write the implementation details of the server, keeping state, message-handling and responding, etc. Instead, you write a gen_server with a set of callbacks, and all you write is the code to actually handle the messages that you want to respond to. Bascially, gen_server is a module that can take another module with appropriate callbacks and handle all of the tasks of making it a

server that looks more or less like (psuedo-code)

```
loop(State) ->
  Message=receive Some_Message
  if Message is an internal message, like shutdown or startup or code-replacement
    let gen_server handle it
    loop(State)
  else it's a message that the programmer wants to respond to
    pass it to the programmer, let them handle it, get new state
    loop(NewState)
```

Our gen_server will have to keep track of its connection to our external program, and on request it will have to send data to the external program and retrieve the result. It will also keep track of its process group (pg2) and adding itself to it

Erlang does something similar for supervisors. A supervisor is a process whose entire job is to watch another process or set of processes. In our case, we just want it to make sure that our external processes are up and running, and to re-launch them in case they crash, and to bring up all of our workers at once so that we don't have to launch them individually. We'll have one supervisor for Markdown which will watch all of our Markdown gen_servers, and another for all of our sanitiser gen_servers. To make managing them easier, we'll use another supervisor that will watch those two supervisors, re-launching them if they crash, and enabling us to bring both up, and all of the associated gen_servers, at once. So the plan is to have a formatter supervisor that will launch a markdown supervisor and a sanitiser supervisor, each of which will launch N workers, like this:

I'll show you working code, and try to explain the fuzzy bits, but I encourage you to research the functions that compose it in Erlang's own docs so that you really know what is going on.
external_program.pl

Since all of this relies on an actual external program, here's a shell to use for a Perl program:

```
1 #!/usr/bin/env perl -w
2
3 # turn on auto-flushing
4 $cfh = select (STDOUT);
5 $| = 1;
6 select ($cfh);
```

```
7
8 sub process {
9   return $_[0];
10 }
11
12 until(eof(STDIN)) {
13   my ($length_bytes,$length,$data,$return);
14
15   read STDIN,$length_bytes,4; # four-byte length-header
16   $length=unpack("N",$length_bytes);
17
18   read STDIN,$data,$length;
19
20   $return=&process($data);
21
22   print STDOUT pack("N",length($return));
23   print STDOUT $return;
24 }
```

It communicates with Erlang via stdin/stdout, and handles packets with 4-byte length headers (with network byte-order, since that's how Erlang sends it). That is just a stub program that returns the data coming in to it, but you can replace process with whatever you want to process/format the data. I have two versions of this, one that applies markdown, and one that sanitises input with HTML-TagFilter.
external_program.erl

Now the for the Erlang bits. First, we'll construct the gen_server to actually communicate with the Perl program. We'll communicate using a Port (which uses a unix pipe). We want to send data in atomic packets so that the Perl program knows when we're done sending data and so that we know when the Perl program is done responding. To do this, we'll send and receive data in packets of up to 232 bytes. That is, we'll send our Perl program a four-byte header (in network byte-order) indicating how much data is to be sent, then we'll send the data, and Perl will send back a four-byte header indicating how much data it is about to send, and then it will send that data. Erlang will handle the packets and headers on its side with the {packet,4} option to open_port/2, but we'll have to handle the packetising ourselves from the Perl side (already done in the above Perl program). This one is pretty generic, so you could use it to communicate with any outside program that can send data in packets, not just a formatter. Here's the code:

```
1 -module(external_program).
2 -behaviour(gen_server).
3
4 % API
5 -export([start_link/3,
6         filter/2]).
7
8 % gen_server callbacks
9 -export([init/1,
10         handle_call/3,
11         handle_cast/2,
12         handle_info/2,
13         code_change/3,
14         terminate/2]).
15
16 -record(state,{port}).
17
18 start_link(Type,Id,ExtProg) ->
19   gen_server:start_link({local,Id},?MODULE,{Type,ExtProg},_Options=[]).
20
21 handle_info({'EXIT', Port, Reason}, #state{port = Port} = State) ->
22   {stop, {port_terminated, Reason}, State}.
23 terminate({port_terminated, _Reason}, _State) ->
24   ok;
25 terminate(_Reason, #state{port = Port} = _State) ->
26   port_close(Port).
27 handle_cast(_Msg, State) ->
28   {noreply, State}.
29 code_change(_OldVsn, State, _Extra) ->
30   {ok, State}.
31
32 init({Type,ExtProg}) ->
33   process_flag(trap_exit, true),
34   ok=pg2:create(Type), % idempotent
35   ok=pg2:join(Type,self()),
36   Port = open_port({spawn, ExtProg}, [binary,{packet,4}]),
37   {ok, #state{port = Port}}.
38
```

```
39 handle_call(_Command={filter,Data},_From,#state{port=Port}=State) ->
40   port_command(Port,Data),
41   receive {Port,{data,Data2}} ->
42       {reply,Data2,State}
43    end.
44
45 filter(Type,Data) ->
46   gen_server:call(pg2:get_closest_pid(Type),{filter,Data}).
```

And here's the blow-by-blow. Skip ahead to externalprogram_supervisor.erl if you understand it. First, we start our module with

```
2 -behaviour(gen_server)
```

This tells the compiler to warn us if we forget any functions for call-back that gen_server is expecting to find. I won't introduce the exports here, I'll introduce the functions we come to them.

We use a record to encode our state, even though the only item in our state is the connection to the external program

```
16 -record(state,{port}).
```

We do this in case we later decide to add anything to the state, it will save us from modifying all of the function headers that refer to the state. Remember that our entire state between calls must go into this record, so if we want to later add information like the time the program was launched (for statistical tracking), or the system user that launched it, this is where it goes.

start_link/3 is how we'll start our gen_server. It's the function that we call from another module (our supervisor), not one called as a gen_server callback, so it can look however we want, but our supervisor is expecting it to return {ok,Pid}.

```
18 start_link(Type,Id,ExtProg) ->
19   gen_server:start_link({local,Id},?MODULE,{Type,ExtProg},_Options=[]).
```

It calls gen_server:start_link/4, which takes:

* {local,Id}: This is how gen_server will register the PID of the server. local means that it will only be registered on the local node, and Id is an atom passed to us by our supervisor that will appear in

process listings. We don't use this anywhere, it's just nice to have for viewing in appmon and other tools, and it's optional (gen_server:start_link/3 takes only the other three arguments)

* the call-back module (i.e. this module)
* the arguments to the gen_server's init/1 function (which we'll get to)
* any special options to gen_server (we don't use any)

It returns {ok,Pid}, where Pid is the Pid of the launched process. While we used gen_server:start_link/4, which takes registered name for the gen_server, we don't use it to address the gen_server. Instead, we're going to use the pg2 group name that we set in init/1 below. This is because we're going to launch several copies of this gen_server, so communicating directly with it isn't practical.

handle_info/2 is a callback function that is called whenever our server receives a message that gen_server doesn't understand.

```
21 handle_info({'EXIT', Port, Reason}, #state{port = Port} = State) ->
22   {stop, {port_terminated, Reason}, State}.
```

The only message that we handle is the Port indicating that it has shut down (for instance, if it crashes). In this case, there's nothing that we can do anymore (since our whole reason for being is to communicate with this Port), so we signal to our caller that we intend to shut down, and gen_server will shut us down, which includes calling our terminate/2 function and do other shutdown tasks.

terminate/2 is a callback function that gen_server calls whenever it is shutting us down for any reason so that we can free up any resources that we have (like our Port). It passes us the Reason for shutdown and the last State of the server

```
23 terminate({port_terminated, _Reason}, _State) ->
24   ok;
25 terminate(_Reason, #state{port = Port} = _State) ->
26   port_close(Port).
```

We have two clauses to handle a shutdown. If we are shutting down because our port terminated (because handle_info said so), then we just let gen_server finish shutting us down. If we shut down for any other reason, then we close the Port before we die so that it's not left open forever. terminate's return value is ignored.

handle_cast/2 handles asynchronous messages to our server

```
27 handle_cast(_Msg, State) ->
28   {noreply, State}.
```

gen_servers can handle two types of in-band messages: synchronous messages generated by calls to gen_server:call/2, and asynchronous messages generated by gen_server:cast/2. (gen_server handles the difference.) Since we're only handling synchronous messages, we just don't return a reply if we receive any asynchronous messages, but if you really wanted you could make a synchronous call and send it as a response

code_change/3 is called when our code is being dynamically replaced

```
29 code_change(_OldVsn, State, _Extra) ->
30   {ok, State}.
```

Erlang supports replacing code while the server is running, with no downtime. In the event that the code is replaced, a gen_server temporarily stops responding to messages (enqueueing them as they come in), waits for the code to be replaced, calls the call-back module's new code_change function, picks up the new state, and starts running again. This could be used if we were to upgrade our code to one that required a new member of the state record, for instance, to upgrade the running state to the state that the new version uses. (If the function returns anything but {ok,NewState}, the gen_server crashes, which would be fine in our case since our supervisor would just restart it under the new code anyway.) Unless we plan to convert the state of the server, which for now we don't, we just return {ok,State}

init/1 is called by gen_server just before the server-loop is run.

```
32 init({Type,ExtProg}) ->
33   process_flag(trap_exit, true),
34   ok=pg2:create(Type), % idempotent
35   ok=pg2:join(Type,self()),
36   Port = open_port({spawn, ExtProg}, [binary,{packet,4}]),
37   {ok, #state{port = Port}}.
```

Its argument comes from gen_server:start_link, which we called in our own start_link/3 function, and is our Type (which becomes our pg2 group name) and the path to the external program that we want to launch (the Perl programs mentioned above). We first indicate that we want to trap exits, so that we can receive 'EXIT' messages from our Port in case it dies. These messages are not recognised by gen_server, so they get passed to our handle_info/2 callback function. Then we create our pg2 group

(this operation is idempotent, which means that doing it several times has the same effect as doing it once), and join this process to the group. Then we open up the external program itself using open_port. It takes a tuple of {Type,Path}, which we use to indicate that we are launching an external program (the spawn type) and where to find it, and a list of options, which we use to tell it that we want to communicate using binaries instead of lists, and that we want it to handle communication to our Port with packets with a length-header of four bytes. Then we tell gen_server that all is well and to start the server with a state record containing a reference to our Port.

handle_call/2 is called whenever we receive a synchronous request from an outside caller, this is the actual guts of our server.

```
39 handle_call(_Command={filter,Data},_From,#state{port=Port}=State) ->
40   port_command(Port,Data),
41   receive {Port,{data,Data2}} ->
42     {reply,Data2,State}
43    end.
```

We just receive the command (which is a {filter,Data} tuple; it doesn't have to be, but that's how we're going to call it later), send it to the Port, get the reply from the Port, and tell gen_server to reply that back to the caller. We didn't have to write any of the server code, just the implementation of our code.

Finally, here's how we actually communicate with our server after it's launched:

```
45 filter(Type,Data) ->
46   gen_server:call(pg2:get_closest_pid(Type),{filter,Data}).
```

We make a synchronous call to the "closest" Pid that is a member of the pg2 group that our programs join when launching. ("Closest" just means that pg2 will first look on the local node before looking on other nodes.) pg2:get_closest_pid randomises which worker it returns within a group, so this should spread the load if we get many simultaneous requests. This could be expanded to try again in the event of a timeout (assuming that the timed-out server will crash and be re-launched, or that pg2:get_closest_pid will return a different Pid next time)

At this point you should be able to create and call a gen_server
externalprogram_supervisor.erl

Now for the supervisor. Again, we'll keep this generic. It handles multiple external_program workers,

launching NumWorkers (passed to start_link/3). We'll use the (aptly named) supervisor behaviour. supervisor launches a group of processes, watches them, and re-launches them if they die. It also provides a way to bring up or down a group of processes all at once, just by launching or shutting down the supervisor. I'm going to explain the functions of order for clarity.

```
1 -module(externalprogram_supervisor).
2 -behavior(supervisor).
3
4 % API
5 -export([start_link/3]).
6
7 % supervisor callbacks
8 -export([init/1]).
9
10 start_link(Type,NumWorkers,ExtProg) ->
11   supervisor:start_link({local,
12                          list_to_atom(atom_to_list(Type) ++ "_supervisor")},
13                         ?MODULE,
14                         {Type,NumWorkers,ExtProg}).
15
16 init({Type, NumWorkers,ExtProg}) ->
17   {ok,
18    {{one_for_one,
19      3,
20      10},
21     [begin
22         Id=list_to_atom(atom_to_list(Type) ++ integer_to_list(Which)),
23         {Id,
24          {external_program,
25           start_link,
26           [Type,Id,ExtProg]},
27          permanent,
28          _Timeout=10*1000,
29          worker,
30          [external_program]}
31        end
32      || Which <- lists:seq(1,NumWorkers)]}}.
33
```

Again, -behaviour(supervisor). tells the compiler to warn us if we forget any callback functions required by supervisor.

supervisor only has one callback, init/1:

```
16 init({Type, NumWorkers,ExtProg}) ->
17   {ok,
18    {{one_for_one,
19      3,
20      10},
21     [begin
22        Id=list_to_atom(atom_to_list(Type) ++ integer_to_list(Which)),
23        {Id,
24         {external_program,
25          start_link,
26          [Type,Id,ExtProg]},
27         permanent,
28         _Timeout=10*1000,
29         worker,
30         [external_program]}
31       end
32      || Which <- lists:seq(1,NumWorkers)]}}.
```

It's only called once when the supervisor is created (or re-started after crashing). It gets its arguments from supervisor:start_link, which we call from our local start_link (that we'll explain in a moment). init is expected to return a restart strategy and a list of Childspecs, where a Childspec looks like (straight from the Erlang supervisor documentation):

```
{Id,StartFunc,Restart,Shutdown,Type,Modules},
  Id = term()
  StartFunc = {M,F,A}
    M = F = atom()
    A = [term()]
  Restart = permanent | transient | temporary
  Shutdown = brutal_kill | int()>=0 | infinity
  Type = worker | supervisor
  Modules = [Module] | dynamic
```

Module = atom()

You can read up on what all of that means in the supervisor documentation, but we return a list of them consisting of NumWorkers items (see the list-comprehension) such that each is launched by calling external_program:start_link/3 (specified by the StartFunc in the Childspec), passing each worker their Id, Type and ExtProg. The Id just an atom that we generate from the Type and the worker-number (which is just determined by their launch-order) that you'll recall being passed to gen_server to register the worker's PID so that in debug listings we can see markdown3 instead of <0.15.3>

The supervisor itself is created by calling our local start_link/3, which calls supervisor:start_link/2 (which actually creates the supervisor, calls our init, etc). This supervisor will be called from formatter_supervisor, defined below, but you could call it done at this point and have a working implementation of an N-worker external program.

```
1> externalprogram_supervisor:start_link(my_server,5,"/usr/local/bin/external_program.pl").
<0.1976.0>
2> pg2:get_members(my_server).
[<0.1981.0>,<0.1980.0>,<0.1979.0>,<0.1978.0>,<0.1977.0>]
3> gen_server:call(pg2:get_closest_pid(my_server),{filter,<<"This is some random text!">>}).
<<"This is some random text!">>
```

formatter_supervisor.erl

Our original goal was to have N workers running for two external programs. So far all of the code we've written has been generic to the external program, but now we're going to get into (very slightly) more specific stuff. The formatter_supervisor will be a supervisor that will launch and watch two supervisors, one for Markdown and one for our santiser. They'll both run the same externalprogram_supervisor code, they'll just be launched with different arguments.

```
1 -module(formatter_supervisor).
2 -behavior(supervisor).
3
4 % External exports
5 -export([start_link/0]).
6
7 % supervisor callbacks
```

```
8 -export([init/1]).
9
10 start_link() ->
11   supervisor:start_link(?MODULE, '_').
12
13 % formatters are assumed to create pg2s registered under their
14 % type-names (at the moment, markdown and sanitiser)
15 init(_Args) ->
16   {ok,
17    {{one_for_one,
18      3,
19      10},
20     [{markdown_supervisor,
21       {externalprogram_supervisor,
22        start_link,
23        [markdown,
24         myapp_config:markdown_workers(),
25         myapp_config:markdown_program()]},
26       permanent,
27       _Timeout1=infinity,
28       supervisor,
29       [externalprogram_supervisor]},
30      {sanitiser_supervisor,
31       {externalprogram_supervisor,
32        start_link,
33        [sanitiser,
34         myapp_config:sanitiser_workers(),
35         myapp_config:sanitiser_program()]},
36       permanent,
37       _Timeout2=infinity,
38       supervisor,
39       [externalprogram_supervisor]}]}}.
```

That looks almost the same as our externalprogram_supervisor, except that the ChildSpecs returned by init are different. We don't have a local version of start_worker_link, we just call externalprogram_supervisor:create_link (since there's nothing we need to do with it after launching it). The ChildSpec list looks like this:

```
20    [{markdown_supervisor,
21      {externalprogram_supervisor,
22       start_link,
23       [markdown,
24        myapp_config:markdown_workers(),
25        myapp_config:markdown_program()]},
26      permanent,
27      _Timeout1=infinity,
28      supervisor,
29      [externalprogram_supervisor]},
30     {sanitiser_supervisor,
31      {externalprogram_supervisor,
32       start_link,
33       [sanitiser,
34        myapp_config:sanitiser_workers(),
35        myapp_config:sanitiser_program()]},
36      permanent,
37      _Timeout2=infinity,
38      supervisor,
39      [externalprogram_supervisor]}]}}.
```

Nothing too special there, except that we pass in the Type atom to externalprogram_supervisor:start_link (markdown or sanitiser), which becomes the name of the pg2 group. The shutdown timeout is set to infinity for supervisors of supervisors (as recommended by the Erlang docs) to give the indirect children all time to shut down, and we've indicated that the worker type is supervisor, so that Erlang knows that we're building a supervision tree. Of course, the arguments to externalprogram_supervisor:start_link assume that myapp_config:markdown_workers/0 and friends have been defined, but you could just replace those with literal values, too. myapp_config:markdown_workers/0 is assumed to return an integer (in my case, five), which is how many copies of the program you want to launch, and myapp_config:markdown_program/0 returns the full path to the program to launch (like "/usr/local/myapp/lib/markdown.pl"). formatter_supervisor is the supervisor that will be started from my application's main supervisor.
format.erl

The actual communication with the programs should be pretty easy now, we just ask external_program to do its magic. Here it is:

```
1 -module(format).
```

```
2
3 % formatters are defined and launched by the formatter_supervisor
4
5 -export([markdown/1,
6         sanitise/1]).
7
8 markdown(Data) ->
9   external_program:filter(markdown,Data).
10
11 sanitise(Data) ->
12   external_program:filter(sanitiser,Data).
```

It should work with lists or a binaries. Add a use Text::Markdown; to the top of our Perl module module (at the beginning of this entry) and replace its process function with:

```
10 sub process {
11   return Text::Markdown::markdown($_[0]);
12 }
```

Launch our formatter_supervisor with formatter_supervisor:start_link() and now we have a working Markdown implementation:

```
(name@node)106> format:markdown("I *really* like cookies").
<<"<p>I <em>really<em> like cookies<p>\n">>
```

1.42 看erlang库实现 理解erlang函数编程

发表时间: 2008-05-21 关键字: erlang 库

erlang的list 和 string 库实现 有我们自己实现函数的时候可以模仿的很多手法， 看熟悉了 容易在头脑里面自然反射代码的实现，俺学c的时候就是学会了字符串操作的时候感觉功力大增。

1.43 看例子写 testserver 和commontest 测试案例

发表时间: 2008-05-22 关键字: testserver commontest

R12发步的时候 附带了testserver和commontest，有文档但是没有详细的叫你如何写测试案例。今天发现官网上有早期的测试案例（R9）的，可以参考下：
http://www.erlang.org/project/test_server/index.html

有3种测试案例：
test_server-3.1.1.tar.gz The test server application.
emulator-tests-2004-05-26.tar.gz Test suites for the emulator.
stdlib-tests-2004-05-26.tar.gz Test suites for the stdlib application.

多写测试案例对于系统的稳定非常的有帮助哦。

1.44 Erlang/OTP R12B-3 released

发表时间: 2008-06-12 关键字: erlang/otp r12b-3 released

Erlang/OTP R12B-3 released

The third service release for Erlang/OTP R12B has been released. (June 11, 2008)

最大的卖点：

An experimental module "re" is added to the emulator which interfaces a publicly available regular expression library for Perl-like regular expressions (PCRE). The interface is purely experimental and *will* be subject to change.

根据这个eep做的 http://www.erlang.org/eeps/eep-0011.html

1.45 gen_tcp 应对对端半关闭

发表时间: 2008-06-13 关键字: gen_tcp shutdown exit_on_close

当tcp对端调用shutdown(RD/WR) 时候， 宿主进程默认将收到{tcp_closed, Socket}消息， 如果这个行为不是你想要的，那么请看：

shutdown(Socket, How) -> ok | {error, Reason}

Types:

Socket = socket()
How = read | write | read_write
Reason = posix()

Immediately close a socket in one or two directions.

How == write means closing the socket for writing, reading from it is still possible.

To be able to handle that the peer has done a shutdown on the write side, the {exit_on_close, false} option is useful.

这样就不会被强制退出了。

1.46 ASN.1协议 适合我们用吗？

发表时间: 2008-06-16 关键字: asn.1 encode decode

R12B发布以后 号称ASN.1解码速度快了好多。前天仔细看了文档，感觉用在自有的系统去不错。ASN.1的结构还是很紧凑的，BER,PER,BER_BIN,PER_BIN都支持。特别是支持选择性decode,这个feature非常赶兴趣。用在自己的项目里面就省去了很多协议编解码的繁琐和易错,毕竟asn compiler也做了几万行 我相信他的性能。

1.47 Erlang Source via Git 终于可以即使获取代码了

发表时间: 2008-08-04 关键字: source git

Erlang Source via Git 2008-08-01

From the erlang-questions mailing list:

"I find Matthew Foemmel' s http://github.com/mfoemmel/erlang-otp/tree git repository to be very handy for browsing the OTP source, working out when features were added (the history goes back to R6B-0) and so on. Erlang/OTP packaging for debian/macports/... is much easier to track against this repository too.

Thanks to Geoff Cant for pointing this out. I wonder what would happen if people started submitting patches to this tree? Hmmm.....

这般老头也是固执，我们都等这么久了。。。

1.48 CN Erlounge III - 发起

发表时间: 2008-11-04 关键字: cn erlounge iii 发起

CN Erlounge III - 发起

1. 时间：2008-12-20 ~ 2008-12-21，为期2天
2. 地点：上海（详细地址待确定）
3. 人物：面向 Erlang 中国社区，但不排斥其他任何对 Erlang、分布式、多核等话题感兴趣的人。
4. 议题： Erlang 语言相关技术、Erlang 应用、Erlang 与其他语言协作、分布式、多核等等。
5. 会议主持：ECUG 会务组

会议形式

1. 多数时间由交流会讲师针对某个 Topic 进行论述，其他人提问（Q&A）方式交流。
2. 留出一小段时间，安排沙龙式的对等交流机会。

会议规则

1. 会议的讲师报销来回路费和住宿（投稿并申请成为讲师）。点击这里可以查看已经确定的讲师名单。
2. 任何人可报名免费参与听讲（注册并申请参加本会议）。

注：由于场地限制，我们可能没法接受所有的与会申请，请谅解。如果名额已满，我们会回信说明。
重要时间点

1. 普通参会者报名截止日期： 2008-12-1
2. 讲师报名&投稿截止日期：2008-12-9
3. 详细会议议程安排公布：2008-12-13
4. 会议日期：2008-12-20 ~ 2008-12-21

1.49 小小的参数设置耗费大量内存

发表时间: 2008-11-04 关键字: +p

> I only have 3Gb of memory in my accelerator at Joyent and I'm trying
> to launch 10 nodes. beam starts with RSS of about 135Mb and quickly
> goes to 500Mb+.
>
> How can I minimize the footprint of my running nodes?
>
> I'm starting slaves like this:
>
> common_args() ->
> "+K true +P 134217727 -smp disable".
>

You will lower the memory footprint significantly by removing +P option or using a much lower value for it (the number of simultaneously existing processes that
the system can handle). I am pretty sure you don't need the maximum value there, you will run out of memory in one node before you reach 134 millions of processes and I am
pretty sure that your system will not handle millions of simultaneous poker players (in one Erlang-node)
The value +P 134217727 will result in a memory requirement of 4 times 134217727 bytes = 536870908 bytes = 536 Mbytes for the Erlang node just to hold the process-table.

/Kenneth Erlang/OTP, Ericsson

536M内存呀， 我们完全用不到这么多进程， 所以系统微调真的要很小心。

1.50 R12B-5发布加入eunit 支持从archive读取代码

发表时间: 2008-11-06 关键字: r12b-5 eunit archive code

1. The eunit application (for unit testing of Erlang modules) by Richard Carlsson is now included in OTP.
2. There is now experimental support for loading of code from archive files. See the documentation of code, init,erl_prim_loader and escript for more info.
3. public_key first version.

还是R12的bug fix版本 没有太多新功能引入.

1.51 group_leader的设计和用途

发表时间: 2008-11-20 关键字: io group_leader spawn

一直对erlang的group_leader这个概念很困惑，因为*nix系统也有类似的名词但是只是和进程组管理有关系。查了很多文档才知道，erlang的group_leader的设计意图和作用，解释如下：

先看下着段代码运作：

```
log_group_leader(OGL) ->
    receive Msg ->
        io:format(user, "Got message ~p~n",[Msg]),
        OGL ! Msg,
        log_group_leader(OGL)
    end.

ioclient(NGL) ->
    group_leader(NGL, self()),
    io:format("Hello World~n"),
    io:format("Hello again~n").

iotest() ->
    OldGroupLeader = group_leader(),
    NewGroupLeader = spawn(?MODULE, log_group_leader, [OldGroupLeader]),
    spawn(?MODULE, ioclient, [NewGroupLeader]).
```

说白了group_leader就是决定erlang的io控制台的输出到那个进程。

这个特性很有帮助。我们在做服务器程序的时候 会有大量的诊断信息通过类似printf打印需要输出到控制台查看。 代码里东一块西一块都是诊断代码， 而且一旦程序调试完毕 我们可能不在需要这些信息污染环境。这个是单机的情况，对于分布式的情况就更复杂，很难把其他主机上打出的诊断信息汇总到一个地方集中查看。

group_leader就是解决这个问题的。 erlang里面的io:format之类的函数执行的时候最早输出会被重定向到该进程的group_leader进程去，而且进程是位置无关的，也就是说在其他主机上的信息都可以汇总。

有了这个特性 比如说我在其他的机器上执行条rpc命令 这个命令的结果是会被截获 传送回来的 而不是要到目标主机的终端上显示。

group_leader是继承的,每当spawn一个进程的时候，会自动继承父进程的这个属性。rpc的实现上也很大努力保证了这个语义不变。

在实践中也要注意这个特性的副作用，就是： 你用rpc执行的命令 就是想在目标机器上显示结果 但是看不到 因为被截获了。

1.52 What is the relation between async threads and SMP

发表时间: 2008-11-29 关键字: async thread smp

The async thread pool has nothing with SMP todo. The asynch threads are only used by the file driver and by user written drivers that specifically uses the thread pool. The file driver uses this to avoid locking of the whole Erlang VM for a longertime period in case of a lengthy file operation. The asynch threads was introduced long before the SMP support in the VM and works for the non SMP VM as well. In fact the asynch threads are even more important for a non SMP system because without it a lengthy file operation will block the whole VM.

1.53 Async线程pool及其作用

发表时间: 2008-12-03 关键字: async pool driver +a smp

erlang能够利用多核心cpu的基础设施有2个 1. 进程调度器 2. async 线程池。

其中 async 线程池主要设计用来 能够在driver里面异步的执行费时操作， 因为driver是在调度器里面调用的 不过费时操作的话 会block掉整个调度器 而且调度器资源有限。

细节参见 erl_async.c

Driver API:

Asynchronous calls
The latest Erlang versions (R7B and later) has provision for asynchronous function calls, using a thread pool provided by Erlang. There is also a select call, that can be used for asynchronous drivers.

long driver_async (ErlDrvPort port, unsigned int* key, void (*async_invoke)(void*), void* async_data, void (*async_free)(void*))

void ready_async(ErlDrvData drv_data, ErlDrvThreadData thread_data)

其中erlang自己的file driver就大量依赖于异步线程池 所以如果文件操作密集型的程序可以考虑加大池的数量。

erl有2个参数和这个池有关：

+a size
Suggested stack size, in kilowords, for threads in the async-thread pool. Valid range is 16-8192 kilowords. The default suggested stack size is 16 kilowords, i.e, 64 kilobyte on 32-bit architectures. This small default size has been chosen since the amount of async-threads might be quite large. The default size is enough for drivers delivered with Erlang/OTP, but might not be sufficiently large for other dynamically linked in drivers that use the driver_async() functionality. Note that the value passed is only a suggestion, and it might even be ignored on some platforms.
（大量池的时候 要考虑栈内存的影响 缩小这个值）

+A size
Sets the number of threads in async thread pool, valid range is 0-1024. Default is 0.

总结起来就是如果需要大量的费时操作 又需要原生的Api来做的话 可以考虑写个driver来利用这个特性。

1.54 Rickard Green erlang smp的主要贡献者

发表时间: 2008-12-06 关键字: rickard green erlang smp

The SMP support has mainly been designed and implemented by Rickard Green, Tony Rogvall (mostly ets), Mikael Pettersson (mostly optimized synchronization primitives, and timer thread), and Patrik Nyblom (mostly dynamic drivers, and Windows port). Also Bj<F6>rn Gustavsson and Raimo Niskanen have contributed.

大量的和smp相关的东西都是他实现的，特别是在锁处理方面非常有经验，致敬。

1.55 Erlang分布的核心技术浅析

发表时间: 2008-12-06 关键字: dist erts_net_message inet_tcp_dist inet_driver rpc

Erlang系统在我看来有3个特性 1. 分布 2. 多核心支持 3. fp。 这这3个特性中分布我认为是erlang最强大的,从第一个版本就支持到现在，相对非常成熟，而多核心支持是这几年才加进去的。

erlang的分布系统做到了2点 1.节点的位置无关性。 2. 对用户分布式透明的。具体体现就是node是靠名字识别的, 进程也是靠pid来识别。

分布系统就要实现节点间通讯，erlang也不列外。 erlang的节点通讯介质是可以替换的 目前官方版本支持tcp, ssl通讯。

可以用 -proto_dist xxxx来选择通道。 目前支持inet_ssl inet_tcp 用户很容易模仿这这2个通讯协议，写个自己的传输通道，就是要求这个通道是可靠的，不能丢失信息。

这几个实现防火墙友好：
{inet_dist_use_interface, ip_address()}
If the host of an Erlang node has several network interfaces, this parameter specifies which one to listen on. See inet(3) for the type definition of ip_address().
{inet_dist_listen_min, First}
See below.
{inet_dist_listen_max, Last}
Define the First..Last port range for the listener socket of a distributed Erlang node.

erlang的内核里面和分布相关的erl模块主要有net_kernel inet_tcp_dist inet_ssl_dist inet_tcp dist_util erlang(trap send/link等语义)。

当用户运行erl -sname xxxxx启动erlang系统的时候 kernel模块就会启动net_kernel和epmd模块。

epmd的作用是提供一个node名称到监听地址端口的映射。epmd值守在知名端口4369上.

net_kernel会启动proto_dist比如说inet_tcp_dist监听端口,同时把端口报到epmd. 这时候erts就准备好了节点通讯。

这时候另外一个节点要和我们通讯的时候，首先要连接，流程大概是这样的：
1. 根据节点名找到节点地址。
2. 查询节点的4369端口,也就是epmd,向它要节点对应的inet_tcp_dist监听的端口。

3. 发起连接, 握手，cookie认证，如果未授权就失败。
4. 记录节点名称和响应的信息。

所以节点要正常通讯要考虑到firewall和nat的影响，以及cookie正常。

具体的可以看另外一篇文章：http://mryufeng.javaeye.com/blog/120666

要给节点发消息首先要保证我们和节点联系过，也就是说我们的节点表里可以查到这个节点。

net_kernel要节点的可用性, 会定期发tick消息坚持节点的可达。对端节点也会主动发送nodeup,nodeup等消息，协助维护。

net_ticktime = TickTime
Specifies the net_kernel tick time. TickTime is given in seconds. Once every TickTime/4 second, all connected nodes are ticked (if anything else has been written to a node) and if nothing has been received from another node within the last four (4) tick times that node is considered to be down. This ensures that nodes which are not responding, for reasons such as hardware errors, are considered to be down.

节点有2种类型可见的和不可见的。erlang节点是可见的， c_interface写的节点不可见，因为c模块提供的节点能力有限。

erlang进程见通讯可以通过 1. pid 2.进程名称 来进行。 erlang系统很大的威力就在用elang实现了名称和pid的全局性维护。也就是说erlang做了个相当复杂的模块来解决名称失效 重复 查询功能。正常情况下名称是全局行的，也就是erlang的节点是全联通的，可以通过来调整。
auto_connect = Value
Specifies when no will be automatically connected. If this parameter is not specified, a node is always automatically connected, e.g when a message is to be sent to that node. Value is one of:

never
Connections are never automatically connected, they must be explicitly connected. See net_kernel(3).
once
Connections will be established automatically, but only once per node. If a node goes down, it must thereafter be explicitly connected. See net_kernel(3).

erlang实现透明进程通信的关键点在于pid的变换：

pid {X,Y,Z} 在发到网络的时候发出去的格式是{sysname, Y, Z}
因为节点之前互相联系过 所以互相知道对方的sysname, 而且sysname在dist_entry里保存，当对端收到dec_pid的时候，用peer sysname 的查到在自己dist_entry里面的索引，然后用这个index 来构造新的pid，即{index, Y, Z}。

erlang给一个pid发消息的时候， 首先检查Pid是本地的还是外部的， 如果是外部的，则进行上面的变换，然后通过inet_tcp_dist模块，沿着inet_tcp, inet_drv这条线发送出去。

这条消息到达对端的时候 inet_drv首先受到这条消息， 照理说应该提交给inet_tcp, 然后再到inet_tcp_dist，net_kernel来处理.但是erlang为了效率的考虑做了个折中。在inet_drv里面driver_output*中检查消息的类型如果是dist来的消息，就给erts_net_message来处理。

这个erts_net_message处理以下几个重要消息：

#define DOP_SEND 2
#define DOP_EXIT 3
#define DOP_UNLINK 4
#define DOP_NODE_LINK 5
#define DOP_REG_SEND 6
#define DOP_GROUP_LEADER 7
#define DOP_EXIT2 8
#define DOP_SEND_TT 12
#define DOP_EXIT_TT 13
#define DOP_REG_SEND_TT 16
#define DOP_EXIT2_TT 18
#define DOP_MONITOR_P 19
#define DOP_DEMONITOR_P 20
#define DOP_MONITOR_P_EXIT 21

如果是DOP_SEND的话，就把message放到变换好的进程的队列中去。

这个核心的功能,由beam的c模块(dist.c, erl_node_tables.c, io.c)和net_kernel模块一起实现。然后在这底层的原语上进一步实现了如rpc 这样的上层模块，进一步方便了用户.

用erlsnoop http://mryufeng.javaeye.com/blog/167695 可以观察到上面的通讯，有助于理解系统的运作。

1.56 erlang内置的port相关的驱动程序

发表时间: 2008-12-07 关键字: fd_driver_entry vanilla_driver_entry spawn_driver_entry

erlang能够利用多核心的优势不仅体现在多线程的smp调度器，更在port上面上体现。通过执行外部程序，接管它的输入输出, 实现了安全性和充分利用cpu计算资源。

erlang的io设计体现了unix一切以文件为中心的思想。在port设计上也是.这3个驱动程序都是把相关的东西转化成文件句柄，登记到poll上面，利用强大的IO poll来实现上层语义的整合。

ErlDrvEntry spawn_driver_entry;

{spawn, Command}
Starts an external program. Command is the name of the external program which will be run. Command runs outside the Erlang work space unless an Erlang driver with the name Command is found. If found, that driver will be started. A driver runs in the Erlang workspace, which means that it is linked with the Erlang runtime system.
When starting external programs on Solaris, the system call vfork is used in preference to fork for performance reasons, although it has a history of being less robust. If there are problems with using vfork, setting the environment variable ERL_NO_VFORK to any value will cause fork to be used instead.

ErlDrvEntry fd_driver_entry;

{fd, In, Out}
Allows an Erlang process to access any currently opened file descriptors used by Erlang. The file descriptor In can be used for standard input, and the file descriptor Out for standard output. It is only used for various servers in the Erlang operating system (shell and user). Hence, its use is very limited.

ErlDrvEntry vanilla_driver_entry;

官方没有文档。它的作用是直接打开文件进行读写。

1.57 smp下async_driver的用途和匠心

发表时间: 2008-12-07 关键字: async_driver

/* INTERNAL use only */
struct erl_drv_entry async_driver_entry;

erlang beam里面有个async_driver设计的比较有意思。 在多处理器beam里面，线程间需要通讯。常规的情况下用condition再有个队列之类的辅助。 但是这个设计不能充分利用io的优势，而且使用起来很麻烦。
async_driver就是设计给这个用途的。 它建立一对pipe，把它登记到poll去，生产者要发送消息的时候，就往pipe写。 poll机制就会通知消费者,消费者根据读出的东西进行动作. ACE框架也大量用这种模式，看来大同世界一般同。但是erlang做的更彻底，概念也更简单。

1.58 erlang的IO高效不是传说

发表时间: 2008-12-07 关键字: io epoll

[root@haserver ~]# erl -smp +K true
Erlang (BEAM) emulator version 5.6.5 [source] [smp:2] [async-threads:0] [hipe] [kernel-poll:true]

Eshell V5.6.5 (abort with ^G)
1> erlang:system_info(check_io).
[{name,erts_poll},
{primary,epoll},
{fallback,poll},
{kernel_poll,epoll},
{memory_size,9064},
{total_poll_set_size,2},
{fallback_poll_set_size,0},
{lazy_updates,true},
{pending_updates,0},
{batch_updates,false},
{concurrent_updates,true},
{max_fds,1024}]

erl_poll.c是erlang的poll实现。对epoll的lazy update和concurrent_update进行了大量的优化，batch update在freebsd下管用。我讲过的框架包括libevent,ACE，haproxy等等。只有haproxy进行了lazy update但是没有concurrent方面的努力。

erlang的IO读写大多采用gather write, scatter read 方式,在语法方面就有强大的优势，列表操作隐含的就是这个语义。

致敬，多年前就这门挖空心思高性能！

1.59 erlang的timer和实现机制

发表时间: 2008-12-07 关键字: timer driver

对于任何网络程序来讲，定时器管理都是重头戏。erlang更是依赖于定时器。基础的timer主要是由time.c erl_time_sup.c实现。timer是基于time wheel的实现，支持time jump detection and correction。 上层的erl_bif_timer.c io.c中实现。

erl +c

Disable compensation for sudden changes of system time.

Normally, erlang:now/0 will not immediately reflect sudden changes in the system time, in order to keep timers (including receive-after) working. Instead, the time maintained by erlang:now/0 is slowly adjusted towards the new system time. (Slowly means in one percent adjustments; if the time is off by one minute, the time will be adjusted in 100 minutes.)

When the +c option is given, this slow adjustment will not take place. Instead erlang:now/0 will always reflect the current system time. Note that timers are based on erlang:now/0. If the system time jumps, timers then time out at the wrong time.

erlang使用timer有3中方式：

1. 语法层面的 receive ... after ...

这个是opcode实现的，一旦timeout立即把process加到调度队列，使用频度比较高。

2. bif

erlang:send_after(Time, Dest, Msg) -> TimerRef

erlang:start_timer(Time, Dest, Msg) -> TimerRef

这个一旦timeout就给Dest发送一条消息，使用频度不是很高。

3.driver层面的。

int driver_set_timer(ErlDrvPort port, unsigned long time);

inet_driver大量使用这个api. tcp/udp进程需要超时处理，所以有大量的连接的时候这种timer的数量非常大。定时器超时后把port_task加到调度队列。

定时器的最早超时时间用于poll的等待时间。

整个定时器由bump_timer来驱动。bump_timer是在schedule的时候不定期调用的。总之使用timer的时候要小心，因为timer实在scheduler的线程里面调用的，不能做非常耗时的操作，否则会阻塞调度器。

1.60 erlang的IO调度

发表时间: 2008-12-07 关键字: erl_port_task

erlang的调度有2种: 1. 进程调度 2. IO调度。网络程序的事件来源基本上只有2种:IO和定时器。IO事件有可能是大量的， 不可预期的，所以在设计上要考虑和进程调度平衡。erlang的
erl_port_task就是为这个目标设计的。

poll检查到io时间的时候，会回调iread和oready函数。这2个会把这个队列加到porttask的调度队列去。

```
static ERTS_INLINE void
iready(Eterm id, ErtsDrvEventState *state)
{
    if (erts_port_task_schedule(id,
&state->driver.select->intask,
ERTS_PORT_TASK_INPUT,
(ErlDrvEvent) state->fd,
NULL) != 0) {
stale_drv_select(id, state, DO_READ);
    }
}

void
ERTS_CIO_EXPORT(erts_check_io)(int do_wait)
{
...
if ((revents & ERTS_POLL_EV_IN)
    || (!(revents & ERTS_POLL_EV_OUT)
&& state->events & ERTS_POLL_EV_IN))
    iready(state->driver.select->inport, state);
else if (state->events & ERTS_POLL_EV_OUT)
    oready(state->driver.select->outport, state);
    }
...
}
```

```
/*
* Run all scheduled tasks for the first port in run queue. If
* new tasks appear while running reschedule port (free task is
* an exception; it is always handled instantly).
*
* erts_port_task_execute() is called by scheduler threads between
* scheduleing of processes. Sched lock should be held by caller.
*/

int erts_port_task_execute(void)
{
...
switch (ptp->type) {
case ERTS_PORT_TASK_FREE: /* May be pushed in q at any time */
    erts_smp_tasks_lock();
    if (io_tasks_executed) {
ASSERT(erts_port_task_outstanding_io_tasks >= io_tasks_executed);
erts_port_task_outstanding_io_tasks -= io_tasks_executed;
    }
    goto free_port;
case ERTS_PORT_TASK_TIMEOUT:  /*driver层面的timer超时时间*/
    erts_port_ready_timeout(pp);
    break;
case ERTS_PORT_TASK_INPUT: /*IO input*/
   erts_port_ready_input(pp, ptp->event);
    io_tasks_executed++;
    break;
case ERTS_PORT_TASK_OUTPUT: /*IO output*/
   erts_port_ready_output(pp, ptp->event);
    io_tasks_executed++;
    break;
case ERTS_PORT_TASK_EVENT:
    erts_port_ready_event(pp, ptp->event, ptp->event_data);
    io_tasks_executed++;
    break;
default:
    erl_exit(ERTS_ABORT_EXIT,
     "Invalid port task type: %d\n",
```

```
    (int) ptp->type);
    break;
}
...
}


void
erts_port_ready_input(Port *p, ErlDrvEvent hndl)
{
    ERTS_SMP_CHK_NO_PROC_LOCKS;
    ERTS_SMP_LC_ASSERT(erts_lc_is_port_locked(p));

    ASSERT((p->status & ERTS_PORT_SFLGS_DEAD) == 0);

    if (!p->drv_ptr->ready_input)
missing_drv_callback(p, hndl, DO_READ);
    else {
(*p->drv_ptr->ready_input)((ErlDrvData) p->drv_data, hndl);  /* 真正干活的地方 */
/* NOTE some windows drivers use ->ready_input for input and output */
if ((p->status & ERTS_PORT_SFLG_CLOSING) && erts_is_port_ioq_empty(p)) {
    terminate_port(p);
}
    }
}
```

在erlang的schedule里会在适当的时间执行erts_port_task_execute消耗掉IO事件。执行的时间和次数主要和process平衡。

1.61 erlang进程的优先级

发表时间: 2008-12-07 关键字: process priority

```
/* process priorities */
#define PRIORITY_MAX        0
#define PRIORITY_HIGH       1
#define PRIORITY_NORMAL     2
#define PRIORITY_LOW        3
#define NPRIORITY_LEVELS    4
```

process_flag(priority, Level)

This sets the process priority. Level is an atom. There are currently four priority levels: low, normal, high, and max. The default priority level is normal. NOTE: The max priority level is reserved for internal use in the Erlang runtime system, and should not be used by others.

Internally in each priority level processes are scheduled in a round robin fashion.

Execution of processes on priority normal and priority low will be interleaved. Processes on priority low will be selected for execution less frequently than processes on priority normal.

When there are runnable processes on priority high no processes on priority low, or normal will be selected for execution. Note, however, that this does not mean that no processes on priority low, or normal will be able to run when there are processes on priority high running. On the runtime system with SMP support there might be more processes running in parallel than processes on priority high, i.e., a low, and a high priority process might execute at the same time.

When there are runnable processes on priority max no processes on priority low, normal, or high will be selected for execution. As with the high priority, processes on lower priorities might execute in parallel with processes on priority max.

Scheduling is preemptive. Regardless of priority, a process is preempted when it has consumed more than a certain amount of reductions since the last time it was selected for execution.

NOTE: You should not depend on the scheduling to remain exactly as it is today. Scheduling, at least on the runtime system with SMP support, is very likely to be modified in the future in order to better utilize available processor cores.

There is currently no automatic mechanism for avoiding priority inversion, such as priority inheritance, or priority ceilings. When using priorities you have to take this into account and handle such scenarios by yourself.

Making calls from a high priority process into code that you don't have control over may cause the high priority process to wait for a processes with lower priority, i.e., effectively decreasing the priority of the high priority process during the call. Even if this isn't the case with one version of the code that you don't have under your control, it might be the case in a future version of it. This might, for

example, happen if a high priority process triggers code loading, since the code server runs on priority normal.

Other priorities than normal are normally not needed. When other priorities are used, they need to be used with care, especially the high priority must be used with care. A process on high priority should only perform work for short periods of time. Busy looping for long periods of time in a high priority process will most likely cause problems, since there are important servers in OTP running on priority normal.

process_flag(save_calls, N)

When there are runnable processes on priority max no processes on priority low, normal, or high will be selected for execution. As with the high priority, processes on lower priorities might execute in parallel with processes on priority max.

N must be an integer in the interval 0..10000. If N > 0, call saving is made active for the process, which means that information about the N most recent global function calls, BIF calls, sends and receives made by the process are saved in a list, which can be retrieved with process_info(Pid, last_calls). A global function call is one in which the module of the function is explicitly mentioned. Only a fixed amount of information is saved: a tuple {Module, Function, Arity} for function calls, and the mere atoms send, 'receive' and timeout for sends and receives ('receive' when a message is received and timeout when a receive times out). If N = 0, call saving is disabled for the process, which is the default. Whenever the size of the call saving list is set, its contents are reset.

不过要慎重使用优先级，除非必要。

1.62 最大文件句柄数对内存的消耗

发表时间: 2008-12-07 关键字: max_files ulimit

sys_max_files在erlang beam里面2处地方消耗内存：

1.

```
static struct driver_data {
    int port_num, ofd, packet_bytes;
    ErtsSysReportExit *report_exit;
    int pid;
    int alive;
    int status;
} *driver_data; /* indexed by fd */
```

7 WORDS

```
driver_data = (struct driver_data *)erts_alloc(ERTS_ALC_T_DRV_TAB, max_files * sizeof(struct
driver_data));
```

2.

```
static struct fd_data {
    char  pbuf[4];   /* hold partial packet bytes */
    int   psz;       /* size of pbuf */
    char  *buf;
    char  *cpos;
    int   sz;
    int   remain;  /* for input on fd */
} *fd_data; /* indexed by fd */
```

5 WORDS + 4

```
fd_data = (struct fd_data *)
erts_alloc(ERTS_ALC_T_FD_TAB, max_files * sizeof(struct fd_data));
```

1个file消耗的内存为 13WORDS 左右， 32位机器大概是64个字节，假如你不小心把 ulimit -n 10000000 也就是说100w个句柄， 那光这个就消耗64M 内存.所以没有必须的时候， 最大文件句柄数还是够用就行。

1.63 erlang运行期的自省机制

发表时间: 2008-12-08 关键字: system_info system_profile system_monitor erts_debug trace

erlang运行期最值得称道的地方之一就是完备的自省机制，也就是说你可以通过这些信息了解整个系统运行的方方面面，给系统的调试，排错，调优，运营提供非常大的便利。在beam的c实现代码中至少1/4的代码在为这个目标服务，信息非常的到位详细，这和爱立信作为商业公司运营交换机的需求有很大的关系。自省分为2个层面的：提供erts运行期信息的和用户进程相关的信息。包括一下一个基础设施:system_flag, system_info,system_profile,system_monitor,erts_debug，the Erlang crash dumps,trace. 以及在otp的os_mon,snmp.

system_flag主要用来微调erts的性能相关参数。

erlang:system_flag(fullsweep_after, Number)

Number is a non-negative integer which indicates how many times generational garbages collections can be done without forcing a fullsweep collection. The value applies to new processes; processes already running are not affected.

In low-memory systems (especially without virtual memory), setting the value to 0 can help to conserve memory.

An alternative way to set this value is through the (operating system) environment variable ERL_FULLSWEEP_AFTER.

erlang:system_flag(min_heap_size, MinHeapSize)

Sets the default minimum heap size for processes. The size is given in words. The new min_heap_size only effects processes spawned after the change of min_heap_size has been made. The min_heap_size can be set for individual processes by use of spawn_opt/N or process_flag/2.

erlang:system_flag(multi_scheduling, BlockState)

BlockState = block | unblock

If multi-scheduling is enabled, more than one scheduler thread is used by the emulator. Multi-scheduling can be blocked. When multi-scheduling has been blocked, only one scheduler thread will schedule Erlang processes.

If BlockState =:= block, multi-scheduling will be blocked. If BlockState =:= unblock and no-one else is blocking multi-scheduling and this process has only blocked one time, multi-scheduling will be unblocked. One process can block multi-scheduling multiple times. If a process has blocked multiple times, it has to unblock exactly as many times as it has blocked before it has released its multi-scheduling block. If a process that has blocked multi-scheduling exits, it will release its blocking of multi-scheduling.

The return values are disabled, blocked, or enabled. The returned value describes the state just after the call to erlang:system_flag(multi_scheduling, BlockState) has been made. The return values are described in the documentation of erlang:system_info(multi_scheduling).

NOTE: Blocking of multi-scheduling should normally not be needed. If you feel that you need to block multi-scheduling, think through the problem at least a couple of times again. Blocking multi-scheduling should only be used as a last resort since it will most likely be a very inefficient way to solve the problem.

See also erlang:system_info(multi_scheduling), erlang:system_info(multi_scheduling_blockers), and erlang:system_info(schedulers).

erlang:system_flag(trace_control_word, TCW)

Sets the value of the node's trace control word to TCW. TCW should be an unsigned integer. For more information see documentation of the set_tcw function in the match specification documentation in the ERTS User's Guide.

system_info提供erts最基本的环境信息

erlang:system_info(Type) -> Res

Types:

Type, Res -- see below

Returns various information about the current system (emulator) as specified by Type:

allocated_areas

Returns a list of tuples with information about miscellaneous allocated memory areas.

Each tuple contains an atom describing type of memory as first element and amount of allocated memory in bytes as second element. In those cases when there is information present about allocated and used memory, a third element is present. This third element contains the amount of used memory in bytes.

erlang:system_info(allocated_areas) is intended for debugging, and the content is highly implementation dependent. The content of the results will therefore change when needed without prior notice.

Note: The sum of these values is not the total amount of memory allocated by the emulator. Some values are part of other values, and some memory areas are not part of the result. If you are interested in the total amount of memory allocated by the emulator see erlang:memory/0,1.

allocator

Returns {Allocator, Version, Features, Settings}.

Types:

* Allocator = undefined | elib_malloc | glibc
* Version = [int()]
* Features = [atom()]
* Settings = [{Subsystem, [{Parameter, Value}]}]
* Subsystem = atom()
* Parameter = atom()
* Value = term()

Explanation:

* Allocator corresponds to the malloc() implementation used. If Allocator equals undefined, the malloc() implementation used could not be identified. Currently elib_malloc and glibc can be identified.
* Version is a list of integers (but not a string) representing the version of the malloc() implementation used.
* Features is a list of atoms representing allocation features used.
* Settings is a list of subsystems, their configurable parameters, and used values. Settings may differ between different combinations of platforms, allocators, and allocation features. Memory sizes are given in bytes.

See also "System Flags Effecting erts_alloc" in erts_alloc(3).

alloc_util_allocators

Returns a list of the names of all allocators using the ERTS internal alloc_util framework as atoms. For more information see the "the alloc_util framework" section in the erts_alloc(3) documentation.

{allocator, Alloc}

Returns information about the specified allocator. As of erts version 5.6.1 the return value is a list of {instance, InstanceNo, InstanceInfo} tuples where InstanceInfo contains information about a specific instance of the allocator. If Alloc is not a recognized allocator, undefined is returned. If Alloc is disabled, false is returned.

Note: The information returned is highly implementation dependent and may be changed, or removed at any time without prior notice. It was initially intended as a tool when developing new allocators, but since it might be of interest for others it has been briefly documented.

The recognized allocators are listed in erts_alloc(3). After reading the erts_alloc(3) documentation, the returned information should more or less speak for itself. But it can be worth explaining some things. Call counts are presented by two values. The first value is giga calls, and the second value is calls. mbcs, and sbcs are abbreviations for, respectively, multi-block carriers, and single-block carriers. Sizes are presented in bytes. When it is not a size that is presented, it is the amount of something. Sizes and amounts are often presented by three values, the first is current value, the second is

maximum value since the last call to erlang:system_info({allocator, Alloc}), and the third is maximum value since the emulator was started. If only one value is present, it is the current value. fix_alloc memory block types are presented by two values. The first value is memory pool size and the second value used memory size.

{allocator_sizes, Alloc}

Returns various size information for the specified allocator. The information returned is a subset of the information returned by erlang:system_info({allocator, Alloc}).

c_compiler_used

Returns a two-tuple describing the C compiler used when compiling the runtime system. The first element is an atom describing the name of the compiler, or undefined if unknown. The second element is a term describing the version of the compiler, or undefined if unknown.

check_io

Returns a list containing miscellaneous information regarding the emulators internal I/O checking. Note, the content of the returned list may vary between platforms and over time. The only thing guaranteed is that a list is returned.

compat_rel

Returns the compatibility mode of the local node as an integer. The integer returned represents the Erlang/OTP release which the current emulator has been set to be backward compatible with. The compatibility mode can be configured at startup by using the command line flag +R, see erl(1).

creation

Returns the creation of the local node as an integer. The creation is changed when a node is restarted. The creation of a node is stored in process identifiers, port identifiers, and references. This makes it (to some extent) possible to distinguish between identifiers from different incarnations of a node. Currently valid creations are integers in the range 1..3, but this may (probably will) change in the future. If the node is not alive, 0 is returned.

debug_compiled

Returns true if the emulator has been debug compiled; otherwise, false.

dist

Returns a binary containing a string of distribution information formatted as in Erlang crash dumps. For more information see the "How to interpret the Erlang crash dumps" chapter in the ERTS User's Guide.

dist_ctrl

Returns a list of tuples {Node, ControllingEntity}, one entry for each connected remote node. The Node is the name of the node and the ControllingEntity is the port or pid responsible for the communication to that node. More specifically, the ControllingEntity for nodes connected via TCP/IP (the normal case) is the socket actually used in communication with the specific node.

driver_version

Returns a string containing the erlang driver version used by the runtime system. It will be on the

form "<major ver>.<minor ver>".

elib_malloc

If the emulator uses the elib_malloc memory allocator, a list of two-element tuples containing status information is returned; otherwise, false is returned. The list currently contains the following two-element tuples (all sizes are presented in bytes):

{heap_size, Size}

Where Size is the current heap size.

{max_alloced_size, Size}

Where Size is the maximum amount of memory allocated on the heap since the emulator started.

{alloced_size, Size}

Where Size is the current amount of memory allocated on the heap.

{free_size, Size}

Where Size is the current amount of free memory on the heap.

{no_alloced_blocks, No}

Where No is the current number of allocated blocks on the heap.

{no_free_blocks, No}

Where No is the current number of free blocks on the heap.

{smallest_alloced_block, Size}

Where Size is the size of the smallest allocated block on the heap.

{largest_free_block, Size}

Where Size is the size of the largest free block on the heap.

fullsweep_after

Returns {fullsweep_after, int()} which is the fullsweep_after garbage collection setting used by default. For more information see garbage_collection described below.

garbage_collection

Returns a list describing the default garbage collection settings. A process spawned on the local node by a spawn or spawn_link will use these garbage collection settings. The default settings can be changed by use of system_flag/2. spawn_opt/4 can spawn a process that does not use the default settings.

global_heaps_size

Returns the current size of the shared (global) heap.

heap_sizes

Returns a list of integers representing valid heap sizes in words. All Erlang heaps are sized from sizes in this list.

heap_type

Returns the heap type used by the current emulator. Currently the following heap types exist:

private

Each process has a heap reserved for its use and no references between heaps of different processes are allowed. Messages passed between processes are copied between heaps.

shared

One heap for use by all processes. Messages passed between processes are passed by reference.

hybrid

A hybrid of the private and shared heap types. A shared heap as well as private heaps are used.

info

Returns a binary containing a string of miscellaneous system information formatted as in Erlang crash dumps. For more information see the "How to interpret the Erlang crash dumps" chapter in the ERTS User's Guide.

kernel_poll

Returns true if the emulator uses some kind of kernel-poll implementation; otherwise, false.

loaded

Returns a binary containing a string of loaded module information formatted as in Erlang crash dumps. For more information see the "How to interpret the Erlang crash dumps" chapter in the ERTS User's Guide.

logical_processors

Returns the number of logical processors detected on the system as an integer or the atom unknown if the emulator wasn't able to detect any.

machine

Returns a string containing the Erlang machine name.

modified_timing_level

Returns the modified timing level (an integer) if modified timing has been enabled; otherwise, undefined. See the +T command line flag in the documentation of the erl(1) command for more information on modified timing.

multi_scheduling

Returns disabled, blocked, or enabled. A description of the return values:

disabled

The emulator has only one scheduler thread. The emulator does not have SMP support, or have been started with only one scheduler thread.

blocked

The emulator has more than one scheduler thread, but all scheduler threads but one have been blocked, i.e., only one scheduler thread will schedule Erlang processes and execute Erlang code.

enabled

The emulator has more than one scheduler thread, and no scheduler threads have been blocked, i.e., all available scheduler threads will schedule Erlang processes and execute Erlang code.

See also erlang:system_flag(multi_scheduling, BlockState), erlang:system_info(multi_scheduling_blockers), and erlang:system_info(schedulers).

multi_scheduling_blockers

Returns a list of PIDs when multi-scheduling is blocked; otherwise, the empty list. The PIDs in the list is PIDs of the processes currently blocking multi-scheduling. A PID will only be present once in the list, even if the corresponding process has blocked multiple times.

See also erlang:system_flag(multi_scheduling, BlockState), erlang:system_info(multi_scheduling), and erlang:system_info(schedulers).

otp_release

Returns a string containing the OTP release number.

process_count

Returns the number of processes currently existing at the local node as an integer. The same value as length(processes()) returns.

process_limit

Returns the maximum number of concurrently existing processes at the local node as an integer. This limit can be configured at startup by using the command line flag +P, see erl(1).

procs

Returns a binary containing a string of process and port information formatted as in Erlang crash dumps. For more information see the "How to interpret the Erlang crash dumps" chapter in the ERTS User's Guide.

scheduler_id

Returns the scheduler id (SchedulerId) of the scheduler thread that the calling process is executing on. SchedulerId is a positive integer; where 1 <= SchedulerId <= erlang:system_info(schedulers). See also erlang:system_info(schedulers).

schedulers

Returns the number of scheduler threads used by the emulator. A scheduler thread schedules Erlang processes and Erlang ports, and execute Erlang code and Erlang linked in driver code.

The number of scheduler threads is determined at emulator boot time and cannot be changed after that.

See also erlang:system_info(scheduler_id), erlang:system_flag(multi_scheduling, BlockState), erlang:system_info(multi_scheduling), and and erlang:system_info(multi_scheduling_blockers).

smp_support

Returns true if the emulator has been compiled with smp support; otherwise, false.

system_version

Returns a string containing the emulator type and version as well as some important properties such as the size of the thread pool, etc.

system_architecture

Returns a string containing the processor and OS architecture the emulator is built for.

threads

Returns true if the emulator has been compiled with thread support; otherwise, false is returned.

thread_pool_size

Returns the number of async threads in the async thread pool used for asynchronous driver calls (driver_async()) as an integer.

trace_control_word

Returns the value of the node's trace control word. For more information see documentation of the function get_tcw in "Match Specifications in Erlang", ERTS User's Guide.

version

Returns a string containing the version number of the emulator.

wordsize

Returns the word size in bytes as an integer, i.e. on a 32-bit architecture 4 is returned, and on a 64-bit architecture 8 is returned.

system_monitor主要用来调优.

erlang:system_monitor(MonitorPid, [Option]) -> MonSettings

Types:

MonitorPid = pid()
Option = {long_gc, Time} | {large_heap, Size} | busy_port | busy_dist_port
Time = Size = int()
MonSettings = {OldMonitorPid, [Option]}
OldMonitorPid = pid()

Sets system performance monitoring options. MonitorPid is a local pid that will receive system monitor messages, and the second argument is a list of monitoring options:

{long_gc, Time}

If a garbage collection in the system takes at least Time wallclock milliseconds, a message {monitor, GcPid, long_gc, Info} is sent to MonitorPid. GcPid is the pid that was garbage collected and Info is a list of two-element tuples describing the result of the garbage collection. One of the tuples is {timeout, GcTime} where GcTime is the actual time for the garbage collection in milliseconds. The other tuples are tagged with heap_size, heap_block_size, stack_size, mbuf_size, old_heap_size, and

old_heap_block_size. These tuples are explained in the documentation of the gc_start trace message (see erlang:trace/3). New tuples may be added, and the order of the tuples in the Info list may be changed at any time without prior notice.

{large_heap, Size}

If a garbage collection in the system results in the allocated size of a heap being at least Size words, a message {monitor, GcPid, large_heap, Info} is sent to MonitorPid. GcPid and Info are the same as for long_gc above, except that the tuple tagged with timeout is not present. Note: As of erts version 5.6 the monitor message is sent if the sum of the sizes of all memory blocks allocated for all heap generations is equal to or larger than Size. Previously the monitor message was sent if the memory block allocated for the youngest generation was equal to or larger than Size.

busy_port

If a process in the system gets suspended because it sends to a busy port, a message {monitor, SusPid, busy_port, Port} is sent to MonitorPid. SusPid is the pid that got suspended when sending to Port.

busy_dist_port

If a process in the system gets suspended because it sends to a process on a remote node whose inter-node communication was handled by a busy port, a message {monitor, SusPid, busy_dist_port, Port} is sent to MonitorPid. SusPid is the pid that got suspended when sending through the inter-node communication port Port.

system_profile也是提供些erts进程和port的信息

erlang:system_profile(ProfilerPid, Options) -> ProfilerSettings

Types:

ProfilerSettings -> {ProfilerPid, Options} | undefined
ProfilerPid = pid() | port()
Options = [Option]
Option = runnable_procs | runnable_ports | scheduler | exclusive

Sets system profiler options. ProfilerPid is a local pid or port that will receive profiling messages. The receiver is excluded from all profiling. The second argument is a list of profiling options:

runnable_procs

If a process is put into or removed from the runqueue a message, {profile, Pid, State, Mfa, Ts}, is sent to ProfilerPid. Running processes that is reinsertet into the runqueue after completing its reductions does not trigger this message.

runnable_ports

If a port is put into or removed from the runqueue a message, {profile, Port, State, 0, Ts}, is sent to ProfilerPid.

scheduler

If a scheduler is put to sleep or awoken a message, {profile, scheduler, Id, State, NoScheds, Ts}, is sent to ProfilerPid.

exclusive

If a synchronous call to a port from a process is done, the calling process is considered not runnable during the call runtime to the port. The calling process is notified as inactive and subsequently active when the port callback return

erts_debug提供最详尽的内部数据信息
erts_debug:get_internal_state()未公开

node_and_dist_references
DbTable_words
next_pid
next_port
check_io_debug
available_internal_state
monitoring_nodes
link_list
monitor_list
channel_number
have_pending_exit

crash dump则是个无价之宝 可以用erlang:halt来强行产生系统的运行映像，分析系统的运行状态。

其他的几个机制如trace文档都写得很清楚,我就不抄书了。

1.64 Hunting Bugs

发表时间: 2008-12-08 关键字: system_monitor

转自：http://www.erlangatwork.com/2008/07/hunting-bugs.html

Our Erlang gateways were developed and deployed in phases starting with AIM/ICQ, GTalk, Yahoo, and finally MSN. Aside from minor protocol implementation bugs there were no problems and we were very satisfied with stability and performance. However, not long after releasing the MSN gateway we noticed that it used a ton of memory, and also periodically suffered massive spikes in memory use which invoked the Linux kernel's OOM killer. This lead to a crash course in debugging running Erlang apps, and a great appreciation for the years of real-world lessons that have influenced the features and design of Erlang/OTP.

The first thing I looked into is why the gateway would eventually use several gigabytes of memory with only a few hundred users online. Since each Erlang process has its own heap, I started by looking for which processes were using the most memory. erlang:processes/0 returns a list of all running processes, and erlang:process_info/1 provides a ton of information about a process including heap use, stack size, etc. So I wrote a quick script to dump the process info of all processes to a file, sorted by total memory use. This was run on the live gateway instance.

It turned out that only a few active MSN sessions were using the majority of the heap, and these sessions were for users with very large contact lists. After initial login, one session could be using > 1GB of heap.

Newer versions of the MSNP protocol use SOAP requests to get authorization tokens, contact lists, allow/block lists, etc. My initial implementation was very simple, using inets to submit the HTTP request, reading the full response body as a list, and then parsing that list with xmerl. These responses could be very large and since the gateway was running on a 64bit Erlang VM, each character would occupy 16 bytes of memory. xmerl's representation of an XML document also requires quite a bit of storage. A simple XML document such as:

<a>foo<c/>

is represented as:

```
{xmlElement,a,a,[],
        {xmlNamespace,[],[]},
        [],1,[],
        [{xmlElement,b,b,[],
                {xmlNamespace,[],[]},
                [{a,1}],
                1,[],
                [{xmlText,[{b,1},{a,1}],1,[],"foo",text}],
                [],"/tmp/",undeclared},
         {xmlElement,c,c,[],
                {xmlNamespace,[],[]},
                [{a,1}],
                2,[],[],[],undefined,undeclared}],
        [],"/tmp/",undeclared}
```

So I rewrote my SOAP module to use the streaming method http:request/4 which returns the HTTP response as a series of binary chunks. xmerl doesn't support parsing binaries so I switched to erlsom, which does, and also converted the XML to a very simple and compact format:

```
{a,[],
[{b,[],[<<"foo">>]},
  {c,[],[]}]}
```

After making these changes the amount of memory used per login decreased by 2.5-3x. However the gateway was still occasionally using up all available memory and dying at what appeared to be random intervals. My best guess was that something in the protocol stream was triggering this problem so I updated the gateway to log each login attempt, and ran tcpdump to capture all MSN traffic. Eventually I was able to correlate the crashes with incoming status text messages from certain contacts of a few heysan users.

MSNP transports status text as an XML payload of the UBX command:

<Data><CurrentMedia></CurrentMedia><PSM>status text</PSM></Data>

I was still using xmerl to parse this small XML document and grab the cdata from the <PSM> tag. The status text of some contacts contained combinations of UTF-8 text and numeric unicode entities such as :. Simply attempting to parse these small XML documents would cause xmerl to allocate more than 8GB of memory and thus kill the emulator. Parsing the UBX payload with erlsom instead of xmerl completely resolved the problem, but was a bit of a letdown after so much time spent hunting hunting such an esoteric bug.

UPDATE: the crash described above is fixed in xmerl-1.1.10, which is included in Erlang/OTP R12B-4.

要善于erlang的基础设施 事半功倍！

1.65 inside-beam-erlang-virtual-machine

发表时间: 2008-12-08 关键字: erlang beam

转自： http://erlangdotnet.net/2007/09/inside-beam-erlang-virtual-machine.html

OTP R11B-5 includes more than one million lines of Erlang. The kernel and standard library are about a third of these lines. The emulator, BEAM, is around 200,000 lines of C.

BEAM's C main functions are platform-specific and live in erts/emulator/sys/unix, win32, and so on. (Windows actually has two mains: a simple one similar to the Unix entrypoints in erl_main.c, and a more complex one in erl_main_sae.c which is used in bootstrapping the toolchain.) The conventional mains immediately call erl_start which processes command line arguments, copies argv into a list of strings, creates a process and runs otp_ring0 with the argument list. otp_ring0 looks up and runs init:boot/1. There is more initialization after this, but from otp_ring0 on, apart from some built-in functions, Erlang is implemented in Erlang.

In practice, the Erlang compiler (another 30,000 lines of Erlang) compiles Erlang to BEAM bytecodes. When erl_start runs otp_ring0 and so on, it is loading BEAM binaries generated by the Erlang compiler earlier. The BEAM instruction set is register-based and has 130 instructions. These include instructions for arithmetic, comparsions and boolean logic; manipulating strings, tuples and lists; stack and heap allocation and freeing; type tests for Erlang primitives (numbers, lists, process IDs, references, and so on); jumps, structured exception handling, and calls and returns; sending messages and reading from the process' mailbox; waiting and timeouts; and so on. The instructions are listed in lib/compiler/src/beam_opcodes.erl. In BEAM, erts/emulator/beam/ops.tab contains rewrite rules that map typed BEAM instructions into emulator internals. A Perl script (beam_makeops) translates this file into C code (mostly macros.)

Some Erlang built-in function (BIFs) calls are translated into an extended instruction set not generated by the compiler. For example, the BIF erlang:self/3 is rewritten as a single 'self' instruction. Other BIF linkage is via a set of instructions for calling BIFs of varying arities. erts/emulator/beam/bif.tab maps BIF labels into C externs. For example, calls to the BIF erlang:spawn/3 boil down to a call to spawn_3 (found in erts/emulator/bif.c), which in turn thunks to erl_create_process.

Incidentally, processes are not related to threads or operating system processes. Instead, BEAM maintains queues of processes that are scheduled to run. There is one queue per priority level. Schedulers pick processes from the queues and run them. Processes maintain their registers and a pointer to the next instruction to execute.

The garbage collector, in erts/emulator/beam/erl_gc.c, is a mark-and-sweep copying collector. The most interesting thing about the BEAM garbage collector is that the garbage collected heaps are per-process. BEAM can do this because Erlang processes communicate explicitly via messages, and not via mutating shared memory. The CLR, in contrast, divides heaps into generations protected by write barriers and does incremental collection to avoid having to garbage collect all of a program's heap at the same time. This extra complexity is part of the price of using a shared heap to communicate between threads in the CLR.

虽然版本比较老 现在是R12B-5不过关键东西还是差不多没变。

1.66 erlang的process等同于lua的coroutine？

发表时间: 2008-12-08 关键字: process lua coroutine

erlang的process是个调度单位 它包含特定的MFA， 消息队列等，调度后由beam_emu来执行其中的opcode 在等待消息或者yield的时候放弃执行权,有消息的时候接着原来的地方继续执行。lua的coroutine也是同样的，只不过他没有消息队列， 它的再执行靠lua_resume来推动。

我的理解是这样的，这是2个语言不同的地方，但是同样的轻量，同样的效率。

1.67 erlang对port子进程退出的处理

发表时间: 2008-12-08 关键字: smp_sig_notify

erlang通过port来spawn外部程序 重定向外部程序的stdin, stdout到一对pipe行通信的,利用poll来检测外部程序的读写事件。但是如果外部程序退出的话，erts如何知道并且加以处理的呢？

erts运行的时候会初始化smp_sig_notify,开启一个信号处理线程，在这个线程里面做具体的信号处理。

```
static void
init_smp_sig_notify(void)
{
    erts_smp_thr_opts_t thr_opts = ERTS_SMP_THR_OPTS_DEFAULT_INITER;
    thr_opts.detached = 1;

    if (pipe(sig_notify_fds) < 0) {
erl_exit(ERTS_ABORT_EXIT,
"Failed to create signal-dispatcher pipe: %s (%d)\n",
erl_errno_id(errno),
errno);
    }

    /* Start signal handler thread */
    erts_smp_thr_create(&sig_dispatcher_tid,
signal_dispatcher_thread_func,
NULL,
&thr_opts);
}

static void *
signal_dispatcher_thread_func(void *unused)
{
    int initialized = 0;
#if !CHLDWTHR
    int notify_check_children = 0;
#endif
#ifdef ERTS_ENABLE_LOCK_CHECK
    erts_lc_set_thread_name("signal_dispatcher");
```

```
#endif
    erts_thread_init_fp_exception();
    while (1) {
char buf[32];
int res, i;
/* Block on read() waiting for a signal notification to arrive... */
res = read(sig_notify_fds[0], (void *) &buf[0], 32);
if (res < 0) {
    if (errno == EINTR)
continue;
    erl_exit(ERTS_ABORT_EXIT,
     "signal-dispatcher thread got unexpected error: %s (%d)\n",
     erl_errno_id(errno),
     errno);
}
for (i = 0; i < res; i++) {
    /*
     * NOTE 1: The signal dispatcher thread should not do work
     *         that takes a substantial amount of time (except
     *         perhaps in test and debug builds). It needs to
     *         be responsive, i.e, it should only dispatch work
     *         to other threads.
     *
     * NOTE 2: The signal dispatcher thread is not a blockable
     *         thread (i.e., it hasn't called
     *         erts_register_blockable_thread()). This is
     *         intentional. We want to be able to interrupt
     *         writing of a crash dump by hitting C-c twice.
     *         Since it isn't a blockable thread it is important
     *         that it doesn't change the state of any data that
     *         a blocking thread expects to have exclusive access
     *         to (unless the signal dispatcher itself explicitly
     *         is blocking all blockable threads).
     */
    switch (buf[i]) {
    case 0: /* Emulator initialized */
initialized = 1;
#if !CHLDWTHR
```

```
if (!notify_check_children)
#endif
    break;
#if !CHLDWTHR
    case 'C': /* SIGCHLD */
if (initialized)
    erts_smp_notify_check_children_needed();
else
    notify_check_children = 1;
break;
#endif
    case 'I': /* SIGINT */
break_requested();
break;
    case 'Q': /* SIGQUIT */
quit_requested();
break;
    case '1': /* SIGUSR1 */
sigusr1_exit();
break;
#ifdef QUANTIFY
    case '2': /* SIGUSR2 */
quantify_save_data(); /* Might take a substantial amount of
time, but this is a test/debug
build */
break;
#endif
    default:
erl_exit(ERTS_ABORT_EXIT,
"signal-dispatcher thread received unknown "
"signal notification: '%c'\n",
buf[i]);
    }
}
ERTS_SMP_LC_ASSERT(!ERTS_LC_IS_BLOCKING);
    }
    return NULL;
}
```

```
void
erts_sys_main_thread(void)
{
   /* Become signal receiver thread... */
#ifdef ERTS_ENABLE_LOCK_CHECK
   erts_lc_set_thread_name("signal_receiver");
#endif

   smp_sig_notify(0); /* Notify initialized */
   while (1) {
/* Wait for a signal to arrive... */
#ifdef DEBUG
int res =
#else
(void)
#endif
   select(0, NULL, NULL, NULL, NULL);
ASSERT(res < 0);
ASSERT(errno == EINTR);
   }
}
```

因为外部的程序是fork exec来执行的，所以退出的时候erts进程就会受到SIGCHLD信号。

```
static int spawn_init()
{
...
sys_sigset(SIGCHLD, onchld); /* Reap children */
...
}
```

onchld就会被调用

```
static RETSIGTYPE onchld(int signum)
{
#if CHLDWTHR
   ASSERT(0); /* We should *never* catch a SIGCHLD signal */
#elif defined(ERTS_SMP)
```

```
    smp_sig_notify('C');
#else
    children_died = 1;
    ERTS_CHK_IO_INTR(1); /* Make sure we don't sleep in poll */
#endif
}


static void
smp_sig_notify(char c)
{
    int res;
    do {
/* write() is async-signal safe (according to posix) */
res = write(sig_notify_fds[1], &c, 1);
    } while (res < 0 && errno == EINTR);
    if (res != 1) {
char msg[] =
    "smp_sig_notify(): Failed to notify signal-dispatcher thread "
    "about received signal";
(void) write(2, msg, sizeof(msg));
abort();
    }
}
```

于是erts_smp_notify_check_children_needed()被调用。

```
void
erts_smp_notify_check_children_needed(void)
{
    ErtsSchedulerData *esdp;
    erts_smp_sched_lock();
    for (esdp = schedulers; esdp; esdp = esdp->next)
esdp->check_children = 1;
    if (block_multi_scheduling) {
/* Also blocked schedulers need to check children */
erts_smp_mtx_lock(&msched_blk_mtx);
for (esdp = schedulers; esdp; esdp = esdp->next)
```

```
    esdp->blocked_check_children = 1;
erts_smp_cnd_broadcast(&msched_blk_cnd);
erts_smp_mtx_unlock(&msched_blk_mtx);
    }
    wake_all_schedulers();
    erts_smp_sched_unlock();
}
```

这个函数设置调度器的check_children的标志 并且唤醒所有的调度器。

调度器的入口process_main我们来看下如何处理的:

```
Process *schedule(Process *p, int calls)
{
...
if (esdp->check_children) {
    esdp->check_children = 0;
    erts_smp_sched_unlock();
    erts_check_children();
    erts_smp_sched_lock();
}
...
}
```

调用erts_check_children。

```
void
erts_check_children(void)
{
    (void) check_children();
}

static int check_children(void)
{
    int res = 0;
    int pid;
    int status;

#ifndef ERTS_SMP
    if (children_died)
```

```
#endif
    {
sys_sigblock(SIGCHLD);
CHLD_STAT_LOCK;
while ((pid = waitpid(-1, &status, WNOHANG)) > 0)
    note_child_death(pid, status);
#ifndef ERTS_SMP
children_died = 0;
#endif
CHLD_STAT_UNLOCK;
sys_sigrelease(SIGCHLD);
res = 1;
    }
    return res;
}

static void note_child_death(int pid, int status)
{
    ErtsSysReportExit **repp = &report_exit_list;
    ErtsSysReportExit *rep = report_exit_list;

    while (rep) {
if (pid == rep->pid) {
    *repp = rep->next;
    ERTS_REPORT_EXIT_STATUS(rep, status);
    break;
}
repp = &rep->next;
rep = rep->next;
    }
}

static ERTS_INLINE void
report_exit_status(ErtsSysReportExit *rep, int status)
{
    Port *pp;
#ifdef ERTS_SMP
    CHLD_STAT_UNLOCK;
```

```
#endif
    pp = erts_id2port_sflgs(rep->port,
    NULL,
    0,
    ERTS_PORT_SFLGS_INVALID_DRIVER_LOOKUP);
#ifdef ERTS_SMP
    CHLD_STAT_LOCK;
#endif
    if (pp) {
if (rep->ifd >= 0) {
    driver_data[rep->ifd].alive = 0;
    driver_data[rep->ifd].status = status;
    (void) driver_select((ErlDrvPort) internal_port_index(pp->id),
rep->ifd,
DO_READ,
1);
}
if (rep->ofd >= 0) {
   driver_data[rep->ofd].alive = 0;
    driver_data[rep->ofd].status = status;
    (void) driver_select((ErlDrvPort) internal_port_index(pp->id),
rep->ofd,
DO_WRITE,
1); }
erts_port_release(pp);
    }
    erts_free(ERTS_ALC_T_PRT_REP_EXIT, rep);
}
```

移除对该port的监视 销毁port.

```
static void ready_input(ErlDrvData e, ErlDrvEvent ready_fd)
{
...
     res = read(ready_fd, read_buf, ERTS_SYS_READ_BUF_SZ);
if (res < 0) {
    if ((errno != EINTR) && (errno != ERRNO_BLOCK))
```

```
port_inp_failure(port_num, ready_fd, res);
}
else if (res == 0)
    port_inp_failure(port_num, ready_fd, res);
else
    driver_output(port_num, (char*) read_buf, res);
erts_free(ERTS_ALC_T_SYS_READ_BUF, (void *) read_buf);
    }....
}

static int port_inp_failure(int port_num, int ready_fd, int res)
/* Result: 0 (eof) or -1 (error) */
{
    int err = errno;

    ASSERT(res <= 0);
    (void) driver_select(port_num, ready_fd, ERL_DRV_READ|ERL_DRV_WRITE, 0);
    clear_fd_data(ready_fd);
    if (res == 0) {
if (driver_data[ready_fd].report_exit) {
    CHLD_STAT_LOCK;

    if (driver_data[ready_fd].alive) {
/*
* We have eof and want to report exit status, but the process
* hasn't exited yet. When it does report_exit_status() will
* driver_select() this fd which will make sure that we get
* back here with driver_data[ready_fd].alive == 0 and
* driver_data[ready_fd].status set.
*/
CHLD_STAT_UNLOCK;
return 0;
    }
    else {
int status = driver_data[ready_fd].status;
CHLD_STAT_UNLOCK;

/* We need not be prepared for stopped/continued processes. */
```

```
if (WIFSIGNALED(status))
    status = 128 + WTERMSIG(status);
else
    status = WEXITSTATUS(status);

driver_report_exit(driver_data[ready_fd].port_num, status);     }
       }
       driver_failure_eof(port_num);
    } else {
driver_failure_posix(port_num, err);
    }
    return 0;
}



void driver_report_exit(int ix, int status)
{
   Port* prt = erts_drvport2port(ix);
   Eterm* hp;
   Eterm tuple;
   Process *rp;
   Eterm pid;
   ErlHeapFragment *bp = NULL;
   ErlOffHeap *ohp;
   ErtsProcLocks rp_locks = 0;

   ERTS_SMP_CHK_NO_PROC_LOCKS;
   ERTS_SMP_LC_ASSERT(erts_lc_is_port_locked(prt));

   pid = prt->connected;
   ASSERT(is_internal_pid(pid));
   rp = erts_pid2proc_opt(NULL, 0, pid, 0, ERTS_P2P_FLG_SMP_INC_REFC);
   if (!rp)
       return;

   hp = erts_alloc_message_heap(3+3, &bp, &ohp, rp, &rp_locks);

   tuple = TUPLE2(hp, am_exit_status, make_small(status));
```

```
  hp += 3;
  tuple = TUPLE2(hp, prt->id, tuple);   erts_queue_message(rp, &rp_locks, bp, tuple, am_undefined);

  erts_smp_proc_unlock(rp, rp_locks);
  erts_smp_proc_dec_refc(rp);
}
```

于是我们收到{Port, {exit_staus, Staus}}事件。

有点复杂吧，不过挺优雅的。记住信号处理函数里面不能做太耗时和调用有害的api。还有会有大量的退出事件发生，让调度器来调度这个事情比较公平，避免系统在处理退出处理上投入！

1.68 erlang的进程调度器工作流程

发表时间: 2008-12-08 关键字: schedule process_main

在多处理器机器上erlang默认是有几个cpu就有几个调度器线程，除非你通过+S N 参数来改变。每个调度器线程的入口函数是process_main, 外加一个主线程阻塞在select上等待中断事件的发生. process_main会调用schedule选择一个合适的process来执行。每个process里面都包含了要执行的的MFA,执行funtcion的opcode。beam_emu的opcode是基于register的，大概有180条左右opcode,每个版本都在增加，特别是最近的版本为了增加binary出来的效率加多了很多.

```
/*
* schedule() is called from BEAM (process_main()) or HiPE
* (hipe_mode_switch()) when the current process is to be
* replaced by a new process. 'calls' is the number of reduction
* steps the current process consumed.
* schedule() returns the new process, and the new process'
* ->fcalls field is initialised with its allowable number of
* reduction steps.
*
* When no process is runnable, or when sufficiently many reduction
* steps have been made, schedule() calls erl_sys_schedule() to
* schedule system-level activities.
*
* We use the same queue for normal and low prio processes.
* We reschedule low prio processes a certain number of times
* so that normal processes get to run more frequently.
*/
Process *schedule(Process *p, int calls)；
```

erlang系统调度的核心就是这个函数。

schedule调度的顺序是这样的：

1. 首先处理timer超时。
2. 处理子进程退出的情况。
3. 处理port_task事件，也就是port的IO事件
4. 如果没有活跃的进程 就sys_schdule阻塞在底层的IO中。
5. 根据process的优先级选出一个进程来调度。

这5个调度schdule要去平衡调度的力度，以期公平。

```
void
erl_sys_schedule(int runnable)
{
#ifdef ERTS_SMP
    ERTS_CHK_IO(!runnable);
    ERTS_SMP_LC_ASSERT(!ERTS_LC_IS_BLOCKING);
#endif
}
```

check_io就阻塞在poll上等待IO时间的发生，也就是说如果系统不繁忙的时候所以的scheduler都在等待IO事件的发生。

1.69 erlang的atom实现

发表时间: 2008-12-09 关键字: atom index dec enc

atom在erlang里面的作用非常大，特别是在消息匹配的时候，所以需要个非常高效的实现。erts的index.c hash.c atom.c用来实现atom. atom在内部表示的时候是个index. atom是个字符串，首先存在hash表里，然后把在hash里的slot,放在index的索引表里。这样要获取atom就非常高效，基本是2次指针运算。当需要把atom传送到外部的时候，第一次的时候就乖乖的送字符串的内容，同时把这个atom cache起来，以后再发送的时候就可以只传送在cache中的index了，大大提高了网络传输效率。具体的常见dec_atom enc_atom, external.c中。

细节中可以看到erlang非常注意性能。

1.70 让erlang支持Huge TLB

发表时间: 2008-12-10 关键字: hugetlb mseg

压榨linux2.6最后点内存分配性能,使用hugetlb.

Huge TLB Filesystem
==========

Most modern architectures support more than one page size. For example,
the IA-32 architecture supports 4KiB pages or 4MiB pages but Linux only
used large pages for mapping the actual kernel image. As TLB slots are a
scarce resource, it is desirable to be able to take advantages of the
large pages especially on machines with large amounts of physical memory.

In 2.6, Linux allows processes to use large pages, referred to as huge
pages. The number of available huge pages is configured by the system
administrator via the /proc/sys/vm/nr_hugepages proc interface. As the
success of the allocation depends on the availability of physically
contiguous memory, the allocation should be made during system startup.

The root of the implementation is a Huge TLB Filesystem (hugetlbfs) which
is a pseudo-filesystem implemented in fs/hugetlbfs/inode.c and based on
ramfs. The basic idea is that any file that exists in the filesystem is
backed by huge pages. This filesystem is initialised and registered as an
internal filesystem at system start-up.

There is two ways that huge pages may be accessed by a process. The first
is by using shmget() to setup a shared region backed by huge pages and the
second is the call mmap() on a file opened in the huge page filesystem.

When a shared memory region should be backed by huge pages, the process
should call shmget() and pass SHM_HUGETLB as one of the flags. This
results in a file being created in the root of the internal filesystem.
The name of the file is determined by an atomic counter called
hugetlbfs_counter which is incremented every time a shared region is
setup.

To create a file backed by huge pages, a filesystem of type hugetlbfs must first be mounted by the system administrator. Once the filesystem is mounted, files can be created as normal with the system call open(). When mmap() is called on the open file, the hugetlbfs registered mmap() function creates the appropriate VMA for the process.

Huge TLB pages have their own function for the management of page tables, address space operations and filesystem operations. The names of the functions for page table management can all be seen in $<$linux/hugetlb.h $>$ and they are named very similar to their ``normal'' page equivalents.

==========

把erl_mseg.c稍稍改造下是可以支持的！

```
void
erts_mseg_init(ErtsMsegInit_t *init)
{
...
#if HAVE_MMAP && !defined(MAP_ANON)
    mmap_fd = open("/dev/zero", O_RDWR);
    if (mmap_fd < 0)
erl_exit(ERTS_ABORT_EXIT, "erts_mseg: unable to open /dev/zero\n");
#endif
...
}
```

mmap_fd改成指向hugetlbfs的文件句柄就好了或者最好的方法就个getenv("MMAP_FILENAME")来获取。

1.71 叹！beam的hybrid还未支持

发表时间: 2008-12-11 关键字: beam hybrid

看下erl_bif_info.c

```
BIF_RETTYPE system_info_1(BIF_ALIST_1)
{
...
} else if (BIF_ARG_1 == am_heap_type) {
#if defined(HYBRID)
        return am_hybrid;
#else
return am_private;
#endif
...
}
```

也就是说目前只打算支持私有堆和混合堆。

```
[root@haserver otp_src_R12B-5]# erl
Erlang (BEAM) emulator version 5.6.5 [source] [smp:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.6.5  (abort with ^G)
1> erlang:system_info(heap_type).
private
```

私有堆模式。

```
[root@haserver ~]# erl -hybrid
erlexec: Error 2 executing '/usr/local/lib/erlang/erts-5.6.5/bin/beam.hybrid'.
```

根本没有生成beam.hybrid 可执行文件

于是重新编译系统

```
./configure --enable-hybrid-heap && make && make install
```

还是没有hybrid。

找了半天发现
Makefile.in

Until hybrid is nofrag, don't build it.
#BUILD_HYBRID_EMU=@ERTS_BUILD_HYBRID_EMU@
BUILD_HYBRID_EMU=no

强行改成yes,编译出错.查看源代码发现hybird INCREMENTAL模式的代码根本没写完。
唉,他们吹出去了，东西还没做好！

目前还不能享受先进科技 大佬们加油呀！！！

1.72 CN Erlounge III 候选讲师名单及议题

发表时间: 2008-12-11 关键字: cn erlounge

候选讲师名单及议题

http://www.ecug.org/lecturer/

要获得讲稿及 DEMO 代码，请到 SVN 库中 checkout 之。

* 对于 Linux 平台用户，请使用命令：svn co http://ecug.googlecode.com/svn/trunk/cn-erlounge/iii
* 对于 Windows 平台用户，请使用 SVN 客户端（如 TortoiseSVN）下载。

1.73 我的演讲主题：Inside the Erlang VM

发表时间: 2008-12-12 关键字: inside the erlang vm

我准备在上海举行的CN Erlounge III聚会中演讲的ppt, 欢迎拍砖头！

附件下载:

- Inside_Erlang_VM.rar (21.6 KB)
- dl.javaeye.com/topics/download/27534954-9498-33e9-bd33-363d6be53d51

1.74 erlang最小系统支持从远端加载beam

发表时间: 2008-12-12 关键字: loader prim_inet prim_file prim_zip

参见这个 http://avindev.javaeye.com/blog/100113

-hosts Hosts
Specifies the IP addresses for the hosts on which Erlang boot servers are running, see erl_boot_server(3). This flag is mandatory if the -loader inet flag is present.
The IP addresses must be given in the standard form (four decimal numbers separated by periods, for example "150.236.20.74". Hosts names are not acceptable, but a broadcast address (preferably limited to the local network) is.

-id Id
Specifies the identity of the Erlang runtime system. If it is run as a distributed node, Id must be identical to the name supplied together with the
-sname or -name flag.

-loader Loader
Specifies the method used by erl_prim_loader to load Erlang modules into the system. See erl_prim_loader(3). Two Loader methods are supported, efile and inet. efile means use the local file system, this is the default. inet means use a boot server on another machine, and the -id, -hosts and -setcookie flags must be specified as well. If Loader is something else, the user supplied Loader port program is started.

preload的模块：

```
struct {
  char* name;
  int size;
  unsigned char* code;
} pre_loaded[] = {
 {"otp_ring0", 512, preloaded_otp_ring0},
 {"init", 19096, preloaded_init},
 {"prim_inet", 25948, preloaded_prim_inet},
 {"prim_file", 12264, preloaded_prim_file},
 {"zlib", 4248, preloaded_zlib},
```

```
  {"prim_zip", 8228, preloaded_prim_zip},
  {"erl_prim_loader", 21616, preloaded_erl_prim_loader},
  {"erlang", 7776, preloaded_erlang},
  {0, 0, 0}
};
```

所以必须预加载这些基础模块以便进一步load所需的beam.

1.75 erlang进程退出时清扫的资源

发表时间: 2008-12-23 关键字: process port driver dist timer

ERTS要终止一个经常的时候调用一下函数

```
void
erts_do_exit_process(Process* p, Eterm reason)
{
   ErtsLink* lnk;
   ErtsMonitor *mon;
#ifdef ERTS_SMP
   erts_pix_lock_t *pix_lock = ERTS_PID2PIXLOCK(p->id);
#endif

   p->arity = 0; /* No live registers */
   p->fvalue = reason;

#ifdef ERTS_SMP
   ERTS_SMP_CHK_HAVE_ONLY_MAIN_PROC_LOCK(p);
   /* By locking all locks (main lock is already locked) when going
      to status P_EXITING, it is enough to take any lock when
      looking up a process (erts_pid2proc()) to prevent the looked up
      process from exiting until the lock has been released. */
   erts_smp_proc_lock(p, ERTS_PROC_LOCKS_ALL_MINOR);
#endif

   if (erts_system_profile_flags.runnable_procs && (p->status != P_WAITING)) {
   profile_runnable_proc(p, am_inactive);
   }

#ifdef ERTS_SMP
   erts_pix_lock(pix_lock);
   p->is_exiting = 1;
#endif

   p->status = P_EXITING;
```

```
#ifdef ERTS_SMP
    erts_pix_unlock(pix_lock);

    if (ERTS_PROC_PENDING_EXIT(p)) {
/* Process exited before pending exit was received... */
p->pending_exit.reason = THE_NON_VALUE;
if (p->pending_exit.bp) {
    free_message_buffer(p->pending_exit.bp);
    p->pending_exit.bp = NULL;
}
    }

    cancel_suspend_of_suspendee(p, ERTS_PROC_LOCKS_ALL);

    ERTS_SMP_MSGQ_MV_INQ2PRIVQ(p);
#endif

    if (IS_TRACED_FL(p,F_TRACE_PROCS))
trace_proc(p, p, am_exit, reason);

    erts_trace_check_exiting(p->id);

    ASSERT((p->trace_flags & F_INITIAL_TRACE_FLAGS) == F_INITIAL_TRACE_FLAGS);


    /* 清除进程用的定时器 */
    cancel_timer(p); /* Always cancel timer just in case */

    /* 清除bif定时器 */
    if (p->bif_timers)
erts_cancel_bif_timers(p, ERTS_PROC_LOCKS_ALL);

    /* 清除ETS 资源*/
    if (p->flags & F_USING_DB)
db_proc_dead(p->id);

#ifdef ERTS_SMP
```

```
    if (p->flags & F_HAVE_BLCKD_MSCHED)
erts_block_multi_scheduling(p, ERTS_PROC_LOCKS_ALL, 0, 1);
#endif

    /* 清除 DLL驱动资源 */
    if (p->flags & F_USING_DDLL) {
erts_ddll_proc_dead(p, ERTS_PROC_LOCKS_ALL);
    }

    /* 清除dist节点监控 */
    if (p->nodes_monitors)
erts_delete_nodes_monitors(p, ERTS_PROC_LOCKS_ALL);

    /*
     * The registered name *should* be the last "erlang resource" to
     * cleanup.
     */
    /* 清除名字登记 */
    if (p->reg)
(void) erts_unregister_name(p, ERTS_PROC_LOCKS_ALL, NULL, p->reg->name);



   {
int pix;

ASSERT(internal_pid_index(p->id) < erts_max_processes);
pix = internal_pid_index(p->id);

erts_smp_proc_tab_lock();
erts_smp_sched_lock();

#ifdef ERTS_SMP
erts_pix_lock(pix_lock);

ASSERT(p->scheduler_data);
ASSERT(p->scheduler_data->current_process == p);
ASSERT(p->scheduler_data->free_process == NULL);
```

```
p->scheduler_data->current_process = NULL;
p->scheduler_data->free_process = p;
p->status_flags = 0;
#endif
process_tab[pix] = NULL; /* Time of death! */
ASSERT(erts_smp_atomic_read(&process_count) > 0);
erts_smp_atomic_dec(&process_count);

#ifdef ERTS_SMP
erts_pix_unlock(pix_lock);
#endif
erts_smp_sched_unlock();

if (p_next < 0) {
    if (p_last >= p_next) {
p_serial++;
p_serial &= p_serial_mask;
    }
    p_next = pix;
}

erts_smp_proc_tab_unlock();
    }

    /*
     * All "erlang resources" have to be deallocated before this point,
     * e.g. registered name, so monitoring and linked processes can
     * be sure that all interesting resources have been deallocated
     * when the monitors and/or links hit.
     */

    mon = p->monitors;
    p->monitors = NULL; /* to avoid recursive deletion during traversal */

    lnk = p->nlinks;
    p->nlinks = NULL;
    p->status = P_FREE;
```

```
    erts_smp_proc_unlock(p, ERTS_PROC_LOCKS_ALL);
    processes_busy--;

    if ((p->flags & F_DISTRIBUTION) && p->dist_entry)
erts_do_net_exits(p->dist_entry, reason);

    /*
     * Pre-build the EXIT tuple if there are any links.
     */
    if (lnk) {
Eterm tmp_heap[4];
Eterm exit_tuple;
Uint exit_tuple_sz;
Eterm* hp;

hp = &tmp_heap[0];

exit_tuple = TUPLE3(hp, am_EXIT, p->id, reason);

exit_tuple_sz = size_object(exit_tuple);

{
    ExitLinkContext context = {p, reason, exit_tuple, exit_tuple_sz};
    erts_sweep_links(lnk, &doit_exit_link, &context);
}
    }

    {
ExitMonitorContext context = {reason, p};
erts_sweep_monitors(mon,&doit_exit_monitor,&context);
    }

    delete_process(p);

#ifdef ERTS_ENABLE_LOCK_CHECK
    erts_smp_proc_lock(p, ERTS_PROC_LOCK_MAIN); /* Make process_main() happy */
    ERTS_SMP_CHK_HAVE_ONLY_MAIN_PROC_LOCK(p);
#endif
```

```
}
```

这些资源是进程拥有的 它有义务释放这些资源。 所以如果发现某些资源被莫名其妙的清除，请检查是不是进程异常退出！

1.76 The Top Ten Erlang News Stories of 2008

发表时间: 2009-01-13

Erlang Inside: The Top Ten Erlang News Stories of 2008
from Planet Trapexit - Erlang/OTP News

http://erlanginside.com/the-top-ten-erlang-news-stories-of-2008-69

It goes without saying that 2008 was a difficult year for many with 2009 looking to be more of the same. But for the Erlang community, it was probably the best year since Joe Armstrong released Programming Erlang. OK, that was 2007... still, it' s a great time to use Erlang. Here are the top ten Erlang related 'events' of 2008, from ' smallest' to 'biggest'. This is my subjective ranking with a focus on the trends that will introduce the most number of developers to the power of Erlang in 2009.

10. Erlang jobs more than Quadrupled in the UK last year, according to IT Jobs Watch. I believe similar results in the US but don' t (yet) have the data to prove it.

9. Sites like LangPop.com recognize that languages like Erlang and Haskell are being talked about more than they' re being used (so far). For instance, Erlang has passed C# and Visual Basic in discussion frequency and sites such as SD Times cover more stories on Erlang

8. In the Top-Ten Self Promotion Department, Erlang Inside releases and tries to fill the gap in Erlang related news sites.

7. Amazon uses more Erlang with its SimpleDB release. OK, this was in Beta in 2007 but not officially released until 2008.

6. Pragmatic Programmers release a set of brilliant Erlang Screencasts.

5. Reia is released in alpha, and provides a scripting language with the power of the Erlang runtime and VM, as covered on Erlang Inside.

4. RabbitMQ - See this Google Talk about RabbitMQ.

3. Nitrogen Web Framework for Erlang - As covered earlier this year, Rusty has come a long way

recently with a great looking website including screencasts showing the power of the framework.

2. CouchDB - CouchDB became the schema-less DB of choice for a growing number of users, was inducted into the Apache Incubator, and everyone starts discussions about it with “Did you know it was written in Erlang?”.

1. And Finally... Facebook launches Chat to 70 Million Users... All At Once. Using Erlang. I can’t think of a better advertisement for the power of Erlang (outside of the Telco space) than the successful release of this feature.

Thoughts on this list? Missed something big? Add other ideas in the comments...

1.77 Erlang ERTS的Trap机制的设计及其用途

发表时间: 2009-02-18 关键字: bif_trap

erlang的trap机制在实现中用的很多，在费时的BIF操作中基本上都可以看到。它的实现需要erl vm的配合。它的作用基本上有3个：

1. 把费时操作分阶段做。由于erlang是个软实时系统，一个进程或者bif不能无限制的占用cpu时间。所以erlang的每个进程执行的时候，最多只能执行一定数量的指令.这个是设计方面的目标。实现上也要配套。所以比如md5,list_member查找这种可能耗时的操作都是用trap机制来实现的，也就是说 当进程调度到的时候 执行一定数量的计算 然后把上下文trap起来 放弃执行 等待下一次的调度 来继续计算。

2. 延迟执行，实现上层的决策。 明显的例子是 send操作。 send的时候 节点间可能未连接，所以这个send的操作不能继续，先trap, 然后在下一次的调度的时候 执行节点连接操作，一旦成功 send操作就继续往下执行。对客户来讲这个操作是透明的。他不知道你幕后的这些事情。

3. 主动放弃CPU yield.

erlang设计还是蛮细致的！

PS：涉及到费时操作的BIF有：
do_bif_utf8_to_list
ets_delete_1
spawn_3
monitor_2
spawn_link_3
spawn_opt_1
send_2
crc32_1
adler32_1
md5_1
send_3
build_utf8_return
build_list_return
finalize_list_to_list
do_bif_utf8_to_list
ets_select_reverse
ets_match_spec_run_r_3

re_run_3

re_exec_trap

keyfind

monitor_node_3.

1.78 Tentative new functions in R13B

发表时间: 2009-02-19 关键字: tentative new functions in r13b

Next major release R13B
The next major release R13B is planned for April 2009
A beta called R13A planned for March
Service releases approximately every second month

计划改进的性能：
1 SMP with multiple run-queues and other optimizations
2 re, new regular expression implementation officially supported
3 More features in the " standalone" Erlang direction
4 Completed the distribution of doc source with built support to produce html and pdf.
5 WxWidgets based GUI library included in the distribution, plan to remove GS from R14
6 Major XML improvements, both speed and functions
7 Unicode support as described in EEP-10
8 Fast search in binaries
9 FFI, Foreign Function Interface or loadable BIF' s
10 Scanner which can preserve complete source (withespace, comments)
11 Megaco improved SMP performance

其中我最感兴趣的是 1，3，5，9. 特别是1对提高整个系统的性能太重要了。

参考附件
附件下载:

- ErlangOTPNews2.pdf (24.4 KB)
- dl.javaeye.com/topics/download/ac9eeeec-1235-3b8b-a616-b9faadd48600

1.79 Erlang Programming for Multi-core 多核编程必读

发表时间: 2009-02-20 关键字: erlang programming for multi-core ulf wiger

The purpose of this presentation is to give a hands-on tutorial on Erlang programming for multi-core.今天闲逛的时候发现了这个PDF,作者Ulf Wiger， 非常强悍哦 ！同学们快看！！

附件下载:

- damp09-wiger-keynote.pdf (848.6 KB)
- dl.javaeye.com/topics/download/e9ddcb2c-528a-3f7e-af53-ef1d5075ac20

1.80 Erlang Message Receive Fundamentals

发表时间: 2009-02-23 关键字: erlang message receive fundamentals ulf

请参考 http://www.duomark.com/erlang/briefings/euc2006/index.html

Ulf大佬写的，质量有保证！

Conclusions About Erlang Receive

* Erlang makes it very easy to construct messaging systems
* Although easy to use, the language does not simplify hard problems
* Prefer to handle messages in the order received in stateless situations
* Multiple receive statements can lead to overlooked message patterns
* Always rely on receive to wait on messages rather than polling
* Work out a good protocol for your messaging
* Do not be afraid of using more processes to keep queues pure
* Sometimes acking of messages is a necessity

写的很好，相信看了后会对消息接收机制有了更深入的理解，改变我们的初级的对消息队列的使用习惯。

1.81 webtool添加了好几个模块

发表时间: 2009-02-24 关键字: webtool

不小心间发现webtool添加了好几个模块：

1.WebCover
2.OrberWeb
3.VisualTestServer
4.WebAppmon
5.CrashDumpViewer

都是很实用的功能哦！！

1.82 R13开始支持binary unicode

发表时间: 2009-02-27 关键字: binary unicode

1 。bitstring语法改动 添加了unicode数据类型

6.16 Bit Syntax Expressions

...

The types utf8, utf16, and utf32 specifies encoding/decoding of the Unicode Transformation Formats UTF-8, UTF-16, and UTF-32, respectively.

When constructing a segment of a utf type, Value must be an integer in one of the ranges 0..16#D7FF, 16#E000..16#FFFD, or 16#10000..16#10FFFF (i.e. a valid Unicode code point). Construction will fail with a badarg exception if Value is outside the allowed ranges. The size of the resulting binary segment depends on the type and/or Value. For utf8, Value will be encoded in 1 through 4 bytes. For utf16, Value will be encoded in 2 or 4 bytes. Finally, for utf32, Value will always be encoded in 4 bytes.

When constructing, a literal string may be given followed by one of the UTF types, for example: <<"abc"/utf8>> which is syntatic sugar for <<$a/utf8,$b/utf8,$c/utf8>>.

A successful match of a segment of a utf type results in an integer in one of the ranges 0..16#D7FF, 16#E000..16#FFFD, or 16#10000..16#10FFFF (i.e. a valid Unicode code point). The match will fail if returned value would fall outside those ranges.

A segment of type utf8 will match 1 to 4 bytes in the binary, if the binary at the match position contains a valid UTF-8 sequence. (See RFC-2279 or the Unicode standard.)

A segment of type utf16 may match 2 or 4 bytes in the binary. The match will fail if the binary at the match position does not contain a legal UTF-16 encoding of a Unicode code point. (See RFC-2781 or the Unicode standard.)

A segment of type utf32 may match 4 bytes in the binary in the same way as an integer segment matching 32 bits. The match will fail if the resulting integer is outside the legal ranges mentioned above.

....

2. 新增加了 binary_to_atom atom_to_binary等bif.

3. re模块也支持unicode匹配。

具体的请参看EEP10.

1.83 A new Erlang book is on it's way

发表时间: 2009-03-03 关键字: concurrent programming with erlang/otp

Concurrent Programming with Erlang/OTP by Martin Logan, Eric Merritt, Richard Carlsson and Robert Calco is now available from Manning Publications Co in an "Early Access Edition". Readers of the Erlang.org site will get a 35% discount by using erlang35 as the promotional code.

1.84 binary的这个bug R13A还没有修复

发表时间: 2009-03-03 关键字: binary

Edwin , thanks for your response!

2008/7/19 Edwin Fine <emofine@gmail.com>:

Litao,

I think this is a bug in Erlang R12B-3. Certainly the documentation can be misleading, because in Programming Examples (Section 4.6: Matching Binaries), it specifically says that this construct is not allowed:

"Size must be an integer literal, or a previously bound variable. Note that the following is not allowed:

```
foo(N, <<X:N,T/binary>>) ->
  {X,T}.
```

The two occurrences of N are not related. The compiler will complain that the N in the size field is unbound."

That being said, if you rewrite the expression as shown below, it will compile (but does not work). If I understand correctly, binary, bits, and bitstream are the same, except that the default bit size for binary is 8, and for bitstring it is 1. Since you are overriding the default size anyway, you can use binary-unit:11 in place of bits-unit:11.

```
decode(<<N:5,Chans:N/binary-unit:11,_/bits>>) ->
   [Chan || <<Chan:11>> <- Chans].


103> Chans3 = <<3:5,2:11,3:11,4:11>>.
<<24,2,0,96,4:6>>
104> bb:decode(Chans3).
N:3, Chans:<<0,64,12,2,0:1>>
** exception error: no case clause matching {<<0,64,12,2,0:1>>}
    in function  bb:'-decode/1-lc$^0/1-0-'/1
```

105>

The code is:

```
-module(bb).
-compile([export_all]).

decode(<<N:5,Chans:N/binary-unit:11,_/bits>>) ->
    io:format("N:~p, Chans:~p~n", [N, Chans]),
    [Chan || <<Chan:11>> <- Chans].
```

So even though the programming examples say that you can't use the same N in the match, it actually does work, but the list comprehension does not. I found out this is because a bitstring generator has to use "<=" and not "<-". So the final working code is:

```
-module(bb).
-compile([export_all]).

decode(<<N:5,Chans:N/binary-unit:11,_/bits>>) ->
    [Chan || <<Chan:11>> <= Chans].

118> c(bb).
{ok,bb}
119> Chans3 = <<3:5,2:11,3:11,4:11>>.
<<24,2,0,96,4:6>>
120> bb:decode(Chans3).
[2,3,4]
```

Hope this helps.

2008/7/18 litao cheng <litaocheng@gmail.com>:

- Hide quoted text -

hi, all.

when I read this paper: Programming Efficiently with Binaries and Bit Strings http://www.erlang.se/euc/07/papers/1700Gustafsson.pdf, I encounter a compile error, the code

snipes is a example to parse the IS 683-PRL protocol:

```
decode(<<N:5,Chans:N/bits-unit:11,_/bits>>) ->
    [Chan || <<Chan:11>> <- Chans].
```

the compiler says:
bit type mismatch (unit) between 11 and 1

I read the erlang reference mannual, the bits unit default is 1, I think the unit can be set, why this compile error occur? thank you!

my erlang emulator is 5.6.3(R12B-3).

---

erlang-questions mailing list
erlang-questions@erlang.org
http://www.erlang.org/mailman/listinfo/erlang-questions

难道他们不想修？？？？我等只能凑合用。

1.85 erlang shell commands

发表时间: 2009-03-03 关键字: shell commands

Shell Commands

b()
Prints the current variable bindings.
f()
Removes all variable bindings.
f(X)
Removes the binding of variable X.
h()
Prints the history list.
history(N)
Sets the number of previous commands to keep in the history list to N. The previous number is returned. The default number is 20.
results(N)
Sets the number of results from previous commands to keep in the history list to N. The previous number is returned. The default number is 20.
e(N)
Repeats the command N, if N is positive. If it is negative, the Nth previous command is repeated (i.e., e(-1) repeats the previous command).
v(N)
Uses the return value of the command N in the current command, if N is positive. If it is negative, the return value of the Nth previous command is used (i.e., v(-1) uses the value of the previous command).
help()
Evaluates shell_default:help().
c(File)
Evaluates shell_default:c(File). This compiles and loads code in File and purges old versions of code, if necessary. Assumes that the file and module names are the same.
catch_exception(Bool)
Sets the exception handling of the evaluator process. The previous exception handling is returned. The default (false) is to kill the evaluator process when an exception occurs, which causes the shell to create a new evaluator process. When the exception handling is set to true the evaluator process lives on which means that for instance ports and ETS tables as well as processes linked to the evaluator process survive the exception.

rd(RecordName, RecordDefinition)

Defines a record in the shell. RecordName is an atom and RecordDefinition lists the field names and the default values. Usually record definitions are made known to the shell by use of the rr commands described below, but sometimes it is handy to define records on the fly.

rf()

Removes all record definitions, then reads record definitions from the modules shell_default and user_default (if loaded). Returns the names of the records defined.

rf(RecordNames)

Removes selected record definitions. RecordNames is a record name or a list of record names. Use '_' to remove all record definitions.

rl()

Prints all record definitions.

rl(RecordNames)

Prints selected record definitions. RecordNames is a record name or a list of record names.

rp(Term)

Prints a term using the record definitions known to the shell. All of Term is printed; the depth is not limited as is the case when a return value is printed.

rr(Module)

Reads record definitions from a module's BEAM file. If there are no record definitions in the BEAM file, the source file is located and read instead. Returns the names of the record definitions read. Module is an atom.

rr(Wildcard)

Reads record definitions from files. Existing definitions of any of the record names read are replaced. Wildcard is a wildcard string as defined in filelib(3) but not an atom.

rr(WildcardOrModule, RecordNames)

Reads record definitions from files but discards record names not mentioned in RecordNames (a record name or a list of record names).

rr(WildcardOrModule, RecordNames, Options)

Reads record definitions from files. The compiler options {i, Dir}, {d, Macro}, and {d, Macro, Value} are recognized and used for setting up the include path and macro definitions. Use '_' as value of RecordNames to read all record definitions.

实用！

1.86 erlang节点间connect_all流程

发表时间: 2009-03-04 关键字: connect_all

The default when a connection is established between two nodes, is to immediately connect all other visible nodes as well. This way, there is always a fully connected network. If there are nodes with different cookies, this method might be inappropriate and the command line flag -connect_all false must be set, see erl(1).

从global.erl摘抄的...

```
%% Suppose nodes A and B connect, and C is connected to A.
%% Here's the algorithm's flow:
%%
%% Node A
%% ------
%% << {nodeup, B}
%%   TheLocker ! {nodeup, ..., Node, ...} (there is one locker per node)
%% B ! {init_connect, ..., {..., TheLockerAtA, ...}}
%% << {init_connect, TheLockerAtB}
%%   [The lockers try to set the lock]
%% << {lock_is_set, B, ...}
%%   [Now, lock is set in both partitions]
%% B ! {exchange, A, Names, ...}
%% << {exchange, B, Names, ...}
%%   [solve conflict]
%% B ! {resolved, A, ResolvedA, KnownAtA, ...}
%% << {resolved, B, ResolvedB, KnownAtB, ...}
%% C ! {new_nodes, ResolvedAandB, [B]}
%%
%% Node C
%% ------
%% << {new_nodes, ResolvedOps, NewNodes}
%%   [insert Ops]
%% ping(NewNodes)
%% << {nodeup, B}
```

%% <ignore this one>
%%

A和B之间先互相换信息，A告诉C有关B的信息， C会主动连接B的 这是在global模块实现的 而不是erts的一部分.

1.87 Hidden Nodes的用处

发表时间: 2009-03-04 关键字: hidden nodes

11.5 Hidden Nodes

In a distributed Erlang system, it is sometimes useful to connect to a node without also connecting to all other nodes. An example could be some kind of O&M functionality used to inspect the status of a system without disturbing it. For this purpose, a hidden node may be used.

A hidden node is a node started with the command line flag -hidden. Connections between hidden nodes and other nodes are not transitive, they must be set up explicitly. Also, hidden nodes does not show up in the list of nodes returned by nodes(). Instead, nodes(hidden) or nodes(connected) must be used. This means, for example, that the hidden node will not be added to the set of nodes that global is keeping track of.

This feature was added in Erlang 5.0/OTP R7.

设计的目的是减少节点见互联互通的通讯成本，不用global模块去跟踪这个节点的名字同步等。特别适合于用ei写的c程序， 因为ei是轻量的，没有这么多资源和必要去做无必须的事情。ei一般用作client, 去请求别的节点做复杂的运算！

1.88 Exit Reasons（备查）

发表时间: 2009-03-04 关键字: exit reasons

9.4 Exit Reasons

When a run-time error occurs, that is an exception of class error, the exit reason is a tuple {Reason,Stack}. Reason is a term indicating the type of error:
Exit Reasons. Reason Type of error
badarg Bad argument. The argument is of wrong data type, or is otherwise badly formed.
badarith Bad argument in an arithmetic expression.
{badmatch,V} Evaluation of a match expression failed. The value V did not match.
function_clause No matching function clause is found when evaluating a function call.
{case_clause,V} No matching branch is found when evaluating a case expression. The value V did not match.
if_clause No true branch is found when evaluating an if expression.
{try_clause,V} No matching branch is found when evaluating the of-section of a try expression. The value V did not match.
undef The function cannot be found when evaluating a function call.
{badfun,F} There is something wrong with a fun F.
{badarity,F} A fun is applied to the wrong number of arguments. F describes the fun and the arguments.
timeout_value The timeout value in a receive..after expression is evaluated to something else than an integer or infinity.
noproc Trying to link to a non-existing process.
{nocatch,V} Trying to evaluate a throw outside a catch. V is the thrown term.
system_limit A system limit has been reached. See Efficiency Guide for information about system limits.

Stack is the stack of function calls being evaluated when the error occurred, given as a list of tuples {Module,Name,Arity} with the most recent function call first. The most recent function call tuple may in some cases be {Module,Name,[Arg]}.

1.89 dist_auto_connect的作用

发表时间: 2009-03-04 关键字: dist_auto_connect

Normally, connections are established automatically when another node is referenced（也就是说给节点发信息 ping rpc等等）. This functionality can be disabled by setting the Kernel configuration parameter dist_auto_connect to false, see kernel(6). In this case, connections must be established explicitly by calling net_kernel:connect_node/1.

这个参数保证了只是你想explictly连接的节点才在你的nodes()列表里面.因为不需要的节点需要维护 发tick包什么的 浪费资源！

节点间如何established automatically请参考另外一篇 http://mryufeng.javaeye.com/admin/blogs/120666

1.90 Internal Representation of Records

发表时间: 2009-03-06 关键字: internal representation of records

8.7 Internal Representation of Records

Record expressions are translated to tuple expressions during compilation. A record defined as

-record(Name, {Field1,...,FieldN}).

is internally represented by the tuple

{Name,Value1,...,ValueN}

where each ValueI is the default value for FieldI.

To each module using records, a pseudo function is added during compilation to obtain information about records:

record_info(fields, Record) -> [Field]
record_info(size, Record) -> Size

Size is the size of the tuple representation, that is one more than the number of fields.

1.91 .erlang文件 系统自动加载运行

发表时间: 2009-03-07 关键字: .erlangrc

programing erlang书里提到了 官方文档貌似没写！

执行的流程比较搞：

start.script

....

{apply,{c,erlangrc,[]}},

....

c.erl

....

```
%% erlangrc(Home)
%%  Try to run a ".erlang" file, first in the current directory
%%  else in home directory.

erlangrc() ->
    case init:get_argument(home) of
{ok,[[Home]]} ->
    erlangrc([Home]);
_ ->
    f_p_e(["."], ".erlang")
    end.
```

...

init运行的时候会执行 {apply,{c,erlangrc,[]}}, 也就是 c.erlangrc() 在用户的目录查找.erlang文件来执行。

所以有需要在erlang系统加载自动运行的东西可以写在.erlang文件里！

```
yufeng@yufeng-desktop:~$ cat .erlang
io:format("hello ~n",[]).
yufeng@yufeng-desktop:~$ erl
Erlang R13A (erts-5.7) [source] [rq:1] [async-threads:0] [hipe] [kernel-poll:false]
```

```
hello
Eshell V5.7  (abort with ^G)
1>
```

1.92 5 tty - A command line interface

发表时间: 2009-03-08 关键字: shell command line edit

记住这些快捷键shell操作的时候速度快很多！

5 tty - A command line interface

tty is a simple command line interface program where keystrokes are collected and interpreted. Completed lines are sent to the shell for interpretation. There is a simple history mechanism, which saves previous lines. These can be edited before sending them to the shell. tty is started when Erlang is started with the command:

erl

tty operates in one of two modes:

* normal mode, in which lines of text can be edited and sent to the shell.
* shell break mode, which allows the user to kill the current shell, start multiple shells etc. Shell break mode is started by typing Control G.

5.1 Normal Mode

In normal mode keystrokes from the user are collected and interpreted by tty. Most of the emacs line editing commands are supported. The following is a complete list of the supported line editing commands.

Note: The notation C-a means pressing the control key and the letter a simultaneously. M-f means pressing the ESC key followed by the letter f.
tty text editing Key Sequence Function
C-a Beginning of line
C-b Backward character
M-b Backward word
C-d Delete character
M-d Delete word
C-e End of line
C-f Forward character

M-f Forward word
C-g Enter shell break mode
C-k Kill line
C-l Redraw line
C-n Fetch next line from the history buffer
C-p Fetch previous line from the history buffer
C-t Transpose characters
C-y Insert previously killed text
5.2 Shell Break Mode

tty enters shell break mode when you type Control G. In this mode you can:

* Kill or suspend the current shell
* Connect to a suspended shell
* Start a new shell

1.93 Erlang for Concurrent Programming by Jim Larson

发表时间: 2009-03-08 关键字: erlang for concurrent programming

Erlang for Concurrent Programming
Designed for concurrency from the ground up, the Erlang language can be a valuable tool to help solve concurrent problems.
Jim Larson, Google

写的还不错哦！
附件下载:

- p26-larson.pdf (207.9 KB)
- dl.javaeye.com/topics/download/bc43d1f2-cbbb-3192-a1ef-9ef17ddc426a

1.94 erlang及其应用PPT

发表时间: 2009-03-11 关键字: erlang application

吹吹水 Erlang适合做什么 能做什么 有什么优点 凑合看吧！

附件下载:

- Erlang及其应用.rar (13.2 KB)
- dl.javaeye.com/topics/download/f70dabbd-bef2-379e-97e5-5b70a30100e3

1.95 release_handler底层指令 热部署的实际执行者

发表时间: 2009-03-12 关键字: release_handler

下面的指令是热部署干活的指令：

```
%%----------------------------------------------------------------
%% An unpurged module is a module for which there exist an old
%% version of the code.  This should only be the case if there are
%% processes running the old version of the code.
%%
%% This functions evaluates each instruction.  Note that the
%% instructions here are low-level instructions.  e.g. lelle's
%% old synchronized_change would be translated to
%%   {load_object_code, Modules},
%%   {suspend, Modules}, [{load, Module}],
%%   {resume, Modules}, {purge, Modules}
%% Or, for example, if we want to do advanced external code change
%% on two modules that depend on each other, by killing them and
%% then restaring them, we could do:
%%   {load_object_code, [Mod1, Mod2]},
%%   % delete old version
%%   {remove, {Mod1, brutal_purge}}, {remove, {Mod2, brutal_purge}},
%%   % now, some procs migth be running prev current (now old) version
%%   % kill them, and load new version
%%   {load, {Mod1, brutal_purge}}, {load, {Mod2, brutal_purge}}
%%   % now, there is one version of the code (new, current)
%%
%% NOTE: All load_object_code must be first in the script,
%%       a point_of_no_return must be present (if load_object_code
%%       is present).
%%
%% {load_object_code, {Lib, LibVsn, [Mod]}
%%    read the files as binarys. do not make code out of them
%% {load, {Module, PrePurgeMethod, PostPurgeMethod}}
%%    Module must have been load_object_code:ed.  make code out of it
%%    old procs && soft_purge => no new release
```

```
%%    old procs && brutal_purge => old procs killed
%%    The new old code will be gc:ed later on, if PostPurgeMethod =
%%    soft_purge.  If it is brutal_purge, the code is purged when
%%    the release is made permanent.
%% {remove, {Module, PrePurgeMethod, PostPurgeMethod}}
%%    make current version old.  no current left.
%%    old procs && soft_purge => no new release
%%    old procs && brutal_purge => old procs killed
%%    The new old code will be gc:ed later on, if PostPurgeMethod =
%%    soft_purge.  If it is brutal_purge, the code is purged when
%%    the release is made permanent.
%% {purge, Modules}
%%    kill all procs running old code, delete old code
%% {suspend, [Module | {Module, Timeout}]}
%%    If a process doesn't repsond - never mind.  It will be killed
%%    later on (if a purge is performed).
%%    Hmm, we must do something smart here... we should probably kill it,
%%    but we cant, because its supervisor will restart it directly!  Maybe
%%    we should keep a list of those, call supervisor:terminate_child()
%%    when all others are suspended, and call sup:restart_child() when the
%%    others are resumed.
%% {code_change, [{Module, Extra}]}
%% {code_change, Mode, [{Module, Extra}]}  Mode = up | down
%%    Send code_change only to suspended procs running this code
%% {resume, [Module]}
%%    resume all previously suspended processes
%% {stop, [Module]}
%%    stop all procs running this code
%% {start, [Module]}
%%    starts the procs that were previously stopped for this code.
%%    Note that this will start processes in exactly the same place
%%    in the suptree where there were procs previously.
%% {sync_nodes, Id, [Node]}
%% {sync_nodes, Id, {M, F, A}}
%%    Synchronizes with the Nodes (or apply(M,F,A) == Nodes).  All Nodes
%%    must also exectue the same line.  Waits for all these nodes to get
%%    to this line.
%% point_of_no_return
```

```
%% restart_new_emulator
%% {stop_application, Appl}      - Impl with apply
%% {unload_application, Appl}    - Impl with {remove..}
%% {load_application, Appl}      - Impl with {load..}
%% {start_application, Appl}     - Impl with apply
%%----------------------------------------------------------------
```

见release_handler_1.erl, 指令不是很多，基本上是bif的封装。

1.96 程序语言的新星:走近Erlang的世界

发表时间: 2009-03-16 关键字: erlang 并行计算 面向过程语言 程序语言 函数式 面向对象语言

开发者在线 Builder.com.cn 更新时间:2008-06-12作者：顾宏军 来源:软件世界
本文关键词： Erlang 并行计算 面向过程语言 程序语言 函数式 面向对象语言

（文章写的相当不错 很好的介绍了erlang的优点）

提起Erlang语言，相信许多人都会挠头，因为它实在是太陌生了。在2007年6月由TIOBE Programming Community提供的程序语言排名中，Erlang占有率仅为0.08%，排名第49位。与之形成鲜明对比的是，Java以20.025%的占有率高居榜首，紧随其后的是C（15.967%）、C++（11.118%）、VB （9.332%）、PHP（8.871%）、Perl（6.177%）、C#（3.483%）、Python（3.161%）、JavaScript （2.616%）和Ruby（2.132%）。相对于传统老牌“大佬”语言相比，Erlang语言绝对算得上是一种“小众”语言，但其未来的发展前景却是无法估量的，因为它可以解决传统语言很难解决在并行计算中的难题，甚至有专家预言可能成为下一个Java，在正在迅猛发展的并行计算时代，Erlang将会迅速的崛起。

认识Erlang

Erlang并非一门新语言，它出现于1987年，只是当时对并发、分布式需求还没有今天这么普遍，当时可谓英雄无用武之地。Erlang语言创始人Joe Armstrong当年在爱立信做电话网络方面的开发，他使用Smalltalk，可惜那个时候Smalltalk太慢，不能满足电话网络的高性能要求。但Joe实在喜欢Smalltalk，于是定购了一台Tektronix Smalltak机器。但机器要两个月时间才到，Joe在等待中百无聊赖，就开始使用Prolog，结果等Tektronix到来的时候，他已经对 Prolog更感兴趣，Joe当然不满足于精通Prolog，经过一段时间的试验，Joe给Prolog加上了并发处理和错误恢复，于是Erlang就诞生了。这也是为什么Erlang的语法和Prolog有不少相似之处，比如它们的List表达都是[Head | Tail]。

1987年Erlang测试版推出，并在用户实际应用中不断完善，于1991年向用户推出第一个版本，带有了编译器和图形接口等更多功能。 1992年，Erlang迎来更多用户，如RACE项目等。同期Erlang被移植到 VxWorks、PC和 Macintosh等多种平台，两个使用Erlang的产品项目也开始启动。1993爱立信公司内部独立的组织开始维护和支持Erlang实现和 Erlang工具。

目前，随着网络应用的兴起，对高并发、分布部署、持续服务的需求增多，Erlang的特性刚好满足这些需求，于是Erlang开始得到更多人的关注。

Erlang特性

Erlang是一种函数式语言，使用Erlang编写出的应用运行时通常由成千上万个轻量级进程组成，并通过消

息传递相互通讯。使用 Erlang来编写分布式应用比其它语言简单许多，因为它的分布式机制是透明的，即对于程序而言并不知道自己是在分布式运行。Erlang运行环境是一个虚拟机，有点类似于Java虚拟机，代码一经编译，同样可以随处运行。它的运行时系统甚至允许代码在不被中断的情况下更新。另外如果需要更高效的话，字节代码也可以编译成本地代码运行。

Erlang的结构图

相较于其它语言，Erlang有很多天生的适应现代网络服务需求的特性:

◆并发性，Erlang具有超强的轻量级进程，这种进程对内存的需求是动态变化的，并且它没有共享内存和通过异步消息传送的通讯。Erlang支持超大量级的并发线程，并且不需要操作系统具有并发机制。

◆分布式，Erlang被设计用于运行在分布式环境下。一个Erlang虚拟机被成为Erlang节点。一个分布式Erlang系统是多个 Erlang节点组成的网络（通常每个处理器被作为一个节点）。一个Erlang节点能够创建运行在其它节点上的并行线程，而其它节点可以使用其余的操作系统。线程依赖不同节点之间的通讯，这完全和它依赖于单一节点一样。

◆ 软实时性 Erlang支持可编程的“软”实时系统，这种系统需要反应时间在毫秒级。而在这种系统中，长时间的垃圾收集（garbage collection）延迟是无法接受的，因此Erlang使用了递增式垃圾收集技术。

◆ 热代码升级 一些系统不能由于软件维护而停止运行。Erlang允许程序代码在运行系统中被修改。旧代码能被逐步淘汰而后被新代码替换。在此过渡期间，新旧代码是共存的。这也使得安装Bug补丁、在运行系统上升级而不干扰系统操作成为了可能。

◆ 递增式代码装载 用户能够控制代码如何被装载的细节。在嵌入式系统中，所有代码通常是在启动时就被完全装载。而在开发系统中，代码是按需装载的，甚至在系统运行时被装载。如果测试到了未覆盖的Bug，只需替换具有Bug的代码即可。

Erlang应用场合

未来的计算是并发计算。现今甚至桌面CPU也是多核的，当用户给服务器购买了越来越多的CPU时，他们更期望能最大限度地利用他们的新投资，但是今天的许多软件系统并不能很好地做到这一点。

整个软件行业也在发生重大变革，由卖工具软件转向卖服务（软件免费，这也是开源软件兴起的过程），由单纯客户端向B/S或C/S转化，相应的存储和计算向服务器端转移，由原来的PC客户端向客户端多元化（如手机、PDA、电视机顶盒等）转化。这些变革趋势，使得用户可以更方便地访问到服务的同时，服务器也要承受越来越高的负荷，并行/分布的需求逐渐增加。

Erlang语言不是用来解决所有问题的语言，至少现在还不是。Erlang最初专门为通信应用设计的，比如控制交换机或者变换协议等，非常适合于构建分布式，实时软并行计算系统。它是一门专注的语言，可以适应现代服务器要求高负荷、高可靠、持续服务的需求。它要解决的问题域包括:高并发、分布式、持续服务、热升级和高可靠等问题。

Erlang应用实例

典型的Erlang应用是由很多被分配不同任务的“节点(Node)”组成的“集群 (Cluster)”。一个Erlang节点就是一个Erlang虚拟机的实例，用户可以在一台机器(服务器、台式机或者笔记本)上运行多个节点。 Erlang节点自动跟踪所有连接着的其他节点。要添加一个节点仅仅需要将其指向任何一个已建节点就可以了。只要这两个节点建立了连接，所有其他节点马上就会感应到新加入的节点。Erlang进程使用进程ID向其他进程传递报文，进程ID包含着运行此进程的节点信息。因此进程不需要理会正在与其交流的其他进程实际在何处运行。一组相互连接的Erlang节点可以看作是一个网格计算体或者一台超级计算机。

erlang的odbc应用程序结构图

Yaws是一个Erlang写的Web服务器。ErLang本身带有一个HTTP Server,叫做inet。Yaws对于inet，就相当于Servlet对于Http Server。Yaws也可说是一个Web开发框架，Yaws的ehtml类似于jsp、 php、ruby template。Yaws并发能力是Apache的15倍，有人利用16台集群服务器所做的显示，Yaws可以承受超八万并发活动，Apache在四千就宕机了。

erlang和ruby的简单测试

Ejabberd也是Erlang很好的应用实例，也是目前可扩展性最好的一种 Jabber/XMPP服务器，支持分布多个服务器，并且具有容错处理，单台服务器失效不影响整个集群运作。Ejabberd基于ErLang+ Mnesia构建，项目已成功发展5年，占据30%左右Jabber服务器市场。

Tsung则是多协议分布式压力测试工具,可用于测试Http、Soap、Postgresql和Jabber/XMPP服务器。而Wings则是一个3D建模程序，软件支持Windows、Mac OSX和Linux等操作系统，这两个项目都基于Erlang构建。

Erlang将会成为一个非常重要的语言。如果有了大公司的支持，它甚至可能成为下一个Java。因为它是个开源项目，非常适合多核处理、Web服务等领域。事实上，它也是编写在多核机器上运行的高可靠性系统的唯一成熟语言。

Erlang始于20年前，是一个并发性Prolog，Joe Armstrong创造了它。第一个大型Erlang项目是一个由几百人创建的电信交换系统，系统有数百万行代码。系统主要关注的就是可靠性，并且系统有难以置信的可靠性历史。据Joe介绍，“它有99.9999999%的可靠性”。

这意味着每10亿秒才有1秒宕机时间，或者说10亿分钟有1分钟宕机时间。十亿秒大概是30年，10亿分钟大概有2000年。99.999%的可靠性大概是每年宕机5分钟，这已经是很好的了。了解可靠性的人都知道，可靠性系统有 99.9999%的，甚至99.99999%的，但是估计没听过有99.9999999%可靠性的，可基于Erlang的系统实现了。

但这还不是令Erlang壮大的理由，因为不是什么人都关注可靠性。也不是因为 Erlang是一个函数式语言，更不是并行Erlang是个面向对象语言。其发展迅速的主要原因是唯一一个有可靠实现和完善类库的成熟的并行开发语言，在不久的将来所有的桌面系统、笔记本电脑都将是多核的，而要让程序在多核上更快的运行就要使程序能充分利用多核处理的能力。

Erlang带有一组类库。多数类库是用于构建各类Internet服务的。 Erlang有Web服务器和数据库。Erlang社区认为它是构建可靠Web服务器和Web服务的首选语言。Erlang是一个构建可靠系统的框架/平台，它构建的平台可以持续运行而无需关闭，可以每天更新软件，甚至可以定期的更换硬件。这些特性是电信应用所需要的，它还是在线银行、在线商城等各类在线应用所迫切需要的。

Joe Armstrong最近写了本书《Programming.Erlang》,所有关注Erlang的人都值得一读。Erlang符合所有面向对象语言特性，虽然它是个函数式语言，而不是面向对象语言。Erlang区分与面向对象语言的一个方面就是它的错误处理。在某消息出错时，进程不是抛出出错的部分，而是直接进程纠错。系统结构被设计为底部是工作进程（它们可能会失败），上层是管理进程，它们可以重新启动失败的进程。

我不相信其它语言能迅速赶上Erlang。对其它语言而言，加入像Erlang这样的语言特征是很容易的。但这将花费他们大量的时间构建一个高质量的VM和成熟的并发性与可靠性类库。因此Erlang很自然会成功。如果将来要在多核系统上进行开发，Erlang是非常理想的选择。

Erlang在中国

目前，Erlang在全球都还是个小众语言，其在中国影响力就更小了，好在有国内的 Erlang爱好者已经组织起来，在进行相关的工作，成立了Erlang-china.org，发布了部分Erlang相关中文文档，并且组织了两次Erlang爱好者聚会，Erlang-China.org将继续为对Erlang感兴趣的中文用户提供便利，促进用户彼此之间的交流，推动对这一语言的深入研究，促成一些Erlang开源项目，帮助中文用户为整个Erlang社区做出贡献。

Erlang没有类似Java、C++的语法，它不是面向对象语言，它是函数编程语言(Functional programming Language)。大量程序员并不熟悉函数式编程，我们的计算机教育里也都是基于面向对象和面向过程语言的，这会是所有想尝试Erlang的用户遇到的首要问题，这会使得培训成本加大，决策人员也需要足够勇气来选择一个新语言来构建应用。

另外，Erlang虽然内建了并行、分布的支持，但是程序员还需要学习和掌握并行的思维模式，并行的思维

模式也许是更加难以跨越的门槛。

要解决计算时代，可伸缩性、容错性以及运行时可更新系统需求，就目前而言，只有 Erlang语言可以很好的解决。Erlang语言也正面临这一场大的变革，从默默无闻走向更多人视野，会向更广的网络应用领域渗透。也许，不久的将来，当你听到Erlang时，就如同听说Java一样平常。

1.97 借鉴erlang odbc架构写外部接口

发表时间: 2009-03-16 关键字: odbc architecture

Architecture of the Erlang odbc application

(对于我们编写c接口很大的借鉴意义,极大的提高了系统的稳定性，数据通信用tcp socket而不是pipe 提高通讯速度，port机制为了扑捉c程序的非正常退出, controlling进程用于提供timeout通知，数据格式直接用term, c程序直接用ei构造和解码格式 一切看起来都简单 设计的太精妙了)

4 Error handling

4.1 Strategy

On a conceptual level starting a database connection using the Erlang ODBC API is a basic client server application. The client process uses the API to start and communicate with the server process that manages the connection. The strategy of the Erlang ODBC application is that programming faults in the application itself will cause the connection process to terminate abnormally.(When a process terminates abnormally its supervisor will log relevant error reports.) Calls to API functions during or after termination of the connection process, will return {error, connection_closed}. Contextual errors on the other hand will not terminate the connection it will only return {error, Reason} to the client, where Reason may be any erlang term.

4.1.1 Clients

The connection is associated with the process that created it and can only be accessed through it. The reason for this is to preserve the semantics of result sets and transactions when select_count/[2,3] is called or auto_commit is turned off. Attempts to use the connection from another process will fail. This will not effect the connection. On the other hand, if the client process dies the connection will be terminated.

4.1.2 Timeouts

All request made by the client to the connection are synchronous. If the timeout is used and expires the client process will exit with reason timeout. Proably the right thing to do is let the client die and perhaps be restarted by its supervisor. But if the client chooses to catch this timeout, it is a good idea to wait a little while before trying again. If there are too many consecutive timeouts that are caught the connection process will conclude that there is something radically wrong and terminate the connection.

4.1.3 Gaurds

All API-functions are guarded and if you pass an argument of the wrong type a runtime error will

occur. All input parameters to internal functions are trusted to be correct. It is a good programming practise to only distrust input from truly external sources. You are not supposed to catch these errors, it will only make the code very messy and much more complex, which introduces more bugs and in the worst case also covers up the actual faults. Put your effort on testing instead, you should trust your own input.

4.2 The whole picture

As the Erlang ODBC application relies on third party products and communicates with a database that proably runs on an other computer in the network there are plenty of things that might go wrong. To fully understand the things that might happen it facilitate to know the design of the Erlang ODBC application, hence here follows a short description of the current design.

Note

Please note that design is something, that not necessarily will, but might change in future releases. While the semantics of the API will not change as it is independent of the implementation.

odbc_app_arc

Architecture of the Erlang odbc application

When you do application:start(odbc) the only thing that happens is that a supervisor process is started. For each call to the API function connect/2 a process is spawned and added as a child to the Erlang ODBC supervisor. The supervisors only tasks are to provide error-log reports, if a child process should die abnormally, and the possibility to do a code change. Only the client process has the knowledge to decide if this connection managing process should be restarted.

The erlang connection process spawned by connect/2, will open a port to a c-process that handles the communication with the database through Microsoft's ODBC API. The erlang port will be kept open for exit signal propagation, if something goes wrong in the c-process and it exits we want know as mush as possible about the reason. The main communication with the c-process is done through sockets. The C-process consists of two threads, the supervisor thread and the database handler thread. The supervisor thread checks for shutdown messages on the supervisor socket and the database handler thread receives requests and sends answers on the database socket. If the database thread seems to hang on some database call, the erlang control process will send a shutdown message on the supervisor socket, in this case the c-process will exit. If the c-process crashes/exits it will bring the erlang-process down too and vice versa i.e. the connection is terminated.

Note

The function connect/2 will start the odbc application if that is not already done. In this case a supervisor information log will be produced stating that the odbc application was started as a

temporary application. It is really the responsibility of the application that uses the API too make sure it is started in the desired way.

4.2.1 Error types

The types of errors that may occur can be divide into the following categories.

* Configuration problems - Everything from that the database was not set up right to that the c-program that should be run through the erlang port was not compiled for your platform.
* Errors discovered by the ODBC driver - If calls to the ODBC-driver fails due to circumstances that can not be controlled by the Erlang ODBC application programmer, an error string will be dug up from the driver. This string will be the Reason in the {error, Reason} return value. How good this error message is will of course be driver dependent. Examples of such circumstances are trying to insert the same key twice, invalid SQL-queries and that the database has gone off line.
* Connection termination - If a connection is terminated in an abnormal way, or if you try to use a connection that you have already terminated in a normal way by calling disconnect/1, the return value will be{error, connection_closed}. A connection could end abnormally because of an programming error in the Erlang ODBC application, but also if the ODBC driver crashes.
* Contextual errors - If API functions are used in the wrong context, the Reason in the error tuple will be a descriptive atom. For instance if you try to call the function last/[1,2] without first calling select_count/[2,3] to associate a result set with the connection. If the ODBC-driver does not support some functions, or if you disabled some functionality for a connection and then try to use it.

附图

1.98 R13A 新增Reltool模块

发表时间: 2009-03-18 关键字: reltool

Reltool Reference Manual
Version 0.2

Reltool is a release management tool. It analyses a given Erlang/OTP installation and determines various dependencies between applications. The graphical frontend depicts the dependencies and enables interactive customization of a target system. The backend provides a batch interface for generation of customized target systems.

在研究干嘛用的！

1.99 Lies, Damned Lies, and Benchmarks（R13A smp性能测试）

发表时间: 2009-03-19 关键字: r13a smp 性能 提升

原文地址： http://www.erlangatwork.com/2009/03/lies-damned-lies-and-benchmarks.html

Erlang/OTP R13A was released today with a number of major SMP improvements. I've been playing with R13 snapshots for a while and wrote a simple HTTP server to compare the SMP performance on R12 and R13. This server uses {packet, http} to decode requests, increments a counter with a transactional mnesia:read/3 and mnesia:write/1, and responds with the counter's previous value. You'll find the source here.

I ran the HTTP server on a x86_64 CentOS 5 machine running Linux 2.6.18-53.el5. The server has two quad-Core Intel Xeon E5450 CPUs and 8GB of RAM. Erlang/OTP R12B-5 and R13A were compiled from source and run as erl -pa ebin +SN -s ehttpd start where N indicated the number of schedulers to run.

To get performance numbers I ran ab on another server connected via a 100 Mb/s private VLAN as ab -c N -n 100000 http://10.0.0.32:8889/ where N was the number of concurrent requests. ab was run 3 times for each value of N and the following chart shows the average requests/sec with 4 and 8 schedulers.

[img]schedulers.png [/img]

R13A's SMP improvements include multiple run queues and improved locking. It also supports binding schedulers to specific CPU cores and hardware threads. Binding isn't enabled by default, so the following chart shows the result of setting erlang:system_flag(scheduler_bind_type, thread_no_node_processor_spread) and running with 100 concurrent requests.

[img]requests_sec.png [/img]

There is a lot missing from these benchmarks, I didn't test kernel polling and only generated load from one client machine. The drop between 500 and 1000 concurrent requests on R13A +S8 looks too steep and may be the result of using ab. That said, the SMP optimizations in R13 are looking very promising!

根据我在ecug上做的实验：8核心的cpu

```
[spawn(ring, run,[["100", "10000000000"]]) || _X <- lists:seq(1,1000)].

R12B5:
CPU  User%  Sys% Wait% Idle|0         |25        |50        |75       100|                                  3
3 1  21.3  62.4   0.0   16.3|UUUUUUUUUUssssssssssssssssssssssssssssss
>|                                    3
3 2  20.9  61.7   0.0   17.4|UUUUUUUUUUsssssssssssssssssssssssssssss      >
|                                   3
3 3  19.9  63.2   0.0   16.9|UUUUUUUUUUssssssssssssssssssssssssssssss
>                                    3
3 4  18.9  64.2   0.0   16.9|UUUUUUUUUUssssssssssssssssssssssssssssss
>                                    3
3 5  19.9  62.7   0.0   17.4|UUUUUUUUUUssssssssssssssssssssssssssssss
>                                    3
3 6  20.9  63.2   0.0   15.9|UUUUUUUUUUssssssssssssssssssssssssssssss
>                                    3
3 7  19.4  62.7   0.0   17.9|UUUUUUUUUUsssssssssssssssssssssssssssss      >
|                                   3
3 8  19.4  63.7   0.0   16.9

R13A:

CPU  User%  Sys% Wait% Idle|0         |25        |50        |75       100|                                  3
3 1  61.2  31.8   0.0    7.0|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssssssssss >
|                                   3
3 2  64.7  29.9   0.0    5.5|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssssssssss >
|                                   3
3 3  62.7  29.9   0.0    7.5|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssssssssss >
|                                   3
3 4  61.0  32.5   0.0    6.5|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssssssssss >
|                                   3
3 5  62.5  30.5   0.0    7.0|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssssssssss >
|                                   3
3 6  64.2  29.4   0.0    6.5|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssssssssss
>|                                    3
3 7  63.7  29.9   0.0    6.5|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssssssssss >
|                                   3
3 8  65.7  27.9   0.0    6.5|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssssssssss >
```

| 3
3 +----------+ 3
3Avg 63.2 30.2 0.0 6.5

sys的调用主要是futex 所有对锁的依赖大量减少！

结论： 速度提高了将近2倍 效果真的很好yeah!

1.100 Dynamically sizing a fragmented mnesia store

发表时间: 2009-03-20 关键字: mnesia fragment

原文地址： http://blog.socklabs.com/2008/02/06/dynamically_sizing_a_fragmente.html

Not too long ago Mark Zweifel introduced me to Erlang. There are plenty of websites that explain what it is, how it works and where it came from so I'm not going to go into those details right here.

What I've really been having fun with is mnesia, the native data storage application. With it you can do some really powerful things like advanced data distribution, federation, complex queriers, full/compartmental replication and the list goes on and on.

The documentation is pretty good and Joe Armstrong's book Programming Erlang: Software for a Concurrent World has a huge collection of example snippets putting many of the concepts in action. What isn't covered includes a lot of the advanced mnesia subjects. These include fragmented tables, multi-node datastores and events.

So over the past few days I've been plowing through a lot of this and figuring out how it works. The first on the was federated/fragmented tables.

Consider a grid of three erlang nodes connected to each other with a common cookie. For the sake of this example each node lives in its own directory on the same server, but this example could apply to nodes on more than one machine. In this example the hostname is set to 'icedcoffee'.

ic:~/tmp/erl1 $ erl -sname node1 -setcookie 622c84b69ee448a07d80de5cbeb13e3d

ic:~/tmp/erl2 $ erl -sname node2 -setcookie 622c84b69ee448a07d80de5cbeb13e3d

ic:~/tmp/erl3 $ erl -sname node3 -setcookie 622c84b69ee448a07d80de5cbeb13e3dTo create our setup we want to define a record and create a fragmented table over nodes 1, 2 and 3. We start by starting the initial connection to node2 and node3 from node1 in the erlang shell through net:ping/1. Then store the list of nodes in a variable and pass it to mnesia:create_schema/1. Then, on each node we start the mnesia application through mnesia:start/0.

```
erl1 erl> net:ping('node2@icedcoffee').
pong
erl1 erl> net:ping('node3@icedcoffee').
```

```
pong
erl1 erl> StorageNodes = [node() | nodes()].
[node1@icedcoffee, node2@icedcoffee, node3@icedcoffee]
erl1 erl> mnesia:create_schema(StorageNodes).
ok

erl1 erl> mnesia:start().
ok

erl2 erl> mnesia:start().
ok

erl3 erl> mnesia:start().
ok
```

Once mnesia is running on our grid we can verify that everything is running as expected using mnesia:system_info/1 and mnesia:info/0.

```
erl1 erl> mnesia:system_info(running_db_nodes).
[node1@icedcoffee, node2@icedcoffee, node3@icedcoffee]
erl1 erl> mnesia:info().
...
[{'node1@icedcoffee',disc_copies},
  {'node2@icedcoffee',disc_copies},
  {'node3@icedcoffee',disc_copies}] = [schema]
...
```

At this point the initial fragmented mnesia table needs to be created and verified across these three nodes. We start by using rd/1 to define a record and then mnesia:create_table/2 to create the table. The table in this demonstration is a simple key/value pair dictionary. When we create our table will will create one fragment for each node (3 fragments) and set the table as disc-only.

```
erl1 erl> rd(dictionary, {key, value}).
dictionary
erl1 erl> FragProps = [{node_pool, StorageNodes}, {n_fragments, 3}, {n_disc_only_copies, 1}].
[{node_pool,[node1@icedcoffee, node2@icedcoffee, node3@icedcoffee]}, {n_fragments,3},
{n_disc_only_copies,1}]
erl1 erl> mnesia:create_table(dictionary, [{frag_properties, FragProps}, {attributes, record_info(fields,
```

```
dictionary)}]).
{atomic,ok}
```

We can verify that the table was created using mnesia:table_info/2. Then verify that each node has one fragment of the table. That is done with mnesia:info/0 and mnesia:table_info/2.

```
erl1 erl> mnesia:table_info(dictionary, frag_properties).
[{base_table,dictionary},
 {foreign_key,undefined},
 {hash_module,mnesia_frag_hash},
 {hash_state,{hash_state,3,2,1,phash2}},
 {n_fragments,3},
 {node_pool,[node1@icedcoffee, node2@icedcoffee, node3@icedcoffee]}]
erl1 erl> mnesia:info().
...
[{node1@icedcoffee,disc_only_copies}] = [dictionary_frag3]
[{node2@icedcoffee,disc_only_copies}] = [dictionary_frag2]
[{node3@icedcoffee,disc_only_copies}] = [dictionary]
...
erl1 erl> Info = fun(Item) -> mnesia:table_info(dictionary, Item) end.
#Fun<erl_eval.6.35866844>
erl1 erl> mnesia:activity(sync_dirty, Info, [frag_dist], mnesia_frag).
[{node1@icedcoffee,1},
 {node2@icedcoffee,1},
 {node3@icedcoffee,1}]
```

The next step is to populate the table with data and verify the data distribution across the fragments and nodes.

```
erl1 erl> WriteFun = fun() -> [mnesia:write({dictionary, K, -K}) || K <- lists:seq(1, 30)], ok end.
#Fun<erl_eval.20.117942162>
erl1 erl> mnesia:activity(sync_dirty, WriteFun, mnesia_frag).
ok
erl1 erl> mnesia:activity(sync_dirty, Info, [frag_size], mnesia_frag).
[{dictionary,3},{dictionary_frag2,15},{dictionary_frag3,12}]
```

At this point we have distributed mnesia table over a set of nodes. We've essentially trippled the number of transactions that we can handle compared to running a non distributed-fragmented

mnesia store. But lets consider the situation that comes when we need to grow to another node, either for mnesia load or table size.

The demonstration continues by lighting an additional node.

ic:~/tmp/erl4 $ erl -sname node4 -setcookie 622c84b69ee448a07d80de5cbeb13e3d

What we do now is start mnesia on node4 and then reconfigure node1 to recognize node4.

```
erl4 erl> mnesia:start().
ok

erl1 erl> net:ping('node4@icedcoffee').
pong
erl1 erl> mnesia:change_config(extra_db_nodes, [node4@icedcoffee]).
{ok,['node4@icedcoffee']}
```

On node4 when we run mnesia:info/0 we will now see all of the other nodes on our grid.

```
erl4 erl> mnesia:info().
...
[{'node1@icedcoffee',disc_copies},
  {'node2@icedcoffee',disc_copies},
  {'node3@icedcoffee',disc_copies},
  {'node4@icedcoffee',ram_copies}] = [schema]
...
```

What you'll notice is that the schema on node4 is still a ram copy. This isn't compatible with the disc-only table setup we have on our other nodes. Before we can continue adding the node to the table schema, we need to transform that using mnesia:change_table_copy_type/3.

```
erl4 erl> mnesia:change_table_copy_type(schema, node(), disc_copies).
{atomic,ok}
erl4 erl> mnesia:info().
...
[{'node1@icedcoffee',disc_copies},
  {'node2@icedcoffee',disc_copies},
  {'node3@icedcoffee',disc_copies},
```

```
 {'node4@icedcoffee',disc_copies}] = [schema]
...
```

Now that all of the nodes are in order we can add the node to the dictionary table schema and add another fragment to the table. This is done with mnesia:change_table_frag/2 calling the add_node and add_frag forms of this function.

```
erl1 erl> mnesia:change_table_frag(dictionary, {add_node, 'node4@icedcoffee'}).
{atomic,ok}
erl1 erl> NodeLayout = mnesia:activity(sync_dirty, Info, [frag_dist], mnesia_frag).
[{'node4@icedcoffee',0},
 {'node1@icedcoffee',1},
 {'node2@icedcoffee',1},
 {'node3@icedcoffee',1}]
erl1 erl> mnesia:change_table_frag(dictionary, {add_frag, NodeLayout}).
 {atomic,ok}
```

On the initial call to mnesia:change_table_frag/2 the second parameter is the {add_node, 'node4@icedcoffee'} tuple. The second call creates a list that represents the node layout that is then fed into the third call to mnesia:change_table_frag/2 that sends the {add_frag, NodeLayout} tuple telling mnesia to add another table fragment to the table. With that done we can see that our table has been adjusted and rebalanced accordingly.

```
erl1 erl> mnesia:activity(sync_dirty, Info, [frag_dist], mnesia_frag).
[{'node1@icedcoffee',1},
 {'node2@icedcoffee',1},
 {'node3@icedcoffee',1},
 {'node4@icedcoffee',1}]
erl1 erl> mnesia:activity(sync_dirty, Info, [frag_size], mnesia_frag).
[{dictionary,3},
 {dictionary_frag2,8},
 {dictionary_frag3,12},
 {dictionary_frag4,7}]
```

If we wanted to add a fragment to a specific node we use the mnesia:change_table_frag/2 function but instead of passing a list we pass the node that we want to adjust.

```
erl1 erl> mnesia:change_table_frag(dictionary, {add_frag, ['node4@icedcoffee']}).
```

```
  {atomic,ok}
erl1 erl> mnesia:change_table_frag(dictionary, {add_frag, ['node4@icedcoffee']}).
  {atomic,ok}
erl1 erl> mnesia:change_table_frag(dictionary, {add_frag, ['node4@icedcoffee']}).
  {atomic,ok}
erl1 erl> mnesia:activity(sync_dirty, Info, [frag_dist], mnesia_frag).
[{'node1@icedcoffee',1},
  {'node2@icedcoffee',1},
  {'node3@icedcoffee',1},
  {'node4@icedcoffee',4}]
erl1 erl> mnesia:activity(sync_dirty, Info, [frag_size], mnesia_frag).
[{dictionary,1},
  {dictionary_frag2,6},
  {dictionary_frag3,6},
  {dictionary_frag4,7},
  {dictionary_frag5,2},
  {dictionary_frag6,2},
  {dictionary_frag7,6}]
```

1.101 Can one mnesia table fragment be further fragmente

发表时间: 2009-03-20 关键字: mnesia fragment

Login : Register
Can one mnesia table fragment be further fragmented?
View: New views
5 Messages — Rating Filter: Alert me

Can one mnesia table fragment be further fragmented?
Click to flag this post

by devdoer bird Jun 18, 2008; 12:33am :: Rate this Message: - Use ratings to moderate (?)

Reply | Reply to Author | Print | View Threaded | Show Only this Message
Hi:
I spilit one mnesia table into 3 fragments which distributed on 3 machines named A,B and C,but now the the machine "A" is very busy,
so I decide split the fragment on the machine "A" into small fragments.Does mnesia support this?
Thanks.

_____
erlang-questions mailing list
erlang-questions@...
http://www.erlang.org/mailman/listinfo/erlang-questions

Re: Can one mnesia table fragment be further fragmented?
Click to flag this post

by Scott Lystig Fritchie Jun 19, 2008; 08:21am :: Rate this Message: - Use ratings to moderate (?)

Reply | Reply to Author | Print | View Threaded | Show Only this Message
devdoer bird <devdoer2@...> wrote:

dd> I spilit one mnesia table into 3 fragments which distributed on 3
dd> machines named A,B and C,but now the the machine "A" is very busy,

dd> so I decide split the fragment on the machine "A" into small
dd> fragments.Does mnesia support this?

No, that isn't possible with Mnesia, not as you wish/state above.

If you have a table 'tab' split into fragments 'tab', 'tab_frag2', and
'tab_frag3', you can add more fragments: 'tab_frag4', 'tab_frag5', etc.
You can choose to put only 1 fragment on node A, 4 fragments on node B,
and 4 fragments on node C. Assuming that access to the table is evenly
distributed over all fragments, then node A should be getting only 1/9
of the load, and B & C will get 4/9 of the load.

-Scott

_____
erlang-questions mailing list
erlang-questions@...
http://www.erlang.org/mailman/listinfo/erlang-questions

Re: Can one mnesia table fragment be further fragmented?
Click to flag this post

by devdoer bird Jun 19, 2008; 10:03am :: Rate this Message: - Use ratings to moderate (?)

Reply | Reply to Author | Print | View Threaded | Show Only this Message
Thanks.

Thant means ,I need fragment table many times not for load balance but for decreasing one specific node's (the node is not as powerful as others) load.

Eg. Node A has 2 fragments,Node B has 2 fragments. Each fragment has equal records(say 100 records each fragment).

Now the Node B is heavily loaded.If I adding one or two fragments in Node A , the load of Node B won't be

decreased ,for the records from Node A will be splited and moved to the new fragments,the records

on Node A is 50+50+50+50=200 ,the total number of Node B's record is still 200.

In order to decreas the load on Node B,I have to calcaulate carefully how many new fragments should be added.

If there's a way for mnesia to specify which node's fragment should be splited when addin a new fragment,the problems above will be solved.Take the example above ,when adding a new fragment,I can tell mnesia to split Node B's fragment.

2008/6/19, Scott Lystig Fritchie <fritchie@...>:

> devdoer bird <devdoer2@...> wrote:
>
> dd> I spilit one mnesia table into 3 fragments which distributed on 3
> dd> machines named A,B and C,but now the the machine "A" is very busy,
>
> dd> so I decide split the fragment on the machine "A" into small
> dd> fragments.Does mnesia support this?
>
> No, that isn't possible with Mnesia, not as you wish/state above.
>
> If you have a table 'tab' split into fragments 'tab', 'tab_frag2', and
> 'tab_frag3', you can add more fragments: 'tab_frag4', 'tab_frag5', etc.
> You can choose to put only 1 fragment on node A, 4 fragments on node B,
> and 4 fragments on node C. Assuming that access to the table is evenly
> distributed over all fragments, then node A should be getting only 1/9
> of the load, and B & C will get 4/9 of the load.
>
> -Scott

Re: Can one mnesia table fragment be further fragmented?
Click to flag this post

by Paul Mineiro Jun 19, 2008; 03:14pm :: Rate this Message: - Use ratings to moderate (?)

Reply | Reply to Author | Print | View Threaded | Show Only this Message
Re: picking what gets split, not with the default frag_hash. You can see
what it's doing in mnesia_frag_hash.erl ... it basically splits the
fragments in order. This means that unless there is a power-of-two number
of fragments they will be of different magnitudes. You could leverage
that for your load balancing by placing bigger fragments on less noded
nodes, etc. In fact, maybe this is why you are seeing
an uneven distribution to begin with?

-- p

p.z. If you are feeling your wheaties, you can make your own frag hash,
and specify the module in the frag_properties when you create the table.

http://www.erlang.org/doc/apps/mnesia/Mnesia_chap5.html#5.3

Don't be tempted to just use phash2 or something like that; in order to go
from N fragments to M fragments you have to pass through all the
intermediate stages N + 1, ..., M - 1, M ; you do not want to rehash all
the data in all the buckets at each step.

On Thu, 19 Jun 2008, devdoer bird wrote:

> Thanks.
>
> Thant means ,I need fragment table many times not for load balance but for
> decreasing one specific node's (the node is not as powerful as others)
> load.
>
> Eg. Node A has 2 fragments,Node B has 2 fragments. Each fragment has equal
> records(say 100 records each fragment).
>
> Now the Node B is heavily loaded.If I adding one or two fragments in Node

> A , the load of Node B won't be
>
> decreased ,for the records from Node A will be splited and moved to the new
> fragments,the records on Node A is 50+50+50+50=200 ,the total number of
> Node B's record is still 200.
>
> In order to decreas the load on Node B,I have to calcaulate carefully how
> many new fragments should be added.
>
> If there's a way for mnesia to specify which node's fragment should be
> splited when addin a new fragment,the problems above will be solved.Take the
> example above ,when adding a new fragment,I can tell mnesia to split Node
> B's fragment.
>
> 2008/6/19, Scott Lystig Fritchie <fritchie@...>:
> >
> > devdoer bird <devdoer2@...> wrote:
> >
> > dd> I spilit one mnesia table into 3 fragments which distributed on 3
> > dd> machines named A,B and C,but now the the machine "A" is very busy,
> >
> > dd> so I decide split the fragment on the machine "A" into small
> > dd> fragments.Does mnesia support this?
> >
> > No, that isn't possible with Mnesia, not as you wish/state above.
> >
> > If you have a table 'tab' split into fragments 'tab', 'tab_frag2', and
> > 'tab_frag3', you can add more fragments: 'tab_frag4', 'tab_frag5', etc.
> > You can choose to put only 1 fragment on node A, 4 fragments on node B,
> > and 4 fragments on node C. Assuming that access to the table is evenly
> > distributed over all fragments, then node A should be getting only 1/9
> > of the load, and B & C will get 4/9 of the load.
> >
> > -Scott
> >
>
... [show rest of quote]

In an artificial world, only extremists live naturally.

-- Paul Graham

_____

erlang-questions mailing list
erlang-questions@...
http://www.erlang.org/mailman/listinfo/erlang-questions

Re: Can one mnesia table fragment be further fragmented?
Click to flag this post

by devdoer bird Jun 20, 2008; 01:23pm :: Rate this Message: - Use ratings to moderate (?)

Reply | Reply to Author | Print | View Threaded | Show Only this Message
Thanks ,Paul. I decide to replace the mnesia_frag_hash with my own hash module .With this customized hash module,I hope I can explicitly pass a fragment number to make the fragment splited. I decide to use a mnesia table or other persitent-able data structure in erlang which can globally seen by all nodes to store the mapping relations between key value range and fragment(Eg. [1..999]-> frag 1,[1000-1999]->frag2), so when I need to split a specifed fragment ,I can rejust the mapping relation table.

Here's my plan,any suggestions are welcome.

2008/6/19, Paul Mineiro <paul-trapexit@...>:

> Re: picking what gets split, not with the default frag_hash. You can see
> what it's doing in mnesia_frag_hash.erl ... it basically splits the
> fragments in order. This means that unless there is a power-of-two number
> of fragments they will be of different magnitudes. You could leverage
> that for your load balancing by placing bigger fragments on less noded
> nodes, etc. In fact, maybe this is why you are seeing
> an uneven distribution to begin with?
>
> -- p
>
> p.z. If you are feeling your wheaties, you can make your own frag hash,

and specify the module in the frag_properties when you create the table.

http://www.erlang.org/doc/apps/mnesia/Mnesia_chap5.html#5.3

Don't be tempted to just use phash2 or something like that; in order to go
from N fragments to M fragments you have to pass through all the
intermediate stages N + 1, ..., M - 1, M ; you do not want to rehash all
the data in all the buckets at each step.

On Thu, 19 Jun 2008, devdoer bird wrote:

> Thanks.
>
> Thant means ,I need fragment table many times not for load balance but for
> decreasing one specific node's (the node is not as powerful as others)
> load.
>
> Eg. Node A has 2 fragments,Node B has 2 fragments. Each fragment has equal
> records(say 100 records each fragment).
>
> Now the Node B is heavily loaded.If I adding one or two fragments in Node
> A , the load of Node B won't be
>
> decreased ,for the records from Node A will be splited and moved to the new
> fragments,the records on Node A is 50+50+50+50=200 ,the total number of
> Node B's record is still 200.
>
> In order to decreas the load on Node B,I have to calcaulate carefully how
> many new fragments should be added.
>
> If there's a way for mnesia to specify which node's fragment should be
> splited when addin a new fragment,the problems above will be solved.Take the
> example above ,when adding a new fragment,I can tell mnesia to split Node
> B's fragment.
>
> 2008/6/19, Scott Lystig Fritchie <fritchie@...>:
> >
> > devdoer bird <devdoer2@...> wrote:

> >
> > dd> I spilit one mnesia table into 3 fragments which distributed on 3
> > dd> machines named A,B and C,but now the the machine "A" is very busy,
> >
> > dd> so I decide split the fragment on the machine "A" into small
> > dd> fragments.Does mnesia support this?
> >
> > No, that isn't possible with Mnesia, not as you wish/state above.
> >
> > If you have a table 'tab' split into fragments 'tab', 'tab_frag2', and
> > 'tab_frag3', you can add more fragments: 'tab_frag4', 'tab_frag5', etc.
> > You can choose to put only 1 fragment on node A, 4 fragments on node B,
> > and 4 fragments on node C. Assuming that access to the table is evenly
> > distributed over all fragments, then node A should be getting only 1/9
> > of the load, and B & C will get 4/9 of the load.
> >
> > -Scott
> >
>

In an artificial world, only extremists live naturally.

-- Paul Graham

_____

erlang-questions mailing list
erlang-questions@...
http://www.erlang.org/mailman/listinfo/erlang-questions

_____

erlang-questions mailing list
erlang-questions@...
http://www.erlang.org/mailman/listinfo/erlang-questions

1.102 Mnesia consumption

发表时间: 2009-03-20 关键字: mnesia consumption

原文地址：http://www.erlang.org/~hakan/

知道这个就可以预估mnesia的资源消耗哦

Mnesia introduces no hard system limits on its own. The hard limits can be found in the underlaying system(s), such as the Erlang Run Time System (ERTS) and the operating system.

At startup Mnesia consumes a small amount of fixed resources, regardless of how the system is used. A few number of static processes are spawned and a few number of ets tables are created to host the meta data of the database. If Mnesia is configured to use disk, a disk_log file is opened. In the following we will discuss the dynamic nature of Mnesia, such as:

- o memory consumtion
- o system limits
- o disk consumption

Memory consumption in Mnesia:

Mnesia itself does not consume especially much memory and we do not discuss this issue here. The resources consumed by Mnesia is affected by many factors that depend of the usage of Mnesia:

- o number of tables
- o number of records per table
- o size of each record
- o table type (set or bag)
- o number of indices per table (including snmp "index")
- o size of key and attributes stored in index
- o number of replicas per table
- o replica storage type
- o number of simultaneous transactions
- o duration of transactions

o number of updates per transaction
o size of each updated record
o frequency of dumping of transaction log
o duration of checkpoints

A table in Mnesia may be replicated to several nodes.
The storage type of each replica may vary from node to node.
Currently Mnesia support three storage types:

o ram_copies - resides in ets tables only
o disc_only_copies - resides in dets tables only
o disc_copies - resides in both ets and dets tables

The records that applications store in Mnesia are stored in ets and dets respectively. Mnesia does not add any extra memory consumption of its own. The record is directly stored in ets and/or dets. ets tables resides in primary memory. dets tables resides on disk. Read about their memory consumption elsewhere (stdlib). When records are read from ets and dets tables a deep copy of the entire records is performed. In dets the copy is performed twice, first the records is read from the file and allocated on the heap of the dets process, then the record is sent to the process that ordered the lookup (your application process) and allocated on its heap.

A table may have indices. Each index is represented as a table itself. In an index table, tuples of {Attribute, PrimaryKey} are stored. Indices for disc_only_copies are stored in dets only. Indices for ram_copies and disc_copies are stored in ets only. Indecies are currently implemented as bags, which means that the performance may degenerate if the indexed attribute has the same value in lots of records in the table. bags with many records with the same key should be avoided.

A table may be accessible via SNMP. If that feature is activated, a special kind of index is attached to the table. In the "index table", tuples of {SnmpKey, PrimaryKey} is stored. The SnmpKey is represented as a flat string according to the SNMP standard. Read about SNMP elsewhere (snmpea). SNMP indices are stored in bplus_tree only. Read

about memory consumption in bplus_tree elsewhere (stdlib).

Tables may be updated in three contexts:

- o transaction
- o dirty (sync or async)
- o ets

Updates performed in transactions are recorded in an ets table local to the transaction. This means that updated records are stored both in the local transaction storage and the main table, where the committed records resides. (They is also stored on the heap until the garbage collector decides to remove them). Info about other updating operations are also stored in the local storage (i.e. write, delete, delete_object). The local storage is erased at the end of the outermost transaction. If the duration of the outermost transaction is long, e.g. because of lock conflicts, or the total size of the updated records is large the memory consumption may become an issue. Especially if the table is replicated, since the contents of the local storage is processed (and temporary allocated on the heap) and parts of it is sent to the other involved nodes as a part of the commit protocol.

Both read and update accesses in transactions requires locks to be acquired. Information about the locks are both stored in the local transaction storage and in private storage administered by the lock manager. This information is erased at the end of the outermost transaction. The memory consumption for the lock info is affected by the number of records accessed in a transaction and the duration of the outer transaction. If Mnesia detects a potential deadlock, it selects a deadlock victim and restarts its outer transaction. Its lock info and local storage is then erased.

Checkpoints in Mnesia is a feature that retains the old state of one or more tables even if the tables are updated. If a table that is a part of a checkpoint is updated, the old record value (may be several records if the table is a bag) will be stored in a checkpoint retainer for the duration of the checkpoint. If the same record is updated

several times, only oldest record version is kept in the retainer. For disc_only_copies replicas the checkpoint retainer is a dets table. For ram_copies and disc_copies the retainer is an ets table. Read about the memory consumption about ets and dets elsewhere (stdlib).

Dirty updates are not performed in a process local storage, the updates are performed directly in the main table containing the committed records. Dirty accesses does not bother about transactions or lock management.

Raw ets updates (performed inside mnesia:ets/1) assumes that the table is an ets table and that it is local. The raw ets updates are performed directly in the table containing the committed records. The raw ets updates does not update indices, checkpoint retainers, subscriptions etc. like the dirty accesses and transaction protected accesses do.

System limits:

There are two system limits that may be worthwhile to think about:

- o number of ets tables
- o number of open file descriptors (ports)

There is an upper system limit of the maximum number of ets tables that may co-exist in an Erlang node. That limit may be reached if the number of tables is large, especially if they have several indices or if there are several checkpoints attached to them. The limit may also be reached if the number of simultaneous transactions becomes large. Read about the actual limit elsewhere (stdlib).

There is an upper limit of the maximum number of open file descriptors (ports) in an Erlang node. That limit may be reached if the number of disk resident Mnesia tables is large. Each replica whose storage type is disc_only_copies occupies constantly a file descriptor (held by the dets server). Updates that involves Mnesia tables residing on disk (disc_copies and disc_only_copies) are written into the transaction log. The records in the transaction log is propagated (dumped) to the

corresponding dets files at regular intervals. The dump log interval is configurable. During a dump of the transaction log the dets files corresponding to the disc_copies tables is opened.

We strongly recommend the Mnesia directory to reside on a local disk. There are several reasons for this. Firstly the access time of files on a local disk, normally is faster than access of remotely mounted disks. The access time is independent of network traffic an load on other computers. Secondly the the whole emulator may be blocked while I/O is performed, if the remote file system daemons does not respond for some reason the emulator may be blocked quite a while.

Disk consumption:

Updates that involves Mnesia tables residing on disk (disc_copies and disc_only_copies) are written into the transaction log. In the log, the same record may occur any number of times. Each update causes the new records to be appended to the log, regardless if they are already stored in the log or not. If the same disk resident record is updated over and over again it becomes really important to configure Mnesia to dump the transaction log rather frequently, in order to reduce the disk consumption.

The transaction log is implemented with disk_log (stdlib). Appending a commit record to disk_log is faster than reading the same commit record and update the corresponding dets files. This means that the transaction log may grow faster than the dumper is able to consume the log. When that happens, a special overload event is sent to Mnesia's event handler. If you suspect that this may become an issue for you, (i.e. the application performs lots of updates in disk resident tables) you should subscribe to Mnesia's system events, and reduce the load on Mnesia when the events occurs. When it is time to dump the log (i.e. any of the dump log thresholds has been reached) the log file is renamed and a new empty one is created. This means that the disk space occupied by the transaction log may be twofold at the end of the dump if the applications are very update intense.

Normally Mnesia updates the dets files in place during the dump log

activity. But there is possible to configure Mnesia first make a copy of the entire (possible large) dets file and apply the log to the copy. When the dump is done the old file is removed and the new one is renamed. This feature may cause the disk space occupied by dets tables to occationally be doubled.

If Mnesia encounters a fatal error Mnesia is shutdown and a core dump is generated. The core dump consists of all log files, the schema, lock info etc. so it may be quite large. By default the core dump is written to file on current directory. It may be a good idea to look for such files, now and then and get rid of them or install an event handler which takes care of the core dump binary.

Hakan Mattsson <hakan@erix.ericsson.se>

1.103 Mnesia upgrade policy (PA3)

发表时间: 2009-03-20 关键字: mnesia upgrade policy

原文地址： http://www.erlang.org/~hakan/

Mnesia upgrade policy (PA3)
==========

This paper describes the upgrade policy of the Mnesia application.
It is divided into the following chapters:

o Architecture overview
o Compatibility
o Configuration management
o Compatibility requirements on other applications
o Upgrade scenarios
o Remaining issues

Architecture overview

Mnesia is a distributed DataBase Management System (DBMS), appropriate
for telecommunications applications and other Erlang applications
which require continuous operation and exhibit soft real-time
properties. Mnesia is entirely implemented in in Erlang.

Meta data about the persistent tables is stored in a public ets table,
called mnesia_gvar. This ets table is accessed concurrently by several
kinds of processes:

* Static - on each node Mnesia has about 10 static
  processes. Eg. monitor, controller, tm, locker, recover, dumper...

* Dynamic - there are several kinds of dynamic processes are created
  for various purposes. Eg. tab_copier, perform_dump, backup_master ...

* Client processes created by Mnesia users. These invokes the Mnesia

API to perform operations such as table lookups, table updates, table reconfigurations...

All these kinds of processes communicates with each other both locally (on the same node) and remotely.

Mnesia may either be configured to use local disc or be to totally disc less. All disc resident data is located under one directory and is hosted by disk_log and dets.

The persistent tables may be backed up to external media. By default backups are hosted by disk_log.

Compatibility

Each release of Mnesia has a unique version identifier. If Mnesia is delivered to somebody outside the development team, the version identifier is always changed, even if only one file differs from the previous release of Mnesia. This means that the exact version of each file in Mnesia can be determined from the version identifier of the Mnesia application.

Mnesia does NOT utilize the concept of OTP "patches". The smallest deliverable is currently the entire application. In future releases we will probably deliver binaries, source code and different kinds of documentation as separate packages.

Changes of the version identifier follows a certain pattern, depending of how compatible the new release is with the previous dito. The version identifier consists of 3 integer parts: Major.Minor.Harmless (or just Major.Minor when Harmless is 0).

If a harmless detail has been changed, the "Harmless" part is incremented. The change must not imply any incompatibility problems for the user. An application upgrade script (.appup) could be delivered in order to utilize code change in a running system, if required.

If the change is more than a harmless detail (or if it is a harmless change but has implied a substantial rewrite of the code), the "Minor" part is incremented. The change must not imply any incompatibility problems for the user. An application upgrade script (.appup) for code change in a running system, is not always possible to deliver if the change is complex. A full restart of Mnesia (and all applications using Mnesia) may be required, potentially on all nodes simultaneously.

All other kinds of changes implies the "Major" part to be incremented. The change may imply that users of Mnesia needs to rewrite their code, restart from backup or other inconveniences. Application upgrade script (.appup) for code change in a running system is probably too complex to write.

In any case it is a good idea to always read the release notes.

Configuration management

Each major release constitutes an own development branch. Bugfixes, new functionality etc. are normally performed in the latest major development branch only. The last release of Mnesia is always the best and customers are encouraged to upgrade to it.

Application upgrade scripts (.appup) are only written if they are explicitly requested by a customer. Preparing the code for a smoothless code change in a running system is very demanding and restrains productive development. The code that handles the upgrade is extremely hard to test, but it must be 100% correct. If it is not 100% correct it is totally useless since the error may easily escalate to a node restart or an inconsistent database.

It may however not always be possible to enforce a customer to accept the latest release of incompatibility reasons. If a customer already has a running system and encounters a serious bug in Mnesia, we may be enforced to fix the bug in an old release. Such a bugfix is performed

in a separate bugfix branch (dedicated for this particular bugfix). All source files in the release is labled with the version identifier (e.g. "mnesia_3.4.1"). An application upgrade script (.appup) is probably needed.

Compatibility requirements on other applications

Mnesia is entirely implemented in Erlang and depends heavily on the Erts, Kernel and StdLib applications.

Changes of storage format in dets and disk_log may cause these files to be automatically upgraded to the new format, but only if Mnesia allows them to perform the "repair". As an effect of such a format upgrade it is likely that the Mnesia files cannot be read by older releases, but it may be acceptable that old releases not are forward compatible.

Mnesia stores its data as binaries. Changes of the external binary format must also be backward compatible. Automatic conversion to the new format is required.

Note, that Mnesia backups may be archived in years and that this requires Erts and Kernel to be backward compatible with very old binary formats and disk_log formats.

Upgrade scenarios

There are a wide range of upgrade scenarios that needs to be analyzed before they occur in reality. Here follows a few of them:

Backup format change.

In the abstract header of the backup there is a version tag that identifies the format of the rest of the backup. Backups may be archived for a long period of time and it is important that

old backups can be loaded into newer versions of the system.
The backup format is Mnesia's survival format.

Changes in the abstract backup format requires the backup version tag to be incremented and code to be written to convert the old format at backup load time. Minor change.

The concrete format is an open format handled by a callback module and it is up to the callback module implementors to handle future changes. The default callback module uses disk_log and Mnesia relies heavily on disk_log's ability to automatically convert old disk_log files into new dito if the concrete format has been changed.

Transaction log file format change.

In the abstract header of the transaction log there is a version tag that identifies the format of the rest of the log. Changes in the abstract transaction log format requires the transaction log version tag to be incremented and code to be written to convert the old format when the log is dumped at startup. Minor change.

The concrete format is hidden by disk_log and Mnesia relies on disk_log's ability to automatically convert old disk_log files into new dito if the concrete format has been changed.

If the abstract format change is severe or if disk_log cannot handle old disk_log formats the entire Mnesia database has to be backed up to external media and then installed as fallback.
It may in worst case be neccessary to implement a new backup callback module that does not make use of disk_log. Major change.

Decision table log file format change.

In the abstract header of the decision table log file there is a version tag that identifies the format of the rest of the log. The concrete format is hidden by disk_log and severe changes in the abstract or concrete formats are handled in the same way as complex changes of the transaction log file format: back the

database up and re-install it as a fallback. Major change.

Minor changes in the abstract format can be handled by conversion code run at startup. Minor change.

.DAT file format change.

.DAT files has no abstract version format identifier.

The concrete format is hidden by dets and Mnesia relies on dets' ability to automatically convert old dets files into new dito if the concrete format has been changed.

Changes in the abstract or incompatible changes in the concrete format are are handled in the same way as complex changes in the transaction log file format: back the database up and re-install it as a fallback.

Core file format change.

The core file is a debugging aid and contains essential information about the database. The core is automatically produced when Mnesia encounters a fatal error or manually by the user for the purpose of enclosing it into a trouble report. The core file consists of list of tagged tuples stored as a large binary. Future changes of the core format can easily be handled by adding a version tag first in the list if neccessary.

Persistent schema format change.

The schema is implemented as an ordinary table with table names as key and a list of table properties as single attribute.
Each table property is represented as a tagged tuple and adding new properties will not break any code. Minor change.

Incompatible changes of the schema representation can be handled in the same way as complex changes of the transaction log file format: back the database up and re-install it as a fallback.

Major change.

If the change is severe and impacts the backup format then it should not be performed at all.

Renaming schema table to mnesia_schema.

All internal (d)ets tables are prefixed with 'mnesia_' in order to avoid name conflicts with other applications. The only exception is the table named 'schema'.

Renaming 'schema' to 'mnesia_schema' is a major change, that may breaks much customer code if it is not done very careful, since the name of the table is a part of the Mnesia API.
The least intrusive change would be to leave the name 'schema' as a part of the API, but internally use the name 'mnesia_schema' and map all usages of 'schema' in all internal modules that accesses the schema table. It is however questionable if the change is worth the work effort.

Transient schema format change

The transient schema information is stored in a public ets table named mnesia_gvar. Changes in the transient schema format affects a lot of processes and it is not feasible to change it dynamically. A restart on all db_nodes is required. Minor change.

Configuration parameter change. *** partly not supported ***

Many of the configuration parameters ought to be possible to be changed dynamically in a running system:

access_module
backup_module
debug
dump_log_load_regulation
dump_log_time_threshold
dump_log_update_in_place

dump_log_write_threshold
event_module
extra_db_nodes

The remaining configuration parameters are only interesting at startup and any dynamic change of these should silently be ignored:

auto_repair
dir
embedded_mnemosyne
ignore_fallback_at_startup
max_wait_for_decision
schema_location

Inter node protocol change. *** partly not supported ***

When Mnesia on one node tries to connect to Mnesia on another node it negotiates with the other node about which inter node communication protocol to use. A list of all known protocols identifiers are sent to the other node which replies with the protocol that it prefers. The other node may also reject the connection. Always when the inter node protocol needs to be changed, the protocol identifier is changed.

If the change is a compatible change and we need to upgrade the protocol in a running system things gets a little bit complicated.
A new version of the software which understands both the old an new protocols must be loaded on all nodes as a first step. All processes that uses old code must somehow switch to use the new code. One severe problem here is to locate all processes that needs a code switch. Server processes are fairly straight forward to manage.
Client processes that are waiting in a receive statement are far more difficult to first locate and then force a code switch.
We may prepare for the code switch by letting the client be able to receive a special code switch message and then continue to wait for interesting messages. (gen_server does not handle this but since Mnesia does not use gen_server's for performance critical processes or crash sensitive processes Mnesia can handle this in

many cases.)

If the new protocol is a pure extension of the old dito we may
go ahead and use the new protocol and bump up the protocol
version identifier.

More complex protocol changes are sometimes possible to handle,
but since they are extreamly hard to test it is probably better
to restart the system.

If the change is an incompatible change, Mnesia must be restarted on
all nodes.

Intra node protocol change.

Changing protocol between processes on the same node is slightly
easier than changing inter node protocols. The difference is that
we may iterate over all nodes and restart them one and one until
the software has started to use the new protocols on all nodes.

Adding a new static process.

Adding a new static process in Mnesia is a minor change,
but cannot be handled by the supervisor "application"
if the shutdown order of the static processes are important.
The most relieable approach here is to restart Mnesia.

Adopt to new functionality in Erts, Kernel or StdLib:

When was the new functionality introduced?
Can Mnesia use the new functionality and still
run on older Erlang/OTP releases?

Changing a function in the Mnesia API.

Changes of module name, function name, arguments or
semantics of a function is likely to be a major change.

Adding a brand new function is a minor change.

Changing a Mnesia internal function.

If it is possible to locate all processes that may invoke the function it may be worth to handle the code change in a running system. The safe approach is to restart the node. The processes must be able to handle 'sys' messages and export 'sys' callback functions. Minor change.

Process state format change.

If it is possible to locate all processes that may invoke the function it may be worth to handle the code change in a running system. The safe approach is to restart the node. The processes must be able to handle 'sys' messages and export 'sys' callback functions. Minor change.

Code change to fix a harmless bug.

Does these exist in reality or only in the Erlang book?

Code change to fix an inconsistency bug.

A bug like all others but the effect of it is an inconsistent database. Mnesia cannot repair the database and it is up to the Mnesia users to make it consistent again. There are three options here:

- ignore the fact that the database is inconsistent
- back up the database and make the backup consistent before installing it as a fallback.
- fix the inconsistency on-line while all applications are accessing the database

Code change to prevent an indefinite wait (deadlock, protocol mismatch).

A bug like all others but the effect of it is hanging processes. Restart Mnesia on one or more nodes.

Remaining issues

The following issues remains to be implemented:

- Dynamic configuration parameter change
- Prepare code switch of client code

1.104 What is the storage capacity of a Mnesia database?

发表时间: 2009-03-20 关键字: mnesia capacity

原文地址： http://stackoverflow.com/questions/421501/what-is-the-storage-capacity-of-a-mnesia-database

Quite large if your question is "what's the storage capacity of an mnesia database made up of a huge number of disc_only_copies tables" - you're largely limited by available disk space.

An easier question to answer is what's the maximum capacity of a single mnesia table of different types. ram_copies tables are limited by available memory. disc_copies tables are limited by their dets backend (Hakan Mattsson on Mnesia) - this limit is 4Gb of data at the moment.

So the simple answer is that simple disc_copies table can store up to 4Gb of data before they run into problems. (*Mnesia doesn't actually crash if you exceed the on-disk size limit - the ram_copies portion of the table continues running, so you can repair this by deleting data or making other arrangements at runtime*)

However if you consider other mnesia features, then the answer is more complicated.

* local_content tables. If the table is a local_content table, then it can have different contents on each node in the mnesia cluster, so the capacity of the table is 4Gb * <number of nodes>
* fragmented tables. Mnesia supports user configurable table partitioning or sharding using table fragments. In this case you can effectively distribute and redistribute the data in your table over a number of primitive tables. These primitive tables can each have their own configuration - say one ram_copies table and the rest disc_only_copies tables. These primitive tables have the same size limits as mentioned earlier and now the effective capacity of the fragmented table is 4Gb * <number of fragments>. (Sadly if you fragment your table, you then have to modify your table access code to use mnesia:activity/4 instead of mnesia:write and friends, but if you plan this in advance it's managable)
* external copies If you like living on the extreme bleeding edge, you could apply the mnesiaex patches to mnesia and store your table data in an external system such as Amazon S3 or Tokyo Cabinet. In this case the capacity of the table is limited by the backend storage.

1.105 How to Fix Erlang Crashes When Using Mnesia

发表时间: 2009-03-22 关键字: out of memory crashes mnesia

原文地址： http://streamhacker.wordpress.com/2008/12/20/how-to-fix-erlang-out-of-memory-crashes-when-using-mnesia/
December 20, 2008 at 9:53 am (erlang) (database, dets, ets, iteration, memory, mnesia, records, transactions)
If you’re getting erlang out of memory crashes when using mnesia, chances are you’re doing it wrong, for various values of it. These out of memory crashes look something like this:

Crash dump was written to: erl_crash.dump
eheap_alloc: Cannot allocate 999999999 bytes of memory (of type "heap")
Possible Causes

You’re doing it wrong
Someone else is doing it wrong
You don’t have enough RAM
While it’s possible that the crash is due to not having enough RAM, or that some other program or process is using too much RAM for itself, chances are it’s your fault.

One of the reasons these crashes can catch you by surprise is that the erlang VM is using a lot more memory than you might think. Erlang is a functional language with single assignment and no shared memory. A major consequence is that when you change a variable or send a message to another process, a new copy of the variable is created. So an operation as simple as dict:update_counter(”foo”, 1, Dict1) consumes twice the memory of Dict1 since Dict1 is copied to create the return value. And anything you do with ets, dets, or mnesia will result in at least 2 copies of every term: 1 copy for your process, and 1 copy for each table. This is because mnesia uses ets and/or dets for storage, which both use 1 process per table. That means every table operation results in a message pass, sending your term to the table or vice-versa. So that’s why erlang may be running out of memory. Here’s how to fix it.

Use Dirty Operations

If you’re doing anything in a transaction, try to figure out how to do it dirty, or at least move as many operations as possible out of the transaction. Mnesia transactions are separate processes with their own temporary ets tables. That means there’s the original term(s) that must be passed in to the transaction or read from other tables, any updated copies that your code creates, copies of terms that

are written to the temporary ets table, the final copies of terms that are written to actual table(s) at the end of the transaction, and copies of any terms that are returned from the transaction process. Here’ s an example to illustrate:

```
example() ->
   T = function() ->
      Var1 = mnesia:read(example_table, "foo"),
      Var2 = update(Var2), % a user-defined function to update Var1
      ok = mnesia:write(Var2),
      Var2
   end,
   {atomic, Var2} = mnesia:transaction(T),
   Var2.
```

First off, we already have a copy of Var1 in example_table. It gets sent to the transaction process when you do mnesia:read, creating a second copy. Var1 is then updated, resulting in Var2, which I’ ll assume has the same memory footprint of Var1. So now we have 3 copies. Var2 is then written to a temporary ets table because mnesia:write is called within a transaction, creating a fourth copy. The transaction ends, sending Var2 back to the original process, and also overwriting Var1 with Var2 in example_table. That’ s 2 more copies, resulting in a total of 6 copies. Let’ s compare that to a dirty operation.

```
example() ->
   Var1 = mnesia:dirty_read(example_table, "foo"),
   Var2 = update(Var1),
   ok = mnesia:dirty_write(Var2),
   Var2.
```

Doing it dirty results in only 4 copies: the original Var1 in example_table, the copy sent to your process, the updated Var2, and the copy sent to mnesia to be written. Dirty operations like this will generally have 2/3 the memory footprint of operations done in a transaction.

Reduce Record Size

Figuring out how to reduce your record size by using different data structures can create huge gains by drastically reducing the memory footprint of each operation, and possibly removing the need to use transaction. For example, let’ s say you’ re storing a large record in mnesia, and using transactions to update it. If the size of the record grows by 1 byte, then each transactional operation like the above will require an additional 5 bytes of memory, or dirty operations will require an additional 3 bytes. For multi-megabyte records, this adds up very quickly. The solution is to figure

how to break that record up into many small records. Mnesia can use any term as a key, so for example, if you’re storing a record with a dict in mnesia such as {dict_record, “foo”, Dict}, you can split that up into many records like [{tuple_record, {"foo", Key1}, Val1}]. Each of these small records can be accessed independently, which could eliminate the need to use transactions, or at least drastically reduce the memory footprint of each transaction.

Iterate in Batches

Instead of getting a whole bunch of records from mnesia all at once, using mnesia:dirty_match_object or mnesia:dirty_select, iterate over the records in batches. This is analagous to using lists operations on mnesia tables. The match_object methods may return a huge number of records, and all those records have to be sent from the table process to your process, doubling the amount of memory required. By iteratively doing operations on batches of records, you’re only accessing a portion at a time, reducing the amount of memory being used at once. Here’s some code examples that only access 1 record at a time. Note that if the table changes during iteration, the behavior is undefined. You could also use the select operations to process records in batches of NObjects at a time.

Dirty Mnesia Foldl

```
dirty_foldl(F, Acc0, Table) ->
    dirty_foldl(F, Acc0, Table, mnesia:dirty_first(Table)).

dirty_foldl(_, Acc, _, '$end_of_table') ->
    Acc;
dirty_foldl(F, Acc, Table, Key) ->
    Acc2 = lists:foldl(F, Acc, mnesia:dirty_read(Table, Key)),
    dirty_foldl(F, Acc2, Table, mnesia:dirty_next(Table, Key)).
```

Dirty Mnesia Foreach

```
dirty_foreach(F, Table) ->
    dirty_foreach(F, Table, mnesia:dirty_first(Table)).

dirty_foreach(_, _, '$end_of_table') ->
    ok;
dirty_foreach(F, Table, Key) ->
    lists:foreach(F, mnesia:dirty_read(Table, Key)),
    dirty_foreach(F, Table, mnesia:dirty_next(Table, Key)).
```

Conclusion

It’ s probably your fault
Do as little as possible inside transactions
Use dirty operations instead of transactions
Reduce record size
Iterate in small batches

Ulf said,
January 7, 2009 at 1:44 pm
Recommending dirty operations over transactions should come with a very big caveat: you change the semantics and forego safety, esp. if you have a replicated system. Dirty operations do not guarantee that replication works, for example. It may even work partially, given certain error situations, causing database inconsistency.

I normally advice people to use mnesia:activity(Type, F) rather than mnesia:transaction(F), and to always start with real transactions, then measure and - only if really necessary (and safe!), switch to dirty where neeeded. This can then be done by just changing Type from ‘transaction’ to ‘async_dirty’ .

In my experience, the “iterate in small batches” should be one of the first points. It is very good advice. Also, monitor ets and mnesia tables to see if they keep growing. Inserting temporary objects and forgetting to delete them is a fairly common source of memory growth.

In other cases, a form of load control may well be what’ s needed, making sure that the system doesn’ t take on more work than it can handle (easy to do in an asynchronous environment). One very simple such device would be a gen_server that workers ask (synchronously) for permission before starting a new task. The server can monitor the ‘run_queue’ to guard against cpu overload, memory usage, number of running processes, etc., depending on where your bottlenecks are. Keep it very simple.

1.106 On bulk loading data into Mnesia

发表时间: 2009-03-22 关键字: bulk loading data mnesia

原文地址： http://www.metabrew.com/article/on-bulk-loading-data-into-mnesia/

On bulk loading data into Mnesia

Consider this a work-in-progress; I will update this post if I find a ‘better’ way to do fast bulk loading

The time has come to replace my ets-based storage backend with something non-volatile. I considered a dets/ets hybrid, but I really need this to be replicated to at least a second node for HA / failover. Mnesia beckoned.

The problem:

15 million [fairly simple] records
1 Mnesia table: bag, disc_copies, just 1 node, 1 additional index
Hardware is a quad-core 2GHz CPU, 16GB Ram, 8x 74Gig 15k rpm scsi disks in RAID-6
Takes ages* to load and spews a load of “Mnesia is overloaded” warnings
* My definition of ‘takes ages’: Much longer than PostgreSQL \copy or MySQL LOAD DATA INFILE

At this point all I want is a quick way to bulk-load some data into a disc_copies table on a single node, so I can get on with running some tests.

Here is the table creation code:

```
mnesia:create_table(subscription,
[
{disc_copies, [node()]},
{attributes, record_info(fields, subscription)},
{index, [subscribee]}, %index subscribee too
{type, bag}
]
)
```

The subscription record is fairly simple:

```
{subscription, subscriber={resource, user, 123}, subscribee={resource, artist, 456}}
```

I’ m starting erlang like so:
erl +A 128 -mnesia dir '"/home/erlang/mnesia_dir"' -boot start_sasl

The interesting thing there is really the +A 128 - this spreads the cpu load better between the 4 cores.

Attempt 0) ‘by the book’ one transaction to rule them all

Something like this:
mnesia:transaction(fun()-> [mnesia:write(S) || S <- Subs] end)

Time taken: Too long, I gave up after 12 hours
Number of “Mnesia overloaded” warnings: lots
Conclusion: Must be a better way
TODO: actually run this test and time it.

Attempt 1) dirty_write

There isn’ t really any need to do this in a transaction, so I tried dirty_write.
[mnesia:dirty_write(S) || S <- Subs]

And here’ s the warning in full:
=ERROR REPORT==== 13-Oct-2008::16:53:57 ===
Mnesia('mynode@myhost'): ** WARNING ** Mnesia is overloaded: {dump_log,
write_threshold}

Time taken: 890 secs
Number of “Mnesia overloaded” warnings: lots
Conclusion: Workable, but nothing to boast about. Those warnings are annoying

Attempt 2) dirty_write, defer index creation

A common trick with traditional RDBMS would be to bulk load the data into the table and add the indexes afterwards. In some scenarios you can avoid costly incremental index update operations. If you are doing this in one gigantic transaction it shouldn’ t matter, and I’ m not really sure how mnesia works under the hood (something I plan to rectify if I end up using it for real).
I tried a similar approach by commenting out the {index, [subscribee]} line above, doing the load, then using mnesia:add_table_index(subscriber, subscribee) afterwards to add the index once all the data was loaded. Note that mnesia was still building the primary index on the fly, but that can’ t be

helped.
Time taken: 883 secs (679s load + 204s index creation)
Number of “Mnesia overloaded” warnings: lots
Conclusion: Insignificant, meh

Attempt 3) mnesia:ets() trickery

This is slightly perverted, but I tried it because I was suspicious that incrementally updating the on-disk data wasn’t especially optimal. The idea is to make a ram_only table and use the mnesia:ets() function to write directly to the ets table (doesn’t get much faster than ets). The table can then be converted to disc_copies. There are caveats - to quote The Fine Manual:

Call the Fun in a raw context which is not protected by a transaction. The Mnesia function call is performed in the Fun are performed directly on the local ets tables on the assumption that the local storage type is ram_copies and the tables are not replicated to other nodes. Subscriptions are not triggered and checkpoints are not updated, but it is extremely fast.

I can live with that. I don’t mind if replication takes a while to setup when I put this into production - I’ll gladly take any optimisations I can get at this stage (testing/development).

Loading a list of subscriptions looks like this:
mnesia:ets(fun()-> [mnesia:dirty_write(S) || S <- Subs] end).
And to convert this into disc_copies once data is loaded in:
mnesia:change_table_copy_type(subscription, node(), disc_copies).

Time taken: 745 secs (699s load + 46s convert to disc_copies)
Number of “Mnesia overloaded” warnings: none!
Conclusion: Fastest yet, bit hacky

Summary

At least the ets() trick doesn’t spew a million warnings. I also need to examine the output of mnesia:dump_to_textfile and see if loading data from that format is any faster.

TODO:

Examine / test using the dum_to_textfile method
Run full transactional load and time it

Try similar thing with PostgreSQL

1.107 scaling mnesia with local_content

发表时间: 2009-03-22 关键字: scaling mnesia local_content

原文地址： http://dukesoferl.blogspot.com/2008/08/scaling-mnesia-with-localcontent.html
so we've been spending the last two weeks trying to scale our mnesia application for a whole bunch of new traffic that's about to show up. previously we had to run alot of ec2 instances since we were using ets (RAM-based) storage; once we solved that we started to wonder how many machines we could turn off.

initial indications were disappointing in terms of capacity. none of cpu, network, memory, or disk i/o seemed particularly taxed. for instance, even though raw sequential performance on an ec2 instance can hit 100mb/s, we were unable to hit above 1mb/s of disk utilization. one clue: we did get about 6mb/s of disk utilization when we started a node from scratch. in that case, 100 parallel mnesia_loader processes grab table copies from remote nodes. thus even under an unfavorable access pattern (writing 100 different tables at once), the machines were capable of more.

one problem we suspected was the registered process mnesia_tm, since all table updates go through this process. the mnesia docs do say that it is "primarily intended to be a memory-resident database". so one thought was that mnesia_tm was hanging out waiting for disk i/o to finish and this was introducing latency and lowering throughput; with ets tables, updates are cpu bound so this design would not be so problematic. (we already have tcerl using the async_thread_pool, but that just means the emulator can do something else, not the mnesia_tm process in particular). thus, we added an option to tcerl to not wait for the return value of the linked-in driver before returning from an operation (and therefore not to check for errors). that didn't have much impact on i/o utilization.

we'd long ago purged transactions from the serving path, but we use sync_dirty alot. we thought maybe mnesia_tm was hanging out waiting for acknowledgements from remote mnesia_tm processes and this was introducing latency and lowering throughput. so we tried async_dirty. well that helped, except that under load the message queue length for the mnesia_tm process began to grow and eventually we would have run out of memory.

then we discovered local_content, which causes a table to have the same definition everywhere, but different content. as a side effect, replication is short-circuited. so with very minor code changes we tested this out and saw a significant performance improvement. of course, we couldn't do this to every table we have; only for data for which we were ok with losing if the node went down. however it's neat because now there are several types of data that can be managed within mnesia, in order of expense:

transactional data. distributed transactions are extremely expensive, but sometimes necessary.
highly available data. when dirty operations are ok, but multiple copies of the data have to be kept, because the data should persist across node failures.
useful data. dirty operations are ok, and it's ok to lose some of the data if a node fails.
the erlang efficiency guide says "For non-persistent database storage, prefer Ets tables over Mnesia local_content tables.", i.e., bypass mnesia for fastest results. so we might do that, but right now it's convenient to have these tables acting like all our other tables.

interestingly, i/o utilization didn't go up that much even though overall capacity improved alot. we're writing about 1.5 mb/s now to local disks. instead we appear cpu bound now; we don't know why yet.

1.108 Making a Mnesia Table SNMP Accessible

发表时间: 2009-03-23 关键字: mnesia table snmp

原文地址： http://www.drxyzzy.org/mnesia-snmp/index.html
手册写的巨明白

Contents

Introduction
Define a Mnesia Table
Create the MIB File
Create the .funcs File
Compile the MIB for OTP
Create sys.config and .conf Files
Create MIB Header for Mnesia
Create Data Directory for Mnesia
Create Erlang Program
Start OTP with Mnesia and SNMP agent
Access the Table from net-snmp
References

Introduction

We occasionally make mnesia data readable via SNMP. Here's a cookbook summary of the process. Certainly the Ericsson guide listed in the references should be consulted - this page merely gives extensive detail for a sample case.

Complete directions are given for creating a sample table, configuring the MIB, relating the two, and accessing the table from a net-snmp client.

Limitations of this tutorial.

SNMP versions 1&2, but not 3, are used.
SMIv1 is used to write the MIB.
The SNMP agent listens on non-privileged port 9161, instead of 161.
Code was compiled and executed using FreeBSD-5.0, OTP-R9B1, and Net-SNMP-5.0.8_1.

Define a Mnesia Table

For this example, use a table for several servers which handle telephone calls. Each row contains the server name, plus two counters - the number of inbound calls and the number of outbound calls. Here's an .hrl file defining the record type:

serverTable.hrl.

Create the MIB File

This example places the monitored data in the SNMP OID tree under iso.org.dod.internet.private.enterprises. Filename must end with ".mib". We use a fictional enterprise name of "bazmatic" with enterprise number 99999. See references below if you need an official enterprise number. Server names will be read-only from SNMP. Call counters will be writable from SNMP.

SAMPLE-MIB.mib.

Create the .funcs File

Since default mnesia functions are used, this step could probably be omitted if RowStatus were used.

SAMPLE-MIB.funcs.

Compile the MIB for OTP

Shell command for compiling the MIB.

```
/usr/home/xxx/otp-snmp>erl
Erlang (BEAM) emulator version 5.2.3.3 [source] [hipe] [threads:0]

Eshell V5.2.3.3  (abort with ^G)
1> snmp:c("SAMPLE-MIB", [{db, mnesia}]).
{ok,"./SAMPLE-MIB.bin"}
```

Result is binary file SAMPLE-MIB.bin.

Create sys.config and .conf Files

Continuing the previous erl session:

```
2> snmp:config().

Simple SNMP configuration tool (v3.0)
----------------------------------------------
Note: Non-trivial configurations still has to be done manually.
IP addresses may be entered as dront.ericsson.se (UNIX only) or 123.12.13.23

1. System name (sysName standard variable) [xxx's agent]sample agent
2. Engine ID (snmpEngineID standard variable)[xxx's engine]sample engine
3. The UDP port the agent listens to. (standard 161) [4000]9161
4. IP address for the agent (only used as id
   when sending traps) []127.0.0.1
5. IP address for the manager (only this manager will have access
   to the agent, traps are sent to this one) []127.0.0.1
6. To what UDP port at the manager should traps
   be sent (standard 162)? [5000]9162
7. What SNMP version should be used (1,2,3,1&2,1&2&3,2&3)? [3]1&2
8. Do you want a none- minimum- or semi-secure configuration?
   Note that if you chose v1 or v2, you won't get any security for these
   requests (none, minimum, semi) [minimum]
9. Where is the configuration directory (absolute)? [/usr/home/xxx/otp-snmp]
10. Current configuration files will now be overwritten. Ok [y]/n?y
-----------------------
Info: 1. SecurityName "initial" has noAuthNoPriv read access and authenticated write access to the
"restricted" subtree.
      2. SecurityName "all-rights" has noAuthNoPriv read/write access to the "internet" subtree.
      3. Standard traps are sent to the manager.
      4. Community "public" is mapped to security name "initial".
      5. Community "all-rights" is mapped to security name "all-rights".
The following files were written: agent.conf, community.conf,
   standard.conf, target_addr.conf, target_params.conf,
   notify.conf vacm.conf, sys.config
-----------------------
ok
Result is the following files: agent.conf, community.conf, context.conf, notify.conf, standard.conf,
sys.config, target_addr.conf, target_params.conf, vacm.conf.
```

Create MIB Header for Mnesia

```
3> snmp:mib_to_hrl("SAMPLE-MIB").
"SAMPLE-MIB.hrl" written.
ok
```

Result is file SAMPLE-MIB.hrl.

Create Data Directory for Mnesia

Will use non-volatile storage (disc_copies) for convenience. Designate a directory on disk where the table will be stored.

```
mkdir /tmp/db
```

Create Erlang Program

Here is a tiny Erlang module with functions to create the table in Mnesia, get and set values in the table, and start the snmp agent.

snmp_sample.erl.

Start OTP with Mnesia and SNMP agent

Here is the transcript of an Erlang shell session starting things up:

```
erl -mnesia dir '"/tmp/db"' -config ./sys -name sample
(sample@xyz.com)1> c(snmp_sample).
{ok,snmp_sample}
(sample@xyz.com)2> mnesia:create_schema([node()]).
ok
(sample@xyz.com)3> mnesia:start().
ok
(sample@xyz.com)4> snmp_sample:create_table().
{atomic,ok}
(sample@xyz.com)5> snmp_sample:set_counts("srv01",3,44).
ok
(sample@xyz.com)6> snmp_sample:get_counts().
[[{serverTable,"srv01",3,44}]]
(sample@xyz.com)7> snmp_sample:init_snmp().
<0.140.0>
```

```
(sample@xyz.com)8> snmp_mgr:g([[1,3,6,1,2,1,1,1,0]]).
ok
* Got PDU: v1, CommunityName = "public"
Response,        Request Id:112362370
(sample@xyz.com)9> snmp_mgr:gn([[1,3,6,1,4,1,99999,1]]).
ok
* Got PDU: v1, CommunityName = "public"
Response,        Request Id:38817076
  [name,5,115,114,118,48,49] = "srv01"
(sample@xyz.com)10>
```

Access the Table from net-snmp

Here is an example of a Bourne shell script reading and setting counter values.

First, the script:

```
# snmpops.sh

# Point MIBDIRS at directory containing the MIB for this example.
MIBDIRS=+/home/xxx/otp-snmp
MIBS=+SAMPLE-MIB
export MIBDIRS MIBS

snmpwalk -Ob -Cc -v 1 -c public localhost:9161 \
  .iso.org.dod.internet.private.enterprises.bazmatic

snmpget -Os -v 1 -c public localhost:9161 \
  .iso.org.dod.internet.private.enterprises.bazmatic.serverTable.serverEntry.callsIn.5.115.114.118.48.49

snmpset -Os -v 1 -c all-rights localhost:9161 \
  .iso.org.dod.internet.private.enterprises.bazmatic.serverTable.serverEntry.callsIn.5.115.114.118.48.49
= 22

snmpget -Os -v 1 -c public localhost:9161 \
  .iso.org.dod.internet.private.enterprises.bazmatic.serverTable.serverEntry.callsIn.5.115.114.118.48.49
```

Then, the results:

```
SAMPLE-MIB::name.5.115.114.118.48.49 = STRING: "srv01"
```

SAMPLE-MIB::callsIn.5.115.114.118.48.49 = INTEGER: 3

SAMPLE-MIB::callsOut.5.115.114.118.48.49 = INTEGER: 44

callsIn."srv01" = INTEGER: 3

callsIn."srv01" = INTEGER: 22

callsIn."srv01" = INTEGER: 22

References

Application for Private Enterprise Number

Erlang/OTP SNMP User's Guide See Chapter 6 - Implementation Example

Standard Names for Versions of the SNMP Protocol

revised Jul 22, 2003 - Hal Snyder & Josh Snyder

1.109 mnesia 分布协调的几个细节

发表时间: 2009-03-23 关键字: mnesia 分布

这3点描述了大概mnesia是如何运作的，多读几遍！

6.6 Loading of Tables at Start-up

At start-up Mnesia loads tables in order to make them accessible for its applications. Sometimes Mnesia decides to load all tables that reside locally, and sometimes the tables may not be accessible until Mnesia brings a copy of the table from another node.

To understand the behavior of Mnesia at start-up it is essential to understand how Mnesia reacts when it loses contact with Mnesia on another node. At this stage, Mnesia cannot distinguish between a communication failure and a "normal" node down.
When this happens, Mnesia will assume that the other node is no longer running. Whereas, in reality, the communication between the nodes has merely failed.

To overcome this situation, simply try to restart the ongoing transactions that are accessing tables on the failing node, and write a mnesia_down entry to a log file.

At start-up, it must be noted that all tables residing on nodes without a mnesia_down entry, may have fresher replicas. Their replicas may have been updated after the termination of Mnesia on the current node. In order to catch up with the latest updates, transfer a copy of the table from one of these other "fresh" nodes. If you are unlucky, other nodes may be down and you must wait for the table to be loaded on one of these nodes before receiving a fresh copy of the table.

Before an application makes its first access to a table, mnesia:wait_for_tables(TabList, Timeout) ought to be executed to ensure that the table is accessible from the local node. If the function times out the application may choose to force a load of the local replica with mnesia:force_load_table(Tab) and deliberately lose all updates that may have been performed on the other nodes while the local node was down. If Mnesia already has loaded the table on another node or intends to do so, we will copy the table from that node in order to avoid unnecessary inconsistency.

Warning
Keep in mind that it is only one table that is loaded by mnesia:force_load_table(Tab) and since committed transactions may have caused updates in several tables, the tables may now become inconsistent due to the forced load.

The allowed AccessMode of a table may be defined to either be read_only or read_write. And it may be toggled with the function mnesia:change_table_access_mode(Tab, AccessMode) in runtime. read_only tables and local_content tables will always be loaded locally, since there are no need for copying the table from other nodes. Other tables will primary be loaded remotely from active replicas on other nodes if the table already has been loaded there, or if the running Mnesia already has decided to load the table there.

At start up, Mnesia will assume that its local replica is the most recent version and load the table from disc if either situation is detected:

mnesia_down is returned from all other nodes that holds a disc resident replica of the table; or,
if all replicas are ram_copies
This is normally a wise decision, but it may turn out to be disastrous if the nodes have been disconnected due to a communication failure, since Mnesia's normal table load mechanism does not cope with communication failures.

When Mnesia is loading many tables the default load order. However, it is possible to affect the load order by explicitly changing the load_order property for the tables, with the function mnesia:change_table_load_order(Tab, LoadOrder). The LoadOrder is by default 0 for all tables, but it can be set to any integer. The table with the highest load_order will be loaded first. Changing load order is especially useful for applications that need to ensure early availability of fundamental tables. Large peripheral tables should have a low load order value, perhaps set below 0.

6.7 Recovery from Communication Failure

There are several occasions when Mnesia may detect that the network has been partitioned due to a communication failure.

One is when Mnesia already is up and running and the Erlang nodes gain contact again. Then Mnesia will try to contact Mnesia on the other node to see if it also thinks that the network has been partitioned for a while. If Mnesia on both nodes has logged mnesia_down entries from each other, Mnesia generates a system event, called {inconsistent_database, running_partitioned_network, Node} which is sent to Mnesia's event handler and other possible subscribers. The default event handler reports an error to the error logger.

Another occasion when Mnesia may detect that the network has been partitioned due to a communication failure, is at start-up. If Mnesia detects that both the local node and another node

received mnesia_down from each other it generates a {inconsistent_database, starting_partitioned_network, Node} system event and acts as described above.

If the application detects that there has been a communication failure which may have caused an inconsistent database, it may use the function mnesia:set_master_nodes(Tab, Nodes) to pinpoint from which nodes each table may be loaded.

At start-up Mnesia's normal table load algorithm will be bypassed and the table will be loaded from one of the master nodes defined for the table, regardless of potential mnesia_down entries in the log. The Nodes may only contain nodes where the table has a replica and if it is empty, the master node recovery mechanism for the particular table will be reset and the normal load mechanism will be used when next restarting.

The function mnesia:set_master_nodes(Nodes) sets master nodes for all tables. For each table it will determine its replica nodes and invoke mnesia:set_master_nodes(Tab, TabNodes) with those replica nodes that are included in the Nodes list (i.e. TabNodes is the intersection of Nodes and the replica nodes of the table). If the intersection is empty the master node recovery mechanism for the particular table will be reset and the normal load mechanism will be used at next restart.

The functions mnesia:system_info(master_node_tables) and mnesia:table_info(Tab, master_nodes) may be used to obtain information about the potential master nodes.

The function mnesia:force_load_table(Tab) may be used to force load the table regardless of which table load mechanism is activated.

6.8 Recovery of Transactions

A Mnesia table may reside on one or more nodes. When a table is updated, Mnesia will ensure that the updates will be replicated to all nodes where the table resides. If a replica happens to be inaccessible for some reason (e.g. due to a temporary node down), Mnesia will then perform the replication later.

On the node where the application is started, there will be a transaction coordinator process. If the transaction is distributed, there will also be a transaction participant process on all the other nodes where commit work needs to be performed.

Internally Mnesia uses several commit protocols. The selected protocol depends on which table that has been updated in the transaction. If all the involved tables are symmetrically replicated, (i.e. they all

have the same ram_nodes, disc_nodes and disc_only_nodes currently accessible from the coordinator node), a lightweight transaction commit protocol is used.

The number of messages that the transaction coordinator and its participants needs to exchange is few, since Mnesia's table load mechanism takes care of the transaction recovery if the commit protocol gets interrupted. Since all involved tables are replicated symmetrically the transaction will automatically be recovered by loading the involved tables from the same node at start-up of a failing node. We do not really care if the transaction was aborted or committed as long as we can ensure the ACID properties. The lightweight commit protocol is non-blocking, i.e. the surviving participants and their coordinator will finish the transaction, regardless of some node crashes in the middle of the commit protocol or not.

If a node goes down in the middle of a dirty operation the table load mechanism will ensure that the update will be performed on all replicas or none. Both asynchronous dirty updates and synchronous dirty updates use the same recovery principle as lightweight transactions.

If a transaction involves updates of asymmetrically replicated tables or updates of the schema table, a heavyweight commit protocol will be used. The heavyweight commit protocol is able to finish the transaction regardless of how the tables are replicated. The typical usage of a heavyweight transaction is when we want to move a replica from one node to another. Then we must ensure that the replica either is entirely moved or left as it was. We must never end up in a situation with replicas on both nodes or no node at all. Even if a node crashes in the middle of the commit protocol, the transaction must be guaranteed to be atomic. The heavyweight commit protocol involves more messages between the transaction coordinator and its participants than a lightweight protocol and it will perform recovery work at start-up in order to finish the abort or commit work.

The heavyweight commit protocol is also non-blocking, which allows the surviving participants and their coordinator to finish the transaction regardless (even if a node crashes in the middle of the commit protocol). When a node fails at start-up, Mnesia will determine the outcome of the transaction and recover it. Lightweight protocols, heavyweight protocols and dirty updates, are dependent on other nodes to be up and running in order to make the correct heavyweight transaction recovery decision.

If Mnesia has not started on some of the nodes that are involved in the transaction AND neither the local node or any of the already running nodes know the outcome of the transaction, Mnesia will by default wait for one. In the worst case scenario all other involved nodes must start before Mnesia can make the correct decision about the transaction and finish its start-up.

This means that Mnesia (on one node)may hang if a double fault occurs, i.e. when two nodes crash simultaneously and one attempts to start when the other refuses to start e.g. due to a hardware error.

It is possible to specify the maximum time that Mnesia will wait for other nodes to respond with a transaction recovery decision. The configuration parameter max_wait_for_decision defaults to infinity (which may cause the indefinite hanging as mentioned above) but if it is set to a definite time period (eg.three minutes), Mnesia will then enforce a transaction recovery decision if needed, in order to allow Mnesia to continue with its start-up procedure.

The downside of an enforced transaction recovery decision, is that the decision may be incorrect, due to insufficient information regarding the other nodes' recovery decisions. This may result in an inconsistent database where Mnesia has committed the transaction on some nodes but aborted it on others.

In fortunate cases the inconsistency will only appear in tables belonging to a specific application, but if a schema transaction has been inconsistently recovered due to the enforced transaction recovery decision, the effects of the inconsistency can be fatal. However, if the higher priority is availability rather than consistency, then it may be worth the risk.

If Mnesia encounters a inconsistent transaction decision a {inconsistent_database, bad_decision, Node} system event will be generated in order to give the application a chance to install a fallback or other appropriate measures to resolve the inconsistency. The default behavior of the Mnesia event handler is the same as if the database became inconsistent as a result of partitioned network (see above).

1.110 Writing a Tsung plugin

发表时间: 2009-03-26 关键字: tsung plugin

原文地址：http://www.process-one.net/en/wiki/Writing_a_Tsung_plugin/

Writing a Tsung plugin

This is a simple tutorial on writing a tsung plugin.

Since tsung is used to test servers lets define a simple server for testing. myserver.erl provides 3 operations: echo, add and subtract.

myserver.erl assumes the first byte to be a control instruction followed by 2 or more byte data. The echo operation merely returns the byte data while add and subtract performs these operations on the 2 byte data before returning the results. See the code of myserver.erl for details.

We assume the source files for tsung-1.2.1 are available. This example was compiled using Erlang OTP R11B-3.

Step 0. Download the source code for this tutorial

The source code for this tutorial is available: tutorial_tsung_1.tgz

Step 1. Create tsung configuration file

Based on the 3 operations we want to test we define the myclient.xml which
will take place of tsung.xml. You will notice myclient.xml looks very much
like any normal tsung configuration file. The main difference is in the
session definition. Here we use the ts_myclient type.

<session probability="100" name="myclient-example" type="ts_myclient">
This indicates which plugin tsung should call when creating a server request.
We choose to name this plugin ts_myclient.

The next difference is within the request element. Here we define a ‘myclient’
element indicates the operations to test followed by the relevant data.

Notice that myclient has 2 attributes: type and arith (optional).

Step 2. Update DTD

We now need to modify the tsung-1.0.dtd or validation would fail.

In the request element we add the myclient tag.

<!ELEMENT request (match*, dyn_variable*, (http | jabber | raw | pgsql | myclient))>
Next we create the myclient element.

<!ELEMENT myclient (#PCDATA) >
Followed by defining the attributes list for myclient.

<!ATTLIST myclient
arith (add | sub) #IMPLIED
type (echo | compute) #REQUIRED >
Next, we add the ts_myclient option into the element’ s type attribute list.

type (ts_http | ts_jabber | ts_pgsql | ts_myclient) #IMPLIED
We do the same to the session’ s type attribute list.

type (ts_jabber | ts_http | ts_raw | ts_pgsql | ts_myclient) #REQUIRED>
Step 3. Create include file

In PATH/include create the include file ts_myclient.hrl.
This include file should have a minimum of two records:

myclient_request: for storing request information parsed from tsung configuration. Will be use to generate server requests.
myclient_dyndata: for storing dynamic information. Not used in this case.
Step 4. Config reader

Create the ts_config_myclient.erl in PATH/src/tsung_controller to parse the XML file. This file must export parse_config/2.
ts_config_myclient:parse_config/2 is called from ts_config:parse/1.

However trying to run ts_config:parse/1 seperately seem to throw an undefined case error.

The important function definition is:

parse_config(Element = #xmlElement{name=myclient}, ...)
Within this pattern we gather the various attributes, echo, compute, add, sub,
and data creating a myclient_request record as needed. Notice for the case
of type compute we parse the data into a list of 2 integers. Ensure you keep
any data manipulation here consistant with calls in ts_myclient:get_message/1.

If your configuration file support several element types then you will need a parse_config function for each.

Step 5. ts_myclient

The final file to create is ts_myclient.erl in PATH/src/tsung.

The get_message/1 function builds the actual data to be transmitted to the server. The function returns a binary even if your protocol uses strings.

In ts_myclient:get_message/1, you can see how we create the message from the myclient_request record. Compare this with myserver:test/3 and myserver:test/1.

parse/2 deals with server responses. It is possible to parse the return data and update monitoring parameters. In the ts_myclient:parse/1 we count the number of single and multi bytes returned from the server. Obviously these must match echo and add / _sub calls.

The ts_mon:add/1 parameters are restricted to:

{count, Type} – increments a running counter
{sum, Type, Val} – adds Val to running counter
{sample_counter, Type, Value} – updates sample_counter
{sample, Type, Value} – updates counter
Step 6. Build and install

Return to PATH and type make followed by make install.

There is no need to update any make files.

Step 7. Running

Start myserver then call myserver:server() in the erlang shell to start listening to the socket.

sh> erl -s myserver start_link
1> myserver:server().
Run tsung, passing it myclient.xml:

sh> tsung -f myclient.xml
Authors

The first version of this tutorial has been written by tty.

1.111 tsung inside

发表时间: 2009-03-26 关键字: tsung inside protocol

需要大规模测试的话 tsung的插件就很好用,写插件的时候需要知道多点tsung内部的信息.

OTP Supervision tree:
==========

The application is now split in two (see tsung-inside.png for an overview):

** a single controller (tsung_controller)

* ts_config_server (gen_server). Configuration server. Session's
  definitions are kept by the config server.

* ts_mon (gen_server). Each client send reports of stats to this
  server. Several types of messages are handled by ts_mon.

* ts_os_mon (gen_server). Use to monitor remote node activity (cpu,
  memory, network traffic). Currently, use an erlang agent on remote nodes.

* ts_timer (used by ts_client when ack is global) (gen_fsm)

  servers used to construct messages:

* ts_msg_server (gen_server)
* ts_user_server (gen_server) used by jabber_* for unicity of users id

** several clients (tsung) Several nodes can be used simultaneously

This application is simpler:

* ts_launcher (gen_fsm) launch simulated users.

* ts_session_cache (gen_server) cache the sessions's definition (ask
  the config_server if it's not yet in the cache)

* 1 process per simultated client (ts_client), under
the supervision of ts_clients_sup (using simple_one_for_one)

Main modules:
==========

1/ ts_launcher. the master process that spawns other simulated clients:

1.1/ client processes: at each simulated client correspond 1 erlang process (ts_client)

1.2/ monitoring process (ts_mon)

2/ statistical module (ts_stats)

3/ protocol-specific modules (ts_jabber and ts_http, for example).

tsung use different types of acknoledgements to determine when a the response of a request is over. For each requests, 4 options are possible:

* parse -> the receiving process parse the response from the server dans can warn the sending process when the response is finish (function parse/2). This is used for HTTP.

* no_ack: as soon as the request has been sent, the next one is executed (it can be a thinktime)

* local: the request is acknoledge once a packet is received

* global: the request is acknoledge once all clients has received an acknoledgement. This has been introduced for Jabber: with that, you can set that users starts talking when everyone is connected.

How to add a new protocol, or extend an existing one:

==========

To add a new protocol, you have to create a module that implement and
exports:

```
-export([init_dynparams/0,
add_dynparams/4,
get_message/1,
session_defaults/0,
        parse/2,
        parse_config/2,
        new_session/0]).
```

There is a template file is doc/ts_template.erl

References:
==========

- Erlang

http://www.erlang.org/

Design principles:
http://www.erlang.org/doc/r7b/doc/design_principles/part_frame.html

- Jabber

http://docs.jabber.org/general/html/protocol.html

- Stochastics models:

For more details on stochastics models and application to Web
workload generators, have a look at:

Nicolas Niclausse. Mod閘isation, analyse de performance et
dimensionnement du World Wide Web. Th鑞e de Doctorat (PhD), Universit?

de Nice - Sophia Antipolis, Juin 1999.
http://www-sop.inria.fr/mistral/personnel/Nicolas.Niclausse/these.html

Z. Liu, N. Niclausse, C. Jalpa-Villanueva & S. Barbier. Traffic
Model and Performance Evaluation of Web Servers Rapport de
recherche INRIA, RR-3840 (http://www.inria.fr/rrrt/rr-3840.html)

内部的结构图在附件里面！

1.112 erlang允许不同的节点有不同的cookie

发表时间: 2009-03-28 关键字: node cookie auth

节点间的认证是通过cookie运算挑战码再比较是否相同而决定节点间可否连接。

11.7 Security

Authentication determines which nodes are allowed to communicate with each other. In a network of different Erlang nodes, it is built into the system at the lowest possible level. Each node has its own magic cookie, which is an Erlang atom.

When a nodes tries to connect to another node, the magic cookies are compared. If they do not match, the connected node rejects the connection.

At start-up, a node has a random atom assigned as its magic cookie and the cookie of other nodes is assumed to be nocookie. The first action of the Erlang network authentication server (auth) is then to read a file named $HOME/.erlang.cookie. If the file does not exist, it is created. The UNIX permissions mode of the file is set to octal 400 (read-only by user) and its contents are a random string. An atom Cookie is created from the contents of the file and the cookie of the local node is set to this using erlang:set_cookie(node(), Cookie). This also makes the local node assume that all other nodes have the same cookie Cookie.

Thus, groups of users with identical cookie files get Erlang nodes which can communicate freely and without interference from the magic cookie system. Users who want run nodes on separate file systems must make certain that their cookie files are identical on the different file systems.

For a node Node1 with magic cookie Cookie to be able to connect to, or accept a connection from, another node Node2 with a different cookie DiffCookie, the function erlang:set_cookie(Node2, DiffCookie) must first be called at Node1. Distributed systems with multiple user IDs can be handled in this way.

The default when a connection is established between two nodes, is to immediately connect all other visible nodes as well. This way, there is always a fully connected network. If there are nodes with different cookies, this method might be inappropriate and the command line flag -connect_all false must be set, see erl(1).

The magic cookie of the local node is retrieved by calling erlang:get_cookie().

我过去以为你个erlang集群只能使用一个cookie但是实际上理解错误.
默认情况下和所有的节点通信都是用本地的cookie, 但是如果别的节点有不同的cookie, 我们可以erlang:set_cookie(Node2, DiffCookie),然后再connect_node().

注意Auth模块已经废弃了，请不要使用.

1.113 读ejabberdctl学先进科技

发表时间: 2009-04-02 关键字: ejabberdctl

最完美的shell和erlang控制的结合，比其他系统要优雅不知道多少，仔细研究你会有很大的收获的，至少可以列出10个卖点.

cat ejabberdctl.template

```
#!/bin/sh

# define default configuration
POLL=true
SMP=auto
ERL_MAX_PORTS=32000
ERL_PROCESSES=250000
ERL_MAX_ETS_TABLES=1400

# define default environment variables
NODE=ejabberd
HOST=localhost
ERLANG_NODE=$NODE@$HOST
ERL=@erl@
ROOTDIR=@rootdir@
EJABBERD_CONFIG_PATH=$ROOTDIR/etc/ejabberd/ejabberd.cfg
LOGS_DIR=$ROOTDIR/var/log/ejabberd/
EJABBERD_DB=$ROOTDIR/var/lib/ejabberd/db/$NODE

# read custom configuration
CONFIG=$ROOTDIR/etc/ejabberd/ejabberdctl.cfg
[ -f "$CONFIG" ] && . "$CONFIG"

# parse command line parameters
ARGS=
while [ $# -ne 0 ] ; do
    PARAM=$1
    shift
    case $PARAM in
```

```
        --) break ;;
        --node) ERLANG_NODE=$1; shift ;;
        --config) EJABBERD_CONFIG_PATH=$1 ; shift ;;
        --ctl-config) CONFIG=$1 ; shift ;;
        --logs) LOGS_DIR=$1 ; shift ;;
        --spool) EJABBERD_DB=$1 ; shift ;;
        *) ARGS="$ARGS $PARAM" ;;
    esac
done

NAME=-name
[ "$ERLANG_NODE" = "${ERLANG_NODE%.*}" ] && NAME=-sname

ERLANG_OPTS="+K $POLL -smp $SMP +P $ERL_PROCESSES"

# define additional environment variables
EJABBERD_EBIN=$ROOTDIR/var/lib/ejabberd/ebin
EJABBERD_MSGS_PATH=$ROOTDIR/var/lib/ejabberd/priv/msgs
EJABBERD_SO_PATH=$ROOTDIR/var/lib/ejabberd/priv/lib
EJABBERD_BIN_PATH=$ROOTDIR/var/lib/ejabberd/priv/bin
EJABBERD_LOG_PATH=$LOGS_DIR/ejabberd.log
SASL_LOG_PATH=$LOGS_DIR/sasl.log
DATETIME=`date "+%Y%m%d-%H%M%S"`
ERL_CRASH_DUMP=$LOGS_DIR/erl_crash_$DATETIME.dump
ERL_INETRC=$ROOTDIR/etc/ejabberd/inetrc
HOME=$ROOTDIR/var/lib/ejabberd

# export global variables
export EJABBERD_CONFIG_PATH
export EJABBERD_MSGS_PATH
export EJABBERD_LOG_PATH
export EJABBERD_SO_PATH
export EJABBERD_BIN_PATH
export ERL_CRASH_DUMP
export ERL_INETRC
export ERL_MAX_PORTS
export ERL_MAX_ETS_TABLES
export HOME
```

```
[ -d $EJABBERD_DB ] || mkdir -p $EJABBERD_DB
[ -d $LOGS_DIR ] || mkdir -p $LOGS_DIR

# Compatibility in ZSH
#setopt shwordsplit 2>/dev/null

# start server
start ()
{
    $ERL \
      $NAME $ERLANG_NODE \
      -noinput -detached \
      -pa $EJABBERD_EBIN \
      -mnesia dir "\"$EJABBERD_DB\"" \
      -s ejabberd \
      -sasl sasl_error_logger \{file,\"$SASL_LOG_PATH\"\} \
      $ERLANG_OPTS $ARGS "$@"
}

# attach to server
debug ()
{
    echo "------------------------------------------------------------------"
    echo ""
    echo "IMPORTANT: we will attempt to attach an INTERACTIVE shell"
    echo "to an already running ejabberd node."
    echo "If an ERROR is printed, it means the connection was not succesfull."
    echo "You can interact with the ejabberd node if you know how to use it."
    echo "Please be extremely cautious with your actions,"
    echo "and exit immediately if you are not completely sure."
    echo ""
    echo "To detach this shell from ejabberd, press:"
    echo "  control+c, control+c"
    echo ""
    echo "------------------------------------------------------------------"
    echo "Press any key to continue"
    read foo
```

```
    echo ""
    $ERL \
      $NAME ${NODE}debug \
      -remsh $ERLANG_NODE \
      $ERLANG_OPTS $ARGS "$@"
}

# start interactive server
live ()
{
    echo "------------------------------------------------------------------"
    echo ""
    echo "IMPORTANT: ejabberd is going to start in LIVE (interactive) mode."
    echo "All log messages will be shown in the command shell."
    echo "You can interact with the ejabberd node if you know how to use it."
    echo "Please be extremely cautious with your actions,"
    echo "and exit immediately if you are not completely sure."
    echo ""
    echo "To exit this LIVE mode and stop ejabberd, press:"
    echo "  q().  and press the Enter key"
    echo ""
    echo "------------------------------------------------------------------"
    echo "Press any key to continue"
    read foo
    echo ""
    $ERL \
      $NAME $ERLANG_NODE \
      -pa $EJABBERD_EBIN \
      -mnesia dir "\"$EJABBERD_DB\"" \
      -s ejabberd \
      $ERLANG_OPTS $ARGS "$@"
}

# common control function
ctl ()
{
    $ERL \
      $NAME ejabberdctl \
```

```
    -noinput \
    -pa $EJABBERD_EBIN \
    -s ejabberd_ctl -extra $ERLANG_NODE $@
  result=$?
  case $result in
  0) ;;;
  *)
    echo ""
    echo "Commands to start an ejabberd node:"
    echo "  start  Start an ejabberd node in server mode"
    echo "  debug  Attach an interactive Erlang shell to a running ejabberd node"
    echo "  live   Start an ejabberd node in live (interactive) mode"
    echo ""
    echo "Optional parameters when starting an ejabberd node:"
    echo "  --config file      Config file of ejabberd:    $EJABBERD_CONFIG_PATH"
    echo "  --ctl-config file  Config file of ejabberdctl: $CONFIG"
    echo "  --logs dir         Directory for logs:         $LOGS_DIR"
    echo "  --spool dir        Database spool dir:         $EJABBERD_DB"
    echo "  --node nodename    ejabberd node name:         $ERLANG_NODE"
    echo "";;
  esac
  return $result
}

# display ctl usage
usage ()
{
  ctl
  exit
}

case $ARGS in
  ' start') start;;
  ' debug') debug;;
  ' live') live;;
  *) ctl $ARGS;;
esac
```

cat ejabberdctl.cfg.example

```
#
# In this file you can configure options that are passed by ejabberdctl
# to the erlang runtime system when starting ejabberd
#

# POLL: Kernel polling ([true|false])
#
# The kernel polling option requires support in the kernel.
# Additionaly, you need to enable this feature while compiling Erlang.
#
# Default: true
#
#POLL=true

# SMP: SMP support ([enable|auto|disable])
#
# Explanation in Erlang/OTP documentation:
# enable: starts the Erlang runtime system with SMP support enabled.
#   This may fail if no runtime system with SMP support is available.
# auto: starts the Erlang runtime system with SMP support enabled if it
#   is available and more than one logical processor are detected.
# disable: starts a runtime system without SMP support.
#
# Default: auto
#
#SMP=auto

# ERL_MAX_PORTS: Maximum number of simultaneously open Erlang ports
#
# ejabberd consumes two or three ports for every connection, either
# from a client or from another Jabber server. So take this into
# account when setting this limit.
#
# Default: 32000
# Maximum: 268435456
```

```
#
#ERL_MAX_PORTS=32000

# PROCESSES: Maximum number of Erlang processes
#
# Erlang consumes a lot of lightweight processes. If there is a lot of activity
# on ejabberd so that the maximum number of proccesses is reached, people will
# experiment greater latency times. As these processes are implemented in
# Erlang, and therefore not related to the operating system processes, you do
# not have to worry about allowing a huge number of them.
#
# Default: 250000
# Maximum: 268435456
#
#PROCESSES=250000

# ERL_MAX_ETS_TABLES: Maximum number of ETS and Mnesia tables
#
# The number of concurrent ETS and Mnesia tables is limited. When the limit is
# reached, errors will appear in the logs:
#   ** Too many db tables **
# You can safely increase this limit when starting ejabberd. It impacts memory
# consumption but the difference will be quite small.
#
# Default: 1400
#
#ERL_MAX_ETS_TABLES=1400

# The next variable allows to explicitly specify erlang node for ejabberd
# It can be given in different formats:
# ERLANG_NODE=ejabberd
#   Lets erlang add hostname to the node (ejabberd uses short name in this case)
# ERLANG_NODE=ejabberd@hostname
#   Erlang uses node name as is (so make sure that hostname is a real
#   machine hostname or you'll not be able to control ejabberd)
# ERLANG_NODE=ejabberd@hostname.domainname
#   The same as previous, but erlang will use long hostname
#   (see erl (1) manual for details)
```

```
#
# Default: ejabberd
#
#ERLANG_NODE=ejabberd
```

cat ejabberd_ctl.erl

```
%%%----------------------------------------------------------------------
%%% File    : ejabberd_ctl.erl
%%% Author  : Alexey Shchepin <alexey@process-one.net>
%%% Purpose : Ejabberd admin tool
%%% Created : 11 Jan 2004 by Alexey Shchepin <alexey@process-one.net>
%%%
%%%
%%% ejabberd, Copyright (C) 2002-2009   ProcessOne
%%%
%%% This program is free software; you can redistribute it and/or
%%% modify it under the terms of the GNU General Public License as
%%% published by the Free Software Foundation; either version 2 of the
%%% License, or (at your option) any later version.
%%%
%%% This program is distributed in the hope that it will be useful,
%%% but WITHOUT ANY WARRANTY; without even the implied warranty of
%%% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
%%% General Public License for more details.
%%%
%%% You should have received a copy of the GNU General Public License
%%% along with this program; if not, write to the Free Software
%%% Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
%%% 02111-1307 USA
%%%
%%%----------------------------------------------------------------------

-module(ejabberd_ctl).
-author('alexey@process-one.net').

-export([start/0,
         init/0,
         process/1,
```

```
        dump_to_textfile/1,
        register_commands/3,
        register_commands/4,
        unregister_commands/3,
        unregister_commands/4]).

-include("ejabberd_ctl.hrl").
-include("ejabberd.hrl").

start() ->
    case init:get_plain_arguments() of
        [SNode | Args] ->
            SNode1 = case string:tokens(SNode, "@") of
                [_Node, _Server] ->
                    SNode;
                _ ->
                    case net_kernel:longnames() of
                        true ->
                            SNode ++ "@" ++ inet_db:gethostname() ++
                                    "." ++ inet_db:res_option(domain);
                        false ->
                            SNode ++ "@" ++ inet_db:gethostname();
                        _ ->
                            SNode
                    end
            end,
            Node = list_to_atom(SNode1),
            Status = case rpc:call(Node, ?MODULE, process, [Args]) of
                        {badrpc, Reason} ->
                            ?PRINT("RPC failed on the node ~p: ~p~n",
                                    [Node, Reason]),
                            ?STATUS_BADRPC;
                        S ->
                            S
                    end,
            halt(Status);
        _ ->
            print_usage(),
```

```
        halt(?STATUS_USAGE)
    end.

init() ->
    ets:new(ejabberd_ctl_cmds, [named_table, set, public]),
    ets:new(ejabberd_ctl_host_cmds, [named_table, set, public]).


process(["status"]) ->
    {InternalStatus, ProvidedStatus} = init:get_status(),
    ?PRINT("Node ~p is ~p. Status: ~p~n",
           [node(), InternalStatus, ProvidedStatus]),
    case lists:keysearch(ejabberd, 1, application:which_applications()) of
        false ->
            ?PRINT("ejabberd is not running~n", []),
            ?STATUS_ERROR;
        {value,_Version} ->
            ?PRINT("ejabberd is running~n", []),
            ?STATUS_SUCCESS
    end;

process(["stop"]) ->
    init:stop(),
    ?STATUS_SUCCESS;

process(["restart"]) ->
    init:restart(),
    ?STATUS_SUCCESS;

process(["reopen-log"]) ->
    ejabberd_logger_h:reopen_log(),
    case application:get_env(sasl,sasl_error_logger) of
        {ok, {file, SASLfile}} ->
            error_logger:delete_report_handler(sasl_report_file_h),
            ejabberd_logger_h:rotate_log(SASLfile),
            error_logger:add_report_handler(sasl_report_file_h,
                                            {SASLfile, get_sasl_error_logger_type()});
        _ -> false
```

```
    end,
    ?STATUS_SUCCESS;

process(["register", User, Server, Password]) ->
    case ejabberd_auth:try_register(User, Server, Password) of
        {atomic, ok} ->
            ?STATUS_SUCCESS;
        {atomic, exists} ->
            ?PRINT("User ~p already registered at node ~p~n",
                   [User ++ "@" ++ Server, node()]),
            ?STATUS_ERROR;
        {error, Reason} ->
            ?PRINT("Can't register user ~p at node ~p: ~p~n",
                   [User ++ "@" ++ Server, node(), Reason]),
            ?STATUS_ERROR
    end;

process(["unregister", User, Server]) ->
    case ejabberd_auth:remove_user(User, Server) of
        {error, Reason} ->
            ?PRINT("Can't unregister user ~p at node ~p: ~p~n",
                   [User ++ "@" ++ Server, node(), Reason]),
            ?STATUS_ERROR;
        _ ->
            ?STATUS_SUCCESS
    end;

process(["backup", Path]) ->
    case mnesia:backup(Path) of
        ok ->
            ?STATUS_SUCCESS;
        {error, Reason} ->
            ?PRINT("Can't store backup in ~p at node ~p: ~p~n",
                   [filename:absname(Path), node(), Reason]),
            ?STATUS_ERROR
    end;

process(["dump", Path]) ->
```

```
    case dump_to_textfile(Path) of
        ok ->
            ?STATUS_SUCCESS;
        {error, Reason} ->
            ?PRINT("Can't store dump in ~p at node ~p: ~p~n",
                   [filename:absname(Path), node(), Reason]),
            ?STATUS_ERROR
    end;

process(["load", Path]) ->
    case mnesia:load_textfile(Path) of
        {atomic, ok} ->
            ?STATUS_SUCCESS;
        {error, Reason} ->
            ?PRINT("Can't load dump in ~p at node ~p: ~p~n",
                   [filename:absname(Path), node(), Reason]),
            ?STATUS_ERROR
    end;

process(["restore", Path]) ->
    case ejabberd_admin:restore(Path) of
        {atomic, _} ->
            ?STATUS_SUCCESS;
        {error, Reason} ->
            ?PRINT("Can't restore backup from ~p at node ~p: ~p~n",
                   [filename:absname(Path), node(), Reason]),
            ?STATUS_ERROR;
        {aborted,{no_exists,Table}} ->
            ?PRINT("Can't restore backup from ~p at node ~p: Table ~p does not exist.~n",
                   [filename:absname(Path), node(), Table]),
            ?STATUS_ERROR;
        {aborted,enoent} ->
            ?PRINT("Can't restore backup from ~p at node ~p: File not found.~n",
                   [filename:absname(Path), node()]),
            ?STATUS_ERROR
    end;

process(["install-fallback", Path]) ->
```

```
    case mnesia:install_fallback(Path) of
        ok ->
            ?STATUS_SUCCESS;
        {error, Reason} ->
            ?PRINT("Can't install fallback from ~p at node ~p: ~p~n",
                   [filename:absname(Path), node(), Reason]),
            ?STATUS_ERROR
    end;

process(["import-file", Path]) ->
    case jd2ejd:import_file(Path) of
        ok ->
            ?STATUS_SUCCESS;
        {error, Reason} ->
            ?PRINT("Can't import jabberd 1.4 spool file ~p at node ~p: ~p~n",
                   [filename:absname(Path), node(), Reason]),
            ?STATUS_ERROR
    end;

process(["import-dir", Path]) ->
    case jd2ejd:import_dir(Path) of
        ok ->
            ?STATUS_SUCCESS;
        {error, Reason} ->
            ?PRINT("Can't import jabberd 1.4 spool dir ~p at node ~p: ~p~n",
                   [filename:absname(Path), node(), Reason]),
            ?STATUS_ERROR
    end;

process(["delete-expired-messages"]) ->
    mod_offline:remove_expired_messages(),
    ?STATUS_SUCCESS;

process(["mnesia"]) ->
    ?PRINT("~p~n", [mnesia:system_info(all)]),
    ?STATUS_SUCCESS;

process(["mnesia", "info"]) ->
```

```
    mnesia:info(),
    ?STATUS_SUCCESS;

process(["mnesia", Arg]) when is_list(Arg) ->
    case catch mnesia:system_info(list_to_atom(Arg)) of
        {'EXIT', Error} -> ?PRINT("Error: ~p~n", [Error]);
        Return -> ?PRINT("~p~n", [Return])
    end,
    ?STATUS_SUCCESS;

process(["delete-old-messages", Days]) ->
    case catch list_to_integer(Days) of
        {'EXIT',{Reason, _Stack}} ->
            ?PRINT("Can't delete old messages (~p). Please pass an integer as parameter.~n",
                   [Reason]),
            ?STATUS_ERROR;
        Integer when Integer >= 0 ->
            {atomic, _} = mod_offline:remove_old_messages(Integer),
            ?PRINT("Removed messages older than ~s days~n", [Days]),
            ?STATUS_SUCCESS;
        _Integer ->
            ?PRINT("Can't delete old messages. Please pass a positive integer as parameter.~n", []),
            ?STATUS_ERROR
    end;

process(["vhost", H | Args]) ->
    case jlib:nameprep(H) of
        false ->
            ?PRINT("Bad hostname: ~p~n", [H]),
            ?STATUS_ERROR;
        Host ->
            case ejabberd_hooks:run_fold(
                   ejabberd_ctl_process, Host, false, [Host, Args]) of
                false ->
                    print_vhost_usage(Host),
                    ?STATUS_USAGE;
                Status ->
                    Status
```

```
        end
    end;

process(Args) ->
    case ejabberd_hooks:run_fold(ejabberd_ctl_process, false, [Args]) of
        false ->
            print_usage(),
            ?STATUS_USAGE;
        Status ->
            Status
    end.


print_usage() ->
    CmdDescs =
        [{"status", "get ejabberd status"},
         {"stop", "stop ejabberd"},
         {"restart", "restart ejabberd"},
         {"reopen-log", "reopen log file"},
         {"register user server password", "register a user"},
         {"unregister user server", "unregister a user"},
         {"backup file", "store a database backup to file"},
         {"restore file", "restore a database backup from file"},
         {"install-fallback file", "install a database fallback from file"},
         {"dump file", "dump a database to a text file"},
         {"load file", "restore a database from a text file"},
         {"import-file file", "import user data from jabberd 1.4 spool file"},
         {"import-dir dir", "import user data from jabberd 1.4 spool directory"},
         {"delete-expired-messages", "delete expired offline messages from database"},
         {"delete-old-messages n", "delete offline messages older than n days from database"},
         {"mnesia [info]", "show information of Mnesia system"},
         {"vhost host ...", "execute host-specific commands"}] ++
        ets:tab2list(ejabberd_ctl_cmds),
    MaxCmdLen =
        lists:max(lists:map(
                    fun({Cmd, _Desc}) ->
                            length(Cmd)
                    end, CmdDescs)),
```

```
    NewLine = io_lib:format("~n", []),
    FmtCmdDescs =
      lists:map(
        fun({Cmd, Desc}) ->
                ["  ", Cmd, string:chars($\s, MaxCmdLen - length(Cmd) + 2),
                 Desc, NewLine]
        end, CmdDescs),
    ?PRINT(
      "Usage: ejabberdctl [--node nodename] command [options]~n"
      "~n"
      "Available commands in this ejabberd node:~n"
      ++ FmtCmdDescs ++
      "~n"
      "Examples:~n"
      "  ejabberdctl restart~n"
      "  ejabberdctl --node ejabberd@host restart~n"
      "  ejabberdctl vhost jabber.example.org ...~n",
     []).

print_vhost_usage(Host) ->
    CmdDescs =
      ets:select(ejabberd_ctl_host_cmds,
                 [{{{Host, '$1'}, '$2'}, [], [{{'$1', '$2'}}]}]),
    MaxCmdLen =
      if
        CmdDescs == [] ->
          0;
        true ->
          lists:max(lists:map(
                      fun({Cmd, _Desc}) ->
                              length(Cmd)
                      end, CmdDescs))
      end,
    NewLine = io_lib:format("~n", []),
    FmtCmdDescs =
      lists:map(
        fun({Cmd, Desc}) ->
                ["  ", Cmd, string:chars($\s, MaxCmdLen - length(Cmd) + 2),
```

```
                Desc, NewLine]
        end, CmdDescs),
    ?PRINT(
      "Usage: ejabberdctl [--node nodename] vhost hostname command [options]~n"
      "~n"
      "Available commands in this ejabberd node and this vhost:~n"
      ++ FmtCmdDescs ++
      "~n"
      "Examples:~n"
      "  ejabberdctl vhost "++Host++" registered-users~n",
     []).

register_commands(CmdDescs, Module, Function) ->
    ets:insert(ejabberd_ctl_cmds, CmdDescs),
    ejabberd_hooks:add(ejabberd_ctl_process,
                       Module, Function, 50),
    ok.

register_commands(Host, CmdDescs, Module, Function) ->
    ets:insert(ejabberd_ctl_host_cmds,
               [{{Host, Cmd}, Desc} || {Cmd, Desc} <- CmdDescs]),
    ejabberd_hooks:add(ejabberd_ctl_process, Host,
                       Module, Function, 50),
    ok.

unregister_commands(CmdDescs, Module, Function) ->
    lists:foreach(fun(CmdDesc) ->
                          ets:delete_object(ejabberd_ctl_cmds, CmdDesc)
                  end, CmdDescs),
    ejabberd_hooks:delete(ejabberd_ctl_process,
                          Module, Function, 50),
    ok.

unregister_commands(Host, CmdDescs, Module, Function) ->
    lists:foreach(fun({Cmd, Desc}) ->
                          ets:delete_object(ejabberd_ctl_host_cmds,
                                            {{Host, Cmd}, Desc})
                  end, CmdDescs),
```

```
    ejabberd_hooks:delete(ejabberd_ctl_process,
                          Module, Function, 50),
    ok.

dump_to_textfile(File) ->
    dump_to_textfile(mnesia:system_info(is_running), file:open(File, write)).
dump_to_textfile(yes, {ok, F}) ->
    Tabs1 = lists:delete(schema, mnesia:system_info(local_tables)),
    Tabs = lists:filter(
             fun(T) ->
                 case mnesia:table_info(T, storage_type) of
                   disc_copies -> true;
                   disc_only_copies -> true;
                   _ -> false
                 end
             end, Tabs1),
    Defs = lists:map(
             fun(T) -> {T, [{record_name, mnesia:table_info(T, record_name)},
                            {attributes, mnesia:table_info(T, attributes)}]}
             end,
             Tabs),
    io:format(F, "~p.~n", [{tables, Defs}]),
    lists:foreach(fun(T) -> dump_tab(F, T) end, Tabs),
    file:close(F);
dump_to_textfile(_, {ok, F}) ->
    file:close(F),
    {error, mnesia_not_running};
dump_to_textfile(_, {error, Reason}) ->
    {error, Reason}.


dump_tab(F, T) ->
    W = mnesia:table_info(T, wild_pattern),
    {atomic,All} = mnesia:transaction(
                     fun() -> mnesia:match_object(T, W, read) end),
    lists:foreach(
      fun(Term) -> io:format(F,"~p.~n", [setelement(1, Term, T)]) end, All).
```

```
%% Function copied from Erlang/OTP lib/sasl/src/sasl.erl which doesn't export it
get_sasl_error_logger_type () ->
    case application:get_env (sasl, errlog_type) of
        {ok, error} -> error;
        {ok, progress} -> progress;
        {ok, all} -> all;
        {ok, Bad} -> exit ({bad_config, {sasl, {errlog_type, Bad}}});
        _ -> all
    end.
```

1.114 另一种实用的接入erlang控制台的方法

发表时间: 2009-04-06 关键字: run_erl to_erl start start_erl

能对运行中的erl系统进行控制是非常重要的一个福利，但是假如你的erl系统是后台运行的， 根本就没有shell可以让你输入。

如果你的节点有name 那么可以用JCL 或者-remsh 接入. 否则的话 你就得用如下方法：

请先参考 Embedded Systems User's Guide. 这种方式的好处是你的所有输入输出都记录在log文件里面 方便你日后查阅。

先运行
[root@localhost R13A]# which erl
/usr/local/bin/erl
确认下你的erl系统安装在那个路径。

[root@localhost bin]# /usr/local/lib/erlang/bin/start
但是我用的R12B5或者R13A发行版这样有点小问题 start没有运行起来， 我调查了半天发现有2个问题：

1. run_erl的log是设定在/usr/local/lib/erlang/log但是没有这个目录， 通过运行mkdir /usr/local/lib/erlang/log搞定

2. /usr/local/lib/erlang/releases/R13A/sys.config文件没有.
其中 R13A可能是R12B5.
通过运行echo "[]." > /usr/local/lib/erlang/releases/R13A/sys.config搞定

做了以上的步骤，现在运行
[root@localhost ~]# ps -ef|grep beam
root 19947 19946 0 03:35 pts/3 00:00:00 /usr/local/lib/erlang/erts-5.7/bin/beam.smp -- -root /usr/local/lib/erlang -progname start_erl -- -home /root -boot /usr/local/lib/erlang/releases/R13A/start -config /usr/local/lib/erlang/releases/R13A/sys
确认beam已经运行, 同时/tmp/目录下有erlang.pipe.1.r erlang.pipe.1.w 的pipe.

如果还没有运行起来 那么就看下 tail /var/log/syslog 查明出错原因

收获的时候到了

[root@localhost bin]# to_erl

Attaching to /tmp/erlang.pipe.3 (^D to exit)

1>

1>

1>

看到熟悉的shell提示符号了。退出的时候记得用^D, 而不是^C.

1.115 xml解释器的选择

发表时间: 2009-04-07 关键字: xmerl ejabberd xml

最近在项目中要用到xml解释器 面临2个选择 xmerl和ejabberd的解释器.
xmerl是官方的纯erlang的解释器， 移植性好， 接口稳定，但是速度稍微慢。
ejabber里面的xml解释器是基于libexpat的 用driver方式实现的 实现的很优雅 速度很快 也很易用。

但是考虑到是第3方的实现，而且libexpat要编译，在平台移植上会有问题，而且我测试了xmerl的速度在解释1个中等规模的xml大概在100us左右 速度也是可以接受的 所以就选择了xmerl. 它的使用可以参考lib里面的应用程序，用起来还是比较简单的。

1.116 erlang编程的小技巧 (持续更新中...)

发表时间: 2009-04-10 关键字: erlang tips

这个文章用于记录我遇到的erlang编码中的技巧。

1.

```
%% http://lukego.livejournal.com/6753.html - that doesn't care
about
%% the order in which results are received.
upmap(F, L) ->
    Parent = self(),
    Ref = make_ref(),
    [receive {Ref, Result} -> Result end
     || _ <- [spawn(fun() -> Parent ! {Ref, F(X)} end) || X <- L]].
```

2.

http://www.erlang.org/ml-archive/erlang-questions/200301/msg00053.html

Good day,

I've surprisingly found Y-combinator very useful to solve practical
problem of writing recursive anonymous functions in Erlang.

Code were translated to Erlang from
http://www.ece.uc.edu/~franco/C511/html/Scheme/ycomb.html.

```
-module(lambda).

-export([y/1, mk_fact/0]).

%% Y-combinator itself
y(X) ->
  F = fun (P) -> X(fun (Arg) -> (P(P))(Arg) end) end,
  F(F).

%% Factorial example
mk_fact() ->
```

```
 F =
   fun (FA) ->
fun (N) ->
   if (N == 0) -> 1;
      true -> N * FA(N-1)
   end
end
   end,
 y(F).
```

>From shell:

```
1> F = mk_fact().
#Fun<lambda.3.115879927>

2> F(5).
120
```

Best Regards,
Vladimir Sekissov

3.

```
1> string:concat("hello ", "world").
"hello world"
2> Module = list_to_atom("string").
string
3> Func = list_to_atom("concat").
concat
4> Module:Func("like", "this").
"likethis"
```

4.
prim模块IO操作与有同步和异步的版本。 但是inet_driver的实现全部是异步的的， 原因是内部使用反应器。

5. vanilla_driver最高效的读文件行的方法

```
1> erlang:open_port("hello.txt", [stream,line, in,eof]).
#Port<0.480>
2> flush().
Shell got {#Port<0.447>,{data,{eol,"hello"}}}
Shell got {#Port<0.447>,{data,{eol,"world"}}}
Shell got {#Port<0.447>,eof}
ok
```

6. Break模式下 'o' 查看port消息 未公开

7.

```
erts_debug:get_internal_state(node_and_dist_references).
/*
  Returns an erlang term on this format:

{{node_references,
   [{{Node, Creation}, Refc,
     [{{ReferrerType, ID},
       [{ReferenceType,References},
'...']},
      '...']},
     '...']},
  {dist_references,
   [{Node, Refc,
     [{{ReferrerType, ID},
       [{ReferenceType,References},
'...']},
      '...']},
    '...']}}
*/
```

8.

```
/*
* The "node_tables module" contain two (hash) tables: the node_table
* and the dist_table.
*
* The elements of the node_table represents a specific incarnation of
```

```
* an Erlang node and has {Nodename, Creation} pairs as keys. Elements
* in the node_table are referred to from node containers (see
* node_container_utils.h).
*
* The elements of the dist_table represents a (potential) connection
* to an Erlang node and has Nodename as key. Elements in the
* dist_table are either referred to from elements in the node_table
* or from the process or port structure of the entity controlling
* the connection.
*
* Both tables are garbage collected by reference counting.
*/
```

9.

```
%%%----------------------------------------------------------------
-
%%% The process structure in kernel is as shown in the
figure.
%%%
%%%              ---------------
%%%             | kernel_sup (A)|
%%%              ---------------
%%%                     |
%%%       ------------------------------
%%%      |             |              |
%%%  <std services> -------------   -------------
%%%   (file,code,  | erl_dist (A)| | safe_sup (1)|
%%%    rpc, ...)    -------------   -------------
%%%                       |              |
%%%                  (net_kernel,  (disk_log, pg2,
%%%                   auth, ...)     ...)
%%%
%%% The rectangular boxes are supervisors.  All supervisors
except
%%% for kernel_safe_sup terminates the enitre erlang node if any
of
%%% their children dies.  Any child that can't be restarted in
case
```

%%% of failure must be placed under one of these supervisors.
Any
%%% other child must be placed under safe_sup. These children
may
%%% be restarted. Be aware that if a child is restarted the old
state
%%% and all data will be lost.

10。 shell内置命令rp用于看格式化的数据。 比如rp(processes()).

11. OTP-6850: Literal lists, tuples, and binaries are no longer constructed at run-time as they used to be, but are stored in a per-module constant pool. Literals that are used more than once are stored only once.

12. lc可以这样用。
Eshell V5.5.3.1 (abort with ^G)
1> Nums = [1,2,3], Nouns = [shoe,box,tape], Verbs = [].
[]
2> [Nums || Nums =/= []] ++ [Nouns || Nouns =/= []] ++ [Verbs || Verbs
=/= []].
[[1,2,3],[shoe,box,tape]]

13.
```
node_port(Node)->
   {_, Owner}=lists:keyfind(owner, 1, element(2, net_kernel:node_info(Node))),
   hd([P|| P<-erlang:ports(), erlang:port_info(P, connected) == {connected,Owner}])
```

14.
inet:setopts(node_port('xx@nd-desktop'), [{high_watermark, 131072}]).

15.
erl -kernel inet_default_connect_options '[{sndbuf, 1048576}, {high_watermark, 131072}]'

16. ICC 编译选项

export \

CC=icc \
CXX=icpc \
CFLAGS=" -O3 -ip -static -static-intel -no-gcc -no-prec-div -mp -unroll2 -xT" \
CXXFLAGS="-O3 -ip -static -static-intel -no-gcc -no-prec-div -mp -unroll2 -xT" \
LDFLAGS='-static -static-intel' \
LD=xild \
AR=xiar

17.
%%
%% Load
regulator
%%
%% This is a poor mans substitute for a fair scheduler
algorithm
%% in the Erlang emulator. The mnesia_dumper process performs
many
%% costly BIF invokations and must pay for this. But since
the
%% Emulator does not handle this properly we must compensate
for
%% this with some form of load regulation of ourselves in order
to
%% not steal all computation power in the Erlang Emulator ans
make
%% other processes starve. Hopefully this is a temporary solution.

18.
The magic commands in the shell. The full list is in the manual, but the ones I use most are:
* f() - forget all variables
* f(X) - forget X
* v(42) - recall result from line 42
* v(-1) - recall result from previous line
* rr(foo) - read record definitions from module foo

19.

beam_lib:chunks can get source code from a beam that was compiled with debug on which can be really usefull

```
{ok,{_,[{abstract_code,{_,AC}}]}} = beam_lib:chunks(Beam,[abstract_code]).
  io:fwrite("~s~n", [erl_prettypr:format(erl_syntax:form_list(AC))]).
```

20.

```
ets:fun2ms(fun({Foo, _, Bar}) when Foo > 0 -> {Foo, Bar} end).
[{{'$1','_','$2'},[{'>','$1',0}],[{{'$1','$2'}}]}]
```

21.

```
# erl
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1> H = fun ({trace, _, exit,S},_)-> io:format("got bad arg~p",[S]); (_,_)->skip end.
#Fun<erl_eval.12.113037538>
2> dbg:tracer(process, {H,null}).
{ok,<0.36.0>}
3> dbg:p(new, [procs]).
{ok,[{matched,nonode@nohost,0}]}
4>
4> spawn(fun() -> abs(x) end).
got bad arg{ok,[{call,2,
                {atom,2,spawn},
                [{'fun',2,
                    {clauses,
                        [{clause,2,[],[],
                            [{call,2,{atom,2,abs},[{atom,2,x}]}]}]}}]}],
            3}got bad arg{badarg,[{erlang,abs,[x]}]}
=ERROR REPORT==== 13-Aug-2009::16:38:56 ===
Error in process <0.40.0> with exit value: {badarg,[{erlang,abs,[x]}]}

<0.40.0>
```

22;

The condition is specified by the user as a module name CModule and a function name CFunction.

When a process reaches the breakpoint, CModule:CFunction(Bindings) will be evaluated. If and only if this function call returns true, the process will stop. If the function call returns false, the breakpoint will be silently ignored.

Bindings is a list of variable bindings. Use the function int:get_binding(Variable,Bindings) to retrieve the value of Variable (given as an atom). The function returns unbound or {value,Value}.

22.

```
export ERL_TOP=/usr/lib/erlang/

dialyzer --build_plt --output_plt $HOME/.dialyzer_otp.plt -r $ERL_TOP/lib/stdlib/ebin
  $ERL_TOP/lib/kernel/ebin $ERL_TOP/lib/mnesia/ebin $ERL_TOP/lib/ssl/ebin
  $ERL_TOP/lib/asn1/ebin $ERL_TOP/lib/compiler/ebin $ERL_TOP/lib/crypto/ebin
  $ERL_TOP/lib/syntax_tools/ebin $ERL_TOP/lib/inets/ebin $ERL_TOP/lib/sasl/ebin
  $ERL_TOP/lib/odbc/ebin
```

23. 查看系统的上下文切换

```
k# cat csw.stp
#!/usr/local/bin/stap -k

global lasttime, times, execnames, switches, csw

probe kernel.function("context_switch") {
#probe scheduler.cpu_off{
  switches ++   # count number of context switches
  now = get_cycles()
  times[pid()] += now-lasttime  # accumulate cycles spent in process
  execnames[pid()] = execname() # remember name of pid
  csw[pid()]++
  lasttime = now
}

probe timer.ms(3000) {

  printf ("\n%10s %30s %20s %20s (%d switches)\n",
          "pid", "execname", "cycles", "csw", switches);
```

```
  foreach ([pid] in times-) # sort in decreasing order of cycle-count
    printf ("%10d %30s %20d %20d\n", pid, execnames[pid], times[pid], csw[pid]);

  # clear data for next report
  delete times
  delete execnames
  delete csw

  switches = 0

}

probe begin {
  lasttime = get_cycles()
}
```

1.117 Unit Testing with Erlang’s Common Test Framework

发表时间: 2009-04-10 关键字: common_test, eunit, make, otp, testing

原文地址： http://streamhacker.wordpress.com/2008/11/26/unit-testing-with-erlangs-common-test-framework/

Unit Testing with Erlang’s Common Test Framework

November 26, 2008 at 10:57 am (erlang) (common_test, eunit, make, otp, testing)

One of the first things people look for when getting started with Erlang is a unit testing framework, and EUnit tends to be the framework of choice. But I always had trouble getting EUnit to play nice with my code since it does parse transforms, which screws up the handling of include files and record definitions. And because Erlang has pattern matching, there’s really no reason for assert macros. So I looked around for alternatives and found that a testing framework called common_test has been included since Erlang/OTP-R12B. common_test (and test_server), are much more heavy duty than EUnit, but don’t let that scare you away. Once you’ve set everything up, writing and running unit tests is quite painless.
Directory Setup

I’m going to assume an OTP compliant directory setup, specifically:

1. a top level directory we’ll call project/
2. a lib/ directory containing your applications at project/lib/
3. application directories inside lib/, such as project/lib/app1/
4. code files are in app1/src/ and beam files are in app1/ebin/

So we end up with a directory structure like this:

```
project/
   lib/
      app1/
         src/
         ebin/
```

Test Suites

Inside the app1/ directory, create a directory called test/. This is where your test suites will go. Generally, you’ll have 1 test suite per code module, so if you have app1/src/module1.erl, then you’ll create app1/test/module1_SUITE.erl for all your module1 unit tests. Each test suite should look something like this: (unfortunately, wordpress doesn’t do syntax highlighting for erlang, so it looks kinda crappy)

```
-module(module1_SUITE).

% easier than exporting by name
-compile(export_all).

% required for common_test to work
-include("ct.hrl").

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% common test callbacks %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Specify a list of all unit test functions
all() -> [test1, test2].

% required, but can just return Config. this is a suite level setup function.
init_per_suite(Config) ->
    % do custom per suite setup here
    Config.

% required, but can just return Config. this is a suite level tear down function.
end_per_suite(Config) ->
    % do custom per suite cleanup here
    Config.

% optional, can do function level setup for all functions,
% or for individual functions by matching on TestCase.
init_per_testcase(TestCase, Config) ->
    % do custom test case setup here
    Config.

% optional, can do function level tear down for all functions,
```

```
% or for individual functions by matching on TestCase.
end_per_testcase(TestCase, Config) ->
    % do custom test case cleanup here
    Config.

%%%%%%%%%%%%%%%
%% test cases %%
%%%%%%%%%%%%%%%

test1(Config) ->
    % write standard erlang code to test whatever you want
    % use pattern matching to specify expected return values
    ok.

test2(Config) -> ok.
```

Test Specification

Now the we have a test suite at project/app1/test/module1_SUITE.erl, we can make a test specification so common_test knows where to find the test suites, and which suites to run. Something I found out that hard way is that common_test requires absolute paths in its test specifications. So instead of creating a file called test.spec, we’ll create a file called test.spec.in, and use make to generate the test.spec file with absolute paths.
test.spec.in

```
{logdir, "@PATH@/log"}.
{alias, app1, "@PATH@/lib/app1"}.
{suites, app1, [module1_SUITE]}.
```

Makefile

```
src:
    erl -pa lib/*/ebin -make

test.spec: test.spec.in
    cat test.spec.in | sed -e "s,@PATH@,$(PWD)," > $(PWD)/test.spec

test: test.spec src
```

```
run_test -pa $(PWD)/lib/*/ebin -spec test.spec
```

Running the Tests

As you can see above, I also use the Makefile for running the tests with the command make test. For this command to work, run_test must be installed in your PATH. To do so, you need to run /usr/lib/erlang/lib/common_test-VERSION/install.sh (where VERSION is whatever version number you currently have). See the common_test installation instructions for more information. I’m also assuming you have an Emakefile for compiling the code in lib/app1/src/ with the make src command.
Final Thoughts

So there you have it, an example test suite, a test specification, and a Makefile for running the tests. The final file and directory structure should look something like this:

```
project/
  Emakefile
  Makefile
  test.spec.in
  lib/
    app1/
      src/
        module1.erl
      ebin/
      test/
        module1_SUITE.erl
```

Now all you need to do is write your unit tests in the form of test suites and add those suites to test.spec.in. There’s a lot more you can get out of common_test, such as code coverage analysis, HTML logging, and large scale testing. I’ll be covering some of those topics in the future, but for now I’ll end with some parting thoughts from the Common Test User’s Guide:

> It’s not possible to prove that a program is correct by testing. On the contrary, it has been formally proven that it is impossible to prove programs in general by testing.
>
> There are many kinds of test suites. Some concentrate on calling every function or command... Some other do the same, but uses all kinds of illegal parameters.
>
> Aim for finding bugs. Write whatever test that has the highest probability of finding a bug, now or

in the future. Concentrate more on the critical parts. Bugs in critical subsystems are a lot more expensive than others.

Aim for functionality testing rather than implementation details. Implementation details change quite often, and the test suites should be long lived.

Aim for testing everything once, no less, no more

1.118 Building a Non-blocking TCP server using OTP...

发表时间: 2009-04-13 关键字: gen_fsm prim_inet async_accpet

原文地址： http://www.trapexit.org.nyud.net:8080/Building_a_Non-blocking_TCP_server_using_OTP_principles

这篇文章很好的演示了几个特性：
1. superbisor 动态添加child.
2. async accept.
3. gen_fsm
4. 流控制
5. application
很值得一看。

Building a Non-blocking TCP server using OTP principles
From Erlang Community
Contents
[hide]
1 Author
2 Overview
3 Server Design
4 Application and Supervisor behaviours
5 Listener Process
6 Client Socket Handling Process
7 Application File
8 Compiling
9 Running
10 Conclusion

[edit] Author
Serge Aleynikov <saleyn at gmail.com>

[edit] Overview
A reader of this tutorial is assumed to be familiar with gen_server and gen_fsm behaviours, TCP socket communications using gen_tcp module, active and passive socket modes, and OTP supervision principles.

OTP provides a convenient framework for building reliable applications. This is in part accomplished by abstracting common functionality into a set of reusable behaviours such as gen_server and gen_fsm that are linked to OTP's supervision hierarchy.

There are several known TCP server designs. The one we are going to cover involves one process listening for client connections and spawning an FSM process per connecting client. While there is support for TCP communications available in OTP through the gen_tcp module, there is no standard behavior for building non-blocking TCP servers using OTP standard guidelines. By non-blocking we imply that the listening process and the client-handling FSMs should not make any blocking calls and be readily responsive to incoming control messages (such as changes in system configuration, restart requests, etc.) without causing timeouts. Note that blocking in the context of Erlang means blocking an Erlang process rather than the emulator's OS process(es).

In this tutorial we will show how to build a non-blocking TCP server using gen_server and gen_fsm behaviours that offers flow control and is fully compliant with OTP application design principles.

A reader who is new to the OTP framework is encouraged to read Joe Armstrong's tutorial on how to build A Fault-tolerant Server using blocking gen_tcp:connect/3 and gen_tcp:accept/1 calls without involving OTP.

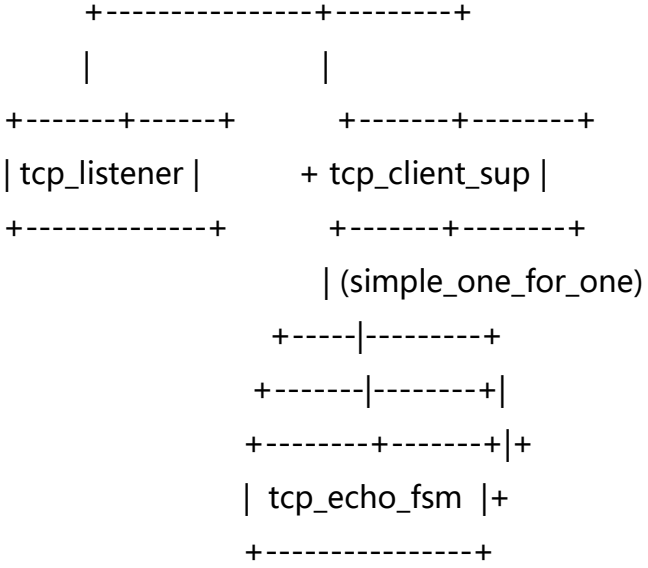
This tutorial was inspired by several threads (e.g. one, two) on the Erlang Questions mailing list mentioning an approach to building non-blocking asynchronous TCP servers.

[edit] Server Design

The design of our server will include the main application's supervisor tcp_server_app process with one_for_one restart strategy and two child specifications. The first one being a listening process implemented as a gen_server behaviour that will wait for asynchronous notifications of client socket connections. The second one is another supervisor tcp_client_sup responsible for starting client handling FSMs and logging abnormal disconnects via standard SASL error reports.

For the sake of simplicity of this tutorial, the client handling FSM (tcp_echo_fsm) will implement an echo server that will echo client's requests back to the client.

```
+----------------+
| tcp_server_app |
+--------+-------+
         | (one_for_one)
```



[edit] Application and Supervisor behaviours

In order to build an OTP application we need to construct modules implementing an application and supervisor behaviour callback functions. While traditionally these functionalities are implemented in separate modules, given their succinctness we'll combine them in one module.

As an added bonus we implement a get_app_env function that illustrates how to process configuration options as well as command-line options given to the emulator at start-up.

The two instances of init/1 function are for two tiers of supervision hierarchy. Since two different restart strategies for each supervisor are needed, we implement them at different tiers.

Upon application's startup the tcp_server_app:start/2 callback function calls supervisor:start_link/2 that creates main application's supervisor calling tcp_server_app:init([Port, Module]) callback. This supervisor creates a tcp_listener process and a child supervisor tcp_client_sup responsible for spawning client connections. The Module argument in the init function is the name of client-connection handling FSM (in this case tcp_echo_fsm).

TCP Server Application (tcp_server_app.erl)

```
-module(tcp_server_app).
-author('saleyn@gmail.com').

-behaviour(application).

%% Internal API
-export([start_client/0]).
```

```
%% Application and Supervisor callbacks
-export([start/2, stop/1, init/1]).

-define(MAX_RESTART,    5).
-define(MAX_TIME,      60).
-define(DEF_PORT,    2222).

%% A startup function for spawning new client connection handling FSM.
%% To be called by the TCP listener process.
start_client() ->
   supervisor:start_child(tcp_client_sup, []).

%%--------------------------------------------------------------------
%% Application behaviour callbacks
%%--------------------------------------------------------------------
start(_Type, _Args) ->
   ListenPort = get_app_env(listen_port, ?DEF_PORT),
   supervisor:start_link({local, ?MODULE}, ?MODULE, [ListenPort, tcp_echo_fsm]).

stop(_S) ->
   ok.

%%--------------------------------------------------------------------
%% Supervisor behaviour callbacks
%%--------------------------------------------------------------------
init([Port, Module]) ->
   {ok,
      {_SupFlags = {one_for_one, ?MAX_RESTART, ?MAX_TIME},
         [
          % TCP Listener
          {   tcp_server_sup,                    % Id       = internal id
             {tcp_listener,start_link,[Port,Module]}, % StartFun = {M, F, A}
             permanent,                          % Restart  = permanent | transient | temporary
             2000,                               % Shutdown = brutal_kill | int() >= 0 | infinity
             worker,                             % Type     = worker | supervisor
             [tcp_listener]                      % Modules  = [Module] | dynamic
          },
          % Client instance supervisor
```

```
        {   tcp_client_sup,
            {supervisor,start_link,[{local, tcp_client_sup}, ?MODULE, [Module]]},
            permanent,                  % Restart  = permanent | transient | temporary
            infinity,                   % Shutdown = brutal_kill | int() >= 0 | infinity
            supervisor,                 % Type     = worker | supervisor
            []                          % Modules  = [Module] | dynamic
        }
       ]
    }
  };

init([Module]) ->
  {ok,
    {_SupFlags = {simple_one_for_one, ?MAX_RESTART, ?MAX_TIME},
      [
       % TCP Client
       {   undefined,                   % Id       = internal id
           {Module,start_link,[]},      % StartFun = {M, F, A}
           temporary,                   % Restart  = permanent | transient | temporary
           2000,                        % Shutdown = brutal_kill | int() >= 0 | infinity
           worker,                      % Type     = worker | supervisor
           []                           % Modules  = [Module] | dynamic
       }
      ]
    }
  }.

%%--------------------------------------------------------------------
%% Internal functions
%%--------------------------------------------------------------------
get_app_env(Opt, Default) ->
  case application:get_env(application:get_application(), Opt) of
  {ok, Val} -> Val;
  _ ->
    case init:get_argument(Opt) of
    [[Val | _]] -> Val;
    error       -> Default
    end
```

end.

[edit] Listener Process

One of the shortcomings of the gen_tcp module is that it only exports interface to a blocking accept call. This leads most of developers working on an implementation of a TCP server build a custom process linked to a supervisor using proc_lib or come up with some other proprietary design.

Examining prim_inet module reveals an interesting fact that the actual call to inet driver to accept a client socket is asynchronous. While this is a non-documented property, which means that the OTP team is free to change this implementation, we will exploit this functionality in the construction of our server.

The listener process is implemented as a gen_server behaviour:

TCP Listener Process (tcp_listener.erl)

```
-module(tcp_listener).
-author('saleyn@gmail.com').

-behaviour(gen_server).

%% External API
-export([start_link/2]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
        code_change/3]).

-record(state, {
               listener,       % Listening socket
               acceptor,       % Asynchronous acceptor's internal reference
               module          % FSM handling module
              }).

%%--------------------------------------------------------------------
%% @spec (Port::integer(), Module) -> {ok, Pid} | {error, Reason}
%
```

```
%% @doc Called by a supervisor to start the listening process.
%% @end
%%--------------------------------------------------------------------
start_link(Port, Module) when is_integer(Port), is_atom(Module) ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [Port, Module], []).

%%%------------------------------------------------------------------------
%%% Callback functions from gen_server
%%%------------------------------------------------------------------------

%%--------------------------------------------------------------------
%% @spec (Port::integer()) -> {ok, State}           |
%%                            {ok, State, Timeout}  |
%%                            ignore                |
%%                            {stop, Reason}
%%
%% @doc Called by gen_server framework at process startup.
%%      Create listening socket.
%% @end
%%--------------------------------------------------------------------
init([Port, Module]) ->
    process_flag(trap_exit, true),
    Opts = [binary, {packet, 2}, {reuseaddr, true},
            {keepalive, true}, {backlog, 30}, {active, false}],
    case gen_tcp:listen(Port, Opts) of
    {ok, Listen_socket} ->
        %%Create first accepting process
        {ok, Ref} = prim_inet:async_accept(Listen_socket, -1),
        {ok, #state{listener = Listen_socket,
                    acceptor = Ref,
                    module   = Module}};
    {error, Reason} ->
        {stop, Reason}
    end.

%%--------------------------------------------------------------------
%% @spec (Request, From, State) -> {reply, Reply, State}          |
%%                                 {reply, Reply, State, Timeout} |
```

```
%%                      {noreply, State}            |
%%                      {noreply, State, Timeout}   |
%%                      {stop, Reason, Reply, State}  |
%%                      {stop, Reason, State}
%% @doc Callback for synchronous server calls.  If `{stop, ...}' tuple
%%      is returned, the server is stopped and `terminate/2' is called.
%% @end
%% @private
%%--------------------------------------------------------------------
handle_call(Request, _From, State) ->
   {stop, {unknown_call, Request}, State}.

%%--------------------------------------------------------------------
%% @spec (Msg, State) ->{noreply, State}          |
%%                      {noreply, State, Timeout} |
%%                      {stop, Reason, State}
%% @doc Callback for asyncrous server calls.  If `{stop, ...}' tuple
%%      is returned, the server is stopped and `terminate/2' is called.
%% @end
%% @private
%%--------------------------------------------------------------------
handle_cast(_Msg, State) ->
   {noreply, State}.

%%--------------------------------------------------------------------
%% @spec (Msg, State) ->{noreply, State}          |
%%                      {noreply, State, Timeout} |
%%                      {stop, Reason, State}
%% @doc Callback for messages sent directly to server's mailbox.
%%      If `{stop, ...}' tuple is returned, the server is stopped and
%%      `terminate/2' is called.
%% @end
%% @private
%%--------------------------------------------------------------------
handle_info({inet_async, ListSock, Ref, {ok, CliSocket}},
            #state{listener=ListSock, acceptor=Ref, module=Module} = State) ->
   try
      case set_sockopt(ListSock, CliSocket) of
```

```
        ok              -> ok;
        {error, Reason} -> exit({set_sockopt, Reason})
        end,

        %% New client connected - spawn a new process using the simple_one_for_one
        %% supervisor.
        {ok, Pid} = tcp_server_app:start_client(),
        gen_tcp:controlling_process(CliSocket, Pid),
        %% Instruct the new FSM that it owns the socket.
        Module:set_socket(Pid, CliSocket),

        %% Signal the network driver that we are ready to accept another connection
        case prim_inet:async_accept(ListSock, -1) of
        {ok,    NewRef} -> ok;
        {error, NewRef} -> exit({async_accept, inet:format_error(NewRef)})
        end,

        {noreply, State#state{acceptor=NewRef}}
    catch exit:Why ->
        error_logger:error_msg("Error in async accept: ~p.\n", [Why]),
        {stop, Why, State}
    end;

handle_info({inet_async, ListSock, Ref, Error}, #state{listener=ListSock, acceptor=Ref} = State) ->
    error_logger:error_msg("Error in socket acceptor: ~p.\n", [Error]),
    {stop, Error, State};

handle_info(_Info, State) ->
    {noreply, State}.

%%--------------------------------------------------------------------
%% @spec (Reason, State) -> any
%% @doc  Callback executed on server shutdown. It is only invoked if
%%       `process_flag(trap_exit, true)' is set by the server process.
%%       The return value is ignored.
%% @end
%% @private
%%--------------------------------------------------------------------
```

```
terminate(_Reason, State) ->
    gen_tcp:close(State#state.listener),
    ok.

%%--------------------------------------------------------------------
%% @spec (OldVsn, State, Extra) -> {ok, NewState}
%% @doc  Convert process state when code is changed.
%% @end
%% @private
%%--------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

%%%------------------------------------------------------------------------
%%% Internal functions
%%%------------------------------------------------------------------------

%% Taken from prim_inet.  We are merely copying some socket options from the
%% listening socket to the new client socket.
set_sockopt(ListSock, CliSocket) ->
    true = inet_db:register_socket(CliSocket, inet_tcp),
    case prim_inet:getopts(ListSock, [active, nodelay, keepalive, delay_send, priority, tos]) of
    {ok, Opts} ->
        case prim_inet:setopts(CliSocket, Opts) of
        ok    -> ok;
        Error -> gen_tcp:close(CliSocket), Error
        end;
    Error ->
        gen_tcp:close(CliSocket), Error
    end.
```

In this module init/1 call takes two parameters - the port number that the TCP listener should be started on and the name of a protocol handling module for client connections. The initialization function opens a listening socket in passive {active, false} mode. This is done so that we have flow control of the data received on the connected client sockets that will inherit this option from the listening socket.

The most interesting part of this code is the prim_inet:async_accept/2 call as well as the handling of asynchronous inet_async messages. In order to get this working we also needed to copy some of the internal OTP code encapsulated in the set_sockopt/2 function that handles socket registration with inet database and copying some options to the client socket.

As soon as a client socket is connected inet driver will notify the listening process using {inet_async, ListSock, Ref, {ok, CliSocket}} message. At this point we'll instantiate a new client socket handling process and set its ownership of the CliSocket.

[edit] Client Socket Handling Process

While tcp_listener is a generic implementation, tcp_echo_fsm is a mere stub FSM for illustrating how to write TCP servers. This modules needs to export two functions - one start_link/0 for a tcp_client_sup supervisor and another set_socket/2 for the listener process to notify the client connection handling FSM process that it is now the owner of the socket, and can begin receiving messages by setting the {active, once} or {active, true} option.

We would like to highlight the synchronization pattern used between the listening process and client connection-handling FSM to avoid possible message loss due to dispatching some messages from the socket to the wrong (listening) process. The process owning the listening socket has it open with {active, false}. After accepting the client's socket that socket inherits its socket options (including {active, false}) from the listener, transfers ownership of the socket to the newly spawned client connection-handling FSM by calling gen_tcp:controlling_process/2 and calls Module:set_socket/2 to notify the FSM that it can start receiving messages from the socket. Until the FSM process enables message delivery by setting the active mode on the socket by calling inet:setopts(Socket, [{active, once}]), the data sent by the TCP sender stays in the socket buffer.

When socket ownership is transfered to FSM in the 'WAIT_FOR_SOCKET' state the FSM sets {active, once} option to let inet driver send it one TCP message at a time. This is the OTP way of preserving flow control and avoiding process message queue flooding with TCP data and crashing the system in case of a fast-producer-slow-consumer case.

The FSM states are implemented by special functions in the tcp_echo_fsm module that use a naming convention with capital case state names enclosed in single quotes. The FSM consists of two states. 'WAIT_FOR_SOCKET' is the initial state in which the FSM is waiting for assignment of socket ownership, and 'WAIT_FOR_DATA' is the state that represents awaiting for TCP message from a client. In this state FSM also handles a special 'timeout' message that signifies no activity from a client and causes the process to stop and close client connection.

TCP Client Socket Handling FSM (tcp_echo_fsm.erl)

```
-module(tcp_echo_fsm).
-author('saleyn@gmail.com').

-behaviour(gen_fsm).

-export([start_link/0, set_socket/2]).

%% gen_fsm callbacks
-export([init/1, handle_event/3,
        handle_sync_event/4, handle_info/3, terminate/3, code_change/4]).

%% FSM States
-export([
   'WAIT_FOR_SOCKET'/2,
   'WAIT_FOR_DATA'/2
]).

-record(state, {
               socket,    % client socket
               addr       % client address
              }).

-define(TIMEOUT, 120000).

%%%------------------------------------------------------------------------
%%% API
%%%------------------------------------------------------------------------

%%-------------------------------------------------------------------------
%% @spec (Socket) -> {ok,Pid} | ignore | {error,Error}
%% @doc To be called by the supervisor in order to start the server.
%%      If init/1 fails with Reason, the function returns {error,Reason}.
%%      If init/1 returns {stop,Reason} or ignore, the process is
%%      terminated and the function returns {error,Reason} or ignore,
%%      respectively.
%% @end
```

```
%%------------------------------------------------------------------------
start_link() ->
    gen_fsm:start_link(?MODULE, [], []).

set_socket(Pid, Socket) when is_pid(Pid), is_port(Socket) ->
    gen_fsm:send_event(Pid, {socket_ready, Socket}).

%%%------------------------------------------------------------------------
%%% Callback functions from gen_server
%%%------------------------------------------------------------------------

%%------------------------------------------------------------------------
%% Func: init/1
%% Returns: {ok, StateName, StateData}          |
%%          {ok, StateName, StateData, Timeout} |
%%          ignore                              |
%%          {stop, StopReason}
%% @private
%%------------------------------------------------------------------------
init([]) ->
    process_flag(trap_exit, true),
    {ok, 'WAIT_FOR_SOCKET', #state{}}.

%%------------------------------------------------------------------------
%% Func: StateName/2
%% Returns: {next_state, NextStateName, NextStateData}          |
%%          {next_state, NextStateName, NextStateData, Timeout} |
%%          {stop, Reason, NewStateData}
%% @private
%%------------------------------------------------------------------------
'WAIT_FOR_SOCKET'({socket_ready, Socket}, State) when is_port(Socket) ->
    % Now we own the socket
    inet:setopts(Socket, [{active, once}, {packet, 2}, binary]),
    {ok, {IP, _Port}} = inet:peername(Socket),
    {next_state, 'WAIT_FOR_DATA', State#state{socket=Socket, addr=IP}, ?TIMEOUT};
'WAIT_FOR_SOCKET'(Other, State) ->
    error_logger:error_msg("State: 'WAIT_FOR_SOCKET'. Unexpected message: ~p\n", [Other]),
    %% Allow to receive async messages
```

```
    {next_state, 'WAIT_FOR_SOCKET', State}.

%% Notification event coming from client
'WAIT_FOR_DATA'({data, Data}, #state{socket=S} = State) ->
    ok = gen_tcp:send(S, Data),
    {next_state, 'WAIT_FOR_DATA', State, ?TIMEOUT};

'WAIT_FOR_DATA'(timeout, State) ->
    error_logger:error_msg("~p Client connection timeout - closing.\n", [self()]),
    {stop, normal, State};

'WAIT_FOR_DATA'(Data, State) ->
    io:format("~p Ignoring data: ~p\n", [self(), Data]),
    {next_state, 'WAIT_FOR_DATA', State, ?TIMEOUT}.

%%-------------------------------------------------------------------------
%% Func: handle_event/3
%% Returns: {next_state, NextStateName, NextStateData}          |
%%          {next_state, NextStateName, NextStateData, Timeout} |
%%          {stop, Reason, NewStateData}
%% @private
%%-------------------------------------------------------------------------
handle_event(Event, StateName, StateData) ->
    {stop, {StateName, undefined_event, Event}, StateData}.

%%-------------------------------------------------------------------------
%% Func: handle_sync_event/4
%% Returns: {next_state, NextStateName, NextStateData}            |
%%          {next_state, NextStateName, NextStateData, Timeout}   |
%%          {reply, Reply, NextStateName, NextStateData}          |
%%          {reply, Reply, NextStateName, NextStateData, Timeout} |
%%          {stop, Reason, NewStateData}                          |
%%          {stop, Reason, Reply, NewStateData}
%% @private
%%-------------------------------------------------------------------------
handle_sync_event(Event, _From, StateName, StateData) ->
    {stop, {StateName, undefined_event, Event}, StateData}.
```

```
%%--------------------------------------------------------------------
%% Func: handle_info/3
%% Returns: {next_state, NextStateName, NextStateData}          |
%%          {next_state, NextStateName, NextStateData, Timeout} |
%%          {stop, Reason, NewStateData}
%% @private
%%--------------------------------------------------------------------
handle_info({tcp, Socket, Bin}, StateName, #state{socket=Socket} = StateData) ->
    % Flow control: enable forwarding of next TCP message
    inet:setopts(Socket, [{active, once}]),
    ?MODULE:StateName({data, Bin}, StateData);

handle_info({tcp_closed, Socket}, _StateName,
            #state{socket=Socket, addr=Addr} = StateData) ->
    error_logger:info_msg("~p Client ~p disconnected.\n", [self(), Addr]),
    {stop, normal, StateData};

handle_info(_Info, StateName, StateData) ->
    {noreply, StateName, StateData}.

%%--------------------------------------------------------------------
%% Func: terminate/3
%% Purpose: Shutdown the fsm
%% Returns: any
%% @private
%%--------------------------------------------------------------------
terminate(_Reason, _StateName, #state{socket=Socket}) ->
    (catch gen_tcp:close(Socket)),
    ok.

%%--------------------------------------------------------------------
%% Func: code_change/4
%% Purpose: Convert process state when code is changed
%% Returns: {ok, NewState, NewStateData}
%% @private
%%--------------------------------------------------------------------
code_change(_OldVsn, StateName, StateData, _Extra) ->
    {ok, StateName, StateData}.
```

[edit] Application File

Another required part of building an OTP application is creation of an application file that includes application name, version, startup module and environment.

Application File (tcp_server.app)

```
{application, tcp_server,
[
  {description, "Demo TCP server"},
  {vsn, "1.0"},
  {id, "tcp_server"},
  {modules,      [tcp_listener, tcp_echo_fsm]},
  {registered,   [tcp_server_sup, tcp_listener]},
  {applications, [kernel, stdlib]},
  %%
  %% mod: Specify the module name to start the application, plus args
  %%
  {mod, {tcp_server_app, []}},
  {env, []}
]
}.
```

[edit] Compiling

Create the following directory structure for this application:

```
./tcp_server
./tcp_server/ebin/
./tcp_server/ebin/tcp_server.app
./tcp_server/src/tcp_server_app.erl
./tcp_server/src/tcp_listener.erl
./tcp_server/src/tcp_echo_fsm.erl
$ cd tcp_server/src
$ for f in tcp*.erl ; do erlc +debug_info -o ../ebin $f
```

[edit] Running

We are going to start an Erlang shell with SASL support so that we can view all progress and error

reports for our TCP application. Also we are going to start appmon application in order to examine visually the supervision hierarchy.

```
$ cd ../ebin
$ erl -boot start_sasl
...
1> appmon:start().
{ok,<0.44.0>}
2> application:start(tcp_server).
ok
```

Now click on the tcp_server button in the appmon's window in order to display supervision hierarchy of the tcp_server application.

```
3> {ok,S} = gen_tcp:connect({127,0,0,1},2222,[{packet,2}]).
{ok,#Port<0.150>}
```

The step above initiated a new client connection to the echo server.

```
4> gen_tcp:send(S,<<"hello">>).
ok
5> f(M), receive M -> M end.
{tcp,#Port<0.150>,"hello"}
```

We verified that the echo server works as expected. Now let's try to crash the client connection on the server and watch for the supervisor generating an error report entry on screen.

```
6> [{_,Pid,_,_}] = supervisor:which_children(tcp_client_sup).
[{undefined,<0.64.0>,worker,[]}]
7> exit(Pid,kill).
true
=SUPERVISOR REPORT==== 31-Jul-2007::14:33:49 ===
     Supervisor: {local,tcp_client_sup}
     Context:    child_terminated
     Reason:     killed
     Offender:   [{pid,<0.77.0>},
                  {name,undefined},
                  {mfa,{tcp_echo_fsm,start_link,[]}},
                  {restart_type,temporary},
                  {shutdown,2000},
                  {child_type,worker}]
```

Note that if you are putting this server under a stress test with many incoming connections, the listener process may fail to accept new connections after the number of open file descriptors reaches the limit set by the operating system. In that case you will see the error:

"too many open files"

If you are running Linux/UNIX, google for a solution (which ultimately boils down to increasing the per-process limit by setting "ulimit -n ..." option).

[edit] Conclusion

OTP provides building blocks for constructing non-blocking TCP servers. This tutorial showed how to create a simple TCP server with flow control using standard OTP behaviours. As an exercise the reader is encouraged to try abstracting generic non-blocking TCP server functionality into a stand-along behaviour

1.119 Linux下Erlang使用UnixODBC连接数据库

发表时间: 2009-04-16 关键字: linux unixodbc mysql

erlang有个odbc的模块可以使用传统的数据库，配置步骤如下：

在ubuntu下首先安装这几个包：
apt-get -y install unixodbc unixodbc-bin libmyodbc

当然要先安装这几个包才能顺利编译erlang.

然后
cp /usr/share/libmyodbc/odbcinst.ini /etc/

然后
root@yufeng-desktop:~# cat /etc/odbc.ini
[MySQL-Test]
Description = test
Driver = /usr/lib/odbc/libmyodbc.so
Server = localhost
Database = test
Port = 3306

unixODBC 有 2 個圖形化介面工具可以使用，分別是：
ODBCConfig
DataManager
运行它们 确保你的数据源可用。

再来验证下
root@yufeng-desktop:~# mysql
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 48
Server version: 5.0.67-0ubuntu6 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> select version();
+-----------------+

```
| version()       |
+-----------------+
| 5.0.67-0ubuntu6 |
+-----------------+
1 row in set (0.00 sec)

mysql> Aborted
```

如果都没有问题 成功一半了。

```
root@yufeng-desktop:/etc# erl
Erlang R13B (erts-5.7.1) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.1  (abort with ^G)
1> application:start(odbc).
ok
2> {ok, Ref}=odbc:connect("DSN=MySQL-Test;UID=root;PWD=", [{trace_driver, on}]).
{ok,<0.42.0>}
3> odbc:sql_query(Ref, "select version();").
{selected,["version()"],[{"5.0.67-0ubuntu6"}]}
4> odbc:disconnect(Ref).
ok
5>
```

O yeah， 可以使用这些传统的数据了.

另外pstree看下：

```
├─sshd───sshd─┬─bash───beam.smp─┬─odbcserver───{odbcserver}
    |        |                 └─5*[{beam.smp}]
    |        ├─bash───pstree
    |        └─bash───wget
```

也就是说odbc是作为一个OS进程运行的 这样保证了erl的稳定性。

1.120 find out which line my Erlang program crash

发表时间: 2009-04-16 关键字: smart_exception 行号

litao同学告诉我的：

The run-time system's error reports tell you which function crashed, but not the line number. Consider this module:

```
-module(crash).
-export([f/0]).

f() ->
  g().

g() ->
  A = 4,
  b = 5,  % Error is on this line.
  A.


3> crash:f().
** exited: {{badmatch,5},[{crash,g,0},{shell,exprs,6},{shell,eval_loop,3}]} **
```

The crash message tells us that the crash was in the function crash:g/1, and that it was a badmatch. For programs with short functions, this is enough information for an experienced programmer to find the problem.

In cases where you want more information, there are two options. The first is to use the erlang debugger to single-step the program.

Another option is to compile your code using the smart exceptions package, available from the jungerl. Smart exceptions then provide line number information:

```
4> c(crash, [{parse_transform, smart_exceptions}]).
{ok,crash}
5> crash:f().
```

** exited: {{crash,g,0},{line,9},match,[5]} **

smart_exception包在这里下载 http://pepper.sherry.jp/mikage/smart_exceptions.tar.gz
这个包的年代比较久了 makefile有些问题
这样编译 erlc *.erl 一堆警告不怕。

附件下载:

- smart_exceptions.tar.gz (78 KB)
- dl.javaeye.com/topics/download/0c644ea6-65e0-3b53-a27b-b7ccf36949f7

1.121 抄书 Drivers in general

发表时间: 2009-04-16 关键字: drivers in general

3.2 The driver

Although Erlang drivers in general may be beyond the scope of this document, a brief introduction seems to be in place.

3.2.1 Drivers in general

An Erlang driver is a native code module written in C (or assembler) which serves as an interface for some special operating system service. This is a general mechanism that is used throughout the Erlang emulator for all kinds of I/O. An Erlang driver can be dynamically linked (or loaded) to the Erlang emulator at runtime by using the erl_ddll Erlang module. Some of the drivers in OTP are however statically linked to the runtime system, but that's more an optimization than a necessity.

The driver data-types and the functions available to the driver writer are defined in the header file erl_driver.h (there is also an deprecated version called driver.h, don't use that one.) seated in Erlang's include directory (and in $ERL_TOP/erts/emulator/beam in the source code distribution). Refer to that file for function prototypes etc.

When writing a driver to make a communications protocol available to Erlang, one should know just about everything worth knowing about that particular protocol. All operation has to be non blocking and all possible situations should be accounted for in the driver. A non stable driver will affect and/or crash the whole Erlang runtime system, which is seldom what's wanted.

The emulator calls the driver in the following situations:

1. When the driver is loaded. This call-back has to have a special name and will inform the emulator of what call-backs should be used by returning a pointer to a ErlDrvEntry struct, which should be properly filled in (see below).

2. When a port to the driver is opened (by a open_port call from Erlang). This routine should set up internal data structures and return an opaque data entity of the type ErlDrvData, which is a data-type large enough to hold a pointer. The pointer returned by this function will be the first argument to all other call-backs concerning this particular port. It is usually called the port handle. The emulator only stores the handle and does never try to interpret it, why it can be virtually anything (well anything not larger than a pointer that is) and can point to anything if it is a pointer. Usually this pointer will refer to a structure holding information about the particular port, as i t does in our example.

3. When an Erlang process sends data to the port. The data will arrive as a buffer of bytes, the interpretation is not defined, but is up to the implementor. This call-back returns nothing to the caller, answers are sent to the caller as messages (using a routine called driver_output available to all drivers). There is also a way to talk in a synchronous way to drivers, described below. There can be an additional call-back function for handling data that is fragmented (sent in a deep io-list). That interface will get the data in a form suitable for Unix writev rather than in a single buffer. There is no need for a distribution driver to implement such a call-back, so we wont.

4. When a file descriptor is signaled for input. This call-back is called when the emulator detects input on a file descriptor which the driver has marked for monitoring by using the interface driver_select. The mechanism of driver select makes it possible to read non blocking from file descriptors by calling driver_select when reading is needed and then do the actual reading in this call-back (when reading is actually possible). The typical scenario is that driver_select is called when an Erlang process orders a read operation, and that this routine sends the answer when data is available on the file descriptor.

5. When a file descriptor is signaled for output. This call-back is called in a similar way as the previous, but when writing to a file descriptor is possible. The usual scenario is that Erlang orders writing on a file descriptor and that the driver calls driver_select. When the descriptor is ready for output, this call-back is called an the driver can try to send the output. There may of course be queuing involved in such operations, and there are some convenient queue routines available to the driver writer to use in such situations.

6. When a port is closed, either by an Erlang process or by the driver calling one of the driver_failure_XXX routines. This routine should clean up everything connected to one particular port. Note that when other call-backs call a driver_failure_XXX routine, this routine will be immediately called and the call-back routine issuing the error can make no more use of the data structures for the port, as this routine surely has freed all associated data and closed all file descriptors. If the queue utility available to driver writes is used, this routine will however not be called until the queue is empty.

7. When an Erlang process calls erlang:driver_control/2, which is a synchronous interface to drivers. The control interface is used to set driver options, change states of ports etc. We'll use this interface quite a lot in our example.

8. When a timer expires. The driver can set timers with the function driver_set_timer. When such timers expire, a specific call-back function is called. We will not use timers in our example.

9. When the whole driver is unloaded. Every resource allocated by the driver should be freed.

1.122 measure memory consumption in an Erlang system

发表时间: 2009-04-19 关键字: measure memory consumption

原文地址：http://erlang.org/faq/how_do_i.html#5.15

Memory consumption is a bit of a tricky issue in Erlang. Usually, you don't need to worry about it because the garbage collector looks after memory management for you. But, when things go wrong, there are several sources of information. Starting from the most general:

Some operating systems provide detailed information about process memory use with tools like top, ps or the linux /proc filesystem:

```
cat /proc/5898/status

VmSize:     7660 kB
VmLck:         0 kB
VmRSS:      5408 kB
VmData:     4204 kB
VmStk:        20 kB
VmExe:       576 kB
VmLib:      2032 kB
```

This gives you a rock-solid upper-bound on the amount of memory the entire Erlang system is using.

erlang:system_info reports interesting things about some globally allocated structures in bytes:

```
3> erlang:system_info(allocated_areas).
[{static,390265},
 {atom_space,65544,49097},
 {binary,13866},
 {atom_table,30885},
 {module_table,944},
 {export_table,16064},
 {register_table,240},
 {loaded_code,1456353},
 {process_desc,16560,15732},
```

```
{table_desc,1120,1008},
{link_desc,6480,5688},
{atom_desc,107520,107064},
{export_desc,95200,95080},
{module_desc,4800,4520},
{preg_desc,640,608},
{mesg_desc,960,0},
{plist_desc,0,0},
{fixed_deletion_desc,0,0}]
```

Information about individual processes can be obtained from erlang:process_info/1 or erlang:process_info/2:

```
2> erlang:process_info(self(), memory).
{memory,1244}
```

The shell's i() and the pman tool also give useful overview information.

Don't expect the sum of the results from process_info and system_info to add up to the total memory use reported by the operating system. The Erlang runtime also uses memory for other things.

A typical approach when you suspect you have memory problems is

1. Confirm that there really is a memory problem by checking that memory use as reported by the operating system is unexpectedly high.

2. Use pman or the shell's i() command to make sure there isn't an out-of-control erlang process on the system. Out-of-control processes often have enormous message queues. A common reason for Erlang processes to get unexpectedly large is an endlessly looping function which isn't tail recursive.

3. Check the amount of memory used for binaries (reported by system_info). Normal data in Erlang is put on the process heap, which is garbage collected. Large binaries, on the other hand, are reference counted. This has two interesting consequences. Firstly, binaries don't count towards a process' memory use. Secondly, a lot of memory can be allocated in binaries without causing a process' heap to grow much. If the heap doesn't grow, it's likely that there won't be a garbage collection, which may cause binaries to hang around longer than expected. A strategically-placed call to

erlang:garbage_collect() will help.

4. If all of the above have failed to find the problem, start the Erlang runtime system with the -instr switch.

1.123 User-Defined Behaviours

发表时间: 2009-04-26 关键字: user-defined behaviours

To implement a user-defined behaviour, write code similar to code for a special process but calling functions in a callback module for handling specific tasks.

If it is desired that the compiler should warn for missing callback functions, as it does for the OTP behaviours, implement and export the function:

```
behaviour_info(callbacks) ->
    [{Name1,Arity1},...,{NameN,ArityN}].
```

where each {Name,Arity} specifies the name and arity of a callback function.

When the compiler encounters the module attribute -behaviour(Behaviour). in a module Mod, it will call Behaviour:behaviour_info(callbacks) and compare the result with the set of functions actually exported from Mod, and issue a warning if any callback function is missing.

Example:

```
%% User-defined behaviour module
-module(simple_server).
-export([start_link/2,...]).
-export([behaviour_info/1]).

behaviour_info(callbacks) ->
    [{init,1},
     {handle_req,1},
     {terminate,0}].

start_link(Name, Module) ->
    proc_lib:start_link(?MODULE, init, [self(), Name, Module]).

init(Parent, Name, Module) ->
    register(Name, self()),
    ...,
    Dbg = sys:debug_options([]),
```

```
    proc_lib:init_ack(Parent, {ok, self()}),
    loop(Parent, Module, Deb, ...).

...
```

In a callback module:

```
-module(db).
-behaviour(simple_server).

-export([init/0, handle_req/1, terminate/0]).

...
```

1.124 未公开的ram_file 内存文件

发表时间: 2009-04-27 关键字: ram_file ram_file_drv

Ram_file是erlang未公开的一个模块, 是在内存文件的一个实现，erts的内置驱动ram_file_drv提供底层快速的内存访问。它的用途是需要文件访问接口的模块如erl_tar之类的可以在内存里面提供高速的文件访问服务。 所以ram_file提供file所提供的正常接口以外，还支持以下接口：

```
%% Specialized file operations
-export([get_size/1, get_file/1, set_file/2, get_file_close/1]).
-export([compress/1, uncompress/1, uuencode/1, uudecode/1]).
```

主要用于zip压缩.

```
root@yufeng-desktop:~/otp_src_R13B/lib/stdlib/src# erl
Erlang R13B (erts-5.7.1) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.1  (abort with ^G)
1> erl_ddll:loaded_drivers().
{ok,["efile","tcp_inet","udp_inet","zlib_drv",
    "ram_file_drv","tty_sl"]}
2> f(R),{ok,R}=ram_file:open("hello", []).
{ok,{file_descriptor,ram_file,#Port<0.453>}}
3> ram_file:get_size(R).
{ok,5}
4> ram_file:get_file(R).
{ok,"hello"}
5> ram_file:compress(R).
{ok,25}
6> ram_file:get_file(R).
{ok,[31,139,8,0,0,0,0,0,0,3,203,72,205,201,201,7,0,134,166,
    16,54,5,0,0,0]}
7> ram_file:uncompress(R).
{ok,5}
8> ram_file:get_file(R).
{ok,"hello"}
9> ram_file:close(R).
ok
```

CTRL+C o可以看到ram_file的port的信息：

=port:#Port<0.456>
Slot: 456
Connected: <0.63.0>
Links: <0.63.0>
Port controls linked-in driver: ram_file_drv

注意ram_file:open的文件名实际上是个数据。

另外如果file:open(xxx, [ram]) 方式打开的话，其返回的实际上是ram_file的handle.

1.125 Garbage Collection in Erlang

发表时间: 2009-04-30 关键字: garbage collection

原文地址：http://prog21.dadgum.com/16.html

Given its "soft real time" label, I expected Erlang to use some fancy incremental garbage collection approach. And indeed, such an approach exists, but it's slower than traditional GC in practice (because it touches the entire the heap, not just the live data). In reality, garbage collection in Erlang is fairly vanilla. Processes start out using a straightforward compacting collector. If a process gets large, it is automatically switched over to a generational scheme. The generational collector is simpler than in some languages, because there's no way to have an older generation pointing to data in a younger generation (remember, you can't destructively modify a list or tuple in Erlang).

The key is that garbage collection in Erlang is per process. A system may have tens of thousands of processes, using a gigabyte of memory overall, but if GC occurs in a process with a 20K heap, then the collector only touches that 20K and collection time is imperceptible. With lots of small processes, you can think of this as a truly incremental collector. But there's still a lurking worst case in Erlang: What if all of those processes run out of memory more or less in the same wall-clock moment? And there's nothing preventing an application from using one massive process (such is the case with the Wings 3D modeller).

Per-process GC allows a slick technique that can completely prevent garbage collection in some circumstances. Using spawn_opt instead of the more common spawn, you can specify the initial heap size for a process. If you know, as discovered through profiling, that a process rapidly grows up to 200K and then terminates, you can give that process an initial heap size of 200K. Data keeps getting added to the end of the heap, and then before garbage collection kicks in, the process heap is deleted and its contents are never scanned.

The other pragmatic approach to reducing the cost of garbage collection in Erlang is that lots of data is kept outside of the per-process heaps:

Binaries > 64 bytes. Large binaries are allocated in a separate heap outside the scope of a process. Binaries can't, by definition, contain pointers to other data, so they're reference counted. If there's a 50MB binary loaded, it's guaranteed never to be copied as part of garbage collection.

Data stored in ETS tables. When you look up key in an ETS table, the data associated with that key is copied into the heap for the process the request originated from. For structurally large values (say, a tuple of 500 elements) the copy from ETS table space to the process heap may become expensive,

but if there's 100MB of total data in a table, there's no risk of all that data being scanned at once by a garbage collector.

Data structure constants. This is new in Erlang.

Atom names. Atom name strings are stored in a separate data area and are not garbage collected. In Lisp, it's common for symbol names to be stored on the main heap, which adds to garbage collection time. But that also means that dynamically creating symbols in Lisp is a reasonable approach to some problems, but it's not something you want to do in Erlang.

1.126 per module constant pool 调查和使用

发表时间: 2009-05-12 关键字: per module constant pool

原文地址 http://noss.github.com/2009/04/02/constant-pool-erlang-hack.html
2009-04-02

Erlang R12B-0 added a per-module memory area for constants, the literal values in a module are stored there. Before, they were allocated on the heap every time they were referenced. This meant that some kinds of optimization that avoided literal lookup-tables became irrelevant in one go (without even recompiling the source). A great example of the kind of improvements that OTP focus on: removing speed-bumps to having beautiful code.

From the release notes of R12B-0
OTP-6850: Literal lists, tuples, and binaries are no longer constructed at run-time as they used to be, but are stored in a per-module constant pool. Literals that are used more than once are stored only once.

This is not a change to the language, only in the details of its implementation. Therefore, the implications of this change is described in the Efficiency Guide.

The erlang efficiency guide on constant pools pretty much say the same thing. But Björn Gustavsson adds this very interesting remark about unloading a module and the constant pool.
If one has very assymetric access patterns to some value, Maybe millions of times more reads than updates, and this is a measured problem, one can reach for hacks such as generating a module containing the values as literals and thus have a global configuration value that will not grow your heap unecessary.
But...

As always, remember when to optimize.

在对beam_load.c:read_literal_table(LoaderState* stp)函数打了补丁 显示出这个模块的literal数目和类型我们可以看到这样的结果：

module base64: num_literals[4]
0: tag[1], size[5], term["="]

1: tag[1], size[6], term["=="]
2: tag[3], size[131],
term[{65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,97,98,99,100,101,102,10
3: tag[3], size[1086],
term[{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-

这个是标准库base64.erl的literal.
从统计可以看出 atom本身已经不是literal， 这里大部分的literal是1，3的类型也是据说是tuple和list， 0是代表binary.

你在程序里面形如：
[1，2，3，4].
{1,2,3,4}.
<<1,2,3,4>>.
的编译器常量就会被放在模块的literal表格里面 在beam加载的时候 这些常量就创建完毕在内存里，同时调整涉及这些常量的模块的op code,让它们直接使用这些常量，而不是每次新建一个。

这样的优化对于base64或者c程序出身的人很有帮助，应为我们最擅长查表格,常数的查询时间。那么现在我们也可以在erlang下这么用了,看下base64.erl代码

```
%% arrays for character translation
%% value -> base64 (zero-based)
encode_tuple() ->
    {$A, $B, $C, $D, $E, $F, $G, $H, $I, $J, $K, $L, $M, $N,
     $O, $P, $Q, $R, $S, $T, $U, $V, $W, $X, $Y, $Z,
     $a, $b, $c, $d, $e, $f, $g, $h, $i, $j, $k, $l, $m, $n,
     $o, $p, $q, $r, $s, $t, $u, $v, $w, $x, $y, $z,
     $0, $1, $2, $3, $4, $5, $6, $7, $8, $9, $+, $/}.

%% base64 -> value (one-based)
decode_tuple() ->
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,62,-1,-1,-1,63,
     52,53,54,55,56,57,58,59,60,61,-1,-1,-1,-1,-1,-1,
     -1,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
```

```
     15,16,17,18,19,20,21,22,23,24,25,-1,-1,-1,-1,-1,
     -1,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
     41,42,43,44,45,46,47,48,49,50,51,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}.

%% accessors
b64e(X, T) ->
    element(X+1, T).

b64d(X, T) ->
    b64d_ok(element(X, T)).

-spec b64d_ok(non_neg_integer()) -> non_neg_integer().
b64d_ok(N) when N >= 0 ->
    N.
```

舒服吧！ 很多时候如果你的erl程序设计到常量的问题 有效率问题的时候 你可以动态生成一个erl文件然后用编译器编译再动态加载，最大程度的提高效率。 你的这些常量不参与GC, 无需创建，无需拷贝，这就是它的意义！

1.127 分析表达式警告的原因

发表时间: 2009-05-21 关键字: useless_building +return sys_core_fold

erlang的表达式如果不用的话，会警告的，但是有些又不警告，比较奇怪，做了下试验，再看了compiler的源码有了以下的结果：

```
root@yufeng-desktop:~# nl expr.erl
     1  -module(expr).
     2  -export([test/0]).
     3  test()->
     4      1,
     5      1.0,
     6      [],
     7      [1,2,3],
     8      <<>>,
     9      <<1,2,3>>,
    10      c:pid(0,1,2),
    11      make_ref(),
    12      atom,
    13      fun()-> ok end,
    14      open_port({spawn, "cat"}, [in, out]),
    15      "hello",
    16      "",
    17      {1,2,3},
    18      {},
    19      true,
    20      false,
    21      1.
    22
root@yufeng-desktop:~# erlc +return expr.erl
./expr.erl:4: Warning: a term is constructed, but never used
./expr.erl:5: Warning: a term is constructed, but never used
./expr.erl:7: Warning: a term is constructed, but never used
./expr.erl:8: Warning: a term is constructed, but never used
./expr.erl:9: Warning: a term is constructed, but never used
./expr.erl:11: Warning: the call to make_ref/0 has no effect
./expr.erl:13: Warning: a term is constructed, but never used
```

```
./expr.erl:15: Warning: a term is constructed, but never used
./expr.erl:17: Warning: a term is constructed, but never used
./expr.erl:18: Warning: a term is constructed, but never used
Compiler function compile:compile/3 returned:
{ok,expr,
   [{"./expr.erl",
     [{4,sys_core_fold,useless_building},
      {5,sys_core_fold,useless_building},
      {7,sys_core_fold,useless_building},
      {8,sys_core_fold,useless_building},
      {9,sys_core_fold,useless_building},
      {11,sys_core_fold,{no_effect,{erlang,make_ref,0}}},
      {13,sys_core_fold,useless_building},
      {15,sys_core_fold,useless_building},
      {17,sys_core_fold,useless_building},
      {18,sys_core_fold,useless_building}]}]}
```

看下lib/compiler/src/sys_core_fold.erl的源代码：

```
expr(#c_literal{val=Val}=L, Ctxt, _Sub) ->
    case Ctxt of
        effect ->
            case Val of
                [] ->
                    %% Keep as [] - might give slightly better code.
                    L;
                _ when is_atom(Val) ->
                    %% For cleanliness replace with void().
                    void();
                _ ->
                    %% Warn and replace with void().
                    add_warning(L, useless_building),
                    void()
            end;
        value -> L
    end;
expr(#c_cons{anno=Anno,hd=H0,tl=T0}=Cons, Ctxt, Sub) ->
    H1 = expr(H0, Ctxt, Sub),
```

```
    T1 = expr(T0, Ctxt, Sub),
    case Ctxt of
        effect ->
            add_warning(Cons, useless_building),
            expr(make_effect_seq([H1,T1], Sub), Ctxt, Sub);
        value ->
            ann_c_cons(Anno, H1, T1)
    end;
expr(#c_tuple{anno=Anno,es=Es0}=Tuple, Ctxt, Sub) ->
    Es = expr_list(Es0, Ctxt, Sub),
    case Ctxt of
        effect ->
          add_warning(Tuple, useless_building),
            expr(make_effect_seq(Es, Sub), Ctxt, Sub);
        value ->
            ann_c_tuple(Anno, Es)
    end;

expr(#c_binary{segments=Ss}=Bin0, Ctxt, Sub) ->
    %% Warn for useless building, but always build the
binary
    %% anyway to preserve a possible exception.
    case Ctxt of
        effect -> add_warning(Bin0, useless_building);
        value -> ok
    end,
    Bin1 = Bin0#c_binary{segments=bitstr_list(Ss, Sub)},
    Bin = bin_un_utf(Bin1),
    eval_binary(Bin);

expr(#c_fun{}=Fun, effect, _) ->
    %% A fun is created, but not used. Warn, and replace with the void
value.
   add_warning(Fun, useless_building),
    void();
```

得出的结论是：

如果编译器能够明确的知道你表达式的值的话，除非这几个东西 atom, [], boolean 不警告以外，其他无用的一

律警告， 而且尽可能的不产生代码。 这就解释为什么大部分的函数返回值是 原子或者 [].

1.128 ETS & SMP

发表时间: 2009-05-23 关键字: ets smp lock multicore

原文地址：http://erlang.org/pipermail/erlang-questions/2008-September/037905.html

还是老大们解释这个比较轻松 源码参见 erl_db.c

2008/9/3 Valentin Micic <>

> Is ETS utilizing the same locking policy for all table types (namely:
> public, protected or private), and if so, would it be possible to relax
> locking for protected and private access?
>

We currently don't eliminate locking on private tables, mainly to support
ets:info/1,2. We may try to eliminate locks
on private tables in a future release. (But we are not sure it is
worthwhile. Taking a lock that is not already held by another
process is relatively cheap - it becomes expensive when several processes
want to take the same lock.)

However, we take different locks depending on the operation to perform. An
operation that only reads from the table
(such as ets:lookup/2) will take a read lock, while an operation that will
update the table (such as ets:insert/2) takes
a write lock. As long as no process has a write lock, any number of
processes can take read locks. If a process takes
a write lock, it will be exclusive (i.e. no other processes can have either
read or write locks).

> We've noticed that if more than one process requires an access to the same
> ets table (in SMP environment), the system slows down considerably due to
> the locking mechanism. It is quite possible to optimize this by fronting
> such a table with a dedicated process for request serialization -- works
> better as there is always only one proccess requesting a lock. Actually... as much as this works well for one table, not so sure how would

> such an "optimization" work for a large number of tables. By relaxing (or
> not having) a locking policy for (at least) tables with a private access,
> there would be no questions about it.
>

To access an ETS table, there are actually two locks that need to be taken.

1) A lock to access the meta table, to convert the numeric table identifier to a pointer to the actual table.

2) The lock for the table itself (either a read or write lock, as described above).

In R12B-4, the locking of the meta table has been optimized. There used to be only one lock for the meta table,
but there are now different locks for different parts of the table;
therefore reducing the number of lock conflicts
for the meta table.

Therefore, if you have an application that accesses many different ETS tables, performance should be slightly better
in R12B-4.

If you have an application that accesses a single ETS table (and write to it frequently), there will still be lock
conflicts on the ETS table itself, so R12B-4 will not make much difference.

/Bjorn
--
Björn Gustavsson, Erlang/OTP, Ericsson AB

1.129 变量不是不可变

发表时间: 2009-05-30 关键字: hipe_bifs:bytearray_update

erlang的变量是不可变的 这是语法层面的事情，照理是绕不过的。但是hipe，erlang的jit模块打开了这扇门，请看：

sf bay facory 来自Facebook的Eugene Letuchy在ppt
http://www.erlang-factory.com/upload/presentations/31/EugeneLetuchy-ErlangatFacebook.pdf 里面提到：

hipe_bifs
Cheating single assignment
▪ Erlang is opinionated:
▪ Destructive assignment is hard because it should be
▪ hipe_bifs:bytearray_update() allows for destructive array assignment
▪ Necessary for aggregating Chat users' presence
▪ Don't tell anyone!

这是一个很好的绕过变量不可变的方法， 我再挖掘下google， 发现如下的文章：
http://erlang.org/pipermail/erlang-questions/2007-February/025331.html
Per Gustafsson < >
Thu Feb 22 17:41:01 CET 2007

* Previous message: [erlang-questions] Deforesting tuple updates
* Next message: [erlang-questions] Is UBF still going, or did it morph/fade?
* Messages sorted by: [date] [thread] [subject] [author]

Joel Reymont wrote:
> I don't think it works for floats or doubles. It's just bytes or
> fixnums if I remember it correctly.
>
> On Feb 22, 2007, at 3:32 PM, Daniel Luna wrote:
>
>
>
>>As long as the values in the array are simple terms, you can use
>>hipe_bifs:bytearray/2.
>

>
> --
> http://wagerlabs.com/
>
>
>
>
>
> _____
> erlang-questions mailing list
>
> http://www.erlang.org/mailman/listinfo/erlang-questions

You could use this code:

```
-module(floats).

-export([new/1,update/3,sum/1,
new2/1,update2/3,sum2/1,
test/0]).

new(N) ->
  hipe_bifs:bytearray(N*8,0).

update(Arr,N,Float) ->
  <<A1,A2,A3,A4,A5,A6,A7,A8>> = <<Float/float>>,
  Start=N*8,
  hipe_bifs:bytearray_update(Arr,Start,A1),
  hipe_bifs:bytearray_update(Arr,Start+1,A2),
  hipe_bifs:bytearray_update(Arr,Start+2,A3),
  hipe_bifs:bytearray_update(Arr,Start+3,A4),
  hipe_bifs:bytearray_update(Arr,Start+4,A5),
  hipe_bifs:bytearray_update(Arr,Start+5,A6),
  hipe_bifs:bytearray_update(Arr,Start+6,A7),
  hipe_bifs:bytearray_update(Arr,Start+7,A8).

sum(Bin) ->
  sum(Bin,0.0).
```

```
sum(<<Float/float,Rest/binary>>, Acc) ->
  sum(Rest,Float+Acc);
sum(<<>>,Acc) -> Acc.
```

Performance is not that great, about 4-5 times faster updates than
gb_trees for arrays of 100000 floats, and summing is slower.

Per

试验下：

```
root@yufeng-desktop:~# cat floats.erl
-module(floats).

-export([new/1,update/3,sum/1]).

new(N) ->
  hipe_bifs:bytearray(N*8,0).

update(Arr,N,Float) ->
  <<A1,A2,A3,A4,A5,A6,A7,A8>> = <<Float/float>>,
  Start=N*8,
  hipe_bifs:bytearray_update(Arr,Start,A1),
  hipe_bifs:bytearray_update(Arr,Start+1,A2),
  hipe_bifs:bytearray_update(Arr,Start+2,A3),
  hipe_bifs:bytearray_update(Arr,Start+3,A4),
  hipe_bifs:bytearray_update(Arr,Start+4,A5),
  hipe_bifs:bytearray_update(Arr,Start+5,A6),
  hipe_bifs:bytearray_update(Arr,Start+6,A7),
  hipe_bifs:bytearray_update(Arr,Start+7,A8).

sum(Bin) ->
  sum(Bin,0.0).

sum(<<Float/float,Rest/binary>>, Acc) ->
  sum(Rest,Float+Acc);
sum(<<>>,Acc) -> Acc.
```

```
root@yufeng-desktop:~# erl
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1>  F = floats:new(5).
<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,...>>
2> rp(F).
<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0>>
ok
3> floats:update(F, 1, 1234.00).
<<0,0,0,0,0,0,0,0,64,147,72,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,...>>
4> rp(F).
<<0,0,0,0,0,0,0,0,64,147,72,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0>>
ok
5> floats:update(F, 2, 5678.00).
<<0,0,0,0,0,0,0,0,64,147,72,0,0,0,0,0,64,182,46,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,...>>
6> rp(F).
<<0,0,0,0,0,0,0,0,64,147,72,0,0,0,0,0,64,182,46,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0>>
ok
7> floats:sum(F).
6912.0
8> 1234+ 5678=6912.
6912
9>
```

恩 没问题的， 不错。
再到源码去挖掘下
...\otp_src_R13A-0\erts\emulator\hipe\hipe_bif0.tab

```
bif hipe_bifs:bytearray/2
bif hipe_bifs:bytearray_sub/2
```

```
bif hipe_bifs:bytearray_update/3
bif hipe_bifs:bitarray/2
bif hipe_bifs:bitarray_sub/2
bif hipe_bifs:bitarray_update/3
bif hipe_bifs:array/2
bif hipe_bifs:array_length/1
bif hipe_bifs:array_sub/2
bif hipe_bifs:array_update/3
bif hipe_bifs:ref/1
bif hipe_bifs:ref_get/1
bif hipe_bifs:ref_set/2

root@yufeng-desktop:~# erl
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1>  A=hipe_bifs:array(10, []).
213598703393161713629270678049091855362742625579401574539046437786960691286075103940386
2> rp(A).
213598703393161713629270678049091855362742625579401574539046437786960691286075103940386
ok
3> hipe_bifs:array_length(A).
10
4> hipe_bifs:array_update(A, 2, 1).
213598703393161713629270678049091855362742625579401574539046437786952768469890085785292
5>
5> hipe_bifs:array_update(A, 1, atom).
213598703393161713629270678049091855362742625579401574539046437786952768469888241112101
6> hipe_bifs:array_sub(A,1).
atom
7> hipe_bifs:array_sub(A,2).
1
8> R=hipe_bifs:ref(1).
31
9> hipe_bifs:ref_get(R).
1
10> hipe_bifs:ref_set(R, 10).
175
```

11> hipe_bifs:ref_get(R).

O Yeah 这些也都OK.

还有很多hipe_bifs的导出在相应的.tab文件里面，读者自己挖掘了。这样我们就可以把状态更新， 无需通过消息 或者 ets, 大大加快效率。

附上8皇后的hipe源码。
附件下载:

- nqueens_ha.rar (1.1 KB)
- dl.javaeye.com/topics/download/3c943488-8561-3d34-8d43-50780cf8e6f7

1.130 新书 erlang programing 出炉

发表时间: 2009-06-30 关键字: erlang programing book

Erlang Programming, A Concurrent Approach to Software Development by Francesco Cesarini and Simon Thompson is planned to be published by O'Reilly before the summer. 现在来了， 有人贴在ecug上。这本书写的不错，补充了the book遗漏的不少topic, 在程序的语法的细节上很详细。

附件是书原版.
附件下载:

- Erlang.Programming.en.rar (3.5 MB)
- dl.javaeye.com/topics/download/64b17e54-860c-3b43-86cf-fd85e550e8be

1.131 R13B SMP Migration logic

发表时间: 2009-07-02 关键字: r13b smp migration logic

原文地址：http://erlang-factory.com/upload/presentations/105/KennethLundin-ErlangFactory2009London-AboutErlangOTPandMulti-coreperformanceinparticular.pdf

Migration logic

* Strive to keep the maximum number of run able processes equal on all schedulers

* Load balancing is performed by the scheduler that first reaches its max limit of reductions.
  1. Collect statistics about the maxlength of all schedulers run-queues
  2. Calculate the average limit per run-queue/prio and setup migration paths
  3. Give away jobs from schedulers over the limit, Take jobs to schedulers under the limit

* Migrations occurs when the scheduler has finished a job and goes on until the limit is reached or a new loadbalancing takes place.

* There is also work-stealing , which occurs when a scheduler gets an emty run-queue

* Running on full load or not!

If all schedulers are not fully loaded, jobs will be migrated to schedulers with lower id’ s and thus making some schedulers inactive.

这个就是为什么R13的多处理器支持强好多的原因。

1.132 Next steps with SMP and Erlang

发表时间: 2009-07-09 关键字: next steps with smp and erlang

Some known bottlenecks to address:

1. Improved handling of process table
2. Separate allocators per scheduler
3. Delayed dealloc (let the right scheduler do it)
4. Use NUMA info for grouping of schedulers
5. Separate poll sets per scheduler (IO)
6. Support Scheduler bindings, cpu_topology on Windows.
7. Dynamically linked in BIF’ s (for C-code , easier to write and more efficient than drivers)
8. Optimize Erlang applications in Erlang/OTP
9. Fine grained parallelism, language and library functions.
10. Better and more benchmarks

太Cool了 如果这些都实现的话 那性能正的好大提升哦。

1.133 yacc&lex 的erlang版本

发表时间: 2009-07-15 关键字: leex yecc

曾经花了很多时间折腾编译原理，大部分的程序不是字符串处理就是文法和语法分析，这个技术对于提高技术素质非常帮助。想起来折腾了lex yacc boost spirt lpeg lemon ragel leex yecc。 这些名词的背后是不同的语言 不同的平台 不同的思想的东西。 从上个版本erlang添加了leex以后，我非常渴望能够把它用于实际项目中去， 准备多花点时间在上面， 有兴趣的同学一起来哦。

1.134 erlang的hipe支持(高级)

发表时间: 2009-07-19 关键字: hipe

erlang的hipe相当于jit, 根据语言评测有hipe支持在纯erlang的运算上会快2-3倍，这个性能的提升对于计算密集型的应用还是比较可观的。以下是如何启用hipe:

先看下erl的版本：

```
root@nd-desktop:~# erl
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1> hipe:version().
"3.7.2
2> hipe:help_options().
HiPE Compiler Options
Boolean-valued options generally have corresponding aliases `no_...',
and can also be specified as `{Option, true}' or `{Option, false}.

General boolean options:
  [debug,load,pp_asm,pp_beam,pp_icode,pp_native,pp_rtl,time,timeout,verbose].

Non-boolean options:
  o#, where 0 =< # =< 3:
    Select optimization level (the default is 2).

Further options can be found below; use `hipe:help_option(Name)' for details.

Aliases:
  pp_all = [pp_beam,pp_icode,pp_rtl,pp_native],
  pp_sparc = pp_native,
  pp_x86 = pp_native,
  pp_amd64 = pp_native,
  pp_ppc = pp_native,
  o0,
  o1 = [x87,inline_fp,pmatch,peephole],
  o2 = [icode_ssa_const_prop,icode_ssa_copy_prop,icode_type,
        icode_inline_bifs,rtl_lcm,rtl_ssa,rtl_ssa_const_prop,spillmin_color,
```

```
        use_indexing,remove_comments,concurrent_comp,binary_opt] ++ o1,
  o3 = [icode_range,{regalloc,coalescing}] ++ o2.
ok
```

默认是emulator启用hipe支持的。

但是.erl 编译成 .beam的时候 也要采用native模式编译才可以：
erlc +native +"{hipe, [o3]}" xxx.erl

erlang的最基础的几个模块是preloaded的， 也是用erl编写的，发布版默认不是native编译的，自己可以修改下Makefile.

经过以上2个步骤 hipe支持就可以了。

缺点： hipe是第3方维护的,所以在一些未公开的特性如模块偏特化等支持上会有问题， 而且不是非常的稳定，要多测试才靠谱, 最好是100%cover过去。

进一步阅读请参考 http://www.it.uu.se/research/group/hipe/documents/hipe_manual.pdf

1.135 How fast can Erlang create processes?

发表时间: 2009-07-25 关键字: 进程 创建 速度

原文地址： http://www.lshift.net/blog/2006/09/10/how-fast-can-erlang-create-processes

Very fast indeed.

```
1> spawntest:serial_spawn(1).
3.58599e+5
```

That’s telling me that Erlang can create and tear down processes at a rate of roughly 350,000 Hz. The numbers change slightly - things slow down - if I’m running the test in parallel:

```
2> spawntest:serial_spawn(10).
3.48489e+5
3> spawntest:serial_spawn(10).
3.40288e+5

4> spawntest:serial_spawn(100).
3.35983e+5
5> spawntest:serial_spawn(100).
3.36743e+5
```

[Update: I forgot to mention earlier that the system seems to spend 50% CPU in user and 50% in system time. Very odd! I wonder what the Erlang runtime is doing to spend so much system time?]

Here’s the code for what I’m doing:

```
-module(spawntest).
-export([serial_spawn/1]).

serial_spawn(M) ->
    N = 1000000,
    NpM = N div M,
    Start = erlang:now(),
    dotimes(M, fun () -> serial_spawn(self(), NpM) end),
    dotimes(M, fun () -> receive X -> X end end),
```

```
    Stop = erlang:now(),
    (NpM * M) / time_diff(Start, Stop).

serial_spawn(Who, 0) -> Who ! done;
serial_spawn(Who, Count) ->
    spawn(fun () ->
        serial_spawn(Who, Count - 1)
      end).

dotimes(0, _) -> done;
dotimes(N, F) ->
    F(),
    dotimes(N - 1, F).

time_diff({A1,A2,A3}, {B1,B2,B3}) ->
    (B1 - A1) * 1000000 + (B2 - A2) + (B3 - A3) / 1000000.0 .
```

This is all on an Intel Pentium 4 running at 2.8GHz, with 1MB cache, on Debian linux, with erlang_11.b.0-3_all.deb.

我的实验如下：

```
root@nd-desktop:~/otp_src_R13B01# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Pentium(R) Dual-Core  CPU      E5200  @ 2.50GHz
stepping        : 6
cpu MHz         : 1200.000
cache size      : 2048 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fdiv_bug        : no
```

hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 10
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc arch_perfmon pebs bts pni dtes64 monitor ds_cpl est tm2 ssse3 cx16 xtpr pdcm lahf_lm
bogomips : 4988.06
clflush size : 64
power management:

^Croot@nd-desktop:~# erl -smp disable
Erlang R13B01 (erts-5.7.2) [source] [rq:1] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2 (abort with ^G)
1> os:getpid().
"14244"
2> spawntest:serial_spawn(1).
749026.6398814741

^Croot@nd-desktop:~/otp_src_R13B01# erl -pa /root
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2 (abort with ^G)
1> spawntest:serial_spawn(1).
402022.00990099803

结论是 beam比beam.smp要快很多，因为是所以的锁都去掉了。
我们来strace -o beam.trace.out -p 14244
root@nd-desktop:~# tail beam.trace.txt
clock_gettime(CLOCK_MONOTONIC, {3240006, 740886937}) = 0
clock_gettime(CLOCK_MONOTONIC, {3240006, 740914525}) = 0
clock_gettime(CLOCK_MONOTONIC, {3240006, 740942182}) = 0
clock_gettime(CLOCK_MONOTONIC, {3240006, 740970048}) = 0

clock_gettime(CLOCK_MONOTONIC, {3240006, 740997775}) = 0
clock_gettime(CLOCK_MONOTONIC, {3240006, 741025363}) = 0
clock_gettime(CLOCK_MONOTONIC, {3240006, 741053090}) = 0
clock_gettime(CLOCK_MONOTONIC, {3240006, 741080956}) = 0
clock_gettime(CLOCK_MONOTONIC, {3240006, 741109242}) = 0
clock_gettime(CLOCK_MONOTONIC, {3240006, 741137318}) = 0
发现大量的sys调用， user和sys的时间都很多，这是不正常的。

gdb看了下 原来是
erl_process.c:alloc_process()
{...
erts_get_emu_time(&p->started); /* 获取进程创建时间*/
...
}

昂贵的系统调用哦 去掉它。。。
重新编译再看下结果。

root@nd-desktop:~/otp_src_R13B01# bin/erl -smp disable -pa /root
Erlang R13B01 (erts-5.7.2) [source] [rq:1] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2 (abort with ^G)
1> spawntest:serial_spawn(1).
2264041.5859158505
2>

root@nd-desktop:~/otp_src_R13B01# bin/erl -pa /root
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2 (abort with ^G)
1> spawntest:serial_spawn(1).
652946.9454488944
2>

看到了吧 在beam vm下每秒最多可创建 2百万个进程 比原来快了3倍， 也就是说 创建一个进程并且销毁的开心是 0.5us 太快了， 我的机器的bogo mips是4988, 哈哈 2500个指令就搞定了 快快快。。。

结论是： 系统调用对于服务器的性能是很大的杀手！！！

1.136 How fast can Erlang send messages?

发表时间: 2009-07-26 关键字: send messages 发送消息 开销 速度

原文地址：http://www.lshift.net/blog/2006/09/10/how-fast-can-erlang-send-messages

My previous post examined Erlang’ s speed of process setup and teardown. Here I’ m looking at how quickly messages can be sent and received within a single Erlang node. Roughly speaking, I’ m seeing 3.4 million deliveries per second one-way, and 1.4 million roundtrips per second (2.8 million deliveries per second) in a ping-pong setup in the same environment as previously - a 2.8GHz Pentium 4 with 1MB cache.

Here’ s the code I’ m using - time_diff and dotimes aren’ t shown, because they’ re the same as the code in the previous post:

```
-module(ipctest).
-export([oneway/0, consumer/0, pingpong/0]).

oneway() ->
    N = 10000000,
    Pid = spawn(ipctest, consumer, []),
    Start = erlang:now(),
    dotimes(N - 1, fun () -> Pid ! message end),
    Pid ! {done, self()},
    receive ok -> ok end,
    Stop = erlang:now(),
    N / time_diff(Start, Stop).

pingpong() ->
    N = 10000000,
    Pid = spawn(ipctest, consumer, []),
    Start = erlang:now(),
    Message = {ping, self()},
    dotimes(N, fun () ->
                Pid ! Message,
                receive pong -> ok end
            end),
    Stop = erlang:now(),
```

```
    N / time_diff(Start, Stop).

consumer() ->
    receive
    message -> consumer();
    {done, Pid} -> Pid ! ok;
    {ping, Pid} ->
        Pid ! pong,
        consumer()
    end.

dotimes(0, _) -> done;
dotimes(N, F) ->
    F(),
    dotimes(N - 1, F).

time_diff({A1,A2,A3}, {B1,B2,B3}) ->
    (B1 - A1) * 1000000 + (B2 - A2) + (B3 - A3) / 1000000.0 .
```

我的实验如下：

```
root@nd-desktop:~/otp_src_R13B01# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Pentium(R) Dual-Core  CPU      E5200  @ 2.50GHz
stepping        : 6
cpu MHz         : 1200.000
cache size      : 2048 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
```

coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 10
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc arch_perfmon pebs bts pni dtes64 monitor ds_cpl est tm2 ssse3 cx16 xtpr pdcm lahf_lm
bogomips : 4988.06
clflush size : 64
power management:

root@nd-desktop:~# erl -smp disable
Erlang R13B01 (erts-5.7.2) [source] [rq:1] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2 (abort with ^G)
1> ipctest:pingpong().
2695648.1187206563
2>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
(v)ersion (k)ill (D)b-tables (d)istribution

trace了下发现大部分的系统调用是
poll([{fd=3, events=POLLIN|POLLRDNORM}, {fd=5, events=POLLIN|POLLRDNORM}, {fd=0, events=POLLIN|POLLRDNORM}], 3, 0) = 0 (Timeout)
clock_gettime(CLOCK_MONOTONIC, {3240948, 582336474}) = 0

^Croot@nd-desktop:~# erl
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2 (abort with ^G)
1> ipctest:pingpong().
709320.2697346376
2>

root@nd-desktop:~# erl -smp disable +K true

Erlang R13B01 (erts-5.7.2) [source] [rq:1] [async-threads:0] [hipe] [kernel-poll:true]

Eshell V5.7.2 (abort with ^G)
1>
1> ipctest:pingpong().
2801110.2480579205
2>

现在的系统调用是：
clock_gettime(CLOCK_MONOTONIC, {3241209, 575644283}) = 0
epoll_wait(3, {}, 256, 0) = 0
clock_gettime(CLOCK_MONOTONIC, {3241209, 575983781}) = 0
epoll_wait(3, {}, 256, 0)

速度从原来的2695648.1187206563变成现在的2801110.2480579205 有10%的提升， 仅仅是因为系统调用从poll到epoll_wait的改变 进出内核的参数少了。

这个速度已经非常理想了 也就是说消息从ping发出-》pong调度-》pong给ping回ok消息-》ping调度，整个流程才花了大概0.4us，这是相当不错的速度。。。

结论： 消息处理很快， 系统调用很费时，beam比beam.smp快很多。

附上erlang进程调度的流程：
1. 处理timer超时
2. 处理子进程退出的情况
3. 处理port_task事件，也就是port的IO事件
4. 如果没有活跃的进程 就sys_schdule阻塞在底层的IO中。
5. 根据process的优先级选出一个进程来调度。

上面epoll_wait的原因就是ping和pong的规约次数到了 让出执行权

附上gdb的断点：

/* Pid ! Message,*/
Breakpoint 2, erts_send_message (sender=0xb7c29aec, receiver=0xb7c2af8c,
receiver_locks=0xbfd79790, message=3081450490, flags=0) at beam/erl_message.c:838
838 {
(gdb) c

Continuing.

/*receive pong -> ok end*/

/* ping进程receive的时候阻塞，目前活跃的进程就一个 也就是说pong进程 */

Breakpoint 1, schedule (p=0xb7c29aec, calls=47) at beam/erl_process.c:5785

5785 {

(gdb) c

Continuing.

/* 处理完成消息 释放*/

Breakpoint 3, free_message (mp=0x81f3870) at beam/erl_message.c:53

53 {

1.137 Adding my own BIF

发表时间: 2009-07-27 关键字: adding bif

原文地址：http://www.trapexit.org/Adding_my_own_BIF

Adding my own BIF
From Erlang Community
caveat

unless you really know what you're doing, you'll be better off using a linked-in driver or a port.
steps

1. run configure
2. add your bifs to erts/emulator/beam/bif.tab

bif re:grep/2 bif re:compile/1

3. create a C file

erts/emulator/beam/erl_bif_re.c

4. add your C file to erts/emulator/<arch>/Makefile

RUN_OBJS = $(OBJDIR)/erl_bif_re.o \

5. implement your bifs by stealing bits from existing erl_bif_*.c files

```
BIF_RETTYPE re_grep_2(BIF_ALIST_2){
  Eterm result;
  result = magic_function();
  BIF_RET(result);
}
```

6. run make; make install

notes

* steps 0-3 need only be done once.

* note that if you add

bif re:grep/2

to bif.tab there should be a erl_bif_re.c that implements

BIF_RETTYPE re_grep_2(BIF_ALIST_2);

为什么要用bif呢? bif比驱动或者port的好处是 bif支持trap， 所以能够让cpu计算平均在各个进程分配，这个网络程序很重要的一个要求. 没有这个特性一个费时的操作会把整个调度器拖死，其他的进程就谈不上什么响应了。下篇文章教你如何写带trap功能的bif。

1.138 进程字典到底有多快

发表时间: 2009-07-29 关键字: process dict get put

一直以来 erlang的几本书的作者都建议不要用process dict,倒不是它的性能不好，而是因为process dict破坏了fp的变量不可变语义，所以引起了不好的印象。其实在很多场合，是很合用的。process dict 有几个好处：

1. 无锁，所以高速。
2. hash实现。
3. 内容参与gc。
4. 实现的很细致。
5. 变量可变。

以下是试验：

```
root@nd-desktop:~# cat dicttest.erl
-module(dicttest).
-export([test_put/1, test_get/1]).

test_put(N)->
    Start = erlang:now(),
    dotimes(N, fun (I) -> put(I, hello) end),
    Stop = erlang:now(),
    N / time_diff(Start, Stop).

test_get(N)->
    Start = erlang:now(),
    dotimes(N, fun (I) -> get(I) end),
    Stop = erlang:now(),
    N / time_diff(Start, Stop).

dotimes(0, _) -> done;
dotimes(N, F) ->
    F(N),
    dotimes(N - 1, F).

time_diff({A1,A2,A3}, {B1,B2,B3}) ->
    (B1 - A1) * 1000000 + (B2 - A2) + (B3 - A3) / 1000000.0 .
```

```
root@nd-desktop:~# erl -smp disable +h 9999999
Erlang R13B01 (erts-5.7.2) [source] [rq:1] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1> dicttest:test_put(1000000).
9174480.26569295
2> dicttest:test_get(1000000).
34172105.390379503
3> length(element(2, process_info(self(), dictionary))).
1000000
```

测试的速度相当的快了。 百万条级别，插入100ns, 查询40ns. 而ets的速度大概是us，差了一个数量级别。

结论：合适的场合 猛用。

1.139 ets 到底有多快

发表时间: 2009-07-30 关键字: ets update lookup insert write_concurrency

R13B的支持ets的并发写，{write_concurrency,bool()} Performance tuning. Default is false, which means that the table is optimized towards concurrent read access. An operation that mutates (writes to) the table will obtain exclusive access, blocking any concurrent access of the same table until finished. If set to true, the table is optimized towards concurrent write access. Different parts of the same table can be mutated (and read) by concurrent processes. This is achieve to some degree at the expense of single access and concurrent reader performance. Table typ ordered_set is not affected by this option in current implementation.

我们来测试下这个性能：

```
root@nd-desktop:~# cat ets_test.erl
-module(ets_test).
-export([new/0, test_insert/2, test_lookup/2, test_update/2]).

new()->
    ets:new(?MODULE, [public, {write_concurrency, true}]).

test_insert(E, N)->
    Start = erlang:now(),
    dotimes(N, fun (I) -> ets:insert(E, {I, hello}) end),
    Stop = erlang:now(),
    N / time_diff(Start, Stop).

test_lookup(E, N)->
    Start = erlang:now(),
    dotimes(N, fun (I) -> ets:lookup(E, I) end),
    Stop = erlang:now(),
    N / time_diff(Start, Stop).

test_update(E, N)->
    Start = erlang:now(),
    P = self(),
    spawn(fun ()->
                dotimes(N, fun (I) -> ets:insert(E, {I, hello}) end),
                P!done
```

```
        end),

    dotimes(N, fun (I) -> ets:insert(E, {I, hello}) end),

%% /*2个进程一起写 要比一个要快哦*/

    receive X -> X end,

    Stop = erlang:now(),
    N / time_diff(Start, Stop).

dotimes(0, _) -> done;
dotimes(N, F) ->
    F(N),
    dotimes(N - 1, F).

time_diff({A1,A2,A3}, {B1,B2,B3}) ->
    (B1 - A1) * 1000000 + (B2 - A2) + (B3 - A3) / 1000000.0 .



root@nd-desktop:~# erl -smp disable +h 9999999
Erlang R13B01 (erts-5.7.2) [source] [rq:1] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1>  E = ets_test:new().
16397
2> ets_test:test_insert(E, 1000000).
855202.2809955239
3>  ets_test:test_lookup(E, 1000000).
1584148.3776736464
4>  ets_test:test_update(E, 1000000).
1068965.36979788
5> ets:info(E, size).
1000000

^Croot@nd-desktop:~# erl +h 9999999
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]
```

```
Eshell V5.7.2  (abort with ^G)
1> E = ets_test:new().
16397
2>
2> ets_test:test_insert(E, 1000000).
673841.2793820066
3> ets_test:test_lookup(E, 1000000).
1250201.595007195
4> ets_test:test_update(E, 1000000).
896912.4685183724
5> ets:info(E, size).
1000000
```

可以看到ets的插入和查询操作基本上是在1us左右的级别，相对于process dict的几十ns, 还是差别很大的，因为ets要用2把锁， 一把保护meta table, 一把保护数据表，锁是系统的读写锁。所以这个开销是不容忽视的。

结论： 相比于无锁的数据结构， ets不是非常的快，不过对于一般的应用是够的。

1.140 Debugging for the Erlang Programmer

发表时间: 2009-08-02 关键字: debug gdb erl_debug hard_debug

原文地址：http://carpanta.dc.fi.udc.es/docs/erlang/dbg.html

这篇文章是迄今为止发现的最系统的erlang的诊断和系统获取的方法，特别是在gdb下获知程序的运行状态. 虽然文档有点过时。

(This is fairly Unix-centric; there may be interesting debug facilities
for the Windows user, but I don't know about them.)

To begin with, there is a difference between knowing in advance that
you want to debug something, and trying to find out in real-time what
is going wrong.

In the former case, you can prepare the code you are running in various
ways, which will be discussed below; in the latter, you can only use
what tools are present in the system.

We'll begin with the latter case; i.e., assuming that you have a running
Erlang system, which is misbehaving in some way, and you want to inspect
it.

* How Erlang was invoked

In many situations when debugging, it is useful to know the exact
way the Erlang system was invoked. If you add the option
"-emu_args" to the "erl" command line, it will show you the full
command line of the call to the Erlang emulator.

* Finding out what is going on in a running system

We assume here that you have an Erlang shell before you. If not, this
may be because the Erlang system is running on a different computer,
or not connected to a terminal. To gain access, read the section on
subject further down.

Apart from typing commands into the shell and have them executed, there
are two other ways of getting the attention of the system. Typing ^C
(i.e., sending SIGINT to Erlang) stops all activity [is this true?]
and presents a number of alternatives to you:

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

'a' exits the whole node. So does another ^C.
'p' shows info about pids and ports. erlang:info(procs) does the same.
'i' shows some general info about the system. erlang:info(info) does
the same.
Undocumented:
'q' like 'a'
'm' message info
'o' port info
DEBUG:
't' timer wheel
'b' bin check
'C' does abort(), i.e., dumps core

While the ^C break interaction takes place "outside" all Erlang process
activity, there is another interaction level, which is synchronous
with Erlang input: the ^G level. Typing ^G while an Erlang process
is waiting for input from you presents this prompt

```
User switch command
-->
```

and typing 'h' shows this command menu:

```
 c [nn]   - connect to job
 i [nn]   - interrupt job
 k [nn]   - kill job
 j        - list all jobs
 s        - start local shell
 r [node] - start remote shell
```

```
q        - quit erlang
? | h    - this message
```

What am I running?
erlang:info(version)
erlang:info(system_version)
init:script_id()
answer that question.

* Accessing a running system

telnet
(kent)

Distributed access

** Interaction with the shell

The usual result from a call is success and a return value; then
the shell just presents it.

If the call hangs, and you want to interrupt it and get back to the
shell, use "^G i".

If the call hangs, and you try "^G i" and it turns out that the shell
died too, use "^G s c" to start a new one.

Here is how to interpret the various messages that the shell may show
you as a result of evaluating a call:
** exited: Reason **
where Reason is whatever exit reason the process exited with.

Apart from the shell's messages, there may also occur error reports,
which either say the same thing, or give additional information (or
are just a nuisance). See the section about error reports.

When a function calls throw/1 and there is no active catch, the shell reports this as usual as:
** exited: nocatch **
but it is possible to report the term which was thrown, and this may be done in a future Erlang version.

** Post-mortem debugging of Erlang processes

Usually, when an Erlang process dies, nothing can be said about it anymore. Other processes may have a pid which refers to it, but all attempts to use it either are no-ops, cause an exit, or return something uninformative like 'undefined'.

Work is in progress on post-mortem debugging, which makes it possible to inspect the data areas of a process after it has exited.

epmd
Debugging epmd (has to be started debugged, but work is in progress to make it possible to turn on debug output when epmd is already running).

* Error reports

One classic debugging tool is trace output. If you have access to the source code of what you want to debug, insert calls which print out something interesting to the screen or a file.

In Erlang, you can use the undocumented BIF erlang:display/1 for the purpose. It writes to stderr, thereby bypassing the normal I/O mechanisms of Erlang. Using io:format is often just as convenient but may hang when done in inconvenient places, since it involves message passing between a number of processes.

Sometimes, calls to error_logger:error_report/[1,2] or similar

functions are already in place, which means that an error report in a standardized format appears in a designated place.

* Error handler

The error handler handles calls to undefined functions. It is possible to install an error handler of your own. One exists (~arndt/erlang/ehan) which tries to find alternatives based on the assumption that there is a misspelling (or forgotten exportation).

* Analysing crashes

A special case is when your system has crashed. It can do this in a number of ways, but most will leave you with an Erlang crash dump, a Unix core dump, or both. (There is also something called a Mnesia dump, which I know nothing about, so I'll not mention it again.)

A core dump is used for doing post-mortem debugging on the C code level. You need to start the debugger (gdb) and tell it the location of the Erlang system which produced the core dump. After this, you can do most of the things you can do while debugging a live system with gdb, with some exceptions.

Among the exceptions are calls to debug functions; since there is no live process anymore, there is no context to execute the functions in.

If the system seems hung, and you suspect it is looping internally, you may want it to produce both an Erlang crash dump and and a core dump. To do that, send the signal SIGUSR1 to it.

* Debug facilities within Erlang

Most Erlang terms which are really references to internal structures, such as ports, refs, funs and binaries, do not usually show much information when printed, but the I/O code for them can be made to show more: ports can be disinguished from each other, binaries are

shown with their size, funs with their arity as well as module, and
refs with their "unique" number.

** BIFs

These BIFs provide some interesting information about the system:

processes/0
erlang:ports/0
registered/0

statistics/1
run_queue
runtime
wall_clock
reductions
garbage_collection
* not documented:
context_switches
io

process_flag/2
trap_exit
error_handler
priority
* not documented:
pre_empt

process_info/2
(process_info/1 leaves out 'memory', 'binary', 'trace'; the latter two are
undocumented)

erlang:port_info/1 (undocumented)
port_info/2

erlang:info/1
info
procs

loaded
dist
system_version
getenv
os_type
os_version
version
machine
garbage_collection
instruction_counts (BEAM only)

erlang:db_all_tables/0
erlang:db_info/2

Not BIFs:

ets:all()
Table = {Name_or_number, Owner}

** minor things

The source code to the small utilities described next can be found
in d.erl in this directory.

** seq trace

** The Erlang debugger/interpreter

** pman

** Hans Nilsson's graphic process display

** appmon

** proc_lib

** process:trace

A simple-minded but useful tracer for Erlang processes:
c("/home/gandalf/arndt/erlang/tracer").
tracer:trace(Pid_to_trace).

** Klacke's top?

~klacke/erlang/top.

** Message size statistics

Work in progress.

** Instrumented Erlang

erl -instr
runs a special version of the emulator, and enables the functions
in the module 'instrument'. You can take a snapshot of the
allocated memory blocks and see what kind of blocks they are.

Mattias has written a program (for Windows) which collects and displays
the memory information graphically.

** Configuring applications

Some applications read the values of application environment variables
to adjust their behaviour. Some of those may be useful when debugging,
such as adjusting timeouts upwards, or enabling trace output.

Example: -kernel net_ticktime 3600

Some applications contain debugging calls in the source code, which
are normally turned off, but can be enabled by making a small change
to the source code and recompiling (or perhaps just define an appropriate
macro on the compiler command line).

* Report Browser - error logs - rb(3)

* Command-line options

The following command-line options are useful for debugging:

+v verbose (only active when compiled with DEBUG)
(enables the "VERBOSE" C macro)

+l auto-load tracing

+debug verbose (only active when compiled with DEBUG)
(enables the "DEBUGF" C macro)

+# sets amount of data to show when displaying BIF errors
from the JAM emulator. Almost worthless, since default
is 100.

+p progress messages while starting (enables the erl_progressf
C function)

-emu_args

On Windows, -console is useful.

* Environment variables

ERLC_EMULATOR is useful to set, when you don't understand what the 'erlc'
command is doing.

ERL_CRASH_DUMP defines the name of the file to write an Erlang crash
dump to.

* C code debugging

If you have access to the C source code, you can compile a version
of Erlang for debugging.

This causes some extra information to be emitted occasionally, and
also allows you to set variables at runtime (using gdb) which will

produce more output.

...

* Static analysis for Erlang

If you have reason to believe that there is a bug somewhere in your Erlang program, it is probably worth while to subject the source code to the available static analysis tools:

check calls for format-like functions (this tool is being written)

compile with warnings turned on (or use erl_lint)

exref can be helpful.

** Type checking

* Static analysis for C

Turn on compiler warnings.

* Dynamic analysis for C

Use purify.

* Debugging with gdb

cerl -gdb
r ...

```
#define BIF_P  A__p
#define BIF_ARG_1  A_1
#define BIF_ARG_2  A_2
#define BIF_ARG_3  A_3
```

The command "show args" is useful.

(gdb) sig 2
sends a ^C to Erlang

To invoke a function within Erlang, do, for example
(gdb) p td(A_1)

char *print_pid(Process *)
ptd(Process *, uint32) "paranoid"
BEAM: pps(Process*, uint32 *stop) "paranoid"
dbg_bt(Process*, uint32 *stop)
dis(uint32 *address, int instructions)

DEBUG:
pat(uint32 atom)
pinfo()
pp(Process *p)
ppi(uint32 process_no)
td(uint32 term)
JAM: pba(Process *, int arity)
ps(Process *, uint32 *stop)
bin_check()
check_tables()
db_bin_check()
p_slpq()

HARD_DEBUG:
check_bins
chk_sys
stack_dump
heap_dump
check_stack
check_heap
check_heap_before

* Debug-compiled Erlang

compile with these flags defined

DEBUG
HARDDEBUG
MESS_DEBUG
OPTRACE
GC_REALLOC
GC_HEAP_TRACE
GC_STACK_TRACE
OLD_HEAP_CREATION_TRACE
...

Only enabled when DEBUG:

erl +v (verbose = 1)

```
VERBOSE(erl_printf(COUT, "System halted by BIF halt(%s)\n", msg););
```

erl +debug (debug_log = 1)

```
DEBUGF(("Using vfork\n"));
```

附上erl_debug.h 里面gdb用到的常用的函数：

```
void upp(byte*, int);
void pat(Eterm);
void pinfo(void);
void pp(Process*);
void ppi(Eterm);
void pba(Process*, int);
void td(Eterm);
void ps(Process*, Eterm*);
```

1.141 Call for CN Erlounge IV !

发表时间: 2009-08-11 关键字: cn erlounge iv

原文地址：http://erlang-china.org/misc/cn-erlounge-iv.html

“Erlounge”是国外 Erlanger 对聚会的特定称谓，而“CN Erlounge”这一名称则是从 2007 年珠海的第二次会议开始，一直沿用至今。在 2008 年致力于 CN Erlounge 会务召集与组织的官方网站 ECUG.org 开通，并成功组织了精彩纷呈的 CN Erlounge III 上海站会议。如今，保持着一贯的热情与高效的 ECUG 会务组又在为我们忙碌的准备着今年的盛会 —— CN Erlounge IV 。让我们感谢他们的辛勤付出，也感谢会议历届的赞助商们。

去年 CN Erlounge III 的内容让人印象深刻，而今年 Erlang 的世界又格外精彩，不知不觉间，已经让人对于此次盛会内容又有了更高的期待。

ECUG 成立于 2007-10-14 日的 CN Erlounge II。全称为 Erlang China User Group（Erlang中国用户组）。它是一个民间团体，致力于促进 Erlang 中文社区的交流，以发展和壮大 Erlang 中国社区（了解“Erlang 中国社区的发展历程”）。

按照 ECUG 的计划，预计每年我们都会举行一次全国性的Erlang开发者大会。这个会议我们简称为 CN Erlounge。下面是历届的 CN Erlang 大会资料：

1. 2007年9月8日，CN Erlounge I，珠三角Erlang爱好者小聚。无会议资料，但酝酿了之后具有里程碑意义的CN Erlounge II。
2. 2007年10月13~14日，CN Erlounge II 在珠海召开。金山为大会主要赞助方。
3. 2008年12月20~21日，CN Erlounge III 在上海召开。盛大网络为大会主要赞助方。

今年 Erlang 中国社区人气有了明显的提高，也陆陆续续有互联网公司使用 Erlang 到他们的产品中。也有很多人开始用 Erlang 风格的并发模型（Erlang Style Concurrency）在自己熟悉的语言（如 C/C++、Java 等）中做事情，一些语言更号称自己已经实现 Erlang Style Concurrency 模型。另外，也踊跃出一批基于 Erlang Style Concurrency 模型的新语言（比如Scala）。在我们看来，Erlang是否会最终非常成功，目前言之过早，但是 Erlang 风格的并发模型（Erlang Style Concurrency）的成功，却是已经不容置疑的事实。

今年将于10月24~25日举行的 Erlang 开发者大会属于第四次 Erlang 开发者大会，简称 CN Erlounge IV。

CN Erlounge 的官方支持站点：ECUG.ORG。

CN Erlounge IV – 发起

1. 时间：2009-10-24 ~ 2008-10-25，为期2天
2. 地点：杭州（详细待定）
3. 议题： 研究、探讨、关注Erlang风格的并发模型（Erlang Style Concurrency）的技术及最新进展（不局限于Erlang语言）
4. 面向人群：对Erlang风格的并发模型有一定了解并有兴趣应用于实际工程的人。
5. 会议主持：ECUG 会务组

会议形式

1. 多数时间由交流会讲师针对某个 Topic 进行论述，其他人提问（Q&A）方式交流。
2. 留出一小段时间，安排沙龙式的对等交流机会。

会议规则

1. 会议的讲师报销来回路费和住宿（申请成为讲师）。点击这里可以查看已经确定的讲师名单。
2. 任何人可报名免费参与听讲（注册并申请参加本会议）。

注：由于场地限制，我们可能没法接受所有的与会申请，请谅解。如果名额已满，我们会回信说明。

重要时间点

1. 讲师注册及Topic征集截止日期：2009-9-15
2. 普通参会者报名截止日期： 2009-10-1
3. 讲师投稿截止日期：2009-10-10
4. 详细会议议程安排公布：2009-10-15
5. 会议日期：2009-10-24 ~ 2009-10-25

CN Erlounge IV – Topic征集

Topic范围

讲师的议题是否必须限定和 Erlang 相关呢？答案是否定的。我们需要Focus的是我们的问题域：如何高效地（包括性能和开发效率）进行分布式编程。我们都关注 Erlang 在这个方向上取得的成就，但不能也不想限制自己的眼界，Erlang 决不是我们唯一。只要你的议题和 Erlang 关注的问题域相关，和分布式、和多核时代面临的挑战相关，就没有“跑题”。Erlang 社区应该是睿智的、包容的。

投稿请发往 ECUG 会务组。

讲稿建议

1. 内容有深度，而不是泛泛而谈。忌局限于一个事实或者一个实践，但是没有任何结论。
2. 内容有一个Focus的问题域。告诉大家你要解决什么问题，它又是如何被解决的。
3. 如果能够结合一个实际的应用实践，那是最棒不过了。

1.142 ets为什么要设计成 内容不参与GC

发表时间: 2009-08-11 关键字: ets gc

原来所有的特性设计都是个妥协的过程哦。。。

Den 2006-12-23 01:16:34 skrev Yariv Sadan <>:

> This is all very interesting to me because the ets interface makes it
> seem as if an ets table is basically a dict hidden behind a
> gen_server, but ets actually has unique concurrency characteristics
> that can't be implemented in pure Erlang.

Well, they _can_, but it's not going to be nearly as efficient.
The difference will be reduced with SMP, since locking will
be required for ets. In the non-SMP version, no locking is
required, since all ets operations are in fact serialized by
the single scheduler.

To further close the gap, one can abstain from using named
public tables. In order to implement named public tables
in a pure erlang version, you either have to add a name
server, or come up with a naming scheme that maps ets
names to process names, and build atoms at runtime, or
have one single process that holds all tables (and keeps
a dictionary mapping names to table ids. The last option
will cause a terrible GC problem if there are lots of
large tables.

For an anonymous table, a gen_server using dict is not much
worse than ets, except in certain situations. For example
if you build a table very quickly, the table server will
cause very rapid memory growth due to the copying GC. This
doesn't happen with the native ets implementation, since
there is no GC of ets objects. Also, if lots of data keeps
changing in the table, GC will be a problem since much of
the table will always be scanned. If the table is mostly
static, the majority of the data will end up on the

"old_heap" and not be scanned very often.

One could say that whole tables are GC:d, though, since
they are automatically removed when the owner process dies.
What all this boils down to is that one of the main
benefits of ets is that it provides non-GC:d data storage
for Erlang processes.

BR,
Ulf W
--
Ulf Wiger

1.143 In a mnesia cluster, which node is queried?

发表时间: 2009-08-12 关键字: mnesia query node

原文地址：http://stackoverflow.com/questions/722665/in-a-mnesia-cluster-which-node-is-queried

Let's say you have a mnesia table replicated on nodes A and B. If on node C, which does not contain a copy of the table, I do mnesia:change_config(extra_db_nodes, [NodeA, NodeB]), and then on node C I do mnesia:dirty_read(user, bob) how does node C choose which node's copy of the table to execute a query on?

According to my own research answer for the question is - it will choose the most recently connected node. I will be grateful for pointing out errors if found - mnesia is a really complex system!

As Dan Gudmundsson pointed out on the mailing list algorithm of selection of the remote node to query is defined in mnesia_lib:set_remote_where_to_read/2. It is the following

```
set_remote_where_to_read(Tab, Ignore) ->
   Active = val({Tab, active_replicas}),
   Valid =
     case mnesia_recover:get_master_nodes(Tab) of
        [] ->  Active;
        Masters -> mnesia_lib:intersect(Masters, Active)
     end,
   Available = mnesia_lib:intersect(val({current, db_nodes}), Valid -- Ignore),
   DiscOnlyC = val({Tab, disc_only_copies}),
   Prefered  = Available -- DiscOnlyC,
   if
     Prefered /= [] ->
        set({Tab, where_to_read}, hd(Prefered));
     Available /= [] ->
        set({Tab, where_to_read}, hd(Available));
     true ->
        set({Tab, where_to_read}, nowhere)
   end.
```

So it gets the list of active_replicas (i.e. list of candidates), optionally shrinks the list to master nodes for the table, remove tables to be ignored (for any reason), shrinks the list to currently connected nodes and then selects in the following order:

1. First non-disc_only_copies
2. Any available node

The most important part is in fact the list of active_replicas, since it determines the order of nodes in the list of candidates.

List of active_replicas is formed by remote calls of mnesia_controller:add_active_replica/* from newly connected nodes to old nodes (i.e. one which were in the cluster before), which boils down to the function add/1 which adds the item as the head of the list.

Hence answer for the question is - it will choose the most recently connected node.

Notes: To check out the list of active replicas on the given node you can use this (dirty hack) code:

[{T,X} || {{T,active_replicas}, X} <- ets:tab2list(mnesia_gvar)].

1.144 avoiding overloading mnesia

发表时间: 2009-08-12 关键字: mnesia overload scale

There are some reoccuring themes when it comes to mnesia:

1 Mnesia handles partitioned networks poorly
2 Mnesia doesn't scale
3 Stay away from transactions

I've argued that Mnesia provides the tools to handle [1], and that most DBMSs that guarantee transaction-level consistency are hard-pressed to do better. A few offer functionality (e.g. MySQL Cluster's Arbitrator) that could be added on top of the basic functionality provided by Mnesia. DBMSs that offer 'Eventual consistency' may fare better. OTOH, one should really think about what the consistency requirements of the application are, and pick a DBMS that aims for that level.

Regarding [2], there are examples of Mnesia databases that have achieved very good scalability. It is not the best regarding writes/second to persistent storage, but as with [1], think about what your requirements are. Tcerl, just to name an example, gives much better write throughput, but requires you to explicitly flush to disk. Chances are that your data loss will be much greater if you suffer e.g. a power failure. Don't take this as criticism of tcerl, but think about what your recovery requirements are.

I am very wary about [3], mainly because I've seen many abuses of dirty operations, and observed that many who use dirty updates do it just because "it has to be fast", without having measured performance using transactions, or thought about what they give up when using dirty updates.

In some cases, transactions can even be faster than dirty. This is mainly true if you are doing batch updates on a

table with many replicas. With dirty, you will replicate once for each write, whereas a transaction will replicate all changes in the commit message. Taking a table lock will more or less eliminate the locking overhead in this case, and sticky locks can make it even cheaper.

Apart from the obvious problems with dirty writes (no concurrency protection above object-level atomicity, no guarantee that the replicas will stay consistent), there is also a bigger problem of overload.

If you have a write-intensive system, and most writes take place from one node, and are replicated to one or more others, consider that the replication requests all go through the mnesia_tm process on the remote node, while the writers perform the 'rpc' from within their own process. Thus, if you have thousands of processes writing dirty to a table, the remote mnesia_tm process(es) may well become swamped.

This doesn't happen as easily with transactions, since all processes using transactions also have to go through their local mnesia_tm.

One thing that can be done to mitigate this is to use sync_dirty. This will cause the writer to wait for the remote mnesia_tm process(es) to reply. If you have some way of limiting the number of writers, you ought to be able to protect against this kind of overload.

My personal preference is to always start with transactions, until they have proven inadequate. Most of the time, I find that they are just fine, but YMMV.

BR,
Ulf W
--
Ulf Wiger

CTO, Erlang Training & Consulting Ltd
http://www.erlang-consulting.com

目前确实存在这个问题

1.145 How to Eliminate Mnesia Overload Events

发表时间: 2009-08-13 关键字: eliminate mnesia overload events

原文地址：http://streamhacker.com/2008/12/10/how-to-eliminate-mnesia-overload-events/

If you’re using mnesia disc_copies tables and doing a lot of writes all at once, you’ve probably run into the following message

=ERROR REPORT==== 10-Dec-2008::18:07:19 ===
Mnesia(node@host): ** WARNING ** Mnesia is overloaded: {dump_log, write_threshold}

This warning event can get really annoying, especially when they start happening every second. But you can eliminate them, or at least drastically reduce their occurance.
Synchronous Writes

The first thing to do is make sure to use sync_transaction or sync_dirty. Doing synchronous writes will slow down your writes in a good way, since the functions won’t return until your record(s) have been written to the transaction log. The alternative, which is the default, is to do asynchronous writes, which can fill transaction log far faster than it gets dumped, causing the above error report.
Mnesia Application Configuration

If synchronous writes aren’t enough, the next trick is to modify 2 obscure configuration parameters. The mnesia_overload event generally occurs when the transaction log needs to be dumped, but the previous transaction log dump hasn’t finished yet. Tweaking these parameters will make the transaction log dump less often, and the disc_copies tables dump to disk more often. NOTE: these parameters must be set before mnesia is started; changing them at runtime has no effect. You can set them thru the command line or in a config file.
dc_dump_limit

This variable controls how often disc_copies tables are dumped from memory. The default value is 4, which means if the size of the log is greater than the size of table / 4, then a dump occurs. To make table dumps happen more often, increase the value. I’ve found setting this to 40 works well for my purposes.
dump_log_write_threshold

This variable defines the maximum number of writes to the transaction log before a new dump is performed. The default value is 100, so a new transaction log dump is performed after every 100

writes. If you’ re doing hundreds or thousands of writes in a short period of time, then there’ s no way mnesia can keep up. I set this value to 50000, which is a huge increase, but I have enough RAM to handle it. If you’ re worried that this high value means the transaction log will rarely get dumped when there’ s very few writes occuring, there’ s also a dump_log_time_threshold configuration variable, which by default dumps the log every 3 minutes.

How it Works

I might be wrong on the theory since I didn’ t actually write or design mnesia, but here’ s my understanding of what’ s happening. Each mnesia activity is recorded to a single transaction log. This transaction log then gets dumped to table logs, which in turn are dumped to the table file on disk. By increasing the dump_log_write_threshold, transaction log dumps happen much less often, giving each dump more time to complete before the next dump is triggered. And increasing dc_dump_limit helps ensure that the table log is also dumped to disk before the next transaction dump occurs.

1.146 erlang高级原理和应用PPT

发表时间: 2009-08-18 关键字: erlang mnesia application

培训用的 凑合看吧 主要讲分布集群以及mnesia的使用，从比较高的角度来看erlang.

同时我的blog会转移到http://blog.yufeng.info谢谢大家的关注！

附件下载:

- Erlang高级原理和应用x.rar (32.3 KB)
- dl.javaeye.com/topics/download/10c230a4-2c82-358a-abba-dcfda2cc1cc4

1.147 系统标准库的hipe支持（高级）

发表时间: 2009-08-23 关键字: hipe

前篇文章http://mryufeng.javaeye.com/blog/428845 讲述了如何启用erlang hipe支持，但是用户程序大量依赖的标准库如stdlib, kernel等默认都不是native模式的， 所以我们的程序虽然启用了hipe,但是只是部分启用了。用oprofile等工具可以看到我们的程序还是在process_main（虚拟机的代码解释在这里）里面打转。 我们来个极致的，通通hipe化。

有2个方案可以解决：
1. 在编译otp_src的时候 export ERL_COMPILE_FLAGS='+native +"{hipe, [o3]}"' 但是这个方案有个问题就是native方式是和beam的模式有关的 如beam和beam.smp它的代码是不同的，但是所有的beam又公用一套库，这样只能舍弃一个了。所以这个方案就比较麻烦。
erl
Erlang R13B01 (erts-5.7.2) [source] [64-bit] [smp:8:8] [rq:8] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2 (abort with ^G)
1> %%没问题

#erl -smp disable
<HiPE (v 3.7.2)> Warning: not loading native code for module fib: it was compiled for an incompatible runtime system; please regenerate native code for this runtime system
....
Erlang R13B01 (erts-5.7.2) [source] [64-bit] [rq:1] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2 (abort with ^G)
1>
这个也可以通过修改 alias erl=erl -smp disable 以便欺骗编译器生成单cpu模式的beam
去绕过去

2. 动态编译, 等系统运行起来以后，动态把相关的模块编译一遍，这个思路看起来最简单。

我做了个原型 证明这样是可行的。。。
cat hi.erl
-module(hi).
-export([do/0]).

do()->

```
[ turn(M, P)|| {M, P} <-code:all_loaded(), P=/=preloaded].

turn(M, P) ->
   P1 = binary_to_list(iolist_to_binary(re:replace(filename:join(filename:dirname(P),
filename:basename(P, ".beam")), "ebin", "src"))),
   L = M:module_info(),
   COpts = get_compile_options(L),

COpts1 = lists:foldr(fun({K, V}, Acc) when is_list(V) and is_integer(hd(V)) ->[{K, tr(V)}] ++ Acc ; (Skip,
Acc) -> Acc ++ [Skip] end, [], COpts),
   c:c(P1, COpts1 ++ [native, "{hipe, [o3]}"]).

tr(P)->
      binary_to_list(iolist_to_binary(re:replace(P, "/net/isildur/ldisk/daily_build/
otp_prebuild_r13b01.2009-06-07_20/", "/home/yufeng/"))).  %%%这个地方要根据实际情况调整 具体的参
看 m(lists).

get_compile_options(L) ->
   case get_compile_info(L, options) of
      {ok,Val} -> Val;
      error -> []
   end.

get_compile_info(L, Tag) ->
   case lists:keysearch(compile, 1, L) of
      {value, {compile, I}} ->
         case lists:keysearch(Tag, 1, I) of
            {value, {Tag, Val}} -> {ok,Val};
            false -> error
         end;
      false -> error
   end.

#erl -nostick
Erlang R13B01 (erts-5.7.2) [source] [64-bit] [smp:8:8] [rq:8] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1> mnesia:start().  %启动我们的应用程序
```

```
ok
2> hi:do().
[{ok,io},
{ok,erl_distribution},
{ok,edlin},
{ok,error_handler},
{ok,io_lib},
{ok,hi},
{ok,filename},
{ok,orddict},
{ok,gb_sets},
{ok,inet_db},
{ok,inet},
{ok,ordsets},
{ok,group},
{ok,gen},
{ok,erl_scan},
{ok,kernel},
{ok,erl_eval},
{ok,ets},
{ok,lists},
{ok,sets},
{ok,inet_udp},
{ok,code},
{ok,ram_file},
{ok,dict},
{ok,packages},
{ok,gen_event},
{ok,heart},
{ok,...},
{...}|...]

3> m(dict).
Module dict compiled: Date: August 23 2009, Time: 17.20
Compiler options:  [{cwd,"/home/yufeng/otp_src_R13B01/lib/stdlib/src"},
                    {outdir,"/home/yufeng/otp_src_R13B01/lib/stdlib/src/../ebin"},
                    {i,"/home/yufeng/otp_src_R13B01/lib/stdlib/src/../include"},
                    {i,"/home/yufeng/otp_src_R13B01/lib/stdlib/src/../../kernel/include"},
```

```
        debug_info,native,"{hipe, [o3]}"]
Object file: /home/yufeng/otp_src_R13B01/lib/stdlib/src/../ebin/dict.beam
```

。。。

看到了是用native模式编译的哦。。。

不过编译过程中有几个模块是有点问题， 得改进下。

1.148 研究Erlang 4000小时以后

发表时间: 2009-08-25 关键字: 研究 调优

历经2年半，花了4000小时以后，对erlang的研究有了很大的进步，从原来的兴趣， 到现在的随意的crack, 调优，改进， 指导erlang程序架构的设计，中间经历了很多。

从一个有20年历史的网络程序身上我学到很多，包括高级服务器程序的架构，调度公平性，网络事件处理，内存管理， 锁管理， SMP管理， 平台移植， 虚拟机，语言的基本构件，用户交互，调试， 诊断， 调优，工具。 也学会了使用OS提供的工具如systemtap, oprofile，内存, CPU工具来诊断，来定位问题，这个可以参考rhel的调优白皮书。

这个成熟系统带来的经验感受如同你窥视一台精密设计的机器，一环套着一环。 看似小小的系统，里面凝聚着多少片论文，多少方法改进，顺着Erlang的演化历史， 你也随着成长，其中的快乐是无法抗拒的，从中学到的东西绝不是一个库或者一个小程序能够带给你的。从中你会体会到一个大型系统是如何变成一个活生生的系统，实现者如何妥协，如何稳健的持续的改进。每一个Roadmap都值得期待。

感谢erlang的开发小组给我们带来这么好的东西，研究还将继续。。。

1.149 erlang到底能够并发发起多少系统调用

发表时间: 2009-08-26 关键字: erlang system call smp

为了测试下erlang的多smp能够每秒并发发起多少系统调用，这个关系到erlang作为网络程序在高并发下的评估。

首先crack下otp_src,因为erlang:now() 是调用了clock_gettime这个系统调用，但是遗憾的是这个now里面设计到很多mutex会导致不可预期的futex调用，所以需要做如下修改，
调用最廉价的getuid系统调用：

emacs otp_src_R13B/erts/emulator/beam/erl_bif_info.c

```
BIF_RETTYPE statistics_1(BIF_ALIST_1)
{
    Eterm res;
    Eterm* hp;

    if (BIF_ARG_1 == am_context_switches) {
        Eterm cs = erts_make_integer(erts_get_total_context_switches(), BIF_P);
        hp = HAlloc(BIF_P, 3);
        res = TUPLE2(hp, cs, SMALL_ZERO);
        BIF_RET(res);
    } else if (BIF_ARG_1 == am_ok) { /* Line 2713 */
      getuid();
      BIF_RET( am_ok);
    } else if (BIF_ARG_1 == am_garbage_collection) {
...
}
```

重新make下otp_src

```
[root@localhost ~]# cat tsmp.erl
-module(tsmp).
-export([start/1]).

loop(I, N)->
```

```
%%   erlang:now(),
%%   os:timestamp(),
    erlang:statistics(ok), %% call getuid

    case N rem 100000 of
        0 ->
            io:format("#~p:~p~n", [I, N]);
        _->
            skip
    end,

    loop(I, N + 1).

start([X])->
    N = list_to_integer(atom_to_list(X)),
    [spawn_opt(fun () -> loop(I, 0) end, [{scheduler, I}]) || I <-lists:seq(1, N)], %%未公开参数 把进程绑定到
cpu上 亲缘性
    receive
        stop ->
            ok
    after 60000 ->
            ok
    end,
    init:stop().


#otp_src_R13B02/bin/erl  -sct db  -s tsmp start 8
。。。
#7:226500000
#1:228000000
#8:152600000
#5:150200000
#4:225600000
#3:222000000
#2:224000000
#6:226400000
#7:226600000
#1:228100000
```

#4:225700000
#8:152700000
#3:222100000

对其中一个调度器线程的trace
[root@wes263 ~]# /usr/bin/strace -c -p 4667
Process 4667 attached - interrupt to quit
PANIC: attached pid 4667 exited with 0
% time seconds usecs/call calls errors syscall
------ ----------- ----------- --------- --------- ----------------
99.87 0.230051 0 3979319 getuid
0.08 0.000189 0 1924 poll
0.05 0.000116 0 1924 clock_gettime
0.00 0.000000 0 147 48 futex
------ ----------- ----------- --------- --------- ----------------
100.00 0.230356 3983314 48 total
调用序列是非常的合理的

机器配置是：
[yufeng@wes263 ~]$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 23
model name : Intel(R) Xeon(R) CPU E5450 @ 3.00GHz
stepping : 10
cpu MHz : 1998.000
cache size : 6144 KB
physical id : 0
siblings : 4
core id : 0
cpu cores : 4
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm syscall nx lm constant_tsc pni monitor ds_cpl vmx est tm2 cx16 xtpr

lahf_lm
bogomips : 5988.98
clflush size : 64
cache_alignment : 64
address sizes : 38 bits physical, 48 bits virtual
power management:

8个核心。

1分钟 erlang发起了getuid()系统调个数 ecug的8核心机器 222，100，000 × 8个核心 = 1700M 合每秒30M个系统调用

结论是：如果合理安排的话 erlang的性能是非常高的 同时可以利用到erlang的smp的巨大优势。

1.150 转：CPU密集型计算 erlang和C 大比拼

发表时间: 2009-08-31 关键字: cpu密集型计 native

原文地址：http://pseudelia.wordpress.com/2009/08/23/erlang-native-code-benchmark/

Normalerweise compiliert Erlang Bytecode (heißt das so in Erlang?). Das coole daran ist, dass man die beam files leicht auf anderen Rechnern benutzen kann. Aber die Geschwindigkeit von diesem Code hat mich nicht überzeugen können. Darum habe ich ausprobiert wie gut der native Code ist den Erlang baut.

Der Versuchsaufbau ist einfach: Ich habe eine simple rekursive Funktion geschrieben, die Fibonaccizahlen berechnet. Dann wir 5-mal Fibonacci von 40 berechnet und die Zeit gemessen. Das ganze mache ich mit nur einem Kern. Diesen Test mache ich insgesamt 3-mal. Einmal mit nativem Erlangcode, einmal mit nicht nativem Erlangcode und einmal mit einem in C geschriebenen Programm. Der Benchmark besteht aus drei Dateien:

cpu_intensive.erl:

```
-module(cpu_intensive).
-compile(export_all).

fib_test() ->
fib(40), fib(40), fib(40), fib(40), fib(40).

fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

cpu_intensive.c

```
unsigned int fib(unsigned int n) {
if (n == 0 || n == 1) {
return 1;
}
return fib(n-1) + fib(n-2);
}
```

```
int main() {
fib(40); fib(40); fib(40); fib(40); fib(40);
return 0;
}
```

Makefile:

```
all: native normal c

native:
@erlc +native cpu_intensive.erl
@echo ""
@echo "Fibonacci Erlang native code"
@time erl -noshell -s cpu_intensive fib_test -s erlang halt

normal:
@erlc cpu_intensive.erl
@echo ""
@echo "Fibonacci Erlang non-native code"
@time erl -noshell -s cpu_intensive fib_test -s erlang halt

c:
@gcc -O0 -o cpu_intensive cpu_intensive.c
@echo ""
@echo "Fibonacci written in C without optimizations"
@time ./cpu_intensive
```

Ich habe obige drei Dateien angelegt und die Makefile ausgeführt. Das Ergebnis war bei meinem Core 2 Duo 8400

```
Fibonacci Erlang native code
13,99 real        13,00 user        0,95 sys

Fibonacci Erlang non-native code
116,81 real       115,46 user       1,00 sys

Fibonacci written in C without optimizations
```

11,14 real 11,10 user 0,00 sys

Man sieht sehr schön, dass der native Erlangcode fast genauso schnell ist wie nicht optimierter C-Code. Ausserdem sieht man, dass der nicht native Erlangcode hier etwa die zehnfache Zeit braucht.

Fazit: Der native Erlangcode ist super wuschig und ich frage mich warum das +native Flag nicht in der Manpage von erlc dokumentiert ist.

1.151 答erlang静态数据查询方式

发表时间: 2009-09-05 关键字: per module constant pool

主题：erlang静态数据查询方式的一种构想 http://www.javaeye.com/topic/461367

解决这个问题有2种方式：

1. 函数匹配
2. per module constant pool

针对这个问题我做了个试验， 构建一个atom->int的查询。

```
yu-fengdemacbook-2:~ yufeng$ cat t.erl
-module(t).
-export([start/1, start/2]).


start([A1, A2])->
    start(list_to_integer(atom_to_list(A1)), A2).

start(N, Meth)->
    Start = erlang:now(),
    dotimes(N, case Meth of m->fun dict_lookup/1; f->fun fun_lookup/1 end),
    Stop = erlang:now(),
   erlang:display( N / time_diff(Start, Stop)).

dotimes(0, _) ->
     done;
dotimes(N, F) ->
    F(N),
    dotimes(N - 1, F).

time_diff({A1,A2,A3}, {B1,B2,B3}) ->
    (B1 - A1) * 1000000 + (B2 - A2) + (B3 - A3) / 1000000.0 .

dict_lookup(I) ->
    {ok, I} = dict:find(list_to_atom("x" ++ integer_to_list(I)), h1:get_dict()) .
```

```
fun_lookup(I) ->
   I = h2:lookup(list_to_atom("x" ++ integer_to_list(I))).

echo "done."yu-fengdemacbook-2:~ yufeng$ cat make_dict
#!/opt/local/bin/escript
main([A])->
N = list_to_integer(A),
L = [{list_to_atom("x" ++ integer_to_list(X)), X} || X<-lists:seq(1, N)],
D = dict:from_list(L),
io:format("-module(h1).~n-export([get_dict/0]).~nget_dict()->~n",[]),
erlang:display(D),
io:format(".~n"),
ok.
yu-fengdemacbook-2:~ yufeng$ cat make_fun
#!/opt/local/bin/escript
main([A])->
N = list_to_integer(A),
io:format("-module(h2).~n-export([lookup/1]).~n",[]),
[io:format("lookup(~p)->~p;~n",[list_to_atom("x" ++ integer_to_list(X)), X]) || X<-lists:seq(1, N)],
io:format("lookup(_)->err.~n", []),
ok.

yu-fengdemacbook-2:~ yufeng$ head h1.erl
-module(h1).
-export([get_dict/0]).
get_dict()->
{dict,100,20,32,16,100,60,{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},{{[[x10|10],[x30|30],[x50|50],[x70|70],[x90|90]],[[x11
.
yu-fengdemacbook-2:~ yufeng$ head h2.erl
-module(h2).
-export([lookup/1]).
lookup(x1)->1;
lookup(x2)->2;
lookup(x3)->3;
lookup(x4)->4;
lookup(x5)->5;
lookup(x6)->6;
```

```
lookup(x7)->7;
lookup(x8)->8;


yu-fengdemacbook-2:~ yufeng$ cat test.sh
#!/bin/bash

#OPT=+native
OPT=

echo "build $1..."
echo "make h1..."
./make_dict $1 >h1.erl
echo "make h2..."
./make_fun $1 >h2.erl
echo "compile h1..."
erlc $OPT  h1.erl
echo "compile h2..."
erlc $OPT h2.erl
echo "compile t..."
erlc $OPT t.erl

echo "running..."
echo "map..."
erl -s t start $1 m -noshell -s erlang halt
echo "fun..."
erl -s t start $1 f -noshell -s erlang halt

yu-fengdemacbook-2:~ yufeng$ ./test.sh 10000
build 10000...
make h1...
make h2...
compile h1...
compile h2...
compile t...
running...
map...
2.767323e+05
```

fun...
2.656819e+05
done.

在10000条记录的情况下 每个查询几个us， 速度不是很快.

结果发现 函数和constant pool在处理上上差不多快的。在实践中根据需要采用把。

1.152 高強度的port(Pipe)的性能測試

发表时间: 2009-09-14 关键字: ring pipe spawn port

在我的項目里面, 很多運算logic是由外部的程序來計算的 那么消息先透過pipe發到外部程序,外部程序讀到消息, 處理消息, 寫消息, erlang程序讀到消息, 這條鏈路很長,而且涉及到pipe讀寫,上下文切換,這個開銷是很大的.但是具體是多少呢?

我設計了個這樣的ring. 每個ring有N個環組成, 每個環開個port. 當ring收到個數字的時候 如果數字不為0, 那么把這個數字發到外部成程序,這個外部程序echo回來數字,收到echo回來的消息后,把數字減1,繼續傳遞.當數字減少到0的時候 銷毀整個ring.

root@nd-desktop:~/test#ulimit -n 1024 /* 注意這個數字非常重要 它影響了Erlang程序3個地方 1. epoll的句柄集大小 2. MAX_PORT 以及port的表格大小 3. open_port的時候 子進程關閉的文件句柄大小*/

root@nd-desktop:~/test# cat pipe_ring.erl

```
-module(pipe_ring).

-export([start/1]).
-export([make_relay/1, run/3]).

make_relay(Next)->
    Port = open_port({spawn, "/bin/cat"}, [in, out, {line, 128}]),
    relay_loop(Next, Port).

relay_loop(Next, Port) ->
    receive
        {Port, {data, {eol, Line}}} ->
            Next ! (list_to_integer(Line) - 1),
            relay_loop(Next, Port);
        K when is_integer(K) andalso K > 0 ->
            port_command(Port, integer_to_list(K) ++ "\n"),
            relay_loop(Next, Port);
        K when is_integer(K) andalso K =:=0 ->
            port_close(Port),
            Next ! K
end.
```

```
build_ring(K, Current, N, F) when N > 1 ->
    build_ring(K, spawn(?MODULE, make_relay, [Current]), N - 1, F);

build_ring(_, Current, _, F) ->
    F(),
    make_relay(Current).

run(N, K, Par) ->
    Parent = self(),
    Cs = [spawn(fun ()-> Parent!run1(N, K, P) end) || P<-lists:seq(1, Par)],
    [receive _-> ok end || _<-Cs].

run1(N, K, P)->
    T1 = now(),
    build_ring(K, self(), N, fun ()-> io:format("(ring~w setup time: ~ws)~n", [P, timer:now_diff(now(), T1)
/1000]), self() ! K end).

start(Args) ->
    Args1 = [N, K, Par] = [list_to_integer(atom_to_list(X)) || X<-Args],
    {Time, _} = timer:tc(?MODULE, run, Args1),
    io:format("(total run (N:~w K:~w Par:~w) ~wms ~w/s)~n", [N, K, Par, round(Time/1000),
round(K*Par*1000000/Time)]),
    halt(0).

root@nd-desktop:~/test# erl +Bd -noshell +K true -smp disable -s pipe_ring start 10 100000 8
(ring1 setup time: 0.021s)
(ring2 setup time: 0.02s)
(ring3 setup time: 0.019s)
(ring4 setup time: 0.03s)
(ring5 setup time: 0.018s)
(ring6 setup time: 0.031s)
(ring7 setup time: 0.027s)
(ring8 setup time: 0.039s)
(total run (N:10 K:100000 Par:8) 23158ms 34546/s)
```

參數的意義:

N K Par

N：ring有幾個環 每個環開一個port

K：每個環傳遞多少消息
Par: 多少ring一起跑

總的消息數是 K * Par.

我們可以看到 每秒可以處理大概 3.4W個消息 我有2個核心. 也就是說每個消息的開銷大概是 30us. 每個port的創建時間不算多, 1ms一個.

```
root@nd-desktop:~/test# dstat
----total-cpu-usage---- -dsk/total- -net/total- ---paging-- ---system--
usr sys idl wai hiq siq| read  writ| recv  send|  in   out | int   csw
33  18  50   0   0   1|   0     0 | 438B 2172B|   0     0 |5329    33k
42  11  48   0   0   0|   0     0 | 212B  404B|   0     0 |5729    58k
41  11  49   0   0   0|   0     0 | 244B 1822B|   0     0 |5540    59k
40  11  49   0   0   0|   0     0 | 304B  404B|   0     0 |4970    60k
```

注意上面的csw 達到6W每秒.

```
root@nd-desktop:~/test# pstree
├─sshd─┬─sshd─┬─bash───pstree
    |      |      └─bash───man───pager
    |      ├─sshd───bash─┬─beam─┬─80*[cat]
    |      |             |      └─{beam}
    |      |             └─emacs
    |      ├─sshd───bash───emacs
    |      └─sshd───bash───nmon
```

我們運行了80個echo程序(/bin/cat)

讀者有興趣的話可以用systemtap 詳細了解 pipe的讀寫花費,以及context_switch情況, 具體腳本可以向我索要.

```
root@nd-desktop:~# cat /proc/cpuinfo
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Pentium(R) Dual-Core  CPU      E5200  @ 2.50GHz
stepping        : 6
```

cpu MHz : 1200.000
cache size : 2048 KB
physical id : 0
siblings : 2
core id : 1
cpu cores : 2
apicid : 1
initial apicid : 1
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 10
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc arch_perfmon pebs bts pni dtes64 monitor ds_cpl em
bogomips : 4987.44
clflush size : 64
power management:

結論是: 用port的這種架構的開銷是可以接受的.

erlang之旅玩的开心

erlang深度分析

作者: mryufeng
http://mryufeng.javaeye.com