

- [Home](#)
- [欢迎加盟](#)
- [关于我们](#)
- [热门文章](#)

[搜索技术博客一淘宝](#)

[关注技术](#) [关注搜索](#) [关注淘宝](#)

- [graph database](#)
- [性能优化](#)
- [搜索引擎](#)
- [前端技术](#)
- [数据挖掘](#)
- [导购搜索](#)
- [搜索动态](#)
- [分布式技术](#)
- [其他](#)

14
三

[X86-64寄存器和栈帧](#)

[jiye](#)

概要

说到x86-64，总不免要说说AMD的牛逼，x86-64是x86系列中集大成者，继承了向后兼容的优良传统，最早由AMD公司提出，代号AMD64；正是由于能向后兼容，AMD公司打了一场漂亮翻身战。导致Intel不得不转而生产兼容AMD64的CPU。这是IT行业以弱胜强的经典战役。不过，大家为了名称延续性，更习惯称这种系统结构为x86-64

X86-64在向后兼容的同时，更主要的是注入了全新的特性，特别的：x86-64有两种工作模式，32位OS既可以跑在传统模式中，把CPU当成i386来用；又可以跑在64位的兼容模式中，更加神奇的是，可以在32位的OS上跑64位的应用程序。有这种好事，用户肯定买账啦，

值得一提的是，X86-64开创了编译器的新纪元，在之前的时代里，Intel CPU的晶体管数量一直以摩尔定律在指数发展，各种新奇功能层出不穷，比如：条件数据传送指令cmovg，SSE指令等。但是GCC只能保守地假设目标机器的CPU是1985年的i386，额。。。这样编译出来的代码效率可想而知，虽然GCC额外提供了大量优化选项，但是这对应用程序开发者提出了很高的要求，会者寥寥。X86-64的出现，给GCC提供了一个绝好的机会，在新的x86-64机器上，放弃保守的假设，进而充分利用x86-64的各种特性，比如：在过程调用中，通过寄存器来传递参数，而不是传统的堆栈。又如：尽量使用条件传送指令，而不是控制跳转指令

寄存器简介

先明确一点，本文关注的是通用寄存器（后简称寄存器）。既然是通用的，使用并没有限制；后面介绍寄存器使用规则或者惯例，只是GCC（G++）遵守的规则。因为我们想对GCC编译的C（C++）程序进行分析，所以了解这些规则就很有帮助。

在体系结构教科书中，寄存器通常被说成寄存器文件，其实就是CPU上的一块存储区域，不过更喜欢使用标识符来表示，而不是地址而已。

X86-64中，所有寄存器都是64位，相对32位的x86来说，标识符发生了变化，比如：从原来的%ebp变成了%rbp。为了向后兼容性，%ebp依然可以使用，不过指向了%rbp的低32位。

X86-64寄存器的变化，不仅体现在位数上，更加体现在寄存器数量上。新增加寄存器%r8到%r15。加上x86的原有8个，一共16个寄存器。

刚刚说到，寄存器集成在CPU上，存取速度比存储器快好几个数量级，寄存器多了，GCC就可以更多的使用寄存器，替换之前的存储器堆栈使用，从而大大提升性能。

让寄存器为己所用，就得了解它们的用途，这些用途都涉及函数调用，X86-64有16个64位寄存器，分别是：%rax, %rbx, %rcx, %rdx, %esi, %edi, %rbp, %rsp, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15。其中：

- %rax 作为函数返回值使用。
- %rsp 栈指针寄存器，指向栈顶
- %rdi, %rsi, %rdx, %rcx, %r8, %r9 用作函数参数，依次对应第1参数，第2参数。。。
- %rbx, %rbp, %r12, %r13, %14, %15 用作数据存储器，遵循被调用者使用规则，简单说就是随使用，调用子函数之前要备份它，以防他被修改
- %r10, %r11 用作数据存储器，遵循调用者使用规则，简单说就是使用之前要先保存原值

63	31	0
%rax	%eax	返回值
%rbx	%ebx	被调用者保护
%rcx	%ecx	第四个参数
%rdx	%edx	第三个参数
%rsi	%esi	第二个参数
%rdi	%edi	第一个参数
%rbp	%ebp	被调用者保护
%rsp	%esp	堆栈指针
%r8	%r8d	第五个参数
%r9	%r9d	第六个参数
%r10	%r10d	调用者保护
%r11	%r11d	调用者保护
%r12	%r12d	被调用者保护
%r13	%r13d	被调用者保护
%r14	%r14d	被调用者保护
%r15	%r15d	被调用者保护

栈帧

栈帧结构

C语言属于面向过程语言，他最大特点就是把一个程序分解成若干过程（函数），比如：入口函数是main，然后调用各个子函数。在对应机器语言中，GCC把过程转化成栈帧（frame），简单的说，每个栈帧对应一个过程。X86-32典型栈帧结构中，由%ebp指向栈帧开始，%esp指向栈顶。



函数进入和返回

函数的进入和退出，通过指令call和ret来完成，给一个例子

```

1  #include
2  #include </code>
3
4  int foo ( int x )
5  {
6  int array[] = {1,3,5};
7  return array[x];
8  }      /* ----- end of function foo ----- */
9
10 int main ( int argc, char *argv[] )
11 {
12 int i = 1;
13 int j = foo(i);
14 fprintf(stdout, "i=%d,j=%d\n", i, j);
15 return EXIT_SUCCESS;
16 }      /* ----- end of function main ----- */

```

命令行中调用gcc，生成汇编语言：

```
1 | Shell > gcc -S -o test.s test.c
```



Main函数第40行的指令Call foo其实干了两件事情：

- Pushl %rip //保存下一条指令（第41行的代码地址）的地址，用于函数返回继续执行
- Jmp foo //跳转到函数foo

Foo函数第19行的指令ret 相当于：

- popl %rip //恢复指令指针寄存器

栈帧的建立和撤销

还是上一个例子，看看栈帧如何建立和撤销

说题外话，以”点”做为前缀的指令都是用来指导汇编器的命令。无意于程序理解，统统忽视之，比如第31行。

栈帧中，最重要的是帧指针%ebp和栈指针%esp，有了这两个指针，我们就可以刻画一个完整的栈帧
函数main的第30~32行，描述了如何保存上一个栈帧的帧指针，并设置当前的指针。

第49行的leave指令相当于：

Movq %rbp %rsp //撤销栈空间，回滚%rsp

Popq %rbp //恢复上一个栈帧的%rbp

同一件事情会有很多的做法，GCC会综合考虑，并作出选择。选择leave指令，极有可能因为该指令需要存储空间少，需要时钟周期也少。

你会发现，在所有的函数中，几乎都是同样的套路，

我们通过gdb观察一下进入foo函数之前main的栈帧，进入foo函数的栈帧，退出foo的栈帧情况

```

1 | Shell> gcc -g -o test test.c
2 | Shell> gdb --args test
3 | Gdb > break main
4 | Gdb > run

```

进入foo函数之前：

```

(gdb) s
13      int j = foo(i);
(gdb) info registers rbp
rbp      0x7fff3e77d6f0    0x7fff3e77d6f0
(gdb) info registers rsp
rsp      0x7fff3e77d6d0    0x7fff3e77d6d0

```

你会发现 $rbp-rsp=0\times 20$ ，这个是由代码第11行造成的。
进入foo函数的栈帧：

```

(gdb) info registers rsp
rsp      0x7fff3e77d6c0    0x7fff3e77d6c0
(gdb) info registers rbp
rbp      0x7fff3e77d6c0    0x7fff3e77d6c0
(gdb) x 0x7fff3e77d6c0-16
0x7fff3e77d6b0: 0x00000001
(gdb) x 0x7fff3e77d6c0-12
0x7fff3e77d6b4: 0x00000003
(gdb) x 0x7fff3e77d6c0-8
0x7fff3e77d6b8: 0x00000005

```

回到main函数的栈帧，rbp和rsp恢复成进入foo之前的状态，就好像什么都没发生一样。

```

(gdb) info registers rbp
rbp      0x7fff3e77d6f0    0x7fff3e77d6f0
(gdb) info registers rsp
rsp      0x7fff3e77d6d0    0x7fff3e77d6d0

```

可有可无的帧指针

你刚刚搞清楚帧指针，是不是很期待要马上派上用场，这样你可能要大失所望，因为大部分的程序，都加了优化编译选项：`-O2`，这几乎是普遍的选择。在这种优化级别，甚至更低的优化级别`-O1`，都已经去除了帧指针，也就是`%ebp`中再也不是保存帧指针，而且另作他途。

在x86-32时代，当前栈帧总是从保存`%ebp`开始，空间由运行时决定，通过不断push和pop改变当前栈帧空间；x86-64开始，GCC有了新的选择，优化编译选项`-O1`，可以让GCC不再使用栈帧指针，下面引用gcc manual 一段话：

```

1 | -O also turns on -fomit-frame-pointer on machines where doing so does not in

```

这样一来，所有空间在函数开始处就预分配好，不需要栈帧指针；通过`%rsp`的偏移就可以访问所有的局部变量。

说了这么多，还是看看例子吧。同一个例子，加上`-O1`选项：

```

1 | Shell>: gcc -O1 -S -o test.s test.c

```

```

21 main:
22 .LFB26:
23   subq $0, %rsp
24   .LCL0:
25   movl %edi, %edi
26   call foo
27   movl %eax, %eax
28   movl $1, %eax
29   movl $.LCL0, %edi
30   movq stdout(%rip), %rdi
31   movl %eax, %eax
32   call fprintf
33   movl %eax, %eax
34   addq $0, %rsp
35   ret

5 foo:
6 .LFB27:
7   movl $1, -24(%rsp)
8   movl $1, -20(%rsp)
9   movl $1, -16(%rsp)
10  movsq %rdi, %rdi
11  movl -16(%rsp, %rdi, 4), %eax
12  ret

```

分析main函数，GCC分析发现栈帧只需要8个字节，于是进入main之后第一条指令就分配了空间(第23行)：

```
1 | Subq $8, %rsp
```

然后在返回上一栈帧之前，回收了空间（第34行）：

```
1 | Addq $8, %rsp
```

等等，为啥main函数中并没有对分配空间的引用呢？这是因为GCC考虑到栈帧对齐需求，故意做出的安排。

再来看foo函数，这里你可以看到%rsp是如何引用栈空间的。

等等，不是需要先预分配空间吗？这里为啥没有预分配，直接引用栈顶之外的地址？这就要涉及x86-64引入的牛逼特性了。

访问栈顶之外

通过readelf查看可执行程序的头信息：

```
try readelf --help for more information.
[admin@search042043.sqa.cm4 test]$ readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  EXEC (Executable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x400400
  Start of program headers:              64 (bytes into file)
  Start of section headers:             2856 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   64 (bytes)
  Size of program headers:               56 (bytes)
  Number of program headers:              8
  Size of section headers:               64 (bytes)
  Number of section headers:             29
  Section header string table index:     26
```

红色区域部分指出了x86-64遵循ABI规则的版本，它定义了一些规范，遵循ABI的具体实现应该满足这些规范，其中，他就规定了程序可以使用栈顶之外128字节的地址。

这说起来很简单，具体实现可有大学问，这超出了本文的范围，具体大家参考虚拟存储器。别的不提，接着上例，我们发现GCC利用了这个特性，干脆就不给foo函数分配栈帧空间了，而是直接使用栈帧之外的空间。@恨少说这就相当于内联函数呗，我要说：这就是编译优化的力量。

寄存器保存惯例

过程调用中，调用者栈帧需要寄存器暂存数据，被调用者栈帧也需要寄存器暂存数据。如果调用者使用了%rbx，那被调用者就需要在使用之前把%rbx保存起来，然后在返回调用者栈帧之前，恢复%rbx。遵循该使用规则的寄存器就是被调用者保存寄存器，对于调用者来说，%rbx就是非易失的。

反过来，调用者使用%r10存储局部变量，为了能在子函数调用后还能使用%r10，调用者把%r10先保存起来，然后在子函数返回之后，再恢复%r10。遵循该使用规则的寄存器就是调用者保存寄存器，对于调用者来说，%r10就是易失的，举个例子：

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
```



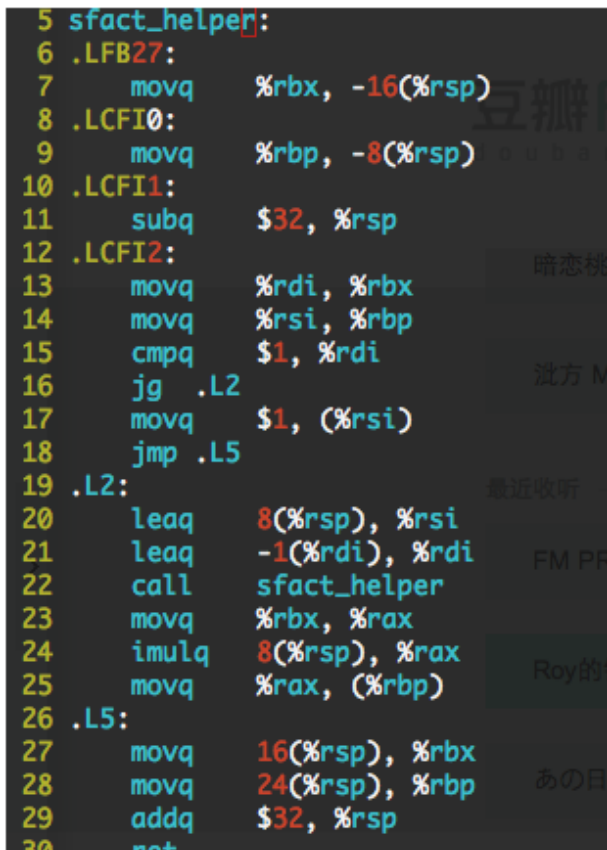
```

3
4 void sfact_helper ( long int x, long int * resultp)
5 {
6     if (x<=1)
7         *resultp = 1;
8     else {
9         long int nresult;
10        sfact_helper(x-1,&nresult);
11        *resultp = x * nresult;
12    }
13 } /* ----- end of function foo ----- */
14
15 long int
16 sfact ( long int x )
17 {
18     long int result;
19     sfact_helper(x, &result);
20     return result;
21 } /* ----- end of function sfact ----- */
22
23 int
24 main ( int argc, char *argv[] )
25 {
26     int sum = sfact(10);
27     fprintf(stdout, "sum=%d\n", sum);
28     return EXIT_SUCCESS;
29 } /* ----- end of function main ----- */

```

命令行中调用gcc，生成汇编语言：

```
1 | Shell>: gcc -O1 -S -o test2.s test2.c
```



```

5 sfact_helper:
6 .LFB27:
7     movq    %rbx, -16(%rsp)
8 .LCFI0:
9     movq    %rbp, -8(%rsp)
10 .LCFI1:
11     subq    $32, %rsp
12 .LCFI2:
13     movq    %rdi, %rbx
14     movq    %rsi, %rbp
15     cmpq    $1, %rdi
16     jg      .L2
17     movq    $1, (%rsi)
18     jmp     .L5
19 .L2:
20     leaq    8(%rsp), %rsi
21     leaq    -1(%rdi), %rdi
22     call    sfact_helper
23     movq    %rbx, %rax
24     imulq   8(%rsp), %rax
25     movq    %rax, (%rbp)
26 .L5:
27     movq    16(%rsp), %rbx
28     movq    24(%rsp), %rbp
29     addq    $32, %rsp
30     ret

```

在函数sfact_helper中，用到了寄存器%rbx和%rbp，在覆盖之前，GCC选择了先保存他们的值，代码6~9说明该行为。在函数返回之前，GCC依次恢复了他们，就如代码27-28展示的那样。

看这段代码你可能会困惑？为什么%rbx在函数进入的时候，指向的是-16（%rsp），而在退出的时候，变成了32（%rsp）。上文不是介绍过一个重要的特性吗？访问栈帧之外的空间，这是GCC不用先分配空

间再使用；而是先使用栈空间，然后在适当的时机分配。第11行代码展示了空间分配，之后栈指针发生变化，所以同一个地址的引用偏移也相应做出调整。

参数传递

X86时代，参数传递是通过入栈实现的，相对CPU来说，存储器访问太慢；这样函数调用的效率就不高，在x86-64时代，寄存器数量多了，GCC就可以利用多达6个寄存器来存储参数，多于6个的参数，依然还是通过入栈实现。了解这些对我们写代码很有帮助，起码有两点启示：

- 尽量使用6个以下的参数列表，不要让GCC为难啊。
- 传递大对象，尽量使用指针或者引用，鉴于寄存器只有64位，而且只能存储整形数值，寄存器存不下大对象

让我们具体看看参数是如何传递的：


```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int foo ( int arg1, int arg2, int arg3, int arg4, int arg5, int arg6, int a
5  {
6      int array[] = {100,200,300,400,500,600,700};
7      int sum = array[arg1] + array[arg7];
8      return sum;
9  }      /* ----- end of function foo ----- */
10
11      int
12 main ( int argc, char *argv[] )
13 {
14     int i = 1;
15     int j = foo(0, 1, 2, 3, 4, 5, 6);
16     fprintf(stdout, "i=%d,j=%d\n", i, j);
17     return EXIT_SUCCESS;
18 }      /* ----- end of function main ----- */

```

命令行中调用gcc，生成汇编语言：

```
1 | Shell> gcc -O1 -S -o test1.s test1.c
```



The left screenshot shows the assembly for the main function. It starts with a prologue (pushing %rbp, setting %rsp) and then pushes arguments 0 through 6 onto the stack. It then calls the foo function. The right screenshot shows the assembly for the foo function. It starts with a prologue (pushing %rbp, setting %rsp) and then pushes arguments 0 through 6 onto the stack. It then calculates the index for the array access: `movl $0, %rdi` (index 0), `movl $1, %rdi` (index 1), `movl $2, %rdi` (index 2), `movl $3, %rdi` (index 3), `movl $4, %rdi` (index 4), `movl $5, %rdi` (index 5), `movl $6, %rdi` (index 6). It then calculates the address of the array element: `movl $0, %rdi` (index 0), `movl $1, %rdi` (index 1), `movl $2, %rdi` (index 2), `movl $3, %rdi` (index 3), `movl $4, %rdi` (index 4), `movl $5, %rdi` (index 5), `movl $6, %rdi` (index 6). It then calculates the address of the array element: `movl $0, %rdi` (index 0), `movl $1, %rdi` (index 1), `movl $2, %rdi` (index 2), `movl $3, %rdi` (index 3), `movl $4, %rdi` (index 4), `movl $5, %rdi` (index 5), `movl $6, %rdi` (index 6).

Main函数中，代码31~37准备函数foo的参数，从参数7开始，存储在栈上，%rsp指向的位置；参数6存储在寄存器%r9d；参数5存储在寄存器%r8d；参数4对应于%ecx；参数3对应于%edx；参数2对应于%esi；参数1对应于%edi。

Foo函数中，代码14~15，分别取出参数7和参数1，参与运算。这里数组引用，用到了最经典的寻址方式，`-40(%rsp, %rdi, 4) = %rsp + %rdi * 4 + (-40)`；其中%rsp用作数组基地址；%rdi用作了数组的下标；数字4表示sizeof(int)=4。

结构体传参

应@桂南要求，再加一节，相信大家也很想知道结构体是如何存储，如何引用的，如果作为参数，会如何传递，如果作为返回值，又会如何返回。

看下面的例子：

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct demo_s {
5      char var8;
6      int  var32;
7      long var64;
8  };
9
10 struct demo_s foo (struct demo_s d)
11 {
12     d.var8=8;
13     d.var32=32;
14     d.var64=64;
15     return d;
16 }      /* ----- end of function foo ----- */
17
18     int
19 main ( int argc, char *argv[] )
20 {
21     struct demo_s d, result;
22     result = foo (d);
23     fprintf(stdout, "demo: %d, %d, %ld\n", result.var8, result.var32, result.var64);
24     return EXIT_SUCCESS;
25 }      /* ----- end of function main ----- */

```

我们缺省编译选项，加了优化编译的选项可以留给大家思考。

```
1 | Shell>gcc -S -o test.s test.c
```

```

34 main:
35 .LFB@:
36     pushq   %rbp
37 .LCFI2:
38     movq    %rbp, %rbp
39 .LCFI3:
40     subq    $04, %rsp
41 .LCFI4:
42     movl    %rdi, -32(%rbp) // 结构体拆成两个部分，放两个寄存器
43     movq    %rsi, -40(%rbp) // 结构体拆成两个部分，放两个寄存器
44     movq    -40(%rbp), %rdi // var8和var32组合在一起作为参数
45     movq    -40(%rbp), %rsi // var64作为参数
46     call    foo
47     movq    %rax, -64(%rbp) // 返回值1
48     movq    %rdx, -56(%rbp) // 返回值2
49     movq    -64(%rbp), %rax
50     movq    %rax, -32(%rbp)
51     leave

```

```

5 foo:
6 .LFB@:
7     pushq   %rbp
8 .LCFI0:
9     movq    %rbp, %rbp
10 .LCFI1:
11     movl    %rdi, %rax
12     movq    %rsi, %rdx
13     movq    %rax, -32(%rbp) // 参数: 包含了 var8和 var32
14     movq    %rdx, -40(%rbp) // 参数: 包含了 var64
15     movb    $8, -32(%rbp) // 写 demo_s.var8
16     movl    $32, -40(%rbp) // 写 demo_s.var32
17     movq    $64, -48(%rbp) // 写 demo_s.var64
18     movq    -32(%rbp), %rax
19     movq    %rax, -16(%rbp)
20     movq    -40(%rbp), %rdx
21     movq    %rdx, -16(%rbp) // 返回值1: var8和 var32
22     movq    -16(%rbp), %rax // 返回值1: var8和 var32
23     movq    -16(%rbp), %rdx // 返回值2: var64
24     leave

```

上面的代码加了一些注释，方便大家理解，

问题1：结构体如何传递？它被分成了两个部分，var8和var32合并成8个字节的大小，放在寄存器%rdi中，var64放在寄存器的%rsi中。也就是结构体分解了。

问题2：结构体如何存储？注意看foo函数的第15~17行注意到，结构体的引用变成了一个偏移量访问。这和数组很像，只不过他的元素大小可变。

问题3：结构体如何返回，原本%rax充当了返回值的角色，现在添加了返回值2：%rdx。同样，GCC用两个寄存器来表示结构体。

恩，即使在缺省情况下，GCC依然是想尽办法使用寄存器。随着结构变的越来越大，寄存器不够用了，那就只能使用栈了。

总结

了解寄存器和栈帧的关系，对于gdb调试很有帮助；过些日子，一定找个合适的例子和大家分享一下。

参考

1. 深入理解计算机体系结构
2. x86系列汇编语言程序设计

Filed under - [性能优化](#) [8 Comments](#) so far.

标签: [Linux](#)

相关文章

- [autoconf AC_ARG_WITH, AC_CACHE_CHECK, AC_TRY_LINK宏学习](#)
- [PHP安全之慎用preg_replace的/e修饰符](#)
- [当cpu飙升时，找出php中可能有问题的代码行](#)
- [如何在Hadoop集群运行JNI程序](#)

8 Responses



1. 枯の灵 on 14 三 2013

Nice~

[回复](#)



2. 名字 on 15 三 2013

好文一篇，不过作为淘宝搜索技术官方博客，建议在截图细节部分严格把关

[回复](#)



3. henry on 20 三 2013

好文章值得收藏

[回复](#)



4. zhanglistar on 18 四 2013

nice!

[回复](#)



5. [jiangwenfeng](#) on 01 五 2013

%rdi, %rsi 应该是 %esi和%esi吧?

[回复](#)



6. ganbo on 24 五 2013

非常建议您再斟酌一下关于amd64 调用约定的部分，据我所知rbx是要求被调用者保存。

[回复](#)



7. silks on 03 一 2014

非常非常好

[回复](#)



8. iesmdx on 26 三 2014

作者有没有研究过GCC函数局部变量的分配以及函数形参保存?

[回复](#)

留言

Name (required)

Email (required)

URL

Comment

留言

订阅

[Subscribe to feed](#)
[get the latest updates!](#)

 搜索

• 最近更新

- [C++插件中使用静态指针变量引起的内存泄露问题](#)
- [Python:\[Errno 32\] Broken pipe 导致线程crash解决方法](#)
- [neo4j 底层存储结构分析\(8\)](#)
- [neo4j 底层存储结构分析\(7\)](#)
- [neo4j 底层存储结构分析\(6\)](#)
- [neo4j 底层存储结构分析\(5\)](#)
- [neo4j 底层存储结构分析\(4\)](#)
- [neo4j 底层存储结构分析\(3\)](#)

• 标签云

[A/B Test](#) [BTS](#) [C++](#) [Cassandra](#) [checkstyle](#) [CppUnit](#) [Google](#) [GPU](#) [graph database](#) [Hadoop](#)
[hbase](#) [java](#) [JNI](#) [JUnit](#) [Leslie Lamport](#) [Linux](#) [mangodb](#) [MapReduce](#) [Multivariate Test](#) [MySQL](#) [neo4j](#)
[nginx](#) [NoSQL](#) [paxos](#) [PHP](#) [Protocol Buffers](#) [redis](#) [Samba](#) [Scrapy](#) [Streaming](#) [zookeeper](#) [一](#)
[淘](#) [分布式抓取](#) [分布式锁](#) [前端](#) [单元测试](#) [导购搜索](#) [性能](#) [性能优化](#) [搜索产品优化](#) [搜索引擎](#) [算法](#) [网络](#)
[爬虫](#) [装载](#) [链接](#)

• 近期评论

- hello发表在《[HQueue：基于HBase的消息队列](#)》
- [good90](#)发表在《[redis超时问题分析](#)》
- [good90](#)发表在《[redis超时问题分析](#)》
- [iteblog](#)发表在《[storm简介](#)》
- 梁亚飞发表在《[轻松写出优雅的Java代码之CheckStyle](#)》
- [TargetNull](#)发表在《[字符串匹配那些事（一）](#)》
- chenjianjun发表在《[Hadoop的那些事儿](#)》
- [TargetNull](#)发表在《[字符串匹配那些事（一）](#)》

• 友情链接

- [淘宝DBA团队](#)
- [淘宝UED团队](#)
- [淘宝招聘](#)

- [淘宝数据平台团队](#)
- [淘宝核心系统团队](#)
- [淘宝质量保障团队](#)
- [量子统计官方博客](#)

• 功能

- [注册](#)
- [登录](#)
- [文章RSS](#)
- [评论RSS](#)

搜索技术博客—淘宝 © 2014. All rights reserved.
A quality product by [KreativeThemes](#)