Linus：利用二级指针删除单向链表

2013年2月4日 Leo 发表评论 阅读评论 54,026 人阅读

感谢网友full_of_bull投递此文（注：此文最初发表在这个这里，我对原文后半段修改了许多，并加入了插图）

Linus大婶在slashdot上回答一些编程爱好者的提问，其中一个人问他什么样的代码是他所喜好的，大婶表述了自己一些观点之后，举了一个指针的例子，解释了什么才是core low-level coding。

下面是Linus的教学原文及翻译——

> “At the opposite end of the spectrum, I actually wish more people understood the really core low-level kind of coding. Not big, complex stuff like the lockless name lookup, but simply good use of pointers-to-pointers etc. For example, I’ve seen too many people who delete a singly-linked list entry by keeping track of the “prev” entry, and then to delete the entry, doing something like。（在这段话的最后，我实际上希望更多的人了解什么是真正的核心底层代码。这并不像无锁文件名查询（注：可能是git源码里的设计）那样庞大、复杂，只是仅仅像诸如使用二级指针那样简单的技术。例如，我见过很多人在删除一个单项链表的时候，维护了一个”prev”表项指针，然后删除当前表项，就像这样）”

```
if (prev)
    prev->next = entry->next;
else
    list_head = entry->next;
```

> and whenever I see code like that, I just go “This person doesn’t understand pointers”. And it’s sadly quite common.（当我看到这样的代码时，我就会想“这个人不了解指针”。令人难过的是这太常见了。）
>
> People who understand pointers just use a “pointer to the entry pointer”, and initialize that with the address of the list_head. And then as they traverse the list, they can remove the entry without using any conditionals, by just doing a “*pp = entry->next”. （了解指针的人会使用链表头的地址来初始化一个“指向节点指针的指针”。当遍历链表的时候，可以不用任何条件判断（注：指prev是否为链表头）就能移除某个节点，只要写）

```
*pp = entry->next
```

> So there's lots of pride in doing the small details right. It may not be big and important code, but I do like seeing code where people really thought about the details, and clearly also were thinking about the compiler being able to generate efficient code (rather than hoping that the compiler is so smart that it can make efficient code *despite* the state of the original source code). （纠正细节是令人自豪的事。也许这段代码并非庞大和重要，但我喜欢看那些注重代码细节的人写的代码，也就是清楚地了解如何才能编译出有效代码（而不是寄望于聪明的编译器来产生有效代码，即使是那些原始的汇编代码））。

Linus举了一个单向链表的例子，但给出的代码太短了，一般的人很难搞明白这两个代码后面的含义。正好，有个编程爱好者阅读了这段话，并给出了一个比较完整的代码。他的话我就不翻译了，下面给出代码说明。

如果我们需要写一个remove_if(link*, rm_cond_func*)的函数，也就是传入一个单向链表，和一个自定义的是否删除的函数，然后返回处理后的链接。

这个代码不难，基本上所有的教科书都会提供下面的代码示例，而这种写法也是大公司的面试题标准模板：

```
typedef struct node
{
    struct node * next;
    ....
} node;

typedef bool (* remove_fn)(node const * v);

// Remove all nodes from the supplied list for which the
// supplied remove function returns true.
// Returns the new head of the list.
node * remove_if(node * head, remove_fn rm)
{
    for (node * prev = NULL, * curr = head; curr != NULL; )
    {
        node * const next = curr->next;
        if (rm(curr))
        {
            if (prev)
                prev->next = next;
            else
                head = next;
            free(curr);
        }
        else
            prev = curr;
        curr = next;
    }
    return head;
}
```

这里remove_fn由调用查提供的一个是否删除当前实体结点的函数指针，其会判断删除条件是否成立。这段代码维护了两个节点指针prev和curr，标准的教科书写法——删除当前结点时，需要一个previous的指针，并且还要这里还需要做一个边界条件的判断——curr是否为链表头。于是，要删除一个节点（不是表头），只要将前一个节点的next指向当前节点的next指向的对象，即下一个节点（即：prev->next = curr->next），然后释放当前节点。

但在Linus看来，这是不懂指针的人的做法。那么，什么是core low-level coding呢？那就是有效地利用二级指针，将其作为管理和操作链表的首要选项。代码如下：

```
void remove_if(node ** head, remove_fn rm)
```

```
2  {
3      for (node** curr = head; *curr; )
4      {
5          node * entry = *curr;
6          if (rm(entry))
7          {
8              *curr = entry->next;
9              free(entry);
10         }
11         else
12             curr = &entry->next;
13     }
14 }
```

同上一段代码有何改进呢？我们看到：不需要prev指针了，也不需要再去判断是否为链表头了，但是，curr变成了一个指向指针的指针。这正是这段程序的精妙之处。（注意，我所highlight的那三行代码）

让我们来人肉跑一下这个代码，对于——

- 删除节点是表头的情况，输入参数中传入head的二级指针，在for循环里将其初始化curr，然后entry就是*head(*curr)，我们马上删除它，那么第8行就等效于*head = (*head)->next，就是删除表头的实现。
- 删除节点不是表头的情况，对于上面的代码，我们可以看到——

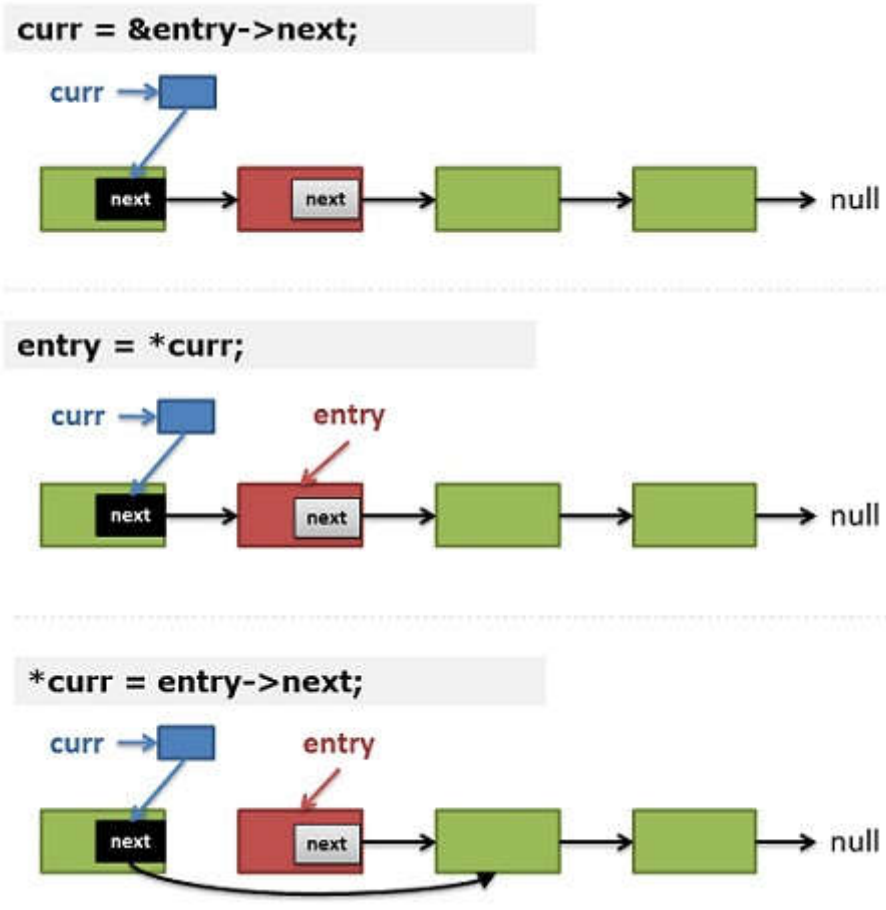
  1）（第12行）如果不删除当前结点 —— curr保存的是当前结点next指针的地址。

  2）（第5行） entry 保存了 *curr —— 这意味着在下一次循环：entry就是prev->next指针所指向的内存。

  3）（第8行）删除结点：*curr = entry->next; —— 于是：prev->next 指向了 entry -> next;

是不是很巧妙？我们可以只用一个二级指针来操作链表，对所有节点都一样。

如果你对上面的代码和描述理解上有困难的话，你可以看看下图的示意：



（全文完）

欢迎关注CoolShell微信公众账号

——=== 访问 酷壳404页面 寻找遗失儿童。 ===——

分类: C/C++语言, Unix/Linux 标签: C++, Coding, Kernel, Linus Torvalds, Linux

(47 人打了分，平均分： 4.55)

相关文章

- 2013年06月30日 Alan Cox：单向链表中prev指针的妙用
- 2009年08月16日 Linus Torvalds 语录 Top 10
- 2010年04月09日 Unix传奇(上篇)
- 2013年07月08日 Alan Cox：大教堂、市集与市议会
- 2011年04月27日 Linux 2.6.39-rc3的一个插曲
- 2011年06月03日 如何写出无法维护的代码
- 2009年04月07日 Linux C 编程一站式学习
- 2011年03月29日 如何学好C语言

评论 (22) Trackbacks (7) 发表评论 Trackback

1. 
liang
2013年12月25日12:35 | #1
回复 | 引用

curr = &entry->next; 要是写成 &(*curr)->next; 可能更加合理直观一点。
最后的图片，如果补充强调一下蓝色的内存区域可能是节点内next 字段也不错。

2. 
郁白
2013年12月26日17:19 | #2
回复 | 引用

这代码可读性不好，使用一个匿名head结点不是更好吗

3. 
Saigut
2014年1月4日13:23 | #3
回复 | 引用

删除节点时的格式：
前一个节点的next指针 = 下一个节点的地址
“前一个节点的next指针”该如何表示？人们很习惯的想到：
PrePtr->next
不过当被删除的节点是第一个的时候，就不能这样用了，因为没有前一个节点，只能这样：
head
所以要判断一下应该用哪个。

但是 PrePtr->next和head的作用几乎一样啊，都是指向下一个节点，有什么办法统一表示吗？
可以，它们的类型是一样的，可以用一个变量比如pp保存它们的地址。
当pp保存的是 PrePtr->next 的地址的时候，它就可以代表 PrePtr->next，当pp保存的是head的地址的时候它就可以代表head。
怎么代表？当然就是*pp。
这样用*pp就可以统一代表“ 前一个节点的next指针 ” ，而且改变*pp所代表的指针也是很容易的（只要改pp所保存的地址就可以了）。

另外一楼 @Leo 所举的例子是不对的。实际上从“第三次解”那里就没有意义了。如果dummy的值是头指针的地址的话，那么*dummy代表头指针，它的值也就是第一个节点的地址。而**dummy代表的是第一个节点，它的类型不是一个指针。因而*(*(*dummy))就没有意义了。多级（大于三级）的指针在c语言中用处应该是不大的。
除非有一个返回值是地址的节点类型才有可能满足你所说的，不过这种类型太复杂吧，c看起

来不适合做这个。

4. 
MuteCoder
2014年1月14日10:22 | #4
回复 | 引用

> Leo ：
> 换言之，如果一个单向链表，next是第一个字段，我们可以用一个二级指针dummy引用一条链表上的所有节点。
> struct node **dummy = &head->next;
> 一次解引用*dummy指向头结点head
> 二次解引用**dummy指向head下一个节点
> 三次解引用*(*(*dummy))指向第三个节点
> 以此类推……
> 如果我们需要找到从head开始的第N个节点，那么对dummy进行N次解引用即可，当然现实工程中一般不会用到这么tricky的语法糖，但使用一个变量同时引用相邻三个节点是很有用的;-)。

只是钻了内存的空而已

5. 
shady
2014年2月12日11:22 | #5
回复 | 引用

这么经典的好文在酷壳上又温习一遍，最后的图解很到位，谢谢 Leo~

6. 
andrew_show
2014年2月14日13:39 | #6
回复 | 引用

linux中的list_head和windows中的LIST_ENTRY都是双向链表的实现，两者的实现基本是等同的，和这里的单链表一样，也使用了一个dummy的头来指示链表的头尾。当dummy的头的prev/next指向dummy自身的时候，链表为空。利用dummy的头，可以避免一些不必要的判断，使代码变得精简。
使用container_of/CONTAINING_RECORD宏可以从list_head/LIST_ENTRY指针反推包含节点的数据结构。
这种实现方式简洁，高效，个人认为这种实现方式可以说是双向链表的极致了。

7. 
骚人墨客
2014年4月18日19:43 | #7
回复 | 引用

看个这段代码，明白了，多谢！@fox的爱智慧

8. 
hong1991
2014年8月5日13:29 | #8
回复 | 引用

@Neuron Teckid 他们还可能会因为看不懂你的代码要求你写得和他们一样。

9. 
nano
2014年10月13日01:12 | #9
回复 | 引用

图画错了，curr是node的二重指针，所以通过两个“箭头”指向以后就应该指向一个node实体，蓝色方块是多余的。

10. 
nano
2014年10月13日01:33 | #10
回复 | 引用

> nano ：
> 图画错了，curr是node的二重指针，所以通过两个“箭头”指向以后就应该指向一个node实体，蓝色方块是多余的。

就是这个错误让我看了半天看不懂，我都要怀疑自己懂不懂指针了T_T

11. 
zxcbryanfury
2015年1月4日23:15 | #11
回复 | 引用

我觉得理解的关键在于，在C++中，变量名的意义是拷贝一个值。之前的常见做法，cur = head->next，仅仅是把指针head->next的值copy给了指针cur，使得cur可以访问链表的下一个节点；而声明为二级指针的cur，*cur不是head->next的copy值，它“就是”head->next这个变量名的别名，它获得的是head->next这个变量的读写权

12. 
liaogang
2015年2月11日14:39 | #12
回复 | 引用

分段处理:

```
ListNode *removeIf(ListNode *head,remove_fn rm)
{
/// find new head.
for ( ;head && rm (head); head=head->next)
{}

/// others.
for ( ListNode *prev = head , *curr = head->next ; curr ; curr = curr->next )
{
if ( rm(curr))
prev->next = curr->next;
}

return head;
}
```

13. 
ming

2015年3月13日12:42 | #13
回复 | 引用

node** curr = head 是必须的吗? 可不可以直接用 head?

```
void remove_if(node ** head, remove_fn rm)
{
while(*head)
{
node * entry = *head;
if (rm(entry))
{
*head = entry->next;
free(entry);
}
else
head = &entry->next;
}
}
```

14. 

nnkken
2015年4月16日21:25 | #14
回复 | 引用

其實完全可以這樣想:
因為remove_if有可能把head刪掉，所以必須1. 返回新的head，然後調用方自行更改head，或2. 傳進&head。
把後一種情況做成遞歸實現:

```
typedef struct _Node {
struct _Node* next;
int value;
} Node;

void remove_if_greater_than(Node** p_head, int limit) {
if (p_head == NULL || *p_head == NULL) {
return;
}
Node* head = *p_head;
if (head->value > limit) {
*p_head = head->next;
free(head);
remove_if_greater_than(p_head, limit);
} else {
remove_if_greater_than(&(head->next), limit);
}
}
```

然後做個尾遞歸優化:

```
void remove_if_greater_than_tail_opt(Node** p_curr, int limit) {
while (p_curr != NULL && *p_curr != NULL) {
Node* curr = *p_curr;
if (curr->value > limit) {
*p_curr = curr->next;
free(curr);
} else {
```

```
p_curr = &(curr->next);
}
}
}
```

就能得到Linus的解法，而且相當直觀。
關鍵在於，我們完全可以把head->next想像成為另一個鏈表的head。
所以你瞧，函數式語言多棒啊，讓你可以理解類型的本質，雖說我還沒真正用過函數式語言……

15. 
fei
2015年5月21日02:55 | #15
回复 | 引用

个人认为，简洁不能以难懂作为代价。也许对于内核开发人员来说，像Linus那样是容易懂的写法。
我会坚持啰嗦的写法。

16. 
feimi
2015年6月21日18:38 | #16
回复 | 引用

> card323 :
> @viho he
> Linus的这个程序和你说的那个特性貌似没什么关系吧
> 你写的node == &node->next应该是想说&node == &node->next的意思吧
> 无论next指针在结构体中的定义位置 linux的程序都是没问题的吧

把“吧”去掉，已经验证过，跟结构体书写没关系。

17. 
0xFFFFFFFF
2015年7月29日05:13 | #17
回复 | 引用

这段代码，乍一看很牛其实仔细一琢磨，唯一的作用是用来炫耀。首先 取地址 运算符 只能作用于lvalue，所以创建的时候 还是需要一个临时变量指向 链表节点。 如果这样 真不如只用一级指针然后
```
void remove_if(node* head, remove_fn rm) {
node H;
H.next = head;
….
return H.next;
}
```

还有 linus 进场喷 C++ 其实 在C++中 node*& head 更优雅

18. 
cscareer
2015年8月6日09:41 | #18
回复 | 引用

@0xFFFFFFFF

如果构造dummy head H很费呢？

19. 

Drogon
2015年8月19日18:55 | #19
回复 | 引用

这个实际上非常简单，主要是两种删除节点的对比。怎么会被解释得这么复杂了呢？
第一种： prev->next = next, 由于head的没有直接的prev的缘故， 所以使用NULL，NULL没有prev所以需要特别处理，同时更新head的值。
这样理解的话就算函数不是返回node *，而是和下面一样使用node **， 返回值为void 也没有
代码：

```
node * remove_if(node * *head, remove_fn rm)
{
for (node * prev = NULL, * curr = *head; curr != NULL; )
{
node * const next = curr->next;
if (rm(curr))
{
if (prev)
prev->next = next;
else
*head = next;
free(curr);
}
else
prev = curr;
curr = next;
}
}
```

这样head的值也被修改了，无需返回node *。

第二种： curr = curr->next, curr的值可以为head，所以不需要特殊判断，但是这样不能更新head的值，因为head就是指针，所以这儿直接使用了二维指针来修改head的值，直接改为*curr = (*curr)->next, 而此处的curr为entry。entry这个是用来free的，要小心。

综上： 这个实际上是两种删除节点方法的比较，和二维指针真心关系不是特别大。

20. 

Drogon
2015年8月19日23:51 | #20
回复 | 引用

@Drogon
同时第二种 还是current 这点确实是使用二维指针 使得curr指向的prev->next的值被修改了。整个思路按照 *curr = (*curr) -> next, 而curr又是prev->next来理解却是简单了很多。

21. 

sidgwick
2015年12月5日20:25 | #21
回复 | 引用

memcached里面的哈希表好像就是这么用的， 当时就觉得好牛逼.

22. 
西北疯
2016年6月22日22:57 | #22
回复 | 引用

> zxcbryanfury :
> 我觉得理解的关键在于，在C++中，变量名的意义是拷贝一个值。之前的常见做法，cur = head->next，仅仅是把指针head->next的值copy给了指针cur，使得cur可以访问链表的下一个节点；而声明为二级指针的cur，*cur不是head->next的copy值，它“就是”head->next这个变量名的别名，它获得的是head->next这个变量的读写权

我觉得你这个回答，是这里最精辟的。我用自己的语言描述一遍：因为“链表操作”的对象都是指针，所以用二级指针来操作“指针”这种特殊的变量，就像一般情况下传递一个指针给函数，来操作普通变量一样。可以获得变量的读写权，由于头指针又同时作为函数的参数，所以这里用二级指针，同时获得了改变外部变量（不用返回新的头指针）和内部的权限，省却一个临时内部变量。
其实，那个所谓的“教科书标准示例代码”，也是可以简化的，不用定义prev指针，也是可以的。但是新的头指针必须返回。没有二级指针，是无法改变函数外“指针”类型的实参的。

评论分页
« 上一页 1 2 3 4 8990

1. 2013年12月21日14:54 | #1
   使用二级指针操作单链表
2. 2013年12月26日16:41 | #2
   2013年个人微博推荐技术资料汇总 | 何登成的技术博客
3. 2014年6月14日16:29 | #3
   次级指针的灵活使用 | tomsawyer
4. 2014年6月21日00:51 | #4
   利用二级指针删除单向链表——笔记 | KryptosX 实验室
5. 2015年1月1日03:46 | #5
   Linus：为何对象引用计数必须是原子的 | 星达红
6. 2015年4月15日14:14 | #6
   二级指针在单向链表中的应用 | Coding and Thinking
7. 2016年5月3日10:19 | #7
   Alan Cox：单向链表中prev指针的妙用 | Codeba.cc | Codeba

Twitter

本站公告

酷壳建议大家多使用RSS访问阅读（本站已经是全文输出，推荐使用cloud.feedly.com 或digg.com）。有相关事宜欢迎电邮：haoel(at)hotmail.com。最后，感谢大家对酷壳的支持和体谅！

最新文章

- 性能测试应该怎么做？

- 让我们来谈谈分工
- Cuckoo Filter：设计与实现
- Docker基础技术：DeviceMapper
- Docker基础技术：AUFS
- Docker基础技术：Linux CGroup
- Docker基础技术：Linux Namespace（上）
- Docker基础技术：Linux Namespace（下）
- 关于移动端的钓鱼式攻击
- Linus：为何对象引用计数必须是原子的
- DHH 谈混合移动应用开发
- HTML6 展望
- Google Inbox如何跨平台重用代码？
- vfork 挂掉的一个问题
- Leetcode 编程训练
- State Threads 回调终结者
- bash代码注入的安全漏洞
- 互联网之子 - Aaron Swartz
- 谜题的答案和活动的心得体会
- 【活动】解谜题送礼物
- 开发团队的效率
- TCP 的那些事儿（下）
- TCP 的那些事儿（上）
- 「我只是认真」聊聊工匠情怀
- 面向GC的Java编程
- C语言的整型溢出问题
- 从LongAdder看更高效的无锁实现
- 从Code Review 谈如何做技术
- C语言结构体里的成员数组和指针
- 无插件Vim编程技巧

全站热门

- 程序员技术练级攻略
- 简明 Vim 练级攻略
- 做个环保主义的程序员
- 如何学好C语言
- AWK 简明教程
- 应该知道的Linux技巧
- TCP 的那些事儿（上）
- “21天教你学会C++”
- 6个变态的C语言Hello World程序
- 由12306.cn谈谈网站性能技术
- 编程能力与编程年龄
- “作环保的程序员，从不用百度开始”
- 28个Unix/Linux的命令行神器
- sed 简明教程
- 我是怎么招聘程序员的
- 性能调优攻略
- Web开发中需要了解的东西
- C++ 程序员自信心曲线图
- Android将允许纯C/C++开发应用
- Lua简明教程
- 如何学好C++语言
- MySQL性能优化的最佳20+条经验
- 二维码的生成细节和原理
- 如何写出无法维护的代码
- 无插件Vim编程技巧

- 20本最好的Linux免费书籍
- Windows编程革命简史
- 编程真难啊
- 深入理解C语言
- 加班与效率

新浪微博

微博

左耳朵耗子　北京 朝阳区

加关注

标签

agile AJAX Algorithm Android Bash C++ Coding CSS Database Design design pattern ebook Flash Game Go Google HTML IE Java Javascript jQuery Linux MySQL OOP password Performance PHP Programmer Programming programming language Puzzle Python Ruby SQL TDD UI Unix vim Web Windows XML 安全 程序员 算法 面试

分类目录

- .NET编程 (3)
- Ajax开发 (9)
- C/C++语言 (71)
- Erlang (1)
- Java语言 (32)
- PHP脚本 (11)
- Python (23)
- Ruby (5)
- Unix/Linux (74)
- Web开发 (103)
- Windows (12)
- 业界新闻 (26)
- 企业应用 (2)
- 技术新闻 (33)
- 技术管理 (13)
- 技术读物 (117)
- 操作系统 (49)
- 数据库 (11)
- 杂项资源 (267)
- 流程方法 (47)
- 程序设计 (85)
- 系统架构 (8)
- 编程工具 (65)
- 编程语言 (174)
- 网络安全 (27)
- 职场生涯 (33)
- 趣味问题 (19)
- 轶事趣闻 (147)

归档

- 2016年七月 (1)
- 2015年十二月 (1)
- 2015年九月 (1)
- 2015年八月 (2)
- 2015年四月 (4)
- 2014年十二月 (3)
- 2014年十一月 (2)
- 2014年十月 (2)
- 2014年九月 (2)
- 2014年八月 (2)
- 2014年六月 (1)
- 2014年五月 (4)
- 2014年四月 (4)
- 2014年三月 (5)
- 2014年二月 (3)
- 2014年一月 (2)
- 2013年十二月 (3)
- 2013年十一月 (1)
- 2013年十月 (6)
- 2013年八月 (1)
- 2013年七月 (8)
- 2013年六月 (2)
- 2013年五月 (3)
- 2013年四月 (3)
- 2013年三月 (3)
- 2013年二月 (5)
- 2013年一月 (1)
- 2012年十二月 (4)
- 2012年十一月 (4)
- 2012年十月 (3)
- 2012年九月 (4)
- 2012年八月 (8)
- 2012年七月 (4)
- 2012年六月 (7)
- 2012年五月 (6)
- 2012年四月 (6)
- 2012年三月 (6)
- 2012年二月 (3)
- 2012年一月 (6)
- 2011年十二月 (5)
- 2011年十一月 (9)
- 2011年十月 (6)
- 2011年九月 (5)
- 2011年八月 (14)
- 2011年七月 (6)
- 2011年六月 (12)
- 2011年五月 (5)
- 2011年四月 (18)
- 2011年三月 (16)
- 2011年二月 (16)
- 2011年一月 (18)
- 2010年十二月 (11)
- 2010年十一月 (11)
- 2010年十月 (19)
- 2010年九月 (15)

- 2010年八月 (10)
- 2010年七月 (20)
- 2010年六月 (9)
- 2010年五月 (13)
- 2010年四月 (12)
- 2010年三月 (11)
- 2010年二月 (7)
- 2010年一月 (9)
- 2009年十二月 (22)
- 2009年十一月 (27)
- 2009年十月 (17)
- 2009年九月 (14)
- 2009年八月 (21)
- 2009年七月 (18)
- 2009年六月 (19)
- 2009年五月 (27)
- 2009年四月 (53)
- 2009年三月 (43)
- 2008年十月 (1)
- 2007年十二月 (1)
- 2006年十一月 (1)
- 2004年六月 (1)

最新评论

- SteelWood: 读老文才发现自己跟大牛的差距至少还有4年
- wings1234: 您好， Java构造时成员初始化的陷阱这篇文章中有个地方不理解，请百 忙之中帮忙解答一下，谢谢。 5.调用preProcess，因为被子类override，所 以调用的是子类的。...
- lgn21st: 请问性格测试应该怎么做呢？;)
- http://www.bipgodz.com/: NemaÄiau tokiÅ³ lempuÄiÅ³ Lietuvos pardose, kur nemeluotÅ³ su darbo laiku, tai yra visos...
- 仙人掌: 目前来看，这类观点老板写得比较少，干活的人写得比较多，这是一 个有意思的地方。
- sean fu: 现在有浩兄这样负责任， 有水平的， 已经找不出几人了
- Justfly: 在web应用中，吞吐量这个概念是会细分为并发数（跟用户数正相 关）和QPS的，在限制latency 都是 1ms 的情况下，并发100 时的QPS和200时的是不一样的，这样我们说系统的承载...
- 起个名字好难: @峰云就她了 你猜我是谁
- advice: 最近浏览你的网站发现一个小问题，在首页点击浏览全文，进入文章 全文，然而直接定位到你浏览的下一行，个人觉得应该定位到你浏览 位置的上两行，因为在跳转的时候人的思维是有短暂性的消失...
- ShiWaXinGe: 对于需要带参数的单例只有1.4版本能用了。1.5+不能带参数 。这是我目前的看法。
- knull: 我也进行过一些性能测试，但是进行的次数越多，就越迷糊：到底该 怎么进行性能测试？（我的性能测试，都是我自己想象中的性能测试 ，自己开发、自己测试；并不是专业的调理） 终于看到耗子大哥这文章了，真的是非常需要。...
- 陈皓: 谢谢，已修正
- iamper: The “mean” is the “average” you’re used to, where you add up all the numbers...
- OrangeCLK: @dy 有出处吗？据我所知没有这个意思，我查了维基百科也没看见。
- OrangeCLK: mean 是均值，median 才是中位数。

友情链接

- 陈皓的博客
- 并发编程

- 四火的唠叨
- HelloGcc Working Group
- 吕毅的Blog
- Todd Wei的Blog
- C++爱好者博客
- HTML5研究小组
- 朱文昊Albert Zhu
- C瓜哥的博客
- 开源吧
- ACMer
- 陈鹏个人博客
- OneCoder
- More Than Vimer
- 运维派
- 书巢

功能

- 注册
- 登录
- 文章RSS
- 评论RSS
- WordPress.org