<https://blog.csdn.net/luotuo44/article/details/38374009>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38374009**](http://blog.csdn.net/luotuo44/article/details/38374009)

        Libevent源码中有一个queue.h文件，位于compat/sys目录下。该文件里面定义了5个数据结构，其中TAILQ\_QUEUE是使得最广泛的。本文就说一下这个数据结构。

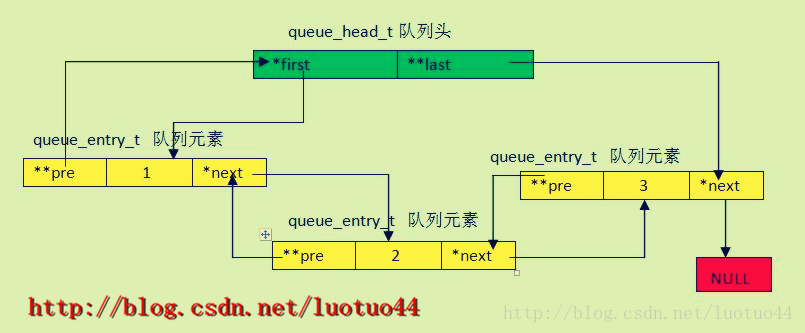
# 队列结构体：

        TAILQ\_QUEUE由下面两个结构体一起配合工作。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. #define TAILQ\_HEAD(name, type)                      \
2. **struct** name {                               \
3. **struct** type \*tqh\_first; /\* first element \*/         \
4. **struct** type \*\*tqh\_last; /\* addr of last next element \*/     \
5. }
7. //和前面的TAILQ\_HEAD不同，这里的结构体并没有name.即没有结构体名。
8. //所以该结构体只能作为一个匿名结构体。所以，它一般都是另外一个结构体
9. //或者共用体的成员
10. #define TAILQ\_ENTRY(type)                       \
11. **struct** {                                \
12. **struct** type \*tqe\_next;  /\* next element \*/          \
13. **struct** type \*\*tqe\_prev; /\* address of previous next element \*/  \
14. }

        由这两个结构体配合构造出来的队列一般如下图所示：



        图中，一级指针指向的是queue\_entry\_t这个结构体，即存储queue\_entry\_t这个结构体的地址值。二级指针存储的是一级地址变量的地址值。所以二级指针指向的是图中的一级指针，而非结构体。图中的1,2, 3为队列元素保存的一些值。

# 队列操作宏函数以及使用例子：

        除了这两个结构体，在queue.h文件中，还为TAILQ\_QUEUE定义了一系列的访问和操作函数。很不幸，它们是一些宏定义。这里就简单贴几个函数（准确来说，不是函数）的代码。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. #define TAILQ\_FIRST(head)       ((head)->tqh\_first)
3. #define TAILQ\_NEXT(elm, field)      ((elm)->field.tqe\_next)
5. #define TAILQ\_INIT(head) do {                       \
6. (head)->tqh\_first = NULL;                    \
7. (head)->tqh\_last = &(head)->tqh\_first;                \
8. } **while** (0)
10. #define TAILQ\_INSERT\_TAIL(head, elm, field) do {            \
11. (elm)->field.tqe\_next = NULL;                    \
12. (elm)->field.tqe\_prev = (head)->tqh\_last;         \
13. \*(head)->tqh\_last = (elm);                   \
14. (head)->tqh\_last = &(elm)->field.tqe\_next;            \
15. } **while** (0)
17. #define TAILQ\_REMOVE(head, elm, field) do {             \
18. **if** (((elm)->field.tqe\_next) != NULL)             \
19. (elm)->field.tqe\_next->field.tqe\_prev =           \
20. (elm)->field.tqe\_prev;               \
21. **else**                                \
22. (head)->tqh\_last = (elm)->field.tqe\_prev;     \
23. \*(elm)->field.tqe\_prev = (elm)->field.tqe\_next;           \
24. } **while** (0)

        这些宏是很难看的，也没必要直接去看这些宏。下面来看一个使用例子。有例子更容易理解。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. //队列中的元素结构体。它有一个值，并且有前向指针和后向指针
2. //通过前后像指针，把队列中的节点(元素)连接起来
3. **struct** queue\_entry\_t
4. {
5. **int** value;
7. //从TAILQ\_ENTRY的定义可知，它只能是结构体或者共用体的成员变量
8. TAILQ\_ENTRY(queue\_entry\_t)entry;
9. };
11. //定义一个结构体，结构体名为queue\_head\_t，成员变量类型为queue\_entry\_t
12. //就像有头节点的链表那样，这个是队列头。它有两个指针，分别指向队列的头和尾
13. TAILQ\_HEAD(queue\_head\_t, queue\_entry\_t);
15. **int** main(**int** argc, **char** \*\*argv)
16. {
17. **struct** queue\_head\_t queue\_head;
18. **struct** queue\_entry\_t \*q, \*p, \*s, \*new\_item;
19. **int** i;
21. TAILQ\_INIT(&queue\_head);
23. **for**(i = 0; i < 3; ++i)
24. {
25. p = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
26. p->value = i;
28. //第三个参数entry的写法很怪，居然是一个成员变量名(field)
29. TAILQ\_INSERT\_TAIL(&queue\_head, p, entry);//在队尾插入数据
30. }
32. q = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
33. q->value = 10;
34. TAILQ\_INSERT\_HEAD(&queue\_head, q, entry);//在队头插入数据
36. //现在q指向队头元素、p指向队尾元素
38. s = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
39. s->value = 20;
40. //在队头元素q的后面插入元素
41. TAILQ\_INSERT\_AFTER(&queue\_head, q, s, entry);

44. s = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
45. s->value = 30;
46. //在队尾元素p的前面插入元素
47. TAILQ\_INSERT\_BEFORE(p, s, entry);
49. //现在进行输出
50. //获取第一个元素
51. s = TAILQ\_FIRST(&queue\_head);
52. printf("the first entry is %d\n", s->value);
54. //获取下一个元素
55. s = TAILQ\_NEXT(s, entry);
56. printf("the second entry is %d\n\n", s->value);
58. //删除第二个元素, 但并没有释放s指向元素的内存
59. TAILQ\_REMOVE(&queue\_head, s, entry);
60. free(s);

63. new\_item = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
64. new\_item->value = 100;
66. s = TAILQ\_FIRST(&queue\_head);
67. //用new\_iten替换第一个元素
68. TAILQ\_REPLACE(&queue\_head, s, new\_item, entry);

71. printf("now, print again\n");
72. i = 0;
73. TAILQ\_FOREACH(p, &queue\_head, entry)//用foreach遍历所有元素
74. {
75. printf("the %dth entry is %d\n", i, p->value);
76. }
78. p = TAILQ\_LAST(&queue\_head, queue\_head\_t);
79. printf("last is %d\n", p->value);
81. p = TAILQ\_PREV(p, queue\_head\_t, entry);
82. printf("the entry before last is %d\n", p->value);
83. }

        例子并不难看懂。这里就不多讲了。

# 展开宏函数：

        下面把这些宏翻译一下(即展开)，显示出它们的本来面貌。这当然不是用人工方式去翻译。而是用gcc 的-E选项。

        阅读代码时要注意，tqe\_prev和tqh\_last都是二级指针，行为会有点难理解。平常我们接触到的双向链表，next和prev成员都是一级指针。对于像链表A->B->C（把它们想象成双向链表），通常B的prev指向A这个结构体本身。此时，B->prev->next指向了本身。但队列Libevent的TAILQ\_QUEUE，B的prev是一个二级指向，它指向的是A结构体的next成员。此时，\*B->prev就指向了本身。当然，这并不能说用二级指针就方便。我觉得用二级指针理解起来更难，编写代码更容易出错。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. //队列中的元素结构体。它有一个值，并且有前向指针和后向指针
2. //通过前后像指针，把队列中的节点连接起来
3. **struct** queue\_entry\_t
4. {
5. **int** value;
7. **struct**
8. {
9. **struct** queue\_entry\_t \*tqe\_next;
10. **struct** queue\_entry\_t \*\*tqe\_prev;
11. }entry;
12. };
14. //就像有头节点的链表那样，这个是队列头。它有两个指针，分别指向队列的头和尾
15. **struct** queue\_head\_t
16. {
17. **struct** queue\_entry\_t \*tqh\_first;
18. **struct** queue\_entry\_t \*\*tqh\_last;
19. };
21. **int** main(**int** argc, **char** \*\*argv)
22. {
23. **struct** queue\_head\_t queue\_head;
24. **struct** queue\_entry\_t \*q, \*p, \*s, \*new\_item;
25. **int** i;
27. //TAILQ\_INIT(&queue\_head);
28. **do**
29. {
30. (&queue\_head)->tqh\_first = 0;
31. //tqh\_last是二级指针，这里指向一级指针
32. (&queue\_head)->tqh\_last = &(&queue\_head)->tqh\_first;
33. }**while**(0);
35. **for**(i = 0; i < 3; ++i)
36. {
37. p = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
38. p->value = i;
40. //TAILQ\_INSERT\_TAIL(&queue\_head, p, entry);在队尾插入数据
41. **do**
42. {
43. (p)->entry.tqe\_next = 0;
44. //tqh\_last存储的是最后一个元素(队列节点)tqe\_next成员
45. //的地址。所以，tqe\_prev指向了tqe\_next。
46. (p)->entry.tqe\_prev = (&queue\_head)->tqh\_last;
48. //tqh\_last存储的是最后一个元素(队列节点)tqe\_next成员
49. //的地址，所以\*(&queue\_head)->tqh\_last修改的是最后一个
50. //元素的tqe\_next成员的值，使得tqe\_next指向\*p(新的队列
51. //节点)。
52. \*(&queue\_head)->tqh\_last = (p);
53. //队头结构体（queue\_head）的tqh\_last成员保存新队列节点的
54. //tqe\_next成员的地址值。即让tqh\_last指向tqe\_next。
55. (&queue\_head)->tqh\_last = &(p)->entry.tqe\_next;
56. }**while**(0);
57. }
59. q = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
60. q->value = 10;
62. //TAILQ\_INSERT\_HEAD(&queue\_head, q, entry);在队头插入数据
63. **do** {
64. //queue\_head队列中已经有节点(元素了)。要对第一个元素进行修改
65. **if**(((q)->entry.tqe\_next = (&queue\_head)->tqh\_first) != 0)
66. (&queue\_head)->tqh\_first->entry.tqe\_prev = &(q)->entry.tqe\_next;
67. **else**//queue\_head队列目前是空的，还没有任何节点（元素）。修改queue\_head即可
68. (&queue\_head)->tqh\_last = &(q)->entry.tqe\_next;
70. //queue\_head的first指针指向要插入的节点\*q
71. (&queue\_head)->tqh\_first = (q);
72. (q)->entry.tqe\_prev = &(&queue\_head)->tqh\_first;
73. }**while**(0);
75. //现在q指向队头元素、p指向队尾元素
77. s = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
78. s->value = 20;
80. //TAILQ\_INSERT\_AFTER(&queue\_head, q, s, entry);在队头元素q的后面插入元素
81. **do**
82. {
83. //q不是最后队列中最后一个节点。要对q后面的元素进行修改
84. **if** (((s)->entry.tqe\_next = (q)->entry.tqe\_next) != 0)
85. (s)->entry.tqe\_next->entry.tqe\_prev = &(s)->entry.tqe\_next;
86. **else**//q是最后一个元素。对queue\_head修改即可
87. (&queue\_head)->tqh\_last = &(s)->entry.tqe\_next;
89. (q)->entry.tqe\_next = (s);
90. (s)->entry.tqe\_prev = &(q)->entry.tqe\_next;
91. }**while**(0);


95. s = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
96. s->value = 30;
98. //TAILQ\_INSERT\_BEFORE(p, s, entry); 在队尾元素p的前面插入元素
99. **do**
100. {
101. //无需判断节点p前面是否还有元素。因为即使没有元素，queue\_head的两个
102. //指针从功能上也相当于一个元素。这点是采用二级指针的一大好处。
103. (s)->entry.tqe\_prev = (p)->entry.tqe\_prev;
104. (s)->entry.tqe\_next = (p);
105. \*(p)->entry.tqe\_prev = (s);
106. (p)->entry.tqe\_prev = &(s)->entry.tqe\_next;
107. }**while**(0);

110. //现在进行输出
112. // s = TAILQ\_FIRST(&queue\_head);
113. s = ((&queue\_head)->tqh\_first);
114. printf("the first entry is %d\n", s->value);
116. // s = TAILQ\_NEXT(s, entry);
117. s = ((s)->entry.tqe\_next);
118. printf("the second entry is %d\n\n", s->value);
120. //删除第二个元素, 但并没有释放s指向元素的内存
121. //TAILQ\_REMOVE(&queue\_head, s, entry);
122. **do**
123. {
124. **if** (((s)->entry.tqe\_next) != 0)
125. (s)->entry.tqe\_next->entry.tqe\_prev = (s)->entry.tqe\_prev;
126. **else** (&queue\_head)->tqh\_last = (s)->entry.tqe\_prev;
128. \*(s)->entry.tqe\_prev = (s)->entry.tqe\_next;
129. }**while**(0);
131. free(s);

134. new\_item = (**struct** queue\_entry\_t\*)malloc(**sizeof**(**struct** queue\_entry\_t));
135. new\_item->value = 100;
137. //s = TAILQ\_FIRST(&queue\_head);
138. s = ((&queue\_head)->tqh\_first);
140. //用new\_iten替换第一个元素
141. //TAILQ\_REPLACE(&queue\_head, s, new\_item, entry);
142. **do**
143. {
144. **if** (((new\_item)->entry.tqe\_next = (s)->entry.tqe\_next) != 0)
145. (new\_item)->entry.tqe\_next->entry.tqe\_prev = &(new\_item)->entry.tqe\_next;
146. **else**
147. (&queue\_head)->tqh\_last = &(new\_item)->entry.tqe\_next;
149. (new\_item)->entry.tqe\_prev = (s)->entry.tqe\_prev;
150. \*(new\_item)->entry.tqe\_prev = (new\_item);
151. }**while**(0);

154. printf("now, print again\n");
155. i = 0;
156. //TAILQ\_FOREACH(p, &queue\_head, entry)//用foreach遍历所有元素
157. **for**((p) = ((&queue\_head)->tqh\_first); (p) != 0;
158. (p) = ((p)->entry.tqe\_next))
159. {
160. printf("the %dth entry is %d\n", i, p->value);
161. }
163. //p = TAILQ\_LAST(&queue\_head, queue\_head\_t);
164. p = (\*(((**struct** queue\_head\_t \*)((&queue\_head)->tqh\_last))->tqh\_last));
165. printf("last is %d\n", p->value);

168. //p = TAILQ\_PREV(p, queue\_head\_t, entry);
169. p = (\*(((**struct** queue\_head\_t \*)((p)->entry.tqe\_prev))->tqh\_last));
170. printf("the entry before last is %d\n", p->value);
171. }

        代码中有一些注释，不懂的可以看看。其实对于链表操作，别人用文字说再多都对自己理解帮助不大。只有自己动手一步步把链表操作都画出来，这样才能完全理解。

# 特殊指针操作：

        最后那两个操作宏函数有点难理解，现在来讲一下。在讲之前，先看一个关于C语言指针的例子。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

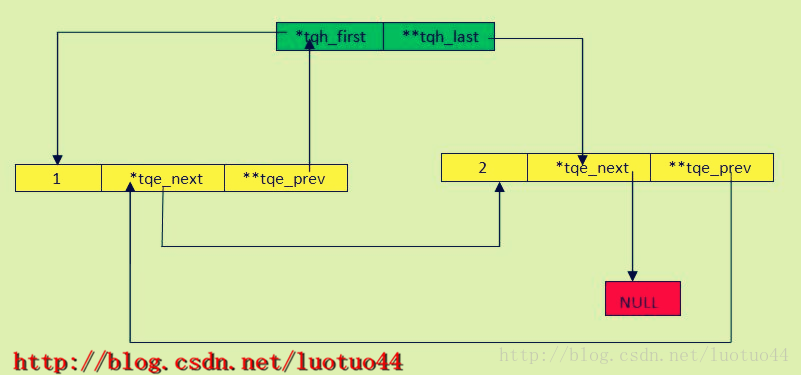
1. #include<stdio.h>
3. **struct** item\_t
4. {
5. **int** a;
6. **int** b;
7. **int** c;
8. };
10. **struct** entry\_t
11. {
12. **int** a;
13. **int** b;
14. };

17. **int** main()
18. {
19. **struct** item\_t item = { 1, 2, 3};
21. entry\_t \*p = (entry\_t\*)(&item.b);
22. printf("a = %d, b = %d\n", p->a, p->b);
24. **return** 0;
25. }

        代码输出的结果是：a = 2, b = 3

        对于entry\_t \*p， 指针p指向的内存地址为&item.b。此时对于编译器来说，它认为从&item.b这个地址开始，是一个entry\_t结构体的内存区域。并且把前4个字节当作entry\_t成员变量a的值，后4个字节当作entry\_t成员变量b的值。所以就有了a = 2, b = 3这个输出。

        好了，现在开始讲解那两个难看懂的宏。先看一张图。



        虽然本文最前面的图布局更好看一点，但这张图才更能反映文中这两个结构体的内存布局。不错，tqe\_next是在tqe\_prev的前面。这使得tqe\_next、tqe\_prev于tqh\_first、tqh\_last的内存布局一样。一级指针在前，二级指针在后。

        现在来解析代码中最后两个宏函数。

## 队尾节点：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. //p = TAILQ\_LAST(&queue\_head, queue\_head\_t);
2. p = (\*(((**struct** queue\_head\_t \*)((&queue\_head)->tqh\_last))->tqh\_last));

        首先是(&queue\_head)->tqh\_last，它的值是最后一个元素的tqe\_next这个成员变量的地址。然后把这个值强制转换成struct queue\_head\_t \*指针。此时，相当于有一个匿名的struct queue\_head\_t类型指针q。它指向的地址为队列的最后一个节点的tqe\_next成员变量的地址。无论一级还是二级指针，其都是指向另外一个地址。只是二级指针只能指向一个一级指针的地址。

        此时，在编译器看来，从tqe\_next这个变量的地址开始，是一个struct queue\_head\_t结构体的内存区域。并且可以将代码简写成：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. p = (\*(q->tqh\_last));

        回想一下刚才的那个例子。q->tqh\_last的值就是上图中最后一个节点的tqe\_prev成员变量的值。所以\*(q->tqh\_last))就相当于\*tqe\_prev。注意，变量tqe\_prev是一个二级指针，它指向倒数第二个节点的tqe\_next成员。所以\*tqe\_prev获取了倒数第二个节点的tqe\_next成员的值。它的值就是最后一个节点的地址。最后，将这个地址赋值给p，此时p指向最后一个节点。完成了任务。好复杂的过程。

## 前一个节点：

        现在来看一下最后那个宏函数，代码如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. //p = TAILQ\_PREV(p, queue\_head\_t, entry);
2. p = (\*(((**struct** queue\_head\_t \*)((p)->entry.tqe\_prev))->tqh\_last));

        注意，右边的p此时是指向最后一个节点(元素)的。所以(p)->entry.tqe\_prev就是倒数第二个节点tqe\_next成员的地址。然后又强制转换成struct queue\_head\_t指针。同样，假设一个匿名的struct queue\_head\_t \*q;此时，宏函数可以转换成:

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. p = (\*((q)->tqh\_last));

        同样，在编译器看来，从倒数第二个参数节点tqe\_next的地址开始，是一个structqueue\_head\_t结构体的内存区域。所以tqh\_last实际值是tqe\_prev变量上的值，即tqe\_prev指向的地址。\*((q)->tqh\_last)就是\*tqe\_prev，即获取tqe\_prev指向的倒数第三个节点的tqe\_next的值。而该值正是倒数第二个节点的地址。将这个地址赋值给p，此时，p就指向了倒数第二个节点。完成了TAILQ\_PREV函数名的功能。

        这个过程确实有点复杂。而且还涉及到强制类型转换。

        其实，在TAILQ\_LAST(&queue\_head, queue\_head\_t);中，既然都可以获取最后一个节点的tqe\_next的地址值，那么直接将该值 + 4就可以得到tqe\_precv的地址值了（假设为pp）。有了该地址值pp，那么直接\*\*pp就可以得到最后一个节点的地址了。代码如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. **struct** queue\_entry\_t \*\*pp = (&queue\_head)->tqh\_last;
2. pp += 1; //加1个指针的偏移量，在32位的系统中，就等于+4
4. //因为这里得到的是二级指针的地址值，所以按理来说，得到的是一个
5. //三级指针。故要用强制转换成三级指针。
6. **struct** queue\_entry\_t \*\*\*ppp = (**struct** queue\_entry\_t \*\*\*)pp;
8. s = \*\*ppp;
9. printf("the last is %d\n", s->value);

        该代码虽然能得到正确的结果，但总感觉直接加上一个偏移量的方式太粗暴了。

        有一点要提出，+1那里并不会因为在64位的系统就不能运行，一样能正确运行的。因为1不是表示一个字节，而是一个指针的偏移量。在64位的系统上一个指针的偏移量为8字节。这种”指针 + 数值”，实际其增加的值为:数值 + sizeof(\*指针)。不信的话，可以试一下char指针、int指针、结构体指针(结构体要有多个成员)。

        好了，还是回到最开始的问题上吧。这个TAILQ\_QUEUE队列是由两部分组成：队列头和队列节点。在Libevent中，队列头一般是event\_base结构体的一个成员变量，而队列节点则是event结构体。比如event\_base结构体里面有一个struct event\_list eventqueue;其中，结构体struct event\_list如下定义：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38374009) [copy](https://blog.csdn.net/luotuo44/article/details/38374009)

1. //event\_struct.h
2. TAILQ\_HEAD (event\_list, event);
4. //所以event\_list的定义展开后如下：
5. **struct** event\_list
6. {
7. **struct** event \*tqh\_first;
8. **struct** event \*\*tqh\_last;
9. };

        在event结构体中，则有几个TAILQ\_ENTRY(event)类型的成员变量。这是因为根据不同的条件，采用不同的队列把这些event结构体连在一起，放到一条队列中。

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/luotuo44/article/details/38374009