[写了个迷你版本的Nginx定时机制的实现](http://blog.chinaunix.net/uid-10106787-id-3150804.html) 2012-03-27 19:59:36

分类： LINUX

我们知道在Nginx中许多地方需要用到定时器，典型的就是当accept接受连接后，如果在一定时间内还未收到用户请求，就超时断开连接。那这个是如何实现的呢？这个问题我是相当的有兴趣。先说点题外话，有一天晚上，坐在床上看电影，但是又到了睡觉的时间，我怕管不住自己，于是就想让电脑定时关机，印象中好像cron还是at可以执行定时任务，但是自己又不会用，打开man page看那些英文觉得太麻烦，干脆自己做一个定时器算了，于是花了半个多小时终于搞定了，但是问题是这个定时器只能处理单任务，倘若有很多任务都要定时，并及时处理超时任务该怎么办呢？在Nginx中找到了答案，于是在之前那个迷你版的Nginx上增加了这个超时的机制。不过我的实现跟Nginx的又有所区别，Nginx是用红黑树来管理定时器的，我为了简单，用了一个排序的双链表管理定时器，其实原理是一样的。

先给出这个双链的接口

1. #ifndef \_LIST\_H\_INCLUDED\_\_
2. #define \_LIST\_H\_INCLUDED\_\_
3. struct list\_node {
4. struct    list\_node        \*next, \*prev;
5. };
6. static inline void
7. list\_init(struct list\_node \*node)
8. {
9. node->next = node->prev = node;
10. }
11. static inline void
12. \_\_list\_add(struct list\_node \*prev, struct list\_node \*next, struct list\_node \*node)
13. {
14. prev->next = node;
15. node->prev = prev;
16. next->prev = node;
17. node->next = next;
18. }
19. static inline void
20. list\_add\_prev(struct list\_node \*cur, struct list\_node \*node)
21. {
22. \_\_list\_add(cur->prev, cur, node);
23. }
24. static inline void
25. list\_add\_next(struct list\_node \*cur, struct list\_node \*node)
26. {
27. \_\_list\_add(cur, cur->next, node);
28. }
29. static inline void
30. list\_add\_tail(struct list\_node \*head, struct list\_node \*node)
31. {
32. list\_add\_prev(head, node);
33. }
34. static inline void
35. list\_add\_head(struct list\_node \*head, struct list\_node \*node)
36. {
37. list\_add\_next(head, node);
38. }
39. static inline void
40. \_\_list\_del(struct list\_node \*prev, struct list\_node \*next)
41. {
42. prev->next = next;
43. next->prev = prev;
44. }
45. static inline void
46. list\_del(struct list\_node \*node)
47. {
48. \_\_list\_del(node->prev, node->next);
49. node->prev = node->next = NULL;
50. }
51. static inline int
52. list\_empty(struct list\_node \*head)
53. {
54. return head->next == head->prev && head->next == head;
55. }
56. #define offsetof(ptype, element) \
57. ((int) &(((ptype) 0)->element))
58. #define container\_of(list, ptype, element) \
59. ((ptype) ((char \*) (list) - offsetof(ptype, element)))
60. #define list\_entry(list, ptype, element) \
61. container\_of(list, ptype, element)
62. #define list\_first\_entry(head, ptype, element) \
63. list\_entry((head)->next, ptype, element)
64. #define list\_for\_each\_entry(node, head, element) \
65. for (node = list\_entry((head)->next, typeof(node), element); \
66. &(node->element) != (head); \
67. node = list\_entry(node->element.next, typeof(node), element))
68. #endif

接下来就是定时器的接口了

1. #ifndef \_TIMER\_H\_INCLUDED\_\_
2. #define \_TIMER\_H\_INCLUDED\_\_
3. #include "httpd.h"
4. struct timer {
5. int                    key;
6. struct list\_node    list;
7. };
8. static inline void
9. list\_timer\_del(struct timer \*p)
10. {
11. list\_del(&p->list);
12. }
13. static inline void
14. list\_timer\_add(struct timer \*timer, struct timer \*p)
15. {
16. struct timer \*t;
17. list\_init(&p->list);
18. list\_for\_each\_entry(t, &timer->list, list) {
19. if (t->key >= p->key) break;
20. }
21. list\_add\_prev(&t->list, &p->list);
22. }
23. static inline struct timer \*
24. list\_timer\_entry(struct list\_node \*list)
25. {
26. return list\_entry(list, struct timer \*, list);
27. }
28. static inline struct timer \*
29. list\_timer\_first\_entry(struct timer \*timer)
30. {
31. return list\_first\_entry(&timer->list, struct timer \*, list);
32. }
33. static inline int
34. list\_timer\_empty(struct timer \*timer)
35. {
36. return list\_empty(&timer->list);
37. }
38. static inline struct timer \*
39. list\_min\_timer(struct timer \*timer)
40. {
41. if (list\_timer\_empty(timer))
42. return NULL;
43. return list\_timer\_first\_entry(timer);
44. }
45. #endif
46. #include "httpd.h"
47. #define TIMER\_LAZY\_DELAY    300
48. struct timer httpd\_timer;
49. void
50. httpd\_event\_timer\_init(void)
51. {
52. list\_init(&httpd\_timer.list);
53. }
54. void
55. httpd\_timer\_del(event\_t \*ev)
56. {
57. list\_timer\_del(&ev->timer);
58. ev->timer\_set = 0;
59. }
60. void
61. httpd\_timer\_add(event\_t \*ev, int msec)
62. {
63. int     diff;
64. uint    key;
65. key = httpd\_current\_msec + msec;
67. if (ev->timer\_set) {
69. diff = key - ev->timer.key;
70. if (abs(diff) < TIMER\_LAZY\_DELAY)
71. return;
72. httpd\_timer\_del(ev);
73. }
74. ev->timer.key = key;
76. list\_timer\_add(&httpd\_timer, &ev->timer);
78. ev->timer\_set = 1;
79. }
80. uint
81. httpd\_find\_timer(void)
82. {
83. int             timer;
84. struct timer \*p;
85. if (list\_timer\_empty(&httpd\_timer))
86. return (uint\_t)-1;
87. p = list\_timer\_entry(httpd\_timer.list.next);
88. timer = p->key - httpd\_current\_msec;
89. return timer > 0 ? timer : 0;
90. }
91. void
92. httpd\_event\_expire\_timers(void)
93. {
94. struct timer     \*p;
95. event\_t            \*ev;
96. for ( ; p = list\_timer\_first\_entry(&httpd\_timer); ) {
97. if ((int)p->key - (int)httpd\_current\_msec <= 0) {
99. ev = list\_entry(p, event\_t \*, timer);
101. list\_timer\_del(p);
103. ev->timer\_set = 0;
104. ev->timedout = 1;
106. ev->handler(ev);
107. }
108. break;
109. }
110. }
111. #include "httpd.h"
112. uint\_t    httpd\_current\_msec;
113. void
114. httpd\_time\_update(void)
115. {
116. struct timeval tv;
117. gettimeofday(&tv, NULL);
118. httpd\_current\_msec = tv.tv\_sec \* 1000 + tv.tv\_usec / 1000;
119. }

我们再来看看整个处理流程是怎么样的呢？

1. #include "httpd.h"
2. int                 ep = -1;
3. int                    nevents;
4. struct epoll\_event    \*event\_list;
5. void
6. epoll\_init(void)
7. {
8. if ((ep = epoll\_create(1024)) < 0)
9. err\_sys("epoll create error");
10. event\_list = calloc(1024, sizeof(struct epoll\_event));
11. nevents = 1024;
12. }
13. void
14. epoll\_add\_event(event\_t \*ev, int event, int flag)
15. {
16. struct epoll\_event      ee;
17. struct connection    \*c;
18. c = ev->connection;
19. ee.events = event | flag;
20. ee.data.ptr = c;
21. if (epoll\_ctl(ep, EPOLL\_CTL\_ADD, c->fd, &ee) < 0)
22. err\_sys("epoll ctl error");
23. ev->active = 1;
24. }
25. void
26. epoll\_process\_events(void)
27. {
28. int              i, events, revents, timer;
29. connection\_t    \*c;
30. event\_t            \*rev, \*wev;
31. timer = httpd\_find\_timer();
32. events = epoll\_wait(ep, event\_list, nevents, timer);
33. httpd\_time\_update();
34. for (i = 0; i < events; i++) {
36. c = event\_list[i].data.ptr;
38. revents = event\_list[i].events;
40. rev = c->rev;
41. if ((revents & EPOLLIN) && rev->active) {
42. rev->handler(rev);
43. }
44. wev = c->wev;
45. if ((revents & EPOLLOUT) && wev->active) {
46. wev->handler(wev);
47. }
48. }
49. httpd\_event\_expire\_timers();
50. }

Nginx事件管理主要是网络事件和定时器事件。下面介绍定时器事件管理，即超时管理。

## 为什么要进行超时管理

Nginx有必要对可能发生超时的事件进行统一管理，并在事件超时时作出相应的处理，比如回收资源，返回错误等。举例来说，当客户端对nginx发出请求连接后，nginx机会accept()并建立对应的连接对象、读取额护短请求的头部信息。而读取这个头部信息显然是要在一定的时间内完成的。如果在一个有限的时间内没有读取到头部信息或者读取读取的头部信息不完整，那么nginx就无法进行正常处理，并且认为这是一个错误/非法的请求，直接返回错误信息并释放相应资源，如果nginx不这样做，那么针对如此的恶意攻击就和容易实施。

## 如何进行超时管理

对于超时管理，要解决两个问题：

（1）超时事件对象的组织  
nginx采用的是红黑树。

（2）超时事件对象的超时检测

有两种方案：

一种是定时检测机制，通过设置定时器，没过一定时间就对红黑树管理的所有超时时间进行一次超级扫描。

另一种是先计算出距离当前最快发生超时的时间是多久。然后等待这个时间之后再去进行一次超时检测。

### 超时事件对象的组织

nginx中封装事件的结构体ngx\_event\_s中有几个字段与超时管理联系紧密：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. //超时
2. unsigned         timedout:1;
3. //事件存在于定时器中
4. unsigned         timer\_set:1;
5. //定时器节点
6. ngx\_rbtree\_node\_t   timer;

//超时

unsigned timedout:1;

//事件存在于定时器中

unsigned timer\_set:1;

//定时器节点

ngx\_rbtree\_node\_t timer;

Nginx设置了两个全局变量以便在程序的任何地方都可以哀诉的访问到这棵红棵树（src/event/ngx\_event\_timer.c）：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. ngx\_thread\_volatile ngx\_rbtree\_t  ngx\_event\_timer\_rbtree;//超时管理的红黑树结构
2. static ngx\_rbtree\_node\_t          ngx\_event\_timer\_sentinel;//红黑树中的哨兵节点

ngx\_thread\_volatile ngx\_rbtree\_t ngx\_event\_timer\_rbtree;//超时管理的红黑树结构

static ngx\_rbtree\_node\_t ngx\_event\_timer\_sentinel;//红黑树中的哨兵节点

红黑树结构的初始化

红黑树的初始化函数ngx\_event\_timer\_init()是在函数ngx\_event\_process\_init()函数内被调用的。即每个worker进程会在自身的初始化时建立这棵红黑树。

源代码（src/eventngx\_event.c）：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. static ngx\_int\_t
2. ngx\_event\_process\_init(ngx\_cycle\_t \*cycle)
3. {
4. ....
5. if (ngx\_event\_timer\_init(cycle->log) == NGX\_ERROR) {
6. return NGX\_ERROR;
7. }
8. ...
9. }

static ngx\_int\_t

ngx\_event\_process\_init(ngx\_cycle\_t \*cycle)

{

....

if (ngx\_event\_timer\_init(cycle->log) == NGX\_ERROR) {

return NGX\_ERROR;

}

...

}

函数ngx\_event\_timer\_init完成了红黑树结构的初始化：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. /\*
2. \* the event timer rbtree may contain the duplicate keys, however,
3. \* it should not be a problem, because we use the rbtree to find
4. \* a minimum timer value only
5. \*/
7. ngx\_int\_t
8. ngx\_event\_timer\_init(ngx\_log\_t \*log)
9. {
10. //红黑树初始化
11. ngx\_rbtree\_init(&ngx\_event\_timer\_rbtree, &ngx\_event\_timer\_sentinel,
12. ngx\_rbtree\_insert\_timer\_value);
14. //多线程处理
15. #if (NGX\_THREADS)
17. if (ngx\_event\_timer\_mutex) {
18. ngx\_event\_timer\_mutex->log = log;
19. return NGX\_OK;
20. }
22. ngx\_event\_timer\_mutex = ngx\_mutex\_init(log, 0);
23. if (ngx\_event\_timer\_mutex == NULL) {
24. return NGX\_ERROR;
25. }
27. #endif
29. return NGX\_OK;
30. }

/\*

\* the event timer rbtree may contain the duplicate keys, however,

\* it should not be a problem, because we use the rbtree to find

\* a minimum timer value only

\*/

ngx\_int\_t

ngx\_event\_timer\_init(ngx\_log\_t \*log)

{

//红黑树初始化

ngx\_rbtree\_init(&ngx\_event\_timer\_rbtree, &ngx\_event\_timer\_sentinel,

ngx\_rbtree\_insert\_timer\_value);

//多线程处理

#if (NGX\_THREADS)

if (ngx\_event\_timer\_mutex) {

ngx\_event\_timer\_mutex->log = log;

return NGX\_OK;

}

ngx\_event\_timer\_mutex = ngx\_mutex\_init(log, 0);

if (ngx\_event\_timer\_mutex == NULL) {

return NGX\_ERROR;

}

#endif

return NGX\_OK;

}

而ngx\_rbtree\_init(tree, s, i) 是宏定义（src/core/ngx\_rbtree.h），即新建了一棵空的红黑树：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. #define ngx\_rbtree\_init(tree, s, i)                                           \
2. ngx\_rbtree\_sentinel\_init(s);                                              \
3. (tree)->root = s;                                                         \
4. (tree)->sentinel = s;                                                     \
5. (tree)->insert = i

#define ngx\_rbtree\_init(tree, s, i) \

ngx\_rbtree\_sentinel\_init(s); \

(tree)->root = s; \

(tree)->sentinel = s; \

(tree)->insert = i

### 对事件进行超时监控

当需要对某个事件进行超时监控时，就会把它加入到这个红黑树内。比如，nginx调用accept接受到客户端的请求并建立对应的连接对象connection后，在连接对象的初始化函数ngx\_http\_init\_connection()内，可以找到这样的代码：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. void
2. ngx\_http\_init\_connection(ngx\_connection\_t \*c)
3. {
4. ...
5. (358L)ngx\_add\_timer(rev, c->listening->post\_accept\_timeout);
6. ...
7. }

void

ngx\_http\_init\_connection(ngx\_connection\_t \*c)

{

...

(358L)ngx\_add\_timer(rev, c->listening->post\_accept\_timeout);

...

}

ngx\_add\_timer的第一个参数是事件对象，第二个参数是超时时限。

### 超时检测

Nginx具体使用哪一种超时检测方案取决于配置项timer\_resolution，比如：

timer\_resolution 100ms；

反应到nginx代码中就是全局变量ngx\_timer\_resolution的值为100；

下面我们看worker进程的核心处理函数ngx\_process\_events\_and\_timers（ngx\_event.c）：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. void
2. ngx\_process\_events\_and\_timers(ngx\_cycle\_t \*cycle)
3. {
4. ngx\_uint\_t  flags;
5. ngx\_msec\_t  timer, delta;
7. if (ngx\_timer\_resolution) {
8. timer = NGX\_TIMER\_INFINITE;
9. flags = 0;
11. } else {
12. timer = ngx\_event\_find\_timer();//将超时检测时间设置为最快发生超时的事件对象的超时时刻与当前时刻之差
13. flags = NGX\_UPDATE\_TIME;
14. ...
15. (void) ngx\_process\_events(cycle, timer, flags);
16. ...
17. }

void

ngx\_process\_events\_and\_timers(ngx\_cycle\_t \*cycle)

{

ngx\_uint\_t flags;

ngx\_msec\_t timer, delta;

if (ngx\_timer\_resolution) {

timer = NGX\_TIMER\_INFINITE;

flags = 0;

} else {

timer = ngx\_event\_find\_timer();//将超时检测时间设置为最快发生超时的事件对象的超时时刻与当前时刻之差

flags = NGX\_UPDATE\_TIME;

...

(void) ngx\_process\_events(cycle, timer, flags);

...

}

可以看到，ngx\_timer\_resolution是否为0影响了两个值timer和flags。

当ngx\_timer\_resolution不为0时，即方案1.timer为无限大。timer在函数ngx\_process\_events()内被用作事件机制被阻塞的最长时间。那么timer为无限大会不会导致事件处理机制无限等待而超时事件得不到及时处理呢？不会。因为正常情况下事件处理机制会监控到某些I/O事件的发生。即便是服务器太闲，没有任何I/O事件发生，工作进程也不会无限等待。因为工作进程一开始就设置好了一个定时器，这实现在初始化函数ngx\_event\_process\_init()内，看代码：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. static ngx\_int\_t
2. ngx\_event\_process\_init(ngx\_cycle\_t \*cycle)
3. {
4. ...
5. sa.sa\_handler = ngx\_timer\_signal\_handler;
6. sigemptyset(&sa.sa\_mask);
7. itv.it\_interval.tv\_sec = ngx\_timer\_resolution / 1000;
8. itv.it\_interval.tv\_usec = (ngx\_timer\_resolution % 1000) \* 1000;
9. itv.it\_value.tv\_sec = ngx\_timer\_resolution / 1000;
10. itv.it\_value.tv\_usec = (ngx\_timer\_resolution % 1000 ) \* 1000;
12. if (setitimer(ITIMER\_REAL, &itv, NULL) == -1) {
13. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
14. "setitimer() failed");
15. }
16. ....
17. }

static ngx\_int\_t

ngx\_event\_process\_init(ngx\_cycle\_t \*cycle)

{

...

sa.sa\_handler = ngx\_timer\_signal\_handler;

sigemptyset(&sa.sa\_mask);

itv.it\_interval.tv\_sec = ngx\_timer\_resolution / 1000;

itv.it\_interval.tv\_usec = (ngx\_timer\_resolution % 1000) \* 1000;

itv.it\_value.tv\_sec = ngx\_timer\_resolution / 1000;

itv.it\_value.tv\_usec = (ngx\_timer\_resolution % 1000 ) \* 1000;

if (setitimer(ITIMER\_REAL, &itv, NULL) == -1) {

ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,

"setitimer() failed");

}

....

}

回调函数ngx\_timer\_signal\_handler：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. static void
2. ngx\_timer\_signal\_handler(int signo)
3. {
4. ngx\_event\_timer\_alarm = 1;
6. #if 1
7. ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, ngx\_cycle->log, 0, "timer signal");
8. #endif
9. }

static void

ngx\_timer\_signal\_handler(int signo)

{

ngx\_event\_timer\_alarm = 1;

#if 1

ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, ngx\_cycle->log, 0, "timer signal");

#endif

}

可以看出它仅仅是将标志变量ngx\_event\_timer\_alarm 设置为1.

只有在ngx\_event\_timer\_alarm 为1 的情况下，工作进程才会更新时间。函数ngx\_epoll\_process\_events中的代码（ngx\_epoll\_module.c）：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. static ngx\_int\_t
2. ngx\_epoll\_process\_events(ngx\_cycle\_t \*cycle, ngx\_msec\_t timer, ngx\_uint\_t flags)
3. {
4. ...
5. events = epoll\_wait(ep, event\_list, (int) nevents, timer);
7. err = (events == -1) ? ngx\_errno : 0;
9. if (flags & NGX\_UPDATE\_TIME || ngx\_event\_timer\_alarm) {
10. ngx\_time\_update();
11. }
12. ...
13. }

static ngx\_int\_t

ngx\_epoll\_process\_events(ngx\_cycle\_t \*cycle, ngx\_msec\_t timer, ngx\_uint\_t flags)

{

...

events = epoll\_wait(ep, event\_list, (int) nevents, timer);

err = (events == -1) ? ngx\_errno : 0;

if (flags & NGX\_UPDATE\_TIME || ngx\_event\_timer\_alarm) {

ngx\_time\_update();

}

...

}

在方案一的情况下，||前面的式子为假，那么ngx\_event\_timer\_alarm 不为1 的情况下，更新函数ngx\_time\_update()不会被执行。那么会导致超时检测函数ngx\_event\_expire\_timers不会被执行。看ngx\_process\_events\_and\_timers函数的代码（ngx\_event.c）：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. void
2. ngx\_process\_events\_and\_timers(ngx\_cycle\_t \*cycle)
3. {
4. ...
5. delta = ngx\_current\_msec;
6. (void) ngx\_process\_events(cycle, timer, flags);//事件处理函数
7. delta = ngx\_current\_msec - delta;
8. ...
9. if (delta) {
10. ngx\_event\_expire\_timers();//超时检测函数
11. }
12. ...
13. }

void

ngx\_process\_events\_and\_timers(ngx\_cycle\_t \*cycle)

{

...

delta = ngx\_current\_msec;

(void) ngx\_process\_events(cycle, timer, flags);//事件处理函数

delta = ngx\_current\_msec - delta;

...

if (delta) {

ngx\_event\_expire\_timers();//超时检测函数

}

...

}

当ngx\_timer\_resolution为0时，执行方案2。timer设置为最快发生超时的事件对象的超时时刻与当前时刻的时间差。具体计算时在函数ngx\_event\_find\_timer内（ngx\_event\_timer.c）。

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. ngx\_msec\_t
2. ngx\_event\_find\_timer(void)
3. {
4. ngx\_msec\_int\_t      timer;
5. ngx\_rbtree\_node\_t  \*node, \*root, \*sentinel;
7. if (ngx\_event\_timer\_rbtree.root == &ngx\_event\_timer\_sentinel) {
8. return NGX\_TIMER\_INFINITE;
9. }
11. ngx\_mutex\_lock(ngx\_event\_timer\_mutex);
13. root = ngx\_event\_timer\_rbtree.root;
14. sentinel = ngx\_event\_timer\_rbtree.sentinel;
16. node = ngx\_rbtree\_min(root, sentinel);
18. ngx\_mutex\_unlock(ngx\_event\_timer\_mutex);
20. timer = (ngx\_msec\_int\_t) (node->key - ngx\_current\_msec);
22. return (ngx\_msec\_t) (timer > 0 ? timer : 0);
23. }

ngx\_msec\_t

ngx\_event\_find\_timer(void)

{

ngx\_msec\_int\_t timer;

ngx\_rbtree\_node\_t \*node, \*root, \*sentinel;

if (ngx\_event\_timer\_rbtree.root == &ngx\_event\_timer\_sentinel) {

return NGX\_TIMER\_INFINITE;

}

ngx\_mutex\_lock(ngx\_event\_timer\_mutex);

root = ngx\_event\_timer\_rbtree.root;

sentinel = ngx\_event\_timer\_rbtree.sentinel;

node = ngx\_rbtree\_min(root, sentinel);

ngx\_mutex\_unlock(ngx\_event\_timer\_mutex);

timer = (ngx\_msec\_int\_t) (node->key - ngx\_current\_msec);

return (ngx\_msec\_t) (timer > 0 ? timer : 0);

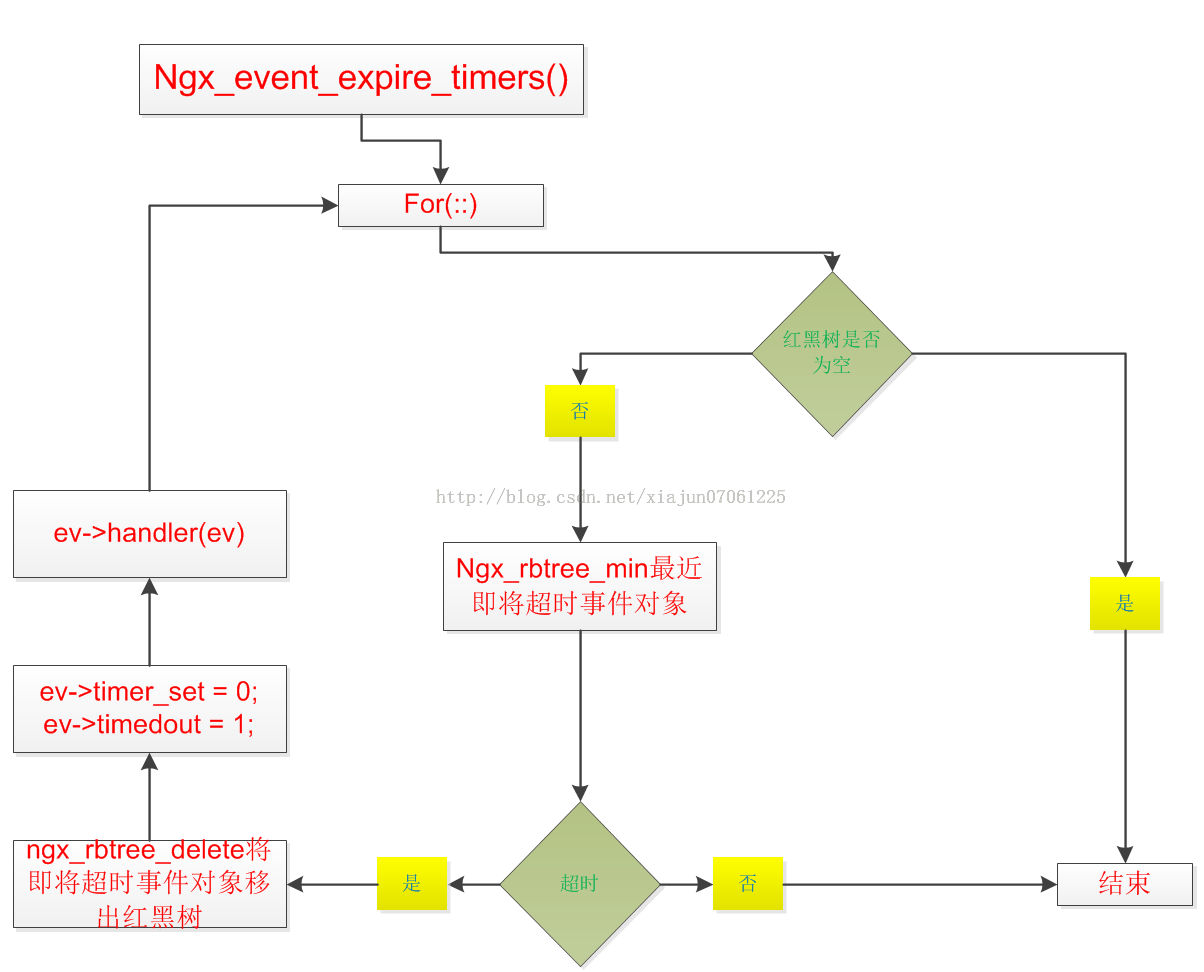
}

该函数从红黑树中找到key值最小的节点，然后用key值减去当前时刻即得到预期timer值。这个值可能是负数，表示已经有事件超时了。因此直接将其设置为0.那么事件处理 机制在开始监控I/O事件时会立即返回，以便马上处理这些超时事件。同时flags被设置为NGX\_UPDATE\_TIME。从ngx\_epoll\_process\_events函数的代码中可以看出ngx\_time\_update()将被执行，事件被更新。即事件处理机制每次返回都会更新时间。如果I/O事件比较多，那么会导致比较频繁地调用gettimeofday()系统函数，这也可以说是超时检测方案2对性能的最大影响。这个时候超时检测函数ngx\_event\_expire\_timers()函数会被执行。

下面看看超时检测函数ngx\_event\_expire\_timers。其完成的主要任务是对超时事件对象是否超时进行扫描检测以及对已超时事件对象的处理。

检测是否有事件对象超时不需遍历扫描所有超时时间对象，而是找到最近的即将超时的超时事件对象。判断其是否超时，如果超时，则将其移出红黑树，设置超时标记，调用回调函数进行处理。之后再判断第二近的即将超时的超时事件对象，如此反复，知道某个超时事件对象还未超时或所有超时事件对象都已超时并处理完毕就结束检测。

此函数流程图：



下面是其核心代码：

**[cpp]** [view plain](http://blog.csdn.net/xiajun07061225/article/details/9284543) [copy](http://blog.csdn.net/xiajun07061225/article/details/9284543)

[print?](http://blog.csdn.net/xiajun07061225/article/details/9284543)

1. void
2. ngx\_event\_expire\_timers(void)
3. {
4. ngx\_event\_t        \*ev;
5. ngx\_rbtree\_node\_t  \*node, \*root, \*sentinel;
7. sentinel = ngx\_event\_timer\_rbtree.sentinel;
8. //循环检测
9. for ( ;; ) {
10. ngx\_mutex\_lock(ngx\_event\_timer\_mutex);
11. root = ngx\_event\_timer\_rbtree.root;
12. if (root == sentinel) {
13. return;
14. }
16. //找到最近的即将超时的超时事件对象
17. node = ngx\_rbtree\_min(root, sentinel);
19. /\* node->key <= ngx\_current\_time \*/
20. //如果已经超时
21. if ((ngx\_msec\_int\_t) (node->key - ngx\_current\_msec) <= 0) {
22. ev = (ngx\_event\_t \*) ((char \*) node - offsetof(ngx\_event\_t, timer));
24. ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, ev->log, 0,
25. "event timer del: %d: %M",
26. ngx\_event\_ident(ev->data), ev->timer.key);
28. //从红黑树中移除这个已超时的超时事件对象
29. ngx\_rbtree\_delete(&ngx\_event\_timer\_rbtree, &ev->timer);
30. ngx\_mutex\_unlock(ngx\_event\_timer\_mutex);
32. #if (NGX\_DEBUG)
33. ev->timer.left = NULL;
34. ev->timer.right = NULL;
35. ev->timer.parent = NULL;
36. #endif
38. //标记：是否已加入红黑树超时管理
39. ev->timer\_set = 0;
40. //标记：是否超时
41. ev->timedout = 1;
42. //调用回调函数
43. ev->handler(ev);
45. continue;
46. }
47. break;
48. }
50. ngx\_mutex\_unlock(ngx\_event\_timer\_mutex);
51. }