<https://blog.csdn.net/luotuo44/article/details/38661787>

转载请注明出处: [**http://blog.csdn.net/luotuo44/article/details/38661787**](http://blog.csdn.net/luotuo44/article/details/38661787)

# 基本时间操作函数：

        Libevent采用的时间类型是struct  timeval，这个类型在很多平台都提供了。此外，Libevent还提供了一系列的时间操作函数。比如两个struct timeval相加、相减、比较大小。有些平台直接提供了一些时间操作函数，但有些则没有，那么Libevent就自己实现。这些宏如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38661787) [copy](https://blog.csdn.net/luotuo44/article/details/38661787)

1. #ifdef \_EVENT\_HAVE\_TIMERADD
2. #define evutil\_timeradd(tvp, uvp, vvp) timeradd((tvp), (uvp), (vvp))
3. #define evutil\_timersub(tvp, uvp, vvp) timersub((tvp), (uvp), (vvp))
4. #else
5. #define evutil\_timeradd(tvp, uvp, vvp)                  \
6. **do** {                                \
7. (vvp)->tv\_sec = (tvp)->tv\_sec + (uvp)->tv\_sec;     \
8. (vvp)->tv\_usec = (tvp)->tv\_usec + (uvp)->tv\_usec;       \
9. **if** ((vvp)->tv\_usec >= 1000000) {          \
10. (vvp)->tv\_sec++;             \
11. (vvp)->tv\_usec -= 1000000;           \
12. }                           \
13. } **while** (0)
14. #define evutil\_timersub(tvp, uvp, vvp)                  \
15. **do** {                                \
16. (vvp)->tv\_sec = (tvp)->tv\_sec - (uvp)->tv\_sec;     \
17. (vvp)->tv\_usec = (tvp)->tv\_usec - (uvp)->tv\_usec;  \
18. **if** ((vvp)->tv\_usec < 0) {             \
19. (vvp)->tv\_sec--;             \
20. (vvp)->tv\_usec += 1000000;           \
21. }                           \
22. } **while** (0)
23. #endif
25. #ifdef \_EVENT\_HAVE\_TIMERCLEAR
26. #define evutil\_timerclear(tvp) timerclear(tvp)
27. #else
28. #define evutil\_timerclear(tvp)  (tvp)->tv\_sec = (tvp)->tv\_usec = 0
29. #endif

32. #define evutil\_timercmp(tvp, uvp, cmp)                  \
33. (((tvp)->tv\_sec == (uvp)->tv\_sec) ?               \
34. ((tvp)->tv\_usec cmp (uvp)->tv\_usec) :                \
35. ((tvp)->tv\_sec cmp (uvp)->tv\_sec))
37. #ifdef \_EVENT\_HAVE\_TIMERISSET
38. #define evutil\_timerisset(tvp) timerisset(tvp)
39. #else
40. #define evutil\_timerisset(tvp)  ((tvp)->tv\_sec || (tvp)->tv\_usec)
41. #endif

        代码中的那些条件宏，是在配置Libevent的时候检查所在的系统环境而定义的。具体的内容，可以参考《[event-config.h指明所在系统的环境](http://blog.csdn.net/luotuo44/article/details/38300965)》一文。

        Libevent的时间一般是用在超时event的。对于超时event，用户只需给出一个超时时间，比如多少秒，而不是一个绝对时间。但在Libevent内部，要将这个时间转换成绝对时间。所以在Libevent内部会经常获取系统时间(绝对时间)，然后进行一些处理，比如，转换、比较。

# cache时间：

        Libevent封装了一个evutil\_gettimeofday函数用来获取系统时间，该函数在POSIX的系统是直接调用gettimeofday函数，在Windows系统是通过\_ftime函数。虽然gettimeofday的[耗时成本不大](http://blog.csdn.net/russell_tao/article/details/7185588" \t "_blank)，不过Libevent还是使用了一个cache保存时间，使得更加高效。在event\_base结构体有一个struct timeval类型的cache变量 tv\_cache。处理超时event的两个函数event\_add\_internal和event\_base\_loop内部都是调用gettime函数获取时间的。gettime函数如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38661787) [copy](https://blog.csdn.net/luotuo44/article/details/38661787)

1. //event.c文件
2. **static** **int**
3. gettime(**struct** event\_base \*base, **struct** timeval \*tp)
4. {
5. **if** (base->tv\_cache.tv\_sec) { //cache可用
6. \*tp = base->tv\_cache;
7. **return** (0);
8. }
10. …//没有cache的时候就使用其他方式获取时间
11. }

        从上面代码可以看到，Libevent优先使用cache时间。tv\_bache变量处理作为cache外，还有另外一个作用，下面会讲到。

        cache的时间也是通过调用系统的提供的时间函数得到的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38661787) [copy](https://blog.csdn.net/luotuo44/article/details/38661787)

1. //event.c文件
2. **static** **inline** **void**
3. update\_time\_cache(**struct** event\_base \*base)
4. {
5. base->tv\_cache.tv\_sec = 0;
6. **if** (!(base->flags & EVENT\_BASE\_FLAG\_NO\_CACHE\_TIME))
7. gettime(base, &base->tv\_cache);
8. }

        tv\_cache是通过调用gettime来获取时间。由于tv\_cache.tv\_sec已经赋值为0，所以它将使用系统提供的时间函数得到时间。代码也展示了，如果event\_base的配置中定义了EVENT\_BASE\_FLAG\_NO\_CACHE\_TIME宏，将不能使用cache时间。关于这个宏的设置可以参考《[配置event\_base](http://blog.csdn.net/luotuo44/article/details/38443569#t5)》一文。

# 处理用户手动修改系统时间：

        如果用户能老老实实，或许代码就不需要写得很复杂。由于用户的不老实，所以有时候要考虑很多很特殊的情况。在Libevent的时间管理这方面也是如此。

        Libevent在实际使用时还有一个坑爹的现象，那就是，用户手动把时钟(wall time)往回调了。比如说现在是上午9点，但用户却把OS的系统时间调成了上午7点。这是很坑爹的。对于超时event和event\_add的第二个参数，都是一个时间长度。但在内部Libevent要把这个时间转换成绝对时间。

         如果用户手动修改了OS的系统时间。那么Libevent把超时时间长度转换成绝对时间将是弄巧成拙。拿上面的时间例子。如果用户设置的超时为1分钟。那么到了9：01就会超时。如果用户把系统时间调成了7点，那么要过2个小时01分才能发生超时。这就和用户原先的设置差得很远了。

        读者可能会说，这个责任应该是由用户负。呵呵，但Libevent提供的函数接口是一个时间长度，既然是时间长度，那么无论用户怎么改变OS的系统时间，这个时间长度都是相对于event\_add ()被调用的那一刻算起，这是不会变的。如果Libevent做不到这一点，这说明是Libevent没有遵循接口要求。

        为此，Libevent提出了一些解决方案。

## 使用monotonic时间：

        问题的由来是因为用户能修改系统时间，所以最简单的解决方案就是能获取到一个用户不能修改的时间，然后以之为绝对时间。因为event\_add提供给用户的接口使用的是一个时间长度，所以无论是使用哪个绝对时间都是无所谓的。

        基于这一点，Libevent找到了monotonic时间，从字面来看monotonic翻译成单调。我们高中学过的单调函数英文名就是monotonic function。monotonic时间就像单调递增函数那样，只增不减的，没有人能手动修改之。

        monotonic时间是boot启动后到现在的时间。用户是不能修改这个时间。如果Libevent所在的系统支持monotonic时间的话，那么Libevent就会选用这个monotonic时间为绝对时间。

        首先，Libevent检查所在的系统是否支持monotonic时间。在event\_base\_new\_with\_config函数中会调用detect\_monotonic函数检测。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38661787) [copy](https://blog.csdn.net/luotuo44/article/details/38661787)

1. //event.c文件
2. **static** **void**
3. detect\_monotonic(**void**)
4. {
5. #if defined(\_EVENT\_HAVE\_CLOCK\_GETTIME) && defined(CLOCK\_MONOTONIC)
6. **struct** timespec ts;
7. **static** **int** use\_monotonic\_initialized = 0;
9. **if** (use\_monotonic\_initialized)
10. **return**;
12. **if** (clock\_gettime(CLOCK\_MONOTONIC, &ts) == 0)
13. use\_monotonic = 1; //系统支持monotonic时间
14. use\_monotonic\_initialized = 1;
15. #endif
16. }

        从上面代码可以看到，如果Libevent所在的系统支持monotonic时间，就将全局变量use\_monotonic赋值1，作为标志。

        如果Libevent所在的系统支持monotonic时间，那么Libevent将使用monotonic时间，也就是说Libevent用于获取系统时间的函数gettime将由monotonic提供时间。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38661787) [copy](https://blog.csdn.net/luotuo44/article/details/38661787)

1. //event.c文件
2. **static** **int**
3. gettime(**struct** event\_base \*base, **struct** timeval \*tp)
4. {
5. EVENT\_BASE\_ASSERT\_LOCKED(base);
7. **if** (base->tv\_cache.tv\_sec) {
8. \*tp = base->tv\_cache;
9. **return** (0);
10. }
12. #if defined(\_EVENT\_HAVE\_CLOCK\_GETTIME) && defined(CLOCK\_MONOTONIC)
13. **if** (use\_monotonic) {
14. **struct** timespec ts;
16. **if** (clock\_gettime(CLOCK\_MONOTONIC, &ts) == -1)
17. **return** (-1);
19. tp->tv\_sec = ts.tv\_sec;
20. tp->tv\_usec = ts.tv\_nsec / 1000;
22. //额外的功能
23. **if** (base->last\_updated\_clock\_diff + CLOCK\_SYNC\_INTERVAL
24. < ts.tv\_sec) {
25. **struct** timeval tv;
26. evutil\_gettimeofday(&tv,NULL);
27. //tv\_clock\_diff记录两种时间的时间差
28. evutil\_timersub(&tv, tp, &base->tv\_clock\_diff);
29. base->last\_updated\_clock\_diff = ts.tv\_sec;
30. }
32. **return** (0);
33. }
34. #endif
36. //如果所在的系统不支持monotonic时间，那么只能使用evutil\_gettimeofday了
37. **return** (evutil\_gettimeofday(tp, NULL));
38. }

        上面的代码虽然首先是使用cache时间，但实际上event\_base结构体的cache时间也是通过调用gettime函数而得到的。上面代码也可以看到：如果所在的系统没有提供monotonic时间，那么就只能使用evutil\_gettimeofday这个函数提供的系统时间了。

        从上面的分析可知，如果Libevent所在的系统支持monotonic时间，那么根本就不用考虑用户手动修改系统时间这坑爹的事情。但如果所在的系统没有支持monotonic时间，那么Libevent就只能使用evutil\_gettimeofday获取一个用户能修改的时间。

## 尽可能精确记录时间差：

        现在来看一下Libevent在这种情况下在怎么解决这个坑爹得的问题。

         Libevent给出的方案是，尽可能精确地计算 用户往回调了多长时间。如果知道了用户往回调了多长时间，那么将小根堆中的全部event的时间都往回调一样的时间即可。Libevent调用timeout\_correct函数处理这个问题。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38661787) [copy](https://blog.csdn.net/luotuo44/article/details/38661787)

1. //event.c文件
2. **static** **void**
3. timeout\_correct(**struct** event\_base \*base, **struct** timeval \*tv)
4. {
5. /\* Caller must hold th\_base\_lock. \*/
6. **struct** event \*\*pev;
7. unsigned **int** size;
8. **struct** timeval off;
9. **int** i;
11. //如果系统支持monotonic时间，那么就不需要校准时间了
12. **if** (use\_monotonic)
13. **return**;
15. //获取现在的系统时间
16. gettime(base, tv);

19. //tv的时间更大，说明用户没有往回调系统时间。那么不需要处理
20. **if** (evutil\_timercmp(tv, &base->event\_tv, >=)) {
21. base->event\_tv = \*tv;
22. **return**;
23. }
25. evutil\_timersub(&base->event\_tv, tv, &off);//off差值，即用户调小了多少
27. pev = base->timeheap.p;
28. size = base->timeheap.n;
29. //用户已经修改了OS的系统时间。现在需要对小根堆的所有event
30. //都修改时间。使得之适应新的系统时间
31. **for** (; size-- > 0; ++pev) {
32. **struct** timeval \*ev\_tv = &(\*\*pev).ev\_timeout;
33. //前面已经用off保存了，用户调小了多少。现在只需
34. //将小根堆的所有event的超时时间(绝对时间)都减去这个off即可
35. evutil\_timersub(ev\_tv, &off, ev\_tv);
36. }
38. //保存现在的系统时间。以防用户再次修改系统时间
39. base->event\_tv = \*tv;
40. }

        Libevent用event\_base的成员变量event\_tv保存用户修改系统时间前的系统时间。如果刚保存完，用户就修改系统时间，这样就能精确地计算出用户往回调了多长时间。但毕竟Libevent是用户态的库，不能做到用户修改系统时间前的一刻保存系统时间。

        于是Libevent采用多采点的方式，即时不时就保存一次系统时间。所以在event\_base\_loop函数中的while循环体里面会有gettime(base, &base->event\_tv);这是为了能多采点。但这个while循环里面还会执行多路IO复用函数和处理被激活event的回调函数(这个回调函数执行多久也是个未知数)。这两个函数的执行需要的时间可能会比较长，如果用户刚才是在执行完这两个函数之后修改系统时间，那么event\_tv保存的时间就不怎么精确了。这也是没有办法的啊！！唉！！

        下面贴出event\_base\_loop函数

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38661787) [copy](https://blog.csdn.net/luotuo44/article/details/38661787)

1. //event.c文件
2. **int**
3. event\_base\_loop(**struct** event\_base \*base, **int** flags)
4. {
5. **const** **struct** eventop \*evsel = base->evsel;
6. **struct** timeval tv;
7. **struct** timeval \*tv\_p;
8. **int** res, done, retval = 0;
10. //要使用cache时间，得在配置event\_base时，没有加入EVENT\_BASE\_FLAG\_NO\_CACHE\_TIME选项
11. clear\_time\_cache(base);

14. **while** (!done) {
15. timeout\_correct(base, &tv);
17. tv\_p = &tv;
18. **if** (!N\_ACTIVE\_CALLBACKS(base) && !(flags & EVLOOP\_NONBLOCK)) {
19. //参考http://blog.csdn.net/luotuo44/article/details/38637671
20. timeout\_next(base, &tv\_p); //获取dispatch的最大等待时间
21. } **else** {
22. evutil\_timerclear(&tv);
23. }
25. //保存系统时间。如果有cache，将保存cache时间。
26. gettime(base, &base->event\_tv);
28. //之所以要在进入dispatch之前清零，是因为进入
29. //dispatch后，可能会等待一段时间。cache就没有意义了。
30. //如果第二个线程此时想add一个event到这个event\_base里面，在
31. //event\_add\_internal函数中会调用gettime。如果cache不清零，
32. //那么将会取这个cache时间。这将取一个不准确的时间。
33. clear\_time\_cache(base);
35. //多路IO复用函数
36. res = evsel->dispatch(base, tv\_p);
38. //将系统时间赋值到cache中
39. update\_time\_cache(base);
41. //处理超时事件。参考http://blog.csdn.net/luotuo44/article/details/38637671
42. timeout\_process(base);
44. **if** (N\_ACTIVE\_CALLBACKS(base)) {
45. **int** n = event\_process\_active(base);//处理激活event
46. }
47. }
49. **return** (retval);
50. }

        可以看到，在dispatch和event\_process\_active之间有一个update\_time\_cache。而前面的gettime(base,&base->event\_tv);实际上取的就是cache的时间。所以，如果该Libevent支持cache的话，会精确那么一些。一般来说，用户为event设置的回调函数，不应该执行太久的时间。这也是tv\_cache时间的另外一个作用。

## 出现的bug：

        由于Libevent的解决方法并不是很精确，所以还是会有一些bug。下面给出一个bug。如果用户是在调用event\_new函数之后，event\_add之前对系统时间进行修改，那么无论用户设置的event超时有多长，都会马上触发超时。下面给出实际的例子。这个例子要运行在不支持monotonic时间的系统，我是在Windows运行的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38661787) [copy](https://blog.csdn.net/luotuo44/article/details/38661787)

1. #include <event2/event.h>
2. #include<stdio.h>

5. **void** timeout\_cb(**int** fd, **short** event, **void** \*arg)
6. {
7. printf("in the timeout\_cb\n");
8. }

11. **int** main()
12. {
13. **struct** event\_base \*base = event\_base\_new();
15. **struct** event \*ev = event\_new(base, -1, EV\_TIMEOUT, timeout\_cb, NULL);
17. **int** ch;
18. //暂停，让用户有时间修改系统时间。可以将系统时间往前1个小时
19. scanf("%c", &ch);
21. **struct** timeval tv = {100, 0};//这个超时时长要比较长。这里取100秒
22. //第二个参数不能为NULL.不然也是不能触发超时的。毕竟没有时间
23. event\_add(ev, &tv);
25. event\_base\_dispatch(base);
27. **return** 0;
28. }

        这个bug的出现是因为，在event\_base\_new\_with\_config函数中有gettime(base,&base->event\_tv)，所以event\_tv记录了修改前的时间。而event\_add是在修改系统时间后才调用的。所以event结构体的ev\_timeout变量使用的是修改系统时间后的超时时间，这是正确的时间。在执行timeout\_correct函数时，Libevent发现用户修改了系统时间，所以就将本来正确的ev\_timeout减去了off。所以ev\_timeout就变得比较修改后的系统时间小了。在后面检查超时时，就会发现该event已经超时了(实际是没有超时)，就把它触发。

        如果该event有EV\_PERSIST属性，那么之后的超时则会是正确的。这个留给读者去分析吧。

        另外，Libevent并没有考虑把时钟往后调，比如现在是9点，用户把系统时间调成10点。上面的代码如果用户是在event\_add之后修改系统时间，就能发现这个bug。

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/38661787