**ev\_timer**

Libev中的超时监视器ev\_timer，是简单的相对时间定时器，它会在给定的时间之后触发超时事件，还可以在固定的时间间隔之后再次触发超时事件。

所谓的相对时间，指的是如果你注册了一个1小时的超时事件，然后调整系统时间到了去年的一月份，该超时事件依然会在1个小时之后触发。

**一：数据结构**

1：超时监视器ev\_timer结构：

**typedef** struct ev\_timer

**{**

int active**;**

int pending**;**

int priority**;**

void **\***data**;**

void **(\***cb**)(**struct ev\_loop **\***loop**,** struct ev\_timer **\***w**,** int revents**);**

ev\_tstamp at**;**

ev\_tstamp repeat**;** */\* rw \*/*

**}** ev\_timer**;**

其中的前五个成员是监视器的公共成员，其中的active在超时监视器中有特殊作用，那就是标明该监视器在堆数组timers中的下标。后两个成员at和repeat是ev\_timer特有的。at表明定时器第一次触发的时间点，这是根据mn\_now设置的，repeat必须大于等于0，它表示每隔repeat秒，该定时器再次触发。如果repeat为0，表明该定时器只触发一次。

2：ev\_watcher\_time结构

**typedef** struct ev\_watcher\_time

**{**

int active**;**

int pending**;**

int priority**;**

void **\***data**;**

void **(\***cb**)(**struct ev\_loop **\***loop**,** struct ev\_watcher\_time **\***w**,** int revents**);**

ev\_tstamp at**;**

**}** ev\_watcher\_time**;**

**typedef** ev\_watcher\_time **\***WT**;**

ev\_watcher\_time的结构与ev\_timer几乎一样，只是少了最后一个成员。该结构其实是ev\_timer和ev\_periodic的父类，它包含了ev\_timer和ev\_periodic的共有成员。

3：堆元素ANHE

#if EV\_HEAP\_CACHE\_AT

**typedef** struct

**{**

ev\_tstamp at**;**

WT w**;**

**}** ANHE**;**

#else

**typedef** WT ANHE**;**

#endif

宏EV\_HEAP\_CACHE\_AT的作用，是为了提高在堆中的缓存利用率，如果没有定义该宏，堆元素就是指向ev\_watcher\_time结构的指针。如果定义了该宏，则还将堆元素的关键成员at进行缓存。

**二：超时监视器函数**

1：设置超时监视器ev\_timer\_set

#define ev\_timer\_set(ev, after\_, repeat\_) do {\

((ev\_watcher\_time \*)(ev))->at = (after\_); \

(ev)->repeat = (repeat\_); \

} while (0)

2：启动超时监视器ev\_timer\_start

#if EV\_HEAP\_CACHE\_AT

#define ANHE\_w(he) (he).w

#define ANHE\_at(he) (he).at

#define ANHE\_at\_cache(he) (he).at = (he).w->at

#else

#define ANHE\_w(he) (he)

#define ANHE\_at(he) (he)->at

#define ANHE\_at\_cache(he)

#endif

void ev\_timer\_start **(**struct ev\_loop **\***loop**,** ev\_timer **\***w**)**

**{**

**if** **(**expect\_false **(**ev\_is\_active **(**w**)))**

**return;**

ev\_at **(**w**)** **+=** mn\_now**;**

assert **((**"libev: ev\_timer\_start called with negative timer repeat value"**,** w**->**repeat **>=** 0.**));**

**++**timercnt**;**

ev\_start **(**EV\_A\_ **(**W**)**w**,** timercnt **+** HEAP0 **-** 1**);**

array\_needsize **(**ANHE**,** timers**,** timermax**,** ev\_active **(**w**)** **+** 1**,** EMPTY2**);**

ANHE\_w **(**timers **[**ev\_active **(**w**)])** **=** **(**WT**)**w**;**

ANHE\_at\_cache **(**timers **[**ev\_active **(**w**)]);**

upheap **(**timers**,** ev\_active **(**w**));**

**}**

代码比较简单，首先设置监视器的at成员，表明在at时间点，超时事件会触发，注意at是根据mn\_now设置的，也就是相对于系统启动时间而言的（或者是日历时间）。之后，就是将该监视器加入到堆timer中，首先将该监视器加到堆中的最后一个元素，然后调用upheap调整堆。注意监视器的active成员，表明该监视器在堆数组中的下标。

3：停止超时监视器ev\_timer\_stop

void ev\_timer\_stop **(**EV\_P\_ ev\_timer **\***w**)** EV\_THROW

**{**

clear\_pending **(**EV\_A\_ **(**W**)**w**);**

**if** **(**expect\_false **(!**ev\_is\_active **(**w**)))**

**return;**

**int** active **=** ev\_active **(**w**);**

**--**timercnt**;**

**if** **(**expect\_true **(**active **<** timercnt **+** HEAP0**))**

**{**

timers **[**active**]** **=** timers **[**timercnt **+** HEAP0**];**

adjustheap **(**timers**,** timercnt**,** active**);**

**}**

ev\_at **(**w**)** **-=** mn\_now**;**

ev\_stop **(**EV\_A\_ **(**W**)**w**);**

**}**

首先调用clear\_pending，如果该监视器已经处于pending状态，将其从pendings中删除。然后根据监视器中的active成员，得到其在timers堆上的索引，将该监视器从堆timers上删除，重新调整堆结构。然后调用ev\_stop停止该监视器。

4：timers\_reschedule更新定时器的时间

在ev\_run中，每次loop中，在调用backend\_poll前后都会调用time\_update更新当前时间，如果发现时间被人调整，则需要更新定时器，更新相对定时器时，调用timers\_reschedule(loop, ev\_rt\_now - mn\_now)，其中，ev\_rt\_now是最新的当前日历时间，mn\_now是之前记录的日历时间，他们之间的差值就表示时间调整了多少，timers\_reschedule代码如下：

static void timers\_reschedule **(**struct ev\_loop **\***loop**,** ev\_tstamp adjust**)**

**{**

int i**;**

**for** **(**i **=** 0**;** i **<** timercnt**;** **++**i**)**

**{**

ANHE **\***he **=** timers **+** i **+** HEAP0**;**

ANHE\_w **(\***he**)->**at **+=** adjust**;**

ANHE\_at\_cache **(\***he**);**

**}**

**}**

代码较简单，就是更新堆timers中的每个元素的at值。

4：timers\_reify将激活的超时事件排队

每次调用backend\_poll之前，都会根据ANHE\_at (timers [HEAP0]) - mn\_now的值，校准backend\_poll的阻塞时间waittime，这样就能尽可能的保证定时器能够按时触发。

调用backend\_poll之后，就会调用timers\_reify查看timers中哪些定时器触发了，代码如下：

void timers\_reify**(**struct ev\_loop **\***loop**)**

**{**

**if** **(**timercnt **&&** ANHE\_at **(**timers **[**HEAP0**])** **<** mn\_now**)**

**{**

do

**{**

ev\_timer **\***w **=** **(**ev\_timer **\*)**ANHE\_w **(**timers **[**HEAP0**]);**

**/\*** first reschedule **or** stop timer **\*/**

**if** **(**w**->**repeat**)**

**{**

ev\_at **(**w**)** **+=** w**->**repeat**;**

**if** **(**ev\_at **(**w**)** **<** mn\_now**)**

ev\_at **(**w**)** **=** mn\_now**;**

ANHE\_at\_cache **(**timers **[**HEAP0**]);**

downheap **(**timers**,** timercnt**,** HEAP0**);**

**}**

**else**

ev\_timer\_stop **(**EV\_A\_ w**);** **/\*** nonrepeating**:** stop timer **\*/**

feed\_reverse **(**EV\_A\_ **(**W**)**w**);**

**}**

**while** **(**timercnt **&&** ANHE\_at **(**timers **[**HEAP0**])** **<** mn\_now**);**

feed\_reverse\_done **(**loop**,** EV\_TIMER**);**

**}**

**}**

void feed\_reverse **(**struct ev\_loop **\***loop**,** W w**)**

**{**

array\_needsize **(**W**,** rfeeds**,** rfeedmax**,** rfeedcnt **+** 1**,** EMPTY2**);**

rfeeds **[**rfeedcnt**++]** **=** w**;**

**}**

void feed\_reverse\_done **(**struct ev\_loop **\***loop**,** int revents**)**

**{**

**do**

ev\_feed\_event **(**EV\_A\_ rfeeds **[--**rfeedcnt**],** revents**);**

**while** **(**rfeedcnt**);**

**}**

如果堆顶元素的超时时间点at小于mn\_now（是小于，而不是等于，原因见：

http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#The\_special\_problem\_of\_being\_too\_ear），说明堆顶元素的超时时间到时，进入循环，取出堆顶元素的监视器，如果w->repeat大于0，则设置该超时监视器下一次触发的时间点，然后调用downheap调整堆结构。否则，直接调用ev\_timer\_stop停止该监视器。最后，调用feed\_reverse将该监视器放入rfeeds数组中，该数组对当前已经触发的超时监视器缓存。然后，继续查看新的堆顶元素是否已经超时，超时的话接着上面的步骤进行处理。

将所有本次触发的超时事件都处理完之后，调用feed\_reverse\_done，将缓存数组rfeeds中每个元素通过ev\_feed\_event添加到pendings数组中。

**三：例子**

void timer\_action**(**struct ev\_loop **\***main\_loop**,**ev\_timer **\***timer\_w**,int** e**)**

**{**

time\_t now**;**

now **=** time**(**NULL**);**

printf**(**"in tiemr cb%d , cur time is %s\n"**,** **(int)(**timer\_w**->**data**),** ctime**(&**now**));**

**}**

**int** main**(int** argc **,char** **\***argv**[])**

**{**

struct ev\_loop **\***main\_loop **=** ev\_default\_loop**(**0**);**

timer\_w1**.**data **=** **(**void **\*)**1**;**

ev\_init**(&**timer\_w1**,**timer\_action**);**

ev\_timer\_set**(&**timer\_w1**,**10**,**5**);**

ev\_timer\_start**(**main\_loop**,&**timer\_w1**);**

timer\_w2**.**data **=** **(**void **\*)**2**;**

ev\_init**(&**timer\_w2**,**timer\_action**);**

ev\_timer\_set**(&**timer\_w2**,**5**,**10**);**

ev\_timer\_start**(**main\_loop**,&**timer\_w2**);**

time\_t now**;**

now **=** time**(**NULL**);**

printf**(**"begin time time is %s\n"**,** ctime**(&**now**));**

ev\_run**(**main\_loop**,**0**);**

**return;**

**}**

结果打印：

begin time time is Wed Oct 21 21:55:31 2015

in tiemr cb2 , cur time is Wed Oct 21 21:55:36 2015

in tiemr cb1 , cur time is Wed Oct 21 21:55:41 2015

in tiemr cb1 , cur time is Wed Oct 21 21:55:46 2015

in tiemr cb2 , cur time is Wed Oct 21 21:55:46 2015

in tiemr cb1 , cur time is Wed Oct 21 21:55:51 2015

in tiemr cb1 , cur time is Wed Oct 21 21:55:56 2015

in tiemr cb2 , cur time is Wed Oct 21 21:55:56 2015

in tiemr cb1 , cur time is Wed Oct 21 21:56:01 2015

...

**ev\_periodic**

Libev中的超时监视器ev\_periodic，是绝对时间定时器，不同于ev\_timer，它是基于日历时间的。比如如果指定一个ev\_periodic在10秒之后触发（ev\_now() + 10），然后将系统时间调整为去年的一月一号，则该定时器会在一年后才触发超时事件。（ev\_timer依然会在10秒之后触发）

**一：数据结构**

超时监视器ev\_ periodic结构：

**typedef** struct ev\_periodic

**{**

int active**;**

int pending**;**

int priority**;**

void **\***data**;**

void **(\***cb**)(**struct ev\_loop **\***loop**,** struct ev\_periodic **\***w**,** int revents**);**

ev\_tstamp at**;**

ev\_tstamp offset**;** */\* rw \*/*

ev\_tstamp interval**;** */\* rw \*/*

ev\_tstamp **(\***reschedule\_cb**)(**struct ev\_periodic **\***w**,** ev\_tstamp now**)** EV\_THROW**;** */\* rw \*/*

**}** ev\_periodic**;**

可见其中的前六个成员与ev\_timer和ev\_watcher\_time是一样的。与ev\_timer类似，ev\_periodic中的active也标明该监视器在堆数组periodics中的下标；at表明超时事件触发的时间点，共有三种设置方法，而且offset、interval和reschedule\_cb都是用来设置触发时间的，这个会在下面说明。

**二：监视器函数**

1：设置超时监视器

#define ev\_periodic\_set(ev,ofs\_,ival\_,rcb\_) do {\

(ev)->offset = (ofs\_); \

(ev)->interval = (ival\_); \

(ev)->reschedule\_cb = (rcb\_); \

}while (0)

#define ev\_periodic\_init(ev,cb,ofs,ival,rcb) do {\

ev\_init ((ev), (cb)); \

ev\_periodic\_set ((ev),(ofs),(ival),(rcb)); \

} while (0)

2：启动监视器ev\_periodic\_start

void ev\_periodic\_start **(**struct ev\_loop **\***loop**,** ev\_periodic **\***w**)**

**{**

**if** **(**expect\_false **(**ev\_is\_active **(**w**)))**

**return;**

**if** **(**w**->**reschedule\_cb**)**

ev\_at **(**w**)** **=** w**->**reschedule\_cb **(**w**,** ev\_rt\_now**);**

**else** **if** **(**w**->**interval**)**

**{**

assert **((**"libev: ev\_periodic\_start called with negative interval value"**,** w**->**interval **>=** 0.**));**

periodic\_recalc **(**EV\_A\_ w**);**

**}**

**else**

ev\_at **(**w**)** **=** w**->**offset**;**

**++**periodiccnt**;**

ev\_start **(**EV\_A\_ **(**W**)**w**,** periodiccnt **+** HEAP0 **-** 1**);**

array\_needsize **(**ANHE**,** periodics**,** periodicmax**,** ev\_active **(**w**)** **+** 1**,** EMPTY2**);**

ANHE\_w **(**periodics **[**ev\_active **(**w**)])** **=** **(**WT**)**w**;**

ANHE\_at\_cache **(**periodics **[**ev\_active **(**w**)]);**

upheap **(**periodics**,** ev\_active **(**w**));**

**}**

共有三种设置超时时间at的方法：

a：如果reschedule\_cb不为空，则忽略interval和offset，而使用reschedule\_cb函数设置超时时间at，该函数以ev\_rt\_now为参数，设置下次超时事件触发的时间，每次重新设置at的时候（periodics\_reschedule，periodics\_reify），都会调用该函数。该函数的一个例子如下：

static ev\_tstamp my\_rescheduler **(**ev\_periodic **\***w**,** ev\_tstamp now**)**

**{**

**return** now **+** 60.**;**

**}**

这就是将at设置为1分钟之后的时间点。

b：reschedule\_cb为空，interval>0，这种情况下，调用periodic\_recalc设置at。该函数的作用就是将at置为下一个的offset + N\*interval时间点，其中的offset一般处于[0, interval]范围内。比如置offset为0，interval为3600，意味着当系统时间是完整的1小时的时候，也就是系统时间可以被3600整除的时候，比如8:00，9:00等，就会触发超时事件。periodic\_recalc的代码见下面。

c：如果reschedule\_cb为空，interval为0，则直接将at置为offset。这是一种绝对值，这种情况下，该监视器不会重复触发，触发一次之后就会停止监视器；而且该监视器也会无视时间调整，比如置at为20110101000000，则只要系统日历时间超过了改时间，就会触发超时事件。

设置好at之后，就是将该监视器加入到堆periodics中，这与ev\_timer的代码是一样的，不再赘述。

3：periodic\_recalc重新计算下一个触发时间点

void periodic\_recalc **(**struct ev\_loop **\***loop**,** ev\_periodic **\***w**)**

**{**

ev\_tstamp interval **=** w**->**interval **>** MIN\_INTERVAL **?** w**->**interval **:** MIN\_INTERVAL**;**

ev\_tstamp at **=** w**->**offset **+** interval **\*** ev\_floor **((**ev\_rt\_now **-** w**->**offset**)** **/** interval**);**

**while** **(**at **<=** ev\_rt\_now**)**

**{**

ev\_tstamp nat **=** at **+** w**->**interval**;**

**if** **(**expect\_false **(**nat **==** at**))**

**{**

at **=** ev\_rt\_now**;**

**break;**

**}**

at **=** nat**;**

**}**

ev\_at **(**w**)** **=** at**;**

**}**

该函数的作用就是将at置为下一个的offset + N\*interval时间点。ev\_floor(x)返回小于x，且最接近x的整数。

举个例子可能会容易明白该代码：interval为10分钟（600），offset为2分钟（120），表示将at置为下一个分钟数为2的时间点。

假设当前为8:01:23，则最终会使得at为8:02:00。计算过程是 ：interval \* ev\_floor ((ev\_rt\_now - w->offset) / interval)就表示7:50:00，然后再加上offset就是7:52:00，进入循环，最终调整得at=8:02:00。

假设当前为8:03:56，则最终会使得at为8:12:00。计算过程是：interval \* ev\_floor ((ev\_rt\_now - w->offset) / interval)就表示8:00:00，然后再加上offset就是8:02:00，进入循环，最终调整得at=8:12:00。

4：停止超时监视器ev\_periodic\_stop

void ev\_periodic\_stop **(**struct ev\_loop **\***loop**,** ev\_periodic **\***w**)**

**{**

clear\_pending **(**EV\_A\_ **(**W**)**w**);**

**if** **(**expect\_false **(!**ev\_is\_active **(**w**)))**

**return;**

int active **=** ev\_active **(**w**);**

**--**periodiccnt**;**

**if** **(**expect\_true **(**active **<** periodiccnt **+** HEAP0**))**

**{**

periodics **[**active**]** **=** periodics **[**periodiccnt **+** HEAP0**];**

adjustheap **(**periodics**,** periodiccnt**,** active**);**

**}**

ev\_stop **(**EV\_A\_ **(**W**)**w**);**

**}**

代码与ev\_timer\_stop几乎完全一致，不再赘述。

5：重新调整超时时间periodics\_reschedule

static void periodics\_reschedule **(**struct ev\_loop **\***loop**)**

**{**

int i**;**

**for** **(**i **=** HEAP0**;** i **<** periodiccnt **+** HEAP0**;** **++**i**)**

**{**

ev\_periodic **\***w **=** **(**ev\_periodic **\*)**ANHE\_w **(**periodics **[**i**]);**

**if** **(**w**->**reschedule\_cb**)**

ev\_at **(**w**)** **=** w**->**reschedule\_cb **(**w**,** ev\_rt\_now**);**

**else** **if** **(**w**->**interval**)**

periodic\_recalc **(**EV\_A\_ w**);**

ANHE\_at\_cache **(**periodics **[**i**]);**

**}**

reheap **(**periodics**,** periodiccnt**);**

**}**

在time\_update中，如果发现日历时间被调整了，则会调用periodics\_reschedule函数，调整ev\_periodic的超时时间点at。调整的方法跟ev\_periodic\_start中的一样，要么使用reschedule\_cb函数调整，要么就是调用periodic\_recalc重新计算at。最后，将periodics堆中所有元素都调整完毕后，调用reheap使periodics恢复堆结构。

6：将激活的超时事件排队periodics\_reify

void periodics\_reify **(**struct ev\_loop **\***loop**)**

**{**

**while** **(**periodiccnt **&&** ANHE\_at **(**periodics **[**HEAP0**])** **<** ev\_rt\_now**)**

**{**

**do{**

ev\_periodic **\***w **=** **(**ev\_periodic **\*)**ANHE\_w **(**periodics **[**HEAP0**]);**

**if** **(**w**->**reschedule\_cb**)**

**{**

ev\_at **(**w**)** **=** w**->**reschedule\_cb **(**w**,** ev\_rt\_now**);**

assert **((**"libev: ev\_periodic reschedule callback returned time in the past"**,** ev\_at **(**w**)** **>=** ev\_rt\_now**));**

ANHE\_at\_cache **(**periodics **[**HEAP0**]);**

downheap **(**periodics**,** periodiccnt**,** HEAP0**);**

**}**

**else** **if** **(**w**->**interval**)**

**{**

periodic\_recalc **(**EV\_A\_ w**);**

ANHE\_at\_cache **(**periodics **[**HEAP0**]);**

downheap **(**periodics**,** periodiccnt**,** HEAP0**);**

**}**

**else**

ev\_periodic\_stop **(**EV\_A\_ w**);**

feed\_reverse **(**EV\_A\_ **(**W**)**w**);**

**}**

**while** **(**periodiccnt **&&** ANHE\_at **(**periodics **[**HEAP0**])** **<** ev\_rt\_now**);**

feed\_reverse\_done **(**EV\_A\_ EV\_PERIODIC**);**

**}**

**}**

主要流程跟timers\_reify一样，只不过在重新计算下次触发时间点at的时候，计算方法跟ev\_periodic\_start中的一样。

**三：例子**

ev\_periodic pw**;**

void periodic\_action**(**struct ev\_loop **\***main\_loop**,**ev\_periodic **\***timer\_w**,**int e**)**

**{**

time\_t now**;**

now **=** time**(NULL);**

printf**(**"cur time is %s\n"**,** ctime**(&**now**));**

**}**

static ev\_tstamp my\_rescheduler **(**ev\_periodic **\***w**,** ev\_tstamp now**)**

**{**

**return** now**+**120**;**

**}**

int main**()**

**{**

time\_t now**;**

now **=** time**(NULL);**

struct ev\_loop **\***main\_loop **=** ev\_default\_loop**(**0**);**

ev\_periodic\_init**(&**pw**,** periodic\_action**,** 0**,** 0**,** my\_rescheduler**);** //1

//ev\_periodic\_init**(&**pw**,** periodic\_action**,** 120, 600**,** NULL**); //2**

//ev\_periodic\_init**(&**pw**,** periodic\_action**,** now+20**,** 0**,** NULL**); //3**

ev\_periodic\_start**(**main\_loop**,&**pw**);**

printf**(**"begin time time is %s\n"**,** ctime**(&**now**));**

ev\_run**(**main\_loop**,**0**);**

**return;**

**}**

采用第一种初始化方法：

ev\_periodic\_init**(&**pw**,** periodic\_action**,** 0**,** 0**,** my\_rescheduler**);**

结果是：

begin time time is Thu Oct 29 21**:**33**:**05 2015

cur time is Thu Oct 29 21**:**35**:**05 2015

cur time is Thu Oct 29 21**:**37**:**05 2015

cur time is Thu Oct 29 21**:**39**:**05 2015

cur time is Thu Oct 29 21**:**41**:**05 2015

...

采用第二种初始化方法：

ev\_periodic\_init**(&**pw**,** periodic\_action**,** 120, 600**,** NULL**);**

结果是：

begin time time is Thu Oct 29 21**:**38**:**29 2015

cur time is Thu Oct 29 21**:**42**:**00 2015

cur time is Thu Oct 29 21**:**52**:**00 2015

cur time is Thu Oct 29 22**:**02**:**00 2015

cur time is Thu Oct 29 22**:**12**:**00 2015

cur time is Thu Oct 29 22**:**22**:**00 2015

...

采用第三种初始化方法：

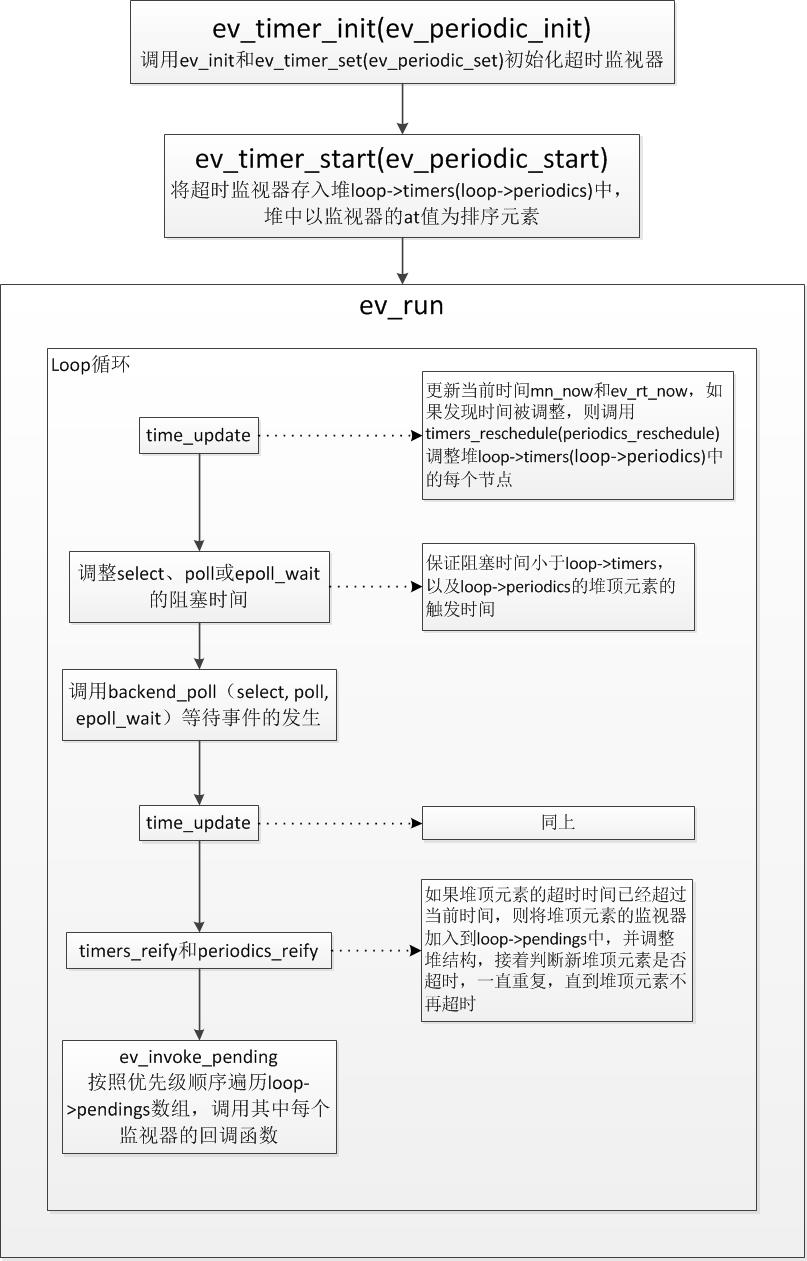
ev\_periodic\_init**(&**pw**,** periodic\_action**,** now+20**,** 0**,** NULL**);**

结果是：

begin time time is Thu Oct 29 21**:**39**:**03 2015

cur time is Thu Oct 29 21**:**39**:**23 2015

**流程图**

****