**一：代码流程**

在Libev中，启动一个IO监视器，等待该监视器上的事件触发，然后调用该监视器的回调函数。整个的流程是这样的：

首先调用ev\_default\_loop初始化struct ev\_loop结构；

然后调用ev\_io\_init初始化监视器中的属性，该宏主要就是调用ev\_init和ev\_io\_set；

然后调用ev\_io\_start启动该监视器，该函数主要是将监视器添加到loop->anfds结构中，将监视的描述符添加到((loop)->fdchanges)中；

调用ev\_run开始等待事件的触发，该函数中：

首先会调用fd\_reify函数，该函数根据((loop)->fdchanges)中记录的描述符，将该描述符上的事件添加到backend所使用的数据结构中，比如select中的fd\_set中；

然后调用time\_update更新当前的时间，如果日历时间被人为调整的话，则相应的调整超时事件和周期事件；

调用backend\_poll开始等待事件的发生，如果事件在规定时间内触发的话，则会调用fd\_event将触发的监视器记录到loop->pendings中；

backend的监听函数（select，poll，epoll\_wait）返回之后，首先再次调用time\_update更新当前的时间，然后调用ev\_invoke\_pending，依次处理loop->pendings中的监视器，调用该监视器的回调函数。

以上就是Libev中IO监视器的工作流程，下面详细分析各个函数：

1：ev\_default\_loop函数

#if EV\_MULTIPLICITY

struct ev\_loop **\***

#else

int

#endif

ev\_default\_loop **(**unsigned int flags**)**

**{**

**if** **(!**ev\_default\_loop\_ptr**)**

**{**

#if EV\_MULTIPLICITY

struct ev\_loop **\***loop **=** ev\_default\_loop\_ptr **=** **&**default\_loop\_struct**;**

#else

ev\_default\_loop\_ptr **=** 1**;**

#endif

loop\_init **(**loop**,** flags**);**

**if** **(**ev\_backend **(**loop**))**

**{**

#if EV\_CHILD\_ENABLE

ev\_signal\_init **(&**childev**,** childcb**,** SIGCHLD**);**

ev\_set\_priority **(&**childev**,** EV\_MAXPRI**);**

ev\_signal\_start **(**EV\_A\_ **&**childev**);**

ev\_unref **(**loop**);** */\* child watcher should not keep loop alive \*/*

#endif

**}**

**else**

ev\_default\_loop\_ptr **=** 0**;**

**}**

**return** ev\_default\_loop\_ptr**;**

**}**

EV\_MULTIPLICITY宏用来决定是否支持多个loop。系统提供了默认的loop结构default\_loop\_struct，和指向其的指针ev\_default\_loop\_ptr。

如果支持多个loop，则default\_loop\_struct就是一个静态的struct ev\_loop类型的结构体，其中包含了各种成员，比如ev\_tstamp ev\_rt\_now; int pendingpri;等等。

ev\_default\_loop\_ptr就是指向struct ev\_loop 类型的指针。

如果不支持多个loop，则上述的struct ev\_loop结构就不复存在，其成员都是以静态变量的形式进行定义，而ev\_default\_loop\_ptr也只是一个int变量，用来表明”loop”是否已经初始化成功。

下面的描述，均以支持多个loop为准。

在ev\_default\_loop中，首先是调用loop\_init初始化loop中的各种成员：

static void loop\_init **(**struct ev\_loop **\***loop**,** unsigned int flags**)**

**{**

**if** **(!**backend**)**

**{**

origflags **=** flags**;**

#if EV\_USE\_REALTIME

**if** **(!**have\_realtime**)**

**{**

struct timespec ts**;**

**if** **(!**clock\_gettime **(**CLOCK\_REALTIME**,** **&**ts**))**

have\_realtime **=** 1**;**

**}**

#endif

#if EV\_USE\_MONOTONIC

**if** **(!**have\_monotonic**)**

**{**

struct timespec ts**;**

**if** **(!**clock\_gettime **(**CLOCK\_MONOTONIC**,** **&**ts**))**

have\_monotonic **=** 1**;**

**}**

#endif

*/\* pid check not overridable via env \*/*

#ifndef \_WIN32

**if** **(**flags **&** EVFLAG\_FORKCHECK**)**

curpid **=** getpid **();**

#endif

**if** **(!(**flags **&** EVFLAG\_NOENV**)** **&&** **!**enable\_secure **()** **&&** getenv **(**"LIBEV\_FLAGS"**))**

flags **=** atoi **(**getenv **(**"LIBEV\_FLAGS"**));**

ev\_rt\_now **=** ev\_time **();**

mn\_now **=** get\_clock **();**

now\_floor **=** mn\_now**;**

rtmn\_diff **=** ev\_rt\_now **-** mn\_now**;**

#if EV\_FEATURE\_API

invoke\_cb **=** ev\_invoke\_pending**;**

#endif

io\_blocktime **=** 0.**;**

timeout\_blocktime **=** 0.**;**

backend **=** 0**;**

backend\_fd **=** **-**1**;**

sig\_pending **=** 0**;**

#if EV\_ASYNC\_ENABLE

async\_pending **=** 0**;**

#endif

pipe\_write\_skipped **=** 0**;**

pipe\_write\_wanted **=** 0**;**

evpipe **[**0**]** **=** **-**1**;**

evpipe **[**1**]** **=** **-**1**;**

#if EV\_USE\_INOTIFY

fs\_fd **=** flags **&** EVFLAG\_NOINOTIFY **?** **-**1 **:** **-**2**;**

#endif

#if EV\_USE\_SIGNALFD

sigfd **=** flags **&** EVFLAG\_SIGNALFD **?** **-**2 **:** **-**1**;**

#endif

**if** **(!(**flags **&** EVBACKEND\_MASK**))**

flags **|=** ev\_recommended\_backends **();**

#if EV\_USE\_IOCP

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_IOCP **))** backend **=** iocp\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_PORT

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_PORT **))** backend **=** port\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_KQUEUE

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_KQUEUE**))** backend **=** kqueue\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_EPOLL

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_EPOLL **))** backend **=** epoll\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_POLL

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_POLL **))** backend **=** poll\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_SELECT

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_SELECT**))** backend **=** select\_init **(**EV\_A\_ flags**);**

#endif

ev\_prepare\_init **(&**pending\_w**,** pendingcb**);**

#if EV\_SIGNAL\_ENABLE || EV\_ASYNC\_ENABLE

ev\_init **(&**pipe\_w**,** pipecb**);**

ev\_set\_priority **(&**pipe\_w**,** EV\_MAXPRI**);**

#endif

**}**

**}**

调用ev\_time初始化ev\_rt\_now，得到当前的日历时间，也就是自19700101000000以来的秒数，该值通过CLOCK\_REALTIME或者gettimeofday得到；

调用get\_clock初始化mn\_now，该变量要么是CLOCK\_MONOTONIC（系统启动时间），要么就是ev\_time的值（日历时间）；

然后就是：

now\_floor **=** mn\_now**;**

rtmn\_diff **=** ev\_rt\_now **-** mn\_now**;**

#if EV\_FEATURE\_API

invoke\_cb **=** ev\_invoke\_pending**;**

#endif

io\_blocktime **=** 0.**;**

timeout\_blocktime **=** 0.**;**

backend **=** 0**;**

backend\_fd **=** **-**1**;**

sig\_pending **=** 0**;**

#if EV\_ASYNC\_ENABLE

async\_pending **=** 0**;**

#endif

pipe\_write\_skipped **=** 0**;**

pipe\_write\_wanted **=** 0**;**

evpipe **[**0**]** **=** **-**1**;**

evpipe **[**1**]** **=** **-**1**;**

#if EV\_USE\_INOTIFY

fs\_fd **=** flags **&** EVFLAG\_NOINOTIFY **?** **-**1 **:** **-**2**;**

#endif

#if EV\_USE\_SIGNALFD

sigfd **=** flags **&** EVFLAG\_SIGNALFD **?** **-**2 **:** **-**1**;**

#endif

之后调用ev\_recommended\_backends得到当前系统支持的backend类型，比如select, poll, epoll等。

接下来就是根据系统支持的backend，按照一定的优先顺序，去初始化backend：

#if EV\_USE\_IOCP

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_IOCP **))** backend **=** iocp\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_PORT

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_PORT **))** backend **=** port\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_KQUEUE

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_KQUEUE**))** backend **=** kqueue\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_EPOLL

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_EPOLL **))** backend **=** epoll\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_POLL

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_POLL **))** backend **=** poll\_init **(**EV\_A\_ flags**);**

#endif

#if EV\_USE\_SELECT

**if** **(!**backend **&&** **(**flags **&** EVBACKEND\_SELECT**))** backend **=** select\_init **(**EV\_A\_ flags**);**

#endif

接下来，初始化loop中的ev\_prepare监视器pending\_w，以及ev\_io监视器pipe\_w

loop\_init返回后，backend已经初始化完成，接着，初始化并启动信号监视器ev\_signal childev。暂不深入。

至此，初始化默认loop的工作就完成了。

2：ev\_init

该函数以宏的形式存在，主要用来设置监视器的公共成员active、pending、priority、cb等。代码如下：

#define ev\_init(ev,cb\_) do { \

((ev\_watcher \*)(void \*)(ev))->active = \

((ev\_watcher \*)(void \*)(ev))->pending = 0; \

ev\_set\_priority ((ev), 0); \

ev\_set\_cb ((ev), cb\_); \

} while (0)

3：ev\_io\_set

该宏主要是设置IO监视器ev\_io的特有成员：要监听的描述符fd和其上的事件event。其中设置event会包含事件掩码EV\_\_IOFDSET，其代码如下：

#define ev\_io\_set(ev,fd\_,events\_) do { (ev)->fd = (fd\_); (ev)->events = (events\_) | EV\_\_IOFDSET; } while (0)

4：ev\_io\_start

void ev\_io\_start **(**struct ev\_loop **\***loop**,** ev\_io **\***w**)** EV\_THROW

**{**

int fd **=** w**->**fd**;**

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

**return;**

assert **((**"libev: ev\_io\_start called with negative fd"**,** fd **>=** 0**));**

assert **((**"libev: ev\_io\_start called with illegal event mask"**,** **!(**w**->**events **&** **~(**EV\_\_IOFDSET **|** EV\_READ **|** EV\_WRITE**))));**

ev\_start **(**loop**,** **(**W**)**w**,** 1**);**

array\_needsize **(**ANFD**,** anfds**,** anfdmax**,** fd **+** 1**,** array\_init\_zero**);**

wlist\_add **(&**anfds**[**fd**].**head**,** **(**WL**)**w**);**

*/\* common bug, apparently \*/*

assert **((**"libev: ev\_io\_start called with corrupted watcher"**,** **((**WL**)**w**)->**next **!=** **(**WL**)**w**));**

fd\_change **(**loop**,** fd**,** w**->**events **&** EV\_\_IOFDSET **|** EV\_ANFD\_REIFY**);**

w**->**events **&=** **~**EV\_\_IOFDSET**;**

**}**

首先对监视器ev做检查：

ev->active == 0： 监视器现在的状态应是未启动的；

fd>=0;

(!(w->events & ~(EV\_\_IOFDSET | EV\_READ | EV\_WRITE)))： IO监视器只能监控EV\_\_IOFDSET，EV\_READ，EV\_WRITE中的事件，其他事件一律不能关心。

调用ev\_start矫正ev的优先级；置ev->active=1表明状态为启动状态；++(loop->activecnt)

根据情况调整((loop)->anfds)数组的大小，然后将监视器ev加入到(loop->anfds)[fd].head的链表中。

loop->anfds是ANFD结构类型的数组，ANFD结构体定义如下：

**typedef** struct

**{**

WL head**;**

unsigned char events**;** */\* the events watched for \*/*

unsigned char reify**;***/\* flag set when this ANFD needs reification (EV\_ANFD\_REIFY, EV\_\_IOFDSET)\*/*

unsigned char emask**;** */\* the epoll backend stores the actual kernel mask in here \*/*

unsigned char unused**;**

#if EV\_USE\_EPOLL

unsigned int egen**;** */\* generation counter to counter epoll bugs \*/*

#endif

#if EV\_SELECT\_IS\_WINSOCKET || EV\_USE\_IOCP

SOCKET handle**;**

#endif

#if EV\_USE\_IOCP

OVERLAPPED or**,** ow**;**

#endif

**}** ANFD**;**

每一个描述符对应着一个ANFD结构，描述符的值就是((loop)->anfds)的下标。每个描述符上可以有若干监视器，同一个描述符上的监视器以链表的形式组织，这里ANFD结构中的head就是链表头指针。

((loop)->anfds)数组是动态变化的，初始为空。(loop)->anfdmax就是该数组的当前大小。

调用**fd\_change(loop, fd, w->events & EV\_\_IOFDSET | EV\_ANFD\_REIFY)，**

void fd\_change **(**struct ev\_loop **\***loop**,** int fd**,** int flags**)**

**{**

unsigned char reify **=** anfds **[**fd**].**reify**;**

anfds **[**fd**].**reify **|=** flags**;**

**if** **(**expect\_true **(!**reify**))**

**{**

**++**fdchangecnt**;**

array\_needsize **(**int**,** fdchanges**,** fdchangemax**,** fdchangecnt**,** EMPTY2**);**

fdchanges **[**fdchangecnt **-** 1**]** **=** fd**;**

**}**

**}**

查看(loop->anfds) [fd].reify的原值，如果原值为0，表明该描述符是第一次加入监控，将其记录到((loop)->fdchanges)数组中，该数组记录了当前监控中的描述符，((loop)->fdchangemax)记录该数组当前实际大小，((loop)->fdchangecnt)记录该数组当前使用大小。

将**w->events & EV\_\_IOFDSET | EV\_ANFD\_REIFY**添加到(loop->anfds) [fd].reify中。

最后，将w->events中的EV\_\_IOFDSET掩码消除：

w->events &= ~EV\_\_IOFDSET;

5：fd\_reify

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

**{**

int i**;**

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

**{**

int fd **=** fdchanges **[**i**];**

ANFD **\***anfd **=** anfds **+** fd**;**

ev\_io **\***w**;**

unsigned char o\_events **=** anfd**->**events**;**

unsigned char o\_reify **=** anfd**->**reify**;**

anfd**->**reify **=** 0**;**

*/\*if (expect\_true (o\_reify & EV\_ANFD\_REIFY)) probably a deoptimisation \*/*

**{**

anfd**->**events **=** 0**;**

**for** **(**w **=** **(**ev\_io **\*)**anfd**->**head**;** w**;** w **=** **(**ev\_io **\*)((**WL**)**w**)->**next**)**

anfd**->**events **|=** **(**unsigned char**)**w**->**events**;**

**if** **(**o\_events **!=** anfd**->**events**)**

o\_reify **=** EV\_\_IOFDSET**;** */\* actually |= \*/*

**}**

**if** **(**o\_reify **&** EV\_\_IOFDSET**)**

backend\_modify **(**loop**,** fd**,** o\_events**,** anfd**->**events**);**

**}**

fdchangecnt **=** 0**;**

**}**

轮训数组((loop)->fdchanges)，从0到((loop)->fdchangecnt -1)之间的所有元素，每个元素代表了一个描述符，根据取得的描述符值fd，找到相应的ANFD结构anfd。

记录原anfd->events和anfd->reify的值，然后：

anfd**->**reify **=** 0**;**

anfd**->**events **=** 0**;**

**for** **(**w **=** **(**ev\_io **\*)**anfd**->**head**;** w**;** w **=** **(**ev\_io **\*)((**WL**)**w**)->**next**)**

anfd**->**events **|=** **(**unsigned char**)**w**->**events**;**

然后调用backend\_modify函数开始对fd及其上的所有事件开始监控。以backend为select例，就是根据anfd->events中的事件，将fd添加到相应的fd\_set中去。

最后，置((loop)->fdchangecnt)为0。

6：time\_update函数

/\* fetch new monotonic and realtime times from the kernel \*/

/\* also detect if there was a timejump, and act accordingly \*/

void time\_update **(**struct ev\_loop **\***loop**,** ev\_tstamp max\_block**)**

**{**

#if EV\_USE\_MONOTONIC

**if** **(**expect\_true **(**have\_monotonic**))**

**{**

int i**;**

ev\_tstamp odiff **=** rtmn\_diff**;**

mn\_now **=** get\_clock **();**

*/\* only fetch the realtime clock every 0.5\*MIN\_TIMEJUMP seconds \*/*

*/\* interpolate in the meantime \*/*

**if** **(**expect\_true **(**mn\_now **-** now\_floor **<** MIN\_TIMEJUMP **\*** .5**))**

**{**

ev\_rt\_now **=** rtmn\_diff **+** mn\_now**;**

**return;**

**}**

now\_floor **=** mn\_now**;**

ev\_rt\_now **=** ev\_time **();**

*/\* loop a few times, before making important decisions.*

*\* on the choice of "4": one iteration isn't enough,*

*\* in case we get preempted during the calls to*

*\* ev\_time and get\_clock. a second call is almost guaranteed*

*\* to succeed in that case, though. and looping a few more times*

*\* doesn't hurt either as we only do this on time-jumps or*

*\* in the unlikely event of having been preempted here.*

*\*/*

**for** **(**i **=** 4**;** **--**i**;** **)**

**{**

ev\_tstamp diff**;**

rtmn\_diff **=** ev\_rt\_now **-** mn\_now**;**

diff **=** odiff **-** rtmn\_diff**;**

**if** **(**expect\_true **((**diff **<** 0. **?** **-**diff **:** diff**)** **<** MIN\_TIMEJUMP**))**

**return;** */\* all is well \*/*

ev\_rt\_now **=** ev\_time **();**

mn\_now **=** get\_clock **();**

now\_floor **=** mn\_now**;**

**}**

*/\* no timer adjustment, as the monotonic clock doesn't jump \*/*

*/\* timers\_reschedule (EV\_A\_ rtmn\_diff - odiff) \*/*

# if EV\_PERIODIC\_ENABLE

periodics\_reschedule **(**EV\_A**);**

# endif

**}**

**else**

#endif

**{**

ev\_rt\_now **=** ev\_time **();**

**if** **(**expect\_false **(**mn\_now **>** ev\_rt\_now **||** ev\_rt\_now **>** mn\_now **+** max\_block **+** MIN\_TIMEJUMP**))**

**{**

*/\* adjust timers. this is easy, as the offset is the same for all of them \*/*

timers\_reschedule **(**EV\_A\_ ev\_rt\_now **-** mn\_now**);**

#if EV\_PERIODIC\_ENABLE

periodics\_reschedule **(**EV\_A**);**

#endif

**}**

mn\_now **=** ev\_rt\_now**;**

**}**

**}**

该函数重新获得mn\_now、ev\_rt\_now等的值，并且如果发现时间被人为调整的话，则在代码中也作出相应的调整。ev\_rt\_now表示日历时间，mn\_now要么表示系统启动时间，要么表示日历时间。

如果宏定义EV\_USE\_MONOTONIC为1，并且have\_monotonic为1（sys\_clock\_gettime支持CLOCK\_MONOTONIC）的话，mn\_now就表示系统启动时间，它不会被人为的调整。

这种情况下，更新系统启动时间mn\_now的值，如果该值与旧的mn\_now的值之差不超过0.5s的话，表示刚刚更新过时间（更新时间不超过0.5s），则更新ev\_rt\_now之后，直接退出。

更新ev\_rt\_now的值，然后根据ev\_rt\_now - mn\_now之差的变化，判断时间是否被人调整。如果ev\_rt\_now - mn\_now之差的浮动小于1s，则说明时间没有调整，直接退出。如果浮动大于1s，则重新更新mn\_now和ev\_rt\_now，再次判断时间差的浮动，如果判断了3次，浮动始终大于1s，说明时间被认为调整了，则需要更新周期事件，这种情况下不调整超时事件（超时事件都是根据mn\_now设置的，在have\_monotonic为1的情况下，mn\_now表示系统启动时间，不会被调整）。

如果宏定义EV\_USE\_MONOTONIC为0，或者have\_monotonic为0（sys\_clock\_gettime不支持CLOCK\_MONOTONIC）的话，mn\_now与ev\_rt\_now一样，也是日历时间。

这种情况下，更新ev\_rt\_now的值，将该值与之前的日历时间比较，如果时间被人调整了，则需要调整超时事件和周期事件。

7：fd\_event

在backend\_poll函数中，如果有些监视器的事件触发了，就会调用fd\_event函数，将触发的描述符fd和事件event记录到pending数组中。

void fd\_event **(**struct ev\_loop **\***loop**,** int fd**,** int revents**)**

**{**

ANFD **\***anfd **=** anfds **+** fd**;**

**if** **(**expect\_true **(!**anfd**->**reify**))**

fd\_event\_nocheck **(**loop, fd**,** revents**);**

**}**

已经触发而还没有处理的事件状态称为PENDING状态。在fd\_event函数中，根据fd找到相应的ANFD结构。然后就是：

**if** **(**expect\_true **(!**anfd**->**reify**))**

fd\_event\_nocheck **(**EV\_A\_ fd**,** revents**);**

fd\_event\_nocheck的代码如下，根据fd找到相应的ANFD结构，轮训其中的监视器链表，如果某监视器上的事件触发了，则调用ev\_feed\_event函数处理：

fd\_event\_nocheck **(**EV\_P\_ int fd**,** int revents**)**

**{**

ANFD **\***anfd **=** anfds **+** fd**;**

ev\_io **\***w**;**

**for** **(**w **=** **(**ev\_io **\*)**anfd**->**head**;** w**;** w **=** **(**ev\_io **\*)((**WL**)**w**)->**next**)**

**{**

int ev **=** w**->**events **&** revents**;**

**if** **(**ev**)**

ev\_feed\_event **(**EV\_A\_ **(**W**)**w**,** ev**);**

**}**

**}**

ev\_feed\_event代码如下：

void ev\_feed\_event **(**struct ev\_loop **\***loop**,** void **\***w**,** int revents**)**

**{**

W w\_ **=** **(**W**)**w**;**

int pri **=** ABSPRI **(**w\_**);**

**if** **(**expect\_false **(**w\_**->**pending**))**

pendings **[**pri**][**w\_**->**pending **-** 1**].**events **|=** revents**;**

**else**

**{**

w\_**->**pending **=** **++**pendingcnt **[**pri**];**

array\_needsize **(**ANPENDING**,** pendings **[**pri**],** pendingmax **[**pri**],** w\_**->**pending**,** EMPTY2**);**

pendings **[**pri**][**w\_**->**pending **-** 1**].**w **=** w\_**;**

pendings **[**pri**][**w\_**->**pending **-** 1**].**events **=** revents**;**

**}**

pendingpri **=** NUMPRI **-** 1**;**

**}**

(loop->pendingcnt)是一个一维整型数组，(loop->pendingcnt)[i]表示当前处于PENDING状态的优先级为i的监视器的个数。

(loop->pendings )是个二维数组，每个元素类型为ANPENDING，该结构的定义如下：

/\* stores the pending event set for a given watcher \*/

**typedef** struct

**{**

ev\_watcher **\***w**;**

int events**;**

**}** ANPENDING**;**

APPENDING结构记录了处于PENDING 状态的监视器以及触发的事件。(loop->pendings )数组，以优先级为第一维，以APPENDING为第二维。

在函数ev\_feed\_event中，判断w\_->pending的值，该值为0表示该监视器第一次被激活，不为0表示的是该监视器已经处于PENDING状态，而其具体的值，代表该监视器在pendings [pri]中的排名（从1开始），也就是(loop->pendingcnt) [pri]的值。

该值不为0，说明该监视器已经处于PENDING状态了，因此只需要：

pendings **[**pri**][**w\_**->**pending **-** 1**].**events **|=** revents**;**

如果该值为0，则

w\_**->**pending **=** **++**pendingcnt **[**pri**];**

array\_needsize **(**ANPENDING**,** pendings **[**pri**],** pendingmax **[**pri**],** w\_**->**pending**,** EMPTY2**);**

pendings **[**pri**][**w\_**->**pending **-** 1**].**w **=** w\_**;**

pendings **[**pri**][**w\_**->**pending **-** 1**].**events **=** revents**;**

8：ev\_invoke\_pending

void ev\_invoke\_pending **(**struct ev\_loop **\***loop **)**

**{**

pendingpri **=** NUMPRI**;**

**while** **(**pendingpri**)** */\* pendingpri possibly gets modified in the inner loop \*/*

**{**

**--**pendingpri**;**

**while** **(**pendingcnt **[**pendingpri**])**

**{**

ANPENDING **\***p **=** pendings **[**pendingpri**]** **+** **--**pendingcnt **[**pendingpri**];**

p**->**w**->**pending **=** 0**;**

EV\_CB\_INVOKE **(**p**->**w**,** p**->**events**);**

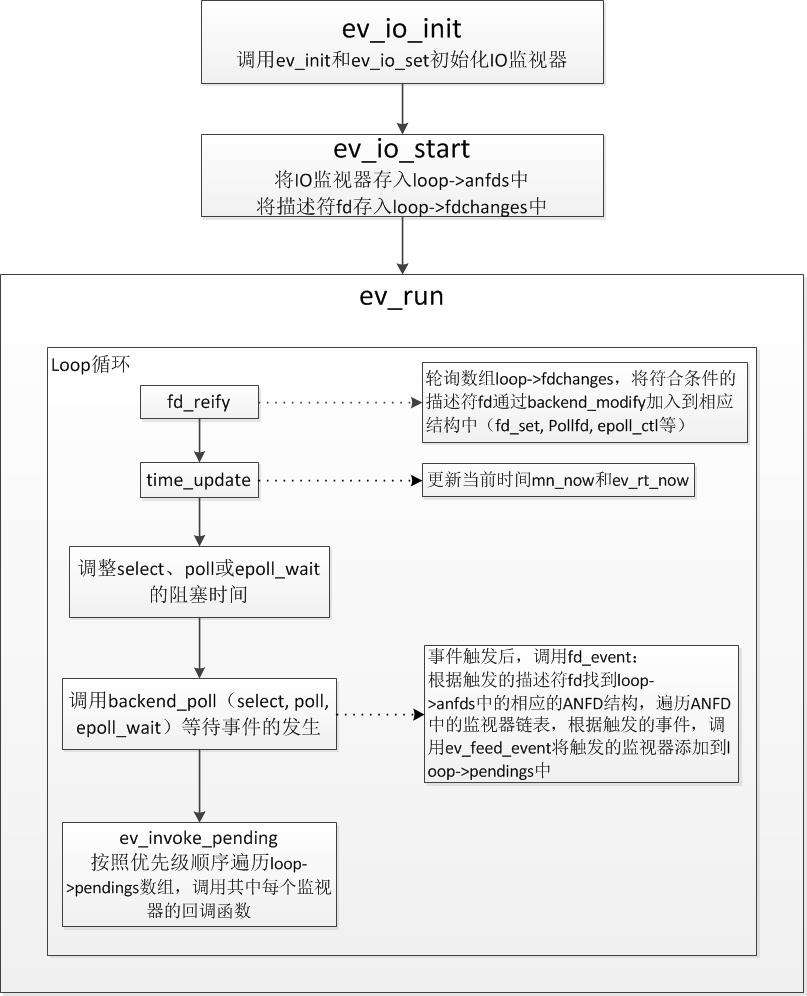
**}**

**}**

**}**

该函数主要是，调用所有当前处于PENDING状态的监视器的回调函数。根据优先级pendingpri从高到底，(loop->pendingcnt) [pendingpri]表示PENDING状态的，优先级为pendingpri的监视器个数。从后向前轮训(loop->pendings) [pendingpri]数组，调用每个监视器的回调函数。并且置w->pending = 0。

**二：总结：**



**三：例子**

ev\_io io\_w**;**

void io\_action**(**struct ev\_loop **\***main\_loop**,** ev\_io **\***io\_w**,** int e**)**

**{**

int rst**;**

char buf**[**1024**]** **=** **{**'\0'**};**

rst **=** read**(**io\_w**->**fd**,** buf**,** **sizeof(**buf**));**

**if(**rst **<=** 0**)**

**{**

close**(**io\_w**->**fd**);**

printf**(**"client over\n"**);**

ev\_io\_stop**(**main\_loop**,**io\_w**);**

**return;**

**}**

buf**[**1023**]** **=** '\0'**;**

printf**(**"Read in a string: %s \n"**,**buf**);**

write**(**io\_w**->**fd**,** buf**,** strlen**(**buf**));**

**}**

int socketfd**()**

**{**

int listenfd **=** socket**(**AF\_INET**,** SOCK\_STREAM**,** 0**);**

**if** **(**listenfd **<** 0**)**

**{**

perror**(**"socket error"**);**

**return** **-**1**;**

**}**

struct sockaddr\_in serveraddr**;**

struct sockaddr\_in clientaddr**;**

int addrlen **=** **sizeof(**struct sockaddr\_in**);**

serveraddr**.**sin\_family **=** AF\_INET**;**

serveraddr**.**sin\_addr**.**s\_addr **=** htonl**(**INADDR\_ANY**);**

serveraddr**.**sin\_port **=** htons**(**8898**);**

**if(**bind**(**listenfd**,** **(**struct sockaddr **\*)&**serveraddr**,** **sizeof(**struct sockaddr\_in**))** **<** 0**)**

**{**

perror**(**"bind error"**);**

**return** **-**1**;**

**}**

**if(**listen**(**listenfd**,** 5**)** **<** 0**)**

**{**

perror**(**"listen error"**);**

**return** **-**1**;**

**}**

int connectfd **=** 0**;**

connectfd **=** accept**(**listenfd**,** **(**struct sockaddr **\*)&**clientaddr**,** **(**socklen\_t **\*)&**addrlen**);**

**if(**connectfd **<** 0**)**

**{**

perror**(**"accept error"**);**

**return** **-**1**;**

**}**

**return** connectfd**;**

**}**

int main**()**

**{**

int fd **=** socketfd**();**

**if(**fd **<** 0**)** **return;**

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

ev\_init**(&**io\_w**,**io\_action**);**

ev\_io\_set**(&**io\_w**,**fd**,**EV\_READ**);**

ev\_io\_start**(**main\_loop**,&**io\_w**);**

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

**return;**

**}**