Libev中的信号监视器，用于监控信号的发生，因信号是异步的，所以Libev的处理方式是尽量的将异步信号同步化。异步信号的同步化方法主要有：signalfd、eventfd、pipe、sigwaitinfo等。这里Libev采用的是前三种方法，最终都是将对异步信号的处理，转化成对文件描述符的处理，也就是将ev\_signal转化为处理ev\_io。

**一：数据结构**

1：ev\_signal

**typedef** struct ev\_signal

**{**

int active**;**

int pending**;**

int priority**;**

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

void **(\***cb**)(**EV\_P\_ struct ev\_signal **\***w**,** int revents**);**

struct ev\_watcher\_list **\***next**;**

int signum**;**

**}** ev\_signal**;**

ev\_signal的结构跟ev\_io的结构十分类似，前6个成员是完全一样的，最后一个signum记录信号值。前六个成员构成了一个ev\_watcher\_list结构，因此信号监视器也是按照链表组织的。

2：ANSIG

**typedef** struct

**{**

sig\_atomic\_t volatile pending**;**

#if EV\_MULTIPLICITY

struct ev\_loop **\***loop**;**

#endif

ev\_watcher\_list **\***head**;**

**}** ANSIG**;**

static ANSIG signals **[**EV\_NSIG **-** 1**];**

ANSIG就是Libev内部用来组织ev\_signal的结构体，它的成员包括：pending表明该信号是否处于未决状态（触发但尚未处理），head表明该信号对应的监视器链表的头指针，另外，Libev不允许同一个信号出现在多个ev\_loop结构中，因此，如果支持多个ev\_loop的话，还有一个loop成员记录该信号对应的ev\_loop。

signals是ANSIG类型的数组，它的下标就是相应的信号值-1，因此，每个信号都有对应的ANSIG结构。

二：初始化ev\_signal

#define ev\_signal\_set(ev,signum\_) do {\

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

} while (0)

#define ev\_signal\_init(ev,cb,signum) do {\

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

ev\_signal\_set ((ev), (signum)); \

} while (0)

**三：使用signalfd处理信号**

各个系统支持的信号同步机制各有不同，针对多种信号同步机制，Libev采用下面的优先级循序：signalfd、eventfd、pipe。

signalfd是最简单方便的信号同步机制，可以很容易的将异步的信号的监听转化成对文件描述符的监听。下面首先看一下使用signalfd时的信号处理流程。

1：ev\_signal\_start

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

**{**

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

**return;**

assert **((**"libev: ev\_signal\_start called with illegal signal number"**,**

w**->**signum **>** 0 **&&** w**->**signum **<** EV\_NSIG**));**

#if EV\_MULTIPLICITY

assert **((**"libev: a signal must not be attached to two different loops"**,**

**!**signals **[**w**->**signum **-** 1**].**loop **||** signals **[**w**->**signum **-** 1**].**loop **==** loop**));**

signals **[**w**->**signum **-** 1**].**loop **=** loop**;**

#endif

#if EV\_USE\_SIGNALFD

**if** **(**sigfd **==** **-**2**)**

**{**

sigfd **=** signalfd **(-**1**,** **&**sigfd\_set**,** SFD\_NONBLOCK **|** SFD\_CLOEXEC**);**

**if** **(**sigfd **<** 0 **&&** errno **==** EINVAL**)**

sigfd **=** signalfd **(-**1**,** **&**sigfd\_set**,** 0**);** */\* retry without flags \*/*

**if** **(**sigfd **>=** 0**)**

**{**

fd\_intern **(**sigfd**);** */\* doing it twice will not hurt \*/*

sigemptyset **(&**sigfd\_set**);**

ev\_io\_init **(&**sigfd\_w**,** sigfdcb**,** sigfd**,** EV\_READ**);**

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

ev\_io\_start **(**EV\_A\_ **&**sigfd\_w**);**

ev\_unref **(**EV\_A**);** */\* signalfd watcher should not keep loop alive \*/*

**}**

**}**

**if** **(**sigfd **>=** 0**)**

**{**

sigaddset **(&**sigfd\_set**,** w**->**signum**);**

sigprocmask **(**SIG\_BLOCK**,** **&**sigfd\_set**,** 0**);**

signalfd **(**sigfd**,** **&**sigfd\_set**,** 0**);**

**}**

#endif

ev\_start **(**EV\_A\_ **(**W**)**w**,** 1**);**

wlist\_add **(&**signals **[**w**->**signum **-** 1**].**head**,** **(**WL**)**w**);**

**...**

**}**

在ev\_signal\_start中，首先对信号监视器w进行验证：

如果它已经被激活，也就是w->active不为0，直接返回；

信号监视器中的信号值应该处于合法范围(0, EV\_NSIG)内，否则进程退出；

该信号对应的ANSIG结构中记录的ev\_loop，应该就是参数loop，否则进程退出；

验证完成后，首先记录一下该信号所在的ev\_loop：

signals **[**w**->**signum **-** 1**].**loop **=** loop**;**

然后，根据宏EV\_USE\_SIGNALFD判断系统是否支持signalfd函数，根据loop->sigfd的值判断用户是否使用signalfd函数，在初始化ev\_loop的函数loop\_init (struct ev\_loop \*loop, unsigned int flags)中有：

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

因此，用户如果想使用signalfd函数，flags参数中必须有EVFLAG\_SIGNALFD，也就是在使用ev\_default\_loop或者ev\_loop\_new初始化ev\_loop时，必须指明EVFLAG\_SIGNALFD标志。

如果系统支持signalfd，并且loop->sigfd为-2的话，则开始调用signalfd函数（这是第一次调用signalfd），创建signalfd文件描述符。如果signalfd支持SFD\_NONBLOCK和SFD\_CLOEXEC标志的话，则直接在signalfd中设置，否则调用fd\_intern，使用fcntl设置（即使signalfd支持这俩标志，也会调用该函数重新设置一遍，无伤大雅）。注意，第一次调用signalfd时，信号集（sigset\_t）sigfd\_set尚未初始化，在下面初始化。

第一次调用signalfd成功之后，首先清空信号集sigfd\_set，然后将signalfd文件描述符加入到Libev内部的IO监视器（ev\_io）sigfd\_w中，并且启动sigfd\_w，注意这里设置sigfd\_w的优先级为最高优先级，回调函数为sigfdcb：

sigemptyset **(&**sigfd\_set**);**

ev\_io\_init **(&**sigfd\_w**,** sigfdcb**,** sigfd**,** EV\_READ**);**

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

ev\_io\_start **(**EV\_A\_ **&**sigfd\_w**);**

ev\_unref **(**EV\_A**);** */\* signalfd watcher should not keep loop alive \*/*

接下来，将信号w->signum加入到信号集sigfd\_set中，阻塞该信号，重新关联signalfd和sigfd\_set：

sigaddset **(&**sigfd\_set**,** w**->**signum**);**

sigprocmask **(**SIG\_BLOCK**,** **&**sigfd\_set**,** 0**);**

signalfd **(**sigfd**,** **&**sigfd\_set**,** 0**);**

之所以不在调用signalfd的时候阻塞该信号并关联signalfd描述符，是因为所有信号仅使用一个signalfd描述符sigfd，一个IO监视器sigfd\_w。当有多个信号监视器时，需要多次调用ev\_signal\_start，在第一次调用ev\_signal\_start成功之后，sigfd的值便已是大于等于0的整数了，这样只需要将信号阻塞，然后重新关联sigfd即可，而无需重新创建一个signalfd描述符并加入到IO监视器sigfd\_w中。

最后，激活该信号监视器w，并且将其加入到相应的ANSIG结构中：

ev\_start **(**EV\_A\_ **(**W**)**w**,** 1**);**

wlist\_add **(&**signals **[**w**->**signum **-** 1**].**head**,** **(**WL**)**w**);**

这样，如果使用signalfd监控信号，ev\_signal\_start函数的流程就结束了。接下来，就是监控IO监视器sigfd\_w了。当sigfd\_set中的一个或多个信号发生时，sigfd变成可读状态，IO监视器sigfd\_w触发，在ev\_run中，调用ev\_invoke\_pending时，就会调用它的回调函数sigfdcb。ev\_invoke\_pending的代码如下：

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**);**

EV\_FREQUENT\_CHECK**;**

**}**

**}**

**}**

该函数中，首先从最高优先级的pendings开始轮训，依次调用其中监视器的回调函数。因IO监视器sigfd\_w具有最高优先级，因此如果信号触发了，则sigfd\_w的回调函数sigfdcb会首先被调用到。

2：sigfdcb

static void sigfdcb **(**struct ev\_loop **\***loop, ev\_io **\***iow**,** int revents**)**

**{**

struct signalfd\_siginfo si**[**2**],** **\***sip**;** */\* these structs are big \*/*

**for** **(;;)**

**{**

ssize\_t res **=** read **(**sigfd**,** si**,** **sizeof** **(**si**));**

*/\* not ISO-C, as res might be -1, but works with SuS \*/*

**for** **(**sip **=** si**;** **(**char **\*)**sip **<** **(**char **\*)**si **+** res**;** **++**sip**)**

ev\_feed\_signal\_event **(**EV\_A\_ sip**->**ssi\_signo**);**

**if** **(**res **<** **(**ssize\_t**)sizeof** **(**si**))**

**break;**

**}**

**}**

该回调函数中，主要是读取sigfd中的信号信息。因struct signalfd\_siginfo结构比较大(128字节)，这里采用的技巧是每次read时最多只读取2个。

针对读取到的信号值，调用ev\_feed\_signal\_event函数。

3：ev\_feed\_signal\_event

void ev\_feed\_signal\_event **(**struct ev\_loop **\***loop**,** int signum**)**

**{**

WL w**;**

**if** **(**expect\_false **(**signum **<=** 0 **||** signum **>=** EV\_NSIG**))**

**return;**

**--**signum**;**

#if EV\_MULTIPLICITY

*/\* it is permissible to try to feed a signal to the wrong loop \*/*

*/\* or, likely more useful, feeding a signal nobody is waiting for\*/*

**if** **(**expect\_false **(**signals **[**signum**].**loop **!=** EV\_A**))**

**return;**

#endif

signals **[**signum**].**pending **=** 0**;**

**for** **(**w **=** signals **[**signum**].**head**;** w**;** w **=** w**->**next**)**

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

**}**

在ev\_feed\_signal\_event中，首先检查信号值是否处于合法范围(0, EV\_NSIG)内，然后检查该信号对应的ev\_loop是否就是当前的loop，如果不是则直接返回。

然后置signals [signum].pending为0，在signals中找到该信号的监视器列表，针对该列表中的所有监视器，调用ev\_feed\_event，将监视器加入到loop->pendings中。

注意，此时添加信号监视器到loop->pendings的流程，还是处于ev\_invoke\_pending函数的流程中的，因此，在ev\_invoke\_pending中，处理完sigfd\_w监视器后，接着就会处理到刚刚加到loop->pendings的信号监视器。从而信号自己的回调函数就会被调用到。

这样使用signalfd监控信号的完整流程就结束了。

**四：使用eventfd、pipe处理信号**

使用eventfd和pipe处理信号的基本思路是一样的，首先创建eventfd描述符或者管道pipe，使用IO监视器监听eventfd描述符或者pipe[0]，当信号发生时时，在信号处理程序中，写入eventfd描述符或者pipe[1]，从而触发IO监视器，调用回调函数pipecb处理信号。

1：ev\_signal\_start

首先看下，当不使用signalfd，或者调用signalfd失败时，ev\_signal\_start的流程：

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

**{**

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

**return;**

assert **((**"libev: ev\_signal\_start called with illegal signal number"**,**

w**->**signum **>** 0 **&&** w**->**signum **<** EV\_NSIG**));**

#if EV\_MULTIPLICITY

assert **((**"libev: a signal must not be attached to two different loops"**,**

**!**signals **[**w**->**signum **-** 1**].**loop **||** signals **[**w**->**signum **-** 1**].**loop **==** loop**));**

signals **[**w**->**signum **-** 1**].**loop **=** EV\_A**;**

#endif

**...**

ev\_start **(**EV\_A\_ **(**W**)**w**,** 1**);**

wlist\_add **(&**signals **[**w**->**signum **-** 1**].**head**,** **(**WL**)**w**);**

**if** **(!((**WL**)**w**)->**next**)**

# if EV\_USE\_SIGNALFD

**if** **(**sigfd **<** 0**)** */\*TODO\*/*

# endif

**{**

struct sigaction sa**;**

evpipe\_init **(**EV\_A**);**

sa**.**sa\_handler **=** ev\_sighandler**;**

sigfillset **(&**sa**.**sa\_mask**);**

sa**.**sa\_flags **=** SA\_RESTART**;** */\* if restarting works we save one iteration\*/*

sigaction **(**w**->**signum**,** **&**sa**,** 0**);**

**if** **(**origflags **&** EVFLAG\_NOSIGMASK**)**

**{**

sigemptyset **(&**sa**.**sa\_mask**);**

sigaddset **(&**sa**.**sa\_mask**,** w**->**signum**);**

sigprocmask **(**SIG\_UNBLOCK**,** **&**sa**.**sa\_mask**,** 0**);**

**}**

**}**

**}**

首先对信号监视器w进行验证，然后激活该信号监视器w，并且将其加入到相应的ANSIG结构中，该过程与使用signalfd的流程一样，不再赘述。

接下来，如果该监视器是当前信号的第一个监视器（((WL)w)->next == NULL），说明这是第一次监听该信号，需要创建该信号的信号处理函数，并且创建eventfd或pipe结构。

首先调用evpipe\_init初始化eventfd或pipe结构，暂且不表，下面详述。

然后调用sigaction建立该信号的处理函数为ev\_sighandler，并且在调用信号处理函数时，阻塞所有信号，且被信号中断的低速系统调用会被重启

sa**.**sa\_handler **=** ev\_sighandler**;**

sigfillset **(&**sa**.**sa\_mask**);**

sa**.**sa\_flags **=** SA\_RESTART**;** */\* if restarting works we save one iteration \*/*

sigaction **(**w**->**signum**,** **&**sa**,** 0**);**

如果在初始化ev\_loop时指定了EVFLAG\_NOSIGMASK标志的话，还需要明确将监听的信号解除阻塞。

**if** **(**origflags **&** EVFLAG\_NOSIGMASK**)**

**{**

sigemptyset **(&**sa**.**sa\_mask**);**

sigaddset **(&**sa**.**sa\_mask**,** w**->**signum**);**

sigprocmask **(**SIG\_UNBLOCK**,** **&**sa**.**sa\_mask**,** 0**);**

**}**

2：evpipe\_init

该函数用来创建eventfd描述符或者pipe，并将信号监视器转换为IO监视器pipe\_w。代码如下：

static void evpipe\_init **(**struct ev\_loop **\***loop**)**

**{**

**if** **(!**ev\_is\_active **(&**pipe\_w**))**

**{**

int fds **[**2**];**

# if EV\_USE\_EVENTFD

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

fds **[**1**]** **=** eventfd **(**0**,** EFD\_NONBLOCK **|** EFD\_CLOEXEC**);**

**if** **(**fds **[**1**]** **<** 0 **&&** errno **==** EINVAL**)**

fds **[**1**]** **=** eventfd **(**0**,** 0**);**

**if** **(**fds **[**1**]** **<** 0**)**

# endif

**{**

**while** **(**pipe **(**fds**))**

ev\_syserr **(**"(libev) error creating signal/async pipe"**);**

fd\_intern **(**fds **[**0**]);**

**}**

evpipe **[**0**]** **=** fds **[**0**];**

**if** **(**evpipe **[**1**]** **<** 0**)**

evpipe **[**1**]** **=** fds **[**1**];** */\* first call, set write fd \*/*

**else**

**{**

*/\* on subsequent calls, do not change evpipe [1] \*/*

*/\* so that evpipe\_write can always rely on its value. \*/*

*/\* this branch does not do anything sensible on windows, \*/*

*/\* so must not be executed on windows \*/*

dup2 **(**fds **[**1**],** evpipe **[**1**]);**

close **(**fds **[**1**]);**

**}**

fd\_intern **(**evpipe **[**1**]);**

ev\_io\_set **(&**pipe\_w**,** evpipe **[**0**]** **<** 0 **?** evpipe **[**1**]** **:** evpipe **[**0**],** EV\_READ**);**

ev\_io\_start **(**EV\_A\_ **&**pipe\_w**);**

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

**}**

**}**

所有信号使用一个IO监视器pipe\_w，如果pipe\_w已经处于激活状态，则说明相应的结构已经创建好了，直接返回即可。

然后根据宏EV\_USE\_EVENTFD判断系统是否支持eventfd，如果支持，则调用eventfd创建eventfd描述符，如果不支持，或者调用eventfd失败，则调用pipe创建管道，并设置管道读端描述符的FD\_CLOEXEC和O\_NONBLOCK标志。

使用evpipe[0]记录读描述符，evpipe[1]记录写描述符，如果使用eventfd，则evpipe[0]为-1，evpipe[1]为eventfd描述符，读写描述符都是eventfd。

调用fd\_intern，使用fcntl设置写描述符evpipe[1]的FD\_CLOEXEC和O\_NONBLOCK标志。

最后启动内部IO监视器pipe\_w，监听读描述符：

ev\_io\_set **(&**pipe\_w**,** evpipe **[**0**]** **<** 0 **?** evpipe **[**1**]** **:** evpipe **[**0**],** EV\_READ**);**

ev\_io\_start **(**EV\_A\_ **&**pipe\_w**);**

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

注意，pipe\_w监视器的初始化，在loop\_init中就已经做了：

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

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

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

#endif

这里设置pipe\_w的回调函数为pipecb，优先级为最高优先级EV\_MAXPRI。

当信号产生时，就会调用到信号处理函数ev\_sighandler，改处理函数仅仅就是调用函数ev\_feed\_signal而已。

3：ev\_feed\_signal

void ev\_feed\_signal **(**int signum**)**

**{**

#if EV\_MULTIPLICITY

struct ev\_loop **\***loop**;**

loop **=** signals **[**signum **-** 1**].**loop**;**

**if** **(!**loop**)**

**return;**

#endif

signals **[**signum **-** 1**].**pending **=** 1**;**

evpipe\_write **(**loop**,** **&**sig\_pending**);**

**}**

该函数首先根据信号值得到该信号所在的ev\_loop，然后置该信号对应的ANSIG的pending为1，最后调用evpipe\_write函数。

4：evpipe\_write

void evpipe\_write **(**struct ev\_loop **\***loop**,** sig\_atomic\_t volatile **\***flag**)**

**{**

**if** **(**expect\_true **(\***flag**))**

**return;**

**\***flag **=** 1**;**

pipe\_write\_skipped **=** 1**;**

**if** **(**pipe\_write\_wanted**)**

**{**

int old\_errno**;**

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

old\_errno **=** errno**;** */\* save errno because write will clobber it\*/*

#if EV\_USE\_EVENTFD

**if** **(**evpipe **[**0**]** **<** 0**)**

**{**

uint64\_t counter **=** 1**;**

write **(**evpipe **[**1**],** **&**counter**,** **sizeof** **(**uint64\_t**));**

**}**

**else**

#endif

**{**

write **(**evpipe **[**1**],** **&(**evpipe **[**1**]),** 1**);**

**}**

errno **=** old\_errno**;**

**}**

**}**

注意，当在一个loop中有多个信号发生时，也仅需要产生一个事件而已。本函数主要作用就是当信号发生时向写描述符写入一个事件，即可触发pipe\_w中的读事件，表明有一个或者多个信号触发了。但是因为信号处理函数的调用时机是完全随机的，因此，需要有一定的手段保证代码的安全性。

a：sig\_pending

该值表示是否有信号处于未决状态（触发但尚未处理）。该值在初始化ev\_loop时置为0，调用evpipe\_write时，会首先判断该值是否为1。如果该值已经为1，表示已经有监听的信号处于未决状态了，无需再向写描述符写入事件了。该值直到调用pipe\_w的回调函数pipecb时，消费掉该事件之后才重置为0，表明从此刻起，若有监听信号触发，才能继续向写描述符写入事件。

b：pipe\_write\_wanted和pipe\_write\_skipped

pipe\_write\_wanted表明是否允许向写描述符写入事件，pipe\_write\_skipped表明是否信号发生了，却因pipe\_write\_wanted的关系被暂时忽略了。这两个值在初始化ev\_loop时置为0。

在evpipe\_write中，首先置pipe\_write\_skipped为1，如果pipe\_write\_wanted此时为0，evpipe\_write直接返回，这就表明触发信号暂时被忽略掉了（仅仅是暂时的）。否则，重置pipe\_write\_skipped为0，向写描述符写入事件。

在ev\_run中，调用backend\_poll之前，会将pipe\_write\_wanted置为1，表明此刻起写描述符才能接受写入事件，如果此刻之前有监听信号发生的话，则会在信号处理函数调用的evpipe\_write中，置pipe\_write\_skipped为1表示信号暂时忽略掉。

在ev\_run中调用backend\_poll后立即置pipe\_write\_wanted为0。如果pipe\_write\_skipped为1，表明有信号被忽略了，调用ev\_feed\_event，直接将pipe\_w标记为pending状态，将pipe\_w加入到loop->pendings中：

**do{**

ev\_tstamp waittime **=** 0.**;**

pipe\_write\_wanted **=** 1**;**

**if** **(**expect\_true **(!(**flags **&** EVRUN\_NOWAIT **||** idleall **||** **!**activecnt **||** pipe\_write\_skipped**)))**

**{**

waittime **=** MAX\_BLOCKTIME**;**

**...**

**}**

backend\_poll **(**EV\_A\_ waittime**);**

pipe\_write\_wanted **=** 0**;** */\* just an optimisation, no fence needed \*/*

**if** **(**pipe\_write\_skipped**)**

**{**

assert **((**"libev: pipe\_w not active, but pipe not written"**,** ev\_is\_active **(&**pipe\_w**)));**

ev\_feed\_event **(**EV\_A\_ **&**pipe\_w**,** EV\_CUSTOM**);**

**}**

EV\_INVOKE\_PENDING**;**

**}while(**expect\_true **(**

activecnt

**&&** **!**loop\_done

**&&** **!(**flags **&** **(**EVRUN\_ONCE **|** EVRUN\_NOWAIT**))**

**))**

上面就是ev\_run中的相关逻辑，注意：调用backend\_poll之后，如果在检测pipe\_write\_skipped之后才有信号发生的话，此时在evpipe\_write中仅设置pipe\_write\_skipped为1后就返回。然后进入下次循环，因pipe\_write\_skipped为1，所以waittime为0，backend\_poll会立即返回，处理pipe\_w的激活事件。

PS：现在还没有想明白，为什么需要pipe\_write\_wanted和pipe\_write\_skipped这两个标志，感觉sig\_pending已经足够了。

在向写描述符写入事件之后，pipe\_w监视器触发，调用回调函数pipecb。

5：pipecb

static void pipecb **(**struct ev\_loop **\***loop**,** ev\_io **\***iow**,** int revents**)**

**{**

int i**;**

**if** **(**revents **&** EV\_READ**)**

**{**

#if EV\_USE\_EVENTFD

**if** **(**evpipe **[**0**]** **<** 0**)**

**{**

uint64\_t counter**;**

read **(**evpipe **[**1**],** **&**counter**,** **sizeof** **(**uint64\_t**));**

**}**

**else**

#endif

**{**

char dummy**[**4**];**

read **(**evpipe **[**0**],** **&**dummy**,** **sizeof** **(**dummy**));**

**}**

**}**

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

#if EV\_SIGNAL\_ENABLE

**if** **(**sig\_pending**)**

**{**

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

**for** **(**i **=** EV\_NSIG **-** 1**;** i**--;** **)**

**if** **(**expect\_false **(**signals **[**i**].**pending**))**

ev\_feed\_signal\_event **(**EV\_A\_ i **+** 1**);**

**}**

#endif

**...**

**}**

在pipecb中，首先从eventfd描述符或者pipe[0]中消费掉事件。然后轮训signals数组中每个ANSIG结构的pending字段，只要是信号触发了，则该字段一定为1，从而可以调用ev\_feed\_signal\_event处理该信号。剩下的流程就与使用signalfd时一样了，不再赘述。

**四：ev\_signal\_stop**

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

**{**

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

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

**return;**

wlist\_del **(&**signals **[**w**->**signum **-** 1**].**head**,** **(**WL**)**w**);**

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

**if** **(!**signals **[**w**->**signum **-** 1**].**head**)**

**{**

#if EV\_MULTIPLICITY

signals **[**w**->**signum **-** 1**].**loop **=** 0**;** */\* unattach from signal \*/*

#endif

#if EV\_USE\_SIGNALFD

**if** **(**sigfd **>=** 0**)**

**{**

sigset\_t ss**;**

sigemptyset **(&**ss**);**

sigaddset **(&**ss**,** w**->**signum**);**

sigdelset **(&**sigfd\_set**,** w**->**signum**);**

signalfd **(**sigfd**,** **&**sigfd\_set**,** 0**);**

sigprocmask **(**SIG\_UNBLOCK**,** **&**ss**,** 0**);**

**}**

**else**

#endif

signal **(**w**->**signum**,** SIG\_DFL**);**

**}**

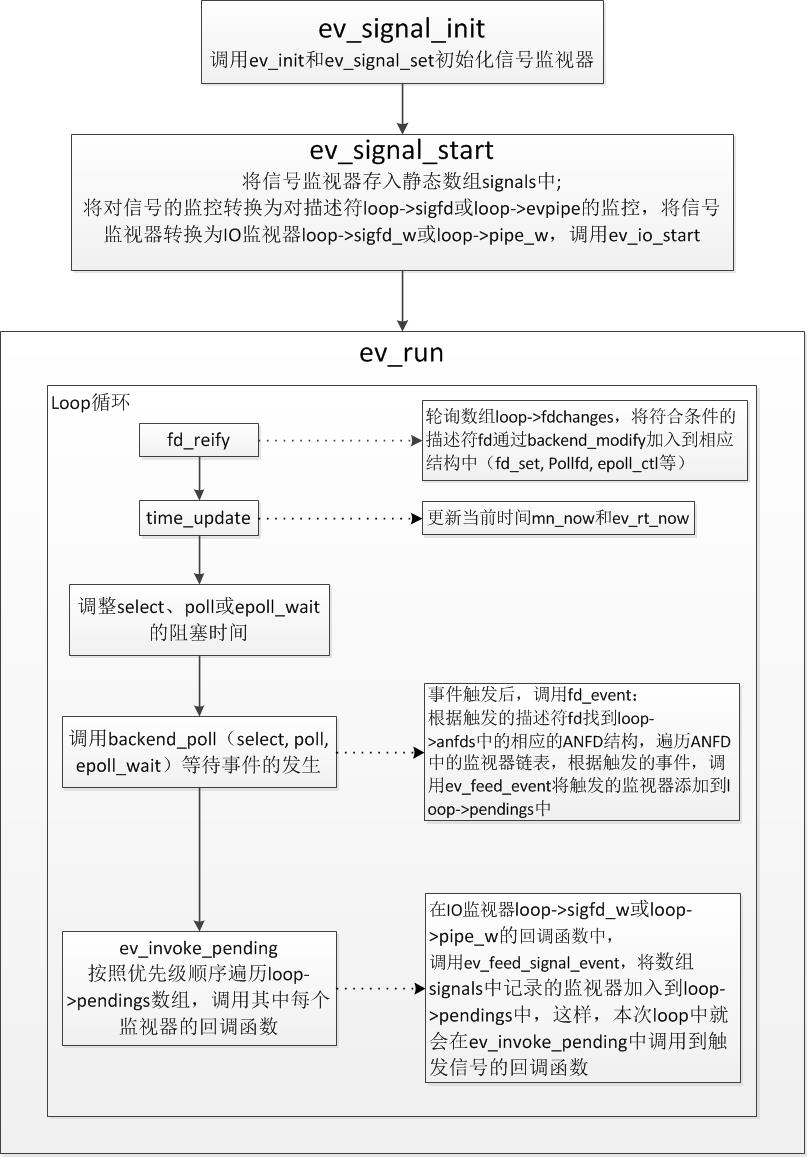
**}**

在ev\_signal\_stop中，首先调用clear\_pending清除监视器w在loop->pendings中的状态，置w->pending = 0。这里有个技巧是将loop->pendings中，原w所在位置直接赋值为内部伪监视器pending\_w，pending\_w的回调函数为空函数，会直接返回。

调用wlist\_del，将w从该信号的监视器列表中删除，调用ev\_stop注销改监视器；

如果相应信号的监视器列表空了，则首先signals [w->signum - 1].loop = 0，然后恢复该信号的处理方式：若使用signalfd，则取消阻塞该信号，将该信号从sigfd描述符关联的信号集中删除；若不使用signalfd，则直接恢复该信号的处理方式为默认方式。

**五：总结**



**六：例子**

ev\_signal signal\_w**;**

void signal\_action**(**struct ev\_loop **\***main\_loop**,**ev\_signal **\***signal\_w**,**int e**)**

**{**

puts**(**"\nin signal cb \n"**);**

**}**

int main**()**

**{**

struct ev\_loop **\***main\_loop **=** ev\_default\_loop**(**EVFLAG\_SIGNALFD**);**

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

ev\_signal\_set**(&**signal\_w**,**SIGINT**);**

ev\_signal\_start**(**main\_loop**,&**signal\_w**);**

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

**return** 0**;**

**}**

结果：

#./a.out

**^**C

in signal cb

**^**C

in signal cb

**^**\Quit **(**core dumped**)**