## Swoole版本：1.7.4-stable

Pipe（管道）用于进程之间的数据交互，Linux系统本身提供了pipe函数用于创建一个半双工通信管道，而在swoole中也利用eventfd和unix sock封装了两种管道，使得进程间的通信更加灵活。

Swoole提供了一个基本结构体swPipe用于操作不同类型的Pipe，其声明在swoole.h文件的315 – 326 行，声明如下：

typedef struct \_swPipe

{

void \*object;

int blocking;

double timeout;

int (\*read)(struct \_swPipe \*, void \*recv, int length);

int (\*write)(struct \_swPipe \*, void \*send, int length);

int (\*getFd)(struct \_swPipe \*, int isWriteFd);

int (\*close)(struct \_swPipe \*);

} swPipe;

其中，首部的void \*object指向具体的管道结构体（Base，Eventfd，Unsock），int blocking标记该管道是否是阻塞类型，double timeout为阻塞的超时时间，余下四个函数用于对管道进行读写、关闭等操作。

同时Swoole声明了三个创建函数用于创建不同类型的管道，其声明在swoole.h文件的328 – 330 行，声明如下：

int swPipeEventfd\_create(swPipe \*p, int blocking, int semaphore, int timeout);

int swPipeUnsock\_create(swPipe \*p, int blocking, int protocol);

第一个函数用于创建一个PipeBase管道，第二个参数标记该管道是否阻塞。

第二个函数用于创建一个PipeEventfd管道，第三个函数用于创建一个PipeUnsock管道，第三个参数protocol指定unix socket使用的协议类型。

一．基于pipe函数的PipeBase

前面提到过，Linux本身提供了一个pipe函数用于创建一个管道，swoole基于这个函数进行了一层封装用于简化操作，并命名为PipeBase，其结构体声明在PipeBase.c文件中，声明如下：

typedef struct \_swPipeBase

{

int pipes[2];

} swPipeBase;

其中的 int pipes[2] 用于存放pipe的读端fd和写端fd。

创建一个PipeBase管道的函数声明在swoole.h文件的328行，声明如下：

int swPipeBase\_create(swPipe \*p, int blocking);

第二个参数标记该管道是否阻塞。该函数具体定义在PipeBase.c文件中，其核心代码如下：

swPipeBase \*object = sw\_malloc(sizeof(swPipeBase));

if (object == NULL)

{

return -1;

}

p->blocking = blocking;

ret = pipe(object->pipes);

if (ret < 0)

{

swWarn("pipe create fail. Error: %s[%d]", strerror(errno), errno);

return -1;

}

else

{

//Nonblock

if (blocking == 0)

{

swSetNonBlock(object->pipes[0]);

swSetNonBlock(object->pipes[1]);

}

else

{

p->timeout = -1;

}

p->object = object;

p->read = swPipeBase\_read;

p->write = swPipeBase\_write;

p->getFd = swPipeBase\_getFd;

p->close = swPipeBase\_close;

}

源码解释：创建一个PipeBase结构体，指定管道阻塞类型为blocking参数，并调用pipe函数创建一个linux pipe，获取pipe fd。如果创建管道成功，若管道为非阻塞模式，则调用swSetNonBlock函数（声明于swoole.h文件的641行，定义于Base.c文件的507 – 529行，该函数的分析在本章末尾）设置两个fd为非阻塞；若管道为阻塞模式，则指定timeout为-1.

PipeBase有四个操作函数分别用于读、写、获取描述符以及关闭管道。其声明如下：

static int swPipeBase\_read(swPipe \*p, void \*data, int length);

static int swPipeBase\_write(swPipe \*p, void \*data, int length);

static int swPipeBase\_getFd(swPipe \*p, int isWriteFd);

static int swPipeBase\_close(swPipe \*p);

1. swPipeBase\_read函数核心源码如下：

if (p->blocking == 1 && p->timeout > 0)

{

if (swSocket\_wait(object->pipes[0], p->timeout \* 1000, SW\_EVENT\_READ) < 0)

{

return SW\_ERR;

}

}

return read(object->pipes[0], data, length);

源码解释：如果该管道为阻塞模式，且指定了timeout时间，则调用swSocket\_wait函数（定义于Base.c文件中的236 – 268行，该函数分析在本章末尾）执行等待，若超时，则返回SW\_ERR；否则，直接调用read方法读取数据（此时，若管道为非阻塞模式，因已经指定了fd的options，所以read方法不会阻塞；若为阻塞模式，则read函数会一直阻塞到有数据可读）

1. swPipeBase\_write函数直接调用write函数写出数据
2. swPipeBase\_getFd函数根据 isWriteFd 判定返回 写fd 或者 读fd
3. swPipeBase\_close函数调用close函数关闭描述符并释放swPipeBase内存。

二．基于eventfd函数的PipeEventfd

Eventfd是Linux提供的一个事件提醒（event notification）功能，eventfd函数创建一个对象用于事件的等待/提醒，可以从内核态发送通知到用户态。一个eventfd创建的对象包括一个无符号的64bit整型计数器，该计数器被内核持有。

eventfd函数的第二个参数用于指定flags，其中EFD\_NONBLOCK指定eventfd是否非阻塞，EFD\_SEMAPHORE用于指定eventfd的读写效果。

Swoole用一个结构体swPipeEventfd代表一个eventfd类型的管道，其声明如下：

typedef struct \_swPipeEventfd

{

int event\_fd;

} swPipeEventfd;

其中event\_fd就是用eventfd函数获得的描述符。

创建一个swPipeEventfd的函数声明在swoole.h的329行，声明如下：

int swPipeEventfd\_create(swPipe \*p, int blocking, int semaphore, int timeout);

其中第三个参数semaphore标记该管道是否仅用于提供通知，第四个参数timeout指定阻塞超时的时间。

该函数具体定义在PipeEventfd.c文件，其核心代码如下：

swPipeEventfd \*object = sw\_malloc(sizeof(swPipeEventfd));

if (object == NULL)

{

return -1;

}

flag = EFD\_NONBLOCK;

if (blocking == 1)

{

if (timeout > 0)

{

flag = 0;

p->timeout = -1;

}

else

{

p->timeout = timeout;

}

}

#ifdef EFD\_SEMAPHORE

if (semaphore == 1)

{

flag |= EFD\_SEMAPHORE;

}

#endif

p->blocking = blocking;

efd = eventfd(0, flag);

源码解释：创建一个swPipeEventfd对象，默认设置管道为非阻塞的，若参数blocking指定管道为阻塞，则指定timeout的值。如果指定了semaphore参数，则将EFD\_SEMAPHORE加入flags中。最后调用eventfd函数获取event\_fd.

这里需要特别说明EFD\_SEMAPHORE参数。如果一个eventfd被指定了该参数，当eventfd的计数器不为0时，对eventfd的read操作会读到一个8byte长度的整数1，然后计数器的值减1；如果没有指定该参数，当eventfd的计数器不为0时，对eventfd的read操作会将计数器的值读出（8 byte的整数），并将计数器置0.

PipeEventfd同样有四个操作函数分别用于读、写、获取描述符以及关闭管道。其声明如下

static int swPipeEventfd\_read(swPipe \*p, void \*data, int length);

static int swPipeEventfd\_write(swPipe \*p, void \*data, int length);

static int swPipeEventfd\_getFd(swPipe \*p, int isWriteFd);

static int swPipeEventfd\_close(swPipe \*p);

1. swPipeEventfd\_read核心源码如下：

if (p->blocking == 1 && p->timeout > 0)

{

if (swSocket\_wait(object->event\_fd, p->timeout \* 1000, SW\_EVENT\_READ) < 0)

{

return SW\_ERR;

}

}

while (1)

{

ret = read(object->event\_fd, data, sizeof(uint64\_t));

if (ret < 0 && errno == EINTR)

{

continue;

}

break;

}

源码解释：如果管道为可阻塞且timeout大于0，则调用swSocket\_wait函数（定义于Base.c文件中的236 – 268行，该函数分析在本章末尾）执行等待。若超时，则返回SW\_ERR；否则，循环调用read函数直到读取到数据。

1. swPipeEventfd\_write函数循环调用write函数直到写入数据成功。
2. swPipeEventfd\_getFd函数返回event\_fd
3. swPipeEventfd\_close函数调用close关闭event\_fd并释放内存。

三．基于unix socket的PipeUnsock

PipeUnsock使用了socketpair函数和AF\_UNIX（Unix Socket）来创建一个全双工的“管道”。这其实类似于Linux的pipe函数，不同的是文件描述符fd换成了套接字描述符sockfd，半双工通讯变成了全双工通讯。

swPipeUnsock结构体声明在PipeUnsock.c文件中，其声明如下：

typedef struct \_swPipeUnsock

{

int socks[2];

} swPipeUnsock;

其中socks用于存放两个socket套接字

创建一个swPipeUnsock的函数声明在swoole.h文件的330行，声明如下：

int swPipeUnsock\_create(swPipe \*p, int blocking, int protocol);

其中第二个参数指定socket使用的协议族（TCP/UDP）

其核心代码如下：

p->blocking = blocking;

ret = socketpair(AF\_UNIX, protocol, 0, object->socks);

if (ret < 0)

{

swWarn("socketpair() failed. Error: %s [%d]", strerror(errno), errno);

return SW\_ERR;

}

else

{

//Nonblock

if (blocking == 0)

{

swSetNonBlock(object->socks[0]);

swSetNonBlock(object->socks[1]);

}

int sbsize = SwooleG.unixsock\_buffer\_size;

setsockopt(object->socks[1], SOL\_SOCKET, SO\_SNDBUF, &sbsize, sizeof(sbsize));

setsockopt(object->socks[1], SOL\_SOCKET, SO\_RCVBUF, &sbsize, sizeof(sbsize));

setsockopt(object->socks[0], SOL\_SOCKET, SO\_SNDBUF, &sbsize, sizeof(sbsize));

setsockopt(object->socks[0], SOL\_SOCKET, SO\_RCVBUF, &sbsize, sizeof(sbsize));

}

源码解释：设置管道阻塞类型，并调用socketpair获取两个套接字，并指定套接字类型为AF\_UNIX，如果管道为非阻塞类型，则调用swSetNonBlock函数设置套接字属性。然后指定socket buffer 的大小（SwooleG是一个swServerG结构体，用于存放一些全局变量，该对象声明在Server.c中，在此不作分析），并调用setsockopt函数设置套接字选项。

swPipeUnsock的操作函数和swPipeBase基本一样，在此不再分析。

至此，Swoole的Pipe模块已分析完成。

附录：Base.c中相关函数的分析

1. **swSetNonBlock**

函数原型：void swSetNonBlock(int sock); （swoole.h 641行）

核心代码：（Base.c ）

do

{

opts = fcntl(sock, F\_GETFL);

}

while(opts <0 && errno == EINTR);

if (opts < 0)

{

swWarn("fcntl(sock,GETFL) fail");

}

opts = opts | O\_NONBLOCK;

do

{

ret = fcntl(sock, F\_SETFL, opts);

}

while(ret <0 && errno == EINTR);

源码解释：调用fcntl函数，通过F\_GETFL命令获得描述符的状态flags（没有一个比较好的翻译……百度翻译成旗标）；将O\_NONBLOCK（非阻塞标记）加入flags，并通过fcntl函数的F\_SETFL命令设置描述符的状态flags。

1. **swSocket\_wait**

函数原型：int swSocket\_wait(int fd, int timeout\_ms, int events) （swoole.h 648行）

核心代码：（Base.c 236 – 268行）

struct pollfd event;

event.fd = fd;

event.events = 0;

if (events & SW\_EVENT\_READ)

{

event.events |= POLLIN;

}

if (events & SW\_EVENT\_WRITE)

{

event.events |= POLLOUT;

}

while (1)

{

int ret = poll(&event, 1, timeout\_ms);

if (ret == 0)

{

return SW\_ERR;

}

else if (ret < 0 && errno != EINTR)

{

swWarn("poll() failed. Error: %s[%d]", strerror(errno), errno);

return SW\_ERR;

}

else

{

return SW\_OK;

}

}

源码解释：创建 pollfd 结构体，指定监听的fd，并根据events的值指定需要监听的事件（读事件or写事件）。接着在循环中调用poll函数监听，并指定poll函数的超时时间为timeout\_ms参数。若成功监听到事件，则返回SW\_OK，否则返回SW\_ERR。