## Swoole版本：1.7.5-stable

Reactor模块可以说是Swoole中最核心的模块之一，正是这些reactor模型为swoole提供了异步操作的基础。Swoole中根据不同的内核函数，提供了四种Reactor封装，ReactorEpoll，ReactorKqueue，ReactorPoll和ReactorSelect。同时，Swoole通过结构体swReactor封装了对于reactor的操作函数和基本属性。本章，我将分析swReactor以及四种Reactor模型中的ReactorEpoll，并回顾一下epoll的相关知识。

一．swReactor

swReactor结构体声明在swoole.h的698 – 721行，其声明如下：

struct swReactor\_s

{

void \*object;

void \*ptr; //reserve

uint32\_t event\_num;

uint32\_t max\_event\_num;

uint16\_t id; //Reactor ID

uint16\_t flag; //flag

char running;

swReactor\_handle handle[SW\_MAX\_FDTYPE]; //默认事件

swReactor\_handle write\_handle[SW\_MAX\_FDTYPE]; //扩展事件1(一般为写事件)

swReactor\_handle error\_handle[SW\_MAX\_FDTYPE]; //扩展事件2(一般为错误事件,如socket关闭)

int (\*add)(swReactor \*, int fd, int fdtype);

int (\*set)(swReactor \*, int fd, int fdtype);

int (\*del)(swReactor \*, int fd);

int (\*wait)(swReactor \*, struct timeval \*);

void (\*free)(swReactor \*);

int (\*setHandle)(swReactor \*, int fdtype, swReactor\_handle);

void (\*onTimeout)(swReactor \*); //发生超时时

void (\*onFinish)(swReactor \*); //完成一次轮询

};

typedef struct swReactor\_s swReactor; // swoole.h 354行

typedef int (\*swReactor\_handle)(swReactor \*reactor, swEvent\*event); // swoole.h 355行

属性分析：object存放实际reactor模型的内存地址，reserve ， event\_num存放现有的事件数目，max\_event\_num存放允许持有的最大事件数目，id用于存放对应reactor的id，flag为标记位，running用于标记该reactor是否正在运行，接下来三个数组用于存放需要监听的事件的响应回调函数，余下的都是对应reactor的操作函数，这些函数将在具体的Reactor模型中被实现并赋值。

对应Reactor的部分通用的操作函数声明在swoole.h文件中827 – 873行。首先是一个枚举类型SW\_EVENTS，用于指定对应的事件id，声明如下：

enum SW\_EVENTS

{

SW\_EVENT\_DEAULT = 256,

SW\_EVENT\_READ = 1u << 9,

SW\_EVENT\_WRITE = 1u << 10,

SW\_EVENT\_ERROR = 1u << 11,

};

SW\_EVENTS中指定了四种事件类型：DEFAULT代表默认事件，READ和WRITE分别代表读写事件，ERROR代表错误。

接着是四个操作事件类型的函数：

// 过滤fdtype中的读、写、错误事件标记

static sw\_inline int swReactor\_fdtype(int fdtype)

{

return fdtype & (~SW\_EVENT\_READ) & (~SW\_EVENT\_WRITE) & (~SW\_EVENT\_ERROR);

}

// 判定是否为读事件和其他swFd\_type类型的监听

static sw\_inline int swReactor\_event\_read(int fdtype)

{

return (fdtype < SW\_EVENT\_DEAULT) || (fdtype & SW\_EVENT\_READ);

}

// 判定是否为事件监听

static sw\_inline int swReactor\_event\_write(int fdtype)

{

return fdtype & SW\_EVENT\_WRITE;

}

// 判定是否为错误事件监听

static sw\_inline int swReactor\_event\_error(int fdtype)

{

return fdtype & SW\_EVENT\_ERROR;

}

接着是四个个通用操作函数，函数声明如下：

int swReactor\_auto(swReactor \*reactor, int max\_event);

int swReactor\_receive(swReactor \*reactor, swEvent \*event);

int swReactor\_setHandle(swReactor \*, int, swReactor\_handle);

swReactor\_handle swReactor\_getHandle(swReactor \*reactor, int event\_type, int fdtype);

这四个函数分别用于自动创建可用类型的reactor模型、从reactor接收到的swEvent中读取数据、设置Reactor的回调函数、获得Reactor的回调函数。函数的具体声明在ReactorBase.c文件中。

这里补充说明swEvent结构体，其声明在swoole.h文件的311 – 317 行，声明如下：

typedef struct \_swEvent

{

int fd; // 描述符

int16\_t from\_id; // 来自哪个reactor

uint8\_t type; // 描述符类型

void \*object; // 数据域

} swEvent;

在ReactorBase.c文件中，具体实现了Reactor的四个通用操作函数。第一个函数是swReactor\_auto，其核心代码如下：

int swReactor\_auto(swReactor \*reactor, int max\_event) // 第二个参数设置允许监听的最大事件数

int ret;

#ifdef HAVE\_EPOLL

ret = swReactorEpoll\_create(reactor, max\_event);

#elif defined(HAVE\_KQUEUE)

ret = swReactorKqueue\_create(reactor, max\_event);

#elif defined(SW\_MAINREACTOR\_USE\_POLL)

ret = swReactorPoll\_create(reactor, max\_event);

#else

ret = swReactorSelect\_create(SwooleG.main\_reactor)

#endif

return ret;

源码解释：根据环境编译中定义的参数决定使用哪种类型的reactor模型，主线程默认使用poll模型。

这里需要提前介绍一个枚举类型swFd\_type。枚举类型swFd\_type指定了描述符fd的一些特殊类型，这些类型主要用于reactor直接辨识某个fd类型的回调函数（同类型的fd共用一个回调函数）。该枚举类型声明在swoole.h文件中的165 – 179 行，如下：

enum swFd\_type

{

SW\_FD\_TCP = 0, //tcp socket

SW\_FD\_LISTEN = 1, //server socket

SW\_FD\_CLOSE = 2, //socket closed

SW\_FD\_ERROR = 3, //socket error

SW\_FD\_UDP = 4, //udp socket

SW\_FD\_PIPE = 5, //pipe

SW\_FD\_WRITE = 7, //fd can write

SW\_FD\_TIMER = 8, //timer fd

SW\_FD\_AIO = 9, //linux native aio

SW\_FD\_SEND\_TO\_CLIENT = 10, //sendto client

SW\_FD\_SIGNAL = 11, //signalfd

SW\_FD\_DNS\_RESOLVER = 12, //dns resolver

};

swReactor\_getHandle函数和swReactor\_setHandle函数分别用于获取和设置相应的回调函数。swReactor\_getHandle函数的核心代码如下：

if (event\_type == SW\_EVENT\_WRITE)

{

//默认可写回调函数SW\_FD\_WRITE

return (reactor->write\_handle[fdtype] != NULL) ? reactor->write\_handle[fdtype] : reactor->handle[SW\_FD\_WRITE];

}

if (event\_type == SW\_EVENT\_ERROR)

{

//默认关闭回调函数SW\_FD\_CLOSE

return (reactor->error\_handle[fdtype] != NULL) ? reactor->error\_handle[fdtype] : reactor->handle[SW\_FD\_CLOSE];

}

return reactor->handle[fdtype];

源码解释：首先判定事件类型是否为写事件，如果是，判定参数fdtype指定的回调是否存在，如果不存在，默认返回SW\_FD\_WRITE回调，否则返回fdtype对应的回调；然后判定事件类型是否为异常事件，如果是，判定参数fdtype指定的回调是否存在，如果不存在，默认返回SW\_FD\_CLOSE回调，否则返回fdtype对应的回调。最后，如果事件类型为其他类型，则直接返回fdtype对应的回调。

swReactor\_setHandle函数的核心代码如下：

int fdtype = swReactor\_fdtype(\_fdtype);

if (fdtype >= SW\_MAX\_FDTYPE)

{

swWarn("fdtype > SW\_MAX\_FDTYPE[%d]", SW\_MAX\_FDTYPE);

return SW\_ERR;

}

else

{

if (swReactor\_event\_read(\_fdtype))

{

reactor->handle[fdtype] = handle;

}

else if (swReactor\_event\_write(\_fdtype))

{

reactor->write\_handle[fdtype] = handle;

}

else if (swReactor\_event\_error(\_fdtype))

{

reactor->error\_handle[fdtype] = handle;

}

else

{

swWarn("unknow fdtype");

return SW\_ERR;

}

}

源码解释：调用swReactor\_fdtype函数去掉\_fdtype参数中的SW\_EVENTS类型变量，获取原始的swFd\_type类型变量fdtype。如果fdtype超过了swoole规定的范围，则返回SW\_ERR；否则，使用swReactor\_event\_\*系列函数判定\_fdtype的实际类型，根据不同的类型将回调函数存入reactor中对应的回调函数数组中。

swReactor\_receive函数只是简单调用swRead方法从event的fd中读取了数据，不再赘述。

二．ReactorEpoll

首先回顾一下epoll的相关知识（在群里很多用PHP做开发的小伙伴似乎根本不了解什么是epoll什么是异步I/O……）epoll是Linux内核提供的一个多路复用I/O模型，它提供和poll函数一样的功能：监控多个文件描述符是否处于I/O就绪状态（可读、可写）。这就是异步最核心的表现：程序不是主动等待一个描述符可以操作，而是当描述符可操作时由系统提醒程序可以操作了，程序在被提醒前可以去做其他的事情（这里的程序、描述符、系统可以更换为其他东西）

Linux提供了三个主要的系统调用：epoll\_create，epoll\_ctl，epoll\_wait。epoll\_create用于创建一个epoll实例并返回这个实例的文件描述符。epoll\_ctl用于将一个需要监控的文件描述符在epoll中注册对应的监听事件，该函数也可用于更改一个已注册描述符的监听事件。epoll\_wait函数用于等待监听的描述符的I/O事件，如果所有描述符都没有就绪，该函数会阻塞直到有至少一个描述符进入就绪状态。（该段描述翻译自 Linux 命令：man epoll ）

上周参与腾讯面试时就被问到了这样的问题：请说明一下epoll函数的水平触发（Level-triggered）和边缘触发（edge-triggered）两种模式的区别。结果我逗比的没答出来……在此重新复习一下这个知识……水平触发和边缘触发是epoll的两种模式，它们的区别在于：水平触发模式下，当一个fd就绪之后，如果没有对该fd进行操作，则系统会继续发出就绪通知直到该fd被操作；边缘触发模式下，当一个fd就绪后，系统仅会发出一次就绪通知。

（相关链接：<http://baike.baidu.com/view/1385104.htm> <http://yaocoder.blog.51cto.com/2668309/888374> ）

Swoole中，所有swReactorEpoll的相关定义都在ReactorEpoll.c中实现。首先说明两个epoll相关的宏：EPOLLRDHUP和EPOLLONESHOT。

EPOLLRDHUP代表的意义是对端断开连接，这个宏是用于弥补epoll在处理对端断开连接时可能会出现的一处Bug。

EPOLLONESHOT用于标记epoll对于每个socket仅监听一次事件，如果需要再次监听这个socket，需要再次将该socket加入epoll的监听队列中。

ReactorEpoll首先声明了一个结构体swFd用于封装一个描述符类型，其声明如下：

#pragma pack(4)

typedef struct \_swFd

{

uint32\_t fd;

uint32\_t fdtype;

} swFd;

#pragma pack()

其中, #pragma pack(4)的含义是指定结构体内的成员变量按照4字节对齐（关于字节对齐请参考<http://baike.baidu.com/view/2317161.htm>

<http://www.cppblog.com/tauruser/archive/2007/02/28/19049.html> ）

该结构体存放两个变量，一个变量为文件描述符，另一个变量为描述符类型。

同样的，Swoole也封装了一个结构体swReactorEpoll用于存放epoll的描述符以及监听的事件列表。该结构体的声明如下：

struct swReactorEpoll\_s

{

int epfd;

struct epoll\_event \*events;

};

创建一个ReactorEpoll的函数声明在swoole.h文件中的852行，其声明如下：

int swReactorPoll\_create(swReactor \*reactor, int max\_event\_num);

该函数的核心源码如下：

swReactorEpoll \*reactor\_object = sw\_malloc(sizeof(swReactorEpoll));

if (reactor\_object == NULL)

{

swWarn("malloc[0] failed.");

return SW\_ERR;

}

bzero(reactor\_object, sizeof(swReactorEpoll));

reactor->object = reactor\_object;

reactor->max\_event\_num = max\_event\_num;

reactor\_object->events = sw\_calloc(max\_event\_num, sizeof(struct epoll\_event));

if (reactor\_object->events == NULL)

{

swWarn("malloc[1] failed.");

return SW\_ERR;

}

//epoll create

reactor\_object->epfd = epoll\_create(512);

if (reactor\_object->epfd < 0)

{

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

return SW\_ERR;

}

源码解释：申请一个swReactorEpoll结构体并初始化。设置reactor的object和max\_event\_num参数。调用epoll\_create函数创建一个epoll实例，参数512指定最大的监听fd数量。

swReactorEpoll共有5个操作函数，其声明如下：

static int swReactorEpoll\_add(swReactor \*reactor, int fd, int fdtype);

static int swReactorEpoll\_set(swReactor \*reactor, int fd, int fdtype);

static int swReactorEpoll\_del(swReactor \*reactor, int fd);

static int swReactorEpoll\_wait(swReactor \*reactor, struct timeval \*timeo);

static void swReactorEpoll\_free(swReactor \*reactor);

这5个函数基于epoll函数家族以及close函数实现，用于对epoll实例的添加fd、设置fd监听事件、移除fd、等待fd事件以及释放epoll实例。同时Swoole还声明了一个内联函数swReactorEpoll\_event\_set用于将自定义的SW\_EVENTS类型转变为标准的epoll事件类型（EPOLLET、EPOLLIN、EPOLLOUT、EPOLLRDHUP）。下面将一一分析这些函数。

1. swReactorEpoll\_add

核心源码：

swReactorEpoll \*object = reactor->object;

struct epoll\_event e;

swFd fd\_;

int ret;

bzero(&e, sizeof(struct epoll\_event));

fd\_.fd = fd;

fd\_.fdtype = swReactor\_fdtype(fdtype);

e.events = swReactorEpoll\_event\_set(fdtype);

memcpy(&(e.data.u64), &fd\_, sizeof(fd\_));

ret = epoll\_ctl(object->epfd, EPOLL\_CTL\_ADD, fd, &e);

if (ret < 0)

{

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

return SW\_ERR;

}

swTraceLog(SW\_TRACE\_EVENT, "add event[reactor\_id=%d|fd=%d]", reactor->id, fd);

reactor->event\_num++;

源码解释：获取reactor中的swReactorEpoll结构体，创建一个epoll\_event结构体e和一个swFd结构体fd\_,初始化fd\_参数并将该对象存放到epoll\_event的data域中的u64变量中。调用epoll\_ctl添加对该fd的监听，并将reactor的event\_num计数加一。

1. swReactorEpoll\_set

核心代码与swReactorEpoll\_add基本一致，唯一不同在于epoll\_ctl函数的第二个参数由EPOLL\_CTL\_ADD变成EPOLL\_CTL\_MOD，代表设置fd的监听事件（而不是新增）。

1. swReactorEpoll\_del

核心源码：

swReactorEpoll \*object = reactor->object;

int ret;

if (fd <= 0)

{

return SW\_ERR;

}

ret = epoll\_ctl(object->epfd, EPOLL\_CTL\_DEL, fd, NULL);

if (ret < 0)

{

swWarn("epoll remove fd[=%d] failed. Error: %s[%d]", fd, strerror(errno), errno);

return SW\_ERR;

}

//close时会自动从epoll事件中移除

//swoole中未使用dup

ret = close(fd);

if (ret >= 0)

{

(reactor->event\_num <= 0) ? reactor->event\_num = 0 : reactor->event\_num--;

}

源码解释：获取reactor中的swReactorEpoll结构体，创建一个epoll\_event结构体e，设置e的data域的fd变量为指定需要删除的fd。调用epoll\_ctl并指定操作位EPOLL\_CTL\_DEL用监听队列中移除对应的监听，并close对应的fd。如果移除成功，更改reactor的event\_num计数。

1. swReactorEpoll\_wait

由于该函数较长且比较重要，在此将分段分析该函数。

swEvent ev;

swReactorEpoll \*object = reactor->object;

swReactor\_handle handle;

int i, n, ret, usec;

int reactor\_id = reactor->id;

int epoll\_fd = object->epfd;

int max\_event\_num = reactor->max\_event\_num;

struct epoll\_event \*events = object->events;

if (timeo == NULL)

{

usec = SW\_MAX\_UINT;

}

else

{

usec = timeo->tv\_sec \* 1000 + timeo->tv\_usec / 1000;

}

源码解释：该段源码声明了所需使用的全部临时变量。ev是相应事件数据的封装，是回调函数handle的第二个参数。object为swReactorEpoll结构体变量。n为每一次epoll\_wait响应后返回的当前处于就绪状态的fd的数量，usec为epoll\_wait的timeout超时时间，由swReactorEpoll\_wait函数的第二个参数struct timeval \*timeo指定。接下来的几个参数，reactor\_id为swReactor的标记，epoll\_fd为epoll实例的描述符，max\_event\_num为允许监听的最大事件数量，events用于存放epoll函数发现的处于就绪状态的事件。

while (SwooleG.running > 0)

源码解释：这是一个核心循环，之所以单独提出来是因为SwooleG变量非常重要。该变量声明在Server.c文件中的53行，并在swoole.h的1081行中通过extern关键字修饰使之可以被其他关联文件访问。该结构体中主要存放了整个swoole运行中需要的一些全局变量，在这里使用running变量用于标记swoole主循环是否正在执行。

n = epoll\_wait(epoll\_fd, events, max\_event\_num, usec);

if (n < 0)

{

if (swReactor\_error(reactor) < 0)

{

swWarn("[Reactor#%d] epoll\_wait failed. Error: %s[%d]", reactor\_id, strerror(errno), errno);

return SW\_ERR;

}

else

{

continue;

}

}

else if (n == 0)

{

if (reactor->onTimeout != NULL)

{

reactor->onTimeout(reactor);

}

continue;

}

源码解释：调用epoll\_wait函数获取已经处于就绪状态的fd的集合，该集合存放在events结构体数组中，其数目为返回值n。如果没有任何描述符处于就绪状态，该函数会阻塞直到有描述符就绪。如果在usec毫秒后仍没有描述符就绪，则返回0。

for (i = 0; i < n; i++)

{

ev.fd = events[i].data.u64;

ev.from\_id = reactor\_id;

ev.type = events[i].data.u64 >> 32;

//read

if (events[i].events & EPOLLIN)

{

//read

handle = swReactor\_getHandle(reactor, SW\_EVENT\_READ, ev.type);

ret = handle(reactor, &ev);

if (ret < 0)

{

swWarn("[Reactor#%d] epoll [EPOLLIN] handle failed. fd=%d. Error: %s[%d]", reactor\_id, ev.fd, strerror(errno), errno);

}

}

//write, ev.fd == 0, connection is closed.

if ((events[i].events & EPOLLOUT) && ev.fd > 0)

{

handle = swReactor\_getHandle(reactor, SW\_EVENT\_WRITE, ev.type);

ret = handle(reactor, &ev);

if (ret < 0)

{

swWarn("[Reactor#%d] epoll [EPOLLOUT] handle failed. fd=%d. Error: %s[%d]", reactor\_id, ev.fd, strerror(errno), errno);

}

}

//error

#ifndef NO\_EPOLLRDHUP

if ((events[i].events & (EPOLLRDHUP | EPOLLERR | EPOLLHUP)) && ev.fd > 0)

#else

if ((events[i].events & (EPOLLERR | EPOLLHUP)) && ev.fd > 0)

#endif

{

handle = swReactor\_getHandle(reactor, SW\_EVENT\_ERROR, ev.type);

ret = handle(reactor, &ev);

if (ret < 0)

{

swWarn("[Reactor#%d] epoll [EPOLLERR] handle failed. fd=%d. Error: %s[%d]", reactor\_id, ev.fd, strerror(errno), errno);

}

}

}

源码解释：这是核心的事件处理逻辑了。循环遍历n个待处理事件，前三行设置swEvent的对应参数（这里需要注意，data域中的u64变量是一个uint64\_t类型，该变量被写入了一个swFd结构体，低位32位存放的是uint32\_t类型的fd，高位32位存放的是uint32\_t类型的fdtype，该fdtype的取值为枚举类型swFd\_type），然后根据events的不同类型进入不同的处理逻辑，如下：

1. EPOLLIN 读事件，根据swEvent中的type类型（fdtype）获取对应的读操作的回调函数，通过该回调将swEvent类型发出。
2. EPOLLOUT 写事件，根据swEvent中的type类型（fdtype）获取对应的写操作的回调函数，通过该回调将swEvent类型发出。
3. 异常事件，根据swEvent中的type类型（fdtype）获取对应的异常操作的回调函数，通过该回调将swEvent类型发出。
4. swReactorEpoll\_free

调用close函数关闭epoll实例的描述符，并释放申请的内存空间

至此，swReactorEpoll分析已经结束。下一章将分析剩下的三种类型poll，select和kqueue。