## Swoole版本：1.7.5-stable

Factory这个命名让我一度认为这是一个工厂模型……这个工厂实际上并不负责生产实例，而是根据类型的不同执行两项任务：Factory实现的功能是一个任务中心，一个task请求进入Factory，会进过dispatch分配、onTask处理、onFinish交付结果一系列流程；FactoryProcess用于管理manager和worker进程，也有对单独的writer线程的管理。

（PS：Swoole源码中有FactoryThread模块，该模块是一个多线程模型，根据开发者Rango韩少的解释，因为PHP不支持多线程，所以无法使用这个模块，因此该模块被废弃了。而实际上，FactoryThread比FactoryProcess要更简洁……）

Factory模块的相关声明在Server.h头文件中。首先是一个在Factory模块中被用到的结构体swDispatchData，该结构体声明在Server.h的145 – 149 行，声明如下：

typedef struct

{

long target\_worker\_id;

swEventData data;

} swDispatchData;

swDispatchData存放了一个目标worker进程的id和一条数据，该结构体用于传递数据给task进程进行处理。

Swoole中用swFactory结构体封装了Factory模块的相关操作，其声明在Server.h文件中的151 – 168 行，其声明如下：

struct \_swFactory

{

void \*object;

void \*ptr; //server object

int last\_from\_id;

swReactor \*reactor; //reserve for reactor

int (\*start)(struct \_swFactory \*);

int (\*shutdown)(struct \_swFactory \*);

int (\*dispatch)(struct \_swFactory \*, swDispatchData \*);

int (\*finish)(struct \_swFactory \*, swSendData \*);

int (\*notify)(struct \_swFactory \*, swDataHead \*); //send a event notify

int (\*end)(struct \_swFactory \*, swDataHead \*);

int (\*onTask)(struct \_swFactory \*, swEventData \*task); //worker function.get a task,goto to work

int (\*onFinish)(struct \_swFactory \*, swSendData \*result); //factory worker finish.callback

};

其中object用于存放一个具体的Factory类型（FactoryProcess or FactoryThread），last\_from\_id存放了最近一个通过该Factory发送消息的reactor的id。

swFactory的全部操作函数都声明在Server.h的176 – 183行，声明如下：

int swFactory\_create(swFactory \*factory);

int swFactory\_start(swFactory \*factory);

int swFactory\_shutdown(swFactory \*factory);

int swFactory\_dispatch(swFactory \*factory, swDispatchData \*req);

int swFactory\_finish(swFactory \*factory, swSendData \*\_send);

int swFactory\_notify(swFactory \*factory, swDataHead \*event);

int swFactory\_end(swFactory \*factory, swDataHead \*cev);

int swFactory\_check\_callback(swFactory \*factory);

这些函数在Factory.c中被定义。其中，start和shutdown函数简单返回SW\_OK，create函数仅将传入的factory中的各个函数赋值（其实onTask和onFinish还是外部赋值……），callback函数仅仅检查onTask和onFinish两个函数指针是否为空，剩下的函数都是根据传入的参数调用对应的PHP回调函数（dispatch调用onTask，notify调用onClose和onConnect，end调用onClose并调用swServer\_connection\_close函数（详细分析见附录）关闭对应的connect连接）

这里需要分析一下swFactory\_finish函数。swFactory\_finish是一个通道，它的作用是将task运行结束后的数据发送给对应的Reactor。其核心源码如下：

//unix dgram

if (resp->info.type == SW\_EVENT\_UNIX\_DGRAM)

{

socklen\_t len;

struct sockaddr\_un addr\_un;

int from\_sock = resp->info.from\_fd;

addr\_un.sun\_family = AF\_UNIX;

memcpy(addr\_un.sun\_path, resp->sun\_path, resp->sun\_path\_len);

len = sizeof(addr\_un);

ret = swSendto(from\_sock, resp->data, resp->info.len, 0, (struct sockaddr \*) &addr\_un, len);

goto finish;

}

//UDP pacakge

else if (resp->info.type == SW\_EVENT\_UDP || resp->info.type == SW\_EVENT\_UDP6)

{

ret = swServer\_udp\_send(serv, resp);

goto finish;

}

else

{

resp->length = resp->info.len;

ret = swReactorThread\_send(resp);

}

源码解释：这里首先判定swDataHead\* resp应答中的type类型，如果type类型是Unix sock的数据包类型，则调用swSendto函数（详细分析见附录）发送数据到指定的Reactor；如果type类型是UDP，则调用swServer\_udp\_send（详细分析见附录）发送数据；否则，调用swReactorThread\_send函数发送数据。

FactoryProcess模块

FactoryProcess模块是Swoole的进程管理模块，是Swoole另一个核心组件。通过该模块，Swoole能有效的调度和管理Master进程和多个Worker进程。

FactoryProcess的结构体swFactoryProcess声明在Server.h文件的170 – 174行，声明如下：

typedef struct \_swFactoryProcess

{

swPipe \*pipes;

int writer\_pti; //current writer id

} swFactoryProcess;

第一个变量pipes存放用于进程间通信的管道，从后续实现分析这个Pipe仅用于接收其他进程发来的消息；第二个参数定义了当前writer线程的id，这个writer线程用于进程发送数据。

swFactoryProcess的四个公共操作函数声明在Server.h文件中的185 – 188行，其声明如下：

int swFactoryProcess\_create(swFactory \*factory, int writer\_num, int worker\_num);

int swFactoryProcess\_start(swFactory \*factory);

int swFactoryProcess\_shutdown(swFactory \*factory);

int swFactoryProcess\_end(swFactory \*factory, swDataHead \*event);

这四个函数都在FactoryProcess.c文件中具体定义。其中，swFactoryProcess\_create函数用于创建一个swFactoryProcess实例，其核心代码如下：

swFactoryProcess \*object;

swServer \*serv = SwooleG.serv;

object = SwooleG.memory\_pool->alloc(SwooleG.memory\_pool, sizeof(swFactoryProcess));

if (object == NULL)

{

swWarn("[Master] malloc[object] failed");

return SW\_ERR;

}

serv->writer\_threads = SwooleG.memory\_pool->alloc(SwooleG.memory\_pool, serv->reactor\_num \* sizeof(swWorkerThread));

if (serv->writer\_threads == NULL)

{

swWarn("[Master] malloc[object->writers] failed");

return SW\_ERR;

}

object->writer\_pti = 0;

源码解释：从全局变量SwooleG的内存池中申请一个swFactoryProcess的内存，并同样从这个内存池中申请和reactor\_num等量的writer线程，并初始化swFactoryProcess的writer\_pti为0.

swFactoryProcess\_shutdown函数不是关闭一个swFactoryProcess，而是关闭SwooleG全局变量中的swServer实例。其核心源码如下：

swServer \*serv = SwooleG.serv;

int i;

//kill manager process

kill(SwooleGS->manager\_pid, SIGTERM);

//kill all child process

for (i = 0; i < serv->worker\_num; i++)

{

swTrace("[Main]kill worker processor");

kill(serv->workers[i].pid, SIGTERM);

}

if (serv->ipc\_mode == SW\_IPC\_MSGQUEUE)

{

serv->read\_queue.free(&serv->read\_queue);

serv->write\_queue.free(&serv->write\_queue);

}

//close pipes

return SW\_OK;

源码解释：杀死manager进程，遍历swServer的worker列表并分别杀死每一个worker进程。如果使用了消息队列模式，则还需将read队列和write队列释放。

swFactoryProcess\_start函数用于启动一个swFactoryProcess实例。这里需要注意，在该函数中涉及到一些没有分析过的函数和结构体，这些对象的实际分析将于后面的章节进行，在此我只简单说明这些函数的作用。swFactoryProcess\_start函数核心源码如下：

if (swFactory\_check\_callback(factory) < 0)

{

swWarn("swFactory\_check\_callback failed");

return SW\_ERR;

}

int i;

swServer \*serv = factory->ptr;

swWorker \*worker;

for (i = 0; i < serv->worker\_num; i++)

{

worker = swServer\_get\_worker(serv, i);

if (swWorker\_create(worker) < 0)

{

return SW\_ERR;

}

}

//必须先启动manager进程组，否则会带线程fork

if (swFactoryProcess\_manager\_start(factory) < 0)

{

swWarn("swFactoryProcess\_manager\_start fail");

return SW\_ERR;

}

if (serv->ipc\_mode == SW\_IPC\_MSGQUEUE)

{

swQueueMsg\_set\_blocking(&serv->read\_queue, 0);

//tcp & message queue require writer pthread

if (serv->have\_tcp\_sock == 1)

{

int ret = swFactoryProcess\_writer\_start(factory);

if (ret < 0)

{

return SW\_ERR;

}

}

}

//主进程需要设置为直写模式

factory->finish = swFactory\_finish;

源码解释：首先判断swFactory是否设置了相应的PHP回调函数（onTask，onFinish）。然后创建worker\_num个worker对象，随后调用swFactoryProcess\_manager\_start函数启动manager进程和对应的worker子进程。如果swServer设置了消息队列属性，则设置swServer的read队列为非阻塞，并启动writer线程。最后设置manager主进程的finish函数。

swFactoryProcess\_end函数用于关闭一个swFactoryProcess实例。其核心代码入下：

swServer \*serv = factory->ptr;

swSendData \_send;

bzero(&\_send, sizeof(\_send));

\_send.info.fd = event->fd;

/\*\*

\* length == 0, close the connection

\*/

\_send.info.len = 0;

/\*\*

\* passive or initiative

\*/

\_send.info.type = event->type;

swConnection \*conn = swServer\_connection\_get(serv, \_send.info.fd);

if (conn == NULL || conn->active == 0)

{

swWarn("can not close. Connection[%d] not found.", \_send.info.fd);

return SW\_ERR;

}

if (serv->onClose != NULL)

{

serv->onClose(serv, event->fd, conn->from\_id);

}

return swFactoryProcess\_finish(factory, &\_send);

源码解释：获取swServer对象，创建需要发送的swSendData并设置fd、长度（长度为0代表关闭连接）和类型。并获取swServer中fd对应的connection连接。随后调用onClose回调函数（如果有），并调用swFactoryProcess\_finish函数发送数据。

除了以上四个共有的操作函数外，swFactoryProcess还拥有三个不同的操作函数，声明如下：

static int swFactoryProcess\_notify(swFactory \*factory, swDataHead \*event);

static int swFactoryProcess\_dispatch(swFactory \*factory, swDispatchData \*buf);

static int swFactoryProcess\_finish(swFactory \*factory, swSendData \*data);

这三个函数是核心的进程通信函数，notify函数用于主进程通知worker进程，dispatch用于reactor向worker进程投递请求，finish用于worker将执行后的数据发送给client。

一个个来看，首先是swFactoryProcess\_notify函数，其核心代码如下：

int swFactoryProcess\_notify(swFactory \*factory, swDataHead \*ev)

{

memcpy(&sw\_notify\_data.\_send, ev, sizeof(swDataHead));

sw\_notify\_data.\_send.len = 0;

return factory->dispatch(factory, (swDispatchData \*) &sw\_notify\_data);

}

源码解释：通过factory的dispatch函数将notify\_data发送给worker进程。

其中sw\_notify\_data的声明如下：

static \_\_thread struct

{

long target\_worker\_id;

swDataHead \_send;

} sw\_notify\_data;

这个结构体是用来匹配swDispatchData结构体的，并没有特殊的意义。

swFactoryDespatch函数的调用来自于ReactorThread或者ReactorProcess（根据Rango的说法，所有线程相关的模块都已经停用了……），其核心源码如下：

uint32\_t schedule\_key;

uint32\_t send\_len = sizeof(task->data.info) + task->data.info.len;

uint16\_t target\_worker\_id;

swServer \*serv = SwooleG.serv;

if (task->target\_worker\_id < 0)

{

//udp use remote port

if (task->data.info.type == SW\_EVENT\_UDP || task->data.info.type == SW\_EVENT\_UDP6

|| task->data.info.type == SW\_EVENT\_UNIX\_DGRAM)

{

schedule\_key = task->data.info.from\_id;

}

else

{

schedule\_key = task->data.info.fd;

}

#ifndef SW\_USE\_RINGBUFFER

if (SwooleTG.factory\_lock\_target)

{

if (SwooleTG.factory\_target\_worker < 0)

{

target\_worker\_id = swServer\_worker\_schedule(serv, schedule\_key);

SwooleTG.factory\_target\_worker = target\_worker\_id;

}

else

{

target\_worker\_id = SwooleTG.factory\_target\_worker;

}

}

else

#endif

{

target\_worker\_id = swServer\_worker\_schedule(serv, schedule\_key);

}

}

else

{

target\_worker\_id = task->target\_worker\_id;

}

if (SwooleTG.type == SW\_THREAD\_REACTOR)

{

return swReactorThread\_send2worker((void \*) &(task->data), send\_len, target\_worker\_id);

}

else

{

swTrace("dispatch to worker#%d", target\_worker\_id);

return swServer\_send2worker\_blocking(serv, (void \*) &(task->data), send\_len, target\_worker\_id);

}

源码解释：首先判定task是否已经指定了worker，如果task的target\_worker\_id小于0（代表没有指定），则需要为其分配worker进程。首先根据task的data域中info的type字段判断task是属于哪个fd，这个fd用于在随后的调用中判断如何分配worker进程。之后，判断是否指定使用RingBuffer，判断SwooleTG（Swoole Thread Global，线程相关全局变量，以\_\_thread关键字修饰，每个线程拥有自己独立的一份，线程之间互不影响）是否指定了当前线程所有的worker\_id，如果没有指定，调用swServer\_worker\_schedule函数分配worker；否则直接使用指定的worker。如果以上条件都未能满足，则直接使用swServer\_worker\_schedule函数分配worker。如果task已经指定了worker，则默认使用该worker。随后判断SwooleTG的type字段，如果为SW\_THREAD\_REACTOR(线程Reacotr模式)，则调用swReactorThread\_send2worker函数（非阻塞）发送task请求；否则（type为SW\_PROCESS\_REACTOR模式），调用swServer\_send2worker\_blocking函数（阻塞）发送请求。

这里解释一下这两个函数的阻塞和非阻塞的原因。swReactorThread\_send2worker调用write函数如果失败，会将数据存放进pipe的缓存中，等待下一次发送；而swServer\_send2worker\_blocking调用write失败后，会重新发送或调用swSocket\_wait等待一段时间后再次发送。

接下来是swFactoryProcess\_finish函数，该函数用于将一个resp应答发送给task的发起者。由于该函数较长，在此分段进行分析：其源码如下：

//unix dgram

if (resp->info.type == SW\_EVENT\_UNIX\_DGRAM)

{

socklen\_t len;

struct sockaddr\_un addr\_un;

int from\_sock = resp->info.from\_fd;

addr\_un.sun\_family = AF\_UNIX;

memcpy(addr\_un.sun\_path, resp->sun\_path, resp->sun\_path\_len);

len = sizeof(addr\_un);

ret = swSendto(from\_sock, resp->data, resp->info.len, 0, (struct sockaddr \*) &addr\_un, len);

goto finish;

}

源码解释：如果判定resp的类型为unixsock的报文模式，则构建相应sockaddr\_un结构体并调用swSendto函数发送resp（swSendto函数分析见附录）。发送完成后，goto到finish标签继续运行。

//UDP pacakge

else if (resp->info.type == SW\_EVENT\_UDP || resp->info.type == SW\_EVENT\_UDP6)

{

ret = swServer\_udp\_send(serv, resp);

goto finish;

}

源码解释：如果是UDP模式，调用swServer\_udp\_send函数发送数据并goto到finish标签。

struct

{

long pti;

swEventData \_send;

} sdata;

//for message queue

sdata.pti = (SwooleWG.id % serv->writer\_num) + 1;

swConnection \*conn = swServer\_connection\_get(serv, fd);

if (conn == NULL || conn->active == 0)

{

swWarn("connection[%d] not found.", fd);

return SW\_ERR;

}

sdata.\_send.info.fd = fd;

sdata.\_send.info.type = resp->info.type;

swWorker \*worker = swServer\_get\_worker(serv, SwooleWG.id);

源码解释：构建了一个临时结构体用于匹配swQueue\_data结构体，并指定对应的writer线程id。随后获取fd对应的connection连接和SwooleWG（Worker进程的全局变量）中存放的当前进程对应的worker\_id。

if (resp->length > 0)

{

int64\_t wait\_reactor;

/\*\*

\* Storage is in use right now, wait notify.

\*/

if (worker->store.lock == 1)

{

worker->notify->read(worker->notify, &wait\_reactor, sizeof(wait\_reactor));

}

swPackage\_response response;

response.length = resp->length;

response.worker\_id = SwooleWG.id;

//swWarn("BigPackage, length=%d|worker\_id=%d", response.length, response.worker\_id);

sdata.\_send.info.from\_fd = SW\_RESPONSE\_BIG;

sdata.\_send.info.len = sizeof(response);

memcpy(sdata.\_send.data, &response, sizeof(response));

/\*\*

\* Lock the worker storage

\*/

worker->store.lock = 1;

memcpy(worker->store.ptr, resp->data, resp->length);

}

源码解释：如果resp的长度大于0，表示这是个较大的应答包（完全不理解为什么大于0就代表big response了……），需要使用共享内存。首先判定worker的store是否已经被占用，如果被占用，则通过notify管道等待唤醒。当store可用后，用swPackage\_response结构体封装数据信息并代替原有的data域，设置from\_id为SW\_RESPONSE\_BIG，并将data复制到worker的store共享内存中并上锁。

#if SW\_REACTOR\_SCHEDULE == 2

sdata.\_send.info.from\_id = fd % serv->reactor\_num;

#else

sdata.\_send.info.from\_id = conn->from\_id;

#endif

源码解释：根据Reactor的Schedule模型确定from\_id， 2代表的是取模。

for (count = 0; count < SW\_WORKER\_SENDTO\_COUNT; count++)

{

if (serv->ipc\_mode == SW\_IPC\_MSGQUEUE)

{

ret = serv->write\_queue.in(&serv->write\_queue, (swQueue\_data \*) &sdata, sendn);

}

else

{

int master\_pipe = swWorker\_get\_write\_pipe(serv, fd);

//swWarn("send to reactor. fd=%d|pipe=%d|reactor\_id=%d|reactor\_pipe\_num=%d", fd, master\_pipe, conn->from\_id, serv->reactor\_pipe\_num);

ret = write(master\_pipe, &sdata.\_send, sendn);

#ifdef SW\_WORKER\_WAIT\_PIPE

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

{

/\*\*

\* Wait pipe can be written.

\*/

if (swSocket\_wait(master\_pipe, SW\_WORKER\_WAIT\_TIMEOUT, SW\_EVENT\_WRITE) == SW\_OK)

{

continue;

}

else

{

goto finish;

}

}

#endif

}

//swTraceLog("wt\_queue->in: fd=%d|from\_id=%d|data=%s|ret=%d|errno=%d", sdata.\_send.info.fd, sdata.\_send.info.from\_id, sdata.\_send.data, ret, errno);

if (ret >= 0)

{

break;

}

else if (errno == EINTR)

{

continue;

}

else if (errno == EAGAIN)

{

swYield();

}

else

{

break;

}

}

源码解释：循环写出数据直到写出成功，最大重写次数为SW\_WORKER\_SENDTO\_COUNT（32）次。每次写出时，先判断是否为消息队列模式，如果是，将数据写入消息队列；如果不是，获取worker的write\_pipe，并通过该管道将数据发送给master进程。

对于FactoryProcess的基本分析到这里就结束了，下一章将分析FactoryProcess.c文件中剩下的操作函数。