## Swoole版本：1.7.4-stable

接下来是RingBuffer。这相当于一个循环数组，每一次申请的一块内存在该数组中占据一个位置，这些内存块是可以不等长的，因此每个内存块需要有一个记录其长度的变量。这里贴出swRingBuffer\_head结构体的代码：

typedef struct \_swRingBuffer\_item

{

volatile uint32\_t lock;

volatile uint32\_t length;

} swRingBuffer\_head;

每一个结构体代表一个RingBuffer中的内存块，其中lock变量标记该内存块是否被占用，length变量标记该内存块的长度。

接着是swRingBuffer结构体的声明：

typedef struct \_swRingBuffer

{

uint8\_t shared; // 可共享

size\_t size; // 内存池大小

volatile off\_t alloc\_offset; // 分配内存的起始长度

volatile off\_t collect\_offset; // 可用内存的终止长度

volatile uint32\_t free\_n; // 有多少个内存块待回收

void \*memory; // 内存池的起始地址

} swRingBuffer;

每一个结构体代表一个RingBuffer内存池。这里先要说明一下RingBuffer的三个变量：alloc\_offset,collect\_offset,free\_n。alloc\_offset变量是分配内存的起始地址，代表的是RingBuffer现有的可用空间的起始地址；collect\_offset变量是分配内存的终止地址，代表的是RingBuffer现有的可用空间的结束地址。为了方便理解，大家可以想象一下循环队列，alloc\_offset和collect\_offset就是标记队头和队尾的标记，每一次分配内存就相当于入队，每一次释放内存就相当于出队。而free\_n变量是用于标记当前还有多少个已释放的内存块待回收。这是因为RingBuffer采用的是连续分配，可能会存在一些已经被free的内存块夹在两个没有free的内存块中间，没有被立即回收，就需要一个变量去通知内存池回收这些内存。。

RingBuffer的创建函数为swRingBuffer\_new，其声明在swoole.h文件的514 – 517行。入下：

/\*\*

\* RingBuffer, In order for malloc / free

\*/

swMemoryPool \*swRingBuffer\_new(size\_t size, uint8\_t shared);

该函数的具体定义在RingBuffer.c中，创建过程与FixedPool基本类似，就不再额外分析，大家自行阅读源码即可。

和FixedPool类似，RingBuffer也拥有4个函数用于操作内存池，其函数声明如下：

static void swRingBuffer\_destory(swMemoryPool \*pool);

static sw\_inline void swRingBuffer\_collect(swRingBuffer \*object);

static void\* swRingBuffer\_alloc(swMemoryPool \*pool, uint32\_t size);

static void swRingBuffer\_free(swMemoryPool \*pool, void \*ptr);

其中alloc、destroy、free三个函数的功能很明确，swRingBuffer\_collect函数用于回收已经不被占用的内存。这里着重分析alloc函数和collect函数。

首先是collect函数。在发现内存池剩余不足或分配内存结束后，RingBuffer都会调用collect函数去回收已经没有被占用的内存。其核心代码如下：

for(i = 0; i<SW\_RINGBUFFER\_COLLECT\_N; i++)

{

item = (swRingBuffer\_head \*) (object->memory + object->collect\_offset);

swTraceLog(SW\_TRACE\_MEMORY, "collect\_offset=%d, item\_length=%d, lock=%d", object->collect\_offset, item->length, item->lock);

//can collect

if (item->lock == 0)

{

object->collect\_offset += (sizeof(swRingBuffer\_head) + item->length);

if (object->free\_n > 0)

{

object->free\_n --;

}

if (object->collect\_offset >= object->size)

{

object->collect\_offset = 0;

}

}

else

{

break;

}

}

源码解释：每一次循环，都会获取当前collect\_offset指向的地址代表的内存块，并获取其swRingBuffer\_head结构，如果该内存块已经被free，则将collect\_offset标记后移该内存块的长度，回收该内存。如果发现collect\_offset超出了内存池大小，则将collect\_offset移到内存池头部。

alloc函数太长，在此不贴出源码，只写出伪代码供分析：

start\_alloc:

if( alloc\_offset < collect\_offset ) // 起始地址在终止地址左侧

{

head\_alloc:

计算剩余内存大小

if( 内存足够 )

goto do\_alloc

else if( 内存不足且已经回收过内存)

return NULL；

else

{

try\_collect = 1;

调用collect

goto start\_alloc

}

}

else // 起始地址在终止地址右侧（终止地址被移动到了首部）

{

计算从alloc\_offset到内存池尾部的剩余内存大小

if( 内存足够 )

goto do\_alloc

else

{

标记尾部剩余内存为可回收状态

将alloc\_offset移动到首部

goto head\_alloc

}

}

do\_alloc:

实际分配内存块并设置属性，移动alloc\_offset标记

如果free\_n大于0，则回收内存。

最后是MemoryGlobal。MemoryGlobal是一个比较特殊的内存池。说实话我没有看懂它的作用，所以我决定先暂时跳过MemoryGlobal，等了解其具体使用场景时再来分析这一块。