## Swoole版本：1.7.4-stable

Swoole中为了更好的进行内存管理，减少频繁分配释放内存空间造成的损耗和内存碎片，Rango设计并实现了三种不同功能的MemoryPool：FixedPool，RingBuffer和MemoryGlobal。

Rango声明了一个swMemoryPool结构体来表示一个内存池，该结构体在swoole.h头文件中 **501-507行**声明，结构如下：

typedef struct \_swMemoryPool

{

void \*object;

void\* (\*alloc)(struct \_swMemoryPool \*pool, uint32\_t size);

void (\*free)(struct \_swMemoryPool \*pool, void \*ptr);

void (\*destroy)(struct \_swMemoryPool \*pool);

} swMemoryPool;

（PS：虽然原来也知道结构体中可以通过存放函数指针模拟一个类，但是现在才学到可以通过传入一个指针参数来模拟this指针，使结构体更像一个类）

这里看到，首先是一个void\*型指针，用于标记内存池的首地址。另外三个函数大家都不会陌生，分别用于从内存池中拿到一块内存、释放一块内存、销毁整个内存池。

基类介绍完了，下面来看具体的子类实现。

首先是FixedPool。FixedPool是随机分配内存池(random alloc/free)，将一整块内存空间切分成**等大小**的一个个小块，每次分配其中的一个小块作为要使用的内存，这些小块以**链表**的形式存储。

FixedPool的全部定义声明均在src/memory/FixedPool.c文件内。Rango声明了两个结构体swFixedPool\_slice和swFixedPool来实现管理内存池。

swFixedPool\_slice为内存池中每个小块的结构声明，其定义如下：

typedef struct \_swFixedPool\_slice

{

uint8\_t lock;

struct \_swFixedPool\_slice \*next;

struct \_swFixedPool\_slice \*pre;

char data[0];

} swFixedPool\_slice;

可以看出，每个slice为一个链表节点（双向链表），unit8\_t lock用于标记该节点是否被占用（0代表空闲，1代表已占用），next和pre为该节点的后继、前驱指针，char data[0]为内存空间指针（关于0长度数组的解释，请看<http://blog.csdn.net/liuaigui/article/details/3680404>）

swFixedPool为内存池实体，包括了内存池相关信息。定义如下：

typedef struct \_swFixedPool

{

void \*memory; // 内存指针，指向一片内存空间

size\_t size; // 内存空间大小

swFixedPool\_slice \*head; // 链表头部节点

swFixedPool\_slice \*tail; // 链表尾部节点，两个指针用于快速访问和移动节点

/\*\*

\* total memory size，节点数目

\*/

uint32\_t slice\_num;

/\*\*

\* memory usage ，已经使用的节点

\*/

uint32\_t slice\_use;

/\*\*

\* Fixed slice size

\*/

uint32\_t slice\_size; 节点大小

/\*\*

\* use shared memory

\*/

uint8\_t shared; 是否共享内存

} swFixedPool;

（参数太多就不文字叙述，直接在变量名后以注释格式解释）

这里先看swFixedPool\_new的具体定义。其声明如下：

/\*\*

\* FixedPool, random alloc/free fixed size memory

\* Location: swoole.h **512L**

\*/

swMemoryPool \* swFixedPool\_new(uint32\_t size, uint32\_t trunk\_size, uint8\_t shared);

第一个参数size为每一个小块的大小，第二个参数trunk\_size为小块的总数，第三个参数shared代表该内存池是否为共享内存，返回的是被创建的内存池指针。下面贴出代码：

size\_t size = slice\_size \* slice\_num;

size\_t alloc\_size = size + sizeof(swFixedPool) + sizeof(swMemoryPool);

void \*memory = (shared == 1) ? sw\_shm\_malloc(alloc\_size) : sw\_malloc(alloc\_size);

swFixedPool \*object = memory;

memory += sizeof(swFixedPool);

bzero(object, sizeof(swFixedPool));

object->shared = shared;

object->slice\_num = slice\_num;

object->slice\_size = slice\_size;

object->size = size;

swMemoryPool \*pool = memory;

memory += sizeof(swMemoryPool);

pool->object = object;

pool->alloc = swFixedPool\_alloc; // 指定三个操作函数

pool->free = swFixedPool\_free;

pool->destroy = swFixedPool\_destroy;

object->memory = memory;

swFixedPool\_init(object);

前两行是计算出内存池实际大小：slice大小\*slice数量+MemoryPool头部大小+FixedPool头部大小。接下来根据是否是共享内存来判定使用哪种分配内存的方法（其中sw\_shm\_malloc方法将在上一章补充）。接下来依次填充swFixedPool、swMemoryPool的相关属性，实际内存空间起始地址存放于swFixedPool的memory中。接下来调用swFixedPool\_init方法初始化内存池。

FixedPool同时拥有四个操作函数分别用于初始化、分配空间、释放空间、销毁内存池。定义如下：

static void swFixedPool\_init(swFixedPool \*object);

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

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

static void swFixedPool\_destroy(swMemoryPool \*pool);

首先是swFixedPool\_init函数。该函数用于初始化一个内存池结构体，其核心代码如下：

void \*cur = object->memory;

void \*max = object->memory + object->size;

do

{

slice = (swFixedPool\_slice \*) cur;

bzero(slice, sizeof(swFixedPool\_slice));

if (object->head != NULL)

{

object->head->pre = slice;

slice->next = object->head;

}

else

{

object->tail = slice;

}

object->head = slice;

cur += (sizeof(swFixedPool\_slice) + object->slice\_size);

slice->pre = (swFixedPool\_slice \*) cur;

} while (cur < max);

源码解释：从内存空间的首部开始，每次初始化一个slice大小的空间，并插入到链表的头部。至于为什么使用链表这个结构会稍后说明。

内存分配好了，如何使用呢？这里再看swFixedPool\_alloc函数。这里要注意，因为FixedPool的内存块大小是固定的，所以函数中第二个参数size只是为了符合swMemoryPool中的声明，不具有实际作用，可以随便填一个数。其核心代码如下：

slice = object->head;

if (slice->lock == 0)

{

slice->lock = 1;

/\*\*

\* move next slice to head (idle list)

\*/

object->head = slice->next;

slice->next->pre = NULL;

/\*

\* move this slice to tail (busy list)

\*/

object->tail->next = slice;

slice->next = NULL;

slice->pre = object->tail;

object->tail = slice;

return slice->data;

}

else

{

return NULL;

}

源码解释：首先获取内存池链表首部的节点，并判断该节点是否被占用，如果被占用，说明内存池已满，返回null（因为所有被占用的节点都会被放到尾部）；如果未被占用，则将该节点的下一个节点移到首部，并将该节点移动到尾部，标记该节点为占用状态，返回该节点的数据域。

当一个内存块用完，需要释放内存，则调用swFixedPool\_free方法。该函数第二个参数为需要释放的数据域。其核心代码如下：

slice = ptr - sizeof(swFixedPool\_slice);

slice->lock = 0;

//list head, AB

if (slice->pre == NULL)

{

return;

}

//list tail, DE

if (slice->next == NULL)

{

slice->pre->next = NULL;

}

//middle BCD

else

{

slice->pre->next = slice->next;

slice->next->pre = slice->pre;

}

slice->pre = NULL;

slice->next = object->head;

object->head->pre = slice;

object->head = slice;

源码解释：首先通过移动ptr指针获得slice对象，并将占用标记lock置为0。如果该节点为头节点，则直接返回。如果不是头节点，则将该节点移动到链表头部。

swFixedPool\_destroy为直接释放整片内存池，在此不再详解。

（PS：目前先放出FixedPool的源码分析，今晚再放出RingBuffer和MmemoyGlobal的分析。）

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。