## Swoole版本：1.7.4-stable

二．Mutex互斥锁

接下来是Mutex（互斥锁）。Swoole的Mutex实现是基于pthread\_mutex\*方法族实现的，Rango用一个swMutex结构体封装了mutex的两个属性，结构体定义如下：

//互斥锁

typedef struct \_swMutex

{

pthread\_mutex\_t \_lock; // 互斥锁对象

pthread\_mutexattr\_t attr; // 互斥锁的属性

} swMutex;

创建一个互斥锁的函数声明在swoole.h文件的 536 行，声明如下：

int swMutex\_create(swLock \*lock, int use\_in\_process);

其中lock为锁对象，use\_in\_process用于标记该锁是否用于进程之间。该函数的具体定义在Mutex.c文件中，其核心代码如下：

lock->type = SW\_MUTEX;

pthread\_mutexattr\_init(&lock->object.mutex.attr);

if (use\_in\_process == 1)

{

pthread\_mutexattr\_setpshared(&lock->object.mutex.attr, PTHREAD\_PROCESS\_SHARED);

}

if ((ret = pthread\_mutex\_init(&lock->object.mutex.\_lock, &lock->object.mutex.attr)) < 0)

{

return SW\_ERR;

}

源码解释：设置锁的类型为SW\_MUTEX，并调用pthread\_mutexattr\_init函数初始化mutex的属性结构体。如果该进程用于进程之间，则调用pthread\_mutexattr\_setpshared函数设置该锁的共享属性为PTHREAD\_PROCESS\_SHARED（进程间共享）。接着调用pthread\_mutex\_init函数初始化互斥锁。

这里要重点分析三个标红的函数。这三个函数都属于pthread\_mutex\*方法族，用于操作mutex以及其attr属性。

1. pthread\_mutexattr\_init方法用于初始化一个pthread\_mutexattr\_t结构体，并默认设置其pshared属性为PTHREAD\_PROCESS\_PRIVATE（表示可以在进程内使用该互斥锁）。
2. pthread\_mutexattr\_setpshared方法用来设置互斥锁变量的作用域。PTHREAD\_PROCESS\_SHARED表示该互斥锁可被多个进程共享使用（前提是该互斥锁是在共享内存中创建）。PTHREAD\_PROCESS\_PRIVATE表示该互斥锁仅能被那些由同一个进程创建的线程处理。
3. pthread\_mutex\_init方法以动态方式创建互斥锁，并通过参数attr指定互斥锁属性。如果参数attr为空，则默认创建快速互斥锁。

swMutex的其他操作函数swMutex\_lock、swMutex\_unlock、swMutex\_trylock、free分别对应pthread\_mutex\_lock、pthread\_mutex\_unlock、pthread\_mutex\_trylock、pthread\_mutex\_destroy，用于加锁、解锁、尝试加锁、销毁swMutex。

三．RWLock读写锁

读写锁的内容其实与互斥锁相差不大，区别仅在于读写锁有读锁、写锁两个区别，同时底层调用的是pthread\_rwlock\*系列函数。读写锁swRWLock结构体在swoole.c文件的 398 – 404 行声明，如下：

//读写锁

typedef struct \_swRWLock

{

pthread\_rwlock\_t \_lock; // 读写锁对象

pthread\_rwlockattr\_t attr; // 读写锁属性

} swRWLock;

创建一个读写锁的函数声明在swoole.h文件的 534 行，声明如下：

int swRWLock\_create(swLock \*lock, int use\_in\_process);

其中lock为锁对象，use\_in\_process用于标记该锁是否用于进程之间。该函数的具体定义在RWLock.c文件中，其核心代码如下：

lock->type = SW\_RWLOCK;

if(use\_in\_process == 1)

{

pthread\_rwlockattr\_setpshared(&lock->object.rwlock.attr, PTHREAD\_PROCESS\_SHARED);

}

if((ret = pthread\_rwlock\_init(&lock->object.rwlock.\_lock, &lock->object.rwlock.attr)) < 0)

{

return SW\_ERR;

}

源码解释：设置锁类型为SW\_RWLOCK，如果该锁用于进程之间，则调用pthread\_rwlockattr\_setpshared函数设置该锁的共享属性为PTHREAD\_PROCESS\_SHARED（进程间共享）。接着调用pthread\_rwlock\_init初始化该读写锁。

这里重点分析两个标红函数。这两个函数属于pthread\_rwlock\*方法族，用于操作rwlock以及其attr属性。

1. pthread\_rwlockattr\_setpshared用于设置一个pthread\_rwlockattr\_t的属性，和mutex一样有PTHREAD\_PROCESS\_SHARED和PTHREAD\_PROCESS\_PRIVATE两个值。
2. pthread\_rwlock\_init用于初始化一个pthread\_rwlock\_t结构体，创建一个读写锁，并通过attr设置其属性。

（实际上rwlock也有pthread\_rwlockattr\_init方法，不知道为什么这里没有调用。所有pthread\_rwlock\*相关的函数请参考<http://blog.163.com/software_ark/blog/static/175614594201181665330631/>）

剩下的就是通过调用pthread\_rwlock的相关操作函数如pthread\_rwlock\_rdlock、pthread\_rwlock\_wrlock来创建读锁、写锁以及解锁。大家可以直接看RWLock.c文件中的定义，在此为节约篇幅不再叙述。

四．SpinLock自旋锁

首先需要说明一下什么是自旋锁。Spinlock本质上也是一种互斥锁，它和mutex的区别在于Spinlock是通过busy\_wait\_loop方式来获得锁，不会使线程状态发生切换（用户态->内核态）,因此减少了系统调用，执行速度快。但缺点也是有的，Spinlock会一直占用cpu，导致cpu busy飙高，因此要谨慎使用Spinlock。

SpinLock自旋锁的特殊性在于其只有一个pthread\_spinlock\_t锁对象，没有对应的属性结构体。其swSpinLock结构体在swoole.c文件中 406 – 412 行声明，如下：

//自旋锁

#ifdef HAVE\_SPINLOCK

typedef struct \_swSpinLock

{

pthread\_spinlock\_t lock\_t; // 自旋锁对象

} swSpinLock;

#endif

创建一个自旋锁的函数声明在swoole.h文件的 534 行，声明如下：

int swSpinLock\_create(swLock \*object, int spin);

其中lock为锁对象，spin（为何不继续使用use\_in\_process……）用于标记该锁是否用于进程之间。该函数的具体定义在SpinLock.c文件中，其核心代码如下：

lock->type = SW\_SPINLOCK;

if ((ret = pthread\_spin\_init(&lock->object.spinlock.lock\_t, use\_in\_process)) < 0)

{

return -1;

}

源码解释：设置锁类型为SW\_SPINLOCK，调用pthread\_spin\_init初始化一个Spinlock对象。

1. pthread\_spin\_init用于初始化一个pthread\_spinlock\_t对象，第二个参数用于指定该spinlock是否可被进程间共享。

其他操作均为直接调用pthread\_spin\*方法族的相关函数，具体内容请看SpinLock.c文件。

五．原子锁

首先我说明一下什么是原子操作。所谓原子操作，就是该操作绝不会在执行完毕前被任何其他任务或事件打断，也就说，它的最小的执行单位，不可能有比它更小的执行单位。通常情况下，原子操作都是用于资源计数。

原子锁就是用来保证操作的原子性。Swoole中自定义了一个原子类型atomic\_t，其声明在atomic.h文件中的14行，这个类型在不同位数的操作系统里有不同的长度。同时，swoole使用了gcc提供的\_\_sync\_\*系列的build-in函数来提供加减和逻辑运算的原子操作。这些函数的声明在atomic.h文件的16 – 20行，如下：

#define sw\_atomic\_cmp\_set(lock, old, set) \_\_sync\_bool\_compare\_and\_swap(lock, old, set)

#define sw\_atomic\_fetch\_add(value, add) \_\_sync\_fetch\_and\_add(value, add)

#define sw\_atomic\_fetch\_sub(value, sub) \_\_sync\_fetch\_and\_sub(value, sub)

#define sw\_atomic\_memory\_barrier() \_\_sync\_synchronize()

#define sw\_atomic\_cpu\_pause() \_\_asm\_\_ ("pause")

这里依次介绍5个函数的作用：

1. sw\_atomic\_cmp\_set(lock, old, set) 如果lock的值等于old的值，将set的值写入lock。如果相等并写入成功则返回true
2. sw\_atomic\_fetch\_add(value, add) 将value的值增加add，返回增加前的值
3. sw\_atomic\_fetch\_sub(value, sub) 将value的值减去sub，返回减去前的值
4. sw\_atomic\_memory\_barrier() 发出一个full barrier。保证指令执行的顺序合理
5. sw\_atomic\_cpu\_pause() 这个函数调用的是\_\_asm\_\_(“pause”)，通过一番查询，这个调用能让cpu等待一段时间，这个时间由处理器定义。

Swoole另外声明了一个结构体swAtomicLock来封装原子锁，该结构体在swoole.h文件中 414 – 419 行声明，如下：

//原子锁Lock-Free

typedef struct \_swAtomicLock

{

atomic\_t lock\_t;

uint32\_t spin;

} swAtomicLock;

其中lock\_t为原子锁对象，spin为一个原子锁可以等待的次数（为2的n次幂）。

原子锁的全部操作函数声明在swoole.h文件的541 – 544行，如下：

int swAtomicLock\_create(swLock \*object, int spin);

sw\_inline int swAtomicLock\_lock(swLock \*lock);

sw\_inline int swAtomicLock\_unlock(swLock \*lock);

sw\_inline int swAtomicLock\_trylock(swLock \*lock);

这里着重介绍swAtomicLock\_lock方法，其核心代码如下：

atomic\_t \*atomic = &lock->object.atomlock.lock\_t;

uint32\_t i, n;

while (1)

{

if (\*atomic == 0 && sw\_atomic\_cmp\_set(atomic, 0, 1))

{

return SW\_OK;

}

if (SW\_CPU\_NUM > 1)

{

for (n = 1; n < lock->object.atomlock.spin; n <<= 1)

{

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

{

sw\_atomic\_cpu\_pause();

}

if (\*atomic == 0 && sw\_atomic\_cmp\_set(atomic, 0, 1))

{

return SW\_OK;

}

}

}

swYield();

}

return SW\_ERR;

源码解释：首先获得atomic\_t对象的指针，接着进入一个无限循环。在该循环里，首先判断atomic的值是否为0，并尝试将其赋值为1，如果成功，则直接返回。若前两个判断条件不成立，接着判断CPU数目是否大于1（多核），如果是则进入循环，循环变量为n，每次循环内先执行n次sw\_atomic\_cpu\_pause()，随后再次尝试设置atomic的值为1。每次循环结束后将n的值左移一位（值\*2，这就是为何spin的值要是2的幂）。如果CPU的数目为1或者直到内循环结束也没有设置成功atomic，就调用swYield方法交出线程的使用权。

这里就是在不停的尝试申请atomic锁，如果失败就等待一段时间后再次申请，每次失败后等待的时间会加长。

六．信号量

Swoole的Semaphore信号量的实现是基于Linux的semget、semop和semctl函数实现的。结构体swSem用于封装信号量相关的属性，其声明在swoole.h的421-427行，如下：

//信号量

typedef struct \_swSem

{

key\_t key; // 关键字，通常由ftok()返回

int semid; // 信号量id

int lock\_num; //

} swSem;

创建一个swSem的函数声明在swoole.h文件的535行，如下：

int swSem\_create(swLock \*lock, key\_t key, int n);

其中lock为信号量对象，key为信号量关键字，n为该信号量拥有的资源数。

该函数具体定义在Semaphore.c文件内，其核心代码如下：

lock->type = SW\_SEM;

if ((ret = semget(key, n, IPC\_CREAT | 0666)) < 0)

{

return SW\_ERR;

}

lock->object.sem.semid = ret;

lock->object.sem.lock\_num = 0;

源码解释：设置锁类型为SW\_SEM，并调用semget函数创造一个信号量，并返回信号的id。设置信号量id并设置lock\_num为0。

swSem的操作函数共有三个：swSem\_lock,swSem\_unlock,swSem\_free。分别对应加锁、解锁和销毁。

swSem\_lock的核心代码如下：

struct sembuf sem;

sem.sem\_flg = SEM\_UNDO;

sem.sem\_num = lock->object.sem.lock\_num;

// sem.sem\_op = 1; 此行为原始代码，经作者确认为bug

sem.sem\_op = -1;

return semop(lock->object.sem.semid, &sem, 1);

源码解释：创建一个struct sembuf 结构体，设置sem\_flg为SEM\_UNFO(程序结束时(不论正常或不正常)，保证信号值会被重设为semop()调用前的值)，指定操作的信号量为lock\_num,设置sem\_op为-1（请求资源），最后调用semop函数操作信号量。

swSem\_unlock的核心代码如下：

struct sembuf sem;

sem.sem\_flg = SEM\_UNDO;

sem.sem\_num = lock->object.sem.lock\_num;

// sem.sem\_op = -1; 此行为原始代码，经作者确认为bug

sem.sem\_op = 1;

return semop(lock->object.sem.semid, &sem, 1);

源码解释：创建一个struct sembuf 结构体，设置sem\_flg为SEM\_UNFO(程序结束时(不论正常或不正常)，保证信号值会被重设为semop()调用前的值)，指定操作的信号量为lock\_num,设置sem\_op为1（释放资源），最后调用semop函数操作信号量。

swSem\_free函数就是调用semctl方法释放信号量。

七．Cond条件变量

条件变量的用处是使线程睡眠等待某种条件出现后唤醒线程，是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个是线程等待“条件成立”而挂起，另一个是线程使“条件成立”并发出信号。为了防止竞争，条件变量通常与一个互斥锁结合在一起。

Swoole中，条件变量的实现使用了pthread\_cond\*方法族，其封装结构体swCond声明在swoole.c文件的451 - 461行，如下：

//Cond

typedef struct \_swCond

{

swLock lock; // 互斥锁

pthread\_cond\_t cond; // 条件变量对象

int (\*wait)(struct \_swCond \*object); // 四个操作函数

int (\*timewait)(struct \_swCond \*object,long,long);

int (\*notify)(struct \_swCond \*object);

int (\*broadcast)(struct \_swCond \*object);

} swCond;

swCond的相关操作函数共六个，声明在swoole.c文件中的546 – 551行，如下：

int swCond\_create(swCond \*cond);

int swCond\_notify(swCond \*cond);

int swCond\_broadcast(swCond \*cond);

int swCond\_timewait(swCond \*cond, long sec, long nsec);

int swCond\_wait(swCond \*cond);

void swCond\_free(swCond \*cond);

这六个函数均在Cond.c文件中被定义。

1. swCond\_create核心代码：

if (pthread\_cond\_init(&cond->cond, NULL) < 0)

{

swWarn("pthread\_cond\_init fail. Error: %s [%d]", strerror(errno), errno);

return SW\_ERR;

}

if (swMutex\_create(&cond->lock, 0) < 0)

{

return SW\_ERR;

}

源码解释：调用pthread\_cond\_init创建并初始化一个条件变量，并创建对应的互斥锁。

1. swCond\_notify方法调用pthread\_cond\_signal方法，向另一个正处于阻塞等待状态的线程发信号，使其脱离阻塞状态。需要注意的是，如果有多个线程正在等待，则根据优先级的高低决定由哪个线程收到信号；若优先级相同，则优先让等待时间最长的线程执行。
2. swCond\_broadcast方法调用pthread\_cond\_broadcast方法，向所有正处于阻塞等待状态的线程发出信号使其脱离阻塞状态。
3. swCond\_timewait方法调用pthread\_cond\_timedwait方法，计时等待其他线程发出信号，等待时间由参数long sec,long nsec指定。
4. swCond\_wait方法调用pthread\_cond\_wait方法，等待其他线程发出信号。
5. swCond\_free方法调用pthread\_cond\_destroy方法销毁信号变量，并销毁互斥锁。

至此，swoole全部的锁和信号已分析完成。