## Swoole版本：1.7.4-stable

写在开头的废话：原本计划是于第四章开始reactor模块的分析，但是发现reactor模块牵扯到太多的其他模块，没法一开始就直接分析reactor，只能将其延后。一番考察之后，我决定从swoole的Lock模块入手继续分析。这一块我的知识储备有很大的空缺，估计会比较吃力……

-------------------------------------------- 我是分割线 -----------------------------------------------

Swoole中关于Lock模块的全部声明在swoole.h文件的 364 – 461行（结构体声明）以及 534 – 551 行（函数声明），包括swFileLock（文件锁），swMutex（互斥锁），swRWLock（读写锁），swSpinLock（自旋锁），swAtomicLock（原子锁），swSem（信号量）和swCond（条件变量）。其中除了swCond以外，其他锁和信号都通过swLock做了进一步封装。

Swoole声明了一个枚举类型SW\_LOCKS用来代表这些不同类型的锁，其定义如下：

enum SW\_LOCKS

{

SW\_RWLOCK = 1,

#define SW\_RWLOCK SW\_RWLOCK

SW\_FILELOCK = 2,

#define SW\_FILELOCK SW\_FILELOCK

SW\_MUTEX = 3,

#define SW\_MUTEX SW\_MUTEX

SW\_SEM = 4,

#define SW\_SEM SW\_SEM

SW\_SPINLOCK = 5,

#define SW\_SPINLOCK SW\_SPINLOCK

SW\_ATOMLOCK = 6,

#define SW\_ATOMLOCK SW\_ATOMLOCK

};

首先是swLock结构体，该结构体是全部锁类型的总封装，其定义在swoole.h文件的429 – 449行，声明如下：

struct \_swLock

{

int type; // 锁的类型，取值于SW\_LOCKS

union // 锁的对象，一个union，稍后分析

{

swMutex mutex;

swRWLock rwlock;

swFileLock filelock;

swSem sem;

swAtomicLock atomlock;

#ifdef HAVE\_SPINLOCK

swSpinLock spinlock;

#endif

} object;

int (\*lock\_rd)(struct \_swLock \*lock); // 以下是六个操作函数

int (\*lock)(struct \_swLock \*lock);

int (\*unlock)(struct \_swLock \*lock);

int (\*trylock\_rd)(struct \_swLock \*lock);

int (\*trylock)(struct \_swLock \*lock);

int (\*free)(struct \_swLock \*lock);

};

这里的union（中译“联合体”）是一个非常好的设计，这保证了在同一时刻，一个swLock只会存在一个类型的锁。这里需要做一下特别说明，lock\_rd和trylock\_rd两个函数是专门为了swFileLock和swRWLock设计的，其他锁没有这两个函数。

那么，一个个来。首先是文件锁，文件锁的结构体声明在swoole.h的 385 – 389 行：

//文件锁

typedef struct \_swFileLock

{

struct flock lock\_t;

int fd;

} swFileLock;

其中struct flock lock\_t为Linux系统的文件锁结构，int fd为被锁控制的文件描述符。（struct flock的详细说明参考<http://www.gnu.org/software/libc/manual/html_node/File-Locks.html>）

创建一个文件锁的函数声明在swoole.h的537行，如下：

int swFileLock\_create(swLock \*lock, int fd);

其定义就是初始化一个type为SW\_FILELOCK的锁并指定对应的操作函数。文件锁的操作函数声明在FileLock.c文件中，其声明如下：

static int swFileLock\_lock\_rd(swLock \*lock);

static int swFileLock\_lock\_rw(swLock \*lock);

static int swFileLock\_unlock(swLock \*lock);

static int swFileLock\_trylock\_rw(swLock \*lock);

static int swFileLock\_trylock\_rd(swLock \*lock);

static int swFileLock\_free(swLock \*lock);

其中rd代表读锁，rw代表写锁。

这里需要介绍一下fcntl函数，其函数原型为：

int fcntl(int fd, int cmd);   
int fcntl(int fd, int cmd, long arg);   
int fcntl(int fd, int cmd, struct flock \*lock);

在这里，该函数仅用于申请和释放文件锁，因此只用到了两个cmd：F\_SETLK 和 F\_SETLKW。F\_SETLK：在指定的字节范围获取锁（F\_RDLCK, F\_WRLCK）或释放锁（F\_UNLCK）。如果和另一个进程的锁操作发生冲突，返回 -1并将errno设置为EACCES或EAGAIN。

F\_SETLKW：行为如同F\_SETLK，除了不能获取锁时会睡眠等待外。如果在等待的过程中接收到信号，会即时返回并将errno置为EINTR。

回到swFileLock。因为每个函数很短，所以这里直接贴出全部操作函数：

// 申请文件读锁RDLOCK

static int swFileLock\_lock\_rd(swLock \*lock)

{

lock->object.filelock.lock\_t.l\_type = F\_RDLCK;

return fcntl(lock->object.filelock.fd, F\_SETLKW, &lock->object.filelock);

}

// 申请文件写锁

static int swFileLock\_lock\_rw(swLock \*lock)

{

lock->object.filelock.lock\_t.l\_type = F\_WRLCK;

return fcntl(lock->object.filelock.fd, F\_SETLKW, &lock->object.filelock);

}

// 释放文件锁

static int swFileLock\_unlock(swLock \*lock)

{

lock->object.filelock.lock\_t.l\_type = F\_UNLCK;

return fcntl(lock->object.filelock.fd, F\_SETLKW, &lock->object.filelock);

}

// 尝试是否能申请写锁

static int swFileLock\_trylock\_rw(swLock \*lock)

{

// lock->object.filelock.lock\_t.l\_type = F\_RDLCK; // 该处为原代码，经作者确认，为bug，下一行是正确的代码

lock->object.filelock.lock\_t.l\_type = F\_WRLCK;

return fcntl(lock->object.filelock.fd, F\_SETLK, &lock->object.filelock);

}

// 尝试是否能申请读锁

static int swFileLock\_trylock\_rd(swLock \*lock)

{

// lock->object.filelock.lock\_t.l\_type = F\_WRLCK; // 该处为原代码，经作者确认，为bug，下一行是正确的代码

lock->object.filelock.lock\_t.l\_type = F\_RDLCK;

return fcntl(lock->object.filelock.fd, F\_SETLK, &lock->object.filelock);

}

// 调用close函数释放锁

static int swFileLock\_free(swLock \*lock)

{

return close(lock->object.filelock.fd);

}

这里可能有的人会有疑问：明明fcntl的函数要求传入一个flock结构体，而这里却传入了一个swFileLock结构体呢？因为这里flock结构体位于swFileLock结构体头部，swFileLock的首地址就是flock结构体的首地址。

（swoole的文件锁本身并没有复杂的结构，这里需要对flock以及fcntl函数有足够的了解，本文仅作简单说明，详细信息请参考<http://www.cnblogs.com/papam/archive/2009/09/02/1559154.html>）