**dispatch\_group\_create**

dispatch\_group\_t

dispatch\_group\_create(void)

{

return (dispatch\_group\_t)dispatch\_semaphore\_create(LONG\_MAX);

}

dispatch\_group\_create其实就是创建了一个value为LONG\_MAX的dispatch\_semaphore信号量

**dispatch\_group\_async**

void

dispatch\_group\_async(dispatch\_group\_t dg, dispatch\_queue\_t dq,

dispatch\_block\_t db)

{

dispatch\_group\_async\_f(dg, dq, \_dispatch\_Block\_copy(db),

\_dispatch\_call\_block\_and\_release);

}

dispatch\_group\_async只是dispatch\_group\_async\_f的封装

**dispatch\_group\_async\_f**

void

dispatch\_group\_async\_f(dispatch\_group\_t dg, dispatch\_queue\_t dq, void \*ctxt,

dispatch\_function\_t func)

{

dispatch\_continuation\_t dc;

\_dispatch\_retain(dg);

dispatch\_group\_enter(dg);

dc = fastpath(\_dispatch\_continuation\_alloc\_cacheonly());

if (!dc) {

dc = \_dispatch\_continuation\_alloc\_from\_heap();

}

dc->do\_vtable = (void \*)(DISPATCH\_OBJ\_ASYNC\_BIT | DISPATCH\_OBJ\_GROUP\_BIT);

dc->dc\_func = func;

dc->dc\_ctxt = ctxt;

dc->dc\_group = dg;

// No fastpath/slowpath hint because we simply don't know

if (dq->dq\_width != 1 && dq->do\_targetq) {

return \_dispatch\_async\_f2(dq, dc);

}

\_dispatch\_queue\_push(dq, dc);

}

从上面的代码我们可以看出dispatch\_group\_async\_f和dispatch\_async\_f相似。dispatch\_group\_async\_f多了dispatch\_group\_enter(dg);，另外在do\_vtable的赋值中dispatch\_group\_async\_f多了一个DISPATCH\_OBJ\_GROUP\_BIT的标记符。既然添加了dispatch\_group\_enter必定会存在dispatch\_group\_leave。在之前《深入理解GCD之dispatch\_queue》介绍\_dispatch\_continuation\_pop函数的源码中有一段代码如下：

\_dispatch\_client\_callout(dc->dc\_ctxt, dc->dc\_func);

if (dg) {

//group需要进行调用dispatch\_group\_leave并释放信号

dispatch\_group\_leave(dg);

\_dispatch\_release(dg);

}

所以dispatch\_group\_async\_f函数中的dispatch\_group\_leave是在\_dispatch\_continuation\_pop函数中调用的。

这里概括一下dispatch\_group\_async\_f的工作流程：

1. 调用dispatch\_group\_enter；
2. 将block和queue等信息记录到dispatch\_continuation\_t结构体中，并将它加入到group的链表中；
3. \_dispatch\_continuation\_pop执行时会判断任务是否为group，是的话执行完任务再调用dispatch\_group\_leave以达到信号量的平衡。

**dispatch\_group\_enter**

void

dispatch\_group\_enter(dispatch\_group\_t dg)

{

dispatch\_semaphore\_t dsema = (dispatch\_semaphore\_t)dg;

(void)dispatch\_semaphore\_wait(dsema, DISPATCH\_TIME\_FOREVER);

}

dispatch\_group\_enter将dispatch\_group\_t转换成dispatch\_semaphore\_t，并调用dispatch\_semaphore\_wait，原子性减1后，进入等待状态直到有信号唤醒。所以说**dispatch\_group\_enter就是对dispatch\_semaphore\_wait的封装**。

**dispatch\_group\_leave**

void

dispatch\_group\_leave(dispatch\_group\_t dg)

{

dispatch\_semaphore\_t dsema = (dispatch\_semaphore\_t)dg;

dispatch\_atomic\_release\_barrier();

long value = dispatch\_atomic\_inc2o(dsema, dsema\_value);//dsema\_value原子性加1

if (slowpath(value == LONG\_MIN)) {//内存溢出，由于dispatch\_group\_leave在dispatch\_group\_enter之前调用

DISPATCH\_CLIENT\_CRASH("Unbalanced call to dispatch\_group\_leave()");

}

if (slowpath(value == dsema->dsema\_orig)) {//表示所有任务已经完成，唤醒group

(void)\_dispatch\_group\_wake(dsema);

}

}

从上面的源代码中我们看到dispatch\_group\_leave将dispatch\_group\_t转换成dispatch\_semaphore\_t后将dsema\_value的值原子性加1。如果value为LONG\_MIN程序crash；如果value等于dsema\_orig表示所有任务已完成，调用\_dispatch\_group\_wake唤醒group（\_dispatch\_group\_wake的用于和notify有关，我们会在后面介绍）。因为在enter的时候进行了原子性减1操作。所以在leave的时候需要原子性加1。

这里先说明一下enter和leave之间的关系：

1. **dispatch\_group\_leave与dispatch\_group\_enter配对使用**。当调用了dispatch\_group\_enter而没有调用dispatch\_group\_leave时，由于value不等于dsema\_orig不会走到唤醒逻辑，dispatch\_group\_notify中的任务无法执行或者dispatch\_group\_wait收不到信号而卡住线程。
2. **dispatch\_group\_enter必须在dispatch\_group\_leave之前出现**。当dispatch\_group\_leave比dispatch\_group\_enter多调用了一次或者说在dispatch\_group\_enter之前被调用的时候，dispatch\_group\_leave进行原子性加1操作，相当于value为LONGMAX+1，发生数据长度溢出，变成LONG\_MIN，由于value == LONG\_MIN成立，程序发生crash。

**dispatch\_group\_notify**

void

dispatch\_group\_notify(dispatch\_group\_t dg, dispatch\_queue\_t dq,

dispatch\_block\_t db)

{

dispatch\_group\_notify\_f(dg, dq, \_dispatch\_Block\_copy(db),

\_dispatch\_call\_block\_and\_release);

}

dispatch\_group\_notify是dispatch\_group\_notify\_f的封装，具体实现在后者。

**dispatch\_group\_notify\_f**

void

dispatch\_group\_notify\_f(dispatch\_group\_t dg, dispatch\_queue\_t dq, void \*ctxt,

void (\*func)(void \*))

{

dispatch\_semaphore\_t dsema = (dispatch\_semaphore\_t)dg;

struct dispatch\_sema\_notify\_s \*dsn, \*prev;

//封装dispatch\_continuation\_t结构体

// FIXME -- this should be updated to use the continuation cache

while (!(dsn = calloc(1, sizeof(\*dsn)))) {

sleep(1);

}

dsn->dsn\_queue = dq;

dsn->dsn\_ctxt = ctxt;

dsn->dsn\_func = func;

\_dispatch\_retain(dq);

dispatch\_atomic\_store\_barrier();

//将结构体放到链表尾部，如果链表为空同时设置链表头部节点并唤醒group

prev = dispatch\_atomic\_xchg2o(dsema, dsema\_notify\_tail, dsn);

if (fastpath(prev)) {

prev->dsn\_next = dsn;

} else {

\_dispatch\_retain(dg);

(void)dispatch\_atomic\_xchg2o(dsema, dsema\_notify\_head, dsn);

if (dsema->dsema\_value == dsema->dsema\_orig) {//任务已经完成，唤醒group

\_dispatch\_group\_wake(dsema);

}

}

}

所以dispatch\_group\_notify函数只是用链表把所有回调通知保存起来，等待调用。

**\_dispatch\_group\_wake**

static long

\_dispatch\_group\_wake(dispatch\_semaphore\_t dsema)

{

struct dispatch\_sema\_notify\_s \*next, \*head, \*tail = NULL;

long rval;

//将dsema的dsema\_notify\_head赋值为NULL，同时将之前的内容赋给head

head = dispatch\_atomic\_xchg2o(dsema, dsema\_notify\_head, NULL);

if (head) {

// snapshot before anything is notified/woken <rdar://problem/8554546>

//将dsema的dsema\_notify\_tail赋值为NULL，同时将之前的内容赋给tail

tail = dispatch\_atomic\_xchg2o(dsema, dsema\_notify\_tail, NULL);

}

//将dsema的dsema\_group\_waiters设置为0，并返回原来的值

rval = dispatch\_atomic\_xchg2o(dsema, dsema\_group\_waiters, 0);

if (rval) {

//循环调用semaphore\_signal唤醒当初等待group的信号量，使得dispatch\_group\_wait函数返回。

// wake group waiters

#if USE\_MACH\_SEM

\_dispatch\_semaphore\_create\_port(&dsema->dsema\_waiter\_port);

do {

kern\_return\_t kr = semaphore\_signal(dsema->dsema\_waiter\_port);

DISPATCH\_SEMAPHORE\_VERIFY\_KR(kr);

} while (--rval);

#elif USE\_POSIX\_SEM

do {

int ret = sem\_post(&dsema->dsema\_sem);

DISPATCH\_SEMAPHORE\_VERIFY\_RET(ret);

} while (--rval);

#endif

}

if (head) {

//获取链表，依次调用dispatch\_async\_f异步执行在notify函数中的任务即Block。

// async group notify blocks

do {

dispatch\_async\_f(head->dsn\_queue, head->dsn\_ctxt, head->dsn\_func);

\_dispatch\_release(head->dsn\_queue);

next = fastpath(head->dsn\_next);

if (!next && head != tail) {

while (!(next = fastpath(head->dsn\_next))) {

\_dispatch\_hardware\_pause();

}

}

free(head);

} while ((head = next));

\_dispatch\_release(dsema);

}

return 0;

}

\_dispatch\_group\_wake主要的作用有两个：

1. 调用semaphore\_signal唤醒当初等待group的信号量，使得dispatch\_group\_wait函数返回。
2. 获取链表，依次调用dispatch\_async\_f异步执行在notify函数中的任务即Block。

到这里我们已经差不多知道了dispatch\_group工作过程，我们用一张图表示：

dispatch\_group

**dispatch\_group\_wait**

long

dispatch\_group\_wait(dispatch\_group\_t dg, dispatch\_time\_t timeout)

{

dispatch\_semaphore\_t dsema = (dispatch\_semaphore\_t)dg;

if (dsema->dsema\_value == dsema->dsema\_orig) {//没有需要执行的任务

return 0;

}

if (timeout == 0) {//返回超时

#if USE\_MACH\_SEM

return KERN\_OPERATION\_TIMED\_OUT;

#elif USE\_POSIX\_SEM

errno = ETIMEDOUT;

return (-1);

#endif

}

return \_dispatch\_group\_wait\_slow(dsema, timeout);

}

dispatch\_group\_wait用于等待group中的任务完成。

**\_dispatch\_group\_wait\_slow**

static long

\_dispatch\_group\_wait\_slow(dispatch\_semaphore\_t dsema, dispatch\_time\_t timeout)

{

long orig;

again:

// check before we cause another signal to be sent by incrementing

// dsema->dsema\_group\_waiters

if (dsema->dsema\_value == dsema->dsema\_orig) {

return \_dispatch\_group\_wake(dsema);

}

// Mach semaphores appear to sometimes spuriously wake up. Therefore,

// we keep a parallel count of the number of times a Mach semaphore is

// signaled (6880961).

(void)dispatch\_atomic\_inc2o(dsema, dsema\_group\_waiters);

// check the values again in case we need to wake any threads

if (dsema->dsema\_value == dsema->dsema\_orig) {

return \_dispatch\_group\_wake(dsema);

}

#if USE\_MACH\_SEM

mach\_timespec\_t \_timeout;

kern\_return\_t kr;

\_dispatch\_semaphore\_create\_port(&dsema->dsema\_waiter\_port);

// From xnu/osfmk/kern/sync\_sema.c:

// wait\_semaphore->count = -1; /\* we don't keep an actual count \*/

//

// The code above does not match the documentation, and that fact is

// not surprising. The documented semantics are clumsy to use in any

// practical way. The above hack effectively tricks the rest of the

// Mach semaphore logic to behave like the libdispatch algorithm.

switch (timeout) {

default:

do {

uint64\_t nsec = \_dispatch\_timeout(timeout);

\_timeout.tv\_sec = (typeof(\_timeout.tv\_sec))(nsec / NSEC\_PER\_SEC);

\_timeout.tv\_nsec = (typeof(\_timeout.tv\_nsec))(nsec % NSEC\_PER\_SEC);

kr = slowpath(semaphore\_timedwait(dsema->dsema\_waiter\_port,

\_timeout));

} while (kr == KERN\_ABORTED);

if (kr != KERN\_OPERATION\_TIMED\_OUT) {

DISPATCH\_SEMAPHORE\_VERIFY\_KR(kr);

break;

}

// Fall through and try to undo the earlier change to

// dsema->dsema\_group\_waiters

case DISPATCH\_TIME\_NOW:

while ((orig = dsema->dsema\_group\_waiters)) {

if (dispatch\_atomic\_cmpxchg2o(dsema, dsema\_group\_waiters, orig,

orig - 1)) {

return KERN\_OPERATION\_TIMED\_OUT;

}

}

// Another thread called semaphore\_signal().

// Fall through and drain the wakeup.

case DISPATCH\_TIME\_FOREVER:

do {

kr = semaphore\_wait(dsema->dsema\_waiter\_port);

} while (kr == KERN\_ABORTED);

DISPATCH\_SEMAPHORE\_VERIFY\_KR(kr);

break;

}

#elif USE\_POSIX\_SEM

//这部分代码省略

#endif

goto again;

}

从上面的代码我们发现\_dispatch\_group\_wait\_slow和\_dispatch\_semaphore\_wait\_slow的逻辑很接近。都利用mach内核的semaphore进行信号的发送。区别在于\_dispatch\_semaphore\_wait\_slow在等待结束后是return，而\_dispatch\_group\_wait\_slow在等待结束是调用\_dispatch\_group\_wake去唤醒这个group。

**总结**

1. dispatch\_group是一个初始值为LONG\_MAX的信号量，group中的任务完成是判断其value是否恢复成初始值。
2. dispatch\_group\_enter和dispatch\_group\_leave必须成对使用并且支持嵌套。
3. 如果dispatch\_group\_enter比dispatch\_group\_leave多，由于value不等于dsema\_orig不会走到唤醒逻辑，dispatch\_group\_notify中的任务无法执行或者dispatch\_group\_wait收不到信号而卡住线程。如果是dispatch\_group\_leave多，则会引起崩溃。

作者：NeroXie  
链接：https://www.jianshu.com/p/e93fd15d93d3  
来源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。