第13节课内容总结

GCD单例的原理

基本思想就是通过状态的判断使得block只被调用一次。
核心代码：

```
void
dispatch_once_f(dispatch_once_t *val, void *ctxt,
    dispatch_function_t func)
{                                把dispatch_once_t转换成dispatch_once_gate_t类型
    dispatch_once_gate_t l = (dispatch_once_gate_t)val;

#if !DISPATCH_ONCE_INLINE_FASTPATH ||
    DISPATCH_ONCE_USE_QUIESCENT_COUNTER
    uintptr_t v = os_atomic_load(&l->dgo_once, acquire);
    if (likely(v == DLOCK_ONCE_DONE)) {
        return;
    }                    如果l的状态为DLOCK_ONCE_DONE 说明block已经被执行过
                                        直接返回
#if DISPATCH_ONCE_USE_QUIESCENT_COUNTER
    if (likely(DISPATCH_ONCE_IS_GEN(v))) {
        return _dispatch_once_mark_done_if_quiesced(l, v);
    }
#endif
#endif   判断l的状态是否为DLOCK_ONCE_DONE，如果不是，则执行block
    if (_dispatch_once_gate_tryenter(l)) {
        return _dispatch_once_callout(l, ctxt, func);
    }
    return _dispatch_once_wait(l);  如果block正在执行当中，则一直等待
}
```

栅栏函数

栅栏函数的效果：等待栅栏函数前添加到队列里面的任务全部执行完成之后，才会执行栅栏函数里面的任务，栅栏函数里面的任务执行完成之后才会执行栅栏函数后面的队列里面的任务。
需要注意的点：

1. 栅栏函数只对同一队列起作用。
2. 栅栏函数对全局并发队列无效。

调度组

调度组的效果：等待调度组前面的任务执行完才会执行dispatch_group_notify函数里面的任务。调度组和队列没有关系，只要是同一调度组就可以。

信号量dispatch_semaphore

dispatch_semaphore主要就是三个方法：

1. dispatch_semaphore_create(long value);这个函数是创建一个dispatch_semaphore_t类型的信号量，并且创建的时候需要指定信号量的大小。
2. dispatch_semaphore_wait(dispatch_semaphore_t dsema, dispatch_time_t timeout); 等待信号量。如果信号量值为0，那么该函数就会一直等待，也就是不返回（相当于阻塞当前线程），直到该函数等待的信号量的值大于等于1，该函数会对信号量的值进行减1操作，然后返回。
3. dispatch_semaphore_signal(dispatch_semaphore_t deem); 发送信号量。该函数会对信号量的值进行加1操作。

通过这三个方法，就能控制GCD的最大并发数量。

信号量在使用的时候需要注意：`dispatch_semaphore_wait`和`dispatch_semaphore_signal`一定要成对出现。因为在信号量释放的时候，如果dsema_orig初始信号量的大小大于dsema_value（通过dispatch_semaphore_wait和dispatch_semaphore_signal改变之后的信号量的大小）就会触发崩溃。

```
void
_dispatch_semaphore_dispose(dispatch_object_t dou,
        DISPATCH_UNUSED bool *allow_free)
{
    dispatch_semaphore_t dsema = dou._dsema;

    //dsema_orig 创建的信号量的大小
    //dsema_value ++--信号量的大小
    if (dsema->dsema_value < dsema->dsema_orig) {
        DISPATCH_CLIENT_CRASH(dsema->dsema_orig - dsema->dsema_value,
                "Semaphore object deallocated while in use");
    }

    _dispatch_sema4_dispose(&dsema->dsema_sema, _DSEMA4_POLICY_FIFO);
}
```

dispatch_source

dispatch_source是用来监听事件的，可以创建不同类型的dispatch_source来监听不同的事件。

dispatch_source可以监听的事件类型：

名称	说明	dispatch_source_get_handle
DISPATCH_SOURCE_TYPE_DATA_ADD	自定义事件，变量增加	n/a
DISPATCH_SOURCE_TYPE_DATA_OR	自定义事件，变量OR	n/a
DISPATCH_SOURCE_TYPE_DATA_REPLACE	自定义事件，变量REPLACE。如果传入的数据为0，将不会出发handler	n/a
DISPATCH_SOURCE_TYPE_MACH_SEND	监听Mach port的deadname通知，handle是具有send权限的Mach port 包括send或send_once	mach port
DISPATCH_SOURCE_TYPE_MACH_RECV	监听Mach port获取待等待处理的消息	mach port
DISPATCH_SOURCE_TYPE_MEMORYPRESSURE	监听系统中的内存压力	n/a
DISPATCH_SOURCE_TYPE_PROC	监听进程事件	进程ID
DISPATCH_SOURCE_TYPE_READ	监听文件描述符是否有可读的数据	文件描述符（int）
DISPATCH_SOURCE_TYPE_SIGNAL	监听当前进程的signal	signal number(int)
DISPATCH_SOURCE_TYPE_TIMER	定时器监听	n/a
DISPATCH_SOURCE_TYPE_VNODE	监听文件描述符事件	文件描述符（int）
DISPATCH_SOURCE_TYPE_WRITE	监听文件描述符使用可用的buffer空间来写数据	文件描述符（int）

dispatch_source的具体用法：在任一线程上调用它的dispatch_source_merge_data函数，会执行dispatch_source事先定义好的句柄（可以把句柄简单理解为一个block）。

dispatch_source的几个方法：

dispatch_source_create	创建源
dispatch_source_set_event_handler	设置源事件回调
dispatch_source_merge_data	源事件设置数据
dispatch_source_get_data	获取源事件数据
dispatch_resume	继续
dispatch_suspend	挂起
dispatch_source_cancel	取消源事件