417 days ago

深入理解 GCD

前言

首先提出一些问题:

1. `dispatch_async` 函数如何实现，分发到主队列和全局队列有什么区别，一定会新建线程执行任务么?
2. `dispatch_sync` 函数如何实现，为什么说 GCD 死锁是队列导致的而不是线程，死锁不是操作系统的概念么?
3. 信号量是如何实现的，有哪些使用场景?
4. `dispatch_group` 的等待与通知、`dispatch_once` 如何实现?
5. `dispatch_source` 用来做定时器如何实现，有什么优点和用途?
6. `dispatch_suspend` 和 `dispatch_resume` 如何实现，队列的的暂停和计时器的暂停有区别么?

以上问题基本都是对 GCD 常用 API 的追问与思考，深入理解这些问题有助于更好地使用 GCD，比如以下代码的执行结果是什么?

```
- (void)viewDidLoad {
    [super viewDidLoad];
    dispatch_queue_t queue = dispatch_queue_create("com.bests
    dispatch_sync(queue, ^{
        NSLog(@"current thread = %@", [NSThread currentThread
        dispatch_sync(dispatch_get_main_queue(), ^{
            NSLog(@"current thread = %@", [NSThread currentTh
        });
    });
}
```

以下内容为个人的学习总结，仅供参考，不一定适合新手入门。最好的学习方法还是自己下载一份 源码 并仔细阅读学习。

文章主要分析了常见 API 的实现原理，因水平所限，不可避免的有理解错误的地方，欢迎指出。如果对具体分析不感兴趣，可以直接跳到文章末尾的"总结"部分。

知识储备

阅读 GCD 源码 之前，需要了解一些相关知识，这样才能在读到源码时不至于一脸懵逼，进而影响理解。

DISPATCH_DECL

GCD 中对变量的定义大多遵循如下格式:

```
#define DISPATCH_DECL(name) typedef struct name##_s *name##_t
```

比如说非常常见的 `DISPATCH_DECL(dispatch_queue);`，它的展开形式是:

```
typedef struct dispatch_queue_s *dispatch_queue_t;
```

这行代码定义了一个 `dispatch_queue_t` 类型的指针，指向一个 `dispatch_queue_s` 类型的结构体。

TSD

TSD(Thread-Specific Data) 表示线程私有数据。在 C++ 中，全局变量可以被所有线程访问，局部变量只有函数内部可以访问。而 TSD 的作用就是能够在同一个线程的不同函数中被访问。在不同线程中，虽然名字相同，但是获取到的数据随线程不同而不同。

通常，我们可以利用 POSIX 库提供的 API 来实现 TSD:

```
int pthread_key_create(pthread_key_t *key, void (*destr_funct
```

这个函数用来创建一个 key，在线程退出时会将 key 对应的数据传入 `destr_function` 函数中进行清理。

我们分别使用 get/set 方法来访问/修改 key 对应的数据:

```
int  pthread_setspecific(pthread_key_t  key,  const   void  *
void * pthread_getspecific(pthread_key_t key)
```

在 GCD 中定义了六个 key，根据名字大概能猜出各自的含义:

```
pthread_key_t dispatch_queue_key;
pthread_key_t dispatch_sema4_key;
pthread_key_t dispatch_cache_key;
pthread_key_t dispatch_io_key;
pthread_key_t dispatch_apply_key;
pthread_key_t dispatch_bcounter_key;
```

fastpath && slowpath

这是定义在 `internal.h` 中的两个宏:

```
#define fastpath(x) ((typeof(x))__builtin_expect((long)(x), ~
#define slowpath(x) ((typeof(x))__builtin_expect((long)(x), 0
```

为了理解所谓的快路径和慢路径，我们需要先学习一点计算机基础知识。比如这段非常简单的代码:

```
if (x)
    return 1;
```

```
else
    return 39;
```

由于计算机并非一次只读取一条指令，而是读取多条指令，所以在读到 if 语句时也会把 `return 1` 读取进来。如果 x 为 0，那么会重新读取 `return 39`，重读指令相对来说比较耗时。

如过 x 有非常大的概率是 0，那么 `return 1` 这条指令每次不可避免的会被读取，并且实际上几乎没有机会执行， 造成了不必要的指令重读。当然，最简单的优化就是:

```
if (!x)
    return 39;
else
    return 1;
```

然而对程序员来说，每次都做这样的判断非常烧脑，而且容易出错。于是 GCC 提供了一个内置函数 `__builtin_expect`:

```
long __builtin_expect (long EXP, long C)
```

它的返回值就是整个函数的返回值，参数 C 代表预计的值，表示程序员知道 EXP 的值很可能就是 C。比如上文中的例子可以这样写:

```
if (__builtin_expect(x, 0))
    return 1;
else
    return 39;
```

虽然写法逻辑不变，但是编译器会把汇编代码优化成 `if(!x)` 的形式。

因此，在苹果定义的两个宏中，`fastpath(x)` 依然返回 x，只是告诉编译器 x 的值一般不为 0，从而编译器可以进行优化。同理，`slowpath(x)` 表

示 x 的值很可能为 0，希望编译器进行优化。

dispatchqueuet

以 `dispatch_queue_create` 的源码为例:

```
dispatch_queue_create(const char *label, dispatch_queue_attr_
    // 省略 label 相关的操作
    dispatch_queue_t dq;
    dq = _dispatch_alloc(DISPATCH_VTABLE(queue),
            sizeof(struct dispatch_queue_s) - DISPATCH_QUEUE_
            DISPATCH_QUEUE_CACHELINE_PAD + label_len + 1);
    _dispatch_queue_init(dq);
    if (fastpath(!attr)) {
        return dq;
    }
    if (fastpath(attr == DISPATCH_QUEUE_CONCURRENT)) {
        dq->dq_width = UINT32_MAX;
        dq->do_targetq = _dispatch_get_root_queue(0, false);
    } else {
        dispatch_debug_assert(!attr, "Invalid attribute");
    }
    return dq;
}
```

我们知道创建队列时， attr 属性有三个值可选，`nil`、`DISPATCH_QUEUE_SERIAL`(实际上就是 nil) 或 `DISPATCH_QUEUE_CONCURRENT`。第一个 if 判断中，苹果认为串行队列，或者 `NULL` 参数更常见，因此 `!attr` 的值很有可能不为 0，这与上文的结论一致。

第二个判断中，参数几乎有只可能是 `DISPATCH_QUEUE_CONCURRENT`，因此 `attr == DISPATCH_QUEUE_CONCURRENT` 这个判断机会不会为 0，依然与 `fastpath` 的作用一致。

`_dispatch_get_root_queue` 会获取一个全局队列，它有两个参数，分别表示优先级和是否支持 overcommit。一共有四个优先级，`LOW`、`DEFAULT`、`HIGH` 和 `BACKGROUND`，因此共有 8 个全局队列。带有 overcommit 的队列表示每当有任务提交时，系统都会新开一个线程处理，这样就不会造成某个线程过载(overcommit)。

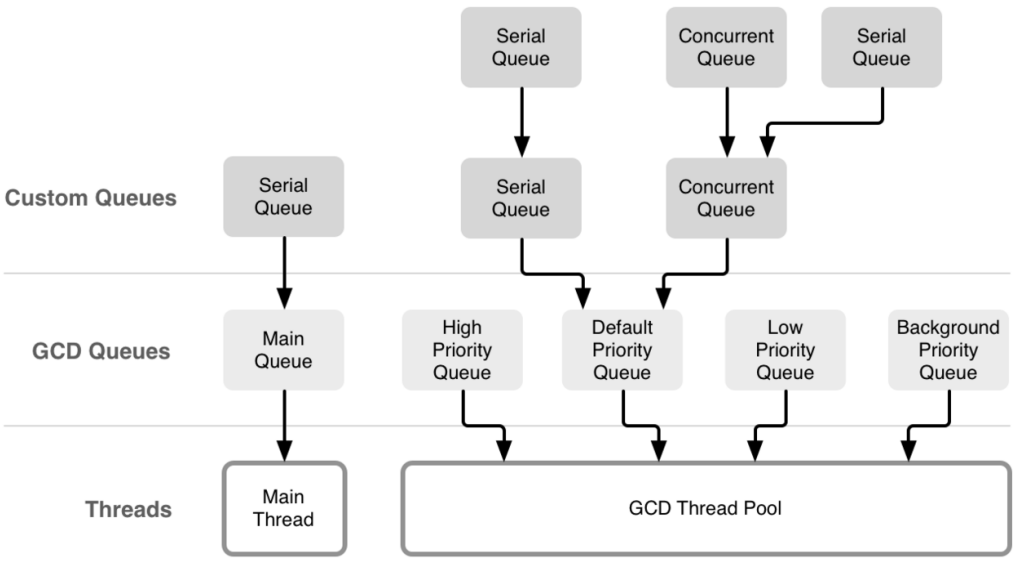
这 8 个全局队列的序列号是 4-11，序列号为 1 的队列是主队列，2 是 manager 队列，用来管理 GCD 内部的任务(比如下文介绍的定时器)，3 这个序列号暂时没有使用。队列 的 `dq_width` 被设置为 `UINT32_MAX`，表示这些队列不限制并发数。

作为对比，在 `_dispatch_queue_init` 中，并发数限制为 1，也就是串行队列的默认设置:

```
static inline void _dispatch_queue_init(dispatch_queue_t dq)
    dq->do_next = DISPATCH_OBJECT_LISTLESS;
    dq->do_targetq = _dispatch_get_root_queue(0, true);
    dq->dq_running = 0;
    dq->dq_width = 1;
}
```

注意这行代码: `dq->do_targetq = _dispatch_get_root_queue(0, true);`，它涉及到 GCD 队列与 block 的一个重要模型，`target_queue`。向任何队列中提交的 block，都会被放到它的目标队列中执行，而普通串行队列的目标队列就是一个支持 overcommit 的全局队列，全局队列的底层则是一个线程池。

借用 objc 的文章 中的图片来表示:



dispatch_async

直接上函数实现:

```
atch_async(dispatch_queue_t queue, dispatch_block_t block) {
dispatch_async_f(dq, _dispatch_Block_copy(work), _dispatch_ca
```

队列其实就是一个用来提交 block 的对象，当 block 提交到队列中后，将按照 “先入先出(FIFO)” 的顺序进行处理。系统在 GCD 的底层会维护一个线程池，用来执行这些 block。

block 参数的类型是 `dispatch_block_t`，它是一个没有参数，没有返回值的 block:

```
typedef void (^dispatch_block_t)(void);
```

`dispatch_async` 的函数很简单，它将 block 复制了一份，然后调用另一个函数 `dispatch_async_f`:

```
dispatch_async_f(dispatch_queue_t queue, void *context, dispa
```

work 参数是一个函数，在实际调用时，会把第二参数 context 作为参数传入，以 `_dispatch_call_block_and_release` 为例:

```
void _dispatch_call_block_and_release(void *block) {
    void (^b)(void) = block;
    b();
    Block_release(b);
}
```

省略各种分支后的 `dispatch_async_f` 函数实现如下:

```
void dispatch_async_f(dispatch_queue_t dq, void *ctxt, dispat
    dispatch_continuation_t dc;
```

```
    if (dq->dq_width == 1) {
        return dispatch_barrier_async_f(dq, ctxt, func);
    }
    dc->do_vtable = (void *)DISPATCH_OBJ_ASYNC_BIT;
    dc->dc_func = func;
    dc->dc_ctxt = ctxt;
    if (dq->do_targetq) {
        return _dispatch_async_f2(dq, dc);
    }
    _dispatch_queue_push(dq, dc);
}
```

可见如果是串行队列 (dq_width = 1)，会调用 `dispatch_barrier_async_f` 函数处理，这个后文会有介绍。如果有 `do_targetq` 则进行转发，否则调用 `_dispatch_queue_push` 入队。

这里的 `dispatch_continuation_t` 其实是对 block 的封装，然后调用 `_dispatch_queue_push` 这个宏将封装好的 block 放入队列中。

把这个宏展开，然后依次分析调用栈，选择一条主干调用线，结果如下:

```
_dispatch_queue_push
└_dispatch_trace_queue_push
    └_dispatch_queue_push
        └_dispatch_queue_push_slow
            └_dispatch_queue_push_list_slow2
                └_dispatch_wakeup
                    └dx_probe
```

队列中保存了一个链表，我们首先将新的 block 添加到链表尾部，然后调用 `dx_probe` 宏，它依赖于 vtable 数据结构，GCD 中的大部分对象，比如队列等，都具有这个数据结构。它定义了对象在不同操作下该执行的方法，比如在这里的 `probe` 操作下，实际上会执行 `_dispatch_queue_wakeup_global` 方法，调用栈如下

```
_dispatch_queue_wakeup_global
└_dispatch_queue_wakeup_global2
    └_dispatch_queue_wakeup_global_slow
```

在 `_dispatch_queue_wakeup_global_slow` 我们见到了熟悉的老朋友，pthread 线程:

```
static void _dispatch_queue_wakeup_global_slow(dispatch_queue
    // 如果线程池已满，则直接调用 _dispatch_worker_thread
    // 否则创建线程池
    pthread_t pthr;
    while ((r = pthread_create(&pthr, NULL, _dispatch_worker_
        if (r != EAGAIN) {
            (void)dispatch_assume_zero(r);
        }
        sleep(1);
    }
    r = pthread_detach(pthr);
    (void)dispatch_assume_zero(r);
}
```

由此可见这里确实使用了线程池。创建线程后会执行 `_dispatch_worker_thread` 回调:

```
_dispatch_worker_thread
└──_dispatch_worker_thread4
    └──_dispatch_continuation_pop
```

在 pop 函数中，我们拿到了最早加入的任务，然后执行:

```
static inline void _dispatch_continuation_pop(dispatch_object
    // ...
    _dispatch_client_callout(dc->dc_ctxt, dc->dc_func);
    if (dg) {
        dispatch_group_leave(dg);
        _dispatch_release(dg);
    }
}
```

`dispatch_async` 的实现比较复杂，主要是因为其中的数据结构较多，分支流程控制比较复杂。但思路其实很简单，用链表保存所有提交的 block，然后在底层线程池中，依次取出 block 并执行。

如果熟悉了相关数据结构和调用流程，接下来研究 GCD 的其他 API 就比较轻松了。

dispatch_sync

同步方法的实现相对来说和异步类似，而且更简单，调用栈如下:

```
dispatch_sync
└──dispatch_sync_f
    └──_dispatch_sync_f2
        └──_dispatch_sync_f_slow
```

```
static void _dispatch_sync_f_slow(dispatch_queue_t dq, void *
    _dispatch_thread_semaphore_t sema = _dispatch_get_thread_
    struct dispatch_sync_slow_s {
        DISPATCH_CONTINUATION_HEADER(sync_slow);
    } dss = {
        .do_vtable = (void*)DISPATCH_OBJ_SYNC_SLOW_BIT,
        .dc_ctxt = (void*)sema,
    };
    _dispatch_queue_push(dq, (void *)&dss);

    _dispatch_thread_semaphore_wait(sema);
    _dispatch_put_thread_semaphore(sema);
    // ...
}
```

这里利用了线程专属信号量，保证了每次只有一个 block 被执行。

这条调用栈有多个分支，如果向当前串行队列提交任务就会走到上述分支，导致死锁。如果是向其它串行队列提交 block，则会利用原子性操作来实现，因此不会有死锁问题。

dispatch_semaphore

关于信号量的 API 不多，主要是三个，`create`、`wait` 和 `signal`。

信号量在初始化时要指定 value，随后内部将这个 value 存储起来。实际操作时会存两个 value，一个是当前的 value，一个是记录初始

value。

信号的 `wait` 和 `signal` 是互逆的两个操作。如果 value 大于 0，前者将 value 减一，此时如果 value 小于零就一直等待。

初始 value 必须大于等于 0，如果为 0 并随后调用 `wait` 方法，线程将被阻塞直到别的线程调用了 `signal` 方法。

dispatchsemaphorewait

首先从这个函数的源码看起:

```
long dispatch_semaphore_wait(dispatch_semaphore_t dsema, disp
    long value = dispatch_atomic_dec2o(dsema, dsema_value);
    dispatch_atomic_acquire_barrier();
    if (fastpath(value >= 0)) {
        return 0;
    }
    return _dispatch_semaphore_wait_slow(dsema, timeout);
}
```

第一行的 `dispatch_atomic_dec2o` 是一个宏，会调用 GCC 内置的函数 `__sync_sub_and_fetch`，实现减法的原子性操作。因此这一行的意思是将 dsema 的值减一，并把新的值赋给 value。

如果减一后的 value 大于等于 0 就立刻返回，没有任何操作，否则进入等待状态。

`_dispatch_semaphore_wait_slow` 函数针对不同的 timeout 参数，分了三种情况考虑:

```
case DISPATCH_TIME_NOW:
    while ((orig = dsema->dsema_value) < 0) {
        if (dispatch_atomic_cmpxchg2o(dsema, dsema_value, ori
            return KERN_OPERATION_TIMED_OUT;
        }
    }
```

这种情况下会立刻判断 `dsema->dsema_value` 与 `orig` 是否相等。如果 while 判断成立，内部的 if 判断一定也成立，此时会将 value 加一(也就是变为 0) 并返回。加一的原因是为了抵消 `wait` 函数一开始的减一操作。此时函数调用方会得到返回值 `KERN_OPERATION_TIMED_OUT`，表示由于等待时间超时而返回。

实际上 while 判断一定会成立，因为如果 value 大于等于 0，在上一个函数 `dispatch_semaphore_wait` 中就已经返回了。

第二种情况是 `DISPATCH_TIME_FOREVER` 这个 case:

```
case DISPATCH_TIME_FOREVER:
    do {
        kr = semaphore_wait(dsema->dsema_port);
    } while (kr == KERN_ABORTED);
    break;
```

进入 do-while 循环后会调用系统的 `semaphore_wait` 方法，`KERN_ABORTED` 表示调用者被一个与信号量系统无关的原因唤醒。因此一旦发生这种情况，还是要继续等待，直到收到 `signal` 调用。

在其他情况下(default 分支)，我们指定一个超时时间，这和 `DISPATCH_TIME_FOREVER` 的处理比较类似，不同的是我们调用了内核提供的 `semaphore_timedwait` 方法可以指定超时时间。

整个函数的框架如下:

```
static long _dispatch_semaphore_wait_slow(dispatch_semaphore_
again:
    while ((orig = dsema->dsema_sent_ksignals)) {
        if (dispatch_atomic_cmpxchg2o(dsema, dsema_sent_ksign
                orig - 1)) {
            return 0;
        }
    }
    switch (timeout) {
        default:  /* semaphore_timedwait */
        case DISPATCH_TIME_NOW: /* KERN_OPERATION_TIMED_OUT *
        case DISPATCH_TIME_FOREVER: /* semaphore_wait */
    }
```

```
    goto again;
}
```

可见信号量被唤醒后，会回到最开始的地方，进入 while 循环。这个判断条件一般都会成立，极端情况下由于内核存在 bug，导致 `orig` 和 `dsema_sent_ksignals` 不相等，也就是收到虚假 `signal` 信号时会忽略。

进入 while 循环后，if 判断一定成立，因此返回 0，正如文档所说，返回 0 表示成功，否则表示超时。

dispatchsemaphoresignal

这个函数的实现相对来说比较简单，因为它不需要阻塞，只用唤醒。简化版源码如下:

```
long dispatch_semaphore_signal(dispatch_semaphore_t dsema) {
    long value = dispatch_atomic_inc2o(dsema, dsema_value);
    if (fastpath(value > 0)) {
        return 0;
    }
    return _dispatch_semaphore_signal_slow(dsema);
}
```

首先会调用原子方法让 value 加一，如果大于零就立刻返回 0，否则返回 `_dispatch_semaphore_signal_slow`:

```
long _dispatch_semaphore_signal_slow(dispatch_semaphore_t dse
    (void)dispatch_atomic_inc2o(dsema, dsema_sent_ksignals);
    _dispatch_semaphore_create_port(&dsema->dsema_port);
    kern_return_t kr = semaphore_signal(dsema->dsema_port);
    return 1;
}
```

它的作用仅仅是调用内核的 `semaphore_signal` 函数唤醒信号量，然后返回 1。这也符合文档中的描述:"如果唤醒了线程，返回非 0，否则返回 0"。

dispatch_group

有了上面的铺垫，group 是一个非常容易理解的概念，我们先看看如何创建 group:

```
dispatch_group_t dispatch_group_create(void) {
    dispatch_group_t dg = _dispatch_alloc(DISPATCH_VTABLE(gro
    _dispatch_semaphore_init(LONG_MAX, dg);
    return dg;
}
```

没错，group 就是一个 value 为 `LONG_MAX` 的信号量。

dispatchgroupasync

它仅仅是 `dispatch_group_async_f` 的封装:

```
void dispatch_group_async_f(dispatch_group_t dg, dispatch_que
    dispatch_continuation_t dc;
    dispatch_group_enter(dg);

    dc = _dispatch_continuation_alloc();
    dc->do_vtable = (void *)(DISPATCH_OBJ_ASYNC_BIT | DISPATC
    dc->dc_func = func;
    dc->dc_ctxt = ctxt;
    dc->dc_data = dg;
    _dispatch_queue_push(dq, dc);
}
```

这个函数和 `dispatch_async_f` 的实现高度一致，主要的不同在于调用了 `dispatch_group_enter` 方法:

```
void dispatch_group_enter(dispatch_group_t dg) {
    dispatch_semaphore_t dsema = (dispatch_semaphore_t)dg;
    (void)dispatch_semaphore_wait(dsema, DISPATCH_TIME_FOREVE
}
```

这个方法也没做什么，就是调用 `wait` 方法让信号量的 value 减一而已。

dispatchgroupwait

这个方法用于等待 group 中所有任务执行完成，可以理解为信号量 wait 的封装:

```
long dispatch_group_wait(dispatch_group_t dg, dispatch_time_t
    dispatch_semaphore_t dsema = (dispatch_semaphore_t)dg;
    if (dsema->dsema_value == dsema->dsema_orig) {
        return 0;
    }
    if (timeout == 0) {
        return KERN_OPERATION_TIMED_OUT;
    }
    return _dispatch_group_wait_slow(dsema, timeout);
}
```

如果当前 value 和原始 value 相同，表明任务已经全部完成，直接返回 0，如果 timeout 为 0 也会立刻返回，否则调用 `_dispatch_group_wait_slow`。这个方法等等待部分和 `_dispatch_semaphore_signal_slow` 几乎一致，区别在于等待结束后它不是 return，而是调用 `_dispatch_group_wake` 去唤醒这个 group。

```
static long _dispatch_group_wait_slow(dispatch_semaphore_t ds
again:
    _dispatch_group_wake(dsema);
    switch (timeout) {/* 三种情况分类 */}
    goto again;
}
```

这里我们暂时跳过 `_dispatch_group_wake`，后面会有详细分析。只要知道这个函数在 group 中所有事件执行完后会被调用即可。

dispatchgroupnotify

老习惯，这个函数仅仅是封装了 `dispatch_group_notify_f`:

```
void dispatch_group_notify_f(dispatch_group_t dg, dispatch_qu
    dispatch_semaphore_t dsema = (dispatch_semaphore_t)dg;
    struct dispatch_sema_notify_s *dsn, *prev;

    dsn->dsn_queue = dq;
    dsn->dsn_ctxt = ctxt;
    dsn->dsn_func = func;
    prev = dispatch_atomic_xchg2o(dsema, dsema_notify_tail, d
    if (fastpath(prev)) {
        prev->dsn_next = dsn;
    } else {/* ... */}
}
```

这种结构的代码我们已经遇到多次了，它其实就是在链表的尾部续上新的元素。所以 notify 方法并没有做过多的处理，只是是用链表把所有回调通知保存起来，等待调用。

dispatchgroupleave

在介绍 `dispatch_async` 函数时，我们看到任务在被执行时，还会调用 `dispatch_group_leave` 函数:

```
void dispatch_group_leave(dispatch_group_t dg) {
    dispatch_semaphore_t dsema = (dispatch_semaphore_t)dg;
    long value = dispatch_atomic_inc2o(dsema, dsema_value);
    if (slowpath(value == dsema->dsema_orig)) {
        (void)_dispatch_group_wake(dsema);
    }
}
```

当 group 的 value 变为初始值时，表示所有任务都已执行完，开始调用 `_dispatch_group_wake` 处理回调。

dispatchgroup_wake

```
static long _dispatch_group_wake(dispatch_semaphore_t dsema)
    struct dispatch_sema_notify_s *next, *head, *tail = NULL;
    long rval;
    head = dispatch_atomic_xchg2o(dsema, dsema_notify_head, N

    if (head) {
        tail = dispatch_atomic_xchg2o(dsema, dsema_notify_tai
    }
    rval = dispatch_atomic_xchg2o(dsema, dsema_group_waiters,
    if (rval) {
        _dispatch_semaphore_create_port(&dsema->dsema_waiter_
        do {
            kern_return_t kr = semaphore_signal(dsema->dsema_
        } while (--rval);
    }
    if (head) {
        // async group notify blocks
        do {
            dispatch_async_f(head->dsn_queue, head->dsn_ctxt,
            next = fastpath(head->dsn_next);
            if (!next && head != tail) {
                while (!(next = fastpath(head->dsn_next))) {
                    _dispatch_hardware_pause();
                }
            }
            free(head);
        } while ((head = next));
    }
    return 0;
}
```

这个函数主要分为两部分，首先循环调用 `semaphore_signal` 告知唤醒当初等待 group 的信号量，因此 `dispatch_group_wait` 函数得以返回。

然后获取链表，依次调用 `dispatch_async_f` 异步执行在 notify 函数中注册的回调。

dispatch_once

`dispatch_once` 仅仅是一个包装，内部直接调用了 `dispatch_once_f`：

```
void dispatch_once_f(dispatch_once_t *val, void *ctxt, dispat
    struct _dispatch_once_waiter_s * volatile *vval = (struct
    struct _dispatch_once_waiter_s dow = { NULL, 0 };
    struct _dispatch_once_waiter_s *tail, *tmp;
```

```
    _dispatch_thread_semaphore_t sema;

    if (dispatch_atomic_cmpxchg(vval, NULL, &dow)) {
        _dispatch_client_callout(ctxt, func);
        tmp = dispatch_atomic_xchg(vval, DISPATCH_ONCE_DONE);
        tail = &dow;
        while (tail != tmp) {
            while (!tmp->dow_next) {
                _dispatch_hardware_pause();
            }
            sema = tmp->dow_sema;
            tmp = (struct _dispatch_once_waiter_s*)tmp->dow_n
            _dispatch_thread_semaphore_signal(sema);
        }
    } else {
        dow.dow_sema = _dispatch_get_thread_semaphore();
        for (;;) {
            tmp = *vval;
            if (tmp == DISPATCH_ONCE_DONE) {
                break;
            }
            dispatch_atomic_store_barrier();
            if (dispatch_atomic_cmpxchg(vval, tmp, &dow)) {
                dow.dow_next = tmp;
                _dispatch_thread_semaphore_wait(dow.dow_sema)
            }
        }
        _dispatch_put_thread_semaphore(dow.dow_sema);
    }
}
```

这段代码比较长，我们考虑三个场景:

1. 第一次调用: 此时外部传进来的 onceToken 还是空指针，所以 vval 为 NULL，if 判断成立。首先执行 block，然后让将 vval 的值设为 `DISPATCH_ONCE_DONE` 表示任务已经完成，同时用 tmp 保存先前的 vval。此时，dow 也为空，因此 while 判断不成立，代码执行结束。
2. 同一线程第二次调用: 由于 vval 已经变成了 `DISPATCH_ONCE_DONE`，因此 if 判断不成立，进入 else 分支的 for 循环。由于 tmp 就是 `DISPATCH_ONCE_DONE`，所以循环退出，没有做任何事。
3. 多个线程同时调用: 由于 if 判断中是一个原子性操作，所以必然只有一个线程能进入 if 分支，其他的进入 else 分支。由于其他线程在调用函数时，vval 还不是 `DISPATCH_ONCE_DONE`，所以进入到 for

循环的后半部分。这里构造了一个链表，链表的每个节点上都调用了信号量的 `wait` 方法并阻塞，而在 if 分支中，则会依次遍历所有的节点并调用 `signal` 方法，唤醒所有等待中的信号量。

dispatchbarrierasync

它调用了 `dispatch_barrier_async_f` 函数，实现原理也和 `dispatch_async_f` 类似:

```
void dispatch_barrier_async_f(dispatch_queue_t dq, void *ctxt
    dispatch_continuation_t dc;
    dc = fastpath(_dispatch_continuation_alloc_cacheonly());
    dc->do_vtable = (void *)(DISPATCH_OBJ_ASYNC_BIT | DISPATC
    dc->dc_func = func;
    dc->dc_ctxt = ctxt;
    _dispatch_queue_push(dq, dc);
}
```

区别在于 `do_vtable` 被设置了两个标志位，多了一个 `DISPATCH_OBJ_BARRIER_BIT` 标记。这个标记在从队列中取出任务时被用到:

```
static _dispatch_thread_semaphore_t _dispatch_queue_drain(dis
    while (dq->dq_items_tail) {
        /* ... */
        if (!DISPATCH_OBJ_IS_VTABLE(dc) && (long)dc->do_vtabl
            if (dq->dq_running > 1) {
                goto out;
            }
        } else {
            _dispatch_continuation_redirect(dq, dc);
            continue;
        }
    }
out:
    /* 不完整的 drain，需要清理现场 */
    return sema; // 返回空的信号量
}
```

这里原来是一个循环，会拿出所有的任务，依次调用 `_dispatch_continuation_redirect`，最终并行处理。一旦遇到 `DISPATCH_OBJ_BARRIER_BIT` 这个标记，就会终止循环。

在 `out` 标签后面，返回了一个空的信号量，随后方法的调用者会把它单独放入队列，等待下一次执行:

```
void _dispatch_queue_invoke(dispatch_queue_t dq) {
    _dispatch_thread_semaphore_t sema = _dispatch_queue_drain
    if (sema) {
        _dispatch_thread_semaphore_signal(sema);
    } else if (tq) {
        return _dispatch_queue_push(tq, dq);
    }
}
```

因此 barrier 方法能等待此前所有任务执行完以后执行 `_dispatch_queue_push`，同时保证自己执行完以后才执行后续的操作。

dispatch_source

source 是一种资源，类似于 生产者/消费者模式中的生产者，而队列则是消费者。当有新的资源(source) 产生时，他们被放到对应的队列上被执行(消费)。

`dispatch_source` 最常见的用途之一就是用来实现定时器，举一个小例子:

```
dispatch_source_t timer = dispatch_source_create(DISPATCH_SOU
dispatch_source_set_timer(timer, dispatch_walltime(NULL, 0),
dispatch_source_set_event_handler(timer, ^{
    // 定时器触发时执行的 block
});
dispatch_resume(timer);
```

使用 GCD Timer 的好处在于不依赖 runloop，因此任何线程都可以使用。由于使用了 block，不会忘记避免循环引用。此外，定时器可以自由控制精度，随时修改间隔时间等。

dispatchsourcecreate

下面从底层源码的角度来研究这几行代码的作用。首先是 `dispatch_source_create` 函数，它和之前见到的 create 函数都差不多，对 dispatch*source*t 对象做了一些初始化工作:

```
dispatch_source_t ds = NULL;
ds = _dispatch_alloc(DISPATCH_VTABLE(source), sizeof(struct d
_dispatch_queue_init((dispatch_queue_t)ds);
ds->do_suspend_cnt = DISPATCH_OBJECT_SUSPEND_INTERVAL;
ds->do_targetq = &_dispatch_mgr_q;
dispatch_set_target_queue(ds, q);
return ds;
```

这里涉及到两个队列，其中 q 是用户指定的队列，表示事件触发的回调在哪个队列执行。而 `_dispatch_mgr_q` 则表示由哪个队列来管理这个 source，mgr 是 manager 的缩写，也是上文提到的序列号为 2 的内部队列。

dispatchsourceset_timer

在这个函数中，首先会有参数处理，过滤掉不符合要求的参数。随后创建了 `dispatch_set_timer_params` 类型的指针 params:

```
struct dispatch_set_timer_params {
    dispatch_source_t ds;
    uintptr_t ident;
    struct dispatch_timer_source_s values;
};
```

这个 params 负责绑定定时器对象与他的参数(存储在 `valus` 属性中)，最后调用:

```
dispatch_barrier_async_f((dispatch_queue_t)ds, params, _dispa
```

这里是把 source 当做队列来使用，因此实际上是调用了 `_dispatch_source_set_timer2(params)` 方法:

```
static void _dispatch_source_set_timer2(void *context) {
    // Called on the source queue
    struct dispatch_set_timer_params *params = context;
    dispatch_suspend(params->ds);
    dispatch_barrier_async_f(&_dispatch_mgr_q, params,
            _dispatch_source_set_timer3);
}
```

这里首先暂停了队列，避免了修改的过程中定时器被触发。然后在 manager 队列上执行 `_dispatch_source_set_timer3(params)`:

```
static void _dispatch_source_set_timer3(void *context) {
    struct dispatch_set_timer_params *params = context;
    dispatch_source_t ds = params->ds;
    // ...
    _dispatch_timer_list_update(ds);
    dispatch_resume(ds);
}
```

`_dispatch_timer_list_update` 函数的作用是根据下一次触发时间将 timer 排序。

接下来，当初分发到 manager 队列的 block 将要被执行，走到 `_dispatch_mgr_invoke` 函数，其中有如下代码:

```
timeoutp = _dispatch_get_next_timer_fire(&timeout);
r = select(FD_SETSIZE, &tmp_rfds, &tmp_wfds, NULL, sel_timeou
```

可见 GCD 的定时器是由系统的 select 方法实现的。

当内层的 manager 队列被唤醒后，还会进一步唤醒外层的队列(当初用户指定的那个)，并在队列上执行 timer 触发时的 block。

dispatch_resume/suspend

GCD 对象的暂停和恢复由 `do_suspend_cnt` 决定，暂停时通过原子操作将改属性的值加 2，对应的在恢复时通过原子操作将该属性减二。

它有两个默认值:

```
#define DISPATCH_OBJECT_SUSPEND_LOCK        1u
#define DISPATCH_OBJECT_SUSPEND_INTERVAL    2u
```

在唤醒队列时有如下代码:

```
void _dispatch_queue_invoke(dispatch_queue_t dq) {
    if (!dispatch_atomic_sub2o(dq, do_suspend_cnt, DISPATCH_O
        if (dq->dq_running == 0) {
            _dispatch_wakeup(dq); // verify that the queue is
        }
    }
}
```

可见能够唤醒队列的前提是 `dq->do_suspend_cnt - 1 = 0`，也就是要求 `do_suspend_cnt` 的值就是 `DISPATCH_OBJECT_SUSPEND_LOCK`。

观察 8 个全局队列和主队列的定义就会发现，他们的 `do_suspend_cnt` 值确实为 `DISPATCH_OBJECT_SUSPEND_LOCK`，因此默认处于启动状态。

而 `dispatch_source` 的 create 方法中，`do_suspend_cnt` 的初始值为 `DISPATCH_OBJECT_SUSPEND_INTERVAL`，因此默认处于暂停状态，需要手动开启。

dispatch_after

`dispatch_after` 其实依赖于定时器的实现，函数内部调用了 `dispatch_after_f`:

```
void dispatch_after_f(dispatch_time_t when, dispatch_queue_t
    uint64_t delta;
    struct _dispatch_after_time_s *datc = NULL;
    dispatch_source_t ds;

    // 如果延迟为 0，直接调用 dispatch_async
    delta = _dispatch_timeout(when);
    if (delta == 0) {
        return dispatch_async_f(queue, ctxt, func);
    }

    ds = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0
    dispatch_assert(ds);

    datc = malloc(sizeof(*datc));
    dispatch_assert(datc);
    datc->datc_ctxt = ctxt;
    datc->datc_func = func;
    datc->ds = ds;

    dispatch_set_context(ds, datc);
    dispatch_source_set_event_handler_f(ds, _dispatch_after_t
    dispatch_source_set_timer(ds, when, DISPATCH_TIME_FOREVER
    dispatch_resume(ds);
}
```

首先将延迟执行的 block 封装在 `_dispatch_after_time_s` 这个结构体中，并且作为上下文，与 timer 绑定，然后启动 timer。

到时以后，执行 `_dispatch_after_timer_callback` 回调，并取出上下文中的 block:

```
static void _dispatch_after_timer_callback(void *ctxt) {
    struct _dispatch_after_time_s *datc = ctxt;
    _dispatch_client_callout(datc->datc_ctxt, datc->datc_func
    // 清理工作
}
```

总结

本文主要整理了 GCD 中常见的 API 以及底层的实现原理。对于队列来说，需要理解它的数据结构，转发机制，以及底层的线程池模型。

`dispatch_async` 会把任务添加到队列的一个链表中，添加完后会唤醒队列，根据 vtable 中的函数指针，调用 wakeup 方法。在 wakeup 方法中，从线程池里取出工作线程(如果没有就新建)，然后在工作线程中取出链表头部指向的 block 并执行。

`dispatch_sync` 的实现略简单一些，它不涉及线程池(因此一般都在当前线程执行)，而是利用与线程绑定的信号量来实现串行。

分发到不同队列时，代码进入的分支也不一样，比如 `dispatch_async` 到主队列的任务由 runloop 处理，而分发到其他队列的任务由线程池处理。

在当前串行队列中执行 `dispatch_sync` 时，由于 `dq_running` 属性(表示在运行的任务数量) 为 1，所以以下判断成立:

```
if (slowpath(!dispatch_atomic_cmpxchg2o(dq, dq_running, 0, 1)
    return _dispatch_barrier_sync_f_slow(dq, ctxt, func);
}
```

在 `_dispatch_barrier_sync_f_slow` 函数中使用了线程对应的信号量并且调用 `wait` 方法，从而导致线程死锁。

如果向其它队列同步提交 block，最终进入 `_dispatch_barrier_sync_f_invoke`，它只是保证了 block 执行的原子性，但没有使用线程对应的信号量。

对于信号量来说，它主要使用 `signal` 和 `wait` 这两个接口，底层分别调用了内核提供的方法。在调用 `signal` 方法后，先将 value 减一，如果大于零立刻返回，否则陷入等待。`signal` 方法将信号量加一，如果 value 大于零立刻返回，否则说明唤醒了某一个等待线程，此时由系统决定哪个线程的等待方法可以返回。

`dispatch_group` 的本质就是一个 value 非常大的信号量，等待 group 完成实际上就是等待 value 恢复初始值。而 notify 的作用是将所有注册的回调组装成一个链表，在 `dispatch_async` 完成时判断 value 是不是恢复初始值，如果是则调用 `dispatch_async` 异步执行所有注册的回调。

`dispatch_once` 通过一个静态变量来标记 block 是否已被执行，同时使用信号量确保只有一个线程能执行，执行完 block 后会唤醒其他所有等待的线程。

`dispatch_barrier_async` 改变了 block 的 `vtable` 标记位，当它将要被取出执行时，会等待前面的 block 都执行完，然后在下一次循环中被执行。

`dispatch_source` 可以用来实现定时器。所有的 source 会被提交到用户指定的队列，然后提交到 manager 队列中，按照触发时间排好序。随后找到最近触发的定时器，调用内核的 `select` 方法等待。等待结束后，依次唤醒 manager 队列和用户指定队列，最终触发一开始设置的回调 block。

GCD 中的对象用 `do_suspend_cnt` 来表示是否暂停。队列默认处于启动状态，而 `dispatch_source` 需要手动启动。

`dispatch_after` 函数依赖于 `dispatch_source` 定时器，它只是注册了一个定时器，然后在回调函数中执行 block。

参考资料

1. Why do we use _*builtin*expect when a straightforward way is to use if-else
2. Posix线程编程指南(2) 线程私有数据
3. 选择 GCD 还是 NSTimer?
4. 从NSTimer的失效性谈起（二）：关于GCD Timer和libdispatch
5. 变态的libDispatch源码分析
