面试题库之GCD

- GCD：GCD使用经验与技巧浅谈
  - dispatch*once*t必须是全局或static变量

    ```
     dispatch_once(&onceToken, ^{
        //单例代码
    });
    ```

  - dispatch*queue*create 创建队列用的，参数只有2个：

    `dispatch_queue_t dispatch_queue_create ( const char *label, dispatch_queue_attr_t attr );`

    普遍教程是这么创建队列的：

    `dispatch_queue_t queue = dispatch_queue_create("com.example.MyQueue", NULL);`

    即第2个参数为NULL，但是dispatch*queue*attr_t类型是有已经创建好的常量的，故为了清晰严谨，最好创建队列如下：

    ```
    dispatch_queue_t queue = dispatch_queue_create("com.example.MyQueue", DISPATCH_QUEUE_SERIA
    L);
    //并行队列
    dispatch_queue_t queue = dispatch_queue_create("com.example.MyQueue", DISPATCH_QUEUE_CONCU
    RRENT);```
    ```

  - `dispatch_after`：延迟提交，不是延迟执行。 即将一个Block在特定的延时以后，加入到指定的队列中，不是在特定的时间后立即运行！。

    示例：

    ```
    //创建串行队列
    dispatch_queue_t queue = dispatch_queue_create("me.tutuge.test.gcd", DISPATCH_QUEUE_SERIAL
    );
    //立即打印一条信息
    NSLog(@"Begin add block...");
    //提交一个block
    dispatch_async(queue, ^{
    //Sleep 10秒
    [NSThread sleepForTimeInterval:10];
    NSLog(@"First block done...");
    });
    //5 秒以后提交block
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(5 * NSEC_PER_SEC)), queue, ^{
    NSLog(@"After...");
    });
    ```

    运行结果：

```
2015-03-31 20:57:27.122 GCDTest[45633:1812016] Begin add block...
2015-03-31 20:57:37.127 GCDTest[45633:1812041] First block done...
2015-03-31 20:57:37.127 GCDTest[45633:1812041] After...
```

- `dispatch_time_t` 用来创建正确合适的时间。

```
dispatch_time_t dispatch_time ( dispatch_time_t when, int64_t delta );
```

第1个参数一般是 `DISPATCH_TIME_NOW`，表示从现在开始.

第2个参数就是真正的延时时间： `delta是纳秒`

```
NSEC_PER_SEC，每秒有多少纳秒。
USEC_PER_SEC，每秒有多少毫秒。（注意是指在纳秒的基础上）
NSEC_PER_USEC，每毫秒有多少纳秒。
```

故延迟1s的写法：

```
dispatch_time(DISPATCH_TIME_NOW, 1 * NSEC_PER_SEC);
dispatch_time(DISPATCH_TIME_NOW, 1000 * USEC_PER_SEC);
dispatch_time(DISPATCH_TIME_NOW, USEC_PER_SEC * NSEC_PER_USEC);
```

- dispatch_suspend != 立即停止队列的运行

`dispatch_suspend`， `dispatch_resume` 提供了 `挂起、恢复` 队列的功能，简单来说，就是可以暂停、恢复队列上的任务。但是这里的“挂起”，并不能保证可以立即停止队列上正在运行的block.

```
dispatch_queue_t queue = dispatch_queue_create("me.tutuge.test.gcd", DISPATCH_QUEUE_SERIAL
);
//提交第一个block，延时5秒打印。
dispatch_async(queue, ^{
[NSThread sleepForTimeInterval:5];
NSLog(@"After 5 seconds...");
});
//提交第二个block，也是延时5秒打印
dispatch_async(queue, ^{
[NSThread sleepForTimeInterval:5];
NSLog(@"After 5 seconds again...");
});
//延时一秒
NSLog(@"sleep 1 second...");
[NSThread sleepForTimeInterval:1];
//挂起队列
NSLog(@"suspend...");
dispatch_suspend(queue);
//延时10秒
NSLog(@"sleep 10 second...");
[NSThread sleepForTimeInterval:10];
//恢复队列
NSLog(@"resume...");
dispatch_resume(queue);
```

运行结果：

```
2015-04-01 00:32:09.903 GCDTest[47201:1883834] sleep 1 second...
2015-04-01 00:32:10.910 GCDTest[47201:1883834] suspend...
2015-04-01 00:32:10.910 GCDTest[47201:1883834] sleep 10 second...
2015-04-01 00:32:14.908 GCDTest[47201:1883856] After 5 seconds...
2015-04-01 00:32:20.911 GCDTest[47201:1883834] resume...
2015-04-01 00:32:25.912 GCDTest[47201:1883856] After 5 seconds again...
（在dispatch_suspend挂起队列后，第一个block还是在运行，并且正常输出）
```

可见 `dispatch_suspend` 并不会立即暂停正在运行的block，而是在当前block执行完成后，暂停后续的block执行。

- “同步”的dispatch_apply:在一个队列（串行或并行）上“运行”多次block，其实就是简化了用循环去向队列依次添加block任务。

```
//创建异步串行队列
dispatch_queue_t queue = dispatch_queue_create("me.tutuge.test.gcd", DISPATCH_QUEUE_SERIAL
);
//运行block3次
dispatch_apply(3, queue, ^(size_t i) {
 NSLog(@"apply loop: %zu", i);
 });
 //打印信息
 NSLog(@"After apply");
```

运行结果：

```
2015-04-01 00:55:40.854 GCDTest[47402:1893289] apply loop: 0
2015-04-01 00:55:40.856 GCDTest[47402:1893289] apply loop: 1
2015-04-01 00:55:40.856 GCDTest[47402:1893289] apply loop: 2
2015-04-01 00:55:40.856 GCDTest[47402:1893289] After apply
```

分析：明明是提交到异步的队列去运行，但是“After apply”居然在apply后打印，也就是说，dispatch*apply将外面的线程（main线程）“阻塞”了！查看官方文档，dispatch*apply确实会“等待”其所有的循环运行完毕才往下执行=。=，看来要小心使用了。

- 避免死锁（涉及到多线程时，不可避免的就会有死锁问题）

1. dispatch_sync导致的死锁：在main线程使用 `同步` 方式提交block，必会死锁.
   故 **`建议少用 dispatch_sync 来避免死锁!`**

   ```
   dispatch_sync(dispatch_get_main_queue(), ^{
    NSLog(@"I am block...");
   });
   ```

2. dispatch*apply导致的死锁： **dispatch*apply的嵌套调用会导致死锁，故要避免！**

- 灵活使用dispatch_group

等待一系列任务（block）执行完成，然后再做一些收尾的工作。 如果是有序的任务，可以分步骤完成的，直接使用串行队列就行。 但是如果是一系列并行执行的任务呢？这个时候，就需要dispatch_group帮忙了。

```
dispatch_group使用步骤：
    1. 创建dispatch_group_t
    2. 添加任务（block）
    3. 添加结束任务（如清理操作、通知UI等）
```

第2步添加任务可以分以下2种情况：

1. 自己创建队列：使用dispatch*group*async。 用dispatch*group*async函数，简单有效：

```
//省去创建group、queue代码。。
dispatch_group_async(group, queue, ^{
 //Do you work...
 });
```

2. 无法直接使用队列变量时，就无法使用dispatch*group*async了，下面以使用AFNetworking时的情况。

   使用 `dispatch_group_enter` ， `dispatch_group_leave` 就可以方便的将一系列网络请求“打包”起来

```
AFHTTPRequestOperationManager *manager = [AFHTTPRequestOperationManager manager];
//Enter group
dispatch_group_enter(group);
[manager GET:@"http://www.baidu.com" parameters:nil success:^(AFHTTPRequestOperation *
operation, id responseObject) {
//Deal with result...
//Leave group
dispatch_group_leave(group);
}
failure:^(AFHTTPRequestOperation *operation, NSError *error) {
//Deal with error...
//Leave group
dispatch_group_leave(group);
}];
//More request...
```

第3步添加结束任务也可以分以下2种情况：

1. 在当前线程阻塞的同步等待：dispatch*group*wait。等待group关联的block执行完毕，也可以设置超时参数。dispatch*group*wait(group, DISPATCH*TIME*FOREVER);
2. 添加一个异步执行的任务作为结束任务：dispatch*group*notify。为group设置通知一个block，当group关联的block执行完毕后，就调用这个block。类似dispatch*barrier*async。

- 使用 `dispatch_barrier_async` , `dispatch_barrier_sync` 的注意事项:

`dispatch_barrier_async` 向某个队列插入一个block，当目前正在执行的block完成后，阻塞这个block后面添加的blocks，只运行该block直至完成，再继续后续的blocks。

注意事项：

a. dispatchbarrier(a)sync只在自己创建的并发队列上有效，在全局(Global)并发队列、串行队列上，效果跟dispatch_(a)sync效果一样。

b. 既然在串行队列上跟dispatch_(a)sync效果一样，那就要小心别死锁

- 使用dispatch_semaphore信号量来处理多线程并发控制。

在GCD中有2个函数是关于dispatch_semaphore的操作的：

```
dispatch_semaphore_create 创建一个semaphore
dispatch_semaphore_signal 发送一个信号，执行完成，释放一个执行并发数目，+1
dispatch_semaphore_wait 等待信号,有空闲则执行，加入队列。消耗一个并发条目。-1
```

实例：

```
dispatch_group_t group = dispatch_group_create();
dispatch_semaphore_t semaphore = dispatch_semaphore_create(10);
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
for (int i = 0; i < 100; i++)
{
dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
dispatch_group_async(group, queue, ^{
NSLog(@"%i",i);
sleep(2);
dispatch_semaphore_signal(semaphore);
});
}
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
dispatch_release(group);
dispatch_release(semaphore);
```

解析：

创建了一个初使值为10的semaphore，每一次for循环都会创建一个新的线程，线程结束的时候会发送一个信号，线程创建之前会信号等待，所以当同时创建了10个线程之后，for循环就会阻塞，等待有线程结束之后会增加一个信号才继续执行，如此就形成了对并发的控制，如上就是一个并发数为10的一个线程队列。