章节四 多线程工具之NSOperationQueue

NSOperationQueue简介

使用NSOperationQueue方式进行多线程编程，不能够像NSThread一样直接管理线程，也不需要管理，但是可以间接的干预线程，这也是该方式的优点。同时引入了Queue（队列）的概念，了解NSOperationQueue使用，首先要了解NSOperation和NSOperationQueue的关系。

一 NSOperationQueue的使用

1.1 NSOperation

NSOperation类是一个抽象类，用来封装单任务的代码和数据。可以参考其英文解释:

> *The NSOperation class is an abstract class you use to encapsulate the code and data associated with a single task.*

因为是抽象类，所以不能直接使用该类，而是创建子类或者一些系统定义的子类（NSInvocationOperation 或者 NSBlockOperation）来完成实际的任务。

> 注意：使用NSOperation的子类对象只能执行任务一次，而且不能再次执行它。你可以将它添加到一个操作队列中执行操作，用NSOperationQueue的实例来完成它。

1.1.1 创建NSOperation子类

示例代码如下:

```
CNOperation *op = [[CNOperation alloc] init];
[op start];



#import "CNOperation.h"

@implementation CNOperation

-(void)main{

    NSLog(@"do something");
}

@end
```

尝试加入断点，或者打印输出，我们可以看到main方法会在主线程中执行。

1.1.2 系统自带的NSOperation子类

- NSInvocationOperation

  示例代码如下：

```
  NSInvocationOperation *op1 = [[NSInvocationOperation alloc]
initWithTarget:self selector:@selector(invoAction) object:nil];

 [op1 start];


  -(void)invoAction{
      NSLog(@"do something");
  }
```

- NSBlockOperation

  示例代码如下：

```
  NSBlockOperation *op2 = [NSBlockOperation blockOperationWithBlock:^{
     NSLog(@"do something");
  }];

  [op2 start];
```

小结：NSOperation默认并不会执行，必须调用start方法，执行时默认在当前线程中执行，即默认同步执行。

1.2 NSOperationQueue

一个NSOperationQueue对象并非一个线程，而是线程管理器，可以帮我们自动创建新的线程。取决于队列的最大并行数。NSOperation对象添加到队列中，默认就成为了异步执行（非当前线程）。

1.2.1 创建NSOperationQueue

代码示例：

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
```

1.2.2 添加NSOperation到队列中

- 添加到Queue中的NSOperation会自动执行，执行的时间取决于两个因素，后文介绍。

代码示例：

```
[queue addOperation:operation];
```

- 可以同时添加多个Operation到Queue中

代码示例：

```
[queue addOperations:operations waitUntilFinished:NO];
```

- 可以隐式的添加一个Operation到Queue中

代码示例：

```
[queue addOperationWithBlock:^{

}];
```

1.2.3 添加NSOperation的依赖对象

NSOperation添加到queue之后,通常短时间内就会得到运行。但是如果存在依赖,或者整个queue被暂停、优先级较低等原因,也可能需要等待。

> 注意：NSOperation添加到queue之后,绝对不要再修改NSOperation对象的状态。因为NSOperation对象可能会在任何时候运行,因此改变NSOperation对象的依赖或数据会产生不利的影响。你只能查看NSOperation对象的状态，比如是否正在运行、等待运行、已经完成等。

- 添加依赖

  添加依赖成功的前提，必须将Operation添加到Queue中。

  依赖关系不局限于相同queue中的NSOperation对象,NSOperation对象会管理自己的依赖, 因此完全可以在不同的queue之间的NSOperation对象创建依赖关系。

  代码示例：

  ```
  [op2 addDependency:op1];
  ```

  注意：不能创建环形依赖，比如A依赖B，B依赖A，这是错误的

依赖关系会影响到NSOperation对象在queue中的执行顺序

代码示例：

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];

NSBlockOperation *operation1 = [NSBlockOperation blockOperationWithBlock:^()
{
    NSLog(@"执行第1次操作，线程：%@", [NSThread currentThread]);
}];

NSBlockOperation *operation2 = [NSBlockOperation blockOperationWithBlock:^()
{
    NSLog(@"执行第2次操作，线程：%@", [NSThread currentThread]);
}];

[queue addOperation:operation1];
[queue addOperation:operation2];
```

可以看出，默认是按照添加顺序执行的，先执行operation1，再执行operation2

添加下列代码：

```
[operation1 addDependency:operation2];
```

结果是先执行operation2，再执行operation1

1.2.3 Operations执行顺序的影响因素

1. 依赖关系
2. 优先级

首先根据依赖关系，确定前后执行顺序，其次在满足依赖关系的前提下，再根据优先级排序。

设置优先级：

iOS8以前属性：

```
@property double threadPriority NS_DEPRECATED(10_6, 10_10, 4_0, 8_0);
```

iOS8以后属性：

```
@property NSQualityOfService qualityOfService NS_AVAILABLE(10_10, 8_0);
```

该属性类型我们在NSThread也见到过。

1.2 NSOperationQueue

上文中提到：一个NSOperationQueue对象并非一个线程，而是线程管理器，可以帮我们自动创建新的线程。取决于队列的最大并行数。

1.2.1 设置队列的最大并发数

```
queue.maxConcurrentOperationCount = 1;
```

虽然NSOperationQueue类设计用于并发执行Operations,你也可以强制单个queue一次只能执行一个Operation。setMaxConcurrentOperationCount:方法可以配置queue的最大并发操作数量。设为1就表示queue每次只能执行一个操作。不过operation执行的顺序仍然依赖于其它因素,比如operation是否准备好和operation的优先级等。因此串行化的operation queue并不等同于后文GCD中的串行dispatch queue。

1.2.2 取消Operations

一旦添加到operation queue,queue就拥有了这个Operation对象并且不能被删除,唯一能做的事情是取消。你可以调用Operation对象的cancel方法取消单个操作,也可以调用operation queue的cancelAllOperations方法取消当前queue中的所有操作。

```
// 取消单个操作
[operation cancel];

// 取消queue中所有的操作
[queue cancelAllOperations];
```

1.2.3 等待operation完成

等待某个Operation完成，阻塞当前线程：

```
// 会阻塞当前线程，等到某个operation执行完毕
[operation waitUntilFinished];
```

也可以同时等待一个queue中的所有操作：

```
// 阻塞当前线程，等待queue的所有操作执行完毕
[queue waitUntilAllOperationsAreFinished];
```

1.2.4 暂停和继续queue

暂停一个queue不会导致正在执行的operation在任务中途暂停,只是简单地阻止调度新

Operation执行。

```
// 暂停queue
[queue setSuspended:YES];

// 继续queue
[queue setSuspended:NO];
```

章节五 多线程工具之GCD

一 GCD简介

前文中一直提到过一个概念——GCD，GCD为我们编写代码提供了非常绝佳的体验。GCD是苹果公司为在多核心设备上实现多线程充分利用多核优势提供的解决方案，在很多技术中，如RunLoop，GCD都扮演了重要的角色。GCD全称Grand Central Dispatch，直译为调度中心，完全基于C语言编写。使用GCD，不需要管理线程，线程管理完全托管给GCD。把线程想象成的火车，我们只需要提交我们的运送任务，GCD会帮我们在合适的火车上运行任务。多线程编程变得如此简单，但对于初学者来讲，GCD学习会有一定难度。因为是纯C编写，所以没有对象的概念，学习GCD你可以忘掉封装继承多态等等概念。

二 dispatch queue

- GCD中的一个重要组成部分就是队列，我们把各种任务提交给队列，队列根据它本身的类型以及当前系统的状态，添加到不同的线程中执行任务。线程的创建和管理都有GCD本身完成，不需要我们参与。
- 系统提供了很多定义好的队列：只管理主线程的main_queue,全局并行的globle_queue。同时，我们也可以自定义自己的queue。

2.1 queue的种类

- 并行queue： 可以多个线程并行，任务可以同时执行
- 串行queue： 所有线程串行，或者只有一个线程，任务依次执行

2.2 创建自定义的queue

queue的类型为 `dispatch_queue_t`

创建函数是dispatch_queue_create()

```
//创建     串行的 队列池
    dispatch_queue_t q_1 =  dispatch_queue_create("task3.queue.1",
DISPATCH_QUEUE_SERIAL);
//创建     并行的 队列池
    dispatch_queue_t q_2 =  dispatch_queue_create("task3.queue.2",
DISPATCH_QUEUE_CONCURRENT);
```

第一个参数代表queue的名字，注意是char型指针，并非NSString；第二个参数表示queue的类型，DISPATCH_QUEUE_SERIAL表示串行queue，DISPATCH_QUEUE_CONCURRENT表示并行queue。

2.2 获得预定义queue

除了自己定义queue以外，我们可以使用系统预定义的queue，包括两种。

2.2.1 获取全局并发queue

获取全局的一个queue，该队列的种类是并行queue，也就是说我们可以直接提交给这个queue，任务会在非主线程的其他线程执行。

```
dispatch_queue_t  queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
```

第一个参数表示queue的优先级，有四中优先级：

```
#define DISPATCH_QUEUE_PRIORITY_HIGH 2
#define DISPATCH_QUEUE_PRIORITY_DEFAULT 0
#define DISPATCH_QUEUE_PRIORITY_LOW (-2)
#define DISPATCH_QUEUE_PRIORITY_BACKGROUND INT16_MIN
```

第二个参数暂时没用

2.2.2 获取管理主线程的queue

main queue的种类属于串行队列，提交给该queue任务，会加入到主线程中

```
dispatch_queue_t  queue = dispatch_get_main_queue();
```

提交给main queue的任务可能不会立马被执行，而是在主线程的Run Loop检测到有dispatch提交过来的任务时才会执行，具体见NSRunLoop。

2.3 queue的内存管理

ARC环境下，不需要手动编写代码。 非ARC环境下需要使用 dispatch_retain()和

dispatch_release() 管理queue的引用计数。原则如同NSObject对象，计数为0时销毁queue

```
dispatch_queue_t q_1 =  dispatch_queue_create("task3.queue.1",
DISPATCH_QUEUE_SERIAL);

dispatch_release(q_1);
```

三 提交任务

提交任务分为两种方式，异步和同步。同步提交任务，会等待任务完成。异步不需要等待任务完成。

3.1 同步提交

同步提交示例：

```
dispatch_sync(q_1, ^{
    NSLog(@"do something");
});

NSLog(@"OK");
```

同步提交函数为dispatch_sync()，两个参数：

- 第一个参数表示提交到的queue
- 第二个参数表示任务详情，这里用block的方式描述一个任务，原因很简单，Block也是纯C实现的，而平时使用的Invocation或者target+selector方式都是面向对象的。

同步提交任务后，先执行完block，然后dispatch_sync()返回

3.2 异步提交

异步提交示例：

```
dispatch_async(q_1, ^{
    NSLog(@"do something");
});

NSLog(@"OK");
```

异步提交函数为dispatch_async()，两个参数：

- 第一个参数表示提交到的queue
- 第二个参数表示任务详情

异步提交任务后，dispatch_async()函数直接返回，无需等待block执行结束。

如果使用同步请求方式，会造成卡死当前线程的后果。

3.3 同步提交多次任务

同时提交多个任务给queue的函数很简单，首先我们先要确定任务数量

```
dispatch_queue_t q_1 =  dispatch_queue_create("task3.queue.1",
DISPATCH_QUEUE_SERIAL);
size_t count = 10;
dispatch_apply(count, q_2, ^(size_t i) {
    NSLog(@"%zu",i);
});

NSLog(@"OK");
```

三个参数分别是任务数，目标queue，任务描述

这里注意描述任务的block会重复多次调用，每次会给我们一个参数，表示任务次序。如果目标queue是串行的，那么任务会依次执行，如果queue是并行的，那么任务会并发的执行，打印的顺序就会被打乱。

3.4 死锁

同步提交在某种情况下会造成死锁，即卡死。

示例1:

```
//在主线程中写入下列请求
dispatch_queue_t mainQ = dispatch_get_main_queue();
dispatch_async(mainQ, ^{

    NSLog(@"----");

});
 NSLog(@"OK");
```

示例2:

```
dispatch_queue_t q_1 =  dispatch_queue_create("task3.queue.1",
DISPATCH_QUEUE_SERIAL);

dispatch_async(q_1, ^{


    NSLog(@"current is in q_1");


    // 在q_1内 使用同步提交blcok给q_1 ，因为q_1是串行队列，所以dispatch_sync执行不完
block不会执行，block不执行完，dispatch_sync不会返回
   dispatch_sync(q_1, ^{

       NSLog(@"this is sync ");
   });


    NSLog(@"this is sync ????");

});
```

综合上面两个示例代码，结论显而易见：在一个串行队列执行的代码中，如果向此队列同步提交一个任务，会造成死锁。

四 queue的暂停和继续

我们可以使用dispatch_suspend函数暂停一个queue以阻止它执行block对象;使用dispatch_resume函数继续dispatch queue。挂起和继续是异步的,而且只在执行block之间（比如在执行一个新的block之前或之后）生效。挂起一个queue不会导致正在执行的block停止。

在非ARC中使用时需要注意：调用dispatch_suspend会增加queue的引用计数,调用dispatch_resume则减少queue的引用计数。当引用计数大于0时,queue就保持挂起状态。因此你必须对应地调用suspend和resume函数。

五 Dispatch Group

什么是dispatch group，就像我们在NSOperation中添加依赖一样，如果我们一个任务需要等待其他一些任务完成才能执行时，我们使用dispatch group是最轻松的解决方式。我们以吃火锅为例：

在一无所有的情况下想吃火锅需要以下操作：

1. 买锅
2. 买食材
3. 买饮料
4. ....
5. 吃火锅

我们可以看到最终吃火锅的这一步，需要前面几步执行完才可以。

5.1 提交任务

1. 首先创建group
2. 提交任务，并且把任务添加到group中

示例：

```
dispatch_queue_t globleQ = dispatch_get_global_queue(0, 0);

dispatch_group_t group = dispatch_group_create();


dispatch_group_async(group, globleQ, ^{


    usleep(20000);
    NSLog(@"买肉");
});

dispatch_group_async(group, globleQ, ^{
            usleep(20000);
    NSLog(@"买锅");
});

dispatch_group_async(group, globleQ, ^{

    sleep(2);
    NSLog(@"买料");
});
```

5.2 提交终极任务

5.2.1 在其他线程执行终极任务

```
//    异步 不阻塞线程
dispatch_group_notify(group, dispatch_get_global_queue(0, 0), ^{
    NSLog(@"吃火锅");

});
```

5.2.2 在本线程执行终极任务

使当前线程堵塞，一直等待group所有任务结束：

```
long res = dispatch_group_wait(group, DISPATCH_TIME_FOREVER);

NSLog(@"吃火锅");
```

也可以自定义超时时间，只等待一定的时间：

```
dispatch_time_t time = dispatch_time(DISPATCH_TIME_NOW, (300 *
NSEC_PER_SEC));

dispatch_group_wait(group, time);

NSLog(@"已饿死");
```