NSOperation

(ИСТОЧНИК HTTP://NSHIPSTER.COM/NSOPERATION/)

Classes

- NSOperationQueue
- NSOperation
  - NSOperation subclass
  - NSBlockOperation
  - NSInvocationOperation

NSOperation - представляет собой единую единицу работы.
Это абстрактный класс, который предлагает полезную, потокобезопасную структуру для моделирования состояния, приоритета, зависимостей и управления.

NSOperationQueue - регулирует одновременное выполнение операций.
Оно действует как приоритетная очередь.
То есть все выполняется не совсем по порядку, а по приоритету.
За приоритет отвечает NSOperation.queuePriority.
Можно ограничить количество выполняемых операций одно временно установив их количество в проверти .

NSOperationQueue

```
@interface NSOperationQueue : NSObject

- (void)addOperation:(NSOperation *)op;
- (void)addOperations:(NSArray<NSOperation *> *)ops waitUntilFinished:(BOOL)wait;

- (void)addOperationWithBlock:(void (^)(void))block;

@property (readonly, copy) NSArray<__kindof NSOperation *> *operations;
@property (readonly) NSUInteger operationCount;

-   (void)waitUntilAllOperationsAreFinished;

@property (getter=isSuspended) BOOL suspended;
@property (nullable, copy) NSString *name;
@property NSQualityOfService qualityOfService;


@end
```

State / Состояние NSOperation

ready → executing → finished

Вместо явного состояния пропеперти, состояние определяется косвенно путем нотификаций kvo на тех keypaths.

Когда NSOperation готова к выполнению, она отправляет нотификацию KVO для готового keypath, соответствующее проверти которого затем возвращает true.

Есть такая реализация путем kvo.

Configuring for concurrent execution

```
@interface CustomOperation () {
    BOOL executing;
    BOOL finished;
}

- (void)completeOperation;

@end

@implementation CustomOperation

- (instancetype)initWithString:(NSString *)param {
    self = [super init];

    if (self) {
        executing = NO;
        finished = NO;

        self.param = param;
    }

    return self;
}

- (BOOL)isConcurrent {
    return YES;
}

- (BOOL)isExecuting {
    return executing;
}

- (BOOL)isFinished {
    return finished;
}
```

```
- (void)start {
    if ([self isCancelled]) {
        [self willChangeValueForKey:@"isFinished"];
        finished = YES;
        [self didChangeValueForKey:@"isFinished"];
        return;
    }

    [self willChangeValueForKey:@"isExecuting"];
    [NSThread detachNewThreadSelector:@selector(main)
                             toTarget:self
                           withObject:nil];
    executing = YES;
    [self didChangeValueForKey:@"isExecuting"];
}

- (void)main {
    @try {
        // Do some work
        [self completeOperation];
    }
    @catch (NSException *exception) {}
}

- (void)completeOperation {
    [self willChangeValueForKey:@"isFinished"];
    [self willChangeValueForKey:@"isExecuting"];

    executing = NO;
    finished = YES;

    [self didChangeValueForKey:@"isExecuting"];
    [self didChangeValueForKey:@"isFinished"];
}
```

(Каждое проверти должно быть взаимоисключающим друг от друга для кодирования согласованного состояния:)

ready: Возвращает true, чтобы указать, что операция готова к выполнению, или false, если есть еще незавершенные шаги инициализации, от которых она зависит.

executing: Возвращает true, если операция в данный момент работает над своей задачей, или false в противном случае.

finished: Возвращает true, если NSOperation успешно завершила выполнение задачи или была отменена.
NSOperationQueue не выкидывает операцию из очереди до завершения изменения true, поэтому очень важно правильно реализовать это в подклассах,чтобы избежать взаимоблокировки.

Cancellation

Подобно состоянию выполнения, NSOperation передает отмену через KVO на cancelled keypath. Когда NSOperation отменяется(cancelled), она должна очистить все внутренние детали и прибыть в соответствующее конечное состояние как можно быстрее.
В частности, значения как для cancelled (отмененных), так и для finished(завершенных) должны стать true, а executing(выполнение) должно стать false.

Priority

Все операции могут быть не одинаково важны. Установка проперти queuePriority будет способствовать или отложить операцию в NSOperationQueue по следующей рейтинга:

```
typedef NS_ENUM(NSInteger, NSOperationQueuePriority) {
    NSOperationQueuePriorityVeryLow = -8L,
    NSOperationQueuePriorityLow = -4L,
    NSOperationQueuePriorityNormal = 0,
    NSOperationQueuePriorityHigh = 4,
    NSOperationQueuePriorityVeryHigh = 8
};
```

Quality of Service - (QoS)

Это новая концепция в iOS 8 и OS X Yosemite, которая создает последовательную семантику высокого уровня для планирования системных ресурсов.
Уровни обслуживания устанавливают обще системный приоритет операции с точки зрения объема ресурсов ЦП, сети и диска.
Чем “выше” уровень тем более будет предоставлена системных возможностей.

```
typedef NS_ENUM(NSInteger, NSQualityOfService) {
    NSQualityOfServiceUserInteractive = 0x21,
    NSQualityOfServiceUserInitiated = 0x19,
    NSQualityOfServiceUtility = 0x11,
    NSQualityOfServiceBackground = 0x09,
    NSQualityOfServiceDefault = -1
} NS_ENUM_AVAILABLE(10_10, 8_0);
```

NSQualityOfServiceUserInteractive - используется для работы, непосредственно связанной с предоставлением UI, такого как обработка событий или рисование на экране.

NSQualityOfServiceUserInitiated - используется для выполнения работы, которая была явно запрошена пользователем и для которой результаты должны быть немедленно представлены, чтобы обеспечить дальнейшее взаимодействие с пользователем. Например, загрузка сообщения электронной почты после его выбора Пользователем в списке Сообщений.

NSQualityOfServiceUtility - используется для выполнения работ, которые пользователь вряд ли будет сразу ждать результатов. Например, периодические обновления содержимого или массовые операции с файлами, такие как импорт мультимедиа.

NSQualityOfServiceBackground - используется для работы, которая не инициируется пользователем или не видна. В общем, пользователь не знает, что эта работа даже происходит.

NSQualityOfServiceDefault - по умолчанию указывает на отсутствие информации о QoS. По мере возможности информация о QoS будет выводиться из других

источников. Если такой вывод невозможен, будет использоваться QoS между UserInitiated и Utility.

Пример задачи приоритета и QoS

```
NSOperation *backgroundOperation = [[NSOperation alloc] init];
backgroundOperation.queuePriority = NSOperationQueuePriorityLow;
backgroundOperation.qualityOfService =
NSOperationQualityOfServiceBackground;

[[NSOperationQueue mainQueue] addOperation:backgroundOperation];
```

Dependencies

Пример:

```
NSOperation *networkingOperation = ...
NSOperation *resizingOperation = ...
[resizingOperation addDependency:networkingOperation];

NSOperationQueue *operationQueue = [NSOperationQueue mainQueue];
[operationQueue addOperation:networkingOperation];
[operationQueue addOperation:resizingOperation];
```

Операция не будет запущена, пока все ее зависимости не вернутся к завершению. (Убедитесь в том, чтобы случайно не создать цикл зависимостей, такой, что а зависит от B, A B зависит от A, например. Это создаст тупик и печаль.)

CompletionBlock

Когда NSOperation завершится, он выполнит свой completionBlock.

```
NSOperation *operation = ...;
operation.completionBlock = ^{
    NSLog("Completed");
};

[[NSOperationQueue mainQueue] addOperation:operation];
```

(источник https://www.objc.io/issues/2-concurrency/concurrency-apis-and-pitfalls/)

Operation Queues

Класс **NSOperationQueue** имеет два разных типа очередей: **main queue** и **custom queues**.

1. **Main queue** (Основная очередь) выполняется в **main thread** (основном потоке).
2. **Custom queues** (кастом очереди) обрабатываются в **background** thread (фоновом режиме).

В любом случае, задачи, которые обрабатываются **NSOperationQueue**, представляются как подклассы **NSOperation**.

Вы можете определить свои собственные **NSOperation** двумя способами:
1. либо переопределив метод -**main**,
2. либо переопределив метод -**start**.

Первое очень просто сделать, но дает вам меньше гибкости.

```
@implementation YourOperation
    - (void)main
    {
        // do your work here ...
    }
@end
```

В свою очередь, свойства состояния, такие как **isExecuting** и **isFinished** управляются для вас, просто предполагая, что операция завершена, когда main возвращается.

Если вам нужно больше управления и, возможно, выполнить асинхронную задачу в рамках операции, вы можете переопределить **start**:

(Примечание: если мы переопределяем метод -**start**. Тогда для того чтобы начать выполнение операции нужно его вызвать. Если нет, то для начала выполнения хватит только добавления в **NSOperationQueue**).

```
@implementation YourOperation
    - (void)start
    {
        self.isExecuting = YES;
        self.isFinished = NO;
        // start your work, which calls finished once it's done ...
    }

    - (void)finished
    {
        self.isExecuting = NO;
        self.isFinished = YES;
    }
```

```
- (void)main
{
    while (notDone && !self.isCancelled) {
        // do your processing
    }
}
@end
```

Теперь добавляем операцию в очередь

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
YourOperation *operation = [[YourOperation alloc] init];
[queue  addOperation:operation];
```

Кроме того, вы также можете добавлять блоки в очереди операций. Это пригодится, например. если вы хотите назначить одноразовые задачи в основной очереди:

```
[[NSOperationQueue mainQueue] addOperationWithBlock:^{
    // do something...
}];
```

Установка зависимостей

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

```
NSBlockOperation *block1 = [NSBlockOperation blockOperationWithBlock:^{
    for (int i = 0; i < kIntervalCount; ++i) {
        [NSThread sleepForTimeInterval:0.5];
        NSLog(@"block1: %d, (%@)", i, [NSThread currentThread]);
    }
}];

NSBlockOperation *block2 = [NSBlockOperation blockOperationWithBlock:^{
    for (int i = 0; i < kIntervalCount; ++i) {
        [NSThread sleepForTimeInterval:0.2];
        NSLog(@"block2: %d, (%@)", i, [NSThread currentThread]);
    }
}];

NSOperationQueue *queue = [NSOperationQueue new];

[queue addOperation:block1];
[queue addOperation:block2];

NSLog(@"Here we go!");
```

```
2016-02-28 19:14:53.027 Here we go!
2016-02-28 19:14:53.237 block2: 0, (<NSThread: 0x610000067440>{number = 2, name = (null)})
2016-02-28 19:14:53.437 block2: 1, (<NSThread: 0x610000067440>{number = 2, name = (null)})
2016-02-28 19:14:53.528 block1: 0, (<NSThread: 0x61000006f140>{number = 3, name = (null)})
2016-02-28 19:14:53.637 block2: 2, (<NSThread: 0x610000067440>{number = 2, name = (null)})
2016-02-28 19:14:53.838 block2: 3, (<NSThread: 0x610000067440>{number = 2, name = (null)})
2016-02-28 19:14:54.029 block1: 1, (<NSThread: 0x61000006f140>{number = 3, name = (null)})
2016-02-28 19:14:54.040 block2: 4, (<NSThread: 0x610000067440>{number = 2, name = (null)})
2016-02-28 19:14:54.530 block1: 2, (<NSThread: 0x61000006f140>{number = 3, name = (null)})
2016-02-28 19:14:55.030 block1: 3, (<NSThread: 0x61000006f140>{number = 3, name = (null)})
2016-02-28 19:14:55.532 block1: 4, (<NSThread: 0x61000006f140>{number = 3, name = (null)})
```

Прочие примеры использования

Создаем операции

NSOperationQueue

```
NSOperation *networkingOperation = ...
NSOperation *resizingOperation = ...

resizingOperation addDependency:networkingOperation];

NSOperationQueue *operationQueue = [NSOperationQueue new];

[operationQueue addOperation:networkingOperation];
[operationQueue addOperation:resizingOperation];
```

Работа с NSBlockOperation

NSInvocationOperation

```
- (void)runWithCount:(NSNumber *)count {
    for (int i = 0; i < [count intValue]; ++i) {
        [NSThread sleepForTimeInterval:0.5];
        NSLog(@"operation: %d", i);
    }
}

- (void)example5 {
    NSInvocationOperation *invOp;
    invOp = [[NSInvocationOperation alloc] initWithTarget:self
                                selector:@selector(runWithCount:)
                                object:@(3)];

    [invOp start];
}
```

NSInvocationOperation result

```
- (NSNumber *)sum {
    int result;

    for (int i = 0; i < kIntervalCount; ++i) {
        result += i;
        NSLog(@"result: %d", result);
    }

    return @(result);
}

- (void)example6 {
    NSInvocationOperation *invOp;
    invOp = [[NSInvocationOperation alloc] initWithTarget:self
                                                 selector:@selector(sum)
                                                   object:nil];

    [invOp start];

    NSNumber *result = [invOp result];
}
```

NSOperationQueue

- Serial

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

- Concurrent

```
queue.maxConcurrentOperationCount = 6;
```

```
static const NSInteger
NSOperationQueueDefaultMaxConcurrentOperationCount = -1;
```

```
@interface CustomOperation : NSOperation

- (instancetype)initWithString:(NSString *)param;

@property(nonatomic, strong) NSString *param;

@end


@implementation CustomOperation

- (instancetype)initWithString:(NSString *)param {
    self = [super init];

    if (self) {
        self.param = param;
    }

    return self;
}

- (void)main {
    for (int i = 0; i < 5; ++i) {
        [NSThread sleepForTimeInterval:.5];
        NSLog(@"%@: %d", self.param, i);
    }
}

@end



    CustomOperation *op1 = [[CustomOperation alloc] initWithString:@"op1"];
    CustomOperation *op2 = [[CustomOperation alloc] initWithString:@"op2"];

    NSOperationQueue *queue = [NSOperationQueue new];

    [queue addOperation:op2];
    [queue addOperation:op1];
```

```
op1: 0
op2: 0
op2: 1
op1: 1
op1: 2
op2: 2
op2: 3
op1: 3
op1: 4
op2: 4
```