



## Урок 3

# Параллельное программирование. NSOperation

Знакомство с библиотекой NSOperation для организации многопоточного кода и параллельного выполнения задач.

[Введение](#)

[Operation](#)

[Очереди класса OperationQueue](#)

[Создание операции](#)

[Жизненный цикл операции](#)

[Асинхронные операции](#)

[Зависимости](#)

[Управление очередями](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

# Введение

На этом уроке мы познакомимся с классом **Operation**, который позволяет абстрагироваться от понятий потоков, блоков кода и очередей. С его помощью создаются операции, которые выполняются на отдельных потоках. Их можно выстроить в цепочки с определенной последовательностью.

Например, можно определить такую очередность выполнения операций: получаем данные из интернета – сохраняем данные в базу – извлекаем данные из базы – обновляем таблицу.

## Очереди класса OperationQueue

Все операции выполняются на очередях. Они представлены классом **OperationQueue**. Но это не те очереди, которые вы видели на уроке про **GCD**. Они и проще, и сложнее одновременно.

Есть только одна готовая очередь по умолчанию – **OperationQueue.main**, связанная с главным потоком. Дополнительные очереди вы создаете по мере необходимости. Обойтись можно всего двумя – главной и одной дополнительной. У **OperationQueue** доступен всего один конструктор, не принимающий никаких параметров.

Рассмотрим пример с двумя очередями.

```
// Главная очередь
OperationQueue.main
// Дополнительная очередь, которую мы создали
let myOwnQueue = OperationQueue()
```

Теперь выполним операцию самым простым способом. Он очень похож на работу с **GCD**.

```
let myOwnQueue = OperationQueue()
    // Добавляем операцию в очередь
    myOwnQueue.addOperation {
        // Выполняем расчеты
        let summ = 4 + 5
        let stringSumm = String(describing: summ)
        // Добавляем операцию на главный поток для работы с UI
        OperationQueue.main.addOperation { [weak self] in
            self?.label.text = stringSumm
        }
    }
```

Разумеется, это не единственная возможность **Operation**. Для таких конструкций лучше использовать **GCD**. Разбираем класс далее.

## Создание операции

Полноценные операции создаются путем наследования от базового класса **Operation**.

Напишем операцию для размытия изображения – как это было с кодом на прошлом уроке.

```

// Операция размытия изображения
class BlurImageOperation: Operation {
// Исходное изображение
    var inputImage: UIImage
// Размытое изображение
    var outputImage: UIImage?

// Логика нашей операции
    override func main() {
// Размываем изображение
        let inputCIImage = CIImage(image: inputImage)!
        let filter = CIFilter(name: "CIGaussianBlur", withInputParameters:
[kCIInputImageKey: inputCIImage])!
        let outputCIImage = filter.outputImage!
        let context = CIContext()

        let cgiImage = context.createCGImage(outputCIImage, from:
outputCIImage.extent)

// Кладем размытое изображение в свойство
        outputImage = UIImage(cgImage: cgiImage!)
    }

// Конструктор для создания операции с изображением
    init(inputImage: UIImage) {
        self.inputImage = inputImage
        super.init()
    }
}

```

Это максимально простой подкласс **Operation**. Мы добавили два свойства для хранения исходного и размытого изображения: **inputImage** и **outputImage**. Также ввели конструктор для установки первоначальной картинки. Главное: мы **переопределили** метод **main**. Он содержит логику операции: здесь читается исходное изображение, затем к нему применяется фильтр размытия и оно помещается в свойство **outputImage** – чтобы с ним можно было работать извне. Добавляем операцию в очередь.

```

let blurTreeOperation = BlurImageOperation(inputImage: UIImage(named:
"treeSmall")!)
blurTreeOperation.completionBlock = {
    OperationQueue.main.addOperation { [weak self] in
        self?.imageView.image = blurTreeOperation.outputImage!
    }
}
queue.addOperation(blurTreeOperation)

```

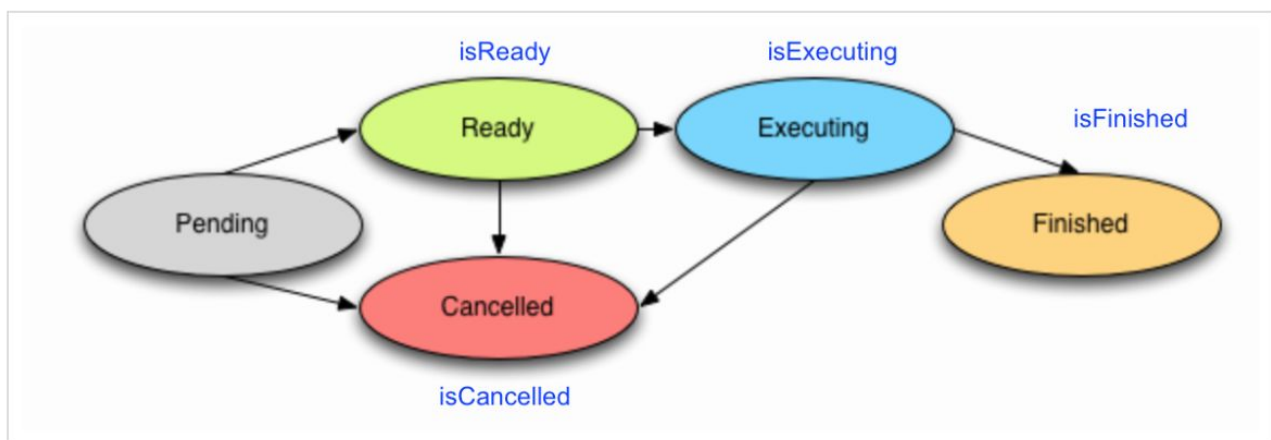
Важный момент из примера – свойство **completionBlock**. Оно позволяет легко отследить завершение операции, так как замыкание в этом свойстве происходит по итогу выполнения задачи. Здесь получаем размытое изображение, после чего устанавливаем его **UIImageView**.

## Жизненный цикл операции

У операции есть несколько состояний, очень похожих на подобные у потока (**Thread**). В конкретный момент времени операция может находиться только в одном состоянии.

1. **Pending** – отложенная.
2. **Ready** – готова к выполнению.
3. **Executing** – выполняется.
4. **Finished** – закончена.
5. **Cancelled** – уничтожена.

Состояния меняются последовательно: попасть в следующее можно, только пройдя предыдущее. Исключение – состояние **isCancelled**, которое осуществимо практически из любого другого.



Когда операция создается и размещается в очереди, ее состояние соответствует статусу **pending**. Спустя некоторое время она принимает состояние **ready** и в любой момент может начать выполнение, перейдя в **executing**. Оно может длиться от миллисекунд до нескольких минут и более. После завершения операция переходит в финальное состояние **finished**. В любой точке этого жизненного цикла операция может быть уничтожена – **cancelled**.

Операция уничтожается с помощью вызова метода **cancel()**. Важно: сам по себе он не отменит выполнение операции, а только установит флаг **isCancelled** в состояние **true**. Обработать это и остановить задачу должны вы сами. Точно так же мы делали, когда создавали свои потоки (**Thread**).

## Асинхронные операции

До сих пор мы не выявили особых отличий в работе **NSOperation** и **GCD**. Отчасти это так: операции всего лишь представляют другой формат работы с заданиями. Пора показать уникальность и пользу **NSOperation** – начнем с асинхронных операций.

Вспомним: асинхронные задачи состоят из двух частей, вторая из которых выполняется не сразу, а по наступлении определенного события.

При работе с такими задачами в **GCD** мы теряем контроль над их завершением. Надо использовать вложенные замыкания, семафоры – словом, постоянно изобретать велосипед. С помощью **Operation** такие задачи можно рассматривать как единое целое, собирать в цепочки и делать множество других интересных вещей.

Напишем операцию загрузки данных из интернета с использованием **alamofire**.

```
class GetDataOperation : Operation {

    enum State: String {
        case ready, executing, finished
        fileprivate var keyPath: String {
            return "is" + rawValue.capitalized
        }
    }

    private var state = State.ready {
        willSet {
            willChangeValue(forKey: state.keyPath)
            willChangeValue(forKey: newValue.keyPath)
        }
        didSet {
            didChangeValue(forKey: state.keyPath)
            didChangeValue(forKey: oldValue.keyPath)
        }
    }

    override var isAsynchronous: Bool {
        return true
    }

    override var isReady: Bool {
        return super.isReady && state == .ready
    }

    override var isExecuting: Bool {
        return state == .executing
    }

    override var isFinished: Bool {
        return state == .finished
    }

    override func start() {
        if isCancelled {
            state = .finished
        } else {
            main()
            state = .executing
        }
    }

    override func cancel() {
        request.cancel()

        super.cancel()
    }
}
```

```

        state = .finished
    }

    private var request: DataRequest =
Alamofire.request("https://jsonplaceholder.typicode.com/posts")
    var data: Data?

    override func main() {
        request.responseData(queue: DispatchQueue.global()) { [weak self]
response in
            self?.data = response.data
            self?.state = .finished
        }
    }
}

```

Не пугаемся внушительного кода – мы можем сделать один универсальный класс для всех сетевых запросов. Пока разберемся, что здесь к чему.

При реализации асинхронной задачи нужно переопределять не только метод **main**, но и методы **start**, **cancel**, а также свойства **isAsynchronous**, **isReady**, **isExecuting**, **isFinished**.

Чтобы операция стала асинхронной, переопределяем свойство **isAsynchronous**, чтобы оно возвращало **true**.

```

    override var isAsynchronous: Bool {
        return true
    }

```

Так как мы не можем менять свойства **isReady**, **isExecuting**, **isFinished**, наведем собственное свойство состояний, которое будет возвращать стандартные флаги. Это будет перечисление – опишем его.

```

enum State: String {
    case ready, executing, finished
    fileprivate var keyPath: String {
        return "is" + rawValue.capitalized
    }
}

```

Всего три состояния – их хватит. Плюс само свойство.

```

private var state = State.ready {
    willSet {
        willChangeValue(forKey: state.keyPath)
        willChangeValue(forKey: newValue.keyPath)
    }
    didSet {
        didChangeValue(forKey: state.keyPath)
        didChangeValue(forKey: oldValue.keyPath)
    }
}

```

```
}
```

Здесь пригодятся знания о структуре свойств с подготовительных курсов. Само свойство простое – переменная **state**, наподобие перечисления **State** с установленным по умолчанию значением **ready**.

Дальше описаны два слушателя изменений этого свойства: **willSet** и **didSet**. В них вызываются методы. **WillChangeValue** отправляет KVO-уведомление о том, что свойство операции собирается измениться. Имена изменяемых свойств поступают из свойства перечисления **keyPath**. **DidChangeValue** отправляет KVO-уведомление, что свойство операции изменилось. Это необходимо, чтобы класс **Operation** был совместим с механизмом KVO из Objective-C.

```
override var isReady: Bool {  
    return super.isReady && state == .ready  
}
```

Теперь учитывается и предопределенное свойство **isReady**, и состояние операции, которое устанавливает ему очередь и значение.

```
override var isExecuting: Bool {  
    return state == .executing  
}  
  
override var isFinished: Bool {  
    return state == .finished  
}
```

Флаги состояния проверяют свойство **state** и возвращают результат.

Так мы сами сможем управлять состоянием операции, меняя значение свойства **state**. А стандартные флаги, переопределенные нами, будут возвращать это состояние. Теперь, пока мы сами не завершим операцию, присвоив значение **finish** свойству **state**, она не завершится.

```
override func start() {  
    if isCancelled {  
        state = .finished  
    } else {  
        main()  
        state = .executing  
    }  
}
```

Изменению подвергается и метод **start**, который очередь вызывает при начале выполнения операции. Проверяем, не была ли отменена операция еще до начала выполнения. Если была, меняем состояние на **finished**. Если нет – вызываем метод **main**, запуская задачу, и определяем состояние как **executing**.

```
private var request: DataRequest =  
Alamofire.request("https://jsonplaceholder.typicode.com/posts")
```

Свойство **request** содержит запрос **alamofire**, готовый к исполнению.



```

override func cancel() {
    request.cancel()

    super.cancel()
    state = .finished
}

```

Переопределенный метод **cancel** вызывает реализацию метода из родительского класса, отменяет запрос и устанавливает **state** в состояние **выполнено (finished)**. Таким образом мы отменили задачу.

```

var data: Data?

```

Свойство **data** служит хранилищем, куда мы поместим данные из интернета.

```

override func main() {
    request.responseData(queue: DispatchQueue.global()) { [weak self]
response in
        self?.data = response.data
        self?.state = .finished
    }
}

```

Переопределенный метод **main** выполняет запрос в интернет, сохраняет данные в свойство **data** и устанавливает состояние операции **finished**. Таким образом, операция после старта будет считаться выполненной, пока мы сами ее не завершим в замыкании получения данных.

Важно: замыкание обработки данных **alamofire** по умолчанию всегда выполняется на главном потоке. Необходимо переключить его на глобальную очередь – ведь смысл нашей операции в фоновом выполнении.

Операция выглядит неоправданно сложной – особенно для написания под каждый запрос. Но основная часть кода является универсальной и наследуется.

```

class AsyncOperation: Operation {
    enum State: String {
        case ready, executing, finished
        fileprivate var keyPath: String {
            return "is" + rawValue.capitalized
        }
    }

    var state = State.ready {
        willSet {
            willChangeValue(forKey: state.keyPath)
            willChangeValue(forKey: newValue.keyPath)
        }
        didSet {
            didChangeValue(forKey: state.keyPath)
            didChangeValue(forKey: oldValue.keyPath)
        }
    }
}

```

```

    }

    override var isAsynchronous: Bool {
        return true
    }

    override var isReady: Bool {
        return super.isReady && state == .ready
    }

    override var isExecuting: Bool {
        return state == .executing
    }

    override var isFinished: Bool {
        return state == .finished
    }

    override func start() {
        if isCancelled {
            state = .finished
        } else {
            main()
            state = .executing
        }
    }

    override func cancel() {
        super.cancel()
        state = .finished
    }
}

```

Создадим общий для всех асинхронных операций класс **AsyncOperation**. Он будет содержать всю необходимую логику, но в нем не будет подробностей о том, чем конкретно операция занимается.

И напомним «наследника», который будет заниматься запросами в сеть.

```

class GetDataOperation: AsyncOperation {

    override func cancel() {
        request.cancel()
        super.cancel()
    }

    private var request: DataRequest =
Alamofire.request("https://jsonplaceholder.typicode.com/posts")
    var data: Data?

    override func main() {
        request.responseData(queue: DispatchQueue.global()) { [weak self]
response in
            self?.data = response.data
            self?.state = .finished
        }
    }
}

```

```
}  
  
}
```

Код стал намного короче, и таких операций можно сделать много, просто «наследуясь» от **AsyncOperation**.

Пойдем дальше: напишем универсальную операцию, которую можно будет использовать для любого запроса.

```
class GetDataOperation: AsyncOperation {  
  
    override func cancel() {  
        request.cancel()  
        super.cancel()  
    }  
  
    private var request: DataRequest  
    var data: Data?  
  
    override func main() {  
        request.responseData(queue: DispatchQueue.global()) { [weak self]  
response in  
            self?.data = response.data  
            self?.state = .finished  
        }  
    }  
  
    init(request: DataRequest) {  
        self.request = request  
    }  
  
}
```


Конкретный запрос передается в конструктор, и операция его загружает. Создаем запрос и операцию на его основе. В **completionBlock** помещаем замыкание для вывода полученных данных в консоль.

```
let request = Alamofire.request("https://jsonplaceholder.typicode.com/posts")  
let op = GetDataOperation(request: request)  
op.completionBlock = {  
    print(op.data)  
}  
opq.addOperation(op)
```

## Зависимости

Operation позволяет работать с классами и объектами при выполнении многопоточных заданий. Это более естественно, чем использование функций и замыканий в GCD.

Замыкания позволяют отследить выполнение операции и обработать ее результат. Но иметь дело с ними становится сложно, если необходимо выполнить много связанных задач, где каждой последующей требуется результат предыдущей. В этом случае приходится размещать одно замыкание в другом, получая так называемый **callback hell**, или **pyramid of doom**, наглядно показанный ниже.



```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

Это не очень приятно писать и поддерживать. Справиться с этой проблемой помогают **зависимости между задачами** – они выстраивают логику выполнения целых цепочек заданий, связанных между собой. Пока все зависимости конкретной задачи не будут выполнены, не начнется ее собственная реализация.

Установим возможные зависимости у нашей операции загрузки данных. Сначала данные надо загрузить, потом преобразовать в необходимый формат и, например, отобразить в таблице. Напишем недостающие операции.

```
struct Post {
    let id: Int
    let title: String
    let body: String
}
```

Это структура поста, полученного из интернета.

```
class ParseData: Operation {
```

```

var outputData: [Post] = []

override fun main() {
    guard let getDataOperation = dependencies.first as? GetDataOperation,
        let data = getDataOperation.data else { return }
    let json = JSON(data: data)
    let posts: [Post] = json.flatMap {
        let id = $0.1["id"].intValue
        let title = $0.1["title"].stringValue
        let body = $0.1["body"].stringValue
        return Post(id: id, title: title, body: body)
    }
    outputData = posts
}
}

```

Это операция парсинга данных. Обратите внимание на строку получения данных из операции загрузки. У каждой операции есть свойство **dependencies**, хранящее все ее зависимости. Так наша операция парсинга будет зависимой от операции загрузки, она будет находиться в этом массиве. Получив ее отсюда, мы сможем взять из нее данные.

```

class ReloadTableController: Operation {
    var controller: TableController

    init(controller: TableController) {
        self.controller = controller
    }

    override fun main() {
        guard let parseData = dependencies.first as? ParseData else { return }
        controller.posts = parseData.outputData
        controller.tableView.reloadData()
    }
}

```

Операция обновления UI содержит ссылку на контроллер, в котором находится нужная нам таблица. В ней мы получаем операцию парсинга данных, сами данные кладем в контроллер и обновляем таблицу.

Осталось этим воспользоваться:

```

let request = Alamofire.request("https://jsonplaceholder.typicode.com/posts")
let getDataOperation = GetDataOperation(request: request)
opq.addOperation(getDataOperation)

let parseData = ParseData()
parseData.addDependency(getDataOperation)
opq.addOperation(parseData)

```

```
let reloadDataController = ReloadTableController(controller: self)
reloadTableController.addDependency(parseData)
OperationQueue.main.addOperation(reloadTableController)
```

1. Создаем задачу загрузки данных и отправляем ее в очередь.
2. Создаем задачу парсинга данных, добавляем ей зависимость и тоже отправляем в очередь.
3. Создаем задачу обновления UI, добавляем ей зависимость и внимание, помещаем на главную очередь – ведь все взаимодействие с UI должно выполняться в главном потоке.

Так можно выполнять операции без замыканий и блокировок главного потока. Более того, можно сделать все операции более абстрактными и собирать из них нужные задачи, как из «Лего».

Например, к операции загрузки данных можно «подцепить» создание изображения, а к нему – размытие или другой фильтр. Можно внедрять в цепочку сохранение кэша, а можно уберечь.

Код становится очень гибким, хотя и требует немного больше времени на проектирование операций. Но увлекаться и превращать весь код в сплошную цепочку связанных операций не надо. Впрочем, если вам очень понравился этот подход, посмотрите на реактивные фреймворки, например RxSwift. Возможно, это ваше.

## Управление очередями

Кроме создания операций, добавления их в очереди и установления между ними зависимостей, можно еще и управлять очередями.

Проще всего поменять очереди приоритет через свойство **qualityOfService**.

```
let queue = OperationQueue()
queue.qualityOfService = .userInteractive
```

Работает это точно так же, как с **GCD**-очередями. Можно менять приоритет и у конкретных операций – это удобно.

```
let first = SimpleOperation(char: "😊")
let second = SimpleOperation(char: "😈")
first.qualityOfService = .userInteractive
opq.addOperation(first)
opq.addOperation(second)
```

Устанавливается количество одновременных операций, которые могут выполняться на очереди. Если задать этот параметр равным единице, очередь станет последовательной.

```
let queue = OperationQueue()
queue.maxConcurrentOperationCount = 1
```

Можно приостанавливать и возобновлять выполнение всех операций на очереди.

```
let queue = OperationQueue()
queue.isSuspended = true
queue.isSuspended = false
```

Это может быть полезно для приостановки операций с изображениями, когда нужно освободить канал передачи данных для более важной информации.

Можно отменить все операции, находящиеся в очереди.

```
let queue = OperationQueue()  
queue.cancelAllOperations()
```

Важно: запущенные операции остановлены не будут, если вы сами не предусмотрели обработку отмены в момент выполнения.

Можно посмотреть количество операций, находящихся в данный момент в очереди, и получить их.

```
queue.operationCount  
queue.operations
```

Можно подождать, пока выполнятся все операции, уже находящиеся в очереди.

```
queue.waitUntilAllOperationsAreFinished()
```

Важно: никогда не использовать этот метод в главном потоке, так как он блокирует поток и ждет завершения задач.

## Практическое задание

1. Перевести всю работу с сетью и парсингом в приложении на GCD.

## Дополнительные материалы

1. <https://habrahabr.ru/post/335756/>
2. <https://developer.apple.com/documentation/foundation/operation>

## Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://developer.apple.com/documentation>