Урок 1

Параллельное программирование. Thread

Знакомство с понятиями многопоточного и асинхронного кода. Управление потоками. RunLoop. Thread.

Параллельное программирование

Параллельное программирование – это способ организации параллельных, одновременных вычислений в вашей программе. В традиционной последовательной модели код выполняется по порядку, и в конкретный момент времени может обрабатываться только одно действие. Параллельное программирование позволяет решать несколько задач одновременно.

Например, вы можете отрисовывать прокрутку таблицы, загружая данные из интернета. А при последовательном способе ваш интерфейс замрет и не будет реагировать на запросы пользователя, пока не завершит загрузку данных.

Для начала разберемся, из чего состоит и как работает ваше приложение.

На каком языке программирования или в какой операционной системе вы бы ни писали приложение, в конечном счете его будет выполнять процессор. Он может распознавать простейшие команды – например, сложение или сравнение двух значений в памяти. Из последовательности таких команд и строится работа компьютера.

Самые первые компьютеры, собственно, ничего другого и не умели – на них не было операционной системы в современном понимании. Программирование представляло собой перевод математических действий в простейшие операции, которые мог выполнить процессор. По сути, это был просто большой и быстрый калькулятор. Кстати, от английского «to compute» – «вычислять» и происходит слово «компьютер», буквально – «вычислитель».

Время шло, компьютеры становились быстрее, их научили не только производить вычисления, но и отображать данные на мониторе, выводить их на печать, считывать символы с клавиатуры. Появилось множество программ, при работе с которыми пользователю понадобилась возможность запускать их одну за другой, выбирать и чередовать. Первоначальный подход, когда компьютер действовал по четкой последовательной инструкции, устарел.

Тогда и появились **операционные системы (ОС)**. Это обычные программы – по типу самых первых программных продуктов. При запуске ОС загружается в память устройства, и процессор начинает ее выполнение. При этом система может запускать в своем контексте другие программы, передавать их код процессору, управлять их выполнением.

Изначально операционная система помогала только выбрать одну программу для выполнения в конкретный момент времени. Прежде чем запустить другую, необходимо было завершить выполняемую. Сейчас это кажется неудобным, но в свое время это был настоящий прорыв.

Прогресс продолжался, вычислительные возможности росли, и простые операции (например, проигрывание музыки) выполнялись очень быстро. В этот момент операционные системы научились совмещать выполнение нескольких программ. А значит, пользователи смогли редактировать тексты под музыку и при этом периодически обновлять почту. А однозадачный процессор по-прежнему физически не мог выполнять две команды одновременно.

Но зато процессор был очень быстрым: одна операция занимала ничтожно малое время. Например, ноутбук с двухъядерным процессором частотой 2 GHz способен выполнять 2 миллиарда операций в секунду. Эта цифра условная – реальную ситуацию диктует множество факторов, но она дает представление о скорости работы устройства.

Тогда и был изобретен **псевдомногозадачный подход**, при котором операционная система чередует выполнение команд из нескольких программ на одном процессоре. Это происходит с такой скоростью, что возникает эффект одновременности. Для современных многоядерных процессоров

этот подход неактуален: они способны фактически выполнять несколько команд одновременно – по одной на ядро.

Следующее понятие – **процесс**. Это **запущенная программа в рамках операционной системы**. Кроме последовательности команд, процесс содержит много других ресурсов: данные в памяти, дескрипторы устройств ввода-вывода, файлов на диске и прочее. Фактически, операционная система состоит из огромного количества запущенных процессов. Одни отвечают за системные функции, работу с сетевыми устройствами и файлами, расчет текущего времени. Другие процессы представляют программы, запущенные пользователем: браузер, плеер, Skype или текстовый редактор.

Для каждого процесса выделяется свой участок в оперативной памяти, где он может размещать данные, переменные, а при наделении эксклюзивными правами – производить запись на файл. При этом доступ к ресурсам других процессов закрыт. Например, открывая файл для записи в одном процессе, вы автоматически блокируете его для других, так как переменные конкретного процесса доступны только ему.

Но процесс – это не наименьшая единица, которая может быть выполнена. **Процессы состоят из потоков**. Wikipedia содержит очень хорошее определение потока, давайте ознакомимся с ним.

Для разработчика iOS-приложений знания о процессоре, процессах и операционной системе носят утилитарный характер. Они открывают картину мира разработки, дают более глубокое понимание принципов работы программы и ее окружения – следовательно, помогают писать более качественный, производительный код. Но создание потоков и управление ими при написании программ – это отдельная тема.

Модели архитектуры

Есть несколько способов организации кода:

1. **Однопоточный** – ваша программа содержит один поток, в него помещаются все задачи и они выполняются поочередно.
2. **Многопоточный** – ваша программа содержит несколько потоков, задачи делятся между потоками и выполняются параллельно (одновременно).
3. **Синхронный** – каждая задача в потоке выполняется от начала и до конца, и пока одна не будет выполнена целиком, начать выполнение другой невозможно.
4. **Асинхронный** – задача в потоке может быть приостановлена в процессе выполнения. Поставив задачу на паузу, можно приступить к выполнению другой, а впоследствии возобновить предыдущую. Кроме того, несколько задач могут быть приостановлены одновременно и многократно.

Под задачей здесь подразумевается последовательность команд, которые должны быть выполнены вместе и дать определенный результат. Задачей может быть одна строка кода, выполняющая простое сложение двух переменных, или же огромный блок, читающий данные с диска, изменяющий их и отправляющий на сервер.

В разных источниках упоминается несколько типов архитектур. Это связано с тем, что способы организации кода могут сочетаться и представлять в итоге такие модели:

1. Синхронную однопоточную.
2. Асинхронную однопоточную.
3. Синхронную многопоточную.
4. Асинхронную многопоточную.

При этом **к архитектурам конкурирующего кода** можно отнести следующие:

1. **Многопоточную модель** – когда программа разбивается на потоки, но задачи внутри потока выполняются синхронно.
2. **Асинхронную модель** – когда программа выполняется в одном потоке, но задачи выполняются асинхронно.

Под конкурирующим кодом подразумевается, что задачи внутри него «соперничают» между собой за право быть выполненными. Именно поэтому мы не рассматриваем модели синхронной и однопоточной архитектуры, так как там нет такой конкуренции.

У вас может возникнуть вопрос: почему многопоточная модель тоже называется конкурирующей? С асинхронной все понятно: задача может быть приостановлена, а другая – запущена вместо нее, – то есть, мы видим борьбу, конкуренцию между ними. В многопоточной же модели такой конкуренции вроде бы нет: задачи выполняются одновременно.

Но у потоков есть общие ресурсы – например, переменные или файлы, в которые задачи из разных потоков записывают свои значения. Или этим задачам может понадобиться доступ к фотокамере устройства, а получить его одновременно нельзя. Как видите, в многопоточной модели конкуренция между задачами идет не за право выполняться, а за владение уникальными ресурсами.

В iOS-приложениях используется асинхронная многопоточная модель. Конечно, как писать код, решать вам. Вы можете выполнять код в одном потоке и синхронно. Но, во-первых, многие системные фреймворки (программные платформы) работают как минимум асинхронно, и вам придется с ними взаимодействовать. А во-вторых, если вы не разделите программу на потоки, ваше приложение будет раздражительно медленным, часто будет игнорировать нажатия пользователя, словом, «лагать».

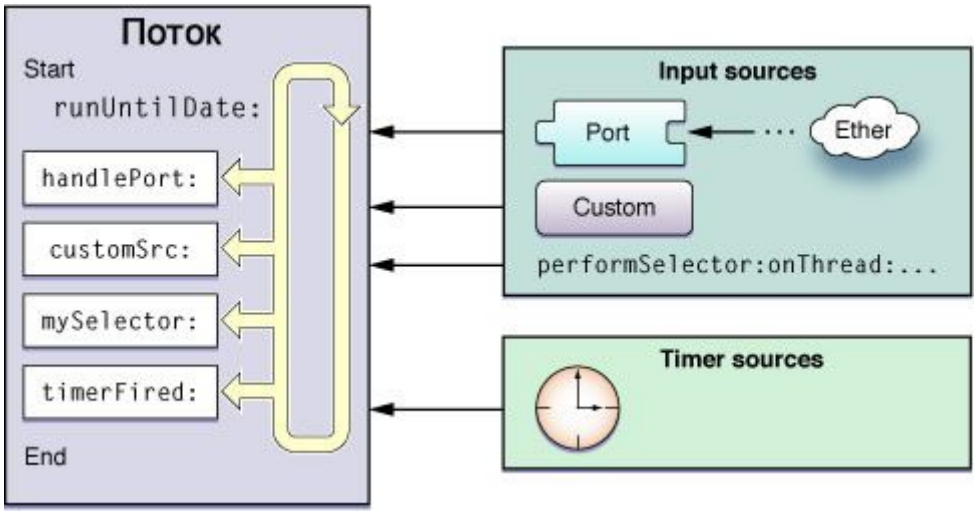
RunLoop

Прежде чем мы начнем говорить, как работать с потоками и асинхронным кодом, познакомимся с RunLoop – сердцем и двигателем вашей программы. RunLoop заставляет весь ваш код не просто выполняться от начала до завершения, а жить в ожидании команд пользователя или иных событий.

RunLoop – часть основной инфраструктуры, связанной с потоками. Это цикл обработки событий, который используются для планирования работы и координации входящих событий. Его цель – держать поток занятым, когда есть работа, и погрузить его в сон, когда ее нет.

Другими словами, ***RunLoop – это бесконечный цикл, который работает в потоке***. На каждой итерации он проверяет, не появилось ли событие, которое следует выполнить. Если такого события нет, RunLoop встает на небольшую паузу, а затем проверяет источники событий повторно.

Вы когда-нибудь думали, что делает запущенное приложение, пока не выполняет никакой работы? Бездействует? Но как же тогда оно узнает, что пора что-то делать, что пользователь нажал на экран, произошло какое-то запланированное событие или пришел ответ от сервера? Ответ прост: оно ждет.



RunLoop – это бесконечный цикл, который ждет событий. Откуда они поступают?

1. **Порты** – через этот источник могут поступать данные извне. Нажатия на экран, системные датчики (например, датчик местоположения), данные из интернета, сообщения от других приложений.
2. **Произвольные источники** – созданные программно и исходящие от другого потока.
3. **Селекторы** – вызов Cocoa-селектора из другого потока.
4. **Таймеры** – отложенные события, созданные потоком для самого себя. Если вы зададите 10-секундный таймер, он не будет тикать непрерывно, а просто разместится в источнике; RunLoop проверит, не пора ли выполнить событие – и в нужный момент сработает. При этом таймер может быть как разовым, так и повторяющимся.

В большинстве случаев вам не нужно волноваться по поводу RunLoop и настраивать его. Например, в главном потоке, который который отвечает за работу пользовательского интерфейса в каждом приложении, уже есть настроенный RunLoop. Многие программисты годами разрабатывают приложения и даже не представляют о его существовании.

Но без понимания принципов работы RunLoop часть логики приложения всегда будет для вас в тумане, так как этот цикл влияет на многие функции приложения: запуск таймеров, обработку интернет-запросов и нажатий. Весь асинхронный код работает на основе RunLoop, и даже освобождение памяти связано с петлей событий. Об этом – подробнее.

Влияние RunLoop на освобождение памяти

Один из аспектов RunLoop связан с освобождением памяти, занимаемой объектами. Ранее мы с вами говорили, что объект стирается из памяти, как только пропадает последняя ссылка на него. Но в некоторых случаях память освобождается не в момент потери последней ссылки, а в конце витка цикла выполнения, на котором она была утеряна.

В большинстве случаев это происходит почти одновременно. Но иногда между двумя этими событиями проходит очень много времени, и память может переполниться. Приложение «падает», программист в замешательстве ищет причину ошибки.

Проведем простой эксперимент. Ниже – код с потенциальной ошибкой переполнения памяти. Добавьте его в приложение, запустите и наблюдайте за объемом занимаемой памяти.

```
print("start test")
for index in 0...UInt.max {
    let string = NSString(format: "test + %d", index)
    print(string)
}
print("end test")
```

Казалось бы, утечек здесь нет – код слишком прост. Создаем в цикле NSString и выводим ее в консоль. В конце каждой итерации цикла NSString должна уничтожаться из памяти, так как исчезает единственная переменная, ссылающаяся на нее.

Но проблема в том, что цикл содержит очень много итераций. Пока он выполняется, петля событий не закончит виток и не очистит всю неиспользуемую память.

NSString – это тип строк из Objective-C и подчиняется он законам ARC из Objective-C. Этот закон гласит, что если имя метода не содержит ключевых слов *new*, *init*, *copy*, *mutableCopy*, объект помещается в специальный контейнер освобождения памяти – **autoreleasepool**.

Он создает свою область видимости, в конце которой происходит уменьшение счетчика ссылок для помещенных в контейнер объектов. По умолчанию в приложении находится *autoreleasepool*, который очищается в конце витка петли событий. Чтобы избежать этой проблемы, необходимо создать свой autoreleasepool, который будет очищаться на каждой итерации цикла.

```
print("start test")
for index in 0...UInt.max {
    autoreleasepool {
        let string = NSString(format: "test + %d", index)
        print(string)
    }
}
print("end test")
```

Влияние RunLoop на асинхронные задачи

Мы уже знаем, что асинхронные задачи можно поставить на паузу в процессе выполнения, а позже возобновить. Но кто решает, в какой момент поставить задачу на паузу, а когда запустить снова?

Это делает сама задача, но приостановить свою работу она может только в определенные моменты. Асинхронные задачи состоят из двух частей: той, которая начинает выполнение задачи, и последующей, которая выполняется асинхронно по наступлении какого-либо события. При этом вторая добавляется в источник для RunLoop и проверяется на каждом витке. Если событие, при котором она должна быть выполнена, наступило, задача выполняется дальше.

Самый простой пример асинхронной задачи – таймер.

```
Timer.scheduledTimer(withTimeInterval: 0.25, repeats: true) { timer in
    print("tick")
}
```

Этот код запускает таймер, который выполняется каждую четверть секунды и выводит в консоль текст *tick*. При этом само создание таймера - это первая часть асинхронной задачи, а блок, переданный методу *scheduledTimer*, - это вторая часть. Задача однопоточная, так как и настройка таймера, и выполнение блока будут проходить на одном и том же потоке. Вместе с тем, задача асинхронная, так как блок будет вызван не сразу, а по истечении некоторого времени, и в этот перерыв поток может выполнять другую работу.

И здесь мы подходим к связи асинхронных задач и RunLoop. Блок таймера будет вызываться каждую четверть секунды, и на каждом витке это условие проверяет RunLoop. Но если поток будет занят выполнением другой длительной задачи и виток не успеет завершиться за 0,25 секунды, чтобы проверить таймер, блок будет вызван с опозданием.

Изменим пример, чтобы в блоке выводилось текущее время.

```
Timer.scheduledTimer(withTimeInterval: 0.25, repeats: true) { timer in
    print(Date())
}
```

И посмотрим, что происходит в консоли:

```
2017-09-26 05:13:16 +0000
2017-09-26 05:13:16 +0000
2017-09-26 05:13:16 +0000
2017-09-26 05:13:16 +0000
2017-09-26 05:13:17 +0000
2017-09-26 05:13:17 +0000
2017-09-26 05:13:17 +0000
2017-09-26 05:13:17 +0000
2017-09-26 05:13:18 +0000
2017-09-26 05:13:18 +0000
2017-09-26 05:13:18 +0000
2017-09-26 05:13:18 +0000
```

Здесь за одну секунду было 4 срабатывания таймера – как мы и ожидали. Добавим в текущий поток еще одну асинхронную задачу, которая будет выполняться раз в секунду и занимать поток ожиданием.

```
Timer.scheduledTimer(withTimeInterval: 0.25, repeats: true) { timer in
    print(Date())
}
Timer.scheduledTimer(withTimeInterval: 1, repeats: true) { timer in
```

```
    sleep(1)
}
```

```
2017-09-26 05:17:33 +0000
2017-09-26 05:17:34 +0000
2017-09-26 05:17:34 +0000
2017-09-26 05:17:34 +0000
2017-09-26 05:17:35 +0000
2017-09-26 05:17:35 +0000
2017-09-26 05:17:36 +0000
2017-09-26 05:17:36 +0000
2017-09-26 05:17:36 +0000
2017-09-26 05:17:37 +0000
2017-09-26 05:17:37 +0000
2017-09-26 05:17:38 +0000
2017-09-26 05:17:38 +0000
2017-09-26 05:17:38 +0000
2017-09-26 05:17:39 +0000
2017-09-26 05:17:39 +0000
2017-09-26 05:17:40 +0000
2017-09-26 05:17:40 +0000
```

Картина вызова первого блока изменилась, промежутки вызова стали неравномерными. Таймер стал срабатывать то 2, то 3 раза в секунду. Это произошло потому, что поток был занят выполнением другого кода и не мог отработать таймер.

Этот пример показывает, что в асинхронном коде нет магии. Он зависит от RunLoop и конкурирует с другими задачами в потоке. Не полагайтесь на чудо при планировании асинхронных задач, а подумайте, смогут ли они выполняться вовремя. Если есть вероятность, что ресурсов потока не хватит для выполнения всех задач, выносите их в отдельные потоки, где ничто не помешает их реализации.

Способы построения многопоточного кода в iOS

В iOS есть три механизма управления потоками:

1. **Thread** – класс, представляющий поток. Позволяет создавать и останавливать потоки. Это фундаментальный инструмент организации многопоточности в iOS.
2. **Grand Central Dispatch (GCD)** – библиотека работы с потоками, построенная поверх Thread. Ее использование абстрагирует разработчика от потоков. Вместо них вы оперируете очередями и блоками кода, которые выполняются в этих очередях.
3. **NSOperation** – еще одна библиотека, более высокого уровня абстракции, чем GCD. В ней используются очереди и специальные классы Operation.

GCD и NSOperation будут рассмотрены в следующих двух уроках, на в этом мы рассмотрим только Thread.

Thread

Фундаментальный способ управления потоками в iOS – использование класса Thread, который представляет собой поток. Последнее время он используется не часто, но понимание принципов его работы необходимо для более эффективного использования остальных технологий, построенных на нем.

Главный поток приложения

Прежде чем перейти к разговору о Thread, разберем единственный поток, без которого приложение не может быть создано – главный, или **main thread**.

Главный поток имеет особую задачу: он обрабатывает пользовательский интерфейс (UI). А значит все, что происходит на экране смартфона, проходит через **main thread**. Ни один другой поток не может работать с UI. И если вы попытаетесь изменить его из второстепенных потоков, в лучшем случае ждите предупреждения, а в худшем – падения вашего приложения. Существует еще и промежуточный вариант: приложение выдаст ошибки, а элементы начнут отображаться неверно.

По умолчанию весь код, что вы пишете, исполняется в главном потоке, но это плохая практика. Чем больше ваши задания отнимают ресурсы главного потока, тем медленнее работает пользовательский интерфейс приложения. Все видели приложения, где прокрутка таблицы и переходы между экранами сопровождаются рывками – это как раз из-за высокой нагрузки на главный поток.

Еще одна особенность главного потока в том, что только в нем запущен RunLoop. Это означает, что во второстепенных потоках не будет работать асинхронный код, и вы не сможете задать в них таймер. Если вам понадобится RunLoop во второстепенном потоке, его нужно запустить.

Создание дополнительных потоков

Теперь мы знаем достаточно, чтобы приступить к созданию своих потоков. Начнем с примера обычного однопоточного кода. Напишем два цикла, оба по 10 итераций. Первый выводит в консоль дьявола, второй ангела.

```
for _ in (0..<10) {
  print("😈")
}

for _ in (0..<10) {
    print("😇")
}
```

После выполнения кода в консоли мы увидим, что сначала вывелись дьяволы, затем ангелы.

```
😈
😈
😈
😇
😇
😇
```

Это ожидаемое поведение. В нашем коде каждый из циклов можно представить как отдельную задачу. В таком случае, сначала выполняется одна задача, а за ней вторая. А если нам нужно выполнить эти задачи одновременно? Для этого надо отправить одну из них в другой поток. Есть несколько способов сделать это, начнем с самого простого.

Самый простой способ выполнить какую-либо задачу в другом потоке – вызвать метод *detachNewThread* класса *Thread* и передать ему замыкание с вашей задачей. При этом немедленно для ее выполнения будет создан новый поток. Минус этого подхода в том, что управлять этим новым потоком вы не сможете, но это и не всегда нужно.

```
Thread.detachNewThread {
    for _ in (0..<10) {
    print("😈")
    }
}

for _ in (0..<10) {
    print("😇")
}
```

В этом примере вывод в консоли изменится.

```
😇
😈
😇
😈
😇
😈
😇
😈
😇
😈
😇
😈
😇
😈
😇
😈
😇
😈
😇
😈
```

Как вы видите, ангелы и демоны чередуются. Это явный признак того, что задачи выполняются одновременно. После выполнения блока кода поток будет закрыт.

Второй способ создать новый поток похож на первый, но уже считается устаревшим – его необходимо использовать только в случае поддержки IOS ниже 10 версии. Это вызов метода *detachNewThreadSelector* класса *Thread* и передача ему селектора на метод какого-либо объекта.

Соответственно, код задания должен быть вынесен в отдельный метод. Его надо пометить ключевым словом @objc, так как вся эта процедура уходит корнями в Objective-C.

```
@objc func printDemon() {
    for _ in (0..<10) {
    print("😈")
    }
}
```

```
Thread.detachNewThreadSelector(#selector(self.printDemon), toTarget: self, with:
nil)

for _ in (0..<10) {
    print("😇")
}
```

Результат при выводе в консоль будет точно таким же, как и в предыдущим случае.

Следующий способ более универсален. Мы создадим экземпляр класса *Thread*, используя конструктор, принимающий блок кода. При этом у нас будет на руках объект потока, которым можно управлять. Важная деталь этого подхода: поток запускается вручную с использованием метода *start*.

```
let thread1 = Thread {
    for _ in (0..<10) {
    print("😈")
    }
}

thread1.start()

for _ in (0..<10) {
    print("😇")
}
```

Также можно воспользоваться **устаревшим вариантом этого метода** – создать объект потока через конструктор с селектором.

```
let thread1 =Thread(target: self, selector: #selector(self.printDemon), object:
nil)
```

Последний способ – создание собственного подкласса потока. При этом надо переопределить метод *main*, так как именно он будет вызван при старте потока. Этот способ стоит использовать для инкапсуляции сложной логики, чтобы сделать программу более читаемой.

```
class ThreadprintDemon: Thread {
    override func main() {
        for _ in (0..<10) {
            print("😈")
        }
    }
}
```

```
let thread1 = ThreadprintDemon()

thread1.start()

for _ in (0..<10) {
    print("😇")
}
```

Управление потоком

Бывают случаи, когда недостаточно просто выполнить код в отдельном потоке, а необходимо управлять им, запускать и отменять его выполнение .

Самое простое, что вы можете сделать с потоком – запустить его. Без этого он не выполнит никакой полезной работы.

```
let thread1 = ThreadprintDemon()
thread1.start()
```

Поток запустится и начнет выполнение метода *main*. Иногда новички допускают ошибку: создают поток и вызывают у него метод main, вместо *start*. В этом случае метод будет выполнен в том потоке, в котором вызван без создания отдельного. Будьте внимательнее.

Второе возможное действие с потоком – отмена. Для этого вызывается метод *cancel*.

```
let thread1 = ThreadprintDemon()
thread1.start()
thread1.cancel()
```

Этот метод служит причиной многочисленных ошибок у новичков. Дело в том, что он не останавливает поток, а только дает команду остановки, которую разработчик должен сам обработать в процессе выполнения потока.

Давайте создадим поток, который будет выполнять бесконечную задачу, и поэкспериментируем с ним.

```
class InfintyLoop: Thread {
    override func main() {
        while true {
            print("😈")
        }
    }
}
```

Создадим экземпляр потока, запустим и сразу остановим его.

```
let thread1 = InfintyLoop()
thread1.start()
thread1.cancel()
```

Именно так новички обычно проверяют работу метода *cancel*, и тут их поджидает первый подводный камень. В результате выполнения этого кода в консоли не будет никаких сообщений: поток остановится, и мы будем думать, что метод *cancel* действительно останавливает поток. Но на самом деле при вызове *cancel* сразу после *start* поток даже не начнет выполняться.

Изменим код, добавим паузу в 2 секунды между запуском и остановкой. За это время поток успеет запуститься.

```
let thread1 = InfintyLoop()

thread1.start()
sleep(2)
thread1.cancel()
```

Видите, демоны наполнили нашу консоль и не думают останавливать свою экспансию. Метод остановки не сработал. Давайте разберемся, почему. В этом нам помогут ***флаги выполнения потока:***

1. **isExecuting** – указывает, выполняется ли поток.
2. **isCancelled** – указывает, был ли поток остановлен.
3. **isFinished** – указывает, был ли поток завершен.

Может показаться, что три флага – это слишком много, ведь все они указывают на состояние потока. На самом деле это не так. Первый флаг *isExecuting* – единственный, который достоверно отображает текущее состояние потока. Второй – *isCancelled* – программист может изменять сам, при этом состояние потока может и не соответствовать флагу. Третий – i*sFinished* – указывает, был ли завершен поток.

Например, поток, который не начал выполняться, имеет состояние *isExecuting – false,* но *isFinished* – тоже *false*, так как и завершен он также еще не был.

Как использовать эти флаги для остановки потока? Надо отслеживать в коде потока флаг *isCancelled* – если он станет *true*, мы просто остановим выполнение задания. В этом примере можем

заменить условие цикла *while*, прописав вместо *true* – *!isCancelled*. Так цикл будет выполняться только до тех пор, пока его не остановят.

```
class InfintyLoop: Thread {
    override func main() {
        while !isCancelled {
            print("😈")
        }
    }
}
```

Теперь пример запуска остановки будет работать, как запланировано. Это дает определенную гибкость: если ваш поток выполняет важную задачу, вы сможете не останавливать ее сразу по получении команды, а позволить ей корректно завершиться.

Остальные флаги носят информационный характер, и применять их внутри потока нет смысла. Можно использовать их снаружи, чтобы посмотреть его состояние.

Приоритет выполнения потоков

Потоки помогают разделять выполнение задач. Создание дополнительных потоков снимает нагрузку с главного и предотвращает рывки пользовательского интерфейса. Но простым вводом нескольких потоков эти задачи не всегда решаются. Вы можете создать несколько сотен потоков, но все равно будете ограничены ресурсами устройства, на котором выполняется приложение. Так что, даже если вынести логику в отдельный поток, он может забирать ресурсы у главного.

Именно для этого существует ***инструмент управления приоритетами потоков***. С его помощью можно, например, вынести в отдельный поток такую ресурсоемкую операцию, как наложение эффектов на изображение и понизить приоритет. И если в главном потоке будет выполняться операция, требующая ресурсов процессора, он их получит и не будет застывать, а изображения будут обработаны, когда появятся свободные ресурсы.

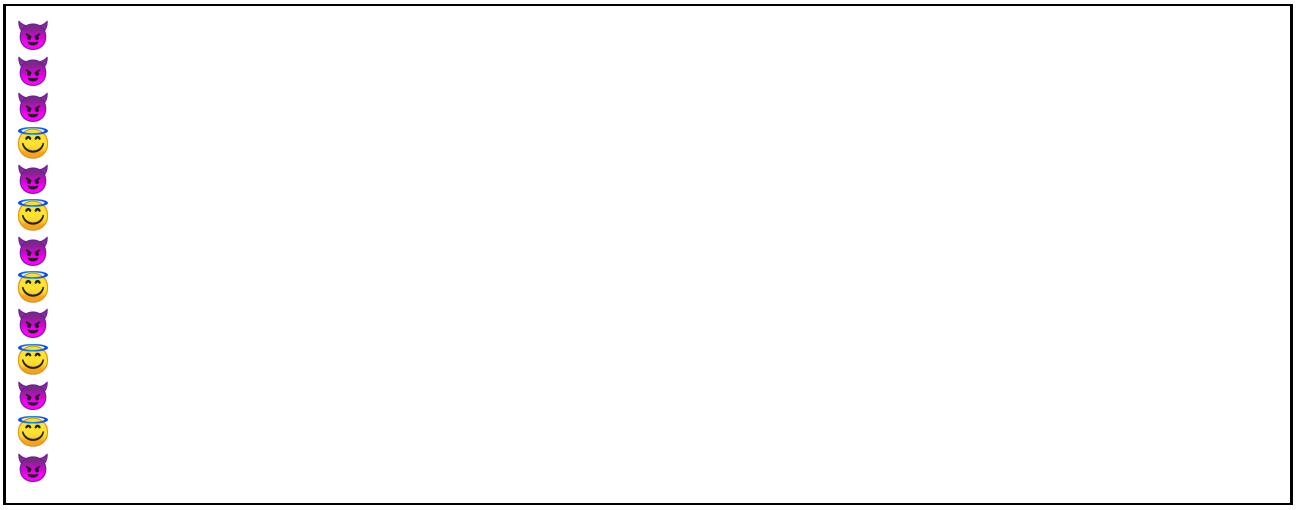
Напишем новый пример: сделаем два потока с циклами вывода сообщений в консоль. При этом увеличим число итераций до 100, чтобы пример был более выразительный.

```
class ThreadprintDemon: Thread {
    override func main() {
        for _ in (0..<100) {
            print("😈")
        }
    }
}
class ThreadprintAngel: Thread {
    override func main() {
        for _ in (0..<100) {
            print("😇")
        }
    }
}
```

Запустим оба потока одновременно. Мы увидим, что демоны и ангелы чередуются. В самом начале демонов может быть больше - это связано с тем, что поток с ними был запущен раньше.

```
let thread1 = ThreadprintDemon()
let thread2 = ThreadprintAngel()

thread1.start()
thread2.start()
```



Изменим приоритеты для потоков. Есть 5 приоритетов:

1. **UserInteractive** – самый высокий приоритет. Используется, когда результат нужен как можно раньше, желательно – прямо сейчас.
2. **UserInitiated** – пониженный приоритет. Применяется, когда результат важен, но допустима небольшая задержка.
3. **Utility** – почти минимальный приоритет – для ситуаций, когда результат может подождать.
4. **Background** – самый минимальный приоритет, используется для фоновых задач.
5. **Default** – приоритет по умолчанию, выбирается между **userInitiated** и **utlility.**

Назначим потоку с демонами низкий приоритет (**utility**), а потоку с ангелами самый высокий (**userInteractive).**

```
let thread1 = ThreadprintDemon()
let thread2 = ThreadprintAngel()

thread1.qualityOfService = .utility
thread2.qualityOfService = .userInteractive

thread1.start()
thread2.start()
```

Результат в консоли кардинально изменился. Теперь почти все ангелы в начале, а демоны в конце. Дело в том, что все ресурсы отдавались потоку с ангелами, а поток с демонами ждал, пока они освободятся. Это очень хорошо демонстрирует приоритеты, но мы рассмотрели ситуацию, когда потоки очень прожорливы, а обычный цикл стремится получить все доступные ресурсы.

Добавим в цикл паузу: при этом потоку понадобится очень мало процессорных ресурсов на выполнение.

```
class ThreadprintDemon: Thread {
    override func main() {
        for _ in (0..<100) {
            print("😈")
            Thread.sleep(forTimeInterval: 1)
        }
    }
}

class ThreadprintAngel: Thread {
    override func main() {
        for _ in (0..<100) {
            print("😇")
            Thread.sleep(forTimeInterval: 1)
        }
    }
}
```

```
let thread1 = ThreadprintDemon()
let thread2 = ThreadprintAngel()

thread1.qualityOfService = .utility
thread2.qualityOfService = .userInteractive

thread1.start()
thread2.start()
```

Несмотря на то, что приоритеты потоков по-прежнему разные, вывод в консоль вернулся к виду, когда ангелы и демоны чередуются. Так происходит потому, что ресурсов достаточно, чтобы выполнять оба потока одновременно.

RunLoop в потоках

По умолчанию RunLoop есть только в главном потоке, в остальных, созданных вами, он отсутствует. Значит, в потоках не работает асинхронный код, таймеры и, к примеру, уведомления Realm. При этом любой поток без петли событий завершится, как только выполнится его код. Но как поступить, если необходимо добавить в потоки таймеры или не дать им закрываться на протяжении длительного времени? Создать и запустить в потоках RunLoop.

RunLoop создается в потоке, как только вы попытаетесь к нему обратиться. Самый простой способ обратиться к петле событий – попытаться вывести на экран текущую петлю.

```
print(RunLoop.current)
```

После этого RunLoop будет создан, даже если его там не было. Но он находится в неактивном состоянии. Для запуска есть метод *Run*.

```
RunLoop.current.run()
```

Код выше создает RunLoop, если его нет, и запускает его. После этого петля событий оживит поток, и он никогда не закроется. После запуска петли действия в потоке будут выполнены, только если они находятся в источниках RunLoop. Значит, добавлять что-то в эти источники, то есть запускать асинхронный код, надо либо до запуска петли событий, либо из другого потока.

Разберем простой пример с таймером. Он не будет работать, так как в потоке нет петли событий.

```
class TimeThread: Thread {
    override func main() {
        // Настраиваем таймер
        Timer.scheduledTimer(withTimeInterval: 0.25, repeats: true) { timer in
            print("Tick")
        }
    }
}
```

Добавим петлю – таймер заработает.

```
class TimeThread: Thread {
    override func main() {
        // Настраиваем таймер
        Timer.scheduledTimer(withTimeInterval: 0.25, repeats: true) { timer in
            print("Tick")
        }
        // Запускаем петлю
        RunLoop.current.run()
    }
}
```

Важно: в примере таймер добавлен до запуска петли событий. После RunLoop заблокирует поток своим циклом, и дальше этого выполнение кода не пойдет.

Вот так таймер работать не будет:

```
class TimeThread: Thread {
    override func main() {
        // Запускаем петлю
        RunLoop.current.run()
        // Настраиваем таймер
        Timer.scheduledTimer(withTimeInterval: 0.25, repeats: true) { timer in
            print("Tick")
        }
    }
}
```

Поток, запущенный таким способом, никогда не остановится. Если остановка требуется, можно запустить RunLoop на какое-то время. Например, на 20 секунд:

```
RunLoop.current.run(until: Date() + 20)
```

Петля событий будет запущена, но по прошествии 20 секунд поток завершится. Можно установить любую дату в качестве ограничения и даже запустить петлю событий на неделю.

С помощью метода *cancel* остановить петлю событий несколько сложнее. Как правило, создается бесконечный цикл на основе *while*, который запускает RunLoop на одну секунду, если поток не остановлен.

```
while !isCancelled {
    RunLoop.current.run(until: Date() + 1)
}
```

Цикл *while* будет выполняться до тех пор, пока поток не получит команду *cancel* и его флаг не станет истинным. А внутри этого цикла запускается петля ровно на одну секунду. Затем она останавливается, и цикл проверяет, не было ли команды остановить поток. Если нет, петля будет запущена вновь.

Практическое задание

На этом курсе вы добавите в приложение, разработанное на прошлом уроке, ленту новостей. Лента будет разрабатываться весь курс, и в процессе изучения новых технологий вы будете улучшать ее производительность.

1. Добавить еще одну вкладку в приложение.
2. На новой вкладке добавить UITableViewController для отображения ленты новостей.
3. В UITableViewController добавить прототип ячейки для новости типа post.
4. *Добавить также прототип ячейки для новости типа photo.

Ячейки обязательно должны содержать автора новости, его аватар, количество лайков и комментариев, репостов и просмотров.

Ячейка поста должна содержать текст поста.

* Ячейка фото должна содержать фото.

Доставать фото из вложений не надо.

Пока можно обойтись примерными прототипами и доработать их позднее.

Высота ячеек должна быть стандартна. Текст должен прокручиваться, если его много.

Дополнительные материалы

1. Многопоточность
2. Параллелизм
3. Поток выполнения

4. Процесс
5. Различия асинхронной и многопоточной архитектуры
6. Параллелизм против многопоточности против асинхронного программирования: разъяснение
7. Цикл событий
8. RunLoop
9. RunLoop на русском

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html
2. https://developer.apple.com/documentation/foundation/thread