Objective-C

Курс лекций и семинаров для студентов,
желающих научиться программировать под iPhone

Осень-Зима 2013

Лекция №9

Multithreading: Threads, Operations, Blocks.

Автор: Дмитрий Волков, iPhone Developer, Sibers

Sibers®

Что вы узнаете сегодня?

- Multithreading
- NSOperation / NSOperationQueue
- Synchronization
- Memory management
- GCD
- Blocks

Multithreading: Technologies

iOS предоставляет несколько технологий для создания и управления потоками:

- **POSIX** — pthread и прочее
- **GCD** — Grand Central Dispatch.
- **NSOperationQueue / NSOperation.**
- **NSThread**.
- **NSRunloop**.

Multithreading: Main thread

- В iOS любое приложение всегда имеет несколько потоков.
- Рендерингом UI на экране занимается главный поток: ***Main thread***.

ВАЖНО: Любое действие с UI, например, изменение размера или цвета ВСЕГДА должно выполняться на главном потоке.

- ***Main thread*** существует всегда, на протяжении всей жизни приложения.

ВАЖНО: Никогда не блокируйте Main thread какими-либо долгими операциями.

Multithreading: Who is who

- Кроме главного потока, вы также можете создавать свои второстепенные потоки для выполнения каких-либо долгих операций.
- Многие системные компоненты сами создают потоки для выполнения своих задач.

ВАЖНО: Некоторые системные компоненты посылают сообщения своему делегату либо оповещают своих наблюдателей не на главном потоке.

- **Cocoa** легко позволяет наладить коммуникацию между потоками — вызывать методы на практически любом потоке и из любого потока.

Multithreading: Cocoa way

В целом, при создания приложений для iOS и Mac OS X, не рекомендуется самим создавать и управлять потоками, так как, скорее всего, это приведет к большому количеству ошибок.

Cocoa предоставляет технологии, позволяющие абстрагироваться от потоков и реализовывать задачи с использованием более высокоуровневых парадигм:

- Представлять обработку данных в виде отдельной операции (task)
- Использовать асинхронный подход для запросов обработки данных.

Многие объекты в **Cocoa** используют асинхронные вызовы для обработки и получения данных.

Multithreading: Run Loops

В Сосоа для каждого потока системой обычно создается свой **Run Loop** — цикл, который обрабатывает таймеры и события, а так же усыпляет поток, если ему нечего делать в текущий момент.

Run Loop поддерживает 2 типа событий:

- **Input sources** — асинхронные события. Обычно это сообщения от других потоков, приложений или системных вызовов.
- **Timer sources** — синхронные события. Таймеры. Вызываются синхронно с известным интервалом.

Каждый **Run Loop** определяет режим, в котором он работает: от режима зависит, какие события будут обработаны и кто будет об этом оповещен.

Multithreading: NSThread

NSThread — объект, описывающий поток. Выполняет заданный метод заданного объекта на второстепенном потоке.

- Ничего особенного, объектная обертка над POSIX-потоками.
- Кроме объекта, чей метод будет выполнен, также позволяет задавать размер стека, приоритет, а так же **NSDictionary** произвольных значений для каждого отдельного потока.
- Хотя, скорее всего, является технологией, которой пользуются более высокоуровневые компоненты, на данный момент **NSThread** напрямую не используется.

NSThread: Example

```
//convenience
[NSThread detachNewThreadSelector:@selector(compute42) target:self object:nil]

NSThread* thread = [[NSThread alloc] initWithTarget:self
action:@selector(compute42) object:nil];
[thread start];
//suspending thread
NSThread* currentThread = [NSThread currentThread];
[currentThread sleepForTimeInterval:5];
//obtaining main thread
NSThread* mainThread = [NSThread mainThread];
```

NSOperation / NSOperationQueue

NSOperation — объект, описывающий часть задачи, которую надо выполнить.

- Абстрагирует конкретную реализацию того, где будет выполнена задача. Вы задаете метод, который следует выполнить и отправляете задачу на исполнение.

NSOperation = single-shot + fire-and-forget

NSOperation — абстрактный класс. Для запуска операций используются его более конкретизированные наследники.

- Обычно используется в связке с **NSOperationQueue**.

NSOperation / NSOperationQueue

- При использовании **NSOperation**, вы сами можете решать, будет ли она выполняться асинхронно в отдельном потоке, либо синхронно в текущем.
- Также **NSOperation** позволяет создавать зависимости между различными задачами, выстраивая более сложную логику исполнения.
- Позволяет выставлять приоритет задачи.
- Позволяет выставлять блок кода, который выполняется по окончании выполнения основной задачи.
- Позволяет добавлять наблюдателей за текущим состоянием задачи (исполняется, отменена, закончена, готова к исполнению).

NSOperation / NSOperationQueue

NSOperationQueue — объект, управляющий очередью выполнения объектов **NSOperation**, добавленных в данную очередь.

- **NSOperationQueue** управляет планированием добавленных в нее операций.
- **NSOperationQueue** планирует, в каком потоке (и на каком ядре, если процессор многоядерный) будет выполнена операция.
- Для **NSOperationQueue** можно задать, сколько операций может выполняться одновременно. Так же позволяет отменять все добавленные операции и останавливать / запускать их.

NSOperation

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

[operation setThreadPriority:0.5]

[operation start];


NSBlockOperation* blockOperation = [NSBlockOperation
blockOperationWithBlock: ^{

int answer = 42;}];

[blockOperation addDependency: operation];

[blockOperation start];
```

NSOperation+NSOperationQueue

```
NSOperationQueue* queue = [NSOperationQueue new];
[queue setSuspended:YES];
NSOperation* operation = ...;
NSOperation* operation2 = ...;
[queue addOperation: operation];
[queue addOperation: operation2];
[queue setMaxConcurrentOperationCount:2];
[queue setSuspended:NO];
[queue waitUntilAllOperationsAreFinished];
```

NSObject threading categories

```
NSObject* object = ...;

[object performSelectorInBackground:@selector(doStuff) withObject:nil];


[object performSelectorOnMainThread:@selector(doStuff) withObject:nil
waitUntilDone:NO];


[object performSelector:@selector(doStuff) onThread:[NSThread
mainThread] withObject: nil waitUntilDone:NO];


[NSObject cancelPreviousPerformRequestsWithTarget: object];
```

Synchronization

Так как в приложении различные потоки обычно разделают одну и ту же область памяти (хотя стек у каждого свой), необходима синхронизация доступа к разделяемым ресурсам.

Cocoa предоставляет объекты-обертки над примитивами синхронизации POSIX:

- NSLock
- NSRecursiveLock
- NSConditionLock
- NSCondition

Synchronization: Locks

```
NSLock* lock = [NSLock new];
BOOL canLock = [lock tryLock];
[lock lock];
....
[lock unlock];

NSRecursiveLock* recursiveLock = [NSRecursiveLock new];
[recursiveLock lock];
...
[recursiveLock unlock];
```

Synchronization

Так же Cocoa предоставляет более высокоуровневые механизмы синхронизации:

- Atomic properties - @property(copy) NSString* name;
- Ключевое слово @synchronize - @synchronize(self) {…}

@synchronize создает мьютексы, позволяющие только одному объекту выполнять код в блоке синхронизации. Передаваемый параметр используется как ключ для определения уникальности мьютекса.

Synchronization: Atomic property

```
- (NSString*) setName:(NSString*) name
{
    @synchronized(self)
    {
        _name = name;
    }
}
```

Synchronization: Atomic property

```
- (NSString *) name
 {
  NSString *retval = nil;
  pthread_mutex_t *self_mutex = LOOK_UP_MUTEX(self);
  pthread_mutex_lock(self_mutex);
  retval = [[_name retain] autorelease];
  pthread_mutex_unlock(self_mutex);
  return retval;
}
```

Memory management

- Важно следить за временем жизни и счетчиком ссылок объектов, доступ к которым могут получать несколько объектов.
- Также, для каждого отдельного потока требуется создавать свой **autorelease pool**.
- Создавать **autorelease pool** требуется, если вы наследуете NSThread и переопределяете метод main
- Объектам, с которыми вы собираетесь работать из нескольких потоков следует послать сообщение retain перед их получением и release после окончания работы с ними.

Memory management

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;

[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[arrayLock unlock];

[anObject doSomething];
```

Memory management

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;

[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[anObject retain];
[arrayLock unlock];

[anObject doSomething];
[anObject release];
```

GCD

- Технология, разработанная Apple для упрощения управлением асинхронными / параллельными задачами.
- Использует паттерн **Thread Pool**
- Доступен, начиная с Mac OS X 10.6 и iOS 4.0
- Если приложение создается с OS version 10.8+ или 6.0+, то ARC так же может управлять dispatch_ типами, как и обычными объектами.

Source code:
http://www.opensource.apple.com/source/libdispatch/libdispatch-228.23/

GCD

- **Blocks** (dispatch_block_t) — блоки, нестандартизированное расширение языка C/C++/Objectve-C/C++ от компании Apple.
- **dispatch_queue** — абстракция очереди заданий, принятых на выполнение
- **dispatch_source** — абстракция системных элементов, за которыми мы можем наблюдать: файлы, процессы, порты и многое другое.
- **dispatch_data** — обертка над управлением потоками данных: запись / чтение / копирование.

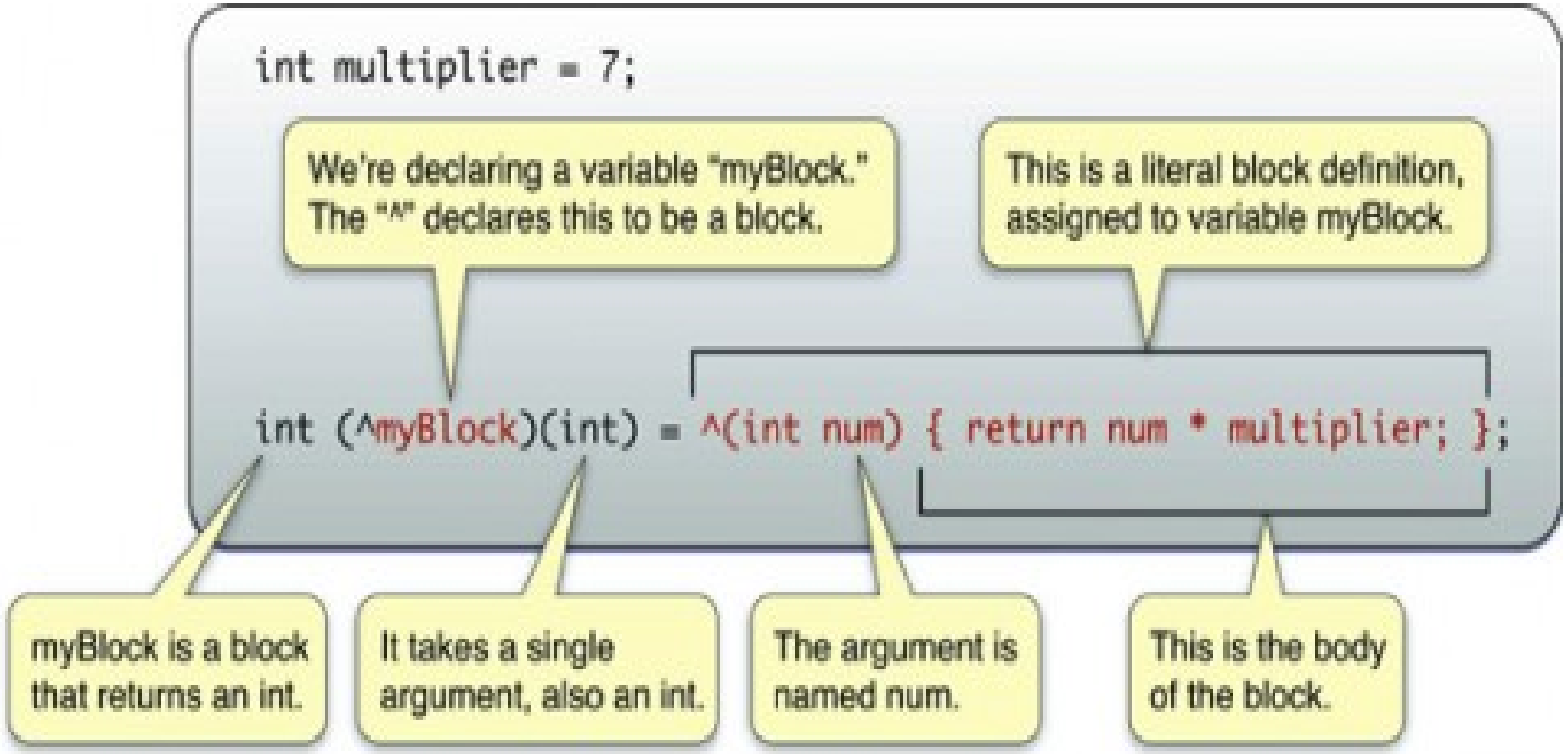
GCD: Blocks

Блоки — нестандартизированное расширение языка C/C++/Objective-C/C++ от компании Apple

- Являются особенностью реализации компилятора, а не языка / фреймворков.
- Собратья замыканий и лямбд — могут «захватывать» значения переменных, окружающих блок.
- Имеют встроенный механизм управления памятью захваченных объектов.
- Обширно используются в большинстве классов Foundation фреймворка

https://developer.apple.com/library/IOS/documentation/Cocoa/Conceptual/Blocks/Articles/bxGettingStarted.html

GCD: Blocks



GCD: Blocks

```
int multiplier = 7;
int (^myBlock)(int) = ^(int num) {
    return num * multiplier;
};

printf("%d", myBlock(3));
// prints "21"
```

GCD: Blocks

Блоки могут управлять жизненным циклом захваченных объектов:

- Для **не-объектов** создаются константные копии.
- Захваченным **объектам-наследникам от NSObject (CF объекты не учитываются)**, либо **объявленными с аттрибутом __attribute__((NSobject))** будет послано сообщение retain при создании блока и release, когда блок будет уничтожен. Все указатели внутри блока при этом считаются константными.
- При указании переменной как **__block** – она будет доступна для изменения в блоке, а так же к ней не будут применяться retain / release сообщения.

GCD: Blocks

Жизненный цикл блоков:

- Блоки всегда создаются на **стеке** (NSConcreteStackBlock)
- Но! Если блок не захватывает каких-либо переменных, то он может быть оптимизирован компилятором: инициализируется классом **NSConcreteGlobalBlock** и становится глобальной переменной.

После создания, блок должен быть скопирован в кучу:

- Макросы ***Block_copy() / Block_release()***
- Так как блок Objectve-C — объект, то ему напрямую можно посылать сообщения ***copy / release / autorelease***.
- !!!Посылка сообщения retain никак не влияет на место жительства блока!!!

GCD: Blocks

- ВАЖНО: так как блоки являются объектами, их можно хранить в переменных и в property классов.
- Если вы определяете блок как property, ВСЕГДА используйте модификатор copy:

typedef (void)(^CompletionBlock)(void);

@property(nonatomic, copy) CompletionBlock block;

GCD: Blocks

Блоки посылают retain сообщения всем объектам, которые они захватывают — ***self*** в том числе:

Явно – вызываем self в блоке:

```
^ { [self doStuff]; };
```

Неявно – обращение к переменным, членам класса:

```
^ { myIvar = 42; }; // неявно преобразуется к self->myIvar = 42
```

GCD: Blocks

Чтобы избежать этого, объявим временную переменную, которая и будет использована в блоке:

```
__block typeof(self) weakSelf = self;//non-ARC
__weak __typeof(self)  weakSelf = self; //ARC 5.0+
__unsafe_unretained __typeof(self) weakSelf = self;//ARC pre-5.0
^ { [weakSelf doStuff]; };
^ { weakSelf->myIvar = 42; };
```

Blocks: Usage

```
typedef (void)(^CompletionBlock)(NSData* data, NSError* error);

@interface DataDownloader : NSObject

+ (void) downloadDataFromURL:(NSURL*) url completionBlock:(CompletionBlock) completionBlock;

@end
```

Blocks: An example

```
@interface DataDownloader ()<NSURLConnectionDelegate,
NSURLConnectionDataDelegate>
{
    NSURLConnection* _connection;
    NSMutableData* _downloadedData;
}
@property(nonatomic, copy) DataDownloaderCompletionBlock
completionBlock;


@end
```

Blocks: An example

```
@implementation DataDownloader

+ (instancetype) downloadDataFromURL:(NSURL*) url
completionBlock:(DataDownloaderCompletionBlock) completionBlock

{

    return [[self alloc] initWithURL:url CompletionBlock:completionBlock];

}

- (id) initWithURL:(NSURL*) url CompletionBlock:(DataDownloaderCompletionBlock)
completionBlock

{

    self = [super init];

    _downloadedData = [NSMutableData new];

    self.completionBlock = completionBlock;

    NSURLRequest* request = [NSURLRequest requestWithURL:url];

    _connection = [NSURLConnection connectionWithRequest:request delegate:self];

    return self;

}
```

Blocks: An example

```
#pragma mark - NSURLConnection delegate methods
- (void) connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    NSLog(@"%@",error);
    self.completionBlock(nil, error);
    self.completionBlock = nil;
}
- (void) connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    [_downloadedData appendData:data];

}
```

Blocks: An example

```
- (void) connectionDidFinishLoading:(NSURLConnection *)connection
{

    self.completionBlock(_downloadedData, nil);
    self.completionBlock = nil;

}

@end
```

Blocks: An example

```
DataDownloader* downloader = [DataDownloader
downloadDataFromURL:[NSURL
URLWithString:@http://www.kittens.com/kitten.png]
completionBlock:^(NSData* data, NSError* error){
    if(!error)
    {
        UIImage* image = [UIImage imageWithData:data];
    }
}];
```

Заключение

iOS предоставляет множество технологий для создания многопоточных приложений.

Доступно все, начиная от примитивных POSIX-потоков, и заканчивая высокоуровневыми классами, абстрагирующими от вас потоки как таковые.

Многие API предоставляют как синхронные, так и асинхронные методы обработки и получения данных.

Так же, в C/Objective-C добавлено нестандартизированное расширение языка

C — блоки, позволяющие намного удобнее писать асинхронный код.

Спасибо за внимание!

Жду ваших вопросов.