

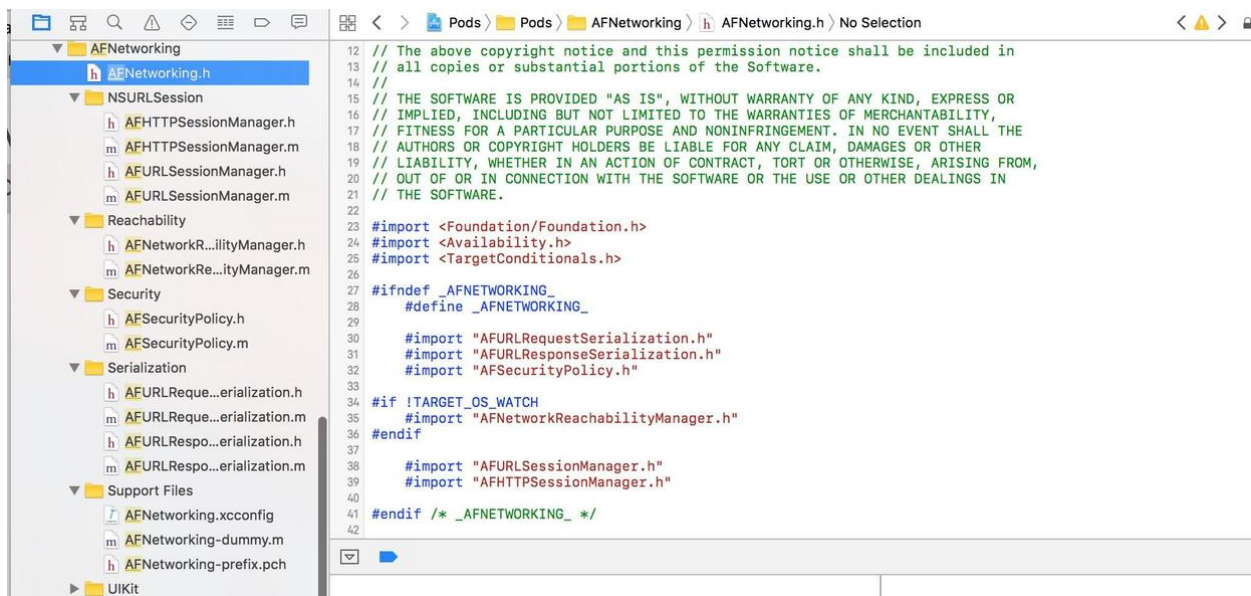
# AFNetworking 源码分析

作者：@涂耀辉

- 作为一个iOS开发，也许你不知道 `NSURLRequest`、不知道 `NSURLConnection`、也不知道 `NSURLSession`。但是你一定知道 `AFNetworking`。
- 大多数人习惯了只要是请求网络都用AF，但是你真的知道AF做了什么吗？为什么我们不用原生的 `NSURLSession` 而选择 `AFNetworking`？
- 本文将从源码的角度去分析 `AF` 的实际作用。或许看完这篇文章，你心里会有一个答案。

## 一. AF3.X的网络请求主流程

首先，我们就一起分析一下该框架的组成。将AF下载并导入工程后，下图为框架源码整个结构：

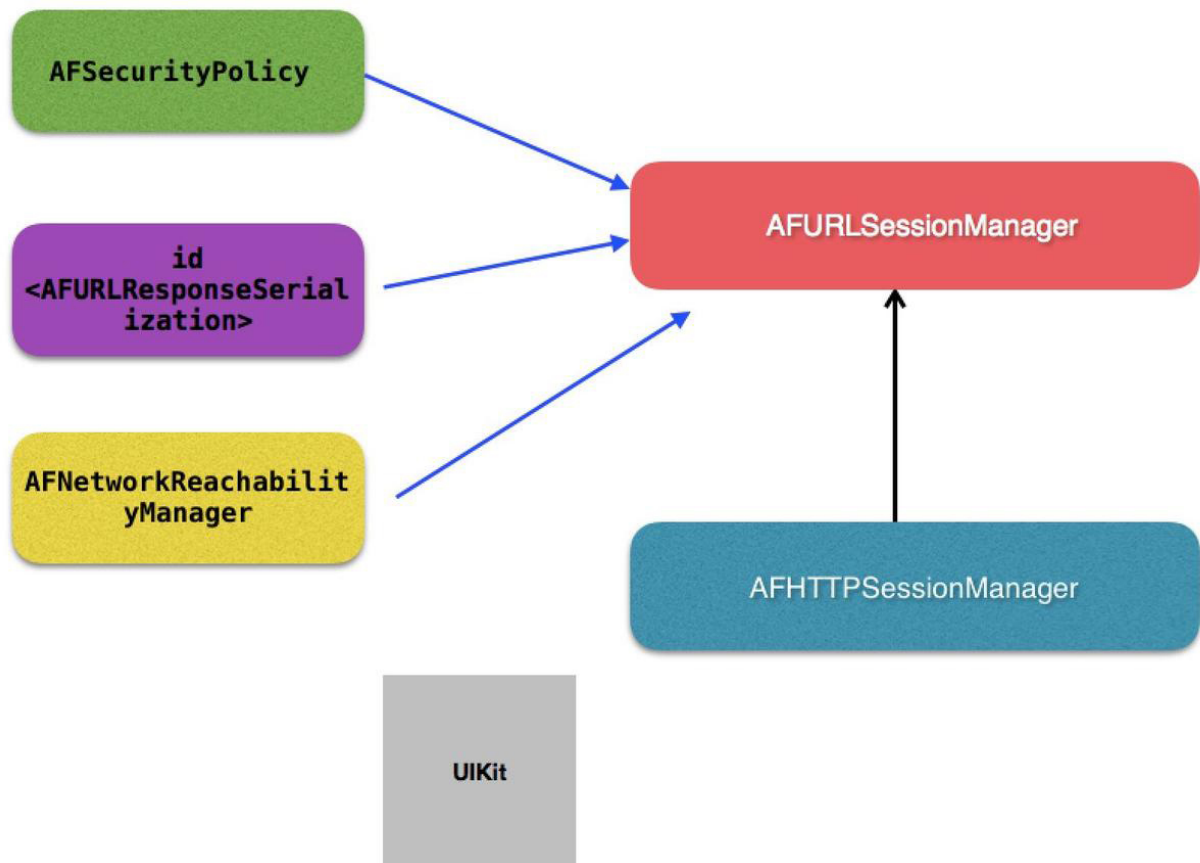


除去Support Files，可以看到AF分为如下5个功能模块：

- 网络通信模块(`AFURLSessionManager`、`AFHTTPSessionManger`)
- 网络状态监听模块(`Reachability`)
- 网络通信安全策略模块(`Security`)
- 网络通信信息序列化/反序列化模块(`Serialization`)
- 对于iOS UIKit库的扩展(`UIKit`)

其核心为网络通信模块`AFURLSessionManager`。

大家都知道，AF3.x是基于 `NSURLSession` 来封装的。所以这个类围绕着 `NSURLSession` 做了一系列的上层封装。而其余的四个模块，均是为了配合 `AFURLSessionManager` 类的网络通信做一些必要的处理工作。如结构关系图如下所示：



其中AFHTTPSessionManager是继承于AFURLSessionManager的，我们一般做网络请求都是用这个类，但是它本身是没有做实事的，只是做了一些简单的封装，把请求逻辑分发给父类AFURLSessionManager去做。

接着我们从源码的角度一个个模块去解读它的作用：

## 1.对外接口类：AFHTTPSessionManager

这个类提供了对外调用的接口，当我们初始化AF实例时候，都是使用这个类的这几个方法：

```
+ (instancetype)manager;
- (instancetype)initWithBaseURL:(nullable NSURL *)url;
- (instancetype)initWithBaseURL:(nullable NSURL *)url
    sessionConfiguration:(nullable NSURLSessionConfiguration
*)configuration;
```

而发起网络请求，都是调用该类以下的方法：

```

- (nullable NSURLSessionDataTask *)GET:(NSString *)URLString
    parameters:(nullable id)parameters
    success:(nullable void (^)(NSURLSessionDataTask *task,
id _Nullable responseObject))success
    failure:(nullable void (^)(NSURLSessionDataTask *
_Nullable task, NSError *error))failure DEPRECATED_ATTRIBUTE;
- (nullable NSURLSessionDataTask *)POST:(NSString *)URLString
    parameters:(nullable id)parameters
    success:(nullable void (^)(NSURLSessionDataTask *task,
id _Nullable responseObject))success
    failure:(nullable void (^)(NSURLSessionDataTask *
_Nullable task, NSError *error))failure DEPRECATED_ATTRIBUTE;

//还有其他的好几个方法，不一一举出了...

```

我们之前说过了这个类不做实事，只是提供对外接口，真正的初始化，以及网络请求，都是交给父类 `AFURLSessionManager` 去做的。

- 类似初始化方法中：

```

- (instancetype)initWithBaseURL:(NSURL *)url
    sessionConfiguration:(NSURLSessionConfiguration *)configuration
{
    self = [super initWithSessionConfiguration:configuration];
    if (!self) {
        return nil;
    }
    //....
    return self;
}

```

- 类似网络请求方法中，调用父类方法拿到 `task`，这个类仅仅是把得到的 `task`，`resume` 即可：

```

dataTask = [self dataTaskWithRequest:request
            uploadProgress:uploadProgress
            downloadProgress:downloadProgress
            completionHandler:^(NSURLResponse * __unused response,
id responseObject, NSError *error) {
    if (error) {
        if (failure) {
            failure(dataTask, error);
        }
    } else {
        if (success) {
            success(dataTask, responseObject);
        }
    }
}];

```

当然这个类还做了一件很重要的事，就是把传过来的参数，编码成我们请求时需要的 `request`，并且传给父类去做网络请求：

```

NSError *serializationError = nil;

//把参数，还有各种东西转化为一个request
NSMutableURLRequest *request = [self.requestSerializer
requestWithMethod:method urlString:[NSURL URLWithString:urlString
relativeToURL:self.baseURL] absoluteString] parameters:parameters
error:&serializationError];

```

接着我们去看看这个 `requestSerializer` 所属类是如何将参数转换成 `request` 的。

## 2.请求参数解析类：AFHTTPRequestSerializer

下面这个方法就是这个类对外的核心方法：

```

- (NSMutableURLRequest *)requestWithMethod:(NSString *)method
                                urlString:(NSString *)URLString
                                parameters:(id)parameters
                                error:(NSError *__autoreleasing *)error
{
    //断言, debug模式下, 如果缺少改参数, crash
    NSParameterAssert(method);
    NSParameterAssert(URLString);

    NSURL *url = [NSURL URLWithString:URLString];

    NSParameterAssert(url);

    NSMutableURLRequest *mutableRequest = [[NSMutableURLRequest alloc]
initWithURL:url];
    mutableRequest.HTTPMethod = method;

    //将request的各种属性循环遍历
    for (NSString *keyPath in AFHTTPRequestSerializerObservedKeyPaths()) {
        //如果自己观察到的发生变化的属性, 在这些方法里
        if ([self.mutableObservedChangedKeyPaths containsObject:keyPath]) {
            //把给自己设置的属性给request设置
            [mutableRequest setValue:[self valueForKeyPath:keyPath]
forKey:keyPath];
        }
    }
    //将传入的parameters进行编码, 并添加到request中
    mutableRequest = [[self requestBySerializingRequest:mutableRequest
withParameters:parameters error:error] mutableCopy];

    return mutableRequest;
}

```

可以根据一个Url和参数, 去构造需要的 `request`, 这个类还把自己的一些属性(用户自定义设置)参数, 加到其中去了。

其中主要部分是参数的编码, 主要经过下面三步变成了一个字符串:

```

@{
    @"name" : @"bang",
    @"phone": @{@"mobile": @"xx", @"home": @"xx"},
    @"families": @[@"father", @"mother"],
    @"nums": [NSSet setWithObjects:@"1", @"2", nil]
}
->
@[
    field: @"name", value: @"bang",
    field: @"phone[mobile]", value: @"xx",
    field: @"phone[home]", value: @"xx",
    field: @"families[ ]", value: @"father",
    field: @"families[ ]", value: @"mother",
    field: @"nums", value: @"1",
    field: @"nums", value: @"2",
]
->
name=bang&phone[mobile]=xx&phone[home]=xx&families[ ]=father&families[ ]=mother
&nums=1&num=2

```

然后根据网络请求是 `GET`、`HEAD`、`DELETE`、`PUT`、`POST` 来决定参数字符串是应该放在 `Url` 后面还是 `HTTP` 请求体 `Body` 中。

至此一个完整的 `NSMutableURLRequest` 就拼接完成了。

接着我们继续顺着请求的线索往下看，我们之前 `AFHTTPSessionManager` 类调用了父类 `AFURLSessionManager` 这么一个方法来生成 `task`：

```
dataTask = [self dataTaskWithRequest:request ...]
```

### 3.AF请求核心类：AFURLSessionManager

我们顺着上述线索来看这个方法的实现之前，先看看这个类的初始化方法，它被我们之前说的 `AFHTTPSessionManager` 类所继承。所以 `AFHTTPSessionManager` 的初始化触发了这个类的所有初始化：

```

- (instancetype)initWithSessionConfiguration:(NSURLSessionConfiguration
*)configuration {
    self = [super init];
    if (!self) {
        return nil;
    }

    if (!configuration) {
        configuration = [NSURLSessionConfiguration
defaultSessionConfiguration];
    }
}

```

```

self.sessionConfiguration = configuration;

self.operationQueue = [[NSOperationQueue alloc] init];
//queue并发线程数为1, 这个是代理回调的queue
self.operationQueue.maxConcurrentOperationCount = 1;

//注意代理, 代理的继承, 实际上NSURLSession去判断了, 你实现了哪个方法会去调用, 包括子代理的方法!
self.session = [NSURLSession
sessionWithConfiguration:self.sessionConfiguration delegate:self
delegateQueue:self.operationQueue];

//各种响应转码
self.responseSerializer = [AFJSONResponseSerializer serializer];

//ssl证书, 是验证证书, 还是公钥, 还是不用
self.securityPolicy = [AFSecurityPolicy defaultPolicy];

#if !TARGET_OS_WATCH
self.reachabilityManager = [AFNetworkReachabilityManager sharedManager];
#endif

// 设置存储NSURLSession task与AFURLSessionManagerTaskDelegate的词典(重点, 在AFNet
中, 每一个task都会被匹配一个AFURLSessionManagerTaskDelegate 来做task的delegate事件
处理) =====
self.mutableTaskDelegatesKeyedByTaskIdentifier = [[NSMutableDictionary
alloc] init];
// ===== 设置AFURLSessionManagerTaskDelegate 词典的锁, 确保词典在多
线程访问时的线程安全=====
self.lock = [[NSLock alloc] init];
self.lock.name = AFURLSessionManagerLockName;

// ===== 为所管理的session的所有task设置完成块, 此方法为生成session之后
就调用
[self.session getTasksWithCompletionHandler:^(NSArray *dataTasks, NSArray
*uploadTasks, NSArray *downloadTasks) {
    //开始的时候应该什么都没有
    for (NSURLSessionDataTask *task in dataTasks) {
        [self addDelegateForDataTask:task uploadProgress:nil
downloadProgress:nil completionHandler:nil];
    }

    for (NSURLSessionUploadTask *uploadTask in uploadTasks) {
        [self addDelegateForUploadTask:uploadTask progress:nil
completionHandler:nil];
    }
}];

```

```

    }

    for (NSURLSessionDownloadTask *downloadTask in downloadTasks) {
        [self addDelegateForDownloadTask:downloadTask progress:nil
        destination:nil completionHandler:nil];
    }
}];

return self;
}

```

这个方法初始化了一些我们后续需要用到的属性，其他的都很简单，唯一比较费解的两处可能是：

```
self.operationQueue.maxConcurrentOperationCount = 1;
```

```

[self.session getTasksWithCompletionHandler:^(NSArray *dataTasks, NSArray
*uploadTasks, NSArray *downloadTasks) {
    //置空处理
}];

```

我们首先来讲讲这两个操作的作用：

- 第一是让回调的代理 `queue` 是串行的，即请求完成的 `task` 只能一个个被回调。
- 第二是清空了 `session` 中所有 `task`。

之所以让人费解的是，这么做的意义是什么？第一个目的我们暂且不说，我们放到文章结尾再来谈。而第二个，我们知道这里是初始化方法，讲道理 `session` 中不会有任何 `task`。但是这是因为大家只知其一，我们有一种后台 `session`，当从后台回来的时候，根据一个ID，就可以重新恢复这个 `session`，这时候其中就会有之前未完成的 `task` 了。

而这里这么做的目的就是防止一些之前的后台请求任务，导致程序的 `crash`，见 [github:https://github.com/AFNetworking/AFNetworking/issues/3499](https://github.com/AFNetworking/AFNetworking/issues/3499)

接着我们回到之前的 `dataTaskWithRequest` 返回task的方法：



```

- (NSURLSessionDataTask *)dataTaskWithRequest:(NSURLRequest *)request
                                uploadProgress:(nullable void (^)(NSProgress
*uploadProgress)) uploadProgressBlock
                                downloadProgress:(nullable void (^)(NSProgress
*downloadProgress)) downloadProgressBlock
                                completionHandler:(nullable void (^)(
(NSURLResponse *response, id _Nullable responseObject, NSError * _Nullable
error)))completionHandler {

    __block NSURLSessionDataTask *dataTask = nil;
    //第一件事，创建NSURLSessionDataTask，里面适配了Ios8以下taskIdentifiers，函数创建task对象。
    //其实现应该是因为iOS 8.0以下版本中会并发地创建多个task对象，而同步有没有做好，导致taskIdentifiers 不唯一...这边做了一个串行处理
    url_session_manager_create_task_safely(^{
        dataTask = [self.session dataTaskWithRequest:request];
    });

    [self addDelegateForDataTask:dataTask uploadProgress:uploadProgressBlock
downloadProgress:downloadProgressBlock completionHandler:completionHandler];

    return dataTask;
}

```

当然，这个类有数个类似的方法如下：

```

M -dataTaskWithRequest:completionHandler:
M -dataTaskWithRequest:uploadProgress:downloadProgress:completionHandler:

M -uploadTaskWithRequest:fromFile:progress:completionHandler:
M -uploadTaskWithRequest:fromData:progress:completionHandler:
M -uploadTaskWithStreamedRequest:progress:completionHandler:

M -downloadTaskWithRequest:progress:destination:completionHandler:
M -downloadTaskWithResumeData:progress:destination:completionHandler:

```

这些方法做的事基本一样，就是下面这两件：

(1). 调用 `session` 的方法，传 `request` 过去去生成 `task`。注意这里调用了 `url_session_manager_create_task_safely` 函数去执行的 `Block`，这个函数实现如下：

```

static void url_session_manager_create_task_safely(dispatch_block_t block) {
    if (NSFoundationVersionNumber <
NSFoundationVersionNumber_With_Fixed_5871104061079552_bug) {
        dispatch_sync(url_session_manager_creation_queue(), block);
    } else {
        block();
    }
}

```

简单来讲就是为了适配 iOS8 以下 task 创建，其中 taskIdentifiers 属性不唯一，而这个属性是我们之后添加代理的 key，它必须是唯一的。

所以这里做了一个判断，如果是 iOS8 以下，则用串行同步的方式去执行这个 Block，也就是创建 session。否则直接执行。

(2). 给每个 task 创建并对应一个 AF 的代理对象，这基本上是这个类的核心所在了，这个代理对象为其对应的 task 做数据拼接及成功回调。

我们来看看这个方法：

```
- (void)addDelegateForDataTask:(NSURLSessionDataTask *)dataTask
    uploadProgress:(nullable void (^)(NSProgress
*uploadProgress)) uploadProgressBlock
    downloadProgress:(nullable void (^)(NSProgress
*downloadProgress)) downloadProgressBlock
    completionHandler:(void (^)(NSURLResponse *response, id
responseObject, NSError *error))completionHandler
{
    AFURLSessionManagerTaskDelegate *delegate =
[[AFURLSessionManagerTaskDelegate alloc] init];

    // AFURLSessionManagerTaskDelegate与AFURLSessionManager建立相互关系
    delegate.manager = self;
    delegate.completionHandler = completionHandler;

    //这个taskDescriptionForSessionTasks用来发送开始和挂起通知的时候会用到,就是用这个
    值来Post通知，来两者对应
    dataTask.taskDescription = self.taskDescriptionForSessionTasks;

    // ***** 将AF delegate对象与 dataTask建立关系
    [self setDelegate:delegate forTask:dataTask];

    // 设置AF delegate的上传进度，下载进度块。
    delegate.uploadProgressBlock = uploadProgressBlock;
    delegate.downloadProgressBlock = downloadProgressBlock;
}
```

```

- (void)setDelegate:(AFURLSessionManagerTaskDelegate *)delegate
    forTask:(NSURLSessionTask *)task
{
    //断言，如果没有这个参数，debug下crash在这
    NSParameterAssert(task);
    NSParameterAssert(delegate);

    //加锁保证字典线程安全
    [self.lock lock];

    // 将AF delegate放入以taskIdentifier标记的词典中（同一个NSURLSession中的
    taskIdentifier是唯一的）
    self.mutableTaskDelegatesKeyedByTaskIdentifier[@(task.taskIdentifier)] =
    delegate;

    // 为AF delegate 设置task 的progress监听
    [delegate setupProgressForTask:task];

    //添加task开始和暂停的通知
    [self addNotificationObserverForTask:task];
    [self.lock unlock];
}

```

就这么两个方法创建了一个 `AFURLSessionManagerTaskDelegate` 的代理，把这个代理和 `task` 的 `taskIdentifier` 一一对应，放在我们最早初始化的字典里，建立起映射。除此之外，我们还添加了一些task进度的监听，和task开始和挂起的通知。当然，这些操作都在我们一开始声明的锁中进行，是线程安全的。

做好这些之后，我们所有的任务开启前的操作就完成了，接着我们还记得在 `AFHTTPSessionManager` 中拿到了返回的task，调用了：

```
[dataTask resume];
```

开启了任务，接着task就开始请求网络了，还记得我们初始化方法中：

```

self.session = [NSURLSession
    sessionWithConfiguration:self.sessionConfiguration delegate:self
    delegateQueue:self.operationQueue];

```

我们把 `AFURLSessionManager` 作为了所有的 `task` 的 `delegate`。当我们请求网络的时候，这些代理开始调用了：

	<b>NSURLSessionDelegate</b>
	-URLSession:didBecomeInvalidWithError:
	-URLSession:didReceiveChallenge:completionHandler:
	-URLSessionDidFinishEventsForBackgroundURLSession:
	<b>NSURLSessionTaskDelegate</b>
	-URLSession:task:willPerformHTTPRedirection:newRequest:completionHandler:
	-URLSession:task:didReceiveChallenge:completionHandler:
	-URLSession:task:needNewBodyStream:
	-URLSession:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend:
	-URLSession:task:didCompleteWithError:
	<b>NSURLSessionDataDelegate</b>
	-URLSession:dataTask:didReceiveResponse:completionHandler:
	-URLSession:dataTask:didBecomeDownloadTask:
	-URLSession:dataTask:didReceiveData:
	-URLSession:dataTask:willCacheResponse:completionHandler:
	<b>NSURLSessionDownloadDelegate</b>
	-URLSession:downloadTask:didFinishDownloadingToURL:
	-URLSession:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:
	-URLSession:downloadTask:didResumeAtOffset:expectedTotalBytes:

- `AFURLSessionManager` 一共实现了如上图所示这么一大堆 `NSURLSession` 相关的代理。（小伙伴们的顺序可能不一样，楼主根据代理隶属重新排序了一下）
- 而只转发了其中3条到AF自定义的delegate中：

	<b>NSURLSessionTaskDelegate</b>
	-URLSession:task:didCompleteWithError:
	<b>NSURLSessionDataTaskDelegate</b>
	-URLSession:dataTask:didReceiveData:
	<b>NSURLSessionDownloadTaskDelegate</b>
	-URLSession:downloadTask:didFinishDownloadingToURL:

这就是我们一开始说的，`AFURLSessionManager` 对这一大堆代理做了一些公共的处理，而转发到AF自定义代理的3条，则负责把每个 `task` 对应的数据回调出去。

至于这些代理详细作用和实现内容我就不一一细说，感兴趣的可以到作者之前这篇文章中去查看：<http://www.jianshu.com/p/856f0e26279d>

总结一下，这些代理主要是做了一些额外的处理，并且调用了它的属性 `Block`：

```

@property (readwrite, nonatomic, copy) AFURLSessionDidBecomeInvalidBlock
sessionDidBecomeInvalid;
@property (readwrite, nonatomic, copy)
AFURLSessionDidReceiveAuthenticationChallengeBlock
sessionDidReceiveAuthenticationChallenge;
@property (readwrite, nonatomic, copy)
AFURLSessionDidFinishEventsForBackgroundURLSessionBlock
didFinishEventsForBackgroundURLSession;
@property (readwrite, nonatomic, copy)
AFURLSessionTaskWillPerformHTTPRedirectionBlock
taskWillPerformHTTPRedirection;
@property (readwrite, nonatomic, copy)
AFURLSessionTaskDidReceiveAuthenticationChallengeBlock
taskDidReceiveAuthenticationChallenge;
@property (readwrite, nonatomic, copy) AFURLSessionTaskNeedNewBodyStreamBlock
taskNeedNewBodyStream;
@property (readwrite, nonatomic, copy) AFURLSessionTaskDidSendBodyDataBlock
taskDidSendBodyData;
@property (readwrite, nonatomic, copy) AFURLSessionTaskDidCompleteBlock
taskDidComplete;
@property (readwrite, nonatomic, copy)
AFURLSessionDataTaskDidReceiveResponseBlock dataTaskDidReceiveResponse;
@property (readwrite, nonatomic, copy)
AFURLSessionDataTaskDidBecomeDownloadTaskBlock dataTaskDidBecomeDownloadTask;
@property (readwrite, nonatomic, copy)
AFURLSessionDataTaskDidReceiveDataBlock dataTaskDidReceiveData;
@property (readwrite, nonatomic, copy)
AFURLSessionDataTaskWillCacheResponseBlock dataTaskWillCacheResponse;
@property (readwrite, nonatomic, copy)
AFURLSessionDownloadTaskDidFinishDownloadingBlock
downloadTaskDidFinishDownloading;
@property (readwrite, nonatomic, copy)
AFURLSessionDownloadTaskDidWriteDataBlock downloadTaskDidWriteData;
@property (readwrite, nonatomic, copy) AFURLSessionDownloadTaskDidResumeBlock
downloadTaskDidResume;

```

我们可以利用这些 `Block`，做一些自定义的处理，`Block` 会随着代理调用而被调用，这些代理帮我们做了一些类似数据分片、断电续传、`https` 认证等工作。

除此之外，有3个代理方法回调了我们的 `task` 的AF代理，包括请求完成的代理，收到数据的代理，以及下载完成的代理，以第一个为例：

```

- (void)URLSession:(NSURLSession *)session
    task:(NSURLSessionTask *)task
didCompleteWithError:(NSError *)error
{
    //根据task去取我们一开始创建绑定的delegate
    AFURLSessionManagerTaskDelegate *delegate = [self delegateForTask:task];

    // delegate may be nil when completing a task in the background
    if (delegate) {
        //把代理转发给我们绑定的delegate
        [delegate URLSession:session task:task didCompleteWithError:error];
        //转发完移除delegate
        [self removeDelegateForTask:task];
    }

    //公用Block回调
    if (self.taskDidComplete) {
        self.taskDidComplete(session, task, error);
    }
}

```

通过我们之前设置的 `task` 和 `AF` 代理映射，去调用 `AF` 代理，并且把这个 `task` 从映射字典中移除。接着就调用了 `AF` 的代理：

```

//AF实现的代理！被从urlsession那转发到这

- (void)URLSession:(__unused NSURLSession *)session
    task:(NSURLSessionTask *)task
didCompleteWithError:(NSError *)error
{
    #pragma clang diagnostic push
    #pragma clang diagnostic ignored "-Wgnu"

    //1) 强引用self.manager，防止被提前释放；因为self.manager声明为weak,类似Block
    __strong AFURLSessionManager *manager = self.manager;

    __block id responseObject = nil;

    //用来存储一些相关信息，来发送通知用的
    __block NSMutableDictionary *userInfo = [NSMutableDictionary dictionary];
    //存储responseSerializer响应解析对象
    userInfo[AFNetworkingTaskDidCompleteResponseSerializerKey] =
manager.responseSerializer;

    //Performance Improvement from #2672

```

//注意这行代码的用法，感觉写的很Nice...把请求到的数据data传出去，然后就不要这个值了释放内存

```
NSData *data = nil;
if (self.mutableData) {
    data = [self.mutableData copy];
    //We no longer need the reference, so nil it out to gain back some
memory.
    self.mutableData = nil;
}

//继续给userinfo填数据
if (self.downloadFileURL) {
    userInfo[AFNetworkingTaskDidCompleteAssetPathKey] =
self.downloadFileURL;
} else if (data) {
    userInfo[AFNetworkingTaskDidCompleteResponseDataKey] = data;
}
//错误处理
if (error) {

    userInfo[AFNetworkingTaskDidCompleteErrorKey] = error;

    //可以自己自定义完成组 和自定义完成queue,完成回调
    dispatch_group_async(manager.completionGroup ?:
url_session_manager_completion_group(), manager.completionQueue ?:
dispatch_get_main_queue(), ^{
        if (self.completionHandler) {
            self.completionHandler(task.response, responseObject, error);
        }
        //主线程中发送完成通知
        dispatch_async(dispatch_get_main_queue(), ^{
            [[NSNotificationCenter defaultCenter]
postNotificationName:AFNetworkingTaskDidCompleteNotification object:task
userInfo:userInfo];
        });
    });
} else {
    //url_session_manager_processing_queue AF的并行队列
    dispatch_async(url_session_manager_processing_queue(), ^{
        NSError *serializationError = nil;

        //解析数据
        responseObject = [manager.responseSerializer
responseObjectForResponse:task.response data:data error:&serializationError];

        //如果是下载文件，那么responseObject为下载的路径
        if (self.downloadFileURL) {
            responseObject = self.downloadFileURL;
        }
    });
}
```

```

        //写入userInfo
        if (responseObject) {
            userInfo[AFNetworkingTaskDidCompleteSerializedResponseKey] =
responseObject;
        }

        //如果解析错误
        if (serializationError) {
            userInfo[AFNetworkingTaskDidCompleteErrorKey] =
serializationError;
        }
        //回调结果
        dispatch_group_async(manager.completionGroup ?:
url_session_manager_completion_group(), manager.completionQueue ?:
dispatch_get_main_queue(), ^{
            if (self.completionHandler) {
                self.completionHandler(task.response, responseObject,
serializationError);
            }

            dispatch_async(dispatch_get_main_queue(), ^{
                [[NSNotificationCenter defaultCenter]
postNotificationName:AFNetworkingTaskDidCompleteNotification object:task
userInfo:userInfo];
            });
        });
    }
#pragma clang diagnostic pop
}

```

虽然这个方法有点长，但是它主要做了两件事：

1. 调用 `responseSerializer` 按照我们设置的格式，解析请求到的数据。
2. 用 `completionHandler` 把数据回调出去，至此数据回到了用户手中。

到这里，`AF` 的整个主线流程就完了，当然，我们跳过了很多细节没有讲，比如 `responseSerializer` 的各种格式的解析过程，还有为了监听 `task` 的开始和挂起通知，所做的 `method swizzling`，这里对 `ios7` 的兼容问题的处理，算是相当精彩了。

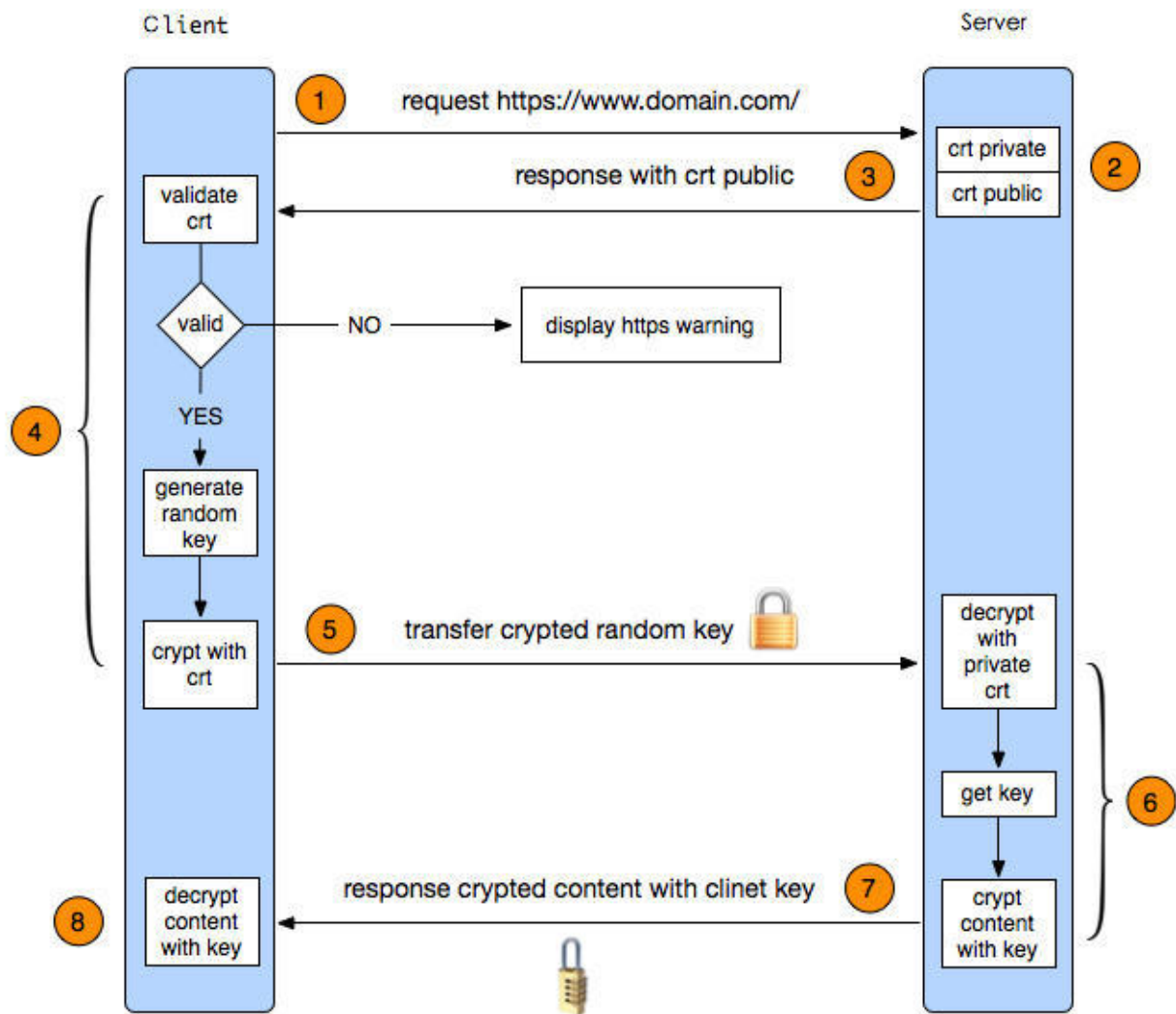
## 二. AF3.X的安全策略。

AF的安全策略主要由 `https` 来保证，那么什么是 `https` 呢？

这里我们简单的理解下`https`：**`https`在`http`请求的基础上多加了一个证书认证的流程**。认证通过之后，数据传输都是加密进行的。

关于`https`的更多概念，我就不赘述了，网上有大量的文章，小伙伴们可以自行查阅。在这里大概的讲讲`https`的认证过程吧，如下图所示：





整个https验证的流程了。简单总结一下：

- 就是用户发起请求，服务器响应后返回一个证书，证书中包含一些基本信息和公钥。
- 用户拿到证书后，去验证这个证书是否合法，不合法，则请求终止。
- 合法则生成一个随机数，作为对称加密的密钥，用服务器返回的公钥对这个随机数加密。然后返回给服务器。
- 服务器拿到加密后的随机数，利用私钥解密，然后再用解密后的随机数（对称密钥），把需要返回的数据加密，加密完成后数据传输给用户。
- 最后用户拿到加密的数据，用一开始的那个随机数（对称密钥），进行数据解密。整个过程完成。

当然这仅仅是一个单向认证，https还会有双向认证，相对于单向认证也很简单。仅仅多了服务端验证客户端这一步，步骤也是一模一样。

我们之前将代理的时候，有关于 `https` 认证的代理，`AF` 做了如下处理：

```

- (void)URLSession:(NSURLSession *)session
didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge
completionHandler:(void (^)(NSURLSessionAuthChallengeDisposition
disposition, NSURLCredential *credential))completionHandler
{
    //挑战处理类型为 默认
    /*
  
```

```

NSURLSessionAuthChallengePerformDefaultHandling: 默认方式处理
NSURLSessionAuthChallengeUseCredential: 使用指定的证书
NSURLSessionAuthChallengeCancelAuthenticationChallenge: 取消挑战
*/
NSURLSessionAuthChallengeDisposition disposition =
NSURLSessionAuthChallengePerformDefaultHandling;
__block NSURLCredential *credential = nil;

// sessionDidReceiveAuthenticationChallenge是自定义方法，用来如何应对服务器端的
认证挑战

if (self.sessionDidReceiveAuthenticationChallenge) {
    disposition = self.sessionDidReceiveAuthenticationChallenge(session,
challenge, &credential);
} else {
    // 此处服务器要求客户端的接收认证挑战方法是
NSURLSessionAuthenticationMethodServerTrust
    // 也就是说服务器端需要客户端返回一个根据认证挑战的保护空间提供的信任（即
challenge.protectionSpace.serverTrust）产生的挑战证书。

    // 而这个证书就需要使用credentialForTrust:来创建一个NSURLCredential对象
    if ([challenge.protectionSpace.authenticationMethod
isEqualToString:NSURLAuthenticationMethodServerTrust]) {

        // 基于客户端的安全策略来决定是否信任该服务器，不信任的话，也就没必要响应挑战
        if ([self.securityPolicy
evaluateServerTrust:challenge.protectionSpace.serverTrust
forDomain:challenge.protectionSpace.host]) {
            // 创建挑战证书（注：挑战方式为UseCredential和
PerformDefaultHandling都需要新建挑战证书）
            credential = [NSURLCredential
credentialForTrust:challenge.protectionSpace.serverTrust];
            // 确定挑战的方式
            if (credential) {
                //证书挑战 设计policy,none, 则跑到这里
                disposition = NSURLSessionAuthChallengeUseCredential;
            } else {
                disposition =
NSURLSessionAuthChallengePerformDefaultHandling;
            }
        } else {
            //取消挑战
            disposition =
NSURLSessionAuthChallengeCancelAuthenticationChallenge;
        }
    } else {
        //默认挑战方式
        disposition = NSURLSessionAuthChallengePerformDefaultHandling;
    }
}

```

```

    }
    //完成挑战
    if (completionHandler) {
        completionHandler(disposition, credential);
    }
}

```

在这里我们大概的讲讲这个方法做了什么：

- 首先指定了https为默认认证方式。
- 判断有没有自定义Block: `sessionDidReceiveAuthenticationChallenge`，有的话，使用我们自定义Block,生成一个认证方式，并且可以给 `credential` 赋值，即我们需要接受认证的证书。然后直接调用 `completionHandler`，去根据这两个参数，执行系统的认证。至于这个系统的认证到底做了什么，可以看文章最后，这里暂且略过。
- 如果没有自定义Block，我们判断如果服务端的认证方法要求是 `NSURLAuthenticationMethodServerTrust`，则只需要验证服务端证书是否安全（即https的单向认证，这是AF默认处理的认证方式，其他的认证方式，只能由我们自定义Block的实现）
- 接着我们就执行了 `AFSecurityPolicy` 相关的一个方法，做了一个AF内部的一个https认证：

```

[self.securityPolicy
evaluateServerTrust:challenge.protectionSpace.serverTrust
forDomain:challenge.protectionSpace.host])

```

AF默认的处理是，如果这行返回NO、说明AF内部认证失败，则取消https认证，即取消请求。返回YES则进入if块，用服务器返回的一个 `serverTrust` 去生成了一个认证证书。（注：这个 `serverTrust` 是服务器传过来的，里面包含了服务器的证书信息，是用来我们本地客户端去验证该证书是否合法用的，后面会更详细的去讲这个参数）然后如果有证书，则用证书认证方式，否则还是用默认的验证方式。最后调用 `completionHandler` 传递认证方式和要认证的证书，去做系统根证书验证。

总结一下：这里 `securityPolicy` 存在的作用就是，使得在系统底层自己去验证之前，AF可以先去验证服务端的证书。如果通不过，则直接越过系统的验证，取消https的网络请求。否则，继续去走系统根证书的验证。

接下来我们看看 `AFSecurityPolicy` 内部是如何做https认证的：

如下方式，我们可以创建一个 `securityPolicy`：

```

AFSecurityPolicy *policy = [AFSecurityPolicy defaultPolicy];

```

内部创建：

```

+ (instancetype)defaultPolicy {
    AFSecurityPolicy *securityPolicy = [[self alloc] init];
    securityPolicy.SSLPinningMode = AFSSLPinningModeNone;
    return securityPolicy;
}

```

默认指定了一个 `SSLPinningMode` 模式为 `AFSSLPinningModeNone`。

对于 `AFSecurityPolicy`，一共有4个重要的属性：

```
//https验证模式
@property (readonly, nonatomic, assign) AFSSLPinningMode SSLPinningMode;
//可以去匹配服务端证书验证的证书
@property (nonatomic, strong, nullable) NSSet <NSData *> *pinnedCertificates;
//是否支持非法的证书（例如自签名证书）
@property (nonatomic, assign) BOOL allowInvalidCertificates;
//是否去验证证书域名是否匹配
@property (nonatomic, assign) BOOL validatesDomainName;
```

它们的作用我添加在注释里了，第一条就是 `AFSSLPinningMode`，共提供了3种验证方式：

```
typedef NS_ENUM(NSUInteger, AFSSLPinningMode) {
    //不验证
    AFSSLPinningModeNone,
    //只验证公钥
    AFSSLPinningModePublicKey,
    //验证证书
    AFSSLPinningModeCertificate,
};
```

我们接着回到代理https认证的这行代码上：

```
[self.securityPolicy
evaluateServerTrust:challenge.protectionSpace.serverTrust
forDomain:challenge.protectionSpace.host]
```

- 我们传了两个参数进去，一个是 `SecTrustRef` 类型的 `serverTrust`，这是什么呢？我们看到苹果文档介绍如下：

CType used for performing X.509 certificate trust evaluations.

大概意思是用于执行X.509证书信任评估，再讲简单点，其实就是一个容器，装了服务器端需要验证的证书的基本信息、公钥等等，不仅如此，它还可以装一些评估策略，还有客户端的锚点证书，这个客户端的证书，可以用来和服务端的证书去匹配验证的。

- 除此之外还把服务器域名传了过去。

我们来到这个方法，代码如下：

```
//验证服务端是否值得信任
- (BOOL)evaluateServerTrust:(SecTrustRef)serverTrust
forDomain:(NSString *)domain
{
    //判断矛盾的条件
    //判断有域名，且允许自建证书，需要验证域名，
    //因为要验证域名，所以必须不能是后者两种：AFSSLPinningModeNone或者添加到项目里的证
```

书为0个。

```
    if (domain && self.allowInvalidCertificates && self.validatesDomainName
    && (self.SSLPinningMode == AFSSLPinningModeNone || [self.pinnedCertificates
    count] == 0)) {
        return NO;
    }

    //用来装验证策略
    NSMutableArray *policies = [NSMutableArray array];
    //要验证域名
    if (self.validatesDomainName) {

        // 如果需要验证domain, 那么就使用SecPolicyCreateSSL函数创建验证策略, 其中第一个
        参数为true表示验证整个SSL证书链, 第二个参数传入domain, 用于判断整个证书链上叶子节点表示
        的那个domain是否和此处传入domain一致
        //添加验证策略
        [policies addObject:(__bridge_transfer id)SecPolicyCreateSSL(true,
        (__bridge CFStringRef)domain)];
    } else {
        // 如果不需要验证domain, 就使用默认的BasicX509验证策略
        [policies addObject:(__bridge_transfer
        id)SecPolicyCreateBasicX509()];
    }

    //serverTrust: X. 509服务器的证书信任。
    // 为serverTrust设置验证策略, 即告诉客户端如何验证serverTrust
    SecTrustSetPolicies(serverTrust, (__bridge CFArrayRef)policies);

    //有验证策略了, 可以去验证了。如果是AFSSLPinningModeNone, 是自签名, 直接返回可信
    任, 否则不是自签名的就去系统根证书里去找是否有匹配的证书。
    if (self.SSLPinningMode == AFSSLPinningModeNone) {
        //如果支持自签名, 直接返回YES, 不允许才去判断第二个条件, 判断serverTrust是否有效
        return self.allowInvalidCertificates ||
        AFServerTrustIsValid(serverTrust);
    }
    //如果验证无效AFServerTrustIsValid, 而且allowInvalidCertificates不允许自签, 返
    回NO
    else if (!AFServerTrustIsValid(serverTrust) &&
    !self.allowInvalidCertificates) {
        return NO;
    }

    //判断SSLPinningMode
    switch (self.SSLPinningMode) {
        // 理论上, 上面那个部分已经解决了self.SSLPinningMode)为
        AFSSLPinningModeNone)等情况, 所以此处再遇到, 就直接返回NO
        case AFSSLPinningModeNone:
        default:
```

```

        return NO;

        //验证证书类型
        case AFSSLPinningModeCertificate: {

            NSMutableArray *pinnedCertificates = [NSMutableArray array];

            //把证书data, 用系统api转成 SecCertificateRef 类型的数
            据, SecCertificateCreateWithData函数对原先的pinnedCertificates做一些处理, 保证返回的
            证书都是DER编码的X.509证书

            for (NSData *certificateData in self.pinnedCertificates) {
                [pinnedCertificates addObject:(__bridge_transfer
                id)SecCertificateCreateWithData(NULL, (__bridge CFDataRef)certificateData)];
            }
            // 将pinnedCertificates设置成需要参与验证的Anchor Certificate (锚点证
            书, 通过SecTrustSetAnchorCertificates设置了参与校验锚点证书之后, 假如验证的数字证书是这个
            锚点证书的子节点, 即验证的数字证书是由锚点证书对应CA或子CA签发的, 或是该证书本身, 则信任该
            证书), 具体就是调用SecTrustEvaluate来验证。
            //serverTrust是服务器来的验证, 有需要被验证的证书。
            SecTrustSetAnchorCertificates(serverTrust, (__bridge
            CFArrayRef)pinnedCertificates);

            //自签在之前是验证通过不了的, 在这一步, 把我们自己设置的证书加进去之后, 就能验
            证成功了。

            //再去调用之前的serverTrust去验证该证书是否有效, 有可能: 经过这个方法过滤
            后, serverTrust里面的pinnedCertificates被筛选到只有信任的那一个证书
            if (!AFServerTrustIsValid(serverTrust)) {
                return NO;
            }

            // obtain the chain after being validated, which *should* contain
            the pinned certificate in the last position (if it's the Root CA)
            //注意, 这个方法和我们之前的锚点证书没关系了, 是去从我们需要被验证的服务端证
            书, 去拿证书链。
            // 服务器端的证书链, 注意此处返回的证书链顺序是从叶节点到根节点
            NSArray *serverCertificates =
            AFCertificateTrustChainForServerTrust(serverTrust);

            //reverseObjectEnumerator逆序
            for (NSData *trustChainCertificate in [serverCertificates
            reverseObjectEnumerator]) {

                //如果我们的证书中, 有一个和它证书链中的证书匹配的, 就返回YES
                if ([self.pinnedCertificates
                containsObject:trustChainCertificate]) {
                    return YES;
                }
            }
        }
    }
}

```

```

    }
    //没有匹配的
    return NO;
}

//公钥验证 AFSSLPinningModePublicKey模式同样是用证书绑定(SSL Pinning)
方式验证，客户端要有服务端的证书拷贝，只是验证时只验证证书里的公钥，不验证证书的有效期等信息。只要公钥是正确的，就能保证通信不会被窃听，因为中间人没有私钥，无法解开通过公钥加密的数据。

case AFSSLPinningModePublicKey: {

    NSUInteger trustedPublicKeyCount = 0;

    // 从serverTrust中取出服务器端传过来的所有可用的证书，并依次得到相应的公钥
    NSArray *publicKeys =
    AFPublicKeyTrustChainForServerTrust(serverTrust);

    //遍历服务端公钥
    for (id trustChainPublicKey in publicKeys) {
        //遍历本地公钥
        for (id pinnedPublicKey in self.pinnedPublicKeys) {
            //判断如果相同 trustedPublicKeyCount+1
            if (AFSecKeyIsEqualToKey((__bridge
    SecKeyRef)trustChainPublicKey, (__bridge SecKeyRef)pinnedPublicKey)) {
                trustedPublicKeyCount += 1;
            }
        }
    }
    return trustedPublicKeyCount > 0;
}

return NO;
}

```

代码的注释很多，这一块确实比枯燥，大家可以参照着源码一起看，加深理解。

这个方法是 `AFSecurityPolicy` 最核心的方法，其他的都是为了配合这个方法。这个方法完成了服务端的证书的信任评估。我们总结一下这个方法做了什么（细节可以看注释）：

(1) . 根据模式，如果是 `AFSSLPinningModeNone`，则肯定是返回YES，不论是自签还是公信机构的证书。

(2) . 如果是 `AFSSLPinningModeCertificate`，则从 `serverTrust` 中去获取证书链，然后和我们一开始初始化设置的证书集合 `self.pinnedCertificates` 去匹配，如果有一对能匹配成功的，就返回YES，否则NO。

看到这可能有小伙伴要问了，什么是证书链？下面这段是我从百科上摘来的：

证书链由两个环节组成—信任锚（CA 证书）环节和已签名证书环节。自我签名的证书仅有一个环节的长度—信任锚环节就是已签名证书本身。

简单来说，证书链就是就是根证书，和根据根证书签名派发得到的证书。

(3) . 如果是 `AFSSLPinningModePublicKey` 公钥验证，则和第二步一样还是从 `serverTrust`，获取证书链每一个证书的公钥，放到数组中。和我们的 `self.pinnedPublicKeys`，去配对，如果一个相同的，就返回YES，否则NO。至于这个 `self.pinnedPublicKeys`，初始化的地方如下：

```
//设置证书数组
- (void)setPinnedCertificates:(NSSet *)pinnedCertificates {

    _pinnedCertificates = pinnedCertificates;

    //获取对应公钥集合
    if (self.pinnedCertificates) {
        //创建公钥集合
        NSMutableSet *mutablePinnedPublicKeys = [NSMutableSet
setWithCapacity:[self.pinnedCertificates count]];
        //从证书中拿到公钥。
        for (NSData *certificate in self.pinnedCertificates) {
            id publicKey = AFPublicKeyForCertificate(certificate);
            if (!publicKey) {
                continue;
            }
            [mutablePinnedPublicKeys addObject:publicKey];
        }
        self.pinnedPublicKeys = [NSSet setWithSet:mutablePinnedPublicKeys];
    } else {
        self.pinnedPublicKeys = nil;
    }
}
```

AF复写了设置证书的set方法，并同时把证书中每个公钥放在了self.pinnedPublicKeys中。

这个方法中关联了一系列的函数，我在这边按照调用顺序一一列出来（有些是系统函数，不在这里列出，会在下文集体描述作用）：

## 函数一：AFServerTrustIsValid



```

//判断serverTrust是否有效
static BOOL AFServerTrustIsValid(SecTrustRef serverTrust) {

    //默认无效
    BOOL isValid = NO;
    //用来装验证结果，枚举
    SecTrustResultType result;

    //__Require_noErr_Quiet 用来判断前者是0还是非0，如果0则表示没错，就跳到后面的表达式所在位置去执行，否则表示有错就继续往下执行。

    //SecTrustEvaluate系统评估证书的是否可信的函数，去系统根目录找，然后把结果赋值给result。评估结果匹配，返回0，否则出错返回非0
    //do while 0 ,只执行一次，为啥要这样写....
    __Require_noErr_Quiet(SecTrustEvaluate(serverTrust, &result), _out);

    //评估没出错走掉这，只有两种结果能设置为有效，isValid= 1
    //当result为kSecTrustResultUnspecified（此标志表示serverTrust评估成功，此证书也被暗中信任了，但是用户并没有显示地决定信任该证书）。
    //或者当result为kSecTrustResultProceed（此标志表示评估成功，和上面不同的是该评估得到了用户认可），这两者取其一就可以认为对serverTrust评估成功
    isValid = (result == kSecTrustResultUnspecified || result ==
kSecTrustResultProceed);

    //out函数块,如果为SecTrustEvaluate，返回非0，则评估出错，则isValid为NO
_out:
    return isValid;
}

```

- 这个方法用来验证serverTrust是否有效，其中主要是交由系统API `SecTrustEvaluate` 来验证的，它验证完之后会返回一个 `SecTrustResultType` 枚举类型的result，然后我们根据这个result去判断是否证书是否有效。
- 其中比较有意思的是，它调用了一个系统定义的宏函数 `__Require_noErr_Quiet`，函数定义如下：

```

#ifndef __Require_noErr_Quiet
#define __Require_noErr_Quiet(errorCode, exceptionLabel)
\
    do
\
    {
\
        if ( __builtin_expect(0 != (errorCode), 0) )
\
        {
\
            goto exceptionLabel;
\
        }
\
    } while ( 0 )
#endif

```

这个函数主要作用就是，判断errorCode是否为0，不为0则，程序用 `goto` 跳到 `exceptionLabel` 位置去执行。这个 `exceptionLabel` 就是一个代码位置标识，类似上面的 `_out`。

说它有意思的地方是在于，它用了 `do...while(0)` 循环，循环条件为0，也就是只执行一次循环就结束。对这么做的原因，楼主百思不得其解...看来系统原生API更是高深莫测...经冰霜大神的提醒，这么做是为了适配早期的API？！

## 函数二、三（两个函数类似，所以放在一起）：获取serverTrust证书链证书，获取serverTrust证书链公钥

```

//获取证书链
static NSArray * AFCertificateTrustChainForServerTrust(SecTrustRef
serverTrust) {
    //使用SecTrustGetCertificateCount函数获取到serverTrust中需要评估的证书链中的证书
    数目, 并保存到certificateCount中
    CFIndex certificateCount = SecTrustGetCertificateCount(serverTrust);
    //创建数组
    NSMutableArray *trustChain = [NSMutableArray arrayWithCapacity:
(NSUInteger)certificateCount];

    //// 使用SecTrustGetCertificateAtIndex函数获取到证书链中的每个证书, 并添加到
    trustChain中, 最后返回trustChain
    for (CFIndex i = 0; i < certificateCount; i++) {
        SecCertificateRef certificate =
SecTrustGetCertificateAtIndex(serverTrust, i);
        [trustChain addObject:(__bridge_transfer NSData
*)SecCertificateCopyData(certificate)];
    }

    return [NSArray arrayWithArray:trustChain];
}

```

```

// 从serverTrust中取出服务器端传过来的所有可用的证书, 并依次得到相应的公钥
static NSArray * AFPublicKeyTrustChainForServerTrust(SecTrustRef serverTrust)
{

    // 接下来的一小段代码和上面AFCertificateTrustChainForServerTrust函数的作用基本一
    致, 都是为了获取到serverTrust中证书链上的所有证书, 并依次遍历, 取出公钥。
    //安全策略
    SecPolicyRef policy = SecPolicyCreateBasicX509();
    CFIndex certificateCount = SecTrustGetCertificateCount(serverTrust);
    NSMutableArray *trustChain = [NSMutableArray arrayWithCapacity:
(NSUInteger)certificateCount];
    //遍历serverTrust里证书的证书链。
    for (CFIndex i = 0; i < certificateCount; i++) {
        //从证书链取证书
        SecCertificateRef certificate =
SecTrustGetCertificateAtIndex(serverTrust, i);
        //数组
        SecCertificateRef someCertificates[] = {certificate};
        //CF数组
        CFArrayRef certificates = CFArrayCreate(NULL, (const void
**)someCertificates, 1, NULL);

        SecTrustRef trust;

        // 根据给定的certificates和policy来生成一个trust对象
        //不成功跳到 _out。

```

```

    __Require_noErr_Quiet(SecTrustCreateWithCertificates(certificates,
policy, &trust), _out);

    SecTrustResultType result;

    // 使用SecTrustEvaluate来评估上面构建的trust
    //评估失败跳到 _out
    __Require_noErr_Quiet(SecTrustEvaluate(trust, &result), _out);

    // 如果该trust符合X.509证书格式, 那么先使用SecTrustCopyPublicKey获取到trust
    的公钥, 再将此公钥添加到trustChain中
    [trustChain addObject:(__bridge_transfer
id)SecTrustCopyPublicKey(trust)];

    _out:
    //释放资源
    if (trust) {
        CFRelease(trust);
    }

    if (certificates) {
        CFRelease(certificates);
    }

    continue;
}
CFRelease(policy);

// 返回对应的一组公钥
return [NSArray arrayWithArray:trustChain];
}

```

两个方法功能类似, 都是调用了一些系统的API, 利用For循环, 获取证书链上每一个证书或者公钥。具体内容看源码很好理解。唯一需要注意的是, 这个获取的证书排序, 是从证书链的叶节点, 到根节点的。

## 函数四：判断公钥是否相同

```

//判断两个公钥是否相同
static BOOL AFSecKeyIsEqualToKey(SecKeyRef key1, SecKeyRef key2) {

#if TARGET_OS_IOS || TARGET_OS_WATCH || TARGET_OS_TV
    //iOS 判断二者地址
    return [(__bridge id)key1 isEqual:(__bridge id)key2];
#else
    return [AFSecKeyGetData(key1) isEqual:AFSecKeyGetData(key2)];
#endif
}

```

方法适配了各种运行环境，做了匹配的判断。

接下来列出验证过程中调用过得系统原生函数：

```
//1.创建一个验证SSL的策略，两个参数，第一个参数true则表示验证整个证书链
//第二个参数传入domain，用于判断整个证书链上叶子节点表示的那个domain是否和此处传入domain一致
SecPolicyCreateSSL(<#Boolean server#>, <#CFStringRef _Nullable hostname#>)
SecPolicyCreateBasicX509();
//2.默认的BasicX509验证策略，不验证域名。
SecPolicyCreateBasicX509();
//3.为serverTrust设置验证策略，即告诉客户端如何验证serverTrust
SecTrustSetPolicies(<#SecTrustRef _Nonnull trust#>, <#CFTypesRef _Nonnull policies#>)
//4.验证serverTrust,并且把验证结果返回给第二参数 result
SecTrustEvaluate(<#SecTrustRef _Nonnull trust#>, <#SecTrustResultType * _Nullable result#>)
//5.判断前者errorCode是否为0，为0则跳到exceptionLabel处执行代码
__Require_noErr(<#errorCode#>, <#exceptionLabel#>)
//6.根据证书data,去创建SecCertificateRef类型的数据。
SecCertificateCreateWithData(<#CFAllocatorRef _Nullable allocator#>, <#CFDataRef _Nonnull data#>)
//7.给serverTrust设置锚点证书，即如果以后再次去验证serverTrust，会从锚点证书去找是否匹配。
SecTrustSetAnchorCertificates(serverTrust, (__bridge CFArrayRef) pinnedCertificates);
//8.拿到证书链中的证书个数
CFIndex certificateCount = SecTrustGetCertificateCount(serverTrust);
//9.去取得证书链中对应下标的证书。
SecTrustGetCertificateAtIndex(serverTrust, i)
//10.根据证书获取公钥。
SecTrustCopyPublicKey(trust)
```

其功能如注释，大家可以对比着源码，去加以理解~

可能看到这，又有些小伙伴迷糊了，讲了这么多，那如果做https请求，真正需要我们自己做的到底是什么呢？这里来解答一下，分为以下两种情况：

1. 如果你用的是付费的公信机构颁发的证书，标准的https，那么无论你用的是AF还是NSUrlSession,什么都不需要做，代理方法也不用实现。你的网络请求就能正常完成。
2. 如果你用的是自签名的证书：
  - 首先你需要在plist文件中，设置可以返回不安全的请求（关闭该域名的ATS）。
  - 其次，如果是NSUrlSession，那么需要在代理方法实现如下：

```

- (void)URLSession:(NSURLSession *)session
didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge
completionHandler:(void (^)(NSURLSessionAuthChallengeDisposition
disposition, NSURLCredential *credential))completionHandler
{
    __block NSURLCredential *credential = nil;

    credential = [NSURLCredential
credentialForTrust:challenge.protectionSpace.serverTrust];
    // 确定挑战的方式
    if (credential) {
        //证书挑战 则跑到这里
        disposition = NSURLSessionAuthChallengeUseCredential;
    }
    //完成挑战
    if (completionHandler) {
        completionHandler(disposition, credential);
    }
}

```

其实上述就是AF的相对于自签证书的实现的简化版。

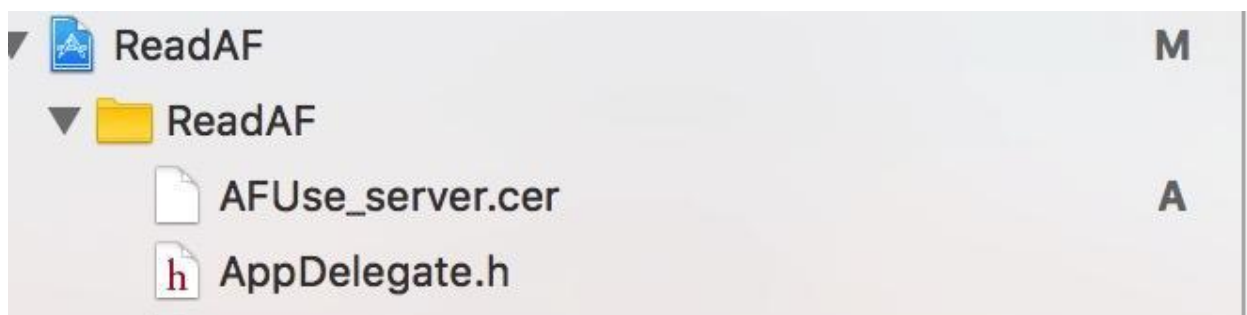
如果是AF，你则需要设置policy：

```

//允许自签名证书，必须的
policy.allowInvalidCertificates = YES;
//是否验证域名的CN字段
//不是必须的，但是如果写YES，则必须导入证书。
policy.validatesDomainName = NO;

```

当然还可以根据需求，你可以去验证证书或者公钥，前提是，你把自签的服务端证书，或者自签的CA根证书导入到项目中：



并且如下设置证书：

```

NSString *certFilePath = [[NSBundle mainBundle]
pathForResource:@"AFUse_server.cer" ofType:nil];
NSData *certData = [NSData dataWithContentsOfFile:certFilePath];
NSSet *certSet = [NSSet setWithObjects:certData, certData, nil];
policy.pinnedCertificates = certSet;

```

这样你就可以使用AF的不同 `AFSSLPinningMode` 去验证了。

最后总结一下，AF之于https到底做了什么：

- **AF可以让你在系统验证证书之前，就去自主验证。**然后如果自己验证不正确，直接取消网络请求。否则验证通过则继续进行系统验证。
- 讲到这，顺便提一下，系统验证的流程：
  - 系统的验证，首先是去系统的根证书找，看是否有能匹配服务端的证书，如果匹配，则验证成功，返回https的安全数据。
- 如果不匹配则去判断ATS是否关闭，如果关闭，则返回https不安全连接的数据。如果开启ATS，则拒绝这个请求，请求失败。

总之一句话：**AF的验证方式不是必须的，但是对有特殊验证需求的用户确是必要的。**

### 三. AF3.X的UIKit扩展实现。

这个类的作用相当简单，就是当网络请求的时候，状态栏上的小菊花就会开始转：



需要的代码也很简单，只需在你需要它的位置中（比如AppDelegate）导入类，并加一行代码即可：

```
#import "AFNetworkActivityIndicatorManager.h"
```

```
[[AFNetworkActivityIndicatorManager sharedManager] setEnabled:YES];
```

接下来我们来讲讲这个类的实现：

1. 这个类的实现也非常简单，还记得我们之前讲的AF对 `NSURLSessionTask` 中做了一个 **Method Swizzling** 吗？大意是把它的 `resume` 和 `suspend` 方法做了一个替换，在原有实现的基础上添加了一个通知的发送。
2. 这个类就是基于这两个通知和task完成的通知来实现的。

首先我们来看看它的初始化方法：

```

+ (instancetype)sharedManager {
    static AFNetworkActivityIndicatorManager *_sharedManager = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _sharedManager = [[self alloc] init];
    });

    return _sharedManager;
}

- (instancetype)init {
    self = [super init];
    if (!self) {
        return nil;
    }
    //设置状态为没有request活跃
    self.currentState = AFNetworkActivityManagerStateNotActive;
    //开始下载通知
    [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(networkRequestDidStart:)
    name:AFNetworkingTaskDidResumeNotification object:nil];
    //挂起通知
    [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(networkRequestDidFinish:)
    name:AFNetworkingTaskDidSuspendNotification object:nil];
    //完成通知
    [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(networkRequestDidFinish:)
    name:AFNetworkingTaskDidCompleteNotification object:nil];
    //开始延迟
    self.activationDelay = kDefaultAFNetworkActivityManagerActivationDelay;
    //结束延迟
    self.completionDelay = kDefaultAFNetworkActivityManagerCompletionDelay;
    return self;
}

```

初始化如上，设置了一个state，这个state是一个枚举：

```

typedef NS_ENUM(NSInteger, AFNetworkActivityManagerState) {
    //没有请求
    AFNetworkActivityManagerStateNotActive,
    //请求延迟开始
    AFNetworkActivityManagerStateDelayingStart,
    //请求进行中
    AFNetworkActivityManagerStateActive,
    //请求延迟结束
    AFNetworkActivityManagerStateDelayingEnd
};

```



这个state一共如上4种状态，其中两种应该很好理解，而延迟开始和延迟结束怎么理解呢？

- 原来这是AF对请求菊花显示做的一个优化处理，试问如果一个请求时间很短，那么菊花很可能闪一下就结束了。如果很多请求过来，那么菊花会不停的闪啊闪，这显然并不是我们想要的效果。
- 所以多了这两个参数：

1) 在一个请求开始的时候，我延迟一会在去转菊花，如果在这延迟时间内，请求结束了，那么我就不需要去转菊花了。

2) 但是一旦转菊花开始，哪怕很短请求就结束了，我们还是会去转一个时间再去结束，这时间就是延迟结束的时间。

- 紧接着我们监听了三个通知，用来监听当前正在进行的网络请求的状态。
- 然后设置了我们前面提到的这个转菊花延迟开始和延迟结束的时间，这两个默认值如下：

```
static NSTimeInterval const kDefaultAFNetworkActivityIndicatorActivationDelay =
1.0;
static NSTimeInterval const kDefaultAFNetworkActivityIndicatorCompletionDelay =
0.17;
```

接着我们来看看三个通知触发调用的方法：

```
//请求开始
- (void)networkRequestDidStart:(NSNotification *)notification {

    if ([AFNetworkRequestFromNotification(notification) URL]) {
        //增加请求活跃数
        [self incrementActivityCount];
    }
}
//请求结束
- (void)networkRequestDidFinish:(NSNotification *)notification {
    //AFNetworkRequestFromNotification(notification)返回这个通知的request,用来判断request是否是有效的
    if ([AFNetworkRequestFromNotification(notification) URL]) {
        //减少请求活跃数
        [self decrementActivityCount];
    }
}
```

方法很简单，就是开始的时候增加了请求活跃数，结束则减少。调用了如下两个方法进行加减：

```

//增加请求活跃数
- (void)incrementActivityCount {

    //活跃的网络数+1, 并手动发送KVO
    [self willChangeValueForKey:@"activityCount"];
    @synchronized(self) {
        _activityCount++;
    }
    [self didChangeValueForKey:@"activityCount"];

    //主线程去做
    dispatch_async(dispatch_get_main_queue(), ^{
        [self updateCurrentStateForNetworkActivityChange];
    });
}

//减少请求活跃数
- (void)decrementActivityCount {
    [self willChangeValueForKey:@"activityCount"];
    @synchronized(self) {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wgnu"
        _activityCount = MAX(_activityCount - 1, 0);
#pragma clang diagnostic pop
    }
    [self didChangeValueForKey:@"activityCount"];

    dispatch_async(dispatch_get_main_queue(), ^{
        [self updateCurrentStateForNetworkActivityChange];
    });
}

```

方法做了什么应该很容易看明白，这里需要注意的是，**task**的几个状态的通知，是会在多线程的环境下发送过来的。所以这里对活跃数的加减，都用了 `@synchronized` 这种方式的锁，进行了线程保护。然后回到主线程调用了 `updateCurrentStateForNetworkActivityChange`

我们接着来看看这个方法：

```

- (void)updateCurrentStateForNetworkActivityChange {
    //如果是允许小菊花
    if (self.enabled) {
        switch (self.currentState) {
            //不活跃
            case AFNetworkActivityManagerStateNotActive:
                //判断活跃数，大于0为YES
                if (self.isNetworkActivityOccurring) {
                    //设置状态为延迟开始
                    [self
setCurrentState:AFNetworkActivityManagerStateDelayingStart];
                }
                break;

            case AFNetworkActivityManagerStateDelayingStart:
                //No op. Let the delay timer finish out.
                break;

            case AFNetworkActivityManagerStateActive:
                if (!self.isNetworkActivityOccurring) {
                    [self
setCurrentState:AFNetworkActivityManagerStateDelayingEnd];
                }
                break;

            case AFNetworkActivityManagerStateDelayingEnd:
                if (self.isNetworkActivityOccurring) {
                    [self
setCurrentState:AFNetworkActivityManagerStateActive];
                }
                break;
        }
    }
}

```

- 这个方法先是判断了我们一开始设置是否需要菊花的 `self.enabled`，如果需要，才执行。
- 这里主要是根据当前的状态，来判断下一个状态应该是什么。其中有这么一个属性 `self.isNetworkActivityOccurring`：

```

//判断是否活跃
- (BOOL)isNetworkActivityOccurring {
    @synchronized(self) {
        return self.activityCount > 0;
    }
}

```

那么这个方法应该不难理解了。

这个类复写了 `currentState` 的 `set` 方法，每当我们改变这个 `state`，就会触发 `set` 方法，而怎么该转菊花也在该方法中：

```

//设置当前小菊花状态
- (void)setCurrentState:(AFNetworkActivityManagerState)currentState {
    @synchronized(self) {
        if (_currentState != currentState) {
            //KVO
            [self willChangeValueForKey:@"currentState"];
            _currentState = currentState;
            switch (currentState) {
                //如果不活跃
                case AFNetworkActivityManagerStateNotActive:
                    //取消两个延迟用的timer
                    [self cancelActivationDelayTimer];
                    [self cancelCompletionDelayTimer];
                    //设置小菊花不可见
                    [self setNetworkActivityIndicatorVisible:NO];
                    break;
                case AFNetworkActivityManagerStateDelayingStart:
                    //开启一个定时器延迟去转菊花
                    [self startActivationDelayTimer];
                    break;
                //如果是活跃状态
                case AFNetworkActivityManagerStateActive:
                    //取消延迟完成的timer
                    [self cancelCompletionDelayTimer];
                    //开始转菊花
                    [self setNetworkActivityIndicatorVisible:YES];
                    break;
                    //延迟完成状态
                case AFNetworkActivityManagerStateDelayingEnd:
                    //开启延迟完成timer
                    [self startCompletionDelayTimer];
                    break;
            }
        }
        [self didChangeValueForKey:@"currentState"];
    }
}

```

这个set方法就是这个类最核心的方法了。它的作用如下：

- 这里根据当前状态，是否需要开始执行一个延迟开始或者延迟完成，又或者是否需要取消这两个延迟。
- 还判断了，是否需要去转状态栏的菊花，调用了 `setNetworkActivityIndicatorVisible:` 方法：

```

- (void)setNetworkActivityIndicatorVisible:
(BOOL)networkActivityIndicatorVisible {
    if (_networkActivityIndicatorVisible != networkActivityIndicatorVisible)
    {
        [self willChangeValueForKey:@"networkActivityIndicatorVisible"];
        @synchronized(self) {
            _networkActivityIndicatorVisible =
networkActivityIndicatorVisible;
        }
        [self didChangeValueForKey:@"networkActivityIndicatorVisible"];

        //支持自定义的Block, 去自己控制小菊花
        if (self.networkActivityIndicatorActionBlock) {
            self.networkActivityIndicatorActionBlock(networkActivityIndicatorVisible);
        } else {
            //否则默认AF根据该Bool, 去控制状态栏小菊花是否显示
            [[UIApplication sharedApplication]
setNetworkActivityIndicatorVisible:networkActivityIndicatorVisible];
        }
    }
}

```

- 这个方法就是用来控制菊花是否转。并且支持一个自定义的Block,我们可以自己去拿到这个菊花是否应该转的状态值, 去做一些自定义的处理。
- 如果我们没有实现这个Block, 则调用:

```

[[UIApplication sharedApplication]
setNetworkActivityIndicatorVisible:networkActivityIndicatorVisible];

```

去转菊花。

回到state的set方法中, 我们除了控制菊花去转, 还调用了以下4个方法:

```

//开始任务到结束的时间，默认为1秒，如果1秒就结束，那么不转菊花，延迟去开始转
- (void)startActivationDelayTimer {
    //只执行一次
    self.activationDelayTimer = [NSTimer
                                timerWithTimeInterval:self.activationDelay
target:self selector:@selector(activationDelayTimerFired) userInfo:nil
repeats:NO];
    //添加到主线程runloop去触发
    [[NSRunLoop mainRunLoop] addTimer:self.activationDelayTimer
forMode:NSRunLoopCommonModes];
}

//完成任务到下一个任务开始，默认为0.17秒，如果0.17秒就开始下一个，那么不停 延迟去结束菊花
转
- (void)startCompletionDelayTimer {
    //先取消之前的
    [self.completionDelayTimer invalidate];
    //延迟执行让菊花不在转
    self.completionDelayTimer = [NSTimer
                                timerWithTimeInterval:self.completionDelay target:self
selector:@selector(completionDelayTimerFired) userInfo:nil repeats:NO];
    [[NSRunLoop mainRunLoop] addTimer:self.completionDelayTimer
forMode:NSRunLoopCommonModes];
}

- (void)cancelActivationDelayTimer {
    [self.activationDelayTimer invalidate];
}

- (void)cancelCompletionDelayTimer {
    [self.completionDelayTimer invalidate];
}

```

这4个方法分别是开始延迟执行一个方法，和结束的时候延迟执行一个方法，和对应这两个方法的取消。其作用，注释应该很容易理解。

我们继续往下看，这两个延迟调用的到底是什么：

```

- (void)activationDelayTimerFired {
    //活跃状态，即活跃数大于1才转
    if (self.networkActivityOccurring) {
        [self setCurrentState:AFNetworkActivityManagerStateActive];
    } else {
        [self setCurrentState:AFNetworkActivityManagerStateNotActive];
    }
}

- (void)completionDelayTimerFired {
    [self setCurrentState:AFNetworkActivityManagerStateNotActive];
}

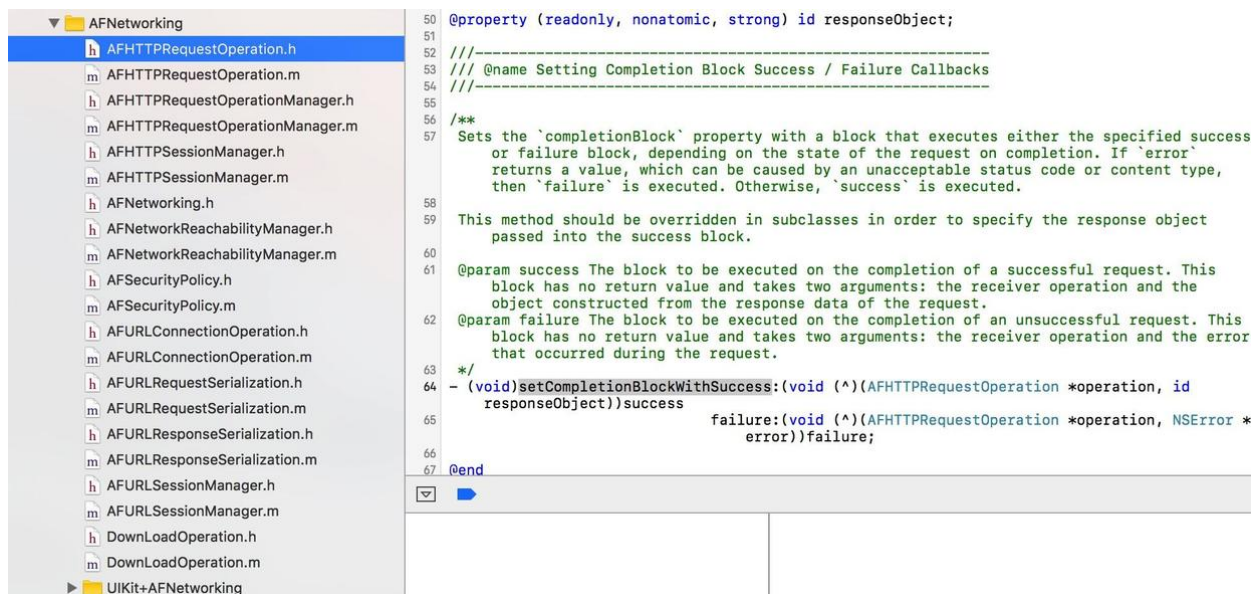
```

一个开始，一个完成调用，都设置了不同的currentState的值，又回到之前state的set方法中了。

至此这个AFNetworkActivityIndicatorManager类就讲完了，代码还是相当简单明了的。

## 四. AF2.x 的网络请求主流程：

首先我们来看看AF2.x的项目目录：



除了UIKit扩展外，大概就是上述这么多类，其中最重要的有3个类：

1) AFURLConnectionOperation

2) AFHTTPRequestOperation

3) AFHTTPRequestOperationManager

- 大家都知道，AF2.x是基于NSURLConnection来封装的，而NSURLConnection的创建以及数据请求，就被封装在AFURLConnectionOperation这个类中。所以这个类基本上是AF2.x最底层也是最核心的类。
- 而AFHTTPRequestOperation是继承自AFURLConnectionOperation，对它父类一些方法做了些封装。
- AFHTTPRequestOperationManager则是一个管家，去管理这些这些operation。

我们接下来按照网络请求的流程去看看AF2.x的实现：

注：本文会涉及一些NSOperationQueue、NSOperation方面的知识，如果对这方面的内容不了解的话，可以先看看雷纯峰的这篇：

[iOS 并发编程之 Operation Queues

](<http://blog.leichunfeng.com/blog/2015/07/29/ios-concurrency-programming-operation-queues/>)

首先，我们来写一个get或者post请求：

```

AFHTTPRequestOperationManager *manager = [AFHTTPRequestOperationManager
manager];
[manager GET:url parameters:params
    success:^(AFHTTPRequestOperation *operation, id responseObject) {

    } failure:^(AFHTTPRequestOperation *operation, NSError *error) {

    }]];

```

就这么简单的几行代码，完成了一个网络请求。

接着我们来看看 `AFHTTPRequestOperationManager` 的初始化方法：

```

+ (instancetype)manager {
    return [[self alloc] initWithBaseURL:nil];
}

- (instancetype)init {
    return [self initWithBaseURL:nil];
}

- (instancetype)initWithBaseURL:(NSURL *)url {
    self = [super init];
    if (!self) {
        return nil;
    }
    // Ensure terminal slash for baseURL path, so that NSURL
+URLWithString:relativeToURL: works as expected
    if ([[url path] length] > 0 && ![url absoluteString] hasSuffix:@"/*"]) {
        url = [url URLByAppendingPathComponent:@""];
    }
    self.baseURL = url;
    self.requestSerializer = [AFHTTPRequestSerializer serializer];
    self.responseSerializer = [AFJSONResponseSerializer serializer];
    self.securityPolicy = [AFSecurityPolicy defaultPolicy];
    self.reachabilityManager = [AFNetworkReachabilityManager sharedManager];
    //用来调度所有请求的queue
    self.operationQueue = [[NSOperationQueue alloc] init];
    //是否做证书验证
    self.shouldUseCredentialStorage = YES;
    return self;
}

```

初始化方法很简单，基本和AF3.x类似，除了一下两点：

- (1) 设置了一个 `operationQueue`，这个队列，用来调度里面所有的 `operation`，在AF2.x中，每一个 `operation` 就是一个网络请求。
- (2) 设置 `shouldUseCredentialStorage` 为YES，这个后面会传给 `operation`，`operation` 会根据这个值，去返回给代理，系统是否做https的证书验证。



然后我们来看看get方法：

```
- (AFHTTPRequestOperation *)GET:(NSString *)URLString
                        parameters:(id)parameters
                        success:(void (^)(AFHTTPRequestOperation *operation,
id responseObject))success
                        failure:(void (^)(AFHTTPRequestOperation *operation,
NSError *error))failure
{
    //拿到request
    NSMutableURLRequest *request = [self.requestSerializer
requestWithMethod:@"GET" urlString:[NSURL URLWithString:URLString
relativeToURL:self.baseURL] absoluteString] parameters:parameters error:nil];

    AFHTTPRequestOperation *operation = [self
HTTPRequestOperationWithRequest:request success:success failure:failure];

    [self.operationQueue addOperation:operation];
    return operation;
}
```

方法很简单，如下：

(1) 用 `self.requestSerializer` 生成了一个request，至于如何生成，可以参考之前的文章，这里就不赘述了。

(2) 生成了一个 `AFHTTPRequestOperation`，然后把这个 `operation` 加到我们一开始创建的 `queue` 中。

其中创建 `AFHTTPRequestOperation` 方法如下：

```

- (AFHTTPRequestOperation *)HTTPRequestOperationWithRequest:(NSURLRequest
*)request
                                success:(void (^)
(AFHTTPRequestOperation *operation, id responseObject))success
                                failure:(void (^)
(AFHTTPRequestOperation *operation, NSError *error))failure
{
    //创建自定义的AFHTTPRequestOperation
    AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation alloc]
initWithRequest:request];
    operation.responseSerializer = self.responseSerializer;
    operation.shouldUseCredentialStorage = self.shouldUseCredentialStorage;
    operation.credential = self.credential;
    //设置自定义的安全策略
    operation.securityPolicy = self.securityPolicy;

    [operation setCompletionBlockWithSuccess:success failure:failure];
    operation.completionQueue = self.completionQueue;
    operation.completionGroup = self.completionGroup;
    return operation;
}

```

方法创建了一个 `AFHTTPRequestOperation`，并把自己的一些参数交给了这个 `operation` 处理。

接着往里看：

```

- (instancetype)initWithRequest:(NSURLRequest *)urlRequest {
    self = [super initWithRequest:urlRequest];
    if (!self) {
        return nil;
    }

    self.responseSerializer = [AFHTTPResponseSerializer serializer];
    return self;
}

```

除了设置了一个 `self.responseSerializer`，实际上是调用了父类，也是我们最核心的类 `AFURLConnectionOperation` 的初始化方法，首先我们要明确这个类是继承自 `NSOperation` 的，然后我们接着往下看：

```

//初始化
- (instancetype)initWithRequest:(NSURLRequest *)urlRequest {
    NSParameterAssert(urlRequest);

    self = [super init];
    if (!self) {
        return nil;
    }

    //设置为ready
    _state = AFOperationReadyState;
    //递归锁
    self.lock = [[NSRecursiveLock alloc] init];
    self.lock.name = kAFNetworkingLockName;
    self.runLoopModes = [NSSet setWithObject:NSRunLoopCommonModes];
    self.request = urlRequest;

    //是否应该咨询证书存储连接
    self.shouldUseCredentialStorage = YES;

    //https认证策略
    self.securityPolicy = [AFSecurityPolicy defaultPolicy];

    return self;
}

```

初始化方法中，初始化了一些属性，下面我们来简单的介绍一下这些属性：

## 1. `_state` 设置为 `AFOperationReadyState` 准备就绪状态

这是个枚举：

```

typedef NS_ENUM(NSInteger, AFOperationState) {
    AFOperationPausedState      = -1,    //停止
    AFOperationReadyState       = 1,     //准备就绪
    AFOperationExecutingState    = 2,     //正在进行中
    AFOperationFinishedState    = 3,     //完成
};

```

这个 `_state` 标志着这个网络请求的状态，一共如上4种状态。这些状态其实对应着 `operation` 如下  
的状态：

```
//映射这个operation的各个状态
static inline NSString * AFKeyPathFromOperationState(AFOperationState state)
{
    switch (state) {
        case AFOperationReadyState:
            return @"isReady";
        case AFOperationExecutingState:
            return @"isExecuting";
        case AFOperationFinishedState:
            return @"isFinished";
        case AFOperationPausedState:
            return @"isPaused";
        default: {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wunreachable-code"
            return @"state";
#pragma clang diagnostic pop
        }
    }
}
```

并且还复写了这些属性的get方法，用来和自定义的state一一对应：

```
//复写这些方法，与自己的定义的state对应
- (BOOL)isReady {
    return self.state == AFOperationReadyState && [super isReady];
}
- (BOOL)isExecuting {
    return self.state == AFOperationExecutingState;
}
- (BOOL)isFinished {
    return self.state == AFOperationFinishedState;
}
```

## 2. `self.lock` 锁

这个锁是用来提供给本类一些数据操作的线程安全，至于为什么要用递归锁，是因为有些方法可能会存在递归调用的情况，例如有些需要锁的方法可能会在一个大的操作环中，形成递归。而AF使用了递归锁，避免了这种情况下死锁的发生。

## 3. 初始化了 `self.runLoopModes`，默认为 `NSRunLoopCommonModes`。

## 4. 生成了一个默认的 `self.securityPolicy`。

这个类为了自定义 `operation` 的各种状态，而且更好的掌控它的生命周期，复写了 `operation` 的 `start` 方法，当这个 `operation` 在一个新线程被调度执行的时候，首先就调入这个 `start` 方法中，接下来我们看看它的实现看看：

```
- (void)start {
    [self.lock lock];

    //如果被取消了就调用取消的方法
    if ([self isCancelled]) {
        //在AF常驻线程中去执行
        [self performSelector:@selector(cancelConnection) onThread:[self
class] networkRequestThread] withObject:nil waitUntilDone:NO modes:
[self.runLoopModes allObjects]];
    }
    //准备好了，才开始
    else if ([self isReady]) {
        //改变状态，开始执行
        self.state = AFOperationExecutingState;
        [self performSelector:@selector(operationDidStart) onThread:[self
class] networkRequestThread] withObject:nil waitUntilDone:NO modes:
[self.runLoopModes allObjects]];
    }
    //注意，发起请求和取消请求都是在同一个线程！！包括回调都是在一个线程

    [self.lock unlock];
}
```

这个方法判断了当前的状态，是取消还是准备就绪，然后去调用了各自对应的方法。

- 注意这些方法都是在另外一个线程中去调用的，我们来看看这个线程：

```

+ (void)networkRequestThreadEntryPoint:(id)__unused object {
    @autoreleasepool {
        [[NSThread currentThread] setName:@"AFNetworking"];

        NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
        //添加端口, 防止runloop直接退出
        [runLoop addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode];
        [runLoop run];
    }
}

+ (NSThread *)networkRequestThread {
    static NSThread *_networkRequestThread = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _networkRequestThread = [[NSThread alloc] initWithTarget:self
        selector:@selector(networkRequestThreadEntryPoint:) object:nil];
        [_networkRequestThread start];
    });

    return _networkRequestThread;
}

```

这两个方法基本上是被许多人举例用过无数次了...

- 这是一个单例, 用 `NSThread` 创建了一个线程, 并且为这个线程添加了一个 `runloop`, 并且加了一个 `NSMachPort`, 来防止 `runloop` 直接退出。
- 这条线程就是 **AF** 用来发起网络请求, 并且接受网络请求回调的线程, 仅仅就这一条线程 (到最后我们来讲为什么要这么做)。和我们之前讲的 AF3.x 发起请求, 并且接受请求回调时的处理方式, 遥相呼应。

我们接着来看如果准备就绪, start调用的方法:

```

//改变状态, 开始执行
self.state = AFOperationExecutingState;
[self performSelector:@selector(operationDidStart) onThread:[self class]
networkRequestThread] withObject:nil waitUntilDone:NO modes:
[self.runLoopModes allObjects]];

```

接着在常驻线程中, 并且不阻塞的方式, 在我们 `self.runLoopModes` 的模式下调用:

```

- (void)operationDidStart {
    [self.lock lock];
    //如果没取消
    if (![self isCancelled]) {
        //设置为startImmediately YES 请求发出, 回调会加入到主线程的 Runloop 下,
        RunloopMode 会默认为 NSDefaultRunLoopMode
        self.connection = [[NSURLConnection alloc]
initWithRequest:self.request delegate:self startImmediately:NO];

        NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
        for (NSString *runLoopMode in self.runLoopModes) {
            //把connection和outputStream注册到当前线程runloop中去, 只有这样, 才能在这个线程中回调
            [self.connection scheduleInRunLoop:runLoop forMode:runLoopMode];
            [self.outputStream scheduleInRunLoop:runLoop
forMode:runLoopMode];
        }
        //打开输出流
        [self.outputStream open];
        //开启请求
        [self.connection start];
    }
    [self.lock unlock];
    dispatch_async(dispatch_get_main_queue(), ^{
        [[NSNotificationCenter defaultCenter]
postNotificationName:AFNetworkingOperationDidStartNotification object:self];
    });
}

```

这个方法做了以下几件事：

(1). 首先这个方法创建了一个 `NSURLConnection`，设置代理为自己，startImmediately为NO，至于这个参数干什么用的，我们来看看官方文档：

```

startImmediately
YES if the connection should begin loading data immediately, otherwise NO. If you pass NO, the connection is not scheduled with a run loop. You can then schedule the connection in the run loop and mode of your choice by calling scheduleInRunLoop:forMode: .

```

大意是，这个值默认为YES，而且任务完成的结果会在主线程的runloop中回调。如果我们设置为NO，则需要调用我们下面看到的：

```

[self.connection scheduleInRunLoop:runLoop forMode:runLoopMode];

```

去注册一个runloop和mode，它会在我们指定的这个runloop所在的线程中回调结果。

(2). 值得一提的是这里调用了：

```
[self.outputStream scheduleInRunLoop:runLoop forMode:runLoopMode];
```

这个 `outputStream` 在 `get` 方法中被初始化了：

```
- (NSOutputStream *)outputStream {
    if (!_outputStream) {
        //一个写入到内存中的流，可以通过NSStreamDataWrittenToMemoryStreamKey拿到写入后的数据
        self.outputStream = [NSOutputStream outputStreamToMemory];
    }
    return _outputStream;
}
```

这里数据请求和拼接并没有用 `NSMutableData`，而是用了 `outputStream`，而且把写入的数据，放到内存中。

- 其实讲道理来说 `outputStream` 的优势在于下载大文件的时候，可以以流的形式，将文件直接保存到本地，这样可以为我们节省很多的内存，调用如下方法设置：

```
[NSOutputStream outputStreamToFileAtPath:@"filePath" append:YES];
```

- 但是这里是把流写入内存中，这样其实这个节省内存的意义已经不存在了。那为什么还要用呢？这里我猜测的就是为了用它这个可以注册在某一个 `runloop` 的指定 `mode` 下。虽然AF使用这个 `outputStream` 是肯定在这个常驻线程中的，不会有线程安全的问题。但是要注意它是被声明在.h中的：

```
@property (nonatomic, strong) NSOutputStream *outputStream;
```










难保外部不会在其他线程对这个数据做什么操作，所以它相对于 `NSMutableData` 作用就体现出来了，就算我们在外部其它线程中去操作它，也不会有线程安全的问题。

(3). 这个 `connection` 开始执行了。

(4). 到主线程发送一个任务开始执行的通知。

接下来网络请求开始执行了，就开始触发 `connection` 的代理方法了：

#### **NSURLConnectionDelegate**

-  `-connection:willSendRequestForAuthenticationChallenge:`
-  `-connectionShouldUseCredentialStorage:`
-  `-connection:willSendRequest:redirectResponse:`
-  `-connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite:`
-  `-connection:didReceiveResponse:`
-  `-connection:didReceiveData:`
-  `-connectionDidFinishLoading:`
-  `-connection:didFailWithError:`
-  `-connection:willCacheResponse:`



AF2.x一共实现了如上这么多代理方法，这些代理方法，作用大部分和我们之前讲的 `NSURLSession` 的代理方法类似，我们只挑几个去讲，如果需要了解其他的方法作用，可以参考楼主之前的文章。

重点讲下面这四个代理：

注意，有一点需要说明，我们之前是把connection注册在我们常驻线程的runloop中了，所以以下所有的代理方法，都是在这仅有的一条常驻线程中回调。

第一个代理

```
//收到响应，响应头类似相关数据
- (void)connection:(NSURLConnection __unused *)connection
didReceiveResponse:(NSURLResponse *)response
{
    self.response = response;
}
```

没什么好说的，就是收到响应后，把response赋给自己的属性。

第二个代理

```
//拼接获取到的数据
- (void)connection:(NSURLConnection __unused *)connection
didReceiveData:(NSData *)data
{
    NSUInteger length = [data length];
    while (YES) {
        NSInteger totalNumberOfBytesWritten = 0;
        //如果outputStream 还有空余空间
        if ([self.outputStream hasSpaceAvailable]) {

            //创建一个buffer流缓冲区，大小为data的字节数
            const uint8_t *dataBuffer = (uint8_t *)[data bytes];

            NSInteger numberOfBytesWritten = 0;

            //当写的长度小于数据的长度，在循环里
            while (totalNumberOfBytesWritten < (NSInteger)length) {
                //往outputStream写数据，系统的方法，一次就写一部分，得循环写
                numberOfBytesWritten = [self.outputStream
write:&dataBuffer[(NSInteger)totalNumberOfBytesWritten] maxLength:(length -
(NSUInteger)totalNumberOfBytesWritten)];
                //如果 numberOfBytesWritten写入失败了。跳出循环
                if (numberOfBytesWritten == -1) {
                    break;
                }
                //加上每次写的长度
                totalNumberOfBytesWritten += numberOfBytesWritten;
            }
        }
    }
}
```

```

        break;
    }

    //出错
    if (self.outputStream.streamError) {
        //取消connection
        [self.connection cancel];
        //调用失败的方法
        [self performSelector:@selector(connection:didFailWithError:)
         withObject:self.connection withObject:self.outputStream.streamError];
        return;
    }
}

//主线程回调下载数据大小
dispatch_async(dispatch_get_main_queue(), ^{
    self.totalBytesRead += (long long)length;

    if (self.downloadProgress) {
        self.downloadProgress(length, self.totalBytesRead,
self.response.expectedContentLength);
    }
});
}

```

这个方法看起来长，其实容易理解而且简单，它只做了3件事：

1. 给 `outputStream` 拼接数据，具体如果拼接，大家可以读注释自行理解下。
2. 如果出错则调用： `connection:didFailWithError:` 也就是网络请求失败的代理，我们一会下面就会讲。
3. 在主线程中回调下载进度。

第三个代理

```

//完成了调用
- (void)connectionDidFinishLoading:(NSURLConnection __unused *)connection {

    //从outputStream中拿到数据 NSDataWrittenToMemoryStreamKey写入到内存中的流
    self.responseData = [self.outputStream
propertyForKey:NSStreamDataWrittenToMemoryStreamKey];

    //关闭outputStream
    [self.outputStream close];

    //如果响应数据已经有了，则outputStream置为nil
    if (self.responseData) {
        self.outputStream = nil;
    }
    //清空connection
    self.connection = nil;
    [self finish];
}

```

这个代理是任务完成之后调用。我们从 `outputStream` 拿到了最后下载数据，然后关闭置空了 `outputStream`。并且清空了 `connection`。调用了 `finish`：

```

- (void)finish {
    [self.lock lock];
    //修改状态
    self.state = AFOperationFinishedState;
    [self.lock unlock];

    //发送完成的通知
    dispatch_async(dispatch_get_main_queue(), ^{
        [[NSNotificationCenter defaultCenter]
postNotificationName:AFNetworkingOperationDidFinishNotification object:self];
    });
}

```

把当前任务状态改为已完成，并且到主线程发送任务完成的通知。这里我们设置状态为已完成。其实调用了我们本类复写的 `set` 的方法（前面遗漏了，在这里补充）：

```

- (void)setState:(AFOperationState)state {

    //判断从当前状态到另一个状态是不是合理，在加上现在是否取消。。大神的框架就是屌啊，这判断严谨的。。一层层
    if (!AFStateTransitionIsValid(self.state, state, [self isCancelled])) {
        return;
    }

    [self.lock lock];

    //拿到对应的父类管理当前线程周期的key
    NSString *oldStateKey = AFKeyPathFromOperationState(self.state);
    NSString *newStateKey = AFKeyPathFromOperationState(state);

    //发出kvo
    [self willChangeValueForKey:newStateKey];
    [self willChangeValueForKey:oldStateKey];
    _state = state;
    [self didChangeValueForKey:oldStateKey];
    [self didChangeValueForKey:newStateKey];
    [self.lock unlock];
}

```

这个方法改变 `state` 的时候，并且发送了 `KVO`。大家了解 `NSOperationQueue` 就知道，如果对应的 `operation` 的属性 `finished` 被设置为 YES，则代表当前 `operation` 结束了，会把 `operation` 从队列中移除，并且调用 `operation` 的 `completionBlock`。这点很重要，因为我们请求到的数据就是从这个 `completionBlock` 中传递回去的（下面接着讲这个完成Block，就能从这里对接上了）。

第四个代理

```

//请求失败的回调，在cancel connection的时候，自己也主动调用了
- (void)connection:(NSURLConnection __unused *)connection
didFailWithError:(NSError *)error
{
    //拿到error
    self.error = error;
    //关闭outputStream
    [self.outputStream close];
    //如果响应数据已经有了，则outputStream置为nil
    if (self.responseData) {
        self.outputStream = nil;
    }
    self.connection = nil;
    [self finish];
}

```

唯一需要说一下的就是这里给 `self.error` 赋值，之后完成Block会根据这个error，去判断这次请求是成功还是失败。

至此我们把 `AFURLConnectionOperation` 的业务主线讲完了。

我们此时数据请求完了，数据在 `self.responseData` 中，接下来我们来看它是怎么回到我们手里的。

我们回到 `AFURLConnectionOperation` 子类 `AFHTTPRequestOperation`，有这么一个方法：

```
- (void)setCompletionBlockWithSuccess:(void (^)(AFHTTPRequestOperation
*operation, id responseObject))success
                                failure:(void (^)(AFHTTPRequestOperation
*operation, NSError *error))failure
{
    // completionBlock is manually nilled out in AFURLConnectionOperation to
    break the retain cycle.
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-retain-cycles"
#pragma clang diagnostic ignored "-Wgnu"
    self.completionBlock = ^{
        if (self.completionGroup) {
            dispatch_group_enter(self.completionGroup);

            dispatch_async(http_request_operation_processing_queue(), ^{
                if (self.error) {
                    if (failure) {
                        dispatch_group_async(self.completionGroup ?:
http_request_operation_completion_group(), self.completionQueue ?:
dispatch_get_main_queue(), ^{
                            failure(self, self.error);
                        });
                    }
                } else {
                    id responseObject = self.responseObject;
                    if (self.error) {
                        if (failure) {
                            dispatch_group_async(self.completionGroup ?:
http_request_operation_completion_group(), self.completionQueue ?:
dispatch_get_main_queue(), ^{
                                    failure(self, self.error);
                                });
                        }
                    } else {
                        if (success) {
                            dispatch_group_async(self.completionGroup ?:
http_request_operation_completion_group(), self.completionQueue ?:
dispatch_get_main_queue(), ^{
                                    success(self, responseObject);
                                });
                        }
                    }
                }
            });
        }
    }
```

```

    }

    if (self.completionGroup) {
        dispatch_group_leave(self.completionGroup);
    }
    });
};
#pragma clang diagnostic pop
}

```

一开始我们在 `AFHTTPRequestOperationManager` 中是调用过这个方法的：

```
[operation setCompletionBlockWithSuccess:success failure:failure];
```

- 我们在把成功和失败的Block传给了这个方法。
- 这个方法也很好理解，就是设置我们之前提到过得 `completionBlock`，当自己数据请求完成，就会调用这个Block。然后我们在这个Block中调用传过来的成功或者失败的Block。如果error为空，说明请求成功，把数据传出去，否则为失败，把error信息传出。
- 这里也类似AF3.x，可以自定义一个完成组和完成队列。数据可以在我们自定义的完成组和队列中回调出去。
- 除此之外，还有一个有意思的地方：

```

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-retain-cycles"
#pragma clang diagnostic ignored "-Wgnu"
#pragma clang diagnostic pop

```

之前我们说过，这是在忽略编译器的一些警告。

- `-Wgnu` 就不说了，是忽略？：。
- 值得提下的是 `-Warc-retain-cycles`，这里忽略了循环引用的警告。我们仔细看看就知道 `self` 持有了 `completionBlock`，而 `completionBlock` 内部持有 `self`。这里确实循环引用了。那么AF是如何解决这个循环引用的呢？

我们在回到 `AFURLConnectionOperation`，还有一个方法我们之前没讲到，它复写了 `setCompletionBlock`这个方法。

```

//复写setCompletionBlock
- (void)setCompletionBlock:(void (^)(void))block {
    [self.lock lock];
    if (!block) {
        [super setCompletionBlock:nil];
    } else {
        __weak __typeof(self)weakSelf = self;
        [super setCompletionBlock:^(
            __strong __typeof(weakSelf)strongSelf = weakSelf;

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wgnu"
            //看有没有自定义的完成组, 否则用AF的组
            dispatch_group_t group = strongSelf.completionGroup ?:
url_request_operation_completion_group();
            //看有没有自定义的完成queue, 否则用主队列
            dispatch_queue_t queue = strongSelf.completionQueue ?:
dispatch_get_main_queue();
#pragma clang diagnostic pop

            //调用设置的Block, 在这个组和队列中
            dispatch_group_async(group, queue, ^{
                block();
            });

            //结束时候置nil, 防止循环引用
            dispatch_group_notify(group,
url_request_operation_completion_queue(), ^{
                [strongSelf setCompletionBlock:nil];
            });
        }];
    }
    [self.lock unlock];
}

```

注意, 它在我们设置的block调用结束的时候, 主动的调用:

```
[strongSelf setCompletionBlock:nil];
```

把Block置空, 这样循环引用不复存在了。

好像我们还遗漏了一个东西, 就是返回的数据做类型的解析。其实还真不是楼主故意这样东一块西一块的, AF2.x有些代码确实是这样零散。。当然仅仅是相对3.x来说。AFNetworking整体代码质量, 以及架构思想已经强过绝大多数开源项目太多了。。这一点毋庸置疑。

我们来接着看看数据解析在什么地方被调用的把:

```

- (id)responseObject {
    [self.lock lock];
    if (!_responseObject && [self isFinished] && !self.error) {
        NSError *error = nil;
        //做数据解析
        self.responseObject = [self.responseSerializer
responseObjectForResponse:self.response data:self.responseData error:&error];
        if (error) {
            self.responseSerializationError = error;
        }
    }
    [self.lock unlock];
    return _responseObject;
}

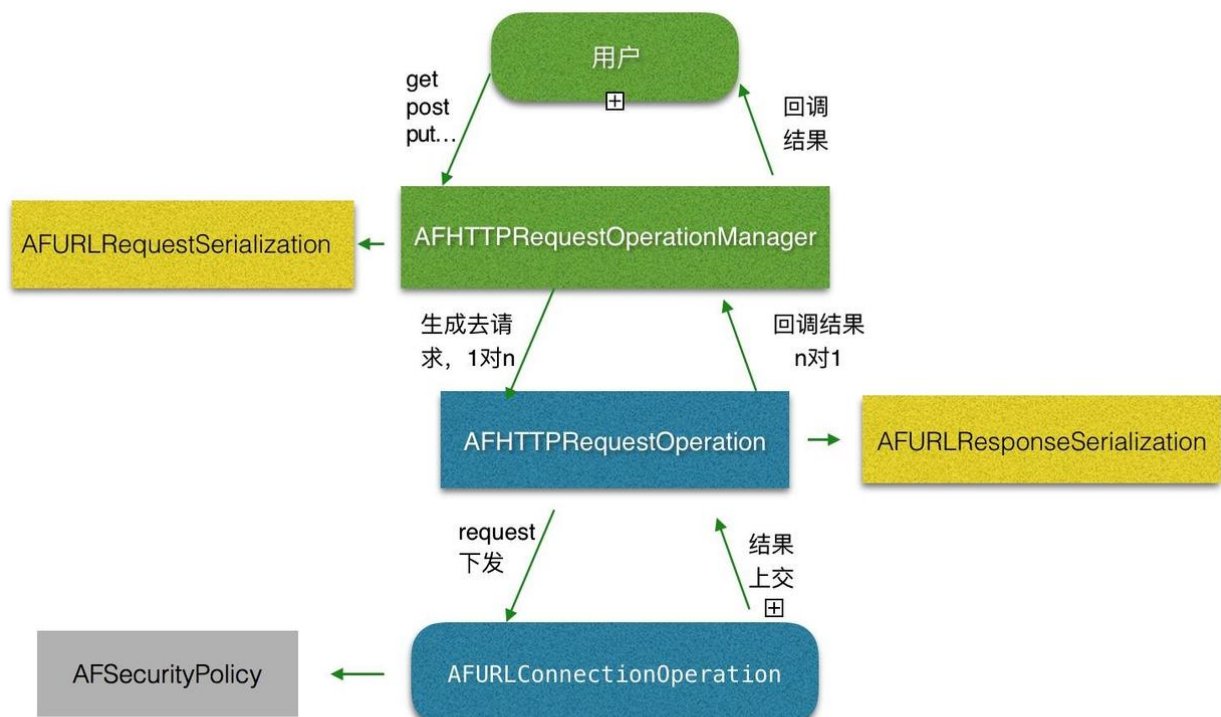
```

`AFHTTPRequestOperation` 复写了 `responseObject` 的get方法，并且把数据按照我们需要的类型（json、xml等等）进行解析。至于如何解析，可以参考楼主之前AF系列的文章，这里就不赘述了。

有些小伙伴可能会说，楼主你是不是把 `AFSecurityPolicy` 给忘了啊，其实并没有，它被在 `AFURLConnectionOperation` 中https认证的代理中被调用，我们之前系列的文章已经讲的非常详细了，感兴趣的朋友可以翻到前面的文章去看看。

至此，AF2.x整个业务流程就结束了。

接下来，我们来总结总结AF2.x整个业务请求的流程：



如上图，我们来梳理一下整个流程：



- 最上层的是 `AFHTTPRequestOperationManager`，我们调用它进行get、post等等各种类型的网络请求
- 然后它去调用 `AFURLRequestSerialization` 做request参数拼装。然后生成了一个 `AFHTTPRequestOperation` 实例，并把request交给它。然后把 `AFHTTPRequestOperation` 添加到一个 `NSOperationQueue` 中。
- 接着 `AFHTTPRequestOperation` 拿到request后，会去调用它的父类 `AFURLConnectionOperation` 的初始化方法，并且把相关参数交给它，除此之外，当父类完成数据请求后，它调用了 `AFURLResponseSerialization` 把数据解析成我们需要的格式（json、XML等等）。
- 最后就是我们AF最底层的类 `AFURLConnectionOperation`，它去数据请求，并且如果是https请求，会在请求的相关代理中，调用 `AFSecurityPolicy` 做https认证。最后请求到的数据返回。

这就是AF2.x整个做网络请求的业务流程。

我们来解决解决之前遗留下来的问题：为什么AF2.x需要一条常驻线程？

首先如果我们用 `NSURLConnection`，我们为了获取请求结果有以下三种选择：

1. 在主线程调异步接口
2. 每一个请求用一个线程，对应一个runloop，然后等待结果回调。
3. 只用一条线程，一个runloop，所有结果回调在这个线程上。

很显然AF选择的是第3种方式，创建了一条常驻线程专门处理所有请求的回调事件，这个模型跟 `nodejs` 有点类似，我们来讨论讨论不选择另外两种方式的原因：

## 原因一：

试想如果我们所有的请求都在主线程中异步调用，好像没什么不可以？那为什么AF不这么做呢...在这里有两点原因（楼主个人总结的，有不同意见，欢迎讨论）：

第一是，如果我们放到主线程去做，势必要这么写：

```
[[NSURLConnection alloc] initWithRequest:request delegate:self
startImmediately:YES]
```

这样NSURLConnection的回调会被放在主线程中 `NSDefaultRunLoopMode` 中，这样我们在其它类似 `UITrackingRunLoopMode` 模式下，我们是得不到网络请求的结果的，这显然不是我们想要的，那么我们势必需要调用：

```
[connection scheduleInRunLoop:[NSRunLoop currentRunLoop]
forMode:NSRunLoopCommonModes];
```

把它加入 `NSRunLoopCommonModes` 中，试想如果有大量的网络请求，同时回调回来，就会影响我们的UI体验了。

## 原因二：

另外一点原因是，如果我们请求数据返回，势必要进行数据解析，解析成我们需要的格式，那么这些解析都在主线程中做，给主线程增加额外的负担。

又或者我们回调回来开辟一个新的线程去做数据解析，那么我们有n个请求回来开辟n条线程带来的性能损耗，以及线程间切换带来的损耗，是不是一笔更大的开销。

所以综述两点原因，我们并不适合在主线程中回调。

我们一开始就开辟n条线程去做请求，然后设置runloop保活住线程，等待结果回调。

其实看到这，大家想想都觉得这个方法很傻，为了等待不确定的请求结果，阻塞住线程，白白浪费n条线程的开销。

综上所述，这就是AF2.x需要一条常驻线程的原因了。

至此我们把AF2.x核心流程分析完了。

## 五. 本文总结：AFNetworking到底做了什么？

相信如果从头看到尾的小伙伴，心里都有了一个属于自己的答案。其实在作者心里，并不想去以一言之词总结它，因为AFNetworking中凝聚了太多大牛的思想，根本不是看完几遍源码所能去议论的。但是想想也知道，如果我说不总结，估计有些看到这的朋友杀人的心都有...所以我还是赶鸭子上架，来总结总结它。

### 1. 首先我们需要明确一点的是：

相对于AFNetworking2.x，AFNetworking3.x确实没那么有用了。AFNetworking之前的核心作用就是为了帮我们去调度所有的请求。但是最核心地方却被苹果的NSURLSession给借鉴过去了，嗯...是借鉴。这些请求的调度，现在完全由NSURLSession给做了，AFNetworking3.x的作用被大大的削弱了。

### 2. 但是除此之外，其实它还是很有用的：

- 首先它帮我们做了各种请求方式request的拼接。想想如果我们用NSURLSession，我们去做请求，是不是还得自己去考虑各种请求方式下，拼接参数的问题。
- 它还帮我们做了一些公用参数（session级别的），和一些私用参数（task级别的）的分离。它用Block的形式，支持我们自定义一些代理方法，如果没有实现的话，AF还帮我们做了一些默认的处理。而如果我们用NSURLSession的话，还得参照AF这么一套代理转发的架构模式去封装。
- 它帮我们做了自定义的https认证处理。看了之前讲的AF的安全策略的朋友就知道，如果我们自己用NSURLSession实现那几种自定义认证，需要多写多少代码...
- 对于请求到的数据，AF帮我们做了各种格式的数据解析，并且支持我们设置自定义的code范围，自定义的数据方式。如果不在这些范围中，则直接调用失败block。如果用NSURLSession呢？这些都自己去写吧...（你要是做过各种除json外其他的数据解析，就会知道这里面坑有多少...）
- 对于成功和失败的回调处理。AF帮我们在数据请求到，到回调给用户之间，做了各种错误的判断，保证了成功和失败的回调，界限清晰。在这过程中，AF帮我们做了太多的容错处理，而NSURLSession呢？只给了一个完成的回调，我们得多做多少判断，才能拿到一个确定能正常显示的数据？
- .....

- 光是这些网络请求的业务逻辑，AF帮我们做的就太多太多，当然还远不仅于此。它用凝聚着许多大牛的经验方式，帮我在有些处理中做了最优的选择，比如我们之前说到的，回调线程数设置为1的问题...帮我们绕开了很多的坑，比如系统内部并行创建 `task` 导致id不唯一等等...

### 3. 而如果我们需要一些UIKit的扩展，AF则提供了最稳定，而且最优化实现方式：

就比如之前说到过得那个状态栏小菊花，如果是我们自己去做，得多写多少代码，而且实现的还没有AF那样质量高。

又或者 `AFImageDownloader`，它对于组图片之间的下载协调，以及缓存使用的之间线程调度。对于线程，锁，以及性能各方面权衡，找出最优化的处理方式，试问小伙伴们自己基于 `NSURLSession` 去写，能到做几分...

所以最后的结论是：**AFNetworking**虽然变弱了，但是它还是很有用的。用它真的不仅仅是习惯，而是因为它确实帮我们做了太多。

# SpriteKit 入门与实践

作者：郭鹏

作为一个 iOS 开发，你应该知道一个 app 是怎样从无到有，但对于游戏却不一定。所以本文将带你了解游戏开发方面的知识。

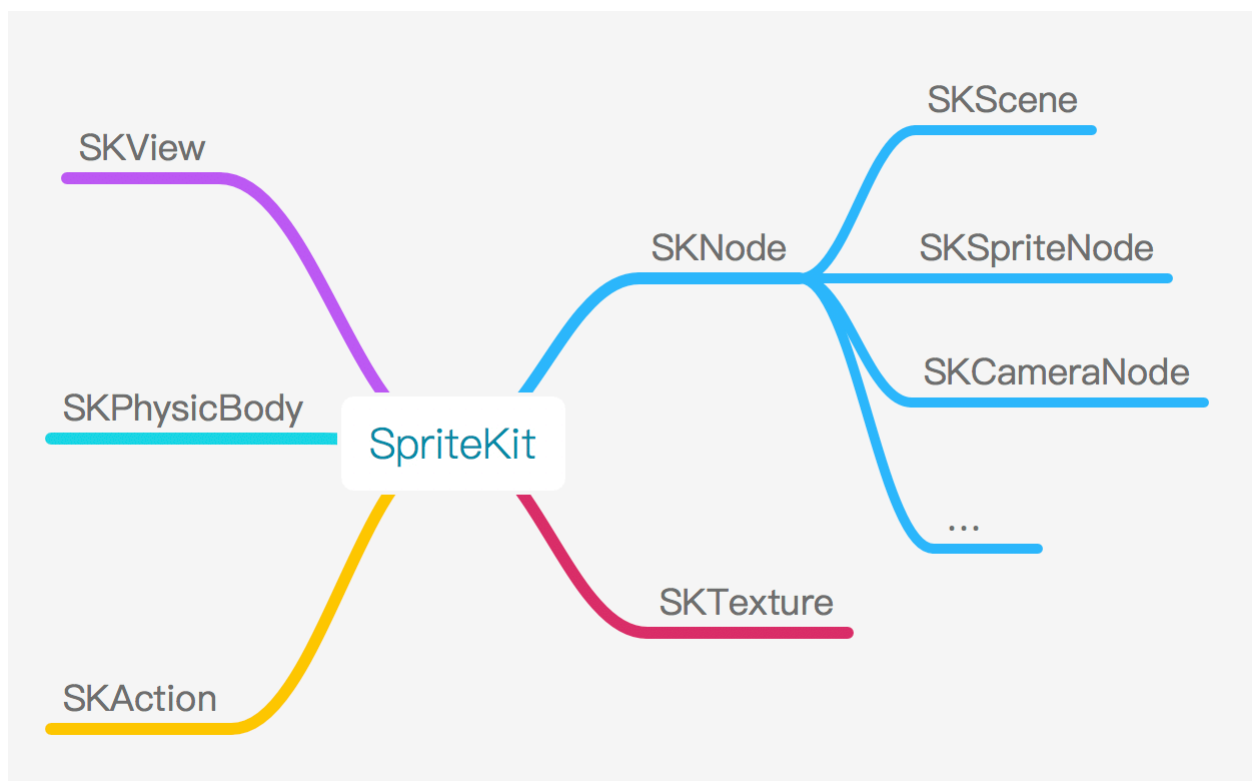
## 为什么是 SpriteKit

首先，我们需要了解目前手机游戏开发的现状。目前手机游戏引擎最流行的是 Cocos2d-x 和 Unity，它们都是跨平台的引擎，然而这两者对于初学者来说，需要额外学习的东西太多。

SpriteKit 是由 Apple 在 WWDC 2013 发布 2D 游戏引擎，目前可用于 iOS, macOS, tvOS 和 watchOS。它有着良好的设计、简单易用、API 和 Cocos2D 非常像。对于已经熟悉 iOS 开发的人来说，其用法和 UIKit 差不多，而且，他并非只能用来开发游戏。本文后面就会用 SpriteKit 实现类似 iTunes Music 的风格选择交互 UI。

## SpriteKit Framework 介绍

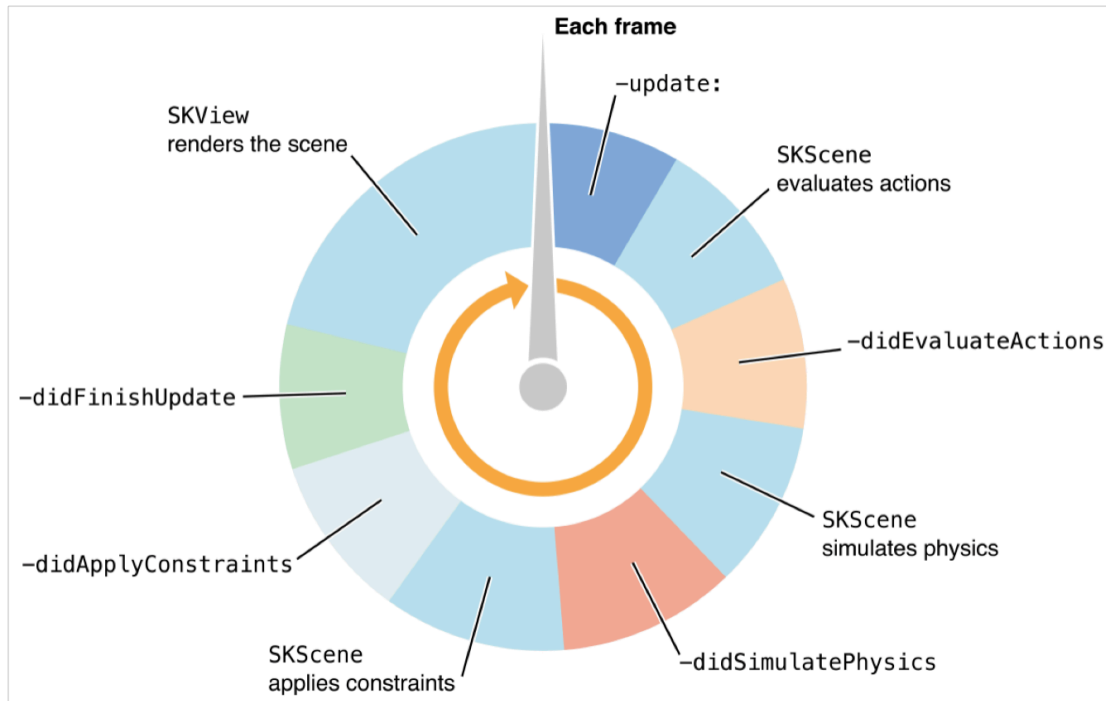
首先我们来看 SpriteKit 主要类结构。



- `SKView` 用来管理和渲染 `SKScene`，继承自 `UIView`。你甚至可以将 `SKView` 与 `UIKit` 里的其他 `View` 结合使用，比如在其之上加一个 `UIButton`。
- `SKNode` 是基础类，和 `UIKit` 类似，`SpriteKit` 有 `node trees` 的概念，实际中一般和其子类打交道。`SKNode` 定义了一些基础属性和方法，如 `position`, `frame`, `alpha`, `physicsBody`, `addChild()`, `runAction()` 等。
- `SKAction` 用来实现位移、缩放等效果，调用 `SKNode.runAction()` 将 `Action` 添加到 `Node`，可以自由组合 `Action` 实现复杂效果。

- `SKPhysicsBody` 定义了一个 Node 的物理属性，设置 `node.physicsBody` 后 node 就可以进行物理计算、碰撞检测。`SKPhysicsBody` 包含了质量、速度、弹性、摩擦力等属性。
- `SKScene` 是 `root node`，定义了 `SKView` 显示的具体内容，`physicsWorld` 属性可设置重力等全局属性，通过 `SKPhysicsContactDelegate` 获得碰撞通知。就游戏来说，其内容是动态变化的，所以 `SKScene` 有一个 rendering loop (见下图)

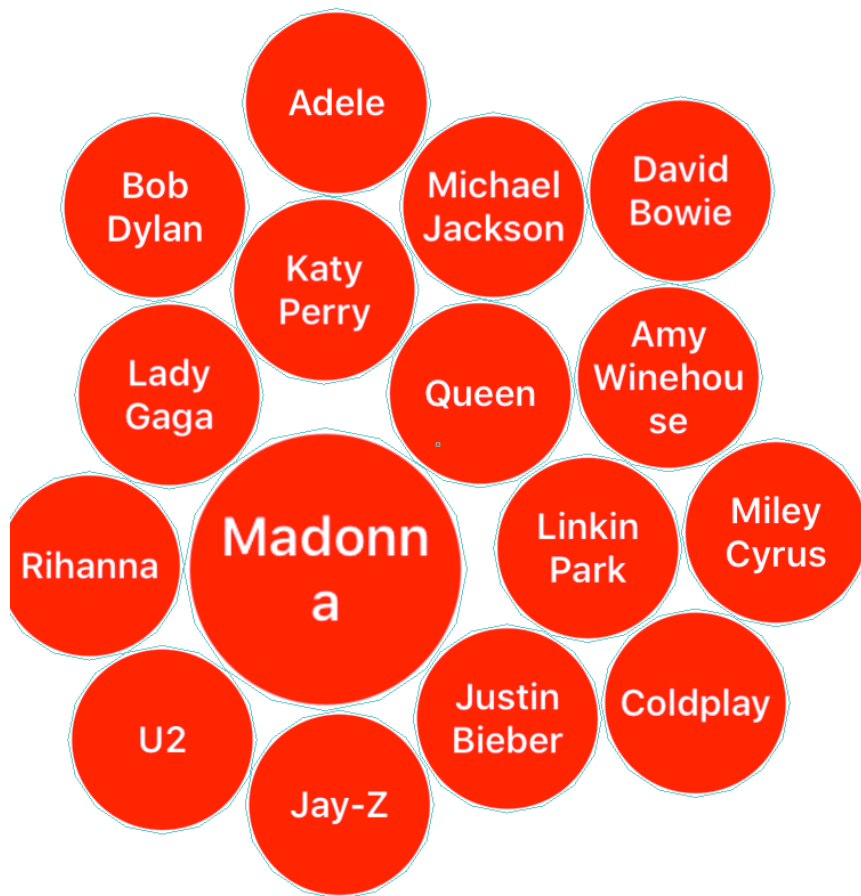
Figure 1 Frame processing in a scene



每一帧都会执行这个 loop，loop 执行完之后被改变的内容才会重新绘制。子类化 `SKScene` 时可以重写 `update(_:)` 和 `didXXX` 方法获得回调。

## 结合 Magnetic 进行代码讲解

上面介绍了 SpriteKit 的一些基本概念，下面我们通过一个示例来介绍如何在普通 app 中结合 SpriteKit 实现一些优雅的交互。此示例受 Github 上的 Magnetic [Magnetic](https://github.com/iblacksun/Magnetic) 项目启发，用 SpriteKit 实现 iTunes Music 的「个人喜好定制」功能，源码见 <https://github.com/iblacksun/Artists>，最后效果如下：



nodes:34 40.7 fps

开始实现

第一步，使用 Xcode 新建一个 Game 类型的项目，Game Technology 选 SpriteKit。简单起见，修改 Deployment Info，使其只支持 iPhone 和 Portrait 方向。Build & Run 之后你会看到我们熟悉的 Hello World。



Hello World 项目中很多模板代码在我们项目中并不需要，删除 GameScene.sks, Actions.sks, GameScene.swift 几个文件；修改 Main.storyboard 将 SKView 的 backgroundColor 修改成白色；修改 GameViewController 替换成如下：

```

class GameViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        guard let skView = self.view as? SKView else{
            return
        }
        skView.ignoresSiblingOrder = true
        skView.showsFPS = true
        skView.showsNodeCount = true
    }
}

```

此时如果运行项目的话应该是一个灰色空白界面，底部显示了 nodes 数量和 fps。

## 构建 Node

现在该主角们登场了，新建 `ArtistsScene.swift` `ArtistNode.swift`，分别继承 `SKScene` 和 `SKShapeNode`。修改 `GameViewController` 的 `viewDidLoad` 方法，在末尾加入代码：

```

let scene = ArtistsScene(size: skView.bounds.size)
skView.presentScene(scene)

```

调用 `presentScene(_:)` 方法呈现 Scene，此时 `ArtistsScene` 就是 root node，因 `SKScene` 的默认背景色是黑色，所以现在运行项目的话会看到一个黑色空白界面。

接下来开始实现功能，重写 `ArtistsScene` 的 `didMove(to:)` 的方法，此方法在 Scene 被 present 时会被调用：



```

override func didMove(to view: SKView) {
    super.didMove(to: view)
    //1.禁用重力
    physicsWorld.gravity = CGVector.zero
    backgroundColor = .white

    let viewWidth = view.frame.size.width
    let viewHeight = view.frame.size.height
    let radius = max(viewWidth, viewHeight)

    //2.添加一个具有向心力的特殊 SKFieldNode.
    let fieldNode = SKFieldNode.radialGravityField()
    fieldNode.region = SKRegion(radius: Float(radius))
    fieldNode.minimumRadius = Float(radius)
    fieldNode.strength = 50
    addChild(fieldNode)

    //3.修改坐标原点
    anchorPoint = CGPoint(x: 0.5, y: 0.5)

    //4. 添加所有 Artist nodes, 初始随机分配在左右两侧, 受向心力作用, 会自动汇聚到
    中心点
    for (index, artistName) in artists.enumerated() {
        let x = (index % 2 == 0) ? -viewWidth/2 : viewWidth/2
        let y = CGFloat.random(-viewHeight/2, viewHeight/2)

        let node = ArtistNode(circleOfRadius: 40)
        node.fillColor = .red
        node.position = CGPoint(x: x, y: y)
        addChild(node)
    }
}

```

1. `physicsWorld.gravity = CGVector.zero` 禁用重力作用, 否则所有设置过 `physicBody` 的 `node` 都会受重力影响自动坠落;
2. 添加一个具有向心力的特殊 `SKFieldNode`, 在其 `region` 内的所有 `node` 都会受影响, 自动向中心移动;
3. SpriteKit 的坐标原点在左下角, 这点和 UIKit 不一样。设置 `anchorPoint = (0.5, 0.5)`, 方便计算后面 `ArtistNode` 的 `position`;
4. 循环添加 `ArtistNode`, 设置其 `position`, 其中 `x` 平均分配到左右两侧, `y` 则取顶部和底部间的随机值。  
此时如果运行的话会发现所有 `ArtistNode` 都停留在屏幕两侧并不会想中心靠拢, 猜猜原因?  
打开 `ArtistNode`, 实现一个 `convenience init` 方法, 传入 `artistName`  
`convenience init(artistName :String) {`

```

self.init()
self.init(circleOfRadius: 40)
fillColor = .red

physicsBody = SKPhysicsBody(circleOfRadius: frame.size.width / 2)
physicsBody?.allowsRotation = false
physicsBody?.friction = 0
physicsBody?.linearDamping = 3

addMultilineTextNode(artistName, radius: 40)

```

```

}

```

在 `init` 中创建一个半径为 40 的圆形 node，填充色是红色；创建一个大小和自身相等的圆形 physicsBody。

接下来的问题是如何将 `artistName` 添加到 ArtistNode? SpriteKit 提供了 `SKLabelNode`，但它不支持文字换行。`addMultilineTextNode(artistName, radius: 40)` 里的代码是将文字转换成图片，然后往 ArtistNode 添加一个 SKSpriteNode。

```

if let image = UIGraphicsGetImageFromCurrentImageContext(){

```

```

    let texture = SKTexture(image: image)
    let spriteNode = SKSpriteNode(texture: texture)
    addChild(spriteNode)

```

```

}

```

```

UIGraphicsEndImageContext()

```

## 添加交互

完成上面代码并运行之后，所有 ArtistNode 会自动从左右两侧缓慢的向中心移动，但并不能拖拽和点击。

SKNode 继承自 UIResponder (NSResponder)，其事件处理和 UIView 一样：通过 `touchesBegan` `touchesMoved` `touchesEnded` `touchesCancelled` 几个方法处理。

因我们的 root node 是 `ArtistsScene`，所以我们可以重写 `ArtistsScene` 的这几个方法来处理拖拽和点击事件。

```

override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let touch = touches.first else {
        return
    }
    let previous = touch.previousLocation(in: self)
    let location = touch.location(in: self)
    if location.distance(from: previous) == 0{
        return
    }
    isMoving = true
    let x = location.x - previous.x
    let y = location.y - previous.y
    for node in children{
        let distance = node.position.distance(from: location)
        let acceleration: CGFloat = 3 * pow(distance, 1/2)
        let direction = CGVector(dx: x * acceleration, dy: y *
acceleration)
        node.physicsBody?.applyForce(direction)
    }
}

override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let touch = touches.first, !isMoving else {
        isMoving = false
        return
    }
    isMoving = false
    let location = touch.location(in: self)
    guard let artistNode = artistNodeAt(location) else{
        return
    }
    artistNode.isSelected = !artistNode.isSelected
}

override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?)
{
    isMoving = false
}

```

1. `touchesMoved` 方法中处理拖拽事件，如果发生拖拽，则向所有 node 施加一个作用力，使之随着手指位置一起移动；
2. `touchesEnded` 中处理点击事件。通过 `isMoving` 区分这个事件是拖拽还是点击，点击的话找出被点击的 `ArtistNode`，设置其 `isSelected` 属性。  
接下来看看 `ArtistNode.isSelected` 的实现：  

```
var isSelected = false{
```

```
didSet{
    guard oldValue != isSelected else {
        return
    }
    removeAction(forKey: "scale")
    let scaleAction = SKAction.scale(to: (isSelected ? 1.5 : 1.0),
    duration: 0.2)
    run(scaleAction, withKey: "scale")
}
```

}

`didSet` 获得设值之后的回调，通过 `run(_ action:)`，选中缩放至 1.5，取消选择还原至 1.0。

## 参考链接

---

- <https://developer.apple.com/spritekit/>
- [SpriteKit Tutorial: Create an Interactive Children's Book with SpriteKit and Swift 3](#)
- [2D Apple Games by Tutorials](#)