WLNetworking网络库

`WLNetworking`是基于`AFNetworking3.0以上`为基础封装的网络请求库，主要采用分布式的API设计。将网络请求与具体第三方库依赖隔离，方便以后更换底层的网络库。该网络库主要参考以下网络库：

```
https://github.com/casatwy/RTNetworking
https://github.com/yuantiku/YTKNetwork
```

文件说明

`Configurations`相关的文件说明如下：

```
1、WLNetworkingConfiguration：网络库的全局定义信息。
2、WLNetwokingConfig：全局的网络库配置信息。包括：全局的请求参数追加，统一的请求返回数据加工协议，运行环境相关信息等。
3、WLNetworkPrivate：网络库的一些操作方法类。
4、WLNetWorkingProcessFilter：网络库的数据统一处理类,对网络请求的返回信息统一处理。遵循WLNetwokingProcessDelegate协议。
```

`WLService`服务器相关文件说明如下：

```
WLServiceInfo：服务器信息的基本抽象类，包换服务器的基本信息，和自定义服务器需要遵循的协议。
WLServiceFactory：服务器信息工厂类，通过它获取访问不同的服务器。
WLNormalService：自定义的服务器信息类，可以添加类似的其它服务器。
```

`WLRequest`相关类文件说明如下：

```
1、WLRequest：一个请求的内容定义类。自定义API请求必须继承该类，并遵循WLRequestAPIDelegate的协议。
2、WLRequest+WLAdd：WLRequest的扩展类，为请求添加Cache缓存功能。
3、WLChainRequest：完成一组存在依赖关系的请求，所有请求完成，请求成功。如请求A请求成功后，通过返回结果，选择是进行B请求还是C请求，所有请求成功，请求结果才算成功。
4、WLBatchRequest：用于处理一组网络请求，统一返回调用结果,其中一个请求失败，则全部
```

失败。

`Manager`相关类文件说明如下：

```
1、WLNetworkManager：Http网络请求的基础类，封装了AFNetworking的网络请求。
2、WLRequestManager：单个WLRequest请求的管理类，用来管理Request发起请求、取消某个请求等操作。
3、WLChainRequestManager：用来管理WLChainRequest的请求信息。
4、WLBatchRequestManager：用来管理WLBatchRequest的请求信息。
```

其它类说明如下：

```
1、WLNetworkingCacheTool：对Cache缓存的相关操作的封装类，目前底层使用的是YYCache的库。
2、WLLogger：打印请求的内容信息，包括请求、请求返回的内容、读取缓存的内容
3、OSSUploadFile：阿里云服务器文件上传的类，采用单例封装。支持单张、多张图片上传和分片文件上传。
4、CustomAPI文件夹：举例的API使用的Demo类。
```

常用的API使用方式

1、服务器信息配置，用来配置服务器的环境，如下所示：

```
1、配置WLNormalService的服务器地址：
/// 测试环境的地址
- (NSString *)debugServiceBaseUrl {
    return [[WLNetwokingConfig sharedInstance] serviceUrlWithType:WLServiceUrlTypeSevnew];
}

// 发布环境的地址
- (NSString *)releaseServiceBaseUrl {
    return [[WLNetwokingConfig sharedInstance] serviceUrlWithType:WLServiceUrlType117_8080];;
}


2、配置WLNetworkingConfiguration中服务器使用的是正式环境还是测试环境，如下所示：

/// 服务器环境,是否正式环境。NO：测试环境    YES：正式环境
static BOOL kWLServiceIsRelease = NO;
```

2、单个API的声明，必须继承`WLRequest`类，并实现`WLRequestAPIDelegate`协议来构造网络请求。例如下面的`WLLoginApi`类声明。
.h文件如下：

```
@interface WLLoginApi : WLRequest<WLRequestAPIDelegate>

/**
 *  @author liuwu     , 16-08-02
 *
 *  获取loginApi实体类
 *  @param params 请求参数
 *  @return 实体类
 *  @since V2.8.3
 */
- (instancetype)initWithParams:(NSDictionary *)params;

@end
```

.m文件的实现如下：

```
@interface WLLoginApi ()

@property (nonatomic, copy) NSDictionary *params;

@end

@implementation WLLoginApi

///获取loginApi实体类
- (instancetype)initWithParams:(NSDictionary *)params {
    self = [super init];
    if (self) {
        self.params = params;
    }
    return self;
}

#pragma mark -  WLRequestAPIDelegate
/// 请求方式，包括Get、Post、Head、Put、Delete、Patch，具体查看 WLRequestMeth
odType
- (WLRequestMethodType)requestMethodType {
    return WLRequestMethodTypePost;
}

/// 接口地址，每个接口都有自己的接口地址
- (NSString *)apiMethodName {
    return @"register/logon";
}
```

```
/// 请求的参数内容
- (NSDictionary *)requestParams {
    NSMutableDictionary *paramsMutableDic = [NSMutableDictionary dictio
naryWithDictionary:_params];
    [paramsMutableDic setObject:kAppVersion forKey:@"version"];
    [paramsMutableDic setObject:kPlatformType forKey:@"platform"];

    [paramsMutableDic setValue:kDeviceUdid forKey:@"device_id"];
    [paramsMutableDic setValue:kDeviceIOSVersion forKey:@"osver"];
    [paramsMutableDic setValue:kDeviceModelInfo forKey:@"model"];

    return [NSDictionary dictionaryWithDictionary:paramsMutableDic];
}

/// 是否需要缓存
//- (BOOL)shouldCache {
//    return YES;
//}

///// 缓存有效时间，如果过期，需要重新请求更新缓存。默认-1:需要更新缓存
//- (NSInteger)cacheTimeInSeconds {
//    return 60;
//}

/// 是否忽略统一的Response加工
//- (BOOL)ignoreUnifiedResponseProcess {
//    return YES;
//}
@end
```

API类定义完成后，API的使用方式如下：

```
[WLHUDView showHUDWithStr:@"登录中..." dim:YES];
WLLoginApi *loginApi = [[WLLoginApi alloc] initWithParams:reqstDic];
[loginApi startWithCompletionBlockWithSuccess:^(WLRequest *request) {
   //登录成功后，做的一些操作
} failure:^(WLRequest *request) {
   //登录失败做的操作
   if (request.error.code == WLNetWorkingSystemErrorTypeAlert) {

   }
   if (request.errorType == WLErrorTypeFailed || request.errorType == W
LErrorTypeNoNetWork) {
            [WLHUDView showErrorHUD:@"登录失败，请重试！"];
   }
}];
```

3、统一的网络数据处理，必须定义一个类，并实现`WLNetwokingProcessDelegate`协议。在程序启动的时候，需要实例化统一处理类，并把它设置给网络管理类。如下定义了`WLNetWorkingProcessFilter`的处理类。在`AppDelegate`中，加入以下代码：

```
//这样只要未设置ignoreUnifiedResponseProcess的API都会通过统一的数据处理
WLNetWorkingProcessFilter *filter = [[WLNetWorkingProcessFilter alloc]
init];
[WLNetwokingConfig sharedInstance].processRule = filter;
```

4、设置统一处理类的情况下，如何在单个API中，不使用统一处理，进行特殊的请求结果处理。设置方式如下：

```
1、如中WLLoginAPI中，实现以下的方法，接口返回数据将不会统一处理：
/**
 *  是否忽略统一的参数加工,默认为NO
 *
 *  @return 返回 YES，那么 self.responseJSONObject 将返回原始的数据
 */
- (BOOL)ignoreUnifiedResponseProcess {
    return YES;
}


2、在当前API中，进行特殊处理，通过实现以下方法：

/**
 *  处理responseJSONObject，当外部访问 self.responseJSONObject 的时候就会返
回这个方法处理后的数据
 *
 *  @param responseObject 输入的 responseObject ，在方法内切勿使用 self.res
ponseJSONObject
 *
 *  @return 处理后的responseJSONObject
 */
- (id)responseProcess:(id)responseObject {

}
```

5、使用`WLChainRequest`的方式请求使用，如下所示：

```
///加载chainRequest方式
- (void)loadChainRequestWithMobile:(NSString *)mobile
                          password:(NSString *)password{
    WLLoginApi *loginApi1 = [[WLLoginApi alloc] initWithMobile:mobile p
assword:password];
```

```
    WLChainRequest *chainReq = [[WLChainRequest alloc] init];
    [chainReq addRequest:loginApi1 callback:^(WLChainRequest *chainRequest, WLRequest *request) {
        WLLoginApi *loginApi1 = (WLLoginApi *)request;
        NSLog(@"requests------>:%@",loginApi1.responseJSONObject);
        WLLoginApi *loginApi4 = [[WLLoginApi alloc] initWithMobile:mobile password:@"1212"];
        [chainRequest addRequest:loginApi4 callback:nil];

    }];
    chainReq.delegate = self;
    // start to send request
    [chainReq start];
}

///请求完成
- (void)chainRequestFinished:(WLChainRequest *)chainRequest {
    // all requests are done

}

///请求失败
- (void)chainRequestFailed:(WLChainRequest *)chainRequest
         failedBaseRequest:(WLRequest *)request {
    // some one of request is failed

}
```

6、使用`WLBatchRequest`的方式发起一组请求的使用，如下所示：

```
///加载Batch请求的方式
- (void)loadBatchDataWithMobile:(NSString *)mobile
                       password:(NSString *)password {
    WLLoginApi *loginApi1 = [[WLLoginApi alloc] initWithMobile:mobile password:@"1212"];
    WLLoginApi *loginApi2 = [[WLLoginApi alloc] initWithMobile:mobile password:password];
    WLLoginApi *loginApi3 = [[WLLoginApi alloc] initWithMobile:mobile password:password];
    WLLoginApi *loginApi4 = [[WLLoginApi alloc] initWithMobile:mobile password:password];
    WLBatchRequest *batchRequest = [[WLBatchRequest alloc] initWithRequestArray:@[loginApi1,loginApi2,loginApi3,loginApi4]];
    [batchRequest startWithCompletionBlockWithSuccess:^(WLBatchRequest *batchRequest) {
        NSArray *requests = batchRequest.requestArray;
        WLLoginApi *loginApi1 = (WLLoginApi *)requests[0];
        WLLoginApi *loginApi2 = (WLLoginApi *)requests[1];
        WLLoginApi *loginApi3 = (WLLoginApi *)requests[2];
        WLLoginApi *loginApi4 = (WLLoginApi *)requests[3];
```

```
        NSLog(@"requests------>::%lu",(unsigned long)loginApi1.requestDataTask.hash);
        NSLog(@"requests------>::%lu",(unsigned long)loginApi2.requestDataTask.hash);
        NSLog(@"requests------>::%lu",(unsigned long)loginApi3.requestDataTask.hash);
        NSLog(@"requests------>::%lu",(unsigned long)loginApi4.requestDataTask.hash);

        ///登录成功进入主页面
        [self.loginView checkToMainView:loginApi1.responseJSONObject];
    } failure:^(WLBatchRequest *batchRequest) {
        NSLog(@"failed");
    }];
}
```

Welian中的统一处理类使用说明

`Welian`中接口返回数据统一处理类`WLNetWorkingProcessFilter`，遵循`WLNetwokingProcessDelegate`协议。

1、配置类接口的统一参数，所有接口统一需要添加以下参数：

```
- (NSDictionary *)urlArgumentsInfos {
    /// 每次启动App时都会新生成
    NSString *sessionId = [NSUserDefaults objectForKey:kWL_UserSessionIdKey];
    if (sessionId.length > 0) {
        return @{@"sessionid" : sessionId, @"version" : kAppVersion};
    }else {
        return @{@"version" : kAppVersion};
    }
}
```

2、统一的加工，用来返回数据返回是否正确，如下所示：

```
- (BOOL)processResponseValidator:(id)response {
    BOOL success = YES;
    ///接口返回的错误
    if ([response isKindOfClass:[NSError class]]) {
        success = NO;
    }else{
        if ([response isKindOfClass:[NSString class]]) {
            success = YES;
        }else{
```

```
            id request = [[response wl_decryptAES256Value] jsonValueDec
oded];
            if ([request isKindOfClass:[NSDictionary class]]) {
                NSDictionary *resultInfo = request;
                NSNumber *state = resultInfo[@"state"];

                //1100 1200 1201 聊天室相关，已废弃，没有用，当前普通错误处理
                //很多地方校验参数失败都返回的是1020,当作普通错误处理
                if (state.integerValue == 1000) {
                    /// 接口返回成功，且数据正确
                    success = YES;
                }else{
                    success = NO;
                }
            }
        }
    }
    return success;
}
```

3、统一的返回数据处理，首先对返回的数据进行统一的解密。如果`State等于1000`的表示接口返回数据成功，返回`data`中的数据给Request。如果错误，对应的错误码定义不同的错误类型。错误类型如下：

```
///网络请求的方式
typedef NS_ENUM (NSUInteger, WLErrorType){
    WLErrorTypeSuccess = 0,     //API请求成功且返回数据正确，此时的数据是可以
直接拿来用的
    WLErrorTypeNoContent,       //API请求成功但返回的数据不正确。
    WLErrorTypeFailed,          ///网络请求失败，接口级别错误
    WLErrorTypeNoNetWork,       //网络不通。在调用API之前会判断一下当前网络是
否通畅，这个也是在调用API之前验证的，和上面超时的状态是有区别的。
};


/// 网络库系统错误类型，用于网络库错误处理
typedef NS_ENUM(NSUInteger, WLNetWorkingSystemErrorType) {
    /// ---------- 普通的接口信息错误 ----------
    WLNetWorkingSystemErrorTypeNormal = 1001,/// 系统普通的错误，不做任何处
理的
    WLNetWorkingSystemErrorTypeHub = 1101,/// 需要hub提醒的，最常用的错误提
醒
    WLNetWorkingSystemErrorTypeAlert = 1102,/// /// 需要alert提醒的

    /// ------- 服务器内部错误 ------------
    WLNetWorkingSystemErrorTypeServer = 2000,/// 服务器错误，只打印

    /// ------- 程序系统级别的错误 ----------
    WLNetWorkingSystemErrorTypeLogout = 3101,/// 退出登录
```

```
    WLNetWorkingSystemErrorTypeAlertAndJump = 3102,/// 需要alert提醒，并且点击后需要跳转指定页面的
};
```

4、对于错误统一的处理，如下所示：

```
- (void)processResponseWithError:(id)error errorType:(WLErrorType)errorType {
    //隐藏接口外面的加载动画
    [WLHUDView hiddenHud];
    switch (errorType) {
        case WLErrorTypeNoContent: {
            if ([error isKindOfClass:[NSError class]]) {
                NSError *errorInfo = error;
                NSString *errorMsg = @"";
                WLNetWorkingSystemErrorType errorType = errorInfo.code;
                switch (errorType) {
                    case WLNetWorkingSystemErrorTypeNormal: {
                        /// 普通的错误类型，不做任何处理
                        errorMsg = errorInfo.localizedDescription;
                        break;
                    }
                    case WLNetWorkingSystemErrorTypeHub: {
                        /// 进行hub提醒
                        errorMsg = errorInfo.localizedDescription;
                        if (errorMsg.length > 0) {
                            [WLHUDView showErrorHUD:errorMsg];
                        }
                        break;
                    }
                    case WLNetWorkingSystemErrorTypeAlert: {
                        /// 进行Alert提醒
                        errorMsg = errorInfo.localizedDescription;
                        if (errorMsg.length > 0) {
                            [UIAlertView bk_showAlertViewWithTitle:@"错误信息"
                                                           message:errorMsg
                                                 cancelButtonTitle:@"确定"
                                                 otherButtonTitles:nil
                                                           handler:^(UIAlertView *alertView, NSInteger buttonIndex) {

                                                           }];
                        }
                        break;
                    }
                    case WLNetWorkingSystemErrorTypeAlertAndJump: {
                        /// 进行Alert提醒，并可以点击确定跳转
```

```
                        errorMsg = errorInfo.localizedDescription;
                        NSString *url = [[error userInfo] objectForKey:
@"url"];
                        if (errorMsg.length > 0) {
                            [UIAlertView bk_showAlertViewWithTitle:@"错
误信息"
                                                          message:erro
rMsg
                                                cancelButtonTitle:@"确
定"
                                                otherButtonTitles:nil
                                                          handler:^(UI
AlertView *alertView, NSInteger buttonIndex) {
                                                              if (url.
length > 0) {
                                                                  [[UI
Application sharedApplication] openURL:[NSURL URLWithString:url]];
                                                              }
                                                          }];
                        }
                        break;
                    }
                    case WLNetWorkingSystemErrorTypeServer: {
                        /// 服务器错误，不做提醒
                        errorMsg = errorInfo.localizedDescription;
                        DLog(@"System Error -----> %@",errorMsg);
                        break;
                    }
                    case WLNetWorkingSystemErrorTypeLogout: {
                        /// 被踢出，需要重新登录
                        errorMsg = errorInfo.localizedDescription;
                        NSDictionary *info = nil;
                        if (errorMsg.length > 0) {
                            info = @{@"msg":errorMsg};
                        }
                        [[NSNotificationCenter defaultCenter] postNotif
icationName:kWLLogoutNotification object:nil userInfo:info];
                        break;
                    }
                }
            }
            break;
        }
        case WLErrorTypeSuccess: {
            ///接口成功，不做处理
            break;
        }
        case WLErrorTypeFailed: {
            ///接口调用失败，交给接口调用的地方来决定是否弹出订制性的错误提醒
            /*
            code -1001:请求超时，请检查网络!
             code -1004:未能连接到服务器，请检查网络!
```

```
             */
            break;
        }
        case WLErrorTypeNoNetWork: {
            /// 无网络连接，交给接口调用的地方来决定是否弹出订制性的错误提醒
            break;
        }
    }
}
```

5、对于网络状态发生变化，当前网络状态。在`WLNetwokingConfig`中进行监听，当网络从无网络到有网络的时候，会像页面发送一个通知。如下所示：

```
///监控网络状态
- (void)checkReachableStatus {
    _isNetWorkConnect = YES;//默认链接状态
    [[AFNetworkReachabilityManager sharedManager] setReachabilityStatus
ChangeBlock:^(AFNetworkReachabilityStatus status) {
        switch (status) {
            case AFNetworkReachabilityStatusUnknown: {
                NSLog(@"Reachability: 未知网络");
                if (!_isNetWorkConnect) {
                    _isNetWorkConnect = YES;
                    [[NSNotificationCenter defaultCenter] postNotificat
ionName:kWLNetworkReContentNotification object:nil];
                }
                break;
            }
            case AFNetworkReachabilityStatusNotReachable: {
                NSLog(@"Reachability: 无网络");
                if (_isNetWorkConnect) {
                    _isNetWorkConnect = NO;
                    //全局通知，没有网络
                    NSDictionary *params = @{@"msg" : kNetworkNoContent
};
                    [[NSNotificationCenter defaultCenter] postNotificat
ionName:kWLNetworkNoContentNotification object:nil userInfo:params];
                }
                break;
            }
            case AFNetworkReachabilityStatusReachableViaWWAN: {
                NSLog(@"Reachability: 手机自带网络");
                if (!_isNetWorkConnect) {
                    _isNetWorkConnect = YES;
                    [[NSNotificationCenter defaultCenter] postNotificat
ionName:kWLNetworkReContentNotification object:nil];
                }
                break;
            }
```

```
            case AFNetworkReachabilityStatusReachableViaWiFi: {
                NSLog(@"Reachability: WIFI");
                if (!_isNetWorkConnect) {
                    _isNetWorkConnect = YES;
                    [[NSNotificationCenter defaultCenter] postNotificat
ionName:kWLNetworkReContentNotification object:nil];
                }
                break;
            }
        }
    }];
}
```