贪吃蛇Storekit 2 改造

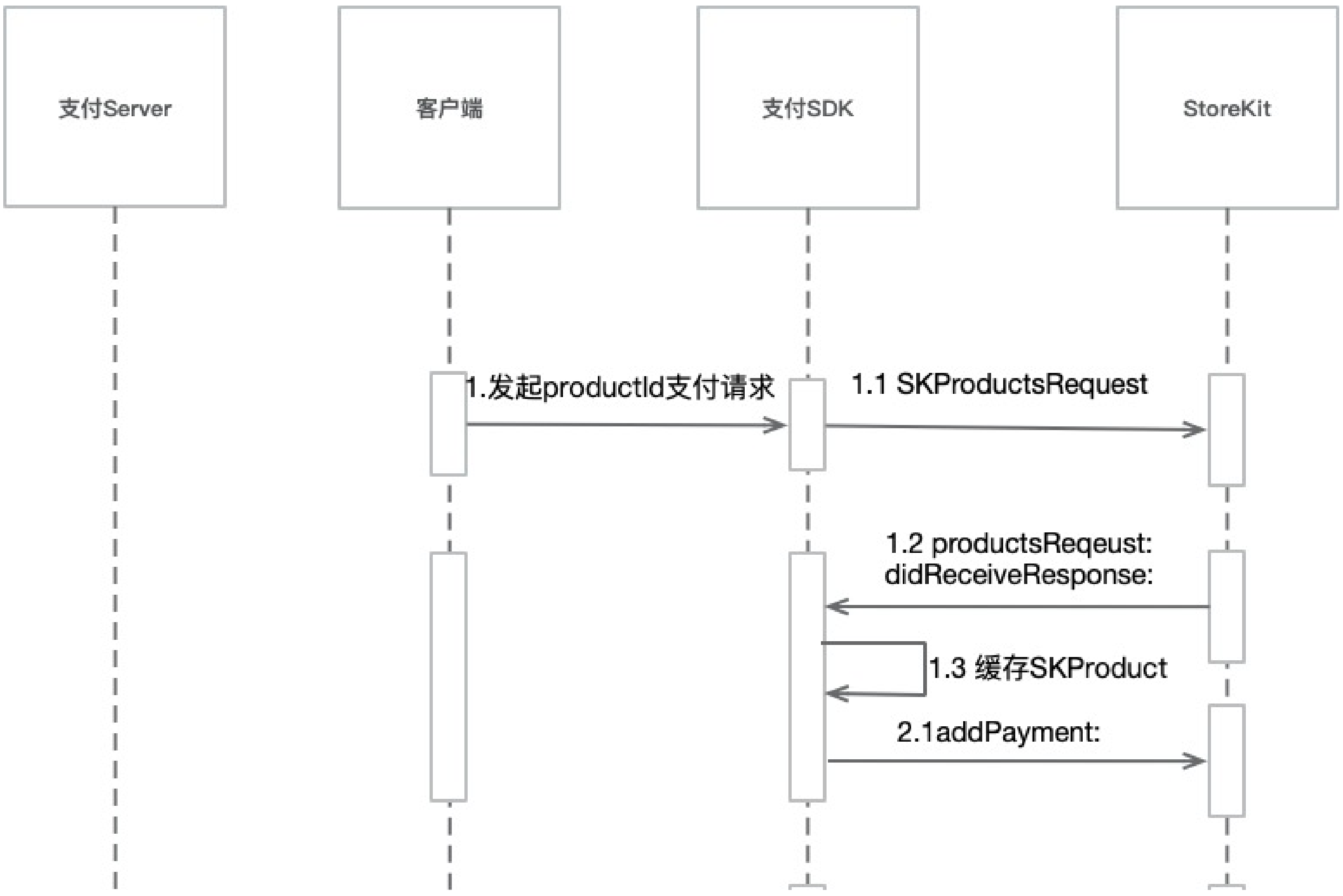
一，Storekit 2 介绍

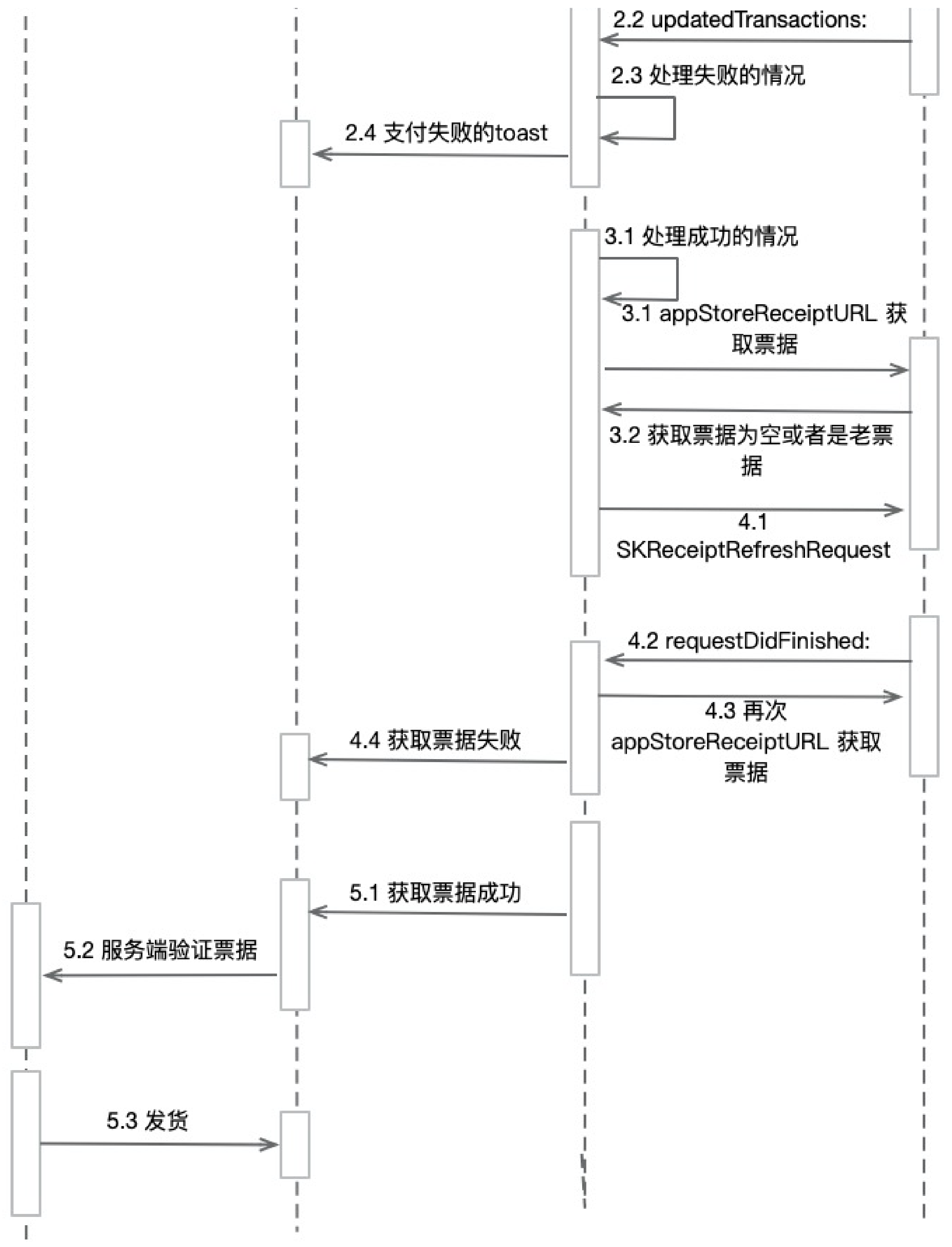
- 支持iOS 15+，swift 5.5 新特性开发
- 可以在苹果交易里面绑定UUID，可以跟自己游戏内订单号映射，便于票据验证的时候，直接通过UUID查找成功订单，发放权益
- 通过productID 返回的 Product 里面能区分消耗品，非消耗品，订阅商品，非连续订阅商品
- 针对于自动连续订阅的第一次购买优惠，isEligibleForIntroOffer 能获取用户当前的 Apple ID 下的是否第一次购买
- 支持 App 内发起退款

二，支付流程改动

1，StoreKit 的流程图

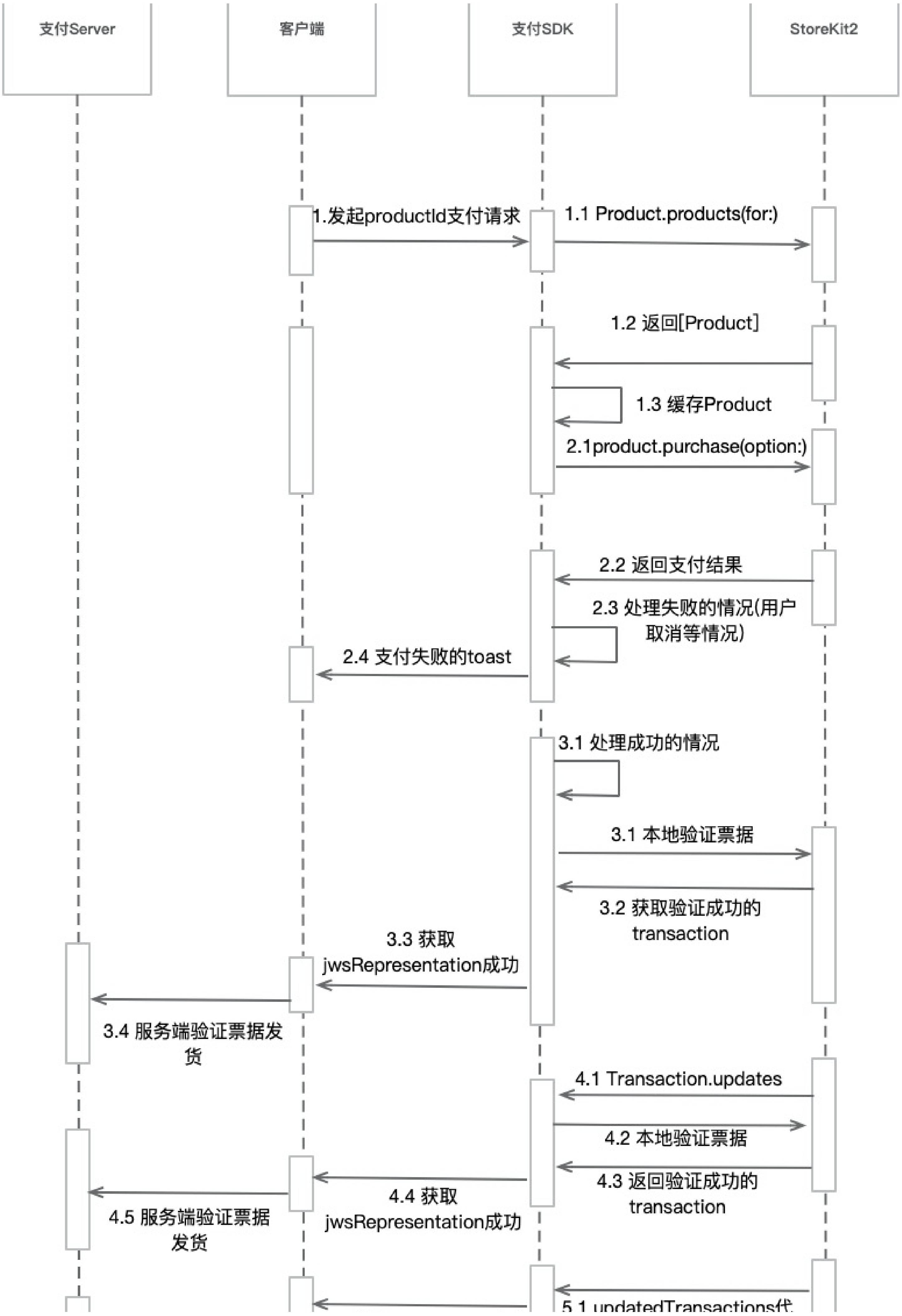
- 需要启动的时候，将支付SDK加入到TransactionObserver, [[SKPaymentQueue defaultQueue] addTransactionObserver:支付SDK];





1，StoreKit2 的流程图

- 需要启动的时候，将支付SDK加入到TransactionObserver, [[SKPaymentQueue defaultQueue] addTransactionObserver:支付SDK];
- 需要添加 Transaction.updates 的task，监听异常支付订单


支付Server
客户端
支付SDK
StoreKit2
1.发起productId支付请求
1.1 Product.products(for:)
1.2 返回[Product]
1.3 缓存Product
2.1product.purchase(option:)
2.2 返回支付结果
2.3 处理失败的情况(用户取消等情况)
2.4 支付失败的toast
3.1 处理成功的情况
3.1 本地验证票据
3.2 获取验证成功的transaction
3.3 获取jwsRepresentation成功
3.4 服务端验证票据发货
4.1 Transaction.updates
4.2 本地验证票据
4.3 返回验证成功的transaction
4.4 获取jwsRepresentation成功
4.5 服务端验证票据发货
5.1 updatedTransactions代

三，接口改动

1，获取商品的方式

1.1 老的处理方式

- 先发SKProductsRequest请求，然后在delegate中处理结果

```
// 1. 请求商品
  SKProductsRequest *request = [[SKProductsRequest alloc]
initWithProductIdentifiers:productIdentifiers];
  request.delegate = self;
  [request start];

// 2. 实现 SKProductsRequestDelegate
- (void)productsRequest:(SKProductsRequest *)request didReceiveResponse:
(SKProductsResponse *)response
{
    // success
    NSArray<SKProduct *> *products = response.products;
}

- (void)request:(SKRequest *)request didFailWithError:(NSError *)error
API_AVAILABLE(ios(3.0), macos(10.7))
{
    // failed
    WPIAPlog(@"error=%@",error);
}
```

1.2 新的处理方式

- 采用 swift 的异步编程特性，async 申明一个异步的方法，返回查询结果
- do-catch 处理异常等失败

```swift
public func loadProducts(withIdentifiers ids: Set<String>!) async -> ([Product],Error) {
    do {
        let skProducts = try await Product.products(for: ids)
        return (skProducts,nil)
    } catch (let err) {
        return ([Product](),err)
    }
}
```

2，商品购买

2.1 老的处理方式

- 通过 SKProduct 生成 SKPayment 对象，添加到 SKPaymentQueue 中，这样会调起苹果的支付弹框，详见代码示例1
- 监听 SKPaymentTransactionObserver 的 updatedTransactions 代理方法，判断 transactionState 处理成功和失败的情况，详见代码示例2
- 如果 transactionState 是失败，就直接把 error 回调到业务层，展示失败 toast，详见代码示例3
- 如果 transactionState 是成功，就尝试获取本地票据，可能会获取不到，会有5次重复获取逻辑，详见代码示例4
- 如果5次重试获取票据还是失败，会发起拉票据的请求，详见代码示例5
- 拉票据请求回调里，继续尝试拿获取本地票据，如果还是拿不到，就抛错误到业务层，展示失败 toast，详见代码示例6
- 如果成功获取到本地支付票据，就将票据回调到业务层，详见代码示例7

```objc
// 示例 1.发起内购
- (void)buyProduct:(SKProduct *)product paymentStateBlock:(WPIAPPaymentStateBlock)paymentStateBlock {
    WPIAPlog(@"iap---开始购买-->%@", product);
    SKPayment *payment = [SKPayment paymentWithProduct:product];
    [[SKPaymentQueue defaultQueue] addPayment:payment];
}

// 示例 2.监听内购结果回调
#pragma mark - SKPaymentTransactionObserver
- (void)paymentQueue:(SKPaymentQueue *)queue updatedTransactions:(NSArray *)transactions {
    for (SKPaymentTransaction *transaction in transactions) {
        WPIAPlog(@"iap---苹果处理流程-->description =%@,transactionState = %ld", transaction.error.localizedDescription,(long)transaction.transactionState);
        switch (transaction.transactionState) {
            case SKPaymentTransactionStatePurchasing:
                WPIAPlog(@"iap---正在付款");
                break;
            case SKPaymentTransactionStateDeferred:
                WPIAPlog(@"iap---正在延迟");
                break;
            case SKPaymentTransactionStatePurchased:
                if (transaction.originalTransaction) {
                    WPIAPlog(@"iap---续订完成");
                } else {
                    WPIAPlog(@"iap---付款完成");
                }
                [self paySuccessFinishTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed:
                // 示例 3.失败的处理
                WPIAPlog(@"iap---付款失败-->%@", transaction.error.localizedDescription);
                [[SKPaymentQueue defaultQueue] finishTransaction:transaction];
                !_buyFailBlock ?: _buyFailBlock(transaction.error);
                break;
            case SKPaymentTransactionStateRestored: // 消耗型和非续期订阅不支持恢复，只有非消耗型(例如游戏地图)和自动订阅型支持恢复(连续包月)
                WPIAPlog(@"iap---付款已恢复");
                [[SKPaymentQueue defaultQueue] finishTransaction:transaction]; // 无论恢复成功与否，都finish掉
                break;
            default:
                break;
        }
    }
}

// 示例 4.开始获取本地支付成功票据
- (void)paySuccessFinishTransaction:(SKPaymentTransaction *)transaction {
    [self getAppStoreReceiptRetryTimes:5 isFirst:YES completeBlock:^(NSString *payReceiptStr, NSError *error) {
        if (!payReceiptStr || error) {
            [self refreshReceiptRequest:^(NSError *error) {
                [self getAppStoreReceiptRetryTimes:5 isFirst:YES completeBlock:^(NSString *payReceiptStr, NSError *error) {
                    !completeBlock? :completeBlock(payReceiptStr,error);
                    if (error || !payReceiptStr) {
                        // 示例 6.获取本地票据失败，回调 error 到业务层
                        !_buyFailBlock ?: _buyFailBlock(error);
                    } else {
                        [self getReceiptSuccessWithReceipt:payReceiptStr];
                    }
                }];
            }];
        }else{
            [self getReceiptSuccessWithReceipt:payReceiptStr];
        });
        }
    }];
}

// 示例 7.成功获取到票据，先本地保存，然后回调到业务层
-  (void)getReceiptSuccessWithReceipt:(NSString *)receipt {
    [UICKeyChainStore setString:payReceiptStr forKey:WPIAPReceiptKey];
    [UICKeyChainStore setString:payReceiptStr forKey:WPLastIAPReceiptKey];
    !_buySuccessBlock? :_buySuccessBlock(payReceiptStr, transaction);
}

// 尝试拿票据，重试 retryTimes 次
- (void)getAppStoreReceiptRetryTimes:(NSInteger)retryTimes isFirst:(BOOL)isFirst completeBlock:(void (^)(NSString *payReceiptStr,NSError *error))completeBlock {
    __block NSInteger blockRetryTimes = retryTimes;
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(isFirst?0.5:1  * NSEC_PER_SEC)), dispatch_get_main_queue(), ^{

        NSString *currrentReceiptString = [self appStoreReceiptString];
        NSString *lastKeyChainReceiptString = [self lastKeyChainReceiptString];

        // 如果新获取的票据跟上次验证成功的票据不一样，就认为内购流程走完，回调此次成功的票据
        if (currrentReceiptString && ![currrentReceiptString isEqualToString:lastKeyChainReceiptString]) {
            !completeBlock? :completeBlock(currrentReceiptString,nil);
        }else{
            if (blockRetryTimes == 0) {
                // 重试次数到了，还没有取到票据，报错"没有获取到凭证"
                if (!currrentReceiptString) {
                    !completeBlock? :completeBlock(currrentReceiptString, [NSError errorWithDomain:@"wepie.snake.com" code:12357 userInfo:@{NSLocalizedDescriptionKey: @"没有获取到凭证"}]);
                }else{
                    // 重试次数到了，还是取到旧的票据，报错"获取到旧的凭证"
                    !completeBlock? :completeBlock(currrentReceiptString, [NSError errorWithDomain:@"wepie.snake.com" code:12358
userInfo:@{NSLocalizedDescriptionKey: @"获取到旧的凭证"}]);
                }
            }else{
                [self getAppStoreReceiptRetryTimes:blockRetryTimes isFirst:NO completeBlock:completeBlock];
            }
        }
    });
}

// 获取本地支付成功票据
- (NSString *)appStoreReceiptString {
    NSURL *url = [[NSBundle mainBundle] appStoreReceiptURL];
    NSData *data = [NSData dataWithContentsOfURL:url];
    NSString *receiptString = [data base64EncodedStringWithOptions:0];
    return receiptString;
}

// 获取本地存贮的上次服务器验证成功的票据
- (NSString *)lastKeyChainReceiptString {
    NSString *receiptString = [UICKeyChainStore stringForKey:WPLastIAPReceiptKey];
    return receiptString;
}

// 示例 5.发起拉票据的请求
- (void)refreshReceiptRequest:(void (^)(NSError *error))refreshReceiptBlock {
    _refreshReceiptBlock = refreshReceiptBlock;
    SKReceiptRefreshRequest *receiptrequest = [[SKReceiptRefreshRequest alloc] init];
    receiptrequest.delegate = self;
    [receiptrequest start];
}

// 刷票据接口调用成功，可以再次拉票据了
#pragma mark - SKRequestDelegate
- (void)requestDidFinish:(SKRequest *)request {
    if ([request isKindOfClass:[SKReceiptRefreshRequest class]]) {
        !_refreshReceiptBlock? :_refreshReceiptBlock(nil);
    }
}
```

2.2 新的处理方式

- 调用 Product 的实例方法 purchase 调起苹果的支付弹框，详见代码示例1

- 通过 purchase 异步返回的结果判断支付状态，详见代码示例2
- 直接拿结果的 jwsRepresentation，这个是类似 StoreKit 的票据，上报给服务器完成校验发货，详见代码示例3
- 启动的时候需要添加内购transaction状态的监听，有一些异常的内购交易会回调到这里，详见代码示例4
- 还会有一部分异常的内购交易会通过paymentQueue:updatedTransactions: 方法回调,详见代码示例5
- 异常情况的订单情况需要上报给业务层，再合适的时候，完成服务器校验发货
- 异常情况的订单可能获取不到jwsRepresentation，新的处理服务器支持通过transactionID去校验支付情况

```swift
// 发起内购，返回支付错误，票据，transactionId
@MainActor
@objc public func buy(_ product:Product,uuid:String) async ->
(Error?,String?,String?) {
    let u = UUID(uuidString: uuid)
    // Begin a purchase.
    do {
        // 订单绑定 UUID，跟 orderID 对应
        let option1 = Product.PurchaseOption.appAccountToken(u ?? UUID())
        // 示例1. 发起支付
        let result = try await product.purchase(options: [option1])

        // 示例2. 验证支付结果
        switch result {
        case .success(let verification):
            let transaction = try _checkVerified(verification)
            return (nil,verification.jwsRepresentation,transaction.originalID)
        case .userCancelled:
            print("【内购流程】swift: userCancelled: \(p.id)")
            let cancelError = WPSwiftIAPError.usercancel
            return (nil,nil,cancelError)
        case .pending:
            print("【内购流程】swift: pending: \(p.id)")
        default:
            print("【内购流程】swift: \(product.id)")
        }
    } catch (let err) {
        return (nil,nil,err)

    }
    return (nil,nil)
}
```

```swift

// 验证票据，返回支付错误，票据，transactionId
@MainActor
private func _verifyReceipt(transaction: Transaction,
                            result: VerificationResult<Transaction>) async -> (Error?,String?,String?) {
    if let skpd = productDic[transaction.productID] {
        // 示例3. 获取jwsRepresentation
        return (nil,result.jwsRepresentation,transaction.originalID)
    } else {
        do {
            let products = try await loadProducts(withIdentifiers: [transaction.productID])
            verifyingIds.remove(transaction.id)
            if let skpd = productDic[transaction.productID] {
                return (nil,result.jwsRepresentation,transaction.originalID)
            }
        } catch (let err) {
            return (err,nil,nil)
        }
    }
    let notFoundError = WPSwiftIAPError.notFound
    return (notFoundError,nil,nil)
}


/// 本地判断是否验证通过
/// - Returns: 验证结果
func _checkVerified<T>(_ result: VerificationResult<T>) throws -> T {
    //Check if the transaction passes StoreKit verification.
    switch result {
    case .unverified:
        //StoreKit has parsed the JWS but failed verification. Don't deliver content to the user.
        throw WPSwiftIAPError.failedVerification
    case .verified(let safe):
        //If the transaction is verified, unwrap and return it.
        return safe
    }
}


// 示例4. 异常处理
// 异常处理
/// 监听购买
/// - Returns: 任务
private func _listenForTransactions() -> Task<Void, Error> {
    return Task.detached { [weak self] in
        //Iterate through any transactions which didn't come from a direct call to `purchase()`.
        for await result in Transaction.updates {
            do {
                if let transaction = try self?._checkVerified(result) {
                    print("【内购流程】swift: Transaction.updates 本地验证收据成功，开始服务器验证: \(transaction.productID) id: \(transaction.id) originalID: \(transaction.originalID)")
                    if let appAccountToken = transaction.appAccountToken {
                        print("【内购流程】swift: Transaction.updates appAccountToken: \(appAccountToken)")
                    }
                    let res = await self?._verifyReceipt(transaction: transaction,
                                                result: result)
                    if let b = self?.delayTransactionComeBlock,res?.0 == nil {
                        DispatchQueue.main.async {
                            b(res?.0, res?.1,res?.2)
                        }
                    }
                }
            } catch {
                print("【内购流程】swift: Transaction.updates 验证收据失败")
            }
        }
    }
}

// 示例5. 异常处理
// 部分订单，还是会通过 updatedTransactions 这个代理方法返回
public func paymentQueue(_ queue: SKPaymentQueue, updatedTransactions transactions: [SKPaymentTransaction]) {
    for transaction in transactions {
        let name = transaction.transactionIdentifier
        print("【内购流程】swift: 交易状态改变:\(transaction.transactionState), id:\(name ?? ""), isMainThread:\(Thread.isMainThread)")
        if transaction.transactionState == .purchased {
            Task {
                if let result = await Transaction.latest(for: productIdentifier) {
                    var shouldCheck = false
                    let transaction = try _checkVerified(result)
                    if let originalID = originalTransactionId {
                        // 如果有originalTransactionId，那么必须验证originalTransactionId是否与当前的最后一单一致
                        if originalID == String(transaction.originalID) {
```

```swift
                            shouldCheck = true
                        }
                    } else {
                        shouldCheck = true
                    }
                    if shouldCheck {
                        print("【内购流程】swift: _checkPurchased 本地验证收据成功，开始服务器验证: \(transaction.productID) id: \(transaction.id) originalID: \(transaction.originalID)")
                        if let appAccountToken = transaction.appAccountToken {
                            print("【内购流程】swift: _checkPurchased appAccountToken: \(appAccountToken)")
                        }
                        let res = await _verifyReceipt(transaction: transaction, result: result)
                        if let receipt = res.1 {
                            delayTransactionComeBlock?(receipt,originalTransactionId)
                        }
                    } else {
                        print("【内购流程】swift: _checkPurchased 本地没查到对应的购买1: \(productIdentifier)")
                        // 调用服务器接口直接做验证
                        if let originalTransactionId {
                            delayTransactionComeBlock?(nil,originalTransactionId)
                        }
                    }
                } else {
                    print("【内购流程】swift: _checkPurchased 本地没查到对应的购买2: \(productIdentifier)")
                    // 调用服务器接口直接做验证
                    if let originalTransactionId {
                        delayTransactionComeBlock?(nil,originalTransactionId)
                    }
                }
            }
        }
    }
}
```

四，服务端改动

3.1 老的处理方式

- 客户端支付成功后，上传 receipt 到服务器，服务器调用Apple的接口解密 receipt，查看是否有未验证的成功订单和对应商品id，并发放权益

3.2 新的处理方式

分三种处理方式，更加灵活一点

- 客户端校验通过后，上传 jwsRepresentation 到服务器，这个是类似StoreKit1的票据，服务器解开它可以获取transactionId，然后去调用Apple接口验证有效性，然后发放权益，跟老的处理方式类似
- 客户端可以直接上传 transactionId，服务器去调用Apple接口验证有效性，然后发放权益
- 服务器接收服务器的支付成功推送，根据通知的内容，确定订单号，对用户发放权益
- 定期调用交易历史查询接口，为在回调通知漏掉的交易补发权益。

3.3 新增的服务器接口

- 内购历史订单查询 API
- 查找收据发票 API，通过用户提供的发票上的订单号，查询到对应的transaction交易信息，便于补单
- 查询用户历史退款信息

五，总结

- StoreKit 2 提供了更多的服务器端接口，服务端验证订单会更高效
- 有UUID和订单号的绑定，服务器端验证更有依据，但是某些异常订单会出现UUID相同，transactionID 不同的情况
- 一些异常订单，StoreKit 2 更多的是主动通知客户端处理，StoreKit 需要自己发request去刷新票据
- StoreKit 2 客户端接口改动较大，但是基本流程类似，异常处理比较多，需要有队列保存异常订单，找时机恢复