App 内购买项目编程指南

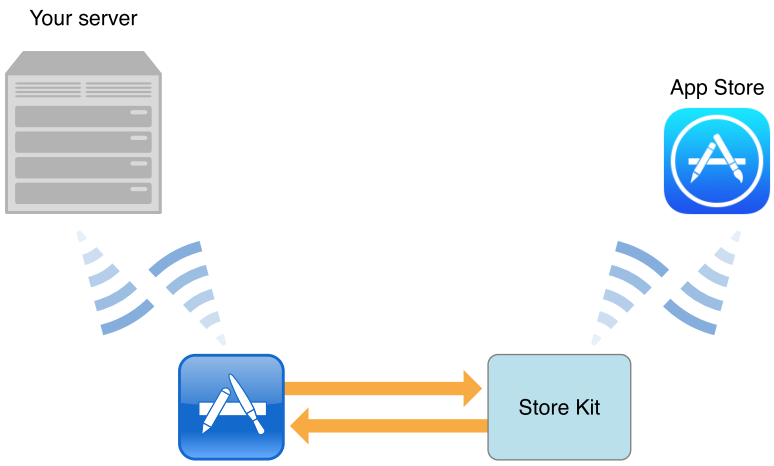
Developer

目录

图形、表格和列表

关于 App 内购买项目

“App 内购买项目”让您可以使用 StoreKit framework（StoreKit 框架）在您的 App 内部嵌入一个商店。此框架代表您的 App 连接到 App Store，以安全地处理来自用户的付款，并提示他们授权付款。该框架随后通知您的 App，让您的 App 向用户提供已购买的项目。使用“App 内购买项目”来针对额外的功能和内容收费。



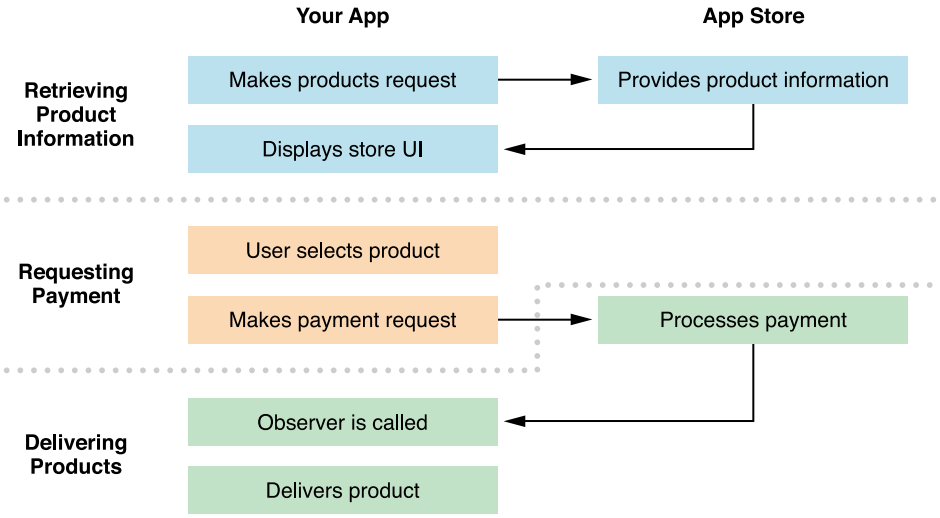
例如，使用”App 内购买项目“，您可以实现以下场景：

- 一个您 App 的基础版本，具有额外的高级功能
- 一个杂志 App，让用户购买并下载新期刊
- 一个游戏，提供新的探索关卡
- 一个在线游戏，允许玩家购买虚拟财产

概况一览

从上层来看，在“App 内购买项目”的处理过程中，用户、您的 App 和 App Store 三者间的交互将分三个阶段进行，如“图 I-1”中所示。首先，用户导航至您 App 中的商店，其中的产品会在 App 中显示；其次，用户选择要购买的产品，您的 App 随即向 App Store 发送付款请求；最后，App Store 处理付款，您的 App 交付已购买产品。

图 I-1　购买过程的各阶段



您在 iTunes Connect 中创建和配置产品

通过理解“App 内购买项目”支持何种产品与行为，您可以设计您的 App 和 App 内商店以充分利用此技术。

相关章节：设计您 App 的产品 （第 9 页）

您的 App 与 App Store 进行交互以销售产品

所有使用“App 内购买项目”的 App 都需要实现这些章节中所描述的核心功能，以便用户进行购买，然后向他们交付已购买产品。

这些开发任务需要按顺序执行。相关章节按照您实现的顺序介绍它们，并在“实施清单 （第 43 页）”中完整列出。为了帮助规划您的开发，您可能想要在开始之前阅读完整的清单。

相关章节：检索产品信息 （第 12 页）、请求付款 （第 19 页）、交付产品 （第 22 页）

订阅需要额外的应用逻辑

提供订阅的App需要跟踪用户何时进行有效订阅、响应过期和续期，并决定用户拥有访问哪些内容的权限。

相关章节：处理订阅 （第 32 页）

用户可以恢复购买

用户可以恢复他们先前购买的产品——例如，将他们已经支付的内容转移到新手机上。

相关章节：恢复已购买产品 （第 39 页）

提交 App 和产品以供审核

当您完成开发和测试后，您可以提交您的 App 和“App 内购买项目”产品以供审核。

相关章节：为App 审核做准备 （第 42 页）

另请参阅

- *《iTunes Connect 的 App 内购买项目配置指南》*描述了如何在 iTunes Connect 中创建和配置您 App 的产品
- “Xcode Help（Xcode 帮助）”中的“Configure capabilities（配置功能）”>“Add a capability（添加功能）”展示了如何为您的 App 启用“App 内购买项目”（您也可以在此启用其他功能）
- *《Receipt Validation Programming Guide（收据验证编程指南）》*描述了如何处理收据，尤其是如何处理购买成功的 App 内购买项目记录

设计您 App 的产品

*产品*是指您想要在您 App 的商店中销售的东西。您在 iTunes Connect 中创建并配置产品，并且您的 App 使用 `SKProduct` 和 `SKProductsRequest` 类与产品进行交互。

了解使用 App 内购买项目可以销售的产品

您可以使用“App 内购买项目”来销售内容、App 功能和服务。

- 内容——交付数字内容或素材，例如杂志、照片和图案。内容也可以供App自身使用，例如，游戏中的额外角色和关卡、摄影 App 中的滤镜和文字处理器中的文具
- **App** 功能——解锁行为并拓展您已交付的功能。例如，提供多人模式作为App内购买项目的免费游戏，或允许用户进行一次性购买以移除广告的免费天气 App
- 服务——让用户为一次性服务（如语音转录）和持续性服务（如数据集访问权限）付费

您*不能*使用“App 内购买项目”销售现实世界的商品和服务，也不能销售不合适的内容。

- 现实世界的商品和服务——您在使用“App 内购买项目”时必须在 App 中提供数字商品或服务。要让您的用户在您的App中购买现实世界的商品和服务，请使用不同的付款机制，例如信用卡或付款服务
- 不合适的内容——请勿使用“App 内购买项目”销售《App 审核准则》不允许的内容，例如，色情作品、仇恨言论或诽谤

有关使用“App 内购买项目”您可以提供的内容的详细信息，请参见您的证书协议和《App 审核准则》。在编程之前仔细阅读准则可以帮助您避免审核过程中的延迟和拒绝。如果准则没有充分详细地列出您的情况，您可以使用在线联系表来向“App 审核”团队咨询具体问题。

在您明确要在您的 App 中销售哪些产品，并确定“App 内购买项目”是销售这些产品的合适方式之后，您需要在 iTunes Connect 中创建这些产品。

在 iTunes Connect 中创建产品

在您开始编程之前，您需要在 iTunes Connect 中配置与您 App 交互的产品。在您开发您的 App 时，您可以添加或移除产品，并优化或重新配置您的现有产品。

每个产品都与一个特定 App 相关联。为某个 App 创建的产品在其他 App 中不可用。不同平台上的配套 App 是不同的 App —— Mac App 中的产品在 iOS App 中不可用，反之亦然。

当您提交您的 App 作为 App 审核过程的一部分时，产品将会被审核。在用户能够购买一个产品之前，它必须经过审核人员的批准，您也必须将它在 iTunes Connect 中标记为“获准销售”。

有关在 iTunes Connect 中处理产品的分步信息，请参见 *《iTunes Connect 的 App 内购买项目配置指南》* 。

产品类型

产品类型让您通过提供几种不同的产品行为，在一系列的 App 中使用“App 内购买项目”。在 iTunes Connect 中，您可以选择以下产品类型之一：

- 消耗型产品——在运行您 App 的过程中逐渐被消耗的项目。例如，VoIP App 中的通话时长，和一次性服务（如语音转录）
- 非消耗型产品——在该用户的所有设备上无限期可用的项目。针对该用户的所有设备可用。例如，内容（如书籍和游戏关卡）和额外的 App 功能
- 自动续期订阅——分集内容。像非消耗型产品一样，自动续期订阅在该用户的所有设备上无限期可用。但与非消耗型产品不同，自动续期订阅具有到期日。您定期提供新的内容，用户在其有效订阅期内拥有访问已发布内容的权限。当自动续期订阅将要过期时，系统会自动代表用户续期
- 非续期订阅——不涉及分集内容提供的订阅。例如，访问历史照片数据库的权限，或航班图的合集。让订阅在用户的所有设备上可用，以及让用户能够恢复购买是您 App 的责任。当您的用户已在您的服务器上拥有一个（您可以在恢复内容时用来识别用户的）帐户时，通常会使用此产品类型。过期和订阅期限也由您的 App（或您的服务器）实施和执行

产品类型之间的差异

每一个产品类型都是为特定用途设计的。不同产品类型的行为在某些方面有所不同，相关总结请查看“表 1-1”和“表 1-2”。

表 **1-1**　产品类型比较

产品类型	非消耗型项目	消耗型项目
用户可以购买	一次	多次
在收据中显示	始终	一次
设备间同步	由系统执行	不同步

产品类型	非消耗型项目	消耗型项目
恢复	由系统执行	不可恢复

表 1-2　订阅类型比较

订阅类型	自动续期	非续期
用户可以购买	多次	多次
在收据中显示	始终	始终
设备间同步	由系统执行	由您的 App 执行
恢复	由系统执行	由您的 App 执行

已消耗或过期的产品——消耗型产品、自动续期订阅和非续期订阅——可以多次购买以再次获得消耗型项目或延长订阅。非消耗型产品解锁的是对用户无限期可用的内容，所以此类产品只能购买一次。

消耗型产品订阅在购买后出现在收据中，但在下次收据更新时会被移除，这在"保持使用 App 收据 （第 25 页）"中有更详细的讨论。在收据中有条目的所有其他类型的产品都没有被移除。

消耗型产品，其性质决定了它不被同步或恢复。例如，用户理解，在 iPhone 上额外购买 10 个 Bubble 不会同样给他们的 iPad 增加 10 个 Bubble。所有其他类型的产品在该用户的所有设备上都可用，也可以被恢复，所以用户即使在购买新设备后也可以继续访问已购买内容。StoreKit 为自动续期订阅和非消耗型产品处理同步和恢复流程。

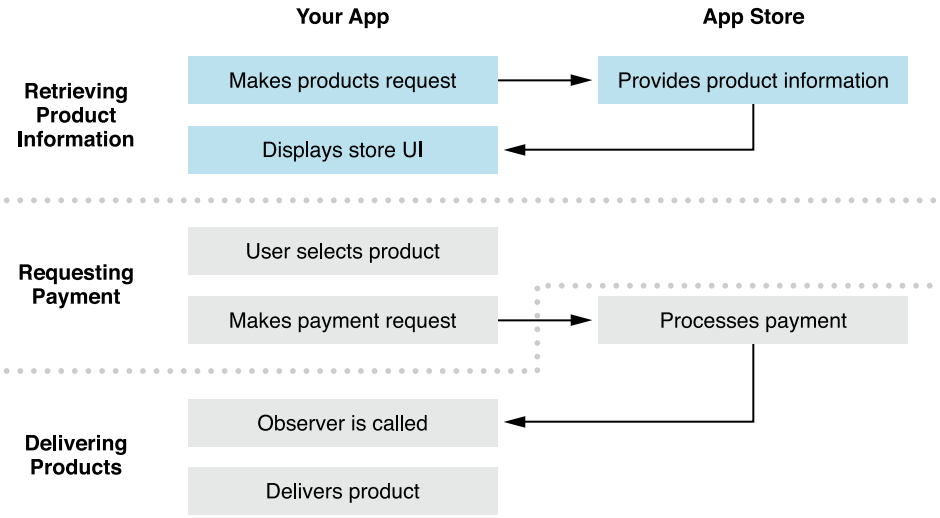
非续期订阅在几个关键方面不同于自动续期订阅，这些差异使您的App具有灵活性，以根据您的需求实施正确的行为，如下所述：

- 您的 App 负责计算有效订阅时间段，并决定哪些内容针对用户可用
- 您的 App 负责监测订阅是否接近到期日，并提示用户再次购买产品以续订订阅
- 您的App负责在用户购买后使订阅在他们的所有设备上可用，并允许用户恢复过去的购买。例如，大多数订阅由服务器提供，您的服务器将需要一些机制来识别用户并将订阅购买与购买它们的用户相关联

检索产品信息

在购买过程中的第一部分，您的 App 会从 App Store 中检索关于其产品的信息，向用户展示其商店 UI，然后让用户选择产品，如“图 2-1”中所示。

图 **2-1**　　购买过程的各阶段——展示商店 UI



获取产品标识符列表

您在您的 App 中销售的每个产品都有其唯一的*产品标识符*。您的 App 使用这些产品标识符从 App Store 上获取产品信息（如定价），并在用户购买这些产品时提交付款请求。您的 App 可以从其 App 数据包的某个文件中读取它的产品标识符列表，也可以从您的服务器上获取。“表 2-1”总结了这两种方法之间的差异。

表 **2-1**　　获取产品标识符的方法比较

	嵌入在 App 数据包中	从您的服务器上获取
用于购买	解锁功能	交付内容
产品列表可以更改	在 App 更新时	任意时间
需要服务器	否	是

如果您的 App 有固定的产品列表，例如用于移除广告或启用功能的 App 内购买项目，请将列表嵌入在 App 数据包中。如果产品标识符列表无需您的 App 更新便可以进行更改，例如一个支持额外关卡或角色的游戏，请让您的 App 从您的服务器上获取该列表。

在 iTunes Connect 中，没有运行时间机制可以为某个特定 App 获取所有配置产品的列表。您必须要负责管理您 App 的产品列表并为您的 App 提供该信息。如果您需要管理大量产品，请考虑使用 iTunes Connect 中的批量 XML 上传／下载功能。

在 App 数据包中嵌入产品 ID

请在您的 App 数据包中包括一个属性列表文件，该文件包含一个产品标识符数组，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <string>com.example.level1</string>
    <string>com.example.level2</string>
    <string>com.example.rocket_car</string>
</array>
</plist>
```

若要从属性列表中获取产品标识符，请在 App 数据包中找到并读取该文件。

```
NSURL *url = [[NSBundle mainBundle] URLForResource:@"product_ids"
                                     withExtension:@"plist"];
NSArray *productIdentifiers = [NSArray arrayWithContentsOfURL:url];
```

从您的服务器上获取产品 ID

在您的服务器上托管一个包含产品标识符的 JSON 文件。例如：

```
[
    "com.example.level1",
    "com.example.level2",
    "com.example.rocket_car"
]
```

若要从您的服务器上获取产品标识符，请获取并读取 JSON 文件，如“列表 2-1”中所示。请考虑将 JSON 文件版本化，以便您 App 的未来版本可以更改其架构，而无需破坏您 App 的旧版本。例如，您可以将使用旧架构的文件命名为 `products_v1.json`，并将使用新架构的文件命名为 `products_v2.json`。当您的 JSON 文件比示例中的简单数组更复杂时，这种做法尤其有用。

列表 **2-1**　从您的服务器上获取产品标识符

```
- (void)fetchProductIdentifiersFromURL:(NSURL *)url delegate:(id)delegate
{
    dispatch_queue_t global_queue =
           dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(global_queue, ^{
        NSError *err;
        NSData *jsonData = [NSData dataWithContentsOfURL:url
                                                 options:NULL
                                                   error:&err];
        if (!jsonData) { /* Handle the error */ }

        NSArray *productIdentifiers = [NSJSONSerialization
            JSONObjectWithData:jsonData options:NULL error:&err];
        if (!productIdentifiers) { /* Handle the error */ }

        dispatch_queue_t main_queue = dispatch_get_main_queue();
        dispatch_async(main_queue, ^{
           [delegate displayProducts:productIdentifiers]; // Custom method
        });
    });
}
```

有关使用 `NSURLConnection` 下载文件的信息，请参见《*URL Loading System Programming Guide（网址加载系统编程指南）*》中的“使用 NSURLConnection”。

为了确保您的 App 保持响应，请使用后台线程下载 JSON 文件并提取产品标识符列表。为了最小化数据传输，请使用标准 HTTP 缓存机制，如 `Last-Modified` 和 `If-Modified-Since` 标头。

检索产品信息

为了确保您的用户仅能看见实际可供购买的产品，请在显示您 App 的商店 UI 之前查询 App Store。

请使用产品请求对象来查询 App Store。首先，创建 `SKProductsRequest` 实例，并使用产品标识符列表将其初始化。请确保对请求对象保持强引用；否则系统可能会在请求完成之前释放请求。

产品请求检索有关有效产品以及无效产品标识符列表的信息，然后调用它的 Delegate（委托）来处理结果。Delegate（委托）必须实施 `SKProductsRequestDelegate` 协议来处理 App Store 的响应。“列表 2-2”展示了两段代码的简单实现。

列表 **2-2**　检索产品信息

```
// Custom method
- (void)validateProductIdentifiers:(NSArray *)productIdentifiers
{
    SKProductsRequest *productsRequest = [[SKProductsRequest alloc]
                    initWithProductIdentifiers:[NSSet
setWithArray:productIdentifiers]];

    // Keep a strong reference to the request.
    self.request = productsRequest;
    productsRequest.delegate = self;
    [productsRequest start];
}

// SKProductsRequestDelegate protocol method
- (void)productsRequest:(SKProductsRequest *)request
     didReceiveResponse:(SKProductsResponse *)response
{
    self.products = response.products;

    for (NSString *invalidIdentifier in response.invalidProductIdentifiers) {
        // Handle any invalid product identifiers.
    }

    [self displayStoreUI]; // Custom method
}
```

当用户购买产品时，您需要相应的产品对象来创建付款请求，所以请对返回 Delegate（委托）的产品对象数组保持引用。如果您 App 销售产品的列表可以更改，您可能想要创建一个自定义类，封装对产品对象的引用以及其他信息——例如，您从您的服务器上获取的图片或描述文本。付款请求在"请求付款（第 19 页）"中有所讨论。

返回为无效的产品标识符通常表示您 App 的产品标识符列表中出现了一个错误，它也可能意味着该产品在 iTunes Connect 中没有被妥善配置。良好的日志和 UI 会帮助您更轻松地解决此类问题。在生产构建版本中，您的 App 需妥善处理失败——通常这意味着在您的商店 UI 中删除无效产品，仅展示剩余的有效产品。在开发构建版本中，请显示错误以使该问题引起关注。无论是在生产构建版本还是开发构建版本中，都请您使用 `NSLog` 将信息写入控制台，以记录无效标识符。如果您的 App 从您的服务器上获取列表，您也可以定义一个日志记录机制来让您的 App 向您的服务器发回无效标识符列表。

展示您 App 的商店 UI

由于您 App 的商店设计对您的 App 内销售有重要影响，因此值得投入时间和精力将其做好。为您的商店 UI 设计用户界面，以使它与您 App 的其他部分融为一体。StoreKit 无法为您提供商店 UI。只有您足够了解您的 App 及其内容，因此也只有您能够设计出充分展现您产品亮点的商店 UI，并与您 App 的其他部分完美契合。

请在设计和实施您 App 的商店 UI 时参考以下准则。

仅在用户能付款时展示商店。若要确定用户是否可以付款，请调用 `SKPaymentQueue` 类的 `canMakePayments` 类方法。如果用户无法进行付款（例如，由于家长限制），您可以在 UI 中显示商店不可用，或完全删除您 UI 中的商店部分。

在您 **App** 的运行流程中自然地展示产品。在您 App 的 UI 中找到展示商店 UI 的最佳位置。当用户可以使用产品时，在情境中展示这些产品——例如，当用户尝试使用高级功能时提示用户解锁该功能。请特别注意用户初次探索您 App 时的体验。

整理产品，以使探索过程简单愉悦。如果您的 App 仅含有少量产品，您可以在一个屏幕中展示所有产品；否则，请将产品分组或分类以使它们易于导航。包含大量产品的 App，例如漫画阅读器或含有很多期刊的杂志，特别受益于可以让用户轻松发现她们想要购买的新项目的界面。为您的不同产品设置独特的名称和视觉效果以便作出清晰的区分——如有必要，请包含明确的比较。

将您的产品价值传达给用户。用户想清楚地知道他们将要购买什么。请将 App Store 上的产品信息（如产品价格和描述）与您服务器上或 App 数据包中的额外数据（如图片或演示）相结合。让用户在购买前以有限的方式与产品进行交互。例如，为用户提供购买新赛车选项的游戏，可以让用户使用新车试驾一圈。类似地，允许用户购买额外笔刷的绘画 App，可以让用户有机会在小便笺本上用新笔刷绘画，并查看不同笔刷间的区别。这样的设计给用户提供了体验产品的机会，并让他们确信想要购买该产品。

使用 **App Store** 返回的语言区和货币，清晰地显示价格。请确保产品价格易于找到和阅读。请勿尝试在您的 UI 中将价格转换为不同的货币，即使用户的语言区和价格的语言区不同。请考虑，例如，一个位于美国的用户偏好英国语言区的单位和时间格式。您的 App 会根据英国语言区展示其 UI，但它仍然需以 App Store 指定语言区展示其产品信息。将价格转化为英国英镑，试图与界面中其他英国语言区部分匹配的行为是错误的。该用户的 App Store 帐户位于美国，并使用美元付款，因此价格会以美元的形式提供至您的 App。类似地，您的 App 也将以美元的形式显示价格。“列表 2-3”展示了如何使用产品的语言区信息正确设置价格格式。

列表 **2-3**　　设置产品价格格式

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4];
[numberFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];
[numberFormatter setLocale:product.priceLocale];
NSString *formattedPrice = [numberFormatter stringFromNumber:product.price];
```

在用户选择了要购买的产品后，您的 App 会连接至 App Store 以请求针对该产品的付款。

推荐的测试步骤

测试您代码的各部分，以验证您已正确将其实施。

使用您的测试帐户登录至 App Store

在 iTunes Connect 中创建一个测试用户帐户，如“创建测试用户帐户”中所述。

在一个开发 iOS 设备的“设置”中，注销 App Store。然后在 Xcode 中构建并运行您的 App。

在一个开发 macOS 设备上，从 Mac 版 App Store 中注销。然后在 Xcode 中构建您的 App，并在 Finder 中启动它。

使用您的 App 来购买 App 内购买项目。当系统提示您登录至 App Store 时，请使用您的测试帐户。请注意，文本“[Environment: Sandbox]”显示为提示的一部分，指明您已连接到此测试环境。

如果文本“[Environment: Sandbox]”没有显示，则您使用的是生产环境。请确保您正在运行一个您 App 的登录至开发环境的构建版本。登录到生产环境的构建版本使用生产环境。

重要事项: 请勿使用您的测试用户帐户登录至生产环境。如果您这样做，测试用户帐户将会失效并无法再使用。

测试获取产品标识符列表

如果您的产品标识符嵌入在您的 App 中，请在它们加载后的代码中设置断点，验证 `NSArray` 实例包含预期的产品标识符列表。

如果您的产品标识符是从服务器上获取的，请手动获取 JSON 文件（使用网页浏览器，如 Safari；或命令行实用程序，如 `Curl`），并验证从您的服务器上返回的数据是否包含预期产品标识符列表。此外，请验证您的服务器正确实施了标准 HTTP 缓存机制。

测试处理无效产品标识符

请有意识地在您 App 的产品标识符列表中包含一个无效产品标识符。（请确保在测试之后将其移除。）

在生产构建版本中，验证 App 是否显示其余的商店 UI，及用户是否可以购买其他产品。在开发构建版本中，验证 App 是否将该问题呈现以引起您的注意。

请检查控制台日志并验证您是否能够正确识别无效产品标识符。

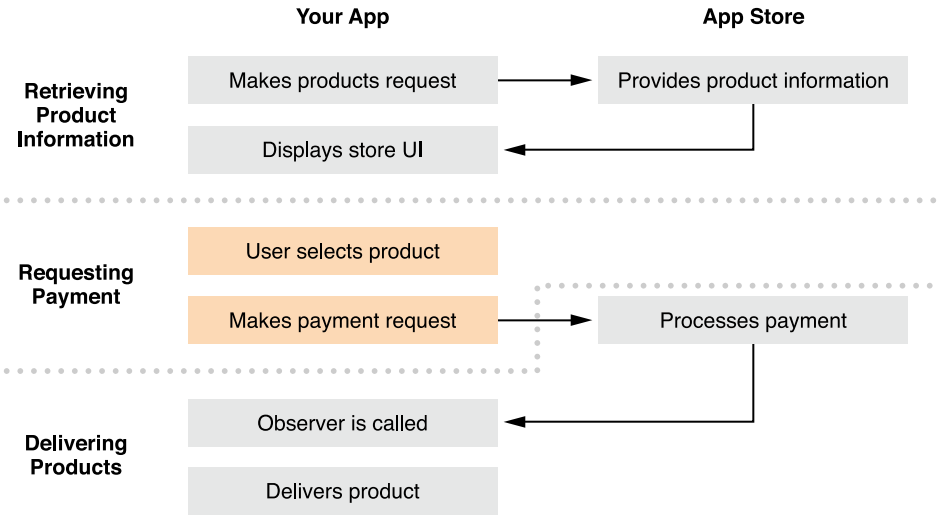
测试产品请求

使用您测试的产品标识符列表，创建并提交一个 `SKProductsRequest` 实例。在您的代码中设置断点，检查有效和无效产品标识符列表。如果存在无效产品标识符，请在 iTunes Connect 中检查您的产品，并更正您的 JSON 文件或属性列表。

请求付款

在购买过程的第二部分，在用户选择购买某个特定产品之后，您的 App 将会向 App Store 提交付款请求，如“图 3-1”中所示。

图 **3-1**　　购买过程的各阶段——请求付款



创建付款请求

当用户选择了要购买的产品时，使用一个产品对象来创建付款请求，并根据需要设置数量，如“列表 3-1”中所示。产品对象来自您 App 的产品请求返回的产品数组，如“检索产品信息 （第 15 页）”中所述。

列表 **3-1**　　创建付款请求

```
SKProduct *product = <# Product returned by a products request #>;
SKMutablePayment *payment = [SKMutablePayment paymentWithProduct:product];
payment.quantity = 2;
```

监测异常活动

App Store 使用异常活动监测引擎来帮助打击欺诈。一些 App 可以提供额外信息以提高引擎监测异常交易的能力。如果您的用户在您的服务器上拥有帐户，除了他们的 App Store 帐户，请在请求付款时提供此项额外信息。

为了说明，请考虑以下两个示例：在正常情况下，许多不同的用户在您的服务器上分别购买在游戏中使用的金币，但每个用户使用不同的 App Store 帐户支付购买费用。相比之下，一个用户在您的服务器上多次购买金币，但每次使用不同的 App Store 帐户支付购买费用，是很异常的。App Store 无法自行监测此类异常活动——它需要您的 App 提供在您的服务器上与每笔交易相关联的帐户信息。

若要提供此信息，请填写付款对象的 `applicationUsername` 属性，使用您服务器上用户帐户名称的单向哈希（one-way hash），如列表 3-2中的示例所示。

列表 **3-2**　　提供应用程序用户名

```
#import <CommonCrypto/CommonCrypto.h>

// Custom method to calculate the SHA-256 hash using Common Crypto
- (NSString *)hashedValueForAccountName:(NSString*)userAccountName
{
    const int HASH_SIZE = 32;
    unsigned char hashedChars[HASH_SIZE];
    const char *accountName = [userAccountName UTF8String];
    size_t accountNameLen = strlen(accountName);

    // Confirm that the length of the user name is small enough
    // to be recast when calling the hash function.
    if (accountNameLen > UINT32_MAX) {
        NSLog(@"Account name too long to hash: %@", userAccountName);
        return nil;
    }
    CC_SHA256(accountName, (CC_LONG)accountNameLen, hashedChars);

    // Convert the array of bytes into a string showing its hex representation.
    NSMutableString *userAccountHash = [[NSMutableString alloc] init];
    for (int i = 0; i < HASH_SIZE; i++) {
        // Add a dash every four bytes, for readability.
```

```
        if (i != 0 && i%4 == 0) {
            [userAccountHash appendString:@"-"];
        }
        [userAccountHash appendFormat:@"%02x", hashedChars[i]];
    }

    return userAccountHash;
}
```

如果您使用其他方式来填写此属性，请确保您提供的值是与您服务器上用户的帐户唯一相关联的不透明标识符。请勿使用您的开发人员帐户的 Apple ID、用户的 Apple ID，或该用户在您服务器上其他未经哈希（unhashed）处理的帐户名称。

提交付款请求

向交易队列添加付款请求会将其提交至 App Store。如果您多次向队列添加同一付款对象，它会被多次提交——用户将会被多次收费，您的 App 也必须要多次交付产品。

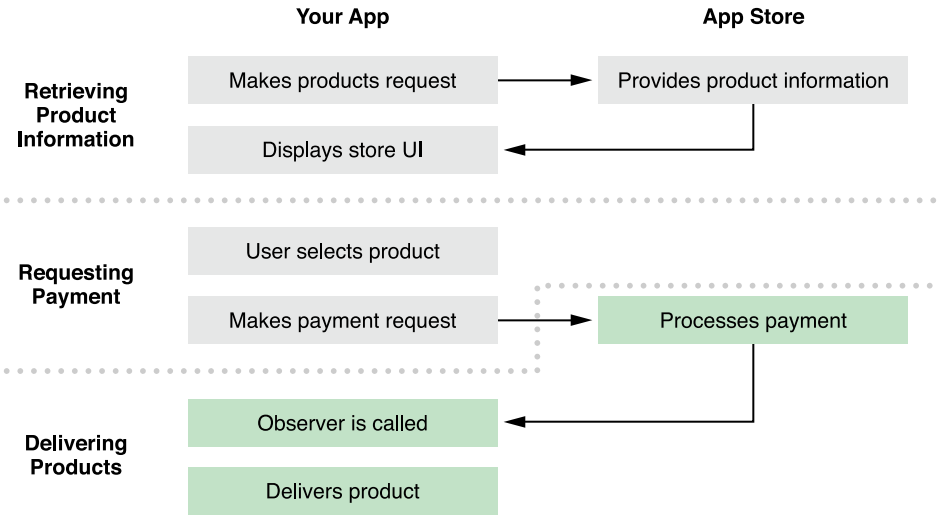
```
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

对于您 App 提交的每个付款请求，App 都会取回一个对应的必须处理的交易。交易和交易队列在"等待 App Store 处理交易 （第 22 页）"中有所讨论。

交付产品

如"图 4-1"所示，在购买过程的最后阶段，您的 App 会等待 App Store 处理付款请求，为将来的启动存储关于购买的信息，下载已购买内容，然后将交易标记为已完成。

图 **4-1**　　购买过程的各阶段——交付产品



等待 App Store 处理交易

*交易队列*在让您的 App 与 App Store 通过 StoreKit framework（StoreKit 框架）进行通信的过程中起到了核心作用。您将 App Store 需要执行操作的工作添加到队列中，例如需要处理的付款请求。当交易状态发生变化（例如，付款请求成功时），StoreKit 会调用 App 的 *Transaction Queue Observer（交易队列观察器）*。您可以决定将哪个类作为 Observer（观察器）。在非常小的 App 中，您可以使用 AppDelegate（App 委托）处理所有的 StoreKit 逻辑，包括观察交易队列。在大多数 App 中，您必须要创建一个单独的类来处理 Observer（观察器）逻辑和 App 的其他存储逻辑。Observer（观察器）必须符合 `SKPaymentTransactionObserver` 协议。

使用 Observer（观察器）意味着您的 App 不会持续轮询其活动交易的状态。除了将交易队列用于付款请求外，您的 App 还使用交易序列下载 Apple 托管内容，并查找完成更新的订阅。

如"列表 4-1"所示，在您的 App 启动时注册 Transaction Queue Observer（交易队列观察器）。确保 Observer（观察器）随时可以处理交易，而不是仅在您将交易添加到队列之后。例如，请考虑用户在您的 App 购买东西后进入隧道的情况。您的 App 会因为没有网络连接而无法交付内容。当您的 App 下一次启动时，

您的 Transaction Queue Observer（交易队列观察器）会再次被 StoreKit 调用，从而交付购买的内容。同样地，如果您的 App 无法将某笔交易标记为完成，StoreKit 将在每一次启动 App 时调用 Observer（观察器），直到交易正确完成。

列表 **4-1**　注册 Transaction Queue Observer（交易队列观察器）

```
- (BOOL)application:(UIApplication *)application
 didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    /* ... */

    [[SKPaymentQueue defaultQueue] addTransactionObserver:observer];
}
```

在您的 Transaction Queue Observer（交易队列观察器）中实现 `paymentQueue:updatedTransactions:` 方法。当交易状态发生变化，如付款请求处理完成时，StoreKit 会调用该方法。如“表 4-1”及“列表 4-2”所示，交易状态告诉您 App 必须执行的操作。队列中的交易可按任意顺序更改状态。您的 App 需要随时准备处理任何活动交易。

表 **4-1**　交易状态和相应操作

状态	您 App 中执行的操作
`SKPaymentTransaction-StatePurchasing`	更新您的 UI 以反映正在进行的状态，并等待再次调用。
`SKPaymentTransactionStateDeferred`	更新您的 UI 以反映延期状态，并等待再次调用。
`SKPaymentTransactionStateFailed`	使用 `error` 属性的值向用户展示信息。有关 error 常量的列表，请参见 `SKErrorDomain`。
`SKPaymentTransaction-StatePurchased`	提供已购买功能。
`SKPaymentTransactionStateRestored`	恢复先前购买的功能。

列表 **4-2**　响应交易状态

```
- (void)paymentQueue:(SKPaymentQueue *)queue
 updatedTransactions:(NSArray *)transactions
{
```

```
    for (SKPaymentTransaction *transaction in transactions) {
        switch (transaction.transactionState) {
            // Call the appropriate custom method for the transaction state.
            case SKPaymentTransactionStatePurchasing:
                [self showTransactionAsInProgress:transaction deferred:NO];
                break;
            case SKPaymentTransactionStateDeferred:
                [self showTransactionAsInProgress:transaction deferred:YES];
                break;
            case SKPaymentTransactionStateFailed:
                [self failedTransaction:transaction];
                break;
            case SKPaymentTransactionStatePurchased:
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored:
                [self restoreTransaction:transaction];
                break;
            default:
                // For debugging
                NSLog(@"Unexpected transaction state %@",
@(transaction.transactionState));
                break;
        }
    }
}
```

为了在等待时使您的用户界面保持最新，Transaction Queue Observer（交易队列观察器）可以从 `SKPaymentTransactionObserver` 协议中实现可选方法，具体如下。当交易从队列移除时，`paymentQueue:removedTransactions:` 方法会被调用。在您实现此方法时，请从您 App 的 UI 中移除对应的项目。当 StoreKit 完成恢复交易后，`paymentQueueRestoreCompletedTransactionsFinished:` 或 `paymentQueue:restoreCompletedTransactionsFailedWithError:` 方法会被调用，取决于过程中是否出现错误。在您实现这些方法时，请更新 App 的 UI 以反映是否成功、有无报错。

保持购买记录

在使产品可用后，您的 App 需要持续记录购买情况。您的 App 在启动时使用该持续记录以便让产品继续可用。如"恢复已购产品 （第 39 页）"所示，它还使用该记录恢复购买。您 App 的持续策略取决于您销售产品的类型以及 iOS 的版本。

- 针对 iOS 7 及更高版本中的非消耗型产品和自动续期订阅，使用 App 收据作为您的持续记录
- 针对低于 iOS 7 的 iOS 版本中的非消耗型产品和自动续期订阅，使用"用户默认设置"系统或 iCloud 来保持持续记录。
- 针对非续期订阅，使用 iCloud 或您的服务器来保持持续记录
- 针对消耗型产品，您的App会更新其内部状态以反映购买，但无需进行持续记录，因为消耗型产品不会在设备间进行恢复或同步。请确保上传后的状态是支持状态保存的对象的一部分（iOS 中），或者，您也可以在 App 启动过程中手动保存状态（iOS 或 macOS 中）。有关状态保存的信息，请参见"状态保存与恢复"。

在使用"用户默认设置"系统或 iCloud 时，您的 App 可以存储数字或布尔型数值，也可以存储交易收据的副本。在 macOS 中，用户可以使用默认按钮编辑"用户默认设置"系统。存储收据需要更多的应用逻辑，但能防止持续记录被篡改。

当通过 iCloud 进行持续记录时，请注意您 App 的持续记录会在设备间同步，但您的 App 必须负责在其他设备上下载任何相关内容。

使用 App 收据保持购买记录

App 收据包含由 Apple 加密签名的用户购买记录。更多信息，请参见《*Receipt Validation Programming Guide（收据验证编程指南）*》。

关于消耗型产品的信息在付款完成后会添加到收据，并保留在收据中，直到交易结束。结束交易后，该信息将在下一次收据更新时被移除，例如，当用户进行下一次购买。

所有其他类型购买的信息在付款完成会添加到收据，并永久保留在收据中。

在"用户默认设置"或 iCloud 中保存值

如需将信息存储在"用户默认设置"或 iCloud 中，请设置 Key。

```
#if USE_ICLOUD_STORAGE
NSUbiquitousKeyValueStore *storage = [NSUbiquitousKeyValueStore defaultStore];
#else
NSUserDefaults *storage = [NSUserDefaults standardUserDefaults];
```

```
#endif

[storage setBool:YES forKey:@"enable_rocket_car"];
[storage setObject:@15 forKey:@"highest_unlocked_level"];

[storage synchronize];
```

在“用户默认设置”或 iCloud 中保存收据

如需将交易收据存储在“用户默认设置”或 iCloud 中，请为该收据的数据设置 Key。

```
#if USE_ICLOUD_STORAGE
NSUbiquitousKeyValueStore *storage = [NSUbiquitousKeyValueStore defaultStore];
#else
NSUserDefaults *storage = [NSUserDefaults standardUserDefaults];
#endif

NSData *newReceipt = transaction.transactionReceipt;
NSArray *savedReceipts = [storage arrayForKey:@"receipts"];
if (!savedReceipts) {
    // Storing the first receipt
    [storage setObject:@[newReceipt] forKey:@"receipts"];
} else {
    // Adding another receipt
    NSArray *updatedReceipts = [savedReceipts arrayByAddingObject:newReceipt];
    [storage setObject:updatedReceipts forKey:@"receipts"];
}

[storage synchronize];
```

保持使用您自己的服务器

将收据的副本连同某些凭据或标识符发送到您的服务器上，以便您能跟踪哪些收据属于哪位特定用户。例如，让用户通过向您的服务器发送电子邮件或用户名加密码的方式标识自己。请不要使用 `UIDevice` 中的 `identifierForVendor` 属性——您不能使用它来识别并恢复同一用户在不同设备上的购买，因为不同设备上该属性的值不同。

解锁 App 功能

如果该产品将启动 App 的某项功能，请设置一个布尔值以启用代码路径，并根据需要更新您的用户界面。要确定什么功能被解锁，请查询交易发生时您的App所做的持续记录。当购买完成和App启动时，您的 App 需要更新该布尔值。

例如，使用 App 收据，您的代码可能如下所示：

```
NSURL *receiptURL = [[NSBundle mainBundle] appStoreReceiptURL];
NSData *receiptData = [NSData dataWithContentsOfURL:receiptURL];

// Custom method to work with receipts
BOOL rocketCarEnabled = [self receipt:receiptData
        includesProductID:@"com.example.rocketCar"];
```

或者，使用“用户默认设置”系统：

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
BOOL rocketCarEnabled = [defaults boolForKey:@"enable_rocket_car"];
```

然后使用该信息在您的 App 中启用响应的代码路径。

```
if (rocketCarEnabled) {
    // Use the rocket car.
} else {
    // Use the regular car.
}
```

交付相关内容

如果产品含有相关内容，您的App需要将该内容交付给用户。例如，在游戏中购买关卡必须要交付定义关卡的文件，在音乐 App 中购买额外的乐器必须要交付能使用户弹奏这些乐器的声音文件。

您可以将内容嵌入 App 数据包中，或在有需要的时候下载，两种方法各有利弊。如果您 App 数据包中的内容太少，即使在进行小型购买时，用户也必须要等待。如果您App数据包中的内容太多，App初始下载耗时过长，对于不购买相应产品的用户来说是一种空间浪费。另外，如果您的App过大，用户将无法通过蜂窝移动网络下载。

请在您的 **App** 中嵌入较小的文件（最多几兆字节），特别是当您希望大多数用户购买该产品时。您 App 数据包的内容在用户购买后应立即可用。但是，若要添加或更新 App 数据包中的内容，您必须要提交一个您 App 的更新版本。

仅在有需要时下载更大的文件。通过分割 App 数据包中的内容，保持较小的 App 初始下载大小。例如，一个游戏可以在 App 数据包中包含第 1 关，让用户在购买时下载其余关卡。若您的 App 从您的服务器上获取产品标识符列表，且没有在 App 数据包中硬编码，则您无需通过重新提交 App 来添加或更新您 App 已下载的内容。

在 iOS 6 及更高版本中，大多数 App 应为已下载的文件使用 Apple 托管内容。您在 Xcode 中使用“App 内购买项目内容” Target 创建 Apple 托管内容包，并将其提交至 iTunes Connect。当您在 Apple 服务器上托管内容时，您无需提供任何服务器，Apple 在存储您 App 的内容时使用的基础架构与支持其他大规模操作，如 App Store 时使用的相同。另外，即使您的 App 未运行，Apple 托管的内容也会自动在后台下载。

如果您已有服务器基础架构、需要支持旧版本的 iOS 或您在多个平台上共享服务器基础架构，您可以选择自己托管内容。

> 注意：您不能为您 App 的二进制文件打补丁，也不可以下载可执行代码。在提交时，您的 App 必须包含所有支持其全部功能的可执行代码。如果新产品必须要更改代码，请提交您 App 的更新版本。

加载本地内容

与您从 App 数据包中加载其他资源一样，请使用 `NSBundle` 类加载本地内容。

```
NSURL *url = [[NSBundle mainBundle] URLForResource:@"rocketCar"
                                     withExtension:@"plist"];
[self loadVehicleAtURL:url];
```

从 Apple 服务器上下载托管内容

当用户购买的产品与 Apple 托管内容相关联时，添加到您的 Transaction Queue Obeserver（交易队列观察器）上的交易还会包括一个 `SKDownload` 实例，让您下载相关联内容。

若要下载该内容，请调用 `SKPaymentQueue` 中的 `startDownloads:` 方法，以从交易的 `downloads` 属性中将下载对象添加到交易队列。如果 `downloads` 属性值为 `nil`，则该交易没有 Apple 托管内容。与下载 App 不同，下载内容不会在内容大小超过某个特定值时自动要求 Wi-Fi 连接。请避免用户在没有明确操作的情况下，使用蜂窝移动网络下载大文件。

请在 Transaction Queue Observer（交易队列观察器）中实现 `paymentQueue:updatedDownloads:` 方法，以响应下载状态的更改，例如，以更新 UI 上的进度的形式。如果下载失败，请使用其 `error` 属性中的信息将错误呈现给用户。

确保您的 App 妥善处理错误。例如，如果设备在下载期间的磁盘空间不足，请让用户选择放弃部分下载或在稍后空间可用时恢复下载。

使用 `progress` 和 `timeRemaining` 属性的值在下载内容时更新您的用户界面。您可以使用您的 UI 里的 `pauseDownloads:` 方法、`resumeDownloads:` 方法和 `cancelDownloads:` 方法（位于 `SKPaymentQueue` 中），让用户控制进行中的下载。请使用 `downloadState` 属性来确定下载是否完成。不要使用下载对象的 `progress` 或 `timeRemaining` 属性检查其状态——这些属性只用于更新您的 UI。

> 注意：在结束交易前，请下载所有 Apple 托管的内容。交易完成后将无法使用其下载对象。

在 iOS 中，您的 App 可以管理下载的文件。这些文件由 StoreKit framework（StoreKit 框架）保存在 `Caches` 目录中，且未设置备份旗标。下载完成后，您的 App 负责将其移动到相应的位置。针对可删除内容，如果设备耗尽磁盘空间（以后由应用程序重新下载），请将文件保留在 `Caches` 目录中。否则，请将文件移动到 `Documents` 文件夹，并将备份标记设置为从用户备份中除去。

列表 **4-3**　从备份中除去已下载内容

```
NSError *error;
BOOL success = [URL setResourceValue:[NSNumber numberWithBool:YES]
                               forKey:NSURLIsExcludedFromBackupKey
                                error:&error];
if (!success) { /* Handle error... */ }
```

在 macOS 中，已下载的文件由系统管理；您的 App 无法直接移动或删除它们。若要在下载之后定位内容，请使用下载对象的 `contentURL` 属性。若要在后续启动时定位文件，请使用 `SKDownload` 中的 `contentURLForProductID:` 类方法。若要删除文件，请使用 `deleteContentForProductID:` 类方法。更多有关从您的 App 收据中读取产品标识符的信息，请参见《*Receipt Validation Programming Guide（收据验证编程指南）*》。

从您的服务器上下载内容

就像您 App 与您服务器进行的其他交互一样，从您自己的服务器上下载内容的过程的细节和机制由您负责。通信至少包含以下步骤：

1. 您的 App 将收据发送到您的服务器，并请求内容。

2. 您的服务器会验证收据，确认内容已被购买，如《*Receipt Validation Programming Guide（收据验证编程指南）*》中所述。
3. 假设收据有效，您的服务器会响应您的 App 并发送内容。

确保您的 App 妥善处理错误。例如，如果设备在下载期间的磁盘空间不足，请让用户选择放弃部分下载或在稍后空间可用时恢复下载。

考虑您托管内容以及您 App 与您服务器通信的安全性后果。更多信息，请参见“*安全性概述*”。

结束交易

结束交易以告知 StoreKit 您已完成购买所需一切工作。未结束的交易在结束之前将保留在队列中，每次您的 App 启动时都会调用 Transaction Queue Observer（交易队列观察器），以便您的 App 能够结束交易。您的 App 必须结束每个交易，无论交易成功与否。

在结束交易前，请完成以下所有操作：

- 保持购买记录
- 下载相关内容
- 更新您的 App UI，以便让用户访问该产品

若要结束交易，请在付款队列中调用 `finishTransaction:` 方法。

```
SKPaymentTransaction *transaction = <# The current payment #>;
[[SKPaymentQueue defaultQueue] finishTransaction:transaction];
```

结束交易后，不要对该交易采取任何行动或者进行任何交付产品的工作。如果仍有工作未完成，则您的 App 尚未准备好结束交易。

> 注意：在交易实际完成之前，不要试图调用 `finishTransaction:` 方法，请尝试使用您 App 中的其他机制跟踪未完成的交易。StoreKit 不是为该用途设计的。这样做会导致您的 App 无法下载 Apple 托管内容并可能会导致其他问题。

建议的测试步骤

测试您代码的各部分以验证您已正确实现功能。

测试付款请求

使用您已经测试的有效产品标识符创建一个 `SKPayment` 的实例。设置断点并检查付款请求。添加付款请求到交易队列，并设置断点，以确认您Observer（观察器）的 `paymentQueue:updatedTransactions:` 方法已被调用。

在测试期间，可以立即结束交易而不提供内容。然而，即使在测试期间，结束交易失败也会导致问题：未完成的事务将无限期保留在队列中，可能干扰后续测试。

验证您的 Observer（观察器）代码

审核 Transaction Observer（交易观察器）对 `SKPaymentTransactionObserver` 协议的实现情况。请验证它是否可以处理交易，即使您当前没有展示您 App 的商店 UI，或您最近没有发起购买。

在您的代码中找到调用 `addTransactionObserver:` 方法，该方法属于 `SKPaymentQueue`。验证您的 App 在启动时会调用该方法。

测试成功的交易

使用测试用户帐户登录 App Store，并在您的 App 中进行购买。在您实现的 Transaction Queue Observer（交易队列观察器）的 `paymentQueue:updatedTransactions:` 方法设置一个断点，并检查交易，以验证其状态是否为 `SKPaymentTransactionStatePurchased`。

在您代码中持续购买的部分设置一个断点，确认该代码在响应成功购买时被调用。检查“用户默认设置”或 iCloud Key 值存储，确认已记录正确信息。

测试中断的交易

在您 Transaction Queue Observer's（交易队列观察器）的 `paymentQueue:updatedTransactions:` 方法中设置一个断点，让您可以控制其是否交付产品。然后在测试环境中像往常一样进行购买，并使用断点暂时忽略该交易，例如，通过使用 LLDB 中的 `thread return` 命令立即从该方法中返回。

终止并重新启动您的 App。StoreKit 在启动后很快会再次调用 `paymentQueue:updatedTransactions:` 方法，这次请让您的 App 正常响应。验证您的 App 已正确交付产品并完成交易。

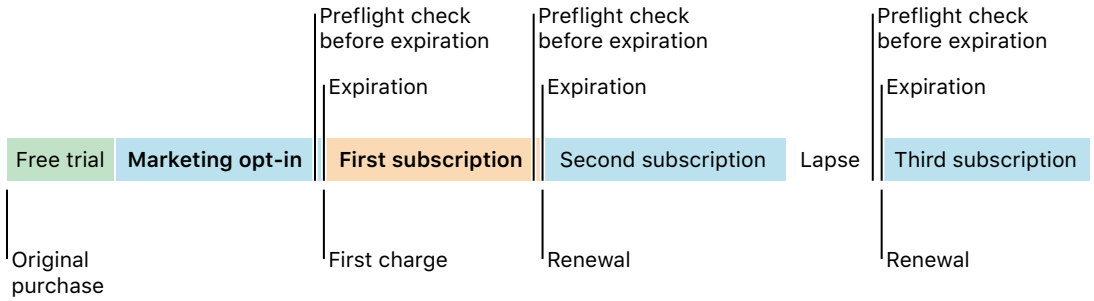
验证交易是否完成

找到您的 App 调用 `finishTransaction:` 方法。验证与该交易相关的所有工作是否已在调用该方法前完成，且每个交易都会调用该方法，无论成功与否。

处理订阅

提供订阅的 App 有一些额外行为和注意事项。由于订阅包含时间元素，您的 App 必须能够确定该订阅当前是否有效，以及过去的订阅状态。您的App还必须响应新订阅、续期订阅和中断订阅，并妥善处理过期订阅。“图 5-1”展示了一个示例订阅时间线，包括了您 App 需要处理的一些复杂事项。

图 **5-1**　　示例订阅时间线



计算订阅有效时间段

您的App需要根据订阅有效的时间段确定用户具有访问哪些内容的权限。例如，一个用户订阅了一份月刊杂志，该杂志在每月的第一天发布新期刊。请考虑“表 5-1”中所示的时间线。

表 **5-1**　　示例按月订阅时间线

日期	事件	订阅状态
2 月 1 日	二月期刊发行。用户尚未订阅，此期刊不可用。	尚未订阅
2 月 20 日	用户开始按月订阅。2 月期刊为最新期刊且立即可用。	有效
3 月 1 日	3 月期刊发行。它立即可用，因为该用户订阅有效。	有效
3 月 20 日	该用户订阅自动续期一个月。	有效
4 月 1 日	4 月期刊发行。它立即可用，因为该用户订阅有效。	有效
4 月 20 日	用户取消订阅，订阅期结束。	中断
5 月 1 日	5 月期刊发行。它不可用，因为用户订阅已中断。	中断

日期	事件	订阅状态
6 月 1 日	6 月期刊发行。它不可用，因为用户订阅已中断。	中断
6 月 17 日	用户重新订阅。6 月期刊立即可用。	有效
7 月 1 日	7 月期刊发行。它立即可用，因为该用户订阅有效。	有效

若要为用户提供访问所有内容的权限，请记录每部分内容的发布日期。请读取每个收据条目的“Original Purchase Date”（原始购买日期）和“Subscription Expiration Date”（订阅到期日）栏来确定该订阅的开始和结束日期。（有关收据的更多信息，请参见《*Receipt Validation Programming Guide（收据验证编程指南）*》。）用户可以访问从订阅开始日期到结束日期之间的所有发布内容、以及购买订阅时最初解锁的内容。如果订阅中断，会有多个在订阅有效期间的时间段、以及在订阅开始时解锁的内容。

要识别订阅中断，请将每个收据条目中的“Subscription Expiration Date”（订阅到期日）栏与之前收据条目中“Purchase Date”（购买日期）栏进行比较。

> 备注：请勿将购买日期和订阅时限相加来计算订阅期。这种方法没有考虑到免费试用期、参与营销期和用户购买订阅后立即可用的内容。
>
> 例如，由于购买订阅时总会解锁内容，对于一份每月第一天发行新期刊的月刊杂志来说，在月中购买按月订阅的用户会在订阅首月获得两份期刊：最新发行的期刊（在购买时解锁）和下月 1 日发行的期刊（在您发行时解锁）。

沿用表 5-1中的例子，收据将显示如下开始和结束日期：

- 2 月 20 日 - 3 月 20 日
- 3 月 20 日 - 4 月 20 日
- （中断时间段为 4 月 20 日 - 6 月 17 日，*没有*明确记录在收据中。）
- 6 月 17 日 - 7 月 17 日

用户可以访问 2 月和 6 月的期刊，因为它们在订阅被购买或重新启动时被最初解锁。

用户可以访问 3 月、4 月、6 月和 7 月的期刊，因为在这些时间订阅有效。

升级与计划更改

用户可以在他们 App Store 的用户设置或您 App 的界面中管理订阅。对每一个订阅，App Store 都会显示该订阅组提供的所有续期选项。用户可以轻松的更改服务级别，并随时选择升级、降级或跨级。任何时限的降级或不同时限的跨级都将在下个续期日生效。

您可以检查收据的“Subscription Auto Renew Preference”（自动续订订阅偏好）栏，了解将在下个续订日起生效的、用户所选的任何计划更改。（关于收据的信息，请参见《*ReceiptValidationProgrammingGuide（收据验证编程指南）*》。）

过期和续期

订阅续期过程开始于到期日前 10 天。在这 10 天中，App Store 会检查是否存在任何可能推迟或阻止订阅自动续期的计费问题，例如：

- 用户不再拥有有效的支付方式
- 用户购买订阅后产品价格上涨
- 或产品不再可用

如果存在任何问题，App Store 会通知用户，以便用户可以在订阅过期之前解决，防止订阅服务中断。

在订阅过期前的 24 小时内，App Store 会开始尝试自动续期。 App Store 会在一段时间内多次尝试自动续期订阅，但如果尝试失败过多，最终会停止尝试。

> 备注：对于计费相关问题，App Store 会在最多 60 天里尝试续期。您可以检查收据中的“Subscription Retry Flag”（订阅重试旗标），确定 App Store 是否仍在尝试续订订阅。

App Store 会在订阅即将过期时续期订阅，以防止订阅中出现任何中断。但是，订阅中断仍然可能出现。例如，如果用户的付款信息不再有效，则第一次续期尝试会失败。如果用户在订阅过期后才更新此信息，那么在到期日和随后自动续期成功的日期之间将会出现一个短暂的订阅中断。用户也可以禁用自动续期，有意使订阅过期，然后在晚些时候续订，形成一个更长的订阅中断。请确保您 App 的订阅逻辑能够正确处理各种时限的中断。您可以检查订阅自动更新状态栏，确定订阅的更新状态。

在订阅成功续期后，StoreKit 会在交易队列中为续期添加一个交易。您的 App 将在启动时检查交易队列，并以处理任何其他交易的相同方式来处理续期。请注意，如果续期订阅时您的 App 正在运行，那么 Transaction Observer（交易观察器）*不会*被调用；您的 App 将会在下次启动时检查到续期。

作为示例，下面的时间线显示了一个用户账户的订阅，该订阅属于一个按月提供服务的 App。在示例中，订阅因为计费问题暂时中断。用户解决了这一问题，订阅在一个新的按月订阅日期进行续期。

表 5-2　示例按月订阅时间线

日期	事件	订阅状态
2 月 20 日	用户订阅，时限为一个月。订阅服务立即可用。	有效
3 月 20 日	该用户订阅自动续期一个月。订阅服务仍然可用。	有效
4 月 19 日	用户的付款方式过期。	有效
4 月 20 日	由于交易失败，用户订阅未能续期。订阅中断。	中断
5 月 5 日	用户更新其付款方式。App Store 能够成功续期订阅服务。订阅服务立即可用。	有效
6 月 5 日	该用户订阅自动续期一个月。订阅服务仍然可用。	有效

取消订阅

在购买订阅时必须要全额支付。用户仅可以通过联系 Apple 顾客服务获得退款。例如，如果用户意外购买了错误的产品，顾客支持可以取消订阅并发起全额或部分退款。顾客可能会在订阅期间取消订阅，但仍必须持续为订阅付费至该订阅期末。

若要检查某个购买是否被“Apple 顾客支持”取消，请在收据中查找“Cancellation Date”栏。如果该栏中含有日期，无论订阅的到期日为何时，该购买都已经被取消。在提供内容或服务方面，取消交易可视作没有发生购买。

根据您 App 提供的产品类型，您可能需要查看当前有效的订阅期，或您可能需要查看过去所有的订阅期。例如，一个杂志 App 需要查看过去所有的订阅期以确定用户应该可以访问哪些期刊。提供流媒体服务的 App 仅需查看当前有效的订阅期以确定用户是否应该拥有访问其服务的权限。

状态更新通知

`statusUpdateNotification` 是针对自动续订订阅的、服务器对服务器的通知服务。通知指定的是通知发送时的订阅状态。

若要在您处理事件时获得最新信息，您的 App 应通过 App Store 验证最新收据。建议您使用状态更新通知服务配合收据验证，来验证用户的当前订阅状态并为其提供服务。关于收据验证的信息，请参见 *《Receipt Validation Programming Guide（收据验证编程指南）》*。

若要接收状态更新通知，请在 iTunes Connect 中为您的 App 配置订阅状态网址（URL）。App Store 将为“表 5-3 （第 36 页）”中列出的关键订阅事件通过 HTTP POST 向您的服务器交付 JSON 对象。您的服务器负责解析、解读和回应所有 `statusUpdateNotification` POST。

> 备注：是否使用服务器对服务器通知服务是可选的。您可以随时选择。

`statusUpdateNotification` 是一种 HTTP POST。POST 的正文包含列在”表 5-3"的数据元素。

表 **5-3** 状态更新通知 Key

Key	Description

`environment`	指定通知针对的是沙箱技术还是生产环境：
	`SANDBOX`
	`PROD`

`notification_type`	描述触发通知的事件类型。请参见“表 5-4 （第 37 页）”中的值。
`password`	该值与验证收据时您 POST 的共享密钥一致。请参见“与 App Store 验证收据”。
`original_transaction_id`	该值与收据中的“Original Transaction Identifier”一致。您可以使用该值为一位个人顾客的订阅关联多个 iOS 6 风格的交易收据。
`cancellation_date`	Apple 顾客支持取消某个交易的时间和日期。仅在 `notification_type` 为 `CANCEL` 时发起 POST。请参见“表 5-4 （第 37 页）”中的值。
`web_order_line_item_id`	用于识别订阅购买项目的首要 Key。仅在 `notification_type` 为 `CANCEL` 时发起 POST。请参见”表 5-4 （第 37 页）“中的值。
`latest_receipt`	最新续期交易的 base-64 编码交易收据。仅在 `notification_type` 为 `RENEWAL` 或 `INTERACTIVE_RENEWAL` 时发起 POST，且仅在续期成功时发起 POST。
`latest_receipt_info`	最新续期的收据的 JSON 表现形式。仅在续期成功时发起 POST。不在 `notification_type` 为 `CANCEL` 时发起 POST。请参见“表 5-4 （第 37 页）”中的值。
`latest_expired_receipt`	最新续期交易的 base-64 编码交易收据。仅在订阅过期时发起 POST。
`latest_expired_receipt_info`	最新续期交易的收据的 JSON 表现形式。仅在 `notification_type` 为 `RENEWAL` 或 `CANCEL`，或续期失败及订阅过期时发起 POST。

Key	Description
auto_renew_status	这与收据中的自动续期状态相同。另请参见“收据栏”。
auto_renew_adam_id	自动续期订阅的当前续期偏好。这是产品的 Apple ID。
auto_renew_product_id	这与收据中的“Subscription Auto Renew Preference”栏相同。另请参见“收据栏”。
expiration_intent	这与收据中的“Subscription Expiration Intent”相同。仅在 notification_type 为 RENEWAL 或 INTERACTIVE_RENEWAL 时发起 POST。另请参见“收据栏”。

App Store 可能在“表 5-4 （第 37 页）”列出的各个条件下发起 POST 通知。这个列表列出了完整的通知类型和相应订阅事件。

表 **5-4** 状态更新通知类型

notification_type	Description
INITIAL_BUY	订阅的初始购买。在您的服务器上将 latest_receipt 存储为令牌，通过将其与 App Store 验证，以随时验证用户的订阅状态。
CANCEL	订阅已被 Apple 顾客支持取消。查看“Cancellation Date”以了解订阅取消的日期和时间。
RENEWAL	针对过期的订阅的自动续期成功。查看“Subscription Expiration Date□□□□□□□”以确定下一次续期的日期和时间。
INTERACTIVE_RENEWAL	顾客可通过使用您 App 的界面，或账户设置里的“App Store”，在订阅中断后以交互的方式进行续订。订阅服务立即可用。
DID_CHANGE_RENEWAL_-PREFERENCE	顾客更改了计划，将于下个订阅续期生效。当前有效计划不受影响。

若要在一个 statusUpdateNotification POST 中指示成功，您的服务器应发送 HTTP 状态码“200”；您的服务器无需返回数据值。若您的服务器发送 HTTP 码“50x”或“40x”，App Store 将再次尝试发送通知。App Store 会在一段时间内多次重试发送通知，但如果尝试失败过多，最终会停止尝试。

安全要求

发送通知前，App Store 尝试通过“App Transport Security（ATS）（App 传输安全）”建立起和您服务器之间的安全网络连接。关于 ATS 要求的更多信息，请参见“使用 ATS 连接的要求”。如果安全连接无法建立，则不会向您的服务器发送通知。关于安全的更多信息，请参见 https://developer.apple.com/security/。

测试环境的状态更新通知

建议您在生产中实施此逻辑之前，先在测试环境中针对交易测试 `statusUpdateNotification`。

若要确定订阅事件的状态更新通知是否位于测试环境里，请查看 `statusUpdateNotification` 这个 JSON 对象中 `environment` Key 的值是否为 `SANDBOX`。

跨平台注意事项

产品标识符与单个 App 相关联。拥有 iOS 版本和 macOS 版本的 App 在两个平台上拥有各自独立的产品和产品标识符。您可以为在 iOS App 上拥有订阅的用户授予访问 macOS App 中内容的权限（反之亦然），但实现该功能是您的责任。您将需要用于识别用户并跟踪他们的订阅内容的系统，这与您在非续期订阅 App 中所实施的类似。

让用户管理订阅

您的 App 可以打开以下网址（URL），而无需执行您自己的订阅管理 UI：

```
https://buy.itunes.apple.com/WebObjects/MZFinance.woa/wa/manageSubscriptions
```

打开此网址（URL）会启动 iTunes 或 iTunes Store，然后显示“编辑订阅”页面。

测试环境

自动续期订阅行为在测试环境和生产环境中有所不同。

在测试环境中，订阅续期速度加快。 自动续期订阅每天最多续期 6 次。这可以让您测试您的 App 如何处理订阅续期、订阅中断，和包含间隔的订阅历史。请参见 *《iTunes Connect 的 App 内购买项目配置指南》* 中的”测试自动续期订阅“部分，以了解测试的订阅期。

由于过期和续期的速度加快，订阅可能会在系统开始尝试续期订阅之前过期，导致订阅期中出现短暂中断。这样的中断在生产中由于各种原因也有可能发生——请确保您的 App 能妥善处理它们。

恢复已购买产品

用户可以恢复交易以保持他们对已购内容的访问权限。例如，当他们升级到新手机时，所有在旧手机上已经购买的项目不会丢失。请在您的 App 中包含一些机制以便让用户恢复他们的购买，例如一个“恢复购买”按钮。恢复购买会针对用户的 App Store 凭证作出提示，这会中断您 App 的运行：因此，请勿自动恢复购买，尤其不要在您的 App 每次启用的时候。

在大多数情况下，您的App需要做的只是刷新它的收据并交付收据中列出的产品。刷新的收据包含用户在此 App 中的购买记录，无论购买是在此设备上还是其他设备上完成的。但是，一些 App 需要采取其他方法，原因如下：

- 如果您使用 Apple 托管的内容，恢复已完成交易将为您的 App 提供用于下载内容的交易对象
- 如果您需要支持低于 iOS 7 的 iOS 版本（App 收据不可用），请恢复已完成的交易
- 如果您的 App 使用非续期订阅，则您的 App 必须要负责恢复过程

刷新收据会向 App Store 请求最新的收据副本，而不会创建新的交易。连续多次刷新的结果与单次刷新的结果相同，不过您应该避免这样的行为。

恢复已完成的交易将会为用户完成的每一个交易创建一个新的交易，基本上是在重播您的 Transaction Queue Observer（交易队列观察器）的历史记录。当交易被恢复时，您的 App 会维持自身的状态，以跟踪其恢复已完成交易的原因及其处理方式。多次恢复会为每个已完成交易多次创建恢复交易。

> 注意：如果用户尝试再次购买已购买产品，而不是使用您 App 的恢复界面，App Store 将会创建一个常规交易而不是恢复交易。用户不会针对该产品被再次收费。对于此类交易，请用您处理原始交易的方式对其进行处理。

请对用户可以重新下载哪些内容进行适当的控制。例如，请勿允许用户一次下载三年的日报或几百兆字节的游戏关卡。

刷新 App 收据

创建收据刷新请求，设置Delegate（委托），并启动请求。该请求支持可选属性，用于在测试期间获取各种状态的收据，例如过期收据——有关详细信息，请参见 `SKReceiptRefreshRequest` 中 `initWithReceiptProperties:` 方法的值。

```
request = [[SKReceiptRefreshRequest alloc] init];
request.delegate = self;
[request start];
```

在收据刷新后，检查收据并交付所添加的产品。

恢复已完成的交易

您的 App 通过调用 `SKPaymentQueue` 中的 `restoreCompletedTransactions` 方法启动进程。这将会向 App Store 发送一个恢复您 App 已完成的所有交易的请求。如果您的 App 为其支付请求的 `applicationUsername` 属性设置了值（如"监测异常活动 （第 20 页）"中所示），请在恢复交易时使用 `restoreCompletedTransactionsWithApplicationUsername:` 方法提供相同的信息。

App Store 为每个先前完成的交易创建一个新的交易。恢复交易参考了原始交易：`SKPaymentTransaction` 的实例包含 `originalTransaction` 属性，收据中的条目也含有"Original Transaction Identifier"栏。

> 注意：日期栏对于恢复的购买有略微不同的含义。有关详细信息，请参见《*Receipt Validation Programming Guide*（*收据验证编程指南*）》中的"Purchase Date"（购买日期）和"Original Purchase Date"（原始购买日期）栏。

每个恢复交易都要调用 Transaction Queue Observer（交易队列观察器）和 `SKPaymentTransactionStateRestored` 状态，如"等待 App Store 处理交易 （第 22 页）"中所述。您在此处采取的行动取决于您 App 的设计。

- 如果您的 App 使用 App 收据且并不包含 Apple 托管的内容，则不需要此代码，因为您的 App 不会恢复已完成的交易。请立即结束所有的恢复交易
- 如果您的 App 使用 App 收据并且含有 Apple 托管的内容，请让用户在开始恢复前选择恢复哪些产品。在恢复过程中，重新下载用户选择的内容并立即结束其他任何交易

```
NSMutableArray *productIDsToRestore = <# From the user #>;
SKPaymentTransaction *transaction = <# Current transaction #>;

if ([productIDsToRestore containsObject:transaction.transactionIdentifier])
 {
    // Re-download the Apple-hosted content, then finish the transaction
    // and remove the product identifier from the array of product IDs.
```

```
} else {
    [[SKPaymentQueue defaultQueue] finishTransaction:transaction];
}
```

- 如果您的App不使用App收据，它会在所有已完成交易恢复时检查它们。它使用与原始购买逻辑相似的代码路径以使产品可用，然后结束交易

  对于具有多个产品的App，尤其是与内容相关联的产品，请让用户选择恢复哪些产品，而不是一次性恢复所有产品。这些App持续追踪有哪些已完成的交易在恢复时需要被处理，哪些交易可以立即结束以将其忽略。

为“App 审核”做准备

在您完成测试后，便可以提交您的 App 以供审核。本章节重点介绍一些帮助您完成审核过程的技巧。

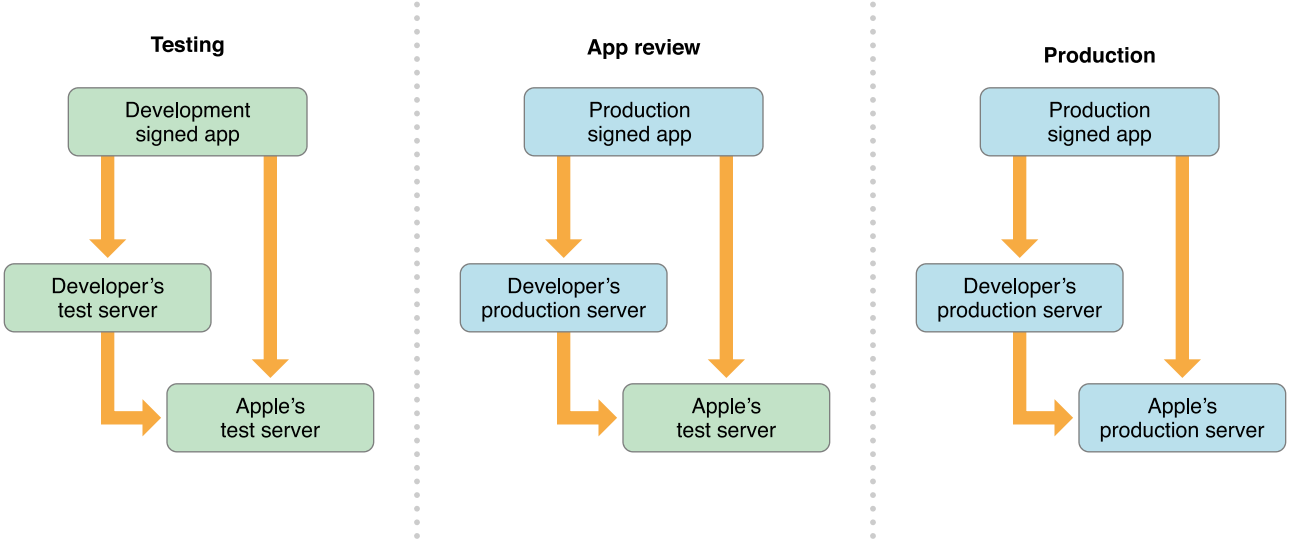
提交产品以供审核

在您第一次提交 App 以供审核时，您还需提交 App 内产品以便同时进行审核。首次提交后，您便可以为您的App和产品提交更新内容，进行各自独立的审核。有关更多信息，请参见《*iTunes Connect 的 App 内购买项目配置指南*》。

测试环境下的收据

如“图 7-1”所示，您的 App 在开发、审核和生产时运行的环境不同。

图 **7-1**　开发、审核和生产环境



在开发过程中，您运行的是登录至开发环境的 App 版本，该版本连接至您的开发服务器和 App Store 测试环境；在生产中，您的用户会运行登录至生产环境的App版本，该版本连接至您的生产服务器和生产环境中的 App Store。但是，在 App 审核期间，您的 App 将在生产／测试混合环境中运行：它登录至生产环境并连接至您的生产服务器，但它也连接 App Store 的测试环境。

当您在服务器上验证收据时，您的服务器需要能够处理登录至生产环境、从 Apple 测试环境中获取收据的 App。建议您的生产服务器始终首先验证生产环境中的 App Store 收据。建议您的生产服务器始终首先验证生产环境中的 App Store 收据。如果验证失败，错误代码为“Sandbox receipt used in production（生产中使用沙箱技术收据）”，则请在测试环境中验证。

实施清单

在您提交 App 以供审核之前，请验证您已实施所有必需操作。确保您已实施以下 App 内购买项目的核心操作（按典型开发过程顺序列出）：

- 在 iTunes Connect 上创建并配置产品

  在整个过程中您都可以更改您的产品，但在测试任何代码前，您需要至少配置 1 个产品。

- 从 App 数据包或您的服务器上获取产品标识符列表。使用 `SKProductsRequest` 的一个实例，将列表发送至 App Store
- 使用 App Store 返回的 `SKProduct` 实例，实现您 App 中的商店的用户界面。开发过程由简单的界面开始，例如一个表格视图或一些按钮

  您可以在开发过程中适时为您 App 中的商店实现最终用户界面。

- 通过向交易队列添加一个 `SKPayment` 实例来请求付款——使用 `SKPaymentQueue` 中的 `addPayment:` 方法
- 由 `paymentQueue:updatedTransactions:` 方法开始，实现一个 Transaction Queue Observer（交易队列观察器）

  您可以在开发过程中适时地使用 `SKPaymentTransactionObserver` 协议中的其他方法。

- 通过为后续启动创建购买的持续记录来交付已购买产品，下载所有相关内容，并最终调用 `SKPaymentQueue` 中的 `finishTransaction:` 方法

  在开发过程中，您可以先实施此代码的简化版——例如，仅在屏幕上显示“Product Delivered（产品已交付）”，然后在开发过程中适时实施真正的版本。

如果您的 App 销售非消耗型项目、自动续期订阅或非续期订阅，请验证您已实现以下恢复逻辑：

- 提供 UI，以便开始恢复过程
- 检索过去购买相关信息——使用 `SKReceiptRefreshRequest` 类刷新 App 收据或使用 `SKPaymentQueue` 类的 `restoreCompletedTransactions` 方法恢复已完成的交易
- 让用户重新下载内容

  如果您使用了 Apple 托管的内容，请恢复完成的交易，并使用交易的 `downloads` 属性，以获得实例 `SKDownload`

  如果您自行管理内容，请对您的服务器发出适当的请求。

如果您的 App 销售自动续期订阅或非续期订阅，请验证您已实施以下订阅逻辑：

- 通过交付最新发布的内容（例如最新一期杂志），以处理新购买的订阅
- 当新内容发布时，将其提供给用户
- 订阅到期时，让用户续订

  如果您的 App 销售自动续期订阅，请让 App Store 处理此过程。不要试图自行处理。

  如果您的 App 销售非续期订阅，您的 App 将负责此过程。
- 当订阅失效时，停止提供新内容。更新您的界面，以便让用户看到再次购买该订阅的选项，以激活订阅
- 实现跟踪内容发布时间的系统。在恢复购买时使用该系统，以便根据订阅激活的时长为用户提供该付费内容的访问权限

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Draft copy. Trademarks go here.

同时在美国和加拿大出版。