**传输平台**

**传送iOS SDK开发人员指南**

**内容**

[第1章简介5](#_bookmark0)

[第2章快速入门6](#_bookmark1)

* 1. [概述6](#_bookmark2)
  2. [将Transmit SDK添加到您的XCode Project6](#_bookmark3) 
     1. [将Non-UI SDK添加到project6](#_bookmark4)
     2. [将UI Layer SDK添加到project7](#_bookmark6)
  3. [添加代码以初始化SDK8](#_bookmark7)
  4. [调用身份验证策略10](#_bookmark8)
  5. [后续步骤10](#_bookmark9)

[第3章为SDK准备应用程序11](#_bookmark10)

* 1. [概述11](#_bookmark11)
  2. [系统要求11](#_bookmark12)
  3. [SDK基本配置11](#_bookmark13) 
     1. [设置SDK钥匙串和数据共享11](#_bookmark14)
     2. [设置与传输服务器的连接12](#_bookmark15)
     3. [添加推送通知支持13](#_bookmark16)
     4. [连接自定义UI处理程序14](#_bookmark17)
     5. [初始化SDK14](#_bookmark18)
  4. [使用SDK插件15](#_bookmark19)

[3.5.2。在运行系统中安装插件](#_bookmark20)

* 1. [高级主题16](#_bookmark21) 
     1. [选择要启用的设备信息收集器16](#_bookmark22)
     2. [配置日志记录17](#_bookmark23)
     3. [配置使用HTTP Cookies18](#_bookmark24)
     4. [配置SSL相互身份验证19](#_bookmark25)
     5. [配置HTTP超时22](#_bookmark26)
     6. [配置自定义传输提供程序23](#_bookmark27)
     7. [使用多个UI处理程序23](#_bookmark28)

[第4章。通用SDK API惯用语25](#_bookmark29)

* 1. [概述25](#_bookmark30)
  2. [SDK实例25](#_bookmark31)
  3. [主要和次要设备会话25](#_bookmark32)
  4. [错误报告26](#_bookmark33)
  5. [穿线28](#_bookmark34)
  6. [异步呼叫28](#_bookmark35)
  7. [客户端上下文30](#_bookmark36)

[第5章调用SDK服务31](#_bookmark37)

* 1. [概述31](#_bookmark38)
  2. [SDK本地状态和用户管理31](#_bookmark39)
  3. [默认UI Handler32](#_bookmark40) 
     1. [设置UI托管Site32](#_bookmark41)
  4. [身份验证，绑定和策略调用32](#_bookmark42) 
     1. [绑定设备33](#_bookmark43)
     2. [调用身份验证策略34](#_bookmark44)
     3. [在现有会话中调用策略34](#_bookmark45)
     4. [登出35](#_bookmark46)
     5. [完整示例：登录或绑定，然后注销35](#_bookmark47)
  5. [管理审批37](#_bookmark48)
  6. [管理设备40](#_bookmark49)
  7. [显示身份验证器配置UI40](#_bookmark50)
  8. [使用TOTP41](#_bookmark51)

[第6章创建UI处理程序43](#_bookmark52)

* 1. [概述43](#_bookmark53)
  2. [客户端上下文和操作上下文43](#_bookmark54)
  3. [根据库存UI44进行定制](#_bookmark55)
  4. [实施自定义UI处理程序46](#_bookmark56) 
     1. [活动指标46](#_bookmark57)
     2. [验证动作47](#_bookmark58)
     3. [实施身份验证器会话接口52](#_bookmark59)
     4. [实施信息和确认处理程序81](#_bookmark60)
     5. [实现JSON数据处理程序82](#_bookmark61)
     6. [处理会话拒绝82](#_bookmark62)
     7. [实施TOTP会话接口83](#_bookmark63)
     8. [实施设备管理会话接口84](#_bookmark64)
     9. [实施批准管理会话界面93](#_bookmark65)
     10. [实施配置菜单会话界面99](#_bookmark66)
     11. [实施促销会话界面106](#_bookmark67)
     12. [策略跟踪回调108](#_bookmark68)
     13. [本地身份验证器无效109](#_bookmark69)
     14. [实施扫描QR码会话界面109](#_bookmark70)
     15. [实施UI处理程序的其他注意事项113](#_bookmark71)

[第7章创建传输提供者114](#_bookmark72)

* 1. [概述114](#_bookmark73)
  2. [实施自定义运输提供商115](#_bookmark75)
  3. [安装自定义传输提供程序116](#_bookmark76)

# 第1章简介

传输安全平台（SP）具有策略实施引擎，该引擎可以实现任意复杂的身份验证策略。 Transmit SDK是Transmit Security Platform的前端接口。 SDK向客户端应用程序公开了一个API，该API允许它调用传输安全平台提供的各种服务。

执行SP服务通常需要与传输服务器进行通信，管理本地设备状态，在本地设备上执行计算并获得用户输入。所有这些任务都由本地设备上的SDK管理和协调。客户端应用程序不必实现它们中的任何一个。从客户端应用程序的角度来看，调用传输服务所需要做的就是调用SDK API调用，并处理该API调用的结果。

尽管SDK无需应用程序开发人员的任何努力就可以管理用户交互，但是通过实现定义良好的接口（SDK调用该接口以请求应用程序获取输入），可以完全自定义用户交互。

代表它。这些界面旨在提供细粒度的自定义选项，使应用程序开发人员可以自定义某些类型的用户交互，而将内置实现用于其他类型。与用户交互类似，SDK还允许客户端应用程序自定义客户端设备与传输服务器之间的通信传输处理。客户端应用程序也通过实现一个简单的界面来完成此操作。

本指南说明了如何将Transmit iOS SDK集成到iOS应用程序中。继介绍之后，第2章从头到尾介绍了集成过程。此过程的目的是使用SDK快速启动和运行应用程序，以此作为进一步配置和自定义的起点。第3章详细讨论了SDK的设置和配置，涵盖了各种SDK初始化

和配置选项。然后，第4章和第5章提供了有关在集成到应用程序后如何实际使用和调用SDK的信息。最后，第6章和第7章介绍了用于自定义SDK操作的各种选项。

尽管本指南涵盖了SDK API的所有主要功能，但它并不试图用作SDK的完整API参考。在单独的文档中提供了完整的SDK API参考以及SDK包，建议在整个SDK集成过程中查阅该文档。

# 第2章快速入门

## 总览

本章介绍了向您的应用程序添加基本SDK支持的分步指南。通过遵循本章中的步骤，您将在应用程序中嵌入一个有效的SDK，并将其用于调用登录策略。

**注意：这并** 不涵盖SDK的所有自定义和配置选项。为简洁起见，省略了一些步骤（例如权限分配）。这意味着在完成此过程后，您将拥有一个可用的SDK，

但禁用了某些功能。请务必参考第2.5节，以确保针对特定应用以最佳方式利用T**ransmit SDK。**

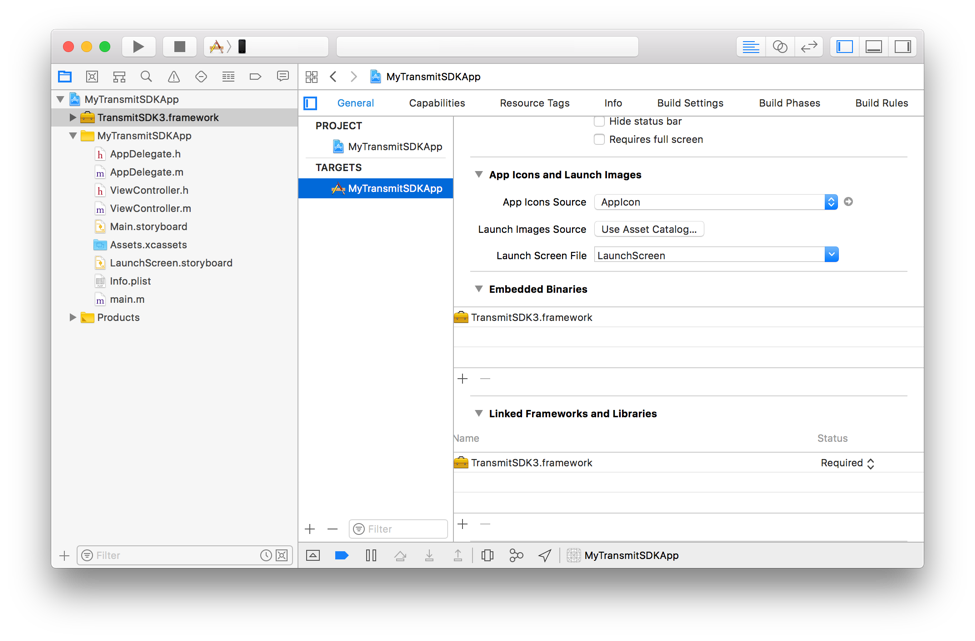
## 将Transmit SDK添加到您的XCode项目

### 将Non-UI SDK添加到项目

集成Non-UI SDK的第一步是将其包含在您的应用程序中。您可以静态或动态链接到SDK。在此示例中，我们正在动态链接。

* + - 1. 在Transmit SDK包中找到文件TransmitSDK3.framework，然后将其复制到XCode项目的根文件夹中。
      2. 将TransmitSDK3.framework添加到XCode中项目的文件列表中。可以通过将框架从Finder拖到项目导航器中来完成。
      3. 将TransmitSDK3.framework添加到要与您的应用程序一起嵌入的框架列表中：
         1. 在XCode中选择项目文件，然后选择应用程序目标。
         2. 在目标设置编辑器中，选择“常规”选项卡。
         3. 通过将框架的文件从项目树拖动到列表中，将框架添加到**Embedded Binaries部**分。
         4. 现在，该框架应同时出现在“嵌入式​​二进制文件”和“链接的框架和库”部分中。如果它没有出**现在列表之一中，请将其拖动**

从项目导航器进入它。请参见图1：将TransmitSDK3.framework添[加到项目中，第7页。](#_bookmark5)



**图1：将TransmitSDK3.framework添加到项目**

### 将UI Layer SDK添加到项目

集成UI层SDK的第一步是将其包含在您的应用程序中。您可以静态或动态链接到SDK。在此示例中，我们正在动态链接。

* + - 1. 在您的Transmit SDK包中找到文件TransmitSDKUILayer.framework，然后将其复制到XCode项目根文件夹中。
      2. 将TransmitSDKUILayer.framework添加到XCode中项目的文件列表中。可以通过将框架从Finder拖到项目导航器中来完成。
      3. 将TransmitSDKUILayer.framework添加到要嵌入到您的应用程序的框架列表中：
         1. 在XCode中选择项目文件，然后选择应用程序目标。
         2. 在目标设置编辑器中，选择“常规”选项卡。
         3. 通过将框架的文件从项目树拖动到列表中，将框架添加到**Embedded Binaries部**分。
         4. 现在，该框架应同时出现在“嵌入式​​二进制文件”和“链接的框架和库”部分中。如果它没有出**现在列表之一中，请将其从项目导航器拖**到其中。

## 添加代码以初始化SDK

您可以在应用程序中多个位置之一中初始化SDK。在此示例中，当通知应用程序启动时，我们将初始化SDK。

1. 找到实现您的应用程序委托类的文件。该文件的默认名称是AppDelegate.m，但在您的应用程序中可能会以不同的方式调用。
2. 向文件中添加一个类似于以下内容的#import指令。应该在文件中已有的任何其他#import指令之后但在任何其他代码之前添加此代码。

#import <TransmitSDK3 / TransmitSDK3.h>

1. 也可以选择模块化导入：

@import TransmitSDK3;

1. 将以下预处理程序宏定义添加到文件中，将其定义值替换为传输服务器的URL，传输服务器中定义的应用程序的应用程序ID和传输服务器中预配置的令牌名称和值（具有访问移动设备的权限）应用程式API。
2. 将以下功能添加到文件中。调用该函数时，它将初始化SDK并为其配置参数以连接到Transmit Server，并将初始化结果异步报告给提供的处理程序。

-（void）initializeTransmitSdk {

//获取SDK共享实例

TSXTransmitSDKXm \* sdk = [TSXTransmitSDKXm sharedInstance];

//配置SDK服务器连接TSXSDKConnectionSettings \* connectionSettings =

[TSXSDKConnectionSettings createWithServerAddress：@“ http://192.168.1.185:8080”

appId：@“ mobile\_app”令牌名称：@“ demo”

令牌：@“ ee1a1455-4425-410b-

a81d-0a40ab2ed16e“]；

[sdk setConnectionSettings：connectionSettings];

//初始化SDK

[SDK initializeAsynchronouslyWithHandler：^（BOOL结果，TSXAuthenticationError

\* authErr）{

如果（authErr）{

NSLog（@“无法初始化SDK（％@）”，authErr.description）；

}

其他{

NSLog（@“ SDK初始化成功”）;

}

}];

}

1. 根据以下示例，调用initailizeTransmitSdk以响应应用程序启动：

-（BOOL）应用程序：（UIApplication \*）应用程序didFinishLaunchingWithOptions：（NSDictionary \*）launchOptions {

// ...

[self initializeTransmitSdk];

// ...

}

例如，这是完整的AppDelegate.m文件，说明了上述步骤的组合：

#import“ AppDelegate.h”

#import <TransmitSDK3 / TransmitSDK3.h> @interface AppDelegate（）

@结束

@implementation AppDelegate

-（void）initializeTransmitSdk {

//获取SDK共享实例

TSXTransmitSDKXm \* sdk = [TSXTransmitSDKXm sharedInstance];

//配置SDK服务器连接TSXSDKConnectionSettings \* connectionSettings =

[TSXSDKConnectionSettings createWithServerAddress：@“ http://192.168.1.208:8080”

appId：@“ mobile\_app”令牌名称：@“ demo”

令牌：@“ ee1a1455-4425-410b-

a81d-0a40ab2ed16e“]；

[sdk setConnectionSettings：connectionSettings];

//初始化SDK

[SDK initializeAsynchronouslyWithHandler：^（BOOL结果，TSXAuthenticationError

\* authErr）{

如果（authErr）{

NSLog（@“无法初始化SDK（％@）”，authErr.description）；

}

其他{

NSLog（@“ SDK初始化成功”）;

}

}];

}

-（BOOL）应用程序：（UIApplication \*）应用程序didFinishLaunchingWithOptions：（NSDictionary \*）launchOptions {

[self initializeTransmitSdk];返回是；

}

@结束

## 调用身份验证策略

SDK初始化后，便可以调用身份验证策略。请注意，使用“发送”，在用户可以使用设备进行身份验证之前，应将设备绑定到用户。因此，登录的通常方式是首先检查设备是否已绑定到用户，然后根据检查结果绑定设备或验证用户身份。通过在代码中添加以下功能并使用用户名进行登录，您可以轻松实现此逻辑：

-（void）loginOrBindForUser：（NSString \*）user {

TSXTransmitSDKXm \* sdk = [TSXTransmitSDKXm sharedInstance];

//检查此设备是否已绑定给用户if（！[sdk isBoundForUserWithUserId：user]）{

[sdk bindWithUserId：user AdditionalParameters：@ {} clientContext：nillyWithWithHandler：^（TSXAuthenticationResult \* \_Nonnull res，

TSXAuthenticationError \* \_Nullable err）{

//处理完成的绑定...

}];

}其他{

[sdk authenticateWithUserId：用户policyId：@“ default\_auth” AdditionalParameters：@ {} clientContext：无

异步lyWithHandler：^（TSXAuthenticationResult \* res，TSXAuthenticationError \* err）{

//处理完成的身份验证...

}];

}

}

一旦将此功能添加到您的应用程序中，只要您想执行登录过程就可以调用它。

## 下一步

恭喜你！您已成功完成Transmit SDK与移动应用程序的基本集成。请执行以下后续步骤，以确保SDK在实际生产环境中的正确运行：

1. 查看第3章，以使SDK设置适应您的需求。本章包含有关此处执行的将SDK嵌入到您的应用程序中的其他步骤的详细信息，例如启用其他SDK所需的权限

功能，从扩展内支持身份验证并扩展日志记录。

1. 阅读第4章和第5章，以了解如何有效地调用不同的SDK服务。其中包括有关主题的信息，例如如何控制应用程序中SDK UI的呈现，如何使用移动批准和设备管理以及如何调用不同类型的策略。
2. SDK提供了全面的自定义选项，可让您更改其UI和通信传输。这些自定义选项在第6章和第7章中介绍。

# 第3章为SDK准备应 用程序

## 总览

本章深入介绍了将SDK集成到应用程序中并使之处于允许应用程序执行功能调用的工作状态的所有必要步骤。

## 系统要求

系统要求

## 基本的SDK配置

以下部分详细介绍了如何初始化和调用SDK。除非另有明确说明，否则以下描述的SDK调用应在应用程序生命周期的早期进行。执行这些操作的一个好地方可能是应用程序委托的application：didFinishLaunchingWithOptions：回调。

**注意：本节** 中描述的所有SDK调用均受第4章中设置的SDK API习惯用法和规则的约束。

### 设置SDK钥匙串和数据共享

Transmit SDK使用钥匙串和文件存储来存储密钥和数据。为了允许在应用程序和扩展或其他应用程序之间共享身份验证数据，SDK应该配置为将数据文件存储在应用程序组中，并将密钥存储在钥匙串组中。 SDK会将这些组用于所有存储。

所有其他将SDK配置为使用相同App Group和Keychain Group的应用程序或扩展，都将能够与该应用程序共享身份验证数据。要指示SDK使用共享组，请调用+ [TSXTransmitSDKXm shareDataWithAppGroup：keychainGroup：]。必须先调用此调用

完成对SDK的任何其他调用，尤其是在尝试获取SDK之前

通过+ [TSXTransmitSDKXm sharedInstance]共享的SDK实例。代码

以下示例演示了如何调用此调用以使用＃unique\_19中图中显示的组。

[TSXTransmitSDKXm shareDataWithAppGroup：@“ group.com.acme.MyTransmitSDKApp”钥匙串组：@“ C88675XXXX.com.acme.MyTransmitSDKApp.TransmitSDK”]；

### 设置与发送服务器的连接

为了连接到传输服务器，SDK要求可以访问该服务器的URL以及在该服务器上配置的API令牌。此外，SDK代表应用程序与服务器通信，并且需要使用服务器上配置的此应用程序的标识符对其进行配置。 SDK中的连接信息由TSXSDKConnectionSettings类表示。此类封装以下属性：

**描述**

可以通过其访问服务器API端点的URL。这应该是将请求传递到服务器根路径的端点。

SDK应连接到的指定服务器内的领域的名称。如果这是一个空字符串，则SDK将连接到默认领域。

传输安全平台服务器中定义的应用程序标识符。

在Transmit Security Platform服务器中为应用程序配置的令牌的名称。

在Transmit Security Platform服务器中为应用程序配置的令牌的值。

数据传输加密模式将用于新身份验证会话的任何连接安全性（例如SSL）之上。

**属性**

服务器地址

领域

appId tokenName令牌cryptoMode

要构造指定这些属性的实例，请使用+ [TSXSDKConnection-设置createWithServerAddress：appId：tokenName：token：]调用。

构造连接设置对象后，可以通过将其传递给-[TSXTransmitSDKXm setConnectionSettings：]来配置SDK。以下代码段显示了此过程的示例：

TSXSDKConnectionSettings \* connectionSettings = [TSXSDKConnectionSettings

createWithCryptoModeWithServerAddress：@“ http://192.168.1.103:8080” appId：@“ mobile\_app”

tokenName：@“演示”

令牌：@“ ee1a1455-4425-410b-a81d-0a40ab2ed16e”

cryptoMode：ConnectionCryptoModeCredentials];

[[TSXTransmitSDKXm sharedInstance] setConnectionSettings：connectionSettings];

* + 1. **数据加密（端到端加密）**

与Transmit Server的连接以常规JSON对象的形式携带数据，并使用TransportProvider对象（请参见第7章）来实现，该对象可以由托管应用程序自定义。这可能允许托管应用程序

在需要时检查要传输的数据的内容。 SDK API提供了加密该数据的功能，而与连接上部署的SSL无关，这意味着即使使用SSL保护了连接，它所携带的数据也可能已经被加密，因此可以“两次”或使用“双重”加密进行保护。 ，这也意味着托管无法检查数据（一旦加密）

应用。通过SDK和传输服务器之间的握手以下面描述的方式来设置加密。如上所述，影响此握手流程的加密模式可以设置为以下值之一：

1. ConnectionCryptoMode.None：传输的数据未加密（部署SSL仍可保护连接）。
2. ConnectionCryptoMode.Credentials：加密握手将在每个用户会话开始时对传输的数据执行（以小猪为后盾），并且加密将在握手完成后立即开始。
3. ConnectionCryptoMode.Full：加密握手将在任何数据传输之前执行，并且加密将在握手完成后立即开始。这意味着需要额外的数据交换往返，以执行加密握手。

**注意：如果** 连接设置为与传输服务器上的应用程序设置的数据传输加密模式不匹配的数据传输加密模式，则连接将失败并且SDK将向托管应用程序报告错误。

### 添加推送通知支持

传输安全平台支持将身份验证批准请求发送到移动设备。根据服务器配置，这些请求可能会附带发送到应用程序的触发通知，以触发批准处理。对于

为了能够向应用程序发送推送通知，平台需要知道可以用于在设备上寻址应用程序的推送通知令牌。如果服务器配置为使用APNS传递推送通知，则此令牌将为APNS推送令牌；否则，此令牌将为APNS推送令牌。否则，后端推送提供商需要使用任何令牌来标识设备上的应用程序。

通过SDK将设备的推送令牌提供给平台。应用程序在本地获取推送通知令牌后，应调用SDK的-[-TSXTransmitSDKXm setPushToken：]调用。此调用应通过包含令牌值的十六进制表示形式的st环传递。对于基于APNS的推送方法，通常是通过应用程序委托的application：didRegisterForRemoteNotificationsWithDeviceToken：回调来完成，并将令牌的编码作为十六进制字符串传递。

以下示例演示了如何获取基于APNS的推送通知令牌并将其传递给SDK。

-（void）应用程序：（UIApplication \*）应用程序didRegisterForRemoteNotificationsWithDeviceToken：（NSData \*）deviceToken {

NSString \* tokenStr = [deviceToken描述];

tokenStr = [tokenStr stringByTrimmingCharactersInSet：[NSCharacterSet characterSetWithCharactersInString：@“ <>”]]];

tokenStr = [tokenStr stringByReplacingOccurrencesOfString：@“” withString：@“”]]; [[TSXTransmitSDKXm sharedInstance] setPushToken：tokenStr];

}

一旦应用程序收到推送通知，它应该调用SDK以便在它是移动批准请求的情况下对其进行处理。在第5.5节中讨论了启动移动批准流程。

### 连接自定义UI处理程序

如果要实现自己的UI处理程序以在整个登录过程中提供自定义的用户体验（请参阅第6章），则应在调用需要在SDK上使用它的调用之前提供该自定义UI处理程序。可以通过调用SDK方法-[TSXTransmitSDKXm setUiHandler：]来完成，如以下示例所示：

TSXSampleAppUIHandler \* uiHandle r = [[TSXSampleAppUIHandler alloc] init];

//初始化处理程序

[[TSXTransmitSDKXm sharedInstance] setUiHandler：uiHandler];

只要您不设置自己的UI处理程序，SDK就会使用Transmit提供的类TSXDefaultUIHandler的UI处理程序。有关在整个应用程序运行时更改UI处理程序的讨论，另请参见3.6.7节。

### 初始化SDK

在SDK上进行任何功能调用之前，应用程序必须调用SDK方法-[TSXTransmitSDKXm initializeAsynchronouslyWithHandler：]。这是执行SDK初始化的异步方法。只有此方法异步完成后，应用程序才可以调用第5章中介绍的功能性SDK调用。应用程序实现必须保证，在此函数异步完成之前，对SDK的所有调用都将被延迟或跳过。

以下代码片段说明了如何进行初始化调用。

[[TSXTransmitSDKXm sharedInstance] initializeAsynchronouslyWithHandler：^（BOOL结果，TSXAuthenticationError

\* authErr）{

如果（authErr）{

NSLog（@“无法初始化SDK（％@）”，authErr.debugDescription）;

}

其他{

NSLog（@“ SDK初始化成功”）;

}

}];

## 使用SDK插件

SDK插件是Transmit SDK加载的扩展其功能的组件。扩展功能可能包括对其他操作，身份验证器，风险引擎等的支持。要了解如何使用特定的插件及其提供的功能，请参阅该插件的文档。

在您的应用程序中包含任何插件都涉及以下步骤：

* + 1. **将插件工件添加到您的应用程序。插件工件作为插件包的一部分提供，应包含在您的应用程序中，以便T**ransmit SDK可以加载它们。
    2. **在运行时安装插件。在通过调用-[-TSXTransmitSDK**Xm initializeAsynchronouslyWithHandler：]初始化SDK之前，应用程序应使用-[TSXTransmitSDKXm install- PluginWithPluginName：config：]来安装插件。

下面将详细介绍这两个步骤。

除了这些步骤之外，每个插件都可以从应用程序中提供其自己的扩展API和设置要求。有关这些的更多信息，请参考插件文档。

### 3.5.2。在运行时安装插件

通过将插件包含在应用程序中使插件对SDK可用后，就该明确指示SDK运行时加载并初始化插件了。这称为“安装插件”。插件安装必须在初始化SDK之前进行，即在调用方法-[TSXTransmitSDKXm initializeAsynchronouslyWithHandler：]之前。

**注意：在应** 用程序中包含插件的工件而不安装插件不会对应用程序产生任何影响。要利用插件的扩展功能，必须在运行时安装它。

通过调用-[TSXTransmitSDKXm installPluginWith- PluginName：config：]方法来安装插件。此方法接收要安装的插件的名称，以及包含用于插件设置的参数的配置对象。在插件文档中定义了为此调用提供的插件名称以及参数对象的结构。

**注意：如果** 应用程序尝试在不包含匹配工件的情况下安装插件，则SDK在-[TSXTransmit- SDKXm initializeAsynchronouslyWithHandler：]调用期间将失败。这是AuthenticationErrorCodeAppImplementation错误。但是，在插件初始化期间失败将无提示地不安装插件并继续执行应用程序。

以下示例演示了如何在运行时中安装插件：

由于插件安装是在运行时进行的，因此应用程序可以在初始化SDK之前应用任何所需的逻辑来决定是否应安装插件。

## 进阶主题

### 选择要启用的设备信息收集器

作为设备识别和设备风险评估的一部分，Transmit SDK收集各种设备属性并将其发送到Transmit服务器。应用程序可以在初始化时间上控制哪些属性将被收集，哪些属性将不被收集。这允许应用程序防止由于例如隐私问题而收集某些数据。要选择将收集哪些数据并将其发送到服务器，请使用-[TSXTransmit- SDKXm setEnabledCollectors：]调用，并提供TSXCollectorType枚举值的组合。默认情况下，所有收集器都是启用的，因此，除非调用-[-TSXTransmitSDKXm setEnabledCollectors：]，否则所有收集器都将保留

已启用。有关可用收集器值及其含义的完整列表，请参阅SDK参考指南。下面的代码说明了此方法的调用。

[[TSXTransmitSDKXm sharedInstance] setEnabledCollectors：（（CollectorTypeDeviceDetails |

CollectorTypeContacts | CollectorTypeOwner）]；

**注意：以下** 收集器是必需的，因为它们收集服务器决策所需的基本数据：

* + - * CollectorType.DeviceDetails，
      * CollectorType.ExternalSDKDetails，
      * CollectorType.HWAuthenticators，
      * CollectorType.LocalEnrollments，
      * CollectorType.Capabilities

因此，如果使用不同的收集器集调用-[TSXTransmitSDKXm setEnabledCollectors：]，则以下列表将保持启用状态。

必须在SDK初始化之前进行-[TSXTransmitSDKXm setEnabledCollectors：]调用。如果应用程序未调用该调用，则将收集最大数量的信息。

**注意：启用** 或禁用收集器可能需要在项目设置过程中执行特定步骤。有关特定收集器所需设置的信息，请参考＃unique\_30。

### 配置日志

Transmit SDK在运行时发出各种日志消息，以提供诊断信息以进行故障排除。默认情况下，SDK使用平台的标准日志记录工具通过内部记录器发出所有日志。该应用程序可以控制SDK日志记录的两个方面：设置日志记录的详细程度以及可能设置外部记录器。

设置日志记录的详细程度仅影响内部SDK的记录器。设置外部记录器会将附加的应用程序定义的记录目标添加到现有内部记录器。

记录详细程度是通过调用SDK的-[TSXTransmitSDKXm set- LogLevel：]方法来控制的。此方法接收日志记录级别作为TSXLogLevel枚举中的值之一。请查阅API参考以获取可用日志记录级别及其含义的列表。

要设置外部记录器，应用程序应通过调用-[TSXTransmitSDKXm set- ExternalLogger：]调用来提供从TSXTransmitSDKLogger类派生的对象。提供给此调用的对象必须实现方法-[TSXTransmitSDKLogger logWithLevel：category：message：]，以接收日志消息并按照应用程序开发人员的意愿对其进行处理。 API参考指南中提供了有关类和此方法的完整说明。

以下代码片段说明了在SDK上设置日志记录详细级别和设置外部记录器的过程。

@interface AppLogger：TSXTransmitSDKLogger {} @end

@implementation AppLogger

-（void）logWithLevel：（TSXLogLevel）级别类别：（NSString \*）类别消息：（NSString \*）消息{

NSLog（@“％d％@％@”，（int）级别，类别，消息）；

}

@结束

/// ....

/// ....

TSXTransmitSDKXm \* sdk = [TSXTransmitSDKXm sharedInstance]; [SDK setLogLevel：LogLevelDebug];

[SDK setExternalLogger：[[AppLogger alloc] init]];

### 配置HTTP Cookie的使用

SDK随附的标准HTTP传输提供程序（TSXHTTPTransportProvider）默认情况下使用非持久性隔离存储来保存在与Transmit端点进行通信期间收到的HTTP cookie。由于商店是非持久性的，在与“传输”端点交换期间收到的cookie不会保存到应用程序生命周期之外。 Cookie存储是隔离的事实意味着它仅包含在SDK与服务器通信期间交换的cookie。

应用程序未通过SDK执行的任何HTTP通信都不会将cookie添加到隔离存储中。例如，如果应用程序和SDK都与同一个域myapp.domain.com通信，并且在应用程序发出的请求期间收到了该域的cookie，则当SDK出现时，该cookie不会发送到服务器发出请求-因为cookie没有保存在HTTP传输提供程序的隔离cookie存储中。

应用程序可能会提供备用cookie存储，以供运输提供商使用。例如，平台共享cookie存储可用于在以下站点之间共享cookie

应用启动和SDK启动的调用。标准的HTTP传输提供程序可以使用从NSHTTPCookieStorage派生的任何对象作为cookie存储。

为了指示默认的传输提供程序使用特定的cookie存储，应用程序应构造自己的TSXHTTPTransportProvider实例，将该实例的TSXHTTPTransportProvider.cookieStorage属性设置为派生自NSHTTPCookieStorage的对象，然后按如下所述将该实例提供给SDK。第3.6.6节。

**注意：此处** 所述的默认传输提供程序的配置必须在首次尝试使用它之前进行。

在下面的示例代码中，传输提供程序配置为使用共享cookie存储。

...

-（BOOL）应用程序：（UIApplication \*）应用程序didFinishLaunchingWithOptions：（NSDictionary \*）launchOptions

{

...

TSXTransmitSDKXm \* sdk = [TSXTransmitSDKXm sharedInstance];

...

TSXHTTPTransportProvider \* transport = [[[TSXHTTPTransportProvider alloc] init]; transport.cookieStorage = [NSHTTPCookieStorage sahredHTTPCookieStorage];

[SDK setTransportProvider：transport];

...

}

...

### 配置SSL相互认证

如以下各节所述，可以通过提供客户端证书或配置公共密钥来配置SSL相互认证。

* + - 1. **客户端证书**

如果应用程序要求SDK通过SSL与传输服务器通信时，SDK将提供客户端证书，则可以通过调用+ [TSXHTTPTransportProvider setAuthChallengeDelegate：]向SDK提供一个实现TSXConnectionAuthenticationChallengeDelegate协议的对象来实现。进行此调用后，SDK将在需要客户端证书时调用TSXConnectionAuthenticationChallengeDelegate-实现对象的clientCredential方法。

**注意：对S** SL相互身份验证的支持是作为SDK随附的标准HTTP传输提供程序的一部分实现的

（TSXHTTPTransportProvider）。如果应用程序提供了自定义传输提供程序（第3.6.6节），则除非将功能委托给标准提供程序，否则这些设置对它没有影响。

在下面的示例代码中，将从myCert.p12受密码保护的文件中读取证书，然后将其返回给SDK。

@interface AppDelegate（）<TSXConnectionAuthenticationChallengeDelegate> @end

@implementation AppDelegate

//实现TSConnectionAuthenticationChallengeDelegate协议

-（NSURLCredential \*）clientCredential

{

NSString \*密码= @“ mySecret123passwoRd！”；

NSData \* p12Data = [NSData dataWithContentsOfFile：[[NSBundle mainBundle] pathForResource：@“ myCert” ofType：@“ p12”]]；

如果（！p12Data）返回nil; CFArrayRef keyref = NULL;

OSStatus状态= SecPKCS12Import（（网桥CFDataRef）p12Data，

（bridge CFDictionaryRef）[NSDictionary

dictionaryWithObject：password forKey :(网桥ID）kSecImportExportPassphrase]，＆keyref）;

如果（状态！= noErr）{

NSLog（@“导入pkcs12 [％d]时出错”，状态）；返回零；

}

CFDictionaryRef identityDict = CFArrayGetValueAtIndex（keyref，0）; SecIdentityRef identityApp =

（SecIdentityRef）CFDictionaryGetValue（identityDict，kSecImportItemIdentity）;

SecCertificateRef certRef; SecIdentityCopyCertificate（identityApp，＆certRef）;

SecCertificateRef certArray [1] = {certRef};

CFArrayRef myCerts = CFArrayCreate（NULL，（void \*）certArray，1，NULL）; CFRelease（certRef）;

NSURLCredential \* myCredential = [NSURLCredential credentialWithIdentity：identityApp证书：( bridge NSArray \*）myCerts持久性：NSURLCredentialPersistencePermanent]；

CFRelease（myCerts）;

返回myCredential;

}

//为TransmitSDK初始化分配一个委托

-（BOOL）应用程序：（UIApplication \*）应用程序didFinishLaunchingWithOptions：（NSDictionary \*）launchOptions

{

...

[TSXHTTPTransportProvider setAuthChallengeDelegate：self];

...

}

@结束

* + - 1. **公钥固定**

SDK附带的标准HTTP传输提供程序（TSXHTTPTransportProvider）允许配置SSL公钥固定的部署。要使用此功能，应用程序应构造自己的TSXHTTPTransportProvider实例，并为它提供域的字典以及使用TSPinnedDomainConfiguration类提供的它们自己的公钥配置。

**注意：安全 警告-滥用此配置选项可能会使您的应用程**序容易受到攻击，因为它绕过了正常的操作系统信任库。它用于企业场景，其中公司可能正在运行自己的内部生产级证书颁发机构以进行调试。

**TSX SSL域配置**

使用initWithPublicHashes：（NSArray）\*哈希和SupportedAlgorithms：（TSXSupportedAlgorithms）算法来初始化TSXSSLDomainConfiguration实例。参数哈希是SSL引脚的数组，其中每个引脚是证书的主题公钥信息的base64编码的SHA-256哈希。参数算法是公钥支持的算法的字节集。 ：

**描述**

RSA 2048

RSA 4096

带有secp256r1曲线的ECDSA带有secp384r1曲线的CDSA

**名称TSX**AlgorithmRsa2048 TSXAlgorithmRsa4096

TSXAlgorithmEcDsaSecp256r1

TSXAlgorithmEcDsaSecp384r1

域配置的其他可选属性：

**属性**

**描述**

forcePinning如果 设置为“ NO”，则SDK不会阻止SSL连接导致的密码或证书验证错误。默认值为“是”

includeSubdomains 如果设置为“ YES”，则还固定指定域的所有子域。默认值为“ NO”

excludeSubdomainFromParentPolicy如果设置为YES，则SDK不会将其固定

如果为该域的父域设置了includeSubdomains，则为特定域。这允许从应用于父域的固定策略中排除特定的子域。

expirationDate 一个过期日期，域的配置的SSL引脚到期，从而禁用固定验证。如果未设置密钥，则引脚不会失效。到期有助于防止应用程序中的连接问题（如用户禁用应用程序更新时）出现问题，这些问题不会对其引脚组进行更新。

reportUris 一个URL数组，应向其报告引脚验证失败。

在下面的示例代码中，使用特定的SSL公钥固定配置初始化了传输提供程序：

...

-（BOOL）应用程序：（UIApplication \*）应用程序didFinishLaunchingWithOptions：（NSDictionary \*）launchOptions

{

...

TSXTransmitSDKXm \* sdk = [TSXTransmitSDKXm sharedInstance];

...

TSXHTTPTransportProvider \* provider = [[TSXHTTPTransportProvider alloc] init];

TSXSSLDomainConfiguration \* config = [[TSXSSLDomainConfiguration alloc] initWithPublicHashes：@ [@“ HXXQgKOueCIU5TTLHob / bPbwcKOKw6DkfsTWYHbxbqTY =”，

@“ 0SDf3cRToyZJaMsoS17oF72VMavLxj / N7WBNasNuiR8 =”] andSupportedAlgorithms：TSXAlgorithmRsa2048];

provider.publicKeyPinningconfig = @ {@“ domain.com”：config}; [SDK setTransportProvider：provider];

...

}

...

### 配置HTTP超时

SDK附带的标准HTTP传输提供程序（TSXHTTPTransportProvider）允许配置HTTP超时间隔

它发出的请求。要使用此功能，应用程序应构造自己的TSXHTTPTransportProvider实例，使用超时值对其进行配置，然后如3.6.6节中所述将此实例提供给SDK。

**注意：此处** 所述的默认传输提供程序的配置必须在首次尝试使用它之前进行。

在下面的示例代码中，传输提供程序配置为在10秒后超时。

...

-（BOOL）应用程序：（UIApplication \*）应用程序didFinishLaunchingWithOptions：（NSDictionary \*）launchOptions

{

...

TSXTransmitSDKXm \* sdk = [TSXTransmitSDKXm sharedInstance];

...

TSXHTTPTransportProvider \* transport = [[[TSXHTTPTransportProvider alloc] init]; transport.timeoutInterval = 10;

[SDK setTransportProvider：transport];

...

}

...

### 配置自定义传输提供程序

该SDK允许应用程序通过提供自己的自定义通信传输提供程序来控制与Transmit Server的通信。每当将消息发送到服务器时，SDK都会调用传输提供程序，并且负责将请求中继到服务器，并将响应从服务器中继回SDK。

第7章介绍了实现自定义传输提供程序。如果应用程序希望SDK使用已实现的自定义传输提供程序，则它必须通过调用SDK调用-[TSXTransmitSDKXm setTransportProvider：]向应用程序提供传输提供程序。有关使用此调用的示例，请参见第7章。

### 使用多个UI处理程序

允许应用程序在整个应用程序生命周期内替换SDK使用的TSXUIHandler实例。如果应用程序希望对不同的身份验证流使用不同的UI实现（例如，对于主要身份验证和辅助身份验证使用不同的UI），这将很有用。为此，应用程序应在UI处理程序每​​次调用-[TSXTransmitSDKXm setUiHandler：]

应该更换。请注意，对于任何给定的SDK调用（例如策略调用），在整个调用执行期间都将使用相同的UI处理程序。当前设置的TSXUIHandler仅在开始新的调用时使用。例如，即使在策略评估期间应用程序使用新的UI处理程序调用-[TSXTransmitSDKXm setUi-Handler：]，新的UI处理程序也将仅用于后续的SDK调用，并且可以确保应用程序确保所有与第一个策略将分派到原始UI处理程序。

# 第4章常规SDK A PI惯用法

## 总览

Transmit SDK向客户端应用程序公开了广泛的API。为了简化SDK的使用，在整个API定义和SDK中都保留了一些API习惯用法

定义了调用规则。本章介绍了这些习语和规则。除非在其定义中另有明确说明，否则所有API均应遵守此处设置的准则。

## SDK实例

大多数SDK API入口点调用都定义为TSXTransmitSDKXm类的一部分。要调用这些调用，应用程序首先需要获取此类的实例。所有SDK配置设置（包括服务器连接，UI处理程序，记录器和传输提供程序）都绑定到该实例。该实例有时称为“ SDK实例”。

SDK在整个应用程序生命周期中都维护TSXTransmitSDKXm类的单个实例，并且该实例可通过调用[TSXTransmitSDKXm sharedInstance]进行访问。该应用程序不得创建其他SDK实例。此外，它不得尝试控制-[TSXTransmitSDKXm sharedInstance]返回的实例的生命周期。

## 主要和次要设备会话

一个SDK实例最多与传输服务器维护一个主要设备会话。设备会话绑定到单个用户和**设备，并且包括由在该会话上进行的各种调用驱动的**状态。例如，状态包括有关在会话中执行的最后一次认证的信息。 SDK的所有未明确声明的功能性API都会在SDK的主要设备会话上起作用。

建立主设备会话是对以下功能之一的调用的一部分：

* + - -[TSXTransmitSDKXm bindWithUserId：AdditionalParameters：clientContext：异步地与处理程序：]
    - -[TSXTransmitSDKXm authenticateWithUserId：policyId：additionalParameters：clientContext：asynchronousWith-Handler：]

在调用-[TSXTransmitSDKXm logoutAsynchronouslyWithHandler：]的过程中，主设备会话被拆除。

某些API调用可能会在不是当前主会话的会话上运行。这些API调用会在调用时创建一个单独的设备会话，并在执行完毕后将其断开。它们不会影响保持不变的主设备会话。以下API调用可能会创建自己的会话，而不是在主设备会话上运行：

* + - -[TSXTransmitSDKXm startAuthenticationConfigurationWithToken- WithConfigurationToken：clientContext：异步地-处理程序：]
    - -[TSXTransmitSDKXm startApprovalsSessionForPushedRequestWith- PushRequestPayload：clientContext：ynchronousWithHandler：]

## 错误报告

SDK报告的所有错误均由TSXAuthenticationError表示

宾语。该对象包含一个错误代码，文本消息，任意其他数据和可选的公共属性，每个属性都与一个特定的错误代码相关。

TSXAuthenticationError对象从API返回或作为异步调用失败返回（请参见4.6节）。错误对象包含以下成员，用于提供有关错误的信息：

**表格1：**

|  |  |
| --- | --- |
| **成员名字** | **描述** |
| -[TSX身份验证错误获取-错误代码] | 此错误对象的错误代码。这是TSXAuthenticationErrorCode值之一。 |
| -[TSXAuthenticationError get-消息] | 人类可读的消息，详细说明有关错误的信息。 |
| -[TSX身份验证错误获取-数据] | 以结构化方式详细描述有关错误的信息的其他数据。依赖于此字段的值，请参见以下说明 |

应用程序可以假定-[TSXAuthentication- Error getErrorCode]返回的错误代码是SDK API的一部分。这样，可以保证在特定情况下此errorCode的值保持固定。除非另有明确说明，否则TSXAuthenticationError对象的所有其他属性的值可能会在SDK版本之间更改，除提供错误信息之类的非规范性信息外，不得将其视为接口的一部分。

感兴趣的两个特殊的errorCode值是AuthenticationErrorCodeInternal和AuthenticationErrorCodeAppImplementation。前者以非预期的方式传达SDK内部的故障。这可能是低级错误，也可能是数据损坏或错误的结果。当SDK识别出最有可能是由于应用程序实现中的错误引起的情况时，将标记后者。例如– SDK调用顺序混乱或参数无效。

* 1. **错误的公共属性**

对于某些特定的错误代码，TSXAuthenticationError对象带有附加的可访问属性，该属性通过附加描述或与特定错误条件有关的数据来增强错误代码。例如，对于与设备上生物特征配置有关的不同生物特征认证失败，可能会报告AuthenticationError- CodeAuthenticatorExternalConfigError（由于指纹未在设备上注册或指纹OS定位失败）

太多失败的身份验证尝试。为了使托管应用程序能够区分这些错误情况，当其携带AuthenticationErrorCodeAuthenticatorExternalConfigError时，其代码可以查询TSXAuthenticationError对象上存在的公共属性的值。可通过使用标识TSXAuthenticationErrorProperty定义的属性的键通过类型不同的getter方法（如下表所述）从TSXAuthenticationError对象检索属性。下表描述了这些getter方法：

**表2：**

|  |  |
| --- | --- |
| **成员名字** | **描述** |
| -[TSXAuthenticationError get- PublicStringPropertyWith-属性：] | 根据以下项的键值检索字符**串属性**  TSXAuthenticationErrorProperty文字，如果不存在这样的属性，则将其作为参数或“ null”传递。如果要检索的属性不是字符串类型，则抛出TSXAuthenticationError。 |
| -[TSXAuthenticationError get- PublicNumberPropertyWith-属性：] | 根据键值中的on检索数字**属性**  TSXAuthenticationErrorProperty文字，如果不存在这样的属性，则将其作为参数或“ null”传递。如果要检索的属性不是数字类型，则抛出TSXAuthenticationError。 |

|  |  |
| --- | --- |
| **成员名字** | **描述** |
| -[TSXAuthenticationError get- PublicBooleanPropertyWith-属性：] | 根据以下项的键值检索布尔**属性**  TSXAuthenticationErrorProperty文字，如果不存在这样的属性，则将其作为参数或“ null”传递。如果要检索的属性不是布尔类型，则抛出TSXAuthenticationError。 |
| -[TSXAuthenticationError get- PublicSymbolicPropertyWith-属性：] | 根据TSXAuthent**icationErrorProperty文字中的键值，检索TSXAuthent**icationErrorPropertySymbol属性，如果没有这样的属性，则将其作为参数或“ null”传递。如果要检索的属性不是TSXAuthenticationErrorPropertySymbol类型，则抛出TSXAuthenticationError。 |

## 穿线

所有SDK调用都必须在应用程序的主线程上进行。 SDK对应用程序提供的回调进行的所有回调调用都将在应用程序的主线程上发生。对于定制接口，每当应用程序提供的方法返回异步结果时，异步结果通知调用都必须在主线程上进行。请注意，尽管所有调用都必须在主线程上进行

在该应用程序中，所有SDK长时间运行的调用均实现为异步调用（请参见4.6节），因此它们不会造成任何UI延迟或延迟。 SDK可能会在运行时生成多个线程供其自己使用。

## 异步呼叫

在整个SDK API中，某些方法被定义为异步方法。异步方法可以由SDK或应用程序代码实现（尤其是在实现自定义接口时）。所有异步方法都遵循相同的定义模式：

* + 1. 他们的Objective-C方法名称的最后一个子句始终为ynchronousWithWithHandler:。此子句后面的参数始终是异步处理程序（请参见下文）。例如：

bindWithUserId：additionalParameters：clientContext：asynchronouslyWithHand

* + 1. 对于除了异步处理程序之外不接收任何参数的方法，方法名称以AsynchronouslyWithHandler：结尾，后跟异步处理程序参数。例如，initializeAsynchronouslyWithHandler:。

异步方法的异步处理程序参数始终是块类型。该块将在完成时由异步方法调用（成功或有错误）。块的第一个（有时也是唯一的）参数始终是异步调用的结果。如果定义了第二个参数，它将

如果呼叫失败，请包含错误信息。此错误将作为TSXAuthenticationError类型的对象传达。以下代码显示了一个简单的无参数异步方法的示例调用：

[[TSXTransmitSDKXm sharedInstance] initializeAsynchronouslyWithHandler：^（BOOL结果，TSXAuthenticationError

\* authErr）{

如果（authErr）{

NSLog（@“无法初始化SDK（％@）”，authErr.debugDescription）;

}

其他{

//可以看结果；包含有效的响应NSLog（@“ SDK初始化成功”）;

}

}];

在为异步调用实现处理程序块时，请注意以下事项：

1. 如果块接收到错误参数，请始终先检查错误参数。如果有错误传递给块，请忽略result参数。
2. 异步处理程序是在主线程上调用的，因此避免对它们执行长时间运行的操作。
3. 永远不要从异步处理程序中引发异常。
4. 除另有明确说明的情况外，请勿从异步处理程序内调用SDK API调用。

实施异步调用时，您有责任调用传递给该调用的块来报告错误或成功结果。异步调用的实现必须遵循以下准则：

1. 永远不要多次调用传递给异步调用的处理程序。应该特别注意由于UI操作（例如，用户按下按钮）而调用的异步调用处理程序。在这些情况下，必须确保防止用户执行多个用户界面操作，这些操作将导致同一处理程序的多次调用。例如，如果使用“提交”按钮通过调用处理程序来完成异步调用，则最好禁用该按钮

作为事件处理的一部分，以防止用户在事件处理完成之前再次单击该按钮。请注意，SDK将检测对完成处理程序的两次调用，并在发生两次时生成严重错误，详细说明错误的确切原因。

1. 异步调用必须通过使用error参数调用异步处理程序来报告所有错误。他们绝不能通过报告异常或任何其他方式来表示错误。

在本手册中，我们可以将传递给异步处理程序的结果参数的值称为“异步返回值”。我们指的是

用异步返回值作为“异步返回”调用异步处理程序的过程。同样，“异步失败”是指使用错误参数调用异步处理程序。

## 客户端上下文

所有可能触发用户交互的SDK API调用（即，可以在UI Handler上调用）都收到一个名为clientContext的客户端上下文参数。这是一个NSDictionary对象，其中包含供UI Handler实现使用的值，将应用程序要传达的任何信息传达给调用站点的UI。使用客户端上下文，应用程序可以配置特定于特定调用的UI实现的各个方面。执行一个

UI处理程序和将使用此UI处理程序的SDK调用的调用站点当然应该协调此客户端上下文对象的结构。

SDK提供的默认UI Handler实现为它希望找到的此客户端上下文定义了特定的结构。这在第5.3节中进行了描述。

# 第5章调用SDK服务

## 总览

本章提供有关如何调用Transmit SDK提供的各种功能服务的说明。它假定成功实施了第3章中介绍的集成步骤，并且充分熟悉了第4章中描述的惯用语和概念。本章的目的不是提供对SDK调用这些服务所提供的API的完整描述。而是应将其用作完整API参考文档的起点和指针。

## SDK本地状态和用户管理

SDK会在设备上本地维护状态。此状态主要存储在文件系统和平台密钥存储中。如第3.4.1节所述，状态可以存储并与其他应用程序或扩展共享。

SDK的状态模型按用户句柄在顶层进行分区：SDK维护的所有数据均绑定到特定用户。用户由标识符标识；这是传递给登录和绑定调用的同一用户句柄（例如john.doe@domain.com）。 SDK在用户分区内保存多个数据项，例如本地身份验证器数据，设备密钥，TOTP生成器和用户首选项。

对于每个用户，SDK都会保留有关设备是否已成功绑定到该用户的指示。如果在服务器中为应用程序定义的绑定策略至少成功为该设备上的用户完成了一次，则该设备将与用户绑定。

当将设备绑定到用户时，将在设备和传输服务器之间建立加密绑定，以便传输服务器可以在后续调用中验证接近该设备的设备确实已绑定到用户。

要查询用户绑定信息，应用程序可以使用-[TSXTransmitSDKXm getUsersInfo]或-[TSXTransmitSDKXm isBoundForUserWithUserId：]。前者返回设备上绑定用户的TSXUserInfo对象的列表，而后者指示SDK是否知道并绑定了设备上的特定用户句柄。

可以分别使用-[TSXTransmitSDKXm clearAllData]和-[TSXTransmitSDKXm clearDataForUserWithUserId：]为所有用户或给定用户清除SDK状态。清除给定用户的状态后，该用户将不受约束。

## 默认UI处理程序

该SDK为UI处理程序提供了内置的实现，称为TSXDefaultUIHandler。可以通过构造此类的对象来显式地将此实现设置为UI处理程序，或者可以通过不为SDK实例设置任何应用程序提供的UI处理程序来隐式设置此实现。

在调用SDK调用时，应用程序可以通过在客户端上下文中为该调用提供设置来控制由默认UI处理程序实现的UI的各个方面。

### 设置UI托管站点

应用程序可以选择默认的UI处理程序在UI层次结构中显示身份验证UI的方式和位置：

* + - * 要通过要求现有的UIViewController呈现SDK实现的视图控制器来呈现UI，请将通过调用创建的客户端上下文字典传递给+ [TSXDefaultUIHandler

createClientContextForPresentingViewController：]。

* + - * 要通过将UI推送到现有UINavigationController来呈现UI，请将通过调用创建的客户端上下文字典传递给+ [TSXDefaultUIHandler createClientContextForNavigationViewController：]。

以下代码片段说明了如何在UINavigationController中调用策略并实现UI。该代码假定当前的UI处理程序是SDK默认的UI处理程序。

UINavigationController \* navVc =…; NSDictionary \* clientContext =

[TSXDefaultUIHandler createClientContextForNavigationViewController：navVc]; [TSXTransmitSDKXm authenticateWithUserId：@“用户”

policyId：@“策略”其他参数：@ {}

clientContext：clientContext异步WithHandler：^（TSXAuthenticationResult \* res，

TSXAuthenticationError \* err）{

//登录完成

}];

如果传递给默认UI处理程序的客户端上下文字典对象为空，则它将尝试基于内部启发式方法呈现其UI。建议使用上述方法之一显式设置托管站点。

## 身份验证，绑定和策略调用

Transmit SDK的最基本服务围绕设备绑定，用户身份验证和策略调用。以下各节描述如何调用服务以将设备绑定到用户，认证用户并在现有会话中调用策略。回想一下，SDK维护着与

设备上的传输服务器（请参阅第4.3节）。使用此会话中讨论的呼叫建立和删除此会话。可以通过对未绑定用户的绑定调用（第5.4.1节）或对先前绑定用户的身份验证调用来建立会话（请参阅第5.4.2节）。

由于绑定和登录是互斥的，因此应用程序通常通过首先检查用户是否绑定到设备（请参见5.2节）并基于该结果调用以下两者来实现会话建立（登录）流程：绑定或认证操作。本节末尾提供的示例对此进行了演示（第5.4.5节）。

### 绑定设备

要将当前设备绑定到用户，请使用-[TSXTransmitSDKXm bindWith- UserId：AdditionalParameters：clientContext：异步-WithHandler：]异步调用。调用此调用将导致SDK向发送服务器发出请求，以将用户绑定到设备。然后，服务器将评估正在调用的应用程序的绑定策略。 SDK和服务器将一起执行策略-在执行用户交互流程的过程中-直到成功完成。

要启动绑定操作，此调用将要绑定的用户ID，附加参数JSON字典作为参数传递给服务器上的策略（这些参数可在执行时作为策略的一部分来访问）以及客户端上下文作为参数。电话。

成功完成绑定请求后，将发生以下情况：

* + - * 该设备被记录为服务器中用户的绑定设备。这涉及存储作为与该设备到用户的特定绑定关联的绑定过程的一部分协商的加密密钥。
      * SDK将该用户记录为该设备的绑定用户。
      * 为SDK实例设置了一个新的主设备会话，绑定的用户为登录用户。

该调用异步返回TSXAuthenticationResult类型的成功认证结果，其中包含可以用作成功认证证据的认证令牌（令牌属性）以及策略可以返回的其他数据（数据属性）。

如果尝试使用现有的绑定用户ID调用-[TSXTransmitSDKXm bindWithUserId：AdditionalParameters：clientContext：ynchronousWithHandler：]，则该调用将异步失败。要重新绑定已绑定在设备上的用户，请首先通过调用-[-TSXTransmitSDKXm clearDataForUserWithUserId：]重置用户在设备上的状态。

### 调用身份验证策略

SDK调用-[TSXTransmitSDKXm authenticateWithUserId：policyId：AdditionalParameters：clientContext：异步lyHandler：] SDK实施身份验证操作-一种操作，通过该操作为已绑定到本地设备的特定用户启动设备会话。调用此调用将使SDK发出请求以启动新会话并验证该会话中的用户身份，从而与传输服务器联系。该请求还包括一个策略名称，该名称标识了为进行身份验证而调用的服务器端策略。服务器将检索指定的策略名称，并且SDK和服务器将一起在执行用户交互流程的过程中在该策略上执行，直到成功完成。

此调用将要认证的用户ID，要传递给服务器上的策略的其他参数JSON字典作为参数接收（这些参数在执行过程中可被策略访问），以及该调用的客户端上下文。另外，此方法还接收服务器端策略的标识符以作为身份验证过程的一部分来调用。

成功完成身份验证请求后，将为SDK实例设置一个新的主设备会话，将已身份验证的用户作为登录用户。

然后，该调用异步返回TSXAuthenticationResult类型的成功认证结果，其中包含可以用作成功认证证明的认证令牌（令牌属性）以及策略可以返回的其他数据（数据属性） 。

如果尝试使用未绑定的用户ID调用-[TSXTransmitSDKXm authenticateWithUserId：policyId：AdditionalParameters：clientContext：异步-WithHandler：]，则调用将异步失败。

### 在现有会话中调用策略

通常希望在现有设备会话的上下文中调用策略，而不是启动新会话。例如，为了执行可选的逐步身份验证，应用程序可以调用策略，该策略将驱动SDK与用户进行交互以对请求进行身份验证和批准，或者仅返回成功的身份验证完成令牌，而无需用户采取任何进一步的措施。显然，在两种情况下，策略都必须在用户已通过身份验证的会话的上下文中执行。为了简化此用例，SDK包含方法-[-TSXTransmitSDKXm invokePolicyWithPolicyId：AdditionalParameters：clientContext：ynchronlyWithHandler：]。此方法的操作与authenticate方法（第5.4.2节）非常相似，主要区别在于它不使用用户句柄作为参数，因为它使用了当前主设备会话的用户句柄。调用此调用将导致SDK接近

传输服务器，请求在当前主设备会话中运行策略。该请求包括一个策略名称，该名称标识为进行身份验证而要调用的服务器端策略，当然还包括该策略的其他参数。的

服务器将检索指定的策略名称，并且SDK和服务器将一起执行该策略（在执行用户交互流程的过程中），直到成功完成。

成功完成策略调用请求后，该调用将异步返回TSXAuthenticationResult类型的成功身份验证结果，并带有

可以用作成功认证证明的认证令牌（

令牌属性）以及策略可能返回的其他数据（数据

属性）。没有对主设备会话进行任何更改。

如果尝试在没有活动的当前主设备会话（或会话已过期）的情况下调用-[TSXTransmitSDKXm invokePolicyWithPolicyId：AdditionalParameters：clientContext：异步lyWithHandler：]，将导致异步失败。

### 注销

为了注销SDK实例并使当前的主设备会话无效，应用程序调用方法-[TSXTransmitSDKXm logout-

AsynchronouslyWithHandler：]。此方法使与服务器和本地的会话无效，然后异步返回。

### 完整示例：登录或绑定以及注销

以下代码片段显示了一个假设的iOS视图控制器背后的代码，该代码允许用户从应用程序登录和注销。我们的用户界面将由用户名输入字段，登录按钮和注销按钮组成。

我们首先定义UI的成员和样板代码：

#import“ ViewController.h”

#import <TransmitSDK3 / TransmitSDK3.h> @interface ViewController（）

@属性（弱，非原子）IBOutlet UITextField \* usernameField; @属性（弱，非原子）IBOutlet UIButton \* loginButton; @属性（弱，非原子）IBOutlet UIButton \* logoutButton;

@结束

@implementation ViewController

接下来，我们定义函数以中继回有关登录状态（已登录/已注销）和错误警报显示的用户指示：

-（void）setLoggedinToken：（NSString \*）令牌{

//如果令牌不是非零，则表示我们已经成功登录。

//否则表示我们已注销。

self.loginButton.enabled =令牌== nil; self.logoutButton.enabled =令牌！=无;

self.usernameField.enabled = self.loginButton.enabled;

}

-（void）reportError：（NSString \*）err {UIAlertController \* errorAlert =

[UIAlertController alertControllerWithTitle：@“错误”

message：err preferredStyle：UIAlertControllerStyleAlert];

弱的UIAlertController \* errorAlertWk = errorAlert; UIAlertAction \* okAction =

[UIAlertAction actionWithTitle：@“确定”

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull action）{

[errorAlertWk dismissViewControllerAnimated：YES

完成：无]；

}];

[errorAlert addAction：okAction];

[自身presentViewController：errorAlert动画：是完成：无]；

}

我们定义了一种返回客户端上下文以与SDK UI调用一起使用的方法。我们返回的客户端上下文将使用当前视图控制器作为UI的呈现视图控制器。

-（NSDictionary \*）clientContext {

返回[TSXDefaultUIHandler createClientContextForPresentingViewController：self];

}

单击登录按钮使我们的应用程序首先检查输入用户句柄是否已绑定。如果已绑定–调用身份验证策略。否则，请调用绑定策略。在这两种情况下，当调用异步返回时，都会更新UI以反映登录状态。

-（IBAction）onLogin：（id）sender {

//抓取SDK实例和用于登录的用户名。 TSXTransmitSDKXm \* transmitSdk = [TSXTransmitSDKXm sharedInstance]; NSString \* user = self.usernameField.text;

//如果尚未将用户绑定到设备，则if（！[transmitSdk isBoundForUserWithUserId：user]）{

//绑定新用户，并在完成时更新登录UI [transmitSdk bindWithUserId：user

AdditionalParameters：@ {} clientContext：[self clientContext]

异步lyWithHandler：^（TSXAuthenticationResult \* \_Nonnull res，

TSXAuthenticationError \* \_Nullable err）{

if（err）{

[self reportError：[错误描述]];

}其他

{

[self setLoggedinToken：res.token];

}

}];

}其他{

//用户已经绑定；验证用户身份并在完成后更新登录UI

[transmitSdk authenticateWithUserId：user

policyId：@“ default\_auth”的其他参数：@ {} clientContext：[self clientContext]

异步lyWithHandler：^（TSXAuthenticationResult \* res，TSXAuthenticationError \* err）{

if（err）{

[self reportError：[错误描述]];

}其他

{

[self setLoggedinToken：res.token];

}

}];

}

}

最后，在注销时–只需调用注销SDK调用，并在完成后更新UI中的登录状态。

-（IBAction）onLogout：（id）sender {

TSXTransmitSDKXm \* transmitSdk = [TSXTransmitSDKXm sharedInstance];

[transmitSdk logoutAsynchronouslyWithHandler：^（BOOL res，TSXAuthenticationError \*

\_Nullable err）{

[self setLoggedinToken：nil];

}];

}

@结束

## 管理审批

使用“移动批准”，身份验证策略可以要求将批准操作（例如登录）的请求发送到一个或多个移动设备，并且该策略

执行将暂停，直到用户在移动设备上满足该请求为止。如果批准请求已在设备上批准，则启动批准的原始策略将继续执行。如果拒绝批准，则原始策略将中止。在移动设备上批准操作时，可能需要用户通过身份验证策略，就像在身份验证或绑定请求中一样，以完成批准过程。审批可能会或可能不会伴随来自以下机构的推送请求

服务器通知未决的批准请求。当应用程序收到指示新的未决批准的推送通知时，通常，应用程序响应将是允许用户处理未决的批准。

SDK提供了实施批准处理的工具。首先，如果应用程序希望启动一个批准会话，在该会话中将向用户显示未决的批准并有机会对其进行响应，则SDK提供方法-[-TSXTransmitSDKXm startApprovalsSessionForCurrentSessionWith- ClientContext：异步地处理程序：]。此电话将：

* 获取主要设备会话用户的当前待审批列表。
* 启动UI流程以向用户呈现批准，从而允许用户批准或拒绝每个批准
* 用户结束批准会话后（例如，通过完成所有待处理的批准的批准过程），异步返回。

下面提供了作为UIKit操作处理程序一部分调用-[TSXTransmitSDKXm startApprovalsSessionFor- CurrentSessionWithClientContext：异步地处理程序：]调用的示例。这建立在第5.4.5节中设置的定义的基础上。

-（IBAction）onShowApprovals：（id）sender {

TSXTransmitSDKXm \* transmitSdk = [TSXTransmitSDKXm sharedInstance]; [transmitSdk

startApprovalsSessionForCurrentSessionWithClientContext：[self clientContext]

异步lyWithHandler：^（BOOL res，TSXAuthenticationError \* \_Nullable err）{

if（err）{

[self reportError：[错误描述]];

}

}];

}

请注意，在没有有效的主设备会话的情况下调用此调用会导致异步失败。

要处理推送通知以进行移动审批，必须首先设置应用程序和SDK以支持推送通知-如第3.4.3节所述。

当报告新批准的推送通知处于待处理状态时，应用程序应启动一个批准会话。推送通知通常会指定要为其处理批准的用户-这应该是为其处理批准的用户。因此，在收到推送通知时，应用程序应确定它是否为批准通知，以及是否提取有关用户的信息并相应地启动会话。 SDK提供了API来完成所有这些任务：

1. 如果使用普通的APNS，并且传输服务器已配置为生成批准推送有效负载，请在传入通知的JSON有效负载上调用+ [TSXMobileApprovePushRequestPayload create- WithJsonPayloadWithPayload：]，以标识移动设备认可的推送通知并返回其解析的有效负载。

**注意：如果** 不使用平台本机推送通知服务，则应用程序可能仍会使用此方法以特定于集成的方式解析附加到通知的JSON-具体取决于推送通知传递的确切架构。该应用程序还可以从推送消息中手动提取信息，并构建TSXMobileApprovePushRequest-

有效载荷对象使用在此对象上定义的其他构造方法；请参阅API参考以获取更多信息。

1. 如果没有返回有效载荷，则该通知消息不是移动设备认可的推送通知，因此应进行正常的通知消息处理。
2. 如果从+ [TSXMobileApprovePush- RequestPayload createWithJsonPayloadWithPayload：]返回了移动推送请求有效负载，则将该有效负载传递给-[TSXTransmitSDKXm startApprovalsSessionForPushedRequestWith- PushRequestPayload：clientContext：ynchronlyWithHandler：]。

下面的代码示例中说明了此过程：

-（BOOL）应用程序：（UIApplication \*）应用程序didFinishLaunchingWithOptions：（NSDictionary \*）launchOptions {

[self initializeTransmitSdk]; BOOL已处理=否； NSDictionary \* payloadJson =

launchOptions [UIApplicationLaunchOptionsRemoteNotificationKey]; if（payloadJson）{

TSXMobileApprovePushRequestPayload \*有效载荷= [TSXMobileApprovePushRequestPayload

createWithJsonPayloadWithPayload：payloadJson];

if（有效载荷）{

[[TSXTransmitSDKXm sharedInstance] startApprovalsSessionForPushedRequestWithPushRequestPayload：payload clientContext：@ {}

异步lyWithHandler：^（BOOL r，TSXAuthenticationError \* e）{

//响应批准结果

}];

返回是；

}

}

//对非批准的推送通知返回YES执行特定于应用程序的处理；

}

为其处理通知的用户可能不同于登录到当前主设备会话的用户。 -[TSXTransmitSDKXm startApprovals- SessionForPushedRequestWithPushRequestPayload：clientContext：异步地处理程序：]将为适当的用户启动批准会话。但是，这不会修改主设备会话。

## 管理设备

通过使用设备管理功能，SDK使用户可以查看该应用程序的用户绑定设备列表并对其进行操作（例如，锁定它们或发送推送通知以标识该设备）。

要启动设备管理UI，应用程序将调用-[TSXTransmit- SDKXm startDeviceManagementSessionWithClientContext：ynchronlyWithHandler：]方法。该方法将：

* 为登录到主设备会话的用户获取当前设备列表。
* 启动用于向用户呈现设备列表的UI流程，并允许用户在这些设备上执行操作（例如锁定，删除，重命名和标识）。
* 用户结束设备管理会话后（例如，通过关闭设备列表），异步返回。

如果用户通过删除当前设备终止了设备管理会话，则此函数异步返回后，当前登录的会话将无效，并且持久用户数据将从设备中删除（这意味着该设备不再与该用户绑定）。

下面提供了作为UIKit操作处理程序一部分调用-[TSXTransmitSDKXm startDeviceManagement- SessionWithClientContext：异步地处理程序：]调用的示例。这建立在第5.4.5节中设置的定义的基础上。

-（IBAction）onShowDevices：{id）sender {

TSXTransmitSDKXm \* transmitSdk = [TSXTransmitSDKXm sharedInstance]; [transmitSdk

startDeviceManagementSessionWithClientContext：[self clientContext]

异步lyWithHandler：^（BOOL res，TSXAuthenticationError \* \_Nullable err）{

if（err）{

[self reportError：[错误描述]];

}

}];

}

请注意，在没有有效的主设备会话的情况下调用此调用会导致异步失败。

## 显示身份验证器配置UI

身份验证器配置UI允许用户设置与身份验证相关的首选项。这包括查看身份验证器状态，注册和注销身份验证器，以及选择默认身份验证器。

要启动身份验证器配置UI，应用程序将调用-[-TSXTransmitSDKXm startAuthenticationConfigurationWithClient-

上下文：asynchronousWithHandler：]调用。作为此调用的一部分，SDK将从服务器获取配置菜单设置，然后在屏幕上显示配置UI。一旦用户关闭了配置UI，该调用将异步返回。

以下代码示例说明了作为UIKit操作处理程序一部分的此调用的调用。该示例基于第5.4.5节中设置的定义。

-（IBAction）onConfigureAuthenticators：{id）sender {

TSXTransmitSDKXm \* transmitSdk = [TSXTransmitSDKXm sharedInstance]; [transmitSdk

startAuthenticationConfigurationWithClientContext：[self clientContext]异步WithHandler：^（BOOL res，TSXAuthenticationError \* \_Nullable err）{

if（err）{

[self reportError：[错误描述]];

}

}];

}

请注意，在没有有效的主设备会话的情况下调用此调用会导致异步失败。

由SDK提供的此调用的另一个版本，-[TSXTransmitSDKXm start- AuthenticationConfigurationWithTokenWithConfigurationToken：clientContext：ynchronallyWithHandler：]，允许在没有活动会话的情况下调用配置屏幕。相反，SDK会接收一个配置令牌，该令牌是从外部提供给应用程序的（例如，通过调用服务器的令牌创建API并将令牌中继到客户端），并基于此令牌可以调用UI屏幕。

## 使用TOTP

传输平台支持基于时间的OTP功能（TOTP）。作为控制流策略执行的一部分，平台可以提供一个或多个TOTP生成器；每个TOTP生成器都能够生成可以由传输服务器进行身份验证的唯一的限时数字代码。调用TOTP生成器时，通常会向用户显示一个UI，显示该生成器的当前代码以及该代码将到期并用新代码替换**之前的剩余时间指示。 UI显示**会定期更新以反映不断变化的代码。由于它仅基于本地存储的机密和时间，因此TOTP生成器可以在离线环境中运行，因此适合作为离线OOB身份验证机制。

在支持的情况下，可以通过密码，指纹或本地人脸检测来保护TOTP生成器；在具有这些保护的情况下，尝试访问TOTP生成器之前将进行身份验证流程。成功认证后，将向用户显示TOTP生成UI。

每个TOTP生成器在配置时都有服务器策略设置的唯一标识符。该标识符在同一设备中的不同生成器之间是唯一的，但是

适用于不同设备中的发电机；例如，在一个应用程序中，所有设备可能都有一个名为“ primary”的生成器和一个名为“ secondary”的生成器。通过省略生成器名称来引用默认生成器。

要启动TOTP生成会话，请调用SDK方法-[TSXTransmitSDKXm startTotpSessionForUserWithUserId：challengeBased：generatorName：clientContext：asynclyWithHandler：]。此异步方法接收为其提供TOTP生成器的用户句柄作为参数，

一个可选的生成器名称（如果省略，将使用默认的TOTP生成器），一个用于UI交互的客户端上下文对象，当然还有一个异步处理程序。此方法将触发TOTP生成器的身份验证UI流程（例如，向用户请求密码或指纹），然后将显示TOTP呈现UI。一旦用户关闭了TOTP演示UI，该方法将异步返回。

下面的示例演示如何为用户句柄调用默认的TOTP生成器。这适合作为绑定到例如启动TOTP的按钮的UIKit操作处理程序。此示例以第5.4.5节中设置的定义为基础。

-（IBAction）onGenereateTotp：{id）sender {

TSXTransmitSDKXm \* transmitSdk = [TSXTransmitSDKXm sharedInstance]; [transmitSdk

startTotpSessionForUserWithUserId：self.usernameField.text generatorName：nil

clientContext：[self clientContext]

异步lyWithHandler：^（BOOL res，TSXAuthenticationError \* \_Nullable err）{if（err）{

[self reportError：[错误描述]];

}

}];

}

应用程序可以通过调用SDK调用来查询SDK是否为用户提供了具有给定ID的TOTP生成器（或默认的TOTP生成器）。

-[TSXTransmitSDKXm isTotpProvisionedForUserWithUserId：generator-名称：]。

-[TSXTransmitSDKXm startTotpSessionWithRequestWithRequest：clientContext：异步lyWithHandler：]本质上执行与-[TSXTransmitSDKXm startTotpSessionForUserWithUser- ID：ChallengeBased：generatorName：clientContext：异步-WithHandler：]相同的操作，但没有提供显式参数（例如生成器名称和它希望在TotpGenerationRequest结构中接收这些参数，该结构可以通过-[TSXTransmitSDKXm totpGeneration- RequestForUserFromCanonicalStringWithUserId：repr：]方法获得，该方法在“规范”字符串上调用，例如，可以在某些字符串上获取部署，

通过扫描由传输服务器构造的QR码，例如当“ Web API”应用程序配置为使用“规范QR码”质询格式通过TOTP身份验证器启用身份验证时。

# 第6章创建UI处理程 序

## 总览

Transmit SDK使客户端应用程序可以为身份验证UI提供其自己的实现。为了提供自定义UI实现，该应用程序提供了一个UI Handler –这是一个实现TSXUIHandler接口的类。这个介面

定义了一组回调，SDK通过这些回调来指导应用程序输入，并且

要调用的用户界面流。

该应用程序可能会在一个不同的应用程序上下文中提供多个自定义UI处理程序和用户不同的UI处理程序，以提供适合这些上下文的用户体验。有关应用程序如何控制SDK对自定义UI处理程序的使用，请参见第3.4.4节和第3.6.7节。

本部分描述了TSXUIHandler接口以及定制UI时客户端应用程序应实现的其他接口，以及创建此类定制的推荐方法。请参考API参考指南，以获得此处讨论的接口的完整形式描述。另外，请参考

SDK随附的示例定制UI处理程序，以获取用于UI定制的源代码级完整示例。

## 客户端上下文和操作上下文

4.7节讨论了客户端上下文（Client Context）–每当调用需要用户交互的操作时，SDK调用者就会提供一个字典对象。提供给SDK的客户端上下文对象是完全不透明的，并按原样传递给UI处理程序。

有效地，UI处理程序实现定义了客户端上下文对象的预期结构。该对象通常包含有关UI上下文的信息，UI处理程序应在其中显示其UI。例如，一种实现

UI处理程序的可能要求提供用于呈现UI视图的UIKit超级视图。客户端上下文还可以用于在不同的回调之间传递整个策略执行过程中的信息。当回调执行时，它们可能会利用其他回调可以使用的信息来丰富它。

客户端在SDK调用时传递的客户端上下文对象将传递给所有接受客户端上下文的TSXUIHandler（和相关接口）方法

并作为此SDK调用的一部分被调用。上下文作为参数传递

名为clientContext，或者如果它是调用的唯一参数，则将用

WithClientContext：后缀添加到通话名称中。请注意，SDK保证

传递完全相同的字典对象引**用，而不仅仅是传递它的副本，因此您可以更新字典并在回调之间**传递信息。

就像客户端上下文可用于获取应用程序提供的用于调用UI的上下文一样，SDK将操作上下文提供给UI处理程序

指示在执行UI处理程序的上下文中的身份验证策略操作的实现。 SDK将其作为TSXUIHandler方法的actionContext参数提供。该参数的类型为TSXPolicyAction。它提供对有关当前运行策略的以下信息的访问：

**描述**

行动类型行动替代标签

**属性**

actionType altLabel

策略动作上下文对象可用于提供其他参数，从Transmit Platform服务器中的动作定义到自定义UI实施。

**注意：Tr** ansmit Server提供了控制UI行为的各种参数，这些参数默认情况下传递给UI处理程序。在使用操作上下文做出UI决定之前，建议您确保平台中没有现有的结构化机制来中继此信息。同样，在使用此机制时，应考虑正常的客户端-服务器协议注意事项，例如，如何确保不同版本的客户端可以正确处理这些参数（向前和向后兼容性），以及如何为这些参数定义语义，例如它们将适用于将来的用例，而无需付出额外的努力。

## 基于库存UI进行自定义

默认的UI处理程序实现TSXDefaultUIHandler实现了SDK随附的常规UI。应用程序可以从此类派生其TSXUIHandler实现，从而仅自定义UI的某些部分，而将现有实现重用于其他部分，或者完全提供其自己的实现。

在后一种情况下，对于较新版本的SDK，应用程序可能需要为其他TSXUIHandler方法提供实现。在前一种情况下，由于引入了新的UI流程，应由应用程序作者决定是否应在库存实现之外对它们进行自定义；如果没有，那么库存实现将自动使用。

将UI处理程序基于TSXDefaultUIHandler时，应通过调用第5.3.1节中描述的TSXDefaultUIHandler上下文工厂方法之一来创建传输到SDK的UI上下文。这些工厂方法

返回可变的字典，调用应用程序可以使用TSXDefaultUIHandler派生的其他数据来扩充该可变字典。 UI处理程序的应用程序提供的实现可以

处理使用TSXDefaultUIHandler创建的客户端上下文对象

通过调用客户端上下文对象上的方法+ [TSXDefaultUIHandler HostingContextFromClientContext：]实现工厂方法。这将返回TSXDefaultUIHandlerHostingContext类型的对象，该对象包含在创建客户端内容时指定的UI托管参数。该对象提供以下服务，用于基于这些设置来实现UI：

* + - -[TSXDefaultUIHandlerHostingContext presentAlertController：]

允许UI处理程序实现呈现UIAlertController。

* + - -[TSXDefaultUIHandlerHostingContext presentViewController：animated：]允许UI处理程序实现呈现任意视图控制器实现。取决于应用程序在上下文对象创建时提供的托管设置，视图控制器可以显示为现有父控制器的子视图控制器，推送到导航视图控制器或由现有视图控制器以模态显示。注意

某些主机设置可能不支持此调用中的animation：参数

–例如，当UI托管为子视图控制器时。

* + - 应用程序使用-[TSXDefaultUIHandlerHostingContext presentViewController：animated：]取消由-[TSXDefaultUIHandlerHostingContext presentViewController：animated：]呈现的视图控制器。

**注意：创建** 不基于TSXDefaultUIHandler类的UI处理程序实现时，您没有

使用由TSXDefaultUIHandler上下文工厂方法+ [TSXDefaultUIHandler HostingContextFromClientContext：]或

TSXDefaultUIHandlerHostingContext类；而是可以在客户端上下文中使用任何字典结构，只要确保在调用SDK时应用程序提供正确的字典以供自定义UI处理程序使用即可。但是，即使您选择不使用TSXDefaultUIHandler，也可以使用所有这些服务。

## 实施自定义UI处理程序

客户端UI处理程序实现包括TSXUIHandler协议的实现以及涵盖特定UI集成的其他协议。 TSXUIHandler协议用作自定义UI处理程序实现的入口点；构成特定UI处理程序实现的所有其他接口都是通过此协议中定义的方法创建的。

用于接收来自应用程序的输入的UI处理程序回调，例如-[-TSXUIHandler selectAuthenticatorWithOptions：actionContext：clientContext：异步lyWithHandler：]，请遵循异步调用模型，在该模型中返回输入（例如用户选择或用户输入）

回调调用后异步进行。这允许UI处理程序从用户收集信息，并在用户可用时将其返回，而不会阻塞输入请求调用。

### 活动指标

SDK分别通过调用-[TSXUIHandler startActivityIndi​​catorWith- ActionContext：clientContext：]和-[TSXUIHandler endActivity- IndicatorWithActionContext：clientContext：]来传达长时间运行的后台操作的开始和终止。对于应用程序进行的给定SDK操作调用，如果没有对同一操作进行中间调用-[TSXUIHandler endActivityIndi​​cator-WithActionContext：clientContext：]，则可能无法两次调用-[TSXUIHandler startActivityIndi​​catorWithActionContext：clientContext：]。但是，对于不同的操作调用，对这些函数的调用没有任何限制。

将TSXUIDefaultUIHandlerHostingContext类用于客户端上下文时，可以使用其activityIndi​​cator属性访问活动指示器设置，以符合应用程序设置的UI托管设置。然后可以根据活动回调显示或隐藏该活动指示器。以下代码片段演示了这一点：

@implementation MyCustomUIHandler

-（void）startActivityIndi​​catorWithActionContext：（TSXPolicyAction \*）actionContext

clientContext：（NSDictionary \*）clientContext {

[TSXDefaultUIHandler HostingContextFromClientContext：clientContext] .activityIndi​​cator.hidden = NO；

}

-（void）endActivityIndi​​catorWithActionContext：（TSXPolicyAction \*）actionContext

clientContext：（NSDictionary \*）clientContext {

[TSXDefaultUIHandler HostingContextFromClientContext：clientContext] .activityIndi​​cator.hidden = YES；

}

@结束

### 验证动作

UI处理程序需要支持的最基本的UI交互是身份验证菜单操作。在执行身份验证菜单操作的过程中，与用户的交互可能包括：

* + - * 允许用户选择要使用的身份验证器，并可能带有其他控制身份验证过程的参数，
      * 显示身份验证器用户界面并获取用户输入，
      * 在整个身份验证用户输入期间向用户提供反馈（例如，通知提供的错误密码或要提供的其他信息），
      * 建议用户在某些情况下将身份验证方法更改为辅助回退方法，
      * 确定如何从某些身份验证错误中恢复，
      * 允许用户以各种方式（例如，完全取消身份验证，选择其他身份验证器等）更改身份验证流程。

这种复杂的用户交互是由UI处理程序实现中的多个回调处理的。

首先，每当SDK需要选择要激活的身份验证器时，它都会调用方法-[TSXUIHandler shouldIncludeDisabledAuthenticatorsIn- MenuWithActionContext：clientContext：]，然后调用方法-[TSXUIHandler selectAuthenticatorWithOptions：actionContext：clientContext：ynchronlyWithHandler：]。第一次调用允许UI处理程序指定

它是否要在选择器的身份验证器列表中接收身份验证器，也要禁用选择的身份验证器（例如，由于被锁定或未注册）。如果此方法返回true，则禁用的身份验证器将包含在选择列表中，应用程序必须确保不将它们作为选定的身份验证器返回。可以通过咨询-[TSXAuthenticator-描述getEnabled]来标识禁用的身份验证器。第二个呼叫接收身份验证选项列表，用户应在其中选择一个要使用的选项。每个选项都代表

由TSXAuthenticationOption对象。该对象包括对身份验证器类型的引用，身份验证器以及建议的身份验证列表

动作参数，suggestedParameters。预期该回调将返回一个TSXAuthenticatorSelectionResult对象，该对象可以指示所选的身份验证器（可能带有某些身份验证参数）或取消当前身份验证过程的请求。此方法的实现可能会要求用户选择验证者，然后异步返回选择，或者可能

以任何其他方式以编程方式做出该决定。请注意，传递给此呼叫的身份验证器列表可能包括也可能不包括禁用的身份验证器-取决于

在对-[TSXUIHandler应该包含-DisabledAuthenticatorsInMenuWithActionContext：clientContext：]的调用中返回给SDK的值。

可用的身份验证操作参数取决于身份验证器类型。

例如，对于可能需要选择目标设备或通道的身份验证器（例如OTP身份验证器），参数可以包括要使用的目标。注意

参数始终是可选的，并且在不提供任何参数的情况下请求启动身份验证过程始终是有效的。

要将验证者选择返回给SDK，此函数的实现应通过调用+ [TSXAuthenticatorSelectionResult createSelectionRequest- WithSelectedAuthenticator：]与调用时传递给它的验证选项之一的验证者对象异步返回TSXAuthenticatorSelectionResult对象。要包括其他参数，请使用

+ [TSXAuthenticatorSelectionResult createParameterizedSelection- RequestWithSelectedAuthenticator：parameters：]除了选择的身份验证器之外，还接收用于身份验证操作的参数列表（通常在建议作为参数一部分的参数中选择这些参数）

身份验证选项）。要向SDK指示应取消该操作，应通过调用+ [TSXAuthenticatorSelection-结果createAbortRequest]构造返回的对象。

下面的代码片段演示了-[TSXUIHandler selectAuthenticatorWithOptions：actionContext：clientContext：异步lyHandler：]调用的实现，其中向用户显示菜单以选择所需的验证器，或完全取消操作。这说明了

以响应基于UI事件的此调用异步返回用户输入。像本章中的所有示例一样，这假定应用程序将使用TSXDefaultUIHandler客户端上下文调用该策略。

@implementation MyCustomUIHandler

-（void）selectAuthenticatorWithOptions：（NSArray <TSXAuthenticatorDescription \*>

\*）选项

actionContext：（TSXPolicyAction \*）actionContext clientContext：（NSDictionary \*）clientContext

ynchronousWithHandler：（void（^）（TSXAuthenticatorSelectionResult \*

\_Nonnull））处理程序

{

//准备身份验证方法菜单UIAlertController \* authOptionMenu =

[UIAlertController alertControllerWithTitle：@“验证”

消息：@“请选择认证

方法：“ preferredStyle：UIAlertControllerStyleActionSheet];

//为（TSXAuthenticatorDescription \* auth in options）{

if（！auth.enabled）{继续;

}

//为身份验证器TSXAuthenticatorDescription创建操作\* currentAuth = auth; UIAlertAction \* authOpt =

[UIAlertAction actionWithTitle：auth.name

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull action）{

处理程序（[TSXAuthenticatorSelectionResult

createSelectionRequestWithSelectedAuthenticator：currentAuth]）；

}];

[authOptionMenu addAction：authOpt];

}

//添加取消按钮UIAlertAction \* cancelOpt =

[UIAlertAction actionWithTitle：@“取消”

样式：UIAlertActionStyle破坏性处理程序：^（UIAlertAction \* \_Nonnull action）{

处理程序（[TSXAuthenticatorSelectionResult

createAbortRequest]）；

}];

[authOptionMenu addAction：cancelOpt];

//向用户显示菜单并恢复auth TSXDefaultUIHandlerHostingContext \* hostingContext =

[TSXDefaultUIHandler hostingContextFromClientContext：clientContext] [hostingContext presentDialogViewController：authOptionMenu];

}

@结束

选择验证者后，SDK将调用TSXUIHandler回调以创建验证者会话对象，该对象将管理与特定选定验证者的交互。第6.4.3节介绍了身份验证器会话及其实现方式。

在与身份验证器进行交互期间，用户（或应用程序）可以向SDK发出取消该特定交互的信号（例如，返回通过调用+ [TSXInputOrControlResponse createControlResponseWith- ControlRequest：]构造的输入响应，并向其传递+ [TSXControlRequest createWith -RequestType：]由ControlRequestTypeCancelAuthenticator调用；请参见第6.4.3节中有关控制请求的讨论。如果发生这种情况，SDK将要求UI处理程序提供响应取消请求采取的措施。这个

通过SDK调用UI处理程序回调-[TSXUIHandler控制-OptionForCancellationRequestInSessionWithValidOptions：会话：alyallyWithHandler：]完成。 SDK调用此回调以将取消请求与不是ControlRequestTypeCancelAuthenticator的控制请求进行“交易”。然后，使用此回调返回的控制请求代替请求取消的原始控制请求。调用此回调时，SDK可能决定启用或禁用某些可能的控制请求类型。例如，如果除了被取消的身份验证器之外没有可用**的身份验**证器，则更改身份验证器的选项将被禁用。启用或允许的控制请求类型在有效选项中可用。返回的控制请求必须是这些请求类型之一。不满足此要求将触发原始SDK操作调用失败。

请注意，此回调不接收客户端上下文对象作为其参数之一；相反，它在发生取消的上下文中接收一个身份验证器会话对象。该会话对象在调用此对象之前已通过客户端上下文传递

回调是正常执行过程的一部分，因此UI处理程序实现可以从中获取客户端上下文。

下面的代码片段演示了-[TSXUIHandler controlOptionForCancellationRequestInSessionWithValidOptions：session：ynchronousWithHandler：]的实现，该方法基本上允许用户选择其中一个选项。

@implementation MyCustomUIHandler

-（void）controlOptionForCancellationRequestInSessionWithValidOptions：

（TSXControlRequestTypeSet）validOptions会话：（TSXUIAuthenticatorSession <TSXInputResponseType \*>

\*）会议

异步处理程序：（void（^）（TSXControlRequest \* \_Nonnull））处理程序

{

UIAlertController \* alert = [UIAlertController

alertControllerWithTitle：@“取消身份验证”消息：@“您要如何进行？” preferredStyle：UIAlertControllerStyleActionSheet];

if（validOptions＆ControlRequestTypeRetryAuthenticator）{

[alert addAction：[UIAlertAction actionWithTitle：@“尝试再次进行身份验证”

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull

动作）{

处理程序（[TSXControlRequest

createWithRequestType：ControlRequestTypeRetryAuthenticator]）；

}]];

}

if（validOptions＆ControlRequestTypeChangeMethod）{

[alert addAction：[UIAlertAction actionWithTitle：@“更改身份验证

方法”

动作）{

处理程序（[TSXControlRequest

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull

createWithRequestType：ControlRequestTypeChangeMethod]）；

}]];

}

if（validOptions＆ControlRequestTypeAbortAuthentication）{

[alert addAction：[UIAlertAction actionWithTitle：@“取消身份验证”

样式：UIAlertActionStyle破坏性处理程序：^（UIAlertAction \* \_Nonnull

动作）{

处理程序（[TSXControlRequest

createWithRequestType：ControlRequestTypeAbortAuthentication]）；

}]];

}

TSXDefaultUIHandlerHostingContext \* hostingContext = [会话responsesToSelector：@selector（clientContext）]吗？ [TSXDefaultUIHandler HostingContextFromClientContext：

[[id <TSXUIViewControllerBasedAuthenticatorSessionProtocol>）会话clientContext]]：

[TSXDefaultUIHandler defaultHostingContext];

[hostingContext presentDialogViewController：alert];

}

@结束

可以使用后备逻辑在传输平台中配置身份验证菜单操作。此功能允许策略作者指定在多次失败后无法通过一个身份验证器进行身份验证，建议用户更改为其他身份验证器-策略中明确指定的身份验证器或身份验证菜单中可用的任何其他身份验证器。当SDK需要执行此后备策略时，它会调用UI处理程序回调-[TSXUIHandler selectAuthenticator- FallbackActionWithValidOptions：fallbackAuth：session：action- Context：clientContext：ynchronlyWithHandler：]。回调将基于当前可用的身份验证器提供有关回退的有效响应的信息，并提供可接受的回退的策略定义（validOptions）。如果

该策略定义了要回退到的特定身份验证者，这也提供给

回调（fallbackAuth）。一个实现可以允许用户选择一个后备操作或运行特定于应用程序的逻辑来做出该决定。该回调应异步返回TSXAuthenticatorFallbackAction枚举值之一。返回的值必须是有效选项参数中列出的值之一。不满足此要求将触发原始SDK操作调用失败。

可能的后备操作的语义如下：

TSXAuthenticatorFallbackAction

值

AuthenticatorFallbackAction-后备

AuthenticatorFallbackAction- AuthMenu

语义学

根据策略配置将身份验证方法更改为后备身份验证器。

调用身份验证方法菜单。

AuthenticatorFallbackAction-重试

AuthenticatorFallbackAction-取消

不要执行回退；继续重试当前的身份验证器。

中止认证流程。

以下代码说明了-[TSXUIHandler selectAuthenticatorFallbackActionWithValidOptions：fallback- Auth：session：actionContext：clientContext：异步地与Handler：]的可能实现，其中，向用户展示选择菜单，其中列出了有效的回退操作，并基于用户选择返回值。

@implementation MyCustomUIHandler

-（void）selectAuthenticatorFallbackActionWithValidOptions：

（TSXAuthenticatorFallbackActionSet）validOptions

fallbackAuth：（TSXAuthenticatorDescription \*）fallbackAuth

会议：

（TSXUIAuthenticatorSession <TSXInputResponseType \*> \*）会话

actionContext：（TSXPolicyAction

\*）actionContext

\*）clientContext

clientContext：（NSDictionaryynchronousWithHandler：

（void（^）（TSXAuthenticatorFallbackAction））处理程序

{

UIAlertController \* alert = [UIAlertController

alertControllerWithTitle：@“身份验证失败”

消息：@“尝试重试太多。您要吗

尝试另一种身份验证方法？”

preferredStyle：UIAlertControllerStyleActionSheet];

if（validOptions＆AuthenticatorFallbackActionFallback）{

[alert addAction：[UIAlertAction actionWithTitle：@“将方法更改为后备”

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull

动作）{

handler（AuthenticatorFallbackActionFallback）;

}]];

}

if（validOptions＆AuthenticatorFallbackActionAuthMenu）{

[alert addAction：[UIAlertAction actionWithTitle：@“使用其他方法”

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull

动作）{

handler（AuthenticatorFallbackActionAuthMenu）;

}]];

}

if（validOptions＆AuthenticatorFallbackActionRetry）{

[alert addAction：[UIAlertAction actionWithTitle：@“请勿更改方法”

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull

动作）{

处理程序（AuthenticatorFallbackActionRetry）;

}]];

}

[alert addAction：[UIAlertAction actionWithTitle：@“中止身份验证”

样式：UIAlertActionStyle破坏性处理程序：^（UIAlertAction \* \_Nonnull

动作）{

处理程序（AuthenticatorFallbackActionCancel）;

}]];

[[TSXDefaultUIHandler HostingContextFromClientContext：clientContext] presentAlertController：alert];

}

@结束

### 实施身份验证器会话接口

用户和认证者之间的交互可以是多步骤过程。要求用户提供一些身份证明，系统可以提供反馈，依此类推，直到验证者获得了需要完成的所有准确证明。作为多步骤过程，需要一些上下文来保留阶段之间的认证步骤。因此，与身份验证器的交互被建模为会话

宾语。身份验证器会话是实现TSXUIAuthenticator- Session接口或从其派生的接口的对象。该对象由应用程序实现，作为UI定制的一部分。当SDK要求创建SDK时，会将其提供给SDK。通过SDK调用下表中列出的身份验证器会话创建功能之一来创建会话。

**表3：**

|  |  |
| --- | --- |
| 验证码类型 | 会话创建回调 |
| 密码验证器 | -[TSXUIHandler创建-PasswordAuthSessionWithTitle：用户名：] |
| 指纹认证器 | -[TSXUIHandler创建-FingerprintAuthSessionWith-标题：用户名：] |
| 本地人脸（FaceID）身份验证器 | -[TSXUIHandler createNative- FaceAuthSessionWithTitle：用户名：] |
| PIN码验证器 | -[TSXUIHandler createPinAuth- SessionWithTitle：用户名：pinLength：] |
| 模式验证器 | -[TSXUIHandler createPattern- AuthSessionWithTitle：用户名：gridWidth：gridHeight：] |
| OTP验证器 | -[TSXUIHandler createOtpAuth- SessionWithTitle：用户名：可能目标：autoExeced-目标：] |
| 面部认证器 | -[TSXUIHandler createFaceAuth- SessionWithTitle：用户名：] |
| 语音认证器 | -[TSXUIHandler createVoiceAuth- SessionWithTitle：用户名：] |
| 移动批准身份验证器 | -[TSXUIHandler createMobile- ApproveAuthSessionWithTitle：用户名：说明：] |
| 托普认证器 | -[TSXUIHandler createTotpAuth- SessionWithTitle：用户名：] |

|  |  |
| --- | --- |
| 安全问题验证器 | -[TSXUIHandler createSecurity- QuestionAuthSessionWithTitle：用户名：] |
| 占位符验证器 | -[TSXUIHandler创建-PlaceholderAuthSessionWith- PlaceholderName：占位符-类型：标题：用户名：authenticatorConfiguredData：serverPayload：] |

一旦选择了身份验证器，身份验证器UI会话就会开始，并在使用身份验证器的身份验证完成或中止时终止（例如，由于用户选择其他身份验证器或完全取消身份验证操作）。

SDK可能需要调用身份验证器以实际验证用户身份或执行初始注册。例如，可以调用PIN身份验证器以收集PIN作为身份验证的一部分，或者为将来设置PIN

身份验证。因此，身份验证器会话可以处于以下两种模式之一（由TSXAuthenticatorSessionMode表示）–身份验证（Authenticator-SessionModeAuthentication）或注册（AuthenticatorSessionMode- Registration）。模式是在会话开始时设置的，但是在会话期间可能会更改。

具体来说，当使用过期的身份验证器进行身份验证时，会话模式将以身份验证开始（因为我们正在对用户进行身份验证），但是一旦成功完成身份验证，会话模式将更改注册，而不是

为注册操作启动一个单独的UI会话。这使定制的UI处理程序可以区分在与（过期）身份验证相同的上下文中完成的注册与独立的注册会话。

会话类与响应类型相关联-会话响应输入的类型。这是用于将输入从身份验证器会话中继到SDK的类型。不同身份验证器类型的身份验证器UI会话将使用不同的响应类型。

* + - 1. **通用身份验证器会话接口**

创建会话后，会话的生命周期始终始于对-[-TSXUIAuthenticatorSession startSessionWithDescription：mode：actionContext：clientContext：]的调用，此调用是在通过上述一种会话创建方法创建会话对象后不久由SDK完成的。在此调用中，会话接收初始会话模式以及用于在其中创建会话的SDK操作调用的客户端和操作上下文。此方法的实现应准备UI以与身份验证器进行交互-通常（但不一定）在屏幕上呈现新的UI元素，记录客户端上下文并将UI设置为提供的会话模式。

以下代码段演示了会话的可能实现

-[TSXUIAuthenticatorSession startSessionWithDescription：mode：actionContext：clientContext：]调用。我们首先创建一个示例会话类声明；在我们的示例中，我们将定义PIN输入身份验证器会话。

@interface MyPinAuthenticationSession：TSXUIAuthenticatorSession {int \_pinLen;

TSXDefaultUIHandlerHostingContext \* \_hostingContext; MyPinAuthenticationSessionVc \* \_viewController;

无效（^ \_currentHandler）（TSXInputOrControlResponse <TSXPinInput \*> \*）;

}

-（instancetype）initWithPinLen：（int）pinLength;

-（void）pin提交：（NSString \*）pin;

-（void）cancelAuthenticator;

@结束

现在不用担心各种成员声明；随着我们继续该示例，它们将变得更加清晰。现在我们已经定义了会话类，我们可以在UI处理程序中实现代码，以根据SDK的请求创建并返回PIN输入会话。

@implementation MyCustomUIHandler

-（TSXUIAuthenticatorSession <TSXPinInput \*> \*）createPinAuthSessionWithTitle：（NSString

\*）标题

用户名：（NSString \*）用户名pinLength：（long）pinLength {

返回[[MyPinAuthenticationSession alloc] initWithPinLen：（int）pinLength];

}

@结束

因此，当SDK要启动与PIN身份验证器的交互时，它将首先调用-[TSXUIHandler createPinAuthSessionWithTitle：username：pin- Length：]，这将依次构造由自定义UI处理程序实现的MyPinAuthetnicationSession对象，并返回该对象到SDK。接下来，SDK将在会话上调用-[TSXUIAuthenticatorSession startSessionWithDescription：mode：actionContext：clientContext：] –这是如何实现的：

@implementation MyPinAuthenticationSession

-（instancetype）initWithPinLen：（int）pinLength {self = [super init];

如果（自己）{

\_pinLen = pinLength;

}

返回自我

}

-（void）startSessionWithDescription：（TSXAuthenticatorDescription \*

\_Nonnull）说明

模式：（TSXAuthenticatorSessionMode）模式操作上下文：（TSXPolicyAction \*

\_Nullable）actionContext

clientContext：（NSDictionary \* \_Nullable）clientContext {

//从客户端上下文中提取托管上下文。

\_hostingContext = [TSXDefaultUIHandler HostingContextFromClientContext：clientContext];

//创建身份验证器UI作为视图控制器

\_viewController = [[MyPinAuthenticationSessionVc alloc] initWithSession：self];

//并使用托管上下文进行呈现

[\_hostingContext presentViewController：\_viewController动画：是完成：无]；

}

@结束

在此实现中，PIN身份验证UI被实现为与当前托管上下文一起呈现的视图控制器。视图控制器具有对创建它的会话对象的引用，并且将取消和PIN提交事件中继到该会话。此外，它还支持显示错误消息的错误提示栏：

@interface MyPinAuthenticationSessionVc（ ）{

MyPinAuthenticationSession \* \_session弱；

}

@结束

@implementation MyPinAuthenticationSessionVc

-（instancetype）initWithSession：（MyPinAuthenticationSession \*）session {self = [super initWithNibName：nil bundle：nil];

如果（自己）{

\_session =会话；

}

返回自我

}

* （IBAction）submitClicked：（id）sender {[\_session pinSubmitted：self.pinField.text];

}

* （IBAction）cancelClicked：（id）sender {[\_session cancelAuthenticator];

}

-（void）setErrorPrompt：（NSString \*）prompt {self.errorBar.hidden =提示== nil; self.errorBar.text =提示;

}

@结束

调用-[TSXUIAuthenticatorSession startSessionWith-描述：mode：actionContext：clientContext：]之后，SDK将调用会话对象的-[TSXUIAuthenticatorSession promiseInput-

AsynchronouslyWithHandler：]异步方法-要求UI会话收集并提供来自用户的输入。此方法将异步返回控制请求或身份验证器输入：

* + - * 控制请求会影响身份验证操作的继续处理。控制请求可以例如发信号通知应当终止认证过程或者应该向用户提供更改认证者的权限。通过调用+ [TSXInputOrControlResponse create-

ControlResponseWithControlRequest：]，将通过调用创建的控制请求对象传递给+ [TSXControlRequest createWithRequestType：]，其中此调用获取下表中列出的TSXControlRequestType枚举值之一：

TSXControlRequestType **Description**

ControlRequestTypeCancel- Authenticator

取消身份验证器。另请参见第6.4.2节。

ControlRequestTypeRetry-身份验证器

使用当前的身份验证器重试身份验证

ControlRequestTypeChangeMethod 允许用户更改

身份验证方法。这将导致在不列出当前身份验证方法的情况下显示身份验证选择菜单。仅当当前验证者之外还有至少一个可用的验证者时，此选项才可用。

ControlRequestTypeSelectMethod 允许用户更改

身份验证方法。这将导致身份验证选择菜单

要显示的内容，包括当前的身份验证方法。

ControlRequestTypeAbort-身份验证

完全中止身份验证过程，从SDK调用返回错误

* + - * 身份验证器输入是从特定于身份验证器类型的类派生的对象，表示对身份验证器的实际输入。例如，对于PIN验证器，这将是一个字符串，其中包含从用户输入的PIN。下表列出了不同的身份验证器输入类型以及如何构造它们的示例。

**认证者**

**输入项**

Password TSXPasswordInput：基于字符串密码输入。

示例：+ [TSXPasswordInput createWithPassword：]

Fingerprint TSXFingerprintInput：指定指纹获取对话框的提示。在iOS中，可以指定后备按钮标题和连接的TSXControlRequestType。

本机faceTSXNa tiveFaceInput：指定本机人脸获取对话框的提示。在iOS中，可以指定后备按钮标题和连接的TSXControlRequestType。

PIN TSXPinInput：基于字符串PIN输入。

Pattern TSXPatternInput：提供编码的模式输入。由模式输入控件创建；见6.4.3.3。

OTP Input可能是以下之一：

* + - * + 目标选择请求，通过调用+ [TSXTargetBased- AuthenticatorInput create- TargetSelectionRequestWith- Target：]创建。
        + 重新发送最新生成的OTP的请求，该请求通过传递给+ [TSXTargetBased- AuthenticatorInput create- AuthenticatorInputWith- AuthenticatorInput：]的调用而创建，该请求的类型为TSXOtpInput- RequestResend，通过调用

+ [TSXOtpInputRequestResend createOtpResendRequest]。

* + - * + 提交给身份验证的OTP值，方法是传递给+ [TSXTargetBased- AuthenticatorInput create- AuthenticatorInputWith-

AuthenticatorInput：]一个OTP

TSXOtp- InputOtpSubmission类型的提交请求。

Face TSXCameraInputResponse：提供基于摄像头采集的面部特征。由图像采集控件创建；见6.4.3.3。

Voice TSXAudioInputResponse：提供基于音频采集的音频功能。由音频采集控件创建；见6.4.3.3。

Mobile Approve Input可能是以下之一：

* + - * + 目标选择请求，通过调用+ [TSXTargetBased- AuthenticatorInput create- TargetSelectionRequestWith- Target：]创建。
        + 通过传递给+ [TSXTargetBased-

AuthenticatorInput创建-AuthenticatorInputWith-AuthenticatorInput：]通过调用+ [TSXMobileApprove- InputRequestPolling create- RequestPollingInput]创建TSXMobileApprove- InputRequestPolling类型的轮询请求。

Totp Input可能是以下之一：

* + - * + 目标选择请求，通过调用+ [TSXTargetBased- AuthenticatorInput create- TargetSelectionRequestWith- Target：]创建。
        + 提交给身份验证的TOTP值，方法是传递给+ [TSXTargetBased- AuthenticatorInput create- AuthenticatorInputWith-

AuthenticatorInput：]一个TOTP

TSXTotp- InputCodeSubmission类型的提交请求。

安全问题TSXSecurityQue stionInput-响应：基于对呈现给用户的安全问题的答案提供文本输入。从构造

TSXSecurityQuestionAndAnswer列表。

Placeholder Input可以是以下之一：

* + - * + TSXPlaceholderAuth- SuccessResponse：成功的身份验证响应，通过调用+ [TSXPlaceholder-

InputResponse createSuccess- ResponseWithPlaceholder-

令牌：]。该方法接收

可选的占位符令牌参数；在以下情况下是必需的：身份验证和基于令牌的注册，而对于配置为允许在没有令牌的情况下进行注册的占位符身份验证器，则可以将其定义为未定义。

* + - * + TSXPlaceholderAuthFailure- WithServerProvidedStatus-响应：失败的身份验证响应，是通过调用+ [-TSXPlaceholderInputResponse createFailedResponseWith- ServerProvidedStatusWith- Error：]创建的。此方法接收失败的身份验证错误对象。身份验证失败将报告给身份验证服务器，并根据身份验证服务器对报告的失败的响应来更新身份验证器状态。
        + TSXPlaceholderAuthFailure-响应：通过调用+ [-TSXPlaceholderInputResponse createdFailedResponseWith- CurrentAuthenticatorStatus：error：]创建的身份验证响应失败。此方法接收失败的身份验证错误对象以及由应用程序后端服务定义的更新的身份验证器状态。身份验证失败

被报告给身份验证服务器，而更改后的身份验证器状态被应用程序忽略，而是使用给定的新状态。

-[TSXUIAuthenticatorSession promiseInput- AsynchronouslyWithHandler：]的典型实现将为UI做好准备，以接受来自用户的输入（例如，清除字段），然后记录作为参数传递的异步完成处理程序，并将其保存以备将来参考。一旦用户提供了输入并在UI中指示完成，UI将调用已保存的完成处理程序，并向其传递输入值。或者，如果用户已发起更改身份验证者的请求或更改身份验证流程，则将调用相同的完成处理程序，这一次传递了控制请求值。

对[[TSXUIAuthenticatorSession promiseInputAsynchronously-]的推论

WithHandler：]是异步调用，完成处理程序仅需调用一次即可完成异步返回；一旦调用处理程序，则应将其丢弃。如果需要从身份验证器UI会话中进一步输入，则-[-

TSXUIAuthenticatorSession promiseInputAsynchronouslyWithHandler：]

将使用新的完成块再次调用。

下面的代码演示了自定义UI处理程序PIN输入会话MyPinAuthenticationSession的-[TSXUIAuthenticator- Session promiseInputAsynchronouslyWithHandler：]的实现。如上所述，实现仅记录完成处理程序块并返回。该代码还包括pinSubmitted和cancelAuthenticator方法的定义。

回想一下，当用户提交PIN或选择取消身份验证时，我们的UI会调用这些方法。这些调用触发-[TSXUIAuthenticatorSession promiseInputAsynchronouslyWith- Handler：]的异步完成，以分别使用收集的输入或认证取消请求继续进行认证过程。

@implementation MyPinAuthenticationSession

-（void）promiseInputAsynchronouslyWithHandler：

（void（^）（TSXInputOrControlResponse <TSXPasswordInput \*> \*））处理程序{

\_currentHandler =处理程序;

}

-（void）pinSubmitted：（NSString \*）pin {

\_currentHandler（[[TSXInputOrControlResponse createInputResponseWithInputResponse：[TSXPinInput createWithPin：pin]]））;

}

-（void）cancelAuthenticator {

\_currentHandler（[[TSXInputOrControlResponse createControlResponseWithControlRequest：[TSXControlRequest

createWithRequestType：ControlRequestTypeCancelAuthenticator]]）;

}

@结束

一旦-[TSXUIAuthenticatorSession promiseInputAsynchronouslyWith- Handler：]异步返回输入，则SDK会对其进行处理。如果输入有错误，则SDK可能会通过调用会话的方法-[TSXUIAuthenticatorSession promiseRecoveryForErrorWith-

错误：validRecoveries：defaultRecovery：异步地与处理程序：]。此方法允许会话指示SDK如何从错误中恢复。 SDK将错误从中进行恢复以及有效的恢复选项传递给会话对象。该方法将所选的恢复方法异步返回给SDK。为了简化实现，该方法还接收默认的建议恢复，该恢复是根据错误的类型和当前的身份验证器会话状态确定的。可用的恢复方法（取决于validRecoveries列表）是：

TSXAuthenticationErrorRecovery

值

AuthenticationErrorRecovery-RetryAuthenticator

AuthenticationErrorRecovery- ChangeAuthenticator

AuthenticationErrorRecovery- SelectAuthenticator

描述

重试使用当前的身份验证器进行身份验证。

允许用户选择其他身份验证器。

允许用户选择一个认证者-包括当前的认证者。

AuthenticationErrorRecoveryFail 完全使身份验证操作失败。

**注意：此方** 法是异步的，以允许实现利用模式UI并对其进行排序，以将错误呈现给用户。然而

实现可以选择以非模式方式显示错误，并在执行后立即异步返回。

根据配置的失败身份验证规则，身份验证器可能会在身份验证过程中被锁定。例如，在多次失败的身份验证尝试之后，身份验证器可能会被锁定。要检测到此情况，身份验证器会话可以查询传递给的TSXAuthenticatorDescription对象。

调用-[TSXUIAuthenticatorSession startSessionWithDescription：mode：actionContext：clientContext：]。该对象公开属性以查询锁定状态以及其他身份验证器状态参数。请注意

SDK返回的错误不会直接反映身份验证器的锁定状态；

例如，在身份验证器由于身份验证尝试失败而被锁定的典型情况下，传递给-[TSXUIAuthenticatorSession promiseRecoveryForErrorWithError：validRecoveries：default-Recovery：异步lyHandler：]的错误很简单，即Authentication-ErrorCodeInvalidInput错误。

如果选择的错误恢复为AuthenticationErrorRecoveryRetry- Authenticator，则SDK将使用当前身份验证重试身份验证

会话–再次调用-[TSXUIAuthenticatorSession promiseInput- AsynchronouslyWithHandler：]。

**注意：根据** 身份验证器的类型，即使在SDK收到成功输入之后，SDK仍可能再次调用-[TSXUIAuthenticator-会话promiseInputAsynchronouslyWithHandler：]来请求其他输入。

下面的代码示例说明了-[-TSXUIAuthenticatorSession promiseRecoveryForErrorWithError：validRecoveries：defaultRecovery：asynclyWithHandler：]的可能实现。此实现告诉SDK使用其提供的默认恢复。但是，基于默认恢复，实现要么在视图控制器上显示嵌入式错误提示，要么弹出模式错误警报。

@implementation MyPinAuthenticationSession

-（void）promiseRecoveryForErrorWithError：（TSXAuthenticationError \*）错误validRecoveries：（TSXAuthenticationErrorRecoverySet）validRecoveries defaultRecovery：（TSXAuthenticationErrorRecovery）defaultRecovery异步WithHandler：（void（^）（TSXAuthenticationErrorRecovery））handler {

if（defaultRecovery == AuthenticationErrorRecoveryRetryAuthenticator）{[\_viewController setErrorPrompt：[错误描述]];

}其他

if（defaultRecovery == AuthenticationErrorRecoveryFail）{handler（defaultRecovery）;

}其他

{

UIAlertController \* alertError =

[UIAlertController alertControllerWithTitle：@“错误”消息：[错误描述] preferredStyle：UIAlertControllerStyleAlert];

弱的UIAlertController \* weakCtler = alertError; [alertError addAction：[UIAlertAction actionWithTitle：@“ OK”

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull action）{

[weakCtler dismissViewControllerAnimated：YES完成：无]； handler（defaultRecovery）;

}]];

[\_hostingContext presentAlertController：alertError];

}

}

@结束

成功完成身份验证后，如果身份验证器已过期，并且SDK现在需要为其注册新的密码，它将调用会话方法-[-

TSXUIAuthenticatorSession changeSessionModeToRegistrationAfter-到期]。

当SDK可以根据目前为止收到的输入完成身份验证流程时，或者由于控制请求或错误情况而需要以其他方式结束会话时，SDK将调用方法-[TSXUIAuthenticatorSession endSession]来发出身份验证信号会话已结束。下面提供了此方法的示例实现：

-（void）endSession {if（\_viewController）{

[\_hostingContext dismissViewController：\_viewController动画：是]；

}

}

* + - 1. **特殊身份验证器会话接口**

虽然大多数身份验证器会话均应实现TSXUIAuthenticatorSession接口并遵循6.4.3.1节中讨论的模式，但以下各节列出了此例外。

* + - * 1. **基于目标的身份验证器会话**

基于目标的身份验证器是涉及带外设备或通信通道的身份验证器。示例可以包括OTP认证者，TOTP认证者和移动批准认证者。使用这些身份验证器时，可以使用一个或多个目标（由身份验证策略定义）

应该为身份验证过程选择。这是通过所有基于目标的身份验证器共有的一组接口来实现的，如下所示。

身份验证目标始终由从TSXAuthenticatorTarget派生的对象表示。每个基于目标的身份验证器都可以使用此接口的不同子类。

基于目标的身份验证器的所有身份验证器会话均源自TSXUIAuthenticatorSessionTargetBased。该会话接口除了继承的常规TSXUIAuthenticatorSession方法外，还包括一种方法，

-[TSXUIAuthenticatorSessionTargetBased setAvailableTargets：]。此方法接收可在此身份验证器会话中使用的目标列表。身份验证器会话应选择一个或多个这些目标来使用（除非它知道已经选择了一个目标）。

如果基于目标的身份验证器没有可用的目标，则在启动身份验证器会话之后且在调用-[TSXUIAuthenticatorSession promise-

InputAsynchronouslyWithHandler：]，将要求身份验证器会话通过对-[TSXUIAuthenticatorSession的调用而从代码为AuthenticationErrorCodeAuthenticator- ExternalConfigError的错误中恢复

promiseRecoveryForErrorWithError：validRecoveries：默认-Recovery：asynclyWithHandler：]。

基于TSXUIAuthenticatorSessionTargetBased的身份验证器会话返回TSXTargetBasedAuthenticatorInput类型的输入响应。这些输入响应可以对身份验证者的秘密输入进行编码，也可以对请求进行编码

更改身份验证目标。要创建请求身份验证目标更改的TSXTargetBased- AuthenticatorInput实例，请为单个目标选择调用+ [-TSXTargetBasedAuthenticatorInput createTargetSelectionRequest- WithTarget：]，或为多个目标选择调用+ [TSXTargetBasedAuthenticator- Input createTargetsSelectionRequestWithTargets：]（前者是特殊的后者的情况，因此可以同时用于

案件。要创建提供具体身份验证器输入的实例，请调用+ [-TSXTargetBasedAuthenticatorInput createAuthenticatorInputWith- AuthenticatorInput：]。这两种实例都可以从对[[TSXUIAuthenticatorSession promiseInputAsynchronouslyWith-Handler：]的调用中返回。

所有基于目标的身份验证器均支持接收类型为TSXAuthenticationActionParameterTargetSelection的身份验证参数。此参数指定最初用于身份验证过程的目标。启动身份验证过程时，最多可以提供此参数的一个实例。通过在对身份验证选择请求的响应中指定此参数（-[TSXUIHandler selectAuthenticatorWithOptions：actionContext：clientContext：asynclyWithHandler：]），实现

可以选择初始目标，而不必从-[TSXUIAuthenticatorSession promise Input-

AsynchronouslyWithHandler：]。当然，该实现仍可以通过请求更改选定的目标来响应-[TSXUIAuthenticatorSession promiseInputAsynchronouslyWith-Handler：]。

为基于目标的身份验证器提供给-[-TSXUIHandler selectAuthenticatorWithOptions：actionContext：clientContext：asynclyWithHandler：]的身份验证选项对象（TSXAuthenticationOption），在每个可用的身份验证目标的建议参数属性中都包括TSXAuthenticationActionParameterTargetSelection类型的类型。然后，-[TSXUIHandler select- AuthenticatorWithOptions：actionContext：clientContext：异步lyWithHandler：]的实现可以查阅此列表，并允许用户选择初始身份验证器目标以及选定的身份验证器类型。

* + - * 1. **OTP身份验证器会话**

OTP身份验证器会话是基于目标的身份验证器（请参见第6.4.3.2节）。使用OTP身份验证器时，SDK将通过调用-[TSXUIHandler createOtpAuthSessionWithTitle：用户名：可能的目标：autoExecedTarget：]方法来创建身份验证器会话。此方法返回的身份验证器会话应实现从TSXUIAuthenticatorSessionTargetBased派生的TSXUIAuthenticatorSessionOtp接口。此接口在基本接口中添加了一种方法：-[TSXUIAuthenticatorSession-

Otp setGeneratedOtpWithFormat：target：]。每次生成OTP时，SDK都会调用此方法。在第一次创建会话之后，或者在处理请求OTP生成的OTP身份验证器输入（例如，由+ [TSXOtpInputRequestResend createOtpResend-Request]构造的输入）之后，或者如第6.4节所述更改或初始选择了身份验证器目标之后，这可能会自动发生。 3.2。该方法接收有关生成的OTP格式和将其发送到的目标设备的信息。该实现应更新OTP输入UI，以向用户反映OTP的格式及其目标设备。

在整个OTP身份验证过程中，生成的OTP可能变为无效；例如由于失败策略，要求客户端在多次尝试失败后重新生成OTP。发生这种情况时，将使用空格式和目标调用-[TSXUIAuthenticatorSessionOtp set- GeneratedOtpWithFormat：target：]函数，以指示先前指定的OTP不再有效。

在-[TSXUIAuthenticatorSession promiseRecoveryForErrorWithError：validRecoveries：default- Recovery：异步地处理程序：]中收到用于恢复的OTP错误时，错误对象的data属性中可能包含以下键：

**表4：**

|  |  |
| --- | --- |
| **关键路径** | **描述** |
| Additional\_data.additional\_error | \_Icf othdies为“ 1”，此错误表明先前生成的OTP不再有效，应用程序应触发OTP的重新生成 |
| Additional\_data.wrong\_inputs\_lef | t如果服务器允许用于当前OTP的错误输入数量有限制，则表明剩余的输入数量。 |
| Additional\_data.result | 对于外部OTP提供者，这可能包括验证者提供的其他结果数据。 |

* + - * 1. **移动批准身份验证器会话**

Mobile Approve身份验证器会话是基于目标的身份验证器（请参见第6.4.3.2节）。使用Mobile Approve身份验证器时，SDK将通过调用-[TSXUIHandler create- MobileApproveAuthSessionWithTitle：username：Instructions：]方法来创建身份验证器会话。此方法返回的身份验证器会话应实现TSXUIAuthenticatorSessionMobileApprove接口，该接口派生自

TSXUIAuthenticatorSessionTargetBased。此接口向基本接口添加一种方法：-[TSXUIAuthenticatorSessionMobileApprove setCreated- ApprovalInfoWithTargets：otp：]。 SDK每次都会调用此方法

创建批准的时间。如第6.4.3.2节所述，在首次创建会话之后，更改或初始选择身份验证器目标之后，这可能会自动发生。该方法将接收有关OTP值和格式的信息（如果该信息是为批准而生成的）以及为其创建批准的目标设备。实施应使用OTP UI更新待批准屏幕，以反映OTP（如果已创建）及其向用户的目标设备。

在整个“移动批准”身份验证过程中，创建的批准可能无效；例如由于批准到期。发生这种情况时，-[-TSXUIAuthenticatorSessionMobileApprove setCreatedApprovalInfo-

WithTargets：otp：]函数将使用空otp和目标来指示

先前创建的批准不再有效。

由于Mobile Approve身份验证中的实际批准是在目标设备上执行的（而不是在当前正在进行身份验证会话的设备上执行），因此一旦创建了批准，会话就无需再做更多的工作来完成身份验证，除非等待批准待批准。这是通过轮询服务器以成功批准来完成的。轮询请求要么导致会话成功结束

如果批准已在目标设备上批准，或者如果批准仍在等待中，则SDK将再次调用-[TSXUIAuthenticatorSession promiseInputAsynchronouslyWithHandler：]，直到批准被批准，拒绝或过期为止。会话应注意不要使服务器超负荷轮询请求流量，因此理想的Mobile Approve身份验证器实现应将计时器设置为在轮询请求之间运行。

要创建轮询请求，会话应使用通过调用-[TSXUIAuthenticatorSession promiseInputAsynchronouslyWithHandler：]创建的对象来解决返回到+ [TSXMobileApproveInputRequestPolling createRequestPolling- Input]调用的承诺。这也是个好地方

显示otp（如果它是在创建批准时设置的），并显示指令文本，因为它是在构建时由服务器中继到会话的。综上所述，轮询流程的简单实现如下所示：

函数promisePolling（）{

$（“＃myInstructionsElement”）。text（this.instructions）; if（this.otp）{

var otpValue = this.otp.getValue（）;开关（this.otp.getType（））{

大小写com.ts.mobile.sdk.OtpFormatType.Numeric：

$（“＃myOtpText”）。text（otpValue）.show（）;打破;

案例com.ts.mobile.sdk.OtpFormatType.QrCode：

$（“＃myOtpQr”）。attr（“ src”，“ data：image / jpeg; base64，” + otpValue）.show（）;

打破;默认：

返回Promise.reject（new Error（“不支持的otp格式”））;

}

}

返回新的Promise（function（resolve，reject）{this.pollingTimer = setTimeout（function（）{

this.common.unsetAriaBusy（this.clientContext）;

解析（com.ts.mobile.sdk.InputOrControlResponse.createInputResponse（

com.ts.mobile.sdk.TargetBasedAuthenticatorInput.createAuthenticatorInput（com.ts.mobile.sdk.MobileApproveInputRequestPolling.createRequestPollingInput（）））））；

}, 4000);

})

}

* + - * 1. **Totp身份验证器会话**

Totp验证器会话是基于目标的验证器（请参见6.4.3.2节）。

使用Totp身份验证器时，SDK将通过调用-[TSXUIHandler createTotpAuthSessionWithTitle：username：]方法来创建身份验证器会话。此方法返回的身份验证器会话应实现从TSXUIAuthenticatorSessionTargetBased派生的TSXUIAuthenticatorSessionTotp接口。身份验证器会话将收到第6.4.3.2节中所述的合格目标，如果为null，则表明不需要选择目标。上面的接口在基础上添加了两种方法

接口：-[TSXUIAuthenticatorSessionTotp setTargetDevices：]和-[TSXUIAuthenticatorSessionTotp setChallenge：]。在选择用于代码生成的目标设备后，SDK会调用前一种方法。如6.4.3.2节所述，这是在更改或初始选择了身份验证器目标之后发生的。该方法接收关于目标设备的信息，这些信息有资格生成用于认证的代码。实现应使用生成的目标设备更新UI。如果在会话上调用-[TSXUIAuthenticatorSession startSessionWith-描述：模式：actionContext：clientContext：]之后不久，如果方法配置为在代码生成中包含质询，则SDK会调用-[TSXUIAuthenticatorSessionTotp setChallenge：]。

SDK希望会话响应此调用显示生成的质询。稍后，TOTP代码生成客户端将获取此挑战，以将其包含在生成中。

如果重新尝试执行Totp身份验证（用户单击重试），则目标选择将变为无效。发生这种情况时，将使用空目标调用-[TSXUIAuthenticatorSessionTotp setTarget- Devices：]函数，以指示先前的选择不再有效。

如果目标Totp种子启用了校验位验证并且验证失败，则将要求身份验证器会话从代码AuthenticationErrorCodeInvalidInput的错误中恢复，该代码的Authentication- ErrorPropertyAuthenticatorInvalidInputErrorDescription属性设置为AuthenticationErrorPropertySymbolAuthenticatorInvalid- InputErrorDescriptionTotpInAuthicicator-通过调用[[Session promiseRecoveryForErrorWithError：

validRecoveries：defaultRecovery：aynchronousWithHandler：]，可以在用户界面中指示，并且可以重新尝试进行身份验证。

* + - * 1. **多步骤身份验证器会话**

各种认证器可能需要一个以上的输入来注册或认证。这些身份验证器的会话对象应实现从TSXUIAuthenticatorSession派生的接口TSXUIMulti-InputAuthenticationSession。此接口添加-[TSXUIMultiInputAuthenticationSession setInputStep- WithStepNumber：maxStepNumber：stepDescription：]方法。在每次调用-[TSXUIAuthenticatorSession promise- InputAsynchronouslyWithHandler：]之前，都会调用此方法，以向会话提供有关所请求的特定输入步骤的信息。

人脸ID，语音ID和安全性问题身份验证器使用TSXUIMultiInputAuthenticationSession会话。

* + - * 1. **安全问题身份验证器会话**

安全问题身份验证器会话继承自TSXUIMultiInput- AuthenticationSession，因此包含-[TSXUIMultiInput- AuthenticationSession setInputStepWithStepNumber：maxStepNumber：stepDescription：]方法。对于安全性问题，此方法通过-[TSXSecurityQuestionStepDescription getSecurityQuestions]提供此步骤的安全性问题列表，以及通过-[TSXSecurityQuestionStepDescription getMinAnswersNeeded]回答的最小问题数。

从ui收集答案后，应通过调用+ [TSXSecurity- QuestionInputResponse createSecurityQuestionAnswersInputResponse- WithAnswers：]来使用输入来构造TSXSecurityQuestionInputResponse对象。此结果应用于完成-[-TSXUIAuthenticatorSession promiseInputAsynchronouslyWithHandler：]创建的承诺。完成诺言的调用会将结果传递回SDK。

以下代码说明了安全问题身份验证器将如何提出问题并收集输入。

@interface TSXUIDefaultQuestionsSession（）{TSXSecurityQuestionStepDescription \* \_currentStepDescription; long \_currentAnswrIndex;

}

@结束

@implementation TSXUIDefaultQuestionsSession

-（instancetype）initWithTitle：（NSString \*）title用户名：（id）username {self = [super init];

返回自我

}

-（TSXUIDefaultQuestionsSessionVc \*）questionsVC {

返回（TSXUIDefaultQuestionsSessionVc \*）self.viewController;

}

-（UIViewController \*）createViewController {

TSXUIDefaultQuestionsSessionVc \* vc = [[TSXUIDefaultQuestionsSessionVc分配]

initWithNibName：无捆绑包：[NSBundle

tsResourcesBundle]];

vc.session =自我；返回vc;

}

-（void）setInputStepWithStepNumber：（long）stepNumber maxStepNumber：（long）maxStepNumber stepDescription：（TSXSecurityQuestionStepDescription \*）stepDescription {

\_currentStepDescription = stepDescription;

[[自我问题VC]显示问题：\_当前步骤描述。安全性问题]；

}

-（void）submitAnswers：（NSArray <TSXSecurityQuestionAndAnswer \*> \*）answers {if（answers.count <\_currentStepDescription.minAnswersNeeded）{

[self showNotEnoughAnsweredQuestionsError];

}

其他{

[自行提供输入：[TSXInputOrControlResponse createInputResponseWithInputResponse：[TSXSecurityQuestionInputResponse createSecurityQuestionAnswersInputResponseWithAnswers：answers]]]]]；

}

}

-（void）showNotEnoughAnsweredQuestionsError {

UIAlertController \* alert = [UIAlertController alertControllerWithTitle：@“没有足够的问题回答”消息：[NSString stringWithFormat：@“请

回答至少％ld个问题”，\_currentStepDescription.minAnswersNeeded] preferredStyle：UIAlertControllerStyleAlert];

弱类型（自我）弱自我=自我;

[alert addAction：[UIAlertAction actionWithTitle：@“ Dismiss”样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull操作）{

[weakSelf.viewController dismissViewControllerAnimated：YES完成：无]；

}]];

[self.viewController presentViewController：alert动画：是完成：无]；

}

@结束

@interface TSXUIDefaultQuestionsSessionVc（）{NSArray <QuestionAnswerView \*> \* \_questionAnswerViews; NSArray <TSXSecurityQuestion \*> \* \_questions;

}

@property（弱）UIStackView \* stackView；

@属性（不安全的，未保留的，非原子的）IBOutlet UIScrollView \* scrollView; @结束

@implementation TSXUIDefaultQuestionsSessionVc

-（void）viewDidLoad {[super viewDidLoad];

[self setupStackView];

self.navigationItem.title = @“问题”; self.navigationItem.hidesBackButton = YES; self.navigationItem.leftBarButtonItems = @ [

[[UIBarButtonItem alloc] initWithTitle：@“取消”样式：UIBarButtonItemStylePlain目标：self.session操作：@selector（cancelAuthenticator）]

];

}

-（void）setupStackView {

UIStackView \* stack = [[UIStackView alloc] initWithFrame：CGRectMake（0，0，100，100）];

stack.distribution = UIStackViewDistributionFill; stack.alignment = UIStackViewAlignmentFill; stack.spacing = 0;

stack.backgroundColor = [UIColor blueColor]; stack.axis = UILayoutConstraintAxisVertical; [self.scrollView addSubview：stack];

stack.translatesAutoresizingMaskIntoConstraints = NO; [[self.scrollView.topAnchor约束EqualToAnchor：stack.topAnchor]

setActive：YES];

[[self.scrollView.leadingAnchor约束EqualToAnchor：stack.leadingAnchor] setActive：YES];

[[self.scrollView.trailingAnchor ConstraintsEqualToAnchor：stack.trailingAnchor] setActive：YES];

[[self.scrollView.bottomAnchor约束EqualToAnchor：stack.bottomAnchor] setActive：YES];

[[self.scrollView.widthAnchor ConstraintsEqualToAnchor：stack.widthAnchor] setActive：YES];

self.stackView =堆栈;

}

-（IBAction）onNext：（id）sender {NSMutableArray <TSXSecurityQuestionAndAnswer \*> \* questionsAndAnswers =

[[NSMutableArray alloc] init];

for（int i = 0; i <\_questionAnswerViews.count; i ++）{如果（\_questionAnswerViews [i] .answer.length）{

[questionsAndAnswers addObject：[TSXSecurityQuestionAndAnswer createAnswerToQuestionWithQuestion：\_questions [i]答案：[TSXSecurityQuestionAnswer createWithTextWithAnswerText：\_questionAnswerViews [i] .answer]]]]；

}

}

[self.session提交答案：问题和答案]；

}

* （void）viewWillAppear：（BOOL）animated {[super viewWillAppear：animated]; [self showQuestions：\_questions];

}

* （void）showQuestions：（NSArray <TSXSecurityQuestion \*> \*）questions {

\_questions =问题；

如果（self.viewLoaded）{

[self removeAllQuestions];

NSMutableArray \* viewsArr = [[NSMutableArray alloc] init]; for（int i = 0; i <questions.count; i ++）{

QuestionAnswerView \*视图= [[QuestionAnswerView分配] initWithFrame：CGRectMake（0,0,100,100）];

view.backgroundColor = [UIColor greenColor]; view.question = [questions [i] securityQuestionText]; [viewsArr addObject：view];

[self.stackView addArrangedSubview：view];

}

\_questions =问题；

\_questionAnswerViews = viewsArr;

}

}

-（void）removeAllQuestions {

如果（！\_questionAnswerViews）返回;

for（int i = 0; i <\_questionAnswerViews.count; i ++）{[self.stackView removeArrangedSubview：\_questionAnswerViews [i]];

}

\_questionAnswerViews = nil;

}

@结束

* + - * 1. **语音验证器会话**

语音身份验证器会话继承自TSXUIMultiInputAuthentication- Session，因此包括-[TSXUIMultiInputAuthenticationSession setInputStepWithStepNumber：maxStepNumber：stepDescription：]方法。对于语音，此方法通过-[-TSXMultiInputStepDescription getStepTag]以字符串常量的形式提供语音记录的状态，如下表所示。

|  |  |
| --- | --- |
| **步骤标签** | **描述** |
| voice\_000 | 有效的密码短语。 |
| voice\_tooLong | 记录的输入对于请求的密码而言太长。 |
| voice\_tooShort | 记录的输入对于请求的密码而言太短。 |
| voice\_tooLoud | 记录的输入声音太大。 |
| voice\_tooNoisy | 记录的输入有太多背景噪音。 |
| voice\_wrongPassphrase | 记录的密码短语与提供的密码短语不匹配。 |

**6.4.3.3。身份验证器UI控件**

一些认证器需要不平凡的输入。例如，面部检测认证器可能需要从面部图像提取的特征向量。此验证器的图像获取实施可能很复杂，因为它需要对获取的图像进行鉴定（例如，确保其中出现面部）并从中提取特征。为了简化自定义UI处理程序的实现，SDK提供了视图和实用程序来帮助从用户那里获取此类输入，并且足够灵活，可以集成到应用程序实现的自定义UI流程中。

**6.4.3.3。音频输入控制器**

音频输入控制器TSXAudioInputController允许获取音频，并在成功获取之后返回TSXAudioInputResponse。

要使用此控制器，请按照下列步骤操作：

1. 使用init指定的初始化程序构造它的实例。
2. 将构造实例的委托属性设置为实现

TSXAudioInputControllerDelegate协议。

1. 当应用程序准备开始记录时（通常是响应UI中的用户操作），请在控制器上调用startSampling。
2. 当应用程序准备结束记录时（通常是响应UI中的用户操作），请在控制器上调用endSampling。
3. 响应-audioInputController：responseCollected：委托在控制器上调用以收集TSXAudioInputResponse，并将其作为从promiseInput异步返回的值传递给SDK。
4. 响应-audioInputController：responseCollectionError：通知用户获取错误。

以下代码说明了语音身份验证器视图控制器将如何使用音频输入控制器来收集音频输入。

@interface VoiceAuthenticationSessionVc（）<TSXAudioInputControllerDelegate> {NSArray <NSDictionary \*> \* \_pendingChallenges;

TSXAudioInputController \* \_audioInputController;

}

@属性（弱，非原子）IBOutlet UIButton \* startButton; @属性（弱，非原子）IBOutlet UIButton \* stopButton; @property TSXAudioAcquisitionStepDescription \* curStep;

@结束

@实现语音认证会话Vc

-（void）viewDidLoad {[super viewDidLoad];

\_audioInputController = [[[TSXAudioInputController alloc] init];

\_audioInputController.delegate =自我;

}

-（void）viewDidAppear：（BOOL）动画{

//可能会在出现之前收到当前步骤-请更新UI。 if（self.curStep）{

[self startAcquisition：self.curStep];

}

}

* （IBAction）startRecordClicked：（id）sender {[\_audioInputController startSampling];

}

* （IBAction）stopRecordClicked：（id）sender {[\_audioInputController endSampling];

}

-（void）startAcquisition：（TSXAudioAcquisitionStepDescription \*）audioStep {self.hintsLabel.text = [audioStep getStepTag]; self.startButton.enabled = YES;

self.curStep = audioStep;

}

* （IBAction）cancelClicked：（id）sender {[self.session cancelAcquisition];

}

-（void）audioInputController：（TSXAudioInputController \*）ctler responseCollected：（TSXAudioInputResponse \*）response {

self.stopButton.enabled =否;

[self.session responseCollected：response];

}

-（void）audioInputController：（TSXAudioInputController \*）view responseCollectionError：（NSError \*）error {

self.stopButton.enabled =否;

UIAlertController \* errCtrler = [UIAlertController alertControllerWithTitle：@“错误”消息：[NSString stringWithFormat：@“无法获取声纹：％@”，错误] preferredStyle：UIAlertControllerStyleAlert];

[errCtrler addAction：[UIAlertAction actionWithTitle：@“重试”样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull操作）{

[\_audioInputController startSampling];

}]];

[errCtrler addAction：[UIAlertAction actionWithTitle：@“ Abort”样式：UIAlertActionStyleCancel处理程序：^（UIAlertAction \* \_Nonnull操作）{

[self cancelClicked：nil];

}]];

[自身presentViewController：errCtrler动画：没有完成：无]；

}

-（void）audioInputControllerStartedSampling：（TSXAudioInputController \*）ctler {self.stopButton.enabled = YES;

}

@结束

**6.4.3.3。相机输入视图**

摄像机输入视图TSXCamerInputView允许根据传输服务器提供的一组采集挑战来采集图像，并返回

成功获取合格图像后，TSXCameraInputResponse。该视图显示摄像机输入的预览以及视觉指示器，以帮助用户调整摄像机以成功获取。

应该创建此视图并将其添加到UI视图层次结构，就像其他任何视图一样。进入视图层次结构后，应采取以下步骤：

1. 通过在下表中设置配置属性来配置控件：

**属性**

waitForCameraAdjusted

useBackCamera

**描述**

视图是否应该在尝试获取图像之前等待操作系统发出的照相机自动对焦和曝光设置已完成的信号？

如果是，则使用设备的后置摄像头进行采样。否则，使用前置摄像头。

detectionTimeout 从满足获取挑战的图像的可用性到实际获取挑战之间要等待的时间间隔。例如，如果将其设置为1秒，则只有当摄像机预览包含满足采集挑战的图像1秒钟时，才可以实际采集样本。

showIndicators 控件上是否应该显示指示器，以帮助用户调整摄像机以成功获取？

视图中的live RectRectangle，即图像的活动部分应驻留的位置。

例如，对于人脸检测，这些是应成功放置用户的人脸的坐标。取决于

采集挑战，矩形的大小和位置可能会影响应用于摄像机输入的缩放级别。

blurMask 一个UIImage，它定义图像预览中要模糊的区域的遮罩。

indicatorColorValidInput Color用于面部查找器指示器何时

看到一张有效的脸。

indicatorColorInvalidInput Color时为人脸查找器指示器

看到一张无效的脸。

1. 将视图的委托属性设置为实现

TSXCameraInputViewDelegate协议。

1. 将视图的acquisitionChallenges属性设置为从Transmit Server接收到的采集挑战。对于面部检测，采集挑战被传递给-

setInputStepWithStepNumber：maxStepNumber：stepDescription：请求会话进行输入之前身份验证会话的方法（请参见第6.4.3.2节）。

1. 当应用程序准备好开始在控件中显示预览时，请在控件上调用startSampling。
2. 当应用程序准备好开始实际开始获取图像时，请将acquisitonEnabled属性设置为YES（直到完成此操作，在预览显示处于活动状态时，即使预览中的图像满足了获取挑战，也不会发生实际的图像获取）。
3. 如果应该中止采样，请在视图上调用cancelSampling。
4. 响应委托呼叫：
   1. -cameraInputView：responseCollected：由视图调用以提供TSXCameraInputResponse。典型的实现是将响应作为promiseInput的异步返回传递给SDK。
   2. -cameraInputView：responseCollectionError：由视图调用以通知获取过程中的错误。典型的实现将通知用户错误。
   3. -cameraInputView：hintsChanged：由视图调用，以请求应用程序向用户显示提示，以指导他完成获取过程。这些提示可以指导用户如何调整相机以成功获取。

以下代码说明了面部认证器视图控制器将如何使用图像采集控制器来收集面部样本。面部认证会话对象也包括在内。注意身份验证会话对象如何提取并向视图控制器传递获取挑战，以及如何将获取挑战传递给图像获取视图（请参阅setInputStepWithStepNumber：…）。

@interface MyFaceAuthenticationSession（）{TSXDefaultUIHandlerHostingContext \* \_hostingContext; NSArray <NSDictionary \*> \* \_acquisition挑战； MyFaceAuthenticationSessionVc \* \_viewController;

void（^ \_responseHandler）（TSXInputOrControlResponse \* \_Nonnull）;

}

@结束

@implementation MyFaceAuthenticationSession

-（void）setInputStepWithStepNumber：（long）stepNumber maxStepNumber：（long）maxStepNumber stepDescription：（TSXCameraAcquisitionStepDescription \*）stepDescription {

\_acquisitionChallenges = [stepDescription acquisitionChallenges];

}

-（void）startSessionWithDescription：（TSXAuthenticatorDescription \* \_Nonnull）描述模式：（TSXAuthenticatorSessionMode）mode actionContext：（TSXPolicyAction \*

\_Nullable）actionContext clientContext：（NSDictionary \* \_Nullable）clientContext {

\_hostingContext = [TSXDefaultUIHandler HostingContextFromClientContext：clientContext];

MyFaceAuthenticationSessionVc \* vc = [[[MyFaceAuthenticationSessionVc分配] initWithNibName：nil bundle：nil];

vc.session =自我；

\_viewController = vc;

[\_hostingContext presentViewController：\_viewController动画：是]；

}

-（void）responseCollected：（TSXCameraInputResponse \*）response {if（\_responseHandler）{

\_responseHandler（[TSXInputOrControlResponse createInputResponseWithInputResponse：response]）;

}

}

-（void）cancelAcquisition {if（\_responseHandler）{

\_responseHandler（[TSXInputOrControlResponse createControlResponseWithControlRequest：[TSXControlRequest createWithRequestType：ControlRequestTypeCancelAuthenticator]]）;;

}

}

-（void）promiseInputAsynchronouslyWithHandler：（void（^）（TSXInputOrControlResponse \*

\_Nonnull））处理程序{

\_responseHandler =处理程序;

[\_viewController startAcquisition：\_acquisitionChallenges];

}

-（void）endSession {

[\_hostingContext dismissViewController：\_viewController动画：是]；

}

@结束

@interface MyFaceAuthenticationSessionVc（）<TSXCameraInputViewDelegate> {NSArray <NSDictionary \*> \* \_pendingChallenges;

}

@属性（弱，非原子）IBOutlet UILabel \*提示标签；

@属性（弱，非原子）IBOutlet TSXCameraInputView \* cameraInputView; @结束

@implementation FaceAuthenticationSessionVc

-（void）viewDidLoad {[super viewDidLoad];

self.cameraInputView.showIndicators = YES; self.cameraInputView.detectionTimeout = 1; self.cameraInputView.acquisitionEnabled = YES;

}

-（void）viewDidAppear：（BOOL）动画{如果（\_pendingChallenges）{

[self startAcquisition：\_pendingChallenges];

\_pendingChallenges = nil;

}

}

-（void）startAcquisition：（NSArray <NSDictionary \*> \*）挑战{if（self.cameraInputView）{

self.cameraInputView.acquisitionChallenges =挑战； [self.cameraInputView startSampling];

}其他{

\_pendingChallenges =挑战；

}

}

-（IBAction）cancelClicked：（id）sender {[self.session cancelAcquisition];

}

-（void）cameraInputView：（TSXCameraInputView \*）视图响应已收集：（TSXCameraInputResponse \*）response {

[self.session responseCollected：response];

}

-（void）cameraInputView：（TSXCameraInputView \*）view responseCollectionError：（NSError \*）error {

UIAlertController \* errCtrler = [UIAlertController alertControllerWithTitle：@“错误”消息：[NSString stringWithFormat：@“无法获取人脸：％@”，错误] PreferredStyle：UIAlertControllerStyleAlert];

[errCtrler addAction：[UIAlertAction actionWithTitle：@“重试”样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull操作）{

[self.cameraInputView startSampling];

}]];

[errCtrler addAction：[UIAlertAction actionWithTitle：@“ Abort”样式：UIAlertActionStyleCancel处理程序：^（UIAlertAction \* \_Nonnull操作）{

[self cancelClicked：nil];

}]];

[自身presentViewController：errCtrler动画：没有完成：无]；

}

-（void）cameraInputView：（TSXCameraInputView \*）视图提示已更改：（NSArray <NSString \*>

\*）提示{

self.hintsLabel.text = hints.firstObject;

}

@结束

**6.4.3.3。模式输入控制**

模式输入视图TSXPatternInputView向用户显示点网格，用户可以在其上通过在点之间拖动手指来输入模式。然后，基于此模式，视图将生成TSXPatternInput对象，该对象可用于为模式验证器提供输入。

应该创建此视图并将其添加到UI视图层次结构，就像其他任何视图一样。进入视图层次结构后，应采取以下步骤：

1. 通过在下表中设置配置属性来配置控件：

**物业描述**

nDotsHorizo​​nt alGrid水平大小（以点为单位）

nDotsVertical Grid垂直大小（以点为单位）

internalSpacingHorizo​​nt al控制框之间的水平边距

和网格周长

internalSpacingVertical 控制框之间的垂直边距

和网格周长

dotSize 每个点的大小

touchRadius 点中心的半径以注册触摸

dotCenterSize 点中心大小

dotStrokeWidth 绘制点的笔划宽度

dotStrokeColor 描画点的描边颜色

dotFillOnTouchColor Dot是输入模式的一部分时填充颜色

dotFillColor Dot不属于输入模式时填充颜色

centerIsFilled 应填充点中心

连接器宽度图案连接线的宽度

连接器颜色图案连接线的颜色

1. 将视图的委托属性设置为实现

TSXPatternInputViewDelegate协议。

1. 响应委托呼叫：
   1. -patternInputView：drawingCompleteWithPattern：由视图调用以通知模式输入已完成。典型的实现是将响应作为promiseInput的异步返回传递给SDK。
   2. –patternInputViewStartedPattern：由视图调用以通知模式输入已开始。
2. 要清除模式输入视图并为输入新模式做准备，请调用-[TSXPatternInputView clearDotsAnimated：withDelay：]。

以下代码片段显示了使用TSXPatternInputView控件的模式输入视图控制器的示例实现。

@interface MyPatternAuthenticationSessionVc（）<TSXPatternInputViewDelegate>

@property（弱）MyPatternAuthenticationSession \* session;

@属性（弱，非原子）IBOutlet TSXPatternInputView \* dotInputView; @属性（弱，非原子）IBOutlet UILabel \* errorBar;

@结束

@implementation MyPatternAuthenticationSessionVc

-（void）viewDidLoad {[super viewDidLoad];

self.dotInputView.nDotsHorizo​​ntal = self.session.gridWidth; self.dotInputView.nDotsVertical = self.session.gridHeight; self.dotInputView.delegate =自我;

[self setPatternDrawDesign];

}

-（void）patternInputView：（id）发送者drawingCompleteWithPattern：（TSXPatternInput

\*）模式{

[self.session提供输入：

[TSXInputOrControlResponse createInputResponseWithInputResponse：pattern]];

}

-（void）patternInputViewStartedPattern：（id）sender {[self setPatternDrawDesign];

}

-（void）setPatternDrawDesign {self.dotInputView.dotStrokeColor = [UIColor orangeColor]; self.dotInputView.connectorColor = [UIColor grayColor];

self.dotInputView.dotFillOnTouchColor = [[[UIColor orangeColor] colorWithAlphaComponent：0.6];

}

-（void）setErrorPrompt：（NSString \*）error {self.errorBar.hidden = NO; self.errorBar.text =错误;

self.dotInputView.dotFillOnTouchColor = [UIColor redColor]; self.dotInputView.connectorColor = [UIColor redColor];

}

@结束

* + 1. **实施信息和确认处理程序**

当SDK想要显示信息屏幕或确认屏幕以响应信息或确认操作时，它将调用-[TSXUIHandler get- InformationResponseWithTitle：text：continueText：actionContext：clientContext：异步lyWithHandler：]或-[TSXUIHandler getConfirmationInputWithTitle：text：continueText ：取消-

分别为文本：actionContext：clientContext：asynclyWithHandler：]。这两种方法均应根据参数呈现UI

提供给他们，并在用户解雇消息后异步返回。以下代码显示了这两个调用的典型实现：

@implementation TSXUIHandler

-（void）getConfirmationInputWithTitle：（NSString \*）title

文本：（NSString \*）文本continueText：（NSString \*）continueText cancelText：（NSString \*）cancelText

actionContext：（TSXPolicyAction \*）actionContext clientContext：（NSDictionary \*）clientContext

ynchronousWithHandler：（void（^）（TSXConfirmationInput \*

\_Nonnull））处理程序

{

UIAlertController \* alert = [UIAlertController

alertControllerWithTitle：标题消息：文本

preferredStyle：UIAlertControllerStyleActionSheet];

[alert addAction：[UIAlertAction actionWithTitle：continueText

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull

动作）{

createWithUserChoice：0]）;

}]];

处理程序（[TSXConfirmationInput

[alert addAction：[UIAlertAction actionWithTitle：cancelText

样式：UIAlertActionStyle破坏性处理程序：^（UIAlertAction \* \_Nonnull

动作）{

createWithUserChoice：1]）;

}]];

处理程序（[TSXConfirmationInput

[[TSXDefaultUIHandler HostingContextFromClientContext：clientContext] presentAlertController：alert];

}

-（void）getInformationResponseWithTitle：（NSString \*）title

文本：（NSString \*）文本continueText：（NSString \*）continueText

actionContext：（TSXPolicyAction \*）actionContext clientContext：（NSDictionary \*）clientContext

ynchronousWithHandler：（void（^）（TSXConfirmationInput \*

\_Nonnull））处理程序

{

UIAlertController \* alert = [UIAlertController

alertControllerWithTitle：标题消息：文本

preferredStyle：UIAlertControllerStyleActionSheet];

[alert addAction：[UIAlertAction actionWithTitle：continueText样式：UIAlertActionStyleDefault

handler：^（UIAlertAction \* \_Nonnull action）{handler（[TSXConfirmationInput createWithUserChoice：-1]）;

}]];

[[TSXDefaultUIHandler HostingContextFromClientContext：clientContext] presentAlertController：alert];

}

@结束

* + 1. **实现JSON数据处理程序**

当策略请求执行JSON数据操作（通过SDK将JSON文档发送到应用程序）时，SDK通过调用UI处理程序的-[-TSXUIHandler processJsonDataWithJsonData：actionContext：client- Context：异步lyWithHandler：]方法进行响应JSON数据。

仅在此方法异步返回TSXJsonDataProcessingResult的实例之后，策略执行才会恢复。您可以使用+ [-TSXJsonDataProcessingResult createWithContinueProcessing：]创建结果的实例，为true可以恢复策略，而为false可以基于JSON处理中止该结果。实施

该调用的完全取决于特定的应用程序需求。以下代码提供了有关如何实现此回调的示例：

-（void）processJsonDataWithJsonData：（NSDictionary \*）jsonData

actionContext：（TSXPolicyAction \*）actionContext clientContext：（NSDictionary \*）clientContext

ynchronousWithHandler：（void（^）（TSXJsonDataProcessingResult \*

\_Nonnull））处理程序{

handler（[TSXJsonDataProcessingResult createWithContinueProcessing：YES]）;

}

* + 1. **处理会话拒绝**

当SDK需要通知应用程序有关策略拒绝的信息时，将调用UI处理程序的-[TSXUIHandler handlePolicyRejectionWithTitle：text：buttonText：failureData：actionContext：clientContext：异步地使用Handler：]方法。只有在

此调用的异步返回。典型的实现方式是根据传递给该函数的参数，提供UI通知用户有关拒绝的信息。以下代码提供了有关如何实现此回调的示例：

-（void）handlePolicyRejectionWithTitle：（NSString \*）标题文本：（NSString \*）文本buttonText：（NSString \*）buttonText rejectData：（NSDictionary \*）rejectionData actionContext：（TSXPolicyAction \*）actionContext clientContext：（NSDictionary

\*）clientContext异步WithHandler：（void（^）（TSXConfirmationInput \*

\_Nonnull））处理程序{

UIAlertController \* alert = [UIAlertController

alertControllerWithTitle：标题消息：文本

preferredStyle：UIAlertControllerStyleActionSheet];

[alert addAction：[UIAlertAction actionWithTitle：buttonText

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull

动作）{

handler（[TSXConfirmationInput createWithUserChoice：-1]）;

}]];

[[TSXDefaultUIHandler HostingContextFromClientContext：clientContext] presentAlertController：alert];

}

* + 1. **实施TOTP会话接口**

当应用程序请求启动TOTP生成会话时（请参见第5.8节），SDK将通过执行身份验证过程来启动-调用第6.4.2节和6.4.3节中所述的身份验证调用。成功完成身份验证后，SDK将启动TOTP显示会话。 SDK通过调用-[TSXUIHandler createTotpGenerationSessionWithUser- Id：generatorName：]来要求UI处理程序创建类型为TSXUITotpGenerationSession的TOTP会话对象，该对象将用于管理TOTP生成流程。此方法接收应为其创建会话的TOTP生成器名称。应该实现它以返回TOTP会话对象，该对象是TSXUITotpGenerationSession接口的实现，准备与该生成器一起使用。

创建会话并将其返回到SDK后，SDK将调用-[TSXUITotp- GenerationSession startSessionWithTotpGenerationSessionServices：actionContext：clientContext：]来指示用于呈现

应该设置TOTP。此方法还接收客户端上下文，操作上下文（如果TOTP生成是由策略操作启动的）和TSXTotpGenerationSession- Services对象，该对象充当自定义UI处理程序的接口，以在会话内的SDK上调用与TOTP会话相关的操作。

UI处理程序应使用-[TSXTotpGenerationSessionServices startCode-Generation]来启动TOTP生成。会话接口包含方法-[TSXUITotpGenerationSession promiseChallengeInputAsynchronously- WithHandler：]，如果该方法配置为包括以下方法，则该方法将由SDK调用：当预计会话将从身份验证客户端获取质询以包括在身份验证代码生成中时挑战。一旦调用-[TSXTotp- GenerationSessionServices startCodeGeneration]方法，

在TOTP生成会话终止之前，SDK将定期调用

-[TSXUITotpGenerationSession setTotpCodeWithCode：ttlSeconds：maintenanceSeconds：]通知会话当前的OTP代码及其到期信息。 UI处理程序应根据传递给此调用的参数更新UI以反映TOTP代码。

在调用TOTP生成请求附带的文本消息的情况下，SDK可以可选地在调用-[TSXUITotpGenerationSession setMessage：]之前调用-[TSXUITotpGenerationSession setMessage：]-例如，可以提取伴随TOTP生成请求的文本消息。

来自TOTP挑战。这取决于使用中的特定TOTP提供程序及其配置。

当UI处理程序希望终止TOTP会话时（通常响应于用户退出TOTP生成UI的操作），它应调用-[TSXTotp-

GenerationSessionServices finishSession]调用。这将终止TOTP会话并调用-[TSXUITotpGenerationSession endSession]，应将其实现为拆除为TOTP显示设置的UI。

* + 1. **实施设备管理会话界面**

SDK的设备管理功能使用户可以查看已为该应用程序注册（绑定）的设备的列表，接收有关设备的信息并对其执行操作（例如删除设备或启用/禁用它们） ）。设备管理在设备管理会话中进行。会话在应用程序调用-[TSXTransmitSDKXm startDevice- ManagementSessionWithClientContext：ynchronlyWithHandler：]时开始（请参见第5.6节），并在用户决定终止它时结束（这将导致该方法异步返回）。

SDK将调用TSXUIHandler方法-[TSXUIHandler create- DeviceManagementSessionWithUserId：]。预期此方法将返回从TSXUIDeviceManagementSession派生的对象，该对象由自定义UI处理程序实现。 SDK将使用此会话对象与UI进行交互

整个会话中的处理程序。会话生命周期始终始于对-[-TSXUIDeviceManagementSession startSessionWithDeviceManagement- SessionServices：actionContext：clientContext：]的调用，该会话在创建会话对象后不久由SDK完成。会话通过此调用接收用于在其中创建会话的SDK操作调用的客户端上下文，以及描述触发该会话创建的策略操作（如果有）的操作上下文。此外，SDK将TSXDeviceManagementSessionServices对象传递到此会话。该对象由SDK实现，并用作应用程序在设备管理会话期间执行操作的接口。

设备管理会话在为用户注册的设备列表上进行操作。此列表可以在会话生存期期间更新，例如在设备上执行操作时。更新列表（或列表中的设备）后，SDK会调用会话对象的[TSXUIDeviceManagementSession setSessionDevicesList：]方法。这种情况至少发生在会话开始之前，并且可能在调用之间

应用程序提供的设备服务。应用程序还可以通过调用-[TSXDeviceManagementSessionServices requestRefreshDevicesAsynchronouslyWithHandler：]来启动对当前设备列表的刷新。设备列表由TSXManagedDevice对象的集合组成。这些对象封装

基于设备的当前状态的设备信息以及可以在该设备上执行的操作的列表。通过这些对象的availableActions属性可以使用此功能。下表列出了对此列表有效的值及其对应的TSXDeviceManagementSessionServices方法和说明。

availableActions Valu eTSXDeviceManagementSession-

服务

服务

DeviceManagementActionRemove - [TSXDeviceManagementSession-

服务removeDeviceWith-设备：clientContext：asynclyWithHandler：]

从用户的已注册设备列表中删除该设备，并取消注册。这个

可能无法在当前设备上调用。

DeviceManagementActionIdentify - [TSXDeviceManagementSession-

服务identifyDevice-WithDevice：clientContext：asynclyWithHandler：]

向设备发送推送通知以进行标识。

DeviceManagementActionRename - [TSXDeviceManagementSession-

服务renameDeviceWith-设备：newName：clientContext：asynclyWithHandler：]

重命名设备。

**注意：上面** 列出的所有操作只能在通过-[TSXUIDeviceManagementSession setSessionDevicesList：]提供给会话的设备对象上调用。

对列表中的设备所做的更改将就地传播到设备对象。表示通过对-[TSXUIDeviceManagementSession setSessionDevicesList：]的调用来更新的列表可以保证具有由相同对象表示的更新列表和先前列表所共有的设备。这意味着当

在设备（包括引用的设备）上进行操作之后，只要未删除设备，对该设备对象保留的引用将保持有效（并且该对象保持最新）。

当应用程序确定设备管理会话完成时（例如，要求用户关闭设备管理UI），它将调用-[TSXDevice- ManagementSessionServices finishSession]服务。当SDK或应用程序决定结束会话时，SDK将调用-[TSXUIDevice- ManagementSession endSession]。

该应用程序可以使用设备管理会话服务对象来注销当前设备。不能仅通过调用-[TSXDevice-

当前设备上的ManagementSessionServices removeDeviceWithDevice：clientContext：异步lyWithHandler：]，因为这只会导致错误。相反，必须调用-[TSXDeviceManagementSessionServices removeCurrent- DeviceAndFinishSessionWithClientContext：ynchronousWithHandler：]来实现此目的。实现此功能是为了处理删除当前设备的特殊情况，并在执行此操作后应有序地清理SDK状态。当此函数成功异步返回时，SDK已从服务器请求删除设备。异步结果处理程序完成后，SDK将继续完成当前设备管理会话，就像调用了[[TSXDeviceManagementSessionServices finishSession]。然后，SDK将清除当前的主会话并删除与该用户关联的所有持久数据。这意味着在执行此操作之后，SDK仍然没有登录用户，并且其设备刚刚注销的用户不再绑定当前设备。

以下代码示例演示了如何实现设备管理UI处理程序的自定义。我们首先实现-[TSXUIHandler createDeviceManagementSessionWithUserId：]，如下所示：

@implementation MyCustomUIHandler

-（TSXUIDeviceManagementSession \*）createDeviceManagementSessionWithUserId：（NSString

\*）用户身份 {

返回[[MyDeviceManagementSession alloc] init]；

}

@结束

此方法的实现只是为TSXUIDeviceManagementSession接口创建并返回我们自己的实现。初始化此实现的实例后，我们只需创建一个视图控制器即可显示一组设备并提供控件以对它们执行操作–如以下代码所示：

@interface MyDeviceManagementSession：TSXUIDeviceManagementSession（）{TSXDefaultUIHandlerHostingContext \* \_hostingContext; MyDeviceManagementViewController \* \_vc;

}

@结束

@implementation MyDeviceManagementSession

-（instancetype）init {self = [super init];如果（自己）{

\_vc = [[MyDeviceManagementViewController alloc] init];

}

返回自我

}

会话开始时，我们记录客户端上下文以及为此会话收到的TSXDevice-ManagementSessionServices对象（然后，我们将使用该对象在设备上执行操作）。然后，我们介绍设备管理视图控制器。

-（void）startSessionWithDeviceManagementSessionServices：（TSXDeviceManagementSessionServices \* \_Nonnull）deviceManagementSessionServices

actionContext：（TSXPolicyAction \*

\_Nullable）actionContext

clientContext：（NSDictionary \*）clientContext

{

\_hostingContext = [TSXDefaultUIHandler HostingContextFromClientContext：clientContext];

\_vc.deviceManagementSessionServices = deviceManagementSessionServices; [\_hostingContext presentViewController：\_vc动画：是]；

}

当调用-[TSXUIDeviceManagementSession setSessionDevicesList：]调用时，我们更新视图控制器显示的设备集（这将触发其刷新）：

-（void）setSessionDevicesList：（NSArray <TSXManagedDevice \*> \* \_Nonnull）设备{

\_vc.devices =设备；

}

最后，当调用-[TSXUIDeviceManagementSession endSession]时，我们将关闭视图控制器：

-（void）endSession {

[\_hostingContext dismissViewController：\_vc动画：是]；

}

@结束

我们的视图控制器实现基本上包括一个常规的表视图，该视图设置为显示设备列表：

@interface MyDeviceManagementViewController（）<UITableViewDataSource，UITableViewDelegate> {

NSArray <TSXManagedDevice \*> \* \_devices;

}

@属性（弱，非原子）IBOutlet UITableView \* devicesTable; @结束

@implementation MyDeviceManagementViewController

-（void）viewWillAppear：（BOOL）动画{[self.devicesTable reloadData];

}

-（UITableViewCell \*）tableView：（UITableView \*）tableView cellForRowAtIndexPath：（NSIndexPath \*）indexPath {

UITableViewCell \* cell = [[UITableViewCell分配] initWithStyle：UITableViewCellStyleSubtitle

repeatIdentifier：nil]; cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator; TSXManagedDevice \* device = [\_devices objectAtIndex：[indexPath row]]; cell.textLabel.text = device.info.name？ device.info.name：@“（未命名的设备）”; cell.detailTextLabel.text = [NSString stringWithFormat：@“％@％@”，

device.info.osType，device.info.osVersion];返回单元

}

-（NSInteger）tableView：（UITableView \*）tableView numberOfRowsInSection：（NSInteger）节{

返回\_devices.count;

}

-（NSInteger）numberOfSectionsInTableView：（UITableView \*）tableView {返回1;

}

-（void）viewDidLoad {[super viewDidLoad];

self.navigationItem.title = @“管理设备”; self.navigationItem.leftBarButtonItem = [[UIBarButtonItem分配]

initWithTitle：@“后退”样式：UIBarButtonItemStyleDone目标：自我操作：@selector（backButtonPressed）];

[self.devicesTable.topAnchor约束EqualToAnchor：self.topLayoutGuide.bottomAnchor] .active = true;

}

单击列表中的设备条目后，我们将打开一个详细视图控制器，显示该设备并允许对其执行操作：

-（void）tableView：（UITableView \*）tableView didSelectRowAtIndexPath：（NSIndexPath

\*）indexPath {

TSXManagedDevice \* device = [\_devices objectAtIndex：[indexPath row]];

MyDeviceManagementDetailsViewController \* detailsVc = [[[MyDeviceManagementDetailsViewController alloc] initWithNibName：nil bundle：nil];

detailsVc.managedDevice =设备；

detailsVc.deviceManagementSessionServices = self.deviceManagementSessionServices;

[self.navigationController pushViewController：detailsVc动画：是]；

}

当设备管理会话对象更新设备集时，我们触发刷新显示的设备表：

-（NSArray <TSXManagedDevice \*> \*）devices {return \_devices;

}

-（void）setDevices：（NSArray \*）devices {

\_devices =设备； [self.devicesTable reloadData];

}

当用户单击“后退”按钮表示他已完成设备管理功能时，**我们要求S**DK结束会话：

-（void）backButtonPressed {[self.deviceManagementSessionServices finishSession];

}

@结束

实现了详细信息视图控制器，以使其在加载时创建时显示有关为其分配的设备的信息。

@interface MyDeviceManagementDetailsViewController（）{UITapGestureRecognizer \* \_tapRecognizer;

}

@属性（弱，非原子）IBOutlet UITextView \* deviceNameEdit; @属性（弱，非原子）IBOutlet UILabel \* deviceDescriptionLabel; @属性（弱，非原子）IBOutlet UILabel \* deviceStatusLabel; @属性（弱，非原子）IBOutlet UILabel \* deviceRegisteredOnLabel; @属性（弱，非原子）IBOutlet UILabel \* deviceLastAccessLabel; @属性（弱，非原子）IBOutlet UILabel \* deviceUseTimesLabel; @属性（弱，非原子）IBOutlet UILabel \* currentDeviceIndicator; @属性（弱，非原子）IBOutlet UIButton \* renameButton;

@属性（弱，非原子）IBOutlet UIButton \* identifyButton; @属性（弱，非原子）IBOutlet UIButton \* removeButton;

@结束

@implementation MyDeviceManagementDetailsViewController

-（void）viewDidLoad {[super viewDidLoad];

self.navigationItem.title = @“设备详细信息”;

}

-（void）viewWillAppear：（BOOL）动画{[self updateDeviceData];

}

-（void）updateDeviceData {

self.deviceNameEdit.text = self.managedDevice.info.name？ self.managedDevice.info.name：@“（未命名的设备）”;

self.deviceDescriptionLabel.text = [NSString stringWithFormat：@“％@;％@％@”，

[self.managedDevice.info.model，self.managedDevice.info.osType，self.managedDevice.info.osVersion]；

开关（self.managedDevice.info.status）{case DeviceStatusDisabled：

self.deviceStatusLabel.text = @“设备已禁用”。打破;

案例DeviceStatusRemoved：

self.deviceStatusLabel.text = @“设备已删除。”;打破;

案例DeviceStatusRecentlyUsed：

self.deviceStatusLabel.text = @“最近使用过的设备。”;打破;

案例DeviceStatusLongInactivity：

self.deviceStatusLabel.text = @“该设备已长时间不使用。”打破;

案例DeviceStatusNoRecentActivity：

self.deviceStatusLabel.text = @“该设备最近未使用。”;打破;

默认：

self.deviceStatusLabel.text = @“未知设备状态”。打破;

}

self.deviceRegisteredOnLabel.text = [NSString stringWithFormat：@“注册于

％@“，self.managedDevice.info.registered]；

self.deviceLastAccessLabel.text = [NSString stringWithFormat：@“最近一次访问

来自（％f，％f）的％@”，

self.managedDevice.info.lastAccess，self.managedDevice.info.lastAccessLocation.longitude，

self.managedDevice.info.lastAccessLocation.latitude

];

self.deviceUseTimesLabel.text = [NSString stringWithFormat：@“使用％ld次”，self.managedDevice.info.useCount];

self.currentDeviceIndicator.hidden =！self.managedDevice.info.isCurrent;

在更新设备详细信息的过程中，视图控制器还根据可用的设备操作启用或禁用用于调用操作的UI元素：

self.renameButton.enabled = self.managedDevice.availableActions＆DeviceManagementActionRename;

self.identifyButton.enabled = self.managedDevice.availableActions＆DeviceManagementActionIdentify;

self.removeButton.enabled = self.managedDevice.availableActions＆DeviceManagementActionRemove;

}

最后，视图控制器响应于用户操作调用各种设备操作。请注意，当这些操作异步返回时，视图如何从先前分配给它的设备对象中重新加载信息。还请注意有多特别

对删除设备操作进行了处理，以识别何时在当前设备上调用该设备并进行相应的更新。

-（IBAction）startRenameDevice：（id）sender {self.deviceNameEdit.editable = YES; [self.deviceNameEdit成为FirstResponder];

\_tapRecognizer = [[UITapGestureRecognizer分配] initWithTarget：自我行动：@selector（endRename）];

[\_tapRecognizer setCancelsTouchesInView：YES]; [self.view addGestureRecognizer：\_tapRecognizer];

self.identifyButton.enabled =否; self.removeButton.enabled =否；

}

-（IBAction）identifyDevice：{id）sender {

[self.deviceManagementSessionServices IdentificationDeviceWithDevice：self.managedDevice

异步lyWithHandler：^（BOOL res，TSXAuthenticationError \* \_Nonnull err）{

if（[self processNonDestructiveAsyncOpResult：res

withError：err]）{

[self showMessage：@“推送通知具有

已发送到所选设备。“ withTitle：@”识别设备“];

}

}];

}

-（IBAction）removeDevice：（id）sender {if（！self.managedDevice.info.isCurrent）{

[self.deviceManagementSessionServices removeDeviceWithDevice：self.managedDevice

clientContext：nil

异步lyWithHandler：^（BOOL res，TSXAuthenticationError \* \_Nonnull err）{

withError：err];

}其他{

}];

[自身processNonDestructiveAsyncOpResult：res

[self showMessage：@“您将要从帐户中取消当前设备的绑定。您将退出当前会话，并且用户设置和身份验证器将被删除。要再次使用此设备，必须重新注册使用该服务。单击“确定”继续，或单击“取消”中止此操作。”

withTitle：@“删除当前设备” withCancel：是

然后：^ {

[self.deviceManagementSessionServices removeCurrentDeviceAndFinishSessionWithClientContext：nilynchronlyWithHandler：^（BOOL res，TSXAuthenticationError \* \_Nul lable err）{

if（err）{

[self showMessage：[错误描述] withTitle：@“删除

当前设备“];

}其他{

[self.navigationController

PopViewControllerImated:否]；

}

}];

}];

}

}

-（BOOL）processNonDestructiveAsyncOpResult：（BOOL）res withError：（TSXAuthenticationError \*）err {

if（err）{

[self showMessage：[NSString stringWithFormat：@“％@”，err.description] withTitle：@“ Error”]；

返回否；

}其他{

[self updateDeviceData];返回是；

}

}

-（void）endRename {

[self.deviceNameEdit endEditing：YES]; self.deviceNameEdit.editable = NO;

[self.view removeGestureRecognizer：\_tapRecognizer];

\_tapRecognizer = nil;

if（！[self.deviceNameEdit.text isEqualToString：self.managedDevice.info.name]）{[self.deviceManagementSessionServices

namedDeviceWithDevice：self.managedDevice

newName：self.deviceNameEdit.textynchronousWithHandler：^（BOOL res，

TSXAuthenticationError \* \_\_Nonnull err）{

[自身processNonDestructiveAsyncOpResult：res

withError：err];

}其他{

[self updateDeviceData];

}];

[self updateDeviceData];

}

}

-（void）showMessage：（NSString \*）错误withTitle：（NSString \*）title

withCancel：（BOOL）取消

然后：（void（^）（void））completion {

UIAlertController \* errCtler = [UIAlertController alertControllerWithTitle：title

讯息：错误

preferredStyle：UIAlertControllerStyleAlert];

弱的UIAlertController \* weakAlert = errCtler; [errCtler addAction：[UIAlertAction actionWithTitle：@“ OK”

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull action）{[weakAlert dismissViewControllerAnimated：YEScomplete：nil]; complete（）;

}]];

如果（取消）{

[errCtler addAction：[UIAlertAction actionWithTitle：@“ Cancel”样式：UIAlertActionStyleCancel处理程序：^（UIAlertAction \* \_Nonnull操作）{

[weakAlert dismissViewControllerAnimated：YES完成：无]；

}]];

}

[自身presentViewController：errCtler动画：是完成：无]；

}

-（void）showMessage：（NSString \*）错误withTitle：（NSString \*）title {

[self showMessage：error withTitle：title withCancel：NO andThen：nil];

}

@结束

* + 1. **实施批准管理会话界面**

有了移动设备批准，传输平台允许将移动设备上的OOB批准作为身份验证策略的一部分。当需要OOB批准时，批准请求将输入合格设备的批准队列。设备上的应用程序可以请求SDK启动批准会话，在该会话中，设备队列中的未决批准将呈现给用户，并使用户有机会批准或拒绝它们。会话从对SDK的请求开始（请参阅第5.5节），并在用户或SDK决定终止它时结束。

SDK将调用TSXUIHandler方法-[TSXUIHandler create- ApprovalsSessionWithUserId：]。预期此方法将返回由TSXUIApprovalsSession派生的对象，该对象由自定义UI处理程序实现。 SDK将使用此会话对象在整个会话期间与UI处理程序进行交互。会话生命周期始终始于对-[TSXUIApprovals-会话startSessionWithApprovalManagementSessionServices：actionContext：clientContext：]的调用，该会话在创建会话对象后不久就由SDK完成。会话通过此调用接收用于在其中创建会话的SDK操作调用的客户端上下文，以及描述触发该会话创建的策略操作（如果有）的操作上下文。此外，SDK将TSXApprovalManagementSessionServices对象传递到此会话。该对象由SDK实施，并用作应用程序的接口，以在批准管理会话期间执行操作，例如批准

或拒绝批准请求。批准会话在设备上待处理的待批准列表上进行操作。此列表可能会在会话生存期内更新

例如批准被确认或拒绝。列表更新后，SDK会调用会话对象的-[TSXUIApprovalsSession setSessionApprovals- List：]方法。这种情况至少发生在会话开始之前，并且可能在应用程序进行的批准服务调用之间发生。应用程序还可以通过调用-[TSXApproval-ManagementSessionServices requestRefreshApprovalsAsynchronously-WithHandler：]启动当前批准列表的刷新。

批准列表包含TSXManagedMobileApproval对象的集合。这些对象根据其当前状态封装有关批准的信息以及可以在其上执行的操作的列表。通过这些对象的availableActions属性可以使用此功能。下表列出了有效的值

此列表及其对应的TSXApprovalManagementSessionServices

方法和说明。

availableActions值

TSXApprovalManagementSession-服务服务

MobileApprovalActionApprove - [TSXApprovalManagementSession-

服务approveWithApproval：clientContext：异步-WithHandler：]

要求确认此批准请求。该调用可以触发策略调用，在此期间可以调用其他UI处理程序回调（例如，执行身份验证操作）。此调用的客户端上下文参数允许UI处理程序为将在此策略调用的上下文中调用的UI处理程序调用提供自己的客户端上下文。如果这

如果client context为null，则将使用与批准会话相同的上下文。

MobileApprovalActionDeny - [TSXApprovalManagementSession-

服务denyWithApproval：asynchronousWithHandler：]

请求拒绝此批准请求。

**注意：只能** 对通过-[TSXUIApprovalsSession setSession- ApprovalsList：]提供给此会话的批准对象调用操作。

当应用程序确定批准管理会话已完成时（例如，要求用户关闭批准管理UI），它将调用-[TSXApproval-

ManagementSessionServices finishSession]服务。当SDK或应用程序决定结束会话时，SDK将调用-[TSXUIApprovals- Session endSession]。

以下代码示例演示了如何实现批准管理UI处理程序的自定义。我们首先实现-[TSXUIHandler createApprovalsSessionWithUserId：]，如下所示：

@implementation MyCustomUIHandler

-（TSXUIApprovalsSession \*）createApprovalsSessionWithUserId：（NSString \*）userId {return [[MyApprovalsSession alloc] init];

}

此方法的实现只是为TSXUIApprovalsSession接口创建并返回我们自己的实现。初始化此实现的实例后，我们只需创建一个UI，即可显示一组设备并提供控件以对其执行操作-如下代码所示：

@interface MyApprovalsSession（）{TSXDefaultUIHandlerHostingContext \* \_hostingContext; MyApprovalsListViewController \* \_vc;

}

@结束

@implementation MyApprovalsSession

-（instancetype）init {self = [super init];如果（自己）{

\_vc = [[[MyApprovalsListViewController alloc] init];

}

返回自我

}

会话开始时，我们记录客户端上下文以及为此会话收到的TSXApproval-ManagementSessionServices对象（然后，我们将使用该对象对批准执行操作）。然后，我们显示批准UI。

-（void）startSessionWithApprovalManagementSessionServices：（TSXApprovalManagementSessionServices \*）approvalManagementSessionServices

actionContext：（TSXPolicyAction \*）actionContext clientContext：（NSDictionary

\*）clientContext {

\_vc.approvalServices =审批管理会话服务; [\_hostingContext presentViewController：\_vc动画：是]；

}

当调用-[TSXUIApprovalsSession setSessionApprovalsList：]调用时，我们更新视图控制器显示的批准集（这将触发其刷新）：

-（void）setSessionApprovalsList：（NSArray <TSXManagedMobileApproval \*> \*）approvals {

\_vc.approvals =批准；

}

@结束

最后，当调用-[TSXUIApprovalsSession endSession]时，我们将关闭UI：

-（void）endSession {

[\_hostingContext dismissViewController：\_vc动画：是]；

}

我们的UI实施基本上包括一个常规的表格视图，该表格设置为显示批准列表：

@interface MyApprovalsListViewController：UIViewController（）{NSArray <TSXManagedMobileApproval \*> \* \_approvals;

}

@属性（弱，非原子）IBOutlet UITableView \* approvalsTable;

@property TSXApprovalManagementSessionServices \* approvalServices; @属性NSArray <TSXManagedMobileApproval \*> \*批准；

@结束

@implementation MyApprovalsListViewController

-（void）viewWillAppear：（BOOL）动画{[self.approvalsTable reloadData];

}

-（void）viewDidLoad {[super viewDidLoad];

self.navigationItem.title = @“批准”; self.navigationItem.leftBarButtonItem = [[UIBarButtonItem分配]

initWithTitle：@“后退”样式：UIBarButtonItemStyleDone目标：自我操作：@selector（backButtonPressed）];

self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc] initWithBarButtonSystemItem：UIBarButtonSystemItemRefresh target：self action：@selector（refreshButtonPressed :)];

[self.approvalsTable.topAnchor约束EqualToAnchor：self.topLayoutGuide.bottomAnchor] .active = true;

}

-（NSInteger）numberOfSectionsInTableView：（UITableView \*）tableView {返回1;

}

-（NSInteger）tableView：（UITableView \*）tableView numberOfRowsInSection：（NSInteger）节{

返回\_approvals.count;

}

-（UITableViewCell \*）tableView：（UITableView \*）tableView cellForRowAtIndexPath：（NSIndexPath \*）indexPath {

UITableViewCell \* cell = [[UITableViewCell分配] initWithStyle：UITableViewCellStyleSubtitle

repeatIdentifier：nil]; cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator; TSXManagedMobileApproval \*批准= [\_approvals objectAtIndex：[indexPath row]]; cell.textLabel.text =批准.approval.title;

NSString \* statusText = @“未知状态”;开关（approval.approval.status）{

case MobileApprovalStatusApproved：statusText = @“已批准”;打破;

案例MobileApprovalStatusDenied：

statusText = @“拒绝”；打破;

case MobileApprovalStatusExpired：statusText = @“ Expired”;打破;

case MobileApprovalStatusPending：statusText = @“待处理”;打破;

}

cell.detailTextLabel.text = [NSString stringWithFormat：@“％@;创建于％@”，

statusText，approval.approval.creationTime]；

返回单元

}

-（IBAction）refreshButtonPressed：（id）sender {

[self.approvalServices requestRefreshApprovalsAsynchronouslyWithHandler：^（BOOL r，TSXAuthenticationError \* e）{

NSLog（@“刷新批准完成。”）;

}];

}

@结束

单击列表中的批准条目时，我们将打开一个详细的UI，其中显示了批准并允许对其执行操作：

-（void）tableView：（UITableView \*）tableView didSelectRowAtIndexPath：（NSIndexPath

\*）indexPath {

TSXManagedMobileApproval \*批准= [\_approvals objectAtIndex：[indexPath row]];

MyApprovalViewController \* approvalVc = [[MyApprovalViewController alloc] initWithNibName：nil bundle：nil];

ApprovalVc.approvalServices = self.approvalServices;批准vc.approval =批准;

[self.navigationController pushViewController：approvalVc动画：是]；

}

当批准会话对象更新批准集时，我们触发刷新显示的表：

-（void）setApprovals：（NSArray <TSXManagedMobileApproval \*> \*）批准{

\_approvals =批准； [\_approvalsTable reloadData];

}

-（NSArray <TSXManagedMobileApproval \*> \*）approvals {return \_approvals;

}

当用户单击“后退”按钮表示他已完成设备管理功能时，**我们要求**SDK结束会话：

-（void）backButtonPressed {[self.approvalServices finishSession];

}

@结束

实现详细的UI，以使其显示有关在创建加载时分配给它的批准的信息。

@interface MyApprovalViewController（）{TSXManagedMobileApproval \* \_approval;

}

@属性（弱，非原子）IBOutlet UITextView \* approvalDetailsText; @属性（弱，非原子）IBOutlet UIButton \* approveButton; @属性（弱，非原子）IBOutlet UIButton \* denyButton;

@结束

@implementation MyApprovalViewController

-（void）setApproval：（TSXManagedMobileApproval \*）approval {

\_approval =批准； [self updateApprovalData];

}

-（TSXManagedMobileApproval \*）approval {return \_approval;

}

-（void）viewDidLoad {[super viewDidLoad];

[self updateApprovalData]; [self.approvalDetailsText.topAnchor

ConstraintsEqualToAnchor：self.topLayoutGuide.bottomAnchor常数：48] .active = YES;

}

-（void）updateApprovalData {

self.navigationItem.title = \_approval.approval.title; self.approvalDetailsText.text = \_approval.approval.details;

在更新批准详细信息的过程中，视图控制器还基于可用的设备批准来启用或禁用用于调用操作的UI元素：

self.approveButton.enabled =（\_approval.availableActions＆MobileApprovalActionApprove）;

self.denyButton.enabled =（\_approval.availableActions＆MobileApprovalActionDeny）;

}

-（void）showErrorAlert：（TSXAuthenticationError \*）error {

UIAlertController \*警报= [UIAlertController alertControllerWithTitle：@“错误”

讯息：[错误

描述]

preferredStyle：UIAlertControllerStyleAlert];

[alert addAction：[UIAlertAction actionWithTitle：@“ OK”

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull

动作）{

}]];

[自己的presentViewController：警报动画：是完成：无]；

}

最后，UI响应用户操作调用批准确认或拒绝操作。

-（IBAction）approveApproval：（id）sender {

[self.approvalServices approveWithApproval：self.approval clientContext：nillyWithWithHandler：^（TSXAuthenticationResult \* \_Nonnull res，TSXAuthenticationError \* \_Nullable err）{

if（err）{

[自我显示错误警报：错误]；

}其他{

[self.navigationController popViewControllerAnimated：YES];

}

}];

}

-（IBAction）denyApproval：（id）sender {[self.approvalServices denyWithApproval：self.approval

异步lyWithHandler：^（BOOL res，TSXAuthenticationError \*

\_Nullable err）{

if（err）{

[自我显示错误警报：错误]；

}其他{

[self.navigationController popViewControllerAnimated：YES];

}

}];

}

@结束

* + 1. **实施配置菜单会话界面**

如5.7节所述，SDK提供了一项服务，该服务使用户可以查看其注册的身份验证器，注销或重新注册它们，并选择默认的身份验证器。当应用程序调用此服务时，SDK将调用方法-[TSXUIHandler createAuthenticationConfiguration- SessionWithUserId：]。预期此方法将返回从TSXUIAuthenticationConfigurationSession派生的对象，该对象由自定义UI处理程序实现。 SDK将使用此会话对象在整个会话期间与UI处理程序进行交互。会话生命周期始终始于对-[TSXUIAuthenticationConfigurationSession startSessionWith- AuthenticationConfigurationSessionServices：actionContext：

clientContext：]，由SDK在创建会话对象后不久完成。通过此调用，会话将在其中接收SDK操作调用的客户端上下文

创建会话的行为，以及描述触发该会话创建的策略操作（如果有）的操作上下文。此外，SDK还通过了TSXAuthentication-

ConfigurationSessionServices对象与此调用有关的会话。此对象由SDK实现，并用作应用程序执行的接口

配置会话期间的操作，例如注册或注销身份验证器。配置会话对设备上用户可用的身份验证器列表进行操作。此列表可以在会话生存期期间更新，例如后

注册或注销身份验证者。列表更新后，SDK会调用会话对象的-[TSXUIAuthenticationConfigurationSession set- AuthenticatorsList：]方法。这种情况至少发生在会话开始之前，并且可能在应用程序进行的批准服务调用之间发生。该应用程序还可以通过调用-[-TSXAuthenticationConfigurationSessionServices requestRefresh- AuthenticatorsAsynchronouslyWithHandler：]启动当前批准列表的刷新。

身份验证器列表由TSXConfigurableAuthenticator对象的集合组成。这些对象根据身份验证器的当前状态封装身份验证器上的信息以及可以在身份验证器上执行的操作的列表。通过这些对象的availableActions属性可以使用此功能。下表

列出了对此列表有效的值及其对应的TSXAuthentication- ConfigurationSessionServices方法和说明。

availableActions Valu eTSXAuthenticationConfiguration-

SessionServices服务

AuthenticatorConfigurationActionRegister - [TSXAuthentication-

ConfigurationSessionServices registerAuthenticatorWith- Authenticator：clientContext：ynchronousWithHandler：]

要求注册该认证者。该调用可以触发策略调用，在此期间可以调用其他UI处理程序回调（例如，执行身份验证操作）。此调用的客户端上下文参数允许UI处理程序为将在此策略调用的上下文中调用的UI处理程序调用提供自己的客户端上下文。如果此客户端上下文为null，则将使用用于配置会话的相同上下文。

AuthenticatorConfigurationActionUnregister-[TSXAuthentication-

ConfigurationSessionServices unregisterAuthenticatorWith- Authenticator：clientContext：asynclyWithHandler：]

要求注销此身份验证器。

AuthenticatorConfigurationActionReregister-[TSXAuthentication-

ConfigurationSessionServices reregisterAuthenticatorWith- Authenticator：clientContext：asynclyWithHandler：]

要求重新注册此身份验证器。该调用可以触发策略调用，在此期间可以调用其他UI处理程序回调（例如，执行身份验证操作）。此调用的客户端上下文参数允许UI处理程序为将在此策略调用的上下文中调用的UI处理程序调用提供自己的客户端上下文。如果这

如果client context为null，则将使用与配置会话相同的上下文。

**注意：只能** 通过-[TSXUIAuthenticationConfiguration- Session setAuthenticatorsList：]提供给此会话的身份验证器对象上调用操作。

应用程序可以通过调用方法为用户设置默认的身份验证者

-[TSXAuthenticationConfigurationSessionServices setDefault- AuthenticatorWithAuthenticator：ynchronousWithHandler：]。

当应用程序确定身份验证器配置会话已完成（例如，要求用户关闭UI）时，它会调用-[TSXAuthenticationConfiguration- SessionServices finishSession]服务。当SDK或应用程序决定结束会话时，SDK将调用-[TSXUIAuthentication- ConfigurationSession endSession]。

以下代码示例显示了自定义身份验证器配置UI处理程序的示例。我们首先实现-[TSXUIHandler create- AuthenticationConfigurationSessionWithUserId：]，如下所示：

@implementation MyCustomUIHandler

-（TSXUIAuthenticationConfigurationSession

\*）createAuthenticationConfigurationSessionWithUserId：（NSString \*）userId {return [[MyAuthenticationConfigurationSession alloc] initWithUserId：userId

HostingVc：self.hostingViewController];

此方法的实现只是为TSXUIAuthenticationConfigurationSession接口创建并返回我们自己的实现。初始化此实现的实例后，我们只需创建一个UI，该UI将显示身份验证器列表并提供控件以对其执行操作-如以下代码所示：

@interface MyAuthenticationConfigurationSession（）{TSXDefaultUIHandlerHostingContext \* \_hostingContext; MyAuthenticationConfigurationViewController \* \_vc;

}

@结束

@implementation MyAuthenticationConfigurationSession

-（instancetype）init {self = [super init];如果（自己）{

\_vc = [[MyAuthenticationConfigurationViewController alloc] init];

}

返回自我

}

当会话开始时，我们记录该会话接收到的客户端上下文和TSXAuthenticationConfigurationSessionServices对象（然后，我们将使用该对象对身份验证器执行操作）。然后，我们展示身份验证器配置UI。

-（void）startSessionWithAuthenticationConfigurationSessionServices：（TSXAuthenticationConfigurationSessionServices

\*）authenticationConfigurationSessionServices actionContext：（TSXPolicyAction

\*）actionContext clientContext：（NSDictionary \*）clientContext {

\_vc.authenticationConfigurationSessionServices = authenticationConfigurationSessionServices;

[\_hostingContext presentViewController：\_vc动画：是]；

}

当调用-[TSXUIAuthenticationConfigurationSession set- AuthenticatorsList：]调用时，我们更新UI显示的身份验证器列表（这将触发其刷新）：

-（void）setAuthenticatorsList：（NSArray <TSXConfigurableAuthenticator \*>

\*）身份验证者{

\_vc.authenticators =验证者；

}

最后，当调用-[TSXUIAuthenticationConfigurationSession endSession]时，我们将关闭UI：

-（void）endSession {

[\_hostingContext dismissViewController：\_vc动画：是]；

我们的视图控制器实现基本上包括一个常规的表视图，该表设置用于显示身份验证器列表：

@interface MyAuthenticationConfigurationViewController：UIViewController（）{NSArray <TSXConfigurableAuthenticator \*> \* \_authenticators;

}

@属性（弱，非原子）IBOutlet UITableView \* authenticatorsTable; @属性NSArray <TSXConfigurableAuthenticator \*> \* authenticators; @property TSXAuthenticationConfigurationSessionServices

\* authenticationConfigurationSessionServices;

@结束

@implementation MyAuthenticationConfigurationViewController

-（void）viewDidLoad {[super viewDidLoad];

[self.authenticatorsTable registerNib：[UINib nibWithNibName：@“ MyConfigurationMenuAuthenticatorTableViewCell”包：无] forCellReuseIdentifier：@“ ConfigurationMenuAuthenticatorTableViewCell”]；

self.navigationItem.title = @“身份验证配置”; self.navigationItem.leftBarButtonItem = [[UIBarButtonItem分配]

initWithTitle：@“后退”样式：UIBarButtonItemStyleDone目标：自我操作：@selector（backButtonPressed）];

self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc] initWithBarButtonSystemItem：UIBarButtonSystemItemRefresh target：self action：@selector（refreshButtonPressed :)];

[self.authenticatorsTable.topAnchor约束EqualToAnchor：self.topLayoutGuide.bottomAnchor] .active = true;

}

-（NSInteger）numberOfSectionsInTableView：（UITableView \*）tableView {返回1;

}

-（NSInteger）tableView：（UITableView \*）tableView numberOfRowsInSection：（NSInteger）节{

返回\_authenticators.count;

}

-（UITableViewCell \*）tableView：（UITableView \*）tableView cellForRowAtIndexPath：（NSIndexPath \*）indexPath {

ConfigurationMenuAuthenticatorTableViewCell \* cell = [self.authenticatorsTable dequeueReusableCellWithIdentifier：@“ ConfigurationMenuAuthenticatorTableViewCell” for IndexPath：indexPath];

cell.authenticationConfigurationSessionServices = self.authenticationConfigurationSessionServices;

cell.authenticator = [\_authenticators objectAtIndex：[indexPath row]];

返回单元

}

-（IBAction）refreshButtonPressed：（id）sender {[self.authenticationConfigurationSessionServices

requestRefreshAuthenticatorsAsynchronouslyWithHandler：^（BOOL r，TSXAuthenticationError \* e）{

NSLog（@“刷新批准完成。”）;

}];

}

@结束

单击列表中的验证者后，我们将其设置为默认验证者：

-（void）tableView：（UITableView \*）tableView didSelectRowAtIndexPath：（NSIndexPath

\*）indexPath {

TSXConfigurableAuthenticator \* authenticator = [\_authenticators objectAtIndex：[indexPath row]];

[self.authenticationConfigurationSessionServices setDefaultAuthenticatorWithAuthenticator：authenticator

异步lyWithHandler：^（BOOL r，TSXAuthenticationError \* \_Nullable err）{NSLog（@“ Default set”）;

}];

}

当验证者配置会话对象更新验证者集时，我们触发刷新显示的表：

-（void）setAuthenticators：（NSArray <TSXConfigurableAuthenticator \*> \*）authenticators {

\_authenticators =验证者； [self.authenticatorsTable reloadData];

}

-（NSArray <TSXConfigurableAuthenticator \*> \*）authenticators {返回\_authenticators;

}

当用户单击“后退”按钮以表明他已完成配置功能时，我**们要求SD**K完成会话：

-（void）backButtonPressed {[self.authenticationConfigurationSessionServices finishSession];

}

@结束

每个身份验证器表视图单元都提供有关其代表的身份验证器的信息：

@interface MyConfigurationMenuAuthenticatorTableViewCell：UITableViewCell（）{TSXConfigurableAuthenticator \* \_authenticator;

}

@属性（弱，非原子）IBOutlet UIButton \* unregisterButton; @属性（弱，非原子）IBOutlet UIButton \* reregisterButton; @属性（弱，非原子）IBOutlet UIButton \* registerButton; @属性（弱，非原子）IBOutlet UILabel \* authenticatorNameLabel;

@属性（弱，非原子）IBOutlet UILabel \* authenticatorIconLabel; @属性（弱，非原子）IBOutlet UILabel \*默认检查标记； @property TSXConfigurableAuthenticator \*身份验证器；

@property TSXAuthenticationConfigurationSessionServices

\* authenticationConfigurationSessionServices; @结束

@实现ConfigurationMenuAuthenticatorTableViewCell

-（void）awakeFromNib {[super awakeFromNib];

//初始化代码

}

-（void）setAuthenticator：（TSXConfigurableAuthenticator \*）authenticator {

\_authenticator =验证者； [self loadAuthenticatorData];

}

-（TSXConfigurableAuthenticator \*）authenticator {return \_authenticator;

}

-（void）loadAuthenticatorData {

self.authenticatorNameLabel.text = \_authenticator.description.name; self.defaultCheckmark.hidden =！\_authenticator.description.defaultAuthenticator;

这还包括基于可用操作启用或禁用UI元素以对身份验证器项目执行操作：

self.registerButton.enabled =（\_authenticator.availableActions＆AuthenticatorConfigurationActionRegister）;

self.reregisterButton.enabled =（\_authenticator.availableActions＆AuthenticatorConfigurationActionReregister）;

self.unregisterButton.enabled =（\_authenticator.availableActions＆AuthenticatorConfigurationActionUnregister）;

}

最后，响应于使用操作来执行配置操作，调用身份验证器配置菜单会话服务。

-（IBAction）unregisterAuthenticator：{id）sender {[self.authenticationConfigurationSessionServices

unregisterAuthenticatorWithAuthenticator：\_authenticator异步WithHandler：^（BOOL r，TSXAuthenticationError \* \_Nullable e）{

NSLog（@“已调用ReRegisterAuthenticator处理程序。”）;

}];

}

-（IBAction）reRegisterAuthenticator：（id）sender {[self.authenticationConfigurationSessionServices

reregisterAuthenticatorWithAuthenticator：\_authenticator clientContext：nilynchronousWithHandler：^（TSXAuthenticationResult \* \_Nonnull r，TSXAuthenticationError \* \_Nullable e）{

NSLog（@“已调用ReRegisterAuthenticator处理程序。”）;

}];

}

-（IBAction）registerAuthenticator：（id）sender {[self.authenticationConfigurationSessionServices

registerAuthenticatorWithAuthenticator：\_authenticator

clientContext：nil

异步lyWithHandler：^（TSXAuthenticationResult \* \_Nonnull r，TSXAuthenticationError \* \_Nullable e）{

NSLog（@“已调用RegisterAuthenticator处理程序。”）;

}];

}

@结束

* + 1. **实施促销会话界面**

现在，传输平台提供了Promotion方法，这是注册未注册支持的Authenticator的另一种方法。使用此方法还可以取消注册，而不会中止控制流程。在执行行程控制流程时，平台可以为用户提供通过促销注册一个或多个受支持的身份验证器的选项。

SDK执行升级时，它将调用+ [TSXPromotionInput create- ControlResponseWithControlRequest：]。调用，并期望返回TSXUIPromotionSession的实例。

TSXUIPromotionSession以标准的SDK调用-[TSXUIPromotion- Session startSessionWithClientContext：actionContext：]开始。然后，SDK发送-[TSXUIPromotionSession提示IntroductionWithTitle：文本：continueText：cancelText：asynchronousWithHandler：]调用。 SDK期望返回PromiseFuture结果，其中包含使用以下方式发送的升级请求类型：+ [TSXPromotionInput createControlResponseWithControl- Request：]。

TSXPromotionControlRequest Enu mDescription值

PromotionControlRequestSkip

这将取消升级，但不会终止其余的身份验证过程。

后跟-[TSXUIPromotion- Session endSession]。

PromotionControlRequestAbort

完全中止身份验证过程，从SDK调用中返回错误。后跟-[-TSXUIPromotionSession end- Session]。

PromotionControlRequestContinue 继续升级过程，以及

调用可用身份验证器列表。后跟-[TSXUIPromotion-会话setPromoted- AuthenticatorsWithOptions：asynchronousWithHandler：]。

这是一个介绍示例：

-（void）promptIntroductionWithTitle：（NSString \* \_Nonnull）标题文本：（NSString \*

\_Nonnull）文本continueText：（NSString \* \_Nonnull）continueText cancelText：（NSString \*

\_Nonnull）取消文本异步处理程序：（void（^ \_Nonnull）（TSXPromotionInput \*

\_Nonnull））处理程序{

UIAlertController \* alert = [UIAlertController alertControllerWithTitle：标题消息：消息preferredStyle：UIAlertControllerStyleAlert];

[alert addAction：[UIAlertAction actionWithTitle：continueText style：UIAlertActionStyleDefault handler：^（UIAlertAction \* \_Nonnull action）{

handler（[TSXPromotionInput createControlResponseWithControlRequest：PromotionControlRequestContinue]）;

}]];

[alert addAction：[UIAlertAction actionWithTitle：data.cancelText样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull操作）{

handler（[TSXPromotionInput createControlResponseWithControlRequest：PromotionControlRequestSkip]）;

}]];

[alert addAction：[UIAlertAction actionWithTitle：@“永不再显示”样式：UIAlertActionStyle破坏性处理程序：^（UIAlertAction \* \_Nonnull action）{

handler（[TSXPromotionInput createControlResponseWithControlRequest：PromotionControlRequestAbort]）;

}]];

[自己的presentViewController：警报动画：真实完成：无]；

}

如果-[-TSXUIPromotionSession提示IntroductionWithTitle：文本：continueText：cancelText：asynclyWithHandler：]提供了PromotionControlRequestContinue，则SDK将使用未注册的验证者列表调用-[TSXUIPromotionSession setPromotedAuthenticatorsWith-选项：asynchronousWithHandler：]调用。 SDK期望返回PromiseFuture结果，该结果包含与-[TSXUIPromotionSession提示Introduction- WithTitle：文本：continueText：cancelText：asynclyWith-Handler：]类似的controlResponse，或通过+ [TSXPromotionInput create- AuthenticatorDescriptionWithSelectedAuthenticator：]注册的身份验证器。

在身份验证会话结束时，如果仍有未注册的身份验证器可用，并且不会调用Promotion-ControlRequestSkip或PromotionControlRequestAbort，则将调用-[TSXUIPromotionSession set- PromotedAuthenticatorsWithOptions：ynchronousWithHandler：]。

这是显示可选身份验证器列表的示例：

@interface TSXUIDefaultPromotionViewController（）<UITableViewDataSource，UITableViewDelegate>

@属性（不安全的，未保留的，非原子的）IBOutlet UITableView \* tableView; @property（readwrite）NSArray <TSXAuthenticatorDescription \*> \* options; @结束

@implementation TSXUIDefaultPromotionViewController

-（void）showAuthenticatorsWithOptions：（NSArray <TSXAuthenticatorDescription \*>

\*）选项{

self.options =选项; self.tableView.dataSource =自我; self.tableView.delegate =自我; [self.tableView reloadData];

}

-（nonnull UITableViewCell \*）tableView：（nonnull UITableView \*）tableView cellForRowAtIndexPath：（nonnull NSIndexPath \*）indexPath {

TSXAuthenticatorDescription \*选项= self.options [indexPath.row]; UITableViewCell \* cell = [[UITableViewCell分配]

initWithStyle：UITableViewCellStyle默认重用标识符：无]； cell.textLabel.text = option.name;

返回单元

}

-（void）tableView：（UITableView \*）tableView didSelectRowAtIndexPath：（NSIndexPath

\*）indexPath {

TSXAuthenticatorDescription \*选项= self.options [indexPath.row]; [handler（[TSXPromotionInput

createAuthenticatorDescriptionWithSelectedAuthenticator：option]）；

}

-（NSInteger）tableView：（nonnull UITableView \*）tableView numberOfRowsInSection：（NSInteger）section {

返回self.options.count;

}

@结束

发送，发送PromotionControlRequestSkip或发送PromotionControlRequestAbort或没有可用的验证者时，会话结束。在这种情况下，SDK将调用-[TSXUIPromotionSession end-Session]方法。

* + 1. **策略跟踪回调**

TSXUIHandler接口包括一组回调函数，SDK在策略控制流执行期间调用这些回调函数，以允许应用程序跟踪并响应控制流的进度。例如，使用跟踪API，应用程序可以知道某个动作何时开始以及何时结束–从而可以将SDK调用的所有回调调用关联为同一动作的一部分（就像它可以使用actionContext参数一样）各种回调）。

SDK将调用-[TSXUIHandler controlFlowStartingWithClient- Context：]来通知应用程序控制流即将开始执行。

此调用接收与正在执行策略的范围内的SDK调用关联的clientContext参数。当控制流执行终止时，SDK将调用-[TSXUIHandler controlFlowEndedWith-

错误：clientContext：] –再次传递同一SDK调用的clientContext，并提供可选的错误参数，如果控制流因错误而终止，则该参数将包含TSXAuthenticationError类型的对象（如果控制流成功完成，则为null）。

在处理正在运行的控制流中的每个动作之前，SDK会调用-[-TSXUIHandler controlFlowActionStartingWithActionContext：clientContext：]。与-[TSXUIHandler controlFlowStartingWithClient- Context：]相似，此调用还接收与正在执行策略的范围内的SDK调用关联的clientContext参数。但是，此回调还接收与正在启动的动作有关的actionContext对象。

确保这两个参数与传递给在此操作的上下文中完成的所有TSXUIHandler回调调用的clientContext和actionContext参数相同。完成动作处理后，SDK会调用

-[TSXUIHandler controlFlowActionEndedWithError：actionContext：clientContext：]将相同的clientContext和actionContext参数传递给-[TSXUIHandler control- FlowActionStartingWithActionContext：clientContext：]的相应调用，以及一个可选的错误参数，该参数将包含类型为TSXAuthentication-的对象如果操作处理以错误结束，则为Error；如果控制操作成功完成，则为null。

* + 1. **本地身份验证器无效**

在某些情况下，本地身份验证器注册变得无效，对此的一个常见示例是设备中新指纹的注册，这会使现有的指纹验证器注册无效。当检测到此类无效时，SDK会通过调用UI处理程序的-[TSXUIHandler localAuthenticator- InvalidatedWithDescription：clientContext：异步-WithHandler：]方法进行响应，并将身份验证器描述符传递给该方法。仅在此方法异步返回后，SDK操作执行才会恢复。该方法可以向用户显示对话框，或者仅出于其目的记录无效事实。此调用的实现完全取决于特定的应用程序需求。

* + 1. **实施扫描QR码会话界面**

有时，旅途可能需要执行扫描QR码操作。在这种情况下，SDK将首先调用UI处理程序的-[TSXUIHandler createScanQrSession- WithActionContext：clientContext：]方法，并期望接收TSXScanQrSession的实例。当SDK调用-[TSXScanQrSession startSessionWithClientContext：action- Context：]方法时，将启动TSXScanQrSession。然后，SDK会调用该会话的-[TSXScanQrSession getScanQrResponseWithInstructions：qrCodeFormat：异步-

WithHandler：]方法。这将返回一个PromiseFuture <>结果，该结果包含包装在TSXInputOrControlResponse对象中的Scan QR Action结果。扫描QR码操作成功完成后，结果将包含

TSXScanQrCodeInput对象，其中包含扫描的QR码。如果扫描QR动作未成功完成，则响应将包含一个带有相关控制请求的TSXControlRequest对象（例如，当用户取消该动作时，ControlControlTypeCancel-Authenticator）。收到结果后，SDK将调用-[TSXScanQrSession endSession]方法。

这是处理“扫描QR码”操作的示例：

#import“ NSBundle + TransmitSecurity.h”

@interface TSXUIDefaultScanQRSession（）<TSXUIDefaultScanQRSessionVcDelegate> {TSXDefaultUIHandlerHostingContext \* \_hostingContext;

NSDictionary \* \_clientContext; TSXUIDefaultScanQRSessionVc \* \_viewController; TSXQrCodeFormat \_qrCodeFormat;

无效（^ \_currentInputHandler）（TSXInputOrControlResponse <TSXScanQrCodeInput \*> \*

\_Nonnull）;

}

@结束

@implementation TSXUIDefaultScanQRSession

-（void）startSessionWithClientContext：（NSDictionary \*）clientContext

actionContext：（TSXPolicyAction \*）actionContext {

\_clientContext = clientContext;

\_hostingContext = [TSXDefaultUIHandler HostingContextFromClientContext：clientContext];

\_viewController = [[TSXUIDefaultScanQRSessionVc alloc] initWithNibName：nil

捆绑包：[NSBundle

tsResourcesBundle]];

\_viewController.delegate =自我;

if（viewController）{

[\_hostingContext presentViewController：\_viewController动画：是]；

}

}

-（void）endSession {if（\_viewController）{

[\_hostingContext dismissViewController：\_viewController动画：是]；

}

}

-（void）getScanQrResponseWithInstructions：（NSString \*）指令

qrCodeFormat：（TSXQrCodeFormat）qrCodeFormat异步处理程序:(无效（^）

（TSXInputOrControlResponse <TSXScanQrCodeInput \*> \* \_Nonnull））处理程序{

\_currentInputHandler =处理程序;

\_ qrCode Format = qrCode Format;

[\_viewController startQrInputWithInstructions：instructions qrCodeFormat：qrCodeFormat];

}

-（void）scanQrSessionVc：（TSXUIDefaultScanQRSessionVc \*）vc完成QrScan：（TSXCameraInputResponse \*）响应{

NSString \* scannedCode = [[[response acquisitionResponse] objectForKey：@“ sample”];

TSXQrCodeResult \* codeResult = [TSXQrCodeResult createQrCodeResultWithQrCode：scannedCode qrCodeFormat：\_qrCodeFormat]；

TSXScanQrCodeInput \* input = [TSXScanQrCodeInput createScanQrCodeInputWithQrCodeResult：codeResult];

[自行解决CurrentPromiseWith：[TSXInputOrControlResponse createInputResponseWithInputResponse：输入]]；

}

-（void）scanQrSessionVcRequestedCancel：（TSXUIDefaultScanQRSessionVc \*）vc {[self resolveCurrentPromiseWith：[TSXInputOrControlResponse

createControlResponseWithControlRequest：[TSXControlRequest createWithRequestType：ControlRequestTypeCancelAuthenticator]]];

}

-（void）resolveCurrentPromiseWith：（TSXInputOrControlResponse <TSXScanQrCodeInput \*> \*

\_Nonnull）响应{

无效（^ handler）（TSXInputOrControlResponse <TSXScanQrCodeInput \*> \* \_Nonnull）=

\_currentInputHandler;

\_currentInputHandler = nil; if（handler）{

处理程序（响应）；

}

}

@结束

@interface TSXUIDefaultScanQRSessionVc（）<TSXCameraInputViewDelegate> {TSXUIDefaultScanQRSession \* \_session;

NSString \* \_instructions; NSArray <NSDictionary \*> \* \_acq挑战；

}

@属性（弱，非原子）IBOutlet UILabel \* instructionsLabel; @属性（弱，非原子）IBOutlet TSXCameraInputView \* cameraInputView; @属性（弱，非原子）IBOutlet UIView \* instructionsBackgroundView;

@结束

@实现TSXUIDefaultScanQRSessionVc

-（instancetype）initWithSession：（TSXUIDefaultScanQRSession \*）session {

自我= [super initWithNibName：nil bundle：[NSBundle tsResourcesBundle]];如果（自己）

{

\_session =会话；

}

返回自我

}

-（void）viewDidLoad {[super viewDidLoad];

self.navigationItem.title = @“扫描QR码”； self.navigationItem.leftBarButtonItems = @ [

[[UIBarButtonItem分配] initWithTitle：@“取消”样式：UIBarButtonItemStylePlain目标：自我操作：@selector（cancelClicked :)]

];

self.cameraInputView.useBackCamera = YES;

//如果有待处理的说明-在加载视图之前进行设置-

// kickstart输入。 if（\_instructions）{

[self startQrInput];

}

}

-（void）startQrInputWithInstructions：（NSString \*）instructionsText

qrCodeFormat：（TSXQrCodeFormat）qrCodeFormat {

\_instructions = InstructionsText;

\_acqChallenges = @ [@ {

@“ type”：@“ qrcode”，@“ acquisition\_constraints”：@ [

@{

}

} ];

if（self.cameraInputView）{[self startQrInput];

}

@“类型”：@“ qrcode.exists”

}

]

-（void）showInstructionsText：（NSString \*）text {self.instructionsLabel.text = text; [self.view layoutSubviews];

self.instructionsBackgroundView.layer.cornerRadius = self.instructionsBackgroundView.frame.size.height / 2;

self.instructionsBackgroundView.layer.masksToBounds = YES; self.instructionsBackgroundView.clipsToBounds = YES;

}

-（void）startQrInput {

[self showInstructionsText：\_instructions];

self.cameraInputView.acquisitionChallenges = \_acqChallenges; [self.cameraInputView startSampling]; self.cameraInputView.acquisitionEnabled = YES;

}

-（void）cameraInputView：（TSXCameraInputView \*）视图提示已更改：（NSArray <NSString \*>

\*）提示{

//此视图中未显示任何提示

}

-（void）cameraInputView：（TSXCameraInputView \*）查看响应已收集：（TSXCameraInputResponse \*）response {

[self.delegate scanQrSessionVc：selfcompletedQrScan：response];

}

-（void）cameraInputView：（TSXCameraInputView \*）查看responseCollectionError：（NSError

\*）错误{

NSString \* errorMessage = [NSString stringWithFormat：@“读取QR代码时出错：％@”，错误]；

UIAlertController \* alert = [UIAlertController alertControllerWithTitle：@“错误”消息：errorMessage

preferredStyle：UIAlertControllerStyleAlert];

[alert addAction：[UIAlertAction actionWithTitle：@“重试”

样式：UIAlertActionStyle默认处理程序：^（UIAlertAction \* \_Nonnull

动作）{

[self startQrInput];

}]];

[alert addAction：[UIAlertAction actionWithTitle：@“ Abort”

样式：UIAlertActionStyle破坏性处理程序：^（UIAlertAction \* \_Nonnull

动作）{

[self cancelClicked：nil];

}]];

[自己的presentViewController：警报动画：是完成：无]；

}

-（void）cancelClicked：（id）sender {

//不再有有效的指令

\_instructions = nil; [self.cameraInputView cancelSampling];

[self.delegate scanQrSessionVcRequestedCancel：self];

}

@结束

* + 1. **实施UI处理程序的其他注意事项**

本节列出了实现自定义UI处理程序的其他注意事项和最佳实践。

**6.4.15。维护身份验证UI机密性**

当应用程序移至后台时，平台会捕获其屏幕截图以供例如在任务切换器中使用。公认的最佳做法是确保没有

截屏时，UI上会显示敏感信息。 UI处理程序的实现必须确保相同。身份验证UI可能包括部分（或全部）身份验证输入和其他敏感信息，因此当应用程序移至后台时必须清除。

# 第7章创建运输提供商

## 总览

在正常运行期间，Transmit SDK与Transmit Server通信以进行各种操作，例如获取身份验证策略和执行

实际的身份验证。通常，SDK通过HTTPS（或仅建议用于开发环境的HTTP）与服务器通信。应用程序作者可以自定义HTTPS协议的某些方面，例如，如第3.6.4节所述，实现SSL证书固定和提供相互SSL身份验证。但是，在某些情况下，需要对与服务器的通信进行更深入的自定义-甚至可能不使用HTTPS

完全是一种特定于应用程序的协议，通过该协议可以传输传输协议。当需要这种定制级别时，SDK将为应用程序提供选项以实现定制传输提供程序。 SDK将使用客户提供者与发送服务器执行消息交换。通常，自定义传输提供程序将与后端服务通信，该后端服务将采取行动

作为传输服务器的网关或代理。该后端服务将接收来自SDK的自定义传输通信，并基于该通信上的信息构造一个HTTP请求，并将其发送到传输服务器。在第114页的图2：传输提供程序体系结构中概述了此体系结构。

移动应用

后端

请求对象

传送SDK

自订

运输

自订通讯协定

HTTP

请求

网关

Transmi服务器

响应对象

自订通讯协定

HTTP

响应

**图2：运输提供商架构**

**注意：实施** 自定义提供程序允许应用程序拦截在传输服务器和SDK之间交换的每条消息。但是，SDK和传输服务器之间的协议可能会发生更改，因此应用程序设计必须不能依赖协议结构来实现功能。始终使用记录在案的API通过SDK访问传输服务器提供的数据。

## 实施自定义运输提供商

要实现自定义传输提供程序，应用程序应创建一个从TSXTransportProvider接口派生的对象。此类需要实现一种方法：-[TSXTransportProvider sendRequestWithRequest：ynchronallyWithHandler：]。当需要将请求发送到服务器时，SDK将调用此调用，并期望异步接收对该消息的响应。

作为TSXTransport-Request类型的对象提供了发送到服务器的请求。这包括字段url，方法，标头和bodyJson。这些字段一起描述了一个HTTP请求，该请求应该被构造并发送到后端的传输服务器。然后，自定义传输提供商有责任以其选择的任何协议和通信方式将此请求的详细信息传达给应用程序的后端。在后端，应根据请求详细信息构造HTTP请求，并将其发送到传输服务器。该请求应构造如下：

* + - 请求方法应设置为TSXTransport-Request对象的method属性。
    - 请求资源路径应设置为由TSXTransportRequest对象的url属性表示的URL的路径部分，包括片段和查询参数（如果存在）。这应从url属性原样获取，不要更改编码或以其他方式修改内容。
    - 该请求必须包含TSXTransportRequest对象的headers集合中指定的headers。请求标头应遵循它们在集合中出现的顺序。不得修改这些标头。
    - 请求正文应设置为TSXTransportRequest对象的bodyJson属性的值。尽管可以保证这是一个有效的JSON对象，但是即使请求保留了JSON语义（例如，空格剥离），请求也不得以任何方式修改此值。

一旦发送服务器接收到HTTP请求，它将发送回响应。此响应应传达回TSXTransportProvider对象，在该对象中将构造一个TSXTransportResponse对象来表示它，并从-[TSXTransportProvider send-

RequestWithRequest：asynchronousWithHandler：]。如果HTTP交换在协议层成功完成，这意味着已收到有效的HTTP响应，则TSXTransportResponse的构造应如下所示：

* + - TSXTransportResponse对象的status属性必须设置为从服务器接收到的HTTP响应的HTTP状态代码。
    - 对于此响应，必须将TSXTransportResponse对象的method属性设置为原始TSXTransportRequest的method属性。
    - TSXTransportResponse对象的headers属性必须包括从Transmit服务器接收的所有HTTP标头（按接收顺序）。
    - TSXTransportResponse对象的bodyJson属性必须包括从服务器接收到的响应正文，而没有对其进行任何转换。

如果发生通信错误，或者从传输服务器接收到无效的HTTP响应，则-[TSXTransportProvider sendRequestWithRequest：异步lyHandler：]方法应异步失败，并返回错误代码为AuthenticationErrorCodeCommunication的错误。

## 安装自定义传输提供程序

要安装自定义传输提供程序，请在SDK初始化之前调用-[-TSXTransmitSDKXm setTransportProvider：]。以下代码显示了此类调用的示例，该调用将实际安装“常规” HTTP传输提供程序。

…

TSXHTTPTransportProvider \* provider = [[TSXHTTPTransportProvider alloc] init]; [[TSXTransmitSDKXm sharedInstance] setTransportProvider：provider];

…