58 同城移动端 Passport SDK 的设计与技术细节

mobilehub

**导读】**58 赶集集团旗下拥有多个 App，且全部使用同一套账号体系，通过 Passport 部门提供的接口进行通信。经过多年迭代，各个 App 中关于 Passport 的功能均出现了一些流程和接口上的差异。为了提高账号安全，统一服务接口和流程，提高用户体验，由此决定开发了一个 Passport SDK，以集成 Passport 的相关功能，并提供给集团内各业务 App 使用。

在项目开始之初，我们在公司内经过调研发现在使用 SDK 时，大家最关心的问题就是 SDK 使用起来是否简单，即接口是否简单、调用流程是否简单、迭代升级是否简单。基于这几个关键问题，我们把设计目标定为：将原本 Passport 功能中繁琐的流程变成 Passport SDK 中简单的功能调用和结果处理，让使用 Passport 功能的开发者不再需要关心那些数量庞大而又无关紧要的部分，取而代之的是享有一个非常良好的开发体验。由此，我们将设计原则定为：

* 接口要精简；
* 服务的流程要黑盒；
* 无感知的迭代升级。

确定了设计原则后，下一步就是明确核心需求。Passport SDK 旨在为 58 同城账号体系下的用户提供通用的登录相关服务页面和接口。所以我们的 SDK 核心需求是提供服务，即通用服务页面和通用服务接口，并在用户调用服务后返回其结果。

**设计简单且有效的接口**

首先我们从需求上明确接口有哪些？答案是数据接口与服务接口，具体如下：

* 数据接口是一些零散的数据存取操作，实际上无法做出太多的精简。
* 服务接口包括各种服务页面的调起和服务接口的调用：在服务页面中，App 用户与服务页面的交互会触发对应的业务事件；在服务接口中，会直接触发对应的业务事件。

它们有一些共同点，比如都是主动发起的服务，都有各自的回调方法，大部分都需要可选或必选参数。

按照正常的设计模式，每个服务页面和服务接口都可以设计为单独的一个接口。但是因为 Passport 提供了数量众多的服务，这种设计会造成大量接口的出现，从而增加 SDK 的接入与维护成本。因此在接口的设计上，必须做减法。

Passport SDK 的服务接口采用了集中式接口，我们把所有的服务页面和服务接口抽象成服务类型。其中，每个服务类型代表一种服务，有自己的参数传递规则，有对应的回调方法。

如图 1 所示，我们使用了接口路由的方法，在接口模块内置了一个路由表，决定服务类型和对应服务（通用服务页面和通用服务接口）的映射。

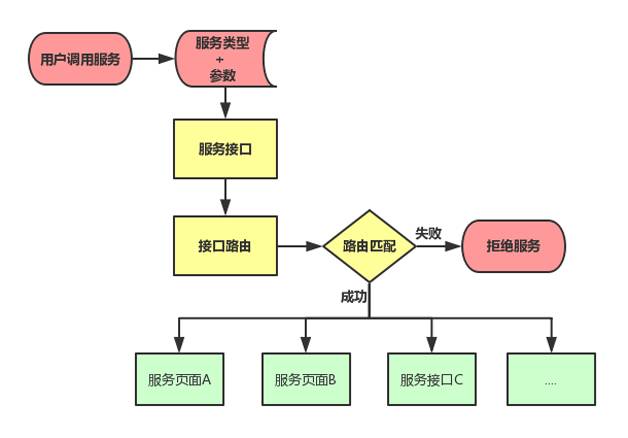


图 1 Passport SDK 服务接口设置

用户只需在这个服务接口里传入服务类型和符合规则的参数即可调用对应服务。服务完成后，会通过服务类型对应的回调方法传递结果：

http://mmbiz.qpic.cn/mmbiz_png/eL7YiacLdzwokEhadMCIRC2zR0jbYX15swvjUMynDk7tluuBOcmCP8fgVTmHPJHlUCicUJKQa3d2uXzDzo87W0ibg/640?wx_fmt=png&wxfrom=5&wx_lazy=1

简单的接口设计会降低接入工作的成本，并使用户获得极好的接入体验。

**服务的实现**

我们采用了分层的框架设计来支撑整个服务的实现，如图 2 所示。

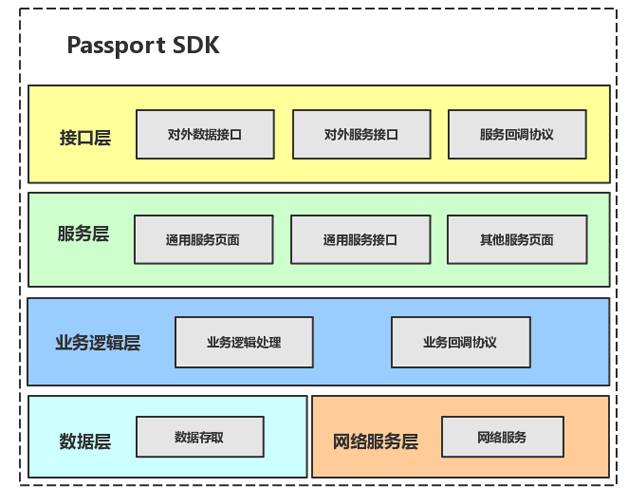


图 2 Passport SDK 架构设计

其中，接口层对外提供接口，对内调度服务：SDK 的用户调用账户登录服务页面的接口，调起账户登录服务页面。

服务层响应接口层的服务调用，发起具体业务事件，处理业务事件回调：账户登录服务页面调起后，App 用户输入账号密码，点击“登录”按钮，触发“账户登录”的业务事件，并在得到账户登录结果后处理页面跳转。

业务逻辑层处理服务层发起的业务事件，通过网络服务层发起对应的网络请求，数据层进行数据存取：“账户登录”的业务事件会首先跟 Passport 服务端通信，得到通信结果后处理数据，返回结果给账户登录服务页面。

整体框架层级划分清晰，可以做到垂直划分，减少了层次之间的依赖。层与层之间通过接口联系，互相影响而又互相独立，接口不变，就可以替换接口所实现的层次：后期服务页面的整体更换、网络服务的重新设计等都需要提前考虑。

层与层之间高内聚、低耦合，各司其职，有利于标准化以及各层逻辑的复用，结构更加明确——通用服务页面和通用服务接口会触发一些相同的业务事件。这种情况下，两者在业务逻辑层使用相同的代码去处理业务逻辑，在不同的服务层处理结果。

由此，既提高了开发编码速度，减小了并发开发难度，同时也降低了后期维护成本和维护时间。各层要实现的功能分配给不同的开发组，通过接口来互相通信，最后完成时组合起来就变成一个完整的功能。

**服务流程的黑盒化**

Passport 的服务流程并不是简单的一步走，以账号密码登录页面服务为例：

图 3 账号密码登录页面服务处理逻辑

如图 3 所示，用户首先需要输入账号和密码，点击登录按钮，这时会触发账号密码登录的业务事件，向 Passport 服务端发起请求，服务端会根据请求的校验结果和账号状态返回对应的结果：正常情形下为成功；异常情形下会根据状态调起对应的服务页面，如图片验证码校验页面—图片验证码页面会触发图片验证码下载的业务事件，得到展示的图片，用户输入正确的图片验证码后，点击确认，会再次触发账号密码登录的业务事件，与 Passport 服务端交互，最终得到成功登录的结果。

由此可见，Passport 的服务流程是由多个业务事件组合起来的，但是这些流程除了必要的交互外，跟用户并没有什么关系，而用户关心的只是最后的结果。所以，我们的设计目标是让用户对整个服务流程实现“无感知”。

图 4 SDK 用户调起服务实现

如图 4 所示，用户通过 Passport SDK 调起通用服务页面或通用服务接口。前者通过与 App 的用户交互触发业务事件，后者则直接触发业务事件。业务事件经过与服务端通信后得到事件的处理结果，标示服务结束或者跳转到其他服务页面，通过页面交互触发下一个业务事件直到服务结束。服务结束后，会通知用户服务的结果。

Passport 的服务可以看作是一个环形，由用户发起，在用户结束，中间有一次或多次的业务事件处理，且无需对用户暴露，所以 Passport 服务的处理过程实际上是一个黑盒——用户调起服务后，只有在出现成功或失败的结果后，他们才会收到服务完成的回调，其他的事件都在 Passport SDK 内部处理，用户只需要调起服务，等待响应服务结果即可。

考虑到回调的接收对象只有一个，为了方便用户，我们会对关键服务（例如登录和注销）的结果进行全局广播，只需要监听对应的频段即可响应。

**服务流程的节点**

图 5 服务流程

如图 5 所示，在服务的流程中，每个业务事件都是一个节点，而这些节点串联起来就形成了整个服务。业务事件的处理结果决定有没有下一个服务页面以及下一个服务页面是什么。所以根据服务端的灵活配置，可以对某个服务的流程进行实时变更，使服务更加贴合实际的需求。

我们还加入了万能节点，用于超出内置服务页面的特殊情况。它是一个简单的 Hybrid 框架，可以承载 Passport 的 Web 服务用于处理紧急事件，并同步信息至 SDK。

**迭代升级**

框架的分层设计使得业务的开发非常便捷——在各个层次添加要实现的功能，并使其通过接口或协议通信即可；对原有功能的修改也只影响内部对应的部分。

但是作为 SDK，不仅要考虑其自身的开发成本，还需要考虑用户的接入成本。

在接入方面，SDK 每次迭代新增的服务具体体现为新增的服务类型和对应的回调方法。接口调用方法不变，也没有增加太多的学习成本，且对原有功能的修改于用户而言也是无感知的。

**技术细节**

**自说明的接口**

下方代码是头文件中手机动态码登录这个服务类型的注释，从中可以了解到该服务类型的功能、是页面服务还是接口服务、参数的种类与功能、参数的示例。

通过接口名称和注释达到功能的自说明，不看示例代码就可以开始写，不亦乐乎？

**集成功能的可选**

有一些服务页面上集成了多个功能，比如多种登录方式、去往其他服务页面的入口等（如图 6 所示）。用户在使用通用服务页面时，可能只是需要其中的某部分功能而已。

图 6 登录页面

所以我们在服务类型的可选参数中给了用户选择的途径——通过参数可配置对应功能，比如入口的开启和关闭、展示及隐藏。

**接口的健壮性**

接口的参数一定要在第一时间做格式校验：接口作为统一的入口，会把参数通过路由传递给服务层，在入口保证参数的合法性，杜绝后面流程中参数可能出现问题的隐患。

其中，对外暴露的全局变量使用 const 修饰来保证这些变量的安全性。

**日志**

Passport SDK 内部内置了一套日志系统，但是不一定满足使用者的需求。所以我们在每一次写日志的时候都会有一次回调，将当前的日志参数传出，用户可以自行处理（如图 7 所示）。

图 7 日志处理流程

**开发调试与实际运行差异**

SDK 是以静态库的方式产出，在开发和调试时则是以源码的方式。所以 SDK 在调试跟实际运行时是两个环境，资源文件的路径会有所不同。

我们通过宏编译来达到在两个环境下使用正确的路径，并保持代码中的宏条件永远处于调试条件。在编译时使用聚合编译，通过 Shell 命令修改宏条件，来进行静态库条件的编译，在编译完成后将此宏条件改回调试条件，即如图 8 所示。

图 8 通过聚合编译的 Shell 脚本修改宏条件

**公用库的使用**

SDK 须尽量避免使用公共库，如果使用，公用库也应该作为外链文件存在，不要编译进静态库中，以避免与使用者的工程冲突（见图 9）。

**总结**

SDK 的设计与开发是一个长期且繁琐的过程。为了保证目标顺利实现，需要针对其做好设计方案，这样可以有效提升开发效率，减少隐藏的技术风险，并加强代码质量，从而使得整体的开发工作不断迭代，日趋完美。

**作者：**张达理，目前就职于 58 同城，任 iOS 客户端架构师。专注于跨端 SDK 研发以及性能优化，主导了 58 Passport SDK 的架构设计及研发。

**责编：**唐小引，技术之路，共同进步。欢迎技术投稿、给文章纠错，请发送邮件至tangxy@csdn.net。

**声明：** 本文为《程序员》03 期原创文章，未经允许请勿转载，更多精彩文章请订阅 2017 年《程序员》。