[SCA] APP 基础服务层统一技术方案

文档历史

版本	修订日期	修订者	修订内容
1.0.0	2021.10.27	Honggang Xiong	初版

1、需求概况

1.1 需求背景

目前 Shopee SupplyChain 下共有 5 款 App（SPX Driver、SPX Ops、Service Point、WMS、TWS），每个 App 的代码工程中都有各自的 Base 模块，在项目初期这种模式可以支持需求的快速迭代，但随着项目的发展、团队规模的扩张，这种模式也会带来以下问题：

- 重复造轮子。在开发人员不熟悉其他 App 的情况下，可能会重复开发一些通用功能。
- 同步成本高。某个 App 新增的功能如果其他 App 也适用，需要手动拷贝代码进行同步。
- 模块边界变模糊。某些 App 虽然拆分了通用 Base 和业务 Base 模块，但由于没有限制，在迭代过程中也会有一部分业务代码沉淀在通用 Base 中。

除 Base 模块外，支撑 App 的大部分基础组件（日志、存储等）都已拆分为独立 maven 库。WMS App 已经开始使用这些基础组件库，其他 App 也要尽早完成切换，否则也会出现上面的问题。

1.2 需求目的

- 优化各 App 的 Base 层，将各 App 的 Base 层统一，Base 层剔除业务相关能力，通过 maven-lib 形式管理确保通用性。
- 基础组件都以 maven-lib 的形式引入，利用 pods 强制规范版本、依赖问题。
- 各 App 按需优化 module 边界、启动管理以及应用生命周期的分发。

1.3 需求功能

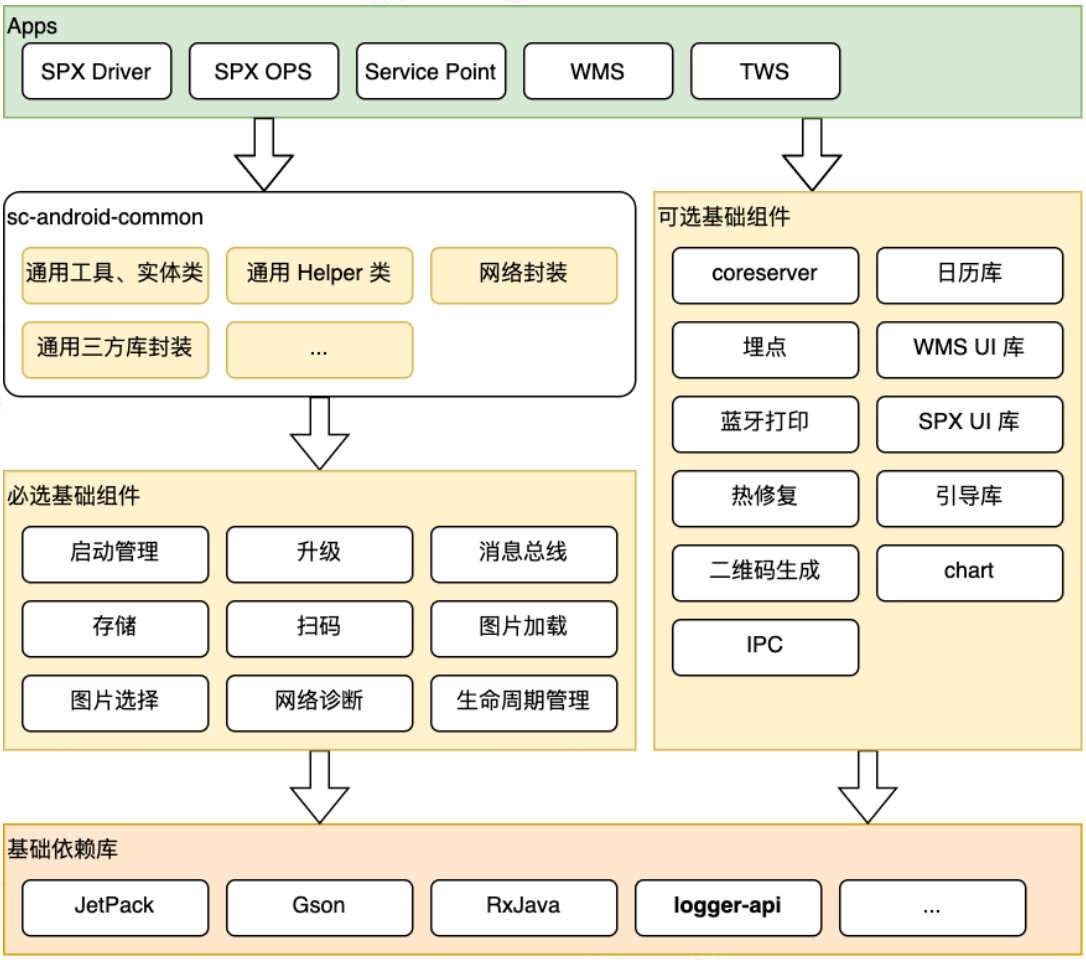
- 提取统一的 App Base 库，封装通用能力。
- 完善图片选择库、SPX UI 库，检查优化其他基础库的依赖关系、能力边界，基础库版本持续迭代。
- 各 App 中的基础组件都以 maven-lib 的形式引入，完成基础组件的的切换适配，基础库版本持续迭代。
- 利用 android pods 强制规范基础库的版本、依赖。
- 完成各 App 中的存储切换及数据迁移，存储库版本持续迭代。
- 完成各 App 中的 module 边界处理，启动管理 & 应用生命周期的分发。

2、技术方案设计

2.1 整体方案设计

可包含方案选型、整体架构、数据流、模块间关系等，如果是部分模块变动，需突出变动及影响部分，需对核心模块及关系进行文字描述，需具备良好的架构特性如通用性、扩展性等

提取统一的 App Base 库（暂取名 sc-android-common），sc-android-common 与其他基础库之间的依赖关系如下：



各层次基础库的介绍如下：

- 基础依赖库：各项目及各基础组件都需要用到的库，大部分为三方库，其中 logger-api 为依赖隔离库，可以统一所有内部代码库的日志并由项目进行个性化管理。
- 必选基础组件：各项目都需要用到的基础组件，主要为内部 maven 库。
- 可选基础组件：各项目按需使用的基础组件，主要为内部 maven 库，其中的部分组件也可以划入必选的范围。
- sc-android-common：封装各项目都适用但又达不到抽离独立库标准的通用能力，包括通用工具类、实体类、Helper 类、网络封装，三方库封装等。

2.2 详细方案设计

可包含核心模块功能和交互实现方案详细设计，如：路由、布局、组件、交互、数据流、状态管理、兼容性、性能优化、异常处理、安全防护、扩展性及注意事项等等

2.2.1、提取统一的 App Base 库，完善图片选择库、SPX UI 库

除 SPX Driver App 外，其他 App 都使用了源自 SPX Ops App 的 common:c-base 模块，那么提取统一的 App Base 库可以分以下几个步骤：

1. 提取各 App 的 common:c-base 模块作为 sc-android-common 的首版，剔除其中的业务代码。
2. 提取 SPX Driver App 的 app 模块（实际为 b-spx-base 模块）中的通用能力至 sc-android-common。
3. 提取 SPX Driver App 的网络封装至 sc-android-common。

同理，SPX Driver App 的图片选择逻辑也可以提取至 WMS 目前在用的 picture_picker 库。

由于历史技术选型或产品差异等原因，相同功能在不同 App 中也会存在或大或小的差异，比如上述的网络封装以及图片选择。针对这些差异代码的提取，可以有以下几种方式：

- a、若代码整体逻辑相同，只存在细小差异，可沉淀一套接口 + 一套实现，通过传入不同的 style 来实现差异化；
- b、若代码整体逻辑大致相同，还能抽象出公共接口，可沉淀一套接口 + 多套实现；
- c、若代码整体逻辑差异较大，可同时沉淀多套代码。

其中方式 c 所需的项目切换成本最小，但同时也可能造成相同功能在实现上差异越来越大的问题。在功能代码提取及项目切换适配的过程中，还是尽量向方式 a 和 b 靠齐。

2.2.2、各 App 中的基础组件都以 maven-lib 的形式引入，完成基础组件的的切换适配

基础组件的的切换适配通常有以下两种方式：

1. 切换至库实现，同时移除旧实现代码。该方式切换最彻底，但会有一段阵痛期，比如迭代分支还在使用旧实现，分支合并后无法编译需要再次适配，直到切换分支上线。
2. 切换至库实现，但保留旧实现代码；该方式虽然减少了分支合并的问题以及多次适配的工作量，但也会带来更多问题，比如切换不彻底，以及新旧实现不一致引发预期之外的问题。

各 App 的切换尽量以方式 1 为主，如有特殊原因需要保留旧实现代码的，也尽量在旧实现中转调用库实现，这样如果其他分支有修改同一处代码，分支合并时会提示冲突，此时可根据具体情况判断差异代码是保留在项目中还是沉淀到基础库中。

2.2.3、利用 android pods 强制规范基础库的版本、依赖

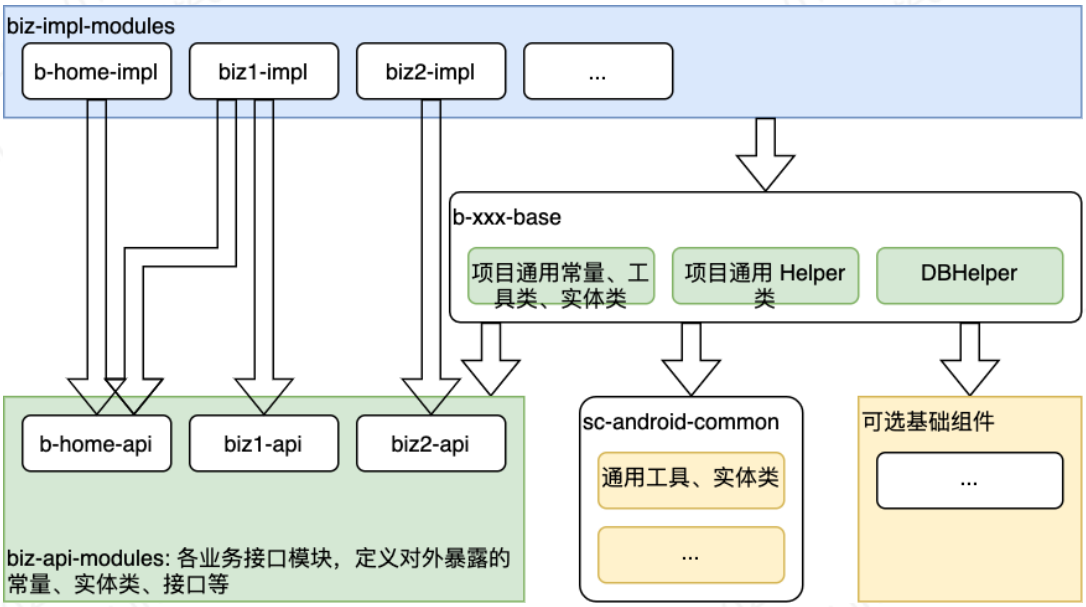
目前各内部 maven 库之间大多没有依赖，版本相互独立，在提取 sc-android-common 库后，sc-android-common 会对必选基础组件形成依赖，可用 android pods 进行库版本管理。（PS：如果各内部库严格遵循开发迭代规范，不用工具约束也是可行的，比如 Jetpack 中的不同库在不同版本下大多都能兼容，这块任重道远）

2.2.4、完成各 App 中的存储切换及数据迁移

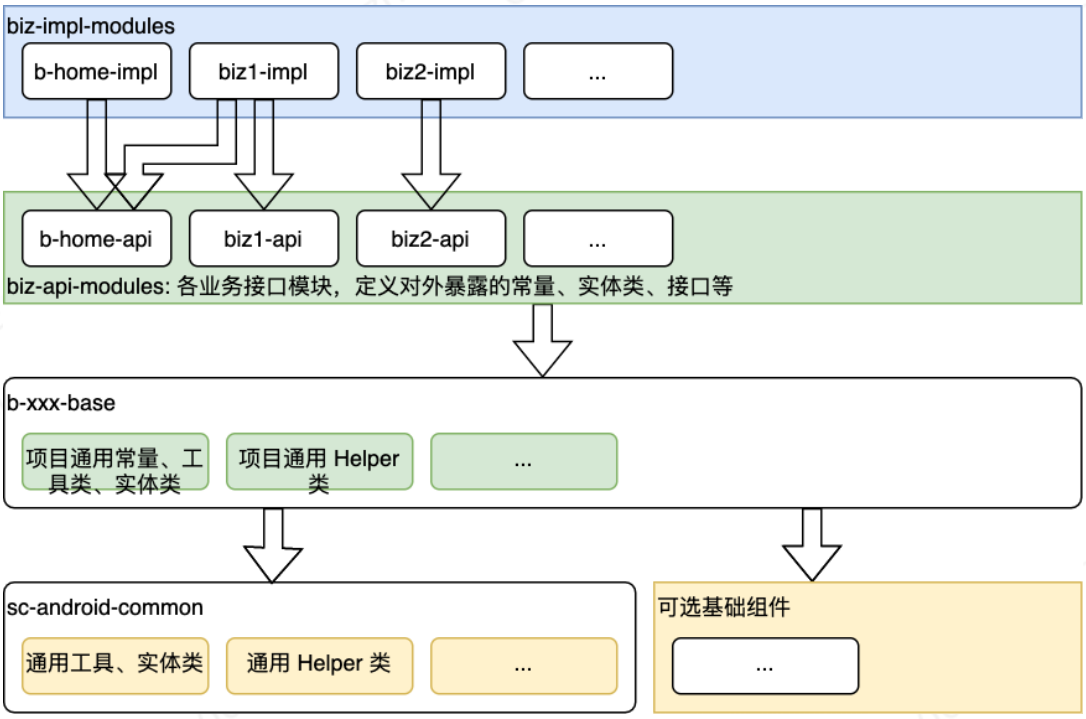
目前各 App 对于 KV、文件、DB 的存储实现各异，WMS 已经切换至存储库实现，其他 App 也需要跟进。在数据迁移方面，KV、文件较为简单，DB 较为复杂，这里重点讨论下 DB 数据迁移可能遇到的问题。

基于目前的存储库实现，DbHelper 管理类需要引用所有的 DatabaseClass，DatabaseClass 需要引用其中存储的所有 EntityClass，如果项目只封装一个 DbHelper 且置于 b-xxx-base 模块中，那项目所有的 EntityClass 都需要置于 b-xxx-base 模块中，将导致 b-xxx-base 模块过于臃肿。

如果采取折中方式，将各业务模块的接口、实体类提取至一个单独 api 模块，实体类集中的问题将得到解决，整体依赖如下：



是不是看着有点奇怪？b-xxx-base 模块居然需要依赖各业务 api 模块，这是因为项目只封装了一个 DbHelper，如果在 b-xxx-base 的通用 DBHelper 之外，每个有存储需求的业务 impl 模块都定义自己的 DbHelper，那整体依赖可以优化成：



当然这样可能造成一个项目包含多个 db，数据迁移更为复杂，但对于 SPX Driver App 来说，大部分单 function driver 用户只有一个 db 会被初始化，该方式带来的收益可覆盖切换成本。其他项目可根据实际情况，选择适合的 DbHelper 封装以及实体类存放方式。

2.2.5、完成各 App 中的 module 边界处理，启动管理 & 应用生命周期的分发

参考 [SCA] Application生命周期分发方案

2.3 接口设计

对外提供的接口相关信息，如接口名称、协议、参数，及注意事项等（如未涉及则不需填写）

2.4 数据存储设计

数据库、表、文件等存储方案设计（如未涉及则不需填写）

2.5 监控方案设计

可包含业务监控如PV、点击、曝光、用户行为等，及质量监控如页面性能、错误、API质量等部分

3、风险评估

列出可能存在的风险及应对策略，如外部依赖、技术成熟度、关联影响、不可控因素等

1、基础库切换适配可能需要多次进行，可适当预留时间进行处理及自测。
2、数据迁移自测难以覆盖所有情况，可能出现预期之外的问题。可采取逐步迁移的方式，首期只迁移不重要的缓存数据，收集迁移异常日志，后续再逐步建议重要数据。

其他

以上未包含的其他补充项说明

参考文档

SC Android 内部 maven 库总览

[SCA] Application生命周期分发方案