[Common] Driver App 组件化架构演进与业务解耦拆分

文档历史

版本	修订日期	修订者	修订内容
1.0.0	2021.04.19	Zhenwei.Dong	初版
1.0.1	2021.04.28	honggang.xiong	增加依赖隔离示意图

1、需求概况

1.1 需求背景

存在的痛点：

- 各业务模块之间代码耦合，随着版本不断迭代，新功能不断增加，业务上的处理逻辑越变越复杂，这就引发一个问题，所维护的代码成本越来越高，有时改个小的功能点就需要回归整个APP测试；
- 目前Driver App 架构方式是单一工程模式，业务规模扩大，随之带来的是团队规模扩大，那就涉及到多人协作问题。无论分包做的再好，随着项目的增大，项目会逐渐失去层次感，别人来接手的时候会很吃力。我们在Debug一个小功能的时候每次修改代码都需要重新build整个项目。多人联合开发在版本管理中很容易出现冲突和代码覆盖的问题；
- 目前比较难去做定制项目，假如现在业务要给Line Haul Driver 或 P2P Driver提供一个独立的轻量级App，成本会非常高。

1.2 需求目的

对于Dev：

- 业务模块解耦，更有利于多人团队协作开发；
- 组件化是功能重用的基石，提升开发效率；
- 提高工程编译速度。

对于业务：

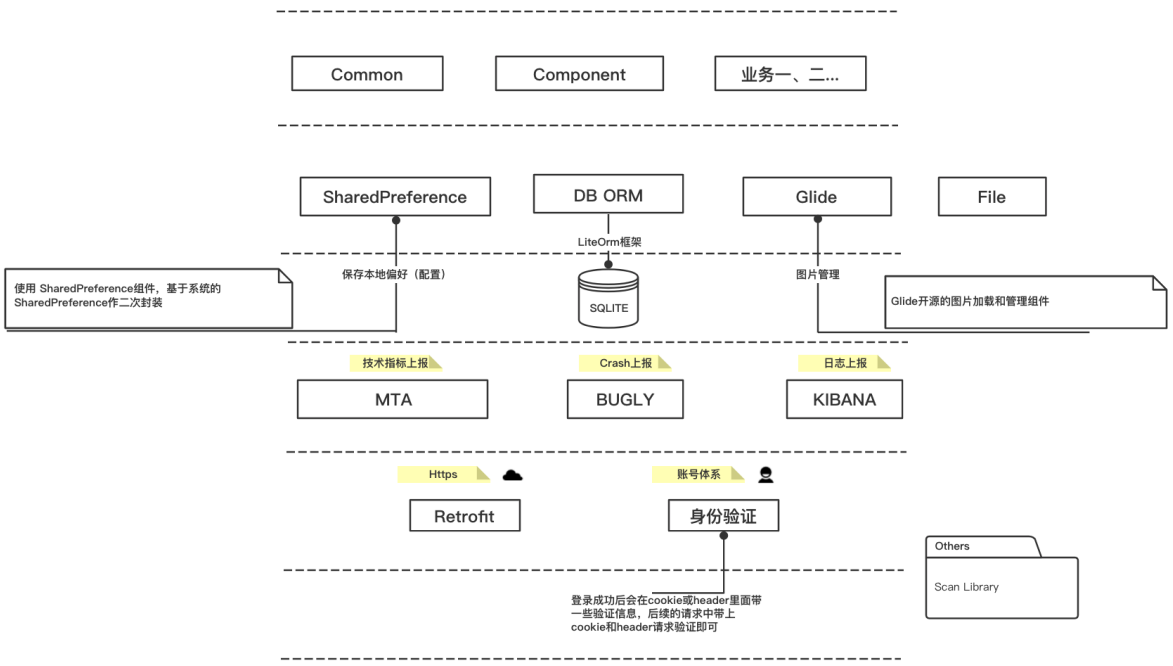
- 定制项目可按需加载，业务模块之间可以灵活组建，快速生成不同类型的定制产品。比如Q3规划中轻量级的LH Driver App，我们可以大大缩短开发周期。

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

目前如下**单一工程架构**模型中，整个项目没有业务模块的概念，只有以业务逻辑划分的文件夹，业务之间是直接相互调用和高度耦合在一起。优点是适合人数较少（1~2人）的项目，写起代码来比较行云流水，反正是基本哪里都能直接访问到。但随着Spx Driver业务的增长和项目开发人员的增加，很明显这样大大降低团队多人协作开发的效率，无法持续健康的做业务承载和扩展。要如何能够快速的承接业务和高效的协作开发，项目架构演进变得至关重要。



按规划架构由单一工程架构演进至**组件化架构**，这种架构的优势在于它很好地解决了业务之间的耦合问题，而业务内部又足够内聚，能很好地适应项目迭代和团队的协作开发。

组件化的好处

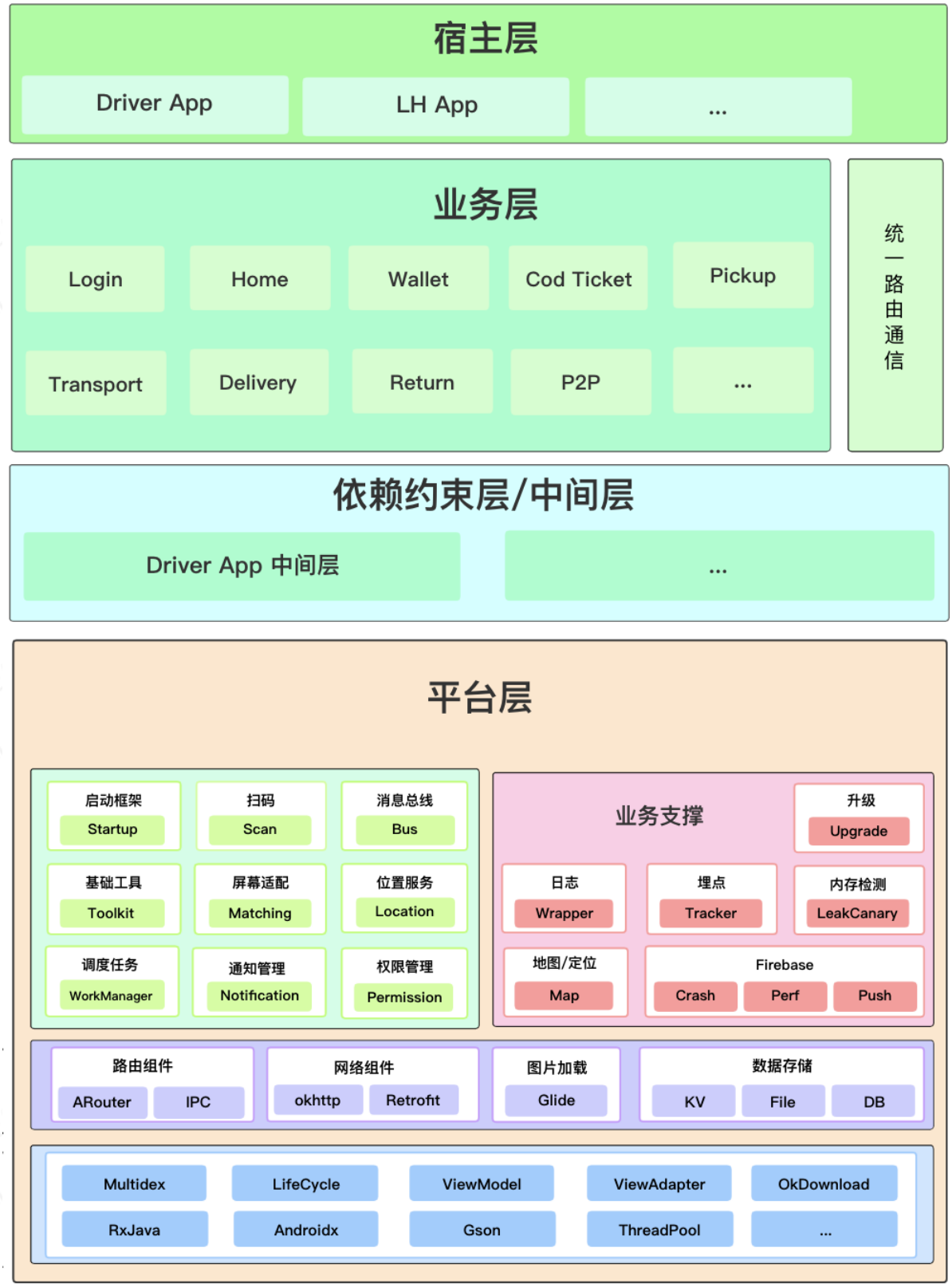
解耦：组件之前通过路由协议访问，没有直接关联，业务清晰独立；

复用：相同能力的基础组件和服务性的业务组件可以跨项目复用；

单一职责：每个组件提供单一功能的业务能力；

编译：可以单业务组件或者组合业务组件进行编译，无需编译所有代码；

扩展：无论是业务能力的扩展还是基础能力的扩展，都是新增一个独立组件，业务基础能力的卸载也是只要移除一个独立组件，这样方便灵活快速扩展业务能力；



从图中可以看到，业务层各组件之间是独立的，互相没有关联，这些业务组件在集成模式下是一个个 Library，被 App 壳工程所依赖，组成一个具有完整业务功能的 App 应用。但是在组件开发模式下，业务组件又变成了一个个 Application，它们可以独立开发和调试。由于在组件开发模式下，单个业务组件的代码量相比完整的项目差了很远，因此在独立运行时可以显著减少编译时间。如上技术架构图：主要分为四层：宿主层、业务层、依赖约束层和平台层，业务层的划分会随着业务的迭代持续递增。

平台层

平台层按照不同的职责大致分为：基础组件、通用组件、业务组件，业务层可以按需引用提供的各个组件，组件之间无相互依赖引用关系。其中基础组件包含业务无关的各种自研组件和第三方开源库。业务组件层，这一层的所有组件都是业务相关的。

依赖约束层

按需引入平台层的能力，作为依赖约束的入口，统一进行管理。将业务层直接依赖基础库的实现变更为依赖接口，拒绝业务层直接访问基础库，做到基础库可插拔，这样当基础库需要替换时，无需牵动业务。

业务层

业务层主要承载具体的业务模块，也是解耦拆分的重头戏，对应的不同的业务线Bundle，统一只引用中间层的库依赖接口作为能力支撑。基于目前的组件化思想，将业务划分为9个业务模块，每个业务模块也可以看做一个独立的组件，每个业务模块都是独立的maven库，每个业务模块内部具有很高的内聚性，模块之间依赖通过路由总线进行相互通信，这样很好的解决了模块之间相互调用产生的耦合问题。

Driver App 就目前而言，业务模块划分为：登录、首页、Wallet、COD Ticket、Pickup、Delivery、Transport、Return、P2P，其中5个主业务模块（可独立打包）、3个可选业务模块、1个必选业务模块，后续随着业务迭代业务划分可能还会有递增。

业务支撑

业务支撑作为平台层的一部分，这里单独提出来说明一下，完善的App离不开辅助性功能支撑，为项目的迭代、稳定、体验提供保障。业务支撑库主要为Driver App项目提供：Transify服务、日志服务、埋点上报、崩溃上报、性能监控以及内存检测等等，业务支撑的能力仍在不断强化。

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、代码解耦拆分

对当前的项目进行模块拆分，模块分为两种类型，一种是功能组件模块，封装一些公共的方法服务等，作为依赖库支撑业务，比如Wallet、Cod ticket等；一种是业务组件模块，专门处理业务逻辑等功能，这些业务组件模块最终按需组装成一个App，比如Pickup、Transport、Delivery、Return、P2P等。

2.2.2、组件间通信

通过接口+实现的结构进行组件间的通信。每个组件声明自己提供的服务 Service API，这些 Service 都是一些接口，组件负责将这些 Service 实现并注册到一个统一的路由 Router 中去，如果要使用某个组件的功能，只需要向Router 请求这个 Service 的实现，具体的实现细节对于上层业务全然不关心，只需获取到所需要的返回结果就可以了。此外 UI 跳转也是组件间通信的一种，但是属于比较特殊的数据传递。目前比较主流的做法是通过在每个 Activity 上添加注解，然后通过 APT 形成具体的逻辑代码。在组件化架构设计图中 平台层 包含了路由服务组件，目前方式是引用阿里的 ARouter 框架，通过注解方式进行页面跳转。

2.2.3、组件独立运行

通过工程的 build.gradle 文件中依赖的 Android Gradle 插件 id (application 和 library) 来配置工程的类型，但是我们的组件既可以单独调试又可以被其他模块依赖，所以这里的插件 id (application 和 library) 我们不应该写死，而是通过在 module 中添加一个 gradle.properties 配置文件，在配置文件中添加一个布尔类型的变量 isBuildModule，在 build.gradle 中通过 isBuildModule 的值来使用不同的插件从而配置不同的工程类型，在单独调试和集成调试时直接修改 isBuildModule 的值即可。此外 ApplicationId 和 AndroidManifest 文件都是可以在 build.gradle 文件中进行配置的，所以我们同样通过动态配置组件工程类型时定义的 isBuildModule 这个变量的值来动态修改 ApplicationId 和 AndroidManifest。首先我们要新建一个 AndroidManifest.xml 文件，加上原有的 AndroidManifest 文件，在两个文件中就可以分别配置单独调试和集成调试时的不同的配置。

2.2.4、集成调试

每个组件单独调试通过并不意味着集成在一起没有问题，因此在开发后期我们需要把几个组件机集成到一个 App 里面去验证。由于经过前面几个步骤保证了组件之间的隔离，所以可以任意选择几个组件参与集成，这种按需索取的加载机制可以保证在集成调试中有很大的灵活性，并且可以加大的加快编译速度。需要注意的一点是，每个组件开发完成之后，需要设置 isBuildModule 并同步，这样主项目就可以通过参数配置统一进行编译。

2.2.5、依赖隔离

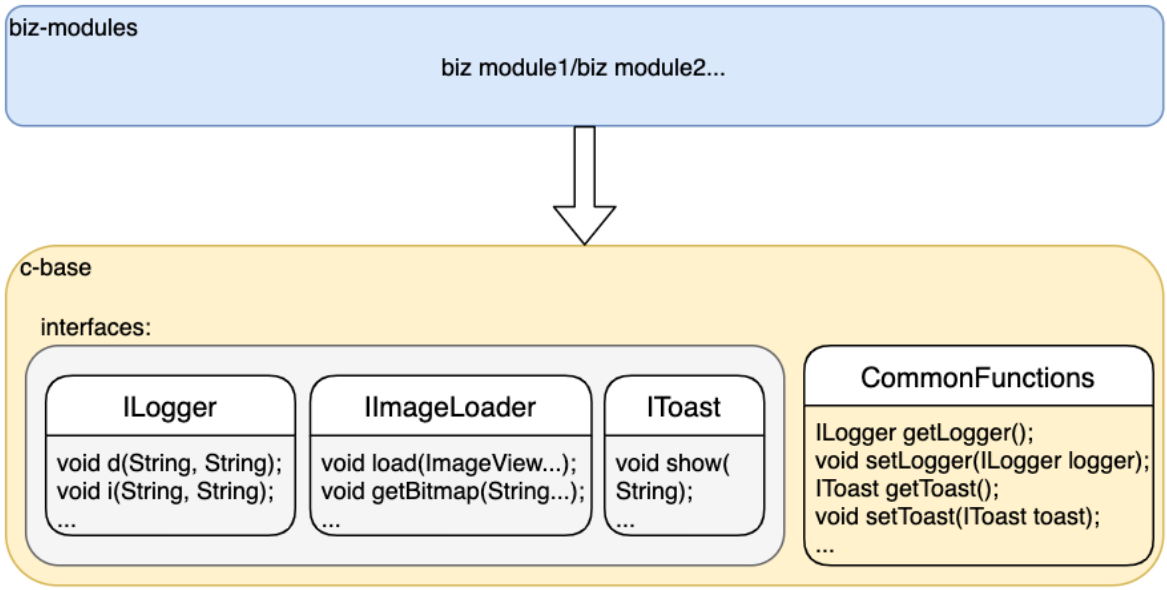
通过以上几个问题的解决方式可以看到，我们在极力的避免组件间及模块与组件间类的直接引用。不过即使通过 平台层 中提供 Service 的方式解决了组件间直接依赖的问题，但是我们在主项目通过 implementation 添加比如对 scan 组件的依赖后，在主项目中依旧是可以访问到 scan 组件中的类的。这种情况下第一是通过面向接口编程，将直接依赖基础库的实现变更为依赖接口，但是只要能直接访问到组件中的类，就存在有意或无意的直接通过访问类的方式使用到组件中的代码的可能。我们希望的业务组件依赖是只有在打包过程中才能直接引用基础库中的类，在开发阶段，基础库中的类我们是不可以访问的。只有这样才能从根本上杜绝直接引用基础库的问题，做到基础库可插拔。对此我们可以在依赖接口的基础上 加上通过 Gradle 提供的方式可以解决，Gradle 3.0 提供了新的依赖方式 runtimeOnly ，通过 runtimeOnly 方式依赖时，依赖项仅在运行时对模块及其消费者可用，编译期间依赖项的代码对其消费者时完全隔离的。

目前整理的依赖隔离方式有以下三种：

1、c-base 提供接口，壳工程注入实现，各模块依赖 c-base 暴露的接口（spx-pda 项目使用该方式）。该方式有以下特点：

- 简单易实现。
- c-base 包含部分接口，存在部分耦合。
- 需要外部注入具体实现。
- 无法版本管理，项目间难以复用。
- 维护成本低，适用于简单项目。

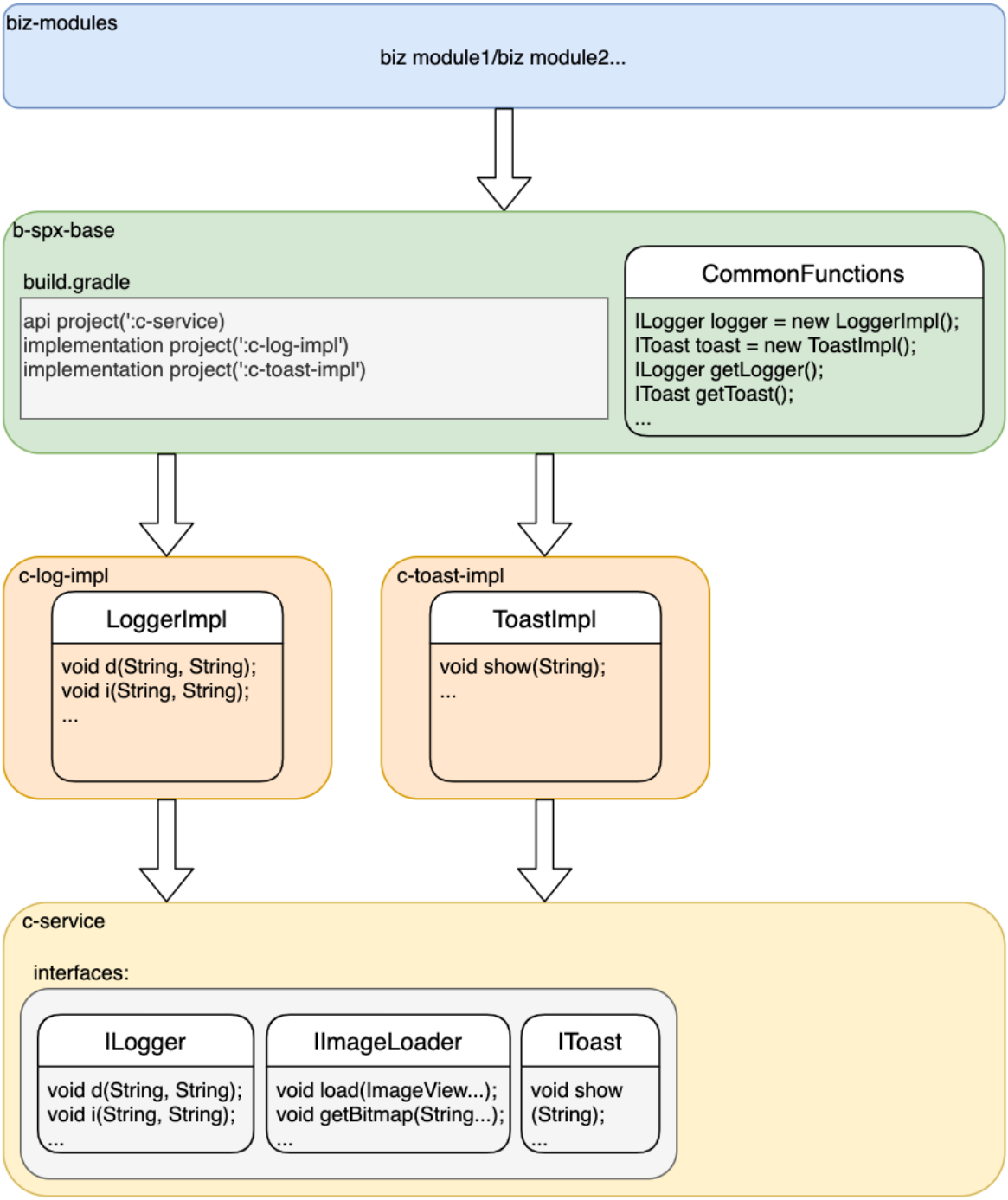
依赖示意图如下：



2、增加依赖约束层 b-spx-base 及接口定义层 c-service，减轻 c-base 负担（wms-pda 项目部分使用该方式）。该方式有以下特点：

- 依赖约束层对外暴露接口，内部指定具体实现。
- c-service 包含所有接口定义，版本管理较难，项目间复用还是有难度。
- 维护成本稍有增加，适用于项目特有的依赖隔离，如 login、home 等业务模块的接口下沉。

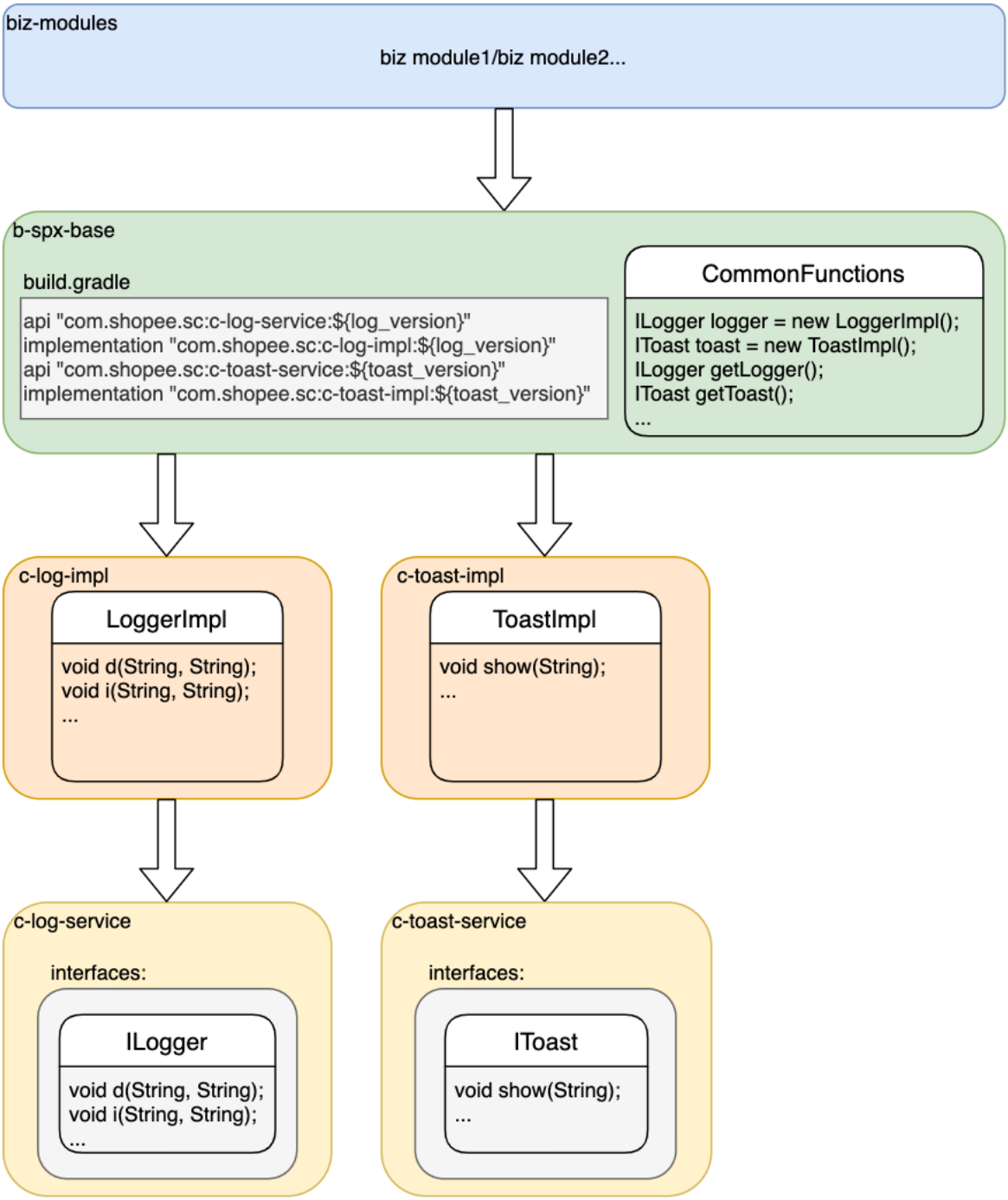
依赖示意图如下：



3、在方式2的基础上，进一步拆分出独立的接口模块。该方式有以下特点：

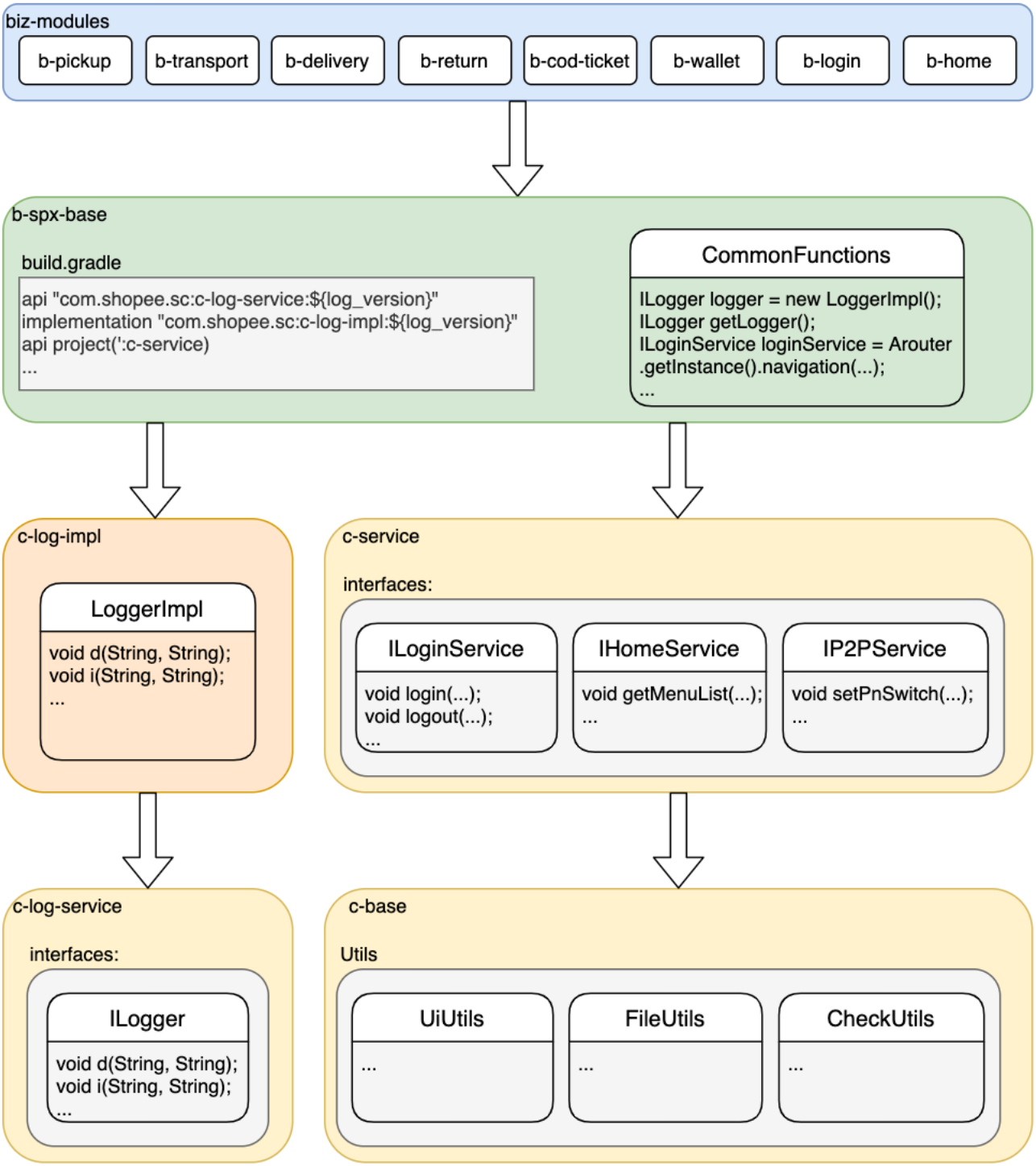
- 特定功能的接口模块及实现模块都可以单独发布至maven，利于版本管理，易于项目间复用；
- 维护成本相对较高，适用于项目间通用且相对稳定的依赖隔离，如 log、imageLoader 等。

依赖示意图如下：



综合考虑开发成本、维护成本及可复用性，建议组件化演进结合使用方式 2 及方式 3 ，其中项目特有的依赖隔离（如 login、home 等业务模块的接口下沉）使用方式2，项目间通用的依赖隔离（如 log、imageLoader 等基础组件）使用方式3。

目标整体依赖示意图如下：



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

3.1 如何把握好在需求同步迭代的节奏中逐步完成拆分

- 业务模块解耦拆分过程中，及时合并Release代码；
- 重构期间，日常迭代业务开发时，新增部分通过开关控制，相对独立的需求进行分包，降低合并的难度；
- 把自测工作做到位，保证自测覆盖和资源投入；

3.2 需要足够的时间、人力来支持对项目进行回归测试

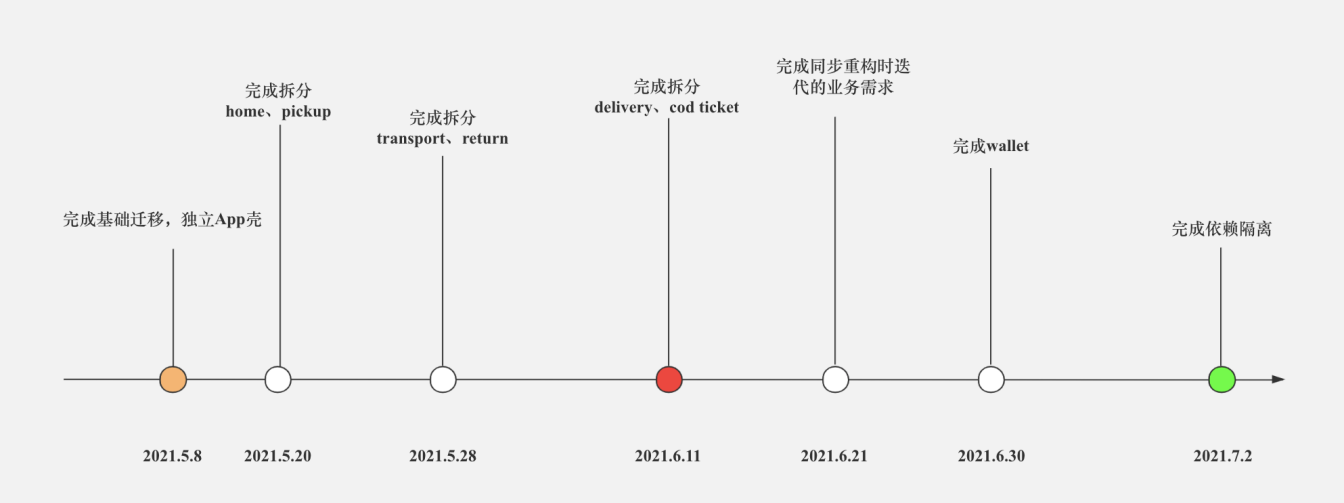
- 根据重构计划协调好QA资源，可分4个主业务模块分阶段进行回归；

其他

以上未包含的其他补充项说明

RoadMap

详见重构计划



上线方案

- 分模块上线，先以当前的App模块作为宿主，拆分出的模块作为独立组件，一起打包上线。业务模块上线顺序： Pickup → Transport/Return → Delivery；