[SCA] 端作业 / 离线框架

1.背景

Driver App 在批量揽收的场景效率比较低，每次扫码-服务端校验都需要 1s 以上，比较影响作业人员的效率，对比 JT 的产品体验，还有待提高。产品侧期望每次扫码校验是瞬时的，有错误可以立刻提醒，提高作业的效率。所以需要把服务端的校验逻辑下发到客户端，由客户端来做校验，甚至是客户端可以实现部分的业务流程的处理，最终把结果同步给服务端即可。

同时，在 Q3，Driver App 支持了 Pickup 和 Delivery 的离线能力。由于离线功能都是在各种业务场景中硬编码实现，维护成本很高，且没法统一做业务开关，业务兜底难度很大。

2.目标

实现一套「端作业 / 离线」技术框架，在客户端作业逻辑的前提下，尽量满足下面条件

1. 低成本业务适配：尽量减少业务侵入，对存量业务尽量降低业务的改造成本，以及后期的维护成本，做到 1 → N 的横向低成本业务适配
2. 业务可用性保障：可以加统一业务开关，实现业务一键兜底
3. 降低整体架构的复杂度：服务端可以不做修改，客户端保证业务执行顺序，降低整体业务架构的复杂度

3.技术方案

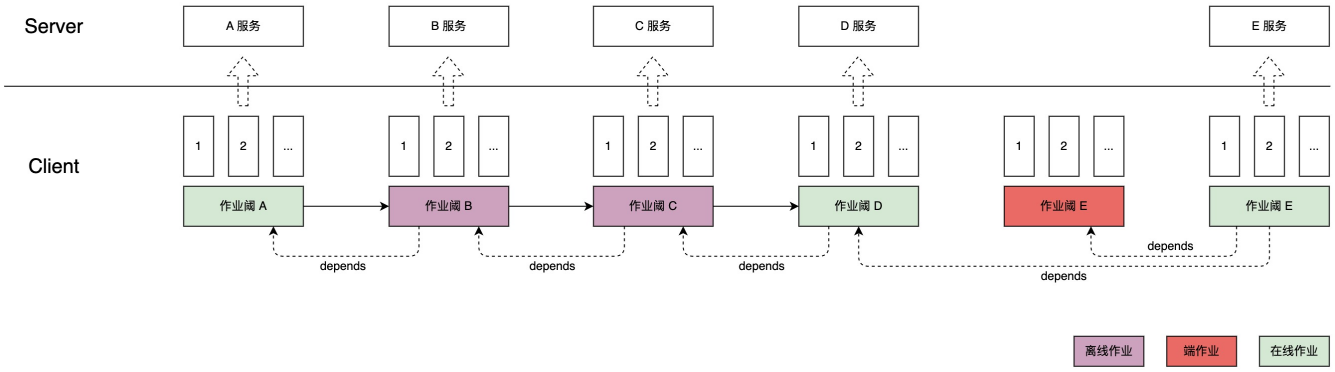
3.1 作业阈抽象

SPX 的业务是一种串型逻辑，技术本质上是一个状态机。一个包裹的主要状态如下：订单创建 → 揽收中 → 揽收完成 → 交接给站点 → 打包出库 → 干线运输 → 派送站入库 → 派送出库 → 派送中 → 派送完成。这里后置任务能执行的前提是前置步骤已经完成，一环扣一环的，所以我们的离线 / 端作业也需要按这个思路进行业务抽象，尤其是要建立业务的相互依赖关系。

作业阈：单个或一组并行操作的集合，对应一个任务状态机中的节点，例如

- Pickup：扫码签收，单个接口
- PickupDone：上传凭证、签名
- Handover：交接

作业阈依赖：在一系列连续的作业中，后置作业需要依赖前置作业完成



离线模式：在线作业 D 执行的时候必须等离线作业 B、C 已经成功才能进行。在我们的案例中的表现可以是

- 作业阈 A：拉取 Pickup 任务列表
- 作业阈 B：扫码揽收
- 作业阈 C：凭证、签名上传
- 作业阈 D：handover，任务到站点交接

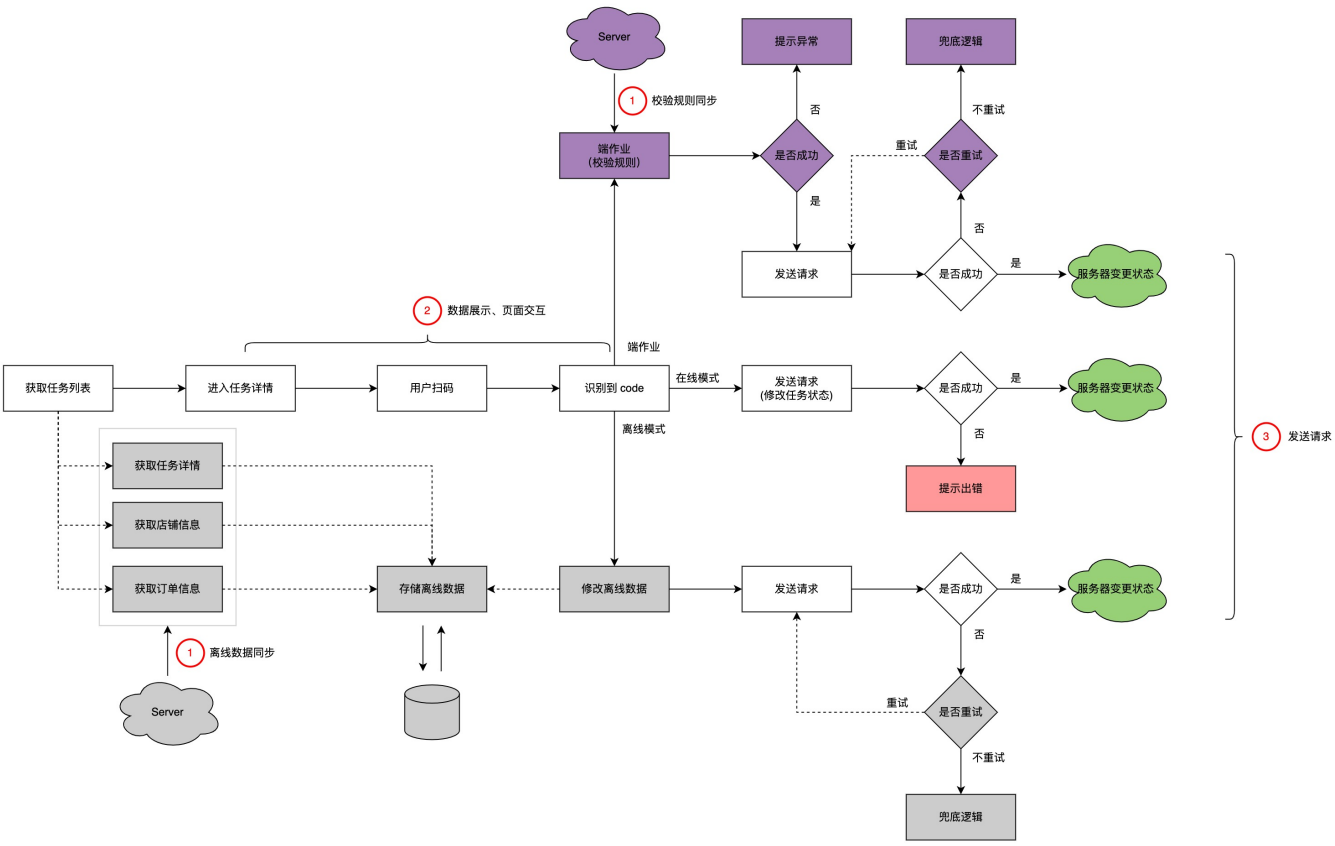
端作业模式：在线作业 E 执行的时候必须等前置作业 E 执行完成，且在线作业 D 执行完成。

3.2 具体作业阈单元的流程

一个具体的业务场景的流程是

- 数据下行：获取数据
- 数据展示：修改数据（页面操作）
- 数据上行：提交数据

以 Pickup 流程为例子，总体流程如下



不管是端作业模式还是离线作业模式，最终调用服务端接口还是正常在线的接口

- 在线模式：直接发送服务端请求，失败直接提示
- 离线模式：修改离线数据，后台重复发送数据，直到成功
- 端作业：优先校验状态，成功后后台重试；失败则立即提醒

这里端作业模式和离线模式的区别在于在于

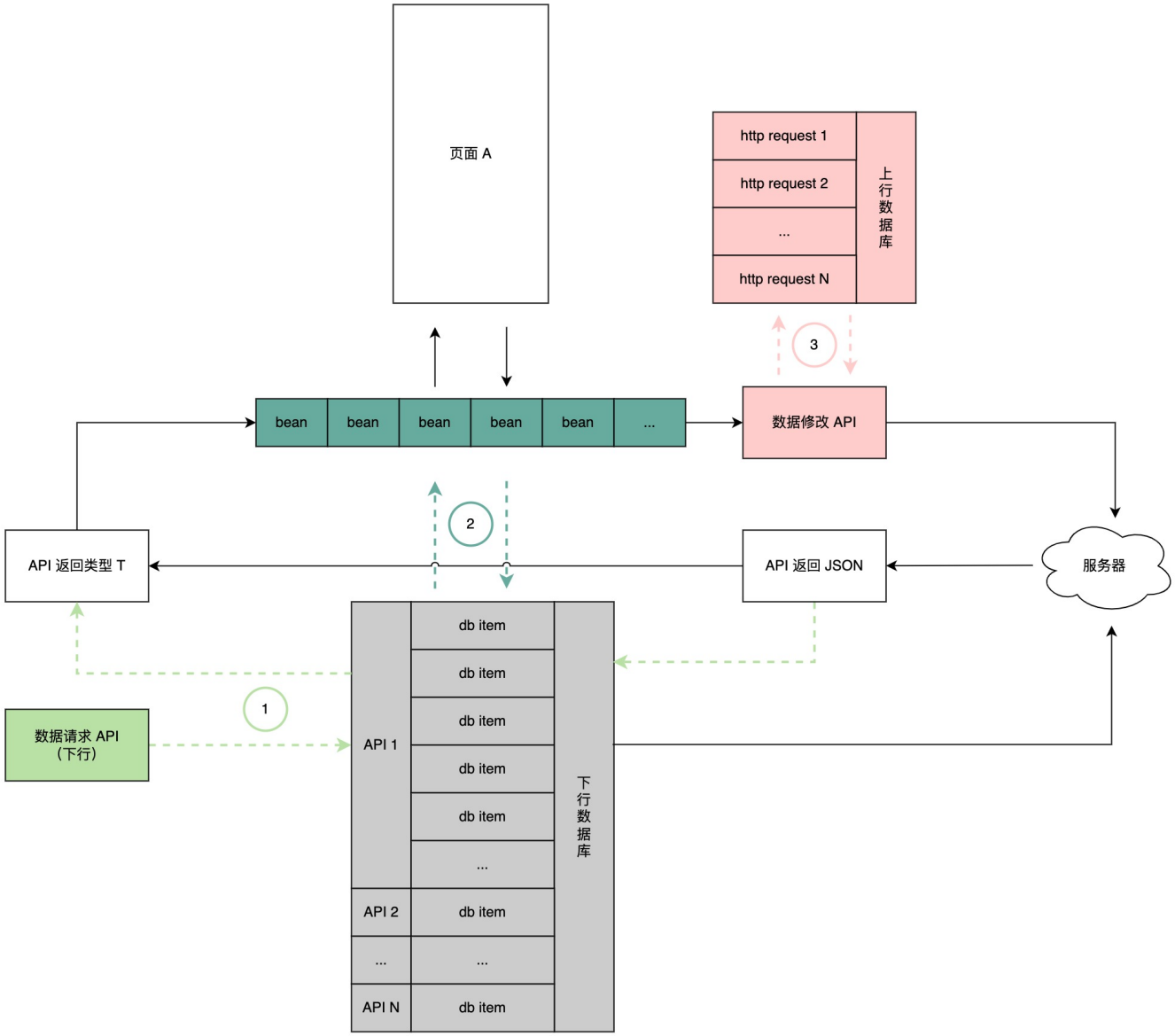
- 端作业模式的数据并非用来做展示的，而是当前接口服务端规则的简版规则
- 离线模式的数据是用来做 UI 展示的

但是不管怎么样，最终都会调用原来的服务端接口。所以在现有流程中，如果要尽量少修改原有业务流程，必须想办法去 hook 原有流程节点，插入离线 / 端作业的校验规则。

这里比较重要的三个点，都要考虑「在线 / 离线 / 端作业」的情况

1. 数据同步（数据下行）
    a. 在线模式：正常获取数据
    b. 离线模式：正常获取数据，需要预获取关联的数据，存储到本地表
    c. 端作业模式：同步检测规则的数据到本地表
2. 页面展示（数据展示）
    a. 显示 UI：几种模式无差别
    b. 执行操作
        i. 在线模式：修改内存数据
        ii. 离线模式：修改内存数据、修改本地表数据
        iii. 端作业模式：校验本地表数据，修改本地表数据
3. 提交重试（数据上行）
    a. 在线模式：直接提交
    b. 离线模式：缓存当前发送数据，不断重试，重启可重新提交
    c. 端作业模式：与离线模式相同

这里在整体实现逻辑的设计上，尽量复用在线逻辑，整体思路如下



将数据下行、数据展示、数据修改三大功能做抽象，统一设计一个框架处理这三个关键点，使业务侵入足够小，接入成本足够低，同时可以在这三个统一的卡口设置统一的开关做兜底，开关关闭，一键切换到在线模式。

3.2.1 数据下行&存储

对于不同的业务逻辑，离线数据 / 端作业数据会有所不同，这里框架需要满足各种业务场景的数据的增删查改，两种方法：

1. 每个场景单独的表，并且把各个 Bean 对象的字段依次存
    a. 优势：DB 对象可以直接映射为 Bean 对象，且可以单独修改某一个列，比较方便
    b. 不足：数据库表会比较复杂，且会有很多张表，DB 不好统一管理
2. 所有场景统一存储为相同的结构，保存完整 Bean 对象字符串的前提下，预留需要修改的字段为单独的列
    a. 优势：表结构简单，字段比较少，不用考虑 Bean 对象修改需要升级 DB 的问题；可以做统一的监控等
    b. 不足：在 DB 对象映射为 Bean 对象的时候需要做一次业务场景硬编码适配

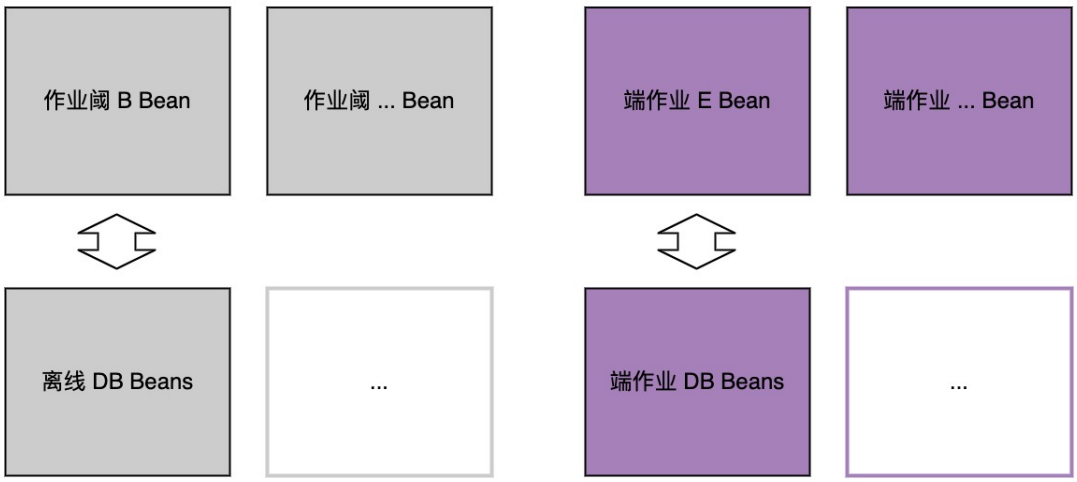
这里倾向于选择第 2 种方式。

DB Bean:

字段名	类型	含义	其他补充
url	String	API 的全路径	
unique_id	String	数据单元的唯一 ID	

tag	String	额外补充	可用来做业务场景标识
total_bytes	byte[]	服务端返回的 JSON 数据	
create_time	long	数据创建时间	
update_time	long	数据更新时间	
max_age	long	数据有效期	
slot_1	String	预留字段	用来给上层业务做快速的修改，而不用修改大的 JSON
slot_2	String	预留字段	
slot_3	String	预留字段	
slot_4	String	预留字段	
slot_5	String	预留字段	
slot_6	String	预留字段	
slot_7	String	预留字段	
slot_8	String	预留字段	
slot_9	String	预留字段	
slot_10	String	预留字段	

- DB Bean：存通用字段，不用考虑 DB 字段变更等问题，便于升级和迁移
- 业务 Bean：将 DB 映射为业务需要识别的具体数据对象



3.2.2 数据展示 & Bindable Bean

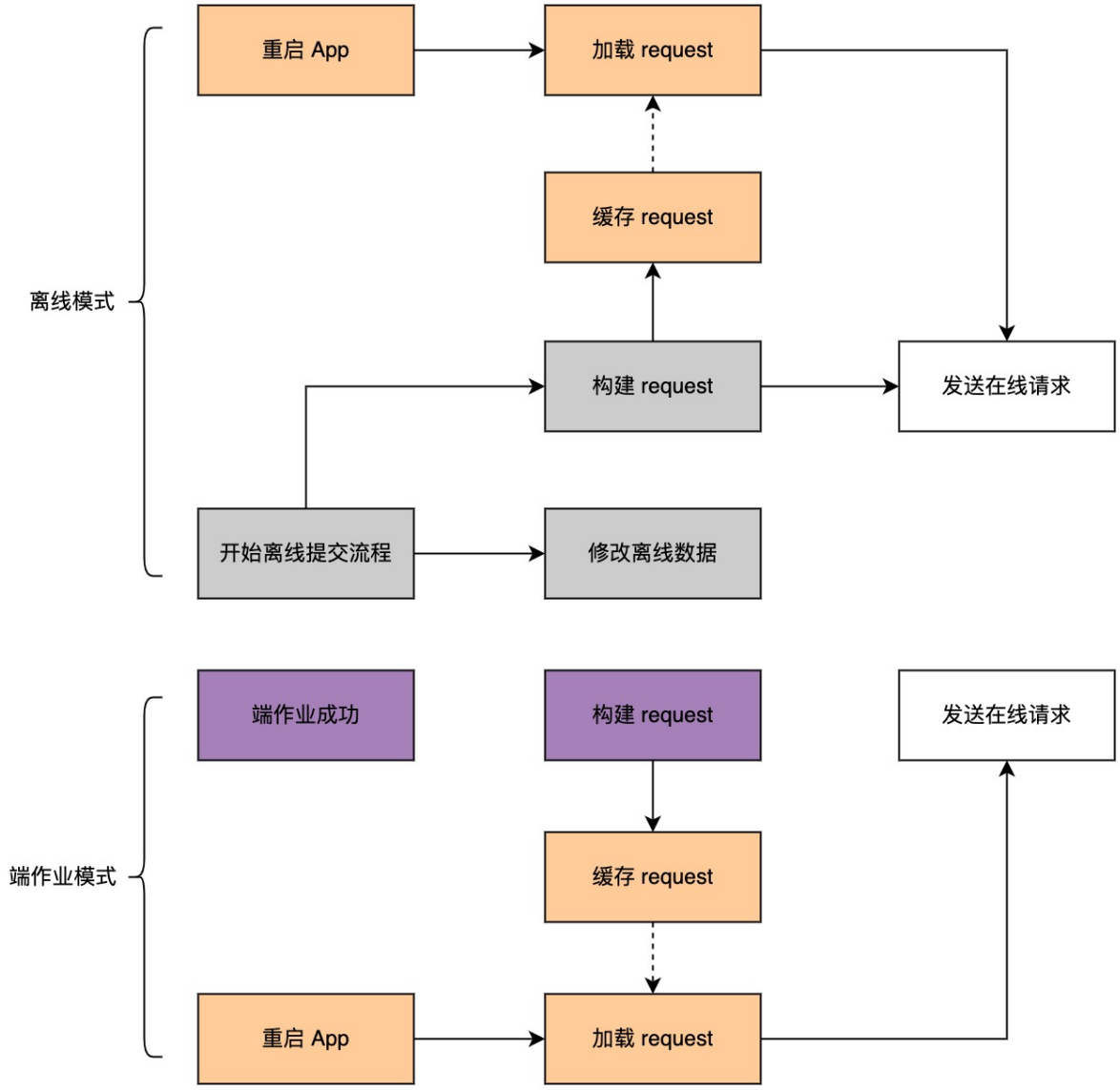
页面显示的 Bean 对象需要与 DB Bean 对象做无缝转换，尽量减少业务数据存储的手工编码环节，下文会详述。

3.2.3 数据上行&重试提交

这里需要缓存当前请求接口的数据，两种方法

1. 将数据提交接口的数据存储为一张表，当用户重启可以使用这张表的数据来重新构建 API request 请求服务端
    a. 优势：暂无
    b. 不足：
        i. 新增了一张表，也存在 DB 管理的问题，每个接口一张表的话表规模会膨胀
        ii. 重启时每个业务场景都需要 load 当前接口的表，构建 request，业务侵入很重
2. 将当前接口最终 http request 信息存储起来（对于离线模式，也要修改下离线字段，保持统一）
    a. 优势：表结构简单，容易管理，且与业务场景无关联，无业务侵入，可以统一监控
    b. 不足：对于离线模式，会多存一条数据

这里会选择第 2 种方案。



Request 数据缓存

分类	字段名	类型	说明
API 配置	tag	String	业务场景标识
	unique_id	String	唯一健
	create_time	Long	创建时间
	update_time	Long	更新时间
	retry	int	一共重试几次
	retry_count	int	当前重试次数
	group_id	String	作业阈 ID
	depend_on	String	作业阈依赖
	job_id	String	作业 ID
请求体	method	String	GET / POST
	url	String	完整的 url
	headers	Map	k=v

	body	byte[]	

3.3 如何减少业务侵入

3.3.1 作业阈配置

使用注解在原有接口定义上增加作业阈配置，不改变原有业务代码逻辑。

- @JobGroup：定义作业阈
- @Job：定义一个 API，抽象一个作业

作业阈

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface JobGroup {
    /**
     * @return  ID
     */
    String id() default "";

    /**
     * @return  ID
     */
    String dependOn() default "";
}

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Job {
    /**
     * Job
     * @return  ID
     */
    String id() default "";
}
```

示例：

```
@JobGroup(id = "request_token")
@Job(id = "create_token")
@Upload()
@GET("/3/authentication/token/new")
Observable<RequestToken> createRequestToken(@NonNull @Query("api_key") String apiKey);

@JobGroup(id = "login_token", dependOn = "request_token")
@Job(id = "login")
@Upload()
@POST("/3/authentication/token/validate_with_login")
Observable<RequestToken> createSessionWithLogin(@NonNull @Query("api_key") String apiKey,
                                                @NonNull @Body SessionWithLoginParams params);
```

这里定义了两个作业阈

- request_token
- login_token

每一个作业阈下面有一个 Job

第二个作业阈依赖第一个作业阈，这样在第二个Job提交的时候必须等待第一个 Job 结果返回，保证了提交的顺序。

3.3.2 数据下行 API 配置

- @Download：定义数据下行的 API

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Download {
    /**
     *  Http  max-age
     * @return  1
     */
    long maxAge() default 60 * 60 * 1000L;
}
```

示例：

```
@Download(maxAge = 60_1000L)
@GET("/3/movie/popular")
Observable<MovieListResult> getPopularMovies(@NonNull @Query("api_key") String apiKey,
                                             @NonNull @Query("language") String language,
                                             @Query("page") int page);
```

3.3.3 数据展示 & Bindable 配置

- @CacheEntity：定义可转换为 DB Bean 的业务 Bean
- @Slot：定义预留字段
- @Unique：定义唯一健，需要用来查询或者更新 DB Bean

示例：

```java
@CacheEntity(tag = MovieItem.SCENE_ITEM_KEY)
public class MovieItem {
    public static final String SCENE_ITEM_KEY = "_item_result_";

    @Unique
    @SerializedName("id")
    public long id = 0L;

    @SerializedName("backdrop_path")
    public String uri = "";

    @SerializedName("title")
    public String title = "";

    @Slot(slotId = OfflineConstants.SLOT_2)
    @SerializedName("vote_average")
    public float score = 0.0f;

    @SerializedName("original_title")
    public String originalTitle = "";

    @SerializedName("release_date")
    public String releaseDate = "";

    @SerializedName("overview")
    public String overview = "";
}
```

这里使用 CacheEntity 定义了 MovieItem 对象，其中 id 是作为后续更新缓存的唯一索引字段，score 是会在业务逻辑中更新的字段，这里绑定到预留字段的第二个上面，在加载 DB 缓存的时候，会从第二个字段中读取出来，赋值到 score 上。

3.3.4 数据上行 API 配置

- @Upload：定义数据上行的接口

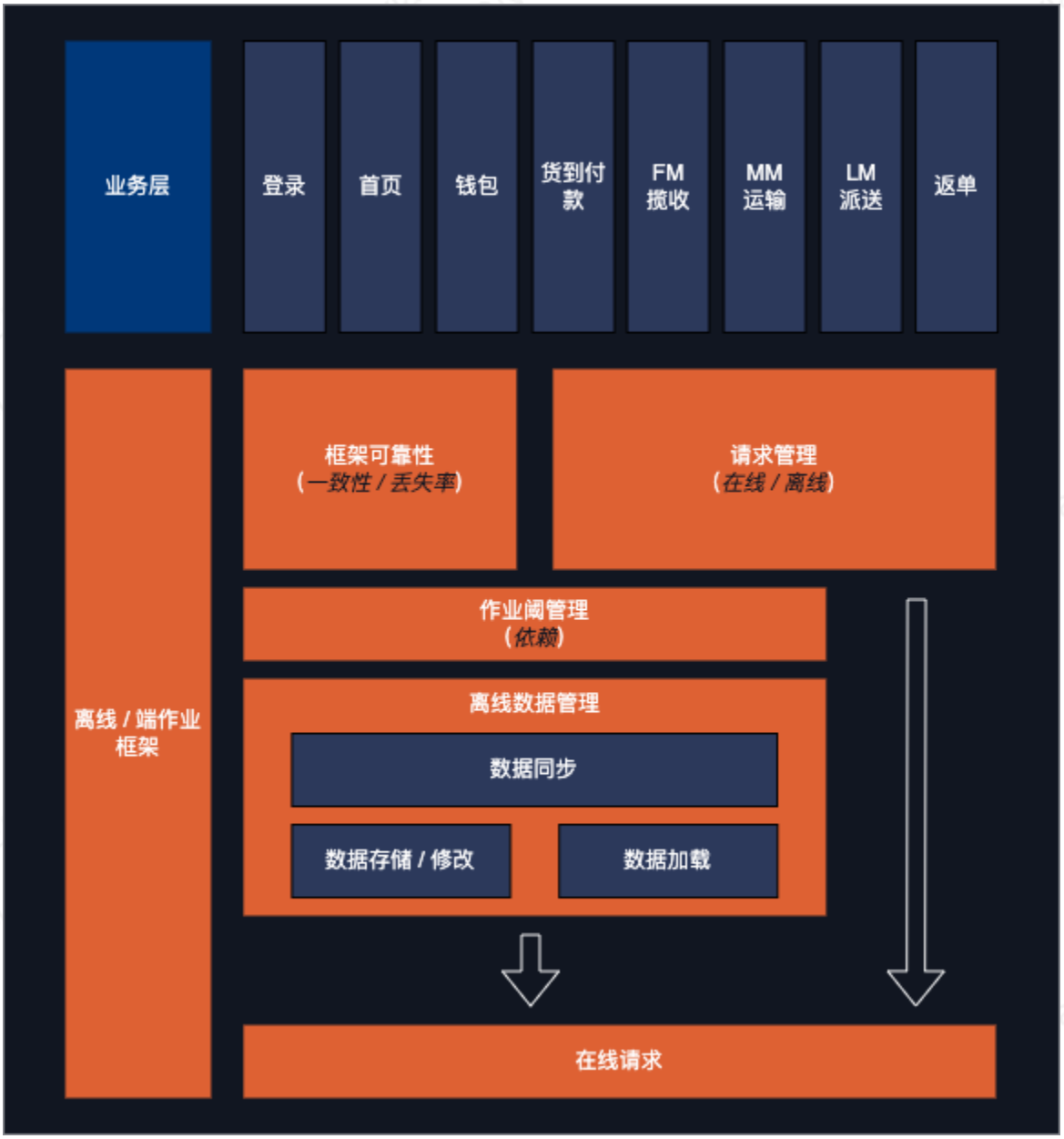
```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Upload {
    /**
     * @return
     */
    boolean firstSilent() default false;

    /**
     *
     * @return
     */
    int retryCount() default Integer.MAX_VALUE;
}
```

示例：

```
@Upload(retryCount = 10)
@POST("/3/movie/{movie_id}/rating")
Observable<RateMovieResult> rateMovie(@Path("movie_id") long movieId,
                                      @NonNull @Query("api_key") String apiKey,
                                      @NonNull @Query("session_id") String sessionId,
                                      @NonNull @Body RateMovieParams params);
```

3.4 完整技术架构图

- 统一在框架的请求管理部分加开关，可一键全部走线上处理



4.推进节奏

时间	完成事项	其他说明
2022.10.31 ~ 2022.11.04	• 技术方案整理 • 与 PM 对齐端作业相关的预期	先做技术储备，暂时还需要等批量扫码上线看用户反馈
2022.11.07 ~	• 技术方案实现	

5.结果

暂无

6.参考资料

- Driver App 离线模式