# 满帮工程技术标准 1.0

## 1. 应用命名标准

### 1.1 工程命名

命名规则：【业务域简称-系统名】 示例：ld-cargo-detail

### 1.2 业务域清单

| 业务域 | 全名 | 简称 |
| --- | --- | --- |
| 长途 | long-distance | ld |
| 冷运 | cold-chain | cc |
| 直客委托 | direct-entrust | de |
| 项目货 | project-cargo | pc |
| 零担专线 | less-than-truckload | ltl |
| 短途 | short-distance | sd |

### 1.3 模块名清单

| 模块后缀名 | 描述 |
| --- | --- |

| | |
|---|---|
| app | Web 应用，提供 H5、App 和小程序 REST 接口，应用规模小的时候，可以提供 RPC 接口 |
| server | Server 应用，提供 RPC 接口，不能提供 REST 接口 |
| admin | Admin 应用，内部人员访问的后台 REST 接口应用 |
| api | 提供 RPC 二方包 API 接口 |

# 2. API 命名标准

## 2.1 通用标准

### 2.1.1 命名应遵循简单易读、统一原则

1. API 应该在业务上下文中被很容易地理解；
2. 统一原则，即对某个动作或实体的描述保持一致，比如同样是描述 `费用金额` ，两个 API 先后使用 `fee` 和 `amount` 来表达，造成前后不一致；
3. API 命名应该通俗易懂，不应使用不易理解的方式来阐述方法适用场景（典型的如使用多重逻辑连接的方式来表达一个 API 的适用场景）。

【反例】

```
/**
 * 根据条件查询最新(也可以说最新购买)的保障包，包含所有状态
 *
 * @param request xxx
 * @return xxx
 */
Result<T>
queryNewestGuaranteePackageContainAllStatus(QueryGuaranteePackageRequest
```

```
request);
```

> 上述反例中，包含哪些状态本身就是查询策略的一部分，没必要体现在方法命名中

### 2.1.2 API 契约应该由 `API 名 + 参数` 共同组成

1. 完整的 API 签名包含 API 名 + 入参类型 + 出参类型；
2. 查询类的接口，如果入参是单个类型，建议在 API 名上采用 `ByXxx` 的后缀来描述查询索引；
3. 当根据主键查询时，采用 `getXxx（Long id）` 形式，无需用 `By` 来标注；

【正 例】

```java
/**
 * xxx
 *
 * @param id xxx
 * @return xxx
 */
public Result<T> getXxx(Long id);

/**
 * xxx
 *
 * @param name xxx
 * @return xxx
 */
public Result<T> getXxxByName(String name);

/**
 * xxx
 *
 * @param code xxx
 * @return xxx
 */
public Result<T> getXxxByCode(String code);
```

4. 如果某个 API 可以通过 API 名 + 参数类型足够说明清楚用途和场景，那么无需通过在 API 名上添加“前缀”或“后缀”来补充说明。
5. 查询类的接口，不建议在 API 名上 采用 By A and B 后缀的方式拼接多个查询条件, 多个条件应采用查询条件的包装类；

【反例】

```
/**
 * xxx
 *
 * @param consigner xx
 * @param shipper xxx
 * @return xxx
 */
public Result<T> getOrderByConsignerAndShipper(Consigner consigner, Shipper shipper)
```

6. 如果查询策略比较多样化，可以考虑包装 `QueryCondition` 对象，增加扩展性的同时且无需频繁增加重载方法。

【示例】

```
/**
 * xxx
 *
 * @param condition xxx
 * @return xxx
 */
public Result<T> getCargoDetail(CargoQueryCondition condition);
```

### 2.1.3 REST 和 RPC API 统一返回 `Result<T>`

1. REST 和 RPC API 统一返回 `commons-api` 中的 `Result` 对象，
2. 不要直接抛业务异常，而是通过不同的 `success` 和 `code` 来表示，不同的返回场景

【Maven 坐标】

```xml
<dependency>
    <groupId>com.ymm.commons</groupId>
    <artifactId>commons-api</artifactId>
    <version>${latest}</version>
</dependency>
```

【示例】

```java
/**
 * xxx
 *
 * @author manbang
 * @since 2023-03-29
 */
public interface UserMapper extends BaseMapper<UserEntity> {

}

/**
 * xxx
 *
 * @author manbang
 * @since 2023-03-04
 */
public class UserServiceImpl implements UserService {

      @Resource
      private UserMapper userMapper;

    /**
     * 根据用户 Id 查询，用户信息
     *
     * @param userId 用户 Id
     * @return xxx
     */
    public Result<UserDTO> getUser(String userId) {
        return userMapper.findById(userId)
              .map(Result::success)
                .orElseGet(Result::failure)
                .mapResult(UserDTO.class);
    }
```

```
34
35 }
```

## 2.2 RPC 接口

### 2.2.1 命名风格

API 方法名以 `动词 + 名词` 动宾短语命名，名词代表资源，动词代表对资源实施的操作。

如：

```java
/**
 * 发布货源
 *
 * @param cargo 货源
 * @return xxx
 */
public Result<CargoDTO> publishCargo(Cargo cargo);

/**
 * 查询订单，返回值不会为 <code>null</code>
 *
 * @param condition 查询订单条件
 * @return 满足条件的订单列表，如果没有符合条件的订单，返回空集合
 */
public Result<List<OrderDTO> queryOrders(OrderCondition condition);
```

API 命名应该采用驼峰方式分割，如果是操作资源下面的子资源，跟在主资源后面即可，

如：

```java
/**
 * 更新指定订单价格项
 *
 * @param order 订单实体
 * @param item 价格项
 * @return 更新成功，返回 <code>true</code>，更新失败，返回 <code>false</code>
 */
public boolean updateOrderPriceItem(Order order, PriceItem item);
```

### 2.2.2 不应使用过多的平铺参数，尤其是包含相同数据类型的情况下

1. 方法参数限制：

    a. 方法的参数个数不应超过 4 个

    b. 如果方法的参数列表超过 **4 个，应该以包装类型进行封装**；增加扩展性的同时且无需频繁增加重载方法

2. 必须使用包装类：

    a. 原始类型不设置初始值，会有默认值

    b. 包装类如果不设置初始值，默认是 `null`

### 2.2.3 API 方法签名和暴露的协议解耦

API 方法签名应该和具体的暴露协议无关，一个被声明的 API，既可以使用 REST 协议暴露 Endpoint，也可以使用 RPC 协议暴露对外端点。因此不建议在 API 方法上添加，如 `RPC` 、 `Pigeon` 和 `IronMan` 等关键字；

### 2.2.4 API 命名中动词只能出现一个

每个 API 应只完成单一的职能，方法名中，动词只能出现一个

【反例】

```
/**
 * 更新支付状态并增加退款金额
 *
 * @param status xxx
 * @param deductAmount xxx
 * @return xxx
 */
public Result<Boolean> updatePayStatusAndIncreaseDeductAmount(PayStatus status, Long deductAmount);
```

## 2.3 REST 接口

### 2.3.1 接口格式

/ 应用名 / 模块名 / [子模块名] / 接口名

### 2.3.2 格式说明

1. 路径字符全部小写，多个单词用中划线 `-` 连接，不能使用下划线 `_`
2. 不允许出现空格，文件扩展名
3. 目录层次尽量少，3 ~ 5 级 为宜
4. 简单好记，体现功能语义，比如： `/{应用名}/cargo-detail/shipper`
5. 不要在path中携带参数，不利于流量分析
6. 尾部不额外加反斜线符号 `/`
7. 子模块名可选

### 2.3.3 通用接口名称规范

【示例】

| 场景 | 命名 | 示例 / 备注 |
| --- | --- | --- |
| 首页 | /{context-path}/{module}/index | |
| 列表页 | /{context-path}/{module}/list | 订单列表： order/list |
| 搜索功能 | /{context-path}/{module}/search | 订单搜索：order/search |
| 详情页 | /{context-path}/{module}/detail | 订单详情： order/detail |

| | | |
|---|---|---|
| 添加 | /{context-path}/{module}/add | |
| 删除 | /{context-path}/{module}/delete | |
| 修改 | /{context-path}/{module}/update | |

### 2.3.4 响应格式

返回 `commons-api` 中的 `Result` 对象，除特殊场景外，响应格式统一为：

```
{
  "success": false,
  "code": "1001",
  "msg": "支付失败",
  "data": {}
}
```

1. success：代表操作是否执行成功，成功返回 `true` ，失败返回 `false`
2. code：代表操作的响应编码，操作失败和成功，都可通过此字段来返回不同具体原因
3. msg：代表操作的响应说明文案，与 `code` 类似，成功与否都可以返回对应文案
4. data：是业务数据部分，由业务自己定义

### 2.3.6 分页标准

1. 页码变量统一为 `page` ，默认从 0 开始；
2. 页大小变量统一为 `size` ；
3. 分页对象返回类型为 `commons-api` 中的 `PageView` 对象；格式为：

```
{
  "page": "0",
  "size": "10",
  "total": "108",
  "pages": "11",
```

```
  "list": [],
}
```

1. page：页码
2. size：单页最大数据数量
3. total: 总数据数量
4. pages: 总页数
5. list：数据列表

# 3. 代码仓库标准

## 3.1 代码仓库目录结构

应用在代码仓库一般为三层结构：

1. **Group**：研发团队负责的业务域
2. **系统名**：业务系统的名称，提供 **git clone** 地址
3. **模块名**：系统内的各个工程模块

### 3.1.1 简单工程

一个新的项目，上线之初，如果功能比较简单，可以直接通过 REST 对外提供服务，此时推荐如下结构：

```
[group]                                  (1)
  ├─[xxx]                                (2)
  │  ├─ xxx-app                          (3)
  │  ├─ xxx-api                          (4)
```

（1）group 名称，即在 gitlab 中创建的 group
（2）业务系统名称，即在 group下创建的 repo，包含其 git clone 地址
（3）app 模块，提供 REST 服务，系统默认推荐的模块

（4）API 模块，提供 RPC 服务(如果需要)

【示例】长途业务下面创建一个货源详情的业务，那么一开始结构大致如下：

```
long-distance                                   (1)
  ├─ cargo-detail                               (2)
  │  ├─ ld-cargo-detail-app                     (3)
  │  ├─ ld-cargo-detail-api                     (4)
```

（1）group 名称
（2）业务系统名称，各模块的父工程，git clone 地址
（3）cargo-deail 下的 app  模块
（4）cargo-deail 下的 API 模块

### 3.1.2 模块化工程

当系统规模逐渐变大，可以逐渐模块化。我们可以按照系统功能进行模块的拆分，此时结构大致如下

```
[group]                                         (1)
  ├─ [xxx]                                      (2)
  │  ├─ xxx-app                                 (3)
  │  ├─ xxx-api                                 (4)
  ...
  │  ├─ xxx-[custom]                            (5)
```

（1）group 名称，即在 gitlab 中创建的 group
（2）业务系统名称，即在 group 下创建的 repo，包含其 git 地址
（3）app 模块，提供 REST 服务，系统默认推荐的模块
（4）api 模块，提供 RPC 服务（如果需要）
（5）业务系统自定义模块，根据自身业务进行拆分

【示例】随着长途下货源业务的越来越复杂，可能会有如下结构

```
long-distance                                   (1)
  ├─ cargo-detail                               (2)
  │  ├─ ld-cargo-detail-app                     (3)
  │  ├─ ld-cargo-detail-api                     (4)
  │  ├─ ld-cargo-detail-blackboard              (5)
  │  ├─ ld-cargo-detail-map
```

```
7    │  ├─ ld-cargo-detail-plugin-visibility
8    ├─ cargo-management                          (6)
9    │  ├─ ld-cargo-management-app                (7)
10   │  ├─ ld-cargo-management-api                  (8)
11   │  ├─ ld-cargo-management-blackboard
12   │  ├─ ld-cargo-management-map
13   │  ├─ ld-cargo-management-plugin-visibility
14   ├─ cargo-admin                               (9)
15   │  ├─ ld-cargo-admin
16
```

(1) 示例的 group 名称
(2) 示例的业务系统名称，cargo-detail 中模块的父工程，提供 git 地址
(3) app 模块，提供 REST 服务
(4) API 模块，提供 RPC 服务
(5) 小黑板模块，跟小黑板相关的功能，内聚在此模块
(6) long-distance可以包含多个业务系统，譬如 cargo-management，提供 git 地址
(7) app 模块，提供 REST 服务
(8) api 模块，提供 RPC 服务
(9) 示例的后台管理系统

## 3.2 托管范围

所有需要上线的文本信息，都应该进入代码仓库，以便能够回溯版本信息；系统编译发布过程中的产物，不应进入仓库。每个代码仓库，必须包含 `README.md` 文件，用于说明系统背景和功能。必要时，不同模块下，也应该有 `README.md` 文件。

### 3.2.1 入库文件

| 文件类型 | 是否强制 | 示例 / 备注 |
| --- | --- | --- |
| 源代码 | ✅ | |
| .gitignore | ✅ | |

| | | |
|---|---|---|
| README.md | ✅ | 应用说明文件 |
| UML 设计文件 | | 数据库建模文件等 |
| SQL DDL | | |

### 3.2.2 黑名单文件

【示例】不允许进入仓库

| 文件类型 | 示例 / 备注 |
|---|---|
| SQL DML | |
| 包含账号密码的文件 | |
| 涉及用户敏感信息的文件 | |
| 其他安全禁止传播的文件 | |
| exe 文件 | |
| 大型且频繁更新的文件 | 大于 100M 的文件 |
| IDE 产生项目管理文件 | .idea<br>.project |
| 编译工具产中的中间件文件 | targapdiset/<br>classes/<br>.class |

### 3.2.3 .gitignore 示例

```
# Created by .ignore support plugin (hsz.mobi)
*.class
*.log
*.ctxt
.mtj.tmp/
*.jar
*.war
*.nar
*.ear
*.zip
*.tar.gz
*.rar
hs_err_pid*
Thumbs.db
Thumbs.db:encryptable
ehthumbs.db
ehthumbs_vista.db
*.stackdump
[Dd]esktop.ini
$RECYCLE.BIN/
*.cab
*.msi
*.msix
*.msm
*.msp
*.lnk
target/
pom.xml.tag
pom.xml.releaseBackup
pom.xml.versionsBackup
pom.xml.next
release.properties
dependency-reduced-pom.xml
buildNumber.properties
.mvn/timing.properties
.mvn/wrapper/maven-wrapper.jar
.idea/
cmake-build-*/
*.iws
```

```
out/
.idea_modules/
atlassian-ide-plugin.xml
com_crashlytics_export_strings.xml
crashlytics.properties
crashlytics-build.properties
fabric.properties
.DS_Store
.AppleDouble
.LSOverride
Icon
._*
.DocumentRevisions-V100
.fseventsd
.Spotlight-V100
.TemporaryItems
.Trashes
.VolumeIcon.icns
.com.apple.timemachine.donotpresent
.AppleDB
.AppleDesktop
Network Trash Folder
Temporary Items
.apdisk
*.iml
### STS ###
.apt_generated
.classpath
.factorypath
.project
.settings
.springBeans
.sts4-cache
**/test-output/

### NetBeans ###
/nbproject/private/
/build/
/nbbuild/
/dist/
/nbdist/
/.nb-gradle/
```

```
82
83 ### idea test config
84 *.http
```

### 3.2.4 版本管理规范

【格式】<主版本>.<次版本>.<补丁版本>

【示例】1.2.34

> 主版本：大版本，有重大重构时，升级版本
>
> 次版本：小版本，有增加接口时，升级版本
>
> 补丁版本：补丁版本，已有功能 Bug 修复时，升级版本

## 3.3 代码分支规范

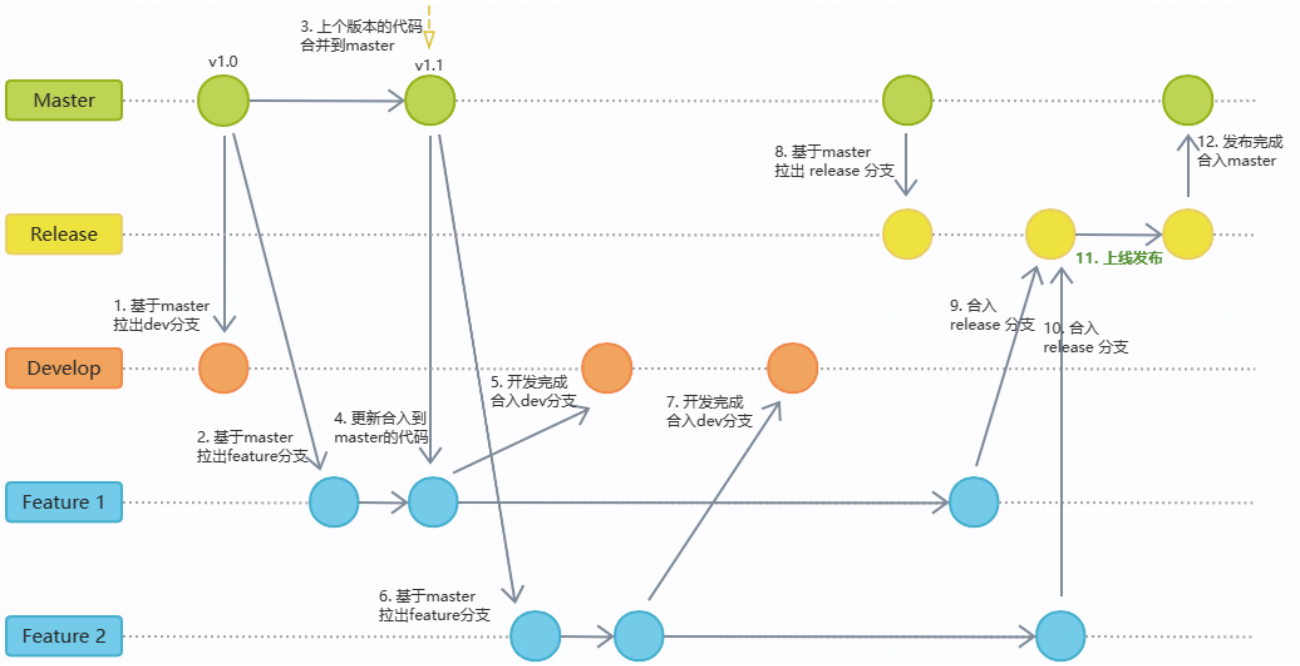
### 3.3.1 分支说明

| 分支名称 | | 分支简介 | 命名规范 | 备注 |
| --- | --- | --- | --- | --- |
| **Master** | 主干分支 | 线上最新代码/主干分支 | master | 1、所有分支均从 Master 创建<br>2、该分支不支持任何直接修改操作<br>3、该分支所有代码更新均应由其他分支合并进入<br>4、原则上，只支持来自 Release（稳定版）分支的代码合入 |

| | | | | |
|---|---|---|---|---|
| **Release** | 发布/回归分支 | QA 环境回归、发布使用分支 | dmpt 自动命名 | 1、该分支代码从 Master 创建<br>2、功能上线验收并灰度完成后，合并至 Master<br>3、原则上，只接受来自 Feature（需求）分支的代码合入<br>4、该分支禁止直接进行功能开发 |
| **Dev** | 开发分支 | 首轮测试使用分支 | dmpt 自动命名 | 1、该分支代码从 Master 创建<br>2、功能开发、自测完成后，代码由 Feature（需求）分支合入<br>3、原则上，只接受来自 Feature（需求）分支的代码合入<br>4、该分支禁止直接进行功能开发<br>5、**该分支禁止合并到其他分支** |
| **Feature** | 需求分支、BugFix 分支 | 需求开发、自测，BugFix 使用分支 | dmpt 自动命名 | 1、该分支代码从 Master 创建<br>2、一个需求的功能开发、自测，对应一个 Feature 分支<br>3、线上 Bug 的修复，使用 Feature 分支，多个 bug 可在同一个分支，通过同一个 dmpt “线上问题”管理<br>4、所有研发过程（开发、自测、联调、功能测试 `QA` 环境回归、发布）中产生的代码变更，必须由 Feature 变更发起，并合并至指定的分支 |

### 3.3.2 开发工作流

多分支并行开发提交与发布的流程：




1. 基于 Master 拉 Dev 分支
    a. 针对需求拉取 Dev 分支
    b. 可通过 dmpt `重建 Dev` 功能创建新分支，`重建 Dev` 前要通知 TL，确认重建的必要性；在 `重建 Dev` 分支后，重建人需集成之前已经集成到 Dev 的分支代码
2. 基于 Master 拉 Feature 1 分支
    a. 原则上一个需求（Feature）拉取一个 Feature 分支
    b. 通过 dmpt `线上变更` 功能创建新分支
3. Master 分支有代码合入
    a. Release 分支上线完成后会自动合入 Master 分支
4. 针对 Master 分支代码合入进行更新
    a. 在开发过程中，若 Master 有代码合入，研发需及时将更新同步至 **Feature** 分支
5. Feature 1完成开发自测，合入 Dev 分支
    a. 可基于 dmpt 进行分支合并操作，涉及到代码冲突时，需手动操作

b. 当联调完成以后，确认提测之前，合入 Dev 分支

c. 涉及到代码冲突时，需要研发手动合并，并在 dmpt 上选择 `手工处理（冲突|异常）` ==> `已手动合入` ，最终 dmpt 上显示 `手动合并` ，代表合并完成

d. 如果 Feature 合并进 Dev 之后再次提交了新的代码，需要在 dmpt 通过 `合并更新` 操作进行更新

e. Dev 分支不要操作 `退出集成` ，容易引发代码混乱

6. 基于 Master 拉 Feature 2 分支

a. 同步骤 2

7. Feature 2 完成开发自测，合入 Dev 分支

a. 同步骤 5

8. 基于 Master 拉 Release 分支

a. 通过 dmpt `+ Release` 建立 Release 分支

9. Feature 1 合入 Release 分支

a. 可基于 dmpt 进行分支合并操作；

b. 如果出现代码冲突，由测试通知研发进行手动合并，冲突解决操作见步骤 5

10. Feature 2 合入 Release 分支

a. 同步骤 9

11. 上线发布

a. 分支无变化

12. 发布完成，合入 Master

a. Release 分支上线完成后会自动合入 Master 分支

# 4. 分包标准

## 4.1 简单应用的分包规范：

简单应用一般只包含两个模块，api 模块和 app 模块。

### 4.1.1 API 模块的分包标准如下

```
com.ymm.xxx.api
  ├─ exception                                (1)
  ├─ util                                     (2)
  └─ [module name]                            (3)
    ├─ service                                (4)
    ├─ dto                                    (5)
    └─ request                                (6)
```

(1) 应用对外暴露的一些自定义异常，命名规范：名词 + Exception
(2) 应用对外提供的基础工具类，命名规范：名词 + Util 举例：TimeUtils
(3) 按业务模块分多个子包，
(4) 此包下放某个业务模块对外提供的服务的定义接口。举例：ICargoPublishService
(5) 所有的传输层对象，且 DTO 对象的字段类型统一使用包装类型，命名规范：XxxDTO
(6) API 接口的请求入参类，命名规范：XxxRequest，入参对象内部可以包含 DTO

### 4.1.2 app 模块的分包标准如下

```
com.ymm.xx
  ├─common
  │  ├─ constant                              (1)
  │  ├─ exception                             (2)
  │  └─ util                                  (3)
  └─ [module name]
     ├─ config                                (4)
     ├─ controller                            (5)
     │  ├─ converter                          (6)
     │  ├─ request                            (7)
     ├─ page                                  (8)
     ├─ component                             (9)
     ├─ field                                 (10)
     ├─ rpc                                   (11)
     ├─ job                                   (12)
```

```
    ├─ mq                                   (13)
    ├─ model                                (14)
    │  └─ converter                         (15)
    ├─ service                              (16)
    │  └─ impl                              (17)
    ├─ repo                                 (18)
    │  ├─ impl                              (19)
    │  ├─ request                           (20)
    └─ dao                                  (21)
       ├─ entity                            (22)
       ├─ mapper                            (23)
       └─ xxx                               (24)
```

（1）常量类和枚举类型

（2）内部自定义抛出的异常类

（3）应用内部定义的工具类

（4）所有的配置类。命名规范：名词 + Config

（5）某个业务域对外提供的所有 REST 类，命名规范：XxxController

（6）所有的 REST 方法的 request 对象、页面模型与 model 模型的转换类

（7） REST 方法的请求参数，命名规范：XxxRequest

（8）面向客户端的页面对象，页面对象里面包含组件对象与表单类对象；

（9）页面内的展示类组件对象，命名规范：XxxComponent

（10）页面内的表单类组件对象，命名规范：XxxField

（11）对外提供 RPC 接口的实现类，主要做参数校验，返回值处理等，无业务逻辑，里面直接调用 service

（12） JOB 入口的 Handler，主要做参数校验，返回值处理等，无业务逻辑，命名规范：XxxJobHandler

（13） MQ 的 Listener，主要做参数校验，返回值处理等，无业务逻辑，业务逻辑写在 service 里

（14）应用内部定义的数据模型

（15）应用内部定义的数据模型与数据库 entity 对象、repo 定义的 rquest 对象

（16）同一个业务模块下面所有 service 接口定义，用来实现具体的业务逻辑。

（17）所有的 service 的接口实现类

（18）所有 repo 层的接口定义，用于对外部服务请求进行封装与防腐处理。

（19） repo 层的接口的实现

（20） repo 接口的请求入参对象（非必选，repo 接口的入参也可以是 model 对象）

（21）所有持久化数据接口的定义。对 dao 层的调用需要经过 repo 层的封装，不可以由 service 直接调用 dao 层

（22）数据库持久层对象，命名：XxxxEntity

（23）所有数据库 DAO 层接口的定义，命名规范：XxxMapper；一个 DAO 接口的定义对应一张表或者一组相关的表

（24）xxx 表示不同的存储介质，比如 Elasticsearch、HBase、Cache；此包下放置与这些存储交互的工具实现类

# 5. 构建标准

## 5.1 通用规范

### 5.1.1 源码级别依赖，必须设置 optional = true

源码级别的依赖，只在源码编译的时候有效，大多数都是 APT（Annotation Processor Tool）的包，这类依赖，对运行时没有任何作用。`<optional>true</optional>` 表示，依赖不传递；`<scope>provided</scope>` 表示，使用方会提供此依赖，也有类似的效果，但寓意不太一样。`<scope>provided</scope>` 用到最多的场景，比如你的系统需要 `servlet-api`，最终要在 Tomcat 容器跑，那应该用这个配置。

【示例】lombok 依赖：

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.10</version>
    <optional>true</optional>
    <scope>provided</scope>
</dependency>
```

### 5.1.2 生产环境依赖必须是正式版本，禁止引用 SNAPSHOT 版本

`SNAPSHOT` 版本随时可能更新，会带来很多稳定性问题，切忌生产使用 `SNAPSHOT` 版本依赖。

## 5.2 打包规范

### 5.2.1 API 包规范

xxx-api 只包含接口定义，不包含实现，按最小化依赖原则，应避免引入二方、三方包，降低使用方依赖冲突。

#### 5.2.1.2 模块独立打包

*API* 模块需要独立打包，接口定义、出入参和异常，都需要统一放在此模块中定义。如果需要公共的返回值包装类或者公共异常类，可以引用公司公共 ***commons-api*** 依赖。如果能纯粹到，只需要依赖 *JDK* 是最好的。

#### 5.2.1.2 不依赖日志

API 只是协议定义，不包含业务逻辑，不要依赖日志组件。

#### 5.2.1.3 校验框架 *validation-api*，建议设置 *optional = true*

`validation-api` 都是基于注解的，不传递也不会导致使用方出错，但如果是自定义的 `validation` 注解，必须标记为 optional = true。

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
    <optional>true</optional>
</dependency>
```

#### 5.2.1.4 不要依赖中间件

不要决定使用方使用中间件的版本；如果确实需要，则设置 `scope = provided` ，自己在API实现里面，应该手动设置中间件的依赖，不能通过二方包的中间件依赖来传递，因为 `provided` ，是让使用方来提供依赖

### 5.2.2 富客户端 SDK 规范

#### 5.2.2.1 扩展实现不应该被全部打进依赖中

全部扩展应该全部独立出来，供使用方选择哪一种扩展；一个框架内部可能会有很多扩展，每一种扩展又有多种实现。那么不需要把所有的扩展实现打进依赖中，而应该作为可选依赖，让用户自己选择。如 `Dubbo` 就将注册中心的扩展单独打成 `jar` 包，默认内置 `Zookeeper` 注册中心。

#### 5.2.2.2 如果需要依赖日志组件，必须依赖日志抽象，而不是日志实现

只能依赖 `slf4j-api` ，不能直接依赖 `log4j2` 、 `logback` 等；使用方系统，一般都会自己选择日志的实现，不要为使用方做决定。

### 5.2.3 应用必选组件

| 名称 | 用途 |
| --- | --- |
| Lion | 配置中心 |
| Hubble | 监控平台 |
| YmmLog | 日志采集 |
| Thaad | 敏感信息加解密 |
| HealthCheck | 健康检查 |

## 5.3 版本规范

### 5.3.1 开发环境版本号定义规范

`DEV` 环境的版本号，统一定义为： `1.0.0-dev-SNAPSHOT` 。

为了避免 `DEV` 环境不同分支之间的多种小版本之间有冲突，dev 分支上的版本号需要固定为上述的统一定义。规避了多个需求并行开发时的打包冲突问题。

### 5.3.2 正式版本号定义规范

版本号需要遵循标准的约定： **<主版本>.<次版本>.<增量版本>**

主版本和次版本之间，以及次版本和增量版本之间用点号分隔。

> 1. **主版本：**表示项目的重大架构变更，进行不向下兼容的修改时，递增主版本号；
> 2. **次版本：**表示较大范围功能的增加和变化，以及修复 Bug，API 保持向下兼容的新增及修改时，递增次版本号；
> 3. **增量版本：**表示已有功能 Bug 的修复，修复问题但不影响 API 时，递增增量版本。

## 5.4 自动化测试构建规范

### 5.4.1 应用接入FTAPI

每个应用服务，都需要**接入 FTAPI**（契约管理平台）

> 自动化测试平台会基于 FTAPI 的扫描出的 API 结果做自动化测试，在应用构建发布完成后，会触发自动化平台进行自动化测试并生成对应的测试报告。对于新增的 API 测试用例的覆盖度，也会被扫描出来。

### 5.4.2 自动化测试覆盖范围规范

#### 5.4.2.1 核心模块配置

主链路的应用，都需要配置核心模块（核心业务的独立可复用模块）。

> 自动化测试平台支持模块的配置，每个独立的模块，可以由多个 API 串行化执行组成。

#### 5.4.2.2 多环境的覆盖范围规范

1. `DEV` 环境需要跑**核心模块级别**的测试用例。
2. `QA` 环境需要跑**应用全量级别**的测试用例。
3. 生产环境需要跑影响**稳定性**的相关测试用例。

### 5.4.3 自动化测试发布准入规范

1. 核心链路的应用，如果线上环境自动化测试通过率低于 100%，则阻断发布；
2. 自动化测试结果中，出现零容忍的异常，则阻断发布。

## 5.5 构建发布规范

### 5.5.1 QA环境发布版本规范

`QA` 环境只能选择 **`master`** 、 **`release`** 、 **`qa`** 开头的分支打包，其余开发分支、临时分支等禁止在 `QA` 环境打包发布。

### 5.5.2 生产环境发布版本规范

生产环境发布的版本，必须使用 `QA` 环境部署过的程序包发布，不可单独使用代码分支重新打包再发布。

# 6. 日志标准

日志输出对问题的快速定位至关重要，但日志量过多也会带来成本的增加，因此需要在正确的地方输出合适的日志。

## 6.1 日志框架

统一使用 `slf4j + log4j2` 组合，代码层只能使用 `slf4j` 提供的接口，不允许直接使用 `log4j2` 的接口。

### 6.1.1 Maven 坐标

公司已经统一包装了日志组件：

```xml
<dependency>
  <groupId>com.ymm.common-plugins</groupId>
  <artifactId>common-logger-log4j2</artifactId>
  <version>1.2.2</version>
</dependency>

<dependency>
  <groupId>com.ymm.common-plugins</groupId>
  <artifactId>common-logger</artifactId>
  <version>1.2.2</version>
</dependency>

<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j-impl</artifactId>
  <version>2.11.1</version>
</dependency>

<!-- Spring Boot 工程 需要排除 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions><!-- 去掉springboot默认配置 -->
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency> <!-- 引入log4j2依赖 -->
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

### 6.1.2 log4j2 配置

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="off" monitorInterval="1800"
 packages="com.ymm.common.log4j2">
  <Appenders>
    <YmmConsole name="ymmConsole" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-5p [%c] [%t] %m%n" />
    </YmmConsole>

    <YmmFile name="ymmlog" />
    <CatAppender name="cat" />
  </Appenders>

  <Loggers>
    <root level="info" includeLocation="true">
      <appender-ref ref="ymmConsole"/>
      <appender-ref ref="ymmlog"/>
      <appender-ref ref="cat"/>
    </root>
  </Loggers>
</Configuration>
```

## 6.2 日志级别

常见的日志级别有5种，分别是 `error` 、 `warn` 、 `info` 、 `debug` 、 `trace` 。日常开发中，我们需要选择恰当的日志级别

1. `error`
    a. 错误日志，指比较严重的错误，对正常业务有影响，需要运维配置监控的；
    b. 业务异常
    c. 系统异常
    d. 用户输入异常
2. `warn`
    a. 警告日志，一般的错误，对业务影响不大，但是需要开发关注；
    b. 弱依赖异常
3. `info`
    a. 信息日志，记录排查问题的关键信息，如调用时间、出参入参等等；
    b. 分支逻辑
4. `debug` 用于开发调试的，关键逻辑里面的运行时数据， `线上禁止使用` ；

5. `trace` 最详细的信息，一般这些信息只记录到日志文件中，`线上禁止使用` 。

## 6.3 日志输出

### 6.3.1 占位符输出

【正例】

```
public void sayHello(String name) {
    log.info("Hello, {}.", name);
    // 业务代码
}
```

【反例】

```
public void sayHello(String name) {
    log.info("Hello, " + name + ".");
}
```

### 6.3.2 不允许提前串化对象

```
@Getter
@Setter
public class Name {
    private String realName;
    private String nickname;

      @Override
    public String toString() {
        return JSON.toJSONString(this);
    }
}
```

如果提前序列化要打印的对象，导致不需要输出的日志，也会执行序列化逻辑，浪费计算。

【正例】

```
public void sayHello(Name name) {
  log.debug("Hello, {}.", name);
  // 业务代码
}
```

【反例】

```
public void sayHello(Name name) {
  log.debug("Hello, {}.", JSON.toJSONString(name));
  // 业务代码
}
```

线上不输出 `debug` 日志，但因为提前串化，还是会走序列化逻辑。

### 6.3.3 不允许直接 `e.printStackTrace()`

【正例】

```
public void sayHello(Name name) {
    try {
        // 业务逻辑代码
    } catch(Exception e) {
        log.error("xxx 出来失败。", e);
    }
}
```

【反例】

```
public void sayHello(Name name) {
    try {
        // 业务逻辑代码
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

### 6.3.4 不能只打印 `e.getMessage()`

【正例】打印整个堆栈信息

```
public void sayHello(Name name) {
    try {
        // 业务逻辑代码
    } catch(Exception e) {
        log.error("xxx 出来失败。", e);
    }
}
```

【反例】

```java
public void sayHello(Name name) {
    try {
        // 业务逻辑代码
    } catch(Exception e) {
        log.error("xxx 出来失败: {}。", e.getMessage());
    }
}
```

### 6.4 日志输出时机

1. REST 日志，网关已经打印好出入参数，无需自行打印
2. RPC 日志，微服务已经打印好日志，无需自行打印
3. 读链路，非必要不打日志
4. 写链路，方法需要打印出入参数
5. 分支逻辑判断，进入分支前，打印判断条件

## 7. 准入与发布标准

### 7.1 环境释义

#### 7.1.1 系统环境

| 环境 | 描述 |
| --- | --- |
| DEV | 开发环境及功能测试环境 |
| QA | 回归测试环境 |
| PROD | 生产环境，正式线上环境 |

#### 7.1.2 网络环境

| 环境 | 描述 |
| --- | --- |
| 金融 | 主要包含小贷、支付等金融相关的业务 |
| 非金融 | 主要包含用户、营销、交易、履约等不涉及金融合规的业务 |

## 7.2 云平台准入

### 7.2.1 适用范围

适用于产研交付的所有后端系统的云平台准入

### 7.2.2 应用准入申请规范

1. 只能申请创建 Spring Boot 及 Standard JAR 应用
2. 应用必须配置应用英文名、中文名及应用描述
3. 应用必须定义服务等级，P1（核心应用，影响主流程）、P2（重要应用，不影响主流程）、P3（一般服务，不影响主流程）
4. 应用必须配置运行环境，例如：JDK 1.8
5. 应用必须配置所属业务线及业务域
6. 配置应用关联的 git 仓库地址以及 git 相对路径
7. 选择可以部署的系统环境及网络环境
8. 选择应用总负责人、稳定性负责人、研发负责人、测试负责人、研发成员及测试成员
9. 配置打包命令及需要部署的文件
10. 应用必须配置优雅发布、网关无损调度、安全扫描
11. 配置应用告警渠道

## 7.3 发布标准

### 7.3.1 适用范围

适用于产研交付的所有后端系统的发布

### 7.3.2 应用发布

#### 7.3.2.1 发布准入标准

1. 待发布应用无 Bug 遗留
2. 待发布应用已经接入灰度、Thaad 、Hubble 、HealthCheck 、YmmLog
3. 待发布代码合入到 Release 分支，在 QA 环境完成功能测试通过及历史功能回归测试通过
4. QA 环境，监控平台上无未知原因的异常（所有已知原因的异常都不是本次代码原因）
5. 如果本应用依赖其他应用，确保被依赖的应用已完成发布，或者同被依赖应用一起灰度发布
6. 所有待发布操作确保有回滚方案

#### 7.3.2.2 发布过程规范

1. 通报机制：发布前在钉钉群里通报，发布完成后在钉钉群通报完成
2. 应用发布前确认依赖的前置 DB 脚本已在生产环境执行成功
3. 备份线上配置
4. 发布到预发容器后，测试验证通过，进行下一步，否则取消发布
5. 所有后端发布需采用灰度发布，并在整个发布期间密切关注发布应用的的监控信息（零容忍告警、代码异常日志、接口性能、JVM 监控）
6. 预发环境验证无问题后，根据应用等级及业务情况自行决定开启多少灰度流量过早高峰，P1 应用建议灰度开启1% 经历第二天早高峰
7. 发布过程中（灰度 100% 之前），如有任何问题（不管是否和本次上线功能相关），立即灰度切

0，并第一时间同步到值班群，通知版本相关的开发、测试同学立即跟进

#### 7.3.2.3 灰度策略

当前，所有后端 P1 应用发布都要求采用灰度发布，如未接入灰度的应用需尽快接入。

**【核心应用】**发布当天切换 1% 流量到灰度容器，并到监控上观察灰度流量是否有异常（检查点包括：灰度分组是否有正确比例的流量进入，灰度分组是否有异常，灰度分组调用是否有明显性能问题，值班群及业务群是否有问题反馈），至少监控 30 分钟；发布次日，11:00 ~ 15:00 逐步完成所有线上流量切换（每隔 30 分钟进行下一步流量切换，并持续观察是否有异常）。

**【非核心应用】**发布当天按 5%， 10%， 20%， 50%， 70%， 100% 共 6个批次完成流量切换，每次间隔 30 分钟，并在监控上观察灰度分组是否有异常（检查点包括：灰度分组是否有正确比例的流量进入，灰度分组是否有异常， 灰度分组调用是否有明显性能问题，值班群及业务群是否有问题反馈），至少监控 30 分钟。

#### 7.3.2.4 发布窗口

| 周一 | 周二 | 周三 | 周四 | 周五 | 周六 | 周日 |
| --- | --- | --- | --- | --- | --- | --- |
| 服务端发布 | | 服务端发布 | | 服务端发布 | | |

1. 产研建议固定发布窗口，每周一、周三、周五业务低峰时间段发布。如果需求需要临时发布的，也可在其他日期发布，注意业务影响
2. 按照灰度发布策略，单次发布持续时间一般在 3 小时
3. 对于以项目运作的批量需求上线，建议项目经理在计划安排时预留发布时间，并在发布前一天由项目 PTM 组织完成项目上线方案
4. 对于涉及后端和 APP 都发版的需求，后端应用发布需给 APP 预留 TF 测试时间，APP 发布 TF 为 2 天，RN 发布的 TF 为 1 天。

### 7.3.3 二方库发布

#### 7.3.3.1 发布准入标准

1. 版本文档中描述本次版本变更的范围、新增的内容、修复的问题
2. 单元测试覆盖率100%，并且全部通过

#### 7.3.3.2 发布过程规范

1. dev 环境只允许发布 SNAPSHOT 版本，用于功能验证
2. 待发布代码 SNAPSHOT 版本号修改为正式版本号
3. 待发布代码已经合入 Release 分支，并且单元测试全部通过
4. 打包完成的正式 JAR 包进行 `To Maven` 操作

# 8. 安全规范

1. 所有系统，dev、qa 环境的服务默认仅在内网访问；
   a. 若开发/测试阶段需要开通公网访问，只能申请临时开通且临时开通最大时长不超过 15 天，到期后运维将及时关闭
   b. 若需要长期开通公网访问，需由 **N-1 级 TL** 以及 **CTO** 审批后运维方可执行；
2. 内部服务（通常主域名为 `amh-group.com` ），默认仅在内网访问；特殊需求找安全评估后方可申请开通公网访问
   a. 注：开通应遵循最小原则：最少时间需要（限期关闭）、最少使用方需要（来源白名单）

# 9. 中间件选型范围

| 中间件 | 可选组件 | 服务端版本 | 客户端产品 | 应用场景 |
| --- | --- | --- | --- | --- |

| | | | | |
|---|---|---|---|---|
| 关系型数据库 | MySQL | 5.7 | ironman-sql | 用于需要事务性 SQL 引擎的任何应用场景，如存储货源、订单、评价、建议反馈等跟业务需求直接相关的数据 |
| 全文检索引擎 | Elasticsearch | 7.5.x | ebatis | 全文检索、需要打分排序的列表、海量近实时数据分析 |
| K/V 数据库 | Redis | 6.0.x | ironman-redis | 内存数据结构存储系统，分布式缓存、分布式锁、频控等 |
| 消息中间件 | RocketMQ | 4.3.0 | ironman-mq | 低延迟、高性能和可靠性、亿级容量同时具备灵活的可伸缩性的分布式消息和流处理平台。异步、解耦、分布式事务、消息顺序收发、削峰填谷、数据同步缓冲 |
| | Kafka | 0.10.1、0.10.0.1、0.11.0.3、2.2.0、2.6.3 | | 高吞吐，日志采集，大数据计算等 |
| | 延迟队列 | 3.0 | | 自定义延迟时间点的消息队列（目前支持10s~7d的范围） |
| 文件存储 | OSS | | ymm-file | 图片存储，视频存储，及其他相关性文档的存储等 |

| | | | | |
|---|---|---|---|---|
| 去标化存储 | Thaad | | thaad | 用户敏感信息统一加密/解密，适用于系统中有存储敏感的明文信息、有敏感的明文信息流过、有明文数据吐出展示场景 |
| 任务调度平台 | YmmJob | | YmmJob | 实现任务调度与任务执行解耦，支持任务服务自动注册、调度策略可配置、JOB 灰度、泳道支持、监控告警 |
| 监控告警 | Hubble | | cat | 提供监控告警、链路追踪、日志搜索等功能的观测平台 |
| 流控降级 | Sentinel | | sentinel | 以服务稳定性为目标的中间件，以流量为切入点，从限流、熔断降级、系统负载保护等多个维度来保障微服务的稳定性. |
| 网关 | Hango | | hango | 基于 `spring-webflux` ，支持注册发现、请求转发、请求 `Mock` 、加解密、统一鉴权、流量控制、路由配置、流量调度、流量记录、IP 黑白名单 |
| 配置中心 | IronMan Config | | ironman-config | 配置中心集中化管理应用不同环境的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、配置灰度、配置发布 / 回滚、状态跟踪等特性 |

| | | | | |
|---|---|---|---|---|
| 远程调用 | IronMan RPC | | ironman-rpc | 面向接口的远程方法调用，具备服务发现、服务调用、服务注册注销、监控分析、服务限流、服务心跳/重连、服务单机测试、服务降级、负载均衡、服务容错等功能 |
| 分布式锁 | IronMan Lock | | ironman-lock | 在分布式场景下需要对资源进行控制，目前支持 `Redis` （CP）和 `Zookeeper` （AP）实现方式 |
| 缓存 | IronMan Cache | | ironman-cache | 需要快速实现本地缓存、分布式缓存、多级缓存等 |
| 幂等 | IronMan Idempotent | | ironman-idempotent | 业务方需要为业务逻辑确定一个幂等 Key，然后借助存储的去重能力，保证相同的幂等 Key 所代表的业务逻辑只会执行一次。 |
| 分布式事务 | IronMan Transaction | | ironman-transaction | 需要解决在分布式环境下，事务的原子性，一致性，隔离性和持久性 |
| 发号器 | Number-Generating | | number-generating | 分布式系统下，需要全局唯一 ID |

| | | | | |
|---|---|---|---|---|
| 冷备系统 | Data Prison | | data-prison | 用户注销时，用户中心会发送用户注销的 MQ 消息各业务系统如果涉及到用户的敏感数据，需要监听用户注销的消息，把敏感数据匿名化同时把可以把敏感数据统一写入到冷备系统 |
| 刷数平台 | Archimedes | -- | archimedes | 统一规范和管理批量更新线上数据的场景 |

# 10. 附录：研发词汇表

| 中文 | 英文 | 业务释义 |
|---|---|---|
| 货主 | shipper | 发货人，托运人，生成货源的运输需求的主体 |
| 司机 | driver | 承运人，运输人，接收货主运输需求，进行运输过程的主体 |
| 货源 | cargo | 货主发出的运输诉求，包含线路、车辆需求、货品类型、费用等信息。 |
| 运单 | waybill | 司机承接货主运输诉求后，基于平台规则，生成的承接关系，包含承接人、托运人，对应货源，以及交易信息（如优惠信息、支付信息等）。 |
| 运费 | freight | 运输需求，关联的交易信息中，货主应支付给司机的报酬。 |
| 小黑板 | blackboard | 货主打开app的首页，包含货主发货中货源列表，各业务发货入口等功能。（源自线下信息部通常使用黑板发布货源信息）。 |

| | | |
|---|---|---|
| 元数据 | metadata | 描述数据的属性信息 |
| 定金 / 保证金 | deposit | 司机承接货主运输诉求时，按照目前业界风俗，需要司机先向货主支付一定金额的费用，确定承运意向，支付定金后，货源下架，避免了其他司机同时接单。 |
| 佣金 | commission | 平台收取的服务费，是在一笔交易达成时，向司机收取的费用，作为平台的盈利方式（个别业务出现过向货主收取佣金的实验性产品方案）。 |
| 退款 | refund | 交易过程中，出现变动时，需要收款方向付款方回退资金。 |
| 担保 | guarantee | 担保交易，在交易过程中，为了保障交易中支付费用方的安全，由平台进行费用托管，双方确认交易无误时，才将资金转入收款方账户。 |
| 线路 | route | 运输线路，指运输需要从A地到B地。如：南京到北京（当前平台支持的运输线路约10W+）。 |
| 回单 | receipt | 司机将货物运达后，需要收货方提供的收据凭证。部分货主需要查验该凭证后进行运输尾款的支付。 |
| 发票 | invoice | 发票就是发生的成本的原始凭证。对于公司来讲，发票主要是公司做账的依据，和缴税的费用凭证；当前平台包含普票业务与专票业务两类。 |
| 协议 | notary | 即运输协议，当前在成交环节完成协议的签署，成交后交易的变更需要一方发起协议变更，然后另一方同意才生效。 |

| | | |
|---|---|---|
| 车辆 | vehicle | 司机运输工具，平台当前使用了车长与车型两个属性。 |
| 预付 | pre-pay | 交易发生后，司机装货完成后，货主可以向司机提前支付部分运输费用，该提前支付的行为，称为预付。 |
| 到付 | arrival-pay | 交易发生后，司机拉货到达目的地后，货主向司机提前支付的运输费用，该运达时支付的行为，称为到付。 |
| 回单付 | receipt-pay | 交易发生后，司机向货主提供回单凭证后，货主向司机支付的运输费用，该收到回单时支付的行为，称为回单付。 |
| 途径点 | waypoint | 实际运输过程中，存在多装多卸的场景，将所有装货地、卸货地放在一起，统称为途径点。 |
| 一口价 | buyout | 平台主要产品，货主发货时可选“电议模式丨一口价模式”；一口价模式时，司机不能电话联系，认可运费时直接进行线上抢单。该模式提升了平台的成交效率，减少了司机货主线下不必要的低效沟通。 |
| 支付 | pay | 交易过程中，一方向另一方付款的行为。 |
| 结算 | settle | 担保交易时，支付方资金进入平台担保账户，结算表示资金从平台担保账户转入支付收款方的行为。 |
| 代付 | payroll | 交易中，非支付方代替支付方进行付款的行为。 |
| 代收 | collect-fee | 交易中，非收款方代替收款方进行收款的行为。 |
| 还款 | repay | 欠款方，基于欠款进行支付的行为。 |

| | | |
|---|---|---|
| 分账 | profit-share | 交易中，收款方涉及多方，将资金转入多个收款方的行为称为分账。 |
| 开始/起点 | start | |
| 结束/终点 | end | |
| 装货 | load | |
| 卸货 | unload | |
| 确认装货 | confirm-loaded | 线下运输过程中，货主进行装货确认的行为。 |
| 确认卸货 | confirm-arrival | 线下运输过程中，货主进行卸货确认的行为。 |
| 确认回单 | confirm-receipt | 线下运输过程中，货主进行回单确认的行为。 |
| 履约凭证 | proof | 线下运输过程中，司机记录的装卸货照片与回单等可以证明履约行为发生的凭证信息。 |
| 地图 | map | |
| 导航 | navigation | 从一个地址，到达另一个地址的形式线路信息。平台内司机侧导航，会额外提供结合司机车型的、车牌信息等专用的货车导航模式。 |
| 评价 | comment | 交易完成后，司机货主双方可以针对此次交易进行相互评价。 |

| | | |
|---|---|---|
| 积分 | point | 用户积分，可在平台积分商城进行购买。 |
| 优惠券 | coupon | 交易发生时，平台给予的折扣/减免等享受优惠的资产。 |
| 活动 | activity | 为了达成营销目的，推出的各类拉新、促活行为。 |
| 充值 | recharge | 用户将资金转入平台钱包的行为。 |
| 奖品 | award | 营销活动，最终给予用户的奖励 |
| 抽奖 | raffle | 营销活动的一种，采用随机的方式获取奖励 |