

iFM 小程序项目规范

前端规范

本项目前端架构


└─api	接口接口目录
└─asses	css、icon资源目录
└─components	符合vue组件规范的uni-app组件目录
└─comp-comp.vue	可复用的a组件
└─config	公共配置文件比如请求域名、端口、缓存等。
└─pages	业务页面文件存放的目录
└─index	
└─index.vue	index页面
└─my	
└─my.vue	my页面
└─types	// 类型文件夹
└─userManage	// 用户管理
└─index.ts	// 类型文件
└─menuManage	// 菜单管理
└─index.ts	// 类型文件
└─...	
└─static	存放应用引用的本地静态资源（如图片、视频等）的目录，注意：静态资源只能存放于此
└─store	Pinia/vuex状态管理目录
└─utils	自定义工具类
└─uni_modules	存放uni_module规范的插件。
└─main.ts	vue初始化入口文件
└─App.vue	应用配置，用来配置App全局样式以及监听 应用生命周期
└─manifest.json	配置应用名称、appid、logo、版本等打包信息
└─pages.json	配置页面路由、导航条、选项卡等页面类信息

TypeScript 规范

类型或接口定义类型或接口定义

- 定义类型或接口时，请给出类型名（接口名）注释和各字段注释。类型名以 T 开头，即 type 的首字母；接口名以 I 开头，即 interface 的首字母。

type

```
//  good
/* 登录表单 */
type TLoginForm = {
  // 用户名
  username: string
  // 密码
  password: string
  // 验证码
  code: string
}
```

```
// or

// ✅ good
/* 登录表单 */
type TLoginForm = {
  // 用户名 | 密码 | 验证码
  [key in 'username' | 'password' | 'code']: string
}
```

interface

```
// ✅ good
/* 登录表单 */
interface ILoginForm {
  // 用户名
  username: string
  // 密码
  password: string
  // 验证码
  code: string
}
```

vue规范

项目命名

项目名使用中横线 (-) 作为分隔符。

原因：

根据 HTML 和 XML 规范，中横线 (-) 被定义为有效的标识符字符。而下划线 (_) 则不是一个有效的标识符字符。

常用搭配词

handle: `handleClick`、`handleDelete` 表达的都是执行一个事件处理函数，用于按钮点击时事件处理。

原因：

约定俗成的命名方法，有助于团队内部协作和提高代码的可维护性。

全局方法

- 全局方法统一以 \$ 符号开头。例如 `$add`。

原因：

方便区分全局方法与页面内定义的方法。

【Vue3 + TypeScript】

```
// @/utils/index.ts
/**
 * 求两数之和
 * @param {number} num1 第一个数
 * @param {number} num2 第二个数
```

```

* @returns {number} 相加结果
*/
export function add(num1: number, num2: number) {
  return num1 + num2
}

// @/main.ts
import { add } from '@/utils/index.ts'

app.config.globalProperties.$add = add

```

公共组件

公共组件统一放到 `@/components` 目录下，且应该给各个组件创建一个独立的文件夹（文件夹采用大驼峰命名，例如 `SearchBox`），文件夹下创建 `SearchBox.vue` 作为组件入口，组件入口需与文件夹同名；

```

my-project
├─ src
│   └─ components          // 公共组件文件夹
│       └─ SearchBox       // 搜索组件
│           └─ SearchBox.vue // 组件入口
│       └─ ...
└─ ...

```

API 接口

所有 API 接口文件统一放到 `@/apis` 目录下，根据各模块不同进行分组放置。

```

my-project
├─ src
│   └─ apis          // API 接口文件夹
│       └─ userManager // 用户管理
│           └─ index.ts // 接口文件
│       └─ menuManage // 菜单管理
│           └─ index.ts // 接口文件
│       └─ ...
└─ ...

```

TypeScript 类型或接口定义

类型或接口定义文件统一放在 `@/types` 目录下，且文件以 `模块名.type.ts` 的范式命名。

Vue3 + TypeScript】

```

my-project
├─ src
│   └─ types          // 类型文件夹
│       └─ userManager.type.ts // 【用户管理】类型文件
│       └─ menuManage.type.ts // 【菜单管理】类型文件
│       └─ ...
└─ ...

```

原因：

类型文件单独存放，避免与页面逻辑文件混放造成干扰开发者的情况。

后端规范

本项目web架构

```
| - iFM-server    // 项目根目录
  | - src
    | - main      // 业务逻辑
      // | - assembly // 基于maven assembly插件的服务化打包方案
      // | - bin     // 模块脚本(启动、停止、重启)
      // | - sbin    // 管理员角色使用的脚本(环境检查、系统检测等等)
      // | - assembly.xml // 配置文件
    | - java      // 源代码
      | - com
        | - pxx
          | - ifmserver
            | - result //用于统一返回格式的枚举类
            | - config // 配置文件POJO
            | - filter // 过滤器
            | - controller // 控制层(将请求通过URL匹配，分配到不同的接收器/方法进行处理，然后返回结果)
            | - utils // 工具
            | - service // 服务层接口
              | - impl // 服务层实现
            | - mapper/repository // 数据访问层，与数据库交互为service提供接口
            | - entity/domain // 实体对象
              | - dto // 持久层需要的实体对象(用于服务层与持久层之间的数据传输对象)
              | - vo // 视图层需要的实体对象(用于服务层与视图层之间的数据传输对象)
            | - *Application.java // 入口启动类
          | - resources // 资源
            | - static // 静态资源(html、css、js、图片等)
            | - templates // 视图模板(jsp、thymeleaf等)
            | - mapper // 存放数据访问层对应的XML配置
              | - *Mapper.xml
              | - ...
            | - application.properties // 是 Spring Boot 的标准配置文件，用于集中管理应用程序的配置属性,将配置信息与代码分离
          | - test // 测试源码
          | - java
            | - com
              | - sharkchili
                | - www
                  | - system
                    | - 根据具体情况按源码目录结构存放编写的测试用例
            | - target // 编译打包输出目录(自动生成，不需要创建)
            | - pom.xml // 该模块的POM文件
          | - sql // 项目需要的SQL脚本
          | - doc // 精简版的开发、运维手册
          | - .gitignore // 哪些文件不用传到版本管控工具中
          | - pom.xml // 工程总POM文件
```

一、命名风格

【强制】VO下的实体类名称后跟随VO, DTO下的实体类不做此要求

- 正例: PostVO / CommentVO

【强制】类名使用 UpperCamelCase 风格, 必须遵从驼峰形式, 但以下情形例外: DO / BO / DTO / VO / AO

- 正例: MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion
- 反例: macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

【强制】方法名、参数名、成员变量、局部变量都统一使用 lowerCamelCase 风格, 必须遵从 驼峰形式。

- 正例: localValue / getHttpMessage() / inputUserId

【强制】常量命名全部大写, 单词间用下划线隔开, 力求语义表达完整清楚, 不要嫌名字长。

- 正例: MAX_STOCK_COUNT 反例: MAX_COUNT

【强制】Model 类中布尔类型的变量, 都不要加 is, 否则部分框架解析会引起序列化错误。

- 反例: 定义为基本数据类型 Boolean isDeleted; 的属性, 它的方法也是 isDeleted(), RPC框架在反向解析的时候, “以为”对应的属性名称是 deleted, 导致属性获取不到, 进而抛出异常。

【强制】对于 Service 和 DAO 类, 基于 SOA 的理念, 暴露出来的服务一定是接口, 内部的实现类用 Impl 的后缀与接口区别。

- 正例: CacheManagerImpl 实现 CacheManager 接口。

【推荐】为了达到代码自解释的目标, 任何自定义编程元素在命名时, 使用尽量完整的单词组合来表达其意。

- 正例: 从远程仓库拉取代码的类命名为PullCodeFromRemoteRepository
- 反例: 变量int a;的随意命名方式。

【推荐】接口类中的方法和属性不要加任何修饰符号 (public 也不要加), 保持代码的简洁性, 并加上有效的 Javadoc 注释。尽量不要在接口里定义变量, 如果一定要定义变量, 肯定是与接口方法相关, 并且是整个应用的基础常量。

- 正例: 接口方法签名: void f(); 接口基础常量表示: String COMPANY = "alibaba";
- 反例: 接口方法定义: public abstract void f(); 说明: JDK8 中接口允许有默认实现, 那么这个 default 方法, 是对所有实现类都有价值的默认实现。

【参考】枚举类名建议带上 Enum 后缀, 枚举成员名称需要全大写, 单词间用下划线隔开。

- 说明: 枚举其实就是特殊的常量类, 且构造方法被默认强制是私有。
- 正例: 枚举名字为 ProcessStatusEnum 的成员名称: SUCCESS / UNKOWN_REASON。

【参考】各层命名规约:

- A) Service/DAO 层方法命名规约
- 1) 获取单个对象的方法用 get 做前缀。
- 2) 获取多个对象的方法用 list 做前缀。

- 3) 获取统计值的方法用 count 做前缀。
- 4) 插入的方法用 save/insert 做前缀。
- 5) 删除的方法用 remove/delete 做前缀。
- 6) 修改的方法用 update 做前缀。

统一接口返回数据的格式

项目开发中返回的是json格式的数据，也就是统一json数据返回格式，一般情况下返回数据的基本格式包含是否成功、响应状态码、返回的消息、以及返回的数据。格式如下

```
{
  "success": 布尔,           // 是否成功
  "code": 数字,              // 响应状态码
  "message": 字符串,         // 返回的消息
  "data": {}                 // 放置响应的数据
}
```

响应状态码约定

对应使用场景	状态码	状态码含义
一般情况	20000	成功
	20001	未知错误
创建新账号/用户修改信息	50001	昵称已被使用
	50002	邮箱已被使用
用户信息查询/登录	60000	账号用户不存在
	60001	密码不匹配
	60002	未登录
Token安全令牌验证	80000	Token安全令牌验证失效,请重新登录

数据库规范

设计规范

1. 字段允许适当冗余，以提高查询性能，但必须考虑数据一致。冗余字段应遵循：

- 不是频繁修改的字段。
- 不是 varchar 超长字段，更不能是 text 字段。

举例说明：

假设有一个在线商店的数据库，其中有两个主要的表：**products**（商品表）和**categories**（商品类目表）。**products** 表中有字段 **product_name**（商品名称）、**category_id**（类目ID）等。**categories** 表中有字段 **category_id**（类目ID）、**category_name**（类目名称）等。在查询商品信息时，如果我们想要显示商品的类目名称，通常需要进行关联查询，如下：

```
SELECT products.product_name, categories.category_name
```

```
FROM products
```

```
JOIN categories ON products.category_id = categories.category_id;
```

如果类目名称（**category_name**）不经常改变，并且字段长度较短，那么可以考虑在 **products** 表中冗余一个 **category_name** 字段。这样，查询商品信息时就不需要进行关联查询，可以直接从 **products** 表中获取类目名称：

```
SELECT product_name, category_name
```

```
FROM products;
```

这样做的好处是提高了查询效率，因为避免了关联操作。但是，我们必须确保每当 **categories** 表中的 **category_name** 发生变化时，**products** 表中所有对应的 **category_name** 也必须同步更新，以保持数据的一致性。

2. id必须是主键，每个表必须有主键，且保持增长趋势的，小型系统可以依赖于 MySQL 的自增主键，大型系统或者需要分库分表时才使用内置的 ID 生成器
3. id类型没有特殊要求，必须使用bigint unsigned，禁止使用int，即使现在的数据量很小。id如果是数字类型的话，必须是8个字节。参见最后例子

- 方便对接外部系统，还有可能产生很多废数据
- 避免废弃数据对系统id的影响
- 未来分库分表，自动生成id，一般也是8个字节

4. 字段尽量设置为 NOT NULL，为字段提供默认值。如字符型的默认值为一个空字符值串";数值型默认值为数值 0;逻辑型的默认值为数值 0;
5. 每个字段和表必须提供清晰的注释
6. 时间统一格式:'YYYY-MM-DD HH:MM:SS'
7. 必须使用utf8mb4字符集

理解：在MySQL中的UTF-8并非才是真正的"UTF-8"，而utf8mb4才是真正的"UTF-8"。

8. 禁止使用外键，如果有外键完整性约束，需要应用程序控制

不得使用外键与级联，一切外键概念必须在应用层解决

9. 单表列数目必须小于30，若超过则应该考虑将表拆分

单表列数太多会使得MySQL处理InnoDB返回数据之间的映射成本太高。

命名规范

1. 表达是与否概念的字段，必须使用 is_xxx 的方式命名，数据类型是 unsigned tinyint (1表示是，0表示否)。

说明:任何字段如果为非负数，必须是 unsigned。

正例:表达逻辑删除的字段名 is_deleted，1 表示删除，0 表示未删除

2. 表名、字段名必须使用小写字母或数字，禁止出现数字开头，禁止两个下划线中间只出现数字。

正例:health_user, rdc_config, level3_name

反例:HealthUser, rdcConfig, level_3_name

3. 表名不使用复数名词。说明:表名应该仅仅表示表里面的实体内容, 不应该表示实体数量, 对应于 DO 类名也是单数形式, 符合表达习惯。
4. 禁用保留字, 如 desc、range、match、delayed 等, 请参考 MySQL 官方保留字。
5. 小数类型为 decimal, 禁止使用 float 和 double。

说明:float 和 double 在存储的时候, 存在精度损失的问题, 很可能在值的比较时, 得到不正确的结果。如果存储的数据范围超过 decimal 的范围, 建议将数据拆成整数和小数分开存储。

6. 表必备三字段:id,gmt_create,gmt_modified。

说明:其中id必为主键, 类型为unsigned bigint、单表时自增、步长为1。gmt_create, gmt_modified 的类型均为 date_time 类型, 前者现在时表示主动创建, 后者过去分词表示被动更新。

SQL创建表代码

```
CREATE TABLE `xxx` (  
  `gmt_create` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建  
时间',  
  `gmt_modified` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE  
CURRENT_TIMESTAMP COMMENT '上次更新时间',  
  `xxx_id` bigint unsigned NOT NULL AUTO_INCREMENT COMMENT '表id',  
  PRIMARY KEY (`xxx_id`)  
);
```

7. 所有命名必须使用全名, 有默认约定的除外, 如果超过 30 个字符, 使用缩写, 请尽量名字易懂简短

如 description --> desc;information --> info;address --> addr 等

8. 表的命名最好是加上“业务名称_表的作用”。

正例:health_user / trade_config

索引规范

1. 业务上具有唯一特性的字段, 即使是多个字段的组合, 也必须建成唯一索引。

不要以为唯一索引影响了 insert 速度, 这个速度损耗可以忽略, 但提高查找速度是明显的;另外, 即使在应用层做了非常完善的校验控制, 只要没有唯一索引, 根据墨菲定律, 必然有脏数据产生。

2. 页面搜索严禁左模糊或者全模糊, 如果需要请走搜索引擎来解决。

索引文件具有 B-Tree 的最左前缀匹配特性, 如果左边的值未确定, 那么无法使用此索引。

3. 超过三个表禁止 join。需要 join 的字段, 数据类型必须绝对一致;多表关联查询时, 保证被关联的字段需要索引

即使双表 join 也要注意表索引、SQL 性能。