

中台技术开发规范-【持续更新中】

技术选型

1. java语言
2. maven、git 版本控制
3. springcloud微服务体系 版本号待定
4. Mysql 8.0以上数据库
5. redis
6. RocketMQ

全局统一输出及异常处理

应用层对外接口处理全局异常处理，设置标准输入输出结构，核心服务返回不需要处理异常，具体见[全局统一输出异常处理方案](#)

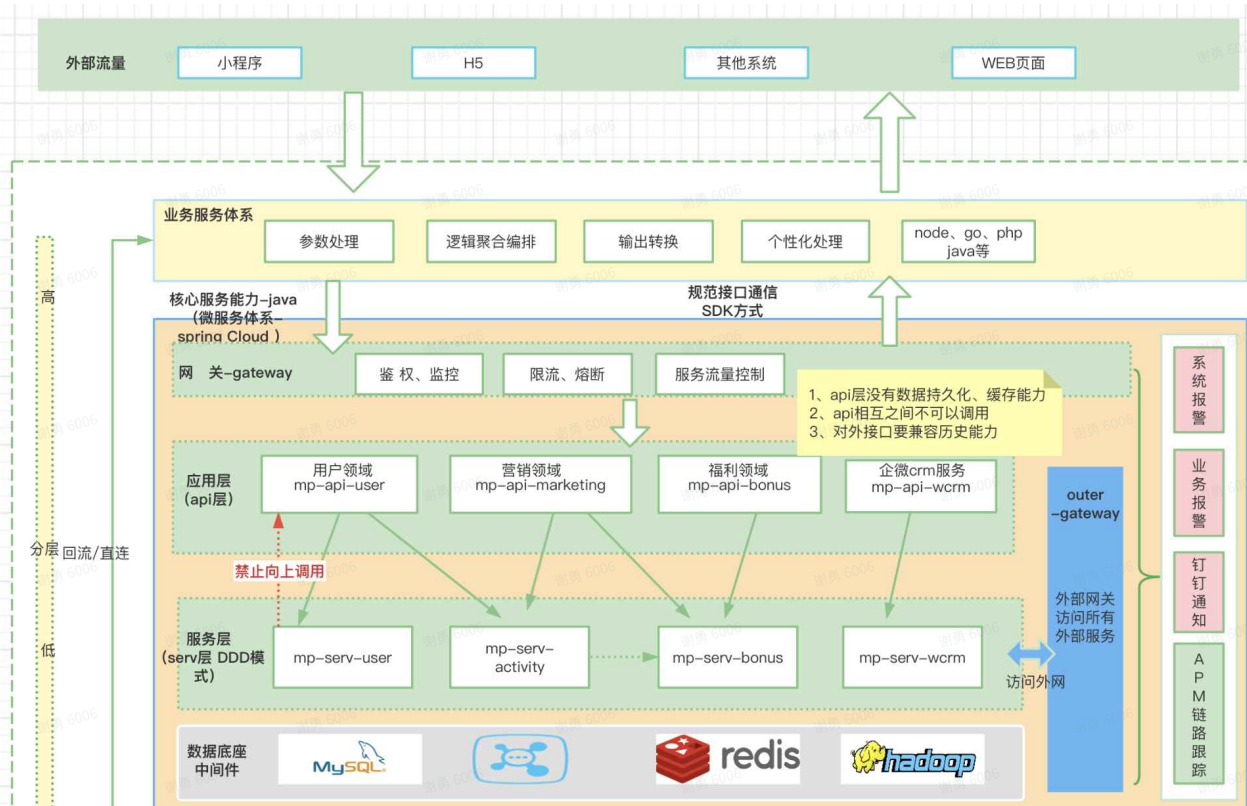
中台项目命名规范

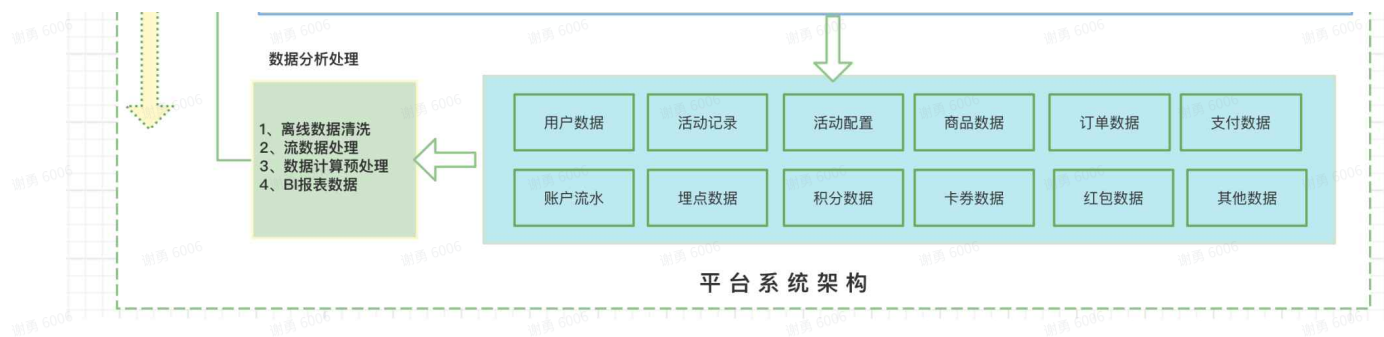
项目结构名称规范

项目统一用 mp-xxxx 开头

基础包路径为 mp.lylb.xxxx

[项目自动生成](#)





中台业务层项目命名格式：mp-api-xxx （xxx：业务领域标识）

中台服务层项目命名格式：mp-serv-xxx （xxx：业务服务标识）

中台管理体系命名格式：mp-admin-xxx

api接口名命名约定

对于API的接口规范我们借鉴restfull的一些约定，我们基本采用POST和GET的方式，允许出现部分动词来明确接口含义。

1. **URI结尾不应包含 (/)**
2. **正斜杠分隔符 (/) 必须用来指示层级关系**
3. **应使用连字符 (-) 来提高URI的可读性**
4. **不得在URI中使用下划线 (_)**
5. **URI路径中全都使用小写字母**
6. **URL带有版本号**

格式：

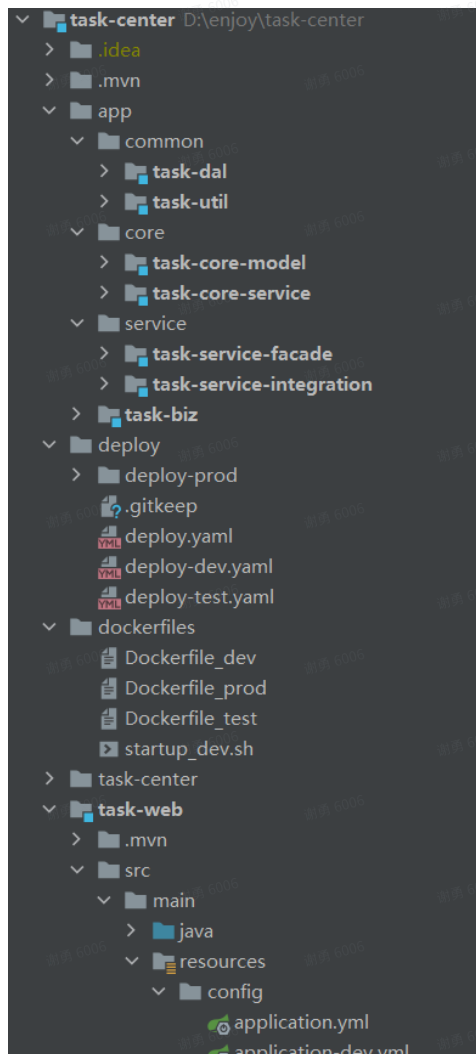
[http://mp-api-dev-util.01lb.vip/\\${project}/\\${version}/\\${resource}/dosomethine](http://mp-api-dev-util.01lb.vip/${project}/${version}/${resource}/dosomethine)

<http://mp-api-dev-util.01lb.vip/task-center/v1/task/get-ba-list>

项目结构

中台服务项目建立统一的父控类项目，所有项目继承此项目

一般项目的目录结构按以下



web:项目的web服务应用

dockfile: 容器文件

app/common/task-dal:数据库 缓存操作模块

app/common/task-util:工具类模块

app/core/model:核心服务的model类对象 转换等

app/core/service:核心服务模块

app/service/facade:对外暴露的api和对象模块

app/service/intergration:引用外部的服务 包括feign调用


app/biz:业务处理模块：处理服务的业务逻辑，实现对外API 已经调用核心服务提供业务支持

(注：biz模块最好全局定义日志切面，打印输入输出参数及耗时)

具体见 [中台系统DDD构思](#) 中技术实现部分

开发流程

根据需求，涉及到大的变化及复杂模块，新的服务项目开发，需要出技术方案并评审通过方可进入开发。

 在工作中遇到的一些问题而总结使用到一些原则，然后作为中台开发使用一些原则，这些原则不仅是可以让你避免很多踩坑，可以写高质量的代码，同时可以让你用架构的思维去思考一些问题，不仅适合中台开发，**会持续更新。。。**

常用原则总结

1. 分层设计相关原则

• 单向依赖原则

原则上只允许较高层级依赖较低层次，不允许反向依赖，同一层的依赖不限制，但是尽量减少同层依赖

在中台里，应用层依赖基础服务层，不可以反过来基础服务层依赖应用层，在基础服务层很难避免同层依赖，可能存在rpc调用，在业务层尽量减少同层依赖。

如果在业务中需要有底层依赖高层，我们可以有几种基本方式：

1. 系统依赖转为数据依赖
2. 接口依赖，通过底层定义SPI，业务层实现，这种做法其实是不得已为之，同时，我们在设计过程中还是尽可能避免走这条路。
3. 通过事件机制解耦依赖。

• 无循环依赖原则

系统设计方案时候，尽量减少系统之间的直接依赖，同时一定要避免系统之间出现循环依赖。

这是微服务场景下最容易、最大概率会遇到的一个问题，尤其是同层的领域系统之间的调用，导致系统容易出现循环调用，循环依赖带来的一个严重的问题是：**影响系统的发布和部署问题**

• 避免跨层调用原则

较高层次不允许直接跨层调用底层

软件设计中分层的一个重要目的是通过分层屏蔽底层实现的细节，如果出现跨层相当于把底层的实现直接暴露了，譬如我们的应用层，绕过我们领域服务，基础服务方法，直接调用DAO进行数据读写

操作（ps：不要惊讶这种方式，很多人默默的都在写，在应用层、聚合层直接引入数据库层操作对象），后果是：一旦需要重构修改DAO接口，或者数据接口发生变化，就发现升级造成成本巨大。

• 单一职责原则

大家对单一职责最熟悉的是类的职责定义问题，该职责是有罗伯.c.马丁于《敏捷软件开发：原则、模式和实践》一书中提出，规定一个类有且仅有一个变化引起它变化的原因，否则类应该被拆分（There should never be more than one reason for a class to change）

这个原则虽然提出时是解决类职责定义的问题，但实际上在对模块上划分也有指导意义，该原则虽然很简单，但是往往也容易被忽视。

eg：如果有一个功能可以放在系统的上游A系统中，也可以放在本身项目B中，那可以结合判断后续扩展等方面决定是否在A 还是B中，同时为了避免这个原则被突破，我们需要在另一个项目中去除所有相关的参数，这样后续功能新增天然就无法放到这个项目中。

• 数据冗余

设计方案和结构设计可以有数据冗余，但是应该使得系统中的数据冗余最小

譬如我们在实践过程中，接口设计时，在javadoc上强制制定接口必传参数，尽量做到最小集，减少上游系统使用接口的成本。另外在要求接口实现时，提前进行参数校验，不让不满足要求的数据冗余到系统中。

！还有一点就是缓存的冗余，我们在实际开发过程中经常为提高系统性能，在子系统/模块必要时需要对数据进行缓存，当发生变化时，必须要有相应的机制保证冗余的缓存数据的一致性和有效性。

2. 质量属性相关原则

• 数据安全

一般需要关注以下三方面的问题：

数据存储安全：敏感数据加密、日志输出脱敏

数据传输安全：包括加密、传输通道规范、最少字段传输（够用原则）

数据输出展示：前端展示需要防止水平越权，另外，前端的展示可以埋点和方便数据采集。

3. 资损防控

- 可核对和可监控：上下游系统的数据模型核对关联关系简单、稳定（具备通用性、和产品无关性）
- 可熔断：对关键资损链路需要做到可熔断。

这些在做架构和模型设计就要考虑，并且需要不定期反复review是否具备可核对和可监控。

4. 并发控制

- 悲观锁：代码编码过程中“一锁二判三更新”。
- 乐观锁：必须在事务内更新。
- 分布式锁：需要注意锁覆盖，释放不是自己加的锁对象问题。

5. 热点问题

避免流量倾斜，导致单台机器/单个数据表/数据库集中读写

这个需要在设计时充分提前预判业务的发展规模和系统的容量问题。在实际实施过程中，我们需要提前按照1~2年左右的业务规模来设计。

6. 数据倾斜

分库分表规则在设计时候需要考虑数据分布均匀，避免单库或者单表数据倾斜，在涉及到Hbase等rowkey设计时也需要考虑次问题。

7. 性能原则

可压测：对性能要求高的链路，需要做到可以压测。

8. 事务控制相关原则

- 优先使用编程式事务：为更好的控制事务，一般要求使用编程式事务，避免潜在的跨事务问题。
- 事务更新需要遵循的一致性：强一致要求还是最终一致，强一致是否涉及到跨库，事务操作时需要相同的记录更新的顺序保证一致。
- 事务中不进行远程调用：这样会把事务拉长，变成一个大事务，影响性能，尽量考虑最终一致性。

9. 一致性相关原则

- 区分系统调用错误和业务失败：远程调用失败，不代表下游系统没有接受请求，更不能作为业务失败依据，需要严格区分系统调用错误和业务失败。这一点目前在企微项目前期就特别严重，切勿模仿，按新的原则执行

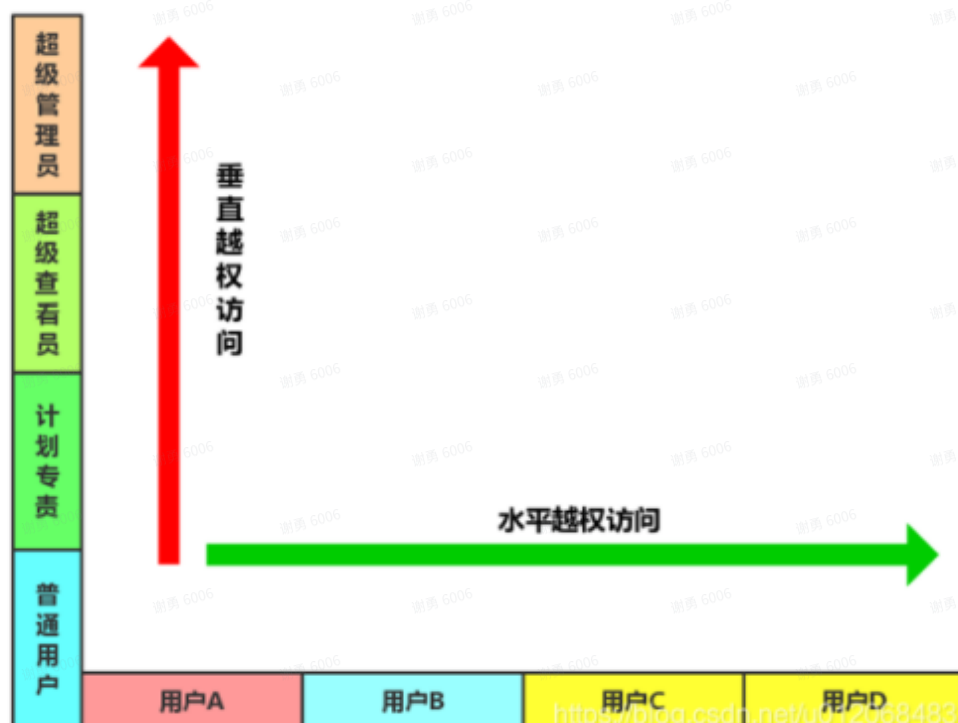
1. 可重试：任何一行代码执行时都有可能系统重启而中断，所以需要支持可重试，这个原则也很重要，目前都是分布式集群部署，最终一致的原则，优先就要考虑幂等等问题。
- 异步处理必须增加核对：最终一致性离不开恢复重试策略，也需要有系统间数据核对用于及时发现数据不一致，同时在核对时需要增加对处理时效的监控，及时返现长时间未处理成功的数据。

API设计相关设计原则

1. 水平越权控制

API设计时需要考虑防范水平越权。

水平越权访问是一种“基于数据的访问控制”设计缺陷引起的漏洞。由于服务器端在接收到请求数据进行操作时没有判断数据的所属人/所属部门而导致的越权数据访问漏洞。还有一种垂直越权访问也是同样的道理。



做法是：从前端到后端，每层需要进行越权校验，特别在应用层，通过从接口设计层防控，避免某一层出现疏忽导致越权的事件发生。

2. 接口幂等控制

调用方必须提供用于幂等控制的参数，为了控制幂等，同一个请求的幂等参数不变

现在都是微服务、集群负载、最终一致性等事务，接口支持幂等是一个必须要达成共识的点。接口设计、方案流程中支持幂等是必然的。

3. 兼容性原则

API升级和调整、需要兼容老的版本。

为了保证接口可以升级，接口设计就会需要较高的要求，接口中不适用枚举、不能是适用java基础类型，同时要求对接口设计具备一定的前瞻性和通用性，尤其对于面向业务领域设计的接口设计，要求对该领域的业务知识有比较多的了解。

! ps: 接口不使用枚举，使用string类替代，主要是会有反序列化问题，服务端升级了枚举类，要带着客户端也要一起升级，其次java基础类型定义，会有缺省值，无法判断是缺省值还是客户端设置的值，使用封装类型，默认是null，譬如 Integer Boolean等。

以上是总结的一些原则，在做设计和方案时遇到同类问题可以做出相对正确的选择，避免重蹈覆辙。另外，**通过在这些大的原则下进行具体化和明确化，能够让大家达成一致，减少沟通成本**，让方案更容易落地，不走偏

未完待续。。。