如此丝滑的API设计，用起来真香

> 分享是最有效的学习方式。
>
>
> 博客：[blog.ktdaddy.com/](http://cxyroad.com/”https://blog.ktdaddy.com/”)

故事

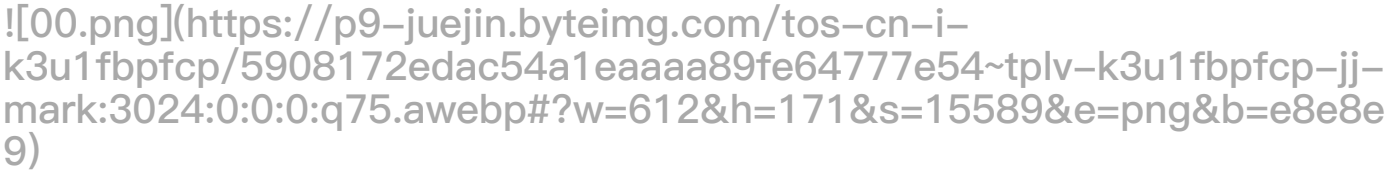
工位上，小猫一边撸着代码，一边吐槽着前人设计的接口。

如下：

“我艹，货架模型明明和商品SKU模型是一对多的关系，接口入参的时候偏偏要以最小粒度的SKU将重复入参进行平铺”。

“一个接口居然做了多件事情，传入参数复杂异常，不是一块业务类型的东西，非得全部揉在一起”。

“如此长的业务流程，接口能快起来么，难怪天天收到接口慢的告警”。



“这都啥啊，这名字怎么能这么取呢，这也太随意了吧....”

.....

小猫一边写着V2版本的新接口，一边骂着现状接口。

聊聊APi设计

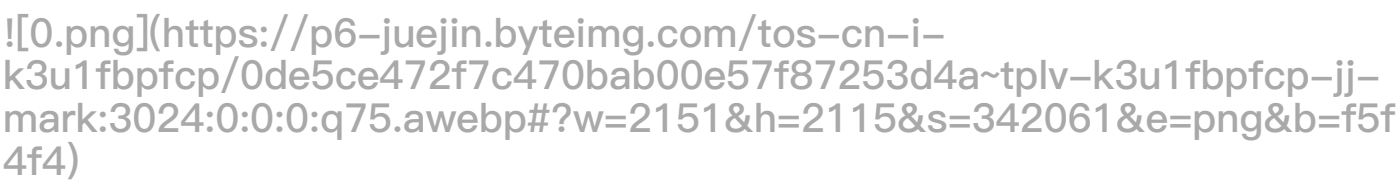
在日常开发过程中，相信大家在维护老代码的时候也多多少少会像小猫一样吐槽现有接口设计。很多项目经过历史沉淀以及业务验证，接口设计问题就慢慢放大暴露出来了。具体原因是这样的：

第一种情况可能是业务发展的必然趋势：不同技术人员对业务的看法和理解不同，一个接口可能经过多人的维护开发迭代，很多时候，新增功能也只是在原有的接口上直接拓展，当业务需求比较紧急的时候，大部分的研发一般都会选择快速去实现，而不会太过去考虑现有接口拓展的合规性。

第二种情况可能是本身开发人员自身能力问题，对业务的把控以及评估不合理导致的最终接口设计缺陷问题。

在系统软件开发过程中，一个好的UI设计可以让用户更好地使用一款产品。那么深入一层，一个好的API设计则可以让开发者高效地使用一个系统的能力，尤其是现在很多大型微服务项目中，API设计更加重要，因为此时的API调用方不仅仅是前端，甚至直接是其他服务。

那么接下来，老猫会和大家从下面的几个方面探讨一下，日常开发中我们应该如何去设计API。



API设计需要明确边界

在实际职场中，部门与部门之间、管理员与管理员之间容易出现扯皮、推诿现象。当然在系统和系统之间API的交互中其实往往也存在这样的情况。打个比方客户端的交互细节让后端代码通过接口来兜，你觉得合理不?

所以这就要求我们遵循下面两个点，咱们分别中两个维度来看，一个是“面向于服务和服务之间的API”，另一个是“面向客户端和服务之间的API”。

1、我们在设计API的过程中应该聚焦软件系统需要提供的服务或者能力。API是系统和外部交互的接口，至于外部如何使用，通过什么途径使用并不是重点。

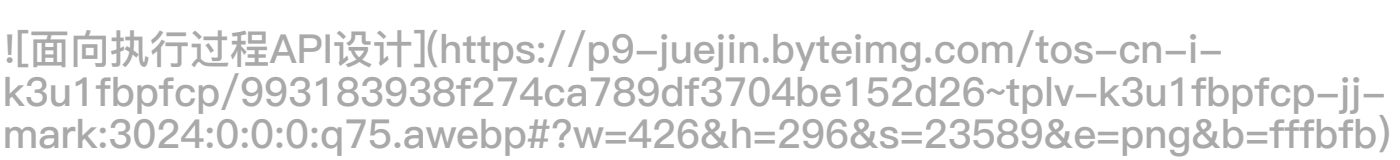
2、对于面向UI的API设计中，我们更应该避免去过多UI的交互细节。交互属于客户端范畴，不同的终端设备，其交互必然也是不一样的。

API设计思路尽量面向结果设计而不是面向过程设计

相信大家应该都知道面向对象编程和面向过程编程吧。

老猫虽说的这里的面向结果设计其实和面向对象的概念有点类似。这种情况下的API应该是根据对象的行为来封装具体的业务逻辑，调用方直接发起请求需要什么就能给出一个最终的结果性质的东西，而不是中间过程中某个状态性质的东西。上层业务无需多次调用底层接口进行组装才能获取最终结果。

如下图：



举个例子。

银行提现逻辑中，

如果面向执行过程设计的API应该是这样的，先查询出余额，然后再进行扣减。于是有了下面这样的伪代码。

```
public interface BankService {
  AccountInfo getAccountByUserName(String userName);
  void updateAccount(AccountInfoReq accountInfoReq);
}
```

如果是面向结果设计，那么应该就是这样的伪代码。

```
public interface BankService {
  AccountInfo withdraw(String userName,Long amount);
}
```

API设计需要尽量保证职责单一

在设计API的时候，应该尽力要求一个API只做一件事情，职责单一的API可以让API的外观更加稳定，没有歧义。并且上层调用层也是一目了然，简单易用。

对于一个API如果符合下面条件的时候，咱们就可以考虑对其进行拆分了。

1、一个API内部完成了多件事情。例如：一个API既可以发布新商品信息，又能更新商品的价格、标题、规格信息、库存等等。如果这些行为在一个接口进行调用，接口复杂度可想而知。
另外的接口的性能也是需要考虑的一部分，再者如果后续涉及权限粒度拆分，其实这种设计就不便于权限管控了。

2、一个API用于处理不同类型对象的业务。例如：一个API编辑不同的商品类型，由于不同类型的商品对应的模型通常是不同的（例如出行类的商品以及卡券类的商品差别就很大），
如果放在一个API中，API的输入和输出参数会非常复杂，使用和维护成本就很高。

其实关于API单一职责相关的话题，老猫在之前的文章中也有提及过，有兴趣的小伙伴可以戳【[忍不了，客户让我在一个接口里兼容多种业务功能](http://cxyroad.com/ "https://mp.weixin.qq.com/s/lnwY-nc6QIdZmfAsVLSuuw")】

API不应该基于实现去设计

在API设计过程中，我们应该避免实现细节。一个API有多种实现，在API层面不应该暴露实现细节，从而误导用户。

例如生成token是最为常见的，生成token的方式也会有很多种。可以通过各种算法生成token,
有的是根据用户信息的hash算法生成，或者也可以用base64生成，甚至雪花算法直接生成。如果对外暴露更多实现细节，其实内部实现的可拓展性就会相当差。
我们来看一下下面的代码。

```
//反例:暴露实现细节
public interface tokenService {
  TokenInfo generateHashTokenByUserName(String userName);
}
//正例:足够抽象、便于拓展
public interface tokenService {
  TokenInfo generateToken(Object key);
}
```

API的命名相当重要

一个好的API名字无疑是相当重要的，使用者一看API的命名就能知道如何使用，可以大大降低调用方的使用成本。所以我们在设计API的时候需要注意下面几个方面。

1、API的名字可以自解释，一个好的API的名称可以清晰准确概括出API本身提供的能力。

2、保持对称性。例如read/write,get/set。

3、基本的API的拼写务必准确。API一旦发布之后，只能增加新的API去订正，旧API完全没有请求量之后才能废弃，错误的API的拼写可能会带给调用方理解上的歧义。

API设计需要避免标志性质的参数

所谓标志性的参数，就是一个接口为了兼容不同的逻辑分支，增加参数让调用方去抉择。这块其实和上述提及的API设计保证职责单一有点重复，但是老猫觉得很重要，所以还是
单独领出来细说一下。举个例子，上述提及的发布商品，在发布商品中既有更新的原有商品信息的功能在，又有新增商品的功能在。于是就有了这样错误的设计，如下：

```
public class PublishProductReq {
  private String title;
  private String headPicUrl;
  private List<Sku> skuList;

  //是否为更新动作，isModify就是所说的标志性质的参数
  private Boolean isModify;
  .....
}
```

那么对应的原始的发布接口为：

```
//反例：内部入参通过isModify抉择区分不同的逻辑
public interface PublishService {
  PublishResult publishProduct(PublishProductReq req);
}
```

比较好的逻辑应将其区分开来,移除原来的isModify标志位：

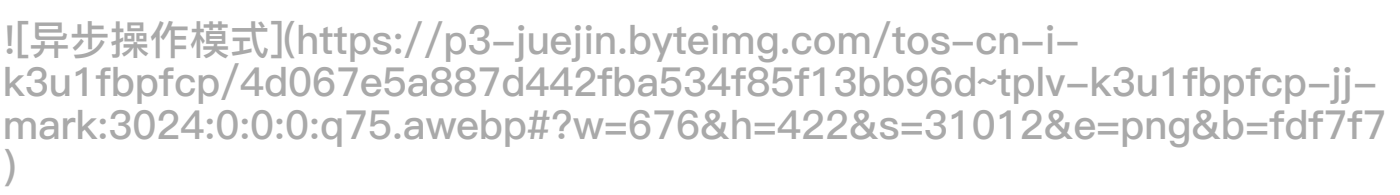
```
public interface PublishService {
  PublishResult addProduct(PublishProductReq req);
  PublishResult editProduct(PublishProductReq req);
}
```

```

### API设计出入参需要保证风格一致

这里所说的出入参的风格一致主要指的是字段的定义需要保持一个，例如对外的订单编号，一会叫做outerNo,一会叫做outerOrderNo。相关的用户在调用的时候八成是会骂娘的。

老猫最近其实在对接供应商的相关API，调用对方创建发货订单之后返回的订单编号是orderNo，后来用户侧完成订单需要通知供应商，入参是outerNo。老猫此时是懵逼的，都不知道这个
outerNo又是个什么，后来找到对面的研发沟通了一轮才知道原来outerNo就是之前返回的orderNo。

于是“我艹,坑笔啊”收尾.....

### API设计的时候考虑性能

最后再聊聊API性能，维护了很多的项目，发现很多小伙伴在设计接口的时候并不会考虑接口性能。或者说当时那么设计确实不会存在接口的性能问题，可是随着业务的增长，数据量的增长，
接口性能问题就暴露出来了。就像上面小猫吐槽的，接口又又又慢了，又在报接口慢警告了。

举个例子，查询API，当数据量少的情况下，一个List作为最终返回搞定没有问题的。但是随着时间的推移，数据量越来越大，List能够cover吗？显然是不行的，此时就要考虑是否需要通过分页去做。
所以原来的List的接口就必须要改造成分页接口。

当然关于API性能的优化提升，老猫整理了如下提升方式。

1、缓存：CRUD的读写性能毕竟是有限的。所以对某些数据进行频繁的读取，这时候，可以考虑将这些数据缓存起来，下次读取时，直接从缓存中读取，减少对数据库的访问，提升API性能。

2、索引优化：很多时候接口慢是由于数据库性能瓶颈，如果不用上述提及的缓存，那么我们就需要看一下接口究竟是慢在哪个环节，可能是某个查询，可能是更新，所以我们就要分析
执行的SQL情况去添加一些索引。当然这里涉及如何进行MYSQL索引优化的知
```

识点了，老猫在此不展开。

3、分页读取：如上述老猫举的例子中，针对的是那种随着数据量增长暴露出来的，那么我们就要对这些数据进行分页读取处理。

4、异步操作：在一个请求中开启多任务模式。



举个例子：订单支付中，支付是核心链路，支付后邮件通知是非核心链路，因此，可以把这些非核心链路的操作，改成异步实现，
这样就可以提升API的性能。常用的异步方式有：线程池，消息队列，事件总线等。当然自从Java8之后还有比较好用的CompletableFuture。

5、Json序列化：JSON可以将复杂的数据结构或对象转换为简单的字符串，以便在网络传输、存储或与其他程序交互时进行数据交换。
优化JSON序列化过程可以提高API性能。使用高效的序列化库，减少不必要的数据字段，以及采用更紧凑的数据格式，都可以减少响应体的大小，从而加快数据传输速度和解析时间。

6、其他提升性能方案：例如运维侧提升带宽以及网速等等

上述罗列了相关API性能提升的一些措施，如果大家还有其他不错的方法，也欢迎留言。

总结
--

谈及软件中的设计，无论是架构设计还是程序设计还是说API设计，
原则其实都差不多，要能够松耦合、易扩展、注意性能。遵循上述这些API的设计规则，
相信大家都能设计出比较丝滑的API。当然如果还有其他的API设计中的注意点也欢迎在评论区留言。

原文链接: https://juejin.cn/post/7369783680427409418