每个工程师都应该了解的：API 的设计和实现

2017-12-22 朱赟

每个工程师都应该了解的：API 的设计和实现
朱赟
- 00:00 / 11:19

在一个初创公司成长的过程中，作为工程师的你也许常常会遇到下面这样的情况。

有一天，你看到一个段代码或一个算法，觉得这些代码不大经得起推敲；于是你用 git blame 命令去寻找代码的主人；结果发现，原来作者是如今早就不写代码的 CTO 或 VP。

之后，在一个偶然的机会里，你和他讲起这件事，他会自豪地告诉你："哦，那时候我们必须在一天之内做出这个产品特性。当时也就我一个程序员吧，一天的时间，这是当时能做出最好的方案了。" 说完，他便陷入了对美好时光的怀念里。

你也可能听说过这样的故事。

有一天你的 CTO 突发奇想，行云流水地提交了一段代码；大家一看很激动啊，很多人跑去观摩大神的代码，结果觉得问题多多，于是在PR（ Pull Request ）上提了一堆评论。

CTO 一看有点傻眼了："几十条评论……现在代码要这么写啊，好麻烦。"于是他就和一位工程师说："你把评论里的问题解决下，合并（Merge）到主分支吧"， 然后就开开心心地该干嘛干嘛去了。

这两个小故事是想说明一个道理：一个公司早期的代码会因为各种历史原因不是那么完美，但是，在特定的时间点，这就是当时最优的方案。

随着公司的发展，成品功能不断叠加，代码架构不断优化，系统会经历一些从简到繁，然后再由繁到简的迭代过程，代码的改动也会相当巨大，也许有一天，你会几乎不认识自己当初的作品了。

API 的设计和实现更是如此。在我们的工作中，很少能见到 API 的设计和实现从最开始就完美无瑕疵。一套成熟的 API，很多时候都是需要通过不断演化迭代出来的。今天我就和你聊聊 API 的设计和实现。

首先第一点，我们先从 API 的签名（Signature）说起。

API 的签名（Signature）

API 的签名，或者叫协议，就是指 API 请求（Request） 和响应（ Response ）支持哪些格式和什么样的参数。

首先，做过 API 的人都知道，一个上线使用的 API 再想改它的签名，会因为兼容性的问题痛苦不堪。因此，API 签名的设计初期，一定要经过反复推敲，尽量避免上线后的改动。

除了一些基本的 RESTful 原则外，签名的定义很多时候是对业务逻辑的抽象过程。一个系统的业务逻辑可能错综复杂，因此 API 设计的时候，就应该做到用最简洁直观的格式去支持所有的需求。

这往往是 API 设计中相对立的两面，我们需要找到平衡。有时候为了支持某一个功能，似乎不得不增加一个很违反设计的接口；而有时候我们为了保证 API 绝对规范，又不得不放弃对某一些功能的直接支持，这些功能就只能通过迭代调用或客户端预处理的方式来实现。

这种设计上的取舍，通常会列出所有可行的方案，从简单的设计到繁杂的设计；然后通过分析各种使用实例的频率和使用某种设计时的复杂度，从实际的系统需求入手，尽可能让常用的功能得到最简单直接的支持；还要一定程度上 "牺牲" 一些极少用到的功能，反复考虑系统使用场景，尽可能获得一个合理的折衷方案。

API 设计原则

在这个折衷的过程中，我们需要始终保证满足这些基本原则。

1. 保证 API 100% RESTful。RESTful 的核心是： everything is a "resource"，所有的行为（ Action ）和接口，都应该是相应 Resource 上的增删改查（CRUD）操作。如果脱离这种设计模式，一定要再三考虑是不是必要？有没有其他方案可以避免破坏 RESTful 风格。
2. 在请求和响应中，应该尽可能地保持参数的结构化。如果是一个哈希（hash），就传一个哈希（不要传 hash.to_string）。API 的序列化和反序列化机制（ Serialization / Deserialization ）会将其自动序列化成字符串。多语言之间的 API，比如 Ruby、Java、C# 之间的调用，通常都是在序列化和反序列化机制中完成不同语言间类型的转换。
3. 认证（Authentication ） 和安全（ Security ）的考虑。 安全的考虑始终应该放在首位，保证对特定的用户永远只暴露相关的接口和权限。可以使用证书和白名单，也可以通 Credentials Token Session / Cookie API Logging

过用户登陆的证书（ ）生成的验证票据（ ），或者 等方式来处理。此外，所有的 层的日志（ ），要保证不记录任何敏感的信息。

4. API 本身应该是客户端无关的。也就是说，一个 API 对请求的处理尽可能避免对客户端是 移动端还是网页端的考虑。客户端相关的响应格式，不应该在 API 中实现。所有的客户端无关的计算和处理，要尽可能在服务器（ Server ） 端统一处理，以提高性能和一致性。
5. 尽可能让 API 是幂等（Idempotent）的。关于幂等，可以参考我之前写的"聊聊幂等"一文。这里面有几个不同层次的含义。举例说明：同一个请求发一遍和发两遍是不是能够保证结果相同？请求失败后重发和第一次发是不是能保证相同结果？当然，要不要做成幂等，具体的实现还要看具体的应用场景。

使用好 API 框架

每个语言都已经提供了很好的 API 框架，你需要在设计前先多了解这些框架。如果你是个小团队，资源没那么充分，选一个合适的框架入手，适当调整，比从零开始造轮子要好得多。等公司长大了，由于各自业务逻辑的特殊需求，最终都会定制一套自己的 API 实现方案。

评估一个 API 框架，可以从以下几个方面考虑：

1. 对访问权限的统一控制
2. 自动测试的支持
3. 对请求和响应的格式，以及序列化和反序列化（ Serialization 和 Deserialization ）的支持
4. 对日志和日志过滤（Logging 和 Logging Filtering ） 的支持
5. 对自动文档生成的支持
6. 对架构以及性能的影响

设计中的平衡

API 设计中存在很多对立的因素，比如简洁还是繁复，兼容性和效率，为现在设计还是为未来打算等等。根据自己的工作实践，我给出以下观点供你参考：

1 自由总是相对的

就好像在一个群体里，如果没有规则，完全行为自由，就会出现各种问题。小群体还好，而对于一个大群体，有人就会被别人的"自由"误伤。

写软件也是一样。一个小的创业公司里，API 怎么设计，代码怎么写，几个人一协商，达成共识，并不需要那么多的条条框框，也照样行的通。

公司越大，代码协作的人越多，个人的自由就会在设计和实现中产生问题，并导致最终的冲突。所以，很多大公司会制定一些 API 的最佳实践，强制要求设计和实现中必须按照某种模式来做。

有些规则虽有道理，但也不是说不这样不行，所以在很多时候，因为这样的规则，我们的 API 设计中会有很多限制，这在表面上似乎给设计带来无谓的难度，但是仔细考量，从规范代码和设计一致性的角度而言，还是有很大好处的。

2 为当前设计，还是为未来设计？

API 设计里很常见的一个情况是：一个目前并没有人使用的系统功能，它的存在只是因为有人提出："这种情况我们以后应该要支持。" 前文中我曾讲过，由于 API 上线后再改很困难，所以在设计初期就要尽可能地考虑未来的发展；但是这些 "可能" 的应用场景因为需求的细节和使用频度都不明确，最容易造成系统的过度设计（ Over-design ）。

我记得有一个 API 设计的经典原则，概括一下就是：要考虑未来的场景，在设计时留有余地，但永远只实现当前产品真正要用的功能。

3 可维护性和效率（ Maintainability v.s. Efficiency）

设计和实现里常常会有一些封装和抽象的概念。某些特殊情况下，封装再分拆的过程可能会在一定程度上影响 API 的响应速度，或者代码质量的优化和性能的优化上有冲突。这个很难一概而论，具体的做法要看代码是否在关键路径上，或者这段代码是不是需要多人协作等等。最终的选择就要具体问题具体分析了。

4 是否采用AOP

AOP 本身就是一个极具争议的话题。概括说来，AOP 的理念是从主关注点中分离出横切关注点。

分离关注点使得解决特定领域问题的代码从业务逻辑中独立出来，业务逻辑的代码中不再含有针对特定领域问题代码的调用，业务逻辑同特定领域问题的关系通过侧面来封装、维护，这样原本分散在在整个应用程序中的变动就可以很好地管理起来。

因为 API 的设计和实现中有很多通用的关注点，如日志（ Logging）、解析（ Parsing）、监控（ Monitoring ）等等，所以 API 成了 AOP 一个很自然的应用领域。

使用 AOP 的 API 设计继承了 AOP 的优势，如：代码的重用性，规整性，以及程序员可以集中关注于系统的核心业务逻辑等；但也会自然而然地继承了 AOP 固有问题，例如代码的剖析（Profiling）和调试（Debugging）困难增加，对程序员的相关经验有更多要求，相互协作的要求也增强了，比如改变某一个功能可能会影响到其它的功能，等等。

是否选择使用 AOP，和你的需求场景，人员技能和设计复杂度息息相关，需要技术决策者根据具体环境做出判断。

今天我从两个小故事入手，和你讨论了 API 的设计和原则，内容分为四个部分：API 的签名、API 的设计原则、使用现有编程语言的 API 框架、如何在 API 设计中取得平衡。

API 设计是现代软件系统中不可或缺的一个环节，不同的系统需求和不同编程语言下，API 的设计都大不相同，但总有一些原则和注意事项是可以提取出来的，今天我和你讨论的就是这些通用的原则，希望对你的实际工作有帮助。

最后，给你留一道思考题，API 的签名（Signature）设计是语言无关的，那你在设计中会引入更多的语言还是更少的语言去实现不同的 API 呢，优点和缺点各是什么？期待你的回复，我们一起进步。下期再见。

戳此获取你的专属海报

水有同象番石榴

2017-12-24

感觉安姐讲这个「每个工程师都应该了解的」系列，比其他的文章长了三倍都不止，安姐一定是特别喜欢技术类的内容，谈起来洋洋洒洒。技术小白看起来有点难，不过有认真的做笔记。
我还在学习阶段 没有上手做内容 不过感觉api的设计和实现这篇讲了很多细节的问题 十分有帮助 谢谢安姐

蓝翔Sean

2017-12-22

安姐 对于API的RESTful有一些疑问 感觉并不是所有的API都能实现成RESTful的 有很多内容是没办法找到对应资源的 比如说login logout 用户其他的一些动作 对这些API的设计有什么建议吗

作者回复

2017-12-26

大部分都可以，你说的 login 和 logout 都是 API 啊，看看 time.geekbang.org 的账户系统就知道了

刘剑

2017-12-28

我们使用Spring Boot构建RESTful风格，我建议用更少的语言实现API以降低系统复杂度，也降低维护成本。

WTF

2017-12-28

API框架可否推荐简评下呢?

13683815260

2018-05-12

建议使用更少的语言创建这样可以通过通用的Aop,日志，限流等功能的实现，对外提供一套统一的api交互方式。
如果实现多语言的实现，可以通过rpc的方式进行封装。

当然如果系统足够复杂也可以通过service mesh的sidecar的方式进行管理。

Geek_746d06

2018-05-03

认真的读完了，带给我不少工作上的启发，也提供了一些很好的见解！zan

gggjtg

2018-03-29

如果有英文版就好了

VincentJiang

2018-02-06

想问下Ruby on Rails下写API比较好的Gem是什么?

王岩

2018-02-05

关于login in/login out，在我现在系统里，就是对于特定授权的创建/删除哈 /auth post和delete

赖晓强

2018-02-05

还不太能看懂，先做笔记。