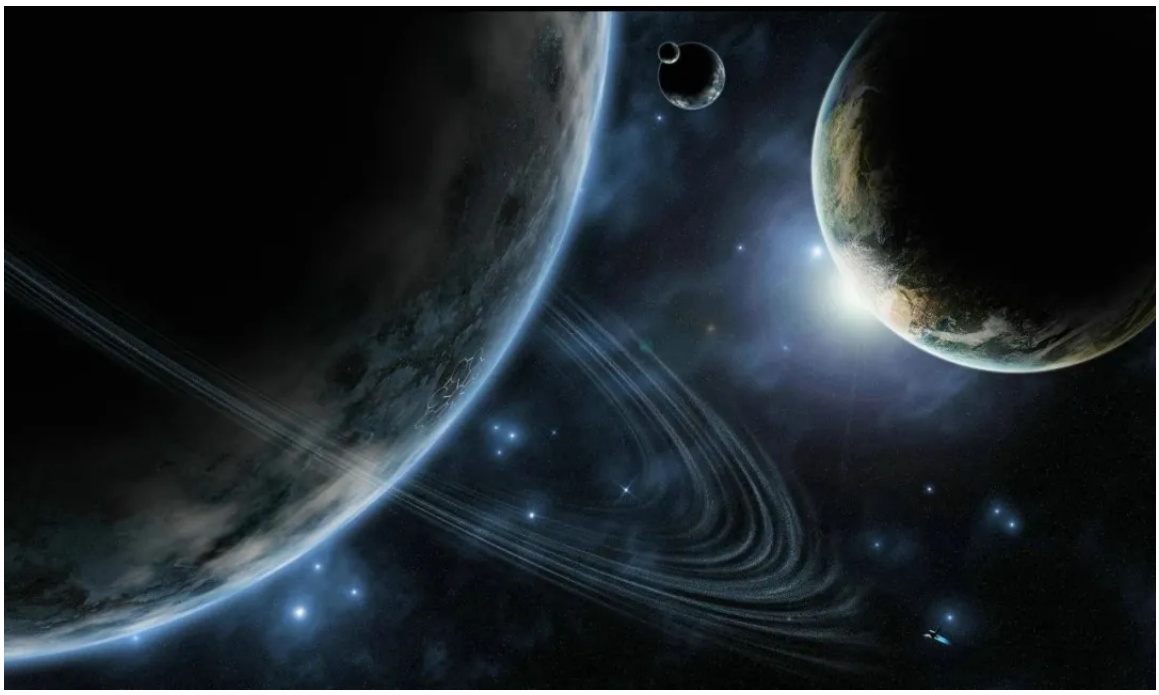
万字详文告诉你如何做 Code Review

服务端思维 前天



作者 |cheaterlin，腾讯 PCG 后台开发工程师
出品 |腾讯技术工程

前言

作为公司代码委员会 golang 分会的理事，我 review 了很多代码，看了很多别人的 review 评论。发现不少同学 code review 与写出好代码的水平有待提高。在这里，想分享一下我的一些理念和思路。

为什么技术人员包括 leader 都要做 code review

谚语曰: 'Talk Is Cheap, Show Me The Code'。知易行难，知行合一难。嘴里要讲出来总是轻松，把别人讲过的话记住，组织一下语言，再讲出来，很容易。绝知此事要躬行。设计理念你可能道听途说了一些，以为自己掌握了，但是你会做么？有能力去思考、改进自己当前的实践方式和实践中的代码细节么？不客气地说，很多人仅仅是知道并且认同了某个设计理念，进而产生了一种虚假的安心感---自己的技术并不差。但是，他根本没有去实践这些设计理念，甚至根本实践不了这些设计理念，从结果来说，他懂不懂这些道理/理念，有什么差别？变成了自欺欺人。

代码，是设计理念落地的地方，是技术的呈现和根本。同学们可以在 review 过程中做到落地沟通，不再是空对空的讨论，可以在实际问题中产生思考的碰撞，互相学习，大家都掌握团队里积累出来最好的实践方式！当然，如果 leader 没时间写代码，仅仅是 review 代码，指出其他同学某些实践方式不好，要给出好的实践的意见，即使没亲手写代码，也是对最佳实践要有很多思考。

为什么同学们要在 review 中思考和总结最佳实践

我这里先给一个我自己的总结：所谓架构师，就是掌握大量设计理念和原则、落地到各种语言及附带工具链（生态）下的实践方法、垂直行业模型理解，定制系统模型设计和工程实践规范细则。进而控制 30+万行代码项目的开发便利性、可维护性、可测试性、运营质量。

厉害的技术人，主要可以分为下面几个方向：

- 奇技淫巧

掌握很多技巧，以及发现技巧一系列思路，比如很多编程大赛，比的就是这个。但是，这个对工程，用处好像并不是很大。

- 领域奠基

比如约翰*卡马克，他创造出了现代计算机图形高效渲染的方法论。不论如果没有他，后面会不会有人发明，他就是第一个发明了。1999 年，卡马克登上了美国时代杂志评选出来的科技领域 50 大影响力人物榜单，并且名列第 10 位。但是，类似的殿堂级位置，没有几个，不够大家分，没我们的事儿。

- 理论研究

八十年代李开复博士坚持采用隐含马尔可夫模型的框架，成功地开发了世界上第一个大词汇量连续语音识别系统 Sphinx。我辈工程师，好像擅长这个的很少。

- 产品成功

小龙哥是标杆。

- 最佳实践

这个是大家都可以做到，按照上面架构师的定义。在这条路上走得好，就能为任何公司组建技术团队，组织建设高质量的系统。

从上面的讨论中，可以看出，我们普通工程师的进化之路，就是不断打磨最佳实践方法论、落地细节。

代码变坏的根源

在讨论什么代码是好代码之前，我们先讨论什么是不好的。计算机是人造的学科，我们自己制造了很多问题，进而去思考解法。

重复的代码

```
// BatchGetQQTinyWithAdmin 获取QQ uin的tinyID, 需要主uin的tiny和登录态
// friendUins 可以是空列表, 只要admin uin的tiny
func BatchGetQQTinyWithAdmin(ctx context.Context, adminUin uint64, friendUin []uint64) (
 adminTiny uint64, sig []byte, frdTiny map[uint64]uint64, err error) {
 var friendAccountList []*basedef.AccountInfo
 for _, v := range friendUin {
  friendAccountList = append(friendAccountList, &basedef.AccountInfo{
   AccountType: proto.String(def.StrQQU),
   Userid:      proto.String(fmt.Sprint(v)),
  })
 }

 req := &cmd0xb91.ReqBody{
  Appid:       proto.Uint32(model.DocAppID),
  CheckMethod: proto.String(CheckQQ),
  AdminAccount: &basedef.AccountInfo{
   AccountType: proto.String(def.StrQQU),
   Userid:      proto.String(fmt.Sprint(adminUin)),
  },
  FriendAccountList: friendAccountList,
 }
```

因为最开始协议设计得不好，第一个使用接口的人，没有类似上面这个函数的代码，自己实现了一个嵌入逻辑代码的填写请求结构结构体的代码，一开始，挺好的。但当有第二个人，第三个人干了类似的事情，我们将无法再重构这个协议，必须做到麻烦的向前兼容。而且每个同学，都要理解一遍上面这个协议怎么填，理解有问题，就触发 bug。或者，如果某个错误的理解，普遍存在，我们就得找到所有这些重复的片段，都修改一遍。

当你要读一个数据，发现两个地方有，不知道该选择哪个。当你要实现一个功能，发现两个 rpc 接口、两个函数能做到，你不知道选哪一个。你有面临过这样的'人生难题'么？其实怎么选并不重要了，你写的这个代码已经在走向 shit 的道路上迈出了坚实的一步。

但是，A little copying is better than a little dependency。这里提一嘴，不展开。

这里，我必须额外说一句。大家使用 trpc。感觉自己被鼓励'每个服务搞一个 git'。那，你这个服务里访问 db 的代码，rpc 的代码，各种可以复用的代码，是用的大家都复用的 git 下的代码么？每次都重复写一遍，db 字段细节改了，每个使用过 db 的 server 对应的 git 都改一遍？这

个通用 git 已经写好的接口应该不知道哪些 git 下的代码因为自己不向前兼容的修改而永远放弃了向前不兼容的修改?

早期有效的决策不再有效

很多时候，我们第一版代码写出来，是没有太大的问题的。比如，下面这个代码

```
// Update 增量更新
func (s *FilePrivilegeStore) Update(key def.PrivilegeKey,
 clear, isMerge bool, subtract []*access.AccessInfo, increment []*access.AccessInfo,
 policy *uint32, adv *access.AdvPolicy, shareKey string, importQQGroupID uint64) error {
 // 获取之前的数据
 info, err := s.Get(key)
 if err != nil {
  return err
 }

 incOnlyModify := update(info, &key, clear, subtract,
  increment, policy, adv, shareKey, importQQGroupID)
 stat := statAndUpdateAccessInfo(info)
 if !incOnlyModify {
  if stat.groupNumber > model.FilePrivilegeGroupMax {
   return errors.Errorf(errors.PrivilegeGroupLimit,
    "group num %d larger than limit %d",
    stat.groupNumber, model.FilePrivilegeGroupMax)
  }
 }

 if !isMerge {
  if key.DomainID == uint64(access.SPECIAL_FOLDER_DOMAIN_ID) &&
   len(info.AccessInfos) > model.FilePrivilegeMaxFolderNum {
   return errors.Errorf(errors.PrivilegeFolderLimit,
    "folder owner num %d larger than limit %d",
    len(info.AccessInfos), model.FilePrivilegeMaxFolderNum)
  }
  if len(info.AccessInfos) > model.FilePrivilegeMaxNum {
   return errors.Errorf(errors.PrivilegeUserLimit,
    "file owner num %d larger than limit %d",
    len(info.AccessInfos), model.FilePrivilegeMaxNum)
  }
 }

 pbDataSt := infoToData(info, &key)
 var updateBuf []byte
```

```
 if updateBuf, err = proto.Marshal(pbDataSt); err != nil {
  return errors.Wrapf(err, errors.MarshalPBError,
   "FilePrivilegeStore.Update Marshal data error, key[%v]", key)
 }
 if err = s.setCKV(generateKey(&key), updateBuf); err != nil {
  return errors.Wrapf(err, errors.Code(err),
   "FilePrivilegeStore.Update setCKV error, key[%v]", key)
 }
 return nil
}
```

现在看，这个代码挺好的，长度没超过 80 行，逻辑比价清晰。但是当 isMerge 这里判断逻辑，如果加入更多的逻辑，把局部行数撑到 50 行以上，这个函数，味道就坏了。出现两个问题：

1）函数内代码不在一个逻辑层次上，阅读代码，本来在阅读着顶层逻辑，突然就掉入了长达 50 行的 isMerge 的逻辑处理细节，还没看完，读者已经忘了前面的代码讲了什么，需要来回看，挑战自己大脑的 cache 尺寸。

2）代码有问题后，再新加代码的同学，是改还是不改前人写好的代码呢？出 bug 谁来背？这是一个灵魂拷问。

过早的优化

这个大家听了很多了，这里不赘述。

对合理性没有苛求

'两种写法都 ok，你随便挑一种吧'，'我这样也没什么吧'，这是我经常听到的话。

```
// Get 获取IP
func (i *IPGetter) Get(cardName string) string {
 i.l.RLock()
 ip, found := i.m[cardName]
 i.l.RUnlock()

 if found {
  return ip
 }

 i.l.Lock()
 var err error
 ip, err = getNetIP(cardName)
```

```
 if err == nil {
  i.m[cardName] = ip
 }

  i.l.Unlock()
 return ip
}
```

i.l.Unlock()可以放在当前的位置，也可以放在 i.l.Lock()下面，做成 defer。两种在最初构造的时候，好像都行。这个时候，很多同学态度就变得不坚决。实际上，这里必须是 defer 的。

```
  i.l.Lock()
 defer i.l.Unlock()

 var err error
 ip, err = getNetIP(cardName)
 if err != nil {
  return "127.0.0.1"
 }

 i.m[cardName] = ip
 return ip
```

这样的修改，是极有可能发生的，它还是要变成 defer，那，为什么不一开始就是 defer，进入最合理的状态？不一开始就进入最合理的状态，在后续协作中，其他同学很可能犯错！

总是面向对象/总喜欢封装

我是软件工程科班出身。学的第一门编程语言是 c++。教材是这本 。当时自己读完教材，初入程序设计之门，对于里面讲的'封装'，惊为天人，多么美妙的设计啊，面向对象，多么智慧的设计啊。但是，这些年来，我看到了大牛'云风'对于'毕业生使用 mysql api 就喜欢搞个 class 封装再用'的嘲讽；看到了各种莫名其妙的 class 定义；体会到了经常要去看一个莫名其妙的继承树，必须要把整个继承树整体读明白才能确认一个细小的逻辑分支；多次体会到了我需要辛苦地压抑住自己的抵触情绪，去细度一个自作聪明的被封装的代码，确认我的 bug。除了 UI 类场景，我认为少用继承、多用组合。

```
template<class _PKG_TYPE>
class CSuperAction : public CSuperActionBase {
  public:
    typedef _PKG_TYPE pkg_type;
    typedef CSuperAction<pkg_type> this_type;
```

```
    ...
}
```

这是 sspp 的代码。CSuperAction 和 CSuperActionBase，一会儿 super，一会儿 base，Super 和 SuperBase 是在怎样的两个抽象层次上，不通读代码，没人能读明白。我想确认任何细节，都要把多个层次的代码都通读了，有什么封装性可言？

好，你说是作者没有把 class name 取得好。那，问题是，你能取得好么？一个刚入职的 T1.2 的同学能把 class name、class 树设计得好么？即使是对简单的业务模型，也需要无数次'坏'的对象抽象实践，才能培养出一个具有合格的 class 抽象能力的同学，这对于大型却松散的团队协作，不是破坏性的？已经有了一套继承树，想要添加功能就只能在这个继承树里添加，以前的继承树不再适合新的需求，这个继承树上所有的 class，以及使用它们的地方，你都去改？不，是个正常人都会放弃，开始堆屎山。

封装，就是我可以不关心实现。但是，做一个稳定的系统，每一层设计都可能出问题。abi，总有合适的用法和不合适的用法，真的存在我们能完全不关心封装的部分是怎么实现的？不，你不能。bug 和性能问题，常常就出现在，你用了错误的用法去使用一个封装好的函数。即使是 android、ios 的 api，golang、java 现成的 api，我们常常都要去探究实现，才能把 api 用好。那，我们是不是该一上来，就做一个透明性很强的函数，才更为合理？使用者想知道细节，进来吧，我的实现很易读，你看看就明白，使用时不会迷路！对于逻辑复杂的函数，我们还要强调函数内部工作方式'可以让读者在大脑里想象呈现完整过程'的可现性，让使用者轻松读懂，有把握，使用时，不迷路！

根本没有设计

这个最可怕，所有需求，上手就是一顿撸，'设计是什么东西？我一个文件 5w 行，一个函数 5k 行，干不完需求？'从第一行代码开始，就是无设计的，随意地踩着满地的泥坑，对于旁人的眼光没有感觉，一个人独舞，产出的代码，完成了需求，毁灭了接手自己代码的人。这个就不举例了，每个同学应该都能在自己的项目类发现这种代码。

必须形而上的思考

常常，同学们听演讲，公开课，就喜欢听一些细枝末节的'干活'。这没有问题。但是，你干了几年活，学习了多少干货知识点？构建起自己的技术思考'面'，进入立体的'工程思维'，把技术细节和系统要满足的需求在思考上连接起来了么？当听一个需求的时候，你能思考到自己的 code package 该怎么组织，函数该怎么组织了么？

那，技术点要怎么和需求连接起来呢？答案很简单，你需要在时间里总结，总结出一些明确的原则、思维过程。思考怎么去总结，特别像是在思考哲学问题。从一些琐碎的细节中，由具体

情况上升到一些原则、公理。同时，大家在接受原则时，不应该是接受和记住原则本身，而应该是结构原则，让这个原则在自己这里重新推理一遍，自己完全掌握这个原则的适用范围。

再进一步具体地说，对于工程最佳实践的形而上的思考过程，就是：

把工程实践中遇到的问题，从问题类型和解法类型，两个角度去归类，总结出一些有限适用的原则，就从点到了面。把诸多总结出的原则，组合应用到自己的项目代码中，就是把多个面结合起来构建了一套立体的最佳实践的方案。当你这套方案能适应 30w+行代码的项目，超过 30 人的项目，你就架构师入门了！当你这个项目，是多端，多语言，代码量超过 300w 行，参与人数超过 300 人，代码质量依然很高，代码依然在高效地自我迭代，每天消除掉过时的代码，填充高质量的替换旧代码和新生的代码。恭喜你，你已经是一个很高级的架构师了！再进一步，你对某个业务模型有独到或者全面的理解，构建了一套行业第一的解决方案，结合刚才高质量实现的能力，实现了这么一个项目。没啥好说的，你已经是专家工程师了。级别再高，我就不了解了，不在这里讨论。

那么，我们要重头开始积累思考和总结？不，有一本书叫做《unix 编程艺术》，我在不同的时期分别读了 3 遍，等一会，我讲一些里面提到的，我觉得在腾讯尤其值得拿出来说的原则。这些原则，正好就能作为 code review 时大家判定代码质量的准绳。但，在那之前，我得讲一下另外一个很重要的话题，模型设计。

model 设计

没读过 oauth2.0 RFC，就去设计第三方授权登陆的人，终归还要再发明一个撇脚的 oauth。

2012 年我刚毕业，我和一个去了广州联通公司的华南理工毕业生聊天。当时他说他工作很不开心，因为工作里不经常写代码，而且认为自己有 ACM 竞赛金牌级的算法熟练度+对 CPP 代码的熟悉，写下一个个指针操作内存，什么程序写不出来，什么事情做不好。当时我觉得，挺有道理，编程工具在手，我什么事情做不了？

现在，我会告诉他，复杂如 linux 操作系统、Chromium 引擎、windows office，你做不了。原因是，他根本没进入软件工程的工程世界。不是会搬砖就能修出港珠澳大桥。但是，这么回答并不好，举证用的论据离我们太遥远了。见微知著。我现在会回答，你做不了，简单如一个权限系统，你知道怎么做么？堆积一堆逻辑层次一维展开的 if else？简单如一个共享文件管理，你知道怎么做么？堆积一堆逻辑层次一维展开的 ife lse？你联通有上万台服务器，你要怎么写一个管理平台？堆积一堆逻辑层次一维展开的 ife lse？

上来就是干，能实现上面提到的三个看似简单的需求？想一想，亚马逊、阿里云折腾了多少年，最后才找到了容器+Kubernetes 的大杀器。这里，需要谷歌多少年在 BORG 系统上的实践，提出了优秀的服务编排领域模型。权限领域，有 RBAC、DAC、MAC 等等模型，到了业务，又会有细节的不同。如 Domain Driven Design 说的，没有良好的领域思考和模型抽象，

逻辑复杂度就是 n^2 指数级的，你得写多少 ifelse，得思考多少可能的 if 路径，来 cover 所有的不合符预期的情况。你必须要有 Domain 思考探索、model 拆解/抽象/构建的能力。有人问过我，要怎么有效地获得这个能力？这个问题我没能回答，就像是在问我，怎么才能获得 MIT 博士的学术能力？我无法回答。唯一回答就是，进入某个领域，就是首先去看前人的思考，站在前人的肩膀上，再用上自己的通识能力，去进一步思考。至于怎么建立好的通识思考能力，可能得去常青藤读个书吧：）或者，就在工程实践中思考和锻炼自己的这个能力！

同时，基于 model 设计的代码，能更好地适应产品经理不断变更的需求。比如说，一个 calendar(日历)应用，简单来想，不要太简单！以'userid_date'为 key 记录一个用户的每日安排不就完成了么？只往前走一步，设计了一个任务，上限分发给 100w 个人，创建这么一个任务，是往 100w 个人下面添加一条记录？你得改掉之前的设计，换 db。再往前走一步，要拉出某个用户和某个人一起要参与的所有事务，是把两个人的所有任务来做 join？好像还行。如果是和 100 个人一起参与的所有任务呢？100 个人的任务来 join？不现实了吧。好，你引入一个群组 id，那么，你最开始的'userid_date'为 key 的设计，是不是又要修改和做数据迁移了？经常来一个需求，你就得把系统推翻重来，或者根本就只能拒绝用户的需求，这样的战斗力，还好意思叫自己工程师？你一开始就应该思考自己面对的业务领域，思考自己的日历应用可能的模型边界，把可能要做的能力都拿进来思考，构建一个 model，设计一套通用的 store 层接口，基于通用接口的逻辑代码。当产品不断发展，就是不停往模型里填内容，而不是推翻重来。这，思考模型边界，构建模型细节，就是两个很重要的能力，也是绝大多数腾讯产品经理不具备的能力，你得具备，对整个团队都是极其有益的。你面对产品经理时，就听取他们出于对用户体验负责思考出的需求点，到你自己这里，用一个完整的模型去涵盖这些零碎的点。

model 设计，是形而上思考中的一个方面，一个特别重要的方面。接下来，我们来抄袭抄袭 unix 操作系统构建的实践为我们提出的前人实践经验和'公理'总结。在自己的 coding/code review 中，站在巨人的肩膀上去思考。不重复地发现经典力学，而是往相对论挺进。

UNIX 设计哲学

不懂 Unix 的人注定最终还要重复发明一个撇脚的 Unix。--Henry Spenncer, 1987.11

下面这一段话太经典，我必须要摘抄一遍(自《UNIX 编程艺术》)：“工程和设计的每个分支都有自己的技术文化。在大多数工程领域中，就一个专业人员的素养组成来说，有些不成文的行业素养具有与标准手册及教科书同等重要的地位(并且随着专业人员经验的日积月累，这些经验常常会比书本更重要)。资深工程师们在工作中会积累大量的隐性知识，他们用类似禅宗'教外别传'的方式，通过言传身教传授给后辈。软件工程算是此规则的一个例外：技术变革如此之快，软件环境日新月异，软件技术文化暂如朝露。然而，例外之中也有例外。确有极少数软件技术被证明经久耐用，足以演进为强势的技术文化、有鲜明特色的艺术和世代相传的设计哲学。“

接下来，我用我的理解，讲解一下几个我们常常做不到的原则。

Keep It Simple Stuped!

KISS 原则，大家应该是如雷贯耳了。但是，你真的在遵守？什么是 Simple？简单？golang 语言主要设计者之一的 Rob Pike 说'大道至简'，这个'简'和简单是一个意思么？

首先，简单不是面对一个问题，我们印入眼帘第一映像的解法为简单。我说一句，感受一下。"把一个事情做出来容易，把事情用最简单有效的方法做出来，是一个很难的事情。"比如，做一个三方授权，oauth2.0 很简单，所有概念和细节都是紧凑、完备、易用的。你觉得要设计到 oauth2.0 这个效果很容易么？要做到简单，就要对自己处理的问题有全面的了解，然后需要不断积累思考，才能做到从各个角度和层级去认识这个问题，打磨出一个通俗、紧凑、完备的设计，就像 ios 的交互设计。简单不是容易做到的，需要大家在不断的时间和 code review 过程中去积累思考，pk 中触发思考，交流中总结思考，才能做得愈发地好，接近'大道至简'。

两张经典的模型图，简单又全面，感受一下，没看懂，可以立即自行 google 学习一下：
RBAC:

logging:

原则 3 组合原则: 设计时考虑拼接组合

关于 OOP，关于继承，我前面已经说过了。那我们怎么组织自己的模块？对，用组合的方式来达到。linux 操作系统离我们这么近，它是怎么架构起来的？往小里说，我们一个串联一个业务请求的数据集合，如果使用 BaseSession，XXXSession inherit BaseSession 的设计，其实，这个继承树，很难适应层出不穷的变化。但是如果使用组合，就可以拆解出 UserSignature 等等各种可能需要的部件，在需要的时候组合使用，不断添加新的部件而没有对老的继承树的记忆这个心智负担。

使用组合，其实就是要让你明确清楚自己现在所拥有的是哪个部件。如果部件过于多，其实完成组合最终成品这个步骤，就会有较高的心智负担，每个部件展开来，琳琅满目，眼花缭乱。比如 QT 这个通用 UI 框架，看它的Class 列表，有 1000 多个。如果不用继承树把它组织起来，平铺展开，组合出一个页面，将会变得心智负担高到无法承受。OOP 在'需要无数元素同时展现出来'这种复杂度极高的场景，有效的控制了复杂度 。'那么，古尔丹，代价是什么呢？'代价就是，一开始做出这个自上而下的设计，牵一发而动全身，每次调整都变得异常困难。

实际项目中，各种职业级别不同的同学一起协作修改一个 server 的代码，就会出现，职级低的同学改哪里都改不对，根本没能力进行修改，高级别的同学能修改对，也不愿意大规模修改，整个项目变得愈发不合理。对整个继承树没有完全认识的同学都没有资格进行任何一个对继承树有调整的修改，协作变得寸步难行。代码的修改，都变成了依赖一个高级架构师高强度监控继承体系的变化，低级别同学们束手束脚的结果。组合，就很好的解决了这个问题，把问题不断细分，每个同学都可以很好地攻克自己需要攻克的点，实现一个 package。产品逻辑代码，只需要去组合各个 package，就能达到效果。

这是 golang 标准库里 http request 的定义，它就是 Http 请求所有特性集合出来的结果。其中通用/异变/多种实现的部分，通过 duck interface 抽象，比如 Body io.ReadCloser。你想知道

哪些细节，就从组合成 request 的部件入手，要修改，只需要修改对应部件。[这段代码后，对比.NET 的 HTTP 基于 OOP 的抽象]

```
// A Request represents an HTTP request received by a server
// or to be sent by a client.
//
// The field semantics differ slightly between client and server
// usage. In addition to the notes on the fields below, see the
// documentation for Request.Write and RoundTripper.
type Request struct {
        // Method specifies the HTTP method (GET, POST, PUT, etc.).
        // For client requests, an empty string means GET.
        //
        // Go's HTTP client does not support sending a request with
        // the CONNECT method. See the documentation on Transport for
        // details.
        Method string

        // URL specifies either the URI being requested (for server
        // requests) or the URL to access (for client requests).
        //
        // For server requests, the URL is parsed from the URI
        // supplied on the Request-Line as stored in RequestURI.  For
        // most requests, fields other than Path and RawQuery will be
        // empty. (See RFC 7230, Section 5.3)
        //
        // For client requests, the URL's Host specifies the server to
        // connect to, while the Request's Host field optionally
        // specifies the Host header value to send in the HTTP
        // request.
        URL *url.URL

        // The protocol version for incoming server requests.
        //
        // For client requests, these fields are ignored. The HTTP
        // client code always uses either HTTP/1.1 or HTTP/2.
        // See the docs on Transport for details.
        Proto      string // "HTTP/1.0"
        ProtoMajor int    // 1
        ProtoMinor int    // 0

        // Header contains the request header fields either received
        // by the server or to be sent by the client.
        //
        // If a server received a request with header lines,
        //
        //      Host: example.com
        //      accept-encoding: gzip, deflate
        //      Accept-Language: en-us
        //      fOO: Bar
        //      foo: two
        //
        // then
        //
```

```
//
//     Header = map[string][]string{
//             "Accept-Encoding": {"gzip, deflate"},
//             "Accept-Language": {"en-us"},
//             "Foo": {"Bar", "two"},
//     }
//
// For incoming requests, the Host header is promoted to the
// Request.Host field and removed from the Header map.
//
// HTTP defines that header names are case-insensitive. The
// request parser implements this by using CanonicalHeaderKey,
// making the first character and any characters following a
// hyphen uppercase and the rest lowercase.
//
// For client requests, certain headers such as Content-Length
// and Connection are automatically written when needed and
// values in Header may be ignored. See the documentation
// for the Request.Write method.
Header Header

// Body is the request's body.
//
// For client requests, a nil body means the request has no
// body, such as a GET request. The HTTP Client's Transport
// is responsible for calling the Close method.
//
// For server requests, the Request Body is always non-nil
// but will return EOF immediately when no body is present.
// The Server will close the request body. The ServeHTTP
// Handler does not need to.
Body io.ReadCloser

// GetBody defines an optional func to return a new copy of
// Body. It is used for client requests when a redirect requires
// reading the body more than once. Use of GetBody still
// requires setting Body.
//
// For server requests, it is unused.
GetBody func() (io.ReadCloser, error)

// ContentLength records the length of the associated content.
// The value -1 indicates that the length is unknown.
// Values >= 0 indicate that the given number of bytes may
// be read from Body.
//
// For client requests, a value of 0 with a non-nil Body is
// also treated as unknown.
ContentLength int64

// TransferEncoding lists the transfer encodings from outermost to
// innermost. An empty list denotes the "identity" encoding.
// TransferEncoding can usually be ignored; chunked encoding is
// automatically added and removed as necessary when sending and
// receiving requests.
```

```
TransferEncoding []string

// Close indicates whether to close the connection after
// replying to this request (for servers) or after sending this
// request and reading its response (for clients).
//
// For server requests, the HTTP server handles this automatically
// and this field is not needed by Handlers.
//
// For client requests, setting this field prevents re-use of
// TCP connections between requests to the same hosts, as if
// Transport.DisableKeepAlives were set.
Close bool

// For server requests, Host specifies the host on which the
// URL is sought. For HTTP/1 (per RFC 7230, section 5.4), this
// is either the value of the "Host" header or the host name
// given in the URL itself. For HTTP/2, it is the value of the
// ":authority" pseudo-header field.
// It may be of the form "host:port". For international domain
// names, Host may be in Punycode or Unicode form. Use
// golang.org/x/net/idna to convert it to either format if
// needed.
// To prevent DNS rebinding attacks, server Handlers should
// validate that the Host header has a value for which the
// Handler considers itself authoritative. The included
// ServeMux supports patterns registered to particular host
// names and thus protects its registered Handlers.
//
// For client requests, Host optionally overrides the Host
// header to send. If empty, the Request.Write method uses
// the value of URL.Host. Host may contain an international
// domain name.
Host string

// Form contains the parsed form data, including both the URL
// field's query parameters and the PATCH, POST, or PUT form data.
// This field is only available after ParseForm is called.
// The HTTP client ignores Form and uses Body instead.
Form url.Values

// PostForm contains the parsed form data from PATCH, POST
// or PUT body parameters.
//
// This field is only available after ParseForm is called.
// The HTTP client ignores PostForm and uses Body instead.
PostForm url.Values

// MultipartForm is the parsed multipart form, including file uploads.
// This field is only available after ParseMultipartForm is called.
// The HTTP client ignores MultipartForm and uses Body instead.
MultipartForm *multipart.Form

// Trailer specifies additional headers that are sent after the request
// body.
```

```
//
// For server requests, the Trailer map initially contains only the
// trailer keys, with nil values. (The client declares which trailers it
// will later send.)  While the handler is reading from Body, it must
// not reference Trailer. After reading from Body returns EOF, Trailer
// can be read again and will contain non-nil values, if they were sent
// by the client.
//
// For client requests, Trailer must be initialized to a map containing
// the trailer keys to later send. The values may be nil or their final
// values. The ContentLength must be 0 or -1, to send a chunked request.
// After the HTTP request is sent the map values can be updated while
// the request body is read. Once the body returns EOF, the caller must
// not mutate Trailer.
//
// Few HTTP clients, servers, or proxies support HTTP trailers.
Trailer Header

// RemoteAddr allows HTTP servers and other software to record
// the network address that sent the request, usually for
// logging. This field is not filled in by ReadRequest and
// has no defined format. The HTTP server in this package
// sets RemoteAddr to an "IP:port" address before invoking a
// handler.
// This field is ignored by the HTTP client.
RemoteAddr string

// RequestURI is the unmodified request-target of the
// Request-Line (RFC 7230, Section 3.1.1) as sent by the client
// to a server. Usually the URL field should be used instead.
// It is an error to set this field in an HTTP client request.
RequestURI string

// TLS allows HTTP servers and other software to record
// information about the TLS connection on which the request
// was received. This field is not filled in by ReadRequest.
// The HTTP server in this package sets the field for
// TLS-enabled connections before invoking a handler;
// otherwise it leaves the field nil.
// This field is ignored by the HTTP client.
TLS *tls.ConnectionState

// Cancel is an optional channel whose closure indicates that the client
// request should be regarded as canceled. Not all implementations of
// RoundTripper may support Cancel.
//
// For server requests, this field is not applicable.
//
// Deprecated: Set the Request's context with NewRequestWithContext
// instead. If a Request's Cancel field and context are both
// set, it is undefined whether Cancel is respected.
Cancel <-chan struct{}

// Response is the redirect response which caused this request
// to be created. This field is only populated during client
```

```
        // redirects.
        Response *Response

        // ctx is either the client or server context. It should only
        // be modified via copying the whole Request using WithContext.
        // It is unexported to prevent people from using Context wrong
        // and mutating the contexts held by callers of the same request.
        ctx context.Context
}
```

看看.NET 里对于 web 服务的抽象，仅仅看到末端，不去看完整个继承树的完整图景，我根本无法知道我关心的某个细节在什么位置。进而，我要往整个 http 服务体系里修改任何功能，都无法抛开对整体完整设计的理解和熟悉，还极容易没有知觉地破坏者整体的设计。

说到组合，还有一个关系很紧密的词，叫插件化。大家都用 vscode 用得很开心，它比 visual studio 成功在哪里？如果 vscode 通过添加一堆插件达到 visual studio 具备的能力，那么它将变成另一个和 visual studio 差不多的东西，叫做 vs studio 吧。大家应该发现问题了，我们很多时候其实并不需要 visual studio 的大多数功能，而且希望灵活定制化一些比较小众的能力，用一些小众的插件。甚至，我们希望选择不同实现的同类型插件。这就是组合的力量，各种不同的组合，它简单，却又满足了各种需求，灵活多变，要实现一个插件，不需要事先掌握一个庞大的体系。体现在代码上，也是一样的道理。至少后端开发领域，组合，比 OOP，'香'很多。

原则 6 吝啬原则: 除非确无它法, 不要编写庞大的程序

可能有些同学会觉得，把程序写得庞大一些才好拿得出手去评 T11、T12。leader 们一看评审方案就容易觉得：很大，很好，很全面。但是，我们真的需要写这么大的程序么？

我又要说了"那么，古尔丹，代价是什么呢？"。代价是代码越多，越难维护，难调整。C 语言之父 Ken Thompson 说"删除一行代码，给我带来的成就感要比添加一行要大"。我们对于代码，要吝啬。能把系统做小，就不要做大。腾讯不乏 200w+行的客户端，很大，很牛。但是，同学们自问，现在还调整得动架构么。手 Q 的同学们，看看自己代码，曾经叹息过么。能小做的事情就小做，寻求通用化，通过 duck interface(甚至多进程，用于隔离能力的多线程)把模块、能力隔离开，时刻想着删减代码量，才能保持代码的可维护性和面对未来的需求、架构，调整自身的活力。客户端代码，UI 渲染模块可以复杂吊炸天，非 UI 部分应该追求最简单，能力接口化，可替换、重组合能力强。

落地到大家的代码，review 时，就应该最关注核心 struct 定义，构建起一个完备的模型，核心 interface，明确抽象 model 对外部的依赖，明确抽象 model 对外提供的能力。其他代码，就是要用最简单、平平无奇的代码实现模型内部细节。

原则 7 透明性原则: 设计要可见，以便审查和调试

首先，定义一下，什么是透明性和可显性。

"如果没有阴暗的角落和隐藏的深度，软件系统就是透明的。透明性是一种被动的品质。如果实际上能预测到程序行为的全部或大部分情况，并能建立简单的心理模型，这个程序就是透明的，因为可以看透机器究竟在干什么。

如果软件系统所包含的功能是为了帮助人们对软件建立正确的'做什么、怎么做'的心理模型而设计，这个软件系统就是可显的。因此，举例来说，对用户而言，良好的文档有助于提高可显性；对程序员而言，良好的变量和函数名有助于提高可显性。可显性是一种主动品质。在软件中要达到这一点，仅仅做到不晦涩是不够的，还必须要尽力做到有帮助。"

我们要写好程序，减少 bug，就要增强自己对代码的控制力。你始终做到，理解自己调用的函数/复用的代码大概是怎么实现的。不然，你可能就会在单线程状态机的 server 里调用有 IO 阻塞的函数，让自己的 server 吞吐量直接掉到底。进而，为了保证大家能对自己代码能做到有控制力，所有人写的函数，就必须具备很高的透明性。而不是写一些看了一阵看不明白的函数/代码，结果被迫使用你代码的人，直接放弃了对掌控力的追取，甚至放弃复用你的代码，另起炉灶，走向了'制造重复代码'的深渊。

透明性其实相对容易做到的，大家有意识地锻炼一两个月，就能做得很好。可显性就不容易了。有一个现象是，你写的每一个函数都不超过 80 行，每一行我都能看懂，但是你层层调用，很多函数调用，组合起来怎么就实现了某个功能，看两遍，还是看不懂。第三遍可能才能大概看懂。大概看懂了，但太复杂，很难在大脑里构建起你实现这个功能的整体流程。结果就是，阅读者根本做不到对你的代码有好的掌控力。

可显性的标准很简单，大家看一段代码，懂不懂，一下就明白了。但是，如何做好可显性？那就是要追求合理的函数分组，合理的函数上下级层次，同一层次的代码才会出现在同一个函数里，追求通俗易懂的函数分组分层方式，是通往可显性的道路。

当然，复杂如 linux 操作系统，office 文档，问题本身就很复杂，拆解、分层、组合得再合理，都难建立心理模型。这个时候，就需要完备的文档了。完备的文档还需要出现在离代码最近的地方，让人'知道这里复杂的逻辑有文档'，而不是其实文档，但是阅读者不知道。再看看上面 golang 标准库里的 http.Request，感受到它在可显性上的努力了么？对，就去学它。

原则 10 通俗原则: 接口设计避免标新立异

设计程序过于标新立异的话，可能会提升别人理解的难度。

一般，我们这么定义一个'点'，使用 x 表示横坐标，用 y 表示纵坐标：

```
type Point struct {
 X float64
```

```
 Y float64
}
```

你就是要不同、精准：

```
type Point struct {
 VerticalOrdinate   float64
 HorizontalOrdinate float64
}
```

很好，你用词很精准，一般人还驳斥不了你。但是，多数人读你的 VerticalOrdinate 就是没有读 X 理解来得快，来得容易懂、方便。你是在刻意制造协作成本。

上面的例子常见，但还不是最小立异原则最想说明的问题。想想一下，一个程序里，你把用'+'这个符号表示数组添加元素，而不是数学'加'，'result := 1+2' --> 'result = []int{1, 2}'而不是'result=3'，那么，你这个标新立异，对程序的破坏性，简直无法想象。"最小立异原则的另一面是避免表象想死而实际却略有不同。这会极端危险，因为表象相似往往导致人们产生错误的假定。所以最好让不同事物有明显区别，而不要看起来几乎一模一样。" -- Henry Spencer。

你实现一个 db.Add()函数却做着 db.AddOrUpdate()的操作，有人使用了你的接口，错误地把数据覆盖了。

原则 11 缄默原则: 如果一个程序没什么好说的，就沉默

这个原则，应该是大家最经常破坏的原则之一。一段简短的代码里插入了各种'log("cmd xxx enter")', 'log("req data " + req.String())'，非常害怕自己信息打印得不够。害怕自己不知道程序执行成功了，总要最后'log("success")'。但是，我问一下大家，你们真的耐心看过别人写的代码打的一堆日志么？不是自己需要哪个，就在一堆日志里，再打印一个日志出来一个带有特殊标记的日志'log("this_is_my_log_" + xxxxx)'？结果，第一个作者打印的日志，在代码交接给其他人或者在跟别人协作的时候，这个日志根本没有价值，反而提升了大家看日志的难度。

一个服务一跑起来，就疯狂打日志，请求处理正常也打一堆日志。滚滚而来的日志，把错误日志淹没在里面。错误日志失去了效果，简单地 tail 查看日志，眼花缭乱，看不出任何问题，这不就成了'为了捕获问题'而让自己'根本无法捕获问题'了么？

沉默是金。除了简单的 stat log，如果你的程序'发声'了，那么它抛出的信息就一定要有效！打印一个 log('process fail')也是毫无价值，到底什么 fail 了？是哪个用户带着什么参数在哪个环

节怎么 fail 了？如果发声，就要把必要信息给全。不然就是不发声，表示自己好好地 work 着呢。不发声就是最好的消息，现在我的 work 一切正常！

"设计良好的程序将用户的注意力视为有限的宝贵资源，只有在必要时才要求使用。"程序员自己的主力，也是宝贵的资源！只有有必要的时候，日志才跑来提醒程序员'我有问题，来看看'，而且，必须要给到足够的信息，让一把讲明白现在发生了什么。而不是程序员还需要很多辅助手段来搞明白到底发生了什么。

每当我发布程序 ，我抽查一个机器，看它的日志。发现只有每分钟外部接入、内部 rpc 的个数/延时分布日志的时候，我就心情很愉悦。我知道，这一分钟，它的成功率又是 100%，没任何问题！

原则 12 补救原则: 出现异常时，马上退出并给出足够错误信息

其实这个问题很简单，如果出现异常，异常并不会因为我们尝试掩盖它，它就不存在了。所以，程序错误和逻辑错误要严格区分对待。这是一个态度问题。

'异常是互联网服务器的常态'。逻辑错误通过 metrics 统计，我们做好告警分析。对于程序错误，我们就必须要严格做到在问题最早出现的位置就把必要的信息搜集起来，高调地告知开发和维护者'我出现异常了，请立即修复我!'。可以是直接就没有被捕获的 panic 了。也可以在一个最上层的位置统一做好 recover 机制，但是在 recover 的时候一定要能获得准确异常位置的准确异常信息。不能有中间 catch 机制，catch 之后丢失很多信息再往上传递。

很多 Java 开发的同学，不区分程序错误和逻辑错误，要么都很宽容，要么都很严格，对代码的可维护性是毁灭性的破坏。"我的程序没有程序错误，如果有，我当时就解决了。"只有这样，才能保持程序代码质量的相对稳定，在火苗出现时扑灭火灾是最好的扑灭火灾的方式。当然，更有效的方式是全面自动化测试的预防：）

具体实践点

前面提了好多思考方向的问题。大的原则问题和方向。我这里，再来给大家简单列举几个细节执行点吧。毕竟，大家要上手，是从执行开始，然后才是总结思考，能把我的思考方式抄过去。下面是针对 golang 语言的，其他语言略有不同。以及，我一时也想不全我所执行的 所有细则，这就是我强调'原则'的重要性，原则是可枚举的。

- 对于代码格式规范，100%严格执行，严重容不得一点沙。
- 文件绝不能超过 800 行，超过，一定要思考怎么拆文件。工程思维，就在于拆文件的时候积累。

- 函数对决不能超过 80 行，超过，一定要思考怎么拆函数，思考函数分组，层次。工程思维，就在于拆文件的时候积累。

- 代码嵌套层次不能超过 4 层，超过了就得改。多想想能不能 early return。工程思维，就在于拆文件的时候积累。

```
if !needContinue {
 doA()
 return
} else {
 doB()
 return
}
```

```
if !needContinue {
 doA()
 return
}

doB()
return
```

下面这个就是 early return，把两端代码从逻辑上解耦了。

- 从目录、package、文件、struct、function 一层层下来 ，信息一定不能出现冗余。比如 file.FileProperty 这种定义。只有每个'定语'只出现在一个位置，才为'做好逻辑、定义分组/分层'提供了可能性。
- 多用多级目录来组织代码所承载的信息，即使某一些中间目录只有一个子目录。
- 随着代码的扩展，老的代码违反了一些设计原则，应该立即原地局部重构，维持住代码质量不滑坡。比如:拆文件；拆函数；用 Session 来保存一个复杂的流程型函数的所有信息；重新调整目录结构。
- 基于上一点考虑，我们应该尽量让项目的代码有一定的组织、层次关系。我个人的当前实践是除了特别通用的代码，都放在一个 git 里。特别通用、修改少的代码，逐渐独立出 git，作为子 git 连接到当前项目 git，让 goland 的 Refactor 特性、各种 Refactor 工具能帮助我们快速、安全局部重构。
- 自己的项目代码，应该有一个内生的层级和逻辑关系。flat 平铺展开是非常不利于代码复用的。怎么复用、怎么组织复用，肯定会变成'人生难题'。T4-T7 的同学根本无力解决这种难题。

- 如果被 review 的代码虽然简短，但是你看了一眼却发现不咋懂，那就一定有问题。自己看不出来，就找高级别的同学交流。这是你和别 review 代码的同学成长的时刻。
- 日志要少打。要打日志就要把关键索引信息带上。必要的日志必须打。
- 有疑问就立即问，不要怕问错。让代码作者给出解释。不要怕问出极低问题。
- 不要说'建议'，提问题，就是刚，你 pk 不过我，就得改！
- 请积极使用 trpc。总是要和老板站在一起！只有和老板达成的对于代码质量建设的共识，才能在团队里更好地做好代码质量建设。
- 消灭重复！消灭重复！消灭重复！

主干开发

最后，我来为'主干开发'多说一句话。道理很简单，只有每次被 review 代码不到 500 行，reviewer 才能快速地看完，而且几乎不会看漏。超过 500 行，reviewer 就不能仔细看，只能大概浏览了。而且，让你调整 500 行代码内的逻辑比调整 3000 行甚至更多的代码，容易很多，降低不仅仅是 6 倍，而是一到两个数量级。有问题，在刚出现的时候就调整了，不会给被revew 的人带来大的修改负担。

关于 CI(continuous integration)，还有很多好的资料和书籍，大家应该及时去学习学习。

《unix 编程艺术》

建议大家把这本书找出来读一读。特别是，T7 及更高级别的同学。你们已经积累了大量的代码实践，亟需对'工程性'做思考总结。很多工程方法论都过时了，这本书的内容，是例外中的例外。它所表达出的内容没有因为软件技术的不断更替而过时。

佛教禅宗讲'不立文字'(不立文字，教外别传，直指人心，见性成佛)，很多道理和感悟是不能用文字传达的，文字的表达能力，不能表达。大家常常因为"自己听说过、知道某个道理"而产生一种安心感，认为"我懂了这个道理"，但是自己却不能在实践中做到。知易行难，知道却做不到，在工程实践里，就和'不懂这个道理'没有任何区别了。

曾经，我面试过一个别的公司的总监，讲得好像一套一套，代码拉出来遛一遛，根本就没做到，仅仅会道听途说。他在工程实践上的探索前路可以说已经基本断绝了。我只能祝君能做好向上管理，走自己的纯管理道路吧。请不要再说自己对技术有追求，是个技术人了！

所以，大家不仅仅是看看我这篇文章，而是在实践中去不断践行和积累自己的'教外别传'吧。

— 本文结束 —

往期推荐

- 漫谈设计模式在 Spring 框架中的良好实践