20讲理论六：我为何说KISS、YAGNI原则看似简单，却经常被用错

上几节课中，我们学习了经典的SOLID原则。今天，我们讲两个设计原则：KISS原则和YAGNI原则。其中，KISS原则比较经典，耳熟能详，但YAGNI你可能没怎么听过，不过它理解起来也不难。

理解这两个原则时候，经常会有一个共同的问题，那就是，看一眼就感觉懂了，但深究的话，又有很多细节问题不是很清楚。比如，怎么理解KISS原则中“简单”两个字？什么样的代码才算“简单”？怎样的代码才算“复杂”？如何才能写出“简单”的代码？YAGNI原则跟KISS原则说的是一回事吗？

如果你还不能非常清晰地回答出上面这几个问题，那恭喜你，又得到了一次进步提高的机会。等你听完这节课，我相信你很自然就能回答上来了。话不多说，让我们带着这些问题，正式开始今天的学习吧！

如何理解“KISS原则”？

KISS原则的英文描述有好几个版本，比如下面这几个。

- Keep It Simple and Stupid.
- Keep It Short and Simple.
- Keep It Simple and Straightforward.

不过，仔细看你就会发现，它们要表达的意思其实差不多，翻译成中文就是：尽量保持简单。

KISS原则算是一个万金油类型的设计原则，可以应用在很多场景中。它不仅经常用来指导软件开发，还经常用来指导更加广泛的系统设计、产品设计等，比如，冰箱、建筑、iPhone手机的设计等等。不过，咱们的专栏是讲代码设计的，所以，接下来，我还是重点讲解如何在编码开发中应用这条原则。

我们知道，代码的可读性和可维护性是衡量代码质量非常重要的两个标准。而KISS原则就是保持代码可读和可维护的重要手段。代码足够简单，也就意味着很容易读懂，bug比较难隐藏。即便出现bug，修复起来也比较简单。

不过，这条原则只是告诉我们，要保持代码“Simple and Stupid”，但并没有讲到，什么样的代码才是“Simple and Stupid”的，更没有给出特别明确的方法论，来指导如何开发出“Simple and Stupid”的代码。所以，看着非常简单，但不能落地，这就有点像我们常说的“心灵鸡汤”。哦，咱们这里应该叫“技术鸡汤”。

所以，接下来，为了能让这条原则切实地落地，能够指导实际的项目开发，我就针对刚刚的这些问题来进一步讲讲我的理解。

代码行数越少就越“简单”吗?

我们先一起看一个例子。下面这三段代码可以实现同样一个功能：检查输入的字符串ipAddress是否是合法的IP地址。

一个合法的IP地址由四个数字组成，并且通过“.”来进行分割。每组数字的取值范围是0~255。第一组数字比较特殊，不允许为0。对比这三段代码，你觉得哪一段代码最符合KISS原则呢？如果让你来实现这个功能，你会选择用哪种实现方法呢？你可以先自己思考一下，然后再看我下面的讲解。

```
// 第一种实现方式：使用正则表达式
public boolean isValidIpAddressV1(String ipAddress) {
  if (StringUtils.isBlank(ipAddress)) return false;
  String regex = "^(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|[1-9])\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)$";
  return ipAddress.matches(regex);
}

// 第二种实现方式：使用现成的工具类
public boolean isValidIpAddressV2(String ipAddress) {
  if (StringUtils.isBlank(ipAddress)) return false;
  String[] ipUnits = StringUtils.split(ipAddress, '.');
  if (ipUnits.length != 4) {
    return false;
  }
  for (int i = 0; i < 4; ++i) {
    int ipUnitIntValue;
    try {
      ipUnitIntValue = Integer.parseInt(ipUnits[i]);
    } catch (NumberFormatException e) {
      return false;
    }
    if (ipUnitIntValue < 0 || ipUnitIntValue > 255) {
      return false;
    }
    if (i == 0 && ipUnitIntValue == 0) {
      return false;
```

```
    }
  }
  return true;
}

// 第三种实现方式：不使用任何工具类
public boolean isValidIpAddressV3(String ipAddress) {
  char[] ipChars = ipAddress.toCharArray();
  int length = ipChars.length;
  int ipUnitIntValue = -1;
  boolean isFirstUnit = true;
  int unitsCount = 0;
  for (int i = 0; i < length; ++i) {
    char c = ipChars[i];
    if (c == '.') {
      if (ipUnitIntValue < 0 || ipUnitIntValue > 255) return false;
      if (isFirstUnit && ipUnitIntValue == 0) return false;
      if (isFirstUnit) isFirstUnit = false;
      ipUnitIntValue = -1;
      unitsCount++;
      continue;
    }
    if (c < '0' || c > '9') {
      return false;
    }
    if (ipUnitIntValue == -1) ipUnitIntValue = 0;
    ipUnitIntValue = ipUnitIntValue * 10 + (c - '0');
  }
  if (ipUnitIntValue < 0 || ipUnitIntValue > 255) return false;
  if (unitsCount != 3) return false;
  return true;
}
```

第一种实现方式利用的是正则表达式，只用三行代码就把这个问题搞定了。它的代码行数最少，那是不是就最符合KISS原则呢？答案是否定的。虽然代码行数最少，看似最简单，实际上却很复杂。这正是因为它使用了正则表达式。

一方面，正则表达式本身是比较复杂的，写出完全没有bug的正则表达本身就比较有挑战；另一方面，并不是每个程序员都精通正则表达式。对于不怎么懂正则表达式的同事来说，看懂并且维护这段正则表达式是比较困难的。这种实现方式会导致代码的可读性和可维护性变差，所以，从KISS原则的设计初衷上来讲，这种实现方式并不符合KISS原则。

讲完了第一种实现方式，我们再来看下其他两种实现方式。

第二种实现方式使用了StringUtils类、Integer类提供的一些现成的工具函数，来处理IP地址字符串。第三种实现方式，不使用任何工具函数，而是通过逐一处理IP地址中的字符，来判断是否合法。从代码行数上来说，这两种方式差不多。但是，第三种要比第二种更加有难度，更容易写出bug。从可读性上来说，第二种实现方式的代码逻辑更清晰、更好理解。所以，在这两种

实现方式中，第二种实现方式更加“简单”，更加符合KISS原则。

不过，你可能会说，第三种实现方式虽然实现起来稍微有点复杂，但性能要比第二种实现方式高一些啊。从性能的角度来说，选择第三种实现方式是不是更好些呢？

在回答这个问题之前，我先解释一下，为什么说第三种实现方式性能会更高一些。一般来说，工具类的功能都比较通用和全面，所以，在代码实现上，需要考虑和处理更多的细节，执行效率就会有所影响。而第三种实现方式，完全是自己操作底层字符，只针对IP地址这一种格式的数据输入来做处理，没有太多多余的函数调用和其他不必要的处理逻辑，所以，在执行效率上，这种类似定制化的处理代码方式肯定比通用的工具类要高些。

不过，尽管第三种实现方式性能更高些，但我还是更倾向于选择第二种实现方法。那是因为第三种实现方式实际上是一种过度优化。除非isValidIpAddress()函数是影响系统性能的瓶颈代码，否则，这样优化的投入产出比并不高，增加了代码实现的难度、牺牲了代码的可读性，性能上的提升却并不明显。

代码逻辑复杂就违背KISS原则吗？

刚刚我们提到，并不是代码行数越少就越“简单”，还要考虑逻辑复杂度、实现难度、代码的可读性等。那如果一段代码的逻辑复杂、实现难度大、可读性也不太好，是不是就一定违背KISS原则呢？在回答这个问题之前，我们先来看下面这段代码：

```
// KMP algorithm: a, b分别是主串和模式串；n, m分别是主串和模式串的长度。
public static int kmp(char[] a, int n, char[] b, int m) {
  int[] next = getNexts(b, m);
  int j = 0;
  for (int i = 0; i < n; ++i) {
    while (j > 0 && a[i] != b[j]) { // 一直找到a[i]和b[j]
      j = next[j - 1] + 1;
    }
    if (a[i] == b[j]) {
      ++j;
    }
    if (j == m) { // 找到匹配模式串的了
      return i - m + 1;
    }
  }
  return -1;
}

// b表示模式串，m表示模式串的长度
private static int[] getNexts(char[] b, int m) {
  int[] next = new int[m];
  next[0] = -1;
  int k = -1;
  for (int i = 1; i < m; ++i) {
    while (k != -1 && b[k + 1] != b[i]) {
      k = next[k];
    }
    if (b[k + 1] == b[i]) {
      ++k;
    }
    next[i] = k;
  }
  return next;
}
```

这段代码来自我的另一个专栏《数据结构与算法之美》中KMP字符串匹配算法的代码实现。这段代码完全符合我们刚提到的逻辑复杂、实现难度大、可读性差的特点，但它并不违反KISS原则。为什么这么说呢?

KMP算法以快速高效著称。当我们需要处理长文本字符串匹配问题（几百MB大小文本内容的匹配），或者字符串匹配是某个产品的核心功能（比如Vim、Word等文本编辑器），又或者字符串匹配算法是系统性能瓶颈的时候，我们就应该选择尽可能高效的KMP算法。而KMP算法本身具有逻辑复杂、实现难度大、可读性差的特点。本身就复杂的问题，用复杂的方法解决，并不违背KISS原则。

不过，平时的项目开发中涉及的字符串匹配问题，大部分都是针对比较小的文本。在这种情况下，直接调用编程语言提供的现

成的字符串匹配函数就足够了。如果非得用KMP算法、BM算法来实现字符串匹配，那就真的违背KISS原则了。也就是说，同样的代码，在某个业务场景下满足KISS原则，换一个应用场景可能就不满足了。

如何写出满足KISS原则的代码?

实际上，我们前面已经讲到了一些方法。这里我稍微总结一下。

- 不要使用同事可能不懂的技术来实现代码。比如前面例子中的正则表达式，还有一些编程语言中过于高级的语法等。
- 不要重复造轮子，要善于使用已经有的工具类库。经验证明，自己去实现这些类库，出bug的概率会更高，维护的成本也比较高。
- 不要过度优化。不要过度使用一些奇技淫巧（比如，位运算代替算术运算、复杂的条件语句代替if-else、使用一些过于底层的函数等）来优化代码，牺牲代码的可读性。

实际上，代码是否足够简单是一个挺主观的评判。同样的代码，有的人觉得简单，有的人觉得不够简单。而往往自己编写的代码，自己都会觉得够简单。所以，评判代码是否简单，还有一个很有效的间接方法，那就是code review。如果在code review的时候，同事对你的代码有很多疑问，那就说明你的代码有可能不够“简单”，需要优化啦。

这里我还想多说两句，我们在做开发的时候，一定不要过度设计，不要觉得简单的东西就没有技术含量。实际上，越是能用简单的方法解决复杂的问题，越能体现一个人的能力。

YAGNI跟KISS说的是一回事吗?

YAGNI原则的英文全称是：You Ain’t Gonna Need It。直译就是：你不会需要它。这条原则也算是万金油了。当用在软件开发中的时候，它的意思是：不要去设计当前用不到的功能；不要去编写当前用不到的代码。实际上，这条原则的核心思想就是：不要做过度设计。

比如，我们的系统暂时只用Redis存储配置信息，以后可能会用到ZooKeeper。根据YAGNI原则，在未用到ZooKeeper之前，我们没必要提前编写这部分代码。当然，这并不是说我们就不需要考虑代码的扩展性。我们还是要预留好扩展点，等到需要的时候，再去实现ZooKeeper存储配置信息这部分代码。

再比如，我们不要在项目中提前引入不需要依赖的开发包。对于Java程序员来说，我们经常使用Maven或者Gradle来管理依赖的类库（library）。我发现，有些同事为了避免开发中library包缺失而频繁地修改Maven或者Gradle配置文件，提前往项目里引入大量常用的library包。实际上，这样的做法也是违背YAGNI原则的。

从刚刚的分析我们可以看出，YAGNI原则跟KISS原则并非一回事儿。KISS原则讲的是“如何做”的问题（尽量保持简单），而YAGNI原则说的是“要不要做”的问题（当前不需要的就不要做）。

重点回顾

好了，今天的内容到此就讲完了。我们现在来总结回顾一下，你需要掌握的重点内容。

KISS原则是保持代码可读和可维护的重要手段。KISS原则中的“简单”并不是以代码行数来考量的。代码行数越少并不代表代码越简单，我们还要考虑逻辑复杂度、实现难度、代码的可读性等。而且，本身就复杂的问题，用复杂的方法解决，并不违背KISS原则。除此之外，同样的代码，在某个业务场景下满足KISS原则，换一个应用场景可能就不满足了。

对于如何写出满足KISS原则的代码，我还总结了下面几条指导原则：

- 不要使用同事可能不懂的技术来实现代码；
- 不要重复造轮子，要善于使用已经有的工具类库；
- 不要过度优化。

课堂讨论

你怎么看待在开发中重复造轮子这件事情？什么时候要重复造轮子？什么时候应该使用现成的工具类库、开源框架？

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言

辣么大

很好奇这三个方法运行效率的高低。测了一下争哥给的代码的执行效率，结果正如争哥文章说，第三个是最快的。
方法一（正则）< 方法二 < 方法三

正则就真的这么不堪么？效率如此之低？其实不然。

Java中正则的最佳实践是：
用于校验和匹配的正则表达式使用时需要预先compile，在类中写做静态变量（经常会重用），这样可以提高效率。Pattern.compile(IP_REGEXP)

优化正则后的效率如何呢?
方法一（正则）< 方法二 < 方法一（正则改进后）< 方法三

测试参数设置：每个方法执行100万次。
实验结果：
方法一：2.057s
方法一优化：0.296s 提前编译正则
方法二：0.622s
方法三：0.019s
有兴趣的小伙伴看看代码，欢迎一起讨论！https://github.com/gdhucoder/Algorithms4/blob/master/designpattern/u20/TestPerformance.java

参考：
https://stackoverflow.com/questions/1720191/java-util-regex-importance-of-pattern-compile

2019-12-18 06:28

失火的夏天

开发中的重复造轮子，这东西怎么说呢。我认为这句话是对公司来说的，但是对自己来说，重复造轮子是有必要的。就好比之前的数据结构和算法，那也是所有轮子都有啊，为什么还要自己写响应代码。这个问题在另一个专栏都说烂了，这里也不再赘述了。

光说不练假把式，轮子用不用的好，自己了解的深入才知道。我们读书的时候，用数学定理去解题，也是在一些已知条件下才能用这个定理。不能上来就套定理，而是要分析是否满足使用情况。只有吃透定义和原理，才能更好的使用轮子。

开发中也一样，我们要排查各种各样的问题，如果轮子内部出了问题，对轮子原理实现都不了解的，怎么排查问题。怎么知道用那种轮子是最好的呢。而且还会存在改造轮子的情况，轮子不满足我们特定的需求，那也是吃透了才能改的动。

总之，轮子这东西，对公司来说，不要重复，对自己来说，要多去理解，多动手。总不希望自己就是个调包侠吧

2019-12-18 07:18

李小四

设计模式_19
今天的内容有一些哲学味道，让我联想到奥卡姆剃刀原理：
如无必要，勿增实体。

用同事不懂的技术，增加了整个团队的理解成本；重复造轮子和过度优化，大多数情况下带来的便利小于增加的成本；

不要写炫技的代码，就像杜峰(CBA广东队教练)说的：“如果没有目的，就不要运球。(因为运球就可能丢球)”，降低出错的概率是一个数学问题，它能够真实得提高软件质量。

回到作业，同上：
只有必须造轮子时，才要造轮子。
那什么又是必须呢?
- Vim如果不用KMP，恐怕就没有人用了。
- MySql的性能(即将)已经不能满足阿里的业务量
- 微信作为国民应用，需要解决各种弱网络下尽可能收发消息。
...

2019-12-18 09:10

Ken张云忠

你怎么看待在开发中重复造轮子这件事情?
轮子:供上层业务应用使用的基础组件.
造轮子:设计并实现基础组件.
重复造轮子:重新设计实现一套新的组件.
开发中重复造轮子:
对于个人可以有深度度地提升对业务与技术认知的理解,还可以提升设计与动手能力,由于掌握了更多细节的知识,以后对于这类问题的排查定位及处理是有益处的.
对于技术团队,重复造出的轮子多了日后就需要有更多的人手和精力维护各个轮子,使轮子的维护成本变高;在使用方面,团队的每个成员为了能够正确的使用轮子就需要花费精力去熟悉各个轮子的特征.
什么时候要重复造轮子?
新的业务场景中原来的轮子不再合用,并且改造的成本高于重新建造的成本.比如原有业务量不大对于性能要求一般时,旧轮子足够满足;但随着业务的迅猛增长,对于性能提出明确苛刻的要求,就可以考虑重新建造新轮子了.
什么时候应该使用现成的工具类库、开源框架?
业务发展的初中期阶段时应该使用.这个阶段业务需求一般比较通用且对性能要求也不高,这时的业务与技术的特点就是要快,快速响应业务占领市场.
但发展到一定规模,性能成为了制约业务发展的瓶颈,拿来主义已经不能满足业务的更高要求了,就必须要动手建造适合自身业务需求的特定轮子.

2019-12-18 08:50

小毅

“不要使用同事可能不懂的技术来实现代码”这一条我觉得是可以值得商榷的~ 比如在项目中引入新技术就可能会违反这一条，我觉得关键点在于这个新技术是否值得引入？新技术是否可以在团队中得到推广?

有时候，在code review看到不理解的新技术时，其实刚好也是可以触发讨论和学习，如果只是单纯的不去使用，很容易造成整个技术团队停滞不前。

2019-12-18 20:40

作者回复

嗯嗯 我的意思不能为了用新技术而引入新技术 不过也没那么绝对 设计问题本来就没有绝对的对重庆更多的是弄明白道理之后根据实际场景自己去权衡

2019-12-20 07:15

下雨天

一.什么时候要重复造轮子?
1. 想学习轮子原理(有轮子但是不意味着你不要了解其原理)
2. 现有轮子不满足性能需求(轮子一般会考虑大而全的容错处理和前置判断，这些对性能都有损耗)
3. 小而简的场景(比如设计一个sdk,最好不宜过大，里面轮子太多不易三方集成)
4.定制化需求

二.什么时候应该使用现成的工具类库、开源框架?
1.第一条中提到的反面情况
2. 敏捷开发，快速迭代，需求急的场景
3. 轮子比自己写的好，BUG少，设计好

4. KISS原则

2019-12-18 10:13

再见孙悟空

很早就知道 kiss 原则了，以前的理解是代码行数少，逻辑简单，不要过度设计这样就符合 kiss 原则。虽然知道这个原则，但是却没有好好在实践中注意到，导致写的代码有时候晦涩难懂，有时候层层调用，跟踪起来很繁琐。看完今天的文章，理解更深了，代码不仅是给自己看的，也是给同事看的，并且尽量不自己造轮子，使用大家普遍知道的技术或方法会比炫技好很多。至于另一个原则，你不需要它这个实际上做的还是不错的。

2019-12-18 08:38

Pismery

争哥好，想请问一个问题，文中说到不要写同事可能不懂的技术实现，这该如何权衡呢；

对于 Java 8 的 lambda 表达式，我认为这样的代码会更为直观；可是由于同事都习惯使用存储过程，Java 7 的语法糖；

是否因为团队大部分人都不使用 lambda，就应该在项目中放弃使用呢?

2019-12-19 08:28

作者回复

应该暂时放弃 等团队慢慢学习接受

2019-12-20 07:12

Ken张云忠

提个疑问:

文中"所以，评判代码是否简单，还有一个很有效的间接方法，那就是 code review。",这里的code review有个前提该是团队成员的技术要有一定的水平,如果全是些初级人员,不按照面条式写代码就看起来费劲,这种代码评审就没意义了,所以前提是评审必须要有一定的内功修为.

2019-12-18 08:55

作者回复

你说的没错 要有大牛 不然code review就流于形式 大眼瞪小眼

2019-12-20 08:44

编程界的小学生

我觉得如果开源类库完全能满足需求的话，那完全没必要造轮子，如果对性能有要求，比如类库太复杂，想要简单高效的，那可以造个轮子，比如我认为shiro也是spring security的轮子，他简化了很多东西，小巧灵活。还有就是觉得类库能满足需求但是相对于当前需求来讲不够可扩展，那也可以采取类库思想造一个全新的轮子来用。

2019-12-18 00:16

小先生

那请问像正则表达式这样的东西是不是就没有用武之地?

2019-12-19 08:29

作者回复

当然不是 简单的正则表达式 大部分人都能看懂 就可以用 别整那种极端难看懂的就行 凡事都有个度 合理把握

2019-12-20 07:11

ca01ei

要不这样吧，如果编程语言里有个地方你弄不明白，而正好又有个人用了这个功能，那就开枪把他打死。这比学习新特性要容易些，然后过不了多久，那些活下来的程序员就会开始用0.9.6版的Python，而且他们只需要使用这个版本中易于理解的那一小部分就好了（眨眼）。

——Tim Peters 传奇的核心开发者，“Python之禅”作者

2019-12-18 09:00

黄林晴

对工作上重复造轮子，没有必要，因为讲究效率问题，别人不会管你实现的功能是复制粘贴的，还是自己实现的，能正常使用就ok，对于自己来说也没必要盲目造轮子，不要造大轮子，除非你觉得你造的轮子可以碾压现有的，造一些小轮子，使用别的轮子的思想和设计还是有些用处的。

2019-12-18 08:20

袁慎建

可能大家会说KISS、YAGNI这些原则听起来容易，做起来难。其实我认为不是做起来难，而是难在为什么要做和为什么不做。回到写代码这份职业来说，我觉出三个不同的问题：

1. 你为什么要写可用代码代码？ -- 赢得公司的信任，让自己能够活下来

2. 你为什么要写简洁可用的代码？ -- 解放自己的生产力，创造更多价值，升职加薪
3. 你为什么要写简洁可用，并影响其他人？ -- 赢得别人尊重，获得职业成就感

我觉得程序员首先要思考上面三个事情为什么要做，而且要回答上这些问题，需要自己持续精进，比如提升编码Sense、设计思维、提升系统思考能力。

在探索上面三个问题的道路上，很多开发人员可能有如下内心小九九：

1. “我要用炫酷的方式Show技能。看，正则，你看得懂吗，哈哈。小样”
2. “我要用最少的代码写出这个而功能。看，我的代码就是比你少”
3. “我要用极致的性能来写这个功能。来，测一测，谁的用时最短，我开始用了快速排序法的”
4. “我要用上5个设计模式。你瞧瞧，这里有5个设计模式的精髓，健壮无比的代码”
5. ....

以上这些内心小九九是我们要去尽量克制的，尽量少做、甚至不做，而为什么不做这些事情，去思考这个问题的答案更重要。

道理可能都懂，缺的是刻意练习，推荐一些实操落的方法论和实践：
1. 简单设计（4原则和优先级）
2. 测试驱动开发（Tasking + TDD）
3. 重构（作者后面会提到）

2020-02-05 23:28

无所从来

KISS用于设计之初的理念，是规划；YAGNI用于设计之后的优化，是减法。

2020-01-02 23:26

阿卡牛

Ken Thompson :拿不准，用穷举

2019-12-20 09:26

whistleman

打卡～

我觉得做项目中是不需要去重复造轮子的，但如果一个轮子特别大，但我只需要这个轮子很小的一部分内容，那是不是考虑借鉴它的思想去造个轮子呢?

2019-12-19 13:49

作者回复

是的 这是重复造轮子吗一个比较重要的理由

2019-12-20 07:06

拖鞋党副长

老师，以java为例，什么样的写法可以被称为不建议使用的过于高级的语法呢

2019-12-19 09:03

作者回复

比如一些函数式编程语法 有人就不了解

2019-12-20 07:09

Boogie 捷

想请问一下老师对于使用正则匹配的看法，因为在平时的工作中还是会时不时看见，而且感觉确实可以省掉一些代码。

2019-12-18 07:27

作者回复

简单的正则表达式可以用的 不要太极端就好

2019-12-20 08:54

张理查rootv

代码评审时，其实KISS原则很好判断，就是代码是在解决问题还是在炫技。KMP解决底层性能瓶颈就是在解决问题，如果是在

解决输入框匹配，就是在炫技。

2020-02-11 17:13