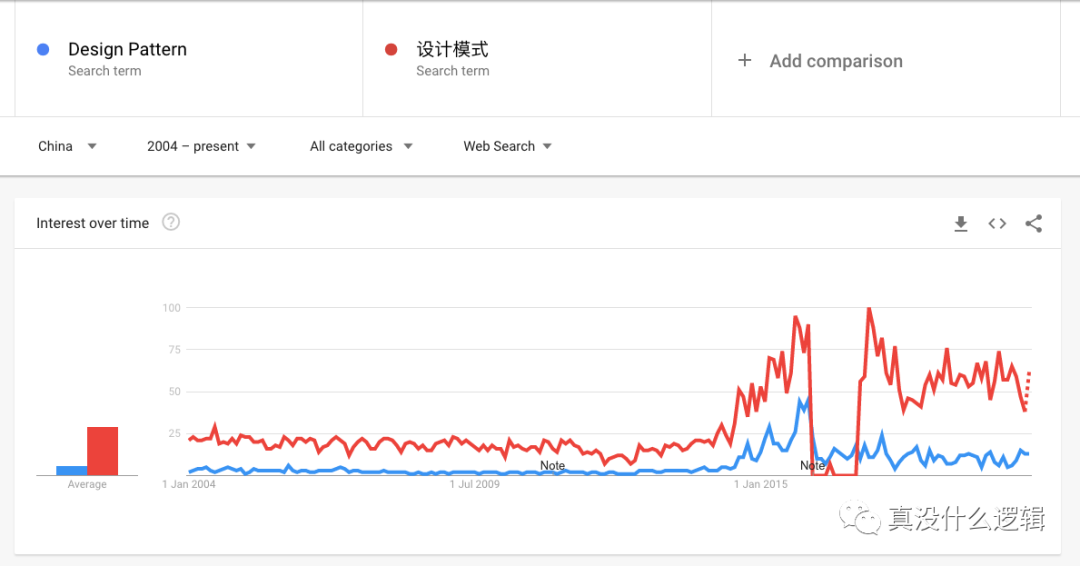
<https://mp.weixin.qq.com/s/Q1QFgJQUOaBSWYvZHkhlZA>

圣杯与银弹 · 没用的设计模式

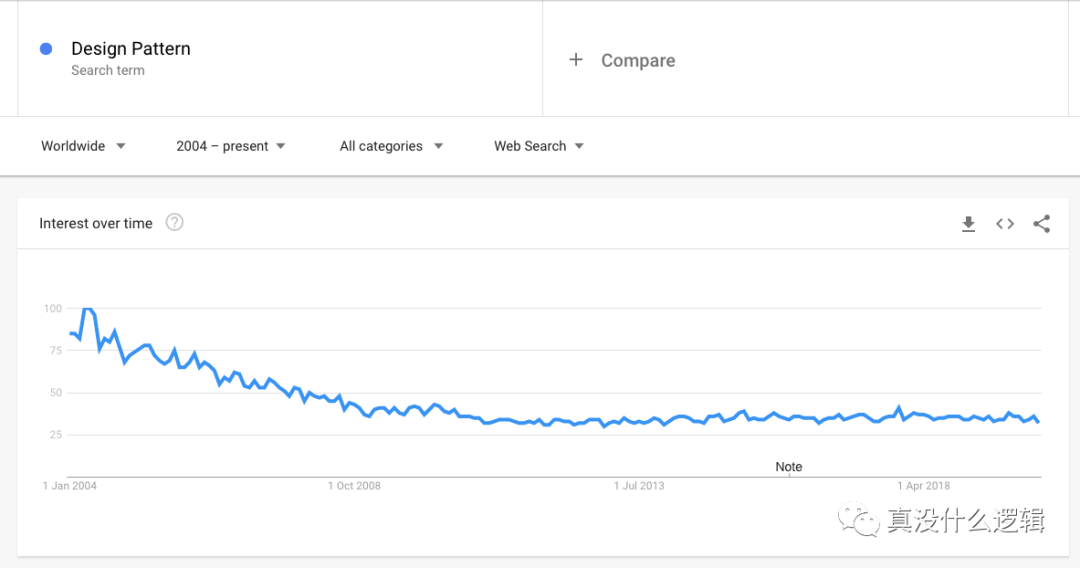
设计模式是软件工程中听起来非常深奥，也非常高端的一个词汇，似乎有了设计模式，我们的代码和项目就能自然的变得非常合理并且易于扩展和维护，然而事情并没有这么简单，软件工程中没有银弹。

我们在今天谈论设计模式时，往往与 1994 年 Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson 四个人出版的《设计模式》一书[^1]有着密切的关系，想要避开这本书讨论设计模式也非常困难。这本书在出版的 6 年后被机械工业出版社于 2000 年引进于中国，刚被引进的时候，因为大环境的原因，这本书还不是特别火热，但是自从 2014 年 5 月开始，由于国内互联网行业的迅速发展和信息产业的进步，设计模式一词也变得越来越热门[^2]。



**图 1 - 设计模式在中国的搜索指数**

很多工程师都在研究设计模式，企图让自己设计的软件变得更加优秀，一些候选人也在简历上写着自己掌握多少种设计模式并在面试时讨论书中的那 23 种设计模式。与设计模式在中国的流行相比，在全世界范围内，设计模式的热度从 2004 年开始却一直在下降[^3]。



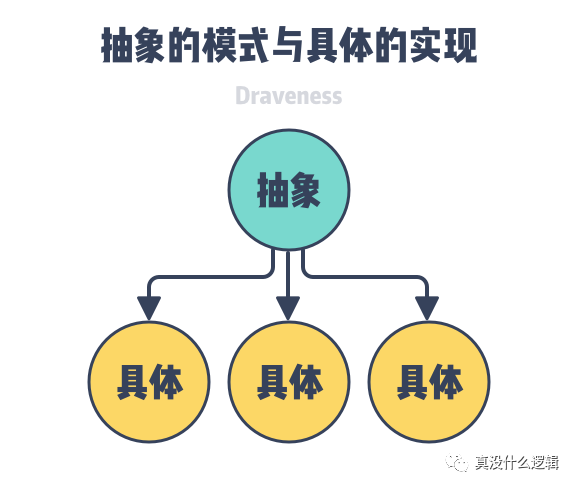
上述数据难道说明全世界的开发者都不在乎如何设计更优秀的软件吗，作者觉得这答案一定是否定的，设计出架构良好的软件是工程师追求的目标，设计糟糕的软件是无法持续维护的，只能一次又一次的重构或者重写。为了减少自己的麻烦，虽然从短期看来工程师都倾向于写出容易实现的逻辑，但是从长期看来工程师更倾向于写出容易维护的代码。

作者在本文的观点是，**学习设计模式与设计优秀的软件并不相关，盲目追求和套用书中的设计模式只能使项目变得更加糟糕**，本文并不是要批判《设计模式》中提出的通用的设计模型，我们需要对书中的内容多一些批判性的思考，想清楚究竟什么样的设计才是能够保留下来的。需要注意的是，这篇文章**充满了作者的主观意见和个人经验**，如果读者不认同本文的观点，欢迎在文章下面留言并讨论。

在讨论设计模式的作用之前，我们需要先理解它的定义。在软件工程中，**软件设计模式是在特定上下文下对于普遍出现问题的可重用解决方案**[^4]。这个定义非常严谨，我们可以发现几个关键的形容词，特定的（Specific）、普遍的（Commonly）和可重用（Reusable），这几个形容词说明了设计模式的特性以及它的局限性，本文将分三个部分分析作者为什么觉得设计模式没用。

## 可重用的解决方案

抽象的设计模式是从不同具体项目中总结出来的通用经验，从具体到抽象的过程相对容易，然而**从抽象的模式套用到具体场景却很困难**，如果没有足够的经验或者思考只会做出拙劣的设计。设计模式是从具体场景中总结的解决方案，它会站在一个更高的角度提炼实现中的关键细节，这样才能提供更好的通用性。

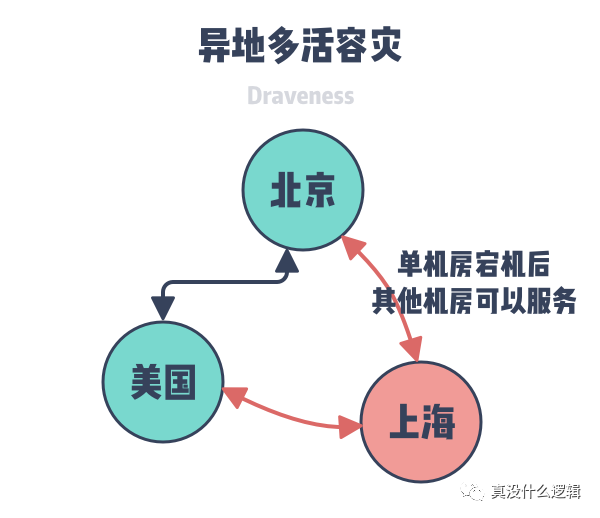


因为我们需要在多个不同的实现和场景中寻找类似的模式，所以在提炼设计模式的过程中一定会失去很多实现细节。作为理论来讲，精炼的、抽象的定义才能够更好的传播和重用，但是不同的读者在理解这些定义时会遇到两个问题：

* 经验较少的工程师 - 虽然书中的例子都看懂了，但是一到具体场景就没有办法利用；
* 有经验的工程师 - 虽然书中的定义和例子都没有问题，但是这些我们早就知道了；

相信稍微有一些编程经验的人都能从经历过的项目中总结出很多模式，从具体到抽象的过程需要积累较多的素材，不过这个过程相对比较容易 — 当我们学习的、实践的项目足够多时，我们自然能够发现其中存在的常见模式。

如果要从抽象的理论直接推导出出具体的实现是比较困难的，我们在这里简单举一个例子，我们都知道增加副本和备用服务器是提高服务可用性的一种常见方式 — **避免单点故障可以提高系统的可用性**。当这种看起来普通的大白话出现在书时，我们往往都会一带而过，觉得这是废话。然而在实践中，我们可以遵循这个理论设计出从简单到复杂的服务架构，例如：同机房的多实例部署和异地多活等方式。



多实例部署和异地多活都是能够避免单点故障的方法，然而实现这些方案需要考虑非常多的细节，这些细节都是在理论中缺失的，是需要通过经验和思考来补齐的，例如：部署多个实例之后，我们是不是还需要考虑服务注册、服务发现以及负载均衡的路由策略；异地多活是不是也要考虑机房之间的网络延迟、专用网络通道的搭建以及数据不一致的问题，这些实现细节在总结成规律抽象的理论时基本都消失了。

The devil is in the detail.[^5] - Ludwig Mies Van Der Rohe

而且并不是居高临下的架构设计才是系统设计，每个包、方法甚至代码中的空行中都体现了作者的设计思路，抽象的理论和模式能够起到指导的作用，但是真正让设计融入系统的还是工程师的丰富经验和深入思考。

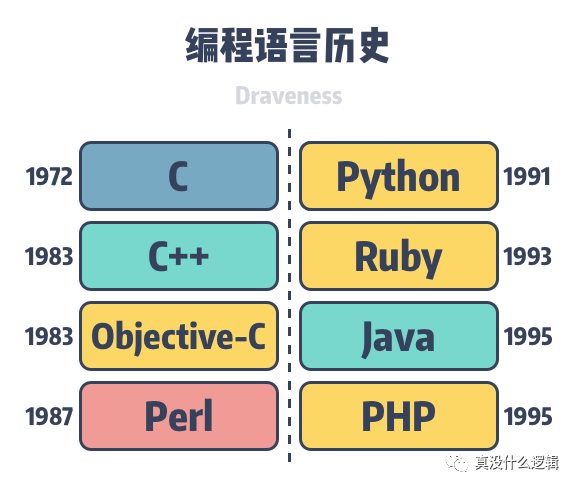
## 锤子和钉子

设计模式既不能让我们学到绝世武功，也不能帮我们打通任督二脉，从此看破系统中的所有巧妙设计，认为自己学到了神功，想要在项目中大展身手套用书中的设计时很有可能会带来错误设计，成为项目中的遗留代码（Legacy code）并被接手的工程师吐槽和重构。如果我们手中只有 23 种设计模式能够指导系统的设计，那么待解决的软件设计问题看起来都像钉子一样，看起来都可以用这些模式『巧妙』的解决，但是场景与场景是非常不同的，**我们要清楚地知道设计模式的局限性以及待解决问题的上下文，才能做出好的设计**。

If all you have is a hammer, everything looks like a nail! [^6] - Charlie Munger

《设计模式》书中描述的 23 种模式是有相当局限性的，如果仔细阅读这本书的副标题可能会让我们在应用具体的设计模式之前变得更加谨慎 — "Elements of reusable Object-Oriented Software."，这个副标题非常清晰的介绍了书中涉及的内容。

我们今天熟悉的大多数编程语言都是在上世纪 80 ~ 90 年代诞生的，包括 C++、Python、Ruby、Java、PHP 和 JavaScript，大多数的语言都是面向对象语言，而书中介绍的设计模式也都是在使用面向对象语言的项目中总结的。

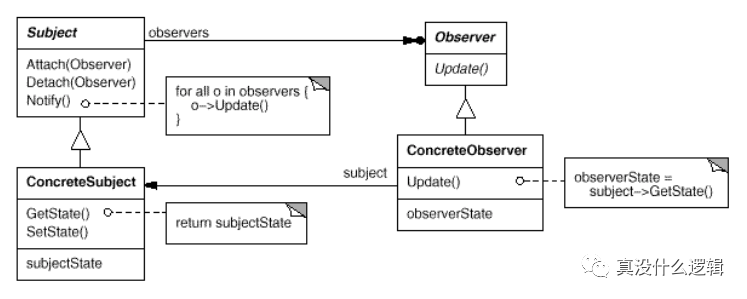


今天的编程语言已经变得越来越复杂，多数语言都同时支持多种编程范式 — 面向对象、函数式编程等，在应用书中的设计模式时，我们需要充分考虑到其解决的关键问题以及我们手中的编程语言是否有更好的实现方式，本节将通过以下两部分内容分析设计模式的局限性：

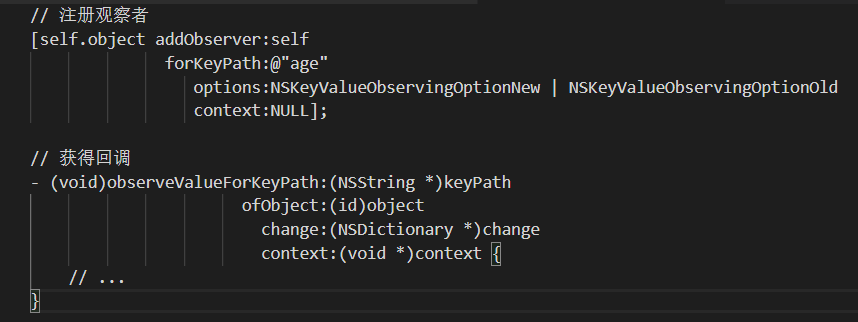
* 不同面向对象编程语言使用不同方式支持三大特性 — 封装、继承和多态；
* 多范式逐渐成为现代编程语言的主流，面向对象中复杂的设计可以被其他编程范式简化；

### 面向对象与面向对象

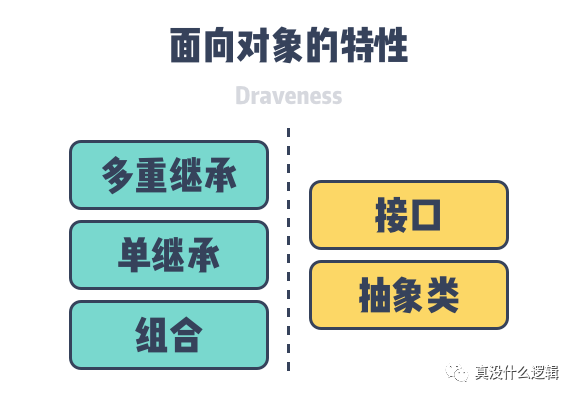
虽然《设计模式》中使用 Smalltalk 和 C++ 介绍一些场景，但是在我们实践中使用的面向对象语言可能具有不同的特性，所以同一模式会有不同的实现方案，这些方案是我们在书中完全学不到的，如果我们套用固定的模式，最终一定会得到诡异的结果。



我们以上图的观察者模式为例，所有的面向对象编程语言都可以实现上述复杂的结构，如果我们套用书中的模式确实可以得到一个标准的观察者模式，但是一些编程语言提供了一些更简洁的设计，iOS 平台上的 Objective-C 语言就使用键值观测机制（Key-Value Observing）实现观察者模式，它的实现与书中提供的观察者模式完全不同：



不仅实现模式的方式不同会导致我们在运用时会遇到诸多问题，不同面向对象语言对封装、抽象和多态的支持也不同。面向对象是一个非常广泛的概念，只要我们的编程范式是基于对象和方法的，就可以被认为支持面向对象的编程范式。

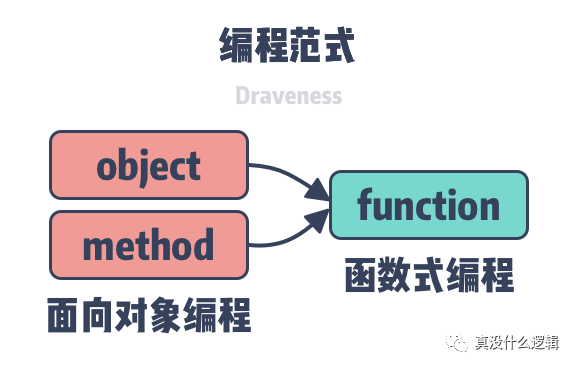


不同的编程语言可能选择支持多重继承、单继承或者组合，也可能使用不同的方式实现接口，Java8 的接口方法可以定义默认实现，而 Objective-C 中的接口（Protocol）只是一组待实现的方法签名，这些语言特性带来的差异是需要我们在研究工具时不断摸索的。

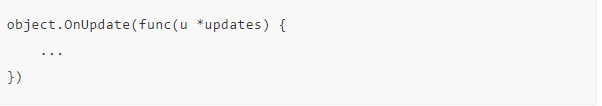
### 面向对象与多编程范式

21 世纪诞生的一些编程语言与过去的编程语言有着很大的不同，不仅新的编程语言开始接收函数式编程中的一些思想和设计，上个世纪诞生的编程语言也在吸纳不同的编程范式，函数和方法成为了语言中的一等公民，我们可以直接向函数中传递函数来简化过去复杂的类关系。

我们在这里还是以观察者模式为例，在纯粹的面向对象语言中，因为函数不能作为参数传入另一个函数，所以我们需要构造一个新的类 Observer，让这个类实现Update 方法，再将这个类的实例传入方法，当事件发生时就可以通过类上附着的方法传递消息。将函数包装到类中、初始化实例并传入函数的过程听起来都非常繁琐，而函数式编程可以简化这个问题：



在多编程范式的编程语言中，我们虽然可以使用这种传统的、啰嗦的方式实现观察者，但是直接传递函数（Procedure）是一种更加高效的做法，为我们省去了很多冗余的中间抽象，能够让程序的设计变得更加简洁：



一些读者可能会认为上述的这种方式与面向对象中的观察者模式没有**本质的不同**，这句话说的没有问题，因为从理论上来讲，几乎所有的通用编程语言都是图灵完备的，我们可以用一门语言实现另一个语言的编译器或者解释器，进而支持其他语言的特性并通过其他语言的语法来编写代码，然而这种本质论在这个讨论下没有太多意义。

不同的观察者模式虽然本质上是一样的，但是因为工具的区别，不同语言的实现也是不同的，**简单和清晰的实现往往都会取代复杂的实现，多范式的编程语言可以使用不同的特性来简化设计并减少不必要的抽象**。

## 通用的术语

设计模式作为通用的术语确实可以增加不同工程师之间的沟通效率，但是降低沟通成本的前提是**双方对同术语有着相同的并且正确的认识**，如果双方的理解有差异，反而会制造更多的困惑。我们可以将 23 种不同的设计模式分成两部分来分析，其中一部分是单例模式、抽象工厂模式这些被广泛接受并理解的模式，另一部分是迭代子模型、命令模式和解释器模式等不容易被理解的复杂模式。

从单例模式以及观察者模式的命名，我们就能猜到它们想要解决的问题，使用类似的术语也很难造成歧义，确实能够起到提高沟通效率的作用；不过，对于复杂的设计模式想要正确理解就非常困难，更不用说用来沟通了。



这里只是一个比较简单的划分，不同人对模式难易的认知有比较大的差异，这里的核心问题是书中列举的 23 种设计模式真的是**被广泛传播和认可**的么。在作者看来，设计模式虽然流传的很广泛，但是并不是所有人都清楚书中每一个模式的定义和场景，这也就丧失了提高沟通效率的作用，而常见的模式也不需要通过阅读《设计模式》来学习。

假设沟通的双方都了解不同的设计模式，那么用设计模式来描述系统中的设计确实能够提升沟通效率，让信息的接收方能够快速对设计有一个总体的概念，这能够帮助我们快速理解设计的框架，但是我们仍然需要补全对细节的认识，就像我们在前面提到的 — 细节在设计中非常重要。

**总结**

软件系统中处处都是设计，学习设计模式无法让我们成为优秀的工程师，如果我们错误的理解了这本书的目的，以为自己学到了软件设计或者面向对象设计的精髓，那就大错特错了。软件设计的能力并不是一朝一夕就能培养出来的，它需要我们不断对代码进行思考，理解可能存在的扩展点并设计合理的抽象。

作者认为《设计模式》一书的定位比较尴尬，没有经验的工程师阅读设计模式一定会觉得云里雾里，书中讲的东西都是废话；经验较少的工程师会尝试套用书中的模式，设计出一些糟糕的代码，反而误以为自己的设计恰到好处；而经验丰富的工程师在阅读这本书时会觉得没有什么作用，只是为过去的设计找到了比较合适的名字。如果有人希望作者能够推荐一些值得阅读的书籍，《设计模式》一定不在这个书单上。

我们在这里重新总结一下本文主张的观点以及支撑几个观点的相关论据，**学习设计模式与设计优秀的软件并不相关，盲目追求和套用书中的设计模式只能使项目变得更加糟糕**：

1. 设计模式都是从很多项目中总结出的通用解决方案，从具体的实现中总结出抽象的模式相对比较简单，但是想要将抽象的模式套用到场景中却非常困难；
2. 设计模式的出现本身就处在一个比较特殊的背景下，今天的编程语言大多采用了多种编程范式，很多在面向对象中需要用模式来解决问题已经被简化；哪怕是对于面向对象语言，不同语言也使用不同方法实现同一特性，我们需要注意该书的副标题以及它的局限性；
3. 设计模式成为通用术语的前提是沟通的双方对术语有着相同的认识和理解，但是很多复杂和偏门的设计模式并不会被人们熟知，单例等常见模式也不需要通过书本专门学习；

通过看书来学习软件设计几乎是一个不可能完成的任务，作者认为**学习如何为程序编写单元测试对学习系统设计极其重要**，提升项目单元测试覆盖率的过程会让我们思考如何写出更利于测试的代码，虽然软件工程中没有银弹，但是单元测试不是银弹可能也所差无几了。

Design is the art of arranging code to work today, and be changeable forever. --Sandi Metz

我们在学习设计的过程中如果不看《设计模式》这本书可能会少走一些弯路，理解了一些抽象出来的定式可能会限制我们的思考，想要学习软件设计就要**去真正优秀的开源项目学习顶层设计到底层实现并在项目中不断实践**，在最后我们会发现系统中到处都是设计和模式，而软件开发中没有圣杯。