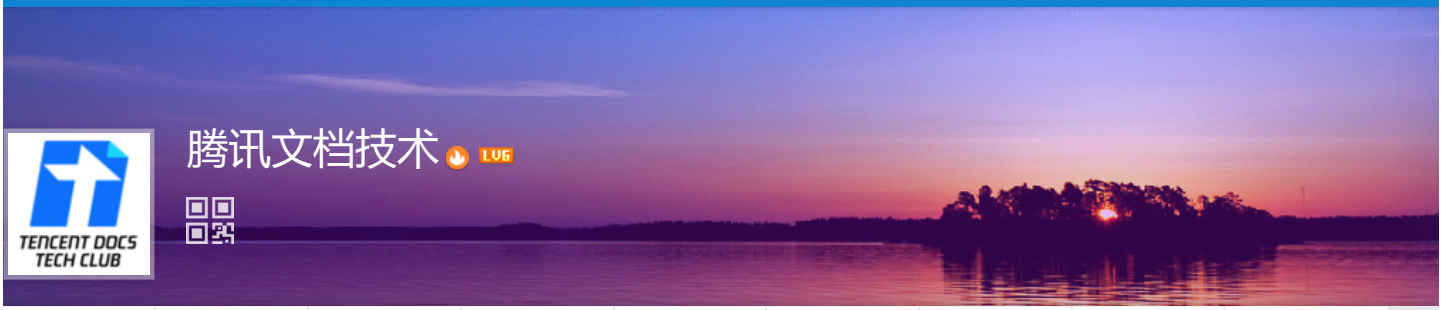


大型web前端架构设计-面向抽象编程入门

svenzeng 2020年12月25日 17:30

| 导语 面向抽象编程，是构建一个大型系统非常重要的参考原则。 但对于许多前端同学来说，对面向抽象编程的理解说不上很深刻。大部分同学的习惯是 拿到需求单和设计稿之后就开始编写UI界面，UI里哪个按钮需要调哪些方法，接下来再编写这些方法，很少去考虑复用性。当某天发生需求变更时，才发现目前的代码很难适应这些变更，只能重写。日复一日，如此循环。

面向具体实现编程：

当第一次看到“将抽象和具体实现分开”这句话的时候，可能很难明白它表达的是什么意思。什么是抽象，

假设我们正在开发一个类似“模拟人生”的程序，并且创造了小明，为了让他的每一天都有规律的生活下去，于是给他的核心程序里设置了如下逻辑：

```
1、8点起床
2、9点吃面包
3、17点打篮球
```

过了一个月，小明厌倦了一成不变的重复生活，某天早上起来之后他突然想吃薯片，而不是面包。等到傍晚的时候他想去踢足球，而不是继续打篮球，于是我们只好修改源代码：

```
1、8点起床
2、9点吃面包 -> 9点吃薯片
3、17点打篮球 -> 17点踢足球
```

又过了一段时间，小明希望周3和周5踢足球，星期天打羽毛球，这时候为了满足需求，我们的程序里可能会被加进很多if、else语句。

为了满足需求的变换，跟现实世界很相似，我们需要深入核心源代码，做大量改动。现在再想想自己的代码里，是不是有很多似曾相识的场景？

这就是一个面向具体实现编程的例子，在这里，吃面包、吃薯片、打篮球、踢足球这些动作都属于具体实现，映射到程序中，它们就是一个模块、一个类，或者一个函数，包含着一些具体的代码，去负责某件具体的事情。

一旦我们想在代码中更改这些实现，必然需要被迫深入和修改核心源代码。当需求发生变更

关于作者

svenzeng(曾探)
PCG\社交协作产品部\前端开发七组...

作者文章

时，一方面，如果核心代码中存在各种各样的大量具体实现，想去全部重写这些具体实现的工作量是巨大的，另一方面，修改代码总是会带来未知的风险，当模块间的联系千丝万缕时，修改任何一个模块都得小心翼翼，否则很可能发生改好1个bug，多出3个bug的情况。

抽取出共同特性

抽象的意思是：从一些事物中抽取出共同的、本质性的特征。

如果我们总是针对具体实现去编写代码，就像上面的例子，要么写死9点吃面包，要么写死9点吃薯片。这样一来，在业务发展和系统迭代过程中，系统就会变得僵硬和修改困难。产品需求总是多变的，我们需要在多变的环境里，尽量让核心源代码保持稳定和不用修改。

方法就是需要抽取出“9点吃面包”和“9点吃薯片”的通用特性，这里可以用“9点吃早餐”来表示这个通用特性。同理，我们抽取出“17点打篮球”和“17点踢足球”的通用特性，用“17点做运动”来代替它们。然后让这段核心源代码去依赖这些“抽象出来的通用特性”，而不再是依赖到底是“吃面包”还是“吃早餐”这种“具体实现”。

我们将这段代码写成：

```
1、 8点起床
2、 9点吃早餐
3、17点做运动
```

这样一来，这段核心源代码就变得相对稳定多了，不管以后小明早上想吃什么，都无需再改动这段代码，只要在后期，由外层程序将“吃早餐”还是“吃薯片”注入进来即可。

真实示例

刚才是一个虚拟的例子，现在看一段真实的代码，这段代码依然很简单，但可以很好的说明抽象的好处。

在某段核心业务代码里，需要利用localstorge储存一些用户的操作信息，代码很快就写好了：

```
import ‘localstorge’ from 'localstorge';

class User{
    save(){
        localstorge.save('xxx');
    }
}

const user = new User();
user.save();
```

这段代码本来工作的很好，但是有一天，我们发现用户信息相关数据量太大，超过了localstorge的储存容量。这时候我们想到了indexdb，似乎用indexdb来存储会更加合理一些。

现在我们需要将localstorge换成indexdb，于是不得不深入User类，将调用localstorge的地方修改为调用indexdb。似乎又回到了熟悉的场景，我们发现程序里，在许多核心业务逻辑深处，不只一个，而是有成百上千个地方调用了localstorge，这个简单的修改都成了灾难。

所以，我们依然需要提取出localstorge和indexdb的共同抽象部分，很显然，localstorge和indexdb的共同抽象部分，就是都会向它的消费者提供一个save方法。作为它的消费者，也就是业务中的这些核心逻辑代码，并不关心它到底是localstorge还是indexdb，这件事情完全可

是业务中的这些核心逻辑代码，并不关心它到底是localstorge还是Indexdb，这件事情完全可以等到程序后期再由更外层的其他代码来决定。

我们可以申明一个拥有save方法的接口：

```
interface DB{
   save(): void;
}
```

然后让核心业务模块 User 仅仅依赖这个接口：

```
import DB from 'DB';

class User{
    constructor(
        private db: DB
   ){

    }
    save(){
        this.db.save('xxx');
    }
}
```

接着让Localstorge和Indexdb分别实现DB接口：

```
class Localstorge implements DB{
    save(str:string){
        ...//do something
    }
}

class Indexdb implements DB{
    save(str:string){
        ...//do something
    }
}

const user = new User( new Localstorge() );
//or
const user = new User( new Indexdb() );

userInfo.save();
```

这样一来，User模块从依赖Localstorge或者Indexdb这些具体实现，变成了依赖DB接口，User模块成了一个稳定的模块，不管以后我们到底是用Localstorge还是用Indexdb，User模块都不会被迫随之进行改动。

让修改远离核心源代码

可能有些同学会有疑问，虽然我们不用再修改User模块，但还是需要去选择到底是用Localstorge还是用Indexdb，我们总得在某个地方改动代码把，这和去改动User模块的代码有

Localstorge还是用Indexdb，我们总得在某个地方改动代码把，这和去改动User模块的代码有什么区别呢?

实际上，我们说的面向抽象编程，通常是针对核心业务模块而言的。User模块是属于我们的核心业务逻辑，我们希望它是尽量稳定的。不想仅仅因为选择使用Localstorge还是Indexdb这种事情就得去改动User模块。因为User模块这些核心业务逻辑一旦被不小心改坏了，就会影响到千千万万个依赖它的外层模块。

如果User模块现在依赖的是DB接口，那它被改动的可能性就变小了很多。不管以后的本地存储怎么发展，只要它们还是对外提供的是save功能，那User模块就不会因为本地存储的变化而发生改变。

相对具体行为而言，接口总是相对稳定的，因为接口一旦要修改，意味着具体实现也要随之修改。而反之当具体行为被修改时，接口通常是不用改动的。

至于选择到底是用Localstorge还是用Indexdb这件事情放在那里做，有很多种实现方式，通常我们会把它放在更容易被修改的地方，也就是远离核心业务逻辑的外层模块，举几个例子:

```
* 在main函数或者其他外层模块中生成Localstorge或者Indexdb对象，在User对象被创建时作为参数
* 用工厂方法创建Localstorge或者Indexdb
* 用依赖注入的容器来绑定DB接口和它具体实现之间的映射
```

内层、外层和单向依赖关系

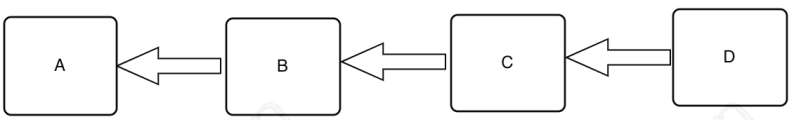
将系统分层，就像建筑师会将大厦分为很多层，每层有特有的设计和功能，这是构建大型系统架构的基础。除了过时的mvc分层架构方式外，目前常用的分层方式有洋葱架构（整洁架构）、DDD（领域驱动设计）架构、六边形架构（端口-适配器架构）等，这里不会详细介绍每个分层模式，但不管是洋葱架构、DDD架构、还是六边形架构，它们的层与层之间，都会被相对而动态地区分为外层和内层。

前面我们也提过好几次内层和外层的概念（大部分书里称为高层和低层），那么在实际业务中，哪些模块会对应内层，而哪些模块应该被放在外层，到底由什么规律来决定呢?

先观察下自然届，地球围绕着太阳转，我们认为太阳是内层，地球是外层。眼睛接收光线后通过大脑成像，我们认为大脑是内层，眼睛是外层。当然这里的内层和外层不是由物理位置决定的，而是基于模块的稳定性，即越稳定越难修改的模块应该被放在越内层，而越易变越可能发生修改的模块应该被放在越外层。就像用积木搭建房子时，我们需要把最坚固的积木搭在下面。

这样的规则设置是很有意义的，因为一个成熟的分层系统都会严格遵守单向依赖关系。

我们看下面这个图:



假设系统中被分为了A、B、C、D这4层，那么A是相对的最内层，外层依次是B、C、D。在一个严格单向依赖的系统中，依赖关系总是只能从外层指向内层。

这是因为，如果最内层的A模块被修改，则依赖A模块的B、C、D模块都会分别受到牵连。在静态类型语言中，这些模块因为A模块的改动都要重新进行编译，而如果它们引用了A模块的

某个变量或者调用了A模块中的某个方法，那么它们很可能因为A模块的修改而需要随之修改。所以我们希望A模块是最稳定的，它最好永远不要发生修改。

但如果外层的模块被修改呢？比如D模块被修改之后，因为它处在最外层，没有其他模块依赖它，它影响的仅仅是自己而已，A、B、C模块都不需要担心它们收到任何影响，所以，当外层模块被修改时，对系统产生的破坏性相对是比较小的。

如果从一开始就把容易变化，经常跟着产品需求变更的模块放在靠近内层，那意味着我们经常会因为这些模块的改动，不得不去跟着调整或者测试系统中依赖它的其他模块。

可以设想一下，造物者也许也是基于单向依赖原则来设置宇宙和自然界的，比如行星依赖恒星，没有地球并不会对太阳造成太大影响，而如果失去了太阳，地球自然也不存在。眼睛依赖大脑，大脑坏了眼睛自然失去了作用，但眼睛坏了大脑的其他功能还能使用。看起来地球只是太阳的一个插件，而眼睛只是大脑的一个插件。

回到具体的业务开发，核心业务逻辑一般是相对稳定的，而越接近用户输入输出的地方（越接近产品经理和设计师，比如UI界面），则越不稳定。比如开发一个股票交易软件，股票交易的核心规则是很少发生变化的，但系统的界面长成什么样子很容易发生变化。所以我们通常会把核心业务逻辑放在内层，而把接近用户输入输出的模块放在外层。

在腾讯文档业务中，核心业务逻辑指的就是将用户输入数据通过一定的规则进行计算，转换成文档数据。这些转换规则和具体计算过程是腾讯文档的核心业务逻辑，它们是非常稳定的，从微软office到谷歌文档到腾讯文档，30多年了也没有太多变化，它们理应被放在系统的内层。另一方面，不管这些核心业务逻辑跑在浏览器、终端或者是node端，它们也都不应该变化。而网络层、存储层，离线层、用户界面这些是易变的，在终端环境里，终端用户界面层和web层的实现就完全不一样。在node端，存储层或许可以直接从系统中剔除掉，因为在node端，我们只需要利用核心业务逻辑模块对函数进行一些计算。同理，在单元测试或者集成测试的时候，离线层和存储层可能都是不需要的。在这些易变的情况下，我们需要把非核心业务逻辑都放在外层，方便它们被随时修改或替换。

所以，遵守单向依赖原则能极大提高系统稳定性，减少需求变更时对系统的破坏性。我们在设计各个模块的时候，要将相当多的时间花在设计层级、模块的切分，以及层级、模块之间的依赖关系上，我们常说“分而治之”，“分”就是指层级、模块、类等如何切分，“治”就是指如何将分好的层级、模块、类合理的联系起来。这些设计比具体的编码细节工作要更加重要。

依赖反转原则

依赖反转原则的核心思想是：内层模块不应该依赖外层模块,它们都应该依赖于抽象。

尽管我们会花很多时间去考虑哪些模块分别放到内层和外层，尽量保证它们处于单向依赖关系。但在实际开发中，总还是有不少内层模块需要依赖外层模块的场景。

比如在Localstorge和Indexdb的例子里，User模块作为内层的核心业务逻辑，却依赖了外层易变的Localstorge和Indexdb模块，导致User模块变得不稳定。

```
import ‘localstorge’ from 'localstorge';

class User{
    save(){
        localstorge.save('xxx');
    }
}

const user = new User();
```

```
user.save();
```

缺图

为了解决User模块的稳定性问题，我们引入了DB抽象接口，这个接口是相对稳定的，User模块改为去依赖DB抽象接口，从而让User变成一个稳定的模块。

```
Interface DB{
   save(): void;
}
```

然后让核心业务模块 User 仅仅依赖这个接口：

```
import DB from 'DB';

class User{
    constructor(
        private db: DB
   ){

    }
    save(){
        this.db.save('xxx');
    }
}
```

接着让Localstorge和Indexdb分别实现DB接口：

```
class Localstorge implements DB{
    save(str:string){
        ...//do something
    }
}
```

依赖关系变成：
缺图

User -> DB <- Localstorge

在图1和图2看来，User模块不再显式的依赖Localstorge，而是依赖稳定的DB接口，DB到底是什么，会在程序后期，由其他外层模块将Localstorge或者Indexdb注入进来，这里的依赖关系看起来被反转了，这种方式被称为“依赖反转”。

找到变化，并将其抽象和封装出来

我们的主题“面向抽象编程”，很多时候其实就是指的“面向接口编程”，面向抽象编程站在系统设计的更宏观角度，指导我们如何构建一个松散的低耦合系统，而面向接口编程则告诉我们具体实现方法。依赖倒置原则告诉我们如何通过“面向接口编程”，让依赖关系总是从外到内，指向系统中更稳定的模块。

知易行难，面向抽象编程虽然概念上不难理解，但在真实实施中却总是不太容易。哪些模块应该被抽象，哪些依赖应该被倒转，系统中引入多少抽象层是合理的，这些问题都没有标准答案。

我们在接到一个需求，对其进行模块设计时，要先分析这个模块以后有没有可能随着需求变更被替换，或是被大范围修改重构？当我们发现可能会存在变化之后，就需要将这些变化封装起来，让依赖它的模块去依赖这些抽象。

比如上面例子中的Localstorge和indexdb，有经验的程序会很容易想到它们是有可能需要被互相替换的，所以它们最好一开始就被设计为抽象的。

同理，我们的数据库也可能产生变化，也许今天使用的是mysql，但明年可能会替换为oracle，那么我们的应用程序里就不应该强依赖mysql或者oracle，而是要让它们依赖mysql和oracle的公共抽象。

再比如，我们经常会在程序中使用ajax来传输用户输入数据，但有一天可能会想将ajax替换为websocket的请求，那么核心业务逻辑也应该去依赖ajax和websocket的公共抽象。

封装变化与设计模式

实际上常见的23种设计模块，都是从封装变化的角度被总结出来的。拿创建型模式来说，要创建一个对象，是一种抽象行为，而具体创建什么对象则是可以变化的，创建型模式的目的就是封装创建对象的变化。而结构型模式封装的是对象之间的组合关系。行为型模式封装的是对象的行为变化。

比如工厂模式，通过将创建对象的变化封装在工厂里，让核心业务不需要依赖具体的实现类，也不需要了解过多的实现细节。当创建的对象有变化的时候，我们只需改动工厂的实现就可以，对核心业务逻辑没有造成影响。

比如模块方法模式，封装的是执行流程顺序，子类会继承父类的模版函数，并按照父类设置好的流程规则执行下去，具体的函数实现细节，则由子类自己来负责实现。

通过封装变化的方式，可以把系统中稳定不变的部分和容易变化的部分隔离开来。在系统的演变过程中，只需要替换或者修改那些容易变化的部分，如果这些部分是已经封装好的，替换起来也相对容易。这可以最大程度地保证程序的稳定性。

避免过度抽象

虽然抽象提高了程序的扩展性和灵活性，但抽象也引入了额外的间接层，带来了额外的复杂度。本来一个模块依赖另外一个模块，这种依赖关系是最简单直接的，但我们在中间每增加了一个抽象层，就意味着需要一直关注和维护这个抽象层。这些抽象层被加入系统中，必然会增加系统的层次和复杂度。

如果我们判断某些模块相对稳定，很长时间内都不会发生变化，那么没必要一开始就让它们成为抽象。

比如java中的String类，它非常稳定，所以并没有对String做什么抽象。

比如一些工具方法，类似utils.getCookie()，我很难想象5年内有什么东西会代替cookie，所以我更喜欢直接写getCookie。

比如腾讯文档excel的数据model，它属于内核中的内核，像整个身体中的骨骼和经脉，已经融入到了各个应用逻辑中，它被替换的可能性非常小，难度也非常大，不亚于重写一个腾讯文档excel，所以也没有必要对model做过度抽象。

结语

面向抽象编程有2个最大好处。

一方面，面向抽象编程可以将系统中经常变化的部分封装在抽象里，保持核心模块的稳定。

另一方面，面向抽象编程可以让核心模块开发者从非核心模块的实现细节中解放出来，将这些非核心模块的实现细节留在后期或者留给其他人。

这篇文章讨论的实际主要偏重第一点，即封装变化。封装变化是构建一个低耦合松散系统的关键。

这篇文章，作为面向抽象编程的入门，希望能帮助一些同学认识面向抽象编程的好处，以及掌握一些基础的面向抽象编程的方法。

团队持续招人中，欢迎联系

最后更新于 2021-01-22 11:06

标签：腾讯文档(1) 设计模式(2) 依赖反转(1)

大家评论

shiangzhang　2020-12-25 17:53:56

顶，好文

顶　回复

freedeng　2020-12-25 17:55:29

赶紧学习，缩小差距

顶 回复

linelin 2020-12-25 18:02:54

👍 前排围观大佬

顶 回复

batiliu 2020-12-25 18:06:33

围观前端大佬

顶 回复

kuanggu 2020-12-25 18:21:33

赶紧学习，缩小差距

顶 回复

shaolongyin 2020-12-25 18:37:44

赶紧学习，缩小差距

顶 回复

aricxu 2020-12-25 19:10:20

赶紧学习，缩小差距

顶 回复

heyli 2020-12-25 22:31:24

tql

顶 回复

dickyduan 2020-12-25 23:31:34

mark下慢慢消化

顶 回复

kellanzhang 2020-12-26 00:08:34

tql

顶 回复

feifeiyuan 2020-12-26 11:44:03

赶紧学习，缩小差距

顶 回复

jurrychen 2020-12-26 15:29:25

最开始能抽离不同层次就很好了。在需求变更之后侵入性的修改原有逻辑。很容易出bug，并且不好验证。

顶 回复

jsonsun 2020-12-28 09:27:38

tql

顶 回复

devinshi 2020-12-28 13:08:29

学习

顶 回复

fancyzhong 2020-12-28 13:34:06

👍，适度的抽象，不断的沉淀稳定的部分到内层，抽象异变的部分到外层

顶 回复

fulali 2020-12-28 14:12:08

以前我写的react都是oop的，所有组件都继承自一个基类，在状态管理和异常控制很有效果，但是发现可以移植性比较差，哈哈哈。

顶 回复 (2)

svenzeng (楼主) 2020-12-28 14:35:12

可以少用继承, 继承将2个类绑定的太紧了

回复

advjiang 2020-12-29 14:15:40

react可以尝试用纯函数

回复

mingxiao 2020-12-28 17:10:53

将抽象到业务，从内稳定到外可变的角度阐述更加容易理解和操作，学习了 

顶 回复

monkeytao 2020-12-28 17:22:01

赶紧学习，缩小差距

顶 回复

willyu 2020-12-28 19:00:32

学习

顶 回复

noopyxu 2020-12-28 19:21:41

Localstore 写成了 Localstory 

顶 回复 (3)

svenzeng (楼主) 2020-12-28 19:23:20

已改，感谢

回复

noopyxu 2020-12-28 19:24:11

自然届->自然界

回复

zainchen 2020-12-29 10:05:40

文中 Localstorge 应该是 LocalStorage，Indexdb 应该是 IndexedDB

回复

initialwu 2020-12-29 11:37:11

DDD 里面貌似用的更多的是 repository 来表达 DB 这个概念

 当然这只是个习惯，本质是一样的

顶 回复 (1)

svenzeng (楼主) 2020-12-29 11:38:36

嗯嗯，只是个简单例子

回复

initialwu 2020-12-29 11:40:07

有本类似主题的书，个人推荐，很值得一读

https://book.douban.com/subject/26975203/

顶 回复 (1)

svenzeng (楼主) 2020-12-29 11:43:32

我去看下，感谢

回复

kevinzqqin 2020-12-29 14:48:02

学习了。

顶 回复

edwarwang 2020-12-29 14:48:49

涨知识了

顶 回复

classyuan 2020-12-29 15:51:15

给你call一波666

顶 回复

切换到更多功能

发表评论

[318/333/399 ms]