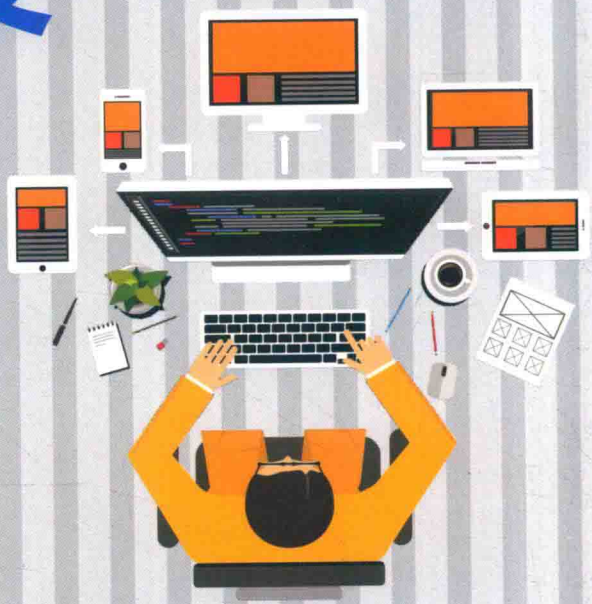
Live
软件开发面面谈

潘　俊◎编著



现实中软件开发会遇到许多具体的问题：

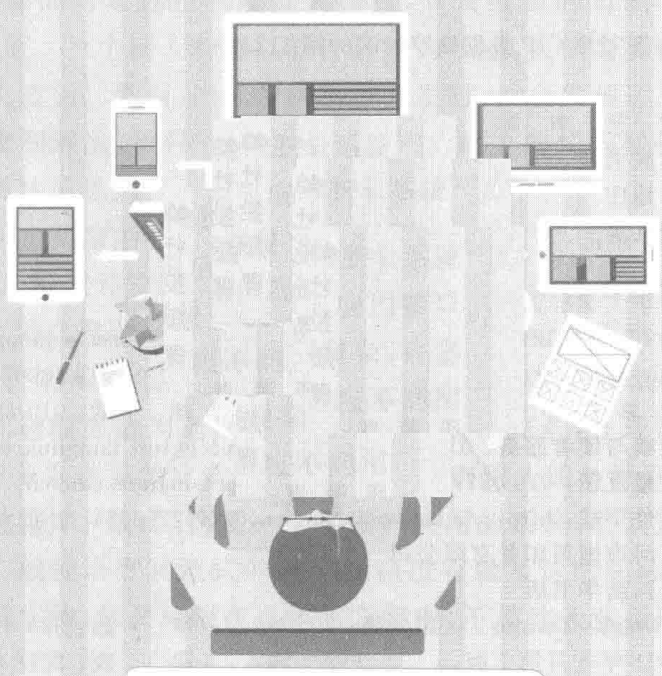
1.如何消除依赖?
2.怎样进行事件驱动编程?
3.如何在迥异的环境中实现MVC架构?
4.怎样在不同的Web开发框架之间选择?
5.文档型数据库与关系型数据库相比有哪些优缺点?
6.如何构建合适的存取控制?

一线软件开发人员答疑解惑，深入思考软件开发的方方面面，追本溯源，融会贯通，有益于在纷繁多变的技术浪潮中看清本质、把握方向，并从行业快速跳动的脉搏中读出一些共性和规律。

清華大學出版社

Live
软件开发面面谈

潘　俊◎编著



清華大學出版社
北　京

内 容 简 介

现实的软件开发会遇到许多具体的问题，例如，如何消除依赖？怎样进行事件驱动编程？如何在迥异的环境中实现 MVC 架构？怎样在不同的 Web 开发框架之间选择？文档型数据库与关系型数据库相比有哪些优缺点？如何构建合适的存取控制？对这些问题，简单的答案、现成的选择、枯燥的代码很多，但是从问题的源头和本质出发，深入全面的分析却很少。本书就软件开发中带有普遍性的重要方面，内容由浅入深地逐渐展开，力图使读者对软件开发实践产生由点及面、融会贯通的理解。

图书在版编目(CIP)数据

Live 软件开发面面谈/潘俊编著. —北京：清华大学出版社，2018
ISBN 978-7-302-50156-5

Ⅰ. ①L…　Ⅱ. ①潘…　Ⅲ. ①移动终端－应用程序－程序设计　Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字(2018)第 112372 号

责任编辑：黄　芝　李　晔
封面设计：迷底书装
责任校对：李建庄
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954

印 装 者：三河市国英印务有限公司
经　　销：全国新华书店
开　　本：170mm×230mm　　**印　　张**：20　　**字　　数**：338 千字
版　　次：2018 年 8 月第 1 版　　**印　　次**：2018 年 8 月第 1 次印刷
印　　数：1～1500
定　　价：59.00 元

产品编号：072748-01

自序

PREFACE

开发软件离不开编写代码，但仅仅具备编程的技能也还不足以胜任开发软件的工作。这就好比一个人会烧砖、砌墙，但要造一间可供人居住的屋子，他还得了解屋子的结构、不同房间的功能、水电管线的敷设、墙面地面的装修等方面的知识。对软件开发人员来说，编程领域的知识往往是最受关注的，它们确实也可以分为多个层次：编程语言本身的知识（如 C、Java），编程范式和思想，面向对象编程和函数式编程，开发框架的知识（如 Spring、AngularJS），等等。一个新人若想以软件开发为职业，大概需要阅读的范围就会集中在以上方面。然而，当他开始项目开发时，就会发现还有许多实际的问题需要考虑和解决，软件开发并不像编程教材上的代码样例和习题那样专注于某个算法或思想。

不妨考虑一个典型的业务系统，它是一个图形用户界面的程序，因而需要采用某种 GUI 框架开发界面；用户在界面上的操作通过事件机制调用相应的处理程序；用户界面、事件处理程序和体现需求的业务逻辑必须组成某种合理的结构，否则系统会随着功能的增加迅速变得难以理解和维护；系统越大，组件越多，越需要适当地保持它们之间的依赖关系，合理地应用接口是关键；这个业务系统显然比所有数据都来自即时输入的计算器复杂，许多信息要往返于数据库；最后，这是一个多用户使用的系统，必须适应不同用户的权限需求。编程语言和范式的理论知识没有触及这些实际的问题，开发框架虽然涉及实践，却又局限在具体的方案中，不易让人获得对知识的一般理解。

软件开发实践中遇到的各个方面的问题往往缺乏系统的理论，程序员凭着各自的理解动手，或者知其然而不知其所以然，或者每个人的所以然有出入甚至矛盾。例如，针对接口编程就是尽量多用接口吗？事件驱动编程

的本质是什么？怎么样算是应用了 MVC 架构？极简主义就是越简单越好吗？文档型数据库和关系型数据库的优劣各体现在什么地方？基于角色的存取控制系统是如何理解权限的？在主流的软件开发理念之外能否另辟蹊径？客户端和浏览器之间的竞争究竟意味着什么？对这类实践中涉及的概念和遇到的问题，如果追根溯源，多思考一些是什么、为什么和怎么做，达到融会贯通的理解，既对实际开发有帮助，又有益于在纷繁多变的技术浪潮中看清技术的本质、把握解决问题的方向。

本书从以上思路出发，逐个讨论软件开发实践中的重要主题。第 1 章辨析对象间的依赖和针对接口编程。第 2 章讨论事件驱动编程的方方面面。第 3 章分析 MVC 架构的思想和实现。第 4 章比较图形用户界面的一些相关或对立的思想和技术，并介绍极简主义潮流。第 5 章分析热门的文档型数据库，并和关系型数据库做对比。第 6 章讨论存取控制的各个环节，分析基于角色的和基于属性的存取控制的优缺点。第 7 章介绍快速的 Lotus Notes 程序开发。第 8 章探讨软件的兴衰和客户端的潮流。顺序上靠前的章节内容具有一般性，不会依赖其后的部分，靠后的章节有可能应用前文的知识。编写风格上每章力图从主题的源头和本质入手，遵循逻辑层层展开，尽量全面地遍历主题涉及的方方面面。书中代码为正在讨论的理念和问题服务，只是示意性地勾勒出核心的部分，无关和繁冗的部分被省略。

野人献曝，未免贻笑大方；愚者千虑，或有一得可鉴。

是为序。

作者

2018 年 5 月

目录

CONTENTS

第1章

接　口

在面向对象编程中，我们将问题拆分成一个个对象来实现，每个对象有其负责的功能，多个对象合作才能形成一个有用的系统。合作在代码中就表现为对象之间的引用和方法调用。调用者与被调用者的关系称为依赖。依赖关系意味着被调用者的变化可能影响和破坏调用者原本正常的运行。当系统变得越来越大，对象越来越多，牵涉方越来越广，持续的时间越长时，设计者就希望这样牵一发而动全身的影响尽可能地小。换句话说，就是希望能消除对象之间的依赖。调用者既要调用被调用者的方法，又不能产生对它的依赖，解决方法便是运用接口。

接口的理念在编程中由来已久，在 Java、C# 等主流语言中更是引入了原生的 Interface 结构，类库中也有大量现成的接口。然而单纯地使用、甚至定义接口，并不能达到消除依赖的目的。广为提倡的尽量使用接口编程，有什么好处？真正能消除依赖的针对接口编程又如何实现？它与常用的工厂模式、服务定位器模式和依赖注入有什么关系？最后，什么时候才有必要针对接口编程？在本章讨论这些问题的过程中，接口、依赖、若干设计模式、配置文件、惯例、元数据等概念的含义将得到深入的挖掘和思考。

1.1 使用接口编程

先来看看在用Java、C#这样的面向对象语言编程时，经常被提倡的尽量使用接口的理念。在用继承基类和实现接口构建的类型层次体系中，越往上的类型越一般和抽象，越往下的类型越具体和多功能。在定义变量时，无论是类字段、方法变量，还是方法的参数和返回值，都尽可能使用抽象的类型。例如Java语言只支持单个基类，类型的大量抽象继承均以接口的方式体现，导致在一个类的层次体系的高层，接口往往比类多，所以尽可能使用的抽象类型就以接口居多，这也就是所谓的使用接口编程。例如下面的C#代码。

```
//Starrow.IdeaDemo.UseInterfaceDemo
using System.Collections;

namespace Starrow.IdeaDemo
{
    public class UseInterfaceDemo
    {
        Hashtable DeclareAndReturnConcreteClass()
        {
            Hashtable hashtable = new Hashtable();
            //...
            return hashtable;
        }
        void PassConcreteClass(Hashtable hashtable)
        {
            hashtable.Add("a", 1);
            //...
        }
        IDictionary DeclareAndReturnInterface()
        {
            IDictionary dict = new Hashtable();
            //...
            return dict;
        }
        void PassInterface(IDictionary dict)
        {
```

```
            dict.Add("a", 1);
            //...
        }
    }
}
//End of Starrow.IdeaDemo.UseInterfaceDemo
```

在以 ConcreteClass 结尾的两个方法中，使用的是具体的类型 Hashtable；而在以 Interface 结尾的两个方法中，使用的是抽象的接口 IDictionary。使用接口的好处是，对于 DeclareAndReturnInterface 方法，将来如果基于业务逻辑或性能的考虑，觉得应该采用另一个更合适的实现 IDictionary 的类，如 SortedList，只需要把 dict 变量初始化成一个 SortedList，后面的代码和返回的类型丝毫不受影响，因而对调用方是透明的；对于 PassInterface 方法，能够接受任何实现了 IDictionary 接口的参数，调用方传入的具体类型发生变动不会影响该方法的运作。简言之，就是使用的类型越一般，代码的应用范围越广，适应性越好。当然，应该是尽可能一般，而不是无条件的最一般。所谓可能，就是指该类型的接口能够满足使用者的需求。例如，在上面的例子中不能使用比 IDictionary 更一般的接口 ICollection，因为它没有 Add 方法。

1.2 依赖反转原则

在模块化或组件化软件设计中，不同模块间保持松耦合。每个模块都定义有清晰的接口，模块间的调用都通过和限于接口，只要接口不变，模块内对接口的实现可以自由修改和演化。这个原则就是著名的“针对接口编程，而不是针对实现”(Program to an interface, not an implementation)。针对接口编程比使用接口编程更具雄心、野心和企图心(More ambitious)，它要更彻底地消除依赖关系。1.1 节的 DeclareAndReturnInterface 方法，虽然 dict 变量声明为 IDictionary 接口，但因为初始化为 Hashtable 类型，该方法所在的类 UseInterfaceDemo 仍然依赖 Hashtable 类。要消除此依赖，必须做到模块之间完全以接口交流。

为了分析依赖和接口的关系，下面介绍最简单的两个模块的情况。应用模块需要使用工具模块的功能。按照传统的设计，应用模块直接引用工具模块。两者的依赖关系如图 1.1 所示。

图 1.1 传统设计中应用模块和工具模块的关系

这种紧密的耦合限制了应用模块的可用性，它只能和特定的工具模块一同工作，当有更好的或实现其他功能的工具模块时，它也不能替换以利用。为了打破这个约束，可以将应用模块需要的工具模块的功能抽象成一个工具接口，应用模块通过这个接口来使用工具模块，工具模块只要实现这个接口，就能被自由替换。此时假如将工具接口置于应用模块内，因为工具模块要引用该接口，两模块间的依赖关系发生了奇妙的倒转，如图 1.2 所示。

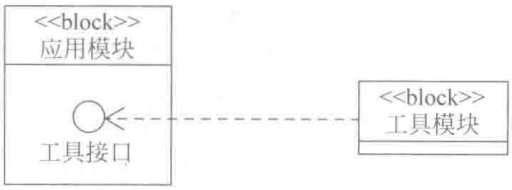


图 1.2 工具接口置于应用模块之内时两模块之间倒转的关系

每个工具模块在开发时都要引用应用模块，当然是不理想的，尤其作为工具模块根本无法知道要使用它们的应用模块可能是什么样的，所以这样反过来的依赖也必须消除。办法就是令工具接口脱离应用模块，成为一个新的独立模块。这样应用模块和工具模块都仅仅引用这个抽象的接口模块。三者的关系如图 1.3 所示。

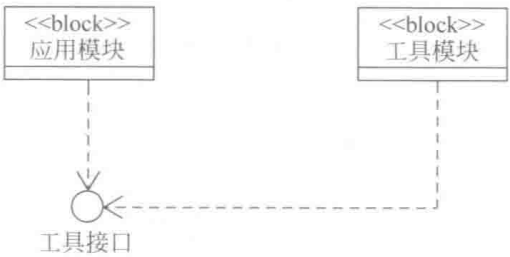


图 1.3 工具接口独立于应用模块和工具模块时三者的关系

这个最终方案可以用一句话来概括：调用模块不应该依赖被调用模块，两者应该依赖抽象出的接口。这个原则被称为依赖反转（dependency inversion），虽然它不是顾名思义地指上面第二种情况。

1.3 如何实现

关于针对接口编程，前面谈了是什么(What)和为什么(Why)，下面介绍怎么做(How)。下面给出一个媒体播放器的实例，实现利用各种编解码器来播放媒体文件的功能。第一个版本的代码如下。

```
//媒体播放器
//Starrow.IdeaDemo.Dependency.MediaPlayerV1
using System.IO;

namespace Starrow.IdeaDemo.Dependency
{
    public class MediaPlayerV1
    {
        public void Play(FileInfo file)
        {
            Stream dataStream;
            string ext = file.Extension;
            switch (ext)
            {
                case "mp3":
                    MP3CodecV1 mp3Codec = new MP3CodecV1();
                    dataStream = mp3Codec.Decode(file);
                    PlayStream(dataStream);
                    break;
                case "mp4":
                    MP4CodecV1 mp4Codec = new MP4CodecV1();
                    dataStream = mp4Codec.Read(file);
                    PlayStream(dataStream);
                    break;
            }
        }

        private void PlayStream(Stream dataStream)
        {
            //...
        }
    }
}
```

```
//End of Starrow.IdeaDemo.Dependency.MediaPlayerV1

// MP3 编解码器
// Starrow.IdeaDemo.Dependency.MP3CodecV1
using System.IO;

namespace Starrow.IdeaDemo.Dependency
{
    public class MP3CodecV1
    {
        public Stream Decode(FileInfo file)
        {
            MemoryStream stream = new MemoryStream();
            //...
            return stream;
        }
    }
}
//End of Starrow.IdeaDemo.Dependency.MP3CodecV1

// MP4 编解码器
// Starrow.IdeaDemo.Dependency.MP4CodecV1
using System.IO;

namespace Starrow.IdeaDemo.Dependency
{
    public class MP4CodecV1
    {
        public Stream Read(FileInfo file)
        {
            MemoryStream stream = new MemoryStream();
            //...
            return stream;
        }
    }
}
//End of Starrow.IdeaDemo.Dependency.MP4CodecV1
```

由于每种编解码器的解码方法的名称均不一样，所以播放器不仅引用不同编解码器的实例，还要了解其 API 的差异，调用正确的方法，这样开发人员当然太痛苦了。首先想到的改进方法就是为编解码器制定统一的接口，之后播放器使用编解码器时就方便得多，于是有了第二个版本的代码。