[1. 耦合的形式 2](#_Toc462845323)

[1-1. 不同层次的耦合度，代码的复杂度是不同的 2](#_Toc462845324)

[1-2. 架构有利于长期，短期肯定是相对麻烦的 3](#_Toc462845325)

[1-2-1. 没有银弹，只是最优选择 3](#_Toc462845326)

[2. 设计模式 3](#_Toc462845327)

[2-1. 为什么我们的程序中对设计模式留意的少 3](#_Toc462845328)

[3. 程序和框架 4](#_Toc462845329)

[3-1. 看网上的Demo 程序的不足 4](#_Toc462845330)

[4. MVC MVP 和MVVM 4](#_Toc462845331)

# 耦合的形式

在面向对象编程时，可以设计高内聚的内。但是，类于类之间又必须要交互才能共同完成一个业务逻辑，也就是类之间必须要耦合。

类之间的耦合两种形式：

* 继承
* 使用

使用又可分为几种形式

* 单向使用： 这时被使用者不知道使用者的存在，所以只是单向的耦合。使用者的变动不会印象被使用者。
* 双向使用：彼此互相依赖，耦合度比单向使用要高

在代码实现上，又有几种方式：

* 使用者里面有被使用者的类的引用
  + 被使用者在使用者里面New
  + 被使用者被类似工厂的Factory New
  + 使用者构造函数中接收被使用者的参数，被使用者被传递给使用者的构造函数（依赖注入）
  + 使用依赖注入框架，使用者在形式上构造函数不需要定义被使用者的参数，依赖于配置。
* 使用者里面有被使用者的接口的引用
  + 被使用者在使用者里面New
  + 被使用者被类似工厂的Factory New
  + 使用者构造函数中接收被使用者的参数，被使用者被传递给使用者的构造函数（依赖注入）
  + 使用依赖注入框架，使用者在形式上构造函数不需要定义被使用者的参数，依赖于配置。

继承的耦合度是很高的，而且不是所有形式的code都适合于继承。

而使用形式的耦合是比较常见的。

## 不同层次的耦合度，代码的复杂度是不同的

A 类直接使用B类 ，是最简单的

A类使用B类的接口，就要多一个接口

如果A类不直接new B类，那就要多一个工厂或者Container 负责new B， 代码更多一些

如果A类对B类的依赖通过配置文件动态配置，那么需要有额外的Code来解析配置文件，根据名字反射出B类的对象。这时A对B的耦合度是最低的，但是这些额外的Code如果自己来写的话，那就有点过于复杂了。有框架支持的话当然好很多

## 架构有利于长期，短期肯定是相对麻烦的

就比如盖一栋楼，4个单元，你可以打地基的时候把4个单元的地基一起打了，盖的时候一层一层的盖。

也可以先打一个单元的地基，然后一个单元一个单元的盖。

第二个release的早，也可以早卖，或者设计有问题也可以早发现。缺点是如果单元之间有关联的话比较麻烦，而且保不齐盖后续单元的话早先的单元会带来障碍，甚至要拆掉重盖。

第一个相对稳扎稳打，避免了第一个问题。每次都工作在一个工序，效率也要更高一些。缺点是release的晚，谁知道你有没有未来呢（针对创业公司），如果资金链断了，那可能一个房子也卖不出去，还有就是如果设计有问题，试错成本比较高。

所以，没有银弹，取决于你的现状：

设计的风险： 看设计人员的能力。设计会不会变？

资金支持： 有没有资金压力

有没有阶段交付成品的需求：如果只是最终交付的话，两个是一样的。但第二种提供了阶段性交付成品的能力。

虽然瀑布也能迭代开发，但往往设计是一次做的。

而且，好好架构的程序对于后期的维护，添加/修改 功能，测试性都是有利的。但谁知道你有没有后期呢，又或者你能找到好的架构师么

### 没有银弹，只是最优选择

不是简单的要不要事先好好架构的问题，就比如不能简单的说住大房子好还是住小房子好的问题。你承担的起，又愿意投入的话当然大房子舒服些。但是大房子贵，打扫起来麻烦，又可能离市区远，通勤时间长。

所以最后的决策是各方综合的结果，不是单就某一个点的优劣做决定。

# 设计模式

## 为什么我们的程序中对设计模式留意的少

一个是我们能力的问题

而是设计模式本来就是把不同程序之间共通的部分总结出来形成模式。所以对于类库的编写者或者基础框架的编写者来说，它需要为不同的程序提供共通的支持，所以它会更多的使用到模式。

而对于应用程序开发的人来说，主要注重的是业务逻辑，而业务逻辑本来就是各不相同的，很难说有太多共通的部分需要提炼成模式的，这也是我们程序中模式体现的不多的一个原因。

# 程序和框架

事情总是要做的，要看都是我们自己写code来做还是框架协助着做。

代码的耦合总是要有的，但一般我们真正关心的是自己的程序的耦合度问题。因为我们自己这部分程序的耦合度低了，程序就能更好的测试，适应变化，分工。

而应用程序的各部分总要粘合起来工作，如果框架本身愿意做这部分工作，我们何乐而不为呢，反正我们只要能开发的高效，可测，性能好，可适应变化就好了。至于框架本身开发起来是不是高效，可测，可适应变化，那是框架开发的公司要考虑的问题了，和我就没关系了！

所以，跟我们有关系的是我们要选择一个靠谱点的框架，兼容性好，脏活累活做的比较多的。毕竟，应用程序最终的运行还是要结合我们自己写的Code和框架一起才行。

## 看网上的Demo 程序的不足

有时候网上的Demo程序，往往为了说明一些基础的知识，所有的类，方法都是自己写的，没有用到任何的框架。

这时候你往往发现，好家伙，按照这个来写，实现各功能怎么要写这么多东西啊。还有就是各个component的连接（耦合）还是要自己去处理，看起来耦合度也不低啊，而且代码也不优雅啊。

需要清楚的是网上的这类Demo更多的是为了说明原理。真正工作的时候，还要依赖于特定的框架来工作。等框架把所有的脏活，累活，高耦合，复杂，不优雅都干完了，我们基于它之上当然就可以写出松耦合，简洁，优雅的代码。当然，我们可能丢失了很多底层细节的控制。

Demo是为了说明原理，框架是为了写程序糊口。不要真用Demo的思维去写程序，也不要用框架去学原理。

# MVC MVP 和MVVM

[MVC\_MVP\_MVVM.docx](Doc/MVC_MVP_MVVM.docx)