

# 第04期-常见应用程序架构指南

2019年11月19日 15:19

新式 Web 应用程序比以往承载着更高的用户期望和要求。当今的 Web 应用需要能在全世界任何地方任意时刻可用，需要可在任何设备或屏幕尺寸上使用。Web 应用程序必须具有安全性、灵活性和可缩放性，以便满足高峰需求。如今复杂方案日益需要由丰富的用户体验来处理，这建立在使用 JavaScript 并通过 Web API 进行有效通信的客户端的基础上。

针对这些要求，微软提供了一个可以参考的应用程序 eShopOnWeb 示例，该应用程序演示了一些设计原则和建议，一个简单在线商店，支持浏览衬衫、咖啡杯和其他市场产品。

## eShopOnWeb 项目介绍

由 Microsoft 使用 ASP.NET Core 技术搭建的一个简单商城系统，采用单体应用架构，使用 DDD 领域分析方法，经典的4层结构，很好地展示了单体应用程序体系结构和部署模型。

<https://github.com/dotnet-architecture/eShopOnWeb>

低内存、高吞吐量、跨平台、模块化和松散耦合、轻松实现自动化测试、支持 MPA 和 SPA 开发、简单的开发和部署方式，使用 DDD 分析问题域。

SPA（单页面应用）和MPA（多页面应用）

在传统 Web 应用和单页应用 (SPA) 之间选择

## 体系结构架构原则

### 分离关注点

简称 SOP。在分层架构设计中，关注点分离是核心设计思想，每一层独自负责不同的职责。从架构上讲，可以通过将核心业务与基础设施和用户界面逻辑分离来实现。该原则旨在避免紧耦合，又可确保各个模块独立发展。

**通过一些手段，将不同人关注的东西放在不同的地方，归纳整理起来，这样看起来也舒服，找起来也方便。**

### 封装

不同模块之间通过公开定义良好的接口进行方法调用，来实现封装，以隔离内部的实现机制。通过封装来确保应用程序间不同部分之间的隔离，正确使用封装有助于在应用程序设计中实现松耦合和模块化。

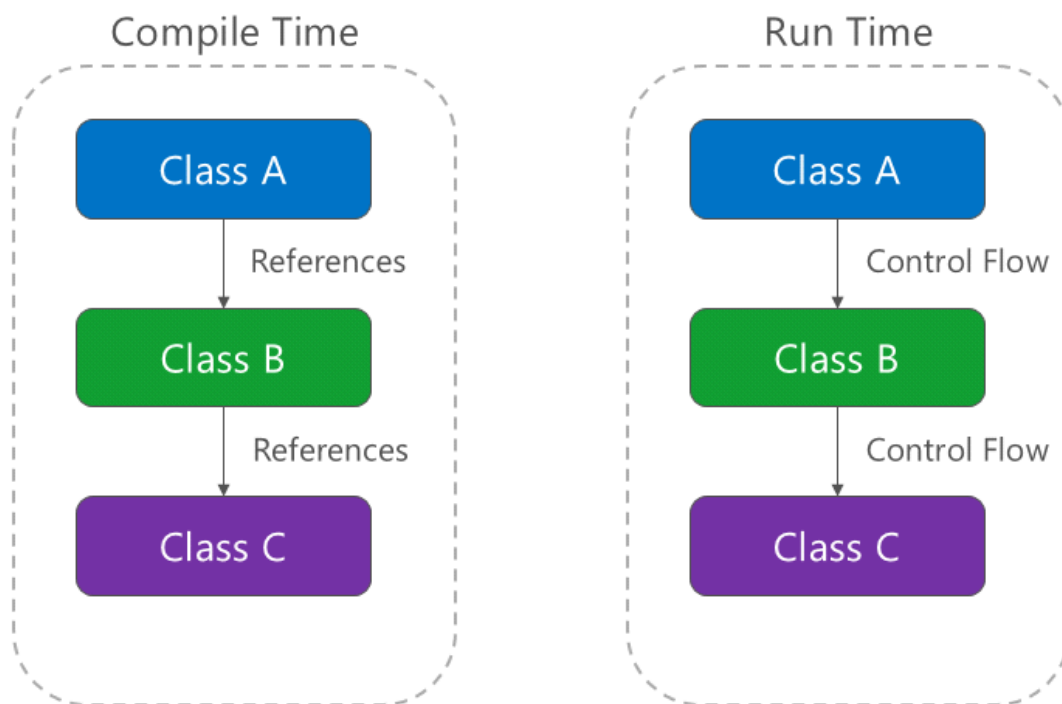
**封装的目的就是不要给外部提供不必要的功能，提供的功能最好是稳定可靠的基于接口。**

### 依赖倒置

应用程序中的依赖关系方向应该是抽象的方向，而不是实现详细信息的方向。如果模块 A 调用模块 B 中的函数，而模

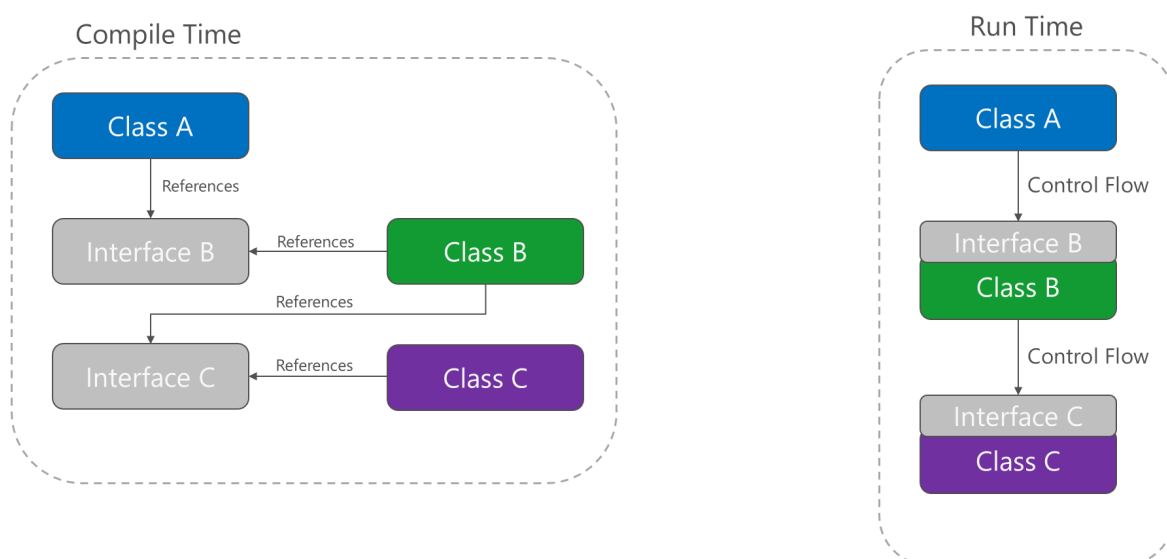
块 B 又调用模块 C 中的函数，则编译时 A 取决于 B，而 B 又取决于 C，如下直接依赖关系图。

# Direct Dependency Graph



应用依赖关系反转原则后，A 可以调用 B 实现的抽象上的方法，让 A 可以在运行时调用 B，而 B 又在编译时依赖于 A 控制的接口（因此，典型的编译时依赖项发生反转）。运行时，程序执行的流程保持不变，但接口引入意味着可以轻松插入这些接口的不同实现。

## Inverted Dependency Graph



依赖项反转是生成松散耦合应用程序的关键一环，因为可以将实现详细信息编写为依赖并实现更高级别的抽象，而不

是相反。因此，生成的应用程序的可测试性、模块化程度以及可维护性更高。遵循依赖关系反转原则可实现依赖关系注入。

**针对接口编程，而不是针对实现编程，大家都用接口，实现变了对你没影响，因为你调用的是接口。**

## 显式依赖关系

方法和类应显式要求正常工作所需的任何协作对象。通过类构造函数，类可以标识其实现有效状态和正常工作所需的内容。如果定义的类可供构造和调用，但仅在具备特定全局组件或基础结构组件时正常工作，则这些类对其客户端而言就不诚实。构造函数协定将告知客户端，它只需要指定的内容（如果类只使用无参数构造函数，则可能不需要任何内容），但随后在运行时，结果发现对象确实需要某些其他内容。

若遵循显式依赖关系原则，类和方法就会诚实地告知客户端其需要哪些内容才能工作。这就可以让代码更好地自我记录，并让代码协定更有利于用户，因为用户相信只要他们以方法或构造函数参数的形式提供所需的内容，他们使用的对象在运行时就能正常工作。

**推荐构造函数注入，一眼就可以看到这个类要运行需要提供那些支撑接口作为条件。**

## 单一职责原则

单一责任原则适用于面向对象的设计，但也可被视为类似于分离关注点的体系结构原则。它指出对象只应有一个责任，并且只能因为一个原因更改对象。具体而言，只在必须更新对象执行其唯一责任的方式时才应更改对象。遵循这一原则有助于生成更松散耦合和模块化的系统，因为许多类型的新行为可以作为新类实现，而不是通过向现有类添加其他责任。添加新类始终比更改现有类安全，因为还没有任何代码依赖于新类。

在整体应用程序中，可以在高级别将单一责任原则应用于应用程序中的层。显示责任应位于 UI 项目中，而数据访问责任应位于基础结构项目中。业务逻辑应位于应用程序核心项目中，该项目易于测试，并且可以独立于其他责任进行逐步改进。

**一个对象干好一件事就可以了，别让它身兼多职，分心分神什么事也干不好，还难以管理。**

## 不要自我重复 (DRY)

应用程序应避免在多个位置指定与特定概念相关的行为，因为这样经常会导致出错。有些时候，对要求中的某处进行更改需要更改此行为，并且该行为可能至少有一个实例无法更新，这种可能性将导致出现不一致的系统行为。

请将逻辑封装在编程构造中，而不要重复该逻辑。让此构造成为针对此行为的单一权限，并让应用程序中需要此行为的任何其他部分都使用新的构造。

**当出现重复时，应该实施重构。避免当功能改进时，需要同时修改多个部分。**

## 持久性无感知

持久性无感知 (PI) 是指需要保持不变的类型，但其代码不受所选择的持久性技术的影响。 .NET 中的这种类型有时被称为普通旧 CLR 对象 (POCO)，因为这种类型无需继承特定的基类或实现特定的接口。

持久性无感知非常有用，因为它可以让相同的业务模型以多种方式保持不变，让应用程序更加灵活。持久性选择可能会随着时间的推移而发生变化，从一种数据库技术变为另一种数据库技术，或除应用程序一开始具备的持久性形式之外还需要其他形式的持久性（例如，除相关数据库之外还需使用 Redis 缓存）。

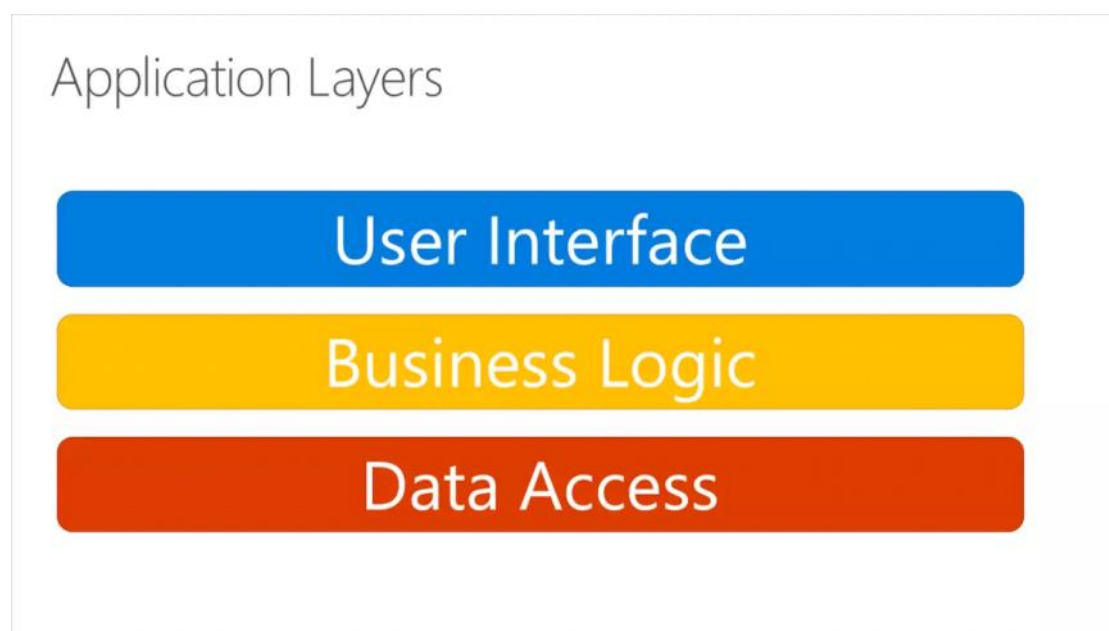
**要求可以轻松切换持久化技术，从而实现持久化无感知（透明持久化）**

## 限界上下文

该概念是DDD战略设计的一部分，通过限界上下文来划分领域，作为领域的显式边界，为领域提供上下文语境，保证在领域之内的一些术语、业务相关对象等（通用语言）有一个确切的含义，没有二义性。

## 传统分层架构和整洁架构

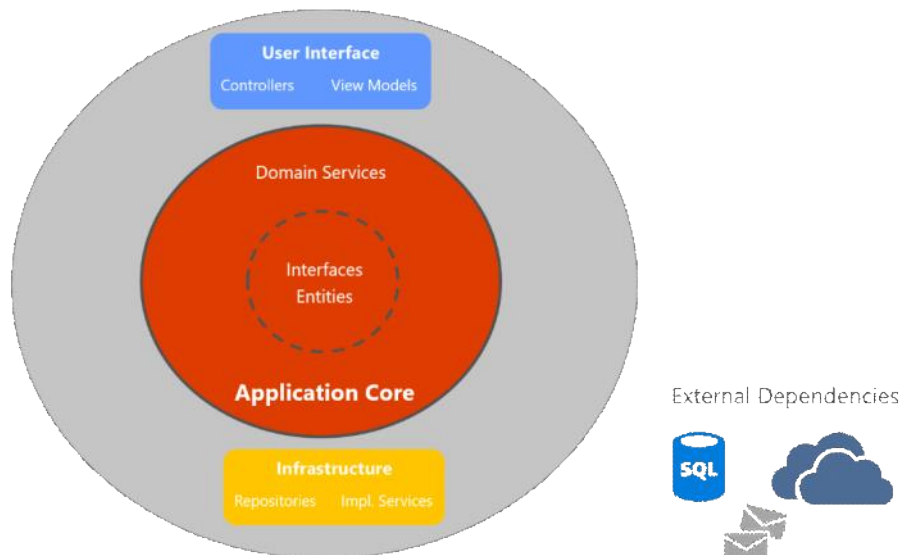
传统的分层架构是大家所熟知的三层架构，传统三层是 UI 层调用 BLL 层，BLL 层调用 DAL 层，每层都有自己熟知的职责，但是缺点是编译时依赖关系由上而下运行，是一种高耦合，依赖程序太大，而在设计原则中应该是低耦合，越低越好。



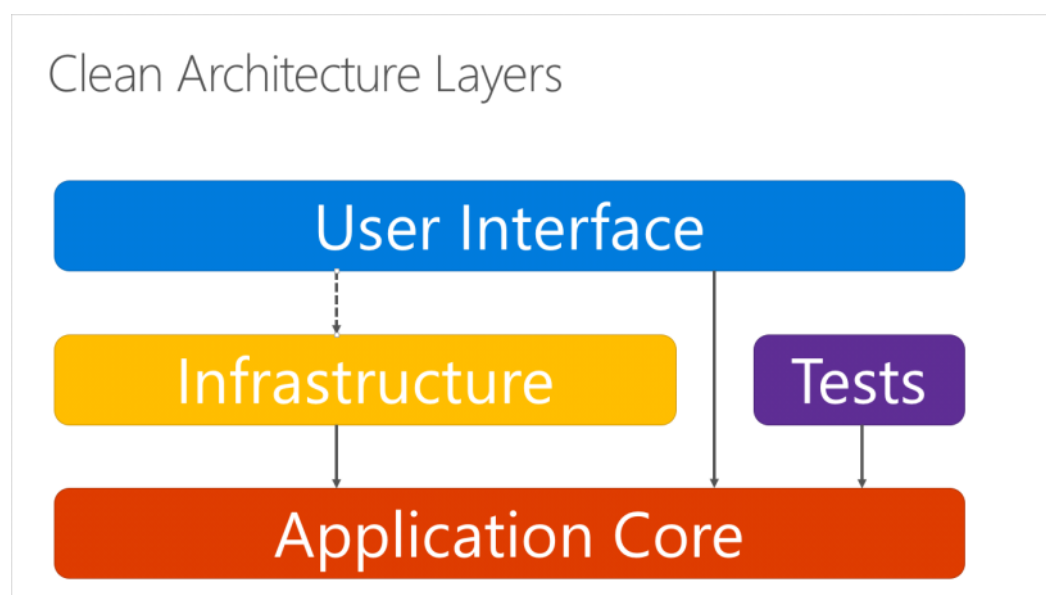
## 整洁架构

整洁架构被称为干净的架构，遵循依赖倒置原则和领域驱动设计原则 (DDD) 的应用程序倾向于达到类似的架构，依赖关系被倒置是：基础设施层的实现细节依赖于领域层，通过在领域层定义抽象或接口，然后由基础设施层中定义的具体类型来实现接口。下图是DDD干净架构多层以"同心圆"形式展示。

# Clean Architecture Layers (Onion view)



下图更好的反映了DDD各层的依赖关系，实线箭头表示编译时依赖关系，而虚线箭头表示仅运行时依赖关系。领域层是架构的核心层，不依赖于基础设施层，该层是可测试的。基础设施层引用领域层来实现数据持久化或横切关注点。



下图是 ASP.NET Core 应用程序在 DDD 领域模型方案中完整构架，展现了各层明确的职责分布，虚线指编译依赖关系，也可以理解为项目引用关系，实线则是运行依赖关系。

# ASP.NET Core Architecture

