服务端 asp.net core

- 基础特案例_____
- 框架设计_____
- efcore

基础特案例

输出多语言版本去掉

```
<SatelliteResourceLanguages>zh-Hans</SatelliteResourceLanguages>
```

异常验证

```
//参数验证不为空
  ArgumentNullException.ThrowIfNull(services);
```

请求头默认传输二进制文件

```
context.Response.ContentType = "application/octet-stream";
context.Response.Headers.Append("Content-Disposition",
new string[] { $"attachment;filename={generate.Domain.TrimStart('*')}.zip" });
```

框架设计

好的，下面我用 **非静态 **（全部用实例对象和依赖注入），结合你的业务场景，给出常见设计模式的实践建议和典型代码片段，并说明哪些地方适合用什么设计模式。

---

1. 单例模式（Singleton）

实践场景: 全局唯一服务（如配置、事件总线、缓存等），用依赖注入的`AddSingleton`实现。

```csharp
public interface IAppConfig { string Get(string key); }
public class AppConfig : IAppConfig
{
    public string Get(string key) => "value";
}
// 注册
services.AddSingleton<IAppConfig, AppConfig>();
```

---

2. 工厂方法/抽象工厂（Factory/AbstractFactory）

实践场景: 仓储、支付、DbContext等对象的创建。

```csharp
public interface IRepositoryFactory
{
    IRepository<T> Create<T>() where T : class;
}
public class RepositoryFactory : IRepositoryFactory
{
    private readonly IServiceProvider _provider;
    public RepositoryFactory(IServiceProvider provider) { _provider = provider; }
    public IRepository<T> Create<T>() where T : class =>
_provider.GetRequiredService<IRepository<T>>();
}
// 注册
services.AddScoped<IRepositoryFactory, RepositoryFactory>();
```

---

3. 策略模式（ Strategy）

**实践场景 **: 支付、鉴权、序列化等可切换算法。

```csharp
public interface IPaymentStrategy { void Pay(); }
public class AlipayStrategy : IPaymentStrategy { public void Pay() { /*...*/ } }
public class WeChatPayStrategy : IPaymentStrategy { public void Pay() { /*...*/ } }

public class PaymentContext
{
    private readonly IPaymentStrategy _strategy;
    public PaymentContext(IPaymentStrategy strategy) { _strategy = strategy; }
    public void ExecutePay() => _strategy.Pay();
}
// 注册
services.AddTransient<IPaymentStrategy, AlipayStrategy>();
services.AddTransient<PaymentContext>();
```

---

4. 装饰器模式（ Decorator）

**实践场景 **: AOP、日志、缓存、权限。

```csharp
public class LoggingRepository<T> : IRepository<T>
{
    private readonly IRepository<T> _inner;
    public LoggingRepository(IRepository<T> inner) { _inner = inner; }
    public void Add(T entity) { Console.WriteLine("日志 "); _inner.Add(entity); }
}
// 注册
services.Decorate(typeof(IRepository<>), typeof(LoggingRepository<>)); // 需用 Scrutor等库
```

---

5. 观察者模式（ Observer）

**实践场景 **: 事件总线、领域事件。

```csharp
public interface IEventHandler<T> { void Handle(T evt); }
```

```csharp
public class EventBus
{
    private readonly IEnumerable<IEventHandler<object>> _handlers;
    public EventBus(IEnumerable<IEventHandler<object>> handlers) { _handlers = handlers; }
    public void Publish<T>(T evt)
    {
        foreach (var h in _handlers.OfType<IEventHandler<T>>()) h.Handle(evt);
    }
}
// 注册
services.AddSingleton<EventBus>();
```

---

6. 责任链模式（Chain of Responsibility）

实践场景：中间件、请求处理管道。

```csharp
public interface IHandler { Task Handle(RequestContext ctx); }
public class AuthHandler : IHandler
{
    private readonly IHandler _next;
    public AuthHandler(IHandler next) { _next = next; }
    public async Task Handle(RequestContext ctx) { /*...*/ await _next.Handle(ctx); }
}
// 注册链
services.AddTransient<IHandler, AuthHandler>();
```

---

7. 适配器模式（Adapter）

实践场景：第三方支付、缓存、消息队列等接口适配。

```csharp
public class AlipayAdapter : IPaymentStrategy
{
    private readonly AlipayService _alipay;
    public AlipayAdapter(AlipayService alipay) { _alipay = alipay; }
    public void Pay() => _alipay.DoPay();
}
// 注册
services.AddTransient<IPaymentStrategy, AlipayAdapter>();
```

---

8. 代理模式（Proxy）

实践场景: 权限、缓存、远程调用。

```csharp
public class RepositoryProxy<T> : IRepository<T>
{
    private readonly IRepository<T> _inner;
    public RepositoryProxy(IRepository<T> inner) { _inner = inner; }
    public void Add(T entity) { /*权限校验*/ _inner.Add(entity); }
}
// 注册
services.Decorate(typeof(IRepository<>), typeof(RepositoryProxy<>));
```

---

9. 命令模式（Command）

实践场景: 任务队列、事件、撤销。

```csharp
public interface ICommand { void Execute(); }
public class CreateOrderCommand : ICommand { public void Execute() { /*...*/ } }
public class CommandInvoker
{
    private readonly IEnumerable<ICommand> _commands;
    public CommandInvoker(IEnumerable<ICommand> commands) { _commands = commands; }
    public void ExecuteAll() { foreach (var cmd in _commands) cmd.Execute(); }
}
// 注册
services.AddTransient<ICommand, CreateOrderCommand>();
```

---

10. 模板方法模式（Template Method）

实践场景: 模块基类、流程控制。

```csharp
public abstract class BaseModule
{
    public void RegisterModule()
    {
```

```
        PreRegister();
        DoRegister();
        PostRegister();
    }
    protected virtual void PreRegister() { }
    protected abstract void DoRegister();
    protected virtual void PostRegister() { }
}
```

---

11. 生成器模式（Builder）

实践场景：复杂对象构建，如配置、SQL、Http请求。

```csharp
public class DbContextBuilder
{
    private string _connStr;
    public DbContextBuilder UseSqlServer(string connStr) { _connStr = connStr; return this; }
    public MyDbContext Build() => new MyDbContext(_connStr);
}
```

---

12. 享元模式（Flyweight）

实践场景：缓存、对象池。

```csharp
public class DbContextPool
{
    private readonly Dictionary<string, MyDbContext> _pool = new();
    public MyDbContext Get(string connStr)
    {
        if (!_pool.ContainsKey(connStr))
            _pool[connStr] = new MyDbContext(connStr);
        return _pool[connStr];
    }
}
```

---

13. 组合模式（Composite）

实践场景：菜单、树形结构、模块组合。

```csharp
public interface IModuleComponent { void Init(); }
public class ModuleGroup : IModuleComponent
{
    private readonly List<IModuleComponent> _children = new();
    public void Add(IModuleComponent child) => _children.Add(child);
    public void Init() { foreach (var c in _children) c.Init(); }
}
```

---

14. 备忘录模式（Memento）

实践场景：快照、撤销、配置回滚。

```csharp
public class ConfigMemento { public string State { get; set; } }
public class ConfigOriginator
{
    public string State { get; set; }
    public ConfigMemento Save() => new ConfigMemento { State = State };
    public void Restore(ConfigMemento m) => State = m.State;
}
```

---

15. 状态模式（State）

实践场景：订单状态、流程引擎。

```csharp
public interface IOrderState { void Handle(OrderContext ctx); }
public class PaidState : IOrderState { public void Handle(OrderContext ctx) { /*...*/ } }
public class OrderContext
{
    public IOrderState State { get; set; }
    public void Request() => State.Handle(this);
}
```

---

16. 访问者模式（Visitor）

实践场景: 对象结构遍历、代码生成。

```csharp
public interface IModuleVisitor { void Visit(BaseModule module); }
public class ModulePrinter : IModuleVisitor { public void Visit(BaseModule module) { /*...*/ } }
```

---

17. 解释器模式（Interpreter）

实践场景: 表达式解析、规则引擎。

```csharp
public interface IExpression { int Interpret(); }
public class NumberExpression : IExpression { public int Value; public int Interpret() => Value; }
```

---

18. 中介者模式（Mediator）

实践场景: 模块通信、事件聚合。

```csharp
public interface IMediator { void Send(string message, object sender); }
public class ModuleMediator : IMediator { /*...*/ }
```

---

19. 桥接模式（Bridge）

实践场景: 抽象与实现分离，如不同数据库、不同支付。

```csharp
public interface IDbImplementor { void Connect(); }
public class SqlServerImplementor : IDbImplementor { public void Connect() { /*...*/ } }
public class DbBridge
{
    private IDbImplementor _impl;
    public DbBridge(IDbImplementor impl) { _impl = impl; }
    public void Connect() => _impl.Connect();
}
```

---

20. 原型模式（Prototype）

实践场景：对象克隆、配置模板。

```csharp
public interface IPrototype<T> { T Clone(); }
public class ConfigPrototype : IPrototype<ConfigPrototype>
{
    public string Value;
    public ConfigPrototype Clone() => (ConfigPrototype)this.MemberwiseClone();
}
```

---

21. 管道/过滤器模式（Pipeline/Filter）

实践场景：请求处理、过滤器链。

```csharp
public interface IPipelineStep { void Process(RequestContext ctx); }
public class AuthStep : IPipelineStep { public void Process(RequestContext ctx) { /*...*/ } }
```

---

22. 外观模式（Facade）

实践场景：简化复杂子系统，如支付、缓存、消息。

```csharp
public class PaymentFacade
{
    private readonly IPaymentStrategy _alipay;
    private readonly IPaymentStrategy _wechat;
    public PaymentFacade(IPaymentStrategy alipay, IPaymentStrategy wechat)
    {
        _alipay = alipay; _wechat = wechat;
    }
    public void PayByAlipay() => _alipay.Pay();
    public void PayByWeChat() => _wechat.Pay();
}
```

---

23. 数据访问对象模式（DAO）

实践场景: 数据持久化，仓储层。

```csharp
public interface IUserDao { User Get(int id); }
public class UserDao : IUserDao { public User Get(int id) { /*...*/ } }
```

---

24. 依赖注入模式（DI）

实践场景: 你的整个框架都在用！

```csharp
// 通过 IServiceCollection 注册和注入
services.AddScoped<IUserService, UserService>();
```

---

25. 发布-订阅模式（Pub/Sub）

实践场景: 事件、消息、通知。

```csharp
public class EventBus
{
    public event Action<string> OnMessage;
    public void Publish(string msg) => OnMessage?.Invoke(msg);
}
```

---

26. 领域驱动设计（DDD）相关模式

- **聚合根（Aggregate Root）**: 你的 BaseEntity、仓储等
- **领域服务（Domain Service）**: Manager 层
- **值对象（Value Object）**: 如 Money、Address

---

27. 空对象模式（Null Object）

实践场景: 避免 null 判断。

```csharp
public class NullLogger : ILogger
{
    public void Log(string msg) { /* 什么都不做 */ }
}
```

---

28. 对象池模式（Object Pool）

实践场景: DbContext、连接池。

```csharp
public class DbContextPool
{
    private readonly Queue<MyDbContext> _pool = new();
    public MyDbContext Rent() => _pool.Count > 0 ? _pool.Dequeue() : new MyDbContext();
    public void Return(MyDbContext ctx) => _pool.Enqueue(ctx);
}
```

---

29. 延迟加载（Lazy Load）

实践场景: 属性懒加载。

```csharp
public class User
{
    private Lazy<List<Order>> _orders;
    public User(Func<List<Order>> orderFactory)
    {
        _orders = new Lazy<List<Order>>(orderFactory);
    }
    public List<Order> Orders => _orders.Value;
}
```

---

30. 组合应用建议

- **仓储层**: 工厂、装饰器、代理、DAO、单例、对象池
- **服务层**: 策略、模板方法、领域服务、状态、命令
- **中间件**: 责任链、装饰器、管道

- **事件/消息**：观察者、发布-订阅、中介者
- **第三方集成**：适配器、桥接、外观
- **配置/缓存**：单例、享元、备忘录、原型
- **模块化**：组合、模板方法、访问者

总结

- 你可以在**仓储、服务、事件、支付、配置、模块、管道**等各层灵活实践上述设计模式。
- 推荐用依赖注入和实例对象，避免静态。
- 设计模式不是越多越好，而是要理解其思想，按需组合。

如需某个模式的详细实现或在你框架中的最佳实践代码，欢迎继续提问！

efcore

dotnet -ef □□□□

```
[Console]::OutputEncoding = [System.Text.Encoding]::UTF8
```