一、相关下载

1 .NET 版本（最新为.NET7 Preview5） 推荐.NET6 LTS

https://dotnet.microsoft.com/download

https://dotnet.microsoft.com/zh-cn/download/dotnet/7.0

2 VS多版本（最新为VS2022 17.1）

https://visualstudio.microsoft.com/zh-hans/downloads/

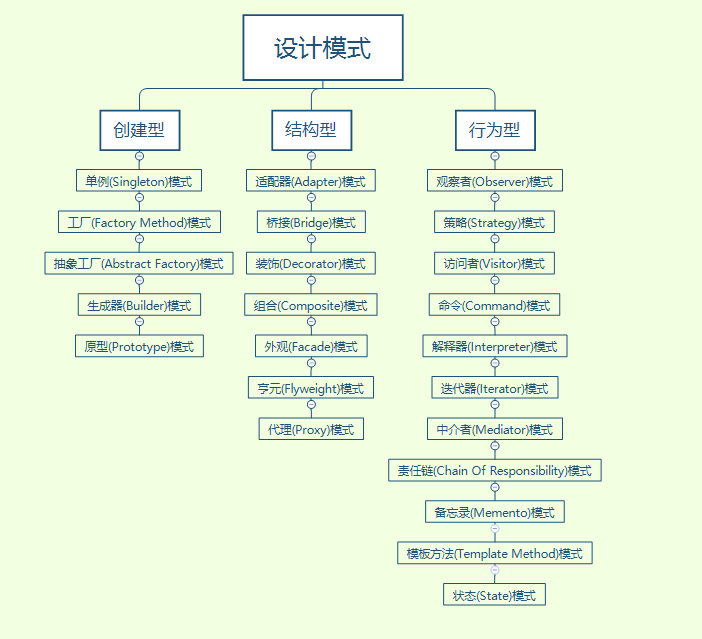
https://visualstudio.microsoft.com/zh-hans/vs/preview/

3 ASP.NET Core源码：

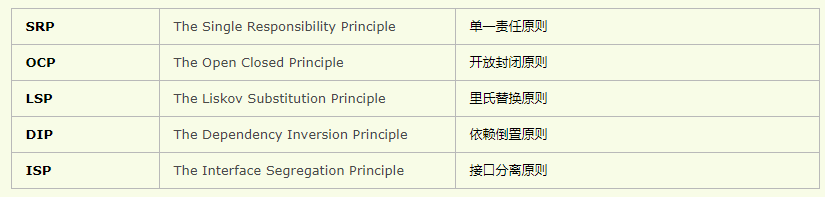
https://github.com/dotnet/aspnetcore #ASP.NET Core

https://github.com/dotnet/runtime #extend扩展库

4.设计模式



五个原则



**SRP - 单一职责**

　　作为开发者，想必大家都有这样的经验，一个方法参杂越多的交叉业务，它的定义就越不明确，复用性就越低。小至方法，大至模块，承担的职责越多，就等于把这些职责耦合在一起，它被复用的可能性就越小。当一个类具有了多项职责，它被更改的可能性也会增加，当其中一个职责发生变化时，可能会影响其他职责的运作，而每一次由于职责变化发生的改动，也会使得bug产生的风险增加。

　　所以SRP强调的是：**引起类变化的因素永远不要多于一个**。这意味着在设计需要的类时，需要考虑使得每个类被设计出来都只有一个目的（或主要目的）。但这并不意味着每个类只能有一个方法，应该是该类中所有的方法都要围绕着该类所描述的主要功能。至于那些有多个职责的类，应该被重新封装成新的类。

**OCP - 开闭原则**

　　该原则强调的是：一个软件实体（指的类、函数、模块等）应该对扩展开放，对修改关闭。即每次发生变化时，要通过添加新的代码来增强现有类型的行为，而不是修改原有的代码。

　　为什么要这样呢？我们都有这样的经历，在接手已经在正式上线的项目时，当需要面对新的需求时，我们首要的任务应该是尽量保证系统的设计框架是稳定的。对于一个功能，一般不会因为新功能的原因而去改变之前已经稳定的功能，因为如果你改变它，很可能你的改变会引发系统的崩溃。

　　OCP提倡的是：**你需要一些额外功能，你应该扩展这个类而不是修改它**。使用这种方式，现有系统不会看到由于新变化的所带来的影响。同时，你只需要测试新创建的类。与SRP一样，该原理通过尽可能减少对现有代码的更改来降低引入新错误的风险。

　　符合开闭原则的最好方式是提供一个固有的接口（或抽象类），为系统提供一个相对稳定的抽象层，然后让所有可能发生变化的类实现该接口，让固定的接口与相关对象进行交互，这样如果需要修改系统的行为，只需在抽象层进行新增业务方法，然后增加新的具体类来实现新的业务功能即可。

**LSP - 里氏替换原则**

　　里氏替换原则（LSP）声明：所有引用基类的地方必须能使用其子类的对象。**也就是说任何基类可以被调用的地方，子类也一定可以被调用。**

　　在软件开发过程中，只有当子类替换掉父类后，此时软件的功能不受影响时，父类才能真正地被复用，而子类也可以在父类的基础上添加新的行为。举个例子：我喜欢运动，那能推断出我一定喜欢跑步，因为跑步是运动的一种；但是从我喜欢跑步却不能推断我喜欢运动，因为我并不喜欢蹦极，虽然它也是运动的一种。

**ISP - 接口分离原则**

　　接口分离原则：**使用多个专门的接口比使用单一的总接口要好。**也就是说不要让一个单一的接口承担过多的职责，而应把每个职责分离到多个专门的接口中，进行接口分离。

　　我们应该都有这样的经历，在一个大的业务类型接口中，定义了非常多的方法，当业务需要在该接口添加新方法时，所有实现该接口的类都要去实现该方法，这样就会导致即使我负责的业务跟你新加的方法没有太大关系，我也得去实现你的方法，并且由于需要实现新的方法，导致客户端会暴露这个方法。

　　根据接口分离原则，推荐的实现的方式应该是：把大的接口拆分，让大类实现多个更小的接口，根据用途对功能进行分组。依赖关系与那些相关联用于松耦合，增加健壮性，灵活性以及可复用性。我们需要注意控制接口的粒度，接口不能太小，如果太小会导致系统中接口泛滥，不利于维护；接口也不能太大，太大的接口将违背接口分离原则，灵活性较差，使用起来很不方便。

　　那这个度是如何控制的呢？这里有一个准则是，基于客户端需要的用途对功能进行分组，仅仅提供客户端需要的行为，不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。

　　我们能看出这是SRP的延伸，所以重构的过程中，可以结合接口隔离原则和单一职责原则进行代码重构。

**DIP - 依赖倒置原则**

　　先说一下这个依赖关系是什么。

　　在实际的开发过程中，我们很多人总是倾向于创建一些高层模块依赖于低层模块的开发策略，因为是低层次提供了对外的接口，高层次只能依赖于低层次所提供的接口，所以这个依赖关系应该是高层依赖于底层。  
　　如何理解高层次和低层次？通俗点说：当A需要用到B时，那A就是高层次，B就是低层次。很明显，如果设计了这些高层模块依赖于低层模块，那么对低层模块的改动就会直接影响到高层模块，从而迫使它们需要作出改动，高层将没有任何的自主性。

　　如何去除这样的关系？依赖倒置原则提倡的是：**高层模块不应该依赖于低层模块**，至此我们应该知道了这个倒置的含义，应该是解除高层和底层的直接耦合关系。很明显，去除这层关系是对的，但如何实现呢？

**依赖倒置的原则是：依赖抽象，不依赖具体实现，**也就是高层和底层的耦合关系通过抽象（接口）来实现。这也是我们提倡的面向接口编程，与之相关的概念是DI（依赖注入）和IoC（控制反转）。

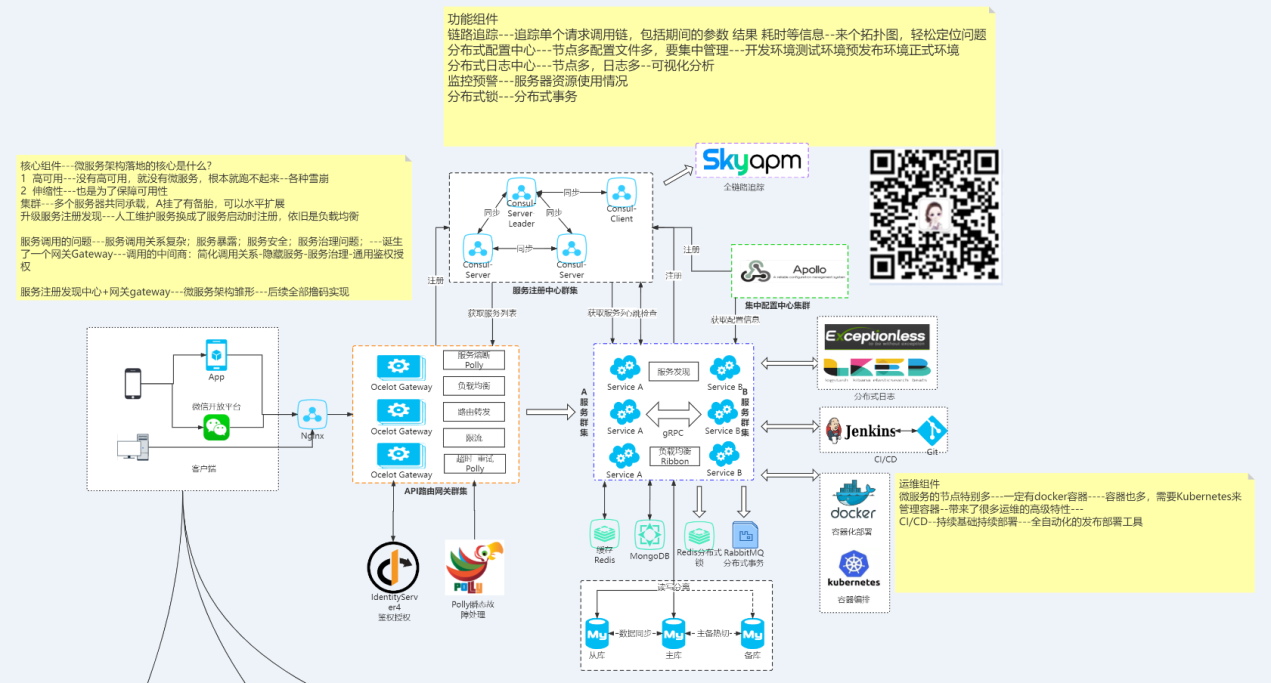
　　该原则规定了在类之间存在依赖关系的情况下，应使用抽象（如接口）来构建它们的耦合关系，而不是直接引用类。 这减少了由低层次模块的变化而导致高层模块的改动。

二、笔记内容

1. Polly是一种.NET弹性和瞬态故障处理库，允许我们以非常顺畅和线程安全的方式来执诸如行重试，断路，超时，故障恢复等策略。

2. Apollo：配置管理平台，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性。

3.框架架构



4.增加依赖注入

依赖注入有三种创建方法：

Transient： 瞬态模式，每一次访问都会创建一个新的实例

Scoped： 域模式，在同一个Scope内只初始化一个实例 ，可以理解为（ 每一个request级别只创建一个实例，同一个http request会在一个 scope内）。对象在一次请求中是相同的，但在不同请求中是不同的。

Singleton ：单例模式，整个应用程序生命周期以内只创建一个实例。

builder.Services.AddControllers();//添加控制器

builder.Services.AddTransient<IUserService, UserService>();//依赖注入