**第一章**

**软件的需求和质量**

**系统的最终目标有一系列需求描述，这些需求最终决定了系统的架构。需求又分为功能需求和非功能需求：**

**功能性需求是指系统需要做的事情，即系统需要为特定的任务和达到用户的目的提供哪些功能。通常而言，一个功能包含输入，行为和输出。软件需求必须“清晰，正确，避免二义，明确且可检验的”**

**非功能需求是指不属于工方面的需求。一个常见的例子是使用(或不使用)某一特定版本的某框架，或是需要让最终产品可以和指定的遗留系统配合使用。还有一些比如支持可访问性，安全性，可靠性等。**

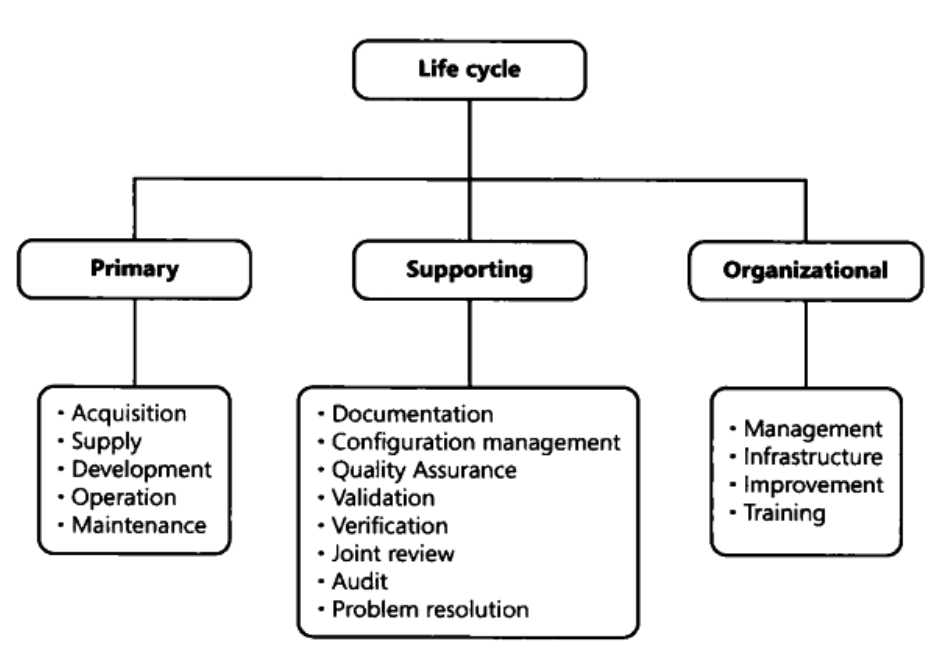
**一般而言，非功能需求表示对系统的一种限制，进而影响系统的质量。**

**架构师的职责：接收需求，拆分系统，确定并评估技术，编写详细说明书。**

**软件的生命周期**

**软件开发是一个用来处理复杂性，并保证程序得到预期结果的流程。软件工程的目的是控制复杂性，而不是增加复杂性。为了达到这个目标，软件项目的各个阶段都需要用某种方法论来指导。**

**根据ISO/IEC 12207标准，软件开发周期分为23个步骤，每个步骤包含一系列的活动和结果。步骤分为3种类型：主要步骤，支持型步骤和组织型步骤。如图所示：**

****

**具体内容可以参考：**[**ISO 12207**](https://baike.baidu.com/item/ISO%2012207/6634612)

**获取(Acquisition)步骤包含引出需求，评估可用方法，协商和约定。**

**供应(Supply)是指供方提供系统，软件产品或软件服务的活动**

**开发(Development)步骤包括分析需求，设计系统，实现系统，测试与部署，这也成为了架构师大显身手的地方。**

**运营(Operation)包括让软件可以在公司中使用，与现有系统集成，进行小规模试验，及辅助用户熟悉软件等。**

**维护(Maintenance)步骤着眼于保持系统良好运行，包括修复Bug，提供新功能等。这其中有限活动也许也需要架构师的参与。**

**墨菲法则：**

1. **为一个赶不上进度的软件项目增加人手只能让其更加落后于进度。**
2. **程序的复杂性会一直增加，直到负责维护的程序员力不从心为止。**
3. **若建筑师按照程序员写的程序那样造房子，那么史上出现的第一只啄木鸟也许会毁掉整个文明。**

**2018-04-11**

**第二章**

**UML图**

**UML是一种面向对象建模的标准语言，它不依赖与任何语言，这是它的一个关键特性，带来好处的同时也带来了坏处，好处在于它让UML成为了一种 能够为任何业务模型流程建模，并工交流的通用语言。而坏处在于它只能用固定的元模型生成大概的模型，这与能组成系统的可编译语言相差甚远。**

**UML依赖于各种图表而存在。最新版本的UML支出13中图表。**

**第三章**

**高内聚，低耦合**

**高内聚表示某个特定的软件模块----无论是一个子程序，一个类型还是一个类库，都应完成一系列极为相关的功能。换句话说，内聚表示同一个类型中不同方法，同一个类库中不同函数，同一个方法中不同操作所表达的逻辑之间的距离。**

**我的理解是你在订单模块中尽量只写有关于订单的代码，不要在里面写关于用户模块的代码。如需要修改用户，你可以调用用于模块的接口完成功能。**

**Ward Cunningham为内聚给出了一个简洁实用的定义。在他看来，如果修改了模块A，且不会影响到模块B，那么A和B这两个模块就有了足够的内聚，值得在系统中使用。**

**还有一句话，可以用来巩固前面提到的内聚概念。Cunningham建议，模块的内聚的高低程度与其担当的职责多少成反比关系。**

**耦合用来度量两个软件模块，例如，类型，函数或类库之间的依赖程度。耦合的一个非常经典的描述来自Cunningham的wiki。若每次修改模块A时都要修改模块B，那么A和B者两个模块是耦合的。**

**低耦合并不是说你的模块应该与其他模块完全隔绝。模块之间显然需要通讯，不过通信应该依赖于一系列设计良好且不易改变的接口。每个模块都应该在无需了解另一个模块内部实现细节的情况下与之良好的配合。**

**相反的，高耦合会阻碍测试过程，降低代码重用性并影响代码的理解。高耦合也是导致软件易碎，坚硬的一个原因。**

**2018-05-21**

**面向对象设计**

**面向对象设计的基本原则：找到合适的对象，尽量降低耦合，尽量保证代码重用。**

**1.我们如何找到合适的对象呢？我们不应该考虑算法，而是专注于设计互相交互的实体。交互的实体就是合适的对象。**

**2.尽量降低耦合，基于接口编程，而不是实现编程。若类型的依赖基于接口，那么也就最小化了类型之间的依赖--即仅依赖于接口中定义的方法。**

**3.尽量保证代码重用，《设计模式》中作者总结出了两种实现重用的方式：白盒重用和黑盒重用。前者基于类型继承会导致一些问题，比如：若派生类继承的父类发生了修改，可能会破坏子类。后者是基于对象组合。**

**对象组合是指，创建一个新的类型，包含基类的实例，用一个私有成员变量保存其引用。这样包装类型，是通过使用原类型，而不是修改原类型来完成工作的。这也就是面向对象设计的第二个原则：尽量使用对象组合，而不是类型继承。**

**我们应该选择类还是结构？**

**在C#定义一个类型时，应该使用类还是结构呢？应该注意的是结构是不能被继承的。因此，如果需要基于该类进行派生，那么必须选择类而不是继承。此外，类是引用类型，分配在堆上。从内存角度考虑，引用类型由垃圾收集器管理。相反的，结构是个值类型，分配在栈上并在离开作用区时消亡。通常而言，值类型比引用类型更轻量一些，除了在需要装箱的时候。**

**什么是可测试性？**

**Bruce Eckel曾提出一句流行的口号，表明软件测试的重要性：**

**若软件没有经过测试，那么可定无法完成工作。**

**测试会位于不同的层次上，比如用单元测试来判断软件的各个独立单元是否符合功能性需求，用集成测试来判断软件能否适应运行环境和基础设施，以及多个组件能否正常协同工作。最后，验收测试用来判断完成的系统是否符合客户的需求。**

**单元测试和集成测试属于开发团队的工作，进而让团队对软件质量产生信心，测试结果会说明团队的开发状态是否朝着良好的方向发展。通常而言这些测试不会覆盖整个代码，代码的覆盖程度和代码的质量一般也没有清晰的关系。**

**而客户通常对但单元测试和集成测试的结果并不在意，他们所关心的仅仅只是验收测试而已。验收测试着眼于完成的系统，属于客户和开发团队合同的一部分。验收测试可以由客户书写，也可以由负责项目的团队和客户协助完成。在验收测试中你会发现如下的核对清单：**

1. **使用如下数据(....)插入一个客户**
2. **使用现有的ID修改该用户**
3. **观察系统反应，得到期望的结果**

**另一个验收测试结果如下：**

1. **在批处理过程中，关闭应用程序服务器的某个节点**
2. **观察系统反应以及事务的结果**

**若验收测试顺利通过，那么项目也就宣告结束，产品也宣告成功。**

**单元测试实战**

**我们推荐用一个专门的程序集包含测试代码。此外，对每个类库都要提供测试。最好的实践是对程序集中每个xxx类，都有一个对应的xxxTestCase类用来测试。**

**我们可以使用伪对象(Fake)消除依赖**

**例子：public class FakeLogger: ILogger  
 {**

**Public void Log(string message)  
 {**

**Return;**

**}**

**}**

**软件可以正常工作的概率与它所需的代码行数成反比。**

**Bug出现的几率与正在查看该软件地人数及这些人的重要程度成正比。**

**任何人都可以写出计算机能够理解的代码，而只有优秀的程序员才能写出别人可以理解的代码。 --Martin Fowler**

**2018-05-23**

**第四章 业务层**

**Layer和Tier的区别**

**对于Tier，我们用其表示系统中物理上的硬件和软件，由执行同样功能的一台或多台服务器定义。相反地，Layer用来表示系统中完成指定任务的逻辑部分。**

**因此多个Layer可以同存在同一台物理Tier中，某个Layer也应该可以轻易移动到另一个Tier上。**

**良好的可重用性加上正确地对功能进行拆分将大大降低系统的维护成本。总而言之，逻辑分层可以很大程度上缩短开发周期，并允许不同团队并行开发。**

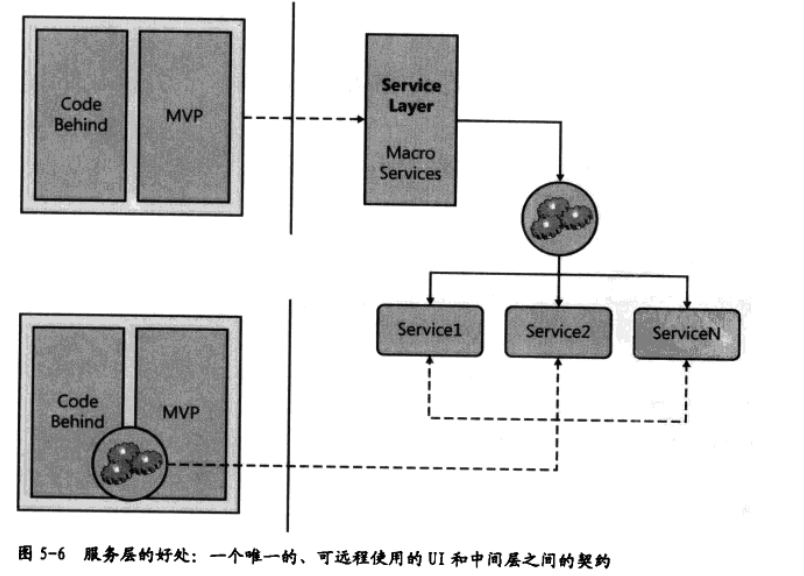
**物理层中一个重要的概念是，一个层就表示一个需要穿越的边界，这个边界可能是进程边界或计算机边界。穿过边界是代价很高的操作，若需要到达远程计算机，那么代价比到达同一台计算机中不同进程高。一个估计比例是穿越边界比进程内部调用慢100倍左右。若需要通过网络访问，还要慢一些。**

**通常来说 ，系统中多个物理层并不算是一个好的特性。物理层会降低整体性能，并增加整体的复杂性，这两个方面会影响到整个系统的创建和维护。因此虽然有些分层不可避免大，但是这并不是好的做法，然而又不得不这样。**

**第五章 服务层**

**服务层中的服务**

**从架构上讲，服务层应用了软件设计中一个通用且人人皆知的原则--低耦合，可能还应用了另一个原则，高内聚，这是通过一个中间层解除用户界面和中间层的耦合来实现。**

****

**在该图中，服务层表示成了一个远端的层，不过实际中并不要求这样实现，其取决于项目的客户端(Web或Windows)。服务层可能就是表现层的一部分。如果那样，服务层不需要设计成Web服务，而是可以用支持服务的组件或一些普通类来实现。这里的重点是服务层所提供的抽象，而不是其具体的实现方式。**

**那么服务层有什么好处呢？**

**在服务器端，服务层中被调用的方法将协调并向领域模型，专门的应用程序服务，工作流等业务逻辑组成部分发起调用，从而组织程序需要的逻辑。**

**若没有服务层，则需要从表现层直接调用到应用程序服务中，这就会造成一个太过细粒度的远程接口，从而导致过度多的交互。因此，你可能不得不为了完成指定的任务而执行多次远程调用。从性能上考虑这当然不是一件好事。**

**怎样绕过服务层？**

**应用程序服务(也叫做微服务)位于服务器端，用来表示应用程序特定的服务以及领域逻辑的功能。举例说明，微服务有执行货币转换，以及基于工作流的服务，用来计算订购货物的数量并保证有足够的库存。**

**一般情况下我们总是推荐使用服务层，不过有时候让表现层直接调用应用程序服务也未尝不可，特别是当该服务特别简单，仅执行了一个单一操作的时候。**

**比如说一个电商网站，前台页面请求后台数据，后台即服务层，服务层获取到请求后，走业务逻辑(调用缓存，调用数据层，调用其他平台的服务)之后，再返回结果到前台。这样做可以前后端分离，提高开发效率。**

**2018-05-25**

**何时使用服务层？**

**服务层应该用在有所有一定复杂度的应用程序中，在多层分布式系统中，除非你只只用一种前端(例如Web端)，且应用程序服务的方法与系统的用例能很好的匹配起来，这时服务层仅仅起到一个分发的作用，并且没有工作需要组织。那些简单的，仅用来把请求转发给业务层的服务确实有些多余。**

**相反，若你有多个前端且应用逻辑较为复杂，那么最好将整个业务逻辑放到统一的位置，而不是为每个应用程序的接口都重新编写一次。**

**服务层的优势与劣质？**

**服务层添加了额外的抽象并除去了两个交互层之间的耦合，若这一点在你的系统中比较重要，那么应该创建服务层。服务层可以帮助你实现一个粗粒度的远程接口，并降低表现层与业务层之间的通信流量。**

**因为抽象是服务层的最主要的长处，所以简单系统中的服务层或许有过度设计之嫌。**

**正如我们刚刚提到的那样，服务层并不一定要使用WCF等专门的服务技术来实现。例如在ASP.NET表现层中，我们一般从代码后置中调用“服务层”，当然服务层和代码后置类都部署在同一台服务器上。而若在这里使用WCF服务作为服务层，则会增加额外的开销，进而影响到性能。在使用WCF作为服务层时，请尽早随时留意性能指标的变化。若性能影响太大，则有必要使用另一种服务层技术。**

**Web前端的服务层**

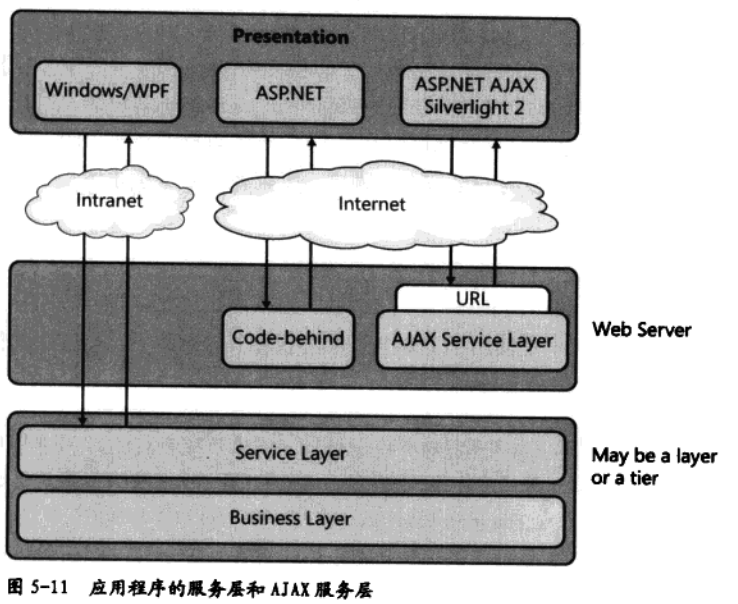
**在ASP.NET Web应用程序中，表现层被分成了两个物理层------运行JavaScript代码的Web浏览器和运行托管代码语言编写的代码后置文件及表现层逻辑的Web服务器。大多数时候，服务层和表现层的逻辑运行在同一个IIS工作进程中，目前看了这没有问题。**

**不过随着AJAX等富Web前端的兴起，情况发生了很大的变化。Web浏览器的功能越来越多，表现层的任务也越来越重。同时，服务层也需要重新设计，以便更好的应对新的挑战，特别是安全方面。**

**在经典的，表现层逻辑完全位于服务器的Web场景中，我们通常使用表单验证来实现安全。未经认证的用户不可能访问到需要回送病访问层的页面。服务层的安全性很容易就能使用代码访问安全，角色等。**

**不过将服务暴露给基于JavaScript的客户端时，你会允许客户端直接连接吗？理论上这样没问题。不过实际上，暴露给JavaScript客户端的服务也就是暴露给Internet上所有的用户。认证信息不能作为消息的一部分同时发送，因为这需要专门的JavaScript功能。更加重要的是这些功能可能会被攻击者偷走，进而威胁到系统的安全。**

**若想发送认证信息，则应该如果ASP.NET表单一样使用加密后的cookie。随后，服务层首先检查cookie，然后再执行逻辑，这个步骤必不可少 ，不过我们不希望将其直接放在服务层中，这也正是为AJAX服务提供额外的一层。**

****

**那么我们如何实现服务层的安全性呢？**

**很简单，我们只需要检查并识别一个合法用户可以提供，而外来者不能提供的信息令牌，最简单也是最有效的令牌就是ASP.NET表单认证所生成的一个认证cookie。**

**2018-05-31**

**第六章 数据访问层**

**数据库的独立性**

**数据库独立性的含义是无论最终数据存储的介质是什么，数据访问层都要能够为系统的其他部分提供相同的服务，例如，创建，读取，更新，修改，事务性和查询等。若你的模型中包含一个名为Customer的对象，那么数据访问层要能够将其存放在不同的存储介质中，例如，SQLServer或Oracle。数据访问层将对业务层隐藏此细节。**

**那么我们如何实现数据库的独立性呢？通常来说，数据库的独立性需要使用一套通用的，跨数据库的应用程序接口。当业务层获取到当前数据访问层的引用之后，业务层以多态的形式，通过公开接口的方法与数据库进行通信。我们可以在配置文件中切换数据库。**

**还有一种方法也可以实现数据库的独立性，就是使用ORM关系映射工具。他会提供一套公共的API，让你只需修改配置文件即可切换到不同的数据库。**

**数据访问层的职责**

**数据访问层对其使用者来说有4个职责。首先，数据访问层需要将数据持久化到物理存储中，并为外界提供CRUD操作 。**

**其次数据访问层还能处理接受所有数据相关请求，数据访问层必须满足事务性的需求。最后，数据访问层也要合理地处理并发问题。**

**1.CRUD服务**

**CRUD服务是一系列的方法，负责将对象保存至关系型数据表，或从关系型数据表中读取数据并加载至新创建的应用程领域模型的类型实例。**

**应该为实现CRUD服务编写怎样的代码呢？对于对象模型的每个类型，我们都要为其创建一个映射类，并实现一个接口。该接口中给出了该类型上所有数据库的相关操作。映射类内部可以使用ADO.NET或Linq To SQL来具体的实现。**

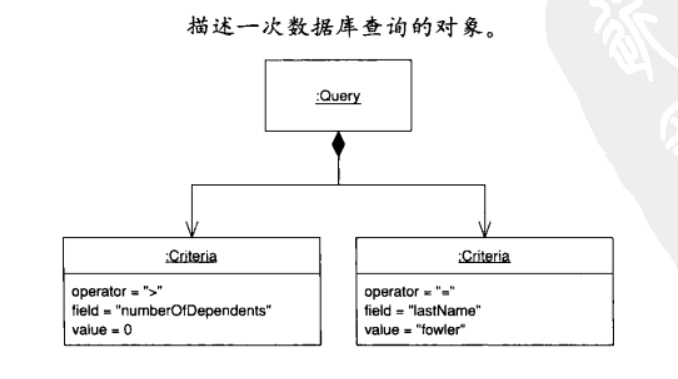
**2.查询服务**

**在某些情况下，我们会需要处理很复杂的查询，而且也不是所有查询都是一样的。有时我们会提取出一些通用的查询，在业务层和服务层的多个位置使用。**

**更好的做法是将所有的硬编码，在业务层被多次使用的查询定义在仓储类中。此外，再定义一个通用的查询对象，能够以编程的方式配置并在执行时生成所需要的SQL代码。**

**该查询对象的主要目的是通过对象的各个属性和方法收集数据，然后为数据库动态生成所需要的SQL代码。**

**查询条件属于查询对象模式的应用，根据该模式，查询对象将定义成类，并包含一系列查询条件的集合。查询条件是一个简单的类，包含一个属性名，一个值，一个逻辑操作符，用来表明属性名和值之间的关系。**

****

**3.事务管理**

**在一个设计良好的数据访问层中，应该有一种机制跟踪一个工作单元内对应用程序数据的所有修改，这样即可在稍后将这些修改一次性地持久化到数据库中保存。这个工作单元就是一个逻辑上的事务，它包括一系列的数据库调用。**

**所以我们应该创建一个能够维护发生变化的领域对象列表的类。这个类同时也提供了事务的语义（开始，提交，回滚），配合数据库的物理事务将修改提交回数据库。**

**4.处理并发**

**所以我们应在一个多用户环境中，与数据库离线操作会导致数据完整性的问题。用户A加载了产品1234的一个拷贝并更新了其描述，因为该操作属于一个较长的事务，我们可以假设该修改并没有立即提交。而同时，假设B也加载了1234的另一个拷贝，并更新了产品图片的URL。若用户B立即提交了它的修改，那么当用户A提交产品修改时，系统又该如何应对呢？**

**你获取会说“这没问题呀”，因为两个用户更新了不同的字段。不过若是两个用户修改了相同的字段又该如何呢？很有可能第一个用户对产品的修改就这样丢失了，这通常叫做“最后写入者获胜”。**

**乐观的并发处理意味着用户可以自由地尝试更新数据库中的任意记录，不过成功与否不能保证。若数据访问层发现即将要更新的记录已经被别人修改过，那么此次修改失败。**

**5.整合所有职责：数据上下文**

**简而言之，你需要一个更高层次的类，用来表示并引导与底层存储介质的交互工作。随后，数据访问层的使用者就可以用该高层次的类作为存储介质的统一操作位置。ORM都有此高层次的类。**

**这个可以统一访问数据访问层及其CRUD，事务和并发服务的高层次类通常叫做数据上下文（DataContext）。**

**2018-06-05**

**第七章 表现层**

**表现层的常见误区**

1. **表现层开发过于依赖RAD工具**

**直接在Click事件的处理函数中编写T-SQL语句当然也可以完成任务，且方便快捷，这也是RAD最为强大的功能。只要移动鼠标，单击，选中某个元素属性，然后双击，工具就为你自动生成了一大批外层代码，只要在其中填写一些逻辑即可。**

**无节制地滥用RAD开发模式会让你创建出很多自治视图。所谓自治视图，是指用一个来编写用户界面的状态和其中需要的所有行为。自治视图目前已经不再流行，最主要的原因就是它难以测试。自治视图在同一个类中同时包含了表现层逻辑和用户界面，这也让测试变得非常困难，甚至是根本不可能。**

1. **表现层的边界**

**根据前面的介绍，我们知道在一个有效，良好设计的表现层中，UI层是一个很薄的层，表现层逻辑仅负责协调正确的数据流入/流出表现层。**

**对于大型，需要长时间使用的企业级系统来说，UI必须与代码隔绝，代码也必须分层实现，以期高度遵守分离关键点原则（SOC）。合理设计的分层不仅能分离关键点，还可以带来很好的可测试性，进而保证产品的质量，因为在高覆盖率的测试中，很容易就能找到并提出大多数早期Bug。此外将逻辑放在其所属的层中，也有助于重构。**

**MVC模式概述**

**MVC模式的主要目标是将应用程序分为各司其职的部分—模型，视图，控制器。模式是指应用程序的状态，封装了应用程序的功能，并通知视图其状态发生了变化。视图是指将要展示给用户的图形化元素的生成，视图还需捕获并处理用户的操作。控制器将用户的操作映射出成程序的操作，并负责选则下一个视图。**

**MVC带来了一下几个优势。首先，它简化了测试对用户界面的操作。将代码从视图中分离开意味着修改图形界面不会影响到用户界面的行为。尽可以地将代码从视图中移走也更有利于结构化，并在逻辑上分层。无论何时想达到高内聚低耦合都可以通过分离关注点达到目的。**

**MVC中的视图，应该非常简单没有任何逻辑，以至于无需进行测试。所有除了处理图形渲染之外的功能理论都应该放在视图外部，即模式和控制器中。**

**视图由一系列交互的控件组成，并等待用户的操作。比如当用户单击某个按钮之后，视图将简单的把这个调用转发给控制器。**

**视图的另一个关键职责是呈现。视图和模型是通过观察者模式绑定到一起的。在观察者模式中，主动方（模型）将在发生变化时通知观察者（视图）。随后，视图从模型中读取当前的状态，并进行相应的更新。**

**使用ViewModel的目的就是让View单个的对象来进行渲染，另一方面可以减少UI展示的逻辑代码，这个是很有必要的，也就是说View唯一的任务就是渲染单个的ViewModel对象，在 MVC 中使用 ViewModel会让你的程序组件之间分离的更好，从而方便维护，记住，单元测试指的是测试小单元。**