# [.Net Web开发技术栈](http://www.cnblogs.com/1996V/p/7700087.html)

有很多朋友有的因为兴趣，有的因为生计而走向了.Net中，有很多朋友想学，但是又不知道怎么学，学什么，怎么系统的学，为此我以我微薄之力总结归纳写了一篇.Net web开发技术栈，以此帮助那些想学，却不知从何起的朋友。

本文整理了当前企业web开发中的管理系统，商城等系统的常用开发技术栈。

## ****C#常见运算符****

* 一元运算符（+、-、!、~、++、--）
* 算术运算符（\*、/、%、+ 、 – ）
* 移位运算符（<< 、>> ）
* 关系和类型测试运算符（==、!=、<、>、<=、>=、is 和 as）
* 逻辑运算符（&、^ 和 | ）
* 条件逻辑运算符（&& 和 || ）
* 空合并运算符（??）条件运算符（它也称为三元运算符，?: ）
* 赋值运算符（=、+=、-=、!=、/=、&=、|=）
* **什么是表达式？**
* 表达式是由运算符和运算对象组成，单独的一个运算对象（常量/变量）也可以叫做表达式，这是最简答的表达式。
* eg：4,4+2，c = 7+5,3>5，……

## ****C#常见语句块****

* Try（用于捕捉在块的执行期间发生的各种异常）
* Checked 语句和 Unchecked（用于控制整型算术运算和转换的溢出检查上下文）
* Lock（获取某个给定对象的互斥锁，执行一个语句，然后释放该锁）
* Using(获取一个或多个资源，执行一个语句，然后释放该资源)

## ****C#常见类修饰符****

* abstract抽象类

C#允许把类和方法声明为abstract。抽象类不能实例化，而抽象方法不能直接实现，必须在非抽象类的派生类中重写

* sealed密封类
* 给类添加sealed修饰符，就不允许创建该类的子类。密封一个方法，

示不能重写该方法

* virtual虚方法

把一个基类方法声明为virtual，就可以在任何类的派生类中重写该方法。也可以将属性声明为virtual

* static静态类

静态类无法被实例化，仅包含静态成员。Static修饰的成员，是属于类的，当这个类第一次加载时，该类下的所有静态成员会被加载，静态成员是唯一的，直到程序退出才会被回收。

* Override 重写方法

C#要求在派生类的函数重写一个函数时，要使用override关键字显示声明

* partial分部修饰
* 访问修饰符（public、protected、internal 和 private ）

|  |  |  |
| --- | --- | --- |
| 修饰符 | 应用于 | 说明 |
| Public | 类和类的成员 | 任何代码均可访问 |
| Protected | 类的成员 | 只有派生的类型能访问 |
| Internal | 类和类的成员 | 只有包含在它的程序集中才能访问能访问 |
| Private | 类的成员 | 只有在类的内部才能访问 |

## ****C#常用预处理指令****

* #region：指定一个可展开或折叠的代码块
* #endregion：它标识着 #region 块的结束
* #define： 它用于定义一系列成为符号的字符
* #undef：它用于取消定义符号
* #if、#else、#elif、#endif：用于创建复合条件指令
* #waring常用于#if语句中，用来提醒调试代码是否启动

## ****委托****

* Delegate、Lambda、Action、Func、Predicate、事件EventArgs
* 要把方法传递给一个方法时就需要用到委托。该方法可以是普通方法，也可以是静态方法。
* 多播委托：委托也可以包含多个方法，如果使用多播委托就可以按循序调用多个方法，为此多播委托的返回值必须为空
* Lambda表达式，从C# 3.0开始，就有一种的新的方法实现委托的赋予：Lambda表达式。只有委托参数类型的地方，就可以使用Lambda表达式。
* 泛型Action<T>委托表示引用一个void返回类型的方法。这个委托至多可以传递16种不同的参数类型。Action<in T>调用带一个参数的方法。
* Func<T>顾名思义它的使用方式类似于方法，所以它允许带返回值。它也可以至多传递16个参数类型。Func<out TResult>可以调用带返回值且无参数的方法
* 事件基于委托，为委托提供了一种发布/订阅机制

## ****泛型****

* 泛型类、泛型方法、泛型约束、泛型接口、协变和抗变
* 泛型的一个主要优点是性能，避免了拆箱和装箱操作
* 泛型的另一个优点是类型安全，在泛型类List<T>中，泛型类型T定义了允许使用的类型。
* 泛型类型的命名规范：
* 泛型类型的名称用字母T作为前缀
* 如果没有特定要求泛型类型允许用任意类替代
* 如果泛型类型有特定的要求，或者使用了两个以上泛型类型，就应该给泛型类型使用描述性的名称，如：TEventArg，TValue，TKey，TInput
* 创建一个泛型类有4点需要注意的：
* 1.默认值，由于泛型类型可能是值类型(默认值0)或引用类型(默认值null)，为了解决该问题，可以使用default关键字，通过default关键字，将null赋予引用类型，0赋予值类型。
* 2.约束，如果泛型类需要调用泛型类型中的方法，就必须添加约束，将T改为Txxx，并在where子句中指定实现的接口。如：public class DocumentManager<TDocument> where TDocument : IDocument {
* 3.继承，泛型类型可是现实泛型接口，也可以派生自一个类，泛型类也可以继承泛型基类，其要求是必须重复接口的泛型类型，或者必须指定基类的类型。
* 4.静态成员，如：public class StaticDemo<T>{ public static int x}，由于同时对一个string类型和一个int类型使用了StaticDemo<T>类，因此存在两组静态字段。
* 使用泛型类型可以定义接口，在接口的定义中可以使用带泛型的参数
* 协变和抗变：在.NET 4之前，泛型接口是不变的。.NET 4通过协变和抗变为泛型接口和泛型委托添加了一个重要拓展。协变和抗变指对参数和返回值的类型进行转换。
* 协变：协变类型用out参数标注，这也意味着返回值只能是T
* 抗变：抗变类型用in参数标注，泛型接口就是抗变的，这样泛型类型T就只能把T作为参数输入
* 泛型方法：除了定义泛型类之外，还可以定义泛型方法，在方法声明中可以定义泛型类。泛型方法可以在非泛型类中定义。

## ****Linq****

* Linq to sql / Linq to DataSet / Linq to Object / Linq to Xml
* 概念：语言集成查询(Language Integrated Query,Linq)在C#中集成了查询语法，可以用相同的语法访问不同的数据源。Linq提供了不同数据源的抽象层，所以可以使用相同的语法。
* 概念：Linq为IEnumerable<T>接口提供了各种拓展方法，以便用户在实现了该接口的任意集合上使用Linq查询。(PS：类似于ArrayList,List,SortedList,Queue,Stack等集合都实现了IEnumerable接口.)
* 特性：在运行期间定义查询表达式时，查询就不会运行，查询会在迭代数据项时运行。
* 特性：并不是所有的查询都可以用Linq查询语法完成。也不是所有的拓展方法都映射到Linq查询子句上。高级查询需要使用拓展方法。
* 语法格式：查询表达式必须以from开头，以select或group结束。在这两个子句中间可以使用where，orderby ，join，let和其他from子句。
* 内连接用join子句合并两个数据源
* 左外连接用join子句和DefaultIfEmpty方法定义
* 常用方法介绍：
* Where()方法:
* Where<TSource>(this IEnumerable<TSource> source, Func<TSource, int, bool> predicate);
* 该方法的委托参数第一个参数TSource表示数据源(即调用它的对象)，第二个参数是索引序列，返回值为bool
* Where<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
* 该方法的委托参数第一个参数TSource表示数据源，返回值为bool
* SelectMany()方法:
* SelectMany<TSource,TResult>(this IEnumerable<TSource> source, Func<TSource, IEnumerable<TResult>> selector);
* 该方法的委托参数第一个参数TSource表示数据源，返回值为合并后的集合
* SelectMany<TSource, TCollection, TResult>(this IEnumerable<TSource> source, Func<TSource, IEnumerable<TCollection>> collectionSelector, Func<TSource, TCollection, TResult> resultSelector);
* 该方法的第一个委托参数第一个参数TSource表示数据源，返回值为合并后的集合，第二个委托参数第一个参数TSource表示相同的数据源，第二个参数表示第一个委托参数的返回值(集合)，第三个参数表示合并后的集合
* OrderBy() OrderByDescending() ThenBy() ThenByDesceding() 方法。
* OrderBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
* 上面四个方法的第一个委托参数TSource表示数据源，返回值为排序的关键字名称TKey
* GroupBy() 方法
* GroupBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
* 第一个委托参数TSource表示数据源，返回值表示分组的关键字名称。它的返回值是IGrouping对象，它定义了Key属性用于访问分组的关键字，还有分组后的TSource对象也可以访问的到.

## ****反射****

* 反射拼接SQL语句、反射转换Json/Datatable...、序列化反序列化（ISerializable）、反射动态调用dll、反射动态配置扩展
* 序列化是把一个对象持久化到磁盘中的过程。另一个应用程序可以反序列化对象，使它的状态与序列化之前一样。
* **XML数据格式**
* 我们可以使用XMLDocument类在文档对象模型(DOM)层次结构内导航
* 也可以使用XMLReader和XMLWriter。使用XML更复杂，但可以读取更大的文件。使用XMLDocument把文档全部加载进了内存中，使用XMLReader可以逐个节点读取。
* 另一个使用XML的方式是System.Xml.Serialization，把.NET对象序列化为XML，也可以把XML反序列化为.NET对象。
* 查询XML内容时，可以使用XML标准XPath或Linq to Xml。
* 对于WCF，XML可以压缩为二进制格式。JSON也可以压缩为二进制格式(BSON) .
* 我们可以使用WSDL描述XML数据，使用Swagger描述JSON数据。
* **XmlReader，XMLDocument和XPathNavigator类**
* XmlReader和XmlWriter类提供了读写大型XML文档的快速方式。
* 常见的遍历文档的方法有Read()方法进入下一个节点。然后验证该节点是否有值(HasValue())，或者该节点是否有特性(HasAttributes())，XmlReader还可以读取强类型的数据，它有几个方法如：ReadElementContentAsDouble()，ReadElementContentAsString()
* XmlDocument类用于在.NET中读写DOM类。与XmlReader和XmlWriter不同的是XmlDocument具有读写的功能，并可以随机访问DOM树。常见的遍历方法有GetElementByTagName()获取指定元素的列表，然后调用子集的OuterXml,InnerXml,NextSibling,PreviousSibling等属性。
* XPathNavigator也可以读写XML文档，特点是可以通过XPath语句访问到指定元素。要注意的是，只用通过XMLDocument创建的对象才可以修改文件，通过XMLPathDocument创建的对象是只读的。。
* **XML序列化**
* .NET Framework为序列化提供了两个名称空间：System.Xml.Serialization和System.Xml.XmlSerializer。它包含的类可用于把对象序列化为Xml文档或者流。这也就表示把对象的公共属性和字段转换为Xml元素或属性。
* 我们可以在POCO的属性上添加XmlElement元素，来定义输出XML的名称，命名空间，类型等。如：ElementName设置XML元素的名称，Namespace设置命名空间的名称，Order设置顺序
* XmlAttribute可以将POCO的属性设置为XML的属性，并且可以设置XML属性的名称，命名空间等。

## ****常用类****

* IO类、Json.NET、Http辅助类（HttpClient、Restsharp、WebClient、WebRequest）、爬虫类、文件压缩类、条形码 、金额、时间、Assembly等类的实现
* **IO类**
* File，FileInfo类可以读取，写入，创建，删除，移动文件
* 在C# 4.0之前ReadAllLines()方法用于读取文件中所有的行，并以数组的形式返回，但是它有一个问题，就是它要等所有的行写入内存之后才能读取第一行内容。
* ReadLines()方法解决了这个问题，它返回IEnumerable<string>而不是string数组，这个新增的方法要高效很多，它不是将所有行一次性加载进内存中，而是每次读取一行。
* 如果要打开一个很大的文件，之前的方法要等整个文件都加入内存中，才能访问第一行。而这个新方法每次只读取一行，更加高效。
* Path类可以读取文件或目录的信息如：扩展名，文件，文件夹名，返回随机文件，创建临时文件等。
* 流是一个用于传输数据的对象，数据可以向两个方向传输：
* 外部源 --> 程序 (读取流)
* 程序 --> 外部源 (写入流)
* 外部源可能是一些网络协议(接口)，命名管道，内存区域
* 我们可以使用System.IO,MemoryStream对象读写内存
* System.Net.Sockets.NetworkStream对象处理网络数据
* Stream类对外部数据源不做任何假定，外部数据源可以是文件流，内存流，网络流或任意数据源。
* 一些流可以链接调用(chain)，可以先使用DeflateStream压缩数据，然后写入到FileStream,MemoryStream或NetworkStream中。
* 直接读写流不是那么容易，但是可以使用阅读器和写入器，流入StringReader和StringWriter类。这是一个关注点分离。
* 使用FileStream类读写文本文件，需要使用字节数组。有更简单的方法：使用读取器(StreamReader)和写入器(StreamWriter)，使用它们无需处理字节数组和编码，比较轻松。

## ****数据层****

* 原生的ADO.NET
  + Connections. 用于连接和管理针对数据库的事务。
  + 为了访问数据库，需要提供某些连接参数，如运行数据库的计算机和登记证书。使用SqlConnection类来连接SQLServer
  + 我们可以在https://www.connectionstrings.com/上找到各个数据库的连接字符串。
  + 连接池，现在关闭连接不会关闭与服务器的连接。相反，连接会添加到连接池中。再次打开连接，它可以从连接池中提取，因此打开连接会非常快速，只有第一次打开连接需要一定的时间。
  + Commands. 用于发出针对数据库的SQL指令。
  + 创建命令：conn.CreateCommand()
  + 赋予SQL语句：command.CommandText = sql
  + 设置参数：command.Parameter.AddWithValue(“ID”,1)
  + 执行命令：ExecuteNonQuery() ---执行命令，但不返回结果
  + ExecuteReader() ---执行命令，返回一个IDataReader
  + ExecuteScalar() ---执行命令，返回结果集中第一行第一列的数据
  + DataReaders. 用于从SQL Server数据源读取只进流的数据记录。
  + 读取数据时，Read方法在while循环中调用。
  + 访问列的时候，调用不同的GetXXX方法
  + DataSets. 用于针对结构型数据，XML数据和关系型数据的存储，远程处理和编程。
  + DataSet 由一组数据表组成
  + 
  + DataAdapters. 用于推送数据到DataSet，并针对数据库协调数据。
  + 赋予SQL语句：new SqlDataAdapter(sql,conn)
  + 填充数据集：adapter.Fill(ds)
  + 存储过程
  + 设置调用的是存储过程，而非SQL语句：command.CommandType = CommandType.StoredProcedure;
  + 将参数传出去：adapter.SelectCommand = command;
* ORM框架
  + EF
  + [Dapper：最常见的轻量级ORM框架（开源）](https://github.com/StackExchange/Dapper)
  + Nhibernate：最常见的重量级ORM框架
  + ...

## ****异步和多线程****

* Thread/ThreadPool/Task/Await/Async/Parallel...
  + 多线程
    - 多线程中的同步、并发、异常等处理
    - ...
  + TPL异步
    - 使用异步编程，方法调用时在后台运行，并且不会阻塞调用线程。
    - **异步编程模型Async Patterns Model(APM)**
    - 异步模式定义了BeginXXX和EndXXX方法。BeginXXX方法接受其同步方法的所有输入参数，EndXXX方法使用同步方法的所有输出参数，并按照同步方法的返回类型来返回结果。
    - **基于事件的异步模式Event-based Async Pattern(EAP)**
    - 基于事件的异步模式定义了一个带有“Async”后缀的方法。例如，对于同步方法DownloadString，WebClient提供了异步方法DownloadStringAsync。并且你需要编写DownloadStringCompleted事件，这个事件会在异步方法完成后调用。
    - 基于事件的异步方法优势在于易于使用。但是，在自定义类中这个模式就没有这么简单了。
    - **基于任务的异步模式Task-based Async Pattern(TAP)**
    - 该模式定义一个带有“Async”后缀的方法，并返回一个Task类型。例如WebClient提供的基于任务的异步方法DownloadStringTaskAsync，该方法返回一个Task<string>，可以用string类接收它，并使用await关键字，await关键字会解除线程(UI线程)的阻塞，完成其他任务。
    - async和await关键字只是编译器的功能。编译器会用Task类创建代码。如果不使用者两个关键字，也可以用C# 4.0和Task类的方法来实现同样的功能，只是没有那么方便。
    - async修饰符只能用于返回.NET类型的Task或void方法，以及Window运行库的IAsyncOperation。
    - Task类的ContinueWith方法定义了任务完成后就调用的代码。指派给ContinueWith方法的委托接收已完成的任务作为参数传入，使用Result属性可以访问任务返回的结果。
    - Task类定义了WhenAll和WhenAny组合器。从WhenAll方法返回的Task，是在所有传入方法的任务都完成了才会返回Task。从WhenAny方法返回的Task，是在其中一个传入方法完成了就会返回Task。

## ****数据结构****

* 算法
* 百鸡算法，贪婪思想，递归
* 线性表
* 二叉树
* 栈与队列
* **算法 (真正的编程应该是：编程=XX语言+数据结构+算法)**
* **递推思想**
* 概念：通过已知条件，利用特定关系逐步递推，最终得到结果为止，核心就是不断的利用现有信息推导出新的东西。
* 分类：“顺推”，“逆推”
* “顺推”是指通过条件推出结果
* “逆推”是指通过结果推出条件
* 例子：“斐波那契”数列，银行取钱
* **递归思想**
* 概念：递归，说白了就是直接或者间接的调用自己的一种算法。它是把求解问题转化为规模较小的子问题，然后通过多次递归一直到可以得出结果的最小解，然后通过最小解逐层向上返回调用，最终得到整个问题的解。总之递归可以概括为一句话就是：“能进则进，不进则退”。
* 三要素：
* 递归中每次循环都必须使问题规模有所缩小。
* 递归操作的每两步都是有紧密的联系，如在“递归”的“归操作时”，前一次的输出就是后一次的输入。
* 当子问题的规模足够小时，必须能够直接求出该规模问题的解，其实也就是必须要有结束递归的条件。
* 注意事项：
* 深层次的递归会涉及到频繁进栈出栈和分配内存空间，所以运行效率比较低，当问题规模较大时，不推荐使用。
* 在递归过程中，每次调用中的参数，方法返回点，局部变量都是存放在堆栈中的，如果当问题规模非常大时，容易造成堆栈溢出。
* 例子：阶乘，十进制转二进制
* **贪心思想**
* 贪心算法（又称贪婪算法）是指，在对[问题求解](https://baike.baidu.com/item/%E9%97%AE%E9%A2%98%E6%B1%82%E8%A7%A3)时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部[最优解](https://baike.baidu.com/item/%E6%9C%80%E4%BC%98%E8%A7%A3)。
* 注意事项：
* 不能保证贪心所得出的解是整体最优的。
* 不能用来求最大解和最小解问题。
* 只能求满足某些约束条件的可行解的范围。
* 例子：背包问题，找零钱
* **枚举思想**
* 将问题的所有可能的答案一一列举，然后根据条件判断此答案是否合适，保留合适的，舍弃不合适的。
* 枚举是我们在无奈之后的最后一击，那么使用枚举时我们应该尽量遵守下面的两个条件：
* 1.地球人都不能给我找出此问题的潜在规律。
* 2.候选答案的集合是一个计算机必须能够承受的。

## ****ASP.NET Mvc框架****

* **Routing路由**
* **Binding模型绑定**
  + new ChildActionValueProviderFactory(),
  + new FormValueProviderFactory(),
  + new JsonValueProviderFactory(),
  + new RouteDataValueProviderFactory(),
  + new QueryStringValueProviderFactory(),
  + new HttpFileCollectionValueProviderFactory()
  + ...
* **AOP面向切面（常用特性(Attribute)及自定义特性）**
  + 元数据
  + 验证基类System.ComponentModel.DataAnnotations.ValidationAttribute
    - System.ComponentModel.DataAnnotations.CompareAttribute
    - System.ComponentModel.DataAnnotations.CustomValidationAttribute
    - System.ComponentModel.DataAnnotations.DataTypeAttribute
    - System.ComponentModel.DataAnnotations.MaxLengthAttribute
    - System.ComponentModel.DataAnnotations.MinLengthAttribute
    - System.ComponentModel.DataAnnotations.RangeAttribute
    - System.ComponentModel.DataAnnotations.RegularExpressionAttribute
    - System.ComponentModel.DataAnnotations.RequiredAttribute
    - System.ComponentModel.DataAnnotations.StringLengthAttribute
    - System.Web.Security.MembershipPasswordAttribute
  + .
  + Filter
    - IAuthenticationFilter(验证过滤器 MVC5)验证是否合理请求，是否合理用户
    - IAuthorizationFilter(授权过滤器)
      * AuthorizeAttribute
      * RequireHttpsAttribute
      * ValidateAntiForgeryTokenAttribute
      * ChildActionOnlyAttribute
      * ...
    - IActionFilter(动作过滤器-前-后)
      * ActionFilterAttribute
      * ...
    - IResultFilter(结果过滤器-前-后)
      * ActionFilterAttribute
      * ...
    - IExceptionFilter(例外过滤器)
      * HandleErrorAttribute(在FilterConfig就已被注册)
      * ...
* **RazorViewEngine视图引擎**
* **WebApi**

## ****数据库技术****

* 数据库服务端、客户端、图形界面管理工具、数据库语言（PL-sql,T-sql,sql）、数据库安全等概念
* 视图、存储过程、游标、中间表、触发器、计划任务等常用技术
* 数据库分库、拆表、分布式、集群、备份、故障调控等方案
* 数据库三大范式与反范式
* RBAC（Role-Based Access Control，基于角色的访问控制）权限模型
* 数据库机制
  + **索引**
  + **存储引擎**
  + **事务**
    - 四大特性(ACID)
      * 原子性 Atomicity
      * 一致性 Consistency
      * 隔离性 Isolation
      * 持续性 Durability
    - 并发产生的问题
      * 脏读
      * 幻读
      * 不可重复读
      * ...
    - 解决方案—隔离级别
      * 读未提交Read uncommitted
      * 读已提交Read committed (mssql、oracle)
      * 可重复读Repeatable read（mysql）
      * 串行化Serializable
  + **锁机制**
    - 种类、粒度、加锁方式
    - 悲观锁:开销大，避免长事务
      * 表锁、行锁/读锁、写锁
    - 乐观锁:开销小，读大于写
      * Version版本号机制
    - ...
* 数据库辅助工具：Power Designer、EZDML、ER Studio、Rose、Microsoft Visio...

## ****关系型数据库****

* Oracle
  + 最贵，功能最多，不开源，最安全...
* Mysql
  + 免费，功能一般，开源，相对不安全...
* MsSql(SqlServer)
  + 中等贵，支持平台最少，不开源，使用最方便...
* PostgreSQL
  + 与其他关系型数据库相比，PostgreSQL独特的地方是它支持高度需要的、完整的面向对象以及关系型数据库的功能
* Sqlite
  + 一个自包含的、基于文件的数据库，可移植性非常好
* ...

## ****非关系型数据库(Nosql-Not Only Sql)****

* MongoDb：是一个基于分布式文件存储的数据库
* Hbase：属于hadoop(分布式系统)生态圈的组件，能存储海量数据
* Cassandra：优异的列式存储NoSQL
* Redis：基于内存的数据存储系统，支持多种类型存储，适用于高并发场合
* Memcache：基于内存的数据存储系统，高性能分布式内存缓存服务，适用于高并发场合
* MongoDB
* **MongoDB的安装**
* 首先下载Linux x64的安装包: <https://www.mongodb.com/download-center#atlas>
* 将安装包放到/usr目录下解压mongodb
* 在mongodb的根路径下创建一个data文件夹，data里在创建一个db文件夹
* 在mongodb的bin目录下打开终端执行：./mongod --dbpath=../data/db启动服务
* 然后我们测试一下是否可以使用MongoDB吧，在bin目录另外打开终端执行：mongo 会看到当前MongoDB的版本。
* 运行mongo 打开客户端，或者使用使用管理工具如robo
* **MongoDB与SQL的对应表**

|  |  |
| --- | --- |
| **SQL** | **MongoDB** |
| Database | database |
| Table | Collection |
| Row | Document |
| Column | Field |
| Index | Index |
| Table join | $lookup,embedded document |
| Primary key | Primary key |
| Aggreagation(聚合) | Aggregation pipeline |

**关系运算符**

等于 => $eq 语法结构:{field:{$eq:value}}

不等于 => $ne 语法结构:{field:{$ne:value}}

大于 => $gt 语法结构:{field:{$gt:value}}

大于等于 => $gte 语法结构:{field:{$gte:value}}

小于 => $lt 语法结构:{field:{$lt:value}}

小于等于 => $lte 语法结构:{field:{$lte:value}}

包含 => $in 语法结构:{field:{$in:[value1,value2,...]}}

不包含 => $nin 语法结构:{field:{$nin:[value1,value2,...]}}

**逻辑运算符**

并且 => $and 语法结构:{$and:[ Expression1, Expression2, ...]}

或者 => $or 语法结构:{$or:[ Expression1, Expression2, ...]}

或非 => $nor 语法结构:{$nor:[ Expression1, Expression2, ...]}

非 => $not 语法结构:{field:{$not:{operation-exrpession}}}

**Expression例子：{price:{$eq:10}}**

**operation-exrpession例子:{$eq:10}**

**元素运算符**

是否存在 => $exists 语法结构:{field:{$exist:<boolean>}}

是否为该类型 => $type 语法结构:{field:{$type:<BSON type>}}

**评估运算符**

求模 => $exists 语法结构:{$field:{$mod:[除数，余数]}}

正则表达式 => $regex 语法结构:{field:{$regex:/pattern/<options>}}

where条件 => $where 语法结构:{$where:function{ return <Boolean>; }}

**options枚举：**

**i => 忽略大小写**

**m => 匹配多条结果**

**x => 忽略所有的空格**

**s => .可以匹配所有的包扩新行在内的所有字符**

**数组运算符**

全部条件符合 => $all 语法结构:{field:{$all:[value1，value2， ...]}}

任意条件符合 => $elemMatch 语法结构: {field:{$elemMatch:[query1，query2，...]}}

数组长度符合 => $size 语法结构: {field:{$size:<int32>}}

**query例子：{$gt : 100} 或者 {$eq: ‘A’ }**

**列修改运算符**

加法 => $inc 语法结构:{$inc:{field1:<amount>},{field2:<amount>},...}

乘法 => $mul 语法结构:{$mul:{field1:<number1>,...}}

改名称 => $rename 语法结构 : {$rename:{<field1>:<NewName1>,<field2>:<NewName2>,...}}

改值 => $set 语法结构 :{$set:{<field1>:<value1>,<field2>:<value2>,...}}

详解：如果修改的值不存在则新增，存在则修改

无法修改时则添加 => $setOnInsert语法结构：{$setOnInsert:<field1>:<value1>,...},{upsert:true}

详细解释: 如果update的更新参数upsert:true，也就是如果要更新的文档不存在的话会插入一条新的记录，$setOnInsert操作符会将指定的值赋值给指定的字段，如果要更新的文档存在那么$setOnInsert操作符不做任何处理；

删除列 => $unset 语法结构：{$unset:{<field1> :“”,... }}

求最小值 => $min 语法结构：{$min:{<field1> : <value1>,...}}

求最大值 => $min 语法结构：{$max:{<field1> : <value1>,...}}

求当前时间 => $currentDate：{$currentDate:{<field>:}}

**数组修改运算符**

首个元素的占位符 => $ 语法结构:{‘<Array>.$’:value}

例子：db.students.updateOne({\_id:1,'grades':80},{$set:{'grades.$':90}});

这句命令的意思是修改id等于1并且grades包含80的数组，将grades里的第一个元素改成90

所有元素的占位符 => $[] 语法结构:{ <update operator>: { "<array>.$[]" : value } }

例子：db.students.update({ \_id:4},

{$inc:{'grades.$[].grade':100} //让所有的grade+100},

{multi:true});

这句命令的意思是，修改\_id等于4的数组，将grades里所有的grade加100

添加元素 => $addToSet 语法结构: {$addToSet:{field1:value1,.....}}

如果不存在则添加，如果存在则不添加。

出栈 => $pop 语法结构: {$pop:{<field>:<-1 | 1>,......}}

例子：{ \_id: 1, scores: [ 8, 9, 10 ] } //元数组

db.students.update( { \_id: 1 }, { $pop: { scores: -1 } } ) //移除第一个元素

{ \_id: 1, scores: [ 9, 10 ] } //出栈后

当field的值为1时移除最后一个元素，为-1时移除第一个元素

删除 => $pull 语法结构:{ $pull: { <field1>: <value|condition>, <field2>: <value|condition>, ... } }

删除数组中所有匹配条件的元素

例子：db.stores.update({},

{$pull:{fruits:{$in:['bananas','oranges']}}},

{multi:true //别忘了加上这个，要不然只能修改一行});

这句命令的意思是删除数组中包含bananas和oranges的元素

**CRUD操作**

新增操作有三个命令：insert,insertOne,insertMany

Insert可以插入一个Document或者Array

InsertOne只能插入一条Document

InsertMany可以插入一个Array

读取操作最简单的命令是find，输入条件可以查询所有数据.

其次还有findAndModify，由此引申出findOneAndDelete,findOneAndUpdate,

findOneAndReplace，这些操作实现了一系列事务操作。

修改操作有四个命令：update，updateOne，updateMany和修改行数插入命令类似。ReplaceOne和findOneAndReplace类似。

删除操作有三个命令：remote，deleteOne，deleteMany

Remote可以删除一行或多行，deleteOne只能删除一行，deleteMany可以删除多行。

**索引操作**

MongoDB中所多种类型的索引：

单键索引： Single Field Index

复合索引： Compound Index

**分布式文件系统**

MongoDB内置一套文件系统名为GridFS（Grid File System）

我们可以使用它来存储大于16M的文件。

它具有多机器存储备份，可以突破一般文件系统对file的限制，分段存储，不像普通file system是整个存储的。这样读取大型文件时就不会占用大量的内存。

支持多种驱动，如Java，C#等

GridFS会将文件存储在两个Document里：

1. Chunks用来存储二进制数据
2. Files用于存储基本文件信息

具体的命令我们可以参考以下链接：

<https://docs.mongodb.com/manual/reference/program/mongofiles/index.html>

其中最简单的包括：

存储文件：mongofiles –d <database> put <filename>

删除文件：mongofiles –d <database> delete <filename>

获取文件：mongofiles –d <database> -l <new\_filename> get <filename>

常用的命令还有：

--host <hostname><:port>

--port <port>

**聚合操作**

MongoDB中有一个聚合工具：Aggregate，类似于SQL语句中的Group By。聚合工具可以让开发人员直接使用MongoDB的原生的命令操作数据库中的数据，并按要求进行聚合，聚合大幅提高了开发效率。

Aggregate PipeLine指的是命令将会按照顺序依次执行，并且管道操作可以重复，它的工作原理如图所示：



Aggreagte是一个数组，其中包含多个对象（命令），通过遍历PipeLine数组对Collection中数据的操作

你可以调用一下的阶段（stages）：

$lookup：相当于左外连接，它的语法如下：

{

$lookup:

{

From:<外联接的集合 >,

localField:<连接的列名>,

foreignField:<外联接的列名>,

as:<输出的列名>

}

}

$match：相当于过滤条件，它的语法如下：

{ $match :{<query>}}

$group：可以分组来计算出结果

{

$group:

{

\_id:<expressing>,

<field1>:{<accumulator1>:<expression1>},

……

}

}

具体解释：

Expression：我们可以使用“$属性名”或“$属性名.属性名….”来表示一个字段

Accumulator：累计运算符有很多中，常见的有$avg,$first,$last,$max,$min,$sum等。

$project：标记要显示的列名

{

$project:

{

\_id:0 or 1,

<field>:0 or 1,

}

}

## ****面向服务架构SOA****

* 通过服务整合来解决系统集成的一种思想
* .
* WSDL
  + Web Services Description Language是一个用来描述Web服务和说明如何与Web服务通信的XML语言。为用户提供详细的接口说明书
* UDDI
  + Universal Description，Discovery and Integration统一描述、发现和集成, 提供一种发布和查找服务描述的方法。UDDI 数据实体提供对定义业务和服务信息的支持。
* SOAP
  + Simple Object AccessProtocol，简单对象访问协议，基于XML 和 HTTP 用于在应用程序之间进行通信的一种通信协议
    - Web Services：基于SOAP协议，数据格式是XML
    - Wcf/Wcf Rest：可以不依赖于IIS，基于SOAP，支持多种通信协议，但配置繁琐
  + 测试工具：SoapUI...
* REST
  + REpresentational State Transfer ，表现层状态转移，是一种架构风格，提供了设计概念原则和约束。
  + RESTful
    - 满足这些原则和条件的就称RESTful架构
    - Web API：RESTful的实现，一个用于构建HTTP服务的框架
  + 测试工具：Fiddler、Postman、Jmeter...
* 微服务
  + 是SOA的一种实现，更侧重于组件化和服务化
* JSON-RPC
  + Remote Procedure Call Protocol，远程过程调用协议，高性能二进制协议，比RESTful要高效（类似于Web Services的使用风格）

## ****面向对象OOP****

* 三大基本特性
  + 继承，封装，多态
    - 抽象类abstract和接口interface的各种用法及实现
    - Virtual、abstract、override、New、extern等关键字的含义及用法
    - 设计模式的应用
    - ...
* 六大基本原则（高内聚，低耦合）
  + 单一职责原则SRP(Single Responsibility Principle)
  + 开放封闭原则OCP(Open－Close Principle)
  + 里式替换原则LSP(the Liskov Substitution Principle)
  + 依赖倒置原则DIP(the Dependency Inversion Principle)
  + 接口分离原则ISP(the Interface Segregation Principle)
  + 最少知识原则LKP(Least Knowledge Principle）（也称：迪米特原则）
* 23种设计模式
  + 创建型
  + 结构型
  + 行为型

## ****缓存及优化技术****

* 浏览器自身缓存技术(cache control,与服务器约定,js /css等缓存)
* Session(SessionId)(会话,为了弥补http协议的无状态特性)
* Cookie(浏览器存储,为了弥补http协议的无状态特性)
* Localstorage(Html5新特性 浏览器本地存储,为了弥补Cookie存储不足)
* Application(服务器全局变量)
* Static(依赖于IIS，进程回收会丢失)
* Cache(服务器缓存)
* 为了更清楚的了解，我们总结出每一种对象应用的具体环境，如下表所示：
* 
* **浏览器缓存**
* 概念：简单来说，浏览器缓存就是把一个已经请求过的Web资源（如html页面，图片，js，数据等）拷贝一份副本储存在浏览器中。缓存会根据进来的请求保存输出内容的副本。当下一个请求来到的时候，如果是相同的URL，缓存会根据缓存机制决定是直接使用副本响应访问请求，还是向源服务器再次发送请求。比较常见的就是浏览器会缓存访问过网站的网页，当再次访问这个URL地址的时候，如果网页没有更新，就不会再次下载网页，而是直接使用本地缓存的网页。只有当网站明确标识资源已经更新，浏览器才会再次下载网页。至于浏览器和网站服务器是如何标识网站页面是否更新的机制，将在后面介绍。
* 报文头里的一些关键信息：
* Expires：Http1.0 中的标准，表明过期时间，注意此处的时间都是指的是服务器的时间。
* Cache-Control：Http1.1 中的标准，可以看成是 expires 的补充。使用的是相对时间的概念。
* Cache-Control的属性：
* max-age: 设置缓存的最大的有效时间，单位为秒（s）。max-age会覆盖掉Expires
* s-maxage: 只用于共享缓存，比如CDN缓存（s -> share）。与max-age 的区别是：max-age用于普通缓存，  
  而s-maxage用于代理缓存。如果存在s-maxage,则会覆盖max-age 和 Expires.
* public：响应会被缓存，并且在多用户间共享。默认是public。
* private: 响应只作为私有的缓存，不能在用户间共享。如果要求HTTP认证，响应会自动设置为private。
* no-cache: 指定不缓存响应，表明资源不进行缓存。但是设置了no-cache之后并不代表浏览器不缓存，而是在缓存前要向服务器确认资源是否被更改。因此有的时候只设置no-cache防止缓存还是不够保险，还可以加上private指令，将过期时间设为过去的时间。
* no-store: 绝对禁止缓存。
* Last-modified: 表明请求的资源上次的修改时间。
* If-Modified-Since：客户端保留的资源上次的修改时间。
* Etag：资源的内容标识。（不唯一，通常为文件的md5或者一段hash值，只要保证写入和验证时的方法一致即可）
* 所以缓存的调用步骤是：  
  1） 查看是否有cache-control 的max-age / s-maxage , 如果有，则用服务器时间date值 + max-age/s-maxage 的秒数计算出新的过期时间，将当前时间与过期时间进行比较，判断是否过期  
  2）如果没有，则用expires 作为过期时间比较
* 3）如果过期了，则请求服务器，服务器接收到请求之后，通过Last-Modified和Etag判断文件是否修改过，如果没有修改过则发送一个304到客户端，告诉浏览器直接从自己本地的缓存取数据；如果修改过那就整个数据重新发给浏览器。
* 
* 参考资料：
* <https://www.cnblogs.com/slly/p/6732749.html#undefined>
* <https://www.cnblogs.com/shixiaomiao1122/p/7591556.html>
* <https://segmentfault.com/a/1190000011212929>
* 具体实现：
* <meta http-equiv="Expires" content="0"/>
* <meta http-equiv="Cache-Control" content="no-cache"/>
* **Session会话**
* 1、 InProc Session Provider是最快的，因为所有数据都存在应用程序的内存里，Session数据在IIS重启，或者站点被回收的情况下丢失，你可以在用户量较小的网站上使用这种模式，但别在Web Farm下使用。
* 2、 State Server模式：Session数据被存储于aspnet\_state.exe应用中，他在Web服务之外保存Session数据，所以Web服务出现问题不会对他的Session数据造成影响，在将Session数据存储到StateServer之前需要序列化对象，在Web Farm中我们能安全地使用这个模式。
* 3、 SQL Server模式：他将Session数据保存到SQL Server中，我们需要提供连接串，我们存储时也需要对对象进行序列化，这种模式在实际Web Farm的生产环境中是非常有用的。
* 4、 Custom模式，当我们需要使用一个已经存在的表来存储Session数据时，在自定义模式中，我们也能创建自定义的Session ID，但是不推荐你自己来实现Provider，推荐使用第三方的Provider。
* 参考资料：
* <https://www.cnblogs.com/ideacore/p/6423281.html>
* [ASP.NET性能优化之分布式Session](http://www.cnblogs.com/luminji/archive/2011/11/03/2195704.html)
* **Cookie机制**
* Cookie的作用
* 比如你登录录邮箱，登录一个页面。我们经常会在此时设置30天内记住我，或者自动登录选项
* Cookie的分类
* cookie分为会话cookie和持久cookie，会话cookie是指在不设定它的生命周期expires时的状态，前面说了，浏览器的开启到关闭就是一次会话，当关闭浏览器时，会话cookie就会跟随浏览器而销毁。当关闭一个页面时，不影响会话cookie的销毁。会话cookie就像我们没有办理积分卡时，单一的买卖过程，离开之后，信息则销毁。
* 持久cookie则是设定了它的生命周期expires，此时，cookie像商品一样，有个保质期，关闭浏览器之后，它不会销毁，直到设定的过期时间。对于持久cookie，可以在同一个浏览器中传递数据，比如，你在打开一个淘宝页面登陆后，你在点开一个商品页面，依然是登录状态，即便你关闭了浏览器，再次开启浏览器，依然会是登录状态。这就是因为cookie自动将数据传送到服务器端，在反馈回来的结果。持久cookie就像是我们办理了一张积分卡，即便离开，信息一直保留，直到时间到期，信息销毁。
* 参考资料：
* <https://blog.csdn.net/u014753892/article/details/52821268>
* LocalStorage和SessionStorage
* HTML5 提供了两种在客户端存储数据的新方法：
* localStorage - 没有时间限制的数据存储
* sessionStorage - 针对一个 session 的数据存储
* 之前，这些都是由 cookie 完成的。但是 cookie 不适合大量数据的存储，因为它们由每个对服务器的请求来传递，这使得 cookie 速度很慢而且效率也不高。
* 在 HTML5 中，数据不是由每个服务器请求传递的，而是只有在请求时使用数据。它使在不影响网站性能的情况下存储大量数据成为可能。
* 对于不同的网站，数据存储于不同的区域，并且一个网站只能访问其自身的数据。
* HTML5 使用 JavaScript 来存储和访问数据。
* 参考网址：
* <http://www.w3school.com.cn/html5/html_5_webstorage.asp>
* Application
* Application相当于应用程序的“全局变量”，如图所示：
* 
* 在这张图中，Web服务器中运行的Web应用程序就是我们所说的Application，每个客户端与Web服务器之间建立的连接就可以看做是一个Session。比如现在服务器端运行的是一个论坛系统，那么现在这个正运行在服务器端的论坛系统的软件就可以看做Application，而每个在线的用户与之建立的连接就相当于一个Session。
* 如果要使用Application要注意的是所有的写操作都要在Application\_OnStart事件中完成（global.Asax），尽管可以使用Application.Lock()避免了冲突，但是它串行化了对Application的请求，会产生严重的性能瓶颈；
* 参考网址：
* <https://blog.csdn.net/lxlj2006/article/details/5608281>
* <https://www.cnblogs.com/Arlar/p/5934651.html>
* Cache
* 在.NET运用中经常用到缓存(Cache)对象。有HttpContext.Current.Cache以及HttpRuntime.Cache，HttpRuntime.Cache是应用程序级别的，而HttpContext.Current.Cache是针对当前WEB上下文定义的。HttpRuntime下的除了WEB中可以使用外，非WEB程序也可以使用
* 在可以的条件，尽量用 HttpRuntime.Cache ，而不是HttpContext.Cache
* 合理的使用缓存

1. 读写小于10:1的情况下，不适合用缓存，我们用缓存的目的就是想分摊下数据库的压力以及利用内存来提速性能，如果读写差不多，或者压根就没读过，这样的死数据就会造成内存资源的浪费。
2. 既然是缓存，就注定了它的资源是有限的，宝贵的，也就注定了我们必须合理利用它的内存空间，也就被迫的让我们清楚的认识到热点数据，不易修改的应该放在缓存，反之不宜放。
3. 大公司在缓存方面做的好的地方就是在一个“控”字上，他们会为缓存专门做一套“缓存系统”，当系统预加载的时候，同时也充当内存数据库使用，将这些元数据加载到缓存系统中，比如“县市区”，“分类信息”等等作为预热数据。

* 局部缓存

## ****软件架构****

* MVC
  + ASP.NET MVC
* MVP
  + 事件模型
    - ASP.NET Web form
      * System.UI.Page
      * ViewState
      * ...
    - Winform
* MVVM
  + 在.Net中(主要应用于WPF、Silverlight、WP7)
    - Prism - Brian Lagunas微软产品经理，微软MVP,一个跨平台的桌面和移动MVVM开发框架，开源。
    - Catel - Catel是一个专注于MVVM（WPF，Silverlight，Windows Phone和WinRT）和MVC（ASP.NET MVC）的应用程序开发平台。 Catel的核心包含一个IoC容器，模型，验证，纪念，消息中介，参数检查等。
    - ...
  + 在前端中
    - Angular
    - Vue
    - Knockout
    - React
    - ...

## ****平台****

* **MONO**
* **.Net Core**
  + CoreCLR
  + CoreRT for .NET Core
  + .NET Core Base Class Library
  + ...
* **.Net Framework**
  + .NET Framework Base Class Library(BCL基类库)
    - System.Web（ASP.NET的核心）
      * ASP.NET与IIS管道模型：
        + 经典模式IIS5/IIS6：

IIS管道模型

ASP.NET管道模型

...

* + - * + 集成模式IIS7：

HttpContext

...

* + - * + IIs的常用功能及使用及配置：

请求筛选、失败请求跟踪等

* + - * + .Net MVC生命周期
        + Httphandler、HttpModule、Globar的编写及应用...
        + ...
    - OWIN解耦
      * Open Web Server Interface for .NET，定义了一种中间件在管道中用于处理请求和相关响应的标准方式，是种规范，用于解耦Web 服务器和Web 应用程序。
      * OWIN的实现：Katana（由微软早期开发的开源OWIN组件集合）
        + OWIN Layer(定义的一系列层)

Application(应用程序)

Middleware(中间件)

Nanacy.Owin

WebApi

Signalr

FubuMVC

Simple.Web

DuoVia.Http

...

Server(服务)

Microsoft.Owin.Host.SystemWeb

Microsoft.Owin.Host.HttpListener

Helios

(Asp.Net5以后就是Microsoft.AspNet命名空间)

...

Host(宿主)

IIS

Custom Host

OwinHost.exe

Node.Js(connect-owin)

...

* + .NET CLR（Commen Language Runtime）公用语言运行时
    - 核心：公共语言基础（Common Language Infrastructure，CLI）,提供了定义的规范接口、与工具
    - 1.项目源代码被 .Net Framework编译器( csc , C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe )编译成EXE或者是DLL文件。
    - 2.被CSC编译后的EXE/DLL文件叫做 IL文件。Intermediate Language （IL）中间语言，在.Net中，称之为 Microsoft IL（MSIL）微软中间语言(右键对项目进行生成的时候实际上就是这一步)
    - 3.CLR会将MSIL通过JIT（Just In Time）编译器进行第二次编译，将其编译成本地平台的CPU指令。(如果拥有能在Linux运行的CLR，代码就能跨平台运行)
    - .
    - C#源码——(CSC编译器)——MSIL文件(dll/exe)——(CLR中的JIT编译器)——CPU执行

## ****Http协议****

OSI协议与TCP/IP协议



TCP/IP协议



* OSI网络通信
  + 物理层
    - 以二进制数据形式在物理媒体上传输数据
  + 数据链路层
    - 传输有地址的帧，错误检测功能
  + 网络层
    - 为数据包选择路由
    - IP，ICMP，RIP...
  + 传输层
    - 提供端对端的接口
    - TCP、UDP...
  + 会话层
    - 建立或解除与其它接点的联系
  + 表示层
    - 数据格式化，代码转换，数据加密
  + 应用层
    - HTTP，FTP，SMTP...
* Http状态码
  + 1xx:信息提示，表示请求已被成功接受，继续处理，请求范围100~101
  + 2xx:成功，服务器成功响应请求，请求范围200~206
  + 3xx:重定向，请求范围300~305
  + 4xx:发送一些服务器无法受理的东西
  + 5xx:一条有效请求，但服务器挂了
* 常用HTTP消息头(Http Header)
  + Request请求头
    - Authorization：Basic base64646464646464=== （验证）
    - Accept-Encoding：gzip （指定浏览器可以支持的web服务器返回内容压缩编码类型）
    - Cache-Control：max-age=60（指定请求和响应遵循的缓存机制，max-age为 当前60秒内再次访问不会去服务器请求）
    - Cookie：ZJF（该请求域名下的所有cookie值一起发送给web服务器）
    - Content-Type：application/x-www-form-urlencoded（是以什么格式来传输，请求的与实体对应的MIME信息）
    - Accept：text/plain（指定客户端能够接收的内容类型）
    - Range：bytes=0-999（常用作断点续传，设置请求的指定范围，服务器则返回Content-Range: bytes 0-999）
    - Referer： （当前请求来源的网页地址）
    - Origin： （和Referer差不多，只存在于POST请求）
    - Host:： （指定请求的服务器的域名和端口号）
    - User-Agent:：Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36（当前客户端的信息）
    - Expect：100-continue（握手失败，请求头域类型不匹配，由此导致的API接口调用失败）
  + Response返回头
    - Connection: keep-alive（连接方式，很重要）
    - Access-Control-Allow-Origin: \*（跨域，很重要）
    - Set-Cookie: UserID=admin; Max-Age=3600; Version=1
    - Content-Encoding:gzip

**表单提交与成功控件**

浏览器并不是将所有的表单控件全部发送到服务器的，而是会查找所有的【**成功控件**】，只将这些成功控件的数据发送到服务端， 什么是成功控件呢？  
简单地来说，成功控件就是：每个表单中的控件都应该有一个name属性和”当前值“， 在提交时，它们将以 name=value 的形式做为提交数据的一部分。

对于一些特殊情况，成功控件还有以下规定：  
1. 控件不能是【禁用】状态，即指定【disabled="disabled"】。即：禁用的控件将不是成功控件。  
2. 如果一个表单包含了多个提交按键，那么仅当用户点击的那个提交按钮才算是成功控件。  
3. 对于checkbox控件来说，只有被用户勾选的才算是成功控件。  
4. 对于radio button来说，只有被用户勾选的才算是成功控件。  
5. 对于select控件来说，所有被选择的选项都做为成功控件，name由select控件提供。  
6. 对于file上传文件控件来说，如果它包含了选择的文件，那么它将是一个成功控件。  
此外，浏览器不会考虑Reset按钮以及OBJECT元素。

控件输入的内容并不是直接发送的， 而是经过一种编码规则来处理的。目前基本上只会只使用二种编码规则：application/x-www-form-urlencoded 和 multipart/form-data ， 这二个规则的使用场景简单地说就是：后者在上传文件时使用，其它情形则使用前者(默认)。

这个规则是在哪里指定的呢？ 其实form还有个enctype属性，用它就可以指定编码规则

浏览器处理表单数据的过程大致分为4个阶段：  
1. 识别所有的成功控件。  
2. 为所有的成功控件创建一个数据集合，它们包含 control-name/current-value 这样的值对。  
3. 按照form.enctype指定的编码规则对前面准备好的数据进行编码。编码规则将放在请求中，用【Content-Type】指出。  
4. 提交编码后的数据。此时会区分post,get二种情况，提交的地址由form.action属性指定的。

* 实时通信方案
  + .Net Signalr+
  + Ajax轮询
  + Comet
  + WebSocket(SuperWebSocket，WebSocket服务器)
  + SSE（Server-sent Events服务器发送事件）
  + ...

## ****域名解析****

* DNS解析流程
* 常见 域名解析记录
  + A记录：用来指定域名或子域名对应的IP地址记录，保证域名指向对应的主机
  + CNAME记录：别名记录，该记录允许你为自己的主机设置别名
  + MX记录：邮件路由记录，当发送邮件时，Mail 服务器先对域名进行解析，查找mx记录
  + TXT记录：是域名系统(DNS)中由域名托管服务商存储的一种资源记录，其作用是防止垃圾邮件滥用行为
  + AAAA记录：指定网站域名对应的IPv6地址记录
  + NS记录：Name Server，用于指定域名是由那个DNS服务器来进行域名解析
  + ...

## ****网站优化****

* JavaScript和Css模块化
* CSS Sprites：将一个页面涉及到的所有零星图片都包含到一张大图中去，这样一来，将N次请求变成一次请求
* 划分资源域
* 合理运用缓存技术
  + 静态文件（图片、css、js、html ）缓存： cdn技术、squid缓存服务器、浏览器(客户端)缓存
  + HTML5 application cahce API离线缓存
  + 应用层缓存：Cache、Redis缓存数据库、memcache缓存数据库
  + 数据库缓存：数据库本身的缓存机制
  + ...
* 了解浏览器工作机制：Css放顶部，Js放底部，压缩Html，能配置Expires/Cathe-Control，ETag，使用 gzip 压缩
* 用ISP(互联网服务提供商)提供的文件存储等技术
* 提高DNS域名解析速度
* 掌握SEO
  + 确保网站是可以被索引的： robots.txt，Robtots 标签
  + 确保网页是可以被索引的：确保爬虫可以看到所有的内容、使用描述性的锚文本的网页
  + 擅长标签编写：带有 H1 标签、关键词放 URL 中、加粗文字、Descripiton 标签、图片文件名、ALT 属性带有关键词
  + ...
* 删除不必要的HTTP响应头
  + 移除X-AspNet-Version HTTP头
  + 移除X-AspNetMvc-Version HTTP头
  + 移除X-Powered-By HTTP头
  + 移除Server HTTP头
  + ...
* 用户数据分析
  + PV IP UV
  + 访问流量来源
    - 浏览器
    - 移动设备
    - ...
* ...

## ****默认隐藏段（拒绝客户端访问）****

* App\_Browsers：包含浏览器定义(.browser文件),ASP.NET会使用这些文件来识别个别浏览器并判断它们的功能
* App\_Code：包含用于公用程序和商务对象(Business Object) , (例如, .cs ; .vb和.jsl文件)的程序源代码,要将它编译为应用程序的一部分
* App\_Data：包含应用程序数据文件.包括MDF文件,XML文件与其他数据库文件
* App\_GlobalResources：包含资源(.resx和.resources文件),这些资源会编译成具有全局范围的组件
* App\_LocalResources：包含资源(.resx和.resources文件),这些资源会与特定的页面,拥护控件或应用程序的主页面(MasterPage)相关联
* App\_WebReferences：包含参考合约文件(.wsdl文件),结构描述(.xsd文件)和探索文件(.disco和.discomap文件),可定义Web应用以用语应用程序
* Bin：包含空间,组件或你要在应用程序中应用其他程序代码的已编译组件(.dll文件).在[Bin]文件夹中以程序代码表示的任何类, 都会自动在应用程序中应用到
* Web.config：用来储存Asp.net Web应用程序的配置信息，通过继承关系，每个Web.config将配置设置应用到它所在的目录及虚拟子目录下
* ...

## ****特殊目录****

* Bin：是二进制binary的英文缩写,用来存放编译的结果
  + Debug 调试版本 开发环境 有调试信息
  + Release 发行版本 代码更小,执行更快,编译更严格,更慢 没有调试信息
* Obj：是object的缩写，用于存放编译过程中生成的中间临时文件,用来加快编译速度
  + Debug
  + Release

## ****需要掌握的前端知识****

* Html5+Css3
* JQuery插件语法
* Js实现封装-继承-多态
* 实现一个简单的MVVM框架
* 实现一个简单的js模板引擎
* Angular
  + 路由、模块化、控制器、指令、作用域、模板、链式函数、过滤器、服务、依赖注入...
* React
  + 虚拟DOM，组件化...
* ...

## ****信息加密技术：****

* 单向散列加密（杂凑函数、Hash函数）
  + 把任意长的输入消息串变化成固定长的输出串且由输出串难以得到输入串的一种函数，用于产生消息摘要，密钥加密
  + MD5，SHA，MAC，CRC...
* 对称加密
  + 使用的密钥(yue)只有一个，发收信双方都使用这个密钥对数据进行加密和解密
  + DES，3DES，AES，TDEA，Blowfish，RC5，IDEA...
* 非对称加密（公私钥加密）
  + 指加密和解密使用不同密钥的加密算法，分公钥和私钥
  + RSA，Elgamal，背包算法，Rabin，D-H，ECC...

## ****SSL****

* Secure Socket Layer，安全套接字层,保护Web通讯,以实现客户端和服务器之间的安全通讯
* HTTPS：是在HTTP上建立SSL加密层，并对传输数据进行加密，是HTTP协议的安全版
  + Let's Encrypt,国外一个公共的免费SSL
  + StartSSL,被各大浏览器所支持,但只免费一年
  + ...

## ****TLS****

* Transport Layer Security，安全传输层协议，用于在两个通信应用程序之间提供保密性和数据完整性，建立在SSL 3.0协议规范之上，是SSL 3.0的后续版本

## ****Web安全****

* 渗透注入
  + Sql注入、命令表达式注入、SSRF注入、XSS注入、CSRF注入、XXE注入...
* 渗透注入检测工具
  + DirBuster目录渗透工具，专门用于探测Web服务器的目录和隐藏文件
  + Nmap网络连接端软件，网络连接端扫描软件，用来扫描网上电脑开放的网络连接端
  + Pangolin Sql注入工具
  + AppScan业界领先的web应用安全监测工具（软件界面可以选择中文，不过是收费的）
  + ...
* 欺骗
  + ClickJacking（点击劫持）
    - 透明iframe
      * HTTP头——X-Frame-Options
    - 图片掩盖接口
* 暴力破解
* DDOS
* 文件安全
  + 配置文件、备份文件暴露读取、未授权修改、未授权上传
* 逻辑安全
  + 未授权访问、上下流程顺序可乱序、无限递归
* 二进制安全
  + （VS中有性能分析工具）
  + 堆栈溢出(很多人都碰过)、缓存区溢出、内存泄漏

## ****Web认证****

* 第三方认证
  + Oauth2.0
  + Json Web Token
  + SSO单点登录
    - 同源策略
    - CAS解决方案
* IIs/.Net认证
  + Basic
  + Digest Authentication
  + Windows
  + Forms

## ****解决方案****

* 代码管理
  + Git：免费、开源的分布式版本控制系统，用的人最多
  + Svn：集中式的版本控制系统，一个强大的备份服务器
  + TFS：微软源代码管理工具
* 自动化测试
  + UI测试
    - VS自带的UI测试工具
    - ...
  + 单元测试(Unit Testing)
    - Tdd测试驱动开发（Test-Driven Development）
      * VS自带的单元测试工具，Xunit，Nunit，Moq，Faker.Net
  + 性能分析
    - VS自带的性能分析工具，MiniProfiler
  + 压力测试
    - 概念
      * 吞吐率(Throughput)
      * QPS每秒查询率(Query Per Second)
      * TPS每秒事务数(Transaction Per Second)，并发用户数
      * ...
    - VS自带的压力测试工具，HttpTest4Net，Jmeter...
* 安全检测
* 自动化部署
  + VS的Web Deploy，Jenkins...
* 监控平台
* 自动化故障调控
* 命名规范
* **命名规范**
* 1.匈牙利命名：
* 开头字母用变量类型的缩写，其余部分用变量的英文或英文的缩写，要求单词的第一个字母大写。
* Ex：
* Int iAge = 10; //i是int的缩写
* Float fPrice = 5999.9; //f是float的缩写
* 其他：   
  前缀类型 a b by c cb cr cx,cy dw fn h i l lp m\_ n np p s sz w （一一对应关系）
* 数组 (Array) 布尔值 (Boolean) 字节 (Byte) 有符号字符 (Char) 无符号字符 (Char Byte，没有多少人用) 颜色参考值 (ColorRef) 坐标差（长度 ShortInt） Double Word 函数 Handle（句柄） 整型 长整型 (Long Int) Long Pointer 类的成员 短整型 (Short Int) Near Pointer Pointer 字符串型 以 null 做结尾的字符串型 (String with Zero End) Word
* 2.驼峰式命名：
* 又叫小驼峰式命名法。   
   第一个单词首字母小写，后面其他单词首字母大写。
* Ex：
* Int age = 10;
* String myName = “LiLy”;
* 3.帕斯卡([Pascal](https://baike.baidu.com/item/Pascal%E8%AF%AD%E8%A8%80))命名
* 又叫大驼峰命名法。
* 每个单词的首字母都大写。
* Ex：
* Int Age = 10;
* String MyName = “Lily”;
* 4 . 还有些许其他的命名规范，如：下划线命名法。

## ****分布式****

* 负载均衡
  + 什么时候才会负载均衡（什么是负载均衡）
  + 负载均衡的几种实现
    - OSI层面，在网络7层协议基础上实现的负载均衡
    - 软件层面
  + 负载均衡算法
    - 权重轮循均衡
    - 权重随机均衡
    - 响应速度均衡
    - 最少连接数均衡
    - 处理能力均衡
    - DNS响应均衡
  + 集群
    - 负载均衡服务器
    - 应用服务服务器
    - 文件系统服务器
    - 数据库服务器
    - 缓存服务器
    - ...
  + 分离模式
    - 前后端分离
    - 动静分离
    - 数据库读写分离
    - 服务分离
    - 业务分离
    - ...
* 故障可弹性
* 服务一致性
* 中心化配置

## ****WCF****

* 服务是端点的集合。每个端点都有自己的地址，绑定，契约。契约揭示了端点的功能，地址就是这些功能在网络上所在的位置，而绑定则指明了如何访问这些功能。
* 端点和契约之间是多对一的关系，如果使用了相同的绑定，则可以位于同一个地址。
* 服务中可以通过多个端点发布同一个契约，则可以通过多个绑定对其进行访问。
* 例如：我们可以发布一个basicHttpBinding以获得广泛的互通；也可以通过使用netTcpBinding发布这个契约，已获得更高的性能。
* 上述名字在配置文件中对应的节点名词：
* 服务 => Service
* 端点 => EndPoint
* 地址 => Address
* 绑定 => Binding 管道
* 契约 => Contract 接口
* **契约**
* 1.服务契约
* 服务契约定义了服务的操作，即服务类所发布的接口方法。服务契约为服务提供正式的 WSDL接口。通过[ServiceContract]与[OperationContract]属性，就可以定义服务契约。默认的操作名称是从类和方法名得来的，不过也可以在定义这些属性的时候指定。服务契约支持3 种消息交换模式：请求一响应模式、单向模式和双工模式。

请求一响应模式的操作契约会令客户端的调用阻塞，也就是客户端等待服务操作结束，

* **请求一响应模式**的操作契约会令客户端的调用阻塞，也就是客户端等待服务操作结束，然后才重新得到执行控制权。因此，这种模式的契约只应该用于服务调用相对快速且用户愿意等待相应结果的情况下。使用请求一响应消息交换模式时，如果服务请求的执行时间很长，客户端可以使用.NET异步模式来避免阻塞等待。
* **单向操作契约**名符其实，是单向的，这种服务不会向客户端返回结果。服务操作的返回值类型必须为void，而且要在[OperationContract]属性上做[IsOneWay=true]的标记。单向契约可以在任何传输方式上实现，包括MSMQ
* **双工操作契约**，对客户端与服务之间的请求与相应的执行进行了解耦，从而具有最大的自由度与最高的性能。客户端与服务之间的双工信道建立完毕后，无论是客户端还是服务都可以发起消息传递\*这种模式非常适于用来对客户端作通知功能。
* 2.数据契约
* 数据契约定义了服务的应用层数据接口。凡是标有[DataComract]和[DataMember]属性的类，均会被包含在代表服务契约的WSDL中的XML模式定义（XSD)部分。其他数据类型也可以通过WSDL对外发布，如基本类型及标有[Serializable]的类型，由于[DataContract]对于序列化的规则是针对互通性设计的，WCF将其作为首选的序列化机制
* 具有[DataMember]的类成员，会被包含在一个数据契约中。这是严格按照自愿原则进行的，因此只有标有此属性的成员才会被包含在契约中。这也令[DataContract]较[Serializable] 更适宜作为序列化机制，毕竟后者会造成服务的内部结构被披露给外部。
* 契约也支持类的层次结构，其命名空间会伴随整个层次机构。为了支持多态及包含其他对象的容器，WCF允许服务公布一系列己知类型,数据契约的设计也考虑了版本控制问题。 只要遵循一定的规则，即使向契约中添加了新成员，现有的客户端仍能继续工作.
* 3.消息契约
* 消息契约是定义于操作之上的契约，用于访问SOAP消息头及消息体。消息的数据类型, 可以通过[DataContract]或[Serializable]定义（有类型消息)，也可以简单地定为Message类型 (无类型消息)。有类型消息之上，可以定义[MessageHeader]与[MessageBody]属性。消息的数 据结构在设计阶段可以是未知的，以取得最大的灵活度，无类型消息也能够访问SOAP消息 头和消息体。

## ****插件开发****

* Mvc Area插件开发
* OSGi.NET
* ...

## ****消息队列****

* 消息队列 - Message Queue(MQ)，是一种应用程序之间的通信机制，将部分无需立即回调获取结果，并且耗时的操作，使用异步处理的方式提高服务器的吞吐量及性能.如秒杀活动，上传任务，日志记录等
* .
* MSMQ
  + 微软官方推出的消息队列框架
* RibbitMq
  + 基于 AMQP 系统协议，由 Erlang 语言开发，开源的消息队列,支持AMQP，XMPP, SMTP, STOMP等协议，使用简单
* ZeroMQ
  + 一个轻量级的消息通信组件,侧重于点对点的消息传输上
* Redis
  + 虽是一个Key-Value的NoSQL数据库,但本身也有Queue队列功能
* ...

在分布式的系统中使用消息队列后，我们的生产者只管向消息队列中甩完数据后立即返回，而不管是哪个消费者来消费，可以看到

其实消息队列有如下三个优点。

1.  加快网站的响应速度

　   这个刚才也说了，应用层直接把消息给消息队列然后直接返回调用端，这样就避免了处理复杂的业务逻辑然后同步的插入到数据库后再返回造成的响应延迟，在很多网站上用户提交订单就是这么处理的，应用层生成一个订单号之后，将订单丢给消息队列，然后直接到订单成功页面，此时后端消费者对订单还没有处理完毕，因为后面会有比较多的数据操作，比如减库存，数据库同步等等，而用户如果想要看到订单详情，需要点击“订单号”才能进入到订单详情页，这种处理也是因为消息队列的非及时性，所以需要得到网站设计方改进和支持。

2. 提供系统的可用性

    既然是异步操作，就造成了生产者不知道消费者的存在，而反过来消费者不知道生产者的存在，如果消费者挂了就不会影响到生产者，生产者还会照常无误的向消息队列甩消息，当消费者恢复正常后就会继续消费消息队列，系统的表现可能就是email或者短信延迟收到，不会对系统造成太大的影响。

3. 并发削峰

   既然是大型网站就免不了高并发的读写操作，很典型的一个例子就是电商中的秒杀，这种高并发的写操作，如果一下子都涌入到数据库里面去了，会导致数据库的压力非常大，从而导致客户端的访问延迟，就是不挂也容易造成数据库的死锁从而造成很多灵异事件，遇到这种一拥而入的情况，我们就必须进行线性化操作，在代码层面上我们可以用lock机制来串行化，在分布式中我们用“消息队列”来[串行化](#串行化)，而且还可以通过逻辑操作来对消息队列进行动态的防洪，控洪。

 在消息队列的选择上，微软有自己的MSMQ，但是在大型网站中，我们的消息队列同样需要集群，并且希望能跑在内存中，并且支持序列

化硬盘，同时在“伸缩性”和“可靠性”上要有好的作为，所以推荐大家用用开源的RabbitMQ，网址：<http://www.rabbitmq.com/>  不过很

多公司都有自己开发的消息队列，比如携程的CMessage，淘宝的MetaQ。

## ****依赖倒置框架****

* 基于面向对象6大原则之 - 依赖倒置原则DIP - 工厂模式实现
* IOC - inversion of control 控制反转，调用者不再创建被调用者的实例，由框架(容器)创建
  + DL - Dependency Lookup 依赖查找，容器提供回调接口和上下文环境给组件
  + DI - Dependency Injection 依赖注入，容器创建好实例后再注入调用者
* .
* Autofac
  + 最为流行的IOC框架之一，轻量且高性能
* Unity
  + 最为流行的IOC框架之一，微软出品
* Spring.NET
  + 以Java版的Spring框架为基础，提供了多种功能
* Ninject
  + 一个轻量级的注入框架
* ...

## ****任务调度****

* Quartz
  + 开源的作业调度框架，是Quartz的C#移植版本
* ...

## ****日志记录****

* Log4net
  + 从java平台下移植过来的非常优秀的日志记录框架
* Nlog
  + 相对于Log4net，配置更为简单
* Microsoft.Framework.Logging
  + ASP.NET5中的日志框架集
* ...

## ****搜索引擎****

* Lucene.net
  + 是Lucene的.net移植版本，它不是一个完整的全文检索引擎，而是一个全文检索引擎的架构
* ElasticSearch
  + 是基于Lucene架构面向企业搜索应用的扩展，提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web接口
* 盘古分词
  + 中文分词组件，作者肖波
* HubbleDotNet
  + 开源数据库全文搜索引擎，作者肖波
* ...

## ****工作流引擎(Workflow - WF)****

* Business Process Management - BPM 业务流程管理
* .
* Windows Workflow Foundation - WWF
  + 微软的工作流产品，提供一套工作流引擎和VS解决方案自带的流程设计器
* NetBPM
  + 从JBpm1移植到.NET平台下的开源工作流项目
* CCFlow
  + 国产开源工作流

## ****.Net常用工具及插件****

* Colorful-IDE：可以更换VS的壁纸
* [ReAttach：可以快速附加到进程（开源）](https://github.com/erlandranvinge/ReAttach)
* C# Outline：VS代码块折叠插件
* Reshaper：即时分析代码质量，消除错误，安全地更改代码库等
* OzCode：VS调试时候的辅助插件
* LINQ Pad：用来测试Linq查询的插件
* [Rextester](http://rextester.com/)：在线测试C#代码
* SmartAssembly：.Net加密混淆工具
* ConfuserEx：.Net加密混淆工具
* ILSpy：.Net反编译工具
* Reflector ：.Net反编译工具

总结归纳不易，如果有疑问或异解请留言给我。

后面我会写一些文章来让大家系统的学习web开发，而这篇技术栈我也会不断更新。

我喜欢和我一样的人交朋友，不被环境影响，自己是自己的老师，欢迎加群 .Net web交流群， QQ群：166843154 欲望与挣扎

## ****Nginx踩过的坑****

**Windows环境下的Nginx环境搭建**

1. 到Nginx官网上下载（http://nginx.org/en/download.html），这里我选择的是1.12.2(不知怎么的我1.14.0怎么也启动不了）
2. 将文件解压缩（路径全英文），启动cmd命令（注意：这里必须用管理员权限才能启动它）
3. 用CD命令进入根目录输入nginx.exe -t即可开启
4. 重启命令是nginx.exe -s reload 关闭命令是nginx.exe -s stop
5. 另外还要设置下打印日志的配置，可以参考https://blog.csdn.net/qq\_35624642/article/details/71056890写的很详细。
6. 好了，访问<http://localhost看到欢迎页面就行>。

串行化-----串行是指多个任务执行时，各个任务按照顺序执行，完成一个才能执行下一个。

并行化-----多个任务可以同时执行。异步是并行化的前提条件。

目录分级-----按照模块将不同的文件区分开，就能提高读取的效率

[回顶部](#顶部)