学习新技术的方法一：

1. 从理论上下手，了解一下是什么东西。（知道）---概念
2. 从使用上下手，了解一下他的语法。（记住）--使用
3. 找出和他相似的技术点，从性能上 比较两者的实现的原理（理解）--性能、原理
4. 从应用场景上推演，加深对技术的理解（练习--举一反三）--应用场景
5. 生命周期 理解什么时候从哪儿开始，什么时候从哪结束

适应新框架的开始：

数据库连接字符串 增删改查 多表联查 文件上传 条件查询 导入导出 系统日 志记录 用户操作日志记录 当前用户信息 角色、权限、模块配置

列表 添加 修改 搜索框 刷新列表 返填 选项卡 配置样式（文本框等长宽高）

调用控制器的方法js 增删改查 添加按钮 非空验证

1.

问题：

VS2015提示：未安装Style的Visual Studio语言支持，代码编辑Intellisense将不可用。服务器控件的标记Intellisense可能不起作用

解决：

（工具-拓展更新）Microsoft ASP.NET and Web Tools

2.

问题：

Datatables（页码）不刷新

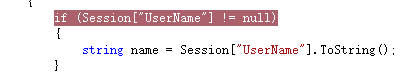
解决：

加上状态: "stateSave":false

1. mvc中的过滤器
2. 授权过滤器：限制进入控制器或控制器的某个行为方法

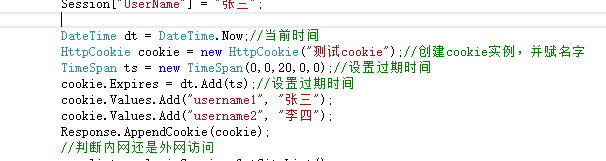
所有过滤器中最先执行IAuthorizationFilter

1. 动作过滤器：动作过滤器IActionFilter
2. 结果过滤器IResultFilter
3. 异常过滤器IExceptionFilter
4. 设置session 在服务端，session[“1423”]=”张三”





1. 设置cookie，创建httpcookie对象并赋名字，创建当前时间，创建timespan实例，赋值过期时间，然后将两者结合的过期时间设置在cookie的expires中，response appendcookie（cookie）添加在浏览器中



# 6.**HashTable, Dictionary, ArrayList、数组**的区别：

1.HashTable和Dictionary 前者在添加数据时，不需要声明类型。而后者需要声明其类型。并且，前者存入的数据都是objct的类型，取数据的时候，需要进行类型转换，对于值类型的数据，需要进行装箱和取消装箱。而后者，因为在声明时候，指定了类型，所以不需要进行类型转换。相比之下，dictionary性能高一些。

前者：数据类型不唯一 后者：体现顺序的情景中（remove之后，顺序被打乱）

1. ArrayList和数组 前者在添加数据的时候，不需要声明其类型。而后者需要声明其类型。并且，前者不需要手动指定数组的长度，会根据数据的变化自动改变数组的长度。后者需要指定数组的长度。

7.viewbag和viewdata的区别：前者是动态类型，viewbag.名称=值 后者是字典类型viewdata【“1320”】=值 的方式，两者只在当前Action中有效，等同于view，两者的值可以互相访问，因为Viewbag的实现中包含了viewdata

8.控制器中actionresult的返回值常用的有 view、content、json 还有file jsvascript css等

9.将实体数据转换成json的几种方式：

1.通过字符串拼接的方式，返回content

2.序列化为json字符串

10.

虚方法可以声明方法体 抽象类和接口不可以声明方法体

/// 抽象类和虚方法 是需要重写的 接口是需要又实现的

/// 抽象类和接口必须重写实现所有的成员 但是虚方法可以先不重写 先不实现

/// 重载是同一个类中 几个方法名相同 参数或参数类型不同的方法

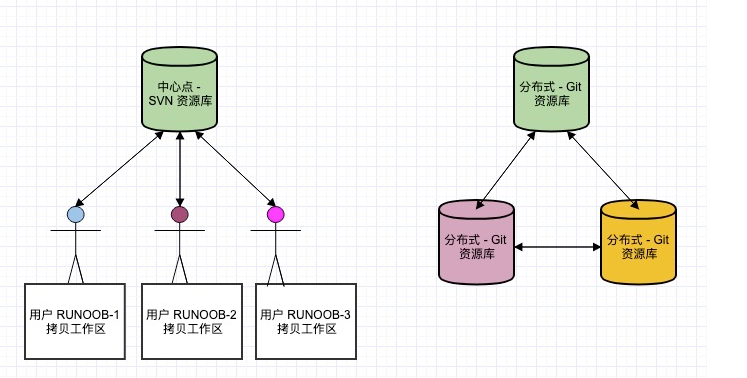
/// 重写是在继承关系中，子类重新定义父类的方法 方法名和参数都相同

1. 静态方法和非静态方法的区别
   1. 使用上：静态方法直接点方法或者属性就好，不需要实例化。而且非静态方法需要实例化之后，才可以调用。
   2. 声明周期不同：
      1. 静态方法又叫类方法，所以静态方法会随着类的定义加载到内存中，一直到线程结束，静态方法才会被销毁。
      2. 非静态方法，在实例化对象之后，才会被创建，直到这个实例化的对象被销毁时，才被销毁。
2. File类和FileInfo类的区别
   1. 两者都可以对文件进行增删改查的操作，他们都是基于Syatem.IO命名空间下的操作类
      1. File 静态类，如果对文件进行单一的操作或者对很多文件进行操作，推荐使用file类，因为它不要实例化对象，可以直接使用。
      2. Fileinfo类，如果对同一文件进行多次操作，推荐使用fileinfo类
3. File类的每个方法都需要对路径进行检查，即使多次对同一个文件进行操作也需要进 行检查。Fileinfo不需要进行多次检查，只需要在创建的时候进行一次检查。所以，在对 同一个文件执行多次操作的时候，用fileinfo。 如果执行一些单一的方法，就用file。

13.

1. 总结思考为荣，浑水摸鱼为耻
2. 定义常量为荣，魔法数字为耻
3. 代码优化为荣，复制粘贴为耻
4. 设计模式为荣，代码重复为耻
5. 参数校验为荣，运行异常为耻
6. 代码自测为荣，过度自信为耻
7. 规范日志为荣，乱打日志为耻
8. 接口兼容为荣，接口裸奔为耻

14.Func委托和Action委托的区别

1. Func委托有返回值 Func<参数类型（传入的参数），返回类型>
2. Action委托没有返回值
3. Out和Ref的区别
   1. 前者只出不进，后者有进有出
   2. 前者在方法内部完成初始化，后者在传入方法参数之前--对其完成初始化
   3. 两者都是按照地址传递的，从而改变参数的值
4. 里氏替换原则
   1. 所有使用基类的地方能够使用子类进行替换，而程序的行为不会发生任何变化（替换为子类之后不会产生错误或异常）
   2. 并且可以安全的将子类的引用赋值给父类（父类=子类）
5. 隐式类型转换和显式类型转换
   1. 前者是 从小的类型转换成大的类型，从派生类转换到基类
   2. 后者是 强制类型转换，需要强制转换运算符
   3. 前者不会造成数据丢失，后者会造成数据丢失
6. git和svn的区别
   1. 前者是分布式的，后者不是
   2. 前者把内容按元数据方式存储，后者是按文件：所有的资源控制系统都是把文件的元信息隐藏在一个类似.svn的文件夹里
   3. Git没有一个全局的版本号，svn有
   4. Git的内容完整性要优于svn
   5. 
7. js闭包
   * 1. Js嵌套函数中 返回子函数，子函数中要有对父函数中私有变量的操作 并把这个父函数赋值给一个变量 在调用这个变量的时候 父函数中的私有变量只声明命一次。
     2. 闭包就是一种保护私有变量的机制，在函数执行时形成私有变量的作用域，保护里面的私有变量不受外界的干扰。
     3. 就是实现了局部变量也能像全局变量一样使用（例子：累加）。
8. 前台线程和后台线程的区别
   1. 应用程序必须在运行完所有的前台线程后退出
   2. 应用程序可以在没有运行完后台线程的情况下直接退出，所有的后台线程在应用程序退出时，自动结束。
   3. 用Thread创建的线程都是前台线程，主线程也是前台线程，线程池的线程默认为后台线程（task创建的默认是线程池中的线程）
   4. IsBackground属性用来设置线程是前台还是后台，需要在线程启动之前运行
   5. 多线程的优点：
      1. 可以让计算机同时坐很多事情，节约时间。
      2. 后台运行程序，提高效率，不会使主界面出现假死或者无响应
9. c#中的? ?? ?. ?””:””
   1. ?可为空（一般在字段中写）
   2. ??如果左边的为空 则 返回右边的
   3. ?.如果左边的不为空 则 执行右边的
   4. ?””:”” 三目运算符
10. 扩展
    1. 给$添加方法:$.extend({ min:function(a,b){return a<b?a:b}, })
       1. 使用$.min(3,5) 返回3
    2. 给jQuery的方法:$.fn.extend({ checkAll:function(){ this.prop(‘checked’,true) }, })
       1. 使用$item.checkAll()
11. 协变Out与逆变in
    1. 和里氏替换原则相似，都体现在隐式的类型转换
    2. 牵扯到里氏替换原则，协变和逆变就是将里氏替换原则的变化体现在泛型的参数上
    3. 协变，父类=子类
    4. 逆变，子类=父类
    5. 值类型不支持协变和逆变
12. 泛型的优点
    1. 性能高：定义数据类型，不需要类型转换，避免拆装箱带来的性能损失；
    2. 比如在ArrayList引用类型中添加数字，就会发生装箱。在List<Int>就不会发生装箱。
    3. 类型安全：定义允许使用的数据类型，在编译时检查类型错误，及早发现错误。
    4. 重用代码
13. dictionary和hashtable的区别：
    1. Hashtable表示键值对的集合
    2. Dictionary表示键和值的集合
    3. hashtable不支持泛型，而dictionary支持泛型
    4. Hashtable元素属于object类型，所以在存储和检索值类型的时候会装箱和拆箱，需要进行类型转换，耗时间。
    5. 单线程推荐使用dictionary，支持泛型，声明已经定义好了类型，不需要进行装箱和拆箱，速度较快。
14. 事务TransactionScope scope = new TransactionScope（）
    1. 提交的时候 支持循环多次操作 然后统一提交（也就是说，循环执行多次操作的时候，只需要在最后提交一次即可）
15. 添加、修改、删除需要的验证
    1. 添加（验证唯一性，可不可以重复添加，提示已经添加）
    2. 修改（验证有没有删除，两天电脑的情况下，提示数据不存在）
    3. 真删除（验证有没有删除，提示已经删除）
16. 预处理器：预处理器指令指导编译器在实际编译开始之前对信息进行预处理。
    1. 所有的预处理器指令都是以#开始
    2. 且在一行上，只有空白字符可以出现在预处理器指令之前。
    3. 预处理器指令不是语句，所以他们不以分号结束
17. varchar和nvarchar的区别
    1. 两者的最大长度设置不一样，前者是1-8000。后者是1-4000
    2. 前者能存储的字节数就是他的长度 后者能存储的字节数是他的两倍。
    3. 存储方式上看，前者是按照字节存储的。后者是按照字符存储的
    4. 存储量来看，前者比较节省空间，因为存储的大小就是字节的实际长度，而后者是双字节存储
    5. 使用上，如果存储内容是英文字符，建议使用前者。如果是带有汉字的，建议使用后者，因为后者使用的Unicode编码，统一的编码，会减少乱码出现的几率。

简单的来说，一般有汉字的使用后者。没有汉字的使用前者。

1. String和stringBuilder的区别
   1. 前者是不可变类，后者是可变类
   2. 前者在拼接的时候，每次拼接都会创建一个新的string字符串，然后gc回收之前的字符串
   3. 后者的对象是变量，对变量进行操作就是直接对该对象进行更改，而不进行额外的创建和回收操作。
   4. 所以在频繁拼接字符串的时候，最好使用后者。
2. 内存泄漏：（内存浪费）未能释放已经不在使用的内存（因为疏忽或者错误导致应用程序产生的情况）
3. 缓冲区溢出：缓冲的数据超出了缓冲区的容量，造成多余的部分溢出。
4. 复杂度
   1. 算法效率：
      1. 时间效率
      2. 空间效率
   2. 时间复杂度：一个算法所花费的时间与其中语句的执行次数成正比例，算法中的基本操作的执行次数，为算法的时间复杂度。（执行的次数，并且根据n的 大小也记录了增长的趋势--比如O（1）表示算法消耗的时间不随规模的增长而增长）
      1. 常用的时间复杂度大小比较：O(1)<O(logn)<O(n)<O(n的2次方)<O(2的n次方)<O（n的阶乘）
      2. 计算时间复杂度的方式，就是算出程序运行的基本操作的执行次数的最坏情况。
   3. 空间复杂度：表示算法需要消耗的最大数据空间。
      1. 是时间复杂度一样 也是用o（n）这样的式子来表示，大小比较也如上所示。
   4. 编码复杂度:是一个定性的概念，并没有什么量化的标准。对于一个问题来说，如果用了冗长的算法思想，那么代码量就会非常巨大，其编码复杂度就会非常大。
5. Unix时间戳：是从1970年1月1日 开始所经过的秒数，不考虑闰秒。
6. 乐观锁和悲观锁
   1. 乐观锁：乐观的认为别人不会同时操作数据，乐观锁默认是不会上锁的。
   2. 悲观锁：悲观的认为别人一定会同时修改数据，任何情况都会上锁。
   3. 乐观锁适合在读取频繁的时候，悲观锁适合在写入频繁的时候。
7. 互斥锁：对共享数据进行锁定，保证同一时刻只能有一个线程去操作。
8. 死锁：指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，他们都将无法进行下去。
9. 主从复制：例：一台redis服务器的数据，复制到其他的redis服务器。前者称为主节点，后者称为从节点，数据的复制是单向的，只能由主节点到从节点。主节点以写为主，从节点以读为主。
   1. 读写分离：就比如主从复制的模式，主节点负责写，从节点负责读。
   2. 数据冗余：主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式
   3. 故障恢复：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复，实际上是一种服务的冗余
   4. 负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写redis数据时，应用连接主节点，读redis数据时，应用连接从节点），分担服务器负载，尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高redis服务的并发量。
   5. 高可用（集群）基石：除了上述作用外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是redis高可用的基础。
10. 在缓存穿透中设计到的布隆过滤器：
11. 哈希函数：hash（英文的意思是混杂、拼凑、重新表述的意思）hash是通过计算哈希值，打破元素之间的关系，使集合中的元素按照散列函数的分类进行排序。

(1)为什么要有hash？

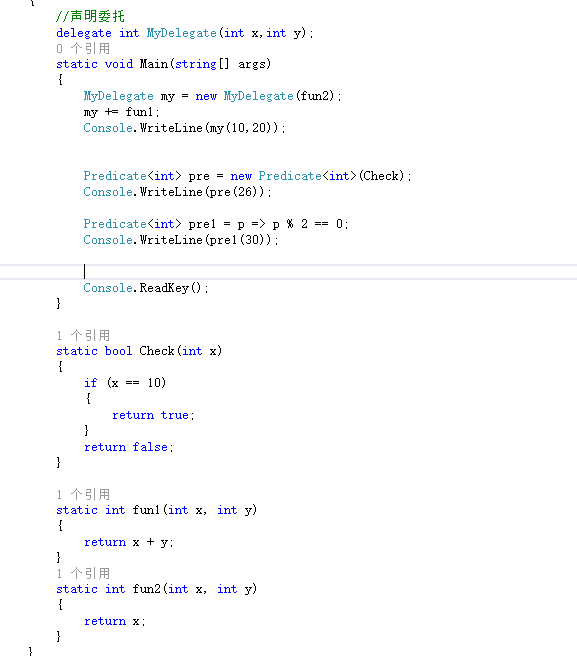
* + 1. Hash函数是一种映射关系，根据数据的关键词key，通过一定的函数关系，计算出该元素存储位置的函数。表示为address=H[key]
    2. 相比较普通的数据或者链表来存储元素，哈希函数查找元素的时候，效率更高。

1. Lua脚本的特性（在redis中）
   1. 减少网络开销，在lua脚本中可以将多个命令放在同一个脚本中执行
   2. 原子操作，redis会将整个脚本作为一个整体执行，中间不会被其他命令插入（redis执行命令是单线程）--因为原子性，所以延伸出的缺点就是，不适合在lua脚本中写入比较复杂的操作，否则会使客户端一直处于等待的状态，影响客户端的其他操作。
   3. 复用性，客户端发送的脚本会永远存储在redis中，其他客户端也可以复用这一脚本来完成同样的逻辑。
2. 接口的添加操作要有幂等性，主要是在添加操作中，不管什么情况，只添加一次。（即成功的条件添加，失败的条件不管）
   1. Cas概念：是一种无锁原子算法，主要是为了保证原子性。
   2. Cas如何保证原子性：总线锁定，缓存锁定
      1. 总线锁定：
         1. 是指cpu使用了总线锁，所谓总线锁就是cpu提供的lock信号，其他cpu的总线请求将被阻塞。（总线：是计算机组件间传输数据的方式，也就是说通过总线，cpu与其他组件连接传输数据，就是靠总线完成的，比如cpu对内存的读写。）
         2. 虽然保证了数据的一致性，但是在锁定期间，总线锁定阻止了被阻塞处理器和所有内存之间的通信，因此总线锁定开销较大。
         3. 所以现代CPU为了提升性能，通过锁定范围缩小的思想设计出缓存行锁定（缓存行是CPU高速缓存存储的最小单位）
      2. 缓存锁定：
         1. 对cpu缓存行进行锁定，是基于缓存一致性实现的。
         2. 某个cpu对缓存数据进行更改时，会通知缓存了该数据的cpu抛弃缓存的数据或者从内存重新读取。
      3. 缓存一致性：
         1. 如果一个变量在多个cpu中都存在缓存，那么就可能存在缓存不一致的问题。因为在缓存中变量没有共享，各自在各自的缓存中操作，那么就存在缓存不一致的问题。
         2. 两种解决办法：总线加锁，缓存一致性协议
         3. 总线加锁
            1. Cpu和其他部件进行通信都是通过总线来进行的，如果对总线加锁的话，那么就阻塞了其他cpu通过总线对其他部件的访问，从而使得只能有一个cpu能使用这个变量的内存。 但是在总线加锁的期间，其他cpu无法进行访问，那么就导致了效率低下。
            2. 缓存一致性协议

保证了每个缓存中使用的共享变量副本是一致的。

当cpu向内存写入数据时，如果发现操作的变量时共享变量，而且在其他cpu中也存在该变量的副本，那么就会发出信号让其他cpu把该变量设置为无效状态，因此当其他cpu读取这个变量的时候，就会发现自己内存中的变量时无效的，那么就会从内存中重新获取。

* 1. Cas原子一致性的问题：
     1. 只能保证一个共享变量的原子操作
        1. Cas只针对一个变量使用，如果是多个变量，最好使用锁，当然也可以是将多个变量整合成一个变量。
     2. 在并发量大的情况下，cas将消耗大量的cpu资源
        1. 很容易发生多个线程反复更新某一个变量，更新不成功，会消耗大量的cpu资源（原子性）
     3. ABA问题
        1. 比如两个线程操作一个数据，线程1工作时间为10秒，线程2工作时间为2秒，
        2. 那么可能在线程1的工作时间内，此共享变量已经被线程2修改了很多次，只不过是最后一次和为操作此变量之前的数据一致
        3. 那么有的业务可能不需要关心中间的过程，只要最后的结果一样就行，但是有的业务就不允许变量有所变化。
        4. 相当于是只关心共享变量的起始值和结束值，不关心过程中共享变量是否被其他线程动过。
        5. 解决：加上版本号，根据版本号来判断该共享变量是否被修改过。

1. Cas原子性和锁的区别
   1. 两者都解决了并发中同步的问题，前者适合一个共享变量的原子操作，后者适合多个共享变量的原子操作
2. 委托：
   1. 概念：委托就是一个类型，里面装的是方法，调用委托的时候，里面的方法将被执行
   2. 定义：public delegate void MyDelete（int number）
      1. 访问修饰符 关键字 返回类型 委托名称 （参数类型 参数名）
   3. 实例化：
      1. 使用new关键字：MyDelegate name = new MyDelegate（方法名）
      2. 使用匿名方法：MyDelegate name = delegate(参数){函数体}；
      3. 使用lambda表达式：MyDelegate name = （参数）=>{函数体}；
   4. 泛型委托：
      1. 例如func委托《参数类型，返回类型》
   5. c#内置泛型委托：
      1. Action<参数类型> 无返回值的方法
      2. Func<参数类型，返回类型> 有返回值的方法
      3. Predicate<参数类型> 返回值是一个bool类型
         1. 
   6. 多播委托：
      1. 就是可以在一个委托类型中，加入多个方法，调用的时候按照加入的顺序依次执行方法。 如果有返回值，那么就返回最后一个方法的返回值。
      2. 先声明委托 然后通过+=的方法加入进去 也可以通过-=的方法移除
      3. 
3. Redis操作中的问题
   1. 缓存雪崩
      1. 大量的缓存请求无法在redis中处理，应用将大量的请求发送到数据库层，导致数据库的压力激增。
      2. 两个原因导致：
         1. 缓存中有大量请求过期，导致大量请求无法得到处理。
            1. 给数据的过期时间设置一个随机数，避免了大量数据在同一时间过期
            2. 服务降级，把数据分为核心数据和非核心数据（商品库存/商品属性），核心数据直接查redis，没有就查数据库，非核心数据直接暂停访问redis和数据库，返回预定义信息、控制或者报错
         2. Redis缓存实例发生故障，宕机了，redis无法处理请求，请求直接累计到数据

库中。

服务熔断和请求限流限制

服务熔断，不调用redis接口 直接返回 redis实例恢复了 再调用

请求限流限制，在业务系统的请求入口限制每秒进入系统的请求数

提前预防

构建redis缓存集群，如果主节点宕机了，那么就让从节点作为主节点继续提供数据服务

* 1. 缓存击穿
     1. 某个访问非常频繁的热点数据过期时，大量的请求直接访问数据库
     2. 对某个热点数据不设置过期时间
  2. 缓存穿透
     1. 访问缓存和数据库中都没有的数据，可能是误删然后访问，也可能是恶意访问
     2. 一旦发生穿透，那么就在redis中设置空值或者默认值，后续的请求再访问的时候，直接返回这个空值或默认的值
     3. 使用布隆过滤器快速查询数据是否存在，避免去数据库中查询数据是否存在
     4. 在请求入口检测，把不合法的、不合理的、请求字段不存在的直接过滤，不让这种请求访问redis和数据库
* 针对缓存雪崩，合理地设置数据过期时间，以及搭建高可靠缓存集群；
* 针对缓存击穿，在缓存访问非常频繁的热点数据时，不要设置过期时间；
* 针对缓存穿透，提前在入口前端实现恶意请求检测，或者规范数据库的数据删除操作，避免误删除。
  1. 

1. 接口和抽象类的区别
   1. 相同点
      1. 都可以被继承
      2. 都不能被实例化
      3. 成员都需要被实现
      4. 都可以包含方法声明
   2. 不同点
      1. 接口可以多继承，抽象类不可以
      2. 接口中的方法没有访问修饰符，抽象类中可以有
      3. 接口是一组规范（规范返回类型/参数），抽象类是一个类。
      4. 抽象类只是把类抽象了，适用于关系密切的类，而接口适合为不相关的类提供通用的功能。
      5. 接口只能提供规则，抽象类既能提供规则也可以有实现的方法。
      6. 抽象类规定了这个类是什么，接口规定了这个类能干什么
         1. 三个类：男，女，人 那么这种情况就可以用 人抽象类
         2. 两个类：动物，人 那么这种情况就使用 吃饭接口
         3. 阿帕奇和鸽子都会飞，他们继承了一个飞的接口。但阿帕奇属于飞机的抽象类，鸽子属于鸟的抽象类
      7. 抽象类一般用在功能相近的类，接口一般用在功能不想近的类
         1. 对第一点的总结，抽象类（男，女 两个类都是人）
         2. 动物，人 动物不能是人，人不能是动物
      8. 抽象类是为了代码的复用 接口是为了实现的多态
         1. 人能做的事情，男女都能做，所以只需要在男女中写入只能是各自可以做的方法 公共的方法写在人类中就可以
            1. 虚方法和抽象方法经过override（重写）再继承关系中，都可以实现多态
            2. 继承自一个接口，同一个接口可以有多种形态。

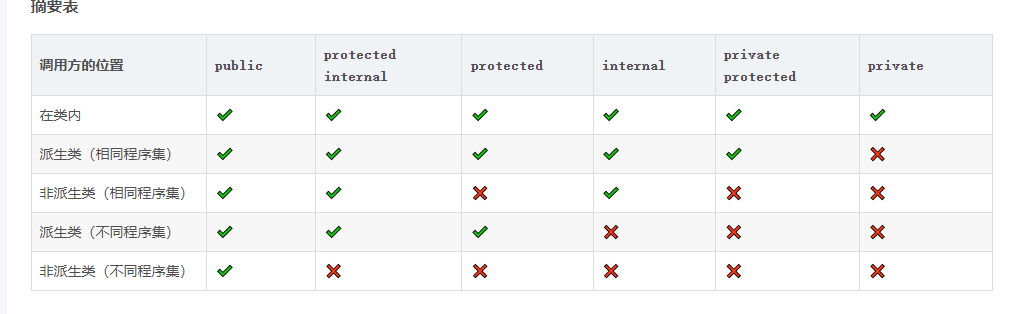
例：男人和女人类 继承跑步的接口

男人可以返回10圈的值

女人则可以返回5圈的值

* 1. 多态
     1. 重载
        1. 方法名相同，参数不同或者返回类型不同
     2. 重写
        1. virtual 继承关系中 通过 override 重写
           1. 应用场景：重写
        2. Abstract 继承关系中 通过override 重写
           1. 应用场景：重写
        3. 接口 不需要继承 调用的时候 可以实现一个接口的多种形态（例：返回值）
           1. 是一个规范，规范方法的参数列表和返回类型（根据这一点来说，就是面向接口编程，只需要知道这是干嘛的，不需要知道其中的实现原理。）以接口为主，干什么都调用接口，所以以使用的角度来说，上层就以来接口，底层也以来接口，接口就是抽象，那么就是上层代码和底层代码都以来抽象。
        4. 隐藏方法 继承关系中 通过再派生类中的方法前面加上new关键字，就可隐藏父类的方法（再非重载形式且名称相同时）
           1. 应用场景：重写基类中不带有virtual的方法（可以这么说）

1. 访问修饰符：



* 1. Public 共有的
  2. Private 私有的
  3. Internal 同一程序集
  4. Protected 基类和派生类
  5. Protected internal 同一程序集或基类和派生类
  6. Private protected 相同程序集的基类和派生类

1. Aop和oop的区别
   1. 
   2. <https://blog.csdn.net/u012165769/article/details/123334989>
   3. https://www.jb51.net/article/241358.htm
   4. AOP主要是面向动词的领域（鉴权、记录日志）
      1. 就是方便在扩展或维护代码功能的时候，可以不用修改旧的业务代码，直接添加一个需要的扩展的功能类，然后直接使用就好。只修改一个类，不需要修改多处，方便扩展维护。
      2. 以功能为主，将通用的功能抽离出来，这就是AOP。
      3. 提高代码重用性、利于维护和扩展
      4. 是OOP思想的一种补充，无法单独存在。
      5. Know：说白了，就是为了方便维护和扩展，将通用的功能封装到一个类，后续需要修改的时候，只修改这一个类就行了，不需要修改其他的类。
   5. OOP主要是面向名字的领域（学生、员工）
2. IOC控制反转（是一种程序设计思想）
   1. 依赖**Dependency**：就是有联系，表示一个类依赖于另一个类（有使用到一个类的地方，就是对这个类有依赖）
   2. 依赖倒置（反转）原则DIP：上层模块和底层模块都应该依赖于抽象（一般为接口），耦合性低，系统稳定，易于扩展。
   3. 控制反转IOC：控制权将不在由上层模块决定（上层模块调用那个底层模块，就创建那个底层模块），而是（将上层和底层模块的对象关系，放在IOC容器中，控制权交给IOC容器）
   4. 依赖注入DI： 就是将抽象（接口）和具体类的关系放在IOC容器中（ServiceCollection），需要的时候直接通过IOC容器拿就行了。
   5. IOC容器： 相当于在全局层面创建实例，然后用的话，直接调用，不需要每一处都创建。一般是注入接口和具体类之间的关系。
      1. .Net的内置容器
         1. ServiceCollection 的三种生命周期
            1. Singleton:单例，全局唯一的实例，每次从容器中获取时，都是同一个实例。
            2. Scoped:作用域，在一个作用域中唯一实例，比如在.Netcore应用程序中，一次请求相当于一个scoped，同一次请求中获取的实例是一个实例，不同的请求获取的不是一个实例。
            3. Transient：瞬时，每次的从容器中获取时，都是一个新的实例。
      2. 第三方
3. EF,Dapper的区别
   1. EF重量级ORM的代表
      1. 优点：
         1. 面向对象的方式操作数据库
         2. 支持codeFirst不用关心数据库，代码先行。能够摆脱sql，不用关心sql怎么写，可移植性比较强
         3. 开发效率高，结合linq，开发速度快
         4. 可以跨数据库，只在配置文件中修改连接字符串就能轻松支持
      2. 缺点：
         1. 相对来说比较复杂，学习曲线大
         2. 不适合统计查询系统，效率不是很高
         3. 对于多表联查，一些复杂的查询实现，需要借助其他方案进行实现
         4. 再查询条件比较多的时候，或者一些嵌套查询的时候，转换sql比较耗时
   2. Dapper轻量级ORM的代表
      1. 优点：
         1. 开源，轻巧，单文件，上手容易
         2. 支持主流的数据库
         3. 执行的效率高
      2. 缺点：
         1. 半自动的ORM，需要开发人员自己写实体
         2. 不支持codefirst，不仅需要维护数据库，还需要维护代码中的实体
4. Params参数（可以让方法参数的数量可变）
   1. 在不确定的情况下使用，不确定几个参数的情况下
   2. 调用的时候，可以不传值，不实用
   3. 参数类型必须是一维数组
   4. 加入params后，后面就不能加其他参数了，必须放在所有参数的最后
   5. 一个方法中只能有一个params关键字
5. Ref和out的区别
   1. 前者，传入的参数必须先被初始化。后者传入的参数必须在方法内初始化。
   2. 前者适合在需要被调用的方法修改调用者的引用的时候，后者适合在需要返回多个返回值的时候。
6. DTO和实体的区别
   1. 两者所差无几，因为都长得一样
   2. 但是前者适合在返回前端的时候，用到的实体
   3. 后者适合在从数据库取到数据时候的映射
7. 委托（）
   1. 可以把方法比作具体任务，委托就是一个任务的标准，只要符合这个标准的任务，都能交给这个委托去处理。
   2. 帮助我们将方法当作参数去传递。
8. Async和await（异步方法的主要语法标志）
   1. 第一个被标记为async的方法，可以没有await调用，只不过会有编译警告。
   2. Async这个关键词是可以省略的，存在的意义是为了向下兼容，为await提供上下文
   3. 主要是await 关键词开启的异步
9. 开启线程的几个类
   1. Thread
   2. ThreadPool
   3. Parallel
   4. Task
   5. BackgroundWorker组件
10. 读写分离：
11. 分库分表：