一：事务

数据库事务是指作为单个逻辑工作单元执行的一系列操作。

**数据库事务**（简称：**事务**）是数据库管理系统执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。

一个数据库事务通常包含了一个序列的对数据库的读/写操作。它的存在包含有以下两个目的：

1. 为数据库操作序列提供了一个从失败中恢复到正常状态的方法，同时提供了数据库即使在异常状态下仍能保持一致性的方法。
2. 当多个应用程序在并发访问数据库时，可以在这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。

必要属性：一个逻辑工作单元要成为事务，必须满足所谓的ACID(原子性、一致性、隔离性和持久性)属性。

应用场景：

网上购物的一次交易，其付款过程至少包括以下几步数据库操作：  
　　· 更新客户所购商品的库存信息  
　　· 保存客户付款信息--可能包括与银行系统的交互  
　　· 生成订单并且保存到数据库中  
　　· 更新用户相关信息，例如购物数量等等

不允许出现在事务中的操作（主要是对数据库本身结果及操作日志的修改）



二.锁

C#锁（c#线程方面）

**1 互斥锁lock（基于Monitor实现）**

定义：

private static readonly object Lock = new object();

使用：

lock (Lock)  
{  
　　//todo  
}

作用：将会锁住代码块的内容，并阻止其他线程进入该代码块，直到该代码块运行完成，释放该锁。

注意：定义的锁对象应该是 私有的，静态的，只读的，引用类型的对象，这样可以防止外部改变锁对象

**2 互斥锁Monitor**

定义：

private static readonly object Lock = new object();

使用：

Monitor.Enter(Lock);  
//todo  
Monitor.Exit(Lock);

作用：将会锁住代码块的内容，并阻止其他线程进入该代码块，直到该代码块运行完成，释放该锁。

注意：定义的锁对象应该是 私有的，静态的，只读的，引用类型的对象，这样可以防止外部改变锁对象

Monitor有TryEnter的功能，可以防止出现死锁的问题，lock没有

**3 互斥锁Mutex**

定义：

private static readonly Mutex mutex = new Mutex();

使用：

mutex.WaitOne();  
//todo  
mutex.ReleaseMutex();

作用：将会锁住代码块的内容，并阻止其他线程进入该代码块，直到该代码块运行完成，释放该锁。

注意：定义的锁对象应该是 私有的，静态的，只读的，引用类型的对象，这样可以防止外部改变锁对象

Mutex本身是可以系统级别的，所以是可以跨越进程的

**4 读写锁ReaderWriterLockSlim**

定义：

private static readonly ReaderWriterLockSlim LockSlim = new ReaderWriterLockSlim();

使用：

写锁

try  
{  
LockSlim.EnterWriteLock();

//todo

}  
catch (Exception ex)  
{  
}  
finally  
{  
LockSlim.ExitWriteLock();  
}

读锁

try  
{  
LockSlim.EnterReadLock();

}  
catch (Exception ex)  
{  
}  
finally  
{  
LockSlim.ExitReadLock();  
}

基本规则：  读读不互斥 读写互斥 写写互斥

作用：当同一个资源被多个线程读，少个线程写的时候，使用读写锁

引用：https://blog.csdn.net/weixin\_40839342/article/details/81189596

问题： 既然读读不互斥，为何还要加读锁

答：     如果只是读，是不需要加锁的，加锁本身就有性能上的损耗

            如果读可以不是最新数据，也不需要加锁

            如果读必须是最新数据，必须加读写锁

            读写锁相较于互斥锁的优点仅仅是允许读读的并发，除此之外并无其他。

注意：不要使用ReaderWriterLock，该类有问题

mysql锁：

**锁是计算机协调多个进程或线程并发访问某一资源的机制。**在数据库中，除传统的 计算资源（如CPU、RAM、I/O等）的争用以外，数据也是一种供许多用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一 个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。从这个角度来说，锁对数据库而言显得尤其重要，也更加复杂。本章我们着重讨论MySQL锁机制 的特点，常见的锁问题，以及解决MySQL锁问题的一些方法或建议。   
Mysql用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。这些锁统称为悲观锁(Pessimistic Lock)。

Mysql锁概述

相对其他数据库而言，MySQL的锁机制比较简单，其最 显著的特点是不同的存储引擎支持不同的锁机制。比如，MyISAM和MEMORY存储引擎采用的是表级锁（table-level locking）；BDB存储引擎采用的是页面锁（page-level locking），但也支持表级锁；InnoDB存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级锁。   
**表级锁：**开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。   
**行级锁：**开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。   
**页面锁：**开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般   
从上述特点可见，很难笼统地说哪种锁更好，只能就具体应用的特点来说哪种锁更合适！仅从锁的角度 来说：表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如Web应用；而行级锁则更适合于有大量按索引条件并发更新少量不同数据，同时又有 并发查询的应用，如一些在线事务处理（OLTP）系统。

三：mvc层次关系

mvc与三层架构：

<https://blog.csdn.net/adolphkevin/article/details/72827560>

webapi三层架构：

主项目（controller）引用 BLL, Model(entity)

BLL 业务逻辑层 引用DAL Model BLL是业务逻辑层，这里面不写sql语句，可以调用DAL层传过来的值做判断分析，并返回相应的值。最后在页面的隐藏代码中调用BLL的方法

DAL 数据访问层 引用Model DAL是数据连接层，写sql语句，添删改查方法，然后有返回值

Model(entity) 实体类 无引用

四：C#中Cookie，Session，Application的用法与区别

1.Application 储存在服务端，没有时间限制，服务器关闭即销毁（前提是自己没写销毁方法）   
2.Session 储存在服务端，客户端（浏览器）关闭即销毁(若长时间不使用 且 浏览器未关闭的情况下， 默认自动销毁时间为20分钟)

3.Cookie 储存在客户端，由用户自己销毁

application：  
程序全局变量对象，对每个用户每个页面都有效  
session：  
用户全局变量，对于该用户的所有操作过程都有效  
cookie：  
客户端信息存放对象，可以把用户的信息保存在用户的本地，  
不必总是访问服务器  
Application用于保存所有用户共用的数据信息，如果被保存的数据在应用程序生存期内根本不会改变或很少改变，用它。但是在asp.net中有个web.config，可能更好点。如果要使用application，一个需要考虑的问题是任何写操作都有要在application\_onstart事件中（Global.asax）中完成。尽管使用application.lock和application.unlock方法来避免操作的同步，但是它串行化了对application的请求，当网站访问量大时会造成性能瓶颈。因此最好不要用它存取大的数据集。  
使用方法：  
//存放信息  
Application["test"] = "100";  
//读取  
String test = Application["test"].ToString();  
  
Session 用于保存每个用户的专用信息，它的生存期是用户持续请求时间再加上一段时间（可以在web.config中设置，默认是20分钟）。Session中的信息保存在服务器的内存中，当然你也可以设置它的保存方法（如存在SQL数据库中）。由于用户停止使用程序后它仍然在内存中保持一段时间，因此使用Session对象保存用户数据的方法效率很低。对于小量的数据。使用Session还是一个不错的选择。

ASP Session的功能的缺陷  
　　目前ASP的开发人员都正在使用Session这一强大的功能，但是在他们使用的过程中却发现了ASP Session有以下缺陷：

进程依赖性：ASP Session状态存于IIS的进程中，也就是inetinfo.exe这个程序。所以当inetinfo.exe进程崩溃时，这些信息也就丢失。另外，重起或者关闭IIS服务都会造成信息的丢失。

五：http请求流程

<https://www.cnblogs.com/xuzekun/p/7527736.html>

**过程概览**

　　1.对www.baidu.com这个网址进行DNS域名解析，得到对应的IP地址

　　2.根据这个IP，找到对应的服务器，发起TCP的三次握手

　　3.建立TCP连接后发起HTTP请求

　　4.服务器响应HTTP请求，浏览器得到html代码

　　5.浏览器解析html代码，并请求html代码中的资源（如js、css图片等）（先得到html代码，才能去找这些资源）

　　6.浏览器对页面进行渲染呈现给用户

注：1.DNS域名解析采用的是**递归查询**的方式，过程是，先去找DNS缓存->缓存找不到就去找根域名服务器->根域名又会去找下一级，这样递归查找之后，找到了，给我们的web浏览器

2.为什么HTTP协议要基于TCP来实现？  TCP是一个端到端的可靠的面相连接的协议，HTTP基于传输层TCP协议不用担心数据传输的各种问题（当发生错误时，会重传）

3.最后一步浏览器是如何对页面进行渲染的？  a）解析html文件构成 DOM树，b）解析CSS文件构成渲染树，  c）边解析，边渲染 ，  d）JS 单线程运行，JS有可能修改DOM结构，意味着JS执行完成前，后续所有资源的下载是没有必要的，所以JS是单线程，会阻塞后续

六：auth2.0

<https://blog.csdn.net/tclzsn7456/article/details/79550249>

七：依赖注入

把依赖的创建交给其他人，自己负责调用，其他人负责丢给你依赖的过程叫做依赖注入

容器：容器负责两件事情：

* 绑定服务与实例之间的关系
* 获取实例，并对实例进行管理（创建与销毁）

.NET Core DI 为我们提供的实例生命周其包括三种：

* 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

八: .netcore中间件

ASP.NET Core 的请求处理管道由一个Server和一组有序排列的中间件构成，前者仅仅完成基本的请求监听、接收和响应的工作，请求接收之后和响应之前的所有工作都交给注册的中间件来完成。ASP.NET Core的中间件通过一个类型Func<RequestDelegate, RequestDelegate>的委托对象来表示，而RequestDelegate也是一个委托，它代表一项请求处理任务。

九：简述 private、 protected、 public、internal 修饰符的访问权限

答

private : 私有成员, 在类的内部才可以访问(只能从其声明上下文中进行访问)。    
protected : 保护成员，该类内部和从该类派生的类中可以访问。

Friend：友元 ，声明 friend 元素的程序集中的代码可以访问该元素，而不能从程序集外部访问。(设置友元: AssemblyInfo.cs 或命名空间上的特性[Assembly:IntervalsVisibleTo(“A”)])

Protected Friend：在派生类或同一程序集内都可以访问。  
public : 公共成员，完全公开，没有访问限制。    
internal: 在同一命名空间内可以访问。 （很少用）