C#的一些内容

# 为什么程序要编译为IL再到CLR再到机器码？

答：

因为为了考虑到不同的操作系统，编译为固定的IL，IL在不同的环境(CLR),因为不同的操作系统，环境是不一样的，再到机器码。

# dll或exe由什么组成？

答：

由metadata和IL组成；metadata表示程序清单，描述了程序集中有什么东西，需要什么东西等内容。

# 程序集如何运行？

答：

CLR首先加载程序集中的metadata部分，去查看程序集中什么东西以及需要什么东西，然后加载，再检查IL的正确性。如果正确就会通过JIT编译为机器码。以前编译过的代码，JIT会做检查不会重复编译。

# 堆栈内存分配？

答：

## 什么内存

程序运行时，进程占有的内存；

## 谁分配

CLR；

## 值类型（struct）

* 分配在栈中，变量和变量对应的值都是分配在栈中;
* 一个值类型中的内容的内存分配：

值类型中的引用类型的属性：值分配在堆中，变量名分配在栈中。

结构体中的方法里面的局部变量：值类型分配在栈中，因为是通过线程来调用方法，从而声明的全新的局部变量，在栈中分配内存。引用类型分配在堆里面

## 引用类型（class）

* 分配在堆中，变量（里面是对象的引用）在栈中，值在堆中;
* 一个对象中的内容的分配：

对象中的属性或字段：分配在堆中，因为这个对象在堆中，所以对象里面的属性也是在堆中。

对象中的方法里面的临时变量：值类型分配在栈中，因为是通过线程来调用方法，从而声明的全新的局部变量，在栈中分配内存。引用类型分配在堆里面。

* 实例化一个对象：Student stu = new Student()执行过程：

1：new的时候去堆中开辟内存，分配内存地址

2：调用构造函数（在构造函数中可以使用this，所以在之前就会有内存）

3：把引用传给变量

## 栈

线程栈，先进后出的数据结构，随着线程而分配的，默认执行方法会分配1M的内存；先进：变量声明占用内存；后出：释放的时候后出；

## 堆

对象堆，进程中独立划出来的一块内存，存放一些不会释放的对象和对象的重用；

# 装箱拆箱（仅是指内存的拷贝动作）

答：

存在内存拷贝，浪费性能，通常都是与object有关；

装箱与拆箱只能发生在父子类型里面，因为只有这样才能进行转换；

# string字符串的内存分配：

答：

## String字符串的内存是不可变的

声明一个字符串变量是在堆中开辟一块内存，放值，再返回一个引用给变量。

* 为什么不可变？

不可变是因为CLR在为字符串分配内存的时候采用了享元模式，可能会有多个变量同时指向一块内存地址，如果字符串发生变化了，其余的变量会受到影响；

堆里面的内存是连续分配的，如果改变字符串的值，会导致大量的数据移动；

* 例如：

string student = “joy”;

string student2 = student;

Console.WriteLine(student); //“joy”

Console.WriteLine(student2); //“joy”

student2 = "joyname"; //虽然修改了值，但是不是修改原本指向内存的值，因为string字符串的内存是不可变的，赋值的过程其实是新开辟了一个内存空间new string(“xx”)；

Console.WriteLine(student); //“joy”

Console.WriteLine(student2); //"joyname"

## 如何实现不可变

CLR在分配字符串内存的时候，会去查找相同的值。如果存在就会重用，不会开辟新的内存空间来存放这个值（享元模式）；

* 例如：

string student = “joy”;

string student2 = "joyname";

student2="joy";

Console.WriteLine(object.ReferenceEquals(student,student2)); //两个变量是否指向同一个地址；true

# 垃圾回收

答：

值类型出现在线程栈里面，每次调用都会有线程栈，用完之后就自己结束了，变量（值类型）会被释放掉，通常值类型是不会垃圾回收的。

引用类型出现在对象堆里面，全局就只有一个对象堆并且空间有限，所以才会需要垃圾回收来释放内存。

## 如何分配内存

在操作系统里面，内存分配是链式分配的（会根据地址指向下一个变量，可能位置在任意的位置），这样可能会有碎片；

在CLR里面，堆中的内存是连续分配的（类似数组），是因为这样分配可以节约空间；

## 如何回收

主动GC：GC.Collect()；

垃圾：完全访问不到的对象；

GC发生在new的时候，new一个对象的时候会检查堆中的空间够不够，如果内存不够（内存不够是指的是当1代对象的总内存达到一个临界值时），就会去遍历堆中的所有对象，并且都标记为垃圾（访问不到），再启动一个线程来清理内存，并且将能够访问的到的对象保留并且挪动位置（像数组一样摆放），在这个时候进程的全部线程都会停止，不允许其他的动作操作内存 ；但是如果被标记为垃圾的对象有析构函数，那么这些对象会放入一个队列中单独的处理，只是不知道什么时候会被调用析构函数；

程序退出时发生GC；

## 哪些不能被回收

静态的内容不会被回收，静态内容所包含的一切也不会被回收；但是如果将静态赋值为null，可能会引起GC回收静态的内容之前所指向的内存（因为那块内存可能不再被引用）

## GC的优化策略

* 分级策略（越是最近分配的内存，越是容易被回收）：

1：首次GC前，全部对象都记为0代；

2：第一次GC后，保留的对象记为1代；

3：再次GC先清理0代的对象，如果内存依旧不够，再清理1代的对象，这之后，剩下的1代对象升级为2代，0代对象升级为1代；

4：如果清理0代，1代，2代对象还不够就会造成内存溢出（内存占用的没有释放，比如静态资源的引用）；

* 大对象策略：如果大于85K的对象，会被单独管理；那些大对象会被组成一个链表，避免频繁的移动内存，但是链表可能会出现内存碎片；

## 析构函数与dispose()：

* 析构函数：

主要用来释放非托管的资源，在GC的时候，系统会自动去执行析构函数里面的内容；

* Dispose()：

来释放非托管资源，主动释放，对象释放的时候，并不会主动去调用dispose()方法，而是需要主动去调用这个方法来释放（或者是使用using的形式快捷调用）

一般在使用dispose()的时候，会提供一个析构函数，是为了防止调用者忘记dispose()的存在而忘记资源释放，再在dispose()中通过GC.SuppressFinalize(this)来通知回收机制不再调用这个对象的析构器；

# Linq

答：

## 委托内传参数的演变

public delegate void DoNothingDelegate();

* .net 1.0,1.1版本中：

DoNothingDelegate del = new DoNothingDelegate(funcName);

* .net 2.0中：

可以用匿名方法：

DoNothingDelegate del = new DoNothingDelegate(delegate (){

Console.WriteLine("demo");

});

* .net 3.0中：

可以使用lambda，=>叫做goes to：

DoNothingDelegate del = new DoNothingDelegate(()=>{

Console.WriteLine("demo");

});

Lambda是什么？

* Lambda的作用像是一个方法，实际上是一个类中类，里面会有与写的lambda方法对应的一个internal方法，由此可以知道在我们注册事件的时候，lambda是不能够去掉的；

如：

DoNothingDelegate del = () => Console.WriteLine("this delegate do nothing");

del += () => Console.WriteLine("this delegate do nothing");

del -= () => Console.WriteLine("this delegate do nothing");

del.Invoke(); //会输出两次，因为每个lambda语句是一个单独的存在，不会因为是同样的式子就重用；

## 匿名类（3.0）

* Object:

// 3.0，编译器不允许使用属性

object mode = new

{

Id = 1,

Name = "joy"

};

Console.WriteLine(mode);

* Var:

//3.0，var 是一个语法糖，由编译器自动推算出类型，且声明的时候必须确定变量的类型（赋值）

//var在编译后是有一个真实的类，属性是只读的，不能修改

var varmode = new

{

Id = 1,

Name = "joy"

};

Console.WriteLine(varmode.Id);

* Dynamic:

//4.0

dynamic dynamic = new

{

Id = 1,

Name = "joy"

};

Console.WriteLine(dynamic.Id);

## 扩展方法3.0

要增加方法，又不能修改类，此时就可以使用扩展方法。

扩展方法：静态类里面的静态方法，第一个参数前加this，参数类型是要扩展的类型。

没有扩展属性；

用途：

1：可以不修改类，增加方法；

缺陷：

1：优先调用实例方法；

2：若扩展的是基类型，那么所有的子类都将拥有这个扩展方法

## Linq to object（enumerable）

操作的数据是放在操作的内存里面

对集合的过滤，通过委托封装来完成通用的代码实现；泛型和迭代器来实现其特性；

public static List<T> ExternWhere1<T>(this List<T> value,Func<T,bool> func)

{

var list = new List<T>();

foreach(var item in value)

{

if (func.Invoke(item))

{

list.Add(item);

}

}

return list;

}

public static IEnumerable<T> ExternWhere2<T>(this IEnumerable<T> value,Func<T,bool> func)

{

foreach(var item in value)

{

if (func.Invoke(item))

{

//yield 迭代器，使用的时候才会调用，与Ienumerable一起使用

yield return item;

}

}

}

## Linq to sql（queryable）

SQL+ADO.NET；

对数据库的过滤，封装了通用代码ADO.NET，表达式树解析SQL

## 为什么框架会提供Action和func这两种不带返回值和带返回值的委托

因为即使很多委托实现的代码，参数，返回值完全一样，但是他们依旧是不同的类型，不能通用。并且在各种框架之间定义相同意义的委托浪费了资源。所以框架为了统一就提供了标准的Action和Func。

## 表达式目录树（expression）

语法树或者是一种数据结构，可以被解析；

表达式目录树只能有一行：只能有表达式，不能包含大括号；

* Lambda声明表达式目录树：

Func<int, int, int> func = (m, n) => m \* n + 2; //这是一个委托：一个方法

//lambda表达式声明表达式目录树

Expression<Func<int, int, int>> expression = (m, n) => m \* n + 2; //这是一个表达式目录树：是一个数据结构

//Expression<Func<int, int, int>> expressionError = { (m, n) => m \* n + 2 }; //编译出错

int result1 = func.Invoke(4, 5);

//compile()：表达式编译成一个委托，只能执行表示lambad表达式的表达式目录树

int result2 = expression.Compile().Invoke(4, 5);

* 自定义表达式目录树：
* ()=>1+2,Expression<Func<int>> funcExpression = () => 1 + 2

//首先定义表达式1+2

ConstantExpression one = Expression.Constant(1); //定义数字1这个常量

ConstantExpression two = Expression.Constant(2);//定义数字2这个常量

BinaryExpression sum = Expression.Add(one, two); //1+2:两个表达式相加

Expression<Action> actExp = Expression.Lambda<Action>(sum, null); //创建（）=>1+2这个表达式目录树,不带返回值

actExp.Compile()(); //actExp.Compile()：表示调用compile这个方法；后一个括号表示执行前面的方法的返回值

Expression<Func<int>> funcExp = Expression.Lambda<Func<int>>(sum, null); //创建（）=>1+2这个表达式目录树，带返回值

int result3 = funcExp.Compile()(); //3

* Expression<Func<int, int, int>> funExpression = (m, n) => m \* n + 2

ConstantExpression two1 = Expression.Constant(2);//定义数字2这个常量表达式

ParameterExpression m1 = Expression.Parameter(typeof(int), "m");//声明一个叫m的变量表达式，第一个参数表示这个参数是什么类型，第二个参数表示这个变量在表达式中叫什么名字

ParameterExpression n1 = Expression.Parameter(typeof(int), "n");

BinaryExpression mn = Expression.Multiply(m1, n1); //二元表达式，值表示两个表达式相乘

BinaryExpression sum1 = Expression.Add(mn, two1);

//构造一个(m,n)=>m\*n+2表达式目录树

Expression<Func<int, int, int>> funcExp1 = Expression.Lambda<Func<int, int, int>>(sum, m1, n1);

int result4 = funcExp1.Compile()(1,1); //3

* 调用一些方法：Expression<Func<Student, bool>> funcExpression = (s) => s.Id.ToString().Equals("3")

//先声明变量

ParameterExpression studentParam = Expression.Parameter (typeof (Student), "s");

//再执行s.Id

var filedId = Expression.Field(studentParam, "Id"); //获取字段，

//var propId = Expression.Property(studentParam, "Id"); //获取属性

//执行.tostring()

var toStringM = typeof(Student).GetMethod("ToString"); //获取到方法

var toStringR = Expression.Call(filedId,toStringM); //执行

//执行equals()

var equalsM = typeof(Student).GetMethod("Equals");

var equalsR = Expression.Call(toStringR, equalsM, Expression.Constant ("3"));

Expression<Func<Student, bool>> funcExpression = Expression.Lambda <Func<Student, bool>>(equalsR, studentParam);

bool result5 = funcExpression.Compile().Invoke(new Student() { Id = 3 });

* 动态生成硬编码，并通过泛型缓存存储起来：

使用场景一：

将一个对象的值复制到另一个对象中，两个对象的属性和字段一致。

（可以使用：直接赋值，反射实现，jsonConvert序列化实现，表达式树动态生成实现）。

static Tout ExpressionTrans<TIn,Tout>(TIn @in)

{

//声明参数

ParameterExpression parameterExpression = Expression.Parameter (typeof(TIn), "s");

//创建数据绑定集合

List<MemberBinding> memberBindings = new List< MemberBindI ng>();

//遍历Tout的属性和字段并且与TIn的属性和字段绑定

foreach(var prop in typeof(Tout).GetProperties())

{

var propertyTIn = Expression.Property(parameterExpression, typeo f(TIn).GetProperty( prop.Name)); //获取到TIn对应的属性，如果In和out存在映射关系，可以通过他们之间的特性来找

var propBind = Expression.Bind(prop, propertyTIn);//out与in绑定

memberBindings.Add(propBind);

}

foreach (var field in typeof(Tout).GetFields())

{

var fieldTIn = Expression.Field(parameterExpression, typeof (Tin ) .GetField(field.Name)); //获取到TIn对应的字段

var fieldBind = Expression.Bind(field, fieldTIn);//out与in绑定

memberBindings.Add(fieldBind);

}

//新建一个out对象

var outObj = Expression.New(typeof(Tout));

//初始化新对象的属性和字段,通过刚才绑定的数据

var memberInit = Expression.MemberInit(outObj, memberBinding s.ToArray());

//创建最后的表达式目录树

Expression<Func<TIn, Tout>> expression = Expressio n.Lambda<Func<TIn, Tout>>(memberInit, parameterExpression);

//最后把这个表达式目录树编译成委托存起来，下一次使用

//expression.Compile();

return expression.Compile().Invoke(@in);

}

使用表达式树动态生成硬编码的好处是：通用性好，性能高。可以在某些时候代替反射。

* ExpressionVisitor

是一个用来访问和解析表达式目录树的一个抽象类。只对外提供了一个Visitor(expression)方法调用，其余均为其子类或者同一命名空间所能够调用的方法。由于表达式目录树可以看作是一个不知道深度的二叉树，所以其与方法需要递归的调用Visitor(expression)方法，直到将表达式目录树解析完成。

* 用途：

1：解析为sql语句（linq to sql原理）：

在其子类内部通过重写一些VisitorXXX()方法来实现对表达式的拆解，提供给使用者使用。

2：表达式目录树链接：

将两个或者多个表达式目录树链接在一起时，可能会因为表达式内的参数问题，而使用ExpressionVisitor来做替换。

# 异步多线程

答：

## 进程，线程：

* 进程

一个程序运行时，占用的全部计算机资源的总和；

* 线程：

程序执行的最小单位，程序的任何操作都是由线程来完成的；线程时依托于进程存在的，一个进程可以有多个线程；线程有自己的计算资源；

* 多线程：

多个线程同时执行；

为什么会出现多线程？

1：CPU执行速度快，所以出现了分时间片来实现多线程（上下文切换：加载环境——计算——保存环境）：

微观上来讲，一个CPU在某一时刻只能执行一个线程；宏观上是多线程并发执行的。

2：多CPU，可以独立工作

## 同步，异步

* 同步：
* 逐行执行；
* 同步方法如果在带界面的程序中会卡住界面，因为主（UI）线程忙于计算；
* 同步方法慢，因为只有一个线程在计算；
* 异步：
* 不会等待调用方法的完成，直接会进入下一行；
* 异步方法如果在带界面的程序中不会卡住界面，因为计算的任务会交给子线程去执行，主线程没有进行那个操作的计算；
* 异步多线程方法速度快，因为是有多个线程并发运算；但是快的速度并不是与同步方法耗时成倍数关系，因为计算机资源换时间，可能资源不够，由此可见线程并不是越多越好；
* 异步多线程方法无序：启动无序，执行时间不确定，结束无序；因为线程是属于操作系统的，程序异步多线程是去OS拿线程来执行，不能够确定谁先得到线程，所以启动无序；因为是按时间片来执行，有的线程或许会一直得到时间片执行快，有的或许需要等待时间片执行慢，所以执行时间不确定，结束无序；由此可见不要通过等几秒的方式来控制线程的启动和等待运行结束；
* 什么时候能够异步多线程执行？

任务能够并行执行，提升效率。

* 异步多线程的回调与等待：

Action action = doSomething;

//线程执行完成之后执行的回调

AsyncCallback asyncCallback = asyncResulti =>

{

//asyncResult与BeginInvoke()的返回值是同一个

Console.WriteLine(asyncResulti.AsyncState); //hello,是beginInvoke方法传递的第三个参数(状态参数)

Console.WriteLine("thread is end,now execute callback func");

};

IAsyncResult asyncResult= action.BeginInvoke(asyncCallback, "hello");

//等待异步线程执行完成，再执行后续的语句；

//内部是通过信号量来控制的

//可以传参数：>0，最多等待一个整数毫秒（限时等待）

//卡住主线程

asyncResult.AsyncWaitHandle.WaitOne();

//等待异步线程执行完成，再执行或许语句

//获取线程的返回值

//并且每一次异步执行只能使用一次endInvoke

//卡住主线程

action.EndInvoke(asyncResult);

{

Func<int, int, int> func = (a, b) => { return a + b; };

IAsyncResult asyncResult2 = func.BeginInvoke(3, 7, r=> {

func.EndInvoke(r); //也可以在回调里面获取返回值

}, null);

//获取结果

int result = func.EndInvoke(asyncResult2);

}

## Thread 1.0

* 创建

ThreadStart threadStart = () => doSomething();

Thread thread = new Thread(threadStart);

//线程启动

thread.Start();

//线程挂起，不执行，但是依旧占用资源，现在已经不推荐使用，容易导致死锁

thread.Suspend();

//线程唤醒，现在已经不推荐使用，容易导致死锁

thread.Resume();

//线程销毁，方式是抛出异常导致线程结束，但不建议使用；

//销毁时可能会存在延迟

thread.Abort();

* 线程等待

thread.Join();//当前线程等待thread执行完成，重载传参数,join(400):最多等待400ms

* 前台线程，后台线程

默认是前台线程，true:后台线程

后台线程：随着进程退出而退出

前台线程：阻止进程退出，直到线程执行完成才退出进程

thread.IsBackground = true;

* 线程优先级

Highest：同时启动的任务，cpu会优先执行Highest的线程，但不代表会优先执行完成

## ThreadPool 2.0

* 为什么会出现线程池？

1：thread提供了太多的API

2：无限制使用线程，增加限制

3：重用线程，避免重复的创建和销毁线程

* 线程创建

ThreadPool.QueueUserWorkItem(t => doSomething());

* 线程等待
* ManualResetEvent类：

包含了一个bool属性；

false:waitOne()方法就会一直等待,知道变为true；

true：waitOne()不用等待直接通过，意味着线程执行完了；

ManualResetEvent manualResetEvent = new ManualResetEvent(false);

ThreadPool.QueueUserWorkItem(t =>

{

doSomething();

//把那个bool值设置为true;

manualResetEvent.Set();

//把那个bool值设置为false;

manualResetEvent.Reset();

});

manualResetEvent.WaitOne();

Console.WriteLine("线程池里的那个线程执行结束了，现在执行后面的语句");

一般来说不要阻塞线程池里面的线程：

因为线程池里面的线程数量有限，如果要创建的线程数大于线程池里面容纳线程数量，此时线程如果阻塞了，就不能够将后面的线程继续创建完成，容易造成死锁。

## Task 3.0

Task是基于线程池的；

* 创建：

Task.Run(() => doSomething());

//4.0

TaskFactory taskFactory = Task.Factory;

taskFactory.StartNew(() => doSomething());

new Task(() => doSomething()).Start();

* 线程等待
* 阻塞主线程

Task.WaitAll(/\*task数组\*/); //等待全部的Task执行完成之后，才会继续执行后面的语句；

Task.WaitAny(/\*task数组\*/); //等待其中某一个Task执行完成之后，才会继续执行后面的语句

* 不阻塞主线程

//等待全部任务完成

Task.WhenAll(/\*task数组\*/).ContinueWith(t =>

{

Console.WriteLine("全部线程完成后，执行的后续异步方法");

});

taskFactory.ContinueWhenAll(null, t =>

{

Console.WriteLine("某一个线程完成之后，执行的后续异步方法");

});

//等某个任务完成

Task.WhenAny(/\*task数组\*/).ContinueWith(t =>

{

Console.WriteLine("某一个线程完成之后，执行的后续异步方法");

});

taskFactory.ContinueWhenAny(null, t =>

{

Console.WriteLine("某一个线程完成之后，执行的后续异步方法");

});

* 回调

Task.Run(() => doSomething()).ContinueWith(t =>

{

Console.WriteLine("task执行之后的回调函数");

});

* 控制线程数目

1：通过设置线程池允许的线程数目，但是这种方式不好，因为线程池是一个全局的，其他的内容也会占用线程池里面的线程；

2：通过一个线程容器来限制，完成一个就移除一个，在加入新的一个；

* Delay、sleep

Tsak.Delay(num)：延迟，后续的动作延迟num毫秒之后再执行；不会卡住当前线程（相当于是把sleep()放在了一个线程里面异步执行）。

Task.Delay(2000).ContinueWith(t=>

{

Console.WriteLine("delay 2s do something");

});

Thread.Sleep(num)：休眠，线程休眠；会卡住当前线程。

* Ww

## Parallel 4.0

并行编程，是在以task为基础做了封装；

并行计算的时候，主线程会参与计算，节约了一个线程，但是如果是带有界面的程序，界面此时就会被卡住（主线程繁忙）；

* 创建

Parallel.Invoke(() => doSomething(), () => doSomething(), () => doSomething());

* For循环

//for循环，每一个循环都是一个线程,但是不清楚0-5之间的数字谁先执行

Parallel.For(0, 5, i =>

{

Console.WriteLine(i);

});

* Foreach遍历

Parallel.ForEach(new int[] { 1, 2, 3, 4, 5 }, i =>

{

Console.WriteLine(i);

});

* 控制线程数目

ParallelOptions parallelOptions = new ParallelOptions();

//并行启动的最大线程数目

parallelOptions.MaxDegreeOfParallelism = 3;

Parallel.For(0, 5, parallelOptions, i =>

{

Console.WriteLine(i);

});

## 线程异常

多线程里面的异常不能在线程外抓住，因为已经离开了主线程的try{}catch{}的范围。

* 抓取线程内部异常的方式：

1：可以通过等待线程执行结束(如：Task.WaitAll())来抓取线程内所有异常（AggregateException）,但这种情况会让主线程等待

2：线程内不允许抛出异常，如果有异常线程内部自己处理异常

try

{

TaskFactory taskFactory = new TaskFactory();

List<Task> tasks = new List<Task>();

for(int i = 0; i < 10; i++)

{

string name = $"name{i}";

Action<object> act = t =>{

if (t.ToString().Equals("name2"))

{

throw new Exception("name2 is ex");

}

if (t.ToString().Equals("name4"))

{

throw new Exception("name4 is ex");

}

Console.WriteLine($"{t.ToString()} is success");

};

tasks.Add(taskFactory.StartNew(act, name));

}

Task.WaitAll(tasks.ToArray());

}catch(AggregateException ex) //为线程准备的异常

{

foreach(var innerEx in ex.InnerExceptions)

{

//InnerExceptions里面含有线程执行期间的所有异常

}

}

## 线程取消

Task是线程外部无法终止的，是因为线程是操作系统的资源，无法掌控其什么时候取消。

* 取消方式（线程自己取消自己）

1：线程内部通过访问自定义取消的标志来取消线程

2：CancellationTokenSource去标志任务是否取消；cancel取消任务；IsCancellationRequested是否已经取消；启动task时传入token，如果此时task已经取消了，这个任务就会放弃启动，抛出一个异常

这些取消线程的方式都存在一定的延迟，因为是以固定的时间频次去访问标志位。当调用取消时，可能还没有去访问标志位

## Await/Async

* 原理：Await/Async是一种语法糖，利用了一种状态机机制。编译器在编译的时候将await之前的内容划为一部分A，await里面的内容划为一部分B，await之后的内容划为一部分C。在通过状态改变来执行相应的部分内容。
* Async使用：任何方法都可以添加async关键字，await放在task前面，通常成对出现；只有async和普通方法没有区别，只是有警告。
* async Task==async void：都表示函数没有返回值，但是两者的区别在于前者所在的函数在调用的时候可以和await连用，后者不能和await连用。
* Async Task<T>：函数的返回值是T类型，如：async Task<int>

带返回值的函数一定要等子线程执行完毕才能获取真正的返回值（Task<T> t = method();T result = t.Result;）Result里面存放的就是Task返回的结果。调用此属性同样会卡住主线程执行，等待执行结果。效果与t.Wait()一样

Task<int> resultTask = AddAsync();

Console.WriteLine($"main method ");

int sum = resultTask.Result; //访问result,主线程等待task执行完成

resultTask.Wait();//与上一行等价，没有返回值的等待

* Await使用：使用await，主线程执行期间遇到await就会退出方法直接返回，执行其他的主线程任务。

Await等待task执行结束，await之后的代码就会作为task的回调函数继续执行效果如同task.ContinueWith()。

Console.WriteLine($"noreturn before await");

TaskFactory taskFactory = new TaskFactory();

Task task = taskFactory.StartNew(() =>

{

Console.WriteLine($"noreturn task do");

Thread.Sleep(100);

Console.WriteLine($"noreturn task do end");

});

await task; //使用await,主线程执行遇到await就退出方法返回，执行其他的主线程任务

/\*

\* await 等待task执行结束，await之后的代码可以看成是task的回调（执行完task之后需要执行的内容,Task.ContinueWith()）

\* 可替换为：task.ContinueWith((t)=>{

\* Console.WriteLine($"noreturn after await");

\* })

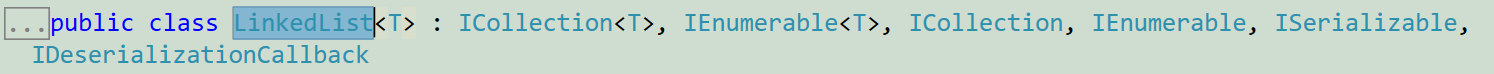
\*/

Console.WriteLine($"noreturn after await");

# 集合

为什么集合类型都实现了很多接口类型？因为接口是用来表示功能的，实现了该接口就意味着具有某些功能，分为多个接口的目的是为了实现接口隔离和重用。





## 数组类型集合

数组类型集合表示在内存上分配是连续的，可以通过下标来访问。

* Array:

特点：内存连续，且元素类型一致，在声明时就已经分配好了空间，长度不可改变；通过下标读取快，增删慢。

使用：

Int[] ii = new int[10];

* ArrayList:

特点：内存连续，元素类型可以不一致，都是当作object类型来处理，所以存在装箱和拆箱的问题；长度可变。

使用：

ArrayList arrayList = new ArrayList();

arrayList.Add("joy");

arrayList.Add(23); //添加元素，并且会增加数组长度

arrayList[2] = "wman"; //索引赋值，不会增加长度，但是如果长度超出当前的索引长度，会报错

object value = arrayList[0]; //读取

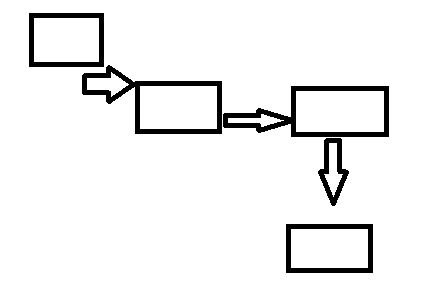
arrayList.RemoveAt(0); //删除

* List<T>：

特点：内存上连续分配；具有泛型的特点，避免了装箱和拆箱；长度可变。

使用：略

## 链表类型集合



* LinkedList<T>：链表

特点：具有泛型的特点；内存上不是连续分配的，每个元素都有记录前后节点；不能通过下标访问(因为下标访问的前提是内存连续，如果要访问的元素只能遍历元素，因此查找不方便)；增删元素方便(修改节点的指向)。

使用：

LinkedList<int> linkList = new LinkedList<int>();

linkList.AddFirst(1);

linkList.AddLast(3);

bool isContain = linkList.Contains(1); //判断元素是否存在

LinkedListNode<int> one = linkList.Find(1); //查询元素

linkList.AddBefore(one, 0); //在1之前添加元素0

linkList.AddAfter(one, 2); //在1之后添加元素2

* Queue<T>：队列

特点：先进先出

使用：

Queue<string> queueList = new Queue<string>();

queueList.Enqueue("joy"); //入队列

queueList.Enqueue("joy1");

//遍历元素

foreach (var item in queueList)

{

Console.WriteLine(item);

}

Console.WriteLine(queueList.Dequeue()); //获取并移除第一个元素

Console.WriteLine(queueList.Peek()); //获取第一个元素，但是不会从队列里移除元素

* Stack<T>：栈

特点：先进后出

使用：

Stack<string> stackList = new Stack<string>();

stackList.Push("joy"); //入栈，每次都是添加在最末尾

stackList.Push("joy1");

//遍历元素

foreach (var item in stackList)

{

Console.WriteLine(item);

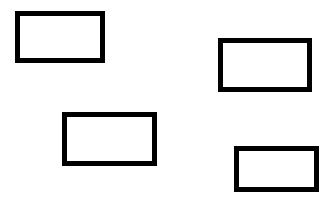
}

Console.WriteLine(stackList.Pop()); //出栈，并移除

Console.WriteLine(stackList.Peek()); //出栈，不移除

## Set类型集合

内存不是连续存储，且能够达到去重的效果。



* HashSet<T>

特点：长度是可变化的；元素之间没有关系并且能够去重(自动去重：每个元素添加进集合是，会对元素通过一个算法计算出元素在内存中的位置，因此如果两个元素一样计算出的位置就会一样，因此就达到了去重的效果)。

使用：

HashSet<string> hashList = new HashSet<string>

{

"joy",

"joy",

"joy1"

};

HashSet<string> hashList1 = new HashSet<string>

{

"joy",

"joy2"

};

hashList1.SymmetricExceptWith(hashList); //得到补集，同时修改了hashList1

hashList1.UnionWith(hashList);//并集，同时修改了hashList1

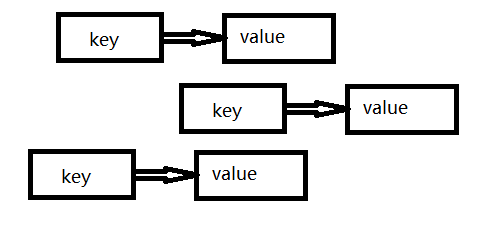
hashList1.ExceptWith(hashList); //差集，同时修改了hashList1

hashList1.IntersectWith(hashList); //交集，同时修改了hashList1

* SortedSet<T>：排序的集合

特点同hashSet。

## Key-value类型的集合



* HashTable

特点：

* 存储：以key-value的形式存放数据，可以是任何类型(都当作是object类型来存放)；在内存中存储不连续，存放在内存的哪个位置是通过key的值来计算出来的位置存放当前这一组key-value(如果计算过程中，不同的key得到了相同的地址，那么后一个地址就会直接+1存放，此时就会得到一个类似于数组一样连续存储的结构)；
* 查找：通过key值计算地址来查找；这种方式查询数据的效率会高很多。但是如果重复的地址太多，就会形成像查询数组一样的下标查询(key计算得到的相同地址，地址中找到值不是指定要找的key，就会将地址+1继续查找，直到找到指定的值为止，这种形式就降低了查询效率)
* 增加，删除，修改的效率高。理论上查询的效率也高；存放的数据无序；是线程安全的，不允许多个线程同时访问；

缺点：存放key，value浪费空间。

使用：

Hashtable hashtable = new Hashtable();

hashtable.Add("11", 111);

hashtable[12] = 222;

hashtable[13] = "joy";

* Dictionary<T1,T2>：字典

特点：具有泛型的特点；存放数据是有序的，是添加的顺序；不是线程安全的，可以多个线程同时修改（可能会发生覆盖）

使用：

Dictionary<int, string> keyValuePairs = new Dictionary<int, string>();

keyValuePairs.Add(1, "11");

* SortedDictionary<T1,T2>：排序的字典

特点：按照key排序

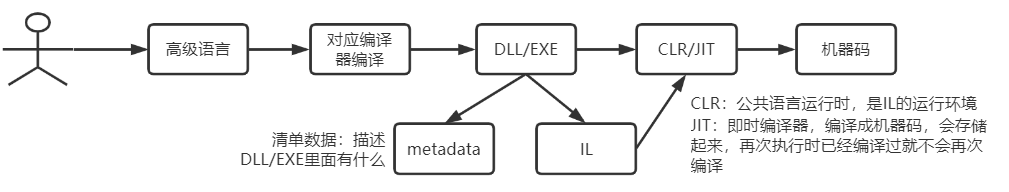
使用：

SortedDictionary<int, string> keyValuePairs = new SortedDictionary<int, string>();

keyValuePairs.Add(2, "22");

keyValuePairs.Add(1, "11");

# CLR



## 内存分配

* 分配形式：

在操作系统里面：内存是链式分配的(链表形式)，可能存在碎片

在CLR里面：由CLR来分配程序运行时，进程所占有的内存，内存是连续分配的(数组形式)，空间由限制，节约空间。

* 进程栈

先进后出的数据结构，随着线程分配的，默认执行方法是分配1MB的内存；每次方法调用都有线程栈，结束就会自己结束，并且释放变量-值类型。

* 对象堆

进程中独立划分出来的一块内存，有一些对象可能不需要释放，放在堆里面可以达到对象的重用，一个进程里就一个堆，空间有限制，所以需要垃圾回收。

* 数据类型
* 值类型

值类型分配在线程栈上面，变量名和值都位于线程栈

* 引用类型

引用类型分配在堆上面，但是变量名在线程栈上(存放的是对象的引用)，值在堆上

new 类名()执行的步骤：

1：new的时候去堆上面开辟内存空间，分配地址

2：调用构造函数(为什么要先开辟空间后调用构造函数？因为在构造函数里面可以使用this)

3：把引用地址赋值给变量

* 装箱和拆箱

仅仅是指内存的拷贝动作，比较浪费性能

* 字符串的内存分配

String value=”value”🡺string value=new string(“value”)赋值的步骤：

1：在堆里面开辟空间，放在“value”值

2：将引用地址赋值给变量名称

* 特性

1：不可变性

Value=”value1”：为value重新赋值，但是并没有修改变量原本指向的内存，而是重新开辟了一块内存，存放新的值，再将新的地址赋值给变量

2：共享性(享元模式)

CLR在赋值的时候会去检查值是否存在，如果存在就会重用

* 为什么字符串不可变？

1：可能有多个变量指向同一个地址，如果字符串变化了，其他变量就会受到影响

2：堆里面的内存是连续分配的，如果修改了内存里面的东西，就可能会导致大量的内存移动，浪费了性能。

string string1 = "string";

string string2 = string1; //把地址赋值一份给string2

Console.WriteLine($"{string1},{string2}"); //"string,string"

string2 = "string2";

Console.WriteLine($"{string1},{string2}"); //"string,string2"

string string3 = "string3";

string string4 = "string4";

string4 = "string3";

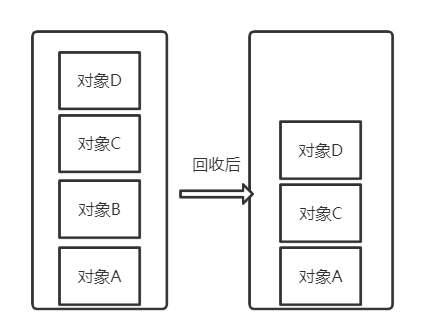
Console.WriteLine(ReferenceEquals(string3,string4)); //true，是否指向同一个引用

## 垃圾回收

主动GC：GC.Collect()

* 垃圾：完全访问不到的内存
* 回收的时机：

1：new一个对象的时候，如果内存不够(这个内存通常指的是1级对象的内存，有一个最大值，但不是指所有的堆空间)，就会发生垃圾回收，释放资源。释放资源之后重新摆放为连续的位置(此时是不允许其他内存操作的)



2：程序退出时也会GC

* 垃圾回收的两个优化策略
* 分级策略(越近分配的内存越容易被GC)：

1：首次发生GC之前，将所有的对象标记为0级

2：第一次GC之后，还存在的对象标记为1级

3：再次发生GC时，优先回收0级对象，如果空间不够，再回收1级对象；之后还存在的对象标记为2级

4：如果回收到了2级对象，空间还是不够，此时就会发生内存溢出的错误

* 大对象策略：

如果一个对象大于85k，这个对象就会以链表的形式单独管理，避免频繁的内存移动

* 析构函数和Dispose()
* 析构函数

在对象销毁时系统自动调用，主要是用来释放非托管资源。在垃圾回收时遇到带有析构函数的对象时，会对其作单独处理(将对象放入一个队列去销毁，时间不确定)

* Dispose()

开发人员Dispose()函数，就要实现Idispose接口。

这个方法是由开发人员主动去调用的，其方法本身没有任何意义，我们需要在方法内部自己去实现对非托管资源的释放。

GC.SuppressFinalize(this)：通知GC不再调用析构函数

* 将对象置为null能够促进GC?
* 普通对象=null；不会促进对象的GC，因为对象实在不能访问之后才会被GC，所以即使不置为null，只要没有被访问就会被回收。
* 静态对象=null；会促进对象被回收，因为将静态对象置为null，就意味着对象没有被访问了，能够被回收。

# ICO/DI、AOP

## IOC(控制反转)

* 是什么：借助第三方来实现具有依赖关系对象间的解耦。

Ioc把高层对底层的依赖，转移给第三方来决定，避免高层直接依赖于底层对象。使得程序架构具有良好的扩展性和依赖性

* 目的是什么：为了程序更方便的分层和解耦

IPhone phone = new IPhone(); //传统

IPhoneInterface phone1 = new IPhone(); //去掉左边细节，依赖于抽象

IPhoneInterface phone2 = ObjectFactory.CreatePhone(); //将具体实现交给第三方，工厂通过反射和配置文件来创建具体对像，ioc初步模型

## DI(依赖注入)

一种实现IOC的方式，即在程序构造对象时，可以自动的通过反射去初始化对象所需要的东西。主要实现方式有：构造函数注入，属性注入，方法注入

* 构造函数注入

根据名称可知，是在创建对象实例时调用指定的构造函数，通过构造函数参数来实现对某对象实例化。

由于类的构造函数默认就是调用参数最多的构造函数，所以可以不使用特性来完成注入；但是也可以通过特性来指定调用哪个构造函数；此种方式也可以避免使用不同IOC实现库带来的改变。

以Unity为例：

[InjectionConstructor] //构造函数注入，默认找参数最多的构造函数

public IPhone(AbstractPhoneButton button)

{

PhoneButton = button;

Console.WriteLine("iphone is created");

}

* 属性注入

在创建对象调用完构造函数之后，自动创建带有指定特性的属性实例，以此来实现对象的自动创建。缺点是需要依赖特性来实现。

以Unity为例：

[Dependency]

public AbstractPhoneButton PhoneButton { get; set; }

* 方法注入

在调用完类的构造函数之后，自动调用带有指定特性的方法，来实现对对象属性的创建与赋值。缺点是增加了一个没有实际意义的方法，只是为了给属性赋值,破坏了类的封装性

[InjectionMethod] //方法注入，并且能够自动创建方法所需要的参数实例

public void PhoneInit()

{

Console.WriteLine("正在方法注入，可以创建对象");

}

## AOP(面向切面)

# C#新语法