目录

一些词汇

<table>
<tr><td>Anti Pattern</td><td>Anti Pattern 表示的是开发时做了一个似乎很好的决策，但后来发现决策是坏的(解决一个 bug 出现两个 bug)</td></tr>
<tr><td>base 构造函数</td><td>在继承父类时候，填写构造函数的时候可以在()后加上：$base(parm)$ 就可以调用父类构造参数，或者$base.method$来调用父类方法</td></tr>
<tr><td>Dry</td><td>Software Practice 的其中一个：不要代码重复</td></tr>
<tr><td>MVC 架构</td><td>MVC 架构表示的是 Model, view, controller
<table>
<tr><td>Model</td><td>View</td><td>Controller</td></tr>
<tr><td>数据，功能 - 程序的内核</td><td>GUI - 软件的外观</td><td>用户可以在 GUI 上做的操作</td></tr>
</table></td></tr>
<tr><td>UML 图表</td><td>UML 图表可以展现出一整个代码项目的框架，对于每个 class，我们需要写上重要的方法和变量
<table>
<tr><td>-</td><td>Private</td></tr>
<tr><td>+</td><td>Public</td></tr>
<tr><td>#</td><td>Protected</td></tr>
<tr><td><u>underline</u></td><td>static</td></tr>
</table>
<table>
<tr><td>Student</td></tr>
<tr><td>$-name: String$
$-uid: int$</td></tr>
<tr><td>$+GetGpa(): Number$
$+CompareStudent(s1, s2): bool$</td></tr>
</table>
并且，Class 之间可以用箭头表示关系，比如 student 和 course 的关系，他们互相独立，一个消失另一个不会，这种弱拥有关系就用空心菱形表示，而强拥有(如学生和成绩)，则是实心菱形箭头表示(学生指向 GPA).</td></tr>
<tr><td>Struct & Class</td><td>大多数时候，我们都会使用 class，但还有一个关键词 - struct. 这个跟 class 极其相似，都是一个类，里面可以包含方法. 但不同的是在实例 struct 后，它是放在栈上面的，而 class 的数据是放在堆上面. 但如果一个 class 里包含 struct，那这个 struct 还是会被存放在堆里面. 还有一点就是 Struct 是 pass by value，而 class object 是 pass by reference</td></tr>
<tr><td>Extensions Syntax</td><td>拓展可以让我们把 C#内置的 class 使用 “.” 来调用我们自己的方法，这样就不用写静态方法了，虽然看起来像是在修改老代码，但实际上是增加新代码功能:
如:

```
1.  static class My_Extensions{//as always choose your own name
2.    public static TYPE function(this TYPE input, ... other parameters){
3.    ...
4.    }
5.  }
```

如我们在 type 填入 string，然后就可以调用字符串实例的我们自己定义的函数了
但这么写不能访问 TYPE 的私有变量</td></tr>
<tr><td>Reflection & Serialization</td><td>Reflection 表示的是代码在运行时可以查看自己的变量，只需要输入 this 就可以.
Serialization 表示的是将类的实例转化成可读取/存储的数据形式(如 XML, Json), 序列化需要反射才可以做到.</td></tr>
<tr><td>Software Practice</td><td>Software practice 表示的是一系列操作让代码更加的易懂.
- 用一些更清晰的变量/方法名，一下就能看懂
- 避免代码复用，因为每次复制粘贴都可能粘贴错误的代码(Dry)
- 用好版本控制和 MSTest</td></tr>
<tr><td>magic Button</td><td>是一种反面模式的例子，就是填完所有表格最后的提交按键，如果其中一项错误，就得完全重新填写.</td></tr>
<tr><td>Judy’s Lay</td><td>如果小问题不重视到后面可能会出现大问题</td></tr>
<tr><td>Profiler</td><td>性能分析器，查看消耗了多少内存</td></tr>
<tr><td>Variable</td><td>
<table>
<tr><td>Const Variable</td><td>常量，永远不会变化，创建必须声明值</td></tr>
<tr><td>ReadOnly Variable</td><td>在对象生命周期中不可变</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td><table>
<tr><td>Global Variable</td><td>全局变量，所有地方都可以访问&修改</td></tr>
<tr><td>Static variable</td><td>静态变量，只有一个实例，对象共享，声明贯穿程序，可以和 Global 一起使用，加 static 后其他文件不能访问</td></tr>
<tr><td>Instance Variable</td><td>实例变量，生命跟对象绑定，会比所以可能会比局部变量更占内存</td></tr>
<tr><td>Local Variable</td><td>局部变量，只有在定义他们的相同代码块才可访问，代码执行完销毁</td></tr>
</table></td></tr>
<tr><td>SOLID</td><td>其实还是跟好代码有关: Solid 表示尽可能的少写错误，而为了做到这点，就需要代码容易理解, DRY ...
具体是以下五点:<table>
<tr><td>Single Responsibility</td><td>每个 Class 最好只负责一个功能，需要其他功能调用就行</td></tr>
<tr><td>Open Closed Principle</td><td>不应该修改旧代码(可能产生 bug)</td></tr>
<tr><td>Interface Segregation</td><td>Class 的接口应该做的比较简单，如 Dispose 接口只有一个 Dispose 方法</td></tr>
<tr><td>Dependency Inversion Principle</td><td>创建实例可以用接口创建，这样以后如果想更改成其他的接口实现 class 就很容易(Abstract SpreadSheet)</td></tr>
<tr><td>Liskov Substitution Principle</td><td>父类的变量在继承到子类之后可能需要变更，且父类替换到子类，输出依然保持一致., 如正方形继承长方形，但正方形不能同时又不同的宽和高.
这个容易和 Dependency Inversion 冲突.</td></tr>
</table></td></tr>
<tr><td>Testing</td><td><table>
<tr><td>Unit Test</td><td>单独测试每个方法来看运行是否正常</td></tr>
<tr><td>Incremental Testing</td><td>Unit Test 之后测试接口之间交互是否正常</td></tr>
<tr><td>Regression Testing</td><td>增加新代码后应该增加新的测试来检测有没有新 bug</td></tr>
<tr><td>Smoke Tests</td><td>初步测试，检测是否能正常运行</td></tr>
<tr><td>Test Suites</td><td>一堆测试合一起就是一个测试套件</td></tr>
<tr><td>White Box Testing</td><td>从代码层面查看代码是否有误(也叫 Transparent box)</td></tr>
<tr><td>Black Box Testing</td><td>检测输出是否正确，不需要明白程序的算法</td></tr>
<tr><td>Gray Box Tesing</td><td>黑盒白盒两个一起做</td></tr>
<tr><td>Integration Testing</td><td>集成测试，一次测试多个模块，查看模块之间的交互</td></tr>
</table></td></tr>
<tr><td>Moore’s Law</td><td>摩尔定律：每两年晶体管密度就会翻一倍，并且由于 Dennard Scaling，同样数量的晶体管就算变小了但功率不变</td></tr>
<tr><td>Concurrent</td><td>Concurrent - 并发表示的是多个任务在一起运行，但这个并不代表程序真正在并行. 如果程序没有在并行但程序看起来像是，这是因为电脑对于每个任务都做一点点工作然后立刻跳到下一个任务. 这样就可以看起来像是多个任务一起运行
如果通过并行达成的并发那么久代表使用多个线程来同时干多个工作</td></tr>
<tr><td>Race Condition</td><td>两个或更多的线程在同时访问同个数据</td></tr>
<tr><td>Lock</td><td>阻止其他线程同时访问同个数据，出现 race condition</td></tr>
<tr><td>Dead Lock</td><td>如果两个线程互相锁住，那么两个线程都会卡住</td></tr>
</table>

Git 的常用操作

- **配置 Git**:
  - git config --global user.name "你的名字"：设置全局用户名。
  - git config --global user.email "你的邮箱"：设置全局邮箱。
- **创建仓库**:
  - git init：初始化本地 Git 仓库。
  - git clone <url>：克隆远程仓库到本地。

- **修改与提交**:
  - git add <file>：添加指定文件到暂存区。
  - git commit -m "message"：提交暂存区到本地仓库，并附上提交信息。
- **分支管理**:
  - git branch <name>：创建新分支。
  - git checkout <name>：切换到指定分支。
  - git merge <name>：将指定分支合并到当前分支。
- **远程操作**:
  - git push origin <branch-name>：将本地分支推送到远程仓库。
  - git pull：拉取远程仓库的变更并合并到本地。
- **撤销与回退**:
  - git checkout <file>：撤销工作区的修改。
  - git reset --hard HEAD^：回退到上一个提交状态。

Properties

在 C#中，对于变量我们可以称作 field，但是还有一种名字叫做 Properties:

```
1.  public class TimePeriod
2.  {
3.      private double _seconds;
4.
5.      public double Hours
6.      {
7.          get { return _seconds / 3600; }
8.          set
9.          {
10.             if (value < 0 || value > 24)
11.                 throw new ArgumentOutOfRangeException(nameof(value),
12.                       "The valid range is between 0 and 24.");
13.
14.             _seconds = value * 3600;
15.         }
16.     }
17. }
```

在上面的例子中, seconds 就是 field，里面有 get set 方法，这样我们就可以用

$TimePeriod.Hours$来获取小时数(前提是我们实例化了这个 time period)

```
public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}
```

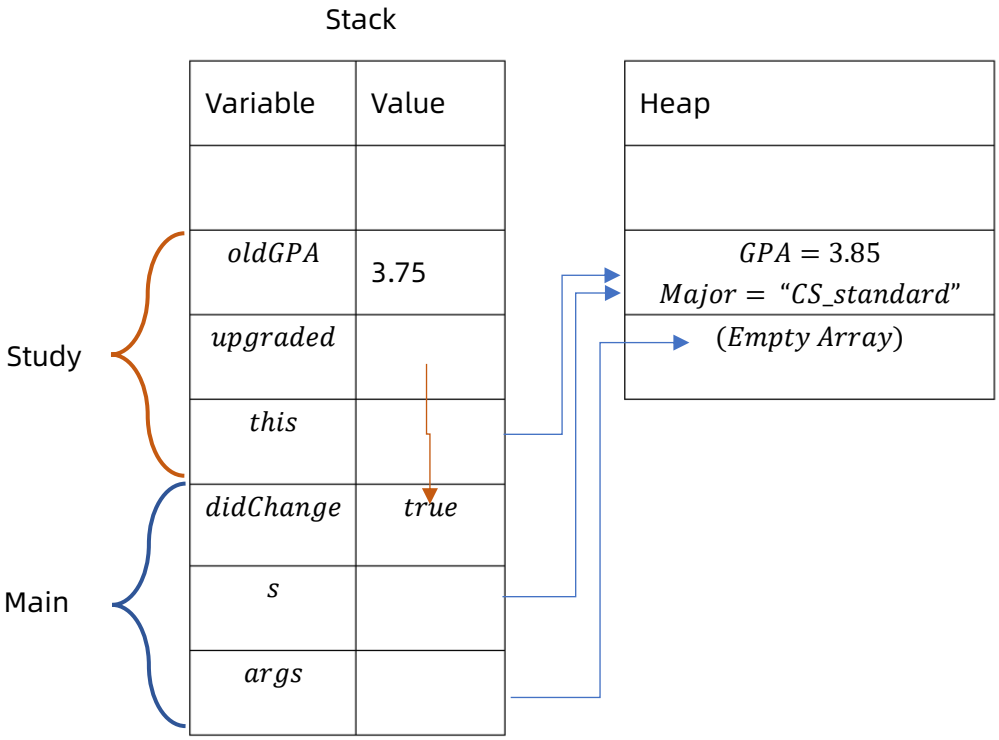
在上面这种情况，我们就命名了一个 Name 属性，但这个属性只有 get method, set method 没有，所以我们可以直接用 lambda 简写出来

```
public class SaleItem
{
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost)
    {
        _name = name;
        _cost = cost;
    }

    public string Name
    {
        get => _name;
        set => _name = value;
    }

    public decimal Price
    {
        get => _cost;
        set => _cost = value;
    }
}
```

Heap & Stack 画法

在 Stack 中，我们从 Main 开始画，然后 stack 存放指针和方法和方法里的 this

```
1.  public class Student
2.  {
3.      private double GPA;
4.      private string major;
5.      public Student(double g, string m)
6.      { GPA = g; major = m; }
7.      public void Study(out bool upgraded)
8.      {
9.          double oldGPA = GPA;
10.         GPA += 0.1;
11.         if (oldGPA < 3.8 && GPA >= 3.8)
12.         {
13.             upgraded = true;
14.             major.Replace("_standard", "_honors");
15.         }
16.         else
17.             upgraded = false;
18.         return;
19.     }
20. }
21. public static void Main(string[] args)
22. {
23.     Student s = new Student(3.75, "CS_standard");
24.     bool didChange;
25.     s.Study(out didChange);
26. }
```



IEnumerable

对于永远无法算完的东西，我们可以用这个关键词.

比如我们想返回一个 List，它包含了所有的质数. 当然，这个根本算不完. 但我们可以把类型改成 IEnumerable，来让代码算出一个就返回一次

```
int possible_prime=3;
IEnumerable next_prime(){
  yield return 2;
  while(true){
```

```
5.      if(is_prime(possible_prime) yield return possible_prime;
6.      possible_prime+=2;
7.    }
8.  }
```

如果要使用就可以用以下操作：

```
1.  static IEnumerator iterator = next_prime().GetEnumerator(); //actually need
    Enum
2.
3.  void button_clicked(){
4.    iterator.MoveNext();//compute next prime and save the state
5.    text_box.Text=iterator.Current;
6.  }
```

这样就可以实现每点击一次就出现下一个质数

在 SS 作业中，我们就用此方法在$GetTokens$中，虽然没有什么必要

Regular Expression

我们通过编写正则表达式，可以设置字符串的要求：

我们能用到的符号其中有：

- () 括号中表示我们要检测的字符
- + 表示我们要检测至少存在一个或一个以上的规定字符
- * 表示我们要检测 0 到无数个规定的字符
- \d 表示的是数字
- [0-9]表示检测从零到 9 的数字

- ^ 放在首位表示开头，放在方括号里表示除了某某字符
- $ 放在末尾表示结束，如abc只会匹配 abc 结尾的字符串，如果 abcd 就不会通过，反之亦然，^符号也是一样

这是一些例子：

- $5*$：是否有 0 个或 0 个以上的 5
- $\backslash d+$：是否存在 0 位以上的数字
- $(-)$：检测是否有减号
- $abc|def$：是否字符串是 abc 或 def
- $[a-gi-z]$：表示除了 h 的所有小写字符

Delegates

Delegate(委托)，跟 JAVA 中的 Function 类型有点相似，他可以让一个函数中固定的地方固定下来，需要变化的地方可以替换成委托，以减少代码量.

比如我们有一个程序需要运算两个数值，但运算有很多种：加减乘除等，但他们的过程都很相似：都是输入两个数字变量，唯一不确定的就是运算符号.

所以，我们可以使用委托来解决：

比如在右侧，我们在 Test 方法中把最终的运算交给委托，我们就可以在外面传入不同的委托来达到不同的目的，不过如果想要使用委托，需要委托(这里是 Cal)和委托绑定的函数(Add/Dec)传入的参数相同，并且绑定的函数需要是 static 的.

```
public delegate double Cal(double x, double y);

internal class DelicateTesting
{
    static void Main() {
        Test(Add);
        Test(Dec);
    }

    public static double Add (double x, double y) { return x + y; }
    public static double Dec (double x, double y) { return x - y; }

    static void Test(Cal f)
    {
        double x = 1.0;
        double y = 2.0;
        Console.WriteLine(f(x, y));
    }
}
```

Func 跟委托很相似，但 Func 可以用在我们并不知道起什么名的时候.

```
Func<Type_Pram_1,...,Type_of_Result>
```

比如:

```
Func<int,int,int>maxFunction = ...
```

表示的就是一个 Function 输入两个 int，返回一个 int

具体什么时候用 Func 取决于我们，就像 Lambda 一样, Lambda 表示的是简写的方法，而 Func 是简写的委托:

```
delegate int MathOnTwoNumbers(int x1, int x2);
MathOnTwoNumbers f1 = (a,b)=>a+b;
Func<int,int,int> f2 = (a,b)=>a+b;
```

这两个是完全相等的，只不过如果我们想生成多个不同的委托如加减乘除，直接定义出来一个委托会稍微方便一点

但需要注意的是, Func 是无法定义返回 Void 的委托，如果我们想定义一个委托只输入函数，我们需要 Action 关键词:

```
Action<Type_Param_1>;
```

Lambda

Lambda 是 software practice 中的一个手段，可以让代码更加简洁，因为它能让代码的功能很清晰的展现出来.

Closure 是 Lambda 的一个特性，它会让函数中的变量本该消失的时候保留下来

```
Func<string,int> GetVariableEvaluator()
{
```

```
    int i = 1;
    return (s) => i++;
}
```

如上面这段代码是一个输入一个 string 返回 int 的函数，通过以下方法调用：

```
var variableVal= GetVariableEvaluator();
int i = variableVal("a");
```

它最开始会返回 1，然后 2, 3... 因为 i 并没有消失，由于 closure 保留了下来

Pass by reference

在 Java / C# 中，我们都认为传参的时候，如果这个参数是一个 Object，那么传递的就是引用. 这个引用指向的是存放在 heap 上的 object. 这就是为什么就算在一个函数中修改 Array list 不用 return 也可以保存函数的修改.

但实际上，这并不是传递引用，而是传递数值，我们传递的是地址的数值，而不是真正的地址，虽然听起来和直接传递引用一样，但是单纯传递地址的值就会让我们虽然可以通过这个地址找到指向的 Object，但是无法直接修改这个指针指向的位置(可以把这个地址的值想象成 primitive type)

这就是为什么我们在函数中把传进来的 ArrayList new 一个新的，它并不会修改函数外的那个 ArrayList，而是创建出一个新的指针指向新的 ArrayList

在 C#中，可以使用in, ref, out在传参中有一些额外的功能

如果在类型前加上 in，则我们不能修改这个参数指向的地址

```
void Change(in Student s){
      s=new Student();//Illegal
      s.Name="a";  //Legal
}
```

如果在类型前加上 ref，那么我们可以修改物体的地址

```
static void Main(string[] args)
{
    // 创建一个对象
    Person p1 = new Person("Alice", 20);
    // 传递对象的引用的值
    ChangePerson(p1);
    // 输出结果：Alice, 21
    Console.WriteLine($"{p1.Name}, {p1.Age}");
    // 传递对象的引用的引用
    ChangePerson(ref p1);
    // 输出结果：Bob, 22
    Console.WriteLine($"{p1.Name}, {p1.Age}");
}
```

如果类型外加上 Out，那么我们可以利用 out 传出多个值

```
static void GetMaxAndMin(int a, int b, out int max, out int min)
{
    //在方法中必须为 out 参数赋值
    if (a > b)
    {
        max = a;
        min = b;
    }
    else
    {
        max = b;
        min = a;
    }
}

//调用方法时，必须在实参前加上 out 关键字
int x = 10;
int y = 20;
int max, min;
GetMaxAndMin(x, y, out max, out min);
Console.WriteLine($"最大值是{max}，最小值是{min}"); //输出：最大值是 20，最小值是 10
```

Nullable

在写代码的时候，经常会被 null 所困扰：只要访问 null 就会崩溃.

但 VS 提供了 nullable 开关，如果开启它，我呢就无法让我们声明一个物体是 null. 如果我们一定想声明，我们可以用问号：

```
1.  Object? X;//Can contain null
```

如果没有加问号，我们的所有 null 的 object 都会高亮出来, (也可以选择在 IDE 中选择如果是 null 直接报错).

需要注意的是, nullable 只是为了让我们更好分辨哪些是 null 哪些不是，但如果我们创建了一个为 null 的物体，然后访问它，就算开启 nullable 仍然会报错

Using

在使用程序中，经常能遇到 “此文件正在使用， 无法命名”, using 关键词也可以让我们实现这个.

```
1.  using(FileReader reader = ...){
2.      reader.DoStuff()
3.  }
```

具体来说就是：

```
1.  FileReader reader = ...
2.  Try{
3.    reader.DoStuff()
4.  }Finally{
5.    reader.Dispose()
6.  }
```

最后的 Finally 部分是表示的是不管 try 部分成没成功，都会运行这部分代码

当然我们正常使用的时候用上面的就行，下面只是告诉 using 到底干了什么

如果在新版 VS 中，可以去掉花括号，节省缩进

```
using FileReader reader = ...;
reader.DoStuff()
```

并行计算

并行计算虽然能给我们代码多倍速度，但也会让代码更难维护

对于线程数量，我们的 main 函数就占用一个线程，如果我们这么写：

```
Thread thread1 = new Thread(DoProgramComputationalWork);
Thread thread2 = new Thread(DoProgramComputationalWork);

thread1.Start();
thread2.Start();
thread1.Join();
```

就是创建两个额外的线程，所以一共是三个线程

最后使用线程合并来让代码等待线程 1 运行完，再执行下面的代码，不然三个线程各干各的

在多线程计算的时候，我们可能多个线程访问同个数据，这样就会造成错误：

```
    public void Demo()
    {

        Thread thread1 = new Thread(DoProgramComputationalWork);
        Thread thread2 = new Thread(DoProgramComputationalWork);

        thread1.Start();
        thread2.Start();

        Stopwatch watch = new Stopwatch();
        watch.Start();

        while ( thread1.IsAlive || thread2.IsAlive )
        {
            Console.WriteLine( "count = " + Count );
            Thread.Sleep( 1000 );  // Don't spew too much to the cons
ole

        }

```

```
21.         watch.Stop();
22.
23.         Console.WriteLine("Took: " + watch.ElapsedMilliseconds + " mi
    lliseconds");
24.
25.         Console.WriteLine("Final value of count = " + Count);
26.         Console.Read();
27.     }
28.
29.
30. private readonly int WorkIterations = 1_000_000_000;
31.     public void DoProgramComputationalWork()
32.     {
33.         for ( int i = 0; i < WorkIterations; i++ )
34.         {
35.             Count++;
36.             Count--;
37.         }
38.     }
```

如上面这个代码，会发现在单线程情况下，每次 print 都应该是 0，但是由于在多线程一起相加/减的时候，可能第一个线程计算完还没写进去数据呢，就被另一个线程读了，导致数据没有减少/增加．所以最后 count 不为 0，这种现象叫 Race Condition，两个线程争着修改

为了解决这个问题，我们可以用线程锁：

```
1.  public void DoProgramComputationalWork()
2.  {
3.      for (int i = 0; i < WorkIterations; i++)
4.      {
5.          lock (Door)
6.          {
7.              Count++;
8.          }
9.          lock (Door)
10.         {
11.             Count--;
12.         }
13.     }
14. }
```

这里的 Door 是一个没有任何用的类，这个 lock 只需要一个 object 就行，填 this 也可以.

但需要注意的是，输入的 object 需要保持一致，如果一个填 Door，一个 this，也会出现计算错误

说白了，我们就用一个指针指向一片内存，用 lock 关键词锁上，如果其他线程访问不了这个内存，就直接暂停执行花括号中的所有代码．所以我们 lock 的地方最好选在那些合并最终结果的地方，那些高开销的运算就放在 lock 外面并行计算

```
Loop - Thread1
Lock(door2){
  Lock(door1){
    Code
  }
}
```

```
Loop - Thread2
Lock(door1){
  Lock(door1){
    Code
  }
}
```

线程死锁：

就像上面的例子，如果两个线程互锁两个指针，就会造成两个线程都卡住然后不干活

异步

异步允许我们同时运行多个任务，但异步和多线程并不是相同的概念，因为异步也可以只在一个线程上运行．异步默认借助线程池，并且不会出现阻塞．

多线程适合长期运行的任务，因为创建和销毁线程开销比使用线程池要大很多．而异步适合小且多的任务，不会出现线程阻塞(显现出来就是程序卡死)的情况

并且，我们还可以再新线程上也一样使用异步，这样的好处是这个线程一样不会堵塞，反而如果我们不使用异步，如果代码不执行完，这个线程就会一直处于阻塞状态

我们使使用 async 在返回值前可以告诉 IDE 这是个异步方法．使用 async 关键字后，我们可在代码中使用 await 关键字，这个 await 可以让代码执行接下来的命令并等待它执行完成(但这并不是线程阻塞，而是换到其他线程在执行，这整个主线程就被切到其他线程上去了，好处就是虽然代码仍在计算，但用户可以同时拖动窗口)

方法的返回值可以是 Task 或者 Void. Task 返回值可以让我们追踪这个异步是在进行还是已经完成或者报错．如果是$Task < int >$就说明这个函数不仅是异步，而且它还有返回值 int

如果使用 Void 作为返回值，我们在调用方法的时候就不能用 await，只能直接调用，因为我们根本不知道任务做没做完．它不是一种很好的写法，因为如果我们写上 Task，他会帮我们自动汇总报错，然而 void 不会，甚至我们在调用前写 try 也没有用，因为已经在另一个线程上了．

```
async Task Main() {
    try{
        await AsyncTask();
    }catch ( Exception ex ) {
        //Cannot catch if Async Task return Void
    }
}

async Task AsyncTask(){
    await Task.Delay( 1000 );
    throw new Exception();
}
```

虽然这个 void 很危险，但是我们也有必须用它的时候，如有个$Event\ Action\ Event_Name$;

我们如果想把异步方法注册进去，必须使用 Void，因为这是个 Action, Action 是不接受返回参数的．

Dependency Injection

程序中，我们可能在很多地方都用到一个数据，比如数据库，但这就表示在创建这个函数的时候也引用数据库或者实例数据库．如果我们用 Global 关键词就可以避免这个麻烦：在程序的哪个类都可以调用这个变量，如 console 就是一个全局变量，但如果我们想把输出到 Console 的东西输出到文件，我们就得修改所有使用到 Global 的变量，这就比较麻烦．

如果我们传入一个类/接口，然后调用它的 writeLine，就可以随时改动功能

```
1.  void foo( Ilogger output)
2.  {
3.      … do stuff …
4.      output.WriteLine($“info”);
5.  }
```

但是这样就会导致我们以后每次调用都会需要加个额外的参数来传递这个$ILogger$接口的实现

这时候就会在想：直接在构造的时候传入一个存起来一直用不就行了?

```
1.  public class Foo {
2.     private readonly ILogger _output;
3.     public Foo( ILogger output)
4.     {
5.          _output = output;
6.     }
```

其实这就是 Dependency Injection．这看起来有点用，但仔细看，发现如果想更改，还得更改所有实例 Foo 所传入的$ILogger$，这不还是很麻烦吗?

这时候我们就可以把这个传入的参数交给第三方 - 依赖注入管理，这样我们只需要修改管理，然后管理就会在每次实例 Foo 的时候决定到底传入什么

创建分为四步:

1.做一个容器的生成类, 2.注册容器的服务信息, 3.生成容器, 4.通过容器进行服务的调用

```
1.  //Another way to do the DI
2.  ServiceCollection services = new ServiceCollection();
```

```
3.  services.AddSingleton<ILoggerProvider, CustomFileLoggerProvider>();
4.  ServiceProvider sp = services.BuildServiceProvider();
5.  _logger = sp.GetService<ILoggerProvider>().CreateLogger("Debug");
```

这里，代码就生成了一个容器 services，然后注册进去了一个 Singleton - 单例，也就是不管什么时候创建多少个都是同一个对象，还有 Scoped, Transit，分别是每个代码块中生成的实例不一样，和每次实例都是不一样的对象.

接着，我们就生成了一个容器，让他为我们提供实例(服务) - sp，最后我们想拿到实例就直接用$sp.GetService$就行了. (获取服务的类型需要跟注册的接口类型保持一致，不能注册接口类型，获取实现类型服务，不然会获得 null object)

虽然看起来有点脱裤子放屁，但这种情况在团队里比较有用，比如写代码的跟注册服务的人不一样，写下面代码的并不用关系到底是怎么实例化这个的，只需通过接口来使用那些方法就行了

并且有意思的是，依赖注入是跟 async 一样有传染性的：如果一个类用了依赖注入，那么其他需要这个有依赖注入才能创建的类一样也会需要依赖注入，这也就是依赖注入实用的地方：

如我在$StroageImpl$类中使用了$Iconfig$接口的实现类，我们在构造的时候就要获得这个$config$，但实际上，我们根本不用担心这个$config$怎么来的，我们只需要用它就行了：所以直接在类里面写上个$private\ readonly\ IConfig\ config$，其他地方直接用

目前为止，好像还跟正常的完全一样(我们只是通过构造函数从外部获得一个实例)，但是接下来的操作就是 DI 牛逼的地方：

如果我们又有一个 Class Controller，它里面也需要一个$StroageImpl$类来运行，我们也通过构造函数传进去. (当然实际情况可能不止有一个储存类，还有很多其他类也用了更多的类，每个类的构造函数都传进来了外部的类). 这时候，我们就在实例化 Controller 的时候，先用

ServiceCollection services来注册进去我们曾经所有在构造函数中所用到的类，接下来 build service 之后，我们直接来一手$var\ c = servicesProvider.GetServivce < Controller > ()$就创建好了 Controller 类，完全不需要给 Controller 传递我们的储存类(当然创建这个类还要提供 config 类... 依次套娃).

但如果我的 Controller 类还需要几个额外的参数，这就需要稍微改动一下，变成工厂模式:

```
1.  public void ConfigureServices(IServiceCollection services)
2.  {
3.      services.AddTransient<IService, ServiceImplementation>();
4.      services.AddTransient<ClassToCreate>(provider =>
5.      {
6.          var service = provider.GetRequiredService<IService>();
7.          int number = // 获取或计算数值
8.          return new ClassToCreate(service, number);
9.      });
10. }
```

例如我们想要创建的类:$ClassToCreate$需要一个$IService$接口实现和一个 int，我们就可以在注册这个类的服务时去先创建好那些$IService$类实现的实例，再手动组合成一个我们想要的类.

这看起来平平无奇，但是其实有趣的在后面：如果我们就是懒，在构造的时候都不想传入一个实例来作为$ILogger$，就传一个 null，这肯定会出现问题，当然也有解决办法:

- 用$outputter?.WriteLine(“hello”)$；来检测，如果$outputter$不是 null，则执行
- 用经典的$if(outputter! = null)$...
- 或者开启 non-Nullable，强制我们写，就算我们不想写也得写

Network

网络发送消息就是跟现实写信一样，我们如果想要给某个网站发消息就要知道地址，我们通常输入地址(3w)但是实际上电脑只认识数字组成的地址，这就需要 DNS 服务器.

DNS

DNS 就是服务器用来专门解析域名.

IPV6

在以前，我们用的是 IPV4，但由于设备越来越多，可用的地址越来越少，甚至几乎所有人的 WIFI 的地址默认都是 192.0.0.1，然后再用不同的 port 来分别不同的设备

用户发送数据到网关都是用 port80，但 return port 不太一样，因为一个网关要把数据发送给不同的人

Port

电脑有大约 64k 的端口，每个端口一般都只被一个程序使用．很多 port 已经被官方认证属于某个机构,如 port52 是专给 DNS 用的，前几千个 port 都几乎被认领过的，防火墙也默认这些 port 就不能其他程序来进行网络通信

如果有程序用了另一个已经在用的 port，就可能出现问题(如 Visual Studio 会用笔记本电脑上的 port 80)比如程序卡死等.

如果想程序之间通讯也就需要这几个东西就行：我和对面的地址，我和对面的 port

IP

IP，也叫网络协议-Internet protocol (它也可以是地址的意思)

有多种网络协议：UDP, TCP, HTTP...

但是对于传输数据，我们都需要用 BYTE 单位来发送

对于我们自己的程序，我们还可以定义自己的协议.

Thread & Internet

在游戏/其他网络软件中，我们需要多线程执行，因为我们需要同时进行多个操作：接收消息，发送消息，执行程序等，所以就需要并行运算

SQL

SQL 也就是一种数据库，他可以用来给程序储存信息，通常是存在云端．它的格式比较像 Excel，如果我们想从 xxx 数据库中选择名字是 Jim 或者名字>M 的人，我们可以这么写

```
1.  SELECT * FROM xxx
2.  WHERE Name = 'JIM' OR Name > 'M'
```

第一行是选择 xxx 中所有的数据，第二行是筛选

如果两个数据库中有两个数据是匹配的，我们想让两个数据库中的配对的数据合并到一起，

如 Patrons 和 Phone 中都存了 Card number，但 Patrons 还存了名字, Phone 还存了手机号.

我们可以这么写:

```
SELECT * Patrons JOIN Phones
ON Patrons.CardNum = Phones.CardNum
```

第一行是我们想把 Patrons 和 Phones 合并，第二行是合并的规则 - 如果卡号相等就合并

但其实也可以这么写:

```
SELECT * Patrons CROSS JOIN Phones
WHERE Patrons.CardNum = Phones.CardNum
```

CORSS JOIN 就是把 Phone 和 Patrons 两个数据库所有的组合都生成，然后再通过筛选. 当然性能肯定不如上面的

Where 和 On 的区别就是 On 是直接定义了 Join 的规则，并且比 Where 优先执行.

如果还想合并更多的表格，就直接在后面继续写 Join xxx On xxx 就行

不仅可以查找，还可以增删

```
INSERT INTO xxx
VALUES(v1,v2,vn)

DELETE FROM xxx
WHERE xxx

UPDATE xxx
Set x=xx
WHERE xxx
```

Values 表示的是这个表格每行的每个变量的值.''