数组越界的异常

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public class Text : MonoBehaviour {

    void Start()
    {
        Fun2();
    }

    void Fun1()
    {
        int[] a={1,2,3};
        //IndexOutOfRangeException: Array index is out of range.
        for (int i = 0; i < a.Length+1; i++)
        {
            Debug.Log(a[i]);
        }
        Debug.Log("遍历完成");
    }

    void Fun2()
    {
        int[] a = { 1, 2, 3 };

        try
        {
            for (int i = 0; i < a.Length + 1; i++)
            {
                Debug.Log(a[i]);
            }
        }
        catch (IndexOutOfRangeException e)
        {
            Debug.Log(e);
        }
```

```
        finally
        {
            Debug.Log("遍历完成");
        }
    }
}
```

栈溢出的异常处理

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public class Text : MonoBehaviour {
    void Start()
    {
        try
        {
            Fun();
        }
        catch (StackOverflowException e)
        {
            //System.StackOverflowException:
            //The requested operation caused a stack overflow.
            Debug.Log(e);
        }
    }

    //死递归
    void Fun()
    {
        int a = 100;
        Fun();
    }
}
```

用户自定义的异常类

用户自定义的异常必须继承于ApplicationException

例：设置用户血量的百分比 用户传递来的比例值小于0的异常

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;


//自定义的异常类 必须继承于ApplicationException
//原则上需要实现一个string类型的构造方法 用于给父类构造赋值
public class ValueLessThanZero:ApplicationException
{
    public ValueLessThanZero(string msg):base(msg)
    {

    }
}

public class Text : MonoBehaviour {

    //血量比例
    float process;

    void SetValue(float value)
    {
        if (value < 0)
        {
            //当值小于0时 抛出自定义的异常对象
            throw (new ValueLessThanZero("Sorry the value lessThan zero!@@"));
        }
        else
        {
            process = value;
        }
    }

    void Start()
```

```
36      {
37          try
38          {
39              SetValue(-1000);
40          }
41          catch (ValueLessThanZero e)
42          {
43              Debug.Log(e);
44          }
45          finally
46          {
47              Debug.Log(process);
48          }
49      }
50  }
```

迭代器

foreach做遍历很方便 然而并不是每一种类型都能使用foreach进行遍历操作

只有实现了IEnumerable接口的类(可枚举类型) 才可以进行foreach的遍历操作

集合和数组已经实现了这个接口 所以能进行foreach遍历

IEnumerable和IEnumerator

IEnumerable：可枚举类型

```
namespace System.Collections
{
    [ComVisible(true)]
    [Guid("496B0ABE-CDEE-11d3-88E8-00902754C43A")]
    public interface IEnumerable
    {
        [DispId(-4)]
        IEnumerator GetEnumerator();
    }
}
```

IEnumerator:枚举器类型

```
namespace System.Collections
{
    [ComVisible(true)]
    [Guid("496B0ABF-CDEE-11D3-88E8-00902754C43A")]
    public interface IEnumerator
    {
        object Current { get; }

        bool MoveNext();
        void Reset();
    }
}
```

Current 返回object类型的引用 第一次的位置是-1

MoveNext 把枚举器的位置指向下一项 返回bool 判断是否新的位置有效

Reset 把位置重置到原始状态

使用枚举器遍历数组集合

```
using System;

public class Text : MonoBehaviour {

    void Start()
    {
        int[] array = { 1, 22, 33, 444 };
        //获取枚举器
        IEnumerator ie =array.GetEnumerator();
        while(ie.MoveNext())//类似于i++ 更新循环变量的操作
        {
             //默认current初始值是索引值为-1的元素
            //所以需要在第一次调用Current之前先进行一次MoveNext的调用 移动到下一项
            int i = (int)ie.Current;
            Debug.Log(i);
        }
    }
}
```

定义自己的可枚举类型和枚举器类型（foreach代码的实现）

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

//定义可枚举类型
class MyEnumerable : IEnumerable
{
    public object[] values;
    public MyEnumerable(object[] values)
    {
        this.values = values;
    }

    //实现IEnumerable的GetEnumerator方法
    public IEnumerator GetEnumerator()
    {
        return new MyEnumerator(this);
    }
}

//定义枚举器类型
class MyEnumerator : IEnumerator
{
    private MyEnumerable data;//迭代对象
    private int position;//游标 索引值

    public MyEnumerator(MyEnumerable data)
    {
        this.data = data;
        //数组元素下标从0开始  初始时默认当前当前游标为-1
        this.position = -1;
    }

    //MoveNext负责更新游标position
    public bool MoveNext()
    {
```

```
        if (position != data.values.Length)
        {
            position++;//更新游标
        }
        Debug.Log("Call MoveNext");
        //返回是否移动到最后一项
        return position < data.values.Length;
    }

    //Current属性 用来得到对应游标的元素
    public object Current
    {
        get {
            //数组越界返回空
            if (position == -1 || position == data.values.Length)
                return null;
            return data.values[position];
        }
    }

    //Reset 重置方法
    public void Reset()
    {
        position = -1;//将游标重置为-1;
    }
}

public class Text : MonoBehaviour {

    void Start()
    {
        object[] nums = new object[] { 12, 22, 3, 412, 123 };
        MyEnumerable intEnum = new MyEnumerable(nums);
        foreach (var temp in intEnum)
        {
            Debug.Log(temp);
        }
    }
}
```

综上代码 C#手动实现非泛型迭代的功能 代码是相当繁琐的

如果是泛型迭代 就需要使用IEnumerator<T> 和 IEnumerable<T> 两个泛型接口 代码会更加繁琐

替代方案：迭代器

C#2.0以后更新的迭代器 迭代器会生成可枚举类型和枚举器类型

迭代器中的关键语句： yield return

迭代器简单理解：使用一个或多个yield return语句 告诉编译器创建枚举器类

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

class MyEnumerable
{
    public IEnumerator<int> GetEnumerator()
    {
        //原本创建枚举器需要实现IEnumerator接口
        //直接创建GetEnumerator无参 返回值是IEnumerator或IEnumerator<T>
        yield return 100;
        yield return 200;
        yield return 300;
        yield return 400;
        yield return 500;//使用yield return 替代了Current MoveNext Rest 这些方法
    }
}

public class Text : MonoBehaviour {

    void Start()
    {
        MyEnumerable x = new MyEnumerable();

        foreach (var item in x)
        {
            Debug.Log("循环一次");
            Debug.Log(item);
```

```
        }
    }
}
```

迭代器可以返回枚举器类型 也可以返回可枚举类型

但是在遍历时有所不同：

```
class MyEnumerable
{
    public IEnumerable<string> GetEnumerable()
    {
        yield return "apple";
        yield return "banana";
        yield return "orange";
    }
}

public class Text : MonoBehaviour {

    void Start()
    {
        MyEnumerable x = new MyEnumerable();

        foreach (var item in x.GetEnumerable())
        {
            Debug.Log("循环一次");
            Debug.Log(item);
        }
    }

}
```

整改：

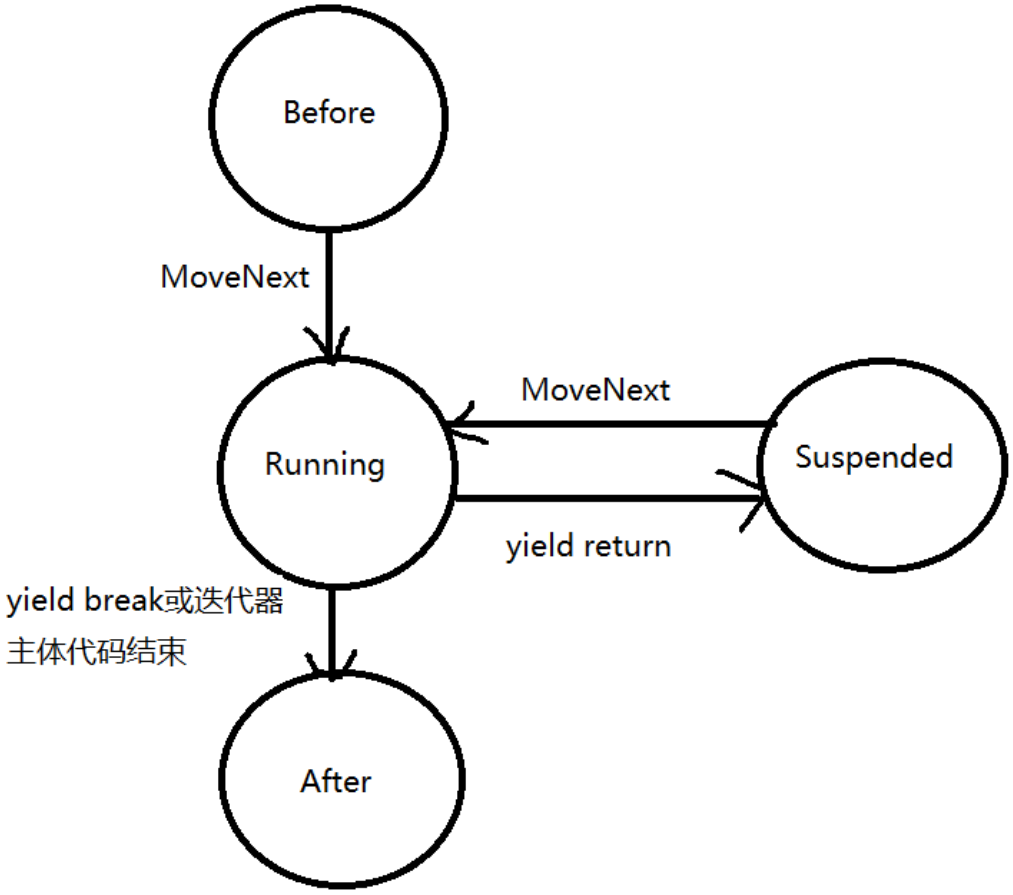
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
```

```
5
6  class MyEnumerable
7  {
8      public IEnumerable<string> GetEnumerable()
9      {
10         yield return "apple";
11         yield return "banana";
12         yield return "orange";
13     }
14
15     public IEnumerator<string> GetEnumerator()
16     {
17         return GetEnumerable().GetEnumerator();
18     }
19 }
20
21 public class Text : MonoBehaviour {
22
23     void Start()
24     {
25         MyEnumerable x = new MyEnumerable();
26
27         foreach (var item in x)
28         {
29             Debug.Log("循环一次");
30             Debug.Log(item);
31         }
32     }
33 }
```

yield return 语法描述：

yield return 可以根据返回类型告诉编译器生成创建可枚举类型还是枚举器类型

yield return 语句指定了枚举器对象的下一个可枚举项



当创建一个可枚举类型（不管是手动实现还是使用迭代器实现）

枚举器实际上都可以看做是包含4个状态的状态机

Before 首次调用MoveNext时的状态

初始位置在第一个可枚举项之前

Running 在调用MoveNext之后进入该状态

在Running状态中 枚举器检测下一项的位置

当遇到yield return语法时会进入挂起（Suspended）状态

在挂起状态再次调用MoveNext之后会退出挂机状态 回到Running

Suspended 等待下一次MoveNext的唤醒

After 已经到了迭代器的最后位置 没有可枚举项 或者执行了yield break语法

练习：

写一个自定义类 MyClass

类型中包含一个int类型的数组成员

使用foreach循环遍历MyClass类型的对象

要求遍历MyClass类型中int数组的所有偶数