1. 关键字
2. Checked checked启用运行时溢出检查
3. Unchecked unchecked启用运行时溢出检查
4. Explicit 
5. Extern 引用外部的dll 常与dll Import 一起使用， 将C++等非托管代码导入到当前程序中去；如图所示

（https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/extern）



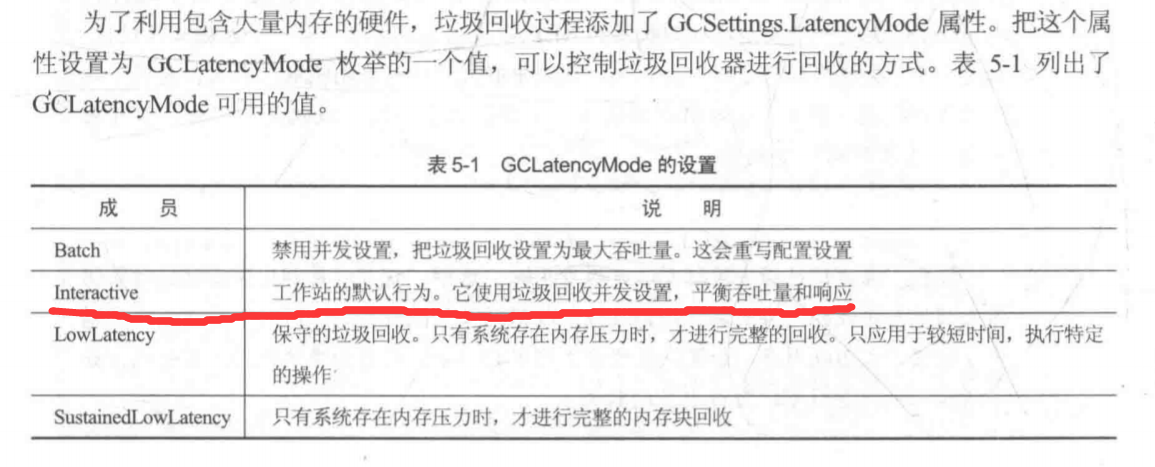
1. Fixed 为托管变量创建一个指针， 在该语句执行过程中单边锁定该变量 如果没有 ，指向可移动的托管变量的指针将几乎没有什么用处，因为垃圾回收可能会不可预见地重新定位变量。 C# 编译器只允许将指针分配给fixed语句中的托管变量。
2. Internal 设置可访问级别 访问仅限于当前程序集
3. Readonly 用以修饰字段， 字段的初始化 可以在声明创建的时候或者 构造的时候
4. Stackalloc 在不安全的代码上 使用stackalloc 在堆栈上面分配内存块 只在初始创建时候有效； 不安全的代码直接访问内存（指针）



1. Volatile 标记一个字段的值可能会被多个线程修改， 每次都会去读取最新的值

https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/volatile

1. Unsafe 在某些情况下，通过移除数组绑定检查，不安全代码可提高应用程序的性能。调用需要指针的本机函数时，需使用不安全代码。
2. Yield 用以生成广义表， 使值能延迟生成。
3. 结构
4. 结构是值类型， 不能被继承， 构造只是为了赋值，
5. 垃圾回收
6. 垃圾回收的改进
7. 将超过85000 字节的对象放入一个特殊堆中， 不放在主堆上面，不执行压缩。回收时， 用第二个线程去处理， 主线程的只需要关注第0，1suo代回收，减少花费时间
8. 垃圾回收的平衡：减少不必要的回收， 例如一个线程占用的内存远大于其他线程所占用的内存， 当这个线程内存占满的时候， 出发回收机制， 引起其他的一起回收， 不高效



1. 强引用忘记回收的情况： 当一个变量被使用过后，设置成null， 但是编译器依旧觉得这个变量还存在着， 所以不能回收-> 使用弱引用（weakreference 类）
2. 非托管资源

  非托管资源指的是.NET不知道如何回收的资源，最常见的一类非托管资源是包装操作系统资源的对象，例如文件，窗口，网络连接，数据库连接，画刷，图标等。这类资源，垃圾回收器在清理的时候会调用Object.Finalize()方法。默认情况下，方法是空的，对于非托管对象，需要在此方法中编写回收非托管资源的代码，以便垃圾回收器正确回收资源。

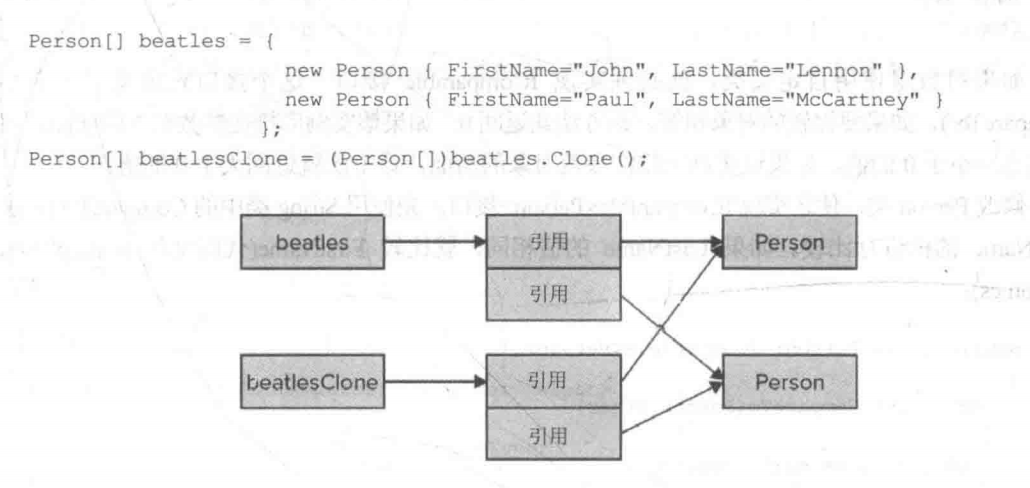
托管资源回收：析构函数中不能释放托管资源； 因为析构函数是垃圾回收器调用的， 可能资源已经释放掉了

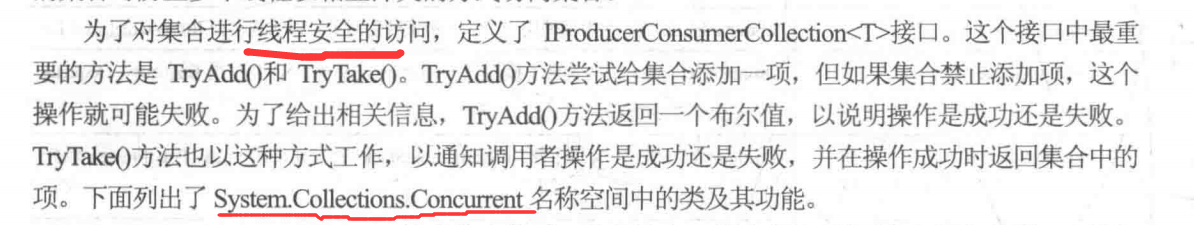
非托管资源回收：IDisposable 接口；实现析构函数， 在析构函数中释放资源

推荐用IDisposable 接口代替析构函数

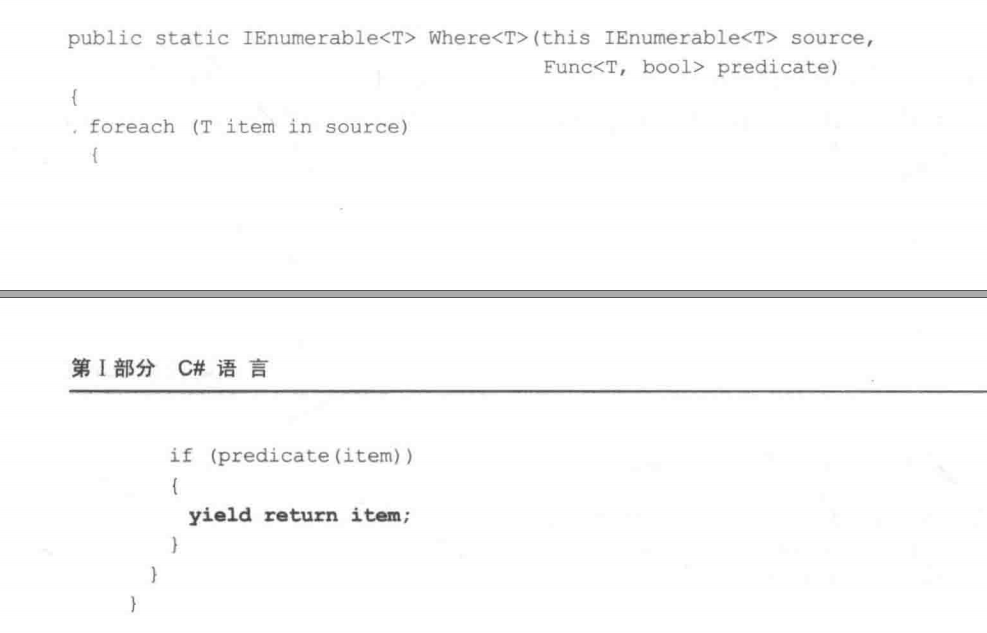
1. 泛型
2. 值类型存储在栈上面， 引用类型存储在堆上面
3. 拆箱装箱， 拆箱是引用类型转值类型（arraylist <int>取int值）， 装箱反之
4. 泛型可以减少不必要的装箱拆箱
5. 泛型约束



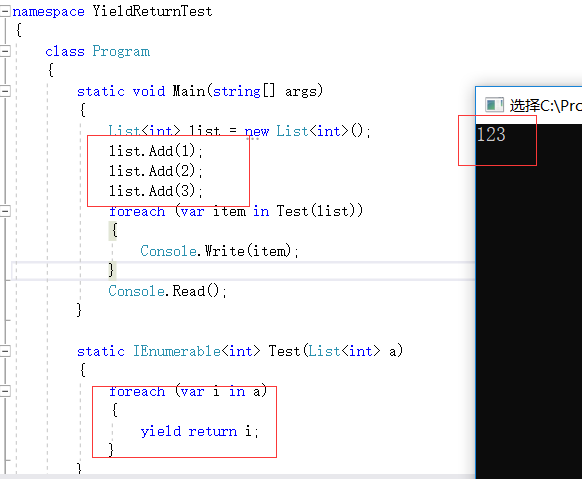
1. 数组
2. 数组中的元素为引用类型时， 当数组复制的时候， 复制引用， 修改时， 另一个也会发生改变
3. 数组排序用到自定义类时， 必须实现IComparable 接口，
4. 委托和事件
5. 委托的目的， 通过观察者模式 实现解耦
6. 用不变的集合来控制线程安全



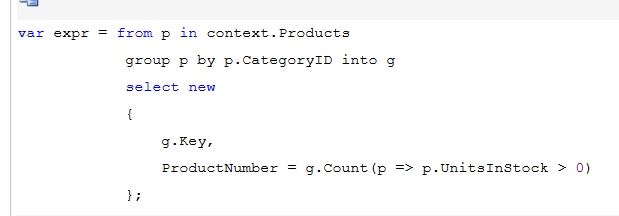
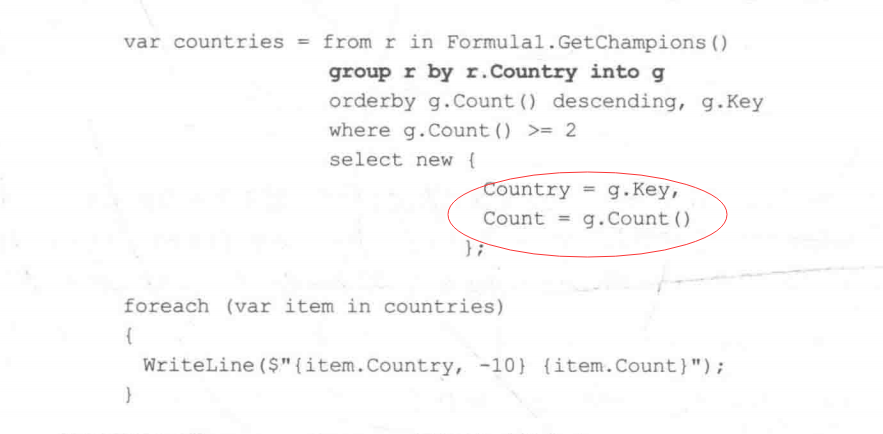
1. 扩展linq where 表达式



1. Yield



1. Linq group by



LINQ 并行查询ParallelEnumerable,

