

C# 中的泛型

Generics In C#

Jesse Liberty

译者: 张子阳

www.tracefact.net

jimmy_dev@163.com

出处: <http://www.ondotnet.com/pub/a/dotnet/2004/05/17/liberty.html>

术语表

generics: 泛型
type-safe: 类型安全
collection: 集合
compiler: 编译器
run time: 程序运行时
object: 对象
.NET library: .Net 类库
value type: 值类型
box: 装箱
unbox: 拆箱
implicit: 隐式
explicit: 显式
linked list: 线性链表
node: 结点
indexer: 索引器

简介

Visual C# 2.0 的一个最受期待的(或许也是最让人畏惧)的一个特性就是对于泛型的支持。这篇文章将告诉你泛型用来解决什么样的问题, 以及如何使用它们来提高你的代码质量, 还有你不必恐惧泛型的原因。

泛型是什么?

很多人觉得泛型很难理解。我相信这是因为他们通常在了解泛型是用来解决什么问题之前, 就被灌输了大量的理论和范例。结果就是你有了解决方案, 但是却没有需要使用这个解决方案的问题。

这篇文章将尝试着改变这种学习流程, 我们将以一个简单的问题作为开始: 泛型是用来做什么的? 答案是: 没有泛型, 将会很难创建类型安全的集合。

C# 是一个类型安全的语言, 类型安全允许编译器(可信赖地)捕获潜在的错误, 而不是在程序运行时才发现(不可信赖地, 往往发生在你将产品出售了以后!)。因此, 在 C#中, 所有的变量都有一个定义了的类型; 当你将一个对象赋值给那个变量的时候, 编译器检查这个赋值是否正确, 如果有问题, 将会给出错误信息。

在 .Net 1.1 版本(2003)中, 当你在使用集合时, 这种类型安全就失效了。由 .Net 类库提供的所有关于集合的类全是用来存储基类型(Object)的, 而 .Net 中所有的一切都是由 Object 基类继承下来的, 因此所有类型都可以放到一个集合中。于是, 相当于根本就没有了类型检测。

更糟的是，每一次你从集合中取出一个 Object，你都必须将它强制转换成正确的类型，这一转换将对性能造成影响，并且产生冗长的代码(如果你忘了进行转换，将会抛出异常)。更进一步地讲，如果你给集合中添加一个值类型(比如，一个整型变量)，这个整型变量就被隐式地装箱了(再一次降低了性能)，而当你从集合中取出它的时候，又会进行一次显式地拆箱(又一次性能的降低和类型转换)。

关于装箱、拆箱的更多内容，请访问 [陷阱 4，警惕隐式的装箱、拆箱。](#)

创建一个简单的线性链表

为了生动地感受一下这些问题，我们将创建一个尽可能简单的线性链表。对于阅读本文的那些从未创建过线性链表的人。你可以将线性链表想像成有一条链子栓在一起的盒子(称作一个结点)，每个盒子里包含着一些数据和 链接到这个链子上的下一个盒子的引用(当然，除了最后一个盒子，这个盒子对于下一个盒子的引用被设置成 NULL)。

为了创建我们的简单线性链表，我们需要下面三个类：

- 1、Node 类，包含数据以及下一个 Node 的引用。
- 2、LinkedList 类，包含链表中的第一个 Node，以及关于链表的任何附加信息。
- 3、测试程序，用于测试 LinkedList 类。

为了查看链接表如何运作，我们添加 Objects 的两种类型到链表中：整型和 Employee 类型。你可以将 Employee 类型想象成一个包含关于公司中某一个员工所有信息的类。出于演示的目的，Employee 类非常的简单。

```
public class Employee{
    private string name;
    public Employee (string name){
        this.name = name;
    }

    public override string ToString(){
        return this.name;
    }
}
```

这个类仅包含一个表示员工名字的字符串类型，一个设置员工名字的构造函数，一个返回 Employee 名字的 ToString() 方法。

链接表本身是由很多的 Node 构成，这些 Note，如上面所说，必须包含数据(整型和 Employee)和链表中下一个 Node 的引用。

```
public class Node{
    Object data;
```

```

Node next;

public Node(Object data){
    this.data = data;
    this.next = null;
}

public Object Data{
    get { return this.data; }
    set { data = value; }
}

public Node Next{
    get { return this.next; }
    set { this.next = value; }
}
}

```

注意构造函数将私有的数据成员设置成传递进来的对象，并且将 next 字段设置成 null。

这个类还包括一个方法，Append，这个方法接受一个 Node 类型的参数，我们将把传递进来的 Node 添加到列表中的最后位置。这个过程是这样的：首先检测当前 Node 的 next 字段，看它是不是 null。如果是，那么当前 Node 就是最后一个 Node，我们将当前 Node 的 next 属性指向传递进来的新结点，这样，我们就把新 Node 插入到了链表的尾部。

如果当前 Node 的 next 字段不是 null，说明当前 node 不是链表中的最后一个 node。因为 next 字段的类型也是 node，所以我们调用 next 字段的 Append 方法(注：递归调用)，再一次传递 Node 参数，这样继续下去，直到找到最后一个 Node 为止。

```

public void Append(Node newNode){
    if ( this.next == null ){
        this.next = newNode;
    }else{
        next.Append(newNode);
    }
}
}

```

Node 类中的 ToString() 方法也被覆盖了，用于输出 data 中的值，并且调用下一个 Node 的 ToString()方法(译注：再一次递归调用)。

```

public override string ToString(){
    string output = data.ToString();

    if ( next != null ){
        output += ", " + next.ToString();
    }
}

```

```
    }  
  
    return output;  
}
```

这样，当你调用第一个 Node 的 ToString() 方法时，将打印出所有链表上 Node 的值。

LinkedList 类本身只包含对一个 Node 的引用，这个 Node 称作 HeadNode，是链表中的第一个 Node，初始化为 null。

```
public class LinkedList{  
    Node headNode = null;  
}
```

LinkedList 类不需要构造函数(使用编译器创建的默认构造函数)，但是我们需要创建一个公共方法, Add(), 这个方法把 data 存储到线性链表中。这个方法首先检查 headNode 是不是 null，如果是，它将使用 data 创建结点，并将这个结点作为 headNode，如果不是 null，它将创建一个新的包含 data 的结点，并调用 headNode 的 Append 方法，如下面的代码所示：

```
public void Add(Object data){  
    if ( headNode == null ){  
        headNode = new Node(data);  
    }else{  
        headNode.Append(new Node(data));  
    }  
}
```

为了提供一点集合的感觉，我们为线性链表创建一个索引器。

```
public object this[ int index ]{  
    get{  
        int ctr = 0;  
        Node node = headNode;  
        while ( node != null && ctr <= index ){  
            if ( ctr == index ){  
                return node.Data;  
            }else{  
                node = node.Next;  
            }  
            ctr++;  
        }  
        return null;  
    }  
}
```

最后，ToString()方法再一次被覆盖，用以调用 headNode 的 ToString()方法。

```
public override string ToString(){
    if ( this.headNode != null ){
        return this.headNode.ToString();
    }else{
        return string.Empty;
    }
}
```

测试线性链表

我们可以添加一些整型值到链表中进行测试：

```
public void Run(){
    LinkedList ll = new LinkedList();
    for ( int i = 0; i < 10; i ++ ){
        ll.Add(i);
    }

    Console.WriteLine(ll);
    Console.WriteLine(" Done. Adding employees...");
}
```

如果你对这段代码进行测试，它会如预计的那样工作：

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Done. Adding employees...
```

然而，因为这是一个 Object 类型的集合，所以你同样可以将 Employee 类型添加到集合中。

```
ll.Add(new Employee("John"));
ll.Add(new Employee("Paul"));
ll.Add(new Employee("George"));
ll.Add(new Employee("Ringo"));

Console.WriteLine(ll);
Console.WriteLine(" Done.");
```

输出的结果证实了，整型值和 Employee 类型都被存储在了同一个集合中。

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Done. Adding employees...
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, John, Paul, George, Ringo
```

Done.

虽然看上去这样很方便，但是负面影响是，你失去了所有类型安全的特性。因为线性链表需要的是一个 Object 类型，每一个添加到集合中的整型值都被隐式装箱了，如同 IL 代码所示：

```
IL_000c: box      [mscorlib]System.Int32
IL_0011: callvirt instance void ObjectLinkedList.LinkedList::Add(object)
```

同样，如果上面所说，当你从你的列表中取出项目的时候，这些整型必须被显式地拆箱（强制转换成整型），Employee 类型必须被强制转换成 Employee 类型。

```
Console.WriteLine("The fourth integer is " + Convert.ToInt32(ll[3]));
Employee d = (Employee) ll[11];
Console.WriteLine("The second Employee is " + d);
```

这些问题的解决方案是创建一个类型安全的集合。一个 Employee 线性链表将不能接受 Object 类型；它只接受 Employee 类的实例（或者继承自 Employee 类的实例）。这样将会是类型安全的，并且不再需要类型转换。一个 整型的 线性链表，这个链表将不再需要装箱和拆箱的操作（因为它只能接受整型值）。

作为示例，你将创建一个 EmployeeNode，该结点知道它的 data 的类型是 Employee。

```
public class EmployeeNode {
    Employee employeeData;
    EmployeeNode employeeNext;
}
```

Append 方法现在接受一个 EmployeeNode 类型的参数。你同样需要创建一个新的 EmployeeLinkedList，这个链表接受一个新的 EmployeeNode：

```
public class EmployeeLinkedList{
    EmployeeNode headNode = null;
}
```

EmployeeLinkedList.Add() 方法不再接受一个 Object，而是接受一个 Employee：

```
public void Add(Employee data){
    if ( headNode == null ){
        headNode = new EmployeeNode(data);}
    else{
        headNode.Append(new EmployeeNode(data));
    }
}
```

类似的，索引器必须被修改成接受 EmployeeNode 类型，等等。这样确实解决了装箱、拆箱

的问题，并且加入了类型安全的特性。你现在可以添加 Employee (但不是整型) 到你新的线性链表中了，并且当你从中取出 Employee 的时候，不再需要类型转换了。

```
EmployeeLinkedList employees = new EmployeeLinkedList();
employees.Add(new Employee("Stephen King"));
employees.Add(new Employee("James Joyce"));
employees.Add(new Employee("William Faulkner"));
/* employees.Add(5); // try to add an integer - won't compile */
Console.WriteLine(employees);
Employee e = employees[1];
Console.WriteLine("The second Employee is " + e);
```

这样多好啊，当有一个整型试图隐式地转换到 Employee 类型时，代码甚至连编译器都不能通过！

但它不好的地方是：每次你需要创建一个类型安全的列表时，你都需要做很多的复制/粘贴。一点也不够好，一点也没有代码重用。同时，如果你是这个类的作者，你甚至不能提前欲知这个链接列表所应该接受的类型是什么，所以，你不得不将添加类型安全这一机制的工作交给类的使用者——你的用户。

使用泛型来达到代码重用

解决方案，如同你所猜想的那样，就是使用泛型。通过泛型，你重新获得了链接列表的代码通用 (对于所有类型只用实现一次)，而当你初始化链表的时候你告诉链表所能接受的类型。这个实现是非常简单的，让我们重新回到 Node 类：

```
public class Node{
    Object data;
    ...
}
```

注意到 data 的类型是 Object，(在 EmployeeNode 中，它是 Employee)。我们将把它变成一个泛型 (通常，由一个大写的 T 代表)。我们同样定义 Node 类，表示它可以被泛型化，以接受一个 T 类型。

```
public class Node <T>{
    T data;
    ...
}
```

读作：T 类型的 Node。T 代表了当 Node 被初始化时，Node 所接受的类型。T 可以是 Object，也可能是整型或者是 Employee。这个在 Node 被初始化的时候才能确定。

注意：使用 T 作为标识只是一种约定俗成，你可以使用其他的字母组合来代替，比如这样：

```
public class Node <UnknownType>{
```

```
UnknownType data;  
...
```

通过使用 T 作为未知类型, next 字段(下一个结点的引用)必须被声明为 T 类型的 Node(意思是说接受一个 T 类型的泛型化 Node)。

```
Node<T> next;
```

构造函数接受一个 T 类型的简单参数:

```
public Node(T data)  
{  
    this.data = data;  
    this.next = null;  
}
```

Node 类的其余部分是很简单的,所有你需要使用 Object 的地方,你现在都需要使用 T。LinkedList 类现在接受一个 T 类型的 Node,而不是一个简单的 Node 作为头结点。

```
public class LinkedList<T>{  
    Node<T> headNode = null;
```

再来一遍,转换是很直白的。任何地方你需要使用 Object 的,现在改做 T,任何需要使用 Node 的地方,现在改做 Node<T>。下面的代码初始化了两个链接表。一个是整型的。

```
LinkedList<int> ll = new LinkedList<int>();
```

另一个是 Employee 类型的:

```
LinkedList<Employee> employees = new LinkedList<Employee>();
```

剩下的代码与第一个版本没有区别,除了没有装箱、拆箱,而且也不可能将错误的类型保存到集合中。

```
LinkedList<int> ll = new LinkedList<int>();  
for ( int i = 0; i < 10; i ++ )  
{  
    ll.Add(i);  
}  
  
Console.WriteLine(ll);  
Console.WriteLine(" Done.");  
  
LinkedList<Employee> employees = new LinkedList<Employee>();  
employees.Add(new Employee("John"));
```

```
employees.Add(new Employee("Paul"));
employees.Add(new Employee("George"));
employees.Add(new Employee("Ringo"));

Console.WriteLine(employees);
Console.WriteLine(" Done.");
Console.WriteLine("The fourth integer is " + ll[3]);
Employee d = employees[1];
Console.WriteLine("The second Employee is " + d);
```

泛型允许你不用复制/粘贴冗长的代码就实现类型安全的集合。而且，因为泛型是在运行时才被扩展成特殊类型。Just In Time 编译器可以在不同的实例之间共享代码，最后，它显著地减少了你需要编写的代码。

本文的源码可以在 <http://www.tracefact.net/SourceCode/Generics-In-CSharp.rar> 下载。