泛型类封装不是特定于具体数据类型的操作。泛型类最常用于集合，如链接列表、哈希表、堆栈、队列、树等。像从集合中添加和移除项这样的操作都以大体上相同的方式执行，与所存储数据的类型无关。

对于大多数需要集合类的方案，推荐的方法是使用 .NET Framework 类库中所提供的类。有关使用这些类的更多信息，请参见[.NET Framework 类库中的泛型（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/ac30fak4.aspx)。

一般情况下，创建泛型类的过程为：从一个现有的具体类开始，逐一将每个类型更改为类型参数，直至达到通用化和可用性的最佳平衡。创建您自己的泛型类时，需要特别注意以下事项：

* 将哪些类型通用化为类型参数。

通常，能够参数化的类型越多，代码就会变得越灵活，重用性就越好。但是，太多的通用化会使其他开发人员难以阅读或理解代码。

* 如果存在约束，应对类型参数应用什么约束（请参见 [类型参数的约束（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/d5x73970.aspx)）。

一条有用的规则是，应用尽可能最多的约束，但仍使您能够处理必须处理的类型。例如，如果您知道您的泛型类仅用于引用类型，则应用类约束。这可以防止您的类被意外地用于值类型，并允许您对 T 使用 **as** 运算符以及检查空值。

* 是否将泛型行为分解为基类和子类。

由于泛型类可以作为基类使用，此处适用的设计注意事项与非泛型类相同。请参见本主题后面有关从泛型基类继承的规则。

* 是否实现一个或多个泛型接口。

例如，如果您设计一个类，该类将用于创建基于泛型的集合中的项，则可能必须实现一个接口，如 [IComparable<T>](https://msdn.microsoft.com/zh-cn/library/4d7sx9hd.aspx)，其中 T 是您的类的类型。

有关简单泛型类的示例，请参见[泛型介绍（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/0x6a29h6.aspx)。

类型参数和约束的规则对于泛型类行为有几方面的含义，特别是关于继承和成员可访问性。您应当先理解一些术语，然后再继续进行。对于泛型类 Node<T>，客户端代码可通过指定类型参数来引用该类，以便创建封闭式构造类型 (Node<int>)。或者可以让类型参数处于未指定状态（例如在指定泛型基类时）以创建开放式构造类型 (Node<T>)。泛型类可以从具体的、封闭式构造或开放式构造基类继承：

class BaseNode { }

class BaseNodeGeneric<T> { }

// concrete type

class NodeConcrete<T> : BaseNode { }

//closed constructed type

class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type

class NodeOpen<T> : BaseNodeGeneric<T> { }

非泛型类（换句话说，即具体类）可以从封闭式构造基类继承，但无法从开放式构造类或类型参数继承，因为在运行时客户端代码无法提供实例化基类所需的类型参数。

//No error

class Node1 : BaseNodeGeneric<int> { }

//Generates an error

//class Node2 : BaseNodeGeneric<T> {}

//Generates an error

//class Node3 : T {}

从开放式构造类型继承的泛型类必须为任何未被继承类共享的基类类型参数提供类型变量，如以下代码所示：

class BaseNodeMultiple<T, U> { }

//No error

class Node4<T> : BaseNodeMultiple<T, int> { }

//No error

class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error

//class Node6<T> : BaseNodeMultiple<T, U> {}

从开放式构造类型继承的泛型类必须指定约束，这些约束是基类型约束的超集或暗示基类型约束：

class NodeItem<T> where T : System.IComparable<T>, new() { }

class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }

泛型类型可以使用多个类型参数和约束，如下所示：

class SuperKeyType<K, V, U>

where U : System.IComparable<U>

where V : new()

{ }

开放式构造类型和封闭式构造类型可以用作方法参数：

void Swap<T>(List<T> list1, List<T> list2)

{

//code to swap items

}

void Swap(List<int> list1, List<int> list2)

{

//code to swap items

}

如果某个泛型类实现了接口，则可以将该类的所有实例强制转换为该接口。

泛型类是不变的。也就是说，如果输入参数指定 List<BaseClass>，则当您尝试提供 List<DerivedClass> 时，将会发生编译时错误。

## 实例

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace csharp\_basic\_generic\_class  {  public class MyStack<T> //自己实现一个模拟栈的类  {  private T[] \_stack;  private int \_size;  private int \_stackPoint;  public MyStack(int size)  {  \_stack = new T[size];  \_size = size;  \_stackPoint = 0;  }  public void Push(T t)  {  if(\_stackPoint ==\_size)  {  Console.WriteLine("栈已经满了,不能添加新数据");  }  else  {  \_stack[\_stackPoint] = t;  \_stackPoint++;  }    }  public void show()  {  for(int i=\_size-1;i>=0;i--)//栈需要后进先出  {  Console.WriteLine(\_stack[i]);  }  }  }  internal class Program  {  static void Main(string[] args)  {  //MyStack<int> nums = new MyStack<int>(3);  //for(int i=0; i<4;i++)//这里设置越界.最后一个无法添加  //{  // nums.Push(i);  //}  //nums.show();  MyStack<string> names = new MyStack<string>(3);  for (int i = 0; i < 3; i++)  {  names.Push($"names{i}");  }  names.show();  }  }  } |

**请参阅**

[System.Collections.Generic](https://msdn.microsoft.com/zh-cn/library/system.collections.generic.aspx)

[C# 编程指南](https://msdn.microsoft.com/zh-cn/library/67ef8sbd.aspx)

[泛型（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/512aeb7t.aspx)

[Saving the State of Enumerators（保存枚举器的状态）](http://go.microsoft.com/fwlink/?LinkId=112390)

[An Inheritance Puzzle, Part One（一个继承难题，第一部分）](http://go.microsoft.com/fwlink/?LinkId=112380)