C#泛型参数化了类型，把类型作为参数抽象出来，从而使我们在实际的运用当中能够更好的实现代码的重复利用，同时它提供了更强的类型安全，更高的效率，不过在约束方面，它只支持显示的约束，这样在灵活性方面就显得不是那么好了。我觉得它之所以能够提供更高的效率是因为泛型在实例化的时候采用了"on-demand"的模式,即按需实例化,发生在JIT(Just In Time)编译时。

    下面来看如何定义一个C#泛型类，很简单，你只需要意识到一点,在这里,类型已经被参数化了:

1. using System;
2. using System.Collections.Generic;
3. using System.Text;
5. namespace GenericTest
6. {
7. class Program
8. {
9. static void Main(string[] args)
10. {
11. //使用string,int来实例化Test< T,S>类
12. Test< string, int> t = new Test< string, int>("SHY520",22);
14. //调用泛型类中的方法
15. t.SetValue();
16. }
17. }
19. /\*\*//// < summary>
20. /// 定义一个泛型类，该类有两个类型参数，分别是T,S
21. /// http://pw.cnblogs.com
22. /// < /summary>
23. /// < typeparam name="T">类型参数< /typeparam>
24. /// < typeparam name="S">类型参数< /typeparam>
25. public class Test< T,S>
26. {
27. //泛型类的类型参数可用于类成员
28. private T name;
29. private S age;
31. public Test(T Name,S Age)
32. {
33. this.name = Name;
34. this.age = Age;
35. }
37. public void SetValue()
38. {
39. Console.WriteLine(name.ToString());
40. Console.WriteLine(age.ToString());
41. }
42. }
43. }

    上面的例子不是很恰当，目的是让初学泛型的你了解一下泛型的定义及实例化方法，如上，我们定义了一个泛型类，那么如何实现C#泛型类的继承呢？这里需要满足下面两点中的任何一点即可：

    1、泛型类继承中，父类的类型参数已被实例化，这种情况下子类不一定必须是泛型类；

    2、父类的类型参数没有被实例化，但来源于子类，也就是说父类和子类都是泛型类，并且二者有相同的类型参数；

1. //如果这样写的话，显然会报找不到类型T,S的错误
2. public class TestChild : Test< T, S> { }
4. //正确的写法应该是
5. **public class TestChild : Test< string, int>{ }**
6. **public class TestChild< T, S> : Test< T, S> { }**
7. **public class TestChild< T, S> : Test< String, int> { }**

    接着我们来看看泛型接口，其创建以及继承规则和上面说的泛型类是一样的，看下面的代码：

1. public interface IList< T>
2. {
3. T[] GetElements();
4. }
5. public interface IDictionary< K,V>
6. {
7. void Add(K key, V value);
8. }
10. // 泛型接口的类型参数要么已实例化
11. // 要么来源于实现类声明的类型参数
12. class List< T> : IList< T>, IDictionary< int, T>
13. {
14. public T[] GetElements() { return null; }
15. public void Add(int index, T value)
16. {}
17. }

    在来看一下C#泛型委托，首先我们定义一个类型参数为T的委托，然后在类中利用委托调用方法：

1. using System;
2. using System.Collections.Generic;
3. using System.Text;
5. namespace GenericTest
6. {
7. //定义一个委托，类型参数为T，返回值类型T
8. //泛型委托支持在返回值和参数上应用类型参数
9. delegate string GenericDelete< T>(T value);
11. class test
12. {
13. static string F(int i) { return "SHY520"; }
14. static string G(string s) { return "SHY520"; }
16. static void Main(string[] args)
17. {
18. GenericDelete< string> G1 = G;
19. GenericDelete< int> G2 = new GenericDelete< int>(F);
20. }
21. }
22. }

    我们再来看C#泛型方法，C#的泛型机制只支持在方法申明上包含类型参数，也即是泛型方法。特别注意的是，泛型不支持在除了方法以外的其他类/接口成员上使用类型参数，但这些成员可以被包含在泛型类型中，并且可以使用泛型类型的类型参数。还有一点需要说的就是，泛型方法可以在泛型类型中，也可以存在于非泛型类型中。下面我们分别看一下泛型类型的申明，调用，重载和覆盖。

1. using System;
2. using System.Collections.Generic;
3. using System.Text;
5. namespace GenericTest
6. {
7. class GenericClass
8. {
9. //申明一个泛型方法
10. public T getvalue< T>(T t)
11. {
12. return t;
13. }
15. //调用泛型方法
16. //注意：在调用泛型方法时，对泛型方法的类型参数实例化
17. public int useMethod()
18. {
19. return this.getvalue< int>(10);
20. }
22. //重载getvalue方法
23. public int getvalue(int i)
24. {
25. return i;
26. }
27. }
29. //下面演示覆盖
30. //要注意的是，泛型方法被覆盖时，约束被默认继承，不需要重新指定约束关系
31. abstract class Parent
32. {
33. public abstract K TEST< K, V>(K k, V v) where K : V;
34. }
36. class Child : Parent
37. {
38. public override T TEST< T, S>(T t, S s)
39. {
40. return t;
41. }
42. }
43. }

    最后我们来看一下C#泛型中的约束：

    C#中的泛型只支持显示的约束，因为这样才能保证C#所要求的类型安全，但显示的约束并非时必须的，如果不加约束，泛型类型参数将只能访问System.Object类型中的公有方法。“显式约束”由where子句表达，可以指定“基类约束”，“接口约束”，“构造器约束”，“值类型/引用类型约束”共四种约束。下面的例子来源于李建忠老师的讲座PPT。

    1、基类约束：

1. class A { public void F1() {} }
2. class B { public void F2() {} }
3. class C< S,T>
4. where S: A // S继承自A
5. where T: B // T继承自B
6. {
7. // 可以在类型为S的变量上调用F1，
8. // 可以在类型为T的变量上调用F2
9. }

    2、接口约束

1. interface IPrintable { void Print(); }
2. interface IComparable< T> { int CompareTo(T v);}
3. interface IKeyProvider< T> { T GetKey(); }
4. class Dictionary< K,V>
5. where K: IComparable< K>
6. where V: IPrintable, IKeyProvider< K>
7. {
8. // 可以在类型为K的变量上调用CompareTo，
9. // 可以在类型为V的变量上调用Print和GetKey
10. }

    3、构造器约束

1. class A { public A() { } }
2. class B { public B(int i) { } }
3. class C< T>
4. where T : new()
5. {
6. //可以在其中使用T t=new T();
7. }
8. C< A> c=new C< A>(); //可以，A有无参构造器
9. C< B> c=new C< B>(); //错误，B没有无参构造器

    4、值/引用类型约束

1. public struct A { }
2. public class B { }
3. class C< T>
4. where T : struct
5. {
6. // T在这里面是一个值类型
7. }
8. C< A> c=new C< A>(); //可以，A是一个值类型
9. C< B> c=new C< B>(); //错误，B是一个引用类型