**[.net泛型理解](http://www.cnblogs.com/ysyn/p/3392075.html)**

**泛型简介：**

　　泛型（Generic Type）是.NET Framework2.0最强大的功能之一。泛型的主要思想是将算法与数据结构完全分离开，使得一次定义的算法能作用于多种数据结构，从而实现高度可重用的开发。通过泛型可以定义类型安全的数据结构，而没有必要使用实际的数据类型，这将显著提高系统性能并得到高质量的代码（因为可以重用数据处理算法，没有必要复制类型特定的代码）。

**泛型工作原理：**

　　通过泛型可以定义类型安全并且对性能或工作效率无损害的类。表面上，C#泛型的语法和C++模板类似，但编译器在实现和支持他们的方式存在重要的差异。与C++模板相比，C#泛型可以提供增强的安全性，但在功能方面也受到某种程度的限制。在一些C++编译器中，在通过特定类型使用模板类之前，编译器甚至不会编译模板代码，当确实指定了类型时，编译器会以内联方式插入代码，并且将每个出现一般类型参数的地方替换为指定的类型。此外，每当使用特定类型时，编译器都会插入特定于该类型的代码，而不管是否已经在应用程序的其他位置为模板类指定了该类型，其中C++链接器负者解决该问题，但并不总是有效，而这也可能会导致代码膨胀，从而增加加载时间和内存足迹。

　　在.net Framewrok 2.0 中，泛型在IL和CLR本身中具有本机支持。在编译泛型c#代码时，像其他任何类型一样，首先编译器会将其编译为IL，但是，IL只包含实际特定类型的参数或占位符，并有专用的IL指令支持泛型操作。泛型代码的元数据中包含泛型信息。真正的泛型实例化工作是以“on-demand”的方式，发生在JIT编译时。当进行JIT编译时，JIT编译器用指定的类型实参来替换泛型IL代码元数据中的T，进行泛型类型的实例化。这会像JIT编译器提供类型特定的IL元数据定义，就好像从未涉及到泛型一样。JIT编译器可以确保参数正确性，实施类型安全检查，甚至执行类型特定的IntelliSense。

　　当.net将泛型IL代码编译为本机代码时，所产生的本机代码取决于指定的类型。JIT编译器跟踪已经生成特定类型的IL代码，如果本机指定的是值类型，则JIT编译器将泛型类型参数替换为指定的值类型，并且将其编译为本机代码。如果JIT编译器用已经编译为本机代码的值类型编译泛型IL代码，则只返回对IL代码的引用。因为JIT编译器在以后的所有场合中都将使用相同的值类型特定的IL代码，所以不会存在代码膨胀问题。如果本机指定的是引用类型，则JIT编译器将泛型IL嗲没种的泛型参数替换为object，并将其编译为本机代码。在以后的任何针对引用类型而不是泛型类型参数的请求中，JIT编译器只会重新使用实际代码，使用该代码，实例仍然按照它们离开托管堆的大小分配空间，并且不会存在强制类型转换。

**泛型实现：**

C#泛型支持包括类、结构、接口、委托四种泛型类型，以及其方法成员。此次，只以泛型类做简单演示：

[复制代码](javascript:void(0);)

class Program

{

static void Main(string[] args)

{

int obj = 1;

Test<int> testInt = new Test<int>(obj);

Console.WriteLine(testInt.obj);//1

string obj1 = "hello";

Test<string> testString = new Test<string>(obj1);

Console.WriteLine(testString.obj);//hello

Console.Read();

}

}

class Test<T>

{

public T obj;

public Test(T obj)

{

this.obj = obj;

}

}

[复制代码](javascript:void(0);)

**泛型约束：**

　　在指定一个类型参数时，可以指定类型参数必须满足的约束条件。这里通过指定类型参数时使用where子句来实现的。泛型约束有：基类约束、接口约束、构造函数约束、引用类型和值类型约束。下面简单一一介绍：

基类约束：

使用基类约束，可以指定某个类型实参必须继承的基类。  
基类约束有两个功能：  
（1）它允许在泛型类中使用由约束指定的基类所定义的成员。例如，可以调用基类的方法或者使用基类的属性。如果没有基类约束，编译器就无法知道某个类型实参拥有哪些成员。通过提供基类约束，编译器将知道所有的类型实参都拥有由指定基类所定义的成员。  
（2）确保类型实参支持指定的基类类型参数。这意味着对于任意给定的基类约束，类型实参必须要么是基类本身，要么是派生于该基类的类，如果试图使用没有继承指定基类的类型实参，就会导致编译错误。  
基类约束使用下面形式的where子句：where T:base-class-name  
T是类型参数的名称，base-class-name是基类的名称，这里只能指定一个基类。

[复制代码](javascript:void(0);)

class A

{

public void Func1()

{ }

}

class B

{

public void Func2()

{ }

}

class C<S, T>

where S : A

where T : B

{

public C(S s,T t)

{

//S的变量可以调用Func1方法

s.Func1();

//T的变量可以调用Func2方法

t.Func2();

}

}

[复制代码](javascript:void(0);)

接口约束：

接口约束用于指定某个类型参数必须应用的接口。接口的两个主要功能和基类约束完全一样。基本形式 where T:interface-name  
interface-name是接口的名称，可以通过使用由逗号分割的列表来同时指定多个接口。如果某个约束同时包含基类和接口，则先指定基类列表，再指定接口列表。

[复制代码](javascript:void(0);)

interface IA<T>

{

T Func1();

}

interface IB

{

void Func2();

}

interface IC<T>

{

T Func3();

}

class MyClass<T, V>

where T : IA<T>

where V : IB, IC<V>

{

public MyClass(T t,V v)

{

//T的对象可以调用Func1

t.Func1();

//V的对象可以调用Func2和Func3

v.Func2();

v.Func3();

}

}

[复制代码](javascript:void(0);)

构造器约束：  
new()构造函数约束允许开发人员实例化一个泛型类型的对象。  
一般情况下，我们无法创建一个泛型类型参数的实例。然而，new()约束改变了这种情况，它**要求类型参数必须提供一个无参数的构造函数**。在使用new()约束时，可以通过调用该无参构造函数来创建对象。基本形式： where T : new()  
使用new()约束时应注意两点：  
（1）它可以与其他约束一起使用，但是必须位于约束列表的末端。  
（2）new()仅允许开发人员使用无参构造函数来构造一个对象，即使同时存在其他的构造函数。换句话说，不允许给类型参数的构造函数传递实参。

[复制代码](javascript:void(0);)

class A

{

public A()

{ }

}

class B

{

public B()

{ }

public B(int i)

{ }

}

class C<T> where T : new()

{

T t;

public C()

{

t = new T();

}

}

class D

{

public void Func()

{

C<A> c = new C<A>();

C<B> d = new C<B>();

}

}

[复制代码](javascript:void(0);)

值类型和引用类型约束：  
如果引用类型和值类型之间的差别对于泛型代码非常重要，那么这些约束就非常有用。 基本形式： where T : class where T : struct 若同时存在其他约束的情况下，class或struct必须位于列表的开头。另外可以通过 使用约束来建立两个类型参数之间的关系 例如 class GenericClass2<T, V> where V:T{} -------- 要求V必须继承于T，这种称为裸类型约束（naked type constraint）。

[复制代码](javascript:void(0);)

public struct A { }

public class B { }

public class C<T> where T : struct

{

}

C<A> c1 = new C<A>();

C<B> c2 = new C<B>();//error

[复制代码](javascript:void(0);)

**泛型的优点：**  
　　泛型使代码可以重用，类型和内部数据可以在不导致代码膨胀的情况下进行更改，而不管是值类型还是引用类型。可以一次性地开发、测试和部署代码，通过任何类型（包括将来的类型）来重用它，并且全部具有编译器支持和类型安全。因为泛型代码不会强行对值类型进行装箱和取消装箱，或者对引用类型进行向下强制类型转换，所以性能得到显著提高。对于值类型，性能通常会提高200%；对于引用类型，在访问该类型时，也可以达到预期性。

简单来说：使用泛型，可以极大地提高代码的重用度，同时还可以获得强类型的支持，避免了隐式的装箱、拆箱，在一定程度上提升了应用程序的性能。

再简单点：可重用、类型安全、高效率。