## 设计一个列表数据的树形适配器

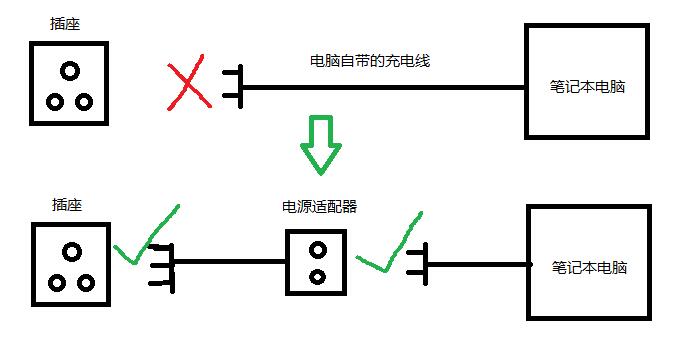
1. 目的

开发中经常会用到很多树形结构的数据，但是他们的存储形式往往是列表形式的，比如数据库表里存储应用程序的菜单信息，查到的原始数据（DataTable）是列表形式的，而我们操作需要基于树形结构。为了方便操作，下面设计一个将列表数据结构适配到树形结构的适配器。

1. 适配器模式

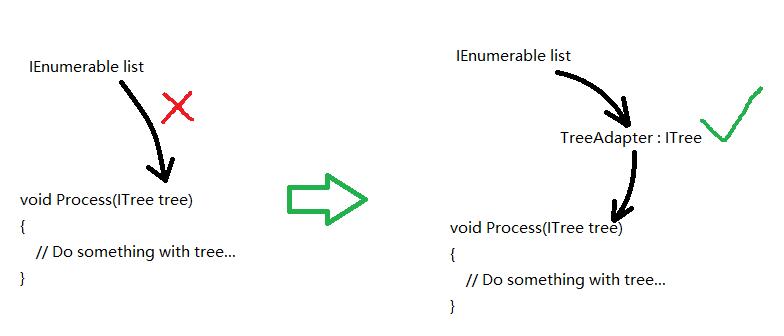
适配器模式是24中经典设计模式中的一种，它的核心动机是**将一个不符合现有接口的对象经过转换（适配），使其符合现有的接口**。分别用现实生活的例子和代码大概地解释这个过程：

1. 电源适配器的例子：



上面的图应该很容易理解（画画水平有待提高，只会画圆和方框），电源适配器的作用在这里起到一个转接的作用，或者是中间人的作用，对于插座来说，它根本不知道笔记本电脑的存在，对笔记本电脑来说，它也不知道插座的存在，它们俩都只知道适配器。

1. 代码的例子



上面的代码其实就是接下来要解决的：现有的代码Process接收一个符合树形结构（实现ITree接口）的对象，但是待处理的数据却是一个列表（实现IEnumerable接口），如何实现一个适配器，将列表数据转换成一个符合树形结构的对象？

1. 定义数据结构

在上面的代码图的右边部分可以看到有几个比较重要的对象：

1. IEnumerable接口：作为一个列表数据的接口，一般在现有的代码库中已经存在，所以不需要我们定义；
2. ITree接口：这个接口在现有的代码中也往往已经存在，至少对于上面的例子来说是这样的，否则还要我们适配干嘛？但是我在这里不仅仅只是实现适配器模式，还要实现一个树形的数据结构，所以我需要定义一个ITree接口。我在这里定义了一个最简单的：

public interface ITree<T>

{

IEnumerable<ITreeNode<T>> Nodes { get; }

}

public interface ITreeNode<T>

{

T Data { get; }

IEnumerable<ITreeNode<T>> Nodes { get; }

}

这里最终定义了两个接口：ITree<T> 和ITreeNode<T>，前者代表树本身，后者代表树节点，树本身由一个节点集合构成，节点由数据和它的子节点构成，这是树形结构最基本的要素了。除此之外为了类型安全我支持泛型。

1. TreeAdapter：这是主角，用于将列表结构的数据适配成树形结构，所以适配器的一个重要特征是它需要实现目标接口，也就是ITree，并且他要接收一个列表数据，所以一个适配器的基本骨架式这样的：

public class TreeAdapter<T> : ITree<T>

{

private IEnumerable m\_list;

public TreeAdapter(IEnumerable list) { m\_list = list; }

public IEnumerable<ITreeNode<T>> Nodes { get; }

}

但是这还不够，上面的适配器结构不是类型安全的，问题在于接收的列表数据，它应该实现IEnumerable<T>，所以修改过后的适配器结构为：

public class TreeAdapter<TFrom, TTo> : ITree<TTo>

{

private IEnumerable<TFrom> m\_list;

public TreeAdapter(IEnumerable<TFrom> list) { m\_list = list; }

public IEnumerable<ITreeNode<TTo>> Nodes { get; }

}

现在这个适配器的作用是：将一个包含类型TFrom数据的列表适配到包含类型TTo数据的树。

1. 实现数据结构

现在需要实现TreeAdapter的适配逻辑，TreeAdapter本质上就是一颗树，我们基于下面这样的一个逻辑：

1. 针对树本身，在原始的列表里筛选并创建出树的根节点；
2. 针对每个节点，再次在原始的列表中筛选出可以作为当前结点子节点的数据，创建子节点；

上面的两个步骤有一个共同的操作：创建节点，所以我把这部分的逻辑单独抽出来：

public class TreeAdapter<TFrom, TTo> : ITree<TTo>

{

//...

private ITreeNode<TTo> GenerateNode(TFrom item)

{

return new DefaultTreeNode<TTo>()

{

Data = Transform(item), /\* 将原始的数据类型转型为目标数据类型 \*/

Nodes = GenerateChildNodes(item), // 针对第二点的实现

};

}

//...

}

我用/\*\*/注释的部分代表我当前的TreeAdapter是未知无法实现的，所以到时候需要定义对应的抽象方法（下同）。DefaultTreeNode 在这里是一个ITreeNode的默认实现，不对外公开。

第一点的实现代码大致是这样的：

public class TreeAdapter<TFrom, TTo> : ITree<TTo>

{

//...

public IEnumerable<ITreeNode<TTo>> Nodes

{

get

{

foreach (var item in m\_list)

{

if (IsRootNode(item)) /\* 判断是否为根节点 \*/

{

yield return GenerateNode(item); }

}

}

}

//...

}

这里用yield关键词是为了支持根节点的延迟加载。

第二点的实现代码大致是这样的：

public class TreeAdapter<TFrom, TTo> : ITree<TTo>

{

//...

private IEnumerable<ITreeNode<TTo>> GenerateChildNodes(TFrom parent)

{

foreach (var item in m\_list)

{

if (IsRelation(parent, item) /\* 判断两个根节点是否有父子关系 \*/)

{

yield return GenerateNode(item);

}

}

}

//...

}

最终TreeAdapter的骨架就变成了下面这样：

public abstract class TreeAdapter<TFrom, TTo> : ITree<TTo>

{

private IEnumerable<TFrom> m\_list;

public TreeAdapter(IEnumerable<TFrom> list) { m\_list = list; }

public IEnumerable<ITreeNode<TTo>> Nodes{ get { /\*第一点实现\*/ } }

private IEnumerable<ITreeNode<TTo>> GenerateChildNodes(TFrom parent)

{ /\*第二点实现\*/}

private ITreeNode<TTo> GenerateNode(TFrom item) { /\*产生节点\*/ }

// 需要子类实现的抽象方法

public abstract bool IsRootNode(TFrom item);

public abstract bool IsRelation(TFrom parent, TFrom child);

public abstract TTo Transform(TFrom from);

}

1. 扩充

到现在一个最基本的适配器已经实现了，并且支持节点的延迟加载。客户端使用时，需要继承TreeAdapter，并且实现它的三个抽象方法：IsRootNode，IsRelation，Transform，分别用于判断是否是根节点，是否存在父子关系以及将原始的列表数据转换成树形数据类型。为了方便客户端的使用，这里根据这个使用方法设计几个便捷类：

1. 不需要进行数据类型转换的 NoTransformTreeAdapter，代码如下：

public abstract class NoTransformTreeAdapter<T> : TreeAdapter<T, T>

{

public NoTransformTreeAdapter(IEnumerable<T> list) : base(list) { }

public override T Transform(T from) { return from; }

}

NoTransformTreeAdapter 实现基类的Transform抽象，不进行转换直接返回原始数据；

1. 用于适配DataTable的 DataTableTreeAdapter， 代码如下：

public abstract class DataTableTreeAdapter<T> : TreeAdapter<DataRow, T>

{

public DataTableTreeAdapter(DataTable data) : base(data.AsEnumerable()) { }

}

1. 用于适配DataTable并且不进行数据类型转换的NoTransformDataTableTreeAdapter：

public abstract class NoTransformDataTableTreeAdapter : NoTransformTreeAdapter<DataRow>

{

public NoTransformDataTableTreeAdapter(DataTable data) : base(data.AsEnumerable()) { }

}

可以看到它直接继承自NoTransformTreeAdapter

1. 延迟树

这里要说的延迟树和上面的节点延迟加载里的延迟并以完全一样，虽然他们的作用都是实现节点的延迟加载，但是实现方式是不一样的，这里的细微区别在于，前面的实现是用于构建树的所有数据都是一次性获取的，这里要实现的延迟是：第一次只取用于构建根节点的数据，以后在需要构建子节点的时候再去获取相应数据（比如从数据库，网络）。这个特性在前面的基础上很容易实现：

public abstract class LazyTreeAdapter<TFrom, TTo> : TreeAdapter<TFrom, TTo>

{

public LazyTreeAdapter(IEnumerable<TFrom> list) : base(list){ }

public override sealed bool IsRootNode(TFrom item)

{

return true;

}

public override sealed bool IsRelation(TFrom parent, TFrom child)

{

throw new NotSupportedException("懒加载的树形适配器不支持该操作");

}

protected override sealed IEnumerable<ITreeNode<TTo>> GenerateChildNodes(TFrom parent)

{

foreach (var child in FetchChildData(parent))

{

yield return base.GenerateNode(child);

}

}

public abstract IEnumerable<TFrom> FetchChildData(TFrom parent);

}

这里需要做两件事：

1. 实现基类的IsRootNode和IsRelation抽象方法，前者返货true，代表传入的数据嗾是根节点；后者不支持操作，为了保证LazyTreeAdapter懒加载契约，我故意将这两个方法密封掉；
2. 将基类TreeAdapter的GenerateChildNodes方法置成受保护的虚方法，然后重写创建子节点的逻辑：这里多了一个FetchChildData的抽象方法，由子类实现获取子节点数据的具体逻辑；

扩充：在这个懒加载适配器的基础上还可以像之前那样提供几个相同作用的便捷类，实际上这两种情况是不同的继承体系了，实际设计时应该考虑将懒加载相关的类放到 .Lazy子命名空间下更合适。

1. 优化
2. 节点的懒加载

这其实是个有点强迫症的优化点。简单来说就是把ITreeNode看成是原始数据的包装器，在创建ITreeNode对象的时候实际并不创建它的Data和Nodes属性，而在实际访问的时候才创建。由于是个原始数据的包装器，那么它的结构应该是这样的：

internal class LazyTreeNode : ITreeNode<TTo>

{

private TFrom m\_data;

public LazyTreeNode(TFrom data) { m\_data = data; }

public TTo Data { get { /\*如何实现？\*/ } }

public IEnumerable<ITreeNode<TTo>> Nodes { get { /\*如何实现？\*/ } }

}

如何实现这里的Data和Nodes代码？发现如果仅仅是这样，并不能实现，因为相关的实现逻辑都在TreeAdapter里，为了使用TreeAdapter里的逻辑，我把LazyTreeNode作为TreeAdapter的内部类来（内部类可以访问外部类的私有数据），最后实现的代码如下：

public abstract class TreeAdapter<TFrom, TTo> : ITree<TTo>

{

//...

internal class LazyTreeNode : ITreeNode<TTo>

{

private TreeAdapter<TFrom, TTo> m\_tree;

private TFrom m\_data;

public LazyTreeNode(TFrom data, TreeAdapter<TFrom, TTo> tree)

{

m\_data = data;

m\_tree = tree;

}

public TTo Data { get { return m\_tree.Transform(m\_data); } }

public IEnumerable<ITreeNode<TTo>> Nodes

{

get { return m\_tree.GenerateChildNodes(m\_data); }

}

}

//...

}

这样，节点的Data和Nodes属性就都是延迟加载的了，当然要修改一下TreeAdapter的GenerateNode方法，让他返回LazyTreeNode对象,而不是原先的DefaultTreeNode：

public abstract class TreeAdapter<TFrom, TTo> : ITree<TTo>

{

//...

private ITreeNode<TTo> GenerateNode(TFrom item)

{

return new LazyTreeNode(item, this);

}

//...

}

1. 筛选逻辑优化

之前的实现一个最大的缺点就是在创建子节点的时候需要重新遍历所有的数据，当数据量大的时候可能会存在性能问题。可以实现这样一个临时的辅助数据结构：

1. 一个先进先出的队列；
2. 支持删除式的枚举，将队首的节点删除或者将新节点添加到尾部（.net自带的Queue不支持此操作）；
3. 枚举结束的标志是当且仅当每个元素在当前一轮的枚举中仅访问了一次；

用这个辅助的队列对上面的筛选逻辑进行优化：

public abstract class TreeAdapter<TFrom, TTo> : ITree<TTo>

{

//...

public IEnumerable<ITreeNode<TTo>> Nodes

{

get

{

foreach (var item in m\_list)

{

if (IsRootNode(item))

{

yield return GenerateNode(item);

}

else

{

m\_queue.Enqueue(item);

}

}

m\_list = m\_queue;

}

}

protected virtual IEnumerable<ITreeNode<TTo>> GenerateChildNodes(TFrom parent)

{

foreach (var item in m\_list)

{

if (IsRelation(parent, item))

{

yield return GenerateNode(item);

}

else

{

m\_queue.Enqueue(item);

}

}

}

//...

}

在筛选根节点的时候将非根节点数据添加到队列中，接下来后面筛选子节点都基于队列进行操作，每筛选一轮就将符合条件的数据添加到队尾，符合条件的就从队首删除。