Introduce the MapReduce pattern in CAP

MapReduce is one of “those” buzz words that is going around at the moment. Mostly in part due to Google using it so successfully for their distributed indexing algorithms.

So what is MapReduce?

According to Wikipedia [http://en.wikipedia.org/wiki/MapReduce]

So MapReduce is an amalgamation of two higher order functions taken from functional programming.

Map and Reduce :. MapReduce.

Google took the concepts of Map and Reduce and designed a distributed computing framework around those two concepts. As it’s almost infinitely horizontally scalable, it lends itself to distributed computing quite easily.

So lets break up MapReduce into its 2 main components.

Map

Map is a higher order function which takes a function and a list, and applies that function to each element of that list and returns the resultant list.

Given the function x = x + 2, and a list of integers 1 -> 3. Map will return the following list

: : / \ / \ 1 : 3 : / \ / \ 2 : 4 : / \ / \ 3 [] 5 []

Returns: 3, 4, 5

Map in C#

static void Main(string[] args){ var testList = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; var mapList = Map<int, int>(x => x + 2, testList);  mapList.ToList<int>().ForEach(i => Console.Write(i + " ")); Console.WriteLine(); Console.ReadKey();} static IEnumerable<TResult> Map<T, TResult>(Func<T,TResult> func, IEnumerable<T> list){ foreach (var i in list) yield return func(i);}

Returns: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Reduce

Reduce takes a list of elements and returns a function that is applied to all of the elements of that list.

Given the function (x, y) => x + y, and a list of elements 1 to 3, Reduce will return an aggregate of the following

: / \ 1 : f(0,1) = 1 / \ \ 2 : f(1,2) = 3 / \ \ 3 [] f(3,3) = 6

Returns: 6

Reduce in C#

static void Main(string[] args){ var testList = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; Console.WriteLine(Reduce<int, int>((x, y) => x + y, testList, 0)); Console.ReadKey();} static T Reduce<T, U>(Func<U, T, T> func, IEnumerable<U> list, T acc){ foreach (var i in list) acc = func(i, acc);  return acc;}

Returns: 55

So how does google apply Map and Reduce to a distributed framework?

Google uses Map and Reduce as higher order abstractions which allow them to hide the parallelization to lower lying functions.

For instance by breaking up a larger list into smaller lists, google can distribute those smaller lists to individual servers to run the map function. Each server can then forward their resultant list to a 2nd set servers running the Reduce function (which may sum up the results of the intermediate list). A final server could then create a list from each of the results from the intermediate servers and then reduce that again giving a final result).

Tier 1: Map(L1), Map(L2), Map(L3), Map(L4), Map(L5)Tier 2: Reduce(R1), Reduce(R2), Reduce(R3), Reduce(R4), Reduce(R5)Tier 3: Reduce(new list {R1, R2, R3, R4, R5}) = N

L1 -> L5 are lists of wordsR1 -> R5 are the intermediate count of instances of a wordFinal Reduce takes a list of all of the intermediate results and then returns the sum

Linq equivalents of Map and Reduce

If you’re lucky enough to have linq then you don’t need to write your own map and reduce functions. C# 3.5 and Linq already has it albeit under different names.

Map = Select | Enumerable.Range(1, 10).Select(x => x + 2);

Reduce = Aggregate | Enumerable.Range(1, 10).Aggregate(0, (acc, x) => acc + x);

Filter = Where | Enumerable.Range(1, 10).Where(x => x % 2 == 0);

Collection Pipeline

Collection pipelines are a programming pattern where you organize some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next. (Common operations are filter, map, and reduce.) This pattern is common in functional programming, and also in object-oriented languages which have lambdas. This article describes the pattern with several examples of how to form pipelines, both to introduce the pattern to those unfamiliar with it, and to help people understand the core concepts so they can more easily take ideas from one language to another.

functional programming

Lambda calculus

"Lambda expression" redirects here. For its use in programming languages, see Anonymous function.

Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution. It was first introduced by mathematician Alonzo Church in the 1930s as part of an investigation into the foundations of mathematics. Lambda calculus is a universal model of computation equivalent to a Turing machine (Church-Turing thesis, 1937[1]). Its namesake, Greek letter lambda (λ), is used in lambda terms (also called lambda expressions) to denote binding a variable in a function.

Lambda calculus may be typed and untyped. In typed lambda calculus functions can be applied only if they are capable of accepting the given input's "type" of data.

Lambda calculus has applications in many different areas in mathematics, philosophy,[2] linguistics,[3][4] and computer science.[5] Lambda calculus has played an important role in the development of the theory of programming languages. Functional programming languages implement the lambda calculus.

Expression Tree

An expression tree provides a method of translating executable code into data. This can be very valuable if you want to modify or transform code before executing it. In particular, it can be useful if you want to transform C# code such as a LINQ query expression into code that operates on another process, such as a SQL database.

But I’m getting ahead of myself. By the end of this post you will find it easy to understand why it is helpful to translate code into data. First I need to provide a little background. Let’s start by seeing the relatively simple syntax for creating an expression tree.

The Syntax for an Expression Tree

Consider the following very simple lambda expression:

Func<int, int, int> function = (a,b) => a + b;

This statement consists of three sections.

1. A declaration: Func<int, int, int> function

2. An equals operator: =

3. A lambda expression: (a,b) => a + b;

The variable function points at raw executable code that knows how to add two numbers. The lambda expression shown in step three above is a short hand way of writing the following method:

public int function(int a, int b){ return a + b;}

One can call either the method shown above, or the lambda expression like this:

int c = function(3, 5);

After the function is called, the variable c will be set equal to 3 + 5, which is 8.

The delegate type Func shown above in the declaration found in step one is declared for us in the System namespace:

public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);

This code looks complicated, but it is used here to help us declare the variable function, which is set equal to a very simple lambda expression that adds two numbers together. Even if you don’t understand delegates and generic functions, it should still be clear that this is a way of declaring a variable that references executable code. In this case it points at very simple executable code.

Translating Code into Data

In the previous section, you saw how to declare a variable that points at raw executable code. Expression trees are not executable code, they are a form of data structure. So how does one translate the raw code found in an expression into an expression tree? How does one translate code into data?

LINQ provides a simple syntax for translating code into a data structure called an expression tree. The first step is to add a using statement to introduce the Linq.Expressions namespace:

using System.Linq.Expressions;

Now we can create an expression tree:

Expression<Func<int, int, int>> expression = (a,b) => a + b;

The identical lambda expression shown in the previous example is converted into an expression tree declared to be of typeExpression<T>. The identifier expression is not executable code; it is a data structure called an expression tree.

The samples that ship with Visual Studio 2008 include a program called the ExpressionTreeVisualizer. It can be used to visualize an expression tree. In Figure 01 you can see a screen shot of a dialog that displays the expression tree for the simple Expression statement shown above. Notice that the lambda expression is displayed at the top of the dialog, and its constituent parts are displayed below it in aTreeView control.

Figure 01: The ExpressionTreeVisualizer sample that ships with the VS 2008 C# Samples creates a symbolic rendering of the expression tree.

CAP

The follow for contest parameter process:

Catchup Batch catch the contest parameter need to be process

Generate the contest request

Contest Catchup Service Process the contest parameter

Add/update the parameter result

Interface:

public interface IContestParameterProcessor

{

long ParameterID { get; }

bool ProcessParameter(ContestTransaction contestTransaction, Int64 parameterID);

IParameter Parameter { get; }

long BonusCreditTarget { get; }

int ProcessingOrder { get; }

}

public interface IMapRule<T, TResult> : IRule<T, IEnumerable<TResult>> {

}

public interface IReduceRule<T, TResult> : IRule {

TResult Execute(IEnumerable<T> list, TResult result);

}

Rule engine

Lexer

Parser

Conext

Engine

Configuration

