# Linq指令执行分析

#### Linq中IEnumerable的结构

Linq在执行聚合操作和ToXxx系统方法之前，一直都是一个数据源和一串指令（下面的讨论都是基于未执行聚合操作和ToXxx系统方法之前）。

大部分linq返回的迭代器都是一个如下的数据结构：

IEnumerable:

source:IEnumerable

指令:针对不同的操作，指令不同，对where来说，就是一个predicate谓词条件，

这个source字段

可以是一个简单的集合，比如List或Array等，

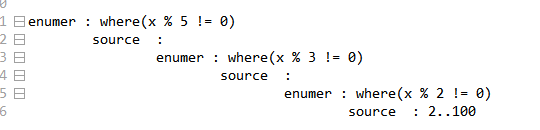
可以不包含指令的IEnumerable，比如Enumerable.Range()方法的结果等，

也可以是另一个包含了source字段和一个指令的IEnumerable实例，

前两种可以认为是基本数据源，后面一种就嵌套了基本数据源的数据源，

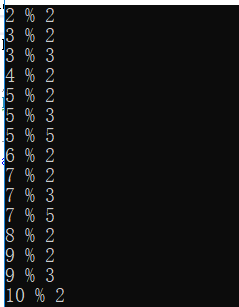
所以一系列连接的linq指令就包含一个基本数据源和一系列的指令，

像这样：



而当执行到聚合方法或ToXxx方法时，IEnumerable就会从最里面开始逐个执行指令，

这里是谓词条件，就会逐个元素判断：



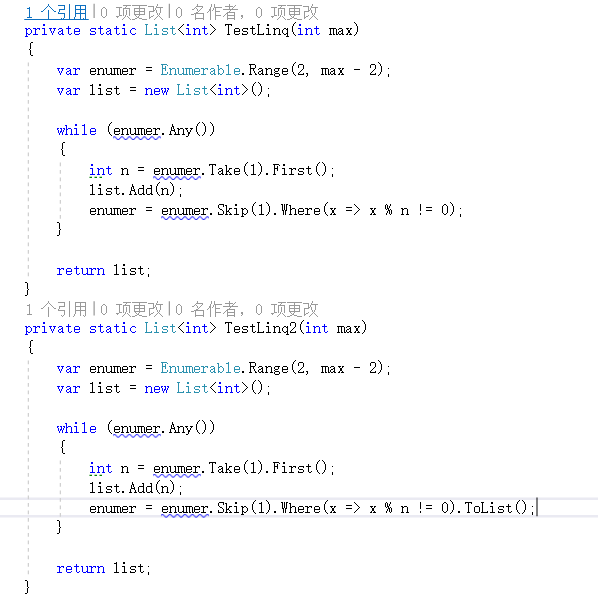
当一个条件不满足条件就跳过这个元素，

当这个条件满足了，就计算下一个条件，

当所有条件都满足了，就是这个一串指令最终结果的元素之一。

把一个基本数据源和一串指令的linq看作是一个链路，而中间使用了ToXxx方法就相当于切断链路。

下面试着分析一下如下两个方法，为什么带ToList会快，



#### Linq指令执行步骤分析

首先第一步

var enumer = Enumerable.Range(2, max - 2);

这个取到的是一个包含2到99的所有数字的迭代器。

第二步，是否满足条件：

enumer.Any()

这时候迭代器还是一个基本数据源，开始迭代，取到2，说明条件成立，退出迭代，进入循环体。

第三步，进入第一次循环体

int n = enumer.Take(1).First();

（这里实际上可以把take(1)这个指令去掉，更快，这个是一个优化的点）

取到数据2，加入到素数列表，然后构造一个新的迭代器：

enumer = enumer.Skip(1).Where(x => x % 2 != 0);

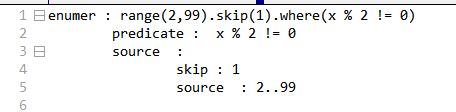
等同于range(2,99).skip(1).where(x%2!=0)

这个迭代器包含一个基本数据源和两个指令，这时候不执行指令，循环体执行结束

第四步，继续判断条件：

enumer.Any()

这个时候的enumer是什么结构呢：



怎么执行呢，还是在基本数据源的基础上迭代，

先取到2，执行skip跳过，

再取到3，判断3%2!=0 ，满足条件，是迭代元素，

enumer.Any()返回true，进入循环体

第五步，第二次进入循环体：

int n = enumer.Take(1).First();

实际上等同于range(2,99).skip(1).where(x%2!=0).take(1).first()

先取到2，执行skip跳过，

再取到3，判断3%2!=0 ，满足条件，是迭代元素，因为take的参数是1，所以迭代结束，n=3，加入到素数列表，

然后构造一个新的迭代器：

enumer = enumer.Skip(1).Where(x => x % 3 != 0);

等同于range(2,99).skip(1).where(x%2!=0).skip(1).where(x%3!=0)

循环体执行结束

第六步，判断enumer.Any()条件，具体步骤如下：

先取到2，执行第一个skip跳过，

再取到3，判断3%2!=0 ，满足条件，是迭代元素，

执行第二个skip跳过，

再取到4，判断4%2==0，不满足条件，忽略元素

再取到5，判断5%2!=0 ，满足条件，继续下一个where指令，判断5%3!=0 ，满足条件，是迭代元素，

于是enumer.Any()的结果是true，进入循环体，

第七步的int n = enumer.Take(1).First();

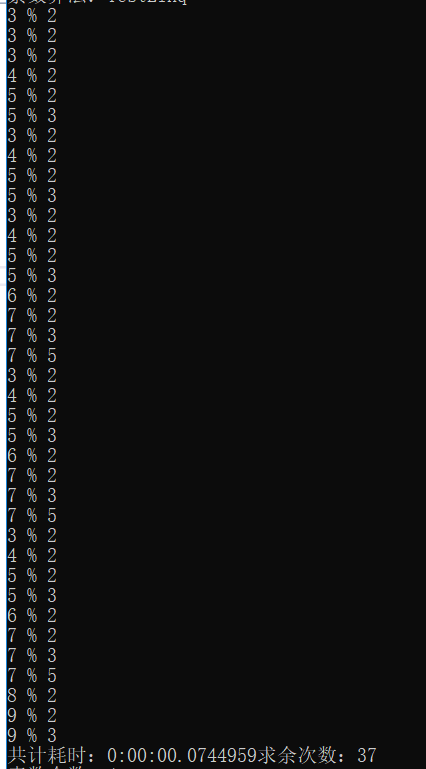
等同于range(2,99).skip(1).where(x%2!=0).skip(1).where(x%3!=0).take(1).first()

为了取出一个第一个数据，重复了第六步的迭代步骤，

后面的每一步都会延长linq指令的长度，执行也就越来越复杂，而无论执行到第几步，迭代步骤都会从3%2开始判断，

所以这个方法中的取余计算呈指数级增长，所以这个算法会很慢。

方法TestLinq中，max=10，所有的取余计算：



#### Linq链路切断分析

但是带上ToList为什么快呢？分析如下：

首先第一步

var enumer = Enumerable.Range(2, max - 2);

这个取到的是一个包含2到99的所有数字的迭代器。

第二步，是否满足条件：

enumer.Any()

这时候迭代器还是一个基本数据源，开始迭代，取到2，说明条件成立，退出迭代，进入循环体。

第三步，进入第一次循环体

int n = enumer.Take(1).First();

取到数据2，加入到素数列表，然后构造一个新的迭代器：

enumer = enumer.Skip(1).Where(x => x % 2 != 0).ToList();

等同于把迭代器range(2,99).skip(1).where(x%2!=0)的结果复制到一个列表list(3,5,7...99)中，

这个列表中已经剔除了所有2 的倍数

原来的迭代器包含一个基本数据源和两个指令，这时候就转换成一个列表，转换成了一个基本数据源，循环体执行结束

第四步，继续判断条件：

enumer.Any()

这个时候的enumer是一个列表集合：

list(3,5,7...99)

怎么执行enumer.Any()呢，就是取list.count>0的值,

list.count是一个属性，步需要迭代集合就能得到值，这个很快，

返回true，进入循环体

第五步，第二次进入循环体：

int n = enumer.Take(1).First();

实际上等同于list(3,5,7...99).take(1).first()

直接取到3，因为take的参数是1，所以迭代结束，n=3，加入到素数列表，

然后构造一个新的迭代器：

enumer = enumer.Skip(1).Where(x => x % 3 != 0).ToList();

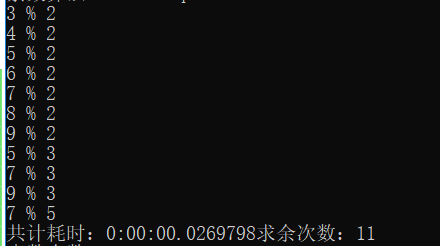
结果是一个新的列表集合，这时候已经剔除了所有2和3的倍数，

等同于list(5,7,11,13,17,19,23,25,29,31,35,37...97)

循环体执行结束

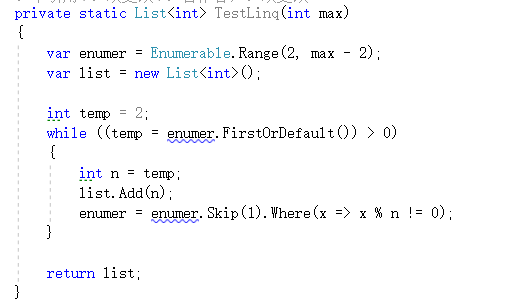
如此继续循环，取数次数大量减少，所有效率比较快。

切断链路后，方法TestLinq2中，max=10，所有的取余计算：

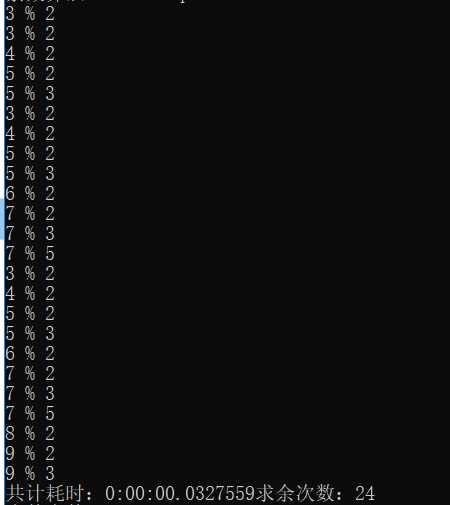


#### 优化

由以上分析可以知道，方法TestLinq中循环条件enumer.Any()和enumer.Take(1).First()，两个指令迭代的步骤几乎完全一样，所以可以去掉一个步骤来优化算法，如下：



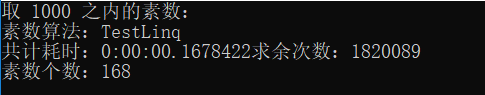
TestLinq优化后，max=10，所有的取余计算：



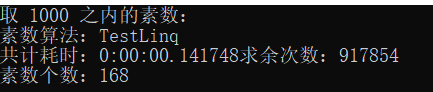
由原来的37次减少到24次，

如果计算规模变大，这个优化效果会更明显，比如max=1000,

优化前：

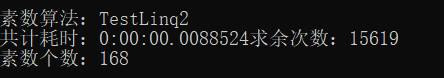


优化后：



 基本上是减少一半的取余操作。

但是比较切断链路的效率还是很高，切断链路取余次数：



#### 总结

经过以上分析得出来结论是，

1. linq在执行聚合方法和ToXxx方法之前，结果一直都是一个基本数据源和一串指令，而每次迭代时都会将所有指令从头到尾执行一遍，所以影响效率。
2. 切断链路可以避免一连串指令的重复执行。
3. 所以在项目中，如果在中指令包含了数据库操作，IO密集操作或CPU密集操作的时候就要小心了，就要及时使用ToXxx方法切断链路。