Парадигмы программирования

Обзор

Что такое парадигма?

- **Парад**и́**гма** (от др.-греч. παράδειγμα, «пример, модель, образец» < παραδείκνυμι — «сравниваю») в философии науки — означает совокупность явных и неявных (и часто не осознаваемых) предпосылок, определяющих научные исследования и признаваемых на данном этапе развития науки
- Система идей, взглядов и понятий, концепций и методов, которыми пользуются для познания чего-либо.
- Скачок в развитии наук, обычно, это смена (сдвиг) парадигмы.

Парадигмы программирования

- **Парадигма программирования** — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию **вычислений** и структурирование **работы**, выполняемой компьютером.
- Основные группы парадигм делятся по тому, что именно описывает программа:
  - **ЧТО** должен вычислить компьютер? («**вычисления**») – **ДЕКЛАРАТИВНАЯ** парадигма.
  - **КАК** он должен это вычислить? («**работа**») – **ИМПЕРАТИВНАЯ** парадигма.

Парадигмы: Императивная vs Декларативная

- Императивное программирование
  - описывается на том или ином уровне детализации **процесс решения задачи и представления результата**;
  - от машины к человеку;
  - Stateful;
  - Довольствуется «тупым» исполнителем программы;
  - Представители - все алгоритмические языки программирования.
- Декларативное программирование
  - задаётся спецификация решения задачи, то есть описывается, **что представляет собой задача и каков ожидаемый результат**;
  - идёт от человека к машине;
  - Stateless;
  - Требует «интеллектуального» исполнителя;
  - Представители – языки ФП, языки запросов, языки разметки и т.п.

Императивная парадигма

- Основные черты:
  - в исходном коде программы записываются инструкции (команды);
  - инструкции должны выполняться последовательно;
  - при выполнении инструкции данные, полученные при выполнении предыдущих инструкций, могут читаться из памяти;
  - данные, полученные при выполнении инструкции, могут записываться в память.
- Главное достоинство:
  - именно так, и только так, работает вычислительная техника.
- Главный недостаток:
  - Переменные –> присваивание –> состояние –> побочные эффекты.
- История:
  - Машинные коды – первый императивный язык
  - Ассемблер
  - Алгоритмические языки – Fortran, Algol, и далее

Развитие императивной парадигмы

- Процедурная парадигма
  - Ввела понятие подпрограмм;
  - Является особенностью архитектуры фон Неймана , где код располагается в общей памяти с произвольным доступом.
- Структурная парадигма
  - Отказ от произвольных переходов между подпрограммами;
  - Построение программы из трех базовых управляющих структур:
    - Последовательность
    - Ветвление
    - Цикл

Декларативная парадигма

- Основные черты:
  - задаётся спецификация решения задачи, то есть описывается, **что** представляет собой проблема и ожидаемый результат;
  - отсутствует явное указание на то, как именно следует решать данную задачу – это решение принимается компилятором/интерпретатором языка;
- Главное достоинство:
  - Позволяет сконцентрироваться на понимании задачи, а не на деталях ее решения, оставляя детали реализации машине. (*«Я не тактик, стратег!»*)
    - При определенных условиях это позволяет легче избегать **побочных эффектов** и эффективнее оптимизировать код.
- Главный недостаток:
  - Сложность стыковки с реальным миром
    - Императивность нижележащих вычислительных средств;
    - Синхронизация с пользователями и внешними процессами;

Развитие декларативной парадигмы

- Функциональная и логическая парадигма:
  - LISP -> Common Lisp, Scheme, Clojure.
  - Prolog -+-> Erlang
  - ML -> Standard ML, OCAML, Haskell, F#, etc.
  - Scala, C# – мультипарадигменные языки
- Языки запросов
  - SQL, XQuery, XSLT, etc.
- Языки разметки
  - HTML, XAML
- Различные DSL

Функциональная парадигма

- Процесс получения результата работы программы трактуется как вычисление значений функций
  - Функции в математическом понимании, **без состояний**;
- Особенности:
  - Функции высшего порядка
    - Функции как first-class citizens;
    - Каррирование (карринг)
  - Чистые функции (без побочных эффектов)
    - Дает возможность ряда оптимизаций (параллельность вычислений, мемоизация, редуцирование графа вычислений)
  - Рекурсия заменяет циклы
    - Не от хорошей жизни (нет состояния – нет условия проверки/счетчика цикла)
    - Хвостовая рекурсия оптимизируется обратно в цикл ☺
  - Ленивое вычисление
    - Бесконечные последовательности и другие
- Проблемы:
  - Иммутабельность данных повышает потребление памяти
  - Проблемы с вводом-выводом (не является чистой функцией)
  - Сложность моделирования систем
  - Сложность адаптации программистов ☺

Развитие декларативной парадигмы

- Функциональная и логическая парадигма:
  - LISP -> Common Lisp, Scheme, Clojure.
  - Prolog -+-> Erlang
  - ML -> Standard ML, OCAML, Haskell, F#, etc.
  - Scala, C# – мультипарадигменные языки
- Языки запросов
  - SQL, XQuery, XSLT, etc.
- Языки разметки
  - HTML, XAML
- Различные DSL

ОО-парадигма

- Ключевые идеи:
  - Программа - это совокупность объектов, способных взаимодействовать друг с другом посредством сообщений;
  - Каждый объект является экземпляром определенного класса;
  - Классы образуют иерархию наследования.
- Фактически, ОО-программа – это работающая модель.
  - Как реализовано поведение элементов этой модели – императивно или декларативно – значения не имеет.
    - Хотя большинство популярных ОО-языков имеют императивные корни (C#, Java, C++), есть и функциональные ОО-языки (OCaml, ОО-диалекты Haskell)
  - Само по себе понятие «метода» не является обязательным для ОО-парадигмы.
    - метод – частный случай реакции на сообщение.

Парадигма – это не догма

- Парадигма – это не про язык, на котором мы пишем, это про то, как мы проектируем нашу программу:
  - Можно всё ;)
    - ООП на ассемблере? Почему бы нет?
    - ФП на C? Почему бы нет?
  - Но специализированные языки:
    - Уменьшают количество работы в данной парадигме;
    - Уменьшают вероятность «отстрелить себе ногу»;
- Многие парадигмы не противоречат друг другу, а взаимодополняют:
  - ООП
  - Обобщенная (generics)
  - Метапрограммирование (рефлексивное программирование)
  - Реактивное программирование и т.д.
- Большинство современных языков/платформ программирования – мультипарадигменные, т.е. явно поддерживают различные парадигмы:
  - JVM – Java/Scala
  - .NET – C#/F#
  - Python и т.д.

Обратная сторона :)

- Т.к. мы исполняем программы на архитектуре фон Неймана, исполняемый код все равно должен быть императивным.
- Т.е., если мы пишем НЕ в императивной парадигме, компилятор ЯП в любом случае должен перевести наш код в императивный.
- Нам необходимо понимать, как он это делает, для того, чтобы видеть правильную картину мира и писать эффективный код.
  - Помните, почему рекурсия должна быть хвостовой?
- Рассмотрим реальный пример простого декларативного языка и как его конструкции транслируются в императивную форму

LINQ – декларативный доступ к данным в .NET

- **LINQ** (**L**anguage-**IN**tegrated **Q**uery) - язык запросов, встроенный в язык программирования.
- **Запрос (query)** представляет собой выражение, извлекающее данные из источника данных.
- Источники данных могут быть разнообразны – коллекции объектов, БД, документы XML и т.п. Соответственно, для каждого типа источников существует множество различных способов описания запросов – операции ЯП, SQL, XPath/XQuery.
- Общая структура запроса:
  - источник (**откуда** берем?) – исходное множество
  - критерии отбора (**что** берем?) – операция селекции
  - выходная структура (**как** берем?) – операция проекции

Запрос. Пример на чистом C#.

- Задача. Есть набор кортежей (имя, возраст). Сформировать набор строк, содержащих имена объектов, чей возраст меньше 20.
- Код на чистом C#:

```
struct Human { public string name; public int age; };
…
var list = new List<Human>();
list.Add(new Human{name = "Vasya", age = 20});
list.Add(new Human{name = "Petya", age = 19});
list.Add(new Human{name = "Masha", age = 21});
…
List<string> s = new List<string>();
foreach (var human in list) //источник и итератор
{
    if (human <=20) //селекция
        s.Add(human.name); //проекция
}
return s;
```

Запрос. Пример LINQ.

```
struct Human { public string name; public int age; };
…
var list = new List<Human>();
list.Add(new Human{name = "Vasya", age = 20});
list.Add(new Human{name = "Petya", age = 19});
list.Add(new Human{name = "Masha", age = 21});
…
var results = from human //переменная-итератор
              in list //источник
              where human.age <= 20 //селекция
              select human.name; //проекция


return results.ToList(); //выполнение
```

Синтаксические элементы языка запросов LINQ

- **from, in** - используются для определения основы любого выражения LINQ, позволяющего извлечь подмножество данных из подходящего контейнера;
- **where** - используется для определения ограничения, в соответствии с которым должны быть извлечены элементы из контейнера;
- **select** - используется для выбора последовательности из контейнера;
- **join, on, equals, into** - выполняет соединения на основе указанного ключа;
- **orderby, ascending, descending** - позволяет результирующему подмножеству быть упорядоченным по возрастанию или убыванию;
- **group, by** - порождает подмножество с данными, группированными по указанному значению.

LINQ. Особенности

- Запросы LINQ строго типизированы. Пусть **var** не вводит в заблуждение.
- Источник данных должен быть объектом запрашиваемого типа (queryable type) – реализовывать **IEnumerable<T>** или производные от него интерфейсы (как **IQueryable<T>**). Некоторые не поддерживающие этот интерфейс типы (Array, ArrayList) отдельно расширены методами расширения, определенными в классе System.Linq.Enumerable.
- Запрос хранится в переменной запроса и инициализируется выражением запроса. Чтобы упростить написание запросов, в C# появился специальный синтаксис запросов (**from**, **where**, **select**, etc.).
- Сама переменная запроса не предпринимает действий и не возвращает никаких данных. Она просто хранит сведения, необходимые для предоставления результатов при последующем выполнении запроса.
- Возвращаемый операторами LINQ объект всегда реализует **IEnumerable<T>**
- Вычисление запроса является **ленивым** и получение данных происходит при:
  - итерировании по объекту результату методами IEnumerable<T>
  - применении аггрегирующих функций (Count, Max, Average и First.)
  - вызове кэширующих методов ToList или ToArray

LINQ. Отложенное исполнение

```
var list = new List<Human>();

list.Add(new Human { name = "Vasya", age = 20 });
list.Add(new Human { name = "Petya", age = 19 });
list.Add(new Human { name = "Masha", age = 21 });

var query = from item in list where item.age < 20 select item.name;

List<string> s = query.ToList<string>();   // <---- (1) Petya

list.Add(new Human { name = “Dima", age = 18 });

//List<string> s = query.ToList<string>();    <---- (2) Petya, Dima
```

LINQ. Комбинирование запросов

```
var list = new List<Human>();


list.Add(new Human { name = "Vasya", age = 20 });
list.Add(new Human { name = "Petya", age = 19 });
list.Add(new Human { name = "Masha", age = 21 });


// тип query – IEnumerable<string>
var query = from item in list where item.age < 20 select
item.name;


query = from name in query where name.StartsWith(“D") select
name;
```

LINQ. Агрегирующие функции

- Стандартные **Average**, **Count**, **Sum**, **Min**, **Max**
- Функция произвольной агрегации **Aggregate**

```
var list = new List<Human>();
list.Add(new Human { name = "Vasya", age = 20 });
list.Add(new Human { name = "Petya", age = 19 });
list.Add(new Human { name = "Masha", age = 21 });
list.Add(new Human { name = "Karapuz", age = 1 });

var avg = list.Average(x => x.age);
var cntQuery=(from item in list where item.age > avg select
1).Count();

Console.WriteLine("{0} элемент имеют возраст выше среднего",
cntQuery);

var aggregateQuery =
    (from item in list orderby item.age select item.name)
    .Aggregate((buffer, value) => buffer + ">" + value);


Console.WriteLine(aggregateQuery);
```

LINQ. Под капотом

- Логика LINQ реализована в классах System.Linq.Enumerable и System.Linq.Queriable в виде методов расширения.
- Оператор from:
  - Устанавливает контекст итерирования
- Оператор **where**:
  - public static IEnumerable<TSource> **Where**<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
  - **list.Where<Human>(item => item.age < 20)**
- Оператор **select**:
  - public static IEnumerable<TResult> **Select**<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector)
  - **list.Where<Human>(item => item.age < 20).Select<Human, string>(item => item.name)**

LINQ. Альтернативный синтаксис

```
var list = new List<Human>();
list.Add(new Human { name = "Vasya", age = 20 });
list.Add(new Human { name = "Petya", age = 19 });
list.Add(new Human { name = "Masha", age = 21 });

//var query = from item in list where item.age < 20 select
item.name;
var query = list
    .Where<Human>(item => item.age < 20)
    .Select<Human, string>(item => item.name);
list.Add(new Human { name = “Dima", age = 18 });
List<string> s = query.ToList<string>();
```

LINQ. IEnumerable

- IEnumerable<T> – крайне простой интерфейс:
  - Единственный «родной» метод:
    - IEnumerator<T> **GetEnumerator**()
  - Множество методов-расширений LINQ:
    - Такие как Select, Where, OrderBy, etc.
- IEnumerator<T> – это классический итератор:
  - Свойство T Current { get; }
  - Методы
    - bool MoveNext();
    - void Reset();

Методы-расширения LINQ

- Реализуют Fluent interface:
  - IEnumerable.Where:
    - Принимает Func<TSource, bool> predicate
    - Возвращает **IEnumerable**<TSource>
  - IEnumerable.Select
    - Принимает Func<TSource, TResult> selector
    - Возвращает **IEnumerable**<TResult>
- Ленивое выполнение:
  - Все переданные делегаты (и др. данные) просто сохраняются внутри объекта, реализующего IEnumerable.
  - Применяются к коллекции в необходимом порядке при вызове IEnumerable.GetEnumerator();

IEnumerable и IQueryable

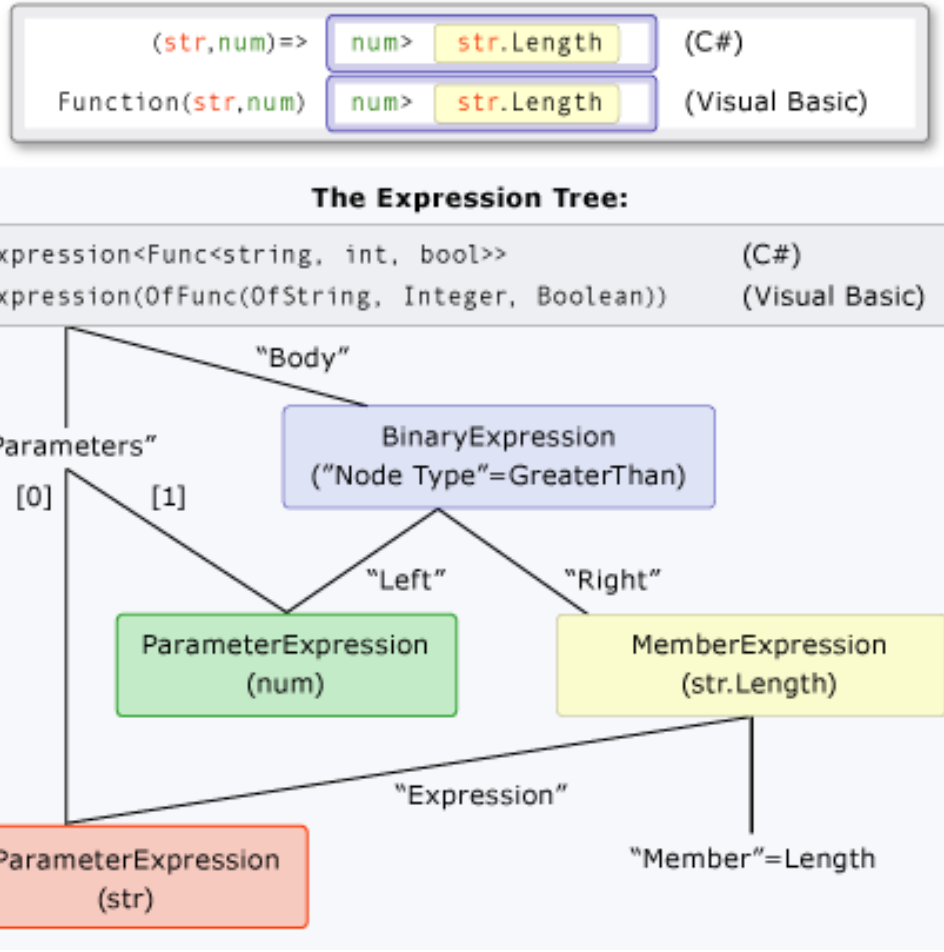
- IEnumerable:
  - реализует работу напрямую через делегаты, как в предыдущем примере.
  - поддерживает перебор коллекций.
- IQueryable:
  - хранит **дерево выражений**, описывающее запрос;
  - поддерживает возможность доступа к произвольному источнику данных, реализующему System.Linq.IQueryProvider – основа LINQ to *;
  - Дает большую гибкость, возможность оптимизации запроса по дереву выражения.

LINQ. IQueryable

- Унаследован от IEnumerable со всеми вытекающими.
- Имеет свойство IQueryProvider Provider
  - который умеет обрабатывать запросы к источнику данных
- Имеет дополнительный набор методов-расширений LINQ, принимающих не делегат, а **Expression** от этого делегата
  - Where
    - **Expression**<Func<TSource, bool>> predicate
  - Select
    - **Expression**<Func<TSource, TResult>> selector

Деревья выражений

- Деревья выражений представляют код в виде данных.
- Данные хранятся в древовидной структуре.
- Каждый узел в дереве выражений представляет выражение, например вызов метода или операцию, такую как x<y.
- Неизменность (immutability). Копирование для изменения.
- Применение деревьев выражений.



Построение деревьев выражений на основе лямбда-выражений

```
using System.Linq.Expressions;

...

Expression<Func<int, bool>> exprTree = num => num < 5;//исходная
    лямбда


ParameterExpression param =
    (ParameterExpression)exprTree.Parameters[0];

BinaryExpression operation = (BinaryExpression)exprTree.Body;

ParameterExpression left = (ParameterExpression)operation.Left;

ConstantExpression right = (ConstantExpression)operation.Right;


Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",

    param.Name, left.Name, operation.NodeType, right.Value);
```

Построение деревьев выражений программно

```
using System.Linq.Expressions;

...

//Вручную строим дерево выражения для лямбды num => num < 5.

ParameterExpression numParam = Expression.Parameter(typeof(int),
    "num");

ConstantExpression five = Expression.Constant(5, typeof(int));

BinaryExpression numLessThanFive = Expression.LessThan(numParam,
    five);


Expression<Func<int, bool>> lambda1 = Expression.Lambda<Func<int,
    bool>>( numLessThanFive, new ParameterExpression[]
    { numParam });


int i = Int32.Parse(Console.ReadLine());

Func<int, bool> f = lambda1.Compile();

Console.WriteLine(f(i));
```

LINQ. Методы IQueryable

- Аналогично IEnumerable, реализуют Fluent Interface, возвращая **IQueryable**
- Добавляемые этими методами лямбды преобразуются в деревья выражений и отправляются провайдеру **IQueryable**.**Provider**, метод **CreateQuery** которого уже и возвращает новый сконструированный **IQueryable**, готовый к выборке данных.
- При инстанцировании итератора через **GetEnumerator** происходит выполнение запроса к источнику данных через **IQueryProvider.Execute**.
- И вот тут уже сконструированный по полученному дереву выражений запрос исполняется на источнике данных.