



Информатика

функциональное
программирование

Императивное программирование

- То, как мы обычно программировали на C#
 - *писали инструкции*, описывающие **как** решать задачу и представить результат
 - инструкции – приказы (imperative)
 - do – сделай, Write – запиши, new - создай
 - *использовали переменные* для хранения состояний
 - *присваивали значения* переменным

Есть и другой подход

- **Декларативный**

- declarative – описательный
- *написание спецификаций* решения
- описание того, **что** представляет задача и ожидаемый результат
- человеко-ориентированный
- нет указаний (как) и машинной детализации

Примеры декларативных языков

- Языки разметки:
 - HTML
 - XML
- *SQL* – язык структурированных запросов к реляционным базам данных

Мы использовали декларативность

- Инициализация

```
List<Person> team = new List<Person>
{
    new Person() { Age = 25, Name = "Anna" },
    new Person() { Age = 30, Name = "Bob" },
    new Person() { Age = 35, Name = "Charlie" },
    new Person() { Age = 30, Name = "Dixin" },
};
```

```
List<Person> team = new List<Person>();
```

```
Person anna = new Person();
    anna.Age = 25;
    anna.Name = "Anna";
team.Add(anna);
Person bob = new Person();
    bob.Age = 30;
    bob.Name = "Bob";
team.Add(bob);
Person charlie = new Person();
    charlie.Age = 35;
    charlie.Name = "Charlie";
team.Add(charlie);
Person dixin = new Person();
    dixin.Age = 30;
    dixin.Name = "Dixin";
team.Add(charlie);
```

Функциональное программирование

Неважно с каким языком вы работаете, программирование в функциональном стиле имеет преимущества. Вам следует использовать его когда это удобно и серьёзно задуматься о своём решении когда оно не удобно.

Джон Кармак
(<http://ubm.io/1HOVGWs>)

ФП vs ООП

Объектно-ориентированное программирование делает код понятнее за счёт инкапсулирования движущихся частей. Функциональное программирование делает код понятнее, минимизируя количество движущихся частей.

Майкл Фезерс
(<http://bit.ly/1HOVBSM>)

Функциональное программирование

- Это *декларативное* программирование
- Основное понятие – **функция**
 - в отличном от императивного программирования смысле
- Исполнение программы – вычисление значений функций
- *Нет переменных и состояний*

ФП

- ФП – программирование с *математическими функциями*
 - не методами класса
- Математическая функция – отображение, преобразующая входные значения в выходные
 - Одни данные – один результат

Рассмотрим концепции ФП

- Принципы, характерные для функциональных языков
- Использование этих принципов в языке C#
- Обсудим специфику, применимость

Ссылочная прозрачность

- Детерминированность
- Когда одни и те же данные приводят к одному и тому же результату
- Недетерминированность – неопределённость, когда такой предсказуемости нет

Одни данные – один результат

```
public double Calculate(double x, double y)
{
    return Math.Sqrt(x*x + y*y);
}
```

```
public long TicsElapsedFrom(int year)
{
    return (DateTime.Now - new DateTime(year, 1, 1))
        .Ticks;
}
```

Одни данные – один результат

Ссылочная прозрачность (referential transparency)

Не влияет и не зависит от глобального состояния

```
public double Calculate(double x, double y)
{
    return Math.Sqrt(x*x + y*y);
}
```



```
public long TicksElapsedFrom(int year)
{
    return (DateTime.Now - new DateTime(year, 1, 1))
        .Ticks;
}
```



Честная сигнатура метода

- Точно описывает все возможные входы и выходы
- Нечестная сигнатура. Почему?

```
public static int Divide(int x, int y)
{
    return x / y;
}
```

Честная сигнатура

- Гарантии за счёт использования собственного типа

```
public static int Divide(int x, NonZeroInteger y)
{
    return x / y.Value;
}
```

- Использование nullable int

```
public static int? Divide(int x, int y)
{
    if (y == 0)
        return null;
    return x / y;
}
```

Уменьшение сложности

- Функции в математическом смысле облегчают:
 - композицию
 - понимание
 - тестирование
- Следовательно, уменьшают сложность

Чистые функции

- *Детерминированные*
обладают ссылочной прозрачностью
ОДИН ВХОД – ОДИН ВЫХОД
- *Не обладающие побочными эффектами*
не изменяют состояние

Неизменяемость

- *Immutability* – невозможность изменить данные
- *Состояние* – данные, изменяемые со временем
- *Побочные эффекты* – изменения некоторого состояния

Почему неизменяемость важна?

Изменяемые операции = Нечестный код

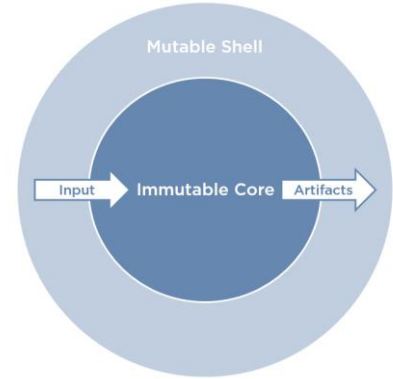
- Улучшает читаемость кода
- Валидация требуется только в одном месте (при создании)
- Автоматически гарантирует потокобезопасность

Борьба с побочными эффектами

- Предположим, нужно уметь изменять данные пользователя
- Разделить операции на команды и запросы (**command-query request separation: CQRS**)
 - *команды* – содержат побочные эффекты, ничего не возвращают, изменяют состояние
 - *запросы* – свободны от побочных эффектов, возвращают результат

Борьба с побочными эффектами

- Выделить
 - логику предметной области
не меняет состояние
 - изменяемое состояние
не содержит логики, но меняет состояние
- *Неизменяемое ядро в изменяемой оболочке*



Исключения

- **Исключения – как goto**
 - может даже хуже, т.к. могут перемещать поток управления между методами разных компонент
- ***Метод с исключениями – не математическая функция***
 - скрывается информация об исключении
- Следует всегда предпочесть возвращение значение исключениям

Когда использовать исключения

- В исключительных ситуациях
- Исключения – как сигнал о баге (неисправимая ситуация)
- Не использовать там, где вы ожидаете возникновение исключения
 - валидация \neq исключительная ситуация
 - валидация – фильтр

Функции первого класса

- В ФП ключевое понятие – функция
- Нужно уметь создавать, изменять и передавать функции в качестве аргументов другим функциям

Функции должны быть отдельными объектами

Типы для функций в C#

- *Делегат* – ссылка на метод(ы)
- Ссылочный тип, содержащий
 - Адрес метода
 - Ссылку на экземпляр
 - Ссылку на массив цепочки делегатов

Делегат

```
internal delegate void Feedback(Int32 value);
```

генерирует после компиляции класс

```
internal class Feedback : System.MulticastDelegate
{
    // Конструктор
    public Feedback(Object object, IntPtr method);
    // Метод, прототип которого задан в исходном тексте
    public virtual void Invoke(Int32 value);
    // Методы, обеспечивающие асинхронный обратный вызов
    public virtual IAsyncResult BeginInvoke
        (Int32 value, AsyncCallback callback, Object object);
    public virtual void EndInvoke(IAsyncResult result);
}
```

Метод как значение делегата

```
// Делегат как класс (правила объявления те же)
delegate int MyDelegate(int x, int y);

static int SumMethod(int x, int y)
{
    return x + y;
}

...

MyDelegate d1 = SumMethod;
```

Анонимные методы

- Позволяют описать тело метода без указания имени (C# 2.0)

```
delegate int MyDelegate(int x, int y);
```

...

```
MyDelegate d2 = delegate(int x, int y) { return x + y; };
```

Лямбда-выражения

- При объявлении анонимного метода ключевое слово `delegate` можно опустить
- И использовать `'=>'` вместо `return`

```
MyDelegate anonymous = delegate(int x, int y)
                        { return x + y; };
```

```
MyDelegate lambda = (int x, int y)=>x + y;
```

Лямбда-исчисление Чёрча

- Формальная система, разработанная Алонзо Чёрчем для формализации понятия вычислимости, использующая функции
- Альтернатива формальной системе Алана Тьюринга (машине Тьюринга)

Анонимные обобщенные делегаты

Начиная с C# 3:

- **Action** – семейство делегатов, не возвращающих значения
- **Func** – семейство делегатов, возвращающих значение
- **Predicate** – делегат, возвращающий bool

Анонимные делегаты

- Позволяют сэкономить время, избавляя от необходимости объявлять делегаты с одинаковыми определениями
- Активно используются с методами расширения и LINQ

```
Func<int, int, int> funcAndLambda = (x, y) => x + y;
```


Action

- Семейство из 17 делегатов, не возвращающих значение

```
public delegate void Action<T>(T obj);  
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);  
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3);  
...  
public delegate void Action<T1, ..., T16>(T1 arg1, ..., T16 arg16);
```

```
Action<string, string> Hello = (x, y) => Console.WriteLine($"{x} {y}");
```

Func

- Семейство из 17 делегатов, возвращающих значение

```
public delegate TResult Func<TResult>();  
public delegate TResult Func<T, TResult>(T arg);  
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);  
public delegate TResult Func<T1, T2, T3, TResult>  
(T1 arg1, T2 arg2, T3 arg3);  
...  
public delegate TResult Func<T1,..., T16, TResult>  
(T1 arg1, ..., T16 arg16);
```

```
Func<int, int, int> funcAndLambda = (x, y) => x + y;
```

Multicast

```
// Реализация метода Invoke класса Delegate
public void Invoke(Int32 value)
{
    Delegate[] delegateSet = _invocationList as Delegate[];
    if (delegateSet != null)
    {
        foreach (Feedback d in delegateSet)
            d(value); // Вызов каждого делегата
    }
    else
    {
        _methodPtr.Invoke(_target, value);
        // Строка выше – имитация реального кода.
    }
}
```

Арифметика на делегатах

```
delegate void HelloWorld();    HelloWorld hello = Hello;
                                hello += World;
static void Hello()           hello();
{
    Console.WriteLine("Hello "); hello -= World;
}                               hello();
static void World()
{
    Console.WriteLine
    ("world!");
}
...
```

Hello world!
Hello

Замыкания

- *Замыкание* – структура данных для хранения функции вместе с окружением
© Wiki
- Замыкание, прикрепленное к родительскому методу имеет доступ к членам, определённым в теле родительского метода

Замыкания

```
public Person FindById(int id)
{
    return this.Find(delegate (Person p)
    {
        return (p.Id == id);
    });
}
```

Захват в замыкание

- Переменная, захваченная в замыкание продлевает время жизни пока живо замыкание
- Не важно, какого она типа: ссылочного или типа значения

Замыкания на переменных цикла

```
var actions = new List<Action>();  
foreach (var i in Enumerable.Range(1, 3))  
{  
    actions.Add(() => Console.WriteLine(i));  
}
```

```
foreach (var action in actions)  
{  
    action();  
}
```

```
// До C# 5: 3 3 3
```

```
// После C# 5: 1 2 3
```

[Статья на хабре](#)

Функции высших порядков

- Функции могут использоваться в качестве аргументов и возвращаемых значений других функций

```
public static string Apply (Func<string,string> func, string value)
{
    return func(value);
}
...
var str=Apply(x => x.ToLower(),"Hello world!");
```

Композиция функций

$$f \circ g(x) = g(f(x))$$

```
static Func<X, Z> Compose<X, Y, Z>(Func<X, Y> f, Func<Y, Z> g)
{
    return (x) => g(f(x));
}
```

```
Func<double, double> sin = Math.Sin;
Func<double, double> exp = Math.Exp;
Func<double, double> exp_sin = Compose(sin, exp);
double y = exp_sin(3);
```

Рекурсия

- Вызов из функции самой себя
- Мы работали с рекурсивными методами
- Пример: вычисление n-го числа Фибоначчи (что в нём плохого?)

```
Func<uint, uint> fib = null;  
fib = x => x > 1 ? fib(x - 1) + fib(x - 2) : x;
```

Мемоизация

```
public static class FuncExtensions
{
    public static Func<A, R> Memoize<A, R>(this Func<A, R> func)
    {
        var dict = new Dictionary<A, R>();
        return a =>
        {
            R r;
            if (!dict.TryGetValue(a, out r))
            {
                r = func(a);    dict.Add(a, r);
            }
            return r;
        };
    }
}
```

Сравнение

```
public static void Main()
{
    Func<uint, uint> fib = null;
    fib = x => x > 1 ? fib(x - 1) + fib(x - 2) : x;

    Stopwatch s = new Stopwatch();
    s.Start();
    Console.WriteLine(fib(37));
    Console.WriteLine("Without optimization: {0}", s.Elapsed);

    fib = fib.Memoize();
    s.Restart();
    Console.WriteLine(fib(37));
    Console.WriteLine("With optimization: {0}", s.Elapsed);
}
```

24157817

Without optimization: 00:00:02.6608453

24157817

Without optimization: 00:00:00.0097345

Частичное применение

- Частично-применённые функции уменьшают количество параметров *подстановкой (фиксацией) значений аргументов*

```
public static Func<T2, TR> Partial1<T1, T2, TResult>  
(this Func<T1, T2, TResult> func, T1 first)  
{  
    return b => func(first, b);  
}
```

Partial application по 1 аргументу

- Фиксируя первый аргумент у `Math.Pow` можно получать функции возведения в степень с фиксированным основанием

```
double x;  
Func<double, double, double> pow = Math.Pow;  
  
Func<double, double> exp = pow.Partial1(Math.E);  
// exp(x) = Math.Pow(Math.E, x)  
Func<double, double> step = pow.Partial1(2);  
// step(x) = Math.Pow(2, x)  
x = exp(Math.PI);  
x = step(Math.Log(int.MaxValue));
```

Partial application по 2 аргументу

- Вместо первого аргумента можно фиксировать значение любого

```
public static Func<T1, TR> Partial2<T1, T2, TR>  
(this Func<T1, T2, TR> func, T2 second)  
{  
    return a => func(a, second);  
}
```


Partial application по 2 аргументу

- Фиксируя второй аргумент у `Math.Pow`, можно получать функции возведения аргумента в некоторую фиксированную степень (корень, квадрат, куб)

```
double x;  
Func<double, double, double> pow = Math.Pow;  
  
Func<double, double> square = pow.Partial2(2); // Math.Pow(x,2)  
Func<double, double> sqrt = pow.Partial2(0.5); // Math.Pow(x,0.5)  
Func<double, double> cube = pow.Partial2(3); // Math.Pow(x,3)  
  
x = square(5); //x = 25  
x = sqrt(9); //x = 3  
x = cube(3); //x = 27
```

Каррирование

- *Преобразование функции от N аргументов в цепочку из N функций одного аргумента*
- Названо в честь Хаскелла Карри
- Удобно для обработки аргументов по одному
- В некоторых функциональных языках каррирование автоматическое

Каррирование вычисления расстояния

- Пусть есть метод, вычисляющий расстояние от центра координат (0,0,0) до точки (x, y, z) в трёхмерном Евклидовом пространстве

```
static double Distance(double x, double y, double z)
{
    return Math.Sqrt(x * x + y * y + z * z);
}
```

Функция каррирования

- Используем функцию высшего порядка для разбиение функции от трёх аргументов в цепочку функций одного аргумента

```
public static Func<T1, Func<T2, Func<T3, TResult>>>  
Curry<T1, T2, T3, TResult>(this Func<T1, T2, T3, TResult> function)  
{  
    return a => b => c => function(a, b, c);  
}
```

Вычисление расстояния

```
Func<double, double, double, double> fnDistance = Distance;
```

```
var curriedDistance = fnDistance.Curry();
```

```
double d = curriedDistance(3)(4)(12);
```

```
double d2 = Distance(3, 4, 12);
```

Раскаририрование

- *Обратный каррированию процесс*
- Цепочка N функций одного аргумента преобразуется в одну функцию от N аргументов

```
public static Func<T1, T2, T3, TR> UnCurry<T1, T2, T3, TR>  
(this Func<T1, Func<T2, Func<T3, TR>>> carriedFunc)  
{  
    return (a, b, c) => carriedFunc(a)(b)(c);  
}
```

Раскарирование

- Пример:

```
var curriedDistance = Curry<double, double, double, double>(Distance);  
double d = curriedDistance(3)(4)(12);
```

```
Func<double, double, double, double> originalDistance =  
curriedDistance.UnCurry();  
d = originalDistance(3, 4, 12);
```

Ошибка на миллиард \$

«Нулевые ссылки: ошибка на миллиард долларов»

Тони Хоар ([QCon 2009](#))

- По сигнатуре метода не понять, может ли результатом быть null или нет, нужно смотреть детали реализации.

Как избежать проблеме 10^9 \$

- Если метод может вернуть null, можно возвращать обёрнутый в Maybe<T> результат метода
 - после этого сигнатура будет честной
- Пустым значением (Nothing) у Maybe<T> будет пустой Maybe<T>

Монады

- В функциональных языках конструкция `Maybe<T>` называется монадой
- Монада – контейнер, инкапсулирующий функции с побочным эффектом от чистых функций
- Концепция монад унаследована из теории категорий
 - монада – моноид в моноидальной категории эндифункторов

Реализация монады Maybe

- Обычно, Maybe включает 2 метода:
 - втягивание в монаду или помещение значения в контейнер (Return)
 - связывание (Bind)
стратегия – «если первое вычисление дало результат, то второе; иначе — отсутствие результата»

Реализация монады Maybe

```
public class Maybe<T>
{
    private readonly T _value;
    public Maybe(T value)
    {
        _value = value;
    }
    private Maybe() { }
    public static Maybe<T> Nothing
        = new Maybe<T>();
}
```

```
public Maybe<T2> Bind<T2>
    (Func<T, Maybe<T2>> func)
{
    return _value != null ?
        func(_value) :
        Maybe<T2>.Nothing;
}
public Maybe<T> Bind
    (Action<T> action)
{
    if (_value == null)
        return Nothing;
    action(_value);
    return this;
} }
```

Использование Maybe

```
var str = new Maybe<string>(Console.ReadLine());  
  
str.Bind(x => new Maybe<string>(x.ToUpper()))  
    .Bind(Console.WriteLine)  
    .Bind(x=>new Maybe<string>(x.ToLower()))  
    .Bind(Console.WriteLine);
```

Отсутствие состояний

- В чисто функциональных языках отсутствуют состояния
- Состояния можно моделировать с помощью монадических вычислений

Другие монады

- **IO** – монада строгой последовательности вычислений
 - стратегия связывания: «сначала первое, затем второе»
- **List** – монада вычислений с несколькими результатами
 - стратегия связывания: «все возможные результаты второго вычисления, применённого к каждому из вычисленных первым значений параметра»
- **State** – монада вычислений с переменной состояния
 - стратегия связывания: «начать второе вычисление с состоянием, изменённым в результате первого»

Вопросы

- В чем отличие императивного и декларативного подхода?
- Как соотносятся функциональное и декларативное программирование?
- Что такое ссылочная прозрачность?
- Что такое честная сигнатура метода?

Вопросы

- Что такое состояние?
- Что такое неизменяемость?
- Что такое побочный эффект и чистая функция?
- Что такое функция первого рода и функции высших порядков?

Вопросы

- Какие механизмы поддержки ФП есть в C#?
- Чем отличается каррирование от частичного применения?
- В чём состоит проблема на миллиард \$ и как можно её решать?

На следующей лекции

- Вычисления на последовательностях
- Ленив и энергичность
- LINQ – язык внутри языка C#
- Алгоритмы Map & Reduce
- Параллельные вычисления в LINQ



Вопросы?

e-mail: marchenko@it.kfu.ru