Contents

- Документация по F#
- Что такое F#
- Начало работы
  - Установка F#
  - F# в Visual Studio
  - F# в Visual Studio Code
  - F# с .NET Core CLI
  - F# в Visual Studio для Mac
- Обзор языка F#
- Учебники
  - Введение в функциональное программирование
    - Функции первого класса
  - Асинхронное и параллельное программирование
    - Асинхронное программирование
  - Поставщики типов
    - Создание поставщика типов
    - Безопасность поставщиков типов
    - Устранение неполадок поставщиков типов
  - F# Interactive
- Новые возможности F#
  - F# 4.7
  - F# 4.6
  - F# 4.5
- Справочник по языку F#
  - Справочные сведения о ключевых словах
  - Справочник символов и операторов
    - Арифметические операторы
    - Логические операторы
    - Побитовые операторы

- Операторы, допускающие значение NULL
- Функции
  - Привязки let
  - Привязки do
  - Лямбда-выражения: ключевое слово fun
  - Рекурсивные функции: ключевое слово rec
  - Точка входа
  - Внешние функции
  - Встраиваемые функции
- Значения
  - Значения NULL
- Литералы
- Типы языка F#
- Вывод типа
- Базовые типы
- Тип Unit
- Строки
- Кортежи
- Типы коллекций F#
- Списки
- Массивы
- Последовательности
- Срезы
- Параметры
- Параметры значений
- Результаты
- Универсальные шаблоны
  - Автоматическое обобщение
  - Ограничения
  - Статически разрешаемые параметры типов
- Записи
- Анонимные записи

Копирование и обновление выражений записей
Размеченные объединения
Перечисления
Сокращенные обозначения типов
Классы
Структуры
Наследование
Интерфейсы
Абстрактные классы
Участники
  Привязки let в классах
  Привязки do в классах
  Свойства
  Индексированные свойства
  Методы
  Конструкторы
  События
  Явные поля. Ключевое слово `val`
Расширения типов
Параметры и аргументы
Перегрузка операторов
Гибкие типы
Делегаты
Выражения объекта
Приведение и преобразование
Контроль доступа
Условные выражения. if...then...else
Выражения match
Сопоставление шаблонов
Активные шаблоны
Циклы. Выражение for...to
Циклы: выражение for...in

- Циклы: выражение while...do
- Утверждения
- Обработка исключений
  - Типы исключения
  - Выражение try...with
  - Выражение try...finally
  - Функция raise
  - Функция failwith
  - Функция invalidArg
- Атрибуты
- Управление ресурсами: ключевое слово use
- Пространства имен
- Модули
- Объявления импорта. Ключевое слово open
- Файлы подписи
- Единицы измерения
- Документация XML
- Отложенные выражения
- Выражения вычисления
- Асинхронные рабочие потоки
- Выражения запросов
- Цитирование кода
- Ключевое слово fixed
- Byref
- Ссылочные ячейки
- Директивы компилятора
- Параметры компилятора
- Параметры F# Interactive
- Идентификаторы Source Line, File и Path
- Сведения о вызывающем
- Подробный синтаксис

Руководство по стилю для F#

- Рекомендации по форматированию кода F#

Соглашения о написании кода на F#
Рекомендации по проектированию компонентов F#
Использование языка F# в Azure
Начало работы с хранилищем BLOB-объектов Azure с помощью языка F#
Начало работы с хранилищем файлов Azure с помощью языка F#
Начало работы с хранилищем очередей Azure с помощью языка F#
Начало работы с хранилищем таблиц Azure с помощью языка F#
Управление пакетами для зависимостей F# в Azure

Что такое F#

23.11.2019 • 3 minutes to read • Edit Online

F#— Это функциональный язык программирования, упрощающий написание правильного и сопровождаемого кода.

F#Программирование в основном включает определение типов и функций, которые выводятся и обобщены автоматически. Это позволяет сосредоточиться на проблемном домене и манипулировать его данными, а не на деталях программирования.

```
open System // Gets access to functionality in System namespace.

// Defines a function that takes a name and produces a greeting.
let getGreeting name =
    sprintf "Hello, %s! Isn't F# great?" name

[<EntryPoint>]
let main args =
    // Defines a list of names
    let names = [ "Don"; "Julia"; "Xi" ]

    // Prints a greeting for each name!
    names
    |> List.map getGreeting
    |> List.iter (fun greeting -> printfn "%s" greeting)

    0
```

F#содержит множество функций, включая:

- Упрощенный синтаксис
- Неизменяемое по умолчанию
- Определение типа и автоматическое обобщение
- Функции первого класса
- Мощные типы данных
- Регулярные выражения
- Асинхронное программирование

Полный набор функций приведен в F# справочнике по языку.

Типы данных с богатыми возможностями

Типы данных, такие как записи и Размеченные объединения , позволяют представлять сложные данные и домены.

```
// Group data with Records
type SuccessfulWithdrawal = {
    Amount: decimal
    Balance: decimal
}

type FailedWithdrawal = {
    Amount: decimal
    Balance: decimal
    IsOverdraft: bool
}

// Use discriminated unions to represent data of 1 or more forms
type WithdrawalResult =
    | Success of SuccessfulWithdrawal
    | InsufficientFunds of FailedWithdrawal
    | CardExpired of System.DateTime
    | UndisclosedFailure
```

F#записи и размеченные объединения не равны NULL, неизменяемы и сравнимы по умолчанию, что делает их очень простыми в использовании.

Принудительная правильность с помощью функций и сопоставления шаблонов

F#функции легко объявляют и являются мощными на практике. В сочетании с сопоставлением шаблоновони позволяют определить поведение, корректность которого обеспечивается компилятором.

```
// Returns a WithdrawalResult
let withdrawMoney amount = // Implementation elided

let handleWithdrawal amount =
    let w = withdrawMoney amount

    // The F# compiler enforces accounting for each case!
    match w with
    | Success s -> printfn "Successfully withdrew %f" s.Amount
    | InsufficientFunds f -> printfn "Failed: balance is %f" f.Balance
    | CardExpired d -> printfn "Failed: card expired on %O" d
    | UndisclosedFailure -> printfn "Failed: unknown :("
```

F#функции являются также первыми классами, то есть их можно передавать как параметры и возвращаться из других функций.

Функции для определения операций с объектами

F#обладает полной поддержкой объектов, которые являются полезными типами данных, когда требуется смешивать данные и функциональность. F#функции используются для работы с объектами.

```
type Set<'T when 'T: comparison>(elements: seq<'T>) =
    member s.IsEmpty = // Implementation elided
    member s.Contains (value) =// Implementation elided
    member s.Add (value) = // Implementation elided
    // ...
    // Further Implementation elided
    // ...
    interface IEnumerable<‘T>
    interface IReadOnlyCollection<‘T>

module Set =
    let isEmpty (set: Set<'T>) = set.IsEmpty

    let contains element (set: Set<'T>) = set.Contains(element)

    let add value (set: Set<'T>) = set.Add(value)
```

Вместо написания кода, который является объектно-ориентированным F#, в часто создается код, который обрабатывает объекты как другой тип данных для функций, которыми нужно управлять. В больших F# программах часто используются такие функции, как универсальные интерфейсы, выражения объектови разумное использование элементов .

Следующие шаги

Дополнительные сведения о большом наборе F# функций см. в этом F# обзоре.

Начало работы с F#

27.11.2019 • 2 minutes to read • Edit Online

Вы можете приступить к F# работе на компьютере или в сети.

Начало работы на компьютере

Существует несколько руководств по установке и использованию F# в первый раз на компьютере. Для принятия решения можно использовать следующую таблицу:

ОС	ПРЕДПОЧИТАТЬ VISUAL STUDIO	ПРЕДПОЧИТАТЬ VISUAL STUDIO CODE	ПРЕДПОЧИТАТЬ КОМАНДНУЮ СТРОКУ
Windows	Начало работы с Visual Studio	Начало работы с Visual Studio Code	Начало работы с .NET Core CLI
macOS	Начало работы с VS для Mac	Начало работы с Visual Studio Code	Начало работы с .NET Core CLI
Linux	Н/Д	Начало работы с Visual Studio Code	Начало работы с .NET Core CLI

Как правило, не существует конкретного, что лучше, чем остальная. Мы рекомендуем использовать все способы использования F# на вашем компьютере, чтобы узнать, что вам нравится лучше!

Начало работы

Если вы не хотите устанавливать F# и .NET на компьютере, вы также можете начать работу с F# в браузере:

- Введение в F# подшивку — это Записная книжка Jupyter , размещенная с помощью бесплатной службы Binder . Регистрация не требуется.
- ФАБЛЕ REPL — это интерактивная, в браузере REPL, которая использует фабле для преобразования F# кода в JavaScript. Ознакомьтесь с многочисленными примерами от F# основ до полноценной видеоигры в вашем браузере.

Установка F#

08.01.2020 • 3 minutes to read • Edit Online

В зависимости от F# среды можно установить несколькими способами.

Установка F# с помощью Visual Studio

1. Если вы скачиваете Visual Studio впервые, сначала будет установлена Visual Studio Installer. Установите соответствующий выпуск Visual Studio из установщика.

   Если вы уже установили Visual Studio, выберите **изменить** рядом с выпуском, который необходимо F# добавить.

2. На странице рабочие нагрузки выберите рабочую нагрузку **ASP.NET and Web Development** , включающую F# и поддержку .net Core для проектов ASP.NET Core.

3. Выберите **изменить** в правом нижнем углу, чтобы установить все, что вы выбрали.

   Затем можно открыть Visual Studio с F# помощью команды **запустить** в Visual Studio Installer.

Установка F# с помощью Visual Studio Code

1. Убедитесь, что на вашем пути установлен и доступен Git . Чтобы проверить правильность установки, введите `git --version` в командной строке и нажмите клавишу **Ввод**.

2. Установите пакет SDK для .NET Core и Visual Studio Code.

3. Щелкните значок расширения и выполните поиск по запросу "Ionide":

   Единственным подключаемым модулем F# , необходимым для поддержки в Visual Studio Code, является Ionide-FSharp. Однако можно также установить Ionide-ПОДдельные , чтобы получить поддельную поддержку и Ionide-пакет , чтобы получить поддержку пакет . Поддельные и пакет — F# это дополнительные инструменты сообщества для создания проектов и управления зависимостями соответственно.

Установка F# с помощью Visual Studio для Mac

F#устанавливается по умолчанию в Visual Studio для Mac, независимо от выбранной конфигурации.

После завершения установки нажмите кнопку **запустить Visual Studio**. Вы также можете открыть Visual Studio с помощью Finder в macOS.

Установка F# на сервере сборки

Если вы используете .NET Core или .NET Framework с помощью пакета SDK для .NET, необходимо просто установить пакет SDK для .NET на сервере сборки. Он содержит все необходимое.

Если вы используете .NET Framework и вы **не** используете пакет SDK для .NET, необходимо установить Visual Studio Build Tools SKU на сервере Windows Server. В установщике выберите **.NET Desktop Build Tools**, а затем выберите компонент **F# компилятора** в правой части меню установщика.

Начало работы с F# Visual Studio

08.01.2020 • 5 minutes to read • Edit Online

F#Визуальные F# средства поддерживаются в интегрированной среде разработки Visual Studio (IDE).

Для начала убедитесь, что у вас установлена поддержка Visual Studio F# с поддержкой.

Создание консольного приложения

Одним из самых основных проектов в Visual Studio является консольное приложение. Вот как это сделать:

1. Запустите Visual Studio 2019.
2. На начальном экране выберите **Создать проект**.
3. На странице **Создание нового проекта** выберите **F#** из списка язык.
4. Выберите шаблон **консольное приложение (.NET Core)** , а затем нажмите кнопку **Далее**.
5. На странице **Настройка нового проекта** введите имя в поле **имя проекта** . Затем нажмите **Создать**.

   Visual Studio создаст новый F# проект. Его можно увидеть в окне обозреватель решений.

Создание кода

Начнем с написания кода. Убедитесь, что файл `Program.fs` открыт, и замените его содержимое следующим:

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

Предыдущий пример кода определяет функцию с именем `square` , которая принимает входные данные с именем `x` и умножает ее на саму себя. Поскольку F# использует вывод типа, указывать тип `x` не нужно. F# Компилятор понимает типы, в которых умножение является допустимым, и назначает тип для `x` в зависимости от способа вызова `square` . При наведении указателя мыши на `square` вы должны увидеть следующее:

```
val square: x:int -> int
```

Это называется сигнатурой типа функции. Его можно прочитать следующим образом: "Square — это функция, которая принимает целое число с именем" x "и создает целое число". Компилятор предоставил `square` тип `int` для Now; Это связано с тем, что умножение не является универсальным для *всех* типов, а вместо закрытого набора типов. F# Компилятор будет корректировать сигнатуру типа при вызове `square` с другим типом входных данных, например `float` .

Определена другая функция `main` , которая снабжена атрибутом `EntryPoint` . Этот атрибут сообщает F# компилятору, что должно начаться выполнение программы. Он соответствует тому же соглашению, что и другие языки программирования в стиле C, где аргументы командной строки могут передаваться в эту

функцию, и возвращается целочисленный код (обычно `0`).

Он находится в функции точки входа `main`, что вызывается функция `square` с аргументом `12`. Затем F# компилятор назначает тип `square` `int -> int` (то есть функция, которая принимает `int` и создает `int`). Вызов `printfn` является отформатированной функцией печати, которая использует строку формата и выводит результат (и новую строку). Строка формата, аналогичная языку программирования в стиле C, имеет параметры (`%d`), соответствующие переданным в него аргументам, в данном случае `12` и `(square 12)`.

Выполнение кода

Можно выполнить код и просмотреть результаты, нажав клавиши **Ctrl**+**F5**. Кроме того, можно выбрать **отладку** > **Запуск без отладки** из строки меню верхнего уровня. При этом запускается программа без отладки.

Следующие выходные данные выводятся в окне консоли, которое открыл Visual Studio:

```
12 squared is: 144!
```

Поздравляем! Вы создали первый F# проект в Visual Studio, написали F# функцию, которая вычисляет и выводит значение, и запускает проект для просмотра результатов.

Следующие шаги

Если вы еще этого не сделали, ознакомьтесь с обзором F# , в котором рассматриваются некоторые основные функции F# языка. В нем представлен обзор некоторых возможностей F# и объемных примеров кода, которые можно скопировать в Visual Studio и выполнить.

Обзор языка F#

См. также:

- F#Справочник по языку
- Определение типа
- Справочник по символам и операторам

Начало работы с F# в Visual Studio Code

08.01.2020 • 12 minutes to read • Edit Online

Вы можете писать F# в Visual Studio Code с подключаемым модулем Ionide , чтобы получить отличное от платформы упрощенное интегрированное окружение разработки (IDE) с технологией IntelliSense и оптимизацией кода. Посетите Ionide.IO , чтобы узнать больше о подключаемом модуле.

Для начала убедитесь, что у вас есть F# и правильно установлен подключаемый модуль Ionide.

Создание первого проекта с помощью Ionide

Чтобы создать новый F# проект, откройте командную строку и создайте новый проект с .NET Core CLI:

```
dotnet new console -lang "F#" -o FirstIonideProject
```

После завершения измените каталог на проект и откройте Visual Studio Code:

```
cd FirstIonideProject
code .
```

После загрузки проекта на Visual Studio Code Вы увидите панель F# Обозреватель решений в левой части окна Открыть. Это означает, что Ionide успешно загрузил только что созданный проект. Вы можете написать код в редакторе до этой точки во времени, но после того, как это произойдет, все готово к загрузке.

Настройка F# интерактивного

Во-первых, убедитесь, что скрипты .NET Core — это среда сценариев по умолчанию:

1. Откройте **параметры Visual Studio Code (настройки > ** **кода** > **Параметры**).
2. Найдите термин **F# сценарий**.
3. Установите флажок **FSharp: использовать скрипты пакета SDK**.

В настоящее время это необходимо из-за некоторых устаревших поведений в сценариях на основе .NET Framework, которые не работают с сценариями .NET Core, и Ionide в настоящее время выполняет эту обратную совместимость. В будущем скрипты .NET Core станут значением по умолчанию.

Написание первого скрипта

После настройки Visual Studio Code для использования сценариев .NET Core перейдите в представление обозревателя в Visual Studio Code и создайте новый файл. Назовите его *мифирстскрипт. fsx*.

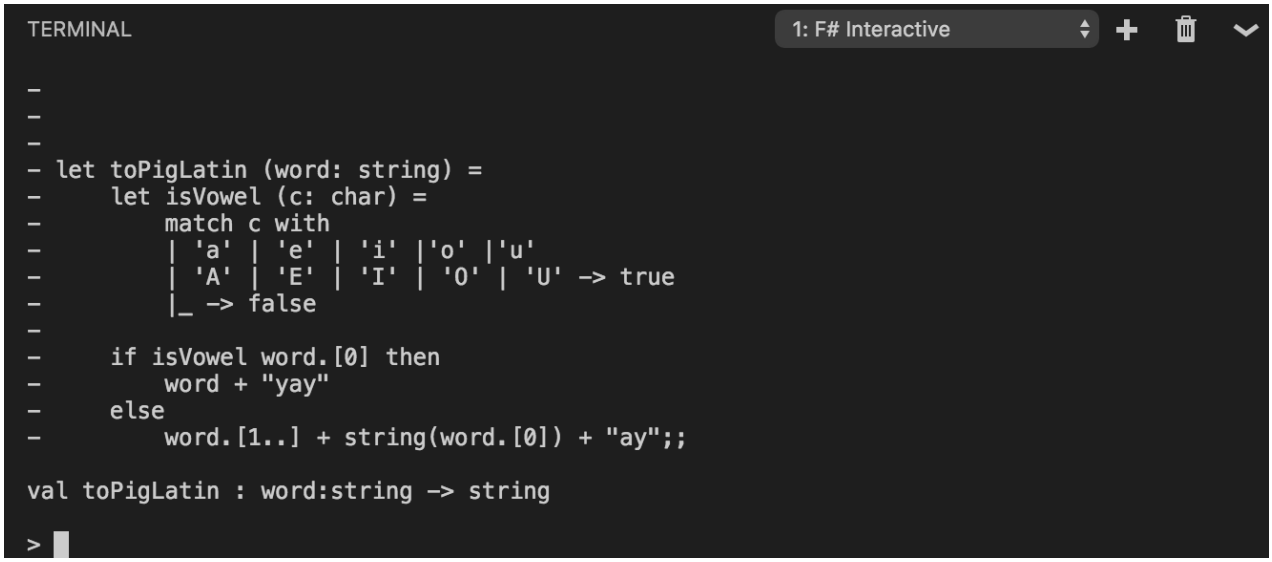
Теперь добавьте в него следующий код:

```
let toPigLatin (word: string) =
    let isVowel (c: char) =
        match c with
        | 'a' | 'e' | 'i' |'o' |'u'
        | 'A' | 'E' | 'I' | 'O' | 'U' -> true
        |_ -> false

    if isVowel word.[0] then
        word + "yay"
    else
        word.[1..] + string(word.[0]) + "ay"
```

Эта функция преобразует слово в форму Pig Latin. Следующим шагом является его оценка с помощью F# интерактивной (FSI).

Выделите всю функцию (она должна составлять 11 строк). После выделения удерживайте клавишу **ALT** и нажмите клавишу **Ввод**. В нижней части экрана появится всплывающее окно терминала, которое должно выглядеть примерно так:



Это сделало три вещи:

1. Он запустил процесс FSI.
2. Он отправил код, выделенный вам для процесса FSI.
3. Процесс FSI проверил код, который вы отправили.

Так как вы передавали функцию, теперь вы можете вызвать эту функцию с помощью FSI! В интерактивном окне введите следующее:

```
toPigLatin "banana";;
```

Вы должны увидеть следующий результат:

```
val it : string = "ananabay"
```

Теперь давайте попробуем использовать гласную в качестве первой буквы. Введите следующий текст:

```
toPigLatin "apple";;
```

Вы должны увидеть следующий результат:

```
val it : string = "appleyay"
```

Вероятно, функция работает должным образом. Поздравляем, вы только что написали первую F# функцию в Visual Studio Code и оценили ее с помощью FSI!

> **NOTE**
> Как вы могли заметить, строки в FSI прерываются с `;;`. Это обусловлено тем, что FSI позволяет вводить несколько строк. `;;` в конце позволяет сообщить ФСС о завершении кода.

Объяснение кода

Если вы не знаете, что делает код, выполните следующие действия.

Как видите, `toPigLatin` — это функция, которая принимает слово в качестве входных данных и преобразует его в представление Pig-Latin этого слова. Ниже приведены правила для этого.

Если первый символ в слове начинается с гласной, добавьте «ура» в конец слова. Если он не начинается с гласной, переместите первый символ в конец слова и добавьте в него «гг».

Вы могли заметить следующее в FSI:

```
val toPigLatin : word:string -> string
```

Это означает, что `toPigLatin` — это функция, которая принимает `string` в качестве входных данных (с именем `word`) и возвращает другой `string`. Это называется сигнатурой типа функции— основной частью F# этого ключа для понимания F# кода. Это также можно заметить при наведении указателя мыши на функцию в Visual Studio Code.

В теле функции можно заметить две различные части:

1. Внутренняя функция, именуемая `isVowel`, определяющая, является ли данный символ (`c`) гласным, проверяя, соответствует ли он одному из указанных шаблонов через сопоставление шаблонов:

   ```
   let isVowel (c: char) =
       match c with
       | 'a' | 'e' | 'i' |'o' |'u'
       | 'A' | 'E' | 'I' | 'O' | 'U' -> true
       |_ -> false
   ```

2. Выражение `if..then..else`, которое проверяет, является ли первый символ гласным, и формирует возвращаемое значение из входных символов на основе того, был ли первый символ гласным или нет:

   ```
   if isVowel word.[0] then
       word + "yay"
   else
       word.[1..] + string(word.[0]) + "ay"
   ```

Таким образом, поток `toPigLatin`:

Проверьте, является ли первый символ входного слова гласным. Если это так, прикрепите "ура" к концу слова. В противном случае переместите первый символ в конец слова и добавьте в него «гг».

Есть одно конечное замечание: нет явных инструкций для возврата из функции, в отличие от многих других

языков. Это происходит потому F# , что основан на выражениях, а Последнее выражение в теле функции является возвращаемым значением. Поскольку `if..then..else` является выражением, в зависимости от входного значения будет возвращаться тело блока `then` или тело блока `else` .

Преобразование консольного приложения в генератор Pig Latin

В предыдущих разделах этой статьи был показан общий первый шаг в написании F# кода: написание начальной функции и ее интерактивное исполнение с помощью FSI. Это называется разработкой на основе REPL, где REPL означает "чтение-вычисление-цикл печати". Это отличный способ поэкспериментировать с функциями, пока не будет выполнена какая-то работа.

Следующим шагом в разработке на основе REPL является перемещение рабочего кода в файл F# реализации. Затем он может быть скомпилирован F# компилятором в сборку, которую можно выполнить.

Чтобы начать, откройте файл *Program. FS* , созданный ранее с помощью .NET Core CLI. Вы заметите, что в нем уже есть код.

Затем создайте новый `module` с именем `PigLatin` и скопируйте в него созданную ранее функцию `toPigLatin` :

```
module PigLatin =
    let toPigLatin (word: string) =
        let isVowel (c: char) =
            match c with
            | 'a' | 'e' | 'i' |'o' |'u'
            | 'A' | 'E' | 'I' | 'O' | 'U' -> true
            |_ -> false

        if isVowel word.[0] then
            word + "yay"
        else
            word.[1..] + string(word.[0]) + "ay"
```

Этот модуль должен быть выше `main` функции и под объявлением `open System` . Порядок объявлений имеет значение F#, поэтому необходимо определить функцию перед вызовом ее в файле.

Теперь в функции `main` вызовите функцию генератора латиницы pig для аргументов:

```
[<EntryPoint>]
let main argv =
    for name in argv do
        let newName = PigLatin.toPigLatin name
        printfn "%s in Pig Latin is: %s" name newName

    0
```

Теперь вы можете запустить консольное приложение из командной строки:

```
dotnet run apple banana
```

Вы увидите, что он выводит тот же результат, что и файл скрипта, но на этот раз в качестве выполняемой программы!

Устранение неполадок Ionide

Вот несколько способов устранения некоторых проблем, с которыми вы можете столкнуться.

1. Чтобы получить возможности редактирования кода Ionide, необходимо сохранить F# файлы на диске и в папке, открытой в Visual Studio Code рабочей области.
2. Если вы внесли изменения в систему или установили необходимые компоненты Ionide с Visual Studio Code открыть, перезапустите Visual Studio Code.
3. Если в каталогах проекта есть недопустимые символы, Ionide может не работать. Переименуйте каталоги проектов, если это так.
4. Если ни одна из команд Ionide не работает, проверьте Visual Studio Codeные привязки ключей , чтобы увидеть, не переопределяете их случайно.
5. Если Ionide не работает на компьютере и ни один из указанных выше проблем не устранил проблему, попробуйте удалить `ionide-fsharp` ный каталог на компьютере и переустановить набор подключаемых модулей.
6. Если не удалось загрузить проект (F# Обозреватель решений покажет это), щелкните правой кнопкой мыши этот проект и выберите команду **Просмотреть подробности** , чтобы получить дополнительные диагностические сведения.

Ionide — это проект с открытым исходным кодом, созданный и обслуживаемый членами F# сообщества. Сообщите о проблемах и вы можете участвовать в репозитории GitHub ionide-vscode-FSharp.

Кроме того, вы можете запросить дополнительную помощь от разработчиков Ionide и F# сообщества в канале Ionide gitter.

Следующие шаги

Дополнительные сведения о F# и функциях языка см. в статье Обзор F# .

Начало работы F# с .NET Core CLI

08.01.2020 • 3 minutes to read • Edit Online

В этой статье описывается, как можно приступить F# к работе в любой операционной системе (Windows, MacOS или Linux) с .NET Core CLI. Он проходит через создание многопроектного решения с библиотекой классов, которая вызывается консольным приложением.

Prerequisites

Для начала необходимо установить последнюю пакет SDK для .NET Core.

В этой статье предполагается, что вы умеете использовать командную строку и предпочитаемый текстовый редактор. Если вы еще не используете его, Visual Studio Code является отличным вариантом текстового редактора для F#.

Создание простого решения с несколькими проектами

Откройте командную строку или терминал и используйте команду DotNet New , чтобы создать новый файл решения с именем `FSNetCore`:

```
dotnet new sln -o FSNetCore
```

Следующая структура каталогов создается после выполнения предыдущей команды:

```
FSNetCore
    ├── FSNetCore.sln
```

Написание библиотеки классов

Перейдите в каталог *фснеткоре*.

Используйте команду `dotnet new`, создайте проект библиотеки классов в папке **src** с именем Library.

```
dotnet new classlib -lang "F#" -o src/Library
```

Следующая структура каталогов создается после выполнения предыдущей команды:

```
└── FSNetCore
    ├── FSNetCore.sln
    └── src
        └── Library
            ├── Library.fs
            └── Library.fsproj
```

Замените содержимое `Library.fs` следующим кодом:

```
module Library

open Newtonsoft.Json

let getJsonNetJson value =
    sprintf "I used to be %s but now I'm %s thanks to JSON.NET!" value (JsonConvert.SerializeObject(value))
```

Добавьте пакет NuGet Newtonsoft. JSON в проект библиотеки.

```
dotnet add src/Library/Library.fsproj package Newtonsoft.Json
```

Добавьте `Library` проект в решение `FSNetCore` с помощью команды DotNet SLN Add :

```
dotnet sln add src/Library/Library.fsproj
```

Запустите `dotnet build`, чтобы выполнить сборку проекта. При сборке будут восстановлены неразрешенные зависимости.

Написание консольного приложения, использующего библиотеку классов

С помощью команды `dotnet new` Создайте консольное приложение в папке **src** с именем App.

```
dotnet new console -lang "F#" -o src/App
```

Следующая структура каталогов создается после выполнения предыдущей команды:

```
└── FSNetCore
    ├── FSNetCore.sln
    └── src
        ├── App
        │   ├── App.fsproj
        │   ├── Program.fs
        └── Library
            ├── Library.fs
            └── Library.fsproj
```

Замените содержимое файла `Program.fs` приведенным ниже кодом.

```
open System
open Library

[<EntryPoint>]
let main argv =
    printfn "Nice command-line arguments! Here's what JSON.NET has to say about them:"

    argv
    |> Array.map getJsonNetJson
    |> Array.iter (printfn "%s")

    0 // return an integer exit code
```

Добавьте ссылку на проект `Library` с помощью DotNet Add Reference.

```
dotnet add src/App/App.fsproj reference src/Library/Library.fsproj
```

Добавьте `App` проект в решение `FSNetCore` с помощью команды `dotnet sln add`:

```
dotnet sln add src/App/App.fsproj
```

Чтобы выполнить сборку проекта, восстановите зависимости NuGet `dotnet restore` и запустите `dotnet build`.

Перейдите в каталог `src/App` проект консоли и запустите проект, передав `Hello World` в качестве аргументов:

```
cd src/App
dotnet run Hello World
```

Вы увидите следующие результаты:

```
Nice command-line arguments! Here's what JSON.NET has to say about them:

I used to be Hello but now I'm ""Hello"" thanks to JSON.NET!
I used to be World but now I'm ""World"" thanks to JSON.NET!
```

Следующие шаги

Далее ознакомьтесь с обзором F# , чтобы получить дополнительные сведения о различных F# функциях.

Начало работы с F# в Visual Studio для Mac

29.11.2019 • 9 minutes to read • Edit Online

F#и визуальные F# средства поддерживаются в интегрированной среде разработки Visual Studio для Mac. Убедитесь, что установлен Visual Studio для Mac.

Создание консольного приложения

Одним из самых основных проектов в Visual Studio для Mac является консольное приложение. Вот как это делается. После открытия Visual Studio для Mac:

1. В меню **файл** укажите пункт **создать решение**.
2. В диалоговом окне Новый проект есть два разных шаблона для консольного приложения. В другом > .NET, предназначенный для .NET Framework. Другой шаблон находится в приложении .NET Core — >, предназначенном для .NET Core. В этой статье должен работать один из шаблонов.
3. В разделе консольное приложение C# при F# необходимости измените на. Нажмите кнопку **Далее** , чтобы перейти вперед.
4. Присвойте проекту имя и выберите нужные параметры для приложения. Обратите внимание, что панель предварительного просмотра находится сбоку экрана, где отображается структура каталогов, которая будет создана на основе выбранных параметров.
5. Нажмите кнопку **Создать**. Теперь вы увидите F# проект в Обозреватель решений.

Написание кода

Приступим к работе, сначала напишем некоторый код. Убедитесь, что файл `Program.fs` открыт, и замените его содержимое следующим:

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

В предыдущем примере кода определена функция `square`, которая принимает входные данные с именем `x` и умножает ее на саму себя. Поскольку F# использует вывод типа, указывать тип `x` не нужно. F# Компилятор понимает типы, в которых умножение является допустимым, и присвоит тип `x` в зависимости от того, как вызывается `square`. При наведении указателя мыши на `square` вы должны увидеть следующее:

```
val square: x:int -> int
```

Это называется сигнатурой типа функции. Его можно прочитать следующим образом: "Square — это функция, которая принимает целое число с именем" x "и создает целое число". Обратите внимание, что компилятор выдает `square` тип `int` для Now, так как умножение не является универсальным для *всех* типов, а является универсальным по отношению к закрытому набору типов. F# Компилятор выбрал `int` на этом

этапе, но при вызове `square` с другим типом входных данных, например `float`, будет настроена сигнатура типа.

Определена другая функция `main`, которая снабжена атрибутом `EntryPoint`, чтобы сообщить F# компилятору, что должно начаться выполнение программы. Он соответствует тому же соглашению, что и другие языки программирования в стиле C, где аргументы командной строки могут передаваться в эту функцию, и возвращается целочисленный код (обычно `0`).

Эта функция вызывает функцию `square` с аргументом `12`. Затем F# компилятор назначает тип `square` `int -> int` (то есть функция, которая принимает `int` и создает `int`). Вызов `printfn` является отформатированной функцией печати, которая использует строку формата, аналогичную языку программирования в стиле C, параметры, соответствующие указанным в строке формата, а затем выводят результат и новую строку.

Выполнение кода

Можно запустить код и просмотреть результаты, щелкнув **запустить** в меню верхнего уровня, а затем **запустить без отладки**. Это приведет к запуску программы без отладки и позволяет просмотреть результаты.

Теперь в окне консоли, которое Visual Studio для Mac выводится на экран, должны отобразиться следующие сведения:

```
12 squared is 144!
```

Поздравляем! Вы создали первый F# проект в Visual Studio для Mac, записали F# функцию на печать результатов вызова этой функции и запустили проект, чтобы увидеть некоторые результаты.

Использование F# интерактивного

Одной из лучших функций визуального F# инструментария в Visual Studio для Mac является F# интерактивное окно. Он позволяет отправить код в процесс, в котором можно вызвать этот код и просмотреть результат в интерактивном режиме.

Чтобы приступить к работе, выделите функцию `square`, определенную в коде. Затем в меню верхнего уровня щелкните **Edit (изменить**). Затем выберите **Отправить выделенный F# фрагмент в интерактивный**. Это приведет к выполнению кода в F# интерактивном окне. Кроме того, можно щелкнуть выделенный фрагмент правой кнопкой мыши и выбрать пункт **Отправить выделенное в F# интерактивное**окно. Вы должны увидеть F# интерактивное окно, в котором отображается следующее:

```
>

val square : x:int -> int

>
```

Здесь показана та же сигнатура функции для функции `square`, которую вы видели ранее при наведении указателя мыши на функцию. Поскольку `square` теперь определен в F# интерактивном процессе, его можно вызвать с различными значениями:

```
> square 12;;
val it : int = 144
>square 13;;
val it : int = 169
```

Эта функция выполняет функцию, привязывает результат к новому имени `it` и отображает тип и значение `it`. Обратите внимание, что каждая строка должна завершаться `;;`. Таким способом интерактивно известно о F# завершении вызова функции. Вы также можете определить новые функции в F# интерактивном режиме:

```
> let isOdd x = x % 2 <> 0;;

val isOdd : x:int -> bool

> isOdd 12;;
val it : bool = false
```

В приведенном выше примере определяется новая функция, `isOdd`, которая принимает `int` и проверяет, нечетна ли она! Эту функцию можно вызвать, чтобы увидеть, что она возвращает с различными входными данными. Функции можно вызывать внутри вызовов функций:

```
> isOdd (square 15);;
val it : bool = true
```

Можно также использовать оператор переадресации канала для передачи значения в две функции:

```
> 15 |> square |> isOdd;;
val it : bool = true
```

Оператор переадресации канала и многое другое рассматривается в последующих руководствах.

Это лишь краткий обзор того, что можно сделать с F# помощью интерактивного. Дополнительные сведения см. в интерактивном программировании F#с помощью .

Следующие шаги

Если вы еще этого не сделали, ознакомьтесь с обзором F# , в котором рассматриваются некоторые основные функции F# языка. Вы получите общие сведения о некоторых F#возможностях и предоставьте достаточное количество примеров кода, которые вы можете скопировать в Visual Studio для Mac и выполнить. Есть также некоторые отличные внешние ресурсы, которые можно использовать, выдемонстрируемые в этом F# руководством.

См. также:

- F#программ
- Обзор языка F#
- F#Справочник по языку
- Определение типа
- Справочник по символам и операторам

Обзор F#

04.11.2019 • 39 minutes to read • Edit Online

Лучший способ узнать о F# них — читать и писать F# код. В этой статье рассматриваются некоторые основные функции F# языка и приводятся некоторые фрагменты кода, которые можно выполнять на компьютере. Чтобы узнать о настройке среды разработки, ознакомьтесь с Начало работы.

Существует два основных понятия F#: функции и типы. В этом обзоре рассматриваются функции языка, которые делятся на эти две концепции.

Исполнение кода в сети

Если на компьютере не F# установлен, можно выполнить все примеры в браузере с помощью F# Try для сборки. Фабле — это диалект F# , который выполняется непосредственно в браузере. Чтобы просмотреть примеры, приведенные в статье REPL, ознакомьтесь с **примерами, > узнать > F# обзор** в левой строке меню фабле repl.

Функции и модули

Наиболее фундаментальными частями любой F# программы являются ***функции*** , организованные в ***модули***. Функции выполняют работу с входными данными для получения выходных данных, и они упорядочены по модулям, которые являются основным способом группирования объектов F#. Они определяются с помощью привязки `let` , которая придает функции имя и определяет ее аргументы.

```
module BasicFunctions =

    /// You use 'let' to define a function. This one accepts an integer argument and returns an integer.
    /// Parentheses are optional for function arguments, except for when you use an explicit type
annotation.
    let sampleFunction1 x = x*x + 3

    /// Apply the function, naming the function return result using 'let'.
    /// The variable type is inferred from the function return type.
    let result1 = sampleFunction1 4573

    // This line uses '%d' to print the result as an integer. This is type-safe.
    // If 'result1' were not of type 'int', then the line would fail to compile.
    printfn "The result of squaring the integer 4573 and adding 3 is %d" result1

    /// When needed, annotate the type of a parameter name using '(argument:type)'.  Parentheses are
required.
    let sampleFunction2 (x:int) = 2*x*x - x/5 + 3

    let result2 = sampleFunction2 (7 + 4)
    printfn "The result of applying the 2nd sample function to (7 + 4) is %d" result2

    /// Conditionals use if/then/elif/else.
    ///
    /// Note that F# uses white space indentation-aware syntax, similar to languages like Python.
    let sampleFunction3 x =
        if x < 100.0 then
            2.0*x*x - x/5.0 + 3.0
        else
            2.0*x*x + x/5.0 - 37.0

    let result3 = sampleFunction3 (6.5 + 4.5)

    // This line uses '%f' to print the result as a float.  As with '%d' above, this is type-safe.
    printfn "The result of applying the 3rd sample function to (6.5 + 4.5) is %f" result3
```

привязки `let` также позволяют привязать значение к имени, аналогично переменной на других языках. `let` привязки являются ***неизменяемыми*** по умолчанию. Это означает, что после привязки значения или функции к имени ее нельзя изменить на месте. Это отличается от переменных в других языках, которые являются ***изменяемыми***, то есть их значения могут быть изменены в любой момент времени. Если требуется изменяемая привязка, можно использовать синтаксис `let mutable ...`.

```
module Immutability =

    /// Binding a value to a name via 'let' makes it immutable.
    ///
    /// The second line of code fails to compile because 'number' is immutable and bound.
    /// Re-defining 'number' to be a different value is not allowed in F#.
    let number = 2
    // let number = 3

    /// A mutable binding.  This is required to be able to mutate the value of 'otherNumber'.
    let mutable otherNumber = 2

    printfn "'otherNumber' is %d" otherNumber

    // When mutating a value, use '<-' to assign a new value.
    //
    // Note that '=' is not the same as this.  '=' is used to test equality.
    otherNumber <- otherNumber + 1

    printfn "'otherNumber' changed to be %d" otherNumber
```

Числа, логические значения и строки

Как язык .NET, F# поддерживает те же базовые примитивные типы , которые существуют в .NET.

Ниже показано, как представлены различные числовые типы F#в.

```
module IntegersAndNumbers =

    /// This is a sample integer.
    let sampleInteger = 176

    /// This is a sample floating point number.
    let sampleDouble = 4.1

    /// This computed a new number by some arithmetic.  Numeric types are converted using
    /// functions 'int', 'double' and so on.
    let sampleInteger2 = (sampleInteger/4 + 5 - 7) * 4 + int sampleDouble

    /// This is a list of the numbers from 0 to 99.
    let sampleNumbers = [ 0 .. 99 ]

    /// This is a list of all tuples containing all the numbers from 0 to 99 and their squares.
    let sampleTableOfSquares = [ for i in 0 .. 99 -> (i, i*i) ]

    // The next line prints a list that includes tuples, using '%A' for generic printing.
    printfn "The table of squares from 0 to 99 is:\n%A" sampleTableOfSquares
```

Ниже приведены логические значения и выполнение базовой условной логики.

```
module Booleans =

    /// Booleans values are 'true' and 'false'.
    let boolean1 = true
    let boolean2 = false

    /// Operators on booleans are 'not', '&&' and '||'.
    let boolean3 = not boolean1 && (boolean2 || false)

    // This line uses '%b'to print a boolean value.  This is type-safe.
    printfn "The expression 'not boolean1 && (boolean2 || false)' is %b" boolean3
```

И вот как выглядит базовая обработка строк :

```
module StringManipulation =

    /// Strings use double quotes.
    let string1 = "Hello"
    let string2  = "world"

    /// Strings can also use @ to create a verbatim string literal.
    /// This will ignore escape characters such as '\', '\n', '\t', etc.
    let string3 = @"C:\Program Files\"

    /// String literals can also use triple-quotes.
    let string4 = """The computer said "hello world" when I told it to!"""

    /// String concatenation is normally done with the '+' operator.
    let helloWorld = string1 + " " + string2

    // This line uses '%s' to print a string value.  This is type-safe.
    printfn "%s" helloWorld

    /// Substrings use the indexer notation.  This line extracts the first 7 characters as a substring.
    /// Note that like many languages, Strings are zero-indexed in F#.
    let substring = helloWorld.[0..6]
    printfn "%s" substring
```

Кортежи

Кортежи очень велики F#. Они представляют собой группирование неименованных, но упорядоченных значений, которые можно рассматривать как сами значения. Их следует рассматривать как значения, агрегированные из других значений. Они имеют множество применений, например, удобный возврат нескольких значений из функции или группирование значений для некоторого нерегламентированного удобства.

```
module Tuples =

    /// A simple tuple of integers.
    let tuple1 = (1, 2, 3)

    /// A function that swaps the order of two values in a tuple.
    ///
    /// F# Type Inference will automatically generalize the function to have a generic type,
    /// meaning that it will work with any type.
    let swapElems (a, b) = (b, a)

    printfn "The result of swapping (1, 2) is %A" (swapElems (1,2))

    /// A tuple consisting of an integer, a string,
    /// and a double-precision floating point number.
    let tuple2 = (1, "fred", 3.1415)

    printfn "tuple1: %A\ttuple2: %A" tuple1 tuple2
```

Начиная с F# 4,1 можно также создавать `struct` кортежи. Они также полностью взаимодействуют с кортежами C# 7/Visual Basic 15, которые также `struct` кортежи:

```
/// Tuples are normally objects, but they can also be represented as structs.
///
/// These interoperate completely with structs in C# and Visual Basic.NET; however,
/// struct tuples are not implicitly convertible with object tuples (often called reference tuples).
///
/// The second line below will fail to compile because of this.  Uncomment it to see what happens.
let sampleStructTuple = struct (1, 2)
//let thisWillNotCompile: (int*int) = struct (1, 2)

// Although you can
let convertFromStructTuple (struct(a, b)) = (a, b)
let convertToStructTuple (a, b) = struct(a, b)

printfn "Struct Tuple: %A\nReference tuple made from the Struct Tuple: %A" sampleStructTuple
(sampleStructTuple |> convertFromStructTuple)
```

Важно отметить, что поскольку `struct` кортежи являются типами значений, они не могут быть неявно преобразованы в ссылочные кортежи или наоборот. Необходимо явно выполнить преобразование между ссылкой и кортежем структуры.

Конвейеры и компоновка

Операторы конвейера, такие как `|>`, широко используются при обработке данных F#в. Эти операторы — это функции, позволяющие гибко устанавливать "конвейеры" функций. В следующем примере показано, как можно воспользоваться преимуществами этих операторов для создания простого функционального конвейера:

```
module PipelinesAndComposition =

    /// Squares a value.
    let square x = x * x

    /// Adds 1 to a value.
    let addOne x = x + 1

    /// Tests if an integer value is odd via modulo.
    let isOdd x = x % 2 <> 0

    /// A list of 5 numbers.  More on lists later.
    let numbers = [ 1; 2; 3; 4; 5 ]

    /// Given a list of integers, it filters out the even numbers,
    /// squares the resulting odds, and adds 1 to the squared odds.
    let squareOddValuesAndAddOne values =
        let odds = List.filter isOdd values
        let squares = List.map square odds
        let result = List.map addOne squares
        result

    printfn "processing %A through 'squareOddValuesAndAddOne' produces: %A" numbers
(squareOddValuesAndAddOne numbers)

    /// A shorter way to write 'squareOddValuesAndAddOne' is to nest each
    /// sub-result into the function calls themselves.
    ///
    /// This makes the function much shorter, but it's difficult to see the
    /// order in which the data is processed.
    let squareOddValuesAndAddOneNested values =
        List.map addOne (List.map square (List.filter isOdd values))

    printfn "processing %A through 'squareOddValuesAndAddOneNested' produces: %A" numbers
(squareOddValuesAndAddOneNested numbers)

    /// A preferred way to write 'squareOddValuesAndAddOne' is to use F# pipe operators.
    /// This allows you to avoid creating intermediate results, but is much more readable
    /// than nesting function calls like 'squareOddValuesAndAddOneNested'
    let squareOddValuesAndAddOnePipeline values =
        values
        |> List.filter isOdd
        |> List.map square
        |> List.map addOne

    printfn "processing %A through 'squareOddValuesAndAddOnePipeline' produces: %A" numbers
(squareOddValuesAndAddOnePipeline numbers)

    /// You can shorten 'squareOddValuesAndAddOnePipeline' by moving the second `List.map` call
    /// into the first, using a Lambda Function.
    ///
    /// Note that pipelines are also being used inside the lambda function.  F# pipe operators
    /// can be used for single values as well.  This makes them very powerful for processing data.
    let squareOddValuesAndAddOneShorterPipeline values =
        values
        |> List.filter isOdd
        |> List.map(fun x -> x |> square |> addOne)

    printfn "processing %A through 'squareOddValuesAndAddOneShorterPipeline' produces: %A" numbers
(squareOddValuesAndAddOneShorterPipeline numbers)
```

В предыдущем примере использовались многие функции F#, включая функции обработки списков, функции первого класса и частичное применение. Хотя глубокое понимание каждого из этих концепций может стать довольно сложным, должно быть ясно, насколько легко можно использовать функции для обработки данных при создании конвейеров.

Списки, массивы и последовательности

Списки, массивы и последовательности — это три основных типа коллекций в F# основной библиотеке.

Списки являются упорядоченными, неизменяемыми коллекциями элементов одного типа. Они представляют собой однонаправленные списки, что означает, что они предназначены для перечисления, но плохо подходит для произвольного доступа и объединения, если они велики. Это в отличие от списков в других популярных языках, которые обычно не используют однонаправленный список для представления списков.

```
module Lists =

    /// Lists are defined using [ ... ].  This is an empty list.
    let list1 = [ ]

    /// This is a list with 3 elements.  ';' is used to separate elements on the same line.
    let list2 = [ 1; 2; 3 ]

    /// You can also separate elements by placing them on their own lines.
    let list3 = [
        1
        2
        3
    ]

    /// This is a list of integers from 1 to 1000
    let numberList = [ 1 .. 1000 ]

    /// Lists can also be generated by computations. This is a list containing
    /// all the days of the year.
    let daysList =
        [ for month in 1 .. 12 do
              for day in 1 .. System.DateTime.DaysInMonth(2017, month) do
                  yield System.DateTime(2017, month, day) ]

    // Print the first 5 elements of 'daysList' using 'List.take'.
    printfn "The first 5 days of 2017 are: %A" (daysList |> List.take 5)

    /// Computations can include conditionals.  This is a list containing the tuples
    /// which are the coordinates of the black squares on a chess board.
    let blackSquares =
        [ for i in 0 .. 7 do
              for j in 0 .. 7 do
                  if (i+j) % 2 = 1 then
                      yield (i, j) ]

    /// Lists can be transformed using 'List.map' and other functional programming combinators.
    /// This definition produces a new list by squaring the numbers in numberList, using the pipeline
    /// operator to pass an argument to List.map.
    let squares =
        numberList
        |> List.map (fun x -> x*x)

    /// There are many other list combinations. The following computes the sum of the squares of the
    /// numbers divisible by 3.
    let sumOfSquares =
        numberList
        |> List.filter (fun x -> x % 3 = 0)
        |> List.sumBy (fun x -> x * x)

    printfn "The sum of the squares of numbers up to 1000 that are divisible by 3 is: %d" sumOfSquares
```

Массивы имеют фиксированный размер, *Изменяемые* коллекции элементов одного типа. Они поддерживают быстрый произвольный доступ к элементам и выполняются быстрее, F# чем списки, так

как они являются только непрерывными блоками памяти.

```
module Arrays =

    /// This is The empty array.  Note that the syntax is similar to that of Lists, but uses `[| ... |]`
instead.
    let array1 = [| |]

    /// Arrays are specified using the same range of constructs as lists.
    let array2 = [| "hello"; "world"; "and"; "hello"; "world"; "again" |]

    /// This is an array of numbers from 1 to 1000.
    let array3 = [| 1 .. 1000 |]

    /// This is an array containing only the words "hello" and "world".
    let array4 =
        [| for word in array2 do
               if word.Contains("l") then
                   yield word |]

    /// This is an array initialized by index and containing the even numbers from 0 to 2000.
    let evenNumbers = Array.init 1001 (fun n -> n * 2)

    /// Sub-arrays are extracted using slicing notation.
    let evenNumbersSlice = evenNumbers.[0..500]

    /// You can loop over arrays and lists using 'for' loops.
    for word in array4 do
        printfn "word: %s" word

    // You can modify the contents of an array element by using the left arrow assignment operator.
    //
    // To learn more about this operator, see: https://docs.microsoft.com/dotnet/fsharp/language-
reference/values/index#mutable-variables
    array2.[1] <- "WORLD!"

    /// You can transform arrays using 'Array.map' and other functional programming operations.
    /// The following calculates the sum of the lengths of the words that start with 'h'.
    let sumOfLengthsOfWords =
        array2
        |> Array.filter (fun x -> x.StartsWith "h")
        |> Array.sumBy (fun x -> x.Length)

    printfn "The sum of the lengths of the words in Array 2 is: %d" sumOfLengthsOfWords
```

Последовательности являются логическими последовательностями элементов одного типа. Это более общий тип, чем списки и массивы, способные «просматривать» любые логические последовательности элементов. Они также выделены, так как могут быть ***отложенными***, то есть элементы могут быть вычислены только при необходимости.

```
module Sequences =

    /// This is the empty sequence.
    let seq1 = Seq.empty

    /// This a sequence of values.
    let seq2 = seq { yield "hello"; yield "world"; yield "and"; yield "hello"; yield "world"; yield "again"
}

    /// This is an on-demand sequence from 1 to 1000.
    let numbersSeq = seq { 1 .. 1000 }

    /// This is a sequence producing the words "hello" and "world"
    let seq3 =
        seq { for word in seq2 do
                  if word.Contains("l") then
                      yield word }

    /// This sequence producing the even numbers up to 2000.
    let evenNumbers = Seq.init 1001 (fun n -> n * 2)

    let rnd = System.Random()

    /// This is an infinite sequence which is a random walk.
    /// This example uses yield! to return each element of a subsequence.
    let rec randomWalk x =
        seq { yield x
              yield! randomWalk (x + rnd.NextDouble() - 0.5) }

    /// This example shows the first 100 elements of the random walk.
    let first100ValuesOfRandomWalk =
        randomWalk 5.0
        |> Seq.truncate 100
        |> Seq.toList

    printfn "First 100 elements of a random walk: %A" first100ValuesOfRandomWalk
```

Рекурсивные функции

Обработка коллекций или последовательностей элементов обычно выполняется с помощью рекурсии в F#. Хотя F# поддерживает циклы и императивное программирование, рекурсия предпочтительнее, так как проще гарантировать правильность.

> **NOTE**
>
> В следующем примере используется сопоставление шаблонов с помощью выражения `match`. Эта фундаментальная конструкция рассматривается далее в этой статье.

```
module RecursiveFunctions =

    /// This example shows a recursive function that computes the factorial of an
    /// integer. It uses 'let rec' to define a recursive function.
    let rec factorial n =
        if n = 0 then 1 else n * factorial (n-1)

    printfn "Factorial of 6 is: %d" (factorial 6)

    /// Computes the greatest common factor of two integers.
    ///
    /// Since all of the recursive calls are tail calls,
    /// the compiler will turn the function into a loop,
    /// which improves performance and reduces memory consumption.
    let rec greatestCommonFactor a b =
        if a = 0 then b
        elif a < b then greatestCommonFactor a (b - a)
        else greatestCommonFactor (a - b) b

    printfn "The Greatest Common Factor of 300 and 620 is %d" (greatestCommonFactor 300 620)

    /// This example computes the sum of a list of integers using recursion.
    let rec sumList xs =
        match xs with
        | []    -> 0
        | y::ys -> y + sumList ys

    /// This makes 'sumList' tail recursive, using a helper function with a result accumulator.
    let rec private sumListTailRecHelper accumulator xs =
        match xs with
        | []    -> accumulator
        | y::ys -> sumListTailRecHelper (accumulator+y) ys

    /// This invokes the tail recursive helper function, providing '0' as a seed accumulator.
    /// An approach like this is common in F#.
    let sumListTailRecursive xs = sumListTailRecHelper 0 xs

    let oneThroughTen = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

    printfn "The sum 1-10 is %d" (sumListTailRecursive oneThroughTen)
```

F#также обладает полной поддержкой оптимизации вызовов с префиксом tail, что позволяет оптимизировать рекурсивные вызовы так, чтобы они были настолько же быстрыми, что и Циклическая конструкция.

Типы записей и размеченных объединений

Типы записей и объединений — это два фундаментальных типа F# данных, используемых в коде, и, как правило, это лучший F# способ представления данных в программе. Хотя это делает их похожими на классы на других языках, одно из их основных отличий состоит в том, что они имеют семантику структурного равенства. Это означает, что они являются сравнимыми по себе, и равенство достаточно просто проверить, равно ли один другой.

Записи представляют собой совокупность именованных значений с необязательными элементами (например, методами). Если вы знакомы с C# языком или Java, они должны быть ПОХОЖИ на POCO или POJO-только структурное равенство и менее формальностей.

```
module RecordTypes =

    /// This example shows how to define a new record type.
    type ContactCard =
        { Name     : string
          Phone    : string
          Verified : bool }

    /// This example shows how to instantiate a record type.
    let contact1 =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false }

    /// You can also do this on the same line with ';' separators.
    let contactOnSameLine = { Name = "Alf"; Phone = "(206) 555-0157"; Verified = false }

    /// This example shows how to use "copy-and-update" on record values. It creates
    /// a new record value that is a copy of contact1, but has different values for
    /// the 'Phone' and 'Verified' fields.
    ///
    /// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/copy-and-update-
record-expressions
    let contact2 =
        { contact1 with
            Phone = "(206) 555-0112"
            Verified = true }

    /// This example shows how to write a function that processes a record value.
    /// It converts a 'ContactCard' object to a string.
    let showContactCard (c: ContactCard) =
        c.Name + " Phone: " + c.Phone + (if not c.Verified then " (unverified)" else "")

    printfn "Alf's Contact Card: %s" (showContactCard contact1)

    /// This is an example of a Record with a member.
    type ContactCardAlternate =
        { Name     : string
          Phone    : string
          Address  : string
          Verified : bool }

        /// Members can implement object-oriented members.
        member this.PrintedContactCard =
            this.Name + " Phone: " + this.Phone + (if not this.Verified then " (unverified)" else "") +
this.Address

    let contactAlternate =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false
          Address = "111 Alf Street" }

    // Members are accessed via the '.' operator on an instantiated type.
    printfn "Alf's alternate contact card is %s" contactAlternate.PrintedContactCard
```

Начиная с F# 4,1 можно также представить записи как `struct`s. Это делается с помощью атрибута `[<Struct>]`:

```
/// Records can also be represented as structs via the 'Struct' attribute.
/// This is helpful in situations where the performance of structs outweighs
/// the flexibility of reference types.
[<Struct>]
type ContactCardStruct =
    { Name     : string
      Phone    : string
      Verified : bool }
```

Размеченные объединения (ветвью) — это значения, которые могут быть числом именованных форм или вариантов. Данные, хранящиеся в типе, могут быть одного из нескольких различных значений.

```
module DiscriminatedUnions =

    /// The following represents the suit of a playing card.
    type Suit =
        | Hearts
        | Clubs
        | Diamonds
        | Spades

    /// A Discriminated Union can also be used to represent the rank of a playing card.
    type Rank =
        /// Represents the rank of cards 2 .. 10
        | Value of int
        | Ace
        | King
        | Queen
        | Jack

        /// Discriminated Unions can also implement object-oriented members.
        static member GetAllRanks() =
            [ yield Ace
              for i in 2 .. 10 do yield Value i
              yield Jack
              yield Queen
              yield King ]

    /// This is a record type that combines a Suit and a Rank.
    /// It's common to use both Records and Discriminated Unions when representing data.
    type Card = { Suit: Suit; Rank: Rank }

    /// This computes a list representing all the cards in the deck.
    let fullDeck =
        [ for suit in [ Hearts; Diamonds; Clubs; Spades] do
              for rank in Rank.GetAllRanks() do
                  yield { Suit=suit; Rank=rank } ]

    /// This example converts a 'Card' object to a string.
    let showPlayingCard (c: Card) =
        let rankString =
            match c.Rank with
            | Ace -> "Ace"
            | King -> "King"
            | Queen -> "Queen"
            | Jack -> "Jack"
            | Value n -> string n
        let suitString =
            match c.Suit with
            | Clubs -> "clubs"
            | Diamonds -> "diamonds"
            | Spades -> "spades"
            | Hearts -> "hearts"
        rankString  + " of " + suitString

    /// This example prints all the cards in a playing deck.
    let printAllCards() =
        for card in fullDeck do
            printfn "%s" (showPlayingCard card)
```

Вы также можете использовать ветвью в качестве *размеченных объединений с одним вариантом*, чтобы упростить моделирование доменов по примитивным типам. Часто строки и другие примитивные типы используются для представления чего-либо, поэтому они задаются определенным значением. Однако использование только примитивного представления данных может привести к ошибочному назначению неверного значения! Представление каждого типа данных в виде отдельного однострочного объединения может обеспечить правильность в этом сценарии.

```
// Single-case DUs are often used for domain modeling.  This can buy you extra type safety
// over primitive types such as strings and ints.
//
// Single-case DUs cannot be implicitly converted to or from the type they wrap.
// For example, a function which takes in an Address cannot accept a string as that input,
// or vice versa.
type Address = Address of string
type Name = Name of string
type SSN = SSN of int

// You can easily instantiate a single-case DU as follows.
let address = Address "111 Alf Way"
let name = Name "Alf"
let ssn = SSN 1234567890

/// When you need the value, you can unwrap the underlying value with a simple function.
let unwrapAddress (Address a) = a
let unwrapName (Name n) = n
let unwrapSSN (SSN s) = s

// Printing single-case DUs is simple with unwrapping functions.
printfn "Address: %s, Name: %s, and SSN: %d" (address |> unwrapAddress) (name |> unwrapName) (ssn |>
unwrapSSN)
```

Как показано в приведенном выше примере, чтобы получить базовое значение в размеченного Union с одним вариантом, необходимо явно разворачивать его.

Кроме того, ветвью также поддерживает рекурсивные определения, что позволяет легко представлять деревья и по сути рекурсивные данные. Например, вот как можно представить двоичное дерево поиска с помощью функций `exists` и `insert`.

```
/// Discriminated Unions also support recursive definitions.
///
/// This represents a Binary Search Tree, with one case being the Empty tree,
/// and the other being a Node with a value and two subtrees.
type BST<'T> =
    | Empty
    | Node of value:'T * left: BST<'T> * right: BST<'T>

/// Check if an item exists in the binary search tree.
/// Searches recursively using Pattern Matching.  Returns true if it exists; otherwise, false.
let rec exists item bst =
    match bst with
    | Empty -> false
    | Node (x, left, right) ->
        if item = x then true
        elif item < x then (exists item left) // Check the left subtree.
        else (exists item right) // Check the right subtree.

/// Inserts an item in the Binary Search Tree.
/// Finds the place to insert recursively using Pattern Matching, then inserts a new node.
/// If the item is already present, it does not insert anything.
let rec insert item bst =
    match bst with
    | Empty -> Node(item, Empty, Empty)
    | Node(x, left, right) as node ->
        if item = x then node // No need to insert, it already exists; return the node.
        elif item < x then Node(x, insert item left, right) // Call into left subtree.
        else Node(x, left, insert item right) // Call into right subtree.
```

Поскольку ветвью позволяет представить рекурсивную структуру дерева в типе данных, работа над этой рекурсивной структурой проста и гарантирует правильность. Он также поддерживается в сопоставлении шаблонов, как показано ниже.

Кроме того, можно представить ветвью как `struct` s с `[<Struct>]` ным атрибутом:

```
/// Discriminated Unions can also be represented as structs via the 'Struct' attribute.
/// This is helpful in situations where the performance of structs outweighs
/// the flexibility of reference types.
///
/// However, there are two important things to know when doing this:
///     1. A struct DU cannot be recursively-defined.
///     2. A struct DU must have unique names for each of its cases.
[<Struct>]
type Shape =
    | Circle of radius: float
    | Square of side: float
    | Triangle of height: float * width: float
```

Однако при этом следует учитывать два важных момента:

1. Невозможно рекурсивно определить структуру DU.
2. Структура DU должна иметь уникальные имена для каждого из своих вариантов.

Несоблюдение приведенных выше действий приведет к ошибке компиляции.

Сопоставление шаблонов

Сопоставление шаблонов — это F# функция языка, которая обеспечивает правильность работы с F# типами. В приведенных выше примерах вы, вероятно, заметили довольно много `match x with ...` синтаксис. Эта конструкция позволяет компилятору, который может понять «форму» типов данных, принудительно учитывать все возможные варианты использования типа данных с помощью того, что известно как исчерпывающее сопоставление шаблонов. Это чрезвычайно мощный вариант для корректности, и его можно использовать для "точности" того, что обычно является проблемой времени выполнения во время компиляции.

```
module PatternMatching =

    /// A record for a person's first and last name
    type Person = {
        First : string
        Last  : string
    }

    /// A Discriminated Union of 3 different kinds of employees
    type Employee =
        | Engineer of engineer: Person
        | Manager of manager: Person * reports: List<Employee>
        | Executive of executive: Person * reports: List<Employee> * assistant: Employee

    /// Count everyone underneath the employee in the management hierarchy,
    /// including the employee. The matches bind names to the properties
    /// of the cases so that those names can be used inside the match branches.
    /// Note that the names used for binding do not need to be the same as the
    /// names given in the DU definition above.
    let rec countReports(emp : Employee) =
        1 + match emp with
            | Engineer(person) ->
                0
            | Manager(person, reports) ->
                reports |> List.sumBy countReports
            | Executive(person, reports, assistant) ->
                (reports |> List.sumBy countReports) + countReports assistant


    /// Find all managers/executives named "Dave" who do not have any reports.
    /// This uses the 'function' shorthand to as a lambda expression.
    let rec findDaveWithOpenPosition(emps : List<Employee>) =
        emps
        |> List.filter(function
                       | Manager({First = "Dave"}, []) -> true // [] matches an empty list.
                       | Executive({First = "Dave"}, [], _) -> true
                       | _ -> false) // '_' is a wildcard pattern that matches anything.
                                     // This handles the "or else" case.
```

Что вы могли заметить, это использование шаблона `_`. Это называется шаблоном с подстановочными знаками, который позволяет сказать: «я не волнует что-то, что происходит». Хотя это удобно, вы можете случайно обойти исчерпывающее сопоставление шаблонов и больше не использовать преимущества, если вы не следите за использованием `_`. Он лучше всего подходит, если вы не следите за определенными фрагментами составного типа при сопоставлении шаблонов или итоговым предложением при перечислении всех осмысленных вариантов в выражении сопоставления шаблонов.

Активные шаблоны — это еще одна мощная конструкция, используемая с сопоставлением шаблонов. Они позволяют секционировать входные данные в пользовательские формы, разбивая их на сайте вызова соответствия шаблону. Они также могут быть параметризованы, тем самым позволяя определять секцию как функцию. Расширение предыдущего примера для поддержки активных шаблонов выглядит примерно так:

```
// Active Patterns are another powerful construct to use with pattern matching.
// They allow you to partition input data into custom forms, decomposing them at the pattern match call
site.
//
// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/active-patterns
let (|Int|_|) = parseInt
let (|Double|_|) = parseDouble
let (|Date|_|) = parseDateTimeOffset
let (|TimeSpan|_|) = parseTimeSpan

/// Pattern Matching via 'function' keyword and Active Patterns often looks like this.
let printParseResult = function
    | Int x -> printfn "%d" x
    | Double x -> printfn "%f" x
    | Date d -> printfn "%s" (d.ToString())
    | TimeSpan t -> printfn "%s" (t.ToString())
    | _ -> printfn "Nothing was parse-able!"

// Call the printer with some different values to parse.
printParseResult "12"
printParseResult "12.045"
printParseResult "12/28/2016"
printParseResult "9:01PM"
printParseResult "banana!"
```

Необязательные типы

Одним из особых случаев для размеченных типов объединений является тип Option, который полезен, так как он является частью F# основной библиотеки.

Тип параметра — это тип, представляющий один из двух вариантов: значение или ничего вообще. Он используется в любом сценарии, где значение может быть или не получено из определенной операции. Тогда вы назначите вам учитывать оба варианта, сделав его проблемой во время компиляции, а не заботу среды выполнения. Они часто используются в интерфейсах API, где `null` используется для представления «Nothing», что устраняет необходимость в `NullReferenceException` во многих обстоятельствах.

```
/// Option values are any kind of value tagged with either 'Some' or 'None'.
/// They are used extensively in F# code to represent the cases where many other
/// languages would use null references.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/options
module OptionValues =

    /// First, define a zip code defined via Single-case Discriminated Union.
    type ZipCode = ZipCode of string

    /// Next, define a type where the ZipCode is optional.
    type Customer = { ZipCode: ZipCode option }

    /// Next, define an interface type that represents an object to compute the shipping zone for the
customer's zip code,
    /// given implementations for the 'getState' and 'getShippingZone' abstract methods.
    type IShippingCalculator =
        abstract GetState : ZipCode -> string option
        abstract GetShippingZone : string -> int

    /// Next, calculate a shipping zone for a customer using a calculator instance.
    /// This uses combinators in the Option module to allow a functional pipeline for
    /// transforming data with Optionals.
    let CustomerShippingZone (calculator: IShippingCalculator, customer: Customer) =
        customer.ZipCode
        |> Option.bind calculator.GetState
        |> Option.map calculator.GetShippingZone
```

Единицы измерения

Одной из F#уникальных функций системы типов является возможность предоставления контекста для числовых литералов через единицы измерения.

Единицы измерения позволяют связать числовой тип с единицей, например в метрах, а функции выполняют работу с единицами, а не с числовыми литералами. Это позволяет компилятору проверять, что типы числовых литералов имеют смысл в определенном контексте, тем самым устраняя ошибки времени выполнения, связанные с этим типом работы.

```
/// Units of measure are a way to annotate primitive numeric types in a type-safe way.
/// You can then perform type-safe arithmetic on these values.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/units-of-measure
module UnitsOfMeasure =

    /// First, open a collection of common unit names
    open Microsoft.FSharp.Data.UnitSystems.SI.UnitNames

    /// Define a unitized constant
    let sampleValue1 = 1600.0<meter>

    /// Next, define a new unit type
    [<Measure>]
    type mile =
        /// Conversion factor mile to meter.
        static member asMeter = 1609.34<meter/mile>

    /// Define a unitized constant
    let sampleValue2  = 500.0<mile>

    /// Compute  metric-system constant
    let sampleValue3 = sampleValue2 * mile.asMeter

    // Values using Units of Measure can be used just like the primitive numeric type for things like
printing.
    printfn "After a %f race I would walk %f miles which would be %f meters" sampleValue1 sampleValue2
sampleValue3
```

F# Базовая библиотека определяет множество типов единиц СИ и преобразования единиц. Чтобы узнать больше, ознакомьтесь с пространством имен Microsoft.FSharp.Data.UnitSystems.Si.

Классы и интерфейсы

F#также обладает полной поддержкой классов .NET, интерфейсов, абстрактных классов, наследованияи т. д.

Классы — это типы, представляющие объекты .NET, которые могут иметь свойства, методы и события в качестве членов.

```
/// Classes are a way of defining new object types in F#, and support standard Object-oriented constructs.
/// They can have a variety of members (methods, properties, events, etc.)
///
/// To learn more about Classes, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/classes
///
/// To learn more about Members, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/members
module DefiningClasses =

    /// A simple two-dimensional Vector class.
    ///
    /// The class's constructor is on the first line,
    /// and takes two arguments: dx and dy, both of type 'double'.
    type Vector2D(dx : double, dy : double) =

        /// This internal field stores the length of the vector, computed when the
        /// object is constructed
        let length = sqrt (dx*dx + dy*dy)

        // 'this' specifies a name for the object's self-identifier.
        // In instance methods, it must appear before the member name.
        member this.DX = dx

        member this.DY = dy

        member this.Length = length

        /// This member is a method.  The previous members were properties.
        member this.Scale(k) = Vector2D(k * this.DX, k * this.DY)

    /// This is how you instantiate the Vector2D class.
    let vector1 = Vector2D(3.0, 4.0)

    /// Get a new scaled vector object, without modifying the original object.
    let vector2 = vector1.Scale(10.0)

    printfn "Length of vector1: %f\nLength of vector2: %f" vector1.Length vector2.Length
```

Определение универсальных классов также очень просто.

```
/// Generic classes allow types to be defined with respect to a set of type parameters.
/// In the following, 'T is the type parameter for the class.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/generics/
module DefiningGenericClasses =

    type StateTracker<'T>(initialElement: 'T) =

        /// This internal field store the states in a list.
        let mutable states = [ initialElement ]

        /// Add a new element to the list of states.
        member this.UpdateState newState =
            states <- newState :: states  // use the '<-' operator to mutate the value.

        /// Get the entire list of historical states.
        member this.History = states

        /// Get the latest state.
        member this.Current = states.Head

    /// An 'int' instance of the state tracker class. Note that the type parameter is inferred.
    let tracker = StateTracker 10

    // Add a state
    tracker.UpdateState 17
```

Для реализации интерфейса можно использовать либо синтаксис `interface ... with`, либо выражение объекта.

```
/// Interfaces are object types with only 'abstract' members.
/// Object types and object expressions can implement interfaces.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/interfaces
module ImplementingInterfaces =

    /// This is a type that implements IDisposable.
    type ReadFile() =

        let file = new System.IO.StreamReader("readme.txt")

        member this.ReadLine() = file.ReadLine()

        // This is the implementation of IDisposable members.
        interface System.IDisposable with
            member this.Dispose() = file.Close()


    /// This is an object that implements IDisposable via an Object Expression
    /// Unlike other languages such as C# or Java, a new type definition is not needed
    /// to implement an interface.
    let interfaceImplementation =
        { new System.IDisposable with
            member this.Dispose() = printfn "disposed" }
```

Какие типы использовать

Наличие классов, записей, размеченных объединений и кортежей приводит к важному вопросу: что следует использовать? Как и многие все это в жизни, ответ зависит от обстоятельств.

Кортежи отлично подходят для возвращения нескольких значений из функции и использования специального статистического выражения значений в качестве самого значения.

Записи — это «шаг с заходом» из кортежей, имеющих именованные метки и поддержку необязательных элементов. Они отлично подходят для формальностей представления данных во время передачи по программе. Так как они имеют структурное равенство, их легко использовать при сравнении.

Размеченные объединения имеют много использования, но основное преимущество заключается в том, чтобы их можно было использовать в сочетании с сопоставлением шаблонов для учета всех возможных "фигур", которые могут иметь данные.

Классы отлично подходят для огромного числа причин, например, когда необходимо представить информацию, а также привязать эту информацию к функциональным возможностям. Как правило, при наличии функциональности, которая концептуально привязана к некоторым данным, использование классов и принципов объектно-ориентированного программирования является большим преимуществом. Классы также являются предпочтительным типом данных при взаимодействии с C# и Visual Basic, так как эти языки используют классы практически для всех.

Следующие шаги

Теперь, когда вы видели некоторые основные возможности языка, вы должны быть готовы к написанию первых F# программ! Ознакомьтесь Начало работы , чтобы узнать, как настроить среду разработки и написать код.

Дальнейшие действия для изучения могут быть любыми, но мы рекомендуем ознакомиться с функциональным программированием в F# , чтобы приступить к работе с основными понятиями функционального программирования. Они необходимы для создания надежных программ в F#.

Кроме того, ознакомьтесь с F# справочником по языку , чтобы просмотреть исчерпывающую коллекцию F#концептуальных материалов по.

Введение в функциональное программирование в F#

04.11.2019 • 14 minutes to read • Edit Online

Функциональное программирование — это стиль программирования, который подчеркивает использование функций и неизменяемых данных. Типизированное функциональное программирование — это то, что функциональное программирование сочетается со F#статическими типами, например с. Как правило, в функциональном программировании используются следующие понятия:

- Функции в качестве основных используемых конструкций
- Выражения вместо операторов
- Неизменяемые значения для переменных
- Декларативное программирование по императивному программированию

В этой серии вы ознакомитесь с основными понятиями и шаблонами в F#функциональном программировании с помощью. Кроме того, вы также узнаете F# об этом.

Терминология

Функциональное программирование, как и другие парадигмы программирования, сопровождается словарем, который со временем потребуется изучить. Ниже приведены некоторые распространенные термины, которые можно увидеть все время:

- **Function** — функция — это конструкция, которая выдает выходные данные при наличии входных данных. Более формально он *сопоставляет* элемент из одного набора с другим набором. Этот формальный метод ликвидируется в конкретную часть, особенно при использовании функций, которые работают с коллекциями данных. Это наиболее простое (и важное) понятие в функциональном программировании.
- **Expression** — выражение — это конструкция в коде, которая создает значение. В F#это значение должно быть привязано или явно игнорироваться. Выражение может быть тривиально заменено вызовом функции.
- **Чистота** — это свойство функции таким, что возвращаемое значение всегда одинаково для одних и тех же аргументов, и его оценка не имеет побочных эффектов. Чистая функция полностью зависит от своих аргументов.
- **Ссылочная** прозрачность — это свойство выражений таким образом, что их можно заменить на выходные данные, не влияя на поведение программы.
- **Неизменность** — это значение не может быть изменено на месте. Это отличается от переменных, которые могут измениться на месте.

Примеры

В следующих примерах демонстрируются эти основные понятия.

Функции

Наиболее распространенной и фундаментальной конструкцией функционального программирования является функция. Вот простая функция, которая добавляет 1 к целому числу:

```
let addOne x = x + 1
```

Сигнатура его типа выглядит следующим образом:

```
val addOne: x:int -> int
```

Сигнатура может считаться "`addOne` принимает `int` с именем `x` и создает `int`". Более формально, `addOne` *сопоставляет* значение из набора целых чисел с набором целых чисел. Маркер `->` обозначает это сопоставление. В F#можно просмотреть сигнатуру функции, чтобы понять, что она делает.

Итак, почему сигнатура важна? В типизированном функциональном программировании реализация функции часто менее важна, чем фактическая сигнатура типа! Тот факт, что `addOne` добавляет значение 1 к целому числу, интересно во время выполнения, но при создании программы тот факт, что он принимает и возвращает `int`, — это то, что именно будет использоваться для этой функции. Кроме того, после правильного использования этой функции (по отношению к сигнатуре типа) Диагностика проблем может быть выполнена только в теле функции `addOne`. Это стимула за типизированное функциональное программирование.

Выражения

Выражения — это конструкции, результатом вычисления которых является значение. В отличие от операторов, выполняющих действие, выражения можно считать выполнением действия, которое возвращает значение. Выражения почти всегда используются в пользу операторов в функциональном программировании.

Рассмотрим предыдущую функцию, `addOne`. Тело `addOne` является выражением:

```
// 'x + 1' is an expression!
let addOne x = x + 1
```

Это результат выражения, определяющего тип результата функции `addOne`. Например, выражение, составляющее эту функцию, может быть изменено на другой тип, например `string`:

```
let addOne x = x.ToString() + "1"
```

Теперь сигнатура функции:

```
val addOne: x:'a -> string
```

Поскольку для любого типа F# в может быть вызван `ToString()`, тип `x` был сделан универсальным (называемым автоматической обобщением), а результирующий тип — `string`.

Выражения — это не только тела функций. Можно использовать выражения, которые возвращают значение, которое используется в других местах. Общий `if`:

```
// Checks if 'x' is odd by using the mod operator
let isOdd x = x % 2 <> 0

let addOneIfOdd input =
    let result =
        if isOdd input then
            input + 1
        else
            input

    result
```

`if` выражение создает значение с именем `result`. Обратите внимание, что можно полностью опустить `result`, сделав `if` выражение телом функции `addOneIfOdd`. Ключевым моментом, который следует помнить о выражениях, является то, что они создают значение.

Существует специальный тип, `unit`, который используется при отсутствии возвращаемых результатов. Например, рассмотрим эту простую функцию:

```
let printString (str: string) =
    printfn "String is: %s" str
```

Подпись выглядит следующим образом:

```
val printString: str:string -> unit
```

Тип `unit` указывает, что фактическое значение не возвращается. Это полезно, если у вас есть подпрограммы, которые должны «работать», несмотря на отсутствие значения, возвращаемого в результате этой работы.

Это очень четкое отличие от императивного программирования, где эквивалентная `if` конструкция является оператором, а создание значений часто осуществляется с помощью изменения переменных. Например, в C#код может быть написан следующим образом:

```
bool IsOdd(int x) => x % 2 != 0;

int AddOneIfOdd(int input)
{
    var result = input;

    if (IsOdd(input))
    {
        result = input + 1;
    }

    return result;
}
```

Стоит отметить, что C# и другие языки в стиле C поддерживают выражение ternary, которое обеспечивает условное программирование на основе выражений.

В функциональном программировании редко изменяются значения с помощью операторов. Хотя некоторые функциональные языки поддерживают инструкции и изменения, использование этих концепций в функциональном программировании не является распространенным.

Чистые функции

Как упоминалось ранее, чистые функции — это функции, которые:

- Всегда вычисляют одно и то же значение для одних и тех же входных данных.
- Не имеют побочных эффектов.

В этом контексте удобнее рассматривать математические функции. В математике функции зависят только от своих аргументов и не имеют побочных эффектов. В математической функции `f(x) = x + 1` значение `f(x)` зависит только от значения `x`. Чистые функции функционального программирования одинаковы.

При написании чистой функции функция должна зависеть только от своих аргументов и не выполнять никаких действий, которые приводят к побочному результату.

Ниже приведен пример функции, не являющейся чистой, так как она зависит от глобального изменяемого

состояния:

```
let mutable value = 1

let addOneToValue x = x + value
```

Функция `addOneToValue` очевидной, поскольку `value` может быть изменена в любое время, чтобы иметь другое значение, отличное от 1. Этот шаблон в зависимости от глобального значения следует избегать в функциональном программировании.

Ниже приведен еще один пример функции, не являющейся чистой, поскольку она выполняет побочный результат:

```
let addOneToValue x =
    printfn "x is %d" x
    x + 1
```

Хотя эта функция не зависит от глобального значения, она записывает значение `x` в выходные данные программы. Хотя в этом нет ничего плохого, это означает, что функция не является чистой. Если другая часть программы зависит от объекта, внешнего для программы, например выходного буфера, то вызов этой функции может повлиять на другую часть программы.

Удаление оператора `printfn` делает функцию чистой:

```
let addOneToValue x = x + 1
```

Несмотря на то что эта функция не является *более лучшей* по сравнению с предыдущей версией с помощью оператора `printfn`, она гарантирует, что вся эта функция возвращает значение. При вызове этой функции любое количество раз дает тот же результат: он просто создает значение. Предсказуемость, предоставляемая чистотой, — это что-то множество функциональных программистов, которые стремится.

Неизменяемости

Наконец, одна из самых фундаментальных концепций типизированного функционального программирования — неизменяемость. В F#все значения по умолчанию являются неизменяемыми. Это означает, что их нельзя изменить на месте, если явно не помечать их как изменяемые.

На практике работа с неизменяемыми значениями означает, что вы меняете подход на программирование с «мне нужно изменить что-нибудь» на «мне нужно получить новое значение».

Например, добавление 1 к значению означает создание нового значения, а не изменение существующего.

```
let value = 1
let secondValue = value + 1
```

В F#следующий код **не** изменяет функцию `value`. Вместо этого он выполняет проверку на равенство:

```
let value = 1
value = value + 1 // Produces a 'bool' value!
```

Некоторые языки функционального программирования не поддерживают изменения вообще. В F#поддерживается, но не является поведением по умолчанию для значений.

Эта концепция расширяется еще до структур данных. В функциональном программировании неизменяемые структуры данных, такие как наборы (и многие другие), имеют разную реализацию, чем может быть

изначально ожидать. По сути, что-то вроде добавления элемента в набор не изменяет набор, он создает *Новый* набор с добавленным значением. На самом деле это часто достигается с помощью другой структуры данных, которая позволяет эффективно отслеживать значение, чтобы в результате можно было получить соответствующее представление данных.

Этот стиль работы со значениями и структурами данных является критически важным, так как он заставляет обрабатывать любые операции, которые изменяют что-то, как при создании новой версии. Это позволяет обеспечить единообразие в программах, таких как равенство и сравнение.

Следующие шаги

В следующем разделе будут тщательно рассмотрены функции, а также различные способы их использования в функциональном программировании.

Функции первого класса анализируют функции глубоко, показывая, как их можно использовать в различных контекстах.

Дополнительные сведения

Помышление функционального ряда — еще один отличный ресурс для изучения функционального F#программирования с помощью. В нем рассматриваются основы функционального программирования на практичном и удобном для чтения виде, с использованием F# функций для иллюстрации концепций.

Функции первого класса

23.10.2019 • 29 minutes to read • Edit Online

Определение характеристик языков функционального программирования — это повышение уровня функций до состояния первого класса. Вы должны иметь возможность выполнять функции, которые можно использовать со значениями других встроенных типов, и иметь возможность сделать это с сравнимой степенью усилий.

К типичным мерам состояния первого класса относятся следующие:

- Можно ли привязать функции к идентификаторам? То есть можно ли присвоить им имена?
- Можно ли хранить функции в структурах данных, например в списке?
- Можно ли передать функцию в качестве аргумента в вызове функции?
- Можно ли вернуть функцию из вызова функции?

Последние две меры определяют, что называются *операциями высшего порядка* или *функциями высшего порядка*. Функции высшего порядка принимают функции в качестве аргументов и возвращают функции в качестве значений вызовов функций. Эти операции поддерживают такие основ функционального программирования, как функции сопоставления и компоновка функций.

Присвоить значение имени

Если функция является значением первого класса, необходимо иметь возможность присвоить ей имя, точно так же, как целые числа, строки и другие встроенные типы. В литературе по функциональному программированию это называется привязкой идентификатора к значению. F# `let <identifier> = <value>` использует `let` привязки для привязки имен к значениям:. В следующем коде показаны два примера.

```
// Integer and string.
let num = 10
let str = "F#"
```

Вы можете легко присвоить имя функции. В следующем примере определяется функция с именем `squareIt` путем привязки идентификатора `squareIt` к лямбда-выражению `fun n -> n * n`. Функция `squareIt` имеет один параметр, `n` и возвращает квадрат этого параметра.

```
let squareIt = fun n -> n * n
```

F#предоставляет следующий более краткий синтаксис для достижения того же результата с меньшей типизацией.

```
let squareIt2 n = n * n
```

В приведенных ниже примерах используется первый стиль, `let <function-name> = <lambda-expression>` чтобы подчеркнуть сходство между объявлением функций и объявлением других типов значений. Однако все именованные функции также можно записать с помощью краткого синтаксиса. Некоторые примеры написаны обоими способами.

Сохранение значения в структуре данных

Значение первого класса может храниться в структуре данных. В следующем коде показаны примеры хранения значений в списках и кортежах.

```
// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator.
let BMICalculator = fun ht wt ->
                    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]


// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )
```

Чтобы убедиться, что имя функции, сохраненное в кортеже, фактически выполняет вычисление функции, в следующем примере используются `fst` операторы `snd` и для извлечения первого и второго элементов из кортежа `funAndArgTuple`. Первый элемент в кортеже имеет `squareIt` значение, а второй элемент —. `num` Идентификатор `num` привязывается в предыдущем примере к целому 10, допустимому аргументу `squareIt` для функции. Второе выражение применяет первый элемент кортежа ко второму элементу в кортеже: `squareIt num`.

```
// You can pull a function out of a tuple and apply it. Both squareIt and num
// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))
```

Точно так же, как `num` идентификатор и целое число 10, можно использовать взаимозаменяемость, `squareIt` поэтому может быть `fun n -> n * n` идентификатором и лямбда-выражением.

```
// Make a tuple of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))
```

Передать значение в качестве аргумента

Если значение имеет состояние первого класса в языке, можно передать его в качестве аргумента функции. Например, в качестве аргументов часто передаются целые числа и строки. В следующем коде показаны целые числа и строки, передаваемые в F#качестве аргументов в.

```
// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)
```

Если функции имеют состояние первого класса, необходимо иметь возможность передавать их как аргументы аналогичным образом. Помните, что это первая характеристика функций высшего порядка.

В следующем примере функция `applyIt` имеет два параметра: `op` и `arg`. Если вы отправляете в функцию, имеющую один параметр `op` для `arg`, и соответствующий аргумент функции, функция возвращает результат применения `op` функции к `arg`. В следующем примере аргументы функции и целочисленный аргумент отправляются одинаковым образом с использованием их имен.

```
// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)
```

Возможность отправки функции в качестве аргумента в другую функцию зависит от общих абстракций в языках функционального программирования, таких как операции Map или Filter. Операция Map, например, — это функция более высокого порядка, которая захватывает вычисления, совместно используемые функциями, которые пошагово проходят по списку, выполняют какие-либо действия с каждым элементом, а затем возвращают список результатов. Может потребоваться увеличить каждый элемент в списке целых чисел или квадратный каждый элемент или изменить каждый элемент в списке строк на верхний. Подверженная ошибкам часть вычислений является рекурсивным процессом, который проходит по списку и создает список возвращаемых результатов. Эта часть захватывается функцией сопоставления. Все, что необходимо написать для конкретного приложения, — это функция, которую необходимо применить к каждому элементу списка по отдельности (Добавление, возведение в изменяемый регистр). Эта функция отправляется в качестве аргумента функции сопоставления, так же как `squareIt` `applyIt` и в предыдущем примере.

F#предоставляет методы Map для большинства типов коллекций, включая списки, массивыи последовательности. В следующих примерах используются списки. Синтаксис: `List.map <the function> <the list>`.

```
// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList

// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot
```

Дополнительные сведения см. в разделе списки.

Возврат значения из вызова функции

Наконец, если функция имеет состояние первого класса в языке, необходимо иметь возможность возвращать его как значение вызова функции, как и другие типы, такие как целые числа и строки.

Следующие вызовы функции возвращают целые числа и отображают их.

```
// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)
```

Следующий вызов функции возвращает строку.

```
// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()
```

Следующий вызов функции, объявленный встроенным, возвращает логическое значение. Отображаемое значение — `True`.

```
System.Console.WriteLine((fun n -> n % 2 = 1) 15)
```

Возможность возвращать функцию в качестве значения вызова функции — это вторая характеристика функций более высокого порядка. В следующем примере `checkFor` определяется как функция, принимающая один аргумент, `item` и возвращающая в качестве значения новую функцию. Возвращаемая функция принимает список `lst` `item` в качестве аргумента, и выполняет поиск в `lst`. Если `item` имеется, функция возвращает `true`. Если `item` параметр не указан, функция возвращает `false` значение. Как и в предыдущем разделе, следующий код использует предоставленную функцию списка List. Existsдля поиска в списке.

```
let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn
```

Следующий код использует `checkFor` для создания новой функции, которая принимает один аргумент, список и выполняет поиск 7 в списке.

```
// integerList and stringList were defined earlier.
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
//let stringList = [ "one"; "two"; "three" ]

// The returned function is given the name checkFor7.
let checkFor7 = checkFor 7

// The result displayed when checkFor7 is applied to integerList is True.
System.Console.WriteLine(checkFor7 integerList)

// The following code repeats the process for "seven" in stringList.
let checkForSeven = checkFor "seven"

// The result displayed is False.
System.Console.WriteLine(checkForSeven stringList)
```

В следующем примере используется состояние первого класса функций в F# для объявления функции `compose`, которая возвращает композицию двух аргументов функции.

```
// Function compose takes two arguments. Each argument is a function
// that takes one argument of the same type. The following declaration
// uses lambda expresson syntax.
let compose =
    fun op1 op2 ->
        fun n ->
            op1 (op2 n)

// To clarify what you are returning, use a nested let expression:
let compose2 =
    fun op1 op2 ->
        // Use a let expression to build the function that will be returned.
        let funToReturn = fun n ->
                            op1 (op2 n)
        // Then just return it.
        funToReturn

// Or, integrating the more concise syntax:
let compose3 op1 op2 =
    let funToReturn = fun n ->
                        op1 (op2 n)
    funToReturn
```

NOTE

Более короткая версия см. в следующем разделе "каррированных функции".

Следующий код отправляет две функции в качестве аргументов в `compose`, оба из которых принимают один аргумент того же типа. Возвращаемое значение — это новая функция, которая представляет собой композицию двух аргументов функции.

```
// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)
```

NOTE

F#предоставляет два оператора, `<<` и `>>`, которые состоят из функций. Например, `let squareAndDouble2 = doubleIt << squareIt` `let squareAndDouble = compose doubleIt squareIt` эквивалентен в предыдущем примере.

Следующий пример возврата функции в качестве значения вызова функции создает простую игру подбора. Чтобы создать игру, вызовите `makeGame` метод с тем значением, `target` в котором вы хотите угадать. Возвращаемое значение функции `makeGame` — это функция, которая принимает один аргумент (Guess) и сообщает, верно ли предположение.

```
let makeGame target =
    // Build a lambda expression that is the function that plays the game.
    let game = fun guess ->
                   if guess = target then
                      System.Console.WriteLine("You win!")
                   else
                      System.Console.WriteLine("Wrong. Try again.")
    // Now just return it.
    game
```

Следующий код вызывает метод `makeGame`, отправляя значение `7` для `target`. Идентификатор `playGame` привязан к возвращенному лямбда-выражению. Таким образом `playGame` , представляет собой функцию, принимающую в качестве одного аргумента `guess` значение для.

```
let playGame = makeGame 7
// Send in some guesses.
playGame 2
playGame 9
playGame 7

// Output:
// Wrong. Try again.
// Wrong. Try again.
// You win!

// The following game specifies a character instead of an integer for target.
let alphaGame = makeGame 'q'
alphaGame 'c'
alphaGame 'r'
alphaGame 'j'
alphaGame 'q'

// Output:
// Wrong. Try again.
// Wrong. Try again.
// Wrong. Try again.
// You win!
```

Каррированных функции

Многие примеры в предыдущем разделе можно написать более кратко, используя преимущества неявного *карринг* в F# объявлениях функций. Карринг — это процесс, преобразуя функцию, имеющую несколько параметров, в ряд встроенных функций, каждый из которых имеет один параметр. В F#функции, имеющие более одного параметра, по сути являются каррированных. Например, `compose` из предыдущего раздела можно написать, как показано в следующем кратком стиле с тремя параметрами.

```
let compose4 op1 op2 n = op1 (op2 n)
```

Однако результатом является функция одного параметра, возвращающая функцию одного параметра, которая, в свою очередь, возвращает другую функцию одного параметра, как показано в `compose4curried`.

```
let compose4curried =
    fun op1 ->
        fun op2 ->
            fun n -> op1 (op2 n)
```

Доступ к этой функции можно получить несколькими способами. Каждый из следующих примеров

возвращает и отображает 18. В любом из `compose4` примеров `compose4curried` можно заменить на.

```
// Access one layer at a time.
System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)

// Access as in the original compose examples, sending arguments for
// op1 and op2, then applying the resulting function to a value.
System.Console.WriteLine((compose4 doubleIt squareIt) 3)

// Access by sending all three arguments at the same time.
System.Console.WriteLine(compose4 doubleIt squareIt 3)
```

Чтобы убедиться, что функция все еще работает, как и раньше, попробуйте повторить исходные тестовые случаи.

```
let doubleAndSquare4 = compose4 squareIt doubleIt
// The following expression returns and displays 36.
System.Console.WriteLine(doubleAndSquare4 3)

let squareAndDouble4 = compose4 doubleIt squareIt
// The following expression returns and displays 18.
System.Console.WriteLine(squareAndDouble4 3)
```

> **NOTE**
> Можно ограничить карринг, заключив параметры в кортежи. Дополнительные сведения см. в разделе "Шаблоны параметров" раздела Параметры и аргументы.

В следующем примере используется неявное карринг для записи более короткой версии `makeGame`. Сведения о том, `makeGame` как конструкции и `game` возвращают функцию, менее явны в этом формате, но можно проверить с помощью исходных тестовых случаев, в которых результат совпадает.

```
let makeGame2 target guess =
    if guess = target then
       System.Console.WriteLine("You win!")
    else
       System.Console.WriteLine("Wrong. Try again.")

let playGame2 = makeGame2 7
playGame2 2
playGame2 9
playGame2 7

let alphaGame2 = makeGame2 'q'
alphaGame2 'c'
alphaGame2 'r'
alphaGame2 'j'
alphaGame2 'q'
```

Дополнительные сведения о карринг см. в разделе "частичное применение аргументов" в функциях.

Идентификатор и определение функции взаимозаменяемы

Имя `num` переменной в предыдущих примерах вычисляется как целое число 10, и это не удивительно, `num` что допустимо, 10 также является допустимым. То же самое касается идентификаторов функций и их значений: где можно использовать имя функции, можно использовать лямбда-выражение, к которому она привязана.

В следующем примере определяется `Boolean` функция с именем `isNegative`, а затем используется имя функции и определение взаимозаменяемой функции. В следующих трех примерах возвращаются и отображаются `False`.

```
let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)
```

Чтобы сделать это на один шаг дальше, замените значение, `applyIt` привязанное к для `applyIt`.

```
System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0)  10)
```

Функции являются значениями первого класса в F#

В примерах, приведенных в предыдущих разделах, F# показано, что функции в удовлетворяют критериям для использования F#значений первого класса в:

- Идентификатор можно привязать к определению функции.

```
let squareIt = fun n -> n * n
```

- Функцию можно сохранить в структуре данных.

```
let funTuple2 = ( BMICalculator, fun n -> n * n )
```

- Функцию можно передать в качестве аргумента.

```
let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]
```

- Функцию можно вернуть в качестве значения вызова функции.

```
let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn
```

Дополнительные сведения о F#см. в справочнике по F# языку.

Пример

Описание

Следующий код содержит все примеры, приведенные в этом разделе.

Код

```
// ** GIVE THE VALUE A NAME **
```

```
// Integer and string.
let num = 10
let str = "F#"



let squareIt = fun n -> n * n



let squareIt2 n = n * n



// ** STORE THE VALUE IN A DATA STRUCTURE **


// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator.
let BMICalculator = fun ht wt ->
                    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]


// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )



// You can pull a function out of a tuple and apply it. Both squareIt and num
```

```
// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))



// Make a list of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))



// ** PASS THE VALUE AS AN ARGUMENT **


// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)



// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)



// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList
```

```
// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot



// ** RETURN THE VALUE FROM A FUNCTION CALL **


// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)


// The following function call returns a string:

// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()



System.Console.WriteLine((fun n -> n % 2 = 1) 15)



let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn



// integerList and stringList were defined earlier.
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
//let stringList = [ "one"; "two"; "three" ]

// The returned function is given the name checkFor7.
let checkFor7 = checkFor 7

// The result displayed when checkFor7 is applied to integerList is True.
System.Console.WriteLine(checkFor7 integerList)

// The following code repeats the process for "seven" in stringList.
let checkForSeven = checkFor "seven"

// The result displayed is False.
System.Console.WriteLine(checkForSeven stringList)



// Function compose takes two arguments. Each argument is a function
// that takes one argument of the same type. The following declaration
// uses lambda expresson syntax.
let compose =
    fun op1 op2 ->
        fun n ->
            op1 (op2 n)

// To clarify what you are returning, use a nested let expression:
let compose2 =
    fun op1 op2 ->
        // Use a let expression to build the function that will be returned.
        let funToReturn = fun n ->
                          op1 (op2 n)
        // Then just return it.
        funToReturn

// Or, integrating the more concise syntax:
```

```
// Or, integrating the more concise syntax:
let compose3 op1 op2 =
    let funToReturn = fun n ->
                        op1 (op2 n)
    funToReturn



// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)



let makeGame target =
    // Build a lambda expression that is the function that plays the game.
    let game = fun guess ->
                   if guess = target then
                      System.Console.WriteLine("You win!")
                   else
                      System.Console.WriteLine("Wrong. Try again.")
    // Now just return it.
    game



let playGame = makeGame 7
// Send in some guesses.
playGame 2
playGame 9
playGame 7

// Output:
// Wrong. Try again.
// Wrong. Try again.
// You win!

// The following game specifies a character instead of an integer for target.
let alphaGame = makeGame 'q'
alphaGame 'c'
alphaGame 'r'
alphaGame 'j'
alphaGame 'q'

// Output:
// Wrong. Try again.
// Wrong. Try again.
// Wrong. Try again.
// You win!



// ** CURRIED FUNCTIONS **


let compose4 op1 op2 n = op1 (op2 n)



let compose4curried =
    fun op1 ->
        fun op2 ->
```

```
            fun n -> op1 (op2 n)



// Access one layer at a time.
System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)

// Access as in the original compose examples, sending arguments for
// op1 and op2, then applying the resulting function to a value.
System.Console.WriteLine((compose4 doubleIt squareIt) 3)

// Access by sending all three arguments at the same time.
System.Console.WriteLine(compose4 doubleIt squareIt 3)



let doubleAndSquare4 = compose4 squareIt doubleIt
// The following expression returns and displays 36.
System.Console.WriteLine(doubleAndSquare4 3)

let squareAndDouble4 = compose4 doubleIt squareIt
// The following expression returns and displays 18.
System.Console.WriteLine(squareAndDouble4 3)



let makeGame2 target guess =
    if guess = target then
       System.Console.WriteLine("You win!")
    else
       System.Console.WriteLine("Wrong. Try again.")

let playGame2 = makeGame2 7
playGame2 2
playGame2 9
playGame2 7

let alphaGame2 = makeGame2 'q'
alphaGame2 'c'
alphaGame2 'r'
alphaGame2 'j'
alphaGame2 'q'



// ** IDENTIFIER AND FUNCTION DEFINITION ARE INTERCHANGEABLE **


let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)



System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0)  10)



// ** FUNCTIONS ARE FIRST-CLASS VALUES IN F# **

//let squareIt = fun n -> n * n
```

```
let funTuple2 = ( BMICalculator, fun n -> n * n )



let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]


//let checkFor item =
//    let functionToReturn = fun lst ->
//                           List.exists (fun a -> a = item) lst
//    functionToReturn
```

См. также

- Списки
- Кортежи
- Функции
- `let` Привязки
- Лямбда-выражения: Ключевое слово `fun`

Асинхронное программирование в F#

08.01.2020 • 22 minutes to read • Edit Online

Асинхронное программирование — это механизм, который необходим для современных приложений по различным причинам. Большинство разработчиков могут столкнуться с двумя основными вариантами использования:

- Представление серверного процесса, который может обслуживать значительное количество одновременных входящих запросов, одновременно сокращая объем системных ресурсов, занятых во время обработки запроса, ожидающих входные данные из систем или служб извне этого процесса.
- Поддержание реагирующего пользовательского интерфейса или основного потока при одновременном выполнении фоновой работы

Хотя фоновая работа часто занимается использованием нескольких потоков, важно учитывать концепцию Асинхронность и многопоточности отдельно. На самом деле, это отдельные проблемы, и одна из них не подразумевает другую. Далее в этой статье подробно описывается это.

Определено асинхронность

Предыдущая точка — то, что асинхронность не зависит от использования нескольких потоков, стоит немного подробнее объяснить. Существуют три концепции, которые иногда связаны, но строго независимы друг от друга:

- Параллелизма когда несколько вычислений выполняются в перекрывающиеся периоды времени.
- Параллелизма Если несколько вычислений или несколько частей одного вычисления выполняются в точно одно и то же время.
- Асинхронность Если одно или несколько вычислений могут выполняться отдельно от основного потока программы.

Все три являются ортогональными концепциями, но их можно легко увеличить, особенно если они используются совместно. Например, может потребоваться параллельное выполнение нескольких асинхронных вычислений. Это не означает, что параллелизм или асинхронность предполагают друг друга.

Если вы считаете, что етимологи слово «асинхронный», участвуют две части:

- "a", что означает "not".
- "Синхронное", означающее "в то же время".

При совместном помещении этих двух терминов вы увидите, что "асинхронное" означает "не одновременно". Вот и все! В этом определении не происходит неарифметического параллелизма или параллелизма. Это также справедливо и на практике.

На практике асинхронные вычисления в F# запланированы на выполнение независимо от основной последовательности программы. Это не подразумевает параллелизм или параллелизм, а также не подразумевает, что вычисления всегда выполняются в фоновом режиме. Фактически асинхронные вычисления могут даже выполняться синхронно, в зависимости от природы вычислений и среды, в которой выполняется вычисление.

Основная мысльа заключается в том, что асинхронные вычисления не зависят от основного потока программы. Хотя существует несколько гарантий относительно того, когда и как выполняется асинхронное вычисление, существует ряд подходов к координации и планированию. Оставшаяся часть этой статьи посвящена основным понятиям F# Асинхронность и использованию типов, функций и выражений,

встроенных в. F#

Основные понятия

В F#асинхронное программирование основано на трех основных понятиях:

- Тип `Async<'T>`, представляющий Составное асинхронное вычисление.
- Функции модуля `Async`, которые позволяют планировать асинхронную работу, создавать асинхронные вычисления и преобразовывать асинхронные результаты.
- `async { }` вычислительное выражение, предоставляющее удобный синтаксис для создания асинхронных вычислений и управления ими.

Эти три концепции можно увидеть в следующем примере:

```
open System
open System.IO

let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn "File %s has %d bytes" fileName bytes.Length
    }

[<EntryPoint>]
let main argv =
    printTotalFileBytes "path-to-file.txt"
    |> Async.RunSynchronously

    Console.Read() |> ignore
    0
```

В примере функция `printTotalFileBytes` имеет тип `string -> Async<unit>`. Вызов функции фактически не выполняет асинхронное вычисление. Вместо этого он возвращает `Async<unit>`, который выступает в качестве *спецификации* работы, выполняемой асинхронно. Он вызывает `Async.AwaitTask` в своем тексте, который преобразует результат WriteAllBytesAsync в соответствующий тип.

Еще одна важная строка — вызов `Async.RunSynchronously`. Это одна из функций запуска асинхронного модуля, которые необходимо вызвать, если необходимо выполнить F# асинхронное вычисление.

Это фундаментальное различие в стиле C#/Visual Basic `async` программировании. В F#асинхронные вычисления можно рассматривать как **холодные задачи**. Они должны быть явным образом запущены для фактического выполнения. Это имеет некоторые преимущества, так как позволяет объединять и последовательно выполнять асинхронную работу гораздо проще, C# чем в или Visual Basic.

Объединение асинхронных вычислений

Ниже приведен пример, который строится на основе предыдущего путем объединения вычислений:

```
open System
open System.IO

let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn "File %s has %d bytes" fileName bytes.Length
    }

[<EntryPoint>]
let main argv =
    argv
    |> Array.map printTotalFileBytes
    |> Async.Parallel
    |> Async.Ignore
    |> Async.RunSynchronously

    0
```

Как видите, функция `main` имеет довольно много вызовов. По сути, он выполняет следующие действия:

1. Преобразуйте аргументы командной строки в `Async<unit>` вычисления с `Array.map`.
2. Создайте `Async<'T[]>`, который планирует и запускает `printTotalFileBytes` вычисления параллельно при запуске.
3. Создайте `Async<unit>`, который будет выполнять параллельное вычисление и игнорировать его результат.
4. Явным образом запустите Последнее вычисление с `Async.RunSynchronously` и заблокируйте его до завершения.

При запуске этой программы `printTotalFileBytes` выполняется параллельно для каждого аргумента командной строки. Поскольку асинхронные вычисления выполняются независимо от последовательности программ, порядок, в котором они печатаются, не задаются и завершается. Вычисления будут планироваться параллельно, но их порядок выполнения не гарантируется.

Асинхронные вычисления последовательности

Поскольку `Async<'T>` является спецификацией работы, а не выполняемой ранее задачей, можно легко выполнять более сложные преобразования. Ниже приведен пример, покоторому набор асинхронных вычислений помещается в последовательность, чтобы они выполнялись один за другим.

```
let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn "File %s has %d bytes" fileName bytes.Length
    }

[<EntryPoint>]
let main argv =
    argv
    |> Array.map printTotalFileBytes
    |> Async.Sequential
    |> Async.Ignore
    |> Async.RunSynchronously
    |> ignore
```

Это запланирует `printTotalFileBytes` выполняться в порядке элементов `argv` вместо того, чтобы планировать их параллельно. Поскольку следующий элемент не будет планироваться до тех пор, пока не завершится выполнение последнего вычисления, вычисления будут упорядочены таким, что в их выполнении

не будет перекрываться.

Важные функции асинхронного модуля

При написании асинхронного кода F# в обычно взаимодействует с платформой, которая обрабатывает планирование вычислений. Однако это не всегда так, поэтому рекомендуется изучить различные начальные функции для планирования асинхронной работы.

Поскольку F# асинхронные вычисления являются *спецификацией* работы, а не представлением уже выполненной работы, они должны быть явно запущены с помощью начальной функции. Существует множество функций асинхронного запуска , которые полезны в разных контекстах. В следующем разделе описаны некоторые из наиболее распространенных начальных функций.

Async. Стартчилд

Запускает дочерние вычисления в асинхронном вычислении. Это позволяет параллельно выполнять несколько асинхронных вычислений. Дочерние вычисления совместно используют токен отмены с родительскими вычислениями. Если родительские вычисления отменены, дочерние вычисления также отменяются.

Сигнатура:

```
computation: Async<'T> - timeout: ?int -> Async<Async<'T>>
```

Когда следует использовать:

- Если требуется одновременное выполнение нескольких асинхронных вычислений, а не по одному, но не запланированных параллельно.
- Если вы хотите привязать время существования дочернего вычисления к отношению к родительскому вычислению.

Что следует отслеживать:

- Запуск нескольких вычислений с `Async.StartChild` не так же, как параллельное планирование. Если вы хотите запланировать вычисления параллельно, используйте `Async.Parallel`.
- Отмена родительских вычислений приведет к срабатыванию отмены всех запущенных дочерних вычислений.

Async. Стартиммедиате

Выполняет асинхронное вычисление, немедленно запуская в текущем потоке операционной системы. Это полезно, если необходимо обновить что-либо в вызывающем потоке во время вычисления. Например, если асинхронное вычисление должно обновить пользовательский интерфейс (например, обновление индикатора выполнения), то `Async.StartImmediate` следует использовать.

Сигнатура:

```
computation: Async<unit> - cancellationToken: ?CancellationToken -> unit
```

Когда следует использовать:

- Когда необходимо обновить что-либо в вызывающем потоке в середине асинхронного вычисления.

Что следует отслеживать:

- Код асинхронного вычисления будет выполняться в любом потоке, для которого выполняется планирование. Это может быть проблематичным, если поток имеет какой-то секрет, например поток

пользовательского интерфейса. В таких случаях `Async.StartImmediate`, скорее всего, будет неприемлемым для использования.

Async. Стартастаск

Выполняет вычисление в пуле потоков. Возвращает Task<TResult>, который будет выполнен в соответствующем состоянии после завершения вычисления (создает результат, вызывает исключение или возвращает значение Cancel). Если токен отмены не указан, используется токен отмены по умолчанию.

Сигнатура:

```
computation: Async<'T> - taskCreationOptions: ?TaskCreationOptions - cancellationToken: ?CancellationToken ->
Task<'T>
```

Когда следует использовать:

- Когда необходимо вызвать API .NET, который требует Task<TResult> для представления результата асинхронного вычисления.

Что следует отслеживать:

- Этот вызов выделит дополнительный объект `Task`, который может увеличить издержки, если он часто используется.

Async. Parallel

Планирует параллельное выполнение последовательности асинхронных вычислений. Степень параллелизма может при необходимости настраиваться и регулироваться путем указания параметра `maxDegreesOfParallelism`.

Сигнатура:

```
computations: seq<Async<'T>> - ?maxDegreesOfParallelism: int -> Async<'T[]>
```

Когда следует использовать:

- Значение, если необходимо одновременно запустить набор вычислений и не иметь никакой зависимости от их порядка выполнения.
- Значение, если не требуется использовать результаты вычислений, запланированных параллельно, пока все они не будут завершены.

Что следует отслеживать:

- Доступ к результирующему массиву значений можно получить только после завершения всех вычислений.
- Вычисления будут выполняться, но они получат расписание. Это означает, что нельзя полагаться на порядок их выполнения.

Асинхронный. последовательный

Планирует выполнение последовательности асинхронных вычислений в том порядке, в котором они передаются. Будут выполнены первые вычисления, далее и т. д. Вычисления не будут выполняться параллельно.

Сигнатура:

```
computations: seq<Async<'T>> -> Async<'T[]>
```

Когда следует использовать:

- Значение, если необходимо выполнить несколько вычислений по порядку.

Что следует отслеживать:

- Доступ к результирующему массиву значений можно получить только после завершения всех вычислений.
- Вычисления будут выполняться в том порядке, в котором они передаются в эту функцию, что может означать, что больше времени должно пройти до возврата результатов.

Async. Аваиттаск

Возвращает асинхронное вычисление, которое ожидает завершения заданного Task<TResult> и возвращает его результат в виде `Async<'T>`

Сигнатура:

```
task: Task<'T>  -> Async<'T>
```

Когда следует использовать:

- При использовании API-интерфейса .NET, возвращающего Task<TResult> в F# асинхронном вычислении.

Что следует отслеживать:

- Исключения упаковываются в AggregateException соответствии с соглашением о библиотеке параллельных задач, и это отличается от асинхронного отображения F# исключений.

Async. catch

Создает асинхронное вычисление, которое выполняет заданный `Async<'T>`, возвращая `Async<Choice<'T, exn>>`. Если заданная `Async<'T>` успешно завершается, возвращается `Choice1Of2` с результирующим значением. Если перед завершением создается исключение, возвращается `Choice2of2` с вызванным исключением. Если он используется в асинхронном вычислении, который состоит из многих вычислений, и одно из этих вычислений создает исключение, охватывающие вычисления будут полностью остановлены.

Сигнатура:

```
computation: Async<'T> -> Async<Choice<'T, exn>>
```

Когда следует использовать:

- При выполнении асинхронной работы, которая может завершиться с исключением, и необходимо обойти это исключение в вызывающем объекте.

Что следует отслеживать:

- При использовании Объединенных или последовательных асинхронных вычислений охватывающие вычисления будут полностью прекращаться, если одно из внутренних вычислений выдаст исключение.

Async. Ignore

Создает асинхронное вычисление, которое выполняет заданное вычисление и игнорирует его результат.

Сигнатура:

```
computation: Async<'T> -> Async<unit>
```

Когда следует использовать:

- При наличии асинхронного вычисления, результат которого не требуется. Это аналогично коду `ignore` для неасинхронного кода.

Что следует отслеживать:

- Если необходимо использовать `Async.Start` или другую функцию, для которой требуется `Async<unit>`, рассмотрите возможность отмены результата. Отмена результатов в соответствии с сигнатурой типа не должна выполняться обычно.

Async. RunSynchronously нельзя вызывать

Выполняет асинхронное вычисление и ожидает его результат в вызывающем потоке. Этот вызов блокируется.

Сигнатура:

```
computation: Async<'T> - timeout: ?int - cancellationToken: ?CancellationToken -> 'T
```

Когда следует использовать:

- Если это необходимо, используйте его только один раз в приложении — в точке входа исполняемого файла.
- Если вы не следите за производительностью и хотите выполнить набор других асинхронных операций одновременно.

Что следует отслеживать:

- Вызов `Async.RunSynchronously` блокирует вызывающий поток до завершения выполнения.

Async. Start

Запускает асинхронное вычисление в пуле потоков, которое возвращает `unit`. Не ждет своего результата. Вложенные вычисления, запущенные с `Async.Start`, запускаются полностью независимо от родительских вычислений, которые вызвали их. Их время существования не привязано к какому-либо родительскому вычислению. Если родительские вычисления отменены, дочерние вычисления не отменяются.

Сигнатура:

```
computation: Async<unit> - cancellationToken: ?CancellationToken -> unit
```

Используйте только в следующих случаях:

- Имеется асинхронное вычисление, которое не дает результата и/или требует обработки одного из них.
- Вам не нужно знать, когда завершается асинхронное вычисление.
- Вы не волнует, на каком потоке выполняется асинхронное вычисление.
- У вас нет необходимости учитывать исключения, возникшие в результате выполнения задачи, и сообщать о них.

Что следует отслеживать:

- Исключения, вызванные вычислениями, запущенными с `Async.Start`, не распространяются на вызывающий объект. Стек вызовов будет полностью развернут.
- Любая работа (например, вызов `printfn`), запущенная с `Async.Start`, не приведет к выполнению этого действия в основном потоке выполнения программы.

Взаимодействие с .NET

Возможно, вы работаете с библиотекой или C# базой кода .NET, которая использует асинхронное программирование в стиле async/await. Поскольку C# и большинство библиотек .NET используют типы Task<TResult> и Task в качестве основных абстракций, а не `Async<'T>`, необходимо пересекать границы между этими двумя подходами и асинхронность.

Как работать с .NET Async и `Task<T>`

Работа с библиотеками асинхронных вычислений .NET и баз кода, использующих Task<TResult> (то есть асинхронные вычисления, которые имеют возвращаемые значения), проста F#и имеет встроенную поддержку.

Для ожидания асинхронного вычисления .NET можно использовать функцию `Async.AwaitTask`:

```
let getValueFromLibrary param =
    async {
        let! value = DotNetLibrary.GetValueAsync param |> Async.AwaitTask
        return value
    }
```

Функцию `Async.StartAsTask` можно использовать для передачи асинхронных вычислений вызывающему объекту .NET:

```
let computationForCaller param =
    async {
        let! result = getAsyncResult param
        return result
    } |> Async.StartAsTask
```

Как работать с .NET Async и `Task`

Для работы с API, которые используют Task (то есть асинхронные вычисления .NET, которые не возвращают значение), может потребоваться добавить дополнительную функцию, которая преобразует `Async<'T>` в Task:

```
module Async =
    // Async<unit> -> Task
    let startTaskFromAsyncUnit (comp: Async<unit>) =
        Async.StartAsTask comp :> Task
```

Уже существует `Async.AwaitTask`, принимающая Task в качестве входных данных. С помощью этой и ранее определенной функции `startTaskFromAsyncUnit` можно запускать и ожидать Task типов из F# асинхронных вычислений.

Связь с многопоточностью

Несмотря на то, что в этой статье упоминается потоковая ситуация, необходимо помнить о двух важных вещах.

1. Нет сходства между асинхронным вычислением и потоком, если только явно не запущен в текущем потоке.
2. Асинхронное программирование F# в не является абстракцией для многопоточности.

Например, вычисление может фактически выполняться в потоке вызывающего объекта в зависимости от природы работы. Вычисления могут также "переходить" между потоками, позаимствование их в течение небольшого количества времени для выполнения полезной работы в период ожидания (например, при передаче сетевого вызова).

Хотя F# предоставляет некоторые возможности для запуска асинхронного вычисления в текущем потоке

(или явно не в текущем потоке), асинхронность обычно не связан с конкретной стратегией потоков.

См. также:

- F# Асинхронная модель программирования
- F# Асинхронное руководством по Jet. com
- F#для получения удовольствия и асинхронного программного программирования прибыли
- Асинхронные C# в F#и: асинхронные проблемы вC#

Поставщики типов

29.11.2019 • 4 minutes to read • Edit Online

Поставщик типов F# — это компонент, предоставляющий типы, свойства и методы для использования в программе. Поставщики типов формируют, что известно как **предоставленные типы**, создаваемые F# компилятором и основанные на внешнем источнике данных.

Например, поставщик F# типов для SQL может формировать типы, представляющие таблицы и столбцы в реляционной базе данных. Фактически, это именно то, что делает поставщик типов дескриптора SqlProvider .

Предоставленные типы зависят от входных параметров от поставщика типов. Такие входные данные могут представлять собой образец источника данных (например, файл схемы JSON), URL-адрес, указывающий непосредственно на внешнюю службу или строку подключения к источнику данных. Поставщик типов также может гарантировать, что группы типов развертываются только по запросу; то есть они развертываются, если в программе на самом деле ссылаются ваши типы. Это позволяет применять прямую интеграцию по запросу широкомасштабных информационных пространств (например, рынков оперативных данных) в строго типизированном виде.

Поставщики регенеративного и стирания типов

Поставщики типов бывают двух видов: регенеративные и стертые.

Поставщики типов регенератив создают типы, которые могут быть записаны как типы .NET в сборку, в которой они создаются. Это позволяет использовать их из кода в других сборках. Это означает, что типизированное представление источника данных обычно должно иметь тип, который можно представить с помощью типов .NET.

При удалении поставщиков типов создаются типы, которые можно использовать только в сборке или проекте, из которых они созданы. Типы являются временными; то есть они не записываются в сборку и не могут использоваться кодом в других сборках. Они могут содержать *отложенные* члены, что позволяет использовать предоставленные типы из потенциально бесконечного пространства сведений. Они полезны при использовании небольшого подмножества большого и взаимосвязанных источников данных.

Часто используемые поставщики типов

Следующие широко используемые библиотеки содержат поставщики типов для различных целей:

- FSharp. Data включает поставщики типов для форматов и ресурсов JSON, XML, CSV и HTML-документов.
- Дескриптора SqlProvider предоставляет строго типизированный доступ к базам данных с помощью сопоставления объектов F# и запросов LINQ к этим источникам данных.
- FSharp. Data. SqlClient имеет набор поставщиков типов для внедрения T-SQL во F#время компиляции.
- Поставщик типов хранилища Azure предоставляет типы для больших двоичных объектов, таблиц и очередей Azure, что позволяет получать доступ к этим ресурсам без необходимости указывать имена ресурсов в качестве строк во всей программе.
- FSharp. Data. графкл содержит **графклпровидер**, который предоставляет типы на основе сервера графкл, заданного URL-адресом.

При необходимости можно создать собственные поставщики пользовательских типовили поставщики ссылочных типов, созданные другими пользователями. Например, предположим, что ваша организация

имеет службы данных, предоставляющие большое и возрастающее число именованных наборов данных, у каждого из которых имеется собственная стабильная схема данных. Можно создать поставщик типов, который считывает схемы и представляет программисту последние доступные наборы данных строго типизированным образом.

См. также:

- Учебник. Создание поставщика типов
- Справочник по языку F#

Учебник. Создание поставщика типов

25.11.2019 • 57 minutes to read • Edit Online

Механизм поставщика типов в F# является важной частью поддержки многофункционального программирования информации. В этом учебнике объясняется, как создавать собственные поставщики типов, проследуя разработку нескольких поставщиков простых типов, чтобы продемонстрировать основные понятия. Дополнительные сведения о механизме поставщика типов в см F#. в разделе Поставщики типов.

F# Экосистема содержит ряд поставщиков типов для часто используемых служб Интернет-и корпоративных данных. Пример:

- FSharp. Data включает поставщики типов для форматов документов JSON, XML, CSV и HTML.
- Дескриптора SqlProvider обеспечивает строго типизированный доступ к базам данных SQL с помощью сопоставления объектов F# и запросов LINQ к этим источникам данных.
- FSharp. Data. SqlClient имеет набор поставщиков типов для внедрения T-SQL во F#время компиляции.
- FSharp. Data. TypeProviders — это старый набор поставщиков типов для использования только с .NET Frameworkным программированием для доступа к службам данных SQL, Entity Framework, ODATA и WSDL.

При необходимости можно создавать пользовательские поставщики типов или ссылаться на поставщики типов, созданные другими пользователями. Например, в Организации может быть служба данных, предоставляющая большое и увеличивающееся количество именованных наборов данных, каждый из которых имеет собственную стабильную схему данных. Можно создать поставщик типов, который считывает схемы и представляет текущие наборы данных программисту строго типизированным способом.

Прежде чем начать

Механизм поставщика типов в основном предназначен для внедрения стабильных данных и пространств сведений о службе в процесс F# программирования.

Этот механизм не предназначен для внедрения информационных пространств, изменения схемы которых производятся во время выполнения программы, в целях, относящихся к логике программы. Кроме того, механизм не предназначен для мета-программирования внутри языка, хотя этот домен содержит некоторые допустимые варианты использования. Этот механизм следует использовать только в том случае, если это необходимо, а разработка поставщика типов дает очень высокое значение.

Не следует писать поставщик типов, если схема недоступна. Аналогично, следует избегать написания поставщика типов, где достаточно обычной (или даже существующей) библиотеки .NET.

Прежде чем начать, вы можете задать следующие вопросы:

- У вас есть схема для источника информации? Если да, то каково сопоставление в системе F# типов .NET и?
- Можно ли использовать существующий (динамически типизированный) API в качестве отправной точки для реализации?
- Будет ли ваша организация иметь достаточный объем использования поставщика типов, чтобы писать его целесообразным? Будет ли нормальная библиотека .NET соответствовать вашим потребностям?

- Сколько будет изменено в схеме?
- Будет ли оно изменяться во время кодирования?
- Будет ли он изменяться между сеансами кодирования?
- Будет ли оно изменяться во время выполнения программы?

Поставщики типов лучше всего подходят для ситуаций, когда схема является стабильной во время выполнения и в течение времени существования скомпилированного кода.

Поставщик простых типов

Этот пример — Samples. хелловорлдтипепровидер, аналогичный образцам в каталоге `examples` в F# пакете SDK поставщика типов. Поставщик предоставляет доступ к "пространству типов", который содержит 100 удаленных типов, как показано в следующем примере кода с F# использованием синтаксиса Signature и пропуск сведений для всех, кроме `Type1`. Дополнительные сведения о стирании типов см. Далее в подразделе сведения о стирании указанных типов .

```
namespace Samples.HelloWorldTypeProvider

type Type1 =
    /// This is a static property.
    static member StaticProperty : string

    /// This constructor takes no arguments.
    new : unit -> Type1

    /// This constructor takes one argument.
    new : data:string -> Type1

    /// This is an instance property.
    member InstanceProperty : int

    /// This is an instance method.
    member InstanceMethod : x:int -> char

    nested type NestedType =
        /// This is StaticProperty1 on NestedType.
        static member StaticProperty1 : string
        …
        /// This is StaticProperty100 on NestedType.
        static member StaticProperty100 : string

type Type2 =
…
…

type Type100 =
…
```

Обратите внимание, что набор типов и членов является статически известным. Этот пример не использует возможности поставщиков для предоставления типов, зависящих от схемы. Реализация поставщика типов описана в следующем коде, а сведения приведены в последующих разделах этой статьи.

> **WARNING**
> Между этим кодом и примерами в сети могут быть различия.

```
namespace Samples.FSharp.HelloWorldTypeProvider

open System
open System.Reflection
open ProviderImplementation.ProvidedTypes
open FSharp.Core.CompilerServices
open FSharp.Quotations

// This type defines the type provider. When compiled to a DLL, it can be added
// as a reference to an F# command-line compilation, script, or project.
[<TypeProvider>]
type SampleTypeProvider(config: TypeProviderConfig) as this =

  // Inheriting from this type provides implementations of ITypeProvider
  // in terms of the provided types below.
  inherit TypeProviderForNamespaces(config)

  let namespaceName = "Samples.HelloWorldTypeProvider"
  let thisAssembly = Assembly.GetExecutingAssembly()

  // Make one provided type, called TypeN.
  let makeOneProvidedType (n:int) =
  …
  // Now generate 100 types
  let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]

  // And add them to the namespace
  do this.AddNamespace(namespaceName, types)

[<assembly:TypeProviderAssembly>]
do()
```

Чтобы использовать этот поставщик, откройте отдельный экземпляр Visual Studio, создайте F# скрипт, а затем добавьте ссылку на поставщик из скрипта с помощью #r, как показано в следующем коде:

```
#r @".\bin\Debug\Samples.HelloWorldTypeProvider.dll"

let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")

let obj2 = Samples.HelloWorldTypeProvider.Type1("some other data")

obj1.InstanceProperty
obj2.InstanceProperty

[ for index in 0 .. obj1.InstanceProperty-1 -> obj1.InstanceMethod(index) ]
[ for index in 0 .. obj2.InstanceProperty-1 -> obj2.InstanceMethod(index) ]

let data1 = Samples.HelloWorldTypeProvider.Type1.NestedType.StaticProperty35
```

Затем найдите типы в пространстве имен `Samples.HelloWorldTypeProvider`, созданном поставщиком типов.

Перед повторной компиляцией поставщика убедитесь, что закрыты все экземпляры Visual Studio и F# Interactive, использующие библиотеку DLL поставщика. В противном случае произойдет ошибка сборки, так как выходная библиотека DLL будет заблокирована.

Чтобы отладить этот поставщик с помощью инструкций Print, создайте сценарий, который предоставляет проблему с поставщиком, а затем используйте следующий код:

```
fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx
```

Чтобы отладить этот поставщик с помощью Visual Studio, откройте Командная строка разработчика для

Visual Studio с учетными данными администратора и выполните следующую команду:

```
devenv.exe /debugexe fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx
```

В качестве альтернативы откройте Visual Studio, откройте меню Отладка, выберите `Debug/Attach to process…` и подключитесь к другому `devenv` процесса, в котором редактируется скрипт. С помощью этого метода можно легко ориентироваться на определенную логику в поставщике типов путем интерактивного ввода выражений во второй экземпляр (с полной технологией IntelliSense и другими функциями).

Можно отключить отладку Только мой код для лучшего обнаружения ошибок в созданном коде. Сведения о том, как включить или отключить эту функцию, см. в разделе Навигация по коду с помощью отладчика. Кроме того, можно также задать перехват первого шанса исключений, открыв меню `Debug`, выбрав `Exceptions` или нажав клавиши CTRL + ALT + E, чтобы открыть диалоговое окно `Exceptions`. В этом диалоговом окне в разделе `Common Language Runtime Exceptions` установите флажок `Thrown`.

Реализация поставщика типов

В этом разделе описываются основные разделы реализации поставщика типов. Сначала необходимо определить тип самого поставщика пользовательского типа:

```
[<TypeProvider>]
type SampleTypeProvider(config: TypeProviderConfig) as this =
```

Этот тип должен быть открытым, и его необходимо пометить атрибутом типепровидер , чтобы компилятор распознал поставщик типов, когда отдельный F# проект ссылается на сборку, содержащую тип. Параметр *config* является необязательным, и, если он F# имеется, содержит контекстные сведения о конфигурации для экземпляра поставщика типов, создаваемого компилятором.

Далее вы реализуете интерфейс ITypeProvider . В этом случае используется тип `TypeProviderForNamespaces` из API `ProvidedTypes` в качестве базового типа. Этот вспомогательный тип может предоставить конечную коллекцию предопределенных пространств имен, каждый из которых непосредственно содержит конечное количество фиксированных, предпредоставленных типов. В этом контексте поставщик *заранее* создает типы, даже если они не нужны или не используются.

```
inherit TypeProviderForNamespaces(config)
```

Затем определите локальные закрытые значения, задающие пространство имен для указанных типов, и найдите саму сборку поставщика типов. В дальнейшем эта сборка используется как логический тип родительского объекта для указанных удаленных типов.

```
let namespaceName = "Samples.HelloWorldTypeProvider"
let thisAssembly = Assembly.GetExecutingAssembly()
```

Затем создайте функцию для предоставления каждого типа Type1… Type100. Эта функция более подробно описана далее в этом разделе.

```
let makeOneProvidedType (n:int) = …
```

Затем создайте типы, предоставленные 100:

```
let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]
```

Затем добавьте типы в качестве предоставленного пространства имен:

```
do this.AddNamespace(namespaceName, types)
```

Наконец, добавьте атрибут сборки, указывающий, что создается библиотека DLL поставщика типов:

```
[<assembly:TypeProviderAssembly>]
do()
```

Предоставление одного типа и его членов

Функция `makeOneProvidedType` выполняет реальную работу по предоставлению одного из типов.

```
let makeOneProvidedType (n:int) =
…
```

Этот шаг описывает реализацию этой функции. Сначала создайте предоставленный тип (например, тип1, если n = 1 или Type57, если n = 57).

```
// This is the provided type. It is an erased provided type and, in compiled code,
// will appear as type 'obj'.
let t = ProvidedTypeDefinition(thisAssembly, namespaceName,
                               "Type" + string n,
                               baseType = Some typeof<obj>)
```

Обратите внимание на следующие моменты:

- Указанный тип стерт. Так как вы указываете, что базовый тип `obj`, экземпляры будут отображаться в виде значений типа obj в скомпилированном коде.
- При указании невложенного типа необходимо указать сборку и пространство имен. Для удаленных типов сборка должна быть самой сборкой поставщика типов.

Затем добавьте XML-документацию в тип. Эта документация является отложенной, то есть вычисленной по запросу, если она требуется компилятору узла.

```
t.AddXmlDocDelayed (fun () -> sprintf "This provided type %s" ("Type" + string n))
```

Далее вы добавите в тип предоставленное статическое свойство:

```
let staticProp = ProvidedProperty(propertyName = "StaticProperty",
                                  propertyType = typeof<string>,
                                  isStatic = true,
                                  getterCode = (fun args -> <@@ "Hello!" @@>))
```

При получении этого свойства всегда будет вычислена строка "Hello!". `GetterCode` для свойства использует F# предложение, представляющее код, формируемый компилятором хоста для получения свойства. Дополнительные сведения о предложениях см. в разделе цитирование кодаF#().

Добавьте XML-документацию в свойство.

```
staticProp.AddXmlDocDelayed(fun () -> "This is a static property")
```

Теперь присоедините предоставленное свойство к предоставленному типу. Необходимо присоединить

предоставленный элемент к одному и только одному типу. В противном случае член никогда не будет доступен.

```
t.AddMember staticProp
```

Теперь создайте предоставленный конструктор, не принимающий параметров.

```
let ctor = ProvidedConstructor(parameters = [ ],
                               invokeCode = (fun args -> <@@ "The object data" :> obj @@>))
```

`InvokeCode` для конструктора возвращает F# предложение, представляющее код, создаваемый компилятором хоста при вызове конструктора. Например, можно использовать следующий конструктор:

```
new Type10()
```

Экземпляр предоставленного типа будет создан с базовыми данными "данные объекта". Код в кавычках включает преобразование в obj , так как этот тип является очистки данного предоставленного типа (как указано при объявлении предоставленного типа).

Добавьте XML-документацию в конструктор и добавьте предоставленный конструктор в предоставленный тип:

```
ctor.AddXmlDocDelayed(fun () -> "This is a constructor")

t.AddMember ctor
```

Создайте второй предоставленный конструктор, который принимает один параметр:

```
let ctor2 =
ProvidedConstructor(parameters = [ ProvidedParameter("data",typeof<string>) ],
                    invokeCode = (fun args -> <@@ (%%(args.[0]) : string) :> obj @@>))
```

`InvokeCode` для конструктора снова возвращает F# предложение, представляющее код, созданный компилятором узла для вызова метода. Например, можно использовать следующий конструктор:

```
new Type10("ten")
```

Экземпляр предоставленного типа создается с базовыми данными «десять». Возможно, вы уже заметили, что функция `InvokeCode` Возвращает предложение. Входными данными для этой функции является список выражений, по одному на каждый параметр конструктора. В этом случае выражение, представляющее одно значение параметра, доступно в `args.[0]`. Код для вызова конструктора приводит возвращаемое значение к удаленному типу `obj`. После добавления второго предоставленного конструктора к типу вы создадите предоставленное свойство экземпляра:

```
let instanceProp =
    ProvidedProperty(propertyName = "InstanceProperty",
                     propertyType = typeof<int>,
                     getterCode= (fun args ->
                        <@@ ((%%(args.[0]) : obj) :?> string).Length @@>))
instanceProp.AddXmlDocDelayed(fun () -> "This is an instance property")
t.AddMember instanceProp
```

Получение этого свойства возвратит длину строки, которая является объектом представления. Свойство `GetterCode` возвращает F# предложение, которое указывает код, формируемый компилятором узла для получения свойства. Как и `InvokeCode`, функция `GetterCode` Возвращает предложение. Компилятор узла вызывает эту функцию со списком аргументов. В этом случае аргументы включают только одно выражение, представляющее экземпляр, на основе которого вызывается метод получения, к которому можно получить доступ с помощью `args.[0]`. Реализация `GetterCode` присоединение к результирующей цитате на удаленном типе `obj`, а приведение используется для обеспечения механизма компилятора для проверки типов, которые объект является строковым. Следующая часть `makeOneProvidedType` предоставляет метод экземпляра с одним параметром.

```
let instanceMeth =
    ProvidedMethod(methodName = "InstanceMethod",
                   parameters = [ProvidedParameter("x",typeof<int>)],
                   returnType = typeof<char>,
                   invokeCode = (fun args ->
                       <@@ ((%%(args.[0]) : obj) :?> string).Chars(%%(args.[1]) : int) @@>))

instanceMeth.AddXmlDocDelayed(fun () -> "This is an instance method")
// Add the instance method to the type.
t.AddMember instanceMeth
```

Наконец, создайте вложенный тип, содержащий 100 вложенных свойств. Создание этого вложенного типа и его свойств отложено, то есть вычисленное по запросу.

```
t.AddMembersDelayed(fun () ->
  let nestedType = ProvidedTypeDefinition("NestedType", Some typeof<obj>)

  nestedType.AddMembersDelayed (fun () ->
    let staticPropsInNestedType =
      [
          for i in 1 .. 100 ->
              let valueOfTheProperty = "I am string "  + string i

              let p =
                ProvidedProperty(propertyName = "StaticProperty" + string i,
                  propertyType = typeof<string>,
                  isStatic = true,
                  getterCode= (fun args -> <@@ valueOfTheProperty @@>))

              p.AddXmlDocDelayed(fun () ->
                  sprintf "This is StaticProperty%d on NestedType" i)

              p
      ]

    staticPropsInNestedType)

  [nestedType])
```

Сведения о стирании указанных типов

Пример в этом разделе содержит только *Удаленные предоставленные типы*, которые особенно полезны в следующих ситуациях.

- При создании поставщика для информационного пространства, содержащего только данные и методы.
- При написании поставщика, где точная семантика типа среды выполнения не критична для практического использования информационного пространства.
- При написании поставщика для информационного пространства настолько крупным и

взаимосвязанным, что не рекомендуется создавать реальные типы .NET для информационного пространства.

В этом примере каждый предоставленный тип удаляется в тип `obj`, и все типы использования типа будут отображаться в скомпилированном коде как тип `obj`. Фактически, базовые объекты в этих примерах являются строками, но тип будет выглядеть как `System.Object` в скомпилированном коде .NET. Как и при использовании типа очистки, можно использовать явную упаковку, распаковку и приведение к обходят удаленным типам. В этом случае при использовании объекта может возникнуть исключение приведения, которое является недопустимым. Среда выполнения поставщика может определять собственный частный тип представления для защиты от ложных представлений. Нельзя определять стертые типы в F# самом себе. Только указанные типы могут быть стерты. Необходимо понимать последствия, как практичные, так и семантические, используя либо стертые типы для поставщика типов, либо поставщик, предоставляющий стертые типы. У стирания типа нет действительного типа .NET. Таким образом, невозможно выполнить точное отражение для типа, и вы можете обходят стереть типы при использовании приведений во время выполнения и других методов, зависящих от семантики типов среды выполнения. Подверсия стертых типов часто приводит к исключениям приведения типов во время выполнения.

Выбор представлений для удаленных предоставленных типов

Для некоторых применений удаленных предоставленных типов представление не требуется. Например, стертный предоставленный тип может содержать только статические свойства и члены без конструкторов, а методы и свойства не возвращают экземпляр типа. Если можно достичь экземпляров удаленных предоставленных типов, необходимо учитывать следующие вопросы.

Что такое очистки указанного типа?

- Очистки предоставленного типа — это то, как тип отображается в скомпилированном коде .NET.
- Очистки указанного типа очищенного класса всегда является первым нестертым базовым типом в цепочке наследования типа.
- Очистки указанного типа удаленных интерфейсов всегда `System.Object`.

Какие представления предоставленного типа?

- Набор возможных объектов для очищенного предоставленного типа называется его представлениями. В примере в этом документе представления всех удаленных предоставленных типов `Type1..Type100` всегда являются строковыми объектами.

Все представления указанного типа должны быть совместимы с очистки указанного типа. (В противном случае F# либо компилятор выдаст ошибку для использования поставщика типов, либо будет создан недопустимый недействительный код .NET. Поставщик типов является недопустимым, если он возвращает код, предоставляющий Недопустимое представление.)

Можно выбрать представление для предоставленных объектов с помощью любого из следующих подходов, которые являются очень распространенными:

- Если вы просто предоставляете строго типизированную оболочку для существующего типа .NET, часто имеет смысл стереть этот тип, использовать экземпляры этого типа как представления или и то, и другое. Этот подход подходит, если большинство существующих методов этого типа все еще имеет смысл при использовании строго типизированной версии.
- Если вы хотите создать API, который значительно отличается от существующего API-интерфейса .NET, имеет смысл создать типы среды выполнения, которые будут очистки типа и представления для предоставленных типов.

В примере в этом документе строки используются как представления предоставленных объектов. Часто может быть целесообразно использовать другие объекты для представления. Например, словарь можно

использовать в качестве контейнера свойств:

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new Dictionary<string,obj>()) :> obj @@>))
```

В качестве альтернативы можно определить тип в поставщике типов, который будет использоваться во время выполнения для формирования представления, а также одной или нескольких операций среды выполнения:

```
type DataObject() =
    let data = Dictionary<string,obj>()
    member x.RuntimeOperation() = data.Count
```

Указанные члены могут затем создавать экземпляры этого типа объектов:

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new DataObject()) :> obj @@>))
```

В этом случае вы можете (необязательно) использовать этот тип в качестве типа очистки, указав этот тип в качестве `baseType` при создании `ProvidedTypeDefinition`:

```
ProvidedTypeDefinition(…, baseType = Some typeof<DataObject> )
…
ProvidedConstructor(…, InvokeCode = (fun args -> <@@ new DataObject() @@>), …)
```

Ключевые уроки

В предыдущем разделе было объяснено, как создать простой поставщик типов стирания, предоставляющий ряд типов, свойств и методов. В этом разделе также объяснено понятие типа очистки, в том числе некоторые преимущества и недостатки предоставления удаленных типов от поставщика типов и обсуждаются представления удаленных типов.

Поставщик типов, использующий статические параметры

Возможность параметризации поставщиков типов статическими данными позволяет реализовать множество интересных сценариев, даже если поставщику не требуется доступ к локальным или удаленным данным. В этом разделе вы узнаете о некоторых основных методах совместного размещения таких поставщиков.

Тип проверенного поставщика регулярных выражений

Представьте, что необходимо реализовать поставщик типов для регулярных выражений, которые создают оболочку для библиотек .NET Regex в интерфейсе, который предоставляет следующие гарантии времени компиляции:

- Проверка того, является ли регулярное выражение допустимым.
- Предоставление именованных свойств в совпадениях, основанных на именах групп в регулярном выражении.

В этом разделе показано, как использовать поставщики типов для создания `RegexTyped` типа, который параметризация шаблона регулярного выражения предоставляет эти преимущества. Компилятор сообщит об ошибке, если предоставленный шаблон недействителен, и поставщик типов может извлекать группы из шаблона, чтобы к ним можно было обращаться с помощью именованных свойств в совпадениях. При проектировании поставщика типов следует учитывать, как предоставляемый API должен выглядеть для

конечных пользователей и как этот проект будет преобразован в код .NET. В следующем примере показано, как использовать такой API для получения компонентов кода области:

```
type T = RegexTyped< @"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)">
let reg = T()
let result = T.IsMatch("425-555-2345")
let r = reg.Match("425-555-2345").Group_AreaCode.Value //r equals "425"
```

В следующем примере показано, как поставщик типов преобразует эти вызовы:

```
let reg = new Regex(@"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)")
let result = reg.IsMatch("425-123-2345")
let r = reg.Match("425-123-2345").Groups.["AreaCode"].Value //r equals "425"
```

Обратите внимание на следующие моменты.

- Стандартный тип Regex представляет параметризованный `RegexTyped` тип.
- Конструктор `RegexTyped` приводит к вызову конструктора Regex, передавая аргумент статического типа для шаблона.
- Результаты метода `Match` представлены стандартным типом Match.
- Каждая именованная группа приводит к предоставленному свойству, а доступ к свойству приводит к использованию индексатора для коллекции `Groups` совпадений.

Следующий код является основой логики для реализации такого поставщика, и в этом примере Добавление всех элементов к предоставленному типу не производится. Сведения о каждом добавленном члене см. в соответствующем разделе далее в этом разделе. Чтобы получить полный код, Скачайте пример из F# примера пакета 3,0 на веб-сайте CodePlex.

```
namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types
    let thisAssembly = Assembly.GetExecutingAssembly()
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
    let baseTy = typeof<obj>
    let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

    let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

    do regexTy.DefineStaticParameters(
        parameters=staticParams,
        instantiationFunction=(fun typeName parameterValues ->

          match parameterValues with
          | [| :? string as pattern|] ->

            // Create an instance of the regular expression.
            //
            // This will fail with System.ArgumentException if the regular expression is not valid.
            // The exception will escape the type provider and be reported in client code.
            let r = System.Text.RegularExpressions.Regex(pattern)

            // Declare the typed regex provided type.
            // The type erasure of this type is 'obj', even though the representation will always be a Regex
            // This, combined with hiding the object methods, makes the IntelliSense experience simpler.
            let ty =
              ProvidedTypeDefinition(
                thisAssembly,
                rootNamespace,
                typeName,
                baseType = Some baseTy)

            ...

            ty
          | _ -> failwith "unexpected parameter values"))

    do this.AddNamespace(rootNamespace, [regexTy])

[<TypeProviderAssembly>]
do ()
```

Обратите внимание на следующие моменты.

- Поставщик типов принимает два статических параметра: `pattern`, обязательный, и `options`, которые являются необязательными (поскольку предоставляется значение по умолчанию).
- После того как будут указаны статические аргументы, создается экземпляр регулярного выражения. Этот экземпляр вызывает исключение, если регулярное выражение имеет неправильный формат, и эта ошибка будет передана пользователям.
- В обратном вызове `DefineStaticParameters` определяется тип, который будет возвращен после указания аргументов.
- Этот код задает для `HideObjectMethods` значение true, чтобы технология IntelliSense оставалась

оптимизированной. Этот атрибут приводит к подавлению членов `Equals`, `GetHashCode`, `Finalize` и `GetType` из списков IntelliSense для предоставленного объекта.

- В качестве базового типа метода используется `obj`, но в качестве представления времени выполнения для этого типа используется объект `Regex`, как показано в следующем примере.
- Вызов конструктора `Regex` вызывает исключение ArgumentException, если регулярное выражение является недопустимым. Компилятор перехватывает это исключение и сообщает пользователю сообщение об ошибке во время компиляции или в редакторе Visual Studio. Это исключение позволяет проверять регулярные выражения без запуска приложения.

Указанный выше тип не полезен, так как он не содержит значимых методов или свойств. Сначала добавьте статический метод `IsMatch`:

```
let isMatch =
    ProvidedMethod(
        methodName = "IsMatch",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = typeof<bool>,
        isStatic = true,
        invokeCode = fun args -> <@@ Regex.IsMatch(%%args.[0], pattern) @@>)

isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified input string."
ty.AddMember isMatch
```

Предыдущий код определяет метод `IsMatch`, который принимает строку в качестве входных данных и возвращает `bool`. Единственная сложная часть — использование аргумента `args` в определении `InvokeCode`. В этом примере `args` представляет собой список предложений, представляющих аргументы для этого метода. Если метод является методом экземпляра, первый аргумент представляет `this` аргумент. Однако для статического метода аргументы являются только явными аргументами метода. Обратите внимание, что тип значения в кавычках должен соответствовать указанному возвращаемому типу (в данном случае `bool`). Также обратите внимание, что этот код использует метод `AddXmlDoc`, чтобы убедиться, что предоставленный метод также имеет полезную документацию, которую можно передать с помощью IntelliSense.

Затем добавьте метод Match для экземпляра. Однако этот метод должен возвращать значение предоставленного типа `Match`, чтобы обеспечить доступ к группам строго типизированным способом. Поэтому сначала объявите тип `Match`. Поскольку этот тип зависит от шаблона, который был передан как статический аргумент, этот тип должен быть вложен в определение параметризованного типа:

```
let matchTy =
    ProvidedTypeDefinition(
        "MatchType",
        baseType = Some baseTy,
        hideObjectMethods = true)

ty.AddMember matchTy
```

Затем добавьте одно свойство в тип сопоставления для каждой группы. Во время выполнения сопоставление представляется в виде Match значения, поэтому для получения соответствующей группы в предложении, определяющем свойство, необходимо использовать Groups индексированное свойство.

```
for group in r.GetGroupNames() do
    // Ignore the group named 0, which represents all input.
    if group <> "0" then
    let prop =
      ProvidedProperty(
        propertyName = group,
        propertyType = typeof<Group>,
        getterCode = fun args -> <@@ ((%%args.[0]:obj) :?> Match).Groups.[group] @@>)
        prop.AddXmlDoc(sprintf @"Gets the ""%s"" group from this match" group)
    matchTy.AddMember prop
```

Обратите внимание, что вы добавляете XML-документацию к предоставленному свойству. Также обратите внимание, что свойство может быть считано, если предоставлена функция `GetterCode`, а свойство может быть записано, если предоставлена функция `SetterCode`, поэтому полученное свойство доступно только для чтения.

Теперь можно создать метод экземпляра, возвращающий значение этого типа `Match`:

```
let matchMethod =
    ProvidedMethod(
        methodName = "Match",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = matchTy,
        invokeCode = fun args -> <@@ ((%%args.[0]:obj) :?> Regex).Match(%%args.[1]) :> obj @@>)

matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this regular expression"

ty.AddMember matchMeth
```

Поскольку создается метод экземпляра, `args.[0]` представляет экземпляр `RegexTyped`, для которого вызывается метод, а `args.[1]` является входным аргументом.

Наконец, предоставьте конструктор, чтобы можно было создать экземпляры указанного типа.

```
let ctor =
    ProvidedConstructor(
        parameters = [],
        invokeCode = fun args -> <@@ Regex(pattern, options) :> obj @@>)

ctor.AddXmlDoc("Initializes a regular expression instance.")

ty.AddMember ctor
```

Конструктор просто удаляется до создания стандартного экземпляра Regex .NET, который снова упаковывается в объект, так как `obj` очистки предоставленного типа. С этим изменением пример использования API, указанный ранее в разделе, работает должным образом. Следующий код является полным и окончательным:

```
namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types.
```

```
    let thisAssembly = Assembly.GetExecutingAssembly()
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
    let baseTy = typeof<obj>
    let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

    let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

    do regexTy.DefineStaticParameters(
        parameters=staticParams,
        instantiationFunction=(fun typeName parameterValues ->

            match parameterValues with
            | [| :? string as pattern|] ->

                // Create an instance of the regular expression.

                let r = System.Text.RegularExpressions.Regex(pattern)

                // Declare the typed regex provided type.

                let ty =
                    ProvidedTypeDefinition(
                        thisAssembly,
                        rootNamespace,
                        typeName,
                        baseType = Some baseTy)

                ty.AddXmlDoc "A strongly typed interface to the regular expression '%s'"

                // Provide strongly typed version of Regex.IsMatch static method.
                let isMatch =
                    ProvidedMethod(
                        methodName = "IsMatch",
                        parameters = [ProvidedParameter("input", typeof<string>)],
                        returnType = typeof<bool>,
                        isStatic = true,
                        invokeCode = fun args -> <@@ Regex.IsMatch(%%args.[0], pattern) @@>)

                isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified
input string"

                ty.AddMember isMatch

                // Provided type for matches
                // Again, erase to obj even though the representation will always be a Match
                let matchTy =
                    ProvidedTypeDefinition(
                        "MatchType",
                        baseType = Some baseTy,
                        hideObjectMethods = true)

                // Nest the match type within parameterized Regex type.
                ty.AddMember matchTy

                // Add group properties to match type
                for group in r.GetGroupNames() do
                    // Ignore the group named 0, which represents all input.
                    if group <> "0" then
                        let prop =
                          ProvidedProperty(
                            propertyName = group,
                            propertyType = typeof<Group>,
                            getterCode = fun args -> <@@ ((%%args.[0]:obj) :?> Match).Groups.[group] @@>)
                        prop.AddXmlDoc(sprintf @"Gets the ""%s"" group from this match" group)
                        matchTy.AddMember(prop)

                // Provide strongly typed version of Regex.Match instance method.
                let matchMeth =
                  ProvidedMethod(
```

```
                    methodName = "Match",
                    parameters = [ProvidedParameter("input", typeof<string>)],
                    returnType = matchTy,
                    invokeCode = fun args -> <@@ ((%%args.[0]:obj) :?> Regex).Match(%%args.[1]) :> obj @@>)
                matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this
regular expression"

                ty.AddMember matchMeth

                // Declare a constructor.
                let ctor =
                  ProvidedConstructor(
                    parameters = [],
                    invokeCode = fun args -> <@@ Regex(pattern) :> obj @@>)

                // Add documentation to the constructor.
                ctor.AddXmlDoc "Initializes a regular expression instance"

                ty.AddMember ctor

                ty
            | _ -> failwith "unexpected parameter values"))

    do this.AddNamespace(rootNamespace, [regexTy])

[<TypeProviderAssembly>]
do ()
```

Ключевые уроки

В этом разделе описано, как создать поставщик типов, который работает с его статическими параметрами. Поставщик проверяет статический параметр и предоставляет операции на основе его значения.

Поставщик типов, который поддерживается локальными данными

Зачастую может потребоваться, чтобы поставщики типов представим интерфейсы API на основе не только статических параметров, но и информации из локальных или удаленных систем. В этом разделе обсуждаются поставщики типов, основанные на локальных данных, например локальные файлы данных.

Поставщик простых CSV-файлов

В качестве простого примера рассмотрим поставщик типов для доступа к научных данным в формате CSV (с разделителями-запятыми). В этом разделе предполагается, что CSV-файлы содержат строку заголовка, за которой следуют данные с плавающей запятой, как показано в следующей таблице.

РАССТОЯНИЕ (СЧЕТЧИК)	ВРЕМЯ (СЕКУНД)
50.0	3.7
100.0	5.2
150.0	6.4

В этом разделе показано, как предоставить тип, который можно использовать для получения строк со свойством `Distance` типа `float<meter>` и свойством `Time` типа `float<second>`. Для простоты выполняются следующие предположения.

- Имена заголовков не должны содержать запятые или имеют форму "имя (единица)".
- Единицы — это все системные международные (Si) единицы, которые определяются модулем Microsoft. FSharp. Data. унитсистемс.F#Si. UnitNames Module () .

- Все единицы являются простыми (например, счетчик), а не составными (например, измерение/секунда).
- Все столбцы содержат данные с плавающей запятой.

Более полный поставщик расставит эти ограничения.

Первый шаг — определить, как должен выглядеть API. При наличии файла `info.csv` с содержимым из предыдущей таблицы (в формате с разделителями-запятыми) Пользователи поставщика должны иметь возможность написать код, похожий на следующий пример:

```
let info = new MiniCsv<"info.csv">()
for row in info.Data do
let time = row.Time
printfn "%f" (float time)
```

В этом случае компилятор должен преобразовать эти вызовы в что-то, как показано в следующем примере:

```
let info = new CsvFile("info.csv")
for row in info.Data do
let (time:float) = row.[1]
printfn "%f" (float time)
```

Оптимальный перевод требует, чтобы поставщик типов определял реальный тип `CsvFile` в сборке поставщика типов. Поставщики типов часто используют несколько вспомогательных типов и методов для создания оболочки для важной логики. Поскольку меры удаляются во время выполнения, можно использовать `float[]` в качестве стирания типа для строки. Компилятор будет обрабатывать различные столбцы с разными типами мер. Например, первый столбец в нашем примере имеет тип `float<meter>`, а второй — `float<second>`. Однако удаленное представление может остаться довольно простым.

В следующем коде показано ядро реализации.

```
// Simple type wrapping CSV data
type CsvFile(filename) =
    // Cache the sequence of all data lines (all lines but the first)
    let data =
        seq {
            for line in File.ReadAllLines(filename) |> Seq.skip 1 ->
                line.Split(',') |> Array.map float
        }
        |> Seq.cache
    member _.Data = data

[<TypeProvider>]
type public MiniCsvProvider(cfg:TypeProviderConfig) as this =
    inherit TypeProviderForNamespaces(cfg)

    // Get the assembly and namespace used to house the provided types.
    let asm = System.Reflection.Assembly.GetExecutingAssembly()
    let ns = "Samples.FSharp.MiniCsvProvider"

    // Create the main provided type.
    let csvTy = ProvidedTypeDefinition(asm, ns, "MiniCsv", Some(typeof<obj>))

    // Parameterize the type by the file to use as a template.
    let filename = ProvidedStaticParameter("filename", typeof<string>)
    do csvTy.DefineStaticParameters([filename], fun tyName [| :? string as filename |] ->

        // Resolve the filename relative to the resolution folder.
        let resolvedFilename = Path.Combine(cfg.ResolutionFolder, filename)
```

```
        // Get the first line from the file.
        let headerLine = File.ReadLines(resolvedFilename) |> Seq.head

        // Define a provided type for each row, erasing to a float[].
        let rowTy = ProvidedTypeDefinition("Row", Some(typeof<float[]>))

        // Extract header names from the file, splitting on commas.
        // use Regex matching to get the position in the row at which the field occurs
        let headers = Regex.Matches(headerLine, "[^,]+")

        // Add one property per CSV field.
        for i in 0 .. headers.Count - 1 do
            let headerText = headers.[i].Value

            // Try to decompose this header into a name and unit.
            let fieldName, fieldTy =
                let m = Regex.Match(headerText, @"(?<field>.+) \((?<unit>.+)\)")
                if m.Success then

                    let unitName = m.Groups.["unit"].Value
                    let units = ProvidedMeasureBuilder.Default.SI unitName
                    m.Groups.["field"].Value, ProvidedMeasureBuilder.Default.AnnotateType(typeof<float>,
[units])

                else
                    // no units, just treat it as a normal float
                    headerText, typeof<float>

            let prop =
                ProvidedProperty(fieldName, fieldTy,
                    getterCode = fun [row] -> <@@ (%%row:float[]).[i] @@>)

            // Add metadata that defines the property's location in the referenced file.
            prop.AddDefinitionLocation(1, headers.[i].Index + 1, filename)
            rowTy.AddMember(prop)

        // Define the provided type, erasing to CsvFile.
        let ty = ProvidedTypeDefinition(asm, ns, tyName, Some(typeof<CsvFile>))

        // Add a parameterless constructor that loads the file that was used to define the schema.
        let ctor0 =
            ProvidedConstructor([],
                invokeCode = fun [] -> <@@ CsvFile(resolvedFilename) @@>)
        ty.AddMember ctor0

        // Add a constructor that takes the file name to load.
        let ctor1 = ProvidedConstructor([ProvidedParameter("filename", typeof<string>)],
            invokeCode = fun [filename] -> <@@ CsvFile(%%filename) @@>)
        ty.AddMember ctor1

        // Add a more strongly typed Data property, which uses the existing property at runtime.
        let prop =
            ProvidedProperty("Data", typedefof<seq<_>>.MakeGenericType(rowTy),
                getterCode = fun [csvFile] -> <@@ (%%csvFile:CsvFile).Data @@>)
        ty.AddMember prop

        // Add the row type as a nested type.
        ty.AddMember rowTy
        ty)

    // Add the type to the namespace.
    do this.AddNamespace(ns, [csvTy])
```

Обратите внимание на следующие моменты реализации:

- Перегруженные конструкторы позволяют считывать либо исходный файл, либо тот, который имеет идентичную схему. Этот шаблон часто используется при написании поставщика типов для локальных

или удаленных источников данных, и этот шаблон позволяет использовать локальный файл в качестве шаблона для удаленных данных.

- Значение TypeProviderConfig , передаваемое в конструктор поставщика типов, можно использовать для разрешения относительных имен файлов.
- Для определения расположения предоставленных свойств можно использовать метод `AddDefinitionLocation`. Поэтому, если вы используете `Go To Definition` для указанного свойства, CSV-файл будет открыт в Visual Studio.
- Тип `ProvidedMeasureBuilder` можно использовать для поиска единиц СИ и создания соответствующих типов `float<_>`.

Ключевые уроки

В этом разделе описано, как создать поставщик типов для локального источника данных с простой схемой, содержащейся в самом источнике данных.

Дальнейшие переходы

В следующих разделах приведены рекомендации для дальнейшего изучения.

Взгляните на скомпилированный код для удаленных типов

Чтобы получить представление о том, как использование поставщика типов соответствует выданному коду, рассмотрим следующую функцию, используя `HelloWorldTypeProvider`, которая используется ранее в этом разделе.

```
let function1 () =
    let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")
    obj1.InstanceProperty
```

Ниже приведен образ результирующего кода, декомпилированного с помощью Ildasm. exe:

```
.class public abstract auto ansi sealed Module1
extends [mscorlib]System.Object
{
.custom instance void [FSharp.Core]Microsoft.FSharp.Core.CompilationMappingAtt
ribute::.ctor(valuetype [FSharp.Core]Microsoft.FSharp.Core.SourceConstructFlags)
= ( 01 00 07 00 00 00 00 00 )
.method public static int32  function1() cil managed
{
// Code size       24 (0x18)
.maxstack  3
.locals init ([0] object obj1)
IL_0000:  nop
IL_0001:  ldstr      "some data"
IL_0006:  unbox.any  [mscorlib]System.Object
IL_000b:  stloc.0
IL_000c:  ldloc.0
IL_000d:  call       !!0 [FSharp.Core_2]Microsoft.FSharp.Core.LanguagePrimit
ives/IntrinsicFunctions::UnboxGeneric<string>(object)
IL_0012:  callvirt   instance int32 [mscorlib_3]System.String::get_Length()
IL_0017:  ret
} // end of method Module1::function1

} // end of class Module1
```

Как показано в примере, все упоминания типа `Type1` и свойство `InstanceProperty` были удалены, в результате чего в них будут находиться только операции с задействованными типами среды выполнения.

Соглашения о проектировании и именовании для поставщиков типов

При создании поставщиков типов Обратите внимание на следующие соглашения.

Поставщики протоколов подключения Как правило, имена большинства библиотек DLL поставщика для протоколов подключения к данным и служб, таких как подключения OData или SQL, должны заканчиваться `TypeProvider` или `TypeProviders`. Например, используйте имя библиотеки DLL, которое напоминает следующую строку:

`Fabrikam.Management.BasicTypeProviders.dll`

Убедитесь, что предоставленные типы являются членами соответствующего пространства имен, и укажите протокол подключения, который вы реализовали:

```
Fabrikam.Management.BasicTypeProviders.WmiConnection<…>
Fabrikam.Management.BasicTypeProviders.DataProtocolConnection<…>
```

Поставщики служебных программ для общего программирования. Для поставщика типов служебной программы, например для регулярных выражений, поставщик типов может быть частью базовой библиотеки, как показано в следующем примере:

```
#r "Fabrikam.Core.Text.Utilities.dll"
```

В этом случае предоставленный тип будет отображаться в соответствующей точке в соответствии с обычными соглашениями проектирования .NET:

```
open Fabrikam.Core.Text.RegexTyped

let regex = new RegexTyped<"a+b+a+b+">()
```

Одноэлементные источники данных. Некоторые поставщики типов подключаются к одному выделенному источнику данных и предоставляют только данные. В этом случае следует удалить суффикс `TypeProvider` и использовать обычные соглашения для именования .NET:

```
#r "Fabrikam.Data.Freebase.dll"

let data = Fabrikam.Data.Freebase.Astronomy.Asteroids
```

Дополнительные сведения см. в описании `GetConnection`ного соглашения о проектировании, которое описано далее в этом разделе.

Конструктивные шаблоны для поставщиков типов

В следующих разделах описываются шаблоны разработки, которые можно использовать при создании поставщиков типов.

Шаблон проектирования соединения

Большинство поставщиков типов должны быть написаны для использования шаблона `GetConnection`, используемого поставщиками типов в FSharp. Data. TypeProviders. dll, как показано в следующем примере:

```
#r "Fabrikam.Data.WebDataStore.dll"

type Service = Fabrikam.Data.WebDataStore<…static connection parameters…>

let connection = Service.GetConnection(…dynamic connection parameters…)

let data = connection.Astronomy.Asteroids
```

Поставщики типов, которые поддерживают удаленные данные и службы

Перед созданием поставщика типов, обеспечиваемого удаленными данными и службами, необходимо учитывать ряд проблем, связанных с подключенным программированием. Ниже приведены рекомендации по следующим вопросам.

- сопоставление схем
- динамическое и недействительность при изменении схемы
- кэширование схемы
- асинхронные реализации операций доступа к данным
- Поддержка запросов, включая запросы LINQ
- учетные данные и проверка подлинности

В этом разделе больше не рассматриваются эти проблемы.

Дополнительные методы разработки

При написании собственных поставщиков типов может потребоваться использовать следующие дополнительные методы.

Создание типов и членов по запросу

API Провидедтипе имеет отложенные версии Аддмембер.

```
type ProvidedType =
    member AddMemberDelayed  : (unit -> MemberInfo)      -> unit
    member AddMembersDelayed : (unit -> MemberInfo list) -> unit
```

Эти версии используются для создания пробелов по запросу типов.

Предоставление типов массивов и создание экземпляров универсальных типов

Вы предоставили члены (чьи подписи включают типы массивов, типы ByRef и экземпляры универсальных типов), используя обычные `MakeArrayType`, `MakePointerType` и `MakeGenericType` в любом экземпляре Type, включая `ProvidedTypeDefinitions`.

> **NOTE**
>
> В некоторых случаях может потребоваться использовать вспомогательный метод в `ProvidedTypeBuilder.MakeGenericType`. Дополнительные сведения см. в документации по поставщику типов SDK.

Предоставление заметок единиц измерения

API Провидедтипес предоставляет вспомогательные методы для предоставления заметок мер. Например, чтобы указать тип `float<kg>`, используйте следующий код:

```
let measures = ProvidedMeasureBuilder.Default
let kg = measures.SI "kilogram"
let m = measures.SI "meter"
let float_kg = measures.AnnotateType(typeof<float>,[kg])
```

Чтобы указать тип `Nullable<decimal<kg/m^2>>`, используйте следующий код:

```
let kgpm2 = measures.Ratio(kg, measures.Square m)
let dkgpm2 = measures.AnnotateType(typeof<decimal>,[kgpm2])
let nullableDecimal_kgpm2 = typedefof<System.Nullable<_>>.MakeGenericType [|dkgpm2 |]
```

Доступ к локальным ресурсам проекта или локальному сценарию

Каждому экземпляру поставщика типов может быть предоставлено `TypeProviderConfig` значение во время создания. Это значение содержит "папку разрешения" для поставщика (то есть папка проекта для компиляции или каталога, содержащего скрипт), список сборок, на которые имеются ссылки, и другие сведения.

Недействительности

Поставщики могут создавать сигналы недействительности для уведомления F# языковой службы о том, что предположения схемы могли измениться. При возникновении недействительности типечекк выполняется повторно, если поставщик размещается в Visual Studio. Этот сигнал будет проигнорирован при размещении поставщика в F# интерактивном режиме или F# компиляторе (FSC. exe).

Кэширование сведений о схеме

Поставщики часто должны кэшировать доступ к сведениям о схеме. Кэшированные данные должны храниться с использованием имени файла, заданного статическим параметром или в виде пользовательских данных. Примером кэширования схемы является параметр `LocalSchemaFile` в поставщиках типов в сборке `FSharp.Data.TypeProviders`. В реализации этих поставщиков этот статический параметр направляет поставщику типов сведения о схеме в указанный локальный файл, а не на доступ к источнику данных по сети. Чтобы использовать кэшированные данные схемы, необходимо также задать для параметра static `ForceUpdate` значение `false`. Аналогичную методику можно использовать для включения доступа к данным в сети и автономном режиме.

Резервная сборка

При компиляции файла `.dll` или `.exe` файл. dll для созданных типов статически связывается с результирующей сборкой. Эта ссылка создается путем копирования определений типов промежуточного языка (IL) и всех управляемых ресурсов из резервной сборки в окончательную сборку. При использовании F# интерактивного файла резервная копия DLL не копируется, а загружается непосредственно в F# интерактивный процесс.

Исключения и диагностика из поставщиков типов

Все варианты использования всех членов предоставленных типов могут вызывать исключения. Во всех случаях, если поставщик типов создает исключение, компилятор узла сообщает об ошибке конкретному поставщику типов.

- Исключения поставщика типов не должны приводить к внутренним ошибкам компилятора.
- Поставщики типов не могут сообщать о предупреждениях.
- Если поставщик типов размещается в F# компиляторе, в F# среде разработки или F# интерактивном режиме, все исключения из этого поставщика перехватываются. Свойство Message всегда является текстом ошибки, и трассировка стека не отображается. Если будет создано исключение, можно создать следующие примеры: `System.NotSupportedException`, `System.IO.IOException``System.Exception`.

Предоставление созданных типов

На данный момент в этом документе было объяснено, как предоставить удаленные типы. Можно также использовать механизм поставщика типов в F# для предоставления сформированных типов, которые добавляются как реальные определения типов .NET в программу пользователя. Необходимо обращаться к созданным предоставленным типам с помощью определения типа.

```
open Microsoft.FSharp.TypeProviders

type Service = ODataService<"http://services.odata.org/Northwind/Northwind.svc/">
```

Вспомогательный код Провидедтипес-0,2, который является частью выпуска F# 3,0, имеет ограниченную поддержку для предоставления сформированных типов. Для созданного определения типа должны быть справедливы следующие инструкции:

- для `isErased` необходимо задать значение `false`.
- Созданный тип должен быть добавлен к вновь созданной `ProvidedAssembly()`, которая представляет контейнер для созданных фрагментов кода.
- Поставщик должен иметь сборку, имеющую фактический резервный файл .NET. dll с совпадающим DLL-файлом на диске.

Правила и ограничения

При записи поставщиков типов учитывайте следующие правила и ограничения.

Предоставленные типы должны быть доступны

Все предоставленные типы должны быть доступны из невложенных типов. Невложенные типы задаются в вызове конструктора `TypeProviderForNamespaces` или вызове `AddNamespace`. Например, если поставщик предоставляет тип `StaticClass.P : T`, необходимо убедиться, что T является либо невложенным типом, либо вложенным в него.

Например, некоторые поставщики имеют статический класс, например `DataTypes`, содержащие эти типы `T1, T2, T3, ...`. В противном случае ошибка говорит о том, что обнаружена ссылка на тип T в сборке A, но тип не найден в этой сборке. Если возникает эта ошибка, убедитесь, что все подтипы доступны из типов поставщиков. Примечание. Эти типы `T1, T2, T3...` называются типами *на лету* . Не забудьте разместить их в доступном пространстве имен или родительском типе.

Ограничения механизма поставщика типов

Механизм поставщика типов в F# имеет следующие ограничения.

- Базовая инфраструктура для поставщиков типов в F# не поддерживает предоставленные универсальные типы или предоставленные универсальные методы.
- Механизм не поддерживает вложенные типы со статическими параметрами.

Советы по разработке

Во время разработки могут оказаться полезными следующие советы:

Запуск двух экземпляров Visual Studio

Вы можете разработать поставщик типов в одном экземпляре и протестировать его в другом, так как тестовая среда IDE заблокирует DLL-файл, который предотвращает перестроение поставщика типов. Поэтому необходимо закрыть второй экземпляр Visual Studio, пока поставщик построен в первом экземпляре, а затем повторно открыть второй экземпляр после сборки поставщика.

Отладка поставщиков типов с помощью вызовов FSC. exe

Поставщики типов можно вызывать с помощью следующих средств:

- FSC. exe (компилятор F# командной строки)
- FSI. exe (F# интерактивный компилятор)
- devenv. exe (Visual Studio)

Часто отладку поставщиков типов можно упростить с помощью FSC. exe в файле скрипта теста (например, Script. fsx). Отладчик можно запустить из командной строки.

```
devenv /debugexe fsc.exe script.fsx
```

Можно использовать ведение журнала печати в stdout.

См. также

- Поставщики типов
- Пакет SDK поставщика типов

Безопасность поставщиков типов

23.10.2019 • 3 minutes to read • Edit Online

Поставщики типов являются сборки (DLL), ссылается на F# проект или скрипт, который содержит код для подключения к внешним источникам данных и область сведений о типах F# типа среды. Как правило, код в сборках, на которую указывает ссылка выполняется только после компиляции и затем выполните код (или в случае сценария, отправить код таким образом, чтобы F# интерактивное окно). Однако тип сборки поставщика выполняется внутри Visual Studio, если код просто просмотреть в редакторе. Это происходит, поскольку поставщики типов должны выполняться для добавления дополнительной информации в редактор, например подсказки с краткими сведениями, варианты завершения IntelliSense и т. д. Таким образом необходимо учитывать дополнительную безопасность для типа сборки поставщика, так как они автоматически применяются внутри процесса Visual Studio.

Диалоговое окно предупреждения безопасности

При использовании сборки поставщика определенного типа в первый раз, Visual Studio отображает диалоговое окно, предупреждающее о том, что поставщик типов сейчас будет запущена. Прежде, чем Visual Studio загружает поставщика типов, это дает возможность решить, если вы доверяете этого конкретного поставщика. Если вы доверяете источнику поставщика типа, затем выберите «Я доверяю данного поставщика типов.» Если вы не доверяете источнику поставщика типа, затем выберите «Я не доверяю данного поставщика типов.» Предоставление доверия поставщика позволяет запускать в Visual Studio и обеспечивает поддержку технологии IntelliSense и разрабатывать функции. Но если поставщик типов, сам окажется хакером, выполнение его кода может поставить под угрозу компьютер.

Если проект содержит код, ссылающийся на поставщиков типов, которые были выбраны в диалоговом окне не на доверие, то во время компиляции, компилятор сообщит об ошибке, указывающее, что поставщик типов не является доверенным. Все типы, которые зависят от поставщика недоверенных типов обозначены красной волнистой линией. Это безопасно, можно просмотреть в редакторе кода.

Если вы решите изменить параметр доверия непосредственно в Visual Studio, выполните следующие действия.

Чтобы изменить параметры доверия для поставщиков типов

1. На `Tools` меню, выберите `Options` и разверните `F# Tools` узла.
2. Выберите `Type Providers` и в списке поставщиков типов, установите флажок для поставщиков типов, вы доверяете и установите флажки для тех, вы не доверяете.

См. также

- Поставщики типов

Устранение неполадок поставщиков типов

23.10.2019 • 3 minutes to read • Edit Online

В этом разделе описаны и представлены возможные решения проблем, которые чаще всего возникают при использовании поставщиков типов.

Возможные проблемы с помощью поставщиков типов

Если возникла проблема при работе с поставщиками типов, можно просмотреть наиболее популярные решения, в следующей таблице.

ПРОБЛЕМА	ПРЕДЛАГАЕМЫЕ ДЕЙСТВИЯ
Изменения схемы. Тип поставщики работают лучше всего, если стабильна схемы источника данных. При добавлении данных таблицы или столбца, или внести еще одно изменение в эту схему, поставщик типов не распознает эти изменения.	Очистке или повторном построении проекта. Чтобы очистить проект, выберите **построения**, **Очистить** *имя_проекта* в строке меню. Чтобы перестроить проект, выберите **построения**, **Перестроить** *имя_проекта* в строке меню. Эти действия переустановка всех состояние поставщика типов и установка поставщика для подключения к источнику данных и получения обновленной схемы сведений.
Сбой подключения. Неверный URL-адрес или строка подключения, сеть отключена, или источник данных или служба недоступна.	Для веб-службы или службы OData вы можете попробовать URL-адреса в Internet Explorer, чтобы проверить правильность URL-адреса и служба доступна. Для строки подключения базы данных, можно использовать средства подключения данных в **обозревателя серверов** для проверки ли строка подключения является допустимым, и база данных доступна. После восстановления подключения, должны затем очистке или повторном построении проекта, таким образом, чтобы поставщик типа позволит приложению повторно подключиться к сети.
Учетные данные не является допустимым. Необходимо иметь допустимые разрешения для данных источника или веб-службы.	Для соединения SQL имя пользователя и пароль, указанные в файле строку или конфигурации подключения должен быть допустимым для базы данных. При использовании проверки подлинности Windows, необходимо иметь доступ к базе данных. Администратор базы данных можно определить разрешения, необходимые для доступа для каждой базы данных и каждого элемента в базе данных. Для веб-службы или службы данных необходимо иметь соответствующие учетные данные. Большинство поставщиков типов предоставляют объект DataContext, который содержит свойство учетные данные, которые можно задать с помощью соответствующего имени пользователя и ключ доступа.
Путь не является допустимым. Недопустимый путь к файлу.	Проверьте правильность пути и файл существует. Кроме того необходимо соответствующим образом Квота любой backlashes в пути или использовать буквальная строка или строку в тройных кавычках.

См. также

- Поставщики типов

Интерактивное программирование с помощью F#

04.11.2019 • 8 minutes to read • Edit Online

> **NOTE**
> Сейчас эта статья описывает только соответствующую процедуру для Windows. Позднее она будет переписана.

> **NOTE**
> Ссылка на справочник по API ведет на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

F# Interactive (fsi.exe) используется для интерактивного запуска кода F# в консоли или для выполнения скриптов F#. Другими словами, F# interactive выполняет команду REPL (чтение, оценка, цикл печати) для языка F#.

Чтобы открыть F# Interactive из консоли, запустите fsi.exe. Файл FSI. exe находится в:

```
C:\Program Files (x86)\Microsoft Visual Studio\2019\<sku>\Common7\IDE\CommonExtensions\Microsoft\FSharp
```

где `sku` имеет значение `Community`, `Professional` или `Enterprise`.

Дополнительные сведения о доступных параметрах командной строки см. в статье Параметры F# Interactive.

Чтобы запустить F# Interactive через Visual Studio, нажмите соответствующую кнопку на панели инструментов, обозначенную **F# Interactive**, или используйте сочетание клавиш **CTRL+ALT+F**. После этого откроется интерактивное окно — окно инструментов с запущенным сеансом F# Interactive. Вы также можете выбрать код, который нужно запустить в интерактивном окне, и нажать клавиши **ALT+ВВОД**. F# Interactive откроется в окне инструментов **F# Interactive**. При использовании этого сочетания клавиш, убедитесь, что фокус находится в редактора.

Независимо от того, пользуетесь вы консолью или Visual Studio, появится командная строка и интерпретатор будет ждать ввода данных. Можно ввести код так же, как это делается в файле кода. Для компиляции и выполнения кода введите две точки с запятой (**;;**), чтобы разграничить одну или несколько строк входных данных.

F# Interactive попытается скомпилировать код и, в случае успеха, выполняет код и напечатает сигнатуру типов и значений, которые он скомпилировал. В случае ошибки интерпретатор напечатает сообщения об ошибке.

Код, введенный в том же сеансе имеет доступ ко всем конструкциям, введенным прежде, так что вы можете надстраивать программы. В случае необходимости благодаря значительному объему буфера в окне инструментов можно скопировать код в файл.

При запуске в Visual Studio F# Interactive работает независимо от проекта, поэтому, например, использовать определенные в проекте конструкции в F# Interactive можно, только если скопировать код функции в интерактивное окно.

Если открыт проект, который ссылается на некоторые библиотеки, на эти же библиотеки можно сослаться в F# Interactive через **обозреватель решений**. Для ссылки на библиотеку в F# Interactive разверните узел **Ссылки**, откройте контекстное меню для библиотеки и выберите пункт **Отправить в F# Interactive**.

Управлять аргументами (параметрами) командной строки F# Interactive можно путем корректировки значений параметров. В меню **Сервис** выберите пункт **Параметры...** , а затем разверните узел **Инструменты F#** . Двумя параметрами, которые можно изменить, являются параметры F# Interactive и параметр **F# Interactive, 64-разрядная версия**, который используется только в том случае, если F# Interactive выполняется на 64-разрядном компьютере. Этот параметр указывает, следует ли выполнять выделенную 64-разрядную версию fsi.exe или fsianycpu.exe, которая использует архитектуру компьютера для определения типа выполняемого процесса — 32-разрядного или 64-разрядного.

Создание сценариев с помощью F#

Для скриптов используется расширение файла **FSX** или **FSSCRIPT**. Вместо компиляции исходного кода и последующего выполнения скомпилированной сборки можно просто запустить **fsi.exe** и указать имя файла скрипта исходного кода F#, а F# Interactive прочтет код и выполнит его в реальном времени.

Различия между интерактивной средой, средой скриптов и скомпилированной средой

При компиляции кода в F# Interactive независимо от того, запущен он интерактивно или для выполнения скрипта, определяется символ **INTERACTIVE**. При компиляции кода в компиляторе определяется символ **COMPILED**. Таким образом, если код должен различаться в режиме компиляции и в интерактивном режиме, можно воспользоваться директивами препроцессора для условной компиляции, чтобы определить, что именно использовать.

Некоторые директивы, доступные при выполнении скриптов в F# Interactive, недоступны при выполнении компилятора. В следующей таблице приведены директивы, доступные при использовании F# Interactive.

ДИРЕКТИВА	ОПИСАНИЕ
#help	Отображает сведения о доступных директивах.
#I	Задает путь поиска сборок в кавычках.
#load	Читает исходный файл, компилирует и запускает его.
#quit	Завершает сеанс F# Interactive.
#r	Ссылается на сборку.
#time ["on"|"off"]	Сама по себе директива **#time** включает или отключает отображение сведений о производительности. Когда она включена, F# Interactive измеряет реальное время, время ЦП и сведения о коллекции мусора для каждого интерпретированного и выполненного код раздела.

При указании файлов или путей в F# Interactive ожидается строковый литерал. Следовательно, файлы и пути должны быть заключены в кавычки; можно использовать обычные escape-символы. Кроме того, можно использовать символ @, чтобы чтобы строка, содержащая путь, интерпретировалась в F# Interactive как буквальная строка. В этом случае F# Interactive игнорирует все escape-символы.

Одно из различий между скомпилированным и интерактивным режимами заключается в том, как можно получить доступ к аргументам командной строки. В скомпилированном режиме используйте **System.Environment.GetCommandLineArgs**. В скриптах используйте **fsi.CommandLineArgs**.

Следующий код показывает, как создать функцию, которая читает аргументы командной строки в скрипте, и дать ссылку на другую сборку из скрипта. Первый файл кода, **MyAssembly.fs**, является кодом для сборки,

на которую дается ссылка. Скомпилируйте этот файл с помощью командной строки: **fsc -a MyAssembly.fs**, а затем запустите второй файл в качестве скрипта с помощью командной строки: **fsi --exec file1.fsx** test

```
// MyAssembly.fs
module MyAssembly
let myFunction x y = x + 2 * y
```

```
// file1.fsx
#r "MyAssembly.dll"

printfn "Command line arguments: "

for arg in fsi.CommandLineArgs do
    printfn "%s" arg

printfn "%A" (MyAssembly.myFunction 10 40)
```

Выходные данные выглядят следующим образом:

```
Command line arguments:
file1.fsx
test
90
```

См. также

ЗАГОЛОВОК	ОПИСАНИЕ
Параметры окна "Интерактивный F#"	Описание синтаксиса и параметров командной строки для F# интерактивного файла FSI. exe.
Справочные материалы по библиотеке F# Interactive	Описание функциональных возможностей библиотек, которые доступны при выполнении кода в F# Interactive.

Новые возможности в F# 4,7

01.12.2019 • 2 minutes to read • Edit Online

F#4,7 добавляет несколько улучшений в F# язык.

Начало работы

F#4,7 доступна во всех дистрибутивах .NET Core и средствах Visual Studio. Дополнительные сведения см. в статье Приступая к работе с F# .

Версия языка

Компилятор F# 4,7 вводит возможность установки действующей языковой версии с помощью свойства в файле проекта:

```
<PropertyGroup>
    <LangVersion>preview</LangVersion>
</PropertyGroup>
```

Для него можно задать значения `4.6` , `4.7` , `latest` и `preview` . Значение по умолчанию: `latest` .

Если задать для него значение `preview` , компилятор будет активировать все F# функции предварительной версии, реализованные в компиляторе.

Неявные yield

Больше не нужно применять ключевое слово `yield` в массивах, списках, последовательностям или вычислительных выражениях, где тип может быть определен. В следующем примере оба выражения требовали `yield` инструкции для каждой записи до F# 4,7:

```
let s = seq { 1; 2; 3; 4; 5 }

let daysOfWeek includeWeekend =
    [
        "Monday"
        "Tuesday"
        "Wednesday"
        "Thursday"
        "Friday"
        if includeWeekend then
            "Saturday"
            "Sunday"
    ]
```

Если вводится одно ключевое слово `yield` , к нему также должен быть применен `yield` .

Неявные yield не активируются при использовании в выражении, которое также использует `yield!` для преобразования последовательности в неявное значение. В таких случаях необходимо продолжать использовать `yield` в настоящее время.

Идентификаторы с подстановочными знаками

В F# коде, включающем классы, самоидентификатор должен всегда быть явным в объявлениях членов. Но в случаях, когда сам идентификатор никогда не используется, он традиционно использовался для обозначения безименных идентификаторов. Теперь можно использовать один символ подчеркивания:

```
type C() =
    member _.M() = ()
```

Это также относится к циклам `for`:

```
for _ in 1..10 do printfn "Hello!"
```

Ограничения отступов

До F# 4,7 требования к отступам для первичного конструктора и аргументов статических членов требовали чрезмерного отступа. Теперь для них требуется только одна область отступов:

```
type OffsideCheck(a:int,
    b:int, c:int,
    d:int) = class end

type C() =
    static member M(a:int,
        b:int, c:int,
        d:int) = 1
```

Новые возможности в F# 4,6

01.12.2019 • 2 minutes to read • Edit Online

F#4,6 добавляет несколько улучшений в F# язык.

Начало работы

F#4,6 доступна во всех дистрибутивах .NET Core и средствах Visual Studio. Дополнительные сведения см. в статье Приступая к работе с F# .

Анонимные записи

Анонимные записи — это новый F# тип, представленный F# в 4,6. Они представляют собой простые статистические выражения именованных значений, которые не нужно объявлять перед использованием. Их можно объявить как структуры или ссылочные типы. По умолчанию они являются ссылочными типами.

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

Они также могут быть объявлены как структуры для случаев, когда требуется группировать типы значений и работать в сценариях с учетом производительности.

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    struct {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

Они являются достаточно мощными и могут использоваться во многих сценариях. Дополнительные сведения см. в статье анонимные записи.

Функции Валуеоптион

Тип Валуеоптион, добавленный F# в 4,5, теперь имеет тип параметра, привязанный к модулю. Ниже приведены некоторые из наиболее часто используемых примеров.

```
// Multiply a value option by 2 if it has  value
let xOpt = ValueSome 99
let result = xOpt |> ValueOption.map (fun v -> v * 2)

// Reverse a string if it exists
let strOpt = ValueSome "Mirror image"
let reverse (str: string) =
    match str with
    | null
    | "" -> None
    | s ->
        str.ToCharArray()
        |> Array.rev
        |> string
        |> Some

let reversedString = strOpt |> Option.bind reverse
```

Это позволяет использовать Валуеоптион так же, как вариант в сценариях, где тип значения повышает производительность.

Новые возможности в F# 4,5

10.01.2020 • 6 minutes to read • Edit Online

F#4,5 добавляет несколько улучшений в F# язык. Многие из этих функций были добавлены вместе, что позволяет писать эффективный код в F# , а также обеспечивать безопасность этого кода. Это означает добавление нескольких концепций к языку и значительного объема анализа компилятора при использовании этих конструкций.

Приступая к работе

F#4,5 доступна во всех дистрибутивах .NET Core и средствах Visual Studio. Дополнительные сведения см. в статье Приступая к работе с F# .

Структуры Span и ByRef-Like

Тип Span<T>, введенный в .NET Core, позволяет представлять буферы в памяти строго типизированным способом, который теперь разрешен в F# начале с F# 4,5. В следующем примере показано, как можно повторно использовать функцию, работающую на Span<T> с различными представлениями буфера.

```
let safeSum (bytes: Span<byte>) =
    let mutable sum = 0
    for i in 0 .. bytes.Length - 1 do
        sum <- sum + int bytes.[i]
    sum

// managed memory
let arrayMemory = Array.zeroCreate<byte>(100)
let arraySpan = new Span<byte>(arrayMemory)

safeSum(arraySpan) |> printfn "res = %d"

// native memory
let nativeMemory = Marshal.AllocHGlobal(100);
let nativeSpan = new Span<byte>(nativeMemory.ToPointer(), 100)

safeSum(nativeSpan) |> printfn "res = %d"
Marshal.FreeHGlobal(nativeMemory)

// stack memory
let mem = NativePtr.stackalloc<byte>(100)
let mem2 = mem |> NativePtr.toVoidPtr
let stackSpan = Span<byte>(mem2, 100)

safeSum(stackSpan) |> printfn "res = %d"
```

Важным аспектом этого является то, что диапазон и другие структуры, подобные ByRef , имеют очень строгий статический анализ, выполняемый компилятором, который ограничивает их использование способами, которые могут оказаться неожиданными. Это фундаментальный компромисс между производительностью, выразительным и безопасностью, представленным в F# 4,5.

Пересмотренные ByRef

До F# 4,5, ByRef в F# были ненадежными и незвуковыми для многочисленных приложений. Проблемы с звуком в параметрах ByRef были устранены в F# 4,5, а также был применен один и тот же статический анализ

для структур, связанных с Span и ByRef.

инреф < > и аутреф < >

Чтобы представить понятие управляемого указателя, доступного только для чтения, только для записи и чтения и записи, F# 4,5 вводит `inref<'T>`, `outref<'T>` типы для представления указателей только для чтения и только для записи, соответственно. Каждый из них имеет разную семантику. Например, невозможно выполнить запись в `inref<'T>`:

```
let f (dt: inref<DateTime>) =
    dt <- DateTime.Now // ERROR - cannot write to an inref!
```

По умолчанию вывод типа будет выводить управляемые указатели как `inref<'T>` в виде неизменяемой природы F# кода, если только что не было объявлено как изменяющееся. Чтобы сделать запись доступной для записи, необходимо объявить тип как `mutable` перед передачей его адреса в функцию или член, управляющий этим элементом. Дополнительные сведения см. в разделе ByRef.

Структуры только для чтения

Начиная с F# 4,5 можно снабдить структуру аннотацией IsReadOnlyAttribute следующим образом:

```
[<IsReadOnly; Struct>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

Это не позволяет объявить изменяемый элемент в структуре и выдает метаданные, которые позволяют F# и C# обрабатывать его как ReadOnly при использовании из сборки. Дополнительные сведения см. в разделе структуры только для чтения.

Указатели void

Тип `voidptr` добавляется в F# 4,5, как в следующих функциях:

- `NativePtr.ofVoidPtr` преобразовать указатель void в собственный указатель int
- `NativePtr.toVoidPtr` преобразовать собственный указатель int в указатель void

Это полезно при взаимодействии с собственным компонентом, который использует указатели void.

Ключевое слово `match!`.

Ключевое слово `match!` улучшает сопоставление шаблонов в вычислительном выражении:

```
// Code that returns an asynchronous option
let checkBananaAsync (s: string) =
    async {
        if s = "banana" then
            return Some s
        else
            return None
    }

// Now you can use 'match!'
let funcWithString (s: string) =
    async {
        match! checkBananaAsync s with
        | Some bananaString -> printfn "It's banana!"
        | None -> printfn "%s" s
}
```

Это позволяет сократить код, который часто включает в себя смешение параметров (или других типов) с вычислительными выражениями, такими как async. Дополнительные сведения см. в разделе Match!.

Ослабленные требования к преобразованию в выражениях массивов, списков и последовательностей

Смешивание типов, которые могут наследовать друг от друга в выражениях массивов, списков и последовательностей, традиционно требовали преобразования любого производного типа в его родительский тип с `:>` или `upcast`. Теперь это неявное, как показано ниже.

```
let x0 : obj list  = [ "a" ] // ok pre-F# 4.5
let x1 : obj list  = [ "a"; "b" ] // ok pre-F# 4.5
let x2 : obj list  = [ yield "a" :> obj ] // ok pre-F# 4.5

let x3 : obj list  = [ yield "a" ] // Now ok for F# 4.5, and can replace x2
```

Ослабление отступов для выражений массива и списка

До F# 4,5 требовалось слишком много отступов массивов и выражений списка при передаче в качестве аргументов в вызовы методов. Это больше не требуется:

```
module NoExcessiveIndenting =
    System.Console.WriteLine(format="{0}", arg = [|
        "hello"
    |])
    System.Console.WriteLine([|
        "hello"
    |])
```

Справочник по языку F#

10.01.2020 • 15 minutes to read • Edit Online

Этот раздел представляет собой ссылку на F# язык, язык программирования с несколькими парадигмами, предназначенный для .NET. Язык F# поддерживает функциональные, объектно-ориентированные и императивные модели программирования.

Токены F#

Следующая таблица содержит статьи, где приведены таблицы ключевых слов, символов и литералов, используемых в F# в качестве токенов.

ЗАГОЛОВОК	ОПИСАНИЕ
Справочник ключевых слов	Содержит ссылки на сведения обо всех ключевых словах языка F#.
Справочник символов и операторов	Содержит таблицу символов и операторов, которые используются в языке F#.
Литералы	Описывает синтаксис для литеральных значений в F# и способы указания сведений о типах для литералов F#.

Основные понятия языка F#

Следующая таблица содержит статьи, где описаны основные понятия языка.

ЗАГОЛОВОК	ОПИСАНИЕ
Функции	Функции являются основным элементом выполнения программы на любом языке программирования. Как и в других языках, функция F# имеет имя, может иметь параметры и принимать аргументы, а также имеет тело. F# также поддерживает конструкции функционального программирования, например, обработку функций как значений, использование неименованных функций в выражениях, объединение функций для образования новых функций, каррированные функции и неявное определение функций посредством частичного применения аргументов функции.
Типы языка F#	Описывает типы, которые используются в F#, а также способы их именования и описания.
Вывод типа	Описывает, как компилятор F# определяет типы значений, переменных, параметров и возвращаемых значений.
Автоматическое обобщение	Описывает универсальные конструкции в F#.

ЗАГОЛОВОК	ОПИСАНИЕ
Наследование	Описывает наследование, которое применяется для моделирования отношения "is-a" — подтипирования — в объектно-ориентированном программировании.
Члены	Описывает элементы типов объектов F#.
Параметры и аргументы	Описывает языковую поддержку для определения параметров и передачи аргументов в функции, методы и свойства. Содержит сведения о передаче по ссылке.
Перегрузка операторов	Описывает перегрузку арифметических операторов в классе или типе записи, а также на глобальном уровне.
Приведение и преобразование	Описывает поддержку для преобразований типов в F#.
Управление доступом	Описывает управление доступом в F#. Управление доступом означает объявление того, какие именно клиенты могут использовать определенные элементы программы, например типы, методы, функции и т. д.
Соответствие шаблону	Описывает шаблоны, представляющие собой правила преобразования входных данных, которые применяются в языке F# для сравнения данных с шаблоном, разложения данных на составные части или извлечения информации из данных различными способами.
Активные шаблоны	Описывает активные шаблоны. Активные шаблоны позволяют определять именованные разделы, на которые подразделяются входные данные. Активные шаблоны можно использовать для разложения данных в настраиваемом порядке для каждого раздела.
Утверждения	Описывает выражение `assert`, являющееся функцией отладки, которую можно использовать для тестирования выражения. При сбое в режиме отладки утверждение создает диалоговое окно системной ошибки.
Обработка исключений	Содержит сведения о поддержке обработки исключений в языке F#.
Атрибуты	Описывает атрибуты, позволяющие применять метаданные к программным конструкциям.
Управление ресурсами: ключевое слово `use`	Описывает ключевые слова `use` и `using`, позволяющие управлять инициализацией и освобождением ресурсов.

ЗАГОЛОВОК	ОПИСАНИЕ
Пространства имен	Описывает поддержку пространств имен в F#. С помощью пространства имен вы можете упорядочить код по областям соответствующей функциональности, подключив имя к группированию элементов программы.
Модули	Описывает модули. Модуль F# — это группирование кода F#, например значений, типов и значений функций, в программе F#. Код группирования в модулях объединяет связанный код и помогает избежать конфликтов имен в программе.
Объявления импорта: ключевое слово `open`	Описывает работу `open`. Объявление импорта указывает модуль или пространство имен, на элементы которого можно ссылаться без использования полного имени.
Сигнатуры	Описывает сигнатуры и их файлы. В файле сигнатур содержатся сведения об открытых сигнатурах набора элементов программы на языке F#, таких как типы, пространства имен и модули. Файл сигнатур можно использовать для указания доступности этих элементов программы.
Документация XML	Описывает создание файлов документации для комментариев к XML-документам. Вы можете создать документацию из комментариев к коду в F#, а также в других языках .NET.
Подробный синтаксис	Описывает синтаксис для конструкций F#, когда упрощенный синтаксис отключен. Подробный синтаксис указывается директивой `#light "off"` в верхней части файла кода.

Типы языка F#

Следующая таблица содержит статьи, где описаны типы, поддерживаемые языком F#.

ЗАГОЛОВОК	ОПИСАНИЕ
Значения	Описывает значения, которые являются величинами, имеющими конкретный тип. Значения могут быть целыми числами или числами с плавающей запятой, символами или текстом, списками, последовательностями, массивами, кортежами, размеченными объединениями, записями, типами классов или значениями функции.
Базовые типы	Описывает основные основные типы, используемые в F# языке. Кроме того, указывает соответствующие типы .NET и минимальное и максимальное значения для каждого из них.

ЗАГОЛОВОК	ОПИСАНИЕ
Тип единиц	Описывает тип `unit`, который указывает на отсутствие конкретного значения. Тип `unit` имеет только одно значение, которое выступает в качестве заполнителя, если другое значение не существует или не требуется.
Строки	Описывает строки в F#. Тип `string` представляет неизменяемый текст в виде последовательности символов Юникода. `string` — это псевдоним для `System.String` в .NET Framework.
Кортежи	Описывает кортежи, представляющие собой группирования неименованных, но упорядоченных значений, которые могут иметь разные типы.
Типы коллекций F#	Обзор типов функциональных коллекций F#, включая типы для массивов, списков, последовательностей (seq), карт и наборов.
Списки	Описывает списки. В языке F# список — это упорядоченная, неизменная серия элементов одного типа.
Параметры	Описывает тип параметра. Параметр в F# используется, когда значение может как существовать, так и не существовать. Параметр имеет базовый тип и может содержать значение этого типа либо не содержать никакого значения.
Последовательности	Описывает последовательности. Последовательность — это логический ряд элементов одного типа. Отдельные элементы последовательности вычисляются только при необходимости, поэтому представление может быть меньше указанного количества элементов литерала.
Массивы	Описывает массивы. Массивы — это изменяемые последовательности элементов данных одного типа, имеющие фиксированный размер и индексируемые с нуля.
Записи	Описывает записи. Записи представляют собой простые агрегаты именованных значений, которые могут иметь элементы.
Размеченные объединения	Описывает размеченные объединения, обеспечивающие поддержку значений, которые могут представлять один из множества именованных вариантов, каждый из которых может иметь разные значения и типы.
Перечисления	Описывает перечисления, которые являются типами, имеющими определенный набор именованных значений. Их можно использовать вместо литералов, чтобы сделать код более понятным и простым в обслуживании.

ЗАГОЛОВОК	ОПИСАНИЕ
Ссылочные ячейки	Описывает ссылочные ячейки, представляющие собой места хранения, которые позволяют создавать изменяющиеся переменные с семантикой ссылок.
Сокращенные формы типов	Описывает сокращенные формы типов, которые являются альтернативными именами типов.
Классы	Описывает классы, которые являются типами, представляющими объекты, которые могут иметь свойства, методы и события.
Структуры	Описывает структуры, представляющие собой компактный тип объекта, который может быть более эффективным, чем класс для типов, имеющих небольшое количество данных и простое поведение.
Интерфейсы	Описывает интерфейсы, которые определяют наборы связанных элементов, реализуемых другими классами.
Абстрактные классы	Описывает абстрактные классы, которые оставляют некоторые или все элементы нереализованными, чтобы реализации могли предоставляться производными классами.
Расширения типов	Описывает расширения типов, которые позволяют добавлять новые элементы в ранее определенный тип объекта.
Гибкие типы	Описывает гибкие типы. Заметка с гибким типом указывает на то, что параметр, переменная или значение имеют тип, который совместим с указанным типом, где совместимость определяется положением в объектно ориентированной иерархии классов или интерфейсов.
Делегаты	Описывает делегаты, которые представляют вызов функции в качестве объекта.
Единицы измерения	Описывает единицы измерения. Значения с плавающей запятой в языке F# могут иметь связанные единицы измерения, которые обычно используются для указания длины, объема, массы и т. д.
Поставщики типов	Описывает поставщики типов и предоставляет ссылки на пошаговые инструкции по использованию встроенных поставщиков типов для доступа к базам данных и веб-службам.

Выражения F#

Следующая таблица содержит список статей, описывающих выражения F#.

ЗАГОЛОВОК	ОПИСАНИЕ
Условные выражения: `if...then...else`	Описывает выражение `if...then...else`, которое выполняет различные ветви кода, а также дает разные значения при вычислении в зависимости от заданного логического выражения.
Выражения match	Описывает выражение `match`, которое позволяет управлять ветвлением за счет сравнения выражения с набором шаблонов.
Циклы: выражение `for...to`	Описывает выражение `for...to`, которое используется для циклической итерации по диапазону значений переменной цикла.
Циклы: выражение `for...in`	Описывает выражение `for...in` — циклическую конструкцию, которая используется для итерации по совпадениям с шаблоном в перечислимой коллекции, такой как выражение диапазона, последовательность, список, массив или другая конструкция, поддерживающая перечисление.
Циклы: выражение `while...do`	Описывает выражение `while...do`, используемое для итеративного выполнения (выполнения в цикле), пока заданное условие теста истинно.
Выражения объекта	Описывает выражения объектов, формирующие новые экземпляры динамически создаваемого анонимного типа объекта, который основан на существующем базовом типе, интерфейсе или наборе интерфейсов.
Отложенные выражения	Описывает отложенные выражения, которые являются вычислениями, которые не оцениваются немедленно, а оцениваются, когда фактически требуется результат.
Выражения вычисления	Описывает выражения вычисления в F#, обеспечивающие удобный синтаксис для записи вычислений, которые можно упорядочивать и комбинировать с помощью привязок и конструкций потока управления. Они позволяют предоставить удобный синтаксис для компонентов функционального программирования *monad*, которые можно использовать для управления данными и побочными эффектами в функциональных программах. Один из типов выражений вычисления — асинхронный рабочий процесс — обеспечивает поддержку асинхронных и параллельных вычислений. Дополнительные сведения см. в статье Асинхронные рабочие потоки.
Асинхронные рабочие потоки	Описывает асинхронные рабочие процессы — функцию языка, позволяющую писать асинхронный код практически аналогично синхронному.
Цитирование кода	Описывает цитирование кода — функцию языка, позволяющую программно создавать выражения кода F# и работать с ними.

ЗАГОЛОВОК	ОПИСАНИЕ
Выражения запросов	Описывает выражения запросов — функцию языка, которая реализует LINQ для F# и позволяет составлять запросы к источнику данных или перечислимой коллекции.

Конструкции, поддерживаемые компилятором

Следующая таблица содержит список статей, где описаны специальные конструкции, поддерживаемые компилятором.

РАЗДЕЛ	ОПИСАНИЕ
Параметры компилятора	Описывает параметры командной строки для компилятора F#.
Директивы компилятора	Описывает директивы процессора и компилятора.
Идентификаторы Source Line, File и Path	Описывает идентификаторы `__LINE__`, `__SOURCE_DIRECTORY__` и `__SOURCE_FILE__`, представляющие собой встроенные значения, которые позволяют получить доступ к номеру исходной строки, каталогу и имени файла в коде.

Справочные сведения о ключевых словах

25.11.2019 • 13 minutes to read • Edit Online

Этот раздел содержит ссылки на сведения обо всех F# ключевых словах языка.

F#Таблица ключевых слов

В следующей таблице показаны все F# ключевые слова в алфавитном порядке, а также краткие описания и ссылки на соответствующие разделы, содержащие дополнительные сведения.

КЛЮЧЕВОЕ СЛОВО	ССЫЛКА	ОПИСАНИЕ
`abstract`	Члены Абстрактные классы	Указывает метод, который либо не имеет реализации в типе, в котором он объявлен, либо является виртуальным и имеет реализацию по умолчанию.
`and`	Привязки `let` Записи Члены Ограничения	Используется в взаимно рекурсивных привязках и записях, в объявлениях свойств и с несколькими ограничениями на универсальные параметры.
`as`	Классы Соответствие шаблону	Используется для присвоения объекту текущего класса имени объекта. Также используется для присвоения имени целому шаблону в соответствии с шаблоном.
`assert`	Утверждения	Используется для проверки кода во время отладки.
`base`	Классы Наследование	Используется в качестве имени объекта базового класса.
`begin`	Подробный синтаксис	В подробном синтаксисе обозначает начало блока кода.
`class`	Классы	В подробном синтаксисе обозначает начало определения класса.
`default`	Члены	Указывает реализацию абстрактного метода; используется вместе с объявлением абстрактного метода для создания виртуального метода.
`delegate`	Делегаты	Используется для объявления делегата.

КЛЮЧЕВОЕ СЛОВО	ССЫЛКА	ОПИСАНИЕ
`do`	Привязки do Циклы: выражение `for...to` Циклы: выражение `for...in` Циклы: выражение `while...do`	Используется в циклических конструкциях или для выполнения императивного кода.
`done`	Подробный синтаксис	В подробном синтаксисе обозначает конец блока кода в цикле выражения.
`downcast`	Приведение и преобразование	Используется для преобразования в тип, который находится ниже в цепочке наследования.
`downto`	Циклы: выражение `for...to`	В выражении `for` используется при подсчете в обратную.
`elif`	Условные выражения: `if...then...else`	Используется в условном ветвлении. Краткая форма `else if`.
`else`	Условные выражения: `if...then...else`	Используется в условном ветвлении.
`end`	Структуры Размеченные объединения Записи Расширения типов Подробный синтаксис	В определениях типов и расширениях типов указывает конец раздела в определениях элементов. В подробном синтаксисе используется для указания конца блока кода, который начинается с ключевого слова `begin`.
`exception`	Обработка исключений Типы исключения	Используется для объявления типа исключения.
`extern`	Внешние функции	Указывает, что объявленный элемент программы определен в другом двоичном файле или сборке.
`false`	Типы-примитивы	Используется в качестве логического литерала.
`finally`	Исключения: выражение `try...finally`	Используется вместе с `try`, чтобы ввести блок кода, который выполняется независимо от того, произошло ли исключение.
`fixed`	Префикс	Используется для «закрепления» указателя на стеке, чтобы предотвратить сбор мусора.

КЛЮЧЕВОЕ СЛОВО	ССЫЛКА	ОПИСАНИЕ
`for`	Циклы: выражение `for...to` Циклы: выражение for...in	Используется в циклических конструкциях.
`fun`	Лямбда-выражения: ключевое слово `fun`	Используется в лямбда-выражениях, также известных как анонимные функции.
`function`	Выражения match Лямбда-выражения: ключевое слово Fun	Используется в качестве более короткой альтернативы ключевому слову `fun` и выражению `match` в лямбда-выражении, которое имеет сопоставление шаблонов для одного аргумента.
`global`	Пространства имен	Используется для ссылки на пространство имен .NET верхнего уровня.
`if`	Условные выражения: `if...then...else`	Используется в конструкциях условного ветвления.
`in`	Циклы: выражение for...in Подробный синтаксис	Используется для выражений последовательности и, в подробном синтаксисе, для разделения выражений от привязок.
`inherit`	Наследование	Используется для указания базового класса или базового интерфейса.
`inline`	Функции Встраиваемые функции	Используется для указания функции, которая должна быть интегрирована непосредственно в код вызывающего объекта.
`interface`	Интерфейсы	Используется для объявления и реализации интерфейсов.
`internal`	Управление доступом	Используется для указания того, что элемент виден внутри сборки, но не за его пределами.
`lazy`	Отложенные выражения	Используется для указания выражения, которое должно выполняться, только если требуется результат.
`let`	Привязки `let`	Используется для связывания или привязки имени со значением или функцией.

КЛЮЧЕВОЕ СЛОВО	ССЫЛКА	ОПИСАНИЕ
`let!`	Асинхронные рабочие потоки Выражения вычисления	Используется в асинхронных рабочих процессах для привязки имени к результату асинхронного вычисления или в других вычислительных выражениях, используемых для привязки имени к результату, который имеет тип вычисления.
`match`	Выражения match	Используется для ветвления путем сравнения значения с шаблоном.
`match!`	Выражения вычисления	Используется для встраивания вызова в вычислительное выражение и сопоставления шаблона с его результатом.
`member`	Члены	Используется для объявления свойства или метода в типе объекта.
`module`	Модули	Используется для связывания имени с группой связанных типов, значений и функций, чтобы логически отделить ее от другого кода.
`mutable`	Привязки let	Используется для объявления переменной, то есть значения, которое может быть изменено.
`namespace`	Пространства имен	Используется для связывания имени с группой связанных типов и модулей, чтобы логически отделить его от другого кода.
`new`	Конструкторы Ограничения	Используется для объявления, определения или вызова конструктора, который создает или может создать объект. Также используется в ограничениях универсальных параметров для указания того, что у типа должен быть определенный конструктор.
`not`	Справочник символов и операторов Ограничения	В действительности не является ключевым словом. Однако `not struct` в сочетании используется в качестве ограничения универсального параметра.
`null`	Значения NULL Ограничения	Указывает на отсутствие объекта. Также используется в ограничениях универсальных параметров.

КЛЮЧЕВОЕ СЛОВО	ССЫЛКА	ОПИСАНИЕ
`of`	Размеченные объединения Делегаты Типы исключения	Используется в размеченных объединениях для указания типа категорий значений, а также в объявлениях делегатов и исключений.
`open`	Объявления импорта: ключевое слово `open`	Используется, чтобы сделать содержимое пространства имен или модуля доступным без квалификации.
`or`	Справочник символов и операторов Ограничения	Используется с логическими условиями в качестве логического оператора `or`. Аналогично параметру `||`. Также используется в ограничениях элементов.
`override`	Члены	Используется для реализации версии абстрактного или виртуального метода, отличающегося от базовой версии.
`private`	Управление доступом	Разрешает доступ к элементу в коде в том же типе или модуле.
`public`	Управление доступом	Разрешает доступ к члену за пределами типа.
`rec`	Функции	Используется для указания, что функция является рекурсивной.
`return`	Асинхронные рабочие потоки Выражения вычисления	Используется для указания значения, которое будет предоставлено в результате вычисления выражения.
`return!`	Выражения вычисления Асинхронные рабочие потоки	Используется для указания вычислительного выражения, которое при вычислении предоставляет результат содержащего его вычислительного выражения.
`select`	Выражения запросов	Используется в выражениях запросов для указания полей или столбцов для извлечения. Обратите внимание, что это контекстное ключевое слово, означающее, что оно не является зарезервированным словом и действует как ключевое слово в соответствующем контексте.
`static`	Члены	Используется для указания метода или свойства, которые могут вызываться без экземпляра типа, или элемента значения, совместно используемого всеми экземплярами типа.

КЛЮЧЕВОЕ СЛОВО	ССЫЛКА	ОПИСАНИЕ
`struct`	Структуры Кортежи Ограничения	Используется для объявления типа структуры. Используется для указания кортежа структуры. Также используется в ограничениях универсальных параметров. Используется для совместимости OCaml в определениях модулей.
`then`	Условные выражения: `if...then...else` Конструкторы	Используется в условных выражениях. Также используется для выполнения побочных эффектов после создания объекта.
`to`	Циклы: выражение `for...to`	Используется в циклах `for` для обозначения диапазона.
`true`	Типы-примитивы	Используется в качестве логического литерала.
`try`	Исключения. Try... Выражение WITH Исключения. Try... Выражение finally	Используется для представления блока кода, который может создать исключение. Используется совместно с `with` или `finally`.
`type`	Типы языка F# Классы Записи Структуры Перечисления Размеченные объединения Сокращенные формы типов Единицы измерения	Используется для объявления класса, записи, структуры, размеченного объединения, типа перечисления, единицы измерения или аббревиатуры типа.
`upcast`	Приведение и преобразование	Используется для преобразования в тип, который находится выше в цепочке наследования.
`use`	Управление ресурсами: ключевое слово `use`	Используется вместо `let` для значений, требующих вызова `Dispose` для освобождения ресурсов.

КЛЮЧЕВОЕ СЛОВО	ССЫЛКА	ОПИСАНИЕ
`use!`	Выражения вычисления Асинхронные рабочие потоки	Используется вместо `let!` в асинхронных рабочих процессах и других вычислительных выражениях для значений, требующих вызова `Dispose` для освобождения ресурсов.
`val`	Явные поля. Ключевое слово `val` Сигнатуры Члены	Используется в сигнатуре для обозначения значения или в типе для объявления члена в ограниченных ситуациях.
`void`	Типы-примитивы	Указывает тип `void` .NET. Используется при взаимодействии с другими языками .NET.
`when`	Ограничения	Используется для логических условий (*Если*условия) соответствует шаблону и для ввода предложения ограничения для параметра универсального типа.
`while`	Циклы: выражение `while...do`	Вводит циклическую конструкцию.
`with`	Выражения match Выражения объекта Копирование и обновление выражений записей Расширения типов Исключения: выражение `try...with`	Используется вместе с ключевым словом `match` в выражениях сопоставления шаблонов. Также используется в выражениях объектов, в выражениях копирования и расширениях типов для представления определений элементов и для ввода обработчиков исключений.
`yield`	Списки, массивы, последовательности	Используется в выражении списка, массива или последовательности для получения значения для последовательности. Обычно можно опустить, так как в большинстве случаев это неявно.
`yield!`	Выражения вычисления Асинхронные рабочие потоки	Используется в вычислительном выражении для добавления результата заданного вычислительного выражения в коллекцию результатов для содержащего его выражения.

Следующие токены зарезервированы F# в, так как они являются ключевыми словами на языке OCaml:

- `asr`
- `land`
- `lor`
- `lsl`

- `lsr`
- `lxor`
- `mod`
- `sig`

Если используется параметр компилятора `--mlcompatibility`, указанные выше ключевые слова можно использовать в качестве идентификаторов.

Следующие токены зарезервированы в качестве ключевых слов для будущего F# расширения языка:

- `atomic`
- `break`
- `checked`
- `component`
- `const`
- `constraint`
- `constructor`
- `continue`
- `eager`
- `event`
- `external`
- `functor`
- `include`
- `method`
- `mixin`
- `object`
- `parallel`
- `process`
- `protected`
- `pure`
- `sealed`
- `tailcall`
- `trait`
- `virtual`
- `volatile`

См. также

- Справочник по языку F#
- Справочник символов и операторов
- Параметры компилятора

Справочник символов и операторов

04.11.2019 • 16 minutes to read • Edit Online

NOTE

Ссылки на справочник по API в этой статье ведут на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

В этом разделе приведена таблица символов и операторов, которые используются в языке F#.

Таблица символов и операторов

Следующая таблица содержит описание символов, которые используются в языке F#, ссылки на разделы с более подробной информацией и краткое описание возможностей использования этих символов. Символы упорядочены в соответствии с порядком кодировки ASCII.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`!`	Ссылочные ячейки Выражения вычисления	• Разыменовывает ссылочную ячейку. • Находясь после ключевого слова, указывает измененное поведение ключевого слова согласно контролю рабочим процессом.
`!=`	Неприменимо.	• Не используется в F#. Для операций неравенства используйте оператор `<>`.
`"`	Литералы Строки	• Разделяет текстовую строку.
`"""`	Строки	Разделяет буквальную текстовую строку. Отличается от `@"..."` тем, что можно указать символ кавычки с помощью одинарной кавычки в строке.
`#`	Директивы компилятора Гибкие типы	• Выступает как префикс директивы препроцессора или компилятора, например `#light`. • При использовании с типом указывает *гибкий тип*, который относится к типу или любому его производному типу.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`$`	Дополнительные сведения недоступны.	• Для внутреннего использования, применяется для определенных создаваемых компилятором имен переменных и функций.
`%`	Арифметические операторы Цитирование кода	• Вычисление целочисленного остатка. • Используется для объединения выражений в типизированные цитаты кода.
`%%`	Цитирование кода	• Используется для объединения выражений в нетипизированные цитаты кода.
`%?`	Операторы, допускающие значение NULL	• Выполняет вычисление целочисленного остатка, если правая часть является типом, допускающим значение null.
`&`	Выражения match	• Вычисляет адрес изменяемого значения, используется при взаимодействии с другими языками. • Используется в шаблонах И.
`&&`	Логические операторы	• Вычисляет значение логической операции И.
`&&&`	Побитовые операторы	• Вычисляет значение побитовой операции И.
`'`	Литералы Автоматическое обобщение	• Разделяет односимвольный литерал. • Указывает параметр универсального типа.
`` ``...`` ```	Дополнительные сведения недоступны.	• Разделяет идентификатор, который в противном случае не будет допустимым идентификатором, например ключевое слово языка.
`( )`	Тип единиц	• Представляет одно значение типа единицы.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`(...)`	Кортежи Перегрузка операторов	• Указывает порядок, в котором вычисляются выражения. • Разделяет кортеж. • Используется в определениях операторов.
`(*...*)`		• Разделяет комментарий, который может занимать несколько строк.
`(|...|)`	Активные шаблоны	• Разделяет активный шаблон. Также называются *скобками с вертикальной чертой*.
`*`	Арифметические операторы Кортежи Единицы измерения	• При использовании в качестве бинарного оператора перемножает левую и правую части. • В типах указывает на спаривание в кортеже. • Используется в единицах типов измерения.
`*?`	Операторы, допускающие значение NULL	• Перемножает левую и правую части, если правая часть является типом, допускающим значение NULL.
`**`	Арифметические операторы	• Вычисляет операцию возведения в степень (`x ** y` означает `x` в степени `y`).
`+`	Арифметические операторы	• При использовании в качестве бинарного оператора складывает левую и правую части. • При использовании в качестве унарного оператора обозначает положительное количество. (Формально выдает то же значение без изменения знака.)
`+?`	Операторы, допускающие значение NULL	• Складывает левую и правую части, если правая часть является типом, допускающим значение NULL.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`,`	Кортежи	• Отделяет элементы кортежа или параметры типа.
`-`	Арифметические операторы	• При использовании в качестве бинарного оператора вычитает из левой части правую. • При использовании в качестве унарного оператора выполняет операцию отрицания.
`-?`	Операторы, допускающие значение NULL	• Вычитает правую часть из левой, если правая часть является типом, допускающим значение NULL.
`->`	Функции Выражения match	• В типах функций разделяет аргументы и возвращаемые значения. • Получает результат выражения (в выражениях последовательности); эквивалент ключевому слову `yield`. • Используется в выражениях сопоставления
`.`	Члены Типы-примитивы	• Обращается к члену и отделяет индивидуальные имена в полном имени. • Задает десятичный разделитель в числах с плавающей запятой.
`..`	Циклы: выражение `for...in`	• Задает диапазон.
`.. ..`	Циклы: выражение `for...in`	• Задает диапазон вместе с приращением.
`.[...]`	Массивы	• Обращается к элементу массива.
`/`	Арифметические операторы Единицы измерения	• Делит левую часть (делимое) на правую часть (делитель). • Используется в единицах типов измерения.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`/?`	Операторы, допускающие значение NULL	• Делит левую часть на правую, если правая часть является типом, допускающим значение NULL.
`//`		• Обозначает начало однострочного комментария.
`///`	Документация XML	• Обозначает XML-комментарий.
`:`	Функции	• В аннотации типа отделяет имя параметра или члена от его типа.
`::`	Списки Выражения match	• Создает список. Элемент в левой части добавляется в начало списка в правой части. • Используется в сопоставлениях с шаблоном для отделения частей списка.
`:=`	Ссылочные ячейки	• Назначает значение ссылочной ячейке.
`:>`	Приведение и преобразование	• Преобразует тип в тип, находящийся на более высоком уровне иерархии.
`:?`	Выражения match	• Возвращает `true`, если значение соответствует заданному типу (в том числе к подтипу); в противном случае возвращает `false` (оператор теста типа).
`:?>`	Приведение и преобразование	• Преобразует тип в тип, находящийся на более низком уровне иерархии.
`;`	Подробный синтаксис Списки Записи	• Отделяет выражения (в основном используется в подробном синтаксисе). • Отделяет элементы списка. • Отделяет поля записи.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`<`	Арифметические операторы	• Вычисляет значение операции "меньше чем".
`<?`	Операторы, допускающие значение NULL	Вычисляет значение операции "меньше чем", если правая часть является типом, допускающим значение NULL.
`<<`	Функции	• Объединяет две функции в обратном порядке. Вторая функция выполняется первой (оператор обратного соединения).
`<<<`	Побитовые операторы	• Сдвигает биты в количестве в левой части влево на количество бит, указанное в правой части.
`<-`	Значения	• Присваивает значение переменной.
`<...>`	Автоматическое обобщение	• Разделяет параметры типа.
`<>`	Арифметические операторы	• Возвращает значение `true`, если левая часть не равна правой части; в противном случае возвращает значение false.
`<>?`	Операторы, допускающие значение NULL	• Вычисляет значение операции "не равно", если правая часть является типом, допускающим значение NULL.
`<=`	Арифметические операторы	• Возвращает значение `true`, если левая часть меньше или равна правой части; в противном случае возвращает значение `false`.
`<=?`	Операторы, допускающие значение NULL	• Вычисляет значение операции "меньше или равно", если правая часть является типом, допускающим значение NULL.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`<|`	Функции	• Передает результат выражения в правой части в функцию левой части (оператор обратного конвейера).
`<||`	Функция Operators.(<||)<'T1,'T2,'U>	• Передает кортеж из двух аргументов в правой части в функцию в левой части.
`<|||`	Функция Operators.(<|||) <'T1,'T2,'T3,'U>	• Передает кортеж из трех аргументов в правой части в функцию в левой части.
`<@...@>`	Цитирование кода	• Разделяет типизированную цитату кода.
`<@@...@@>`	Цитирование кода	• Разделяет нетипизированную цитату кода.
`=`	Арифметические операторы	• Возвращает значение `true`, если левая часть равна правой части; в противном случае возвращает значение `false`.
`=?`	Операторы, допускающие значение NULL	• Вычисляет значение операции "равно", если правая часть является типом, допускающим значение NULL.
`==`	Неприменимо.	• Не используется в F#. Используйте `=` для операции равенства.
`>`	Арифметические операторы	• Возвращает значение `true`, если левая часть больше правой части; в противном случае возвращает значение `false`.
`>?`	Операторы, допускающие значение NULL	• Выполняет операцию "больше, чем", если правая часть является типом, допускающим значение null.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`>>`	Функции	• Объединяет две функции (оператор прямого соединения).
`>>>`	Побитовые операторы	• Сдвигает биты в количестве в левой части влево на количество позиций, указанное в правой части.
`>=`	Арифметические операторы	• Возвращает `true`, если левая часть больше или равна правой части; в противном случае возвращает `false`.
`>=?`	Операторы, допускающие значение NULL	• Вычисляет значение операции "больше чем или равно", если правая часть является типом, допускающим значение NULL.
`?`	Параметры и аргументы	• Задает необязательный аргумент. • Используется как оператор для вызова динамических методов и свойств. Необходимо предоставить собственную реализацию.
`? ... <- ...`	Дополнительные сведения недоступны.	• Используется как оператор для настройки динамических свойств. Необходимо предоставить собственную реализацию.
`?>=`, `?>`, `?<=`, `?<`, `?=`, `?<>`, `?+`, `?-`, `?*`, `?/`	Операторы, допускающие значение NULL	• Эквивалент соответствующих операторов без суффикса ?, где тип, допускающий значение NULL, находится слева.
`>=?`, `>?`, `<=?`, `<?`, `=?`, `<>?`, `+?`, `-?`, `*?`, `/?`	Операторы, допускающие значение NULL	• Эквивалент соответствующих операторов без суффикса ?, где тип, допускающий значение NULL, находится справа.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`?>=?`, `?>?`, `?<=?`, `?<?`, `?=?`, `?<>?`, `?+?`, `?-?`, `?*?`, `?/?`	Операторы, допускающие значение NULL	• Эквивалент соответствующих операторов без окружающих вопросительных знаков, где обе стороны являются типами, допускающими значение NULL.
`@`	Списки Строки	• Объединяет два списка. • При размещении перед строковым литералом, указывает, что строку следует интерпретировать буквально, без интерпретации escape-символов.
`[...]`	Списки	• Разделяет элементы списка.
`[|...|]`	Массивы	• Разделяет элементы массива.
`[<...>]`	Атрибуты	• Разделяет атрибут.
`\`	Строки	• Указывает на пропуск следующего символа; используется в символьных и строковых литералах.
`^`	Статически разрешаемые параметры типов Строки	• Задает параметры типа, которые должны разрешаться во время компиляции, а не во время выполнения. • Объединяет строки.
`^^^`	Побитовые операторы	• Вычисляет значение операции побитового исключающего ИЛИ.
`_`	Выражения match Универсальные шаблоны	• Обозначает шаблон с подстановочными знаками. • Задает анонимный универсальный параметр.
`` ` ``	Автоматическое обобщение	• Для внутреннего использования, указывает параметр универсального типа.

СИМВОЛ ИЛИ ОПЕРАТОР	СВЯЗЕЙ	ОПИСАНИЕ
`{...}`	Последовательности Записи	• Разделяет выражения последовательности и вычислительные выражения. • Используется в определениях записей.
`|`	Выражения match	• Разделяет отдельные случаи соответствия, отдельные случаи размеченного объединения и значения перечисления.
`||`	Логические операторы	• Вычисляет значение логической операции ИЛИ.
`|||`	Побитовые операторы	• Вычисляет значение побитовой операции ИЛИ.
`|>`	Функции	• Передает результат левой части в функцию в правой части (оператор прямого конвейера).
`||>`	Функция Operators.(||>)<'T1,'T2,'U>	• Передает кортеж из двух аргументов в левой части в функцию в правой части.
`|||>`	Функция Operators.(|||>) <'T1,'T2,'T3,'U>	• Передает кортеж из трех аргументов в левой части в функцию в правой части.
`~~`	Перегрузка операторов	• Используется для объявления перегрузки для оператора унарного отрицания.
`~~~`	Побитовые операторы	• Вычисляет значение операции побитового НЕ.
`~-`	Перегрузка операторов	• Используется для объявления перегрузки для оператора унарного минуса.
`~+`	Перегрузка операторов	• Используется для объявления перегрузки для оператора унарного плюса.

Приоритет операторов

В следующей таблице указана очередность применения операторов и других ключевых слов выражений для языка F# (в порядке возрастания приоритета). Также, когда это возможно, указана ассоциативность.

ОПЕРАТОРА	АССОЦИАТИВНОСТЬ
`as`	Правый
`when`	Правый
`|` (вертикальная черта)	По левому краю
`;`	Правый
`let`	Неассоциативный
`function`, `fun`, `match`, `try`	Неассоциативный
`if`	Неассоциативный
`not`	Правый
`->`	Правый
`:=`	Правый
`,`	Неассоциативный
`or`, `||`	По левому краю
`&`, `&&`	По левому краю
`:>`, `:?>`	Правый
`<` *Op*, `>` *Op*, `=`, `|` *op*, `&` *Op*, `&` (включая `<<<`, `>>>`, `|||`, `&&&`)	По левому краю
`^` *op* (включая `^^^`)	Правый
`::`	Правый
`:?`	Не ассоциативен
`-` *op*, `+` *op*	Применяется к инфиксному использованию этих символов
`*` *op*, `/` *op*, `%` *op*	По левому краю
`**` *op*	Правый

ОПЕРАТОРА	АССОЦИАТИВНОСТЬ
`f x` (применение функции)	По левому краю
`|` (совпадение с шаблоном)	Правый
Префиксные операторы (`+` *op*, `-` *op*, `%`, `%%`, `&`, `&&`, `!` *op*, `~` *op*)	По левому краю
`.`	По левому краю
`f(x)`	По левому краю
`f<` *типы* `>`	По левому краю

F# поддерживает перегрузку пользовательских операторов. Это значит, что пользователь может определять собственные операторы. В предыдущей таблице *op* может быть любой допустимой (возможно, пустой) последовательностью символов операторов, встроенных или пользовательских. Таким образом, этой таблицей можно пользоваться, чтобы определить, какую последовательность символов использовать для пользовательского оператора с целью достижения нужного уровня приоритета. Начальные символы `.` игнорируются, когда компилятор определяет приоритет.

См. также

- Справочник по языку F#
- Перегрузка операторов

Арифметические операторы

23.10.2019 • 6 minutes to read • Edit Online

В этом разделе описываются арифметические операторы, доступные на F# языке.

Сводка бинарных арифметических операторов

В следующей таблице приведена сводка бинарных арифметических операторов, доступных для неупакованных целочисленных и типов с плавающей запятой.

БИНАРНЫЙ ОПЕРАТОР	ПРИМЕЧАНИЯ
`+` (сложение, плюс)	Unchecked. Возможное условие переполнения при добавлении чисел вместе с суммой, превышающей максимальное абсолютное значение, поддерживаемое типом.
`-` (вычитание, минус)	Unchecked. Возможное условие потери значимости при вычитании типов без знака или если значения с плавающей запятой слишком малы для представления типом.
`*` (умножение, время)	Unchecked. Возможное условие переполнения при умножении чисел и превышении максимального абсолютного значения, поддерживаемого типом.
`/` (деление на)	Деление на ноль приводит к DivideByZeroException созданию целочисленных типов. Для типов с плавающей запятой деление на ноль дает специальные значения `+Infinity` с плавающей запятой или. `-Infinity` Существует также возможное условие потери значимости, если число с плавающей запятой слишком мало для представления типом.
`%` (остаток, ОСТ)	Возвращает оставшуюся часть операции деления. Знак результата совпадает с знаком первого операнда.
`**` (возведение в степень)	Возможное условие переполнения, если результат превышает максимальное абсолютное значение для типа. Оператор возведения в степень работает только с типами с плавающей запятой.

Сводка унарных арифметических операторов

В следующей таблице перечислены унарные арифметические операторы, доступные для целочисленных и типов с плавающей запятой.

УНАРНЫЙ ОПЕРАТОР	ПРИМЕЧАНИЯ

УНАРНЫЙ ОПЕРАТОР	ПРИМЕЧАНИЯ
`+` минус	Может применяться к любому арифметическому выражению. Не изменяет знак значения.
`-` (отрицание, отрицательное значение)	Может применяться к любому арифметическому выражению. Изменяет знак значения.

Поведение в случае переполнения или потери значимости для целочисленных типов заключается в заключении в оболочку. Это приводит к неправильному результату. Целочисленное переполнение — это потенциально серьезная проблема, которая может повлиять на проблемы безопасности, когда программное обеспечение не записывается в учетную запись. Если это важно для вашего приложения, рассмотрите возможность использования операторов Checked в `Microsoft.FSharp.Core.Operators.Checked`.

Общие сведения об бинарных операторах сравнения

В следующей таблице показаны бинарные операторы сравнения, доступные для целочисленных и типов с плавающей запятой. Эти операторы возвращают значения типа `bool`.

Числа с плавающей запятой никогда не следует сравнивать непосредственно на равенство, так как представление с плавающей запятой IEEE не поддерживает точную операцию равенства. Два числа, которые можно легко проверить на равенство, при проверке кода могут иметь разные битовые представления.

ОПЕРАТОР	ПРИМЕЧАНИЯ
`=` (равенство, равно)	Это не оператор присваивания. Он используется только для сравнения. Это универсальный оператор.
`>` (больше)	Это универсальный оператор.
`<` (меньше)	Это универсальный оператор.
`>=` (больше или равно)	Это универсальный оператор.
`<=` (меньше или равно)	Это универсальный оператор.
`<>` (не равно)	Это универсальный оператор.

Перегруженные и универсальные операторы

Все операторы, описанные в этом разделе, определены в пространстве имен **Microsoft. FSharp. Core. Operators** . Некоторые операторы определяются с помощью статически разрешаемых параметров типа. Это означает, что существуют отдельные определения для каждого конкретного типа, который работает с этим оператором. Все унарные и Бинарные арифметические и битовые операторы находятся в этой категории. Операторы сравнения являются универсальными и поэтому работают с любым типом, а не только с примитивными арифметическими типами. Размеченные объединения и типы записей имеют собственные пользовательские реализации, создаваемые F# компилятором. Типы классов используют метод Equals.

Универсальные операторы являются настраиваемыми. Чтобы настроить функции сравнения, переопределите Equals для предоставления собственного пользовательского сравнения на равенство, а затем IComparableреализуйте. Интерфейс содержит один метод CompareTo , метод. System.IComparable

Операторы и определение типа

Использование оператора в выражении ограничивает вывод типа для этого оператора. Кроме того, использование операторов предотвращает автоматическую обобщение, поскольку использование операторов подразумевает арифметический тип. В отсутствие любой другой информации F# компилятор выводит `int` в качестве типа арифметических выражений. Это поведение можно переопределить, указав другой тип. Поэтому типы аргументов `function1` и возвращаемый тип в следующем коде выводятся `int` как, но типы для `function2` `float` выводятся как.

```
// x, y and return value inferred to be int
// function1: int -> int -> int
let function1 x y = x + y

// x, y and return value inferred to be float
// function2: float -> float -> float
let function2 (x: float) y = x + y
```

См. также

- Справочник символов и операторов
- Перегрузка операторов
- Побитовые операторы
- Логические операторы

Логические операторы

23.10.2019 • 2 minutes to read • Edit Online

В этом разделе описывается поддержка логических операторов в F# языка.

Сводка по логические операторы

В следующей таблице перечислены логические операторы, доступные в F# языка. Единственный тип, поддерживаемый эти операторы — `bool` типа.

ОПЕРАТОР	ОПИСАНИЕ
`not`	Логическое отрицание
`||`	Логическое или
`&&`	Логическое и

Логические и и или операторы выполнения *сокращенным вычислением*, то есть они вычисляют выражение справа от оператора, только в том случае, когда это необходимо для определения результата выражения. Второе выражение `&&` оператора определяется только в том случае, если первое выражение, результатом которого является `true`; второе выражение `||` оператора определяется только в том случае, если первое выражение, результатом которого является `false`.

См. также

- Побитовые операторы
- Арифметические операторы
- Справочник символов и операторов

Побитовые операторы

23.10.2019 • 2 minutes to read • Edit Online

В этом разделе описываются побитовые операторы, доступные в F# языка.

Побитовые операторы

В следующей таблице описываются побитовые операторы, которые поддерживаются для неупакованных целочисленных типов в F# языка.

ОПЕРАТОР	ПРИМЕЧАНИЯ
`&&&`	Побитовый оператор AND. Биты в результирующем имеют значение 1, только в том случае, если соответствующие биты в обоих операндах равны 1.
`|||`	Битовый оператор или. Биты в результате имеют значение 1, если один из соответствующих бита в источнике операндов равны 1.
`^^^`	Побитового исключающего или. Биты в результирующем имеют значение 1, только в том случае, если бит в операндах имеют разные значения.
`~~~`	Побитовый оператор отрицания. Это является унарным оператором и выдает результат, в котором все биты 0 исходного операнда преобразуются в 1, а все биты 1 преобразуются в 0 разрядов.
`<<<`	Битовый оператор сдвига влево. Результат — первый операнд с биты сдвинуты влево числа битов, во втором операнде. Биты, смещенные из наиболее значимых положения не перемещаются в позиции самого младшего. Младших битов заполняются нулями. Тип второго аргумента — `int32`.
`>>>`	Битовый оператор сдвига вправо. Результатом является первый операнд с битами, сдвигаются вправо на количество битов, во втором операнде. Биты, смещенные из наименее важного положения не перемещаются в наиболее важное положение. Для типов без знака старшие биты заполняются нулями. Для типов со знаком с отрицательными значениями старшие биты заполняются с хэшами. Тип второго аргумента — `int32`.

Следующие типы можно использовать с побитовые операторы: `byte`, `sbyte`, `int16`, `uint16`, `int32 (int)`, `uint32`, `int64`, `uint64`, `nativeint`, и `unativeint`.

См. также

- Справочник символов и операторов
- Арифметические операторы
- Логические операторы

Операторы, допускающие значение NULL

04.11.2019 • 6 minutes to read • Edit Online

Операторы, допускающие значение null, являются бинарными арифметическими или операторами сравнения, работающими с арифметическими типами, допускающими значение null, на одной Типы, допускающие значение null, часто возникают при работе с данными из таких источников, как базы данных, которые допускают значения NULL вместо фактических значений. Операторы, допускающие значения NULL, часто используются в выражениях запросов. В дополнение к операторам, допускающим значения NULL для арифметических операций и сравнения, операторы преобразования можно использовать для преобразования между типами, допускающими значение null. Кроме того, существуют версии определенных операторов запросов, допускающие значения NULL.

Таблица операторов, допускающих значения NULL

В следующей таблице перечислены операторы Nullable, F# поддерживаемые в языке.

NULLABLE ПО ЛЕВОМУ КРАЮ	ДОПУСКАЕТ ЗНАЧЕНИЯ NULL СПРАВА	ОБЕ СТОРОНЫ ДОПУСКАЮТ ЗНАЧЕНИЯ NULL
? > =	> =?	? > =?
? >	>?	? >?
? < =	< =?	? < =?
? <	<?	? <?
?=	=?	?=?
? < >	< >?	? < >?
?+	+?	?+?
?-	-?	?-?
?*	*?	?*?
?/	/?	?/?
?%	%?	?%?

Заметки

Операторы, допускающие значение null, включены в модуль функция nullableoperators в пространстве имен Microsoft. FSharp. LINQ. Тип данных, допускающий значение null, — `System.Nullable<'T>`.

В выражениях запросов типы, допускающие значение null, возникают при выборе данных из источника данных, который допускает значения NULL вместо значений. В SQL Server базе данных каждый столбец данных в таблице имеет атрибут, указывающий, разрешены ли значения NULL. Если разрешены значения NULL, данные, возвращаемые из базы данных, могут содержать значения NULL, которые не могут быть

представлены простым типом данных, таким как `int`, `float` и т. д. Таким образом, данные возвращаются в виде `System.Nullable<int>` вместо `int` и `System.Nullable<float>` вместо `float`. Фактическое значение можно получить из объекта `System.Nullable<'T>` с помощью свойства `Value`, а также определить, имеет ли объект `System.Nullable<'T>` значение, вызвав метод `HasValue`. Еще один полезный метод — метод `System.Nullable<'T>.GetValueOrDefault`, который позволяет получить значение или значение по умолчанию соответствующего типа. Значением по умолчанию является любая форма нулевого значения, например 0, 0,0 или `false`.

Типы, допускающие значение null, могут быть преобразованы в простые типы, не допускающие значения NULL, с помощью стандартных операторов преобразования, таких как `int` или `float`. Также можно преобразовать один тип, допускающий значение null, в другой тип, допускающий значение null, с помощью операторов преобразования для типов, допускающих значение null. Соответствующие операторы преобразования имеют те же имена, что и стандартные, но они находятся в отдельном модуле, модулем, допускающим значение NULL, в пространстве имен Microsoft. FSharp. LINQ. Как правило, это пространство имен открывается при работе с выражениями запросов. В этом случае можно использовать операторы преобразования, допускающие значения NULL, добавив префикс `Nullable.` к соответствующему оператору преобразования, как показано в следующем коде.

```
open Microsoft.FSharp.Linq

let nullableInt = new System.Nullable<int>(10)

// Use the Nullable.float conversion operator to convert from one nullable type to another nullable type.
let nullableFloat = Nullable.float nullableInt

// Use the regular non-nullable float operator to convert to a non-nullable float.
printfn "%f" (float nullableFloat)
```

В результате получается `10.000000`.

Операторы запроса для полей данных, допускающих значения NULL, таких как `sumByNullable`, также существуют для использования в выражениях запросов. Операторы запросов для типов, не допускающих значения NULL, не совместимы с типами, допускающими значение null, поэтому при работе со значениями данных, допускающими значение null, необходимо использовать версию соответствующего оператора запроса, допускающую значение null. Дополнительные сведения см. в разделе выражения запросов.

В следующем примере показано использование операторов Nullable в выражении F# запроса. В первом запросе показано, как создать запрос без оператора, допускающего значение null. второй запрос показывает эквивалентный запрос, использующий оператор, допускающий значение null. Полный контекст, включая настройку базы данных для использования этого примера кода, см. в разделе Пошаговое руководство. доступ к базе данных SQL с помощью поставщиков типов.

```
open System
open System.Data
open System.Data.Linq
open Microsoft.FSharp.Data.TypeProviders
open Microsoft.FSharp.Linq

[<Generate>]
type dbSchema = SqlDataConnection<"Data Source=MYSERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;">

let db = dbSchema.GetDataContext()

query {
    for row in db.Table2 do
    where (row.TestData1.HasValue && row.TestData1.Value > 2)
    select row
} |> Seq.iter (fun row -> printfn "%d %s" row.TestData1.Value row.Name)

query {
    for row in db.Table2 do
    // Use a nullable operator ?>
    where (row.TestData1 ?> 2)
    select row
} |> Seq.iter (fun row -> printfn "%d %s" (row.TestData1.GetValueOrDefault()) row.Name)
```

См. также

- Поставщики типов
- Выражения запросов

Функции

04.11.2019 • 14 minutes to read • Edit Online

Функции являются основным элементом выполнения программы на любом языке программирования. Как и в других языках, функция F# имеет имя, может иметь параметры и принимать аргументы, а также имеет тело. F# также поддерживает конструкции функционального программирования, например, обработку функций как значений, использование неименованных функций в выражениях, объединение функций для образования новых функций, каррированные функции и неявное определение функций посредством частичного применения аргументов функции.

Функции определяются с помощью ключевого слова `let`, или, если функция является рекурсивной, сочетания ключевых слов `let rec`.

Синтаксис

```
// Non-recursive function definition.
let [inline] function-name parameter-list [ : return-type ] = function-body
// Recursive function definition.
let rec function-name parameter-list = recursive-function-body
```

Заметки

Элемент *function-name* — это идентификатор, который представляет функцию. Элемент *parameter-list* состоит из последовательных параметров, разделенных пробелами. Вы можете указать явный тип для каждого параметра, как описано в разделе "Параметры". Если не указать конкретный тип аргумента, компилятор пытается определить тип из тела функции. Элемент *function-body* состоит из выражения. Выражение, представляющее тело функции, обычно является составным. Оно состоит из нескольких выражений, которые в результате дают итоговое выражение, являющееся возвращаемым значением. Элемент *return-type* представляет собой двоеточие, за которым следует тип, и является необязательным. Если вы явно не указываете тип возвращаемого значения, компилятор определяет тип возвращаемого значения из итогового выражения.

Простое определение функции выглядит следующим образом:

```
let f x = x + 1
```

В предыдущем примере `f` является именем функции, `x` — аргументом, имеющим тип `int`, `x + 1` является телом функции, а возвращаемое значение имеет тип `int`.

Функции могут быть помечены как `inline`. Сведения о `inline` см. в статье Встраиваемые функции.

Область

На любом уровне области, отличной от области модуля, не будет ошибкой повторно использовать значение или имя функции. При повторном использовании имя, объявленное позже, затемняет имя, объявленное раньше. Но в области верхнего уровня в модуле имена должны быть уникальными. Например, следующий код вызывает ошибку, когда отображается в области модуля, и не вызывает ее при отображении внутри функции:

```
let list1 = [ 1; 2; 3]
// Error: duplicate definition.
let list1 = []
let function1 =
   let list1 = [1; 2; 3]
   let list1 = []
   list1
```

Но следующий код допустим на любом уровне области:

```
let list1 = [ 1; 2; 3]
let sumPlus x =
// OK: inner list1 hides the outer list1.
   let list1 = [1; 5; 10]
   x + List.sum list1
```

Параметры

Имена параметров указываются после имени функции. Можно указать тип для параметра, как показано в следующем примере:

```
let f (x : int) = x + 1
```

Если тип указан, он стоит после имени параметра и отделяется от него двоеточием. Если тип параметра не указан, он определяется компилятором. Например, в следующем определении функции аргумент `x` определяется, как относящийся к типу `int`, так как 1 имеет тип `int`.

```
let f x = x + 1
```

Но компилятор попытается сделать функцию как можно более универсальной. Например, изучите следующий код:

```
let f x = (x, x)
```

Функция создает кортеж из одного аргумента любого типа. Так как тип не задан, функция может использоваться с любым типом аргумента. Дополнительные сведения см. в статье Автоматическое обобщение.

Тела функций

Тело функции может содержать определения локальных переменных и функций. Такие переменные и функции находятся в области действия внутри тела текущей функции, но не за его пределами. Если включен упрощенный синтаксис, нужно использовать отступ, чтобы указать, что определение находится внутри тела функции, как показано в следующем примере:

```
let cylinderVolume radius length =
    // Define a local value pi.
    let pi = 3.14159
    length * pi * radius * radius
```

Дополнительные сведения см. в статьях Рекомендации по форматированию кода и Подробный синтаксис.

Возвращаемые значения

Компилятор использует итоговое выражение в теле функции, чтобы определить возвращаемое значение и его тип. Компилятор может вывести тип итогового выражения из предыдущих выражений. В функции `cylinderVolume`, показанной в предыдущем разделе, тип `pi` определяется по типу литерала `3.14159` как `float`. Компилятор использует тип `pi`, чтобы определить тип выражения `h * pi * r * r` как `float`. Таким образом, общим типом возвращаемого значения функции является `float`.

Чтобы явно задать возвращаемое значение, составьте код следующим образом:

```
let cylinderVolume radius length : float =
   // Define a local value pi.
   let pi = 3.14159
   length * pi * radius * radius
```

В приведенном выше коде компилятор применяет **float** ко всей функции. Если требуется применить его и к типам параметров, используйте следующий код:

```
let cylinderVolume (radius : float) (length : float) : float
```

Вызов функции

Вы можете вызывать функции, указав имя функции, за которыми следует пробел, а затем аргументы, разделенные пробелами. Например, чтобы вызвать функцию **cylinderVolume** и назначить результат значению **vol**, составьте следующий код:

```
let vol = cylinderVolume 2.0 3.0
```

Частичное применение аргументов

Если число предоставленных аргументов меньше заданного их количества, создается новая функция, ожидающая оставшиеся аргументы. Такой метод работы с аргументами называется *каррингом* и характерен для языков функционального программирования, таких как F#. Предположим, вы работаете с двумя размерами труб: одна имеет радиус **2,0**, а другая — радиус **3,0**. Можно создать функции, определяющие объем трубы следующим образом:

```
let smallPipeRadius = 2.0
let bigPipeRadius = 3.0

// These define functions that take the length as a remaining
// argument:

let smallPipeVolume = cylinderVolume smallPipeRadius
let bigPipeVolume = cylinderVolume bigPipeRadius
```

Затем можно предоставить дополнительный аргумент для различных длин трубы двух разных размеров:

```
let length1 = 30.0
let length2 = 40.0
let smallPipeVol1 = smallPipeVolume length1
let smallPipeVol2 = smallPipeVolume length2
let bigPipeVol1 = bigPipeVolume length1
let bigPipeVol2 = bigPipeVolume length2
```

Рекурсивные функции

Рекурсивными называются функциями, которые вызывают сами себя. Для них нужно указать ключевое слово **rec**, следующее за ключевым словом **let**. Вызов рекурсивной функции в теле функции выполняется так же, как и для любой другой функции. Следующая рекурсивная функция выполняет вычисление n-го числа Фибоначчи. Последовательность чисел Фибоначчи известна с античных времен. В ней каждый последующий элемент равен сумме двух предыдущих чисел.

```
let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)
```

Некоторые рекурсивные функции могут переполнить стек программы или работать неэффективно, если при их написании не принимать меры предосторожности и не использовать специальные методики, такие как накапливаемые значения и продолжения.

Значения функции

В языке F# все функции считаются значениями. На самом деле они называются *значениями функции*. Так как функции являются значениями, они могут использоваться в качестве аргументов для других функций или в других контекстах, где применяются значения. Ниже приведен пример функции, которая принимает значение функции в качестве аргумента:

```
let apply1 (transform : int -> int ) y = transform y
```

Укажите тип значения функции с помощью токена `->`. В левой части токена находится тип аргумента, а в правой части — возвращаемое значение. В предыдущем примере `apply1` является функцией, которая принимает функцию `transform` как аргумент, где `transform` — это функция, которая принимает целое число и возвращает другое целое число. В следующем коде показано использование `apply1`:

```
let increment x = x + 1

let result1 = apply1 increment 100
```

После выполнения предыдущего кода `result` будет иметь значение 101.

Несколько аргументов разделяются токенами `->`, как показано в следующем примере:

```
let apply2 ( f: int -> int -> int) x y = f x y

let mul x y = x * y

let result2 = apply2 mul 10 20
```

Результат равен 200.

Лямбда-выражения

Лямбда-выражение — это неименованная функция. В предыдущих примерах вместо определения именованных функций **increment** и **mul** можно было воспользоваться лямбда-выражением:

```
let result3 = apply1 (fun x -> x + 1) 100

let result4 = apply2 (fun x y -> x * y ) 10 20
```

Лямбда-выражения определяются с помощью ключевого слова `fun`. Лямбда-выражение напоминает определение функции, за исключением того, что вместо токена `=` для отделения списка аргументов от тела функции используется токен `->`. Как и в обычном определении функции, типы аргументов могут выводиться или указываться явно, а тип возвращаемого значения лямбда-выражения выводится из типа последнего выражения в теле. Дополнительные сведения см. в статье Лямбда-выражения: ключевое слово `fun`.

Композиция и конвейеризация функций

В F# функции могут состоять из других функций. Композицией двух функций **function1** и **function2** является другая функция, которая представляет применение функции **function1** с последующим применением функции **function2**:

```
let function1 x = x + 1
let function2 x = x * 2
let h = function1 >> function2
let result5 = h 100
```

Результат равен 202.

Конвейеризация позволяет объединять вызовы функций в виде последовательных операций. Конвейеризация работает следующим образом:

```
let result = 100 |> function1 |> function2
```

Результат снова равен 202.

Операторы композиции принимают две функции и возвращают функцию. Операторы конвейера принимают функцию и аргумент и возвращают значение. В следующем примере кода показано различие между операторами конвейера и композиции. Там продемонстрировано отличие в сигнатурах и использовании функций.

```
// Function composition and pipeline operators compared.

let addOne x = x + 1
let timesTwo x = 2 * x

// Composition operator
// ( >> ) : ('T1 -> 'T2) -> ('T2 -> 'T3) -> 'T1 -> 'T3
let Compose2 = addOne >> timesTwo

// Backward composition operator
// ( << ) : ('T2 -> 'T3) -> ('T1 -> 'T2) -> 'T1 -> 'T3
let Compose1 = addOne << timesTwo

// Result is 5
let result1 = Compose1 2

// Result is 6
let result2 = Compose2 2

// Pipelining
// Pipeline operator
// ( |> ) : 'T1 -> ('T1 -> 'U) -> 'U
let Pipeline2 x = addOne x |> timesTwo

// Backward pipeline operator
// ( <| ) : ('T -> 'U) -> 'T -> 'U
let Pipeline1 x = addOne <| timesTwo x

// Result is 5
let result3 = Pipeline1 2

// Result is 6
let result4 = Pipeline2 2
```

Перегрузка функций

Можно перегружать методы типа, но не функции. Дополнительные сведения см. в статье Методы.

См. также

- Значения
- Справочник по языку F#

Привязки let

23.10.2019 • 8 minutes to read • Edit Online

Привязка связывает идентификатор со значением или функцией. Используйте `let` ключевое слово для привязки имени к значению или функции.

Синтаксис

```
// Binding a value:
let identifier-or-pattern [: type] =expressionbody-expression
// Binding a function value:
let identifier parameter-list [: return-type ] =expressionbody-expression
```

Примечания

`let` Ключевое слово используется в выражениях привязки для определения значений или значений функций для одного или нескольких имен. Простейшая форма `let` выражения привязывает имя к простому значению, как показано ниже.

```
let i = 1
```

Если выражение отделяется от идентификатора с помощью новой строки, необходимо задать отступ для каждой строки выражения, как показано в следующем коде.

```
let someVeryLongIdentifier =
    // Note indentation below.
    3 * 4 + 5 * 6
```

Вместо просто имени можно указать шаблон, содержащий имена, например кортеж, как показано в следующем коде.

```
let i, j, k = (1, 2, 3)
```

Выражение body — это выражение, в котором используются имена. Выражение тела отображается в отдельной строке с отступом до первого символа в `let` ключевом слове:

```
let result =

    let i, j, k = (1, 2, 3)

    // Body expression:
    i + 2*j + 3*k
```

`let` Привязка может отображаться на уровне модуля, в определении типа класса или в локальных областях, например в определении функции. Для `let` привязки на верхнем уровне модуля или в типе класса не требуется выражение тела, но на других уровнях области требуется выражение тела. Привязанные имена можно использовать после точки определения, но не в любой точке перед `let` отображением привязки, как показано в следующем коде.

```
// Error:
printfn "%d" x
let x = 100
// OK:
printfn "%d" x
```

Привязки функций

Привязки функций следуют правилам для привязок значений, за исключением того, что привязки функций включают имя функции и параметры, как показано в следующем коде.

```
let function1 a =
    a + 1
```

Как правило, параметры — это шаблоны, такие как шаблон кортежа:

```
let function2 (a, b) = a + b
```

Выражение `let` привязки принимает значение последнего выражения. Таким образом, в следующем примере кода значение `result` вычисляется из `100 * function3 (1, 2)` `300`, результатом вычисления которого является.

```
let result =
    let function3 (a, b) = a + b
    100 * function3 (1, 2)
```

См. дополнительные сведения о функциях.

Аннотации типов

Можно указать типы для параметров, добавив двоеточие (:) , за которым следует имя типа, заключенное в круглые скобки. Можно также указать тип возвращаемого значения, добавив двоеточие и тип после последнего параметра. Полные аннотации типа для `function1`, в качестве типов параметров которых есть целые числа, будут выглядеть следующим образом.

```
let function1 (a: int) : int = a + 1
```

Если нет явных параметров типа, для определения типов параметров функций используется определение типа. Это может включать автоматическое обобщение типа параметра как универсального.

Дополнительные сведения см. в разделе Автоматическое обобщение и Вывод типов.

Привязки let в классах

`let` Привязка может присутствовать в типе класса, но не в типе структуры или записи. Чтобы использовать привязку let в типе класса, класс должен иметь первичный конструктор. Параметры конструктора должны находиться после имени типа в определении класса. Привязка в типе класса определяет закрытые поля и члены для этого типа класса, а вместе с `do` привязками в типе формирует код для первичного конструктора для типа. `let` В следующих примерах кода показан класс `MyClass` с закрытыми `field1` полями `field2` и.

```
type MyClass(a) =
    let field1 = a
    let field2 = "text"
    do printfn "%d %s" field1 field2
    member this.F input =
        printfn "Field1 %d Field2 %s Input %A" field1 field2 input
```

Области `field1` и `field2` ограничены типом, в котором они объявляются. Дополнительные сведения см. `let` в разделе привязки в классах и классах.

Параметры типа в привязках let

`let` Привязка на уровне модуля, в типе или в вычислительном выражении может иметь явные параметры типа. Привязка let в выражении, например в определении функции, не может иметь параметры типа. Дополнительные сведения см. в статье Универсальные шаблоны.

Атрибуты в привязках let

Атрибуты могут применяться к привязкам верхнего уровня `let` в модуле, как показано в следующем коде.

```
[<Obsolete>]
let function1 x y = x + y
```

Область и доступность привязок let

Область сущности, объявленной с привязкой let, ограничена частью содержащей его области (например, функции, модуля, файла или класса) после отображения привязки. Таким образом, можно сказать, что привязка let вводит имя в область. В модуле значение, привязанное к let, может быть доступно клиентам модуля, если модуль доступен, так как привязки let в модуле компилируются в открытые функции модуля. В отличие от этого, привязки let в классе являются частными для класса.

Обычно функции в модулях должны уточняться именем модуля при использовании клиентского кода. Например, если модуль `Module1` содержит функцию `function1`, пользователи будут `Module1.function1` ссылаться на функцию.

Пользователи модуля могут использовать объявление импорта, чтобы сделать функции в этом модуле доступными для использования без уточнения по имени модуля. В этом примере пользователи модуля могут открыть модуль с помощью открытого `Module1` объявления импорта, а затем напрямую обратиться к `function1` нему.

```
module Module1 =
    let function1 x = x + 1.0

module Module2 =
    let function2 x =
        Module1.function1 x

open Module1

let function3 x =
    function1 x
```

Некоторые модули имеют атрибут рекуирекуалифиедакцесс, что означает, что предоставляемые им функции должны уточняться именем модуля. Например, у модуля F# List есть этот атрибут.

Дополнительные сведения о модулях и контроле доступа см. в разделе модули и Управление доступом.

См. также

- Функции
- Привязки `let` в классах

Привязки do

23.10.2019 • 2 minutes to read • Edit Online

Привязка `do` используется для выполнения кода без определения функции или значения. Кроме того, привязки `do` можно использовать в классах см привязки `do` в классах.

Синтаксис

```
[ attributes ]
[ do ]expression
```

Примечания

Используйте привязки `do`, если требуется выполнить код независимо от определения функции или значения. Выражение в привязке `do` должно возвращать значение `unit`. Код в привязке `do` верхнего уровня выполняется при инициализации модуля. Ключевое слово `do` является необязательным.

Атрибуты могут быть применены к привязке `do` верхнего уровня. Например, если программа использует COM-взаимодействия, может потребоваться применить в программе атрибут `STAThread`. Это можно сделать с помощью атрибута привязки `do`, как показано в следующем коде.

```
open System
open System.Windows.Forms

let form1 = new Form()
form1.Text <- "XYZ"

[<STAThread>]
do
   Application.Run(form1)
```

См. также

- Справочник по языку F#
- Функции

Лямбда-выражения: Ключевое слово FunF#()

23.10.2019 • 2 minutes to read • Edit Online

`fun` Ключевое слово используется для определения лямбда-выражения, то есть анонимной функции.

Синтаксис

```
fun parameter-list -> expression
```

Примечания

Список параметров обычно состоит из имен и (необязательно) типов параметров. В общем случае *параметр-List* может состоять из любых F# шаблонов. Полный список возможных шаблонов см. в разделе сопоставление шаблонов. Списки допустимых параметров включают следующие примеры.

```
// Lambda expressions with parameter lists.
fun a b c -> ...
fun (a: int) b c -> ...
fun (a : int) (b : string) (c:float) -> ...

// A lambda expression with a tuple pattern.
fun (a, b) -> …

// A lambda expression with a list pattern.
fun head :: tail -> …
```

Выражение является телом функции — последним выражением, которое создает возвращаемое значение. Примерами допустимых лямбда-выражений могут быть следующие:

```
fun x -> x + 1
fun a b c -> printfn "%A %A %A" a b c
fun (a: int) (b: int) (c: int) -> a + b * c
fun x y -> let swap (a, b) = (b, a) in swap (x, y)
```

Использование лямбда-выражений

Лямбда-выражения особенно полезны, когда требуется выполнить операции со списком или другой коллекцией и необходимо избежать дополнительной работы по определению функции. Многие F# библиотечные функции принимают значения функций в качестве аргументов, и в таких случаях может быть особенно удобно использовать лямбда-выражение. Следующий код применяет лямбда-выражение к элементам списка. В этом случае анонимная функция добавляет 1 к каждому элементу списка.

```
let list = List.map (fun i -> i + 1) [1;2;3]
printfn "%A" list
```

См. также

- Функции

Рекурсивные функции: Ключевое слово REC

23.10.2019 • 2 minutes to read • Edit Online

Ключевое слово используется вместе `let` с ключевым словом для определения рекурсивной функции. `rec`

Синтаксис

```
// Recursive function:
let rec function-nameparameter-list =
function-body

// Mutually recursive functions:
let rec function1-nameparameter-list =
function1-body
and function2-nameparameter-list =
function2-body
...
```

Примечания

Рекурсивные функции, которые вызывают сами себя, определяются явным образом на F# языке. Это делает определяемый идентификатор доступным в области действия функции.

В следующем коде показана рекурсивная функция, которая выполняет вычисление *n*-го числа Фибоначчи.

```
let rec fib n =
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
```

> **NOTE**
> На практике код, подобный приведенному выше, — непроизводительна памяти и процессорного времени, так как он включает перерасчет ранее вычисленных значений.

Методы неявно являются рекурсивными в пределах типа; нет необходимости добавлять `rec` ключевое слово. Привязки let в классах не являются неявно рекурсивными.

Взаимные рекурсивные функции

Иногда функции являются *взаимно рекурсивными*, то есть вызывают форму круга, где одна функция вызывает другую, которая, в свою очередь, вызывает первую, с любым числом вызовов между. Необходимо определить такие функции вместе в одной `let` привязке, `and` используя ключевое слово, чтобы связать их вместе.

В следующем примере показаны две взаимно рекурсивные функции.

```
let rec Even x =
    if x = 0 then true
    else Odd (x-1)
and Odd x =
    if x = 0 then false
    else Even (x-1)
```

См. также

- Функции

Точка входа

23.10.2019 • 2 minutes to read • Edit Online

В этом разделе описывается метод, используемый для задания точки входа в F# программу.

Синтаксис

```
[<EntryPoint>]
let-function-binding
```

Примечания

В предыдущем синтаксисе *let-Function-Binding* является определением функции в `let` привязке.

Точкой входа для программы, компилируемой как исполняемый файл, является место, где выполнение формально начинается. Вы указываете точку входа для F# приложения, применяя `EntryPoint` атрибут `main` к функции программы. Эта функция (созданная с помощью `let` привязки) должна быть последней функцией в последнем скомпилированном файле. Последний скомпилированный файл — это последний файл в проекте или последний файл, который передается в командную строку.

Функция точки входа имеет тип `string array -> int`. Аргументы, указанные в командной строке, передаются `main` в функцию в массиве строк. Первый элемент массива является первым аргументом; Имя исполняемого файла не включается в массив, так как оно находится на других языках. Возвращаемое значение используется в качестве кода выхода для процесса. Ноль обычно указывает на успешное выполнение; ненулевые значения указывают на ошибку. Не существует соглашения для конкретного значения ненулевых кодов возврата; значения кодов возврата зависят от конкретного приложения.

В следующем примере показана простая `main` функция.

```
[<EntryPoint>]
let main args =
    printfn "Arguments passed to function : %A" args
    // Return 0. This indicates success.
    0
```

При выполнении этого кода с помощью командной строки `EntryPoint.exe 1 2 3` выходные данные выглядят следующим образом.

```
Arguments passed to function : [|"1"; "2"; "3"|]
```

Неявная точка входа

Если у программы нет атрибута **EntryPoint** , который явно указывает на точку входа, то в качестве точки входа используются привязки верхнего уровня в последнем компилируемом файле.

См. также

- Функции
- Привязки let

Внешние функции

23.10.2019 • 2 minutes to read • Edit Online

В этом разделе F# описывается языковая поддержка для вызова функций в машинном коде.

Синтаксис

```
[<DllImport( arguments )>]
extern declaration
```

Примечания

В приведенном выше синтаксисе *аргументы* представляют аргументы, передаваемые `System.Runtime.InteropServices.DllImportAttribute` атрибуту. Первым аргументом является строка, представляющая имя библиотеки DLL, содержащей эту функцию, без расширения DLL. Дополнительные аргументы могут быть указаны для любого из открытых свойств `System.Runtime.InteropServices.DllImportAttribute` класса, например соглашения о вызовах.

Предположим, что у вас C++ есть собственная библиотека DLL, содержащая следующую экспортированную функцию.

```
#include <stdio.h>
extern "C" void __declspec(dllexport) HelloWorld()
{
    printf("Hello world, invoked by F#!\n");
}
```

Эту функцию можно вызвать из F# с помощью следующего кода.

```
open System.Runtime.InteropServices

module InteropWithNative =
    [<DllImport(@"C:\bin\natived11", CallingConvention = CallingConvention.Cdecl)>]
    extern void HelloWorld()

InteropWithNative.HelloWorld()
```

Взаимодействие с машинным кодом называется *вызовом* неуправляемого кода и является функцией среды CLR. Дополнительные сведения см. в разделе Взаимодействие с неуправляемым кодом. Сведения в этом разделе применимы к F#.

См. также

- Функции

Встраиваемые функции

23.10.2019 • 3 minutes to read • Edit Online

Встроенные функции — это функции, интегрированные непосредственно в вызывающий код.

Использование встроенных функций

При использовании статических параметров типа все функции, параметризованные параметрами типа, должны быть встроенными. Это гарантирует, что компилятор может разрешить эти параметры типа. При использовании обычных параметров универсального типа такое ограничение отсутствует.

Кроме включения использования ограничений элементов, встроенные функции могут быть полезны при оптимизации кода. Однако чрезмерное использование встроенных функций может привести к тому, что ваш код будет менее устойчив к изменениям в оптимизации компилятора и реализации библиотечных функций. По этой причине следует избегать использования встроенных функций для оптимизации, если не были предприняты другие методы оптимизации. Создание встроенной функции или метода иногда может повысить производительность, но это не всегда так. Поэтому следует также использовать измерения производительности, чтобы убедиться в том, что выполнение любой заданной функции имеет положительный результат.

`inline` Модификатор можно применять к функциям на верхнем уровне, на уровне модуля или на уровне метода в классе.

В следующем примере кода показана встроенная функция на верхнем уровне, встроенный метод экземпляра и встроенный статический метод.

```
let inline increment x = x + 1
type WrapInt32() =
    member inline this.incrementByOne(x) = x + 1
    static member inline Increment(x) = x + 1
```

Встроенные функции и вывод типов

Наличие `inline` влияет на вывод типа. Это обусловлено тем, что встроенные функции могут иметь статически разрешаемые параметры типа, тогда как невстроенные функции не могут. В следующем примере кода показан случай, когда `inline` используется функция с статически разрешаемым параметром типа `float` , оператором преобразования.

```
let inline printAsFloatingPoint number =
    printfn "%f" (float number)
```

Без модификатора вывод типа заставляет функцию принять конкретный тип, в данном случае `int` . `inline` Но с помощью `inline` модификатора функция также выводится для статического разрешаемого параметра типа. При использовании `inline` модификатора тип выводится следующим образом:

```
^a -> unit when ^a : (static member op_Explicit : ^a -> float)
```

Это означает, что функция принимает любой тип, поддерживающий преобразование в **float**.

См. также

- Функции
- Ограничения
- Статически разрешаемые параметры типов

Значения

23.10.2019 • 5 minutes to read • Edit Online

Значения в F# являются величинами, имеющими конкретный тип. Значения могут быть целыми числами или числами с плавающей запятой, символами или текстом, списками, последовательностями, массивами, кортежами, размеченными объединениями, записями, типами классов или значениями функции.

Привязка значения

Термин *привязка* означает сопоставление имени с определением. Ключевое слово `let` привязывает значение, как показано в следующих примерах:

```
let a = 1
let b = 100u
let str = "text"

// A function value binding.

let f x = x + 1
```

Тип значения выводится из определения. Для типа-примитива, такого как целое число или число с плавающей запятой, тип определяется типом литерала. Таким образом, в предыдущем примере компилятор определяет тип `b` как `unsigned int`, а тип `a` — как `int`. Тип значения функции определяется по возвращаемому значению в теле функции. Дополнительные сведения о типах значений функций см. в статье Функции. Дополнительные сведения о типах литералов см. в статье Литералы.

Компилятор не выдает диагностику о неиспользуемых привязках по умолчанию. Чтобы получить эти сообщения, включите в файле проекта предупреждение 1182 или при вызове компилятора (см `--warnon` . раздел параметры компилятора).

Для чего нужны неизменяемые значения?

Неизменяемые значения — это значения, которые не могут изменяться в течение всего выполнения программы. Если вы привыкли к таким языкам, как C++, Visual Basic или C#, вас может удивить, что в F# предпочтение отдается неизменяемым значениям, а не переменным, которым можно назначать новые значения во время выполнения программы. Неизменяемые данные являются важным элементом функционального программирования. В многопоточной среде управлять общими изменяемыми переменными, которые могут изменяться множеством разных потоков, довольно сложно. Кроме того, при работе с изменяемыми переменными иногда бывает трудно определить, может ли переменная изменяться при передаче в другую функцию.

В чисто функциональных языках переменные отсутствуют, а функции ведут себя строго как математические функции. Там, где код на процедурном языке присваивает переменную для изменения значения, эквивалентный код на функциональном языке использует неизменяемое значение, относящееся к входным данным, неизменяемую функцию и разные неизменяемые значения в качестве выходных данных. Такая математическая строгость позволяет точнее организовать работу программы. Это позволяет компиляторам строже проверять код и эффективнее оптимизировать его, а также помогает разработчикам читать и составлять код правильно. Таким образом, по сравнению с обычным процедурным кодом отладка функционального кода, скорее всего, будет выполнять проще.

Хотя F# и не является чисто функциональным языком, он полностью поддерживает возможности

функционального программирования. Применять неизменяемые значения крайне полезно, так как это позволяет коду использовать важные преимущества функционального программирования.

Изменяемые переменные

Вы можете использовать ключевое слово `mutable` для задания переменной, которую можно изменить. В общем случае изменяемые переменные в F# должны иметь ограниченную область действия, например в виде поля типа или локального значения. Изменяемыми переменными с ограниченной областью легче управлять, кроме того, для них ниже вероятность ошибочного изменения.

Вы можете присвоить изменяемой переменной начальное значение с помощью ключевого слова `let` точно так же, как и при определении значения. Отличие заключается в том, что позднее изменяемым переменным можно присваивать новые значения с помощью оператора `<-`, как показано в следующем примере.

```
let mutable x = 1
x <- x + 1
```

Значения, `mutable` помеченные как, могут `'a ref` автоматически передаваться в, если захватывается замыканием, включая формы `seq` , создающие замыкания, такие как построители. Если вы хотите получать уведомления в этом случае, включите предупреждение 3180 в файле проекта или при вызове компилятора.

См. также

ЗАГОЛОВОК	ОПИСАНИЕ
Привязки let	Предоставляет сведения об использовании `let` ключевого слова для привязки имен к значениям и функциям.
Функции	Содержит общие сведения о функциях в F#.

См. также

- Значения NULL
- Справочник по языку F#

Значения NULL

23.10.2019 • 5 minutes to read • Edit Online

В этом разделе описывается использование значения NULL в F#.

Значение null

Значение NULL обычно не используется в F# для значений или переменных. Однако в некоторых ситуациях значение null отображается как ненормальное. Если тип определен в F#, значение NULL не может использоваться как обычное значение, если атрибут AllowNullLiteral не применяется к типу. Если тип определен на другом языке .NET, значение null является возможным значением, а при взаимодействии с такими типами F# код может столкнуться со значениями NULL.

Для типа F# F#, определенного в и используемого исключительно в, единственным способом создания значения NULL с F# помощью библиотеки напрямую является использование unchecked . дефаултоф или Array. зерокреате. Однако для F# типа, используемого из других языков .NET, или при использовании этого типа с API, который не написан на F#языке, например .NET Framework, могут возникать значения NULL.

Можно использовать `option` тип в F# , если вы можете использовать ссылочную переменную с возможным значением NULL в другом языке .NET. Вместо значения NULL с F# `option` типом используется значение `None` параметра, если объект отсутствует. Значение `Some(obj)` параметра можно использовать с объектом `obj` при наличии объекта. Дополнительные сведения: Параметры. Обратите внимание, что вы по `null` - прежнему можете упаковать значение в параметр `Some x` , `x` если, для `null` , будет. Поэтому важно использовать `None` , когда значение равно `null` .

Ключевое слово является допустимым в F# языке и его необходимо использовать при работе с .NET Framework API или другими API, написанными на другом языке .NET. `null` При вызове API .NET и передаче значения NULL в качестве аргумента, а также при интерпретации возвращаемого значения или выходного параметра в вызове метода .NET могут потребоваться значения NULL.

Чтобы передать значение NULL в метод .NET, просто используйте `null` ключевое слово в вызывающем коде. Это показано в следующем примере кода.

```
open System

// Pass a null value to a .NET method.
let ParseDateTime (str: string) =
    let (success, res) = DateTime.TryParse(str, null, System.Globalization.DateTimeStyles.AssumeUniversal)
    if success then
        Some(res)
    else
        None
```

Чтобы интерпретировать значение null, полученное из метода .NET, используйте сопоставление шаблонов, если это возможно. В следующем примере кода показано, как использовать сопоставление шаблонов для интерпретации значения NULL, возвращаемого `ReadLine` при попытке чтения после конца входного потока.

```
// Open a file and create a stream reader.
let fileStream1 =
    try
        System.IO.File.OpenRead("TextFile1.txt")
    with
        | :? System.IO.FileNotFoundException -> printfn "Error: TextFile1.txt not found."; exit(1)

let streamReader = new System.IO.StreamReader(fileStream1)

// ProcessNextLine returns false when there is no more input;
// it returns true when there is more input.
let ProcessNextLine nextLine =
    match nextLine with
    | null -> false
    | inputString ->
        match ParseDateTime inputString with
        | Some(date) -> printfn "%s" (date.ToLocalTime().ToString())
        | None -> printfn "Failed to parse the input."
        true

// A null value returned from .NET method ReadLine when there is
// no more input.
while ProcessNextLine (streamReader.ReadLine()) do ()
```

Значения NULL для F# типов также могут создаваться другими способами, например при использовании `Array.zeroCreate` метода, который вызывает. `Unchecked.defaultof` Необходимо соблюдать осторожность при использовании такого кода, чтобы обеспечить инкапсуляцию значений NULL. В библиотеке, предназначенной только F#для, нет необходимости проверять значения NULL во всех функциях. При написании библиотеки для взаимодействия с другими языками .NET может потребоваться добавить проверки для входных параметров NULL и создать `ArgumentNullException` исключение, как в случае с C# кодом или Visual Basic.

Чтобы проверить, имеет ли произвольное значение null, можно использовать следующий код.

```
match box value with
| null -> printf "The value is null."
| _ -> printf "The value is not null."
```

См. также

- Значения
- Выражения match

Литералы

29.10.2019 • 4 minutes to read • Edit Online

NOTE

Справочные материалы по API в этой статье помогут вам перейти на MSDN (сейчас).

В этом разделе приводится таблица, в которой показано, как указать тип литерала F#в.

Типы литералов

В следующей таблице показаны типы литералов в F#. Символы, представляющие цифры в шестнадцатеричном формате, не учитывают регистр; символы, которые определяют тип, учитывают регистр.

TYPE	ОПИСАНИЕ	СУФФИКС ИЛИ ПРЕФИКС	ПРИМЕРЫ
sbyte	8-разрядное целое число со знаком	Y	`86y` `0b00000101y`
byte	8-разрядное натуральное число без знака	Uy	`86uy` `0b00000101uy`
int16	16-разрядное целое число со знаком	s	`86s`
uint16	16-разрядное натуральное число без знака	us	`86us`
int int32	32-разрядное целое число со знаком	l или нет	`86` `86l`
uint uint32	32-разрядное натуральное число без знака	u или UL	`86u` `86ul`
нативеинт	собственный указатель на натуральное число со знаком	n	`123n`
unativeint	собственный указатель как беззнаковое натуральное число	понижен	`0x00002D3Fun`
int64	64-разрядное целое число со знаком	L	`86L`

TYPE	ОПИСАНИЕ	СУФФИКС ИЛИ ПРЕФИКС	ПРИМЕРЫ
uint64	64-разрядное натуральное число без знака	UL	`86UL`
Single, float32	32-разрядное число с плавающей запятой	F или f	`4.14F` или `4.14f`
		возврата	`0x000000001f`
сделать Дважды	64-разрядное число с плавающей запятой	Нет	`4.14` , `2.3E+32` или `2.3e+32`
		ВОЗВРАТА	`0x0000000000000000LF`
bigint	целое число не ограничено 64-разрядным представлением	I	`99999999999999999999999999I`
decimal	дробное число, представленное в виде фиксированной точки или рационального числа	M или m	`0.7833M` или `0.7833m`
Char	символ Юникода	Нет	`'a'` или `'\u0061'`
Строковое	Строка Юникода	Нет	`"text\n"` или `@"c:\filename"` или `"""<book title="Paradise Lost">"""` или `"string1" + "string2"` См. также строки.
byte	Символ ASCII	C	`'a'B`
byte[]	Строка ASCII	C	`"text"B`
String или Byte []	буквальная строка	префикс @	`@"\\server\share"` (Юникод) `@"\\server\share"B` (ASCII)

Именованные литералы

Значения, которые должны быть константами, могут быть помечены атрибутом Literal . Этот атрибут

приводит к тому, что значение компилируется как константа.

В выражениях сопоставления шаблонов идентификаторы, начинающиеся с символов нижнего регистра, всегда обрабатываются как переменные для привязки, а не как литералы, поэтому при определении литералов обычно следует использовать начальные прописные буквы.

```
[<Literal>]
let SomeJson = """{"numbers":[1,2,3,4,5]}"""

[<Literal>]
let Literal1 = "a" + "b"

[<Literal>]
let FileLocation =   __SOURCE_DIRECTORY__ + "/" + __SOURCE_FILE__

[<Literal>]
let Literal2 = 1 ||| 64

[<Literal>]
let Literal3 = System.IO.FileAccess.Read ||| System.IO.FileAccess.Write
```

Заметки

Строки в Юникоде могут содержать явные кодировки, которые можно указать с помощью `\u`, за которым следует 16-разрядный шестнадцатеричный код (0000-FFFF) или кодировка UTF-32, которую можно указать с помощью `\U`, за которым следует 32-разрядный шестнадцатеричный код, представляющий любые символы Юникода. кодовая точка (00000000-0010FFFF).

Использование других побитовых операторов, отличных от `|||`, запрещено.

Целые числа в других базовых базах

32-разрядные целые числа со знаком можно также указывать в шестнадцатеричном, восьмеричном или двоичном формате с помощью префикса `0x`, `0o` или `0b` соответственно.

```
let numbers = (0x9F, 0o77, 0b1010)
// Result: numbers : int * int * int = (159, 63, 10)
```

Знаки подчеркивания в числовых литералах

Цифры можно разделять символом подчеркивания (`_`).

```
let value = 0xDEAD_BEEF

let valueAsBits = 0b1101_1110_1010_1101_1011_1110_1110_1111

let exampleSSN = 123_456_7890
```

См. также

- Класс Core. Литералаттрибуте

Типы языка F#

10.01.2020 • 8 minutes to read • Edit Online

В этом разделе описываются типы, используемые в F# , а также F# способы именования и описания типов.

Сводка по F# типам

Некоторые типы считаются *примитивными типами*, такими как логический тип `bool`, а также типы целочисленных и плавающих точек различных размеров, включая типы для байтов и символов. Эти типы описаны в типах-примитивах.

Другие типы, встроенные в язык, включают кортежи, списки, массивы, последовательности, записи и размеченные объединения. Если у вас есть опыт работы с другими языками .NET F#и они изучаются, следует ознакомиться с разделами для каждого из этих типов. Ссылки на дополнительные сведения об этих типах содержатся в подразделе « связанные разделы » этого раздела. Эти F#специальные типы поддерживают стили программирования, которые являются общими для функциональных языков программирования. Многие из этих типов имеют связанные модули в F# библиотеке, которые поддерживают общие операции с этими типами.

Тип функции включает сведения о типах параметров и возвращаемых типах.

.NET Framework является источником типов объектов, типов интерфейсов, типов делегатов и других. Вы можете определить собственные типы объектов так же, как и на любом другом языке .NET.

Кроме того F# , код может определять псевдонимы, представляющие собой именованные *сокращения типов*, которые являются альтернативными именами для типов. Аббревиатуры типов можно использовать, когда тип может измениться в будущем и необходимо избежать изменения кода, который зависит от типа. Также можно использовать аббревиатуру типа в качестве понятного имени для типа, который может облегчить чтение и понимание кода.

F#предоставляет полезные типы коллекций, разработанные с учетом функционального программирования. Использование этих типов коллекций помогает написать код, который более функциональн в стиле. Дополнительные сведения см. в разделе F# типы коллекций.

Синтаксис типов

В F# коде часто приходится записывать имена типов. Каждый тип имеет синтаксическую форму, и вы используете эти синтаксические формы в аннотациях типов, объявлениях абстрактных методов, объявлениях делегатов, сигнатурах и других конструкциях. При объявлении новой конструкции программы в интерпретаторе интерпретатор выводит имя конструкции и синтаксис для его типа. Этот синтаксис может быть только идентификатором для определяемого пользователем типа или встроенным идентификатором, например для `int` или `string`, но для более сложных типов синтаксис более сложен.

В следующей таблице показаны аспекты синтаксиса типов для F# типов.

ТИП	СИНТАКСИС ТИПА	ПРИМЕРЫ
тип-примитив	*имя типа*	`int` `float` `string`

ТИП	СИНТАКСИС ТИПА	ПРИМЕРЫ
агрегатный тип (класс, структура, объединение, запись, перечисление и т. д.)	*имя типа*	`System.DateTime` `Color`
Сокращенная форма типа	*Тип-сокращенное имя*	`bigint`
полный тип	*пространства имен. Type-Name* or *modules. Type-Name* or *пространства имен. modules. Type-Name*	`System.IO.StreamWriter`
array	*Type-Name*[] или *тип-имя* массива	`int[]` `array<int>` `int array`
двухмерный массив	*Type-Name*[,]	`int[,]` `float[,]`
трехмерный массив	*Type-Name*[,,]	`float[,,]`
tuple	*Type-name1 * Type-имя2 ...*	Например, `(1,'b',3)` имеет тип `int * char * int`
универсальный тип	*тип-параметр Generic-Type-Name* or *Generic-type-name<тип-parameter-list>*	`'a list` `list<'a>` `Dictionary<'key, 'value>`
сконструированный тип (универсальный тип с указанным аргументом типа)	*Argument-аргумент типа Generic-Type-Name* or *Generic-type-name<тип-argument-List>*	`int option` `string list` `int ref` `option<int>` `list<string>` `ref<int>` `Dictionary<int, string>`

ТИП	СИНТАКСИС ТИПА	ПРИМЕРЫ
тип функции с одним параметром	*параметр-type1 -> возвращаемого типа*	Функция, которая принимает `int` и возвращает `string` имеет тип `int -> string`
тип функции с несколькими параметрами	*параметр-type1 -> параметр-тип2 ->...-> return-Type*	Функция, которая принимает `int` и `float` и возвращает `string` типа `int -> float -> string`
функция с более высоким порядком в качестве параметра	(*функция-тип*)	`List.map` имеет тип `('a -> 'b) -> 'a list -> 'b list`
delegate	Делегат *функции-типа*	`delegate of unit -> int`
гибкий тип	*имя типа* #	`#System.Windows.Forms.Control` `#seq<int>`

См. также

РАЗДЕЛ	ОПИСАНИЕ
Типы-примитивы	Описывает встроенные простые типы, такие как целочисленные типы, логический тип и символьные типы.
Тип единиц	Описывает тип `unit`, тип, имеющий одно значение и обозначенный (); эквивалентно `void` в C# и `Nothing` в Visual Basic.
Кортежи	Описывает тип кортежа — тип, состоящий из связанных значений любого типа, сгруппированных по парам, Triple, четверных и т. д.
Параметры	Описывает тип параметра — тип, который может иметь значение или быть пустым.
Списки	Описывает списки, которые являются упорядоченными, неизменяемые последовательности элементов одного типа.
Массивы	Описывает массивы, которые представляют собой упорядоченные наборы изменяемых элементов того же типа, которые занимают непрерывный блок памяти и имеют фиксированный размер.
Последовательности	Описывает тип последовательности, представляющий логический ряд значений; отдельные значения вычисляются только по мере необходимости.
Записи	Описывает тип записи, небольшое статистическое выражение именованных значений.

РАЗДЕЛ	ОПИСАНИЕ
Размеченные объединения	Описывает тип размеченного объединения — тип, значения которого могут быть любого одного из набора возможных типов.
Функции	Описывает значения функций.
Классы	Описывает тип класса — тип объекта, соответствующий ссылочному типу .NET. Типы классов могут содержать члены, свойства, реализованные интерфейсы и базовый тип.
Структуры	Описывает тип `struct` — тип объекта, соответствующий типу значения .NET. Тип `struct` обычно представляет небольшое статистическое выражение данных.
Интерфейсы	Описывает типы интерфейсов, представляющие набор элементов, которые предоставляют определенные функциональные возможности, но не содержат данных. Тип интерфейса должен быть реализован типом объекта, чтобы быть полезным.
Делегаты	Описывает тип делегата, который представляет функцию в виде объекта.
Перечисления	Описывает типы перечислений, значения которых принадлежат набору именованных значений.
Атрибуты	Описывает атрибуты, которые используются для указания метаданных для другого типа.
Типы исключения	Описывает исключения, которые указывают сведения об ошибке.

Вывод типа

23.10.2019 • 4 minutes to read • Edit Online

В F# этом разделе описывается, как компилятор определяет типы значений, переменных, параметров и возвращаемых значений.

Определение типа в целом

Идея определения типа заключается в том, что нет необходимости указывать типы F# конструкций, за исключением случаев, когда компилятор не может вывести тип. Пропуск сведений о явном типе не означает, F# что является динамически типизированным языком или что значения F# в слабо типизированы. F#— Это язык со статической типизацией, который означает, что компилятор выводит точный тип для каждой конструкции во время компиляции. Если у компилятора недостаточно сведений для вывода типов каждой конструкции, необходимо предоставить дополнительные сведения о типе, как правило, путем добавления явных заметок типа в любое место кода.

Вывод типов параметров и возвращаемых данных

В списке параметров не нужно указывать тип каждого параметра. И, тем F# не менее, является языком со статической типизацией, поэтому во время компиляции каждое значение и выражение имеют определенный тип. Для тех типов, которые не указываются явным образом, компилятор выводит тип на основе контекста. Если тип не указан иным образом, он выводится как универсальный. Если код использует несогласованное значение, таким образом, что не существует единственного выводимого типа, удовлетворяющего всем условиям использования значения, компилятор сообщает об ошибке.

Тип возвращаемого значения функции определяется типом последнего выражения в функции.

Например, `a` в следующем коде типы параметров и `b` и возвращаемый тип выводятся `int` так, что литерал `100` имеет тип `int`.

```
let f a b = a + b + 100
```

Можно повлиять на вывод типа, изменив литералы. Если сделать `100` а `u` `b` ,добавив `a` суффикс, типы `uint32`, и возвращаемое значение будут выведены как. `uint32`

Также можно повлиять на вывод типа с помощью других конструкций, которые подразумевают ограничения на тип, например функции и методы, работающие только с определенным типом.

Кроме того, можно применить явные аннотации типа к параметрам функции или метода или переменным в выражениях, как показано в следующих примерах. Ошибки возникают при возникновении конфликтов между разными ограничениями.

```
// Type annotations on a parameter.
let addu1 (x : uint32) y =
    x + y

// Type annotations on an expression.
let addu2 x y =
    (x : uint32) + y
```

Можно также явно указать возвращаемое значение функции, указав аннотацию типа после всех

параметров.

```
let addu1 x y : uint32 =
   x + y
```

Типичный случай, когда аннотация типа полезен для параметра, — это то, что параметр является типом объекта и требуется использовать член.

```
let replace(str: string) =
    str.Replace("A", "a")
```

Автоматическое обобщение

Если код функции не зависит от типа параметра, компилятор считает, что параметр является универсальным. Это называется *автоматической*обобщением и может быть мощным средством для написания универсального кода без повышения сложности.

Например, следующая функция объединяет два параметра любого типа в кортеж.

```
let makeTuple a b = (a, b)
```

Тип выводится как

```
'a -> 'b -> 'a * 'b
```

Дополнительные сведения

Определение типа описывается более подробно в спецификации F# языка.

См. также

- Автоматическое обобщение

Базовые типы

23.10.2019 • 2 minutes to read • Edit Online

В этом разделе перечислены базовые типы, определенные в F# языка. Эти типы представляют собой фундаментальные в F#, являющееся основой практически все F# программы. Они являются подмножеством типов-примитивов .NET.

ТИП	ТИП .NET	ОПИСАНИЕ
`bool`	Boolean	Возможными значениями являются `true` и `false`.
`byte`	Byte	Значения от 0 до 255.
`sbyte`	SByte	Значения от -128 до 127.
`int16`	Int16	Значения от -32768 до 32767.
`uint16`	UInt16	Значения от 0 до 65535.
`int`	Int32	Значения от -2147483648 до 2147483647.
`uint32`	UInt32	Значения от 0 до 4 294 967 295.
`int64`	Int64	Значения от -9223372036854775808 до 9223372036854775807.
`uint64`	UInt64	Значения от 0 до 18446744073709551615.
`nativeint`	IntPtr	Собственный указатель в виде целого числа со знаком.
`unativeint`	UIntPtr	Собственный указатель в виде целого числа без знака.
`char`	Char	Символы Юникода.
`string`	String	Текст в Юникоде.
`decimal`	Decimal	С плавающей запятой в тип данных, по крайней мере 28 значащих цифр.

ТИП	ТИП .NET	ОПИСАНИЕ
`unit`	Неприменимо	Указывает на отсутствие значения. Тип имеет только одно значение, указав `()`. Значение единицы измерения, `()`, часто используется как заполнитель, в котором требуется значение, но нет реальные значения доступны или имеет смысл.
`void`	Void	Указывает без типа или значения.
`float32`, `single`	Single	32-разрядное число с плавающей запятой.
`float`, `double`	Double	64-разрядный тип с плавающей запятой.

NOTE

Позволяет выполнять вычисления с целыми числами слишком велик для 64-разрядному целочисленному типу с помощью bigint типа. `bigint` не является базовый тип; он представляет собой сокращение от `System.Numerics.BigInteger`.

См. также

- Справочник по языку F#

Тип Unit

04.11.2019 • 3 minutes to read • Edit Online

Тип `unit` — это тип, который указывает на отсутствие определенного значения; Тип `unit` имеет только одно значение, которое выступает в качестве заполнителя, если другие значения не существуют или не требуются.

Синтаксис

```
// The value of the unit type.
()
```

Заметки

Каждое F# выражение должно иметь значение. Для выражений, которые не создают значение, представляющее интерес, используется значение типа `unit`. Тип `unit` похож на тип `void` в таких языках, как C# и. C++

Тип `unit` имеет одно значение, и это значение обозначается токеном `()`.

Значение типа `unit` часто используется в F# программировании для хранения места, где значение является обязательным для синтаксиса языка, но если значение не требуется или необходимо. Примером может быть возвращаемое значение функции `printf`. Поскольку важные действия `printf` операции выполняются в функции, функция не должна возвращать фактическое значение. Поэтому возвращаемое значение имеет тип `unit`.

Некоторые конструкции предполагают `unit` значение. Например, ожидается, что для привязки `do` или любого кода на верхнем уровне модуля вычисляется значение `unit`. Компилятор сообщает о предупреждении, когда `do` привязка или код на верхнем уровне модуля выдает результат, отличный от неиспользуемого `unit`ого значения, как показано в следующем примере.

```
let function1 x y = x + y
// The next line results in a compiler warning.
function1 10 20
// Changing the code to one of the following eliminates the warning.
// Use this when you do want the return value.
let result = function1 10 20
// Use this if you are only calling the function for its side effects,
// and do not want the return value.
function1 10 20 |> ignore
```

Это предупреждение является характеристикой функционального программирования. Он не отображается в других языках программирования .NET. В чисто функциональной программе, в которой функции не имеют побочных эффектов, Последнее возвращаемое значение является единственным результатом вызова функции. Поэтому при игнорировании результата возможной ошибкой программирования. Несмотря F# на то, что не является чисто функциональным языком программирования, рекомендуется всегда следовать стилю функционального программирования, когда это возможно.

См. также

- Простого
- Справочник по языку F#

Строки

23.10.2019 • 6 minutes to read • Edit Online

NOTE

Ссылки на справочник по API в этой статье ведут на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

`string` Тип представляет неизменяемый текст как последовательность символов Юникода. `string` — это псевдоним для `System.String` в .NET Framework.

Примечания

Строковые литералы разделяются символом кавычки ("). Символ обратной косой \ черты () используется для кодирования определенных специальных символов. Обратная косая черта и следующий символ вместе называются *escape-последовательностью*. В следующей таблице показаны F# escape-последовательности, поддерживаемые в строковых литералах.

ЗНАК	ESCAPE-ПОСЛЕДОВАТЕЛЬНОСТЬ
Предупреждение	`\a`
Backspace	`\b`
Перевод страницы	`\f`
Новая строка	`\n`
Возврат каретки	`\r`
Вкладка	`\t`
Вертикальная табуляция	`\v`
Обратная косая черта	`\\`
Кавычки	`\"`
Заменяет	`\'`
символ Юникода	`\DDD` (где `D` обозначает десятичную цифру, диапазон 000-255 `\231` , например = "c")
символ Юникода	`\xHH` (где `H` обозначает шестнадцатеричную цифру; диапазон от 00 до FF; например `\xE7` , = "c")

ЗНАК	ESCAPE-ПОСЛЕДОВАТЕЛЬНОСТЬ
символ Юникода	`\uHHHH` (UTF-16) (где `H` обозначает шестнадцатеричную цифру, диапазон — 0000-FFFF; Например, `\u00E7` = "ç")
символ Юникода	`\U00HHHHHH` (UTF-32) (где `H` обозначает шестнадцатеричную цифру, диапазон 000000-10FFFF; Например, `\U0001F47D` = "👽")

IMPORTANT

`\DDD` Escape-последовательность — десятичная нотация, а не восьмеричная нотация, как в большинстве других языков. Таким образом, `8` цифры `9` и являются допустимыми `\032` , а последовательность представляет пробел (U + 0020), тогда как в восьмеричной нотации будет использоваться `\040` та же кодовая точка.

NOTE

Если ограничиться диапазоном 0-255 (0xFF), то `\DDD` `\x` escape-последовательности по сути являются набором символов ISO – 8859-1 , так как они соответствуют первым 256 кодовых позиций Юникода.

Если предшествует символ @, литерал является буквальной строкой. Это означает, что все escape-последовательности игнорируются, за исключением того, что две кавычки считаются символом одной кавычки.

Кроме того, строка может быть заключена в тройные кавычки. В этом случае все escape-последовательности игнорируются, включая двойные кавычки. Чтобы указать строку, содержащую внедренную строку в кавычках, можно использовать буквальную строку или строку с тройными кавычками. При использовании буквальной строки необходимо указать две кавычки, чтобы указать символ одинарной кавычки. При использовании строки, заключенной в тройные кавычки, можно использовать символы одинарной кавычки без синтаксического анализа в качестве конца строки. Этот метод может быть полезен при работе с XML или с другими структурами, включающими в себя внедренные кавычки.

```
// Using a verbatim string
let xmlFragment1 = @"<book author=""Milton, John"" title=""Paradise Lost"">"

// Using a triple-quoted string
let xmlFragment2 = """<book author="Milton, John" title="Paradise Lost">"""
```

В коде строки, имеющие разрывы строк, принимаются, а разрывы строк интерпретируется как символы новой строки, если только символ обратной косой черты не является последним символом перед разрывом строки. Начальные пробелы в следующей строке игнорируются при использовании символа обратной косой черты. Следующий `str1` код создает строку со значением `"abc\ndef"` и строкой `str2` , имеющей значение `"abcdef"` .

```
let str1 = "abc
def"
let str2 = "abc\
def"
```

Получить доступ к отдельным символам в строке можно с помощью синтаксиса, подобного массиву, как показано ниже.

```
printfn "%c" str1.[1]
```

В результате получается `b`.

Кроме того, можно извлечь подстроки с помощью синтаксиса среза массива, как показано в следующем коде.

```
printfn "%s" (str1.[0..2])
printfn "%s" (str2.[3..5])
```

Выходные данные выглядят следующим образом.

```
abc
def
```

Строки ASCII можно представлять массивами беззнаковых байтов типа `byte[]`. Добавьте суффикс `B` к строковому литералу, чтобы указать, что это строка ASCII. Строковые литералы ASCII, используемые с массивами байтов, поддерживают те же escape-последовательности, что и строки в Юникоде, за исключением escape-последовательностей Юникода.

```
// "abc" interpreted as a Unicode string.
let str1 : string = "abc"
// "abc" interpreted as an ASCII byte array.
let bytearray : byte[] = "abc"B
```

Строковые операторы

Существует два способа сцепления строк: с помощью `+` оператора или с `^` помощью оператора. `+` Оператор обеспечивает совместимость с функциями обработки строк .NET Framework.

В следующем примере показано объединение строк.

```
let string1 = "Hello, " + "world"
```

Класс String

Так как тип строки в F# фактически является `System.String` типом .NET Framework `System.String`, доступны все элементы. Сюда входит `+` оператор, который используется для сцепления строк `Length`, свойства и `Chars` свойства, которые возвращают строку в виде массива символов Юникода. Дополнительные сведения о строках см. `System.String` в разделе.

С помощью `Chars` свойства объекта `System.String` можно получить доступ к отдельным символам в строке, указав индекс, как показано в следующем коде.

```
let printChar (str : string) (index : int) =
    printfn "First character: %c" (str.Chars(index))
```

Строковый модуль

Дополнительные функции обработки строк включаются в `String` модуль `FSharp.Core` в пространстве

имен. Дополнительные сведения см. в разделе Модуль Core. String.

См. также

- Справочник по языку F#

Кортежи

23.10.2019 • 8 minutes to read • Edit Online

Кортеж — это группирование неименованных, но упорядоченных значений, возможно, для разных типов. Кортежи могут быть либо ссылочными типами, либо структурами.

Синтаксис

```
(element, ... , element)
struct(element, ... ,element )
```

Примечания

Каждый *элемент* в предыдущем синтаксисе может быть любым допустимым F# выражением.

Примеры

Примеры кортежей включают пары, триадные и т. д. одни и те же или разные типы. Некоторые примеры иллюстрируются в следующем коде.

```
(1, 2)

// Triple of strings.
("one", "two", "three")

// Tuple of generic types.
(a, b)

// Tuple that has mixed types.
("one", 1, 2.0)

// Tuple of integer expressions.
(a + 1, b + 1)

// Struct Tuple of floats
struct (1.025f, 1.5f)
```

Получение отдельных значений

Можно использовать сопоставление шаблонов для доступа и назначения имен для элементов кортежа, как показано в следующем коде.

```
let print tuple1 =
   match tuple1 with
    | (a, b) -> printfn "Pair %A %A" a b
```

Можно также деконструировать кортеж с помощью сопоставления шаблонов вне выражения с `match` помощью `let` привязки:

```
let (a, b) = (1, 2)

// Or as a struct
let struct (c, d) = struct (1, 2)
```

Можно также обозначить совпадение кортежей в качестве входных данных для функций:

```
let getDistance ((x1,y1): float*float) ((x2,y2): float*float) =
    // Note the ability to work on individual elements
    (x1*x2 - y1*y2)
    |> abs
    |> sqrt
```

Если требуется только один элемент кортежа, можно использовать подстановочный знак (символ подчеркивания), чтобы не создавать новое имя для ненужного значения.

```
let (a, _) = (1, 2)
```

Копирование элементов из ссылочного кортежа в кортеж структуры также является простым:

```
// Create a reference tuple
let (a, b) = (1, 2)

// Construct a struct tuple from it
let struct (c, d) = struct (a, b)
```

Функции `fst` и `snd` (только ссылочные кортежи) возвращают первый и второй элементы кортежа соответственно.

```
let c = fst (1, 2)
let d = snd (1, 2)
```

Встроенная функция, которая возвращает третий элемент Triple, не существует, но ее можно легко написать следующим образом.

```
let third (_, _, c) = c
```

Как правило, лучше использовать сопоставление шаблонов для доступа к отдельным элементам кортежа.

Использование кортежей

Кортежи предоставляют удобный способ возвращения нескольких значений из функции, как показано в следующем примере. Этот пример выполняет деление целых чисел и возвращает округленный результат операции как первый элемент пары кортежей и остаток как второй элемент пары.

```
let divRem a b =
   let x = a / b
   let y = a % b
   (x, y)
```

Кортежи также можно использовать в качестве аргументов функции, если нужно избежать неявного карринг аргументов функции, который подразумевается обычным синтаксисом функции.

```
let sumNoCurry (a, b) = a + b
```

Обычный синтаксис для определения функции `let sum a b = a + b` позволяет определить функцию, которая является частичным приложением первого аргумента функции, как показано в следующем коде.

```
let sum a b = a + b

let addTen = sum 10
let result = addTen 95
// Result is 105.
```

Использование кортежа в качестве параметра отключает карринг. Дополнительные сведения см. в разделе "частичное применение аргументов" в функциях.

Имена типов кортежей

При записи имени типа, который является кортежем, для разделения элементов используется `*` символ. Для кортежа, состоящего из `int` `float`„ `(10, 10.0, "ten")` и `string`, например, тип будет записан следующим образом.

```
int * float * string
```

Взаимодействие с C# кортежами

C#7,0 в язык добавлены кортежи. Кортежи в C# являются структурами и эквивалентны кортежам структуры в F#. Если требуется взаимодействие с C#, необходимо использовать кортежи структуры.

Это легко сделать. Например, представьте, что необходимо передать кортеж в C# класс, а затем использовать его результат, который также является кортежем:

```
namespace CSharpTupleInterop
{
    public static class Example
    {
        public static (int, int) AddOneToXAndY((int x, int y) a) =>
            (a.x + 1, a.y + 1);
    }
}
```

В F# коде можно передать кортеж структуры в качестве параметра и использовать результат в качестве кортежа структуры.

```
open TupleInterop

let struct (newX, newY) = Example.AddOneToXAndY(struct (1, 2))
// newX is now 2, and newY is now 3
```

Преобразование между кортежами ссылок и Кортежями структуры

Поскольку эталонные кортежи и кортежи структур имеют совершенно другое базовое представление, они не могут быть неявно преобразованы. Таким образом, код, подобный приведенному ниже, не будет компилироваться:

```
// Will not compile!
let (a, b) = struct (1, 2)

// Will not compile!
let struct (c, d) = (1, 2)

// Won't compile!
let f(t: struct(int*int)): int*int = t
```

Необходимо зашаблонировать совпадение в одном кортеже и создать другой с составляющими частями. Например:

```
// Pattern match on the result.
let (a, b) = (1, 2)

// Construct a new tuple from the parts you pattern matched on.
let struct (c, d) = struct (a, b)
```

Скомпилированная форма ссылочных кортежей

В этом разделе объясняется форма кортежей при их компиляции. Информация здесь не требуется для чтения, если не используется целевая версия .NET Framework 3,5 или ниже.

Кортежи компилируются в объекты одного из нескольких универсальных типов с именами `System.Tuple`, которые перегружены по арности или количество параметров типа. Типы кортежей отображаются в этой форме при их просмотре с другого языка, например C# или Visual Basic, или при использовании средства, не осведомленного о F# конструкциях. `Tuple` Типы были введены в .NET Framework 4. При использовании более ранней версии .NET Framework компилятор использует версии System. кортежа из версии 2,0 F# основной библиотеки. Типы в этой библиотеке используются только для приложений, предназначенных для версий 2,0, 3,0 и 3,5 .NET Framework. Пересылка типов используется для обеспечения совместимости двоичных компонентов между .NET Framework 2,0 и F# .NET Framework 4.

Скомпилированная форма кортежей структур

Кортежи структуры (например, `struct (x, y)`) являются фундаментально отличающимися от ссылочных кортежей. Они компилируются в ValueTuple тип, перегружается по арности или в число параметров типа. Они эквивалентны C# кортежам 7,0 и Visual Basicным кортежам 2017и взаимодействуют с двунаправленным письмом.

См. также

- Справочник по языку F#
- Типы языка F#

Типы коллекций F#

08.01.2020 • 28 minutes to read • Edit Online

Изучив этот раздел, можно определить, какой F# тип коллекции лучше подходит для конкретной потребности. Эти типы коллекций отличаются от типов коллекций в .NET Framework, например, в пространстве имен `System.Collections.Generic`, в том, что типы F# коллекций спроектированы с точки зрения функционального программирования, а не объектно-ориентированной перспективы. В частности, только коллекция массивов содержит изменяемые элементы. Таким образом, при изменении коллекции создается экземпляр измененной коллекции вместо изменения исходной коллекции.

Типы коллекций также отличаются в типе структуры данных, в которой хранятся объекты. Структуры данных, такие как хэш-таблицы, связанные списки и массивы, имеют различные характеристики производительности и разные наборы доступных операций.

Типы коллекций F#

В следующей таблице показаны F# типы коллекций.

ТИП	ОПИСАНИЕ	СВЯЗАННЫЕ ССЫЛКИ
List	Упорядоченная, неизменяемая серия элементов одного и того же типа. Реализуется как связанный список.	Списки Модуль List
массив.	Изменяемая коллекция последовательных элементов данных с фиксированным размером, начинающимся с нуля, которые имеют одинаковый тип.	Массивы Модуль Array Модуль Array2D Модуль Array3D
порядк	Логический ряд элементов одного типа. Последовательности особенно полезны при наличии большой упорядоченной коллекции данных, но не обязательно должны использовать все элементы. Отдельные элементы последовательности вычисляются только по мере необходимости, поэтому последовательность может выполняться лучше, чем список, если не все элементы используются. Последовательности представлены типом `seq<'T>`, который является псевдонимом для `IEnumerable<T>`. Таким образом, в качестве последовательности можно использовать любой тип .NET Framework, реализующий `System.Collections.Generic.IEnumerable<'T>`.	Последовательности Seq, модуль
Карта	Неизменяемый словарь элементов. Доступ к элементам осуществляется по ключу.	Модуль Map
Set	Неизменяемый набор, основанный на двоичных деревьях, где сравнение F# является структурной функцией сравнения, которая потенциально использует реализации интерфейса `System.IComparable` в значениях ключа.	Задать модуль

Таблица функций

В этом разделе сравниваются функции, доступные в F# типах коллекций. Определенная сложность функции определяется, где N — это размер первой коллекции, а M — размер второй коллекции, если таковой имеется. Тире (-) указывает, что эта функция недоступна в коллекции. Так как последовательности вычисляются отложенно, функция, например seq. DISTINCT, может быть O (1), поскольку она возвращает значение немедленно, хотя она по-прежнему влияет на производительность последовательности при перечислении.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
append	O (M)	O (N)	O (N)	-	-	Возвращает новую коллекцию, содержащую элементы первой коллекции, за которыми следуют элементы второй коллекции.
добавление	-	-	-	O (log N)	O (log N)	Возвращает новую коллекцию с добавленным элементом.
Среднее значение	O (N)	O (N)	O (N)	-	-	Возвращает среднее значение элементов в коллекции.
averageBy	O (N)	O (N)	O (N)	-	-	Возвращает среднее значение для результатов предоставленной функции, примененной к каждому элементу.
блит	O (N)	-	-	-	-	Копирует раздел массива.
кэш	-	-	O (N)	-	-	Выполняет вычисление и сохраняет элементы последовательности.
приведение	-	-	O (N)	-	-	Преобразует элементы в указанный тип.
choose	O (N)	O (N)	O (N)	-	-	Применяет заданную функцию `f` к каждому элементу `x` списка. Возвращает список, содержащий результаты для каждого элемента, в котором функция возвращает `Some(f(x))`.
сбор	O (N)	O (N)	O (N)	-	-	Применяет заданную функцию к каждому элементу коллекции, сцепляет все результаты и возвращает объединенный список.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
компаревис	-	-	O (N)	-	-	Сравнивает две последовательности, используя заданную функцию сравнения, элемент по элементу.
concat	O (N)	O (N)	O (N)	-	-	Объединяет заданное перечисление перечисления в виде одного объединенного перечисления.
содержит	-	-	-	-	O (log N)	Возвращает значение true, если набор содержит указанный элемент.
containsKey	-	-	-	O (log N)	-	Проверяет, находится ли элемент в домене схемы.
количество	-	-	-	-	O (N)	Возвращает количество элементов в наборе.
каунтби	-	-	O (N)	-	-	Применяет функцию создания ключа к каждому элементу последовательности и возвращает последовательность, которая получает уникальные ключи и их количество в исходной последовательности.
копирование	O (N)	-	O (N)	-	-	Копирует коллекцию.
создание	O (N)	-	-	-	-	Создает массив целых элементов, которые изначально имеют заданное значение.
отложить	-	-	O(1)	-	-	Возвращает последовательность, построенную из заданной отложенной спецификации последовательности.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
различие	-	-	-	-	O (M * log N)	Возвращает новый набор с элементами второго набора, удаленных из первого набора.
distinct			O (1)*			Возвращает последовательность, которая не содержит повторяющихся записей в соответствии с универсальным хэшем и сравнениями на равенство для записей. Если элемент встречается в последовательности несколько раз, последующие вхождения отбрасываются.
дистинктби			O (1)*			Возвращает последовательность, которая не содержит повторяющихся записей в соответствии с универсальным хэшем и сравнениями на равенство по ключам, возвращаемым данной функцией формирования ключа. Если элемент встречается в последовательности несколько раз, последующие вхождения отбрасываются.
пустых	O(1)	O(1)	O(1)	O(1)	O(1)	Создает пустую коллекцию.
exists	O (N)	O (N)	O (N)	O (log N)	O (log N)	Проверяет, удовлетворяет ли какой либо элемент последовательности заданному предикату.
exists2-	O (минимум (N, M))	-	O (минимум (N, M))			Проверяет, удовлетворяет ли ни одной паре соответствующих элементов входной последовательности заданному предикату.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
fill	O (N)					Задает диапазон элементов массива для заданного значения.
фильтр	O (N)	O (N)	O (N)	O (N)	O (N)	Возвращает новую коллекцию, содержащую только элементы коллекции, для которой заданный предикат возвращает `true`.
find	O (N)	O (N)	O (N)	O (log N)	-	Возвращает первый элемент, для которого заданная функция возвращает `true`. Возвращает `System.Collections.Gener`, если такого элемента не существует.
findIndex	O (N)	O (N)	O (N)	-	-	Возвращает индекс первого элемента в массиве, удовлетворяющего заданному предикату. Вызывает `System.Collections.Gener`, если ни один элемент не удовлетворяет предикату.
финдкэй	-	-	-	O (log N)	-	Вычисляет функцию для каждого сопоставления в коллекции и возвращает ключ для первого сопоставления, в котором функция возвращает `true`. Если такого элемента не существует, эта функция вызывает `System.Collections.Gener`.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
папка	O (N)	O (N)	O (N)	O (N)	O (N)	Применяет функцию к каждому элементу коллекции, перечисляя аргумент накапливаемое в потоке с помощью вычисления. Если входная функция — f, а элементы — i0... в эта функция выполняет вычисление f (... (f s I0)...) окне.
fold2	O (N)	O (N)	-	-	-	Применяет функцию к соответствующим элементам двух коллекций, перечисляя аргумент накапливаемое в потоке с помощью вычисления. Коллекции должны иметь одинаковые размеры. Если входная функция — f, а элементы — i0... в и j0... ЖН, эта функция вычислит f (... (f s, i0 j0)...) в ЖН.
foldBack	O (N)	O (N)	-	O (N)	O (N)	Применяет функцию к каждому элементу коллекции, перечисляя аргумент накапливаемое в потоке с помощью вычисления. Если входная функция — f, а элементы — i0... в эта функция выполняет вычисление f i0 (... (f в s)).

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
foldBack2	O (N)	O (N)	-	-	-	Применяет функцию к соответствующим элементам двух коллекций, перечисляя аргумент накапливаемое в потоке с помощью вычисления. Коллекции должны иметь одинаковые размеры. Если входная функция — f, а элементы — i0... в и j0... ЖН, эта функция вычислит f i0 j0 (... (f в ЖН s)).
ForAll	O (N)	O (N)	O (N)	O (N)	O (N)	Проверяет, соответствуют ли все элементы коллекции заданному предикату.
forall2	O (N)	O (N)	O (N)	-	-	Проверяет, соответствуют ли все соответствующие элементы коллекции заданному попарному предикату.
Get/n	O(1)	O (N)	O (N)	-	-	Возвращает элемент из коллекции по заданному индексу.
head	-	O(1)	O(1)	-	-	Возвращает первый элемент коллекции.
init	O (N)	O (N)	O(1)	-	-	Создает коллекцию, заданную измерением, и функцию генератора для вычислений элементов.
инитинфините	-	-	O(1)	-	-	Формирует последовательность, которая при переборе возвращает последовательные элементы путем вызова заданной функции.
секать	-	-	-	-	O (журнал N * м)	Вычисление пересечения двух наборов.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
интерсектмани	-	-	-	-	O (N1 * N2...)	Вычисление пересечения последовательности наборов. Последовательность не должна быть пустой.
isEmpty	O(1)	O(1)	O(1)	O(1)	-	Возвращает `true`, если коллекция пуста.
испроперсубсет	-	-	-	-	O (M * log N)	Возвращает `true`, если все элементы первого набора находятся во втором наборе, и хотя бы один элемент второго набора не находится в первом наборе.
испроперсуперсет	-	-	-	-	O (M * log N)	Возвращает `true`, если все элементы второго набора находятся в первом наборе, и хотя бы один элемент первого набора не входит во второй набор.
подмножество	-	-	-	-	O (M * log N)	Возвращает `true`, если все элементы первого набора находятся во втором наборе.
Супермножество	-	-	-	-	O (M * log N)	Возвращает `true`, если все элементы второго набора находятся в первом наборе.
iter	O (N)	O (N)	O (N)	O (N)	O (N)	Применяет заданную функцию к каждому элементу коллекции.
iteri	O (N)	O (N)	O (N)	-	-	Применяет заданную функцию к каждому элементу коллекции. Целое число, передаваемое в функцию, указывает индекс элемента.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
iteri2	O (N)	O (N)	-	-	-	Применяет заданную функцию к паре элементов, созданных из совпадающих индексов в двух массивах. Целое число, передаваемое в функцию, указывает индекс элементов. Длина двух массивов должна быть одинаковой.
iter2	O (N)	O (N)	O (N)	-	-	Применяет заданную функцию к паре элементов, созданных из совпадающих индексов в двух массивах. Длина двух массивов должна быть одинаковой.
последний	O(1)	O (N)	O (N)	-	-	Возвращает последний элемент в применимой коллекции.
длина	O(1)	O (N)	O (N)	-	-	Возвращает количество элементов в коллекции.
map	O (N)	O (N)	O(1)	-	-	Создает коллекцию, элементы которой являются результатами применения заданной функции к каждому элементу массива.
map2	O (N)	O (N)	O(1)	-	-	Создает коллекцию, элементы которой являются результатами применения заданной функции к соответствующим элементам двух коллекций попарно. Длина двух входных массивов должна быть одинаковой.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
map3	-	O (N)	-	-	-	Создает коллекцию, элементы которой являются результатами применения заданной функции к соответствующим элементам трех коллекций одновременно.
действует	O (N)	O (N)	O (N)	-	-	Создает массив, элементы которого являются результатами применения заданной функции к каждому элементу массива. Целочисленный индекс, передаваемый функции, указывает индекс преобразуемого элемента.
mapi2	O (N)	O (N)	-	-	-	Выполняет сборку коллекции, элементы которой являются результатами применения заданной функции к соответствующим элементам двух коллекций попарно, также передавая индекс элементов. Длина двух входных массивов должна быть одинаковой.
max	O (N)	O (N)	O (N)	-	-	Возвращает наибольший элемент в коллекции по сравнению с использованием оператора Max .
максби	O (N)	O (N)	O (N)	-	-	Возвращает наибольший элемент в коллекции, по сравнению с использованием параметра Max в результатах функции.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
макселемент	-	-	-	-	O (log N)	Возвращает наибольший элемент в наборе согласно упорядочению, используемому для набора.
min	O (N)	O (N)	O (N)	-	-	Возвращает элемент наименьшего уровня в коллекции по сравнению с использованием оператора min .
минби	O (N)	O (N)	O (N)	-	-	Возвращает элемент наименьшего уровня в коллекции по сравнению с использованием оператора min в результатах функции.
минелемент	-	-	-	-	O (log N)	Возвращает нижний элемент в наборе согласно упорядочению, используемому для набора.
офаррай	-	O (N)	O(1)	O (N)	O (N)	Создает коллекцию, содержащую те же элементы, что и заданный массив.
офлист	O (N)	-	O(1)	O (N)	O (N)	Создает коллекцию, содержащую те же элементы, что и заданный список.
ofSeq	O (N)	O (N)	-	O (N)	O (N)	Создает коллекцию, содержащую те же элементы, что и заданная последовательность.
кэширован	-	-	O (N)	-	-	Возвращает последовательность каждого элемента во входной последовательности и его предшественника , за исключением первого элемента, который возвращается только в качестве предшественника второго элемента.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
partition	O (N)	O (N)	-	O (N)	O (N)	Разделяет коллекцию на две коллекции. Первая коллекция содержит элементы, для которых заданный предикат возвращает `true`, а вторая коллекция содержит элементы, для которых заданный предикат возвращает `false`.
permute	O (N)	O (N)	-	-	-	Возвращает массив со всеми элементами переставляются в соответствии с заданной перестановкой.
брать	O (N)	O (N)	O (N)	O (log N)	-	Применяет заданную функцию к последовательным элементам, возвращая первый результат, в котором функция возвращает часть. Если функция никогда не возвращает некоторые, возникает `System.Collections.Gener` .

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
readonly	-	-	O (N)	-	-	Создает объект последовательности, который делегирует заданному объекту последовательности. Эта операция гарантирует, что приведение типа не сможет повторно обнаружить и изменить исходную последовательность. Например, если задан массив, возвращаемая последовательность будет возвращать элементы массива, но возвращаемый объект последовательности нельзя привести к массиву.
reduce	O (N)	O (N)	O (N)	-	-	Применяет функцию к каждому элементу коллекции, перечисляя аргумент накапливаемое в потоке с помощью вычисления. Эта функция начинается с применения функции к первым двум элементам, передает этот результат в функцию вместе с третьим элементом и т. д. Функция возвращает окончательный результат.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
редуцебакк	O (N)	O (N)	-	-	-	Применяет функцию к каждому элементу коллекции, перечисляя аргумент накапливаемое в потоке с помощью вычисления. Если входная функция — f, а элементы — i0... в эта функция выполняет вычисление f i0 (... (f в-1 в)).
remove	-	-	-	O (log N)	O (log N)	Удаляет элемент из домена на карте. Если элемент отсутствует, исключение не возникает.
водил	-	O (N)	-	-	-	Создает список указанной длины с каждым набором элементов на заданное значение.
расширения	O (N)	O (N)	-	-	-	Возвращает новый список с элементами в обратном порядке.
наличия	O (N)	O (N)	O (N)	-	-	Применяет функцию к каждому элементу коллекции, перечисляя аргумент накапливаемое в потоке с помощью вычисления. Эта операция применяет функцию ко второму аргументу и первому элементу списка. Затем операция передает этот результат в функцию вместе со вторым элементом и т. д. Наконец, операция возвращает список промежуточных результатов и окончательный результат.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
scanBack	O (N)	O (N)	-	-	-	Напоминает операцию foldBack, но возвращает и промежуточные, и окончательные результаты.
singleton	-	-	O(1)	-	O(1)	Возвращает последовательность, которая получает только один элемент.
set	O(1)	-	-	-	-	Задает для элемента массива указанное значение.
skip	-	-	O (N)	-	-	Возвращает последовательность, которая пропускает N элементов базовой последовательности, а затем возвращает оставшиеся элементы последовательности.
skipWhile	-	-	O (N)	-	-	Возвращает последовательность, которая при переборе пропускает элементы базовой последовательности, пока заданный предикат возвращает `true`, а затем выдает оставшиеся элементы последовательности.
sort	O (N log N) среднее O (N ^ 2) наихудший случай	O (N log N)	O (N log N)	-	-	Сортирует коллекцию по значению элемента. Элементы сравниваются с помощью функции Compare.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
sortBy	O (N log N) среднее O (N ^ 2) наихудший случай	O (N log N)	O (N log N)	-	-	Сортирует заданный список с помощью ключей, предоставляемых заданной проекцией. Ключи сравниваются с помощью функции Compare.
сортинплаце	O (N log N) среднее O (N ^ 2) наихудший случай	-	-	-	-	Сортирует элементы массива, изменяя его на месте и используя заданную функцию сравнения. Элементы сравниваются с помощью функции Compare.
сортинплацеби	O (N log N) среднее O (N ^ 2) наихудший случай	-	-	-	-	Сортирует элементы массива, изменяя его на месте и используя заданную проекцию для ключей. Элементы сравниваются с помощью функции Compare.
сортинплацевис	O (N log N) среднее O (N ^ 2) наихудший случай	-	-	-	-	Сортирует элементы массива, изменяя его на месте и используя данную функцию сравнения в качестве порядка.
сортвис	O (N log N) среднее O (N ^ 2) наихудший случай	O (N log N)	-	-	-	Сортирует элементы коллекции, используя заданную функцию сравнения в качестве порядка и возвращая новую коллекцию.
sub	O (N)	-	-	-	-	Создает массив, содержащий заданный поддиапазон, заданный начальным индексом и длиной.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
сумма	O (N)	O (N)	O (N)	-	-	Возвращает сумму элементов в коллекции.
sumBy	O (N)	O (N)	O (N)	-	-	Возвращает сумму результатов, созданных путем применения функции к каждому элементу коллекции.
односторонне	-	O(1)	-	-	-	Возвращает список без первого элемента.
take	-	-	O (N)	-	-	Возвращает элементы последовательности вплоть до указанного числа.
takeWhile	-	-	O(1)	-	-	Возвращает последовательность, которая при переборе возвращает элементы базовой последовательности, пока заданный предикат возвращается `true` а затем возвращает больше элементов.
toArray	-	O (N)	O (N)	O (N)	O (N)	Создает массив из заданной коллекции.
toList	O (N)	-	O (N)	O (N)	O (N)	Создает список из заданной коллекции.
toSeq	O(1)	O(1)	-	O(1)	O(1)	Создает последовательность из заданной коллекции.
truncate	-	-	O(1)	-	-	Возвращает последовательность, которая при перечислении возвращает не более N элементов.
tryFind	O (N)	O (N)	O (N)	O (log N)	-	Выполняет поиск элемента, удовлетворяющего заданному предикату.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
tryFindIndex	O (N)	O (N)	O (N)	-	-	Выполняет поиск первого элемента, удовлетворяющего заданному предикату, и возвращает индекс соответствующего элемента или `None`, если такого элемента не существует.
трифиндкэй	-	-	-	O (log N)	-	Возвращает ключ первого сопоставления в коллекции, удовлетворяющего заданному предикату, или возвращает `None`, если такого элемента не существует.
tryPick	O (N)	O (N)	O (N)	O (log N)	-	Применяет заданную функцию к последовательным элементам, возвращая первый результат, в котором функция возвращает `Some` для некоторого значения. Если такого элемента не существует, операция возвращает `None`.
Unfold	-	-	O (N)	-	-	Возвращает последовательность, содержащую элементы, создаваемые данным вычислением.
объединение	-	-	-	-	O (M * log N)	Вычисление объединения двух наборов.
унионмани	-	-	-	-	O (N1 * N2...)	Выполняет вычисление объединения последовательности наборов.
unzip	O (N)	O (N)	O (N)	-	-	Разделяет список пар на два списка.
unzip3	O (N)	O (N)	O (N)	-	-	Разделяет список триад на три списка.

ФУНКЦИЯ	МАССИВ	СПИСОК	SEQUENCE	КАРТА	ЗАДАТЬ	ОПИСАНИЕ
оконные	-	-	O (N)	-	-	Возвращает последовательность, которая дает скользящие окна элементов, которые были выведены из входной последовательности. Каждое окно возвращается в виде нового массива.
zip	O (N)	O (N)	O (N)	-	-	Объединяет две коллекции в список пар. Два списка должны иметь одинаковую длину.
zip3	O (N)	O (N)	O (N)	-	-	Объединяет три коллекции в список Тройн. Списки должны иметь одинаковую длину.

См. также:

- Типы языка F#
- Справочник по языку F#

Списки

23.10.2019 • 34 minutes to read • Edit Online

> **NOTE**
> Ссылки на справочник по API в этой статье ведут на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

В языке F# список — это упорядоченная, неизменная серия элементов одного типа. Для выполнения основных операций со списками используйте функции в модуле List.

Создание и инициализация списков

Список можно определить путем прямого перечисления элементов, разделенных точкой с запятой и заключенных в квадратные скобки, как показано в следующей строке кода.

```
let list123 = [ 1; 2; 3 ]
```

Вместо точки с запятой для разделения элементов можно также использовать разрыв строки. Такой синтаксис позволяет получить более удобный для чтения код, если список содержит длинные выражения инициализации или к каждому элементу необходимо написать комментарий.

```
let list123 = [
    1
    2
    3 ]
```

Обычно все элементы в списке должны быть одного типа. Исключением является список, в котором элементы основного типа могут иметь элементы производных типов. Следующий вариант считается приемлемым, так как типы `Button` и `CheckBox` являются производными от типа `Control`.

```
let myControlList : Control list = [ new Button(); new CheckBox() ]
```

Определить элементы списка можно также с помощью диапазона, который будет ограничен целыми числами, разделенными оператором (`..`), как показано в следующем коде.

```
let list1 = [ 1 .. 10 ]
```

Пустой список определяется парой квадратных скобок, между которыми ничего не указано.

```
// An empty list.
let listEmpty = []
```

Также список можно создать с помощью выражения последовательности. Дополнительные сведения см. в разделе выражения последовательностей . Например, в следующем коде создается список квадратов целочисленных значений от 1 до 10.

```
let listOfSquares = [ for i in 1 .. 10 -> i*i ]
```

Операторы для работы со списками

Оператор `::` позволяет добавлять элементы в список. Если список `list1` включает `[2; 3; 4]`, то следующий код создает список `list2` как `[100; 2; 3; 4]`.

```
let list2 = 100 :: list1
```

Оператор `@` позволяет объединять списки совместимых типов, как показано в следующем коде. Если список `list1` включает `[2; 3; 4]`, список `list2` — `[100; 2; 3; 4]`, то следующий код создает список `list3` как `[2; 3; 4; 100; 2; 3; 4]`.

```
let list3 = list1 @ list2
```

Функции для выполнения операций с списками доступны в модуле List.

Поскольку списки в языке F# являются неизменными, то операции изменения не изменяют существующие списки, а создают новые.

Списки в F# реализуются как однонаправленные списки. Это означает, что операции, обращающиеся только к заголовку списка, являются o (1), а доступ к элементу — o (*n*).

Свойства

Тип списка поддерживает следующие свойства.

СВОЙСТВО.	TYPE	ОПИСАНИЕ
Глава	`'T`	Первый элемент
Указано	`'T list`	Статическое свойство, которое возвращает пустой список соответствующего типа.
IsEmpty	`bool`	`true` значение, если список не содержит элементов.
Элемент	`'T`	Элемент с указанным индексом (начинается с нуля).
Длина	`int`	Количество элементов
Односторонне	`'T list`	Список без первого элемента

Ниже приведены некоторые примеры использования данных свойств.

```
let list1 = [ 1; 2; 3 ]

// Properties
printfn "list1.IsEmpty is %b" (list1.IsEmpty)
printfn "list1.Length is %d" (list1.Length)
printfn "list1.Head is %d" (list1.Head)
printfn "list1.Tail.Head is %d" (list1.Tail.Head)
printfn "list1.Tail.Tail.Head is %d" (list1.Tail.Tail.Head)
printfn "list1.Item(1) is %d" (list1.Item(1))
```

Использование списков

Программирование с использованием списков позволяет выполнять сложные операции с небольшим количеством кода. В данном разделе описываются операции со списками, важные для функционального программирования.

Рекурсия со списками

Списки однозначно подходят для техник рекурсивного программирования. Рассмотрим операцию, в которой должен участвовать каждый элемент списка. Это можно сделать рекурсивно, т. е. сначала обработать начало списка, затем перейти к хвосту — более короткому списку, состоящему из первоначального списка без первого элемента, а потом снова перейти на следующий уровень рекурсии.

Для написания такой рекурсивной функции используется оператор (`::`) в сопоставлении шаблонов, который позволяет отделить начало списка от хвоста.

Следующий пример кода показывает, как использовать сопоставление шаблонов для реализации рекурсивной функции, выполняющей операции над списком.

```
let rec sum list =
   match list with
   | head :: tail -> head + sum tail
   | [] -> 0
```

Предыдущий код хорошо работает для небольших списков, но при работе со списками большого размера может случиться переполнение стека. Следующий код улучшает предыдущий за счет использования аргумента аккумулирования — это стандартная техника работы с рекурсивными функциями. Использование аргумента аккумулирования делает функцию рекурсивной по отношению к хвосту, что экономит место в стеке.

```
let sum list =
   let rec loop list acc =
       match list with
       | head :: tail -> loop tail (acc + head)
       | [] -> acc
   loop list 0
```

Функция `RemoveAllMultiples` — это рекурсивная функция, которая обрабатывает два списка. Первый список содержит цифры, кратные которым будут удалены, а второй представляет собой список, из которого будут удаляться цифры. Код в следующем примере использует рекурсивную функцию для удаления всех непростых чисел из списка. После его выполнения в списке остаются только простые числа.

```
let IsPrimeMultipleTest n x =
   x = n || x % n <> 0

let rec RemoveAllMultiples listn listx =
   match listn with
   | head :: tail -> RemoveAllMultiples tail (List.filter (IsPrimeMultipleTest head) listx)
   | [] -> listx


let GetPrimesUpTo n =
    let max = int (sqrt (float n))
    RemoveAllMultiples [ 2 .. max ] [ 1 .. n ]

printfn "Primes Up To %d:\n %A" 100 (GetPrimesUpTo 100)
```

Выходные данные выглядят следующим образом:

```
Primes Up To 100:
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71; 73; 79; 83; 89; 97]
```

Функции модуля

Модуль List предоставляет функции, которые обращаются к элементам списка. Самым легким и быстрым для доступа является первоначальный элемент. Используйте заголовок свойства или список функций модуля . Head. Можно получить доступ к заключительному фрагменту списка с помощью свойства tail или функции List. tail . Чтобы найти элемент по индексу, используйте функцию List. nth . `List.nth` проходит по списку. Таким образом, это O (*n*). Если в коде часто используется `List.nth`, то вместо списка можно использовать массив. Доступ к элементам массива осуществляется через O(1).

Логические операции со списками

Функция List. isEmpty определяет, содержит ли список какие-либо элементы.

Функция List. Exists применяет логический тест к элементам списка и возвращает `true` значение, если какой-либо элемент удовлетворяет данному тесту. List. exists2- аналогичен, но работает с последовательными парами элементов в двух списках.

В следующем коде показано использование функции `List.exists`.

```
// Use List.exists to determine whether there is an element of a list satisfies a given Boolean
expression.
// containsNumber returns true if any of the elements of the supplied list match
// the supplied number.
let containsNumber number list = List.exists (fun elem -> elem = number) list
let list0to3 = [0 .. 3]
printfn "For list %A, contains zero is %b" list0to3 (containsNumber 0 list0to3)
```

Выходные данные выглядят следующим образом:

```
For list [0; 1; 2; 3], contains zero is true
```

В следующем примере показано использование функции `List.exists2`.

```
// Use List.exists2 to compare elements in two lists.
// isEqualElement returns true if any elements at the same position in two supplied
// lists match.
let isEqualElement list1 list2 = List.exists2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
let list1to5 = [ 1 .. 5 ]
let list5to1 = [ 5 .. -1 .. 1 ]
if (isEqualElement list1to5 list5to1) then
    printfn "Lists %A and %A have at least one equal element at the same position." list1to5 list5to1
else
    printfn "Lists %A and %A do not have an equal element at the same position." list1to5 list5to1
```

Выходные данные выглядят следующим образом:

```
Lists [1; 2; 3; 4; 5] and [5; 4; 3; 2; 1] have at least one equal element at the same position.
```

List. forall можно использовать, если требуется проверить, соответствуют ли все элементы списка условию.

```
let isAllZeroes list = List.forall (fun elem -> elem = 0.0) list
printfn "%b" (isAllZeroes [0.0; 0.0])
printfn "%b" (isAllZeroes [0.0; 1.0])
```

Выходные данные выглядят следующим образом:

```
true
false
```

Аналогичным образом List. forall2 определяет, соответствуют ли все элементы в соответствующих позициях в двух списках логическому выражению, включающему в себя каждую пару элементов.

```
let listEqual list1 list2 = List.forall2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
printfn "%b" (listEqual [0; 1; 2] [0; 1; 2])
printfn "%b" (listEqual [0; 0; 0] [0; 1; 0])
```

Выходные данные выглядят следующим образом:

```
true
false
```

Операции сортировки списков

Функции List. Sort, List. sortByи List. сортвис сортируют списки сортировки. Функция сортировки определяет, какую из этих трех функций использовать. `List.sort` использует универсальное сравнение по умолчанию. Общее сравнение выполняется с помощью глобальных операторов на основе функции общего сравнения значений. Оно эффективно работает с различными типами элементов, такими как числовые типы, кортежи, записи, размеченные объединения, списки, массивы и любой другой тип, включающий `System.IComparable`. Для типов, включающих `System.IComparable`, общее сравнение выполняется с помощью функции `System.IComparable.CompareTo()`. Общее сравнение также работает со строками, но использует культурно-независимый порядок сортировки. Общее сравнение не следует применять к неподдерживаемым типам, например типам функций. К тому же выполнение общего сравнения по умолчанию лучше всего подходит для слабо структурированных типов. Для сильно структурированных типов, которые необходимо часто сравнивать и сортировать, можно использовать функцию `System.IComparable` и метод `System.IComparable.CompareTo()`.

`List.sortBy` принимает функцию, которая возвращает значение, используемое в качестве критерия сортировки, и `List.sortWith` принимает в качестве аргумента функцию сравнения. Две последние функции полезны при работе с типами, которые не поддерживают сравнение, а также если сравнение требует более сложной семантики, например в случае со строками, учитывающими язык и регион.

В следующем примере показано использование функции `List.sort`.

```
let sortedList1 = List.sort [1; 4; 8; -2; 5]
printfn "%A" sortedList1
```

Выходные данные выглядят следующим образом:

```
[-2; 1; 4; 5; 8]
```

В следующем примере показано использование функции `List.sortBy`.

```
let sortedList2 = List.sortBy (fun elem -> abs elem) [1; 4; 8; -2; 5]
printfn "%A" sortedList2
```

Выходные данные выглядят следующим образом:

```
[1; -2; 4; 5; 8]
```

В следующем примере показано использование `List.sortWith`. В этом примере обычная функция сравнения `compareWidgets` используется сначала для сравнения одного поля пользовательского типа, а затем другого, если значения первого поля равны.

```
type Widget = { ID: int; Rev: int }

let compareWidgets widget1 widget2 =
   if widget1.ID < widget2.ID then -1 else
   if widget1.ID > widget2.ID then 1 else
   if widget1.Rev < widget2.Rev then -1 else
   if widget1.Rev > widget2.Rev then 1 else
   0

let listToCompare = [
    { ID = 92; Rev = 1 }
    { ID = 110; Rev = 1 }
    { ID = 100; Rev = 5 }
    { ID = 100; Rev = 2 }
    { ID = 92; Rev = 1 }
    ]

let sortedWidgetList = List.sortWith compareWidgets listToCompare
printfn "%A" sortedWidgetList
```

Выходные данные выглядят следующим образом:

```
[{ID = 92;
Rev = 1;}; {ID = 92;
Rev = 1;}; {ID = 100;
Rev = 2;}; {ID = 100;
Rev = 5;}; {ID = 110;
Rev = 1;}]
```

Операции поиска в списках

Списки поддерживают различные операции поиска. Простейшая функция List. Findпозволяет найти первый элемент, соответствующий заданному условию.

В следующем примере кода показано, как использовать `List.find` для поиска первого числа в списке, которое делится на 5.

```
let isDivisibleBy number elem = elem % number = 0
let result = List.find (isDivisibleBy 5) [ 1 .. 100 ]
printfn "%d " result
```

Результат — 5.

Если элементы должны быть преобразованы первыми, вызовите List. Pick, который принимает функцию, возвращающую параметр, и ищет первое значение параметра, равное `Some(x)`. Вместо возвращения элемента функция `List.pick` возвращает результат `x`. Если совпадения не найдены, функция `List.pick` возвращает `System.Collections.Generic.KeyNotFoundException`. В следующем коде показано использование `List.pick`.

```
let valuesList = [ ("a", 1); ("b", 2); ("c", 3) ]

let resultPick = List.pick (fun elem ->
                    match elem with
                    | (value, 2) -> Some value
                    | _ -> None) valuesList
printfn "%A" resultPick
```

Выходные данные выглядят следующим образом:

```
"b"
```

Другая группа операций поиска, List. tryFind и связанных функций, возвращает значение параметра. Функция `List.tryFind` возвращает первый элемент списка, который удовлетворяет условию, если такой элемент есть, и значение параметра `None`, если нет. Список вариантов . tryFindIndex возвращает индекс элемента, если он найден, а не сам элемент. Эти функции представлены в следующем коде.

```
let list1d = [1; 3; 7; 9; 11; 13; 15; 19; 22; 29; 36]
let isEven x = x % 2 = 0
match List.tryFind isEven list1d with
| Some value -> printfn "The first even value is %d." value
| None -> printfn "There is no even value in the list."

match List.tryFindIndex isEven list1d with
| Some value -> printfn "The first even value is at position %d." value
| None -> printfn "There is no even value in the list."
```

Выходные данные выглядят следующим образом:

```
The first even value is 22.
The first even value is at position 8.
```

Арифметические операции со списками

Общие арифметические операции, такие как Sum и Average, встроены в модуль List. Для работы с List. Sumтип элемента списка должен поддерживать `+` оператор и иметь нулевое значение. Все встроенные

арифметические типы удовлетворяют этим условиям. Для работы с List. Averageтип элемента должен поддерживать деление без остатка, который исключает целочисленные типы, но допускает типы с плавающей запятой. Функции List. sumBy и List. averageBy принимают в качестве параметра функцию, а результаты этой функции используются для вычисления значений суммы или среднего значения.

В следующем коде показано использование `List.sum`, `List.sumBy` и `List.average`.

```
// Compute the sum of the first 10 integers by using List.sum.
let sum1 = List.sum [1 .. 10]

// Compute the sum of the squares of the elements of a list by using List.sumBy.
let sum2 = List.sumBy (fun elem -> elem*elem) [1 .. 10]

// Compute the average of the elements of a list by using List.average.
let avg1 = List.average [0.0; 1.0; 1.0; 2.0]

printfn "%f" avg1
```

В результате получается `1.000000`.

В следующем коде показано использование `List.averageBy`.

```
let avg2 = List.averageBy (fun elem -> float elem) [1 .. 10]
printfn "%f" avg2
```

В результате получается `5.5`.

Списки и кортежи

Для работы со списками, содержащими кортежи, можно использовать функции упаковки и распаковки. Они объединяют два списка с одним значением в один список кортежей или разбивают один список кортежей на два списка с одним значением. Простейшая функция List. zip принимает два списка отдельных элементов и создает один список пар кортежей. Другая версия, List. zip3, принимает три списка отдельных элементов и создает один список кортежей с тремя элементами. В следующем коде показано использование функции `List.zip`.

```
let list1 = [ 1; 2; 3 ]
let list2 = [ -1; -2; -3 ]
let listZip = List.zip list1 list2
printfn "%A" listZip
```

Выходные данные выглядят следующим образом:

```
[(1, -1); (2, -2); (3; -3)]
```

В следующем коде показано использование функции `List.zip3`.

```
let list3 = [ 0; 0; 0]
let listZip3 = List.zip3 list1 list2 list3
printfn "%A" listZip3
```

Выходные данные выглядят следующим образом:

```
[(1, -1, 0); (2, -2, 0); (3, -3, 0)]
```

Соответствующие версии распаковать, List. unzip и List. unzip3, принимают списки кортежей и возвращаемых списков в кортеже, где первый список содержит все элементы, которые были первыми в каждом кортеже, а второй список содержит второй элемент каждого кортеж и т. д.

В следующем примере кода показано использование List. unzip.

```
let lists = List.unzip [(1,2); (3,4)]
printfn "%A" lists
printfn "%A %A" (fst lists) (snd lists)
```

Выходные данные выглядят следующим образом:

```
([1; 3], [2; 4])
[1; 3] [2; 4]
```

В следующем примере кода показано использование List. unzip3.

```
let listsUnzip3 = List.unzip3 [(1,2,3); (4,5,6)]
printfn "%A" listsUnzip3
```

Выходные данные выглядят следующим образом:

```
([1; 4], [2; 5], [3; 6])
```

Операции с элементами списка

F# поддерживает различные операции с элементами списка. Самый простой — List. iter, который позволяет вызывать функцию для каждого элемента списка. Варианты включают List. iter2, который позволяет выполнять операции над элементами двух списков, List. iteri, которые подобны `List.iter` , за исключением того, что индекс каждого элемента передается в качестве аргумента функции, вызываемой для каждого Element и List. iteri2, являющийся сочетанием функций `List.iter2` и. `List.iteri` Эти функции показаны в следующем примере кода.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
List.iter (fun x -> printfn "List.iter: element is %d" x) list1
List.iteri(fun i x -> printfn "List.iteri: element %d is %d" i x) list1
List.iter2 (fun x y -> printfn "List.iter2: elements are %d %d" x y) list1 list2
List.iteri2 (fun i x y ->
                printfn "List.iteri2: element %d of list1 is %d element %d of list2 is %d"
                  i x i y)
            list1 list2
```

Выходные данные выглядят следующим образом:

```
List.iter: element is 1
List.iter: element is 2
List.iter: element is 3
List.iteri: element 0 is 1
List.iteri: element 1 is 2
List.iteri: element 2 is 3
List.iter2: elements are 1 4
List.iter2: elements are 2 5
List.iter2: elements are 3 6
List.iteri2: element 0 of list1 is 1; element 0 of list2 is 4
List.iteri2: element 1 of list1 is 2; element 1 of list2 is 5
List.iteri2: element 2 of list1 is 3; element 2 of list2 is 6
```

Другой часто используемой функцией, которая преобразует элементы списка, является List. Map, которая позволяет применить функцию к каждому элементу списка и поместит все результаты в новый список. List. map2 и List. map3 — это варианты, принимающие несколько списков. Можно также использовать List. MAPI и List. mapi2, если в дополнение к элементу, функции необходимо передать индекс каждого элемента. Единственное различие между `List.mapi2` и `List.mapi` состоит в том, что функция `List.mapi2` работает с двумя списками. В следующем примере показан List. Map.

```
let list1 = [1; 2; 3]
let newList = List.map (fun x -> x + 1) list1
printfn "%A" newList
```

Выходные данные выглядят следующим образом:

```
[2; 3; 4]
```

В следующем коде показано использование функции `List.map2`.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
let sumList = List.map2 (fun x y -> x + y) list1 list2
printfn "%A" sumList
```

Выходные данные выглядят следующим образом:

```
[5; 7; 9]
```

В следующем коде показано использование функции `List.map3`.

```
let newList2 = List.map3 (fun x y z -> x + y + z) list1 list2 [2; 3; 4]
printfn "%A" newList2
```

Выходные данные выглядят следующим образом:

```
[7; 10; 13]
```

В следующем коде показано использование функции `List.mapi`.

```
let newListAddIndex = List.mapi (fun i x -> x + i) list1
printfn "%A" newListAddIndex
```

Выходные данные выглядят следующим образом:

```
[1; 3; 5]
```

В следующем коде показано использование функции `List.mapi2`.

```
let listAddTimesIndex = List.mapi2 (fun i x y -> (x + y) * i) list1 list2
printfn "%A" listAddTimesIndex
```

Выходные данные выглядят следующим образом:

```
[0; 7; 18]
```

List. собирающий аналогичен `List.map`, за исключением того, что каждый элемент создает список, а все эти списки объединяются в окончательный список. В следующем коде каждый элемент списка генерирует три числа. Все они собираются в один список.

```
let collectList = List.collect (fun x -> [for i in 1..3 -> x * i]) list1
printfn "%A" collectList
```

Выходные данные выглядят следующим образом:

```
[1; 2; 3; 2; 4; 6; 3; 6; 9]
```

Можно также использовать List. Filter, который принимает логическое условие и создает новый список, состоящий только из элементов, которые соответствуют заданному условию.

```
let evenOnlyList = List.filter (fun x -> x % 2 = 0) [1; 2; 3; 4; 5; 6]
```

Результатом является список `[2; 4; 6]`.

Сочетание Map и Filter, List. Choose позволяет одновременно преобразовывать и выбирать элементы. `List.choose` применяет функцию, возвращающую параметр для каждого элемента списка, и возвращает новый список результатов для элементов, когда функция возвращает значение `Some` параметра.

В следующем коде показано использование функции `List.choose` для выбора из списка слов с заглавными буквами.

```
let listWords = [ "and"; "Rome"; "Bob"; "apple"; "zebra" ]
let isCapitalized (string1:string) = System.Char.IsUpper string1.[0]
let results = List.choose (fun elem ->
    match elem with
    | elem when isCapitalized elem -> Some(elem + "'s")
    | _ -> None) listWords
printfn "%A" results
```

Выходные данные выглядят следующим образом:

```
["Rome's"; "Bob's"]
```

Операции с несколькими списками

Списки могут быть объединены. Чтобы объединить два списка в один, используйте List. append. Для объединения более двух списков используйте List. Concat.

```
let list1to10 = List.append [1; 2; 3] [4; 5; 6; 7; 8; 9; 10]
let listResult = List.concat [ [1; 2; 3]; [4; 5; 6]; [7; 8; 9] ]
List.iter (fun elem -> printf "%d " elem) list1to10
printfn ""
List.iter (fun elem -> printf "%d " elem) listResult
```

Операции сворачивания и сканирования

Некоторые операции со списками включают взаимозависимости между всеми элементами списка. Операции свертывания и сканирования подобны `List.iter` и `List.map` в том, что вы вызываете функцию для каждого элемента, но эти операции предоставляют дополнительный параметр, называемый *накопительным* , который передает информацию через вычисления.

`List.fold` можно использовать для выполнения расчетов со списком.

В следующем примере кода показано использование List. fold для выполнения различных операций.

Лист обходится. Аккумулятор `acc` — это значение, которое передается дальше, пока продолжается расчет. Первый аргумент забирает аккумулятор и элемент списка и возвращает промежуточный результат расчета для этого элемента списка. Второй аргумент является исходным значением аккумулятора.

```
let sumList list = List.fold (fun acc elem -> acc + elem) 0 list
printfn "Sum of the elements of list %A is %d." [ 1 .. 3 ] (sumList [ 1 .. 3 ])

// The following example computes the average of a list.
let averageList list = (List.fold (fun acc elem -> acc + float elem) 0.0 list / float list.Length)

// The following example computes the standard deviation of a list.
// The standard deviation is computed by taking the square root of the
// sum of the variances, which are the differences between each value
// and the average.
let stdDevList list =
    let avg = averageList list
    sqrt (List.fold (fun acc elem -> acc + (float elem - avg) ** 2.0 ) 0.0 list / float list.Length)

let testList listTest =
    printfn "List %A average: %f stddev: %f" listTest (averageList listTest) (stdDevList listTest)

testList [1; 1; 1]
testList [1; 2; 1]
testList [1; 2; 3]

// List.fold is the same as to List.iter when the accumulator is not used.
let printList list = List.fold (fun acc elem -> printfn "%A" elem) () list
printList [0.0; 1.0; 2.5; 5.1 ]

// The following example uses List.fold to reverse a list.
// The accumulator starts out as the empty list, and the function uses the cons operator
// to add each successive element to the head of the accumulator list, resulting in a
// reversed form of the list.
let reverseList list = List.fold (fun acc elem -> elem::acc) [] list
printfn "%A" (reverseList [1 .. 10])
```

Версии этих функций с цифрой в имени функции работают с несколькими списками. Например, List. fold2 выполняет вычисления в двух списках.

В следующем примере показано использование функции `List.fold2`.

```
// Use List.fold2 to perform computations over two lists (of equal size) at the same time.
// Example: Sum the greater element at each list position.
let sumGreatest list1 list2 = List.fold2 (fun acc elem1 elem2 ->
                                            acc + max elem1 elem2) 0 list1 list2

let sum = sumGreatest [1; 2; 3] [3; 2; 1]
printfn "The sum of the greater of each pair of elements in the two lists is %d." sum
```

`List.fold` и List. Scan различаются в, `List.fold` возвращающие окончательное значение дополнительного параметра, но `List.scan` возвращает список промежуточных значений (вместе с окончательным значением) дополнительного параметра.

Каждая из этих функций включает обратную вариацию, например List. foldBack, которая отличается в порядке обхода списка и порядком аргументов. Кроме того `List.fold` , `List.foldBack` и имеют варианты List. fold2 и List. foldBack2, которые принимают два списка одинаковой длины. Функция, которая выполняется по каждому элементу, может использовать соответствующие элементы обоих списков для выполнения некоторых действий. Типы элементов этих списков могут отличаться, как в следующем примере, где один список содержит суммы транзакций на банковском счете, а другой — типы транзакций (внесение или снятие).

```
// Discriminated union type that encodes the transaction type.
type Transaction =
    | Deposit
    | Withdrawal

let transactionTypes = [Deposit; Deposit; Withdrawal]
let transactionAmounts = [100.00; 1000.00; 95.00 ]
let initialBalance = 200.00

// Use fold2 to perform a calculation on the list to update the account balance.
let endingBalance = List.fold2 (fun acc elem1 elem2 ->
                                match elem1 with
                                | Deposit -> acc + elem2
                                | Withdrawal -> acc - elem2)
                                initialBalance
                                transactionTypes
                                transactionAmounts
printfn "%f" endingBalance
```

При расчете суммы функции `List.fold` и `List.foldBack` действуют одинаково, так как результат не зависит от порядка обхода. В следующем примере кода функция `List.foldBack` используется для добавления элемента в список.

```
let sumListBack list = List.foldBack (fun acc elem -> acc + elem) list 0
printfn "%d" (sumListBack [1; 2; 3])

// For a calculation in which the order of traversal is important, fold and foldBack have different
// results. For example, replacing fold with foldBack in the listReverse function
// produces a function that copies the list, rather than reversing it.
let copyList list = List.foldBack (fun elem acc -> elem::acc) list []
printfn "%A" (copyList [1 .. 10])
```

В следующем примере снова используется банковский счет. В этот раз добавляется новый тип транзакций: расчет процентов. Конечный баланс теперь зависит от порядка транзакций.

```
type Transaction2 =
    | Deposit
    | Withdrawal
    | Interest

let transactionTypes2 = [Deposit; Deposit; Withdrawal; Interest]
let transactionAmounts2 = [100.00; 1000.00; 95.00; 0.05 / 12.0 ]
let initialBalance2 = 200.00

// Because fold2 processes the lists by starting at the head element,
// the interest is calculated last, on the balance of 1205.00.
let endingBalance2 = List.fold2 (fun acc elem1 elem2 ->
                                 match elem1 with
                                 | Deposit -> acc + elem2
                                 | Withdrawal -> acc - elem2
                                 | Interest -> acc * (1.0 + elem2))
                                 initialBalance2
                                 transactionTypes2
                                 transactionAmounts2
printfn "%f" endingBalance2


// Because foldBack2 processes the lists by starting at end of the list,
// the interest is calculated first, on the balance of only 200.00.
let endingBalance3 = List.foldBack2 (fun elem1 elem2 acc ->
                                 match elem1 with
                                 | Deposit -> acc + elem2
                                 | Withdrawal -> acc - elem2
                                 | Interest -> acc * (1.0 + elem2))
                                 transactionTypes2
                                 transactionAmounts2
                                 initialBalance2
printfn "%f" endingBalance3
```

Список функций . reduce в некоторой степени `List.fold` аналогичен `List.scan` и, за исключением того, что вместо `List.reduce` передачи отдельного агрегата функция принимает функцию, которая принимает два аргумента типа элемента, а не только один, и один из них аргументы действуют как совокупность, что означает сохранение промежуточного результата вычисления. `List.reduce` начинается с работы на первых двух элементах списка, а затем использует результат операции вместе со следующим элементом. Так как здесь нет отдельного аккумулятора с собственным типом, функция `List.reduce` может использоваться вместо `List.fold` только в том случае, если аккумулятор и элемент имеют одинаковые типы. В следующем коде показано использование функции `List.reduce`. `List.reduce` создает исключение, если предоставленный список не содержит элементов.

В следующем коде первый вызов лямбда-выражения дает аргументы 2 и 4 и возвращает 6. В следующем вызове даются аргументы 6 и 10 и возвращается результат 16.

```
let sumAList list =
    try
        List.reduce (fun acc elem -> acc + elem) list
    with
       | :? System.ArgumentException as exc -> 0

let resultSum = sumAList [2; 4; 10]
printfn "%d " resultSum
```

Конвертация списков и другие типы коллекций

Модуль `List` предоставляет функции для прямой и обратной конвертации обоих последовательностей и массивов. Для преобразования в последовательность или из последовательности используйте List. toSeq или List. ofSeq. Для преобразования в массив или из массива используйте List. ToArray или List.

офаррай.

Дополнительные операции

Дополнительные сведения о дополнительных операциях со списками см. в разделе Справочник по библиотеке Collections. List.

См. также

- Справочник по языку F#
- Типы языка F#
- Последовательности
- Массивы
- Варианты

Массивы

23.10.2019 • 26 minutes to read • Edit Online

NOTE

Ссылка на справочник по API ведет на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

Массивы — это изменяемые коллекции последовательных элементов данных с фиксированным размером (от нуля), которые имеют один и тот же тип.

Создание массивов

Массивы можно создавать несколькими способами. Можно создать небольшой массив, перечисляя последовательные значения между `[|` и `|]` и разделив их точкой с запятой, как показано в следующих примерах.

```
let array1 = [| 1; 2; 3 |]
```

Можно также разместить каждый элемент на отдельной строке, в этом случае разделитель точкой с запятой является необязательным.

```
let array1 =
    [|
        1
        2
        3
     |]
```

Тип элементов массива выводится из используемых литералов и должен быть единообразным. Следующий код вызывает ошибку, поскольку 1,0 является числом с плавающей запятой и 2, а 3 — целыми числами.

```
// Causes an error.
// let array2 = [| 1.0; 2; 3 |]
```

Для создания массивов также можно использовать выражения последовательности. Ниже приведен пример, создающий массив квадратов целых чисел от 1 до 10.

```
let array3 = [| for i in 1 .. 10 -> i * i |]
```

Чтобы создать массив, в котором все элементы инициализируются нулем, используйте `Array.zeroCreate`.

```
let arrayOfTenZeroes : int array = Array.zeroCreate 10
```

Доступ к элементам

Доступ к элементам массива можно получить с помощью оператора с точкой (`.`) и квадратных скобок (`[` и `]`).

```
array1.[0]
```

Индексы массива начинаются с 0.

Можно также получить доступ к элементам массива с помощью нотации среза, что позволяет указать поддиапазон массива. Ниже приведены примеры нотаций среза.

```
// Accesses elements from 0 to 2.

array1.[0..2]

// Accesses elements from the beginning of the array to 2.

array1.[..2]

// Accesses elements from 2 to the end of the array.

array1.[2..]
```

При использовании нотации среза создается новая копия массива.

Типы массивов и модули

Тип всех F# массивов является типом .NET Framework System.Array. Поэтому F# массивы поддерживают все функциональные возможности, доступные в System.Array.

Модуль библиотеки `Microsoft.FSharp.Collections.Array` поддерживает операции с одномерным массивами. Модули `Array2D` , `Array3D` и `Array4D` содержат функции, поддерживающие операции с массивами из двух, трех и четырех измерений соответственно. Массивы с рангом, равным четырем, можно создать с помощью System.Array.

Простые функции

`Array.get` Возвращает элемент. `Array.length` задает длину массива. `Array.set` задает для элемента указанное значение. В следующем примере кода показано использование этих функций.

```
let array1 = Array.create 10 ""
for i in 0 .. array1.Length - 1 do
    Array.set array1 i (i.ToString())
for i in 0 .. array1.Length - 1 do
    printf "%s " (Array.get array1 i)
```

Выходные данные выглядят следующим образом.

```
0 1 2 3 4 5 6 7 8 9
```

Функции, создающие массивы

Несколько функций создают массивы, не требуя наличия существующего массива. `Array.empty` создает новый массив, который не содержит элементов. `Array.create` создает массив указанного размера и задает для всех элементов указанные значения. `Array.init` создает массив с учетом измерения и функции для создания элементов. `Array.zeroCreate` создает массив, в котором все элементы инициализируются значением нулевого значения для типа массива. Эти функции показаны в следующем коде.

```
let myEmptyArray = Array.empty
printfn "Length of empty array: %d" myEmptyArray.Length



printfn "Array of floats set to 5.0: %A" (Array.create 10 5.0)


printfn "Array of squares: %A" (Array.init 10 (fun index -> index * index))

let (myZeroArray : float array) = Array.zeroCreate 10
```

Выходные данные выглядят следующим образом.

```
Length of empty array: 0
Area of floats set to 5.0: [|5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0|]
Array of squares: [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

`Array.copy` создает новый массив, содержащий элементы, скопированные из существующего массива. Обратите внимание, что копия является неполной копией, что означает, что если тип элемента является ссылочным, копируется только ссылка, а не базовый объект. Это показано в следующем примере кода.

```
open System.Text

let firstArray : StringBuilder array = Array.init 3 (fun index -> new StringBuilder(""))
let secondArray = Array.copy firstArray
// Reset an element of the first array to a new value.
firstArray.[0] <- new StringBuilder("Test1")
// Change an element of the first array.
firstArray.[1].Insert(0, "Test2") |> ignore
printfn "%A" firstArray
printfn "%A" secondArray
```

Результат выполнения приведенного выше кода выглядит следующим образом:

```
[|Test1; Test2; |]
[|; Test2; |]
```

Строка `Test1` отображается только в первом массиве, так как операция создания нового элемента перезаписывает ссылку в `firstArray`, но не влияет на исходную ссылку на пустую строку, которая все еще присутствует в `secondArray`. Строка `Test2` отображается в обоих массивах, так как операция `Insert` для типа System.Text.StringBuilder влияет на базовый объект System.Text.StringBuilder, на который имеется ссылка в обоих массивах.

`Array.sub` создает новый массив из поддиапазона массива. Укажите поддиапазон, указав начальный индекс и длину. В следующем коде показано использование функции `Array.sub`.

```
let a1 = [| 0 .. 99 |]
let a2 = Array.sub a1 5 10
printfn "%A" a2
```

Выходные данные показывают, что подмассив начинается в элементе 5 и содержит 10 элементов.

```
[|5; 6; 7; 8; 9; 10; 11; 12; 13; 14|]
```

`Array.append` создает новый массив, объединяя два существующих массива.

В следующем примере кода демонстрируется **массив. append**.

```
printfn "%A" (Array.append [| 1; 2; 3|] [| 4; 5; 6|])
```

Результат приведенного выше кода выглядит следующим образом.

```
[|1; 2; 3; 4; 5; 6|]
```

`Array.choose` выбирает элементы массива для включения в новый массив. Следующий код демонстрирует `Array.choose`. Обратите внимание, что тип элемента массива не обязательно должен совпадать с типом значения, возвращаемого в типе параметра. В этом примере тип элемента — `int`, а параметр — результат функции полинома, `elem*elem - 1` в качестве числа с плавающей запятой.

```
printfn "%A" (Array.choose (fun elem -> if elem % 2 = 0 then
                                            Some(float (elem*elem - 1))
                                        else
                                            None) [| 1 .. 10 |])
```

Результат приведенного выше кода выглядит следующим образом.

```
[|3.0; 15.0; 35.0; 63.0; 99.0|]
```

`Array.collect` выполняет указанную функцию для каждого элемента массива существующего массива, а затем собирает элементы, созданные функцией, и объединяет их в новый массив. Следующий код демонстрирует `Array.collect`.

```
printfn "%A" (Array.collect (fun elem -> [| 0 .. elem |]) [| 1; 5; 10|])
```

Результат приведенного выше кода выглядит следующим образом.

```
[|0; 1; 0; 1; 2; 3; 4; 5; 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
```

`Array.concat` принимает последовательность массивов и объединяет их в один массив. Следующий код демонстрирует `Array.concat`.

```
Array.concat [ [|0..3|] ; [|4|] ]
//output [|0; 1; 2; 3; 4|]

Array.concat [| [|0..3|] ; [|4|] |]
//output [|0; 1; 2; 3; 4|]
```

Результат приведенного выше кода выглядит следующим образом.

```
[|(1, 1, 1); (1, 2, 2); (1, 3, 3); (2, 1, 2); (2, 2, 4); (2, 3, 6); (3, 1, 3);
(3, 2, 6); (3, 3, 9)|]
```

`Array.filter` принимает логическую функцию Condition и создает новый массив, содержащий только те элементы из входного массива, для которых условие истинно. Следующий код демонстрирует `Array.filter`.

```
printfn "%A" (Array.filter (fun elem -> elem % 2 = 0) [| 1 .. 10|])
```

Результат приведенного выше кода выглядит следующим образом.

```
[|2; 4; 6; 8; 10|]
```

`Array.rev` создает новый массив, переменяя порядок существующего массива на другой. Следующий код демонстрирует `Array.rev`.

```
let stringReverse (s: string) =
    System.String(Array.rev (s.ToCharArray()))

printfn "%A" (stringReverse("!dlrow olleH"))
```

Результат приведенного выше кода выглядит следующим образом.

```
"Hello world!"
```

В модуле массива можно легко сочетать функции, которые преобразуют массивы с помощью конвейерного оператора (`|>`), как показано в следующем примере.

```
[| 1 .. 10 |]
|> Array.filter (fun elem -> elem % 2 = 0)
|> Array.choose (fun elem -> if (elem <> 8) then Some(elem*elem) else None)
|> Array.rev
|> printfn "%A"
```

Выходные данные:

```
[|100; 36; 16; 4|]
```

Многомерные массивы

Можно создать многомерный массив, но для записи литерала многомерного массива нет синтаксиса. Используйте оператор `array2D` , чтобы создать массив из последовательности последовательностей элементов массива. Последовательности могут быть литералами массива или списка. Например, следующий код создает двумерный массив.

```
let my2DArray = array2D [ [ 1; 0]; [0; 1] ]
```

Можно также использовать функцию `Array2D.init` для инициализации массивов двух измерений, и аналогичные функции доступны для массивов из трех и четырех измерений. Эти функции принимают функцию, которая используется для создания элементов. Чтобы создать двумерный массив, содержащий элементы, для которых задано начальное значение вместо указания функции, используйте функцию `Array2D.create` , которая также доступна для массивов размером до четырех. В следующем примере кода сначала показано, как создать массив массивов, содержащих нужные элементы, а затем использовать `Array2D.init` для создания нужного двумерного массива.

```
let arrayOfArrays = [| [| 1.0; 0.0 |]; [|0.0; 1.0 |] |]
let twoDimensionalArray = Array2D.init 2 2 (fun i j -> arrayOfArrays.[i].[j])
```

Синтаксис индексирования и среза массива поддерживается для массивов вплоть до ранга 4. При указании индекса в нескольких измерениях для разделения индексов используются запятые, как показано в следующем примере кода.

```
twoDimensionalArray.[0, 1] <- 1.0
```

Тип двумерного массива записывается как `<type>[,]` (например, `int[,]`, `double[,]`), а тип трехмерного массива записывается как `<type>[,,]` и т. д. для массивов с большими размерами.

Для многомерных массивов также доступен только подмножество функций, доступных для одномерных массивов. Дополнительные сведения см. в разделе `Collections.Array Module`, `Collections.Array2D Module`, `Collections.Array3D Module` и `Collections.Array4D Module`.

Создание срезов массива и многомерных массивов

В двухмерном массиве (матрице) можно извлечь подматрицу, указав диапазоны и используя символ шаблона (`*`) для указания целых строк или столбцов.

```
// Get rows 1 to N from an NxM matrix (returns a matrix):
matrix.[1.., *]

// Get rows 1 to 3 from a matrix (returns a matrix):
matrix.[1..3, *]

// Get columns 1 to 3 from a matrix (returns a matrix):
matrix.[*, 1..3]

// Get a 3x3 submatrix:
matrix.[1..3, 1..3]
```

Начиная с F# 3,1 можно разбивать многомерный массив на подмассивы того же или более низкого измерения. Например, можно получить вектор из матрицы, указав одну строку или столбец.

```
// Get row 3 from a matrix as a vector:
matrix.[3, *]

// Get column 3 from a matrix as a vector:
matrix.[*, 3]
```

Этот синтаксис среза можно использовать для типов, реализующих операторы доступа к элементам и перегруженные методы `GetSlice`. Например, следующий код создает тип матрицы, который заключает в F# оболочку 2D-массив, реализует свойство Item для обеспечения поддержки индексирования массива и реализует три версии `GetSlice`. Если этот код можно использовать в качестве шаблона для типов матриц, можно использовать все операции по фрагментированию, описанные в этом разделе.

```
type Matrix<'T>(N: int, M: int) =
    let internalArray = Array2D.zeroCreate<'T> N M

    member this.Item
        with get(a: int, b: int) = internalArray.[a, b]
        and set(a: int, b: int) (value:'T) = internalArray.[a, b] <- value

    member this.GetSlice(rowStart: int option, rowFinish : int option, colStart: int option, colFinish : int
option) =
        let rowStart =
            match rowStart with
            | Some(v) -> v
            | None -> 0
        let rowFinish =
            match rowFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(0) - 1
        let colStart =
            match colStart with
            | Some(v) -> v
            | None -> 0
        let colFinish =
            match colFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(1) - 1
        internalArray.[rowStart..rowFinish, colStart..colFinish]

    member this.GetSlice(row: int, colStart: int option, colFinish: int option) =
        let colStart =
            match colStart with
            | Some(v) -> v
            | None -> 0
        let colFinish =
            match colFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(1) - 1
        internalArray.[row, colStart..colFinish]

    member this.GetSlice(rowStart: int option, rowFinish: int option, col: int) =
        let rowStart =
            match rowStart with
            | Some(v) -> v
            | None -> 0
        let rowFinish =
            match rowFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(0) - 1
        internalArray.[rowStart..rowFinish, col]

module test =
    let generateTestMatrix x y =
        let matrix = new Matrix<float>(3, 3)
        for i in 0..2 do
            for j in 0..2 do
                matrix.[i, j] <- float(i) * x - float(j) * y
        matrix

    let test1 = generateTestMatrix 2.3 1.1
    let submatrix = test1.[0..1, 0..1]
    printfn "%A" submatrix

    let firstRow = test1.[0,*]
    let secondRow = test1.[1,*]
    let firstCol = test1.[*,0]
    printfn "%A" firstCol
```

Логические функции в массивах

Функции `Array.exists` и `Array.exists2` проверяют элементы либо в одном, либо в двух массивах соответственно. Эти функции принимают тестовую функцию и возвращают `true`, если существует элемент (или пара элементов для `Array.exists2`), удовлетворяющий условию.

В следующем коде демонстрируется использование `Array.exists` и `Array.exists2`. В этих примерах новые функции создаются путем применения только одного из аргументов, в таких случаях аргумент функции.

```
let allNegative = Array.exists (fun elem -> abs (elem) = elem) >> not
printfn "%A" (allNegative [| -1; -2; -3 |])
printfn "%A" (allNegative [| -10; -1; 5 |])
printfn "%A" (allNegative [| 0 |])


let haveEqualElement = Array.exists2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (haveEqualElement [| 1; 2; 3 |] [| 3; 2; 1|])
```

Результат приведенного выше кода выглядит следующим образом.

```
true
false
false
true
```

Аналогичным образом функция `Array.forall` проверяет массив, чтобы определить, удовлетворяет ли каждый элемент логическому условию. Вариант `Array.forall2` выполняет одно и то же действие с помощью логической функции, включающей элементы двух массивов с одинаковой длиной. В следующем коде показано использование этих функций.

```
let allPositive = Array.forall (fun elem -> elem > 0)
printfn "%A" (allPositive [| 0; 1; 2; 3 |])
printfn "%A" (allPositive [| 1; 2; 3 |])


let allEqual = Array.forall2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (allEqual [| 1; 2 |] [| 1; 2 |])
printfn "%A" (allEqual [| 1; 2 |] [| 2; 1 |])
```

Выходные данные для этих примеров выглядят следующим образом.

```
false
true
true
false
```

Поиск массивов

`Array.find` принимает логическую функцию и возвращает первый элемент, для которого функция возвращает `true`, или вызывает System.Collections.Generic.KeyNotFoundException, если не найдено ни одного элемента, удовлетворяющего условию. `Array.findIndex` аналогичен `Array.find`, за исключением того, что он возвращает индекс элемента, а не сам элемент.

В следующем коде используется `Array.find` и `Array.findIndex`, чтобы узнать число, которое является как идеальным квадратом, так и идеальным кубом.

```
let arrayA = [| 2 .. 100 |]
let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let element = Array.find (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
let index = Array.findIndex (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
printfn "The first element that is both a square and a cube is %d and its index is %d." element index
```

Выходные данные выглядят следующим образом.

```
The first element that is both a square and a cube is 64 and its index is 62.
```

`Array.tryFind` аналогичен `Array.find`, за исключением того, что его результат является типом параметра и возвращает `None`, если элемент не найден. `Array.tryFind` следует использовать вместо `Array.find`, если неизвестно, находится ли соответствующий элемент в массиве. Аналогичным образом `Array.tryFindIndex` аналогичен `Array.findIndex`, за исключением того, что тип параметра является возвращаемым значением. Если элемент не найден, параметр имеет значение `None`.

В следующем коде показано использование функции `Array.tryFind`. Этот код зависит от предыдущего кода.

```
let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let lookForCubeAndSquare array1 =
    let result = Array.tryFind (fun elem -> isPerfectSquare elem && isPerfectCube elem) array1
    match result with
    | Some x -> printfn "Found an element: %d" x
    | None -> printfn "Failed to find a matching element."

lookForCubeAndSquare [| 1 .. 10 |]
lookForCubeAndSquare [| 100 .. 1000 |]
lookForCubeAndSquare [| 2 .. 50 |]
```

Выходные данные выглядят следующим образом.

```
Found an element: 1
Found an element: 729
Failed to find a matching element.
```

Используйте `Array.tryPick`, если необходимо преобразовать элемент в дополнение к его поиску. Результатом является первый элемент, для которого функция возвращает преобразованный элемент в виде значения параметра, или значение `None`, если такой элемент не найден.

В следующем коде показано использование `Array.tryPick`. В этом случае вместо лямбда-выражения для упрощения кода определены несколько локальных вспомогательных функций.

```
let findPerfectSquareAndCube array1 =
    let delta = 1.0e-10
    let isPerfectSquare (x:int) =
        let y = sqrt (float x)
        abs(y - round y) < delta
    let isPerfectCube (x:int) =
        let y = System.Math.Pow(float x, 1.0/3.0)
        abs(y - round y) < delta
    // intFunction : (float -> float) -> int -> int
    // Allows the use of a floating point function with integers.
    let intFunction function1 number = int (round (function1 (float number)))
    let cubeRoot x = System.Math.Pow(x, 1.0/3.0)
    // testElement: int -> (int * int * int) option
    // Test an element to see whether it is a perfect square and a perfect
    // cube, and, if so, return the element, square root, and cube root
    // as an option value. Otherwise, return None.
    let testElement elem =
        if isPerfectSquare elem && isPerfectCube elem then
            Some(elem, intFunction sqrt elem, intFunction cubeRoot elem)
        else None
    match Array.tryPick testElement array1 with
    | Some (n, sqrt, cuberoot) -> printfn "Found an element %d with square root %d and cube root %d." n sqrt
cuberoot
    | None -> printfn "Did not find an element that is both a perfect square and a perfect cube."

findPerfectSquareAndCube [| 1 .. 10 |]
findPerfectSquareAndCube [| 2 .. 100 |]
findPerfectSquareAndCube [| 100 .. 1000 |]
findPerfectSquareAndCube [| 1000 .. 10000 |]
findPerfectSquareAndCube [| 2 .. 50 |]
```

Выходные данные выглядят следующим образом.

```
Found an element 1 with square root 1 and cube root 1.
Found an element 64 with square root 8 and cube root 4.
Found an element 729 with square root 27 and cube root 9.
Found an element 4096 with square root 64 and cube root 16.
Did not find an element that is both a perfect square and a perfect cube.
```

Выполнение вычислений с массивами

Функция `Array.average` Возвращает среднее значение каждого элемента в массиве. Ограничены типами элементов, которые поддерживают точное деление на целое число, которое включает типы с плавающей запятой, но не целочисленные типы. Функция `Array.averageBy` Возвращает среднее значение результатов вызова функции для каждого элемента. Для массива целочисленного типа можно использовать `Array.averageBy`, чтобы функция могла преобразовать каждый элемент в тип с плавающей запятой для вычисления.

Используйте `Array.max` или `Array.min`, чтобы получить максимальный или минимальный элемент, если тип элемента поддерживает его. Аналогичным образом `Array.maxBy` и `Array.minBy` позволяют сначала выполнить функцию, возможно, для преобразования в тип, поддерживающий сравнение.

`Array.sum` добавляет элементы массива, а `Array.sumBy` вызывает функцию для каждого элемента и добавляет результаты вместе.

Чтобы выполнить функцию для каждого элемента в массиве без сохранения возвращаемых значений, используйте `Array.iter`. Для функции, включающей два массива одинаковой длины, используйте `Array.iter2`. Если необходимо также удержать массив результатов функции, используйте `Array.map` или `Array.map2`, который работает с двумя массивами за раз.

Варианты `Array.iteri` и `Array.iteri2` позволяют использовать индекс элемента в вычислениях; то же

самое относится к `Array.mapi` и `Array.mapi2`.

Функции `Array.fold`, `Array.foldBack`, `Array.reduce`, `Array.reduceBack`, `Array.scan` и 1 выполняют алгоритмы выполнения, использующие все элементы массива. Аналогичным образом, варианты `Array.fold2` и `Array.foldBack2` выполняют вычисления с двумя массивами.

Эти функции для выполнения вычислений соответствуют функциям с одинаковыми именами в модуле List. Примеры использования см. в разделе списки.

Изменение массивов

`Array.set` задает для элемента указанное значение. `Array.fill` задает указанное значение для диапазона элементов в массиве. В следующем коде приведен пример `Array.fill`.

```
let arrayFill1 = [| 1 .. 25 |]
Array.fill arrayFill1 2 20 0
printfn "%A" arrayFill1
```

Выходные данные выглядят следующим образом.

```
[|1; 2; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 23; 24; 25|]
```

Для копирования подраздела одного массива в другой массив можно использовать `Array.blit`.

Преобразование в другие типы и из них

`Array.ofList` создает массив из списка. `Array.ofSeq` создает массив из последовательности. `Array.toList` и `Array.toSeq` преобразуют в другие типы коллекций из типа массива.

Сортировка массивов

Используйте `Array.sort` для сортировки массива с помощью универсальной функции сравнения. Используйте `Array.sortBy`, чтобы указать функцию, которая создает значение, называемое *ключом*, для сортировки с помощью универсальной функции сравнения для ключа. Если вы хотите предоставить пользовательскую функцию сравнения, используйте `Array.sortWith`. `Array.sort`, `Array.sortBy` и `Array.sortWith` все возвращают отсортированный массив как новый массив. Варианты `Array.sortInPlace`, `Array.sortInPlaceBy` и `Array.sortInPlaceWith` изменяют существующий массив вместо того, чтобы возвращать новый.

Массивы и кортежи

Функции `Array.zip` и `Array.unzip` преобразуют массивы пар кортежей в кортежи массивов и наоборот. `Array.zip3` и `Array.unzip3` похожи, за исключением того, что они работают с кортежами трех элементов или кортежей из трех массивов.

Параллельные вычисления на массивах

Модуль `Array.Parallel` содержит функции для выполнения параллельных вычислений с массивами. Этот модуль недоступен в приложениях, предназначенных для версий .NET Framework до версии 4.

См. также

- Справочник по языку F#
- Типы языка F#

Последовательности

25.11.2019 • 28 minutes to read • Edit Online

NOTE

Ссылки на справочник по API в этой статье ведут на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

Последовательность — это логический ряд элементов одного типа. Последовательности особенно полезны при наличии большой упорядоченной коллекции данных, но не обязательно должны использовать все элементы. Отдельные элементы последовательности вычисляются только по мере необходимости, поэтому последовательность может обеспечить лучшую производительность, чем список в ситуациях, когда не все элементы используются. Последовательности представлены типом `seq<'T>`, который является псевдонимом для IEnumerable<T>. Таким образом, в качестве последовательности можно использовать любой тип .NET, реализующий интерфейс IEnumerable<T>. Модуль Seq обеспечивает поддержку манипуляций с последовательностями.

Выражения последовательности

Выражение последовательности — это выражение, результатом которого является последовательность. Выражения последовательности могут принимать несколько форм. В самой простой форме указывается диапазон. Например, `seq { 1 .. 5 }` создает последовательность, содержащую пять элементов, включая конечные точки 1 и 5. Можно также указать шаг приращения (или уменьшение) между двумя двойными точками. Например, следующий код создает последовательность кратных 10.

```
// Sequence that has an increment.
seq { 0 .. 10 .. 100 }
```

Выражения последовательности состоят из F# выражений, создающих значения последовательности. Значения также можно формировать программно:

```
seq { for i in 1 .. 10 -> i * i }
```

В предыдущем примере используется оператор `->`, который позволяет указать выражение, значение которого станет частью последовательности. `->` можно использовать только в том случае, если каждая часть кода, следующая за ней, возвращает значение.

Кроме того, можно указать ключевое слово `do` с необязательным `yield` следующим образом:

```
seq { for i in 1 .. 10 do yield i * i }

// The 'yield' is implicit and doesn't need to be specified in most cases.
seq { for i in 1 .. 10 do i * i }
```

Следующий код создает список пар координат и индексов в массиве, представляющем сетку. Обратите внимание, что для первого выражения `for` требуется указать `do`.

```
let (height, width) = (10, 10)

seq {
    for row in 0 .. width - 1 do
        for col in 0 .. height - 1 ->
            (row, col, row*width + col)
    }
```

Выражение `if`, используемое в последовательности, является фильтром. Например, чтобы создать последовательность только простых чисел, при условии, что имеется функция, `isprime` типа `int -> bool`, создайте последовательность следующим образом.

```
seq { for n in 1 .. 100 -> if isprime n then n }
```

Как упоминалось ранее, `do` требуется здесь, так как нет `else` ветви, которая перемещается с `if`. При попытке использовать `->` вы получите сообщение об ошибке, сообщающее, что не все ветви возвращают значение.

Ключевое слово `yield!`

Иногда может потребоваться включить последовательность элементов в другую последовательность. Чтобы включить последовательность в другую последовательность, необходимо использовать ключевое слово `yield!`:

```
// Repeats '1 2 3 4 5' ten times
seq {
    for _ in 1..10 do
        yield! seq { 1; 2; 3; 4; 5}
}
```

Другой способ подумать `yield!` заключается в том, что он выполняет сведение внутренней последовательности, а затем включает его в содержащую последовательность.

Если в выражении используется `yield!`, то все остальные одиночные значения должны использовать ключевое слово `yield`:

```
// Combine repeated values with their values
seq {
    for x in 1..10 do
        yield x
        yield! seq { for i in 1..x -> i}
}
```

Указание только `x` в предыдущем примере приведет к тому, что последовательность не будет формировать значения.

Примеры

В первом примере используется выражение последовательности, содержащее итерацию, фильтр и оператор yield для создания массива. Этот код выводит на консоль последовательность простых чисел от 1 до 100.

```
// Recursive isprime function.
let isprime n =
    let rec check i =
        i > n/2 || (n % i <> 0 && check (i + 1))
    check 2

let aSequence =
    seq {
        for n in 1..100 do
            if isprime n then
                n
    }

for x in aSequence do
    printfn "%d" x
```

В следующем примере создается таблица умножения, состоящая из кортежей из трех элементов, каждый из которых состоит из двух факторов и продукта:

```
let multiplicationTable =
    seq {
        for i in 1..9 do
            for j in 1..9 ->
                (i, j, i*j)
    }
```

В следующем примере показано использование `yield!` для объединения отдельных последовательностей в одну последнюю последовательность. В этом случае последовательности для каждого поддерева в двоичном дереве объединяются в рекурсивную функцию для создания конечной последовательности.

```
// Yield the values of a binary tree in a sequence.
type Tree<'a> =
   | Tree of 'a * Tree<'a> * Tree<'a>
   | Leaf of 'a

// inorder : Tree<'a> -> seq<'a>
let rec inorder tree =
    seq {
      match tree with
          | Tree(x, left, right) ->
              yield! inorder left
              yield x
              yield! inorder right
          | Leaf x -> yield x
    }

let mytree = Tree(6, Tree(2, Leaf(1), Leaf(3)), Leaf(9))
let seq1 = inorder mytree
printfn "%A" seq1
```

Использование последовательностей

Последовательности поддерживают многие из тех же функций, что и списки. Последовательности также поддерживают операции, такие как группирование и подсчет, с помощью функций создания ключа. Последовательности также поддерживают более разнообразные функции для извлечения подпоследовательностей.

Многие типы данных, такие как списки, массивы, наборы и карты, являются неявными

последовательностями, так как они являются перечислимыми коллекциями. Функция, которая принимает последовательность в качестве аргумента, работает с любыми общими F# типами данных в дополнение к любому типу данных .NET, реализующему `System.Collections.Generic.IEnumerable<'T>`. Сравните это с функцией, которая принимает список в качестве аргумента, который может принимать только списки. Тип `seq<'T>` — это аббревиатура типа для `IEnumerable<'T>`. Это означает, что любой тип, реализующий универсальный `System.Collections.Generic.IEnumerable<'T>`, включающий массивы, списки, наборы и карты F#в, а также большинство типов коллекций .NET, совместим с типом `seq` и может использоваться везде, где ожидается последовательность.

Функции модуля

Модуль Seq в пространстве имен Microsoft. FSharp. Collections содержит функции для работы с последовательностями. Эти функции также работают с списками, массивами, картами и наборами, так как все эти типы являются перечислимыми и поэтому могут рассматриваться как последовательности.

Создание последовательностей

Последовательности можно создавать с помощью выражений последовательности, как описано выше, или с помощью определенных функций.

Можно создать пустую последовательность с помощью Seq. Emptyили создать последовательность только одного указанного элемента с помощью Seq. Singleton.

```
let seqEmpty = Seq.empty
let seqOne = Seq.singleton 10
```

Для создания последовательности, для которой создаются элементы с помощью предоставляемой функции, можно использовать Seq. init . Вы также предоставляете размер последовательности. Эта функция аналогична List. init, за исключением того, что элементы не создаются до выполнения итерации по последовательности. В следующем коде показано использование `Seq.init`.

```
let seqFirst5MultiplesOf10 = Seq.init 5 (fun n -> n * 10)
Seq.iter (fun elem -> printf "%d " elem) seqFirst5MultiplesOf10
```

Выходные данные:

```
0 10 20 30 40
```

С помощью функции Seq. офаррай и Seq.<офлист> tможно создавать последовательности из массивов и списков. Однако массивы и списки можно также преобразовать в последовательности с помощью оператора приведения. В следующем коде показаны оба метода.

```
// Convert an array to a sequence by using a cast.
let seqFromArray1 = [| 1 .. 10 |] :> seq<int>

// Convert an array to a sequence by using Seq.ofArray.
let seqFromArray2 = [| 1 .. 10 |] |> Seq.ofArray
```

С помощью Seq. Castможно создать последовательность из слабо типизированной коллекции, например, определенных в `System.Collections`. Такие слабо типизированные коллекции имеют тип элемента `System.Object` и перечисляются с помощью неуниверсального `System.Collections.Generic.IEnumerable`1` типа. Следующий код иллюстрирует использование

`Seq.cast` для преобразования `System.Collections.ArrayList` в последовательность.

```
open System

let arr = ResizeArray<int>(10)

for i in 1 .. 10 do
    arr.Add(10)

let seqCast = Seq.cast arr
```

Можно определить бесконечные последовательности с помощью функции Seq. инитинфините . Для такой последовательности необходимо предоставить функцию, которая создает каждый элемент из индекса элемента. Бесконечные последовательности возможны из-за отложенного вычисления. элементы создаются по мере необходимости путем вызова указанной функции. В следующем примере кода создается неограниченная последовательность чисел с плавающей запятой, в данном случае чередующийся ряд квадратов последовательных целых чисел.

```
let seqInfinite =
    Seq.initInfinite (fun index ->
        let n = float (index + 1)
        1.0 / (n * n * (if ((index + 1) % 2 = 0) then 1.0 else -1.0)))

printfn "%A" seqInfinite
```

Seq. unfold создает последовательность из вычислительной функции, которая принимает состояние и преобразует его для создания каждого последующего элемента последовательности. Состояние — это просто значение, используемое для расчета каждого элемента, и может изменяться при вычислении каждого элемента. Вторым аргументом для `Seq.unfold` является начальное значение, используемое для запуска последовательности. `Seq.unfold` использует тип параметра для состояния, что позволяет завершить последовательность, возвращая значение `None`. В следующем коде показаны два примера последовательностей: `seq1` и `fib`, которые создаются операцией `unfold`. Первая, `seq1`, — это просто последовательность с числами до 20. Во втором `fib` используется `unfold` для вычислить последовательность Фибоначчи. Поскольку каждый элемент последовательности Фибоначчи является суммой двух предыдущих чисел Фибоначчи, значение состояния является кортежем, состоящим из двух предыдущих чисел в последовательности. Начальное значение — `(1,1)`, первые два числа в последовательности.

```
let seq1 =
    0 // Initial state
    |> Seq.unfold (fun state ->
        if (state > 20) then
            None
        else
            Some(state, state + 1))

printfn "The sequence seq1 contains numbers from 0 to 20."

for x in seq1 do
    printf "%d " x

let fib =
    (1, 1) // Initial state
    |> Seq.unfold (fun state ->
        if (snd state > 1000) then
            None
        else
            Some(fst state + snd state, (snd state, fst state + snd state)))

printfn "\nThe sequence fib contains Fibonacci numbers."
for x in fib do printf "%d " x
```

Выходные данные выглядят следующим образом:

```
The sequence seq1 contains numbers from 0 to 20.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

The sequence fib contains Fibonacci numbers.

2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Ниже приведен пример кода, в котором используются многие из описанных здесь функций модуля последовательности для создания и расчета значений бесконечных последовательностей. Выполнение кода может занять несколько минут.

```
// generateInfiniteSequence generates sequences of floating point
// numbers. The sequences generated are computed from the fDenominator
// function, which has the type (int -> float) and computes the
// denominator of each term in the sequence from the index of that
// term. The isAlternating parameter is true if the sequence has
// alternating signs.
let generateInfiniteSequence fDenominator isAlternating =
    if (isAlternating) then
        Seq.initInfinite (fun index ->
            1.0 /(fDenominator index) * (if (index % 2 = 0) then -1.0 else 1.0))
    else
        Seq.initInfinite (fun index -> 1.0 /(fDenominator index))

// The harmonic alternating series is like the harmonic series
// except that it has alternating signs.
let harmonicAlternatingSeries = generateInfiniteSequence (fun index -> float index) true

// This is the series of reciprocals of the odd numbers.
let oddNumberSeries = generateInfiniteSequence (fun index -> float (2 * index - 1)) true

// This is the series of recipocals of the squares.
let squaresSeries = generateInfiniteSequence (fun index -> float (index * index)) false

// This function sums a sequence, up to the specified number of terms.
let sumSeq length sequence =
    (0, 0.0)
    |>
    Seq.unfold (fun state ->
        let subtotal = snd state + Seq.item (fst state + 1) sequence
        if (fst state >= length) then
            None
        else
            Some(subtotal, (fst state + 1, subtotal)))

// This function sums an infinite sequence up to a given value
// for the difference (epsilon) between subsequent terms,
// up to a maximum number of terms, whichever is reached first.
let infiniteSum infiniteSeq epsilon maxIteration =
    infiniteSeq
    |> sumSeq maxIteration
    |> Seq.pairwise
    |> Seq.takeWhile (fun elem -> abs (snd elem - fst elem) > epsilon)
    |> List.ofSeq
    |> List.rev
    |> List.head
    |> snd

// Compute the sums for three sequences that converge, and compare
// the sums to the expected theoretical values.
let result1 = infiniteSum harmonicAlternatingSeries 0.00001 100000
printfn "Result: %f  ln2: %f" result1 (log 2.0)

let pi = Math.PI
let result2 = infiniteSum oddNumberSeries 0.00001 10000
printfn "Result: %f pi/4: %f" result2 (pi/4.0)

// Because this is not an alternating series, a much smaller epsilon
// value and more terms are needed to obtain an accurate result.
let result3 = infiniteSum squaresSeries 0.0000001 1000000
printfn "Result: %f pi*pi/6: %f" result3 (pi*pi/6.0)
```

Поиск и поиск элементов

Функции поддержки последовательностей доступны в списках: Seq. Exists, Seq. exists2-, Seq. Find, Seq. findIndex, Seq. Pick, Seq. tryFindи Seq. tryFindIndex. Версии этих функций, доступные для

последовательностей, оценивают последовательность только до элемента, для которого выполняется поиск. Примеры см. в разделе списки.

Получение подпоследовательностей

Seq. Filter и Seq. Choose подобны соответствующим функциям, доступным для списков, за исключением того, что фильтрация и выбор не выполняются до оценки элементов последовательности.

Seq. truncate создает последовательность из другой последовательности, но ограничивает ее заданным числом элементов. Seq. take создает новую последовательность, содержащую только заданное число элементов из начала последовательности. Если число элементов в последовательности меньше, чем указано, `Seq.take` выдает `System.InvalidOperationException`. Различие между `Seq.take` и `Seq.truncate` заключается в том, что `Seq.truncate` не создает ошибку, если число элементов меньше указанного числа.

В следующем коде показано поведение и различия между `Seq.truncate` и `Seq.take`.

```
let mySeq = seq { for i in 1 .. 10 -> i*i }
let truncatedSeq = Seq.truncate 5 mySeq
let takenSeq = Seq.take 5 mySeq

let truncatedSeq2 = Seq.truncate 20 mySeq
let takenSeq2 = Seq.take 20 mySeq

let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""

// Up to this point, the sequences are not evaluated.
// The following code causes the sequences to be evaluated.
truncatedSeq |> printSeq
truncatedSeq2 |> printSeq
takenSeq |> printSeq
// The following line produces a run-time error (in printSeq):
takenSeq2 |> printSeq
```

Выходные данные до возникновения ошибки выглядят следующим образом.

```
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
```

С помощью Seq. TakeWhileможно указать функцию предиката (логическую функцию) и создать последовательность из другой последовательности, состоящие из элементов исходной последовательности, для которых предикат `true`, но останавливаться перед первым элементом для , который предикат возвращает `false`. Seq. Skip возвращает последовательность, которая пропускает заданное число первых элементов другой последовательности и возвращает остальные элементы. Seq. SkipWhile возвращает последовательность, которая пропускает первые элементы другой последовательности, если предикат возвращает `true`, а затем возвращает оставшиеся элементы, начиная с первого элемента, для которого предикат возвращает `false`.

В следующем примере кода показано поведение и различия между `Seq.takeWhile`, `Seq.skip` и `Seq.skipWhile`.

```
// takeWhile
let mySeqLessThan10 = Seq.takeWhile (fun elem -> elem < 10) mySeq
mySeqLessThan10 |> printSeq

// skip
let mySeqSkipFirst5 = Seq.skip 5 mySeq
mySeqSkipFirst5 |> printSeq

// skipWhile
let mySeqSkipWhileLessThan10 = Seq.skipWhile (fun elem -> elem < 10) mySeq
mySeqSkipWhileLessThan10 |> printSeq
```

Выходные данные выглядят следующим образом.

```
1 4 9
36 49 64 81 100
16 25 36 49 64 81 100
```

Преобразование последовательностей

Функция Seq. парная создает новую последовательность, в которой последовательные элементы входной последовательности группируются по кортежам.

```
let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""
let seqPairwise = Seq.pairwise (seq { for i in 1 .. 10 -> i*i })
printSeq seqPairwise

printfn ""
let seqDelta = Seq.map (fun elem -> snd elem - fst elem) seqPairwise
printSeq seqDelta
```

Seq. windowd выглядит так `Seq.pairwise`, за исключением того, что вместо создания последовательности кортежей он создает последовательность массивов, содержащую копии смежных элементов (*окно*) из последовательности. Вы указываете число соседних элементов в каждом массиве.

В следующем коде показано использование функции `Seq.windowed`. В этом случае число элементов в окне равно 3. В примере используется `printSeq`, который определен в предыдущем примере кода.

```
let seqNumbers = [ 1.0; 1.5; 2.0; 1.5; 1.0; 1.5 ] :> seq<float>
let seqWindows = Seq.windowed 3 seqNumbers
let seqMovingAverage = Seq.map Array.average seqWindows
printfn "Initial sequence: "
printSeq seqNumbers
printfn "\nWindows of length 3: "
printSeq seqWindows
printfn "\nMoving average: "
printSeq seqMovingAverage
```

Выходные данные выглядят следующим образом.

Начальная последовательность:

```
1.0 1.5 2.0 1.5 1.0 1.5

Windows of length 3:
[|1.0; 1.5; 2.0|] [|1.5; 2.0; 1.5|] [|2.0; 1.5; 1.0|] [|1.5; 1.0; 1.5|]

Moving average:
1.5 1.666666667 1.5 1.333333333
```

Операции с несколькими последовательностями

Seq. zip и Seq. zip3 принимают две или три последовательности и создают последовательность кортежей. Эти функции подобны соответствующим функциям, доступным для списков. Нет соответствующей функциональности для разделения одной последовательности на две или более последовательностей. Если эта функция необходима для последовательности, преобразуйте последовательность в список и используйте List. unzip.

Сортировка, сравнение и группирование

Функции сортировки, поддерживаемые для списков, также работают с последовательностями. Сюда входят Seq. Sort и Seq. sortBy. Эти функции выполняют итерацию всей последовательности.

Для сравнения двух последовательностей используется функция Seq. компаревис . Функция сравнивает последовательные элементы в свою очередь и останавливается при обнаружении первой неравной пары. Все дополнительные элементы не участвуют в сравнении.

В следующем коде показано использование `Seq.compareWith`.

```
let sequence1 = seq { 1 .. 10 }
let sequence2 = seq { 10 .. -1 .. 1 }

// Compare two sequences element by element.
let compareSequences =
    Seq.compareWith (fun elem1 elem2 ->
        if elem1 > elem2 then 1
        elif elem1 < elem2 then -1
        else 0)

let compareResult1 = compareSequences sequence1 sequence2
match compareResult1 with
| 1 -> printfn "Sequence1 is greater than sequence2."
| -1 -> printfn "Sequence1 is less than sequence2."
| 0 -> printfn "Sequence1 is equal to sequence2."
| _ -> failwith("Invalid comparison result.")
```

В предыдущем коде вычисляются и анализируются только первый элемент, а результат равен-1.

Seq. каунтби принимает функцию, которая создает значение, именуемое *ключом* для каждого элемента. Ключ создается для каждого элемента путем вызова этой функции для каждого элемента. `Seq.countBy` возвращает последовательность, содержащую значения ключа, и количество элементов, которые создали каждое значение ключа.

```
let mySeq1 = seq { 1.. 100 }

let printSeq seq1 = Seq.iter (printf "%A ") seq1

let seqResult =
    mySeq1
    |> Seq.countBy (fun elem ->
        if elem % 3 = 0 then 0
        elif elem % 3 = 1 then 1
        else 2)

printSeq seqResult
```

Выходные данные выглядят следующим образом.

```
(1, 34) (2, 33) (0, 33)
```

В предыдущих выходных данных показано, что имелось 34 элементов исходной последовательности, которые производили значение ключа 1, 33, которое вызвало ключ 2, и значение 33, полученное с помощью ключа 0.

Элементы последовательности можно сгруппировать, вызвав Seq. GroupBy. `Seq.groupBy` принимает последовательность и функцию, которая создает ключ из элемента. Функция выполняется для каждого элемента последовательности. `Seq.groupBy` возвращает последовательность кортежей, где первый элемент каждого кортежа является ключом, а второй — последовательностью элементов, которые создают этот ключ.

В следующем примере кода показано использование `Seq.groupBy` для разделения последовательности чисел от 1 до 100 на три группы, имеющие уникальные значения ключа 0, 1 и 2.

```
let sequence = seq { 1 .. 100 }

let printSeq seq1 = Seq.iter (printf "%A ") seq1

let sequences3 =
    sequences
    |> Seq.groupBy (fun index ->
        if (index % 3 = 0) then 0
        elif (index % 3 = 1) then 1
        else 2)

sequences3 |> printSeq
```

Выходные данные выглядят следующим образом.

```
(1, seq [1; 4; 7; 10; ...]) (2, seq [2; 5; 8; 11; ...]) (0, seq [3; 6; 9; 12; ...])
```

Можно создать последовательность, которая устраняет повторяющиеся элементы, вызывая Seq. DISTINCT. Или можно использовать Seq. дистинктби, который принимает функцию создания ключа для каждого элемента. Результирующая последовательность содержит элементы исходной последовательности, имеющие уникальные ключи. последующие элементы, которые создают дубликат ключа для более раннего элемента, отбрасываются.

В следующем примере кода показано использование `Seq.distinct` . `Seq.distinct` демонстрируется путем создания последовательностей, представляющих двоичные числа, а затем показывается, что единственными отдельными элементами являются 0 и 1.

```
let binary n =
    let rec generateBinary n =
        if (n / 2 = 0) then [n]
        else (n % 2) :: generateBinary (n / 2)

    generateBinary n
    |> List.rev
    |> Seq.ofList

printfn "%A" (binary 1024)

let resultSequence = Seq.distinct (binary 1024)
printfn "%A" resultSequence
```

Следующий код демонстрирует `Seq.distinctBy`, начиная с последовательности, содержащей отрицательные и положительные числа, и используя функцию абсолютного значения в качестве функции создания ключа. В результирующей последовательности отсутствуют все положительные числа, соответствующие отрицательным числам в последовательности, так как отрицательные числа отображаются ранее в последовательности и, следовательно, выбираются вместо положительных чисел, имеющих одинаковые абсолютные значения значение или ключ.

```
let inputSequence = { -5 .. 10 }
let printSeq seq1 = Seq.iter (printf "%A ") seq1

printfn "Original sequence: "
printSeq inputSequence

printfn "\nSequence with distinct absolute values: "
let seqDistinctAbsoluteValue = Seq.distinctBy (fun elem -> abs elem) inputSequence
printSeq seqDistinctAbsoluteValue
```

Последовательностей только для чтения и кэширование

Seq. ReadOnly создает копию последовательности, доступную только для чтения. `Seq.readonly` удобно использовать, если имеется коллекция для чтения и записи, например массив, и вы не хотите изменять исходную коллекцию. Эта функция может использоваться для сохранения инкапсуляции данных. В следующем примере кода создается тип, содержащий массив. Свойство предоставляет массив, но вместо возвращения массива он возвращает последовательность, созданную из массива с помощью `Seq.readonly`.

```
type ArrayContainer(start, finish) =
    let internalArray = [| start .. finish |]
    member this.RangeSeq = Seq.readonly internalArray
    member this.RangeArray = internalArray

let newArray = new ArrayContainer(1, 10)
let rangeSeq = newArray.RangeSeq
let rangeArray = newArray.RangeArray
// These lines produce an error:
//let myArray = rangeSeq :> int array
//myArray.[0] <- 0
// The following line does not produce an error.
// It does not preserve encapsulation.
rangeArray.[0] <- 0
```

Seq. Cache создает сохраненную версию последовательности. Используйте `Seq.cache`, чтобы избежать повторного вычисления последовательности или при наличии нескольких потоков, использующих последовательность, но необходимо убедиться, что каждый элемент действует только один раз. При

наличии последовательности, используемой несколькими потоками, у вас может быть один поток, который перечисляет и выполняет вычисление значений для исходной последовательности, а оставшиеся потоки могут использовать кэшированную последовательность.

Выполнение вычислений в последовательностях

Простые арифметические операции аналогичны спискам, например Seq. Average, Seq. Sum, Seq. averageBy, Seq. sumByи т. д.

Функции Seq. fold, Seq. reduceи Seq. Scan подобны соответствующим функциям, доступным для списков. Последовательности поддерживают подмножество всех вариантов этих функций, которые поддерживаются в списках. Дополнительные сведения и примеры см. в разделе списки.

См. также

- Справочник по языку F#
- Типы языка F#

Срезы

11.01.2020 • 5 minutes to read • Edit Online

В F#срез — это подмножество любого типа данных, имеющего `GetSlice` метод в его определении или в расширении типав области. Чаще всего он используется с F# массивами и списками. В этой статье объясняется, как выполнять срезы F# из существующих типов и как определять собственные срезы.

Срезы похожи на индексаторы, но вместо получения одного значения из базовой структуры данных они получают несколько.

F#в настоящее время имеет встроенную поддержку для срезов строк, списков, массивов и двумерных массивов.

Базовые срезы с F# списками и массивами

Наиболее распространенными типами данных, которые являются срезами F# , являются списки и массивы. В следующем примере показано, как это сделать с помощью списков.

```
// Generate a list of 100 integers
let fullList = [ 1 .. 100 ]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullList.[1..5]
printfn "Small slice: %A" smallSlice

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullList.[..5]
printfn "Unbounded beginning slice: %A" unboundedBeginning

// Create a slice from an index to the end of the list
let unboundedEnd = fullList.[94..]
printfn "Unbounded end slice: %A" unboundedEnd
```

Массивы срезов так же, как и списки срезов:

```
// Generate an array of 100 integers
let fullArray = [| 1 .. 100 |]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullArray.[1..5]
printfn "Small slice: %A" smallSlice

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullArray.[..5]
printfn "Unbounded beginning slice: %A" unboundedBeginning

// Create a slice from an index to the end of the list
let unboundedEnd = fullArray.[94..]
printfn "Unbounded end slice: %A" unboundedEnd
```

Многомерные массивы с срезами

F#поддерживает многомерные массивы в F# основной библиотеке. Как и в случае с одномерным массивами, можно также использовать срезы многомерных массивов. Однако введение дополнительных измерений требует немного другого синтаксиса, чтобы можно было создавать срезы конкретных строк и столбцов.

В следующих примерах показано, как выполнить срез 2D-массива:

```
// Generate a 3x3 2D matrix
let A = array2D [[1;2;3];[4;5;6];[7;8;9]]
printfn "Full matrix:\n %A" A

// Take the first row
let row0 = A.[0,*]
printfn "Row 0: %A" row0

// Take the first column
let col0 = A.[*,0]
printfn "Column 0: %A" col0

// Take all rows but only two columns
let subA = A.[*,0..1]
printfn "%A" subA

// Take two rows and all columns
let subA' = A.[0..1,*]
printfn "%A" subA'

// Slice a 2x2 matrix out of the full 3x3 matrix
let twoByTwo = A.[0..1,0..1]
printfn "%A" twoByTwo
```

В F# настоящее время основная библиотека не определяет `GetSlice` для трехмерных массивов. Если вы хотите разделить трехмерные массивы или другие массивы большего размера, определите элемент `GetSlice` самостоятельно.

Определение срезов для других структур данных

F# Основная библиотека определяет срезы для ограниченного набора типов. Если вы хотите определить срезы для большего числа типов данных, это можно сделать либо в самом определении типа, либо в расширении типа.

Например, ниже показано, как можно определить срезы для класса ArraySegment<T>, чтобы обеспечить удобную обработку данных:

```
open System

type ArraySegment<'TItem> with
    member segment.GetSlice(start, finish) =
        let start = defaultArg start 0
        let finish = defaultArg finish segment.Count
        ArraySegment(segment.Array, segment.Offset + start, finish - start)

let arr = ArraySegment [| 1 .. 10 |]
let slice = arr.[2..5] //[ 3; 4; 5]
```

Используйте встраивание, чтобы избежать упаковки-преобразования в случае необходимости

При определении срезов для типа, который фактически является структурой, рекомендуется `inline` элемент `GetSlice`. F# Компилятор оптимизирует необязательные аргументы, избегая выделения кучи в результате среза. Это критически важно для конструкций среза, таких как Span<T>, которые не могут быть выделены в куче.

```
open System

type ReadOnlySpan<'T> with
    // Note the 'inline' in the member definition
    member inline sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)

type Span<'T> with
    // Note the 'inline' in the member definition
    member inline sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)

let printSpan (sp: Span<int>) =
    let arr = sp.ToArray()
    printfn "%A" arr

let sp = [| 1; 2; 3; 4; 5 |].AsSpan()
printSpan sp.[0..] // [|1; 2; 3; 4; 5|]
printSpan sp.[..5] // [|1; 2; 3; 4; 5|]
printSpan sp.[0..3] // [|1; 2; 3|]
printSpan sp.[1..2] // |2; 3|]
```

Встроенные F# срезы являются конечными

Все внутренние срезы в F# являются конечными включительно; то есть верхняя граница включается в срез. Для данного среза с начальным индексом `x` и конечным `y`ом индекса результирующий срез будет включать значение *ИС* .

```
// Define a new list
let xs = [1 .. 10]

printfn "%A" xs.[2..5] // Includes the 5th index
```

См. также:

- Индексированные свойства

Параметры

23.10.2019 • 6 minutes to read • Edit Online

Тип параметра в F# используется, если фактическое значение для именованного значения или переменной может не существовать. Параметр имеет базовый тип и может содержать значение этого типа или не может иметь значения.

Примечания

В следующем коде показана функция, которая создает тип параметра.

```
let keepIfPositive (a : int) = if a > 0 then Some(a) else None
```

Как видите, создается `a` `Some(a)` значение, если входные данные больше 0. `None` В противном случае создается.

Значение `None` используется, если параметр не имеет фактического значения. В противном случае `Some( ... )` выражение дает параметру значение. Значения `Some` `true` и `None` полезны при сопоставлении шаблонов, как в следующей функции `exists`, которая возвращает, если параметр имеет значение, а `false` если нет.

```
let exists (x : int option) =
    match x with
    | Some(x) -> true
    | None -> false
```

Использование параметров

Параметры обычно используются, если поиск не возвращает соответствующий результат, как показано в следующем коде.

```
let rec tryFindMatch pred list =
    match list with
    | head :: tail -> if pred(head)
                      then Some(head)
                      else tryFindMatch pred tail
    | [] -> None

// result1 is Some 100 and its type is int option.
let result1 = tryFindMatch (fun elem -> elem = 100) [ 200; 100; 50; 25 ]

// result2 is None and its type is int option.
let result2 = tryFindMatch (fun elem -> elem = 26) [ 200; 100; 50; 25 ]
```

В приведенном выше коде рекурсивный поиск по списку выполняется рекурсивно. Функция `tryFindMatch` принимает функцию `pred` предиката, которая возвращает логическое значение, и список для поиска. Если найден элемент, удовлетворяющий предикату, рекурсия завершается, а функция возвращает значение в виде параметра в выражении `Some(head)`. Рекурсия завершается при сопоставлении пустого списка. В этот момент значение `head` не было найдено и `None` возвращается.

Многие F# библиотечные функции, которые выполняют поиск в коллекции значений, которые могут или

не существовать, `option` возвращают тип. По соглашению эти функции начинаются с `try` префикса, `Seq.tryFindIndex` например.

Параметры также могут быть полезны, если значение может не существовать, например, если возможно, что при попытке создания значения возникнет исключение. Это показано в следующем примере кода.

```
open System.IO
let openFile filename =
    try
        let file = File.Open (filename, FileMode.Create)
        Some(file)
    with
        | ex -> eprintf "An exception occurred with message %s" ex.Message
                None
```

Функция в предыдущем примере имеет тип `string -> File option` , `File` так как она возвращает объект, если файл открыт успешно, и `None` если возникает исключение. `openFile` В зависимости от ситуации может не подходить к перехвату исключения, а не разрешать его распространение.

Кроме того, по-прежнему можно передать `null` или значение, равное NULL `Some` , в случае параметра. Обычно это следует избегать, и обычно это происходит в обычной F# программе, но возможно из-за природы ссылочных типов в .NET.

Свойства и методы параметров

Тип параметра поддерживает следующие свойства и методы.

СВОЙСТВО ИЛИ МЕТОД	ТИП	ОПИСАНИЕ
None	`'T option`	Статическое свойство, которое позволяет создать значение параметра, имеющее `None` значение.
Не задано	`bool`	Возвращает `true` , если параметр `None` имеет значение.
Часть	`bool`	Возвращает `true` , если параметр имеет значение, не `None` равное.
Некоторых	`'T option`	Статический член, создающий параметр, который имеет значение, не `None` равное.
Значение	`'T`	Возвращает базовое значение или создает исключение, `System.NullReferenceException` если значение равно. `None`

Модуль параметров

Существует модуль, параметр, содержащий полезные функции, которые выполняют операции с параметрами. Некоторые функции позволяют повторить функциональность свойств, но они полезны в контекстах, где требуется функция. Option. some и Option. None являются функциями модуля, которые проверяют, содержит ли параметр значение. Параметр. Get получает значение, если таковое имеется. Если

значение отсутствует, вызывается исключение `System.ArgumentException`.

Функция Option. Bind выполняет функцию для значения, если имеется значение. Функция должна принимать ровно один аргумент, и ее тип параметра должен быть типом параметра. Возвращаемое значение функции является другим типом параметра.

Модуль Option также включает функции, которые соответствуют функциям, доступным для списков, массивов, последовательностей и других типов коллекций. К этим функциям `Option.iter` относятся `Option.forall` `Option.map`, `Option.exists`, , `Option.foldBack`, , `Option.fold` и . `Option.count` Эти функции позволяют использовать параметры как коллекцию с нулевым или одним элементом. Дополнительные сведения и примеры см. в обсуждении функций сбора в списках.

Преобразование в другие типы

Параметры можно преобразовать в списки или массивы. При преобразовании параметра в любую из этих структур данных Результирующая структура данных не может содержать ни одного элемента. Чтобы преобразовать параметр в массив, используйте `Option.toArray`. Чтобы преобразовать параметр в список, используйте `Option.toList`.

См. также

- Справочник по языку F#
- Типы языка F#

Параметры значений

05.12.2019 • 2 minutes to read • Edit Online

Тип параметра значения в используется F# в следующих двух обстоятельствах:

1. Сценарий подходит для F# параметра.
2. Использование структуры обеспечивает выигрыш в производительности в вашем сценарии.

Не все сценарии с учетом производительности являются "решенными" с помощью структур. При использовании вместо ссылочных типов необходимо учитывать дополнительные затраты на копирование. Однако крупные F# программы часто создают множество необязательных типов, которые проходят через критические пути. в таких случаях структуры часто могут повысить общую производительность в течение всего времени существования программы.

Определение

Параметр значения определен как размеченное объединение структуры , похожее на тип параметра ссылки. Его определение можно рассматривать следующим образом:

```
[<StructuralEquality; StructuralComparison>]
[<Struct>]
type ValueOption<'T> =
    | ValueNone
    | ValueSome of 'T
```

Параметр value соответствует структурному равенству и сравнению. Основное отличие состоит в том, что имя, имя типа и имена вариантов, как и названия регистров, указывают, что это тип значения.

Использование параметров значения

Параметры значения используются так же, как и Параметры. `ValueSome` используется для указания на присутствие значения, а `ValueNone` используется, если отсутствует значение:

```
let tryParseDateTime (s: string) =
    match System.DateTime.TryParse(s) with
    | (true, dt) -> ValueSome dt
    | (false, _) -> ValueNone

let possibleDateString1 = "1990-12-25"
let possibleDateString2 = "This is not a date"

let result1 = tryParseDateTime possibleDateString1
let result2 = tryParseDateTime possibleDateString2

match (result1, result2) with
| ValueSome d1, ValueSome d2 -> printfn "Both are dates!"
| ValueSome d1, ValueNone -> printfn "Only the first is a date!"
| ValueNone, ValueSome d2 -> printfn "Only the second is a date!"
| ValueNone, ValueNone -> printfn "None of them are dates!"
```

Как и в случае с параметрами, соглашение об именовании для функции, возвращающей `ValueOption`, имеет префикс `try`.

Свойства и методы параметра value

В настоящее время имеется одно свойство параметров значения: `Value`. Если при вызове этого свойства значение не указано, вызывается InvalidOperationException.

Функции параметров значений

Модуль `ValueOption` в FSharp. Core содержит эквивалентные функции для модуля `Option`. Существует несколько отличий в имени, например `defaultValueArg`:

```
val defaultValueArg : arg:'T voption -> defaultValue:'T -> 'T
```

Это действует так же, как `defaultArg` в модуле `Option`, но работает вместо параметра значения.

См. также:

- Параметры

Результаты

04.11.2019 • 2 minutes to read • Edit Online

Начиная с F# 4,1 существует тип `Result<'T,'TFailure>`, который можно использовать для написания отказоустойчивого кода, который можно составить.

Синтаксис

```
// The definition of Result in FSharp.Core
[<StructuralEquality; StructuralComparison>]
[<CompiledName("FSharpResult`2")>]
[<Struct>]
type Result<'T,'TError> =
    | Ok of ResultValue:'T
    | Error of ErrorValue:'TError
```

Заметки

Обратите внимание, что тип результата — это отличительное объединение структуры, которое представляет собой еще F# один компонент, представленный в 4,1. Семантика структурного равенства применяется здесь.

Тип `Result` обычно используется в собственной обработке ошибок, которую часто называют раилвай-ориентированным программированием в F# сообществе. Следующий тривиальный пример демонстрирует этот подход.

```
// Define a simple type which has fields that can be validated
type Request =
    { Name: string
      Email: string }

// Define some logic for what defines a valid name.
//
// Generates a Result which is an Ok if the name validates;
// otherwise, it generates a Result which is an Error.
let validateName req =
    match req.Name with
    | null -> Error "No name found."
    | "" -> Error "Name is empty."
    | "bananas" -> Error "Bananas is not a name."
    | _ -> Ok req

// Similarly, define some email validation logic.
let validateEmail req =
    match req.Email with
    | null -> Error "No email found."
    | "" -> Error "Email is empty."
    | s when s.EndsWith("bananas.com") -> Error "No email from bananas.com is allowed."
    | _ -> Ok req

let validateRequest reqResult =
    reqResult
    |> Result.bind validateName
    |> Result.bind validateEmail

let test() =
    // Now, create a Request and pattern match on the result.
    let req1 = { Name = "Phillip"; Email = "phillip@contoso.biz" }
    let res1 = validateRequest (Ok req1)
    match res1 with
    | Ok req -> printfn "My request was valid! Name: %s Email %s" req.Name req.Email
    | Error e -> printfn "Error: %s" e
    // Prints: "My request was valid!  Name: Phillip Email: phillip@consoto.biz"

    let req2 = { Name = "Phillip"; Email = "phillip@bananas.com" }
    let res2 = validateRequest (Ok req2)
    match res2 with
    | Ok req -> printfn "My request was valid! Name: %s Email %s" req.Name req.Email
    | Error e -> printfn "Error: %s" e
    // Prints: "Error: No email from bananas.com is allowed."

test()
```

Как видите, в цепочку можно легко объединить различные функции проверки, если вы принудительно возвращаете `Result`. Это позволяет разбивать такие функциональные возможности на небольшие части, которые являются взаимокомпозициями по мере необходимости. Это также имеет добавленное *значение применения* сопоставления шаблонов в конце круга проверки, что в свою сторону принуждает к более высокой степени правильности программы.

См. также

- Размеченные объединения
- Соответствие шаблону

Универсальные шаблоны

23.10.2019 • 10 minutes to read • Edit Online

Значения функции, методы, свойства и агрегатные типы, например классы, записи и размеченные объединения, в F# могут быть *универсальными*. Универсальные конструкции содержат по меньшей мере один параметр типа, который обычно задается пользователем такой конструкции. Универсальные функции и типы позволяют писать код, который работает с множеством типов без повторения кода для каждого из них. В F# можно легко сделать код универсальным, так как зачастую код неявно определяется как универсальный механизмами определения типов и автоматического обобщения в компиляторе.

Синтаксис

```
// Explicitly generic function.
let function-name<type-parameters> parameter-list =
function-body

// Explicitly generic method.
[ static ] member object-identifer.method-name<type-parameters> parameter-list [ return-type ] =
method-body

// Explicitly generic class, record, interface, structure,
// or discriminated union.
type type-name<type-parameters> type-definition
```

Примечания

Объявления явно универсальной функции или явно универсального типа похоже на объявление неуниверсальных функций или типов, за исключением задания (и использования) параметров типа в угловых скобках после имени функции или типа.

Объявления часто являются неявно универсальными. Если вы не указываете полностью тип каждого параметра, составляющего функцию или тип, компилятор пытается определить тип каждого параметра, значения и переменной из написанного кода. Дополнительные сведения см. в статье Определение типа. Если код для типа или функции не ограничивают типы параметров иным образом, то функция или тип являются неявно универсальными. Этот процесс называется *автоматическим обобщением*. Автоматическое обобщение имеет некоторые ограничения. Например, если компилятору F# не удается определить типы для универсальной конструкции, он выдает ошибку, ссылающуюся на ограничение под названием *ограничение значения*. В этом случае может потребоваться добавить некоторые заметки с типом. Дополнительные сведения об автоматическом обобщении и ограничении значения, а также об изменении кода для решения этой проблемы см. в статье Автоматическое обобщение.

В приведенном выше синтаксисе *type-parameters* — это разделенный запятыми список параметров, представляющих неизвестные типы, каждый из которых начинается с одинарной кавычки. Может также присутствовать предложение ограничения, которое дополнительно ограничивает перечень допустимых типов для этого параметра типа. Синтаксис для различных предложений ограничения и прочие сведения об ограничениях см. в статье Ограничения.

Компонент *type-definition* в синтаксисе совпадает с определением типа для неуниверсального типа. Он содержит параметры конструктора для типа класса, необязательное предложение `as`, символ равенства, поля записей, предложение `inherit`, варианты для размеченного объединения, привязки `let` и `do`, определения элементов и все остальное, что разрешено в определении неуниверсального типа.

Остальные компоненты синтаксиса являются такими же, что и в неуниверсальных функциях и типах. Например, *object-identifier* — это идентификатор, представляющий сам содержащий объект.

Свойства, поля и конструкторы не могут быть более универсальными, чем включающий тип. Кроме того, значения в модуле не могут быть универсальными.

Неявно универсальные конструкции

Когда компилятор F# определяет типы в коде, он автоматически рассматривает как универсальную любую функцию, которая может быть таковой. Если указать тип явно, например, тип параметра, автоматическое обобщение не выполняется.

В следующем примере кода `makeList` является универсальной, хотя ни она, ни ее параметры не объявляются явно как универсальные.

```
let makeList a b =
    [a; b]
```

Подпись функции определяется как `'a -> 'a -> 'a list`. Обратите внимание, что `a` и `b` в этом примере определяются, как имеющие одинаковый тип. Это вызвано тем, что они включаются в список вместе, а все элементы списка должны иметь один тип.

Кроме того, функцию можно сделать универсальной, применив синтаксис с одинарной кавычкой в заметке типа, чтобы показать, что тип параметра является параметром универсального типа. В следующем коде `function1` является универсальной, так как ее параметры объявляются указанным образом — как параметры типа.

```
let function1 (x: 'a) (y: 'a) =
    printfn "%A %A" x y
```

Явно универсальные конструкции

Вы также можете сделать функцию универсальной, явно объявив ее параметры типа в угловых скобках (`<type-parameter>`). Это проиллюстрировано в следующем коде.

```
let function2<'T> x y =
    printfn "%A, %A" x y
```

Использование универсальных конструкций

При использовании универсальных функций или методов вам может быть не нужно указывать аргументы типа. Компилятор использует определение типа для выведения подходящих аргументов типа. Если по-прежнему присутствует неоднозначность, можно указать аргументы типа в угловых скобках, разделяя их запятыми.

Следующий код показывает использование функций, определенных в предыдущих разделах.

```
// In this case, the type argument is inferred to be int.
function1 10 20
// In this case, the type argument is float.
function1 10.0 20.0
// Type arguments can be specified, but should only be specified
// if the type parameters are declared explicitly. If specified,
// they have an effect on type inference, so in this example,
// a and b are inferred to have type int.
let function3 a b =
    // The compiler reports a warning:
    function1<int> a b
    // No warning.
    function2<int> a b
```

NOTE

Сослаться на универсальный тип по имени можно двумя способами. Например, `list<int>` и `int list` представляют два способа сослаться на универсальный тип `list` с одним аргументом типа `int`. Последняя форма обычно используется только со встроенными типами F#, такими как `list` и `option`. При наличии нескольких аргументов типа обычно используется синтаксис `Dictionary<int, string>`, но можно также использовать синтаксис `(int, string) Dictionary`.

Подстановочные знаки как аргументы типа

Чтобы указать, что аргумент типа должен определяться компилятором, можно использовать символ подчеркивания или подстановочный знак (`_`) вместо именованного аргумента типа. Это показано в приведенном ниже коде.

```
let printSequence (sequence1: Collections.seq<_>) =
   Seq.iter (fun elem -> printf "%s " (elem.ToString())) sequence1
```

Ограничения в универсальных типах и функциях

В определении универсального типа или универсальной функции можно использовать только те конструкции, которые доступны для параметра универсального типа. Это необходимо для проверки вызовов функций и методов во время компиляции. Если вы объявляете параметры типа явно, можно применить к параметру универсального типа явное ограничение, чтобы уведомить компилятор о доступности некоторых методов и функций. Тем не менее, если разрешить компилятору F# выводить универсальные типы параметров, он самостоятельно определит подходящие ограничения. Дополнительные сведения см. в статье Ограничения.

Статически разрешаемые параметры типов

Существует два вида параметров типа, которые можно использовать в программах на языке F#. Первый — это параметры универсального типа, описанные в предыдущих разделах. Первый вид параметров типа эквивалентен параметрам универсального типа, которые используются в таких языках, как Visual Basic и C#. Другой вид параметров типа имеется только в F# и называется *статически разрешаемым параметром типа*. Сведения об этих конструкциях см. в статье Статически разрешаемые параметры типов.

Примеры

```
// A generic function.
// In this example, the generic type parameter 'a makes function3 generic.
let function3 (x : 'a) (y : 'a) =
    printf "%A %A" x y

// A generic record, with the type parameter in angle brackets.
type GR<'a> =
    {
        Field1: 'a;
        Field2: 'a;
    }

// A generic class.
type C<'a>(a : 'a, b : 'a) =
    let z = a
    let y = b
    member this.GenericMethod(x : 'a) =
        printfn "%A %A %A" x y z

// A generic discriminated union.
type U<'a> =
    | Choice1 of 'a
    | Choice2 of 'a * 'a

type Test() =
    // A generic member
    member this.Function1<'a>(x, y) =
        printfn "%A, %A" x y

    // A generic abstract method.
    abstract abstractMethod<'a, 'b> : 'a * 'b -> unit
    override this.abstractMethod<'a, 'b>(x:'a, y:'b) =
         printfn "%A, %A" x y
```

См. также

- Справочник по языку
- Типы
- Статически разрешаемые параметры типов
- Универсальные шаблоны
- Автоматическое обобщение
- Ограничения

Автоматическое обобщение

23.10.2019 • 6 minutes to read • Edit Online

F#использует определение типа для вычисления типов функций и выражений. В этом разделе описывается F# , как автоматически обобщает аргументы и типы функций, чтобы они работали с несколькими типами, когда это возможно.

Автоматическое обобщение

F# Компилятор при выполнении определения типа для функции определяет, может ли заданный параметр быть универсальным. Компилятор проверяет каждый параметр и определяет, зависит ли функция от конкретного типа этого параметра. Если это не так, тип выводится как универсальный.

В следующем примере кода показана функция, которую компилятор выводит как универсальный.

```
let max a b = if a > b then a else b
```

Тип выводится `'a -> 'a -> 'a` как.

Тип указывает, что это функция, принимающая два аргумента одного и того же неизвестного типа и возвращающая значение того же типа. Одна из причин, по которой функция Previous может быть универсальной, заключается в том, что оператор `>` «больше» () является универсальным. Оператор "больше чем" имеет сигнатуру `'a -> 'a -> bool`. Не все операторы являются универсальными, и если код в функции использует тип параметра вместе с неуниверсальной функцией или оператором, этот тип параметра не может быть обобщенным.

Поскольку `max` является универсальным, его можно использовать с типами, такими `int` как `float`, и т. д., как показано в следующих примерах.

```
let biggestFloat = max 2.0 3.0
let biggestInt = max 2 3
```

Однако оба аргумента должны иметь один и тот же тип. Сигнатура — `'a -> 'a -> 'a`, а `'a -> 'b -> 'a` не. Поэтому следующий код выдает ошибку, так как типы не совпадают.

```
// Error: type mismatch.
let biggestIntFloat = max 2.0 3
```

`max` Функция также работает с любым типом, поддерживающим оператор "больше чем". Таким образом, его также можно использовать в строке, как показано в следующем коде.

```
let testString = max "cab" "cat"
```

Ограничение значения

Компилятор выполняет автоматическое обобщение только для полных определений функций, имеющих явные аргументы, и для простых неизменяемых значений.

Это означает, что компилятор выдает ошибку при попытке компиляции кода, который не ограничен

конкретным типом, но также не является обобщением. Сообщение об ошибке для этой проблемы связано с этим ограничением на автоматическую обобщение значений в качестве *ограничения*по значению.

Как правило, ошибка ограничения значения возникает либо в том случае, если требуется, чтобы конструкция была универсальной, но компилятор имеет недостаточную информацию для его обобщения, или если непреднамеренно опустить достаточно сведений о типе в неуниверсальной конструкции. Решением для ошибки ограничения значения является предоставление более явных сведений для более полного ограничения проблемы вывода типа одним из следующих способов:

- Ограничьте тип как неуниверсальный, добавив явную аннотацию типа к значению или параметру.
- Если проблема заключается в использовании необобщенной конструкции для определения универсальной функции, например композиции функций или неполностью примененных аргументов функции, попробуйте переписать функцию как обычное определение функции.
- Если проблема является выражением, которое является слишком сложным для обобщенного типа, сделайте его функцией, добавив дополнительный неиспользуемый параметр.
- Добавьте явные параметры универсального типа. Этот параметр редко используется.
- В следующих примерах кода демонстрируется каждый из этих сценариев.

Вариант 1. Слишком сложное выражение. В этом примере список `counter` должен быть `int option ref`, но не определен как простое неизменяемое значение.

```
let counter = ref None
// Adding a type annotation fixes the problem:
let counter : int option ref = ref None
```

Вариант 2. Использование необобщенной конструкции для определения универсальной функции. В этом примере конструкция не является обобщением, так как она включает в себя частичное применение аргументов функции.

```
let maxhash = max << hash
// The following is acceptable because the argument for maxhash is explicit:
let maxhash obj = (max << hash) obj
```

Вариант 3. Добавление дополнительного неиспользуемого параметра. Поскольку это выражение недостаточно просто для обобщения, компилятор выдает ошибку ограничения значения.

```
let emptyList10 = Array.create 10 []
// Adding an extra (unused) parameter makes it a function, which is generalizable.
let emptyList10 () = Array.create 10 []
```

Вариант 4. Добавление параметров типа.

```
let arrayOf10Lists = Array.create 10 []
// Adding a type parameter and type annotation lets you write a generic value.
let arrayOf10Lists<'T> = Array.create 10 ([]:'T list)
```

В последнем случае значение превращается в функцию типа, которая может использоваться для создания значений различных типов, например следующим образом:

```
let intLists = arrayOf10Lists<int>
let floatLists = arrayOf10Lists<float>
```

См. также

- Вывод типа
- Универсальные шаблоны
- Статически разрешаемые параметры типов
- Ограничения

Ограничения

05.11.2019 • 6 minutes to read • Edit Online

В этом разделе описываются ограничения, которые можно применять к параметрам универсального типа для указания требований для аргумента типа в универсальном типе или функции.

Синтаксис

```
type-parameter-list when constraint1 [ and constraint2]
```

Заметки

Существует несколько различных ограничений, которые можно применить для ограничения типов, которые могут использоваться в универсальном типе. В следующей таблице перечислены и описаны эти ограничения.

ОГРАНИЧЕНИЕ	СИНТАКСИС	ОПИСАНИЕ
Ограничение типа	*тип-параметр* :> *тип*	Указанный тип должен быть равен или производному от указанного типа, или, если тип является интерфейсом, предоставленный тип должен реализовывать интерфейс.
Ограничение null	*тип-параметр* : NULL	Указанный тип должен поддерживать литерал null. Сюда входят все типы объектов .NET, но F# не типы списка, кортежа, функции, класса, записи или объединения.
Ограничение явного члена	[(]*тип-параметр* [или... или *Type-Parameter*)]: (*подпись члена*)	По крайней мере один из указанных аргументов типа должен иметь член с указанной сигнатурой. не предназначено для общего использования. Члены должны быть либо явно определены в типе, либо частью расширения неявного типа, чтобы быть допустимыми целевыми объектами для ограничения явного члена.
Ограничение конструктора	*тип* — *параметр* : (New: unit-> 'a)	Указанный тип должен иметь конструктор без параметров.
Ограничение типа значения	: структура	Указанный тип должен быть типом значения .NET.
Ограничение ссылочного типа	: не структура	Указанный тип должен быть ссылочным типом .NET.

ОГРАНИЧЕНИЕ	СИНТАКСИС	ОПИСАНИЕ
Ограничение типа перечисления	: перечисление<*базовым типом*>	Указанный тип должен быть перечислимым типом с указанным базовым типом. не предназначено для общего использования.
Ограничение делегата	: делегат<*кортеж — тип параметра*, *возвращаемый тип*>	Указанный тип должен быть типом делегата с указанными аргументами и возвращаемым значением. не предназначено для общего использования.
Ограничение сравнения	: сравнение	Указанный тип должен поддерживать сравнение.
Ограничение на равенство	: равенство	Указанный тип должен поддерживать равенство.
Неуправляемое ограничение	: неуправляемый	Указанный тип должен иметь неуправляемый тип. Неуправляемые типы являются определенными примитивными типами (`sbyte` , `byte` , `char` , `nativeint` , `unativeint` , `float32` , `float` , `int16` , `uint16` , `int32` , `uint32` , `int64` , `uint64` , или `decimal`), типы перечислений, `nativeptr<_>` или неуниверсальная структура, поля которой являются неуправляемыми типами.

Необходимо добавить ограничение, если в коде должна использоваться функция, доступная в типе ограничения, но не для типов в целом. Например, если для указания типа класса используется ограничение типа, можно использовать любой из методов этого класса в универсальной функции или типе.

Указание ограничений иногда требуется при явном написании параметров типа, поскольку без ограничения компилятор не может проверить, что используемые функции будут доступны для любого типа, который может быть предоставлен во время выполнения для типа. параметр.

Наиболее распространенные ограничения, используемые в F# коде, — это ограничения типов, задающих базовые классы или интерфейсы. Другие ограничения используются F# библиотекой для реализации определенных функций, таких как ограничение явного члена, которое используется для реализации перегрузки операторов для арифметических операторов или предоставляется в основном из-за того, что F# поддерживает полный набор ограничений, поддерживаемых средой CLR.

Во время процесса определения типа некоторые ограничения выводятся автоматически компилятором. Например, при использовании оператора `+` в функции компилятор выводит явное ограничение на типы переменных, которые используются в выражении.

В следующем коде показаны некоторые объявления ограничений:

```
// Base Type Constraint
type Class1<'T when 'T :> System.Exception> =
class end

// Interface Type Constraint
type Class2<'T when 'T :> System.IComparable> =
class end

// Null constraint
type Class3<'T when 'T : null> =
class end

// Member constraint with instance member
type Class5<'T when 'T : (member Method1 : 'T -> int)> =
class end

// Member constraint with property
type Class6<'T when 'T : (member Property1 : int)> =
class end

// Constructor constraint
type Class7<'T when 'T : (new : unit -> 'T)>() =
member val Field = new 'T()

// Reference type constraint
type Class8<'T when 'T : not struct> =
class end

// Enumeration constraint with underlying value specified
type Class9<'T when 'T : enum<uint32>> =
class end

// 'T must implement IComparable, or be an array type with comparable
// elements, or be System.IntPtr or System.UIntPtr. Also, 'T must not have
// the NoComparison attribute.
type Class10<'T when 'T : comparison> =
class end

// 'T must support equality. This is true for any type that does not
// have the NoEquality attribute.
type Class11<'T when 'T : equality> =
class end

type Class12<'T when 'T : delegate<obj * System.EventArgs, unit>> =
class end

type Class13<'T when 'T : unmanaged> =
class end

// Member constraints with two type parameters
// Most often used with static type parameters in inline functions
let inline add(value1 : ^T when ^T : (static member (+) : ^T * ^T -> ^T), value2: ^T) =
value1 + value2

// ^T and ^U must support operator +
let inline heterogenousAdd(value1 : ^T when (^T or ^U) : (static member (+) : ^T * ^U -> ^T), value2 : ^U)
=
value1 + value2

// If there are multiple constraints, use the and keyword to separate them.
type Class14<'T,'U when 'T : equality and 'U : equality> =
class end
```

См. также

- Универсальные шаблоны
- Ограничения

Статически разрешаемые параметры типов

04.11.2019 • 5 minutes to read • Edit Online

Статически разрешаемый параметр типа — это параметр типа, который заменяется фактическим типом во время компиляции, а не во время выполнения. Им предшествует символ каретки (^).

Синтаксис

```
^type-parameter
```

Заметки

На F# языке существует два различных вида параметров типа. Первый тип является стандартным параметром универсального типа. Они обозначены апострофом ('), как в `'T` и `'U`. Они эквивалентны параметрам универсального типа в других .NET Framework языках. Другой тип является статически разрешаемым и обозначается символом курсора, как в `^T` и `^U`.

Статически разрешаемые параметры типа в первую очередь полезны в сочетании с ограничениями элементов, которые являются ограничениями, позволяющими указать, что аргумент типа должен иметь определенный элемент или элементы для использования. Невозможно создать этот тип ограничения с помощью обычного параметра универсального типа.

В следующей таблице перечислены сходства и различия между двумя типами параметров типа.

ВОЗМОЖНОСТЬ	УНИВЕРСАЛЬНЫЙ	СТАТИЧЕСКОЕ РАЗРЕШЕНИЕ
Синтаксис	`'T`, `'U`	`^T`, `^U`
Время разрешения	Выполнение	Время компиляции
Ограничения элементов	Не может использоваться с ограничениями элементов.	Может использоваться с ограничениями элементов.
Создание кода	Тип (или метод) со стандартными параметрами универсального типа приводит к созданию одного универсального типа или метода.	Создается несколько экземпляров типов и методов, по одному для каждого требуемого типа.
Использование с типами	Может использоваться для типов.	Не может использоваться для типов.
Использование со встроенными функциями	Номер Встроенная функция не может быть параметризована со стандартным параметром универсального типа.	Да. Статически разрешаемые параметры типа нельзя использовать в функциях и методах, которые не являются встроенными.

Многие F# функции основной библиотеки, в частности, операторы, имеют статически разрешаемые параметры типа. Эти функции и операторы являются встроенными и приводят к эффективному созданию кода для числовых вычислений.

Встроенные методы и функции, использующие операторы, или другие функции, которые имеют статически

разрешаемые параметры типа, также могут использовать статически разрешаемые параметры типа. Как правило, вывод типа выводит такие встроенные функции в статически разрешаемые параметры типа. В следующем примере показано определение оператора, для которого определен статически разрешаемый параметр типа.

```
let inline (+@) x y = x + x * y
// Call that uses int.
printfn "%d" (1 +@ 1)
// Call that uses float.
printfn "%f" (1.0 +@ 0.5)
```

Разрешенный тип `(+@)` основан на использовании как `(+)`, так и `(*)`, оба из которых вызывают определение типа для определения ограничений членов в статически разрешаемых параметрах типа. Разрешенный тип, как показано в F# интерпретаторе, выглядит следующим образом.

```
^a -> ^c -> ^d
when (^a or ^b) : (static member ( + ) : ^a * ^b -> ^d) and
(^a or ^c) : (static member ( * ) : ^a * ^c -> ^b)
```

Выходные данные выглядят следующим образом.

```
2
1.500000
```

Начиная с F# 4,1, можно также указать конкретные имена типов в статически разрешаемых сигнатурах параметров типа. В предыдущих версиях языка имя типа может быть выделено компилятором, но фактически не может быть указано в сигнатуре. Начиная с F# 4,1 можно также указать конкретные имена типов в статически разрешаемых сигнатурах параметров типа. Ниже приведен пример:

```
let inline konst x _ = x

type CFunctor() =
    static member inline fmap (f: ^a -> ^b, a: ^a list) = List.map f a
    static member inline fmap (f: ^a -> ^b, a: ^a option) =
        match a with
        | None -> None
        | Some x -> Some (f x)

    // default implementation of replace
    static member inline replace< ^a, ^b, ^c, ^d, ^e when ^a :> CFunctor and (^a or ^d): (static member fmap:
(^b -> ^c) * ^d -> ^e) > (a, f) =
        ((^a or ^d) : (static member fmap : (^b -> ^c) * ^d -> ^e) (konst a, f))

    // call overridden replace if present
    static member inline replace< ^a, ^b, ^c when ^b: (static member replace: ^a * ^b -> ^c)>(a: ^a, f: ^b) =
        (^b : (static member replace: ^a * ^b -> ^c) (a, f))

let inline replace_instance< ^a, ^b, ^c, ^d when (^a or ^c): (static member replace: ^b * ^c -> ^d)> (a: ^b,
f: ^c) =
        ((^a or ^c): (static member replace: ^b * ^c -> ^d) (a, f))

// Note the concrete type 'CFunctor' specified in the signature
let inline replace (a: ^a) (f: ^b): ^a0 when (CFunctor or  ^b): (static member replace: ^a *  ^b ->  ^a0) =
    replace_instance<CFunctor, _, _, _> (a, f)
```

См. также

- Универсальные шаблоны
- Вывод типа
- Автоматическое обобщение
- Ограничения
- Встраиваемые функции

Записи

23.10.2019 • 10 minutes to read • Edit Online

Записи представляют собой простые агрегаты именованных значений, которые могут иметь элементы. Они могут быть либо структурами, либо ссылочными типами. По умолчанию они являются ссылочными типами.

Синтаксис

```
[ attributes ]
type [accessibility-modifier] typename =
    { [ mutable ] label1 : type1;
      [ mutable ] label2 : type2;
      ... }
    [ member-list ]
```

Примечания

В предыдущем синтаксисе *TypeName* — это имя типа записи, *Label1* и *ярлык2* — имена значений, называемые *метками*, а *Type1* и *mun2* — типы этих значений. *member-list* — это необязательный список элементов для типа. `[<Struct>]` Атрибут можно использовать для создания записи структуры, а не для записи, которая является ссылочным типом.

Ниже приведены некоторые примеры.

```
// Labels are separated by semicolons when defined on the same line.
type Point = { X: float; Y: float; Z: float; }

// You can define labels on their own line with a semicolon.
type Customer =
    { First: string
      Last: string
      SSN: uint32
      AccountNumber: uint32; }

// A struct record.
[<Struct>]
type StructPoint =
    { X: float
      Y: float
      Z: float }
```

Если каждая метка находится в отдельной строке, точка с запятой является необязательной.

Значения можно задавать в выражениях, называемых *выражениями записи*. Компилятор выводит тип из используемых меток (если метки достаточно отличаются от других типов записей). Фигурные скобки ({}) заключают выражение записи. В следующем коде показано выражение записи, которое инициализирует запись с тремя элементами с плавающей запятой `x` с `y` метками и `z`.

```
let mypoint = { X = 1.0; Y = 1.0; Z = -1.0; }
```

Не используйте сокращенную форму, если возможно наличие другого типа, который также имеет

одинаковые метки.

```
type Point = { X: float; Y: float; Z: float; }
type Point3D = { X: float; Y: float; Z: float }
// Ambiguity: Point or Point3D?
let mypoint3D = { X = 1.0; Y = 1.0; Z = 0.0; }
```

Метки последнего объявленного типа имеют приоритет над элементами ранее объявленного типа, поэтому в предыдущем примере `mypoint3D` выводится `Point3D` как. Можно явно указать тип записи, как показано в следующем коде.

```
let myPoint1 = { Point.X = 1.0; Y = 1.0; Z = 0.0; }
```

Методы могут быть определены для типов записей так же, как для типов классов.

Создание записей с помощью выражений записи

Можно инициализировать записи с помощью меток, определенных в записи. Выражение, которое делает это, называется *выражением записи*. Используйте фигурные скобки, чтобы заключить выражение записи и использовать точку с запятой в качестве разделителя.

В следующем примере показано, как создать запись.

```
type MyRecord =
    { X: int
      Y: int
      Z: int }

let myRecord1 = { X = 1; Y = 2; Z = 3; }
```

Точки с запятой после последнего поля в выражении записи и в определении типа являются необязательными независимо от того, все ли поля находятся в одной строке.

При создании записи необходимо указать значения для каждого поля. Нельзя ссылаться на значения других полей в выражении инициализации для любого поля.

В следующем коде тип `myRecord2` выводится из имен полей. При необходимости можно явно указать имя типа.

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

Другая форма создания записи может быть полезной, если необходимо скопировать существующую запись и, возможно, изменить некоторые значения полей. Это показано в следующей строке кода.

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

Эта форма выражения записи называется *выражением копирования и обновления записи*.

По умолчанию записи являются неизменяемыми; Однако вы можете легко создавать измененные записи с помощью выражения копирования и обновления. Можно также явно указать изменяемое поле.

```
type Car =
    { Make : string
      Model : string
      mutable Odometer : int }

let myCar = { Make = "Fabrikam"; Model = "Coupe"; Odometer = 108112 }
myCar.Odometer <- myCar.Odometer + 21
```

Не используйте атрибут DefaultValue с полями записей. Лучшим подходом является определение экземпляров по умолчанию для записей с полями, которые инициализируются значениями по умолчанию, а затем использование выражения копирования и обновления записей для установки любых полей, которые отличаются от значений по умолчанию.

```
// Rather than use [<DefaultValue>], define a default record.
type MyRecord =
    { Field1 : int
      Field2 : int }

let defaultRecord1 = { Field1 = 0; Field2 = 0 }
let defaultRecord2 = { Field1 = 1; Field2 = 25 }

// Use the with keyword to populate only a few chosen fields
// and leave the rest with default values.
let rr3 = { defaultRecord1 with Field2 = 42 }
```

Создание взаимно рекурсивных записей

Иногда при создании записи может потребоваться, чтобы она зависела от другого типа, который вы хотите определить позже. Это ошибка компиляции, если не определить типы записей для взаимной рекурсивной рекурсии.

Определение взаимно рекурсивных записей выполняется с помощью `and` ключевого слова. Это позволяет связать 2 или более типов записей вместе.

Например, следующий код определяет `Person` тип и `Address` как взаимно рекурсивный:

```
// Create a Person type and use the Address type that is not defined
type Person =
  { Name: string
    Age: int
    Address: Address }
// Define the Address type which is used in the Person record
and Address =
  { Line1: string
    Line2: string
    PostCode: string
    Occupant: Person }
```

Если бы вы определили предыдущий пример без `and` ключевого слова, он не будет компилироваться. Для взаимно рекурсивных определений требуется ключевоеслово. `and`

Сопоставление шаблонов с записями

Записи можно использовать с сопоставлением шаблонов. Можно явно указать некоторые поля и предоставить переменные для других полей, которые будут назначаться при совпадении. Это показано в следующем примере кода.

```
type Point3D = { X: float; Y: float; Z: float }
let evaluatePoint (point: Point3D) =
    match point with
    | { X = 0.0; Y = 0.0; Z = 0.0 } -> printfn "Point is at the origin."
    | { X = xVal; Y = 0.0; Z = 0.0 } -> printfn "Point is on the x-axis. Value is %f." xVal
    | { X = 0.0; Y = yVal; Z = 0.0 } -> printfn "Point is on the y-axis. Value is %f." yVal
    | { X = 0.0; Y = 0.0; Z = zVal } -> printfn "Point is on the z-axis. Value is %f." zVal
    | { X = xVal; Y = yVal; Z = zVal } -> printfn "Point is at (%f, %f, %f)." xVal yVal zVal

evaluatePoint { X = 0.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 100.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 10.0; Y = 0.0; Z = -1.0 }
```

Выходные данные этого кода выглядят следующим образом.

```
Point is at the origin.
Point is on the x-axis. Value is 100.000000.
Point is at (10.000000, 0.000000, -1.000000).
```

Различия между записями и классами

Поля записей отличаются от классов тем, что они автоматически предоставляются как свойства и используются при создании и копировании записей. Построение записей также отличается от создания класса. В типе записи нельзя определить конструктор. Вместо этого применяется синтаксис создания, описанный в этом разделе. Классы не имеют прямой связи между параметрами конструктора, полями и свойствами.

Как и типы Union и Structure, записи имеют семантику структурного равенства. Классы имеют семантику равенства ссылок. Это действие представлено в следующем примере кода:

```
type RecordTest = { X: int; Y: int }

let record1 = { X = 1; Y = 2 }
let record2 = { X = 1; Y = 2 }

if (record1 = record2) then
    printfn "The records are equal."
else
    printfn "The records are unequal."
```

Результат выполнения этого кода выглядит следующим образом:

```
The records are equal.
```

Если написать тот же код с классами, то два объекта класса будут неравными, так как два значения будут представлять два объекта в куче и будут сравниваться только адреса (если только тип класса не переопределит `System.Object.Equals` метод).

Если для записей требуется равенство ссылок, добавьте атрибут `[<ReferenceEquality>]` над записью.

См. также

- Типы языка F#
- Классы
- Справочник по языку F#

- Равенство ссылок
- Соответствие шаблону

Анонимные записи

01.12.2019 • 11 minutes to read • Edit Online

Анонимные записи — это простые статистические выражения именованных значений, которые не нужно объявлять перед использованием. Их можно объявить как структуры или ссылочные типы. По умолчанию они являются ссылочными типами.

Синтаксис

В следующих примерах демонстрируется синтаксис анонимных записей. Элементы, разделенные `[item]`, являются необязательными.

```
// Construct an anonymous record
let value-name = [struct] {| Label1: Type1; Label2: Type2; ...|}

// Use an anonymous record as a type parameter
let value-name = Type-Name<[struct] {| Label1: Type1; Label2: Type2; ...|}>

// Define a parameter with an anonymous record as input
let function-name (arg-name: [struct] {| Label1: Type1; Label2: Type2; ...|}) ...
```

Основное использование

Анонимные записи лучше рассматривать как F# типы записей, которые не нужно объявлять перед созданием экземпляра.

Например, вот как можно взаимодействовать с функцией, которая создает анонимную запись:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

В следующем примере предыдущий объект разворачивается с помощью функции `printCircleStats`, которая принимает анонимную запись в качестве входных данных:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let printCircleStats r (stats: {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats
```

Вызов `printCircleStats` с любым типом анонимных записей, который не имеет такой же "формы", как и тип входных данных, не может компилироваться:

```
printCircleStats r {| Diameter = 2.0; Area = 4.0; MyCircumference = 12.566371 |}
// Two anonymous record types have mismatched sets of field names
// '["Area"; "Circumference"; "Diameter"]' and '["Area"; "Diameter"; "MyCircumference"]'
```

Анонимные записи структуры

Анонимные записи также можно определить как структуру с необязательным `struct` ключевым словом. В следующем примере предыдущий объект расширяется путем создания и использования анонимной записи структуры:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    // Note that the keyword comes before the '{| |}' brace pair
    struct {| Area = a; Circumference = c; Diameter = d |}

// the 'struct' keyword also comes before the '{| |}' brace pair when declaring the parameter type
let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats
```

Определение структуры

Анонимные записи структуры также позволяют «выведение структуры», где не нужно указывать ключевое слово `struct` в месте вызова. В этом примере вы елиде ключевое слово `struct` при вызове `printCircleStats`:

```
let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

printCircleStats r {| Area = 4.0; Circumference = 12.6; Diameter = 12.6 |}
```

Обратный шаблон — указывает, `struct` если входной тип не является структурой анонимной записи, компиляция не будет выполнена.

Внедрение анонимных записей в другие типы

Полезно объявлять Размеченные объединения , варианты которых являются записями. Но если данные в записях имеют тот же тип, что и размеченное объединение, необходимо определить все типы как взаимные рекурсивные. Использование анонимных записей позволяет избежать этого ограничения. Ниже приведен пример типа и функции, которые соответствуют шаблону:

```
type FullName = { FirstName: string; LastName: string }

// Note that using a named for Manager and Executive would require mutually recursive definitions.
type Employee =
    | Engineer of FullName
    | Manager of {| Name: FullName; Reports: Employee list |}
    | Executive of {| Name: FullName; Reports: Employee list; Assistant: Employee |}

let getFirstName e =
    match e with
    | Engineer fullName -> fullName.FirstName
    | Manager m -> m.Name.FirstName
    | Executive ex -> ex.Name.FirstName
```

Копирование и обновление выражений

Анонимные записи поддерживают конструкцию с выражениями копирования и обновления. Например, вот как можно создать новый экземпляр анонимной записи, который копирует существующие данные.

```
let data = {| X = 1; Y = 2 |}
let data' = {| data with Y = 3 |}
```

Однако в отличие от именованных записей анонимные записи позволяют создавать совершенно разные формы с выражениями копирования и обновления. Следующий пример принимает ту же анонимную запись из предыдущего примера и разворачивает ее в новую анонимную запись:

```
let data = {| X = 1; Y = 2 |}
let expandedData = {| data with Z = 3 |} // Gives {| X=1; Y=2; Z=3 |}
```

Кроме того, можно создавать анонимные записи из экземпляров именованных записей.

```
type R = { X: int }
let data = { X = 1 }
let data' = {| data with Y = 2 |} // Gives {| X=1; Y=2 |}
```

Также можно скопировать данные в анонимные записи ссылок и структуры и обратно:

```
// Copy data from a reference record into a struct anonymous record
type R1 = { X: int }
let r1 = { X = 1 }

let data1 = struct {| r1 with Y = 1 |}

// Copy data from a struct record into a reference anonymous record
[<Struct>]
type R2 = { X: int }
let r2 = { X = 1 }

let data2 = {| r1 with Y = 1 |}

// Copy the reference anonymous record data into a struct anonymous record
let data3 = struct {| data2 with Z = r2.X |}
```

Свойства анонимных записей

Анонимные записи имеют ряд характеристик, необходимых для полного понимания того, как их можно использовать.

Анонимные записи являются номинальными

Анонимные записи — это Номинальные типы. Они лучше рассматривать как именованные типы записей (которые также являются номинальными), для которых не требуется объявление переднего плана.

Рассмотрим следующий пример с двумя анонимными объявлениями записей:

```
let x = {| X = 1 |}
let y = {| Y = 1 |}
```

Значения `x` и `y` имеют разные типы и несовместимы друг с другом. Они не являются сопоставимыми и не являются сравнимыми. Чтобы проиллюстрировать это, рассмотрим эквивалент именованной записи:

```
type X = { X: int }
type Y = { Y: int }

let x = { X = 1 }
let y = { Y = 1 }
```

При сравнении с их эквивалентами типов анонимные записи по-разному не различаются.

Анонимные записи используют структурное равенство и сравнение

Как и типы записей, анонимные записи имеют структурную равенство и сравнимы. Это справедливо только в том случае, если все составные типы поддерживают равенство и сравнение, как и типы записей. Для поддержки равенства или сравнения две анонимные записи должны иметь одинаковую форму.

```
{| a = 1+1 |} = {| a = 2 |} // true
{| a = 1+1 |} > {| a = 1 |} // true

// error FS0001: Two anonymous record types have mismatched sets of field names '["a"]' and '["a"; "b"]'
{| a = 1 + 1 |} = {| a = 2;  b = 1|}
```

Анонимные записи являются сериализуемыми

Можно выполнять сериализацию анонимных записей так же, как и с именованными записями. Ниже приведен пример использования Newtonsoft. JSON:

```
open Newtonsoft.Json

let phillip = {| name="Phillip"; age=28 |}
JsonConvert.SerializeObject(phillip)

let phillip = JsonConvert.DeserializeObject<{|name: string; age: int|}>(str)
printfn "Name: %s Age: %d" phillip.name phillip.age
```

Анонимные записи полезны для отправки облегченных данных по сети без необходимости определения домена для сериализованных и десериализованных типов.

Анонимные записи взаимодействуют с C# анонимными типами

Можно использовать API-интерфейс .NET, который требует использования C# анонимных типов. C#Анонимные типы являются тривиальными для взаимодействия с с помощью анонимных записей. В следующем примере показано, как использовать анонимные записи для вызова перегрузки LINQ , для которой требуется анонимный тип:

```
open System.Linq

let names = [ "Ana"; "Felipe"; "Emilia"]
let nameGrouping = names.Select(fun n -> {| Name = n; FirstLetter = n.[0] |})
for ng in nameGrouping do
    printfn "%s has first letter %c" ng.Name ng.FirstLetter
```

Существует множество других API-интерфейсов, используемых в .NET, требующих передачи анонимного типа. Анонимные записи — это средство для работы с ними.

Ограничения

Анонимные записи имеют некоторые ограничения в их использовании. Некоторые из них относятся к их проектированию, но другие податлива.

Ограничения, связанные с сопоставлением шаблонов

Анонимные записи не поддерживают сопоставление шаблонов, в отличие от именованных записей. Существует три причины.

1. Шаблон должен учитывать каждое поле анонимной записи, в отличие от именованных типов записей. Это обусловлено тем, что анонимные записи не поддерживают структурную подтипизацию — они являются номинальными типами.
2. Из-за (1) нет возможности иметь дополнительные шаблоны в выражении соответствия шаблону, так как каждый отдельный шаблон подразумевает другой анонимный тип записи.
3. Из-за (3) любой анонимный шаблон записи будет более подробным, чем использование нотации "точка".

Существует предложение Open Language, позволяющее сопоставлять шаблоны в ограниченных контекстах.

Ограничения с изменяемостью

В настоящее время невозможно определить анонимную запись с `mutable` данными. Существует предложение Open Language , разрешающее изменяющиеся данные.

Ограничения для анонимных записей структуры

Невозможно объявить анонимные записи структуры как `IsByRefLike` или `IsReadOnly`. Существует предложение Open Language для `IsByRefLike` и `IsReadOnly` анонимных записей.

Копирование и обновление выражений записей

23.10.2019 • 2 minutes to read • Edit Online

Выражение записи копирования и обновления — это выражение, которое копирует существующую запись, обновляет указанные поля и возвращает обновленную запись.

Синтаксис

```
{ record-name with
    updated-labels }

{| anonymous-record-name with
    updated-labels |}
```

Примечания

Записи и анонимные записи являются неизменяемыми по умолчанию, поэтому обновление существующей записи невозможно. Чтобы создать обновленную запись, необходимо снова указать все поля записи. Чтобы упростить эту задачу, можно использовать *выражение копирования и обновления* . Это выражение принимает существующую запись, создает новый объект того же типа, используя указанные поля из выражения и отсутствующее поле, указанное выражением.

Это может быть полезно, если необходимо скопировать существующую запись и, возможно, изменить некоторые значения полей.

Возьмем для экземпляра созданную запись.

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

Если необходимо обновить только поле в этой записи, можно использовать *выражение копирования и обновления записей* следующим образом:

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

См. также

- Записи
- Анонимные записи
- Справочник по языку F#

Размеченные объединения

23.10.2019 • 14 minutes to read • Edit Online

Размеченные объединения обеспечивают поддержку значений, которые могут быть одного из нескольких именованных вариантов, возможно, с разными значениями и типами. Размеченные объединения полезны для разнородных данных. данные, которые могут иметь особые случаи, включая допустимые и ошибочные варианты; данные, которые могут различаться в типе от одного экземпляра к другому; а также в качестве альтернативы для небольших иерархий объектов. Кроме того, рекурсивные размеченные объединения используются для представления древовидных структур данных.

Синтаксис

```
[ attributes ]
type [accessibility-modifier] type-name =
    | case-identifier1 [of [ fieldname1 : ] type1 [ * [ fieldname2 : ] type2 ...]
    | case-identifier2 [of [fieldname3 : ]type3 [ * [ fieldname4 : ]type4 ...]

    [ member-list ]
```

Примечания

Размеченные объединения похожи на типы объединения на других языках, но существуют различия. Как и в случае с типом C++ объединения в или типа variant в Visual Basic, данные, хранящиеся в значении, не являются фиксированными; Это может быть один из нескольких различных параметров. Однако в отличие от объединений в этих других языках, каждому из возможных параметров присваивается *идентификатор варианта*. Идентификаторы вариантов — это имена различных возможных типов значений, которые могут быть объектами этого типа. значения являются необязательными. Если значения не указаны, регистр эквивалентен регистру перечисления. При наличии значений каждое значение может быть одним значением указанного типа или кортежем, который объединяет несколько полей одного или разных типов. Можно присвоить имя отдельному полю, но имя является необязательным, даже если другие поля в том же регистре называются.

По умолчанию `public` специальные возможности для размеченных объединений имеют значение.

Например, рассмотрим следующее объявление типа фигуры.

```
type Shape =
    | Rectangle of width : float * length : float
    | Circle of radius : float
    | Prism of width : float * float * height : float
```

Приведенный выше код объявляет фигуру размеченного объединения, которая может иметь значения любого из трех вариантов: Прямоугольник, круг и Prism. Каждый вариант имеет другой набор полей. В прямоугольнике есть два именованных поля типа `float`, которые имеют имена Width и length. В круглом варианте есть только одно именованное поле, RADIUS. В Prism Case есть три поля, два из которых (ширина и высота) называются полями. Безымянные поля называются анонимными полями.

Объекты создаются путем предоставления значений для именованных и анонимных полей в соответствии со следующими примерами.

```
let rect = Rectangle(length = 1.3, width = 10.0)
let circ = Circle (1.0)
let prism = Prism(5., 2.0, height = 3.0)
```

Этот код показывает, что можно либо использовать именованные поля в инициализации, либо полагаться на порядок полей в объявлении и просто предоставить значения для каждого поля в свою очередь. При вызове конструктора `rect` для в предыдущем коде используются именованные поля, но при вызове конструктора `circ` для используется упорядочение. Можно смешивать упорядоченные поля и именованные поля, как в конструкции `prism`.

Тип является простым размеченное объединением в F# основной библиотеке. `option` `option` Тип объявляется следующим образом.

```
// The option type is a discriminated union.
type Option<'a> =
    | Some of 'a
    | None
```

Предыдущий код указывает, что тип `Option` — это размеченное объединение, которое имеет два варианта: `Some` и `None`. Вариант имеет связанное значение, состоящее из одного анонимного поля, тип которого представлен параметром `'a` типа. `Some` `None` Регистр не имеет связанного значения. Таким же `option` типом определяется универсальный тип, который либо имеет значение некоторого типа, либо не имеет значения. Тип `Option` также имеет псевдоним типа "нижний регистр" `option`, который чаще всего используется.

Идентификаторы вариантов можно использовать в качестве конструкторов для типа размеченного объединения. Например, следующий код используется для создания значений `option` типа.

```
let myOption1 = Some(10.0)
let myOption2 = Some("string")
let myOption3 = None
```

Идентификаторы вариантов также используются в выражениях сопоставления шаблонов. В выражении сопоставления шаблонов идентификаторы предоставляются для значений, связанных с отдельными вариантами. Например, в следующем коде `x` является идентификатором, заданным значением, связанным `Some` с вариантом `option` типа.

```
let printValue opt =
    match opt with
    | Some x -> printfn "%A" x
    | None -> printfn "No value."
```

В выражениях сопоставления шаблонов можно использовать именованные поля для указания совпадений размеченного объединения. Для типа фигуры, объявленного ранее, можно использовать именованные поля, как показано в следующем коде, чтобы извлечь значения полей.

```
let getShapeHeight shape =
    match shape with
    | Rectangle(height = h) -> h
    | Circle(radius = r) -> 2. * r
    | Prism(height = h) -> h
```

Как правило, идентификаторы вариантов можно использовать без уточнения их именем объединения.

Если необходимо, чтобы имя всегда было дополнено именем объединения, можно применить атрибут рекуирекуалифиедакцесс к определению типа объединения.

Разтекание размеченных объединений

В F# размеченных объединениях часто используются в моделировании доменов для упаковки одного типа. Также можно легко извлечь базовое значение с помощью сопоставления шаблонов. Не нужно использовать выражение соответствия для одного варианта:

```
let ([UnionCaseIdentifier] [values]) = [UnionValue]
```

Следующий пример демонстрирует это:

```
type ShaderProgram = | ShaderProgram of id:int

let someFunctionUsingShaderProgram shaderProgram =
    let (ShaderProgram id) = shaderProgram
    // Use the unwrapped value
    ...
```

Сопоставление шаблонов также разрешено непосредственно в параметрах функции, поэтому можно разворачивать один случай:

```
let someFunctionUsingShaderProgram (ShaderProgram id) =
    // Use the unwrapped value
    ...
```

Размеченные объединения структуры

Можно также представить размеченные объединения как структуры. Это делается с помощью `[<Struct>]` атрибута.

```
[<Struct>]
type SingleCase = Case of string

[<Struct>]
type Multicase =
    | Case1 of Case1 : string
    | Case2 of Case2 : int
    | Case3 of Case3 : double
```

Так как это типы значений, а не ссылочные типы, существуют дополнительные рекомендации по сравнению со ссылочными объединениями.

1. Они копируются как типы значений и имеют семантику типов значений.
2. Нельзя использовать определение рекурсивного типа с многовариантным размеченное объединением.
3. Необходимо указать уникальные имена вариантов для размеченного объединения с многовариантными структурами.

Использование размеченных объединений вместо иерархий объектов

Вы часто можете использовать размеченное объединение в качестве более простой альтернативы небольшой иерархии объектов. Например, можно использовать следующее размеченное объединение

вместо `Shape` базового класса, который имеет производные типы для окружности, квадрата и т. д.

```
type Shape =
  // The value here is the radius.
| Circle of float
  // The value here is the side length.
| EquilateralTriangle of double
  // The value here is the side length.
| Square of double
  // The values here are the height and width.
| Rectangle of double * double
```

Вместо виртуального метода для вычисления области или периметра, как при использовании в объектно-ориентированной реализации, можно использовать сопоставление шаблонов для ветвления в соответствующих формулах для вычисления этих количеств. В следующем примере для вычисления области используются различные формулы, в зависимости от фигуры.

```
let pi = 3.141592654

let area myShape =
    match myShape with
    | Circle radius -> pi * radius * radius
    | EquilateralTriangle s -> (sqrt 3.0) / 4.0 * s * s
    | Square s -> s * s
    | Rectangle (h, w) -> h * w

let radius = 15.0
let myCircle = Circle(radius)
printfn "Area of circle that has radius %f: %f" radius (area myCircle)

let squareSide = 10.0
let mySquare = Square(squareSide)
printfn "Area of square that has side %f: %f" squareSide (area mySquare)

let height, width = 5.0, 10.0
let myRectangle = Rectangle(height, width)
printfn "Area of rectangle that has height %f and width %f is %f" height width (area myRectangle)
```

Выходные данные выглядят следующим образом:

```
Area of circle that has radius 15.000000: 706.858347
Area of square that has side 10.000000: 100.000000
Area of rectangle that has height 5.000000 and width 10.000000 is 50.000000
```

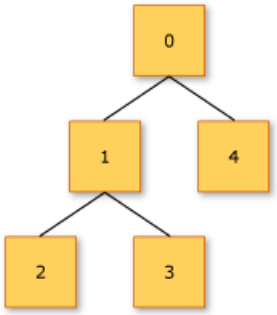
Использование размеченных объединений для структур данных дерева

Размеченные объединения могут быть рекурсивными, что означает, что объединение может включаться в тип одного или нескольких вариантов. Рекурсивные размеченные объединения можно использовать для создания древовидных структур, которые используются для моделирования выражений в языках программирования. В следующем коде рекурсивное размеченное объединение используется для создания структуры данных в двоичном дереве. Объединение состоит из двух вариантов `Node`:, которые представляют собой узел с целочисленным значением, а также левые и правые поддеревья, и `Tip`, который завершает дерево.

```
type Tree =
    | Tip
    | Node of int * Tree * Tree

let rec sumTree tree =
    match tree with
    | Tip -> 0
    | Node(value, left, right) ->
        value + sumTree(left) + sumTree(right)
let myTree = Node(0, Node(1, Node(2, Tip, Tip), Node(3, Tip, Tip)), Node(4, Tip, Tip))
let resultSumTree = sumTree myTree
```

В предыдущем коде `resultSumTree` имеет значение 10. На следующем рисунке показана древовидная структура `myTree` для.



Размеченные объединения хорошо работают, если узлы в дереве являются разнородными. В следующем коде тип `Expression` представляет дерево абстрактного синтаксиса выражения в простом языке программирования, который поддерживает сложение и умножение чисел и переменных. Некоторые варианты объединения не являются рекурсивными и представляют либо числа (`Number`), либо переменные (`Variable`). Другие варианты являются рекурсивными и представляют операции (`Add` и `Multiply`), где операнды также являются выражениями. `Evaluate` Функция использует выражение match для рекурсивной обработки дерева синтаксиса.

```
type Expression =
    | Number of int
    | Add of Expression * Expression
    | Multiply of Expression * Expression
    | Variable of string

let rec Evaluate (env:Map<string,int>) exp =
    match exp with
    | Number n -> n
    | Add (x, y) -> Evaluate env x + Evaluate env y
    | Multiply (x, y) -> Evaluate env x * Evaluate env y
    | Variable id    -> env.[id]

let environment = Map.ofList [ "a", 1 ;
                               "b", 2 ;
                               "c", 3 ]

// Create an expression tree that represents
// the expression: a + 2 * b.
let expressionTree1 = Add(Variable "a", Multiply(Number 2, Variable "b"))

// Evaluate the expression a + 2 * b, given the
// table of values for the variables.
let result = Evaluate environment expressionTree1
```

При выполнении этого кода значение `result` равно 5.

Участники

Можно определить элементы в размеченных объединениях. В следующем примере показано, как определить свойство и реализовать интерфейс:

```
open System

type IPrintable =
    abstract Print: unit -> unit

type Shape =
    | Circle of float
    | EquilateralTriangle of float
    | Square of float
    | Rectangle of float * float

    member this.Area =
        match this with
        | Circle r -> 2.0 * Math.PI * r
        | EquilateralTriangle s -> s * s * sqrt 3.0 / 4.0
        | Square s -> s * s
        | Rectangle(l, w) -> l * w

    interface IPrintable with
        member this.Print () =
            match this with
            | Circle r -> printfn "Circle with radius %f" r
            | EquilateralTriangle s -> printfn "Equilateral Triangle of side %f" s
            | Square s -> printfn "Square with side %f" s
            | Rectangle(l, w) -> printfn "Rectangle with length %f and width %f" l w
```

Общие атрибуты

В размеченных объединениях обычно встречаются следующие атрибуты:

- `[<RequireQualifiedAccess>]`
- `[<NoEquality>]`
- `[<NoComparison>]`
- `[<Struct>]`

См. также

- Справочник по языку F#

Перечисления

23.10.2019 • 3 minutes to read • Edit Online

Перечисления, также известные как *перечисления*, являются целочисленными типами, где метки присваиваются подмножеству значений. Их можно использовать вместо литералов, чтобы сделать код более понятным и простым в обслуживании.

Синтаксис

```
type enum-name =
| value1 = integer-literal1
| value2 = integer-literal2
...
```

Примечания

Перечисление во многом похоже на размеченное объединение с простыми значениями, за исключением того, что можно указать значения. Значениями обычно являются целые числа, начинающиеся с 0 или 1, или целые числа, представляющие битовые позиции. Если перечисление предназначено для представления битовых позиций, следует также использовать атрибут Flags.

Базовый тип перечисления определяется из используемого литерала, поэтому, например, можно использовать литералы с суффиксом, например `1u`, `2u` и т. д., для типа целого числа без знака (`uint32`).

При ссылке на именованные значения необходимо использовать имя самого типа перечисления в качестве квалификатора, то есть `enum-name.value1`, а не только. `value1` Это поведение отличается от различенных объединений. Это происходит потому, что перечисления всегда имеют атрибут рекуирекуалифиедакцесс .

В следующем коде показано объявление и использование перечисления.

```
// Declaration of an enumeration.
type Color =
   | Red = 0
   | Green = 1
   | Blue = 2
// Use of an enumeration.
let col1 : Color = Color.Red
```

Можно легко преобразовать перечисления в базовый тип с помощью соответствующего оператора, как показано в следующем коде.

```
// Conversion to an integral type.
let n = int col1
```

`sbyte` Перечисляемые типы могут иметь один из следующих базовых типов:, `uint64` `int64` `int16` `uint32` `uint16` `byte`,, `int32`,,,, `uint16`, и `char`. Типы перечисления представлены в .NET Framework как типы, унаследованные от `System.Enum`, которые, в свою очередь, наследуются от. `System.ValueType` Таким образом, они являются типами значений, расположенными в стеке или встроенными в содержащий их объект, а любое значение базового типа является допустимым значением перечисления. Это важно при сопоставлении шаблонов со значениями перечисления, поскольку необходимо предоставить шаблон,

который перехватывает неименованные значения.

Функцию в F# библиотеке можно использовать для создания значения перечисления, даже для значения, отличного от одного из стандартных именованных значений. `enum` Используйте `enum` функцию следующим образом.

```
let col2 = enum<Color>(3)
```

Функция по `enum` умолчанию работает с `int32` типом. Поэтому его нельзя использовать с типами перечисления, имеющими другие базовые типы. Вместо этого используйте следующий.

```
type uColor =
   | Red = 0u
   | Green = 1u
   | Blue = 2u
let col3 = Microsoft.FSharp.Core.LanguagePrimitives.EnumOfValue<uint32, uColor>(2u)
```

Кроме того, варианты для перечислений всегда создаются как `public`. Это так, что они согласовываются C# с и остальной частью платформы .NET.

См. также

- Справочник по языку F#
- Приведение и преобразование

Сокращенные обозначения типов

23.10.2019 • 2 minutes to read • Edit Online

Аббревиатура типа — это псевдоним или альтернативное имя для типа.

Синтаксис

```
type [accessibility-modifier] type-abbreviation = type-name
```

Примечания

Аббревиатуры типов можно использовать для присвоения типу более понятного имени, чтобы облегчить чтение кода. Их также можно использовать для создания простого в использовании имени типа, который в противном случае будет громоздким для записи. Кроме того, можно использовать сокращения типов, чтобы упростить изменение базового типа, не изменяя весь код, использующий этот тип. Ниже приведена простая аббревиатура типа.

По умолчанию в качестве специальных возможностей для `public` аббревиатур типа используется.

```
type SizeType = uint32
```

Аббревиатуры типов могут включать в себя универсальные параметры, как показано в следующем коде.

```
type Transform<'a> = 'a -> 'a
```

В предыдущем коде `Transform` — это аббревиатура типа, представляющая функцию, которая принимает один аргумент любого типа и возвращает одно значение этого же типа.

Сокращения типов не сохраняются в .NET Framework коде MSIL. Поэтому при использовании F# сборки на другом языке .NET Framework необходимо использовать базовое имя типа для аббревиатуры типов.

Аббревиатуры типов также можно использовать в единицах измерения. Дополнительные сведения см. в разделе единицы измерения.

См. также

- Справочник по языку F#

Классы

23.10.2019 • 14 minutes to read • Edit Online

Классы — это типы, представляющие объекты, которые могут иметь свойства, методы и события.

Синтаксис

```
// Class definition:
type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as identifier ] =
[ class ]
[ inherit base-type-name(base-constructor-args) ]
[ let-bindings ]
[ do-bindings ]
member-list
...
[ end ]
// Mutually recursive class definitions:
type [access-modifier] type-name1 ...
and [access-modifier] type-name2 ...
...
```

Примечания

Классы представляют фундаментальное описание типов объектов .NET. класс является концепцией основного типа, поддерживающей объектно-ориентированное F#программирование в.

В приведенном выше синтаксисе `type-name` — любой допустимый идентификатор. `type-params` Описывает необязательные параметры универсального типа. Он состоит из имен параметров типа и ограничений, заключенных в угловые `>` скобки (`<` и). Дополнительные сведения см. в разделе универсальные шаблоны и ограничения. `parameter-list` Описывает параметры конструктора. Первый модификатор доступа относится к типу; второй объект относится к основному конструктору. В обоих случаях значение по умолчанию `public` —.

Базовый класс для класса указывается с помощью `inherit` ключевого слова. Для конструктора базового класса необходимо указать аргументы в скобках.

Поля или значения функций, локальные для класса, объявляются с помощью `let` привязок, поэтому необходимо следовать общим правилам для `let` привязок. `do-bindings` Раздел содержит код, выполняемый при создании объекта.

`member-list` Содержит дополнительные конструкторы, объявления экземпляров и статических методов, объявления интерфейсов, абстрактные привязки и объявления свойств и событий. Они описаны в разделе элементы.

Объект `identifier` , используемый с ключевым словом Optional `as` , задает имя переменной экземпляра или собственный идентификатор, который можно использовать в определении типа для ссылки на экземпляр типа. Дополнительные сведения см. в подразделе «собственные идентификаторы» далее в этом разделе.

Ключевые слова `class` и `end` , которые отмечают начало и конец определения, являются необязательными.

Взаимно рекурсивные типы, которые являются типами, ссылающимися друг на друга, объединяются

вместе с ключевым `and` словом так же, как и взаимно рекурсивные функции. Пример см. в разделе взаимно рекурсивные типы.

Конструкторы

Конструктор — это код, создающий экземпляр типа класса. Конструкторы для классов работают несколько иначе, F# чем в других языках .NET. В F# классе всегда существует первичный конструктор, аргументы которого описаны в `parameter-list` разделе, который следует за именем типа, а `let` тело состоит из привязок (и `let rec`) в начале объявления класса и `do` следующие привязки. Аргументы первичного конструктора находятся в области действия во всем объявлении класса.

Можно добавить дополнительные конструкторы, используя `new` ключевое слово для добавления члена следующим образом:

`new` (`argument-list`) = `constructor-body`

Тело нового конструктора должно вызывать первичный конструктор, который указан в верхней части объявления класса.

В следующем примере показана эта концепция. В следующем коде `MyClass` имеет два конструктора — первичный конструктор, который принимает два аргумента и еще один конструктор, не принимающий аргументов.

```
type MyClass1(x: int, y: int) =
   do printfn "%d %d" x y
   new() = MyClass1(0, 0)
```

Привязка let и do

Привязки `let` и `do` в определении класса формируют тело конструктора первичного класса и поэтому выполняются при каждом создании экземпляра класса. `let` Если привязка является функцией, то она компилируется в элемент. `let` Если привязка является значением, которое не используется ни в одной функции или члене, то оно компилируется в переменную, которая является локальной для конструктора. В противном случае он компилируется в поле класса. Приведенные ниже `do` выражения компилируются в основной конструктор и выполняют код инициализации для каждого экземпляра. Поскольку любые дополнительные конструкторы всегда вызывают первичный конструктор, `let` привязки и `do` привязки всегда выполняются независимо от того, какой конструктор вызывается.

К полям, созданным `let` с помощью привязок, можно обращаться через методы и свойства класса, однако к ним нельзя получить доступ из статических методов, даже если статические методы принимают переменную экземпляра в качестве параметра. К ним нельзя получить доступ с помощью собственного идентификатора, если он существует.

Собственные идентификаторы

Собственный идентификатор — это имя, представляющее текущий экземпляр. Собственные идентификаторы `this` похожи на ключевое слово в C++ `Me` C# или в Visual Basic. Собственный идентификатор можно определить двумя разными способами в зависимости от того, должен ли сам идентификатор находиться в области видимости для определения всего класса или только для отдельного метода.

Чтобы определить собственный идентификатор для всего класса, используйте `as` ключевое слово после закрывающих круглых скобок списка параметров конструктора и укажите имя идентификатора.

Чтобы определить собственный идентификатор только для одного метода, укажите собственный

идентификатор в объявлении члена непосредственно перед именем метода и точкой (.) в качестве разделителя.

В следующем примере кода показаны два способа создания собственного идентификатора. В первой строке `as` ключевое слово используется для определения собственного идентификатора. В пятой строке идентификатор `this` используется для определения собственного идентификатора, область действия которого ограничена методом. `PrintMessage`

```
type MyClass2(dataIn) as self =
    let data = dataIn
    do
        self.PrintMessage()
    member this.PrintMessage() =
        printf "Creating MyClass2 with Data %d" data
```

В отличие от других языков .NET, можно присвоить себе собственный идентификатор. Вы не ограничены такими именами, как `self`, `Me` или `this`.

Собственный идентификатор, объявленный с `as` ключевым словом, не инициализируется до тех пор, `let` пока не будут выполнены привязки. Поэтому его нельзя использовать в `let` привязках. Можно использовать собственный идентификатор в `do` разделе привязок.

Параметры универсального типа

Параметры универсального типа задаются в угловых `>` скобках (`<` и) в виде одиночной кавычки, за которой следует идентификатор. Несколько параметров универсального типа разделяются запятыми. Параметр универсального типа находится в области видимости во всем объявлении. В следующем примере кода показано, как задать параметры универсального типа.

```
type MyGenericClass<'a> (x: 'a) =
   do printfn "%A" x
```

Аргументы типа выводятся при использовании типа. В следующем коде выводимый тип является последовательностью кортежей.

```
let g1 = MyGenericClass( seq { for i in 1 .. 10 -> (i, i*i) } )
```

Указание наследования

`inherit` Предложение определяет прямой базовый класс, если таковой имеется. В F#поддерживается только один прямой базовый класс. Интерфейсы, реализуемые классом, не считаются базовыми классами. Интерфейсы обсуждаются в разделе интерфейсы .

Доступ к методам и свойствам базового класса из производного класса можно получить с помощью ключевого слова `base` Language в качестве идентификатора, за которым следует точка (.) и имя члена.

Дополнительные сведения см. в разделе Наследование.

Раздел членов

В этом разделе можно определить статические методы или экземпляры методов, свойства, реализации интерфейса, абстрактные элементы, объявления событий и дополнительные конструкторы. Привязки let и Do не могут присутствовать в этом разделе. Поскольку члены могут добавляться в различные F# типы в дополнение к классам, они обсуждаются в отдельной теме, членах.

Взаимно рекурсивные типы

При определении типов, ссылающихся друг на друга циклически, вы объединяете определения типов с помощью `and` ключевого слова. Ключевое слово `type` заменяет ключевое слово на все, кроме первого определения, следующим образом. `and`

```
open System.IO

type Folder(pathIn: string) =
  let path = pathIn
  let filenameArray : string array = Directory.GetFiles(path)
  member this.FileArray = Array.map (fun elem -> new File(elem, this)) filenameArray

and File(filename: string, containingFolder: Folder) =
   member this.Name = filename
   member this.ContainingFolder = containingFolder

let folder1 = new Folder(".")
for file in folder1.FileArray do
   printfn "%s" file.Name
```

Выходные данные представляют собой список всех файлов в текущем каталоге.

Когда следует использовать классы, объединения, записи и структуры

Учитывая различные типы выборки, необходимо хорошо понимать, что каждый тип предназначен для выбора подходящего типа для конкретной ситуации. Классы предназначены для использования в контекстах объектно-ориентированного программирования. Объектно-ориентированное программирование является главным парадигмой, используемой в приложениях, написанных для .NET Framework. Если F# код должен тесно работать с .NET Framework или другой объектно-ориентированной библиотекой, и особенно в том случае, если необходимо расширить объектно-ориентированную систему типов, например библиотеку пользовательского интерфейса, то классы, вероятно, подходят.

Если вы не тесно взаимодействуете с объектно-ориентированным кодом или пишете код, который является автономным и, следовательно, защищен от частого взаимодействия с объектно-ориентированным кодом, следует рассмотреть возможность использования записей и размеченных объединений. В качестве более простой альтернативы иерархии объектов часто можно использовать одно, хорошо продуманное размеченное объединение, а также соответствующий код сопоставления шаблонов. Дополнительные сведения о размеченных объединениях см. в разделе Размеченные объединения.

Записи имеют преимущество, чем классы, но записи не подходят, если требования типа превышают возможности, которые могут быть выполнены с простотой. Записи по сути являются простыми статистическими значениями, без отдельных конструкторов, которые могут выполнять пользовательские действия без скрытых полей и без реализации наследования и интерфейса. Хотя элементы, такие как свойства и методы, можно добавлять к записям, чтобы сделать их поведение более сложным, поля, хранящиеся в записи, по-прежнему представляют собой простую статистическую функцию значений. Дополнительные сведения о записях см. в разделе записи.

Структуры также полезны для небольших статистических данных, но они отличаются от классов и записей тем, что они являются типами значений .NET. Классы и записи являются ссылочными типами .NET. Семантика типов значений и ссылочных типов различается в том, что типы значений передаются по значению. Это означает, что они копируются бит для бита, когда они передаются в качестве параметра или возвращаются из функции. Они также хранятся в стеке или, если они используются в качестве поля, внедрены в родительский объект, а не хранятся в отдельном месте в куче. Таким образом,

структуры подходят для часто используемых данных, когда издержки доступа к куче являются проблемой. Дополнительные сведения о структурах см. в разделе структуры.

См. также

- Справочник по языку F#
- Члены
- Наследование
- Интерфейсы

Структуры

23.10.2019 • 7 minutes to read • Edit Online

Структура — это тип компактного объекта, который может быть более эффективным, чем класс для типов, имеющих небольшой объем данных и простое поведение.

Синтаксис

```
[ attributes ]
type [accessibility-modifier] type-name =
    struct
        type-definition-elements-and-members
    end
// or
[ attributes ]
[<StructAttribute>]
type [accessibility-modifier] type-name =
    type-definition-elements-and-members
```

Примечания

Структуры являются *типами значений*. Это означает, что они хранятся непосредственно в стеке или, когда они используются в качестве полей или элементов массива, встроены в родительский тип. В отличие от классов и записей структуры имеют семантику pass-by-value (передача по значению). Это означает, что они используются в основном для небольших объемов данных, которые часто используются и копируются.

В предыдущем синтаксисе показаны две формы. Первая не является упрощенным синтаксисом, но, тем не менее, используется довольно часто, так как при использовании ключевых слов `struct` и `end` можно опустить атрибут `StructAttribute`, который есть во второй форме. `StructAttribute` можно сократить до `Struct`.

В предыдущем синтаксисе *типы-Definition-Elements-and-Members* представляют объявления и определения членов. Структуры могут иметь конструкторы, изменяемые и неизменяемые поля. Они также могут объявлять элементы и реализации интерфейсов. Дополнительные сведения см. в разделе Members.

Структуры не могут участвовать в наследовании, не могут содержать привязки `let` или `do`, не могут рекурсивно содержать поля собственного типа (хотя могут содержать ссылочные ячейки, которые ссылаются на собственный тип).

Так как структуры не допускают использование привязок `let`, в структурах необходимо объявлять поля с помощью ключевого слова `val`. Ключевое слово `val` определяет поле и его тип, но не разрешает выполнять инициализацию. Вместо этого объявления `val` инициализируются с нулем или значением null. Поэтому для структур с неявным конструктором (то есть параметрами, заданными непосредственно после имени структуры в объявлении) требуется, чтобы объявления `val` были помечены атрибутом `DefaultValue`. Структуры, имеющие определенный конструктор, по-прежнему поддерживают инициализацию с нулевым значением. Таким образом, атрибут `DefaultValue` является объявлением того, что для поля допускается нулевое значение. Неявные конструкторы для структур не выполняют каких-либо действий, так как привязки `let` и `do` не разрешены для типа, однако переданные значения параметров неявного конструктора доступны в виде закрытых полей.

Явные конструкторы могут способствовать инициализации значений полей. Структура, которая имеет

явный конструктор, по-прежнему поддерживают инициализацию с нулевым значением. Тем не менее, в объявлениях `DefaultValue` не следует использовать атрибут `val`, поскольку он вступает в конфликт с явным конструктором. Дополнительные сведения об `val` объявлениях см. в разделе явные поля: Ключевоеслово. `val`

В структурах допускается использование атрибутов и модификаторов доступности, для которых действуют те же правила, что и для других типов. Дополнительные сведения см. в разделе атрибуты и Управление доступом.

В следующих примерах кода показаны определения структуры.

```
// In Point3D, three immutable values are defined.
// x, y, and z will be initialized to 0.0.
type Point3D =
    struct
        val x: float
        val y: float
        val z: float
    end

// In Point2D, two immutable values are defined.
// It also has a member which computes a distance between itself and another Point2D.
// Point2D has an explicit constructor.
// You can create zero-initialized instances of Point2D, or you can
// pass in arguments to initialize the values.
type Point2D =
    struct
        val X: float
        val Y: float
        new(x: float, y: float) = { X = x; Y = y }

        member this.GetDistanceFrom(p: Point2D) =
            let dX = (p.X - this.X) ** 2.0
            let dY = (p.Y - this.Y) ** 2.0

            dX + dY
            |> sqrt
    end
```

Структуры Бирефлике

Вы можете определить собственные структуры, которые могут соответствовать `byref` семантике, приведенной ниже. Дополнительные сведения см. в разделе ByRef . Это делается с помощью IsByRefLikeAttribute атрибута:

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` не подразумевается `Struct`. Оба должны присутствовать в типе.

" `byref` -Like" структура в F# — это тип значения, привязанный к стеку. Он никогда не выделяется в управляемой куче. Структура `byref`, похожая на структуру, полезна для высокопроизводительного программирования, так как она применяется с набором строгих проверок на время существования и без записи. Правила:

- Их можно использовать в качестве параметров функций, параметров методов, локальных переменных, возвращаемых методом.
- Они не могут быть статическими или членами экземпляров класса или обычной структуры.
- Они не могут быть захвачены ни одной конструкцией замыкания (`async` методами или лямбда-выражениями).
- Их нельзя использовать в качестве универсального параметра.

Хотя эти правила очень сильно ограничивают использование, они делают это для обеспечения безопасности высокопроизводительных вычислительных систем надежным способом.

Структуры только для чтения

Можно закомментировать структуры с помощью IsReadOnlyAttribute атрибута. Например:

```
[<IsReadOnly; Struct>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsReadOnly` не подразумевается `Struct`. Необходимо добавить оба варианта, чтобы иметь `IsReadOnly` структуру.

Использование этого атрибута создает метаданные, позволяя F# C# ему обрабатываться как `inref<'T>` и `in ref` соответственно.

Определение изменяемого значения внутри структуры, предназначенной только для чтения, приводит к ошибке.

Записи структуры и размеченные объединения

Записи и Размеченные объединения можно представить в виде структур с `[<Struct>]` атрибутом. Дополнительные сведения см. в статье.

См. также

- Справочник по языку F#
- Классы
- Записи
- Члены

Наследование

23.10.2019 • 5 minutes to read • Edit Online

Наследование используется для моделирования связи «является-а» или подтипов в объектно-ориентированном программировании.

Указание отношений наследования

Отношения наследования указываются с помощью `inherit` ключевого слова в объявлении класса. В следующем примере показана базовая синтаксическая форма.

```
type MyDerived(...) =
    inherit MyBase(...)
```

Класс может иметь не более одного прямого базового класса. Если базовый класс не указан с помощью `inherit` ключевого слова, класс неявно наследует от. `System.Object`

Унаследованные члены

Если класс наследуется от другого класса, методы и члены базового класса становятся доступными для пользователей производного класса, как если бы они были прямыми членами производного класса.

Все привязки let и параметры конструктора являются закрытыми для класса, поэтому к ним нельзя получить доступ из производных классов.

Ключевое `base` слово доступно в производных классах и ссылается на экземпляр базового класса. Он используется как идентификатор Self.

Виртуальные методы и переопределения

Виртуальные методы (и свойства) работают несколько иначе F# по сравнению с другими языками .NET. Для объявления нового виртуального члена используется `abstract` ключевое слово. Это делается независимо от того, предоставляется ли реализация по умолчанию для этого метода. Таким образом, полное определение виртуального метода в базовом классе соответствует следующему шаблону:

```
abstract member [method-name] : [type]

default [self-identifier].[method-name] [argument-list] = [method-body]
```

В производном классе переопределение этого виртуального метода выполняется следующим образом:

```
override [self-identifier].[method-name] [argument-list] = [method-body]
```

Если опустить реализацию по умолчанию в базовом классе, базовый класс станет абстрактным классом.

В следующем примере кода показано объявление нового виртуального метода `function1` в базовом классе и его переопределение в производном классе.

```
type MyClassBase1() =
   let mutable z = 0
   abstract member function1 : int -> int
   default u.function1(a : int) = z <- z + a; z

type MyClassDerived1() =
   inherit MyClassBase1()
   override u.function1(a: int) = a + 1
```

Конструкторы и наследование

Конструктор для базового класса должен вызываться в производном классе. Аргументы для конструктора базового класса появятся в списке аргументов в `inherit` предложении. Используемые значения должны быть определены из аргументов, передаваемых конструктору производного класса.

В следующем коде показан базовый класс и производный класс, где производный класс вызывает конструктор базового класса в предложении inherit:

```
type MyClassBase2(x: int) =
   let mutable z = x * x
   do for i in 1..z do printf "%d " i


type MyClassDerived2(y: int) =
   inherit MyClassBase2(y * 2)
   do for i in 1..y do printf "%d " i
```

В случае с несколькими конструкторами можно использовать следующий код. Первая строка конструкторов производного класса является `inherit` предложением, а поля отображаются как явные поля, объявленные `val` с ключевым словом. Дополнительные сведения см. в разделе явные поля: Ключевоеслово. `val`

```
type BaseClass =
    val string1 : string
    new (str) = { string1 = str }
    new () = { string1 = "" }

type DerivedClass =
    inherit BaseClass

    val string2 : string
    new (str1, str2) = { inherit BaseClass(str1); string2 = str2 }
    new (str2) = { inherit BaseClass(); string2 = str2 }

let obj1 = DerivedClass("A", "B")
let obj2 = DerivedClass("A")
```

Альтернативы наследованию

В случаях, когда требуется незначительное изменение типа, рассмотрите возможность использования выражения объекта в качестве альтернативы наследованию. В следующем примере показано использование выражения объекта в качестве альтернативы созданию нового производного типа:

```
open System

let object1 = { new Object() with
      override this.ToString() = "This overrides object.ToString()"
      }

printfn "%s" (object1.ToString())
```

Дополнительные сведения о выражениях объектов см. в разделе выражения объекта.

При создании иерархий объектов рассмотрите возможность использования размеченного объединения вместо наследования. Размеченные объединения могут также моделировать поведение различных объектов, совместно использующих общий тип. Одно размеченное объединение часто может устранить необходимость в ряде производных классов, которые являются незначительными вариантами друг друга. Сведения о размеченных объединениях см. в разделе Размеченные объединения.

См. также

- Выражения объекта
- Справочник по языку F#

Интерфейсы

23.10.2019 • 5 minutes to read • Edit Online

Интерфейсы указывают наборы связанных элементов, реализуемых другими классами.

Синтаксис

```
// Interface declaration:
[ attributes ]
type [accessibility-modifier] interface-name =
    [ interface ]     [ inherit base-interface-name ...]
    abstract member1 : [ argument-types1 -> ] return-type1
    abstract member2 : [ argument-types2 -> ] return-type2
    ...
[ end ]

// Implementing, inside a class type definition:
interface interface-name with
    member self-identifier.member1argument-list = method-body1
    member self-identifier.member2argument-list = method-body2

// Implementing, by using an object expression:
[ attributes ]
let class-name (argument-list) =
    { new interface-name with
        member self-identifier.member1argument-list = method-body1
        member self-identifier.member2argument-list = method-body2
        [ base-interface-definitions ]
    }
    member-list
```

Примечания

Объявления интерфейсов похожи на объявления классов, за исключением того, что ни один член не реализован. Вместо этого все члены являются абстрактными, как указано ключевым словом `abstract`. Не предоставляется тело метода для абстрактных методов. Однако можно предоставить реализацию по умолчанию, включив отдельное определение члена в качестве метода вместе с `default` ключевым словом. Это эквивалентно созданию виртуального метода в базовом классе на других языках .NET. Такой виртуальный метод можно переопределить в классах, реализующих интерфейс.

По умолчанию для интерфейсов `public` используется уровень доступности.

При необходимости можно присвоить каждому параметру метода имя с F# помощью обычного синтаксиса:

```
type ISprintable =
    abstract member Print : format:string -> unit
```

В приведенном `ISprintable` выше примере `Print` метод имеет единственный параметр типа `string` с именем `format`.

Реализовать интерфейсы можно двумя способами: с помощью выражений объектов и типов классов. В любом случае тип класса или выражение объекта предоставляет тела методов для абстрактных методов интерфейса. Реализации относятся к каждому типу, реализующему интерфейс. Поэтому методы

интерфейса для разных типов могут отличаться друг от друга.

Ключевые слова `interface` и `end`, которые отмечают начало и конец определения, являются необязательными при использовании упрощенного синтаксиса. Если эти ключевые слова не используются, компилятор пытается определить, является ли тип классом или интерфейсом, анализируя используемые конструкции. При определении члена или использовании другого синтаксиса класса тип интерпретируется как класс.

Стиль написания кода .NET — это начало всех интерфейсов с заглавной буквой `I`.

Реализация интерфейсов с помощью типов классов

Можно реализовать один или несколько интерфейсов в типе класса с помощью `interface` ключевого слова, имени интерфейса `with` и ключевого слова, а затем определений членов интерфейса, как показано в следующем коде.

```
type IPrintable =
   abstract member Print : unit -> unit

type SomeClass1(x: int, y: float) =
   interface IPrintable with
      member this.Print() = printfn "%d %f" x y
```

Реализации интерфейса наследуются, поэтому все производные классы не нуждаются в их повторной реализации.

Вызов методов интерфейса

Методы интерфейса могут вызываться только через интерфейс, а не через какой-либо объект типа, который реализует интерфейс. Поэтому для вызова этих методов может потребоваться выполнить приведение к типу интерфейса с помощью `:>` оператора `upcast` или оператора.

Чтобы вызвать метод интерфейса при наличии объекта типа `SomeClass`, необходимо выполнить операцию приведения объекта к типу интерфейса, как показано в следующем коде.

```
let x1 = new SomeClass1(1, 2.0)
(x1 :> IPrintable).Print()
```

Альтернативой является объявление метода для объекта, который выполняет приведение и вызов метода интерфейса, как показано в следующем примере.

```
type SomeClass2(x: int, y: float) =
   member this.Print() = (this :> IPrintable).Print()
   interface IPrintable with
      member this.Print() = printfn "%d %f" x y

let x2 = new SomeClass2(1, 2.0)
x2.Print()
```

Реализация интерфейсов с помощью выражений объектов

Выражения объектов предоставляют короткий способ реализации интерфейса. Они полезны в тех случаях, когда не нужно создавать именованный тип и нужен только объект, поддерживающий методы интерфейса, без каких-либо дополнительных методов. В следующем коде показано выражение объекта.

```
let makePrintable(x: int, y: float) =
    { new IPrintable with
              member this.Print() = printfn "%d %f" x y }
let x3 = makePrintable(1, 2.0)
x3.Print()
```

Наследование интерфейса

Интерфейсы могут наследовать от одного или нескольких базовых интерфейсов.

```
type Interface1 =
    abstract member Method1 : int -> int

type Interface2 =
    abstract member Method2 : int -> int

type Interface3 =
    inherit Interface1
    inherit Interface2
    abstract member Method3 : int -> int

type MyClass() =
    interface Interface3 with
        member this.Method1(n) = 2 * n
        member this.Method2(n) = n + 100
        member this.Method3(n) = n / 10
```

См. также

- Справочник по языку F#
- Выражения объекта
- Классы

Абстрактные классы

23.10.2019 • 7 minutes to read • Edit Online

Абстрактные классы — это классы, которые оставляют некоторые или все элементы нереализованными, чтобы реализации могли предоставляться производными классами.

Синтаксис

```
// Abstract class syntax.
[<AbstractClass>]
type [ accessibility-modifier ] abstract-class-name =
[ inherit base-class-or-interface-name ]
[ abstract-member-declarations-and-member-definitions ]

// Abstract member syntax.
abstract member member-name : type-signature
```

Примечания

В объектно-ориентированном программировании абстрактный класс используется в качестве базового класса иерархии и представляет общие функциональные возможности различных наборов типов объектов. Как предполагает имя "abstract", абстрактные классы часто не соответствуют конкретным сущностям в области, в которой возникла проблема. Однако они показывают, сколько различных конкретных сущностей встречается чаще всего.

Абстрактные классы должны иметь `AbstractClass` атрибут. Они могут иметь реализованные и нереализованные члены. Использование термина *abstract* при применении к классу аналогично условию в других языках .NET; Однако использование термина *abstract* при применении к методам (и свойствам) немного отличается F# от его использования в других языках .NET. В F#при условии, что метод помечен с `abstract` помощью ключевого слова, это означает, что элемент имеет запись, называемую *виртуальным слотом диспетчеризации*, во внутренней таблице виртуальных функций для этого типа. Иными словами, метод является виртуальным, хотя `virtual` ключевое слово не используется в F# языке. Ключевое `abstract` слово используется в виртуальных методах независимо от того, реализован ли метод. Объявление виртуального слота диспетчеризации отделено от определения метода для этого слота диспетчеризации. Таким образом, F# эквивалент объявления виртуального метода и определения в другом языке .NET является сочетанием объявления абстрактного метода и отдельного определения с `default` ключевым словом или `override` ключевым словом. Дополнительные сведения и примеры см. в разделе методы.

Класс считается абстрактным, только если существуют абстрактные методы, которые объявлены, но не определены. Поэтому классы, имеющие абстрактные методы, не обязательно являются абстрактными классами. Если класс не имеет неопределенных абстрактных методов, не используйте атрибут **абстракткласс** .

В предыдущем синтаксисе *Модификатор Accessibility* может иметь `public` `private` значение или `internal` . Дополнительные сведения см. в статье Управление доступом.

Как и в случае с другими типами, абстрактные классы могут иметь базовый класс и один или несколько базовых интерфейсов. Каждый базовый класс или интерфейс отображается в отдельной строке вместе с `inherit` ключевым словом.

Определение типа абстрактного класса может содержать полностью определенные члены, но оно также

может содержать абстрактные члены. Синтаксис абстрактных элементов показан отдельно в предыдущем синтаксисе. В этом синтаксисе *сигнатура типа* элемента является списком, который содержит типы параметров в порядке и возвращаемые типы, разделенные токенами и `->` (или `*`) маркерами в соответствии с переданным и кортежными параметрами. Синтаксис сигнатур типов абстрактных элементов такой же, как и в файлах сигнатур, которые отображаются в IntelliSense в редакторе Visual Studio Code.

В следующем коде показана абстрактная фигура класса, которая имеет два неабстрактных производных класса: квадратные и круговые. В примере показано, как использовать абстрактные классы, методы и свойства. В примере фигура абстрактного класса представляет общие элементы конкретных сущностей Circle и Square. Общие функции всех фигур (в двухмерной системе координат) являются абстрактными в классе Shape: положение в сетке, угол поворота, а также свойства области и периметра. Они могут быть переопределены, за исключением расположения, поведения отдельных фигур, которые не могут изменяться.

Метод вращения можно переопределить, как в классе Circle, который является неизменяемым в связи с его симметрией. Поэтому в классе Circle метод вращения заменяется методом, который ничего не делает.

```
// An abstract class that has some methods and properties defined
// and some left abstract.
[<AbstractClass>]
type Shape2D(x0 : float, y0 : float) =
    let mutable x, y = x0, y0
    let mutable rotAngle = 0.0

    // These properties are not declared abstract. They
    // cannot be overriden.
    member this.CenterX with get() = x and set xval = x <- xval
    member this.CenterY with get() = y and set yval = y <- yval

    // These properties are abstract, and no default implementation
    // is provided. Non-abstract derived classes must implement these.
    abstract Area : float with get
    abstract Perimeter : float  with get
    abstract Name : string with get

    // This method is not declared abstract. It cannot be
    // overridden.
    member this.Move dx dy =
       x <- x + dx
       y <- y + dy

    // An abstract method that is given a default implementation
    // is equivalent to a virtual method in other .NET languages.
    // Rotate changes the internal angle of rotation of the square.
    // Angle is assumed to be in degrees.
    abstract member Rotate: float -> unit
    default this.Rotate(angle) = rotAngle <- rotAngle + angle

type Square(x, y, sideLengthIn) =
    inherit Shape2D(x, y)
    member this.SideLength = sideLengthIn
    override this.Area = this.SideLength * this.SideLength
    override this.Perimeter = this.SideLength * 4.
    override this.Name = "Square"

type Circle(x, y, radius) =
    inherit Shape2D(x, y)
    let PI = 3.141592654
    member this.Radius = radius
    override this.Area = PI * this.Radius * this.Radius
    override this.Perimeter = 2. * PI * this.Radius
    // Rotating a circle does nothing, so use the wildcard
    // character to discard the unused argument and
    // evaluate to unit.
```

```
    override this.Rotate(_) = ()
    override this.Name = "Circle"

let square1 = new Square(0.0, 0.0, 10.0)
let circle1 = new Circle(0.0, 0.0, 5.0)
circle1.CenterX <- 1.0
circle1.CenterY <- -2.0
square1.Move -1.0 2.0
square1.Rotate 45.0
circle1.Rotate 45.0
printfn "Perimeter of square with side length %f is %f, %f"
        (square1.SideLength) (square1.Area) (square1.Perimeter)
printfn "Circumference of circle with radius %f is %f, %f"
        (circle1.Radius) (circle1.Area) (circle1.Perimeter)

let shapeList : list<Shape2D> = [ (square1 :> Shape2D);
                                  (circle1 :> Shape2D) ]
List.iter (fun (elem : Shape2D) ->
              printfn "Area of %s: %f" (elem.Name) (elem.Area))
          shapeList
```

Выходные данные:

```
Perimeter of square with side length 10.000000 is 40.000000
Circumference of circle with radius 5.000000 is 31.415927
Area of Square: 100.000000
Area of Circle: 78.539816
```

См. также

- Классы
- Члены
- Методы
- Свойства

Члены

04.11.2019 • 2 minutes to read • Edit Online

Эта статья описывает элементы типов объектов F#.

Заметки

Элементы являются компонентами, которые входят в состав определения типа и объявляются с помощью ключевого слова `member`. Типы объектов F#, такие как записи, классы, размеченные объединения, интерфейсы и структуры, поддерживают элементы. Дополнительные сведения см. в статьях Записи, Классы, Размеченные объединения, Интерфейсы и Структуры.

Обычно элементы составляют открытый интерфейс для типа, поэтому они считаются открытыми, если не указано иное. Элементы также можно объявлять как закрытые или внутренние. Дополнительные сведения см. в статье Управление доступом. Сигнатуры для типов также можно использовать, чтобы предоставлять или не предоставлять определенные элементы типа. Дополнительные сведения см. в статье Сигнатуры.

Частные поля и привязки `do`, которые используются только с классами, не являются подлинными элементами, так как они никогда входят в состав открытого интерфейса типа и не объявляются с помощью ключевого слова `member`, но они также описаны в этой статье.

См. также

РАЗДЕЛ	ОПИСАНИЕ
Привязки `let` в классах	Описывает определение частных полей и функций в классах.
Привязки `do` в классах	Описывает указание кода инициализации объекта.
Свойства	Описывает элементы свойств в классах и других типах.
Индексированные свойства	Описывает массивоподобные свойства в классах и других типах.
Методы	Описывает функции, являющиеся элементами типа.
Конструкторы	Описывает специальные функции, инициализирующие объекты типа.
Перегрузка операторов	Описывает определение настраиваемых операторов для типов.
События	Описывается определение событий и поддержку обработки событий в F#.
Явные поля. Ключевое слово `val`	Описывает определение неинициализированных полей в типе.

Привязки let в классах

23.10.2019 • 3 minutes to read • Edit Online

Закрытые поля и закрытые функции для F# классов можно определить с `let` помощью привязок в определении класса.

Синтаксис

```
// Field.
[static] let [ mutable ] binding1 [ and ... binding-n ]

// Function.
[static] let [ rec ] binding1 [ and ... binding-n ]
```

Примечания

Предыдущий синтаксис отображается после объявления класса и объявлений наследования, но перед определениями элементов. Синтаксис выглядит так же, `let` как в привязках за пределами классов, но имена, определенные в классе, имеют область, ограниченную классом. `let` Привязка создает закрытое поле или функцию; чтобы предоставить доступ к данным или функциям, объявите свойство или метод члена.

Привязка, которая не является статической, называется привязкой `let` экземпляра. `let` Привязки `let` экземпляров выполняются при создании объектов. Статические `let` привязки являются частью статического инициализатора класса, который гарантированно выполняется перед первым использованием типа.

Код в привязках `let` экземпляров может использовать параметры первичного конструктора.

Атрибуты и модификаторы доступности не разрешены в `let` привязках в классах.

В следующих примерах кода показано несколько типов `let` привязок в классах.

```
type PointWithCounter(a: int, b: int) =
    // A variable i.
    let mutable i = 0

    // A let binding that uses a pattern.
    let (x, y) = (a, b)

    // A private function binding.
    let privateFunction x y = x * x + 2*y

    // A static let binding.
    static let mutable count = 0

    // A do binding.
    do
       count <- count + 1

    member this.Prop1 = x
    member this.Prop2 = y
    member this.CreatedCount = count
    member this.FunctionValue = privateFunction x y

let point1 = PointWithCounter(10, 52)

printfn "%d %d %d %d" (point1.Prop1) (point1.Prop2) (point1.CreatedCount) (point1.FunctionValue)
```

Выходные данные выглядят следующим образом.

```
10 52 1 204
```

Альтернативные способы создания полей

Для создания закрытого поля `val` также можно использовать ключевое слово. При использовании `val` ключевого слова поле не получает значение при создании объекта, а инициализируется со значением по умолчанию. Дополнительные сведения см. в разделе явные поля: Ключевое словоVal.

Можно также определить закрытые поля в классе, используя определение элемента и добавив ключевое слово `private` в определение. Это может быть полезно, если вы планируете изменить доступность члена, не переписывая код. Дополнительные сведения см. в статье Управление доступом.

См. также

- Члены
- Привязки `do` в классах
- `let` Привязки

Привязки do в классах

23.10.2019 • 4 minutes to read • Edit Online

Привязка в определении класса выполняет действия при создании объекта или для статической `do` привязки при первом использовании типа. `do`

Синтаксис

```
[static] do expression
```

Примечания

Привязка отображается вместе с привязками или `let` после них, но до определений элементов в определении класса. `do` Хотя ключевое слово является необязательным для `do` привязок на уровне модуля, оно не является обязательным `do` для привязок в определении класса. `do`

Для создания каждого объекта любого заданного типа нестатические `do` привязки и нестатические `let` привязки выполняются в том порядке, в котором они указаны в определении класса. В `do` одном типе может быть несколько привязок. Нестатические `let` привязки и нестатические `do` привязки становятся телом основного конструктора. Код в разделе нестатические `do` привязки может ссылаться на параметры первичного конструктора и любые значения или функции, определенные `let` в разделе привязки.

Нестатические `do` привязки могут обращаться к членам класса, если у класса есть собственный идентификатор, определяемый `as` ключевым словом в заголовке класса, и при условии, что все варианты использования этих членов дополнены идентификатором Self для класса.

Поскольку `let` привязки инициализируют закрытые поля класса, что часто требуется, чтобы гарантировать, что элементы будут вести себя ожидаемым `do` образом, привязки обычно помещаются `let` после привязок, чтобы код в `do` привязке мог выполните с полностью инициализированным объектом. Если код пытается использовать член до завершения инициализации, создается исключение InvalidOperationException.

Статические `do` привязки могут ссылаться на статические члены или поля включающего класса, но не на элементы или поля экземпляра. Статические `do` привязки становятся частью статического инициализатора класса, который гарантированно выполняется перед первым использованием класса.

Атрибуты для `do` привязок в типах игнорируются. Если для кода, выполняемого в `do` привязке, требуется атрибут, он должен быть применен к основному конструктору.

В следующем коде класс имеет статическую `do` привязку и нестатичную `do` привязку. Объект содержит конструктор с двумя параметрами: `a` `b`, и два закрытых `let` поля определены в привязках для класса. Также определяются два свойства. Все они находятся в области в разделе нестатические `do` привязки, как показано в строке, в которой печатаются все эти значения.

```
open System

type MyType(a:int, b:int) as this =
    inherit Object()
    let x = 2*a
    let y = 2*b
    do printfn "Initializing object %d %d %d %d %d %d"
               a b x y (this.Prop1) (this.Prop2)
    static do printfn "Initializing MyType."
    member this.Prop1 = 4*x
    member this.Prop2 = 4*y
    override this.ToString() = System.String.Format("{0} {1}", this.Prop1, this.Prop2)

let obj1 = new MyType(1, 2)
```

Выходные данные выглядят следующим образом.

```
Initializing MyType.
Initializing object 1 2 2 4 8 16
```

См. также

- Члены
- Классы
- Конструкторы
- Привязки `let` в классах
- `do` Привязки

Свойства

23.10.2019 • 11 minutes to read • Edit Online

Свойства — это члены, представляющие значения, связанные с объектом.

Синтаксис

```
// Property that has both get and set defined.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with [accessibility-modifier] get() =
    get-function-body
and [accessibility-modifier] set parameter =
    set-function-body

// Alternative syntax for a property that has get and set.
[ attributes-for-get ]
[ static ] member [accessibility-modifier-for-get] [self-identifier.]PropertyName =
    get-function-body
[ attributes-for-set ]
[ static ] member [accessibility-modifier-for-set] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName =
    get-function-body

// Alternative syntax for property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with get() =
    get-function-body

// Property that has set only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Automatically implemented properties.
[ attributes ]
[ static ] member val [accessibility-modifier] PropertyName = initialization-expression [ with get, set ]
```

Примечания

Свойства представляют собой отношение "имеет" в объектно-ориентированном программировании, представляющее данные, связанные с экземплярами объектов, или для статических свойств с типом.

Свойства можно объявить двумя способами, в зависимости от того, нужно ли явно указать базовое значение (также называемое резервным хранилищем) для свойства, или если вы хотите разрешить компилятору автоматически создавать резервное хранилище. Как правило, следует использовать более явный способ, если свойство имеет нетривиальные реализации и автоматический способ, если свойство является просто простой оболочкой для значения или переменной. Чтобы объявить свойство явным образом, используйте `member` ключевое слово. За этим декларативным синтаксисом следует синтаксис,

который задает `get` методы `set` и, а также именованные способы *доступа*. Различные формы явного синтаксиса, показанного в разделе синтаксиса, используются для свойств для чтения и записи, только для чтения и только для записи. Для свойств только для чтения определяется только `get` метод; для свойств только для записи определите `set` только метод. Обратите внимание, что если свойство `get` имеет `set` методы доступа и, альтернативный синтаксис позволяет указать атрибуты и модификаторы доступа, которые отличаются для каждого метода доступа, как показано в следующем коде.

```
// A read-only property.
member this.MyReadOnlyProperty = myInternalValue
// A write-only property.
member this.MyWriteOnlyProperty with set (value) = myInternalValue <- value
// A read-write property.
member this.MyReadWriteProperty
    with get () = myInternalValue
    and set (value) = myInternalValue <- value
```

Для свойств чтения и записи, `get` имеющих оба `get` метода и `set` , порядок и `set` может быть отменен. Кроме того, можно указать синтаксис, отображаемый `get` только для, и синтаксис, `set` отображаемый только вместо использования объединенного синтаксиса. Это упрощает комментирование отдельных `get` методов или `set` , если это может потребоваться. В следующем коде показана альтернатива использованию объединенного синтаксиса.

```
member this.MyReadWriteProperty with get () = myInternalValue
member this.MyReadWriteProperty with set (value) = myInternalValue <- value
```

Закрытые значения, которые содержат данные для свойств, называются *резервными хранилищами*. Чтобы компилятор автоматически создает резервное хранилище, используйте ключевые слова `member val` , опустите сам себя, а затем укажите выражение для инициализации свойства. Если свойство должно быть изменяемым, включите `with get, set` . Например, следующий тип класса включает два автоматически реализуемых свойства. `Property1` параметр доступен только для чтения и инициализируется с аргументом, предоставленным первичному конструктору `Property2` , и является устанавливаемым свойством, инициализированным в виде пустой строки:

```
type MyClass(property1 : int) =
member val Property1 = property1
member val Property2 = "" with get, set
```

Автоматически реализованные свойства являются частью инициализации типа, поэтому они должны быть включены перед любыми другими определениями элементов, `let` как привязки и `do` привязки в определении типа. Обратите внимание, что выражение, которое инициализирует автоматически реализуемое свойство, вычисляется только при инициализации, а не всякий раз, когда осуществляется доступ к свойству. Это поведение отличается от поведения явно реализованного свойства. Это фактически означает, что код для инициализации этих свойств добавляется в конструктор класса. Рассмотрим следующий код, демонстрирующий это различие:

```
type MyClass() =
    let random  = new System.Random()
    member val AutoProperty = random.Next() with get, set
    member this.ExplicitProperty = random.Next()

let class1 = new MyClass()

printfn "class1.AutoProperty = %d" class1.AutoProperty
printfn "class1.AutoProperty = %d" class1.AutoProperty
printfn "class1.ExplicitProperty = %d" class1.ExplicitProperty
printfn "class1.ExplicitProperty = %d" class1.ExplicitProperty
```

Выходные данные

```
class1.AutoProperty = 1853799794
class1.AutoProperty = 1853799794
class1.ExplicitProperty = 978922705
class1.ExplicitProperty = 1131210765
```

Выходные данные приведенного выше кода показывают, что значение автосвойства не изменяется при повторном вызове, в то время как ЕксплиЦитпроперти изменяется при каждом вызове. Это показывает, что выражение для автоматически реализуемого свойства не вычисляется каждый раз, как и метод считывания для явного свойства.

> **WARNING**
> Существуют некоторые библиотеки, например Entity Framework (`System.Data.Entity`), выполняющие пользовательские операции в конструкторах базовых классов, которые не подходят для инициализации автоматически реализуемых свойств. В таких случаях попробуйте использовать явные свойства.

Свойства могут быть членами классов, структур, размеченных объединений, записей, интерфейсов и расширений типов, а также могут быть определены в выражениях объекта.

Атрибуты могут применяться к свойствам. Чтобы применить атрибут к свойству, запишите атрибут в отдельной строке перед свойством. Дополнительные сведения см. в разделе Атрибуты.

По умолчанию свойства являются открытыми. Модификаторы доступа также могут применяться к свойствам. Чтобы применить модификатор доступа, добавьте его непосредственно перед именем свойства, если оно `get` должно применяться к методам и `set` `get` . Добавьте его перед ключевыми словами и `set` , если они отличаются. требуется для каждого метода доступа. *Модификатором Accessibility* может быть один из следующих: `public` , `private` , `internal` . Дополнительные сведения см. в статье Управление доступом.

Реализации свойств выполняются каждый раз при доступе к свойству.

Статические и свойства экземпляра

Свойства могут быть статическими или свойствами экземпляра. Статические свойства могут вызываться без экземпляра и использоваться для значений, связанных с типом, а не с отдельными объектами. Для статических свойств опустите собственный идентификатор. Для свойств экземпляра необходим собственный идентификатор.

Следующее Статическое определение свойства основано на сценарии, в котором имеется статическое поле `myStaticValue` , являющееся резервным хранилищем для свойства.

```
static member MyStaticProperty
    with get() = myStaticValue
    and set(value) = myStaticValue <- value
```

Свойства также могут быть подобны массиву, в этом случае они называются *индексированными свойствами*. Дополнительные сведения см. в разделе индексированные свойства.

Аннотация типа для свойств

Во многих случаях компилятор имеет достаточно информации для определения типа свойства из типа резервного хранилища, но можно явно задать тип, добавив аннотацию типа.

```
// To apply a type annotation to a property that does not have an explicit
// get or set, apply the type annotation directly to the property.
member this.MyProperty1 : int = myInternalValue
// If there is a get or set, apply the type annotation to the get or set method.
member this.MyProperty2 with get() : int = myInternalValue
```

Использование методов доступа set свойств

Свойства, предоставляющие `set` методы доступа, можно задать `<-` с помощью оператора.

```
// Assume that the constructor argument sets the initial value of the
// internal backing store.
let mutable myObject = new MyType(10)
myObject.MyProperty <- 20
printfn "%d" (myObject.MyProperty)
```

Выходные данные имеют значение **20**.

Абстрактные свойства

Свойства могут быть абстрактными. Как и в случае `abstract` с методами, просто означает, что существует виртуальная диспетчеризация, связанная со свойством. Абстрактные свойства могут быть действительно абстрактными, то есть без определения в том же классе. Таким образом, класс, содержащий такое свойство, является абстрактным классом. Кроме того, abstract может означать, что свойство является виртуальным, и в этом случае определение должно присутствовать в том же классе. Обратите внимание, что абстрактные свойства не должны быть частными, и если один метод доступа является абстрактным, то другой также должен быть абстрактным. Дополнительные сведения о абстрактных классах см. в разделе абстрактные классы.

```
// Abstract property in abstract class.
// The property is an int type that has a get and
// set method
[<AbstractClass>]
type AbstractBase() =
   abstract Property1 : int with get, set

// Implementation of the abstract property
type Derived1() =
   inherit AbstractBase()
   let mutable value = 10
   override this.Property1 with get() = value and set(v : int) = value <- v

// A type with a "virtual" property.
 type Base1() =
   let mutable value = 10
   abstract Property1 : int with get, set
   default this.Property1 with get() = value and set(v : int) = value <- v

// A derived type that overrides the virtual property
type Derived2() =
   inherit Base1()
   let mutable value2 = 11
   override this.Property1 with get() = value2 and set(v) = value2 <- v
```

См. также

- Члены
- Методы

Индексированные свойства

25.11.2019 • 4 minutes to read • Edit Online

При определении класса, который является абстрактным по отношению к упорядоченным данным, иногда может быть полезно предоставить индексированный доступ к этим данным без предоставления базовой реализации. Это выполняется с помощью элемента `Item`.

Синтаксис

```
// Indexed property that can be read and written to
member self-identifier.Item
    with get(index-values) =
        get-member-body
    and set index-values values-to-set =
        set-member-body

// Indexed property can only be read
member self-identifier.Item
    with get(index-values) =
        get-member-body

// Indexed property that can only be set
member self-identifier.Item
    with set index-values values-to-set =
        set-member-body
```

Заметки

Формы предыдущего синтаксиса показывают, как определить индексированные свойства, которые содержат как `get`, так и метод `set`, иметь только метод `get` или только метод `set`. Можно также сочетать синтаксис, показанный только для Get, и синтаксис, показанный только для Set, и создать свойство, которое имеет и Get, и Set. Эта последняя форма позволяет размещать в методах get и Set разные модификаторы и атрибуты доступа.

При использовании имени `Item` компилятор рассматривает свойство как индексированное свойство по умолчанию. *Индексированное свойство по умолчанию* — это свойство, доступ к которому можно получить с помощью синтаксиса, подобного массиву, в экземпляре объекта. Например, если `o` является объектом типа, определяющего это свойство, то синтаксис `o.[index]` используется для доступа к свойству.

Синтаксис для доступа к индексированному свойству, не являющемуся по умолчанию, заключается в предоставлении имени свойства и индекса в круглых скобках, как и обычный элемент. Например, если свойство в `o` называется `Ordinal`, для доступа к нему `o.Ordinal(index)`ся запись.

Независимо от используемой формы следует всегда использовать каррированных вида метода Set для индексированного свойства. Дополнительные сведения о каррированных функциях см. в разделе функции.

Пример

В следующем примере кода показано определение и использование индексированных свойств по умолчанию и не являющихся свойствами по умолчанию, имеющих методы get и Set.

```
type NumberStrings() =
   let mutable ordinals = [| "one"; "two"; "three"; "four"; "five";
                             "six"; "seven"; "eight"; "nine"; "ten" |]
   let mutable cardinals = [| "first"; "second"; "third"; "fourth";
                              "fifth"; "sixth"; "seventh"; "eighth";
                              "ninth"; "tenth" |]
   member this.Item
      with get(index) = ordinals.[index]
      and set index value = ordinals.[index] <- value
   member this.Ordinal
      with get(index) = ordinals.[index]
      and set index value = ordinals.[index] <- value
   member this.Cardinal
      with get(index) = cardinals.[index]
      and set index value = cardinals.[index] <- value

let nstrs = new NumberStrings()
nstrs.[0] <- "ONE"
for i in 0 .. 9 do
  printf "%s " (nstrs.[i])
printfn ""

nstrs.Cardinal(5) <- "6th"

for i in 0 .. 9 do
  printf "%s " (nstrs.Ordinal(i))
  printf "%s " (nstrs.Cardinal(i))
printfn ""
```

Вывод

```
ONE two three four five six seven eight nine ten
ONE first two second three third four fourth five fifth six 6th
seven seventh eight eighth nine ninth ten tenth
```

Индексированные свойства с несколькими значениями индекса

Индексированные свойства могут иметь более одного значения индекса. В этом случае при использовании свойства значения разделяются запятыми. Метод Set в таком свойстве должен иметь два каррированных аргумента, первый из которых является кортежем, содержащим ключи, а второй — заданным значением.

В следующем коде показано использование индексированного свойства с несколькими значениями индекса.

```
open System.Collections.Generic

/// Basic implementation of a sparse matrix based on a dictionary
type SparseMatrix() =
    let table = new Dictionary<(int * int), float>()
    member _.Item
        // Because the key is comprised of two values, 'get' has two index values
        with get(key1, key2) = table.[(key1, key2)]

        // 'set' has two index values and a new value to place in the key's position
        and set (key1, key2) value = table.[(key1, key2)] <- value

let sm = new SparseMatrix()
for i in 1..1000 do
    sm.[i, i] <- float i * float i
```

См. также

- Члены

Методы

25.11.2019 • 11 minutes to read • Edit Online

Метод — это функция, которая связана с типом. В объектно-ориентированном программировании методы используются для предоставления и реализации функциональных возможностей и поведения объектов и типов.

Синтаксис

```
// Instance method definition.
[ attributes ]
member [inline] self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Static method definition.
[ attributes ]
static member [inline] method-name parameter-list [ : return-type ] =
    method-body

// Abstract method declaration or virtual dispatch slot.
[ attributes ]
abstract member method-name : type-signature

// Virtual method declaration and default implementation.
[ attributes ]
abstract member method-name : type-signature
[ attributes ]
default self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Override of inherited virtual method.
[ attributes ]
override self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Optional and DefaultParameterValue attributes on input parameters
[ attributes ]
[ modifier ] member [inline] self-identifier.method-name ([<Optional; DefaultParameterValue( default-value
)>] input) [ : return-type ]
```

Заметки

В предыдущем синтаксисе можно увидеть различные формы объявлений и определений методов. В более длинных теле метода разрыв строки следует за знаком равенства (=), а весь текст метода — с отступом.

Атрибуты могут применяться к любому объявлению метода. Они предшествуют синтаксису определения метода и обычно перечисляются в отдельной строке. Дополнительные сведения см. в разделе Атрибуты.

Методы могут быть помечены `inline`. Сведения о `inline` см. в статье Встраиваемые функции.

Невстроенные методы можно использовать рекурсивно в пределах типа. нет необходимости явно использовать ключевое слово `rec`.

Методы экземпляра

Методы экземпляра объявляются с помощью ключевого слова `member` и *самоидентификатора*, за

которым следует точка (.) и имя и параметры метода. Как и в случае с `let` ными привязками, *параметр-List* может быть шаблоном. Как правило, параметры метода заключаются в круглые скобки в форме кортежа, то есть методы отображаются в F# , когда они создаются на других языках .NET Framework. Однако также часто встречается каррированных формы (параметры, разделенные пробелами), а также поддерживаются и другие шаблоны.

В следующем примере показано определение и использование неабстрактного метода экземпляра.

```
type SomeType(factor0: int) =
   let factor = factor0
   member this.SomeMethod(a, b, c) =
      (a + b + c) * factor

   member this.SomeOtherMethod(a, b, c) =
      this.SomeMethod(a, b, c) * factor
```

В методах экземпляра не используйте собственный идентификатор для доступа к полям, определенным с помощью привязок let. Используйте собственный идентификатор при доступе к другим членам и свойствам.

Статические методы

Ключевое слово `static` используется, чтобы указать, что метод может быть вызван без экземпляра и не связан с экземпляром объекта. В противном случае методы являются методами экземпляра.

В примере в следующем разделе показаны поля, объявленные с ключевым словом `let`, членами свойств, объявленными с ключевым словом `member`, и статическим методом, объявленным с помощью ключевого слова `static`.

В следующем примере показано определение и использование статических методов. Предположим, что эти определения методов находятся в классе `SomeType` в предыдущем разделе.

```
static member SomeStaticMethod(a, b, c) =
   (a + b + c)

static member SomeOtherStaticMethod(a, b, c) =
   SomeType.SomeStaticMethod(a, b, c) * 100
```

Абстрактные и виртуальные методы

Ключевое слово `abstract` указывает, что метод имеет виртуальный слот диспетчеризации и может не иметь определения в классе. *Виртуальный слот диспетчеризации* — это запись во внутренней таблице функций, которая используется во время выполнения для поиска вызовов виртуальных функций в объектно-ориентированном типе. Виртуальный механизм диспетчеризации — это механизм, который реализует *полиморфизм*, важный компонент объектно-ориентированного программирования. Класс, имеющий по крайней мере один абстрактный метод без определения, является *абстрактным классом*, что означает, что экземпляры этого класса создавать нельзя. Дополнительные сведения о абстрактных классах см. в разделе абстрактные классы.

Объявления абстрактных методов не включают тело метода. Вместо этого за именем метода следует двоеточие (:) и сигнатура типа для метода. Сигнатура типа метода такая же, как и при использовании IntelliSense при наведении указателя мыши на имя метода в редакторе Visual Studio Code, за исключением случаев, когда имена параметров не указаны. Сигнатуры типов также отображаются интерпретатором (FSI.exe) при работе в интерактивном режиме. Сигнатура типа метода формируется путем перечисления типов параметров, за которыми следует возвращаемый тип с соответствующими символами-разделителями.

Каррированных параметра разделяются `->` и параметры кортежа разделяются `*`. Возвращаемое значение всегда отделяется от аргументов `->` символом. Круглые скобки можно использовать для группирования сложных параметров, например, если тип функции является параметром или чтобы указать, когда кортеж обрабатывается как один параметр, а не как два параметра.

Можно также предоставить определения абстрактных методов по умолчанию, добавив определение в класс и используя ключевое слово `default`, как показано в блоке синтаксиса в этом разделе. Абстрактный метод, имеющий определение в том же классе, эквивалентен виртуальному методу на других языках .NET Framework. Независимо от того, существует ли определение, ключевое слово `abstract` создает новый слот диспетчеризации в таблице виртуальной функции для класса.

Независимо от того, реализует ли базовый класс его абстрактные методы, производные классы могут предоставлять реализации абстрактных методов. Чтобы реализовать абстрактный метод в производном классе, определите метод с тем же именем и сигнатурой в производном классе, за исключением использования ключевого слова `override` или `default` и предоставления тела метода. Ключевые слова `override` и `default` означают точно то же самое. Используйте `override`, если новый метод переопределяет реализацию базового класса; Используйте `default` при создании реализации в том же классе, что и исходное абстрактное объявление. Не используйте ключевое слово `abstract` в методе, который реализует метод, объявленный абстрактным в базовом классе.

В следующем примере показан абстрактный метод `Rotate` с реализацией по умолчанию, эквивалентом виртуального метода .NET Framework.

```
type Ellipse(a0 : float, b0 : float, theta0 : float) =
    let mutable axis1 = a0
    let mutable axis2 = b0
    let mutable rotAngle = theta0
    abstract member Rotate: float -> unit
    default this.Rotate(delta : float) = rotAngle <- rotAngle + delta
```

В следующем примере показан производный класс, переопределяющий метод базового класса. В этом случае переопределение изменяет поведение таким образом, чтобы метод не выполнит никаких действий.

```
type Circle(radius : float) =
    inherit Ellipse(radius, radius, 0.0)
     // Circles are invariant to rotation, so do nothing.
    override this.Rotate(_) = ()
```

Перегруженные методы

Перегруженные методы — это методы, имеющие одинаковые имена в заданном типе, но имеющие разные аргументы. В F#необязательные аргументы обычно используются вместо перегруженных методов. Однако перегруженные методы разрешены на языке, при условии, что аргументы находятся в виде кортежа, а не в каррированных форме.

Необязательные аргументы

Начиная с F# 4,1, можно также иметь необязательные аргументы со значением параметра по умолчанию в методах. Это помогает упростить взаимодействие с C# кодом. Следующий пример демонстрирует синтаксис:

```
// A class with a method M, which takes in an optional integer argument.
type C() =
    _.M([<Optional; DefaultParameterValue(12)>] i) = i + 1
```

Обратите внимание, что значение, передаваемое для `DefaultParameterValue`, должно соответствовать типу входных данных. В приведенном выше примере это `int`. Попытка передать нецелочисленное значение в `DefaultParameterValue` приведет к ошибке компиляции.

Пример: свойства и методы

В следующем примере содержится тип, который содержит примеры полей, закрытых функций, свойств и статического метода.

```
type RectangleXY(x1 : float, y1: float, x2: float, y2: float) =
    // Field definitions.
    let height = y2 - y1
    let width = x2 - x1
    let area = height * width
    // Private functions.
    static let maxFloat (x: float) (y: float) =
      if x >= y then x else y
    static let minFloat (x: float) (y: float) =
      if x <= y then x else y
    // Properties.
    // Here, "this" is used as the self identifier,
    // but it can be any identifier.
    member this.X1 = x1
    member this.Y1 = y1
    member this.X2 = x2
    member this.Y2 = y2
    // A static method.
    static member intersection(rect1 : RectangleXY, rect2 : RectangleXY) =
       let x1 = maxFloat rect1.X1 rect2.X1
       let y1 = maxFloat rect1.Y1 rect2.Y1
       let x2 = minFloat rect1.X2 rect2.X2
       let y2 = minFloat rect1.Y2 rect2.Y2
       let result : RectangleXY option =
         if ( x2 > x1 && y2 > y1) then
           Some (RectangleXY(x1, y1, x2, y2))
         else
           None
       result

// Test code.
let testIntersection =
    let r1 = RectangleXY(10.0, 10.0, 20.0, 20.0)
    let r2 = RectangleXY(15.0, 15.0, 25.0, 25.0)
    let r3 : RectangleXY option = RectangleXY.intersection(r1, r2)
    match r3 with
    | Some(r3) -> printfn "Intersection rectangle: %f %f %f %f" r3.X1 r3.Y1 r3.X2 r3.Y2
    | None -> printfn "No intersection found."

testIntersection
```

См. также

- Члены

Конструкторы

23.10.2019 • 10 minutes to read • Edit Online

В этой статье описывается определение и использование конструкторов для создания и инициализации объектов класса и структуры.

Создание объектов класса

Объекты типов классов имеют конструкторы. Существует два вида конструкторов. Один из них является основным конструктором, параметры которого отображаются в круглых скобках сразу после имени типа. Другие, необязательные дополнительные конструкторы указываются с `new` помощью ключевого слова. Все такие дополнительные конструкторы должны вызывать первичный конструктор.

Первичный конструктор содержит `let` и `do` привязки, которые отображаются в начале определения класса. Привязка объявляет закрытые поля и методы класса `do` ; привязка выполняет код. `let` Дополнительные сведения о `let` привязках в конструкторах классов см. в разделе `let` привязки в классах. Дополнительные сведения о `do` привязках в конструкторах см. в разделе `do` привязки в классах.

Независимо от того, является ли конструктор, который необходимо вызвать, основным конструктором или дополнительным конструктором, можно создавать объекты с помощью `new` выражения с `new` ключевым словом или без него. Объекты инициализируются вместе с аргументами конструктора, путем перечисления аргументов по порядку и разделенных запятыми и заключенных в круглые скобки, а также с помощью именованных аргументов и значений в круглых скобках. Вы также можете задать свойства объекта во время создания объекта, используя имена свойств и назначив значения так же, как при использовании именованных аргументов конструктора.

В следующем коде показан класс, имеющий конструктор, и различные способы создания объектов:

```
// This class has a primary constructor that takes three arguments
// and an additional constructor that calls the primary constructor.
type MyClass(x0, y0, z0) =
    let mutable x = x0
    let mutable y = y0
    let mutable z = z0
    do
        printfn "Initialized object that has coordinates (%d, %d, %d)" x y z
    member this.X with get() = x and set(value) = x <- value
    member this.Y with get() = y and set(value) = y <- value
    member this.Z with get() = z and set(value) = z <- value
    new() = MyClass(0, 0, 0)

// Create by using the new keyword.
let myObject1 = new MyClass(1, 2, 3)
// Create without using the new keyword.
let myObject2 = MyClass(4, 5, 6)
// Create by using named arguments.
let myObject3 = MyClass(x0 = 7, y0 = 8, z0 = 9)
// Create by using the additional constructor.
let myObject4 = MyClass()
```

Выходные данные выглядят следующим образом:

```
Initialized object that has coordinates (1, 2, 3)
Initialized object that has coordinates (4, 5, 6)
Initialized object that has coordinates (7, 8, 9)
Initialized object that has coordinates (0, 0, 0)
```

Создание структур

Структуры соответствуют всем правилам классов. Поэтому у вас может быть первичный конструктор, и можно предоставить дополнительные конструкторы с помощью `new`. Однако существует одно важное различие между структурами и классами: структуры могут иметь конструктор без параметров (то есть один без аргументов), даже если первичный конструктор не определен. Конструктор без параметров инициализирует все поля значением по умолчанию для этого типа, обычно ноль или его эквивалент. Все конструкторы, определяемые для структур, должны иметь по крайней мере один аргумент, чтобы они не конфликтовали с конструктором без параметров.

Кроме того, структуры часто имеют поля, создаваемые с помощью `val` ключевого слова; классы также могут иметь эти поля. Структуры и классы, имеющие поля, определенные с помощью `val` ключевого слова, можно также инициализировать в дополнительных конструкторах с помощью выражений записи, как показано в следующем коде.

```
type MyStruct =
    struct
       val X : int
       val Y : int
       val Z : int
       new(x, y, z) = { X = x; Y = y; Z = z }
    end

let myStructure1 = new MyStruct(1, 2, 3)
```

Дополнительные сведения см. в разделе явные поля: Ключевоеслово. `val`

Исполнение побочных эффектов в конструкторах

Основной конструктор в классе может выполнять код в `do` привязке. Но что делать, если необходимо выполнить код в дополнительном конструкторе без `do` привязки? Для этого используется `then` ключевое слово.

```
 // Executing side effects in the primary constructor and
// additional constructors.
type Person(nameIn : string, idIn : int) =
    let mutable name = nameIn
    let mutable id = idIn
    do printfn "Created a person object."
    member this.Name with get() = name and set(v) = name <- v
    member this.ID with get() = id and set(v) = id <- v
    new() =
        Person("Invalid Name", -1)
        then
            printfn "Created an invalid person object."

let person1 = new Person("Humberto Acevedo", 123458734)
let person2 = new Person()
```

Побочные эффекты основного конструктора по-прежнему выполняются. Таким образом, выходные данные выглядят следующим образом:

```
Created a person object.
Created a person object.
Created an invalid person object.
```

Собственные идентификаторы в конструкторах

В других элементах вы предоставляете имя для текущего объекта в определении каждого члена. Можно также разместить собственный идентификатор в первой строке определения класса с помощью ключевого слова, `as` непосредственно следующего за параметрами конструктора. Этот синтаксис показан в следующем примере.

```
type MyClass1(x) as this =
    // This use of the self identifier produces a warning - avoid.
    let x1 = this.X
    // This use of the self identifier is acceptable.
    do printfn "Initializing object with X =%d" this.X
    member this.X = x
```

В дополнительных конструкторах можно также определить собственный идентификатор, поместив `as` предложение сразу после параметров конструктора. Этот синтаксис показан в следующем примере:

```
type MyClass2(x : int) =
    member this.X = x
    new() as this = MyClass2(0) then printfn "Initializing with X = %d" this.X
```

Проблемы могут возникать при попытке использования объекта до его полного определения. Таким образом, использование собственного идентификатора может привести к тому, что компилятор выдаст предупреждение и вставит дополнительные проверки, чтобы гарантировать, что элементы объекта не будут доступны до инициализации объекта. Собственный идентификатор следует использовать только в `do` привязках первичного конструктора или `then` после ключевого слова в дополнительных конструкторах.

Имя собственного идентификатора не обязательно должно быть `this`. Это может быть любой допустимый идентификатор.

Присвоение значений свойствам при инициализации

Можно присвоить значения свойствам объекта класса в коде инициализации, добавив список назначений формы `property = value` в список аргументов для конструктора. Это показано в следующем примере кода:

```
 type Account() =
    let mutable balance = 0.0
    let mutable number = 0
    let mutable firstName = ""
    let mutable lastName = ""
    member this.AccountNumber
       with get() = number
       and set(value) = number <- value
    member this.FirstName
       with get() = firstName
       and set(value) = firstName <- value
    member this.LastName
       with get() = lastName
       and set(value) = lastName <- value
    member this.Balance
       with get() = balance
       and set(value) = balance <- value
    member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
    member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount


let account1 = new Account(AccountNumber=8782108,
                           FirstName="Darren", LastName="Parker",
                           Balance=1543.33)
```

Следующая версия предыдущего кода иллюстрирует сочетание обычных аргументов, необязательных аргументов и параметров свойств в одном вызове конструктора:

```
type Account(accountNumber : int, ?first: string, ?last: string, ?bal : float) =
   let mutable balance = defaultArg bal 0.0
   let mutable number = accountNumber
   let mutable firstName = defaultArg first ""
   let mutable lastName = defaultArg last ""
   member this.AccountNumber
      with get() = number
      and set(value) = number <- value
   member this.FirstName
      with get() = firstName
      and set(value) = firstName <- value
   member this.LastName
      with get() = lastName
      and set(value) = lastName <- value
   member this.Balance
      with get() = balance
      and set(value) = balance <- value
   member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
   member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount


let account1 = new Account(8782108, bal = 543.33,
                           FirstName="Raman", LastName="Iyer")
```

Конструкторы в унаследованном классе

При наследовании от базового класса, имеющего конструктор, необходимо указать его аргументы в предложении Inherit. Дополнительные сведения см. в разделе конструкторы и наследование.

Статические конструкторы или конструкторы типов

Помимо указания кода для создания объектов, статические `let` и `do` привязку можно создавать в типах классов, которые выполняются до первого использования типа для выполнения инициализации на уровне

типа. Дополнительные сведения см. в разделе `let` привязки в классах и `do` привязках в классах.

См. также

- Члены

События

23.10.2019 • 10 minutes to read • Edit Online

NOTE

Ссылки на справочник по API в этой статье ведут на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

События позволяют связывать вызовы функций с действиями пользователя и являются важным элементом в программировании графического интерфейса пользователя. События могут также инициироваться приложениями или операционной системой.

Обработка событий

При использовании библиотеки графического интерфейса пользователя, такой как Windows Forms или Windows Presentation Foundation (WPF), значительная часть кода в приложении выполняется в ответ на события, предопределенные в библиотеке. Эти предопределенные события являются членами классов графического интерфейса пользователя, таких как формы и элементы управления. Можно добавить произвольное поведение к уже существующему событию, такому как нажатие кнопки, сославшись на интересующее именованное событие (например, событие `Click` класса `Form`) и вызвав метод `Add`, как показано в следующем коде. При запуске этого кода из F# Interactive вызов метода `System.Windows.Forms.Application.Run(System.Windows.Forms.Form)` следует опустить.

```
open System.Windows.Forms

let form = new Form(Text="F# Windows Form",
                    Visible = true,
                    TopMost = true)

form.Click.Add(fun evArgs -> System.Console.Beep())
Application.Run(form)
```

Тип метода `Add` — `('a -> unit) -> unit`. Следовательно, метод, обрабатывающий событие, принимает один параметр — обычно аргументы события — и возвращает значение типа `unit`. В предыдущем примере обработчик события показан как лямбда-выражение. Обработчик события также может представлять собой значение функции, как в следующем примере кода. В следующем примере кода также показано использование параметров обработчика события, содержащих данные, зависящие от типа события. Для события `MouseMove` система передает объект `System.Windows.Forms.MouseEventArgs`, содержащий координаты `X` и `Y` положения указателя.

```
open System.Windows.Forms

let Beep evArgs =
    System.Console.Beep( )


let form = new Form(Text = "F# Windows Form",
                    Visible = true,
                    TopMost = true)

let MouseMoveEventHandler (evArgs : System.Windows.Forms.MouseEventArgs) =
    form.Text <- System.String.Format("{0},{1}", evArgs.X, evArgs.Y)

form.Click.Add(Beep)
form.MouseMove.Add(MouseMoveEventHandler)
Application.Run(form)
```

Создание пользовательских событий

F#события представлены F# классом событий , реализующим интерфейс IEvent . `IEvent` является интерфейсом, объединяющим функциональность двух других интерфейсов `System.IObservable<'T>` и IDelegateEvent. Следовательно, события `Event` обладают функциональными возможностями, эквивалентными возможностям делегатов в других языках, и дополнительно функциональными возможностями интерфейса `IObservable`; это означает, что события F# поддерживают фильтрацию событий и использование функций первого класса и лямбда-выражений языка F# в качестве обработчиков событий. Эта функция предоставляется в модуле Event.

Чтобы создать для класса событие, которое ведет себя точно так же, как любое другое событие платформы .NET Framework, добавьте в класс привязку `let`, определяющую событие `Event` как поле в классе. В качестве аргумента типа можно указать требуемый тип аргумента события или оставить его пустым, чтобы соответствующий тип был выведен компилятором. Необходимо также определить член события, предоставляющего это событие как событие CLI. Этот элемент должен иметь атрибут CLIEvent . Он объявляется как свойство, а его реализация — просто вызовом свойства публикации события. Пользователи класса могут использовать метод `Add` опубликованного события для добавления обработчика. Аргумент метода `Add` может быть лямбда-выражением. Для вызова события можно использовать его свойство `Trigger`, передавая аргументы функции обработчика. Это показано в следующем примере кода. В этом примере выведенный аргумент типа для события — кортеж, представляющий аргументы для лямбда-выражения.

```
open System.Collections.Generic

type MyClassWithCLIEvent() =

    let event1 = new Event<_>()

    [<CLIEvent>]
    member this.Event1 = event1.Publish

    member this.TestEvent(arg) =
        event1.Trigger(this, arg)

let classWithEvent = new MyClassWithCLIEvent()
classWithEvent.Event1.Add(fun (sender, arg) ->
        printfn "Event1 occurred! Object data: %s" arg)

classWithEvent.TestEvent("Hello World!")

System.Console.ReadLine() |> ignore
```

Выходные данные выглядят следующим образом.

```
Event1 occurred! Object data: Hello World!
```

Здесь иллюстрируется дополнительная функциональная возможность, обеспечиваемая модулем `Event`. Следующий пример кода иллюстрирует основное использование функции `Event.create` для создания события и метода-триггера, добавления двух обработчиков события в виде лямбда-выражений и последующего инициирования события для выполнения обоих лямбда-выражений.

```
type MyType() =
    let myEvent = new Event<_>()

    member this.AddHandlers() =
       Event.add (fun string1 -> printfn "%s" string1) myEvent.Publish
       Event.add (fun string1 -> printfn "Given a value: %s" string1) myEvent.Publish

    member this.Trigger(message) =
       myEvent.Trigger(message)

let myMyType = MyType()
myMyType.AddHandlers()
myMyType.Trigger("Event occurred.")
```

Результат выполнения приведенного кода будет следующим.

```
Event occurred.
Given a value: Event occurred.
```

Обработка потоков событий

Вместо добавления обработчика событий для события с помощью функции Event. Add можно использовать функции в `Event` модуле для обработки потоков событий в очень настраиваемых способах. Это делается путем использования оператора прямого конвейера (`|>`) вместе с событием в качестве первого значения в серии вызовов функций и функций модуля `Event` в качестве последующих вызовов функций.

В следующем примере кода демонстрируется, как настроить событие, обработчик которого вызывается только при определенных условиях.

```
let form = new Form(Text = "F# Windows Form",
                    Visible = true,
                    TopMost = true)
form.MouseMove
    |> Event.filter ( fun evArgs -> evArgs.X > 100 && evArgs.Y > 100)
    |> Event.add ( fun evArgs ->
        form.BackColor <- System.Drawing.Color.FromArgb(
            evArgs.X, evArgs.Y, evArgs.X ^^^ evArgs.Y) )
```

Наблюдаемый модуль содержит аналогичные функции, которые работают с наблюдаемыми объектами. Наблюдаемые объекты аналогичны событиям, но они активно подписываются на события только при создании подписки на такой объект.

Реализация события интерфейса

Разработка компонентов пользовательского интерфейса часто начинается с создания новой формы или нового элемента управления, наследуемых от существующих формы или элемента управления. События

часто определяются в интерфейсе. В этом случае для реализации события необходимо реализовать интерфейс. Интерфейс `System.ComponentModel.INotifyPropertyChanged` определяет одно событие `System.ComponentModel.INotifyPropertyChanged.PropertyChanged`. В представленном ниже коде показана реализация события, которое определил этот унаследованный интерфейс:

```
module CustomForm

open System.Windows.Forms
open System.ComponentModel

type AppForm() as this =
    inherit Form()

    // Define the propertyChanged event.
    let propertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs -> this.Property1 <- "text2"
        this.Property2 <- "text3")

    // This property does not have the property-changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property-changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            propertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

    // Expose the PropertyChanged event as a first class .NET event.
    [<CLIEvent>]
    member this.PropertyChanged = propertyChanged.Publish

    // Define the add and remove methods to implement this interface.
    interface INotifyPropertyChanged with
        member this.add_PropertyChanged(handler) = propertyChanged.Publish.AddHandler(handler)
        member this.remove_PropertyChanged(handler) = propertyChanged.Publish.RemoveHandler(handler)

    // This is the event-handler method.
    member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
        let newProperty = this.GetType().GetProperty(args.PropertyName)
        let newValue = newProperty.GetValue(this :> obj) :?> string
        printfn "Property %s changed its value to %s" args.PropertyName newValue

// Create a form, hook up the event handler, and start the application.
let appForm = new AppForm()
let inpc = appForm :> INotifyPropertyChanged
inpc.PropertyChanged.Add(appForm.OnPropertyChanged)
Application.Run(appForm)
```

Если требуется подключить событие в конструкторе, код будет несколько сложнее, поскольку подключение события должно находиться в блоке `then` дополнительного конструктора, как показано в следующем примере:

```
module CustomForm

open System.Windows.Forms
open System.ComponentModel

// Create a private constructor with a dummy argument so that the public
// constructor can have no arguments.
type AppForm private (dummy) as this =
    inherit Form()

    // Define the propertyChanged event.
    let propertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs -> this.Property1 <- "text2"
        this.Property2 <- "text3")

    // This property does not have the property changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            propertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

    [<CLIEvent>]
    member this.PropertyChanged = propertyChanged.Publish

    // Define the add and remove methods to implement this interface.
    interface INotifyPropertyChanged with
        member this.add_PropertyChanged(handler) = this.PropertyChanged.AddHandler(handler)
        member this.remove_PropertyChanged(handler) = this.PropertyChanged.RemoveHandler(handler)

    // This is the event handler method.
    member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
        let newProperty = this.GetType().GetProperty(args.PropertyName)
        let newValue = newProperty.GetValue(this :> obj) :?> string
        printfn "Property %s changed its value to %s" args.PropertyName newValue

    new() as this =
        new AppForm(0)
        then
            let inpc = this :> INotifyPropertyChanged
            inpc.PropertyChanged.Add(this.OnPropertyChanged)

// Create a form, hook up the event handler, and start the application.
let appForm = new AppForm()
Application.Run(appForm)
```

См. также

- Члены
- Обработка и вызов событий
- Лямбда-выражения: Ключевое слово `fun`
- Модуль Control. Event
- > Класс Control. Event<
- Класс args> элемента<управления. Event

Явные поля. Ключевое слово val

23.10.2019 • 7 minutes to read • Edit Online

Ключевое слово `val` используется для объявления места хранения значения в типе класса или структуры без его инициализации. Места хранения, объявленные таким образом, называются *явные поля*. Ключевое слово `val` можно также использовать вместе с ключевым словом `member` для объявления автоматически реализуемого свойства. Дополнительные сведения об автоматически реализуемых свойствах см. в разделе свойства.

Синтаксис

```
val [ mutable ] [ access-modifier ] field-name : type-name
```

Примечания

Обычно поля в типе класса или структуры задаются с помощью привязки `let`. Однако привязки `let` должны инициализироваться как часть конструктора класса, что не всегда возможно, необходимо или желательно. Если требуется неинициализированное поле, можно использовать ключевое слово `val`.

Явные поля могут быть статическими или не статическими. *Модификатор доступа* может иметь `public` значение, `private` или `internal`. По умолчанию явные поля являются открытыми. Это отличается от привязок `let` в классах, которые всегда являются закрытыми.

Атрибут DefaultValue необходим для явных полей в типах классов, имеющих первичный конструктор. Этот атрибут указывает, что поле инициализируется нулевым значением. Тип поля должен поддерживать инициализацию нулем. Тип поддерживает инициализацию нулем, если он является одним из следующих типов:

- Тип-примитив, который имеет нулевое значение.
- Тип, поддерживающий значение null как нормальное значение, как аномальное значение или как представление значения. Сюда входят классы, кортежи, записи, функции, интерфейсы, ссылочные типы .NET, тип `unit` и типы размеченного объединения.
- Тип значения .NET.
- Структура, все поля которой поддерживают нулевое значение по умолчанию.

Например, неизменяемое поле с именем `someField` имеет резервное поле в скомпилированном в .NET представлении с именем `someField@`, и вы обращаетесь к хранимому значению, используя свойство с именем `someField`.

Для изменяемого поля скомпилированное в .NET представление является полем .NET.

> **WARNING**
> Пространство имен `System.ComponentModel` .NET Framework содержит атрибут с таким же именем. Сведения об этом атрибуте см. в разделе DefaultValueAttribute.

В следующем коде показано использование явных полей и для сравнения привязки `let` в классе, имеющем первичный конструктор. Обратите внимание, что привязанное с помощью `let` поле `myInt1`

является закрытым. Если на привязанное с помощью `let` поле `myInt1` существует ссылка из метода-члена, идентификатор самого поля `this` не требуется. Но если вы обращаетесь к явным полям `myInt2` и `myString`, идентификатор самого поля обязателен.

```
type MyType() =
    let mutable myInt1 = 10
    [<DefaultValue>] val mutable myInt2 : int
    [<DefaultValue>] val mutable myString : string
    member this.SetValsAndPrint( i: int, str: string) =
       myInt1 <- i
       this.myInt2 <- i + 1
       this.myString <- str
       printfn "%d %d %s" myInt1 (this.myInt2) (this.myString)

let myObject = new MyType()
myObject.SetValsAndPrint(11, "abc")
// The following line is not allowed because let bindings are private.
// myObject.myInt1 <- 20
myObject.myInt2 <- 30
myObject.myString <- "def"

printfn "%d %s" (myObject.myInt2) (myObject.myString)
```

Выходные данные выглядят следующим образом:

```
11 12 abc
30 def
```

В следующем коде показывается использование явных полей в классе, в котором нет первичного конструктора. В этом случае атрибут `DefaultValue` не требуется, но все поля должны быть инициализированы в конструкторах, определенных для типа.

```
type MyClass =
    val a : int
    val b : int
    // The following version of the constructor is an error
    // because b is not initialized.
    // new (a0, b0) = { a = a0; }
    // The following version is acceptable because all fields are initialized.
    new(a0, b0) = { a = a0; b = b0; }

let myClassObj = new MyClass(35, 22)
printfn "%d %d" (myClassObj.a) (myClassObj.b)
```

В результате получается `35 22`.

В следующем коде демонстрируется использование явных полей в структуре. Поскольку структура является типом значения, она автоматически имеет конструктор без параметров, который устанавливает значения его полей равными нулю. Таким образом, атрибут `DefaultValue` не является обязательным.

```
type MyStruct =
    struct
        val mutable myInt : int
        val mutable myString : string
    end

let mutable myStructObj = new MyStruct()
myStructObj.myInt <- 11
myStructObj.myString <- "xyz"

printfn "%d %s" (myStructObj.myInt) (myStructObj.myString)
```

В результате получается `11 xyz`.

Учтите, что если вы собираетесь инициализировать структуру с `mutable` полями без `mutable` ключевого слова, назначения будут работать с копией структуры, которая будет удалена сразу после назначения. Поэтому структура не изменится.

```
```fsharp
[<Struct>]
type Foo =
  val mutable bar: string
  member self.ChangeBar bar = self.bar <- bar
  new (bar) = {bar = bar}

let foo = Foo "1"
foo.ChangeBar "2" //make implicit copy of Foo, changes the copy, discards the copy, foo remains unchanged
printfn "%s" foo.bar //prints 1

let mutable foo' = Foo "1"
foo'.ChangeBar "2" //changes foo'
printfn "%s" foo'.bar //prints 2
```
```

Явные поля не предназначены для обычного использования. Как правило, по возможности следует использовать привязку `let` в классе вместо явного поля. Явные поля удобны в некоторых сценариях взаимодействия, например если необходимо определить структуру, которая будет использоваться в вызове платформой собственного API или в сценариях COM-взаимодействия. Дополнительные сведения см. в разделе внешние функции. Кроме того, явное поле может потребоваться при работе с генератором кода F#, который порождает классы без первичного конструктора. Явные поля также полезны для потокостатических переменных или подобных конструкций. Дополнительные сведения см. в разделе `System.ThreadStaticAttribute`.

Если ключевые слова `member val` появляются вместе в определении типа, то это определение автоматически реализуемого свойства. Дополнительные сведения см. в разделе Свойства.

См. также

- Свойства
- Члены
- Привязки `let` в классах

Расширения типов

08.01.2020 • 8 minutes to read • Edit Online

Расширения типов (также называемые *приращениями*) — это семейство функций, которые позволяют добавлять новые члены к ранее определенным типам объектов. Доступны следующие три функции.

- Встроенные расширения типов
- Необязательные расширения типов
- Методы расширения

Каждый из них можно использовать в различных сценариях и имеет различные компромиссы.

Синтаксис

```
// Intrinsic and optional extensions
type typename with
    member self-identifier.member-name =
        body
    ...

// Extension methods
open System.Runtime.CompilerServices

[<Extension>]
type Extensions() =
    [static] member self-identifier.extension-name (ty: typename, [args]) =
        body
    ...
```

Встроенные расширения типов

Внутреннее расширение типа — это расширение типа, расширяющее определяемый пользователем тип.

Встроенные расширения типов должны быть определены в том же файле **и** в том же пространстве имен или модуле, что и расширяемый тип. Любое другое определение приведет к тому, что они станут дополнительными расширениями типа.

Встроенные расширения типов иногда являются четким способом разделения функциональных возможностей из объявления типа. В следующем примере показано, как определить внутреннее расширение типа:

```
namespace Example

type Variant =
    | Num of int
    | Str of string

module Variant =
    let print v =
        match v with
        | Num n -> printf "Num %d" n
        | Str s -> printf "Str %s" s

// Add a member to Variant as an extension
type Variant with
    member x.Print() = Variant.print x
```

Использование расширения типа позволяет разделить каждый из следующих элементов:

- Объявление типа `Variant`
- Функции печати `Variant` класса в зависимости от его "формы"
- Способ доступа к функции печати с помощью `.`-нотации в стиле объекта

Это альтернатива определению всех элементов в `Variant`. Хотя это и не является оптимальным подходом, в некоторых ситуациях это может быть понятным представлением функциональных возможностей.

Встроенные расширения типов компилируются как члены типа, которые они расширяют, и отображаются в типе при проверке типа с помощью отражения.

Необязательные расширения типов

Необязательное расширение типа — это расширение, которое отображается вне исходного модуля, пространства имен или сборки расширяемого типа.

Необязательные расширения типов полезны для расширения типа, который вы не определили самостоятельно. Например:

```
module Extensions

type IEnumerable<'T> with
    /// Repeat each element of the sequence n times
    member xs.RepeatElements(n: int) =
        seq {
            for x in xs do
                for _ in 1 .. n -> x
        }
```

Теперь можно получить доступ к `RepeatElements`, как если бы он был членом IEnumerable<T>, если модуль `Extensions` открыт в области, в которой вы работаете.

Дополнительные расширения не отображаются в расширенном типе при проверке с помощью отражения. Необязательные расширения должны находиться в модулях, и они находятся только в области, если модуль, содержащий расширение, открыт или в другой области.

Необязательные элементы расширения компилируются в статические члены, для которых экземпляр объекта передается неявно в качестве первого параметра. Однако они действуют так, как если бы они были членами экземпляров или статическими членами в соответствии с их объявлением.

Необязательные члены расширения также не видны C# или Visual Basic потребителям. Их можно использовать только в другом F# коде.

Универсальное ограничение встроенных и необязательных расширений типов

Можно объявить расширение типа для универсального типа, в котором переменная типа ограничена. Требование заключается в том, что ограничение объявления расширения соответствует ограничению объявленного типа.

Однако даже при совпадении ограничений между объявленным типом и расширением типа можно вывести ограничение в тексте расширенного члена, который накладывает иное требование для параметра типа, чем объявленный тип. Например:

```
open System.Collections.Generic

// NOT POSSIBLE AND FAILS TO COMPILE!
//
// The member 'Sum' has a different requirement on 'T than the type IEnumerable<'T>
type IEnumerable<'T> with
    member this.Sum() = Seq.sum this
```

Невозможно получить этот код для работы с дополнительным расширением типа:

- Как есть, элемент `Sum` имеет другое ограничение для `'T` (`static member get_Zero` и `static member (+)`), чем определено расширением типа.
- Изменение расширения типа с тем же ограничением, что и `Sum` больше не будет соответствовать определенному ограничению в `IEnumerable<'T>`.
- Изменение `member this.Sum` на `member inline this.Sum` приведет к ошибке, в которой ограничения типа не совпадают.

Нужны такие статические методы, как "плавающее место", и их можно представить так, как если бы они расширялись типом. Именно здесь методы расширения становятся необходимыми.

Методы расширения

Наконец, методы расширения (иногда называемыеC# "членами расширения стиля") можно объявить F# в виде статического метода-члена для класса.

Методы расширения полезны при определении расширений для универсального типа, которые ограничивают бы переменную типа. Например:

```
namespace Extensions

open System.Runtime.CompilerServices

[<Extension>]
type IEnumerableExtensions() =
    [<Extension>]
    static member inline Sum(xs: IEnumerable<'T>) = Seq.sum xs
```

При использовании этого кода он будет выглядеть так, как если бы `Sum` был определен в IEnumerable<T>, пока `Extensions` открыт или находится в области.

Другие замечания

Расширения типов также имеют следующие атрибуты:

- Любой тип, к которому можно получить доступ, можно расширить.

- Встроенные и необязательные расширения типов могут определять *любые* типы членов, а не только методы. Итак, свойства расширения также возможны, например.
- Токен `self-identifier` в синтаксисе представляет экземпляр вызываемого типа, как и обычные члены.
- Расширенные элементы могут быть статическими или членами экземпляра.
- Переменные типа в расширении типа должны соответствовать ограничениям объявленного типа.

Для расширений типов также существуют следующие ограничения.

- Расширения типов не поддерживают виртуальные или абстрактные методы.
- Расширения типов не поддерживают методы переопределения в качестве дополнений.
- Расширения типов не поддерживают статически разрешаемые параметры типа.
- Необязательные расширения типов не поддерживают конструкторы как дополнения.
- Расширения типов не могут быть определены для сокращений типов.
- Расширения типов недопустимы для `byref<'T>` (хотя они могут быть объявлены).
- Расширения типов недопустимы для атрибутов (хотя они могут быть объявлены).
- Можно определить расширения, которые перегружают другие методы с тем же именем, но F# компилятор дает предпочтение методам, не являющимся расширениями, если имеется неоднозначный вызов.

Наконец, если существует несколько встроенных расширений типов для одного типа, все элементы должны быть уникальными. Для необязательных расширений типов члены в разных расширениях типов могут иметь одинаковые имена. Ошибки неоднозначности возникают, только если клиентский код открывает две различные области, определяющие одинаковые имена членов.

См. также:

- Справочник по языку F#
- Члены

Параметры и аргументы

05.12.2019 • 16 minutes to read • Edit Online

В этом разделе описывается языковая поддержка для определения параметров и передачи аргументов в функции, методы и свойства. Он содержит сведения о том, как передавать данные по ссылке, а также определять и использовать методы, которые могут принимать переменное число аргументов.

Параметры и аргументы

Параметр term используется для описания имен для значений, которые должны быть указаны. *Аргумент* term используется для значений, предоставленных для каждого параметра.

Параметры могут быть указаны в кортеже или в каррированных форме или в некоторой комбинации этих двух значений. Аргументы можно передать с помощью явного имени параметра. Параметры методов можно указать как необязательные и задавая значение по умолчанию.

Шаблоны параметров

Параметры, предоставляемые функциям и методам, обычно являются шаблонами, разделенными пробелами. Это означает, что в принципе любой из шаблонов, описанных в выражениях Match , можно использовать в списке параметров для функции или элемента.

Обычно методы используют форму кортежа передаваемых аргументов. Это дает более четкий результат с точки зрения других языков .NET, поскольку форма кортежа соответствует способу передачи аргументов в методах .NET.

Каррированных формы чаще всего используются с функциями, созданными с помощью привязок `let`.

В следующем псевдокоде показаны примеры кортежа и каррированных аргумента.

```
// Tuple form.
member this.SomeMethod(param1, param2) = ...
// Curried form.
let function1 param1 param2 = ...
```

Объединенные формы возможны, когда некоторые аргументы находятся в кортежах, а некоторые — нет.

```
let function2 param1 (param2a, param2b) param3 = ...
```

Другие шаблоны также можно использовать в списках параметров, но если шаблон параметра не соответствует всем возможным входным данным, то во время выполнения может быть неполным совпадением. Исключение `MatchFailureException` создается, если значение аргумента не соответствует шаблонам, указанным в списке параметров. Компилятор выдает предупреждение, если шаблон параметра допускает неполные соответствия. По крайней мере один другой шаблон обычно полезен для списков параметров, и это шаблон с подстановочными знаками. Шаблон подстановочного знака используется в списке параметров, если нужно просто игнорировать любые аргументы. Следующий код иллюстрирует использование шаблона с подстановочными знаками в списке аргументов.

```
let makeList _ = [ for i in 1 .. 100 -> i * i ]
// The arguments 100 and 200 are ignored.
let list1 = makeList 100
let list2 = makeList 200
```

Шаблон подстановочного знака может быть полезен, если не нужны передаваемые аргументы, например, в главной точке входа в программу, если вы не заинтересованы в аргументах командной строки, которые обычно предоставляются в виде массива строк, как показано в следующем коде.

```
[<EntryPoint>]
let main _ =
    printfn "Entry point!"
    0
```

Другие шаблоны, которые иногда используются в аргументах, — это шаблон `as`, а также шаблоны идентификаторов, связанные с размеченные объединениями и активными шаблонами. Шаблон размеченного объединения с одним вариантом можно использовать следующим образом.

```
type Slice = Slice of int * int * string

let GetSubstring1 (Slice(p0, p1, text)) =
    printfn "Data begins at %d and ends at %d in string %s" p0 p1 text
    text.[p0..p1]

let substring = GetSubstring1 (Slice(0, 4, "Et tu, Brute?"))
printfn "Substring: %s" substring
```

Выходные данные выглядят следующим образом.

```
Data begins at 0 and ends at 4 in string Et tu, Brute?
Et tu
```

Активные шаблоны могут быть полезны в качестве параметров, например при преобразовании аргумента в нужный формат, как показано в следующем примере:

```
type Point = { x : float; y : float }

let (| Polar |) { x = x; y = y} =
    ( sqrt (x*x + y*y), System.Math.Atan (y/ x) )

let radius (Polar(r, _)) = r
let angle (Polar(_, theta)) = theta
```

Можно использовать шаблон `as` для сохранения совпадающего значения в качестве локального значения, как показано в следующей строке кода.

```
let GetSubstring2 (Slice(p0, p1, text) as s) = s
```

Другой шаблон, который используется иногда, — это функция, которая оставляет последний аргумент без имени, предоставляя в качестве тела функции лямбда-выражение, которое сразу же выполняет сопоставление шаблона с неявным аргументом. Ниже приведен пример кода.

```
let isNil = function [] -> true | _::_ -> false
```

Этот код определяет функцию, которая принимает универсальный список и возвращает `true`, если список пуст, и `false` в противном случае. Использование таких методов может сделать код более трудным для чтения.

Иногда шаблоны, использующие Неполные совпадения, полезны, например, если известно, что списки в программе содержат только три элемента, в списке параметров можно использовать такой шаблон, как показано ниже.

```
let sum [a; b; c;] = a + b + c
```

Использование шаблонов с неполными соответствиями лучше всего зарезервировано для быстрого создания прототипов и других временных применений. Компилятор выдаст предупреждение для такого кода. Такие шаблоны не могут охватывать все возможные входные данные и поэтому не подходят для API-интерфейсов компонентов.

Именованные аргументы

Аргументы для методов могут быть заданы по положению в списке аргументов с разделителями-запятыми или могут быть переданы в метод явным образом путем указания имени, за которым следует знак равенства и значение, которое необходимо передать. Если указано имя, оно может находиться в другом порядке, отличном от того, который используется в объявлении.

Именованные аргументы могут сделать код более удобочитаемым и более адаптируемым к определенным типам изменений в API, например изменить порядок параметров метода.

Именованные аргументы допускаются только для методов, а не для функций, связанных с `let`, значений функций или лямбда-выражений.

В следующем примере кода показано использование именованных аргументов.

```
type SpeedingTicket() =
    member this.GetMPHOver(speed: int, limit: int) = speed - limit

let CalculateFine (ticket : SpeedingTicket) =
    let delta = ticket.GetMPHOver(limit = 55, speed = 70)
    if delta < 20 then 50.0 else 100.0

let ticket1 : SpeedingTicket = SpeedingTicket()
printfn "%f" (CalculateFine ticket1)
```

При вызове конструктора класса можно задать значения свойств класса, используя синтаксис, аналогичный именованным аргументам. Этот синтаксис показан в следующем примере.

```
type Account() =
   let mutable balance = 0.0
   let mutable number = 0
   let mutable firstName = ""
   let mutable lastName = ""
   member this.AccountNumber
      with get() = number
      and set(value) = number <- value
   member this.FirstName
      with get() = firstName
      and set(value) = firstName <- value
   member this.LastName
      with get() = lastName
      and set(value) = lastName <- value
   member this.Balance
      with get() = balance
      and set(value) = balance <- value
   member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
   member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount


let account1 = new Account(AccountNumber=8782108,
                           FirstName="Darren", LastName="Parker",
                           Balance=1543.33)
```

Дополнительные сведения см. в разделе конструкторыF#().

Необязательные параметры

Для метода можно указать необязательный параметр, используя вопросительный знак перед именем параметра. Необязательные параметры интерпретируется как F# тип параметра, поэтому их можно запросить обычным способом запроса типов параметров с помощью `match` выражения с `Some` и `None`. Необязательные параметры разрешены только для членов, но не для функций, созданных с помощью привязок `let`.

Можно передать существующие необязательные значения в метод по имени параметра, например `?arg=None` или `?arg=Some(3)` или `?arg=arg`. Это может быть полезно при создании метода, который передает необязательные аргументы другому методу.

Можно также использовать функцию `defaultArg`, которая задает значение по умолчанию для необязательного аргумента. Функция `defaultArg` принимает необязательный параметр в качестве первого аргумента и значение по умолчанию в качестве второго.

В следующем примере показано использование необязательных параметров.

```
type DuplexType =
    | Full
    | Half

type Connection(?rate0 : int, ?duplex0 : DuplexType, ?parity0 : bool) =
    let duplex = defaultArg duplex0 Full
    let parity = defaultArg parity0 false
    let mutable rate = match rate0 with
                       | Some rate1 -> rate1
                       | None -> match duplex with
                                 | Full -> 9600
                                 | Half -> 4800
    do printfn "Baud Rate: %d Duplex: %A Parity: %b" rate duplex parity

let conn1 = Connection(duplex0 = Full)
let conn2 = Connection(duplex0 = Half)
let conn3 = Connection(300, Half, true)
let conn4 = Connection(?duplex0 = None)
let conn5 = Connection(?duplex0 = Some(Full))

let optionalDuplexValue : option<DuplexType> = Some(Half)
let conn6 = Connection(?duplex0 = optionalDuplexValue)
```

Выходные данные выглядят следующим образом.

```
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false
Baud Rate: 300 Duplex: Half Parity: true
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false
```

В целях C# и Visual Basic Interop можно использовать атрибуты `[<Optional; DefaultParameterValue<(...)>]` в F#, чтобы вызывающие объекты могли видеть аргумент как необязательный. Это эквивалентно определению аргумента в качестве необязательного в C# , как в `MyMethod(int i = 3)`.

```
open System
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultParameterValue("Hello world")>] message) =
        printfn "%s" message
```

Можно также указать новый объект в качестве значения параметра по умолчанию. Например, элемент `Foo` может иметь необязательный `CancellationToken` в качестве входных данных:

```
open System.Threading
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultParameterValue(CancellationToken())>] ct: CancellationToken) =
        printfn "%A" ct
```

Значение, заданное в качестве аргумента для `DefaultParameterValue`, должно соответствовать типу параметра. Например, следующее не разрешено:

```
type C =
    static member Wrong([<Optional; DefaultParameterValue("string")>] i:int) = ()
```

В этом случае компилятор выдает предупреждение и будет полностью игнорировать оба атрибута.

Обратите внимание, что значение по умолчанию `null` должно быть снабжено заметками типа, так как в противном случае компилятор выводит неверный тип, т. е. `[<Optional; DefaultParameterValue(null:obj)>] o:obj`.

Передача по ссылке

Передача F# значения по ссылке включает в себя ByRef, которые являются типами управляемых указателей. Ниже приведены рекомендации по использованию типа.

- Используйте `inref<'T>`, если требуется только чтение указателя.
- Используйте `outref<'T>`, если требуется только запись в указатель.
- Используйте `byref<'T>`, если требуется как чтение, так и запись в указатель.

```
let example1 (x: inref<int>) = printfn "It's %d" x

let example2 (x: outref<int>) = x <- x + 1

let example3 (x: byref<int>) =
    printfn "It'd %d" x
    x <- x + 1

let test () =
    // No need to make it mutable, since it's read-only
    let x = 1
    example1 &x

    // Needs to be mutable, since we write to it
    let mutable y = 2
    example2 &y
    example3 &y // Now 'y' is 3
```

Поскольку параметр является указателем и значение является изменяемым, любые изменения значения сохраняются после выполнения функции.

Можно использовать кортеж в качестве возвращаемого значения для хранения любых `out` параметров в методах библиотеки .NET. Кроме того, параметр `out` можно рассматривать как параметр `byref`. В следующем примере кода показаны оба способа.

```
// TryParse has a second parameter that is an out parameter
// of type System.DateTime.
let (b, dt) = System.DateTime.TryParse("12-20-04 12:21:00")

printfn "%b %A" b dt

// The same call, using an address of operator.
let mutable dt2 = System.DateTime.Now
let b2 = System.DateTime.TryParse("12-20-04 12:21:00", &dt2)

printfn "%b %A" b2 dt2
```

Массивы параметров

Иногда необходимо определить функцию, которая принимает произвольное число параметров разнородного типа. Было бы нецелесообразным создавать все возможные перегруженные методы для учета всех типов, которые можно использовать. Реализации .NET обеспечивают поддержку таких методов с помощью функции массива параметров. Метод, принимающий в сигнатуру массив параметров, может предоставляться с произвольным числом параметров. Параметры помещаются в массив. Тип элементов массива определяет типы параметров, которые могут быть переданы в функцию. Если вы определили

массив параметров с `System.Object` в качестве типа элемента, клиентский код может передавать значения любого типа.

В F#массивы параметров можно определять только в методах. Их нельзя использовать в отдельных функциях или функциях, определенных в модулях.

Массив параметров определяется с помощью атрибута `ParamArray`. Атрибут `ParamArray` можно применить только к последнему параметру.

В следующем коде показано, как вызвать метод .NET, который принимает массив параметров, и определение типа в F# , имеющее метод, принимающий массив параметров.

```
open System

type X() =
    member this.F([<ParamArray>] args: Object[]) =
        for arg in args do
            printfn "%A" arg

[<EntryPoint>]
let main _ =
    // call a .NET method that takes a parameter array, passing values of various types
    Console.WriteLine("a {0} {1} {2} {3} {4}", 1, 10.0, "Hello world", 1u, true)

    let xobj = new X()
    // call an F# method that takes a parameter array, passing values of various types
    xobj.F("a", 1, 10.0, "Hello world", 1u, true)
    0
```

При запуске в проекте выходные данные предыдущего кода выглядят следующим образом:

```
a 1 10 Hello world 1 True
"a"
1
10.0
"Hello world"
1u
true
```

См. также:

- Члены

Перегрузка операторов

23.10.2019 • 10 minutes to read • Edit Online

В этом разделе описывается перегрузка арифметических операторов в классе или типе записи, а также на глобальном уровне.

Синтаксис

```
// Overloading an operator as a class or record member.
static member (operator-symbols) (parameter-list) =
    method-body
// Overloading an operator at the global level
let [inline] (operator-symbols) parameter-list = function-body
```

Примечания

В предыдущем синтаксисе *оператор-Symbol* является одним из `+`, `*` `-`, `/`„ `=` и т. д. *Параметр-List* задает операнды в том порядке, в котором они отображаются в обычном синтаксисе для этого оператора. *Метод-Body* конструирует результирующее значение.

Перегрузки операторов для операторов должны быть статическими. Перегрузки операторов для унарных операторов, таких `+` как `-` и, должны использовать символ тильды (`~`) в *операторе-Symbol* , чтобы указать, что оператор является унарным оператором, а не бинарным оператором, как показано ниже. повторно.

```
static member (~-) (v : Vector)
```

В следующем коде показан класс Vector, имеющий только два оператора: один для унарного минуса и один для умножения скаляром. В этом примере требуется две перегрузки для скалярного умножения, так как оператор должен работать независимо от порядка, в котором отображаются Vector и scalar.

```
type Vector(x: float, y : float) =
   member this.x = x
   member this.y = y
   static member (~-) (v : Vector) =
     Vector(-1.0 * v.x, -1.0 * v.y)
   static member (*) (v : Vector, a) =
     Vector(a * v.x, a * v.y)
   static member (*) (a, v: Vector) =
     Vector(a * v.x, a * v.y)
   override this.ToString() =
     this.x.ToString() + " " + this.y.ToString()

let v1 = Vector(1.0, 2.0)

let v2 = v1 * 2.0
let v3 = 2.0 * v1

let v4 = - v2

printfn "%s" (v1.ToString())
printfn "%s" (v2.ToString())
printfn "%s" (v3.ToString())
printfn "%s" (v4.ToString())
```

Создание новых операторов

Можно перегружать все стандартные операторы, но можно также создавать новые операторы из последовательностей определенных символов. Допустимые символы оператора `!`: `%`, `&`, `*`, `+`„ `-`, , `/`„„ , `>` `=` `<` `.` `?` , `@`, `^` ,и `~`. `|` `~` Символ имеет особое значение, что делает оператор унарным и не является частью последовательности символов оператора. Не все операторы можно сделать унарными.

В зависимости от того, какая последовательность символов используется, оператор будет иметь определенный приоритет и ассоциативность. Ассоциативность может быть либо слева направо, либо справа налево, и используется всякий раз, когда операторы одного и того же уровня приоритета отображаются в последовательности без скобок.

Символ `.` оператора не влияет на приоритет, поэтому, например, если необходимо определить собственную версию умножения, имеющую тот же приоритет и ассоциативность, что и обычное умножение, можно создать такие операторы, как `.*`.

Только операторы `?` и `?<-` могут начинаться с `?`.

Таблица, в которой показан приоритет всех операторов в F# , можно найти в справочнике по символам и операторам.

Имена перегруженных операторов

Когда F# компилятор компилирует выражение оператора, он создает метод с именем, созданным компилятором для этого оператора. Это имя, которое отображается на языке MSIL для метода, а также в отражении и IntelliSense. Обычно вам не нужно использовать эти имена в F# коде.

В следующей таблице показаны стандартные операторы и соответствующие им созданные имена.

ОПЕРАТОР	СОЗДАННОЕ ИМЯ
`[]`	`op_Nil`
`::`	`op_Cons`

ОПЕРАТОР	СОЗДАННОЕ ИМЯ
`+`	`op_Addition`
`-`	`op_Subtraction`
`*`	`op_Multiply`
`/`	`op_Division`
`@`	`op_Append`
`^`	`op_Concatenate`
`%`	`op_Modulus`
`&&&`	`op_BitwiseAnd`
`|||`	`op_BitwiseOr`
`^^^`	`op_ExclusiveOr`
`<<<`	`op_LeftShift`
`~~~`	`op_LogicalNot`
`>>>`	`op_RightShift`
`~+`	`op_UnaryPlus`
`~-`	`op_UnaryNegation`
`=`	`op_Equality`
`<=`	`op_LessThanOrEqual`
`>=`	`op_GreaterThanOrEqual`
`<`	`op_LessThan`
`>`	`op_GreaterThan`
`?`	`op_Dynamic`
`?<-`	`op_DynamicAssignment`
`|>`	`op_PipeRight`
`<|`	`op_PipeLeft`

ОПЕРАТОР	СОЗДАННОЕ ИМЯ
`!`	`op_Dereference`
`>>`	`op_ComposeRight`
`<<`	`op_ComposeLeft`
`<@ @>`	`op_Quotation`
`<@@ @@>`	`op_QuotationUntyped`
`+=`	`op_AdditionAssignment`
`-=`	`op_SubtractionAssignment`
`*=`	`op_MultiplyAssignment`
`/=`	`op_DivisionAssignment`
`..`	`op_Range`
`.. ..`	`op_RangeStep`

Другие сочетания символов операторов, которые не перечислены здесь, могут использоваться в качестве операторов и иметь имена, которые объединяются путем объединения имен для отдельных символов из следующей таблицы. Например, +! будет `op_PlusBang`.

СИМВОЛ ОПЕРАТОРА	NAME
`>`	`Greater`
`<`	`Less`
`+`	`Plus`
`-`	`Minus`
`*`	`Multiply`
`/`	`Divide`
`=`	`Equals`
`~`	`Twiddle`
`%`	`Percent`
`.`	`Dot`

СИМВОЛ ОПЕРАТОРА	NAME
`&`	`Amp`
`|`	`Bar`
`@`	`At`
`^`	`Hat`
`!`	`Bang`
`?`	`Qmark`
`(`	`LParen`
`,`	`Comma`
`)`	`RParen`
`[`	`LBrack`
`]`	`RBrack`

Операторы prefix и инфиксные

Операторы *префикса* должны размещаться перед операндом или операндами, во многом подобно функции. Операторы *инфиксные* должны располагаться между двумя операндами.

В качестве префиксных операторов можно использовать только определенные операторы. Некоторые операторы всегда являются префиксными операторами, другие могут быть инфиксные или префиксом, а остальные всегда инфиксные операторами. Операторы, которые начинаются `!` с оператора `!=` `~`, за исключением and, или повторяющихся последовательностей `~`, всегда являются префиксными операторами. `+` Операторы „ `&&` „„ и `%%` могут быть префиксными операторами или операторами инфиксные. `-` `+.` `-.` `&` `%` Префиксную версию этих операторов можно отличить от версии инфиксные, добавляя в `~` начале префиксного оператора, когда он определен. `~` Не используется при использовании оператора, только если он определен.

Пример

В следующем коде показано использование перегрузки операторов для реализации типа дробной части. Дробная часть представляется числителем и знаменателем. Функция `hcf` используется для определения самого высокого общего фактора, который используется для сокращения дробей.

```
// Determine the highest common factor between
// two positive integers, a helper for reducing
// fractions.
let rec hcf a b =
  if a = 0u then b
  elif a<b then hcf a (b - a)
  else hcf (a - b) b

// type Fraction: represents a positive fraction
// (positive rational number).
```

```
// (positive rational number).
type Fraction =
   {
      // n: Numerator of fraction.
      n : uint32
      // d: Denominator of fraction.
      d : uint32
   }

   // Produce a string representation. If the
   // denominator is "1", do not display it.
   override this.ToString() =
      if (this.d = 1u)
        then this.n.ToString()
        else this.n.ToString() + "/" + this.d.ToString()

   // Add two fractions.
   static member (+) (f1 : Fraction, f2 : Fraction) =
      let nTemp = f1.n * f2.d + f2.n * f1.d
      let dTemp = f1.d * f2.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Adds a fraction and a positive integer.
   static member (+) (f1: Fraction, i : uint32) =
      let nTemp = f1.n + i * f1.d
      let dTemp = f1.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Adds a positive integer and a fraction.
   static member (+) (i : uint32, f2: Fraction) =
      let nTemp = f2.n + i * f2.d
      let dTemp = f2.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Subtract one fraction from another.
   static member (-) (f1 : Fraction, f2 : Fraction) =
      if (f2.n * f1.d > f1.n * f2.d)
        then failwith "This operation results in a negative number, which is not supported."
      let nTemp = f1.n * f2.d - f2.n * f1.d
      let dTemp = f1.d * f2.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Multiply two fractions.
   static member (*) (f1 : Fraction, f2 : Fraction) =
      let nTemp = f1.n * f2.n
      let dTemp = f1.d * f2.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Divide two fractions.
   static member (/) (f1 : Fraction, f2 : Fraction) =
      let nTemp = f1.n * f2.d
      let dTemp = f2.n * f1.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // A full set of operators can be quite lengthy. For example,
   // consider operators that support other integral data types,
   // with fractions, on the left side and the right side for each.
   // Also consider implementing unary operators.

let fraction1 = { n = 3u; d = 4u }
let fraction2 = { n = 1u; d = 2u }
let result1 = fraction1 + fraction2
let result2 = fraction1 - fraction2
let result3 = fraction1 * fraction2
```

```
let result3 = fraction1 * fraction2
let result4 = fraction1 / fraction2
let result5 = fraction1 + 1u
printfn "%s + %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result1.ToString())
printfn "%s - %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result2.ToString())
printfn "%s * %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result3.ToString())
printfn "%s / %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result4.ToString())
printfn "%s + 1 = %s" (fraction1.ToString()) (result5.ToString())
```

Выходные данные:

```
3/4 + 1/2 = 5/4
3/4 - 1/2 = 1/4
3/4 * 1/2 = 3/8
3/4 / 1/2 = 3/2
3/4 + 1 = 7/4
```

Операторы на глобальном уровне

Операторы также можно определять на глобальном уровне. В следующем коде определяется оператор `+?`.

```
let inline (+?) (x: int) (y: int) = x + 2*y
printf "%d" (10 +? 1)
```

Выходные данные приведенного выше кода имеют `12` значение.

Таким образом можно переопределить обычные арифметические операторы, так как правила определения области F# определяют, что новые определенные операторы имеют приоритет над встроенными операторами.

Ключевое `inline` слово часто используется с глобальными операторами, которые часто являются маленькими функциями, которые лучше всего интегрируются в вызывающий код. Использование встроенных функций операторов также позволяет им работать с статически разрешаемыми параметрами типов для формирования статически разрешаемого универсального кода. Дополнительные сведения см. в разделе встроенные функции и статически разрешаемые параметры типа.

См. также

- Члены

Гибкие типы

23.10.2019 • 4 minutes to read • Edit Online

Заметка гибкого типа указывает, что параметр, переменная или значение имеют тип, совместимый с указанным типом, где совместимость определяется положением в объектно-ориентированной иерархии классов или интерфейсов. Гибкие типы полезны в частности, если автоматическое преобразование в типы выше в иерархии типов не выполняется, но все равно требуется обеспечить работу функций с любым типом в иерархии или любым типом, реализующим интерфейс.

Синтаксис

```
#type
```

Примечания

В предыдущем синтаксисе *тип* представляет базовый тип или интерфейс.

Гибкий тип эквивалентен универсальному типу, имеющему ограничение, которое ограничивает допустимые типы типами, совместимыми с базовым или интерфейсным типом. То есть следующие две строки кода эквивалентны.

```
#SomeType

'T when 'T :> SomeType
```

Гибкие типы полезны в ситуациях нескольких типов. Например, если имеется функция более высокого порядка (функция, которая принимает функцию в качестве аргумента), часто бывает полезно, чтобы функция возвращала гибкий тип. В следующем примере использование гибкого типа с аргументом последовательности в `iterate2` позволяет функции более высокого порядка работать с функциями, создающими последовательности, массивы, списки и любые другие перечислимые типы.

Рассмотрим следующие две функции, одна из которых возвращает последовательность, а другая — гибкий тип.

```
let iterate1 (f : unit -> seq<int>) =
    for e in f() do printfn "%d" e
let iterate2 (f : unit -> #seq<int>) =
    for e in f() do printfn "%d" e

// Passing a function that takes a list requires a cast.
iterate1 (fun () -> [1] :> seq<int>)

// Passing a function that takes a list to the version that specifies a
// flexible type as the return value is OK as is.
iterate2 (fun () -> [1])
```

В качестве другого примера рассмотрим функцию Seq. Concat Library:

```
val concat: sequences:seq<#seq<'T>> -> seq<'T>
```

В эту функцию можно передать любую из следующих перечислимых последовательностей:

- Список списков
- Список массивов
- Массив списков
- Массив последовательностей
- Любое другое сочетание перечислимых последовательностей

В следующем коде используется `Seq.concat` для демонстрации сценариев, которые можно поддерживать с помощью гибких типов.

```
let list1 = [1;2;3]
let list2 = [4;5;6]
let list3 = [7;8;9]

let concat1 = Seq.concat [ list1; list2; list3]
printfn "%A" concat1

let array1 = [|1;2;3|]
let array2 = [|4;5;6|]
let array3 = [|7;8;9|]

let concat2 = Seq.concat [ array1; array2; array3 ]
printfn "%A" concat2

let concat3 = Seq.concat [| list1; list2; list3 |]
printfn "%A" concat3

let concat4 = Seq.concat [| array1; array2; array3 |]
printfn "%A" concat4

let seq1 = { 1 .. 3 }
let seq2 = { 4 .. 6 }
let seq3 = { 7 .. 9 }

let concat5 = Seq.concat [| seq1; seq2; seq3 |]

printfn "%A" concat5
```

Выходные данные выглядят следующим образом.

```
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
```

В F#, как и в других объектно-ориентированных языках, существуют контексты, в которых производные типы или типы, реализующие интерфейсы, автоматически преобразуются в базовый тип или тип интерфейса. Эти автоматические преобразования происходят в прямых аргументах, но не в том случае, если тип находится в подчиненной положении, как часть более сложного типа, например тип возвращаемого значения типа функции, или как аргумент типа. Таким образом, нотация гибкого типа особенно полезна, когда тип, к которому он применяется, является частью более сложного типа.

См. также

- Справочник по языку F#
- Универсальные шаблоны

Делегаты

23.10.2019 • 4 minutes to read • Edit Online

Делегат представляет вызов функции как объект. В F#обычно значения функций следует использовать для представления функций в качестве значений первого класса; Однако делегаты используются в .NET Framework и поэтому необходимы при взаимодействии с интерфейсами API, ожидающими их. Их также можно использовать при создании библиотек, предназначенных для использования на других языках .NET Framework.

Синтаксис

```
type delegate-typename = delegate of type1 -> type2
```

Примечания

В предыдущем синтаксисе `type1` представляет тип аргумента или типы и `type2` представляет тип возвращаемого значения. Типы аргументов, представленные `type1` , автоматически являются каррированных. Это предполагает, что для этого типа используется форма кортежа, если аргументы целевой функции являются переданными, и кортеж в круглых скобках для аргументов, которые уже находятся в форме кортежа. Автоматически карринг удаляет набор круглых скобок, в результате чего остается аргумент кортежа, соответствующий целевому методу. Синтаксис, который следует использовать в каждом случае, см. в примере кода.

Делегаты можно прикреплять F# к значениям функций, статическим или методам экземпляра. F#значения функций могут передаваться непосредственно в качестве аргументов конструкторам делегатов. Для статического метода делегат создается с помощью имени класса и метода. Для метода экземпляра вы предоставляете экземпляр объекта и метод в одном аргументе. В обоих случаях используется оператор доступа к членам (`.`).

`Invoke` Метод в типе делегата вызывает инкапсулированную функцию. Кроме того, делегаты можно передавать как значения функций, ссылаясь на имя метода Invoke без скобок.

В следующем коде показан синтаксис для создания делегатов, представляющих различные методы в классе. В зависимости от того, является ли метод статическим методом или методом экземпляра и имеет ли он аргументы в форме кортежа или в каррированных форме, синтаксис объявления и присваивания делегата немного отличается.

```
type Test1() =
  static member add(a : int, b : int) =
     a + b
  static member add2 (a : int) (b : int) =
     a + b

  member x.Add(a : int, b : int) =
     a + b
  member x.Add2 (a : int) (b : int) =
     a + b


// Delegate1 works with tuple arguments.
type Delegate1 = delegate of (int * int) -> int
// Delegate2 works with curried arguments.
type Delegate2 = delegate of int * int -> int

let InvokeDelegate1 (dlg : Delegate1) (a : int) (b: int) =
   dlg.Invoke(a, b)
let InvokeDelegate2 (dlg : Delegate2) (a : int) (b: int) =
   dlg.Invoke(a, b)

// For static methods, use the class name, the dot operator, and the
// name of the static method.
let del1 : Delegate1 = new Delegate1( Test1.add )
let del2 : Delegate2 = new Delegate2( Test1.add2 )

let testObject = Test1()

// For instance methods, use the instance value name, the dot operator, and the instance method name.
let del3 : Delegate1 = new Delegate1( testObject.Add )
let del4 : Delegate2 = new Delegate2( testObject.Add2 )

for (a, b) in [ (100, 200); (10, 20) ] do
  printfn "%d + %d = %d" a b (InvokeDelegate1 del1 a b)
  printfn "%d + %d = %d" a b (InvokeDelegate2 del2 a b)
  printfn "%d + %d = %d" a b (InvokeDelegate1 del3 a b)
  printfn "%d + %d = %d" a b (InvokeDelegate2 del4 a b)
```

В следующем коде показаны различные способы работы с делегатами.

```
type Delegate1 = delegate of int * char -> string

let replicate n c = String.replicate n (string c)

// An F# function value constructed from an unapplied let-bound function
let function1 = replicate

// A delegate object constructed from an F# function value
let delObject = new Delegate1(function1)

// An F# function value constructed from an unapplied .NET member
let functionValue = delObject.Invoke

List.map (fun c -> functionValue(5,c)) ['a'; 'b'; 'c']
|> List.iter (printfn "%s")

// Or if you want to get back the same curried signature
let replicate' n c =  delObject.Invoke(n,c)

// You can pass a lambda expression as an argument to a function expecting a compatible delegate type
// System.Array.ConvertAll takes an array and a converter delegate that transforms an element from
// one type to another according to a specified function.
let stringArray = System.Array.ConvertAll([|'a';'b'|], fun c -> replicate' 3 c)
printfn "%A" stringArray
```

Выходные данные предыдущего примера кода выглядят следующим образом.

```
aaaaa
bbbbb
ccccc
[|"aaa"; "bbb"|]
```

См. также

- Справочник по языку F#
- Параметры и аргументы
- События

Выражения объекта

23.10.2019 • 2 minutes to read • Edit Online

Объекта выражение является выражение, которое создает новый экземпляр динамически создаваемого анонимного типа объекта, который основан на существующем базовом типе, интерфейсе или набор интерфейсов.

Синтаксис

```
// When typename is a class:
{ new typename [type-params]arguments with
    member-definitions
    [ additional-interface-definitions ]
}
// When typename is not a class:
{ new typename [generic-type-args] with
    member-definitions
    [ additional-interface-definitions ]
}
```

Примечания

В приведенном выше синтаксисе *typename* представляет собой существующий тип класса или тип интерфейса. *Tun params* описывает необязательные параметры универсального типа. *Аргументы* используются только для типов классов, которые требуют параметры конструктора. *Определения членов* переопределения методов базового класса или реализаций абстрактных методов базового класса или интерфейса.

Следующий пример иллюстрирует несколько разных типов выражений объекта.

```
// This object expression specifies a System.Object but overrides the
// ToString method.
let obj1 = { new System.Object() with member x.ToString() = "F#" }
printfn "%A" obj1

// This object expression implements the IFormattable interface.
let delimiter(delim1: string, delim2: string, value: string) =
    { new System.IFormattable with
        member x.ToString(format: string, provider: System.IFormatProvider) =
            if format = "D" then
                delim1 + value + delim2
            else
                value }

let obj2 = delimiter("{","}", "Bananas!");

printfn "%A" (System.String.Format("{0:D}", obj2))

// Define two interfaces
type IFirst =
  abstract F : unit -> unit
  abstract G : unit -> unit

type ISecond =
  inherit IFirst
  abstract H : unit -> unit
  abstract J : unit -> unit

// This object expression implements both interfaces.
let implementer() =
    { new ISecond with
        member this.H() = ()
        member this.J() = ()
      interface IFirst with
        member this.F() = ()
        member this.G() = () }
```

С использованием выражений объекта

Объект выражения используются в том случае, если вы хотите избежать дополнительного кода и издержки, который необходим для создания нового именованного типа. Если объект выражения позволяют свести к минимуму количество типов, созданный в приложении, можно сократить число строк кода и предотвратить ненужное увеличение числа типов. Вместо создания множества типов для различных ситуаций, можно использовать выражение объекта, который настраивает существующий тип или предоставляет подходящую реализацию интерфейса для конкретного случая.

См. также

- Справочник по языку F#

Приведение и преобразование (F#)

23.10.2019 • 10 minutes to read • Edit Online

В этом разделе описывается поддержка преобразований типов F#в.

Арифметические типы

F#предоставляет операторы преобразования для арифметических преобразований между различными типами-примитивами, такими как целочисленные и плавающие точки. Операторы целочисленного и символьного преобразования имеют проверенные и непроверенные формы; Операторы с плавающей точкой и `enum` оператор преобразования не имеют значения. Непроверенные формы определяются в `Microsoft.FSharp.Core.Operators` , а проверенные формы определяются в. `Microsoft.FSharp.Core.Operators.Checked` Проверенная форма проверяет наличие переполнения и создает исключение во время выполнения, если полученное значение превышает ограничения целевого типа.

Имя каждого из этих операторов совпадает с именем целевого типа. Например, в следующем коде, в котором типы помечены явно, `byte` отображаются с двумя разными значениями. Первое вхождение — это тип, а второй — оператор преобразования.

```
let x : int = 5

let b : byte = byte x
```

В следующей таблице показаны операторы преобразования, определенные F#в.

ОПЕРАТОР	ОПИСАНИЕ
`byte`	Преобразование в Byte, 8-разрядный тип без знака.
`sbyte`	Преобразовать в байт со знаком.
`int16`	Преобразует в 16-разрядное целое число со знаком.
`uint16`	Преобразование в 16-битовое целое число без знака.
`int32, int`	Преобразование в 32-разрядное целое число со знаком.
`uint32`	Преобразование в 32-битовое целое число без знака.
`int64`	Преобразование в 64-разрядное целое число со знаком.
`uint64`	Преобразование в 64-битовое целое число без знака.
`nativeint`	Преобразовать в собственное целое число.
`unativeint`	Преобразование в собственное целое число без знака.

ОПЕРАТОР	ОПИСАНИЕ
`float, double`	Преобразование в 64-разрядное число с плавающей запятой двойной точности (Double).
`float32, single`	Преобразование в 32-разрядное число с плавающей запятой одиночной точности (однозначная).
`decimal`	Преобразовать в `System.Decimal`.
`char`	Преобразование в `System.Char` символ Юникода.
`enum`	Преобразование в перечисляемый тип.

Помимо встроенных типов-примитивов, эти операторы можно использовать с типами, которые реализуют `op_Explicit` методы `op_Implicit` или с соответствующими сигнатурами. Например, `int` оператор преобразования работает с любым типом, предоставляющим статический метод `op_Explicit` , который принимает тип в качестве параметра и возвращает `int`. В качестве специального исключения для общего правила, которое методы не могут перегружаться типом возвращаемого значения, это можно сделать для `op_Explicit` и `op_Implicit`.

Перечислимые типы

Оператор является универсальным оператором, принимающим один параметр типа, представляющий тип `enum` преобразуемого объекта. `enum` При преобразовании в перечислимый тип определение типа пытается определить тип `enum` объекта, в который необходимо выполнить преобразование. В следующем примере переменная `col1` не объявляется явно, но ее тип выводится из более поздней проверки на равенство. Таким образом, компилятор может вывести на себя `Color` преобразование в перечисление. Кроме того, можно указать аннотацию типа, как `col2` показано в следующем примере.

```
type Color =
    | Red = 1
    | Green = 2
    | Blue = 3

// The target type of the conversion is determined by type inference.
let col1 = enum 1
// The target type is supplied by a type annotation.
let col2 : Color = enum 2
do
    if (col1 = Color.Red) then
       printfn "Red"
```

Можно также явно указать целевой тип перечисления в качестве параметра типа, как показано в следующем коде:

```
let col3 = enum<Color> 3
```

Обратите внимание, что приведенные перечисления работают только в том случае, если базовый тип перечисления совместим с преобразуемым типом. В следующем коде не удается скомпилировать преобразование из-за несоответствия между `int32` и. `uint32`

```
// Error: types are incompatible
let col4 : Color = enum 2u
```

Дополнительные сведения см. в разделе перечисления.

Приведение типов объектов

Преобразование между типами в иерархии объектов является фундаментальным для объектно-ориентированного программирования. Существует два базовых типа преобразований: приведение к исполнению (исходящее) и приведение вниз (образование производных). Приведение вверх по иерархии означает приведение производного объекта к ссылке на базовый объект. Такое приведение гарантированно будет работать, если базовый класс находится в иерархии наследования производного класса. Приведение вниз к иерархии из базового объекта в ссылку на производный объект завершается, только если объект фактически является экземпляром правильного (производного) типа или типа, производного от целевого типа.

F#предоставляет операторы для таких типов преобразований. Оператор выполняет приведение к иерархии вверх, `:?>` и оператор приводит вниз к иерархии. `:>`

Treat

Во многих объектно-ориентированных языках приведение является неявным; в F#правила отличаются друг от друга. При передаче аргументов в методы в типе объекта применяется автоматическое приведение. Однако для функций, связанных с let, в модуле не выполняется автоматическое приведение, если тип параметра не объявлен как гибкий тип. Дополнительные сведения см. в разделе гибкие типы.

`:>` Оператор выполняет статическое приведение, которое означает, что успешность приведения определяется во время компиляции. Если приведение, использующее `:>` компиляцию, успешно выполняется, оно является допустимым приведением и не имеет шансов на ошибку во время выполнения.

Можно также использовать `upcast` оператор для выполнения такого преобразования. Следующее выражение задает преобразование в иерархию:

```
upcast expression
```

При использовании оператора CAST компилятор пытается определить тип, к которому выполняется преобразование, из контекста. Если компилятору не удается определить тип целевого объекта, компилятор сообщает об ошибке.

Опуститься

`:?>` Оператор выполняет динамическое приведение, то есть успешность приведения определяется во время выполнения. Приведение, использующее `:?>` оператор, не проверяется во время компиляции, но во время выполнения предпринимается попытка приведения к указанному типу. Если объект совместим с целевым типом, приведение будет выполняться с ошибкой. Если объект несовместим с целевым типом, среда выполнения создает исключение `InvalidCastException`.

`downcast` Оператор также можно использовать для выполнения динамического преобразования типов. Следующее выражение задает преобразование иерархии в тип, выводимый из контекста программы:

```
downcast expression
```

Как и для `upcast` оператора, если компилятор не может вывести конкретный целевой тип из контекста, он сообщает об ошибке.

В следующем коде показано использование `:>` операторов и. `:?>` Этот код показывает, что `:?>` оператор лучше использовать, если известно, что преобразование будет успешным, так как при `InvalidCastException` неудачном завершении преобразование вызывает исключение. Если неизвестно, что преобразование будет выполнено, то тест типа, использующий `match` выражение, лучше, так как он позволяет избежать издержек, вызванных созданием исключения.

```
type Base1() =
    abstract member F : unit -> unit
    default u.F() =
     printfn "F Base1"

type Derived1() =
    inherit Base1()
    override u.F() =
      printfn "F Derived1"


let d1 : Derived1 = Derived1()

// Upcast to Base1.
let base1 = d1 :> Base1

// This might throw an exception, unless
// you are sure that base1 is really a Derived1 object, as
// is the case here.
let derived1 = base1 :?> Derived1

// If you cannot be sure that b1 is a Derived1 object,
// use a type test, as follows:
let downcastBase1 (b1 : Base1) =
   match b1 with
   | :? Derived1 as derived1 -> derived1.F()
   | _ -> ()

downcastBase1 base1
```

Поскольку универсальные `downcast` операторы `upcast` и полагаются на определение типа для определения аргумента и типа возвращаемого значения, в приведенном выше коде можно заменить

```
let base1 = d1 :> Base1
```

на

```
let base1 = upcast d1
```

В приведенном выше коде тип аргумента и возвращаемые типы `Derived1` имеют `Base1` значение и соответственно.

Дополнительные сведения о типах тестов см. в разделе выражения Match.

См. также

- Справочник по языку F#

Управление доступом

04.11.2019 • 4 minutes to read • Edit Online

Контроль доступа означает объявление того, какие клиенты могут использовать определенные элементы программы, такие как типы, методы и функции.

Основы управления доступом

В F#службах описатели управления доступом `public`, `internal` и `private` могут применяться к модулям, типам, методам, определениям значений, функциям, свойствам и явным полям.

- `public` указывает, что доступ к сущности может осуществляться всеми вызывающими объектами.
- `internal` указывает, что доступ к сущности можно получить только из той же сборки.
- `private` указывает, что доступ к сущности можно получить только из включающего типа или модуля.

> **NOTE**
>
> Спецификатор доступа `protected` не используется в F#, хотя он приемлем, если вы используете типы, созданные на языках, поддерживающих `protected` доступ. Поэтому при переопределении защищенного метода метод остается доступным только внутри класса и его потомков.

В общем случае описатель помещается перед именем сущности, за исключением случаев, когда используется спецификатор `mutable` или `inline`, который появляется после описателя контроля доступа.

Если спецификатор доступа не используется, по умолчанию используется `public`, за исключением привязок `let` в типе, которые всегда `private` типу.

Сигнатуры F# в предоставляют еще один механизм управления доступом F# к программным элементам. Для управления доступом подписи не требуются. Дополнительные сведения см. в статье Сигнатуры.

Правила для контроля доступа

Управление доступом подчиняется следующим правилам.

- Объявления наследования (то есть использование `inherit` для указания базового класса для класса), объявления интерфейсов (т. е. Указание, что класс реализует интерфейс), а абстрактные элементы всегда имеют тот же уровень доступа, что и включающий тип. Поэтому в этих конструкциях нельзя использовать описатель управления доступом.
- Специальные возможности для отдельных вариантов в размеченного Union определяются уровнем доступности размеченного объединения. То есть конкретный вариант объединения не менее доступен, чем сам союз.
- Специальные возможности для отдельных полей типа записи определяются специальными возможностями записи. Это означает, что конкретная метка записи не менее доступна, чем сама запись.

Пример

В следующем коде показано использование описателей управления доступом. В проекте есть два файла: `Module1.fs` и `Module2.fs`. Каждый файл является неявным модулем. Таким образом, существует два модуля: `Module1` и `Module2`. Частный тип и внутренний тип определяются в `Module1`. Доступ к закрытому типу из `Module2` невозможен, но внутренний тип может.

```
// Module1.fs

module Module1

// This type is not usable outside of this file
type private MyPrivateType() =
   // x is private since this is an internal let binding
   let x = 5
   // X is private and does not appear in the QuickInfo window
   // when viewing this type in the Visual Studio editor
   member private this.X() = 10
   member this.Z() = x * 100

type internal MyInternalType() =
   let x = 5
   member private this.X() = 10
   member this.Z() = x * 100

// Top-level let bindings are public by default,
// so "private" and "internal" are needed here since a
// value cannot be more accessible than its type.
let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

// let bindings at the top level are public by default,
// so result1 and result2 are public.
let result1 = myPrivateObj.Z
let result2 = myInternalObj.Z
```

Следующий код проверяет доступность типов, созданных в `Module1.fs`.

```
// Module2.fs
module Module2

open Module1

// The following line is an error because private means
// that it cannot be accessed from another file or module
// let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

let result = myInternalObj.Z
```

См. также

- Справочник по языку F#
- Сигнатуры

Условные выражения: `if...then...else`

23.10.2019 • 2 minutes to read • Edit Online

`if...then...else` Выражение выполняет различные ветви кода, а также вычисляет различные значения в зависимости от заданного логического выражения.

Синтаксис

```
if boolean-expression then expression1 [ else expression2 ]
```

Примечания

В предыдущем синтаксисе *expression1* выполняется, когда логическое выражение принимает `true` значение; в противном случае — *Expression2* .

В отличие от других языков, `if...then...else` конструкция является выражением, а не оператором. Это означает, что он создает значение, которое является значением последнего выражения в ветви, в которой выполняется. Типы значений, создаваемых в каждой ветви, должны совпадать. Если явная `else` ветвь отсутствует, ее тип — `unit` . Поэтому, если тип `then` ветви отличается от `unit` типа `else` , необходимо наличие ветви с таким же типом возвращаемого значения. При совместном `if...then...else` связывании выражений можно использовать ключевое слово `elif` , а `else if` не. они эквивалентны.

Пример

В следующем примере показано, `if...then...else` как использовать выражение.

```
let test x y =
  if x = y then "equals"
  elif x < y then "is less than"
  else "is greater than"

printfn "%d %s %d." 10 (test 10 20) 20

printfn "What is your name? "
let nameString = System.Console.ReadLine()

printfn "What is your age? "
let ageString = System.Console.ReadLine()
let age = System.Int32.Parse(ageString)

if age < 10
then printfn "You are only %d years old and already learning F#? Wow!" age
```

```
10 is less than 20
What is your name? John
How old are you? 9
You are only 9 years old and already learning F#? Wow!
```

См. также

- Справочник по языку F#

Выражения соответствия

23.10.2019 • 5 minutes to read • Edit Online

`match` Выражение обеспечивает управление ветвлением, основанное на сравнении выражения с набором шаблонов.

Синтаксис

```
// Match expression.
match test-expression with
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...

// Pattern matching function.
function
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...
```

Примечания

Выражения сопоставления шаблонов позволяют выполнять сложное ветвление на основе сравнения тестового выражения с набором шаблонов. В выражении выражение *места* сравнивается с каждым шаблоном, а при обнаружении совпадения вычисляется соответствующее результирующее выражение, а итоговое значение возвращается в качестве значения выражения соответствия. `match`

Функция сопоставления шаблонов, показанная в предыдущем синтаксисе, представляет собой лямбда-выражение, в котором сопоставление шаблонов выполняется непосредственно над аргументом. Функция сопоставления шаблонов, показанная в предыдущем синтаксисе, эквивалентна следующей.

```
fun arg ->
    match arg with
    | pattern1 [ when condition ] -> result-expression1
    | pattern2 [ when condition ] -> result-expression2
    | ...
```

Дополнительные сведения о лямбда-выражениях см . в разделе Лямбда-выражения: Ключевоеслово. `fun`

Весь набор шаблонов должен охватывать все возможные совпадения входной переменной. Часто шаблон подстановочного знака (`_`) используется в качестве последнего шаблона для сопоставления с любыми ранее несовпадающими входными значениями.

В следующем коде показаны некоторые способы `match` использования выражения. Справочные сведения и примеры всех возможных шаблонов, которые можно использовать, см. в разделе сопоставление шаблонов.

```
let list1 = [ 1; 5; 100; 450; 788 ]

// Pattern matching by using the cons pattern and a list
// pattern that tests for an empty list.
let rec printList listx =
    match listx with
    | head :: tail -> printf "%d " head; printList tail
    | [] -> printfn ""

printList list1

// Pattern matching with multiple alternatives on the same line.
let filter123 x =
    match x with
    | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
    | a -> printfn "%d" a

// The same function written with the pattern matching
// function syntax.
let filterNumbers =
    function | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
             | a -> printfn "%d" a
```

Условия для шаблонов

`when` Предложение можно использовать для указания дополнительного условия, которым должна удовлетворять переменная для соответствия шаблону. Такое предложение называется *условием*. Выражение, следующее `when` за ключевым словом, не вычисляется, если в шаблон, связанный с этим условием, не установлено соответствие.

В следующем примере показано использование условия для указания числового диапазона для шаблона переменной. Обратите внимание, что несколько условий объединяются с помощью логических операторов.

```
let rangeTest testValue mid size =
    match testValue with
    | var1 when var1 >= mid - size/2 && var1 <= mid + size/2 -> printfn "The test value is in range."
    | _ -> printfn "The test value is out of range."

rangeTest 10 20 5
rangeTest 10 20 10
rangeTest 10 20 40
```

Обратите внимание, что, поскольку значения, отличные от литералов, не могут использоваться в шаблоне, необходимо использовать `when` предложение, если необходимо сравнить некоторую часть входных данных со значением. Это продемонстрировано в приведенном ниже коде.

```
// This example uses patterns that have when guards.
let detectValue point target =
    match point with
    | (a, b) when a = target && b = target -> printfn "Both values match target %d." target
    | (a, b) when a = target -> printfn "First value matched target in (%d, %d)" target b
    | (a, b) when b = target -> printfn "Second value matched target in (%d, %d)" a target
    | _ -> printfn "Neither value matches target."
detectValue (0, 0) 0
detectValue (1, 0) 0
detectValue (0, 10) 0
detectValue (10, 15) 0
```

Обратите внимание, что если шаблон объединения охватывается условием, условие применяется ко **всем** шаблонам, а не только к последнему. Например, при наличии следующего кода условие `when a > 12` применяется `A a` к и `B a`:

```
type Union =
    | A of int
    | B of int

let foo() =
    let test = A 42
    match test with
    | A a
    | B a when a > 41 -> a // the guard applies to both patterns
    | _ -> 1

foo() // returns 42
```

См. также

- Справочник по языку F#
- Активные шаблоны
- Соответствие шаблону

Сопоставление шаблонов

29.10.2019 • 19 minutes to read • Edit Online

Шаблоны — это правила для преобразования входных данных. Они используются на всех F# языках для сравнения данных с логической структурой или структурами, разложения данных в составные части или извлечения информации из данных различными способами.

Заметки

Шаблоны используются во многих языковых конструкциях, например в `match` выражении. Они используются при обработке аргументов для функций в `let` привязках, лямбда-выражениях и в обработчиках исключений, связанных с выражением `try...with`. Дополнительные сведения см. в разделе выражения Match, Привязка let, лямбда-выражения: ключевое слово `fun` и исключения: выражение `try...with`.

Например, в выражении `match` *шаблон* соответствует символу вертикальной черты.

```
match expression with
| pattern [ when condition ] -> result-expression
...
```

Каждый шаблон выступает в качестве правила для преобразования входных данных каким-либо образом. В `match` выражении каждый шаблон просматривается, в свою очередь, чтобы проверить, совместимы ли входные данные с шаблоном. При обнаружении совпадения выполняется результирующее выражение. Если совпадение не найдено, проверяется следующее правило шаблона. Необязательная часть *условия* when объясняется в выражениях Match.

Поддерживаемые шаблоны показаны в следующей таблице. Во время выполнения входные данные тестируются по каждому из следующих шаблонов в порядке, указанном в таблице, а шаблоны применяются рекурсивно, от первого до последнего, как они отображаются в коде, и слева направо для шаблонов в каждой строке.

НАЗВАНИЕ	ОПИСАНИЕ	ПРИМЕР
Шаблон константы	Любой числовой, символьный или строковый литерал, константа перечисления или определенный литеральный идентификатор	`1.0`, `"test"`, `30`, `Color.Red`
Шаблон идентификатора	Значение варианта для размеченного объединения, метки исключения или активного варианта шаблона	`Some(x)` `Failure(msg)`
Шаблон переменной	*identifier*	`a`
шаблон `as`	*шаблон* в качестве *идентификатора*	`(a, b) as tuple1`
ИЛИ шаблон	*pattern1* | *pattern2*	`([h] | [h; _])`

НАЗВАНИЕ	ОПИСАНИЕ	ПРИМЕР
И шаблон	*pattern1* & *pattern2*	`(a, b) & (_, "test")`
Шаблон «против»	*идентификатор* :: *List-identifier*	`h :: t`
Шаблон списка	[*pattern_1*;...; *pattern_n*]	`[ a; b; c ]`
Шаблон массива	[| *pattern_1*;..; *pattern_n* |]	`[| a; b; c |]`
Шаблон в круглых скобках	(*шаблон*)	`( a )`
Шаблон кортежа	(*pattern_1*,..., *pattern_n*)	`( a, b )`
Шаблон записи	{ *идентификатор1* = *pattern_1*;...; *identifier_n* = *pattern_n* }	`{ Name = name; }`
Шаблон подстановочного знака		`_`
Шаблон вместе с аннотацией типа	*шаблон* : *тип*	`a : int`
Шаблон проверки типа	:? *введите* [как *идентификатор*]	`:? System.DateTime as dt`
Шаблон NULL	null	`null`

Шаблоны констант

Шаблоны констант — это числовые, символьные и строковые литералы, константы перечисления (с включаемым именем типа перечисления). Выражение `match`, имеющее только постоянные шаблоны, можно сравнивать с оператором case на других языках. Входные данные сравниваются с литеральным значением, а шаблон соответствует, если значения равны. Тип литерала должен быть совместим с типом входных данных.

В следующем примере демонстрируется использование литеральных шаблонов, а также используется шаблон переменной и шаблон или.

```
[<Literal>]
let Three = 3

let filter123 x =
    match x with
    // The following line contains literal patterns combined with an OR pattern.
    | 1 | 2 | Three -> printfn "Found 1, 2, or 3!"
    // The following line contains a variable pattern.
    | var1 -> printfn "%d" var1

for x in 1..10 do filter123 x
```

Еще один пример литерального шаблона — это шаблон, основанный на константах перечисления. При использовании констант перечисления необходимо указать имя типа перечисления.

```
type Color =
    | Red = 0
    | Green = 1
    | Blue = 2

let printColorName (color:Color) =
    match color with
    | Color.Red -> printfn "Red"
    | Color.Green -> printfn "Green"
    | Color.Blue -> printfn "Blue"
    | _ -> ()

printColorName Color.Red
printColorName Color.Green
printColorName Color.Blue
```

Шаблоны идентификаторов

Если шаблон представляет собой строку символов, образующих допустимый идентификатор, форма идентификатора определяет способ сопоставления шаблона. Если идентификатор длиннее одного символа и начинается с символа верхнего регистра, компилятор пытается выполнить сопоставление с шаблоном идентификатора. Идентификатором для этого шаблона может быть значение, помеченное атрибутом Literal, вариантом размеченного объединения, идентификатором исключения или активным шаблоном. Если соответствующий идентификатор не найден, сопоставление завершается ошибкой, а следующее правило шаблона, шаблон переменной, сравнивается с входными данными.

Шаблоны размеченного объединения могут быть простыми именованными вариантами или иметь значение или кортеж, содержащий несколько значений. Если имеется значение, необходимо указать идентификатор для этого значения. В случае кортежа необходимо предоставить шаблон кортежа с идентификатором для каждого элемента кортежа или идентификатором с именем поля для одного или нескольких именованных полей объединения. Примеры см. в примерах кода в этом разделе.

Тип `option` — это размеченное объединение, которое имеет два варианта: `Some` и `None`. Один вариант (`Some`) имеет значение, а другое (`None`) — только именованный вариант. Таким образом, `Some` должен иметь переменную для значения, связанного с `Some` ным вариантом, но `None` должен быть сам по себе. В следующем коде переменной `var1` присваивается значение, полученное путем сопоставления с `Some` ным вариантом.

```
let printOption (data : int option) =
    match data with
    | Some var1  -> printfn "%d" var1
    | None -> ()
```

В следующем примере `PersonName` размеченное объединение содержит сочетание строк и символов, представляющих возможные формы имен. Варианты размеченного объединения: `FirstOnly` , `LastOnly` и `FirstLast` .

```
type PersonName =
    | FirstOnly of string
    | LastOnly of string
    | FirstLast of string * string

let constructQuery personName =
    match personName with
    | FirstOnly(firstName) -> printf "May I call you %s?" firstName
    | LastOnly(lastName) -> printf "Are you Mr. or Ms. %s?" lastName
    | FirstLast(firstName, lastName) -> printf "Are you %s %s?" firstName lastName
```

Для размеченных объединений с именованными полями используется знак равенства (=) для извлечения значения именованного поля. Например, рассмотрим размеченное объединение с объявлением, как показано ниже.

```
type Shape =
    | Rectangle of height : float * width : float
    | Circle of radius : float
```

Именованные поля можно использовать в выражении сопоставления шаблонов следующим образом.

```
let matchShape shape =
    match shape with
    | Rectangle(height = h) -> printfn "Rectangle with length %f" h
    | Circle(r) -> printfn "Circle with radius %f" r
```

Использование именованного поля является необязательным, поэтому в предыдущем примере оба `Circle(r)` и `Circle(radius = r)` имеют одинаковый результат.

При указании нескольких полей используйте точку с запятой (;) в качестве разделителя.

```
match shape with
| Rectangle(height = h; width = w) -> printfn "Rectangle with height %f and width %f" h w
| _ -> ()
```

Активные шаблоны позволяют определить более сложное сопоставление пользовательских шаблонов. Дополнительные сведения об активных шаблонах см. в разделе Активные закономерности.

Случай, когда идентификатор является исключением, используется в сопоставлении шаблонов в контексте обработчиков исключений. Сведения о сопоставлении шаблонов в обработке исключений см. в разделе Exceptions: the `try...with` выражение.

Шаблоны переменных

Шаблон переменной присваивает значение, совпадающее с именем переменной, которое затем можно использовать в выражении выполнения справа от символа `->`. Только шаблон переменной соответствует любым входным данным, но шаблоны переменных часто появляются в других шаблонах, поэтому для реализации более сложных структур, таких как кортежи и массивы, следует разложить их на переменные.

В следующем примере демонстрируется шаблон переменной в шаблоне кортежа.

```
let function1 x =
    match x with
    | (var1, var2) when var1 > var2 -> printfn "%d is greater than %d" var1 var2
    | (var1, var2) when var1 < var2 -> printfn "%d is less than %d" var1 var2
    | (var1, var2) -> printfn "%d equals %d" var1 var2

function1 (1,2)
function1 (2, 1)
function1 (0, 0)
```

Шаблон as

Шаблон `as` — это шаблон, к которому добавляется предложение `as`. Предложение `as` привязывает сопоставленное значение к имени, которое может использоваться в выражении выполнения выражения `match`, или, если этот шаблон используется в привязке `let`, имя добавляется в качестве привязки в локальную область.

В следующем примере используется шаблон `as`.

```
let (var1, var2) as tuple1 = (1, 2)
printfn "%d %d %A" var1 var2 tuple1
```

ИЛИ шаблон

Шаблон или используется, если входные данные могут соответствовать нескольким шаблонам и вы хотите выполнить тот же код в результате. Типы обеих сторон шаблона или должны быть совместимы.

В следующем примере демонстрируется шаблон или.

```
let detectZeroOR point =
    match point with
    | (0, 0) | (0, _) | (_, 0) -> printfn "Zero found."
    | _ -> printfn "Both nonzero."
detectZeroOR (0, 0)
detectZeroOR (1, 0)
detectZeroOR (0, 10)
detectZeroOR (10, 15)
```

И шаблон

Шаблон и требует, чтобы входные данные соответствовали двум шаблонам. Типы обеих сторон шаблона и должны быть совместимы.

Следующий пример похож на `detectZeroTuple`, показанный в разделе шаблон кортежа далее в этом разделе, но в данном случае `var1` и `var2` получаются в виде значений с помощью шаблона и.

```
let detectZeroAND point =
    match point with
    | (0, 0) -> printfn "Both values zero."
    | (var1, var2) & (0, _) -> printfn "First value is 0 in (%d, %d)" var1 var2
    | (var1, var2)  & (_, 0) -> printfn "Second value is 0 in (%d, %d)" var1 var2
    | _ -> printfn "Both nonzero."
detectZeroAND (0, 0)
detectZeroAND (1, 0)
detectZeroAND (0, 10)
detectZeroAND (10, 15)
```

Шаблон «против»

Шаблон недостатков используется для разбиения списка на первый элемент, *заголовок*и список, содержащий остальные элементы, *хвост*.

```
let list1 = [ 1; 2; 3; 4 ]

// This example uses a cons pattern and a list pattern.
let rec printList l =
    match l with
    | head :: tail -> printf "%d " head; printList tail
    | [] -> printfn ""

printList list1
```

Шаблон списка

Шаблон списка позволяет разложить списки на несколько элементов. Сам шаблон списка может сопоставлять только списки определенного числа элементов.

```
// This example uses a list pattern.
let listLength list =
    match list with
    | [] -> 0
    | [ _ ] -> 1
    | [ _; _ ] -> 2
    | [ _; _; _ ] -> 3
    | _ -> List.length list

printfn "%d" (listLength [ 1 ])
printfn "%d" (listLength [ 1; 1 ])
printfn "%d" (listLength [ 1; 1; 1; ])
printfn "%d" (listLength [ ] )
```

Шаблон массива

Шаблон массива напоминает шаблон списка и может использоваться для разложения массивов определенной длины.

```
// This example uses array patterns.
let vectorLength vec =
    match vec with
    | [| var1 |] -> var1
    | [| var1; var2 |] -> sqrt (var1*var1 + var2*var2)
    | [| var1; var2; var3 |] -> sqrt (var1*var1 + var2*var2 + var3*var3)
    | _ -> failwith "vectorLength called with an unsupported array size of %d." (vec.Length)

printfn "%f" (vectorLength [| 1. |])
printfn "%f" (vectorLength [| 1.; 1. |])
printfn "%f" (vectorLength [| 1.; 1.; 1.; |])
printfn "%f" (vectorLength [| |] )
```

Шаблон в круглых скобках

Круглые скобки могут быть сгруппированы вокруг шаблонов для достижения требуемой ассоциативности. В следующем примере круглые скобки используются для управления ассоциативностью между шаблоном и и шаблоном недостатка.

```
let countValues list value =
    let rec checkList list acc =
       match list with
       | (elem1 & head) :: tail when elem1 = value -> checkList tail (acc + 1)
       | head :: tail -> checkList tail acc
       | [] -> acc
    checkList list 0

let result = countValues [ for x in -10..10 -> x*x - 4 ] 0
printfn "%d" result
```

Шаблон кортежа

Шаблон кортежа соответствует входным данным в форме кортежа и позволяет разложить кортеж в составные элементы с помощью переменных сопоставления шаблонов для каждой из позиций в кортеже.

В следующем примере демонстрируется шаблон кортежа, а также используются литеральные шаблоны, шаблоны переменных и шаблон с подстановочными знаками.

```
let detectZeroTuple point =
    match point with
    | (0, 0) -> printfn "Both values zero."
    | (0, var2) -> printfn "First value is 0 in (0, %d)" var2
    | (var1, 0) -> printfn "Second value is 0 in (%d, 0)" var1
    | _ -> printfn "Both nonzero."
detectZeroTuple (0, 0)
detectZeroTuple (1, 0)
detectZeroTuple (0, 10)
detectZeroTuple (10, 15)
```

Шаблон записи

Шаблон записи используется для разложения записей, чтобы извлечь значения полей. Шаблон не должен ссылаться на все поля записи; все пропущенные поля просто не участвуют в сопоставлении и не извлекаются.

```
// This example uses a record pattern.

type MyRecord = { Name: string; ID: int }

let IsMatchByName record1 (name: string) =
    match record1 with
    | { MyRecord.Name = nameFound; MyRecord.ID = _; } when nameFound = name -> true
    | _ -> false

let recordX = { Name = "Parker"; ID = 10 }
let isMatched1 = IsMatchByName recordX "Parker"
let isMatched2 = IsMatchByName recordX "Hartono"
```

Шаблон подстановочного знака

Шаблон подстановочного знака представлен символом подчеркивания (`_`) и соответствует любым входным данным, как и шаблону переменной, за исключением того, что входные данные удаляются, а не присваиваются переменной. Шаблон шаблона часто используется в других шаблонах в качестве заполнителя для значений, которые не требуются в выражении справа от символа `->` . Шаблон с подстановочным знаком также часто используется в конце списка шаблонов для сопоставления с любыми несоответствующими входными данными. Шаблон с подстановочными знаками демонстрируется во многих примерах кода в этом разделе. См. Приведенный выше код для одного примера.

Шаблоны с аннотациями типов

Шаблоны могут иметь аннотации типов. Они ведут себя так же, как и другие аннотации типа и пошаговое определение, как и другие заметки типа. Для заметок типа в шаблонах требуются круглые скобки. В следующем коде показан шаблон с аннотацией типа.

```
let detect1 x =
    match x with
    | 1 -> printfn "Found a 1!"
    | (var1 : int) -> printfn "%d" var1
detect1 0
detect1 1
```

Шаблон проверки типа

Шаблон проверки типа используется для сопоставления входных данных с типом. Если тип входных данных соответствует типу (или производному типу) типа, указанного в шаблоне, сопоставление выполняется с ошибкой.

В следующем примере показан шаблон проверки типа.

```
open System.Windows.Forms

let RegisterControl(control:Control) =
    match control with
    | :? Button as button -> button.Text <- "Registered."
    | :? CheckBox as checkbox -> checkbox.Text <- "Registered."
    | _ -> ()
```

Если проверяется только идентификатор определенного производного типа, `as identifier` часть шаблона не требуется, как показано в следующем примере:

```
type A() = class end
type B() = inherit A()
type C() = inherit A()

let m (a: A) =
    match a with
    | :? B -> printfn "It's a B"
    | :? C -> printfn "It's a C"
    | _ -> ()
```

Шаблон NULL

Шаблон NULL соответствует значению NULL, которое может появиться при работе с типами, допускающими значение null. Шаблоны NULL часто используются при взаимодействии с .NET Frameworkным кодом. Например, возвращаемое значение API .NET может быть входными данными для `match` выражения. Можно управлять выполнением программы в зависимости от того, имеет ли возвращаемое значение значение null, а также другие характеристики возвращаемого значения. Можно использовать шаблон NULL, чтобы предотвратить распространение значений NULL на остальную часть программы.

В следующем примере используется шаблон NULL и шаблон переменной.

```
let ReadFromFile (reader : System.IO.StreamReader) =
    match reader.ReadLine() with
    | null -> printfn "\n"; false
    | line -> printfn "%s" line; true

let fs = System.IO.File.Open("..\..\Program.fs", System.IO.FileMode.Open)
let sr = new System.IO.StreamReader(fs)
while ReadFromFile(sr) = true do ()
sr.Close()
```

См. также

- Выражения match
- Активные шаблоны
- Справочник по языку F#

Активные шаблоны

04.11.2019 • 9 minutes to read • Edit Online

Активные шаблоны позволяют определить именованные секции, которые подделят входные данные, чтобы эти имена можно было использовать в выражении сопоставления шаблонов точно так же, как для размеченного объединения. Активные шаблоны можно использовать для разложения данных в настраиваемом порядке для каждого раздела.

Синтаксис

```
// Active pattern of one choice.
let (|identifier|) [arguments] valueToMatch= expression

// Active Pattern with multiple choices.
// Uses a FSharp.Core.Choice<_,...,_> based on the number of case names. In F#, the limitation n <= 7
applies.
let (|identifer1|identifier2|...|) valueToMatch = expression

// Partial active pattern definition.
// Uses a FSharp.Core.option<_> to represent if the type is satisfied at the call site.
let (|identifier|_|) [arguments ] valueToMatch = expression
```

Заметки

В предыдущем синтаксисе идентификаторы представляют собой имена секций входных данных, представленных *аргументами*, или, иными словами, имена подмножеств набора всех значений аргументов. В определении активного шаблона может быть до семи секций. *Выражение* описывает форму, в которую разбиваются данные. Определение активного шаблона можно использовать для определения правил, определяющих, какие из именованных секций являются значениями, заданными в качестве аргументов. Символы (| и |) называются *скрепками в виде полукруглого* символа, а функция, созданная с помощью этого типа привязки let, называется *активным распознавателем*.

В качестве примера рассмотрим следующий активный шаблон с аргументом.

```
let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd
```

Активный шаблон можно использовать в выражении сопоставления шаблонов, как показано в следующем примере.

```
let TestNumber input =
   match input with
   | Even -> printfn "%d is even" input
   | Odd -> printfn "%d is odd" input

TestNumber 7
TestNumber 11
TestNumber 32
```

Выходные данные этой программы выглядят следующим образом:

```
7 is odd
11 is odd
32 is even
```

Другим применением активных шаблонов является разбиение типов данных несколькими способами, например, когда одни и те же базовые данные имеют различные возможные представления. Например, объект `Color` можно разбить на представление RGB или в представление HSB.

```
open System.Drawing

let (|RGB|) (col : System.Drawing.Color) =
     ( col.R, col.G, col.B )

let (|HSB|) (col : System.Drawing.Color) =
   ( col.GetHue(), col.GetSaturation(), col.GetBrightness() )

let printRGB (col: System.Drawing.Color) =
   match col with
   | RGB(r, g, b) -> printfn " Red: %d Green: %d Blue: %d" r g b

let printHSB (col: System.Drawing.Color) =
   match col with
   | HSB(h, s, b) -> printfn " Hue: %f Saturation: %f Brightness: %f" h s b

let printAll col colorString =
  printfn "%s" colorString
  printRGB col
  printHSB col

printAll Color.Red "Red"
printAll Color.Black "Black"
printAll Color.White "White"
printAll Color.Gray "Gray"
printAll Color.BlanchedAlmond "BlanchedAlmond"
```

Выходные данные приведенной выше программы выглядят следующим образом:

```
Red
 Red: 255 Green: 0 Blue: 0
 Hue: 360.000000 Saturation: 1.000000 Brightness: 0.500000
Black
 Red: 0 Green: 0 Blue: 0
 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.000000
White
 Red: 255 Green: 255 Blue: 255
 Hue: 0.000000 Saturation: 0.000000 Brightness: 1.000000
Gray
 Red: 128 Green: 128 Blue: 128
 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.501961
BlanchedAlmond
 Red: 255 Green: 235 Blue: 205
 Hue: 36.000000 Saturation: 1.000000 Brightness: 0.901961
```

В сочетании эти два способа использования активных шаблонов позволяют секционировать и разбивать данные на соответствующую форму и выполнять соответствующие вычисления с соответствующими данными в форме, наиболее удобной для вычислений.

Результирующие выражения сопоставления шаблонов позволяют написать данные удобным для чтения способом, значительно упрощая потенциально сложную ветвление и код анализа данных.

Частичные активные шаблоны

Иногда необходимо секционировать только часть входного пространства. В этом случае вы пишете набор частичных шаблонов, каждый из которых соответствует некоторым входным данным, но не соответствует другим входным данным. Активные шаблоны, которые не всегда создают значение, называются *частичными активными шаблонами*; они имеют возвращаемое значение, которое является типом параметра. Чтобы определить частичный активный шаблон, используйте подстановочный знак (_) в конце списка шаблонов в неменяющих роликах. Следующий код иллюстрирует использование частично активного шаблона.

```
let (|Integer|_|) (str: string) =
   let mutable intvalue = 0
   if System.Int32.TryParse(str, &intvalue) then Some(intvalue)
   else None

let (|Float|_|) (str: string) =
   let mutable floatvalue = 0.0
   if System.Double.TryParse(str, &floatvalue) then Some(floatvalue)
   else None

let parseNumeric str =
   match str with
     | Integer i -> printfn "%d : Integer" i
     | Float f -> printfn "%f : Floating point" f
     | _ -> printfn "%s : Not matched." str

parseNumeric "1.1"
parseNumeric "0"
parseNumeric "0.0"
parseNumeric "10"
parseNumeric "Something else"
```

Выходные данные предыдущего примера выглядят следующим образом:

```
1.100000 : Floating point
0 : Integer
0.000000 : Floating point
10 : Integer
Something else : Not matched.
```

При использовании частичных активных шаблонов иногда отдельные варианты могут быть несоединенными или взаимоисключающими, но они не должны быть. В следующем примере прямоугольный квадрат и куб шаблона не являются несвязанными, так как некоторые числа представляют собой квадраты и Кубы, например 64. Следующая программа использует шаблон и для объединения квадратных и шаблонных шаблонов Куба. Он выводит все целые числа до 1000, которые являются квадратами и кубами, а также только Кубы.

```
let err = 1.e-10

let isNearlyIntegral (x:float) = abs (x - round(x)) < err

let (|Square|_|) (x : int) =
  if isNearlyIntegral (sqrt (float x)) then Some(x)
  else None

let (|Cube|_|) (x : int) =
  if isNearlyIntegral ((float x) ** ( 1.0 / 3.0)) then Some(x)
  else None

let examineNumber x =
   match x with
      | Cube x -> printfn "%d is a cube" x
      | _ -> ()
   match x with
      | Square x -> printfn "%d is a square" x
      | _ -> ()

let findSquareCubes x =
   match x with
       | Cube x & Square _ -> printfn "%d is a cube and a square" x
       | Cube x -> printfn "%d is a cube" x
       | _ -> ()


[ 1 .. 1000 ] |> List.iter (fun elem -> findSquareCubes elem)
```

Выходные данные выглядят следующим образом:

```
1 is a cube and a square
8 is a cube
27 is a cube
64 is a cube and a square
125 is a cube
216 is a cube
343 is a cube
512 is a cube
729 is a cube and a square
1000 is a cube
```

Параметризованные активные шаблоны

Активные шаблоны всегда принимают по крайней мере один аргумент для сопоставляемого элемента, но они могут также принимать дополнительные аргументы, в этом случае применяется к *параметризованному активному шаблону* Name. Дополнительные аргументы позволяют специализированные шаблоны. Например, активные шаблоны, использующие регулярные выражения для анализа строк, часто содержат регулярное выражение в качестве дополнительного параметра, как в следующем коде, который также использует частичный активный шаблон `Integer`, определенный в предыдущем примере кода. В этом примере строки, в которых используются регулярные выражения для различных форматов даты, предоставляются для настройки общего Парсережекс активного шаблона. Целочисленный активный шаблон используется для преобразования совпадающих строк в целые числа, которые могут быть переданы конструктору DateTime.

```
open System.Text.RegularExpressions

// ParseRegex parses a regular expression and returns a list of the strings that match each group in
// the regular expression.
// List.tail is called to eliminate the first element in the list, which is the full matched expression,
// since only the matches for each group are wanted.
let (|ParseRegex|_|) regex str =
   let m = Regex(regex).Match(str)
   if m.Success
   then Some (List.tail [ for x in m.Groups -> x.Value ])
   else None

// Three different date formats are demonstrated here. The first matches two-
// digit dates and the second matches full dates. This code assumes that if a two-digit
// date is provided, it is an abbreviation, not a year in the first century.
let parseDate str =
   match str with
     | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{1,2})$" [Integer m; Integer d; Integer y]
          -> new System.DateTime(y + 2000, m, d)
     | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{3,4})" [Integer m; Integer d; Integer y]
          -> new System.DateTime(y, m, d)
     | ParseRegex "(\d{1,4})-(\d{1,2})-(\d{1,2})" [Integer y; Integer m; Integer d]
          -> new System.DateTime(y, m, d)
     | _ -> new System.DateTime()

let dt1 = parseDate "12/22/08"
let dt2 = parseDate "1/1/2009"
let dt3 = parseDate "2008-1-15"
let dt4 = parseDate "1995-12-28"

printfn "%s %s %s %s" (dt1.ToString()) (dt2.ToString()) (dt3.ToString()) (dt4.ToString())
```

Выходные данные предыдущего кода выглядят следующим образом:

```
12/22/2008 12:00:00 AM 1/1/2009 12:00:00 AM 1/15/2008 12:00:00 AM 12/28/1995 12:00:00 AM
```

Активные шаблоны не ограничиваются только выражениями, соответствующими шаблонам, их также можно использовать в привязке let.

```
let (|Default|) onNone value =
    match value with
    | None -> onNone
    | Some e -> e

let greet (Default "random citizen" name) =
    printfn "Hello, %s!" name

greet None
greet (Some "George")
```

Выходные данные предыдущего кода выглядят следующим образом:

```
Hello, random citizen!
Hello, George!
```

См. также

- Справочник по языку F#
- Выражения match

Циклы. Выражение for...to

27.11.2019 • 2 minutes to read • Edit Online

Выражение `for...to` используется для прохода по циклу над диапазоном значений переменной цикла.

Синтаксис

```
for identifier = start [ to | downto ] finish do
    body-expression
```

Примечания

Тип идентификатора выводится из типа выражений *начала* и *окончания* . Типы для этих выражений должны быть 32-разрядными целыми числами.

Хотя технически это выражение, `for...to` более похоже на традиционный оператор в императивном языке программирования. Тип возвращаемого значения для *выражения тела* должен быть `unit`. В следующих примерах показаны различные варианты использования выражения `for...to`.

```
// A simple for...to loop.
let function1() =
  for i = 1 to 10 do
    printf "%d " i
  printfn ""

// A for...to loop that counts in reverse.
let function2() =
  for i = 10 downto 1 do
    printf "%d " i
  printfn ""

function1()
function2()

// A for...to loop that uses functions as the start and finish expressions.
let beginning x y = x - 2*y
let ending x y = x + 2*y

let function3 x y =
  for i = (beginning x y) to (ending x y) do
     printf "%d " i
  printfn ""

function3 10 4
```

Результат выполнения приведенного кода будет следующим.

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

См. также:

- Справочник по языку F#
- Циклы: выражение `for...in`
- Циклы: выражение `while...do`

Циклы: выражение for...in

23.10.2019 • 5 minutes to read • Edit Online

Эта Циклическая конструкция используется для перебора совпадений шаблона в перечислимой коллекции, такой как выражение диапазона, последовательность, список, массив или другая конструкция, поддерживающая перечисление.

Синтаксис

```
for pattern in enumerable-expression do
    body-expression
```

Примечания

Выражение можно сравнить `for each` с инструкцией в других языках .NET, так как она используется для циклического перебора значений в перечислимой коллекции. `for...in` `for...in` Однако также поддерживает сопоставление шаблонов для коллекции вместо просто итерации по всей коллекции.

Перечислимое выражение можно указать как перечисляемую коллекцию или с помощью `..` оператора в качестве диапазона для целочисленного типа. Перечислимые коллекции включают списки, последовательности, массивы, наборы, карты и т. д. Можно использовать любой тип `System.Collections.IEnumerable` , реализующий реализацию.

При выражении диапазона с помощью `..` оператора можно использовать следующий синтаксис.

Запуск . Финишер

Можно также использовать версию, которая включает инкремент, который называется *Skip*, как показано в следующем коде.

Запуск . пропустить . Финишер

При использовании целочисленных диапазонов и простой переменной счетчика в качестве шаблона типичное поведение заключается в том, чтобы увеличить переменную счетчика на 1 при каждой итерации, но если диапазон включает значение Skip, счетчик увеличивается на значение Skip.

Значения, соответствующие шаблону, можно также использовать в выражении тела.

В следующих примерах кода показано использование `for...in` выражения.

```
// Looping over a list.
let list1 = [ 1; 5; 100; 450; 788 ]
for i in list1 do
   printfn "%d" i
```

Выходные данные выглядят следующим образом.

```
1
5
100
450
788
```

В следующем примере показано, как выполнить цикл по последовательности и как использовать шаблон кортежа вместо простой переменной.

```
let seq1 = seq { for i in 1 .. 10 -> (i, i*i) }
for (a, asqr) in seq1 do
  printfn "%d squared is %d" a asqr
```

Выходные данные выглядят следующим образом.

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

В следующем примере показано, как выполнить цикл над простым целочисленным диапазоном.

```
let function1() =
  for i in 1 .. 10 do
    printf "%d " i
  printfn ""
function1()
```

Выходные данные функция1 выглядят следующим образом.

```
1 2 3 4 5 6 7 8 9 10
```

В следующем примере показано, как выполнить цикл над диапазоном с пропуском 2, который включает все остальные элементы диапазона.

```
let function2() =
  for i in 1 .. 2 .. 10 do
     printf "%d " i
  printfn ""
function2()
```

Выходные данные `function2` имеют следующий вид.

```
1 3 5 7 9
```

В следующем примере показано, как использовать диапазон символов.

```
let function3() =
  for c in 'a' .. 'z' do
    printf "%c " c
  printfn ""
function3()
```

Выходные данные `function3` имеют следующий вид.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

В следующем примере показано, как использовать отрицательное значение Skip для обратных итераций.

```
let function4() =
    for i in 10 .. -1 .. 1 do
        printf "%d " i
    printfn " ... Lift off!"
function4()
```

Выходные данные `function4` имеют следующий вид.

```
10 9 8 7 6 5 4 3 2 1 ... Lift off!
```

Начало и конец диапазона также могут быть выражениями, такими как функции, как в следующем коде.

```
let beginning x y = x - 2*y
let ending x y = x + 2*y

let function5 x y =
  for i in (beginning x y) .. (ending x y) do
     printf "%d " i
  printfn ""

function5 10 4
```

Выходные данные `function5` со следующими входными данными выглядят следующим образом.

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

В следующем примере показано использование подстановочного знака (_), если элемент не требуется в цикле.

```
let mutable count = 0
for _ in list1 do
   count <- count + 1
printfn "Number of elements in list1: %d" count
```

Выходные данные выглядят следующим образом.

```
Number of elements in list1: 5
```

`Note` Можно использовать `for...in` в выражениях последовательности и других вычислительных выражениях. в этом случае используется настроенная `for...in` версия выражения. Дополнительные сведения см. в разделе последовательности, асинхронные рабочие процессыи вычислительные выражения.

См. также

- Справочник по языку F#
- Бираются `for...to` Выражение
- Бираются `while...do` Выражение

Циклы: выражение while...do

23.10.2019 • 2 minutes to read • Edit Online

`while...do` Выражение используется для выполнения итеративного выполнения (цикла), пока заданное условие теста имеет значение true.

Синтаксис

```
while test-expression do
    body-expression
```

Примечания

Тестовое выражение вычисляется; Если это так ,товыполняетсявыражениеbody,атестовоевыражениесновавычисляется`true`. *Выражение текста* должно иметь тип `unit`. Если тестовое выражение имеет `false`значение, итерация завершается.

В следующем примере показано использование `while...do` выражения.

```
open System

let lookForValue value maxValue =
  let mutable continueLooping = true
  let randomNumberGenerator = new Random()
  while continueLooping do
    // Generate a random number between 1 and maxValue.
    let rand = randomNumberGenerator.Next(maxValue)
    printf "%d " rand
    if rand = value then
       printfn "\nFound a %d!" value
       continueLooping <- false

lookForValue 10 20
```

Выходные данные предыдущего кода — это поток случайных чисел от 1 до 20, последний из которых равен 10.

```
13 19 8 18 16 2 10
Found a 10!
```

NOTE

Можно использовать `while...do` в выражениях последовательности и других вычислительных выражениях. в этом случае используется настроенная `while...do` версия выражения. Дополнительные сведения см. в разделе последовательности, асинхронные рабочие процессыи вычислительные выражения.

См. также

- Справочник по языку F#
- Бираются `for...in` Выражение

- Бираются `for...to` Выражение

Утверждения

25.10.2019 • 2 minutes to read • Edit Online

Выражение `assert` — это функция отладки, которую можно использовать для проверки выражения. При сбое в режиме отладки утверждение создает диалоговое окно системной ошибки.

Синтаксис

```
assert condition
```

Заметки

Выражение `assert` имеет тип `bool -> unit`.

Функция `assert` разрешается в Debug.Assert. Это означает, что его поведение аналогично вызову Debug.Assert напрямую.

Проверка утверждения включена только при компиляции в режиме отладки. то есть, если определена константа `DEBUG`. В системе проекта по умолчанию константа `DEBUG` определена в конфигурации отладки, но не в конфигурации выпуска.

Ошибка при сбое утверждения не может быть перехвачена с F# помощью обработки исключений.

Пример

В следующем примере кода показано использование выражения `assert`.

```
let subtractUnsigned (x : uint32) (y : uint32) =
    assert (x > y)
    let z = x - y
    z
// This code does not generate an assertion failure.
let result1 = subtractUnsigned 2u 1u
// This code generates an assertion failure.
let result2 = subtractUnsigned 1u 2u
```

См. также

- Справочник по языку F#

Обработка исключений

04.11.2019 • 2 minutes to read • Edit Online

Эта статья содержит сведения о поддержке обработки исключений в языке F#.

Основы обработки исключений

Обработка исключений является стандартным способом для обработки ошибок в .NET Framework. Поэтому этот механизм должен поддерживаться любым языком .NET, включая F#. *Исключение* — это объект, инкапсулирующий информацию об ошибке. При возникновении ошибки возникают исключения и обычное выполнение останавливается. Вместо этого среда выполнения ищет подходящий обработчик для исключения. Поиск начинается в текущей функции и продолжается вверх по стеку через все уровни вызывающих объектов, пока не будет найден соответствующий обработчик. Затем этот обработчик запускается.

Кроме того, по мере раскрутки стека среда выполнения выполняет весь код в блоках `finally`, обеспечивая правильную очистку объектов во время процесса раскрутки.

См. также

ЗАГОЛОВОК	ОПИСАНИЕ
Типы исключения	Описывает объявление типа исключения.
Исключения: выражение `try...with`	Описывает языковую конструкцию, поддерживающую обработку исключений.
Исключения: выражение `try...finally`	Описывает языковую конструкцию, позволяющую выполнять код очистки по мере раскрутки стека при возникновении исключения.
Исключения: функция `raise`	Описывает активацию объекта исключения.
Исключения: функция `failwith`	Описывает создание общего исключения F#.
Исключения: функция `invalidArg`	Описывает создание исключения недопустимого аргумента.

Типы исключения

23.10.2019 • 2 minutes to read • Edit Online

Существует две категории исключений в F#: типы исключений .NET и F# типы исключений. В этом разделе описывается определение и использование F# типов исключений.

Синтаксис

```
exception exception-type of argument-type
```

Примечания

В предыдущем синтаксисе *Exception-Type* — это имя нового F# типа исключения, а *аргумент-Type* представляет тип аргумента, который может быть указан при вызове исключения этого типа. Можно указать несколько аргументов с помощью типа кортежа для *типа аргумента*.

Типичное определение F# исключения напоминает следующее.

```
exception MyError of string
```

Исключение этого типа можно создать с помощью `raise` функции, как показано ниже.

```
raise (MyError("Error message"))
```

Тип F# исключения можно использовать непосредственно в фильтрах в `try...with` выражении, как показано в следующем примере.

```
exception Error1 of string
// Using a tuple type as the argument type.
exception Error2 of string * int

let function1 x y =
   try
      if x = y then raise (Error1("x"))
      else raise (Error2("x", 10))
   with
      | Error1(str) -> printfn "Error1 %s" str
      | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

Тип исключения, определяемый с помощью `exception` ключевого слова в F# , является новым типом, который наследует от `System.Exception` .

См. также

- Обработка исключений
- Исключения: функция `raise`
- Иерархия исключений

Исключения. Выражение try...with

23.10.2019 • 4 minutes to read • Edit Online

В этом разделе описывается `try...with` выражение, которое используется для обработки исключений в F# языке.

Синтаксис

```
try
    expression1
with
| pattern1 -> expression2
| pattern2 -> expression3
...
```

Примечания

Выражение используется для управления исключениями в F# `try...with` Он аналогичен `try...catch` оператору в C#. В приведенном выше синтаксисе код в *expression1* может вызвать исключение. `try...with` Выражение возвращает значение. Если исключение не создается, выражение целиком возвращает значение *expression1*. При возникновении исключения каждый *шаблон* сравнивается, в свою очередь, с исключением, а для первого совпадающего шаблона соответствующее *выражение*, известное как *обработчик исключений*для этой ветви, выполняется, а общее выражение Возвращает значение выражения в этом обработчике исключений. Если шаблон не соответствует, исключение распространяет стек вызовов до тех пор, пока не будет найден соответствующий обработчик. Типы значений, возвращаемых каждым выражением в обработчиках исключений, должны соответствовать типу, возвращаемому из выражения в `try` блоке.

Как правило, возникновение ошибки также означает, что нет допустимого значения, которое может быть возвращено из выражений в каждом обработчике исключений. Часто шаблоном является то, что тип выражения должен быть типом параметра. Этот шаблон показан в следующем примере кода.

```
let divide1 x y =
   try
      Some (x / y)
   with
      | :? System.DivideByZeroException -> printfn "Division by zero!"; None

let result1 = divide1 100 0
```

Исключения могут быть исключениями .NET, или они могут F# быть исключениями. Исключения можно определить F# с помощью `exception` ключевого слова.

Для фильтрации по типу исключения и другим условиям можно использовать разнообразные шаблоны. параметры приведены в следующей таблице.

ШАБЛОН	ОПИСАНИЕ
:? *тип исключения*	Соответствует указанному типу исключения .NET.

ШАБЛОН	ОПИСАНИЕ
:? *тип Exception* — *идентификатор*	Соответствует указанному типу исключения .NET, но присваивает исключению именованное значение.
имя исключения (*аргументы*)	Соответствует типу F# исключения и привязывает аргументы.
identifier	Соответствует любому исключению и привязывает имя к объекту исключения. Эквивалентно **:? System. Exception в**качестве_идентификатора_
идентификатор при *условии*	Соответствует любому исключению, если условие истинно.

Примеры

В следующих примерах кода показано использование различных шаблонов обработчиков исключений.

```
// This example shows the use of the as keyword to assign a name to a
// .NET exception.
let divide2 x y =
  try
    Some( x / y )
  with
    | :? System.DivideByZeroException as ex -> printfn "Exception! %s " (ex.Message); None

// This version shows the use of a condition to branch to multiple paths
// with the same exception.
let divide3 x y flag =
  try
     x / y
  with
     | ex when flag -> printfn "TRUE: %s" (ex.ToString()); 0
     | ex when not flag -> printfn "FALSE: %s" (ex.ToString()); 1

let result2 = divide3 100 0 true

// This version shows the use of F# exceptions.
exception Error1 of string
exception Error2 of string * int

let function1 x y =
   try
      if x = y then raise (Error1("x"))
      else raise (Error2("x", 10))
   with
      | Error1(str) -> printfn "Error1 %s" str
      | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

NOTE

Конструкция представляет собой отдельное выражение `try...finally` из выражения. `try...with` Поэтому, если в коде требуется `with` блок `finally` и блок, необходимо вложить два выражения.

NOTE

Можно использовать `try...with` в асинхронных рабочих процессах и других вычислительных выражениях. в этом случае используется `try...with` настроенная версия выражения. Дополнительные сведения см. в разделе асинхронные рабочие процессыи вычислительные выражения.

См. также

- Обработка исключений
- Типы исключения
- Исключения. `try...finally` Выражение

Исключения. Выражение try...finally

23.10.2019 • 2 minutes to read • Edit Online

`try...finally` Выражение позволяет выполнять код очистки, даже если блок кода создает исключение.

Синтаксис

```
try
    expression1
finally
    expression2
```

Примечания

Выражение можно использовать для выполнения кода в *Expression2* в предыдущем синтаксисе независимо от того, создано ли исключение во время выполнения *expression1*. `try...finally`

Тип *Expression2* не влияет на значение всего выражения; тип, возвращаемый, если исключение не возникает, является последним значением в *expression1*. При возникновении исключения значение не возвращается, а поток управления передается в следующий обработчик исключений, находящийся выше по стеку вызовов. Если обработчик исключений не найден, программа завершает работу. Перед выполнением кода в обработчике сопоставления или завершения программы выполняется код в `finally` ветви.

В следующем коде показано использование `try...finally` выражения.

```
let divide x y =
   let stream : System.IO.FileStream = System.IO.File.Create("test.txt")
   let writer : System.IO.StreamWriter = new System.IO.StreamWriter(stream)
   try
      writer.WriteLine("test1");
      Some( x / y )
   finally
      writer.Flush()
      printfn "Closing stream"
      stream.Close()

let result =
  try
     divide 100 0
  with
     | :? System.DivideByZeroException -> printfn "Exception handled."; None
```

На консоль выводятся следующие выходные данные.

```
Closing stream
Exception handled.
```

Как видно из выходных данных, поток был закрыт до обработки внешнего исключения, а файл `test.txt` содержит текст `test1`, указывающий на то, что буферы были сброшены и записаны на диск, даже если исключение передано. Управление внешним обработчиком исключений.

Обратите внимание `try...with` , что конструкция является отдельной конструкцией `try...finally` из

конструкции. Поэтому, если в коде требуется `with` блок `finally` и блок, необходимо вложить две конструкции, как показано в следующем примере кода.

```
exception InnerError of string
exception OuterError of string

let function1 x y =
   try
     try
        if x = y then raise (InnerError("inner"))
        else raise (OuterError("outer"))
     with
      | InnerError(str) -> printfn "Error1 %s" str
   finally
      printfn "Always print this."


let function2 x y =
  try
     function1 x y
  with
     | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

В контексте вычислительных выражений, включая выражения последовательности и асинхронные рабочие процессы, **попробуйте... выражения finally** могут иметь пользовательскую реализацию. Дополнительные сведения см. в разделе выражения вычислений.

См. также

- Обработка исключений
- Исключения. `try...with` Выражение

Исключения: функция raise

23.10.2019 • 2 minutes to read • Edit Online

`raise` Функция используется для указания на то, что возникла ошибка или исключительная ситуация. Сведения об ошибке захватываются в объекте исключения.

Синтаксис

```
raise (expression)
```

Примечания

`raise` Функция создает объект исключения и инициирует процесс очистки стека. Процесс очистки стека управляется средой CLR, поэтому поведение этого процесса аналогично тому, что находится на любом другом языке .NET. Процесс очистки стека — это поиск обработчика исключений, соответствующего созданному исключению. Поиск начинается в текущем `try...with` выражении, если таковое имеется. Каждый шаблон в `with` блоке проверяется по порядку. При обнаружении соответствующего обработчика исключений исключение считается обработанным. в противном случае стек будет снят и `with` блокирует цепочку вызовов, пока не будет найден соответствующий обработчик. Все `finally` блоки, обнаруженные в цепочке вызовов, также выполняются последовательно в виде очистки стека.

Функция эквивалентна C# в или C++. `throw` `raise` Используйте `reraise` в обработчике Catch для распространения одного и того же исключения вверх по цепочке вызовов.

В следующих примерах кода показано использование `raise` функции для создания исключения.

```
exception InnerError of string
exception OuterError of string

let function1 x y =
   try
     try
        if x = y then raise (InnerError("inner"))
        else raise (OuterError("outer"))
     with
      | InnerError(str) -> printfn "Error1 %s" str
   finally
      printfn "Always print this."


let function2 x y =
  try
     function1 x y
  with
     | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

`raise` Функцию также можно использовать для вызова исключений .NET, как показано в следующем примере.

```
let divide x y =
  if (y = 0) then raise (System.ArgumentException("Divisor cannot be zero!"))
  else
     x / y
```

См. также

- Обработка исключений
- Типы исключения
- Исключения. `try...with` Выражение
- Исключения. `try...finally` Выражение
- Исключения. `failwith` Функция
- Исключения. `invalidArg` Функция

Исключения. Функция failwith

23.10.2019 • 2 minutes to read • Edit Online

`failwith` Функция создает F# исключение.

Синтаксис

```
failwith error-message-string
```

Примечания

Строка Error-Message- в предыдущем синтаксисе представляет собой литеральную строку или значение типа `string`. Он станет `Message` свойством исключения.

Исключение, формируемое `failwith` , `System.Exception` — это исключение, которое представляет собой ссылку с именем `Failure` в F# коде. Следующий код иллюстрирует использование `failwith` для создания исключения.

```
let divideFailwith x y =
  if (y = 0) then failwith "Divisor cannot be zero."
  else
    x / y

let testDivideFailwith x y =
  try
     divideFailwith x y
  with
     | Failure(msg) -> printfn "%s" msg; 0

let result1 = testDivideFailwith 100 0
```

См. также

- Обработка исключений
- Типы исключения
- Исключения. `try...with` Выражение
- Исключения. `try...finally` Выражение
- Исключения: функция `raise`

Исключения. Функция invalidArg

23.10.2019 • 2 minutes to read • Edit Online

`invalidArg` Функция создает исключение аргумента.

Синтаксис

```
invalidArg parameter-name error-message-string
```

Примечания

Параметр-Name в предыдущем синтаксисе представляет собой строку с именем параметра, аргумент которого был недопустимым. *Строка сообщения Error-Message-* представляет собой литеральную строку или значение типа `string`. Он станет `Message` свойством объекта исключения.

Исключение, созданное `invalidArg` , `System.ArgumentException` является исключением. Следующий код иллюстрирует использование `invalidArg` для создания исключения.

```
let months = [| "January"; "February"; "March"; "April";
                "May"; "June"; "July"; "August"; "September";
                "October"; "November"; "December" |]

let lookupMonth month =
   if (month > 12 || month < 1)
     then invalidArg "month" (sprintf "Value passed in was %d." month)
   months.[month - 1]

printfn "%s" (lookupMonth 12)
printfn "%s" (lookupMonth 1)
printfn "%s" (lookupMonth 13)
```

Ниже приведены выходные данные, за которыми следует трассировка стека (не показана).

```
December
January
System.ArgumentException: Month parameter out of range.
```

См. также

- Обработка исключений
- Типы исключения
- Исключения. `try...with` Выражение
- Исключения. `try...finally` Выражение
- Исключения: функция `raise`
- Исключения. `failwith` Функция

Атрибуты

04.11.2019 • 6 minutes to read • Edit Online

Атрибуты позволяют применять метаданные к конструкции программирования.

Синтаксис

```
[<target:attribute-name(arguments)>]
```

Заметки

В предыдущем синтаксисе *целевой объект* является необязательным и, если он есть, указывает тип сущности программы, к которой применяется атрибут. Допустимые значения для *Target* показаны в таблице ниже в этом документе.

Имя атрибута — это имя (возможно, дополненное пространствами имен) допустимого типа атрибута с суффиксом или без него `Attribute`, который обычно используется в именах типов атрибутов. Например, тип `ObsoleteAttribute` можно сократить, чтобы просто `Obsolete` в этом контексте.

Аргументы являются аргументами конструктора для типа атрибута. Если атрибут имеет конструктор без параметров, список аргументов и круглые скобки можно опустить. Атрибуты поддерживают как аргументы, так и именованные аргументы. *Позиционированные аргументы* — это аргументы, которые используются в том порядке, в котором они отображаются. Именованные аргументы можно использовать, если атрибут имеет открытые свойства. Эти значения можно задать с помощью следующего синтаксиса в списке аргументов.

```
property-name = property-value
```

Такие инициализации свойств могут быть в любом порядке, но они должны следовать любым запозиционированным аргументам. Ниже приведен пример атрибута, в котором используются позиционированные аргументы и инициализации свойств.

```
open System.Runtime.InteropServices

[<DllImport("kernel32", SetLastError=true)>]
extern bool CloseHandle(nativeint handle)
```

В этом примере атрибут имеет значение `DllImportAttribute`, который используется в сокращенной форме. Первый аргумент — это параметр с позиционированием, а второй — свойством.

Атрибуты — это конструкция программирования .NET, которая позволяет объекту, известному как *атрибут* , быть связанным с типом или другим программным элементом. Элемент Program, к которому применяется атрибут, называется *целевым объектом атрибута*. Атрибут обычно содержит метаданные о своем целевом объекте. В этом контексте метаданные могут быть любыми данными о типе, отличном от его полей и членов.

Атрибуты в F# могут применяться к следующим конструкциям программирования: функции, методы, сборки, модули, типы (классы, записи, структуры, интерфейсы, делегаты, перечисления, объединения и т. д.), конструкторы, свойства, поля, параметры, параметры типов и возвращаемые значения. Атрибуты не

допускаются в привязках `let` в классах, выражениях или выражениях рабочих процессов.

Как правило, объявление атрибута отображается непосредственно перед объявлением целевого объекта атрибута. Несколько объявлений атрибутов можно использовать вместе следующим образом:

```
[<Owner("Jason Carlson")>]
[<Company("Microsoft")>]
type SomeType1 =
```

Вы можете запрашивать атрибуты во время выполнения с помощью отражения .NET.

Можно объявить несколько атрибутов по отдельности, как в предыдущем примере кода, или можно объявить их в одном наборе квадратных скобок, если использовать точку с запятой для разделения отдельных атрибутов и конструкторов следующим образом:

```
[<Owner("Darren Parker"); Company("Microsoft")>]
type SomeType2 =
```

Обычно такие атрибуты включают атрибут `Obsolete`, атрибуты для обеспечения безопасности, атрибуты для поддержки COM, атрибуты, относящиеся к владению кодом, и атрибуты, указывающие, можно ли сериализовать тип. В следующем примере демонстрируется использование атрибута `Obsolete`.

```
open System

[<Obsolete("Do not use. Use newFunction instead.")>]
let obsoleteFunction x y =
  x + y

let newFunction x y =
  x + 2 * y

// The use of the obsolete function produces a warning.
let result1 = obsoleteFunction 10 100
let result2 = newFunction 10 100
```

Для целевых объектов атрибута `assembly` и `module` атрибуты применяются к привязке `do` верхнего уровня в сборке. В объявление атрибута можно включить слово `assembly` или `module` следующим образом:

```
open System.Reflection
[<assembly:AssemblyVersionAttribute("1.0.0.0")>]
do
   printfn "Executing..."
```

Если опустить атрибут Target для атрибута, применяемого к привязке `do`, F# компилятор пытается определить целевой объект атрибута, который имеет смысл для этого атрибута. Многие классы атрибутов имеют атрибут типа `System.AttributeUsageAttribute`, который содержит сведения о возможных целевых объектах, поддерживаемых для этого атрибута. Если `System.AttributeUsageAttribute` указывает, что атрибут поддерживает функции в качестве целевых объектов, то атрибут применяется к главной точке входа программы. Если `System.AttributeUsageAttribute` указывает, что атрибут поддерживает сборки в качестве целевых объектов, компилятор принимает атрибут для применения к сборке. Большинство атрибутов не применяются к функциям и сборкам, но в тех случаях, когда они делают, применяется атрибут для применения к функции Main программы. Если целевой объект атрибута указан явно, атрибут применяется к указанному целевому объекту.

Хотя обычно не требуется явно указывать целевой объект атрибута, допустимые значения для *Target* в атрибуте вместе с примерами использования показаны в следующей таблице.

ЦЕЛЕВОЙ ОБЪЕКТ АТРИБУТА	ПРИМЕР
сборка	`[<assembly: AssemblyVersionAttribute("1.0.0.0")>]`
return	`let function1 x : [<return: Obsolete>] int = x + 1`
поле	`[<field: DefaultValue>] val mutable x: int`
свойство;	`[<property: Obsolete>] this.MyProperty = x`
param	`member this.MyMethod([<param: Out>] x : ref<int>) = x := 10`
тип	`[<type: StructLayout(Sequential)>]` `type MyStruct =` `struct` `x : byte` `y : int` `end`

См. также

- Справочник по языку F#

Управление ресурсами: Ключевое слово use

23.10.2019 • 5 minutes to read • Edit Online

В этом разделе описывается ключевое `use` слово `using` и функция, которые могут управлять инициализацией и освобождением ресурсов.

Ресурсы

Термин *ресурс* используется более чем одним способом. Да, ресурсы могут быть данными, используемыми приложением, например строками, графикой и т. д., но в этом контексте *ресурсы* относятся к программному обеспечению или ресурсам операционной системы, таким как контексты графических устройств, дескрипторы файлов, сеть и база данных соединения, объекты параллелизма, такие как дескрипторы ожидания и т. д. Использование этих ресурсов приложениями предполагает приобретение ресурса из операционной системы или другого поставщика ресурсов, за которым следует более поздний выпуск ресурса в пул, чтобы его можно было предоставить другому приложению. Проблемы возникают, когда приложения не освобождают ресурсы в общий пул.

Управление ресурсами

Чтобы эффективно и ответственно управлять ресурсами в приложении, необходимо быстро освобождать ресурсы и предсказуемым образом. .NET Framework помогает сделать это, предоставляя `System.IDisposable` интерфейс. Тип, реализующий `System.IDisposable` , `System.IDisposable.Dispose` имеет метод, который правильно освобождает ресурсы. Хорошо написанные приложения гарантируют `System.IDisposable.Dispose` , что он вызывается немедленно, когда любой объект, содержащий ограниченный ресурс, больше не нужен. К счастью, большинство языков .NET предоставляют поддержку для упрощения и F# не являются исключением. Существует две полезные языковые конструкции, поддерживающие шаблон удаления: `use` привязка `using` и функция.

использовать привязку

Ключевое слово имеет форму, похожую на эту `let` привязку: `use`

использовать*выражение значения* =

Он предоставляет те же функциональные возможности, `let` что и привязка, но добавляет `Dispose` вызов к значению, когда значение выходит за пределы области. Обратите внимание, что компилятор вставляет проверку значения на null, поэтому, если значение равно `null` , `Dispose` вызов метода не выполняется.

В следующем примере показано, как автоматически закрыть файл с помощью `use` ключевого слова.

```
open System.IO

let writetofile filename obj =
   use file1 = File.CreateText(filename)
   file1.WriteLine("{0}", obj.ToString() )
   // file1.Dispose() is called implicitly here.

writetofile "abc.txt" "Humpty Dumpty sat on a wall."
```

> **NOTE**
> Можно использовать `use` в вычислительных выражениях, в этом случае используется настроенная версия `use` выражения. Дополнительные сведения см. в разделе последовательности, асинхронные рабочие процессыи вычислительные выражения.

Использование функции

`using` Функция имеет следующую форму:

`using` (*expression1*) *Function-или-лямбда*

В выражении expression1 создает объект, который должен быть удален. `using` Результат *expression1* (объект, который должен быть удален) становится аргументом, *значением*, *функцией или лямбда*-выражением, представляющим собой либо функцию, которая ожидает один оставшийся аргумент типа, совпадающий со значением, полученным *expression1*или лямбда-выражение, которое принимает аргумент этого типа. В конце выполнения функции среда выполнения вызывает `Dispose` и освобождает ресурсы (если значение не равно `null`), в этом случае попытка вызова Dispose не выполняется.

В следующем примере показано `using` выражение с лямбда-выражением.

```
open System.IO

let writetofile2 filename obj =
    using (System.IO.File.CreateText(filename)) ( fun file1 ->
        file1.WriteLine("{0}", obj.ToString() )
    )

writetofile2 "abc2.txt" "The quick sly fox jumps over the lazy brown dog."
```

В следующем примере показано `using` выражение с функцией.

```
let printToFile (file1 : System.IO.StreamWriter) =
    file1.WriteLine("Test output");

using (System.IO.File.CreateText("test.txt")) printToFile
```

Обратите внимание, что функция может быть функцией, к которой уже применены некоторые аргументы. Это действие представлено в следующем примере кода: Он создает файл, содержащий строку `XYZ`.

```
let printToFile2 obj (file1 : System.IO.StreamWriter) =
    file1.WriteLine(obj.ToString())

using (System.IO.File.CreateText("test.txt")) (printToFile2 "XYZ")
```

`using` Функция`use` и привязка являются практически эквивалентными способами выполнения одной и той же задачи. Ключевое слово обеспечивает больший контроль над вызовом метода when `Dispose`. `using` `using` При использовании `Dispose` метод вызывается в конце функции или лямбда-выражения `use` ; при использовании ключевого слова `Dispose` метод вызывается в конце содержащего его блока кода. Как правило, предпочтительнее использовать `use` вместо `using` функции.

См. также

- Справочник по языку F#

Пространства имен

04.11.2019 • 7 minutes to read • Edit Online

Пространство имен позволяет организовать код в области связанных функций, позволяя присоединить имя к группированию элементов F# программы. Пространства имен обычно являются элементами верхнего уровня в F# файлах.

Синтаксис

```
namespace [rec] [parent-namespaces.]identifier
```

Заметки

Если вы хотите поместить код в пространство имен, первое объявление в файле должно объявлять пространство имен. Содержимое всего файла затем становится частью пространства имен, если другое объявление пространств имен не существует в файле. В этом случае весь код наследуется до тех пор, пока следующее объявление пространства имен не станет частью первого пространства имен.

Пространства имен не могут содержать непосредственно значения и функции. Вместо этого значения и функции должны включаться в модули, а модули включаются в пространства имен. Пространства имен могут содержать типы, модули.

Комментарии XML doc можно объявить над пространством имен, но они не учитываются. Директивы компилятора также можно объявить над пространством имен.

Пространства имен могут быть объявлены явно с помощью ключевого слова namespace или неявно при объявлении модуля. Чтобы объявить пространство имен явным образом, используйте ключевое слово namespace, за которым следует имя пространства имен. В следующем примере показан файл кода, в котором объявляется пространство имен `Widgets` с типом и модулем, входящим в это пространство имен.

```
namespace Widgets

type MyWidget1 =
    member this.WidgetName = "Widget1"

module WidgetsModule =
    let widgetName = "Widget2"
```

Если все содержимое файла находится в одном модуле, можно также объявить пространства имен неявно, используя ключевое слово `module` и указав новое имя пространства имен в полном имени модуля. В следующем примере показан файл кода, в котором объявляется пространство имен `Widgets` и модуль `WidgetsModule`, который содержит функцию.

```
module Widgets.WidgetModule

let widgetFunction x y =
   printfn "%A %A" x y
```

Следующий код эквивалентен предыдущему коду, но модуль является объявлением локального модуля. В этом случае пространство имен должно располагаться в отдельной строке.

```
namespace Widgets

module WidgetModule =

    let widgetFunction x y =
        printfn "%A %A" x y
```

Если в одном файле или нескольких пространствах имен требуется несколько модулей, необходимо использовать объявления локальных модулей. При использовании объявлений локального модуля нельзя использовать полное пространство имен в объявлениях модуля. В следующем коде показан файл с объявлением пространства имен и двумя объявлениями локального модуля. В этом случае модули содержатся непосредственно в пространстве имен. не существует неявно созданного модуля, имя которого совпадает с именем файла. Любой другой код в файле, например привязка `do`, находится в пространстве имен, но не во внутренних модулях, поэтому необходимо уточнить члена модуля `widgetFunction` с помощью имени модуля.

```
namespace Widgets

module WidgetModule1 =
   let widgetFunction x y =
      printfn "Module1 %A %A" x y
module WidgetModule2 =
   let widgetFunction x y =
      printfn "Module2 %A %A" x y

module useWidgets =

  do
     WidgetModule1.widgetFunction 10 20
     WidgetModule2.widgetFunction 5 6
```

Выходные данные этого примера выглядят следующим образом.

```
Module1 10 20
Module2 5 6
```

Дополнительные сведения см. в разделе модули.

Вложенные пространства имен

При создании вложенного пространства имен необходимо полностью уточнить его. В противном случае создается новое пространство имен верхнего уровня. Отступ не учитывается в объявлениях пространств имен.

В следующем примере показано, как объявить вложенное пространство имен.

```
namespace Outer

    // Full name: Outer.MyClass
    type MyClass() =
       member this.X(x) = x + 1

// Fully qualify any nested namespaces.
namespace Outer.Inner

    // Full name: Outer.Inner.MyClass
    type MyClass() =
       member this.Prop1 = "X"
```

Пространства имен в файлах и сборках

Пространства имен могут охватывать несколько файлов в одном проекте или компиляции. Фрагмент «термин *пространства имен* » описывает часть пространства имен, включенную в один файл. Пространства имен могут также охватывать несколько сборок. Например, пространство имен `System` включает в себя весь .NET Framework, который охватывает множество сборок и содержит множество вложенных пространств имен.

Глобальное пространство имен

Для размещения имен в пространстве имен .NET верхнего уровня используется предопределенное `global` пространства имен.

```
namespace global

type SomeType() =
    member this.SomeMember = 0
```

Можно также использовать Global для ссылки на пространство имен .NET верхнего уровня, например, чтобы разрешить конфликты имен с другими пространствами имен.

```
global.System.Console.WriteLine("Hello World!")
```

Рекурсивные пространства имен

Пространства имен также можно объявить как рекурсивные, чтобы обеспечить взаимную рекурсивную рекурсию для всего автономного кода. Это выполняется с помощью `namespace rec`. Использование `namespace rec` может сократить некоторые трудности, не позволяя писать взаимно ссылочный код между типами и модулями. Ниже приведен пример.

```
namespace rec MutualReferences

type Orientation = Up | Down
type PeelState = Peeled | Unpeeled

// This exception depends on the type below.
exception DontSqueezeTheBananaException of Banana

type BananaPeel() = class end

type Banana(orientation : Orientation) =
    member val IsPeeled = false with get, set
    member val Orientation = orientation with get, set
    member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled] with get, set

    member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
    member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends on the
exception above.

module BananaHelpers =
    let peel (b: Banana) =
        let flip (banana: Banana) =
            match banana.Orientation with
            | Up ->
                banana.Orientation <- Down
                banana
            | Down -> banana

        let peelSides (banana: Banana) =
            banana.Sides
            |> List.map (function
                         | Unpeeled -> Peeled
                         | Peeled -> Peeled)

        match b.Orientation with
        | Up ->   b |> flip |> peelSides
        | Down -> b |> peelSides
```

Обратите внимание, что исключение `DontSqueezeTheBananaException` и класс `Banana` оба ссылаются друг на друга. Кроме того, модуль `BananaHelpers` и класс `Banana` также ссылаются друг на друга. Это было бы невозможно выразить в F# случае удаления ключевого слова `rec` из пространства имен `MutualReferences`.

Эта функция также доступна для модулейверхнего уровня.

См. также

- Справочник по языку F#
- Модули
- F#RFC FS-1009 — разрешить взаимно ссылочные типы и модули в более крупных областях внутри файлов

Модули

04.11.2019 • 10 minutes to read • Edit Online

В контексте F# языка *модуль* — это группа F# кода, например значения, типы и значения функций, в F# программе. Код группирования в модулях объединяет связанный код и помогает избежать конфликтов имен в программе.

Синтаксис

```
// Top-level module declaration.
module [accessibility-modifier] [qualified-namespace.]module-name
declarations
// Local module declaration.
module [accessibility-modifier] module-name =
    declarations
```

Заметки

F# Модуль — это группа конструкций F# кода, таких как типы, значения, значения функций и код в привязках `do`. Он реализуется как класс среды CLR, имеющий только статические члены. Существует два типа объявлений модулей в зависимости от того, включен ли в модуль весь файл: объявление модуля верхнего уровня и объявление локального модуля. Объявление модуля верхнего уровня включает в себя весь файл в модуле. Объявление модуля верхнего уровня может использоваться только в качестве первого объявления в файле.

В синтаксисе объявления модуля верхнего уровня необязательное *полное-пространство имен* — это последовательность вложенных имен пространств имен, содержащих модуль. Полное имя пространства имен не обязательно должно быть объявлено ранее.

Добавлять отступы в модуль верхнего уровня не требуется. Необходимо задать отступ для всех объявлений в локальных модулях. В объявлении локального модуля только объявления, имеющие отступ под этим объявлением модуля, являются частью модуля.

Если файл кода не начинается с объявления модуля верхнего уровня или объявления пространства имен, все содержимое файла, включая все локальные модули, станет частью неявно созданного модуля верхнего уровня, который имеет то же имя, что и файл, без расширения. с первой буквы, преобразованной в верхний регистр. Например, рассмотрим следующий файл.

```
// In the file program.fs.
let x = 40
```

Этот файл будет скомпилирован так, как если бы он был написан таким образом:

```
module Program
let x = 40
```

Если в файле имеется несколько модулей, необходимо использовать локальное объявление модуля для каждого модуля. Если объявлено включающее пространство имен, эти модули являются частью включающего пространства имен. Если включающее пространство имен не объявлено, модули становятся частью неявно созданного модуля верхнего уровня. В следующем примере кода показан файл кода,

содержащий несколько модулей. Компилятор неявно создает модуль верхнего уровня с именем `Multiplemodules`, а `MyModule1` и `MyModule2` вложены в этот модуль верхнего уровня.

```
// In the file multiplemodules.fs.
// MyModule1
module MyModule1 =
    // Indent all program elements within modules that are declared with an equal sign.
    let module1Value = 100

    let module1Function x =
        x + 10

// MyModule2
module MyModule2 =

    let module2Value = 121

    // Use a qualified name to access the function.
    // from MyModule1.
    let module2Function x =
        x * (MyModule1.module1Function module2Value)
```

При наличии нескольких файлов в проекте или в одной компиляции или при построении библиотеки необходимо включить объявление пространства имен или объявление модуля в начало файла. F# Компилятор явно определяет имя модуля только в том случае, если в проекте или в командной строке компиляции имеется только один файл, и вы создаете приложение.

Модификатор доступности может быть одним из следующих: `public`, `private`, `internal`. Дополнительные сведения см. в статье Управление доступом. Значение по умолчанию — "public" (открытый).

Ссылка на код в модулях

При ссылке на функции, типы и значения из другого модуля необходимо либо использовать полное имя, либо открыть модуль. При использовании полного имени необходимо указать пространства имен, модуль и идентификатор нужного элемента программы. Каждая часть полного пути разделяются точкой (.), как показано ниже.

```
Namespace1.Namespace2.ModuleName.Identifier
```

Чтобы упростить код, можно открыть модуль или одно или несколько пространств имен. Дополнительные сведения об открытии пространств имен и модулей см. в разделе Объявления импорта: ключевое слово `open`.

В следующем примере кода показан модуль верхнего уровня, содержащий весь код до конца файла.

```
module Arithmetic

let add x y =
    x + y

let sub x y =
    x - y
```

Чтобы использовать этот код из другого файла в том же проекте, необходимо либо использовать полные имена, либо открыть модуль перед использованием функций, как показано в следующих примерах.

```
// Fully qualify the function name.
let result1 = Arithmetic.add 5 9
// Open the module.
open Arithmetic
let result2 = add 5 9
```

Вложенные модули

Модули могут быть вложенными. Внутренние модули должны быть с отступом до объявлений внешних модулей, чтобы указать, что они являются внутренними модулями, а не новыми модулями. Например, Сравните следующие два примера. Модуль `Z` является внутренним модулем в следующем коде.

```
module Y =
    let x = 1

    module Z =
        let z = 5
```

Но `Z` модуля — это одноуровневый `Y` модуля в следующем коде.

```
module Y =
    let x = 1

module Z =
    let z = 5
```

Модуль `Z` также является родственным модулем в следующем коде, так как он не имеет отступов, пока другие объявления в модуле `Y`.

```
module Y =
        let x = 1

    module Z =
        let z = 5
```

Наконец, если внешний модуль не имеет объявлений и за ним сразу же поступает объявление другого модуля, объявление нового модуля считается внутренним модулем, но компилятор выдаст предупреждение, если второе определение модуля не имеет отступа дальше, чем началь.

```
// This code produces a warning, but treats Z as a inner module.
module Y =
module Z =
    let z = 5
```

Чтобы устранить это предупреждение, помещает внутренний модуль в отступ.

```
module Y =
    module Z =
        let z = 5
```

Если требуется, чтобы весь код в файле настроился в одном внешнем модуле и вы хотели использовать внутренние модули, внешний модуль не требует знака равенства, а объявления, включая любые внутренние объявления модулей, которые будут находиться во внешнем модуле, не должны иметь отступов. Объявления внутри внутренних объявлений модуля должны иметь отступы. Этот случай

показан в следующем коде.

```
// The top-level module declaration can be omitted if the file is named
// TopLevel.fs or topLevel.fs, and the file is the only file in an
// application.
module TopLevel

let topLevelX = 5

module Inner1 =
    let inner1X = 1
module Inner2 =
    let inner2X = 5
```

Рекурсивные модули

F#в 4,1 введено понятие модулей, которые позволяют взаимно рекурсивно выполнять весь автономный код. Это выполняется с помощью `module rec`. Использование `module rec` может сократить некоторые трудности, не позволяя писать взаимно ссылочный код между типами и модулями. Ниже приведен пример.

```
module rec RecursiveModule =
    type Orientation = Up | Down
    type PeelState = Peeled | Unpeeled

    // This exception depends on the type below.
    exception DontSqueezeTheBananaException of Banana

    type BananaPeel() = class end

    type Banana(orientation : Orientation) =
        member val IsPeeled = false with get, set
        member val Orientation = orientation with get, set
        member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled] with get, set

        member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
        member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends
on the exception above.

    module BananaHelpers =
        let peel (b: Banana) =
            let flip (banana: Banana) =
                match banana.Orientation with
                | Up ->
                    banana.Orientation <- Down
                    banana
                | Down -> banana

            let peelSides (banana: Banana) =
                banana.Sides
                |> List.map (function
                             | Unpeeled -> Peeled
                             | Peeled -> Peeled)

            match b.Orientation with
            | Up ->   b |> flip |> peelSides
            | Down -> b |> peelSides
```

Обратите внимание, что исключение `DontSqueezeTheBananaException` и класс `Banana` оба ссылаются друг на друга. Кроме того, модуль `BananaHelpers` и класс `Banana` также ссылаются друг на друга. Это невозможно, F# если вы удалили ключевое слово `rec` из модуля `RecursiveModule`.

Эта возможность также возможна в пространствах имен с F# 4,1.

См. также

- Справочник по языку F#
- Пространства имен
- F#RFC FS-1009 — разрешить взаимно ссылочные типы и модули в более крупных областях внутри файлов

Объявления импорта. Ключевое слово `open`

23.10.2019 • 5 minutes to read • Edit Online

> **NOTE**
> Ссылки на справочник по API в этой статье ведут на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

В *объявлении импорта* указан модуль или пространство имен, элементы которого можно ссылаться без использования полного имени.

Синтаксис

```
open module-or-namespace-name
```

Примечания

Создание ссылки на код с помощью полного пространства имен или пути к модулю каждый раз может создать код, который трудно писать, читать и обслуживать. Вместо этого можно использовать `open` ключевое слово для часто используемых модулей и пространств имен, чтобы при ссылке на член этого модуля или пространства имен можно было использовать краткую форму имени вместо полного имени. Это ключевое слово похоже на `using` ключевое C#слово `using namespace` в, C++в Visual `Imports` и в Visual Basic.

Предоставленный модуль или пространство имен должен находиться в том же проекте или в проекте или сборке, на которую имеется ссылка. Если это не так, можно добавить ссылку на проект или использовать `-reference` параметр командной `-` строки `-r` (или его сокращение). Дополнительные сведения см. в разделе Параметры компилятора.

Объявление импорта делает имена доступными в коде, который следует за объявлением, вплоть до конца включающего пространства имен, модуля или файла.

При использовании нескольких объявлений импорта они должны отображаться в разных строках.

В следующем коде показано использование `open` ключевого слова для упрощения кода.

```
// Without the import declaration, you must include the full
// path to .NET Framework namespaces such as System.IO.
let writeToFile1 filename (text: string) =
  let stream1 = new System.IO.FileStream(filename, System.IO.FileMode.Create)
  let writer = new System.IO.StreamWriter(stream1)
  writer.WriteLine(text)

// Open a .NET Framework namespace.
open System.IO

// Now you do not have to include the full paths.
let writeToFile2 filename (text: string) =
  let stream1 = new FileStream(filename, FileMode.Create)
  let writer = new StreamWriter(stream1)
  writer.WriteLine(text)

writeToFile2 "file1.txt" "Testing..."
```

F# Компилятор не выдает ошибку или предупреждение при возникновении неоднозначности, если одно и то же имя встречается в нескольких открытых модулях или пространствах имен. При возникновении неоднозначности F# дает предпочтение более последнему открытому модулю или пространству имен. Например, в следующем коде `empty` это означает, что, несмотря `empty` на то, что `Seq.empty` находится в `List` обоих `Seq` модулях и.

```
open List
open Seq
printfn "%A" empty
```

Поэтому будьте внимательны при открытии модулей или пространств имен, таких как `List` или `Seq` , которые содержат элементы с одинаковыми именами. вместо этого рекомендуется использовать полные имена. Следует избегать ситуаций, в которых код зависит от порядка объявлений импорта.

Пространства имен, открытые по умолчанию

Некоторые пространства имен часто используются в F# коде, который они открывают неявно, без необходимости явного объявления импорта. В следующей таблице показаны пространства имен, открытые по умолчанию.

ПРОСТРАНСТВО ИМЕН	ОПИСАНИЕ
`Microsoft.FSharp.Core`	Содержит базовые F# определения типов для встроенных типов, таких как `int` и `float` .
`Microsoft.FSharp.Core.Operators`	Содержит базовые арифметические операции, `+` такие `*` как и.
`Microsoft.FSharp.Collections`	Содержит неизменяемые классы коллекций, `List` такие `Array` как и.
`Microsoft.FSharp.Control`	Содержит типы для конструкций элементов управления, таких как отложенная оценка и асинхронные рабочие процессы.
`Microsoft.FSharp.Text`	Содержит функции для форматированного ввода-вывода, `printf` например функции.

Атрибут Автооткрытия

`AutoOpen` Атрибут можно применить к сборке, если необходимо автоматически открывать пространство имен или модуль при ссылке на сборку. Можно также применить `AutoOpen` атрибут к модулю для автоматического открытия этого модуля при открытии родительского модуля или пространства имен. Дополнительные сведения см. в разделе класс Core. AutoOpenAttribute.

Атрибут Рекуирекуалифиедакцесс

Некоторые модули, записи или типы объединений могут задавать `RequireQualifiedAccess` атрибут. При ссылке на элементы этих модулей, записей или объединений необходимо использовать полное имя независимо от того, включено ли объявление импорта. Если этот атрибут используется стратегически для типов, определяющих часто используемые имена, можно избежать конфликтов имен и, таким образом, сделать код более устойчивым к изменениям в библиотеках. Дополнительные сведения см. в разделе класс Core. рекуирекуалифиедакцессаттрибуте.

См. также

- Справочник по языку F#
- Пространства имен
- Модули

Сигнатуры

23.10.2019 • 8 minutes to read • Edit Online

В файле сигнатур содержатся сведения об открытых сигнатурах набора элементов программы на языке F#, таких как типы, пространства имен и модули. Файл сигнатур можно использовать для указания доступности этих элементов программы.

Примечания

Для каждого файла кода F# может иметься *файл сигнатур*— файл с тем же именем, что и файл кода, но с расширением FSI вместо FS. Файлы сигнатур также можно добавлять в командную строку компиляции, если командная строка используется напрямую. Для различения файлов кода и файлов сигнатур файлы кода иногда называют *файлами реализации*. В проекте файл сигнатур должен предшествовать связанному с ним файлу кода.

Файл сигнатур описывает пространства имен, модули, типы и члены в соответствующем файле реализации. Информацию в файле сигнатур можно использовать для указания того, к каким частям кода в соответствующем файле реализации возможен доступ из кода, находящегося за пределами файла реализации, и какие части являются внутренними по отношению к файлу реализации. Пространства имен, модули и типы, включаемые в файл сигнатур, должны представлять собой подмножество имен пространств, модулей и типов, включаемых в файл реализации. За некоторыми исключениями, рассмотренными ниже в этом разделе, языковые элементы, отсутствующие в файле сигнатур, считаются закрытыми по отношению к файлу реализации. Если в проекте или в командной строке не найден файл сигнатур, используется доступность по умолчанию.

Дополнительные сведения о специальных возможностях по умолчанию см. в разделе Управление доступом.

В файле сигнатур не нужно повторять определение типов и реализаций для каждого метода или функции. Вместо этого для каждого метода и функции используется сигнатура, выступающая в качестве полной спецификации функциональности, реализуемой фрагментом модуля или пространства имен. Синтаксис сигнатуры типа идентичен используемому в объявлениях абстрактных методов в интерфейсах и абстрактных классах; он также отображается IntelliSense и интерпретатором языка F# (fsi.exe) при выводе правильно скомпилированных входных данных.

Если в сигнатуре типа недостаточно информации для определения того, является ли тип запечатанным или типом интерфейса, нужно добавить атрибут, по которому компилятор сможет определить природу типа. Используемые с этой целью атрибуты приведены в таблице ниже.

АТРИБУТ	ОПИСАНИЕ
`[<Sealed>]`	Для типа, который не имеет абстрактных членов или не должен расширяться.
`[<Interface>]`	Для типа, являющегося интерфейсом.

Компилятор выдает ошибку, если атрибуты в сигнатуре и в объявлении в файле реализации не соответствуют друг другу.

Для создания сигнатуры для значения или значения функции используется ключевое слово `val` . Сигнатура типа предваряется ключевым словом `type` .

Сформировать файл сигнатур можно с помощью параметра `--sig` компилятора. Как правило, вручную FSI-файлы не пишутся. Вместо этого вы создаете FSI-файлы с помощью компилятора, добавляете их в проект (при работе с проектом) и редактируете их, удаляя методы и функции, которые не должны быть доступными.

В отношении сигнатур типов действует ряд правил.

- Аббревиатуры типов в файле реализации не должны совпадать с полными названиями типов в файле сигнатур.
- Записи и размеченные объединения должны предоставлять либо все свои поля и конструкторы, либо никакие из них, и порядок в сигнатуре должен совпадать с порядком в файле реализации. Классы могут открывать некоторые, все или никакие из своих полей и методов в сигнатуре.
- Классы и структуры, имеющие конструкторы, должны предоставлять объявления своих базовых классов (объявление `inherits`). Кроме того, классы и структуры, имеющие конструкторы, должны предоставлять все свои абстрактные методы и объявления интерфейсов.
- Типы интерфейсов должны открывать все свои методы и интерфейсы.

К сигнатурам значений применяются приведенные ниже правила.

- Модификаторы доступности (`public` , `internal` и т. д.) и модификаторы `inline` и `mutable` в сигнатуре должны совпадать с модификаторами в реализации.
- Количество параметров универсального типа (неявно выведенных или явно объявленных) должно совпадать. Также должны совпадать типы и ограничения типов в параметрах универсального типа.
- Если используется атрибут `Literal` , он должен присутствовать и в сигнатуре, и в реализации, и в обоих случая должно использоваться одно и то же литеральное значение.
- Шаблон параметров (также называемый *арностью*) сигнатур должен соответствовать шаблону параметров реализаций.
- Если имена параметров в файле сигнатур отличаются от соответствующего файла реализации, вместо него будет использоваться имя файла сигнатуры, что может вызвать проблемы при отладке или профилировании. Если вы хотите получать уведомления о таких несовпадениях, включите предупреждение 3218 в файле проекта или при вызове компилятора (см `--warnon` . раздел параметры компилятора).

В приведенном ниже примере показан файл сигнатур, содержащий сигнатуры пространства имен, модуля, значения функции и типов вместе с соответствующими атрибутами. Показан также соответствующий файл реализации.

```
// Module1.fsi

namespace Library1
  module Module1 =
    val function1 : int -> int
    type Type1 =
        new : unit -> Type1
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Sealed>]
    type Type2 =
        new : unit -> Type2
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

В приведенном ниже коде показан файл реализации.

```
namespace Library1

module Module1 =

    let function1 x = x + 1


    type Type1() =
        member type1.method1() =
            printfn "type1.method1"
        member type1.method2() =
            printfn "type1.method2"


    [<Sealed>]
    type Type2() =
        member type2.method1() =
            printfn "type2.method1"
        member type2.method2() =
            printfn "type2.method2"

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

См. также

- Справочник по языку F#
- Управление доступом
- Параметры компилятора

Единицы измерения

23.10.2019 • 12 minutes to read • Edit Online

Целочисленные значения с плавающей запятой F# и со знаком в могут иметь связанные единицы измерения, которые обычно используются для обозначения длины, объема, массы и т. д. Используя количественные единицы, вы включаете компилятор, чтобы убедиться, что арифметические связи имеют правильные единицы, что помогает предотвратить ошибки программирования.

Синтаксис

```
[<Measure>] type unit-name [ = measure ]
```

Примечания

В предыдущем синтаксисе параметр *Unit-Name* определяется как единица измерения. Необязательная часть используется для определения новой меры с точки зрения ранее определенных единиц. Например, в следующей строке определяется мера `cm` (сантиметр).

```
[<Measure>] type cm
```

Следующая строка определяет меру `ml` (миллилитер) как кубический сантиметр (`cm^3`).

```
[<Measure>] type ml = cm^3
```

В предыдущем синтаксисе *мера* — это формула, в которой участвуют единицы. В формулах, в которых задействованы единицы, целочисленные степени поддерживаются (положительные и отрицательные), пробелы между единицами указывают `*` на произведение двух единиц, также указывают `/` на произведение единиц и обозначает частное число единиц. Для обратной единицы можно использовать отрицательное целое число или значение `/` , которое обозначает разделение между числителем и знаменателем формулы единицы. Несколько единиц в знаменателе должны быть заключены в круглые скобки. Единицы, разделенные пробелами `/` после, обрабатываются как часть знаменателя, но все единицы после а `*` обрабатываются как часть числителя.

Можно использовать 1 в выражениях единиц измерения, чтобы указать количество без измерения или вместе с другими единицами, например в числителе. Например, единицы курса записываются как `1/s` , где `s` — секунды. Круглые скобки не используются в формулах единиц. Константы числового преобразования не указываются в формулах единиц; Однако константы преобразования можно определить с помощью единиц по отдельности и использовать их в вычислениях с проверкой единиц измерения.

Формулы единиц, означающие то же самое, можно написать различными способами. Таким образом, компилятор преобразует формулы единиц в согласованную форму, которая преобразует отрицательные степени в обратные, группирует единицы в один числитель и знаменатель и алфабетизес единицы в числителе и знаменателе.

Например, формулы `kg m s^-2` единиц и `m /s s * kg` преобразуются в `kg m/s^2` .

Единицы измерения используются в выражениях с плавающей запятой. Использование чисел с плавающей запятой вместе со связанными единицами измерения позволяет добавить другой уровень безопасности

типа и помогает избежать ошибок несоответствия единиц, которые могут возникать в формулах при использовании слабо типизированных чисел с плавающей запятой. При написании выражения с плавающей запятой, в котором используются единицы измерения, единицы в выражении должны совпадать.

Литералы можно закомментировать с помощью формулы единицы в угловых скобках, как показано в следующих примерах.

```
1.0<cm>
55.0<miles/hour>
```

Между числом и угловой скобкой не ставится пробел. Однако можно включить литеральный суффикс `f`, например, как показано в следующем примере.

```
// The f indicates single-precision floating point.
55.0f<miles/hour>
```

Такая Аннотация изменяет тип литерала с его примитивного типа (например `float` `float<miles/hour>`,) на размер измерения, `float<cm>` например или, в данном случае. Заметка единицы измерения `<1>` указывает на неизмерение количества, и его тип эквивалентен типу-примитиву без параметра Unit.

Тип единицы измерения — это целочисленный тип с плавающей запятой или со знаком, а также дополнительный комментарий к единице, обозначенный в квадратных скобках. Таким образом, при написании типа преобразования из `g` (грамм) в `kg` (килограммы) эти типы описываются следующим образом.

```
let convertg2kg (x : float<g>) = x / 1000.0<g/kg>
```

Единицы измерения используются для проверки единиц времени компиляции, но не сохраняются в среде выполнения. Поэтому они не влияют на производительность.

Единицы измерения можно применять к любому типу, а не только к типам с плавающей запятой. Тем не менее, только типы с плавающей запятой, целочисленные типы со знаком и десятичные типы поддерживают измерения количества. Поэтому имеет смысл использовать единицы измерения только для примитивных типов и для агрегатов, содержащих эти примитивные типы.

В следующем примере показано использование единиц измерения.

```
// Mass, grams.
[<Measure>] type g
// Mass, kilograms.
[<Measure>] type kg
// Weight, pounds.
[<Measure>] type lb

// Distance, meters.
[<Measure>] type m
// Distance, cm
[<Measure>] type cm

// Distance, inches.
[<Measure>] type inch
// Distance, feet
[<Measure>] type ft

// Time, seconds.
[<Measure>] type s

// Force, Newtons.
[<Measure>] type N = kg m / s^2

// Pressure, bar.
[<Measure>] type bar
// Pressure, Pascals
[<Measure>] type Pa = N / m^2

// Volume, milliliters.
[<Measure>] type ml
// Volume, liters.
[<Measure>] type L

// Define conversion constants.
let gramsPerKilogram : float<g kg^-1> = 1000.0<g/kg>
let cmPerMeter : float<cm/m> = 100.0<cm/m>
let cmPerInch : float<cm/inch> = 2.54<cm/inch>

let mlPerCubicCentimeter : float<ml/cm^3> = 1.0<ml/cm^3>
let mlPerLiter : float<ml/L> = 1000.0<ml/L>

// Define conversion functions.
let convertGramsToKilograms (x : float<g>) = x / gramsPerKilogram
let convertCentimetersToInches (x : float<cm>) = x / cmPerInch
```

В следующем примере кода показано, как преобразовать из числа с плавающей запятой, не определяющего, в измерение с плавающей запятой. Вы просто умножаете на 1,0, применяя измерения к 1,0. Это можно представить в виде функции, например `degreesFahrenheit`.

Кроме того, при передаче измеренных значений в функции, которые предполагают безразмерные числа с плавающей запятой, необходимо отказаться от `float` единиц или привести `float` к типу с помощью оператора. В этом примере вы делите `1.0<degC>` на `printf` аргументы в, так как `printf` предполагается количество безразмерных измерений.

```
[<Measure>] type degC // temperature, Celsius/Centigrade
[<Measure>] type degF // temperature, Fahrenheit

let convertCtoF ( temp : float<degC> ) = 9.0<degF> / 5.0<degC> * temp + 32.0<degF>
let convertFtoC ( temp: float<degF> ) = 5.0<degC> / 9.0<degF> * ( temp - 32.0<degF>)

// Define conversion functions from dimensionless floating point values.
let degreesFahrenheit temp = temp * 1.0<degF>
let degreesCelsius temp = temp * 1.0<degC>

printfn "Enter a temperature in degrees Fahrenheit."
let input = System.Console.ReadLine()
let parsedOk, floatValue = System.Double.TryParse(input)
if parsedOk
   then
      printfn "That temperature in Celsius is %8.2f degrees C." ((convertFtoC (degreesFahrenheit
floatValue))/(1.0<degC>))
   else
      printfn "Error parsing input."
```

В следующем примере сеанса показаны выходные данные и входные данные этого кода.

```
Enter a temperature in degrees Fahrenheit.
90
That temperature in degrees Celsius is    32.22.
```

Использование универсальных единиц

Можно создавать универсальные функции, которые работают с данными, имеющими связанную единицу измерения. Для этого нужно указать тип вместе с универсальной единицей в качестве параметра типа, как показано в следующем примере кода.

```
// Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

let genericSumUnits ( x : float<'u>) (y: float<'u>) = x + y

let v1 = 3.1<m/s>
let v2 = 2.7<m/s>
let x1 = 1.2<m>
let t1 = 1.0<s>

// OK: a function that has unit consistency checking.
let result1 = genericSumUnits v1 v2
// Error reported: mismatched units.
// Uncomment to see error.
// let result2 = genericSumUnits v1 x1
```

Создание агрегатных типов с универсальными единицами

В следующем коде показано, как создать агрегатный тип, состоящий из отдельных значений с плавающей запятой, имеющих универсальные единицы измерения. Это позволяет создать один тип, который работает с различными единицами измерения. Кроме того, универсальные единицы измерения сохраняют безопасность типа, гарантируя, что универсальный тип, имеющий один набор единиц, отличается от типа того же универсального типа с другим набором единиц. На основе этого метода `Measure` можно применить атрибут к параметру типа.

```
 // Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

// Define a vector together with a measure type parameter.
// Note the attribute applied to the type parameter.
type vector3D<[<Measure>] 'u> = { x : float<'u>; y : float<'u>; z : float<'u>}

// Create instances that have two different measures.
// Create a position vector.
let xvec : vector3D<m> = { x = 0.0<m>; y = 0.0<m>; z = 0.0<m> }
// Create a velocity vector.
let v1vec : vector3D<m/s> = { x = 1.0<m/s>; y = -1.0<m/s>; z = 0.0<m/s> }
```

Единицы во время выполнения

Единицы измерения используются для проверки статических типов. При компиляции значений с плавающей запятой единицы измерения исключаются, поэтому они теряются во время выполнения. Таким образом, любая попытка реализовать функциональность, зависящую от проверки единиц во время выполнения, невозможна. Например, невозможно реализовать `ToString` функцию для вывода единиц на печать.

Преобразования

Для преобразования типа с единицами (например, `float<'u>`) в тип, не имеющий единиц, можно использовать стандартную функцию преобразования. Например, можно использовать `float` для преобразования `float` в значение, не имеющее единиц, как показано в следующем коде.

```
[<Measure>]
type cm
let length = 12.0<cm>
let x = float length
```

Чтобы преобразовать значение без единиц измерения в значение, имеющее единицы измерения, можно умножить его на значение 1 или 1,0 с заметками соответствующих единиц. Однако для написания уровней взаимодействия существуют также некоторые явные функции, которые можно использовать для преобразования бесмодульных значений в значения с единицами измерения. Они находятся в модуле Microsoft. FSharp. Core. LanguagePrimitives . Например, чтобы выполнить преобразование из бесмодульного `float` модуля `float<cm>` в, используйте флоатвисмеасуре, как показано в следующем коде.

```
open Microsoft.FSharp.Core
let height:float<cm> = LanguagePrimitives.FloatWithMeasure x
```

Единицы измерения в F# основной библиотеке

Библиотека единиц измерения доступна в `FSharp.Data.UnitSystems.SI` пространстве имен. Она включает в себя единицы СИ как в виде символов (`m` например, для счетчика `UnitSymbols`) в подпространстве имен, так и `meter` их полное имя (например, для `UnitNames` счетчика) в подпространстве имен.

См. также

- Справочник по языку F#

Документация XML

08.01.2020 • 5 minutes to read • Edit Online

Вы можете создать документацию по комментариям к коду с тройной косой чертой F#(///) в. Комментарии XML могут предшествовать объявлениям в файлах кода (FS) или файлах сигнатуры (FSI).

Создание документации из комментариев

Поддержка в F# для создания документации из комментариев такая же, как и на других языках .NET Framework. Как и в других .NET Framework языках, параметр компилятора-doc позволяет создать XML-файл, содержащий сведения, которые можно преобразовать в документацию с помощью такого средства, как DocFX или Sandcastle. Документация, созданная с помощью средств, предназначенных для использования со сборками, написанными на других языках .NET Framework, обычно создает представление интерфейсов API, основанных на скомпилированной F# форме конструкций. Если средства специально не F#поддерживаются, документация, создаваемая этими инструментами, F# не соответствует представлению API.

Дополнительные сведения о создании документации из XML см. в разделе (комментарии XML-документации руководство# по)программированию C.

Рекомендуемые теги

Существует два способа написания комментариев XML-документации. Один из них — просто написать документацию непосредственно в комментарии с тройной косой чертой без использования XML-тегов. В этом случае весь текст комментария будет создан как сводная документация для конструкции кода, которая сразу же следует за. Используйте этот метод, если нужно написать только краткую сводку по каждой конструкции. Другой способ — использовать XML-теги для предоставления более структурированной документации. Второй метод позволяет указать отдельные примечания для краткой сводки, дополнительные примечания, документацию для каждого параметра и параметра типа, а также описание возвращаемого значения. В следующей таблице описаны XML-теги, распознаваемые F# в комментариях к коду XML.

СИНТАКСИС ТЕГА	ОПИСАНИЕ
<c> *text* **</c>**	Указывает, что *текст* является кодом. Этот тег может использоваться генераторами документации для показа текста в шрифте, подходящем для кода.
<summary> *text* **</Summary>**	Указывает, что *текст* является кратким описанием элемента Program. Описание обычно состоит из одного или двух предложений.
<замечания> *text* **</РЕМАРКС>**	Указывает, что *текст* содержит дополнительные сведения об элементе Program.
<param Name = " *Name* **">** *Description* **</param Returns>**	Задает имя и описание для параметра функции или метода.
<typeparam Name = " *имя* **">** *Description* **</типепарам>**	Задает имя и описание для параметра типа.

СИНТАКСИС ТЕГА	ОПИСАНИЕ
<возвращает> *text* **</Returns>**	Указывает, что *текст* описывает возвращаемое значение функции или метода.
<cref Exception = " *mun* **"> *Description* </ексцептион>**	Указывает тип исключения, которое может быть создано, и обстоятельства, при которых оно вызывается.
<см. в разделе cref = " *Reference* **"> *Text* </Си>**	Указывает встроенную ссылку на другой элемент программы. *Ссылка* — это имя, которое отображается в XML-файле документации. *Текст* — это текст, отображаемый в ссылке.
<seeAlso cref = " *reference* **"/>**	Указывает см. также ссылку на документацию для другого типа. *Ссылка* — это имя, которое отображается в XML-файле документации. См. также ссылки, которые обычно отображаются в нижней части страницы документации.
<para> *text* **</пара>**	Задает абзац текста. Используется для разделения текста внутри тега **remarks** .

Пример

Описание

Ниже приведен типичный комментарий XML-документации в файле сигнатуры.

Код

```
/// <summary>Builds a new string whose characters are the results of applying the function <c>mapping</c>
/// to each of the characters of the input string and concatenating the resulting
/// strings.</summary>
/// <param name="mapping">The function to produce a string from each character of the input string.</param>
///<param name="str">The input string.</param>
///<returns>The concatenated string.</returns>
///<exception cref="System.ArgumentNullException">Thrown when the input string is null.</exception>
val collect : (char -> string) -> string -> string
```

Пример

Описание

В следующем примере показан альтернативный метод без XML-тегов. В этом примере весь текст комментария рассматривается как сводка. Обратите внимание, что если тег Summary не указан явно, не следует указывать другие теги, такие как **param** или **Returns** .

Код

```
/// Creates a new string whose characters are the result of applying
/// the function mapping to each of the characters of the input string
/// and concatenating the resulting strings.
val collect : (char -> string) -> string -> string
```

См. также:

- Справочник по языку F#
- Параметры компилятора

Отложенные выражения

23.10.2019 • 2 minutes to read • Edit Online

Отложенные выражения — это выражения, которые не оцениваются немедленно, а оцениваются, когда требуется результат. Это поможет повысить производительность кода.

Синтаксис

```
let identifier = lazy ( expression )
```

Примечания

В приведенном выше синтаксисе *выражение* — это код, который вычисляется только тогда, когда требуется результат, а *идентификатор* — это значение, в котором хранится результат. Значение имеет тип `Lazy<'T>`, где фактический тип, используемый для `'T`, определяется из результата выражения.

Отложенные выражения позволяют повысить производительность за счет ограниченного выполнения выражений только теми ситуациями, в которых требуется результат.

Чтобы принудительно выполнить выражения, вызовите метод `Force`. `Force` приводит к выполнению выполнения только один раз. Последующие вызовы `Force` возвращают тот же результат, но не выполняют никакой код.

В следующем коде показано использование отложенных выражений и использование `Force`. В `result` этом коде тип имеет значение `Lazy<int>`, `int` а `Force` метод возвращает.

```
let x = 10
let result = lazy (x + 10)
printfn "%d" (result.Force())
```

Отложенная оценка, но не `Lazy` тип, также используется для последовательностей. Дополнительные сведения см. в разделе последовательности.

См. также

- Справочник по языку F#
- Модуль Лазекстенсионс

Выражения вычисления

01.12.2019 • 21 minutes to read • Edit Online

Вычислительные выражения F# в предоставляют удобный синтаксис для написания вычислений, которые могут быть упорядочены и объединены с помощью конструкций и привязок потока управления. В зависимости от типа вычислительного выражения их можно представить как способ выражения составных, моноидс, нестандартных преобразователей и аппликативе операторов. Однако, в отличие от других языков (например , *в* Haskell), они не привязаны к одной абстракции и не полагаются на макросы или другие формы метапрограммирование для выполнения удобного и контекстно-чувствительного синтаксиса.

Обзор

Вычисления могут принимать множество форм. Наиболее распространенной формой вычислений является однопотоковое выполнение, которое легко понять и изменить. Однако не все формы вычислений являются простым выполнением с одним потоком. Ниже приведены некоторые примеры таких ситуаций.

- Недетерминированные вычисления
- Асинхронные вычисления
- Вычисления с эффективностью
- Регенеративные вычисления

Как правило, существуют *зависящие от контекста* вычисления, которые необходимо выполнить в определенных частях приложения. Написание контекстно-зависимого кода может быть непростой задачей, так как легко «утечку» вычислений за пределами данного контекста без абстракций, чтобы предотвратить это. Эти абстракции часто сложны в написании самих себя, т F# . е. у них есть обобщенный способ сделать это, называемый **вычислительными выражениями**.

Вычислительные выражения предлагают единый синтаксис и модель абстракции для кодирования зависящих от контекста вычислений.

Каждое вычислительное выражение поддерживается типом *построителя* . Тип построителя определяет операции, доступные для вычислительного выражения. См. раздел Создание нового типа вычислительного выражения, в котором показано, как создать пользовательское вычислительное выражение.

Обзор синтаксиса

Все выражения вычислений имеют следующий вид:

```
builder-expr { cexper }
```

где `builder-expr` — имя типа построителя, определяющего вычислительное выражение, а `cexper` — тело выражения вычисления. Например, `async` код выражения вычисления может выглядеть следующим образом:

```
let fetchAndDownload url =
    async {
        let! data = downloadData url

        let processedData = processData data

        return processedData
    }
```

В вычислительном выражении доступен Специальный дополнительный синтаксис, как показано в предыдущем примере. В выражениях вычислений возможны следующие формы выражений:

```
expr { let! ... }
expr { do! ... }
expr { yield ... }
expr { yield! ... }
expr { return ... }
expr { return! ... }
expr { match! ... }
```

Каждое из этих ключевых слов и другие стандартные F# ключевые слова доступны только в вычислительном выражении, если они были определены в типе построителя резервных копий. Единственным исключением из этого является `match!`, которое само по себе является синтаксическим SugarCRM для использования `let!`, за которым следует соответствие шаблона в результате.

Тип построителя — это объект, который определяет специальные методы, определяющие способ объединения фрагментов вычислительного выражения. то есть его методы управляют тем, как работает вычислительное выражение. Другой способ описать класс построителя — сказать, что он позволяет настраивать работу многих F# конструкций, таких как циклы и привязки.

`let!`

Ключевое слово `let!` привязывает результат вызова другого вычислительного выражения к имени:

```
let doThingsAsync url =
    async {
        let! data = getDataAsync url
        ...
    }
```

При привязке вызова к вычислительному выражению с `let` не будет получен результат вычисления выражения. Вместо этого вы привязываете значение *нереализованного* вызова к этому вычислительному выражению. Для привязки к результату используйте `let!`.

`let!` определяется членом `Bind(x, f)` в типе построителя.

`do!`

Ключевое слово `do!` предназначено для вызова вычислительного выражения, возвращающего тип, схожий с `unit` (определяется элементом `Zero` в построителе):

```
let doThingsAsync data url =
    async {
        do! submitData data url
        ...
    }
```

Для асинхронного рабочего процессаэтот тип `Async<unit>`. Для других вычислительных выражений тип,

скорее всего, будет `CExpType<unit>`.

`do!` определяется членом `Bind(x, f)` в типе построителя, где `f` создает `unit`.

`yield`

Ключевое слово `yield` используется для возвращения значения из вычислительного выражения, чтобы его можно было использовать как IEnumerable<T>:

```
let squares =
    seq {
        for i in 1..10 do
            yield i * i
    }

for sq in squares do
    printfn "%d" sq
```

В большинстве случаев его можно опустить вызывающими методами. Наиболее распространенным способом опустить `yield` является оператор `->`:

```
let squares =
    seq {
        for i in 1..10 -> i * i
    }

for sq in squares do
    printfn "%d" sq
```

Для более сложных выражений, которые могут выдавать множество различных значений, и, возможно, условный, просто Пропуск ключевого слова может выполнять следующие действия:

```
let weekdays includeWeekend =
    seq {
        "Monday"
        "Tuesday"
        "Wednesday"
        "Thursday"
        "Friday"
        if includeWeekend then
            "Saturday"
            "Sunday"
    }
```

Как и в случае с ключевым C#словом yield в , каждый элемент в вычислительном выражении получает обратную передачу по мере выполнения итерации.

`yield` определяется членом `Yield(x)` в типе построителя, где `x` — это элемент, который необходимо вернуть.

`yield!`

Ключевое слово `yield!` предназначено для выравнивания коллекции значений из вычислительного выражения:

```
let squares =
    seq {
        for i in 1..3 -> i * i
    }

let cubes =
    seq {
        for i in 1..3 -> i * i * i
    }

let squaresAndCubes =
    seq {
        yield! squares
        yield! cubes
    }

printfn "%A" squaresAndCubes // Prints - 1; 4; 9; 1; 8; 27
```

При вычислении вычислительное выражение, вызываемое `yield!`, будет возвращать возвращенные элементы по одному, сведеня результат.

`yield!` определяется членом `YieldFrom(x)` в типе построителя, где `x` является коллекцией значений.

В отличие от `yield`, необходимо явно указать `yield!`. Его поведение не является неявным в вычислительных выражениях.

`return`

Ключевое слово `return` заключает в оболочку значение в типе, соответствующем вычислительному выражению. Помимо вычислительных выражений, использующих `yield`, оно используется для полного выражения вычислений:

```
let req = // 'req' is of type is 'Async<data>'
    async {
        let! data = fetch url
        return data
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req
```

`return` определяется членом `Return(x)` в типе построителя, где `x` — это элемент для переноса.

`return!`

Ключевое слово `return!` вычисляет значение вычислительного выражения и заключает результат в тип, соответствующий вычислительному выражению:

```
let req = // 'req' is of type is 'Async<data>'
    async {
        return! fetch url
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req
```

`return!` определяется членом `ReturnFrom(x)` в типе построителя, где `x` является другим вычислительным выражением.

`match!`

Ключевое слово `match!` позволяет встроено вызвать в другое вычислительное выражение и сопоставить шаблон с его результатом:

```
let doThingsAsync url =
    async {
        match! callService url with
        | Some data -> ...
        | None -> ...
    }
```

При вызове вычислительного выражения с `match!` будет выдаваться результат такого вызова, как `let!`. Это часто используется при вызове вычислительного выражения, в котором результат является необязательным.

Встроенные вычислительные выражения

F# Базовая библиотека определяет три встроенных вычислительных выражения: выражения последовательности, асинхронные рабочие процессыи выражения запросов.

Создание нового типа вычислительного выражения

Вы можете определить характеристики собственных вычислительных выражений, создав класс построителя и определив определенные специальные методы в классе. Класс построителя может дополнительно определять методы, перечисленные в следующей таблице.

В следующей таблице описаны методы, которые можно использовать в классе построителя рабочих процессов.

МЕТОД	ТИПИЧНЫЕ ПОДПИСИ	ОПИСАНИЕ
`Bind`	`M<'T> * ('T -> M<'U>) -> M<'U>`	Вызывается для `let!` и `do!` в вычислительных выражениях.
`Delay`	`(unit -> M<'T>) -> M<'T>`	Заключает в оболочку вычислительное выражение как функцию.
`Return`	`'T -> M<'T>`	Вызывается для `return` в вычислительных выражениях.
`ReturnFrom`	`M<'T> -> M<'T>`	Вызывается для `return!` в вычислительных выражениях.
`Run`	`M<'T> -> M<'T>` или `M<'T> -> 'T`	Выполняет вычислительное выражение.
`Combine`	`M<'T> * M<'T> -> M<'T>` или `M<unit> * M<'T> -> M<'T>`	Вызывается для виртуализации в вычислительных выражениях.

МЕТОД	ТИПИЧНЫЕ ПОДПИСИ	ОПИСАНИЕ
`For`	`seq<'T> * ('T -> M<'U>) -> M<'U>` или `seq<'T> * ('T -> M<'U>) -> seq<M<'U>>`	Вызывается для `for...do` выражений в вычислительных выражениях.
`TryFinally`	`M<'T> * (unit -> unit) -> M<'T>`	Вызывается для `try...finally` выражений в вычислительных выражениях.
`TryWith`	`M<'T> * (exn -> M<'T>) -> M<'T>`	Вызывается для `try...with` выражений в вычислительных выражениях.
`Using`	`'T * ('T -> M<'U>) -> M<'U> when 'T :> IDisposable`	Вызывается для `use` привязок в вычислительных выражениях.
`While`	`(unit -> bool) * M<'T> -> M<'T>`	Вызывается для `while...do` выражений в вычислительных выражениях.
`Yield`	`'T -> M<'T>`	Вызывается для `yield` выражений в вычислительных выражениях.
`YieldFrom`	`M<'T> -> M<'T>`	Вызывается для `yield!` выражений в вычислительных выражениях.
`Zero`	`unit -> M<'T>`	Вызывается для пустых `else` ветвей `if...then` выражений в вычислительных выражениях.
`Quote`	`Quotations.Expr<'T> -> Quotations.Expr<'T>`	Указывает, что вычислительное выражение передается в элемент `Run` в качестве предложения. Он преобразует все экземпляры вычислений в кавычки.

Многие методы в классе построителя используют и возвращают `M<'T>` конструкцию, которая обычно представляет собой отдельный определяемый тип, который характеризует тип объединяемых вычислений, например `Async<'T>` для асинхронных рабочих процессов и `Seq<'T>` для рабочих процессов последовательности. Сигнатуры этих методов позволяют объединять и вкладывать друг в друга, чтобы объект рабочего процесса, возвращенный из одной конструкции, можно было передать далее. Компилятор при синтаксическом анализе вычислительного выражения преобразует выражение в ряд вложенных вызовов функций, используя методы из предыдущей таблицы и код в вычислительном выражении.

Вложенное выражение имеет следующий вид:

```
builder.Run(builder.Delay(fun () -> {| cexpr |}))
```

В приведенном выше коде вызовы `Run` и `Delay` опущены, если они не определены в классе построителя вычислительных выражений. Текст вычислительного выражения, обозначенный как

`{| cexpr |}`, преобразуется в вызовы, в которых используются методы класса Builder, с помощью переводов, описанных в следующей таблице. Выражение вычисления `{| cexpr |}` определяется рекурсивно в соответствии с этими переводами, где `expr` является F# выражением, а `cexpr` является вычислительным выражением.

ВЫРАЖЕНИЕ	ПРЕОБРАЗОВАНИЕ
`{ let binding in cexpr }`	`let binding in {| cexpr |}`
`{ let! pattern = expr in cexpr }`	`builder.Bind(expr, (fun pattern -> {| cexpr |}))`
`{ do! expr in cexpr }`	`builder.Bind(expr, (fun () -> {| cexpr |}))`
`{ yield expr }`	`builder.Yield(expr)`
`{ yield! expr }`	`builder.YieldFrom(expr)`
`{ return expr }`	`builder.Return(expr)`
`{ return! expr }`	`builder.ReturnFrom(expr)`
`{ use pattern = expr in cexpr }`	`builder.Using(expr, (fun pattern -> {| cexpr |}))`
`{ use! value = expr in cexpr }`	`builder.Bind(expr, (fun value -> builder.Using(value, (fun value -> { cexpr }))))`
`{ if expr then cexpr0 |}`	`if expr then { cexpr0 } else binder.Zero()`
`{ if expr then cexpr0 else cexpr1 |}`	`if expr then { cexpr0 } else { cexpr1 }`
`{ match expr with | pattern_i -> cexpr_i }`	`match expr with | pattern_i -> { cexpr_i }`
`{ for pattern in expr do cexpr }`	`builder.For(enumeration, (fun pattern -> { cexpr }))`
`{ for identifier = expr1 to expr2 do cexpr }`	`builder.For(enumeration, (fun identifier -> { cexpr }))`
`{ while expr do cexpr }`	`builder.While(fun () -> expr, builder.Delay({ cexpr }))`
`{ try cexpr with | pattern_i -> expr_i }`	`builder.TryWith(builder.Delay({ cexpr }), (fun value -> match value with | pattern_i -> expr_i | exn -> reraise exn)))`
`{ try cexpr finally expr }`	`builder.TryFinally(builder.Delay( { cexpr }), (fun () -> expr))`
`{ cexpr1; cexpr2 }`	`builder.Combine({ cexpr1 }, { cexpr2 })`
`{ other-expr; cexpr }`	`expr; { cexpr }`
`{ other-expr }`	`expr; builder.Zero()`

В предыдущей таблице `other-expr` описывает выражение, которое не указано в таблице в других случаях. Классу построителя не требуется реализовывать все методы и поддерживать все переводы, перечисленные в предыдущей таблице. Эти конструкции, которые не реализованы, недоступны в вычислительных выражениях этого типа. Например, если не требуется поддерживать ключевое слово `use` в вычислительных выражениях, можно опустить определение `Use` в классе построителя.

В следующем примере кода показано вычислительное выражение, которое инкапсулирует вычисление в виде последовательности шагов, которые можно оценить по одному шагу за раз. Тип размеченного объединения, `OkOrException`, кодирует состояние ошибки выражения, как было оценено до сих пор. Этот код демонстрирует несколько типовых шаблонов, которые можно использовать в вычислительных выражениях, таких как стандартные реализации некоторых методов построителя.

```
// Computations that can be run step by step
type Eventually<'T> =
    | Done of 'T
    | NotYetDone of (unit -> Eventually<'T>)

module Eventually =
    // The bind for the computations. Append 'func' to the
    // computation.
    let rec bind func expr =
        match expr with
        | Done value -> func value
        | NotYetDone work -> NotYetDone (fun () -> bind func (work()))

    // Return the final value wrapped in the Eventually type.
    let result value = Done value

    type OkOrException<'T> =
        | Ok of 'T
        | Exception of System.Exception

    // The catch for the computations. Stitch try/with throughout
    // the computation, and return the overall result as an OkOrException.
    let rec catch expr =
        match expr with
        | Done value -> result (Ok value)
        | NotYetDone work ->
            NotYetDone (fun () ->
                let res = try Ok(work()) with | exn -> Exception exn
                match res with
                | Ok cont -> catch cont // note, a tailcall
                | Exception exn -> result (Exception exn))

    // The delay operator.
    let delay func = NotYetDone (fun () -> func())

    // The stepping action for the computations.
    let step expr =
        match expr with
        | Done _ -> expr
        | NotYetDone func -> func ()

    // The rest of the operations are boilerplate.
    // The tryFinally operator.
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryFinally expr compensation =
        catch (expr)
        |> bind (fun res ->
            compensation();
            match res with
            | Ok value -> result value
            | Exception exn -> raise exn)

    // The tryWith operator.
```

```
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryWith exn handler =
        catch exn
        |> bind (function Ok value -> result value | Exception exn -> handler exn)

    // The whileLoop operator.
    // This is boilerplate in terms of "result" and "bind".
    let rec whileLoop pred body =
        if pred() then body |> bind (fun _ -> whileLoop pred body)
        else result ()

    // The sequential composition operator.
    // This is boilerplate in terms of "result" and "bind".
    let combine expr1 expr2 =
        expr1 |> bind (fun () -> expr2)

    // The using operator.
    let using (resource: #System.IDisposable) func =
        tryFinally (func resource) (fun () -> resource.Dispose())

    // The forLoop operator.
    // This is boilerplate in terms of "catch", "result", and "bind".
    let forLoop (collection:seq<_>) func =
        let ie = collection.GetEnumerator()
        tryFinally
            (whileLoop
                (fun () -> ie.MoveNext())
                (delay (fun () -> let value = ie.Current in func value)))
            (fun () -> ie.Dispose())

// The builder class.
type EventuallyBuilder() =
    member x.Bind(comp, func) = Eventually.bind func comp
    member x.Return(value) = Eventually.result value
    member x.ReturnFrom(value) = value
    member x.Combine(expr1, expr2) = Eventually.combine expr1 expr2
    member x.Delay(func) = Eventually.delay func
    member x.Zero() = Eventually.result ()
    member x.TryWith(expr, handler) = Eventually.tryWith expr handler
    member x.TryFinally(expr, compensation) = Eventually.tryFinally expr compensation
    member x.For(coll:seq<_>, func) = Eventually.forLoop coll func
    member x.Using(resource, expr) = Eventually.using resource expr

let eventually = new EventuallyBuilder()

let comp = eventually {
    for x in 1..2 do
        printfn " x = %d" x
    return 3 + 4 }

// Try the remaining lines in F# interactive to see how this
// computation expression works in practice.
let step x = Eventually.step x

// returns "NotYetDone <closure>"
comp |> step

// prints "x = 1"
// returns "NotYetDone <closure>"
comp |> step |> step

// prints "x = 1"
// prints "x = 2"
// returns "Done 7"
comp |> step |> step |> step |> step
```

Вычислительное выражение имеет базовый тип, который возвращает выражение. Базовый тип может

представлять вычисленный результат или отложенное вычисление, которое может быть выполнено, или может предоставить способ итерации некоторого типа коллекции. В предыдущем примере базовый тип был в **конечном итоге**. Для выражения последовательности базовым типом является System.Collections.Generic.IEnumerable<T>. Для выражения запроса базовый тип System.Linq.IQueryable. Для асинхронного рабочего процесса базовый тип `Async`. Объект `Async` представляет работу, которую необходимо выполнить для вычисления результата. Например, вызовите `Async.RunSynchronously`, чтобы выполнить вычисление и вернуть результат.

Пользовательские операции

Можно определить пользовательскую операцию в вычислительном выражении и использовать настраиваемую операцию в качестве оператора в вычислительном выражении. Например, оператор запроса можно включить в выражение запроса. При определении пользовательской операции необходимо определить методы yield и for в вычислительном выражении. Чтобы определить пользовательскую операцию, добавьте ее в класс построителя для вычислительного выражения, а затем примените `CustomOperationAttribute`. Этот атрибут принимает строку в качестве аргумента, который является именем, используемым в пользовательской операции. Это имя поступает в область в начале открывающей фигурной скобки вычислительного выражения. Поэтому не следует использовать идентификаторы, имена которых совпадают с именами пользовательских операций в этом блоке. Например, Избегайте использования идентификаторов, таких как `all` или `last` в выражениях запросов.

Расширение существующих построителей с помощью новых настраиваемых операций

Если у вас уже есть класс построителя, его пользовательские операции можно расширить за пределами этого класса построителя. Расширения должны быть объявлены в модулях. Пространства имен не могут содержать элементы Extension, кроме в том же файле и в той же группе объявлений пространства имен, где определен тип.

В следующем примере показано расширение существующего класса `Microsoft.FSharp.Linq.QueryBuilder`.

```
type Microsoft.FSharp.Linq.QueryBuilder with

    [<CustomOperation("existsNot")>]
    member _.ExistsNot (source: QuerySource<'T, 'Q>, predicate) =
        Enumerable.Any (source.Source, Func<_,_>(predicate)) |> not
```

См. также:

- Справочник по языку F#
- Асинхронные рабочие потоки
- Последовательности
- Выражения запросов

Асинхронные рабочие потоки

23.10.2019 • 8 minutes to read • Edit Online

> **NOTE**
> Ссылка на справочник по API ведет на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

В этом разделе описывается поддержка F# в для асинхронного выполнения вычислений, т. е. без блокировки выполнения другой работы. Например, асинхронные вычисления можно использовать для написания приложений, которые имеют пользовательские интерфейсы, которые продолжают реагировать на запросы пользователей, так как приложение выполняет другую работу.

Синтаксис

```
async { expression }
```

Примечания

В предыдущем синтаксисе вычисление, представленное `expression` , настроено для асинхронного выполнения, то есть без блокировки текущего вычислительного потока при выполнении асинхронных операций сна, ввода-вывода и других асинхронных операций. Асинхронные вычисления часто запускаются в фоновом потоке, а выполнение продолжаются в текущем потоке. Выражение имеет `Async<'T>` тип, где `'T` — тип, возвращаемый выражением при `return` использовании ключевого слова. Код в таком выражении называется асинхронным блоком илиасинхронным блоком.

Существует множество способов программирования в асинхронном режиме, а `Async` класс предоставляет методы, поддерживающие несколько сценариев. Общий подход заключается в создании `Async` объектов, представляющих вычисления или вычисления, которые необходимо выполнять асинхронно, а затем запускайте эти вычисления с помощью одной из функций запуска. Различные функции, вызывающие срабатывание, предоставляют различные способы выполнения асинхронных вычислений, и какое из них зависит от того, хотите ли вы использовать текущий поток, фоновый поток или объект задачи .NET Framework, а также наличие продолжения. функции, которые должны выполняться после завершения вычисления. Например, чтобы запустить асинхронное вычисление в текущем потоке, можно использовать `Async.StartImmediate`. Когда вы запускаете асинхронное вычисление из потока пользовательского интерфейса, вы не блокируете главный цикл событий, который обрабатывает действия пользователя, такие как нажатия клавиш и действия мыши, чтобы приложение оставалось реагировать на запросы.

Асинхронная привязка с помощью let!

В асинхронном рабочем процессе некоторые выражения и операции являются синхронными, а некоторые — более длинными вычислениями, которые предназначены для асинхронного возврата результатов. При асинхронном вызове метода вместо обычной `let` привязки используется. `let!` Результат `let!` заключается в том, чтобы обеспечить продолжение выполнения в других вычислениях или потоках при выполнении вычислений. После того как правая часть `let!` привязки возвращаются, остальная часть асинхронного рабочего процесса возобновляет выполнение.

В следующем коде показано различие между `let` и. `let!` Строка кода, в которой используется `let` , просто создает асинхронное вычисление как объект, который можно запустить позднее с помощью, `Async.StartImmediate` например или `Async.RunSynchronously` . Строка кода, в которой используется `let!` , начинает вычисление, а затем поток приостанавливается до тех пор, пока результат не станет доступным, после чего выполнение продолжится.

```
// let just stores the result as an asynchronous operation.
let (result1 : Async<byte[]>) = stream.AsyncRead(bufferSize)
// let! completes the asynchronous operation and returns the data.
let! (result2 : byte[])  = stream.AsyncRead(bufferSize)
```

Кроме `let!` того, можно использовать `use!` для выполнения асинхронных привязок. Разница между `let!` и `use!` совпадает с разностью между `let` и `use` . Для `use!` объект удаляется при закрытии текущей области. Обратите внимание, что в текущем F# выпуске `use!` языка не позволяет инициализировать значение со значением `use` null, несмотря на это.

Асинхронные примитивы

Метод, выполняющий одну асинхронную задачу и возвращающий результат, называется *асинхронным примитивом*, и они предназначены специально для использования `let!` с. В F# основной библиотеке определены несколько асинхронных примитивов. Два таких метода для веб-приложений определяются в модуле `Microsoft.FSharp.Control.WebExtensions` : `WebRequest.AsyncGetResponse` и `WebClient.AsyncDownloadString` . Оба примитива загружают данные с веб-страницы по указанному URL-адресу. `AsyncGetResponse` Создает объект и `AsyncDownloadString` создает строку, представляющую код HTML для веб-страницы. `System.Net.WebResponse`

В `Microsoft.FSharp.Control.CommonExtensions` модуль включаются несколько примитивов для асинхронных операций ввода-вывода. Эти методы `System.IO.Stream` расширения класса являются `Stream.AsyncRead` и `Stream.AsyncWrite` .

Можно также написать собственные асинхронные примитивы, определив функцию, полный текст которой заключен в блок async.

Для использования асинхронных методов в .NET Framework, предназначенных для других асинхронных моделей с F# асинхронной моделью программирования, создается функция, возвращающая F# `Async` объект. F# Библиотека содержит функции, упрощающие это.

Сюда входит один пример использования асинхронных рабочих процессов; в документации есть множество других методов класса Async.

В этом примере показано, как использовать асинхронные рабочие процессы для параллельного выполнения вычислений.

В следующем примере кода функция `fetchAsync` получает текст HTML, возвращаемый веб-запросом. `fetchAsync` Функция содержит асинхронный блок кода. Если привязка выполняется к результату асинхронного примитива, в данном случае `AsyncDownloadString` — Let! используется вместо let.

Функция `Async.RunSynchronously` используется для выполнения асинхронной операции и ожидания ее результата. Например, можно выполнить несколько асинхронных операций параллельно, используя `Async.Parallel` функцию вместе `Async.RunSynchronously` с функцией. Функция принимает список `Async` объектов, настраивает код для каждого `Async` объекта задачи `Async` для параллельного выполнения и возвращает объект, представляющий параллельное вычисление. `Async.Parallel` Точно так же, как и для одной операции `Async.RunSynchronously` , вы вызываете, чтобы начать выполнение.

`runAll` Функция запускает три асинхронных рабочих процесса в параллельном режиме и ожидает до

завершения всех.

```
open System.Net
open Microsoft.FSharp.Control.WebExtensions

let urlList = [ "Microsoft.com", "http://www.microsoft.com/"
                "MSDN", "http://msdn.microsoft.com/"
                "Bing", "http://www.bing.com"
              ]

let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
    }

let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

runAll()
```

См. также

- Справочник по языку F#
- Выражения вычисления
- Класс Control. Async

Выражения запросов

04.11.2019 • 47 minutes to read • Edit Online

> **NOTE**
> Ссылки на справочник по API в этой статье ведут на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

Выражения запросов позволяют запрашивать источник данных и размещать данные в нужной форме. Выражения запросов обеспечивают поддержку LINQ в F#.

Синтаксис

```
query { expression }
```

Заметки

Выражения запросов — это тип вычислительного выражения, аналогичный выражениям последовательности. Точно так же, как вы указываете последовательность, предоставляя код в выражении последовательности, вы указываете набор данных, предоставляя код в выражении запроса. В выражении последовательности ключевое слово `yield` определяет данные, возвращаемые как часть результирующей последовательности. В выражениях запроса ключевое слово `select` выполняет ту же функцию. Помимо ключевого слова `select`, F# также поддерживает несколько операторов запросов, которые во многом аналогичны частям инструкции SQL SELECT. Ниже приведен пример простого выражения запроса вместе с кодом, который подключается к источнику OData базы данных Northwind.

```
// Use the OData type provider to create types that can be used to access the Northwind database.
// Add References to FSharp.Data.TypeProviders and System.Data.Services.Client
open Microsoft.FSharp.Data.TypeProviders

type Northwind = ODataService<"http://services.odata.org/Northwind/Northwind.svc">
let db = Northwind.GetDataContext()

// A query expression.
let query1 =
    query {
        for customer in db.Customers do
            select customer
    }

// Print results
query1
|> Seq.iter (fun customer -> printfn "Company: %s Contact: %s" customer.CompanyName customer.ContactName)
```

В предыдущем примере кода выражение запроса заключено в фигурные скобки. Значение кода в выражении — возвращает каждого клиента в таблице Customers в базе данных в результатах запроса. Выражения запроса возвращают тип, реализующий IQueryable<T> и IEnumerable<T>, поэтому их можно итеративно использовать в модуле seq , как показано в примере.

Каждый тип вычислительного выражения строится на основе класса построителя. Класс построителя для выражения вычисления запроса `QueryBuilder`. Дополнительные сведения см. в разделе выражения

вычислений и класс LINQ. QueryBuilder.

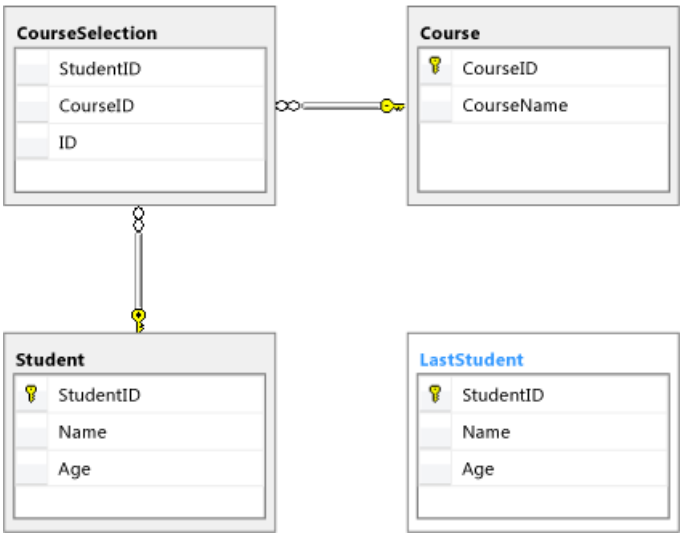
Операторы запроса

Операторы запросов позволяют указать сведения о запросе, например, чтобы задать условия для возвращаемых записей, или указать порядок сортировки результатов. Источник запроса должен поддерживать оператор запроса. При попытке использовать неподдерживаемый оператор запроса возникнет `System.NotSupportedException`.

В выражениях запросов допускаются только выражения, которые могут быть преобразованы в SQL. Например, в выражениях нельзя использовать вызовы функций при использовании оператора запроса `where`.

В таблице 1 показаны доступные операторы запросов. Кроме того, см. раздел Table2, в котором сравниваются SQL запросы F# и эквивалентные выражения запросов далее в этом разделе. Некоторые поставщики типов не поддерживают некоторые операторы запросов. В частности, поставщик типов OData ограничен в операторах запросов, которые он поддерживает, из-за ограничений в OData. Дополнительные сведения см. в разделе ODataService Type ProviderF#().

В этой таблице предполагается, что база данных имеет следующий вид:



Код в приведенных ниже таблицах также предполагает наличие следующего кода подключения к базе данных. Проекты должны добавлять ссылки на сборки System. Data, System. Data. LINQ и FSharp. Data. TypeProviders. Код, создающий эту базу данных, содержится в конце этого раздела.

```
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq
open Microsoft.FSharp.Linq

type schema = SqlDataConnection< @"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;" >

let db = schema.GetDataContext()

// Needed for some query operator examples:
let data = [ 1; 5; 7; 11; 18; 21]
```

Таблица 1. Операторы запроса

ОПЕРАТОРА	ОПИСАНИЕ

`contains`	Определяет, включает ли выбранные элементы указанный элемент. `query {` `  for student in db.Student do` `  select student.Age.Value` `  contains 11` `}`
`count`	Возвращает число выбранных элементов. `query {` `  for student in db.Student do` `  select student` `  count` `}`
`last`	Выбирает последний элемент выбранных на данный момент элементов. `query {` `  for number in data do` `  last` `}`
`lastOrDefault`	Выбирает последний элемент выбранных до сих пор или значение по умолчанию, если элемент не найден. `query {` `  for number in data do` `  where (number < 0)` `  lastOrDefault` `}`
`exactlyOne`	Выбирает один конкретный элемент, выбранный на данный момент. Если имеется несколько элементов, возникает исключение. `query {` `  for student in db.Student do` `  where (student.StudentID = 1)` `  select student` `  exactlyOne` `}`

`exactlyOneOrDefault`	Выбирает один конкретный элемент выбранных выше или значение по умолчанию, если этот элемент не найден. `query { for student in db.Student do where (student.StudentID = 1) select student exactlyOneOrDefault }`
`headOrDefault`	Выбирает первый элемент выбранных до сих пор или значение по умолчанию, если последовательность не содержит элементов. `query { for student in db.Student do select student headOrDefault }`
`select`	Проецирует каждый из элементов, выбранных на данный момент. `query { for student in db.Student do select student }`
`where`	Выбирает элементы на основе указанного предиката. `query { for student in db.Student do where (student.StudentID > 4) select student }`
`minBy`	Выбирает значение для каждого выбранного элемента на данный момент и возвращает минимальное результирующее значение. `query { for student in db.Student do minBy student.StudentID }`

`maxBy`	Выбирает значение для каждого выбранного элемента на данный момент и возвращает максимальное результирующее значение. `query { for student in db.Student do maxBy student.StudentID }`
`groupBy`	Группирует элементы, выбранные на данный момент, в соответствии с заданным селектором ключа. `query { for student in db.Student do groupBy student.Age into g select (g.Key, g.Count()) }`
`sortBy`	Сортирует выбранные элементы в порядке возрастания по заданному ключу сортировки. `query { for student in db.Student do sortBy student.Name select student }`
`sortByDescending`	Сортирует выбранные элементы на данный момент в порядке убывания по заданному ключу сортировки. `query { for student in db.Student do sortByDescending student.Name select student }`
`thenBy`	Выполняет последующее упорядочение элементов, выбранных на данный момент в возрастающем порядке по заданному ключу сортировки. Этот оператор можно использовать только после `sortBy`, `sortByDescending`, `thenBy` или `thenByDescending`. `query { for student in db.Student do where student.Age.HasValue sortBy student.Age.Value thenBy student.Name select student }`

`thenByDescending`	Выполняет последующее упорядочение элементов, выбранных на данный момент в порядке убывания по заданному ключу сортировки. Этот оператор можно использовать только после `sortBy`, `sortByDescending`, `thenBy` или `thenByDescending`. `query {` `  for student in db.Student do` `  where student.Age.HasValue` `  sortBy student.Age.Value` `  thenByDescending student.Name` `  select student` `}`
`groupValBy`	Выбирает значение для каждого выбранного элемента на данный момент и группирует элементы по заданному ключу. `query {` `  for student in db.Student do` `  groupValBy student.Name student.Age into g` `  select (g, g.Key, g.Count())` `}`
`join`	Сопоставляет два набора выбранных значений на основе совпадающих ключей. Обратите внимание, что порядок клавиш вокруг знака = в выражении Join является существенным. Во всех объединениях, если строка разделяется после `->` символа, отступ должен быть меньше, чем ключевое слово `for`. `query {` `  for student in db.Student do` `  join selection in db.CourseSelection` `  on (student.StudentID =` `selection.StudentID)` `  select (student, selection)` `}`

`groupJoin`	Сопоставляет два набора выбранных значений на основе совпадающих ключей и группирует результаты. Обратите внимание, что порядок клавиш вокруг знака = в выражении Join является существенным. `query { for student in db.Student do groupJoin courseSelection in db.CourseSelection on (student.StudentID = courseSelection.StudentID) into g for courseSelection in g do join course in db.Course on (courseSelection.CourseID = course.CourseID) select (student.Name, course.CourseName) }`
`leftOuterJoin`	Сопоставляет два набора выбранных значений на основе совпадающих ключей и группирует результаты. Если какая-либо из групп пуста, вместо нее используется группа с одним значением по умолчанию. Обратите внимание, что порядок клавиш вокруг знака = в выражении Join является существенным. `query { for student in db.Student do leftOuterJoin selection in db.CourseSelection on (student.StudentID = selection.StudentID) into result for selection in result.DefaultIfEmpty() do select (student, selection) }`
`sumByNullable`	Выбирает значение, допускающее значение null, для каждого выбранного элемента на данный момент и возвращает сумму этих значений. Если значение NULL не имеет значения, оно игнорируется. `query { for student in db.Student do sumByNullable student.Age }`
`minByNullable`	Выбирает значение, допускающее значение null, для каждого выбранного элемента до настоящего момента и возвращает минимальное из этих значений. Если значение NULL не имеет значения, оно игнорируется. `query { for student in db.Student do minByNullable student.Age }`

`maxByNullable`	Выбирает значение, допускающее значение null, для каждого выбранного элемента на данный момент и возвращает максимальное из этих значений. Если значение NULL не имеет значения, оно игнорируется. `query { for student in db.Student do maxByNullable student.Age }`
`averageByNullable`	Выбирает значение, допускающее значение null, для каждого выбранного элемента на данный момент и возвращает среднее из этих значений. Если значение NULL не имеет значения, оно игнорируется. `query { for student in db.Student do averageByNullable (Nullable.float student.Age) }`
`averageBy`	Выбирает значение для каждого выбранного элемента на данный момент и возвращает среднее из этих значений. `query { for student in db.Student do averageBy (float student.StudentID) }`
`distinct`	Выбирает отдельные элементы из выбранных ранее элементов. `query { for student in db.Student do join selection in db.CourseSelection on (student.StudentID = selection.StudentID) distinct }`

`exists`	Определяет, удовлетворяет ли ни одному элементу, выбранному до сих пор условие. `query { for student in db.Student do where (query { for courseSelection in db.CourseSelection do exists (courseSelection.StudentID = student.StudentID) }) select student }`
`find`	Выбирает первый выбранный элемент, удовлетворяющий указанному условию. `query { for student in db.Student do find (student.Name = "Abercrombie, Kim") }`
`all`	Определяет, соответствуют ли все элементы, выбранные на данный момент, определенному условию. `query { for student in db.Student do all (SqlMethods.Like(student.Name, "%,%")) }`
`head`	Выбирает первый элемент из выбранных на данный момент элементов. `query { for student in db.Student do head }`
`nth`	Выбирает элемент по указанному индексу среди выбранных до сих пор. `query { for numbers in data do nth 3 }`

`skip`	Обходит указанное число выбранных элементов на данный момент, а затем выбирает оставшиеся элементы. `query { for student in db.Student do skip 1 }`
`skipWhile`	Обходит элементы в последовательности, пока заданное условие имеет значение true, а затем выбирает оставшиеся элементы. `query { for number in data do skipWhile (number < 3) select student }`
`sumBy`	Выбирает значение для каждого выбранного элемента на данный момент и возвращает сумму этих значений. `query { for student in db.Student do sumBy student.StudentID }`
`take`	Выбирает указанное количество смежных элементов из выбранных выше. `query { for student in db.Student do select student take 2 }`
`takeWhile`	Выбирает элементы из последовательности, пока заданное условие имеет значение true, а затем пропускает оставшиеся элементы. `query { for number in data do takeWhile (number < 10) }`

`sortByNullable`	Сортирует выбранные элементы на данный момент в порядке возрастания по заданному ключу сортировки, допускающему значение null. `query { for student in db.Student do sortByNullable student.Age select student }`
`sortByNullableDescending`	Сортирует элементы, выбранные на данный момент в порядке убывания, по заданному ключу сортировки, допускающему значение null. `query { for student in db.Student do sortByNullableDescending student.Age select student }`
`thenByNullable`	Выполняет последующее упорядочение элементов, выбранных на данный момент в возрастающем порядке по заданному ключу сортировки, допускающему значение null. Этот оператор может использоваться только сразу после `sortBy`, `sortByDescending`, `thenBy` или `thenByDescending` или их разновидностей, допускающих значение null. `query { for student in db.Student do sortBy student.Name thenByNullable student.Age select student }`
`thenByNullableDescending`	Выполняет последующее упорядочение элементов, выбранных на данный момент в убывающем порядке по заданному ключу сортировки, допускающему значение null. Этот оператор может использоваться только сразу после `sortBy`, `sortByDescending`, `thenBy` или `thenByDescending` или их разновидностей, допускающих значение null. `query { for student in db.Student do sortBy student.Name thenByNullableDescending student.Age select student }`

Сравнение выражений запросов Transact-SQL и F#

В следующей таблице показаны некоторые распространенные запросы Transact-SQL и их эквиваленты F#в.

Код в этой таблице также предполагает ту же базу данных, что и Предыдущая таблица, и тот же начальный код для настройки поставщика типов.

Таблица 2. Выражения запросов Transact-SQL и F#

TRANSACT-SQL (БЕЗ УЧЕТА РЕГИСТРА)	F#ВЫРАЖЕНИЕ ЗАПРОСА (С УЧЕТОМ РЕГИСТРА)
Выбрать все поля из таблицы. `SELECT * FROM Student`	`// All students.` `query {` `  for student in db.Student do` `  select student` `}`
Подсчет записей в таблице. `SELECT COUNT( * ) FROM Student`	`// Count of students.` `query {` `  for student in db.Student do` `  count` `}`
`EXISTS` `SELECT * FROM Student` `WHERE EXISTS` `  (SELECT * FROM CourseSelection` `  WHERE CourseSelection.StudentID = Student.StudentID)`	`// Find students who have signed up at least one course.` `query {` `  for student in db.Student do` `  where` `  (query {` `  for courseSelection in db.CourseSelection do` `  exists (courseSelection.StudentID = student.StudentID) })` `  select student` `}`
Группирование `SELECT Student.Age, COUNT( * ) FROM Student` `GROUP BY Student.Age`	`// Group by age and count.` `query {` `  for n in db.Student do` `  groupBy n.Age into g` `  select (g.Key, g.Count())` `}` `// OR` `query {` `  for n in db.Student do` `  groupValBy n.Age n.Age into g` `  select (g.Key, g.Count())` `}`
Группирование с условием. `SELECT Student.Age, COUNT( * )` `FROM Student` `GROUP BY Student.Age` `HAVING student.Age > 10`	`// Group students by age where age > 10.` `query {` `  for student in db.Student do` `  groupBy student.Age into g` `  where (g.Key.HasValue && g.Key.Value > 10)` `  select (g.Key, g.Count())` `}`

Группирование с условием подсчета.

```
SELECT Student.Age, COUNT( * )
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
```

```
// Group students by age and count number of
students
// at each age with more than 1 student.
query {
    for student in db.Student do
    groupBy student.Age into group
    where (group.Count() > 1)
    select (group.Key, group.Count())
}
```

Группирование, подсчет и суммирование.

```
SELECT Student.Age, COUNT( * ),
SUM(Student.Age) as total
FROM Student
GROUP BY Student.Age
```

```
// Group students by age and sum ages.
query {
    for student in db.Student do
    groupBy student.Age into g
    let total =
        query {
            for student in g do
            sumByNullable student.Age
        }
    select (g.Key, g.Count(), total)
}
```

Группирование, подсчет и упорядочивание по количеству.

```
SELECT Student.Age, COUNT( * ) as myCount
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
ORDER BY COUNT( * ) DESC
```

```
// Group students by age, count number of
students
// at each age, and display all with count > 1
// in descending order of count.
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Count() > 1)
    sortByDescending (g.Count())
    select (g.Key, g.Count())
}
```

`IN` набор указанных значений

```
SELECT *
FROM Student
WHERE Student.StudentID IN (1, 2, 5, 10)
```

```
// Select students where studentID is one of a
given list.
let idQuery =
    query {
        for id in [1; 2; 5; 10] do
        select id
    }
query {
    for student in db.Student do
    where (idQuery.Contains(student.StudentID))
    select student
}
```

`LIKE` и `TOP`.

```
-- '_e%' matches strings where the second
character is 'e'
SELECT TOP 2 * FROM Student
WHERE Student.Name LIKE '_e%'
```

```
// Look for students with Name match _e% pattern
and take first two.
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "_e%")
)
    select student
    take 2
}
```

`LIKE` с набором совпадений шаблона.

```
-- '[abc]%' matches strings where the first
character is
-- 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[abc]%'
```

```
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "
[abc]%") )
    select student
}
```

`LIKE` с шаблоном исключения Set.

```
-- '[^abc]%' matches strings where the first
character is
-- not 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
// Look for students with name matching [^abc]%%
pattern.
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "
[^abc]%") )
    select student
}
```

`LIKE` в одном поле, но выберите другое поле.

```
SELECT StudentID AS ID FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
query {
    for n in db.Student do
    where (SqlMethods.Like( n.Name, "[^abc]%") )
    select n.StudentID
}
```

`LIKE` с поиском подстроки.

```
SELECT * FROM Student
WHERE Student.Name like '%A%'
```

```
// Using Contains as a query filter.
query {
    for student in db.Student do
    where (student.Name.Contains("a"))
    select student
}
```

Простая `JOIN` с двумя таблицами.

```
SELECT * FROM Student
JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
// Join Student and CourseSelection tables.
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID =
selection.StudentID)
    select (student, selection)
}
```

`LEFT JOIN` с двумя таблицами.

```
SELECT * FROM Student
LEFT JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
//Left Join Student and CourseSelection tables.
query {
    for student in db.Student do
    leftOuterJoin selection in
db.CourseSelection
        on (student.StudentID =
selection.StudentID) into result
    for selection in result.DefaultIfEmpty() do
    select (student, selection)
}
```

`JOIN` с `COUNT`

```
SELECT COUNT( * ) FROM Student
JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
// Join with count.
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    count
}
```

`DISTINCT`

```
SELECT DISTINCT StudentID FROM CourseSelection
```

```
// Join with distinct.
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID =
selection.StudentID)
    distinct
}
```

Число различных объектов.

```
SELECT DISTINCT COUNT(StudentID) FROM
CourseSelection
```

```
// Join with distinct and count.
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    distinct
    count
}
```

`BETWEEN`

```
SELECT * FROM Student
WHERE Student.Age BETWEEN 10 AND 15
```

```
// Selecting students with ages between 10 and
15.
query {
    for student in db.Student do
    where (student.Age ?>= 10 && student.Age ?<
15)
    select student
}
```

`OR`

```
SELECT * FROM Student
WHERE Student.Age = 11 OR Student.Age = 12
```

```
// Selecting students with age that's either 11
or 12.
query {
    for student in db.Student do
    where (student.Age.Value = 11 ||
student.Age.Value = 12)
    select student
}
```

`OR` с упорядочением

```
SELECT * FROM Student
WHERE Student.Age = 12 OR Student.Age = 13
ORDER BY Student.Age DESC
```

```
// Selecting students in a certain age range and
sorting.
query {
    for n in db.Student do
    where (n.Age.Value = 12 || n.Age.Value = 13)
    sortByNullableDescending n.Age
    select n
}
```

`TOP` , `OR` и упорядочение.

```
SELECT TOP 2 student.Name FROM Student
WHERE Student.Age = 11 OR Student.Age = 12
ORDER BY Student.Name DESC
```

```
// Selecting students with certain ages,
// taking account of the possibility of nulls.
query {
    for student in db.Student do
    where
        ((student.Age.HasValue &&
student.Age.Value = 11) ||
         (student.Age.HasValue &&
student.Age.Value = 12))
    sortByDescending student.Name
    select student.Name
    take 2
}
```

`UNION` двух запросов.

```
SELECT * FROM Student
UNION
SELECT * FROM lastStudent
```

```
let query1 =
    query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

let query2 =
    query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

query2.Union (query1)
```

Пересечение двух запросов.

```
SELECT * FROM Student
INTERSECT
SELECT * FROM LastStudent
```

```
let query1 =
    query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

let query2 =
    query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

query1.Intersect(query2)
```

условие `CASE`.

```
SELECT student.StudentID,
CASE Student.Age
  WHEN -1 THEN 100
  ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special value.
query {
    for student in db.Student do
    select
        (if student.Age.HasValue &&
student.Age.Value = -1 then
             (student.StudentID,
System.Nullable<int>(100), student.Age)
         else (student.StudentID, student.Age,
student.Age))
}
```

Несколько вариантов.

```
SELECT Student.StudentID,
CASE Student.Age
  WHEN -1 THEN 100
  WHEN 0 THEN 1000
  ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special values.
query {
    for student in db.Student do
    select
        (if student.Age.HasValue &&
student.Age.Value = -1 then
             (student.StudentID,
System.Nullable<int>(100), student.Age)
         elif student.Age.HasValue &&
student.Age.Value = 0 then
             (student.StudentID,
System.Nullable<int>(1000), student.Age)
         else (student.StudentID, student.Age,
student.Age))
}
```

Несколько таблиц

```
SELECT * FROM Student, Course
```

```
// Multiple table select.
query {
    for student in db.Student do
    for course in db.Course do
    select (student, course)
}
```

Множественные объединения.

```
SELECT Student.Name, Course.CourseName
FROM Student
JOIN CourseSelection
ON CourseSelection.StudentID =
Student.StudentID
JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Multiple joins.
query {
    for student in db.Student do
    join courseSelection in db.CourseSelection
        on (student.StudentID =
courseSelection.StudentID)
    join course in db.Course
        on (courseSelection.CourseID =
course.CourseID)
    select (student.Name, course.CourseName)
}
```

Несколько левых внешних соединений.

```
SELECT Student.Name, Course.CourseName
FROM Student
LEFT OUTER JOIN CourseSelection
ON CourseSelection.StudentID =
Student.StudentID
LEFT OUTER JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Using leftOuterJoin with multiple joins.
query {
    for student in db.Student do
    leftOuterJoin courseSelection in
db.CourseSelection
        on (student.StudentID =
courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty()
do
    leftOuterJoin course in db.Course
        on (courseSelection.CourseID =
course.CourseID) into g2
    for course in g2.DefaultIfEmpty() do
    select (student.Name, course.CourseName)
}
```

Следующий код можно использовать для создания образца базы данных для этих примеров.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

USE [master];
GO

IF EXISTS (SELECT * FROM sys.databases WHERE name = 'MyDatabase')
DROP DATABASE MyDatabase;
GO

-- Create the MyDatabase database.
CREATE DATABASE MyDatabase COLLATE SQL_Latin1_General_CP1_CI_AS;
GO

-- Specify a simple recovery model
-- to keep the log growth to a minimum.
ALTER DATABASE MyDatabase
SET RECOVERY SIMPLE;
GO

USE MyDatabase;
GO

CREATE TABLE [dbo].[Course] (
[CourseID]   INT           NOT NULL,
[CourseName] NVARCHAR (50) NOT NULL,
PRIMARY KEY CLUSTERED ([CourseID] ASC)
);

CREATE TABLE [dbo].[Student] (
[StudentID] INT           NOT NULL,
```

```
[StudentID] INT           NOT NULL,
[Name]      NVARCHAR (50) NOT NULL,
[Age]       INT           NULL,
PRIMARY KEY CLUSTERED ([StudentID] ASC)
);

CREATE TABLE [dbo].[CourseSelection] (
[ID]        INT NOT NULL,
[StudentID] INT NOT NULL,
[CourseID]  INT NOT NULL,
PRIMARY KEY CLUSTERED ([ID] ASC),
CONSTRAINT [FK_CourseSelection_ToTable] FOREIGN KEY ([StudentID]) REFERENCES [dbo].[Student] ([StudentID]) ON
DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT [FK_CourseSelection_Course_1] FOREIGN KEY ([CourseID]) REFERENCES [dbo].[Course] ([CourseID]) ON
DELETE NO ACTION ON UPDATE NO ACTION
);

CREATE TABLE [dbo].[LastStudent] (
[StudentID] INT           NOT NULL,
[Name]      NVARCHAR (50) NOT NULL,
[Age]       INT           NULL,
PRIMARY KEY CLUSTERED ([StudentID] ASC)
);

-- Insert data into the tables.
USE MyDatabase
INSERT INTO Course (CourseID, CourseName)
VALUES(1, 'Algebra I');
INSERT INTO Course (CourseID, CourseName)
VALUES(2, 'Trigonometry');
INSERT INTO Course (CourseID, CourseName)
VALUES(3, 'Algebra II');
INSERT INTO Course (CourseID, CourseName)
VALUES(4, 'History');
INSERT INTO Course (CourseID, CourseName)
VALUES(5, 'English');
INSERT INTO Course (CourseID, CourseName)
VALUES(6, 'French');
INSERT INTO Course (CourseID, CourseName)
VALUES(7, 'Chinese');

INSERT INTO Student (StudentID, Name, Age)
VALUES(1, 'Abercrombie, Kim', 10);
INSERT INTO Student (StudentID, Name, Age)
VALUES(2, 'Abolrous, Hazen', 14);
INSERT INTO Student (StudentID, Name, Age)
VALUES(3, 'Hance, Jim', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(4, 'Adams, Terry', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(5, 'Hansen, Claus', 11);
INSERT INTO Student (StudentID, Name, Age)
VALUES(6, 'Penor, Lori', 13);
INSERT INTO Student (StudentID, Name, Age)
VALUES(7, 'Perham, Tom', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(8, 'Peng, Yun-Feng', NULL);

INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(1, 1, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(2, 1, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(3, 1, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(4, 2, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(5, 2, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(6, 2, 6);
```

```
VALUES(6, 2, 6);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(7, 2, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(8, 3, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(9, 3, 1);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(10, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(11, 4, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(12, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(13, 5, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(14, 5, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(15, 7, 3);
```

Следующий код содержит образец кода, представленный в этом разделе.

```
#if INTERACTIVE
#r "FSharp.Data.TypeProviders.dll"
#r "System.Data.dll"
#r "System.Data.Linq.dll"
#endif
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq

type schema = SqlDataConnection<"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;">

let db = schema.GetDataContext()

let data = [1; 5; 7; 11; 18; 21]

type Nullable<'T when 'T : ( new : unit -> 'T) and 'T : struct and 'T :> ValueType > with
    member this.Print() =
        if this.HasValue then this.Value.ToString()
        else "NULL"

printfn "\ncontains query operator"
query {
    for student in db.Student do
    select student.Age.Value
    contains 11
}
|> printfn "Is at least one student age 11? %b"

printfn "\ncount query operator"
query {
    for student in db.Student do
    select student
    count
}
|> printfn "Number of students: %d"

printfn "\nlast query operator."
let num =
    query {
        for number in data do
        sortBy number
        last
    }
printfn "Last number: %d" num
```

```
printfn "Last number: %d" num

open Microsoft.FSharp.Linq

printfn "\nlastOrDefault query operator."
query {
    for number in data do
    sortBy number
    lastOrDefault
}
|> printfn "lastOrDefault: %d"

printfn "\nexactlyOne query operator."
let student2 =
    query {
        for student in db.Student do
        where (student.StudentID = 1)
        select student
        exactlyOne
    }
printfn "Student with StudentID = 1 is %s" student2.Name

printfn "\nexactlyOneOrDefault query operator."
let student3 =
    query {
        for student in db.Student do
        where (student.StudentID = 1)
        select student
        exactlyOneOrDefault
    }
printfn "Student with StudentID = 1 is %s" student3.Name

printfn "\nheadOrDefault query operator."
let student4 =
    query {
        for student in db.Student do
        select student
        headOrDefault
    }
printfn "head student is %s" student4.Name

printfn "\nselect query operator."
query {
    for student in db.Student do
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nwhere query operator."
query {
    for student in db.Student do
    where (student.StudentID > 4)
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nminBy query operator."
let student5 =
    query {
        for student in db.Student do
        minBy student.StudentID
    }

printfn "\nmaxBy query operator."
let student6 =
    query {
        for student in db.Student do
        maxBy student.StudentID
    }
```

```
printfn "\ngroupBy query operator."
query {
    for student in db.Student do
    groupBy student.Age into g
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "Age: %s Count at that age: %d" (age.Print()) count)

printfn "\nsortBy query operator."
query {
    for student in db.Student do
    sortBy student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nsortByDescending query operator."
query {
    for student in db.Student do
    sortByDescending student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nthenBy query operator."
query {
    for student in db.Student do
    where student.Age.HasValue
    sortBy student.Age.Value
    thenBy student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\nthenByDescending query operator."
query {
    for student in db.Student do
    where student.Age.HasValue
    sortBy student.Age.Value
    thenByDescending student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\ngroupValBy query operator."
query {
    for student in db.Student do
    groupValBy student.Name student.Age into g
    select (g, g.Key, g.Count())
}
|> Seq.iter (fun (group, age, count) ->
    printfn "Age: %s Count at that age: %d" (age.Print()) count
    group |> Seq.iter (fun name -> printfn "Name: %s" name))

printfn "\n sumByNullable query operator"
query {
    for student in db.Student do
    sumByNullable student.Age
}
|> (fun sum -> printfn "Sum of ages: %s" (sum.Print()))

printfn "\n minByNullable"
query {
    for student in db.Student do
    minByNullable student.Age
}
|> (fun age -> printfn "Minimum age: %s" (age.Print()))
```

```
printfn "\n maxByNullable"
query {
    for student in db.Student do
    maxByNullable student.Age
}
|> (fun age -> printfn "Maximum age: %s" (age.Print()))

printfn "\n averageBy"
query {
    for student in db.Student do
    averageBy (float student.StudentID)
}
|> printfn "Average student ID: %f"

printfn "\n averageByNullable"
query {
    for student in db.Student do
    averageByNullable (Nullable.float student.Age)
}
|> (fun avg -> printfn "Average age: %s" (avg.Print()))

printfn "\n find query operator"
query {
    for student in db.Student do
    find (student.Name = "Abercrombie, Kim")
}
|> (fun student -> printfn "Found a match with StudentID = %d" student.StudentID)

printfn "\n all query operator"
query {
    for student in db.Student do
    all (SqlMethods.Like(student.Name, "%,%"))
}
|> printfn "Do all students have a comma in the name? %b"

printfn "\n head query operator"
query {
    for student in db.Student do
    head
}
|> (fun student -> printfn "Found the head student with StudentID = %d" student.StudentID)

printfn "\n nth query operator"
query {
    for numbers in data do
    nth 3
}
|> printfn "Third number is %d"

printfn "\n skip query operator"
query {
    for student in db.Student do
    skip 1
}
|> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

printfn "\n skipWhile query operator"
query {
    for number in data do
    skipWhile (number < 3)
    select number
}
|> Seq.iter (fun number -> printfn "Number = %d" number)

printfn "\n sumBy query operator"
query {
    for student in db.Student do
    sumBy student.StudentID
}
```

```
|> printfn "Sum of student IDs: %d"

printfn "\n take query operator"
query {
    for student in db.Student do
    select student
    take 2
}
|> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

printfn "\n takeWhile query operator"
query {
    for number in data do
    takeWhile (number < 10)
}
|> Seq.iter (fun number -> printfn "Number = %d" number)

printfn "\n sortByNullable query operator"
query {
    for student in db.Student do
    sortByNullable student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n sortByNullableDescending query operator"
query {
    for student in db.Student do
    sortByNullableDescending student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n thenByNullable query operator"
query {
    for student in db.Student do
    sortBy student.Name
    thenByNullable student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n thenByNullableDescending query operator"
query {
    for student in db.Student do
    sortBy student.Name
    thenByNullableDescending student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "All students: "
query {
    for student in db.Student do
    select student
}
|> Seq.iter (fun student -> printfn "%s %d %s" student.Name student.StudentID (student.Age.Print()))

printfn "\nCount of students: "
query {
    for student in db.Student do
    count
}
|> (fun count -> printfn "Student count: %d" count)
```

```
printfn "\nExists."
query {
    for student in db.Student do
    where
        (query {
            for courseSelection in db.CourseSelection do
            exists (courseSelection.StudentID = student.StudentID) })
    select student
}
|> Seq.iter (fun student -> printfn "%A" student.Name)

printfn "\n Group by age and count"
query {
    for n in db.Student do
    groupBy n.Age into g
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "%s %d" (age.Print()) count)

printfn "\n Group value by age."
query {
    for n in db.Student do
    groupValBy n.Age n.Age into g
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "%s %d" (age.Print()) count)

printfn "\nGroup students by age where age > 10."
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Key.HasValue && g.Key.Value > 10)
    select (g, g.Key)
}
|> Seq.iter (fun (students, age) ->
    printfn "Age: %s" (age.Value.ToString())
    students
    |> Seq.iter (fun student -> printfn "%s" student.Name))

printfn "\nGroup students by age and print counts of number of students at each age with more than 1
student."
query {
    for student in db.Student do
    groupBy student.Age into group
    where (group.Count() > 1)
    select (group.Key, group.Count())
}
|> Seq.iter (fun (age, ageCount) ->
    printfn "Age: %s Count: %d" (age.Print()) ageCount)

printfn "\nGroup students by age and sum ages."
query {
    for student in db.Student do
    groupBy student.Age into g
    let total = query { for student in g do sumByNullable student.Age }
    select (g.Key, g.Count(), total)
}
|> Seq.iter (fun (age, count, total) ->
    printfn "Age: %d" (age.GetValueOrDefault())
    printfn "Count: %d" count
    printfn "Total years: %s" (total.ToString()))

printfn "\nGroup students by age and count number of students at each age, and display all with count > 1 in
descending order of count."
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Count() > 1)
    sortByDescending (g.Count())
```

```
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, myCount) ->
    printfn "Age: %s" (age.Print())
    printfn "Count: %d" myCount)

printfn "\n Select students from a set of IDs"
let idList = [1; 2; 5; 10]
let idQuery =
    query { for id in idList do select id }
query {
    for student in db.Student do
    where (idQuery.Contains(student.StudentID))
    select student
}
|> Seq.iter (fun student ->
    printfn "Name: %s" student.Name)

printfn "\nLook for students with Name match _e%% pattern and take first two."
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "_e%") )
    select student
    take 2
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with Name matching [abc]%% pattern."
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "[abc]%") )
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern."
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "[^abc]%") )
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern and select ID."
query {
    for n in db.Student do
    where (SqlMethods.Like( n.Name, "[^abc]%") )
    select n.StudentID
}
|> Seq.iter (fun id -> printfn "%d" id)

printfn "\n Using Contains as a query filter."
query {
    for student in db.Student do
    where (student.Name.Contains("a"))
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nSearching for names from a list."
let names = [|"a";"b";"c"|]
query {
    for student in db.Student do
    if names.Contains (student.Name) then select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nJoin Student and CourseSelection tables."
query {
```

```
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID = selection.StudentID)
    select (student, selection)
}
|> Seq.iter (fun (student, selection) -> printfn "%d %s %d" student.StudentID student.Name
selection.CourseID)

printfn "\nLeft Join Student and CourseSelection tables."
query {
    for student in db.Student do
    leftOuterJoin selection in db.CourseSelection
        on (student.StudentID = selection.StudentID) into result
    for selection in result.DefaultIfEmpty() do
    select (student, selection)
}
|> Seq.iter (fun (student, selection) ->
    let selectionID, studentID, courseID =
        match selection with
        | null -> "NULL", "NULL", "NULL"
        | sel -> (sel.ID.ToString(), sel.StudentID.ToString(), sel.CourseID.ToString())
    printfn "%d %s %d %s %s %s" student.StudentID student.Name (student.Age.GetValueOrDefault()) selectionID
studentID courseID)

printfn "\nJoin with count"
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    count
}
|> printfn "%d"

printfn "\n Join with distinct."
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID = selection.StudentID)
    distinct
}
|> Seq.iter (fun (student, selection) -> printfn "%s %d" student.Name selection.CourseID)

printfn "\n Join with distinct and count."
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    distinct
    count
}
|> printfn "%d"

printfn "\n Selecting students with age between 10 and 15."
query {
    for student in db.Student do
    where (student.Age.Value >= 10 && student.Age.Value < 15)
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students with age either 11 or 12."
query {
    for student in db.Student do
    where (student.Age.Value = 11 || student.Age.Value = 12)
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students in a certain age range and sorting."
```

```
printfn "\n Selecting students in a certain age range and sorting."
query {
    for n in db.Student do
    where (n.Age.Value = 12 || n.Age.Value = 13)
    sortByNullableDescending n.Age
    select n
}
|> Seq.iter (fun student -> printfn "%s %s" student.Name (student.Age.Print()))

printfn "\n Selecting students with certain ages, taking account of possibility of nulls."
query {
    for student in db.Student do
    where
        ((student.Age.HasValue && student.Age.Value = 11) ||
         (student.Age.HasValue && student.Age.Value = 12))
    sortByDescending student.Name
    select student.Name
    take 2
}
|> Seq.iter (fun name -> printfn "%s" name)

printfn "\n Union of two queries."
module Queries =
    let query1 = query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

    let query2 = query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

    query2.Union (query1)
    |> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

printfn "\n Intersect of two queries."
module Queries2 =
    let query1 = query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

    let query2 = query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

    query1.Intersect(query2)
    |> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

printfn "\n Using if statement to alter results for special value."
query {
    for student in db.Student do
    select
        (if student.Age.HasValue && student.Age.Value = -1 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
         else (student.StudentID, student.Age, student.Age))
}
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

printfn "\n Using if statement to alter results special values."
query {
    for student in db.Student do
    select
        (if student.Age.HasValue && student.Age.Value = -1 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
         elif student.Age.HasValue && student.Age.Value = 0 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
```

```
         else (student.StudentID, student.Age, student.Age))
}
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

printfn "\n Multiple table select."
query {
    for student in db.Student do
    for course in db.Course do
    select (student, course)
}
|> Seq.iteri (fun index (student, course) ->
    if index = 0 then
        printfn "StudentID Name Age CourseID CourseName"
    printfn "%d %s %s %d %s" student.StudentID student.Name (student.Age.Print()) course.CourseID
course.CourseName)

printfn "\nMultiple Joins"
query {
    for student in db.Student do
    join courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID)
    join course in db.Course
        on (courseSelection.CourseID = course.CourseID)
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)

printfn "\nMultiple Left Outer Joins"
query {
    for student in db.Student do
    leftOuterJoin courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty() do
    leftOuterJoin course in db.Course
        on (courseSelection.CourseID = course.CourseID) into g2
    for course in g2.DefaultIfEmpty() do
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)
```

Ниже приведен полный результат выполнения этого кода в F# интерактивном режиме.

```
--> Referenced 'C:\Program Files (x86)\Reference Assemblies\Microsoft\FSharp\3.0\Runtime\v4.0\Type
Providers\FSharp.Data.TypeProviders.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.Linq.dll'

contains query operator
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp5E3C.dll'...
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp611A.dll'...
Is at least one student age 11? true

count query operator
Number of students: 8

last query operator.
Last number: 21

lastOrDefault query operator.
lastOrDefault: 21

exactlyOne query operator.
Student with StudentID = 1 is Abercrombie, Kim

exactlyOneOrDefault query operator.
Student with StudentID = 1 is Abercrombie, Kim
```

```
Student with StudentID = 1 is Abercrombie, Kim

headOrDefault query operator.
head student is Abercrombie, Kim

select query operator.
StudentID, Name: 1 Abercrombie, Kim
StudentID, Name: 2 Abolrous, Hazen
StudentID, Name: 3 Hance, Jim
StudentID, Name: 4 Adams, Terry
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 6 Penor, Lori
StudentID, Name: 7 Perham, Tom
StudentID, Name: 8 Peng, Yun-Feng

where query operator.
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 6 Penor, Lori
StudentID, Name: 7 Perham, Tom
StudentID, Name: 8 Peng, Yun-Feng

minBy query operator.

maxBy query operator.

groupBy query operator.
Age: NULL Count at that age: 1
Age: 10 Count at that age: 1
Age: 11 Count at that age: 1
Age: 12 Count at that age: 3
Age: 13 Count at that age: 1
Age: 14 Count at that age: 1

sortBy query operator.
StudentID, Name: 1 Abercrombie, Kim
StudentID, Name: 2 Abolrous, Hazen
StudentID, Name: 4 Adams, Terry
StudentID, Name: 3 Hance, Jim
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 8 Peng, Yun-Feng
StudentID, Name: 6 Penor, Lori
StudentID, Name: 7 Perham, Tom

sortByDescending query operator.
StudentID, Name: 7 Perham, Tom
StudentID, Name: 6 Penor, Lori
StudentID, Name: 8 Peng, Yun-Feng
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 3 Hance, Jim
StudentID, Name: 4 Adams, Terry
StudentID, Name: 2 Abolrous, Hazen
StudentID, Name: 1 Abercrombie, Kim

thenBy query operator.
StudentID, Name: 10 Abercrombie, Kim
StudentID, Name: 11 Hansen, Claus
StudentID, Name: 12 Adams, Terry
StudentID, Name: 12 Hance, Jim
StudentID, Name: 12 Perham, Tom
StudentID, Name: 13 Penor, Lori
StudentID, Name: 14 Abolrous, Hazen

thenByDescending query operator.
StudentID, Name: 10 Abercrombie, Kim
StudentID, Name: 11 Hansen, Claus
StudentID, Name: 12 Perham, Tom
StudentID, Name: 12 Hance, Jim
StudentID, Name: 12 Adams, Terry
StudentID, Name: 13 Penor, Lori
```

```
StudentID, Name: 14 Abolrous, Hazen

groupValBy query operator.
Age: NULL Count at that age: 1
Name: Peng, Yun-Feng
Age: 10 Count at that age: 1
Name: Abercrombie, Kim
Age: 11 Count at that age: 1
Name: Hansen, Claus
Age: 12 Count at that age: 3
Name: Hance, Jim
Name: Adams, Terry
Name: Perham, Tom
Age: 13 Count at that age: 1
Name: Penor, Lori
Age: 14 Count at that age: 1
Name: Abolrous, Hazen

sumByNullable query operator
Sum of ages: 84

minByNullable
Minimum age: 10

maxByNullable
Maximum age: 14

averageBy
Average student ID: 4.500000

averageByNullable
Average age: 12

find query operator
Found a match with StudentID = 1

all query operator
Do all students have a comma in the name? true

head query operator
Found the head student with StudentID = 1

nth query operator
Third number is 11

skip query operator
StudentID = 2
StudentID = 3
StudentID = 4
StudentID = 5
StudentID = 6
StudentID = 7
StudentID = 8

skipWhile query operator
Number = 5
Number = 7
Number = 11
Number = 18
Number = 21

sumBy query operator
Sum of student IDs: 36

take query operator
StudentID = 1
StudentID = 2

takeWhile query operator
```

```
Number = 1
Number = 5
Number = 7

sortByNullable query operator
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 7 Perham, Tom 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 2 Abolrous, Hazen 14

sortByNullableDescending query operator
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 8 Peng, Yun-Feng NULL

thenByNullable query operator
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12

thenByNullableDescending query operator
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12
All students:
Abercrombie, Kim 1 10
Abolrous, Hazen 2 14
Hance, Jim 3 12
Adams, Terry 4 12
Hansen, Claus 5 11
Penor, Lori 6 13
Perham, Tom 7 12
Peng, Yun-Feng 8 NULL

Count of students:
Student count: 8

Exists.
"Abercrombie, Kim"
"Abolrous, Hazen"
"Hance, Jim"
"Adams, Terry"
"Hansen, Claus"
"Perham, Tom"

Group by age and count
NULL 1
10 1
11 1
12 3
```

```
13 1
14 1

Group value by age.
NULL 1
10 1
11 1
12 3
13 1
14 1

Group students by age where age > 10.
Age: 11
Hansen, Claus
Age: 12
Hance, Jim
Adams, Terry
Perham, Tom
Age: 13
Penor, Lori
Age: 14
Abolrous, Hazen

Group students by age and print counts of number of students at each age with more than 1 student.
Age: 12 Count: 3

Group students by age and sum ages.
Age: 0
Count: 1
Total years:
Age: 10
Count: 1
Total years: 10
Age: 11
Count: 1
Total years: 11
Age: 12
Count: 3
Total years: 36
Age: 13
Count: 1
Total years: 13
Age: 14
Count: 1
Total years: 14

Group students by age and count number of students at each age, and display all with count > 1 in descending
order of count.
Age: 12
Count: 3

Select students from a set of IDs
Name: Abercrombie, Kim
Name: Abolrous, Hazen
Name: Hansen, Claus

Look for students with Name match _e% pattern and take first two.
Penor, Lori
Perham, Tom

Look for students with Name matching [abc]% pattern.
Abercrombie, Kim
Abolrous, Hazen
Adams, Terry

Look for students with name matching [^abc]% pattern.
Hance, Jim
Hansen, Claus
Penor, Lori
```

```
Perham, Tom
Peng, Yun-Feng

Look for students with name matching [^abc]% pattern and select ID.
3
5
6
7
8

Using Contains as a query filter.
Abercrombie, Kim
Abolrous, Hazen
Hance, Jim
Adams, Terry
Hansen, Claus
Perham, Tom

Searching for names from a list.

Join Student and CourseSelection tables.
2 Abolrous, Hazen 2
3 Hance, Jim 3
5 Hansen, Claus 5
2 Abolrous, Hazen 2
5 Hansen, Claus 5
6 Penor, Lori 6
3 Hance, Jim 3
2 Abolrous, Hazen 2
1 Abercrombie, Kim 1
2 Abolrous, Hazen 2
5 Hansen, Claus 5
2 Abolrous, Hazen 2
3 Hance, Jim 3
2 Abolrous, Hazen 2
3 Hance, Jim 3

Left Join Student and CourseSelection tables.
1 Abercrombie, Kim 10 9 3 1
2 Abolrous, Hazen 14 1 1 2
2 Abolrous, Hazen 14 4 2 2
2 Abolrous, Hazen 14 8 3 2
2 Abolrous, Hazen 14 10 4 2
2 Abolrous, Hazen 14 12 4 2
2 Abolrous, Hazen 14 14 5 2
3 Hance, Jim 12 2 1 3
3 Hance, Jim 12 7 2 3
3 Hance, Jim 12 13 5 3
3 Hance, Jim 12 15 7 3
4 Adams, Terry 12 NULL NULL NULL
5 Hansen, Claus 11 3 1 5
5 Hansen, Claus 11 5 2 5
5 Hansen, Claus 11 11 4 5
6 Penor, Lori 13 6 2 6
7 Perham, Tom 12 NULL NULL NULL
8 Peng, Yun-Feng 0 NULL NULL NULL

Join with count
15

Join with distinct.
Abercrombie, Kim 2
Abercrombie, Kim 3
Abercrombie, Kim 5
Abolrous, Hazen 2
Abolrous, Hazen 5
Abolrous, Hazen 6
Abolrous, Hazen 3
Hance, Jim 2
```

```
Hance, Jim 2
Hance, Jim 1
Adams, Terry 2
Adams, Terry 5
Adams, Terry 2
Hansen, Claus 3
Hansen, Claus 2
Perham, Tom 3

Join with distinct and count.
15

Selecting students with age between 10 and 15.
Abercrombie, Kim
Abolrous, Hazen
Hance, Jim
Adams, Terry
Hansen, Claus
Penor, Lori
Perham, Tom

Selecting students with age either 11 or 12.
Hance, Jim
Adams, Terry
Hansen, Claus
Perham, Tom

Selecting students in a certain age range and sorting.
Penor, Lori 13
Perham, Tom 12
Hance, Jim 12
Adams, Terry 12

Selecting students with certain ages, taking account of possibility of nulls.
Hance, Jim
Adams, Terry

Union of two queries.
Abercrombie, Kim 10
Abolrous, Hazen 14
Hance, Jim 12
Adams, Terry 12
Hansen, Claus 11
Penor, Lori 13
Perham, Tom 12
Peng, Yun-Feng NULL

Intersect of two queries.

Using if statement to alter results for special value.
1 10 10
2 14 14
3 12 12
4 12 12
5 11 11
6 13 13
7 12 12
8 NULL NULL

Using if statement to alter results special values.
1 10 10
2 14 14
3 12 12
4 12 12
5 11 11
6 13 13
7 12 12
8 NULL NULL

Multiple table select
```

```
Multiple table select:
StudentID Name Age CourseID CourseName
1 Abercrombie, Kim 10 1 Algebra I
2 Abolrous, Hazen 14 1 Algebra I
3 Hance, Jim 12 1 Algebra I
4 Adams, Terry 12 1 Algebra I
5 Hansen, Claus 11 1 Algebra I
6 Penor, Lori 13 1 Algebra I
7 Perham, Tom 12 1 Algebra I
8 Peng, Yun-Feng NULL 1 Algebra I
1 Abercrombie, Kim 10 2 Trigonometry
2 Abolrous, Hazen 14 2 Trigonometry
3 Hance, Jim 12 2 Trigonometry
4 Adams, Terry 12 2 Trigonometry
5 Hansen, Claus 11 2 Trigonometry
6 Penor, Lori 13 2 Trigonometry
7 Perham, Tom 12 2 Trigonometry
8 Peng, Yun-Feng NULL 2 Trigonometry
1 Abercrombie, Kim 10 3 Algebra II
2 Abolrous, Hazen 14 3 Algebra II
3 Hance, Jim 12 3 Algebra II
4 Adams, Terry 12 3 Algebra II
5 Hansen, Claus 11 3 Algebra II
6 Penor, Lori 13 3 Algebra II
7 Perham, Tom 12 3 Algebra II
8 Peng, Yun-Feng NULL 3 Algebra II
1 Abercrombie, Kim 10 4 History
2 Abolrous, Hazen 14 4 History
3 Hance, Jim 12 4 History
4 Adams, Terry 12 4 History
5 Hansen, Claus 11 4 History
6 Penor, Lori 13 4 History
7 Perham, Tom 12 4 History
8 Peng, Yun-Feng NULL 4 History
1 Abercrombie, Kim 10 5 English
2 Abolrous, Hazen 14 5 English
3 Hance, Jim 12 5 English
4 Adams, Terry 12 5 English
5 Hansen, Claus 11 5 English
6 Penor, Lori 13 5 English
7 Perham, Tom 12 5 English
8 Peng, Yun-Feng NULL 5 English
1 Abercrombie, Kim 10 6 French
2 Abolrous, Hazen 14 6 French
3 Hance, Jim 12 6 French
4 Adams, Terry 12 6 French
5 Hansen, Claus 11 6 French
6 Penor, Lori 13 6 French
7 Perham, Tom 12 6 French
8 Peng, Yun-Feng NULL 6 French
1 Abercrombie, Kim 10 7 Chinese
2 Abolrous, Hazen 14 7 Chinese
3 Hance, Jim 12 7 Chinese
4 Adams, Terry 12 7 Chinese
5 Hansen, Claus 11 7 Chinese
6 Penor, Lori 13 7 Chinese
7 Perham, Tom 12 7 Chinese
8 Peng, Yun-Feng NULL 7 Chinese

Multiple Joins
Abercrombie, Kim Trigonometry
Abercrombie, Kim Algebra II
Abercrombie, Kim English
Abolrous, Hazen Trigonometry
Abolrous, Hazen English
Abolrous, Hazen French
Abolrous, Hazen Algebra II
Hance, Jim Trigonometry
Hance, Jim Algebra I
```

```
Adams, Terry Trigonometry
Adams, Terry English
Adams, Terry Trigonometry
Hansen, Claus Algebra II
Hansen, Claus Trigonometry
Perham, Tom Algebra II

Multiple Left Outer Joins
Abercrombie, Kim Trigonometry
Abercrombie, Kim Algebra II
Abercrombie, Kim English
Abolrous, Hazen Trigonometry
Abolrous, Hazen English
Abolrous, Hazen French
Abolrous, Hazen Algebra II
Hance, Jim Trigonometry
Hance, Jim Algebra I
Adams, Terry Trigonometry
Adams, Terry English
Adams, Terry Trigonometry
Hansen, Claus Algebra II
Hansen, Claus Trigonometry
Penor, Lori
Perham, Tom Algebra II
Peng, Yun-Feng

type schema
val db : schema.ServiceTypes.SimpleDataContextTypes.MyDatabase1
val student : System.Data.Linq.Table<schema.ServiceTypes.Student>
val data : int list = [1; 5; 7; 11; 18; 21]
type Nullable<'T
                when 'T : (new : unit ->  'T) and 'T : struct and
                     'T :> System.ValueType> with
  member Print : unit -> string
val num : int = 21
val student2 : schema.ServiceTypes.Student
val student3 : schema.ServiceTypes.Student
val student4 : schema.ServiceTypes.Student
val student5 : int = 1
val student6 : int = 8
val idList : int list = [1; 2; 5; 10]
val idQuery : seq<int>
val names : string [] = [|"a"; "b"; "c"|]
module Queries = begin
  val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
  val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end
module Queries2 = begin
  val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
  val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end
```

См. также

- Справочник по языку F#
- Класс LINQ. QueryBuilder
- Выражения вычисления

Цитирование кода

23.10.2019 • 10 minutes to read • Edit Online

> **NOTE**
> Ссылка на справочник по API ведет на сайт MSDN. Работа над справочником по API docs.microsoft.com не завершена.

В этом разделе описываются *цитаты кода*, функции языка, позволяющие программно создавать выражения кода F# и работать с ними. Эта функция позволяет создать абстрактное дерево синтаксиса, представляющее F# код. Затем дерево абстрактного синтаксиса можно просмотреть и обработать в соответствии с потребностями приложения. Например, дерево можно использовать для создания F# кода или создания кода на другом языке.

Выражения в кавычках

Выражение в кавычках — это F# выражение в коде, которое отделяется таким образом, что оно не компилируется как часть программы, а компилируется в объект, представляющий F# выражение. Можно пометить выражение в кавычках одним из двух способов: с информацией о типе или без сведений о типе. Если вы хотите включить сведения о типах, используйте символы `<@` и `@>` для разделения выражения в кавычках. Если сведения о типах не требуются, используются символы `<@@` и. `@@>` В следующем коде показаны типизированные и нетипизированные предложения.

```
open Microsoft.FSharp.Quotations
// A typed code quotation.
let expr : Expr<int> = <@ 1 + 1 @>
// An untyped code quotation.
let expr2 : Expr = <@@ 1 + 1 @@>
```

Обход большого дерева выражения выполняется быстрее, если не включать сведения о типе. Результирующий тип выражения, заключенного в кавычки с типизированными `Expr<'T>` символами, — это, где параметр типа имеет тип выражения, определяемый F# алгоритмом определения типа компилятора. При использовании цитат кода без сведений о типе тип выражения, заключенного в кавычки, является выражением типа, не являющимся универсальным типом expr. Для получения нетипизированного `Expr` объекта можно вызвать свойство `Expr` RAW для типизированного класса.

Существуют разнообразные статические методы, позволяющие программно создавать F# объекты выражений в `Expr` классе без использования заключенных в кавычки выражений.

Обратите внимание, что цитата кода должна включать выражение Complete. Например, для привязки требуется как определение связанного имени, так и дополнительное выражение, использующее привязку. `let` В подробном синтаксисе это выражение, следующее за `in` ключевым словом. На верхнем уровне в модуле это просто следующее выражение в модуле, но в кавычках оно требуется явным образом.

Поэтому следующее выражение является недопустимым.

```
// Not valid:
// <@ let f x = x + 1 @>
```

Однако следующие выражения являются допустимыми.

```
// Valid:
<@ let f x = x + 10 in f 20 @>
// Valid:
<@
    let f x = x + 10
    f 20
@>
```

Для вычисления F# предложений необходимо использовать F# средство оценки предложений. Он обеспечивает поддержку оценки и исполнения F# объектов выражений.

Тип выражения

Экземпляр `Expr` типа представляет F# выражение. Универсальные и неуниверсальные `Expr` типы описаны в документации по F# библиотеке. Дополнительные сведения см. в статье о пространстве имен Microsoft. FSharp. цитирований и предложениях. expr.

Операторы объединения

Объединение позволяет комбинировать цитаты с литеральным кодом с выражениями, созданными программно или из другой цитаты в виде кода. Операторы `%` и `%%` позволяют добавлять объект F# выражения в цитату кода. `%` Оператор используется для вставки объекта типизированного выражения в типизированную кавычку; `%%` оператор используется для вставки нетипизированного объекта выражения в нетипизированное предложение. Оба оператора являются унарными префиксными операторами. Таким образом `expr` , если является нетипизированным выражением `Expr` типа, следующий код является допустимым.

```
<@@ 1 + %%expr @@>
```

И если `expr` является типизированной кавычкой типа `Expr<int>` , следующий код является допустимым.

```
<@ 1 + %expr @>
```

Пример

Описание

В следующем примере показано использование цитат кода для помещения F# кода в объект выражения, а затем печать F# кода, представляющего выражение. Определена `println` функция, которая содержит рекурсивную `print` функцию, которая отображает объект F# выражения (типа `Expr`) в удобном формате. Существует несколько активных шаблонов в модулях Microsoft. FSharp. цитирований. Patterns и Microsoft. FSharp. цитирований. активный шаблон derivedpatterns , которые можно использовать для анализа объектов выражений. Этот пример не включает в себя все возможные закономерности, которые могут отображаться F# в выражении. Любой нераспознанный шаблон активирует совпадение с шаблоном шаблона (`_`) и подготавливается к просмотру `ToString` с помощью метода `Expr` , который в типе позволяет знать активный шаблон для добавления в выражение соответствия.

Код

```
module Print
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns
```

```
let println expr =
    let rec print expr =
        match expr with
        | Application(expr1, expr2) ->
            // Function application.
            print expr1
            printf " "
            print expr2
        | SpecificCall <@@ (+) @@> (_, _, exprList) ->
            // Matches a call to (+). Must appear before Call pattern.
            print exprList.Head
            printf " + "
            print exprList.Tail.Head
        | Call(exprOpt, methodInfo, exprList) ->
            // Method or module function call.
            match exprOpt with
            | Some expr -> print expr
            | None -> printf "%s" methodInfo.DeclaringType.Name
            printf ".%s(" methodInfo.Name
            if (exprList.IsEmpty) then printf ")" else
            print exprList.Head
            for expr in exprList.Tail do
                printf ","
                print expr
            printf ")"
        | Int32(n) ->
            printf "%d" n
        | Lambda(param, body) ->
            // Lambda expression.
            printf "fun (%s:%s) -> " param.Name (param.Type.ToString())
            print body
        | Let(var, expr1, expr2) ->
            // Let binding.
            if (var.IsMutable) then
                printf "let mutable %s = " var.Name
            else
                printf "let %s = " var.Name
            print expr1
            printf " in "
            print expr2
        | PropertyGet(_, propOrValInfo, _) ->
            printf "%s" propOrValInfo.Name
        | String(str) ->
            printf "%s" str
        | Value(value, typ) ->
            printf "%s" (value.ToString())
        | Var(var) ->
            printf "%s" var.Name
        | _ -> printf "%s" (expr.ToString())
    print expr
    printfn ""


let a = 2

// exprLambda has type "(int -> int)".
let exprLambda = <@ fun x -> x + 1 @>
// exprCall has type unit.
let exprCall = <@ a + 1 @>

println exprLambda
println exprCall
println <@@ let f x = x + 10 in f 10 @@>
```

Вывод

```
fun (x:System.Int32) -> x + 1
a + 1
let f = fun (x:System.Int32) -> x + 10 in f 10
```

Пример

Описание

Вы также можете использовать три активных шаблона в модуле експршапе для прохода по деревьям выражений с меньшим количеством активных шаблонов. Эти активные шаблоны могут оказаться полезными, если требуется пройти по дереву, но не требуется вся информация в большинстве узлов. При использовании F# этих шаблонов любое выражение соответствует одному из следующих трех шаблонов: `ShapeVar` если выражение является переменной, `ShapeLambda` если выражение является лямбда-выражением, или `ShapeCombination` если выражение является любым другим. Если вы просматриваете дерево выражения с помощью активных шаблонов, как в предыдущем примере кода, необходимо использовать гораздо больше шаблонов для управления всеми возможными F# типами выражений, и код будет более сложен. Дополнительные сведения см. в разделе експршапе.|Шапевар|шапеламбда шапекомбинатион Active Pattern.

Следующий пример кода можно использовать в качестве основания для более сложных обходов. В этом коде дерево выражения создается для выражения, включающего вызов функции, `add`. Активный шаблон спеЦификкалл используется для обнаружения любого вызова `add` в дереве выражения. Этот активный шаблон назначает аргументы вызова `exprList` значения. В этом случае существует только два, поэтому они извлекаются, и функция вызывается рекурсивно для аргументов. Результаты вставляются в цитату кода, которая представляет вызов `mul` с помощью оператора splice (`%%`). `println` Функция из предыдущего примера используется для вывода результатов.

Код в других активных ветвях шаблонов просто повторно создает одно и то же дерево выражения, поэтому единственное изменение в результирующем выражении — это изменение `add` с на. `mul`

Код

```
module Module1
open Print
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.DerivedPatterns
open Microsoft.FSharp.Quotations.ExprShape

let add x y = x + y
let mul x y = x * y

let rec substituteExpr expression =
    match expression with
    | SpecificCall <@@ add @@> (_, _, exprList) ->
        let lhs = substituteExpr exprList.Head
        let rhs = substituteExpr exprList.Tail.Head
        <@@ mul %%lhs %%rhs @@>
    | ShapeVar var -> Expr.Var var
    | ShapeLambda (var, expr) -> Expr.Lambda (var, substituteExpr expr)
    | ShapeCombination(shapeComboObject, exprList) ->
        RebuildShapeCombination(shapeComboObject, List.map substituteExpr exprList)

let expr1 = <@@ 1 + (add 2 (add 3 4)) @@>
println expr1
let expr2 = substituteExpr expr1
println expr2
```

Вывод

```
1 + Module1.add(2,Module1.add(3,4))
1 + Module1.mul(2,Module1.mul(3,4))
```

См. также

- Справочник по языку F#

Ключевое слово fixed

23.10.2019 • 2 minutes to read • Edit Online

F#представляет 4.1 `fixed` ключевое слово, которое позволяет «закрепить» локальный в стек для предотвращения ее собранные или перемещен во время сбора мусора. Он используется для низкоуровневых сценариев программирования.

Синтаксис

```
use ptr = fixed expression
```

Примечания

При этом расширяется синтаксис выражений для извлечения указатель и связывание его с именем которой предотвращается собранные или перемещается во время сбора мусора.

Указатель на основе выражения является фиксированным через `fixed` ключевое слово привязывается к идентификатору с помощью `use` ключевое слово. Это семантика аналогичную управление ресурсами с помощью `use` ключевое слово. Указатель является фиксированным, пока он находится в области, и когда она выходит из области действия, он больше не является фиксированным. `fixed` нельзя использовать вне контекста `use` привязки. Необходимо привязать указатель к имени с `use`.

Использование `fixed` должно находиться в пределах выражения в функции или метода. Он не может использоваться в области сценария уровня или уровня модуля.

Как и весь код указатель это является небезопасный функцией и выдает предупреждение при использовании.

Пример

```
open Microsoft.FSharp.NativeInterop

type Point = { mutable X: int; mutable Y: int}

let squareWithPointer (p: nativeptr<int>) =
    // Dereference the pointer at the 0th address.
    let mutable value = NativePtr.get p 0

    // Perform some work
    value <- value * value

    // Set the value in the pointer at the 0th address.
    NativePtr.set p 0 value

let pnt = { X = 1; Y = 2 }
printfn "pnt before - X: %d Y: %d" pnt.X pnt.Y // prints 1 and 2

// Note that the use of 'fixed' is inside a function.
// You cannot fix a pointer at a script-level or module-level scope.
let doPointerWork() =
    use ptr = fixed &pnt.Y

    // Square the Y value
    squareWithPointer ptr
    printfn "pnt after - X: %d Y: %d" pnt.X pnt.Y // prints 1 and 4

doPointerWork()
```

См. также

- Nativeptr-модуль

Byref

11.01.2020 • 9 minutes to read • Edit Online

F#имеет две основные функциональные области, которые имеют место в пространстве низкоуровневого программирования:

- `byref` / `inref` /типы `outref` , которые являются управляемыми указателями. Они имеют ограничения на использование, поэтому нельзя компилировать программу, недопустимую во время выполнения.
- Структура, подобная `byref` , которая представляет собой структуру , которая имеет подобную семантику и те же ограничения времени компиляции, что и `byref<'T>` . Один из примеров — Span<T>.

Синтаксис

```
// Byref types as parameters
let f (x: byref<'T>) = ()
let g (x: inref<'T>) = ()
let h (x: outref<'T>) = ()

// Calling a function with a byref parameter
let mutable x = 3
f &x

// Declaring a byref-like struct
open System.Runtime.CompilerServices

[<Struct; IsByRefLike>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

ByRef, инреф и аутреф

Существует три формы `byref` :

- `inref<'T>` — управляемый указатель для чтения базового значения.
- `outref<'T>` — управляемый указатель для записи в базовое значение.
- `byref<'T>` — управляемый указатель для чтения и записи базового значения.

Можно передать `byref<'T>` , где ожидается `inref<'T>` . Аналогичным образом можно передать `byref<'T>` , где ожидается `outref<'T>` .

Использование ByRef

Чтобы использовать `inref<'T>` , необходимо получить значение указателя с `&` :

```
open System

let f (dt: inref<DateTime>) =
    printfn "Now: %s" (dt.ToString())

let usage =
    let dt = DateTime.Now
    f &dt // Pass a pointer to 'dt'
```

Для записи в указатель с помощью `outref<'T>` или `byref<'T>` необходимо также сделать значение, которое захватит указатель на `mutable`.

```
open System

let f (dt: byref<DateTime>) =
    printfn "Now: %s" (dt.ToString())
    dt <- DateTime.Now

// Make 'dt' mutable
let mutable dt = DateTime.Now

// Now you can pass the pointer to 'dt'
f &dt
```

Если вы пишете только указатель, а не читаете его, рассмотрите возможность использования `outref<'T>` вместо `byref<'T>`.

Семантика инреф

Рассмотрим следующий код.

```
let f (x: inref<SomeStruct>) = x.SomeField
```

В семантическом виде это означает следующее:

- Владелец указателя `x` может использовать его только для чтения значения.
- Любой указатель, полученный к `struct` полям, вложенным в `SomeStruct`, получает `inref<_>` типа.

Также верно следующее:

- Нет никаких последствие того, что другие потоки или псевдонимы не имеют доступа на запись к `x`.
- Нечего никакое `SomeStruct` неизменяемо, поскольку `x` является `inref`.

Однако для F# типов значений, которые **являются** неизменяемыми, `this` указатель выводится как `inref`.

Все эти правила вместе означают, что владелец указателя `inref` не может изменить немедленное содержимое памяти, на которое указывает.

Семантика аутреф

Цель `outref<'T>` — указать, что указатель должен быть записан только в. Неожиданно `outref<'T>` позволяет считывать базовое значение, несмотря на его имя. Это происходит в целях совместимости. Семантически `outref<'T>` не отличается от `byref<'T>`.

Взаимодействие с#ом С

C#поддерживает ключевые слова `in ref` и `out ref`, а также возвращает `ref`. В следующей таблице показано, F# как интерпретирует C# выдачи:

C#СОЗДАНИЯ	F#ВЫВОДИТ
`ref` возвращаемое значение	`outref<'T>`
`ref readonly` возвращаемое значение	`inref<'T>`
Параметр `in ref`	`inref<'T>`
Параметр `out ref`	`outref<'T>`

В следующей таблице показано, F# какие выдачи:

F#СОЗДАНИЯ	ВЫПУЩЕННАЯ КОНСТРУКЦИЯ
Аргумент `inref<'T>`	атрибут `[In]` в аргументе
`inref<'T>` возврат	атрибут `modreq` в значении
`inref<'T>` в абстрактном слоте или реализации	`modreq` аргумента или Return
Аргумент `outref<'T>`	атрибут `[Out]` в аргументе

Определение типа и перегрузка правил

Тип `inref<'T>` выводится F# компилятором в следующих случаях:

1. Параметр или возвращаемый тип .NET, имеющий атрибут `IsReadOnly`.
2. `this` указатель на тип структуры, не имеющий изменяемых полей.
3. Адрес области памяти, полученной из другого указателя `inref<_>`.

При использовании неявного адреса `inref` перегрузку с аргументом типа `SomeType` предпочтительнее для перегрузки с аргументом типа `inref<SomeType>`. Например:

```
type C() =
    static member M(x: System.DateTime) = x.AddDays(1.0)
    static member M(x: inref<System.DateTime>) = x.AddDays(2.0)
    static member M2(x: System.DateTime, y: int) = x.AddDays(1.0)
    static member M2(x: inref<System.DateTime>, y: int) = x.AddDays(2.0)

let res = System.DateTime.Now
let v =  C.M(res)
let v2 =  C.M2(res, 4)
```

В обоих случаях перегрузки, принимающие `System.DateTime`, разрешаются вместо перегрузок, принимающих `inref<System.DateTime>`.

Структуры, аналогичные ByRef

Помимо `byref` / `inref` / `outref` три, можно определить собственные структуры, которые могут соответствовать семантике `byref`. Это делается с помощью атрибута IsByRefLikeAttribute:

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` не подразумевает `Struct`. Оба должны присутствовать в типе.

"`byref`-похожая" структура в F# — это тип значения, привязанный к стеку. Он никогда не выделяется в управляемой куче. Структура, подобная `byref`, полезна для высокопроизводительного программирования, так как она применяется к набору строгих проверок времени существования и без записи. Правила таковы.

- Их можно использовать в качестве параметров функций, параметров методов, локальных переменных, возвращаемых методом.
- Они не могут быть статическими или членами экземпляров класса или обычной структуры.
- Они не могут быть захвачены какой-либо конструкцией замыкания (`async` методами или лямбда-выражениями).
- Их нельзя использовать в качестве универсального параметра.

Последний момент является ключевым для F# программирования в стиле конвейера, так как `|>` является универсальной функцией, которая выполняет параметризацию входных типов. Это ограничение может быть ослаблено для `|>` в будущем, так как оно встроено и не вызывает невстроенные универсальные функции в своем тексте.

Хотя эти правила строго ограничивают использование, они делают это для обеспечения безопасности высокопроизводительных вычислительных систем.

Возвраты по ссылке

Функция ByRef возвращает F# из функций или членов, которые могут быть созданы и использованы. При использовании метода, возвращающего `byref`, значение неявно разыменовано. Например:

```
let safeSum(bytes: Span<byte>) =
    let mutable sum = 0
    for i in 0 .. bytes.Length - 1 do
        sum <- sum + int bytes.[i]
    sum

let sum = safeSum(mySpanOfBytes)
printfn "%d" sum // 'sum' is of type 'int'
```

Чтобы избежать неявного разыменования, например передачи ссылки через несколько цепочек вызовов, используйте `&x` (где `x` является значением).

Можно также напрямую назначить `byref` возврата. Рассмотрим следующую (очень императивную) программу:

```
type C() =
    let mutable nums = [| 1; 3; 7; 15; 31; 63; 127; 255; 511; 1023 |]

    override _.ToString() = String.Join(' ', nums)

    member _.FindLargestSmallerThan(target: int) =
        let mutable ctr = nums.Length - 1

        while ctr > 0 && nums.[ctr] >= target do ctr <- ctr - 1

        if ctr > 0 then &nums.[ctr] else &nums.[0]

[<EntryPoint>]
let main argv =
    let c = C()
    printfn "Original sequence: %s" (c.ToString())

    let v = &c.FindLargestSmallerThan 16

    v <- v*2 // Directly assign to the byref return

    printfn "New sequence:      %s" (c.ToString())

    0 // return an integer exit code
```

Результат.

```
Original sequence: 1 3 7 15 31 63 127 255 511 1023
New sequence:      1 3 7 30 31 63 127 255 511 1023
```

Определение области для ByRef

Значение с привязкой к `let` не может превышать область, в которой он был определен. Например, запрещены следующие возможности:

```
let test2 () =
    let x = 12
    &x // Error: 'x' exceeds its defined scope!

let test () =
    let x =
        let y = 1
        &y // Error: `y` exceeds its defined scope!
    ()
```

Это предотвращает получение различных результатов в зависимости от того, выполняется ли компиляция с оптимизацией.

Ссылочные ячейки

23.10.2019 • 4 minutes to read • Edit Online

Ссылочные ячейки — это места хранения, которые позволяют создавать изменяемые значения с семантикой ссылок.

Синтаксис

```
ref expression
```

Примечания

Для создания новой ссылочной ячейки, инкапсулирующей значение, перед значением ставится оператор `ref`. Базовое значение затем можно изменить, так как оно является изменяемым.

Ссылочная ячейка содержит фактическое значение; это не просто адрес. При создании ссылочной ячейки с помощью оператора `ref` создается копия базового значения в качестве инкапсулированного изменяемого значения.

Разыменовать ссылочную ячейку можно с помощью оператора `!` (восклицательного знака).

Следующий пример кода иллюстрирует объявление и использование ссылочных ячеек.

```
// Declare a reference.
let refVar = ref 6

// Change the value referred to by the reference.
refVar := 50

// Dereference by using the ! operator.
printfn "%d" !refVar
```

В результате получается `50`.

Ссылочные ячейки являются экземплярами универсального типа записей `Ref`, который объявляется следующим образом.

```
type Ref<'a> =
{ mutable contents: 'a }
```

Тип `'a ref` является синонимом `Ref<'a>`. В интегрированной среде разработки в компиляторе и в IntelliSense отображается первое обозначение данного типа, однако базовым определением является второе.

Оператор `ref` создает новую ссылочную ячейку. Следующий код представляет собой объявление оператора `ref`.

```
let ref x = { contents = x }
```

В следующей таблице перечислены возможности, доступные для ссылочной ячейки.

ОПЕРАТОР, ЧЛЕН ИЛИ ПОЛЕ	ОПИСАНИЕ	ТИП	ОПРЕДЕЛЕНИЕ
`!` (оператор разыменования)	Возвращает базовое значение.	`'a ref -> 'a`	`let (!) r = r.contents`
`:=` (оператор присваивания)	Изменяет базовое значение.	`'a ref -> 'a -> unit`	`let (:=) r x = r.contents <- x`
`ref` (оператор)	Инкапсулирует значение в новую ссылочную ячейку.	`'a -> 'a ref`	`let ref x = { contents = x }`
`Value` (свойство)	Получает или задает базовое значение.	`unit -> 'a`	`member x.Value = x.contents`
`contents` (поле записи)	Получает или задает базовое значение.	`'a`	`let ref x = { contents = x }`

Существует несколько способов доступа к базовому значению. Значение, возвращаемое оператором разыменования (`!`), не является присваиваемым значением. Следовательно, при изменении базового значения нужно вместо этого оператора использовать оператор присваивания (`:=`).

И свойство `Value` , и поле `contents` являются присваиваемыми значениями. Следовательно, их можно использовать для доступа к базовому значению или его изменения, как показано в следующем коде.

```
let xRef : int ref = ref 10

printfn "%d" (xRef.Value)
printfn "%d" (xRef.contents)

xRef.Value <- 11
printfn "%d" (xRef.Value)
xRef.contents <- 12
printfn "%d" (xRef.contents)
```

Выходные данные выглядят следующим образом.

```
10
10
11
12
```

Поле `contents` предусмотрено для совместимости с другими версиями языка ML, и его наличие приводит к выводу предупреждения в процессе компиляции. Для отключения этого предупреждения используется параметр компилятора `--mlcompatibility` . Дополнительные сведения см. в разделе Параметры компилятора.

C#программистам следует помнить `ref` , C# что в не `ref` то же самое, F#что и в. Эквивалентными конструкциями в F# являются ByRef, которые являются разными концепциями из ссылочных ячеек.

Значения, помеченные как, `mutable` могут быть автоматически повышены до `'a ref` , если захватывается замыканием; см. значения.

См. также

- Справочник по языку F#
- Параметры и аргументы
- Справочник символов и операторов

- Значения

Директивы компилятора

23.10.2019 • 7 minutes to read • Edit Online

В этом разделе описываются директивы процессора и директивы компилятора.

Директивы препроцессора

Директива препроцессора дополняется префиксом с символом «#» и отображается в строке сама по себе. Она интерпретируется препроцессором, который запускается перед самим компилятором.

В следующей таблице перечислены директивы препроцессора, имеющиеся в F#.

ДИРЕКТИВА	ОПИСАНИЕ
`#if` *символ*	Поддерживает условную компиляцию. Код в разделе после `#if` включается, если *символ* определен. Символ также может быть инвертирован с помощью `!`.
`#else`	Поддерживает условную компиляцию. Помечает раздел кода, который следует включить, если символ, используемый с предыдущей директивой `#if`, не определен.
`#endif`	Поддерживает условную компиляцию. Помечает конец условного раздела кода.
`#` штрих *int*, `#` штрих *mun int строка*, `#` штрих *mun int буквальная строка*	Указывает исходную строку кода и имя файла для отладки. Эта возможность предназначена для средств, создающих исходный код F#.
`#nowarn` *варнингкоде*	Отключает предупреждение или предупреждения компилятора. Чтобы отключить предупреждение, найдите его номер в выходных данных компилятора и заключите его в кавычки. Не указывайте префикс «FS». Чтобы отключить несколько номеров предупреждений в одной строке, заключите каждый номер в кавычки и отделяйте каждую строку пробелом. Например:

```
#nowarn "9" "40"
```

Результат отключения предупреждения применяется ко всему файлу, включая фрагменты файла, предшествующие директиве. |

Директивы условной компиляции

Код, Деактивируемый одной из этих директив, недоступен в редакторе Visual Studio Code.

> **NOTE**
>
> Поведение директив условной компиляции отличается от их поведения в других языках. Например, нельзя использовать логические выражения с символами, а `true` и `false` не имеют особого значения. Символы, используемые в директиве `if`, должны задаваться с помощью командной строки или в параметрах проекта; в этом языке отсутствует директива препроцессора `define`.

В следующем коде демонстрируется применение директив `#if`, `#else` и `#endif`. В данном примере код содержит две версии определения `function1`. Если `VERSION1` определяется с помощью параметра компилятора-define, активируется код между `#if` директивой и `#else` директивой. В противном случае активируется код между директивами `#else` и `#endif`.

```
#if VERSION1
let function1 x y =
   printfn "x: %d y: %d" x y
   x + 2 * y
#else
let function1 x y =
   printfn "x: %d y: %d" x y
   x - 2*y
#endif

let result = function1 10 20
```

В языке F# отсутствует директива препроцессора `#define`. Вы должны использовать параметр компилятора или параметры проекта для определения символов, используемых директивой `#if`.

Директивы условной компиляции не могут быть вложенными. В директивах препроцессора отступ не важен.

Можно также инвертировать символ с помощью `!`. В этом примере значением строки является нечто, только если *не* выполняется отладка:

```
#if !DEBUG
let str = "Not debugging!"
#else
let str = "Debugging!"
#endif
```

Директивы строк

При сборке компилятор сообщает об ошибках в коде F #, ссылаясь на номера строк, в которых возникли ошибки. Номера строк начинаются с 1 для первой строки в файле. Тем не менее при создании исходного кода F# из другого средства номера строк в сформированном коде обычно не представляют интереса, поскольку ошибки в сформированном коде F#, скорее всего, проистекают из другого источника. Директива `#line` позволяет разработчикам средств, формирующих исходный код F#, передавать сведения о номерах исходных строк и исходных файлах в сформированный код F#.

При использовании директивы `#line` необходимо заключать имена файлов в кавычки. Если в начале строки не указывается токен verbatim (`@`), то чтобы использовать в пути символы обратной косой черты, необходимо их экранировать, указывая две обратные косые черты вместо одной. Далее приводятся допустимые токены строк. В этих примерах предполагается, что исходный файл `Script1` при запуске в соответствующем средстве приводит к автоматическому созданию файла кода F# и что код в месте расположения этих директив формируется из некоторых токенов в строке 25 файла `Script1`.

```
# 25
#line 25
#line 25 "C:\\Projects\\MyProject\\MyProject\\Script1"
#line 25 @"C:\Projects\MyProject\MyProject\Script1"
# 25 @"C:\Projects\MyProject\MyProject\Script1"
```

Эти токены указывают, что код F#, сформированный в этом месте, является производным от некоторых конструкций в строке `25` или рядом с ней в файле `Script1`.

Директивы компилятора

Директивы компилятора сходны с директивами препроцессора, поскольку они имеют префикс в виде знака #, но они не интерпретируются препроцессором; компилировать и действовать в соответствии с этими директивами должен компилятор.

В следующей таблице перечислены директивы компилятора, доступные в языке F#.

ДИРЕКТИВА	ОПИСАНИЕ
`#light` ["on"|"OFF"]	Включает или отключает упрощенный синтаксис для совместимости с другими версиями ML. Упрощенный синтаксис включен по умолчанию. Подробный синтаксис всегда включен. Таким образом, вы можете использовать как упрощенный, так и подробный синтаксис. Директива `#light` сама по себе эквивалентна `#light "on"`. Если указана директива `#light "off"`, то для всех языковых конструкций необходимо использовать подробный синтаксис. Синтаксис в документации по F# представлен исходя из предположения, что используется упрощенный синтаксис. Дополнительные сведения см. в разделе подробный синтаксис.

Директивы интерпретатора (FSI. exe) см. в разделе интерактивное F#программирование с помощью .

См. также

- Справочник по языку F#
- Параметры компилятора

Параметры компилятора

04.11.2019 • 15 minutes to read • Edit Online

В F# этом разделе описываются параметры командной строки компилятора для компилятора FSC. exe. Среду компиляции также можно контролировать, задав свойства проекта.

Параметры компилятора в алфавитном порядке

В следующей таблице перечислены параметры компилятора в алфавитном порядке. Некоторые параметры F# компилятора похожи на параметры C# компилятора. В этом случае предоставляется ссылка на раздел параметров C# компилятора.

ПАРАМЕТР КОМПИЛЯТОРА	ОПИСАНИЕ
`-a filename.fs`	Создает библиотеку из указанного файла. Этот параметр является краткой формой `--target:library filename.fs`.
`--baseaddress:address`	Указывает предпочтительный базовый адрес для загрузки DLL. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора в BaseAddress.
`--codepage:id`	Указывает, какую кодовую страницу следует использовать во время компиляции, если требуемая страница не является текущей кодовой страницей по умолчанию для системы. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в разделе /кодовые# страницы) (C параметры компилятора.
`--consolecolors`	Указывает, что ошибки и предупреждения используют цветовую маркировку текста в консоли.
`--crossoptimize[+|-]`	Включает или отключает оптимизацию между модулями.
`--delaysign[+|-]`	Отложенная подпись сборки с использованием только открытой части ключа строгого имени. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора delaysign C.

ПАРАМЕТР КОМПИЛЯТОРА	ОПИСАНИЕ
`--checked[+|-]`	Включает или отключает создание проверок переполнения. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (проверенные# параметры)компилятора C.
`--debug[+|-]` `-g[+|-]` `--debug:[full|pdbonly]` `-g: [full|pdbonly]`	Включает или отключает создание отладочной информации или определяет тип создаваемой отладочной информации. Значение по умолчанию — Full, что позволяет присоединиться к выполняющейся программе. Выберите **pdbonly** , чтобы получить ограниченную отладочную информацию, хранящуюся в файле PDB (база данных программы). Эквивалентно параметру C# компилятора с тем же именем. Дополнительные сведения см. в разделе Отладка ()параметров компилятора C# /
`--define:symbol` `-d:symbol`	Определяет символ для использования в условной компиляции.
`--deterministic[+|-]`	Создает детерминированную сборку (включая GUID версии модуля и метку времени). Этот параметр нельзя использовать с номерами версий с подстановочными знаками и поддерживает только внедренные и переносимые типы отладки
`--doc:xmldoc-filename`	Указывает компилятору создавать комментарии XML-документации к указанному файлу. Дополнительные сведения см. в разделе XML Documentation. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора doc C.
`--fullpaths`	Указывает компилятору создавать полные пути. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора fullpaths C.
`--help` `-?`	Отображает сведения об использовании, включая краткое описание всех параметров компилятора.

ПАРАМЕТР КОМПИЛЯТОРА	ОПИСАНИЕ
`--highentropyva[+|-]`	Включение или отключение функции случайного использования макета адресного пространства с высокой энтропией (ASLR), улучшенной безопасности. Операционная система случайным образом накладывает место в памяти, где загружена инфраструктура для приложений (например, стек и куча). Если этот параметр включен, операционные системы могут использовать этот случай, чтобы использовать полный 64-битный адрес пространства на 64-разрядном компьютере.
`--keycontainer:key-container-name`	Задает контейнер ключа для строгого имени.
`--keyfile:filename`	Указывает имя файла открытого ключа для подписи создаваемой сборки.
`--lib:folder-name` `-I:folder-name`	Указывает каталог для поиска сборок, на которые имеются ссылки. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора в lib.
`--linkresource:resource-info`	Связывает указанный ресурс с сборкой. Формат сведений о ресурсе — `filename[name[public|private]]` Связывание одного ресурса с этим параметром является альтернативой внедрению всего файла ресурсов с параметром `--resource`. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора linkresource C.
`--mlcompatibility`	Игнорирует предупреждения, которые появляются при использовании функций, предназначенных для совместимости с другими версиями ML.
`--noframework`	Отключает ссылку по умолчанию на сборку .NET Framework.
`--nointerfacedata`	Указывает компилятору на необходимость пропуска ресурса, который обычно добавляется в сборку F#, включающую метаданные, связанные с.
`--nologo`	Не отображает текст баннера при запуске компилятора.
`--nooptimizationdata`	Указывает компилятору включать только оптимизацию, необходимую для реализации встроенных конструкций. Запрещает встраивание между модулями, но повышает совместимость двоичных файлов.

ПАРАМЕТР КОМПИЛЯТОРА	ОПИСАНИЕ
`--nowin32manifest`	Указывает компилятору пропустить манифест Win32 по умолчанию.
`--nowarn:warning-number-list`	Отключает конкретные предупреждения, перечисленные по номеру. Разделите каждый номер предупреждения на запятую. Вы можете узнать номер предупреждения для любого предупреждения из выходных данных компиляции. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в разделе /параметры (компилятора# с)параметром "Warn".
`--optimize[+|-][optimization-option-list]` `-O[+|-] [optimization-option-list]`	Включает или отключает оптимизацию. Некоторые параметры оптимизации можно отключить или включить выборочно, перечисляя их. Это: `nojitoptimize`, `nojittracking`, `nolocaloptimize`, `nocrossoptimize`, `notailcalls`.
`--out:output-filename` `-o:output-filename`	Указывает имя скомпилированной сборки или модуля. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора C.
`--pdb:pdb-filename`	Называет выходной файл отладки PDB (база данных программы). Этот параметр применяется только в том случае, если включено также `--debug`. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора PDB.
`--platform:platform-name`	Указывает, что созданный код будет выполняться только на указанной платформе (`x86`, `Itanium` или `x64`) или, если выбрано `anycpu` "имя платформы", указывает, что созданный код может выполняться на любой платформе. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора платформы C.
`--preferreduilang:lang`	Указывает предпочтительное имя языка и региональных параметров для вывода (например, `es-ES`, `ja-JP`).
`--quotations-debug`	Указывает, что для выражений, которые являются производными от F# литералов кавычек и отраженными определениями, должны быть созданы дополнительные отладочные данные. Отладочная информация добавляется к пользовательским атрибутам узла F# дерева выражения. См. раздел цитирование кода и выражение expr. CustomAttributes.

ПАРАМЕТР КОМПИЛЯТОРА	ОПИСАНИЕ
`--reference:assembly-filename` `-r:assembly-filename`	Делает код из сборки F# или .NET Framework доступной для компилируемого кода. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (Справочник# по)параметрам компилятора C.
`--resource:resource-filename`	Внедряет управляемый файл ресурсов в созданную сборку. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора в Resource C.
`--sig:signature-filename`	Создает файл сигнатуры на основе созданной сборки. Дополнительные сведения о файлах сигнатур см. в разделе сигнатуры.
`--simpleresolution`	Указывает, что ссылки на сборки должны быть разрешены с помощью правил Mono на основе каталога, а не разрешения MSBuild. По умолчанию используется разрешение MSBuild, кроме случаев, когда выполняется в Mono.
`--standalone`	Указывает на создание сборки, содержащей все ее зависимости, чтобы она выполнялась сама по себе без необходимости в дополнительных сборках, таких как F# библиотека.
`--staticlink:assembly-name`	Статически связывает заданную сборку и все связанные с ней библиотеки DLL, которые зависят от этой сборки. Используйте имя сборки, а не имя библиотеки DLL.
`--subsystemversion`	Указывает версию подсистемы ОС, используемую создаваемым исполняемым файлом. Используйте 6,02 для Windows 8.1, 6,01 для Windows 7, 6,00 для Windows Vista. Этот параметр применяется только к исполняемым файлам, а не к DLL и должен использоваться только в том случае, если приложение зависит от определенных функций безопасности, доступных только в определенных версиях ОС. Если используется этот параметр и пользователь пытается выполнить приложение в более ранней версии операционной системы, он завершится с сообщением об ошибке.
`--tailcalls[+|-]`	Включает или отключает использование инструкции IL с префиксом tail, которая приводит к повторному использованию кадра стека для заключительных рекурсивных функций. Этот параметр по умолчанию включен.

ПАРАМЕТР КОМПИЛЯТОРА	ОПИСАНИЕ
`--target:[exe|winexe|library|module] filename`	Указывает тип и имя файла созданного скомпилированного кода. • `exe` означает консольное приложение. • `winexe` означает приложение Windows, которое отличается от консольного приложения тем, что не имеет определенных стандартных потоков ввода-вывода (stdin, stdout и stderr). • `library` — это сборка без точки входа. • `module` — это .NET Frameworkный модуль (.netmodule), который впоследствии можно объединить с другими модулями в сборку. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора целевого языка C.
`--times`	Отображает сведения о времени для компиляции.
`--utf8output`	Включает вывод компилятора печати в кодировке UTF-8.
`--warn:warning-level`	Задает уровень предупреждений (от 0 до 5). Уровень по умолчанию — 3. Каждому предупреждению присваивается уровень, основанный на его серьезности. Уровень 5 дает больше, но менее серьезных предупреждений, чем уровень 1. Предупреждения уровня 5:21 (рекурсивное использование, проверяемое во время выполнения), 22 (`let rec` вычисляется в неупорядоченном виде), 45 (полная абстракция) и 52 (защитная копия). Все остальные предупреждения имеют уровень 2. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (предупреждение# о параметрах)компилятора C.
`--warnon:warning-number-list`	Включение некоторых предупреждений, которые могут быть отключены по умолчанию или отключены другим параметром командной строки. В F# 3,0 по умолчанию только предупреждение 1182 (неиспользуемые переменные) отключено.

ПАРАМЕТР КОМПИЛЯТОРА	ОПИСАНИЕ
`--warnaserror[+|-] [warning-number-list]`	Включает или отключает параметр для вывода предупреждений как ошибок. Вы можете указать конкретные номера предупреждений, которые будут отключены или включены. Параметры, приведенные далее в командной строке, переопределяют параметры, указанные ранее в командной строке. Например, чтобы указать предупреждения, которые не должны выводиться как ошибки, укажите `--warnaserror+` `--warnaserror-:warning-number-list`. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора warnaserror C.
`--win32manifest:manifest-filename`	Добавляет файл манифеста Win32 в компиляцию. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в /разделе (параметры#)компилятора win32manifest C.
`--win32res:resource-filename`	Добавляет файл ресурсов Win32 в компиляцию. Этот параметр компилятора эквивалентен параметру C# компилятора с тем же именем. Дополнительные сведения см. в разделе /параметры()компилятора#win32res (C).

Связанные статьи

ЗАГОЛОВОК	ОПИСАНИЕ
Параметры окна "Интерактивный F#"	Описывает параметры командной строки, поддерживаемые F# интерпретатором, FSI. exe.
Справочник по свойствам проектов	Описывает пользовательский интерфейс для проектов, включая страницы свойств проекта, которые предоставляют параметры сборки.

Параметры F# Interactive

08.01.2020 • 7 minutes to read • Edit Online

> **NOTE**
> Сейчас эта статья описывает только соответствующую процедуру для Windows. Позднее она будет переписана.

В этом разделе описываются параметры командной строки, поддерживаемые F# интерактивной `fsi.exe`. F#Интерактивное взаимодействие принимает многие из тех же параметров командной строки F# , что и компилятор, но также принимает некоторые дополнительные параметры.

Использование F# функции Interactive для создания сценариев

F#Интерактивный, `fsi.exe`, может быть запущен в интерактивном режиме, или его можно запустить из командной строки для выполнения скрипта. Синтаксис командной строки:

```
> fsi.exe [options] [ script-file [arguments] ]
```

Расширение файла для F# файлов скрипта — `.fsx`.

Таблица F# интерактивных параметров

В следующей таблице перечислены параметры, поддерживаемые интерактивной F# средой. Эти параметры можно задать в командной строке или в интегрированной среде разработки Visual Studio. Чтобы задать эти параметры в интегрированной среде разработки Visual Studio, откройте меню **Сервис** , выберите **Параметры...** , затем разверните узел **F# средства** и выберите **F# интерактивный**.

Если списки отображаются в F# аргументах интерактивного параметра, элементы списка разделяются точкой с запятой (`;`).

ПАРАМЕТР	ОПИСАНИЕ
--	Используется для указания F# интерактивности обрабатывать оставшиеся аргументы как аргументы командной строки для F# программы или скрипта, к которым можно получить доступ в коде с помощью списка **FSI. CommandLineArgs**.
--checked[**+** | **-**]	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--codepage:<int>	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--консолеколорс[**+** | **-**]	Выводит предупреждения и сообщения об ошибках в цвете.
--кроссоптимизе[**+** | **-**]	Включает или отключает оптимизацию между модулями.

ПАРАМЕТР	ОПИСАНИЕ
--Debug[+ | -] **--Debug:** [**полный**||**переносимый**|**внедренный**] **-g**[+ | -] **-g:** [**полный**|**pdbonly**|**переносимый**|**внедренный**]	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--define:<строка>	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--детерминированный[+ | -]	Создает детерминированную сборку (включая GUID версии модуля и метку времени).
--exec	Указывает, F# что Interactive будет выходить после загрузки файлов или запуска файла скрипта, заданного в командной строке.
--fullpaths	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--GUI[+ | -]	Включает или отключает цикл обработки событий Windows Forms. По умолчанию этот параметр включен.
--Справка **-?**	Используется для вывода синтаксиса командной строки и краткого описания каждого параметра.
--lib:<папка-список> **-l:<список папок>**	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--Load:<имя файла>	Компилирует заданный исходный код при запуске и загружает скомпилированные F# конструкции в сеанс. Если целевой источник содержит директивы скрипта, такие как **#use** или **#load**, необходимо использовать параметр **--use** или **#use** вместо **--Load** или **#load**.
--млкомпатибилити	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--.NET Framework	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе параметры компилятора .
--Эмблема	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--warn:<Warning-List>	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.

ПАРАМЕТР	ОПИСАНИЕ
--optimize[+ | -]	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--preferreduilang:<lang>	Указывает предпочтительное имя языка и региональных параметров на языке вывода (например, ES-ES, ja-JP).
--quiet	Подавлять F# выходные данные интерактивного вывода в поток **stdout** .
--quotations-debug	Указывает, что для выражений, которые являются производными от F# литералов кавычек и отраженными определениями, должны быть созданы дополнительные отладочные данные. Отладочная информация добавляется к пользовательским атрибутам узла F# дерева выражения. См. раздел цитирование кода и выражение expr. CustomAttributes.
--ReadLine[+ | -]	Включение или отключение заполнения нажатием клавиши TAB в интерактивном режиме.
--Reference:<имя файла> **-r:<имя файла>**	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--шадовкопиреференцес[+ | -]	Предотвращает блокировку ссылок F# интерактивным процессом.
--симплересолутион	Разрешает ссылки на сборки с помощью правил на основе каталога, а не разрешения MSBuild.
--таилкаллс[+ | -]	Включение или выключение использования инструкции IL, которая приводит к повторному использованию кадра стека для заключительных рекурсивных функций. Этот параметр по умолчанию включен.
--targetprofile: строка<>	Указывает профиль целевой платформы для этой сборки. Допустимые значения: mscorlib, netcore или netstandard. Значение по умолчанию — mscorlib.
--use:<filename>	Указывает интерпретатору использовать заданный файл при запуске в качестве начального ввода.
--utf8output	То же, что и параметр компилятора FSC. exe. Дополнительные сведения см. в разделе Параметры компилятора.
--warn:<> уровня предупреждения	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.
--warnaserror[+ | -]	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.

ПАРАМЕТР	ОПИСАНИЕ
--warnaserror[**+** | **-**]: **<int-list>**	То же, что и параметр компилятора **FSC. exe** . Дополнительные сведения см. в разделе Параметры компилятора.

Связанные разделы

ЗАГОЛОВОК	ОПИСАНИЕ
Параметры компилятора	Описывает параметры командной строки, доступные для F# компилятора, **FSC. exe**.

Идентификаторы Source Line, File и Path

23.10.2019 • 2 minutes to read • Edit Online

Идентификаторы `__LINE__` `__SOURCE_DIRECTORY__` и `__SOURCE_FILE__` являются встроенными значениями, которые позволяют получить доступ к номеру исходной строки, каталогу и имени файла в коде.

Синтаксис

```
__LINE__
__SOURCE_DIRECTORY__
__SOURCE_FILE__
```

Примечания

Каждое из этих значений имеет тип `string`.

В следующей таблице перечислены идентификаторы исходной строки, файла и пути, доступные в F#. Эти идентификаторы не являются макросами препроцессора; они представляют собой встроенные значения, распознаваемые компилятором.

ПРЕДОПРЕДЕЛЕННЫЙ ИДЕНТИФИКАТОР	ОПИСАНИЕ
`__LINE__`	Вычисляется до текущего номера строки, учитывая `#line` директивы.
`__SOURCE_DIRECTORY__`	Вычисляет текущий полный путь к исходному каталогу, учитывая `#line` директивы.
`__SOURCE_FILE__`	Вычисляет текущее имя исходного файла без пути, учитывая `#line` директивы.

Дополнительные сведения об директиве `#line` см. в разделе директивы компилятора.

Пример

В следующем примере кода показано использование этих значений.

```
let printSourceLocation() =
    printfn "Line: %s" __LINE__
    printfn "Source Directory: %s" __SOURCE_DIRECTORY__
    printfn "Source File: %s" __SOURCE_FILE__
printSourceLocation()
```

Результат

```
Line: 4
Source Directory: C:\Users\username\Documents\Visual Studio 2017\Projects\SourceInfo\SourceInfo
Source File: Program.fs
```

См. также

- Директивы компилятора
- Справочник по языку F#

Сведения о вызывающем объекте

25.11.2019 • 4 minutes to read • Edit Online

С помощью информационных атрибутов вызывающего объекта можно получить сведения о вызывающем объекте метода. Можно получить путь к файлу исходного кода, номер строки в исходном коде и имя вызывающего объекта. Эти сведения полезны для трассировки, отладки и создания средств диагностики.

Для получения этих сведений используются атрибуты, которые применяются к необязательным параметрам, каждый из которых имеет значение по умолчанию. В следующей таблице перечислены атрибуты сведений о вызывающем объекте, определенные в пространстве имен System. Runtime. CompilerServices :

АТРИБУТ	ОПИСАНИЕ	TYPE
каллерфилепас	Полный путь исходного файла, содержащего вызывающий объект. Это путь к файлу во время компиляции.	`String`
каллерлиненумбер	Номер строки в исходном файле, в которой вызывается метод.	`Integer`
CallerMemberName	Имя свойства или метода вызывающего объекта. См. раздел имена членов далее в этом разделе.	`String`

Пример

В следующем примере показано, как с помощью этих атрибутов можно отслеживать вызывающий объект.

```
open System.Diagnostics
open System.Runtime.CompilerServices
open System.Runtime.InteropServices

type Tracer() =
    member _.DoTrace(message: string,
                     [<CallerMemberName; Optional; DefaultParameterValue("")>] memberName: string,
                     [<CallerFilePath; Optional; DefaultParameterValue("")>] path: string,
                     [<CallerLineNumber; Optional; DefaultParameterValue(0)>] line: int) =
        Trace.WriteLine(sprintf "Message: %s" message)
        Trace.WriteLine(sprintf "Member name: %s" memberName)
        Trace.WriteLine(sprintf "Source file path: %s" path)
        Trace.WriteLine(sprintf "Source line number: %d" line)
```

Заметки

Атрибуты сведений о вызывающем объекте могут применяться только к необязательным параметрам. Атрибуты сведений о вызывающем объекте приводят к тому, что компилятор записывает правильное значение для каждого необязательного параметра, дополненного атрибутом сведений о вызывающем объекте.

Информационные значения вызывающего объекта передаются как литералы в IL во время компиляции. В отличие от результатов свойства StackTrace для исключений, на результаты не влияет маскировка.

Можно явно передать необязательные аргументы, чтобы управлять сведениями о вызывающем объекте или скрывать сведения о вызывающем объекте.

Имена членов

Можно использовать атрибут `CallerMemberName` , чтобы не указывать имя члена в качестве аргумента `String` вызываемого метода. С помощью этого метода можно избежать проблем, при которых Рефакторинг переименования не изменяет значения `String` . Это особенно полезно при выполнении следующих задач:

- Использование процедур трассировки и диагностики.
- Реализация интерфейса INotifyPropertyChanged при привязке данных. Этот интерфейс позволяет свойству объекта уведомлять связанный элемент управления об изменении свойства, чтобы элемент управления мог отображать обновленные сведения. Без атрибута `CallerMemberName` необходимо указать имя свойства в виде литерала.

На следующей диаграмме показаны имена элементов, возвращаемых при использовании атрибута CallerMemberName.

ФРАГМЕНТ КОДА, В ПРЕДЕЛАХ КОТОРОГО ПРОИСХОДИТ ВЫЗОВ	РЕЗУЛЬТАТ ИМЕНИ ЧЛЕНА
Метод, свойство или событие	Имя метода, свойства или события, из которого происходил вызов.
Конструктор	Строка ".ctor"
Статический конструктор	Строка ".cctor"
Деструктор	Строка "Finalize"
Определяемые пользователем операторы и преобразования	Созданное имя члена, например, "op_Addition".
Конструктора атрибута	Имя члена, к которому применяется атрибут. Если атрибут — любой элемент внутри члена (например, параметр, возвращаемое значение или параметр универсального типа), то результат — имя члена, который связан с этим элементом.
Нет содержащего члена (например, уровень сборки или атрибуты, примененные к типам)	Значение необязательного параметра по умолчанию.

См. также

- Атрибуты
- Именованные аргументы
- Необязательные параметры

Подробный синтаксис

04.11.2019 • 2 minutes to read • Edit Online

Существует две формы синтаксиса, доступные для многих конструкций F# языка: *подробный синтаксис* и *упрощенный синтаксис*. Синтаксис verbose не так часто используется, но имеет преимущество менее чувствительно к отступу. Упрощенный синтаксис короче и использует отступы для обозначения начала и окончания конструкций, а не дополнительные ключевые слова, такие как `begin`, `end`, `in` и т. д. Синтаксис по умолчанию — упрощенный синтаксис. В этом разделе описывается синтаксис F# конструкций, если упрощенный синтаксис не включен. Подробный синтаксис всегда включен, поэтому даже если включен упрощенный синтаксис, для некоторых конструкций по-прежнему можно использовать подробный синтаксис. Упрощенный синтаксис можно отключить с помощью директивы `#light "off"`.

Таблица конструкций

В следующей таблице показан упрощенный и подробный синтаксис для F# языковых конструкций в контекстах, где существует разница между двумя формами. В этой таблице угловые скобки (<>) заключают определяемые пользователем элементы синтаксиса. Более подробные сведения о синтаксисе, используемом в этих конструкциях, см. в документации по каждой языковой конструкции.

КОНСТРУКЦИЯ ЯЗЫКА	УПРОЩЕННЫЙ СИНТАКСИС	ПОДРОБНЫЙ СИНТАКСИС
составные выражения	`<expression1>` `<expression2>`	`<expression1>; <expression2>`
вложенные привязки `let`	`let f x =` `  let a = 1` `  let b = 2` `  x + a + b`	`let f x =` `  let a = 1 in` `  let b = 2 in` `  x + a + b`
блок кода	`(` `  <expression1>` `  <expression2>` `)`	`begin` `  <expression1>;` `  <expression2>;` `end`
`for...do`	`for counter = start to` `finish do` `  ...`	`for counter = start to` `finish do` `  ...` `done`

`while...do`	`while <condition> do` `    ...`	`while <condition> do` `    ...` `done`
`for...in`	`for var in start .. finish` `do` `    ...`	`for var in start .. finish` `do` `    ...` `done`
`do`	`do` `    ...`	`do` `    ...` `in`
record	`type <record-name> =` `    {` `        <field-declarations>` `    }` `    <value-or-member-definitions>`	`type <record-name> =` `    {` `        <field-declarations>` `    }` `    with` `        <value-or-member-definitions>` `    end`
класс	`type <class-name>(<params>)` `=` `    ...`	`type <class-name>(<params>)` `=` `    class` `        ...` `    end`
структура	`[<StructAttribute>]` `type <structure-name> =` `    ...`	`type <structure-name> =` `    struct` `        ...` `    end`
размеченное объединение	`type <union-name> =` `    | ...` `    | ...` `    ...` `    <value-or-member definitions>`	`type <union-name> =` `    | ...` `    | ...` `    ...` `    with` `        <value-or-member-definitions>` `    end`

interface	`type <interface-name> =` `    ...`	`type <interface-name> =` `    interface` `        ...` `    end`
выражение объекта	`{ new <type-name>` `    with` `        <value-or-member-definitions>` `        <interface-implementations>` `}`	`{ new <type-name>` `    with` `        <value-or-member-definitions>` `    end` `    <interface-implementations>` `}`
реализация интерфейсов	`interface <interface-name>` `    with` `        <value-or-member-definitions>`	`interface <interface-name>` `    with` `        <value-or-member-definitions>` `    end`
расширение типа	`type <type-name>` `    with` `        <value-or-member-definitions>`	`type <type-name>` `    with` `        <value-or-member-definitions>` `    end`
module	`module <module-name> =` `    ...`	`module <module-name> =` `    begin` `        ...` `    end`

См. также

- Справочник по языку F#
- Директивы компилятора
- Рекомендации по форматированию кода

F#руководство по стилю

23.10.2019 • 5 minutes to read • Edit Online

В следующих статьях описываются рекомендации по форматирование F# кода и тематических обеспечению возможности языка, и как их использовать.

В этом руководстве было сформулировать основанная на использовании F# в больших баз кода с разносторонней группы программистов. В этом руководстве, обычно приводит к Успешное использование F# и сводит к минимуму разочарований при изменении требований для программ со временем.

Хорошее пять принципов F# кода

Помните следующие принципы всякий раз при написании F# кода, особенно в системах, которые будут меняться со временем. Каждый фрагмент рекомендации в последующих статьях порождаемой эти пять точек.

1. **Хороший F# кода емкий, выразительный и составных**

   F#имеет множество функций, которые позволяют указать действия в меньшее число строк кода и повторно использовать универсальную функцию. F# Основной библиотеки также содержит множество полезных типы и функции для работы с стандартные коллекции данных. Создание собственных функций и их на F# основной библиотеки (или другие библиотеки) является частью подпрограммы идиоматичной F# программирования. Как правило если решение проблемы в меньшее число строк кода, можно выразить другими разработчиками (или в будущем self) будет заметного. Также настоятельно рекомендуется использовать библиотеку например FSharp.Core, обширной библиотеки .NET , F# выполняется, или пакет на независимых производителей NuGet при необходимости выполните нетривиальную задачу.

2. **Хороший F# кода с возможностью взаимодействия**

   Взаимодействие может занять несколько форм, включая использование кода на разных языках. Границы, прочие вызывающие программы взаимодействовать с кода являются критически важных фрагмента, чтобы получить верно, даже если вызывающим объектам также присутствуют в F#. При написании F#, вы должны всегда думать о том, как другие код будет вызывать код, который вы пишете, включая при этом из другого языка, например C#. F# Рекомендации по проектированию компонентов взаимодействие подробно описывают.

3. **Хороший F# код позволяет использовать объект программирования, не object ориентации**

   F#имеет полную поддержку для программирования с объектами в .NET, включая классы, интерфейсы, модификаторы доступа, абстрактные классы, и т. д. Для более сложных функциональный код, например функции, которые необходимо иметь в виду контекст объекты можно легко инкапсулировать контекстную информацию способами, недоступными в функции. Такие функции, как необязательные параметры и Аккуратное использование перегрузка можно упростить использование этих функций для вызывающих объектов.

4. **Хороший F# код выполняет хорошо, не раскрывая изменений**

   Это не секрет, что для написания кода для высокой производительности, необходимо использовать изменений. Это, как работают компьютеров, в конце концов. Такой код часто ошибок и способствует совершению трудной. Избегайте предоставления изменений вызывающим объектам. Вместо этого сборки функциональное интерфейс, который скрывает реализацию на основе изменений когда важна производительность.

5. **Хороший F# читаемого вспомогательного кода**

   Средства являются очень полезными для работы в больших базах кода, а также можно написать F# кода таким образом, он может использоваться более эффективно с F# средств языка. Одним из примеров нужно убедиться, что вы не переборщить с без использования точки порождает стиль программирования, таким образом, чтобы промежуточных значений, которые можно проверить с помощью отладчика. Другим примером является использование комментарии XML-документации описания конструкции, таким образом, что всплывающие подсказки в редакторах можно отобразить эти комментарии на сайте вызова. Всегда думайте о как ваш код будет прочитана, переход, отладке и управлять другими программистами с их средствами.

Следующие шаги

F# Рекомендациями по форматированию кода рекомендации о том, как для форматирования кода, так как это удобном для чтения.

F# Соглашения о написании кода рекомендации по F# идиомы, которые помогут долгосрочного обслуживания более крупных программирования F# базы кода.

F# Рекомендации по проектированию компонентов предоставляют рекомендации по разработке F# компонентов, таких как библиотеки.

Рекомендации по форматированию кода F#

25.11.2019 • 27 minutes to read • Edit Online

В этой статье приводятся рекомендации по форматированию кода, чтобы F# код:

- Обычно просматривается как более разборчиво
- В соответствии с соглашениями, применяемыми средствами форматирования в Visual Studio и других редакторах
- Аналогично другому коду в Интернете

Эти рекомендации основаны на исчерпывающем руководстве по F# форматированию соглашений с помощью AHX-dung Phan.

Общие правила для отступов

F#по умолчанию использует значащие пробелы. Следующие рекомендации предназначены для указания того, как решать некоторые проблемы, которые могут накладываться.

Использование пробелов

Если требуется отступ, необходимо использовать пробелы, а не символы табуляции. Требуется по крайней мере один пробел. Ваша организация может создавать стандарты кодирования для указания количества пробелов, используемых для отступов; два, три или четыре пробела на каждом уровне, где происходит отступ, являются типичными.

Для отступов рекомендуется использовать по 4 пространства.

С другой стороны, отступы программ являются субъективным вопросом. Варианты — OK, но первым правилом, которое следует следовать, является *согласованность отступов*. Выберите общепринятый стиль отступов и используйте его систематически во всей базе кода.

Форматирование пробелов

F#учитывает пробельные символы. Хотя большая семантика из пустого пространства охватывается правильным отступом, необходимо учитывать некоторые другие моменты.

Операторы форматирования в арифметических выражениях

Всегда используйте пробелы вокруг двоичных арифметических выражений:

```
let subtractThenAdd x = x - 1 + 3
```

Унарные операторы `-` всегда должны иметь значение, которое они отменяют сразу после:

```
// OK
let negate x = -x

// Bad
let negateBad x = - x
```

Добавление символа пробела после оператора `-` может привести к путанице в других случаях.

В целом, важно всегда:

- Заключить бинарные операторы в пробелы
- Никогда не иметь конечных пробелов после унарного оператора

Особенно важна рекомендация бинарных арифметических операторов. Если не заключить бинарный `-` оператор, в сочетании с определенными вариантами форматирования, может привести к интерпретации его как унарного `-`.

Заключить определение пользовательского оператора с помощью пробела

Всегда используйте пробелы, чтобы заключить определение оператора:

```
// OK
let ( !> ) x f = f x

// Bad
let (!>) x f = f x
```

Для любого пользовательского оператора, начинающегося с `*` и имеющего более одного символа, необходимо добавить пробел в начало определения, чтобы избежать неоднозначности компилятора. Поэтому рекомендуется просто заключить определения всех операторов в один символ пробела.

Стрелки параметров вокруг функции с пробелами

При определении сигнатуры функции используйте пробел вокруг `->` символа:

```
// OK
type MyFun = int -> int -> string

// Bad
type MyFunBad = int->int->string
```

Аргументы функции вокруг пробелов

При определении функции следует использовать пробел вокруг каждого аргумента.

```
// OK
let myFun (a: decimal) b c = a + b + c

// Bad
let myFunBad (a:decimal)(b)c = a + b + c
```

Размещайте параметры в новой строке для слишком длинных определений элементов

При наличии слишком длинного определения члена разместите параметры на новых строках и поставьте отступ для одной области.

```
type C() =
    member _.LongMethodWithLotsOfParameters(
        aVeryLongType: AVeryLongTypeThatYouNeedToUse
        aSecondVeryLongType: AVeryLongTypeThatYouNeedToUse
        aThirdVeryLongType: AVeryLongTypeThatYouNeedToUse) =
        // ... the body of the method follows
```

Это также относится к конструкторам:

```
type C(
    aVeryLongType: AVeryLongTypeThatYouNeedToUse
    aSecondVeryLongType: AVeryLongTypeThatYouNeedToUse
    aThirdVeryLongType: AVeryLongTypeThatYouNeedToUse) =
    // ... the body of the class follows
```

Аннотации типов

Заметки о типе аргумента функции в правой панели

При определении аргументов с аннотациями типа используйте пробел после `:` символа:

```
// OK
let complexFunction (a: int) (b: int) c = a + b + c

// Bad
let complexFunctionBad (a :int) (b :int) (c:int) = a + b + c
```

Заметка возвращаемого типа вокруг пробелов

В заметке функции, привязанной к let, или типе значения (возвращаемый тип в случае функции) используйте пробелы до и после символа `:`:

```
// OK
let expensiveToCompute : int = 0 // Type annotation for let-bound value
let myFun (a: decimal) b c : decimal = a + b + c // Type annotation for the return type of a function
// Bad
let expensiveToComputeBad1:int = 1
let expensiveToComputeBad2 :int = 2
let myFunBad (a: decimal) b c:decimal = a + b + c
```

Форматирование пустых строк

- Отдельные определения функций и классов верхнего уровня с двумя пустыми строками.
- Определения методов внутри класса разделяются одной пустой строкой.
- Для разделения групп связанных функций можно использовать дополнительные пустые строки (с осторожностью). Пустые строки могут быть опущены между несколькими строками с одной строкой (например, набором фиктивных реализаций).
- Для обозначения логических разделов используйте в функциях пустые строки.

Форматирование комментариев

Обычно в качестве комментариев блока в стиле ML предпочтительно несколько комментариев с двойной косой чертой.

```
// Prefer this style of comments when you want
// to express written ideas on multiple lines.

(*
    ML-style comments are fine, but not a .NET-ism.
    They are useful when needing to modify multi-line comments, though.
*)
```

Встроенные комментарии должны заменять первую букву прописной.

```
let f x = x + 1 // Increment by one.
```

Соглашения об именах

Использовать camelCase для привязанных к классам значений и функций, связанных с шаблонами, и функциями

Распространенный и принятый F# стиль для использования camelCase для всех имен, привязанных как локальные переменные или в соответствии с шаблонами и определениями функций.

```
// OK
let addIAndJ i j = i + j

// Bad
let addIAndJ I J = I+J

// Bad
let AddIAndJ i j = i + j
```

Функции с локальной привязкой в классах также должны использовать camelCase.

```
type MyClass() =

    let doSomething () =

    let firstResult = ...

    let secondResult = ...

    member x.Result = doSomething()
```

Использование camelCase для открытых функций, привязанных к модулю

Если функция, привязанная к модулю, является частью общедоступного API, она должна использовать camelCase:

```
module MyAPI =
    let publicFunctionOne param1 param2 param2 = ...

    let publicFunctionTwo param1 param2 param3 = ...
```

Использовать camelCase для внутренних и частных привязанных к модулю значений и функций

Используйте camelCase для частных значений, привязанных к модулю, включая следующие:

- Специальные функции в скриптах
- Значения, составляющие внутреннюю реализацию модуля или типа

```
let emailMyBossTheLatestResults =
    ...
```

Использование camelCase для параметров

Все параметры должны использовать camelCase в соответствии с соглашениями об именовании .NET.

```
module MyModule =
    let myFunction paramOne paramTwo = ...

type MyClass() =
    member this.MyMethod(paramOne, paramTwo) = ...
```

Использование PascalCase для модулей

Все модули (верхний, внутренний, частный, вложенный) должны использовать PascalCase.

```
module MyTopLevelModule

module Helpers =
    module private SuperHelpers =
        ...

    ...
```

Использование PascalCase для объявлений типов, элементов и меток

Классы, интерфейсы, структуры, перечисления, делегаты, записи и размеченные объединения должны иметь имена с PascalCase. Элементы в типах и метках для записей и размеченных объединений должны также использовать PascalCase.

```
type IMyInterface =
    abstract Something: int

type MyClass() =
    member this.MyMethod(x, y) = x + y

type MyRecord = { IntVal: int; StringVal: string }

type SchoolPerson =
    | Professor
    | Student
    | Advisor
    | Administrator
```

Использование PascalCase для конструкций, встроенных в .NET

Пространства имен, исключения, события и имена проектов и `.dll` также должны использовать PascalCase. Это не только делает использование других языков .NET более естественным для потребителей, но также согласуется с соглашениями об именовании .NET, которые, скорее всего, встречаются.

Не используйте знаки подчеркивания в именах

Исторически в некоторых F# библиотеках в именах используются символы подчеркивания. Однако он больше не принимается частично, поскольку он противоречит соглашениям об именовании .NET. С другой стороны, F# некоторые программисты часто используют подчеркивания, частично по историческим причинам, а чувствительность и уважение важны. Однако имейте в виду, что стиль часто отличается от других, которые имеют возможность выбрать, следует ли использовать его.

Некоторые исключения включают взаимодействие с собственными компонентами, в которых знаки подчеркивания очень распространены.

Использовать стандартные F# операторы

Следующие операторы определены в F# стандартной библиотеке и должны использоваться вместо определения эквивалентов. Рекомендуется использовать эти операторы, так как он, как правило, делает код более читаемым и идиоматическим. Разработчики с фоном в OCaml или другом языке функционального программирования могут прилагаться к различным идиомам. В следующем списке перечислены рекомендуемые F# операторы.

```
x |> f // Forward pipeline
f >> g // Forward composition
x |> ignore // Discard away a value
x + y // Overloaded addition (including string concatenation)
x - y // Overloaded subtraction
x * y // Overloaded multiplication
x / y // Overloaded division
x % y // Overloaded modulus
x && y // Lazy/short-cut "and"
x || y // Lazy/short-cut "or"
x <<< y // Bitwise left shift
x >>> y // Bitwise right shift
x ||| y // Bitwise or, also for working with “flags” enumeration
x &&& y // Bitwise and, also for working with “flags” enumeration
x ^^^ y // Bitwise xor, also for working with “flags” enumeration
```

Использовать префиксный синтаксис для универсальных типов (`Foo<T>`) в качестве предпочтительного для постфиксного синтаксиса (`T Foo`)

F#наследует постфиксный стиль именования универсальных типов (например, `int list`), а также стиль префикса .NET (например, `list<int>`). Предпочитать стиль .NET, за исключением пяти конкретных типов:

1. Для F# списков используйте постфиксную форму: `int list` , а не `list<int>` .
2. Для F# параметров используйте постфиксную форму: `int option` , а не `option<int>` .
3. Для F# параметров значения используйте постфиксную форму: `int voption` , а не `voption<int>` .
4. Для F# массивов используйте синтаксические имена `int[]` а не `int array` или `array<int>` .
5. Для ссылочных ячеек используйте `int ref` , а не `ref<int>` или `Ref<int>` .

Для всех остальных типов используйте форму префикса.

Форматирование кортежей

Создание экземпляра кортежа должно быть заключено в круглые скобки, а за разделителями в пределах должно быть один пробел, например: `(1, 2)` , `(x, y, z)` .

Обычно можно опустить круглые скобки в шаблоне, сопоставленном с кортежами:

```
let (x, y) = z // Destructuring
let x, y = z // OK

// OK
match x, y with
| 1, _ -> 0
| x, 1 -> 0
| x, y -> 1
```

Также обычно можно опустить круглые скобки, если кортеж является возвращаемым значением функции:

```
// OK
let update model msg =
    match msg with
    | 1 -> model + 1, []
    | _ -> model, [ msg ]
```

В заключение следует предпочесть создание экземпляров кортежей в круглых скобках, но при использовании кортежей для сопоставления шаблонов или возвращаемого значения оно считается точным, чтобы избежать круглых скобок.

Форматирование объявлений размеченного объединения

Отступ `|` в определении типа по 4 пробелам:

```
// OK
type Volume =
    | Liter of float
    | FluidOunce of float
    | ImperialPint of float

// Not OK
type Volume =
| Liter of float
| USPint of float
| ImperialPint of float
```

Форматирование размеченных объединений

Экземпляры с различенными объединениями, разделенными по нескольким строкам, должны предоставлять новой области с отступами следующие данные.

```
let tree1 =
    BinaryNode
        (BinaryNode(BinaryValue 1, BinaryValue 2),
         BinaryNode(BinaryValue 3, BinaryValue 4))
```

Закрывающая круглая скобка также может находиться на новой строке:

```
let tree1 =
    BinaryNode(
        BinaryNode(BinaryValue 1, BinaryValue 2),
        BinaryNode(BinaryValue 3, BinaryValue 4)
    )
```

Форматирование объявлений записей

Понизить `{` в определении типа на 4 пробела и начать список полей в той же строке:

```
// OK
type PostalAddress =
    { Address: string
      City: string
      Zip: string }
    member x.ZipAndCity = sprintf "%s %s" x.Zip x.City

// Not OK
type PostalAddress =
  { Address: string
    City: string
    Zip: string }
    member x.ZipAndCity = sprintf "%s %s" x.Zip x.City

// Unusual in F#
type PostalAddress =
    {
        Address: string
        City: string
        Zip: string
    }
```

Помещение открывающего маркера в новую строку и закрывающий маркер в новой строке предпочтительнее, если объявить реализации интерфейса или элементы в записи:

```
// Declaring additional members on PostalAddress
type PostalAddress =
    {
        Address: string
        City: string
        Zip: string
    } with
    member x.ZipAndCity = sprintf "%s %s" x.Zip x.City

type MyRecord =
    {
        SomeField: int
    }
    interface IMyInterface
```

Форматирование записей

Короткие записи можно записать в одной строке:

```
let point = { X = 1.0; Y = 0.0 }
```

Записи, которые больше не должны использовать новые строки для меток:

```
let rainbow =
    { Boss = "Jeffrey"
      Lackeys = ["Zippy"; "George"; "Bungle"] }
```

Размещение открывающего маркера на новой строке, содержимое, вкладка над одной областью, и закрывающий маркер в новой строке предпочтительнее, если вы:

- Перемещение записей в коде с разными областями отступов
- Передача их в функцию

```
let rainbow =
    {
        Boss1 = "Jeffrey"
        Boss2 = "Jeffrey"
        Boss3 = "Jeffrey"
        Boss4 = "Jeffrey"
        Boss5 = "Jeffrey"
        Boss6 = "Jeffrey"
        Boss7 = "Jeffrey"
        Boss8 = "Jeffrey"
        Lackeys = ["Zippy"; "George"; "Bungle"]
    }

type MyRecord =
    {
        SomeField: int
    }
    interface IMyInterface

let foo a =
    a
    |> Option.map (fun x ->
        {
            MyField = x
        })
```

Для списка и элементов массива применяются те же правила.

Форматирование выражений записи копирования и обновления

Выражение записи копирования и обновления по-прежнему является записью, поэтому применяются аналогичные рекомендации.

Короткие выражения могут помещаться в одну строку:

```
let point2 = { point with X = 1; Y = 2 }
```

В более длинных выражениях должны использоваться новые строки:

```
let rainbow2 =
    { rainbow with
        Boss = "Jeffrey"
        Lackeys = ["Zippy"; "George"; "Bungle"] }
```

Как и в руководстве по записям, может потребоваться выделить отдельные строки для фигурных скобок и задать отступ для одной области справа с помощью выражения. Обратите внимание, что в некоторых особых случаях, например при переносе значения с необязательными скобками, может потребоваться разместить фигурную скобку в одной строке:

```
type S = { F1: int; F2: string }
type State = { F:  S option }

let state = { F = Some { F1 = 1; F2 = "Hello" } }
let newState =
    {
        state with
            F = Some {
                    F1 = 0
                    F2 = ""
                }
    }
```

Форматирование списков и массивов

Напишите `x :: l` с пробелами вокруг оператора `::` (`::` является оператором инфиксные, поэтому он заключен в пробелы).

Список и массивы, объявленные в одной строке, должны содержать пробел после открывающей скобки и перед закрывающей скобкой:

```
let xs = [ 1; 2; 3 ]
let ys = [| 1; 2; 3; |]
```

Всегда используйте хотя бы один пробел между двумя различными операторами, похожими на фигурные скобки. Например, оставьте пробел между `[` и `{`.

```
// OK
[ { IngredientName = "Green beans"; Quantity = 250 }
  { IngredientName = "Pine nuts"; Quantity = 250 }
  { IngredientName = "Feta cheese"; Quantity = 250 }
  { IngredientName = "Olive oil"; Quantity = 10 }
  { IngredientName = "Lemon"; Quantity = 1 } ]

// Not OK
[{ IngredientName = "Green beans"; Quantity = 250 }
 { IngredientName = "Pine nuts"; Quantity = 250 }
 { IngredientName = "Feta cheese"; Quantity = 250 }
 { IngredientName = "Olive oil"; Quantity = 10 }
 { IngredientName = "Lemon"; Quantity = 1 }]
```

Одно и то же правило применимо к спискам или массивам кортежей.

Списки и массивы, разделенные по нескольким строкам, следуют тому же правилу, что и записи:

```
let pascalsTriangle =
    [|
        [| 1 |]
        [| 1; 1 |]
        [| 1; 2; 1 |]
        [| 1; 3; 3; 1 |]
        [| 1; 4; 6; 4; 1 |]
        [| 1; 5; 10; 10; 5; 1 |]
        [| 1; 6; 15; 20; 15; 6; 1 |]
        [| 1; 7; 21; 35; 35; 21; 7; 1 |]
        [| 1; 8; 28; 56; 70; 56; 28; 8; 1 |]
    |]
```

И, как и в случае с записями, объявление открывающих и закрывающих квадратных скобок в собственной

строке сделает код более простым и походит к функциям.

При создании массивов и списков программным способом предпочтительнее `->` для `do ... yield` при создании значения всегда:

```
// Preferred
let squares = [ for x in 1..10 -> x*x ]

// Not preferred
let squares' = [ for x in 1..10 do yield x*x ]
```

Более старые версии F# языка требовали указания `yield` в ситуациях, когда данные могут создаваться условно, или для вычисления последовательных выражений. Рекомендуется опустить эти ключевые слова `yield`, если не требуется компиляция с использованием F# более старой версии языка:

```
// Preferred
let daysOfWeek includeWeekend =
    [
        "Monday"
        "Tuesday"
        "Wednesday"
        "Thursday"
        "Friday"
        if includeWeekend then
            "Saturday"
            "Sunday"
    ]

// Not preferred
let daysOfWeek' includeWeekend =
    [
        yield "Monday"
        yield "Tuesday"
        yield "Wednesday"
        yield "Thursday"
        yield "Friday"
        if includeWeekend then
            yield "Saturday"
            yield "Sunday"
    ]
```

В некоторых случаях `do...yield` может помочь в удобочитаемости. В этих случаях следует учитывать субъективные ситуации.

Форматирование выражений if

Отступы условий зависят от размеров выражений, составляющих их. Если `cond`, `e1` и `e2` короткие, просто запишите их на одной строке:

```
if cond then e1 else e2
```

Если один из `cond`, `e1` или `e2` больше, но не является многострочным:

```
if cond
then e1
else e2
```

Если любое из выражений имеет несколько строк:

```
if cond then
    e1
else
    e2
```

Несколько условий с `elif` и `else` отображаются с отступом в той же области, что и `if`:

```
if cond1 then e1
elif cond2 then e2
elif cond3 then e3
else e4
```

Конструкции сопоставления шаблонов

Используйте `|` для каждого предложения соответствия без отступов. Если выражение является коротким, можно использовать одну строку, если каждая часть выражения также является простой.

```
// OK
match l with
| { him = x; her = "Posh" } :: tail -> x
| _ :: tail -> findDavid tail
| [] -> failwith "Couldn't find David"

// Not OK
match l with
    | { him = x; her = "Posh" } :: tail -> x
    | _ :: tail -> findDavid tail
    | [] -> failwith "Couldn't find David"
```

Если выражение справа от стрелки сопоставления шаблонов слишком велико, переместите его в следующую строку с отступом на один шаг из `match` / `|`.

```
match lam with
| Var v -> 1
| Abs(x, body) ->
    1 + sizeLambda body
| App(lam1, lam2) ->
    sizeLambda lam1 + sizeLambda lam2
```

Шаблон сопоставления анонимных функций, начиная с `function`, не должен слишком далеко иметь отступы. Например, чтобы задать отступ для одной области, сделайте следующее:

```
lambdaList
|> List.map (function
    | Abs(x, body) -> 1 + sizeLambda 0 body
    | App(lam1, lam2) -> sizeLambda (sizeLambda 0 lam1) lam2
    | Var v -> 1)
```

Сопоставление шаблонов в функциях, определенных `let` или `let rec`, должно иметь отступ в 4 пробелах после начала `let`, даже если используется ключевое слово `function`:

```
let rec sizeLambda acc = function
    | Abs(x, body) -> sizeLambda (succ acc) body
    | App(lam1, lam2) -> sizeLambda (sizeLambda acc lam1) lam2
    | Var v -> succ acc
```

Не рекомендуется выровняйте стрелки.

Форматирование выражений try и with

Сопоставление шаблонов для типа исключения должно иметь отступ на том же уровне, что и `with`.

```
try
    if System.DateTime.Now.Second % 3 = 0 then
        raise (new System.Exception())
    else
        raise (new System.ApplicationException())
with
| :? System.ApplicationException ->
    printfn "A second that was not a multiple of 3"
| _ ->
    printfn "A second that was a multiple of 3"
```

Приложение параметров функции форматирования

Как правило, большинство параметров функции выполняются в одной строке.

Если вы хотите применить параметры к функции в новой строке, понизить их до одной области.

```
// OK
sprintf "\t%s - %i\n\r"
     x.IngredientName x.Quantity

// OK
sprintf
     "\t%s - %i\n\r"
     x.IngredientName x.Quantity

// OK
let printVolumes x =
    printf "Volume in liters = %f, in us pints = %f, in imperial = %f"
        (convertVolumeToLiter x)
        (convertVolumeUSPint x)
        (convertVolumeImperialPint x)
```

Те же рекомендации применяются для лямбда-выражений в качестве аргументов функции. Если тело лямбда-выражения, текст может иметь другую строку, с отступом на одну область

```
let printListWithOffset a list1 =
    List.iter
        (fun elem -> printfn "%d" (a + elem))
        list1

// OK if lambda body is long enough
let printListWithOffset a list1 =
    List.iter
        (fun elem ->
            printfn "%d" (a + elem))
        list1
```

Однако если тело лямбда-выражения является более одной строкой, рекомендуется разбить его на отдельную функцию, а не использовать многострочную конструкцию, применяемую в качестве одного аргумента для функции.

Форматирование операторов инфиксные

Разделяйте операторы по пробелам. Очевидными исключениями из этого правила являются операторы `!` и `.`.

Выражения инфиксные являются допустимыми для сопоставления по одному и тому же столбцу:

```
acc +
(sprintf "\t%s - %i\n\r"
     x.IngredientName x.Quantity)

let function1 arg1 arg2 arg3 arg4 =
    arg1 + arg2 +
    arg3 + arg4
```

Форматирование операторов конвейера

Операторы конвейера `|>` должны идти в соответствии с выражениями, над которыми они работают.

```
// Preferred approach
let methods2 =
    System.AppDomain.CurrentDomain.GetAssemblies()
    |> List.ofArray
    |> List.map (fun assm -> assm.GetTypes())
    |> Array.concat
    |> List.ofArray
    |> List.map (fun t -> t.GetMethods())
    |> Array.concat

// Not OK
let methods2 = System.AppDomain.CurrentDomain.GetAssemblies()
            |> List.ofArray
            |> List.map (fun assm -> assm.GetTypes())
            |> Array.concat
            |> List.ofArray
            |> List.map (fun t -> t.GetMethods())
            |> Array.concat
```

Модули форматирования

Код в локальном модуле должен иметь отступ относительно модуля, но код в модуле верхнего уровня не должен иметь отступы. Элементы пространства имен не обязательно должны иметь отступы.

```
// A is a top-level module.
module A

let function1 a b = a - b * b
```

```
// A1 and A2 are local modules.
module A1 =
    let function1 a b = a*a + b*b

module A2 =
    let function2 a b = a*a - b*b
```

Форматирование выражений и интерфейсов объекта

Выражения объектов и интерфейсы должны быть выровнены так же, как `member` с отступом после 4 пробелов.

```
let comparer =
    { new IComparer<string> with
          member x.Compare(s1, s2) =
              let rev (s: String) =
                  new String (Array.rev (s.ToCharArray()))
              let reversed = rev s1
              reversed.CompareTo (rev s2) }
```

Форматирование пробелов в выражениях

Избегайте появления лишних F# пробелов в выражениях.

```
// OK
spam (ham.[1])

// Not OK
spam ( ham.[ 1 ] )
```

У именованных аргументов также не должно быть пробелов, окружающих `=`:

```
// OK
let makeStreamReader x = new System.IO.StreamReader(path=x)

// Not OK
let makeStreamReader x = new System.IO.StreamReader(path = x)
```

Атрибуты форматирования

Атрибуты помещаются над конструкцией:

```
[<SomeAttribute>]
type MyClass() = ...

[<RequireQualifiedAccess>]
module M =
    let f x = x

[<Struct>]
type MyRecord =
    { Label1: int
      Label2: string }
```

Атрибуты форматирования для параметров

Атрибуты также могут быть размещаются в параметрах. В этом случае поместите его в ту же строку, что и параметр, и перед именем:

```
// Defines a class that takes an optional value as input defaulting to false.
type C() =
    member _.M([<Optional; DefaultParameterValue(false)>] doSomething: bool)
```

Форматирование нескольких атрибутов

Если к конструкции, которая не является параметром, применяется несколько атрибутов, их следует размещать таким образом, чтобы в каждой строке был один атрибут:

```
[<Struct>]
[<IsByRefLike>]
type MyRecord =
    { Label1: int
      Label2: string }
```

При применении к параметру они должны находиться в той же строке и разделяться разделителем `;`.

Литералы форматирования

литералы, использующие атрибут `Literal`, должны располагать атрибут в отдельной строке и использовать именование PascalCase: F#

```
[<Literal>]
let Path = __SOURCE_DIRECTORY__ + "/" + __SOURCE_FILE__

[<Literal>]
let MyUrl = "www.mywebsitethatiamworkingwith.com"
```

Старайтесь не размещать атрибут в той же строке, что и значение.

Соглашения о написании кода на F#

25.11.2019 • 39 minutes to read • Edit Online

Следующие соглашения формируются из опыта работы с большими F# кодовыми средами. Пять принципов работы с хорошим F# кодом являются основой каждой рекомендации. Они связаны с F# рекомендациями по проектированию компонентов, но применимы для F# любого кода, а не только для таких компонентов, как библиотеки.

Организация кода

F#два основных способа организации кода: модули и пространства имен. Они похожи, но имеют следующие отличия.

- Пространства имен компилируются как пространства имен .NET. Модули компилируются как статические классы.
- Пространства имен всегда имеют верхний уровень. Модули могут быть верхнего уровня и вложены в другие модули.
- Пространства имен могут охватывать несколько файлов. Модули не могут.
- Модули можно снабдить `[<RequireQualifiedAccess>]` и `[<AutoOpen>]`.

Следующие рекомендации помогут вам использовать их для организации кода.

Предпочитать пространства имен на верхнем уровне

Для любого общедоступного кода пространства имен являются предпочтительными для модулей на верхнем уровне. Поскольку они компилируются как пространства имен .NET, они могут быть потреблены от C# без каких-либо проблем.

```
// Good!
namespace MyCode

type MyClass() =
    ...
```

Использование модуля верхнего уровня может не отличаться при вызове только из F#, но для C# потребителей вызывающие объекты могут быть удивлены тем, что необходимо уточнить `MyClass` с помощью модуля `MyCode`.

```
// Bad!
module MyCode

type MyClass() =
    ...
```

Тщательное применение `[<AutoOpen>]`

Конструкция `[<AutoOpen>]` может засоряла область действия, которая доступна для вызывающих объектов, и ответ на то, что поступает от "волшебного". Как правило, это не хорошая вещь. Исключением из F# этого правила является сама основная библиотека (хотя этот факт также является битом спорной).

Однако это удобство, если у вас есть вспомогательная функциональность для общедоступного API, который вы хотите упорядочить отдельно от этого общедоступного API.

```
module MyAPI =
    [<AutoOpen>]
    module private Helpers =
        let helper1 x y z =
            ...

    let myFunction1 x =
        let y = ...
        let z = ...

        helper1 x y z
```

Это позволяет четко отделить сведения о реализации от открытого API функции, не требуя полного определения вспомогательного метода при каждом его вызове.

Кроме того, предоставление методов расширения и построителей выражений на уровне пространства имен можно аккуратно выразить с помощью `[<AutoOpen>]`.

Используйте `[<RequireQualifiedAccess>]` всякий раз, когда имена могут конфликтовать, или вы считаете, что они помогают удобочитаемости

Добавление атрибута `[<RequireQualifiedAccess>]` к модулю означает, что модуль не может быть открыт и что ссылки на элементы модуля должны явно иметь полный доступ. Например, модуль `Microsoft.FSharp.Collections.List` имеет этот атрибут.

Это полезно, когда функции и значения в модуле имеют имена, которые, скорее всего, конфликтуют с именами в других модулях. Обязательное получение полного доступа может значительно увеличить долгосрочное обслуживание и развитию библиотеки.

```
[<RequireQualifiedAccess>]
module StringTokenization =
    let parse s = ...

...

let s = getAString()
let parsed = StringTokenization.parse s // Must qualify to use 'parse'
```

Инструкции сортировки `open` топологически

В F#порядок объявлений имеет значение, включая инструкции `open`. Это не похоже C#на то, где воздействие `using` и `using static` не зависит от порядка этих инструкций в файле.

В F#элементы, открытые в области, могут уже быть скрыты другими. Это означает, что Переупорядочение инструкций `open` может изменить значение кода. В результате некоторая Любая произвольная сортировка всех инструкций `open` (например, буквенно-цифровых) обычно не рекомендуется, допустим вы создаете другое поведение, которое вы можете ожидать.

Вместо этого рекомендуется отсортировать их топологически; то есть закажите операторы `open` в том порядке, в котором определены *уровни* системы. Также можно учитывать алфавитно-цифровые сортировки в разных слоях топологическом.

Ниже приведен пример сортировки топологическом для файла общедоступного API F# -интерфейса службы компилятора:

```
namespace Microsoft.FSharp.Compiler.SourceCodeServices

open System
open System.Collections.Generic
open System.Collections.Concurrent
open System.Diagnostics
open System.IO
open System.Reflection
open System.Text

open Microsoft.FSharp.Compiler
open Microsoft.FSharp.Compiler.AbstractIL
open Microsoft.FSharp.Compiler.AbstractIL.Diagnostics
open Microsoft.FSharp.Compiler.AbstractIL.IL
open Microsoft.FSharp.Compiler.AbstractIL.ILBinaryReader
open Microsoft.FSharp.Compiler.AbstractIL.Internal
open Microsoft.FSharp.Compiler.AbstractIL.Internal.Library

open Microsoft.FSharp.Compiler.AccessibilityLogic
open Microsoft.FSharp.Compiler.Ast
open Microsoft.FSharp.Compiler.CompileOps
open Microsoft.FSharp.Compiler.CompileOptions
open Microsoft.FSharp.Compiler.Driver
open Microsoft.FSharp.Compiler.ErrorLogger
open Microsoft.FSharp.Compiler.Infos
open Microsoft.FSharp.Compiler.InfoReader
open Microsoft.FSharp.Compiler.Lexhelp
open Microsoft.FSharp.Compiler.Layout
open Microsoft.FSharp.Compiler.Lib
open Microsoft.FSharp.Compiler.NameResolution
open Microsoft.FSharp.Compiler.PrettyNaming
open Microsoft.FSharp.Compiler.Parser
open Microsoft.FSharp.Compiler.Range
open Microsoft.FSharp.Compiler.Tast
open Microsoft.FSharp.Compiler.Tastops
open Microsoft.FSharp.Compiler.TcGlobals
open Microsoft.FSharp.Compiler.TypeChecker
open Microsoft.FSharp.Compiler.SourceCodeServices.SymbolHelpers

open Internal.Utilities
open Internal.Utilities.Collections
```

Обратите внимание, что разрыв строки разделяет слои топологическом, при этом каждый слой сортируется в алфавитном порядке. Это четко организует код без случайного затенения значений.

Использование классов для хранения значений, имеющих побочные эффекты

Существует много случаев, когда инициализация значения может иметь побочные эффекты, такие как создание экземпляра контекста для базы данных или другого удаленного ресурса. Он позволяет инициализировать такие вещи в модуле и использовать его в последующих функциях:

```
// This is bad!
module MyApi =
    let dep1 = File.ReadAllText "/Users/{your name}/connectionstring.txt"
    let dep2 = Environment.GetEnvironmentVariable "DEP_2"

    let private r = Random()
    let dep3() = r.Next() // Problematic if multiple threads use this

    let function1 arg = doStuffWith dep1 dep2 dep3 arg
    let function2 arg = doSutffWith dep1 dep2 dep3 arg
```

Это неплохое идея по нескольким причинам:

Во-первых, Конфигурация приложения помещается в базу кода с помощью `dep1` и `dep2`. Это сложно поддерживать в больших базах кода.

Во вторых, статически инициализированные данные не должны содержать значения, которые не являются потокобезопасными, если компонент будет использовать несколько потоков. Это явно нарушается `dep3`.

Наконец, инициализация модуля компилируется в статический конструктор для всей единицы компиляции. Если какая-либо ошибка возникает при инициализации значения с привязкой let в этом модуле, она переносится в качестве `TypeInitializationException`, которая кэшируется в течение всего времени существования приложения. Это может быть трудно диагностировать. Обычно существует внутреннее исключение, которое можно попытаться определить, но в противном случае нет сведений о причине возникновения основной причины.

Вместо этого просто используйте простой класс для хранения зависимостей:

```
type MyParametricApi(dep1, dep2, dep3) =
    member _.Function1 arg1 = doStuffWith dep1 dep2 dep3 arg1
    member _.Function2 arg2 = doStuffWith dep1 dep2 dep3 arg2
```

Это позволяет выполнять следующие действия:

1. Отправка любого зависимого состояния вне самого API.
2. Теперь конфигурацию можно выполнять за пределами API.
3. Ошибки инициализации зависимых значений, скорее всего, не будут пере`TypeInitializationException`ся в манифест.
4. Теперь API проще тестировать.

Управление ошибками

Управление ошибками в больших системах является сложной и устойчивой задачей, и нет ни одного серебристого маркера, обеспечивающего отказоустойчивость и поведение систем. Приведенные ниже рекомендации должны содержать рекомендации по переходу на эту трудную область.

Представляет случаи ошибок и недопустимое состояние в типах, встроенных в ваш домен

С помощью размеченных объединений F# дает возможность представить состояние неисправной программы в системе типов. Пример:

```
type MoneyWithdrawalResult =
    | Success of amount:decimal
    | InsufficientFunds of balance:decimal
    | CardExpired of DateTime
    | UndisclosedFailure
```

В этом случае может произойти сбой из-за трех известных способов снятия денег с банковского счета. Каждый вариант ошибки представлен в типе и, таким образом, может быть достаточно безопасен во всей программе.

```
let handleWithdrawal amount =
    let w = withdrawMoney amount
    match w with
    | Success am -> printfn "Successfully withdrew %f" am
    | InsufficientFunds balance -> printfn "Failed: balance is %f" balance
    | CardExpired expiredDate -> printfn "Failed: card expired on %O" expiredDate
    | UndisclosedFailure -> printfn "Failed: unknown"
```

В общем случае, если вы можете моделировать различные способы **сбоя** в вашем домене, код обработки ошибок больше не обрабатывается в дополнение к обычной последовательности программ. Это просто часть обычного потока программы, не признанная **исключительной**. Существует два основных преимущества:

1. С течением времени мы проще поддерживать изменения в домене.
2. Случаи ошибок проще в модульном тесте.

Использовать исключения, если ошибки не могут быть представлены с помощью типов

Не все ошибки могут быть представлены в домене проблемы. Эти типы ошибок являются *исключительными* по своей природе, поэтому возможность вызывать и перехватывать исключения в F#.

Во-первых, рекомендуется ознакомиться с рекомендациями по проектированию исключений. Они также применимы к F#.

Основные конструкции, доступные в, F# в целях создания исключений следует учитывать в следующем порядке предпочтения:

ФУНКЦИЯ	СИНТАКСИС	ЦЕЛЬ
`nullArg`	`nullArg "argumentName"`	Вызывает `System.ArgumentNullException` с указанным именем аргумента.
`invalidArg`	`invalidArg "argumentName" "message"`	Вызывает `System.ArgumentException` с указанным именем аргумента и сообщением.
`invalidOp`	`invalidOp "message"`	Вызывает `System.InvalidOperationException` с указанным сообщением.
`raise`	`raise (ExceptionType("message"))`	Универсальный механизм для генерации исключений.
`failwith`	`failwith "message"`	Вызывает `System.Exception` с указанным сообщением.
`failwithf`	`failwithf "format string" argForFormatString`	Вызывает `System.Exception` с сообщением, определяемым строкой формата и его входными данными.

Используйте `nullArg`, `invalidArg` и `invalidOp` в качестве механизма вызова `ArgumentNullException`, `ArgumentException` и `InvalidOperationException`, если это уместно.

Обычно следует избегать функций `failwith` и `failwithf`, поскольку они создают базовый тип `Exception`, а не определенное исключение. В соответствии с рекомендациями по проектированию исключений, когда это возможно, необходимо вызвать более конкретные исключения.

Использование синтаксиса обработки исключений

F#поддерживает шаблоны исключений с помощью синтаксиса `try...with`:

```
try
    tryGetFileContents()
with
| :? System.IO.FileNotFoundException as e -> // Do something with it here
| :? System.Security.SecurityException as e -> // Do something with it here
```

Согласование функций, выполняемых в случае исключения с сопоставлением шаблонов, может быть немного сложным, если вы хотите оставить код нечетким. Одним из способов обработки этого является использование активных шаблонов в качестве средства для группирования функциональности, окружающей ошибку, с самим исключением. Например, вы можете использовать API, который, когда он создает исключение, заключает в метаданные исключения ценную информацию. Распаковка полезного значения в тексте перехваченного исключения в активном шаблоне и возвращение этого значения может оказаться полезной в некоторых ситуациях.

Не используйте собственную обработку ошибок для замены исключений

Исключения рассматриваются как несколько табу в функциональном программировании. Действительно, исключения нарушают чистоту, поэтому их можно спокойно считать недостаточной функциональностью. Однако это пропускает реальность, где должен выполняться код, и могут возникнуть ошибки времени выполнения. Как правило, написание кода предполагает, что большинство вещей не являются ни чистым, ни итоговым, чтобы максимально упростить неприятные сюрпризы.

Важно учитывать следующие основные сильные стороны и аспекты исключений в отношении их релевантности и адекватности в среде выполнения .NET и многоязыковой экосистемы в целом:

1. Они содержат подробные диагностические сведения, которые очень полезны при отладке проблемы.
2. Они хорошо понятны среде выполнения и другим языкам .NET.
3. Они могут уменьшить значительный шаблон при сравнении с кодом, который *не позволяет избежать* исключений, путем реализации некоторого подмножества их семантики на произвольной основе.

Этот третий момент является критически важным. Для нетривиальных сложных операций невозможность использования исключений может привести к работе с такими структурами:

```
Result<Result<MyType, string>, string list>
```

Что может легко привести к ненадежному коду, такому как сопоставление шаблонов при ошибках со строковыми типами:

```
let result = doStuff()
match result with
| Ok r -> ...
| Error e ->
    if e.Contains "Error string 1" then ...
    elif e.Contains "Error string 2" then ...
    else ... // Who knows?
```

Кроме того, может возникнуть желание проглотить любое исключение в случае, если для "простой" функции возвращается тип "лучше":

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with _ -> None
```

К сожалению, `tryReadAllText` может вызывать многочисленные исключения в зависимости от множества вещей, которые могут произойти в файловой системе, и этот код отклоняет любые сведения о том, что в вашей среде может быть неверно. Если заменить этот код типом результата, то будет выполнен синтаксический анализ сообщения об ошибке с вводом строкового типа:

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Ok
    with e -> Error e.Message

let r = tryReadAllText "path-to-file"
match r with
| Ok text -> ...
| Error e ->
    if e.Contains "uh oh, here we go again..." then ...
    else ...
```

И размещение объекта исключения в конструкторе `Error` просто заставляет вас правильно работать с типом исключения в месте вызова, а не в функции. Это позволяет эффективно создавать проверенные исключения, которые, в свою унфун, могут работать как вызывающий объект API.

Хорошим альтернативой приведенным выше примерам является перехват *конкретных* исключений и возврат осмысленного значения в контексте этого исключения. Если изменить функцию `tryReadAllText` следующим образом, `None` имеет большее значение:

```
let tryReadAllTextIfPresent (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with :? FileNotFoundException -> None
```

Вместо того, чтобы работать в качестве блока catch-all, эта функция теперь будет правильно обработана, если файл не найден и присвоить это значение возвращаемому значению. Это возвращаемое значение может сопоставляться с этим случаем ошибки, не удаляя никаких контекстных сведений и не требуя от вызывающих объектов обращения к ситуации, которая может не быть актуальной в этой точке кода.

Типы, такие как `Result<'Success, 'Error>`, подходят для основных операций, в которых они не F# являются вложенными, а необязательные типы идеально подходят для представления, когда что-либо может вернуть *что -либо* или *ничего*. Однако они не являются заменой для исключений и не должны использоваться при попытке заменить исключения. Вместо этого их следует применять внимательно, чтобы решить определенные аспекты политики исключений и управления ошибками.

Частичное программирование приложений и без точек

F#поддерживает частичное применение приложений и, таким образом, различные способы программирования в стиле "без точки". Это может быть полезно для повторного использования кода в модуле или реализации чего-либо, но обычно не является каким-либо общедоступным. Как правило, программирование без точки зрения не является приростом самого себя и может добавить значительный захватывающий барьер для людей, не вошедших в стиль.

Не используйте частичные приложения и карринг в общедоступных API

При небольшом исключении использование частичного приложения в общедоступных API может вызвать

путаницу для потребителей. Обычно значения, привязанные к `let` F# в коде, являются **значениями**, а не **значениями функций**. Сочетание значений и значений функций может привести к сохранению небольшого числа строк кода в Exchange для довольно большого количества системных издержек, особенно в сочетании с операторами, такими как `>>` для создания функций.

Примите во внимание особенности разработки средств для программирования без точки зрения

Каррированных функции не помечают свои аргументы. Это влияет на средства. Рассмотрим следующие две функции:

```
let func name age =
    printfn "My name is %s and I am %d years old!" name age

let funcWithApplication =
    printfn "My name is %s and I am %d years old!"
```

Оба являются допустимыми функциями, но `funcWithApplication` является каррированных функцией. При наведении указателя мыши на типы в редакторе вы увидите следующее:

```
val func : name:string -> age:int -> unit

val funcWithApplication : (string -> int -> unit)
```

В месте вызова всплывающие подсказки в инструментарии, такие как Visual Studio, не предоставляют осмысленной информации о том, как фактически представляются типы входных данных `string` и `int`.

Если вы столкнулись с кодом без точки, например `funcWithApplication`, который является общедоступным, рекомендуется выполнить полное расширение η, чтобы средства могли получать осмысленные имена для аргументов.

Кроме того, код без поддержки точек отладки может быть непростым, если это невозможно. Средства отладки зависят от значений, привязанных к именам (например, `let` привязок), чтобы можно было проверять промежуточные значения в середине по исполнению. Если код не имеет значений для проверки, отладка не требуется. В будущем средства отладки могут развиваться для синтезирования этих значений с учетом ранее выполненных путей, но не рекомендуется хеджирование свои элементы на *потенциальные* функции отладки.

Рассмотрите частичные приложения как методику сокращения внутреннего стандартного

В отличие от предыдущей точки, частичное применение — это замечательное средство для сокращения стандартного в приложении или более глубоких внутренних компонентов API. Он может быть полезен для модульного тестирования реализации более сложных интерфейсов API, где часто возникает проблема с шаблоном. Например, в следующем коде показано, как можно добиться того, что большинство инфраструктурных макетов позволит вам не брать внешнюю зависимость от такой платформы, а также изучать связанный API-интерфейс.

Например, рассмотрим следующее решение топографии:

```
MySolution.sln
|_/ImplementationLogic.fsproj
|_/ImplementationLogic.Tests.fsproj
|_/API.fsproj
```

`ImplementationLogic.fsproj` может представлять код, например:

```
module Transactions =
    let doTransaction txnContext txnType balance =
        ...

type Transactor(ctx, currentBalance) =
    member _.ExecuteTransaction(txnType) =
        Transactions.doTransaction ctx txtType currentBalance
        ...
```

Модульное тестирование `Transactions.doTransaction` в `ImplementationLogic.Tests.fsproj` — это просто:

```
namespace TransactionsTestingUtil

open Transactions

module TransactionsTestable =
    let getTestableTransactionRoutine mockContext = Transactions.doTransaction mockContext
```

Частичное применение `doTransaction` с объектом контекста макетирования позволяет вызывать функцию во всех модульных тестах, не требуя каждый раз создавать макет макета.

```
namespace TransactionTests

open Xunit
open TransactionTypes
open TransactionsTestingUtil
open TransactionsTestingUtil.TransactionsTestable

let testableContext =
    { new ITransactionContext with
        member _.TheFirstMember() = ...
        member _.TheSecondMember() = ... }

let transactionRoutine = getTestableTransactionRoutine testableContext

[<Fact>]
let ``Test withdrawal transaction with 0.0 for balance``() =
    let expected = ...
    let actual = transactionRoutine TransactionType.Withdraw 0.0
    Assert.Equal(expected, actual)
```

Этот метод не должен применяться к всей базе кода в универсальном коде, но это хороший способ сокращения стандартного для сложных внутренних компонентов и модульного тестирования этих внутренних компонентов.

Управление доступом

F#имеет несколько параметров для управления доступом, наследуемых от доступных в среде выполнения .NET. Они не просто могут использоваться для типов. их также можно использовать для функций.

- Предпочитать типы и члены, не относящиеся к `public`, до тех пор, пока они не понадобятся для общего использования. Это также сводится к уменьшению числа потребителей, с которыми связана пара.
- Оставайтесь в курсе всех вспомогательных функций `private`.
- Рекомендуется использовать `[<AutoOpen>]` в частном модуле вспомогательных функций, если они становятся многочисленными.

Определение типа и универсальные шаблоны

Определение типа может помочь вам в вводе большого числа шаблонов. И автоматическое обобщение в F# компиляторе может помочь в написании более универсального кода с практически без дополнительных усилий. Однако эти функции не являются универсальными.

- Рекомендуется помечать имена аргументов явными типами в общедоступных API и не полагаться на вывод типа для этого.

  Причина в том, что **вы** должны контролировать форму API, а не компилятор. Несмотря на то, что компилятор может выполнять хорошее задание в выведение типов, можно изменить форму API, если внутренние компоненты, от которых он зависит, изменили типы. Это может быть то, что вам нужно, но это почти наверняка приведет к прерыванию изменения API, с которым потом нижестоящим потребителям придется справиться. Вместо этого, если вы явно управляете формой открытого API, вы можете управлять этими критическими изменениями. В терминах DDD это можно рассматривать как уровень защиты от повреждений.

- Рекомендуется дать универсальным аргументам понятное имя.

  Если вы не пишете действительно универсальный код, который не относится к конкретному домену, понятное имя может помочь другим программистам понять, в каком домене они работают. Например, параметр типа с именем `'Document` в контексте взаимодействия с базой данных документов делает более ясно, что универсальные типы документов могут быть приняты функцией или членом, с которым вы работаете.

- Рассмотрите возможность именования параметров универсального типа с помощью PascalCase.

  Это общий способ выполнить действия в .NET, поэтому рекомендуется использовать PascalCase, а не snake_case или camelCase.

Наконец, автоматическое обобщение не всегда является boonом для тех, кто не является F# новым в или большой базе кода. При использовании универсальных компонентов существуют накладные расходы на переприятие. Более того, если автоматически обобщенные функции не используются с различными типами входных данных (то есть если они предназначены для использования), то в этот момент времени не существует реальных преимуществ. Всегда учитывайте, что код, который вы пишете, будет действительно выгодным.

Производительность

F#значения по умолчанию являются неизменяемыми, что позволяет избежать определенных классов ошибок (особенно связанных с параллелизмом и параллелизмом). Однако в некоторых случаях, чтобы достичь оптимальной (или даже разумной) эффективности времени выполнения или выделения памяти, объем работы можно реализовать с помощью изменения состояния на месте. Это можно сделать F# с помощью ключевого слова `mutable`.

Однако использование `mutable` в F# может быть, скорее всего, с функциональной чистотой. Это нормально, если вы настраиваете ожидания от чистоты до ссылочной прозрачности. Ссылочная прозрачность — не чистота — конечная цель при F# написании функций. Это позволяет написать функциональный интерфейс с реализацией на основе изменений для критичного в производительности кода.

Перенос изменяемого кода в неизменяемые интерфейсы

При использовании ссылочной прозрачности в качестве цели очень важно написать код, который не предоставляет изменяемую ненужные функции, критические для производительности. Например, следующий код реализует функцию `Array.contains` в F# основной библиотеке:

```
[<CompiledName("Contains")>]
let inline contains value (array:'T[]) =
    checkNonNull "array" array
    let mutable state = false
    let mutable i = 0
    while not state && i < array.Length do
        state <- value = array.[i]
        i <- i + 1
    state
```

Многократное обращение к этой функции не приводит к изменению базового массива, а также не требует поддержки какого либо изменяющегося состояния при его использовании. Эта функция прозрачна, даже если в ней почти каждая строка кода использует изменения.

Рассмотрите возможность инкапсуляции изменяемых данных в классы

В предыдущем примере использовалась одна функция для инкапсуляции операций с использованием изменяемых данных. Это не всегда достаточно для более сложных наборов данных. Рассмотрим следующие наборы функций:

```
open System.Collections.Generic

let addToClosureTable (key, value) (t: Dictionary<_,_>) =
    if not (t.ContainsKey(key)) then
        t.Add(key, value)
    else
        t.[key] <- value

let closureTableCount (t: Dictionary<_,_>) = t.Count

let closureTableContains (key, value) (t: Dictionary<_, HashSet<_>>) =
    match t.TryGetValue(key) with
    | (true, v) -> v.Equals(value)
    | (false, _) -> false
```

Этот код является выполняемым, но он предоставляет структуру данных на основе изменений, которые вызывающие объекты отвечают за обслуживание. Это можно обернуть внутри класса без базовых членов, которые могут измениться:

```
open System.Collections.Generic

/// The results of computing the LALR(1) closure of an LR(0) kernel
type Closure1Table() =
    let t = Dictionary<Item0, HashSet<TerminalIndex>>()

    member _.Add(key, value) =
        if not (t.ContainsKey(key)) then
            t.Add(key, value)
        else
            t.[key] <- value

    member _.Count = t.Count

    member _.Contains(key, value) =
        match t.TryGetValue(key) with
        | (true, v) -> v.Equals(value)
        | (false, _) -> false
```

`Closure1Table` инкапсулирует базовую структуру данных на основе изменений, тем самым не представляя вызывающие объекты поддерживать базовую структуру данных. Классы — это мощный способ инкапсуляции данных и подпрограмм, основанных на возможностях, без предоставления сведений

вызывающим объектам.

Предпочитать `let mutable` для ссылок на ячейки

Ссылочные ячейки — это способ представления ссылки на значение, а не само значение. Хотя они могут использоваться для критичного в производительности кода, они обычно не рекомендуются. Рассмотрим следующий пример.

```
let kernels =
    let acc = ref Set.empty

    processWorkList startKernels (fun kernel ->
        if not ((!acc).Contains(kernel)) then
            acc := (!acc).Add(kernel)
        ...)

    !acc |> Seq.toList
```

Использование ссылочной ячейки теперь «засоряет» все последующие коды с целью разыменования и повторной ссылки на базовые данные. Вместо этого рассмотрите `let mutable`:

```
let kernels =
    let mutable acc = Set.empty

    processWorkList startKernels (fun kernel ->
        if not (acc.Contains(kernel)) then
            acc <- acc.Add(kernel)
        ...)

    acc |> Seq.toList
```

Помимо единственной точки изменения в середине лямбда-выражения, весь остальной код, который касается `acc`, может сделать это так, что не отличается от использования обычного неизменяемого значения, привязанного к `let`. Это упростит изменение со временем.

Программирование объектов

F#обеспечивает полную поддержку объектов и объектно-ориентированных концепций (OO). Хотя многие основные понятия OO являются мощными и полезными, не все из них идеально подходят для использования. В следующих списках приведены рекомендации по категориям функций OO на высоком уровне.

Рассмотрите возможность использования этих функций во многих ситуациях.

- Точечная нотация (`x.Length`)
- Члены экземпляра
- Неявные конструкторы
- Статические члены
- Нотация индексатора (`arr.[x]`)
- Именованные и необязательные аргументы
- Реализации интерфейсов и интерфейсов

Не обращайтесь к этим функциям первыми, но применяйте их разумным образом, когда они удобны для решения проблемы:

- Перегрузка методов
- Инкапсулированные изменяемые данные

- Операторы в типах
- Автоматические свойства
- Реализация `IDisposable` и `IEnumerable`
- Расширения типов
- события
- структурам;
- Делегаты
- перечислениям;

Как правило, Избегайте этих функций, если их не нужно использовать:

- Иерархии типов на основе наследования и наследование реализации
- Значения NULL и `Unchecked.defaultof<_>`

Предпочитать композицию наследования

Композиция с наследованием — это долгий идиом, который F# может придерживаться хорошего кода. Основной принцип заключается в том, что не следует предоставлять базовый класс и заставить вызывающие объекты наследовать от этого базового класса для получения функциональности.

Использование выражений объектов для реализации интерфейсов, если не требуется класс

Выражения объектов позволяют реализовать интерфейсы на лету, привязывая реализованный интерфейс к значению без необходимости делать это внутри класса. Это удобно, особенно если требуется *только* реализовать интерфейс и не требуется полный класс.

Например, ниже приведен код, который выполняется в Ionide для предоставления действия по исправлению кода, если вы добавили символ, для которого у вас нет оператора `open`:

```
    let private createProvider () =
        { new CodeActionProvider with
            member this.provideCodeActions(doc, range, context, ct) =
                let diagnostics = context.diagnostics
                let diagnostic = diagnostics |> Seq.tryFind (fun d -> d.message.Contains "Unused open
statement")
                let res =
                    match diagnostic with
                    | None -> [||]
                    | Some d ->
                        let line = doc.lineAt d.range.start.line
                        let cmd = createEmpty<Command>
                        cmd.title <- "Remove unused open"
                        cmd.command <- "fsharp.unusedOpenFix"
                        cmd.arguments <- Some ([| doc |> unbox; line.range |> unbox; |] |> ResizeArray)
                        [|cmd |]
                res
                |> ResizeArray
                |> U2.Case1
        }
```

Поскольку при взаимодействии с Visual Studio Code API не требуется класс, выражения объектов являются идеальным средством для этого. Они также полезны при модульном тестировании, когда необходимо создать заглушку интерфейса с помощью подпрограмм тестирования нерегламентированным способом.

Рекомендуется использовать сокращения типов для сокращения сигнатур

Аббревиатуры типов — это удобный способ назначения метки другому типу, например сигнатуре функции или более сложному типу. Например, следующий псевдоним назначает метку, необходимую для

определения вычислений с помощью CNTK, библиотеки глубокого обучения:

```
open CNTK

// DeviceDescriptor, Variable, and Function all come from CNTK
type Computation = DeviceDescriptor -> Variable -> Function
```

Имя `Computation` — это удобный способ обозначить любую функцию, совпадающую с сигнатурой, которая является псевдонимом. Использование сокращенных обоподобие типов удобно и позволяет использовать более сжатый код.

Избегайте использования сокращений типов для представления домена

Хотя аббревиатуры типов удобно использовать для присвоения имени сигнатурам функций, они могут вызывать путаницу, если аббревиатинг другие типы. Рассмотрим следующее сокращение:

```
// Does not actually abstract integers.
type BufferSize = int
```

Это может быть затруднено несколькими способами:

- `BufferSize` не является абстракцией; Это просто другое имя для целого числа.
- Если `BufferSize` предоставляется в общедоступном API, он легко интерпретируется как неправильное, что означает не только `int`. Как правило, типы домена имеют несколько атрибутов и не являются примитивными типами, такими как `int`. Это сокращение нарушает это допущение.
- Регистр `BufferSize` (PascalCase) подразумевает, что этот тип содержит больше данных.
- Этот псевдоним не обеспечивает повышенную четкость по сравнению с предоставлением именованного аргумента функции.
- Аббревиатура не будет сокомпилироваться в скомпилированном IL; Это всего лишь целое число, а этот псевдоним является конструкцией времени компиляции.

```
module Networking =
    ...
    let send data (bufferSize: int) = ...
```

В целом, ловушки с сокращениями типов заключаются в том, что они **не** являются абстракциями для типов, которые они аббревиатинг. В предыдущем примере `BufferSize` — это просто `int`, который не содержит дополнительных данных, а также любые преимущества системы типов, Кроме того, что `int` уже есть.

Альтернативный подход к использованию сокращений типов для представления домена — Использование размеченных объединений с одним вариантом. Предыдущий пример можно смоделировать следующим образом:

```
type BufferSize = BufferSize of int
```

При написании кода, который работает в терминах `BufferSize` и его базовом значении, необходимо создать один из них вместо передачи любого произвольного целого числа:

```
module Networking =
    ...
    let send data (BufferSize size) =
    ...
```

Это уменьшает вероятность ошибочного переподачи произвольного целого числа в функцию `send`, так как вызывающий объект должен создать `BufferSize` тип для заключения значения перед вызовом функции.

F#рекомендации по проектированию компонентов

23.10.2019 • 46 minutes to read • Edit Online

Этот документ представляет собой набор рекомендации по проектированию компонентов для F# программирования, на основе F# рекомендации по проектированию компонентов, 14, перестанут, Microsoft Research и другая версия изначально проверенный и обслуживается F# Software Foundation.

В этом документе предполагается, что вы знакомы с F# программирования. Многие выражаем благодарность F# сообщества за их вклад и полезные отзывы на различные версии данного руководства.

Обзор

В этом документе рассматриваются некоторые проблемы, связанные с F# компонент проектирования и кодирования. Компонент может означать одно из следующих:

- Слой в вашей F# проект, имеющий внешних потребителей в рамках проекта.
- Это библиотека, предназначенная для использования F# кода за пределами сборки.
- Библиотека, предназначенный для использования с любым языком программирования .NET за пределами сборки.
- Библиотеки, предназначенные для распространения с помощью репозитория пакетов, таких как NuGet.

Методы, описанные в этой статье выполните хорошее пять принципов F# кодаи таким образом использовать функций и объектов, программирование соответствующим образом.

Независимо от того, методику конструктор компонента и библиотека сталкивается ряд практических и прозаическом проблем при попытке создать API, который проще всего использовать разработчиками. Conscientious применения рекомендации по разработке библиотек .NET будет управлять процессом к созданию согласованный набор API, которые являются приятной для использования.

Общие рекомендации

Существует несколько универсальных рекомендаций, которые применяются к F# библиотек, независимо от того, целевая аудитория для библиотеки.

Дополнительные рекомендации по разработке библиотек .NET

Независимо от типа объекта F# кодирования вы выполняете, важно иметь опыт работы с рекомендации по разработке библиотек .NET. Большинство других F# программистов .NET будет ознакомиться с рекомендациями, и ожидать, что код .NET в соответствии с их.

Рекомендации по разработке библиотек .NET предоставляют общие рекомендации по именованию, конструирование классов и интерфейсов, член конструктора (свойства, методы, события, и т.д.) и сведения и полезные первый точку ссылки для разнообразных руководство по проектированию.

Добавьте в код комментарии XML-документации

XML-документации в открытых API убедитесь, что пользователи могут получить великолепное Intellisense и кратких сведений, с помощью этих типов и членов и включить создание документации файлов для библиотеки. См. в разделе XML-документации о различных XML-теги, которые могут использоваться для дополнительной разметки внутри комментариев xmldoc.

```
/// A class for representing (x,y) coordinates
type Point =

    /// Computes the distance between this point and another
    member DistanceTo: otherPoint:Point -> float
```

С помощью комментариев XML краткая форма (`/// comment`), или стандартные XML-комментариев (`///<summary>comment</summary>`).

Рассмотрите возможность использования файлов явные сигнатур (расширение FSI) для стабильной библиотеки и API-интерфейсы компонента

С помощью явных подписей файлов F# библиотека предоставляет краткий сводку открытый API, они также располагаются позволяет гарантировать, что вы знаете полный открытый поверхности библиотеки, а также обеспечивает четкое разделение между общедоступной документации по и внутренние сведения о реализации. Обратите внимание на то, что файлы сигнатур добавить трения изменение открытого API-интерфейса, при необходимости вносить изменения в файлах и реализации и подпись. Таким образом файлы подписей должны обычно только быть предоставлены при API становятся упрочило и больше не может быть значительно изменить.

Всегда следуйте рекомендациям по использованию строк в .NET

Выполните советы и рекомендации по использованию строк в .NET рекомендации. В частности, всегда явно указывать *региональные намерение* в преобразование и сравнение строк (если применимо).

Рекомендации по F#-с выходом библиотеки

В этом разделе представлены рекомендации по разработке общедоступных F#-с выходом библиотеки; то есть библиотеки, предоставляя открытый API-интерфейсы, которые должны использоваться F# разработчиков. Существуют различные рекомендации по проектированию библиотеки, применимо специально к F#. В случае отсутствия определенные рекомендации, выполните рекомендации по разработке библиотек .NET являются резервной рекомендации.

Соглашения об именах

Используйте соглашения об именовании и регистр букв .NET

Соглашения об именовании и регистр букв .NET учтена в следующей таблице. Существует небольшой дополнения для включения F# конструкции.

КОНСТРУКЦИЯ	CASE	ОТДЕЛЕНИЕ	ПРИМЕРЫ	ПРИМЕЧАНИЯ
Конкретные типы	PascalCase	Существительное / прилагательными	Список, Double, сложный	Конкретные типы структур, классов, перечислений, делегаты, записей и объединений. Хотя имена типов, традиционно нижний регистр в OCaml, F# введена новая схема именования .NET для типов.
библиотеки DLL	PascalCase		Fabrikam.Core.dll	

КОНСТРУКЦИЯ	CASE	ОТДЕЛЕНИЕ	ПРИМЕРЫ	ПРИМЕЧАНИЯ
Объединения тегов	PascalCase	Существительное	Некоторые из них, добавить, успех	Не используйте префикс общедоступных интерфейсов API. При необходимости используйте префикс при внутренней, такие как `type Teams = TAlpha | TBeta | TDelta.`
событие	PascalCase	Команда	ValueChanged / ValueChanging	
Исключения	PascalCase		WebException	Имя должно заканчиваться на «Exception».
Поле	PascalCase	Существительное	CurrentName	
Типы интерфейса	PascalCase	Существительное / прилагательными	IDisposable	Имя должно начинаться с «I».
Метод	PascalCase	Команда	ToString	
Пространство имен	PascalCase		Microsoft.FSharp.Core	Обычно используется `<Organization>.<Technology>[.<Subnamespace>]`, но удалить организации, если эта технология не зависит от организации.
Параметры	camelCase	Существительное	Имя типа, преобразование, диапазон	
Разрешить значения (внутренний)	camelCase или PascalCase	Существительное-глагол	getValue myTable	

КОНСТРУКЦИЯ	CASE	ОТДЕЛЕНИЕ	ПРИМЕРЫ	ПРИМЕЧАНИЯ
Разрешить значения (внешний)	camelCase или PascalCase	Существительное глагол	List.map Dates.Today	привязки let значения часто являются открытыми, при использовании традиционных функциональные шаблоны разработки. Тем не менее обычно используется PascalCase, когда идентификатор может использоваться в других языках .NET.
Свойство	PascalCase	Существительное / прилагательными	IsEndOfFile, BackColor	Логические свойства обычно использование является и могут и должны быть выразил, как и в IsEndOfFile, не IsNotEndOfFile.

Избегать сокращений

Рекомендации по .NET не рекомендует использовать сокращения (например, «использовать `OnButtonClick` вместо `OnBtnClick` «). Сокращений, таких как `Async` для «Асинхронной», которая допустима. Иногда это правило игнорируется для функционального программирования; например `List.iter` использует сокращение для «итерация». По этой причине, использующие сокращения стремится скорректировать в большей степени в F#- к -F# программирования, но по-прежнему обычно следует избегать в открытого компонента конструктора.

Избежать конфликтов имен регистр

Рекомендации по .NET сказать, что регистр отдельно не может использоваться для однозначного определения конфликтов имен, так как некоторые языки клиента (например, Visual Basic), без учета регистра.

Используйте акронимов, если это уместно

Акронимы, таких как XML, не являются сокращения и широко используются в библиотеках .NET в параметр формы (Xml). Только хорошо известного, широко признанный акронимов.

Использующие PascalCase для имен универсальных параметров

Использующие PascalCase для универсального параметра имен общедоступных интерфейсов API, в том числе для F#-с выходом библиотеки. В частности, используйте имена, такие как `T` , `U` , `T1` , `T2` для произвольных универсальных параметров, а также при определенными именами имеет смысл, затем F#-имена, как использовать библиотеки с выходом `Key` , `Value` , `Arg` (но не к примеру, `TKey`).

Использовать для открытых функций и значения в PascalCase или camelCase F# модулей

camelCase используется для открытых функций, которые предназначены для использования неполное (например, `invalidArg`) и для «функции стандартной коллекции» (например, List.map). В обоих случаях имена функций действуют во многом аналогично ключевых слов на языке.

Объект, тип и модуль разработки

Используйте пространства имен или модули для размещения типов и модули

Каждый F# в компоненте должен начинаться с объявлением пространства имен или объявлению модуля.

```
namespace Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
     ...

module CommonOperations =
    ...
```

или

```
module Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
    ...

module CommonOperations =
    ...
```

Далее приведены различия между использованием модулей и пространств имен для организации кода верхнего уровня.

- Пространства имен могут охватывать несколько файлов
- Пространства имен не может содержать F# функции, если они находятся в внутренним
- Код для любого заданного модуля должны содержаться в одном файле
- Модули верхнего уровня может содержать F# функции без необходимости внутренним

Выбор между пространством имен верхнего уровня или модуль влияет на форме скомпилированного кода и таким образом повлияет на представлении из других языков .NET следует API со временем использовать за пределами F# кода.

Использовать методы и свойства для операций, характерными для типов объектов

При работе с объектами, нужно убедиться, что готовых к использованию функциональность реализуется как методы и свойства этого типа.

```
type HardwareDevice() =

    member this.ID = ...

    member this.SupportedProtocols = ...

type HashTable<'Key,'Value>(comparer: IEqualityComparer<'Key>) =

    member this.Add(key, value) = ...

    member this.ContainsKey(key) = ...

    member this.ContainsValue(value) = ...
```

Не, основная часть функций для данного элемента должны обязательно будут реализованы в этот элемент, но должно быть готовых к использованию части эту функциональность.

Использование классов, инкапсулирующих изменяемое состояние

В F#, это действие необходимо выполнить где что состояние не уже инкапсулируется другой языковой

конструкции, такие как замыкание, выражения последовательности или асинхронное вычисление.

```
type Counter() =
    // let-bound values are private in classes.
    let mutable count = 0

    member this.Next() =
        count <- count + 1
        count
```

Используйте интерфейсы для группирования операций, связанных с

Используйте типы интерфейсов для представления набора операций. Это предпочтительнее другие параметры, например кортежей, функций или записей функций.

```
type Serializer =
    abstract Serialize<'T>: preserveRefEq: bool -> value: 'T -> string
    abstract Deserialize<'T>: preserveRefEq: bool -> pickle: string -> 'T
```

В ссылке к:

```
type Serializer<'T> = {
    Serialize: bool -> 'T -> string
    Deserialize: bool -> string -> 'T
}
```

Интерфейсы являются понятиями первостепенными в .NET, который можно использовать для достижения, что Функторы обычно отображаются. Кроме того они могут использоваться для кодирования типов уже существует в вашу программу, которой нет записей функций.

Использование модуля группы функций, которые действуют в коллекциях

При определении типа коллекции, подберите стандартный набор операций, например `CollectionType.map` и `CollectionType.iter`) для новых типов коллекций.

```
module CollectionType =
    let map f c =
        ...
    let iter f c =
        ...
```

При включении таких модулей, следуйте стандартным соглашениям об именовании для функций, содержащихся в FSharp.Core.

Использование модуля группы функций для общих канонических функций, особенно в math и библиотеки DSL

Например `Microsoft.FSharp.Core.Operators` — автоматического открытия коллекция функции верхнего уровня (например `abs` и `sin`), предоставляемые FSharp.Core.dll.

Аналогичным образом, библиотека статистики может включать модуль с функциями `erf` и `erfc`, где этот модуль предназначен должны быть открыты автоматически.

Рассмотрите возможность использования RequireQualifiedAccess и тщательно применять атрибуты AutoOpen

Добавление `[<RequireQualifiedAccess>]` атрибут к модулю указывает, что модуль не может быть открыт, и необходимость явной ссылки на элементы модуля полного доступа. Например `Microsoft.FSharp.Collections.List` модуля с этим атрибутом.

Это полезно в том случае, если функции и значения в модуле имеют имена, которые могут конфликтовать с именами в других модулях. Требуется метод доступа может значительно увеличивать долгосрочной

поддержки и развиваемости библиотеки.

Добавление `[<AutoOpen>]` атрибут к модулю означает, что модуль будет открываться при открытии содержащего пространства имен. `[<AutoOpen>]` Атрибут также может быть применен к сборке, чтобы указать модуль, который автоматически открывается при ссылке на сборку.

Например, библиотеку статистики **MathsHeaven.Statistics** может содержать `module MathsHeaven.Statistics.Operators` функции `erf` и `erfc`. Следует отметить этот модуль как `[<AutoOpen>]`. Это означает, что `open MathsHeaven.Statistics` будет также открыть этот модуль и вывести имена `erf` и `erfc` в область действия. Использование другой good `[<AutoOpen>]` для модулей, содержащих методы расширения.

Излишнее применение параметров `[<AutoOpen>]` приводит к его засорения пространств имен, а атрибуту следует использовать с осторожностью. Для определенных библиотек в определенных доменах, разумно использовать `[<AutoOpen>]` может привести к более удобной.

Рекомендуется определять оператор членов в классах, в которых подходит использование хорошо известного операторов

Иногда классы используются для моделирования математической конструкции, такие как векторы. Если домен моделируемой содержит хорошо известных операторов, определяя их в качестве членов, встроенные в класс полезен.

```
type Vector(x: float) =

    member v.X = x

    static member (*) (vector: Vector, scalar: float) = Vector(vector.X * scalar)

    static member (+) (vector1: Vector, vector2: Vector) = Vector(vector1.X + vector2.X)

let v = Vector(5.0)

let u = v * 10.0
```

В этом руководстве соответствует общих рекомендаций .NET для этих типов. Тем не менее, может быть Кроме того, важно в F# кодирования, так как он разрешает этих типов для использования в сочетании с F# функций и методов с ограничениями элементов, таких как List.sumBy.

Рассмотрите возможность использования CompiledName для предоставления. NET-понятное имя для других потребителей язык .NET

Иногда вы можете назвать в один стиль для F# потребителей (таких как статический член в нижнем регистре, чтобы он располагался как будто связанного с модулем функции), но имеют другой стиль для имени, при компиляции в сборку. Можно использовать `[<CompiledName>]` атрибут для предоставления свой стиль не F# кода, использующего сборку.

```
type Vector(x:float, y:float) =

    member v.X = x
    member v.Y = y

    [<CompiledName("Create")>]
    static member create x y = Vector (x, y)

let v = Vector.create 5.0 3.0
```

С помощью `[<CompiledName>]`, можно использовать соглашения об именовании .NET для не F# пользователей сборки.

Используйте перегрузки метода для функций-членов, если такой подход обеспечивает простой API

Перегрузка метода является мощным средством для упрощения API, который может потребоваться

выполнить аналогичные функциональные возможности, но разные параметры и аргументы.

```
type Logger() =

    member this.Log(message) =
        ...
    member this.Log(message, retryPolicy) =
        ...
```

В F#, чаще всего для перегрузки на число аргументов, а не типы аргументов.

Скрыть представления типы объединений и записи, если разработка этих типов — вероятно эволюционирование

Избегайте раскрытия конкретные представления объектов. Например, конкретные представление DateTime значения не раскрываются внешних, открытый в API для разработки библиотеки .NET. Во время выполнения среда CLR знает зафиксированных реализации, который будет использоваться на протяжении выполнения. Тем не менее скомпилированный код не сам получают зависимостей от конкретного представления.

Избегайте использования реализации наследования для расширяемости

В F#, наследование реализации используется редко. Кроме того часто иерархий наследования являются сложными и трудно изменить, когда поступают новые требования. Реализация наследования по-прежнему находится в F# для совместимости и редких случаях, когда он является лучшим решением проблемы альтернативных методик, необходимо обратиться в вашей F# программ при разработке для полиморфизма, например интерфейс Реализация.

Функция и элемент подписи

Использование кортежей для возвращаемых значений, при возврате небольшое количество несколько несвязанных значений

Вот хороший пример использования в возвращаемом типе кортежа:

```
val divrem: BigInteger -> BigInteger -> BigInteger * BigInteger
```

Для возврата типов, содержащих множество компонентов или там, где компоненты связаны с одной идентифицировать сущности, рассмотрите возможность использования именованного типа вместо кортежа.

Используйте `Async<T>` для асинхронного программирования в F# границе API

Если имеется соответствующий синхронную операцию с именем `Operation` , возвращающий `T` , то асинхронная операция должна называться `AsyncOperation` при его использовании возвращается `Async<T>` или `OperationAsync` при его использовании возвращается `Task<T>` . Для часто используемых типов .NET, которые предоставляют методов Begin и End, рассмотрите возможность использования `Async.FromBeginEnd` записываемый методы расширения в качестве оболочки для предоставления F# асинхронную модель программирования для этих API-интерфейсы .NET.

```
type SomeType =
    member this.Compute(x:int): int =
        ...
    member this.AsyncCompute(x:int): Async<int> =
        ...

type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        ...
```

Исключения

См. в разделе управление обработкой ошибок Дополнительные сведения о соответствующее

использование исключений, результаты и параметры.

Члены расширений

Тщательно применить F# члены расширений в F#- к -F# компоненты

F#члены расширений следует обычно использоваться только для операций, которые находятся в закрытии внутренних операций, связанных с типом в большинстве режимов, его использования. Один обычно используется для предоставления API, которые являются более устойчивым к F# для различных типов .NET:

```
type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        Async.FromBeginEnd(this.BeginReceive, this.EndReceive)

type System.Collections.Generic.IDictionary<'Key,'Value> with
    member this.TryGet key =
        let ok, v = this.TryGetValue key
        if ok then Some v else None
```

Типы объединений

Использование размеченных объединений вместо иерархий классов для древовидной данных

Древовидной структуры являются рекурсивного определения. Это неудобно с наследованием, но элегантного с размеченные объединения.

```
type BST<'T> =
    | Empty
    | Node of 'T * BST<'T> * BST<'T>
```

Представления данных дерева с размеченные объединения также позволяет использовать преимущества exhaustiveness при сопоставлении шаблонов.

Используйте `[<RequireQualifiedAccess>]` на типы объединения вариантов, имена которых не являются достаточно уникальными

Вы обнаружите себя в домене, где имя рекомендации для выполнения различных задач, таких как случаи размеченного объединения тем же именем. Можно использовать `[<RequireQualifiedAccess>]` для однозначного определения вариантов имена во избежание активации заблуждение ошибок из-за затенение зависит от порядка `open` инструкций

Скрыть представления размеченные объединения для двоичных совместимых API, если разработка этих типов — вероятно эволюционирование

Типы объединений зависят от F# поиска совпадения с шаблоном формы, для краткости модель программирования. Как упоминалось ранее, следует избегать раскрытия представления конкретных данных, если разработка этих типов — вероятно эволюционирование.

Например, представление размеченное объединение можно скрыть с помощью объявление закрытого или внутреннего или с помощью подписи файла.

```
type Union =
    private
    | CaseA of int
    | CaseB of string
```

Если вы раскрываете информацию размеченные объединения беспорядочно, может оказаться трудно версии библиотеки без нарушения кода пользователя. Вместо этого рассмотрите возможность раскрытия один или несколько активных шаблонов, позволяющая сопоставления над значениями типа шаблона.

Активные шаблоны предоставляют альтернативный способ предоставления F# потребителей с избегая предоставления сопоставление шаблонов F# непосредственно типы объединений.

Встроенные функции и ограничения члена

Определите универсальный числовые алгоритмы, с помощью встроенных функций с ограничениями подразумеваемых членов и статически разрешаемым универсальные типы

Ограничения арифметических члена и F# ограничения на сравнение являются стандартом для F# программирования. Рассмотрим следующий пример кода:

```
let inline highestCommonFactor a b =
    let rec loop a b =
        if a = LanguagePrimitives.GenericZero<_> then b
        elif a < b then loop a (b - a)
        else loop (a - b) b
    loop a b
```

Тип этой функции выглядит следующим образом:

```
val inline highestCommonFactor : ^T -> ^T -> ^T
                when ^T : (static member Zero : ^T)
                and ^T : (static member ( - ) : ^T * ^T -> ^T)
                and ^T : equality
                and ^T : comparison
```

Это подходящая функция для общего API в математической библиотеки.

Старайтесь не использовать член ограничения для имитации классов типа и типизации по признакам

Можно имитировать с помощью «типизации по признакам» F# ограничениями членов. Однако члены, которые делают с помощью этого следует использовать не в общие в F#- к -F# библиотеки макетов. Это обусловлено тем, как правило, вызвать пользовательский код стать неудобными и привязаны к одной конкретной платформой шаблон библиотеки макетов на основе незнакомых или нестандартные неявное ограничений.

Кроме того есть шанс, что активно использовать ограничения член таким образом может привести компиляции очень много времени.

Оператор определения

Не рекомендуется определять пользовательские операторы, символьные

Пользовательские операторы необходимы в некоторых случаях и очень удобен в обозначениям устройства в больших частях кода реализации. Для новых пользователей библиотеки именованных функций зачастую проще в использовании. Кроме того пользовательские символьные операторы могут возникнуть трудности с документов и пользователям будет сложно более поиск справки по операторам, из-за ограничений существующих обработчиков интегрированной среды разработки и поиска.

Таким образом рекомендуется для публикации в виде именованных функций и члены и Кроме того, только в том случае, если значимых преимуществ перевешивают документация и cognitive затраты на их, предоставлять операторы для использования этой функции.

Единицы измерения

Внимательно использовать единицы измерения для добавленных строгой типизации в F# кода

Дополнительные сведения о типизации для единицы измерения удаляется при его просмотре на других языках .NET. Имейте в виду, что компоненты .NET и средства отражения будут видеть типы sans единиц. Например, C# пользователи увидят `float` вместо `float<kg>`.

Сокращенные обозначения типов

Внимательно использовать сокращенные формы типов для упрощения F# кода

Компоненты .NET и средства отражения не увидит сокращенные имена для типов. Значительное использование сокращенные формы типов также можно сделать домена отображаются более сложным, чем он фактически является, которой удалось вводят в заблуждение потребителей.

Избегайте сокращенные формы типов для открытых типов, членов и свойств должны отличаться по своей природе для тех, которые доступны для типа сокращается

В этом случае тип сокращается показывает, что слишком многое о представлении фактический тип определяемого. Вместо этого рассмотрите возможность упаковки аббревиатура в тип класса или одиночным размеченного объединения (или, если важна производительность, рассмотрите возможность использования типа структуры программы-оболочки для сокращения).

Например, это подмывает вас определить карту несколькими как особый вариант F# сопоставления, например:

```
type MultiMap<'Key,'Value> = Map<'Key,'Value list>
```

Тем не менее операции логического нотацию для этого типа не так же, как операции на карте — например, разумно, оператор поиска были сопоставлены. [key] Возврат пустой список, если ключа нет в словаре, а не вызывает исключение.

Рекомендации для библиотек для использования из других языков .NET

При разработке библиотек для использования в других языках .NET, очень важно придерживаться рекомендации по разработке библиотек .NET. Эти библиотеки в этом документе, обозначаемый vanilla библиотеки .NET, в отличие от F#-с выходом библиотеки, использующие F# конструкции без ограничений. Проектирование vanilla библиотек .NET означает, предоставляя знакомые и устойчивым API согласования с остальной частью платформы .NET Framework, сводя к минимуму использование F#-конкретных конструкций в открытом API. В следующих разделах описаны правила.

Пространство имен и тип конструктора (для библиотек для использования из других языков .NET)

Применить соглашения об именовании .NET для открытого API-интерфейса компонентов

Обратите внимание на использование сокращенные имена и регистр букв рекомендациям .NET.

```
type pCoord = ...
    member this.theta = ...

type PolarCoordinate = ...
    member this.Theta = ...
```

Используйте пространства имен, типы и члены как основной организационной структурой для компонентов

Все файлы, содержащие открытые методы должны начинаться с `namespace` объявления и только общедоступные сущности в пространствах имен, которые должны быть типы. Не используйте F# модулей.

Используйте модули не являющиеся открытыми для размещения код реализации, служебные типы и служебных функций.

Статические типы лучше использовать модули, как они позволяют использовать перегрузки и другие принципы проектирования .NET API, которые не могут использоваться в будущих Эволюция API F# модулей.

Например, вместо следующих открытого API-интерфейса:

```
module Fabrikam

module Utilities =
    let Name = "Bob"
    let Add2 x y = x + y
    let Add3 x y z = x + y + z
```

Попробуйте вместо этого:

```
namespace Fabrikam

[<AbstractClass; Sealed>]
type Utilities =
    static member Name = "Bob"
    static member Add(x,y) = x + y
    static member Add(x,y,z) = x + y + z
```

Используйте F# типах записей в обычный API-интерфейсы .NET, если не будет развиваться разработки типов

F#типы записей скомпилирована в простой класс .NET. Они подходят для некоторых типов простых и стабильных в API-интерфейсы. Следует рассмотреть возможность использования `[<NoEquality>]` и `[<NoComparison>]` атрибуты, чтобы отключить автоматическое создание интерфейсов. Также Избегайте поля изменяемых записей в обычный API-интерфейсы .NET как эти предоставляет открытое поле. Всегда учитывайте, будет ли класс предоставляет более гибкий вариант для будущего развития API-интерфейса.

Например, следующая F# код предоставляет открытый API для C# потребителей:

F#:

```
[<NoEquality; NoComparison>]
type MyRecord =
    { FirstThing: int
        SecondThing: string }
```

C#:

```
public sealed class MyRecord
{
    public MyRecord(int firstThing, string secondThing);
    public int FirstThing { get; }
    public string SecondThing { get; }
}
```

Скрыть представление F# типам объединений в обычный API-интерфейсы .NET

F#типы объединений не распространены через границы компонентов, даже для F#- к -F# кодирования. Это устройство отличную реализации при для внутреннего использования в компоненты и библиотеки.

При разработке обычный .NET API, можно скрыть представление типа объединения, используя объявление закрытого или файл сигнатур.

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True
```

Кроме того, может быть расширено типов, которые внутренне используют представление union с членами для обеспечения требуемой. NET API-интерфейсы.

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True

    /// A public member for use from C#
    member x.Evaluate =
        match x with
        | And(a,b) -> a.Evaluate && b.Evaluate
        | Not a -> not a.Evaluate
        | True -> true

    /// A public member for use from C#
    static member CreateAnd(a,b) = And(a,b)
```

Проектирование графического пользовательского интерфейса и другими компонентами с помощью шаблонов разработки платформы

Существует множество различных платформ в .NET, например WinForms, WPF и ASP.NET. Соглашения об именовании и проектирования для каждого можно использовать при разработке компонентов, используемых в этих платформ. Например для программирования WPF, внедряйте WPF шаблоны разработки для классов, которые вы разрабатываете. Для моделей в программировании пользовательского интерфейса, используйте шаблоны проектирования, таких как события и коллекций на основе уведомлений, например из статьи System.Collections.ObjectModel.

Объекты и элементы проектирования (для библиотек для использования из других языков .NET)

Используется для предоставления событий .NET CLIEvent-атрибут

Создать `DelegateEvent` с определенной .NET делегат типа, который принимает объект и `EventArgs` (вместо `Event`, которой пользуется только `FSharpHandler` тип по умолчанию) таким образом, чтобы события, публикуются в знакомый способ других языков .NET.

```
type MyBadType() =
    let myEv = new Event<int>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish

type MyEventArgs(x: int) =
    inherit System.EventArgs()
    member this.X = x

    /// A type in a component designed for use from other .NET languages
type MyGoodType() =
    let myEv = new DelegateEvent<EventHandler<MyEventArgs>>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish
```

Предоставлять асинхронные операции, как методы, которые возвращают задачи .NET

Задачи используются в .NET для представления active асинхронных вычислений. Задачи, в целом, менее композиционной, чем F# `Async<T>` объектов, так как они представляют задачи «уже выполняется» и не может состоять вместе одним из способов, которые выполняют параллельные композиции или который скрыть распространение сигналы отмены и другие контекстные параметры.

Однако несмотря на это, методы, которые возвращают задачи, стандартное представление асинхронного программирования на платформе .NET.

```
/// A type in a component designed for use from other .NET languages
type MyType() =

    let compute (x: int): Async<int> = async { ... }

    member this.ComputeAsync(x) = compute x |> Async.StartAsTask
```

Вы будете часто Обраб явного токена отмены:

```
/// A type in a component designed for use from other .NET languages
type MyType() =
    let compute(x: int): Async<int> = async { ... }
    member this.ComputeAsTask(x, cancellationToken) = Async.StartAsTask(compute x, cancellationToken)
```

Используйте типы делегата .NET вместо F# типы функций

Здесь "F# типы функций» означает «стрелка» типов, таких как `int -> int`.

Вместо этого:

```
member this.Transform(f: int->int) =
    ...
```

Выполните следующее.

```
member this.Transform(f: Func<int,int>) =
    ...
```

F# Тип функции отображается как `class FSharpFunc<T,U>` на других языках .NET и этот процесс менее подходит для функций языка и инструменты, которые понимают типы делегатов. При создании метода более высокого порядка, предназначенных для .NET Framework 3.5 или более поздней версии, `System.Func` и `System.Action` делегаты являются правой API-интерфейсы для публикации позволяет разработчикам .NET использовать эти API-интерфейсы в виде низким коэффициентом трения. (При нацеливании на .NET Framework 2.0, типы делегатов, определяемые системой более ограничены; рассмотрите возможность использования делегата предопределенные типы, такие как `System.Converter<T,U>` или определение определенному типу делегата.)

С другой стороны, делегаты .NET не являются естественным F#-с выходом библиотек (в следующем разделе на F#-с выходом библиотек). Таким образом, общая стратегия реализации при разработке методы более высокого порядка для соответствующего ванили библиотек .NET предусматривает создание все реализации с помощью F# типы функций, а затем создать открытый интерфейс API с использованием делегатов в качестве тонкой оболочки поверх фактический F#реализации.

Использование шаблона TryGetValue вместо возвращения F# значения параметров, а предпочитаете перегрузка методов для ведения F# параметр значения в качестве аргументов

Общие шаблоны использования F# типа параметра в API-интерфейсы работают лучше реализации в ваниль методы разработки API-интерфейсы .NET, с помощью standard .NET. Вместо возвращения F# значение параметра, рассмотрите возможность использования возвращаемый тип bool, а также как в шаблоне «TryGetValue» выходной параметр. И вместо использования F# значения как параметры, то рассмотрите возможность использования перегрузка или необязательные аргументы метода.

```
member this.ReturnOption() = Some 3

member this.ReturnBoolAndOut(outVal: byref<int>) =
    outVal <- 3
    true

member this.ParamOption(x: int, y: int option) =
    match y with
    | Some y2 -> x + y2
    | None -> x

member this.ParamOverload(x: int) = x

member this.ParamOverload(x: int, y: int) = x + y
```

Использовать интерфейс коллекции .NET типы IEnumerable<T> и IDictionary<ключ, значение> для параметров и возвращаемых значений

Не рекомендуется использовать конкретный набор типов, таких как массивы .NET `T[]`, F# типы `list<T>`, `Map<Key,Value>` и `Set<T>`, и .NET конкретные типы коллекций, такие как `Dictionary<Key,Value>`. Рекомендации по разработке библиотек .NET имеют хороший совет, касающийся необходимости использования различных типов коллекций, такие как `IEnumerable<T>`. Некоторые использование массивов (`T[]`) допустима в некоторых случаях на территории производительности. Обратите внимание, особенно что `seq<T>` является просто F# псевдоним для `IEnumerable<T>`, и поэтому seq часто соответствующих типов для обычный .NET API.

Вместо F# перечислены:

```
member this.PrintNames(names: string list) =
    ...
```

Используйте F# последовательности:

```
member this.PrintNames(names: seq<string>) =
    ...
```

Используйте тип единицы измерения как единственный входной тип метода для определения метода нулевой аргумент, или как единственный возвращаемый тип для определения метода, возвращающего значение void

Избегайте других вариантов использования тип единицы измерения. Это хороший:

```
✔ member this.NoArguments() = 3

✔ member this.ReturnVoid(x: int) = ()
```

Это неправильный:

```
member this.WrongUnit( x: unit, z: int) = ((), ())
```

Проверять наличие значений null по границам стандартный API для .NET

F#код реализации обычно имеют меньшее количество значений null, из-за неизменяемый конструктивные шаблоны и ограничения использования литералы null для F# типов. Другими языками .NET часто используют значение null гораздо чаще. По этой причине F# кода, которая предоставляет обычный .NET API Проверьте параметры со значением NULL на границе API и предотвратить эти значения из потока более подробная F# код реализации. `isNull` Функции или шаблоны на `null` шаблон может использоваться.

```
let checkNonNull argName (arg: obj) =
    match arg with
    | null -> nullArg argName
    | _ -> ()

let checkNonNull` argName (arg: obj) =
    if isNull arg then nullArg argName
    else ()
```

Старайтесь не использовать кортежи как возвращаемые значения

Вместо этого предпочитают, возвращая именованный тип, содержащий статистические данные, или использование выходных параметров для возврата нескольких значений. Несмотря на то, что кортежи и кортежи, структура существует в .NET (в том числе поддержка языка C# для структурных кортежей), они чаще всего не будет предоставлять идеальная и ожидаемый интерфейс API для разработчиков .NET.

Избегайте использования каррирование параметров

Вместо этого использовать соглашения о вызовах .NET `Method(arg1,arg2,…,argN)`.

```
member this.TupledArguments(str, num) = String.replicate num str
```

Совет. Если вы разрабатываете библиотек для использования в любом языке .NET, то есть не является заменой делать некоторые экспериментальные C# и Visual Basic, чтобы убедиться, что библиотеки «вид справа» из этих языков программирования. Средства, такие как .NET Reflector и обозревателя объектов Visual Studio также можно гарантией библиотек и связанную документацию для разработчиков должным образом.

Приложение

End-to-end примером проектирование F# код для использования с другими языками .NET

Рассмотрим следующий класс:

```
open System

type Point1(angle,radius) =
    new() = Point1(angle=0.0, radius=0.0)
    member x.Angle = angle
    member x.Radius = radius
    member x.Stretch(l) = Point1(angle=x.Angle, radius=x.Radius * l)
    member x.Warp(f) = Point1(angle=f(x.Angle), radius=x.Radius)
    static member Circle(n) =
        [ for i in 1..n -> Point1(angle=2.0*Math.PI/float(n), radius=1.0) ]
```

Выводимые F# типа этого класса выглядит следующим образом:

```
type Point1 =
    new : unit -> Point1
    new : angle:double * radius:double -> Point1
    static member Circle : n:int -> Point1 list
    member Stretch : l:double -> Point1
    member Warp : f:(double -> double) -> Point1
    member Angle : double
    member Radius : double
```

Рассмотрим, как это F# тип появится программисту, используя другой язык .NET. Например приблизительное C# «подпись» выглядит следующим образом:

```
// C# signature for the unadjusted Point1 class
public class Point1
{
    public Point1();

    public Point1(double angle, double radius);

    public static Microsoft.FSharp.Collections.List<Point1> Circle(int count);

    public Point1 Stretch(double factor);

    public Point1 Warp(Microsoft.FSharp.Core.FastFunc<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}
```

Существуют некоторые важные особенности о том, как F# представляет конструкции здесь. Пример:

- Метаданные, такие как имена аргументов был сохранен.
- F#методы, которые принимают два аргумента становятся C# методы, которые принимают два аргумента.
- Функции и списки становятся ссылки на соответствующие типы в F# библиотеки.

Ниже показано, как настроить этот код необходимо учитывать следующее.

```
namespace SuperDuperFSharpLibrary.Types

type RadialPoint(angle:double, radius:double) =

    /// Return a point at the origin
    new() = RadialPoint(angle=0.0, radius=0.0)

    /// The angle to the point, from the x-axis
    member x.Angle = angle

    /// The distance to the point, from the origin
    member x.Radius = radius

    /// Return a new point, with radius multiplied by the given factor
    member x.Stretch(factor) =
        RadialPoint(angle=angle, radius=radius * factor)

    /// Return a new point, with angle transformed by the function
    member x.Warp(transform:Func<_,_>) =
        RadialPoint(angle=transform.Invoke angle, radius=radius)

    /// Return a sequence of points describing an approximate circle using
    /// the given count of points
    static member Circle(count) =
        seq { for i in 1..count ->
                RadialPoint(angle=2.0*Math.PI/float(count), radius=1.0) }
```

Выводимые F# тип кода выглядит следующим образом:

```
type RadialPoint =
    new : unit -> RadialPoint
    new : angle:double * radius:double -> RadialPoint
    static member Circle : count:int -> seq<RadialPoint>
    member Stretch : factor:double -> RadialPoint
    member Warp : transform:System.Func<double,double> -> RadialPoint
    member Angle : double
    member Radius : double
```

Сигнатура C# — теперь следующим образом:

```
public class RadialPoint
{
    public RadialPoint();

    public RadialPoint(double angle, double radius);

    public static System.Collections.Generic.IEnumerable<RadialPoint> Circle(int count);

    public RadialPoint Stretch(double factor);

    public RadialPoint Warp(System.Func<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}
```

Внесены исправления для подготовки к использованию этого типа, как часть стандартного библиотеки .NET являются следующим образом:

- Изменить несколько имен: `Point1`, `n`, `l`, и `f` стали `RadialPoint`, `count`, `factor`, и `transform`, соответственно.
- Используемый тип возвращаемого значения `seq<RadialPoint>` вместо `RadialPoint list` путем изменения конструкции списка с помощью `[ ... ]` для создания последовательности с помощью `IEnumerable<RadialPoint>`.
- Используемый тип делегата .NET `System.Func` вместо F# тип функции.

Это делает гораздо лучше, для использования в коде C#.

Использование языка F# в Azure

15.01.2020 • 9 minutes to read • Edit Online

F# — это великолепный язык для облачного программирования, который часто используется для создания веб-приложений, облачных служб, размещаемых в облаке микрослужб, а также для масштабируемой обработки данных.

В следующих разделах приведены ресурсы по использованию различных служб Azure с F#.

> **NOTE**
> Если в этом наборе документации не упомянута определенная служба Azure, обратитесь к документации по Функциям Azure или платформе .NET для этой службы. Некоторые службы Azure не упоминаются здесь, так как не зависят от языка и не требуют документацию для конкретного языка.

Использование виртуальных машин Azure с F#

Azure поддерживает широкий спектр конфигураций виртуальных машин (ВМ), см. статью Linux и виртуальные машины Azure.

Чтобы установить F# на виртуальной машине для выполнения, компиляции кода и/или написания скриптов, см. статьи Использование F# в Linux и Использование F# в Windows.

Использование функций Azure с F#

Функции Azure — это решение для легкого выполнения небольших фрагментов кода — или "функций" — в облаке. Вы можете ограничиться написанием кода, достаточного для решения поставленной задачи, не беспокоясь о полноценном приложении или инфраструктуре для его запуска. Ваши функции подключены к событиям в службе хранилища Azure и другим размещенным в облаке ресурсам. Данные передаются в функции F# через аргументы функции. Вы можете использовать любой привычный язык разработки, переложив задачи по масштабированию на Azure.

Функции Azure поддерживают F# в качестве первоклассного языка с эффективным, реактивным и масштабируемым выполнением кода F#. В справочнике разработчика по F# для Функций Azure приведены ссылки на документацию по использованию языка F# с Функциями Azure.

Другие ресурсы, посвященные использованию Функций Azure и F#.

- Масштабирование Функций Azure в F# с помощью Suave
- Создание функции Azure в F#
- Использование поставщика типов Azure с функциями Azure

Использование службы хранилища Azure с F#

Служба хранилища Azure является базовым уровнем служб хранения для современных приложений, которым необходима устойчивость, доступность и масштабируемость для удовлетворения потребностей пользователей. F#программы могут напрямую взаимодействовать со службами хранилища Azure, используя методы, описанные в следующих статьях.

- Начало работы с хранилищем BLOB-объектов Azure с помощью языка F#
- Начало работы с хранилищем файлов Azure с помощью языка F#

- Начало работы с хранилищем очередей Azure с помощью языка F#
- Начало работы с хранилищем таблиц Azure с помощью языка F#

Службу хранилища Azure также можно использовать совместно с Функциями Azure посредством декларативной конфигурации вместо явных вызовов API. В статье Триггеры и привязки Функций Azure для службы хранилища Azure приведены примеры на F#.

Использование службы приложений Azure с F#

Служба приложений Azure — это облачная платформа для создания эффективных мобильных и веб-приложений, подключающихся к данным где угодно — в облаке или локальной среде.

- Пример веб-API Azure на F#
- Размещение F# в веб-приложении на платформе Azure

Использование Apache Spark и F# с Azure HDInsight

Apache Spark для Azure HDInsight — это платформа обработки с открытым исходным кодом, в которой выполняются крупномасштабные приложения для анализа данных. Azure делает развертывание Apache Spark простым и экономичным. Разработайте свое приложение Spark в F#, используя Mobius — API платформы .NET для Spark.

- Реализация приложений Spark в F# с помощью Mobius
- Пример приложений Spark на F# с использованием Mobius

Использование Azure Cosmos DB с F#

Azure Cosmos DB — это служба NoSQL для высокодоступных, глобально распределенных приложений.

Azure Cosmos DB можно использовать с F# двумя способами:

1. С помощью создания функций F# Azure, реагирующих на или вызывающих изменения в Azure Cosmos DB коллекциях. См. раздел привязки Azure Cosmos DB для функций Azure.
2. С помощью пакета SDK для Azure Cosmos DB .NET для SQL API. Связанные образцы находятся в C#.

Использование концентраторов событий Azure с F#

Центры событий Azure обеспечивают прием телеметрии с веб-сайтов, устройств и из приложений в масштабах облака.

Центры событий Azure можно использовать с F# двумя способами.

1. Посредством создания Функций Azure на F#, активируемых событиями. См. статью Триггеры Функций Azure для центров событий.
2. Посредством пакета SDK .NET для Azure. Обратите внимание, что эти примеры написаны на C#.

Использование концентраторов уведомлений Azure с F#

Концентраторы уведомлений Azure — это многоплатформенная и масштабируемая инфраструктура, позволяющая отправлять мобильные push-уведомления из любой серверной системы (в облаке или локальной среде) для всех мобильных платформ.

Концентраторы уведомлений Azure можно использовать с F# двумя способами.

1. Посредством создания Функций Azure на F#, которые отправляют результаты в концентратор уведомлений. См. статью Триггеры вывода Функций Azure для концентраторов уведомлений.

2. Посредством пакета SDK .NET для Azure. Обратите внимание, что эти примеры написаны на C#.

Реализация веб-перехватчиков в Azure с помощью F#

Объект webhook представляет собой обратный вызов, активируемый через веб-запрос. Объекты webhook используются сайтами, например GitHub, для сигнализации о событиях.

Объекты webhook можно реализовать в F# и разместить в Azure с помощью функции Azure в F# с привязкой webhook.

Использование веб-заданий с F#

Веб-задания — это программы, которые можно выполнять в веб-приложении службы приложений тремя способами: по запросу, непрерывно или по расписанию.

Пример веб-задания на F#

Реализация таймеров в Azure с помощью F#

Таймер активирует функции вызова на основе расписания — один раз или регулярно.

Таймеры можно реализовать в F# и разместить в Azure с помощью функции Azure в F# с триггером таймера.

Развертывание ресурсов Azure и управление ими с помощью скриптов F#

Виртуальные машины Azure можно программно развертывать и контролировать из скриптов F# с помощью API и пакетов Microsoft.Azure.Management. Например, см. статьи Приступая к работе с библиотеками управления для .NET и Использование Azure Resource Manager.

Аналогичным образом из скриптов F# можно развернуть и контролировать и другие ресурсы Azure. Например, вы можете создавать учетные записи хранения, развертывать облачные службы Azure, создавать Azure Cosmos DB экземпляры и управлять центрами F# уведомлений Azure программными средствами из скриптов.

Использовать скрипты F# для развертывания ресурсов и управления ими в общем случае необязательно. Например, ресурсы Azure также могут быть развернуты непосредственно из описаний шаблонов JSON, которые могут быть параметризованы. В статье Шаблоны Azure Resource Manager приведены примеры, например шаблоны быстрого запуска Azure.

Другие ресурсы

- Полная документация по всем службам Azure

Приступая к работе с хранилищем BLOB-объектов Azure с помощью F#

15.01.2020 • 20 minutes to read • Edit Online

Хранилище BLOB-объектов Azure — это служба, которая сохраняет неструктурированные данные в облаке в виде BLOB-объектов. В хранилище BLOB-объектов можно хранить любые типы текстовых или двоичных данных, таких как документы, файлы мультимедиа или установщики приложений. Хранилище BLOB-объектов также называют хранилищем объектов.

В этой статье показано, как выполнять общие задачи с помощью хранилища BLOB-объектов. Примеры написаны с F# помощью клиентской библиотеки службы хранилища Azure для .NET. В рассмотренных задачах содержатся сведения о передаче, перечислении, скачивании и удалении больших двоичных объектов.

Общие сведения о хранилище BLOB-объектов см. в этомразделе.

Prerequisites

Для работы с этим руководством необходимо сначала создать учетную запись хранения Azure. Вам также потребуется ключ доступа к хранилищу для этой учетной записи.

Создание F# скрипта и запуск F# интерактивного

Примеры в этой статье можно использовать как в F# приложении, так и в F# сценарии. Чтобы создать F# скрипт, создайте файл с расширением `.fsx`, например `blobs.fsx`, в среде F# разработки.

Затем используйте Диспетчер пакетов , например пакет или NuGet , для установки пакетов `WindowsAzure.Storage` и `Microsoft.WindowsAzure.ConfigurationManager`, а также ссылки на `WindowsAzure.Storage.dll` и `Microsoft.WindowsAzure.Configuration.dll` в скрипте с помощью директивы `#r`.

Добавление объявлений пространств имен

Добавьте в начало файла `blobs.fsx` следующие инструкции `open`:

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Blob // Namespace for Blob storage types
```

Получение строки подключения

Для работы с этим руководством требуется строка подключения к службе хранилища Azure. Дополнительные сведения о строках подключения см. в разделе Настройка строк подключения к хранилищу.

В этом руководстве вы вводите строку подключения в скрипте следующим образом:

```
let storageConnString = "..." // fill this in from your storage account
```

Однако это **не рекомендуется** для реальных проектов. Ключ учетной записи хранения похож на корневой пароль для вашей учетной записи хранения. Не забудьте защитить ключ учетной записи хранения. Не сообщайте его другим пользователям, не определяйте его в коде и не храните его в текстовом файле,

доступном другим пользователям. Вы можете повторно создать ключ с помощью портала Azure, если считаете, что он был скомпрометирован.

Для реальных приложений лучшим способом поддержания строки подключения к хранилищу является файл конфигурации. Чтобы получить строку подключения из файла конфигурации, можно сделать следующее:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

Использование диспетчера конфигураций Azure не является обязательным. Можно также использовать API, например тип `ConfigurationManager` .NET Framework.

Проанализируйте строку подключения

Чтобы проанализировать строку подключения, используйте:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

Это возвращает `CloudStorageAccount`.

Создание локальных фиктивных данных

Прежде чем начать, создайте в каталоге нашего скрипта некоторые фиктивные локальные данные. Позже вы отправите эти данные.

```
// Create a dummy file to upload
let localFile = __SOURCE_DIRECTORY__ + "/myfile.txt"
File.WriteAllText(localFile, "some data")
```

Создание клиента службы BLOB-объектов

Тип `CloudBlobClient` позволяет извлекать контейнеры и большие двоичные объекты, хранящиеся в хранилище BLOB-объектов. Вот один из способов создать клиента службы.

```
let blobClient = storageAccount.CreateCloudBlobClient()
```

Теперь вы можете написать код, который считывает и записывает данные в хранилище BLOB-объектов.

Создание контейнера

В этом примере показано, как создать контейнер:

```
 // Retrieve a reference to a container.
let container = blobClient.GetContainerReference("mydata")

// Create the container if it doesn't already exist.
container.CreateIfNotExists()
```

По умолчанию новый контейнер является закрытым. Это значит, что вам нужно указать ключ доступа к хранилищу, чтобы загрузить большие двоичные объекты из этого контейнера. Чтобы сделать файлы в этом контейнере доступными для всех пользователей, сделайте контейнер открытым, используя следующий код:

```
let permissions = BlobContainerPermissions(PublicAccess=BlobContainerPublicAccessType.Blob)
container.SetPermissions(permissions)
```

Любой пользователь в Интернете может видеть большие двоичные объекты в открытом контейнере, но изменить или удалить их можно только при наличии ключа доступа или подписанного URL-адреса.

Отправка BLOB-объекта в контейнер

Хранилище BLOB-объектов Azure поддерживает блочные и страничные BLOB-объекты. В большинстве случаев рекомендуемым типом для использования является Блочный BLOB-объект.

Для передачи файла в блочный BLOB-объект получите ссылку на контейнер и используйте ее для получения ссылки на блочный BLOB-объект. После получения ссылки на большой двоичный объект можно передать в него любой поток данных, вызвав метод `UploadFromFile`. Эта операция создает большой двоичный объект, если он не существовал ранее, или перезаписывает его, если он существует.

```
// Retrieve reference to a blob named "myblob.txt".
let blockBlob = container.GetBlockBlobReference("myblob.txt")

// Create or overwrite the "myblob.txt" blob with contents from the local file.
do blockBlob.UploadFromFile(localFile)
```

Перечисление BLOB-объектов в контейнере

Для перечисления BLOB-объектов в контейнере сначала необходимо получить ссылку на контейнер. Затем можно использовать метод `ListBlobs` контейнера для извлечения больших двоичных объектов и (или) каталогов внутри него. Чтобы получить доступ к обширному набору свойств и методов для возвращаемого `IListBlobItem`, необходимо привести его к `CloudBlockBlob`, `CloudPageBlob` или объекту `CloudBlobDirectory`. Если тип неизвестен, можно использовать проверку типов, чтобы определить нужный тип. Следующий код демонстрирует, как получить и вывести URI каждого элемента в контейнере `mydata`:

```
// Loop over items within the container and output the length and URI.
for item in container.ListBlobs(null, false) do
    match item with
    | :? CloudBlockBlob as blob ->
        printfn "Block blob of length %d: %O" blob.Properties.Length blob.Uri

    | :? CloudPageBlob as pageBlob ->
        printfn "Page blob of length %d: %O" pageBlob.Properties.Length pageBlob.Uri

    | :? CloudBlobDirectory as directory ->
        printfn "Directory: %O" directory.Uri

    | _ ->
        printfn "Unknown blob type: %O" (item.GetType())
```

Кроме того, в именах больших двоичных объектов можно присвоить сведения о пути. Таким образом, создается такая структура виртуальных каталогов, которую можно организовывать и просматривать, как и традиционную файловую систему. Обратите внимание, что используется только структура виртуальных каталогов, так как единственные ресурсы, доступные в хранилище больших двоичных объектов, — контейнеры и большие двоичные объекты. Однако клиентская библиотека хранилища предоставляет объект `CloudBlobDirectory` для ссылки на виртуальный каталог и упрощает процесс работы с большими двоичными объектами, организованными таким образом.

Например, рассмотрим следующий набор блочных BLOB-объектов в контейнере с именем `photos`:

photo1. jpg
\ 2015/архитектура/описание. txt

При вызове `ListBlobs` для контейнера (как в приведенном выше примере) возвращается иерархический список. Если он содержит объекты `CloudBlobDirectory` и `CloudBlockBlob`, представляющие каталоги и большие двоичные объекты в контейнере соответственно, результирующие выходные данные выглядят примерно так:

```
Directory: https://<accountname>.blob.core.windows.net/photos/2015/
Directory: https://<accountname>.blob.core.windows.net/photos/2016/
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

При необходимости можно задать для параметра `UseFlatBlobListing` метода `ListBlobs` значение `true`. В этом случае каждый большой двоичный объект в контейнере возвращается как объект `CloudBlockBlob`. Вызов `ListBlobs` для возврата плоского списка выглядит следующим образом:

```
// Loop over items within the container and output the length and URI.
for item in container.ListBlobs(null, true) do
    match item with
    | :? CloudBlockBlob as blob ->
        printfn "Block blob of length %d: %O" blob.Properties.Length blob.Uri

    | _ ->
        printfn "Unexpected blob type: %O" (item.GetType())
```

в зависимости от текущего содержимого контейнера результаты выглядят следующим образом:

```
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2015/architecture/description.txt
Block blob of length 314618: https://<accountname>.blob.core.windows.net/photos/2015/architecture/photo3.jpg
Block blob of length 522713: https://<accountname>.blob.core.windows.net/photos/2015/architecture/photo4.jpg
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2016/architecture/description.txt
Block blob of length 419048: https://<accountname>.blob.core.windows.net/photos/2016/architecture/photo5.jpg
Block blob of length 506388: https://<accountname>.blob.core.windows.net/photos/2016/architecture/photo6.jpg
Block blob of length 399751: https://<accountname>.blob.core.windows.net/photos/2016/photo7.jpg
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

Скачивание больших двоичных объектов

Чтобы скачать большие двоичные объекты, сначала извлеките ссылку на большой двоичный объект, а затем вызовите метод `DownloadToStream`. В следующем примере используется метод `DownloadToStream` для передачи содержимого большого двоичного объекта в объект потока, который затем можно сохранить в локальном файле.

```
// Retrieve reference to a blob named "myblob.txt".
let blobToDownload = container.GetBlockBlobReference("myblob.txt")

// Save blob contents to a file.
do
    use fileStream = File.OpenWrite(__SOURCE_DIRECTORY__ + "/path/download.txt")
    blobToDownload.DownloadToStream(fileStream)
```

Можно также использовать метод `DownloadToStream` для загрузки содержимого большого двоичного объекта в виде текстовой строки.

```
let text =
    use memoryStream = new MemoryStream()
    blobToDownload.DownloadToStream(memoryStream)
    Text.Encoding.UTF8.GetString(memoryStream.ToArray())
```

удаление больших двоичных объектов.

Чтобы удалить большой двоичный объект, сначала получите ссылку на большой двоичный объект, а затем вызовите для него метод `Delete`.

```
// Retrieve reference to a blob named "myblob.txt".
let blobToDelete = container.GetBlockBlobReference("myblob.txt")

// Delete the blob.
blobToDelete.Delete()
```

Асинхронное перечисление BLOB-объектов в страницах

Если вам нужно расположить большое количество BLOB-объектов или вы хотите управлять отображением количества объектов в результате запроса, вы можете задать расположение BLOB-объектов на странице. В этом примере вы узнаете, как расположить запрошенные результаты на странице асинхронно для того, чтобы не блокировать выполнение задачи ожиданием большого объема возвращаемых данных.

В этом примере показан плоский список больших двоичных объектов, но можно также выполнить иерархическое построение, установив для параметра `useFlatBlobListing` метода `ListBlobsSegmentedAsync` значение `false`.

В примере определяется асинхронный метод с помощью блока `async`. Ключевое слово `let!` приостанавливает выполнение метода Sample до завершения задачи листинга.

```
let ListBlobsSegmentedInFlatListing(container:CloudBlobContainer) =
    async {

        // List blobs to the console window, with paging.
        printfn "List blobs in pages:"

        // Call ListBlobsSegmentedAsync and enumerate the result segment
        // returned, while the continuation token is non-null.
        // When the continuation token is null, the last page has been
        // returned and execution can exit the loop.

        let rec loop continuationToken (i:int) =
            async {
                let! ct = Async.CancellationToken
                // This overload allows control of the page size. You can return
                // all remaining results by passing null for the maxResults
                // parameter, or by calling a different overload.
                let! resultSegment =
                    container.ListBlobsSegmentedAsync(
                        "", true, BlobListingDetails.All, Nullable 10,
                        continuationToken, null, null, ct)
                    |> Async.AwaitTask

                if (resultSegment.Results |> Seq.length > 0) then
                    printfn "Page %d:" i

                for blobItem in resultSegment.Results do
                    printfn "\t%O" blobItem.StorageUri.PrimaryUri

                printfn ""

                // Get the continuation token.
                let continuationToken = resultSegment.ContinuationToken
                if (continuationToken <> null) then
                    do! loop continuationToken (i+1)
            }

        do! loop null 1
    }
```

Теперь мы можем использовать эту асинхронную подпрограммы, как показано ниже. Сначала необходимо отправить некоторые фиктивные данные (с помощью локального файла, созданного ранее в этом руководстве).

```
// Create some dummy data by uploading the same file over and over again
for i in 1 .. 100 do
    let blob  = container.GetBlockBlobReference("myblob" + string i + ".txt")
    use fileStream = System.IO.File.OpenRead(localFile)
    blob.UploadFromFile(localFile)
```

Теперь вызовите подпрограммы. Для принудительного выполнения асинхронной операции используется `Async.RunSynchronously`.

```
ListBlobsSegmentedInFlatListing container |> Async.RunSynchronously
```

Запись в расширенный большой двоичный объект

Добавочный большой двоичный объект оптимизирован для операций добавления, например ведения журналов. Как и блочный BLOB-объект, добавочный большой двоичный объект состоит из блоков, но при добавлении нового блока в добавочный большой двоичный объект он всегда добавляется в конец этого

объекта. Вы не можете обновить или удалить существующий блок в добавочном большом двоичном объекте. Идентификаторы блоков в добавочном большом двоичном объекте не отображаются, как в блочном BLOB-объекте.

Каждый блок в добавочном большом двоичном объекте может иметь разный размер (не более 4 МБ), кроме того, добавочный большой двоичный объект может содержать не более 50 000 блоков. Таким образом, максимальный размер добавочного большого двоичного объекта немного превышает 195 ГБ (4 МБ X 50 000 блоков).

В следующем примере создается новый добавочный BLOB-объект и к нему добавляются некоторые данные, имитируя простую операцию ведения журнала.

```
// Get a reference to a container.
let appendContainer = blobClient.GetContainerReference("my-append-blobs")

// Create the container if it does not already exist.
appendContainer.CreateIfNotExists() |> ignore

// Get a reference to an append blob.
let appendBlob = appendContainer.GetAppendBlobReference("append-blob.log")

// Create the append blob. Note that if the blob already exists, the
// CreateOrReplace() method will overwrite it. You can check whether the
// blob exists to avoid overwriting it by using CloudAppendBlob.Exists().
appendBlob.CreateOrReplace()

let numBlocks = 10

// Generate an array of random bytes.
let rnd = new Random()
let bytes = Array.zeroCreate<byte>(numBlocks)
rnd.NextBytes(bytes)

// Simulate a logging operation by writing text data and byte data to the
// end of the append blob.
for i in 0 .. numBlocks - 1 do
    let msg = sprintf "Timestamp: %u \tLog Entry: %d\n" DateTime.UtcNow bytes.[i]
    appendBlob.AppendText(msg)

// Read the append blob to the console window.
let downloadedText = appendBlob.DownloadText()
printfn "%s" downloadedText
```

Дополнительные сведения о различиях между тремя типами больших двоичных объектов см. в статье Основные сведения о блочных, страничных и добавочных BLOB-объектах.

Одновременный доступ

Чтобы реализовать одновременный доступ нескольких клиентов или экземпляров процесса к BLOB-объекту, можно использовать **ETags** или **lease**.

- **Etag** — позволяет обнаружить, что BLOB-объект или контейнер были изменены другим процессом
- **Lease** — аренда позволяет получить монопольный обновляемый доступ к BLOB-объекту на запись или удаление в течение заданного периода времени.

Дополнительные сведения см. в разделе Управление параллелизмом в Служба хранилища Microsoft Azure.

Именование контейнеров

Каждый BLOB-объект в Azure должен располагаться в контейнере. Контейнер составляет часть имени BLOB-

объекта. Например, `mydata` — имя контейнера в таких взятых в качестве образца URI BLOB-объектов:

- https://storagesample.blob.core.windows.net/mydata/blob1.txt
- https://storagesample.blob.core.windows.net/mydata/photos/myphoto.jpg

Имя контейнера должно быть допустимым DNS-именем и соответствовать указанным ниже правилам именования.

1. Имена контейнеров должны начинаться с буквы или цифры и могут содержать только буквы, цифры и тире (-).
2. Каждое тире (-) должно стоять непосредственно перед буквой или цифрой и после нее. В именах контейнеров запрещено использовать несколько тире подряд.
3. Все знаки в имени контейнера должны быть строчными.
4. Имя контейнера должно содержать от 3 до 63 знаков.

Обратите внимание, что имя контейнера должно содержать только строчные буквы. При использовании заглавных букв в имени контейнера или другом нарушении правил именования контейнера может появиться ошибка 400 (Ошибка запроса).

Управление системой безопасности больших двоичных объектов

По умолчанию служба хранилища Azure защищает данные, ограничивая доступ к учетной записи пользователя, который владеет ключами доступа к учетной записи. Если вы хотите предоставить доступ к данным больших двоичных объектов в своей учетной записи хранения, важно сделать это без ущерба для безопасности ключей доступа к учетной записи. Кроме того, вы можете зашифровать данные больших двоичных объектов, чтобы обеспечить их безопасную отправку по сети в службу хранилища Azure.

Управление доступом к данным больших двоичных объектов

По умолчанию данные больших двоичных объектов в учетной записи хранения доступны только владельцу учетной записи хранения. По умолчанию для проверки подлинности запросов к хранилищу BLOB-объектов требуется ключ доступа к учетной записи. Однако может потребоваться предоставить другим пользователям доступ к данным большого двоичного объекта.

Шифрование данных больших двоичных объектов

Служба хранилища Azure поддерживает шифрование данных большого двоичного объекта как на клиенте, так и на сервере.

Следующие шаги

Вы ознакомились с базовыми понятиями о хранилище BLOB-объектов. Дополнительные сведения см. по следующим ссылкам.

Средства .

- F# Азуресторажетипепровидер
  Поставщик F# типов, который можно использовать для просмотра ресурсов хранилища Azure BLOB-объектов, таблиц и очередей, а также для простого применения операций CRUD к ним.

- FSharp. Azure. Storage
  F# API для использования службы хранилища таблиц Microsoft Azure

- Обозреватель службы хранилища Microsoft Azure (MASE)
  Бесплатное автономное приложение от корпорации Майкрософт, позволяющее визуально работать с данными службы хранилища Azure в Windows, OS X и Linux.

Справочная документация по хранилищу BLOB-объектов

- API-интерфейсы хранилища Azure для .NET
- Azure Storage Services REST API Reference (Справочник по REST API службы хранилища Azure)

Связанные руководства

- начало работы с хранилищем BLOB-объектов Azure вC#
- Перенос данных с помощью служебной программы командной строки AzCopy в Windows
- Перенос данных с помощью служебной программы командной строки AzCopy в Linux
- Настройка строк подключения службы хранилища Azure
- Блог рабочей группы службы хранилища Azure
- Краткое руководство. Создание большого двоичного объекта в хранилище объектов с помощью .NET

Начало работы с хранилищем файлов Azure с помощью F#

15.01.2020 • 10 minutes to read • Edit Online

Хранилище файлов Azure — это служба, которая предлагает доступ к общим папкам в облаке с использованием стандартного протокола SMB. Поддерживаются версии SMB 2.1 и SMB 3.0. Хранилище файлов Azure позволяет быстро и без дорогостоящей перезаписи выполнить перенос приложений прежних версий, связанных с общими папками. Приложения, работающие на виртуальных машинах Azure, в облачных службах или на локальных клиентах, могут подключать общую папку в облаке так же, как настольное приложение подключает обычную общую папку SMB. Любое количество компонентов приложений может одновременно подключаться и получать доступ к ресурсам хранилища файлов.

Общие сведения о хранилище файлов см. в этомразделе.

Prerequisites

Для работы с этим руководством необходимо сначала создать учетную запись хранения Azure. Вам также потребуется ключ доступа к хранилищу для этой учетной записи.

Создание F# скрипта и запуск F# интерактивного

Примеры в этой статье можно использовать как в F# приложении, так и в F# сценарии. Чтобы создать F# скрипт, создайте файл с расширением `.fsx`, например `files.fsx`, в среде F# разработки.

Затем используйте Диспетчер пакетов , например пакет или NuGet , для установки пакета `WindowsAzure.Storage` и ссылки на `WindowsAzure.Storage.dll` в скрипте с помощью директивы `#r`.

Добавление объявлений пространств имен

Добавьте в начало файла `files.fsx` следующие инструкции `open`:

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.File // Namespace for File storage types
```

Получение строки подключения

Для работы с этим руководством вам потребуется строка подключения к службе хранилища Azure. Дополнительные сведения о строках подключения см. в разделе Настройка строк подключения к хранилищу.

В этом руководстве вы введете строку подключения в сценарий следующим образом:

```
let storageConnString = "..." // fill this in from your storage account
```

Однако это **не рекомендуется** для реальных проектов. Ключ учетной записи хранения похож на корневой пароль для вашей учетной записи хранения. Не забудьте защитить ключ учетной записи хранения. Не сообщайте его другим пользователям, не определяйте его в коде и не храните его в текстовом файле, доступном другим пользователям. Вы можете повторно создать ключ с помощью портала Azure, если считаете, что он был скомпрометирован.

Для реальных приложений лучшим способом поддержания строки подключения к хранилищу является файл конфигурации. Чтобы получить строку подключения из файла конфигурации, можно сделать следующее:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

Использование диспетчера конфигураций Azure не является обязательным. Можно также использовать API, например тип `ConfigurationManager` .NET Framework.

Проанализируйте строку подключения

Чтобы проанализировать строку подключения, используйте:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

Это приведет к возврату `CloudStorageAccount`.

Создание клиента службы файлов

Тип `CloudFileClient` позволяет программно использовать файлы, хранящиеся в хранилище файлов. Вот один из способов создать клиента службы.

```
let fileClient = storageAccount.CreateCloudFileClient()
```

Теперь все готово для написания кода, считывающего данные из хранилища файлов и записывающего их в хранилище.

Создание общей папки

В этом примере показано, как создать файловый ресурс, если он еще не существует:

```
let share = fileClient.GetShareReference("myfiles")
share.CreateIfNotExists()
```

Создание корневого каталога и подкаталога

Здесь вы получаете корневой каталог и подкаталогом корневого каталога. Они создаются, если они еще не существуют.

```
let rootDir = share.GetRootDirectoryReference()
let subDir = rootDir.GetDirectoryReference("myLogs")
subDir.CreateIfNotExists()
```

Отправка текста в виде файла

В этом примере показано, как передать текст в виде файла.

```
let file = subDir.GetFileReference("log.txt")
file.UploadText("This is the content of the log file")
```

Скачать файл в локальную копию файла

Здесь вы скачиваете только что созданный файл, добавляя содержимое в локальный файл.

```
file.DownloadToFile("log.txt", FileMode.Append)
```

Установка максимального размера для файлового ресурса

В приведенном ниже примере показано, как проверить текущее использование данных в файловом ресурсе, а также задать для него квоту. для заполнения `Properties` общего ресурса необходимо вызвать `FetchAttributes` и `SetProperties` распространить локальные изменения в хранилище файлов Azure.

```
// stats.Usage is current usage in GB
let stats = share.GetStats()
share.FetchAttributes()

// Set the quota to 10 GB plus current usage
share.Properties.Quota <- stats.Usage + 10 |> Nullable
share.SetProperties()

// Remove the quota
share.Properties.Quota <- Nullable()
share.SetProperties()
```

Создание подписи общего доступа для файла или файлового ресурса

Для файлового ресурса или отдельного файла можно создать подписанный URL-адрес (SAS). Также можно создать политики общего доступа на файловом ресурсе, чтобы управлять подписями общего доступа. Создание политики общего доступа рекомендуется, так как она позволяет отменить SAS, если она скомпрометирована.

Здесь вы создаете политику общего доступа для общей папки, а затем используете эту политику для предоставления ограничений для SAS для файла в общей папке.

```
// Create a 24-hour read/write policy.
let policy =
    SharedAccessFilePolicy
       (SharedAccessExpiryTime = (DateTimeOffset.UtcNow.AddHours(24.) |> Nullable),
        Permissions = (SharedAccessFilePermissions.Read ||| SharedAccessFilePermissions.Write))


// Set the policy on the share.
let permissions = share.GetPermissions()
permissions.SharedAccessPolicies.Add("policyName", policy)
share.SetPermissions(permissions)

let sasToken = file.GetSharedAccessSignature(policy)
let sasUri = Uri(file.StorageUri.PrimaryUri.ToString() + sasToken)

let fileSas = CloudFile(sasUri)
fileSas.UploadText("This write operation is authenticated via SAS")
```

Дополнительные сведения см. в статьях Использование подписанных URL-адресов (SAS) и Создание и использование подписанного URL-адреса в службе BLOB-объектов.

Копирование файлов

Файл можно скопировать в другой файл или в большой двоичный объект или в большой двоичный объект в файл. Если вы копируете большой двоичный объект в файл или файл в большой двоичный объект, *необходимо* использовать подписанный URL-адрес (SAS) для проверки подлинности исходного объекта, даже если вы копируете его в ту же учетную запись хранения.

Копирование файла в другой файл

Здесь вы копируете файл в другой файл в той же общей папке. Так как эта операция копирования осуществляет копирование между файлами в одной учетной записи хранения, для выполнения копирования можно использовать проверку подлинности Shared Key.

```
let destFile = subDir.GetFileReference("log_copy.txt")
destFile.StartCopy(file)
```

Копирование файла в большой двоичный объект

Здесь вы создаете файл и копируете его в большой двоичный объект в той же учетной записи хранения. Создайте SAS для исходного файла, который служба использует для проверки подлинности доступа к исходному файлу во время операции копирования.

```
// Get a reference to the blob to which the file will be copied.
let blobClient = storageAccount.CreateCloudBlobClient()
let container = blobClient.GetContainerReference("myContainer")
container.CreateIfNotExists()
let destBlob = container.GetBlockBlobReference("log_blob.txt")

let filePolicy =
    SharedAccessFilePolicy
        (Permissions = SharedAccessFilePermissions.Read,
         SharedAccessExpiryTime = (DateTimeOffset.UtcNow.AddHours(24.) |> Nullable))

let fileSas2 = file.GetSharedAccessSignature(filePolicy)
let sasUri2 = Uri(file.StorageUri.PrimaryUri.ToString() + fileSas2)
destBlob.StartCopy(sasUri2)
```

Таким же образом можно скопировать BLOB-объект в файл. Если исходным объектом является BLOB-объект, создайте SAS для проверки подлинности доступа к BLOB-объекту во время операции копирования.

Устранение неполадок хранилища файлов с помощью метрик

Аналитика Службы хранилища Azure поддерживает метрики для хранилища файлов. Данные метрик позволяют отслеживать запросы и диагностировать проблемы.

Вы можете включить метрики для хранилища файлов на портале Azure. также можно сделать это F# следующим образом:

```
open Microsoft.Azure.Storage.File.Protocol
open Microsoft.Azure.Storage.Shared.Protocol

let props =
    FileServiceProperties(
       (HourMetrics = MetricsProperties(
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = (14 |> Nullable),
            Version = "1.0"),
        MinuteMetrics = MetricsProperties(
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = (7 |> Nullable),
            Version = "1.0"))

fileClient.SetServiceProperties(props)
```

Следующие шаги

Дополнительную информацию о хранилище файлов Azure см. по этим ссылкам.

Тематические статьи и видео

- Хранилище файлов Azure: удобная облачная файловая система SMB для Windows и Linux
- Использование хранилища файлов Azure в Linux

Средства для работы с хранилищем файлов

- Использование Azure PowerShell со службой хранилища Azure
- Использование AzCopy со службой хранилища Microsoft Azure
- Использование интерфейса командной строки (CLI) Azure со службой хранилища Azure

Справочные сведения

- Справочник по клиентской библиотеке хранилища для .NET
- Справочник по REST API службы файлов

Записи блогов

- Хранилище файлов Azure стало общедоступным
- Inside Azure File Storage (Хранилище файлов Azure: взгляд изнутри)
- Введение в службы файлов Microsoft Azure
- Сохраняемые подключения к файлам Microsoft Azure

Приступая к работе с хранилищем очередей Azure с помощью F#

15.01.2020 • 11 minutes to read • Edit Online

Хранилище очередей Azure — это служба, обеспечивающая обмен сообщениями в облаке между компонентами приложения. При разработке приложений для масштабирования компоненты приложения часто не связаны между собой, так что они могут масштабироваться независимо друг от друга. Хранилище очередей обеспечивает асинхронный обмен сообщениями для взаимодействия между компонентами приложения независимо от того, где они выполняются: в облаке, на рабочем столе, локальном сервере или мобильном устройстве. Хранилище очередей также поддерживает управление асинхронными задачами и создание рабочих процессов.

Сведения об этом руководстве

В этом руководстве показано, как F# написать код для некоторых распространенных задач с помощью хранилища очередей Azure. В число описываемых задач входят создание и удаление очередей, а затем Добавление, чтение и удаление сообщений очереди.

Общие сведения о хранилище очередей см. в разделе "рекомендации по .NET" для хранилища очередей.

Prerequisites

Для работы с этим руководством необходимо сначала создать учетную запись хранения Azure. Вам также потребуется ключ доступа к хранилищу для этой учетной записи.

Создание F# скрипта и запуск F# интерактивного

Примеры в этой статье можно использовать как в F# приложении, так и в F# сценарии. Чтобы создать F# скрипт, создайте файл с расширением `.fsx`, например `queues.fsx`, в среде F# разработки.

Затем используйте Диспетчер пакетов, например пакет или NuGet, для установки пакета `WindowsAzure.Storage` и ссылки на `WindowsAzure.Storage.dll` в скрипте с помощью директивы `#r`.

Добавление объявлений пространств имен

Добавьте в начало файла `queues.fsx` следующие инструкции `open`:

```
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Queue // Namespace for Queue storage types
```

Получение строки подключения

Для работы с этим руководством вам потребуется строка подключения к службе хранилища Azure. Дополнительные сведения о строках подключения см. в разделе Настройка строк подключения к хранилищу.

В этом руководстве вы введете строку подключения в сценарий следующим образом:

```
let storageConnString = "..." // fill this in from your storage account
```

Однако это **не рекомендуется** для реальных проектов. Ключ учетной записи хранения похож на корневой пароль для вашей учетной записи хранения. Не забудьте защитить ключ учетной записи хранения. Не

сообщайте его другим пользователям, не определяйте его в коде и не храните его в текстовом файле, доступном другим пользователям. Вы можете повторно создать ключ с помощью портала Azure, если считаете, что он был скомпрометирован.

Для реальных приложений лучшим способом поддержания строки подключения к хранилищу является файл конфигурации. Чтобы получить строку подключения из файла конфигурации, можно сделать следующее:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

Использование диспетчера конфигураций Azure не является обязательным. Можно также использовать API, например тип `ConfigurationManager` .NET Framework.

Проанализируйте строку подключения

Чтобы проанализировать строку подключения, используйте:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

Это приведет к возврату `CloudStorageAccount` .

Создание клиента службы очередей

Класс `CloudQueueClient` позволяет получать очереди, хранящиеся в хранилище очередей. Вот один из способов создать клиента службы.

```
let queueClient = storageAccount.CreateCloudQueueClient()
```

Теперь вы можете написать код, который считывает и записывает данные в хранилище очередей.

Создать очередь

В этом примере показано, как создать очередь, если она еще не существует:

```
// Retrieve a reference to a container.
let queue = queueClient.GetQueueReference("myqueue")

// Create the queue if it doesn't already exist
queue.CreateIfNotExists()
```

Вставка сообщения в очередь

Чтобы вставить сообщение в существующую очередь, сначала создайте новый `CloudQueueMessage` . Затем вызовите метод `AddMessage` . `CloudQueueMessage` можно создать из строки (в формате UTF-8) или массива `byte` следующим образом:

```
// Create a message and add it to the queue.
let message = new CloudQueueMessage("Hello, World")
queue.AddMessage(message)
```

Просмотр следующего сообщения

Можно взглянуть на сообщение в начале очереди, не удаляя его из очереди, вызвав метод `PeekMessage`.

```
// Peek at the next message.
let peekedMessage = queue.PeekMessage()
let msgAsString = peekedMessage.AsString
```

Получить следующее сообщение для обработки

Вы можете получить сообщение в начале очереди для обработки, вызвав метод `GetMessage`.

```
// Get the next message. Successful processing must be indicated via DeleteMessage later.
let retrieved = queue.GetMessage()
```

Позднее можно указать успешную обработку сообщения с помощью `DeleteMessage`.

Изменение содержимого сообщения в очереди

Вы можете изменить содержимое извлеченного сообщения на месте в очереди. Если сообщение представляет собой рабочую задачу, можно использовать эту функцию для обновления состояния рабочей задачи. Следующий код добавляет новое содержимое в очередь сообщений и продлевает время ожидания видимости еще на 60 секунд. Это сохраняет состояние работы, связанной с данным сообщением, и позволяет клиенту продолжить работу с сообщением на протяжении еще одной минуты. Этот метод можно использовать для отслеживания многошаговых рабочих процессов по сообщениям в очереди без необходимости начинать с самого начала в случае сбоя шага обработки в связи с ошибкой аппаратного или программного обеспечения. Как правило, вы также сохраняете число повторных попыток, и если сообщение повторяется несколько раз, его можно удалить. Это обеспечивает защиту от сообщений, которые инициируют ошибку приложения при каждой попытке обработки.

```
// Update the message contents and set a new timeout.
retrieved.SetMessageContent("Updated contents.")
queue.UpdateMessage(retrieved,
    TimeSpan.FromSeconds(60.0),
    MessageUpdateFields.Content ||| MessageUpdateFields.Visibility)
```

Удаление следующего сообщения из очереди

Код удаляет сообщение из очереди в два этапа. При вызове `GetMessage` вы получаете следующее сообщение в очереди. Сообщение, возвращаемое методом `GetMessage`, становится невидимым для другого кода, считывающего сообщения из этой очереди. По умолчанию это сообщение остается невидимым в течение 30 секунд. Чтобы завершить удаление сообщения из очереди, необходимо также вызвать `DeleteMessage`. Этот двухэтапный процесс удаления сообщения позволяет удостовериться, что если коду не удастся обработать сообщение из-за сбоя оборудования или программного обеспечения, другой экземпляр кода сможет получить то же сообщение и повторить попытку. Код вызывает `DeleteMessage` сразу после обработки сообщения.

```
// Process the message in less than 30 seconds, and then delete the message.
queue.DeleteMessage(retrieved)
```

Использование асинхронных рабочих процессов с общими интерфейсами API хранилища очередей

В этом примере показано, как использовать асинхронный рабочий процесс с общими интерфейсами API хранилища очередей.

```
async {
    let! exists = queue.CreateIfNotExistsAsync() |> Async.AwaitTask

    let! retrieved = queue.GetMessageAsync() |> Async.AwaitTask

    // ... process the message here ...

    // Now indicate successful processing:
    do! queue.DeleteMessageAsync(retrieved) |> Async.AwaitTask
}
```

Дополнительные параметры для удаления сообщений из очереди

Способ извлечения сообщения из очереди можно настроить двумя способами. Во-первых, можно получить пакет сообщений (до 32 сообщений). Во-вторых, можно задать более длительное или короткое время ожидания видимости, чтобы предоставить коду больше или меньше времени на полную обработку каждого сообщения. В следующем примере кода используется `GetMessages` для получения 20 сообщений в одном вызове и последующей обработки каждого сообщения. Он также задает время ожидания невидимости 5 минут для каждого сообщения. Обратите внимание, что 5 минут начинается для всех сообщений одновременно, поэтому спустя 5 минут после вызова `GetMessages` все сообщения, которые не были удалены, снова станут видимыми.

```
for msg in queue.GetMessages(20, Nullable(TimeSpan.FromMinutes(5.))) do
        // Process the message here.
        queue.DeleteMessage(msg)
```

Получение длины очереди

Вы можете узнать приблизительное количество сообщений в очереди. Метод `FetchAttributes` запрашивает у служба очередей получение атрибутов очереди, включая число сообщений. Свойство `ApproximateMessageCount` возвращает последнее значение, полученное методом `FetchAttributes`, без вызова служба очередей.

```
queue.FetchAttributes()
let count = queue.ApproximateMessageCount.GetValueOrDefault()
```

Удаление очереди

Чтобы удалить очередь и все сообщения, содержащиеся в ней, вызовите метод `Delete` для объекта Queue.

```
// Delete the queue.
queue.Delete()
```

Следующие шаги

Вы изучили основные сведения о хранилище очередей. Дополнительные сведения о более сложных задачах по использованию хранилища можно найти по следующим ссылкам.

- API-интерфейсы хранилища Azure для .NET

- Поставщик типов службы хранилища Azure
- Блог рабочей группы службы хранилища Azure
- Настройка строк подключения службы хранилища Azure
- Azure Storage Services REST API Reference (Справочник по REST API службы хранилища Azure)

Приступая к работе с хранилищем таблиц Azure и Azure Cosmos DB API таблиц с помощью F#

15.01.2020 • 15 minutes to read • Edit Online

Хранилище таблиц Azure — это служба в облаке, в которой хранятся структурированные данные NoSQL. Хранилище таблиц — это хранилище ключей и атрибутов, реализованное в виде бессхемной конструкции. Такая конструкция хранилища таблиц позволяет легко адаптировать данные по мере расширения приложения. Быстрый и экономичный доступ к данным предоставляется приложениям всех видов. Табличное хранилище обычно значительно дешевле, чем традиционная база данных SQL для тех же объемов данных.

Табличное хранилище можно использовать для хранения гибких наборов данных, например пользовательских данных для веб-приложений, адресных книг, сведений об устройстве, а также метаданных любого другого типа, которые требуются вашей службе. В таблице можно хранить любое количество сущностей, а учетная запись хранения может содержать любое количество таблиц в пределах емкости учетной записи.

Azure Cosmos DB предоставляет API таблиц для приложений, написанных для хранилища таблиц Azure и требующих таких возможностей:

- Комплексные возможности глобального распределения.
- Выделенная пропускная способность по всему миру.
- задержка меньше 10 миллисекунд на уровне 99-го процентиля;
- гарантированно высокий уровень доступности;
- автоматическое вторичное индексирование.

С помощью API таблицы вы можете перенести приложения, написанные для хранилища таблиц Azure, в Azure Cosmos DB, не изменяя код, и воспользоваться возможностями уровня "Премиум". API таблицы включает клиентские пакеты SDK для .NET., Java, Python и Node.js.

Дополнительные сведения см. в статье Введение в Azure Cosmos DB API таблиц.

Сведения об этом руководстве

В этом учебнике показано, F# как написать код для выполнения некоторых распространенных задач с помощью хранилища таблиц Azure или Azure Cosmos DB API таблиц, включая создание и удаление таблицы, а также вставку, обновление, удаление и выполнение запросов к данным таблицы.

Prerequisites

Для работы с этим руководством необходимо сначала создать учетную запись хранения Azure или Azure Cosmos DB.

Создание F# скрипта и запуск F# интерактивного

Примеры в этой статье можно использовать как в F# приложении, так и в F# сценарии. Чтобы создать F# скрипт, создайте файл с расширением `.fsx`, например `tables.fsx`, в среде F# разработки.

Затем используйте Диспетчер пакетов , например пакет или NuGet , для установки пакета `WindowsAzure.Storage` и ссылки на `WindowsAzure.Storage.dll` в скрипте с помощью директивы `#r`. Повторите

эту операцию для `Microsoft.WindowsAzure.ConfigurationManager`, чтобы получить пространство имен Microsoft. Azure.

Добавление объявлений пространств имен

Добавьте в начало файла `tables.fsx` следующие инструкции `open`:

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Table // Namespace for Table storage types
```

Получение строки подключения к службе хранилища Azure

Если вы подключаетесь к службе таблиц службы хранилища Azure, вам потребуется строка подключения для этого руководства. Строку подключения можно скопировать из портал Azure. Дополнительные сведения о строках подключения см. в разделе Настройка строк подключения к хранилищу.

Получение строки подключения Azure Cosmos DB

Если вы подключаетесь к Azure Cosmos DB, вам потребуется строка подключения для этого руководства. Строку подключения можно скопировать из портал Azure. В портал Azure в учетной записи Cosmos DB перейдите в раздел **параметры** > **строка подключения**и нажмите кнопку **Копировать** , чтобы скопировать основную строку подключения.

Для этого руководства введите в скрипт строку подключения, как показано в следующем примере:

```
let storageConnString = "..." // fill this in from your storage account
```

Однако это **не рекомендуется** для реальных проектов. Ключ учетной записи хранения похож на корневой пароль для вашей учетной записи хранения. Не забудьте защитить ключ учетной записи хранения. Не сообщайте его другим пользователям, не определяйте его в коде и не храните его в текстовом файле, доступном другим пользователям. Вы можете повторно создать ключ с помощью портала Azure, если считаете, что он был скомпрометирован.

Для реальных приложений лучшим способом поддержания строки подключения к хранилищу является файл конфигурации. Чтобы получить строку подключения из файла конфигурации, можно сделать следующее:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

Использование диспетчера конфигураций Azure не является обязательным. Можно также использовать API, например тип `ConfigurationManager` .NET Framework.

Проанализируйте строку подключения

Чтобы проанализировать строку подключения, используйте:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

Это возвращает `CloudStorageAccount`.

Создание клиента службы таблиц

Класс `CloudTableClient` позволяет извлекать таблицы и сущности в хранилище таблиц. Вот один из способов

создать клиента службы.

```
// Create the table client.
let tableClient = storageAccount.CreateCloudTableClient()
```

Теперь вы можете написать код, который считывает и записывает данные в хранилище таблиц.

Создание таблицы

В этом примере показано, как создать таблицу, если она не существует:

```
// Retrieve a reference to the table.
let table = tableClient.GetTableReference("people")

// Create the table if it doesn't exist.
table.CreateIfNotExists()
```

Добавление сущности в таблицу

Сущность должна иметь тип, который наследуется от `TableEntity`. Можно расширять `TableEntity` любым способом, но у типа *должен* быть конструктор без параметров. В таблице Azure хранятся только свойства, которые содержат как `get`, так и `set`.

Ключ секции и строки сущности уникально идентифицируют сущность в таблице. Сущности с одинаковым ключом раздела можно запрашивать быстрее, чем с разными ключами раздела, но использование различных ключей разделов обеспечивает более высокую масштабируемость параллельных операций.

Ниже приведен пример `Customer`, в котором в качестве ключа секции используется `lastName`, а в качестве ключа строки — `firstName`.

```
type Customer(firstName, lastName, email: string, phone: string) =
    inherit TableEntity(partitionKey=lastName, rowKey=firstName)
    new() = Customer(null, null, null, null)
    member val Email = email with get, set
    member val PhoneNumber = phone with get, set

let customer =
    Customer("Walter", "Harp", "Walter@contoso.com", "425-555-0101")
```

Теперь добавьте `Customer` в таблицу. Для этого создайте `TableOperation`, который выполняется для таблицы. В этом случае создается операция `Insert`.

```
let insertOp = TableOperation.Insert(customer)
table.Execute(insertOp)
```

Вставка пакета сущностей

Пакет сущностей можно вставить в таблицу с помощью одной операции записи. Пакетные операции позволяют объединять операции в одно выполнение, но имеют некоторые ограничения.

- Обновления, удаления и вставки можно выполнять в одной и той же пакетной операции.
- Пакетная операция может включать до 100 сущностей.
- Все сущности в пакетной операции должны иметь один и тот же ключ секции.
- Хотя можно выполнить запрос в пакетной операции, он должен быть единственной операцией в пакете.

Ниже приведен код, объединяющий две операции вставки в пакетную операцию:

```
let customer1 =
    Customer("Jeff", "Smith", "Jeff@contoso.com", "425-555-0102")

let customer2 =
    Customer("Ben", "Smith", "Ben@contoso.com", "425-555-0103")

let batchOp = TableBatchOperation()
batchOp.Insert(customer1)
batchOp.Insert(customer2)
table.ExecuteBatch(batchOp)
```

Получение всех сущностей в разделе

Чтобы запросить все сущности из таблицы, используйте объект `TableQuery`. Здесь выполняется фильтрация для сущностей, где "Иванов" — ключ секции.

```
let query =
    TableQuery<Customer>().Where(
        TableQuery.GenerateFilterCondition(
            "PartitionKey", QueryComparisons.Equal, "Smith"))

let result = table.ExecuteQuery(query)
```

Теперь результаты будут напечатаны следующим образом:

```
for customer in result do
    printfn "customer: %A %A" customer.RowKey customer.PartitionKey
```

Получение диапазона сущностей в разделе

Если вы не хотите запрашивать все сущности в разделе, можно указать диапазон, объединив фильтр ключа раздела с фильтром ключа строк. Здесь используются два фильтра для получения всех сущностей в разделе "Smith", где ключ строки (имя) начинается с буквы, предшествующей "M", в алфавите.

```
let range =
    TableQuery<Customer>().Where(
        TableQuery.CombineFilters(
            TableQuery.GenerateFilterCondition(
                "PartitionKey", QueryComparisons.Equal, "Smith"),
            TableOperators.And,
            TableQuery.GenerateFilterCondition(
                "RowKey", QueryComparisons.LessThan, "M")))

let rangeResult = table.ExecuteQuery(range)
```

Теперь результаты будут напечатаны следующим образом:

```
for customer in rangeResult do
    printfn "customer: %A %A" customer.RowKey customer.PartitionKey
```

Извлечение одной сущности

Можно написать запрос для получения отдельной сущности. Здесь вы используете `TableOperation`, чтобы указать клиента «Бен Смит». Вместо коллекции вы получаете `Customer`. Указание как ключа секции, так и ключа строки в запросе — самый быстрый способ извлечь одну сущность из службы таблиц.

```
let retrieveOp = TableOperation.Retrieve<Customer>("Smith", "Ben")

let retrieveResult = table.Execute(retrieveOp)
```

Теперь результаты будут напечатаны следующим образом:

```
// Show the result
let retrieveCustomer = retrieveResult.Result :?> Customer
printfn "customer: %A %A" retrieveCustomer.RowKey retrieveCustomer.PartitionKey
```

Замена сущности

Чтобы обновить сущность, извлеките ее из службы таблиц, измените объект сущности, а затем сохраните изменения обратно в службу таблиц с помощью операции `Replace`. Это приводит к тому, что сущность будет полностью заменена на сервере, если только сущность на сервере не изменялась с момента получения. в этом случае операция завершается ошибкой. Этот сбой заключается в предотвращении непреднамеренной перезаписи изменений из других источников в приложении.

```
try
    let customer = retrieveResult.Result :?> Customer
    customer.PhoneNumber <- "425-555-0103"
    let replaceOp = TableOperation.Replace(customer)
    table.Execute(replaceOp) |> ignore
    Console.WriteLine("Update succeeeded")
with e ->
    Console.WriteLine("Update failed")
```

Вставка или замена сущности

Иногда неизвестно, существует ли сущность в таблице. Если это так, текущие значения, хранящиеся в нем, больше не нужны. С помощью `InsertOrReplace` можно создать сущность или заменить ее, если она существует, независимо от ее состояния.

```
try
    let customer = retrieveResult.Result :?> Customer
    customer.PhoneNumber <- "425-555-0104"
    let replaceOp = TableOperation.InsertOrReplace(customer)
    table.Execute(replaceOp) |> ignore
    Console.WriteLine("Update succeeeded")
with e ->
    Console.WriteLine("Update failed")
```

Запрос подмножества свойств сущности

Запрос таблицы может получить лишь несколько свойств сущности, а не все. Этот метод, называемый проекцией, может повысить производительность запросов, особенно для больших сущностей. Здесь вы возвращаете только адреса электронной почты с помощью `DynamicTableEntity` и `EntityResolver`. Обратите внимание, что проекция не поддерживается в локальном эмуляторе хранения, поэтому этот код выполняется только при использовании учетной записи хранения в службе таблиц.

```
// Define the query, and select only the Email property.
let projectionQ = TableQuery<DynamicTableEntity>().Select [|"Email"|]

// Define an entity resolver to work with the entity after retrieval.
let resolver = EntityResolver<string>(fun pk rk ts props etag ->
    if props.ContainsKey("Email") then
        props.["Email"].StringValue
    else
        null
    )

let resolvedResults = table.ExecuteQuery(projectionQ, resolver, null, null)
```

Асинхронное получение объектов со страниц

Если вы читаете большое количество сущностей и хотите обрабатывать их по мере их извлечения, а не ждать их возврата, можно использовать сегментированный запрос. Здесь вы возвращаете результаты на страницах с помощью асинхронного рабочего процесса, чтобы выполнение не блокировалось, пока не будет возвращен большой набор результатов.

```
let tableQ = TableQuery<Customer>()

let asyncQuery =
    let rec loop (cont: TableContinuationToken) = async {
        let! ct = Async.CancellationToken
        let! result = table.ExecuteQuerySegmentedAsync(tableQ, cont, ct) |> Async.AwaitTask

        // ...process the result here...

        // Continue to the next segment
        match result.ContinuationToken with
        | null -> ()
        | cont -> return! loop cont
    }
    loop null
```

Теперь это вычисление выполняется синхронно:

```
let asyncResults = asyncQuery |> Async.RunSynchronously
```

Удаление сущности

Сущность можно удалить после ее получения. Как и при обновлении сущности, эта ошибка возникает, если сущность изменилась со времени ее получения.

```
let deleteOp = TableOperation.Delete(customer)
table.Execute(deleteOp)
```

Удаление таблицы

Вы можете удалить таблицу из учетной записи хранения. Удаленную таблицу нельзя воссоздать в течение определенного времени после удаления.

```
table.DeleteIfExists()
```

Следующие шаги

Теперь, когда вы узнали основные сведения о хранилище таблиц, перейдите по следующим ссылкам, чтобы узнать о более сложных задачах хранилища и Azure Cosmos DB API таблиц.

- Общие сведения об API таблиц Azure Cosmos DB
- Справочник по клиентской библиотеке хранилища для .NET
- Поставщик типов службы хранилища Azure
- Блог рабочей группы службы хранилища Azure
- Настройка строк подключения

Управление пакетами для зависимостей F# в Azure

23.10.2019 • 3 minutes to read • Edit Online

Получение пакетов для разработки Azure несложно при использовании диспетчера пакетов. Два варианта — пакет и NuGet.

Использование пакет

Если вы используете пакет в качестве диспетчера зависимостей, вы можете использовать это `paket.exe` средство для добавления зависимостей Azure. Например:

```
> paket add nuget WindowsAzure.Storage
```

Или, если вы используете Mono для кросс-платформенной разработки .NET:

```
> mono paket.exe add nuget WindowsAzure.Storage
```

При этом будет `WindowsAzure.Storage` добавлен набор зависимостей пакета для проекта в текущем каталоге, `paket.dependencies` изменен файл и загружен пакет. Если вы ранее настроили зависимости или работаете с проектом, в котором зависимости были настроены другим разработчиком, вы можете разрешить и установить зависимости в локальной среде следующим образом:

```
> paket install
```

Или, для разработки Mono:

```
> mono paket.exe install
```

Вы можете обновить все зависимости пакета до последней версии следующим образом:

```
> paket update
```

Или, для разработки Mono:

```
> mono paket.exe update
```

Использование NuGet

Если вы используете NuGet в качестве диспетчера зависимостей, это `nuget.exe` средство можно использовать для добавления зависимостей Azure. Например:

```
> nuget install WindowsAzure.Storage -ExcludeVersion
```

Или, для разработки Mono:

```
> mono nuget.exe install WindowsAzure.Storage -ExcludeVersion
```

Это приведет к `WindowsAzure.Storage` добавлению набора зависимостей пакета для проекта в текущем каталоге и загрузке пакета. Если вы ранее настроили зависимости или работаете с проектом, в котором зависимости были настроены другим разработчиком, вы можете разрешить и установить зависимости в локальной среде следующим образом:

```
> nuget restore
```

Или, для разработки Mono:

```
> mono nuget.exe restore
```

Вы можете обновить все зависимости пакета до последней версии следующим образом:

```
> nuget update
```

Или, для разработки Mono:

```
> mono nuget.exe update
```

Ссылки на сборки

Чтобы использовать пакеты в F# скрипте, необходимо сослаться на сборки, содержащиеся в пакетах, с помощью `#r` директивы. Например:

```
> #r "packages/WindowsAzure.Storage/lib/net40/Microsoft.WindowsAzure.Storage.dll"
```

Как видите, необходимо указать относительный путь к библиотеке DLL и полное имя библиотеки DLL, которое может не совпадать с именем пакета. Путь будет содержать версию платформы и, возможно, номер версии пакета. Чтобы найти все установленные сборки, можно использовать в командной строке Windows нечто подобное:

```
> cd packages/WindowsAzure.Storage
> dir /s/b *.dll
```

Или в оболочке UNIX примерно так:

```
> find packages/WindowsAzure.Storage -name "*.dll"
```

При этом будут предоставлены пути к установленным сборкам. После этого можно выбрать правильный путь к версии платформы.