EBook Gratis

APRENDIZAJE functional-programming

Free unaffiliated eBook created from **Stack Overflow contributors.**

#functional-programming

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: functional-programming

It is an unofficial and free functional-programming ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official functional-programming.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con la programación funcional.

Observaciones

La programación funcional es un paradigma de programación que modela los cálculos (y, por lo tanto, los programas) como la evaluación de funciones matemáticas. Tiene sus raíces en el cálculo lambda, que fue desarrollado por Alonzo Church en su investigación sobre computabilidad.

La programación funcional tiene algunos conceptos interesantes:

- **Funciones de orden superior**
- **Pureza**
- **Recursion**
- **pereza**
- **Transparencia referencial**
- **Zurra**
- **Functores**
- **Mónadas**
- **Memoización y optimización de llamadas de cola**
- **Unidad funcional de prueba**

Algunos ejemplos de lenguajes de programación funcionales son Lisp , Haskell , Scala y Clojure , pero también otros lenguajes, como Python , R y Javascript permiten escribir (partes de) sus programas en un estilo funcional. Incluso en Java , la programación funcional ha encontrado su lugar con Lambda Expressions y la API Stream que se introdujeron en Java 8.

Examples

Funciones puras

Las funciones puras son autónomas y no tienen efectos secundarios. Dado el mismo conjunto de entradas, una función pura siempre devolverá el mismo valor de salida.

La siguiente función es pura:

```
function pure(data) {
    return data.total + 3;
}
```

Sin embargo, esta función no es pura, ya que modifica una variable externa:

```
function impure(data) {
    data.total += 3;
```

```
    return data.total;
}
```

Ejemplo:

```
data = {
    total: 6
};

pure(data);   // outputs: 9
impure(data); // outputs: 9 (but now data.total has changed)
impure(data); // outputs: 12
```

Funciones de orden superior

Las funciones de orden superior toman otras funciones como argumentos y / o las devuelven como resultados. Forman los bloques de construcción de la programación funcional. La mayoría de los lenguajes funcionales tienen alguna forma de función de filtro, por ejemplo. Esta es una función de orden superior, que toma una lista y un predicado (función que devuelve verdadero o falso) como argumentos.

Las funciones que no hacen ninguna de estas acciones a menudo se denominan funciones de `first-order functions`.

```
function validate(number,predicate) {
    if (predicate) {    // Is Predicate defined
        return predicate(number);
    }
    return false;
}
```

Aquí "predicado" es una función que probará alguna condición que involucre sus argumentos y devolverá verdadero o falso.

Un ejemplo de llamada para lo anterior es:

```
validate(someNumber, function(arg) {
    return arg % 10 == 0;
    }
);
```

Un requisito común es agregar números dentro de un rango. Al usar funciones de orden superior, podemos ampliar esta capacidad básica, aplicando una función de transformación en cada número antes de incluirla en la suma.

Desea agregar todos los enteros dentro de un rango dado (usando Scala)

```
def sumOfInts(a: Int, b: Int): Int = {
  if(a > b) 0
  else a + sumOfInts(a+1, b)
}
```

Desea agregar cuadrados de todos los enteros dentro de un rango dado

```
def square(a: Int): Int = a * a

def sumOfSquares(a: Int, b: Int): Int = {
  if(a > b) 0
  else square(a) + sumOfSquares(a + 1, b)
}
```

Tenga en cuenta que estas cosas tienen una cosa en común, que desea aplicar una función en cada argumento y luego agregarlas.

Permite crear una función de orden superior para hacer ambas cosas:

```
def sumHOF(f: Int => Int, a: Int, b: Int): Int = {
  if(a > b) 0
  else f(a) + sumHOF(f, a + 1, b)
}
```

Puedes llamarlo así:

```
def identity(a: Int): Int = a

def square(a: Int): Int = a * a
```

Tenga en cuenta que `sumOfInts` y `sumOfSquare` se pueden definir como:

```
def sumOfInts(a: Int, b: Int): Int = sumHOF(identity, a, b)

def sumOfSquares(a: Int, b: Int): Int = sumHOF(square, a, b)
```

Como puede ver en este ejemplo simple, las funciones de orden superior proporcionan soluciones más generalizadas y reducen la duplicación de código.

> He usado Scala By Example, por Martin Odersky como referencia.

Zurra

Currying es el proceso de transformar una función que toma múltiples argumentos en una secuencia de funciones, cada una de las cuales tiene un solo parámetro. El currying está relacionado con, pero no lo mismo que, la aplicación parcial.

Consideremos la siguiente función en JavaScript:

```
var add = (x, y) => x + y
```

Podemos usar la definición de curry para reescribir la función de agregar:

```
var add = x => y => x + y
```

Esta nueva versión toma un solo parámetro, `x` , y devuelve una función que toma un solo parámetro, `y` , que finalmente devolverá el resultado de agregar `x` y `y` .

```
var add5 = add(5)
var fifteen = add5(10) // fifteen = 15
```

Otro ejemplo es cuando tenemos las siguientes funciones que ponen corchetes alrededor de cadenas:

```
var generalBracket = (prefix, str, suffix) => prefix + str + suffix
```

Ahora, cada vez que usamos `generalBracket` tenemos que pasar entre paréntesis:

```
var bracketedJim = generalBracket("{", "Jim", "}") // "{Jim}"
var doubleBracketedJim = generalBracket("{{", "Jim", "}}") // "{{Jim}}"
```

Además, si pasamos las cadenas que no son corchetes, nuestra función todavía devuelve un resultado incorrecto. Vamos a arreglar eso:

```
var generalBracket = (prefix, suffix) => str => prefix + str + suffix
var bracket = generalBracket("{", "}")
var doubleBracket = generalBracket("{{", "}}")
```

Observe que tanto `bracket` como `doubleBracket` ahora son funciones que esperan su parámetro final:

```
var bracketedJim = bracket("Jim") // "{Jim}"
var doubleBracketedJim = doubleBracket("Jim") // "{{Jim}}"
```

Inmutabilidad

En los lenguajes tradicionales orientados a objetos, `x = x + 1` es una expresión simple y legal. Pero en la Programación Funcional, es *ilegal* .

Las variables no existen en la programación funcional. Los valores almacenados todavía se llaman variables solo por el historial. De hecho, son *constantes* . Una vez que `x` toma un valor, es ese valor para la vida.

Entonces, si una *variable* es una *constante* , ¿cómo podemos cambiar su valor?

La programación funcional se ocupa de los cambios en los valores de un registro al hacer una copia del registro con los valores cambiados.

Por ejemplo, en lugar de hacer:

```
var numbers = [1, 2, 3];
numbers[0] += 1; // numbers = [2, 2, 3];
```

Tú lo haces:

```
var numbers = [1, 2, 3];
var newNumbers = numbers.map(function(number) {
    if (numbers.indexOf(number) == 0)
        return number + 1
    return number
});
console.log(newNumbers) // prints [2, 2, 3]
```

Y no hay bucles en la Programación Funcional. Usamos funciones de recursión o de orden superior como `map` , `filter` y `reduce` para evitar los bucles.

Vamos a crear un simple bucle en JavaScript:

```
var acc = 0;
for (var i = 1; i <= 10; ++i)
    acc += i;
console.log(acc); // prints 55
```

Todavía podemos hacerlo mejor cambiando la vida útil de `acc` de global a local:

```
function sumRange(start, end, acc) {
    if (start > end)
        return acc;
    return sumRange(start + 1, end, acc + start)
}
console.log(sumRange(1, 10, 0)); // 55
```

Ninguna variable o bucle significa un código más simple, más seguro y más legible (especialmente cuando se realiza la depuración o la prueba; no necesita preocuparse por el valor de `x` después de una serie de declaraciones, nunca cambiará).

Lea Empezando con la programación funcional. en línea: https://riptutorial.com/es/functional-programming/topic/3184/empezando-con-la-programacion-funcional-

Capítulo 2: Bucles por Funciones Recursivas y Cola Recursivas

Introducción

Como ya sabe, en aras de la inmutabilidad, no puede procesar datos utilizando bucles para bucles y mientras bucles. Así que tenemos funciones recursivas para rescatar.

Examples

no recursivo (donde la inmutabilidad no es una preocupación)

```
function sum(numbers) {
    var total = 0;
    for (var i = numbers.length - 1; i >= 0; i--) {
        total += numbers[i];
    }
    return total;
}
```

Es un código de procedimiento con mutaciones (sobre el `total`).

recursivo al rescate

```
function sum(numbers) {
    if(numbers.length == 0) {
        return 0;
    }
    return numbers[0] + sum(numbers.slice(1));
}
```

Esta es la versión recursiva. no hay mutación, pero estamos haciendo una pila de llamadas como la de abajo, que usa memoria adicional.

suma ([10, 5, 6, 7]);

10 + suma ([5, 6, 7]);

10 + 5 + suma ([6, 7]);

10 + 5 + 6 + suma ([7]);

10 + 5 + 6 + 7 + suma ([]);

10 + 5 + 6 + 7 + 0;

Cola recursiva para optimizar.

```
function sum(numbers) {
    return tail_sum(numbers, 0);
}

function tail_sum(numbers, acc) {
    if(numbers.length == 0) {
        return acc;
    }
    return tail_sum(numbers.slice(1), acc + numbers[0]);
}
```

en la versión recursiva de cola, el valor de retorno de la función no necesita esperar hasta el final para realizar sus cálculos, por lo que no hay una pila enorme aquí; Solo dos niveles.

suma ([10, 5, 6, 7]);

tail_sum ([10, 5, 6, 7], 0);

tail_sum ([5, 6, 7], 10);

tail_sum ([6, 7], 15);

tail_sum ([7], 21);

tail_sum ([], 28);

28;

Lea Bucles por Funciones Recursivas y Cola Recursivas en línea: https://riptutorial.com/es/functional-programming/topic/10819/bucles-por-funciones-recursivas-y-cola-recursivas

Creditos

S. No	Capítulos	Contributors
1	Empezando con la programación funcional.	AJW, Avinash Anand, Community, Huy Vo, Keelan, MauroPorrasP, Mr Tsjolder, rasmeister, rst_ack, tomturton, Zoyd
2	Bucles por Funciones Recursivas y Cola Recursivas	Dariush Alipour