

**UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS**  
**CIÊNCIA DA COMPUTAÇÃO**

**ANA BEATRIZ STAHL**  
**THAIS SANTOS LANDFELDT**

**TUTORIAL INTRODUTÓRIO SOBRE OS PARADIGMAS DE PROGRAMAÇÃO**  
**CONCORRENTE E FUNCIONAL**

São Leopoldo

2024

# **1 PARADIGMA CONCORRENTE**

## **1.1 INTRODUÇÃO**

### **1.1.1 Definição**

O paradigma concorrente é um estilo de programação que permite a execução de múltiplas operações simultaneamente. Ele se concentra em dividir um programa em partes independentes que podem ser executadas em paralelo, aproveitando ao máximo os recursos do sistema, como processadores multi-core, para melhorar o desempenho e a eficiência.

### **1.1.2 Características**

As principais características do paradigma concorrente são:

- Multithreading: A capacidade de um programa criar e gerenciar múltiplas threads de execução;
- Sincronização: Mecanismos para controlar a interação entre threads, evitando condições de corrida e garantindo a consistência dos dados;
- Comunicação entre processos: Técnicas que permitem a troca de informações entre diferentes threads ou processos;
- Escalonamento: A maneira como o sistema operacional gerencia a execução dos threads, atribuindo tempo de CPU e recursos.

### **1.1.3 Histórico**

A programação concorrente começou a ganhar relevância com o advento dos sistemas operacionais multitarefa na década de 1960. Com o aumento do poder computacional e a introdução de processadores multi-core, a necessidade de programação concorrente tornou-se mais pronunciada. Linguagens como Java e C++ introduziram bibliotecas e frameworks para facilitar a implementação de programas concorrentes.

## 1.2 EXEMPLOS DE CÓDIGO

### 1.2.1 Multithreading em Java

```
1 public class ExampleThread extends Thread {
2     private String threadName;
3
4     ExampleThread(String name) {
5         threadName = name;
6     }
7
8     public void run() {
9         for (int i = 0; i < 5; i++) {
10             System.out.println(threadName + " is running: " + i);
11             try {
12                 Thread.sleep(50);
13             } catch (InterruptedException e) {
14                 System.out.println(threadName + " interrupted.");
15             }
16         }
17     }
18
19     public static void main(String[] args) {
20         ExampleThread t1 = new ExampleThread("Thread 1");
21         ExampleThread t2 = new ExampleThread("Thread 2");
22
23         t1.start();
24         t2.start();
25     }
26 }
27
```

Mostrar menos

A classe `ExampleThread` estende a classe `Thread`, o que permite criar uma definição de thread personalizada. No método `run()`, o código a ser executado pelo thread é definido, consistindo em um loop que imprime o nome do thread e um contador. O método `Thread.sleep(50)` faz a thread "dormir" por 50 milissegundos, simulando uma pausa na execução. O método `main()` cria duas instâncias de `ExampleThread` e as inicia com `start()`, fazendo com que ambas os threads sejam executados simultaneamente.

## 1.2.2 Sincronização em Java

```
1 class Counter {
2     private int count = 0;
3
4     public synchronized void increment() {
5         count++;
6     }
7
8     public synchronized int getCount() {
9         return count;
10    }
11 }
12
13 public class SyncExample {
14     public static void main(String[] args) {
15         Counter counter = new Counter();
16
17         Thread t1 = new Thread(() -> {
18             for (int i = 0; i < 1000; i++) {
19                 counter.increment();
20             }
21         });
22
23         Thread t2 = new Thread(() -> {
24             for (int i = 0; i < 1000; i++) {
25                 counter.increment();
26             }
27         });
28
29         t1.start();
30         t2.start();
31
32         try {
33             t1.join();
34             t2.join();
35         } catch (InterruptedException e) {
36             e.printStackTrace();
37         }
38
39         System.out.println("Final count: " + counter.getCount());
40     }
41 }
42
```

Mostrar menos

A classe `Counter` possui métodos `increment` e `getCount` sincronizados. A palavra-chave `synchronized` garante que apenas um thread por vez possa executar esses métodos, evitando condições de corrida. No `main()`, dois threads são criados, cada um incrementando o contador 1000 vezes. A chamada `join()` faz com que o `main()` espere a conclusão dos threads antes de imprimir o valor final do contador. Isso garante que o valor impresso seja a soma correta dos incrementos, evitando inconsistências.

### 1.2.3 Comunicação entre Processos em Python

```
1 from multiprocessing import Process, Queue
2
3 def worker(queue):
4     queue.put('Hello from worker')
5
6 if __name__ == '__main__':
7     queue = Queue()
8     process = Process(target=worker, args=(queue,))
9     process.start()
10    print(queue.get())
11    process.join()
12
```

Em Python, a biblioteca multiprocessing permite criar processos que podem se comunicar usando filas (`Queue`). A função worker coloca uma mensagem na fila. No main(), uma fila é criada e passada ao processo worker. Após iniciar o processo, o main() recupera a mensagem da fila e a imprime. A chamada join() garante que o main() espere a conclusão do processo antes de continuar. Este exemplo demonstra como dados podem ser trocados entre processos de maneira segura.

### 1.2.4 Escalonamento em Python com asyncio

```
1 import asyncio
2
3 async def say_hello():
4     print('Hello')
5     await asyncio.sleep(1)
6     print('World')
7
8 async def main():
9     await asyncio.gather(say_hello(), say_hello())
10
11 asyncio.run(main())
12
```

O módulo asyncio em Python fornece uma forma de realizar escalonamento cooperativo, onde a execução é voluntariamente cedida pelas tarefas, permitindo a execução concorrente de múltiplas tarefas. A função say\_hello é uma coroutine que usa await asyncio.sleep(1) para simular uma operação de bloqueio. A função main usa asyncio.gather para executar duas instâncias de say\_hello concorrentemente. A chamada asyncio.run(main()) inicia o loop de eventos asyncio, que gerencia a execução das coroutines.

### 1.3 COMPARAÇÃO COM OUTROS PARADIGMAS

Comparado com o paradigma funcional, que foca na imutabilidade e funções puras, evitando estado compartilhado e simplificando a concorrência, o paradigma concorrente é mais direto para modelar problemas que envolvem estados mutáveis. No entanto, a concorrência no paradigma funcional pode ser mais segura devido à ausência de efeitos colaterais. Em relação ao paradigma orientado a objetos, que encapsula estado e comportamento dentro de objetos, a programação concorrente requer mais cuidados com sincronização para evitar condições de corrida. Já o paradigma lógico, que se baseia em regras e inferência lógica, não se concentra na concorrência, mas sim em resolver problemas declarando fatos e regras, delegando a resolução ao sistema de inferência.

#### 1.3.1 Vantagens do Paradigma Concorrente

A principal vantagem do paradigma concorrente é a eficiência. Ele permite uma melhor utilização dos recursos de hardware, aproveitando múltiplos núcleos de processamento para executar tarefas simultaneamente, o que pode levar a um aumento significativo no desempenho, especialmente em tarefas complexas e computacionalmente intensivas. Além disso, a concorrência pode melhorar a responsividade de aplicações, como interfaces gráficas e servidores web, ao permitir que diferentes partes do programa sejam executadas ao mesmo tempo.

#### 1.3.2 Desvantagens do Paradigma Concorrente

Com o uso do paradigma concorrente a complexidade do código aumenta, pois é necessário gerenciar a sincronização e a comunicação entre threads, o que pode levar a problemas difíceis de depurar, como condições de corrida e deadlocks. Esses problemas podem resultar em comportamento imprevisível e erros sutis que são difíceis de identificar e corrigir. Portanto, embora a concorrência ofereça benefícios substanciais, ela também exige um maior nível de cuidado e experiência por parte dos desenvolvedores.

## 1.4 RECURSOS ADICIONAIS

Leituras Adicionais:

- [Java Concurrency in Practice](#) por Brian Goetz
- [Concurrency in Go](#) por Katherine Cox-Buday

Ferramentas:

- [Apache Kafka](#): Plataforma de streaming distribuída
- [Akka](#): Toolkit para construir aplicações concorrentes em Scala e Java

## 2 PARADIGMA FUNCIONAL

### 2.1 INTRODUÇÃO

#### 2.1.1 Definição

O paradigma funcional é um estilo de programação que trata a computação como a avaliação de funções matemáticas e evita mudanças de estado e dados mutáveis. Em vez de comandos sequenciais, a ênfase está em funções puras, imutabilidade e expressões. Este método enfatiza a aplicação de funções, as quais são tratadas como valores de primeira importância, ou seja, funções podem ser parâmetros ou valores de entrada para outras funções e podem ser os valores de retorno ou saída de uma função.

#### 2.1.2 Característica

As principais características do paradigma funcional são:

- **Funções Puras:** Funções que, para o mesmo conjunto de argumentos, sempre retornam o mesmo resultado e não têm efeitos colaterais;
- **Imutabilidade:** Os dados não são alterados após serem criados. Em vez disso, novas estruturas de dados são criadas a partir das antigas;
- **Funções de Ordem Superior:** Funções podem ser passadas como argumentos para outras funções, retornadas como resultados e atribuídas a variáveis;
- **Recursão:** Uso de funções que se chamam a si mesmas como uma alternativa a loops imperativos;
- **Transparência Referencial:** Uma expressão pode ser substituída por seu valor sem alterar o comportamento do programa.

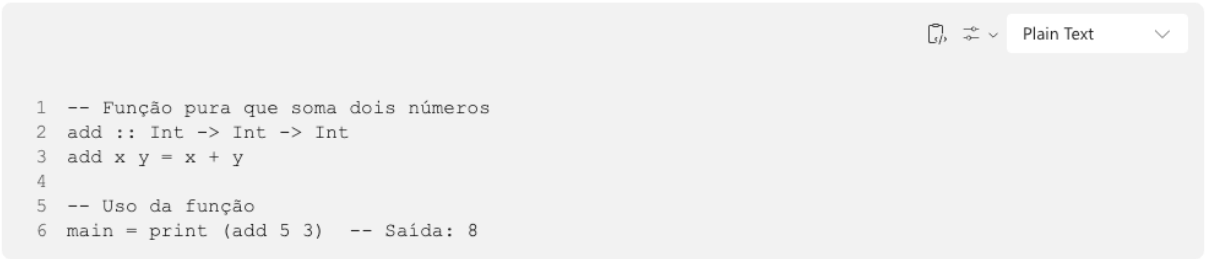


### 2.1.3 Histórico

O paradigma funcional tem suas raízes no cálculo lambda, uma formalização da computação desenvolvida por Alonzo Church na década de 1930. Linguagens funcionais como Lisp (1958), Haskell (1990), e mais recentemente, Scala e F#, têm promovido e popularizado esse estilo de programação. O interesse renovado em programação funcional surgiu devido à sua capacidade de lidar bem com concorrência e paralelismo.

## 2.2 EXEMPLOS DE CÓDIGO

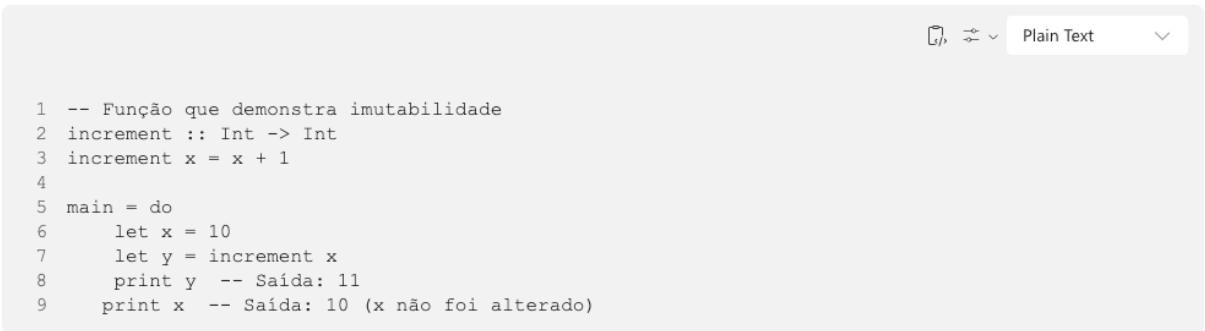
### 2.2.1 Funções Puras

A code editor window with a light gray background. In the top right corner, there are icons for file operations and a dropdown menu labeled "Plain Text". The code is as follows:

```
1 -- Função pura que soma dois números
2 add :: Int -> Int -> Int
3 add x y = x + y
4
5 -- Uso da função
6 main = print (add 5 3) -- Saída: 8
```

A função `add` é pura porque, para qualquer par de inteiros `x` e `y`, ela sempre retorna a soma de `x` e `y` sem efeitos colaterais. Executar `add 5 3` sempre resultará em 8.

### 2.2.2 Imutabilidade

A code editor window with a light gray background. In the top right corner, there are icons for file operations and a dropdown menu labeled "Plain Text". The code is as follows:

```
1 -- Função que demonstra imutabilidade
2 increment :: Int -> Int
3 increment x = x + 1
4
5 main = do
6   let x = 10
7   let y = increment x
8   print y -- Saída: 11
9   print x -- Saída: 10 (x não foi alterado)
```

Neste exemplo, `increment` recebe um valor `x` e retorna `x + 1`. As variáveis `x` e `y` são imutáveis. Depois de incrementarmos `x` e atribuirmos a `y`, `x` mantém seu valor original.

### 2.2.3 Funções de Ordem Superior

```
1 -- Função que aplica outra função duas vezes
2 applyTwice :: (a -> a) -> a -> a
3 applyTwice f x = f (f x)
4
5 -- Uso da função
6 main = print (applyTwice (+3) 10) -- Saída: 16
```

A função `applyTwice` recebe uma função `f` e um valor `x`. Ela aplica `f` a `x` duas vezes. No exemplo, `(+3)` é aplicado duas vezes ao valor `10`, resultando em `16`.

### 2.2.4 Recursão

```
1 -- Função recursiva para calcular o fatorial
2 factorial :: Int -> Int
3 factorial 0 = 1
4 factorial n = n * factorial (n - 1)
5
6 main = print (factorial 5) -- Saída: 120
```

A função `factorial` usa recursão para calcular o fatorial de um número. A base da recursão é `factorial 0 = 1`. Para outros valores, `factorial n` é calculado como `n * factorial (n - 1)`.

### 2.2.5 Transparência Referencial

```
1 -- Definindo uma função pura
2 square :: Int -> Int
3 square x = x * x
4
5 -- Usando a função em uma expressão
6 main :: IO ()
7 main = do
8     let a = 3
9     let b = square a
10    let c = square 3
11    print b -- Saída: 9
12    print c -- Saída: 9
```

Substituindo `square a` diretamente pelo seu valor `9`, não alteramos o comportamento do programa.

## 2.3 COMPARAÇÃO COM OUTROS PARADIGMAS

Comparado com o paradigma concorrente, que permite a execução de múltiplas operações simultaneamente e é essencial para desenvolver softwares eficiente em sistemas modernos, o paradigma funcional simplifica a concorrência por evitar estado mutável e efeitos colaterais, eliminando a necessidade de mecanismos complexos de sincronização. Em relação ao paradigma orientado a objetos, que encapsula estado e comportamento dentro de objetos e pode levar a um código mais fácil de entender em termos de modelos do mundo real, a programação funcional evita mudanças de estado, o que pode resultar em um código mais previsível, embora menos intuitivo para modelar certas situações. Já o paradigma lógico, que se baseia em regras e inferência lógica, oferece uma abordagem declarativa diferente, focando mais em encontrar soluções que satisfazem certas condições, enquanto o paradigma funcional é mais sobre funções e transformações de dados.

### 2.3.1 Vantagens do Paradigma Funcional

A programação funcional oferece várias vantagens, incluindo a facilidade de raciocínio sobre o código devido à ausência de efeitos colaterais, tornando o comportamento do programa mais previsível e fácil de testar. A imutabilidade inerente simplifica a programação concorrente, pois elimina a necessidade de mecanismos de sincronização complexos. Além disso, funções puras são altamente reutilizáveis e podem ser combinadas de maneiras poderosas, facilitando a construção de programas modulares e componíveis. Essas características tornam o paradigma funcional especialmente atraente para projetos que requerem alta confiabilidade e facilidade de manutenção.

### 2.3.2 Desvantagens do Paradigma Funcional

Por outro lado, o paradigma funcional também apresenta algumas desvantagens. A criação de novas estruturas de dados em vez de modificar as existentes pode ser menos eficiente em termos de desempenho, especialmente em sistemas que exigem operações intensivas de entrada e saída. A curva de aprendizado pode ser íngreme

para desenvolvedores acostumados a paradigmas imperativos, exigindo uma mudança significativa na maneira de pensar e escrever código. Além disso, a falta de familiaridade com conceitos funcionais pode dificultar a adoção em equipes de desenvolvimento mais tradicionais, e algumas tarefas podem ser menos intuitivas de implementar em um estilo puramente funcional.

## 2.4 RECURSOS ADICIONAIS

Leituras Adicionais:

- [Learn You a Haskell for Great Good!](#) por Miran Lipovača
- [Functional Programming in Scala](#) por Paul Chiusano e Runar Bjarnason

Ferramentas:

- [Scalaz](#) para programação funcional em Scala

### 3 CONCLUSÃO

A programação concorrente é essencial para desenvolver softwares eficiente e escalável em sistemas modernos. Apesar de sua complexidade, o uso adequado de técnicas e ferramentas concorrentes pode levar a melhorias significativas no desempenho das aplicações. Com prática e estudo, os desenvolvedores podem dominar este paradigma e aproveitar seus benefícios ao máximo.

A programação funcional oferece uma abordagem elegante e matemática para resolver problemas, favorecendo a imutabilidade e a ausência de efeitos colaterais. Isso torna o código mais previsível e fácil de testar, além de simplificar a concorrência. Apesar da curva de aprendizado e das possíveis ineficiências, suas vantagens fazem dela uma escolha poderosa e cada vez mais popular para uma variedade de aplicações.

## 4 REFERÊNCIAS

BAÚ, Gabriel Giuliani. **Vantagens e desvantagens do paradigma da programação concorrente.** Disponível em: <https://www.inf.unioeste.br/~jorge/LINGUAGENS%20DE%20PROGRAMA%C7%C3O/ANO%202011/ARTIGOS%20e%20LINKS%20INTERESSANTES/ARTIGOS/Programa%E7%E3o%20Concorrente%20-%20vantagens%20desvantagens.pdf>. Acesso em: 15 jun. 2024.

MIT. **Concurrency.** Disponível em: <https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/>. Acesso em: 15 jun. 2024.

BAIRESDEV. **Java Concurrency: Master the Art of Multithreading.** Disponível em: <https://www.bairesdev.com/blog/java-concurrency/>. Acesso em: 15 jun. 2024.

MCCURDY, Marcus. **Python Multithreading and Multiprocessing Tutorial.** Disponível em: <https://www.toptal.com/python/beginners-guide-to-concurrency-and-parallelism-in-python>. Acesso em: 15 jun. 2024.

JENKOV, Jakob. **Java Synchronized Blocks.** Disponível em: <https://jenkov.com/tutorials/java-concurrency/synchronized.html>. Acesso em: 15 jun. 2024.

MICROSOFT. **Functional vs. imperative programming.** Disponível em: <https://learn.microsoft.com/pt-br/dotnet/standard/linq/functional-vs-imperative-programming>. Acesso em: 14 jun. 2024.

LOCAWEB. **Programação funcional e POO: veja as diferenças.** Disponível em: <https://www.locaweb.com.br/blog/temas/codigo-aberto/programacao-funcional-e-poo-veja-as-diferencas/>. Acesso em: 14 jun. 2024.

GONÇALVES, Ronaldo. **História do Lisp: abra os olhos para a programação funcional**. Disponível em: <https://imasters.com.br/desenvolvimento/historia-lisp-abra-os-olhos-para-programacao-funcional>. Acesso em: 14 jun. 2024.

SILVA, Romildo Martins. **Programação Funcional com OCaml**. Disponível em: <http://www.decom.ufop.br/romildo/2015-1/bcc222/slides/02-ocaml.pdf>. Acesso em: 14 jun. 2024.

NUBANK. **15 anos de Clojure: uma retrospectiva da linguagem de programação**. Disponível em: <https://blog.nubank.com.br/15-anos-de-clojure-uma-retrospectiva-da-linguagem-de-programacao/>. Acesso em: 14 jun. 2024.

DON, Syme. **The early history of F#**. Disponível em: <https://www.microsoft.com/en-us/research/publication/the-early-history-of-f/>. Acesso em: 14 jun. 2024.

COSTA, Sérgio. **Paradigma funcional**. Disponível em: <https://sergiocosta.medium.com/paradigma-funcional-3194924a8d20>. Acesso em: 14 jun. 2024.