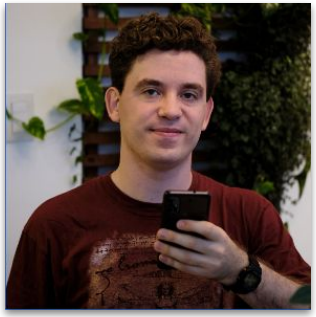
Project Loom

A Era do Java Reativo acabou

Lucas Rogério Masotti



Lucas Rogério Masotti

- 12+ anos de experiência, principalmente backend e setor financeiro
- Staff Engineer Meios de Pagamentos - Sicredi
- Vice coordenador do ARGU
- Pós UX, MBA Gestão Marketing e pós Gestão de Negócios
- **Arquitetura, Design Organizacional e Diminuição de Complexidade Acidental**

Quem aqui utiliza WebFlux?

LUROMA

+ PERFORMANCE
+ ESCALABILIDADE
+ RESILIÊNCIA

LUROMA

+ PERFORMANCE
+ ESCALABILIDADE
+ RESILIÊNCIA
- **CUSTO**

Qual a pegadinha?

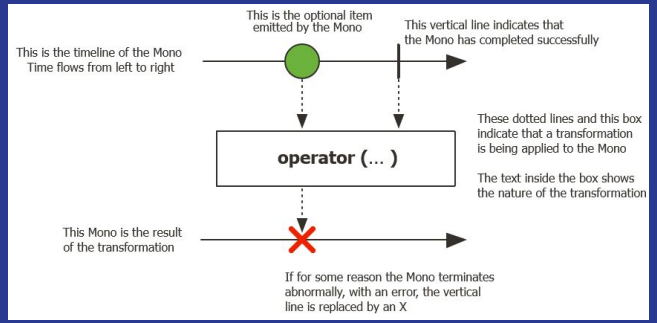
```java
@PostMapping("/transferencias")
@ResponseStatus(HttpStatus.CREATED)
public TransferenciaResponse transferir(@RequestBody TransferenciaRequest request) {

    validador.validar(request);

    if (request.valor().compareTo(LIMITE_ANTIFRAUDE) > 0) {
        var analise = antifraudeService.analisar(request);
        if (!analise.aprovada()) {
            throw new TransferenciaRecusadaException(request.id());
        }
    }

    if (!favorecidoService.existe(request.favorecido())) {
        favorecidoService.cadastrar(request.favorecido());
    }

    var resultado = transferenciaService.efetivar(request);
    return TransferenciaResponse.de(resultado);
}
```

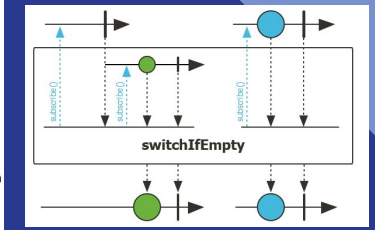
```
@PostMapping("/transferencias")
public Mono<ResponseEntity<TransferenciaResponse>> transferir(
        @RequestBody Mono<TransferenciaRequest> requestMono) {

    return requestMono
        .flatMap(request -> validador.validar(request)
            .onErrorMap(ValidacaoException.class,
                e -> new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage()))
            .thenReturn(request))
        .flatMap(request -> {
            // "if valor > limite, chama antifraude" — não dá pra usar if com
            // o resultado, porque o resultado ainda não existe: só o Mono dele
            if (request.valor().compareTo(LIMITE_ANTIFRAUDE) > 0) {
                return antifraudeService.analisar(request)
                    .filter(AnaliseAntifraude::aprovada)
                    .switchIfEmpty(Mono.error(
                        new TransferenciaRecusadaException(request.id())))
                    .thenReturn(request);
            }
            return Mono.just(request);
        })
```

```
        .flatMap(request ->
            // "if favorecido novo, cadastra" — o boolean vem de outro serviço,
            // então o if vira flatMap com ternário devolvendo Mono
            favorecidoService.existe(request.favorecido())
                .flatMap(existe -> existe
                    ? Mono.just(request)
                    : favorecidoService.cadastrar(request.favorecido())
                        .thenReturn(request)))
        .flatMap(request -> transferenciaService.efetivar(request))
        .map(resultado -> ResponseEntity
            .status(HttpStatus.CREATED)
            .body(TransferenciaResponse.de(resultado)))
        .onErrorResume(TransferenciaRecusadaException.class,
            e -> Mono.just(ResponseEntity
                .unprocessableEntity()
                .body(TransferenciaResponse.recusada(e.getId()))));
}
```

```java
@PostMapping("/transferencias")
public Mono<ResponseEntity<TransferenciaResponse>> transferir(
        @RequestBody Mono<TransferenciaRequest> requestMono) {

    return requestMono
        .flatMap(request -> validador.validar(request)
            .onErrorMap(ValidacaoException.class,
                e -> new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage()))
            .thenReturn(request))
        .flatMap(request -> {
            // "if valor > limite, chama antifraude" — não dá pra usar if com
            // o resultado, porque o resultado ainda não existe: só o Mono dele
            if (request.valor().compareTo(LIMITE_ANTIFRAUDE) > 0) {
                return antifraudeService.analisar(request)
                    .filter(AnaliseAntifraude::aprovada)
                    .switchIfEmpty(Mono.error(
                        new TransferenciaRecusadaException(request.id())))
                    .thenReturn(request);
            }
            return Mono.just(request);
        })
        .flatMap(request ->
            // "if favorecido novo, cadastra" — o boolean vem de outro serviço,
            // então o if vira flatMap com ternário devolvendo Mono
            favorecidoService.existe(request.favorecido())
                .flatMap(existe -> existe
                    ? Mono.just(request)
                    : favorecidoService.cadastrar(request.favorecido())
                        .thenReturn(request)))
        .flatMap(request -> transferenciaService.efetivar(request))
        .map(resultado -> ResponseEntity
            .status(HttpStatus.CREATED)
            .body(TransferenciaResponse.de(resultado)))
        .onErrorResume(TransferenciaRecusadaException.class,
            e -> Mono.just(ResponseEntity
                .unprocessableEntity()
                .body(TransferenciaResponse.recusada(e.getId()))));
}
```

Mono? Flux? FlatMap? switchIfEmpty?





LUROMA

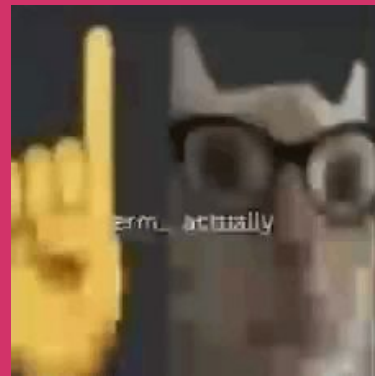
Vale a pena?

Com as virtual threads, **não.**

Vale a pena?

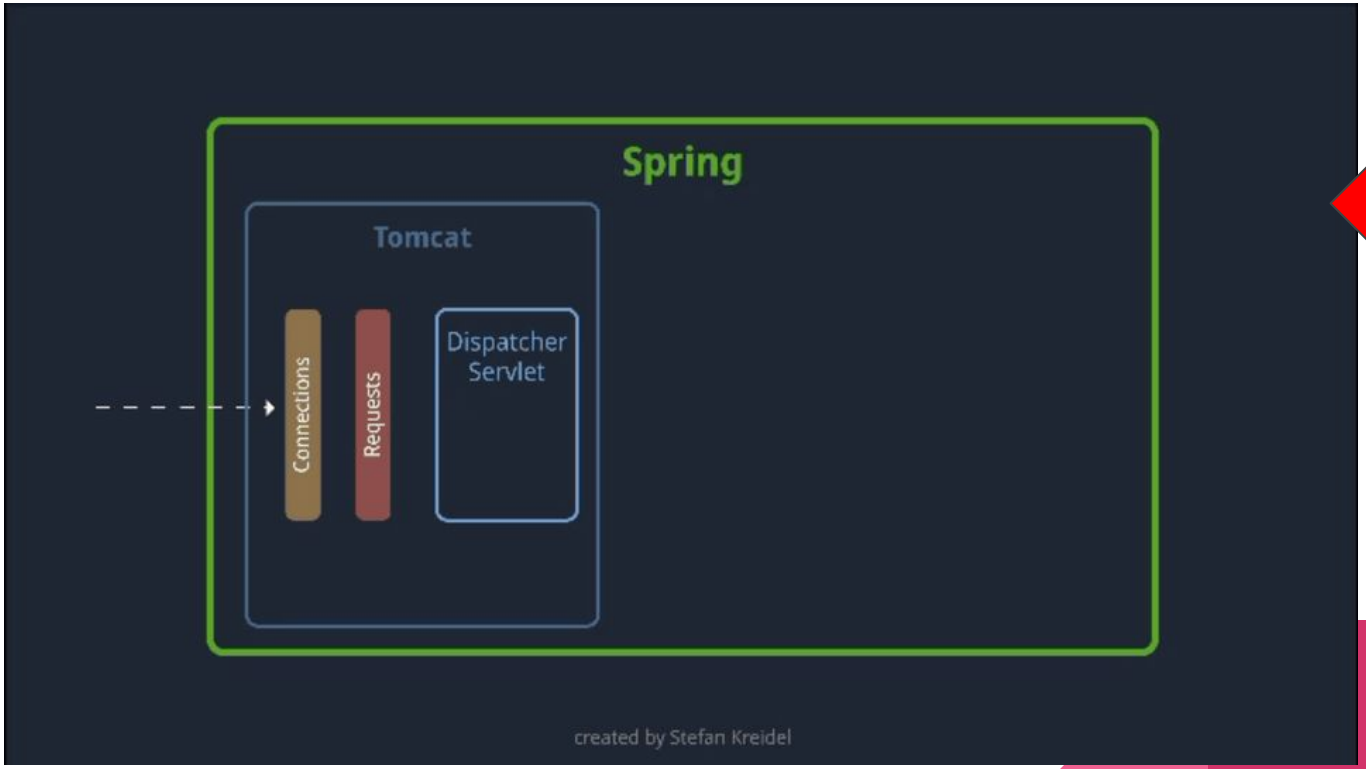
Com as virtual threads, **não.**

Em alguns cenários ainda pode fazer, por recursos como o backpressure, streaming de dados, múltiplas fontes assíncronas etc.



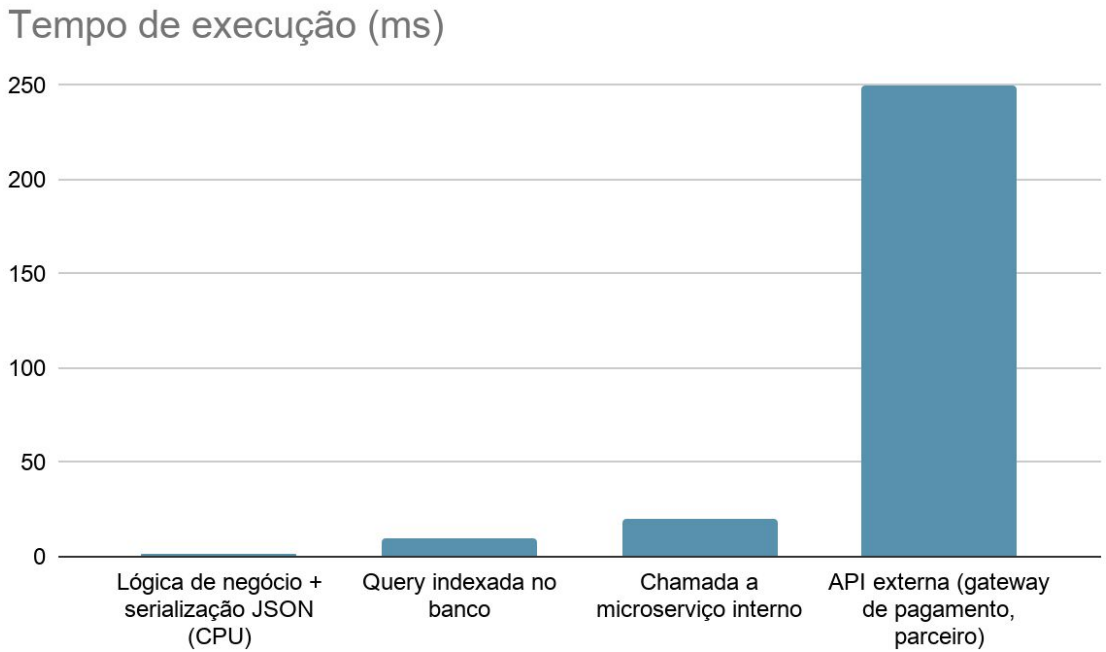
Por que o reativo surgiu?

Thread-per-request

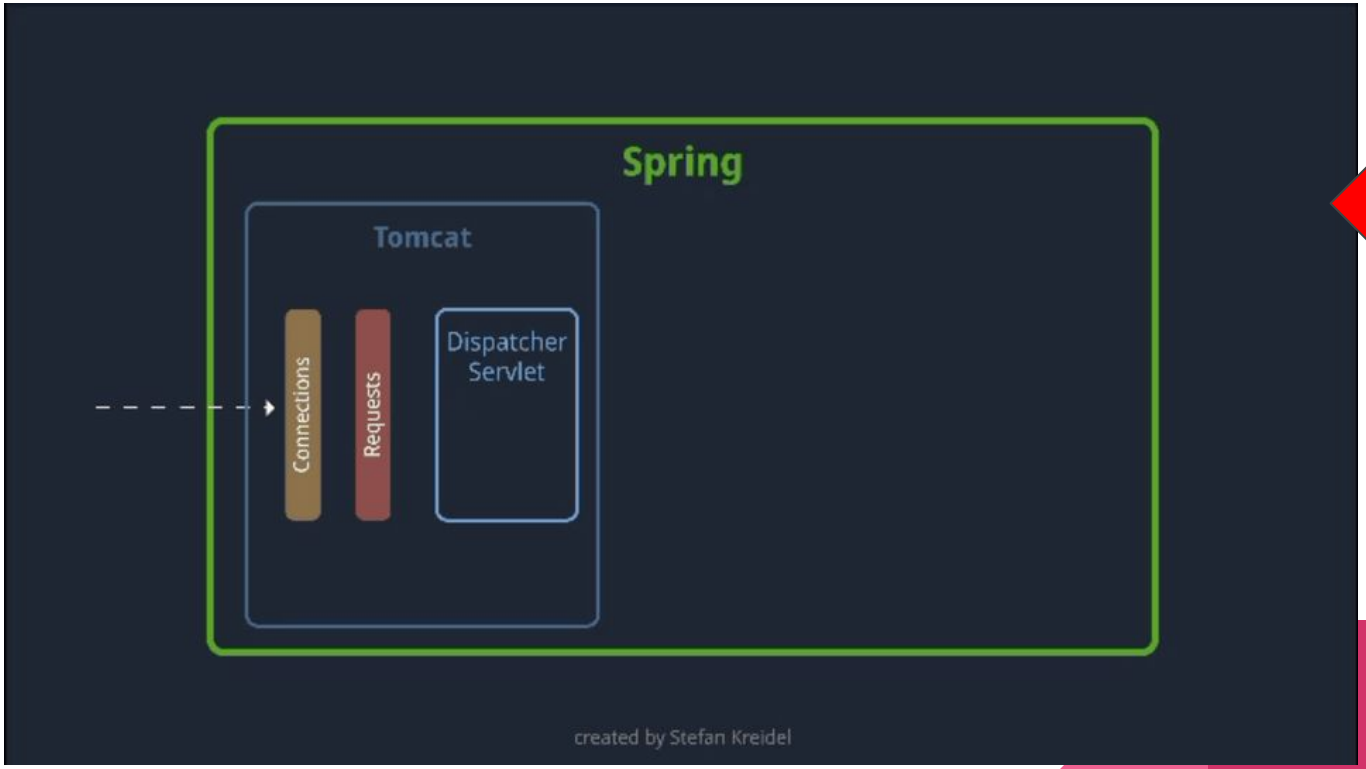


https://www.stefankreidel.io/blog/spring-webmvc-servlet-threading

Em aplicações web, a maior parte do tempo é blocking I/O



Thread-per-request



https://www.stefankreidel.io/blog/spring-webmvc-servlet-threading

MODELO DE THREADS DO JAVA

Cada Thread do Java é uma thread do sistema operacional

✕ **limite do SO** `java.lang.OutOfMemoryError: unable to create native thread`

Memória por thread

1 MB

de stack reservada — 10.000 threads ≈ 10 GB

Custo de criação

~100 µs

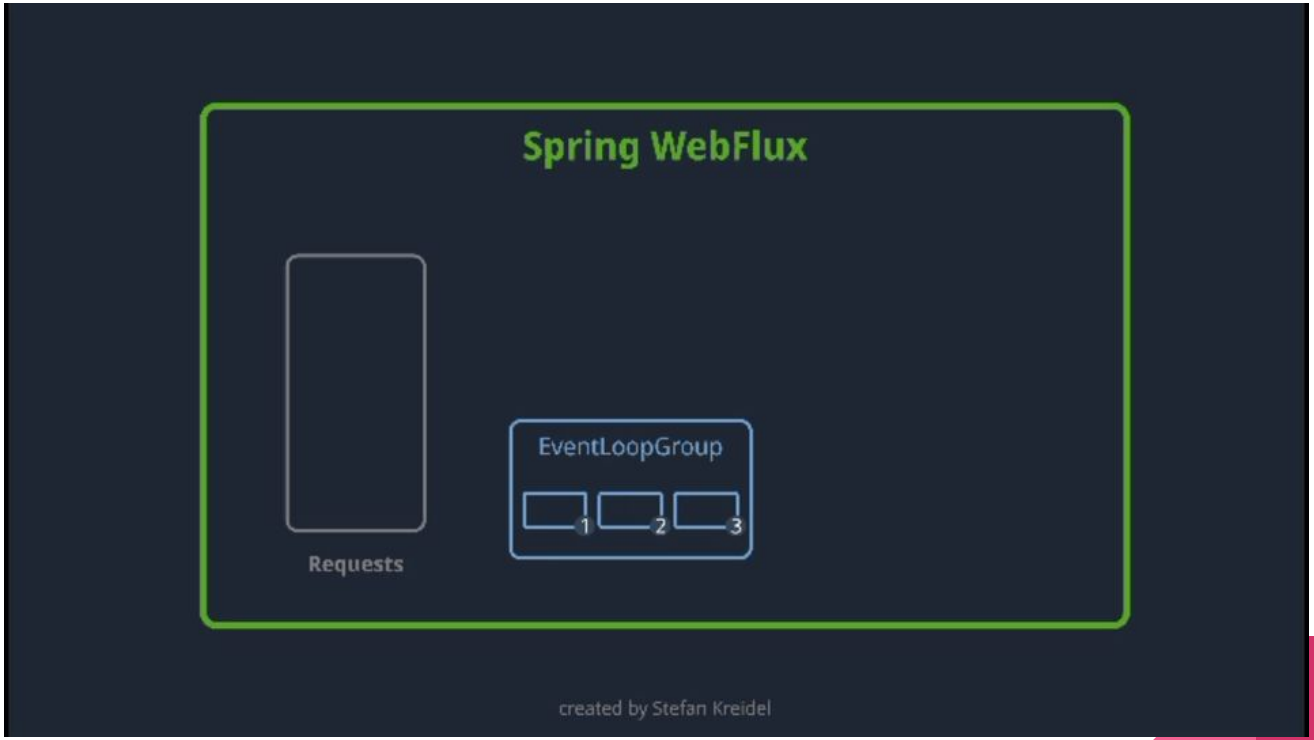
syscall + stack + registro no scheduler — por isso existem pools

Zona de risco

~10 mil

threads por JVM antes de degradar ou estourar (ulimit, max_map_count)

Reativo



https://www.stefankreidel.io/blog/spring-webflux

WebFlux: poucas threads, nenhuma esperando

Event loops (Reactor + Netty)

Event loops

≈ 1 por core

8 threads atendem dezenas de milhares de conexões — a concorrência deixa de depender de threads

Threads esperando I/O

0

a espera vira um registro no epoll + callback — sem 1 MB de stack parado por requisição

Basta

1 bloqueio

um JDBC ou sleep no event loop trava todas as requisições daquele loop — a regra é nunca bloquear

LUROMA

A que custo?

- Curva alta de aprendizagem (Mono/Flux)
- Stack trace e debug difíceis
- Risco de chamadas bloqueantes
- Custo de manutenção
- Dificuldade de encontrar mão de obra qualificada

A partir de quantos TPS o bloqueante quebra?

TPS máximo = threads ÷ latência

Lei de Little (L = λ·W)

	pool 200 Tomcat default	pool 2.000 ≈ 2 GB de stacks	pool 5.000
50 ms latência boa	4.000	40.000	100.000
100 ms latência típica	2.000	20.000	50.000
500 ms API externa lenta	400	4.000	10.000

teto de TPS do modelo bloqueante — mais escuro = teto mais alto

A dor precoce

400 TPS

pool default + dependência de 500 ms — o nicho onde reativo fazia sentido (gateways, BFFs, conexões longas)

A CPU acaba antes

~8.000 TPS

2 ms de CPU/req × 16 cores — nenhum modelo de programação cria CPU

O banco acaba antes

10

conexões no pool default do HikariCP — reativo só muda a fila de lugar

A maioria dos sistemas corporativos roda a centenas ou poucos milhares de TPS — dentro do teto até com o pool default.

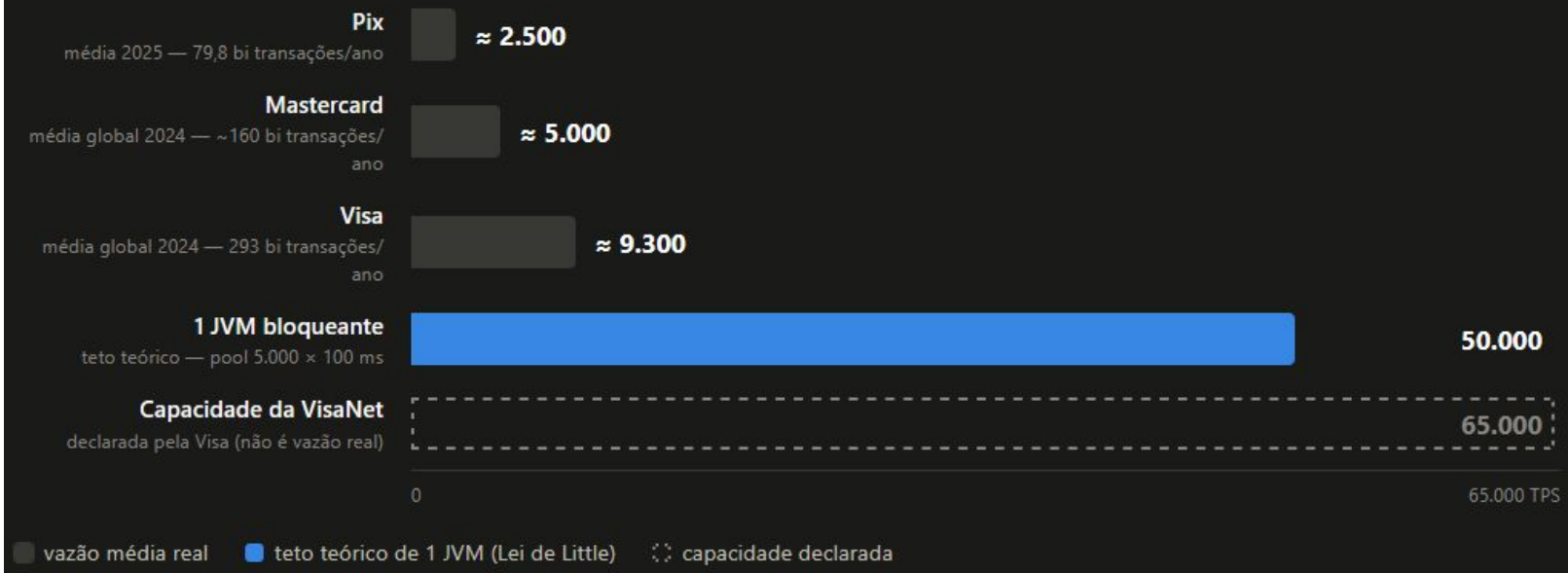
COLOCANDO O TETO EM PERSPECTIVA

E quanto processa a Visa?

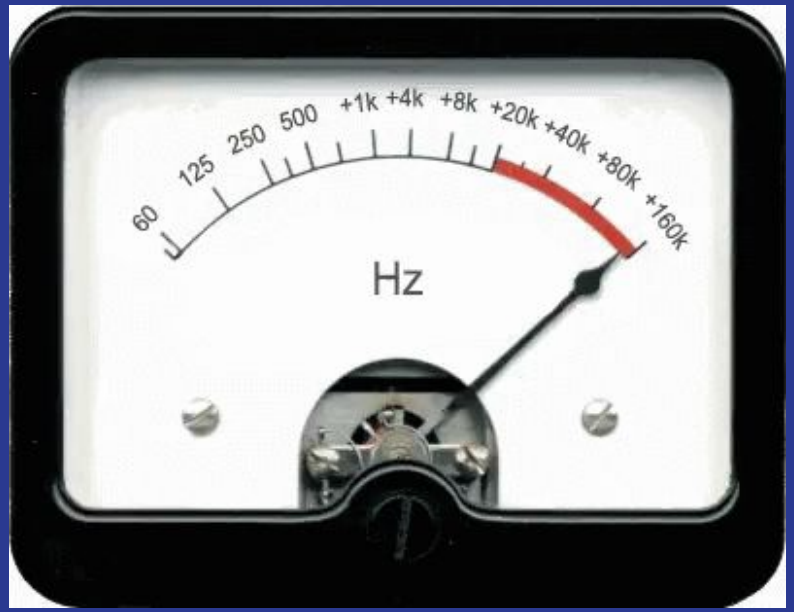
Vazão média global dos maiores sistemas de pagamento vs o teto teórico de uma única JVM bloqueante

5×

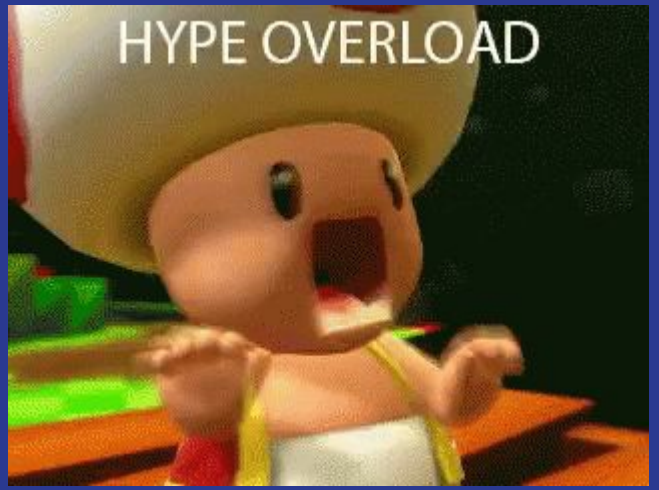
a média global da Visa cabe **cinco vezes** no teto de uma única JVM bloqueante com pool de 5.000 threads e 100 ms de latência



Será que no seu contexto realmente era necessário?



Medidor de Hype



2018

As of 5.0 the `RestTemplate` is in maintenance mode, with only minor requests for changes and bugs to be accepted going forward. Please, consider using the WebClient which offers a more modern API and supports sync, async, and streaming scenarios.

org.springframework.web.client

Class RestTemplate

java.lang.Object
- org.springframework.http.client.support.HttpAccessor
  - org.springframework.http.client.support.InterceptingHttpAccessor
    - org.springframework.web.client.RestTemplate

All Implemented Interfaces:

RestOperations

```
public class RestTemplate
extends InterceptingHttpAccessor
implements RestOperations
```

Synchronous client to perform HTTP requests, exposing a simple, template method API over underlying HTTP client libraries such as the JDK `HttpURLConnection`, Apache HttpComponents, and others. RestTemplate offers templates for common scenarios by HTTP method, in addition to the generalized `exchange` and `execute` methods that support of less frequent cases.

RestTemplate is typically used as a shared component. However, its configuration does not support concurrent modification, and as such its configuration is typically prepared on startup. If necessary, you can create multiple, differently configured RestTemplate instances on startup. Such instances may use the same the underlying `ClientHttpRequestFactory` if they need to share HTTP client resources.

NOTE: As of 5.0 this class is in maintenance mode, with only minor requests for changes and bugs to be accepted going forward. Please, consider using the `org.springframework.web.reactive.client.WebClient` which has a more modern API and supports sync, async, and streaming scenarios.

NOTE: As of 5.0 this class is in maintenance mode, with only minor requests for changes and bugs to be accepted going forward. Please, consider using the `org.springframework.web.reactive.client.WebClient` which has a more modern API and supports sync, async, and streaming scenarios.

https://docs.spring.io/spring-framework/docs/5.3.x/reference/html/integration.html#rest-resttemplate

Reativo como plano B

“Para um programador comum, a **coisa mais simples é pensar em escrever e analisar código sequencial**.
Isso é muito mais fácil, eles estão acostumados a isso e existem ferramentas para tal.

Portanto, **essa deve ser o plano A**. Fazer **algo fora do padrão deve ser o plano B**.“

Project Loom e Virtual Threads

JEP 444: Virtual Threads

Author	Ron Pressler & Alan Bateman
Owner	Alan Bateman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	21
Component	core-libs
Discussion	loom dash dev at openjdk dot org
Relates to	JEP 436: Virtual Threads (Second Preview) JEP 491: Synchronize Virtual Threads without Pinning
Reviewed by	Alex Buckley
Endorsed by	Brian Goetz
Created	2023/03/06 18:00
Updated	2025/10/30 21:01
Issue	8303683

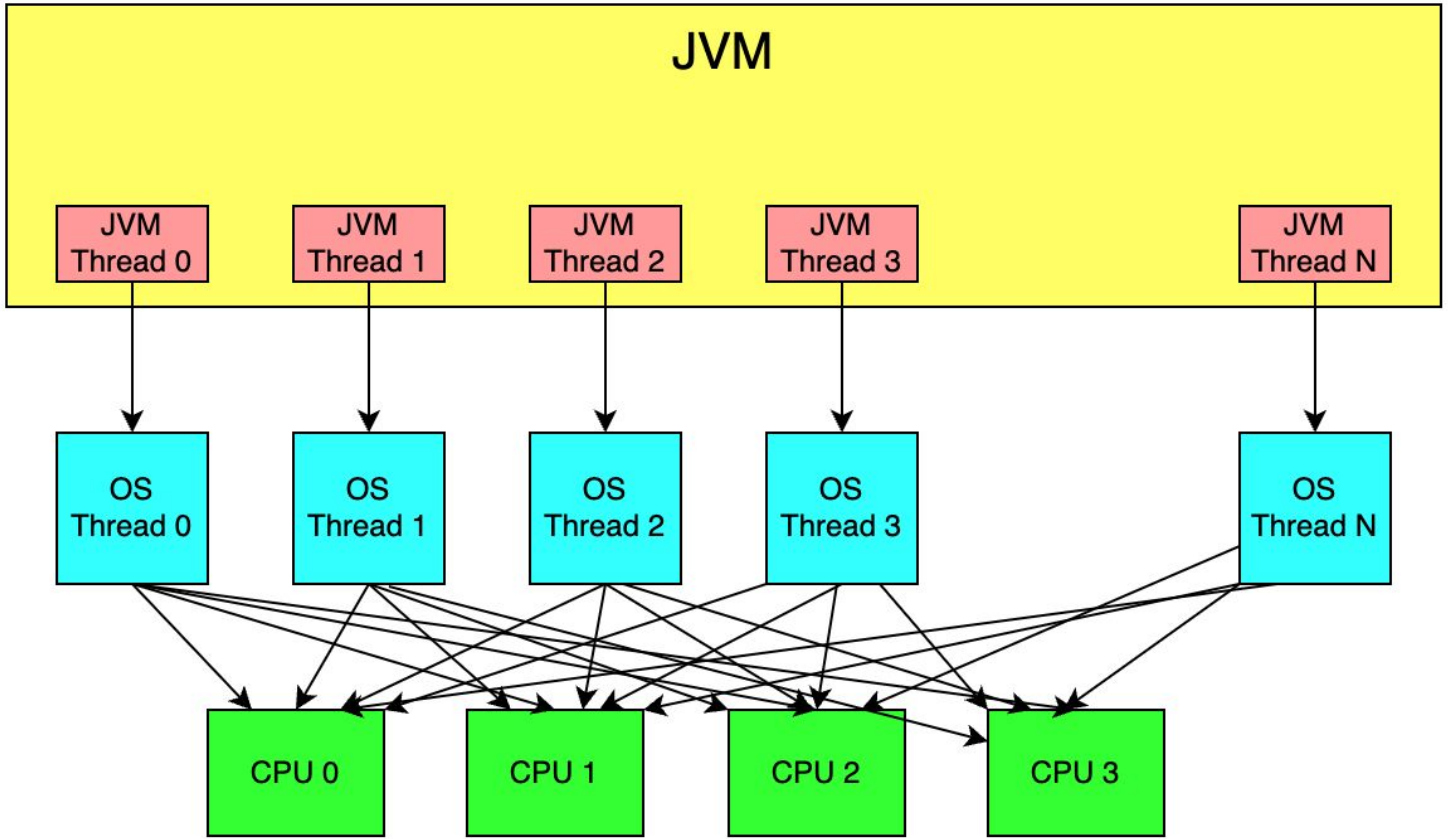
Summary

Introduce *virtual threads* to the Java Platform. Virtual threads are lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.

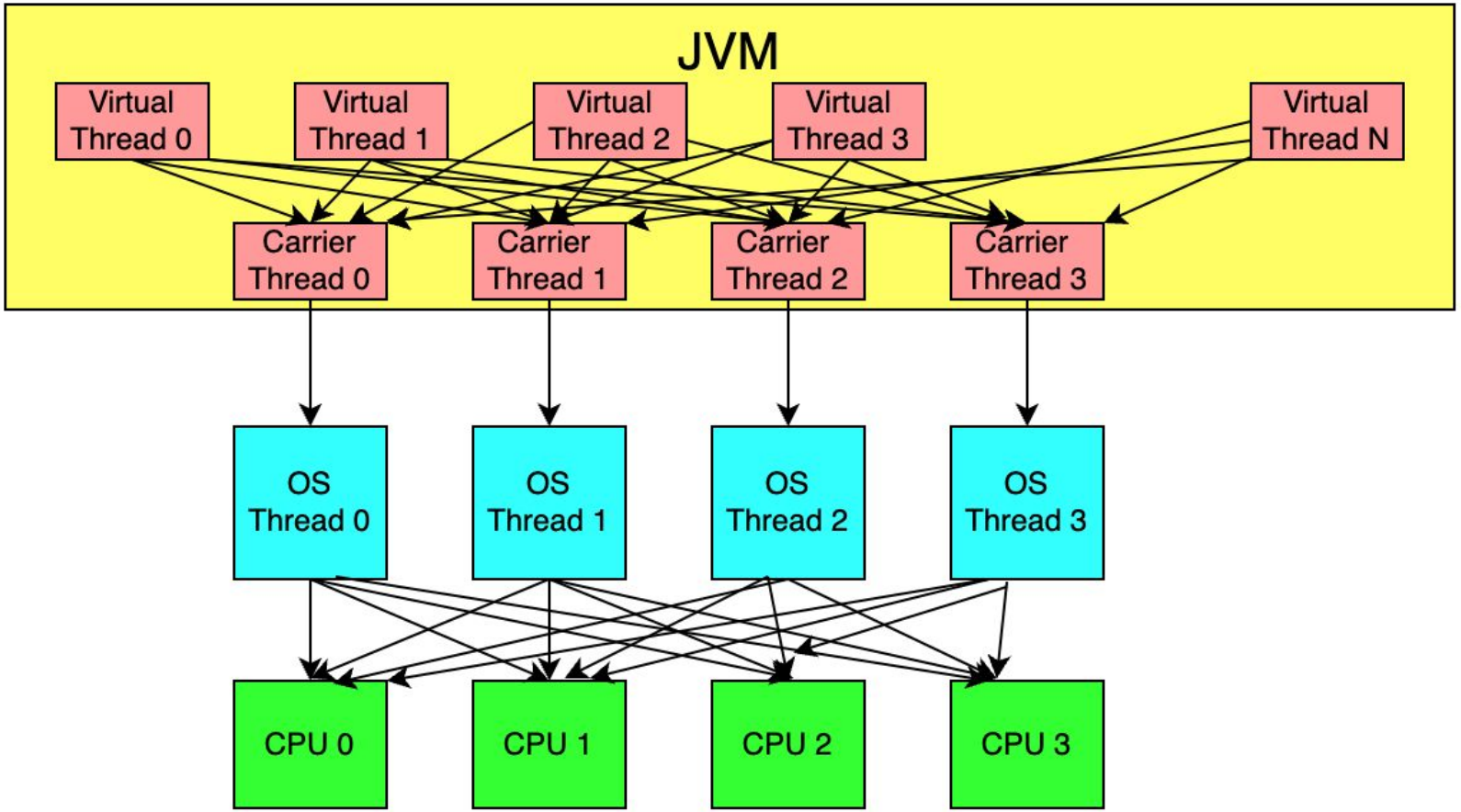
Código síncrono
e imperativo
sem sacrificar
escalabilidade

JDK Enhancement Proposal 444: Virtual Threads

- Permitir que aplicações de servidor escritas no estilo simples de uma thread por requisição **sejam escaláveis com utilização de hardware quase ideal.**
- Permitir que o código existente que utiliza a API java.lang.Thread adote threads virtuais **com alterações mínimas.**
- Permitir a **fácil resolução de problemas, depuração** e criação de perfis de threads virtuais com as ferramentas JDK existentes.



https://dzone.com/articles/java-virtual-threads-and-scaling



https://dzone.com/articles/java-virtual-threads-and-scaling

MEMÓRIA POR THREAD

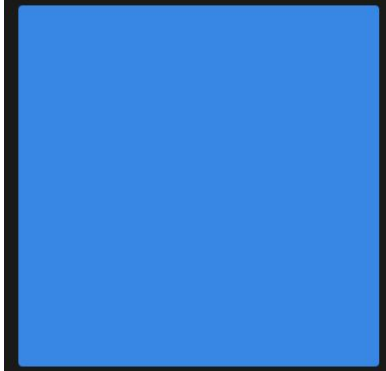
1 thread = 1 MB parado

Área proporcional à memória de stack — escala real (1000 : 1)

Platform thread

1 MB

stack reservada por thread do SO



Virtual thread

~1 KB

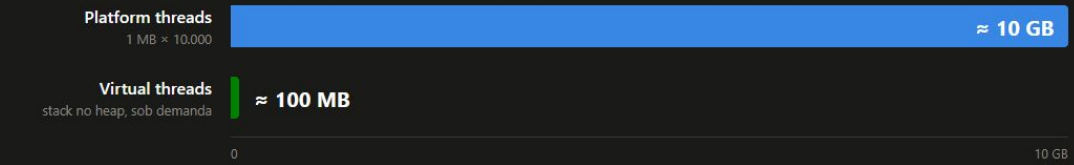
stack inicial no heap, cresce sob demanda

Fora da JVM ainda somam ~16 KB de stack de kernel e estruturas do scheduler por thread de SO.

ESCALANDO O THREAD-PER-REQUEST

10.000 requisições concorrentes

Memória de stacks necessária para segurar 10 mil requisições ao mesmo tempo



▸ Números e fontes deste card

Embracing Virtual Threads

ENGINEERING | MARK PALUCH | OCTOBER 11, 2022 | 5 MIN READ | 23 COMMENTS

Project Loom has made it into the JDK through JEP 425. It's available since Java 19 in September 2022 as a preview feature. Its goal is to dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.

Where Virtual Threads make sense

This makes lightweight Virtual Threads an exciting approach for application developers and the Spring Framework. Past years indicated a trend towards applications that communicate over the network with each other. Many applications make use of data stores, message brokers, and remote services. I/O-intensive applications are the primary ones that benefit from Virtual Threads if they were built to use blocking I/O facilities such as `InputStream` and synchronous HTTP, database, and message broker clients. Running such workloads on Virtual Threads helps reduce the memory footprint compared to Platform Threads and in certain situations, Virtual Threads can increase concurrency.

Higher concurrency can be achieved if the system has additional resources necessary for concurrency. Specifically, these are:

1. Available connections in a connection pool
2. Sufficient memory to serve the increased load
3. Unused CPU time

Use of Virtual Threads clearly is not limited to the direct reduction of memory footprints or an increase in concurrency. The introduction of Virtual Threads also prompts a broader revisit of decisions made for a runtime when only Platform Threads were available.

spring.threads.virtual.enabled=true


NATURE IS HEALING

Conclusões

- Cuidado com o hype!
- Use o reativo como plano B, quando necessário
- *Premature optimization is the root of all evil*
- **Simplicidade como norteador**
- Muito cuidado com a complexidade acidental