



Java Microbenchmark Harness - JMH

Do Básico ao Avançado

No mundo do desenvolvimento de software, a otimização de código é um desafio constante. Pequenos ajustes podem gerar grandes impactos na performance de sistemas de alto desempenho.

No entanto, medir corretamente o tempo de execução de trechos de código na JVM (Java Virtual Machine) pode ser mais complicado do que parece.

A JVM possui otimizações internas, como *Just-In-Time Compilation* (JIT), *Garbage Collection* (GC), e outros mecanismos que podem distorcer medições simplistas com `System.nanoTime()`.

Para solucionar esse problema, a equipe do OpenJDK desenvolveu o **JMH (Java Microbenchmark Harness)**, uma ferramenta robusta para criar **microbenchmarks confiáveis**.

O JMH ajuda a medir com precisão a performance de métodos e algoritmos, protegendo contra otimizações enganosas, aquecimento da JVM (*warm-up*), e concorrência.

Neste artigo, exploraremos o JMH desde a configuração inicial até o uso avançado, incluindo **testes concorrentes, análise de throughput e melhores práticas para benchmarking eficiente**.

1. Configuração do JMH no Projeto

Para utilizar o JMH, você pode configurá-lo via **Maven** ou **Gradle**.

1.1 Dependências com Maven

Adicione as seguintes dependências ao seu `pom.xml`:

```
xml
<dependencies>
  <dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>1.38</version>
  </dependency>
  <dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
```



```
<version>1.38</version>
<scope>provided</scope>
</dependency>
</dependencies>
```

1.2 Dependências com Gradle

Se estiver usando **Gradle**, adicione:

```
gradle
dependencies {
    implementation 'org.openjdk.jmh:jmh-core:1.38'
    annotationProcessor 'org.openjdk.jmh:jmh-generator-annprocess:1.38'
}
```

2. Conceitos Fundamentais do JMH

O **JMH** usa anotações para definir benchmarks. Vamos explorar as principais:

- **@Benchmark** → Indica o método que será testado.
- **@BenchmarkMode** → Define o modo de medição:
 - Throughput: Mede o número de operações por segundo.
 - AverageTime: Mede o tempo médio por operação.
 - SampleTime: Mede o tempo de execução baseado em amostras.
 - SingleShotTime: Mede o tempo de uma única execução.
- **@OutputTimeUnit** → Define a unidade de tempo dos resultados (nanossegundos, microssegundos, milissegundos).
- **@State(Scope.Thread)** → Controla o escopo da instância do benchmark.
- **@Warmup** → Define execuções iniciais para compensar o *warm-up* da JVM.
- **@Fork** → Define quantas vezes o benchmark será executado em processos separados.
- **@Threads** → Define quantas threads serão usadas para execução concorrente.

3. Criando um Benchmark Simples

Agora que entendemos os conceitos básicos, vamos criar nosso primeiro benchmark para medir a execução de um **loop de soma**:

```
import org.openjdk.jmh.annotations.*;

import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime) // Mede tempo médio de execução
@OutputTimeUnit(TimeUnit.NANOSECONDS) // Unidade: nanossegundos
@State(Scope.Thread) // Cada thread terá sua própria instância do benchmark
public class SimpleBenchmark {

    @Benchmark
    public int calcularSoma() {
        int soma = 0;
        for (int i = 0; i < 1000; i++) {
```



```
        soma += i;
    }
    return soma;
}
}
```

Executando o Benchmark

Gere o JAR do benchmark e execute:

```
mvn clean install
java -jar target/benchmarks.jar
```

Saída esperada:

Benchmark	Mode	Cnt	Score	Error	Units
SimpleBenchmark.soma	avgt	10	12.345	± 0.123	ns/op

Isso significa que a operação de soma leva **12.345 nanossegundos por execução**.

4. Modos de Benchmarking no JMH

O **JMH** permite diferentes modos para analisar o desempenho:

4.1 Medindo Throughput

Mede quantas operações são realizadas por unidade de tempo.

```
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
@State(Scope.Thread)
public class ThroughputBenchmark {

    @Benchmark
    public int processarDados() {
        int resultado = 0;
        for (int i = 0; i < 10000; i++) {
            resultado += i;
        }
        return resultado;
    }
}
```

4.2 Medindo Tempo Médio

Mede o tempo médio de cada operação.

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@State(Scope.Thread)
```



```
public class AverageTimeBenchmark {  
  
    @Benchmark  
    public void executarTarefa() {  
        try {  
            Thread.sleep(5); // Simula uma operação I/O  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        }  
    }  
}
```

5. Benchmarks Concorrentes no JMH

Agora, vamos medir o desempenho de um contador compartilhado usando **AtomicInteger** e um contador normal.

```
import java.util.concurrent.atomic.AtomicInteger;  
import org.openjdk.jmh.annotations.*;  
  
import java.util.concurrent.TimeUnit;  
  
@BenchmarkMode({Mode.Throughput})  
@OutputTimeUnit(TimeUnit.MILLISECONDS)  
@State(Scope.Group)  
public class ConcurrencyBenchmark {  
  
    private final AtomicInteger atomicCounter = new AtomicInteger();  
    private int normalCounter = 0;  
  
    @Benchmark  
    @Group("atomic")  
    public int incrementarAtomic() {  
        return atomicCounter.incrementAndGet();  
    }  
  
    @Benchmark  
    @Group("normal")  
    public int incrementarNormal() {  
        return ++normalCounter;  
    }  
}
```

Aqui, podemos comparar **AtomicInteger** com um contador comum para analisar os impactos de contenção e concorrência.



6. Evitando Armadilhas no Benchmarking

Mesmo com o JMH, erros comuns podem prejudicar os resultados:

1. **Dead Code Elimination** → Se o resultado não for usado, o JIT pode removê-lo. Sempre retorne valores no benchmark.
2. **Loop Unrolling** → Pequenos loops podem ser otimizados pelo compilador.
3. **Warm-up** → Sempre utilize `@Warmup` para evitar que otimizações da JVM distorçam os resultados.
4. **Efeito do Garbage Collector (GC)** → O GC pode interferir na medição. Use `-XX:+PrintGC` para monitorá-lo.
5. **Influência de Outros Processos** → Execute benchmarks em uma máquina ociosa para evitar interferências externas.

O **JMH** é uma ferramenta essencial para quem deseja medir a performance de código Java com precisão.

Ele elimina os problemas de medições ingênuas e permite analisar desde pequenas operações até cenários complexos envolvendo concorrência.

Exploramos desde benchmarks básicos até análises avançadas de throughput e concorrência.

Com isso, você pode otimizar código crítico, evitar armadilhas do JIT e garantir que suas aplicações alcancem o melhor desempenho possível.

Se você trabalha com otimização de código ou desenvolvimento de sistemas de alto desempenho, o JMH deve fazer parte do seu arsenal de ferramentas.

Referências

- Documentação oficial: <https://openjdk.org/projects/code-tools/jmh/>
- Código-fonte do JMH: <https://github.com/openjdk/jmh>

EducaCiência FastCode para a comunidade