Ejemplo básico con JMH

JMH es la mejor opción cuando se quiere medir el rendimiento de métodos específicos en Java.

1 Añadir dependencia de JMH (Maven)

```
<dependencies>
  <dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>1.35</version>
  </dependency>
  <dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
    <version>1.35</version>
  </dependency>
</dependencies>
```

2 Ejemplo de benchmark con JMH

Este código compara el rendimiento de tres enfoques diferentes para calcular la suma de una lista de números:

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

import java.util.List;
import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;
import java.util.stream.Collectors;

@BenchmarkMode(BenchmarkMode.AverageTime)  // Medimos el tiempo medio de ejecución
@OutputTimeUnit(TimeUnit.MILLISECONDS)    // La salida se mide en milisegundos
@State(Scope.Thread)                      // Cada hilo tiene su propia instancia del benchmark
public class ListSumBenchmark {
```

```
private static final int SIZE = 1_000_000; // Tamaño de la lista a procesar
private List<Integer> numbers;

@Setup(Level.Iteration) // Se ejecuta antes de cada iteración del benchmark
public void setUp() {
  Random random = new Random();
  numbers = IntStream.range(0, SIZE)
            .map(i -> random.nextInt(100))
            .boxed()
            .collect(Collectors.toList());
}

@Benchmark
public long sumWithForLoop() {
  long sum = 0;
  for (int num : numbers) {
    sum += num;
  }
  return sum;
}

@Benchmark
public long sumWithStream() {
  return numbers.stream().mapToLong(Integer::longValue).sum();
}

@Benchmark
public long sumWithParallelStream() {
  return numbers.parallelStream().mapToLong(Integer::longValue).sum();
}
```

}

Explicación del código

1. **Generamos una lista de 1,000,000 números aleatorios entre 0 y 99** en @Setup(Level.Iteration).
2. **Tres métodos distintos para calcular la suma**:
    - sumWithForLoop(): Usa un for-each, ideal para rendimiento predecible.
    - sumWithStream(): Usa Stream de Java 8.
    - sumWithParallelStream(): Usa parallelStream() para procesamiento concurrente.
3. **Se usa @Benchmark en cada método** para medir el rendimiento de cada implementación.

Compilar y ejecutar el Benchmark

1 Compilar

Ejecuta el siguiente comando:

mvn clean package

2 Ejecutar el benchmark

java -jar target/benchmarks.jar

Podemos usar ManagementFactory para medir: **Uso de CPU**

Memoria usada antes y después de ejecutar cada benchmark

Código actualizado:

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

import java.lang.management.ManagementFactory;
import java.lang.management.ThreadMXBean;
import java.util.List;
import java.util.Random;
import java.util.concurrent.TimeUnit;
```

```
import java.util.stream.Collectors;
import java.util.stream.IntStream;

@BenchmarkMode(BenchmarkMode.AverageTime)  // Modo de medición: tiempo medio
@OutputTimeUnit(TimeUnit.MILLISECONDS)    // Salida en milisegundos
@State(Scope.Thread)                  // Cada hilo tiene su propia instancia del benchmark
public class ListSumBenchmark {

  private static final int SIZE = 1_000_000; // Tamaño de la lista
  private List<Integer> numbers;
  private ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();

  @Setup(Level.Iteration)
  public void setUp() {
    Random random = new Random();
    numbers = IntStream.range(0, SIZE)
        .map(i -> random.nextInt(100))
        .boxed()
        .collect(Collectors.toList());
  }

  private long getCPUTime() {
    return threadMXBean.getCurrentThreadCpuTime(); // Tiempo en nanosegundos
  }

  private long getMemoryUsage() {
    return Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
  }

  @Benchmark
  public void sumWithForLoop() {
```

```
        long cpuBefore = getCPUTime();
        long memBefore = getMemoryUsage();

        long sum = 0;
        for (int num : numbers) {
            sum += num;
        }

        long cpuAfter = getCPUTime();
        long memAfter = getMemoryUsage();

        System.out.println("ForLoop - CPU: " + (cpuAfter - cpuBefore) / 1_000_000 + " ms, Memoria: " + (memAfter - memBefore) / 1024 + " KB");
    }

    @Benchmark
    public void sumWithStream() {
        long cpuBefore = getCPUTime();
        long memBefore = getMemoryUsage();

        long sum = numbers.stream().mapToLong(Integer::longValue).sum();

        long cpuAfter = getCPUTime();
        long memAfter = getMemoryUsage();

        System.out.println("Stream - CPU: " + (cpuAfter - cpuBefore) / 1_000_000 + " ms, Memoria: " + (memAfter - memBefore) / 1024 + " KB");
    }

    @Benchmark
    public void sumWithParallelStream() {
        long cpuBefore = getCPUTime();
```

```
        long memBefore = getMemoryUsage();


        long sum = numbers.parallelStream().mapToLong(Integer::longValue).sum();


        long cpuAfter = getCPUTime();
        long memAfter = getMemoryUsage();


        System.out.println("ParallelStream - CPU: " + (cpuAfter - cpuBefore) / 1_000_000 + "
ms, Memoria: " + (memAfter - memBefore) / 1024 + " KB");
    }
}
```

Explicación del código

1. **Medimos CPU** con getCurrentThreadCpuTime(), obteniendo tiempo de CPU en nanosegundos antes y después del benchmark.
2. **Medimos memoria** con Runtime.getRuntime().totalMemory() - freeMemory(), mostrando el uso en **KB**.
3. **Imprimimos los resultados** dentro del benchmark para tenerlos junto a los tiempos de ejecución de JMH.