《Java 性能优化实战 21 讲》

李国 前京东、陌陌高级架构师

— 拉勾教育出品 —

05 |工具实践：基准测试 JMH 精确测量方法性能

工具实践：基准测试 JMH，精确测量方法性能

测量某段具体代码的性能：写一些统计执行时间的代码

这些代码穿插逻辑中，进行一些简单的计时运算

```
long start = System.currentTimeMillis();
//logic
long cost = System.currentTimeMillis() - start;
System.out.println("Logic cost : " + cost);
```

工具实践：基准测试 JMH，精确测量方法性能

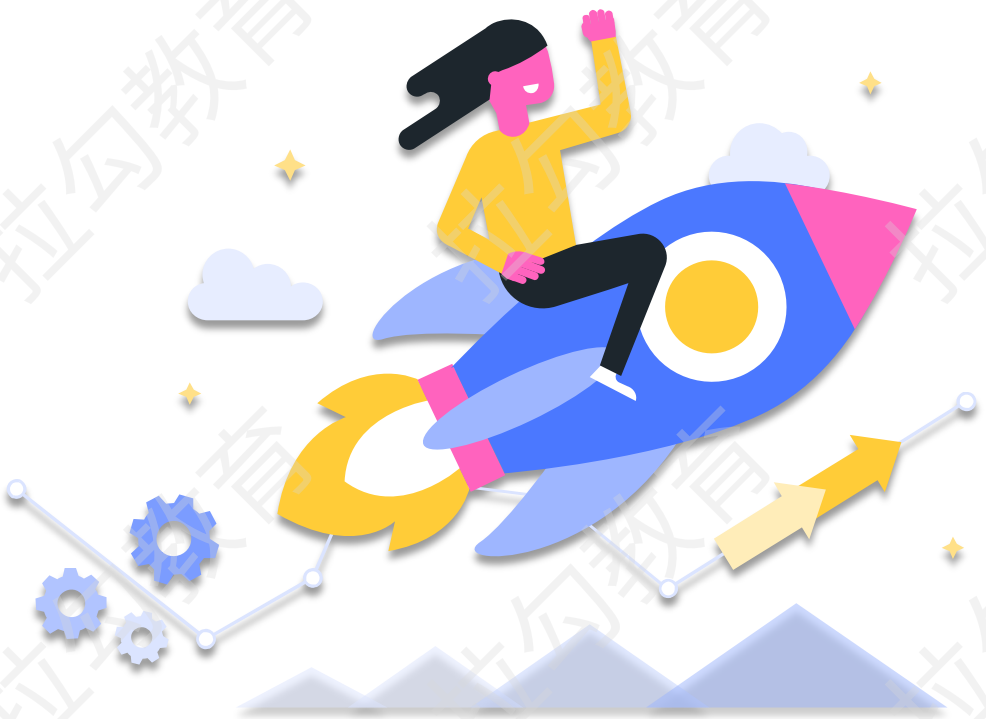
拉勾教育
互联网人实战大学

JVM 在执行时，会对一些代码块，或者一些频繁执行的逻辑，进行 JIT 编译和内联优化

在得到一个稳定的测试结果之前，需要先循环上万次进行预热

评估性能，有很多的指标

JMH—基准测试工具



JMH（the Java Microbenchmark Harness）

基准测试的工具

测量精度非常高，可达纳秒级别

JMH—基准测试工具

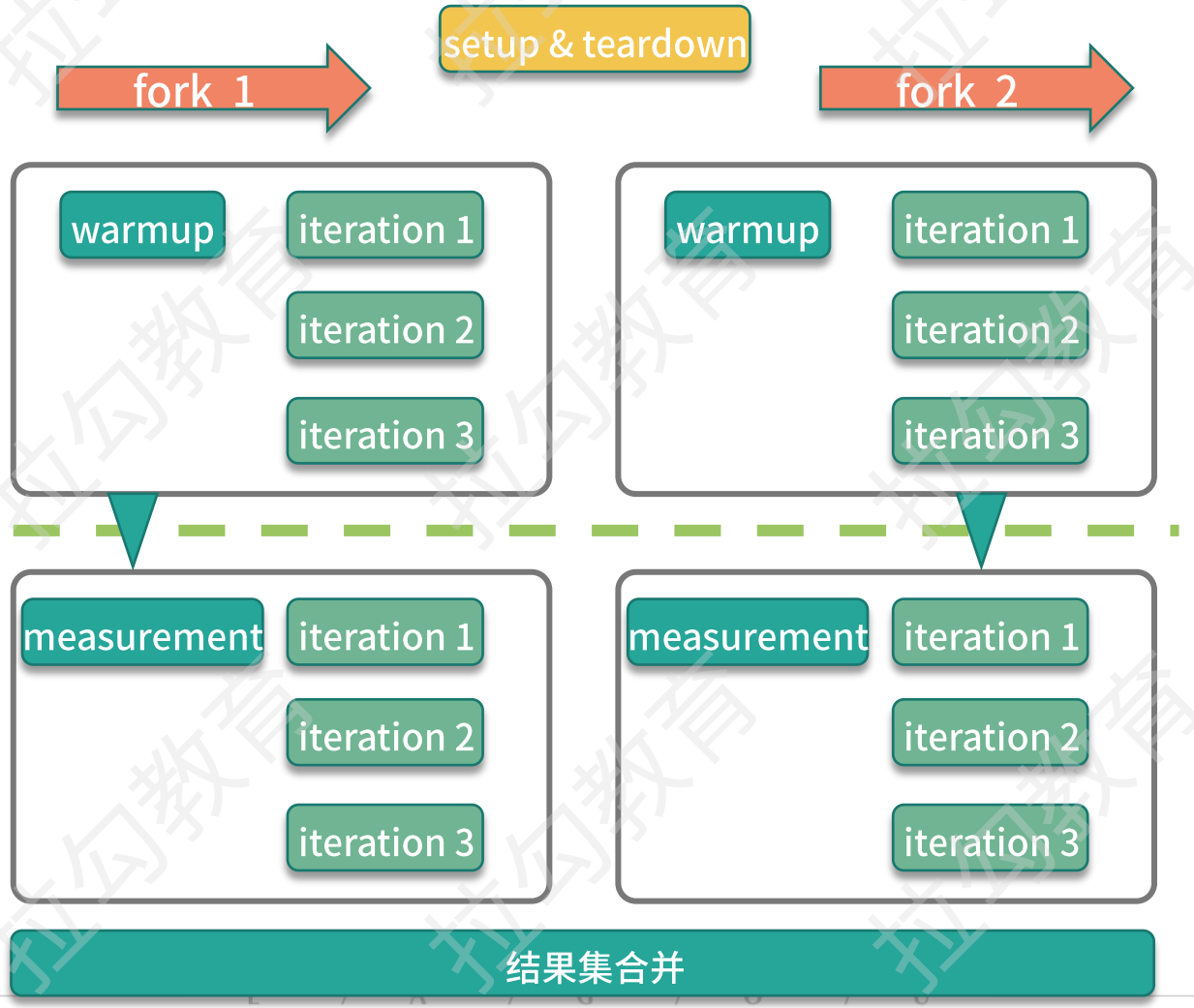

```
<dependencies>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
      <artifactId>jmh-core</artifactId>
      <version>1.23</version>
    </dependency>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
      <artifactId>jmh-generator-annprocess</artifactId>
      <version>1.23</version>
      <scope>provided</scope>
    </dependency>
</dependencies>
```

JMH—基准测试工具

JMH 是一个 jar 包，可以通过**注解**进行一些基础配置

这些配置很多可以通过 main 方法的 OptionsBuilder 进行设置的

JMH—基准测试工具



JMH—基准测试工具


```
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
@Warmup(iterations = 3, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(1)
@Threads(2)
public class BenchmarkTest {
  @Benchmark
  public long shift() {
    long t = 455565655225562L;
    long a = 0;
    for (int i = 0; i < 1000; i++) {
      a = t >> 30;
    }
    return a;
  }

  @Benchmark
  public long div() {
```

JMH—基准测试工具


```
    }

    @Benchmark
    public long div() {
        long t = 455565655225562L;
        long a = 0;
        for (int i = 0; i < 1000; i++) {
            a = t / 1024 / 1024 / 1024;
        }
        return a;
    }

    public static void main(String[] args) throws Exception {
        Options opts = new OptionsBuilder()
                .include(BenchmarkTest.class.getSimpleName())
                .resultFormat(ResultFormatType.JSON)
                .build();
        new Runner(opts).run();
    }
}
```

@Warmup


```
@Warmup(
iterations = 5,
time = 1,
timeUnit = TimeUnit.SECONDS)
```

对代码预热总计 5 秒
（迭代 5 次，每次一秒）

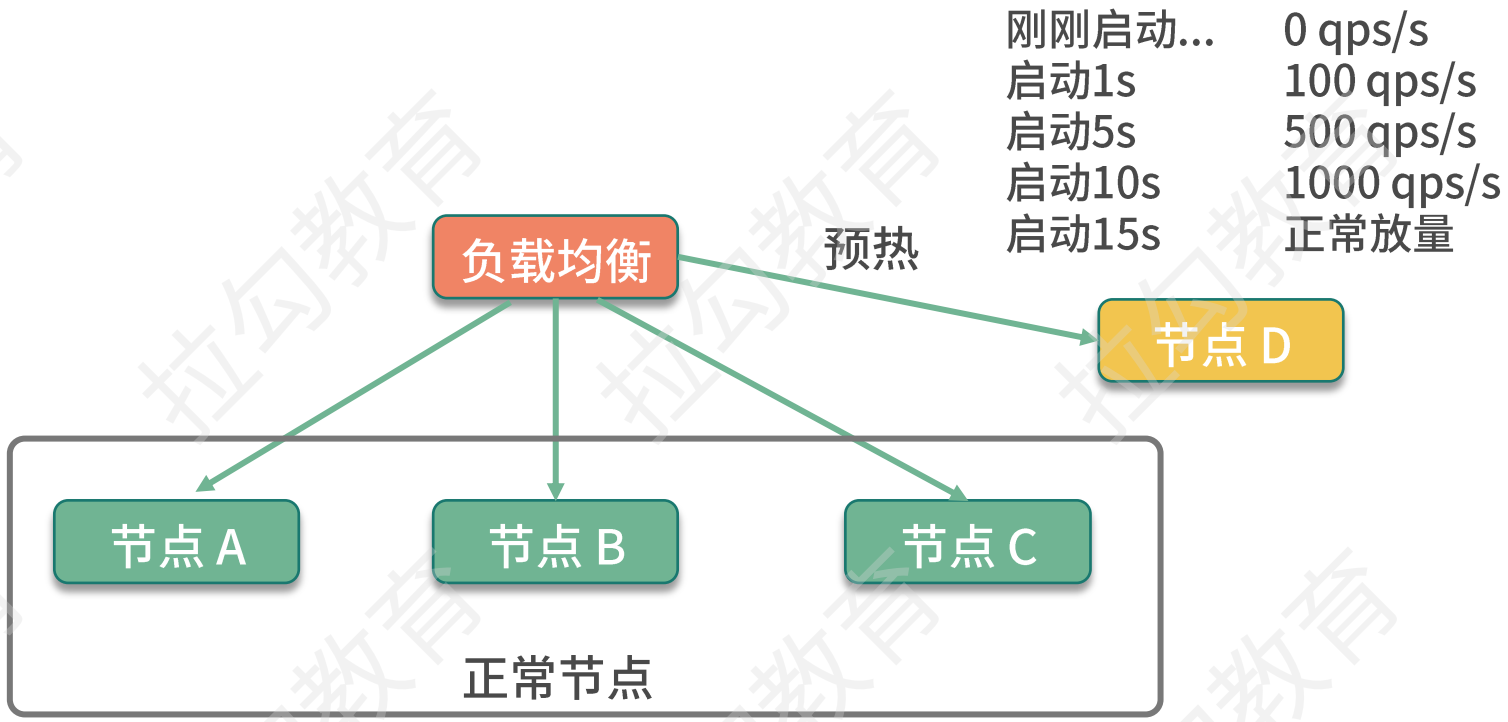
- timeUnit：时间的单位，默认的单位是秒
- iterations：预热阶段的迭代数
- time：每次预热的时间
- batchSize：批处理大小，指定了每次操作调用几次方法

@Warmup

执行的效果

```
# Warmup: 3 iterations, 1 s each
# Warmup Iteration   1: 0.281 ops/ns
# Warmup Iteration   2: 0.376 ops/ns
# Warmup Iteration   3: 0.483 ops/ns
```

@Warmup



@Measurement

样例如下

```
@Measurement(
iterations = 5,
time = 1,
timeUnit = TimeUnit.SECONDS)
```

@Measurement

执行过程

```
# Measurement: 5 iterations, 1 s each
Iteration   1: 1646.000 ns/op
Iteration   2: 1243.000 ns/op
Iteration   3: 1273.000 ns/op
Iteration   4: 1395.000 ns/op
Iteration   5: 1423.000 ns/op
```

@Measurement

在测试时，给机器充足的资源，保持一个稳定的环境

在分析结果时，更加关注不同代码实现方式下的**性能差异**

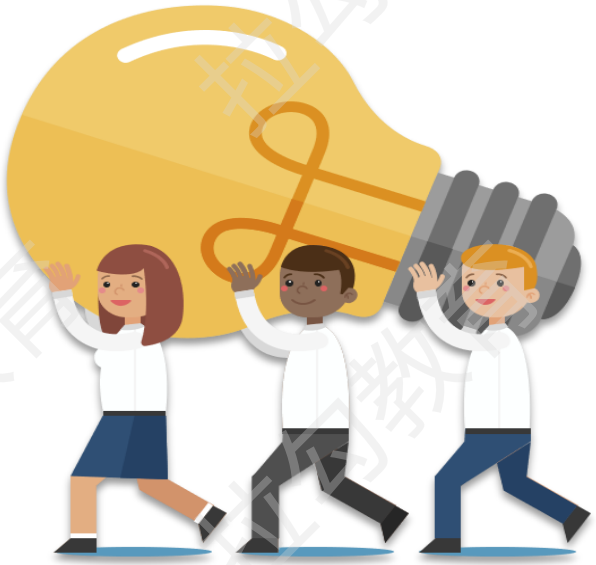
@BenchmarkMode

@BenchmarkMode——指定基准测试类型，对应 Mode 选项，用来修饰类和方法

value，是一个数组，可以配置多个统计维度

@BenchmarkMode({Throughput,Mode.AverageTime})

统计吞吐量和平均执行时间两个指标



@BenchmarkMode

- **Throughput**：整体吞吐量，比如 QPS，单位时间内的调用量等
- **AverageTime**：平均耗时，指的是每次执行的平均时间
- **SampleTime**：随机取样
- **SingleShotTime**：可以测试仅仅一次的性能，比如第一次初始化花了多长时间
- **All**：所有的指标，都算一遍

@BenchmarkMode

大体的执行结果

```
Result "com.github.xjjdog.tuning.BenchmarkTest.shift":
  2.068 ± (99.9%) 0.038 ns/op [Average]
  (min, avg, max) = (2.059, 2.068, 2.083), stdev = 0.010
  CI (99.9%): [2.030, 2.106] (assumes normal distribution)
```

@BenchmarkMode

最终的耗时时间

Benchmark	Mode	Cnt	Score	Error	Units
BenchmarkTest.div	avgt	5	2.072	± 0.053	ns/op
BenchmarkTest.shift	avgt	5	2.068	± 0.038	ns/op

在衡量这些指标时，都有一个时间维度，是通过 @OutputTimeUnit 注解进行配置的

@BenchmarkMode

拉勾教育
互联网人实战大学

@OutputTimeUnit 注解指明了基准测试结果的时间类型，可用于类或者方法上

一般选择秒、毫秒、微秒，纳秒都是针对的速度非常快的方法

@BenchmarkMode

@BenchmarkMode(Mode.Throughput) 和 @OutputTimeUnit(TimeUnit.MILLISECONDS) 进行组合，代表的是**每毫秒的吞吐量**

Benchmark	Mode	Cnt	Score	Error	Units
BenchmarkTest.div	thrpt	5	482999.685 ±	6415.832	ops/ms
BenchmarkTest.shift	thrpt	5	480599.263 ±	20752.609	ops/ms

@Fork

- 一般设置成 1——只使用一个进程进行测试
- 如果这个数字大于 1——启用新的进程进行测试
- 如果设置成 0，程序是在用户的 JVM 进程上运行的

@Fork

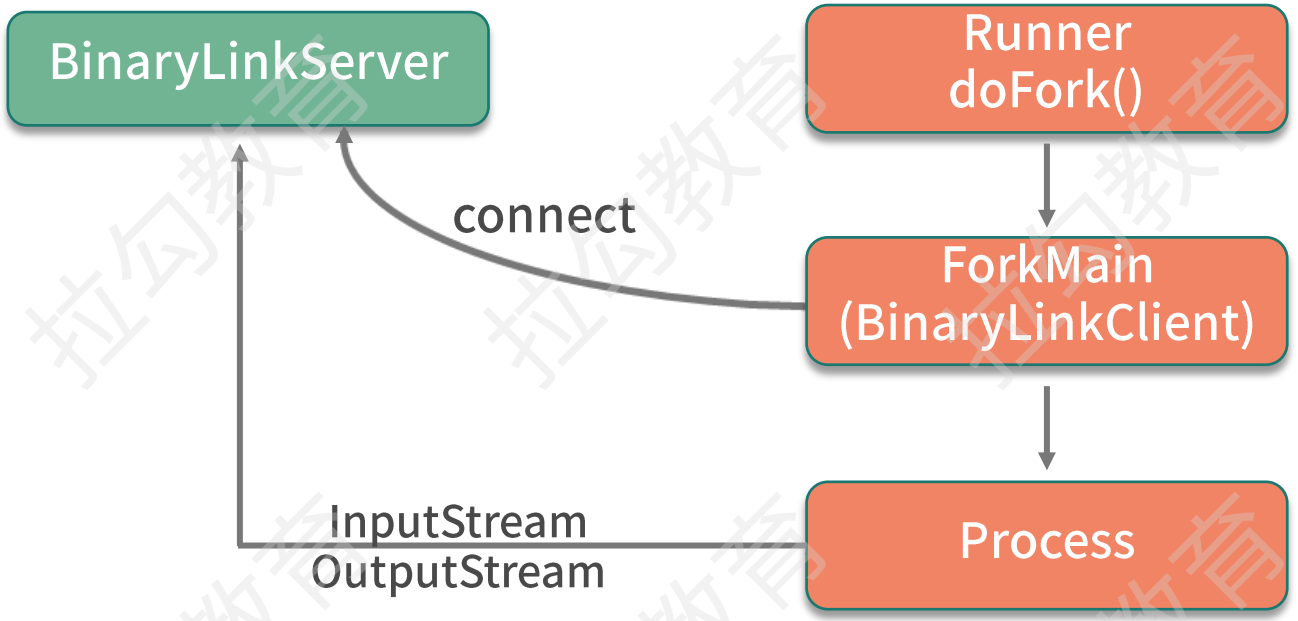

```
# Fork: N/A, test runs in the host VM
# *** WARNING: Non-forked runs may silently omit JVM options,
mess up profilers, disable compiler hints, etc. ***
# *** WARNING: Use non-forked runs only for debugging
purposes, not for actual performance runs. ***
```

@Fork

fork 到底是在进程还是线程环境里运行呢？

每个 fork 进程是单独运行在 Proccess 进程里的

@Fork



@Fork

拉勾教育
互联网人实战大学

jvmArgsAppend——传递一些 JVM 的参数

```
@Fork(value = 3, jvmArgsAppend = {"-Xmx2048m", "-server", "-XX:+AggressiveOpts"})
```

@Threads

拉勾教育
互联网人实战大学

Threads 是面向线程的

指定 Threads 后，将会开启并行测试

如果配置了 Threads.MAX ，则使用和处理机器核数相同的线程数

@Group

@Group 注解只能加在方法上，用来把测试方法进行归类

@GroupThreads 注解——在归类的基础上，再进行一些线程方面的设置

@State

@State 指定了在类中变量的作用范围，用于声明某个类是一个“状态”

Scope 参数——表示该状态的共享范围

@State 注解必须加在类上，否则提示无法运行

@State

Scope 有三种值：

- **Benchmark**：变量的作用范围是某个基准测试类
- **Thread**：每个线程一份副本，如果配置了 Threads 注解，则每个 Thread 都拥有一份变量
- **Group**：在同一个 Group 里，将会共享同一个变量实例

@State

在 JMHSample04DefaultState 测试文件中

演示了变量 x 的默认作用范围是 Thread

```
@State(Scope.Thread)
public class JMHSample_04_DefaultState {
    double x = Math.PI;
    @Benchmark
    public void measure() {
        x++;
    }
}
```

@Setup 和 @TearDown

@Setup ——基准测试前的初始化动作

@TearDown ——基准测试后的动作，做一些全局的配置

Level 值——标明方法运行的时机

- Trial：默认的级别——Benchmark 级别
- Iteration：每次迭代都会运行
- Invocation：每次方法调用都会运行，这个是粒度最细的

@Setup 和 @TearDown

如果初始化操作和方法相关，最好使用 Invocation 级别

但大多数场景是一些全局的资源

一个 Spring 的 DAO，就使用默认的 Trial，只初始化一次就可以

@Param

@Param 注解只能修饰字段

用来测试不同的参数，对程序性能的影响

配合 @State注解，可以同时制定这些参数的执行范围

@Param

拉勾教育
互联网人实战大学

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(1)
@State(Scope.Benchmark)
public class JMHSample_27_Params {
    @Param({"1", "31", "65", "101", "103"})
    public int arg;
    @Param({"0", "1", "2", "4", "8", "16", "32"})
    public int certainty;
    @Benchmark
    public boolean bench() {
        return BigInteger.valueOf(arg).isProbablePrime(certainty);
    }
    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(JMHSample_27_Params.class.getSimpleName())
```

@Param

拉勾教育
互联网人实战大学

```
public class JMHSample_27_Params {
    @Param({"1", "31", "65", "101", "103"})
    public int arg;
    @Param({"0", "1", "2", "4", "8", "16", "32"})
    public int certainty;
    @Benchmark
    public boolean bench() {
        return BigInteger.valueOf(arg).isProbablePrime(certainty);
    }

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(JMHSample_27_Params.class.getSimpleName())
//                .param("arg", "41", "42") // Use this to selectively constrain/override parameters
                .build();

        new Runner(opt).run();
    }
}
```

@Param

拉勾教育
互联网人实战大学

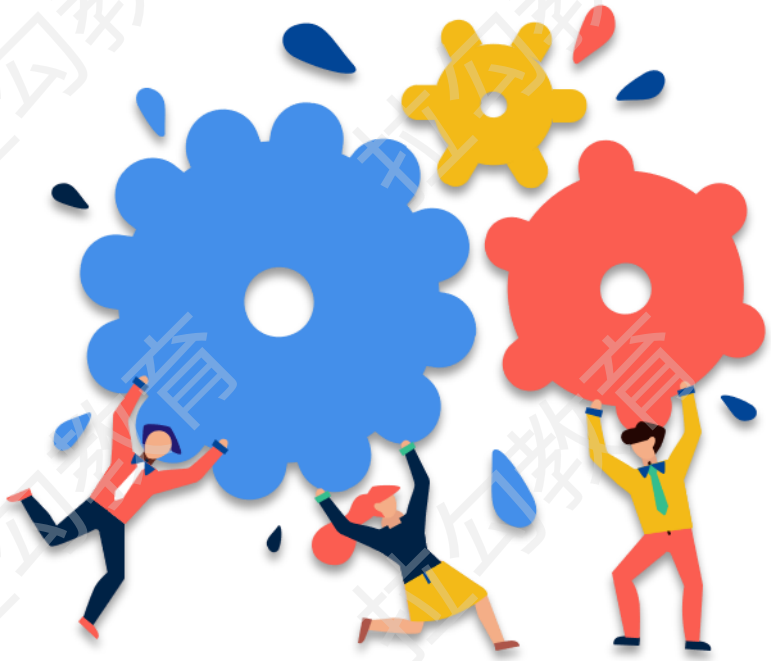
Benchmark	(arg)	(certainty)	Mode	Cnt	Score	Error	Units
JMHSample_27_Params.bench	1	0	avgt	5	4.088 ±	0.065	ns/op
JMHSample_27_Params.bench	1	1	avgt	5	6.416 ±	0.141	ns/op
JMHSample_27_Params.bench	1	2	avgt	5	6.854 ±	0.339	ns/op
JMHSample_27_Params.bench	1	4	avgt	5	6.768 ±	0.283	ns/op
JMHSample_27_Params.bench	1	8	avgt	5	6.826 ±	0.194	ns/op
JMHSample_27_Params.bench	1	16	avgt	5	7.476 ±	2.800	ns/op
JMHSample_27_Params.bench	1	32	avgt	5	7.652 ±	1.183	ns/op
JMHSample_27_Params.bench	31	0	avgt	5	7.077 ±	3.206	ns/op
JMHSample_27_Params.bench	31	1	avgt	5	427.488 ±	108.293	ns/op
JMHSample_27_Params.bench	31	2	avgt	5	390.451 ±	55.742	ns/op
JMHSample_27_Params.bench	31	4	avgt	5	767.300 ±	187.980	ns/op
JMHSample_27_Params.bench	31	8	avgt	5	1471.293 ±	326.468	ns/op
JMHSample_27_Params.bench	31	16	avgt	5	3019.979 ±	1284.910	ns/op
JMHSample_27_Params.bench	31	32	avgt	5	6122.833 ±	2590.576	ns/op

@CompilerControl

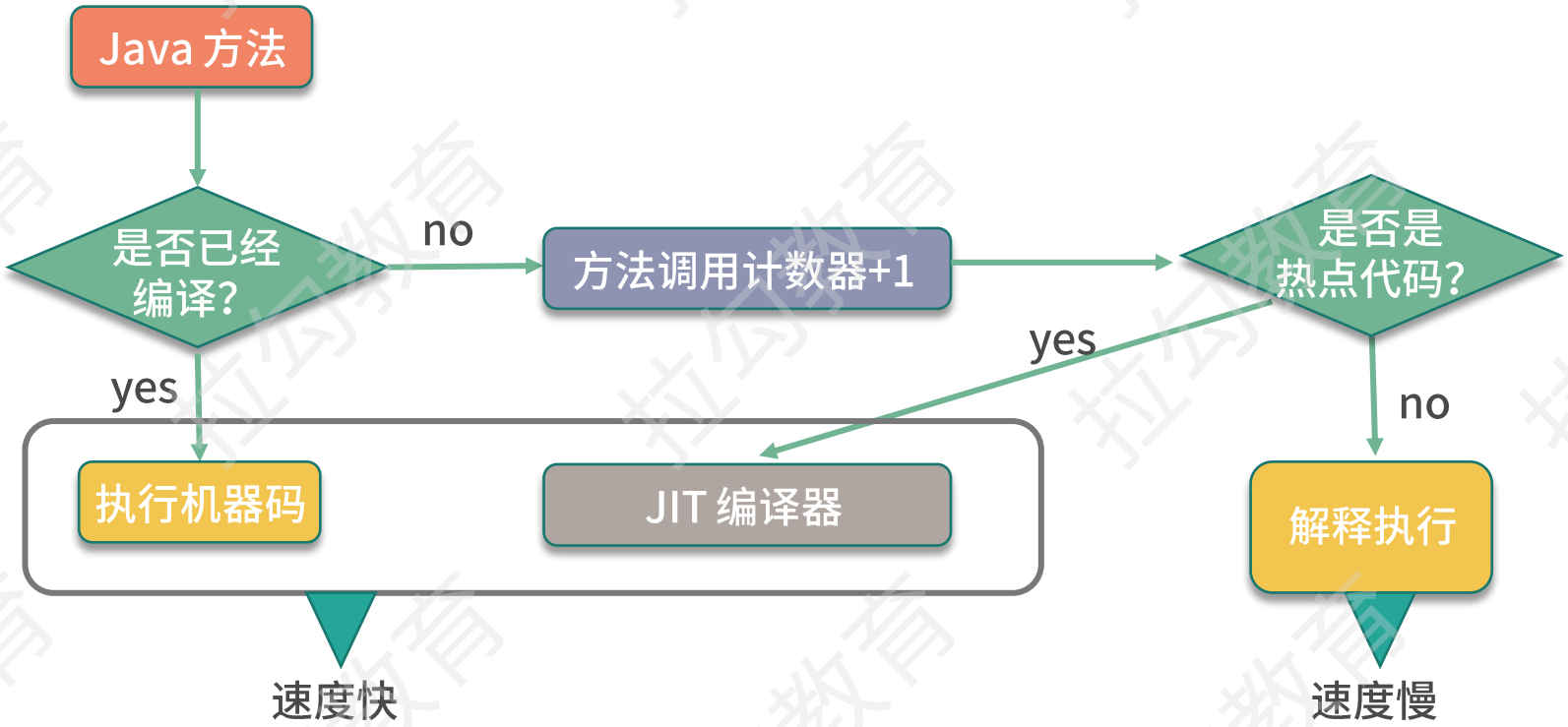
Java 中方法调用的开销是比较大的，尤其是在调用量非常大的情况下

getter/setter 方法在 Java 代码中大量存在

在访问时，需要创建相应的栈帧，访问到需要的字段后，再弹出栈帧，恢复原程序的执行



@CompilerControl



@CompilerControl

@CompilerControl 注解可以用在类或者方法上，能够控制方法的编译行为

强制使用内联
（INLINE）

禁止使用内联
（DONT_INLINE）

禁止方法编译
（EXCLUDE）

将结果图形化

使用 JMH 测试的结果，可以二次加工，进行图形化展示

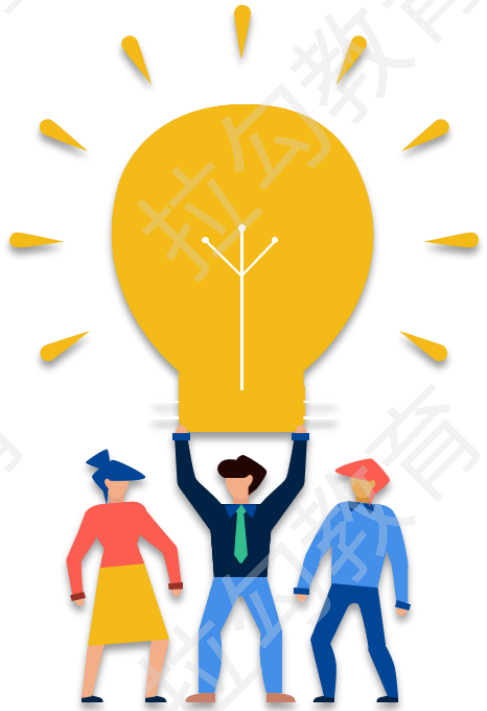
通过运行时，指定输出的格式文件，即可获得相应格式的性能测试结果

```
Options opt = new OptionsBuilder()
    .resultFormat(ResultFormatType.JSON)
    .build();
```

将结果图形化

JMH 支持 5 种格式结果：

- **TEXT** 导出文本文件
- **CSV** 导出 csv 格式文件
- **SCSV** 导出 scsv 等格式的文件
- **JSON** 导出成 json 文件
- **LATEX** 导出到 latex，一种基于 TEX 的排版系统



将结果图形化

Benchmark	Score	Score Error (9	Unit
baseline	1422.122932	63.066118	ops/us
baseline:reader	189.933906	4.832569	ops/us
baseline:writer	1232.189027	61.781028	ops/us
contended	1485.241841	58.845509	ops/us
contended:reader	273.259729	43.361216	ops/us
contended:writer	1211.982112	30.155933	ops/us
hierarchy	1613.593228	42.477062	ops/us
hierarchy:reader	411.706328	12.434878	ops/us
hierarchy:writer	1201.8869	35.72735	ops/us
padded			
padded:reader			
padded:writer			
sparse			
sparse:reader			
sparse:writer			

结果图形化制图工具

JMH Visualizer

有一个开源的项目，通过导出 json 文件

上传至 JMH Visualizer（https://jmh.morethan.io/），可得到简单的统计结果

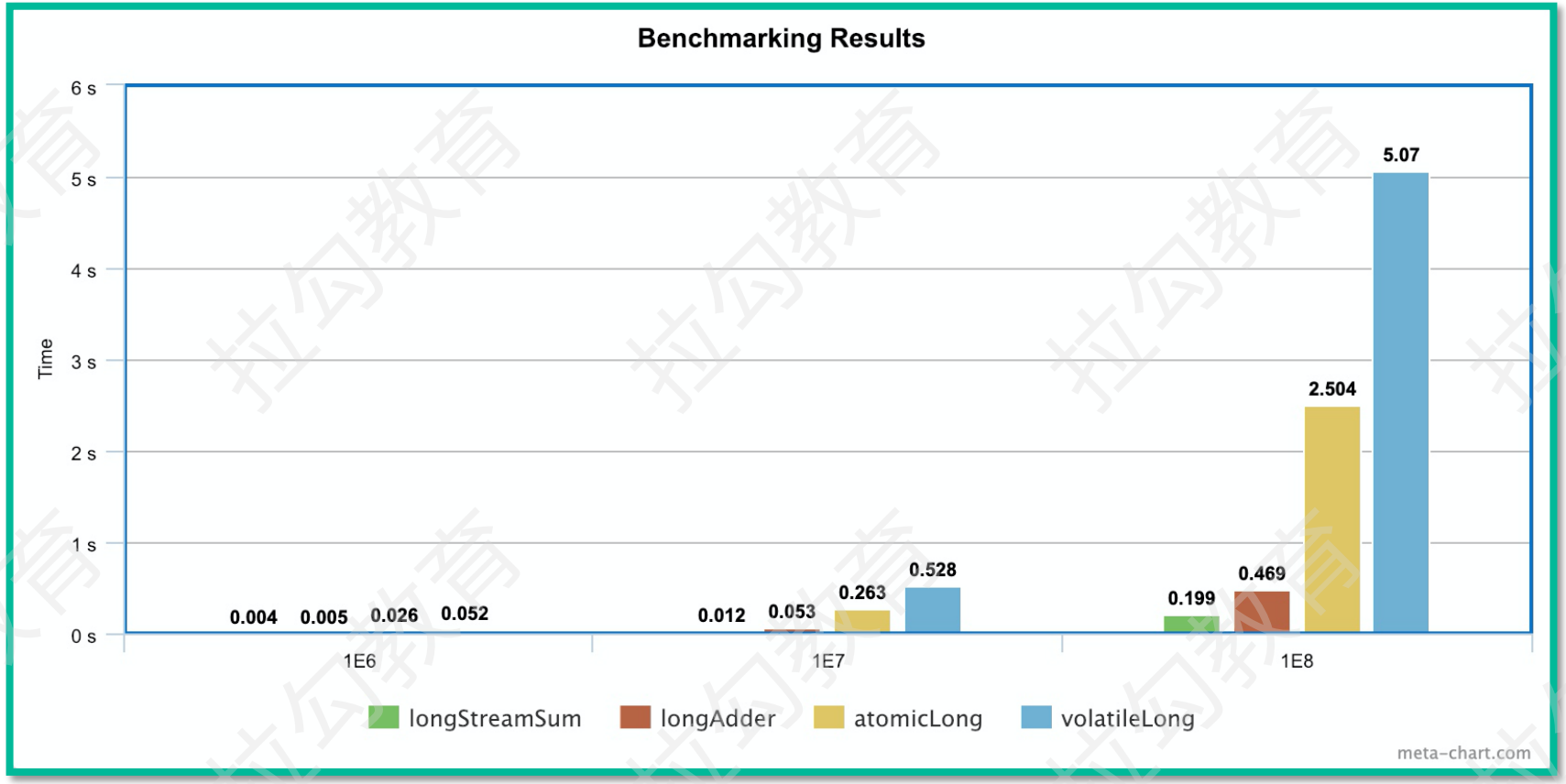
结果图形化制图工具

JMH Visual Chart

结果图形化制图工具

拉勾教育
互联网人实战大学

meta-chart



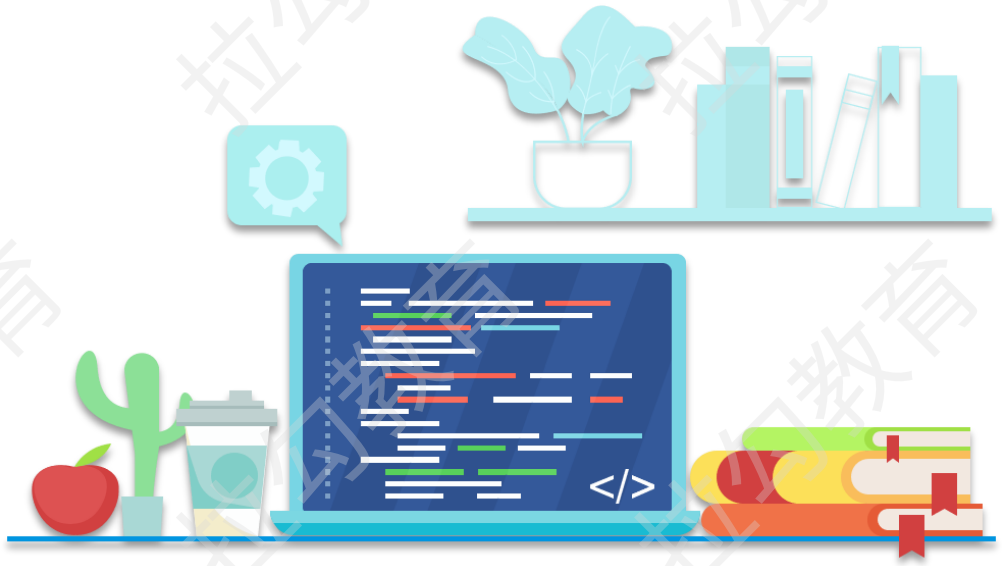
小结

本课时主要介绍了**基准测试工具——JMH**

官方有丰富的 JMH 的示例，比如伪共享（FalseSharing）的影响等高级话题

JMH 可以使用确切的测试数据，来支持我们的分析结果

定位到热点代码，需要使用基准测试工具进行专项优化，直到性能有了显著的提升



Next：06 | 《案例分析：缓冲区如何让代码加速》

拉勾教育

—互联网人实战大学—

关注拉勾「教育公众号」
获取更多课程信息

L / A / G / O / U