设置的参数值

```
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
@Threads(1)
@Fork(1)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
```

理由

1. @BenchmarkMode(Mode.AverageTime)
对于 `indexOf` 和 `replace` 方法，通常更关心的是单次操作的执行时间，比吞吐量（Throughput）能更准确反映每个操作的开销。而且测试中比较了长短字符串，用时间更加直观。

2. @Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
对于简单操作，5 次热身有助于 JVM 进入优化状态，1 秒的时间足以触发 JIT 编译。

3. @Measurement(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
为了获得足够的样本，测量迭代次数设置为 5 次，每次持续 2 秒。这样可以更好地捕获性能波动，并避免因为测试时间过短导致数据偏差。由于 `indexOf` 和 `replace` 是相对快速的操作，这个时间设置足够获得准确数据。

4. @Threads(1)
测试的是单个字符串的`indexOf`和`replace`方法，都是串行的、非并发的，使用单线程更能反映其原始性能。

5. @Fork(1)
测试的是简单的字符串操作，不太涉及GC、多线程等等干扰。如果要更精确的数据，可以增加到 2 次 Fork，但对于这种简单场景，1 次 Fork 是合理的权衡。

6. @OutputTimeUnit(TimeUnit.NANOSECONDS)
`indexOf` 和 `replace` 方法执行时间很短，纳秒是合理的单位，否则会是0开头的小数。

另外，特意与java.lang.String作比较，以显示差距。

结果

```
Benchmark.myStringIndexOfLong          avgt    5    160.580 ±    1.375  ns/op
Benchmark.myStringIndexOfShort         avgt    5    113.858 ±   31.243  ns/op
Benchmark.myStringReplaceLong          avgt    5  28347.729 ± 4823.478  ns/op
Benchmark.myStringReplaceShort         avgt    5    280.999 ±   50.287  ns/op
Benchmark.standardStringIndexOfLong    avgt    5      4.750 ±    1.287  ns/op
Benchmark.standardStringIndexOfShort   avgt    5     19.718 ±    4.710  ns/op
Benchmark.standardStringReplaceLong    avgt    5   5625.513 ±  374.832  ns/op
Benchmark.standardStringReplaceShort   avgt    5     72.153 ±    2.637  ns/op
```

使用java.lang.String的indexOf，“长字符串”比“短字符串”快。
这仅仅是因为long的测试数据长10000字符，模式串出现频率为1%，平均得到的index约为100；
而short的答案为固定值103。