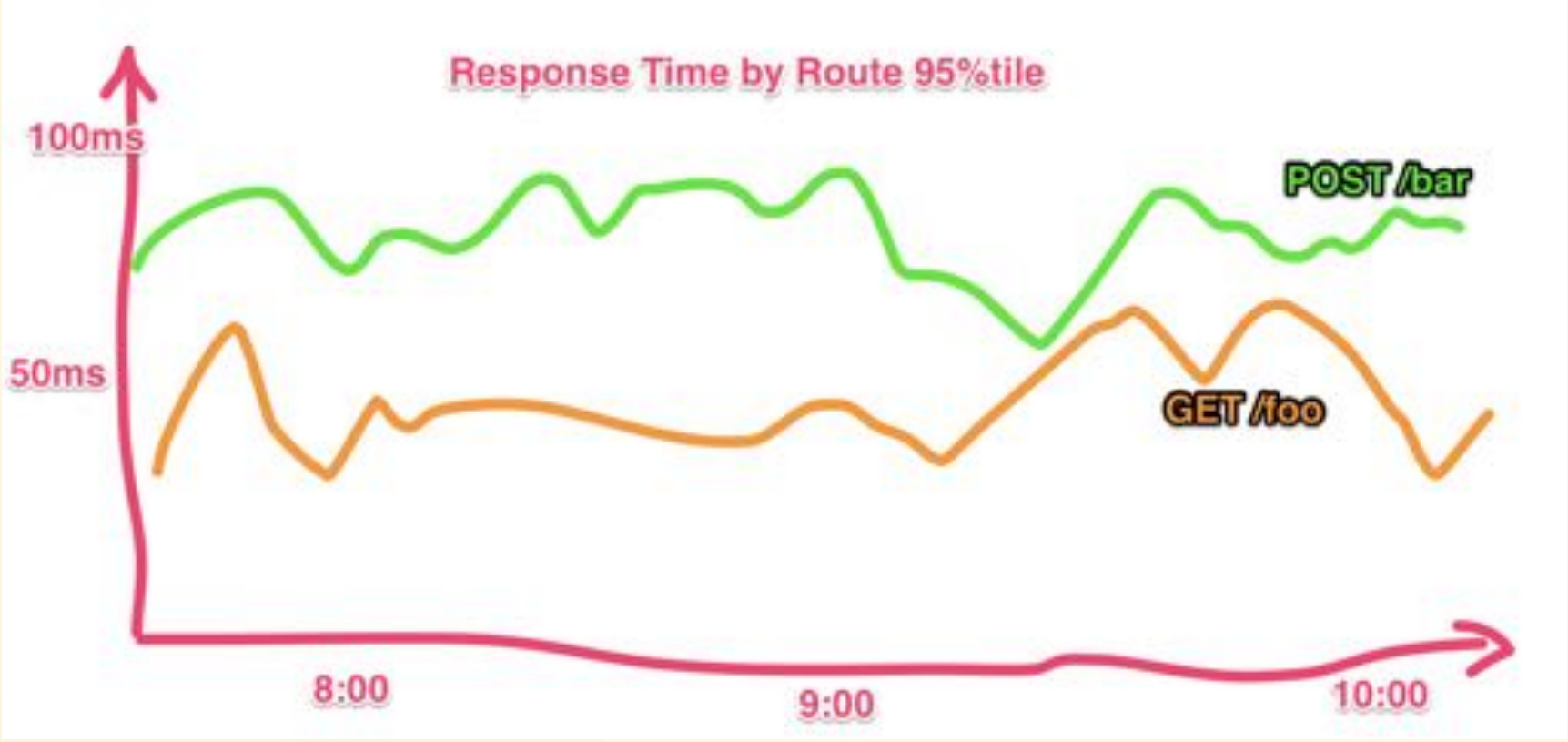
JVM Web Application Metrics & Monitoring

- 株式会社 FOLIO
- 海津 研 @krrrr38


Response Time by Route 95%tile
100ms
50ms
POST /bar
GET /foo
8:00
9:00
10:00

1. メトリクス収集方法
2. メトリクスの形式について
3. JVM Web Application メトリクス

① メトリクス収集方法

Metrics 取得方法

- メトリクスを取る
- メトリクスを出す
- メトリクスを集める
- メトリクスを描画する・監視する

Metrics 取得方法

- メトリクスを取る
  - デフォルトでJVMが提供しているもの
  - アプリケーション内部で集めるもの
- メトリクスを出す
  - httpでメトリクスを提供する
  - 指定のツールを使い、JMX経由で取る
    - jcmd, jstat, ...

Metrics 取得方法

- メトリクスを集める
  - 複数server/processから提供されるものを集約する
  - pull型/push型アーキテクチャ
- メトリクスを描画する・監視する
  - 良い感じにグラフにする
  - ダッシュボードを作る
  - 得られたメトリクスに対する監視を設定する

pull型/push型アーキテクチャ

- push型
  - 収集先へデータを投げる
  - server -(process)-> metrics storage gateway
  - e.g. mackerel agent
- pull型
  - 収集側がデータを取りに来る
  - server <- metrics
  - e.g. prometheus

Example 1

- mackerel-agent (mackerel-jvm-plugin)
- mackerel
  - https://mackerel.io/

下線はprocessが動いているホストで主に行うもの

jvm app (jmx, perf,...)

↓

jstat, jcmd

↓

mackerel-agent で取って

↓

mackerel に投げる

↓

mackerel で描画する

Example 2

prometheus, grafana

jvm app: /metrics から prom metrics

↓

prometheus で集めて

↓

grafana で描画

prometheus で監視

Example 3

- telegraf
  - metrics interface gateway
  - 取り敢えずなげて、telegrafは後がどこかへ投げたりする

https://github.com/influxdata/telegraf

② メトリクスの形式について

key value 形式は古い

- JMX
- dropwizard metrics

key+labels value 形式

key+labels value 形式

- prometheus
- kamon
- **micrometer**
  - http://micrometer.io/

```
response_time_GET_foo = 123ms
response_time_GET_bar = 250ms
response_time_POST_bar = 210ms
response_time_POST_foo_bar = 512ms
```

average/max/95%tile を別に取る

```
response_time_GET_foo_average = 123ms
response_time_GET_foo_max = 123ms
response_time_GET_foo_95% = 123ms
response_time_GET_bar_average = 250ms
response_time_GET_bar_max = 250ms
response_time_GET_bar_95% = 250ms
response_time_POST_bar_average = 210ms
response_time_POST_bar_max = 210ms
response_time_POST_bar_95% = 210ms
response_time_POST_foo_bar_average = 512ms
response_time_POST_foo_bar_max = 512ms
response_time_POST_foo_bar_95% = 512ms
```

全ての95%tileのみ描画/監視するには...?

- 全列挙

```
SELECT * FROM metrics
  WHERE key IN (
response_time_GET_foo_95%,
response_time_GET_bar_95%,
response_time_POST_bar_95%,
response_time_POST_foo_bar_95%
)
```

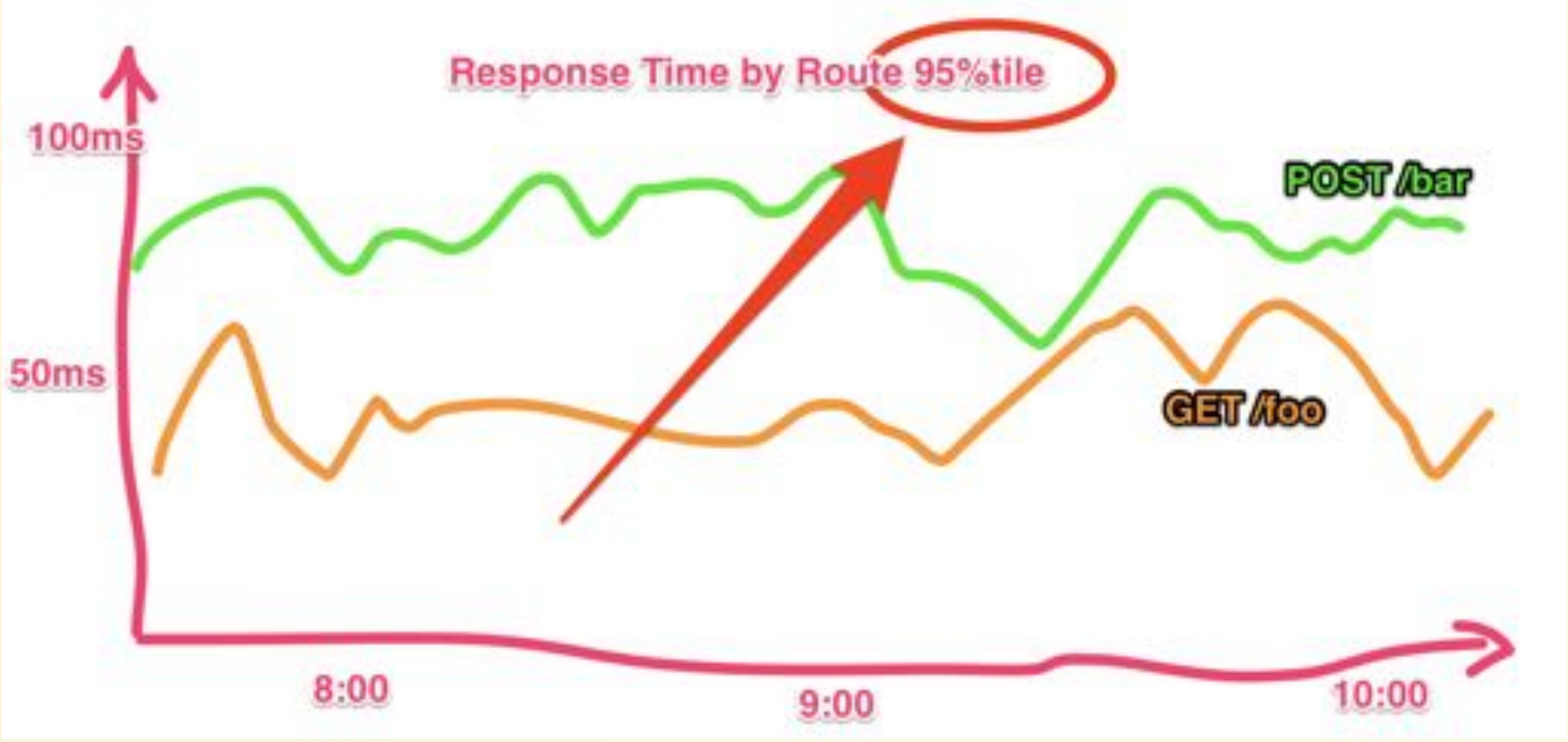
- index 無視

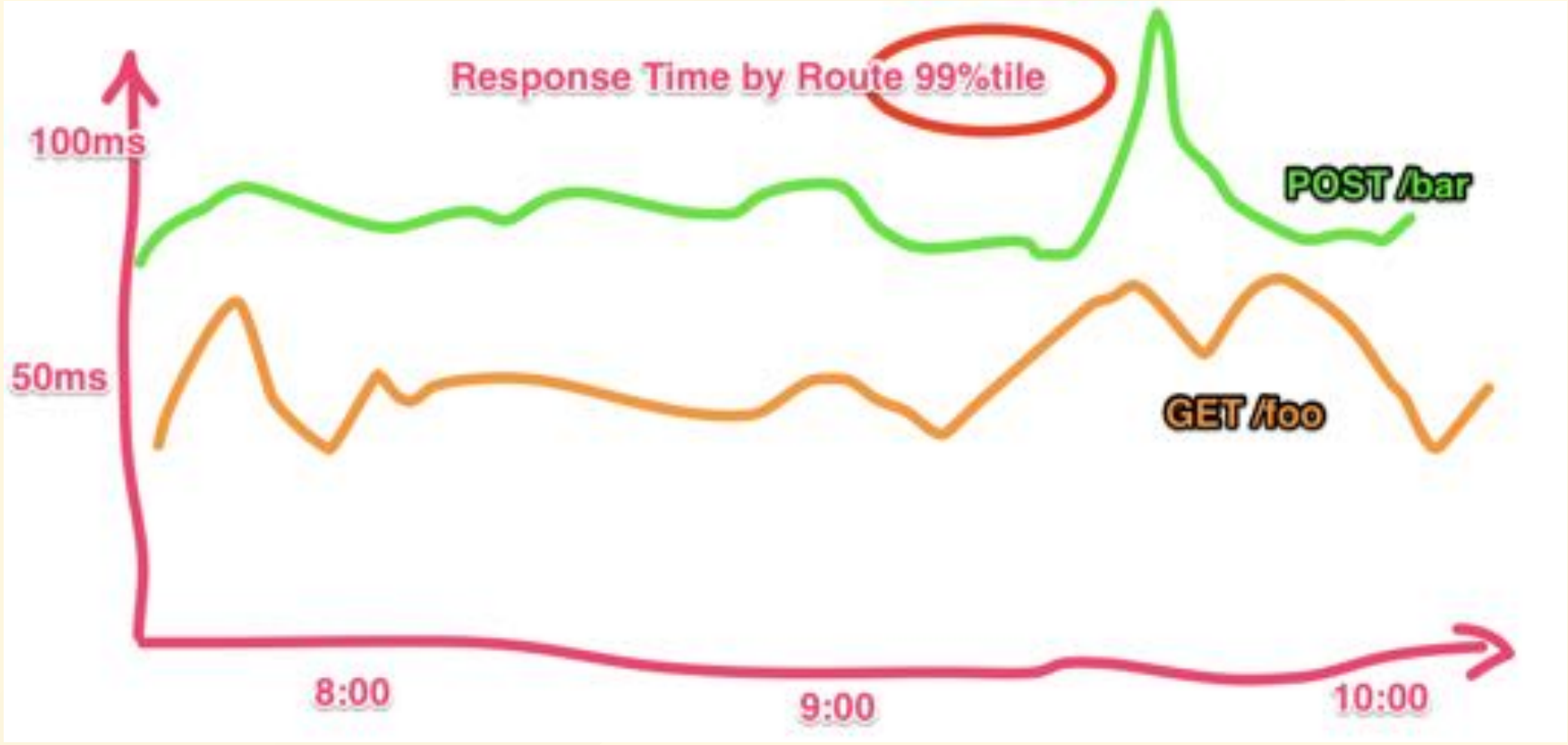
```
SELECT * FROM metrics
  WHERE key LIKE 'response_time_%_95\%'
```

描画・監視する際の絞込が難しい

```
response_time{"method"=GET, path=foo, percent=50} = 123ms
response_time{"method"=GET, path=foo, percent=95} = 123ms
response_time{"method"=GET, path=foo, percent=100} = 123ms
```

- **描画・監視する際の絞込が簡単**
- **描画時の変数template化が用意**
  - `percent`の部分のみ変数にしておいて切り替える
  - `method=POST`のみ見たい


Response Time by Route 95%tile
100ms
50ms
POST /bar
GET /foo
8:00
9:00
10:00


Response Time by Route 99%tile
100ms
50ms
POST /bar
GET /foo
8:00
9:00
10:00

③ JVM Web Application Metrics

ここだけは見て欲しいもの

- heap usage
- gc lifecycle
- thread pool
- connection pool
- cpu
- アプリケーションメトリクス
  - request count / response time
  - queue
  - ...

heap usage / gc lifecycle

- memory leak 調査など
  - 基本的にはXmxなどをした上で、その中で適切に gc されているかを確認
- 設定一例 (もちろんケースバイケース)
  - in-memory cacheなどを利用しない場合は512M~2G程など設定
    - 大きすぎる場合のFullGCのSTWなどが怖い
    - gc のlifecycleが必要

thread / connection pool

- blockingなサーバの場合それらを処理するthread poolのactive率の監視
  - e.g. servlet の not nio な api を利用している
- jedis/jdbc/... thread pool in 非同期application
  - blockingな処理が入る場合は、基本的に専用のthread poolが用意される
  - 専用のthread poolには必ず監視
- implicit global
  - defaultで利用されるForkjoinPoolは無限に増える可能性がある

server cpu / network / etc

- 綺麗な非同期applicationをかけた場合に、ようやくcpuがネック
- networkはもちろん設定に応じて上限に達するので監視が必要

request count / response time / queue / ...

- 異常系
  - 遅くなっている
  - 処理しきれていない
  - バグっている

後は泥臭くとっていく

micrometer

micrometer

Java界隈のデファクトスタンダードになりそう？

key+labels value 形式のデータを取れる

- アプリケーションのメトリクスを取得
  - plugin機構
    - JVM System metrics, GC, logback, tomcat, jetty, hikaricp
- メトリクスを出す
  - jmx, prom, influx, datadog agent,...

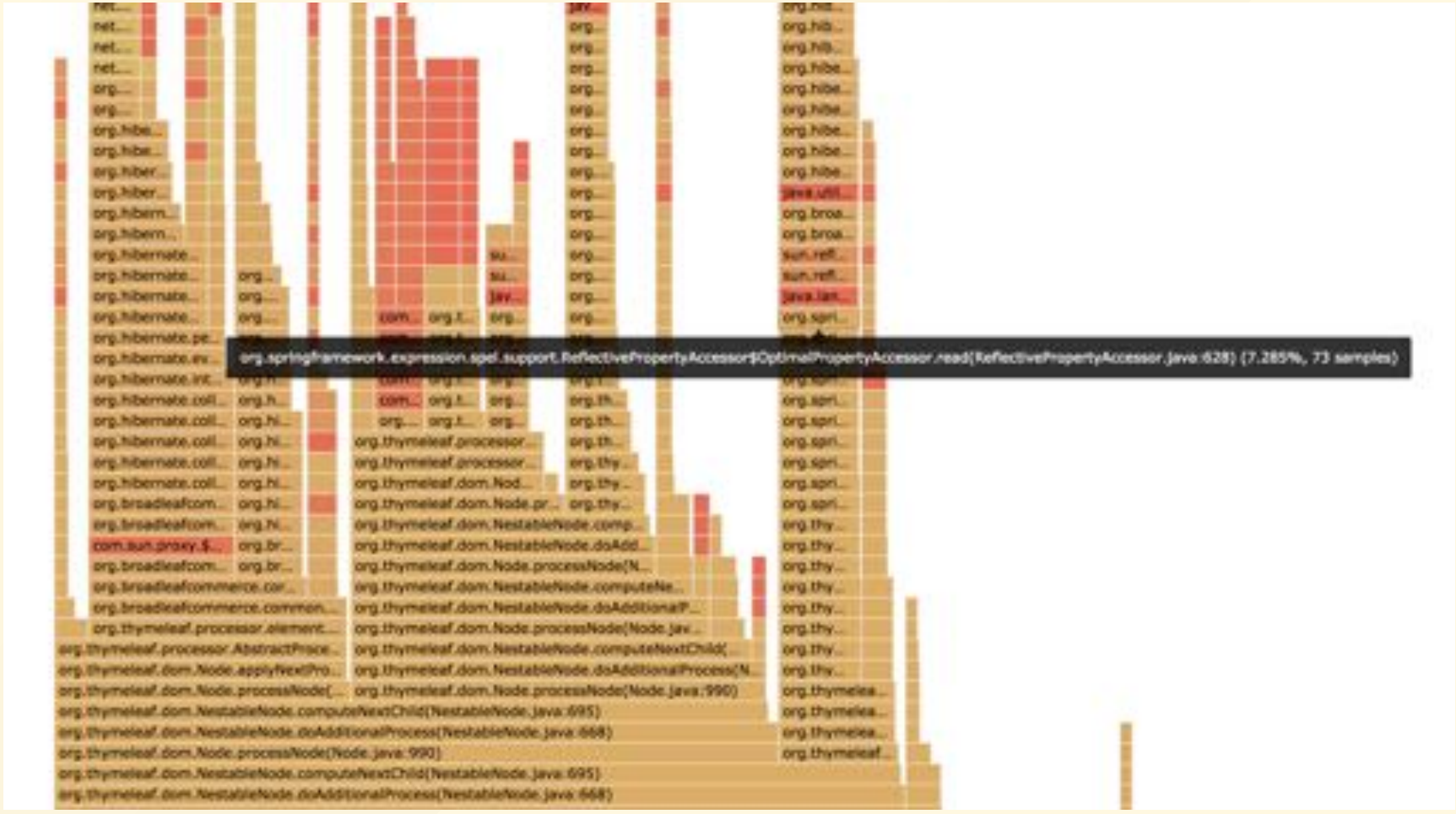
他にも取れるものはある

jmx, java agent, `-XX:+FlightRecorder`, ...

agent

`java -jar` など実行する際に合わせて指定するもの

e.g. NewRelic


org.springframework.expression.spel.support.ReflectivePropertyAccessor$OptimalPropertyAccessor.read(ReflectivePropertyAccessor.java:628) (7.285%, 73 samples)
org.thymeleaf.dom.Node.processNode(Node.java:990)
org.thymeleaf.dom.NestableNode.computeNextChild(NestableNode.java:695)
org.thymeleaf.dom.NestableNode.doAdditionalProcess(NestableNode.java:668)
org.thymeleaf.dom.Node.processNode(Node.java:990)
org.thymeleaf.dom.NestableNode.computeNextChild(NestableNode.java:695)
org.thymeleaf.dom.NestableNode.doAdditionalProcess(NestableNode.java:668)

- framegraph: http://www.brendangregg.com/blog/2015-11-06/java-mixed-mode-flame-graphs.html
  - https://github.com/jvm-profiling-tools/perf-map-agent
  - https://glowroot.org/