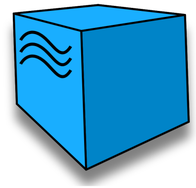
Parallel test execution

Теория и практика

Тучс Дмитрий Heisenbug Piter 09.04.2021

О чем



gRPC

{REST}

тесты

JVM

mvn test

gradle test

P1

P2

P3

P4

```
@Order(-1)
```

С вами ближайший час

- QA lead в «highload сердце» AdTech компании PropellerAds
- Всю карьеру (с 2008) в IT
- Использую JUnit 5 с версии 5.0.0-ALPHA
- Увлекаюсь менторством

С момента появления в версии 5.3 (*06.2018*) JUnit parallel test execution находится в статусе EXPERIMENTAL API

Следите за изменениями на https://junit.org

История вопроса

TestNG

```
<suite name="My suite" parallel="methods" thread-count="5">
```

```
@Test(threadPoolSize = 3, invocationCount = 10,  timeOut = 10000)
public void testServer() {
```

JUnit 4

```
@Test
 public void test() {
   Class[] cls={ParallelTest1.class,ParallelTest2.class };

   //Parallel among classes
   JUnitCore.runClasses(ParallelComputer.classes(), cls);

   //Parallel among methods in a class
   JUnitCore.runClasses(ParallelComputer.methods(), cls);

   //Parallel all methods in all classes
   JUnitCore.runClasses(new ParallelComputer(true, true), cls);
 }
public static class ParallelTest1{
   @Test public void a(){}
   @Test public void b(){}
}
public static class ParallelTest2{
   @Test public void a(){}
   @Test public void b(){}
}
```

```
<configuration>
  <parallel>methods</parallel>
  <threadCount>10</threadCount>
</configuration>
```

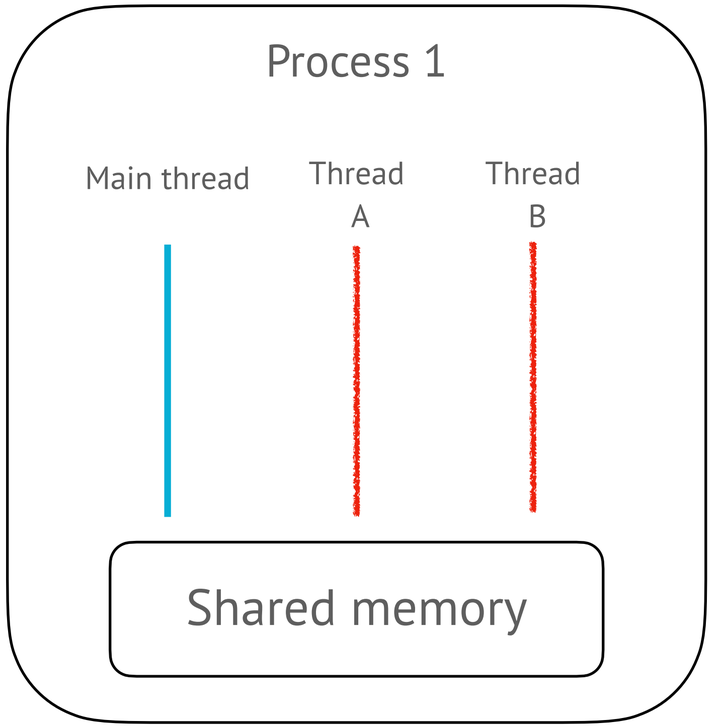
```
 <forkCount>3</forkCount>
 <reuseForks>true</reuseForks>
```

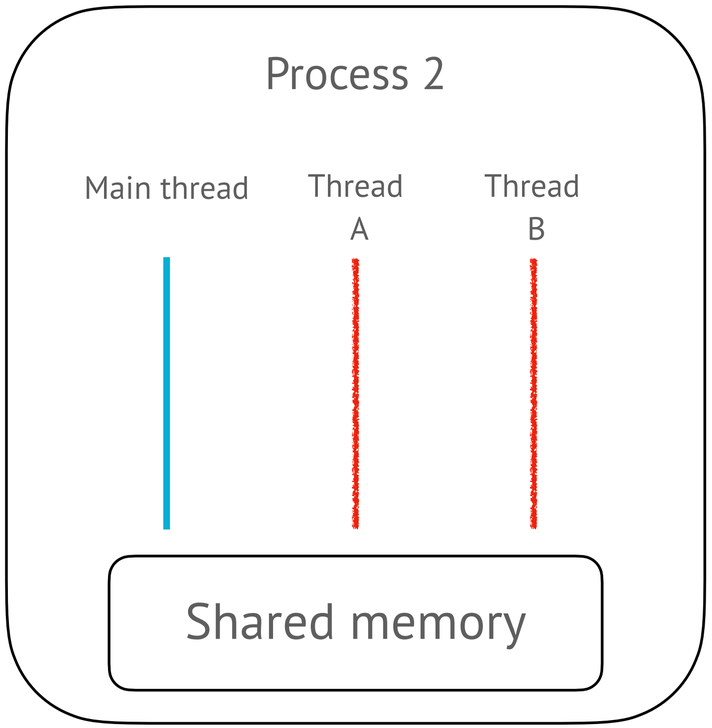
build.gradle

```
tasks.withType(Test).configureEach {
    maxParallelForks = 4
}
```

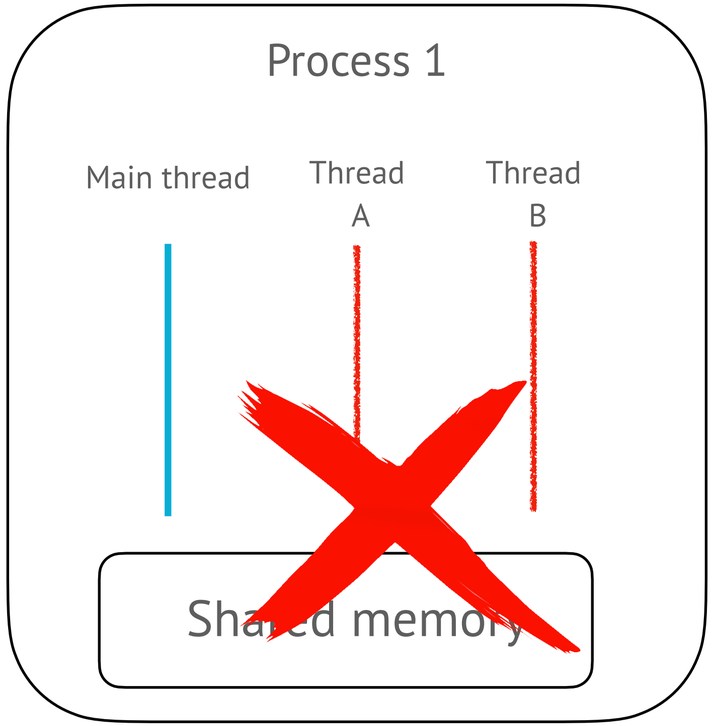
Процессы и потоки

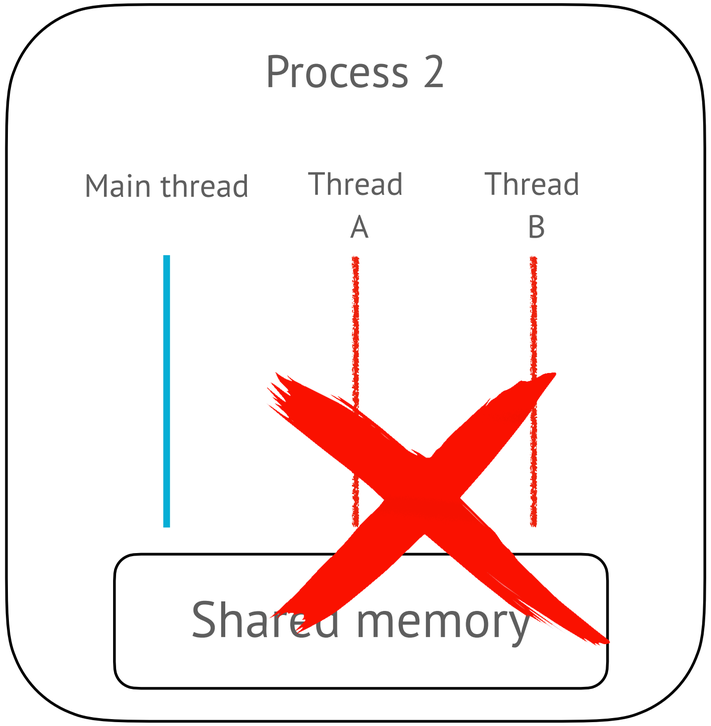
- Форки (forks) gradle / surefire порождают процессы
- Параллелизация тестов в JUnit5 осуществляется внутри процесса (jvm) на уровне потоков (threads)





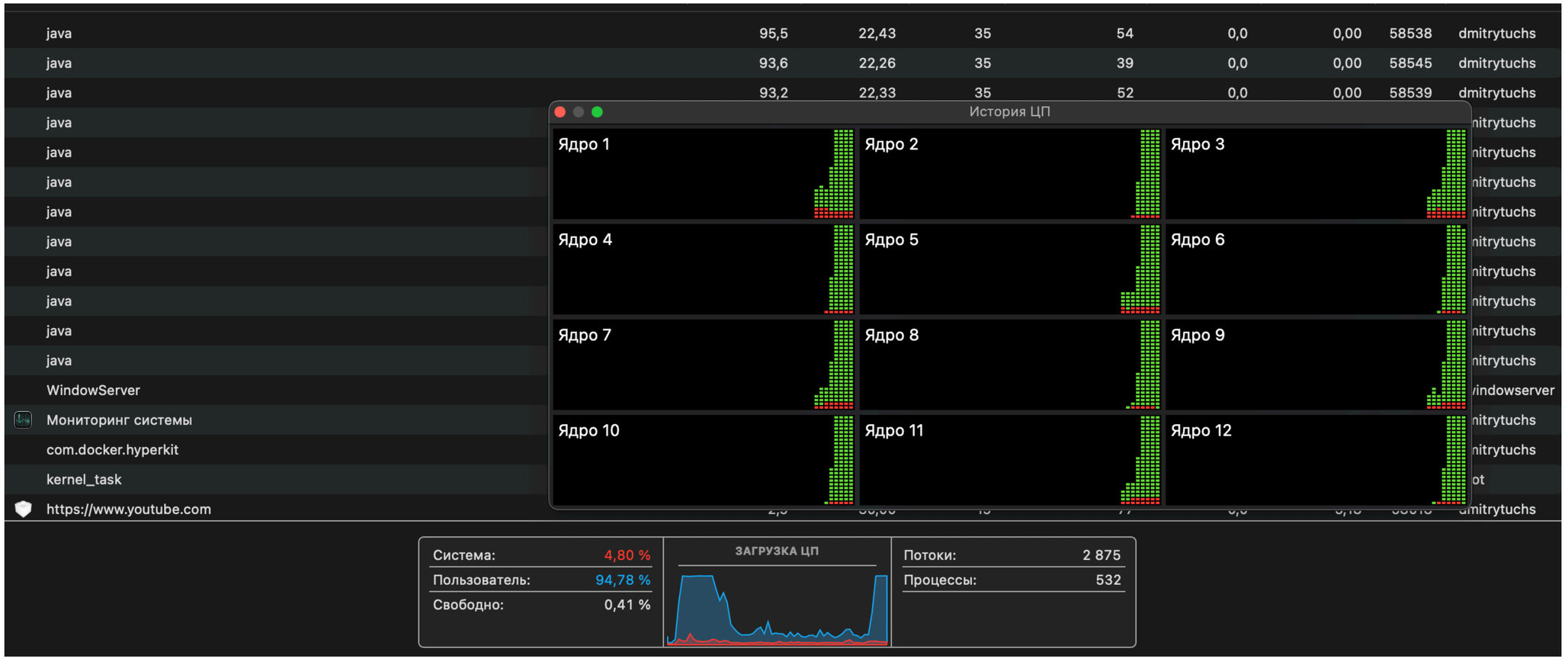
Форки - за что мы их любим?





```
@Test
void aTest1() {
```

И не любим?


java 95,5 22,43 35 54 0,0 0,00 58538 dmitrytuchs
java 93,6 22,26 35 39 0,0 0,00 58545 dmitrytuchs
java 93,2 22,33 35 52 0,0 0,00 58539 dmitrytuchs
java
java
java
java
java
java
java
java
WindowServer
Мониторинг системы
com.docker.hyperkit
kernel_task
https://www.youtube.com
История ЦП
Ядро 1
Ядро 2
Ядро 3
Ядро 4
Ядро 5
Ядро 6
Ядро 7
Ядро 8
Ядро 9
Ядро 10
Ядро 11
Ядро 12
Система: 4,80 %
Пользователь: 94,78 %
Свободно: 0,41 %
ЗАГРУЗКА ЦП
Потоки: 2 875
Процессы: 532

JUnit 5 и Maven

- Встроенные в Maven Surefire plugin механизмы параллелизации **не поддерживают** JUnit 5. И это хорошо!

```
<plugins>
[...]
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M5</version>
    <configuration>
        <forkCount>3</forkCount>
        <reuseForks>true</reuseForks>
```

https://maven.apache.org/surefire/maven-surefire-plugin/examples/junit-platform.html

… и Gradle

- В Gradle все несколько сложнее. Форки могут использоваться совместно с JUnit parallel execution, но это не имеет особого смысла

```
test {
    useJUnitPlatform()
    systemProperties(System.getProperties())
    maxParallelForks = Runtime.runtime.availableProcessors().intdiv(2) ?: 1
}
```

marcphilipp commented on 15 Feb 2019

Personally, I would just use JUnit's parallelism because it should have less overhead.

👍 2

Marc Philipp · 1-й

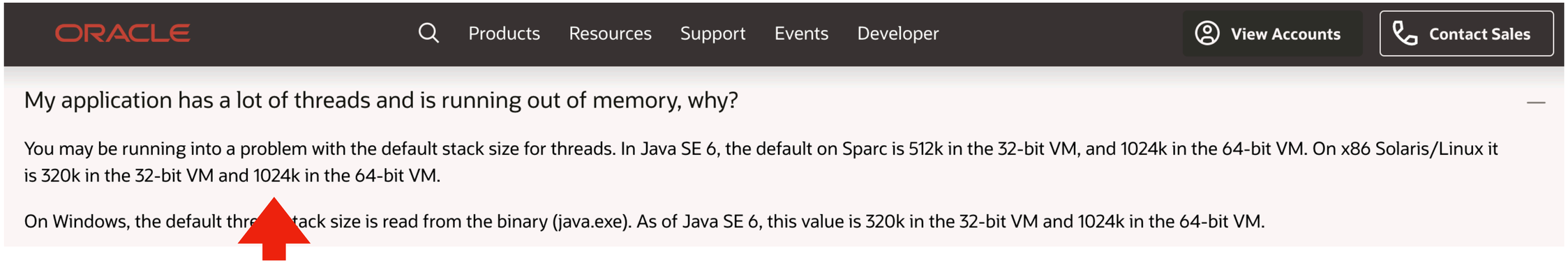
Software Engineer bei Gradle Inc., JUnit Team Lead

https://github.com/gradle/gradle/issues/6453#issuecomment-463939748

Цена потока

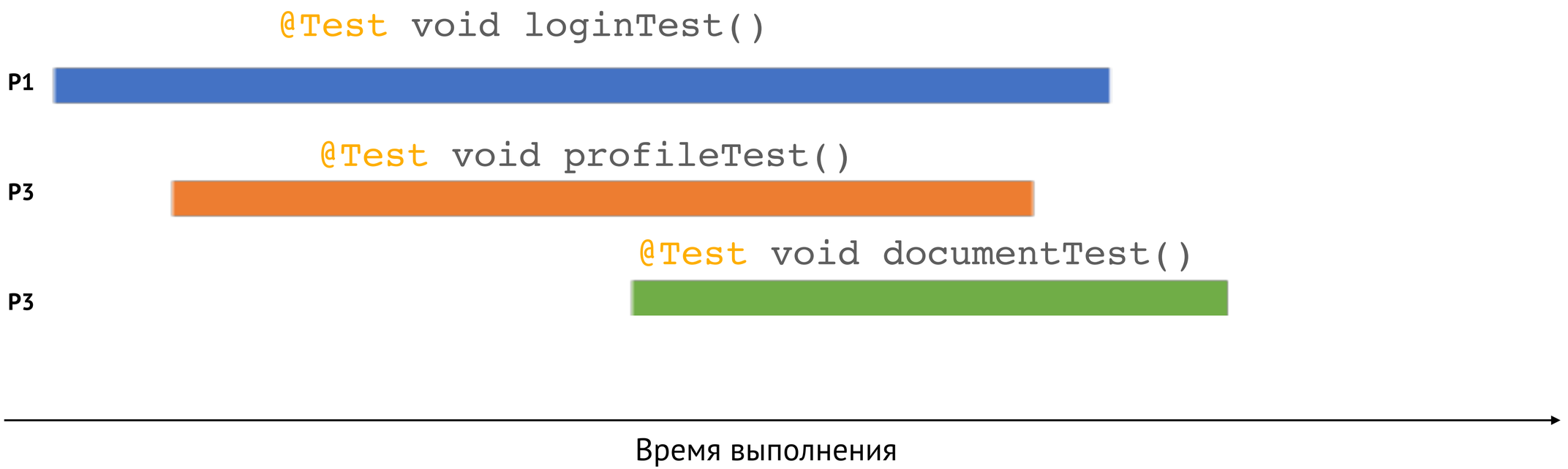
2.5.2. Java Virtual Machine Stacks

Each Java Virtual Machine thread has a private *Java Virtual Machine stack*, created at the same time as the thread.



https://www.oracle.com/java/technologies/hotspotfaq.html#threads_oom

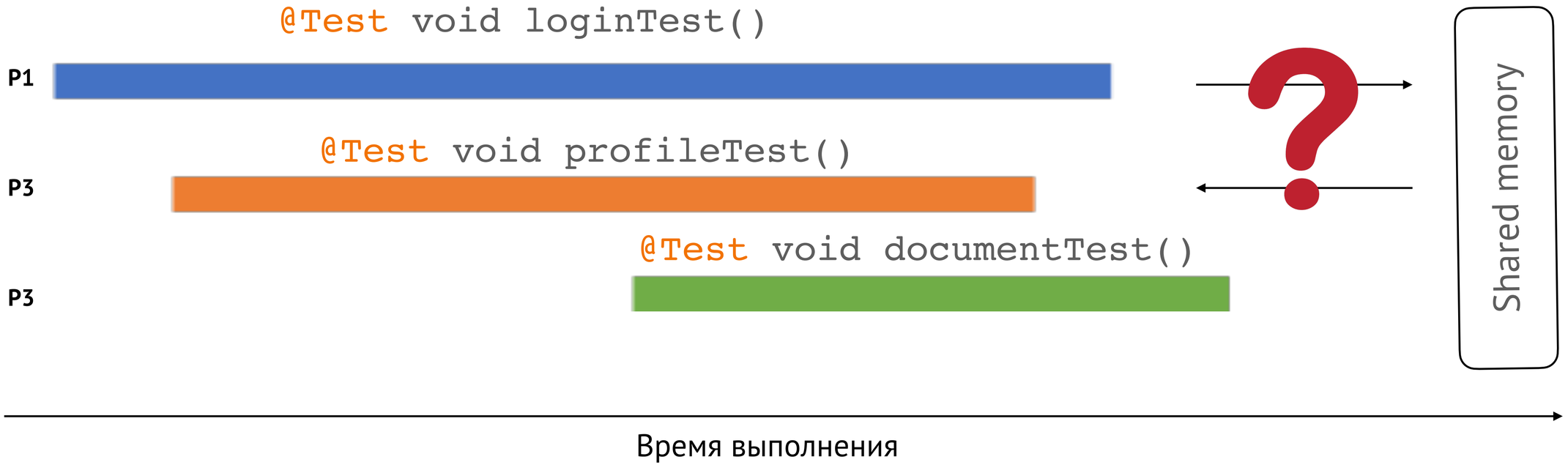
Многопоточное программирование и тесты



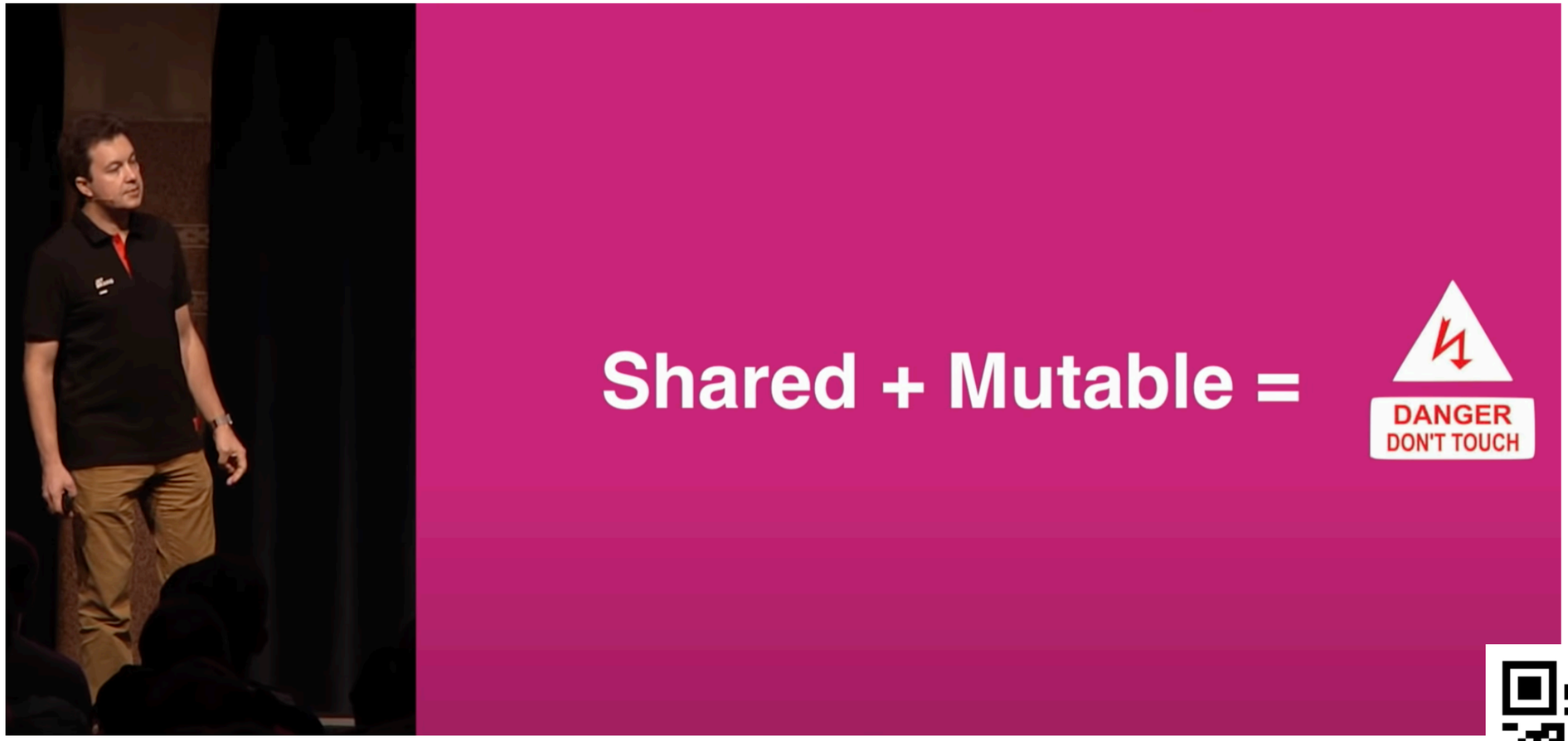
Теория многопоточного программирования рассматривает только взаимодействие между потоками через shared memory и/или отправку сообщений.

Теоретически тесты возможно (и желательно) написать так, что никакого взаимодействия между потоками не будет.

Многопоточное программирование и тесты



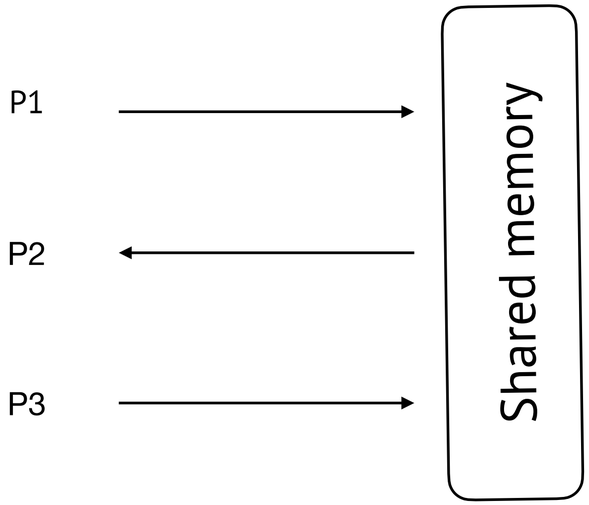
Thread safety



KotlinConf 2018 - Kotlin Coroutines in Practice by Roman Elizarov https://www.youtube.com/watch?v=a3agLJQ6vt8

А если очень хочется?

- Не надо. Не разделяйте объекты между потоками вообще (**Thread Confinement**)
- Использовать **immutable** объекты в shared memory
- Использовать **stateless** объекты
- Синхронизировать **любое** обращение к состоянию mutable объекта в shared memory

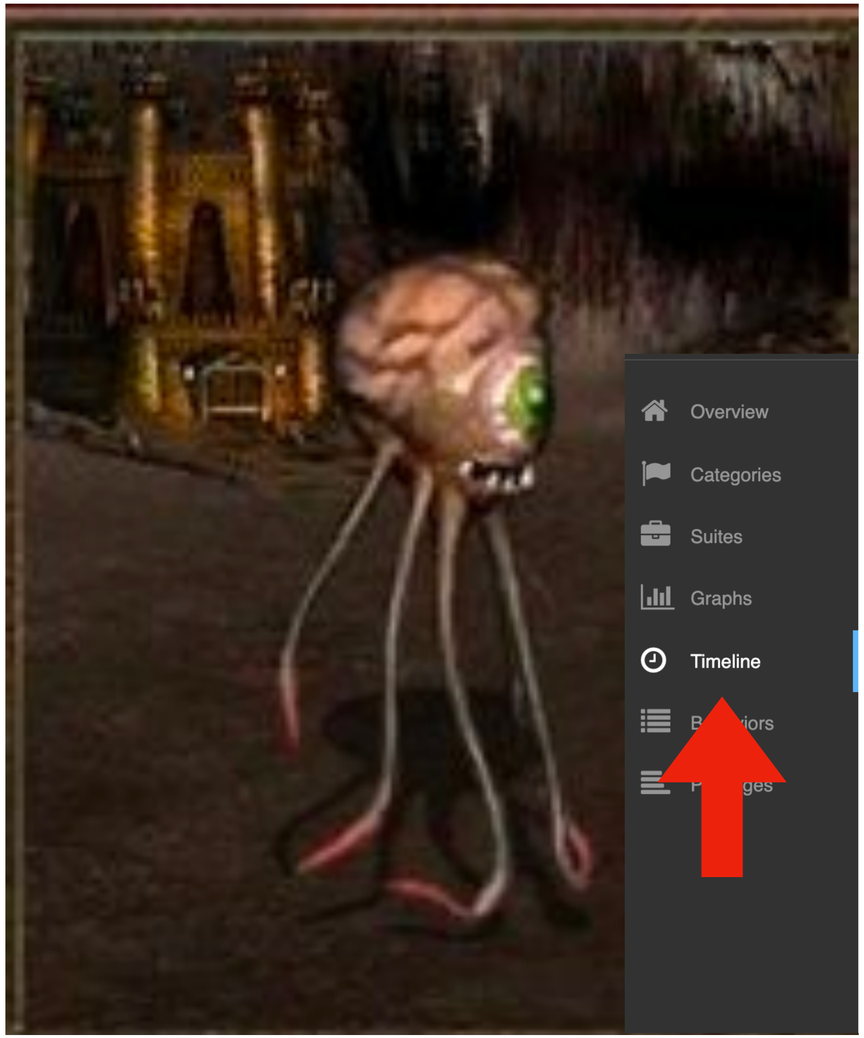


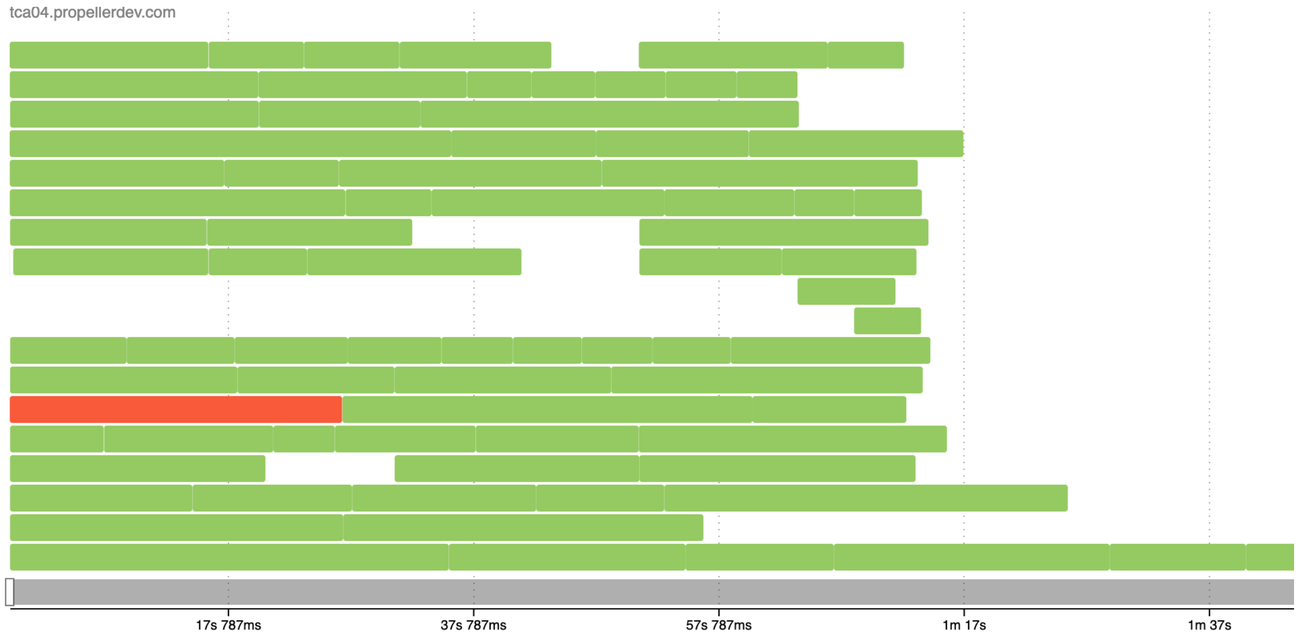
Thread confinement - что за зверь?

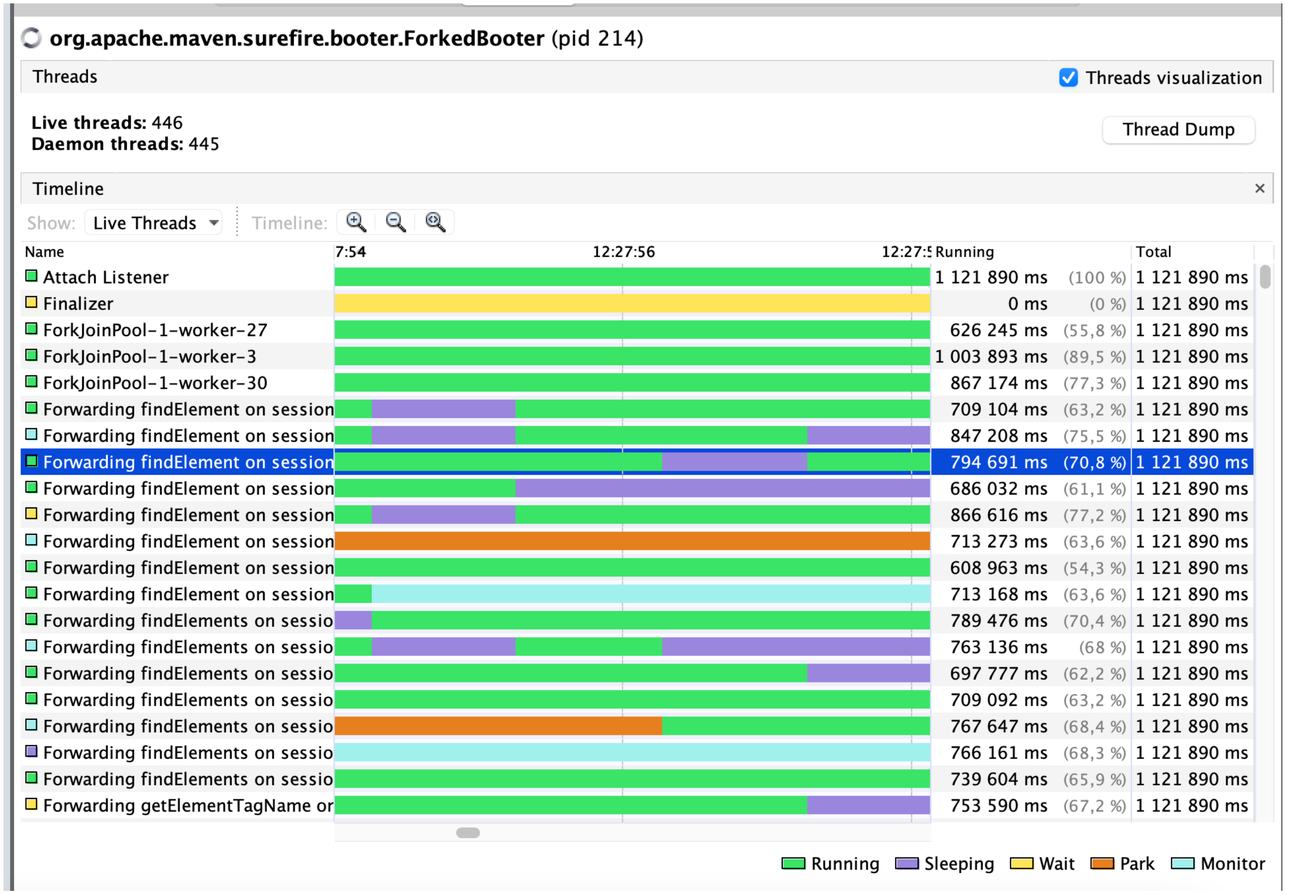
- Объект ограничен одним потоком и недоступен из других
- Часто это скрыто за реализацией DI контейнеров или фреймворков (например, Selenide)
- В Java нет примитивов и механизмов обеспечения Thread Confinement - этот принцип обеспечивается только дизайном программы
- ThreadLocal<T> может помочь, но сам по себе не обеспечивает выполнение Thread Confinement

От теории к практике

```
junit.jupiter.execution.parallel.enabled=false
junit.jupiter.execution.parallel.mode.default=concurrent
junit.jupiter.execution.parallel.mode.classes.default=concurrent
junit.jupiter.execution.parallel.config.strategy=fixed
junit.jupiter.execution.parallel.config.fixed.parallelism=10
```


Overview
Categories
Suites
Graphs
Timeline


tca04.propellerdev.com
17s 787ms
37s 787ms
57s 787ms
1m 17s
1m 37s


org.apache.maven.surefire.booter.ForkedBooter (pid 214)
Threads
Threads visualization
Live threads: 446
Daemon threads: 445
Thread Dump
Timeline
Show: Live Threads
Timeline:
Name
7:54
12:27:56
12:27:5
Running
Total
Attach Listener 1 121 890 ms (100 %) 1 121 890 ms
Finalizer 0 ms (0 %) 1 121 890 ms
ForkJoinPool-1-worker-27 626 245 ms (55,8 %) 1 121 890 ms
ForkJoinPool-1-worker-3 1 003 893 ms (89,5 %) 1 121 890 ms
ForkJoinPool-1-worker-30 867 174 ms (77,3 %) 1 121 890 ms
Forwarding findElement on session 709 104 ms (63,2 %) 1 121 890 ms
Forwarding findElement on session 847 208 ms (75,5 %) 1 121 890 ms
Forwarding findElement on session 794 691 ms (70,8 %) 1 121 890 ms
Forwarding findElement on session 686 032 ms (61,1 %) 1 121 890 ms
Forwarding findElement on session 866 616 ms (77,2 %) 1 121 890 ms
Forwarding findElement on session 713 273 ms (63,6 %) 1 121 890 ms
Forwarding findElement on session 608 963 ms (54,3 %) 1 121 890 ms
Forwarding findElement on session 713 168 ms (63,6 %) 1 121 890 ms
Forwarding findElements on sessio 789 476 ms (70,4 %) 1 121 890 ms
Forwarding findElements on sessio 763 136 ms (68 %) 1 121 890 ms
Forwarding findElements on sessio 697 777 ms (62,2 %) 1 121 890 ms
Forwarding findElements on sessio 709 092 ms (63,2 %) 1 121 890 ms
Forwarding findElements on sessio 767 647 ms (68,4 %) 1 121 890 ms
Forwarding findElements on sessio 766 161 ms (68,3 %) 1 121 890 ms
Forwarding findElements on sessio 739 604 ms (65,9 %) 1 121 890 ms
Forwarding getElementTagName or 753 590 ms (67,2 %) 1 121 890 ms
Running
Sleeping
Wait
Park
Monitor

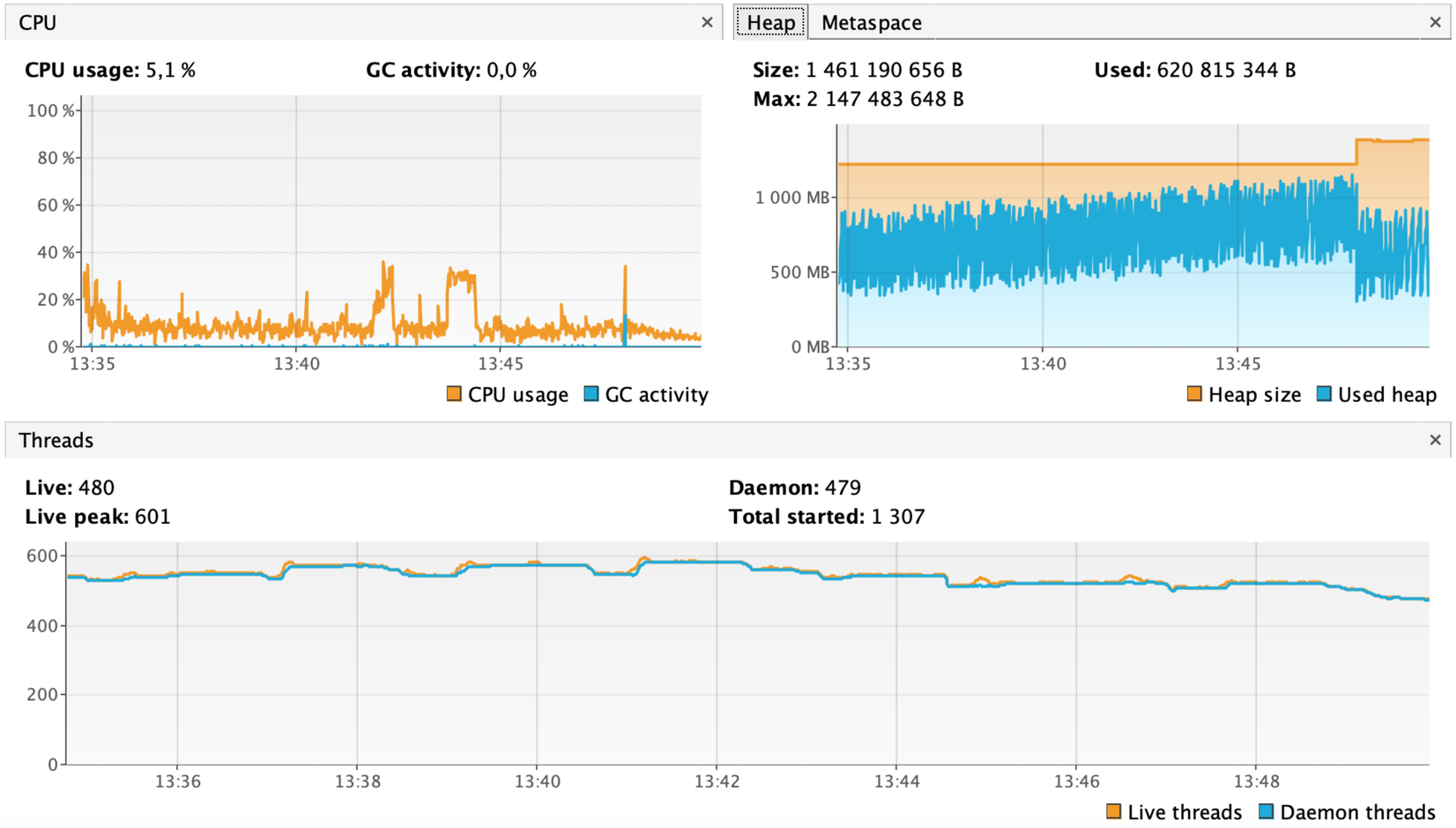
VisualVM

- На чем потоки заблокированы
- Сколько и где спят
- Чем заняты
- Thread dump !

Потоки часто чего-то ждут, освобождая процессорное время

```
"ForkJoinPool-1-worker-18" #46 daemon prio=5 os_prio=31 tid=
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at com.codeborne.selenide.Stopwatch.sleep(Stopwatch.
        at com.codeborne.selenide.impl.SelenideElementProxy.
        at com.codeborne.selenide.impl.SelenideElementProxy.
        at com.sun.proxy.$Proxy66.should(Unknown Source)
        at com.propellerads.page.BaseSspPage.checkPageLoaded
        at com.propellerads.BaseWebTest.open(BaseWebTest.jav
```

«Обычные» тесты, как правило, имеют большой i/o и дают малую нагрузку на CPU



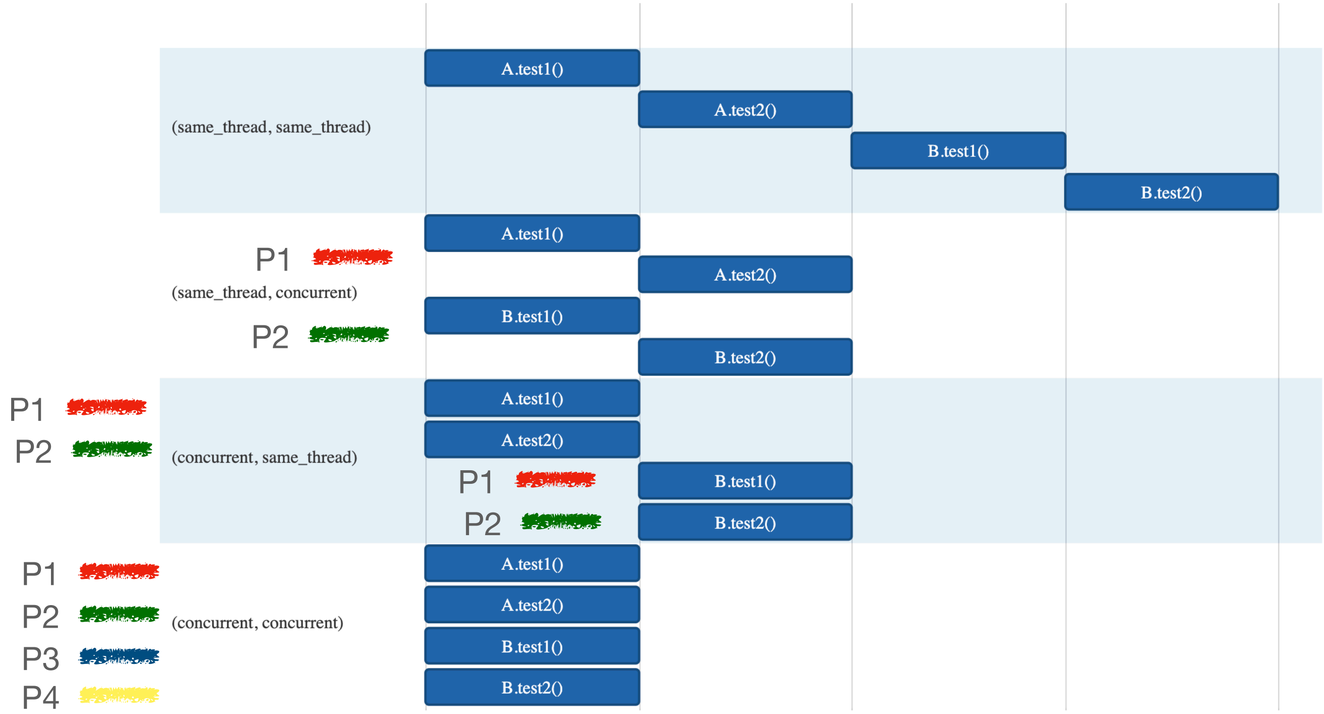
Сколько вешать concurrent в граммах?

```
junit.jupiter.execution.parallel.enabled=false
junit.jupiter.execution.parallel.mode.default=concurrent
junit.jupiter.execution.parallel.mode.classes.default=concurrent
junit.jupiter.execution.parallel.config.strategy=fixed
junit.jupiter.execution.parallel.config.fixed.parallelism=10
```

О картинках в документации

Или разговор про execution_mode

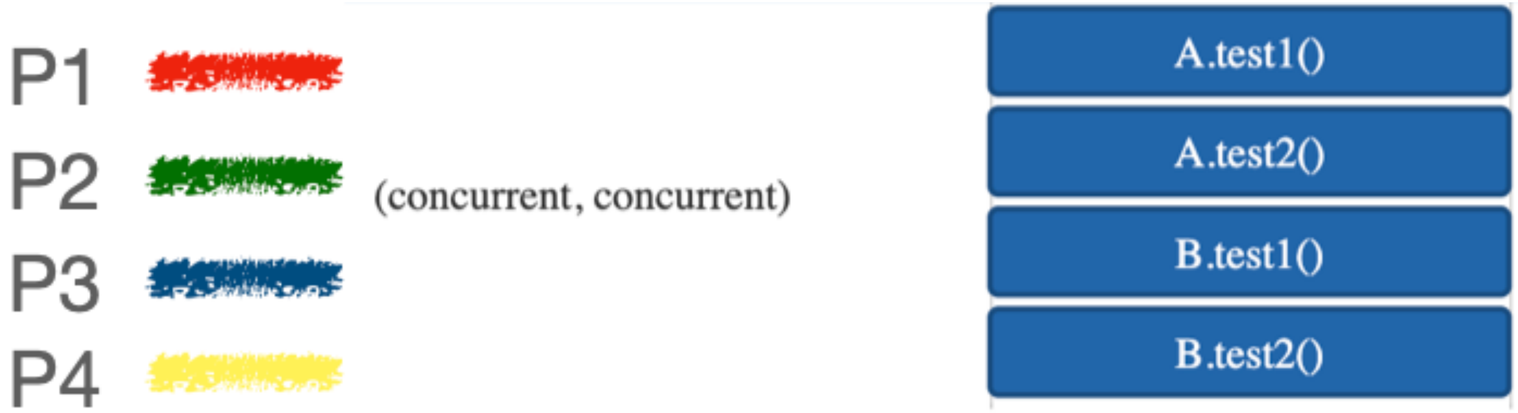
The following diagram illustrates how the execution of two top-level test classes `A` and `B` with two test methods per class behaves for all four combinations of `junit.jupiter.execution.parallel.mode.default` and `junit.jupiter.execution.parallel.mode.classes.default` (see labels in first column).



Default execution mode configuration combinations

Concurrent / Concurrent

И классы, и методы в них работают параллельно.



Concurrent / Concurrent

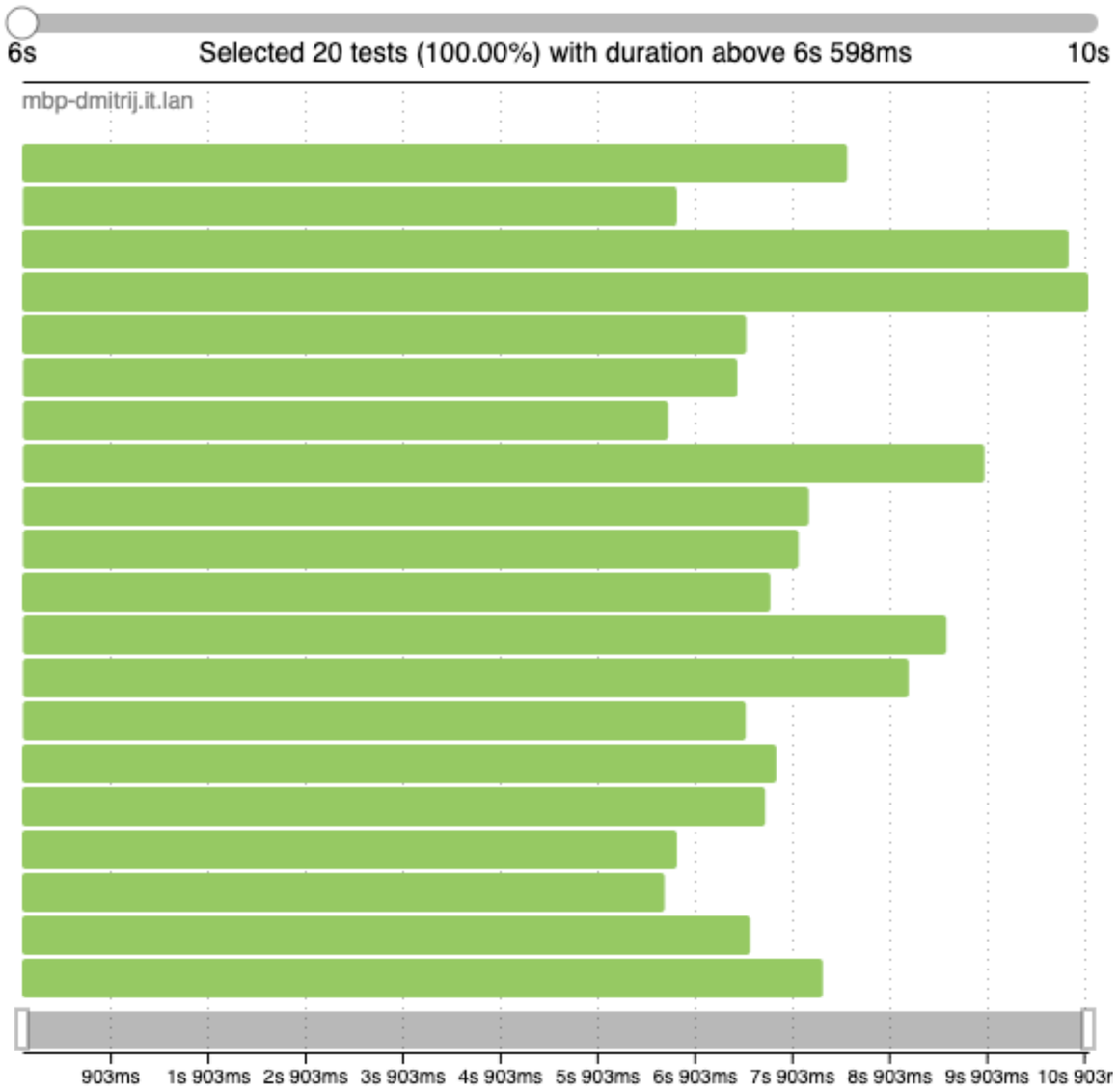
И классы, и методы в них работают параллельно.

```
@Test
void checkAboutText() {...}

@ValueSource(strings = {
        "Владимир Ситников",
        "Иван Пономарёв"
})
@ParameterizedTest
void checkExperts(String expert) {...}

@Test
void checkPartners() {...}
```

```
@ValueSource(strings = {
        "Olga",
        "Petr",
        "Anna"})
@ParameterizedTest
void sendMessageToThirdPartyService(
        String name) {...}
```

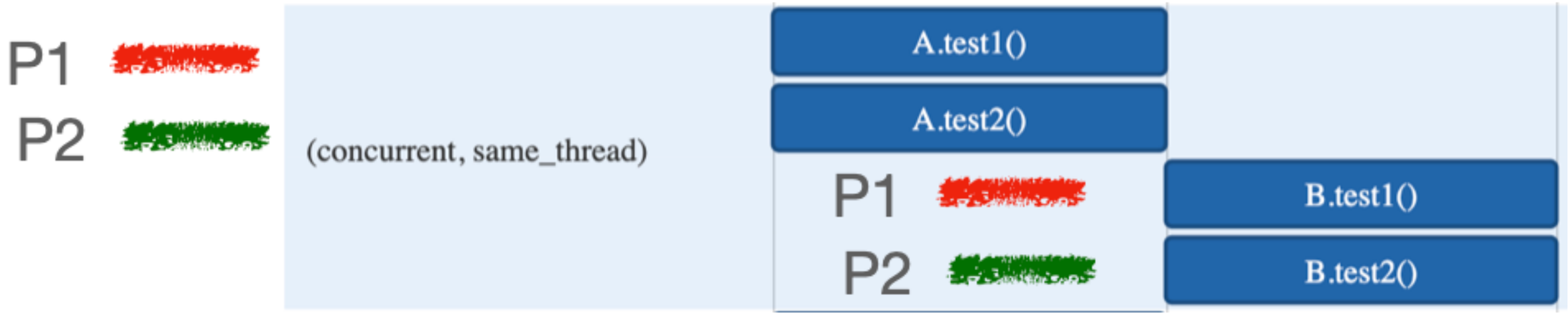
```
parallel.mode.default = concurrent

parallel.mode.classes.default = concurrent
```

- Самое эффективное использование ForkJoinPool
- Самый требовательный к синхронизации и/или выполнению Thread confinement
- Использование общих ресурсов (к примеру http-mock) затруднено



Concurrent / Same_thread

Методы в классе работают параллельно, но классы выполняются последовательно



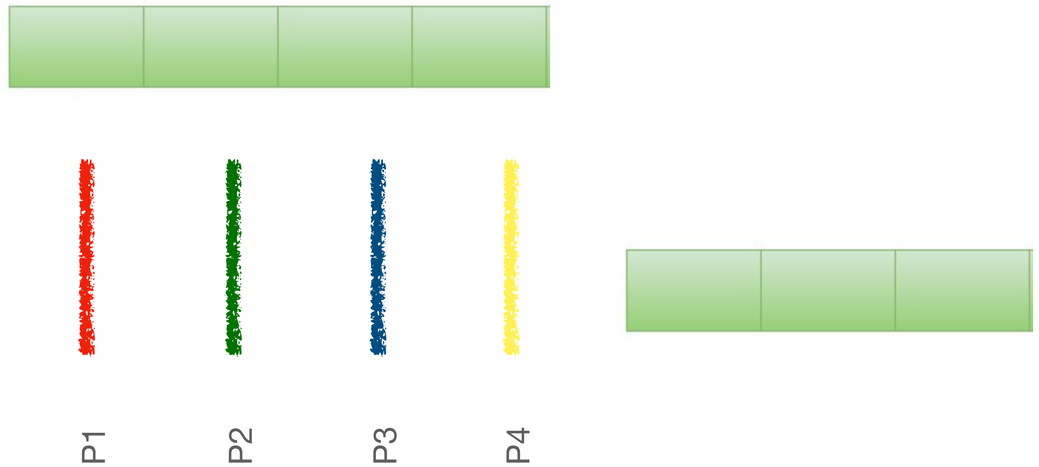
Concurrent / Same_thread

Методы в классе работают параллельно, но классы выполняются последовательно

```
@Test
void checkAboutText() {...}

@ValueSource(strings = {
        "Владимир Ситников",
        "Иван Пономарёв"
})
@ParameterizedTest
void checkExperts(String expert) {...}

@Test
void checkPartners() {...}
```

```
@ValueSource(strings = {
        "Olga",
        "Petr",
        "Anna"})
@ParameterizedTest
void sendMessageToThirdPartyService(
        String name) {...}
```

`ForkJoinPool.WorkQueue` // work-stealing mechanics



Concurrent / Same_thread

Когда и зачем?

CHROME

```
@Test
void checkAboutText() {...}

@ValueSource(strings = {
        "Владимир Ситников",
        "Иван Пономарёв"
})
@ParameterizedTest
void checkExperts(String expert) {...}

@Test
void checkPartners() {...}
```

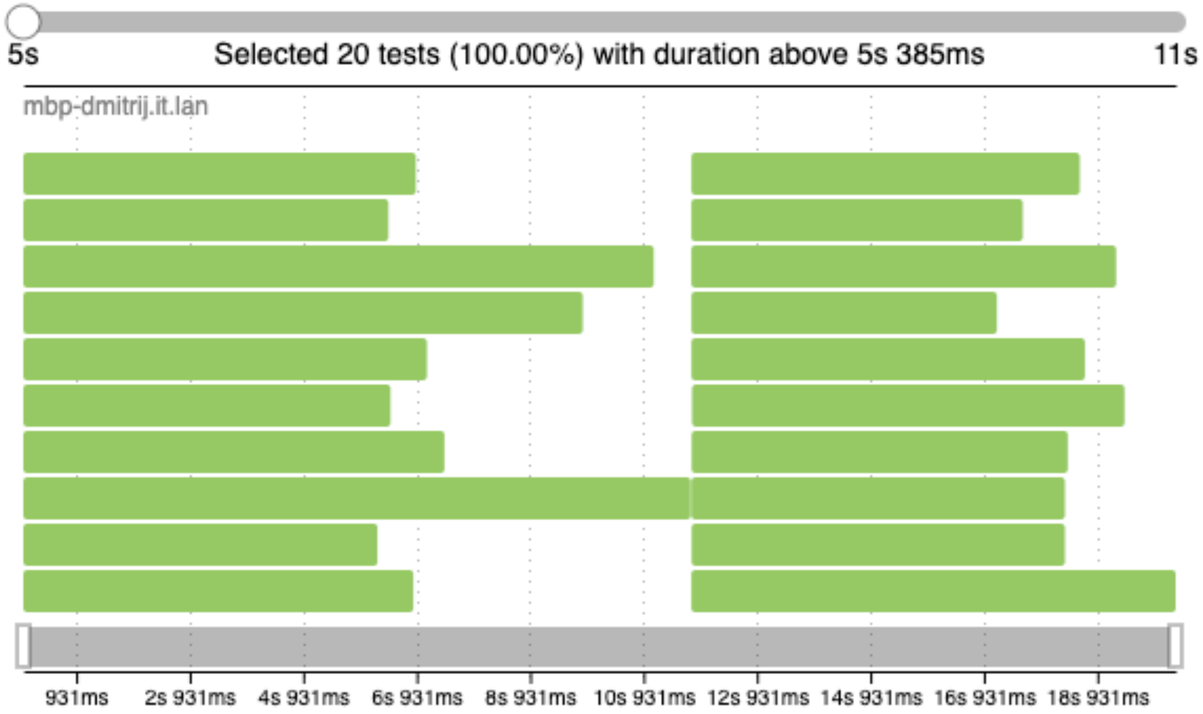
FIREFOX

```
@ValueSource(strings = {
        "Olga",
        "Petr",
        "Anna"})
@ParameterizedTest
void sendMessageToThirdPartyService(
        String name) {...}
```

CHROME -> FIREFOX

```
parallel.mode.default = concurrent

parallel.mode.classes.default = same_thread
```

- В реальной жизни у вас будет большой ForkJoinPool и использование такой конфигурации очень накладно
- Использование общих ресурсов или статических общих объектов на уровне класса легко реализуемо, но слишком дорогой ценой



Сделаем лучше: Thread Confinement + Extension

Andrei Solntsev @asolntsev 13:15

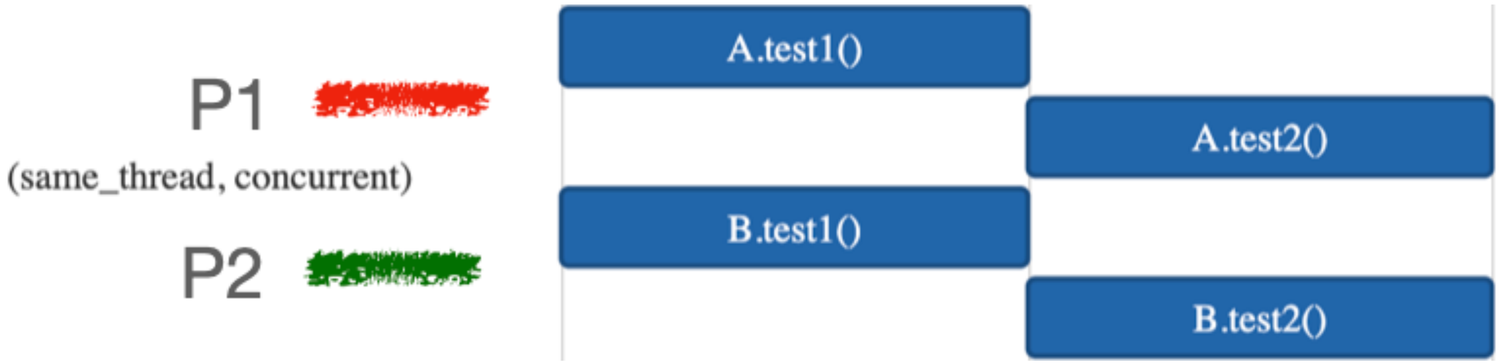
В Селениде по умолчанию всегда один активный вебдрайвер на поток.

Тебе не нужно это никак указывать, оно просто работает.

Просто не суй своё хозяйство в статические поля :)

Same_thread / Concurrent

Методы в классе выполняются последовательно, но классы выполняются параллельно

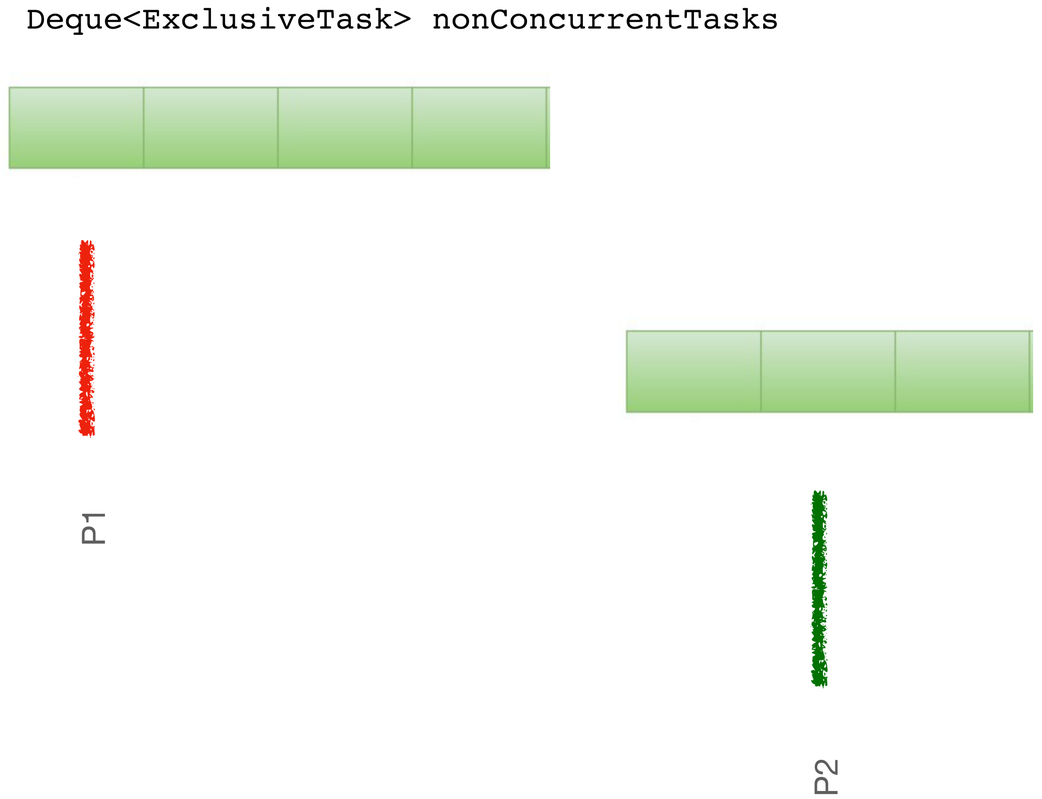


Same_thread / Concurrent

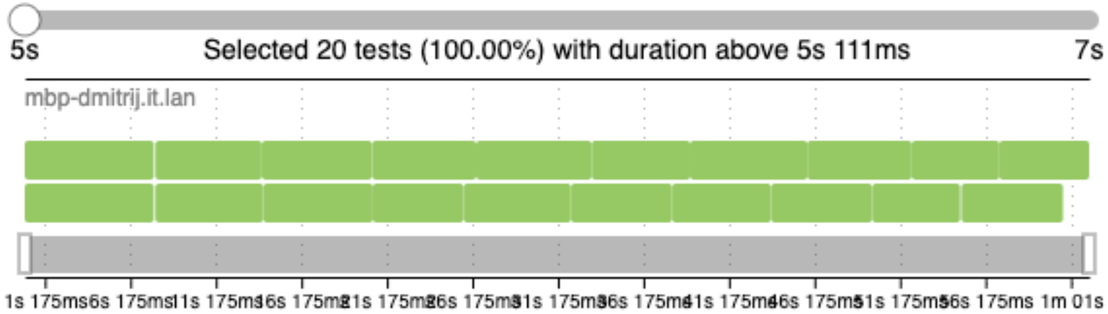
Методы в классе выполняются последовательно, но классы выполняются параллельно

```
@Test
void checkAboutText() {...}

@ValueSource(strings = {
        "Владимир Ситников",
        "Иван Пономарёв"
})
@ParameterizedTest
void checkExperts(String expert) {...}

@Test
void checkPartners() {...}
```

```
@ValueSource(strings = {
        "Olga",
        "Petr",
        "Anna"})
@ParameterizedTest
void sendMessageToThirdPartyService(
        String name) {...}
```



```
parallel.mode.default = same_thread

parallel.mode.classes.default = concurrent
```



- Близок по утилизации CPU / общему времени выполнения к concurrent/ concurrent только при условии classCount* >= threadsCount (но скорее всего, так и будет)
- Все тесты в классе выполняются в одном потоке и последовательно
- Class ordering (JUnit 5.8) здесь очень кстати
- Есть смысл использовать ограниченно, с помощью аннотации `@Isolated` или `@Execution(SAME_THREAD)`

Порядок выполнения -

Когда ClassOrder - это хорошо?

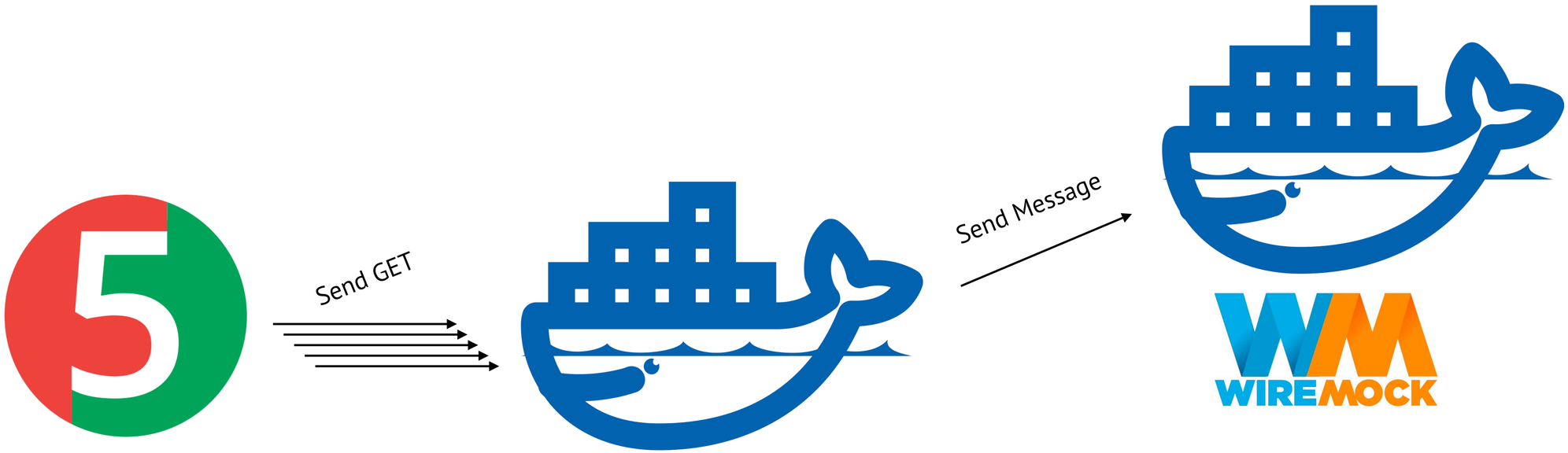
```
junit.jupiter.execution.parallel.mode.default=same_thread
junit.jupiter.execution.parallel.mode.classes.default=concurrent
```

Об изоляции и блокировках

```
@Isolated
@Execution(ExecutionMode.SAME_THREAD)
@ResourceLock("LOCK")
class Test {}
```

Или shared state может быть не только в вашем коде!

Shared state может быть не только в вашем коде




```
assertThat( А был ли мессадж? );
 assertThat( Я должен быть уверен, что его не было! );
    assertThat( А был ли мессадж? );
       assertThat( Я должен быть уверен, что его не было! );
```

О главном: правильная ООП программа

* По версии Е. Бугаенко

```
new TkFork(
  new FkRegex(
    "/css/.+",
    new TkWithType(
      new TkFork(
        new FkHitRefresh(
          "./src/main/resources/foo/scss/**",
          "mvn sass:compile", // what to run
          new TkFiles("./target/css")
        )
        new FkFixed(new TkClasspath())
      ),
      "text/css"
    )
  )
)
```

https://github.com/yegor256/takes

О главном: правильный тест

```
new Test() // это инстанс класса - свой для каждого теста
    new Listener() // это что-нибудь для Селенида, например
        new DataBaseClient() // это поле в классе - тоже свое!
            <- ThreadLocal<Connection> // это сделали за вас
                new PageObject() // это поле в классе - тоже свое!
                    new WebElement() // это внутри наших PO
                        new WebDriver() // это тест уже пошел
                            -> реально новый браузер;
```

JUnit начал эту цепочку за вас

для каждого теста. Но есть пара «но»

Extensions

```
BeforeAllCallback (1)

    @BeforeAll (2)
    LifecycleMethodExecutionExceptionHandler
    #handleBeforeAllMethodExecutionException (3)

        BeforeEachCallback (4)

            @BeforeEach (5)
            LifecycleMethodExecutionExceptionHandler
            #handleBeforeEachMethodExecutionException (6)

                BeforeTestExecutionCallback (7)

                    @Test (8)
                    TestExecutionExceptionHandler (9)

                AfterTestExecutionCallback (10)

            @AfterEach (11)
            LifecycleMethodExecutionExceptionHandler
            #handleAfterEachMethodExecutionException (12)

        AfterEachCallback (13)

    @AfterAll (14)
    LifecycleMethodExecutionExceptionHandler
    #handleAfterAllMethodExecutionException (15)

AfterAllCallback (16)
```

```
new Test()
  new Listener()
    new DataBaseClient()
      <- ThreadLocal<Connection>
        new PageObject()
          new WebElement()
            new WebDriver()
```

Extension code

User code

Extension и ArgumentConverter / Aggregator надо считать синглтонами (хотя в некоторых случаях это не так)

Они обязаны быть thread-safe.

Если игнорировать эти правила

```
32  @ParametersAreNonnullByDefault
33  public class SoftAssertsExtension implements BeforeEachCallback, AfterEachCallback {
34    private final ErrorsCollector errorsCollector;
35
36    public SoftAssertsExtension() {
37      errorsCollector = new ErrorsCollector();
38    }
39
40    public ErrorsCollector getErrorsCollector() {
41      return errorsCollector;
42    }
43
44    @Override
45    public void beforeEach(final ExtensionContext context) {
46      errorsCollector.clear();
47      addListener(LISTENER_SOFT_ASSERT, errorsCollector);
48    }
49
50    @Override
51    public void afterEach(final ExtensionContext context) {
52      removeListener(LISTENER_SOFT_ASSERT);
53      errorsCollector.failIfErrors(context.getDisplayName());
54    }
55  }
```

Разбираться в проблемах, связанных с многопоточностью - сложно

Moiseev Daniil in **Allure Framework**

привет) @eroshenkoam
ОБычный selenide-allure, запускаю тесты на selenoid параллельно через jUnit5, вот так в pom файле.
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default = concurrent
junit.jupiter.execution.parallel.config.strategy=dynamic

В итоге одна ошибка попала не в тот шаг

t.me/allure_ru/7588 Nov 25, 2020 at 22:30

Лучше просто не допускать

```
 */
@ParametersAreNonnullByDefault
public class SoftAssertsExtension implements BeforeEachCallback, AfterEachCallback {
-   private final ErrorsCollector errorsCollector;
-
-   public SoftAssertsExtension() {
-     errorsCollector = new ErrorsCollector();
-   }
-
-   public ErrorsCollector getErrorsCollector() {
-     return errorsCollector;
-   }

    @Override
    public void beforeEach(final ExtensionContext context) {
-     errorsCollector.clear();
      addListener(LISTENER_SOFT_ASSERT, errorsCollector);
    }

    @Override
    public void afterEach(final ExtensionContext context) {
      removeListener(LISTENER_SOFT_ASSERT);

      errorsCollector.failIfErrors(context.getDisplayName());
    }
}
```

```
 */
@ParametersAreNonnullByDefault
public class SoftAssertsExtension implements BeforeEachCallback, AfterEachCallback {
+   public static final ExtensionContext.Namespace namespace =
create(SoftAssertsExtension.class);

    @Override
    public void beforeEach(final ExtensionContext context) {
+     ErrorsCollector errorsCollector = new ErrorsCollector();
      addListener(LISTENER_SOFT_ASSERT, errorsCollector);
+     context.getStore(namespace).put(LISTENER_SOFT_ASSERT, errorsCollector);
    }

    @Override
    public void afterEach(final ExtensionContext context) {
      removeListener(LISTENER_SOFT_ASSERT);
+     ErrorsCollector errorsCollector = (ErrorsCollector)
context.getStore(namespace).get(LISTENER_SOFT_ASSERT);
      errorsCollector.failIfErrors(context.getDisplayName());
    }
}
```

https://github.com/selenide/selenide/pull/1334

Используйте ExtensionContext.Store для хранения и передачи состояния внутри Extension

И публичные нэймспейсы. Вам скажут спасибо.

Выводы

- Изолируйте тесты друг от друга прежде всего у себя в голове
- JUnit сделал очень многое, чтобы дать возможность использовать parallel execution не думая про проблемы с Concurrency
- Но, к сожалению, не всё
- TestEngine - главный фреймворк автоматизатора - загляните в него
- JUnit - главный TestEngine Java

Давайте писать тесты

Готовые к parallel execution, даже если пока вам это кажется не нужным.

В конце концов это просто красиво.

«Истинное искусство многопроцессорного программирования заключается в том, чтобы его избежать»

Роман Елизаров, 2012
https://elizarov.livejournal.com/22506.html

Спасибо! И вопросы

* эта картинка должна присутствовать в любом докладе про потоки. Я тоже хотел её добавить, но и без этого не успел рассказать все, что хотел. Для экономии вашего времени она здесь.