**Fork Опции и параллельное выполнение тестов для Junit4**

Выбор правильной стратегии разветвления и настроек параллельного выполнения может оказать существенное влияние на требования к памяти и время выполнения вашей системы сборки.

Surefire предлагает различные варианты параллельного выполнения тестов, что позволяет максимально эффективно использовать имеющееся в вашем распоряжении оборудование. Но, в частности, разветвление также может помочь снизить требования к памяти.

На этой странице вы найдете некоторые идеи о том, как настроить выполнение теста так, чтобы оно лучше всего подходило для вашей среды.

**Параллельное выполнение тестов**

По сути, в плагине maven-surefire-plugin есть два способа добиться параллельного выполнения тестов:

**1.** Самый очевидный из них — использование параметра parallel. Возможные значения зависят от используемого поставщика тестов. Для JUnit 4.7 и более поздних версий это могут быть methods, classes, both, suites, suitesAndClasses, suitesAndMethods, classesAndMethods or  all. В качестве предварительного условия в тестах JUnit4 исполнитель JUnit должен расширять org.junit.runners.ParentRunner. Если в аннотации @org.junit.runner.RunWith не указан ранер, необходимое условие выполнено.

Начиная с maven-surefire-plugin:2.16, значение «both» устарело, но его все еще можно использовать и оно ведет себя так же, как и classAndMethods.

Подробности смотрите на страницах примеров для JUnit и TestNG.

Расширение параллелизма настраивается с использованием следующих параметров. Параметр useUnlimitedThreads допускает неограниченное количество потоков. Если useUnlimitedThreads=true, параметр threadCount можно использовать с необязательным параметром perCoreThreadCount=true (по умолчанию true). Параметры useUnlimitedThreads и threadCount следует интерпретировать в контексте значения, указанного для параллельного параметра.

Начиная с maven-surefire-plugin:2.16, можно накладывать ограничения на количество потоков для наборов, классов или методов, используя один или несколько параметров threadCountSuites, threadCountClasses и threadCountMethods. Если указан только threadCount, плагин maven-surefire пытается оптимизировать количество потоков для наборов, классов и методов и повторно использует потоки в пользу **leaf**, например. параллельные методы (возможно увеличение количества параллельных методов).

В первом примере с неограниченным количеством потоков можно использовать максимум три параллельных потока для выполнения наборов: Parallel=all, useUnlimitedThreads=true, threadCountSuites=3.

Во втором примере количество параллельных методов строго не ограничено:parallel=classesAndMethods, threadCount=8, threadCountClasses=3. Здесь количество параллельных методов варьируется от 5 до 7. Соответственно Parallel=all, но сумма threadCountSuites и threadCountClasses не должна превышать определенную (threadCount - 1). Возможны другие комбинации с **leaf** с неуказанным числом потоков. Убедитесь, что **leaf** является последним из параллельно упорядоченных наборов-классов-методов.

В третьем примере количество потоков представляет собой соотношение, например. для parallel=all, threadCount=16, threadCountSuites=2, threadCountClasses=3, threadCountMethods=5. Таким образом, параллельные наборы составят 20%, параллельные классы — 30% и параллельные методы — 50%.

Наконец, threadCount и useUnlimitedThreads не обязательно настраиваются, если для значения parallel указано эквивалентное количество потоков.

Плагин maven-surefire пытается повторно использовать потоки, тем самым оптимизируя количество потоков и предпочитая справедливость потоков. Оптимизация parallelOptimized числа потоков включена по умолчанию, например. количество запускающих Suite не обязательно должно тратить ресурсы Threads Suite. Если используется threadCount, то **leaf** с неограниченным количеством потоков может ускориться, особенно в конце фазы тестирования.

Параметры ParallelTestsTimeoutInSeconds и ParallelTestsTimeoutForcedInSeconds используются для указания дополнительного времени ожидания при параллельном выполнении. Если время ожидания истекло, плагин печатает сводный журнал со строками ОШИБКИ: «Эти тесты были выполнены до операции выключения» и «Эти тесты не завершены», если работающие потоки были прерваны.

Примечание. По задумке исполнителей JUnit, статические методы, помеченные, например, @Parameters, @BeforeClass и @AfterClass вызываются в родительском потоке. Ради видимости памяти между потоками синхронизируйте методы. См. ключевые слова: *olatile*, *synchronized*, **immutable** and *final* в [Java Memory Model - JSR-133](https://jcp.org/en/jsr/detail?id=133).

При использовании параллельного варианта важно помнить: параллелизм происходит внутри одного и того же процесса JVM. Это эффективно с точки зрения памяти и времени выполнения, но вы можете быть более уязвимы к условиям гонки или другому неожиданному и трудно воспроизводимому поведению.

**2.** Другая возможность параллельного выполнения теста — установка для параметра forkCount значения выше 1. В следующем разделе описаны подробности об этом и связанном с ним параметре reuseForks. Использование reuseForks=true (по умолчанию) и разветвление тестовых классов в многоразовых JVM может привести к той же проблеме с общим статическим кодом в инициализаторах классов @BeforeClass при использовании parallel без разветвления. Поэтому установка reuseForks=false может помочь, однако не гарантирует правильную работу некоторых функций, например. skipAfterFailureCount.

**Параллельное выполнение тестов и выполнение одного потока**

Как упоминалось выше, выполнение теста parallel используется с определенным количеством потоков. Начиная с maven-surefire-plugin:2.18, вы можете применить аннотацию JCIP @net.jcip.annotations.NotThreadSafe к классу Java теста JUnit (чистый тестовый класс, набор, параметризованный и т. д.), чтобы выполнить его в одном Экземпляре потока. Поток имеет имя *maven-surefire-plugin@NotThreadSafe* и выполняется в конце тестового запуска.

Просто используйте зависимость проекта *net.jcip:jcip-annotations:1.0* или другой артефакт *com.github.stephenc.jcip:jcip-annotations:1.0-1* с Apache License 2.0.

1. <dependencies>
   1. [...]
   2. <dependency>
      1. <groupId>junit</groupId>
      2. <artifactId>junit</artifactId>
      3. <!-- 4.7 or higher -->
      4. <version>4.7</version>
      5. <scope>test</scope>
   3. </dependency>
   4. <dependency>
      1. <groupId>com.github.stephenc.jcip</groupId>
      2. <artifactId>jcip-annotations</artifactId>
      3. <version>1.0-1</version>
      4. <scope>test</scope>
   5. </dependency>
   6. [...]
2. </dependencies>

Таким образом, параллельное выполнение классов тестов, аннотированных @NotThreadSafe, разветвляется в экземпляре одного потока (не имеется в виду разветвленный процесс JVM).

Если *Suite* или *Parameterized* аннотирован с помощью @NotThreadSafe, классы набора выполняются в одном потоке. Вы также можете аннотировать отдельный тестовый класс, на который ссылается Suite, а другие неаннотированные тестовые классы в Suite могут запускаться параллельно. Таким образом, вы можете изолировать конфликтующие группы тестов и при этом запускать отдельные тесты параллельно.

Примечание. По задумке исполнителей JUnit, статические методы, помеченные, например, @Parameters, @BeforeClass и @AfterClass вызываются в родительском потоке. Назначьте классы @NotThreadSafe Suite, чтобы избежать этой проблемы. Если вы не хотите менять иерархию своих тестовых классов, вы можете синхронизировать такие методы ради улучшения видимости памяти в качестве упрощенного решения. См. ключевые слова: *volatile*, *synchronized*, **immutable** and *final* в [Java Memory Model - JSR-133](https://jcp.org/en/jsr/detail?id=133).

**Параллельное выполнение набора тестов (Test-Suite)**

Например, предположим, что ваши тесты расположены следующим образом:

1. src/test/java
2. +-features.areaA
3. | +-SomeTest.java
4. | +-AnotherTest.java
5. +-features.areaB
6. | +-DifferentTest.java
7. | +-OtherTest.java

Вы должны добавить TestSuite.java для каждого пакета, который будет выглядеть примерно так:

1. package features.areaA;
3. import org.junit.runner.RunWith;
4. import org.junit.runners.Suite;
6. @RunWith(Suite.class)
7. @Suite.SuiteClasses({SomeTest.class,AnotherTest.class})
8. public class TestSuite {
9. }

и аналогично для пакета Features.areaB.

Затем вы настроите плагин maven-surefire следующим образом:

1. <plugin>
   1. <groupId>org.apache.maven.plugins</groupId>
   2. <artifactId>maven-surefire-plugin</artifactId>
   3. <version>3.2.5</version>
   4. <configuration>
      1. <includes>
         1. <include>features.\*.TestSuite</include>
      2. </includes>
      3. <parallel>suites</parallel>
      4. <threadCountSuites>2</threadCountSuites>
      5. <perCoreThreadCount>false</perCoreThreadCount>
   5. </configuration>
2. </plugin>

Затем будет выполняться до двух потоков, и каждый поток будет получать *TestSuite.class* по одному за раз. Тестовые классы в этом наборе будут выполняться последовательно в порядке, указанном для класса Suite.

Обратите внимание, что perCoreThreadCount является своего рода «множителем». Если для этого параметра установлено значение true (значение по умолчанию), вы получите указанное количество потоков на одно ядро ЦП.

Также обратите внимание, что в данном конкретном случае threadCountSuites=2 имеет тот же эффект, что и threadCount=2, поскольку мы используем Parallel=Suites.

**Параллельное выполнение maven-surefire-plugin в многомодульной параллельной сборке Maven**

Ядро Maven позволяет параллельно создавать модули многомодульных проектов с помощью параметра командной строки -T. Это умножает степень параллелизма, настроенную непосредственно в плагине maven-surefire-plugin.

**Разветвленное выполнение теста**

Параметр forkCount определяет максимальное количество процессов JVM, которые maven-surefire-plugin будет запускать одновременно для выполнения тестов. Он поддерживает тот же синтаксис, что и -T в maven-core: если вы завершите значение буквой «C», это значение будет умножено на количество доступных ядер ЦП в вашей системе. Например, forkCount=2.5C в четырехъядерной системе приведет к разделению до десяти одновременных процессов JVM, выполняющих тесты.

Параметр reuseForks используется, чтобы определить, следует ли завершить порожденный процесс после одного тестового класса и создать новый процесс для следующего теста в строке (reuseForks=false) или повторно использовать процессы для выполнения следующих тестов (reuseForks=true) ).

Значение по умолчанию — forkCount=1/reuseForks=true, что означает, что плагин maven-surefire-plugin создает один новый процесс JVM для выполнения всех тестов в одном модуле Maven.

forkCount=1/reuseForks=false выполняет каждый тестовый класс в своем собственном процессе JVM, один за другим. Это создает самый высокий уровень разделения при выполнении тестов, но, вероятно, также обеспечит самое продолжительное время выполнения из всех доступных вариантов. Рассматривайте это как крайнюю меру.

С помощью свойства argLine вы можете указать дополнительные параметры, которые будут переданы в ответвленный процесс JVM, например настройки памяти. Переменные системных свойств из основного процесса maven также передаются в раздвоенный процесс. Кроме того, вы можете использовать элемент systemPropertyVariables для указания переменных и значений, которые будут добавлены к свойствам системы во время выполнения теста.

Вы можете использовать заполнитель ${surefire.forkNumber} в argLine, EnvironmentVariables (начиная с maven-surefire-plugin:3.2.0) или в свойствах системы (как тех, которые указаны через mvn test -D..., так и через systemPropertyVariables). Перед выполнением тестов плагин Surefire заменяет этот заполнитель номером фактически выполняющегося процесса, считая от 1 до эффективного значения forkCount, умноженного на максимальное количество параллельных выполнений в параллельных сборках Maven, то есть эффективное значение команды -T. строковый аргумент ядра Maven.

В случае отключенного разветвления (forkCount=0) заполнитель будет заменен на 1.

Ниже приведен пример конфигурации, в которой используется до трех разветвленных процессов, которые выполняют тесты и затем завершаются. Системное свойство DatabaseSchema передается процессам, которые должны указать схему базы данных, которая будет использоваться во время тестов. Значениями для этого будут *MY\_TEST\_SCHEMA\_1, MY\_TEST\_SCHEMA\_2 и MY\_TEST\_SCHEMA\_3* для трех процессов. Кроме того, если указать собственный рабочий каталог, каждый из процессов будет выполняться в отдельном рабочем каталоге, чтобы обеспечить изоляцию на уровне файловой системы.

1. <plugins>
2. [...]
   1. <plugin>
      1. <groupId>org.apache.maven.plugins</groupId>
      2. <artifactId>maven-surefire-plugin</artifactId>
      3. <version>3.2.5</version>
      4. <configuration>
         1. <forkCount>3</forkCount>
         2. <reuseForks>true</reuseForks>
         3. <argLine>-Xmx1024m -XX:MaxPermSize=256m</argLine>
         4. <systemPropertyVariables>
         5. <databaseSchema>MY\_TEST\_SCHEMA\_${surefire.forkNumber}</databaseSchema>
         6. </systemPropertyVariables>
         7. <workingDirectory>FORK\_DIRECTORY\_${surefire.forkNumber}</workingDirectory>
      5. </configuration>
   2. </plugin>
3. [...]
4. </plugins>

В случае многомодульного проекта с тестами в разных модулях вы также можете использовать, скажем, mvn -T 2 ... для запуска сборки, что даст значения ${surefire.forkNumber} в диапазоне от 1 до 6.

Представьте, что вы выполняете несколько тестов, использующих контекст JPA, который имеет значительное начальное время запуска. Установив reuseForks=true, вы можете повторно использовать этот контекст для последовательных тестов. А поскольку многие тесты, как правило, используют одни и те же тестовые данные и получают к ним доступ, вы можете избежать блокировок базы данных во время одновременного выполнения, используя отдельные, но унифицированные схемы базы данных.

Номера портов и имена файлов — это другие примеры ресурсов, совместное использование которых при одновременном выполнении тестов может оказаться затруднительным или нежелательным.

И т.д. на странице <https://maven.apache.org/surefire/maven-surefire-plugin/examples/fork-options-and-parallel-execution.html>