Jenkins

Coding Continuous Delivery – Performance-Optimierung für die Jenkins-Pipeline

Johannes Schnatterer, Triology GmbH

Nachdem in der letzten Ausgabe [1] die Grundbegriffe und eine erste Jenkins-Pipeline beschrieben wurden, zeigt dieser Artikel wie man mit Parallelisierung und Nightly Builds die Laufzeit der Pipelines verkürzen und damit schnelleres Feedback erhalten kann.

Im Folgenden werden die Pipeline-Beispiele aus der letzten Ausgabe sukzessive erweitert, um die Features der Pipeline zu zeigen. Dabei sind die Änderungen jeweils in „declarative"- und in „scripted"-Syntax realisiert. Der aktuelle Stand jeder Erweiterung lässt sich bei GitHub *[2]* nachverfolgen und ausprobieren. Hier gibt es für jeden Abschnitt unter der in der Überschrift genannten Nummer jeweils für „declarative" und „scripted" einen Branch, der das vollständige Beispiel umfasst. Das Ergebnis der Builds jedes Branch lässt sich außerdem direkt auf unserer Jenkins-Instanz *[3]* einsehen.

Wie im ersten Teil sind auch hier die Features des Jenkins-Pipeline-Plug-ins anhand eines typischen Java-Projekts gezeigt. Als Beispiel dient damit auch diesmal der „kitchensink"-Quickstart von WildFly. In der letzten Ausgabe wurden mit simpler Pipeline, eigenen Steps, Stages, Fehlerbehandlung und Properties/Archivierung fünf Beispiele gezeigt.

Parallelisierung

Dank des „parallel"-Steps ist es sehr einfach, Steps oder Stages nebenläufig auszuführen und so die Gesamtlaufzeit der Pipeline zu verkürzen. Am einfachsten ist es, dabei die Schritte parallel auf einem Node auszuführen. Dabei muss jedoch bedacht werden, dass die nebenläufigen Ausführungen im selben Verzeichnis laufen und sie sich dadurch unerwartet gegenseitig beeinflussen könnten. Beispielweise löschen zwei nebenläufig ausgeführte Maven Builds mit „clean"-Phase dasselbe „target"-Verzeichnis, wodurch ein Build wahrscheinlich scheitert.

Alternativ kann man auch mit mehreren Nodes parallelisieren, also mehrere Jenkins Build Executors belegen. Dies verursacht jedoch deutlich mehr Overhead. Es muss dann auf jedem Node der Workspace neu angelegt werden – also Git Clone, gegebenenfalls Maven Dependencies geladen werden etc. Daher ist die Parallelisierung innerhalb eines Node in den meisten Fällen die erste Wahl.

An dieser Stelle eine kleine Begebenheit aus der Praxis zur Motivation für die Parallelisierung: In einem kleineren Projekt (etwa zehn KLOC) mit hoher Testabdeckung (rund 80 Prozent) konnte die Build-Dauer von elf auf acht Minuten verkürzt werden, indem die Unit- und Integrationstest-Stages parallel ausgeführt wurden. Darüber hinaus ließ sich das ursprünglich sequenzielle Deployment von zwei Artefakten durch Parallelisierung von vier auf zweieinhalb Minuten verkürzen. Es lohnt sich also, den „parallel"-Step im Hinterkopf zu behalten, um mit wenig Aufwand schnelleres Feedback zu bekommen. Das Beispiel in *Listing 1* zeigt in „scripted"-Syntax, wie man die Unit- und Integrationstest-Stages gleichzeitig ausführt.

Dem „parallel"-Step wird eine Map übergeben, in der man die verschiedenen Ausführungszweige mit Namen versehen kann und in einem Closure definiert. Anhand des Namens können dann bei der Ausführung die Ausgaben im Log den jeweiligen Zweigen zugeordnet werden. Hier spielt das schon im ersten Teil erwähnte Blue-Ocean-Plug-in seine Stärken aus: Statt wie in der klassischen Ansicht die Pipeline in einer Reihe anzuzeigen , werden gleichzeitig ausgeführte Zweige untereinander dargestellt *(siehe Abbildung 1)*. Außerdem kann man sich die Konsolen-Ausgabe der jeweiligen Branches isoliert ansehen und muss nicht wie in der klassischen Ansicht anhand des in der Pipeline vergebenen Namens unterscheiden *(siehe Listing 2)*.

Der „parallel"-Step kann auch in der „declarative"-Syntax verwendet werden. *Listing 3* zeigt, wie sich dies über geschachtelte Stages abbil-

den lässt. Für komplexere Szenarien sind zur Synchronisierung mehrerer gleichzeitiger Builds Locks *[4]* und/oder das Milestone-Plug-in *[5]* möglich. Unter *[6]* steht ein Beispiel, das beides verwendet.

Nightly Builds

Viele Teams lassen ihren CI-Server lang laufende oder regelmäßig aufzuführende Aufgaben einmal am Tag erledigen, typischerweise über Nacht. Ein Beispiel dafür ist die Überprüfung ihrer Dependencies auf bekannte Sicherheitslücken *[7]*. Diese Nightly Builds sind auch in einer Jenkins-Pipeline möglich. Einen Build regelmäßig auszuführen, ist dabei sehr einfach. *Listing 4* zeigt, wie dies in der „scripted"-Syntax über die im ersten Artikel beschriebenen Properties angegeben wird.

Bei der Ausführung des Jenkins-Files werden dann die angegebenen Ausführungen von Jenkins eingeplant. In *Listing 4* ist zu sehen, dass – wie in einer an Cron-Jobs angelehnten Syntax – ein Schedule angegeben werden kann:

- Minute (0-59)
- Stunde (0-23)
- Tag des Monats (1-31)
- Monat (1-12)
- Tag der Woche (0-7), 0 und 7 sind Sonntag

Der Asterisk (*) steht für jeden validen Wert; in *Listing 4* bedeutet das beispielsweise „jeden Tag in jedem Monat". Das „H" steht für den Hash des Job-Namens. Dabei wird ein Zahlenwert auf Basis des Hashwerts des Job-Namens generiert. Dies führt dazu, dass nicht alle Jobs mit gleichem Schedule zu Lastspitzen führen. So würde „0 0 * * 1-5" dazu führen, dass an jedem Werktag um Punkt 0 Uhr alle Jobs gleichzeitig starten. „H H(0-3) * * 1-5" hingegen verteilt diese Last auf die Zeit zwischen 0 und 3 Uhr. Diese Best Practice sollte so oft wie möglich zum Einsatz kommen.

Wem das zu kompliziert ist, dem stehen die Aliase „@yearly", „@annually", „@monthly", „@weekly", „@daily", „@midnight" und „@hourly" zur Verfügung. Auch diese nutzen zur Lastverteilung das oben beschriebene Hash-System. Beispielsweise bedeutet „@midnight" konkret die Zeit zwischen 0 und 2:59 Uhr. Auf die gleiche Weise spezifiziert man den Schedule auch in der „declarative"-Syntax *(siehe Listing 5)*, allerdings wird diese hier in einer eigenen „triggers"-Directive angegeben.

Die größere Herausforderung liegt allerdings darin zu entscheiden, wo man die nächtlich auszuführende Logik beschreibt. Dafür bieten sich zwei Möglichkeiten an: Man legt ein weiteres Jenkins-File im Repository an, beispielsweise „Jenkinsfile-nightly". Außerdem kreiert man einen neuen Pipeline- oder Multibranch-Pipeline-Job in

```
parallel(
  unitTest: {
     stage('Unit Test') {
        mvn 'test'
     }
  },
  integrationTest: {
        stage('Integration Test') {
           mvn 'verify -DskipUnitTests -Parq-wildfly-swarm '
        }
     }
)
```

Listing 1

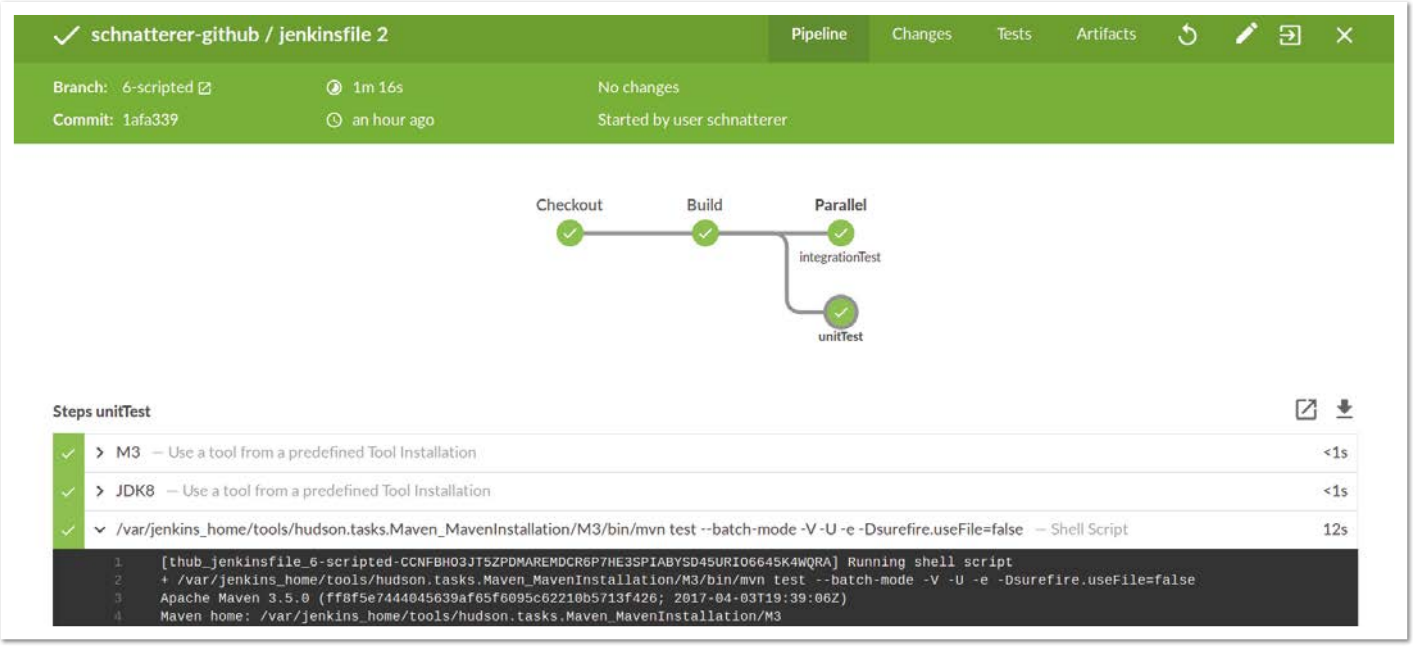

Abbildung 1: Parallelisierung und Log im Blue-Ocean-Theme

```
[integrationTest] [thub_jenkinsfile_6-scripted-CCNFBHO3JT5ZPDMAREMDCR6P7HE3SPIABYSD45URIO6645K4WQRA] Running shell
script
[integrationTest] + /var/jenkins_home/tools/hudson.tasks.Maven_MavenInstallation/M3/bin/mvn verify -DskipUnitTests
-Parq-wildfly-swarm --batch-mode -V -U -e -Dsurefire.useFile=false
[unitTest] [thub_jenkinsfile_6-scripted- CCNFBHO3JT5ZPDMAREMDCR6P7HE3SPIABYSD45URIO6645K4WQRA] Running shell script
[unitTest] + /var/jenkins_home/tools/hudson.tasks.Maven_MavenInstallation/M3/bin/mvn test --batch-mode -V -U -e
-Dsurefire.useFile=false
```

Listing 2

```
stage('Tests') {
  parallel {
     stage('Unit Test') {
        steps {
           mvn 'test'
        }
     }
     stage('Integration Test') {
        steps {
           mvn 'verify -DskipUnitTests -Parq-wildfly-swarm '
        }
     }
  }
}
```

Listing 3

```
properties([
  pipelineTriggers([cron('H H(0-3) * * 1-5')])
])
```

Listing 4

```
pipeline {
  agent any
  triggers {
     cron('H H(0-3) * * 1-5')
  }
}
```

Listing 5

```
boolean isTimeTriggered = isTimeTriggeredBuild()
node {
  // ...
  stage('Integration Test') {
     if (isTimeTriggered) {
        mvn 'verify -DskipUnitTests -Parq-wildfly-swarm '
     }
  }
  //...
}
boolean isTimeTriggeredBuild() {
  for (Object currentBuildCause : script.currentBuild.rawBuild.getCauses()) {
     return currentBuildCause.class.getName().contains(‚TimerTriggerCause‘)
  }
  return false
}
```

Listing 6

Jenkins und gibt dort erneut das Repository sowie den Namen des neuen Jenkins-Files an, das gelesen werden soll. Der Vorteil ist, dass dies einfach aufzusetzen ist und eine gewisse Separation of Concerns bietet.

Statt eines monolithischen Jenkins-Files mit sehr vielen Stages, die abhängig vom Trigger ausgeführt werden, hat man zwei Jenkins-Files, bei denen immer alle Stages durchlaufen werden. Dies steht allerdings im Gegensatz zum Pipeline-Gedanken, bei dem jeder Build immer die gleichen Stages durchläuft. Zudem lässt sich bei den beiden Pipelines eine gewisse Redundanz nicht vermeiden. Beispielsweise benötigt man meist in beiden die Build-Stage. Dieser Redundanz kann man zwar mit Shared Libraries oder dem „load"-Step begegnen (siehe unten), trotzdem erhöht dies den Aufwand und die Komplexität.

Außerdem ist ein zusätzlicher Job schwerer zu verwalten, insbesondere wenn man Multibranch-Pipeline-Jobs oder gar eine GitHub-Organization betreibt. Dort werden pro Repository und Branch dynamisch neue Jobs angelegt *(siehe [1])*. Man hat dann mehrere Multibranch-Build-Jobs, bei GitHub-Organizations sogar einen weiteren Multibranch-Build-Job pro Repository, das in den Nightly Builds enthalten ist.

Alternativ pflegt man alle Stages in einem Jenkins-File und unterscheidet, welche Stages immer und welche nur nächtlich ausgeführt werden. Man kann zudem festlegen, welche Branches nächtlich gebaut werden sollen. Dieser Ansatz entspricht dem Pipeline-Gedanken. Hier hat man dann die vollen Vorteile von Multibranch-Pipeline-Jobs, da jeder dynamisch generierte Branch, falls gewünscht, auch nächtlich gebaut wird. Im Moment hat dieser Ansatz noch den großen Nachteil, dass das Abfragen des Build-Triggers umständlich ist.

Aufgrund der beschriebenen Nachteile eines weiteren Jobs liegt hier die Unterscheidung innerhalb des Jobs näher. Ähnlich wie bei den klassischen Freestyle-Jobs kann man in der Pipeline die Auslöser des Builds („Build Causes") abfragen. Im Moment geht dies jedoch nur über das „currentBuild.rawBuild"-Objekt. Den Zugriff darauf muss man sich von einem Jenkins-Administrator freigeben lassen („Script Approval"). Dies ist per Web auf *„https://JENKINSURL/scriptApproval/"* oder direkt im Dateisystem unter „JENKINS_HOME/script-approval.xml" möglich.

Jenkins empfiehlt, den Zugriff auf das Objekt aus Sicherheitsgründen nicht freizugeben. Trotzdem ist dies derzeit die einzige Methode, an die Build Causes zu kommen *[8]*. Die Lösung dafür ist allerdings schon unterwegs: Perspektivisch wird man die Build Causes direkt über das „currentBuild"-Objekt abfragen können *[9]*, wofür kein Script Approval notwendig sein wird. Dies vorausgesetzt, kann man beispielsweise die Ausführung der Integrationstests auf den Nightly Build beschränken *(siehe Listing 6)*.

Auch hier ist die Logik für das Abfragen des Build Cause in einen eigenen Step ausgelagert. Auffällig ist, dass dieser Step außerhalb des Node aufgerufen wird. Man könnte diesen auch direkt in der „Integration Test"-Stage abrufen. Allerdings würde man dann nicht die Aufforderung zum Script Approval bekommen, da sie vom Build Executor (Node) nicht zurück zum Master übertragen wird. Um dieses Beispiel auszuführen, sind die beiden Einträge aus *Listing 7* in der „script-approval.xml" erforderlich.

Wie beschrieben, kann man die Methoden auch per Web freigeben. Allerdings müssen die Methoden einzeln nacheinander zugelassen werden:

- Build ausführen
- Fehlschlag
- „getRawBuild" zulassen, Build ausführen
- Fehlschlag
- „getCauses" zulassen, Build ausführen
- Erfolg

Wem das zu kompliziert ist, der kann sich auch mit dem Workaround aus *Listing 8* behelfen. Hier wird einfach anhand der Uhrzeit entschieden, ob der Build in der Nacht läuft. Dies hat den Nachteil, dass auch anders angestoßene Builds (beispielsweise durch SCM oder manuell gestartet) als Nightly Build betrachtet werden. Außerdem ist hier darauf zu achten, dass sich die Zeiten auf die Zeitzone des Jenkins-Servers beziehen. Unabhängig vom Build Cause hat man immer die Möglichkeit, nur bestimmte Branches in der Nacht zu bauen. *Listing 9* zeigt, wie das in „scripted"-Syntax aussieht.

In der „declarative"-Syntax lässt sich dasselbe erreichen, ist aber anders auszudrücken. Hier gibt es die „when"-Directive, um über die Ausführung einer Stage zu entscheiden. In dieser kann man jedoch nur die von Jenkins oder einer Shared Library (siehe unten) bereitgestellten Steps verwenden. Innerhalb der Pipeline definierte Steps (wie „isTimeTriggeredBuild()" oder „isNightly()") kann man nicht aufrufen. Allerdings lässt sich der Code der Methoden dort direkt verwenden *(siehe Listing 10)*.

Ein Vorteil der „declarative"-Syntax ist die bessere Integration in Blue Ocean. Wird eine Stage aufgrund des negativen Ergebnisses

```
  <approvedSignatures>
     <string>method hudson.model.Run getCauses</string>
     <string>method org.jenkinsci.plugins.workflow.
support.steps.build.RunWrapper getRawBuild</string>
  </approvedSignatures>
```

Listing 7

```
boolean isNightly() {
  return Calendar.instance.get(Calendar.HOUR_OF_DAY) in 0..3
}
```

Listing 8

```
node {
  properties([
     pipelineTriggers(createPipelineTriggers())
  ])
  // ...
}
def createPipelineTriggers() {
  if (env.BRANCH_NAME == 'master') {
     return [cron('H H(0-3) * * 1-5')]
  }
  return []
}
```

Listing 9

```
stage('Integration Test') {
  when { expression { return Calendar.instance.get(Calendar.HOUR_OF_DAY) in 0..3 } }
  steps {
     mvn 'verify -DskipUnitTests -Parq-wildfly-swarm '
  }
}
```

Listing 10

Abbildung 2: Mittels „declarative"-Syntax übersprungene Stage in Blue Ocean

der „when"-Directive übersprungen, ist dies in Blue Ocean entsprechend visualisiert *(siehe Abbildung 2)*. Dies ist mit „scripted"-Syntax nicht möglich.

Es ist nicht intuitiv, mit der „declarative"-Syntax nur bestimmte Branches in der Nacht zu bauen, denn die gezeigte „triggers"-Directive kann nicht konditional ausgeführt werden. Als Notlösung bieten sich der „script"-Step oder das Aufrufen selbst definierter Steps an – dort lässt sich „scripted"-Syntax ausführen. Damit kann man sich die Trigger in den „scripted"-Properties festlegen. Nachteil ist, dass dies nur innerhalb einer Stage ausgeführt werden kann *(siehe Listing 11)*.

Fazit und Ausblick

Dieser Artikel zeigt, wie man die Ausführungszeit der Pipeline verkürzen kann. Mit Parallelisierung ist dies sehr einfach möglich. Eine weitere Möglichkeit ist die Auslagerung lang laufender Stages in den Nightly Build. Sie ist derzeit aber noch aufwendig.

Die Konsolidierung der Pipeline ist ein Vorgeschmack auf den Artikel in der nächsten Ausgabe, in dem mit Shared Libraries und Docker weitere nützliche Werkzeuge vorgestellt werden. Diese vereinfachen den Umgang mit Pipelines durch Wiederverwendung über verschiedene Jobs hinweg, Unit-Testing des Pipeline-Codes und den Einsatz von Containern.

```
Pipeline {
// ...
  stages {
     stage('Build') {
        steps {
           // ...
           createPipelineTriggers()
        }
     }
  }
  // ...
}
void createPipelineTriggers() {
  script {
     def triggers = []
     if (env.BRANCH_NAME == 'master') {
        triggers = [cron('H H(0-3) * * 1-5')]
     }
     properties([
        pipelineTriggers(triggers)
     ])
  }
}
```

Listing 11

Weitere Informationen

[1] J.Schnatterer, D.Behrwind, Coding Continuous Delivery – Grundlagen des Jenkins Pipeline Plug-ins", Java aktuell 01/2018
[2] Jenkinsfile Repository GitHub: *https://github.com/triologygmbh/jenkinsfile*
[3] Triology Open Source Jenkins: *https://opensource.triology.de/jenkins/job/triologygmbh-github/job/jenkinsfile/*
[4] Locks: *https://jenkins.io/doc/pipeline/steps/lockable-resources/#lock-lock-shared-resource*
[5] Milestone Plug-in: *https://plugins.jenkins.io/pipeline-milestone-step*
[6] Locks und Milestone Beispiel: *https://github.com/jenkinsci/workflow-aggregator-plugin/blob/691c8b0c16690f12c8d425d5b3a6aa9019d6bcfb/demo/repo/Jenkinsfile*
[7] J.Schnatterer, Automatisierte Überprüfung von Sicherheitslücken in Abhängigkeiten von Java-Projekten, Java aktuell 01/2017
[8] Pipeline Examples - Build Cause: *https://jenkins.io/doc/pipeline/examples/#get-build-cause*
[9] Jenkins Issue 41272: *https://issues.jenkins-ci.org/browse/JENKINS-41272*
[10] Groovy Scripts vs. Classes: *http://docs.groovy-lang.org/latest/html/documentation/index.html#_scripts_versus_classes*
[11] Groovy Call Operator: *http://docs.groovy-lang.org/latest/html/documentation/#_call_operator*
[12] cps-global-lib-plugin Pull Request 37: *https://github.com/jenkinsci/workflow-cps-global-lib-plugin/pull/37*

Johannes Schnatterer
johannes.schnatterer@triology.de

Johannes Schnatterer ist Solution Architect bei der TRIOLOGY GmbH in Braunschweig. Technologisch ist er dort in den Bereichen „Java EE" und „Web" tätig und versucht, mit besonderem Fokus auf Qualität, Open-Source-Enthusiasmus, einem Hauch von Pedantismus und der Pfadfinderregel die IT-Welt jeden Tag ein bisschen besser zu machen. Derzeit arbeitet er am Cloudogu Ecosystem.

„Es wird zurzeit alles getan, um Java in die richtige Richtung zu lenken …"

Simon Ritter ist Deputy CTO bei Azul Systems und großer Fan des neuen Release-Plans für Java Standard Edition und Java Enterprise Edition, da man damit beiden Anwendergruppen gerecht werde. „Die Übergabe von Java EE an die Eclipse Foundation sorgt für mehr Stabilität und eine bessere Einbeziehung der Community", so Ritter.

Im DOAG.tv-Interview *(siehe „https://www.doag.org/de/home/news/es-wird-zurzeit-alles-getan-um-java-in-die-richtige-richtung-zu-lenken/detail")* mit Andrea Badelt, stellv. Leiter der DOAG Java Community, spricht Ritter außerdem über das Java-Ökosystem, JDK, Support Features und mehr.