

JavaTM magazine

By and for the Java community 

JANUARY/FEBRUARY 2017

ローカル変数の型推論 48 | SCALA 35 | ブロックチェーン 24

ツール

05

多言語:XMLによらない
MAVENへの進化

10

ビルド・ツールのアーキ
テクチャの仕組み

17

独自のJVMデバッグ・
ツールを作る



著者: Jason van Zyl, Manfred Moser
Ruby、YAML、Groovyなどの言語でPOMファイルを記述できる新しいツールをMavenの開発者が作成

表紙画像: I-HUA CHEN

02
編集長より
多言語の未来: プログラミングの性質が変化した今、本誌でJavaScriptを定期的に取り上げるべきか

24
暗号通貨
ブロックチェーン: Javaで暗号通貨を使う
著者: Conor Svensson
web3jによりJavaアプリケーションにEthereumブロックチェーンを統合する

35
JVM言語
Scala: 完全な関数型と純粋なオブジェクト指向の融合
著者: Adriaan Moors
JVMのための成熟した実用的な型保証言語

10
最新のビルド・ツールの設計と構造
著者: Cédric Beust
最新のJVMビルド・ツールの仕組み—アーキテクチャと実装

17
独自のデバッグ・ツールを作る
著者: Andrei Pangin
JDKのサービサビリティ・テクノロジーによりJVMを監視して複雑なデバッグ問題を解決する

40
クラウド
クラウド上のコンテナでのJavaの実行
著者: Harshad Oak
DockerコンテナのJavaアプリをOracle Application Container Cloud Serviceにデプロイする

51
Fix This
著者: Simon Roberts
最新のコードのクイズ

09
ユーザー・グループ
Chicago JUG

48
Java注目の提案
JEP 286: ローカル変数の型推論



//from the editor /



多言語の未来

プログラミングの性質が変化した今、本誌でJavaに加えJavaScriptを取り上げるべきか

現在提供されているほぼすべての商用プログラムが2つ以上の言語に依存しているのは周知の事実です。Javaの守備範囲はUIからバックエンド処理にまで及び（まさにエンド・ツー・エンド）、そのJavaの役割が、他の言語で書かれた機能によって少なからず補完されています。Javaを補完する機能は、スクリプトやJVM言語で作成されることもあります。JavaScriptで書かれたUIでJavaコアを取り囲むのがより一般的なやり方です。この多言語化という現象は多言語プログラミングと呼ばれ、2つ以上の言語を使用することを通常は意味します。ここでJava以外にもう1つだけ言語を選ぶとすれば、圧倒的にJavaScriptでしょう。

いろいろな意味で、JavaScriptはこれまで、繰り返し現れるJavaの相棒として歩んできました。NetscapeがJavaの人気にあやかって付けた「JavaScript」という名前（この名前ののおかげで、エンジニアでない人々に終わりなき混乱を招いています）に始まり、JavaとJavaScriptと一緒にWebアプリケーションで使用されるようになり、それがモバイル環境にも広がりました。そしてご承知のとおり、Nashornが搭載

されたことで、Javaのスクリプト言語としてJavaScriptが使用されるようになりました。また、JavaをJavaScriptにトランスコンパイルするGoogle Web Toolkitのようなツールにより、2つの言語の絆はさらに深まりました。

JavaScriptはバックエンド・コンピューティングにも浸透しています。この傾向は、Node（旧称node.js）を見れば一目瞭然です。Nodeは、「軽量化と効率化を図るイベント駆動型のノンブロッキングI/Oモデル」を提供することにより、ビジネス・ロジックのためのサーバー類似環境を作るJavaScriptフレームワークです。Nodeの出現は、10年ほど前に小規模プロジェクト向けの便利なソリューションとして登場したRuby on Railsに似ています。Nodeは、幅広く使用されている言語に依存することから、小規模プロジェクト分野で存在感を増すことになりそうです。

ここで忘れてはならないのが、JavaコミュニティがNodeの影響を受けて素晴らしいフレームワークVert.xを生み出したことです。Vert.xは、Nodeフレームワークの全機能を含む幅広い機能を備え、急速に支持を集めてい

写真：BOB ADLER/VERBATIM

ORACLE®



Level Up at Oracle Code

Step up to modern cloud development. At the Oracle Code roadshow, expert developers lead labs and sessions on PaaS, Java, mobile, and more.

Get on the list for event updates:

go.oracle.com/oraclecoderoadshow

developer.oracle.com

#developersrule



//from the editor /

ます。Vert.xの大きな魅力の1つは、その名前の「.x」に象徴されます。この部分は、Nodeの元の名前に含まれる「.js」とは対照的になっています。つまり、NodeがJavaScriptフレームワークであるのに対し、Vert.xの「x」は多くの言語がサポートされることを表しています。私たちはまた多言語プログラミングの誕生を目にしているわけです。

オラクルで働く開発者は数万名に上り、オラクルとつながりのある開発者に至っては百万人を下りません。多言語化の傾向は現実であり、持続するものであり、重要である、というのがオラクルの結論です。この信念の根底には、さまざまなスタックおよびツールチェーンの設定やテストが容易なクラウドによって多言語の普及が加速するという確信があります。これを踏まえて『Java Magazine』の使命を広げ、まずはJavaScriptに関する記事を盛り込むような形で多言語プログラミングを取り上げるべきか、オラクルから意見を求められました。

タイトルにJavaを掲げる出版物として、JavaScriptの記事を掲載するか否かをわざわざ検討するのも奇妙に思われるかも知れませんが、タイトルのことはしばし忘れましょう。この2年間、本誌では定期的に他のJVM言語について取り上げてきました。そういった意味では、誌面にはすでに多言語に関する記事が定期的に姿を見せていたわけです。ただ、今回受けた相談には、JVM言語だけでなく、多言語の実情にもっと深く関わるという意味合いがあります。さて、本誌でJavaScriptに関する記事を定期的に掲載すべきなのでしょう

うか。

皆さんのための『Java Magazine』ですから、皆さんの意見をぜひ参考にさせていただきます。この誌面でJavaScriptのチュートリアルを、ゆくゆくは定期的な記事としてご覧になりたいですか。私が特に皆さんに伺いたいのは、JavaScriptについてのどの程度取り上げれば皆さんのプログラミングに役立つかということです。「まったく不要」から「たっぷり」まで、遠慮なく意見をお聞かせください。また、今号に掲載されている記事の構成に関する感想や提案もお待ちしています。

Editor in Chief, Andrew Binstock

javamag_us@oracle.com
[@platypusguy](https://twitter.com/platypusguy)

The poster is for the 'Oracle Code' event. It features a red background with a pattern of small white diamonds. At the top right, there is a large orange circle containing the text 'ORACLE CODE' in white. Below this, the text 'Register Now' is written in white. The main title 'Oracle Code' is in large white letters. Underneath, it says 'New One-Day, Free Event | 20 Cities Globally'. A yellow box contains the text 'Explore the Latest Developer Trends:'. Below this, there is a list of topics: 'DevOps, Containers, Microservices & APIs', 'MySQL, NoSQL, Oracle & Open Source Databases', 'Development Tools & Low Code Platforms', 'Open Source Technologies', and 'Machine Learning, Chatbots & AI'. At the bottom right, it says 'Find an event near you: developer.oracle.com/code'. The bottom left has the text 'Live for the Code' in white, and the bottom right has the 'ORACLE' logo in white.

ORACLE CODE

Register Now

Oracle Code

New One-Day, Free Event | 20 Cities Globally

Explore the Latest Developer Trends:

- DevOps, Containers, Microservices & APIs
- MySQL, NoSQL, Oracle & Open Source Databases
- Development Tools & Low Code Platforms
- Open Source Technologies
- Machine Learning, Chatbots & AI

Find an event near you:
developer.oracle.com/code

Live for the Code

ORACLE



JavaとJVMツール

POLYGLOT FOR MAVEN 05 | ビルド・ツールの仕組み 10 | JVMのデバッグ 17 | ローカル変数の型推論 48

私

たち開発者にとって、ツールほど心躍るものはそうそうありません。何をやるにせよ、ほとんどのことはツールを使用することで間接的に成り立っているわけです。実は、優秀な開発者は共通の特徴として、使用するツールと、そのツールを最大限に活かす方法についての深い知識を備えています。このことを念頭に、最初の記

事 (5ページ) では、Mavenの最新の進化形、Polyglot for Mavenについて考察します。Polyglot for Mavenにより、XMLでのビルド・ファイルの作成から解放され、真のスクリプト言語を使用したビルド・ファイル作成への扉が開かれます。新たなビルド・ツールKobaltに関する記事 (10ページ) では、このアイデアをさらに一歩先に進めます。Kobaltは、Javaに類似したJVM言語であるKotlinで作成されています。この記事では、ビルド・ツールのアーキテクチャに注目し、その仕組みを詳しく説明します。ただし、著者の設計では、ビルドを定義する言語としてKotlinを使用しています。こうすることで、著者が指摘しているように、ビルド・ファイルで真の表現性が得られるだけでなく、ビルドを記述しながらIDEでエラーを捕捉することができます (人気上昇中のビルド・ツールGradleも、ビルド・ファイルをGroovy構文からKotlinに移行している点で同じ方向に向かっています)。



コードを修正するために必要な一部の情報がデバッガで得られないという、一般的でなくとも、まれとは言えない状況についても取り上げます (17ページ)。この記事では、実行中のプロセスに関する情報を抽出し、変数やクラスを表示し、カウンタに問い合わせ、必要な詳細情報を取得することができる、JVM用のデバッグAPIについて検証します。この情報には驚くほど簡単にアクセスでき、一般的でない問題や複雑な問題を解決する単発のデバッグ・スクリプトを作成できます。

さらに、この情報を補完するために、オラクルのBrian Goetzが、ローカル変数の型推論に役立つ、言語の字句変更について論じます (48ページ)。Scalaについて掘り下げる記事も掲載しています (35ページ)。最後に、暗号通貨を支えるテクノロジー、ブロックチェーンのプログラミングを始める方法について取り上げます (24ページ)。これまで本誌に掲載された中でも最高に素晴らしい記事の1つです。Java言語に関する高度なクイズ (51ページ) もあります。今号が役立つ情報で満載だと皆さんに感じていただければ幸いです。



JASON VAN ZYL
MANFRED MOSER

Polyglot for Maven: Mavenの多言語化とスクリプトの導入

Ruby、YAML、Groovyなどの言語でPOMファイルを記述できる新しいツールをMavenの開発者が作成

Apache Mavenは、JVMでもっとも広く使われているビルド・ツールです。Mavenは、ここ10年ほどで依存性管理やリポジトリの使用などの新しい考え方を開発エコシステムにもたらしてきました。Maven Centralと呼ばれるセントラル・リポジトリは、Mavenのデフォルト・リポジトリです。ここからプラグインや依存性をダウンロードします。セントラル・リポジトリは、JVMエコシステムにおけるオープンソース・アーティファクトのいわば交換拠点であり、MavenをはじめとするGradle、sbt、Ant/Ivyなどのビルド・ツールによって使用されています。そのため、Mavenはオープンソース・コミュニティだけでなく、エンタープライズ・ユーザーにとって貴重なツールであり続けています。

XMLの功罪

Mavenユーザーが何年も口にしてはいるよくある不満の1つは、プロジェクト・オブジェクト・モデル（略して、POMまたはpom.xml）のメイン・ビルド・ファイルのマークアップ言語としてXMLが使われていることです。

Mavenが10年以上前に作られたとき、XMLが採用されたのは自然な結果でした。XMLは、安定した構造、よく知られた構文、利用できる強力なツールなどのおかげで、とても有用であることが実証されています。しかし、時代が変わり、XMLは最新かつ最良の選択肢ではなくなっています。現在の開発者が求めているのは、簡潔な構文とインライン・スクリプトなどの機能です。オープンソース・プロジェクトである[Polyglot for Maven](#) (Mavenの多言語化) は、Ruby、YAML、Groovyなどの言語を使ってPOMを定義できるようにするものです。本記事では、その仕組みを説明します。

表面的に見れば、Mavenは常にpom.xmlファイルを使用していま

す。最小限のpom.xmlファイルは、プロジェクトの識別子を定義しただけのもので、groupId、artifactId、versionという3つの値だけで構成されています。これは、頭文字をとって、GAV座標とも呼ばれています(リスト1参照)。

■ リスト1:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example.project</groupId>
  <artifactId>hello-world-demo</artifactId>
  <version>1.0.0</version>
</project>
```

この単純な設定でJARアーカイブをビルドでき、ビルドの一環として、ソースの作成、ソース・コードのコンパイルとテスト、アーカイブのパッケージ化を行うこともできます。この単純な例から、Mavenの力強さを理解できます。この強さは、いくつかのXMLに備わっている表記規則に由来するものですが、冗長にならざるをえないXMLファイルに重要な情報が埋められる場合があることもわかるでしょう。

実際、依存性やプラグインの構成情報、現在のビルドやプロジェクトについてのその他の情報を含めた場合、一般的にpom.xmlファイルはさらに長くなります。

プロジェクト・ディレクトリからpom.xmlファイルを取得して解析し、プラグインを順次起動してビルドを始めるという作業は、すべてMavenに組み込まれています。内部的に見ると、pom.xmlファイル


```

groupid: io.takari.polyglot
artifactId: yaml-project
version: 0.0.1-SNAPSHOT
name: 'YAML Maven Love'
properties: {sisuInjectVersion: 0.0.0.M2a}

```

YAMLの構文を使うと、1つの単純な親ノードがある任意のリストを宣言できます。リスト4の例は、この構文と、依存性スコープを含むGAV座標の短縮構文を利用した依存性リストを示しています。

■ リスト4:

```

dependencies:
- {artifactId: junit, groupId: junit, scope: test,
  version: 4.12}
- org.codehaus.groovy:groovy-all:2.4.3

```

YAMLフォーマットのその他の例は、[メイン・プロジェクト](#)のtestリソースで公開されています。YAMLは、成熟して安定したフォーマットです。[SnakeYAMLproject](#)は、日々の開発やリリースの作業を対象として積極的にYAMLフォーマットをサポートし、活用しています。

YAMLフォーマットと同様の簡潔さを備える、XMLベースでないもう1つのフォーマットがAtomです。サンプル・プロジェクトを開いて構文を確認してみてください。

マークアップ言語によるPolyglot for Mavenの実装には、オリジナルと同様にXMLを使ったpolyglot-xmlもあります。オリジナルのコアMavenのpom.xmlファイルは要素だけを使っていますが、新しいバージョンでは属性を使っているため、XMLベースのツールをそのまま使い続けられる一方で、大幅にXMLファイルを短くすることができます。

YAML、Atom、XMLいずれのフォーマットでも、それぞれ独自の簡潔なPOM用の構文が提供されています。開発者は、可読性が高く自然なフォーマットだと感じることでしょう。

言語系のフォーマット

Polyglot for Mavenは、マークアップ言語向けのフォーマットだけでなく、完全なプログラミング言語（Groovy、Ruby、Clojure、Scala）をベースとした方言によるマークアップもサポートしています。すなわち、それぞれの言

語でおなじみの構文を使ってpom.groovy、pom.rb、pom.clj、pom.scalaファイルを定義できるようになります。さらに、POMに実行可能なコード・スニペットを埋め込むこともできるようになります。

こういったフォーマットは簡潔であると同時に、それぞれのプログラミング言語を使っている開発者にはおなじみです。各フォーマットでは、polyglot-maven-pluginとそのexecuteゴールを経由したコードを実行できます。リスト5のように、最小限のpom.rbファイルは、単純なexecute

ブロックがあれば有効です。

■ リスト5:

```

pom.ruby snippet with Ruby execution
execute :id => 'hello', :phase => 'validate' do
  puts 'hello world'
end

```

Groovyで書かれたリスト6のpom.groovyスニペットでは、同様の機能がbuildセクションでラップされています。

■ リスト6:

```

build {
  $execute(id: 'hello', phase: 'validate') {
    println "Hello! I am Groovy inside Maven."
  }
}

```

実際の使用例

言語ベースのフォーマットのメリットは、その言語をサポートしているIDEがすでに存在するという点にあります。そして、[IntelliJ IDEA](#)と[Eclipse IDE](#)の両方でPolyglot for Mavenのサポートが始まっています。

今のところ、Polyglot for Mavenを一番活用しているのはJRubyチームです。JRubyチームのプロジェクトの[ソース・コード](#)では、大きく複雑なpom.rbファイルの良い例を見ることができます。pom.rbファイルの冒頭部分をリスト7に示します。



■ リスト7:

```
version = ENV['JRUBY_VERSION']
||
File.read( File.join( basedir,
'Version' ) ).strip
```

```
project 'JRuby', 'https://github.
com/jruby/jruby' do
```

```
  model_version '4.0.0'
  inception_year '2001'
  id 'org.jruby:jruby-parent', ver-
sion
  inherit 'org.sonatype.oss:oss-
parent:7'
  packaging 'pom'
```

```
organization 'JRuby', 'http://jruby.org'
```

```
[ 'headius', 'enebo', 'wmeissner', 'BanzaiMan',
'mkristian' ].each
do |name|
  developer name do
    name name
    roles 'developer'
  end
end
```

リスト7には、興味深い利用パターンがいくつかあります。たとえば、バージョン番号が含まれた外部プロパティ・ファイルを読み取り、その値をPOMに利用している部分です。inherit宣言では、親のPOMが定義されています。複数の開発者がコレクションとして宣言されており、このコレクションがeachで反復処理されて、POMのdeveloperコレクションが定義されています。また、リポジトリにpom.xmlファイルが存在していることにも気づくでしょう。これにより、ビルドの定義以外に、pom.xmlファイルが果たしている重要な役割がわかります。

リリースの準備ができた際には、セントラル・リポジトリまたは内部リポジトリのマネージャに対して、正しいpom.xmlが発行されるようにする必要があります。

POMは、プロジェクトの多くの側面（名前、URL、ソース・コード・マネージャ情報など）を管理しているメタデータ・コンテナです。ここで重要なのは、POMには親のPOMや依存性も含まれている点です。

依存性の定義は、アーティファクトを使用する際に欠かせません。この定義が存在するため、セントラル・リポジトリやその他のMavenリポジトリを使用するMavenなどのビルド・ツールは、直接宣言されている依存性やプラグインに加えてダウンロードする依存性を決定できます。これらのツールは、すべてpom.xmlフォーマットを解釈しています。

そのため、リリースの準備ができた際には、セントラル・リポジトリまたは内部リポジトリのマネージャに対して、アーティファクトとともに正しいpom.xmlが発行されるようにする必要があります。そうすることによって、アーティファクトにアクセスするツールがメタデータを読み出して、必要なものをすべてダウンロードできるようになります。

このダウンロードは、数多くのさまざまなツールで行うことができます。具体的にはGradle、sbt、Leiningen、Ant/Ivyといった他のビルド・ツールなどが挙げられ、これらはすべてMavenリポジトリ・フォーマットを処理して依存性を取得します。この機能はオリジナルのXMLフォーマットのPOMに基づくもので、オープンソース・コミュニティの相互運用性という要素にとって重要なものです。

そのため、Rubyなどの言語向けのPolyglot for Mavenでは、RubyでのPOMに加えて、XMLフォーマットのPOMも生成するようになっています。コンシューマ用のXML版POMを作成する機能は、今後いくつかの言語でサポートする必要があります。このような教訓は、コアMavenプロジェクトにも還元されます。今後のバージョンのMavenでは、ネイティブのXML版POMフォーマットを進化させ、すでに公開されているアーティファクトとの下位互換性を維持する方法として、コンシューマPOMとビルドPOMを分離することになるかもしれません。

JRubyコミュニティは、次のステップとしてMavenのネイティブ・ラッパーの作成に取りかかっています。このラッパーを使うと、MavenのRuby拡張機能が自動的にインクルードされたrmvnコマンドの使用や、Rubyスクリプト内からのMavenの実行が可能になります（リスト8参照）。

■ リスト8:

```
require 'ruby-maven'
RubyMaven.exec( '--version' )
```



まとめ

数年前にPolyglot for Mavenが生まれてから、この拡張機能への関心は高まり、採用数も増加を続けており、オープンソース・コミュニティや企業へと徐々に広まっています。XMLのみに依存しないPOMファイルをサポートすることで、簡単かつ明瞭にビルドを表現し、記述する方法が実現しています。ぜひPolyglot for Mavenをお試しいただき、フィードバックをお寄せください。 </article>

Jason van Zyl: Apache Mavenプロジェクトをはじめとする多くの実績あるオープンソース・プロジェクトを創設。オープンソース・コミュニティ随一のアーティファクト交換拠点であるセントラル・リポジトリを開始した実績を持つ。Fortune 500企業の開発ツール/開発者サポート・チームを率いて、その経験をオープンソース・コミュニティ全体に還元している。

Manfred Moser: Apache Mavenコミッター、プラグイン開発者。多数のMavenユーザーのトレーニングやサポートに関わっており、トレーナー、著者、デベロッパー・アドボケートの立場からvan Zyl氏をサポートしている。

CHICAGO JUG



Chicago Java Users
Group (CJUG) は、1995年に Philip McGlauchlin氏によって設立されました。2016年現在、2,000名近くのメンバー数に成長している米国有数の JUG です。現在は、Java 関連の著者、講演者、オープンソース・プロジェクトのリーダー、JCP メンバー、Apache Software Foundation、Java Champion な

どが幹事を務めています。

CJUGは、シカゴのJavaエコシステムに貢献する人々の裾野を広げることに重点的に取り組んでおり、その実現のため、ここ2年間にわたって Adopt-a-JSRプログラムを継続しています。特に力を入れているのは、JSR 366 (Java EE 8) とJava 9であり、Josh Juneau氏とBob Paulin氏がその陣頭指揮にあたっています。このプログラムの今年の目玉となるのは、四半期ごとに行われる参加型会議です。

CJUGは、[Chicago Tech on Slack](#)というオンラインSlackコミュニティを通じた、メンバーのサポート活動も行っています。また、さまざまなシカゴの企業を回って定期的に会合を行うという、スポンサーへの独特なアプローチ方法をとっています。これによって、開発者はCJUGの隔月のミーティングに参加するだけで、シカゴの幅広い企業の一面に触れることができます。

著名な講演者も参加しているCJUGですが、グループが重視しているのは、四半期に一度行われるライトニング・トークを通じて地元のメンバーの講演スキルを磨くことです。現在、このコミュニティを率いているのは、Java ChampionのFreddy Guime氏です。Guime氏は、最新ニュースやJava分野のリーダーへのインタビューを提供する[Java OffHeapポッドキャスト](#)の開設者でもあります。

CJUGは、シカゴをJava開発者にとって絶好の場所にするという使命に貢献していただける方の参加を常にお待ちしています。

最新のビルド・ツールの 設計と構造

ここ20年間で、JVMはさまざまなビルド・ツールを目の当たりにしてきました。JavaだけでもAnt、Maven、Gradle、Gantなどが存在し、JVM言語では、Scalaでsbtが、ClojureでLeiningenが使われています。これらのツールは、開発者やバックエンド・システムによって1日に何度も実行されていますが、実装方法、実現されている機能、提供すべき機能やその理由などのほとんどは、公式ドキュメントに記載されていません。

本記事に記載する内容は、筆者が数十年にわたってソフトウェアを構築してきた経験や、現在開発中のKobaltという実験的なビルド・ツールから得たものです。これは本誌で以前に取り上げたJVM言語Kotlin用のビルド・ツールです。

Kobaltを作ることになったのは、Gradleは融通性やビルド・ファイルの構文の表現力という面で明らかに前進したツールであるものの、いくつかの欠点もあり、その一部はGroovyに依存しているために生じていると感じたからです。そのため、Gradleを参考にしつつ、JetBrainsが開発したKotlin言語をベースにしたビルド・ツールを作ることになりました。

Kobaltは、Kotlinで書かれているだけでなく、そのビルド・ファイルも有効なKotlinプログラムであり、ベテランのGradleユーザーにはおなじみのシンプルなDSLになっています（なお、Kotlinは最新版のGradleのビルド・ファイルの構文にも採用されており、Groovyのビルド・ファイ

本記事では、ビルド・ファイルの一般的なコンセプトや、それに対するKobaltのアプローチについて説明します。Kotlinの構文はJavaとよく似ているため、Kotlinに詳しくなくても問題ありません。

- ユーザーは、アプリケーションの作成に必要なステップを記述した1つ (またはそれ以上) のビルド・ファイルを作成する必要があります。ビルド・ファイルを記述する構文は、単純なプロパティ・ファイルから有効なプログラミング言語のソース・ファイル、あるいはその中間的なものまで、さまざまです。
- ビルド・ツールは、ビルド・ファイルを解析し、特定の順番で実行する必要があります、タスクのグラフに変換します。
- その後、ツールはこれらのタスクを適切な順番で実行します。タスクには、ビルド・ツールの内部でコード化されるものと、外部プロセスを呼び出さなければならないものがあります。ビルド・ツールは、ほとんどあらゆることをタスクに実行させることができます。

JVM最古のビルド・ツールの1つがAntです。Antは、それまで長く使われてきた標準ツールmakeから派生したものです。Antでは、ビルド・ファイルの言語にXMLが採用されましたが、当時としては画期的なことでした。また、Antでタスクという概念が生まれました。タスクは、XMLファイルで定義するか、またはツールで検索する外部タスクとして、Javaで実装して呼び出すことができます。Antはレガシー・ソフト

ウェアで依然として使用されているものの、現在では基本的に時代遅れであると考えられており、ビルド・ツールの「アセンブリ言語」のようなものだと考えられています。つまり、とても柔軟なツールであり、妥当なプラグイン・アーキテクチャも提供されていますが、もっとも単純なビルド・プロジェクトにも大量の定型挿入文が要求されるということです。

それでは、ビルド・ツールに期待されることについて、もう少し具体的に見てみることにします。

構文

まずは、ビルド・ツールのもっとも目に触れやすい側面、ビルド・ファイルの構文から始めます。

当然ですが、筆者はビルド・ツールに簡潔で直感的な構文を求めます。しかし、「簡潔な構文」とは何なのかについて一般的なコンセンサスを得ることは困難です。そのため、この定義は考えないことにします。筆者は、構文自体よりも、ビルド・ファイルの記述や編集を容易に行えることの方が重要な場合があるとも思っています。たとえば、Mavenのpom.xmlファイルは構文としては冗長ですが、ファイルのスキーマを認識しているエディタを使えば、とても簡単に編集できます。

Gradleが人気である大きな理由は、ビルド・ファイルの構文がXMLよりはるかに簡潔なGroovyであることに由来しています。筆者の個人的な経験では、Gradleのビルド・ファイルは読みやすいものの書きにくいと感じています。この点には後ほど触れることにしますが、大局的に見れば、Gradleの構文は有用であるという見方が一般的です。

その点よりも議論の余地があるのは、ビルド・ファイルは純粋な宣言的構文（たとえば、XMLやJSON）を使用したものであるべきか、それとも何らかのプログラミング言語による有効なプログラムであるべきかという問題でしょう。

筆者の経験によれば、宣言的構文を使いたい（大半のビルド・ファイルはこれで十分であるためです）という一方で、必要になった場合は完全なプログラミング言語の力も使いたいという結論になります。また、ビルド・ファイルでは同じ構文を使う必要があるという要件が存在する場合に、他の言語に逃げるのを強いられたくはありません。この判断は、ビルド・ファイルでオブジェクト指向プログラミングが持つすべての力を活用したいとよく実感していたことによるものです。そうすれば、タスクのベース・クラスを作成しつつ、あちこちでいくつかのバリ

エーションを使って特殊化することができます。たとえば、いくつかのライブラリを作成するプロジェクトで、すべてのライブラリで同じフラグを付けてコンパイルするものの、それぞれの名前やバージョンを別々にする必要があるという場合です。この種の問題は、継承とメソッドのオーバーライドが可能でオブジェクト指向言語では簡単に解決できます。そのため、ビルド・ファイルにもこの種の柔軟性があることが望まれます。

さらに一般的なレベルでは、複雑なプロジェクトは多くの場合、さまざまなレベルの共通設定や動作を共有するモジュールで作成されています。複数の場所や異なるファイルでそういった設定を繰り返さなくてもよいビルド・システムになっていればいるほど、さらに簡単にビルドのメンテナンスや拡張を行えるようになります。

すべてのビルド・ツールは、さまざまなレベルでのモジュール、依存性、共有パラメータなどの設定を支援するものです。しかし、筆者のビルド・ツールを含め、この「継承と特殊化」という側面を十分直感的に実現しているものがあるとは言えません。おそらく、その解決策となるのは、ビルド・ファイルのプログラミング言語内でモジュールを実際のクラスで表し、おなじみの「継承とオーバーライド」パターンを使ってこの再利用を実現することでしょう。

それができるまでの間は、有効なプログラムであり、必要になったときにはすべてのプログラミング言語の機能（if、else、クラス、継承、合成など）を提供してくれるDSLを使用するアプローチによって、望む方向への明確な一歩を踏み出せるのではないかと思います。

依存性

ビルド・ツールの一番の仕事は、一連のタスクをある順番で実行し、いずれかのタスクが失敗した場合にさまざまなアクションを行うことです。そのため、（JVMやそれ以外で）筆者が見たことがあるあらゆるビ

プロジェクト・ディレクティブ（実際はKotlinの関数）には、パラメータとして複数の依存するプロジェクトを指定できます。Kobaltは、この情報に基づいて依存性ツリーをビルドします。



ルド・ツールで、タスクの定義と、タスク間の依存性の定義が可能であることには何の驚きもありません。しかし、多くのツールでは、プロジェクトの依存性に十分には対応していません。プロジェクトAとBをビルドした後にのみプロジェクトCをビルドできることは、どうすれば指定できるでしょうか。

Gradleでは、複数のbuild.gradleファイルを巧みに使って複数のsettings.gradleファイルを参照する必要があります。この設計では、役割が異なる複数のファイル (build.gradle と settings.gradle) にまたがって依存モジュールを定義する必要があります。しかしこれによって、依存性の把握が難しくなる可能性があります。

筆者がKobaltに着手した頃、それよりも直感的なアプローチを採用して、1つのビルド・ファイルで複数のプロジェクトを定義できるようにしたいと考えました。こうすれば、依存性ツリーははるかにわかりやすくなります。たとえば、次のようになります。

```
val lib1 = project {
    name = "lib1"
    version = "0.1"
}
```

```
val lib2 = project {
    name = "lib2"
    version = "0.1"
}
```

```
val mainProject = project(lib1, lib2) {
    name = "mainProject"
    version = "0.1"
}
```

高機能なビルド・ツールであっても、開発者が日々遭遇するすべてのシナリオに対処することはできないため、拡張性は必要です。

プロジェクト・ディレクティブ (実際はKotlin関数) には、パラメータとして複数の依存するプロジェクトを指定できます。Kobaltは、この情報に基づいて依存性ツリーをビルドします。すべてのプロジェクトはそのトポロジに基づいてソートされます。つまり、ビルドの順序は「lib1、lib2、mainProject」または「lib2、lib1、mainProject」のいずれかになります。この両方とも有効です。

さらに、先ほどの例は有効で完全なビルド・ファイルです (繰り返しの部分は短縮して書くこともできますが、ここではシンプルな状態にしておきます)。

プロジェクトの依存性を1箇所で集中して管理できるため、たとえ複雑なプロジェクトであっても、はるかに簡単にビルドを理解し、変更することができるようになります。なお、各サブプロジェクトのディレクトリにビルド・ファイルを配置して複数のビルド・ファイルを使うオプションも残されています。

単純なビルドは簡単に、複雑なビルドもできるように

前のセクションに記載した機能の直接的な結果として、できる限り最低限必要なものだけを含むビルド・ファイルが作成できるようになります。

設定より規約: ほとんどのプロジェクトには通常、1つのモジュールのみが含まれており、本質的にとても単純なものです。したがって、ビルド・ツールではそのようなプロジェクトのビルド・ファイルを短くし、できる限り多くの実用的なデフォルト値を実装する必要があります。そのようなデフォルト値のいくつかを表1に示します。

このようなデフォルト値があれば、もっとも単純なプロジェクトのビルド・ファイルは、文字どおり4行以下で書くことができます。もちろん、インポートに基づいて特定の依存性を推論するなど、ビルド・ツール側でさらに分析や推測を行うこともできます。

複雑なビルド: 単純なプロジェクトの次の段階では、ビルドを簡単に修正できる

機能が重要になります。このような変更は静的 (単純にビルド・ファイルを変更する) にも動的 (ビルドを実行する際にスイッチや値を渡して特定の設定を変更する) にも行われる可能性があります。通常、後者の操作はプロファイルを使って実行します。プロファイルとは、ビルド・ファイ



ルを変更せずに、ビルド時にさまざまなアクションをトリガーする値です。

Mavenはプロファイルをネイティブでサポートしています。しかし、Gradleでは、Groovyで環境の値を抽出することによって同様の結果を実現しているため、柔軟性に欠けます。Kobaltのプロファイルは、この2つのアプローチを条件で結合したものです。次のように、プロファイルは通常のKotlinの値として定義します。

```
val experimental = false
val premium = false
```

これらの値は、通常の条件文としてビルド・ファイルの任意の場所で使用できます。以下に例を示します。

```
val p = project {
    name = if (experimental) "project-exp"
           else "project"
    version = "1.3"
    ...
}
```

プロファイルはコマンドラインでアクティブ化できます。

名前	現在のディレクトリの名前
バージョン	"0.1"
言語	自動検出
ソース・ディレクトリ	src/main/java, src, src/main/{language}
メイン・リソース	src/main/resources
テスト・ディレクトリ	src/test/java, test, src/test/{language}
テスト・リソース	src/test/resources
MAVENリポジトリ	MAVEN CENTRAL, JCENTER
バイナリ出力ディレクトリ	{SOMEROOT}/classes
アーティファクト出力ディレクトリ	{SOMEROOT}/libs

表1: Java対応ビルド・ツール向けの実用的なデフォルト値

```
./kobaltw -profiles \
    experimental,premium assemble
```

これは、プログラミング言語でビルド・ファイルを記述することによって実際にメリットがもたらされる部分です。そのプログラミング言語で認められている限り、プロファイルでトリガーされる操作をどこにでも挿入できるからです。

```
dependencies {
    if (experimental)
        "com.squareup.okhttp:okhttp:2.5.0"
    else
        "com.squareup.okhttp:okhttp:2.4.0"
}
```

if (experimental)の部分では、コマンドラインで指定されたプロファイルが参照されます。

パフォーマンス

ビルド・ツールはできる限り高速であることが望めます。。そのためには、ツールのオーバーヘッドを最小限にとどめ、ビルド時間の大半が、呼び出す外部ツールによって消費されるようにする必要があります。何よりも明白な要件は、スピードに必要な2つの重要な機能である、増分タスクと並列ビルドをビルド・ツールがサポートしていることです。

増分タスク:ビルド・ツールにおいては、実行が必要かどうかを自力で検出できるタスクを増分タスクと呼びます。通常、この判定は、現在の実行の出力が前回の実行と同じになるかどうかを計算し、同じになる場合はただちに処理を戻すことによって行われます。程度の差はありますが、増分タスクはほとんどのビルド・ツールでサポートされています。この機能のレベルを確認するためには、ビルド・ツールで同じコマンドを2回連続で実行し、2回目の実行時にどのくらいの作業が行われるかを確認します。

並列ビルド:増分タスクとは対照的に、並列ビルドは一部のビルド・ツールでしかサポートされていません。その理由は、ソフトウェアよりも



この図から、coreモジュールを最初にビルドしなければならないことがわかります。その後は、いくつかのモジュールを並列にビルドできます。たとえば、locationsとfeaturesなどです（この2つは両方ともcoreに依存していますが、お互いに依存してはいません）。モジュールのビルドが進むにつれ、図1の依存性グラフに基づいて、他のモジュールのビルドがスケジューリングされてゆきます。

並列ビルドを行うと、大幅にビルド時間を削減できます。たとえば、並列ビルドで68秒かかった最新版のビルドには、逐次ビルドで260秒かかりました。実に4倍となっています。

高機能なビルド・ツールであっても、開発者が日々遭遇するすべてのシナリオに対処することはできないため、拡張性は必要です。従来より、拡張性はプラグイン・アーキテクチャを公開することによって実現されてきました。開発者は、プラグインでツールを拡張することによって、設計された当初の用途とは異なるタスクを実行できます。ビルド・ツールの有用性は、プラグイン・エコシステムの健全性や規模と直接的な相関性があります。

興味深いことに、OSGiは明確に定義されている優れたプラグインAPIアーキテクチャですが、OSGiを使ったビルド・ツールは見たことがあ

```
graph TD; samplesPost[samplesPost] --> samples[samples]; samplesPost --> locations[locations]; samplesAsync[samplesAsync] --> samples; samplesTestable[samplesTestable] --> samples; samplesJson[samplesJson] --> samples; samplesHello[samplesHello] --> samples; samplesHello --> jetty[jetty]; samplesEmbedded[samplesEmbedded] --> samples; samplesEmbedded --> netty[netty]; samplesEmbedded --> tomcat[tomcat]; websockets[websockets] --> samples; websockets --> jetty; websockets --> netty; websockets --> tomcat; websockets --> core[core]; websockets --> features[features]; jetty --> servlet[servlet]; netty --> servlet; tomcat --> servlet; servlet --> hosts[hosts]; hosts --> core; freemarker[freemarker] --> features; serverSessions[serverSessions] --> features; features --> core; locations --> core; samples --> core;
```

14

りません。残念ながら、ビルド・ツールは独自のプラグイン・アーキテクチャを生み出す傾向にあります。

この点を詳しく説明すると、本の1章分ほどの内容になってしまうため、ここでは、プラグイン・アーキテクチャは基本的に2つの方法で実現できることだけを述べておきます。最初の方法は、プラグインが完全にツールの内部構造にアクセスできるようにするアプローチです。Gradleはこの方式を採用しています（Gradleの土台となっているGroovyがこの機能をもたらしています）。

それとは異なり、Eclipse、IntelliJ IDEAといった開発環境やKobaltは、ドキュメント化されたエンドポイントを公開しています。プラグインはエンドポイントに接続して環境内の値の監視や変更のために使用することができますが、ビルド・ツールがこの環境を完全に制御しています。静的に検証できることに加え、ドキュメント化とメンテナンスがはるかに簡単であるため、筆者はこちらの方式を好んでいます。

パッケージ管理

Mavenは、汎用的で画期的なビルド・システムであるだけでなく、それをはるかに超えて使われ続ける遺産となるであろう機能を導入しました。その機能とは、リポジトリです。たとえMavenが古くなって使われなくなったとしても、現在利用できるさまざまなMavenリポジトリのパッケージは、この先も参照され、ダウンロードされ続けるでしょう。

こういったリポジトリの人気を考えれば、最新のビルド・ツールは以下のことを実現するべきです。

- リポジトリから依存性を透過的かつ自動的にダウンロードできるようにする（Maven、Gradle、Kobaltはすべてこれを実現しています。Antでは追加のツールが必要です）。

最近では、Webサイトにアクセスしてパッケージをダウンロードし、手動でインストールする時間があるような人はいません。**ビルド・ツールも例外ではなく、ツールの自動更新が必要です。**

- できる限り簡単に自分のプロジェクトをリポジトリで利用できるようにする（MavenとGradleでこれを実現するためには、プラグインが必要となります。ビルド・ファイルにもかなりの量の定型挿入文を書かなければなりません。Kobaltはネイティブでこのようなアップロードに対応しています）。

IDEでの自動補完

開発者は、毎日何時間もIDEを使用します。そのため、当然ながら、ビルド・ファイルを編集する際にも、IDEの全機能が使えることを期待するでしょう。ビルド・ツールでは部分的にしか構文がサポートされていないことが多いため、この点で大きな成功を収めているとは言えません。

興味深いことに、MavenとPOMファイルは、IDEで常に十分にサポートされてきました。これは、POMファイルが、XMLスキーマで完全に表現できるXMLフォーマットに依存しているためです。このファイルは冗長ですが、精度の高い自動補完を利用できます。さらに、Mavenのスキーマは、非常に厳密なルールを持つ適切なXMLファイルを定義する方法のよい例にもなっています（たとえば、まったく属性が使われていないため、データを入れる方法で悩むことはありません）。

さらに新しいGradleは、動的に型付けされる言語であるGroovyに依存しているため、この点でMavenよりも普及しているとは言えません。Kobaltのビルド・ファイルはKotlinに依存しており、特別なことをしなくてもIntelliJ IDEAで自動補完を利用できます。今後予定されているKotlinベースのGradleでは、間違いなくこれと同様のレベルの自動補完が利用できるようになるでしょう。

自動更新

最近では、Webサイトにアクセスしてパッケージをダウンロードし、手動でインストールする時間があるような人はいません。ビルド・ツールも例外ではなく、ツールの自動更新（少なくとも更新を簡単にする仕組み）が必要です。システムの標準パッケージ・マネージャ（brew、dpkgなど）からツールを利用できるに越したことはありませんが、複数のオペレーティング・システム間で統一された形でツール自体を更新できるようにするべきでもあります。しかし、ビルドのうちで更新し続ける必要があるのは、ツール自体だけではありません。依存性もとても重要であるため、依存性の新しいリリースが利用できるようになった際にビ



ルド・ツールから通知を行い、常に最新の状態が維持されるようにする必要があります。

まとめ

2016年にビルドされたソフトウェアは、20年前のものよりはるかに複雑です。また、ツールを高水準に維持することも重要です。ライブラリ、IDE、デザイン・パターンは、ソフトウェアのニーズが増えるにつれて適応させ、改善する必要があるコンポーネントの一部に過ぎません。ビルド・ツールも例外ではありません。使っているありとあらゆる種類のツールと同じく、ビルド・ツールにも妥協すべきではないでしょう。 </article>

Cédric Beust (@cbeust) : 1996年よりJavaコードを書き続けており、長年にわたって言語やライブラリの発展に積極的に貢献。フランスのニース大学でコンピュータ・サイエンスの博士号を取得。JVMのアノテーションの設計を行った専門家グループの一員。

learn more

[Gradle](#)
[Kotlin](#)

O'REILLY®

Software Architecture

ENGINEERING THE FUTURE OF SOFTWARE

The brightest minds in tech
are coming to New York.



Practical training in the tools, techniques,
and leadership skills you need for the evolving
world of software architecture.

April 2-5, 2017

softwarearchitecturecon.com/ny

JAVA MAGAZINE READERS GET
20% OFF WITH CODE **JAVA**





ANDREI PANGIN

独自のデバッグ・ツールを作る

JDKのサビリティ・テクノロジーを活用し、JVMに入り込んで難解なデバッグ問題を解決する

Javaは単なるプログラミング言語ではなく、アプリケーション・ソフトウェアを開発し、実行するための包括的なプラットフォームです。Javaプラットフォームの非常によく知られたメリットに、さまざまなサービス性や保守性を提供する機能があります。この機能は、エンタープライズ・アプリケーションにとって極めて重要です。多くの組み込みツールに加えて、トラブルシューティングに役立つサード・パーティ製ソフトウェアも多数存在します。

適切なJava用IDEにはどれも、ステップ実行、ブレーク・ポイント、ウォッチなどの機能をサポートした強力なデバッガが搭載されています。パフォーマンスの問題に対処する場合は、Oracle Java Mission Control、JProfiler、YourKitなどの有名なプロファイラを使うことができます。メモリ・リークもよく発生する問題ですが、メモリの問題に対応するツールも存在します。たとえば、VisualVMやEclipse Memory Analyzerなどです。こういった汎用ツールは、開発者がよく遭遇する一般的な問題を解消するには便利ですが、一般的ではない状況に必ずしもうまく対応できるとは限りません。ほとんどの読者の皆さんは標準ツールの使い方をご存知だと思いますので、ここでは固有の事例向けにカスタマイズした新しいツールの作り方について説明します。

カスタム・ツールを構築する理由

外出中に、本番サーバーのアプリケーションが動かなくなったという状況を考えてみてください。分析用にヒープ・ダンプを取ると役立つかもしれませんが、数ギガバイトのダンプ・ファイルをダウンロードできるほどインターネット接続が良好でない場合もあるでしょう。できることは、何か小さなものをリモートで実行することだけです。そのような場合、どのツールを実行すればよいのでしょうか。特定のケースで役立つかどうかわからない既存のツールを探すよりも、自分で特別なツールを作ってしまう方が早いことがあります。

カスタム・ツールが効果的な場合がある別の例は、例外無視の問題

です。宣言されている例外をキャッチし、処理せずに破棄してしまうのは、ありがちなミスです。修正できないサード・パーティ製ライブラリの中で、この例外無視が発生している場合、事態はいつそう悪いものになります。筆者は、独自仕様のJDBCドライバでこのバグに遭遇したことがあります。データベース・サーバーで発生した予期せぬエラーを処理していないため、古い接続を適切に無効化できていませんでした。サード・パーティ製ライブラリの例外は制御できなくても、特別なツールでその例外をインターセプトすることはできます。本記事では、その方法を紹介します。

また、サービスを中断させることなく実行中のアプリケーションにパッチを当てたいと思ったことのある方であれば、ロードされたコードを変更できるツールに興味があるかもしれません。もちろん、このような動作をする商用ソリューションも存在します。しかし、わずか数行のコードで問題を自力解決する方が魅力的ではないでしょうか。

カスタム・ツールによって恩恵を受ける可能性があるタスクは枚挙に暇がありません。Java SE Development Kit (JDK) に含まれているサビリティ・コンポーネントのおかげで、このようなツールの作成はかなり簡単です。本来であれば、このようなコンポーネントはそれぞれを1本の記事にできるほどのものですが、以降では、それらのテクノロジーで何ができるかについて、簡単に見てゆくことにします。

jvmstatパフォーマンス・カウンタ

システムが正常に動作していることを確認するためによく使われるアプローチの1つが、JVMのモニタリングです。Java HotSpot VMは、jvmstatパフォーマンス・カウンタを通してさまざまな遠隔測定データを提供します。このデータには、クラスのローディング、ガベージ・コレクション、マルチスレッド、JITコンパイルなど、JVMのほぼすべての範囲をカバーする数百個のインジケータが含まれています。このツールは、名前とは異なり、実際はカウンタだけで構成されているわけではありません。また、すべてがパ



パフォーマンスに関連するわけでもありません。しかし、JVMの健全性をモニタリングするためには非常に便利です。たとえるなら、パフォーマンス・カウンタを飛行機のコックピットにある計器類やダイヤルのようなもの考えるとよいかもかもしれません。

jvmstatカウンタは無償で利用できます。というより、使うか使わないかにかかわらず、Java HotSpot VMはjvmstatカウンタをエクスポートします。JVMは、ファイル・システム上の一時ディレクトリにメモリマップド・ファイルとして遠隔測定データを生成します。多くの場合、一時ディレクトリは/tmp/hspanferdata_{user}/{pid} ({pid}はJavaのプロセスID) です。ツールが、システムで実行されているJavaプロセスを見つけることができるのは、このネーミング規則があるためです。

うれしいことに、このディレクトリ・スキャン・ロジックを複製する必要はありません。この用途に使える便利なJava APIがすでに存在するためです。jvmstat APIは標準ではありませんが、標準JDKバンドルに含まれています。そのため、クラスパスに{JAVA_HOME}/lib/tools.jarを含めるだけで使うことができます。

以下に示すのは、システムで実行されているすべてのJava HotSpot VMのプロセスIDを取得する方法です。

```
import sun.jvmstat.monitor.MonitoredHost;
...
MonitoredHost host =
    MonitoredHost.getMonitoredHost((String) null);

Set<Integer> processIds = host.activeVms();
```

このコードで、`null`はローカル・ホストを表しています。リモート・ホストで`jstatd`ユーティリティが実行されていれば、リモート仮想マシンのモニタリングも可能です。プロセスIDがわかれば、`MonitoredVm`のインス

具体的には、Javaヒープのダンプ、スタック・トレースの出力、特定のVMフラグの変更、エージェント・ライブラリのロードなどの処理を**JVMに行わせることができます。**

タンスを取得してjvmstatカウンタ（またはモニター）を読み出すことができます。`Monitor`の型は、Integer、Long、Stringのいずれかです。たとえば、`sun.rt.javaCommand`という名前のモニターには、指定されたJavaアプリケーションを開始する際に使われたメイン・クラスと引数が含まれています。指定した正規表現に一致するすべてのモニターを取得するためには、次のようにして`findByPattern`メソッドを使います。

```
MonitoredVm vm = host.getMonitoredVm(
    new VmIdentifier(processId.toString()));

vm.findByPattern(".*").forEach(monitor -> {
    System.out.println(monitor.getName() + " = " +
        monitor.getValue());
});
```

このシンプルなコードで、利用可能なすべてのモニターとその値が一覧表示されます。出力には、標準ツールではわからないような興味深い指標が含まれています。たとえば、次のようなものです。

```
// クラス・イニシャライザの実行にかかった時間
sun.cls.classInitTime = 2545394
// 競合した同期の数
sun.rt._sync_ContentdedLockAttempts = 55
// ガベージ・コレクションによってVM全体が停止した時間
sun.rt.safepointTime = 811588
```

たとえば、セーフポイント時間は、VMによってアプリケーション・スレッドが強制的に停止された時間を示しているため、低レイテンシが必須となるアプリケーションではとても重要になります。ここには250以上のカウンタがありますが、セーフポイントでの停止などの一部のカウンタをモニタリングできることだけでも、優れたモニタリング・ツールだと言えないでしょうか。また、さらに改善して、長時間にわたって収集した動的プロファイルを表示することもできます。

しかし、読取り専用のカウンタだけでは、残念ながらさほど大したことではできません。JVMとの双方向通信を行うためには、別のテクノロジーが必要になります。

動적アタッチとインスツルメンテーションAPI

動적アタッチのメカニズムを使うと、実行中のVMに接続して、事前に定義された何種類かのコマンドのいずれかを実行することができます。具体的には、Javaヒープのダンプ、スタック・トレースの出力、特定のVMフラグの変更、エージェント・ライブラリのロードなどの処理をJVMに行わせることができます。VM自体がコマンドを実行することになるため、応答を得るにはVMが正常に動作している必要があります。

先ほどクラスパスに追加したtools.jarファイルには、動적アタッチ用Java APIが含まれています。なお、このAPIはOpenJDKおよびOracleのJDKのみに適用できるベンダー固有のものである点に注意してください。

実行中のJavaプロセスへのアタッチは簡単です。次のコードのように、ターゲットのプロセスID (pid) さえわかればアタッチできます（動적アタッチに特別なVMオプションは不要です。開始時に-XX:+DisableAttachMechanismフラグが指定された場合を除き、任意のローカルHotSpot JVMに接続できます）。

```
import com.sun.tools.attach.VirtualMachine;
...
VirtualMachine vm = VirtualMachine.attach(pid);
try {
    vm.loadAgent(agentJarPath, options);
} finally {
    vm.detach();
}
```

このコードが示しているのは、実行中のVMにJavaエージェントをインジェクションする方法です。Javaエージェントとは、アプリケーションのインスツルメント処理を行うユーティリティ・プログラムのことです。Javaエージェントは、`agentmain`メソッドを持つクラスとともにJARファイルに格納しておく必要があります。

インスツルメンテーションAPIを使うと、Javaエージェントで既存クラスのバイトコードを変換できます。インスツルメンテーションAPIを動적アタッチと併用すると、アプリケーションがデバッグ機能なしに開始されていても、実行中のアプリケーションのコードを変更できるようになります。次に、新しいバージョンのMyClassをインストールする単純なエージェントの例を示します。

```
public static void agentmain(String args,
    Instrumentation instr) throws Exception {

    Class oldClass = Class.forName("org.pkg.MyClass");
    Path newFile = Paths.get("/path/to/MyClass.class");
    byte[] newData = Files.readAllBytes(newFile);

    instr.redefineClasses(
        new ClassDefinition(oldClass, newData));
}
```

`defineClasses` APIには制限があることに留意してください。新しいバージョンのクラス・ファイルに新しいメソッドやフィールドを追加することはできません。また、既存のメンバーを削除することもできません。変更できるのは基本的にメソッド本体のみですが、通常、ホット・フィックスにはこれで十分です。

実行中のJavaプロセスのインスツルメント処理を行うことができるため、エンタープライズ・アプリケーションのメンテナンスにとって、アタッチAPIは重要です。動적アタッチのメカニズムには、JVMとの完全な連携処理が必要です。JVMがハング状態や応答しない状態になると、役に立たなくなります。そのような状態になったときは、Serviceability Agentなどで力づくの処理を行う必要があります。

Serviceability Agent

HotSpot Serviceability Agent (SA) は、VM側から見たJavaプロセスの詳細な情報を提供するものです。このエージェントは、ヒープ・レイアウト、システム・ディクショナリ、コンパイルされたコード、スレッド、スタックなど、Java HotSpot VMの内部構造をすべて認識しています。さらに、こういった情報をシンプルでわかりやすいJava API経由で利用できるように、開発者は逆アセンブラやハッキング・ツールを使わずにその情報を活用できます。

もともと、SAはJava HotSpot VMエンジニアがJDK内部のクラッシュをデバッグするために考案したものです。しかしその後、さまざまな開発者グループでも役立つ可能性があることがわかったため、現在は通常のJDKにバンドルされています。SAを使うためには、クラスパスに{JAVA_



HOME}/lib/sa-jdi.jarを含めます。ただし、APIは標準ではなく、将来のJDKリリースによって変更される可能性がある点に留意してください。

通常、カスタム・ツールは既存のToolクラスを拡張して作成します。Toolクラスには、引数の解析や実行中のVMにアタッチする機能がすでに含まれているため、runメソッドをオーバーライドし、その中にカスタム・ロジックを実装するだけで済みます。

```
import sun.jvm.hotspot.runtime.VM;
import sun.jvm.hotspot.tools.Tool;
```

```
public class MyTool extends Tool {

    @Override
    public void run() {
        // 実際の実装
        VM.getVM()...
    }

    public static void main(String[] args) {
        new MyTool().execute(args);
    }
}
```

VM.getVM()は、Java HotSpot VMの内部構造にアクセスする際の起点になります。次の例では、クラス・ローダーでロードしたすべてのクラスに対し、SystemDictionaryを使って横断的なアクセスを行っています。クラスのロードに関連するメモリ・リークの検出にも同様の手法が有用でしょう。

```
VM.getVM().getSystemDictionary()
    .classesDo((klass, loader) -> {
        String className = klass.getName().asString();
        System.out.print(className);

        String loaderName = (loader == null)
```

```
        ?"Bootstrap ClassLoader"
        : loader.getClass().getName().asString();
        System.out.println(" loaded by " + loaderName);
    });
```

かなりシンプルなプログラムです。SAの真の力は、VM構造の復元にあります。この復元は、実行中のJavaプロセスのメモリから行うことができます。また、オペレーティング・システムが異常終了したプロセスのコア・ダンプを作成するように設定されている場合は、そのダンプからも復元できます。SAを使うと、リフレクションによく似たAPIでJavaオブジェクトを検査し、必要なフィールドを抽出することができます。しかし、同じプロセス内で動作するリフレクションとは異なり、SAでは、別プロセスのメモリの読み出しや、コア・ダンプ・ファイルの解析が可能です。この機能に基づいたツールを使うと、実行中のWebサーバーから秘密鍵を盗み出すような衝撃的なこともできます。次のコードは、ターゲット・プロセスのヒープをスキャンしてjava.security.PrivateKeyのインスタンスを探し、その内容を表示するものです。

```
Klass keyClass = VM.getVM().getSystemDictionary()
    .find("java/security/PrivateKey", null, null);
```

```
VM.getVM().getObjectHeap()
    .iterateObjectsOfKlass(new DefaultHeapVisitor() {
        @Override
        public boolean doObj(Oop obj) {
            InstanceKlass c = (InstanceKlass) obj.getClass();
            OopField f = (OopField) c.findField("key", "[B");
            TypeArray key = (TypeArray) f.getValue(obj);
            key.printOn(System.out);
            return false;
        }
    }, keyClass);
```

SAが動作するためには、ターゲットJVMからの協力を得る必要がなく、SAの動作による影響を防ぐ方法はありません。しかし、心配する必要はありません。通常、実行中のプロセスにSAをアタッチするためには、root権限が必要です。また、SAがアタッチされている間、ターゲットJVMは一時停止



した状態になることも覚えておいてください。

ここまでは、JDKの内部APIに注目してきました。もう少し標準的な方法で独自のツールを作成したいという方は、次に説明するJVMツール・インタフェースを検討してください。

JVMツール・インタフェース

JVMツール・インタフェース (JVM TI) は、ソフトウェアのデバッグ、モニタリング、プロファイリングに特化して設計された標準プログラミング・インタフェースであり、JVM上で実行することを想定したものです。JVM TIが優れているのは、特定の実装との結びつきがないパブリックな仕様になっている点です。すべてのJVMですべてのJVM TI機能を提供することが規定されているわけではありませんが、ほとんどの著名なJVMはすべての機能を提供しています。

このインタフェースは、Cヘッダー・ファイルjvmti.hで公開されています。JVM TIベースのツールはエージェントと呼ばれ、通常はCかC++で記述します。エージェントは同じプロセス内で実行され、JVM TI関数を呼び出して直接JVMと通信します。このインタフェースは、Java Native Interface (JNI) に若干似ています。そのため、JNIコードを書いたことがある方なら、JVM TIの基本的な使い方を簡単に理解できるでしょう。

エージェントは、JVMブートストラップで起動 (-agentlibまたは-agentpath JVM引数を指定した場合) することも、後ほど実行時に動的アタッチのメカニズムを使ってロードすることもできます。この2つのオプションをサポートするために、エージェントでは1つまたは複数のエントリ・ポイントを定義します。

- Agent_OnLoad : JVM起動時の早い段階で自動的に呼ばれます。
- Agent_OnAttach : ライブラリが実行時にロードされた際に呼ばれます。

通常、エージェントが最初に行うことは、JVM TI関数の呼出しに必要な

JVMツール・インタフェース (JVM TI) は、ソフトウェアのデバッグ、モニタリング、プロファイリングに特化して設計された**標準プログラミング・インタフェース**であり、JVM上で実行することを想定したものです。

JVM TI環境 (jvmtiEnv) への参照を取得することです。

```
#include <jvmti.h>
```

```
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm, char *args, void *unused) {
    jvmtiEnv *jvmti;
    vm->GetEnv((void**)&jvmti, JVMTI_VERSION_1_0);
```

```
// 初期化コード
```

```
    return 0;
}
```

JVM TIは、デバッガが一般的に行うあらゆる機能を備えています。具体的には、スレッドの管理、スタックの確認、Javaヒープに対する反復処理、ローカル変数の問合せ、ブレーク・ポイントの設定、Javaクラスの操作、ネイティブ・メソッドのインターセプトなどが可能です。それに加えて、エージェントはイベントの通知を受け取ることができます。JVMは、イベントが発生するたびに指定済みのコールバック関数を呼び出します。

JVM TIには、機能ベースでアクセスを行います。すなわち、エージェントは利用する機能を明示的にリクエストする必要があります。大半は実行時に利用できる機能ですが、中にはOnLoadフェーズ (クラスのロードもバイトコードの実行も行われていないタイミング) にのみリクエストできる機能もあります。たとえば、can_access_local_variables機能は、起動時にのみ利用できます。JVMですべてのローカル変数についての情報を保持できるように、特定の最適化機能をあらかじめ無効化しなければならないためです。

次の例では、例外イベントを生成する機能をリクエストし、スローされたすべてのJava例外 (キャッチされたものとされていないものの両方) についての通知を受け取るコールバックを設定しています。

```
jvmtiCapabilities capabilities = {0};
capabilities.can_generate_exception_events = 1;
jvmti->AddCapabilities(&capabilities);
```



```
jvmtiEventCallbacks cb = {0};
cb.Exception = ExceptionCallback;

jvmti->SetEventCallbacks(&cb, sizeof(cb));
jvmti->SetEventNotificationMode(
    JVMTI_ENABLE, JVMTI_EVENT_EXCEPTION, NULL);
```

コールバック関数は、スレッド、メソッド、スローされた例外のバイトコード・インデックスなど、例外についてのあらゆる詳細情報を受け取ります。JNI環境への参照もコールバックに渡されます。つまり、コールバック内から任意のJNI関数を呼び出すことができます。たとえば、JNIを使って `Throwable.printStackTrace()` を呼び出すことも可能です。そのため、エージェントは、無視される例外を含むすべての例外を、キャッチされる直前に出力できます。

```
void JNICALL ExceptionCallback(
    jvmtiEnv *jvmti, JNIEnv *env, jthread thread,
    jmethodID method, jlocation location,
    jobject exception, jmethodID catch_method,
    jlocation catch_location)
{
    jclass cls = env->FindClass("java/lang/Throwable");
    jmethodID print_method = env->
        GetMethodID(cls, "printStackTrace", "()V");
    env->CallVoidMethod(exception, print_method);
}
```

JVM TIを使うと、他にもさまざまな便利な機能を実現できます。例外だけでなく、クラスのロード、ガベージ・コレクション、ロックの競合、スレッドのアクティビティなどもトレースできます。

JVM TIは、Javaデバッガ・エージェントと混同されることがよくあります。また、多くの方が、JVM TIはセキュリティを弱め、Javaアプリケーションのパフォーマンスを低下させると誤解しています。

しかし、Java Debug Wire Protocol (JDWP) エージェントは、JVM TIをベースとするツールの1つの例にすぎません。このテクノロジー自体は、セキ

ュリティやパフォーマンスに影響するものではありません。アプリケーションがエージェントのオーバーヘッドによって影響を受けるかどうかは、エージェントがどの機能をリクエストして何を行うかのみにかかっています。JVM TIは、JNIの拡張機能のようなものと考えてください。このテクノロジーを試してみる価値は十分にあります。

JDK 9での変更点

以上で説明したすべてのテクノロジーは、プライベートAPIを含め、今後リリースされる予定のJDK 9にも機能が引き継がれます。ただし、新しいモジュール・システムによって、これらのAPIにアクセスする方法が多少の制限を受けることになります。tools.jarやsa-jdi.jarも個別のライブラリではなくなる予定です。JDK 9のサービサビリティ機能は、専用のモジュールで提供されます。表1に、Java 9における主要なJARファイルの場所を示します。

デフォルトでは、アプリケーションは、パッケージを外部にエクスポートしていないモジュールからAPIにアクセスすることはできません。プライベートAPIを使用するためには、`--add-exports` JVMコマンドライン引数で明示的にカプセル化を解除する必要があります。たとえば、`sun.jvmstat.monitor` パッケージを利用するツールを実行するためには、次のようにします。

機能	モジュール	パブリック
JVMSTAT パフォーマンス・カウンタ	jdk.jvmstat	いいえ
動的アタッチAPI	jdk.attach	はい*
インスツルメンテーションAPI	java.instrument	はい
SERVICEABILITY AGENT	jdk.hotspot.agent	いいえ
* com.sun.tools.attach.VirtualMachineには外部からアクセスできますが、sun.tools.attach.HotSpotVirtualMachineにはアクセスできません。		

表1: Java 9でのサービサビリティAPIの場所



```
java --add-exports jdk.jvmstat/sun.jvmstat.monitor=
ALL-UNNAMED MyTool
```

[編集注:この行は、=記号の後にスペースを入れずに1行で記述する必要があります。]

個人用途であれば、プライベートAPIは自由に使って構いません。ただし、その際は十分注意してください。APIが将来のJDKアップデートで利用できる保証はありません。

まとめ

Javaプラットフォームには、デバッグ、モニタリング、トラブルシューティングを行うことができるカスタム・ツールを構築するための多様なテクノロジーが搭載されています。その中には、Java SE標準でカバーされているものも、OpenJDKとOracleのJDKに固有なものもあります。プライベートAPIに詳細なドキュメントはありませんが、サービスサビリティ・テクノロジーについて学習する際には、OpenJDKプロジェクトのソース・コード（とりわけ、JDK組込みツールのソース・コード）はよい着手点になるでしょう。

適切なツールなしにソフトウェアの開発やメンテナンスが成功することはほとんどありません。多くのツールが販売されていますが、あらゆる問題に対処できる特効薬はありません。Java開発者である皆さんは、独自のツールを作って、他のソフトウェアでは解決できないタスクを解決できます。JDKのサービスサビリティ・テクノロジーは皆さんの力強い味方です。 </article>

Andrei Pangin (@AndreiPangin) : Odnoklassnikiソーシャル・ネットワークの開発リーダー。以前はHotSpot JVMに携わっており、このJVMの話題を専門とする。Javaカンファレンスで頻繁に講演を行っているほか、Stack OverflowでもJVM関連の質問に数多く解答している。

ORACLE[®]
UNIVERSITY

Get Java Certified

Earn Your #javatshirt

Danny Muthama
@DannyMuthama
#oca #javatshirt @OracleCert achievement unlocked
8 Dec 2016

swati sharma
@swati_sharma_
Got my new #javatshirt. Preparing for OCP
16 Oct 2016

Abdullah Shabbir
@FarigLog
Thanks Oracle for the Java tshirt. Now preparing for Java OCP. #javatshirt
14 Sept 2016

백상빈
@bsbin150
#javatshirt 이벤트 참여 완료 ~
27 Aug 2016

The Java certification recognition t-shirt is available to all candidates who earn a Java certification. Offer expires March 31, 2017.

ORACLE[®]



CONOR SVENSSON

ブロックチェーン： Javaで暗号通貨を使う

web3jによりJavaアプリケーションにEthereumブロックチェーンを統合する

テクノロジーや金融関連の報道において、ブロックチェーンが話題に上がらない日はほとんどありません。しかし、なぜこのテクノロジーがこれほど騒がれているのでしょうか。また、どうすればこれをJavaアプリケーションから使うことができるのでしょうか。その連携を実現するライブラリであるweb3jに話を移す前に、まずは、ブロックチェーンとは何であり、どのように動作するのかを説明することにします。

簡単な歴史

ブロックチェーン・テクノロジーは、暗号通貨ビットコインとともに始まりました。ビットコインが登場したのは2008年のことです。ビットコインは、最初に提唱された暗号通貨ではないものの、発行にも取引の検証にも中央機関を必要としない初めての完全非中央集権型暗号通貨でした。ビットコインは、すべてのビットコイン取引の明細が含まれる分散型の台帳を管理しています。すべてのビットコインの所有権はこの台帳のエントリから求められます。このエントリが、ビットコインのブロックチェーンです。ブロックチェーン上のトランザクションは、簡単なスクリプト言語で生成されます。

その5年後の2013年になると、19歳のVitalik Buterin氏が新しい非中央集権型プラットフォームの開発を始めました。このプラットフォームはビットコインの考え方に基づいたもので、アプリケーションの開発用にさらに堅牢なスクリプト言語を提供するものでした。Ethereumと名付けられたこのプラットフォームによって、チューリング完全な言語が提供されることになりました。Ethereumが初めて提唱されたのは2014年のはじめてで、2015年7月にリリースされました。

それ以来、さまざまなグループがいくつかのブロックチェーン・テクノロジーを公開しているものの、現在群を抜いてもっとも成熟（と言えるの

であれば）しているのがEthereumです。本記事は、このEthereumを取り上げます。

Ethereumブロックチェーンでは、実績ある暗号通貨Etherが使われます。Etherはビットコインに次いで2番目に普及している暗号通貨で、時価総額は約10億米ドルです。ちなみに、ビットコインの時価総額は100億米ドルです。

Ethereumは、安全な非中央集権型インターネット・バックエンドを提供することを意図したものです。発生するのはピア同士の通信であり、単一のエンティティや組織と直接やり取りする必要はありません。本記事では、Ethereumと連携するための軽量Javaライブラリweb3jを使用します。

ブロックチェーンとは

ブロックチェーンは、状態遷移を受ける非中央集権型の不変データ構造と考えることができます。ブロックチェーンに対するトランザクションや操作が繰り返し適用された状態は、その詳細がブロックチェーンに書き込まれます。このようなトランザクションは、連結されたブロックに分かれており、そのブロックが集まってブロックチェーンを形成しています（図1参照）。イベント・ソーシングに使われるイベント・ログとよく似ていますが、以前に起こった状態遷移を再生することによって現在の状態を導き出すことができます。

ブロックチェーンは分散型台帳テクノロジー（DLT）です。この用語は、ブロックチェーンのような非中央集権型データ・ストレージを提供するテクノロジーを表現するために生まれたものです。すべてのDLTの内部でEthereumのように共通のブロックチェーンが使われているわけではありません。取引に参加した当事者間にしかデータを公開しないDLTもありま



す。

Ethereumでは、データはEthereum仮想マシン (EVM) に書き込まれ、その中で状態遷移が起こります。EVMは、Java仮想マシンと同様に、バイトコード形式を解釈してトランザクション内の命令を実行します。しかし、JVMとは異なり、EVMは完全非中央集権型ノードのネットワークを使用します。各ノードは命令をローカルで実行し、ローカルのブロックチェーンが以前にEthereumブロックチェーンに書き込まれたものと一致することを確認します。あるノードが新しいブロック (新しいトランザクションを含むもの) の追加を成功させるには、新しい状態がどうなるべきかについて、ネットワーク上の大半のノードと合意する必要があります。この新しいブロックを作成するノードを採掘者と呼びます。

採掘

採掘ノードは、プルーフ・オブ・ワークと呼ばれる合意メカニズムを使用します (偶然ですが、これはビットコインの合意メカニズムと同じスタイルです)。ブロックチェーン上に新しいブロックを作成するためには、コンピュータの計算能力が必要です。ブロックは、まだブロックに割り当てられていない新しいトランザクションのグループを継続的にハッシングすることによって作成されます。事前に定義された計算問題 (これを採掘難易度と呼びます) を解くトランザクションの組合せが見つかったら、解答を見つけた採掘者はネットワークから報酬、すなわち新しく造られた通貨が与えられ、ブロックがブロックチェーンに追加されます。

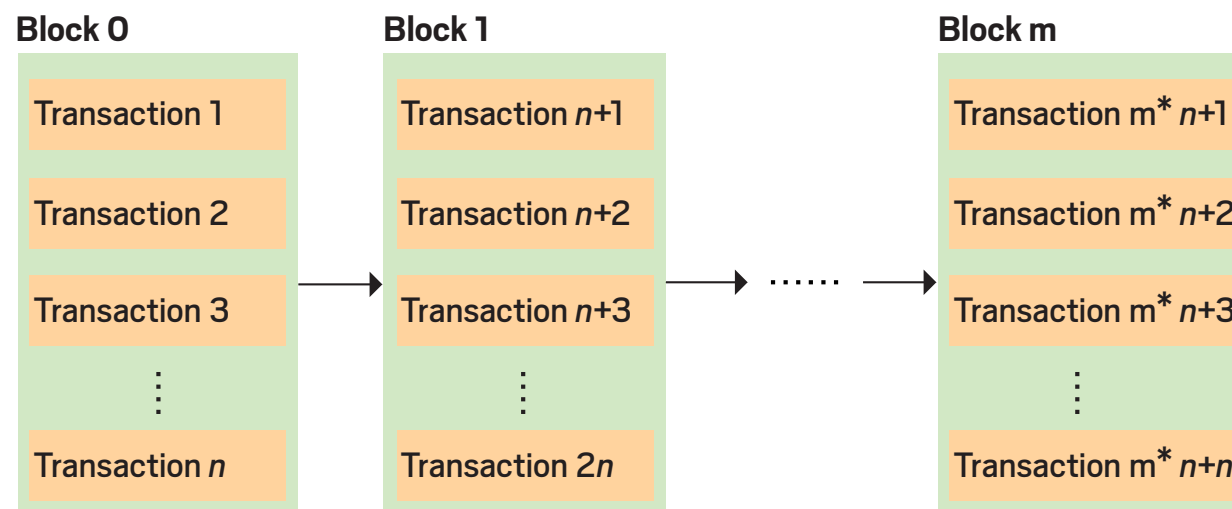


図1: ブロックチェーンの構造

スマート・コントラクト

EVMで実行するプログラムをスマート・コントラクトと呼びます。スマート・コントラクトは、コンピュータにおける契約です。Ethereumのスマート・コントラクトは、コード (関数) とEthereumブロックチェーンのアドレスに関連付けられたデータ (状態) の集合体です。

Ethereumを使ってみる

Ethereumネットワークを使ってみるには、Ethereumクライアント (ノードまたはピアとも呼びます) にアクセスする必要があります。Ethereumクライアントは、Ethereumブロックチェーンと相互に同期し合い、ブロックチェーンとやり取りするためのゲートウェイを提供します。

完全なEthereumクライアントとして普及しているのが、[Geth](#)と[Parity](#)の2つです。

いずれかのクライアントをダウンロードして起動します。

```
$ geth --rpcapi personal,db,eth,net,web3 \
--rpc --testnet
```

```
$ parity --chain testnet
```

これでクライアントが開始され、testnet Ethereumブロックチェーン (これは数ギガバイトの大きさです) との同期が始まります。Ethereumネットワークには、mainnetとtestnet (Ropstenとも呼ばれています) という2つのパブリック・バージョンがあります。これらのバージョンはそれぞれ、本番環境とテスト環境を示しています。本記事ではtestnetを使用します。そうしないと、実際のお金を使うことになってしまいます。

また、最新バージョンの[web3j](#) コマンドライン・ツールも必要になります。このツールには、Ethereumを使う際に便利ないくつかのユーティリティ・コマンドが含まれています。

mainnetまたはtestnetでの取引には、Ether暗号通貨が必要です。

トランザクションの処理には、ユーザーが公開鍵と秘密鍵のペアを持つ公開鍵暗号化が使われます。ユーザーが秘密鍵 (ユーザーのみが知っている鍵) を使用して署名したトランザクションが、公開鍵とともに公開されます。他のネットワーク参加者は、公開鍵とトランザクションの署名に基づいて、取引が真正であることを検証できます。

Ethereumやweb3jのユーザーにとってありがたいことは、公開鍵暗号化を除いて考えられる点です。そのため、Ethereumと取引をするためのアカウント資格証明を含むデジタル・ファイルであるEthereumウォレットだけを扱えばよいことになります。ユーザーは、任意の数のウォレット・ファイルを持つことができます。ウォレット・ファイルには、パスワードで暗号化された秘密鍵と、公開鍵から得られたアドレスが含まれています（なお、公開鍵とアドレスはいずれも秘密鍵から得ることができます）。ネットワーク上のあらゆるトランザクションは、そのようなアドレスに紐付けられています。

次のようにweb3jコマンドライン・ツールを実行すると、安全なEthereum用ウォレット・ファイルを新規作成できます。

```
$ ./web3j-1.0.7/bin/web3j wallet create
```

```

      _ _ _ _ _
      || |__ ( ) ( )
    _ _ _ _ _ || _ // _ _ _ _
  \ \ / / _ \ ' \ \ \ | || / _ \
   \ V / _ / | | _ // / | || ( ) |
    \ \ / \ _ | _ \ / \ | | ( ) | \ _ /
              _/ |
              | _/

```

Please enter a wallet file password:

Please re-enter the password:

Please enter a destination directory location

[~/ethereum/testnet-keystore]:

Wallet file <timestamp>--<UUID>.json created in:

~/ethereum/testnet-keystore

パスワードには安全なものを選び、忘れないようにします。パスワードを忘れた場合や、何者かがウォレット・ファイルを盗み出してパスワードが知られてしまった場合、二度と資金を取り戻すことはできなくなります。

ウォレットのアドレスがわかったら、Etherscanにアクセスして現在の残高とこのアドレスに関連するすべてのトランザクションの詳細を確認します。図2にEtherscanでのウォレット・アドレスの残高を示します。

testnetではEtherを簡単に採掘できるようになっているため、クライアントを採掘モードで起動すると、ネットワーク上で取引を行うのに十分なEtherを数分で手に入れることができます。

採掘の実行方法の詳細については、[Geth](#)または[Parity](#)のドキュメントをご覧ください。

web3jを使ってみる

Ethereumクライアントが実行されていれば、web3j経由でクライアントに接続するJavaコードを書く準備は完了です。なお、以降のコードの完全なリストは、[GitHub](#)で公開されています。

Ethereumクライアントは、JSON-RPCにいくつかのメソッドを公開しています。これらのメソッドは、JSONを使用したステートレスなリモート・プロシージャ・コール (RPC) プロトコルです。web3jはすべてのEthereum JSON-RPC APIをサポートしています。しかし、web3jは単にJSON-RPCプロトコルのラッパーを提供しているだけではなく、その背後でさまざまな処理も行っています。JSON-RPC APIはHTTPやプロセス間通信 (IPC) で利用でき、GethではWebSocketも実装されています。ただし、もっとも一般的な実装はHTTPであり、以下の例ではHTTPを使っています。

web3jのリリースは、Maven Centralと、JFrogのJCenterの両方で公開されています。本記事の執筆時点での最新リリースは1.0.9ですが、[web3jメイン・プロジェクト・ページ](#)を確認して最新のリリースをプロジェクトに使用することをお勧めします。それでは、関連する次の依存性をビルド・ファイルに追加します。

Maven:

```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core</artifactId>
  <version>1.0.9</version>
</dependency>
```

Gradle:

```
compile ('org.web3j:core:1.0.9')
```

続いて、実行可能クラスに次のコードを加えます。このコード



は、Ethereumクライアントのバージョンを表示するためのものです。

```
// デフォルトはhttp://localhost:8545/
Web3j web3 = Web3j.build(new HttpService());
Web3ClientVersion clientVersion =
    web3.web3ClientVersion().sendAsync().get();

if (!clientVersion.hasError()) {
    System.out.println("Client is running version: " +
```

```
clientVersion.getWeb3ClientVersion());
}
```

コードに問題がない場合、実行するとEthereumクライアントの詳細なバージョンが表示されます。

Client is running version:
Geth/v1.5.4-stable-b70acf3c/darwin/go1.7.3

Etherscan The Ethereum Block Explorer

MORDEN TESTNET Search by Address / Txhash / BlockNo GO LANGUAGE

HOME BLOCKCHAIN ACCOUNT TOKEN MISC

Address 0x19e03255F667BdFD50A32722df860B1Eeaf4d635 Home / Normal Accounts / Address

Overview

ETH Balance: 2,911.497469554 Ether (\$32,841.69)

Mined: 566 blocks and 29 uncles

No Of Transactions: 68 txns

Misc

Contracts created: 39 Contracts

QR CODE

Transactions Mined Blocks Mined Uncles

Latest 25 txns from a total Of 68 transactions View All

TxHash	Block	Age	From	To	Value	[TxFee]
0x4ae6990fd070423...	1809881	2 days 22 hrs ago	0x19e03255f667bdf...	OUT 0x10f8f215ca0249a9...	0 Ether	0.0022557
0xe175356ebe34fa2...	1809879	2 days 23 hrs ago	0x19e03255f667bdf...	OUT 0x10f8f215ca0249a9...	0 Ether	0.00255905
0x1d553a58e9d706...	1809877	2 days 23 hrs ago	0x19e03255f667bdf...	OUT Contract Creation	0 Ether	0.0351496

図2: EtherscanでのEthereumウォレット・アドレスの残高

web3jのすべてのJSON-RPCメソッド実装では、次の構造が使われています。ここで、`web3`はweb3j実装のインスタンスを示します（このインスタンスがリクエストを管理しています）。

```
web3.<method name>([param1, ..., paramN])
    .[send()|sendAsync()]
```

メソッド名とパラメータはJSON-RPC仕様に従います。リクエストの送信は、`send()`を使うと同期的に、`sendAsync()`を使うと非同期的に行われます。

これで、Ethereumクライアントへの接続が確認され、Ethereumブロックチェーンを使う準備ができました。

Ethereumブロックチェーン上のトランザクション

通常、Ethereumブロックチェーン上に新しいトランザクションを作成するためには、次に示す3つのアクションのいずれかを行います。

- あるアカウントから別のアカウントにEtherを送金する
- 新しいスマート・コントラクトをデプロイする
- 既存のスマート・コントラクトの状態を変化させるメソッド・コールを発行する

既存のスマート・コントラクトを調べる、読取り専用のメソッド・コールも別途あります。このメソッド・コールを実行しても、ブロックチェーン上にトランザクションは作成されません。

このようなトランザクションのやり取りを行うには、Ethereumクライアントに対して背後でJSON-RPC経由の複数のコールを行う必要があります。web3jはそういった低レベルの機能を処理しますが、ユーザーが独自の実装を望む場合に備えて、すべてのJSON-RPCコールが行えるようになっています。

Etherの送金

もっとも基本的なタイプのトランザクションから説明を始めます。ここでは、あるアカウントから別のアカウントに0.2 Etherを送金します。先ほど作成したEthereumウォレット・ファイルの場所を忘れないようにしてください。この情報がないと、一切送金はできません。

```
Web3j web3 = Web3j.build(new HttpService());
```

```
Credentials credentials =
    WalletUtils.loadCredentials(
        "my password", "/path/to/walletfile");
```

```
TransactionReceipt transactionReceipt =
    Transfer.sendFundsAsync(
        web3, credentials,
        "0x...", BigDecimal.valueOf(0.2),
        Convert.Unit.ETHER).get();
```

```
System.out.println(
    "Funds transfer completed, transaction hash: " +
    transactionReceipt.getTransactionHash() +
    ", block number: " +
    transactionReceipt.getBlockNumber());
```

このコードを実行すると、次のように出力されます（読みやすくするため、トランザクション・ハッシュの一部を省略し、行を折り返しています）。

```
Funds transfer completed, transaction hash:
0x16e41aa9d97d1c3374a...34,
block number:1840479
```

トランザクションとブロックのハッシュは、Ethereumブロックチェーン上のトランザクションの識別子となります。この背後では、次のような実際のトランザクション処理（図3）が行われています。

1. Ether送金リクエストがweb3jに送信される。
2. トランザクション・メッセージがEthereumクライアントに送信される。
3. クライアントがトランザクションを検証する。続いて以下の処理を行う。
 - a. トランザクションを他のEthereumノードに伝える。
 - b. 送信されたトランザクションのハッシュを取得し、同期HTTPレスポンスとしてクライアントに送信する
4. 採掘者は、このトランザクションと他の新しいトランザクションを組



み合わせ、Ethereumネットワーク上にブロックを形成する。有効なブロックが形成されると、ブロックとそれに関連するトランザクションの詳細がブロックチェーン上で確定される。

再びEtherscanにアクセスすると、トランザクションの詳細を確認できます(図4)。

図5は、トランザクションが存在するブロックチェーン上のブロックの内容を示しています。

ガス

先ほど触れたように、EtherはEVMでのコード実行の対価として支払われるものです。トランザクションに対して支払うコストに関するパラメータとして、ガス価格とガス上限という2つのパラメータを指定する必要があります。ガス価格は、ガス単位あたりで支払う額で、単位はEtherです。各EVMのオペコードには、コードに対応するガス・コストが含まれています。ガス上限は、トランザクションを実行する際に消費してもよいというガス使用量の合計値です。これによって、すべてのトランザクションが有限の

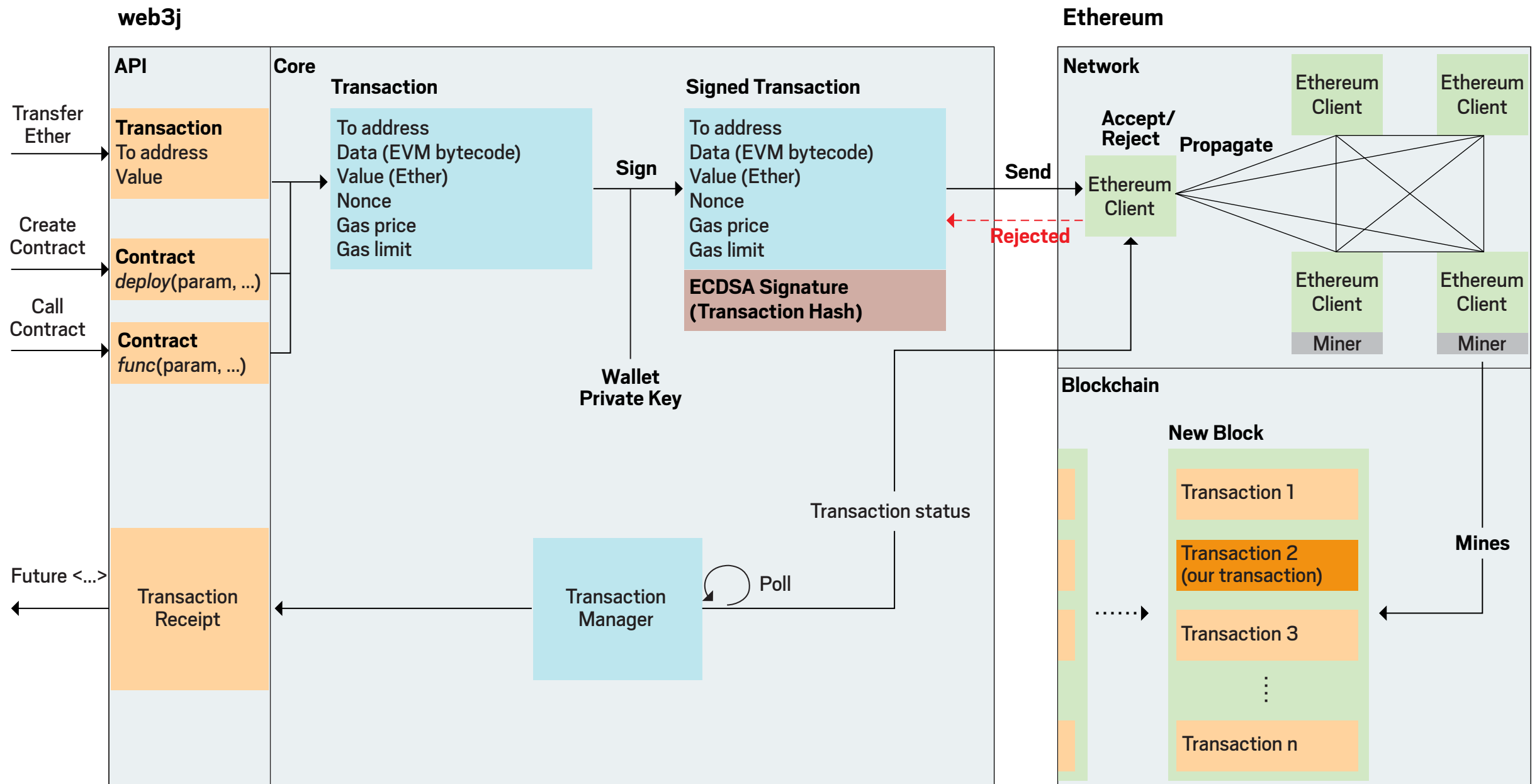


図3: web3jによるEthereumトランザクション

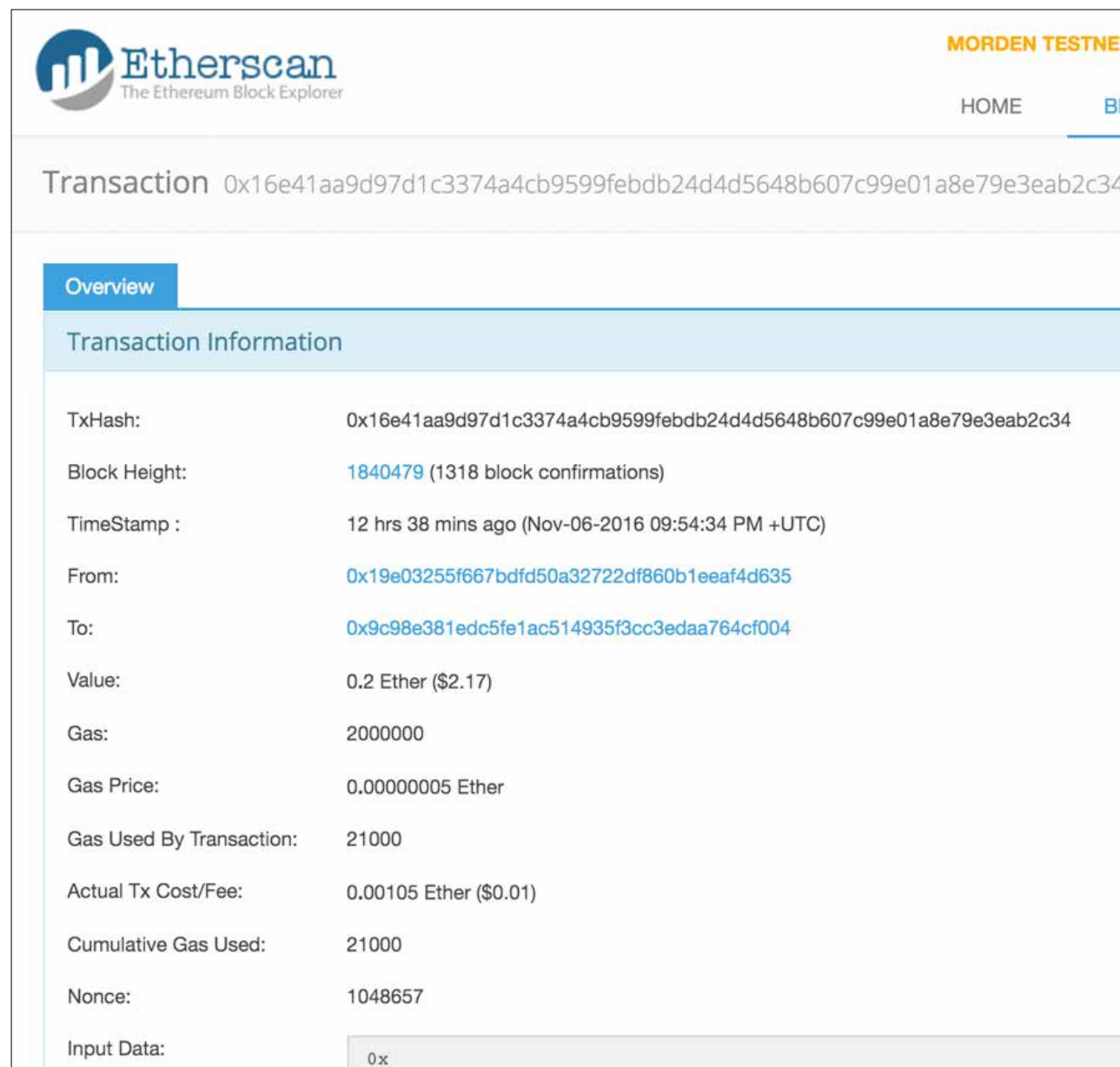
実行コストを持つことが保証されます。トランザクションに関連するガスの詳細は、トランザクションのレシートとEtherscanで確認できます。

EthereumでのHello World

単純なトランザクションについての説明が終わり、いよいよおもしろい部分に入ってゆきます。早速、最初のスマート・コントラクトを作成してみます。通常、Ethereumのスマート・コントラクトはSolidityという静的に型付けされた高水準言語で書かれます。Solidityの使用方法を説明すると際

限がないため、ここではできる限り簡単な例で説明します。Solidityの詳細については、[オンライン](#)で確認できます。

ここでは、[Greeterコントラクトの例](#)を使って説明します。Greeterコントラクトは、Ethereumのスマート・コントラクトにおける「Hello World」の例です。コントラクトをデプロイする際に、UTF-8でエンコードした文字列をコンストラクタに渡します。その後、デプロイしたコントラクトを呼び出すたびに、リクエストを処理するEthereumネットワーク上のノードからその文字列の値が返されます。




 MORDEN TESTNET	
Transaction 0x16e41aa9d97d1c3374a4cb9599febdb24d4d5648b607c99e01a8e79e3eab2c34	
Overview	
Transaction Information	
TxHash:	0x16e41aa9d97d1c3374a4cb9599febdb24d4d5648b607c99e01a8e79e3eab2c34
Block Height:	1840479 (1318 block confirmations)
TimeStamp :	12 hrs 38 mins ago (Nov-06-2016 09:54:34 PM +UTC)
From:	0x19e03255f667bdfd50a32722df860b1eeaf4d635
To:	0x9c98e381edc5fe1ac514935f3cc3edaa764cf004
Value:	0.2 Ether (\$2.17)
Gas:	2000000
Gas Price:	0.00000005 Ether
Gas Used By Transaction:	21000
Actual Tx Cost/Fee:	0.00105 Ether (\$0.01)
Cumulative Gas Used:	21000
Nonce:	1048657
Input Data:	0x

図4： トランザクションのレシート

```
contract mortal {
  /* address型の変数ownerを定義する */
  address owner;

  /* この関数は初期化時に実行され、
  コントラクトの所有者を示すownerを設定する */
  function mortal() { owner = msg.sender; }
  /* コントラクト上の資金を回収するための
  関数 */
  function kill() {
    if (msg.sender == owner) suicide(owner);
  }
}
```

```
contract greeter is mortal {
  /* string型の変数greetingを定義する */
  string greeting;

  /* コントラクトが実行されるたびに実行される */
  function greeter(string _greeting) public {
    greeting = _greeting;
  }

  /* メイン関数 */
  function greet() constant returns (string) {
    return greeting;
  }
}
```




```
private Greeter(
    String contractAddress, Web3j web3j,
    Credentials credentials,
    BigInteger gasPrice, BigInteger gasLimit) {
    super(
        contractAddress, web3j, credentials,
        gasPrice, gasLimit);
}

...

public Future<Utf8String> greet() {
    Function function = new Function("greet",
        Arrays.<Type>asList(),
        Arrays.<TypeReference<?>>asList(
            new TypeReference<Utf8String>() {}));
    return executeCallSingleValueReturnAsync(
        function);
}

public static Future<Greeter> deploy(
    Web3j web3j, Credentials credentials,
    BigInteger gasPrice, BigInteger gasLimit,
    BigInteger initialValue,
    Utf8String _greeting) {
    String encodedConstructor =
        FunctionEncoder.encodeConstructor(
            Arrays.<Type>asList(_greeting));
    return deployAsync(
        Greeter.class, web3j, credentials,
        gasPrice, gasLimit,
        BINARY, encodedConstructor, initialValue);
}

public static Greeter load(
```

```
String contractAddress, Web3j web3j,
Credentials credentials,
BigInteger gasPrice, BigInteger gasLimit) {
    return new Greeter(
        contractAddress, web3j, credentials,
        gasPrice, gasLimit);
}
}
```

これで、スマート・コントラクトをデプロイして呼び出せるようになります。

```
Credentials credentials =
    WalletUtils.loadCredentials(
        "my password", "/path/to/walletfile");

Greeter contract = Greeter.deploy(
    web3, credentials, BigInteger.ZERO,
    new Utf8String("Hello blockchain world!"))
    .get();
```

```
Utf8String greeting = contract.greet().get();
System.out.println(greeting.getTypeAsString());
```

このコードを実行すると、次の出力が表示されます。

Hello blockchain world!

次は、もう少し複雑なスマート・コントラクトについて見てみます。

Solidityの型に関する注意事項

Solidityには、いくつかの種類のネイティブ型が存在します。その中には、Javaで利用できる型に似たものもありますが、web3jではJavaのネイティブ型をSolidityの型に変換する必要があります。この要件は、Ethereumブロックチェーンに対して書込みや読取りを行うデータの一貫性を保証するためのものです。



なお、EVMではデフォルトで符号なしの256ビット整数を使用することを覚えておいてください。web3jで[BigInteger](#)型を使うことになるのはそのためです。

独自の仮想トークンの発行

スマート・コントラクトを使うと、トークン所有権の発行や管理を行うことができます。ここで言うトークンとは、現実の資産（場合によっては独自の仮想通貨）の所有比率を表すものです。たとえば、ある財産の共同所有権を表すスマート・コントラクトが考えられます。その場合、この財産への出資者には、所有比率を示す数のトークンが提供されることになるでしょう。

Ethereumでは、[トークン用の標準コントラクトAPI](#)が定義されています。このAPIには、スマート・コントラクトを扱うための次のメソッドが定義されています。

// トークンの全供給量

```
function totalSupply() constant  
    returns (uint256 totalSupply)
```

// アドレスに紐付けられたトークン

```
function balanceOf(address _owner) constant  
    returns (uint256 balance)
```

// 送信者のアカウントからアドレスにトークンを送信

```
function transfer(address _to, uint256 _value)  
    returns (bool success)
```

// 消費者_spenderに対し、最大_valueのトークンの

// 消費を許可

```
function approve(address _spender, uint256 _value)  
    returns (bool success)
```

// 許可された消費者による、トークンの送信の委譲

```
function transferFrom(address _from, address _to,  
    uint256 _value) returns (bool success)
```

// 消費者が作成を許可されているトークン数の

// 取得

```
function allowance(address _owner, address _spender)  
    constant returns (uint256 remaining)
```

このスマート・コントラクトには、イベントも導入されています。イベントは、ブロックチェーン上でスマート・コントラクトが実行されている間にEthereumで具体的な明細を記録する際に使用します。Ethereumでは、スマート・コントラクトのトランザクションは値を返せないため、発生したトランザクションについての情報を問い合わせる際はこれらのイベントを使用します。

// トークンの送信を通知

```
event Transfer(address indexed _from,  
    address indexed _to, uint256 _value)
```

// トークンの送信の委譲が許可されたことを

// 通知

```
event Approval(address indexed _owner,  
    address indexed _spender, uint256 _value)
```

Consensysが、[このスマート・コントラクトの完全な実装](#)を公開しています。ダウンロードした実装は、次のようにしてSolidityコンパイラでコンパイルできます。

```
$ solc HumanStandardToken.sol --bin --abi \  
    --optimize -o build/
```

続いて、次のようにしてこのスマート・コントラクトのラッパーを生成できます（ここでも、見やすくするためにフルパスは省略しています）。

```
$ ./web3j-1.0.7/bin/web3j solidity generate \  
    build/HumanStandardToken.bin \  
    build/HumanStandardToken.abi \  
    -p org.web3j.example.generated -o src/main/java/
```



これで、自分のトークンの発行といった処理を行えるようになります。

```
// スマート・コントラクトのデプロイ
HumanStandardToken contract = HumanStandardToken
    .deploy(
        web3, credentials, BigInteger.ZERO,
        new Uint256(BigInteger.valueOf(1000000)),
        new Utf8String("web3j tokens"),
        new Uint8(BigInteger.TEN),
        new Utf8String("w3j$"))
    .get();
```

```
// 発行されている全供給量を表示
Uint256 totalSupply = contract.totalSupply().get();
System.out.println("Token supply issued: " +
    totalSupply.getValue());
```

```
// トークンの残高を確認
Uint256 balance = contract.balanceOf(
    new Address(credentials.getAddress()))
    .get();
System.out.println("Your current balance is: w3j$" +
    balance.getValue());
```

```
// トークンを別のアドレスに送信
TransactionReceipt transferReceipt =
    contract.transfer(
        new Address("0x<宛先アドレス>"),
        new Uint256(BigInteger.valueOf(100))).get();
```

完全な例は、本記事の[付属コード](#)をご覧ください。

まとめ

本記事では、Ethereumブロックチェーンの処理について、ほんの一部を紹介しました。やむを得ず、詳しく説明しなかった部分やまったく触れなかった部分も数多くありますが、本記事が、このすばらしいテクノロジーの機能を理解する助けになれば幸いです。

さらに詳しい情報については、[web3jプロジェクトのソース・コード](#)や[ドキュメント](#)を参照してください。この短い記事で説明できるよりもはるかに多くのEthereumやweb3jの背景情報が満載されています。

</article>

Conor Svensson (@conors10) : アプリケーションに Ethereum ブロックチェーンを統合するための Java ライブラリ web3j の作者。以前に coHome、Huffle というスタートアップ企業を共同設立し、現在は Othera でブロックチェーン融資プラットフォームおよび取引所の構築を行っている。テクノロジーや金融に関する ブログ を執筆しており、スクリーンの前にいないときは、オーストラリアのシドニー近郊にある マルブラ という地元の砂浜でサーフィンを楽しんでいる。

ADRIAAN
MOORS

Scala: 完全な関数型と 純粋なオブジェクト指向の融合

JVMのための成熟した実用的な型保証言語

Scalaは、パターン・マッチングや不変コレクションといった包括的な関数型プログラミング (FP) 機能一式を搭載し、強力な型推論とコンパクトな構文を使用する実用的なオブジェクト指向JVM言語です。多種多様な業界で広く採用されており、そのコードベースには数百万行のものまであります。また、Apache Sparkのおかげで、Scalaはデータ・サイエンス・コミュニティでも広く使われています。

Scalaの中核をなす特徴は、Javaとの相互運用性です。バージョン2.12には、Java 8プラットフォームの機能も含まれるようになっています。たとえば、Scala 2.12ではJava 8と同じようにラムダ式をコンパイルできます。同様に、Scalaトレイトはデフォルト・メソッドを持つJavaインタフェースにコンパイルできます。

Scalaの起源

もともとScalaは、スイス有数の工科大学であるスイス連邦工科大学ローザンヌ校 (EPFL) のMartin Odersky氏らの研究グループが開発したものです。Odersky氏は、Scalaを設計する前にGeneric Java (GJ) を共同設計していました。これによって、FPからJavaにジェネリクスが導入されることになり、Odersky氏はそのためのコンパイラを開発しました。Sun Microsystemsは、バージョン1.3からGJコンパイラを標準のjavacコンパイラとして採用しました (ジェネリクスはJava 5で有効になりました)。

その間に、Odersky氏らの研究グループはScalaに取りかかり、2006年にバージョン2.0をリリースしました。2007年前後には、TwitterやFoursquareといった著名なインターネット関連のスタートアップ企業が本番環境でScalaを使い始め、それ以来採用が拡大しています。2011年には、Akkaミドルウェア・フレームワークを使ったマルチコアやクラウド・コンピューティング用の言語としてのScalaの商用利用を促進するため

に、Typesafeという企業が設立されました。

現在、Typesafeは社名をLightbendに変更しています。Lightbendに在籍する筆者のチームは、活発なオープンソース・コミュニティと連携してScalaコンパイラやライブラリの開発を続けています。最新リリースに対するコミットの3分の1はコミュニティによるものです。また、その他のベンダーもScalaに投資しています。たとえば、JetBrainsのIntelliJ IDEA用Scalaプラグインは500万回近くダウンロードされています。

Scalaの理念

本記事では、Scalaの中核をなすいくつかの概念を説明し、小さなコード・スニペットを使って例示します。Scalaを大きな言語だと見なしている開発者もありますが、実際は純粋なオブジェクト指向型の小さなコアがあるだけです。このコアは非常に柔軟性が高く、さまざまな形を取ることができます。ぜひこの点を理解していただきたいと思います。

Scalaの設計の背景となった重要なアイデアは、関数型プログラミングとオブジェクト指向プログラミングは互いに補完し合うものであるため、1つの言語にうまく組み合わせるというものです。FPには複雑な型と難解な数学という印象が付きまとっていますが、ScalaでのFPの真の魅力は、わかりやすくスケール・アップしやすい形で共通タスクの実装が効率化される点にあります。Java 8で普及したラムダ式がこのよい例です。

キーワードの選定 (不変変数には`val`、可変変数には`var`) から、変数名の前でなく後に型帰属がある点 (型推論のおかげで多くの場合は省略可能)、パターン・マッチング (オブジェクト指向コアと密接に統合)、ワイルドカードよりも定義側変位を好む点 (ワイルドカードを使うとユーザーのコードが煩雑になりやすい) に至るまで、関数型思考と関数型設計は、ScalaのDNAに深く根付いています。



Scalaの関数は、いくつかの入力のみから結果が決まる単一の計算を表す際に適した小さな抽象化の単位です。関数は分離した方がわかりやすく、多くのコアやマシンに分散しやすいものとも言えます。FPでは、オブジェクトがオブジェクト指向プログラミングで担っている役割を関数が果たすこととなります。関数は第一級オブジェクトであり、プログラムの動作を決定する鍵となる構成要素です。オブジェクトとクラスは、多くのさまざまなメンバー間の相互作用をカプセル化する大きな抽象化を表します。関数を作ることはアルゴリズムを表しますが、オブジェクトを作ることはシステムのモデル化を表します。

Scalaは、型推論によって、静的型チェックの安全性やパフォーマンスと、スクリプト言語の使いやすさを両立させています。型推論のおかげで、ほとんどの型アノテーションを適宜省略できます。型がよく省略されるのは、可読性が低下して、将来的なリファクタリング、プロトタイプの迅速な開発、またはScalaの対話型シェルやデータ・サイエンス・ワークシートでの新たな問題の調査の妨げとなる場合です。

多くのプログラマーは、FPは可変状態（もう少し一般的に言えば、副作用）を否定し、遅延評価や深い数学的理由を要求するものと考えています。Scalaでは、関数型プログラミングをそのようにとらえてはいません。コードは、関数型であろうとオブジェクト指向であろうと、いずれ外の世界に必ず影響を与えるものです。しかし、通常、不必要な副作用や予期しない副作用はメンテナンス性やスケーラビリティに悪影響を与えます。これは、関数型プログラミングとオブジェクト指向プログラミングの両方に言えることです。Scalaでは、不変変数や再帰を好む「純粋な」プログラミングか、可変変数やwhileループを使用する命令型に近いプログラミングかの選択権はプログラマーに与えられます。

しかし確かに、Scalaは副作用から離れるように導こうとしています。Scalaには、文という独立した概念はありません。文は単なる結果を計算するための式と見なされるからです。Unit型の式（voidに相当します）も1つの型であり、ジェネリクスに統合されています。通常は破棄され

Scalaは、制限や例外がわずかしかない、規則性を持った簡潔な表記を意図しています。定義上、冗長な部分はすべて省略可能です。

るたった1つの値（単なる結果）をたまたま持っているというだけのものです。式と文がこのような1つのものとして扱われているため、いくつかのことが簡単になっています。たとえば、関数の本体が文である（すなわち、voidを返す）か、式であるかを区別する必要はありません。

究極的には、Scalaは純粋なオブジェクト指向言語です。つまり、概念的には、Scalaではすべての値はオブジェクトであり、すべての操作はメソッド呼出しです。関数など、言語におけるその他の高レベルな「便利機能」は、内部的にメソッド呼出しに展開することによって、オブジェクト指向コアに変換されます。

この機能によって、Scalaでは簡単にドメイン固有言語（DSL）を書けるようになっていきます。そのようなDSLの利用は、Apache Spark（データ・サイエンス）、Slick（データベース問合せ）、およびsbtビルド・ツール（宣言的ビルド定義）の実装に成果をもたらしています。

ScalaがFPと深く統合されていることを示す側面の1つは、関数の適用f(x)がメソッド呼出しf.apply(x)に変換されている点です。関数はapplyメソッドを持つオブジェクトとしてモデル化されます。そのため、このメソッドを持つすべてのオブジェクトでこの簡易記法を利用できます。

同じ考え方によって、さらに高度なScala機能もオブジェクト指向コアに変換されています。for (x <- coll) println(x)など、forを使ったコードは、coll.foreach(x => println(x))の省略表現です。この展開によって、少しばかり複雑な内包表記の形に汎用化され、標準ライブラリの外で定義されたコレクション型に対して等しく適用されます。その結果、SQL問合せやビッグ・データ分析などの表現に利用できるようになります。

筆者が気に入っている変換の1つに、s"Hello \$name"などの文字列補間があります。これは、通常のメソッド呼出しStringContext("Hello ", "").s(name)という形に展開されます。この仕組みを使うと、2種類のフックによってその動作を独自の用途に適応させることができます。1つは、スコープに独自のStringContextクラスを定義して標準の補間子をオーバーライドする方法であり、もう1つは、Scalaのメカニズムを使って既存のStringContextに機能を追加する方法（遡及的拡張）です。いずれの場合も、sqlメソッドを使用した、sql"SELECT \$col FROM \$table"といった操作をサポートできるようになります。

Scalaは、制限や例外がわずかしかない、規則性を持った簡潔な表記を意図しています。定義上、冗長な部分はすべて省略可能です。推論可能な場合はメソッドのシグネチャを省略でき、メソッドの本体を省略すると



`abstract`を宣言したことになります。デフォルトの可視性は`public`で、定義は自由にネストさせることができます（メソッド内にローカル・メソッドを作成することもできます）。

ユーザー定義クラスを自然な形でプリミティブ型とともに使えるように、識別子を記号にすることや、メソッド呼出しでピリオドを省略することができるようになっています。つまり、`x + y`と`x.(+)(y)`に概念的な違いはないということです。いずれも、単に`+`というメソッドを呼び出しているだけです。プリミティブ型の値はオブジェクトと見なされますが、内部的には効率的なバイトコードにコンパイルされます。さらに記号を減らせるように、改行から推論できる場合は、式と式を区切るセミコロンも省略できるようになっています。

いくつかのサンプル

それでは、もう少し具体的な話に移ります。以降の例では、`scala>`プロンプトの後にあるものはScalaシェル（`scala`コマンドで起動します）への入力を示し、その他の行はシェルからのフィードバックやプログラムからの出力を示しています。

まずは、`message`という単一のメソッドを持つ`C`クラスを定義してみます。

```
scala>
class C { def message = "hello" }
defined class C
```

コンパイラは、`message`の結果の型を推論しています。通常、パブリック・メソッドでは、結果の型を指定するとよいでしょう。

```
scala>
```

すべてのメソッド呼出しはオブジェクトを対象にするという一貫性を維持するために、Scalaにはstaticキーワードがありません。その代わりに、オブジェクト定義によって直接的にシングルトン・デザイン・パターンをサポートできるようになっています。

```
class C { def message:String = "hello" }
defined class C
```

結果の型を省略しない場合と省略した場合の定義方法をそろえるために、メソッドの結果の型は最後に記述します。さらに一般的に言えば、型帰属は常に対象エンティティの後に記述します。

もちろん、クラスのメッセージをハードコーディングするべきではありません。一般的に、コンストラクタに渡された各引数は、後で使用するために即座にフィールドに保存されます。その場合、クラスのシグネチャで引数のリストを定義し、それぞれのメンバーの前に`val`または`var`のキーワードを付けることができます。

```
scala> class C(val message:String)
defined class C
```

```
scala> new C("Hi!").message
res1:String = Hi!
```

Eiffelプログラミング言語で統一形式アクセスの原則として提唱された統一性に従っているため、メッセージを`def`から`val`に変更しても、そのクラスのユーザーが変更を意識することはありません。透過的に`val`を遅延評価することもできます。その場合、最初にアクセスされた際に計算され、キャッシュされます。この点も汎用化されており、getterとsetterでモデル化される可変フィールドでも同様です。

次のスニペットでは、トレイトで抽象可変変数を宣言しています。getterとsetterに関する実装はサブクラスに先送りします（おそらく検証を追加するでしょう）。`???`メソッドを使って実装を省略しています。このメソッドは、標準ライブラリで`NotImplementedError`をスローするように定義されています。

```
trait T {
  var v:Int
}
```

```
class Sub extends T {
  def v:Int = ???
  def v_=(x:Int):Unit = ???
}
```



}

トレイトでは、具象型の`val`または`var`のメンバーを定義することもできます。この場合、コンパイラがトレイトのサブクラスで自動的にメンバーを実装します。

すべてのメソッド呼出しはオブジェクトを対象にするという一貫性を維持するために、Scalaには`static`キーワードがありません。その代わり、オブジェクト定義によって直接的にシングルトン・デザイン・パターンをサポートできるようになっています。同じ名前のクラスとオブジェクトを同時に定義するという手法は一般的に使われており、Javaで静的メソッドにあたるものは、クラスの「コンパニオン・オブジェクト」のメソッドになります。

次の例では、明示的に新しいインスタンスを作成するのではなく、Cクラスのシングルトン・インスタンスを定義しています。

```
scala> object o extends C("Hi!")
defined object o
```

```
scala> println(o.message)
Hi!
```

新しいメンバーが追加されるわけではないため、オブジェクトの定義は遅延評価される`val o = new C("Hi!")`に非常に似たものになります。

ここで、Scalaの統一性に関する別の側面に気付いた方もいらっしゃるかもしれません。定義は常にキーワード (`val`、`var`、`def`、`object`、`trait`、`class`、`type`のいずれか) に付随するもので、その前にいくつかの修飾子 (`lazy`、`private`など) が付く場合もあります。その後に名前とシグネチャが続き、最後に右辺が記述されます。

データの処理では、多くの場合、タプルとしていくつかの値をまとめると便利です。タプルはとても一般的であるため、ScalaはN個のフィールドを持つ一連のTupleNケース・クラスを標準で提供しています。

ケース・クラスとパターン・マッチング

Scalaが不変デザインを促しているもう1つの例として挙げられるのは、とても好都合なことに、コンストラクタの引数を不変の`val`へとただちに格納できる点です (または、`var`に格納することもできます)。これをさらに便利にするために、ケース・クラスを定義する際は`val`および`new`というキーワードを省略できるようになっています。

```
scala> case class C(x:Int)
defined class C
```

ケース・クラスは、不変構造化データをモデル化したものです。これがケース・クラスと呼ばれるのは、パターン・マッチングを行う際に、`case`キーワードと合わせて次の例のようにとても簡単に使えるためです。

```
scala> val y = C(1) match { case C(x) => x }
y:Int = 1
```

`y`の右辺の式では、新しいオブジェクト`C(1)`を作成し、即座にそれを分解して、パターン・マッチングを使ってパターン`C(x)`と比較しています。パターンは、変数にバインドされ、値を抽出するための「穴」がある値と考えることができます。この例では、パターン変数`x`が1 (`C`の最初のコンストラクタ引数に渡された値) に割り当てられ、マッチ式の結果に使われています。データ構造内で目的とする部分に直接到達できるように、パターンをネストさせることもできます。通常は、すべての考え得るケースがカバーされていることと、すべてに到達可能であることはコンパイラが保証します。

データの処理では、多くの場合、タプルとしていくつかの値をまとめると便利です。タプルはとても一般的であるため、ScalaはN個のフィールドを持つ一連のTupleNケース・クラス (Nは1~22) を標準で提供しています。そのため、タプルは通常のケース・クラスのように動作しますが、コンストラクタを呼ぶ際に名前を使う必要がないという点は異なります。たとえば、`Tuple2("a", 1)`は省略して`("a", 1)`と書くことができます。`Unit`は、唯一の値`()`を持つTuple0だと考えるとよいでしょう。

先ほどの例のような1つのケースに対するパターン・マッチは、すでに説明したように、変数`y`を`C`の最初の引数の値にバインドして`val C(y) = C(1)`と省略して書くことができます。この記法は、タプルを扱



う際に特に便利です。メソッドから返された複数の値に対して即座に名前を付けることができるからです。パターン・マッチング、とりわけタプルは、中間結果を(可変)ローカル変数に格納するコーディング・パターンから離れ、Apache SparkやApache Flinkなどをなどを使用して巨大なデータセットを効率的に計算することに役立ちます。

まとめ

Scalaを使ってみるもっとも簡単な方法は、ScalaをサポートしているIDE (IntelliJ IDEAまたはScala IDE for Eclipse) をインストールすることです。これらのScala IDEは、read-eval-printループ (REPL) と同等の対話型ワークシートを提供しています。コマンドラインを使う場合は、[Scalaディストリビューション](#)をインストールしてscalaコマンドを実行するか、[Scalaビルド・ツールsbt](#)をインストールします。

sbtで空のScalaプロジェクトを簡単に作成するためには、コマンド・プロンプトで `sbt new scala/scala-seed.g8` を実行します。新しいプロジェクト用に作成されたディレクトリからsbtを起動し、コンソール・タスクを使って対話型シェルを起動すると、本記事の例を試すことができます。

Scalaを使ってみるもっとも簡単な方法は、ScalaをサポートしているIDE (IntelliJ IDEAまたはScala IDE for Eclipse) をインストールすることです。

Learn More

さらに詳しく学びたい方には、Scalaに関するすばらしい書籍 (『Scala for the Impatient』、『Programming Scala』、『Programming in Scala』など) やCourseraの無料コース (最初のコースは「[Functional Programming Principles in Scala](#)」)、その他多くのオンライン・リソース ([TwitterのScala School](#)など) をお勧めします。お楽しみください。 [</article>](#)

Adriaan Moors (@adriaanm) :LightbendのScalaチーム・リーダー。2007年、Scalaコンパイラへの初めての貢献として型コンストラクタのポリモフィズムをサポートし、それをテーマにした博士論文を執筆。EPFLのMartin Odersky研究室で博士研究員を務めた後、2012年にTypesafe (現Lightbend) に入社した。その後は、基礎理論 (依存オブジェクト型計算) からコンパイラ実装、継続的インテグレーション、リリース・インフラストラクチャに至るまで、Scala言語のあらゆる側面に携わっている。



DockerコンテナのJavaアプリをOracle Application Container Cloud Serviceにデプロイする

HARSHAD
OAK

ここ数年、アプリケーション・コンテナが大きな注目を集めています。アプリケーションのコンテナ化は、仮想マシンよりもはるかに少ないリソースしか消費しない独立した環境で「Build once, run anywhere」（一度ビルドすればどこでも実行できる）を確約するものです。

コンテナ・ソリューションが大きな注目を集め、採用が拡大し続けていることを考慮し、多くのベンダーがこのソリューションを提供する方法を模索しています。Oracle Application Container Cloud Serviceは、Dockerを活用してJava、PHP、Node.jsアプリケーションを実行できるサービスを提供しています。本記事では、Oracle Application Container Cloud ServiceのJavaおよびJava EE機能について説明します。本記事の内容を理解するためには、Maven、Java EE、コンテナ化についての基礎的な知識が必要になります。

Oracle Application Container Cloud Serviceについて見てゆく前に、実績あるJava EEがデプロイされたサーバーからコンテナに切り替える際に考慮する必要がある点について簡単に触れておきます。

まず、現在のJava EEの使いやすさや、最新のJava EEツールおよびサーバーのスマートさ、自己完結性、追加設定不要といった側面を考慮することが重要です。

コンテナ化されたアプリケーションが軽量なのは、必要最低限のものしか搭載していないためです。つまり、どのリソース、依存性、バージョンを含めるかを正確に知っていなければなりません。この把握には、ある程度の慣れが必要な場合があります。アプリケーションが使うかもしれないすべてのものを提供してくれるアプリケーション・サーバーを常用しているJava EE開発者は、特に慣れが必要です。現在のほとんどのJava EEアプリケーションは、アプリケーションのコードさえ提供すればよいものになっています。他のほぼすべてのことは、サーバーがアプリケーションの規約や構成から判断してくれます。

コンテナ化されたアプリケーションは、本質的にパブリック・クラウドへのデプロイに適したものと思われます。しかし、従来型のJava EEデプロイが必要になる場合でも、クラウドで適切にJava EEを実行する多くのPlatform as a Service (PaaS) クラウド・ソリューションがあります。多くの場合は、基盤となるハードウェア、リソース、サーバーについて心配する必要はありません。こういったデプロイについては、以前のJava Magazineの記事で取り上げました。最近では、Oracle Java Cloud ServiceでJava EEアプリケーションを使う方法について、「[Oracle Java Cloud Serviceを使い始める](#)」という記事で紹介しました。

また、実績あるJava EEサーバーベースのアプローチには、多くの専門知識が蓄積され、ツールも用意されています。一方のコンテナ市場は、まだ流動的な状況です。

採用するコンテナ・ソリューションの種類を決定できるのは、コンテナでアプリケーションを動かせるという確実な見通しが立ってからになります。それでは、Oracleのソリューションを詳しく見てみます。

Oracle Application Container Cloud Serviceは、Oracle Cloud内の専用のDockerコンテナでアプリケーションを実行するシンプルなソリューションで、現在のところ、Java、JVM言語、PHP、Node.jsがサポートされています。また、Oracle Cloudの独立したコンテナ環境内では、高速で使いやすい、アプリケーションのセルフサービス・プロビジョニングを行えることが保証されています。

このサービスは、Oracle Database Cloud ServiceやOracle Java Cloud Serviceなど、オラクルが提供する他のクラウド・サービスとも統合されています。さらに、Oracle Developer Cloud Serviceを併せて利用すれば、継続的インテグレーションやそれを活用した開発を行うこともできます。

Oracle Application Container Cloud Serviceのアプリケーションは、Oracle LinuxベースのDockerコンテナ内で実行されますが、サービスはDockerコンテナを実行することではなく、アプリケーションを実行することを意図したものになっています。

独自にDockerコンテナを実行する場合と比べてOracle Application Container Cloud Serviceが優れている主な点としては、Javaアプリケーション用のサービスとしてFlight Recorder診断やプロファイリング機能を含むOracle Java SE Advancedが持つ機能がすでに設定されていることが挙げられます。Javaのパッチ適用やアップグレードは、オラクルが行います。また、ロードバランシングやアプリケーションのスケールリングもサービス側で行われます。そのため、このサービスによって高度な抽象化を実現でき、基盤となるJavaの設定やOracle Linuxベース、Dockerコンテナのことを気にする必要はなくなります。

Oracle Application Container Cloud Serviceへの デプロイ方法

Oracle Application Container Cloud Serviceを使う際も、今までのアプリケーションのコードを変える必要はありません。アプリケーションのパッケージ化やデプロイ、管理の方法がわかれば、サービスの大半を理解したことになります。

本記事では、詳しいコード・スニペットを見てゆく代わりに、サービスのドキュメントに掲載されているサンプル・アプリケーションの1つを取り上げ、それを微調整してパッケージ化し、Oracle Application Container Cloud Serviceで実行する方法を紹介します。わかりやすく説明するために、アプリケーションの作成にはNetBeansを使用します。

まずは、チュートリアル[のコードをダウンロード](#)します。今回使うのは、「Java SE 8: Creating a Web App with Bootstrap and Tomcat Embedded for Oracle Application Container Cloud Service」です。このチュートリアルで取り上げているのは、サーブレット、JavaServer Pages (JSP)、Bootstrapフロントエンド・フレームワーク、組み込み版のTomcatを使用したシンプルなスタンドアロンWebアプリケーションです。

アプリケーション・サーバーへのWARファイルやEARファイルのデプロイに慣れているJava EE開発者の方は、なぜ組込みのTomcatサーバーが必要なのかを疑問に思うかもしれません。ここで説明しておく必要があるのは、Oracle Application Container Cloud Service環境ではJavaが

動作していますが、Java EEアプリケーションを実行するサーバーは存在しないことです。つまり、開発者がアプリケーションの一部としてサーバーをパッケージ化する必要があります。

ダウンロードしたコードを解凍すると、Mavenプロジェクトや必要なプロジェクト・ファイルが作成されます。ここでは、プロジェクトとその構成要素、特にOracle Application Container Cloud Serviceへのデプロイ用にプロジェクトがどのように設定されているかがわかりやすいように、NetBeansを使っています。MavenプロジェクトはほとんどのIDEでサポートされているため、別のIDEを使うこともできます。または、コマンド・プロンプトからMavenコマンドを使うことも可能です。

NetBeansとMavenの使用

NetBeans IDEを開き、メニューから「File」→「Open Project」を選択します。そして、コードを解凍したディレクトリを選択します。NetBeansはそれがMavenプロジェクトであることを検出し、適切に処理します。

図1に示すように、WebページといくつかのJavaクラス、依存性、Mavenのpom.xmlファイルがあることがわかります。プロジェクト・オブジェクト・モデルには、プロジェクト情報とMavenがプロジェクトをビルドする際に使用する設定が含まれています。NetBeansでpom.xmlファイルを開くと、ファイルがグラフ・ビューで表示されるため、プロジェクトの依存性を容易に確認できます。

今回は、ソース・ビューを開いてXMLを見てみます。まずは、上から順番にプロジェクトの重要なタグのいくつかを見てゆきます。

リスト1に示すように、Tomcatのバージョンは7.0.57に指定されています。Tomcatは、サポ

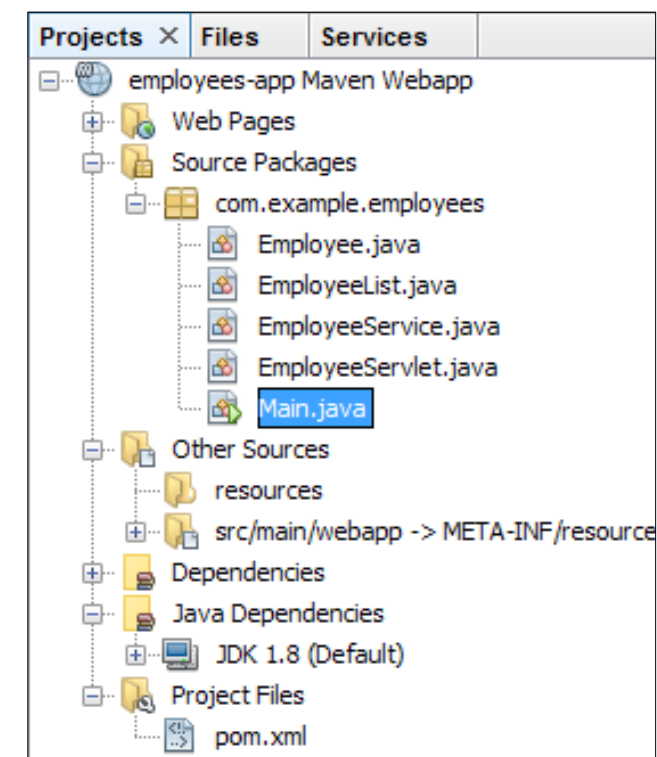


図1: NetBeansのプロジェクト・ビュー

ブリ・プラグインのタグが定義されています。

■ リスト3:

```
<configuration>
  <descriptorRefs>
    <descriptorRef>
      jar-with-dependencies
    </descriptorRef>
  </descriptorRefs>
  <finalName>
    ${project.build.finalName}-${project.version}
  </finalName>
  <archive>
    <manifest>
      <mainClass>
        com.example.employees.Main
      </mainClass>
    </manifest>
  </archive>
</configuration>
```

ここで、`mainClass`として`com.example.employees`
`.Main`が指定されています。この宣言は重要です。パッケージ化された
JARファイルを`java -jar`コマンドで実行すると、このクラスが実行される
ためです。それでは、この`mainClass`を見てみます。リスト4をご覧ください。

■ リスト4:

```
public class Main {
    public static final Optional<String> PORT =
        Optional.ofNullable(System.getenv("PORT"));
    public static final Optional<String> HOSTNAME =
        Optional.ofNullable(System.getenv("HOSTNAME"));

    public static void main(String[] args)
        throws Exception {
```

さらにファイルの続きを見てみると、リスト3に示すように、Mavenアセン

```

String contextPath = "/";
String appBase = ".";
Tomcat tomcat = new Tomcat();
tomcat.setPort(Integer.valueOf(
    PORT.orElse("8080") ));
tomcat.setHostname(
    HOSTNAME.orElse("localhost"));
tomcat.getHost().setAppBase(appBase);
tomcat.addWebapp(contextPath, appBase);
tomcat.start();
tomcat.getServer().await();
}
}

```

このコードでは、環境からポートとホスト名を取得し、Tomcatサーバーを起動しています。

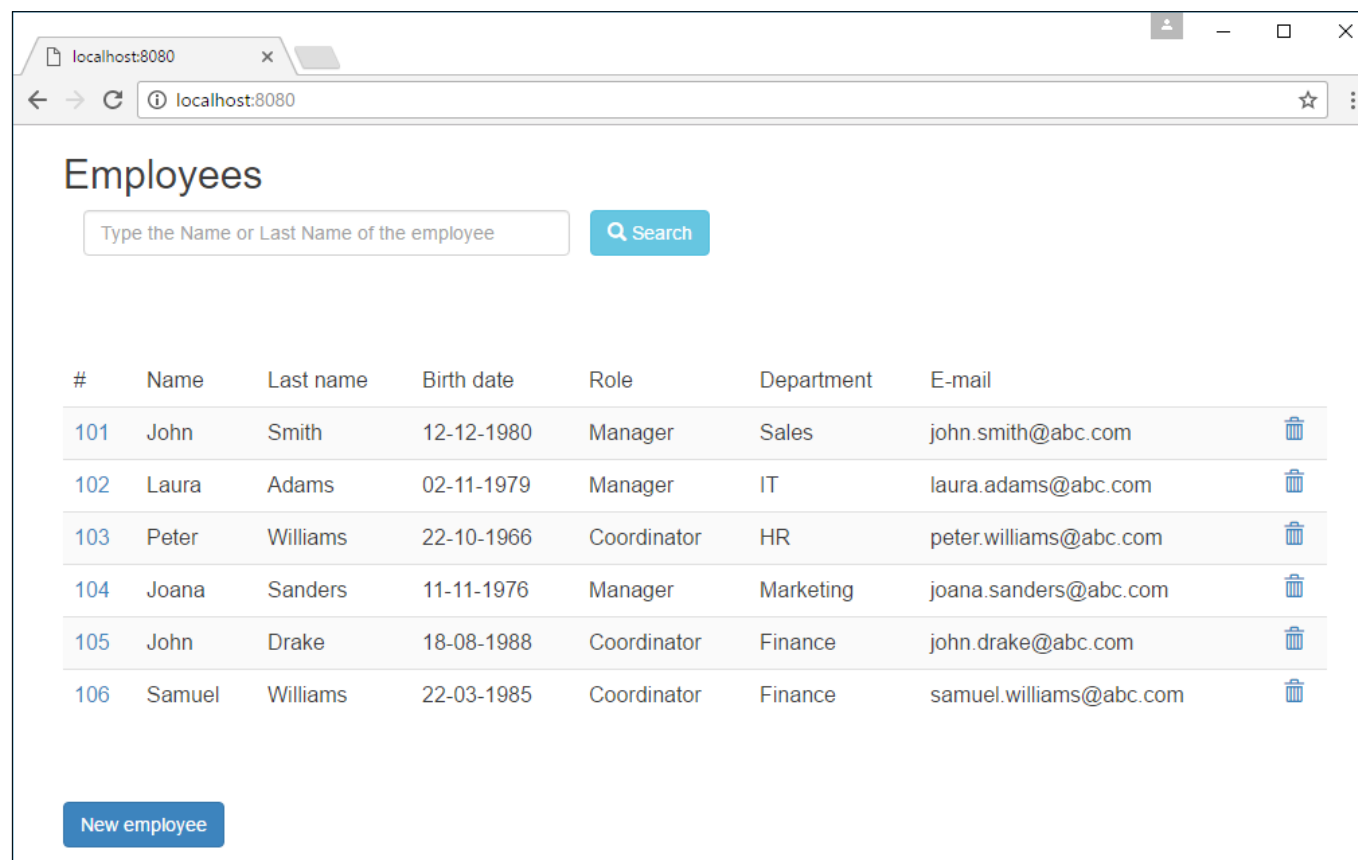


図2: Webアプリケーションの実行

プロジェクトのその他のコードは、実際のWebアプリケーションを構築するためのものであるため、Oracle Application Container Cloudでアプリケーションを実行する際に直接重要になるものではありません。

アプリケーションのビルド

Oracle Application Container Cloud Serviceにデプロイする前に、アプリケーションをビルドする必要があります。ビルドを行うには、プロジェクト名を右クリックして「Build」を選択します。Mavenが必要なJARファイルを取得してコピーし、アプリケーションのコンパイルとパッケージ化を行い、ターゲット・ディレクトリに実行可能JARファイルが生成されます。この様子はログから確認できます。このJARファイルには、コードだけでなく依存ライブラリも含まれています。そのため、このようなJARファイルは、ファットJARまたはウーバーJARと呼ばれています。生成されたJARファイルには、manifest.mfファイルも含まれており、その中では、JARファイルを実行する際のメイン・クラスとして`com.example.employees.Main`が指定されています。

`java -jar employees-app-1.0-SNAPSHOT-jar-with-dependencies.jar` コマンドを使ってこのJARファイルをローカル・マシンで実行すれば、Tomcatサーバーがローカル・マシン上で起動し、そこでアプリケーションが実行されます。ブラウザから`http://localhost:8080`を開くと、図2に示すページが表示されます。

しかし、今回の目的はアプリケーションをOracle Application Container Cloud Serviceで実行することであるため、もう少し作業を続ける必要があります。

manifest.jsonの設定

アプリケーションをクラウド・サービスにアップロードしてデプロイするためには、manifest.jsonファイルをアプリケーション・アーカイブに含める必要があります。アーカイブは、.zip、.tgz、.tar、.gzのいずれかのファイルです。manifest.jsonファイルでは、使用するJavaのバージョンと、アプリケーションを起動するために実行するファイルを指定します。マニフェストには、任意で他の設定値を含めることもできます。manifest.jsonファイルをリスト5に示します。



■ リスト5:

```
{
  "runtime": {
    "majorVersion": "8"
  },
  "command":
    "java -jar employees-app-1.0-
    SNAPSHOT-jar-with-dependencies.jar"
}
```

[編集注:スペースの関係上、最後の行は折り返しています。] 次に、manifest.jsonファイルとemployees-app-1.0-SNAPSHOT-jar-with-dependencies.jarファイルをzipファイルにパッケージ化します。これで、Oracle Application Container Cloud Serviceにアプリケーションをデプロイする準備は完了です。

デプロイメント

Oracle Application Container Cloud Serviceにログインすると、ダッシュボード画面が表示されます。右側にある「Create Application」ボタンをクリックし、アプリケーション・プラットフォームを尋ねられたら、「Java SE」をクリックします。すると、図3の画面が表示されます。

この画面はかなりシンプルですが、ここにはOracle Application Container Cloud Serviceが提供する重要な構成オプションのほとんどが表示されています。この画面では、サブスクリプション・モデルを選択でき、アプリケーション・アーカイブのアップロードまたはOracle Storage Cloudへのリンクの指定が可能です。さらに、アプリケーションのインスタンス数や各インスタンスに割り当てるメモリの量も選択できます。

アプリケーションに名前を付け、先ほど作成した、JARファイルとmanifest.jsonファイルを含むzipファイルをアップロードします。

「Instances」では、デフォルトの設定を使います。アプリケーションには、1GBのメモリを割り当てた単一のインスタンスを指定します。

「Processing Archive」というメッセージの後、画面には図4に示すアプリケーション・ページが表示されます。

アプリケーションの作成には、数分かかる場合があります。現在の状況は、図4の「Activity」セクションまたはメインのクラウド・サービス・ダッシュボードで確認できます。

リソースの変更

図4には、インスタンス数と割り当てられているメモリが表示されています。コンテナ化の主な特徴の1つは、必要に応じて、同じコンテナを簡単に新しく追加できることです。インスタンス数を1から16の間の任意の数に変更すると、数分で同じコンテナを実行できます。この処理が行われている間も、既存のコンテナは中断することなく実行され続けます。

メモリは1GBから20GBの間で設定できます。ただし、メモリ割当てを変更した場合、すべてのコンテナが再起動される点に注意してください。メモリのスケール・アップにはすべてのコンテナの再起動が必要で、

図3: アプリケーションの作成



The screenshot displays the Oracle Cloud Applications console interface. At the top, the application icon (a green circle with 'SE' and a flame) is shown next to the text 'Applications / EmployeeApp' and the URL 'https://EmployeeApp-harshad.apaas.us2.oraclecloud.com'. The main content area is divided into several sections:

- Overview:** Shows '1 Instances'.
- Resources:** A light blue box containing three metrics: '1 Instances', '1 Memory (GB)', and '25.8% Average Memory Usage'.
- Instances:** A section with a 'Health Check' button. It shows a single instance named 'web.1' with 'Memory: 1GB'. Below this, a 'Performance Metrics' box indicates 'Memory used(%): 25.78%' as of Nov 25, 2016 5:13:49 PM UTC.
- More Information:** A section with two columns of metadata:

Last Deployed On: Nov 25, 2016 5:12:54 PM UTC	Created On: Nov 25, 2016 5:12:54 PM UTC
Current Version: 1.0	Identity Domain: harshad
Runtime: Java SE 8u102	Subscription Type: MONTHLY
Runtime Version: 1.8.0_102-b31	Notes: Employee Web App with Bootstrap ...
- Activity:** A section titled 'Activity Summary' showing a log entry:

Application Created Name: EmployeeApp Operation: Create Status: Succeeded	Start Time: Nov 25, 2016 5:12:55 PM UTC End Time: Nov 25, 2016 5:14:12 PM UTC
---	--

On the left side, there is a navigation menu with sections: Overview, Deployments (0 Service Bindings, 0 User-defined Variables), and Administration (0 Updates Available, 0 Logs, 0 Recordings).


図4: アプリケーションの概要ページ



停止時間が発生する可能性があるため、Oracle Application Container Cloud Serviceではローリング・リスタート・オプションが提供されています。この機能を使うと、コンテナのインスタンスを1つずつ停止して再起動することができます。ローリング・リスタートは時間がかかりますが、アプリケーションの停止時間をなくすることができるため、便利な機能です。

更新、ログ、記録

図4と図5の左側に表示されている「Administration」セクションをクリックすると、Javaランタイムのバージョンの詳細がわかる「Updates」タブや、各コンテナのログが格納された「Logs」タブが表示されます。さらに、Java Flight Recorderによる診断やプロファイリング用の記録を開始できる「Recordings」タブもあります。


Applications / EmployeeApp

URL: <https://EmployeeApp-harshad.apaas.us2.oraclecloud.com>

Overview

3
Instances

Deployments

0
Service Bindings

0
User-defined Variables

Administration

0
Updates Available


1
Logs

0
Recordings


Updates
 Logs
 Recordings


As of Nov 25, 2016 5:58:07 PM UTC

You can download logs for your application instances from your Oracle Storage Cloud Service account.


 Instance: web.1
 [Get Log](#)


1 Logs

Name	Size	Last Uploaded
 server.out.zip	674 B	Nov 25, 2016 5:37:55 PM UTC


 Instance: web.2
 [Get Log](#)

No Logs

Name	Size	Last Uploaded
------	------	---------------


 Instance: web.3
 [Get Log](#)

No Logs

Name	Size	Last Uploaded
------	------	---------------

Logs Capture History

図5: Administrationオプション (Logsタブを表示)

サービスのバインド

図5の左側にある「Deployments」セクションをクリックすると、「Service Bindings」セクションが表示されます。Oracle Java Cloud ServiceまたはOracle Database Cloud Serviceで既存のサービスを使っている場合、ここからアプリケーションをサービスにバインドできます。このバインドは、deployment.jsonファイルで指定することも可能です。

以上の結果、短時間でさほど労力をかけることなく、Oracle Cloudの独立した複数の同じコンテナで完全なWebアプリケーションが実行されます。外からは見えないものの、裏ではOracle Application Container Cloud Serviceが提供するロードバランサも動作しており、トラフィックをアプリケーション・コンテナにルーティングしています。

本記事では、既存のアプリケーションを使ってOracle Application Container Cloud Serviceを試してみました。しかし、新しいMavenプロジェクトを作成することや、既存のプロジェクトを更新してOracle Application Container Cloud Serviceでアプリケーションを実行することもできます。Oracle Application Container Cloud ServiceでWebアプリケーションを実行する際に必要となる重要なポイントは、(1) 依存性のパッケージ化、(2) 組込みサーバーのパッケージ化、(3) パッケージ化したアプリケーションとサーバーの実行方法の指定です。

Oracle Cloudには複数のクラウド・サービスが統合されており、それらのサービスが協調してソリューションを提供しています。Oracle Application Container Cloud Serviceに関連する部分では、Oracle Java Cloud ServiceやOracle Database Cloud Serviceへのバインドの他に、[前号](#)で取り上げたOracle Developer Cloud Serviceについても確認してみるとよいでしょう。Oracle Developer Cloud Serviceは、さまざまな「Developer Environment as a Service」（サービスとしての開発環境）機能の他に、アプリケーションのビルドやOracle Application Container Cloud Serviceへのデプロイを簡略化および自動化できる機能も提供しています。

短時間でさほど労力をかけることなく、Oracle Cloudの独立した複数の同じコンテナでWebアプリケーションが実行されます。

まとめ

多くの開発者や企業が、一貫性のある独立した環境やリソースの効率性を保証するコンテナに興味を示しています。Oracle Application Container Cloud Serviceは、高速で利用や管理が簡単な方式のコンテナ化ソリューションを提供します。このサービスは、[Oracle Application Container Cloud Service Webページ](#)から、無償で試用できます。

</article>

Harshad Oak: Java Champion、Oracle Ace Director。IndicThreadsとRighthrix Solutionsの設立者。『Pro Jakarta Commons』（Apress、2004年）や、Java EEに関連する何冊かの書籍を執筆しており、さまざまな国のカンファレンスでも講演を行っている。

learn more

[Dockerコンテナ](#)
[Oracle Developer Cloud Service](#)





BRIAN GOETZ

JEP 286

ローカル変数の型推論: varの導入とvalの可能性

[編集注: 本記事は、オラクルのJava言語アーキテクト、Brian Goetzの公開している文書を基に、情報や意見を加筆したものです。言語設計者は、見かけ上小さな機能追加であっても慎重に検討を行っていることがわかります。ここでは、変数宣言に必要な定型挿入句を減らすために、**var**、**val**、**let**といったキーワードの追加が検討されています。今回取り上げる提案の構文については、[Java Magazine](#)の2016年3月/4月号でも触れています。]

JEP 286に関する2段階の調査には、2,500名を超える人々が参加しました。ローカル変数に型推論を追加することに対する全体的な反応は非常に好意的で、74%が強く賛成、12%がどちらかと言えば賛成、10%が反対という結果でした。

寄せられたコメントで見れば反対の比率がもう少し高くなりますが、これは想定範囲内でしょう。一般的に、賛成する人よりも反対する人の方が多くを語るからです。肯定的なコメントは非常に肯定的なものでした。また、否定的なコメントは非常に否定的なものでした。中には痛烈な反対意見もあったものの、数が雄弁に物語っているように、ほとんどの開発者がこの機能を望んでいます（他の言語からJavaに移行した開発者ももっとも多くリクエストしているのがこの機能です）。結局、どのような選択をすることになっても、非常に落胆する人々がいるということになります。

選択肢の中で、一番多く選ばれた構文は「valとvar」であり、次に「varのみ」が選ばれています。しかし、これらの選択肢についてどう考えるかを尋ねると、反応は一樣ではありませんでした。「valとvar」が「好きだ」と回答した人数も、「嫌い」と回答した人数も、多く見られました。人は自分が気に入っている構文にこだわりを示すものです。しかし、覚えておくべき重要な点は、構文は表面上のものでしかないということです。一見単純に見えても、このような言語機能は内部的には非常に複雑なのです。

写真:
BOB ADLER/VERBATIM

可読性

もっとも多かった類の反対意見は、可読性が低下するのではないかという懸念に関するものでした（ただし、この意見のほとんどは、ローカル変数の型推論を使ったことがない人々からのものでした。別の言語でこの機能を使ってきた人々からの反応では、肯定的な意見が圧倒的でした）。Java言語の根幹をなす揺るぎない設計原則となっているのは、コードの書きやすさよりも読みやすさを重視するという点です。しかし、多くの人々は、ローカル変数の型推論を導入すれば、コードの可読性低下は避けられないと考えていました。簡単な例を考えてみます。

```
var x = y.getFoo()
```

これは、ローカル変数の型推論によって可読性が低下する好例と言えるかもしれません。しかし、よく見てみれば、ここでの可読性の問題は、**x**という変数の名前の選び方がよくないことによるものであることがわかります（マニフェスト型を付ければプログラマーの怠惰の埋め合わせにはなるかもしれませんが、本来であれば、適切な変数名を選ぶ方が望ましいと言えるでしょう）。

どの表記規則にも言えることですが、**var**も不適切に使えば、甚大な被害を生む可能性があります。しかし、Javaチームは、これを適切に使えば、実際には可読性の向上につながる可能性があると考えています。変数名がコードの中の目に付きやすい位置に移動することがその理由です。次のローカル変数のブロックについて考えてみます。

```
UserModelHandle userDB = broker.findUserDB();  
List<User> users = db getUsers();  
Map<User, Address> addressesByUser =  
    db.getAddresses();
```

多くの場合、これらの行でもっとも重要になるのが変数名です。変数名は



現在のプログラムでの変数の役割を示すものだからです。しかし、前述の例では、変数名はコード中の視覚的にわかりにくい場所にあります。各行の中央にあるうえ、各行の中での位置も一定していません。

型推論を使うことで、変数名が目立つようになります。型推論を使って前述のブロックを書き直した場合、次のようになります。

```
var userDB = broker.findUserDB();
var users = db.getUsers();
var addressesByUser = db.getAddresses();
```

これによって、コードの真の意図がはるかにわかりやすくなります。変数名がほぼ先頭の目に付きやすい位置に来ているためです。適切な変数名が選ばれているため、実体型が付いていなくても問題にはなりません。

可変性 (Mutability)

寄せられたコメントの多くは、型推論の使用に関することではなく、可変性に関することでした。finalを付けるという冗長な表記を減らしたいと考えている人が多いということです。私たちもそう考えています。

当初、ローカル変数の型推論は、実質的にfinalであるローカル変数のみを対象とすることが検討されていました。しかし、プロトタイプに着手すると、この考え方は先ほど述べた「変数名を目立たせる」という要請と相いれないことがすぐにわかりました。可変ローカル変数が不格好に飛び出して見えるようになるからです。

```
var immutableLocal = ...
var anotherImmutableLocal = ...
var alsoImmutable = ...
LocalDateTime latestDateSeenSoFar = ...
var thisOneDoesntChangeEither = ...
```

この不規則な見た目は、コードを読む者には不快に映りました。Scalaと同じように、final varという意味でvalを使いたいという方は多いでしょう。この2つはとてもよく似ているため、読む者がその違いを無視しようと思えばできると私たちは考えていました。しかし、ユーザビリティの実験から、先ほどの例のような大きなローカル変数のブロックでは、varとvalは似すぎていて混

乱すると感じる人がいることがわかりました (varとvalよりも違いが明確であり、Swiftで使用されているvarとletでも、混乱すると感じる人もいました)。

Javaチームは、不変性は重要だと考えています。しかし、現実問題として、不変性の実現に向けた仕組みが必要な場所の中で、ローカル変数はもっとも重要性が低い部分であるという結論に至りました。可変性によって生じる最大のリスクはデータ競合ですが、ローカル変数ではデータ競合は起こりません。さらに、大半のローカル変数はいずれにしても実質的にfinalです。不変性をサポートする仕組みがいつそう必要とされているのは、フィールドです。しかし、フィールドに型推論を適用するというのはおかしい話でしょう。

さらに、var/valの区別によってJavaで見込まれるメリットよりも、他の言語にもたらされているメリットの方が大きなものとなっています。たとえばScalaでは、ローカル変数もフィールド変数も一様に、すべての変数をval name : typeという形で宣言します。型推論を使う場合は、: typeを省略できます。そのため、可変性と型推論が分離されているだけでなく、var/valの区別は、さまざまな状況で利用される強力なものになっています。一方のJavaでは、型推論を必要とするローカル変数のみが対象になる予定です。すなわち、var/valはきれいに考え方を引き継いでいるように見えますが、実際はその過程で大きく意味が変わっているのです。

構文の選択

ここまで、長々と賛否両論について見てきましたが、Javaチームにとって結論は明確でした。それは、varのみという案です。その理由の一部を以下に示します。

- 調査で一番多く選ばれた選択肢ではありませんが、明らかに大半の人々が納得できるものでした。「varとval」を好まない人は多く、「varとlet」を好まない人もいました。しかし、「varのみ」を好まない人はほとんどいませんでした。
- C#で「varのみ」が採用されていることから、Javaのような言語ではこの案が妥当であることがわかります。C#では、valを望む大きな声は上がっていません。

不変性に伴う
冗長なコードを
減らしたいとい
う声は妥当であ
るものの、この
場合は見当違い
です。

- 不変性に伴う冗長なコードを減らしたいという声は妥当であるものの、この場合は見当違いです。Java開発者に必要なのは、フィールドにおける不変性であり、ローカル変数における不変性ではありません。ところが、`var`と`val`はフィールドには適用されず、今後適用される可能性もほとんどありません。
- 可変性修飾子よりも変数名が重要だと考えれば、変数名は可変性を示す修飾子よりも重要だということになります。

おわかりのように、わざわざJavaチームにご意見をお寄せいただいた多くの開発者の皆さんによるご支援がなければ、このような決定はできませんでした。 [</article>](#)

Brian Goetz (@BrianGoetz) : オラクルのJava言語アーキテクト。『Java Concurrency in Practice』(Addison-Wesley Professional、2006年)で筆頭著者を務め、Javaカンファレンスでも頻繁に講演を行っている。

ORACLE®



Java Is Just the Beginning

Your Java applications need high-performance and battle-tested platform and infrastructure services to build, test, deploy, and monitor. Oracle Cloud delivers.

Start with Java in the cloud—or choose whatever language, database, compute service, and OS option you need. Rapid scalability. True portability. It's all here. Now.

Oracle Cloud.
Built for modern app dev.
Built for you.

Start here:
developer.oracle.com

#developersrule



クイズに挑戦

新

年のよいスタートを切るために、[1Z0-809 Programmer II試験](#)の練習問題をさらに出題します。なお、この試験は、Java 8のプログラミングの基本的知識を有することが認められ、さらに高度な専門知識を有することを証明しようとしている開発者向けの認定資格試験です。

設問2. 次のコードについて:

さらに次のコードについて:

設問1.次のコードについて:

正しいものはどれですか。2つ選んでください。

- どのような結果が出力されますか。1つ選んでください。

- 



//fix this /

- Alaïs
- c. Sonia
Mélina
Alaïs
- d. Sonia
Jaquelina
Alaïs
- e. 出力の順序はプラットフォームに依存する

設問3. 次のコードがあるとします。

```
// line n1
Connection conn = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/sample;" +
    "user=app;password=app");
// line n2
Statement statement = conn.createStatement();
// line n3
ResultSet rs = statement.executeQuery(
    "SELECT * FROM APP.CUSTOMER");
// line n4
while (rs.next()) {
    String name = rs.getString("NAME");
    System.out.printf("%s\n", name);
}
// line n5
```

すべてのデータベース・リソースを解放するのはどれですか。 1つ選んでください。

- a. 以下を挿入:
line n1に `try (`
line n3に `) {`
line n5に `}`
- b. 以下を挿入:
line n2に `try (`
line n4に `) {`
line n5に `}`

- c. 以下を挿入:
line n3に `try (`
line n4に `) {`
line n5に `}`
- d. line n5に挿入: `rs.close(); statement.close();`



設問1. 正解は選択肢CとEです。この問題は、インナー・クラスについての理解を問うものです。インナー・クラスという用語は極めてあいまいな意味で使われることが多いのですが、実際のドキュメントではかなり具体的な意味を持っています。インナー・クラスとは、別のクラスの中で宣言された、静的でないクラスです。クラスが静的である場合は、インナー・クラスではなくネスト・クラスと見なされます。インナー・クラスには、特別なルールがあります。Java言語仕様のセクション8.1.3「インナー・クラスとエンクロージング・インスタンス」には、「インナー・クラスが明示的または暗黙的に静的なメンバーを宣言している場合、メンバーが定数変数である場合を除き、コンパイル時エラーとする」と書かれています。ここで、定数変数とは何だろうと思う方もいるかもしれませんが。仕様のセクション4.12.4を見ると、定数変数とは「定数式で初期化されるプリミティブ型またはString型のfinal変数」と書かれています。興味深いことに、定数式にnull値を割り当ててもコンパイル・エラーとなります。

以上の規則から考えると、設問のコードは動作しません。インナー・クラスで定数変数ではない静的変数を宣言しようとしているためです。そのため、選択肢Aは誤りです。さらに、ドキュメントにはアクセス修飾子によってこの規則が変わるとは書かれていないため、選択肢BとDも誤りです。

この時点で、残った2つの選択肢CとEが正解であることがわかります。しかし、正しくないものを除外するだけでなく（シャーロック・ホームズはそれで十分かもしれませんが）、なぜこの2つが正しいのかを説明しておく必要があるでしょう。



まず、選択肢Cについて考えてみます。宣言をfinalにすれば、定数変数になります。定数変数の定義には、変数がfinalであり、かつ定数式で初期化されるプリミティブまたはStringが必要なことを思い出してください。念のため、この式が定数式であることを確認しておきます。仕様のセクション15.28「定数式」には、定数式の基準としてかなり長いリストが記載されていますが、その最初の項目にStringリテラルが挙げられています。そのため、選択肢Cは正解です。

選択肢Eで述べられているのは、実のところ、フィールドを静的フィールドからインスタンス・フィールドに戻すということです。インナー・クラスの通常のインスタンス・フィールドには何の制限もないため、選択肢Eも正解です。

ここでもう1点考慮すべきことがあり、そのことこそが、このような規則が存在する理由になっています。それほど根拠はないと思われるかもしれませんが、この規則が妥当だと納得できる説明はできます。この点を頭に入れておけば、実際には論理的に導き出された規則ではなくても、覚えやすくなります。

インナー・クラス、すなわち静的でないインナー・クラスでは、そのクラス自体がエンクロージング・インスタンス（インナー・クラス（内部クラス）を囲む型のインスタンス）のメンバーであると考えられます。つまり、外部クラスのインスタンスの生成にはフィールドやメソッドが生成（複製）されるように、内部クラスの生成には外部クラスのインスタンスが必要です。この場合、インナー・クラスは他のすべてのインナー・クラスと同じソース・コードに基づいています。各インナー・クラスはそれぞれ独自の静的メンバーを持つこととなりますが、それではこれらの静的メンバーをどのように参照すればよいのでしょうか。そのような静的なフィールドを使う場合は、どのクラスを参照するかを指定することが必要になるでしょう。このような参照指定は、複雑で美しくなく、エラーも起こりやすく、歴史的に見ても不必要なものでしょう（この制限は、Java 1.1でインナー・クラスが生まれて以来ずっと継続されています）。

この考え方は、言語設計者が意図したものとは異なる可能性があることを繰り返しておきたいと思いますが、言語設計者がこの機能をインナー・オブジェクトではなくインナー・クラスと呼んだのは確かなことです。いずれにせよ、インナー・クラスで静的メンバーは許可されていません。そのことを記憶する際に、この考え方が役立てば幸いです。一方、ネスト・クラスにはこのような制限はありません。

設問2.正解は選択肢Cです。この設問は、Java APIのセットの動作について問うものです。セットのもっとも根本的な動作として、重複したエントリを許可しない点が挙げられます。SetインタフェースのAPIドキュメントでは、この要件がequalsメソッドを使って説明されています。具体的には、nullでない2つのエントリをequalsメソッドで比較してもtrueを返すことはなく、nullアイテムも最大で1つしか格納できません。そのため、新しいアイテムを実際に追加する前に、そのアイテムがすでにセットの中に存在しているかどうかを確認する必要があります。その結果として単純にすべてのアイテムをチェックした場合、セット内のアイテム数が多くなると特に、とても時間がかかることとなります。そこで、Javaの具体的な実装では、このチェック速度を上げようとしています。

TreeSetでは、アイテムが順序正しくセットに追加されます。そのため、セットは二分探索を使ってアイテムを探すことができます（または、アイテムが存在しないと判定できます）。セット内のアイテム数が多くなると特に、二分探索は極めて効率的です。しかし、設問の例では、順序はorderというintフィールドのみによって決まります。つまり、orderが9である2つのオブジェクトは、明らかに異なる値を持っているにもかかわらず、順序付けでは同じ場所に配置されることとなります。Javaドキュメントでは、この例のように異なる（equalsがtrueにならない）オブジェクトが同じ順番になることを「equalsと一貫性のない順序付け」と呼んでいます。TreeSetのドキュメントには、「Setインタフェースを正しく実装するためには...セットによって維持される順序付けがequalsと一貫性のあるものでなければならない」と書かれています。

したがってTreeSetでは、orderフィールドの値が9である2つのオブジェクトは等しいものとして扱われます（実際は、TreeSetの実装でセット内のアイテムを考慮する際にequalsメソッドが使われることはありません）。その結果、orderが9である2つ目のアイテムは重複と見なされます。この例では、equalsメソッドがオーバーライドされていません。これを厳密に解釈するなら、すべてのオブジェクトは一意であると見なされ、同じフィールド値を使って作成されたオブジェクトであっても、同時にセットの

順序付けするために、equalsと一貫性のない順序付けを行うことができるコンパレータを使う場合は**注意が必要です**。



メンバーになることができるはずであるという意味になります。

結論としては、この例にある名前がMélinaとJaquelinaである2つのアイテムは重複と見なされます。そのため、この2つを同時にセットに格納することはできません。したがって、選択肢AとBは誤りです。

次に考慮するのは、重複するアイテムが提示されたときに、Setが厳密にどのような動作をするかという点です。すなわち、新しいアイテムを拒否するのか、既存のアイテムを置き換えるかということです。この情報から、選択肢CまたはDがセットの内容を正しく表現したものであるかどうかを判断できます。この点は、Setインタフェースのaddメソッドのドキュメントからわかります。ここには、「指定された要素がセット内になかった場合、セットに追加します。...セット内にすでにこの要素がある場合、呼出しはセットを変更しません」と書かれています。

そのため、orderが同じ値である2つのアイテムのうち、最初のアイテムがセット内に残り、2番目のアイテムは拒否されることがわかります。つまり、Mélinaが残り、Jaquelinaはセットに格納されません。このことから、選択肢Cは正解である可能性が残りますが、選択肢Dは誤りであることがわかります。

最後に、選択肢Eを見てみます。この選択肢は、順番は予測できない可能性があると言っています。これは興味深い問題です。Setでは、Listとは異なり、アイテムが追加された順番は守られませんが、順番が保証されていないのは、Setの実装が内部の順序を自由に使って重複の検索を高速化しているためです。TreeSetの場合は、ツリー構築メカニズムに使う順番がその順番になります。この例では、OrderedクラスのcompareToメソッドで定義された順序です。この時点で、実装の詳細によるという選択肢に対して適切に反論できるでしょう。これが正解を支える証拠にもなります。ただし、TreeSetは別のインタフェースNavigableSetも実装しています。そのため、順序はパブリックな規約の一部にもなっています。さらに、TreeSetのドキュメントには、NavigableSetの実装では、使用されたコンストラクタに応じて、自然順序付け、またはセットの作成時に指定したコンパレータによって要素

Java 7 では、try-with-resources 機能を活用できるように JDBC API が拡張されました。

の順序付けが行われると書かれています。

そのため、順序が実装によって異なることはなく、選択肢Eは誤りで、正解は選択肢Cになります。

最後に、もう1点確認しておきます。JavaのAPIドキュメントでは、できる限り自然順序付けに「equalsとの一貫性」を持たせることを推奨しています。ComparableのAPIドキュメントには、「自然順序付けにequalsとの一貫性があることは、必須ではありませんが強く推奨されます。これは、明示的なコンパレータを指定しないソート・セットやソート・マップを、自然順序付けにequalsとの一貫性がない要素またはキーと一緒に使用した場合、ソート・セットやソート・マップの動作が「奇妙になる」からです。特に、このようなソート・セットやソート・マップは、セットまたはマップの(equalsメソッドの観点で定義された) 一般的な規約に違反します。ドキュメントには、この設問で取り上げてきた奇妙な動作が明示的に記述されている点に注目してください。

このような動作は、Comparableインタフェースと「自然順序付け」とっては妥当であり、目的に適合するものです。しかし、Comparatorインタフェースは、クラスで複数の異なる順序表現を許容することに主眼を置いたものです。これが重要なのは、学生リストの例で考えるなら、成績順などのさまざまなリストが必要になる可能性があるからです。バスケットボール・チームの監督であれば学生の身長順のリストが必要かもしれませんし、会計担当者であれば学生の未納金額順のリストが必要かもしれません。複数の順序を持たせつつ、そのすべてにequalsとの一貫性を期待するのは不可能です。しかし、一般的に言えば、可能な場合にequalsとの一貫性を持たせるのはやはりよいことだと考えられています。Comparatorのドキュメントには、「ソート・セット（またはソート・マップ）を順序付けするために、equalsと一貫性のない順序付けを行うことができるコンパレータを使う場合は注意が必要です」と書かれています。

しかし、ほとんどの場合は、順序と等価性が一貫しない場合にどのような問題が生じるかを把握しておくことが重要になります。

設問3.正解は選択肢Aです。サーバー型プログラムを長期にわたって適切に実行するためには、多くの場合、メモリ以外のリソースの正しい解放も欠かせません。もちろん、データベース接続は適切な取扱いを求められる重要なリソースです。Java 7では、try-with-resources機能を活用できるようにJDBC APIが拡張されまし



た。とりわけ、Connection、Statement、ResultSetのすべてでAutoCloseableが実装されています。

try-with-resourcesを使っていない選択肢Dは誤りだと考えがちですが、好みでないからと言って安易に除外することはできません。しかし、詳しく見てみると、この選択肢ではメインのデータベース接続がクローズされていないことがわかります。使用した接続の解放は絶対的に必要となる要件ではありませんが、この設問では、すべてのリソースを解放している選択肢が明確に問われています。そのため、接続がオープンされたままになる選択肢Dは正式に除外できます。

残った3つの選択肢は、すべてtry-with-resourcesメカニズムを使っていますが、それぞれ異なるリソースを自動クローズします。選択肢Aは、`Connection`と`Statement`をクローズしますが、`ResultSet`はクローズしません。選択肢Bは、`Statement`と`ResultSet`をクローズしますが、`Connection`はクローズしません。選択肢Cは、`ResultSet`のみをクローズします。設問の要件を考えたとき、このいずれもが完全には動作しないということはあるのでしょうか。とにかく、これらの動作の仕様を確認してみます。`Connection.close()`メソッドのAPIドキュメントには、このメソッドが「ただちにこのConnectionオブジェクトのデータベースとJDBCリソースを解放」と書かれています。ここから、`Connection`のみを自動クローズすれば、この設問の要件を満たせることがわかります。しかし、そのような選択肢はありません。

選択肢AはConnectionとStatementの両方をクローズします。これは逆順でクローズされるため、Statementが先にクローズされ、その後にConnectionがクローズされます。とは言うものの、すべてのリソースがクローズされることに変わりはありません。実際、Statement.close()メソッドのドキュメントには、Connection.close()メソッドと同じように、「ただちにこのStatementオブジェクトのデータベースとJDBCリソースを解放します。...データベースのリソースの占有を避けるために、通常は、作業が終了したらすぐにリソースを解放するようにしてください」というコメントが記載されています。ここから、選択肢AではResultSetが明示的にはクローズされないものの、求められる結果は実現することがわかります。

このタイプのコーディングでは、「サブリソース」の自動クローズが重要になることがあります。それでは、必要なデータを含む結果セットを作成した後、行儀よく不要なStatementとConnectionを解放しようとする

るとどうなるでしょうか。残念ながら、この操作によってResultSetもクローズされ、おそらくは使い終わる前にデータがなくなってしまうでしょう。

</article>

Simon Roberts : Sun Microsystemsがイギリスで初めてJavaの研修を行う少し前にSunに入社し、Sun認定Javaプログラマー試験とSun認定Java開発者試験の作成に携わる。複数のJava認定ガイドを執筆し、現在はフリーランスでシリコン・バレーや世界中の多くの大企業の研修を行う。OracleのJava認定プロジェクトにも継続的に関わっている。