

Java™ magazine

By and for the Java community 

JANUARY/FEBRUARY 2016

UIの裏側 WEB アプリケーション 開発

- 05 **SPRING BOOT:**
早く簡単な
WEBアプリケーション
- 13 **JAX-RS 2.0:**
徹底活用のすすめ

//from the editor /

Andrew Binstock: Java Magazine編集長。以前はDr. Dobbの編集者であり、さまざまな技術文書の出版も担当。また、オープンソースPDFライブラリ「iText」の商業部門として、iText Softwareを共同で創業。さまざまなオープンソース・プロジェクトにも貢献している。著書に、『Practical Algorithms』(Addison-Wesley、1995)を含む3冊のプログラミング関連書籍がある。



2015年の言語の盛衰

去年はJava言語、そしてJVM言語にとって、いつになく良い年でした。

私 はここ数年、新年最初の編集作業で、その前年のプログラミング言語の状況を見てきました。言語の人気に対する一般的認識はゆっくりと移り変わるものですが、ときにプログラミング言語は急激に低迷したり(Objective-C)、逆に思いもよらず浮上することもあります(Java。後ほど説明します)。その人気をはかる上で広く使われている手法がTIOBE Indexです。TIOBE Indexでは、言語が検索された回数をカウントして、その結果を検索総数に対する割合として標準化します。Web検索がはたして人気をはかる正確な代替指標であるのかどうかは議論の余地がありますが、TIOBE Indexには大きな長所が1つあります。それは、15年分を遡って指標のデータを取得できることでの数年にわたる傾向を簡単に確認できます。

当然ながら、言語の人気について確実に分析するには他のソースも必要です。私はGitHubから抽出したデータや、Open Hub、Google Trendsも使っています。GitHubは、オープンソース・プロジェクトや非公開プロジェクトのホストとして一般的です。また、Open Hubはすべてのアクティブなオープンソース・プロジェクトを対象として調査しています。それぞれの基準によって定量化されるデータは異なります

が、データの背景も見て、その後そのデータからどのような有益な情報が得られるかを判断することが大切です。

ほとんどの基準で、Javaにとっては最高の1年でした。先日、TIOBEはJavaに2015年の「Language of the Year」を贈りました。どの言語よりも検索の割合が急激に伸びたという理由からです。TIOBEは、その躍進の原因をAndroidでのJavaの利用だと見ています。私もそう思いますが、それは物語の一面に過ぎません。Java 8が急速に採用されたことも一因でしょう。「Javaはもう終わりなのか」を問う、思わずクリックしたくなる記事が流行りましたが、Javaの躍進はその流行を終わらせました。結局、そのような記事ではいつも長々とした説明の後に「Javaは終わっていない」と締めくくられていたのですが。

GitHubは、若いプログラマーの間の人気をはかる良い手段です。ここでもJavaは活況です。現在の人気順でJavaの上にあるのはJavaScriptだけで、Javaは驚くほど順位を上げました。GitHubは最初の頃、Rubyコミュニティで人気を集めたため、2008年のGitHub人気言語ランキングでJavaは7位。1位はRubyでした。それから2015年までの間にJavaが5つ順位を上げたのは、他に類を見ません。その同じ期間で、

写真: BOB ADLER/GETTY IMAGES

CREATE
THE FUTURE

oracle.com/java

20
YEARS
1995-2015
Java™
ORACLE®





ABHISHEK GUPTA

JAX-RS 2.0: 良いものはすべて利用する

新規クライアントAPI、フィルタ、インターセプタ、その他の有用なREST機能

JAX-RSフレームワークは、RESTfulの原理を実装するアプリケーションやサービスを構築するためのAPIセットです。Plain Old Java Object (POJO) をアノテーションによって修飾して、どこでも使用できるWebプロトコルであるHTTP経由でサービスやビジネス・ロジックの一部を公開できる、単純でありながらも強力なプログラミング・モデルを提供します。

JAX-RSは2008年、JCP仕様のJSR 311リリース1.0として標準化されました。また、Java EE 6仕様(JSR 316)の一部にもなりました。2013年にJAX-RS 2.0(JSR 339)がリリースされました。JAX-RS 2.0は、Java EE 7仕様(JSR 342)の不可欠な要素です。次のエディションであるJAX-RS 2.1(JSR 370)の作業が現在進行中であり、Java EE 8プラットフォームの一部としてリリースされる予定です。

広く普及しているJAX-RS実装として、Jersey(リファレンス実装)、RESTEasy、Apache CXFなどがあります。これらの実装ではいずれも、基本のJAX-RS仕様に準拠し、仕様をサポートしながら、さらに便利な機能も追加されています。

本記事では、JAX-RS 2.0の最新機能に焦点を当てます。JAX-RS 2.0の最新機能には、新規クライアントAPI、フィルタ、インターセプタ、新しい例外処理方法、非同期プログラミングの拡張機能、その他多くの追加機能があります。

新規クライアントAPI

本格的なクライアントAPIが追加される前は、HTTP指向の(REST)サービスと通信するために、開発者はサード・パーティの実装に頼るか、JDKのURLConnection APIを操作する必要がありました。新規クライアントAPI(javax.ws.rs.clientパッケージ内)はかなりコンパクトで無駄がなく、流れるようなAPIです。その一部のクラスとインタ

フェースについて紹介します。

ClientBuilderは、呼出しプロセスを開始するためのクラスであり、buildメソッドや、オーバーロードされたnewClientメソッドがエントリーポイントとなっています。Clientは、オーバーロードされたtargetメソッドによってWebTargetインスタンスを作成するためのインタフェースです。WebTargetは、HTTPリクエスト呼出し用のURIエンドポイントを表したものです。WebTargetを使用して、クエリー、マトリックス、パス・パラメータなどのさまざまな属性を設定できます。WebTargetは、オーバーロードされたrequestメソッドを公開しており、これらのメソッドによってInvocation.Builderのインスタンスを取得します。Invocation.Builderには、さらにHTTPリクエストを構成して、ヘッダー、Cookie、キャッシュ管理などの属性や、メディア・タイプ、言語、エンコーディングなどのコンテンツ・ネゴシエーション関連のパラメータを設定する機能があります。加えて、ClientのbuildXXXメソッドのいずれかを使用して、Invocationオブジェク

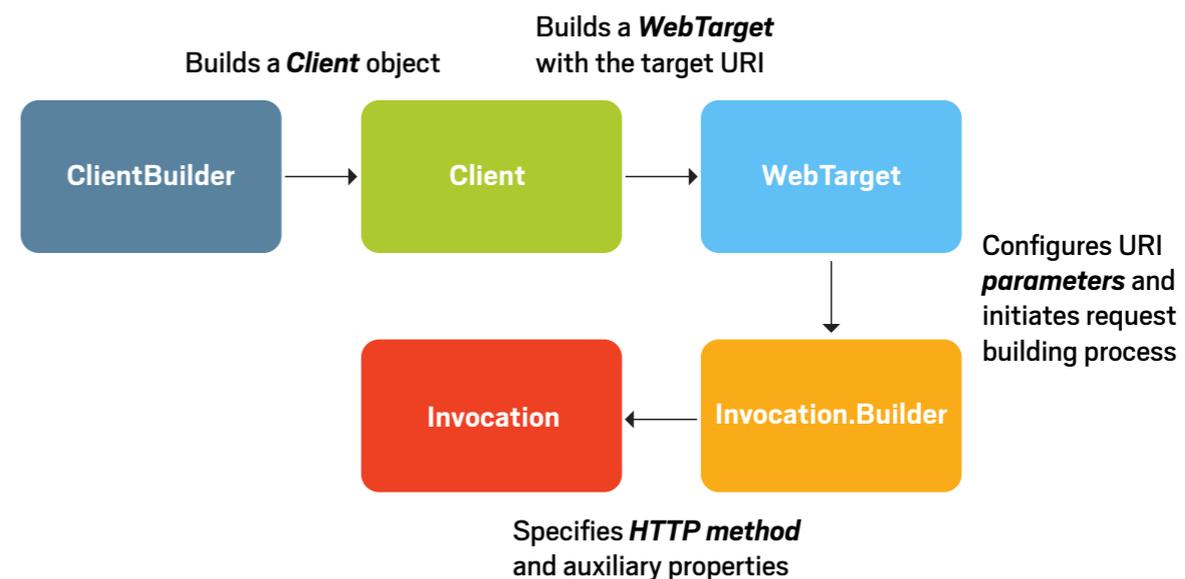


図1: JAX-RS 2.xクライアント



ともできます。図2に、この新しい例外階層を示します。

これらの例外クラスと、そのそれぞれに対応するHTTPエラー・コードのマッピングについては、表1を参照してください。

その他の注目すべき拡張機能

これまでに紹介した目玉機能の他にも、JAX-RS 2.0には、アプリケーションの要件に合致するならばぜひ使用すべき便利な拡張機能が搭載されています。

@BeanParam:このアノテーションを使用して、カスタムのPOJOまたはBeanを、そのインスタンスにParam系の各種アノテーションを付加できる場合にインジェクションできます。Param系のアノテーションとは、@HeaderParam、@CookieParam、@PathParam、@QueryParamなどです。以下に示すとおり、HTTPリクエストの個別の要素をJAX-RSリソースに

インジェクションする代わりに、単純なPOJOを利用してHTTP URIのパラメータを取得できるため便利です。

```
public class CustomerSearchRequest{
    @QueryParam("id")
    private String userid;

    @HeaderParam("Accept")
    private String accept;

    @CookieParam("lastAccessed")
    private Date lastAccessed;
    //getters to fetch the values
}
```

JAX-RSランタイムは、@javax.ws.rs.BeanParamアノテーションが付加されたメソッド引数(またはインスタンス変数)のインスタンスをインジェクションします。

@ConstrainedTo:JAX-RSには、サーバー・サイドとクライアント・サイドに適用可能なプロバイダ・コンポーネントがいくつかあります。インターセプタはその好例で、ReaderInterceptorとWriterInterceptorの両方を、サーバー・サイドにもクライアント・サイドにも適用できます。この場合、状況によってクライアント・サイドまたはサーバー・サイドのいずれかにのみ使用できるように明示的に制限する目的で、@javax.ws.rs.ConstrainedToアノテーションをプロバイダ・クラスに適用できます。

ParamConverter:ParamConverterインターフェースは、自動インジェクションの前提条件を満たしていないカスタムのオブジェクトまたはPOJOを処理する場合に便利です(この前提条件とは、String引数を受け取るパブリック・コンストラクタがあること、またはString引数を受け取り、同じタイプのインスタンスをPOJOとして返す静的なvalueOfメソッドがあることです)。ParamConverterインターフェースの実装を使用して、Param系のアノテーション(@QueryParam、@PathParamなど)によって取得したString型のパラメータをカスタムのPOJOに変換するためのカスタム・ロジックを指定できます。

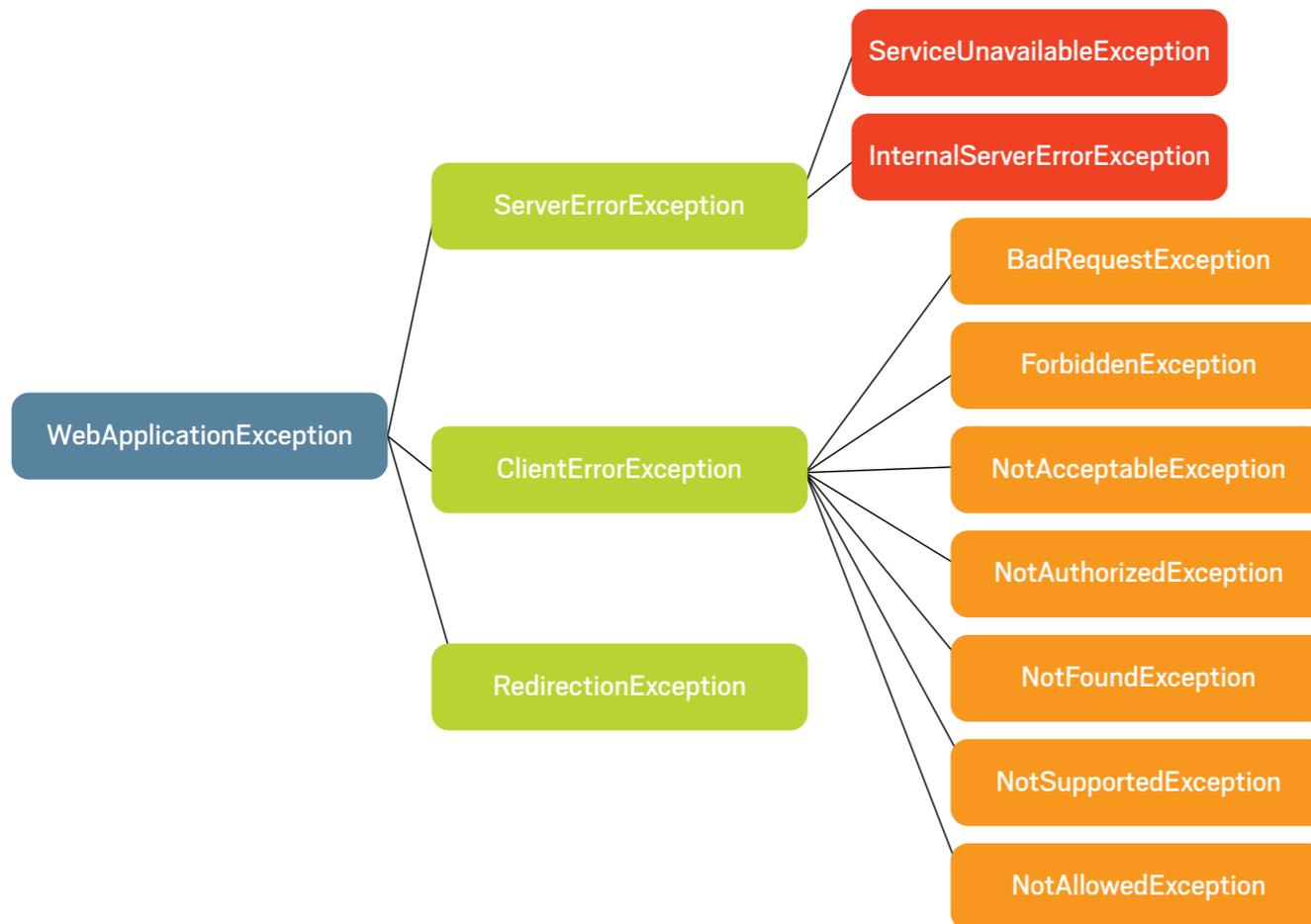


図2: JAX-RS 2.0の例外階層



しかし、アプリケーションでもっと高レベルの構造化オブジェクトを使用してメッセージを表したいと考える場合もよくあります。メッセージ処理メソッドでカスタム・オブジェクトを扱うには、エンドポイントと一緒にWebSocketのDecoder実装を提供する必要があります。ランタイムはこの実装を使用して、受信メッセージをカスタム・オブジェクト型のインスタンスに変換します。反対に、送信メソッドでカスタム・オブジェクトを扱うには、WebSocketのEncoder実装を提供する必要があります。ランタイムはこの実装を使用して、カスタム・オブジェクトのインスタンスをネイティブWebSocketメッセージに変換します。この仕組みを要約したものが図1になります。

図1の上部に示したエンドポイントはクライアントとの間で文字列を交換しており、下部のエンドポイントはエンコーダとデコーダを使用して、FooオブジェクトからWebSocketテキスト・メッセージへの変換とその逆を実行しています。

Java WebSocket APIには、javax.websocket.Decoderインタフェースおよびjavax.websocket.Encoderインタフェースのファミリがあり、希望の変換タイプに応じて選択できます。たとえば、テキスト・メッセージをFooというカスタム開発者クラスのインスタンスに変換するDecoderを実装するためには、Fooを総称型として使用してDecoder.Text<T>インタフェースを実装します。具体的には、以下のメソッドを実装します。

```
public void sendObject(Object message)
```

throwsIOException, EncodeException

これはデコーダの主要メソッドで、新しいテキスト・メッセージを受け取るたびに呼び出されて、Fooクラスのインスタンスを生成します。その後で、ランタイムが、エンドポイントのメッセージ処理メソッドにこのクラスを渡すことができます。

Decoderにはきょうだいクラスがあり、ブロッキングI/Oストリーム(こちらサポート対象)の形で受け取るWebSocketメッセージとバイナリのWebSocketメッセージをデコードします。

カスタム開発者クラスFooのインスタンスをWebSocketテキスト・メッセージに変換するEncoderを実装するためには、Fooを総称型として使用してEncoder.Text<T>インタフェースを実装します。具体的には、以下のメソッドを実装します。

```
public String encode(Foo foo)
throws EncodeException
```

この実装により、Fooインスタンスが文字列に変換されます。RemoteEndpointのsendObject()メソッド(前述)を呼び出して、Fooクラスのインスタンスを渡す場合、Java WebSocketランタイムはこの文字列を必要とします。Decoderと同じように、Encoderにもきょうだいクラスがあり、メッセージを送信するために、カスタム・オブジェクトをバイナリ・メッセージに変換するか、またはカスタム・オブジェクトをブロッキングI/Oストリームに書き込みます。

この方法をエンドポイントで使用するの簡単です。@ClientEndpointと@ServerEndpointの定義で確認したように、エンドポイントで使用するデコーダおよびエンコーダの実装を、それぞれdecoders()属性とencoders()属性に指定するだけで完了です。

Javaプリミティブ型のエンコーダまたはデコーダを独自に設定した場合、ランタイムでその型のデフォルトになっているエンコーダまたはデコーダは、当然ながらオーバーライドされます。

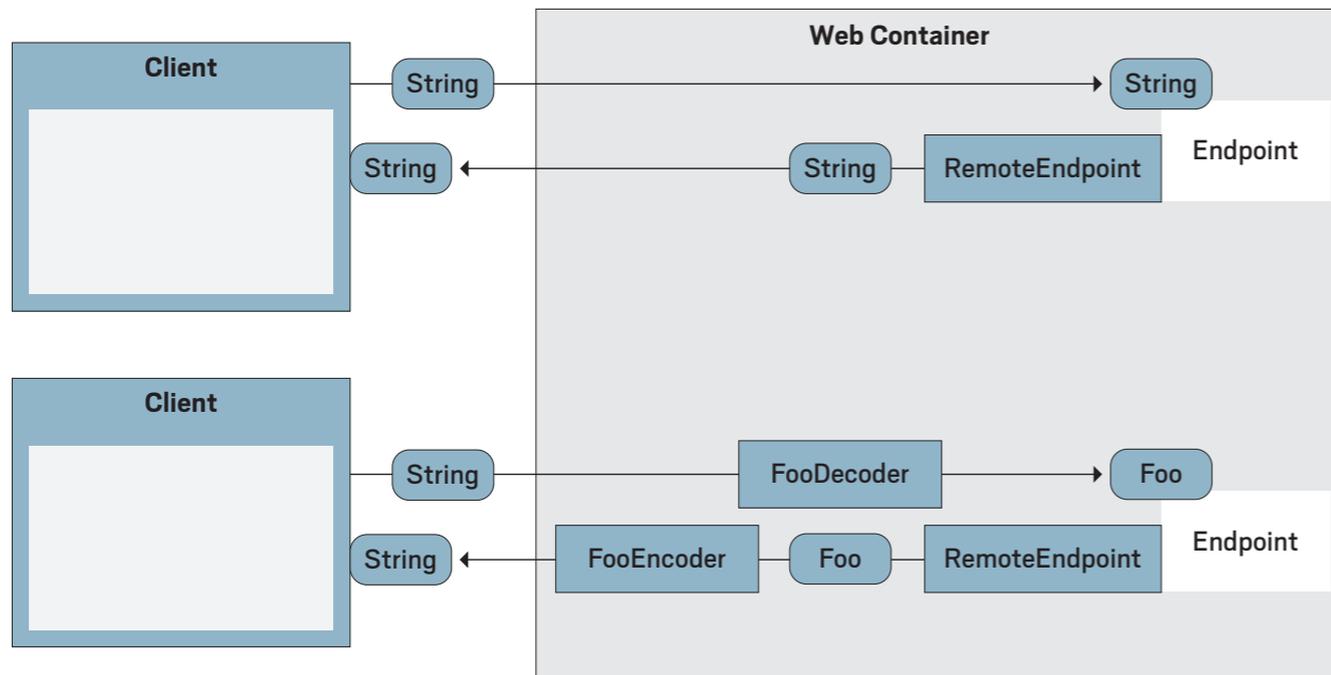


図1:エンコーダとデコーダ



コードを詳しく調べる前に、アプリケーション構築の全体像を把握しましょう。このWebページは、JavaScript WebSocketクライアントを使用して、すべてのチャット・メッセージを送受信しています。Webサーバー上に、ChatServerという1つのJava WebSocketエンドポイントがあり、誰かがチャット・ルームに入退室するか、グループ宛てにメッセージが送信されるたびに、複数のクライアントからのチャット・メッセージをすべて処理し、アクティブに参加しているクライアントを追跡し、トランスクリプトを維持し、すべての接続クライアントに更新をブロードキャストします。このアプリケーションは、カスタム・オブジェクトに対してWebSocketのEncodersとDecodersを使用して、すべてのチャット・メッセージをモデル化しています。

リスト4のChatServerエンドポイントに注目してください。[サイズの関係上、リスト4は記事内に記載していません。本記事の[ダウンロード・ページ](#)からダウンロードできます (編集部より)]

このコードには着目すべき点が多くあります。第一に、これは、/chat-serverという相対URIにマッピングされたサーバー・エンドポイントです。このエンドポイントは、それぞれChatEncoderおよびChatDecoderというエンコーダ・クラスとデコーダ・クラスを使用しています。

初めてJava WebSocketエンドポイントを見るときは、ライフサイクル・メソッドからチェックすると良いでしょう。ご存知のとおり、ライフサイクル・メソッドには、@OnOpen、@OnMessage、@OnError、@OnCloseというアノテーションが付加されています。このようにしてChatServerクラスを見ると、新規クライアントの接続時にChatServer WebSocketが最初に処理することは、チャットのトランスクリプト、セッション、EndpointConfigを参照するインスタンス変数の設定であると分かります。クライアントが接続するたびに、新しいエンドポイント・インスタンスが作成されるため、チャット・ルーム内の参加者ごとに、独自のチャット・サーバー・インスタンスがエンドポイントに関連付けられます。EndpointConfigの数は、論理的なWebSocketエンドポイントごとに常に1つになるため、各ChatServerインスタンスのendpointConfigインスタンス変数が指すのは、EndpointConfigクラスの単一共有インスタンスになります。このインスタンスはシングルトンであり、任意のアプリケーション・ステートを格納できるユーザー・マップを保持しています。このため、アプリケーションのグローバル・ステートを格納する場所として最適です。クライアント接続ごとに固有のセッション・オブジェクトが作成されるため、それぞれのChatServerインスタンスは、独自のSessionインスタンスを参照します。このインスタンスは、Transcriptクラスのコードに従って、関連付けられたクライアントを表します。詳しくは、リスト5を参照してください。

■ リスト5: Transcriptクラス

```
import java.util.ArrayList;
import java.util.List;
import javax.websocket.*;

public class Transcript {
    private List<String> messages =
        new ArrayList<>();
    private List<String> usernames =
        new ArrayList<>();
    private int maxLines;
    private static String
        TRANSCRIPT_ATTRIBUTE_NAME =
            "CHAT_TRANSCRIPT_AN";

    public static Transcript
        getTranscript(EndpointConfig ec) {
        if (!ec.getUserProperties().
            containsKey(TRANSCRIPT_ATTRIBUTE_NAME)) {
            ec.getUserProperties()
                .put(TRANSCRIPT_ATTRIBUTE_NAME,
                    new Transcript(20));
            return (Transcript) c.getUserProperties()
                .get(TRANSCRIPT_ATTRIBUTE_NAME);
        }

        Transcript(int maxLines) {
            this.maxLines = maxLines;
        }

        public String getLastUsername() {
            return usernames.get(usernames.size() - 1);
        }

        public String getLastMessage() {
            return messages.get(messages.size() - 1);
        }
    }
}
```



鋭い開発者の方なら、このコードを見てmoduleがJava 9での新しいキーワードであることに気付くことでしょう。このモジュールをコンパイルするためには、ディレクトリ構造を新しいモジュール式ソースのルールに従って配置する必要があります。今回は、com.jdk9exというモジュールであるため、次のような構造になります。

```
|____src
| |____com.jdk9ex
| | |____com
| | | |____jdk9ex
| | | | |____Main.java
| | |____module-info.java
```

モジュール名は、モジュール内にあるJavaパッケージの通常の命名規則とは関係ないことに注意してください。このモジュール・コードをコンパイルするためには、モジュールを認識したビルドが行われるように指定してJavacを実行する必要があります。

```
$ javac -d modules/com.jdk9ex \
  src/com.jdk9ex/module-info.java \
  src/com.jdk9ex/com/jdk9ex/*
```

このコンパイルによって、一連のモジュール式クラス・ファイルが生成されますが、jimageは生成されません。このコードを実行する場合も、モジュールが必要であることを明示的にJVMに伝える必要があります。

```
$ java -modulepath modules -m \
  com.jdk9ex/com.jdk9ex.Main
Hello Modules!
```

基本的な「Hello World」サンプルは、プログラマーにとっておなじみのパターンです。Java 9のモジュール化の現状を完全に理解したい開発者の方には、モジュール・システムを実際に触ってみて、どのように動作するかを確認することを強くお勧めします。

問題が起きる可能性: どんな新技術でも、早期に導入したユーザーに問題が起きることは避けられません。ここでは基本的な問題の事例をいくつか見てゆきます。特に、ビルドに失敗した場合や、モジュール式のJavaを正しく呼び出せなかった場合は、新しい種類の例外が登場します。

```
$ java -modulepath modules -m com.jdk9ex.Main
Error occurred during initialization of VM
java.lang.module.ResolutionException:
  Module com.jdk9ex.Main not found at
java.lang.module.Resolver.fail
  (java.base@9.0/Resolver.java:880)at
java.lang.module.Resolver.resolve
  (java.base@9.0/Resolver.java:193)at
java.lang.module.Resolver.resolve
  (java.base@9.0/Resolver.java:173)at
java.lang.module.Configuration.resolve
  (java.base@9.0/Configuration.java:229)at
jdk.internal.module.ModuleBootstrap.boot
  (java.base@9.0/ModuleBootstrap.java:174)at
java.lang.System.initPhase2
  (java.base@9.0/System.java:1242)
```

[編集部注:スペースの関係上、一部の行を折り返しています。]

この1つのスタック・トレースには、次のような新しい項目が表示されています。

- java.lang.moduleなどのパッケージ
 - ResolutionExceptionなどの例外
 - java.base@9.0/ModuleBootstrap.javaなどのコード上の位置
 - java.lang.System::initPhase2などの起動用エントリ・ポイント
- この内容から、jimages用の新しいモジュール・パスがスタック・トレースに直接存在するようになったこともわかります。

では、内部実装クラスにアクセスできるかどうか試してみましょう。次のコードは、Java 8でコンパイルして実行できます。



```
import sun.invoke.util.BytecodeName;

public class InternalsCheck {
    public static void main(String[] args) {
        String bName =
            BytecodeName
                .toBytecodeName("java.lang.Object");
        System.out.println(bName);
    }
}
```

ご想像のとおり、このコードはjava.lang.Objectの内部名を表示するものです。しかし、モジュールを使用する場合、実装クラスへのアクセスが制限されるため、このコードはコンパイルすらできなくなります。

```
$ javac -d modules/com.jdk9ex \
    src/com.jdk9ex/module-info.java \
    src/com.jdk9ex/com/jdk9ex/*
```

```
src/com.jdk9ex/com/jdk9ex/InternalsCheck.java:8:
error: package sun.invoke.util does not exist
import sun.invoke.util.BytecodeName;
                        ^
```

```
src/com.jdk9ex/com/jdk9ex/InternalsCheck.java:12:
error: cannot find symbol
    String bName = BytecodeName.
toBytecodeName("java.lang.Object");
                  ^
```

```
symbol:   variable BytecodeName
location: class InternalsCheck
2 errors
```

この2つのエラーは、Javaモジュール・システムの目標の1つである、API整合性が強力に保証されていることを証明しています。現在のところ、javacは実装クラスBytecodeNameへのアクセスを防ぐようにアップグレードされていますが、IDEやJava環境のその他のツールはまだこの新たな現実を認識していません。これも、JDK 9が一般の開発者向けに公開されるまでの道のりはまだ長いことを示す印の1つです。

よい知らせなのは、すべての主要なIDEベンダーが積極的に新し

いバージョンの開発に取り組んでおり、Java 9を完全にサポートしようとしていることです。たとえば、OracleはNetBeans 9のJDK 9サポートに注力することを発表し、EclipseのJDTもすべての新機能の開発をJDK 9ブランチに向けています。

Unsafeについて: Javaプラットフォーム、特にOpenJDKプロジェクトに関わる開発者の立場から見れば、「ショットガン・プライバシー」は重大な問題です。javaパッケージやjavaxパッケージは「禁止パッケージ」としてSecurityExceptionメカニズムによって保護されていますが、この仕組みは内部実装クラスには適用されません。

Javaのライブラリやフレームワークでは、そういった内部実装の詳細に直接アクセスすることは想定されていません。Javaの幹事は、そのようなことをするのは危険な行為だと言いつけてきましたが、このルールを破って内部実装の詳細にリンクしようとする開発者に対応しようとはしてきませんでした。

警告が繰り返されてきたにもかかわらず、多くの有名なJavaライブラリがこのルールを破り、プラットフォームの内部仕様にアクセスしています。これによってJDK開発者たちは非常に難しい状況に追い込まれていることから、大きな問題となっています。

1つの選択肢は、内部仕様へのアクセスを禁止する変更は現状では非常に難しいと認め、いくつかの内部仕様を標準APIの一部に取り入れ、サポート対象に加えることです。もう1つの選択肢は、内部仕様を変更し、無数の本番アプリケーションが依存しているライブラリやフレームワークに影響を与えるか、または動かなくしてしまうことです。

もっとも有名で広く使用されており、もっとも危険だと言える直接アクセスの例が、sun.misc.Unsafeクラスです。このクラスには、モジュール・システム採用の圧力が大きくかけられています。JDK 9でUnsafeへのアクセスを単純に遮断してしまうと、ほぼすべてのJavaアプリケーションをアップグレードできなくなってしまうでしょう。一般的なアプリケーションは、ほぼ例外なく直接的または間接的にUnsafeの機能に依存したライブラリを

多くの有名なJavaライブラリがこのルールを破り、プラットフォームの内部仕様にアクセスしています。これによってJDK開発者たちは非常に難しい状況に追い込まれていることから、大きな問題となっています。



- スケジューラ戦略:** コンテナを実行する最適なノードを決定するために適用できるもので、spread (デフォルト)、binpack、randomから選択できます。デフォルトの戦略では、実行するコンテナの数が最小になるようにノードが最適化されます。
- ノード検出サービス:** Swarmマネージャは、ホストされている検出サービスと通信します。検出サービスは、Swarmクラスタ内のIPアドレスの一覧を保持しています。Dockerハブでは、開発用の検出サービスがホストされています。本番環境では、デフォルトの検出サービスをetcd、Consul、ZooKeeperなどの別のサービスで置き換えます。静的なファイルを使用することも可能です。
- 標準Docker API:** Docker Swarmは、標準Docker APIに対応しています。そのため、単一のDockerホストと通信するツールであればどのようなものでも、複数のホストにシームレスに拡張できます。したがって、Docker CLIで複数のDockerホストを構成するシェル・スクリプトを使用していた場合、同じCLIでDocker Swarmクラスタと通信できます。Couchbaseを使用するマルチコンテナJavaアプリケーションを複数のホストにまたがるDocker Swarmクラスタにデプロイするためには、次の手順が必要になります。

1. 検出サービスの作成
 2. Docker Swarmクラスタの作成
 3. アプリケーション環境の起動
 4. Couchbaseサーバーの構成とサンプル・データのロード
 5. Docker Swarmクラスタへのアプリケーションのデプロイ
- デプロイされたアプリケーションのノード全体への配布は、Docker Swarmが行います。

図2は、アプリケーション内の各コンポーネントの概念図です。

- クラスタには2つのノードがあります。1つはマスター、もう1つはワーカー・ノードです。
- クラスタでアプリケーション環境を起動するために、Docker Composeを使用します。
- 検出サービスは、クラスタの外部に存在する別のマシンにホストされています。

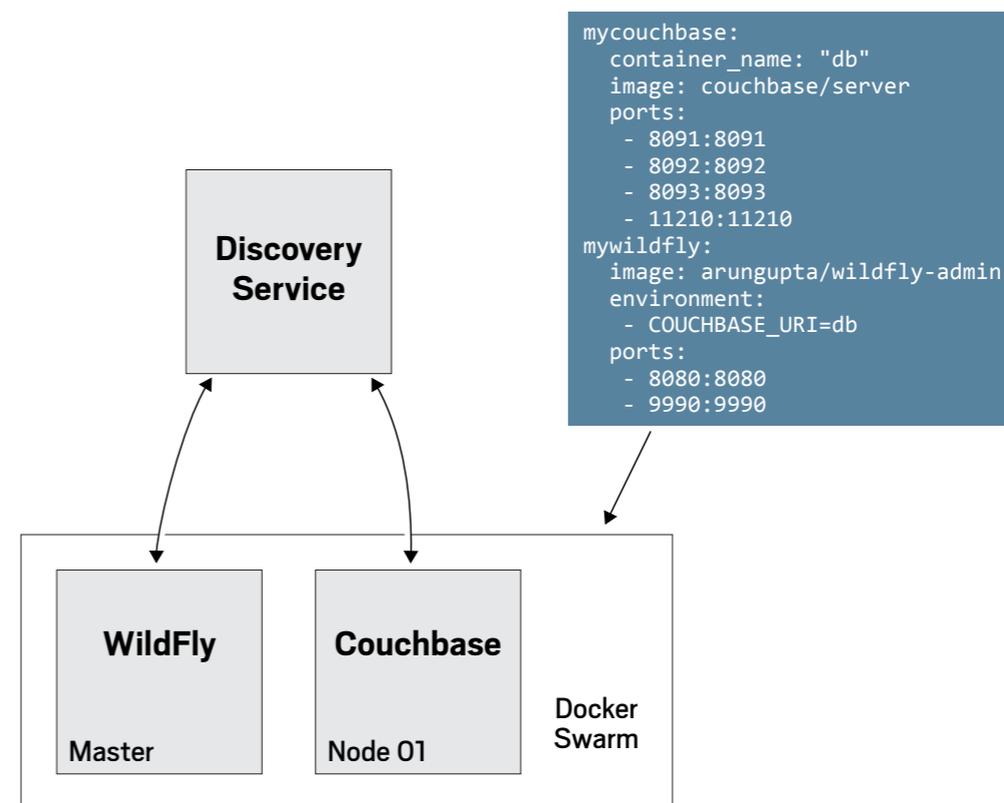


図2: Docker Swarmで実行されるシンプルなサービス

Couchbaseの構成とWildFlyへのアプリケーションのデプロイには、Mavenを使用します。**注:** 本記事では、WildFlyとCouchbaseを使用していますが、Tomcat、GlassFishなどのアプリケーション・サーバーや、MySQL、JavaDBなどのデータベース・サーバーも簡単に使用できます。通常、イメージとそれに含まれるアプリケーションは適切に更新する必要があります。

それでは、この構成の詳しい設定方法について見てゆきましょう。

検出サービスの作成

この例では、検出サービスにConsulを使用しますが、etcdやZooKeeperなどの同様のツールを使用することもできます。

まず、検出サービスをホストするDockerマシンを新しく作成します。



