

# Java7モジュールリティ、 やさしく教えます。

---

フリーエンジニア／テクニカルライター

白石俊平

# はじめに

---

## □ 自己紹介

- 白石俊平
  - フリーエンジニア・テクニカルライターをやっております。(よく載る媒体: マイコミジャーナル・ZDNet) Java/AJAXを中心として活動中。GoogleさんやAdobeさんと絡むことが多いです。
  - 近々起業するかもしれません。以後お見知りおきを！
-

# 当セッションの概要

---

- 文法の変更どころじゃない、Java開発に確信をもたらす(かもしれない)仕様、「スーパーパッケージ」と「モジュールシステム」についてお話しします。
  - 今回の内容は**すべてアーリードラフト版**です。正式リリース時には思い切り変わっているかもしれませんので、ご容赦を。
-

# セッションの流れ

---

- 5分でわかるモジュラリティ概要
  - JSR-294 SuperPackageについて
  - JSR-277 JavaModuleSystemについて
  - JSR-294とJSR-277について
  - まとめ
-

# 何を解決するのか？

---

- Javaにはモジュールと言う概念がありません。これによって以下のような問題点が発生します。
    - 「Javaのpublicはpublicすぎる」
    - JARファイルの取り扱いが面倒
    - クラスパスの取り扱いが面倒
-

# Javaのpublicはpublicすぎる

---

- 外のパッケージから利用されるクラスを作るには、publicとして宣言するしかない
  - → クラスパス上のあらゆるクラスから利用可能になってしまう
  - → 互換性を担保する義務が生じる！
  - ユーティリティクラスなどに多い
    - プロジェクト内だけで、XXXUtilsクラスを広く使いたい…
-

# JARファイルの取り扱いが面倒

---

## □ 依存関係

- JARだけあっても、何に依存しているのやら。
- 「hibernate3.jar」が単体で落ちてても、使いようがありません・・・

## □ 配布方法

- Apache/Jakarta/SourceForge/Java.net... のプロジェクトページにアクセスして、zipをダウンロードして、解凍してJarを取り出して・・・
  - 結局、アプリやライブラリに全部同梱されている
-

# ために

- HDD全体から、「commons-logging\*.jar」を検索してみました。

242 ファイルが見  
た。探していたファ



# JSR-294 スーパーパッケージ

---

- 正式名称は「Improved Modularity Support in the Java Programming Language」(Javaプログラミング言語におけるモジュラリティサポートの改善)
  - Java言語の要素を拡張し、「publicすぎる」問題を解決
-

# JSR-271 Java Module System

---

- JARファイルが抱える問題を解決しようとするもの
    - モジュールとしての役目を果たさない
    - 配布方法も面倒
  
  - 「JAR、やめちゃえ」→JAMファイル
    - 「Java Module」の略
    - JARファイルを基にし、厳密なモジュールとしてのメタデータを追加したもの
-

# では、いよいよ

---

- 詳しい説明に入っていきます。
  - ここからの流れ
    - JSR-294 スーパーパッケージの詳細
    - JSR-277 Java Module Systemの詳細
    - 両仕様の関係
    - まとめ
    - 補足
-

# JSR-294 スーパーパッケージ

---

- Java言語の要素を拡張し、「publicすぎる」問題を解決
  - 既存のJavaパッケージとはそれほど関連のない、「スーパーパッケージ」という枠組により、**publicクラスの公開範囲を自在に指定**
-

# 具体例：内部的なユーティリティクラス

---

- 以下のようなクラスがあるとします。
  - `com.examples.pub.PublicClass`  
(外部からも広く使われることを想定したクラス)
  - `com.examples.util.PublicUtils`  
(外部からも広く使われることを想定したクラス)
  - `com.examples.util.InternalUtils`  
(内部的だが、パッケージ外からも使われるのでpublicなクラス)
  - `InternalUtils`クラスをモジュール外部から隠蔽し、  
その他は外部から利用可能にするには？
-

# スーパーパッケージの作成

---

## □ スーパーパッケージの利用手順

- まず、対象となるクラスすべてを名前付きのスーパーパッケージに含める→アクセスできなくなる
- その後、外からアクセスしてよいものだけ「エクスポート」する

## □ スーパーパッケージの作成手順

- 適切と思われる場所(今回はcom.example)に、「super-package.java」と言うファイルを作成
  - 同ファイルに、パッケージ定義を記述
-

# super-package.java

---

```
superpackage com. examples {  
    member package com. examples. pub;  
    member package com. examples. util;  
  
    export com. examples. pub. PublicClass;  
    export com. examples. util. PublicUtil;  
    // InternalUtilは書かない  
}
```

---

# super-package.java

---

予約語

スーパーパッ  
ケージ名

```
superpackage com. examples {  
    member package com. examples. pub;  
    member package com. examples. util;  
  
    export com. examples. pub. PublicClass;  
    export com. examples. util. PublicUtil;  
    // InternalUtilは書かない  
}
```

# super-package.java

---

予約語

スーパーパッ  
ケージ名

メンバー指定

```
superpackage com. examples {  
  member package com. examples. pub;  
  member package com. examples. util;  
}
```

```
export com. examples. pub. PublicClass;  
export com. examples. util. PublicUtil;  
// InternalUtilは書かない
```

エクスポート  
指定

```
}
```

---

# super-package.java

---

予約語

スーパーパッ  
ケージ名

メンバー指定

```
superpackage com. examples {  
  member package com. examples. pub;  
  member package com. examples. util;  
}
```

```
export com. examples. pub. PublicClass;  
export com. examples. util. PublicUtil;  
// InternalUtilは書かない
```

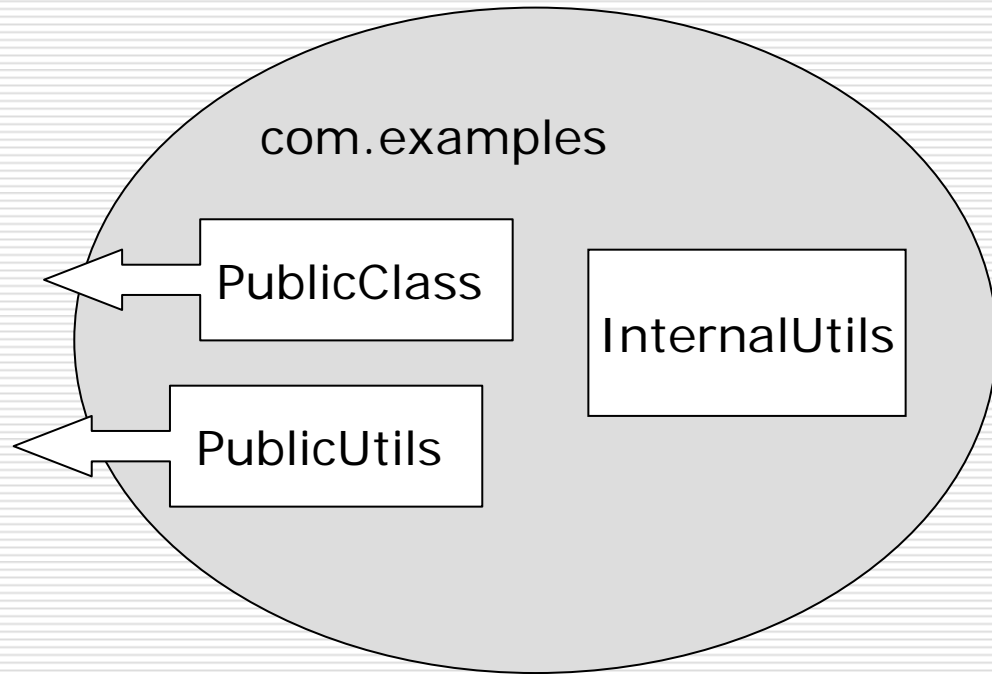
エクスポート  
指定

```
}
```

---

# 図にするとこんなかんじ

---



# 完成後のファイル構成

---

- /com/examples/**super-package.java**
  - /com/examples/pub/PublicClass.java
  - /com/examples/util/PublicUtils.java
  - /com/examples/util/InternalUtils.java
-

## エクスポートされたライブラリを使うには

---

- 前述のスーパーパッケージ  
「com.examples」でエクスポートされていた  
クラスを、「hoge.Hoge」クラスから使いた  
い！
  - スーパーパッケージ内のクラスにアクセスする  
ためには、
    - 使う側のクラスにもスーパーパッケージを指定
    - その上で、使う側が使われる側をメンバに含める
-

# エクスポートされたライブラリを使うには

---

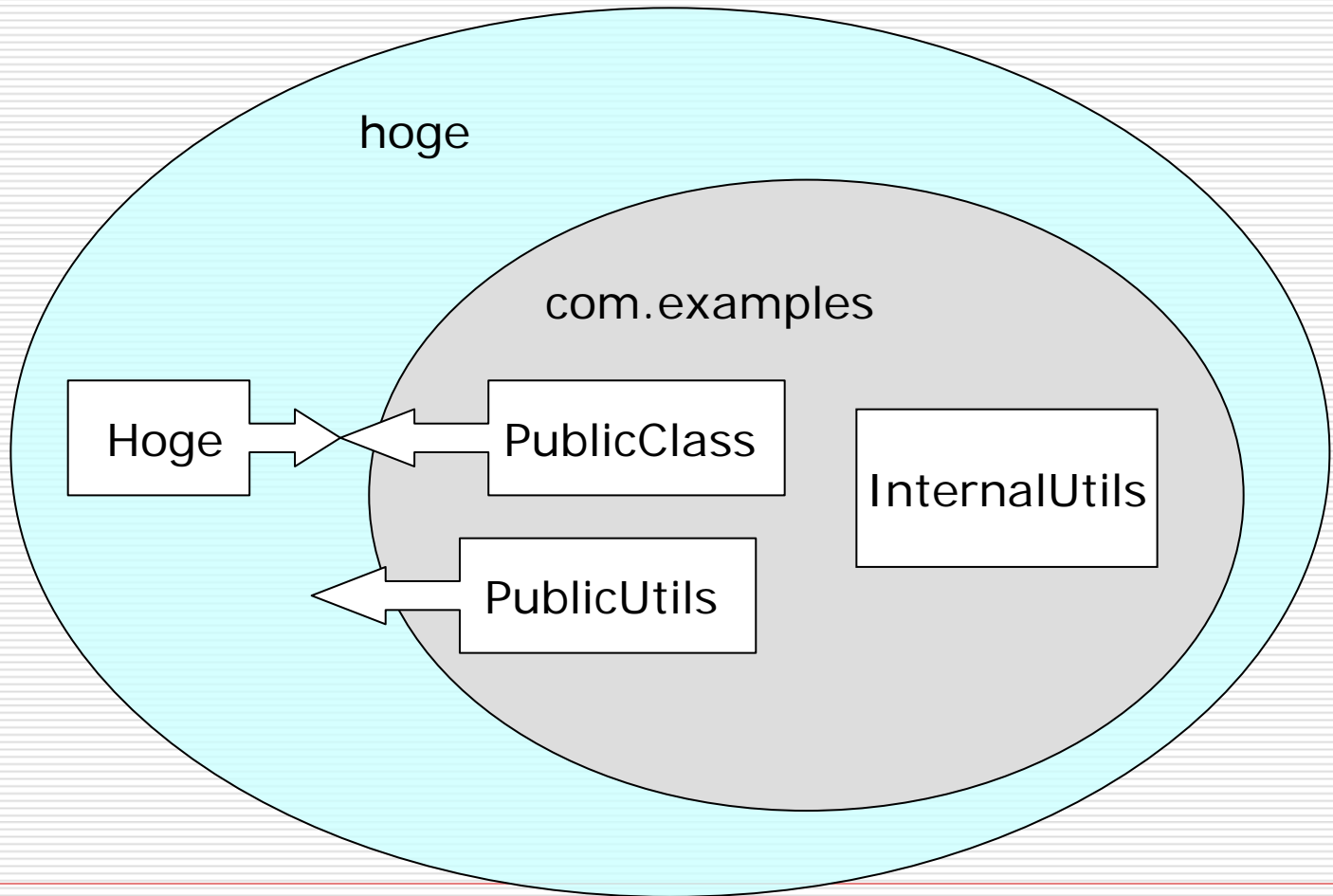
## □ hoge/super-package.javaを作成

```
superpackage hoge {  
    member package hoge;  
    member superpackage com. examples;  
}
```

hogeパッケージとcom.examplesが  
同じスーパーパッケージに属するので、  
アクセス可能になる

# 図にするとこんなかんじ

---



## ここまでのまとめ

---

- スーパーパッケージのメンバになったクラスは、他のクラスからアクセスできなくなる。
  - そこに風穴を空けるのがエクスポート
  - エクスポートされたクラスは、そのスーパーパッケージをメンバとするスーパーパッケージ (エンクロージングスーパーパッケージ) 内で使用可能となる。
-

# スーパーパッケージは入れ子にできる

---

- 複数のスーパーパッケージを合わせて、ひとつの大きなスーパーパッケージを作ると言うイメージ
  - 入れ子になったスーパーパッケージごとに細かくアクセス制御ができる
-

## 例：先ほどの例を少し改造

---

- com.examplesの「子供」として  
com.example.utilスーパーパッケージを作る
  - アクセス制御はそのまま
-

# 例：先ほどの例を少し改造

---

## □ com.examples.utilスーパーパッケージ

```
superpackage com.examples.util
  member com.examples {

  member package com.examples.util;
  export com.examples.util.*;
}
```

# 例：先ほどの例を少し改造

---

## □ com.examples.utilスーパーパッケージ

```
superpackage com.examples.util  
  member com.examples {  
  
  member package com.examples.util;  
  export com.examples.util.*;  
}
```

com.examplesのメンバーであることを宣言

# 例：先ほどの例を少し改造

---

## □ com.examplesスーパーパッケージ

```
superpackage com. examples {  
  member package com. examples. pub;  
  member superpackage com. examples. util;  
  
  export com. examples. pub. PublicClass;  
  export com. examples. util. PublicUtil;  
  // InternalUtilは書かない  
}
```

# 例：先ほどの例を少し改造

---

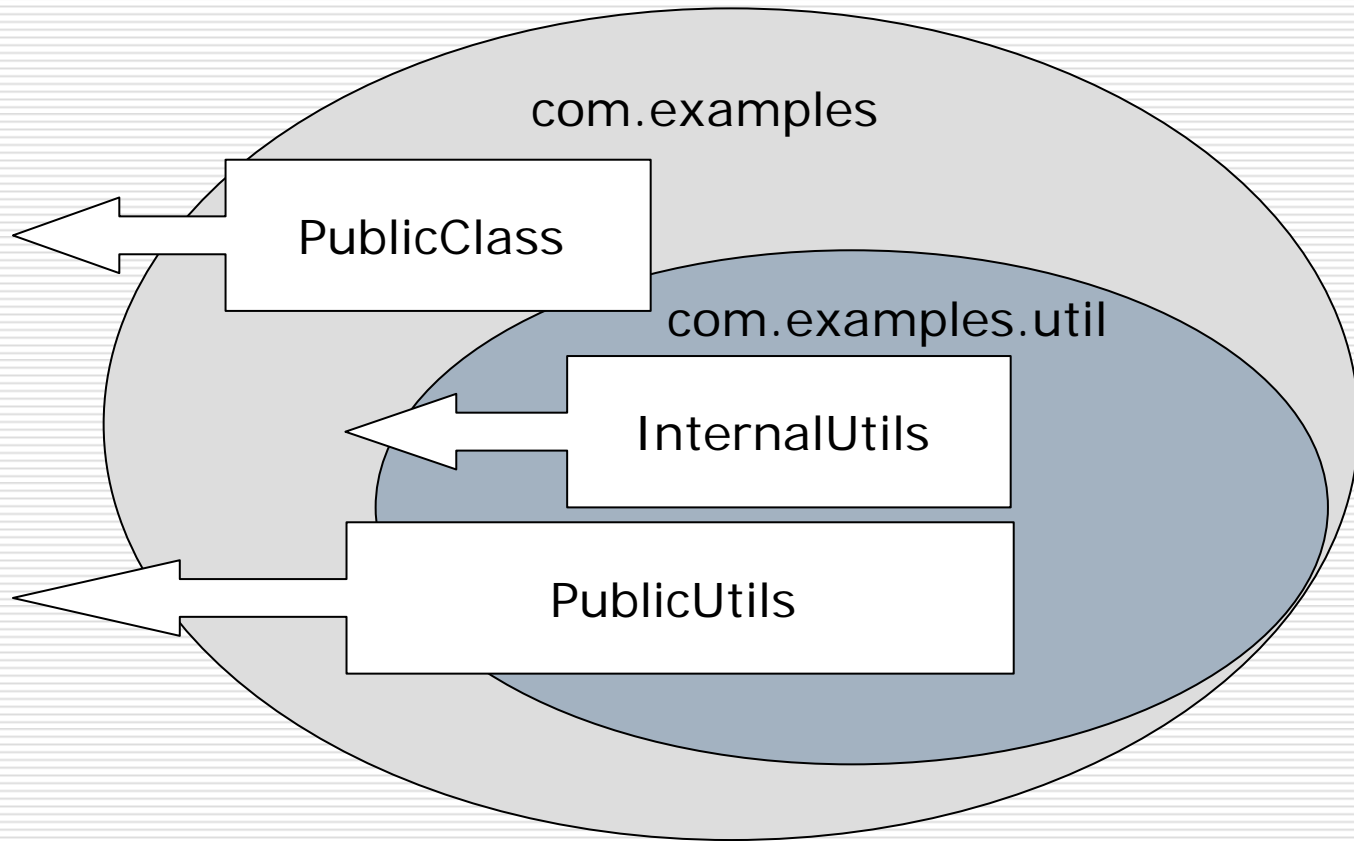
## □ com.examplesスーパーパッケージ

```
superpackage com. examples {  
  member package com. examples. pub;  
  member superpackage com. examples. util;  
  
  export com. examples. pub. PublicClass;  
  export com. examples. util. PublicUtil;  
  // InternalUtilは書かない
```

utilパッケージをメンバに含める

# 図にするとこんなかんじ

---



# わかりにくかったら

---

- member句を消してみるとわかるかも

```
superpackage com.examples.util
  member com.examples {

    member package com.examples.util;
    export com.examples.util.*;
  }
```

エクスポートの範囲を想像してみてください！

---

# スーパーパッケージのまとめ

---

- super-package.javaファイルにて定義
  - スーパーパッケージに入ったメンバは、基本的に外からアクセス不可。
  - エクスポートにより、外部に公開可能。
  - スーパーパッケージは入れ子にもできる
-

# 大して重要じゃない(私見)話

---

- スーパーパッケージを指定されていないクラス (つまり、Java6までと同じ) は、「**匿名スーパーパッケージ**」に属する。
  - 匿名スーパーパッケージに属する型は、どこからでもアクセス可能 (つまり、Java6までと同じ)
-

# JSR-277 Java Module System

---

- Javaプラットフォームに「モジュール」という概念を持ち込むためのJSR
  - Javaモジュールとは？
    - “カプセル化の単位です” (Early Draftより)
    - 仕様書では、以下のように書かれている
      - 「開発モジュール」・・・スーパーパッケージ
      - 「デプロイメントモジュール」・・・JSR-277
    - 「JAMのこと」だと思っていれば間違いない
-

# モジュールの構成要素

---

## □ 代表的な構成要素

- 名前・・・他とかぶらないよう、逆ドメイン名を基本とする
  - バージョン・・・「1.1.0-rc1」など
    - メジャー[.マイナー[.マイクロ[.アップデート]]][-修飾子]
  - 依存関係、その他メタデータ
    - 後述します。
-

# モジュールの物理的な形式

---

- JAM(JAVA Module)ファイル
    - JARファイルフォーマットを基本として、モジュールメタデータ  
(MODULE-INF/METADATA.moduleファイル。  
詳しくは後述)を付け加えたもの。
  - ファイル名は「名前-バージョン.jam」となる
    - さらに、プラットフォームとCPUアーキテクチャも付け加えることができる
-

# JARからJAMへ・・・ファイル名だけ

---

- struts-1.2.jar  
→ `org.apache.struts-1.2.jam`
  - フル修飾名に変更
  - 拡張子はjam
-

# JARからJAMへ・・・ファイル名だけ

---

- swarmcache-1.0rc2.jar →  
net.sf.swarmcache-1.0-rc2.jam
  - 「rc2」の部分は修飾名とする
-

# JARからJAMへ・・・ファイル名だけ

---

- hibernate-core-3.2.6-ga.jar  
→ org.hibernate.core-3.2.6.jam
  - 「GA」(Generally Available、つまり正式リリース版)などの場合は、修飾子を使用しない
-

# JARからJAMへ・・・ファイル名だけ

---

- mail.jar  
→ com.sun.mail-1.4.jam(?)
  - 「ダメ、ゼッタイ。」の代名詞とされてきた、JavaMailのJARファイル名もこのとおり。
-

# JAMファイルの詳細

---

## □ JAM(JAVA Module)ファイル

- JARファイルフォーマットを基本として、モジュールメタデータ  
(MODULE-INF/METADATA.moduleファイル)を付け加えたもの。

xxxx-1.2.3.jam

/META-INF/MANIFEST.MF

**/MODULE-INF/METADATA.module**

/com/example/A.class

/com/example/B.class

---

# モジュールメタデータファイルの内容

---

- 何を記述するのか？
    - 名前
    - インポート
    - エクスポートするクラス
    - モジュールのメンバ
    - 拡張メタデータ
  - ファイルフォーマットは次のページに
-

# メタデータファイルフォーマット

---

## □ こんな感じ

```
(  
name=com.wombat.webservice  
extensible-metadata=[@Version( "1.0" )]  
imports=[ImportModule(org.foo.xml, @VersionConstraint( "1.0+" )),  
         ImportModule(org.foo.soap, @VersionConstraint( "2.0+" ))]  
class-exports=[x.y.z.ClassA, x.y.z.InterfaceB]  
members=[x.y.z.ClassA, x.y.z.InterfaceB, x.y.z.ClassC]  
)
```

## □ JSONでもYAMLでもスクリプトでもない・・・なんだか半端な形式。。

---

# インポート

---

- このモジュールが依存しているモジュールを記述する。

```
ImportModule(org. foo. xml, @VersionConstraint( "1.0+" )
```

- 柔軟なバージョン指定が可能(さらに、プログラムによる指定も可)
    - 「1.2+」・・・「1.2 <= version」
    - 「1.2\*」・・・「1.2 <= version < 1.3」
    - 「1.2[.2+]」・・・「1.2.2 <= version < 1.3」
-

# クラスのエクスポート

---

- モジュール外からアクセスできるクラスを絞り込むことができる。

```
class-exports=[x. y. z. ClassA, x. y. z. InterfaceB]
```

- エクスポート対象は、モジュールのメンバ内から

```
members=[x. y. z. ClassA, x. y. z. InterfaceB, x. y. z. ClassC]
```

---

# 拡張メタデータ

---

- さまざまなデータを格納できる
  - モジュールのバージョン番号
  - エクスポートするリソース(画像ファイルなど)
  - メインクラス

```
extensible-metadata=[@Version( "1.0" )]
```

---

## その他、JAMファイルのイケてるところ

---

- 「レガシー」JARファイルをそのまま格納できる。
    - 「MODULE-INF/lib」内に格納する
    - JAMへの移行中にはお世話になりそうな機能
    - 「レガシー」と呼ばれてしまうのが少しショック
  - ネイティブライブラリにもばっちり対応だ！
    - 「MODULE-INF/bin」内に格納する
    - 例：  
「MODULE-INF/bin/windows/i386/test.dll」
-

# リポジトリ

---

- 「JAMファイル置き場」=リポジトリ
  - モジュールのクラスをロード  
(ClassLoader.loadClass())する際、リポジトリからロードする
  - インターネット越しにJAMファイルをロードすることも可能
-

# JAM後の開発、配布はこうなる!

---

- 例: 同じ部署のAさんに、バッチファイルの作成を頼まれた。
  - モジュール名は「babatch」、バージョンは「1.0」とする。
  - 部署内の共有リポジトリ「<http://192.168.1.18>」がある。
-

# JAM後の開発、配布はこうなる!

---

- Hibernate3を使って作成完了。リポジトリにインストールした後、Aさんに「できたよ」と伝える。
- Aさんは、以下のコマンドを実行。

```
java -repository http://192.168.1.18 -module babatch
```

- すると、babatchモジュールのみならず、依存しているHibernate3、さらにHibernate3が依存するモジュールもすべてダウンロードされ、バッチが実行される！
-

# Java Module Systemのまとめ

---

- JAMファイルは、「JARファイル+モジュールメタデータ」という形式のファイル。
  - モジュールは名前やバージョン、依存関係などを持つ
  - モジュールはリポジトリから取り出される。リポジトリはネットワーク越しにあってもかまわない。
-

# これら二つの仕様の関係は？

---

- 「開発用モジュール」→スーパーパッケージ、「デプロイメントモジュール」→JAM、とされている。
  - が、現在までのところはおそらく、まだまだ仕様が確定していない。
  - ドキュメントで触れられているのは、「スーパーパッケージとモジュールメタデータは重複する情報が多いので、どちらかを自動生成しよう」ということだけ。
-

# スーパーパッケージから METADATA.moduleを生成する例

---

```
@MainClass("hello.Main")
@Version("1.0")
  @ImportModules({
    @ImportModule(name="java.se")
  })
superpackage hello {
  export hello.Main;
}
```



```
(
  name=hello
  extensible-
  metadata=[@Version("1.0"), @MainClass("hello.Main")]
  imports=[ImportModule(java.se)]
)
```

# まとめ

---

- スーパーパッケージは、Java言語を拡張して publicアクセスを詳細に制御する仕組み
  - Java Module Systemは、JAMフォーマットの導入により、「モジュール」と言うカプセル化／配布の単位を構築すると同時に、リポジトリによる利便性も提供する
  - これら二つは、密接に関連すると思われるが、まだ詳細が決まっていない
-

# 補足: JSR-291 (OSGi)との関連

---

- 「JSR-291 Dynamic Component Support for JavaSE」は、OSGi仕様による動的モジュラリティをJavaに追加する仕様
  - Java Module Systemと仕様が重なる部分も数多い。
  - OSGiは、Eclipseをはじめとしたさまざまなプロダクトで用いられており、すでに完成度は高い。
  - 両仕様の現在における状態・・・ケンカ中。。
-

# 補足: JSR-291とJSR-277の比較

ここに挙げたのは、差異の中でも代表的なものだけです。

	OSGi	JMS
配布形式	JAR	JAM
モジュラリティ	動的	静的
実現方法	Java上で実装	JVM組み込み
JavaMEのサポート	あり	なし
サポートするJavaのバージョン	1.3以上	7以上

# 補足: JSR-291 (OSGi)との関連

---

- 詳しいことを知りたい方は、Glyn Normingtonさんのブログを参照してください。  
<http://underlap.blogspot.com/2007/06/comparison-of-jsr-277-and-jsr-291.html>
  - OSGiとの関係は、はっきり言って「競合」。どちらが勝ったところで、開発者にとっては少しも得にならないので、仲良く仕様を統一してほしいです。
-

補足: Java7モジュラリティは、実際に試せます。

---

- OpenJDK7プロジェクトが、すでに成果物を発表済みです。
  - <http://openjdk.java.net/projects/modules/>
  - ただし、ソースからJDKをビルドしなくてはなりません。白石も試しましたが、今回は間に合いませんでした・・・
-

# 参考文献・URL

---

- Glyn Normingtonさんのブログ・・・  
<http://underlap.blogspot.com/search/label/JSR%20291>
  - InfoQ  
<http://www.infoq.com/jsr294>  
<http://www.infoq.com/jsr291>  
<http://www.infoq.com/jsr277>
  - JCP  
<http://jcp.org/en/jsr/detail?id=294>  
<http://jcp.org/en/jsr/detail?id=291>  
<http://jcp.org/en/jsr/detail?id=277>
  - OpenJDK (Module)  
<http://openjdk.java.net/projects/modules/#docs>
  - 先取りJava SE 7!  
<http://www.thinkit.co.jp/free/article/0708/9/1/>
  - Andreas Sterbenzさんのブログ  
[http://blogs.sun.com/andreas/entry/superpackages\\_in\\_jsr\\_294](http://blogs.sun.com/andreas/entry/superpackages_in_jsr_294)
  - Wikipedia (OSGi)  
<http://ja.wikipedia.org/wiki/OSGi>
  - Equinox  
<http://www.eclipse.org/equinox/documents/quickstart.php>
  - 各JSRの仕様書(JCPよりダウンロード可能)
-