<https://spring.pleiades.io/spring-boot/docs/current/reference/html/appendix-executable-jar-format.html>

# 実行可能な Jar 形式

spring-boot-loader モジュールにより、Spring Boot は実行可能な jar および war ファイルをサポートできます。Maven プラグインまたは Gradle プラグインを使用する場合、実行可能な jar は自動的に生成されるため、通常、それらがどのように機能するかの詳細を知る必要はありません。

別のビルドシステムから実行可能 jar を作成する必要がある場合、または基礎となるテクノロジに興味がある場合は、この付録で背景を説明します。

**1. ネストされた JAR**

Java は、ネストされた jar ファイル（つまり、jar 内に含まれる jar ファイル）をロードする標準的な方法を提供しません。これは、コマンドラインから展開せずに実行できる自己完結型アプリケーションを配布する必要がある場合に問題になる可能性があります。

この問題を解決するために、多くの開発者は「シェーディングされた」jar ファイルを使用します。シェーディングされた jar は、すべての jar からのすべてのクラスを単一の「uber jar」にパッケージ化します。シェーディングされた jar ファイルの問題は、どのライブラリが実際にアプリケーションに含まれているかを見にくくなることです。また、複数の jar で同じファイル名（ただし、異なるコンテンツ）が使用されている場合、問題が発生する可能性があります。Spring Boot は異なるアプローチを採用しており、実際に jar を直接ネストできます。

**1.1. 実行可能な Jar ファイル構造**

Spring Boot ローダー互換の jar ファイルは、次のように構成する必要があります。

example.jar

|

+-META-INF

| +-MANIFEST.MF

+-org

| +-springframework

| +-boot

| +-loader

| +-<spring boot loader classes>

+-BOOT-INF

+-classes

| +-mycompany

| +-project

| +-YourClasses.class

+-lib

+-dependency1.jar

+-dependency2.jar

アプリケーションクラスは、ネストされた BOOT-INF/classes ディレクトリに配置する必要があります。依存関係は、ネストされた BOOT-INF/lib ディレクトリに配置する必要があります。

**1.2. 実行可能な War ファイル構造**

Spring Boot ローダー互換の war ファイルは、次のように構成する必要があります。

example.war

|

+-META-INF

| +-MANIFEST.MF

+-org

| +-springframework

| +-boot

| +-loader

| +-<spring boot loader classes>

+-WEB-INF

+-classes

| +-com

| +-mycompany

| +-project

| +-YourClasses.class

+-lib

| +-dependency1.jar

| +-dependency2.jar

+-lib-provided

+-servlet-api.jar

+-dependency3.jar

依存関係は、ネストされた WEB-INF/lib ディレクトリに配置する必要があります。組み込みの実行時に必要であるが、従来の Web コンテナーにデプロイする場合には不要な依存関係は、WEB-INF/lib-provided に配置する必要があります。

**1.3. インデックスファイル**

Spring Boot ローダー互換の jar および war アーカイブには、BOOT-INF/ ディレクトリに追加の索引ファイルを含めることができます。classpath.idx ファイルは jar と war の両方に提供でき、jar をクラスパスに追加する順序を提供します。layers.idx ファイルは jar にのみ使用でき、jar を論理レイヤーに分割して Docker/OCI イメージを作成できます。

インデックスファイルは YAML 互換の構文に準拠しているため、サードパーティのツールで簡単に解析できます。ただし、これらのファイルは内部的に YAML として解析され*ない*ため、使用するには以下に説明する形式で正確に記述する必要があります。

**1.4. クラスパスインデックス**

クラスパスインデックスファイルは BOOT-INF/classpath.idx で提供できます。jar 名（ディレクトリを含まない）のリストを、クラスパスに追加する順序で提供します。各行はダッシュスペース（"-·"）で始まり、名前は二重引用符で囲む必要があります。

例：次の jar が与えられた場合：

example.jar

|

+-META-INF

| +-...

+-BOOT-INF

+-classes

| +...

+-lib

+-dependency1.jar

+-dependency2.jar

インデックスファイルは次のようになります。

- "dependency2.jar"

- "dependency1.jar"

**1.5. レイヤーインデックス**

レイヤーインデックスファイルは BOOT-INF/layers.idx で提供できます。レイヤーのリストと、レイヤー内に含まれる jar のパーツを提供します。レイヤーは、Docker/OCI イメージに追加する順序で書き込まれます。レイヤー名は、ダッシュスペース（"-·"）とコロン（":"）サフィックスが前に付いた引用符付き文字列として記述されます。レイヤーコンテンツは、スペーススペースダッシュスペース（"··-·"）を前に付けた引用符付き文字列として記述されたファイルまたはディレクトリ名です。ディレクトリ名は / で終わりますが、ファイル名は終わりません。ディレクトリ名が使用されている場合、そのディレクトリ内のすべてのファイルが同じレイヤーにあることを意味します。

レイヤーインデックスの典型的な例は次のとおりです。

- "dependencies":

- "BOOT-INF/lib/dependency1.jar"

- "BOOT-INF/lib/dependency2.jar"

- "application":

- "BOOT-INF/classes/"

- "META-INF/"

**2. Spring Boot の「JarFile」クラス**

ネストされた jar のロードをサポートするために使用されるコアクラスは org.springframework.boot.loader.jar.JarFile です。これにより、標準の jar ファイルまたはネストされた子 jar データから jar コンテンツをロードできます。最初にロードされるとき、各 JarEntry の場所は、次の例に示すように、外側の jar の物理ファイルオフセットにマップされます。

myapp.jar

+-------------------+-------------------------+

| /BOOT-INF/classes | /BOOT-INF/lib/mylib.jar |

|+-----------------+||+-----------+----------+|

|| A.class ||| B.class | C.class ||

|+-----------------+||+-----------+----------+|

+-------------------+-------------------------+

^ ^ ^

0063 3452 3980

上記の例は、0063 の myapp.jar の /BOOT-INF/classes で A.class を見つける方法を示しています。ネストされた jar からの B.class は、実際には myapp.jar の位置 3452 にあり、C.class は位置 3980 にあります。

この情報を使用して、外部 jar の適切な部分をシークすることにより、特定のネストされたエントリをロードできます。アーカイブを解凍する必要はなく、すべてのエントリデータをメモリに読み込む必要もありません。

**2.1. 標準 Java「JarFile」との互換性**

Spring Boot Loader は、既存のコードおよびライブラリとの互換性を維持するよう努めています。org.springframework.boot.loader.jar.JarFile は java.util.jar.JarFile から拡張されており、ドロップイン置換として機能するはずです。getURL() メソッドは、java.net.JarURLConnection と互換性のある接続を開き、Java の URLClassLoader で使用できる URL を返します。

**3. 実行可能ファイルの起動**

org.springframework.boot.loader.Launcher クラスは、実行可能な jar のメインエントリポイントとして使用される特別なブートストラップクラスです。これは、jar ファイルの実際の Main-Class であり、適切な URLClassLoader をセットアップし、最終的に main() メソッドを呼び出すために使用されます。

3 つのランチャーサブクラス（JarLauncher、WarLauncher および PropertiesLauncher）があります。それらの目的は、ディレクトリ内のネストされた jar ファイルまたは war ファイルからリソース（.class ファイルなど）をロードすることです（明示的にクラスパスにあるファイルとは対照的です）。JarLauncher および WarLauncher の場合、ネストされたパスは固定されています。JarLauncher は BOOT-INF/lib/ を検索し、WarLauncher は WEB-INF/lib/ と WEB-INF/lib-provided/ を検索します。さらに必要な場合は、これらの場所に追加の jar を追加できます。PropertiesLauncher は、デフォルトでアプリケーションアーカイブの BOOT-INF/lib/ を検索します。loader.properties （ディレクトリ、アーカイブ、またはアーカイブ内のディレクトリのコンマ区切りリスト）で LOADER\_PATH または loader.path と呼ばれる環境変数を設定することにより、追加の場所を追加できます。

**3.1. ランチャーマニフェスト**

META-INF/MANIFEST.MF の Main-Class 属性として適切な Launcher を指定する必要があります。起動する実際のクラス（つまり、main メソッドを含むクラス）は、Start-Class 属性で指定する必要があります。

次の例は、実行可能な jar ファイルの一般的な MANIFEST.MF を示しています。

Main-Class: org.springframework.boot.loader.JarLauncher

Start-Class: com.mycompany.project.MyApplication

war ファイルの場合、次のようになります。

Main-Class: org.springframework.boot.loader.WarLauncher

Start-Class: com.mycompany.project.MyApplication

|  |  |
| --- | --- |
|  | マニフェストファイルで Class-Path エントリを指定する必要はありません。クラスパスは、ネストされた jar から推定されます。 |

**4. PropertiesLauncher の機能**

PropertiesLauncher には、外部プロパティ（システムプロパティ、環境変数、マニフェストエントリ、または loader.properties）で有効にできる特別な機能がいくつかあります。次の表で、これらのプロパティについて説明します。

| **キー** | **目的** |
| --- | --- |
| loader.path | lib,${HOME}/app/lib などのコンマ区切りのクラスパス。javac コマンドラインの通常の -classpath のように、以前のエントリが優先されます。 |
| loader.home | loader.path の相対パスを解決するために使用されます。例： loader.path=lib を指定すると、${loader.home}/lib はクラスパスの場所です（そのディレクトリ内のすべての jar ファイルとともに）。このプロパティは、次の例 [/opt/app](file:///\\opt\app) のように、loader.properties ファイルを見つけるためにも使用されます。デフォルトは ${user.dir} です。 |
| loader.args | main メソッドのデフォルト引数（スペースで区切られます）。 |
| loader.main | 起動するメインクラスの名前（例： com.app.Application）。 |
| loader.config.name | プロパティファイルの名前（例： launcher）。デフォルトは loader です。 |
| loader.config.location | プロパティファイルへのパス（例： classpath:loader.properties）。デフォルトは loader.properties です。 |
| loader.system | すべてのプロパティをシステムプロパティに追加する必要があることを示すブールフラグ。デフォルトは false です。 |

環境変数またはマニフェストエントリとして指定する場合、次の名前を使用する必要があります。

| **キー** | **マニフェストエントリ** | **環境変数** |
| --- | --- | --- |
| loader.path | Loader-Path | LOADER\_PATH |
| loader.home | Loader-Home | LOADER\_HOME |
| loader.args | Loader-Args | LOADER\_ARGS |
| loader.main | Start-Class | LOADER\_MAIN |
| loader.config.location | Loader-Config-Location | LOADER\_CONFIG\_LOCATION |
| loader.system | Loader-System | LOADER\_SYSTEM |

|  |  |
| --- | --- |
|  | ビルドプラグインは、fat jar がビルドされると、Main-Class 属性を Start-Class に自動的に移動します。それを使用する場合は、Main-Class 属性を使用して Start-Class を除外することにより、起動するクラスの名前を指定します。 |

PropertiesLauncher の操作には、次の規則が適用されます。

* loader.properties は loader.home で検索され、次にクラスパスのルートで検索され、次に classpath:/BOOT-INF/classes で検索されます。その名前のファイルが存在する最初の場所が使用されます。
* loader.home は、loader.config.location が指定されていない場合のみ、追加のプロパティファイルのディレクトリの場所です（デフォルトを上書きします）。
* loader.path には、ディレクトリ（jar および zip ファイルを再帰的にスキャンする）、アーカイブパス、jar ファイルをスキャンするアーカイブ内のディレクトリ（たとえば dependencies.jar!/lib）、またはワイルドカードパターン（デフォルトの JVM 動作の場合）を含めることができます。アーカイブパスは、loader.home または jar:file: プレフィックスを持つファイルシステム内の任意の場所からの相対パスにすることができます。
* loader.path （空の場合）は、デフォルトで BOOT-INF/lib （ローカルディレクトリまたはアーカイブから実行する場合はネストされたディレクトリを意味します）になります。このため、追加の構成が提供されていない場合、PropertiesLauncher は JarLauncher と同じように動作します。
* loader.path を使用して loader.properties の場所を構成することはできません（loader.properties の検索に使用されるクラスパスは、PropertiesLauncher の起動時の JVM クラスパスです）。
* プレースホルダーの置換は、システムおよび環境変数に加えて、使用前にすべての値のプロパティファイル自体から行われます。
* プロパティの検索順序（複数の場所を調べるのが理にかなっている場合）は、環境変数、システムプロパティ、loader.properties、展開されたアーカイブマニフェスト、およびアーカイブマニフェストです。

**5. 実行可能な Jar の制限**

Spring Boot Loader パッケージアプリケーションを使用するときは、次の制限を考慮する必要があります。

* Zip エントリの圧縮: ネストされた jar の ZipEntry は、ZipEntry.STORED メソッドを使用して保存する必要があります。これは、ネストされた jar 内の個々のコンテンツを直接検索できるようにするために必要です。ネストされた jar ファイル自体のコンテンツは、外部 jar の他のエントリと同様に、圧縮することができます。
* システムクラスローダー: 起動されたアプリケーションは、クラスをロードするときに Thread.getContextClassLoader() を使用する必要があります（ほとんどのライブラリとフレームワークはデフォルトでそうします）。ClassLoader.getSystemClassLoader() でネストされた jar クラスをロードしようとすると失敗します。java.util.Logging は常にシステムクラスローダーを使用します。このため、異なるロギング実装を検討する必要があります。

**6. 代替シングル Jar ソリューション**

上記の制限により Spring Boot ローダーを使用できないことを意味する場合、以下の代替案を検討してください。

* [Maven Shade プラグイン : Apache (英語)](https://maven.apache.org/plugins/maven-shade-plugin/)
* [JarClassLoader (英語)](http://www.jdotsoft.com/JarClassLoader.php)
* [OneJar (英語)](https://sourceforge.net/projects/one-jar/)
* [Gradle Shadow プラグイン (英語)](https://imperceptiblethoughts.com/shadow/)