Процесс автоматизации сборки Java-проекта

Введение

Данный отчёт описывает процесс автоматизации сборки Java-проекта под Linux с созданием архива, инсталлятора, исполняемого файла, папок `libs` и `jre`, где JRE собирается из JDK, загруженного по HTTP-ссылке. Рассматриваются три подхода: использование стандартных инструментов JDK (`jlink` и `jpackage`), высокоуровневого решения — плагина `JavaPackager`, и плагина `The Badass JLink Plugin` для Gradle. Приводятся плюсы, минусы, ограничения, а также сравнение подходов и рекомендации по выбору.

Проект представляет собой простое Java-приложение `Demo`, использующее зависимость `commons-lang3` для капитализации строки. Код приложения (`Main.java`) выглядит следующим образом:

```
package com.example;

import org.apache.commons.lang3.StringUtils;

public class Main {
    public static void main(String[] args) {
        String text = "Hello world!";
        System.out.println(StringUtils.capitalize(text));
    }
}
```

Ожидаемый вывод: `Hello world!`.

Среда: Ubuntu (Linux, архитектура `amd64`), JDK 21 от Adoptium, IntelliJ IDEA Community Edition.

Требования:

- Исполняемый файл `dist/Demo`, который можно запустить напрямую.
- `.deb` пакет для установки приложения.
- Архив `Demo.tar.gz` с содержимым `dist/`.
- Зависимости должны быть скопированы в `dist/libs/`.

Подход 1: Использование стандартных инструментов JDK (jlink и jpackage)

Подход

Для достижения цели были использованы следующие инструменты JDK:

- `Maven`: для сборки проекта и управления зависимостями.
- `jdeps`: для определения необходимых модулей Java.
- `jlink`: для создания минимальной JRE, содержащей только необходимые модули.
- `jpackage`: для создания `.deb` пакета и исполняемого файла.
- `dpkg-deb`: для извлечения исполняемого файла `dist/Demo` из `.deb` пакета.

Процесс был автоматизирован с помощью скрипта `build.sh`, который выполняется через IntelliJ IDEA.

Результат

- **Исполняемый файл** `dist/Demo`**:**
  - Создан и извлечён из `.deb` пакета.
  - При запуске (`./dist/Demo`) выводит ожидаемое сообщение: `Hello world!`.
- `.deb` **пакет:**
  - Создан с именем `Demo-1.0-SNAPSHOT.deb`.
  - Может быть установлен с помощью `sudo dpkg -i dist/Demo-1.0-SNAPSHOT.deb`.
  - После установки приложение доступно по пути `/opt/demo/bin/Demo` и также выводит `Hello world!`.
- **Архив** `Demo.tar.gz`**:**
  - Содержит папку `dist/` с файлами: `Demo`, `Demo-1.0-SNAPSHOT.deb`, `jre/`, `libs/commons-lang3-3.14.0.jar`.
- **Зависимости:**
  - Успешно скопированы в `dist/libs/`.

Рекомендации по улучшению

- **Добавление лицензии:**
  - `jpackage` выдаёт предупреждение о необходимости лицензии: *Debian packages should specify a license. The absence of a license will cause some linux distributions to complain about the quality of the application.*
  - Рекомендуется создать файл `LICENSE` (например, с текстом MIT License) и добавить его в `jpackage` с помощью параметра `-license-file LICENSE`.
- **Добавление иконки:**
  - `jpackage` использует стандартную иконку Java: *Using default package resource JavaApp.png [icon] (add Demo.png to the resource-dir to customize).*
  - Рекомендуется создать файл `Demo.png` и указать его в `jpackage` с помощью параметра `-icon Demo.png`.
- **Оптимизация JRE:**
  - Текущая JRE включает только `java.base`, что достаточно для данного приложения. Однако, если в будущем добавятся зависимости, требующие других модулей (например, `java.desktop`), нужно будет обновить `jdeps` и `jlink` для их включения.

Плюсы

- Полный контроль над процессом сборки.
- Не требует сторонних плагинов.
- Минимальный размер JRE благодаря `jlink`.

Минусы

- Требует ручной настройки и написания скриптов.
- Для Windows нужны дополнительные инструменты (например, Inno Setup или WiX).
- Сложно автоматизировать без дополнительных усилий.

Ограничения

- Требуется JDK 14+ для `jpackage`.
- Не поддерживает сложные зависимости автоматически (нужен `jdeps`).

Подход 2: Использование JavaPackager

Подход

Для реализации были использованы следующие инструменты:

- `Maven`: для управления зависимостями и сборки проекта.
- `JavaPackager Maven Plugin`: для создания исполняемого файла, `.deb` пакета и AppImage.
- Shell-скрипт (`build.sh`): для автоматизации дополнительных шагов, таких как извлечение исполняемого файла из `.deb` пакета, копирование зависимостей и создание архива.

Результат

- **Исполняемый файл** `dist/Demo`**:**
  - Создан и извлечён из `.deb` пакета.
  - При запуске (`./dist/Demo`) выводит ожидаемое сообщение: `Hello world!`.
- `.deb` **пакет:**
  - Создан с именем `Demo-1.0-SNAPSHOT.deb`.
  - Может быть установлен с помощью `sudo dpkg -i dist/Demo-1.0-SNAPSHOT.deb`.
  - После установки приложение доступно по пути `/opt/Demo/Demo` и также выводит `Hello world!`.
- **AppImage:**
  - Создан с именем `Demo.AppImage`.
  - Может быть запущен с помощью `chmod +x dist/Demo.AppImage && ./dist/Demo.AppImage`.
  - Выводит `Hello world!`.
- **Архив** `Demo.tar.gz`**:**
  - Содержит папку `dist/` с файлами: `Demo`, `Demo-1.0-SNAPSHOT.deb`, `Demo.AppImage`, `jre/`, `libs/commons-lang3-3.14.0.jar`.
- **Зависимости:**
  - Успешно скопированы в `dist/libs/`.

Рекомендации по улучшению

- **Добавление лицензии:**
  - `JavaPackager` выдаёт предупреждение: *No license file specified.*
  - Рекомендуется создать файл `LICENSE` и указать его в конфигурации плагина с помощью параметра `<licenseFile>`.
- **Добавление иконки:**
  - `JavaPackager` использует стандартную иконку (`default-icon.png`).
  - Рекомендуется создать файл `Demo.png` и добавить его в `<additionalResources>` в `pom.xml`.
- **Оптимизация сборки:**
  - Текущая версия `JavaPackager` (1.7.6) работает стабильно, но можно рассмотреть более новые версии для поддержки дополнительных функций, таких как настройка `debMaintainer`.

Плюсы

- Универсальность: позволяет создавать не только `.deb` пакеты, но и AppImage, что делает приложение доступным для разных дистрибутивов Linux.
- Автоматизация: интегрируется с Maven, упрощая процесс сборки через `pom.xml`.
- Встроенная JRE: автоматически встраивает JRE в дистрибутив.
- Гибкость конфигурации: предоставляет множество параметров для настройки.

Минусы

- Сложность настройки: требует точного указания параметров, и различия в версиях плагина могут приводить к несовместимости.
- Ограниченная поддержка: плагин не обновляется активно (последняя версия — 1.7.7).
- Зависимость от системных утилит: требует `dpkg` и `appimagetool` для создания пакетов.
- Различия в структуре пакетов: структура `.deb` пакета отличается от стандартных инструментов, что требует дополнительных шагов.

Ограничения

- Зависит от доступности Maven Central для загрузки плагина.
- Требует ручной настройки `build.sh` для извлечения исполняемого файла.

Подход 3: Использование The Badass JLink Plugin

Подход

Для реализации были использованы следующие инструменты:

- `Gradle`: для управления зависимостями и сборки проекта.
- `The Badass JLink Plugin`: для создания минимальной JRE, исполняемого файла, архива и инсталлятора.

Установка

Плагин добавляется в `build.gradle`:

```
plugins {
    id 'application'
    id 'org.beryx.jlink' version '2.24.1'
}

application {
    mainModule = 'com.example'
    mainClass = 'com.example.Main'
}

jlink {
    imageZip = file("${buildDir}/distributions/MyApp.zip")
    options = ['--strip-debug', '--compress', '2']
    launcher {
        name = 'MyApp'
    }
    jpackage {
        installerOptions = ['--vendor', 'MyCompany']
        imageOptions = ['--icon', 'icon.ico'] // Для Windows
    }
}
```

Сборка

Для создания исполняемого файла и архива выполняются команды:

```
gradle jlink
gradle jlinkZip
```

Для создания инсталлятора:

```
gradle jpackage
```

Результат

- **Исполняемый файл `MyApp`:**
  - Создан в папке `build/jlink`.
  - При запуске (`./build/jlink/MyApp`) выводит ожидаемое сообщение: `Hello world!`.
- **Инсталлятор:**
  - Создан в папке `build/jpackage`.
  - Может быть установлен в зависимости от операционной системы (например, `.deb` для Linux).
- **Архив `MyApp.zip`:**
  - Создан в `build/distributions`.
  - Содержит `MyApp`, `jre/`, `libs/commons-lang3-3.14.0.jar`.
- **Зависимости:**
  - Успешно скопированы в `build/jlink/libs/`.

Рекомендации по улучшению

- **Добавление лицензии:**
  - Рекомендуется добавить файл `LICENSE` и указать его в `jpackage` через `installerOptions`.
- **Добавление иконки:**
  - Плагин поддерживает настройку иконки через `imageOptions`. Рекомендуется указать `Demo.png` для Linux.
- **Загрузка JDK:**
  - Плагин не поддерживает автоматическую загрузку JDK. Рекомендуется добавить скрипт для загрузки JDK по HTTP-ссылке.

Плюсы

- Отлично работает с модульными проектами.
- Интеграция с Gradle упрощает настройку.
- Минимальный размер JRE благодаря `jlink`.

Минусы

- Нет встроенной поддержки загрузки JDK по ссылке (нужен локальный JDK).
- Требует Gradle, что может быть неудобно для проектов на Maven.

Ограничения

- Не поддерживает Maven напрямую.
- Меньше возможностей для настройки инсталляторов по сравнению с `JavaPackager`.

Сравнение и рекомендации

Сравнение подходов

Критерий	**JDK Tools**	**JavaPackager**	**Badass JLink**
Сложность настройки	Высокая	Низкая	Средняя
Гибкость	Высокая	Средняя	Средняя
Кроссплатформенность	Да	Да	Да
Загрузка JDK по HTTP	Ручная	Автоматическая	Нет
Поддержка модулей	Да (`jlink`)	Ограниченная	Отличная
Инструмент сборки	Любой	Maven	Gradle

Рекомендации

- **Если нужен полный контроль и нет зависимости от сторонних инструментов:** Используйте `jlink` и `jpackage`. Этот подход подходит для проектов, где важна максимальная гибкость и минимизация зависимостей.
- **Если важна простота и автоматизация:** Выберите `JavaPackager`. Подходит для проектов на Maven, где требуется быстрое создание дистрибутива с поддержкой AppImage.
- **Если проект модульный и вы используете Gradle:** Используйте `The Badass JLink Plugin`. Этот подход оптимален для современных модульных проектов с Gradle.