Gradle User Guide

Version 2.2-20140924021627+0000

Translated by

Hayashi Masatoshi [FAMILY Given]
Sekiya Kazuchika [FAMILY Given]
Sue Nobuhiro [FAMILY Given]
Mochida Shinya [FAMILY Given]

製作著作 © 2007-2012 Hans Dockter, Adam Murdoch
Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.
このドキュメントは、個人利用目的および第三者に配布するためにコピーして使用できます。ただし、印刷

目次

翻訳版について

1. はじめに
    1.1. このユーザーガイドについて

2. 概要
    2.1. 特長
    2.2. なぜGroovyなのか?

3. チュートリアル
    3.1. 始めてみよう

4. Gradleのインストール
    4.1. 必要環境
    4.2. ダウンロード
    4.3. 解凍
    4.4. 環境変数
    4.5. 正しくインストールされたことをテストする
    4.6. JVMオプション

5. トラブルシューティング
    5.1. トラブルに対処する
    5.2. ヘルプを求めるには

6. ビルドスクリプトの基本
    6.1. プロジェクトとタスク
    6.2. Hello world
    6.3. タスク定義のショートカット
    6.4. ビルドスクリプトはコードです
    6.5. タスクの依存関係
    6.6. 動的なタスク
    6.7. 既存のタスクを操作する
    6.8. 略記法
    6.9. 拡張タスクプロパティ
    6.10. Antタスクの使用
    6.11. メソッドの使用
    6.12. デフォルトタスク
    6.13. DAGによる設定
    6.14. 次のステップは？

7. Javaクイックスタート
    7.1. Javaプラグイン
    7.2. 基本的なJavaプロジェクト
    7.3. マルチプロジェクトのJavaビルド
    7.4. 次のステップは？

8. 依存関係管理の基本
    8.1. 依存関係の管理とは
    8.2. 依存関係の宣言
    8.3. 依存関係のコンフィグレーション
    8.4. 外部依存関係
    8.5. リポジトリ
    8.6. アーティファクトの公開
    8.7. 次のステップは？

9. Groovyクイックスタート
- 9.1. 基本的なGroovyプロジェクト
- 9.2. まとめ

10. Webアプリケーションクイックスタート
- 10.1. WARファイルのビルド
- 10.2. Webアプリケーションの実行
- 10.3. まとめ

11. Gradleのコマンドラインを使う
- 11.1. 複数のタスクを実行する
- 11.2. タスクを除外してビルドする
- 11.3. エラー発生時にビルドを継続する
- 11.4. タスク名の省略
- 11.5. ビルドスクリプトを指定して実行する
- 11.6. ビルドに関する情報を取得する
- 11.7. 空実行
- 11.8. まとめ

12. GradleのGUIを使う
- 12.1. Task Tree（タスク・ツリー）
- 12.2. Favorites（お気に入り）
- 12.3. Command Line（コマンドライン）
- 12.4. Setup（セットアップ）

13. ビルドスクリプトの記述
- 13.1. Gradleビルド言語
- 13.2. プロジェクトAPI
- 13.3. スクリプトAPI
- 13.4. 変数の宣言
- 13.5. Groovyの基本

14. 色々なチュートリアル
- 14.1. ディレクトリの作成
- 14.2. Gradleプロパティとシステムプロパティ
- 14.3. 外部のビルドスクリプトをプロジェクトに取り込む
- 14.4. 任意のオブジェクトを組み立てる
- 14.5. 外部スクリプトで任意のオブジェクトを組み立てる
- 14.6. キャッシング

15. タスク詳解
- 15.1. タスクの定義
- 15.2. タスクを配置する
- 15.3. タスクの設定を変更する
- 15.4. タスクに依存関係を追加する
- 15.5. Ordering tasks
- 15.6. タスクに説明書きを追加する
- 15.7. タスクの置き換え
- 15.8. タスクをスキップする
- 15.9. 更新されていないタスクをスキップする
- 15.10. タスクルール
- 15.11. Finalizer tasks
- 15.12. まとめ

16. ファイルを取り扱う
- 16.1. ファイルを参照する
- 16.2. ファイルコレクション

16.3. ファイルツリー
16.4. アーカイブの内容をファイルツリーとして使う
16.5. 入力ファイルセットを指定する
16.6. ファイルをコピーする
16.7. Syncタスクを使う
16.8. アーカイブを作成する

17. GradleからAntを使う
17.1. ビルドでのAntタスクとタイプの利用
17.2. Antビルドのインポート
17.3. Antプロパティとリファレンス
17.4. API

18. ロギング
18.1. ログレベルの選択
18.2. ログメッセージを書く
18.3. 外部ツールやライブラリからのログについて
18.4. Gradleがロギングするものを変更する

19. Gradleデーモン
19.1. デーモン入門
19.2. デーモンの再利用と期限切れ
19.3. 使い方とトラブルシューティング
19.4. デーモンの設定

20. ビルド環境
20.1. gradle.propertiesを使用したビルド環境の構築
20.2. プロクシ経由のWebアクセス

21. Gradleのプラグインについて
21.1. Types of plugins
21.2. プラグインの適用
21.3. Applying plugins with the plugins DSL
21.4. Finding community plugins
21.5. プラグインがすること
21.6. 規約
21.7. プラグインをさらに詳しく知るには

22. 標準Gradleプラグイン
22.1. 言語プラグイン
22.2. 試験的な言語プラグイン
22.3. 統合プラグイン
22.4. 試験的な統合プラグイン
22.5. ソフトウェア開発プラグイン
22.6. 試験的なソフトウェア開発プラグイン
22.7. ベースプラグイン
22.8. サードパーティプラグイン

23. Javaプラグイン
23.1. 使用方法
23.2. ソースセット
23.3. タスク
23.4. プロジェクトレイアウト
23.5. 依存関係の管理
23.6. 規約プロパティ
23.7. ソースセットの利用
23.8. Javadoc

23.9. Clean
23.10. リソース
23.11. CompileJava
23.12. Incremental Java compilation
23.13. テスト
23.14. Jar
23.15. アップロード

24. Groovyプラグイン
24.1. 使用方法
24.2. タスク
24.3. プロジェクトレイアウト
24.4. 依存関係の管理
24.5. Groovyクラスパスの自動設定
24.6. 規約プロパティ
24.7. ソースセットプロパティ
24.8. GroovyCompile

25. Scalaプラグイン
25.1. 使用方法
25.2. タスク
25.3. プロジェクトレイアウト
25.4. 依存関係の管理
25.5. scalaClasspathの自動設定
25.6. 規約プロパティ
25.7. ソースセットプロパティ
25.8. Fast Scala Compiler
25.9. 別プロセスでコンパイルする
25.10. インクリメンタルコンパイル
25.11. Eclipse Integration
25.12. IntelliJ IDEA Integration

26. War プラグイン
26.1. 使用方法
26.2. タスク
26.3. プロジェクトレイアウト
26.4. 依存関係の管理
26.5. 規約プロパティ
26.6. War
26.7. カスタマイズ

27. Earプラグイン
27.1. 使用方法
27.2. タスク
27.3. プロジェクトレイアウト
27.4. 依存関係の管理
27.5. 規約プロパティ
27.6. Ear
27.7. カスタマイズ
27.8. カスタムのディスクリプタファイルを使う

28. Jetty プラグイン
28.1. 使用方法
28.2. タスク
28.3. プロジェクトレイアウト
28.4. 依存関係の管理

28.5. 規約プロパティ

29. Checkstyleプラグイン
29.1. 使用方法
29.2. タスク
29.3. プロジェクトレイアウト
29.4. 依存関係の管理
29.5. 設定

30. CodeNarcプラグイン
30.1. 使用方法
30.2. タスク
30.3. プロジェクトレイアウト
30.4. 依存関係の管理
30.5. 設定

31. FindBugsプラグイン
31.1. 使用方法
31.2. タスク
31.3. 依存関係の管理
31.4. 設定

32. JDependプラグイン
32.1. 使用方法
32.2. タスク
32.3. 依存関係の管理
32.4. 設定

33. PMDプラグイン
33.1. 使用方法
33.2. タスク
33.3. 依存関係の管理
33.4. 設定

34. The JaCoCo Plugin
34.1. Getting Started
34.2. Configuring the JaCoCo Plugin
34.3. JaCoCo Report configuration
34.4. JaCoCo specific task configuration
34.5. Tasks
34.6. Dependency management

35. Sonarプラグイン
35.1. 使用方法
35.2. マルチプロジェクトビルドの解析
35.3. カスタムソースセットの解析
35.4. Java言語以外の解析
35.5. カスタムSonarプロパティの設定
35.6. コマンドラインでSonarの設定を行う
35.7. タスク

36. The Sonar Runner Plugin
36.1. Sonar Runner version and compatibility
36.2. Getting started
36.3. Configuring the Sonar Runner
36.4. Specifying the Sonar Runner version
36.5. Analyzing Multi-Project Builds

36.6. Analyzing Custom Source Sets
36.7. Analyzing languages other than Java
36.8. More on configuring Sonar properties
36.9. Setting Sonar Properties from the Command Line
36.10. Controlling the Sonar Runner process
36.11. Tasks

37. OSGiプラグイン
37.1. 使用方法
37.2. 暗黙的に適用されるプラグイン
37.3. タスク
37.4. 依存関係の管理
37.5. 規約オブジェクト
37.6.

38. Eclipse プラグイン
38.1. 使用方法
38.2. タスク
38.3. 設定
38.4. 生成されたファイルをカスタマイズする

39. IDEAプラグイン
39.1. 使用方法
39.2. タスク
39.3. 設定
39.4. 生成するファイルのカスタマイズ
39.5. その他の注意事項

40. ANTLRプラグイン
40.1. 使用方法
40.2. タスク
40.3. プロジェクトレイアウト
40.4. 依存関係管理
40.5. 規約プロパティ
40.6. ソースセットプロパティ

41. プロジェクトレポートプラグイン
41.1. 使用法
41.2. タスク
41.3. プロジェクトレイアウト
41.4. 依存関係
41.5. 規約プロパティ

42. 通知プラグイン
42.1. 使用方法
42.2. 設定

43. ビルド通知プラグイン
43.1. 使用方法

44. The Distribution Plugin
44.1. Usage
44.2. Tasks
44.3. Distribution contents

45. アプリケーション プラグイン
45.1. 使用方法
45.2. タスク

45.3. 規約プロパティ
45.4. ディストリビューションに他のリソースを含める

46. The Java Library Distribution Plugin
46.1. Usage
46.2. Tasks
46.3. Including other resources in the distribution

47. Build Init Plugin
47.1. Tasks
47.2. What to set up
47.3. Build init types

48. Wrapper Plugin
48.1. Usage
48.2. Tasks

49. The Build Dashboard Plugin
49.1. Usage
49.2. Tasks
49.3. Project layout
49.4. Dependency management
49.5. Configuration

50. The Java Gradle Plugin Development Plugin
50.1. Usage

51. 依存関係の管理
51.1. はじめに
51.2. 依存関係管理のベストプラクティス
51.3. 依存関係のコンフィギュレーション
51.4. 依存関係の定義方法
51.5. 依存関係を使った作業
51.6. リポジトリ
51.7. 依存関係解決の仕組み
51.8. 依存関係解決処理の微調整
51.9. 依存関係のキャッシュ
51.10. 推移的依存関係を管理するための戦略

52. アーティファクトの公開
52.1. はじめに
52.2. アーティファクトとコンフィギュレーション
52.3. アーティファクトの宣言
52.4. アーティファクトの公開
52.5. プロジェクトライブラリについての追記事項

53. Mavenプラグイン
53.1. 使用方法
53.2. タスク
53.3. 依存関係管理
53.4. 規約プロパティ
53.5. 規約メソッド
53.6. Mavenリポジトリとの相互作用

54. 署名プラグイン
54.1. 使用方法
54.2. 署名者の資格情報
54.3. 署名対象を指定する

54.4. 署名を公開する
54.5. POMファイルに署名する

55. Building native binaries
55.1. Supported languages
55.2. Tool chain support
55.3. Tool chain installation
55.4. Component model
55.5. Building a library
55.6. Building an executable
55.7. Tasks
55.8. Finding out more about your project
55.9. Language support
55.10. Configuring the compiler, assembler and linker
55.11. Windows Resources
55.12. Library Dependencies
55.13. Native Binary Variants
55.14. Tool chains
55.15. Visual Studio IDE integration
55.16. CUnit support

56. ビルドのライフサイクル
56.1. ビルドフェーズ
56.2. 設定ファイル
56.3. マルチプロジェクトのビルド
56.4. 初期化
56.5. シングルプロジェクトの設定と実行
56.6. ライフサイクルからの通知に応答する

57. マルチプロジェクトのビルド
57.1. クロスプロジェクト設定
57.2. サブプロジェクトの設定
57.3. マルチプロジェクトのビルド実行ルール
57.4. 絶対パスによるタスクの実行
57.5. プロジェクトとタスクのパス
57.6. 依存関係 - なんの依存関係？
57.7. プロジェクト依存関係
57.8. Parallel project execution
57.9. 分離されたプロジェクト
57.10. マルチプロジェクトのビルドとテスティング
57.11. Multi Project and buildSrc
57.12. プロパティとメソッドの継承
57.13. まとめ

58. カスタムタスクの作成
58.1. タスククラスのパッケージング
58.2. 単純タスクの作成
58.3. スタンドアロンプロジェクト
58.4. Incremental tasks

59. カスタムプラグインの作成
59.1. プラグインのパッケージング
59.2. シンプルなプラグインの作成
59.3. ビルドから入力を得る
59.4. カスタムタスクやプラグインでファイルを扱う
59.5. スタンドアロンプロジェクト

59.6. 複数のドメインオブジェクトの管理

60. ビルドロジックの体系化
60.1. プロパティとメソッドの継承
60.2. 設定のインジェクション
60.3. buildSrcプロジェクトのソースをビルドする
60.4. 別のGradleビルドを、現在のビルドから呼び出して実行する
60.5. ビルドスクリプトで外部ライブラリを使うときの依存関係設定
60.6. Antオプショナルタスクの依存関係
60.7. まとめ

61. 初期化スクリプト
61.1. 基本的な使い方
61.2. 初期化スクリプトを使う
61.3. 初期化スクリプトを記述する
61.4. 初期化スクリプトの外部依存関係
61.5. Init script plugins

62. Gradleラッパー
62.1. 設定
62.2. Unixファイルパーミッション

63. Embedding Gradle
63.1. Introduction to the Tooling API
63.2. Tooling API and the Gradle Build Daemon
63.3. Quickstart

64. Comparing Builds
64.1. Definition of terms
64.2. Current Capabilities
64.3. Comparing Gradle Builds

65. Ivy Publishing (new)
65.1. The “ivy-publish” Plugin
65.2. Publications
65.3. Repositories
65.4. Performing a publish
65.5. Generating the Ivy module descriptor file without publishing
65.6. Complete example
65.7. Future features

66. Maven Publishing (new)
66.1. The “maven-publish” Plugin
66.2. Publications
66.3. Repositories
66.4. Performing a publish
66.5. Publishing to Maven Local
66.6. Generating the POM file without publishing

A. Gradleサンプル集
A.1. サンプル customBuildLanguage
A.2. サンプル customDistribution
A.3. サンプル customPlugin
A.4. サンプル java/multiproject

B. 陥りがちな罠
B.1. Groovyスクリプトの変数
B.2. 設定フェーズと実行フェーズ

C. 機能のライフサイクル
C.1. 状態
C.2. 後方互換性ポリシー

D. Gradle コマンドライン
D.1. デーモン コマンドラインオプション:
D.2. システムプロパティ
D.3. 環境変数

E. IDE対応の現状と、IDEによらない開発支援
E.1. IntelliJ
E.2. Eclipse
E.3. IDEサポートなしでGradleを使う

用語集 / Glossary

例目次

6.1. 初めてのビルドスクリプト
6.2. ビルドスクリプトの実行
6.3. タスク定義のショートカット
6.4. GradleタスクでGroovyを使う
6.5. GradleタスクでGroovyを使う
6.6. タスク間の依存関係を宣言する
6.7. 遅延評価のdependsOn - タスクがまだ宣言されていない場合
6.8. 動的なタスク定義
6.9. APIからタスクにアクセスする - 依存関係の追加
6.10. APIからタスクにアクセスする - アクションの追加
6.11. ビルドスクリプトのプロパティとして既存のタスクにアクセスする
6.12. 拡張プロパティをタスクに追加する
6.13. AntBuilderを使ってant.loadfileターゲットを実行する
6.14. メソッドを抽出してビルドロジックを整理する
6.15. デフォルトタスクの定義
6.16. 選択したタスクによって異なる結果を得る
7.1. Javaプラグインの使用
7.2. Javaプロジェクトのビルド
7.3. Mavenリポジトリの追加
7.4. 依存関係の追加
7.5. MANIFEST.MFのカスタマイズ
7.6. テスト用システムプロパティの追加
7.7. JARファイルの公開
7.8. Eclipseプラグイン
7.9. Javaの例 - 完全なビルドファイル
7.10. マルチプロジェクトビルド - 階層レイアウト
7.11. マルチプロジェクトビルド - settings.gradleファイル
7.12. マルチプロジェクトビルド - 共通設定
7.13. マルチプロジェクトビルド - プロジェクト間の依存関係
7.14. マルチプロジェクトビルド - 配布ファイル
8.1. 依存関係の宣言
8.2. 外部依存関係の定義
8.3. 外部依存関係定義のショートカット形式

8.4. Mavenセントラルリポジトリの使用
8.5. リモートMavenリポジトリの使用
8.6. リモートIvyリポジトリの使用
8.7. ローカルのIvyディレクトリを使う
8.8. Ivyリポジトリに公開する
8.9. Mavenリポジトリへの公開
9.1. Groovyプラグイン
9.2. Dependency on Groovy
9.3. Groovy用のビルドファイル（全体）
10.1. Warプラグイン
10.2. JettyプラグインによるWebアプリケーションの実行
11.1. 複数のタスクの実行
11.2. タスクの除外
11.3. タスク名の省略
11.4. キャメルケースのタスク名を省略
11.5. ビルドスクリプトを指定してビルドするプロジェクトを選択する
11.6. プロジェクトディレクトリを使ってプロジェクトを選択する
11.7. プロジェクトに関する情報を取得する
11.8. プロジェクトに説明を添付する
11.9. タスクに関する情報を取得する
11.10. タスクレポートの内容を変更する
11.11. タスクに関してもっと多くの情報を取得する
11.12. タスクについて詳細なヘルプ情報を取得する
11.13. 依存関係の情報を取得する
11.14. 依存関係のレポートをコンフィギュレーションでフィルタする
11.15. 個別の依存関係に対する解析情報を取得する
11.16. プロパティに関する情報
12.1. GUIの起動
13.1. Projectオブジェクトへのアクセス
13.2. ローカル変数を使用する
13.3. 拡張プロパティを使用する
13.4. Groovy JDKのメソッド
13.5. プロパティアクセサ
13.6. カッコなしのメソッド呼び出し
13.7. マップリテラル、リストリテラル
13.8. メソッドのクロージャ引数
13.9. クロージャのdelegate
14.1. mkdirでディレクトリを作成する
14.2. gradle.propertiesでプロパティを設定する
14.3. 外部のビルドスクリプトファイルでプロジェクトの設定を行う
14.4. 任意のオブジェクトを組み立てる
14.5. 外部スクリプトで任意のオブジェクトを組み立てる
15.1. タスクを定義する
15.2. タスクを定義する - タスク名に文字列を使用
15.3. その他のタスク定義方法
15.4. タスクにプロパティとしてアクセスする
15.5. tasksコレクションからタスクにアクセスする
15.6. パスを使ってタスクにアクセスする

15.7. copyタスクの作成
15.8. タスクの設定 - 様々な方法
15.9. タスクの設定 - クロージャの使用
15.10. クロージャを伴うタスク定義
15.11. 別プロジェクトのタスクとの依存関係を定義する
15.12. taskオブジェクトを使った依存関係定義
15.13. クロージャを使った依存関係定義
15.14. Adding a 'must run after' task ordering
15.15. Adding a 'should run after' task ordering
15.16. Task ordering does not imply task execution
15.17. A 'should run after' task ordering is ignored if it introduces an ordering cycle
15.18. タスクに説明書きを追加する
15.19. タスクの上書き
15.20. 述語でタスクをスキップ
15.21. StopExecutionExceptionでタスクをスキップ
15.22. タスクの有効化と無効化
15.23. 生成タスク
15.24. タスクの入力と出力を宣言
15.25. タスクルール
15.26. ルールベース・タスクの依存関係
15.27. Adding a task finalizer
15.28. Task finalizer for a failing task
16.1. ファイルを参照する
16.2. ファイルコレクションの作成
16.3. ファイルコレクションを使う
16.4. ファイルコレクションを実装する
16.5. ファイルツリーを作成する
16.6. ファイルツリーを使う
16.7. アーカイブをファイルツリーとして使う
16.8. ファイルセットを指定する
16.9. ファイルセットを指定する
16.10. Copyタスクでファイルをコピーする
16.11. Copyタスクのコピー元と宛先を指定する
16.12. コピーするファイルを選択する
16.13. copy()メソッドで更新チェックせずにファイルをコピーする
16.14. copy()メソッドで更新チェックを実施してファイルをコピーする
16.15. コピー時にファイルをリネームする
16.16. コピー時にファイルをフィルタリングする
16.17. 入れ子構造のコピー仕様
16.18. Syncタスクで依存関係をコピーする
16.19. ZIPアーカイブの作成
16.20. ZIPアーカイブの作成
16.21. アーカイブタスクの設定 - カスタムアーカイブ名
16.22. アーカイブタスクの設定 - appendix & classifier
17.1. Antタスクの利用
17.2. Antタスクにネストされたテキストを渡す
17.3. Antタスクにネストされた要素を渡す
17.4. Antタイプの利用

17.5. カスタムAntタスクの利用
17.6. カスタムAntタスクに対するクラスパスの宣言
17.7. カスタムAntタスクと依存関係管理を併用
17.8. Antビルドのインポート
17.9. Antターゲットに依存するタスク
17.10. Antターゲットにふるまいを追加
17.11. Ant target that depends on Gradle task
17.12. Renaming imported Ant targets
17.13. Antプロパティの設定
17.14. Antプロパティの取得
17.15. Antリファレンスの設定
17.16. Antリファレンスの取得
18.1. ログに標準出力を使う
18.2. 自分でログメッセージを書く
18.3. SLF4Jでログを出力する
18.4. 標準出力のキャプチャ設定
18.5. タスク実行時の標準出力キャプチャ設定
18.6. Gradleがロギングするものを変更する
20.1. HTTPプロクシの設定
20.2. HTTPSプロクシの設定
21.1. プラグインの適用
21.2. Applying a script plugin
21.3. Applying a binary plugin
21.4. Applying a binary plugin by type
21.5. 型でプラグインを適用する
21.6. Applying a community plugin
21.7. プラグインにより追加されたタスク
21.8. プラグインのデフォルトを変更する
21.9. プラグインの規約オブジェクト
23.1. Javaプラグインの使用
23.2. Javaソースレイアウトのカスタマイズ
23.3. ソースセットへのアクセス
23.4. ソースセットのソースディレクトリの設定
23.5. ソースセットの定義
23.6. ソースセットの依存関係定義
23.7. ソースセットのコンパイル
23.8. ソースセットのJARを生成
23.9. ソースセットのJavadocを生成
23.10. ソースセットのテストを実行
23.11. Filtering tests in the build script
23.12. JUnit Categories
23.13. Grouping TestNG tests
23.14. Creating a unit test report for subprojects
23.15. MANIFEST.MFのカスタマイズ
23.16. manifestオブジェクトの作成
23.17. 特定のアーカイブ用にMANIFEST.MFを分離
23.18. 特定のアーカイブ用にMANIFEST.MFを分離
24.1. Groovyプラグインの使用

24.2. Groovyソースレイアウトのカスタマイズ
24.3. Groovyプラグインの設定
24.4. Groovyテスト用の依存関係設定
24.5. 同梱のGroovyを使用する依存関係設定
24.6. Groovyをファイル依存関係で設定する
25.1. Scalaプラグインを使う
25.2. Scalaソースレイアウトのカスタマイズ
25.3. 製品コードに使うScalaへの依存関係の宣言
25.4. テストコードに使うScalaへの依存関係の宣言
25.5. Fast Scala Compilerを有効にする
25.6. メモリ設定の調整
25.7. Zincベースのコンパイラを有効にする
26.1. Using the War plugin
26.2. Customization of war plugin
27.1. Earプラグインの利用
27.2. Earプラグインのカスタマイズ
28.1. Using the Jetty plugin
29.1. Checkstyleプラグインの使用
30.1. CodeNarcプラグインの使用
31.1. FindBugsプラグインの使用
32.1. JDependプラグインの使用
33.1. PMDプラグインの使用
34.1. Applying the JaCoCo plugin
34.2. Configuring JaCoCo plugin settings
34.3. Configuring test task
34.4. Configuring test task
34.5. Using application plugin to generate code coverage data
34.6. Coverage reports generated by applicationCodeCoverageReport
35.1. Sonarプラグインの適用
35.2. Sonar接続設定のコンフィグレーション
35.3. Sonarプロジェクト設定のコンフィグレーション
35.4. マルチプロジェクトビルドにおけるグローバルコンフィグレーション
35.5. マルチプロジェクトビルドにおける共通のプロジェクトコンフィグレーション
35.6. マルチプロジェクトビルドにおけるプロジェクト個別コンフィグレーション
35.7. 解析対象の言語のコンフィグレーション
35.8. プロパティ文法の利用
35.9. カスタムソースセットの解析
35.10. Java言語以外の解析
35.11. カスタムグローバルプロパティ設定
35.12. カスタムプロジェクトプロパティ設定
35.13. カスタムのコマンドラインプロパティを実装する
36.1. Applying the Sonar Runner plugin
36.2. Configuring Sonar connection settings
36.3. Configuring Sonar runner version
36.4. Global configuration settings
36.5. Shared configuration settings
36.6. Individual configuration settings
36.7. Skipping analysis of a project

36.8. Analyzing custom source sets
36.9. Analyzing languages other than Java
36.10. setting custom Sonar Runner fork options
37.1. OSGiプラグインの利用
37.2. OSGiのMANIFEST.MFファイルの設定
38.1. Eclipseプラグインの使用方法
38.2. classpath の一部を上書き
38.3. project の一部を上書き
38.4. 依存関係のエクスポート
38.5. 依存関係のエクスポート
38.6. XML のカスタマイズ
38.7. XML のカスタマイズCustomizing the XML
39.1. IDEAプラグインを使う
39.2. モジュールの部分的な上書き
39.3. プロジェクトの部分的な上書き
39.4. Export Dependencies
39.5. XMLをカスタマイズする
40.1. ANTLRプラグインの利用
40.2. ANTLRバージョン宣言
42.1. 通知プラグインを使用する
42.2. 通知プラグインの設定
42.3. 通知プラグインを使用する
43.1. ビルド通知プラグインの利用
43.2. 初期化スクリプトからビルド通知プラグインを使う
44.1. Using the distribution plugin
44.2. Adding extra distributions
44.3. Configuring the main distribution
45.1. Using the application plugin
45.2. Configure the application main class
45.3. Configure default JVM settings
45.4. 他タスクの出力をアプリケーションのディストリビューションに含める
45.5. ディストリビューションのファイルを自動的に作成する
46.1. Using the Java library distribution plugin
46.2. Configure the distribution name
46.3. Include files in the distribution
49.1. Using the Build Dashboard plugin
50.1. Using the Java Gradle Plugin Development plugin
51.1. コンフィギュレーションの定義
51.2. コンフィギュレーションへのアクセス
51.3. コンフィギュレーションの設定変更
51.4. モジュール依存関係
51.5. アーティファクトオンリー記法
51.6. 分類子付きの依存関係
51.7. あるコンフィグレーションの内部を列挙する
51.8. クライアントモジュール依存関係 - 推移的な依存関係
51.9. プロジェクト依存関係
51.10. ファイル依存関係
51.11. 生成されるファイルへの依存関係

51.12. Gradle API依存関係
51.13. Gradleに同梱されているGroovyへの依存関係
51.14. 推移的な依存関係の除外
51.15. 依存関係のオプション属性
51.16. 依存関係定義のコレクション、配列
51.17. 依存するコンフィギュレーションの指定
51.18. 依存するプロジェクトのコンフィギュレーション
51.19. Configuration.copy
51.20. 宣言した依存関係にアクセスする
51.21. Configuration.files
51.22. フィルター付きConfiguration.files
51.23. Configuration.copy
51.24. Configuration.copy vs. Configuration.files
51.25. Declaring a Maven and Ivy repository
51.26. Providing credentials to a Maven and Ivy repository
51.27. Mavenセントラルリポジトリを追加する
51.28. BintrayのJCenter Mavenリポジトリを追加する
51.29. Using Bintrays's JCenter with HTTP
51.30. Mavenのローカルキャッシュをリポジトリとして追加する
51.31. カスタムMavenリポジトリを追加する
51.32. JARファイル用の追加リポジトリを設定する
51.33. パスワード保護されたMavenリポジトリへのアクセス
51.34. フラットディレクトリ・リゾルバ
51.35. Ivyリポジトリ
51.36. Ivy repository with named layout
51.37. パターンレイアウトを指定したIvyリポジトリ
51.38. Maven互換レイアウトのIvyリポジトリ
51.39. Ivy repository with pattern layout
51.40. Ivy repository with multiple custom patterns
51.41. Ivy repository with Maven compatible layout
51.42. パスワードで保護されたIvyリポジトリ
51.43. リポジトリ定義へのアクセス
51.44. リポジトリの設定変更
51.45. カスタムリポジトリの定義
51.46. あるグループのライブラリ全てで一貫したバージョンを使用するよう強制する
51.47. 独自のバージョニング体制を実装する
51.48. バージョンのブラックリスト指定と差し替
51.49. 解決時に依存関係のグループ名や名前を変更する
51.50. Declaring module replacement
51.51. 動的解決モードを有効にする
51.52. 'Latest' version selector
51.53. Custom status scheme
51.54. Ivy component metadata rule
51.55. Component selection rule
51.56. Component selection rule with module target
51.57. Component selection rule with metadata
51.58. 動的バージョンのキャッシュ制御
51.59. 変更性モジュールのキャッシュ制御

52.1. アーカイブタスクを使ってアーティファクトを宣言する
52.2. アーティファクトをファイルで宣言する
52.3. アーティファクトのカスタマイズ
52.4. ファイルでアーティファクトを宣言するMapスタイルの文法
52.5. アップロードタスクの設定
53.1. Mavenプラグインの利用
53.2. スタンドアロンPOMの生成
53.3. リモートMavenリポジトリへのファイルアップロード
53.4. SSH経由でのファイルアップロード
53.5. pomのカスタマイズ
53.6. ビルダースタイルでpomをカスタマイズする
53.7. 自動生成された内容を変更する
53.8. Mavenインストーラーのカスタマイズ
53.9. 複数POMの生成
53.10. マッピングコンフィグレーションへのアクセス
54.1. 署名プラグインの使用
54.2. コンフィグレーションに署名する
54.3. コンフィグレーションの出力に署名する
54.4. タスクを署名する
54.5. タスクの出力に署名する
54.6. 条件付き署名
54.7. Signing a POM for deployment
55.1. Defining a library component
55.2. Defining executable components
55.3. The components report
55.4. The 'cpp' plugin
55.5. C++ source set
55.6. The 'c' plugin
55.7. C source set
55.8. The 'assembler' plugin
55.9. The 'objective-c' plugin
55.10. The 'objective-cpp' plugin
55.11. Settings that apply to all binaries
55.12. Settings that apply to all shared libraries
55.13. Settings that apply to all binaries produced for the 'main' executable component
55.14. Settings that apply only to shared libraries produced for the 'main' library component
55.15. The 'windows-resources' plugin
55.16. Configuring the location of Windows resource sources
55.17. Building a resource-only dll
55.18. Providing a library dependency to the source set
55.19. Providing a library dependency to the binary
55.20. Declaring project dependencies
55.21. Defining build types
55.22. Configuring debug binaries
55.23. Defining platforms
55.24. Defining flavors
55.25. Targeting a component at particular platforms
55.26. Building all possible variants

55.27. Defining tool chains
55.28. Reconfigure tool arguments
55.29. Defining target platforms
55.30. Registering CUnit tests
55.31. Registering CUnit tests
55.32. Running CUnit tests
56.1. シングルプロジェクトのビルド
56.2. 階層構造のレイアウト
56.3. フラットなレイアウト
56.4. プロジェクト・ツリーの属性を変更する
56.5. プロジェクトツリーの属性を変更する
56.6. 特定プロパティを持つプロジェクトにタスクを追加する
56.7. 通知
56.8. すべてのタスクにプロパティ値を設定する
56.9. タスク実行の開始時および終了時にロギングを行う
57.1. マルチプロジェクト・ツリー - warter & bluewhale プロジェクト
57.2. water(親プロジェクト)のビルドスクリプト
57.3. マルチプロジェクトツリー - water, bluewhaleそしてkrill
57.4. Waterプロジェクトのビルドスクリプト
57.5. サブプロジェクト共通の振る舞いとすべてのプロジェクト共通の振る舞いをそれぞれ定義する
57.6. プロジェクト個別の振る舞いを定義する
57.7. krillプロジェクトに個別の振る舞いを定義する
57.8. プロジェクトに振る舞いを追加する(プロジェクト名によるフィルタリング)
57.9. プロジェクトに振る舞いを追加する(プロパティによるフィルタリング)
57.10. サブプロジェクトからビルドを実行する
57.11. プロジェクトの評価と実行
57.12. プロジェクトの評価と実行
57.13. 絶対パスによるタスクの実行
57.14. 依存関係とビルド実行順序
57.15. 依存関係とビルド実行順序
57.16. 依存関係とビルド実行順序
57.17. 依存関係を宣言する
57.18. 依存関係を宣言する
57.19. プロジェクトにまたがるタスク間の依存関係
57.20. 評価順序の依存関係
57.21. 評価順序の依存関係 - evaluationDependsOn
57.22. 評価順序の依存関係
57.23. 依存関係 - 実生活の例 - クロスプロジェクト設定
57.24. プロジェクト依存関係
57.25. プロジェクト依存関係
57.26. 細かい依存関係の制御
57.27. シングルプロジェクトのビルドとテスト
57.28. シングルプロジェクトの部分ビルドとテスト
57.29. 依存プロジェクトのビルドとテスト
57.30. 依存されているプロジェクトのビルドとテスト
58.1. カスタムタスクの定義
58.2. hello worldタスク
58.3. カスタマイズ可能なhello worldタスク

58.4. カスタムタスクのビルド
58.5. カスタムタスク
58.6. カスタムタスクを別のプロジェクトで使う
58.7. カスタムタスクのテスト
58.8. Defining an incremental task action
58.9. Running the incremental task for the first time
58.10. Running the incremental task with unchanged inputs
58.11. Running the incremental task with updated input files
58.12. Running the incremental task with an input file removed
58.13. Running the incremental task with an output file removed
58.14. Running the incremental task with an input property changed
59.1. カスタムプラグイン
59.2. カスタムプラグインのextension
59.3. コンフィグレーションクロージャ付きのカスタムプラグイン
59.4. ファイルプロパティの遅延評価
59.5. カスタムプラグインに対するビルド
59.6. カスタムプラグインに対するワイヤリング
59.7. 別のプロジェクトでカスタムプラグインを使う
59.8. Applying a community plugin with the plugins DSL
59.9. カスタムプラグインのテスト
59.10. Using the Java Gradle Plugin Development plugin
59.11. ドメインオブジェクトの管理
60.1. プロパティとメソッドの継承を使う
60.2. プロパティとメソッドのインジェクション
60.3. カスタムbuildSrcビルドスクリプト
60.4. buildSrcルートプロジェクトにサブプロジェクトを追加する
60.5. 別のビルドを呼び出す
60.6. ビルドスクリプトのクラスパスを宣言する
60.7. 外部ライブラリをビルドスクリプトで使用する
60.8. Ant optional dependencies
61.1. プロジェクト評価前に初期化スクリプトで追加的な設定を行う
61.2. 初期化スクリプトの外部依存関係定義
61.3. 外部依存関係を持つ初期化スクリプト
61.4. Using plugins in init scripts
62.1. ラッパータスク
62.2. ラッパーにより生成されるファイル
65.1. Applying the “ivy-publish” plugin
65.2. Publishing a Java module to Ivy
65.3. Publishing additional artifact to Ivy
65.4. customizing the publication identity
65.5. Customizing the module descriptor file
65.6. Publishing multiple modules from a single project
65.7. Declaring repositories to publish to
65.8. Choosing a particular publication to publish
65.9. Publishing all publications via the “publish” lifecycle task
65.10. Generating the Ivy module descriptor file
65.11. Publishing a Java module
65.12. Example generated ivy.xml

66.1. Applying the 'maven-publish' plugin
66.2. Adding a MavenPublication for a Java component
66.3. Adding additional artifact to a MavenPublication
66.4. customizing the publication identity
66.5. Modifying the POM file
66.6. Publishing multiple modules from a single project
66.7. Declaring repositories to publish to
66.8. Publishing a project to a Maven repository
66.9. Publish a project to the Maven local repository
66.10. Generate a POM file without publishing
B.1. 変数のスコープ：ローカルスコープとスクリプトスコープ
B.2. 設定フェーズと実行フェーズの区別

翻訳版について

本ドキュメントは、Gradleユーザーガイドを非公式に日本語に翻訳したものです。翻訳作業は
github上のプロジェクト
で行っていますので、誤訳、不自然な点など発見されましたらプロジェクトサイト上からご連絡ください。

なお、原文は添付の代替スタイルシート(original)を使用すれば表示されます。

1 はじめに

Gradleは、Java(JVM)環境におけるビルドシステムであり、従来のビルド技術を大きく躍進させるもので

- あらゆる目的に使用できる汎用ビルドツールです。GradleはAntと同じような目的で使うことができま
- その一方でMavenのような規約によるビルドフレームワークを提供します。その規約も、気に入らな
- マルチプロジェクトを強力にサポートします。
- 強力な依存関係管理 (Apache Ivy ベース)を提供します。
- Maven/Ivyリポジトリを完全にサポートします。
- リモートリポジトリや`pom.xml`、`ivy.xml`を使用しなくても推移的な依存関係を管理できます。
- Antのタスクとプロジェクトがファーストクラス・オブジェクトとして組み込まれており、デフォルト
- Groovyでビルドスクリプトを記述します。
- ビルドを記述する際、オブジェクト指向のリッチなドメインモデルを使用可能です。

2章概要でGradleの概要について記述しています。チュートリアル
も用意しているので楽しんでいってください(^^)

1.1. このユーザーガイドについて

このユーザーガイドは、Gradle同様、現在活発に更新されているところです。まだGradleについて必要な
また、記載内容には明確でない部分やGradleについて読者が知っている以上の知識を要求する部分があ
このガイドを改良していくため、力を貸してください。
ドキュメンテーションへ参加、貢献していただける方はGradleのウェブサイトをご参照ください。

Throughout the user guide, you will find some diagrams that represent dependency relationships between Gradle tasks. These use something analogous to the UML dependency notation, which renders an arrow from one task to the task that the first task depends on.

2
概要

2.1. 特長

Gradleの特長は以下の通りです。

宣言的なビルドの記述と規約によるビルド
　Gradleの核となっているのは、拡張性豊富なGroovyベースのDSLです。
　それは、好きなように組み立てて記述できる、ビルド用の宣言型プログラミング言語とも言えるもの
　そのプログラミング言語には、JavaやGroovy、OSGi、WebそしてScalaなどのプロジェクトを一般的
　さらに、このプログラミング言語はとても拡張性豊富なものです。
　新たな言語機能を追加したり、既存の言語機能を拡張したりすることで、簡潔でメンテナンス性の良

タスクグラフ用のプログラミング言語
　Gradleの提供する宣言型プログラミング言語は、グラフ構造をもつ汎用のタスク群の上に構築されて
　皆さんのビルドシステムには様々なニーズがあると思いますが、Gradleは、それらに適合できる、究

ビルドの構造化
　Gradleは柔軟性があり機能が豊富なので、ビルドを記述する際にプログラミングにおける一般的な設

深いAPI
　ビルドが実行されるライフサイクル全体に、たくさんのフックを埋め込むことができます。
　なので、非常に深い部分までGradleの設定や振る舞いをモニタリングしたりカスタマイズしたりする

ビルドの分割
　Gradleでは、ビルドの分割を強力にサポートしています。このことは、単純なシングルプロジェクト

マルチプロジェクトのビルド
　Gradleは突出したマルチプロジェクトのサポートが特長です。プロジェクト間の依存関係はGradleの

　Gradleでは部分ビルドが可能です。一つのサブプロジェクトをビルドすれば、Gradleはそのプロジェ

依存関係を管理する多くの方法
　外部ライブラリの解決には、チームごとにさまざまな方法があるものです。リモートのMavenやIvyの

他のビルドツールとの統合
　Gradleでは、Antのタスクはファーストクラス・オブジェクトです。さらにおもしろいことに、Antの

　Gradleは、既存のMaven/Ivyリポジトリのインフラ、依存関係の定義や解決といった機能を完全にサ
　p o m . x m l
　をGradleのスクリプトに変換するコンバーターも用意されています。実行時にその変換を行う機能も

容易に移行可能
　Gradleは、どんな構造のプロジェクトにも適用できます。なので、今製品のビルドを行っているブラ

Groovy
　GradleのビルドスクリプトはXMLではなくGroovyで記述します。しかし、それは単純に動的言語のス

Gradleラッパー
　Gradleラッパーを使うと、Gradleをインストールしていないマシンでビルドを実行できます。たとえ
　administration)というアプローチでもあり、企業向けと考えてもとても面白いものだといえます。使

フリーかつオープンソース
　Gradleはオープンソースプロジェクトであり、ASLのもとで公開されています。

2.2. なぜGroovyなのか?

ビルドスクリプト
を記述するということを考えたとき、XMLに対する内部DSL(動的言語ベース)のアドバンテージというの
ただ、動的な言語はいくつかあるのに、なぜGroovyなのでしょうか。
その答えは、Gradleが想定しているシチュエーション、環境にあります。
Gradleは本質的に言えば一般的な利用が可能な汎用のビルドツールですが、メインとしてフォーカスして
そのようなプロジェクトの場合、チームのメンバーがJavaに習熟しているのは明らかでしょう。
私たちは、ビルドはすべての
チームメンバーに対してできるだけ分かりやすいものであるべきだと考えています。

それならGroovyでなくJavaを使えばいいのにと思われるかもしれません。それはもっともな疑問です。
ビルド用の言語にJavaを使えば、開発チームにとって最高に分かりやすく、学習曲線が最小限緩やかなビ
しかし、Javaはビルド用の言語として使うには表現力や機能に限界があるのです。[2]
このような用途には、PythonやGroovy、Rubyのような言語の方が向いています。
私たちは、Java世界の住人にとって断然飛び抜けた分かりやすさを提供するGroovyを選択しました。
Groovyの文法は、型システム、パッケージ構造、その他諸々についてJavaをベースにしています。Groo

Javaの開発者であっても、PythonやRubyの知識を持っていたり、それらを楽しく学んでいるのなら上記
GradleはビルドスクリプトエンジンとしてJRubyやJythonを使えるよう適切にデザインされています。
私達にとって、目下のところはそれらの優先度は高くありませんが。
私たちは、そういった追加のビルドスクリプトエンジンを作っていただけるあらゆるコミュニティを喜ん

---

[2] http://www.defmacro.org/ramblings/lisp.html
に、AntとXML、Java、Lispを比較している興味深い記事が載っています。
「もしJavaがこの文法を持っていたら」という前提で書かれている構文が、まさにGroovyの文法になって

3 チュートリアル

3.1. 始めてみよう

次のチュートリアルはGradleの基本機能を紹介しているものです。Gradleを始める手助けになるでしょ

4章Gradleのインストール
: Gradleのインストール方法について

6章ビルドスクリプトの基本
: ビルドスクリプトの基本要素、プロジェクトとタスクの紹介

7章Javaクイックスタート
: Javaプロジェクトにおける、Gradleの規約によるビルドのチュートリアル

8章依存関係管理の基本
: Gradleの依存関係管理のチュートリアル

9章Groovyクイックスタート
: Groovyプロジェクトにおける、Gradleの規約によるビルドのチュートリアル

10章Webアプリケーションクイックスタート
: Webアプリケーションプロジェクトにおける、Gradleの規約によるビルドのチュートリアル

4
Gradleのインストール

4.1. 必要環境

Gradleを使用するには、バージョン6以上のJava JDKもしくはJREが必要です(**`java -version`**
で 確 認 し て く だ さ い) 。
GradleにはGroovyライブラリが同梱されているので、別途Groovyをインストールする必要はありません

Gradleは、システムのパス上にあるJDKを使用します。 パス上のJDKの代わりに、環境変数
`JAVA_HOME`
にJDKをインストールしたディレクトリをセットして、使用したいJDKを指定することもできます。

4.2. ダウンロード

Gradleはウェブサイトからダウンロードできます。

4.3. 解凍

Gradleの配布物はZIPで圧縮されています。Gradleのフルパッケージには以下のものが含まれています。

- Gradleのバイナリ
- ユーザーガイド(HTMLとPDF)
- DSLのリファレンスガイド
- APIドキュメント(JavadocとGroovydoc)
- ユーザーガイドから参照されているサンプルを含め、大量のサンプルが同梱されています。複雑なビ
- Gradleバイナリのソースコード。これは参照専用のものです。Gradleをビルドするには、ソースを別 Gradleのウェブサイトを参照してください。

4.4. 環境変数

`GRADLE_HOME``/bin`を環境変数`PATH`
に追加してください。Gradleの実行に必要な環境変数の設定は、通常これだけです。

4.5. 正しくインストールされたことをテストする

Gradleは、gradle
コマンドで実行します。Gradleが正しくインストールされていることを確認するには、gradle -v
と入力してください。Gradleのバージョンとローカルの実行環境(groovyやjvmのバージョンなど)が表示

4.6. JVMオプション

Gradle実行時に引き渡すJVMオプションは、環境変数GRADLE_OPTSとJAVA_OPTS
で設定します。両方一緒に使うこともできます。 JAVA_OPTS
に設定したオプションは、慣習により多くのJavaアプリケーションで共有されるものです。
典型的な例で言えば、HTTPプロキシはJAVA_OPTSに設定し、使用メモリに関する設定は
GRADLE_OPTSにセットする、といった使い分けが考えられるでしょう。 これらの設定はgradleや
gradlewスクリプトの頭に記述することもできます。

Note that it's not currently possible to set JVM options for Gradle on the command line.

5
トラブルシューティング

この章は現在執筆途中です。

Gradleを使っていると（他のソフトウェアでも同じですが）、問題に突き当たる可能性が常にあります。
ある機能をどうやって使えばいいのか分からなかったり、バグに遭遇したりといった場合もありますし、

この章ではトラブルシューティングのためのアドバイスや、突き当たった問題に対して手助けを求める方

5.1. トラブルに対処する

何かトラブルが起こったとき、まず試すことができるのは、最新版のGradleを使ってみることです。Gra

Gradleデーモンを使っているなら、一時的にデーモンを停止してみてください（`--no--daemon`
オプションをつけて実行）。　デーモン使用時のトラブルシューティングについて、詳しくは19章
Gradleデーモン を参照してください。

5.2. ヘルプを求めるには

Gradleに関して手助けを求めたいときは、http://forums.gradle.orgに行ってみましょう。
Gradleフォーラムでは、Gradleのデベロッパーやコミュニティメンバーに問題を報告したり、質問したり

何か問題が発生したら、フォーラムに質問や報告を投げるのが最も早く手助けを得られる方法です。
フォーラムでは、改善のための提案や新しいアイデアを投稿することもできます。
また、Gradleの開発チームが頻繁にニュースやリリースのアナウンスを投稿していますので、Gradle開発

6 ビルドスクリプトの基本

6.1. プロジェクトとタスク

Gradleの根本にあるのは、プロジェクトとタスクという二つの基本的な概念です。

Gradleによるビルドは、一つ以上のプロジェクト
から構成されます。プロジェクトとは、ビルドされるソフトウェアを構成するコンポーネントのことです

それぞれのプロジェクトは、一つ以上のタスク
から構成されます。タスクとは、分割不能な何らかの作業単位を表す概念です。たとえば、クラスをコン

では、実際にプロジェクト上でいくつか簡単なタスクを定義してビルドしてみましょう。マルチプロジェ

6.2. Hello world

Gradleのビルドは、gradleコマンドで実行します。gradleコマンドはカレントディレクトリの`build.gr`
ファイルを参照してビルドを行います。[4]このファイルは、一般的にはビルドスクリプト
と呼ばれます(後述しますが、正確には「ビルドの設定を行うスクリプト」です)。Gradleのプロジェクト

次のようなスクリプトを、`build.gradle`という名前で作成して試してみてください。

例6.1 初めてのビルドスクリプト

build.gradle

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
```

コンソールを開いてこの`build.gradle`のあるディレクトリに移動して、**`gradle -q hello`**
と打ち込んでください。ビルドスクリプトが実行され、以下のように出力されるはずです。

例6.2 ビルドスクリプトの実行

`gradle -q hello` の出力

```
> gradle -q hello
Hello world!
```

-q オプションって何?

このユーザーガイドでは、ほぼすべ
- q

何が起こったのでしょう。ビルドスクリプトは、`hello` というタスクを一つ定義していて、このタスクにアクションを一つ追加しています。**`gradle hello`**と実行することで、Gradleがこの`hello` を実行し、`hello` に追加したアクションが実行されたのです。アクションはGroovyコードを含む単なるクロージャです。

をつけてコマンドを実行しています。第8章 ロギングにもっと詳しく記載されています。

Antのtargetに似ているなと思われたかもしれません。実際、GradleのtaskはAntでいえばtargetに相当す taskという言葉が、target という言葉よりも実態に合った表現だと考えたためです。しかし、残念ながらtask という用語はAntでもcopyやjavac といったコマンドを示すものとして使われており、Gradleのtaskと競合してしまいます。このため、Ant ant taskと明示的にいい、単にtaskとしたときはGradleのタスクのことを示すものとします。

6.3. タスク定義のショートカット

タスク定義は、実際にはもっと短く書けます。先ほどの`hello` タスクも、次のように簡潔に書くことができます。

例6.3 タスク定義のショートカット

build.gradle

```
task hello << {
    println 'Hello world!'
}
```

この例も先ほどと同様、実行するクロージャを一つだけもった`hello` を定義するものです。このユーザーガイドではこちらの定義方法を使用します。

6.4. ビルドスクリプトはコードです

GradleのビルドスクリプトではGroovyの機能をすべて使うことができます。手始めに次の例をご覧くだ

例6.4 GradleタスクでGroovyを使う

build.gradle

```
task upper << {
    String someString = 'mY_nAmE'
    println "Original: " + someString
    println "Upper case: " + someString.toUpperCase()
}
```

`gradle -q upper` の出力

```
> gradle -q upper
Original: mY_nAmE
Upper case: MY_NAME
```

さらに

例6.5 GradleタスクでGroovyを使う

build.gradle

```
task count << {
    4.times { print "$it " }
}
```

gradle -q count の出力

```
> gradle -q count
0 1 2 3
```

6.5. タスクの依存関係

ご想像の通り、タスク間の依存関係を宣言できます。

例6.6 タスク間の依存関係を宣言する

build.gradle

```
task hello << {
    println 'Hello world!'
}
task intro(dependsOn: hello) << {
    println "I'm Gradle"
}
```

gradle -q intro の出力

```
> gradle -q intro
Hello world!
I'm Gradle
```

依存関係にタスクを追加するときは、そのタスクがその時点で宣言されていなくてもかまいません。

例6.7 遅延評価のdependsOn - タスクがまだ宣言されていない場合

build.gradle

```
task taskX(dependsOn: 'taskY') << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
```

gradle -q taskX の出力

```
> gradle -q taskX
taskY
taskX
```

`taskX`が`taskY`に依存していることが宣言されていますが、宣言した時点では`taskY`

の定義文はありません。このことは、マルチプロジェクトのビルドで非常に重要になってきます。タスク
「タスクに依存関係を追加する」でさらに詳しく述べられています。

なお、未定義のタスクを参照する場合、略記法(「略記法」
参照)は使えませんので注意してください。

6.6. 動的なタスク

Groovyが力を発揮するのは、タスクが実行するアクションを記述するときだけではありません。たとえば

例6.8 動的なタスク定義

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
```

gradle -q task1 の出力

```
> gradle -q task1
I'm task number 1
```

6.7. 既存のタスクを操作する

タスクが公開しているAPIを使えば、既存のタスクにアクセスして定義内容を操作することができます。

例6.9 APIからタスクにアクセスする - 依存関係の追加

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
task0.dependsOn task2, task3
```

gradle -q task0 の出力

```
> gradle -q task0
I'm task number 2
I'm task number 3
I'm task number 0
```

また、既存のタスクにアクションを追加することもできます。

例6.10 APIからタスクにアクセスする - アクションの追加

build.gradle

```
task hello << {
    println 'Hello Earth'
}
hello.doFirst {
    println 'Hello Venus'
}
hello.doLast {
    println 'Hello Mars'
}
hello << {
    println 'Hello Jupiter'
}
```

gradle -q hello の出力

```
> gradle -q hello
Hello Venus
Hello Earth
Hello Mars
Hello Jupiter
```

d o F i r s t や d o L a s t
は何回でも呼び出すことができ、既存のタスクが実行する一連のアクションの最初や最後に、新しいアク
<<はdoLastの単なるエイリアスです。

6.8. 略記法

先ほどの例をみたとき、おやっと思われたかもしれません。既存の
タスクにアクセスするときは、便利な略記法を使うことができます。ただのプロパティとしてすべてのタ

例6.11 ビルドスクリプトのプロパティとして既存のタスクにアクセスする

build.gradle

```
task hello << {
    println 'Hello world!'
}
hello.doLast {
    println "Greetings from the $hello.name task."
}
```

gradle -q hello の出力

```
> gradle -q hello
Hello world!
Greetings from the hello task.
```

この略記法は、コードをとても読みやすいものにします。特に、プラグインなどで外部から提供されるタ
compile)にアクセスするときには威力を発揮します。

6.9. 拡張タスクプロパティ

独自のプロパティをタスクに追加することができます。例えば、myProperty
というプロパティを追加するには、ext.myPropertyに初期値を設定してください。
その時点で、このプロパティを定義済みのプロパティと同じように読み書きできるようになります。

例6.12 拡張プロパティをタスクに追加する

build.gradle

```
task myTask {
    ext.myProperty = "myValue"
}

task printTaskProperties << {
    println myTask.myProperty
}
```

gradle -q printTaskProperties の出力

```
> gradle -q printTaskProperties
myValue
```

拡張プロパティを追加できるのは、タスクだけではありません。詳細については、「拡張プロパティ
」を参照してください。

6.10. Antタスクの使用

AntのタスクはGradleのファーストクラス・オブジェクトであり、デフォルトで使用できるようになって
AntBuilderが組み込まれているのです。Antのタスクは、build.xml
で使うよりGradleで使ったほうが便利で強力です。次の例を見てください。Antのタスクを実行する方法

例6.13 AntBuilderを使ってant.loadfileターゲットを実行する

build.gradle

```
task loadfile << {
    def files = file('../antLoadfileResources').listFiles().sort()
    files.each { File file ->
        if (file.isFile()) {
            ant.loadfile(srcFile: file, property: file.name)
            println " *** $file.name ***"
            println "${ant.properties[file.name]}"
        }
    }
}
```

gradle -q loadfile の出力

```
> gradle -q loadfile
*** agile.manifesto.txt ***
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration  over contract negotiation
Responding to change over following a plan
 *** gradle.manifesto.txt ***
Make the impossible possible, make the possible easy and make the easy elegant.
(inspired by Moshe Feldenkrais)
```

Antのタスクを使ってできることはもっとたくさんあります。17章GradleからAntを使うをご参照ください。

6.11. メソッドの使用

Gradleでは、段階的にビルドロジックを体系化していくことができます。最初の一歩として、上の例を›

例6.14 メソッドを抽出してビルドロジックを整理する

build.gradle

```
task checksum << {
    fileList('../antLoadfileResources').each {File file ->
        ant.checksum(file: file, property: "cs_$file.name")
        println "$file.name Checksum: ${ant.properties["cs_$file.name"]}"
    }
}

task loadfile << {
    fileList('../antLoadfileResources').each {File file ->
        ant.loadfile(srcFile: file, property: file.name)
        println "I'm fond of $file.name"
    }
}

File[] fileList(String dir) {
    file(dir).listFiles({file -> file.isFile() } as FileFilter).sort()
}
```

gradle -q loadfile の出力

```
> gradle -q loadfile
I'm fond of agile.manifesto.txt
I'm fond of gradle.manifesto.txt
```

後でマルチプロジェクトの各プロジェクトでメソッドを共有する方法について学びます。それでもビルト
60章ビルドロジックの体系化)

6.12. デフォルトタスク

例6.15 デフォルトタスクの定義

build.gradle

```
defaultTasks 'clean', 'run'

task clean << {
    println 'Default Cleaning!'
}

task run << {
    println 'Default Running!'
}

task other << {
    println "I'm not a default task!"
}
```

gradle -q の出力

```
> gradle -q
Default Cleaning!
Default Running!
```

これは、gradle clean run
と入力するのと同じです。マルチプロジェクトのビルドでは、すべてのサブプロジェクトが個別にデフォ

6.13. DAGによる設定

詳しくは56章ビルドのライフサイクル
で説明しますが、Gradleのビルドはいくつかのフェーズに分かれて実行されます。Gradleはどのタスクを

次の例では、distributionタスクを実行するかreleaseタスクを実行するかでversion
変数に違う値を割り当てています。

例6.16 選択したタスクによって異なる結果を得る

build.gradle

```
task distribution << {
    println "We build the zip with version=$version"
}

task release(dependsOn: 'distribution') << {
    println 'We release now'
}

gradle.taskGraph.whenReady {taskGraph ->
    if (taskGraph.hasTask(release)) {
        version = '1.0'
    } else {
        version = '1.0-SNAPSHOT'
    }
}
```

gradle -q distribution の出力

```
> gradle -q distribution
We build the zip with version=1.0-SNAPSHOT
```

gradle -q release の出力

```
> gradle -q release
We build the zip with version=1.0
We release now
```

ここで大事なことは、releaseタスクが実行予定にあるという事実を、releaseタスクが実行される前に利
プライマリタスク、つまりgradleコマンドに引き渡されているタスクでなくてもかまいません。

6.14. 次のステップは？

この章で、私たちは初めてタスクというものに触れました。しかし、タスクについての話はこれで全部で
15章タスク詳解を見てみてください。

そうでなければ、7章Javaクイックスタートと8章依存関係管理の基本
に進み、チュートリアルを続けてください。

[4]　この動作はコマンドラインスイッチを使用して変更できます。付録D　Gradle　コマンドラインを参照してください。

7 Javaクイックスタート

7.1. Javaプラグイン

ここまで見てきたように、Gradleは汎用のビルドツールです。ビルドスクリプトで実装したいと思うよう

多くのJavaプロジェクトは、基本的な部分では非常に似通っています。つまり、Javaソースファイルをコ
プラグイン
によってこの問題を解決します。プラグインは、何らかの方法（よくあるのは設定済みの一連の有用なタ
Javaプラグイン
です。このプラグインは、Javaソースコードのコンパイルやユニットテスト、JARファイルの作成などを

Javaプラグインは規約ベースです。すなわち、このプラグインはプロジェクトのさまざまな点、例えばJa

Javaプラグイン、依存関係の管理、マルチプロジェクトのビルドについては、後の章で多くの実例付きで

7.2. 基本的なJavaプロジェクト

シンプルな例から見ていきましょう。Javaプラグインを使うためには、ビルドファイルに以下を加えます

例7.1 Javaプラグインの使用

build.gradle

```
apply plugin: 'java'
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/java/quickstart

Javaプロジェクトの定義に必要なのはこれだけです。これでプロジェクトにJavaプラグインが適用され、

Gradleは、製品のソースコードが`src/main/java`
に、テストのソースコードが`src/test/java`
にあることを想定しています。さらに、`src/main/resources`
の下にあるファイルはすべてリソースとしてJARに入れられ、また
`src/test/resources`

どんなタスクが利用可能

gradle tasks
を使えば、プロジェクトのタスクを

にあるファイルはテスト実行時に使われるクラスパスに入れられます。すべての出力ファイルは buildディレクトリの下に作られ、最終的にbuild/libs ディレクトリにJARファイルが作られます。

7.2.1. プロジェクトのビルド

Javaプラグインはかなり多くのタスクをプロジェクトに追加します。しかし、プロジェクトをビルドする buildタスクです。**gradle build** を実行すると、Gradleはコードをコンパイルしてテストし、メインクラスやリソースを含んだJARファイ

例7.2 Javaプロジェクトのビルド

gradle build の出力

```
> gradle build
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL

Total time: 1 secs
```

他にも便利なタスクがあります：

clean
: buildディレクトリを削除し、ビルドしたすべてのファイルを削除します。

assemble
: コードをコンパイルしJARを生成しますが、ユニットテストは実行しません。他のプラグインはこのタ

check
: Compiles and tests your code. Other plugins add more checks to this task. For example, if you use the checkstyle plugin, this task will also run Checkstyle against your source code.

  コードをコンパイルし、テストします。他のプラグインはこのタスクにより多くの検査項目を追加し

7.2.2. External dependencies

Usually, a Java project will have some dependencies on external JAR files. To reference these JAR files in the project, you need to tell Gradle where to find them. In Gradle, artifacts such as JAR files, are located in a repository. A repository can be used for fetching the dependencies of a project, or for publishing the artifacts of a project, or both. For this example, we will use the public Maven repository:

JavaプロジェクトはD外部のJARファイルに依存することが普通です。プロジェクトで使うこういったJAR
リ　　ポ　　ジ　　ト　　リ
に置かれます。リポジトリは、プロジェクトが依存するものを取得したり、プロジェクトのアーティファ

例7.3 Mavenリポジトリの追加

build.gradle

```
repositories {
    mavenCentral()
}
```

依存関係をいくつか追加してみましょう。製品クラスはコンパイル時にcommonsコレクションに依存し

例7.4 依存関係の追加

build.gradle

```
dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

詳しい説明は51章依存関係の管理 をご覧ください。

7.2.3. プロジェクトのカスタマイズ

Javaプラグインは多くのプロパティをプロジェクトに追加します。こういったプロパティには普通、十分
manifestにいくつか属性を追加しています。

例7.5 MANIFEST.MFのカスタマイズ

build.gradle

```
sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}
```

Javaプラグインが追加するタスクはごく普通のタスクで、ビルドファイルで宣言されるものとまったく同
Test型であるtest
タスクを設定して、テスト実行時のシステムプロパティを追加しています：

どんなプロパティが利用

gradle properties
を使えば、プロジェクトのプロパティ

例7.6 テスト用システムプロパティの追加

build.gradle

```
test {
    systemProperties 'property': 'value'
}
```

7.2.4. JARファイルの公開

通常は、JARファイルをどこかに公開する必要があるでしょう。このため、GradleにJARファイルの公開:

例7.7 JARファイルの公開

build.gradle

```
uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

JARファイルを公開するには、**`gradle uploadArchives`**を実行してください。

7.2.5. Eclipseプロジェクトの作成

`.project`
のようなEclipse固有の設定ファイルを生成するには、別のプラグインをビルドファイルに追加する必要が

例7.8 Eclipseプラグイン

build.gradle

```
apply plugin: 'eclipse'
```

これで**`gradle eclipse`**
コマンドを実行すればEclipseのプロジェクトファイルが生成されます。Eclipseタスクの詳細は38章 Eclipse プラグインをご覧ください。

7.2.6. まとめ

これまでの例の完全なビルドファイルを以下に示します：

例7.9 Javaの例 - 完全なビルドファイル

build.gradle

```
apply plugin: 'java'
apply plugin: 'eclipse'

sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}

test {
    systemProperties 'property': 'value'
}

uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

7.3. マルチプロジェクトのJavaビルド

さて、ここで典型的なマルチプロジェクトのビルドを見てみましょう。プロジェクトのレイアウトは以下

例7.10 マルチプロジェクトビルド - 階層レイアウト

Build layout

```
multiproject/
  api/
  services/webservice/
  shared/
  services/shared/
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/java/multiproject

この例には三つのプロジェクトがあります。プロジェクトapi
は、このwebサービス用のJavaクライアントを提供するために顧客に出荷されるJARファイルを生成しま
webserviceはXMLを返すwebアプリケーションです。プロジェクトsharedは、apiとwebservice
の両方で使われる共通コードです。

7.3.1. マルチプロジェクトビルドの定義

マルチプロジェクトのビルドを定義するためには、settingsファイル
を作成する必要があります。settingsファイルはソースツリーのルートディレクトリに置かれ、このビル
settings.gradle
でなければなりません。この例では、シンプルな階層レイアウトを使っています。対応するsettingsファ

例7.11 マルチプロジェクトビルド - settings.gradleファイル

settings.gradle

```
include "shared", "api", "services:webservice", "services:shared"
```

settingsファイルの詳細は57章マルチプロジェクトのビルドをご覧ください。

7.3.2. 共通設定

大抵のマルチプロジェクトビルドでは、すべてのプロジェクトに共通な設定があります。この例では、
設定のインジェクション
と呼ばれる技法を使い、こういった共通設定をルートプロジェクトで定義します。ここでは、ルートプロ
subprojects
メソッドはコンテナの全要素、すなわちこの実例の中の全プロジェクト、に対して反復しつつ指定された

例7.12 マルチプロジェクトビルド - 共通設定

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'eclipse-wtp'

    repositories {
       mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.11'
    }

    version = '1.0'

    jar {
        manifest.attributes provider: 'gradle'
    }
}
```

この例で、各サブプロジェクトにJavaプラグインが適用されていることに注意してください。
これは、前の節で述べたタスクや設定プロパティが各サブプロジェクトでも利用可能であることを意味し

そんなわけで、ルートプロジェクトディレクトリから**gradle build**を実行することによって、すべてのプロジェクトをコンパイルし、テストし、JARを生成できるのです。

Also note that these plugins are only applied within the `subprojects` section, not at the root level, so the root build will not expect to find Java source files in the root project, only in the subprojects.

7.3.3. プロジェクト間の依存関係

同じビルド内のプロジェクト間の依存関係を追加することもできます。したがって例えば、一つのプロシ
a p i の ビ ル ド フ ァ イ ル に s h a r e d
プロジェクトが作るJARへの依存関係を追加します。この依存関係のため、Gradleは`shared`
プロジェクトが`api`プロジェクトより前にビルドされることを確実にします。

例7.13 マルチプロジェクトビルド - プロジェクト間の依存関係

api/build.gradle

```
dependencies {
    compile project(':shared')
}
```

この機能を無効にする方法については「依存プロジェクトをビルドしないようにする」
を参照してください。

7.3.4. 配布物の作成

顧客に出荷する配布物も追加しておきましょう：

例7.14 マルチプロジェクトビルド - 配布ファイル

api/build.gradle

```
task dist(type: Zip) {
    dependsOn spiJar
    from 'src/dist'
    into('libs') {
        from spiJar.archivePath
        from configurations.runtime
    }
}

artifacts {
   archives dist
}
```

7.4. 次のステップは？

この章では、Javaベースのプロジェクトをビルドするときに、一般的に必要になる作業のやり方を見てき
23章Javaプラグインで詳しく知ることができます。また、Gradleの配布物に含まれる`sample/java`
にはもっと多くのサンプルがあります。

そうでない場合は、続けて8章依存関係管理の基本 に進んでください。

8
依存関係管理の基本

本章では、Gradleにおける依存関係の管理について、基本的なところを紹介します。

8.1. 依存関係の管理とは

おおざっぱに言って、依存関係の管理とは二つの部分に分けられます。まず、Gradleは、あなたのプロシ
依 存 関 係 と 呼 び ま す 。
次に、Gradleはあなたのプロジェクトをビルドし、生成されたものをアップロードする必要があります。
公開物と呼びます。

ほとんどのプロジェクトは、完全に自己完結しているわけではありません。コンパイル、テスト、その他

これらの入力ファイルは、プロジェクトの依存関係から読み込まれます。
Gradleにプロジェクトの依存関係を教えて、プロジェクトが依存しているファイルを見つけたり、ビルト
依存ファイルは、リモートのMavenやIvyのリポジトリからダウンロードされるかもしれませんし、ロー‌

Often, the dependencies of a project will themselves have dependencies. For example, Hibernate core requires several other libraries to be present on the classpath with it runs. So, when Gradle runs the tests for your project, it also needs to find these dependencies and make them available. We call these transitive dependencies.

また、依存しているファイルが、それ自身の依存関係を持っていることが多々あります。例えば、Hiber
推移的な依存関係と呼びます。

ほとんどのプロジェクトは、ファイルを作成し、それらをプロジェクトの外部で使うということを主な目
例えば、もしあなたのプロジェクトがJavaライブラリを作成するものならば、あなたはjarを作成し、とき

ファイルの公開は、プロジェクトの公開設定から読み込まれて行われます。Gradleは、この重要な仕事も
「公開する」ということの正確な意味は、何をしたいのかによって異なります。ローカルディレクトリに
これらの処理を、公開と呼んでいます。

8.2. 依存関係の宣言

さて、では依存関係の宣言についていくつか見ていきましょう。ここに、ある基本的なビルドスクリプト

例8.1 依存関係の宣言

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

一体、何が起きるでしょうか。このビルドスクリプトにはそれほど多くの情報はありません。まず、この
core　　　　　　　　3.6.7を使用すると言っています。それはすなわち、実行時にHibernate
coreとその依存関係が必要になるということでもあります。
このスクリプトはまた、バージョン4.0以上のJUnitのどれかを、プロジェクトのテストをコンパイルする

8.3. 依存関係のコンフィグレーション

Gradleでは、依存関係は、コンフィグレーション
によってグループ化されます。コンフィグレーションとは、単なる名前の付いた依存関係の集まりです。
依存関係設定
として参照されます。プロジェクトが外部に依存しているものを宣言するのに使われ、また後で見るよう

Javaプラグインは、いくつかの標準コンフィグレーションを定義します。これらのコンフィグレーション
表23.5「Javaプラグイン - 依存関係のコンフィギュレーション」を参照してください。

compile
: プロジェクトのプロダクトコードをコンパイルするのに必要な依存関係

runtime
: プロダクトのクラスを実行するときに必要になる依存関係。デフォルトで、コンパイル時の依存関係

testCompile
: プロジェクトのテストコードをコンパイルするのに必要な依存関係。デフォルトで、コンパイルされ

testRuntime
: テストを実行するのに必要な依存関係。デフォルトで、compile、runtime、testCompileの各依存関

多くのプラグインが、このような標準コンフィグレーションを追加します。また、自分のビルドに使うた
「依存関係のコンフィギュレーション 」を参照してください。

8.4. 外部依存関係

定義できる依存関係には、さまざまなタイプがあります。そのうちの一つが、外部依存関係
です。これは、今のビルドとは別に作られ、MavenセントラルリポジトリやIvyリポジトリ、ローカルフ

外部依存関係を定義するには、それを依存関係のコンフィグレーションに追加します。

例8.2 外部依存関係の定義

build.gradle

```
dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
}
```

外部依存関係は、group属性、name属性、そしてversion属性を用いて定義します。
使用しているリポジトリの種類によっては、group属性とversion
属性はオプショナルになるかもしれません。

The shortcut form for declaring external dependencies looks like “*group:name:version*”.

外部依存関係を定義するときは、ショートカット形式を使うこともできます。"*group:name:versior*
という文字列を使う形式です。

例8.3 外部依存関係定義のショートカット形式

build.gradle

```
dependencies {
    compile 'org.hibernate:hibernate-core:3.6.7.Final'
}
```

依存関係を定義して取り扱う方法について、詳しくは「依存関係の定義方法 」を見てください。

8.5. リポジトリ

Gradleは、どうやって外部依存関係を見つけるのでしょうか？　　Gradleは、それらをリポジトリ
から探します。リポジトリの実体は、ただのファイルの集合です。それらのファイルは、、、そして
によって分類されています。Gradleは、MavenやIvyなど、様々なリポジトリ形式に対応できます。また

デフォルトでは、Gradleはリポジトリを一切定義していません。外部依存関係を使うには、少なくとも
選択肢の一つは、Mavenのセントラルリポジトリを使うことです。

例8.4 Mavenセントラルリポジトリの使用

build.gradle

```
repositories {
    mavenCentral()
}
```

または、別のMavenリモートリポジトリを使います。

例8.5 リモートMavenリポジトリの使用

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }
}
```

または、リモートのIvyリポジトリを使います。

例8.6 リモートIvyリポジトリの使用

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
    }
}
```

さらに、ローカルファイルシステム上にリポジトリを持つこともできます。これは、Maven、Ivy双方の

例8.7 ローカルのIvyディレクトリを使う

build.gradle

```
repositories {
    ivy {
        // URL can refer to a local directory
        url "../local-repo"
    }
}
```

一つのプロジェクトが、複数のリポジトリを持つことができます。Gradleは、それぞれのリポジトリを、

リポジトリを定義し、取り扱う方法について、詳しくは「リポジトリ 」を参照してください。

8.6. アーティファクトの公開

依存関係のコンフィグレーションは、ファイルを公開するのにも使用します。[6]
これらのファイルを、公開アーティファクト、大体は単にアーティファクトと呼びます。

プラグインが非常に適切にプロジェクトのアーティファクトを定義してくれるので、何を公開したいかG
uploadArchives
タスクに結びつけることで行います。次のコードは、リモートのIvyリポジトリに公開する例です。

例8.8 Ivyリポジトリに公開する

build.gradle

```
uploadArchives {
    repositories {
        ivy {
            credentials {
                username "username"
                password "pw"
            }
            url "http://repo.mycompany.com"
        }
    }
}
```

これで**gradle uploadArchives**を走らせれば、Gradleがjarをビルドしてアップロードしてくれます。 さらに、`ivy.xml`も生成されてアップロードされます。

Mavenリポジトリに公開することもできますが、その文法はすこし異なります[8]。
また、Mavenリポジトリへ公開するには、Mavenプラグインを使う必要もあるので注意してください。こ
`pom.xml`も生成してアップロードします。

例8.9 Mavenリポジトリへの公開

build.gradle

```
apply plugin: 'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
        }
    }
}
```

アーティファクトの公開について、詳しくは52章アーティファクトの公開 をご参照ください。

8.7. 次のステップは？

依存関係解決については、51章依存関係の管理
に全て詳細に載っています。そして、アーティファクトの公開については52章
アーティファクトの公開 に載っています。

本章で使ったDSLに興味があるのであれば、`Project.configurations{}`や
`Project.repositories{}`、`Project.dependencies{}`を見てください。

それ以外の場合は、チュートリアルから他のチュートリアルを進めてください。

私たちは、これは混乱の元になると考えています。そのため、GradleのDSLでは、段階的に二つに概念に

私たちは、8Mavenリポジトリからの取得とMavenリポジトリへの公開を一貫した文法で行えるよう作業し

9
Groovyクイックスタート

Groovyプロジェクトをビルドするのには、Groovyプラグイン
を使います。このプラグインはJavaプラグインを拡張して、Groovyのコンパイル能力をプロジェクトにた
プロジェクトはGroovyソースコード、Javaソースコード、またはこれらの両方を含むことができます。こ
7章Javaクイックスタートですでに見たJavaプロジェクトを同じです。

9.1. 基本的なGroovyプロジェクト

実例を見てみましょう。Groovyプラグインを使うためには、ビルドファイルに以下を記述を追加します

例9.1 Groovyプラグイン

build.gradle

```
apply plugin: 'groovy'
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/groovy/quickstart

これにより、（もしまだなら）プロジェクトにはJavaプラグインも適用されます。Groovyプラグインは
`compile`タスクを拡張して`src/main/groovy`
ディレクトリにあるソースファイルも参照するようにし、また`compileTest`タスクを拡張して`src/tes`
ディレクトリにあるテスト用ソースファイルも参照するようにします。compileタスクはこれらのディレ

Groovy用のcompileタスクを使うには、Groovyのバージョンと Groovyライブラリを探してくる場所を宣
`compile`
コンフィギュレーションに依存関係を追加することによって行います。この例では、Groovy
2.2.0を使用します。

例9.2 Dependency on Groovy

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.6'
}
```

以下にビルドファイルの全体を示します：

例9.3 Groovy用のビルドファイル（全体）

build.gradle

```
apply plugin: 'eclipse'
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.6'
    testCompile 'junit:junit:4.11'
}
```

gradle buildを実行すれば、プロジェクトがコンパイル、テストされ、JARが生成されます。

9.2. まとめ

この章では非常にシンプルなGroovyプロジェクトについて説明しました。通常、実プロジェクトではこ(
でもあるので、Javaプロジェクトで可能なことなら何でもGroovyプロジェクトでも可能です。

２４章Ｇｒｏｏｖｙプラグイン
では、Groovyプラグインをより詳しく説明しています。また、Gradle配布物の`samples/groovy`
ディレクトリには、Groovyプロジェクトの実例が多く含まれています。

10
Webアプリケーションクイックスタート

この章は執筆途中です。

この章ではGradleのWebアプリケーションサポートについて紹介します。
GradleはWebアプリケーション開発に対応する2つのプラグイン、WarプラグインとJettyプラグインを提
WarプラグインはJavaプラグインを拡張して、プロジェクトでWARファイルのビルドができるようにしま
JettyプラグインはWarプラグインを拡張して、組み込みJettyコンテナにWebアプリケーションをデプロイ

10.1. WARファイルのビルド

WARファイルをビルドするために、プロジェクトにWarプラグインを適用します:

例10.1 Warプラグイン

build.gradle

```
apply plugin: 'war'
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/webApplication/quickstart

これにより、プロジェクトにはJavaプラグインも適用されます。 **gradle build** を実行すれば、プロジェクトがコンパイル、テストされ、WARが生成されます。 Gradleは`src/main/w`にあるファイルからWARに含めるものを探します。
コンパイル済みクラスや実行時に必要となる依存ライブラリもWARに追加されます。

10.2. Webアプリケーションの実行

Groovy Webアプリケーション

単一のプロジェクト内で複数のプラ
GroovyベースのWebアプリケーショ
その場合、適切なGroovyライブラリ

Webアプリケーションを実行するためには、プロジェクトにJettyプラグインを適用します:

例10.2 JettyプラグインによるWebアプリケーションの実行

build.gradle

```
apply plugin: 'jetty'
```

これにより、プロジェクトにはWarプラグインも適用されます。 **gradle jettyRun** を実行すれば、組み込みJettyコンテナ上でWebアプリケーションが実行されます。 **gradle jettyRunWar** を実行するとWARファイルが生成され、その後組み込みWebコンテナで実行されます。

TODO: URLの確認方法、ポートの設定、ソースファイルの編集と動的リロードの方法など

10.3. まとめ

Warプラグインについての詳細は26章War プラグイン、 Jettyプラグインの詳細については28章Jettyプラグインを参照してください。 Gradle配布物の`samples/webApplication`ディレクトリにサンプルJavaプロジェクトがあります。

11 Gradleのコマンドラインを使う

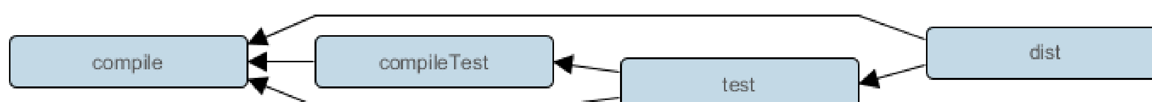
本章では、Gradleのコマンドラインについて基本的な部分を紹介します。これまで動かしてきたように、gradleコマンドで実行します。

11.1. 複数のタスクを実行する

一回のビルドで、複数のタスクを実行することができます。ビルドを実行するときに、タスクを並べて指 **gradle compile test**というコマンドを実行すれば、`compile`タスクと`test`タスクが実行されます。Gradleはタスクを列挙された順に実行するほか、それらが依存しているタスクも

この例では、四つのタスクが定義されており、`dist`と`test`はどちらも`compile`タスクに依存しています。このビルドスクリプトを呼び出して**gradle dist test**と実行しても、実行された二つのタスクから依存されている`compile`タスクは一度だけしか実行されません。

図11.1 Task間の依存関係



例11.1 複数のタスクの実行

build.gradle

```
task compile << {
    println 'compiling source'
}

task compileTest(dependsOn: compile) << {
    println 'compiling unit tests'
}

task test(dependsOn: [compile, compileTest]) << {
    println 'running unit tests'
}

task dist(dependsOn: [compile, test]) << {
    println 'building the distribution'
}
```

gradle dist test の出力

```
> gradle dist test
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

タスクが一度だけ実行されるということは、**gradle test test**と打ち込んでも**gradle test**と打ち込んでも、実行される処理は全く同じだということです。

11.2. タスクを除外してビルドする

- x
オプションでタスクの名前を指定することで、そのタスクを除外してビルドを実行することができます。

例11.2 タスクの除外

gradle dist -x test の出力

```
> gradle dist -x test
:compile
compiling source
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

distタスクはtestに依存しているのですが、この例ではtest
タスクは実行されていないことが分かります。また、testタスクが依存しているタスク、たとえばcomp
タスクもやはり実行されていません。しかし、testタスクが依存しているタスクのうち、test
以外からも依存されているタスク、つまりcompileタスクは実行されているのです。

11.3. エラー発生時にビルドを継続する

デフォルトでは、タスクが失敗するとすぐにGradleは実行を中止してビルドを失敗させます。これはビル
一回のビルド実行でできるだけ多くのエラーを発見するには、--continue
オプションを使用します。

- - c o n t i n u e
オプションをつけてビルドすると、エラー発生時に即ビルドが停止されるのではなく、失敗せずに完了し
全 て 実 行 さ れ る よ う に な り ま す 。
また、ビルド中に発生したエラーはビルド終了時に全てレポートされます。

タスクが失敗した場合、依存関係上の後続タスクは安全に実行できないため行われません。
例えば、テストの際、コードのコンパイルでエラーが発生するとテストは実施されません。テストタスク

11.4. タスク名の省略

タスク名をコマンドラインで指定するとき、タスク名をすべて入力する必要はありません。タスク名を一
gradle dと入力すればdistを実行できます。

例11.3 タスク名の省略

gradle di の出力

```
> gradle di
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

キャメルケースで分割されたそれぞれの単語を省略することもできます。たとえば、compileTestというタスクだと、**gradle compTest**と入力したり、**gradle cT**と入力したりすることで実行できます。

例11.4 キャメルケースのタスク名を省略

gradle cT の出力

```
> gradle cT
:compile
compiling source
:compileTest
compiling unit tests

BUILD SUCCESSFUL

Total time: 1 secs
```

-xオプションで除外タスクを指定するときにも、同じようにタスク名を省略できます。

11.5. ビルドスクリプトを指定して実行する

gradleコマンドは、デフォルトではカレントディレクトリにあるビルドスクリプトを探して実行しますが、-bオプションを使えば別のビルドスクリプトを指定することもできます。次の例を見てください。なお、-bを使うと、settings.gradleファイルは使用されなくなります。

例11.5 ビルドスクリプトを指定してビルドするプロジェクトを選択する

subdir/myproject.gradle

```
task hello << {
    println "using build file '$buildFile.name' in '$buildFile.parentFile.name'."
}
```

gradle -q -b subdir/myproject.gradle hello の出力

```
> gradle -q -b subdir/myproject.gradle hello
using build file 'myproject.gradle' in 'subdir'.
```

別 の 方 法 と し て 、 -p
オプションを指定して、使用するプロジェクトディレクトリを変更することもできます。
マルチプロジェクトの場合は、-bオプションでなく-pオプションを使用するべきです。

例11.6 プロジェクトディレクトリを使ってプロジェクトを選択する

gradle -q -p subdir hello の出力

```
> gradle -q -p subdir hello
using build file 'build.gradle' in 'subdir'.
```

11.6. ビルドに関する情報を取得する

ビルドに関する様々な情報を表示するために、組み込みのタスクがいくつか用意されています。自分の作

この組み込みタスクに加えて、プロジェクトレポートプラグイン
というプラグインを使うこともできます。このプラグインは、これらの情報をレポートに保存するタスク

11.6.1. プロジェクトの一覧

g r a d l e p r o j e c t s
を実行すると、指定したプロジェクトのサブプロジェクトを一覧できます。プロジェクトの一覧は階層構

例11.7 プロジェクトに関する情報を取得する

gradle -q projects の出力

```
> gradle -q projects
------------------------------------------------------------
Root project
------------------------------------------------------------

Root project 'projectReports'
+--- Project ':api' - The shared API for the application
\--- Project ':webapp' - The Web application implementation

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

このレポートは、プロジェクトに説明書きが添付されている場合、その説明も表示されます。プロジェク

例11.8 プロジェクトに説明を添付する

build.gradle

```
description = 'The shared API for the application'
```

11.6.2. タスクの一覧

gradle tasks を実行すると、プロジェクトに設定されているメインタスクの一覧とデフォルトタスクを表示します。さ

例11.9 タスクに関する情報を取得する

gradle -q tasks の出力

```
> gradle -q tasks
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Default tasks: dists

Build tasks
-----------
clean - Deletes the build directory (build)
dists - Builds the distribution
libs - Builds the JAR

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
components - Displays the components produced by root project 'projectReports'.
dependencies - Displays all dependencies declared in root project 'projectReports'.
dependencyInsight - Displays the insight into a specific dependency in root project ':
help - Displays a help message.
projects - Displays the sub-projects of root project 'projectReports'.
properties - Displays the properties of root project 'projectReports'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the d

To see all tasks and more detail, run with --all.
```

デフォルトでは、このレポートは何らかのタスクグループに属しているタスクしか表示されません。タス
group
プロパティを設定することでタスクにグループを割り当てることができます。また、このレポートでタス
descriptionプロパティに説明書きを設定してください。

例11.10 タスクレポートの内容を変更する

build.gradle

```
dists {
    description = 'Builds the distribution'
    group = 'build'
}
```

- - a l l
オプションを追加すると、タスクリストについてもっと多くの情報を取得できます。プロジェクトにある

例11.11 タスクに関してもっと多くの情報を取得する

gradle -q tasks --all の出力

```
> gradle -q tasks --all
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Default tasks: dists

Build tasks
-----------
clean - Deletes the build directory (build)
api:clean - Deletes the build directory (build)
webapp:clean - Deletes the build directory (build)
dists - Builds the distribution [api:libs, webapp:libs]
    docs - Builds the documentation
api:libs - Builds the JAR
    api:compile - Compiles the source files
webapp:libs - Builds the JAR [api:libs]
    webapp:compile - Compiles the source files

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
components - Displays the components produced by root project 'projectReports'.
api:components - Displays the components produced by project ':api'.
webapp:components - Displays the components produced by project ':webapp'.
dependencies - Displays all dependencies declared in root project 'projectReports'.
api:dependencies - Displays all dependencies declared in project ':api'.
webapp:dependencies - Displays all dependencies declared in project ':webapp'.
dependencyInsight - Displays the insight into a specific dependency in root project ':
api:dependencyInsight - Displays the insight into a specific dependency in project ':
webapp:dependencyInsight - Displays the insight into a specific dependency in project
help - Displays a help message.
api:help - Displays a help message.
webapp:help - Displays a help message.
projects - Displays the sub-projects of root project 'projectReports'.
api:projects - Displays the sub-projects of project ':api'.
webapp:projects - Displays the sub-projects of project ':webapp'.
properties - Displays the properties of root project 'projectReports'.
api:properties - Displays the properties of project ':api'.
webapp:properties - Displays the properties of project ':webapp'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the d
api:tasks - Displays the tasks runnable from project ':api'.
webapp:tasks - Displays the tasks runnable from project ':webapp'.
```

11.6.3. タスクの使い方を詳しく表示する

gradle help --task someTask を実行すると、ある特定のタスク、マルチプロジェクトの場合は指定した名前にマッチする全てのタスク
表示される情報の例は以下の通りです。

例11.12 タスクについて詳細なヘルプ情報を取得する

gradle -q help --task libs の出力

```
> gradle -q help --task libs
Detailed task information for libs

Paths
     :api:libs
     :webapp:libs

Type
     Task (org.gradle.api.Task)

Description
     Builds the JAR
```

上記の通り、表示される情報にはタスクのフルパス、使用可能なコマンドラインオプション、またタスク

11.6.4. プロジェクトの依存関係一覧

gradle dependencies はプロジェクトの依存関係をコンフィギュレーションごとに一覧表示します。プロジェクトに直接設定さ

例11.13 依存関係の情報を取得する

gradle -q dependencies api:dependencies webapp:dependencies の出力

```
> gradle -q dependencies api:dependencies webapp:dependencies
------------------------------------------------------------
Root project
------------------------------------------------------------

No configurations

------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

compile
\--- org.codehaus.groovy:groovy-all:2.3.6

testCompile
\--- junit:junit:4.11
     \--- org.hamcrest:hamcrest-core:1.3

------------------------------------------------------------
Project :webapp - The Web application implementation
------------------------------------------------------------

compile
+--- project :api
|    \--- org.codehaus.groovy:groovy-all:2.3.6
\--- commons-io:commons-io:1.2

testCompile
No dependencies
```

依存関係のレポートは巨大になることもあるので、レポート内容を特定のコンフィギュレーションのみに
これは、オプションの**--configuration**を指定することで実現できます。

例11.14 依存関係のレポートをコンフィギュレーションでフィルタする

gradle -q api:dependencies --configuration testCompile の出力

```
> gradle -q api:dependencies --configuration testCompile
------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

testCompile
\--- junit:junit:4.11
     \--- org.hamcrest:hamcrest-core:1.3
```

11.6.5. 個別の依存関係に対する解析情報を取得する

gradle dependencyInsight
を実行すると、個別の、または複数の依存関係を指定して、それぞれに対する解析情報を取得することか

例11.15 個別の依存関係に対する解析情報を取得する

gradle -q webapp:dependencyInsight --dependency groovy --configuration compile
の出力

```
> gradle -q webapp:dependencyInsight --dependency groovy --configuration compile
org.codehaus.groovy:groovy-all:2.3.6
\--- project :api
     \--- compile
```

dependencyInsightは、依存関係がどのように解決されているか調査するときにとても便利なタスクです
詳細については、`DependencyInsightReportTask`を参照してください。

組み込みのdependencyInsightタスクは、Helpグループに属しています。
このタスクは、依存関係およびコンフィグレーションを指定しないと使用できません。
レポートが出力される際は、指定されたコンフィグレーションの指定された依存関係を探しに行きます。
Java関連のプラグインが適用されていれば、dependencyInsightタスクは既にcompileコンフィグレーシ
また、調査対象の依存関係を「--dependency」オプションで指定しなければなりません。
さらに、デフォルトのコンフィグレーションが気に入らないときは「--configuration」オプションで明示
詳しくは`DependencyInsightReportTask`を参照してください。

11.6.6. プロジェクトのプロパティ一覧

gradle properties
はプロジェクトのプロパティを一覧表示します。出力例の一部を以下に示します。

例11.16 プロパティに関する情報

gradle -q api:properties の出力

```
> gradle -q api:properties
------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

allprojects: [project ':api']
ant: org.gradle.api.internal.project.DefaultAntBuilder@12345
antBuilderFactory: org.gradle.api.internal.project.DefaultAntBuilderFactory@12345
artifacts: org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler_Decorated@123
asDynamicObject: org.gradle.api.internal.ExtensibleDynamicObject@12345
baseClassLoaderScope: org.gradle.api.internal.initialization.DefaultClassLoaderScope@
buildDir: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build
buildFile: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build.grad
```

11.6.7. ビルドのプロファイリング

コマンドラインオプションで**--profile**
をつけてビルドを実行すると、ビルドに要した時間のうち有用と思われるいくつかの情報を計測して、結
build/reports/profile
ディレクトリを出力します。レポート名には、ビルドを行った時刻が付加されます。

このレポートでは、ビルド時間に関する概要のほか、設定フェーズとタスク実行に要した時間がタスクこ

なお、buildSrcディレクトリを使っている場合、buildSrc/buildにbuildSrc
のレポートが別途出力されます。

Profiled with tasks: -xtest build

Summary

Total Build Time	2:01.164
Startup	0.313
Settings and BuildSrc	4.078
Loading Projects	0.074
Configuring Projects	3.208
Total Task Execution	1:52.671

Configuration

:	2.804
:docs	0.576
:core	0.203
:announce	0.084
:ui	0.036
:openApi	0.035
:maven	0.033
:codeQuality	0.033
:wrapper	0.022
:eclipse	0.021
:idea	0.021
:plugins	0.020
:launcher	0.020
:antlr	0.017
:osgi	0.014
:jetty	0.014
:scala	0.012

Task E

:docs	
	:docs:userguideSingleHtml
	:docs:userguidePdf
	:docs:checkstyleApi
	:docs:userguideStyleSheets
	:docs:groovydoc
	:docs:samples
	:docs:javadoc
	:docs:userguideFragmentS
	:docs:distDocs
	:docs:samplesDocs
	:docs:userguideXhtml
	:docs:userguideHtml
	:docs:userguideDocbook
	:docs:remoteUserguideDo
	:docs:samplesDocbook
	:docs:docs
	:docs:userguide
:core	
	:core:compileTestGroovy
	:core:codenarcTest
	:core:checkstyleMain
	:core:compileTestJava

11.7. 空実行

コマンドラインにタスク名を並べて実行したときに、どのタスクがどの順番で実行されるのか知りたくな
-mオプションをつけて実行してください。たとえば、**gradle -m clean compile**
と実行すれば、cleanタスクとcompileタスクの実行にあたって実際に実行されるすべてのタスクが実行順
tasksタスクの補足にもなります。

11.8. まとめ

この章では、コマンドラインで使えるGradleの機能をいくつか見てきました。gradle
コマンドについては付録D Gradle コマンドラインにも記載してあります。

12 GradleのGUIを使う

伝統的なコマンドラインインターフェースに加え、Gradleはグラフィカルユーザーインターフェースも扌
--guiオプションを使って起動できます。

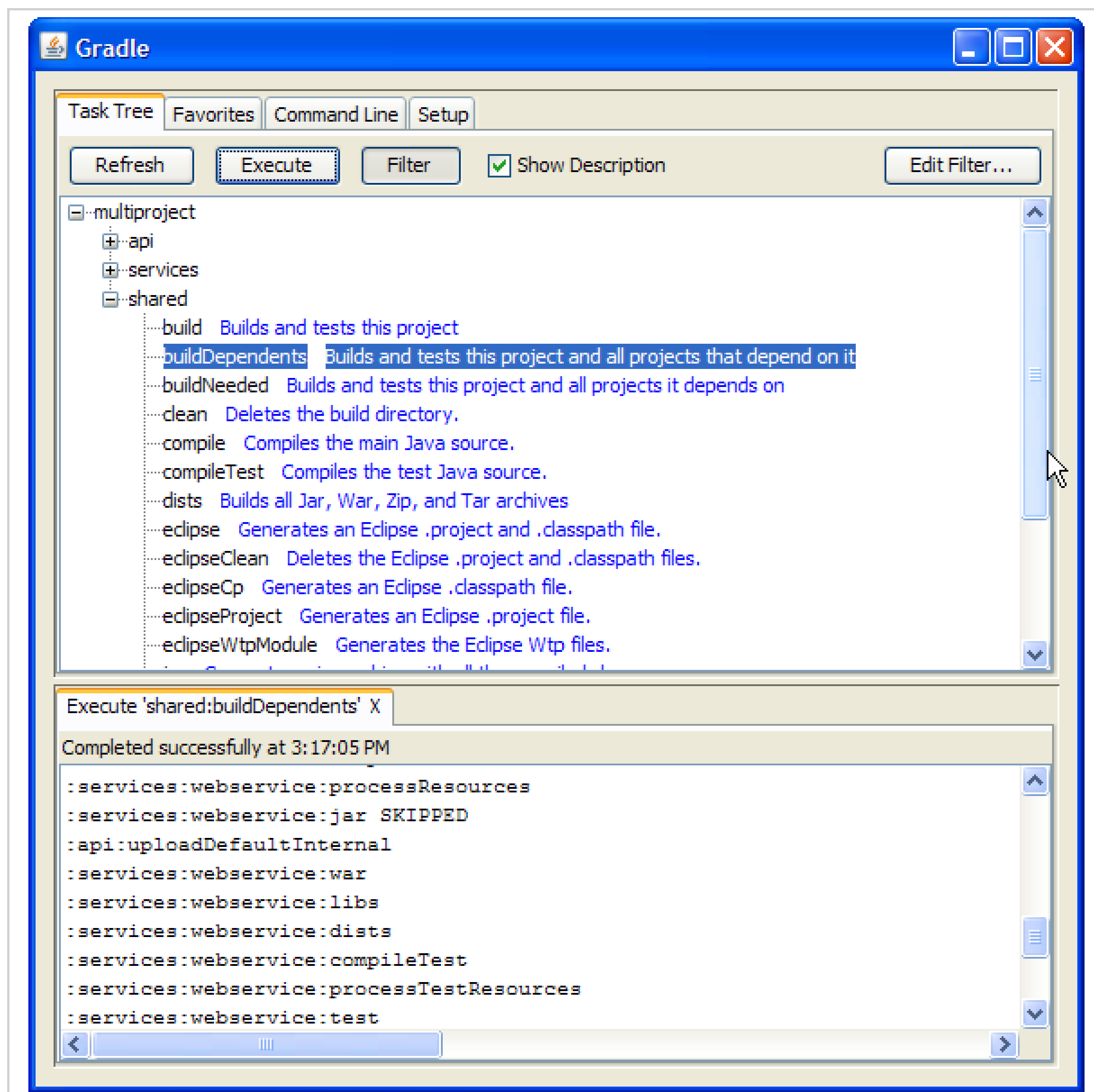
例12.1 GUIの起動

```
gradle --gui
```

このコマンドは、GUIをクローズするまでブロックされることに注意してください。*nixでは、バックグ
gradle --gui&)で実行した方がいいかもしれません。

このコマンドをgradleプロジェクトの作業ディレクトリで実行すると、タスクのツリーが表示されること

図12.1 GUI Task Tree



このコマンドは、自分のgradleプロジェクトのディレクトリで実行した方がいいでしょう。そうすれば、

UIには上部に4つのタブがあり、下部には出力を表示するウィンドウがあります。

12.1. Task Tree（タスク・ツリー）

T a s k
Treeは、すべてのプロジェクトとタスクの階層構造を表示します。タスクをダブルクリックすると、その

また、フィルターも用意されているので、一般的でないタスクは非表示にすることができます。フィルタ
フィルター編集画面では、どのタスクとプロジェクトを表示するかを設定します。非表示に設定された
注 意 :
新しく作成されたタスクは、デフォルトでは表示に設定されます（デフォルトで非表示に設定されるので

タスクツリーのコンテキストメニューでは、以下の機能も使うことができます。

- 依存関係を無視したタスクの実行する。依存プロジェクトを再ビルドしません（-aオプションと同じ
- タスクをお気に入りに追加する（Favoritesを参照してください）。
- 選択したタスクを隠す。タスクをフィルターに追加します。
- build.gradleファイルの編集。注意:
  この機能を使うには、1.6以上のバージョンのJavaが必要です。また、OS上で.gradleファイルの関連

12.2. Favorites（お気に入り）

Favoriteタブでは、よく実行するコマンドをお気に入りとして保存できます。保存するコマンドは複雑な

お気に入りは、好きなように並び替えることができます。また、お気に入りをディスクにエクスポートし
お気に入りを追加、編集するとき、"Always Show Live Output"というオプションがあるはずです。
これは、'Only Show Output When Errors
Occur'（エラー発生時のみ出力を表示する）設定を有効にしているときのみ適用されるオプションです。
このオプションを有効にすると、その設定を上書きし常に出力が表示されるようになります。

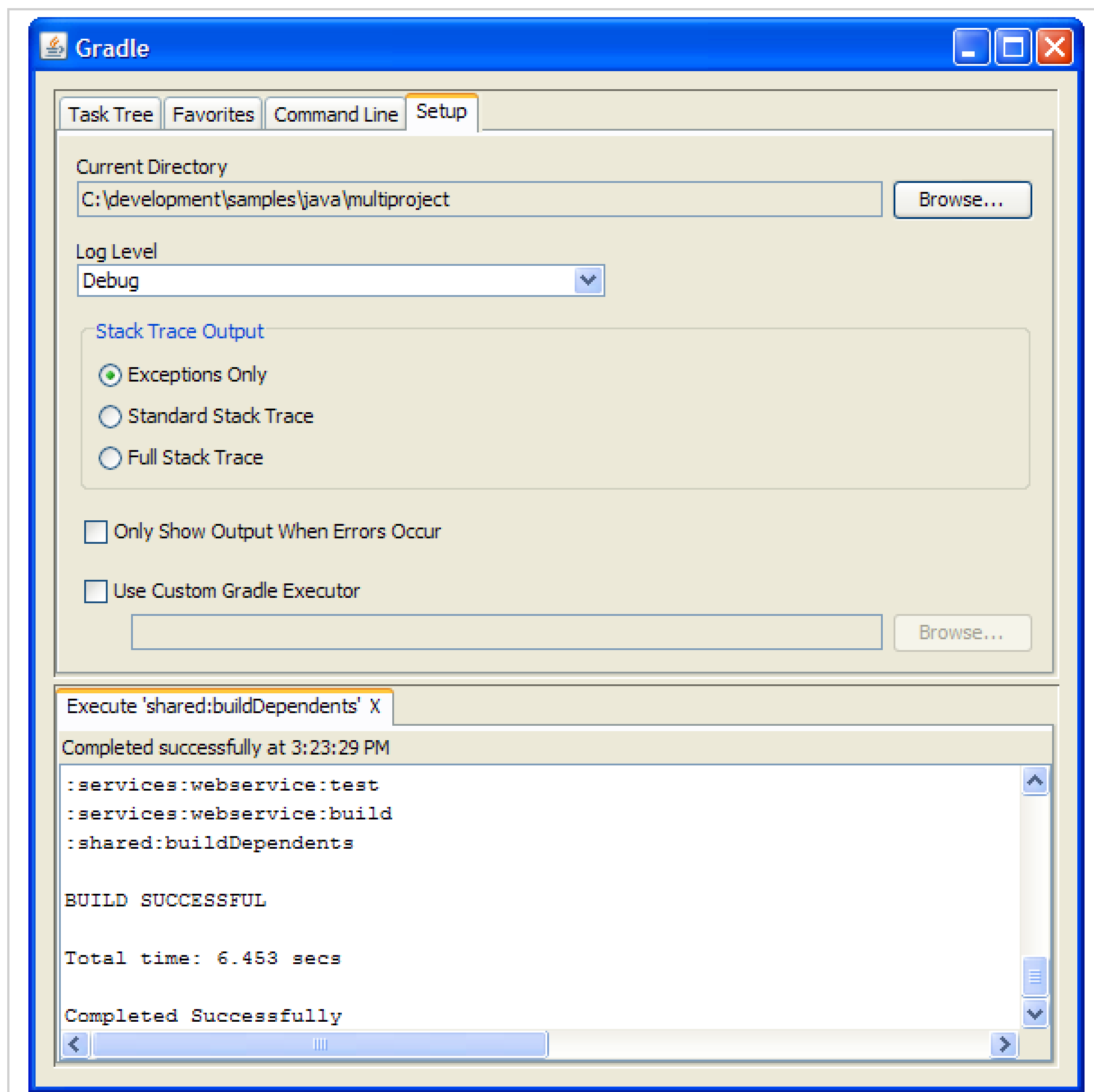
12.3. Command Line（コマンドライン）

Command
Lineタブでは、ひとつのgradleコマンドを直接実行できます。いつも'gradle'のあとに入力しているコマン

12.4. Setup（セットアップ）

Setupタブは、一般的な設定を行うところです。

図12.2 GUI Setup



- Current Directory
  gradleプロジェクトのルートディレクトリを設定します（典型的には、build.gradleが置かれているテ
- Stack Trace Output
  ここでは、エラーが発生したとき、どれだけの情報をスタックトレースに書き出すかを設定します。：
  スタックトレースレベルをCommand
  LineタブやFavoritesタブで指定している場合、ここで設定したスタックトレースレベルは上書きされ
- Only Show Output When Errors Occur
  有効にすると、ビルドに失敗しない限り出力が表示されないようになります。
- Use Custom Gradle Executor - 高度な機能
  gradleコマンドの実行方法を変更できます。これは、自分のプロジェクトが、バッチファイルやシェ

13
ビルドスクリプトの記述

本章では、ビルドスクリプトの記述についてもう少し詳しく見ていきます。

13.1. Gradleビルド言語

Gradleは、ビルドシステムを記述するためのドメイン特化言語(DSL)を提供します。
このビルド用言語は、Groovyをベースに、ビルドシステムの記述を簡単にするための機能をいくつか追ｶ

A build script can contain any Groovy language element. [9] Gradle assumes that each build script is encoded using UTF-8.

13.2. プロジェクトAPI

例えば、7章Javaクイックスタートのチュートリアルで使用した`apply()`
メソッド、このメソッドはどこから来たのでしょうか。まず、前述のようにビルドスクリプトはGradleの
ビルドに含まれるプロジェクトそれぞれに対し、`Project`
型のインスタンスが一つ作られ、ビルドスクリプトと関連づけられます。ビルドが実行されると、このﾌ

- ビルドスクリプトで呼び出されたメソッドのうち、そのスクリプトで定義されていないものは`Project`オブジェクトに委譲されます。
- ビルドスクリプトでのプロパティアクセスも、そのスクリプトで定義されていないプロパティについては`Project`オブジェクトに委譲されます。

ビルドスクリプトのリフ

ビルドスクリプトは、GradleのAPIを
`Project`
インターフェースが、GradleのAPIを
`Project`
インターフェースのドキュメントを見

プロジェクトオブジェクトの`name`
プロパティにアクセスして試してみましょう。

例13.1 Projectオブジェクトへのアクセス

build.gradle

```
println name
println project.name
```

gradle -q check の出力

```
> gradle -q check
projectApi
projectApi
```

どちらのprintln
も出力結果は同じです。最初のprintlnは、ビルドスクリプトで定義されていないプロパティへのアクセス
projectプロパティを使って、nameにアクセスしています。project
プロパティには、ビルドスクリプトのどこからでもアクセスできます。プロジェクトオブジェクトのプロ
projectプロパティからアクセスすることになるでしょう。

13.2.1. 標準のプロジェクトプロパティ

プロジェクトオブジェクトは標準でいくつかのプロパティを提供しており、ビルドスクリプトでこれらの

表13.1 プロジェクトプロパティ

プロパティ名	型	デフォルト値
project	Project	Projectインスタンス
name	String	プロジェクトディレクトリの名前
path	String	プロジェクトの絶対パス
description	String	プロジェクトの説明
projectDir	File	ビルドスクリプトのあるディレクトリ
buildDir	File	*projectDir*/build
group	Object	
version	Object	
ant	AntBuilder	AntBuilderインスタンス

13.3. スクリプトAPI

ビルドスクリプトが実行されるときは、スクリプトはScript
を実装したクラスにコンパイルされます。つまり、ビルドスクリプトではScript
インターフェースで宣言されているプロパティとメソッドを全て使うことができるということです。

13.4. 変数の宣言

ビルドスクリプトでは、二つの方法で変数を宣言できます。ローカル変数と拡張プロパティです。

13.4.1. ローカル変数

ローカル変数は、def
キーワードで宣言する変数です。変数のスコープは、変数が宣言されている場所に制限されます。なお、

例13.2 ローカル変数を使用する

build.gradle

```
def dest = "dest"

task copy(type: Copy) {
    from "source"
    into dest
}
```

13.4.2. 拡張プロパティ

Gradleのドメインモデルを構成する全ての拡張オブジェクトには、ユーザー定義のプロパティ、拡張プロ
拡張プロパティを追加できるオブジェクトには、projects、tasks、source
setsを始め、様々なオブジェクトがあります。
拡張プロパティをセットしたり読み込んだりするには、対象オブジェクトの`ext`
プロパティを使用して下さい。また、`ext`
ブロックを使って複数のプロパティを一気に追加することもできます。

例13.3 拡張プロパティを使用する

build.gradle

```
apply plugin: "java"

ext {
    springVersion = "3.1.0.RELEASE"
    emailNotification = "build@master.org"
}

sourceSets.all { ext.purpose = null }

sourceSets {
    main {
        purpose = "production"
    }
    test {
        purpose = "test"
    }
    plugin {
        purpose = "production"
    }
}

task printProperties << {
    println springVersion
    println emailNotification
    sourceSets.matching { it.purpose == "production" }.each { println it.name }
}
```

gradle -q printProperties の出力

```
> gradle -q printProperties
3.1.0.RELEASE
build@master.org
main
plugin
```

この例では、extブロックで二つの拡張プロパティをprojectオブジェクトに追加しています。
さらに、ext.purposeにnullをセットすることで（null
でも構いません）、全てのソースセットにpurposeという拡張プロパティを追加しています。
一度プロパティが追加されれば、定義済みのプロパティと同じように読み込んだり書き込んだりすること

For further details on extra properties and their API, see the `ExtraPropertiesExtension` class in the API documentation.

13.5. Groovyの基本

GroovyにはDSLを作るための機能が豊富に用意されていて、Gradleもその長所を活用しています。Grad

13.5.1. Groovy JDK

Groovyは、たくさんの便利なメソッドをJavaの標準クラスに追加しています。たとえば、`Iterable`には`each`というメソッドが追加されていて、要素を走査するときに次のように書くことができます。

例13.4 Groovy JDKのメソッド

build.gradle

```
// Iterable gets an each() method
configurations.runtime.each { File f -> println f }
```

詳細はhttp://groovy.codehaus.org/groovy-jdk/をご参照ください。

13.5.2. プロパティのアクセサ

Groovyは、プロパティ参照を自動的にゲッター/セッターメソッドの呼び出しに変換してくれます。

例13.5 プロパティアクセサ

build.gradle

```
// Using a getter method
println project.buildDir
println getProject().getBuildDir()

// Using a setter method
project.buildDir = 'target'
getProject().setBuildDir('target')
```

13.5.3. メソッド呼び出し時のカッコを省略できる

Groovyではメソッド呼び出しのカッコは省略できます。

例13.6 カッコなしのメソッド呼び出し

build.gradle

```
test.systemProperty 'some.prop', 'value'
test.systemProperty('some.prop', 'value')
```

13.5.4. リストリテラル、マップリテラル

Groovyでは、`List`や`Map`のインスタンスを作るときにショートカット記法が使えます。

For instance, the “`apply`” method (where you typically apply plugins) actually takes a map parameter. However, when you have a line like “`apply plugin:'java'`”, you aren't actually using a map literal, you're actually using “named parameters”, which have almost exactly the same syntax as a map literal (without the wrapping brackets). That named parameter list gets converted to a map when the method is called, but it doesn't start out as a map.

例13.7 マップリテラル、リストリテラル

build.gradle

```
// List literal
test.includes = ['org/gradle/api/**', 'org/gradle/internal/**']

List<String> list = new ArrayList<String>()
list.add('org/gradle/api/**')
list.add('org/gradle/internal/**')
test.includes = list

// Map literal.
Map<String, String> map = [key1:'value1', key2: 'value2']

// Groovy will coerce named arguments
// into a single map argument
apply plugin: 'java'
```

13.5.5. メソッド引数の最後にクロージャを使う

GradleのDSLでは、クロージャ(詳細はこちら
)を多用します。Groovyでは、メソッドの最後の引数がクロージャになっている場合、そのメソッドをク

例13.8 メソッドのクロージャ引数

build.gradle

```
repositories {
    println "in a closure"
}
repositories() { println "in a closure" }
repositories({ println "in a closure" })
```

13.5.6. クロージャのdelegateオブジェクト

すべてのクロージャは、delegate
オブジェクトを持っています。クロージャ内では、(クロージャ引数やローカル変数以外の)変数やメソッ
delegateオブジェクトから検索されます。Gradleの設定用クロージャでは、delegate
オブジェクトに設定対象のオブジェクトがセットされています。

例13.9 クロージャのdelegate

build.gradle

```
dependencies {
    assert delegate == project.dependencies
    testCompile('junit:junit:4.11')
    delegate.testCompile('junit:junit:4.11')
}
```

---

[9] Any language element except for statement labels.

14
色々なチュートリアル

14.1. ディレクトリの作成

複数のタスクが、ある一つのディレクトリが存在していることを前提としている、ということがよくあり
それらのタスク全てで`mkdir`を呼ぶことももちろんできますが、それではやはり冗長です。
この場合、次のように、ディレクトリを必要とするタスクにdependsOn関係を追加するのが良い方法で

例14.1 mkdirでディレクトリを作成する

build.gradle

```
def classesDir = new File('build/classes')

task resources << {
    classesDir.mkdirs()
    // do something
}
task compile(dependsOn: 'resources') << {
    if (classesDir.isDirectory()) {
        println 'The class directory exists. I can operate'
    }
    // do something
}
```

gradle -q compile の出力

```
> gradle -q compile
The class directory exists. I can operate
```

14.2. Gradleプロパティとシステムプロパティ

Gradleでは、さまざまな方法でビルドにプロパティ値を渡すことができます。 `-D`
オプションを使えばGradleを実行しているJVMにシステムプロパティを渡すことができます。
gradleコマンドは`-D`オプションをjavaコマンドへの`-D`オプションと同じように処理します。

propertiesファイルを記述することで、projectオブジェクトに直接プロパティを追加することもできます
このプロパティファイル`gradle.properties`はGradleのホームディレクトリ (デフォルトでは*USER_H*
) かプロジェクトディレクトリに作成します。
マルチプロジェクトの場合、gradle.properteisはすべてのサブプロジェクトディレクトリに置くことがで
`gradle.properties`

に記述されたプロパティはprojectオブジェクトからアクセスすることができます。
なお、ホームディレクトリのgradle.propertiesは、プロジェクトディレクトリのgradle.propert
の定義を上書きし、優先的に使用されます。

projectオブジェクトに追加するプロパティは、コマンドラインオプションの-P
オプションで直接設定することもできます。

Gradle can also set project properties when it sees specially-named system properties or environment variables. This feature is very useful when you don't have admin rights to a continuous integration server and you need to set property values that should not be easily visible, typically for security reasons. In that situation, you can't use the -P option, and you can't change the system-level configuration files. The correct strategy is to change the configuration of your continuous integration build job, adding an environment variable setting that matches an expected pattern. This won't be visible to normal users on the system. [10]

projectオブジェクトに追加するプロパティは、コマンドラインオプションの-P
オプションで設定することもできます。また、さらにエキゾチックな使い方に備えて、環境変数やシステ
直　　　　　　　　　接
取り込む機能もあります。たとえば、管理権限のないマシン上のCIサーバーで継続的なビルドを行う場合
-　　　　　　　　　P
オプションは使えません。このときは、プロジェクトの管理者のみアクセスできるようなCIサーバー上の
[11] ここで設定する環境変数名にはあるパターンがあり、ORG_GRADLE_PROJECT_*propertyName*=sor
という環境変数を設定した場合propertyName
というプロパティがプロジェクトに設定されます。また、システムプロパティでも同様のメカニズムを実
org.gradle.project.*propertyName*となっていること以外は環境変数の場合と同じです。

If the environment variable name looks like ORG_GRADLE_PROJECT_*prop*=somevalue, then Gradle will set a prop property on your project object, with the value of somevalue. Gradle also supports this for system properties, but with a different naming pattern, which looks like org.gradl
.

You can also set system properties in the gradle.properties file. If a property name in such a file has the prefix “systemProp.”, like “systemProp.propName”, then the property and its value will be set as a system property, without the prefix. In a multi project build, “systemProp.
” properties set in any project except the root will be ignored. That is, only the root project's gradl
file will be checked for properties that begin with the “systemProp.” prefix.

gradle.propertiesファイルでもシステムプロパティを設定することができます。
ファイル内でプレフィックスにsystemProp
をつけてプロパティを設定することで、プレフィックス部分をのぞいたプロパティ名でシステムプロパテ

例14.2 gradle.propertiesでプロパティを設定する

gradle.properties

```
gradlePropertiesProp=gradlePropertiesValue
sysProp=shouldBeOverWrittenBySysProp
envProjectProp=shouldBeOverWrittenByEnvProp
systemProp.system=systemValue
```

build.gradle

```
task printProps << {
    println commandLineProjectProp
    println gradlePropertiesProp
    println systemProjectProp
    println envProjectProp
    println System.properties['system']
}
```

gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.pr
の出力

```
> gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project
commandLineProjectPropValue
gradlePropertiesValue
systemPropertyValue
envPropertyValue
systemValue
```

14.2.1. プロジェクトプロパティの確認

プロジェクトに設定されたプロパティは、ビルドスクリプト内では単にプロパティ名でアクセスできます
`hasProperty('propertyName')`
がtrueやfalseを返すので、それでプロパティの存在を確認できます。

14.3. 外部のビルドスクリプトをプロジェクトに取り込

外部のビルドスクリプトを取り込んでプロジェクトを設定することができます。取り込む外部スクリプト

例14.3 外部のビルドスクリプトファイルでプロジェクトの設定を行う

build.gradle

```
apply from: 'other.gradle'
```

other.gradle

```
println "configuring $project"
task hello << {
    println 'hello from other script'
}
```

gradle -q hello の出力

```
> gradle -q hello
configuring root project 'configureProjectUsingScript'
hello from other script
```

14.4. 任意のオブジェクトを組み立てる

次のような非常に読みやすい方法で、あらゆるオブジェクトを設定することができます。

例14.4 任意のオブジェクトを組み立てる

build.gradle

```
task configure << {
    def pos = configure(new java.text.FieldPosition(10)) {
        beginIndex = 1
        endIndex = 5
    }
    println pos.beginIndex
    println pos.endIndex
}
```

gradle -q configure の出力

```
> gradle -q configure
1
5
```

14.5. 外部スクリプトで任意のオブジェクトを組み立て

外部スクリプトを使って、任意のオブジェクトを組み立てることもできます。

例14.5 外部スクリプトで任意のオブジェクトを組み立てる

build.gradle

```
task configure << {
    def pos = new java.text.FieldPosition(10)
    // Apply the script
    apply from: 'other.gradle', to: pos
    println pos.beginIndex
    println pos.endIndex
}
```

other.gradle

```
```

gradle -q configure の出力

```
> gradle -q configure
1
5
```

14.6. キャッシング

応答速度を上げるため、Gradleはコンパイル済みのスクリプトをデフォルトですべてキャッシュします。
. g r a d l e
ディレクトリをプロジェクトディレクトリに作成します。そこにコンパイル済みのスクリプトが保存さ
- - r e c o m p i l e - s c r i p t s
オプションをつけてGradleを実行すれば、キャッシュはすべて破棄され、再度コンパイル、保存されます

---

[11] TeamcityやBambooといったCIサーバーがこの機能を持っています。

15
タスク詳解

「６ 章 ビ ル ド ス ク リ プ ト の 基 本
」の入門的なチュートリアルでは、簡単なタスクの作り方や、後からタスクにアクションを追加する方法
さ ら に 、 タ ス ク 間 の 依 存 関 係 を 定 義 す る 方 法 に つ い て も 学 び ま し た 。
単純なタスクについてはこんなところですが、Gradleはタスクという概念をもっと進化させています。
拡張タスク、つまり自身のプロパティやメソッドを持ったタスクを使うことができるのです。
こ れ は 、 今 ま で 使 わ れ て き たAntの タ ー ゲ ッ ト と は 全 く 異 な る 概 念 で す 。
拡張タスクは、Gradleがデフォルトで用意しているものもありますし、自分自身で新たに作成することも

15.1. タスクの定義

「６ 章 ビ ル ド ス ク リ プ ト の 基 本
」で、キーワードを使ったタスク定義を紹介しましたが、タスクの定義方法は他にも少しあります。とき

例15.1 タスクを定義する

build.gradle

```
task(hello) << {
    println "hello"
}

task(copy, type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

タスク名に文字列を使うこともできます。

例15.2 タスクを定義する - タスク名に文字列を使用

build.gradle

```
task('hello') <<
{
    println "hello"
}

task('copy', type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

他にもいくつかシンタックスが用意されています。もしかすると、これらのシンタックスの方が好みに合

例15.3 その他のタスク定義方法

build.gradle

```
tasks.create(name: 'hello') << {
    println "hello"
}

tasks.create(name: 'copy', type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

この例では、`tasks`コレクションにタスクを追加しています。`create()`メソッドのバリエーションについては`TaskContainer`をご参照ください。

15.2. タスクを配置する

ビルドファイルから、定義済みのタスクを探して取得したい、というケースがよくあります。例えば、既

例15.4 タスクにプロパティとしてアクセスする

build.gradle

```
task hello

println hello.name
println project.hello.name
```

`tasks`コレクションからタスクを持ってくることもできます。

例15.5 tasksコレクションからタスクにアクセスする

build.gradle

```
task hello

println tasks.hello.name
println tasks['hello'].name
```

`tasks.getByPath()`を使えば、あらゆるプロジェクトのタスクにアクセスできます。`getByPath()`にはタスク名のほか、タスクの相対パスや絶対パスを渡すことが可能です。

例15.6 パスを使ってタスクにアクセスする

build.gradle

```
project(':projectA') {
    task hello
}

task hello

println tasks.getByPath('hello').path
println tasks.getByPath(':hello').path
println tasks.getByPath('projectA:hello').path
println tasks.getByPath(':projectA:hello').path
```

gradle -q hello の出力

```
> gradle -q hello
:hello
:hello
:projectA:hello
:projectA:hello
```

そのほかの方法については、`TaskContainer`をご参照ください。

15.3. タスクの設定を変更する

例として、Gradleに用意されている`Copy`タスクを見てみましょう。`Copy`
タスクを自分のビルドで作成するには、ビルドスクリプトで次のように宣言します。

例15.7 copyタスクの作成

build.gradle

```
task myCopy(type: Copy)
```

作成されたコピータスクには、デフォルトでは何の動作も設定されていません。
タスクは、タスクのAPI(`Copy`参照)を使って設定することができます。
次のいくつかの例は、どれも同じ設定を行うものです。

Just to be clear, realize that the name of this task is “`myCopy`”, but it is of type “`Copy`”. You can have multiple tasks of the same type, but with different names. You'll find this gives you a lot of power to implement cross-cutting concerns across all tasks of a particular type.

例15.8 タスクの設定 - 様々な方法

build.gradle

```
Copy myCopy = task(myCopy, type: Copy)
myCopy.from 'resources'
myCopy.into 'target'
myCopy.include('**/*.txt', '**/*.xml', '**/*.properties')
```

これは、普段私たちがJavaオブジェクトを設定するのと同じ方法で設定する方法です。この方法では、設
myCopy)を何度も書く必要があります。これは冗長だし、あまり可読性もよくありません。

タスクの設定には別の方法があります。この方法でも同様にコンテキストは保持されますし、おそらく最

例15.9 タスクの設定 - クロージャの使用

build.gradle

```
task myCopy(type: Copy)

myCopy {
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

この方法は、あらゆるタスクで使用できます。3行目のコードは、tasks.getByName()
の単なるショートカットです。大事なことは、getByName()
メソッドにクロージャを渡すと、そのクロージャがタスクが実行されるときではなく、設定
されるときに適用されるという点です。

また、タスクを定義するときに、同時に設定クロージャを使うこともできます。

例15.10 クロージャを伴うタスク定義

build.gradle

```
task copy(type: Copy) {
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

15.4. タスクに依存関係を追加する

タスクの依存関係を定義する方法はいくつかあります。
「タスクの依存関係」
では、タスク名を使って依存関係を定義する方法を紹介しました。タスク名を使うと、同じプロジェクト
projectA:taskXからprojectB:taskY
への依存関係を定義しています。

> **Don't forget about the build phases**
>
> A task has both configuration and actions. When using the <<, you are simply using a shortcut to define an action. Code defined in the configuration section of your task will get executed during the configuration phase of the build regardless of what task was targeted. See 56章 ビルドのライフサイクル for more details about the build lifecycle.

例15.11 別プロジェクトのタスクとの依存関係を定義する

build.gradle

```
project('projectA') {
    task taskX(dependsOn: ':projectB:taskY') << {
        println 'taskX'
    }
}

project('projectB') {
    task taskY << {
        println 'taskY'
    }
}
```

gradle -q taskX の出力

```
> gradle -q taskX
taskY
taskX
```

タスク名を使う代わりに、次の例のように、Task
オブジェクトを使って依存関係を定義することも出来ます。

例15.12 taskオブジェクトを使った依存関係定義

build.gradle

```
task taskX << {
    println 'taskX'
}

task taskY << {
    println 'taskY'
}

taskX.dependsOn taskY
```

gradle -q taskX の出力

```
> gradle -q taskX
taskY
taskX
```

より上級ユーザー向けの方法として、クロージャを使って依存関係を定義することが出来ます。評価フェ

例15.13 クロージャを使った依存関係定義

build.gradle

```
task taskX << {
    println 'taskX'
}

taskX.dependsOn {
    tasks.findAll { task -> task.name.startsWith('lib') }
}

task lib1 << {
    println 'lib1'
}

task lib2 << {
    println 'lib2'
}

task notALib << {
    println 'notALib'
}
```

gradle -q taskX の出力

```
> gradle -q taskX
lib1
lib2
taskX
```

さらにタスクの依存関係に関する情報を得るには、`Task` APIを参照してください。

15.5. Ordering tasks

> Task ordering is an incubating feature. Please be aware that this feature may change in later Gradle versions.

In some cases it is useful to control the order in which 2 tasks will execute, without introducing an explicit dependency between those tasks. The primary difference between a task ordering and a task dependency is that an ordering rule does not influence which tasks will be executed, only the order in which they will be executed.

Task ordering can be useful in a number of scenarios:

- Enforce sequential ordering of tasks: eg. 'build' never runs before 'clean'.
- Run build validations early in the build: eg. validate I have the correct credentials before starting the work for a release build.
- Get feedback faster by running quick verification tasks before long verification tasks: eg. unit tests should run before integration tests.
- A task that aggregates the results of all tasks of a particular type: eg. test report task

combines the outputs of all executed test tasks.

There are two ordering rules available: "must run after" and "should run after".

When you use the "must run after" ordering rule you specify that `taskB` must always run after `tas`, whenever both `taskA` and `taskB` will be run. This is expressed as `taskB.mustRunAfter(taskA)`. The "should run after" ordering rule is similar but less strict as it will be ignored in two situations. Firstly if using that rule introduces an ordering cycle. Secondly when using parallel execution and all dependencies of a task have been satisfied apart from the "should run after" task, then this task will be run regardless of whether its "should run after" dependencies have been run or not. You should use "should run after" where the ordering is helpful but not strictly required.

With these rules present it is still possible to execute `taskA` without `taskB` and vice-versa.

例15.14 Adding a 'must run after' task ordering

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskY.mustRunAfter taskX
```

gradle -q taskY taskX の出力

```
> gradle -q taskY taskX
taskX
taskY
```

例15.15 Adding a 'should run after' task ordering

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskY.shouldRunAfter taskX
```

gradle -q taskY taskX の出力

```
> gradle -q taskY taskX
taskX
taskY
```

In the examples above, it is still possible to execute `taskY` without causing `taskX` to run:

例15.16 Task ordering does not imply task execution

gradle -q taskY の出力

```
> gradle -q taskY
taskY
```

To specify a “must run after” or “should run after” ordering between 2 tasks, you use the `Task.mustRunAfter()` and `Task.shouldRunAfter()` methods. These methods accept a task instance, a task name or any other input accepted by `Task.dependsOn()`.

Note that “`B.mustRunAfter(A)`” or “`B.shouldRunAfter(A)`” does not imply any execution dependency between the tasks:

- It is possible to execute tasks `A` and `B` independently. The ordering rule only has an effect when both tasks are scheduled for execution.
- When run with `--continue`, it is possible for `B` to execute in the event that `A` fails.

As mentioned before, the “should run after” ordering rule will be ignored if it introduces an ordering cycle:

例15.17 A 'should run after' task ordering is ignored if it introduces an ordering cycle

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
task taskZ << {
    println 'taskZ'
}
taskX.dependsOn taskY
taskY.dependsOn taskZ
taskZ.shouldRunAfter taskX
```

gradle -q taskX の出力

```
> gradle -q taskX
taskZ
taskY
taskX
```

15.6. タスクに説明書きを追加する

定義したタスクに、説明書きを追加できます。追加した説明は、たとえば**gradle tasks**を実行したときなどに表示されます。

例15.18 タスクに説明書きを追加する

build.gradle

```
task copy(type: Copy) {
   description 'Copies the resource directory to the target directory.'
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

15.7. タスクの置き換え

ときには、タスクを置き換えたくなることがあります。例えば、Javaプラグインによって追加されたタス

例15.19 タスクの上書き

build.gradle

```
task copy(type: Copy)

task copy(overwrite: true) << {
    println('I am the new one.')
}
```

gradle -q copy の出力

```
> gradle -q copy
I am the new one.
```

ここでは、 Copy
タイプのタスクを、シンプルな別のタスクに置き換えています。タスクを作成するときに、overwrite
プロパティをtrueにセットしなければなりません。そうしないと、Gradleは例外を投げ、すでに同名の5

15.8. タスクをスキップする

Gradleは、タスクの実行をスキップする方法を複数用意しています。

15.8.1. 述語を使う

onlyIf()
メソッドを使って、タスクに述語を付けることができます。タスクは、付加された述語がtrueと評価され

例15.20 述語でタスクをスキップ

build.gradle

```
task hello << {
    println 'hello world'
}

hello.onlyIf { !project.hasProperty('skipHello') }
```

gradle hello -PskipHello の出力

```
> gradle hello -PskipHello
:hello SKIPPED

BUILD SUCCESSFUL

Total time: 1 secs
```

15.8.2. StopExecutionExceptionを使う

タスクスキップのルールが述語で表現できない場合、StopExecutionException
を使うことができます。この例外がタスクのアクションから投げられたら、実行中のアクション、およ

例15.21 StopExecutionExceptionでタスクをスキップ

build.gradle

```
task compile << {
    println 'We are doing the compile.'
}

compile.doFirst {
    // Here you would put arbitrary conditions in real life.
    // But this is used in an integration test so we want defined behavior.
    if (true) { throw new StopExecutionException() }
}
task myTask(dependsOn: 'compile') << {
   println 'I am not affected'
}
```

gradle -q myTask の出力

```
> gradle -q myTask
I am not affected
```

この機能は、Gradleが標準提供しているタスクが絡んでいるときに役に立ちます。このようなビルドイン
条件付きで実行させることができるのです。[13]

15.8.3. タスクの有効化と無効化

すべてのタスクは、enabled
フラグを持っており、デフォルトではtrueに設定されています。このフラグをfalseにすると、そのタス

例15.22 タスクの有効化と無効化

build.gradle

```
task disableMe << {
    println 'This should not be printed if the task is disabled.'
}
disableMe.enabled = false
```

gradle disableMe の出力

```
> gradle disableMe
:disableMe SKIPPED

BUILD SUCCESSFUL

Total time: 1 secs
```

15.9. 更新されていないタスクをスキップする

JavaプラグインのようなGradleが提供しているタスクを使ったとき、Gradleが未更新(UP-TO-DATE)のタ

15.9.1. タスクの入力物と出力物を宣言する

一つ例を見てみましょう。ここでは、定義したタスクが、XMLのソースファイルを元にいくつかのファィ

例15.23 生成タスク

build.gradle

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

gradle transform の出力

```
> gradle transform
:transform
Transforming source file.
```

gradle transform の出力

```
> gradle transform
:transform
Transforming source file.
```

タスクを2回目に実行したとき、なにも変更されていないにもかかわらず、タスクの実行がスキップされ

すべてのタスクにはinputsプロパティとoutputs
プロパティがあります。これが入力と出力を宣言するために使うプロパティです。次の例では、先ほどの

例15.24 タスクの入力と出力を宣言

build.gradle

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    inputs.file srcFile
    outputs.dir destDir
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

gradle transform の出力

```
> gradle transform
:transform
Transforming source file.
```

gradle transform の出力

```
> gradle transform
:transform UP-TO-DATE
```

これで、GradleはタスクがUP-TO-DATEなのかどうかを決定するため、どのファイルをチェックするべき

タスクのinputsプロパティは`TaskInputs`型、outputsプロパティは`TaskOutputs`型です。

A task with no defined outputs will never be considered up-to-date. For scenarios where the outputs of a task are not files, or for more complex scenarios, the `TaskOutputs.upToDateWhen()` method allows you to calculate programmatically if the tasks outputs should be considered up to date.

A task with only outputs defined will be considered up-to-date if those outputs are unchanged since the previous build.

15.9.2. どのように動作するのか？

タスクが初回に実行されるとき、実行前にGradleは入力物のスナップショットをとります。このスナップ

この後は、タスクが実行される前に毎回、入力物と出力物のスナップショットを新しく作成します。
もし、新しいスナップショットが、前回のスナップショットと同じなら、Gradleは出力がUP-TO-DATEな
スナップショットが異なっていれば、Gradleはタスクを実行し、次回実行されるときのためにスナップシ

Note that if a task has an output directory specified, any files added to that directory since the last time it was executed are ignored and will NOT cause the task to be out of date. This is so

unrelated tasks may share an output directory without interfering with each other. If this is not the behaviour you want for some reason, consider using `TaskOutputs.upToDateWhen()`

15.10. タスクルール

ときには、広範囲、または無限大な範囲のパラメータに応じた動作を行うタスクを定義したくなることカ

例15.25 タスクルール

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << {
            println "Pinging: " + (taskName - 'ping')
        }
    }
}
```

gradle -q pingServer1 の出力

```
> gradle -q pingServer1
Pinging: Server1
```

文字列が引数に渡されていますが、これはルールの説明として使用されます。例えば、**gradle tasks** を実行したときなどに表示されます。

ルールは、ただコマンドラインからのタスク呼び出しにのみ使えるものではありません。dependsOnに

例15.26 ルールベース・タスクの依存関係

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << {
            println "Pinging: " + (taskName - 'ping')
        }
    }
}

task groupPing {
    dependsOn pingServer1, pingServer2
}
```

gradle -q groupPing の出力

```
> gradle -q groupPing
Pinging: Server1
Pinging: Server2
```

If you run “`gradle -q tasks`” you won't find a task named “`pingServer1`” or “`pingServer2`”, but this script is executing logic based on the request to run those tasks.

15.11. Finalizer tasks

Finalizers tasks are an incubating feature (see 「試験的」).

Finalizer tasks are automatically added to the task graph when the finalized task is scheduled to run.

例15.27 Adding a task finalizer

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}

taskX.finalizedBy taskY
```

gradle -q taskX の出力

```
> gradle -q taskX
taskX
taskY
```

Finalizer tasks will be executed even if the finalized task fails.

例15.28 Task finalizer for a failing task

build.gradle

```
task taskX << {
    println 'taskX'
    throw new RuntimeException()
}
task taskY << {
    println 'taskY'
}

taskX.finalizedBy taskY
```

gradle -q taskX の出力

```
> gradle -q taskX
taskX
taskY
```

On the other hand, finalizer tasks are not executed if the finalized task didn't do any work, for example if it is considered up to date or if a dependent task fails.

Finalizer tasks are useful in situations where the build creates a resource that has to be cleaned up regardless of the build failing or succeeding. An example of such a resource is a web

container that is started before an integration test task and which should be always shut down, even if some of the tests fail.

To specify a finalizer task you use the `Task.finalizedBy()` method. This method accepts a task instance, a task name, or any other input accepted by `Task.dependsOn()`.

15.12. まとめ

以前にAntを使ったことがあれば、Copy
のような拡張タスクは、Antのターゲットとタスクの間をとったようなものに見えるでしょう。
Antのタスクとターゲットは実際には異なるエンティティですが、Gradleはこれらの記法を1つのエンテ
単純なGradleのタスクはAntのターゲットのようなものだし、拡張タスクはAntタスクの性質を含んでい
すべてのGradleタスクは、同じのAPIを共有していて、タスク間の依存関係を定義することができます。
このため、GradleのタスクはAntタスクに比べて、より設定しやすいものになっているでしょう。
Gradleのタスクは、型システムをフル活用しており、より表現力豊かで、メンテナンス性もよくなってい

---

[13] なぜimport文も完全修飾名も使用せずStopExecutionExceptionにアクセスできているのか不思議に思わ
付録E IDE対応の現状と、IDEによらない開発支援参照)。

16
ファイルを取り扱う

ファイルの取り扱いは、ほとんどのビルドで発生する作業です。なので、それを補助するため、Gradleは

16.1. ファイルを参照する

`Project.file()`
メソッドで、プロジェクトディレクトリからの相対参照でファイルを取得できます。

例16.1 ファイルを参照する

build.gradle

```
// Using a relative path
File configFile = file('src/config.xml')

// Using an absolute path
configFile = file(configFile.absolutePath)

// Using a File object with a relative path
configFile = file(new File('src/config.xml'))
```

`file()`
メソッドには、あらゆるオブジェクトを渡すことができます。渡したオブジェクトは、絶対参照の`File`に変換されます。大抵は`String`か`File`
のインスタンスを渡すことになるでしょう。渡されたオブジェクトの`toString()`
が呼ばれ、その返値がファイルパスとして使われます。もしそのパスが絶対パスだった場合、そのパスで
`File`
インスタンスが構築されます。そうでなければ、プロジェクトディレクリのパスを先頭に追加してから`File`インスタンスが構築されます。また、`file()`メソッドは`file:/some/path.xml`
のようなURLにも対応しています。

このメソッドを使って、ユーザーが指定した値を簡単に絶対参照の`File`
インスタンスに変換できます。 これは、`new File(somePath)`を使うよりもよい方法でしょう。`file`
メソッドは、常にプロジェクトディレクリからの相対参照で解決されるからです。ワーキングディレクト

16.2. ファイルコレクション

ファイルコレクションは、単なるファイルの集合です。これは、FileCollection
インターフェースで表現されます。Gradle
APIに含まれる多くのオブジェクトがこのインターフェースを実装しています。例えば、依存関係のコン
もこのFileCollectionインターフェースを実装しています。

FileCollectionを取得する方法の一つは、Project.files()
メソッドを使うことです。このメソッドには任意の数のオブジェクトを渡すことができ、渡したオブジェ
Fileオブジェクトの集合に変換されます。これらのオブジェクトは、「ファイルを参照する 」
で述べたfile()
メソッドと同じように、プロジェクトディレクトリからの相対参照で解決されます。
また、コレクション、イテレーブル、マップ、配列を渡すこともできます。これらは、フラット化されて
Fileインスタンスに変換されます。

例16.2 ファイルコレクションの作成

build.gradle

```
FileCollection collection = files('src/file1.txt',
                                  new File('src/file2.txt'),
                                  ['src/file3.txt', 'src/file4.txt'])
```

ファイルコレクションはイテレーブルです。さらに、as
演算子で様々な型に変換することができます。 また、二つのファイルコレクションを+
演算子で合成したり、-
演算子でファイルコレクションから別のファイルコレクションの要素を取り除いたりすることができます
次の例は、ファイルコレクションでどのようなことができるかを示すものです。

例16.3 ファイルコレクションを使う

build.gradle

```
// Iterate over the files in the collection
collection.each {File file ->
    println file.name
}

// Convert the collection to various types
Set set = collection.files
Set set2 = collection as Set
List list = collection as List
String path = collection.asPath
File file = collection.singleFile
File file2 = collection as File

// Add and subtract collections
def union = collection + files('src/file3.txt')
def different = collection - files('src/file3.txt')
```

files()メソッドには、クロージャやCallableのインスタンスを渡すこともできます。

これはそのファイルコレクションの内容が要求されたときに実行され、返値がFile
インスタンスの集合に変換されます。返値はfiles()
に渡すことができるオブジェクトならなんでも構いません。これは、FileCollection
インタフェースを簡単に「実装する」方法とも言えます。

例16.4 ファイルコレクションを実装する

build.gradle

```
task list << {
    File srcDir

    // Create a file collection using a closure
    collection = files { srcDir.listFiles() }

    srcDir = file('src')
    println "Contents of $srcDir.name"
    collection.collect { relativePath(it) }.sort().each { println it }

    srcDir = file('src2')
    println "Contents of $srcDir.name"
    collection.collect { relativePath(it) }.sort().each { println it }
}
```

gradle -q list の出力

```
> gradle -q list
Contents of src
src/dir1
src/file1.txt
Contents of src2
src2/dir1
src2/dir2
```

他にもいくつかfiles()メソッドに渡せる型があります。

FileCollection
フラット化されてから、その内容がファイルコレクションに追加される。

Task
タスクの出力ファイルがファイルコレクションに追加される。

TaskOutputs
TaskOutputsの出力ファイルがファイルコレクションに追加される。

留意すべき重要な点は、ファイルコレクションの内容は、必要になったときに遅延評価されるということ
これはつまり、例えば未来にタスクなどによって作られるであろうファイル群を表すFileCollection
を作ることもできる、ということを意味します。

16.3. ファイルツリー

ファイルツリーは、階層構造を持つファイルの集合です。例えば、ディレクトリツリーやZIPの中身を表
`FileTree`インターフェースによって表現されます。`FileTree`インターフェースは`FileCollection`
インターフェースを継承しています。なので、FileTreeはFileCollectionと全く同じ方法で取り扱うことが
いくつかのGradleオブジェクトは`FileTree`を実装しています。例えば、ソースセットなどです。

`FileTree`インスタンスを取得する方法の一つは、`Project.fileTree()`
メ ソ ッ ド を 使 用 す る こ と で す 。
このメソッドは、ベースディレクトリとAntスタイルのinclude/excludeパターンを指定して`FileTree`
を構築します。

例16.5 ファイルツリーを作成する

build.gradle

```
// Create a file tree with a base directory
FileTree tree = fileTree(dir: 'src/main')

// Add include and exclude patterns to the tree
tree.include '**/*.java'
tree.exclude '**/Abstract*'

// Create a tree using path
tree = fileTree('src').include('**/*.java')

// Create a tree using closure
tree = fileTree('src') {
    include '**/*.java'
}

// Create a tree using a map
tree = fileTree(dir: 'src', include: '**/*.java')
tree = fileTree(dir: 'src', includes: ['**/*.java', '**/*.xml'])
tree = fileTree(dir: 'src', include: '**/*.java', exclude: '**/*test*/**')
```

ファイルツリーは、ファイルコレクションと同じ方法で使うことができます。さらに、ツリー構造を辿っ

例16.6 ファイルツリーを使う

build.gradle

```
// Iterate over the contents of a tree
tree.each {File file ->
    println file
}

// Filter a tree
FileTree filtered = tree.matching {
    include 'org/gradle/api/**'
}

// Add trees together
FileTree sum = tree + fileTree(dir: 'src/test')

// Visit the elements of the tree
tree.visit {element ->
    println "$element.relativePath => $element.file"
}
```

16.4. アーカイブの内容をファイルツリーとして使う

ZIPやTARファイルなどのアーカイブを、ファイルツリーとして使うことができます。そのためのメソッ `Project.zipTree()`や`Project.tarTree()`です。これらのメソッドは、`FileTree` インスタンスを返すもので、返されたインスタンスは、他のファイルツリーやファイルコレクションと同 例えば、内容をコピーすることでアーカイブを解凍したり、別のアーカイブとマージしたりできます。

例16.7 アーカイブをファイルツリーとして使う

build.gradle

```
// Create a ZIP file tree using path
FileTree zip = zipTree('someFile.zip')

// Create a TAR file tree using path
FileTree tar = tarTree('someFile.tar')

//tar tree attempts to guess the compression based on the file extension
//however if you must specify the compression explicitly you can:
FileTree someTar = tarTree(resources.gzip('someTar.ext'))
```

16.5. 入力ファイルセットを指定する

多くのGradleオブジェクトが入力ファイルセットを格納できるプロパティを持っています。例えば、 JavaCompileタスクにはsource
プロパティがあり、コンパイルするべきソースファイルの集合をそこに定義します。
このプロパティの値には、files()でサポートされている、上記の全ての型が使用可能です。
つまり、File、String、コレクション、FileCollection
、クロージャでさえセット可能ということです。次の例を見てください。

例16.8 ファイルセットを指定する

build.gradle

```
// Use a File object to specify the source directory
compile {
    source = file('src/main/java')
}

// Use a String path to specify the source directory
compile {
    source = 'src/main/java'
}

// Use a collection to specify multiple source directories
compile {
    source = ['src/main/java', '../shared/java']
}

// Use a FileCollection (or FileTree in this case) to specify the source files
compile {
    source = fileTree(dir: 'src/main/java').matching { include 'org/gradle/api/**'
}

// Using a closure to specify the source files.
compile {
    source = {
        // Use the contents of each zip file in the src dir
        file('src').listFiles().findAll {it.name.endsWith('.zip')}.collect { zipTre
    }
}
```

大抵、そのプロパティと同名のメソッドも定義されていて、そのメソッドでファイルセットを追加できる
files()でサポートされている全ての型を渡すことができます。

例16.9 ファイルセットを指定する

build.gradle

```
compile {
    // Add some source directories use String paths
    source 'src/main/java', 'src/main/groovy'

    // Add a source directory using a File object
    source file('../shared/java')

    // Add some source directories using a closure
    source { file('src/test/').listFiles() }
}
```

16.6. ファイルをコピーする

C　o　p　y
タスクを使ってファイルをコピーできます。このタスクはとても柔軟で、コピーしたファイルの内容をフ

C　o　p　y
タスクを使うには、コピーすべきソースファイルと、コピー先のディレクトリを指定しなければなりませ
Copy仕様を使って行います。Copy仕様は、`CopySpec`インターフェースで表現されており、`Copy`
タスクはこのインターフェースを実装したものです。ソースファイルの指定には、
`CopySpec.from()`メソッドを使います。コピー先ディレクトリの指定には、`CopySpec.into()`
メソッドを使います。

例16.10 Copyタスクでファイルをコピーする

build.gradle

```
task copyTask(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
}
```

`from()`メソッドが受け付ける型は、`files()`
メソッドと同じです。ディレクトリパスに解決されるような引数を渡した場合、そのディレクトリ以下の
もし、引数に渡したパスにファイルがない場合、その引数は無視されます。
引数にタスクを渡した場合、そのタスクの出力ファイル(タスクが作成するファイル)がコピーされ、その
`into()`メソッドは、`file()`と同じ型を引数に取ります。以下にもう一つ例を挙げます。

例16.11 Copyタスクのコピー元と宛先を指定する

build.gradle

```
task anotherCopyTask(type: Copy) {
    // Copy everything under src/main/webapp
    from 'src/main/webapp'
    // Copy a single file
    from 'src/staging/index.html'
    // Copy the output of a task
    from copyTask
    // Copy the output of a task using Task outputs explicitly.
    from copyTaskWithPatterns.outputs
    // Copy the contents of a Zip file
    from zipTree('src/main/assets.zip')
    // Determine the destination directory later
    into { getDestDir() }
}
```

Antスタイルのinclude/excludeパターンかクロージャを用いて、コピーするファイルを選択することがで

例16.12 コピーするファイルを選択する

build.gradle

```
task copyTaskWithPatterns(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    include '**/*.html'
    include '**/*.jsp'
    exclude { details -> details.file.name.endsWith('.html') &&
                         details.file.text.contains('staging') }
}
```

また、`Project.copy()`
メソッドでファイルをコピーすることもできます。これはタスクの場合と大体同じように動作しますが、
`copy()`メソッドはインクリメンタルには実施されません(
「更新されていないタスクをスキップする」参照)。

例16.13 copy()メソッドで更新チェックせずにファイルをコピーする

build.gradle

```
task copyMethod << {
    copy {
        from 'src/main/webapp'
        into 'build/explodedWar'
        include '**/*.html'
        include '**/*.jsp'
    }
}
```

次に、タスクがコピー元として使われる場合(`from()`メソッドの引数になる場合)でも、`copy()`
メソッドはタスクの依存関係を考慮しません。copy()はあくまでメソッドであってタスクではないからで
よって、copy()メソッドをタスクアクションの中で呼び出す場合は、正しく動作させるために明示的に全

例16.14 copy()メソッドで更新チェックを実施してファイルをコピーする

build.gradle

```
task copyMethodWithExplicitDependencies{
    // up-to-date check for inputs, plus add copyTask as dependency
    inputs.file copyTask
    outputs.dir 'some-dir' // up-to-date check for outputs
    doLast{
        copy {
            // Copy the output of copyTask
            from copyTask
            into 'some-dir'
        }
    }
}
```

で き る だ け ｃ ｏ ｐ ｙ
タスクのほうを使用するようにしてください。そうすれば、余計な手間をかけずにインクリメンタルビル
一方、copy()メソッドは、あるタスクの実装の一部として
ファイルコピーを組み込むことができます。　つまり、copy()メソッドはカスタムタスク(58章
カ ス タ ム タ ス ク の 作 成
参照)の機能でファイルコピーが必要になった場合に使用されることを想定しています。
そのカスタムタスクでは、ファイルコピーの実際の仕様に基づいて適切に入力と出力が宣言されていなけ

16.6.1. ファイルをリネームする

例16.15 コピー時にファイルをリネームする

build.gradle

```
task rename(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // Use a closure to map the file name
    rename { String fileName ->
        fileName.replace('-staging-', '')
    }
    // Use a regular expression to map the file name
    rename '(.+)-staging-(.+)', '$1$2'
    rename(/(.+)-staging-(.+)/, '$1$2')
}
```

16.6.2. ファイルをフィルタリングする

例16.16 コピー時にファイルをフィルタリングする

build.gradle

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens

task filter(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // Substitute property tokens in files
    expand(copyright: '2009', version: '2.3.1')
    expand(project.properties)
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter)
    filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])
    // Use a closure to filter each line
    filter { String line ->
        "[$line]"
    }
}
```

A “token” in a source file that both the “expand” and “filter” operations look for, is formatted like “@tokenName@” for a token named “tokenName”.

16.6.3. CopySpecクラスを使う

コピー仕様は階層構造を構成でき、宛先パス、include/excludeパターン、コピー動作、ファイル名のマ

例16.17 入れ子構造のコピー仕様

build.gradle

```
task nestedSpecs(type: Copy) {
    into 'build/explodedWar'
    exclude '**/*staging*'
    from('src/dist') {
        include '**/*.html'
    }
    into('libs') {
        from configurations.runtime
    }
}
```

16.7. Syncタスクを使う

S y n c は C o p y
タスクを継承したタスクです。このタスクは、宛先ディレクトリにファイルをコピーし、その後、コピー

次の例では、build/libsにある実行時依存関係のコピーをメンテナンスしています。

例16.18 Syncタスクで依存関係をコピーする

build.gradle

```
task libs(type: Sync) {
    from configurations.runtime
    into "$buildDir/libs"
}
```

16.8. アーカイブを作成する

一つのプロジェクトで、JARファイルを好きなだけ作成することができます。WAR、ZIP、TARなどのア-
アーカイブは、 `Zip`、 `Tar`、 `Jar`、 `War`、 `Ear`などのアーカイブタスクを使って作成します。
これらは全て同じ使い方なので、ここではZIPファイルの作成方法を見てみましょう。

例16.19 ZIPアーカイブの作成

build.gradle

```
apply plugin: 'java'

task zip(type: Zip) {
    from 'src/dist'
    into('libs') {
        from configurations.runtime
    }
}
```

アーカイブタスクは、全て`Copy`
タスクと全く同じように動作します。これらのタスクは、Copyタスク同様、
`CopySpec`インターフェースを実装しています。`Copy`
タスクと同じように`from()`
メソッドで入力ファイルを指定し、必須ではありませんが`into()`
メソッドで最終的にアーカイブが出力される場所を指定します。ファイルのフィルタリング、リネーム、

なぜJavaプラグインを使

Javaプラグインは、アーカイブタス

16.8.1. アーカイブのネーミング

The format of *`projectName-version.type`* is used for generated archive file names. For example:

作成されるアーカイブの名前は、デフォルトでは*`projectName-version.type`*です。例えば、

例16.20 ZIPアーカイブの作成

build.gradle

```
apply plugin: 'java'

version = 1.0

task myZip(type: Zip) {
    from 'somedir'
}

println myZip.archiveName
println relativePath(myZip.destinationDir)
println relativePath(myZip.archivePath)
```

gradle -q myZip の出力

```
> gradle -q myZip
zipProject-1.0.zip
build/distributions
build/distributions/zipProject-1.0.zip
```

ここでは、myZipという名前のZipアーカイブタスクが、zipProject-1.-.zip
というZIPファイルを作成しています。大事なことは、アーカイブタスクの名前と、そのタスクで作成し
archivesBaseName
プロパティで変更できます。そのアーカイブ名も、後からいつでも変更可能です。

アーカイブタスクにセットできるプロパティはたくさんあります。それらを下の
表16.1「アーカイブタスク - ネーミングプロパティ」
にリストしました。例えば、アーカイブの名前を変更したい場合は次のようにします。

例16.21 アーカイブタスクの設定 - カスタムアーカイブ名

build.gradle

```
apply plugin: 'java'
version = 1.0

task myZip(type: Zip) {
    from 'somedir'
    baseName = 'customName'
}

println myZip.archiveName
```

gradle -q myZip の出力

```
> gradle -q myZip
customName-1.0.zip
```

アーカイブ名をさらにカスタマイズすることもできます。

例16.22 アーカイブタスクの設定 - appendix & classifier

build.gradle

```
apply plugin: 'java'
archivesBaseName = 'gradle'
version = 1.0

task myZip(type: Zip) {
    appendix = 'wrapper'
    classifier = 'src'
    from 'somedir'
}

println myZip.archiveName
```

gradle -q myZip の出力

```
> gradle -q myZip
gradle-wrapper-1.0-src.zip
```

表16.1 アーカイブタスク - ネーミングプロパティ

プロパティ名	型	デフォルト値
archiveName	String	*baseName-appendix-version-classifier.extension* プロパティが空だった場合、それに伴う - も追加されない。
archivePath	File	*destinationDir/archiveName*
destinationDir	File	アーカイブの種類に依存する。JAR、WARの場合は*project.build* 。ZIP、TARの場合は*project.buildDir*/distributions。
baseName	String	*project.name*
appendix	String	null
version	String	*project.version*
classifier	String	null
extension	String	アーカイブの種類に依存する。TARファイルの場合、圧縮方法にも依 , jar, war, tar, tgz or tbz2.

16.8.2. 複数のアーカイブで中身を共有する

`Project.copySpec()`メソッドを使ってアーカイブ間で中身を共有できます。

アーカイブを別のプロジェクトで使えるようにするために、そのアーカイブを公開したくなることがよく
52章アーティファクトの公開 に記載されています。

17 GradleからAntを使う

Gradleは優れたAntとの統合機能を提供しています。
Gradleビルドの中で、個別のAntタスクや、Antビルド全体を利用することができます。
実際、AntのXMLフォーマットを利用するよりも、Gradleビルドの中でAntタスクを使う方がはるかに簡
Gradleを単に強力なAntスクリプティングツールとして使ってもよいくらいです。

Antは二つのレイヤーに分割できます。 第一のレイヤーはAnt言語で、`build.xml`
の文法やターゲットの取り扱い、およびmacrodefのような特別な構成要素などを提供します。
別の言い方では、Antタスクとタイプを除くすべてのものです。
Gradleはこの言語を理解し、GradleプロジェクトにAntの`build.xml`
を直接インポートすることを可能にしています。
そのため、AntビルドのターゲットをGradleのタスクであるかのように利用することができます。

Antの第二のレイヤーは、`javac`や`copy`、`jar`といった豊富なAntタスクやタイプの資産です。
このレイヤーに対してはGroovyのすばらしい`AntBuilder`
が利用できるため、Gradleはシンプルな統合機能を提供するのみです。

最後に、ビルドスクリプトはGroovyスクリプトなので、いつでもAntビルドを外部プロセスとして実行で
ビルドスクリプトは`"ant clean compile".execute()`
のような実行文を含んでいてもかまいません。[15]

GradleのAnt統合機能を、AntからGradleへのビルド移行パスとして利用することもできます。
例えば、既存のAntビルドをインポートするところから始めてもよいでしょう。
それから、依存関係の宣言をAntスクリプトから新しいビルドファイルへ移動していきます。
最後に、タスクを新しいビルドファイルへ移動するか、Gradleプラグインで置き換えます。
このプロセスは一部分づつ段階的に実施でき、プロセス全体を通してGradleによるビルドを利用できます

17.1. ビルドでのAntタスクとタイプの利用

ビルドスクリプトではGradleによってプロパティ`ant`が提供されます。 これは`AntBuilder`
インスタンスへの参照です。 この`AntBuilder`
はビルドスクリプトからAntタスクやタイプ、プロパティへのアクセスに利用します。 Antの`build.xml`
フォーマットからGroovyへのマッピングは非常に簡単です。 以下で説明します。

Antタスクを実行するには、`AntBuilder`インスタンスのメソッドを呼び出します。
Antタスク名がメソッド名になります。 例えば、Antの`echo`タスクを実行する場合は、`ant.echo()`
メソッドを呼び出します。 Antタスクの属性は、メソッドにMap型のパラメータとして渡します。
以下は`echo`タスクを実行するサンプルです。

Groovyコードとアntタスクのマークアップを混在できることに注意してください。
これは非常に強力です。

例17.1 Antタスクの利用

build.gradle

```
task hello << {
    String greeting = 'hello from Ant'
    ant.echo(message: greeting)
}
```

gradle hello の出力

```
> gradle hello
:hello
[ant:echo] hello from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

Antタスクのメソッド呼び出しのパラメータとして渡してやることで、ネストされたテキストをAntタス
この例では、echoタスクに対するメッセージをネストされたテキストとして渡しています。

例17.2 Antタスクにネストされたテキストを渡す

build.gradle

```
task hello << {
    ant.echo('hello from Ant')
}
```

gradle hello の出力

```
> gradle hello
:hello
[ant:echo] hello from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

Antタスクにネストされた要素を渡す場合はクロージャの内部に記述します。
ネストされた要素はタスクと同じように、指定したい要素名と同じメソッドを呼び出すことで指定できま

例17.3 Antタスクにネストされた要素を渡す

build.gradle

```
task zip << {
    ant.zip(destfile: 'archive.zip') {
        fileset(dir: 'src') {
            include(name: '**.xml')
            exclude(name: '**.java')
        }
    }
}
```

Antタスクにアクセスするのと同じ方法、すなわちタイプ名をメソッド名として利用することでAntタイ
メソッド呼び出しの戻り値はAntデータ型なので、ビルドスクリプトの中で直接利用できます。
次の例では、Antの`path`オブジェクトを生成し、その内容をすべてイテレートしています。

例17.4 Antタイプの利用

build.gradle

```
task list << {
    def path = ant.path {
        fileset(dir: 'libs', includes: '*.jar')
    }
    path.list().each {
        println it
    }
}
```

`AntBuilder`に関する詳細は「Groovy in Action」の8.4節、ないしはGroovy Wiki を参照してください。

17.1.1. ビルドでカスタムAntタスクを使う

ビルドでカスタムタスクを有効にするためには、`build.xml`ファイルと同じくAntタスクの`taskdef`（大抵はこちらの方が簡単です）か`typedef`を利用します。
これにより組み込みのAntタスクと同様にカスタムAntタスクが参照できるようになります。

例17.5 カスタムAntタスクの利用

build.gradle

```
task check << {
    ant.taskdef(resource: 'checkstyletask.properties') {
        classpath {
            fileset(dir: 'libs', includes: '*.jar')
        }
    }
    ant.checkstyle(config: 'checkstyle.xml') {
        fileset(dir: 'src')
    }
}
```

カスタムタスクを利用するために必要なクラスパスの設定には、Gradleの依存関係管理機能が利用でき

このために、クラスパスに対するカスタムコンフィグレーションを定義して、コンフィグレーションに対
詳細については「依存関係の定義方法 」に記述されています。

例17.6 カスタムAntタスクに対するクラスパスの宣言

build.gradle

```
configurations {
    pmd
}

dependencies {
    pmd group: 'pmd', name: 'pmd', version: '4.2.5'
}
```

クラスパスを設定するには、カスタムコンフィグレーションのasPathプロパティを利用します。

例17.7 カスタムAntタスクと依存関係管理を併用

build.gradle

```
task check << {
    ant.taskdef(name: 'pmd',
                classname: 'net.sourceforge.pmd.ant.PMDTask',
                classpath: configurations.pmd.asPath)
    ant.pmd(shortFilenames: 'true',
            failonruleviolation: 'true',
            rulesetfiles: file('pmd-rules.xml').toURI().toString()) {
        formatter(type: 'text', toConsole: 'true')
        fileset(dir: 'src')
    }
}
```

17.2. Antビルドのインポート

`ant.importBuild()`
メソッドを利用して、GradleプロジェクトにAntビルドをインポートできます。
Antビルドをインポートした場合、各AntターゲットはGradleのタスクとして扱われます。
つまり、AntターゲットとGradleタスクをまったく同一の方法で操作することができるということです。

例17.8 Antビルドのインポート

build.gradle

```
ant.importBuild 'build.xml'
```

build.xml

```
<project>
    <target name="hello">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

gradle hello の出力

```
> gradle hello
:hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

Antターゲットに依存するタスクを追加することもできます:

例17.9 Antターゲットに依存するタスク

build.gradle

```
ant.importBuild 'build.xml'

task intro(dependsOn: hello) << {
    println 'Hello, from Gradle'
}
```

gradle intro の出力

```
> gradle intro
:hello
[ant:echo] Hello, from Ant
:intro
Hello, from Gradle

BUILD SUCCESSFUL

Total time: 1 secs
```

Antターゲットにふるまいを追加することも可能です:

例17.10 Antターゲットにふるまいを追加

build.gradle

```
ant.importBuild 'build.xml'

hello << {
    println 'Hello, from Gradle'
}
```

gradle hello の出力

```
> gradle hello
:hello
[ant:echo] Hello, from Ant
Hello, from Gradle

BUILD SUCCESSFUL

Total time: 1 secs
```

GradleタスクにA依存するAntターゲットを定義することも可能です:

例17.11 Ant target that depends on Gradle task

build.gradle

```
ant.importBuild 'build.xml'

task intro << {
    println 'Hello, from Gradle'
}
```

build.xml

```
<project>
    <target name="hello" depends="intro">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

gradle hello の出力

```
> gradle hello
:intro
Hello, from Gradle
:hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

Sometimes it may be necessary to “rename” the task generated for an Ant target to avoid a naming collision with existing Gradle tasks. To do this, use the `AntBuilder.importBuild()` method.

例17.12 Renaming imported Ant targets

build.gradle

```
ant.importBuild('build.xml') { antTargetName ->
    'a-' + antTargetName
}
```

build.xml

```
<project>
    <target name="hello">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

gradle a-hello の出力

```
> gradle a-hello
:a-hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

Note that while the second argument to this method should be a `Transformer`, when programming in Groovy we can simply use a closure instead of an anonymous inner class (or similar) due to Groovy's support for automatically coercing closures to single-abstract-method types.

17.3. Antプロパティとリファレンス

Antタスクで利用できるようにAntプロパティを設定する方法は複数あります。 `AntBuilder` インスタンスのプロパティを直接設定できます。
Antプロパティは変更可能なMapとしても利用可能です。 Antの`property`タスクも利用できます。
以下はこれらをどのように実行するのかを示すサンプルです。

例17.13 Antプロパティの設定

build.gradle

```
ant.buildDir = buildDir
ant.properties.buildDir = buildDir
ant.properties['buildDir'] = buildDir
ant.property(name: 'buildDir', location: buildDir)
```

build.xml

```
<echo>buildDir = ${buildDir}</echo>
```

多くのAntタスクは実行時にプロパティを設定します。
これらのプロパティを取得する方法も複数あります。 `AntBuilder` インスタンスからプロパティを直接取得できます。 AntプロパティはMapとしても参照可能です。

以下はサンプルです。

例17.14 Antプロパティの取得

build.xml

```
<property name="antProp" value="a property defined in an Ant build"/>
```

build.gradle

```
println ant.antProp
println ant.properties.antProp
println ant.properties['antProp']
```

Antリファレンスを設定する方法は複数あります:

例17.15 Antリファレンスの設定

build.gradle

```
ant.path(id: 'classpath', location: 'libs')
ant.references.classpath = ant.path(location: 'libs')
ant.references['classpath'] = ant.path(location: 'libs')
```

build.xml

```
<path refid="classpath"/>
```

Antリファレンスを取得する方法も複数あります:

例17.16 Antリファレンスの取得

build.xml

```
<path id="antPath" location="libs"/>
```

build.gradle

```
println ant.references.antPath
println ant.references['antPath']
```

17.4. API

Ant統合機能は`AntBuilder`によって提供されています。

---

[15] Groovyでは文字列をコマンドとして実行できます。外部プロセスの実行について詳しく学ぶには、「Groovy in Action」の9.3.2節か、Groovy Wikiを参照してください。

18
ロギング

ログは、ビルドツールのユーザーインターフェースとも言えるものです。あまりにくどいと本当に重要な
表１８．１「ログレベル」
)を規定しています。だいたいはどこか見たことのあるログレベルだと思いますが、QUIETと
LIFECYCLEの2つはGradle特有のものです。LIFECYCLE
はデフォルトのログレベルで、ビルドプロセスの進行状況をレポートするために使われます。

表18.1 ログレベル

レベル	用途
ERROR	エラーメッセージ
QUIET	重要な情報メッセージ
WARNING	警告メッセージ
LIFECYCLE	進行状況を示す情報メッセージ
INFO	情報メッセージ
DEBUG	デバッグメッセージ

18.1. ログレベルの選択

表示させるログレベルは、コマンドラインオプション(
表18.2「ログレベルに関するコマンドラインオプション」)で変更できます。
表18.3「スタックトレースに関するコマンドラインオプション」
には、スタックトレースログをどうするか指定する方法を載せています。

表18.2 ログレベルに関するコマンドラインオプション

オプション	出力ログレベル
ロギングオプション未指定	LIFECYCLE以上
`-q` または `--quiet`	QUIET以上
`-i` または `--info`	INFO以上
`-d` または `--debug`	DEBUG以上(要するに全部)

表18.3 スタックトレースに関するコマンドラインオプション

オプション	意味
スタックトレースオプション未指定	ビルドエラー(コンパイルエラーとか)のスタックトレースはコン が指定されている場合、切り詰められたスタックトレースが常に
-s または --stacktrace	切り詰められたスタックトレースが表示される。Groovyのフル コードのどこがダメなのか有用な情報がでてこない)ので、こち
-S または --full-stacktrace	フルスタックトレースを表示する。

18.2. ログメッセージを書く

ログメッセージを書く簡単な方法は、標準出力に書き込むことです。Gradleは、標準出力に書かれたすべ
QUIETレベルでリダイレクトします。

例18.1 ログに標準出力を使う

build.gradle

```
println 'A message which is logged at QUIET level'
```

また、ビルドスクリプト内ではloggerプロパティも使えます。このプロパティには、Logger
(SLF4JのLogger
インターフェースにGradle特有のメソッドを少し追加したもの)のインスタンスがセットされています。

例18.2 自分でログメッセージを書く

build.gradle

```
logger.quiet('An info log message which is always logged.')
logger.error('An error log message.')
logger.warn('A warning log message.')
logger.lifecycle('A lifecycle info log message.')
logger.info('An info log message.')
logger.debug('A debug log message.')
logger.trace('A trace log message.')
```

さらに、ビルドで使っている別のクラス(たとえばbuildSrc
ディレクトリのクラス)の中から、Gradleのロギングシステムへログを出力することもできます。やり方

例18.3 SLF4Jでログを出力する

build.gradle

```
import org.slf4j.Logger
import org.slf4j.LoggerFactory

Logger slf4jLogger = LoggerFactory.getLogger('some-logger')
slf4jLogger.info('An info log message logged using SLF4J')
```

18.3. 外部ツールやライブラリからのログについて

内部的には、GradleはAntとIvyを使っています。どちらも、自身で独自のロギングシステムを持っていま
TRACEだけはGradleのDEBUG
に対応)。つまり、Gradleのデフォルトログレベルだと、Ant、Ivyのメッセージはエラーやワーニングで

ログに標準出力を使っているツールは、いまだにたくさんあります。デフォルトでは、Gradleは標準出
QUIETレベルログに、標準エラーをERROR
レベルログにリダイレクトします。また、この動作は設定で変更可能です。プロジェクトオブジェクトの
LoggingManager
を使えば、ビルドスクリプト実行時の標準出力、標準エラーをどのログレベルにリダイレクトするか変更

例18.4 標準出力のキャプチャ設定

build.gradle

```
logging.captureStandardOutput LogLevel.INFO
println 'A message which is logged at INFO level'
```

さらに、タスク実行時のログリダイレクト設定も変更できるよう、タスクにもLoggingManager
があります。

例18.5 タスク実行時の標準出力キャプチャ設定

build.gradle

```
task logInfo {
    logging.captureStandardOutput LogLevel.INFO
    doFirst {
        println 'A task message which is logged at INFO level'
    }
}
```

Gradleは、Java標準のロギング実装や、Jakarta CommonsのCommons
Logging、Log4jも統合しています。ビルドで使っているクラスが、これらのツールキットでログメッセー

18.4. Gradleがロギングするものを変更する

GradleのロギングUIを、独自のものに差し替えることができます。もっと多くの情報を出したい場合、
Gradle.useLogger()
メソッドを使います。次の例では、初期化スクリプトを使って、タスク実行、ビルドの完了をロギングす

例18.6 Gradleがロギングするものを変更する

init.gradle

```
useLogger(new CustomEventLogger())

class CustomEventLogger extends BuildAdapter implements TaskExecutionListener {

    public void beforeExecute(Task task) {
        println "[$task.name]"
    }

    public void afterExecute(Task task, TaskState state) {
        println()
    }

    public void buildFinished(BuildResult result) {
        println 'build completed'
        if (result.failure != null) {
            result.failure.printStackTrace()
        }
    }
}
```

gradle -I init.gradle build の出力

```
> gradle -I init.gradle build
[compile]
compiling source

[testCompile]
compiling test source

[test]
running unit tests

[build]

build completed
```

独自ロガーは、以下にリストしたリスナーインターフェースを実装できます。登録したロガーのインター
「ライフサイクルからの通知に応答する」をご参照ください。

- `BuildListener`
- `ProjectEvaluationListener`
- `TaskExecutionGraphListener`
- `TaskExecutionListener`
- `TaskActionListener`

19
Gradleデーモン

19.1. デーモン入門

Gradleデーモン(ビルドデーモンと呼ばれることもあります)はGradleの起動と実行の速度を改善します。

デーモン実行が非常に有用だと思われるユースケースをいくつか考えてみました。
例えば、少数の、比較的実行が早く終わるようなタスクを、何度も実行しなければならないような場面か

- テスト駆動開発にて、ユニットテストを度々実行するような場合
- Webアプリケーション開発にて、アプリケーションを何度も構築する場合
- ビルドの挙動を調べるために、**`gradle tasks`**を何度も実行する場合

これらの場合、Gradleを起動・実行するための時間は出来る限り短いことが重要でしょう。

さらに、Gradleモデルがより素早く構築されるようになれば、ユーザーインターフェースが面白い機能を
例えば、次のようなケースでデーモンが役に立つかもしれません。

- 統合開発環境で入力補完する場合
- GUIでビルド状況を表示する場合
- コマンドラインインターフェースでのタブ補完する場合

一般的にいって、ビルドツールが軽快に動作して困るということはありません。
ローカル環境のビルドでデーモンを試してみてください。デーモンを使わない普通のビルドにはもう戻れ

ツールAPI (参照 Chapter 63, Embedding Gradle)は常にデーモンを使用します。つまり、通常、デーモンなしでツールAPIは使えません。
したがって、EclipseのSTS GradleプラグインやIntelliJ IDEAのGradleサポートを使っているときは、常にデーモンが背後で起動しています。

今後は下記の機能が追加される予定です。

- 軽快な更新チェック: ファイルシステムネイティブの変更通知を(jdk7 nio.2経由などで)使い、プロジェクトが更新されたかどうか前もって解析する
- 更なるビルドの高速化: あらかじめプロジェクトを評価し、次回のGradle実行時にプロジェクトモデルを準備しておく
- 更なるビルドの高速化、はもう言いましたっけ、とにかく、デーモンは潜在的にはあらかじめ依存関
- 再利用可能なプロセスのプールを活用して、コンパイル、テストを実施する。
  例えばGroovyとScalaコンパイラーの起動にはどちらも多大なコストがかかります。
  しかし、ビルドデーモンは、GroovyやScalaが既にロードされたプロセスを維持しておくことができま
- コンパイルなど、ある種のタスクは事前に実行しておく。より早いフィードバックが可能になります。

- 速くて正確なbash上のタブ補完
- Gradleキャッシュを定期的にガーベージコレクトする

19.2. デーモンの再利用と期限切れ

デーモン実行の基本にある考え方は、gradleコマンドがデーモンプロセスを起動して、そのデーモンが実
そうすれば、後続のgradleコマンドがそのデーモンプロセスを再利用でき、起動に掛かるコストを抑える
しかし、既存のデーモンプロセスが使用できないこともあります。例えば、プロセスがビジーだったり、
新しいデーモンプロセスが、正確にはいつ起動されるかの詳細については、以下を参照してください。
デーモンプロセスは、アイドル状態で3時間経過すると自動的に期限切れになります。

デーモンプロセスが新しく起動される場面を全て列挙すると、

- 既存のデーモンがある処理を実行中の場合、新たなデーモンが起動されます。
- デーモンプロセスは、java
  homeごとに起動されます。アイドル状態のデーモンがビルドリクエストを待機していたとしても、別
  homeからビルドを実行した場合デーモンプロセスが新しく起動されます。
- ビルドのJVM引数が大きく異なる場合、デーモンプロセスが新しく起動されます。いくつかのシステ
- 起動されているデーモンは、特定のバージョンのGradleと関連づいています。
  そのため、アイドル状態のデーモンが存在したとしても、それと別のバージョンのGradleでビルドを
  これは、`--stop`コマンドについてあることを教唆していますーーつまり、`--stop`
  で停止できるデーモンは、その`--stop`
  を実行しているGradleと同じバージョンのGradleで起動されたデーモンだけだということです。

今後、デーモンの管理方法、プール方法の改善を行う予定です。

19.3. 使い方とトラブルシューティング

コマンドラインの利用方法については別節付録D　Gradle　コマンドラインを参照してください。
同じコマンドラインを何度も何度も指定してうんざりしている場合は、
「gradle.propertiesを使用したビルド環境の構築」を参照してください。
上記の節には、デーモンの振る舞いをより「永続的に」設定する方法が(デフォルトでデーモンを起動す

デーモンのトラブルシューティングについていくつか方法を示します。

- ビルドで問題が発生した場合、一時的にデーモンを無効化してみてください(コマンドラインにスイッ
  `--no-daemon`を渡す)。
- `--stop`
  オプションや、もっと強制的な方法でデーモンを停止しなければならない場面もあります。
- デーモンのログファイルは、デフォルトではGradleユーザーホームディレクトリーにあります。
- ビルドの実行状況を見るには、`--feoreground`を使ってデーモンを起動します。

19.4. デーモンの設定

デーモン設定のうち、JVM引数、メモリーの設定、Java homeなどは設定ファイルなどで変更できます。詳細については「gradle.propertiesを使用したビルド環境の構築」を参照してください。

20
ビルド環境

20.1. gradle.propertiesを使用したビルド環境の構築

Gradleでは、ビルドを実行するJavaプロセスの設定を簡単に変更するための方法が複数用意されています
ローカルの環境変数でGRADLE_OPTSやJAVA_OPTSを使って設定することもできますが、一部の設定値、
home、デーモンのオンオフなどについてはプロジェクトのVCSに格納してバージョン管理しておけば、
こうした一貫した設定値は、`gradle.properties`に設定値を保存するだけで実現可能です。
設定値は以下の順序で適用されます（複数箇所で設定されている場合は最後の一つが優先されます）。

- プロジェクトのビルドディレクトリに置かれた`gradle.properties`
- `gradle`に置かれた`gradle.properties`
- `-Dsome.property`などで指定されたシステムプロパティ

以下のプロパティがGradleのビルド環境を設定するために使用されます。

`org.gradle.daemon`
: `true`に設定するとデーモンによりビルドが実行されます。
開発者がローカルでビルドを実行する際は、このプロパティを設定することをおすすめします。開発
一方、CI環境では並行性、信頼性といった点を重視したいため、CIビルド（というか実行時間の長い

`org.gradle.java.home`
: GradleのビルドシステムでJava homeを指定します。 設定するのは`jdk`でも`jre`
構いませんが、プロジェクトによってはjdkを指定した方が安全かもしれません。
指定しなければ、適当なデフォルト値が設定されます。

`org.gradle.jvmargs`
: デーモンプロセスに渡すJVM引数を指定します。
この設定はメモリ設定を調節するのに特に便利です。
デフォルト値はメモリに応じて適切に設定されます。

`org.gradle.configureondemand`
: Gradleがプロジェクトを選択的に設定する試験的なモードを有効にします。
関係のあるプロジェクトのみを評価することで、巨大なマルチプロジェクト・ビルドをより高速にビ
詳しくは「Configuration on demand」を参照してください。

`org.gradle.parallel`
: この設定を行うと、Gradleの試験的な並列ビルドモードが有効化されます。

20.1.1. Javaプロセスのフォーク

JVMに関する設定の多く(Javaバージョンや最大ヒープサイズなど)は、新しくJVMプロセスを起動すると
これはつまり、Gradleが解析した多様な`gradle.properties`
に従ってビルドを実行するには、JVMプロセスを新しく起動しなければならないということを意味します
デーモンを実行しているときは、適切なパラメーターが設定されたJVMが起動され、デーモンごとに再利
デーモンを使用せずにビルドを実行したときは、GradleのスタートスクリプトGが起動したJVMとたまたま

ビルドを実行する度にJVMを余分に起動しなければならないというのは、とても大きなコストになります
`org.gradle.java.home`や`org.gradle.jvmargs`
を設定しているのであればGradleデーモンを使ってビルドすることを強く推奨します。詳細については
19章Gradleデーモン を参照してください。

20.2. プロクシ経由のWebアクセス

(例えば依存関係のダウンロードするための)HTTPプロクシの設定は
標準的なJVMのシステムプロパティを使って行います。これらの設定は、直接ビルドスクリプト上で設定
例えばプロクシのホストの為に `System.setProperty('http.proxyHost', 'www.somehost.org`
と設定できます。 あるいは、ビルドのルートディレクトリ上 又は
Gradleホームディレクトリ上にある gradle.propertiesファイルにそれらの値を設定できます。

例20.1 HTTPプロクシの設定

gradle.properties

```
systemProp.http.proxyHost=www.somehost.org
systemProp.http.proxyPort=8080
systemProp.http.proxyUser=userid
systemProp.http.proxyPassword=password
systemProp.http.nonProxyHosts=*.nonproxyrepos.com|localhost
```

HTTPS設定の分離

例20.2 HTTPSプロクシの設定

gradle.properties

```
systemProp.https.proxyHost=www.somehost.org
systemProp.https.proxyPort=8080
systemProp.https.proxyUser=userid
systemProp.https.proxyPassword=password
systemProp.https.nonProxyHosts=*.nonproxyrepos.com|localhost
```

使用可能な全てのプロクシ設定を概観できる良い資料については残念ながら発見することができませんで
Subversionへのリンク を載せておきます。他には、JDKドキュメントに
ネットワーク設定値のページ がありました。
その他、もっと良い資料をご存知であれば、メーリングリストでお知らせください。

20.2.1. NTLM認証

NTLM認証が必要なプロクシを使う場合、ユーザ名とパスワードだけでなく認証ドメインを設定する必要

- システムプロパティ`http.proxyUser`に、*domain*/*username*のように値を設定する。
- システムプロパティ`http.auth.ntlm.domain`に認証ドメインを設定する。

21
Gradleのプラグインについて

Gradleのコア部分には、実際の自動化に役に立つような機能は含まれていません。
これは意図的なもので、Javaのコードをコンパイルしたりといった便利な機能は、全てプラグイン
に よ り 追 加 さ れ ま す 。
プラグインは、コア部分のオブジェクトや他のプラグインのオブジェクトを拡張するだけでなく、新しい`JavaCompile`など)やドメインオブジェクト(`SourceSet`など)、規約(Javaのコードは`src/main/java`に置く)、などをGradleに追加します。

この章では、プラグインを使う方法、プラグインまわりの用語や概念について議論します。

21.1. Types of plugins

There are two general types of plugins in Gradle, script plugins and binary plugins. Script plugins are additional build scripts that further configure the build and usually implement a declarative approach to manipulating the build. They are typically used within a build although they can be externalized and accessed from a remote location. Binary plugins are classes that implement the `Plugin` interface and adopt a programmatic approach to manipulating the build. Binary plugins can reside within a build script, within the project hierarchy or externally in a plugin jar.

21.2. プラグインの適用

プラグインの、いわゆる適用は、`Project.apply()`メソッドで行います。

例21.1 プラグインの適用

build.gradle

```
apply plugin: 'java'
```

21.2.1. Script plugins

例21.2 Applying a script plugin

build.gradle

```
apply from: 'other.gradle'
```

Script plugins can be applied from a script on the local filesystem or at a remote location. Filesystem locations are relative to the project directory, while remote script locations are specified with an HTTP URL. Multiple script plugins (of either form) can be applied to a given build.

21.2.2. Binary plugins

例21.3 Applying a binary plugin

build.gradle

```
apply plugin: 'java'
```

Core plugins register a short name. In the above case, we are using the short name 'java' to apply the JavaPlugin. Plugins also have a plugin id that takes a fully qualified form like com.githu
, although some legacy plugins may still utilize the short, unqualified form.

This method can also accept a class to identify the plugin:

例21.4 Applying a binary plugin by type

build.gradle

```
apply plugin: JavaPlugin
```

The JavaPlugin symbol in the above sample refers to the the JavaPlugin. This class does not strictly need to be import as the org.gradle.api.plugins package is automatically imported in all build scripts (see 付録E IDE対応の現状と、IDEによらない開発支援). Furthermore, it is not necessary to append .class to identify a class literal in Groovy as it is in Java.

The application of plugins is idempotent. That is, a plugin can be applied multiple times. If the plugin has previously been applied, any further applications will have no effect.

21.2.2.1. Locations of binary plugins

A plugin is simply any class that implements the Plugin interface. Gradle provides the core plugins as part of its distribution so simply applying the plugin as above is all you need to do. However, non-core binary plugins need to be available to the build classpath before they can be applied. This can be achieved in a number of ways, including:

- Defining the plugin as an inline class declaration inside a build script.
- Defining the plugin as a source file under the buildSrc directory in the project.
- Including the plugin from an external jar defined as a buildscript dependency (see 「ビルドスクリプトで外部ライブラリを使うときの依存関係設定」).
- Including the plugin from the plugin portal using the plugins DSL (see 「Applying plugins with the plugins DSL」).

For more on defining your own plugins, see 59章カスタムプラグインの作成.

21.3. Applying plugins with the plugins DSL

> The plugins DSL is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The new plugins DSL provides a more succinct and convenient way to declare plugin dependencies. It works with the new Gradle plugin portal to provide easy access to both core and community plugins. The plugins script block configures an instance of `PluginDependenciesSpec`.

プラグインは、自分自身を示す短縮名を保持、公開しています。 上の例では、短縮名'`java`'を使って`JavaPlugin`を適用しています。

以下は短縮名を使わなかった場合の例です。

例21.5 型でプラグインを適用する

build.gradle

```
apply plugin: JavaPlugin
```

例21.6 Applying a community plugin

build.gradle

```
plugins {
    id "com.jfrog.bintray" version "0.4.1"
}
```

No further configuration is necessary. Specifically, there is no need to configure the buildscript classpath. Gradle will resolve the plugin in the plugin portal, locate it, and make it available to the build.

See `PluginDependenciesSpec` for more information on using the Plugin DSL.

21.4. Finding community plugins

Gradle has a vibrant community of plugin developers who contribute plugins for a wide variety of capabilities. The Gradle plugin portal provides an interface for searching and exploring community plugins.

また、プラグインの作り方は59章カスタムプラグインの作成に詳しく記載されています。

21.5. プラグインがすること

プラグインを適用することで、そのプロジェクトの機能を拡張することができます。例えば、

- プロジェクトにタスクを追加する(compileやtestなど)。
- タスクに、便利なデフォルト値を事前に設定しておく。
- 依存関係のコンフィギュレーションをプロジェクトに追加する(「依存関係のコンフィグレーション 」参照)。
- 既存の型に新しいプロパティやメソッドを追加する。

次の例を見てください。

例21.7 プラグインにより追加されたタスク

build.gradle

```
apply plugin: 'java'

task show << {
    println relativePath(compileJava.destinationDir)
    println relativePath(processResources.destinationDir)
}
```

gradle -q show の出力

```
> gradle -q show
build/classes/main
build/resources/main
```

Javaプラグインは、`compileJava`タスクと`processResource`タスクをプロジェクトに追加し、それぞれのタスクに`destinationDir`プロパティを設定しています。

21.6. 規約

プラグインは、CoC(設定より規約)をというスマートな方法をサポートするため、プロジェクトを事前に
Gradleにはこのメカニズムと洗練されたサポートがあり、パワフルかつ簡潔なビルドスクリプトを記述す

先ほどの例、Javaプラグインが`compileJava`タスクを追加し、そこに`destinationDir`プロパティ(コンパイルしたソースが出力される場所)を追加したコードを思い出してください。
Javaプラグインは、デフォルトでこのプロパティが、プロジェクトディレクトリの`build/classes/ma`を指すよう設定します。これは、合理的なデフォルト値を設定するCoCの一例です。

このプロパティは、新しい値を設定するだけで簡単に変更できます。

例21.8 プラグインのデフォルトを変更する

build.gradle

```
apply plugin: 'java'

compileJava.destinationDir = file("$buildDir/output/classes")

task show << {
    println relativePath(compileJava.destinationDir)
}
```

gradle -q show の出力

```
> gradle -q show
build/output/classes
```

ただ、クラスファイルの出力場所が関係するタスクは、おそらくcompileJava
タスクだけではないでしょう。

Javaプラグインは、プロジェクトにソースセットという概念を追加します(SourceSet参照)。
ソースセットは、ソースコード一式を示す概念であり、それらがコンパイルされた際にクラスファイルが
Javaプラグインは、compileJavaタスクのdestinationDir
プロパティを、ソースセットが示す出力場所にマップしているのです。

クラスファイルの書き出される場所を、ソースセットで変更することができます。

例21.9 プラグインの規約オブジェクト

build.gradle

```
apply plugin: 'java'

sourceSets.main.output.classesDir = file("$buildDir/output/classes")

task show << {
    println relativePath(compileJava.destinationDir)
}
```

gradle -q show の出力

```
> gradle -q show
build/output/classes
```

この例で、Javaプラグインは以下のようなことを含む様々な処理を行っています。

- 新しいドメインオブジェクト、SourceSetを追加。
- mainソースセットに、デフォルト値(つまり規約)を設定。
- タスクがこれらのプロパティを使って動作するように設定。

この全てが、apply plugin: "java"
のステップで行われています。例では、規定の設定が行われた後、クラスファイルの場所を変更
しました。compileJava.destinationDir
の値が、設定の変更を反映して変わっていることに注目してください。

別のタスクが、クラスファイルを使用するケースを考えてください。そのタスクが、クラスファイルの場
`sourceSets.main.output.classesDir`
を参照して取得するようになっていれば、ソースセットを変更するだけで`compileJava`
とそのタスク、両方の設定が更新されます。

他のタスクの設定値をいつでも(つまり、変更されたときでさえ)反映するよう、オブジェクトのプロパテ
規約マッピングとして知られています。
この方法により、GradleはCoCと合理的なデフォルト値による簡潔さを達成しています。また、規定のラ
もしこの方法を採用していなければ、クラスファイルの設定をするのに全てのオブジェクトを変更してま

21.7. プラグインをさらに詳しく知るには

この章は、プラグインと、プラグインがGradleで果たしている役割について簡単に紹介するためのものて
プラグインの内部動作について、さらに詳しく知るには、59章カスタムプラグインの作成
を参照してください。

22
標準Gradleプラグイン

Gradleの配布物にはいくつかのプラグインが同梱されています。以下にその一覧を掲載します。

22.1. 言語プラグイン

これらのプラグインは、Gradleに様々な言語サポートを追加します。JVMに向けたコンパイルやJVM上て

表22.1 言語プラグイン

プラグインID	自動適用	協調して動作	説明
java	java-base	-	Javaのコンパイル、テスト、バンドル機能をプロジ 7章Javaクイックスタートも参照してください。
groovy	java, groovy-base	-	Groovyプロジェクトのビルド機能を追加します。
scala	java, scala-base	-	Scalaプロジェクトのビルド機能を追加します。
antlr	java	-	Antlrを使ったパーサーの生成をサポートします。

22.2. 試験的な言語プラグイン

これらのプラグインは、以下のような様々な言語のサポートを追加します。

表22.2 言語プラグイン

プラグインID	自動適用	協調して動作	説明
assembler	-	-	Adds native assembly language capabilities to a project.
c	-	-	Adds C source compilation capabilities to a project.
cpp	-	-	Adds C++ source compilation capabilities to a project.
objective-c	-	-	Adds Objective-C source compilation capabilities to a project.
objective-cpp	-	-	Adds Objective-C++ source compilation capabilities to a project.
windows-resources	-	-	Adds support for including Windows resources in native binaries.

22.3. 統合プラグイン

これらのプラグインは、様々なランタイムをプロジェクトに統合して使用できるようにします。

表22.3 統合プラグイン

プラグインID	自動適用	協調して動作	説明
application	java	-	プロジェクトをコマンドラインアプリケーションと
ear	-	java	J2EEアプリケーションのビルド機能を追加します。
jetty	war	-	組み込みのJettyウェブコンテナに、アプリケーシ Webアプリケーションクイックスタートも参照して
maven	-	java, war	Mavenリポジトリにアーティファクトを公開できる
osgi	java-base	java	OSGiバンドルのビルドをサポートします。
war	java	-	ウェブアプリケーションのWARファイルをビルド Webアプリケーションクイックスタートも参照して

22.4. 試験的な統合プラグイン

これらのプラグインは、様々なランタイムをプロジェクトに統合して使用できるようにします。

表22.4 試験的な統合プラグイン

プラグインID	自動適用	協調して動作	説明
distribution	-	-	ZIPおよびTARの配布物をビル
java-library-distribution	java, distribution	-	JavaライブラリをZIPおよびT
ivy-publish	-	java, war	Ivyリポジトリへアーティファ
maven-publish	-	java, war	Mavenリポジトリへアーティ

22.5. ソフトウェア開発プラグイン

これらのプラグインは、ソフトウェアの開発プロセスを支援します。

表22.5 ソフトウェア開発プラグイン

プラグインID	自動適用	協調して動作	説明
announce	-	-	TwitterやGrowlなど、任意のプラ
build-announcements	announce	-	ビルドライフサイクルの各種イベ
checkstyle	java-base	-	Checkstyleを使ってJavaソースコ
codenarc	groovy-base	-	CodeNarcを使ってGroovyソース
eclipse	-	java,groovy, scala	Eclipse IDEが使用するファイ Javaクイックスタートも参照して
eclipse-wtp	-	ear, war	eclipseプラグインが生成する Platform)の設定ファイル eclipseにインポートすれば、war Javaクイックスタートも参照して
findbugs	java-base	-	FindBugsを使ってプロジェクトに
idea	-	java	Intellij IDEA IDEが使用するファイ
jdepend	java-base	-	JDependを使ってソースコードの
pmd	java-base	-	PMDを使ってJavaソースコードの
project-report	reporting-base	-	Gradleビルドに関する有用な情報
signing	base	-	ビルドしたファイルとアーティフ
sonar	-	java-base, java, jacoco	Sonarコード品質プラットフォー プラグインが作成されています。

22.6. 試験的なソフトウェア開発プラグイン

これらのプラグインは、ソフトウェアの開発プロセスを支援します。

表22.6 ソフトウェア開発プラグイン

プラグインID	自動適用	協調して動作	説明
`build-dashboard`	reporting-base	-	Generates build dashboard report.
`build-init`	wrapper	-	Adds support for initializing a n Handles converting a Maven build to
`cunit`	-	-	Adds support for running CUnit tests.
`jacoco`	reporting-base	java	Provides integration with the JaCo library for Java.
`sonar-runner`	-	java-base, java, jacoco	S o n a r コード品質プラットフォームとの連携 プラグインの後継プラグインです。
`visual-studio`	-	native language plugins	Adds integration with Visual Studio.
`wrapper`	-	-	Adds a `Wrapper` task for generatin files.
`java-gradle-plugin`	java		Assists with development of Gr providing standard plugin build validation.

22.7. ベースプラグイン

これらのプラグインは、他のプラグインを組み立てるための基本的なビルド用の部品を構成します。また
しかし、これらのプラグインはまだGradleの正式な公開APIとはなっていないことに注意してください。

表22.7 ベースプラグイン

プラグインID	説明
base	一般的なライフサイクルを構成するタスクを追加し、アーカイブタスクにリーズナ • build*ConfigurationName* タスクを追加します。これらのタスクは、特定のコンフィグレーションに属する • upload*ConfigurationName* タスクを追加します。これらのタスクは、特定のコンフィグレーション属するア • 全てのアーカイブタスク（`AbstractArchiveTask`を継承したタスク）に適切な タイプや`Tar`タイプ、`Zip`タイプのタスクがアーカイブタスクです。 これらのタ プロパティを前もって設定します。 これにより、アーカイブの命名規約やビルド後の配置場所に一貫性が生まれるた
java-base	プロジェクトにソースセットの概念を導入します。実際のソースセットは追加されま
groovy-base	プロジェクトにGroovyソースセットの概念を導入します。
scala-base	プロジェクトにScalaソースセットの概念を導入します。
reporting-base	レポート生成に関するいくつかの一般的な規約プロパティをプロジェクトに追加しま

22.8. サードパーティプラグイン

サードパーティ製のプラグインはGradleプラグインポータルで見つけることができます。

23
Javaプラグイン

Javaプラグインは、プロジェクトにJavaのコンパイル、テスト、そしてビルド能力を与えます。このプラ

23.1. 使用方法

Javaプラグインを使うためには、ビルドスクリプトに下記を含めます：

例23.1 Javaプラグインの使用

build.gradle

```
apply plugin: 'java'
```

23.2. ソースセット

Javaプラグインは ソースセット
の概念を導入します。ソースセットは、一緒にコンパイルされ実行されるソースファイルのグループです

ソースセットを使うと、ソースファイルを論理的なグループに分割し、それぞれ別個の目的を持たせるこ

Javaプラグインは、`main`と`test`という二つの標準ソースセットを定義します。`main`
ソースセットは、コンパイルされJARファイルを構成する製品ソースコードを含みます。`test`
ソースセットは、コンパイルされJUnitやTestNGで実行されるユニットテストコードを含みます。

23.3. タスク

Javaプラグインは、以下に示すような数多くのタスクをプロジェクトに追加します。

表23.1 Javaプラグイン - タスク

タスク名	依存先
compileJava	コンパイル時クラスパスを作り出すすべてのタスク。 compile コンフィギュレーションに含まれる、プロジェクトの依存関係のためのjarタスクを含む
processResources	-
classes	compileJava と processResources。一部のプラグインはさらにコンパイルタスクを追加する
compileTestJava	compile 、そしてテストのコンパイル時クラスパスを作り出すすべてのタスク
processTestResources	-
testClasses	compileTestJava と processTestResources 。一部のプラグインはさらにテストコンパイルタスクを追加する
jar	compile
javadoc	compile
test	compile と compileTest 、そしてテストの実行時クラスパスを作り出すすべてのタスク
uploadArchives	archives コンフィギュレーションのアーティファクトを生成するタスク （ jar を含む）
clean	-
clean*TaskName*	-

Javaプラグインは、プロジェクトに追加されるソースセットの各々について以下のコンパイルタスクを追

表23.2 Javaプラグイン - ソースセットタスク

タスク名	依存先	型
compile*SourceSet*Java	このソースセットのコンパイル時クラスパスを作り出すすべてのタスク	Java
process*SourceSet*Resources	-	Copy
*sourceSet*Classes	compile*SourceSet*Java と process*SourceSet*Resources 。一部のプラグインはさらにこのソースセットのコンパイルタスクを追加する	Task

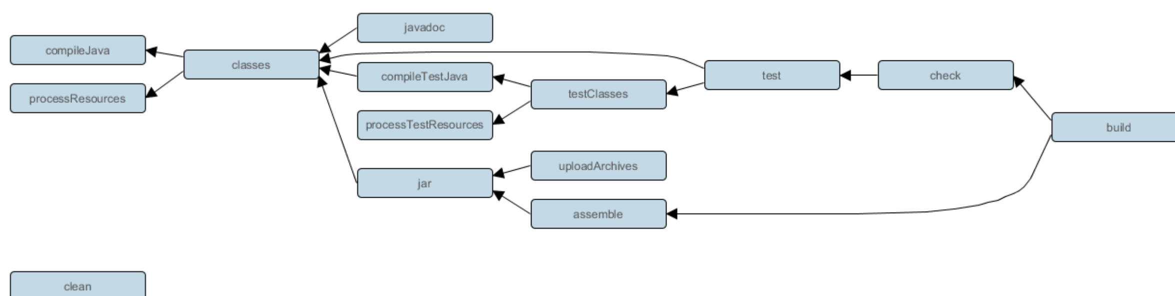
Javaプラグインは、プロジェクトのライフサイクルを構成する数多くのタスクも追加します：

表23.3 Javaプラグイン - ライフサイクルタスク

タスク名	依存先
assemble	jarを含む、プロジェクトのすべてのアーカイブタスク。 一部のプラグインはプロジェクトにさらにアーカイブタスクを追加する
check	testを含む、プロジェクトのすべての検証タスク。 一部のプラグインはプロジェクトにさらに検証タスクを追加する
build	check と assemble
buildNeeded	build と、testRuntimeコンフィギュレーションのプロジェクトライブラリ依 タスク
buildDependents	build と、testRuntime コンフィギュレーションのプロジェクトライブラリ依存先がこのプロジェクト buildDependentsタスク
build*ConfigName*	*ConfigName*コンフィギュレーションのアーティファクトを作り出すタスク
upload*ConfigName*	*ConfigName*コンフィギュレーションのアーティファクトをアップロードする

下の図は、上記のタスク間の関係を示しています。

図23.1 Javaプラグイン - タスク



23.4. プロジェクトレイアウト

Javaプラグインは、以下のようなプロジェクトレイアウトを仮定しています。これらのディレクトリは、

表23.4 Javaプラグイン - デフォルトプロジェクトレイアウト

ディレクトリ	意味
`src/main/java`	製品のJavaソース
`src/main/resources`	製品のリソース
`src/test/java`	テストのJavaソース
`src/test/resources`	テストのリソース
src/*sourceSet*/java	特定のソースセットのJavaソース
src/*sourceSet*/resources	特定のソースセットのリソース

23.4.1. プロジェクトレイアウトの変更

プロジェクトのレイアウトは、適切なソースセットを設定することでカスタマイズできます。これについ

例23.2 Javaソースレイアウトのカスタマイズ

build.gradle

```
sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
    }
}
```

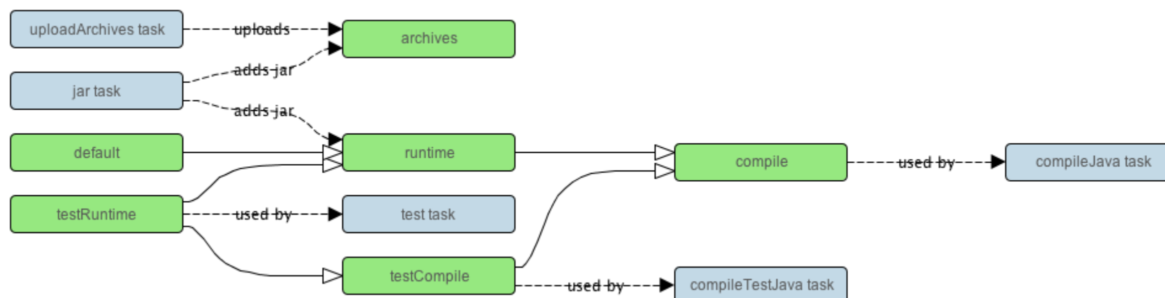
23.5. 依存関係の管理

Javaプラグインは、以下のような多くの依存関係のコンフィギュレーションをプロジェクトに追加します
これらのコンフィギュレーションは`compileJava`や`test`といったタスクに割り当てられます。

表23.5 Javaプラグイン - 依存関係のコンフィギュレーション

名前	拡張元	利用するタスク	意味
compile	-	compileJava	コンパイル時の依存関係
runtime	compile	-	実行時の依存関係
testCompile	compile	compileTestJava	テストのコンパイル時の追加依存関係
testRuntime	runtime, testCompile	test	テストの実行時の追加依存関係
archives	-	uploadArchives	このプロジェクトが生成するアーティファクト(ja
default	runtime	-	このプロジェクトに依存するプロジェクトが使用

図23.2 Javaプラグイン - 依存関係のコンフィギュレーション



プロジェクトに追加されたそれぞれのソースセットに対し、Javaプラグインは次の依存関係コンフィグレ

表23.6 Javaプラグイン - ソースセットの依存関係コンフィグレーション

名前	拡張元	利用するタスク	意味
*sourceSet*Compile		compile*SourceSet*Java	対象ソースセットのコンパイル時依存関係
*sourceSet*Runtime	*sourceSet*Compile		対象ソースセットの実行時依存関係

23.6. 規約プロパティ

Javaプラグインは、以下のような多くの規約プロパティをプロジェクトに追加します。
これらは、ビルドスクリプトの中で、あたかもprojectオブジェクトのプロパティであるかのように使うこ
(「規約」参照)。

表23.7 Javaプラグイン - ディレクトリプロパティ

プロパティ名	型	デフォルト値	説明
reportsDirName	String	reports	レポートを生成するラ
reportsDir	File (read-only)	*buildDir/reportsDirName*	レポートを生成するラ
testResultsDirName	String	test-results	テスト結果の.xmlファ
testResultsDir	File (read-only)	*buildDir/testResultsDirName*	テスト結果の.xmlファ
testReportDirName	String	tests	テストレポートを生成
testReportDir	File (read-only)	*reportsDir*/testReportDirName	テストレポートを生成
libsDirName	String	libs	ライブラリを生成する
libsDir	File (read-only)	*buildDir/libsDirName*	ライブラリを生成する
distsDirName	String	distributions	ディストリビューショ
distsDir	File (read-only)	*buildDir/distsDirName*	ディストリビューショ
docsDirName	String	docs	ドキュメントを生成す
docsDir	File (read-only)	*buildDir/docsDirName*	ドキュメントを生成す
dependencyCacheDirName	String	dependency-cache	ソースの依存関係情報
dependencyCacheDir	File (read-only)	*buildDir/dependencyCacheDirName*	ソースの依存関係情報

表23.8 Javaプラグイン - その他のプロパティ

プロパティ名	型	デフォルト値
sourceSets	SourceSetContainer (read-only)	非null
sourceCompatibility	JavaVersion。StringやNumberで設定することも可能。例： '1.5' や 1.5	Value of the current used JVM
targetCompatibility	JavaVersion。StringやNumberで設定することも可能。例： '1.5' や 1.5	*sourceCompatibilit*
archivesBaseName	String	*projectName*
manifest	Manifest	空のマニフェスト

これらのプロパティは、JavaPluginConvention、 BasePluginConvention

などの型の規約オブジェクトによって提供されます。

23.7. ソースセットの利用

例23.3 ソースセットへのアクセス

build.gradle

```
// Various ways to access the main source set
println sourceSets.main.output.classesDir
println sourceSets['main'].output.classesDir
sourceSets {
    println main.output.classesDir
}
sourceSets {
    main {
        println output.classesDir
    }
}

// Iterate over the source sets
sourceSets.all {
    println name
}
```

既存のソースセットを設定するには、上記のアクセスメソッドの一つを使って、そのソースセットのプロ
プロパティについては後述します。以下に、mainのJavaソースとリソースディレクトリを設定する例を示

例23.4 ソースセットのソースディレクトリの設定

build.gradle

```
sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
    }
}
```

23.7.1. ソースセットプロパティ

以下の表は、ソースセットの重要なプロパティの一覧です。 さらに詳しくは、`SourceSet`のAPIドキュメントをご覧ください。

表23.9 Javaプラグイン - ソースセットプロパティ

プロパティ名	型	デフォルト値	説明
`name`	`String` (read-only)	非null	このソース
`output`	`SourceSetOutput` (read-only)	Not null	このソース
`output.classesDir`	`File`	*buildDir*/classes/*name*	このソース
`output.resourcesDir`	`File`	*buildDir*/resources/*name*	このソース
`compileClasspath`	`FileCollection`	compile*SourceSet* コンフィグレーション	このソース
`runtimeClasspath`	`FileCollection`	output + runtime*SourceSet* コンフィグレーション	このソース
`java`	`SourceDirectorySet` (read-only)	非null	このソース ファイルの
`java.srcDirs`	`Set<File>`。 設定には「入力ファイルセットを指定する」で述べる方法ならどれでも使用可	[*projectDir*/src/*name*/java]	このソース
`resources`	`SourceDirectorySet` (read-only)	非null	このソース Groovyプ
`resources.srcDirs`	`Set<File>`。 設定には「入力ファイルセットを指定する」で述べる方法ならどれでも使用可	[*projectDir*/src/*name*/resour	このソース
`allJava`	`SourceDirectorySet` (read-only)	`java`	このソース Groovyプ
`allSource`	`SourceDirectorySet` (read-only)	`resources + java`	このソース Groovyプ

23.7.2. 新しいソースセットの定義

新しいソースセットは、sourceSets { }ブロックの中でそれを参照するだけで定義できます。以下がその例です。:

例23.5 ソースセットの定義

build.gradle

```
sourceSets {
    intTest
}
```

ソースセットを定義すると、Javaプラグインは、そのソースセットのために表23.6「Javaプラグイン - ソースセットの依存関係コンフィグレーション」

で示した依存関係のコンフィグレーションを追加します。これらのコンフィグレーションを、そのソース

例23.6 ソースセットの依存関係定義

build.gradle

```
sourceSets {
    intTest
}

dependencies {
    intTestCompile 'junit:junit:4.11'
    intTestRuntime 'org.ow2.asm:asm-all:4.0'
}
```

また、Javaプラグインは、表23.2「Javaプラグイン - ソースセットタスク」に示したような、そのソースセットのクラスを構築するためのタスクを追加します。
例えば、intTestというソースセットを追加すると、**gradle intTestClasses** と実行することでそのソースセットのクラスをコンパイルできるようになります。

例23.7 ソースセットのコンパイル

gradle intTestClasses の出力

```
> gradle intTestClasses
:compileIntTestJava
:processIntTestResources
:intTestClasses

BUILD SUCCESSFUL

Total time: 1 secs
```

23.7.3. ソースセットの実例

ソースセットのクラスを含んだJARを追加します：

例23.8 ソースセットのJARを生成

build.gradle

```
task intTestJar(type: Jar) {
    from sourceSets.intTest.output
}
```

ソースセットのJavadocを生成します：

例23.9 ソースセットのJavadocを生成

build.gradle

```
task intTestJavadoc(type: Javadoc) {
    source sourceSets.intTest.allJava
}
```

ソースセットに含まれるテストを実行するテストスイートを追加します：

例23.10 ソースセットのテストを実行

build.gradle

```
task intTest(type: Test) {
    testClassesDir = sourceSets.intTest.output.classesDir
    classpath = sourceSets.intTest.runtimeClasspath
}
```

23.8. Javadoc

javadocタスクはJavadocのインスタンスです。 javadocの主なオプションと、javadocコマンドのリファレンスガイドに記載されている標準docletのオプションをサポートしています。
サポートしているjavadocオプションの完全なリストについては、次のクラスのAPIドキュメントを参照し
CoreJavadocOptionsおよびStandardJavadocDocletOptions。

表23.10 Javaプラグイン - Javadocプロパティ

プロパティ名	型	デフォルト値
classpath	FileCollection	sourceSets.main.output + sourceSe
source	FileTree。 設定には「入力ファイルセットを指定する」で述べる方法ならどれでも使用可	sourceSets.main.allJava
destinationDir	File	*docsDir*/javadoc
title	String	プロジェクトの名前とバージョン

23.9. Clean

cleanタスクはDeleteのインスタンスです。 dir
プロパティで示されるディレクトリを削除します。

表23.11 Javaプラグイン - Cleanプロパティ

タスクプロパティ	型	デフォルト値
dir	File	*buildDir*

23.10. リソース

Javaプラグインは、リソースの処理にCopyタスクを使います。
プロジェクトの各ソースセットに対して一つのインスタンスを追加します。
copyタスクの詳細については、「ファイルをコピーする 」をご覧ください。

表23.12 Javaプラグイン- processResourcesプロパティ

タスクプロパティ	型	デフォルト値
srcDirs	Object。 設定には「入力ファイルセットを指定する 」で述べる方法ならどれでも使用可	*sourceSet*.resources
destinationDir	File。 設定には「ファイルを参照する 」で述べる方法ならどれでも使用可	*sourceSet*.output.resourc

23.11. CompileJava

Javaプラグインは、プロジェクトの各ソースセットに対して一つのJavaCompile
インスタンスを追加します。
compileタスクは、実際のコンパイル作業をAntのjavacタスクに委譲します。
Antのjavacタスクの大部分のプロパティが設定可能です。

表23.13 Javaプラグイン - Compileプロパティ

タスクプロパティ	型	デフォルト値
classpath	FileCollection	*sourceSet*.compileClass
source	FileTree。 設定には「入力ファイルセットを指定する 」で述べる方法ならどれでも使用可	*sourceSet*.java
destinationDir	File.	*sourceSet*.output.class

デフォルトでは、javaコンパイラはGradleプロセス上で実行されます。設定を options.fork を true にすると、 個別のプロセスでコンパイルを実行します。 Antのjavacタスクを例に挙げると, コンパイルタスクを実行するたびに新しいプロセスが実行される形で
遅延実行させることも可能です。反対に、Gradleのダイレクトコンパイラの利用を有効にした場合 (上記参照) 出来るだけ 同一のプロセスでコンパイルを実行しようとします.
両方利用するケースの場合は、全ての処理実行オプションは options.forkOptions によって指定されるでしょう。

23.12. Incremental Java compilation

Starting with Gradle 2.1, it is possible to compile Java incrementally. This feature is still incubating. See the JavaCompile task for information on how to enable it.

Main goals for incremental compilations are:

- Avoid wasting time compiling source classes that don't have to be compiled. This means faster builds, especially when a change to a source class or a jar does not incur recompilation of many source classes that depend on the changed input.

- Change as few output classes as possible. Classes that don't need to be recompiled remain unchanged in the output directory. An example scenario when this is really useful is using JRebel - the fewer output classes are changed the quicker the jvm can use refreshed classes.

The incremental compilation at a high level:

- The detection of the correct set of stale classes is reliable at some expense of speed. The algorithm uses bytecode analysis and deals gracefully with compiler optimizations (inlining of non-private constants), transitive class dependencies, etc. Example: When a class with a public constant changes, we eagerly compile everything to avoid problems with constants inlined by the compiler. Down the road we will tune the algorithm and caching so that incremental Java compilation can be a default setting for every compile task.
- To make incremental compilation fast, we cache class analysis results and jar snapshots. The initial incremental compilation can be slower due to the cold caches.

23.13. テスト

`test`タスクは`Test`のインスタンスです。 このタスクは、`test`
ソースセットに含まれるすべてのユニットテストを自動的に検出し、実行します。
また、テストの実行が完了するとレポートを生成します。JUnitとTestNGの両方がサポートされています
完全なAPIについては、`Test`を見てください。

23.13.1. テストの実行

テストの実行は別のJVMで行われ、メインのビルドプロセスからは分離されます。`Test`
タスクのAPIを使って、テストの実行を制御できます。

テストプロセスの起動を制御するための多くのプロパティが用意されています。
これにはシステムプロパティや、JVMへの引数、Java実行コマンドなどが含まれます。

テストが並列に実行されるかどうかを指定することができます。
Gradleでは、複数のテストプロセスを同時に実行する並列テスト実行が可能です。
各テストプロセスは一度に一つのテストだけを実行するので、通常、これを活用するために何か特別な準
`maxParallelForks`プロパティで、同時に実行されるテストプロセスの最大個数を指定します。
デフォルトは1で、これはテストが並列実行されないということです。

テストプロセスでは、そのテストプロセスの一意な識別子である`org.gradle.test.worker`
システムプロパティが設定されます。
これは例えばファイル名の一部や、その他のリソースの識別子などに利用することができます。

一定のテストクラスを実行し終わったら、そのテストプロセスが再起動されるように指定することができ
これは、テストプロセスに巨大なヒープを割り当てることに対する便利な代替手段になりえます。 `fork`
プロパティは、一つのテストプロセス内で実行されるテストクラスの最大数を指定します。
デフォルトでは各テストプロセスで無制限な数のテストを実行できます。

このタスクは、テストが失敗したときの振る舞いを制御する`ignoreFailures`
プロパティを持っています。 Testはいつも見つけたテストをすべて実行します。 `ignoreFailures`

がfalseのとき、失敗したテストがあるとそれ以降のビルドは停止されます。 ignoreFailures のデフォルト値はfalseです。

t e s t L o g g i n g
プロパティで、どのテストイベントを、どのログレベルでログに出すべきか設定することができます。デフォルトでは、全ての失敗したテストに対して簡潔なメッセージがログ出力されます。好みに応じてテ TestLoggingContainerを参照してください。

23.13.2. Debugging

The test task provides a `Test.getDebug()` property that can be set to launch to make the JVM wait for a debugger to attach to port 5005 before proceeding with test execution.

This can also be enabled at invocation time via the **`--debug-jvm`** task option (since Gradle 1.12).

23.13.3. Test filtering

Starting with Gradle 1.10, it is possible to include only specific tests, based on the test name pattern. Filtering is a different mechanism than test class inclusion / exclusion that will be described in the next few paragraphs (`-Dtest.single`, `test.include` and friends). The latter is based on files, e.g. the physical location of the test implementation class. File-level test selection does not support many interesting scenarios that are possible with test-level filtering. Some of them Gradle handles now and some will be satisfied in future releases:

- Filtering at the level of specific test methods; executing a single test method
- Filtering based on custom annotations (future)
- Filtering based on test hierarchy; executing all tests that extend ceratain base class (future)
- Filtering based on some custom runtime rule, e.g. particular value of a system property or some static state (future)

Test filtering feature has following characteristic:

- Fully qualified class name or fully qualified method name is supported, e.g. “org.gradle.SomeTest”, “org.gradle.SomeTest.someMethod”
- Wildcard '*' is supported for matching any characters
- Command line option “--tests” is provided to conveniently set the test filter. Especially useful for the classic 'single test method execution' use case. When the command line option is used, the inclusion filters declared in the build script are ignored.
- Gradle tries to filter the tests given the limitations of the test framework API. Some advanced, synthetic tests may not be fully compatible with filtering. However, the vast majority of tests and use cases should be handled neatly.
- Test filtering supersedes the file-based test selection. The latter may be completely replaced in future. We will grow the the test filtering api and add more kinds of filters.

例23.11 Filtering tests in the build script

build.gradle

```
test {
    filter {
        //include specific method in any of the tests
        includeTestsMatching "*UiCheck"

        //include all tests from package
        includeTestsMatching "org.gradle.internal.*"

        //include all integration tests
        includeTestsMatching "*IntegTest"
    }
}
```

For more details and examples please see the `TestFilter` reference.

Some examples of using the command line option:

- `gradle test --tests org.gradle.SomeTest.someSpecificFeature`
- `gradle test --tests *SomeTest.someSpecificFeature`
- `gradle test --tests *SomeSpecificTest`
- `gradle test --tests all.in.specific.package*`
- `gradle test --tests *IntegTest`
- `gradle test --tests *IntegTest*ui*`
- `gradle someTestTask --tests *UiTest someOtherTestTask --tests *WebTest*ui`

23.13.4. Single test execution via System Properties

> This mechanism has been superseded by 'Test Filtering', described above.

Setting a system property of *taskName.single* = *testNamePattern* will only execute tests that match the specified *testNamePattern*. The *taskName* can be a full multi-project path like “:sub1:sub2:test” or just the task name. The *testNamePattern* will be used to form an include pattern of “**/testNamePattern*.class” ;. If no tests with this pattern can be found an exception is thrown. This is to shield you from false security. If tests of more than one subproject are executed, the pattern is applied to each subproject. An exception is thrown if no tests can be found for a particular subproject. In such a case you can use the path notation of the pattern, so that the pattern is applied only to the test task of a specific subproject. Alternatively you can specify the fully qualified task name to be executed. You can also specify multiple patterns. Examples:

taskName.single = *testNamePattern*というシステムプロパティを設定すると、指定された*testNamePattern*にマッチするテストだけが実行されます。 *taskName*には、":sub1:sub2:test"のような完全なマルチプロジェクトパスか、単なるタスク名が使えます。 *testNamePattern*は、"**/testNamePattern*.class"という形式のincludesパターンを構成するのに使われます。

このパターンでテストが一つも見つからない場合は例外が発生します。
これは手違いによる偽りの安心を防ぐためです。
複数のサブプロジェクトに対してテストが実行される場合、パターンは各サブプロジェクトに適用されま
ある一つのサブプロジェクトでテストが見つからなかった場合でも例外が発生します。
このような場合は、パターンのパス記法を使って、パターンが特定のサブプロジェクトのtestタスクだけ
あるいは、完全修飾タスク名を使って実行するタスクを指定することも可能です。
パターンは複数指定することもできます。以下は実例です：

- `gradle -Dtest.single=ThisUniquelyNamedTest test`
- `gradle -Dtest.single=a/b/ test`
- `gradle -DintegTest.single=*IntegrationTest integTest`
- `gradle -Dtest.single=:proj1:test:Customer build`
- `gradle -DintegTest.single=c/d/ :proj1:integTest`

23.13.5. テストの検出

T e s t
タスクは、コンパイル済みのtestクラスを調べて、どのクラスがテストクラスかを見つけだします。
デフォルトでは、すべての`.class`ファイルがスキャンされます。 カスタムの includes / excludes
を設定して、特定のクラスだけをスキャンするようにもできます。
テストクラスの検出には、使用するテストフレームワーク(JUnit /
TestNG)によって異なる基準が使われます。

JUnitを使っている場合は、JUnit 3と4両方のテストクラスをスキャンします。
以下のいずれかの基準に合致する場合、そのクラスはJUnitのテストクラスだと見なされます：

- そのクラスまたはスーパークラスが `TestCase`か`GroovyTestCase`をextendsしている
- そのクラスまたはスーパークラスに `@RunWith`アノテーションが付いている
- そのクラスまたはスーパークラスが`@Test`アノテーションが付いたメソッドを含んでいる

TestNGを使っている場合は、`@Test`アノテーションが付いたメソッドをスキャンします。

抽象クラスは実行されないことに注意してください。
Gradleは、testクラスパス上のjarファイルに含まれる継承ツリーもスキャンします。

上記のテストクラス検出を使いたくないときは、`scanForTestClasses`
をfalseに設定することで、これを無効にすることができます。
この場合、testタスクがテストクラスを探すときにはincludes / excludesだけが使われます。 `scanForT`
が無効になっていて、かつincludesやexcludesパターンが指定されていないときは、
デフォルトとしてincludesには`"**/*Tests.class"`, `"**/*Test.class"`、
そしてexcludesには`"**/Abstract*.class"`が使われます。

23.13.6. Test grouping

JUnit and TestNG allows sophisticated groupings of test methods.

For grouping JUnit test classes and methods JUnit 4.8 introduces the concept of categories. [16]
The `test` task allows the specification of the JUnit categories you want to include and exclude.

例23.12 JUnit Categories

build.gradle

```
test {
    useJUnit {
        includeCategories 'org.gradle.junit.CategoryA'
        excludeCategories 'org.gradle.junit.CategoryB'
    }
}
```

The TestNG framework has a quite similar concept. In TestNG you can specify different test groups. [17] The test groups that should be included or excluded from the test execution can be configured in the test task.

例23.13 Grouping TestNG tests

build.gradle

```
test {
    useTestNG {
        excludeGroups 'integrationTests'
        includeGroups 'unitTests'
    }
}
```

23.13.7. Test reporting

The `Test` task generates the following results by default.

- An HTML test report.
- The results in an XML format that is compatible with the Ant JUnit report task. This format is supported by many other tools, such as CI servers.
- Results in an efficient binary format. The task generates the other results from these binary results.

There is also a stand-alone `TestReport` task type which can generate the HTML test report from the binary results generated by one or more `Test` task instances. To use this task type, you need to define a `destinationDir` and the test results to include in the report. Here is a sample which generates a combined report for the unit tests from subprojects:

例23.14 Creating a unit test report for subprojects

build.gradle

```
subprojects {
    apply plugin: 'java'

    // Disable the test report for the individual test task
    test {
        reports.html.enabled = false
    }
}

task testReport(type: TestReport) {
    destinationDir = file("$buildDir/reports/allTests")
    // Include the results from the `test` task in all subprojects
    reportOn subprojects*.test
}
```

You should note that the `TestReport` type combines the results from multiple test tasks and needs to aggregate the results of individual test classes. This means that if a given test class is executed by multiple test tasks, then the test report will include executions of that class, but it can be hard to distinguish individual executions of that class and their output.

23.13.7.1. TestNG parameterized methods and reporting

TestNG supports parameterizing test methods, allowing a particular test method to be executed multiple times with different inputs. Gradle includes the parameter values in its reporting of the test method execution.

Given a parameterized test method named `aTestMethod` that takes two parameters, it will be reported with the name: `aTestMethod(toStringValueOfParam1, toStringValueOfParam2)`. This makes identifying the parameter values for a particular iteration easy.

23.13.8. 規約値

表23.14 Javaプラグイン - テストプロパティ

タスクプロパティ	型	デフォルト値
`testClassesDir`	`File`	`sourceSets.test.output.classesDir`
`classpath`	`FileCollection`	`sourceSets.test.runtimeClasspath`
`testResultsDir`	`File`	`testResultsDir`
`testReportDir`	`File`	`testReportDir`
`testSrcDirs`	`List<File>`	`sourceSets.test.java.srcDirs`

23.14. Jar

j　　a　　r
タスクは、プロジェクトのクラスファイルとリソースを含んだJARファイルを生成します。このJARファ
a　r　c　h　i　v　e　s
という依存関係のコンフィギュレーションで、アーティファクトとして宣言されています。
これは、（このプロジェクトに）依存しているプロジェクトのクラスパスでこのJARが利用可能というこ
プロジェクトをリポジトリにアップロードする場合、このJARは依存関係のディスクリプタの一部として
アーカイブの扱いについては「アーカイブを作成する　」、アーティファクトの設定については52章
アーティファクトの公開 で、さらに詳しく学ぶことができます。

23.14.1. Manifest

各jarやwarオブジェクトは、それぞれManifestのインスタンスであるmanifest
プロパティを持ちます。　　　　　　　アーカイブが生成されるとき、対応するMANIFEST.MF
ファイルがアーカイブ内に作成されます。

例23.15 MANIFEST.MFのカスタマイズ

build.gradle

```
jar {
    manifest {
        attributes("Implementation-Title": "Gradle",
                   "Implementation-Version": version)
    }
}
```

独立したManifestインスタンスを生成することもできます。
これは例えば、複数のjar間でマニフェスト情報を共有する目的などに利用できます。

例23.16 manifestオブジェクトの作成

build.gradle

```
ext.sharedManifest = manifest {
    attributes("Implementation-Title": "Gradle",
               "Implementation-Version": version)
}
task fooJar(type: Jar) {
    manifest = project.manifest {
        from sharedManifest
    }
}
```

Manifestオブジェクトに他のマニフェストをマージすることもできます。
マージ対象のマニフェストは、ファイルパス、または上の例のように別のManifest
オブジェクトの参照によって指定できます。

例23.17 特定のアーカイブ用にMANIFEST.MFを分離

build.gradle

```
task barJar(type: Jar) {
    manifest {
        attributes key1: 'value1'
        from sharedManifest, 'src/config/basemanifest.txt'
        from('src/config/javabasemanifest.txt',
             'src/config/libbasemanifest.txt') {
            eachEntry { details ->
                if (details.baseValue != details.mergeValue) {
                    details.value = baseValue
                }
                if (details.key == 'foo') {
                    details.exclude()
                }
            }
        }
    }
}
```

マニフェストは`from`文で宣言された順にマージされます。
元のマニフェストとマージするマニフェストの双方に同じキーを持つ値が定義されている場合、デフォル
マージの振る舞いは、マージで生成されるマニフェストの各エントリに対する
`ManifestMergeDetails`インスタンスにアクセスする、`eachEntry`
アクションを追加することによって自在にカスタマイズ可能です。
マージはfrom文によって即座に実行されるわけではありません。 jarの生成時、または`writeTo`や `effe`
の呼び出し時に、遅延して実行されます。

マニフェストをディスクに書き出すのは簡単です。

例23.18 特定のアーカイブ用にMANIFEST.MFを分離

build.gradle

```
jar.manifest.writeTo("$buildDir/mymanifest.mf")
```

23.15. アップロード

作成したアーカイブをアップロードする方法は、52章アーティファクトの公開 で説明しています。

---

[16] The JUnit wiki contains a detailed description on how to work with JUnit categories: https://github.com/junit-team/junit/wiki/Categories.

[17] The TestNG documentation contains more details about test groups: http://testng.org/doc/documentation-main.html#test-groups.

24
Groovyプラグイン

Groovyプラグインは、Javaプラグインを拡張し、Groovyプロジェクトのサポートを追加したものです。
Groovyコードと Groovy/Javaの混合コードをサポートするほか、Javaコードのみのプロジェクトを取り扱
また、JavaとGroovyのジョイントコンパイルをサポートしており、GroovyコードとJavaコードを自由に
例えば、GroovyのクラスはJavaのクラスを継承できますし、更にそのJavaクラスがGroovyクラスを継承

24.1. 使用方法

Groovyプラグインを使うためには、ビルドスクリプトに下記を含めます：

例24.1 Groovyプラグインの使用

build.gradle

```
apply plugin: 'groovy'
```

24.2. タスク

Groovyプラグインは、以下のタスクをプロジェクトに追加します。

表24.1 Groovyプラグイン - タスク

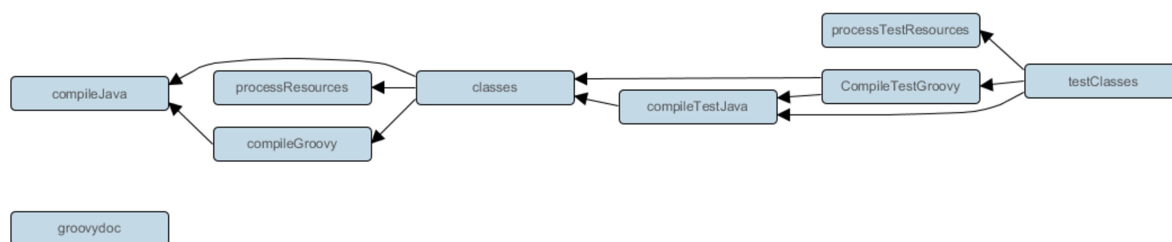
タスク名	依存先	型	説明
compileGroovy	compileJava	GroovyCompile	製品のGroovyソースファイルを
compileTestGroovy	compileTestJava	GroovyCompile	テストのGroovyソースファイル
compile*SourceSet*Groovy	compile*SourceSet*Java	GroovyCompile	特定のソースセットのGroovyソ
groovydoc	-	Groovydoc	製品のGroovyソースファイルの

Groovyプラグインは、Javaプラグインによって追加されたタスクに以下の依存関係を追加します。

表24.2 Groovyプラグイン - タスクの追加依存関係

タスク名	依存先
classes	compileGroovy
testClasses	compileTestGroovy
*sourceSet*Classes	compile*SourceSet*Groovy

図24.1 Groovyプラグイン - タスク



24.3. プロジェクトレイアウト

Groovyプラグインでは、表24.3「Groovyプラグイン - プロジェクトレイアウト」に示したようなプロジェクトレイアウトを想定しています。すべてのGroovyソースディレクトリは、Gro
お よ び
Javaコードを含むことができます。Javaソースディレクトリは、Javaソースコードだけしか含むことがで
[1 9]
これらのどのディレクトリも、存在しなかったり、何も含んでいなくてもかまいません。Groovyプラグ

表24.3 Groovyプラグイン - プロジェクトレイアウト

ディレクトリ	意味
`src/main/java`	
`src/main/resources`	
`src/main/groovy`	製品のGroovyソース。ジョイントコンパイルするJavaソースを含んでもよ
`src/test/java`	
`src/test/resources`	
`src/test/groovy`	テストのGroovyソース。ジョイントコンパイルするJavaソースを含んでも
src/*sourceSet*/java	
src/*sourceSet*/resources	
src/*sourceSet*/groovy	特定のソースセットのGroovyソース。ジョイントコンパイルするJavaソー

24.3.1. プロジェクトレイアウトの変更

Javaプラグインと同じように、Groovyプラグインも製品コードやテストコードの場所を変更することがで

例24.2 Groovyソースレイアウトのカスタマイズ

build.gradle

```
sourceSets {
    main {
        groovy {
            srcDirs = ['src/groovy']
        }
    }

    test {
        groovy {
            srcDirs = ['test/groovy']
        }
    }
}
```

24.4. 依存関係の管理

Gradleのビルド言語はGroovyをベースにしており、Gradleの一部もGroovyで実装されているので、Gra
ここで宣言したGroovyへの依存関係は、コンパイルクラスパスおよび実行時クラスパスに追加されます。

Groovyが製品コードで使われている場合は、Groovy依存関係は`compile`
コンフィギュレーションに追加してください。

例24.3 Groovyプラグインの設定

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.6'
}
```

Groovyがテストコードのみで使われている場合は、Groovy依存関係は`testCompile`
コンフィギュレーションに追加してください。

例24.4 Groovyテスト用の依存関係設定

build.gradle

```
dependencies {
    testCompile "org.codehaus.groovy:groovy:2.3.6"
}
```

Gradleに同梱されているGroovyを使うには、`localGroovy()`
を依存関係で宣言してください。ただ注意してほしいのは、Gradleのバージョンが異なれば同梱されてい
`localGroovy()`は、普通の宣言方法に比べると安全とは言えません。

例24.5 同梱のGroovyを使用する依存関係設定

build.gradle

```
dependencies {
    compile localGroovy()
}
```

Groovyライブラリは、必ずしもリモートのリポジトリから取得しなければならないわけではありません。l i b
ディレクトリなどに格納して、ソースコード管理システムにチェックインしたりすることもできます。

例24.6 Groovyをファイル依存関係で設定する

build.gradle

```
repositories {
    flatDir { dirs 'lib' }
}

dependencies {
    compile module('org.codehaus.groovy:groovy:1.6.0') {
        dependency('asm:asm-all:2.2.3')
        dependency('antlr:antlr:2.7.7')
        dependency('commons-cli:commons-cli:1.2')
        module('org.apache.ant:ant:1.9.3') {
            dependencies('org.apache.ant:ant-junit:1.9.3@jar',
                         'org.apache.ant:ant-launcher:1.9.3')
        }
    }
}
```

The “module” reference may be new to you. See 51章依存関係の管理 for more information about this and other information about dependency management.

24.5. Groovyクラスパスの自動設定

GroovyCompileタスクとGroovydocタスクは二つの用途でGroovyを使用します。タスクのclasspath と g r o o v y C l a s s p a t h
です。前者はソースコードから参照されているクラスを配置するパスで、Groovyライブラリも典型的に

groovyClasspathが明示的に設定されていない場合、Groovy(base)プラグインはタスクのclasspath から以下のように設定値を推論しようとします。

- groovy-all(-indy)のJarがclasspathに見つかれば、そのjarをgroovyClasspath にも追加します。
- groovy(-indy)のJarがclasspath に見つかれば、そして少なくとも一つのリポジトリがプロジェクトに設定されていれば、リポジトリ groovy(-indy)の依存関係を取得してgroovyClasspathに追加します。
- それ以外の場合は、groovyClasspath の推論に失敗した旨のメッセージを出力してタスクは失敗します。

Note that the “-indy” variation of each jar refers to the version with invokedynamic support.

24.6. 規約プロパティ

Groovyプラグインは、プロジェクトには規約プロパティを追加しません。

24.7. ソースセットプロパティ

Groovyプラグインは、プロジェクトの各ソースセットに以下の規約プロパティを追加します。これらの
「規約 」参照)。

表24.4 Groovyプラグイン - ソースセットプロパティ

プロパティ名	型	デフォルト値	説明
groovy	SourceDirectorySet (読取り専用)	非null	このソースセットのG ファイルを含み、その
groovy.srcDirs	Set<File>。「入力ファイルセットを指定する 」で説明されたものなら何でも設定可能	[*projectDir*/src/*name*/groovy]	このソースセットのG
allGroovy	FileTree (読取り専用)	非null	このソースセットの全

上記プロパティはGroovySourceSet型の規約オブジェクトにより提供されます。

また、Groovyプラグインは、いくつかのソースセットプロパティを修正します：

表24.5 Groovyプラグイン - ソースセットプロパティ

プロパティ名	変更点
allJava	Groovyソースディレクトリにあるすべての.javaファイルを追加
allSource	Groovyソースディレクトリにあるすべてのソースファイルを追加

24.8. GroovyCompile

Groovyプラグインは、プロジェクトの各ソースセットにGroovyCompile
タスクを追加します。このタスクはJavaCompileタスクを拡張したものです(「CompileJava」
参照)。groovyOptions.useAnt
がtrueになっていなければ、GradleネイティブのGroovyコンパイラ統合が使用されます。Antベースのコ
このGroovyCompile
タスクは、公式Groovyコンパイラのオプションをほとんど全てサポートしています。

表24.6 Groovyプラグイン - GroovyCompileプロパティ

タスクプロパティ	型	デフォルト値
`classpath`	`FileCollection`	*`sourceSet`*`.compileClasspath`
`source`	`FileTree`。「入力ファイルセットを指定する 」で説明されたものなら何でも設定可能	*`sourceSet`*`.groovy`
`destinationDir`	`File`.	*`sourceSet`*`.output.classesDir`
`groovyClasspath`	`FileCollection`	`groovy` configuration if non-empty; Groovy library found on `classpath` otherwise

---

[19] Russel WinderのGant (http://gant.codehaus.org)が導入したのと同じ規約を使っています。

25 Scalaプラグイン

Scalaプラグインは、Javaプラグインを拡張し、Scalaプロジェクトのサポートを追加したものです。
ScalaコードとScala/Javaの混合コードをサポートするほか、Javaコードのみのプロジェクトを取り扱う
また、JavaとScalaのジョイントコンパイルをサポートしており、ScalaコードとJavaコードを自由に混せ
例えば、ScalaのクラスはJavaのクラスを継承できますし、更にそのJavaクラスがScalaクラスを継承して

25.1. 使用方法

Scalaプラグインを使うには、ビルドスクリプトに以下の行を追加します。

例25.1 Scalaプラグインを使う

build.gradle

```
apply plugin: 'scala'
```

25.2. タスク

Scalaプラグインは、プロジェクトに以下のタスクを追加します。

表25.1 Scalaプラグイン - タスク

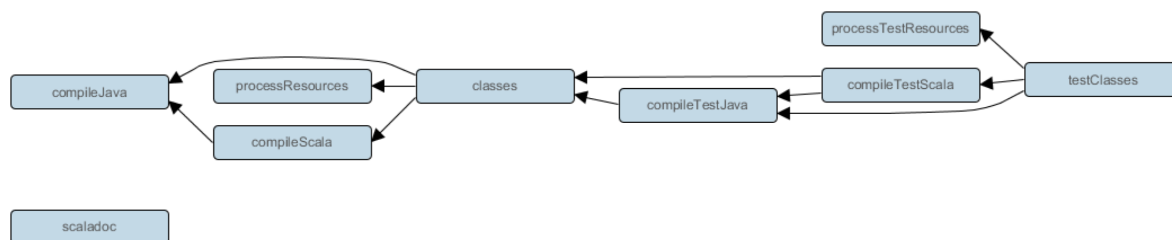
タスク	依存先	型	説明
compileScala	compileJava	ScalaCompile	Scalaの製品ソースファイルをコン
compileTestScala	compileTestJava	ScalaCompile	Scalaのテストソースファイルをコ
compile*SourceSet*Scala	compile*SourceSet*Java	ScalaCompile	指定したソースセットのScalaソー
scaladoc	-	ScalaDoc	Scala製品ソースファイルのAPIド

Scalaプラグインは、Javaプラグインによって追加されたタスクに以下の依存関係を追加します。

表25.2 Scalaプラグイン - タスクの追加依存関係

タスク名	依存先
classes	compileScala
testClasses	compileTestScala
*sourceSet*Classes	compile*SourceSet*Scala

図25.1 Scalaプラグイン - タスク



25.3. プロジェクトレイアウト

Scalaプラグインは次のようなプロジェクトレイアウトを想定しています。
すべてのScalaソースディレクトリにはScalaだけでなく
Javaコードも含めることができますが、JavaソースディレクトリにはJavaコードしか含めることができま
また、これらのどのディレクトリも、存在しなかったり、何も含んでいなくてもかまいません。Scalaプ

表25.3 Scalaプラグイン

ディレクトリ	意味
src/main/java	
src/main/resources	
src/main/scala	製品のScalaソース。ジョイントコンパイルするJavaソースを含んでもよい
src/test/java	
src/test/resources	
src/test/scala	テストのScalaソース。ジョイントコンパイルするJavaソースを含んでもよ
src/*sourceSet*/java	
src/*sourceSet*/resources	
src/*sourceSet*/scala	特定のソースセットのScalaソース。ジョイントコンパイルするJavaソース

25.3.1. プロジェクトレイアウトの変更

Javaプラグインと同じように、Scalaプラグインも製品コードやテストコードの場所を変更することがで

例25.2 Scalaソースレイアウトのカスタマイズ

build.gradle

```
sourceSets {
    main {
        scala {
            srcDirs = ['src/scala']
        }
    }
    test {
        scala {
            srcDirs = ['test/scala']
        }
    }
}
```

25.4. 依存関係の管理

Scalaプロジェクトでは、`scala-library`への依存関係を宣言する必要があります。
この依存関係は、Scalaコードをコンパイルしたり実行したりする際のクラスパスに追加されるほか、Sc
[20]

Scalaが製品コードで使われている場合は、`scala-library`への依存関係は`compile`
コンフィグレーションに追加してください。

例25.3 製品コードに使うScalaへの依存関係の宣言

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.scala-lang:scala-library:2.11.1'
}
```

Scalaがテストコードのみで使われている場合は、`scala-library`への依存関係は`testCompile`
コンフィグレーションに追加してください。

例25.4 テストコードに使うScalaへの依存関係の宣言

build.gradle

```
dependencies {
    testCompile "org.scala-lang:scala-library:2.11.1"
}
```

25.5. scalaClasspathの自動設定

ScalaCompileタスクとScalaDocタスクは二つの用途でScalaを使用します。タスクのclasspath と s c a l a C l a s s p a t h です。前者はソースコードから参照されているクラスを配置するパスで、scala-libraryも典型的には他の

scalaClasspathが明示的に設定されていない場合、Scala(base)プラグインはタスクのclasspath から以下のように設定値を推論しようとします。

- scala-libraryのJarがclasspath に見つかれば、そして少なくとも一つのリポジトリがプロジェクトに設定されていれば、リポジトリ scala-compilerの依存関係を取得してscalaClasspathに追加します。
- それ以外の場合は、scalaClasspath の推論に失敗した旨のメッセージを出力してタスクは失敗します。

25.6. 規約プロパティ

Scalaプラグインは、プロジェクトに規約プロパティを追加しません。

25.7. ソースセットプロパティ

Scalaプロジェクトの各ソースセットに以下の規約プロパティを追加します。これらのプロパティは、あ 「規約 」参照)

表25.4 Scalaプラグイン - ソースセットプロパティ

プロパティ名	型	デフォルト値	説明
scala	SourceDirectorySet (読取り専用)	非null	このソースセットのSc ファイルを含み、その
scala.srcDirs	「入力ファイルセットを指定する 」で説明されたものなら何でも設定可能	[*projectDir*/src/*name*/scala]	このソースセットのSc
allScala	FileTree (読取り専用)	非null	このソースセットの全

これらの規約プロパティはScalaSourceSet型の規約オブジェクトにより提供されます。

また、Scalaプラグインはいくつかのソースセットプロパティを修正します。

表25.5 Scalaプラグイン

プロパティ名	変更点
allJava	Scalaソースディレクトリにあるすべての.javaファイルを追加
allSource	Scalaソースディレクトリにあるすべてのファイルを追加

25.8. Fast Scala Compiler

Scalaプラグインには、fsc、Fast Scala Compilerのサポートが含まれています。

例25.5 Fast Scala Compilerを有効にする

build.gradle

```
compileScala {
    scalaCompileOptions.useCompileDaemon = true

    // optionally specify host and port of the daemon:
    scalaCompileOptions.daemonServer = "localhost:4243"
}
```

ただし、次の点に注意してください。`fsc`は、コンパイル時のクラスパスに含まれるもの（訳注：jarなど）が変更される度に再起動させる必要があります。（コンパイル時クラスパス自体の変
このため、マルチプロジェクトとの相性は悪くなります。

25.9. 別プロセスでコンパイルする

`scalaCompileOptions.fork`
オプションがtrueにセットされていると、コンパイル処理は外部プロセスで実行されます。
フォーク処理の詳細はどのコンパイラを使用するかにより異なり、Antベースのコンパイラを使用する場
`scalaCompileOptions.useAnt = true`）、`ScalaCompile`
タスクが実行される度に新しいプロセスがフォークされます。また、デフォルトではフォークされません
Zincベースのコンパイラを使う場合（`scalaCompileOptions.useAnt = false`
）、Gradleのコンパイラデーモンを使用してコンパイルされます。Antベースのコンパイラと異なり、こ

外部プロセスのメモリ設定は、デフォルトではJVMのデフォルト値が使用されます。メモリ設定を調整す
`scalaCompileOptions.forkOptions`を必要に応じて設定してください。

例25.6 メモリ設定の調整

build.gradle

```
tasks.withType(ScalaCompile) {
    configure(scalaCompileOptions.forkOptions) {
        memoryMaximumSize = '1g'
        jvmArgs = ['-XX:MaxPermSize=512m']
    }
}
```

25.10. インクリメンタルコンパイル

前回コンパイルしたときからの変更差分、およびそれにより影響を受けるクラスのみを再コンパイルする
これは、コードを小さく変更して頻繁にコンパイルするようなとき、開発中にはしばしばそういうことが

Scalaプラグインは、Zincという、sbt
のインクリメンタルScalaコンパイラのスタンドアローン版を統合することでインクリメンタルコンパイ

例25.7 Zincベースのコンパイラを有効にする

build.gradle

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.useAnt = false
}
```

APIドキュメント
に記載されている部分を除き、ZincベースのコンパイラはAntベースのコンパイラと全く同じ設定オプシ
ただ、ZincコンパイラはJava6以上のバージョンを必要とするので注意してください。これは、Gradle自

Scalaプラグインは、zincという名前のコンフィグレーションを追加し、Zincライブラリとその依存関係
GradleがデフォルトでZincのバージョンをオーバーライドするには、Zincへの依存関係（例えば
`zinc "com.typesafe.zinc:zinc:0.1.4"`）を明示的に指定してください。
どのバージョンのZincが使用されているかにかかわらず、Zincは常にscalaToolsコンフィグレーションに

Antベースのコンパイラ同様、ZincベースのコンパイラもJavaとScalaのジョイントコンパイルをサポート
デフォルトでは、`src/main/scala`
にある全てのJavaとScalaのコードがジョイントコンパイルに使用されます。Zincベースのコンパイラは

インクリメンタルコンパイルを行うには、ソースコードを解析する必要があります。解析結果は、`scala`
（適切なデフォルト値が設定されています）で指定されたファイルに保存されます。
マルチプロジェクトの場合、この解析ファイルは下流の`ScalaCompile`
タスクに渡されるので、プロジェクトをまたいでのインクリメンタルコンパイルが可能です。
Scalaプラグインにより追加された`ScalaCompile`
タスクはデフォルトでそのように動作するので、追加の設定は必要ありません。 その他の`ScalaCompil`
タスクでこの動作を有効にするには、`scalaCompileOptions.incrementalOptions.publishedCo`
が適切なクラスフォルダやJarアーカイブを指すように設定し、下流のタスクのコンパイルクラスパスに
`publishedCode`
が正しく設定されていない場合、上流のソースが変更されても下流のタスクの再コンパイルが適切に動か

解析処理に伴うオーバーヘッドにより、クリーンコンパイルや大規模な変更を行った後のコンパイルは、

ZincのNailgunベースのデーモンモードには対応していません。代わりに、Gradle自身のコンパイラデー

25.11. Eclipse Integration

When the Eclipse plugin encounters a Scala project, it adds additional configuration to make the project work with Scala IDE out of the box. Specifically, the plugin adds a Scala nature and dependency container.

25.12. IntelliJ IDEA Integration

When the IDEA plugin encounters a Scala project, it adds additional configuration to make the project work with IDEA out of the box. Specifically, the plugin adds a Scala facet and a Scala compiler library that matches the Scala version on the project's class path.

26
War プラグイン

War プラグインは WEB アプリケーションの WAR ファイルを生成できるように Java プラグインを拡張したものです。 Java プラグインの 標準の JAR ファイルは生成されなくなり、WAR アーカイブ タスクが追加されます。

26.1. 使用方法

War プラグインを使うためには、ビルドスクリプトに下記を含めます：

例26.1 Using the War plugin

build.gradle

```
apply plugin: 'war'
```

26.2. タスク

War プラグインは 次のタスクをプロジェクトに追加します。

表26.1 War プラグイン - タスク

タスク名	依存先	型	説明
war	compile	War	WAR ファイルを生成します。

War プラグインは Java プラグインで追加されたタスクに対し、次の依存関係を追加します。

表26.2 War プラグイン - 追加されたタスクの依存関係

タスク名	依存先
assemble	war

図26.1 War プラグイン - タスク

26.3. プロジェクトレイアウト

表26.3 War プラグイン - プロジェクトレイアウト

ディレクトリ	意味
`src/main/webapp`	WEB アプリケーションソース

26.4. 依存関係の管理

War プラグインは `providedCompile` と `providedRuntime` の２つの依存構成を追加します。これらの構成は WAR アーカイブには追加されないという点を除けば、それぞれ `compile`、`runtime` と同じスコープを持ちます。 `provided` 構成が推移的に機能することは特筆すべき点です。そうですね。例えば provided 構成の何れかに `commons-httpclient:commons-httpclient:3.0` を追加したとします。この依存ライブラリは `commons-codec` に依存します。 これは、たとえ、`commo` が `compile` 構成に依存していることが明らかであったとしても WAR ファイルには `httpclient` も `commons-codec` も追加されないことを意味します。 もし、この推移的な挙動を望まないのであれば、単純に `commons-httpclient:commons-httpclier` を dependencies の `provided` 構成に定義してください。

26.5. 規約プロパティ

表26.4 War プラグイン - ディレクトリプロパティ

プロパティ名	型	デフォルト値	説明
`webAppDirName`	`String`	`src/main/webapp`	WEB アプリケーションソースディレクトリの名前
`webAppDir`	`File` (読取専用)	*`projectDir/webAppDirName`*	WEB アプリケーションソースディレクトリ

これらのプロパティは `WarPluginConvention` オブジェクトによって提供されています。

26.6. War

The `War` class in the API documentation has additional useful information.

26.7. カスタマイズ

以下は最も重要なカスタマイズ例です。

例26.2 Customization of war plugin

build.gradle

```
configurations {
   moreLibs
}

repositories {
   flatDir { dirs "lib" }
   mavenCentral()
}

dependencies {
    compile module(":compile:1.0") {
        dependency ":compile-transitive-1.0@jar"
        dependency ":providedCompile-transitive:1.0@jar"
    }
    providedCompile "javax.servlet:servlet-api:2.5"
    providedCompile module(":providedCompile:1.0") {
        dependency ":providedCompile-transitive:1.0@jar"
    }
    runtime ":runtime:1.0"
    providedRuntime ":providedRuntime:1.0@jar"
    testCompile "junit:junit:4.11"
    moreLibs ":otherLib:1.0"
}

war {
    from 'src/rootContent' // adds a file-set to the root of the archive
    webInf { from 'src/additionalWebInf' } // adds a file-set to the WEB-INF dir.
    classpath fileTree('additionalLibs') // adds a file-set to the WEB-INF/lib dir.
    classpath configurations.moreLibs // adds a configuration to the WEB-INF/lib dir.
    webXml = file('src/someWeb.xml') // copies a file to WEB-INF/web.xml
}
```

もちろん、excludes と includes を定義したクロージャを使って異なるファイルセットを設定することもできます。

27
Earプラグイン

EarプラグインはWebアプリケーションのEARファイルアセンブリサポートを追加します。
Javaプラグインは必須ではありませんが、Javaプラグインを利用するプロジェクトに対してはデフォルト

27.1. 使用方法

Earプラグインを使うには、ビルドスクリプトに以下を含めてください:

例27.1 Earプラグインの利用

build.gradle

```
apply plugin: 'ear'
```

27.2. タスク

Earプラグインはプロジェクトに以下のタスクを追加します。

表27.1 Earプラグイン - タスク

タスク名	依存先	型	説明
ear	compile (Javaプラグインが同時に適用されている場合のみ)	Ear	アプリケーションのEARファ

Earプラグインはベースプラグインが追加したタスクに以下の依存関係を追加します。

表27.2 Earプラグイン - タスク依存関係の追加

タスク名	依存先
assemble	ear

27.3. プロジェクトレイアウト

表27.3 Earプラグイン - プロジェクトレイアウト

ディレクトリ	意味
src/main/application	META-INFディレクトリなどのEarリソース

27.4. 依存関係の管理

Earプラグインは2つの依存関係コンフィグレーションを追加します: deployおよびearlib deployコンフィグレーションにおけるすべての依存関係はEARアーカイブのルートに配置され、それらは推移的ではありません。 earlib コンフィグレーションにおけるすべての依存関係はEARアーカイブの'lib'ディレクトリに配置され、推移的です。

27.5. 規約プロパティ

表27.4 Earプラグイン - ディレクトリプロパティ

プロパティ名	型	デフォルト値
appDirName	String	src/main/application
libDirName	String	lib
deploymentDescriptor	org.gradle.plugins.ear.descriptor.DeploymentDescriptor	デプロイメントディスクリプター、デフォ

これらのプロパティは規約オブジェクトEarPluginConventionが提供します。

27.6. Ear

Earタスクのデフォルトのふるまいは、src/main/application の内容をアーカイブのルートにコピーすることです。 application ディレクトリがデプロイメントディスクリプタMETA-INF/application.xml を含まない場合は、生成してくれます。

Earも参照してみてください。

27.7. カスタマイズ

最も重要なカスタマイズオプションを使う例はこちらです:

例27.2 Earプラグインのカスタマイズ

build.gradle

```
apply plugin: 'ear'
apply plugin: 'java'

repositories { mavenCentral() }

dependencies {
    // The following dependencies will be the ear modules and
    // will be placed in the ear root
    deploy project(':war')

    // The following dependencies will become ear libs and will
    // be placed in a dir configured via the libDirName property
    earlib group: 'log4j', name: 'log4j', version: '1.2.15', ext: 'jar'
}

ear {
    appDirName 'src/main/app'  // use application metadata found in this folder
    // put dependent libraries into APP-INF/lib inside the generated EAR
    libDirName 'APP-INF/lib'
    deploymentDescriptor {  // custom entries for application.xml:
//      fileName = "application.xml"  // same as the default value
//      version = "6"  // same as the default value
        applicationName = "customear"
        initializeInOrder = true
        displayName = "Custom Ear"  // defaults to project.name
        // defaults to project.description if not set
        description = "My customized EAR for the Gradle documentation"
//      libraryDirectory = "APP-INF/lib"  // not needed, above libDirName setting doe
//      module("my.jar", "java")  // won't deploy as my.jar isn't deploy dependency
//      webModule("my.war", "/")  // won't deploy as my.war isn't deploy dependency
        securityRole "admin"
        securityRole "superadmin"
        withXml { provider -> // add a custom node to the XML
            provider.asNode().appendNode("data-source", "my/data/source")
        }
    }
}
```

`from`や`metaInf`といった、`Ear`タスクが提供するカスタマイズオプションも利用できます。

27.8. カスタムのディスクリプタファイルを使う

ear.deploymentDescriptorセクションで設定するかわりに、既にapplication.xmlがあって、それを使いたいとしましょう。 そのためにはMETA-INF/application.xmlをソースフォルダ内の適切な場所に配置します(appDirNameプロパティを参照)。
既存のファイルの内容が利用され、ear.deploymentDescriptorの設定内容は無視されます。

28
Jetty プラグイン

JettyプラグインはWarプラグインを拡張し、あなたのWEBアプリケーションをJettyコンテナにビルドし

28.1. 使用方法

Jettyプラグインを使うためには、ビルドスクリプトに下記を含めます：

例28.1 Using the Jetty plugin

build.gradle

```
apply plugin: 'jetty'
```

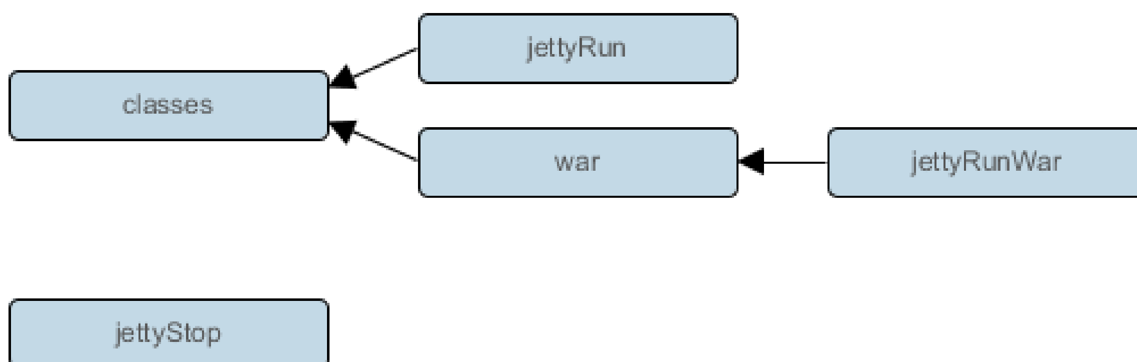
28.2. タスク

Jettyプラグインは、以下のタスクを定義します。

表28.1 Jettyプラグイン - タスク

タスク名	依存先	型	説明
jettyRun	compile	JettyRun	Jettyのインスタンスを開始し、webアプリケーショ
jettyRunWar	war	JettyRunWar	Jettyのインスタンスを開始し、WARファイルをデプ
jettyStop	-	JettyStop	Jettyのインスタンスを停止します

図28.1 Jettyプラグイン - タスク



28.3. プロジェクトレイアウト

Jettyプラグインは、Warプラグインと同じレイアウトを使用します。

28.4. 依存関係の管理

Jettyプラグインは、依存関係を定義しません。

28.5. 規約プロパティ

Jettyプラグインは、規約プロパティに従って定義されます。

表28.2 Jettyプラグイン - プロパティ

プロパティ名	型	デフォルト値	説明
`contextPath`	`String`	WAR file base name	The application deployment location within the Jetty container.
`httpPort`	`Integer`	`8080`	JettyのHTTPリクエストの待受けTCPポート
`stopPort`	`Integer`	`null`	Jettyの管理者リクエストの待受けTCPポート
`stopKey`	`String`	`null`	停止要求時にJettyに渡す文字列キー

上記プロパティは`JettyPluginConvention`型の規約オブジェクトにより提供されます。

29
Checkstyleプラグイン

Checkstyleプラグインは Checkstyle を使用して、あなたのプロジェクトのJavaソースファイルの品質チェックを行ないます。
そして、これらのチェック結果から結果レポートを生成します。

29.1. 使用方法

Checkstyleプラグインを使うためには、ビルドスクリプトに下記を含めます：

例29.1 Checkstyleプラグインの使用

build.gradle

```
apply plugin: 'checkstyle'
```

このプラグインは、品質チェックを行なうプロジェクトへの多くのタスクを加えます。
貴方はタスク実行時にチェック処理を実行させることが出来ます。 **gradle check**.

29.2. タスク

Checkstyleプラグインは、以下のタスクをプロジェクトに追加します:

表29.1 Checkstyleプラグイン - タスク

タスク名	依存先	型	説明
checkstyleMain	classes	Checkstyle	製品のJavaソースを対象にCheckstyleを実行
checkstyleTest	testClasses	Checkstyle	テストのJavaソースを対象にCheckstyleを実
checkstyle*SourceSet*	*sourceSet*Classes	Checkstyle	特定のソースセットで指定されたJavaソース

Checkstyleプラグインは、Javaプラグインによって追加されたタスクに以下の依存関係を追加します。

表29.2 Checkstyleプラグイン - タスクの追加依存関係

タスク名	依存先
check	全てのCheckstyleタスクは、checkstyleMain と checkstyleTestを含みます。

29.3. プロジェクトレイアウト

Checkstyleプラグインは、以下のようなプロジェクトレイアウトを想定しています:

表29.3 Checkstyle プラグイン - プロジェクトレイアウト

ファイル	意味
`config/checkstyle/checkstyle.xml`	Checkstyle の設定ファイル

29.4. 依存関係の管理

Checkstyleプラグインは、以下の依存関係設定を追加します。

表29.4 Checkstyleプラグイン - 依存関係設定

名前	意味
`checkstyle`	Checkstyleのライブラリを使用します。

29.5. 設定

`CheckstyleExtension`を参照。

30
CodeNarcプラグイン

CodeNarcプラグインは CodeNarc を使用して、あなたのプロジェクトのGroovyソースファイルの品質チェックを行ないます。
そして、これらのチェック結果から結果レポートを生成します。

30.1. 使用方法

CodeNarcプラグインを使うためには、ビルドスクリプトに下記を含めます：

例30.1 CodeNarcプラグインの使用

build.gradle

```
apply plugin: 'codenarc'
```

このプラグインは、品質チェックを行なうプロジェクトへの多くのタスクを加えます。
貴方はタスク実行時にチェック処理を実行させることが出来ます。 **gradle check**.

30.2. タスク

CodeNarcプラグインを使うためには、ビルドスクリプトに下記を含めます：

表30.1 CodeNarcプラグイン - タスク

タスク名	依存先	型	説明
codenarcMain	-	CodeNarc	製品のGroovyソースを対象にCodeNarcを実行します。
codenarcTest	-	CodeNarc	Runs CodeNarc against the test Groovy source files.
codenarc*SourceSet*		CodeNarc	特定のソースセットで指定されたGroovyソースを対象にC

CodeNarcプラグインは、Groovyプラグインによって追加されたタスクに以下の依存関係を追加します。

表30.2 CodeNarcプラグイン - タスクの追加依存関係

タスク名	依存先
check	全てのCodeNarcタスクは、codenarcMain と codenarcTestを含みます。

30.3. プロジェクトレイアウト

CodeNarcプラグインは、以下のようなプロジェクトレイアウトを想定しています:

表30.3 CodeNarc プラグイン - プロジェクトレイアウト

ファイル	意味
`config/codenarc/codenarc.xml`	CodeNarc の設定ファイル

30.4. 依存関係の管理

CodeNarcプラグインは、以下の依存関係設定を追加します。

表30.4 CodeNarcプラグイン - 依存関係設定

名前	意味
`codenarc`	CodeNarcのライブラリを使用します。

30.5. 設定

`CodeNarcExtension`を参照。

31
FindBugsプラグイン

FindBugsプラグインは FindBugs
を使用して、あなたのプロジェクトのJavaソースファイルの品質チェックを行ないます。
そして、これらのチェック結果から結果レポートを生成します。

31.1. 使用方法

FindBugsプラグインを使うためには、ビルドスクリプトに下記を含めます：

例31.1 FindBugsプラグインの使用

build.gradle

```
apply plugin: 'findbugs'
```

このプラグインは、品質チェックを行なうプロジェクトへの多くのタスクを加えます。
貴方はタスク実行時にチェック処理を実行させることが出来ます。 **gradle check**.

31.2. タスク

FindBugsプラグインは、以下のタスクをプロジェクトに追加します:

表31.1 FindBugsプラグイン - タスク

タスク名	依存先	型	説明
findbugsMain	classes	FindBugs	製品のJavaソースを対象にFindBugsを実行します
findbugsTest	testClasses	FindBugs	テストのJavaソースを対象にFindBugsを実行しま
findbugs*SourceSet*	*sourceSet*Classes	FindBugs	特定のソースセットで指定されたJavaソースを対

FindBugsプラグインは、Javaプラグインによって追加されたタスクに以下の依存関係を追加します。

表31.2 FindBugsプラグイン - タスクの追加依存関係

タスク名	依存先
check	全てのFindBugsタスクは、findbugsMain と findbugsTestを含みます。

31.3. 依存関係の管理

FindBugsプラグインは、以下の依存関係設定を追加します。

表31.3 FindBugsプラグイン - 依存関係設定

名前	意味
`findbugs`	FindBugsのライブラリを使用します。

31.4. 設定

`FindBugsExtension`を参照。

32 JDependプラグイン

JDependプラグインは JDepend を使用して、あなたのプロジェクトのJavaソースファイルの品質チェックを行ないます。
そして、これらのチェック結果から結果レポートを生成します。

32.1. 使用方法

JDependプラグインを使うためには、ビルドスクリプトに下記を含めます：

例32.1 JDependプラグインの使用

build.gradle

```
apply plugin: 'jdepend'
```

このプラグインは、品質チェックを行なうプロジェクトへの多くのタスクを加えます。
貴方はタスク実行時にチェック処理を実行させることが出来ます。 **gradle check**.

32.2. タスク

JDependプラグインは、以下のタスクをプロジェクトに追加します:

表32.1 JDependプラグイン - タスク

タスク名	依存先	型	説明
jdependMain	classes	JDepend	製品のJavaソースを対象にJDependを実行します。
jdependTest	testClasses	JDepend	テストのJavaソースを対象にJDependを実行します。
jdepend*SourceSet*	*sourceSet*Classes	JDepend	特定のソースセットで指定されたJavaソースを対象に

JDependプラグインは、Javaプラグインによって追加されたタスクに以下の依存関係を追加します。

表32.2 JDependプラグイン - タスクの追加依存関係

タスク名	依存先
check	全てのJDependタスクは、jdependMain と jdependTestを含みます。

32.3. 依存関係の管理

JDependプラグインは、以下の依存関係設定を追加します。

表32.3 JDependプラグイン - 依存関係設定

名前	意味
`jdepend`	JDependのライブラリを使用します。

32.4. 設定

`JDependExtension`を参照。

33
PMDプラグイン

ＰＭＤプラグインは ＰＭＤ
を使用して、あなたのプロジェクトのJavaソースファイルの品質チェックを行ないます。
そして、これらのチェック結果から結果レポートを生成します。

33.1. 使用方法

PMDプラグインを使うためには、ビルドスクリプトに下記を含めます：

例33.1 PMDプラグインの使用

build.gradle

```
apply plugin: 'pmd'
```

このプラグインは、品質チェックを行なうプロジェクトへの多くのタスクを加えます。
貴方はタスク実行時にチェック処理を実行させることが出来ます。 **gradle check**.

33.2. タスク

PMDプラグインは、以下のタスクをプロジェクトに追加します:

表33.1 PMDプラグイン - タスク

タスク名	依存先	型	説明
pmdMain	-	Pmd	製品のJavaソースを対象にPMDを実行します。
pmdTest	-	Pmd	テストのJavaソースを対象にPMDを実行します。
pmd*SourceSet*	-	Pmd	特定のソースセットで指定されたJavaソースを対象にPMDを実行しま

PMDプラグインは、Javaプラグインによって追加されたタスクに以下の依存関係を追加します。

表33.2 PMDプラグイン - タスクの追加依存関係

タスク名	依存先
check	全てのPMDタスクは、pmdMain と pmdTestを含みます。

33.3. 依存関係の管理

PMDプラグインは、以下の依存関係設定を追加します。

表33.3 PMDプラグイン - 依存関係設定

名前	意味
`pmd`	PMDのライブラリを使用します。

33.4. 設定

`PmdExtension`を参照。

34
The JaCoCo Plugin

> The JaCoCo plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The JaCoCo plugin provides code coverage metrics for Java code via integration with JaCoCo.

34.1. Getting Started

To get started, apply the JaCoCo plugin to the project you want to calculate code coverage for.

Example 34.1. Applying the JaCoCo plugin

build.gradle

```
apply plugin: "jacoco"
```

If the Java plugin is also applied to your project, a new task named `jacocoTestReport` is created that depends on the `test` task. The report is available at *`$buildDir`*`/reports/jacoco/te`
. By default, a HTML report is generated.

34.2. Configuring the JaCoCo Plugin

The JaCoCo plugin adds a project extension named `jacoco` of type `JacocoPluginExtension`, which allows configuring defaults for JaCoCo usage in your build.

Example 34.2. Configuring JaCoCo plugin settings

build.gradle

```
jacoco {
    toolVersion = "0.7.1.201405082137"
    reportsDir = file("$buildDir/customJacocoReportDir")
}
```

Table 34.1. Gradle defaults for JaCoCo properties

Property	Gradle default
reportsDir	“*$buildDir*/reports/jacoco”

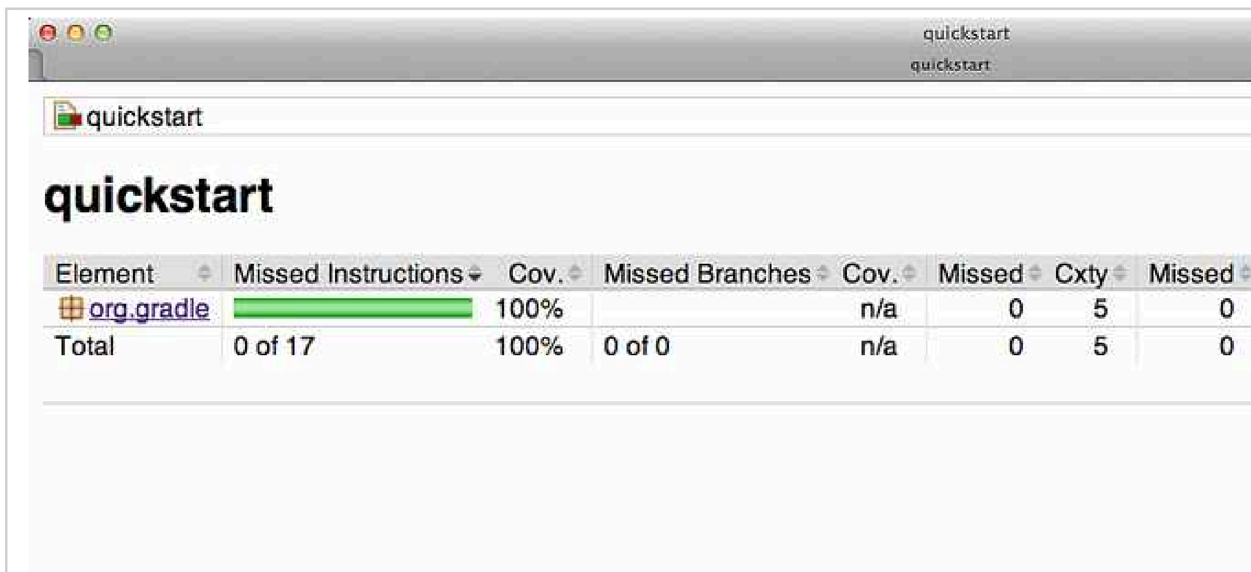
34.3. JaCoCo Report configuration

The `JacocoReport` task can be used to generate code coverage reports in different formats. It implements the standard Gradle type `Reporting` and exposes a report container of type `JacocoReportsContainer`.

Example 34.3. Configuring test task

build.gradle

```
jacocoTestReport {
    reports {
        xml.enabled false
        csv.enabled false
        html.destination "${buildDir}/jacocoHtml"
    }
}
```



34.4. JaCoCo specific task configuration

The JaCoCo plugin adds a `JacocoTaskExtension` extension to all tasks of type `Test`. This extension allows the configuration of the JaCoCo specific properties of the test task.

Example 34.4. Configuring test task

build.gradle

```
test {
    jacoco {
        append = false
        destinationFile = file("$buildDir/jacoco/jacocoTest.exec")
        classDumpFile = file("$buildDir/jacoco/classpathdumps")
    }
}
```

Table 34.2. Default values of the JaCoCo Task extension

Property	Gradle default
enabled	true
destPath	*$buildDir*/jacoco
append	true
includes	[]
excludes	[]
excludeClassLoaders	[]
sessionId	`auto-generated`
dumpOnExit	`true`
output	`Output.FILE`
address	-
port	-
classDumpPath	-
jmx	`false`

While all tasks of type `Test` are automatically enhanced to provide coverage information when the `java` plugin has been applied, any task that implements `JavaForkOptions` can be enhanced by the JaCoCo plugin. That is, any task that forks Java processes can be used to generate coverage information.

For example you can configure your build to generate code coverage using the `application` plugin.

Example 34.5. Using application plugin to generate code coverage data

build.gradle

```
apply plugin: "application"
apply plugin: "jacoco"

mainClassName = "org.gradle.MyMain"

jacoco {
    applyTo run
}

task applicationCodeCoverageReport(type:JacocoReport){
    executionData run
    sourceSets sourceSets.main
}
```

ノート：
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/testing/jacoco/application

Example 34.6. Coverage reports generated by applicationCodeCoverageReport

Build layout

```
application/
  build/
    jacoco/
      run.exec
    reports/jacoco/applicationCodeCoverageReport/html/
      index.html
```

34.5. Tasks

For projects that also apply the Java Plugin, The JaCoCo plugin automatically adds the following tasks:

Table 34.3. JaCoCo plugin - tasks

Task name	Depends on	Type	Description
`jacocoTestReport`	-	`JacocoReport`	Generates code coverage report for the test task.

34.6. Dependency management

The JaCoCo plugin adds the following dependency configurations:

Table 34.4. JaCoCo plugin - dependency configurations

Name	Meaning
`jacocoAnt`	The JaCoCo Ant library used for running the `JacocoReport` and `JacocoMerge` tasks.
`jacocoAgent`	The JaCoCo agent library used for instrumenting the code under test.

35
Sonarプラグイン

> このプラグインの代わりに、新しくできたSonnar Runnerを使いたくなる場面もあるかもしれません。
> 特に、Sonar3.4以上をサポートしているのはSonar Runnerプラグインだけです。

Sonarプラグインは、コード品質モニタリングのためのWebベースのプラットフォームであるSonarとの統合機能を提供します。 プラグインにより追加される`sonarAnalyze`タスクは、プラグインが適用されたプロジェクト、およびそのサブプロジェクトを解析することができま
このプラグインはベースにSonar Runnerが使われていて、Sonar 2.11以上を必要とします。

`sonarAnalyze`タスクは明示的に実行する必要のあるスタンドアロンタスクで、
他のいかなるタスクにも依存しません。
ソースコードだけでなく、タスクはクラスファイルと(可能であれば)テスト結果ファイルも解析します。
そのため、最大限の結果を得るために、解析の前にフルビルドを実行することが推奨されます。
典型的なセットアップでは、解析はビルドサーバー上で1日1回実行されるようにします。

35.1. 使用方法

必要最小限、Sonarプラグインがプロジェクトに適用されている必要があります。

例35.1 Sonarプラグインの適用

build.gradle

```
apply plugin: "sonar"
```

Sonarがデフォルト設定でローカル実行されるのでなければ、Sonarサーバーとデータベースへの接続設

例35.2 Sonar接続設定のコンフィグレーション

build.gradle

```
sonar {
    server {
        url = "http://my.server.com"
    }
    database {
        url = "jdbc:mysql://my.server.com/sonar"
        driverClassName = "com.mysql.jdbc.Driver"
        username = "Fred Flintstone"
        password = "very clever"
    }
}
```

代わりに、いくつかの、または全ての接続設定をコマンドラインで設定することもできます(
「コマンドラインでSonarの設定を行う 」参照)。
プロジェクト設定によって、プロジェクトをどのように解析するのかが決まります。
標準的なJavaプロジェクトに対してはデフォルト設定で事が足りますが、さまざまな方法でカスタマイス

例35.3 Sonarプロジェクト設定のコンフィグレーション

build.gradle

```
sonar {
    project {
        coberturaReportPath = file("$buildDir/cobertura.xml")
    }
}
```

サンプルの`sonar`、`server`、`database`、および`project`
ブロックは、それぞれコンフィグレーションオブジェクト `SonarRootModel`、 `SonarServer`、
`SonarDatabase`、 および`SonarProject`に対応します。
詳細はAPIドキュメントを参照してください。

35.2. マルチプロジェクトビルドの解析

Sonarプラグインはプロジェクト階層全体を一度に解析する機能をもちます。
集約されたメトリクスとサブプロジェクトへのドリルダウン機能を含む階層ビューをSonarのWebインタ
個別のプロジェクトごとに解析を行うよりも高速でもあります。

プロジェクト階層を解析するには、Sonarプラグインを階層の最上位のプロジェクトに適用する必要があ
これは、典型的には(必須ではないですが)ルートプロジェクトです。 そのプロジェクトの`sonar`
ブロックが `SonarRootModel`型のオブジェクトを構成します。
これはすべてのグローバルコンフィグレーションを保持しますが、
最も重要なのはサーバーとデータベース接続設定です。

例35.4 マルチプロジェクトビルドにおけるグローバルコンフィグレーション

build.gradle

```
apply plugin: "sonar"

sonar {
    server {
        url = "http://my.server.com"
    }
    database {
        url = "jdbc:mysql://my.server.com/sonar"
        driverClassName = "com.mysql.jdbc.Driver"
        username = "Fred Flintstone"
        password = "very clever"
    }
}
```

階層の各プロジェクトは独自のプロジェクトコンフィグレーションを持ちます。
共通の値は親のビルドスクリプトから設定可能です。

例35.5 マルチプロジェクトビルドにおける共通のプロジェクトコンフィグレーション

build.gradle

```
subprojects {
    sonar {
        project {
            sourceEncoding = "UTF-8"
        }
    }
}
```

サブプロジェクトの`sonar`ブロックは `SonarProjectModel`型のオブジェクトを構成します.

プロジェクトを個別に構成することもできます。 例えば、`skip`プロパティを`true`に設定することで、
プロジェクト(およびそのサブプロジェクト)を解析対象から除外することができます。
スキップされたプロジェクトはSonar Webインターフェースには表示されません。

例35.6 マルチプロジェクトビルドにおけるプロジェクト個別コンフィグレーション

build.gradle

```
project(":project1") {
    sonar {
        project {
            skip = true
        }
    }
}
```

プロジェクト単位のコンフィグレーションでもう一つ典型的なのは、解析対象のプログラム言語です。
Sonarはプロジェクトあたり一つの言語しか解析できないことに注意してください。

例35.7 解析対象の言語のコンフィグレーション

build.gradle

```
project(":project2") {
    sonar {
        project {
            language = "groovy"
        }
    }
}
```

一度に単一のプロパティしか設定しないときは、等価なプロパティの文法はより簡潔になります:

例35.8 プロパティ文法の利用

build.gradle

```
project(":project2").sonar.project.language = "groovy"
```

35.3. カスタムソースセットの解析

デフォルトでは、Sonarプラグインは`main`ソースセットのプロダクションソースと、`test`ソースセットのテストソースを解析します。
これはプロジェクトのソースディレクトリレイアウトとは独立して動作します。
ソースセットは必要に応じて追加できます。

例35.9 カスタムソースセットの解析

build.gradle

```
sonar.project {
    sourceDirs += sourceSets.custom.allSource.srcDirs
    testDirs += sourceSets.integTest.allSource.srcDirs
}
```

35.4. Java言語以外の解析

Java以外の言語で書かれたコードを解析するには、対応するSonarプラグインをインストールして、次のように`sonar.project.language`を設定してください。

例35.10 Java言語以外の解析

build.gradle

```
sonar.project {
    language = "grvy" // set language to Groovy
}
```

Sonarでは、バージョン3.4現在、一プロジェクトにつき一つの言語の解析しかできません。しかし、マ

35.5. カスタムSonarプロパティの設定

最 終 的 に は 、 コ ン フ ィ グ レ ー シ ョ ン の ほ と ん ど は 、
Sonarプロパティとして知られるkey-valueペアの形でSonarコードアナライザに渡されます。
プラグインのオブジェクトモデルのプロパティが対応するSonarプロパティにどのようにしてマップされ
APIドキュメントの`SonarProperty`アノテーションが示してます。
Sonarプラグインは、Sonarプロパティがコードアナライザに渡される前の後処理のためのフックを提供
このフックは、プラグインのオブジェクトモデルがカバーしていないプロパティを追加するためにも利用

グローバルSonarプロパティに対しては、`SonarRootModel`の`withGlobalProperties`
フックを利用してください:

例35.11 カスタムグローバルプロパティ設定

build.gradle

```
sonar.withGlobalProperties { props ->
    props["some.global.property"] = "some value"
    // non-String values are automatically converted to Strings
    props["other.global.property"] = ["foo", "bar", "baz"]
}
```

プロジェクト毎のSonarプロパティに対しては、`SonarProject`の`withProjectProperties`
フックを利用してください:

例35.12 カスタムプロジェクトプロパティ設定

build.gradle

```
sonar.project.withProjectProperties { props ->
    props["some.project.property"] = "some value"
    // non-String values are automatically converted to Strings
    props["other.project.property"] = ["foo", "bar", "baz"]
}
```

使用できるSonarプロパティはSonarのドキュメントで調べることができます。
ただし、これらのプロパティのほとんどはプラグインのオブジェクトモデルに対応するプロパティがあり
`withGlobalProperties`や`withProjectProperties`
フ ッ ク を 使 っ て 設 定 す る 必 要 は あ り ま せ ん 。
サードパーティのSonarプラグインに対しては、それらのドキュメントを参照してください。

35.6. コマンドラインでSonarの設定を行う

次 に 示 す プ ロ パ テ ィ は 、 `sonarAnalyze`
タスクのパラメーターとしてコマンドラインからでも設定できます。
タスクパラメーターは、ビルドスクリプト内の対応する値を全て上書きします。

- `server.url`
- `database.url`

- database.driverClassName
- database.username
- database.password
- showSql
- showSqlResults
- verbose
- forceAnalysis

以下に使用例を示します。

```
gradle sonarAnalyze --server.url=http://sonar.mycompany.com
--database.password=myPassword --verbose
```

他の値をコマンドラインから設定する必要がある場合は、システムプロパティが使用できます。

例35.13 カスタムのコマンドラインプロパティを実装する

build.gradle

```
sonar.project {
    language = System.getProperty("sonar.language", "java")
}
```

ただ、たいていの場合は設定値をビルドスクリプトに格納し、ソースコード管理システムで管理するのが

35.7. タスク

Sonarプラグインはプロジェクトに以下のタスクを追加します。

表35.1 Sonarプラグイン - タスク

タスク名	依存先	タイプ	説明
sonarAnalyze	-	SonarAnalyze	プロジェクト階層全体を解析し、結果をSonarデータ

36 The Sonar Runner Plugin

The Sonar Runner plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

It is intended that this plugin will replace the older Sonar Plugin in a future Gradle version.

The Sonar Runner plugin provides integration with Sonar, a web-based platform for monitoring code quality. It is based on the Sonar Runner, a Sonar client component that analyzes source code and build outputs, and stores all collected information in the Sonar database. Compared to using the standalone Sonar Runner, the Sonar Runner plugin offers the following benefits:

Automatic provisioning of Sonar Runner
: The ability to execute the Sonar Runner via a regular Gradle task makes it available anywhere Gradle is available (developer build, CI server, etc.), without the need to manually download, setup, and maintain a Sonar Runner installation.

Dynamic configuration from Gradle build scripts
: All of Gradle's scripting features can be leveraged to configure Sonar Runner as needed.

Extensive configuration defaults
: Gradle already has much of the information needed for Sonar Runner to successfully analyze a project. By preconfiguring the Sonar Runner based on that information, the need for manual configuration is reduced significantly.

36.1. Sonar Runner version and compatibility

The default version of the Sonar Runner used by the plugin is 2.3, which makes it compatible with Sonar 3.0 and higher. For compatibility with Sonar versions earlier than 3.0, you can configure the use of an earlier Sonar Runner version (see Section 36.4, “Specifying the Sonar Runner version”).

36.2. Getting started

To get started, apply the Sonar Runner plugin to the project to be analyzed.

Example 36.1. Applying the Sonar Runner plugin

build.gradle

```
apply plugin: "sonar-runner"
```

Assuming a local Sonar server with out-of-the-box settings is up and running, no further mandatory configuration is required. Execute **`gradle sonarRunner`** and wait until the build has completed, then open the web page indicated at the bottom of the Sonar Runner output. You should now be able to browse the analysis results.

Before executing the `sonarRunner` task, all tasks producing output to be analysed by Sonar need to be executed. Typically, these are compile tasks, test tasks, and code coverage tasks. To meet these needs, the plugins adds a task dependency from `sonarRunner` on `test` if the `java` plugin is applied. Further task dependencies can be added as needed.

36.3. Configuring the Sonar Runner

The Sonar Runner plugin adds a `SonarRunnerRootExtension` extension to the project and a `SonarRunnerExtension` extension to its subprojects, which allows you to configure the Sonar Runner via key/value pairs known as Sonar properties. A typical base line configuration includes connection settings for the Sonar server and database.

Example 36.2. Configuring Sonar connection settings

build.gradle

```
sonarRunner {
    sonarProperties {
        property "sonar.host.url", "http://my.server.com"
        property "sonar.jdbc.url", "jdbc:mysql://my.server.com/sonar"
        property "sonar.jdbc.driverClassName", "com.mysql.jdbc.Driver"
        property "sonar.jdbc.username", "Fred Flintstone"
        property "sonar.jdbc.password", "very clever"
    }
}
```

Alternatively, Sonar properties can be set from the command line. See 「コマンドラインでSonarの設定を行う 」 for more information.

For a complete list of standard Sonar properties, consult the Sonar documentation. If you happen to use additional Sonar plugins, consult their documentation.

In addition to set Sonar properties, the `SonarRunnerRootExtension` extension allows the configuration of the Sonar Runner version and the `JavaForkOptions` of the forked Sonar Runner process.

The Sonar Runner plugin leverages information contained in Gradle's object model to provide smart defaults for many of the standard Sonar properties. The defaults are summarized in the tables below. Notice that additional defaults are provided for projects that have the `java-base`

or `java` plugin applied. For some properties (notably server and database connection settings), determining a suitable default is left to the Sonar Runner.

Table 36.1. Gradle defaults for standard Sonar properties

Property	Gradle default
sonar.projectKey	"$project.group:$project.name" (for root project of analysed hierarchy; left to Sonar Runner otherwise)
sonar.projectName	project.name
sonar.projectDescription	project.description
sonar.projectVersion	project.version
sonar.projectBaseDir	project.projectDir
sonar.working.directory	"$project.buildDir/sonar"
sonar.dynamicAnalysis	"reuseReports"

Table 36.2. Additional defaults when **`java-base`** plugin is applied

Property	Gradle default
sonar.java.source	project.sourceCompatibility
sonar.java.target	project.targetCompatibility

Table 36.3. Additional defaults when **`java`** plugin is applied

Property	Gradle default
sonar.sources	sourceSets.main.allSource.srcDirs (filtered to only include existing directories)
sonar.tests	sourceSets.test.allSource.srcDirs (filtered to only include existing directories)
sonar.binaries	sourceSets.main.runtimeClasspath (filtered to only include directories)
sonar.libraries	sourceSets.main.runtimeClasspath (filtering to only include files; `rt.jar` added if necessary)
sonar.surefire.reportsPath	test.testResultsDir (if the directory exists)
sonar.junit.reportsPath	test.testResultsDir (if the directory exists)

Table 36.4. Additional defaults when **`jacoco`** plugin is applied

Property	Gradle default
sonar.jacoco.reportPath	jacoco.destinationFile

36.4. Specifying the Sonar Runner version

By default, version 2.3 of the Sonar Runner is used. To specify an alternative version, set the `SonarRunnerRootExtension.getToolVersion()` property of the `sonarRunner` extension of the project the plugin was applied to to the desired version. This will result in the Sonar Runner dependency `org.codehaus.sonar.runner:sonar-runner-dist:«toolVersion»` being used as the Sonar Runner.

Example 36.3. Configuring Sonar runner version

build.gradle

```
sonarRunner {
    toolVersion = '2.3' // default
}
```

36.5. Analyzing Multi-Project Builds

The Sonar Runner is capable of analyzing whole project hierarchies at once. This yields a hierarchical view in the Sonar web interface, with aggregated metrics and the ability to drill down into subprojects. Analyzing a project hierarchy also takes less time than analyzing each project separately.

To analyze a project hierarchy, apply the Sonar Runner plugin to the root project of the hierarchy. Typically (but not necessarily) this will be the root project of the Gradle build. Information pertaining to the analysis as a whole, like server and database connections settings, have to be configured in the `sonarRunner` block of this project. Any Sonar properties set on the command line also apply to this project.

Example 36.4. Global configuration settings

build.gradle

```
sonarRunner {
    sonarProperties {
        property "sonar.host.url", "http://my.server.com"
        property "sonar.jdbc.url", "jdbc:mysql://my.server.com/sonar"
        property "sonar.jdbc.driverClassName", "com.mysql.jdbc.Driver"
        property "sonar.jdbc.username", "Fred Flintstone"
        property "sonar.jdbc.password", "very clever"
    }
}
```

Configuration shared between subprojects can be configured in a `subprojects` block.

Example 36.5. Shared configuration settings

build.gradle

```
subprojects {
    sonarRunner {
        sonarProperties {
            property "sonar.sourceEncoding", "UTF-8"
        }
    }
}
```

Project-specific information is configured in the `sonarRunner` block of the corresponding project.

Example 36.6. Individual configuration settings

build.gradle

```
project(":project1") {
    sonarRunner {
        sonarProperties {
            property "sonar.language", "grvy"
        }
    }
}
```

To skip Sonar analysis for a particular subproject, set `sonarRunner.skipProject` to `true`.

Example 36.7. Skipping analysis of a project

build.gradle

```
project(":project2") {
    sonarRunner {
        skipProject = true
    }
}
```

36.6. Analyzing Custom Source Sets

By default, the Sonar Runner plugin passes on the project's `main` source set as production sources, and the project's `test` source set as test sources. This works regardless of the project's source directory layout. Additional source sets can be added as needed.

Example 36.8. Analyzing custom source sets

build.gradle

```
sonarRunner {
    sonarProperties {
        properties["sonar.sources"] += sourceSets.custom.allSource.srcDirs
        properties["sonar.tests"] += sourceSets.integTest.allSource.srcDirs
    }
}
```

36.7. Analyzing languages other than Java

To analyze code written in a language other than Java, you'll need to set `sonar.project.language` accordingly. However, note that your Sonar server has to have the Sonar plugin that handles that programming language.

Example 36.9. Analyzing languages other than Java

build.gradle

```
sonarRunner {
    sonarProperties {
        property "sonar.language", "grvy" // set language to Groovy
    }
}
```

As of Sonar 3.4, only one language per project can be analyzed. It is, however, possible to analyze a different language for each project in a multi-project build.

36.8. More on configuring Sonar properties

Let's take a closer look at the `sonarRunner.sonarProperties {}` block. As we have already seen in the examples, the `property()` method allows you to set new properties or override existing ones. Furthermore, all properties that have been configured up to this point, including all properties preconfigured by Gradle, are available via the `properties` accessor.

Entries in the `properties` map can be read and written with the usual Groovy syntax. To facilitate their manipulation, values still have their "idiomatic" type (`File`, `List`, etc.). After the sonarProperties block has been evaluated, values are converted to Strings as follows: Collection values are (recursively) converted to comma-separated Strings, and all other values are converted by calling their `toString()` method.

Because the `sonarProperties` block is evaluated lazily, properties of Gradle's object model can be safely referenced from within the block, without having to fear that they have not yet been set.

36.9. Setting Sonar Properties from the Command Line

Sonar Properties can also be set from the command line, by setting a system property named exactly like the Sonar property in question. This can be useful when dealing with sensitive information (e.g. credentials), environment information, or for ad-hoc configuration.

```
gradle sonarRunner -Dsonar.host.url=http://sonar.mycompany.com -Dsonar.jdbc.passwor
```

While certainly useful at times, we do recommend to keep the bulk of the configuration in a (versioned) build script, readily available to everyone.

A Sonar property value set via a system property overrides any value set in a build script (for the same property). When analyzing a project hierarchy, values set via system properties apply to the root project of the analyzed hierarchy. Each system property starting with ""sonar." will taken into account for the sonar runner setup.

36.10. Controlling the Sonar Runner process

The Sonar Runner is executed in a forked process. This allows fine grained control over memory settings, system properties etc. just for the Sonar Runner process. The `forkOptions` property of the `sonarRunner` extension of the project that applies the `sonar-runner` plugin (Usually the `rootI` but not necessarily) allows the process configuration to be specified. This property is not available in the `SonarRunnerExtension` extension applied to the subprojects.

Example 36.10. setting custom Sonar Runner fork options

build.gradle

```
sonarRunner {
    forkOptions {
        maxHeapSize = '512m'
    }
}
```

For a complete reference about the available options, see `JavaForkOptions`.

36.11. Tasks

The Sonar Runner plugin adds the following tasks to the project.

Table 36.5. Sonar Runner plugin - tasks

Task name	Depends on	Type	Description
`sonarRunner`	-	`SonarRunner`	Analyzes a project hierarchy and stores the results in the Sonar database.

37 OSGiプラグイン

OSGiプラグインは `OsgiManifest` オブジェクトを生成するためのファクトリメソッドを提供します。 `OsgiManifest` は `Manifest` を継承しています。一般的な manifest 操作に関しての詳細については、「Manifest」を参照してください。
Javaプラグインが適用されている場合、OSGiプラグインはデフォルトのjarのmanifestオブジェクトを `OsgiManifest` オブジェクトに置き換えます。置き換えられたmanifestは新しいmanifestにマージされます。

OSGiプラグインは Peter Kriens の BND tool を大いに活用しています。

37.1. 使用方法

OSGiプラグインを使うためには、ビルドスクリプトに以下のコードを追加します:

例37.1 OSGiプラグインの利用

build.gradle

```
apply plugin: 'osgi'
```

37.2. 暗黙的に適用されるプラグイン

Java ベースプラグインを適用します。

37.3. タスク

このプラグインはタスクを追加しません。

37.4. 依存関係の管理

TBD

37.5. 規約オブジェクト

OSGiプラグインは以下の規約オブジェクトを追加します: `OsgiPluginConvention`

37.5.1. 規約プロパティ

OSGiプラグインはプロジェクトに規約プロパティを追加しません。

37.5.2. 規約メソッド

OSGiプラグインは以下の規約メソッドを追加します。詳細については規約オブジェクトのAPIドキュメン

表37.1 OSGiメソッド

メソッド	戻り値	説明
osgiManifest()	`OsgiManifest`	OsgiManifestオブジェクトを返す。
osgiManifest(Closure cl)	`OsgiManifest`	クロージャによって構成されたOsgiManifestオブジェクト

classesディレクトリにあるクラスのパッケージ依存関係と公開するパッケージが解析され、その情報に基 Manifestの Import-Package と Export-Package の値が決定されます。クラスパスにOSGiバンドル形式のjarがある場合、Import-Package の値を決定するためにバンドルの情報が参照されます。 `OsgiManifest` オブジェクトに明示的にinstructionプロパティを追加することでメタデータを追加できます。

例37.2 OSGiのMANIFEST.MFファイルの設定

build.gradle

```
jar {
    manifest { // the manifest of the default jar is of type OsgiManifest
        name = 'overwrittenSpecialOsgiName'
        instruction 'Private-Package',
                'org.mycomp.package1',
                'org.mycomp.package2'
        instruction 'Bundle-Vendor', 'MyCompany'
        instruction 'Bundle-Description', 'Platform2: Metrics 2 Measures Framework'
        instruction 'Bundle-DocURL', 'http://www.mycompany.com'
    }
}
task fooJar(type: Jar) {
    manifest = osgiManifest {
        instruction 'Bundle-Vendor', 'MyCompany'
    }
}
```

instruction の第1引数にはプロパティのキー、第2引数には値を指定します。それらはGradleによって , で結合されます。 利用可能な instruction の詳細については BND tool を参照してください。

38 Eclipse プラグイン

Eclipse プラグインは Eclipse IDE が利用するファイルを生成し、(ファイル - インポート... - 既存プロジェクトをワークスペースへ で) Eclipse 内にプロジェクトをインポートできるようにします。 外部依存関係 (ソースと Javadoc も含む) とプロジェクト依存関係の両方が考慮されます。

1.0-milestone-4 以降、WTP生成コードはリファクタリングされ、`eclipse-wtp` プラグインに分離されました。 もしWTP が必要であれば、`eclipse-wtp` プラグインを適用するだけです。必要でなければ、`eclipse` プラグインを適用するだけでよいです。 この変更は WTP を利用せずに `war` プラグインや `ear` プラグインだけを利用するという Eclipse ユーザの要望によるものです。 `eclipse-wtp` プラグインは 内部的に `eclipse` プラグインも適用するので 両方を適用する必要はありません。

Eclipse プラグインが何を生成するかは どのプラグインと一緒に利用するかで決まります。

表38.1 Eclipse プラグインの振舞い

プラグイン	説明
なし	最小構成の .project ファイルを生成します。
Java	Java の構成を `.project` ファイルに追加します。 `.classpath` ファイルと JDT 設定ファイルを生成します。
Groovy	Groovy の構成を`.project` ファイルに追加します。
Scala	Adds Scala support to `.project` and `.classpath` files.
War	WEBアプリケーションのサポートを`.project`ファイルと`.classpath` ファイルに追加します。 `eclipse-wtp` プラグインを適用している場合のみ WTP 設定ファイルを生成します。
Ear	EAR アプリケーションのサポートを `.project` ファイルに追加します。 `eclipse-wtp` プラグインを適用している場合のみ WTP 設定ファイルを生成します。

Eclipseプラグインは、カスタマイズが容易なように作られています。
生成されたファイルに内容を追加したり削除したりするための、標準化されたフックが用意されています

38.1. 使用方法

Eclipseプラグインを使う為には、ビルドスクリプトに下記を含めます：

例38.1 Eclipseプラグインの使用方法

build.gradle

```
apply plugin: 'eclipse'
```

Eclipseプラグインは多くのタスクをプロジェクトに追加しますが、主に利用するのは eclipse と clea
タスクだけです。

38.2. タスク

Eclipseプラグインはプロジェクトに以下のタスクを追加します。

表38.2 Eclipse プラグイン - タスク

タスク名	依存先	型
eclipse	eclipseProject, eclipseClasspath, eclipseJdt, eclipseWtpComponent, cleanEclipseWtpFacet	Task
cleanEclipse	cleanEclipseProject, cleanEclipseClasspath, cleanEclipseJdt, cleanEclipseWtpComponent, cleanEclipseWtpFacet	Delete
cleanEclipseProject	-	Delete
cleanEclipseClasspath	-	Delete
cleanEclipseJdt	-	Delete
cleanEclipseWtpComponent	-	Delete
cleanEclipseWtpFacet	-	Delete
eclipseProject	-	GenerateEclipseProject
eclipseClasspath	-	GenerateEclipseClasspat
eclipseJdt	-	GenerateEclipseJdt
eclipseWtpComponent	-	GenerateEclipseWtpCompo
eclipseWtpFacet	-	GenerateEclipseWtpFacet

38.3. 設定

表38.3 Eclipseプラグインの設定

モデル	参照名	説明
EclipseModel	eclipse	DSLにあったやり方でEclipseプラグインの
EclipseProject	eclipse.project	プロジェクトの情報を設定できます。
EclipseClasspath	eclipse.classpath	クラスパスの情報を設定できます。 Allow information
EclipseJdt	eclipse.jdt	JDT の情報を設定できます。(source/targ
EclipseWtpComponent	eclipse.wtp.component	eclipse-wtp プラグインを適用した場合のみ、WTPコン
EclipseWtpFacet	eclipse.wtp.facet	eclipse-wtp プラグインを適用した場合のみ、WTPファ

38.4. 生成されたファイルをカスタマイズする

Eclipseプラグインは、生成されるメタデータファイルをカスタマイズできるようになっています。
このプラグインでは、DSLを使ってモデルオブジェクトの設定を行うことができます。このモデルオブジ
DSLで設定したEclipseのモデルオブジェクトは、既存のXMLメタデータとマージされ、最終的な新しい
メタデータの内容は、ドメインオブジェクトという形で表現されます。マージ前後にそのドメインオブジ
また、さらに低レベルなフックも用意されており、メタデータをファイルに保存する前に、生のXMLデー
そのフックを使えば、プラグインがモデル化していない設定要素を微調整したり編集したりすることも可

38.4.1. マージする

既存のEclipseファイルの、対象のセクションだけを生成された内容で変更、もしくは上書きし、それ以

38.4.1.1. マージせずに完全に上書きする

既存の Eclipse ファイルを完全に上書きしたい場合は、対応する生成タスクと一緒に clean タスクも実行します。 例えば **`gradle cleanEclipse eclipse`** のように… (もちろんこの順番で…) もし、この動きをデフォルトにしたい場合は、ビルドスクリプトに `eclipse.dependsOn(cleanEclipse)` を追加します。 そうすれば 明示的に clean タスクを実行する必要はありません。

This strategy can also be used for individual files that the plugin would generate. For instance, this can be done for the “`.classpath`” file with “**`gradle cleanEclipseClasspath eclipseC`**”.

個々のファイルについても同じように完全に上書きすることができます。例えば、 **`gradle cleanEcl`** のように…

38.4.2. 生成ライフサイクルにフックする

Eclipse プラグインは Gradle が生成する Eclipse ファイルのセクションをモデリングしたドメインクラスを提供します。
生成ライフサイクルは次の通りです。

1. ファイルが読み込まれる。ファイルが存在しない場合は、Gradleがデフォルトの内容を用意する。
2. `beforeMerged` フックが実行され、既存の設定内容を表すドメインオブジェクトがフックに引き渡される。
3. Gradleのビルド内容から推測される、またはEclipse DSLで明示された設定内容と、既存の設定内容がマージされる。
4. `whenMerged` フックが実行され、保存される予定の設定内容を表すドメインオブジェクトがフックに引き渡される。
5. `withXml`フックが実行され、保存される予定の生XMLがフックに引き渡される。
6. 最終的なXMLが保存される。

以下は Eclipseモデルの型毎の利用ドメインオブジェクトの一覧表です。

表38.4 高度な構成フック

モデル	**beforeMerged { arg -> }** 引数型	**whenMerged { arg -> }** 引数型	**withXı** 引数型
`EclipseProject`	`Project`	`Project`	`XmlPr`
`EclipseClasspath`	`Classpath`	`Classpath`	`XmlPr`
`EclipseJdt`	`Jdt`	`Jdt`	
`EclipseWtpComponent`	`WtpComponent`	`WtpComponent`	`XmlPr`
`EclipseWtpFacet`	`WtpFacet`	`WtpFacet`	`XmlPr`

38.4.2.1. 既存の内容の一部を上書きする

完全に上書きしてしまうと 既存の内容を全て破棄してしまいます。 それにより IDE で直接変更した内容が消されてしまいます。`beforeMerged` フックを使えば 既存の内容の一部だけを上書きすることができます。 次の例は 存在する全ての依存関係を `classpath` ドメインオブジェクトから削除します。

例38.2 classpath の一部を上書き

build.gradle

```
eclipse.classpath.file {
    beforeMerged { classpath ->
        classpath.entries.removeAll { entry -> entry.kind == 'lib' || entry.kind == '
    }
}
```

この例で生成された `.classpath` ファイルには Gradleによって生成された依存関係のエントリしか含みません。
元のファイルにあった他の依存関係のエントリは全く含みません。(依存関係のエントリの場合はこれが

.classpathファイルの他のセクションは そのまま残るかマージされるでしょう。 .project ファイルの ネーチャーでも同様です。

例38.3 project の一部を上書き

build.gradle

```
eclipse.project.file.beforeMerged { project ->
    project.natures.clear()
}
```

38.4.2.2. 完全に populate されたドメインオブジェクトを変更する

whenMergedフックは完全にデータが設定されたドメインオブジェクトを操作することができます。 これは Eclipse のファイルをカスタマイズするのにはよい方法です。 以下は Eclipse プロジェクトの全ての依存関係をエクスポートする方法です:

例38.5 依存関係のエクスポート

build.gradle

```
eclipse.classpath.file {
    whenMerged { classpath ->
        classpath.entries.findAll { entry -> entry.kind == 'lib' }*.exported = fals
    }
}
```

38.4.2.3. XML表現を変更する

withXmlフックはディスクに書き出される直前のメモリ内の XML 表現を操作することができます。 ただ、Groovy の XML サポートがいろいろと補ってくれたとしても、このアプローチはドメインオブジェクトを操作するほど使 その代わりに、生成されたファイルを通して全ての操作が可能です。もちろん、ドメインオブジェクトに

例38.7 XML のカスタマイズCustomizing the XML

build.gradle

```
apply plugin: 'eclipse-wtp'

eclipse.wtp.facet.file.withXml { provider ->
    provider.asNode().fixed.find { it.@facet == 'jst.java' }.@facet = 'jst2.java'
}
```

39
IDEAプラグイン

IDEAプラグインは、IntelliJ IDEAが使用するファイルを生成し、プロジェクトをIDEA(File - Open Project)で開けるようにします。
外部依存関係(関連するソースやjavadocファイルを含む)とプロジェクト依存関係についても考慮されてい

IDEAプラグインが実際に生成するものは、使用されているプラグインによって異なります。

表39.1 IDEAプラグインの振る舞い

プラグイン	説明
なし	IDEAモジュールファイルを生成する。プロジェクトがルートプロジェクトだった場合は
Java	モジュールファイルとプロジェクトファイルに、Javaの設定を追加する。

IDEAプラグインで重視されているものの一つがカスタマイズ性です。生成するファイルに内容を追加し

39.1. 使用方法

IDEAプラグインを使うには、ビルドスクリプトに次の行を追加します。

例39.1 IDEAプラグインを使う

build.gradle

```
apply plugin: 'idea'
```

IDEAプラグインは、プロジェクトにいくつかのタスクを追加します。主に使うものは`idea`タスクと`clea`
タスクでしょう。

39.2. タスク

IDEAプラグインはプロジェクトに以下のタスクを追加します。 `clean`タスクが`cleanIdeaWorkspace`
タスクに依存していないことに注意してください。
ワークスペースにはユーザーごとの一時データが多く含まれており、一般的にはIDEAの外からは操作さ

表39.2 IDEAプラグイン - タスク

タスク名	依存先	型	説明
idea	ideaProject, ideaModule, ideaWorkspace		全てのIDEA設定フ
cleanIdea	cleanIdeaProject, cleanIdeaModule	Delete	IDEAの設定ファイ
cleanIdeaProject	-	Delete	IDEAのプロジェク
cleanIdeaModule	-	Delete	IDEAのモジュール
cleanIdeaWorkspace	-	Delete	IDEAのワークスペ
ideaProject	-	GenerateIdeaProject	.ipr ファイルを生成す
ideaModule	-	GenerateIdeaModule	.imlファイルを作
ideaWorkspace	-	GenerateIdeaWorkspace	.iws ファイルを生成す

39.3. 設定

表39.3 IDEAプラグインの設定

モデル	参照名	説明
IdeaModel	idea	DSLフレンドリーな方法でIDEAプラグインの設定を行うため
IdeaProject	idea.project	プロジェクト情報の設定ができる
IdeaModule	idea.module	モジュールの設定ができる
IdeaWorkspace	idea.workspace	ワークスペースXMLの設定ができる

39.4. 生成するファイルのカスタマイズ

IDEAプラグインには、生成される内容をカスタマイズするためのフックと特徴があります。
なお、ワークスペースファイルは、対応するドメインオブジェクトが実質的に空なので、`withXml` フックを使用しないと効果的に操作できません。

IDEAプラグインのタスクは、既存のIDEAファイルを検知し、生成された内容をそれにマージします。

39.4.1. マージ

既存のIDEAファイルのセクションのうち、生成される内容の対象になっているものは、セクションごとに
それ以外のセクションはそのまま残されます。

39.4.1.1. マージを無効にして、完全に上書きする

既存のIDEAファイルを完全に上書きするには、クリーンタスクを、対応する生成タスクと一緒に使用して
gradle cleanIdea ideaをこの順序で使用します。
デフォルトでそのように動作させたいのであれば、`tasks.idea.dependsOn(cleanIdea)`
というコードをビルドスクリプトに追加してください。明示的にクリーンタスクを実行する必要がなくな

この戦略はプラグインが生成する個々のファイルに対しても適用可能で、例えば “`.iml`
”ファイルに対して" **gradle cleanIdeaModule ideaModule**"のようにできます。

39.4.2. 生成ライフサイクルへフックする

このプラグインでは、様々なオブジェクトにより生成されるメタデータファイルの各セクションを表現し
メタデータファイルを生成する際のライフサイクルは次のとおりです。

1. ファイルが読み込まれる。ファイルが存在しない場合は、Gradleがデフォルトの内容を用意する。
2. `beforeMerged`
   フックが実行され、既存の設定内容を表すドメインオブジェクトがフックに引き渡される。
3. Gradleのビルド内容から推測される、またはDSLで明示された設定内容と、既存の設定内容がマージ
4. `whenMerged`
   フックが実行され、保存される予定の設定内容を表すドメインオブジェクトがフックに引き渡される。
5. `withXml`フックが実行され、保存される予定の生XMLがフックに引き渡される。
6. 最終的なXMLが保存される。

各モデルで使われるドメインオブジェクトは、次の表の通りです。

表39.4 IDEAプラグインフック

モデル	**beforeMerged { arg -> }** 引数の型	**whenMerged { arg -> }** 引数の型	**withXml { ar** 引数の型
`IdeaProject`	`Project`	`Project`	`XmlProvider`
`IdeaModule`	`Module`	`Module`	`XmlProvider`
`IdeaWorkspace`	`Workspace`	`Workspace`	`XmlProvider`

39.4.2.1. 既存の内容を部分的に上書きする

完全な上書きは、既存の内容を全て破棄するので、IDEで直接行った変更も失われてしまいます。
`beforeMerged`フックを使うと、既存の内容のうち特定の部分だけを上書きすることができます。
次の例では、`Module`ドメインオブジェクトから、既存の依存関係を全て削除しています。

例39.2 モジュールの部分的な上書き

build.gradle

```
idea.module.iml {
    beforeMerged { module ->
        module.dependencies.clear()
    }
}
```

結果として生成されるモジュールファイルには、Gradleが生成した依存関係のエントリのみが含まれてま
そして、モジュールファイルの別のセクションはそのまま、またはマージされた状態で残されます。
同じことはプロジェクトファイルのモジュールパスを使用しても実現可能です。

例39.3 プロジェクトの部分的な上書き

build.gradle

```
idea.project.ipr {
    beforeMerged { project ->
        project.modulePaths.clear()
    }
}
```

39.4.2.2. 読み込まれた全ドメインオブジェクトを編集する

`whenMerged`フックを使うと、読み込まれた全てのドメインオブジェクトを操作できます。
IDEAファイルをカスタマイズするときは、多くの場合このフックを使うのが好ましい方法です。例えば、

例39.4 Export Dependencies

build.gradle

```
idea.module.iml {
    whenMerged { module ->
        module.dependencies*.exported = true
    }
}
```

39.4.2.3. XMLデータを編集する

`w i t h X m l`
フックを使うと、メモリ内のXMLデータを、ファイルに書き出される直前に操作できます。
GroovyのXMLサポートで大幅にフォローされるものの、このアプローチはドメインオブジェクトを操作
その代わり、ドメインオブジェクトで表現されないような部分も含め、生成されるファイルの全てを制御

例39.5 XMLをカスタマイズする

build.gradle

```
idea.project.ipr {
    withXml { provider ->
        provider.node.component
                .find { it.@name == 'VcsDirectoryMappings' }
                .mapping.@vcs = 'Git'
    }
}
```

39.5. その他の注意事項

生成されるIDEAファイルでは、依存関係が絶対パスで保存されます。
しかし、Gradleキャッシュを場所を参照するパス変数を定義すれば、IDEAは依存関係の絶対パスをその
そのようなパス変数を使いたい場合は、`idea.pathVariables`
を使って変数を設定してください。それにより、重複なしで正しくマージを行うこともできます。

40
ANTLRプラグイン

ANTLRプラグインはJavaプラグインを拡張して、ANTLR
を利用したパーサーの生成に対するサポートを追加します。

40.1. 使用方法

ANTLRプラグインを使うには、ビルドスクリプトに以下を含めます:

例40.1 ANTLRプラグインの利用

build.gradle

```
apply plugin: 'antlr'
```

40.2. タスク

ANTLRプラグインは以下に示すいくつかのタスクをプロジェクトに追加します。

表40.1 ANTLR plugin - タスク

タスク名	依存先	タイプ	説明
generateGrammarSource	-	AntlrTask	すべてのプロダクションANTLRグラマー
generateTestGrammarSource	-	AntlrTask	すべてのテストANTLRグラマーに対す
generate*SourceSet*GrammarSource	-	AntlrTask	指定されたソースセットですべてのAN

ANTLRプラグインはJavaプラグインによって追加されたタスクに以下の依存関係を追加します。

表40.2 ANTLR plugin - 追加のタスク依存関係

タスク名	依存先
compileJava	generateGrammarSource
compileTestJava	generateTestGrammarSource
compile*SourceSet*Java	generate*SourceSet*GrammarSource

40.3. プロジェクトレイアウト

表40.3 ANTLRプラグイン - プロジェクトレイアウト

ディレクトリ	意味
`src/main/antlr`	プロダクションANTLRグラマーファイル
`src/test/antlr`	テストANTLRグラマーファイル
`src/`*`sourceSet`*`/antlr`	指定されたソースセットに対するANTLRグラマーファイル

40.4. 依存関係管理

ANTLRプラグインは依存性コンフィグレーション`antlr`を追加します。
これは利用したいANTLRのバージョンを宣言するのに使えます。

例40.2 ANTLRバージョン宣言

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    antlr 'antlr:antlr:2.7.7'
}
```

40.5. 規約プロパティ

ANTLRプラグインはいかなる規約プロパティも追加しません。

40.6. ソースセットプロパティ

ANTLRプラグインはプロジェクトのソースセット毎に以下のプロパティを追加します。

表40.4 ANTLRプラグイン - ソースセットプロパティ

プロパティ名	タイプ	デフォルト値	説明
`antlr`	`SourceDirectorySet` (read-only)	Not null	このソースセットの ANTLRソースディレ を含み、それ以外の
`antlr.srcDirs`	`Set<File>`。 「入力ファイルセットを指定する 」 で説明した任意のパターンが設定可能。	`[`*`projectDir`*`/src/`*`name`*`/antlr]`	このソースセットの

41
プロジェクトレポートプラグイン

プロジェクトレポートプラグインは、ビルドに関する有益な情報を含むレポートを生成するタスクをプロ
これらのタスクは、**gradle tasks**、**gradle dependencies** および**gradle properties**
によるコマンドラインレポートとまったく同じ内容を生成します (
「ビルドに関する情報を取得する」を参照)。
コマンドラインレポートとは対照的に、レポートプラグインはレポートをファイルに出力します。
プラグインによって追加されたすべてのレポートタスクに依存する集約タスクもあります。

Gradleの将来のリリースではもっと多くの情報をレポートに追加する予定です。

41.1. 使用法

プロジェクトレポートプラグインを使うには、ビルドスクリプトに以下を含めます:

```
apply plugin: 'project-report'
```

41.2. タスク

プロジェクトレポートプラグインは以下のタスクを定義しています:

表41.1 プロジェクトレポートプラグイン - タスク

タスク名	依存先	タイプ	説明
dependencyReport	-	DependencyReportTask	プロジ
htmlDependencyReport	-	HtmlDependencyReportTask	Gene deper set of
propertyReport	-	PropertyReportTask	プロジ
taskReport	-	TaskReportTask	プロジ
projectReport	dependencyReport, propertyReport, taskReport, htmlDependencyReport	Task	すべて

41.3. プロジェクトレイアウト

プロジェクトレポートプラグインは特定のプロジェクトレイアウトを要求しません。

41.4. 依存関係

プロジェクトレポートプラグインはいかなる依存関係コンフィグレーションも定義しません。

41.5. 規約プロパティ

プロジェクトレポートプラグインは以下の規約プロパティを定義します:

表41.2 プロジェクトレポートプラグイン - 規約プロパティ

プロパティ名	タイプ	デフォルト値	説明
`reportsDirName`	`String`	`reports`	レポートを生成するデ
`reportsDir`	`File` (read-only)	*`buildDir/reportsDirName`*	レポートを生成するデ
`projects`	`Set<Project>`	`1Set`	レポート生成対象のプ
`projectReportDirName`	`String`	`project`	プロジェクトレポート
`projectReportDir`	`File` (read-only)	*`reportsDir/projectReportDirName`*	プロジェクトレポート

規約プロパティは `ProjectReportsPluginConvention` 型の規約オブジェクトによって提供されます。

42
通知プラグイン

通知プラグインを使うと、ビルド実行中に通知を送信することができます。以下の通知システムがサポー

- Twitter
- notify-send (Ubuntu)
- Snarl (Windows)
- Growl (Mac OS X)

42.1. 使用方法

通知プラグインを使うには、まずビルドスクリプトにプラグインを適用してください。

Example 42.1. 通知プラグインを使用する

build.gradle

```
apply plugin: 'announce'
```

次に、選んだ通知方法に応じて設定を行います。設定用のプロパティについては、以下のテーブルを参照

Example 42.2. 通知プラグインの設定

build.gradle

```
announce {
  username = 'myId'
  password = 'myPassword'
}
```

最後に、announceメソッドで通知を送信します。

Example 42.3. 通知プラグインを使用する

build.gradle

```
task helloWorld << {
    println "Hello, world!"
}

helloWorld.doLast {
    announce.announce("helloWorld completed!", "twitter")
    announce.announce("helloWorld completed!", "local")
}
```

a n n o u n c e
メソッドは二つの引数を取ります。一つは通知メッセージで、もう一つは使用する通知用サービスです。サポートされている通知用サービスとその設定値は以下の通りです。

Table 42.1. 通知プラグイン対応サービス

通知サービス	オペレーティングシステム	設定プロパティ	備考
twitter	何でも	username, password	
snarl	Windows		
growl	Mac OS X		
notify-send	Ubuntu		notify-sendパッケージをインストーを実行してインストールしてくださ
local	Windows, Mac OS X, Ubuntu		snarl、growl、notify-sendの中から

42.2. 設定

AnnouncePluginExtensionを参照してください。

43
ビルド通知プラグイン

ビルド通知プラグインはまだ試験的な
プラグインです。DSLやその他設定が後のバージョンで変更される可能性もありますので注意してくだ

ビルド通知プラグインは、通知
プラグインを使って、ビルドで重要なイベントが発生したときにローカル上で通知します。

43.1. 使用方法

ビルド通知プラグインを使うには、ビルドスクリプトに以下を含めます:

例43.1 ビルド通知プラグインの利用

build.gradle

```
apply plugin: 'build-announcements'
```

通知先を調整したい場合は、通知
プラグインのコンフィグレーションでローカル通知機能を切り替えられます。

また、初期化スクリプトを使ってプラグインを適用することもできます:

例43.2 初期化スクリプトからビルド通知プラグインを使う

init.gradle

```
rootProject {
    apply plugin: 'build-announcements'
}
```

44
The Distribution Plugin

> The distribution plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The distribution plugin facilitates building archives that serve as distributions of the project. Distribution archives typically contain the executable application and other supporting files, such as documentation.

44.1. Usage

To use the distribution plugin, include the following in your build script:

Example 44.1. Using the distribution plugin

build.gradle

```
apply plugin: 'distribution'
```

The plugin adds an extension named "`distributions`" of type `DistributionContainer` to the project. It also creates a single distribution in the distributions container extension named "`mai`
". If your build only produces one distribution you only need to configure this distribution (or use the defaults).

You can run "**`gradle distZip`**" to package the main distribution as a ZIP, or "**`gradle distTar`**
" to create a TAR file. The files will be created at "`$buildDir/distributions/$project.name`
".

You can run "**`gradle installDist`**" to assemble the uncompressed distribution into "`$buildL`
".

44.2. Tasks

The Distribution plugin adds the following tasks to the project:

Table 44.1. Distribution plugin - tasks

Task name	Depends on	Type	Description
`distZip`	-	`Zip`	Creates a ZIP archive of the distribution contents
`distTar`	-	`Tar`	Creates a TAR archive of the distribution contents
`installDist`	-	`Sync`	Assembles the distribution content and installs it on the current machine

For each extra distribution set you add to the project, the distribution plugin adds the following tasks:

Table 44.2. Multiple distributions - tasks

Task name	Depends on	Type	Description
`${distribution.name}DistZip`	-	`Zip`	Creates a ZIP archive of the distribution contents
`${distribution.name}DistTar`	-	`Tar`	Creates a TAR archive of the distribution contents
`install${distribution.name.capitalize()}Dist`	-	`Sync`	Assembles the distribution content and installs it on the current machine

Example 44.2. Adding extra distributions

build.gradle

```
apply plugin: 'distribution'

version = '1.2'
distributions {
    custom {}
}
```

This will add following tasks to the project:

- customDistZip
- customDistTar
- installCustomDist

Given that the project name is “`myproject`” and version “`1.2`”, running “**`gradle customDis`**” will produce a ZIP file named “`myproject-custom-1.2.zip`”.

Running “**`gradle installCustomDist`**” will install the distribution contents into “*`$buildDir/`*”.

44.3. Distribution contents

All of the files in the "`src/$distribution.name/dist`" directory will automatically be included in the distribution. You can add additional files by configuring the `Distribution` object that is part of the container.

Example 44.3. Configuring the main distribution

build.gradle

```
apply plugin: 'distribution'

distributions {
    main {
        baseName = 'someName'
        contents {
            from { 'src/readme' }
        }
    }
}
```

In the example above, the content of the "`src/readme`" directory will be included in the distribution (along with the files in the "`src/dist/main`" directory which are added by default).

The "`baseName`" property has also been changed. This will cause the distribution archives to be created with a different name.

45
アプリケーション プラグイン

Gradle アプリケーション プラグインは言語プラグインを一般的なアプリケーションの関連タスクで拡張しています。 JVM 用アプリケーションの実行 及び ビルドが可能です。

45.1. 使用方法

アプリケーション プラグインを使うためには、ビルドスクリプトに下記を含めます：

例45.1 Using the application plugin

build.gradle

```
apply plugin:'application'
```

アプリケーションのメインクラスを定義するには、次のように mainClassName プロパティを設定するだけでよいです。

例45.2 Configure the application main class

build.gradle

```
mainClassName = "org.gradle.sample.Main"
```

あとは **gradle run** を実行するだけで、アプリケーションを実行することができます。 Gradle は アプリケーションクラスのビルドから 実行時の依存解決、 適切なクラスパスでのアプリケーションの起動まで全て面倒みてくれます。

このプラグインではアプリケーション用のディストリビューションをビルドすることもできます。 ディストリビューションには OS 依存の起動スクリプトと一緒に アプリケーションの実行時の依存ライブラリもパッケージングされます。 **gradle install** で build/ 配下に アプリケーションイメージを生成することができます。 **gradle distZip** で ディストリビューションを含む ZIP ファイルを生成することができます。

If your Java application requires a specific set of JVM settings or system properties, you can configure the `applicationDefaultJvmArgs` property. These JVM arguments are applied to the `run` task and also considered in the generated start scripts of your distribution.

例45.3 Configure default JVM settings

build.gradle

```
applicationDefaultJvmArgs = ["-Dgreeting.language=en"]
```

45.2. タスク

アプリケーション プラグインは 次のタスクをプロジェクトに追加します。

表45.1 アプリケーション プラグイン - タスク

タスク名	依存先	型	説明
run	classes	JavaExec	アプリケーションを起動します。
startScripts	jar	CreateStartScripts	プロジェクトを JVM アプリケーシ
installApp	jar, startScripts	Sync	指定したディレクトリ配下にアプリ
distZip	jar, startScripts	Zip	実行ライブラリと OS 依存スクリプ アーカイブを生成します。
distTar	jar, startScripts	Tar	実行ライブラリとOS依存スクリプ

45.3. 規約プロパティ

アプリケーション プラグインは いくつかのプロパティをプロジェクトに追加します。そのプロパティを設定すれば挙動が変えられます。Project を参照してください。

45.4. ディストリビューションに他のリソースを含め

このプラグインは、applicationDistributionというCopySpec 型の規約プロパティを追加します。 これは、installAppタスクとdistZip タスクにより何がディストリビューションに含められるのか、その仕様を定義したものです。 実行用スクリプトをディストリビューションのbinディレクトリに、必要なjarをlib にコピーするほか、src/distディレクトリにある全てのファイルをコピーします。 何らかの静的ファイルをディストリビューションに含めたい場合は、単にsrc/dist ディレクトリにそれらのファイルを置くだけで大丈夫です。

あるプロジェクトが、ディストリビューションに含めるファイル(例えばドキュメントなど)を生成する、そのような場合、applicationDistribution コピー仕様を追加することで、ファイルをディストリビューションに追加することができます。

例45.4 他タスクの出力をアプリケーションのディストリビューションに含める

build.gradle

```
task createDocs {
    def docs = file("$buildDir/docs")
    outputs.dir docs
    doLast {
        docs.mkdirs()
        new File(docs, "readme.txt").write("Read me!")
    }
}

applicationDistribution.from(createDocs) {
    into "docs"
}
```

ディストリビューションにタスクの出力(「タスクの入力物と出力物を宣言する」
参照)を格納するように指定すると、
Gradleは、ディストリビューションを作成する前にそのタスクを実行しなければならないのだと判断し、

例45.5 ディストリビューションのファイルを自動的に作成する

`gradle distZip` の出力

```
> gradle distZip
:createDocs
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:startScripts
:distZip

BUILD SUCCESSFUL

Total time: 1 secs
```

46
The Java Library Distribution Plugin

> The Java library distribution plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Java library distribution plugin adds support for building a distribution ZIP for a Java library. The distribution contains the JAR file for the library and its dependencies.

46.1. Usage

To use the Java library distribution plugin, include the following in your build script:

Example 46.1. Using the Java library distribution plugin

build.gradle

```
apply plugin: 'java-library-distribution'
```

To define the name for the distribution you have to set the `baseName` property as shown below:

Example 46.2. Configure the distribution name

build.gradle

```
distributions {
    main{
        baseName = 'my-name'
    }
}
```

The plugin builds a distribution for your library. The distribution will package up the runtime dependencies of the library. All files stored in `src/main/dist` will be added to the root of the archive distribution. You can run “**`gradle distZip`**” to create a ZIP file containing the distribution.

46.2. Tasks

The Java library distribution plugin adds the following tasks to the project.

Table 46.1. Java library distribution plugin - tasks

Task name	Depends on	Type	Description
`distZip`	`jar`	`Zip`	Creates a full distribution ZIP archive including runtime libraries.

46.3. Including other resources in the distribution

All of the files from the `src/dist` directory are copied. To include any static files in the distribution, simply arrange them in the `src/dist` directory, or add them to the content of the distribution.

Example 46.3. Include files in the distribution

build.gradle

```
distributions {
    main {
        baseName = 'my-name'
        contents {
            from { 'src/dist' }
        }
    }
}
```

47
Build Init Plugin

The Build Init plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Gradle Build Init plugin can be used to bootstrap the process of creating a new Gradle build. It supports creating brand new projects of different types as well as converting existing builds (e.g. An Apache Maven build) to be Gradle builds.

Gradle plugins typically need to be applied to a project before they can be used (see 「プラグインの適用 」). The Build Init plugin is an automatically applied plugin, which means you do not need to apply it explicitly. To use the plugin, simply execute the task named `init` where you would like to create the Gradle build. There is no need to create a "stub" `build.gradl` file in order to apply the plugin.

It also leverages the `wrapper` task from the Wrapper plugin (see Chapter 48, Wrapper Plugin), which means that the Gradle Wrapper will also be installed into the project.

47.1. Tasks

The plugin adds the following tasks to the project:

Table 47.1. Build Init plugin - tasks

Task name	Depends on	Type	Description
`init`	`wrapper`	`InitBuild`	Generates a Gradle project.
`wrapper`	-	`Wrapper`	Generates Gradle wrapper files.

47.2. What to set up

The `init` supports different build setup types. The type is specified by supplying a `--type` argument value. For example, to create a Java library project simply execute: `gradle init --type`
.

If a `--type` parameter is not supplied, Gradle will attempt to infer the type from the environment. For example, it will infer a type value of "`pom`" if it finds a `pom.xml` to convert to

a Gradle build.

If the type could not be inferred, the type "`basic`" will be used.

All build setup types include the setup of the Gradle Wrapper.

47.3. Build init types

> As this plugin is currently incubating, only a few build init types are currently supported. More types will be added in future Gradle releases.

47.3.1. "`pom`" (Maven conversion)

The "`pom`" type can be used to convert an Apache Maven build to a Gradle build. This works by converting the POM to one or more Gradle files. It is only able to be used if there is a valid "`pom.`" file in the directory that the `init` task is invoked in. This type will be automatically inferred if such a file exists.

The Maven conversion implementation was inspired by the maven2gradle tool that was originally developed by Gradle community members.

The conversion process has the following features:

- Uses effective POM and effective settings (support for POM inheritance, dependency management, properties)
- Supports both single module and multimodule projects
- Supports custom module names (that differ from directory names)
- Generates general metadata - id, description and version
- Applies maven, java and war plugins (as needed)
- Supports packaging war projects as jars if needed
- Generates dependencies (both external and inter-module)
- Generates download repositories (inc. local Maven repository)
- Adjusts Java compiler settings
- Supports packaging of sources and tests
- Supports TestNG runner
- Generates global exclusions from Maven enforcer plugin settings

47.3.2. "`java-library`"

The "`java-library`" build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`java`" plugin
- Uses the "`mavenCentral`" dependency repository
- Uses JUnit for testing
- Has directories in the conventional locations for source code

- Contains a sample class and unit test, if there are no existing source or test files

47.3.3. “`scala-library`”

The “`scala-library`” build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the “`scala`” plugin
- Uses the “`mavenCentral`” dependency repository
- Uses Scala 2.10
- Uses ScalaTest for testing
- Has directories in the conventional locations for source code
- Contains a sample scala class and an associated ScalaTest test suite, if there are no existing source or test files

47.3.4. “`groovy-library`”

The “`groovy-library`” build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the “`groovy`” plugin
- Uses the “`mavenCentral`” dependency repository
- Uses Groovy 2.x
- Uses Spock testing framework for testing
- Has directories in the conventional locations for source code
- Contains a sample Groovy class and an associated Spock specification, if there are no existing source or test files

47.3.5. “basic”

The “`basic`” build init type is useful for creating a fresh new Gradle project. It creates a sample `build.gradle` file, with comments and links to help get started.

This type is used when no type was explicitly specified, and no type could be inferred.

48
Wrapper Plugin

> The wrapper plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Gradle wrapper plugin allows the generation of Gradle wrapper files by adding a `Wrapper` task, that generates all files needed to run the build using the Gradle Wrapper. Details about the Gradle Wrapper can be found in 62章Gradleラッパー .

48.1. Usage

Without modifying the `build.gradle` file, the wrapper plugin can be auto-applied to the root project of the current build by running “`gradle wrapper`” from the command line. This applies the plugin if no task named `wrapper` is already defined in the build.

48.2. Tasks

The wrapper plugin adds the following tasks to the project:

Table 48.1. Wrapper plugin - tasks

Task name	Depends on	Type	Description
`wrapper`	-	`Wrapper`	Generates Gradle wrapper files.

49
The Build Dashboard Plugin

> The build dashboard plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Build Dashboard plugin can be used to generate a single HTML dashboard that provides a single point of access to all of the reports generated by a build.

49.1. Usage

To use the Build Dashboard plugin, include the following in your build script:

Example 49.1. Using the Build Dashboard plugin

build.gradle

```
apply plugin: 'build-dashboard'
```

Applying the plugin adds the `buildDashboard` task to your project. The task aggregates the reports for all tasks that implement the `Reporting` interface from all projects in the build. It is typically only applied to the root project.

The `buildDashboard` task does not depend on any other tasks. It will only aggregate the reporting tasks that are independently being executed as part of the build run. To generate the build dashboard, simply include this task in the list of tasks to execute. For example, “**`gradle bui:`** ” will generate a dashboard for all of the reporting tasks that are dependents of the `build` task.

49.2. Tasks

The Build Dashboard plugin adds the following task to the project:

Table 49.1. Build Dashboard plugin - tasks

Task name	Depends on	Type	Description
`buildDashboard`	-	`GenerateBuildDashboard`	Generates build dashboard report.

49.3. Project layout

The Build Dashboard plugin does not require any particular project layout.

49.4. Dependency management

The Build Dashboard plugin does not define any dependency configurations.

49.5. Configuration

You can influence the location of build dashboard plugin generation via `ReportingExtension`.

50
The Java Gradle Plugin Development Plugin

> The Java Gradle plugin development plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Java Gradle Plugin development plugin can be used to assist in the development of Gradle plugins. It automatically applies the Java plugin, adds the `gradleApi()` dependency to the compile configuration and performs validation of plugin metadata during `jar` task execution.

50.1. Usage

To use the Java Gradle Plugin Development plugin, include the following in your build script:

Example 50.1. Using the Java Gradle Plugin Development plugin

build.gradle

```
apply plugin: 'java-gradle-plugin'
```

Applying the plugin automatically applies the Java plugin and adds the `gradleApi()` dependency to the compile configuration. It also decorates the `jar` task with validations.

The following validations are performed:

- There is a plugin descriptor defined for the plugin.
- The plugin descriptor contains an `implementation-class` property.
- The `implementation-class` property references a valid class file in the jar.

Any failed validations will result in a warning message.

51
依存関係の管理

51.1. はじめに

依存関係の管理は、あらゆるビルドで決定的に重要になる機能です。Gradleも、理解しやすく、既存の様
Gradleの管理機能はとても柔軟でカスタマイズ性の高いものですが、MavenやIvyでの管理に慣れ親しん

Gradleの依存関係管理機能で特筆すべき点を挙げると、

- 推移的な依存関係の管理:
  Gradleは、プロジェクトの依存関係ツリー全体を完全にコントロールできます。
- 未管理の依存関係もサポート:
  プロジェクトの依存ファイルを単純にバージョン管理システムや共有ドライブに格納したい場合も、
- 依存関係定義のカスタマイズをサポート:
  Gradleでは、依存関係の階層構造をビルドスクリプトで直接定義できます。
- 非常にカスタマイズ性の高い依存関係の解決方法:
  Gradleでは依存関係を解決する際のルールをカスタマイズでき、簡単に依存関係を差し替えることが
- Maven、Ivyとの完全な互換性:
  既にMavenのPOMやIvyファイルで依存関係を定義しているかもしれません。Gradleは人気のある様々
- 既存の依存関係管理インフラとの連携:
  Gradleは、Maven、Ivyどちらの形式のリポジトリとも互換性があります。既にArchivaやNexus、Art

現在無数に存在するオープンソースライブラリは、それぞれ互いに依存し合っていて、さらにバージョン

51.1.1. ビルドツール移行に必要な依存関係管理の柔軟性

依存関係の管理で特に問題が発生しやすいのは、あるビルドシステムから別のビルドシステムへ移行する
AntやMavenからGradleへ移行する際には、難しい問題に突き当たることがあるかもしれません。
例えば、よくあるAntプロジェクトですが、バージョンのないjarファイルがファイルシステムに置かれて
一方Gradleでは、依存関係をどこから、どんなメタデータで取得していたとしても、新しいビルドでそれ

プロジェクトが独自の方法で依存関係を管理していたとしても、例え依存関係のマスターデータ
にEclipseの.classpathを使っているようなプロジェクトであったとしても、Gradleでそのデータを使うた

51.1.2. Javaにおける依存関係管理

Javaはその豊富なオープンソースライブラリで知られる言語ですが、皮肉なことにライブラリやバージョ
Javaでは、JVMに、今使っているものがHibernateのバージョン3.0.5であると伝える標準的な方法はあり
`foo-1.0.jar`が`bar-2.0.jar`に依存していると伝える方法も標準では存在しないのです。
このため、これを実現するための様々なソリューションがサードパーティにより考案されてきました。ま

どちらのツールも、ある特定のjarファイルについて、その依存関係の情報をXML形式のディスクリプタ
どちらも依存関係を解決するための標準的な方法となっており、Gradleも最初は依存関係管理機能の水面
Gradleは、今ではそのように直接Ivyに依存するのではなく、Gradle独自の依存関係解決エンジンを使用

51.2. 依存関係管理のベストプラクティス

Gradleは依存関係の管理についてとても強固な指針を持ってはいますが、ユーザーには二つのオプション
この節では、Gradle
Projectが推奨する依存関係管理のベストプラクティスについて概要を記載します。

開発言語にかかわらず、依存関係を正しく管理することはあらゆるプロジェクトで大事なファクターです
数百ものオープンソースライブラリに依存した複雑なエンタープライズJavaアプリケーションから、わず
例えば、再配布して使用されることを想定したライブラリプロジェクトの場合、巨大なソフトウェアやイ
Projectでは全てのプロジェクトで以下の指針に従うことを推奨しています。

51.2.1. ファイル名にバージョン番号を含める(jarのバージョニング)

ライブラリのバージョンはファイル名で簡単に分かるようになっていなければなりません。jarのバージョ
もし、20個のjarファイルがズラッと並べられて、これを見てほしいと言われたら、どちらがいいと思い
`commons-beanutils-1.3.jar`のようなファイル群と、`spring.jar`
のようなファイル群と、どちらの方がわかりやすいと思いますか？
ファイル名にバージョン番号が付いていれば、プロジェクトが依存しているjarのバージョンを簡単に、

不明瞭なバージョンは、発見するのが非常に難しい、微妙なバグに繋がります。
例えば、Hibernate2.5を使っているプロジェクトがあるとしましょう。Hibernate3.0.5にアップグレー
数週間後、インテグレーションサーバーで例外が発生します。しかし他のマシンでは再現しません。結果

jarのファイル名にバージョン付けすることは、プロジェクトの表現性を高め、メンテナンスを簡単にしま

51.2.2. 推移的な依存関係の管理

推移的な依存関係の管理とは、プロジェクトがあるライブラリに依存しているとき、その依存ライブラリ
このように推移的な依存関係が再帰的に発生すると、プロジェクトが直接依存する第一層の依存、それか
階層的にモデリングされていない依存関係は、構造化されないまま複雑にふくれあがり、簡単に制御を失
Gradle自身のことを考えてみても、Gradleプロジェクト自体は直接の、つまり第一層の依存関係はそう多
さらに規模が大きくなれば、例えばSpring、Hibernate、その他のライブラリを使うエンタープライズプ

このように依存関係ツリーが巨大になってくると、それを修正する際、バージョンの競合が発生しやすく
例えば、あるバージョンのロギングライブラリを必要としているオープンソースライブラリがあるとしま

これらの問題を、手で何とかしようと頑張ることはできるかもしれませんが、すぐにそうした方法には拡
例えば、第一層、直接の依存関係を取り除きたいとなったとき、関連する他のjarも削除して良いものかと
削除しようとする依存関係の依存ライブラリだと思っていたjarが、それ自身第一層の依存関係に含まれて
自分で、手で推移的な依存関係を管理しようとすれば、その行き着く先は壊れやすく脆いビルドです。
思い切ってそのプロジェクトの依存関係を変更しようなどという人はどこにもいません。ビルドを壊して
プロジェクトのクラスパスはいつか完全にめちゃくちゃになって、ひとたびそのクラスパスで問題が発生

ノート:

あるプロジェクトで、私たちはミステリーに遭遇しました。クラスパスにあるLDAP関連のjarなのです
この謎のjarは、推移的な、第四層の依存関係に当たるものでした。推移的な依存関係を管理してやろ

Gradleは、直接の依存関係および推移的な依存関係を表現するための手段を複数用意しています。複数の

51.2.3. バージョン競合の解決

同じjar間でバージョンの競合が検出された場合、それを解決するか、エラーを発生させなければなりませ
たくさんの開発者が依存関係を変更しているような巨大プロジェクトの場合、依存関係の順序がビルドの

もしクラスパスのバージョンが競合するという不幸に苦しんだことがないようでしたら、あなたを待ち受
30のサブモジュールからなるある巨大なプロジェクトがありました。ここにあるサブモジュールに依存関
ビルドはまだ正常に動き続けているのに、開発者たちは無数の驚くような、というか驚くくらいひどいバ
さらに悪いことに、この意図しないSpringのダウングレードはいくつかのセキュリティ的な脆弱性をシス

要するに、バージョンの競合は悪いことなので、それを避けるために推移的な依存関係は管理しなければ
また、競合したバージョンが組織全体で何処で使われているのか調べ、一つのバージョンに統一したいと
もしバージョン競合など自分には起こるわけがないと思っているようでしたら、どうかもう一度考え直し
第一層の依存関係が、更に別のライブラリに依存していて、それが他の依存関係とバージョンが競合して
また、JVMには同じjarの違うバージョンをクラスパスで共存させられるような簡単な方法がありません(
「Javaにおける依存関係管理 」参照)。

Gradleでは、バージョン競合を解決するための戦略として以下のものを用意しています。

- Newest:
  もっとも新しいバージョンの依存関係を使用します。これはGradleのデフォルトの戦略で、バージョ
- Fail:
  バージョンが競合した場合、ビルドを失敗させます。この戦略は、ビルドスクリプトで全てのバージ
  特定のバージョンを明示的に選択する方法については`ResolutionStrategy`
  を参照してください。

上記の戦略で大抵の場合はほとんどの競合を解決できますが、Gradleはさらに微調整を行うためのメカニ

- プロジェクトが直接依存する第一層の依存関係をforcedとして強制できます。
  これは、競合した依存関係が第一層の依存関係だったときに便利なアプローチです。
  例については`DependencyHandler`を参照してください。
- 依存関係は、第一層のものか推移的なものかにかかわらずとにかくforced
  として強制することもできます。
  これは、推移的な依存関係で競合が発生してしまったときに便利なアプローチです。
  また、第一層の依存関係も、前述の方法でなくこちらの方法で強制させることもできます。
  例については`ResolutionStrategy`を参照してください。
- 依存関係解決ルールというGradle1.4で導入された試験的な
  機能があり、特定の依存関係において選択されるバージョンを細かく調整することができます。

バージョン競合により発生する様々な問題に対応するときには、依存関係グラフを表示するレポートも非

51.2.4. 動的バージョンと変更性モジュールを使う

あるモジュールについて、常に最新版を使いたい、またはある範囲においての最新版を使いたいといった
例えば、開発作業中や、ある範囲のバージョンのライブラリと協調して動作するよう設計されたライブラ
この種の、絶えず変更されるような依存関係は、動的バージョン
を使うと簡単に参照することができます。 動的バージョンは、バージョンの範囲で指定すること(`2.+`
など)も、使用できる最新版を表すプレースホルダで指定すること(`latest.integration`
など)もできます。

また、要求しているモジュールのバージョンそのものは変わらないのに、中身だけが変更されていく場合
変更性モジュールの典型的な例は、Mavenの`SNAPSHOT`
モジュールです。このモジュールは、公開された最新のモジュールを常にポイントし続けます。
言い換えれば、標準的なMavenのスナップショットは静的な状態にとどまっているものではありません。

動的バージョンと変更性モジュールの主な違いは、モジュール名に付加されるバージョンです。
動的バージョンが静的な、実際のバージョンに変換されて付加されるのに対し、変更性モジュール
は要求したバージョン名がそのまま付加されます。その上で、モジュールの中身が時間が経つにつれて変

デフォルトでは、Gradleは動的バージョンと変更性モジュールのキャッシュを24時間保持します。デフ
コマンドラインオプションで変更できます。 また、ビルド時の実際のキャッシュの有効期限は
を使って設定できます(「依存関係キャッシュ制御の微調整 」参照)。

51.3. 依存関係のコンフィギュレーション

Gradleでは、依存関係は「コンフィギュレーション」によりグループ化されます。コンフィギュレーシ
Gradleプラグインの多くが、独自のコンフィグレーションを事前に定義してプロジェクトに追加します。
たとえば、Javaプラグインは、いくつかのコンフィグレーションをプロジェクトに追加して、必要とされ
「依存関係の管理」をご参照ください。
もちろん、定義済みのものだけでなく、自分でコンフィギュレーションを定義して追加することも可能で

プロジェクトのコンフィギュレーションは`configurations`
オブジェクトで管理されています。このオブジェクトに渡したクロージャは、configurationsのAPIに対し
APIの詳細については`ConfigurationContainer`をご参照ください。

コンフィギュレーションを定義するには、

例51.1 コンフィギュレーションの定義

build.gradle

```
configurations {
    compile
}
```

コンフィギュレーションにアクセスするには、

例51.2 コンフィギュレーションへのアクセス

build.gradle

```
println configurations.compile.name
println configurations['compile'].name
```

コンフィギュレーションの設定を変更するには、

例51.3 コンフィギュレーションの設定変更

build.gradle

```
configurations {
    compile {
        description = 'compile classpath'
        transitive = true
    }
    runtime {
        extendsFrom compile
    }
}
configurations.compile {
    description = 'compile classpath'
}
```

51.4. 依存関係の定義方法

定義できる依存関係にはいくつかの種類があります。

表51.1 依存関係の種類

種類	説明
外部モジュール依存関係	どこかのリポジトリにある外部モジュールへの依存関係
プロジェクト依存関係	同じビルドに含まれる別プロジェクトへの依存関係
ファイル依存関係	ローカルファイルシステム上のファイル群への依存関係
クライアントモジュール依存関係	どこかのリポジトリにあるアーティファクトへ依存しているが、
Gradle APIへの依存関係	現在使用しているバージョンのGradleが持つAPIへの依存関係。G
ローカルGroovy依存関係	現在使用中のGradleで使われているGroovyへの依存関係。これも

51.4.1. 外部モジュール依存関係

外部モジュール依存関係は、もっともポピュラーな種類の依存関係です。このタイプの依存関係は、外部

例51.4 モジュール依存関係

build.gradle

```
dependencies {
    runtime group: 'org.springframework', name: 'spring-core', version: '2.5'
    runtime 'org.springframework:spring-core:2.5',
            'org.springframework:spring-aop:2.5'
    runtime(
        [group: 'org.springframework', name: 'spring-core', version: '2.5'],
        [group: 'org.springframework', name: 'spring-aop', version: '2.5']
    )
    runtime('org.hibernate:hibernate:3.0.5') {
        transitive = true
    }
    runtime group: 'org.hibernate', name: 'hibernate', version: '3.0.5', transitive
    runtime(group: 'org.hibernate', name: 'hibernate', version: '3.0.5') {
        transitive = true
    }
}
```

より多くの例と完全なリファレンスは`DependencyHandler`を参照してください。

Gradleはモジュール依存関係を定義するために二つの記法を用意しています。文字列記法とマップ記法て
`ExternalModuleDependency`も併せてご覧ください。
このAPIはプロパティとメソッドを公開しており、文字列記法ではプロパティのうち一部を指定すること
マップ記法を使うと全てのプロパティにアクセスできます。全てのAPIにアクセスするには、文字列記法

モジュール依存関係が宣言されていると、Gradleは対応するモジュールディスクリプタファイル(`pom.xı`
または`ivy.xml`
)をリポジトリに探しに行きます。結果、ディスクリプタファイルが見つかった場合は、それをパースし、
`hibernate-3.0.5.jar`など)および推移的依存関係(cglibなど)をダウンロードします。
ディスクリプタファイルが見つからなければ、Gradleは今度は`hibernate-3.0.5.jar`
という名前のファイルを探して取得しようとします。
Mavenのモジュールは、アーティファクトを一つしか持つことができません。GradleとIvyは一つのモジ

51.4.1.1. 複数のアーティファクトを持つモジュールへの依存関係

前述のように、Mavenのモジュールはアーティファクトを一つしか持てません。そのため、プロジェクト
しかし、GradleやIvyの場合は違います。Ivyの依存関係モデル(`ivy.xml`
)では、一つのモジュールに複数のアーティファクトを宣言できるからです。 詳細については`ivy.xml`
のリファレンスを参照してください。
Gradleでivyモジュールへ依存していることを宣言したとき、実際にはそのモジュールの`default`
コンフィギュレーションへの依存関係を宣言しています。
つまり、プロジェクトが実際に依存しているアーティファクト(典型的にはjar)は、モジュールの`default`
コンフィギュレーションに格納された全アーティファクトです。
このことが重要になるのは、以下のような場合です。

- `default`コンフィグレーションに、不必要なアーティファクトが含まれている場合。この場合、`def`
  コンフィグレーション全てに依存するのではなく、必要なアーティファクトへの依存関係のみを宣言
- 必要なアーティファクトが`default`
  コンフィグレーション以外のコンフィグレーションに含まれている場合。この場合、そのコンフィグ

依存関係の宣言を微調整しなければならないようなシチュエーションはまだまだあります。
その他の例や依存関係宣言の完全なリファレンスについては`DependencyHandler`
をご覧ください。

51.4.1.2. アーティファクトオンリー記法

上記のように、モジュールのディスクリプタファイルが見つからなかった場合、Gradleはデフォルトでモ
しかし、リポジトリにディスクリプタファイルがあるアーティファクトでも、場合によってはjarのみを少
[22]
その他にも、ディスクリプタファイルを持たないリポジトリからzipファイルをダウンロードしたいとい
アーティファクトオンリー
記法というものをサポートしています。書き方は簡単で、ダウンロードしたいものの拡張子を「@」で指

例51.5 アーティファクトオンリー記法

build.gradle

```
dependencies {
    runtime "org.groovy:groovy:2.2.0@jar"
    runtime group: 'org.groovy', name: 'groovy', version: '2.2.0', ext: 'jar'
}
```

アーティファクトオンリー記法は、指定した拡張子のファイルのみをダウンロードする、というモジュー

51.4.1.3. 分類子(Classifiers)

Mavenには分類子(classifiers)という記法があります。[23]
Gradleでもこの記法をサポートしています。Mavenリポジトリから分類子つきの依存関係を取得するには

例51.6 分類子付きの依存関係

build.gradle

```
compile "org.gradle.test.classifiers:service:1.0:jdk15@jar"
    otherConf group: 'org.gradle.test.classifiers', name: 'service', version: '1.0'
```

上記の一行目にあるように、分類子は明示的に拡張子を指定して(アーティファクトオンリー記法で)使う

コンフィグレーションに含まれる依存アーティファクトの一覧は簡単に列挙できます。

例51.7 あるコンフィグレーションの内部を列挙する

build.gradle

```
task listJars << {
    configurations.compile.each { File file -> println file.name }
}
```

gradle -q listJars の出力

```
> gradle -q listJars
hibernate-core-3.6.7.Final.jar
antlr-2.7.6.jar
commons-collections-3.1.jar
dom4j-1.6.1.jar
hibernate-commons-annotations-3.2.0.Final.jar
hibernate-jpa-2.0-api-1.0.1.Final.jar
jta-1.1.jar
slf4j-api-1.6.1.jar
```

51.4.2. クライアントモジュール依存関係

クライアントモジュール依存関係を使うと、モジュールの推移的な
依存関係をビルドスクリプトの中で直接定義できます。これは外部リポジトリにある実際のモジュールテ

例51.8 クライアントモジュール依存関係 - 推移的な依存関係

build.gradle

```
dependencies {
    runtime module("org.codehaus.groovy:groovy:2.3.6") {
        dependency("commons-cli:commons-cli:1.0") {
            transitive = false
        }
        module(group: 'org.apache.ant', name: 'ant', version: '1.9.3') {
            dependencies "org.apache.ant:ant-launcher:1.9.3@jar",
                         "org.apache.ant:ant-junit:1.9.3"
        }
    }
}
```

この例では、Groovyに依存していることが宣言されています。Groovyは、それ自身別の依存関係を持っ
クライアントモジュールの依存関係としては、普通のモジュール依存関係の他、アーティファクトオンリ
`ClientModule`も参考にしてください。

現在、クライアントモジュールには一つの制限があります。プロジェクトがライブラリをビルドするもの

51.4.3. プロジェクト依存関係

Gradleでは、外部モジュール依存関係と、マルチプロジェクト内で別のプロジェクトを参照する依存関係
後者はプロジェクト依存関係として宣言します。

例51.9 プロジェクト依存関係

build.gradle

```
dependencies {
    compile project(':shared')
}
```

APIドキュメントProjectDependencyに詳しい情報が記載されています。

また、マルチプロジェクトについては、57章マルチプロジェクトのビルドで詳述されています。

51.4.4. ファイル依存関係

ファイル依存関係は、前もってファイルをリポジトリに格納することなく、直接コンフィギュレーション

方 法 も 簡 単 で 、 単 に フ ァ イ ル コ レ ク シ ョ ン
を依存関係としてコンフィギュレーションに渡すだけです。

例51.10 ファイル依存関係

build.gradle

```
dependencies {
    runtime files('libs/a.jar', 'libs/b.jar')
    runtime fileTree(dir: 'libs', include: '*.jar')
}
```

ファイル依存関係は、公開されるディスクリプタファイルには含まれません。しかし、同じビルド内から
つまり、ファイル依存関係は、そのビルドの外部からは使用されませんが、同じビルド内のプロジェクト

ファイル依存関係では、どのタスクがファイルを生成するのか宣言することができます。依存ファイルか

例51.11 生成されるファイルへの依存関係

build.gradle

```
dependencies {
    compile files("$buildDir/classes") {
        builtBy 'compile'
    }
}

task compile << {
    println 'compiling classes'
}

task list(dependsOn: configurations.compile) << {
    println "classpath = ${configurations.compile.collect {File file -> file.name}}"
}
```

gradle -q list の出力

```
> gradle -q list
compiling classes
classpath = [classes]
```

51.4.5. Gradle API依存関係

`DependencyHandler.gradleApi()`
メソッドを使うと、そのビルドで使用しているGradleのAPIへ依存していることを宣言できます。
Gradleのカスタムタスクやプラグインを開発しているときによく使われます。

例51.12 Gradle API依存関係

build.gradle

```
dependencies {
    compile gradleApi()
}
```

51.4.6. ローカルGroovy依存関係

`DependencyHandler.localGroovy()`
を使うと、Gradleに同梱されているGroovyへの依存関係を宣言できます。
これもGradleのカスタムタスクやプラグインを開発しているときによく使われます。

例51.13 Gradleに同梱されているGroovyへの依存関係

build.gradle

```
dependencies {
    compile localGroovy()
}
```

51.4.7. 推移的な依存関係の除外

コンフィギュレーション単位、または依存関係単位で、特定の推移的な依存関係を除外することができま

例51.14 推移的な依存関係の除外

build.gradle

```
configurations {
    compile.exclude module: 'commons'
    all*.exclude group: 'org.gradle.test.excludes', module: 'reports'
}

dependencies {
    compile("org.gradle.test.excludes:api:1.0") {
        exclude module: 'shared'
    }
}
```

コンフィギュレーションに対してある依存関係を除外すると、そのコンフィギュレーション、および全て
また、全てのコンフィギュレーションにまとめて除外設定を適用したい場合は、例にあるようにGroovy
除外設定は、organizationのみを指定することもできますし、モジュール名やその両方で指定しても構い
APIドキュメントの`Dependency`や`Configuration`を参照してみてください。

全ての推移的な依存関係が除外できるとは限りません。アプリケーションの動作に必須な依存関係もあり
一般的には、実行時に使われることがないものや、対象環境やプラットフォームでそれが使えると保証て

除外設定は、依存関係単位かコンフィギュレーション単位、どちらで設定するべきでしょうか？
ほとんどの場合は、コンフィギュレーション単位で設定することになるでしょう。
推移的な依存関係を除外したいと考える、代表的な理由を挙げてみます。
なお、これらの中には、依存関係を除外してしまうよりも優れた解決手段があるものもあります。頭に留

- ライセンス上の理由により、その依存関係を除外したい
- いずれかのリモートリポジトリで、その依存関係が使用できない
- 実行時にその依存関係が不要
- その依存関係のバージョンが競合していて欲しいバージョンの依存関係が取得できない。この場合、「バージョン競合の解決 」やResolutionStrategyを参照してみてください。

基本的には、ほとんどのケースでコンフィギュレーション単位の除外設定を使うべきです。これにより、
例えば、除外ルールを適用していない別の依存関係が、除外したはずの依存関係を引っ張ってきてしまう

リファレンスには、除外設定の他の例も記載されています。ModuleDependencyや
DependencyHandlerを見てみてください。

51.4.8. オプション属性

依存関係宣言の属性は、nameを除いて全てオプションです。ただ、実際にリポジトリから依存関係を見
「リポジトリ 」を参照してください。
例えば、Mavenリポジトリを使うときは、組織名、モジュール名、およびバージョンを指定する必要があ

例51.15 依存関係のオプション属性

build.gradle

```
dependencies {
    runtime ":junit:4.10", ":testng"
    runtime name: 'testng'
}
```

また、依存関係定義のコレクションや属性をコンフィギュレーションに割り当てることもできます。

例51.16 依存関係定義のコレクション、配列

build.gradle

```
List groovy = ["org.codehaus.groovy:groovy-all:2.3.6@jar",
               "commons-cli:commons-cli:1.0@jar",
               "org.apache.ant:ant:1.9.3@jar"]
List hibernate = ['org.hibernate:hibernate:3.0.5@jar',
                  'somegroup:someorg:1.0@jar']
dependencies {
    runtime groovy, hibernate
}
```

51.4.9. 依存するコンフィギュレーションの指定

Gradleでは、依存先のモジュールが複数のコンフィギュレーションを公開していることがあります(自分(
特に明示しないかぎり、Gradleはその依存関係のdefaultコンフィギュレーションを使用します。
Mavenリポジトリではdefaultコンフィギュレーションが使用できる唯一のコンフィギュレーションなので

例51.17 依存するコンフィギュレーションの指定

build.gradle

```
dependencies {
    runtime group: 'org.somegroup', name: 'somedependency', version: '1.0', configu
}
```

プロジェクト依存関係で同じことをするには、次のように定義する必要があります。

例51.18 依存するプロジェクトのコンフィギュレーション

build.gradle

```
dependencies {
    compile project(path: ':api', configuration: 'spi')
}
```

51.4.10. 依存関係のレポート

Gradleには、依存関係に関するレポートを生成できるコマンドが用意されています(
「プロジェクトの依存関係一覧」参照)。 また、プロジェクトレポートプラグイン(41章
プロジェクトレポートプラグイン
)を使えば、ビルドのタスクでそのレポートを作成することもできます。

`ResolvableDependencies.getResolutionResult()`. Potential usages of the `ResolutionResult` API:
Gradle1.2からは、解決した依存関係に関する情報に、プログラム的にアクセスできるAPIもあります。
先ほど述べた依存関係のレポートを作成するときも、水面下でこのAPIが使用されています。
このAPIを使えば、解決した依存関係グラフを走査したり、情報を取得したりすることが可能です。
また、近くこのAPIを更に拡張し、解決結果についてもっと情報を取得できるようにする予定です。
APIについての詳細はJavadoc(`ResolvableDependencies.getResolutionResult()`
)を参照してください。 API(`ResolutionResult`)の使い道として考えられるのは、

- 自分のユースケースに適した、もっと踏み込んだレポートを作成するため
- 依存関係グラフの中身に応じて処理を切り替えるようなビルドロジックを実装するため

51.5. 依存関係を使った作業

例えば、次のような依存関係をセットアップしたとします。

例51.19 Configuration.copy

build.gradle

```
configurations {
    sealife
    alllife
}

dependencies {
    sealife "sea.mammals:orca:1.0", "sea.fish:shark:1.0", "sea.fish:tuna:1.0"
    alllife configurations.sealife
    alllife "air.birds:albatross:1.0"
}
```

このセットアップにおける推移的な依存関係は、以下の通りです。

shark-1.0 -> seal-2.0, tuna-1.0

orca-1.0 -> seal-1.0

tuna-1.0 -> herring-1.0

このコンフィギュレーションを使って、宣言された依存関係やその一部にアクセスできます。

例51.20 宣言した依存関係にアクセスする

build.gradle

```
task dependencies << {
    configurations.alllife.dependencies.each { dep -> println dep.name }
    println()
    configurations.alllife.allDependencies.each { dep -> println dep.name }
    println()
    configurations.alllife.allDependencies.findAll { dep -> dep.name != 'orca' }
        .each { dep -> println dep.name }
}
```

gradle -q dependencies の出力

```
> gradle -q dependencies
albatross

albatross
orca
shark
tuna

albatross
shark
tuna
```

`dependencies`はコンフィギュレーションに明示的に属しているもののみを返します。`allDependencies`は継承したコンフィギュレーションに含まれるものも返します。

コンフィギュレーションの依存関係として設定されているライブラリファイルを取得するには、次のよう

例51.21 Configuration.files

build.gradle

```
task allFiles << {
    configurations.sealife.files.each { file ->
        println file.name
    }
}
```

gradle -q allFiles の出力

```
> gradle -q allFiles
orca-1.0.jar
shark-1.0.jar
tuna-1.0.jar
herring-1.0.jar
seal-2.0.jar
```

コンフィギュレーションに属する依存関係のうち、一部の依存関係のみ、例えば一つの依存関係のみライ

例51.22 フィルター付きConfiguration.files

build.gradle

```
task files << {
    configurations.sealife.files { dep -> dep.name == 'orca' }.each { file ->
        println file.name
    }
}
```

gradle -q files の出力

```
> gradle -q files
orca-1.0.jar
seal-2.0.jar
```

`Configuration.files`メソッドは、常にそのコンフィギュレーション全体
から全てのアーティファクトを返すので、ここでは依存関係を指定してフィルターしました。例にあるよ

コンフィギュレーションはコピーすることもできます。さらに、元のコンフィギュレーションから一部の
コピー用のメソッドは二つ用意されており、`copy`
メソッドは明示的にコンフィギュレーションに属している依存関係のみコピーします。もう一つの
`copyRecursive`
メソッドは継承元のコンフィギュレーションに含まれる依存関係もすべてコピーします。

例51.23 Configuration.copy

build.gradle

```
task copy << {
    configurations.alllife.copyRecursive { dep -> dep.name != 'orca' }
        .allDependencies.each { dep -> println dep.name }
    println()
    configurations.alllife.copy().allDependencies
        .each { dep -> println dep.name }
}
```

gradle -q copy の出力

```
> gradle -q copy
albatross
shark
tuna

albatross
```

注意するべきなのは、コピーされたコンフィギュレーションから返される依存関係が、常にオリジナルの
「コピーされたサブセット」と、「コピーされなかった残りのサブセット」の間でバージョンの競合があ

例51.24 Configuration.copy vs. Configuration.files

build.gradle

```
task copyVsFiles << {
    configurations.sealife.copyRecursive { dep -> dep.name == 'orca' }
        .each { file -> println file.name }
    println()
    configurations.sealife.files { dep -> dep.name == 'orca' }
        .each { file -> println file.name }
}
```

gradle -q copyVsFiles の出力

```
> gradle -q copyVsFiles
orca-1.0.jar
seal-1.0.jar

orca-1.0.jar
seal-2.0.jar
```

上の例では、`orca`が`seal-1.0`に依存しており、`shark`は`seal-2.0`
に依存しています。なので、オリジナルのコンフィギュレーションではバージョンの競合が発生している
そのため、この依存関係はより新しいバージョンである`seal-2.0`
に解決され、推移的な依存関係として`seal-2.0`が表示されています。
一方、コピー先のコンフィギュレーションには`orca`
しか含まれていません。そのためバージョンの競合は発生しておらず、推移的な依存関係として
`seal-1.0`が表示されているのです。

一度解決されたコンフィギュレーションは不変(immutable)です。
その状態を変更したり、コンフィギュレーション内の依存関係の状態を変更したりすると例外が発生しま

ConfigurationクラスのAPIをさらに詳しく知りたいときは、APIドキュメント(`Configuration`
)をご参照ください。

51.6. リポジトリ

G r a d l e の リ ポ ジ ト リ 管 理 機 能 は 、 A p a c h e
Ivyをベースにしたもので、リポジトリのレイアウトや参照方法を自在に定義できます。さらに、Gradle

リポジトリはいくつでも定義できますが、Gradleからはそれぞれが独立して取り扱われます。
あるリポジトリでモジュールのディスクリプタが見つかれば、Gradleはそのモジュールのアーティファク
同 じ リ ポ ジ ト リ か ら ダ ウ ン ロ ー ド し よ う と し ま す 。
つまり、モジュールのメタデータと全てのアーティファクトは同じリポジトリになければなりません。し

宣言できるリポジトリには、以下のものがあります。

表51.2 リポジトリの種類

種類	説明
Mavenセントラルリポジトリ	Mavenセントラルにある依存関係を探すための定義済みリポジト
Maven JCenterリポジトリ	BintrayのJCenterにある依存関係を探すための定義済みリポジト
Mavenローカルリポジトリ	Mavenのローカルリポジトリにある依存関係を探すための定義済
Mavenリポジトリ	Mavenリポジトリ。ローカルファイルシステムまたはリモートに
Ivyリポジトリ	Ivyリポジトリ。ローカルファイルシステムまたはリモートに配置
フラットディレクトリリポジトリ	ローカルファイルシステム上のシンプルなリポジトリ。メタデー

51.6.1. Supported repository transport protocols

Maven and Ivy repositories support the use of various transport protocols. At the moment the following protocols are supported:

表51.3 Repository transport protocols

Type	Authentication schemes
`file`	none
`http`	username/password
`https`	username/password
`sftp`	username/password

To define a repository use the `repositories` configuration block. Within the `repositories` closure, a Maven repository is declared with `maven`. An Ivy repository is declared with `ivy`. The transport protocol is part of the URL definition for a repository. The following build script demonstrates how to create a HTTP-based Maven and Ivy repository:

例51.25 Declaring a Maven and Ivy repository

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }

    ivy {
        url "http://repo.mycompany.com/repo"
    }
}
```

If authentication is required for a repository, the relevant credentials can be provided. The following example shows how to provide username/password-based authentication for SFTP repositories:

例51.26 Providing credentials to a Maven and Ivy repository

build.gradle

```
repositories {
    maven {
        url "sftp://repo.mycompany.com:22/maven2"
        credentials {
            username 'user'
            password 'password'
        }
    }

    ivy {
        url "sftp://repo.mycompany.com:22/repo"
        credentials {
            username 'user'
            password 'password'
        }
    }
}
```

51.6.2. Mavenセントラルリポジトリ

Maven2のセントラルリポジトリ(http://repo1.maven.org/maven2
)は、次のように簡単に追加できます。

例51.27 Mavenセントラルリポジトリを追加する

build.gradle

```
repositories {
    mavenCentral()
}
```

Warning: Be aware that the central Maven 2 repository is HTTP only and HTTPS is not supported. If you need a public HTTPS enabled central repository, you can use the JCenter public repository (see 「Maven JCenterリポジトリ」).

これで、Gradleはこのリポジトリから依存関係を探すようになります。

51.6.3. Maven JCenterリポジトリ

B i n t r a y
のJCenterには、Bintrayに直接公開されたアーティファクトだけでなく、ポピュラーなMaven上のOSSラ

JCenter Mavenリポジトリ(https://jcenter.bintray.com)は、次のように簡単に追加できます。

例51.28 BintrayのJCenter Mavenリポジトリを追加する

build.gradle

```
repositories {
    jcenter()
}
```

これで、GradleはJCenterリポジトリから依存関係を探すようになります。 jcenter() uses HTTPS to connect to the repository. If you want to use HTTP you can configure `jcenter()`:

例51.29 Using Bintrays's JCenter with HTTP

build.gradle

```
repositories {
    jcenter {
        url "http://jcenter.bintray.com/"
    }
}
```

51.6.4. ローカルMavenリポジトリ

Mavenのローカルキャッシュをリポジトリとして使うには、次のようにします。

例51.30 Mavenのローカルキャッシュをリポジトリとして追加する

build.gradle

```
repositories {
    mavenLocal()
}
```

Gradleは、Mavenが使っているのと同じロジックでローカルキャッシュの場所を決定します。
もしローカルリポジトリの場所が`settings.xml`で定義されていれば、そこが使用されます。 *USER_HC*
の`settings.xml`は、*M2_HOME*`/conf`の`settings.xml`よりも優先されます。 使用できる`settings.:`
がなければ、デフォルトの*USER_HOME*`/.m2/repository`
がローカルキャッシュの場所になります。

51.6.5. Mavenリポジトリ

カスタムのMavenリポジトリを追加するには、次のようにします。

例51.31 カスタムMavenリポジトリを追加する

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }
}
```

POMとJARなどのアーティファクトを別々の場所に配置するようなリポジトリもあります。このようなリ

例51.32 JARファイル用の追加リポジトリを設定する

build.gradle

```
repositories {
    maven {
        // Look for POMs and artifacts, such as JARs, here
        url "http://repo2.mycompany.com/maven2"
        // Look for artifacts here if not found at the above location
        artifactUrls "http://repo.mycompany.com/jars"
        artifactUrls "http://repo.mycompany.com/jars2"
    }
}
```

Gradleは、最初のURLを、POMとJARを探すために使用します。JARがそこで見つからなかった場合は、

51.6.5.1. パスワードで保護されたMavenリポジトリへのアクセス

BASIC認証で保護されたMavenリポジトリにアクセスするには、リポジトリ定義でユーザー名とパスワー

例51.33 パスワード保護されたMavenリポジトリへのアクセス

build.gradle

```
repositories {
    maven {
        credentials {
            username 'user'
            password 'password'
        }
        url "http://repo.mycompany.com/maven2"
    }
}
```

ユーザー名、パスワードは、ビルドファイルに直接指定するのではなく、`gradle.properties` に外出しすることを推奨します。

51.6.6. フラットディレクトリリポジトリ

ファイルシステム上の(フラットな)ディレクトリをリポジトリとして使用したい場合は、単に次のように

例51.34 フラットディレクトリ・リゾルバ

build.gradle

```
repositories {
    flatDir {
        dirs 'lib'
    }
    flatDir {
        dirs 'lib1', 'lib2'
    }
}
```

これで、依存関係を取得するリポジトリとして複数のディレクトリが追加されました。
このフラットディレクトリ・リゾルバのみ使うのであれば、依存関係定義ですべての属性を定義する必要
「オプション属性 」をご参照ください。

51.6.7. Ivyリポジトリ

51.6.7.1. Defining an Ivy repository with a standard layout

例51.35 Ivyリポジトリ

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
    }
}
```

51.6.7.2. Defining a named layout for an Ivy repository

You can specify that your repository conforms to the Ivy or Maven default layout by using a named layout.

例51.36 Ivy repository with named layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "maven"
    }
}
```

Valid named layout values are `'gradle'` (the default), `'maven'` and `'ivy'`. See `IvyArtifactRepository.layout()` in the API documentation for details of these named layouts.

51.6.7.3. Ivyリポジトリのカスタムパターンを定義する

標準的でないレイアウトのIvyリポジトリを使うには、リポジトリのパターンレイアウトを定義します。

例51.37 パターンレイアウトを指定したIvyリポジトリ

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "pattern", {
            artifact "[module]/[revision]/[type]/[artifact].[ext]"
        }
    }
}
```

51.6.7.4. Maven互換レイアウトのIvyリポジトリ

オプションとして、パターンレイアウトを定義する際、「organisation」部分をMavenスタイルに展開し
例えば、`my.company`という組織名は`my/company`と表現されます。

例51.38 Maven互換レイアウトのIvyリポジトリ

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "pattern", {
            artifact "[organisation]/[module]/[revision]/[artifact]-[revision].[ext]"
            m2compatible = true
        }
    }
}
```

Valid named layout values are `'gradle'` (the default), `'maven'` and `'ivy'`. See `IvyArtifactRepository.layout()` in the API documentation for details of these named layouts.

51.6.7.5. アーティファクトとivyファイルの場所が異なるIvyリポジトリの設定

例51.39 Ivy repository with pattern layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "pattern", {
            artifact "[module]/[revision]/[type]/[artifact].[ext]"
        }
    }
}
```

To define an Ivy repository which fetches Ivy files and artifacts from different locations, you can define separate patterns to use to locate the Ivy files and artifacts:

Each `artifact` or `ivy` specified for a repository adds an additional pattern to use. The patterns are used in the order that they are defined.

例51.40 Ivy repository with multiple custom patterns

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "pattern", {
            artifact "3rd-party-artifacts/[organisation]/[module]/[revision]/[artifac
            artifact "company-artifacts/[organisation]/[module]/[revision]/[artifact]
            ivy "ivy-files/[organisation]/[module]/[revision]/ivy.xml"
        }
    }
}
```

Optionally, a repository with pattern layout can have its 'organisation' part laid out in Maven style, with forward slashes replacing dots as separators. For example, the organisation `my.company` would then be represented as `my/company`.

例51.41 Ivy repository with Maven compatible layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "pattern", {
            artifact "[organisation]/[module]/[revision]/[artifact]-[revision].[ext]"
            m2compatible = true
        }
    }
}
```

51.6.7.6. パスワードで保護されたIvyリポジトリへのアクセス

BASIC認証で保護されたIvyリポジトリへアクセスするには、リポジトリ定義の際にユーザー名とパスワー

例51.42 パスワードで保護されたIvyリポジトリ

build.gradle

```
repositories {
    ivy {
        url 'http://repo.mycompany.com'
        credentials {
            username 'user'
            password 'password'
        }
    }
}
```

51.6.8. リポジトリを使った作業

リポジトリ定義にアクセスするには、

例51.43 リポジトリ定義へのアクセス

build.gradle

```
println repositories.localRepository.name
    println repositories['localRepository'].name
```

リポジトリの設定を変更するには、

例51.44 リポジトリの設定変更

build.gradle

```
repositories {
    flatDir {
        name 'localRepository'
    }
}
repositories {
    localRepository {
        dirs 'lib'
    }
}
repositories.localRepository {
    dirs 'lib'
}
```

51.6.9. Ivyリゾルバについて

Gradleは、水面下でIvyを使っているおかげで、リポジトリに関しては非常に高い柔軟性を持っています。

- リポジトリとの通信プロトコル(ファイルシステム、http、ssh、等々)がたくさん用意されている
- The protocol sftp currently only supports username/password-based authentication.
- 一つ一つのリポジトリが、それぞれ独自のレイアウトを持つことができる

`junit:junit:3.8.2`
へ依存していることを定義したとしましょう。このとき、Gradleはどのようにしてリポジトリからjunitを
[24]

```
// Maven2  (Maven2())
someroot/[organisation]/[module]/[revision]/[module]-[revision].[ext]

// Ivy()
someroot/[organisation]/[module]/[revision]/[type]s/[artifact].[ext]

// ()
someroot/[artifact]-[revision].[ext]
```

上記のどのリポジトリも、以下のようにして参照できます。

例51.45 カスタムリポジトリの定義

build.gradle

```
repositories {
    ivy {
        ivyPattern "$projectDir/repo/[organisation]/[module]-ivy-[revision].xml"
        artifactPattern "$projectDir/repo/[organisation]/[module]-[revision](-[classi
    }
}
```

Ivyで(つまりGradleで)どんなリゾルバを使えるかはここ
で見ることができます。Gradleでは、XMLを経由することなく直接これらのAPIを操作できます。

51.7. 依存関係解決の仕組み

Gradleは、ビルドスクリプトの依存関係宣言とリポジトリ定義を読み込み、依存関係解決
と呼ばれる処理によって依存関係を全てダウンロードしようと試みます。以下に、その処理がどのように

- 必要な依存関係が指示されると、Gradleはまず、リポジトリを順に検査して、その依存関係の
  モジュールを解決しようとします。　モジュールディスクリプタ
  ファイル(POMまたはIvyファイル)が見つかれば、リポジトリにモジュールがあると判断します。ディ
  モジュールアーティファクトファイルがあるかどうかでモジュールの有無を判断します。
  - もし、依存関係が動的バージョン(`1.+`
    のような)で指定されていれば、Gradleは動的バージョンを、リポジトリ内の最新の静的バージョ
    `1.2`など)に解決します。この処理は、Mavenリポジトリに対しては`maven-metadata.xml`
    ファイルを使って行われ、Ivyリポジトリの場合はディレクトリ内の一覧表示が使用されます。
  - モジュールディスクリプタが、親POMの宣言を含む`POM`
    ファイルだった場合、GradleはPOMの親モジュールを再帰的に解決しようとします。
- 全てのリポジトリでモジュールの調査が終わると、Gradleはそれらのリポジトリの中から、使うのに
  - 動的バージョンに関しては、なるべく新しい静的バージョンが使用できるリポジトリが好ましい。
  - モジュールディスクリプタ(ivyやpomファイル)で宣言されているモジュールは、アーティファクト
  - 早く見に行ったリポジトリのほうが、後のリポジトリよりも優先される。

  依存関係が静的バージョンで宣言されており、かつモジュールディスクリプタがリポジトリにあれば
- モジュール内の全アーティファクトを、上記処理で選択したリポジトリと同じリポジトリ
  に取得しに行きます。

51.8. 依存関係解決処理の微調整

Gradleの依存関係管理は、ほとんどの場合、デフォルトのままでビルドに必要な依存関係を解決できます

Gradleの依存関係解決処理は、様々な部分で調整可能です。

51.8.1. モジュールのバージョンを強制する

指定した依存関係について、（推移的かどうかにかかわらず）指定したバージョンを使うよう解決処理を
依存関係は、グループ名と名前で指定します。この機能は、バージョンの競合に四苦八苦している際にと
「バージョン競合の解決 」を参照してください。

また、推移的な依存関係のメタデータが壊れている場合もこれで対処できることがあります。推移的な依
R e s o l u t i o n S t r a t e g y
の例を参照してください。また、「依存関係解決ルール」は壊れたモジュールを置き換える更に強力な機
「バージョンのブラックリストと差し替え 」を参照してください。

51.8.2. 依存関係解決ルールを使う

依存関係解決ルールは、全ての依存関係に対して実行され、要求された依存関係を解決前に操作する強力
こ の 機 能 は 試 験 的 な
ものですが、要求した依存関係のグループ名や名前、バージョンを変更し、解決処理中に依存関係を完全

依存関係解決ルールは依存関係をコントロールする非常に強力な手段で、依存関係管理における先進的な
いくつかのパターンの概要を以下に紹介します。詳細な情報やコードサンプル等は
ResolutionStrategyをご参照ください。

51.8.2.1. リリース可能単位を表現する

組織によっては、いくつかのライブラリをまとめて一つのバージョンでリリースすることがあります。そ
そのようなライブラリ群は、「リリース可能単位」として構成され、一緒に使用されることが想定されて

しかし、この制限は、推移的な依存関係の中で簡単に破られてしまいます。以下の例をご覧ください。

- module-a depends on releasable-unit:part-one:1.0
- module-b depends on releasable-unit:part-two:1.1

m o d u l e - a と m o d u l e - b
の両方に依存しているビルドがあれば、それぞれのモジュールがリリース可能単位を構成するライブラリ

依存関係解決ルールを使うと、ビルドで使用するリリース可能単位を強制できます。
'org.gradle'グループに属する全てのライブラリが、あるリリース可能単位を構成していると想像してくだ

例51.46 あるグループのライブラリ全てで一貫したバージョンを使用するよう強制する

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.gradle') {
            details.useVersion '1.4'
        }
    }
}
```

51.8.2.2. 独自のバージョニング体制を実装する

社内環境によっては、gradleのビルドで使用できるモジュールのバージョンリストを外部的に監視、維持
依存関係解決ルールはこのようなパターンを簡潔に実装できます。

- ビルドスクリプトでは、使用するモジュールのグループと名前を開発者が記述しますが、バージョン
  default」などのプレースホルダで宣言します。

- 「default
  」バージョンは、依存関係解決ルールがある特定のバージョンに置換します。その際、承認されたモ

このルールの実装は、整理して社内プラグインにカプセル化し、組織内の全てのビルドで共有することも

例51.47 独自のバージョニング体制を実装する

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.version == 'default') {
            def version = findDefaultVersionInCatalog(details.requested.group, deta
            details.useVersion version
        }
    }
}

def findDefaultVersionInCatalog(String group, String name) {
    //some custom logic that resolves the default version into a specific version
    "1.0"
}
```

51.8.2.3. バージョンのブラックリストと差し替え

依存関係解決ルールを使って、あるモジュールの特定のバージョンをブラックリストに指定し、バージョ
これは、あるモジュールにおいて特定のバージョンが壊れていて使いたくないという場合に有用で、その
一例を挙げると、どこの公開リポジトリにも存在しないライブラリに依存しているようなモジュールは

以下の例では、バージョン1.2.1に重要な修正が含まれていて、1.2
ではなく必ずそちらを使いたい、という状況を想定しています。 例のルールは、単にバージョン1.2
を見つけたら必ず1.2.1
に差し替えられるということを保証しているだけですが、注目すべき点は、これが上の方で述べたバージ
例えばバージョン1.3もどこかで推移的に引っ張られている場合、競合解決戦略「newest」によりバージ

例51.48 バージョンのブラックリスト指定と差し替

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.software' && details.requested.name == 's
            //prefer different version which contains some necessary fixes
            details.useVersion '1.2.1'
        }
    }
}
```

51.8.2.4. 依存モジュールを互換性のあるモジュールに差し替える

要求されたモジュールを、全く異なる別モジュールに差し替えることができる場合もあります。
例にあるのは、「`groovy-all`」から「`groovy`」への差し替えと「`log4j`」から「`log4j-over-slf4`
」 へ の 差 し 替 え で す 。
Gradle1.5から、このような置き換えも依存関係解決ルールで可能になりました。

例51.49 解決時に依存関係のグループ名や名前を変更する

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.name == 'groovy-all') {
            //prefer 'groovy' over 'groovy-all':
            details.useTarget group: details.requested.group, name: 'groovy', vers:
        }
        if (details.requested.name == 'log4j') {
            //prefer 'log4j-over-slf4j' over 'log4j', with fixed version:
            details.useTarget "org.slf4j:log4j-over-slf4j:1.7.7"
        }
    }
}
```

51.8.2.5. Declaring that a legacy library is replaced by a new one

A good example when a new library replaced a legacy one is the "google-collections" -> "guava" migration. The team that created google-collections decided to change the module name from "com.google.collections:google-collections" into "com.google.guava:guava". This a legal scenario in the industry: teams need to be able to change the names of products they maintain, including the module coordinates. Renaming of the module coordinates has impact on conflict resolution.

To explain the impact on conflict resolution, let's consider the "google-collections" -> "guava" scenario. It may happen that both libraries are pulled into the same dependency graph. For example, "our" project depends on guava but some of our dependencies pull in a legacy version of google-collections. This can cause runtime errors, for example during test or application execution. Gradle does not automatically resolve the google-collections VS guava conflict because it is not considered as a "version conflict". It's because the module coordinates for both libraries are completely different and conflict resolution is activated when "group" and "name" coordinates are the same but there are different versions available in the dependency graph (for more info, please refer to the section on conflict resolution). Traditional remedies to this problem are:

- Declare exclusion rule to avoid pulling in "google-collections" to graph. It is probably the most popular approach.
- Avoid dependencies that pull in legacy libraries.
- Upgrade the dependency version if the new version no longer pulls in a legacy library.
- Downgrade to "google-collections". It's not recommended, just mentioned for completeness.

Traditional approaches work but they are not general enough. For example, an organisation

wants to resolve the google-collections VS guava conflict resolution problem in all projects. Starting from Gradle 2.2 it is possible to declare that certain module was replaced by other. This enables organisations to include the information about module replacement in the corporate plugin suite and resolve the problem holistically for all Gradle-powered projects in the enterprise.

例51.50 Declaring module replacement

build.gradle

```
dependencies {
    components {
        module("com.google.collections:google-collections").replacedBy("com.google.gu
    }
}
```

For more examples and detailed API, please refer to the DSL reference for `ComponentMetadataHandler`.

What happens when we declare that "google-collections" are replaced by "guava"? Gradle can use this information for conflict resolution. Gradle will consider every version of "guava" newer/better than any version of "google-collections". Also, Gradle will ensure that only guava jar is present in the classpath / resolved file list. Please note that if only "google-collections" appears in the dependency graph (e.g. no "guava") Gradle will not eagerly replace it with "guava". Module replacement is an information that Gradle uses for resolving conflicts. If there is no conflict (e.g. only "google-collections" or only "guava" in the graph) the replacement information is not used.

Currently it is not possible to declare that certain modules is replaced by a set of modules. However, it is possible to declare that multiple modules are replaced by a single module.

51.8.3. Ivyの動的解決モードを有効にする

GradleのIvyレポジトリ実装は、Ivyの動的解決モードと同等のものをサポートしています。通常、Gradle
`ivy.xml`ファイルで定義されている`rev`属性を使用しますが、動的解決モードでは、Gradleは`rev`属性の代わりに`revConstraint`属性を優先して使用します。`revConstraint`が未定義の時は`rev`属性を使用します。

動的解決モードを有効にするには、リポジトリ定義に適切なオプションを設定する必要があります。いく
`DependencyResolver`実装のIvyリポジトリでは使用できません。

例51.51 動的解決モードを有効にする

build.gradle

```
// Can enable dynamic resolve mode when you define the repository
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        resolve.dynamicMode = true
    }
}

// Can use a rule instead to enable (or disable) dynamic resolve mode for all reposit
repositories.withType(IvyArtifactRepository) {
    resolve.dynamicMode = true
}
```

51.8.4. Component metadata rules

Each module (also called component) has metadata associated with it, such as its group, name, version, dependencies, and so on. This metadata typically originates in the module's descriptor. Metadata rules allow certain parts of a module's metadata to be manipulated from within the build script. They take effect after a module's descriptor has been downloaded, but before it has been selected among all candidate versions. This makes metadata rules another instrument for customizing dependency resolution.

One piece of module metadata that Gradle understands is a module's status scheme. This concept, also known from Ivy, models the different levels of maturity that a module transitions through over time. The default status scheme, ordered from least to most mature status, is `integrat` , `milestone`, `release`. Apart from a status scheme, a module also has a (current) status, which must be one of the values in its status scheme. If not specified in the (Ivy) descriptor, the status defaults to `integration` for Ivy modules and Maven snapshot modules, and `release` for Maven modules that aren't snapshots.

A module's status and status scheme are taken into consideration when a `latest` version selector is resolved. Specifically, `latest.someStatus` will resolve to the highest module version that has status `someStatus` or a more mature status. For example, with the default status scheme in place, `latest.integration` will select the highest module version regardless of its status (because `integration` is the least mature status), whereas `latest.release` will select the highest module version with status `release`. Here is what this looks like in code:

例51.52 'Latest' version selector

build.gradle

```
dependencies {
    config1 "sea.fish:tuna:latest.integration"
    config2 "sea.fish:tuna:latest.release"
}

task listFish << {
    configurations.config1.each { println it.name }
    println()
    configurations.config2.each { println it.name}
}
```

`gradle -q listFish` の出力

```
> gradle -q listFish
tuna-1.5.jar

tuna-1.4.jar
```

The next example demonstrates `latest` selectors based on a custom status scheme declared in a module metadata rule:

例51.53 Custom status scheme

build.gradle

```
dependencies {
    config3 "air.birds:albatross:latest.silver"
    components {
        eachComponent { ComponentMetadataDetails details ->
            if (details.id.group == "air.birds") {
                details.statusScheme = ["bronze", "silver", "gold", "platinum"]
            }
        }
    }
}

task listBirds << {
    configurations.config3.each { println it.name }
}
```

`gradle -q listBirds` の出力

```
> gradle -q listBirds
albatross-2.0.jar
```

Gradle can also create component metadata rules utilizing Ivy-specific metadata for modules resolved from an Ivy repository. Values from the Ivy descriptor are made available via the `IvyModuleDescriptor` interface.

例51.54 Ivy component metadata rule

build.gradle

```
dependencies {
    components {
        eachComponent { ComponentMetadataDetails details, IvyModuleDescriptor ivyMo
            if (details.id.group == 'my.org' && ivyModule.branch == 'testing') {
                details.changing = true
            }
        }
    }
}
```

51.8.5. Component Selection Rules

Component selection rules define how versions should be selected when multiple versions are available that match a given set of coordinates. Rules are applied against every available version and allow the version to be explicitly rejected by rule. This allows Gradle to select alternate versions when the alternate version is not known in advance.

Rules are configured via the `ComponentSelectionRules` object. Each rule configured will be called with a `ComponentSelection` object as an argument which contains information about the version being requested and the candidate version being considered. Calling `ComponentSelection.reject()` causes the given candidate version to be explicitly rejected, in which case the candidate will not be considered for the selector.

The following example shows a rule that disallows a particular version of a module but allows the default version selection rules to choose the next best candidate. If the version requested was `1.+`, and versions 1.1, 1.0.1 and 1.0 were available, it would reject 1.1 and select the next newest version available (1.0.1).

例51.55 Component selection rule

build.gradle

```
configurations {
    rejectConfig {
        resolutionStrategy {
            componentSelection {
                // Accept the highest version matching the requested version that isn
                all { ComponentSelection selection ->
                    if (selection.candidate.group == 'org.sample' && selection.canc
                        selection.reject("version 1.1 is broken for 'org.sample:api'"
                    }
                }
            }
        }
    }
}
```

Note that version selection is applied starting with the highest version first. The version selected will be the first version found that all component selection rules accept. A version is considered

accepted if a rule does not explicitly reject it.

Similarly, rules can be targeted at specific modules. Modules must be specified in the form of "group:module".

例51.56 Component selection rule with module target

build.gradle

```
configurations {
    targetConfig {
        resolutionStrategy {
            componentSelection {
                module("org.sample:api") { ComponentSelection selection ->
                    if (selection.candidate.version == "1.1") {
                        selection.reject("known bad version")
                    }
                }
            }
        }
    }
}
```

Component selection rules can also consider component metadata when selecting a version. Possible metadata arguments that can be considered are `ComponentMetadata` and `IvyModuleDescriptor`.

例51.57 Component selection rule with metadata

build.gradle

```
configurations {
    rejectConfig {
        resolutionStrategy {
            componentSelection {
                // Reject any versions with a status of 'experimental'
                all { ComponentSelection selection, ComponentMetadata metadata ->
                    if (selection.candidate.group == 'org.sample' && selection.cand
                        if (metadata.status == 'experimental') {
                            selection.reject("don't use experimental candidates of 'o
                        }
                    }
                }
                // Accept the highest version with a branch of 'testing' or a status
                all { ComponentSelection selection, IvyModuleDescriptor descriptor
                    if (selection.candidate.group == 'org.sample' && selection.cand
                        if (descriptor.branch != 'testing' && metadata.status != 'mil
                            selection.reject("'org.sample:api' must have testing bran
                        }
                    }
                }
            }
        }
    }
}
```

Note that a `ComponentSelection` argument is always required when declaring a component

selection rule but the metadata arguments are optional.

51.9. 依存関係のキャッシュ

Gradleは、非常に洗練された依存関係のキャッシング機構を持っており、依存関係の解決の際、リモートI
また、その際、依存関係解決の正確性、再現性を最大限保証できるような仕組みを備えています。

Gradleの依存関係キャッシュは、二種類のストレージで構成されています。

- ダウンロードされたアーティファクトを保存する、ファイルベースのストレージ。pomファイルやivy
  ダウンロード済みのアーティファクトを保存するストレージパスには、SHA1のチェックサムが含まれ
- 解決済みのメタデータを保存するバイナリストレージ。動的バージョン、モジュールのディスクリプ

メタデータのキャッシュとダウンロードしたアーティファクトを別のストレージに分けることで、ファイ

Gradleのキャッシュでは、ローカルキャッシュによって問題が隠蔽されるようなことはありませんし、多
また、その実装は帯域、ストレージ双方において効率的なものです。それにより、Gradleはエンタープラ

51.9.1. Gradleにおける依存関係キャッシュの重要な機能

51.9.1.1. 分離されたメタデータ・キャッシュ

Gradleは、依存関係を解決した結果を様々な観点から記録し、バイナリ形式でメタデータ・キャッシュに
メタデータ・キャッシュに保存される情報には以下のようなものがあります。

- (`1.+`のような)動的バージョンを具体的なバージョン(`1.2`など)に解決したその結果。
- それぞれのモジュールの解決済みメタデータ。モジュールのアーティファクトや依存関係など。
- それぞれのアーティファクトの、解決済みメタデータ。ダウンロードしたファイルへのポインタを含む
- あるモジュールやアーティファクトがリポジトリに<u>無い</u>
  ことを記録し、存在しないリソースに繰り返しアクセスしに行くことを防ぐ。

メタデータ・キャッシュの全てのエントリには、リポジトリに関する記録が含まれていて、キャッシュの

51.9.1.2. リポジトリキャッシュの独立性

前述の通り、リポジトリにはそれぞれ独立したメタデータ・キャッシュが存在します。リポジトリはURL
も し <u>そ の リ ポ</u> ジ ト リ
から、あるモジュールやアーティファクトをまだ解決したことがない場合、Gradleはリポジトリに対して
「アーティファクトの再利用 」」を参照してください)。

ビルドで指定されたどのリポジトリにも要求されたアーティファクトがない場合、たとえ別のリポジトリ
リポジトリの独立性を保つことで、それぞれのビルドも高度に独立したものにしているのです。これは、

51.9.1.3. アーティファクトの再利用

アーティファクトをダウンロードする前に、Gradleは、そのアーティファクトに結びついたshaファイル
チェックサムをダウンロードできれば、そして既に同じIDとチェックサムを持つアーティファクトが存在
チェックサムをリモートサーバーから取得できない場合、アーティファクトは常にダウンロードされます

Gradleは、別のリポジトリからダウンロードされたアーティファクトだけでなく、Mavenのローカルリ

51.9.1.4. チェックサムベースのストレージ

同じ識別子でアーティファクトを要求したのに、それぞれのリポジトリが別のバイナリを返すことがあり
Gradleは、SHA1チェックサムに基づいてアーティファクトをキャッシングするので、同じアーティファ
これはつまり、あるリポジトリからアーティファクトを解決した際、別のリポジトリのファイルキャッシ
これを、それぞれのリポジトリごとに別個にアーティファクトファイルのキャッシュを持つことなく実現

51.9.1.5. キャッシュのロック

Gradleの依存関係キャッシュは、並列に実行される複数のGradleプロセスから安全に使用できるよう、
このロックは、メタデータバイナリストレージの読み込みや書き込みの際には常に取得されますが、リモ

51.9.2. キャッシングに関するコマンドラインオプション

51.9.2.1. オフライン

`--offline`
オプションを使うと、常にキャッシュされた依存モジュールを使用するようGradleに指示できます。依存
オフラインで実行されると、Gradleは依存関係を解決するときにネットワークにアクセスしに行くことは
要求されたモジュールが依存関係のキャッシュにない場合は、ビルドは失敗します。

51.9.2.2. リフレッシュ

Gradleの依存関係キャッシュが、設定されたリポジトリの実際の状態と同期が取れていないこともありま
おそらくは最初のリポジトリ設定が間違っていたか、未変更のモジュールが間違って公開されたのでしょ
`--refresh-dependencies`オプションを使用してください。

`--refresh-dependencies`
オプションは、解決済みのモジュールとアーティファクトに対する全てのキャッシュエントリを無視する
対象のリポジトリ全てに対して、新規に解決処理が実行されます。動的バージョンは再取得され、モジュ
ただ、実際にアーティファクトをダウンロードする前に、Gradleは以前ダウンロードしたものが使用でき
これは、リポジトリで公開されているSHA1と、ダウンロード済みのアーティファクトのSHA1を比較する

51.9.3. 依存関係キャッシュ制御の微調整

キャッシングについて、いくつかの動作は設定で`ResolutionStrategy`
を使うことで微調整することができます。

デフォルトでは、Gradleは動的バージョンを24時間キャッシュします。動的バージョンを具体的なバー

例51.58 動的バージョンのキャッシュ制御

build.gradle

```
configurations.all {
    resolutionStrategy.cacheDynamicVersionsFor 10, 'minutes'
}
```

デフォルトでは、Gradleは変更性モジュールを24時間キャッシュします。変更性モジュールのメタデー

例51.59 変更性モジュールのキャッシュ制御

build.gradle

```
configurations.all {
    resolutionStrategy.cacheChangingModulesFor 4, 'hours'
}
```

詳しくはAPIドキュメントで`ResolutionStrategy`を参照してください。

51.10. 推移的依存関係を管理するための戦略

多くのプロジェクトがMavenセントラルリポジトリ
に依存していますが、これが時に問題になることもあります。

- Mavenセントラルリポジトリはダウンする可能性もあり、またレスポンスが返るまで長い時間がかか
- POMファイルに間違った情報を入れているプロジェクトがたくさんあります(たとえば、`commons-h` はJUnitをruntime依存関係に設定しています)。
- 「ただ一つの正当な依存関係」というものが決められないようなプロジェクトもたくさんあります(つ

もしプロジェクトがMavenセントラルリポジトリに依存しているのであれば、カスタムリポジトリを追加

- まだMavenセントラルにアップロードされていない依存関係が必要になった。
- You want to deal properly with invalid metadata in a Maven Central POM file.
  MavenセントラルのPOMファイルに誤ったメタデータが含まれていた。
- プロジェクトをビルドしたいと思う人々がいて、彼らをMavenセントラルのダウンタイムや、時折発

使用しているライブラリをバージョン管理システムに入れているプロジェクトがありますが、それはこの

Gitのような分散バージョン管理システムを使っている場合は、リポジトリにライブラリを格納したくない

これらを複合させた戦略をとることもあります。POMファイルの不良メタデータをなんとかしたい、カス
もちろんjarの取得だけをMaven2リポジトリやカスタムリポジトリから行うこともできますし、推移的な

51.10.1. 暗黙的な推移的依存関係

The trick is to use only artifact dependencies and group them in lists. This will directly express your first level dependencies and your transitive dependencies (see 「オプション属性 」). The problem with this is that Gradle dependency management will see this as specifying all dependencies as first level dependencies. The dependency reports won't show your real dependency graph and the `compile` task uses all dependencies, not just the first level dependencies. All in all, your build is less maintainable and reliable than it could be when using client modules, and you don't gain anything.

そのトリックは、アーティファクトオンリー記法を使い、それらをリストでグループ化するというもので
この方法でも、なにが直接の依存関係で、何が推移的な依存関係なのか、なんとか表現することはできま
「オプション属性 」を参照してください)。
しかし、この方法の欠点は、Gradleの依存関係管理システムが、すべての依存関係を直接の依存関係とし

c o m p i l e
タスクでも、直接的な依存関係だけでなく、すべての依存関係が使われてしまうことになります。クライ

---

[22] Gradleはマルチプロジェクトの部分ビルドをサポートしています(57章マルチプロジェクトのビルド参照)。

[23] http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-project-relationships.ht

[24] ivyパターンについてはhttp://ant.apache.org/ivy/history/latest-milestone/concept.htmlで詳細な情報を知ることができます。

[25] Mavenセントラルのダウンからプロジェクトを防御したい場合は、もう少し複雑になります。
おそらく、そのためにリポジトリのプロキシをセットアップしたくなると思います。
エンタープライズ環境では珍しくないシチュエーションと言えますが、オープンソースプロジェクトの場

52
アーティファクトの公開

この章は、Gradle1.0の段階で使用されている既存の
公開システムについて書かれています。Gradle1.3からは、公開処理に新しい仕組みが導入されました
この仕組みは現在試験的な
段階にあり、まだ完成していませんが、Gradleの公開処理をより強力にする新しい概念、機能がいく
新しい公開プラグインについては、Chapter 65, Ivy Publishing (new)およびChapter 66,
Maven Publishing (new)
に詳しく記載されています。ぜひ試用してみてください。何かあればフィードバックしてもらえると

52.1. はじめに

この章では、プロジェクトが公開するアーティファクト(成果物)の定義方法と取り扱い(アップロードする

52.2. アーティファクトとコンフィギュレーション

依存関係と同じように、アーティファクトもコンフィギュレーションによってグループ化されます。実際

Gradleは、プロジェクト内のすべてのコンフィギュレーションに対して、`upload` タスクと`build` タスクを提供します。[27]
これらのタスクを実行すると、対象のコンフィギュレーションに属しているアーティファクトをビルドし

表23.5「Javaプラグイン - 依存関係のコンフィギュレーション」
に、Javaプラグインが追加するコンフィギュレーションの一覧を載せています。これらのうち、アーティ
`archives`
コンフィギュレーションは、プロジェクトのアーティファクトを含めるための標準のコンフィギュレーシ
「プロジェクトライブラリについての追記事項」では、`runtime`
コンフィギュレーションについて詳しく解説します。また、アーティファクトを追加するための独自コン

52.3. アーティファクトの宣言

52.3.1. アーカイブタスク・アーティファクト

アーカイブタスクを使ってアーティファクトを宣言できます。

例52.1 アーカイブタスクを使ってアーティファクトを宣言する

build.gradle

```
task myJar(type: Jar)

artifacts {
    archives myJar
}
```

重要なことですが、独自のアーカイブ作成をビルドに組み込んでも、そのアーカイブが自動的になんらか

52.3.2. ファイル・アーティファクト

アーティファクトは、ファイルで宣言することもできます。

例52.2 アーティファクトをファイルで宣言する

build.gradle

```
def someFile = file('build/somefile.txt')

artifacts {
    archives someFile
}
```

Gradleは、ファイル名を元にアーティファクトのプロパティを決定します。このプロパティは、以下のよ

例52.3 アーティファクトのカスタマイズ

build.gradle

```
task myTask(type:  MyTaskType) {
    destFile = file('build/somefile.txt')
}

artifacts {
    archives(myTask.destFile) {
        name 'my-artifact'
        type 'text'
        builtBy myTask
    }
}
```

ファイルでアーティファクトを宣言する際には、Mapスタイルの文法を使うこともできます。そのMapに

例52.4 ファイルでアーティファクトを宣言するMapスタイルの文法

build.gradle

```
task generate(type:  MyTaskType) {
    destFile = file('build/somefile.txt')
}

artifacts {
    archives file: generate.destFile, name: 'my-artifact', type: 'text', builtBy: g
}
```

52.4. アーティファクトの公開

先ほど、すべてのコンフィギュレーションにアップロード用のタスクが作成されることを説明しました。
「リポジトリ」
)が、勝手にアップロードに使用されることはありません。実際、それらのリポジトリには、ダウンロー

例52.5 アップロードタスクの設定

build.gradle

```
repositories {
    flatDir {
        name "fileRepo"
        dirs "repo"
    }
}

uploadArchives {
    repositories {
        add project.repositories.fileRepo
        ivy {
            credentials {
                username "username"
                password "pw"
            }
            url "http://repo.mycompany.com"
        }
    }
}
```

この例のように、定義済みのリポジトリを参照することもできますし、新しくリポジトリ定義を作って追
「Ivyリゾルバについて」
のときと同じように、アップロードでもすべてのIvyリポジトリリゾルバを使うことができます。

アップロード先のリポジトリ定義に複数のレイアウトパターンが含まれている場合、どのパターンに従っ
デフォルトでは、Gradleは`url`パラメーターを参照します。`layout`
が定義されていれば、それも併せて使用します。 `url`

パラメーターが未定義の場合は、定義されている`artifactPattern`
のうち最初のものがアップロードに使用され、また、定義されている`ivyPattern`
のうち最初のものがivyファイルのアップロードに使用されます。

Mavenリポジトリへのアップロードについては、「Mavenリポジトリとの相互作用」
をご参照ください。

52.5. プロジェクトライブラリについての追記事項

プロジェクトがライブラリとして使用される場合、そのライブラリのアーティファクトは何か、アーティ
`runtime`というコンフィギュレーションを追加します。これは、公開したいアーティファクトは
`runtime`
依存関係に依存しているはずだ、という暗黙的な仮定の上での仕様ですが、もちろんこのあたりの設定は

誰かがあなたのプロジェクトをライブラリとして使いたいとしましょう。そのときすべきことは、あなた
`configuration`
というプロパティを指定できるようになっています。もしこのプロパティが指定されていなければ、
`default`コンフィギュレーションが使用されます(「依存するコンフィギュレーションの指定」
をご参照ください)。
あるプロジェクトがライブラリとして使われるのには、二つのケースがあります。マルチプロジェクトの
「Mavenリポジトリとの相互作用」をご覧ください。

---

[7] 正確に言うと、Baseプラグインがこれらのタスクを提供します。Baseプラグインは、Javaプラグインを使

53
Mavenプラグイン

この章は執筆途中です。

Mavenプラグインは、Mavenリポジトリへのアーティファクトのデプロイに対するサポートを追加しま

53.1. 使用方法

Mavenプラグインを使うには、ビルドスクリプトに以下を含めます:

例53.1 Mavenプラグインの利用

build.gradle

```
apply plugin: 'maven'
```

53.2. タスク

Mavenプラグインは以下のタスクを定義しています:

表53.1 Mavenプラグイン - タスク

タスク名	依存先	型	説明
install	関連するアーカイブをビルドするすべてのタスク	Upload	Mavenメタデータの生成 デフォルトではinstallタ このコンフィグレーショ ローカルリポジトリへの を参照のこと。

53.3. 依存関係管理

Mavenプラグインはいかなる依存関係のコンフィグレーションも定義していません。

53.4. 規約プロパティ

Mavenプラグインは以下の規約プロパティを定義しています:

表53.2 Mavenプラグイン - プロパティ

プロパティ名	型	デフォルト値	説明
pomDirName	String	poms	生成されたPOMを書
pomDir	File (read-only)	*buildDir/pomDirName*	生成されたPOMを書
conf2ScopeMappings	Conf2ScopeMappingContainer	n/a	Gradleコンフィグレ 「依存関係のマッピ

これらのプロパティは規約オブジェクト`MavenPluginConvention`が提供します。

53.5. 規約メソッド

MavenプラグインはPOM生成のためのファクトリメソッドを提供します。
これはMavenリポジトリへのアップロードを必要としないPOMが必要なときに便利です。

例53.2 スタンドアロンPOMの生成

build.gradle

```
task writeNewPom << {
    pom {
        project {
            inceptionYear '2008'
            licenses {
                license {
                    name 'The Apache Software License, Version 2.0'
                    url 'http://www.apache.org/licenses/LICENSE-2.0.txt'
                    distribution 'repo'
                }
            }
        }
    }.writeTo("$buildDir/newpom.xml")
}
```

そのほか、Gradleは多言語Mavenと同じビルダー文法をサポートします。 Gradle Maven POMオブジェクトについてもっと学ぶためには、 `MavenPom`を参照してください。また、`MavenPluginConvention`も有用です。

53.6. Mavenリポジトリとの相互作用

53.6.1. はじめに

Gradleでは、リモートMavenリポジトリにデプロイすることも、ローカルMavenリポジトリにインスト-
これにはすべてのMavenメタデータの操作が含まれ、Mavenスナップショットに対しても適用できます。
実際、GradleのデプロイはネィティブのMaven
Antタスクを内部で利用するのと同じく、100%Maven互換です。

もしPOMがないとしたら、Mavenリポジトリへのデプロイは楽しさ半減です。
幸運なことに、GradleはPOM自身の持つ依存関係の情報を用いて、このPOMを生成してくれます

53.6.2. Mavenリポジトリへのデプロイ

プロジェクトが単にデフォルトのJARファイルを生成するだけだとしましょう。
このJARファイルをリモートMavenリポジトリにデプロイしたいとします。

例53.3 リモートMavenリポジトリへのファイルアップロード

build.gradle

```
apply plugin: 'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
        }
    }
}
```

これが全てです。 `uploadArchives`タスクを呼び出すことでPOMが生成され、
アーティファクトとPOMが指定されたリポジトリにデプロイされます。

`file`以外のプロトコルをサポートする必要があるなら、他にも少しやるべきことがあります。
この場合、処理を委譲するネィティブMavenコードが追加のライブラリを必要とします。
どのライブラリが必要になるかは、必要なプロトコルによります。
利用可能なプロトコルと対応するライブラリは表53.3「Mavenデプロイに対するプロトコルJAR」
にリストアップされています [29]
(推移的な依存関係を持つこれらのライブラリは、さらに推移的な依存関係を持ちます)。
例えば、SSHプロトコルを使うには次のようにできます:

例53.4 SSH経由でのファイルアップロード

build.gradle

```
configurations {
    deployerJars
}

repositories {
    mavenCentral()
}

dependencies {
    deployerJars "org.apache.maven.wagon:wagon-ssh:2.2"
}

uploadArchives {
    repositories.mavenDeployer {
        configuration = configurations.deployerJars
        repository(url: "scp://repos.mycompany.com/releases") {
            authentication(userName: "me", password: "myPassword")
        }
    }
}
```

Mavenデプロイヤーには多くのコンフィグレーションオプションがあります。 コンフィグレーションはGroovyビルダーで行うことができます。 このツリーのすべての要素はJava beansです。 単純な属性を設定するためには、beanの要素にmapを渡してやります。 親要素に別のbean要素を追加するには、クロージャを使います。 上記の例では、repositoryとauthenticationがbean要素です。 利用可能なbean要素と対応するクラスのJavaDocへのリンクは、表53.4「MavenDeployerのコンフィグレーション要素」にリストアップされています。 JavaDocでは特定の要素にセットできる属性を確認できます。

Mavenではリポジトリに加えて、オプションでスナップショットリポジトリを定義できます。 スナップショットリポジトリが定義されていない場合、 リリースとスナップショットの両方が`reposit`要素で指定された先にデプロイされます。 そうでない場合は、スナップショットは`snapshotReposito`要素で指定された先にデプロイされます。

表53.3 Mavenデプロイに対するプロトコルJAR

プロトコル	ライブラリ
http	org.apache.maven.wagon:wagon-http:2.2
ssh	org.apache.maven.wagon:wagon-ssh:2.2
ssh-external	org.apache.maven.wagon:wagon-ssh-external:2.2
ftp	org.apache.maven.wagon:wagon-ftp:2.2
webdav	org.apache.maven.wagon:wagon-webdav:1.0-beta-2
file	-

表53.4 MavenDeployerのコンフィグレーション要素

要素	Javadoc
root	`MavenDeployer`
repository	org.apache.maven.artifact.ant.RemoteRepository
authentication	org.apache.maven.artifact.ant.Authentication
releases	org.apache.maven.artifact.ant.RepositoryPolicy
snapshots	org.apache.maven.artifact.ant.RepositoryPolicy
proxy	org.apache.maven.artifact.ant.Proxy
snapshotRepository	org.apache.maven.artifact.ant.RemoteRepository

53.6.3. ローカルリポジトリへのインストール

Mavenプラグインはプロジェクトに`install`タスクを追加します。 このタスクは`archives`コンフィグレーションのすべてのarchivesタスクに依存します。 これらのアーカイブはローカルMavenリポジトリにインストールされます。 もし、ローカルリポジトリの場所がMavenの`settings.xml`で再定義されていたら、このタスクはそれを反映します。

53.6.4. MavenのPOM生成

Mavenリポジトリにアーティファクトをデプロイする際、Gradleは自動的にデプロイ用のPOMを生成し POMにデフォルトで設定される`groupId`、`artifactId`、`version`、`packaging`要素の値は以下の表を参照してください。 `dependency`要素は、プロジェクトで宣言されている依存関係設定から作成されます。

表53.5 Maven POM生成のためのデフォルト値

Maven要素	デフォルト値
groupId	project.group
artifactId	uploadTask.repositories.mavenDeployer.pom.artifactId (if set) or archiveTask.baseName.
version	project.version
packaging	archiveTask.extension

ここで、`uploadTask`と`archiveTask`は、それぞれ、実際にアーカイブをアップロードするタスクと生成するタスクのことを指しています(例; `uploadArchives`と`jar`)。 `archiveTask.baseName`のデフォルト値は`project.archivesBaseNam`で、そのデフォルト値は`project.name`です。

> `archiveTask.baseName`をデフォルト以外の値に設定したときは、`uploadTask.repositories.`を同じ値に設定することを忘れないでください。

さもないと、近いうちに、同じビルド内の他プロジェクトで生成されたPOMから、プロジェクトが誤IDで参照されてしまうかもしれません。

生成されたPOMは、`<buildDir>/poms`に出力されます。そのPOMは、さらに`MavenPom`
API で カ ス タ マ イ ズ す る こ と が で き ま す 。
例えば、Mavenにデプロイするアーティファクトには、Gradleで生成するアーティファクトとは別のバー

例53.5 pomのカスタマイズ

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
            pom.version = '1.0Maven'
            pom.artifactId = 'myMavenName'
        }
    }
}
```

POMに追加的な内容を加えるには、`pom.project`ビルダーを使います。　このビルダーを使えば、Maven POMリファレンスに載っている全ての要素を追加することができます。

例53.6 ビルダースタイルでpomをカスタマイズする

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
            pom.project {
                licenses {
                    license {
                        name 'The Apache Software License, Version 2.0'
                        url 'http://www.apache.org/licenses/LICENSE-2.0.txt'
                        distribution 'repo'
                    }
                }
            }
        }
    }
}
```

注意: `groupId`、`artifactId`、`version`および`packaging`は、常にpomオブジェクトに直接設定するべきです。

例53.7 自動生成された内容を変更する

build.gradle

```
def installer = install.repositories.mavenInstaller
def deployer = uploadArchives.repositories.mavenDeployer

[installer, deployer]*.pom*.whenConfigured {pom ->
    pom.dependencies.find {dep -> dep.groupId == 'group3' && dep.artifactId == 'runti
}
```

2つ以上のアーティファクトを公開したい場合は、すこし難しくなります。
「複数アーティファクトを含むプロジェクト」を参照してください。

Mavenインストーラー(「ローカルリポジトリへのインストール」
参照)の設定をカスタマイズするには、次のようにします。

例53.8 Mavenインストーラーのカスタマイズ

build.gradle

```
install {
    repositories.mavenInstaller {
        pom.version = '1.0Maven'
        pom.artifactId = 'myName'
    }
}
```

53.6.4.1. 複数アーティファクトを含むプロジェクト

Mavenはプロジェクトあたり1つのアーティファクトしか扱うことができません。
これはMavenのPOMの構造を反映しています。
我々は、プロジェクトが2つ以上のアーティファクトを含むことが合理的であるシチュエーションも多い
この場合、複数のPOMを生成する必要があります。
このような状況では、Mavenリポジトリに発行するそれぞれのアーティファクトを明示的に宣言する必要
`MavenDeployer`およびMavenInstallerが、これに対するAPIを提供します:

例53.9 複数POMの生成

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
            addFilter('api') {artifact, file ->
                artifact.name == 'api'
            }
            addFilter('service') {artifact, file ->
                artifact.name == 'service'
            }
            pom('api').version = 'mySpecialMavenVersion'
        }
    }
}
```

発行するそれぞれのアーティファクトに対してfilterを宣言する必要があります。
filterではGradleアーティファクトが受け付ける論理式を定義します。
それぞれのfilterは関連付けられたPOMを持ち、コンフィグレーション可能です。
より詳しく学ぶためには、`PomFilterContainer`および関連クラスを参照してください。

53.6.4.2. 依存関係のマッピング

Mavenプラグインは、JavaプラグインとWarプラグインによって追加されたGradleコンフィグレーション
Mavenスコープとの間のデフォルトのマッピングを構成します。
ほとんどの場合これに触れる必要はなく、この節は読み飛ばしていただいてかまいません。
マッピングは次のように動作します。
コンフィグレーションはただ一つのスコープにのみマップできます。
異なるコンフィグレーションは一つ、もしくは異なるスコープにマップできます。
特定のコンフィグレーション-スコープマッピングに優先度を割り当てることもできます。 詳しくは`Conf2ScopeMappingContainer`を参照してください。
マッピングコンフィグレーションにアクセスするには次のようにします:

例53.10 マッピングコンフィグレーションへのアクセス

build.gradle

```
task mappings << {
    println conf2ScopeMappings.mappings
}
```

Gradleのexcludeルールは、可能であればMavenのexcludeに変換されます。
この変換は、Gradleのexcludeルールにおいてgroupとモジュール名の両方が指定されている場合に可能(Ivyとは対照的に、Mavenは両方を必要とするため)。
コンフィグレーション毎のexcludeも、変換可能であればMavenのPOMに含まれます。

---

[29] これらは将来のリリースで標準サポートされる予定です

54
署名プラグイン

署名プラグインは、ビルドされたファイルやアーティファクトに電子署名を行う機能を追加します。
この電子署名は、いつその署名がされたのかなどの情報だけでなく、署名されたアーティファクトを誰か

現在署名プラグインがサポートしているのは、PGP署名(
Mavenセントラルリポジトリに公開するのに必要な署名形式)の生成のみです。

54.1. 使用方法

署名プラグインを使うには、以下の行をビルドスクリプトに追加します。

例54.1 署名プラグインの使用

build.gradle

```
apply plugin: 'signing'
```

54.2. 署名者の資格情報

PGP署名を作成するには、キーペアが必要です(GnuPGツール
を使用してキーペアを作成する手順については、GnuPGのHOWTO文書
で見つけることができます)。
署名プラグインでは、このキー情報、つまり以下の三つの情報を設定することができます。

- 公開鍵ID(8文字の16進数文字列)
- 秘密鍵を保存したキーリングファイルの絶対パス
- 秘密鍵を保護するパスフレーズ

これらの情報は、それぞれsigning.keyId、signing.secretKeyRingFile、signing.password
というプロジェクトプロパティで設定する必要があります。 良いプラクティスは、ユーザーのgradle.p
ファイル(「Gradleプロパティとシステムプロパティ」
で解説されています)にこれらを保存することです。

```
signing.keyId=24875D73
signing.password=secret
signing.secretKeyRingFile=/Users/me/.gnupg/secring.gpg
```

これらの情報をgradle.properties

ファイルで設定するのが環境の問題で難しい場合は、手動でプロジェクトのプロパティに設定させること

```
import org.gradle.plugins.signing.Sign

gradle.taskGraph.whenReady { taskGraph ->
    if (taskGraph.allTasks.any { it instanceof Sign }) {
        // Use Java 6's console to read from the console (no good for
        // a CI environment)
        Console console = System.console()
        console.printf "\n\nWe have to sign some things in this build." +
                       "\n\nPlease enter your signing details.\n\n"

        def id = console.readLine("PGP Key Id: ")
        def file = console.readLine("PGP Secret Key Ring File (absolute path): ")
        def password = console.readPassword("PGP Private Key Password: ")

        allprojects { ext."signing.keyId" = id }
        allprojects { ext."signing.secretKeyRingFile" = file }
        allprojects { ext."signing.password" = password }

        console.printf "\nThanks.\n\n"
    }
}
```

54.3. 署名対象を指定する

どのように署名するか(つまり署名者情報の指定)だけでなく、何を署名するかというのも設定しなければ
署名プラグインは、署名すべきタスクやコンフィギュレーションを指定するためのDSLを提供しています

54.3.1. コンフィグレーションに署名する

あるコンフィグレーション内のアーティファクトに対して署名をしたいというのは良くあることです。例
Javaプラグインは、jarをビルドするよう定義し、そのjarをアーティファクトとして`archives`
コンフィグレーションに追加します。
署名DSLを使えば、このコンフィグレーションに含まれる全てのアーティファクトに署名するよう設定で

例54.2 コンフィグレーションに署名する

build.gradle

```
signing {
    sign configurations.archives
}
```

これにより、「`signArchives`」という名前のタスク(`Sign`型)が追加されます。 このタスクは、`archi`
のアーティファクトを全て(必要なら)ビルドし、それからそれらのアーティファクト用に署名を作成しま

例54.3 コンフィグレーションの出力に署名する

gradle signArchives の出力

```
> gradle signArchives
:compileJava
:processResources
:classes
:jar
:signArchives

BUILD SUCCESSFUL

Total time: 1 secs
```

54.3.2. タスクに署名する

署名したいものが、コンフィギュレーションに含まれていないというケースもあります。このケースでに

例54.4 タスクを署名する

build.gradle

```
task stuffZip (type: Zip) {
    baseName = "stuff"
    from "src/stuff"
}

signing {
    sign stuffZip
}
```

これにより、「signStuffZip」という名前のタスク(Sign型)がプロジェクトに追加されます。
このタスクは、入力物を(必要なら)ビルドしてアーカイブを作成し、それからそのアーカイブに署名しま

例54.5 タスクの出力に署名する

gradle signStuffZip の出力

```
> gradle signStuffZip
:stuffZip
:signStuffZip

BUILD SUCCESSFUL

Total time: 1 secs
```

「署名可能」なタスクは、何らかのアーカイブを作成するものに限られます。この種類のタスクには、TarやZip、Jar、War、Earがあります。

54.3.3. 条件付き署名

よくあるビルドのパターンに、ある特定の条件下に限ってアーティファクトに署名する、というものがあ
これは、署名DSLを必要なロジックで囲めば実現できます。

例54.6 条件付き署名

build.gradle

```
version = '1.0-SNAPSHOT'
ext.isReleaseVersion = !version.endsWith("SNAPSHOT")

signing {
    required { isReleaseVersion && gradle.taskGraph.hasTask("uploadArchives") }
    sign configurations.archives
}
```

In this example, we only want to require signing if we are building a release version and we are going to publish it. Because we are inspecting the task graph to determine if we are going to be publishing, we must set the `signing.required` property to a closure to defer the evaluation. See `SigningExtension.setRequired()` for more information.

54.4. 署名を公開する

署名DSLで署名対象を指定すると、生成された署名は自動的に`signatures`
コンフィギュレーションと`archives`コンフィギュレーションに追加されます。
つまり、署名をアーティファクトと一緒に配布用リポジトリに公開したい場合は、通常通りただ`upload`
タスクを実行すればいいということです。

54.5. POMファイルに署名する

When deploying signatures for your artifacts to a Maven repository, you will also want to sign the published POM file. The signing plugin adds a `signing.signPom()` (see: `SigningExtension.signPom()`) method that can be used in the `beforeDeployment()` block in your upload task configuration.

例54.7 Signing a POM for deployment

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            beforeDeployment { MavenDeployment deployment -> signing.signPom(deploy
        }
    }
}
```

When signing is not required and the POM cannot be signed due to insufficient configuration (i.e. no credentials for signing) then the `signPom()` method will silently do nothing.

55
Building native binaries

> The Gradle support for building native binaries is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The various native binary plugins add support for building native software components, such as executables or shared libraries, from code written in C++, C and other languages. While many excellent build tools exist for this space of software development, Gradle offers developers its trademark power and flexibility together with dependency management practices more traditionally found in the JVM development space.

55.1. Supported languages

The following source languages are currently supported:

- C
- C++
- Objective-C
- Objective-C++
- Assembly
- Windows resources

55.2. Tool chain support

Gradle offers the ability to execute the same build using different tool chains. When you build a native binary, Gradle will attempt to locate a tool chain installed on your machine that can build the binary. You can fine tune exactly how this works, see Section 55.14, “Tool chains” for details.

The following tool chains are supported:

Operating System	Tool Chain	Notes
Linux	GCC	
Linux	Clang	
Mac OS X	XCode	Uses the Clang tool chain bundled with XCode.
Windows	Visual C++	Windows XP and later, Visual C++ 2010 and later.
Windows	GCC with Cywin 32	Windows XP and later.
Windows	GCC with MinGW	Windows XP and later. Mingw-w64 is currently not supported.

The following tool chains are unofficially supported. They generally work fine, but are not tested continuously:

Operating System	Tool Chain	Notes
Mac OS X	GCC from Macports	
Mac OS X	Clang from Macports	
Windows	GCC with Cywin 64	Windows XP and later.
UNIX-like	GCC	
UNIX-like	Clang	

55.3. Tool chain installation

Note that if you are using GCC then you currently need to install support for C++, even if you are not building from C++ source. This caveat will be removed in a future Gradle version.

To build native binaries, you will need to have a compatible tool chain installed:

55.3.1. Windows

To build on Windows, install a compatible version of Visual Studio. The native plugins will discover the Visual Studio installations and select the latest version. There is no need to mess around with environment variables or batch scripts. This works fine from a Cygwin shell or the Windows command-line.

Alternatively, you can install Cygwin with GCC or MinGW. Clang is currently not supported.

55.3.2. OS X

To build on OS X, you should install XCode. The native plugins will discover the XCode installation using the system PATH.

The native plugins also work with GCC and Clang bundled with Macports. To use one of the Macports tool chains, you will need to make the tool chain the default using the **`port select`** command and add Macports to the system PATH.

55.3.3. Linux

To build on Linux, install a compatible version of GCC or Clang. The native plugins will discover GCC or Clang using the system PATH.

55.4. Component model

To build native binaries using Gradle, your project should define one or more native components. Each component represents either an executable or a library that Gradle should build. A project can define any number of components. Gradle does not define any components by default.

For each component, Gradle defines a source set for each language that the component can be built from. A source set is essentially just a set of source directories containing source files. For example, when you apply the `c` plugin and define a library called `helloworld`, Gradle will define, by default, a source set containing the C source files in the `src/helloworld/c` directory. It will use these source files to build the `helloworld` library. This is described in more detail below.

For each component, Gradle defines one or more binaries as output. To build a binary, Gradle will take the source files defined for the component, compile them as appropriate for the source language, and link the result into a binary file. For an executable component, Gradle can produce executable binary files. For a library component, Gradle can produce both static and shared library binary files. For example, when you define a library called `helloworld` and build on Linux, Gradle will, by default, produce `libhelloworld.so` and `libhelloworld.a` binaries.

In many cases, more than one binary can be produced for a component. These binaries may vary based on the tool chain used to build, the compiler/linker flags supplied, the dependencies provided, or additional source files provided. Each native binary produced for a component is referred to as variant. Binary variants are discussed in detail below.

55.5. Building a library

To build either a static or shared native library, you define a library component in the `libraries` container. The following sample defines a library called `hello`:

Example 55.1. Defining a library component

build.gradle

```
libraries {
    hello {}
}
```

A library component is represented using `NativeLibrarySpec`. Each library component can produce at least one shared library binary (`SharedLibraryBinarySpec`) and at least one static library binary (`StaticLibraryBinarySpec`).

55.6. Building an executable

To build a native executable, you define an executable component in the `executables` container. The following sample defines an executable called `main`:

Example 55.2. Defining executable components

build.gradle

```
executables {
    main {}
}
```

An executable component is represented using `NativeExecutableSpec`. Each executable component can produce at least one executable binary (`NativeExecutableBinarySpec`).

For each component defined, Gradle adds a `FunctionalSourceSet` with the same name. Each of these functional source sets will contain a language-specific source set for each of the languages supported by the project.

55.7. Tasks

For each `NativeBinarySpec` that can be produced by a build, a single lifecycle task is constructed that can be used to create that binary, together with a set of other tasks that do the actual work of compiling, linking or assembling the binary.

Component Type	Native Binary Type	Lifecycle task	Locatio binary
NativeExecutableSpec	NativeExecutableBinarySpec	*${component.name}*Exe[illegible]	${[illegible]
NativeLibrarySpec	SharedLibraryBinarySpec	*${component.name}*Sha[illegible]	${[illegible]
NativeLibrarySpec	StaticLibraryBinarySpec	*${component.name}*Sta[illegible]	${[illegible]

55.7.1. Working with shared libraries

For each executable binary produced, the `cpp` plugin provides an `install${binary.name}` task, which creates a development install of the executable, along with the shared libraries it requires. This allows you to run the executable without needing to install the shared libraries in their final locations.

55.8. Finding out more about your project

Gradle provides a report that you can run from the command-line that shows some details about the components and binaries that your project produces. To use this report, just run **`gradle compo`** . Below is an example of running this report for one of the sample projects:

Example 55.3. The components report

gradle components の出力

```
> gradle components
:components

------------------------------------------------------------
Root project
------------------------------------------------------------

Native library 'hello'
----------------------

Source sets
    C++ source 'hello:cpp'
        src/hello/cpp

Binaries
    Shared library 'hello:sharedLibrary'
        build using task: :helloSharedLibrary
        platform: current
        build type: debug
        flavor: default
        tool chain: Tool chain 'clang' (Clang)
        shared library file: build/binaries/helloSharedLibrary/libhello.dylib
    Static library 'hello:staticLibrary'
        build using task: :helloStaticLibrary
        platform: current
        build type: debug
        flavor: default
        tool chain: Tool chain 'clang' (Clang)
        static library file: build/binaries/helloStaticLibrary/libhello.a

Native executable 'main'
------------------------

Source sets
    C++ source 'main:cpp'
        src/main/cpp

Binaries
    Executable 'main:executable'
        build using task: :mainExecutable
        platform: current
        build type: debug
        flavor: default
        tool chain: Tool chain 'clang' (Clang)
        executable file: build/binaries/mainExecutable/main

Note: currently not all plugins register their components, so some components may not

BUILD SUCCESSFUL

Total time: 1 secs
```

55.9. Language support

Presently, Gradle supports building native binaries from any combination of source languages listed below. A native binary project will contain one or more named `FunctionalSourceSet` instances (eg 'main', 'test', etc), each of which can contain `LanguageSourceSet`s containing source files, one for each language.

- C
- C++
- Objective-C
- Objective-C++
- Assembly
- Windows resources

55.9.1. C++ sources

C++ language support is provided by means of the `'cpp'` plugin.

Example 55.4. The 'cpp' plugin

build.gradle

```
apply plugin: 'cpp'
```

C++ sources to be included in a native binary are provided via a `CppSourceSet`, which defines a set of C++ source files and optionally a set of exported header files (for a library). By default, for any named component the `CppSourceSet` contains `.cpp` source files in `src/${name}/cpp`, and header files in `src/${name}/headers`.

While the `cpp` plugin defines these default locations for each `CppSourceSet`, it is possible to extend or override these defaults to allow for a different project layout.

Example 55.5. C++ source set

build.gradle

```
sources {
    main {
        cpp {
            source {
                srcDir "src/source"
                include "**/*.cpp"
            }
        }
    }
}
```

For a library named 'main', header files in `src/main/headers` are considered the “public” or “exported” headers. Header files that should not be exported should be placed inside the `src/ma` directory (though be aware that such header files should always be referenced in a manner

relative to the file including them).

55.9.2. C sources

C language support is provided by means of the `'c'` plugin.

Example 55.6. The 'c' plugin

build.gradle

```
apply plugin: 'c'
```

C sources to be included in a native binary are provided via a `CSourceSet`, which defines a set of C source files and optionally a set of exported header files (for a library). By default, for any named component the `CSourceSet` contains `.c` source files in `src/${name}/c`, and header files in `src/${name}/headers`.

While the `c` plugin defines these default locations for each `CSourceSet`, it is possible to extend or override these defaults to allow for a different project layout.

Example 55.7. C source set

build.gradle

```
sources {
    hello {
        c {
            source {
                srcDir "src/source"
                include "**/*.c"
            }
            exportedHeaders {
                srcDir "src/include"
            }
        }
    }
}
```

For a library named 'main', header files in `src/main/headers` are considered the "public" or "exported" headers. Header files that should not be exported should be placed inside the `src/ma` directory (though be aware that such header files should always be referenced in a manner relative to the file including them).

55.9.3. Assembler sources

Assembly language support is provided by means of the `'assembler'` plugin.

Example 55.8. The 'assembler' plugin

build.gradle

```
apply plugin: 'assembler'
```

Assembler sources to be included in a native binary are provided via a `AssemblerSourceSet`, which defines a set of Assembler source files. By default, for any named component the `AssemblerSourceSet` contains `.s` source files under `src/${name}/asm`.

55.9.4. Objective-C sources

Objective-C language support is provided by means of the `'objective-c'` plugin.

Example 55.9. The 'objective-c' plugin

build.gradle

```
apply plugin: 'objective-c'
```

Objective-C sources to be included in a native binary are provided via a `ObjectiveCSourceSet`, which defines a set of Objective-C source files. By default, for any named component the `ObjectiveCSourceSet` contains `.m` source files under `src/${name}/objectiveC`.

55.9.5. Objective-C++ sources

Objective-C++ language support is provided by means of the `'objective-cpp'` plugin.

Example 55.10. The 'objective-cpp' plugin

build.gradle

```
apply plugin: 'objective-cpp'
```

Objective-C++ sources to be included in a native binary are provided via a `ObjectiveCppSourceSet`, which defines a set of Objective-C++ source files. By default, for any named component the `ObjectiveCppSourceSet` contains `.mm` source files under `src/${name}/o`
.

55.10. Configuring the compiler, assembler and linker

Each binary to be produced is associated with a set of compiler and linker settings, which include command-line arguments as well as macro definitions. These settings can be applied to all binaries, an individual binary, or selectively to a group of binaries based on some criteria.

Example 55.11. Settings that apply to all binaries

build.gradle

```
binaries.all {
    // Define a preprocessor macro for every binary
    cppCompiler.define "NDEBUG"

    // Define toolchain-specific compiler and linker options
    if (toolChain in Gcc) {
        cppCompiler.args "-O2", "-fno-access-control"
        linker.args "-Xlinker", "-S"
    }
    if (toolChain in VisualCpp) {
        cppCompiler.args "/Zi"
        linker.args "/DEBUG"
    }
}
```

Each binary is associated with a particular `NativeToolChain`, allowing settings to be targeted based on this value.

It is easy to apply settings to all binaries of a particular type:

Example 55.12. Settings that apply to all shared libraries

build.gradle

```
// For any shared library binaries built with Visual C++,
// define the DLL_EXPORT macro
binaries.withType(SharedLibraryBinarySpec) {
    if (toolChain in VisualCpp) {
        cCompiler.args "/Zi"
        cCompiler.define "DLL_EXPORT"
    }
}
```

Furthermore, it is possible to specify settings that apply to all binaries produced for a particular `exec` or `library` component:

Example 55.13. Settings that apply to all binaries produced for the 'main' executable component

build.gradle

```
executables {
    main {
        binaries.all {
            if (toolChain in VisualCpp) {
                assembler.args "/Zi"
            } else {
                assembler.args "-g"
            }
        }
    }
}

sources {
    i386_masm {
        asm(AssemblerSourceSet) {
            source.srcDir "src/main/asm_i386_masm"
        }
    }
    i386_gcc {
        asm(AssemblerSourceSet) {
            source.srcDir "src/main/asm_i386_gcc"
        }
    }
}
```

The example above will apply the supplied configuration to all `executable` binaries built.

Similarly, settings can be specified to target binaries for a component that are of a particular type: eg all shared libraries for the main library component.

Example 55.14. Settings that apply only to shared libraries produced for the 'main' library componen

build.gradle

```
libraries {
    main {
        binaries.withType(SharedLibraryBinarySpec) {
            // Define a preprocessor macro that only applies to shared libraries
            cppCompiler.define "DLL_EXPORT"
        }
    }
}
```

55.11. Windows Resources

When using the `VisualCpp` tool chain, Gradle is able to compile Window Resource (`rc`) files and link them into a native binary. This functionality is provided by the `'windows-resources'` plugin.

Example 55.15. The 'windows-resources' plugin

build.gradle

```
apply plugin: 'windows-resources'
```

Windows resources to be included in a native binary are provided via a `WindowsResourceSet`, which defines a set of Windows Resource source files. By default, for any named component the `WindowsResourceSet` contains `.rc` source files under `src/${name}/rc`.

As with other source types, you can configure the location of the windows resources that should be included in the binary.

Example 55.16. Configuring the location of Windows resource sources

build-resource-only-dll.gradle

```
sources {
    helloRes {
        rc {
            source {
                srcDirs "src/hello/rc"
            }
            exportedHeaders {
                srcDirs "src/hello/headers"
            }
        }
    }
}
```

You are able to construct a resource-only library by providing Windows Resource sources with no other language sources, and configure the linker as appropriate:

Example 55.17. Building a resource-only dll

build-resource-only-dll.gradle

```
libraries {
    helloRes {
        binaries.all {
            rcCompiler.args "/v"
            linker.args "/noentry", "/machine:x86"
        }
    }
}
```

The example above also demonstrates the mechanism of passing extra command-line arguments to the resource compiler. The `rcCompiler` extension is of type `PreprocessingTool`.

55.12. Library Dependencies

Dependencies for native components are binary libraries that export header files. The header files are used during compilation, with the compiled binary dependency being used during linking and execution.

55.12.1. Dependencies within the same project

A set of sources may depend on header files provided by another binary component within the same project. A common example is a native executable component that uses functions provided by a separate native library component.

Such a library dependency can be added to a source set associated with the `executable` component:

Example 55.18. Providing a library dependency to the source set

build.gradle

```
sources {
    main {
        cpp {
            lib libraries.hello
        }
    }
}
```

Alternatively, a library dependency can be provided directly to the `NativeExecutableBinary` for the `executable`.

Example 55.19. Providing a library dependency to the binary

build.gradle

```
executables {
    main {
        binaries.all {
            // Each executable binary produced uses the 'hello' static library binary
            lib libraries.hello.static
        }
    }
}
```

55.12.2. Project Dependencies

For a component produced in a different Gradle project, the notation is similar.

Example 55.20. Declaring project dependencies

build.gradle

```
project(":lib") {
    apply plugin: "cpp"
    libraries {
        main {}
    }
    // For any shared library binaries built with Visual C++,
    // define the DLL_EXPORT macro
    binaries.withType(SharedLibraryBinarySpec) {
        if (toolChain in VisualCpp) {
            cppCompiler.define "DLL_EXPORT"
        }
    }
}

project(":exe") {
    apply plugin: "cpp"

    executables {
        main {}
    }

    sources {
        main {
            cpp {
                lib project: ':lib', library: 'main'
            }
        }
    }
}
```

55.13. Native Binary Variants

For each executable or library defined, Gradle is able to build a number of different native binary variants. Examples of different variants include debug vs release binaries, 32-bit vs 64-bit binaries, and binaries produced with different custom preprocessor flags.

Binaries produced by Gradle can be differentiated on build type, platform, and flavor. For each of these 'variant dimensions', it is possible to specify a set of available values as well as target each component at one, some or all of these. For example, a plugin may define a range of support platforms, but you may choose to only target Windows-x86 for a particular component.

55.13.1. Build types

A `build type` determines various non-functional aspects of a binary, such as whether debug information is included, or what optimisation level the binary is compiled with. Typical build types are 'debug' and 'release', but a project is free to define any set of build types.

Example 55.21. Defining build types

build.gradle

```
model {
    buildTypes {
        debug
        release
    }
}
```

If no build types are defined in a project, then a single, default build type called 'debug' is added.

For a build type, a Gradle project will typically define a set of compiler/linker flags per tool chain.

Example 55.22. Configuring debug binaries

build.gradle

```
binaries.all {
    if (toolChain in Gcc && buildType == buildTypes.debug) {
        cppCompiler.args "-g"
    }
    if (toolChain in VisualCpp && buildType == buildTypes.debug) {
        cppCompiler.args '/Zi'
        cppCompiler.define 'DEBUG'
        linker.args '/DEBUG'
    }
}
```

At this stage, it is completely up to the build script to configure the relevant compiler/linker flags for each build type. Future versions of Gradle will automatically include the appropriate debug flags for any 'debug' build type, and may be aware of various levels of optimisation as well.

55.13.2. Platform

An executable or library can be built to run on different operating systems and cpu architectures, with a variant being produced for each platform. Gradle defines each OS/architecture combination as a `NativePlatform`, and a project may define any number of platforms. If no platforms are defined in a project, then a single, default platform 'current' is added.

Presently, a `Platform` consists of a defined operating system and architecture. As we continue to develop the native binary support in Gradle, the concept of Platform will be extended to include things like C-runtime version, Windows SDK, ABI, etc. Sophisticated

builds may use the extensibility of Gradle to apply additional attributes to each platform, which can then be queried to specify particular includes, preprocessor macros or compiler arguments for a native binary.

Example 55.23. Defining platforms

build.gradle

```
model {
    platforms {
        x86 {
            architecture "x86"
        }
        x64 {
            architecture "x86_64"
        }
        itanium {
            architecture "ia-64"
        }
    }
}
```

For a given variant, Gradle will attempt to find a `NativeToolChain` that is able to build for the target platform. Available tool chains are searched in the order defined. See the tool chains section below for more details.

55.13.3. Flavor

Each component can have a set of named `flavors`, and a separate binary variant can be produced for each flavor. While the `build type` and `target platform` variant dimensions have a defined meaning in Gradle, each project is free to define any number of flavors and apply meaning to them in any way.

An example of component flavors might differentiate between 'demo', 'paid' and 'enterprise' editions of the component, where the same set of sources is used to produce binaries with different functions.

Example 55.24. Defining flavors

build.gradle

```
model {
    flavors {
        english
        french
    }
}

libraries {
    hello {
        binaries.all {
            if (flavor == flavors.french) {
                cppCompiler.define "FRENCH"
            }
        }
    }
}
```

In the example above, a library is defined with a 'english' and 'french' flavor. When compiling the 'french' variant, a separate macro is defined which leads to a different binary being produced.

If no flavor is defined for a component, then a single default flavor named 'default' is used.

55.13.4. Selecting the build types, platforms and flavors for a component

For a default component, Gradle will attempt to create a native binary variant for each and every combination of `buildType`, `platform` and `flavor` defined for the project. It is possible to override this on a per-component basis, by specifying the set of `targetBuildTypes`, `targetPlatf` and/or `targetFlavors`.

Example 55.25. Targeting a component at particular platforms

build.gradle

```
executables {
    main {
        targetPlatforms "x86", "x64"
    }
}
```

Here you can see that the `TargetedNativeComponent.targetPlatforms()` method is used to select the set of platforms to target for `executables.main`.

A similar mechanism exists for selecting `TargetedNativeComponent.targetBuildTypes()` and `TargetedNativeComponent.targetFlavors()`.

55.13.5. Building all possible variants

When a set of build types, target platforms, and flavors is defined for a component, a `NativeBinarySpec` model element is created for every possible combination of these. However, in many cases it is not possible to build a particular variant, perhaps because no tool chain is available to build for a particular platform.

If a binary variant cannot be built for any reason, then the `NativeBinarySpec` associated with that variant will not be `buildable`. It is possible to use this property to create a task to generate all possible variants on a particular machine.

Example 55.26. Building all possible variants

build.gradle

```
task buildAllExecutables {
    dependsOn binaries.withType(NativeExecutableBinary).matching {
        it.buildable
    }
}
```

55.14. Tool chains

A single build may utilize different tool chains to build variants for different platforms. To this end, the core 'native-binary' plugins will attempt to locate and make available supported tool chains. However, the set of tool chains for a project may also be explicitly defined, allowing additional cross-compilers to be configured as well as allowing the install directories to be specified.

55.14.1. Defining tool chains

The supported tool chain types are:

- `Gcc`
- `Clang`
- `VisualCpp`

Example 55.27. Defining tool chains

build.gradle

```
model {
    toolChains {
        visualCpp(VisualCpp) {
            // Specify the installDir if Visual Studio cannot be located
            // installDir "C:/Apps/Microsoft Visual Studio 10.0"
        }
        gcc(Gcc) {
            // Uncomment to use a GCC install that is not in the PATH
            // path "/usr/bin/gcc"
        }
        clang(Clang)
    }
}
```

Each tool chain implementation allows for a certain degree of configuration (see the API documentation for more details).

55.14.2. Using tool chains

It is not necessary or possible to specify the tool chain that should be used to build. For a given variant, Gradle will attempt to locate a `NativeToolChain` that is able to build for the target platform. Available tool chains are searched in the order defined.

> When a platform does not define an architecture or operating system, the default target of the tool chain is assumed. So if a platform does not define a value for `operatingSystem`, Gradle will find the first available tool chain that can build for the specified `architecture`.

The core Gradle tool chains are able to target the following architectures out of the box. In each case, the tool chain will target the current operating system. See the next section for information on cross-compiling for other operating systems.

Tool Chain	Architectures
GCC	x86, x86_64
Clang	x86, x86_64
Visual C++	x86, x86_64, ia-64

So for GCC running on linux, the supported target platforms are 'linux/x86' and 'linux/x86_64'. For GCC running on Windows via Cygwin, platforms 'windows/x86' and 'windows/x86_64' are supported. (The Cygwin POSIX runtime is not yet modelled as part of the platform, but will be in the future.)

If no target platforms are defined for a project, then all binaries are built to target a default platform named 'current'. This default platform does not specify any `architecture` or `operatingS`
value, hence using the default values of the first available tool chain.

Gradle provides a hook that allows the build author to control the exact set of arguments passed to a tool chain executable. This enables the build author to work around any limitations in Gradle, or assumptions that Gradle makes. The arguments hook should be seen as a 'last-resort' mechanism, with preference given to truly modelling the underlying domain.

Example 55.28. Reconfigure tool arguments

build.gradle

```
model {
    toolChains {
        visualCpp(VisualCpp) {
            eachPlatform {
                cppCompiler.withArguments { args ->
                    args << "-DFRENCH"
                }
            }
        }
        clang(Clang) {
            eachPlatform {
                cCompiler.withArguments { args ->
                    Collections.replaceAll(args, "CUSTOM", "-DFRENCH")
                }
                linker.withArguments { args ->
                    args.remove "CUSTOM"
                }
                staticLibArchiver.withArguments { args ->
                    args.remove "CUSTOM"
                }
            }
        }
    }
}
```

55.14.3. Cross-compiling with GCC

Cross-compiling is possible with the `Gcc` and `Clang` tool chains, by adding support for additional target platforms. This is done by specifying a target platform for a toolchain. For each target platform a custom configuration can be specified.

Example 55.29. Defining target platforms

build.gradle

```
model {
    toolChains {
        gcc(Gcc) {
            target("arm"){
                cppCompiler.withArguments { args ->
                    args << "-m32"
                }
                linker.withArguments { args ->
                    args << "-m32"
                }
            }
            target("sparc")
        }
    }
    platforms {
        arm {
            architecture "arm"
        }
        sparc {
            architecture "sparc"
        }
    }
}
```

55.15. Visual Studio IDE integration

Gradle has the ability to generate Visual Studio project and solution files for the native components defined in your build. This ability is added by the `visual-studio` plugin. For a multi-project build, all projects with native components should have this plugin applied.

When the `visual-studio` plugin is applied, a task name `${component.name}VisualStudio` is created for each defined component. This task will generate a Visual Studio Solution file for the named component. This solution will include a Visual Studio Project for that component, as well as linking to project files for each depended-on binary.

The content of the generated visual studio files can be modified via API hooks, provided by the `visu` extension. Take a look at the 'visual-studio' sample, or see `VisualStudioExtension.getProjects()` and `VisualStudioExtension.getSolutions()` in the API documentation for more details.

55.16. CUnit support

The Gradle `cunit` plugin provides support for compiling and executing CUnit tests in your native-binary project. For each `NativeExecutableSpec` and `NativeLibrarySpec` defined in your project, Gradle will create a matching `CUnitTestSuiteSpec` component, named `${componen`
.

55.16.1. CUnit sources

Gradle will create a `CSourceSet` named 'cunit' for each `CUnitTestSuiteSpec` component in the project. This source set should contain the cunit test files for the component sources. Source files can be located in the conventional location (`src/${component.name}Test/cunit`) or can be configured like any other source set.

Gradle initialises the CUnit test registry and executes the tests, utilising some generated CUnit launcher sources. Gradle will expect and call a function with the signature `void gradle_cunit_re` that you can use to configure the actual CUnit suites and tests to execute.

Example 55.30. Registering CUnit tests

suite_operators.c

```
#include <CUnit/Basic.h>
#include "gradle_cunit_register.h"
#include "test_operators.h"

int suite_init(void) {
    return 0;
}

int suite_clean(void) {
    return 0;
}

void gradle_cunit_register() {
    CU_pSuite pSuiteMath = CU_add_suite("operator tests", suite_init, suite_clean)
    CU_add_test(pSuiteMath, "test_plus", test_plus);
    CU_add_test(pSuiteMath, "test_minus", test_minus);
}
```

> Due to this mechanism, your CUnit sources may not contain a `main` method since this will clash with the method provided by Gradle.

55.16.2. Building CUnit executables

A `CUnitTestSuiteSpec` component has an associated `NativeExecutableSpec` or `NativeLibrarySpec` component. For each `NativeBinarySpec` configured for the main component, a matching `CUnitTestSuiteBinarySpec` will be configured on the test suite component. These test suite binaries can be configured in a similar way to any other binary instance:

Example 55.31. Registering CUnit tests

build.gradle

```
binaries.withType(CUnitTestSuiteBinarySpec) {
    lib library: "cunit", linkage: "static"

    if (flavor == flavors.failing) {
        cCompiler.define "PLUS_BROKEN"
    }
}
```

Both the CUnit sources provided by your project and the generated launcher require the core CUnit headers and libraries. Presently, this library dependency must be provided by your project for each `CUnitTestSuiteBinarySpec`.

55.16.3. Running CUnit tests

For each `CUnitTestSuiteBinarySpec`, Gradle will create a task to execute this binary, which will run all of the registered CUnit tests. Test results will be found in the `${build.dir}/test-resu` directory.

Example 55.32. Running CUnit tests

build.gradle

```
apply plugin: "c"
apply plugin: "cunit"

model {
    flavors {
        passing
        failing
    }
    repositories {
        libs(PrebuiltLibraries) {
            cunit {
                headers.srcDir "lib/cunit/2.1-2/include"
                binaries.withType(StaticLibraryBinary) {
                    staticLibraryFile =
                        file("lib/cunit/2.1-2/lib/" +
                             findCUnitLibForPlatform(targetPlatform))
                }
            }
        }
    }
}

libraries {
    operators {}
}
binaries.withType(CUnitTestSuiteBinarySpec) {
    lib library: "cunit", linkage: "static"

    if (flavor == flavors.failing) {
        cCompiler.define "PLUS_BROKEN"
    }
}
```

ノート：

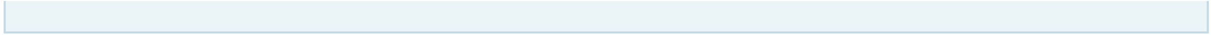
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所 **samples/native-binaries/cunit**

gradle -q runFailingOperatorsTestCUnitExe の出力

```
> gradle -q runFailingOperatorsTestCUnitExe
There were test failures:
  1. /home/user/gradle/samples/native-binaries/cunit/src/operatorsTest/c/test_plus.c:
  2. /home/user/gradle/samples/native-binaries/cunit/src/operatorsTest/c/test_plus.c:
```

The current support for CUnit is quite rudimentary. Plans for future integration include:

- Allow tests to be declared with Javadoc-style annotations.
- Improved HTML reporting, similar to that available for JUnit.
- Real-time feedback for test execution.
- Support for additional test frameworks.



56
ビルドのライフサイクル

前述の通り、Gradleというのは言ってしまえば依存性を記述する言語です。Gradleの用語でいえば、タス
無 閉 路 有 向 グ ラ フ (D A G)
を構築するのです。タスクを実行するときにこのようなタスクグラフを組み立てるビルドツールもありま

ビルドスクリプトは、このタスクグラフを定義します。したがって、ビルドスクリプトとは、厳密には
ビルド設定スクリプトだと言えるでしょう。

56.1. ビルドフェーズ

Gradleには、独立した三つのビルドフェーズがあります。

初期化
: Gradleはシングルプロジェクト、マルチプロジェクトの双方をサポートします。初期化フェーズでは
`Project`インスタンスを生成します。

設定
: プロジェクトのオブジェクトが設定されるフェーズです。ビルドに含まれるすべて
の プ ロ ジ ェ ク ト の ビ ル ド ス ク リ プ ト が 実 行 さ れ ま す 。
Gradle1.4で、「オンデマンド設定」という機能が試験的に導入されました。
オンデマンド設定モードでは、Gradleは全てのプロジェクトではなく、実行時に必要なプロジェクト

実行
: 設定フェーズで初期化と設定が完了したタスクのうち、実行するべきタスクを抽出して実行します。
gradleコマンドに引き渡されたタスク名と、コマンドの実行ディレクトリから決定されます。

56.2. 設定ファイル

ビルドスクリプトの他、Gradleではビルドの「設定ファイル」を作成することができます。設定ファイル
`s e t t i n g s . g r a d l e`
です。この設定ファイルをGradleがどのように探すのかについては、後ほど解説します。

設定ファイルは、初期化フェーズで使用されます。マルチプロジェクトの場合、`settings.gradle`
ファイルは必須です。ルートとなるプロジェクトに必ず配置しなければなりません。この設定ファイルに
5 7 章 マ ル チ プ ロ ジ ェ ク ト の ビ ル ド
参照)。シングルプロジェクトの場合は設定ファイルはなくてもかまいません。ビルドスクリプトのクラス
6 0 章 ビ ル ド ロ ジ ッ ク の 体 系 化
参照)。設定ファイルの読み込み、フェーズ実行の内部動作を、シングルプロジェクトの例を使って見て

例56.1 シングルプロジェクトのビルド

settings.gradle

```
println 'This is executed during the initialization phase.'
```

build.gradle

```
println 'This is executed during the configuration phase.'

task configured {
    println 'This is also executed during the configuration phase.'
}

task test << {
    println 'This is executed during the execution phase.'
}

task testBoth {
    doFirst {
      println 'This is executed first during the execution phase.'
    }
    doLast {
      println 'This is executed last during the execution phase.'
    }
    println 'This is executed during the configuration phase as well.'
}
```

gradle test testBoth の出力

```
> gradle test testBoth
This is executed during the initialization phase.
This is executed during the configuration phase.
This is also executed during the configuration phase.
This is executed during the configuration phase as well.
:test
This is executed during the execution phase.
:testBoth
This is executed first during the execution phase.
This is executed last during the execution phase.

BUILD SUCCESSFUL

Total time: 1 secs
```

ビルドスクリプト内でのプロパティアクセスやメソッド呼び出しは、Projectインスタンスに委譲されま
Settingsをご参照ください。

56.3. マルチプロジェクトのビルド

マルチプロジェクトとは、ある一つのプロジェクトをビルドしているときに、関連する別のプロジェクト
57章マルチプロジェクトのビルド参照)。

56.3.1. プロジェクトの配置

マルチプロジェクトは、常に、あるプロジェクトをルートにしたプロジェクトのツリー構造となります。
マルチプロジェクト・ツリーを構成するそれぞれのプロジェクトは、自分がツリー内のどこに位置してい
ほとんどのケースで、そのパス情報は、ファイルシステム上でのプロジェクトの物理的な位置を反映した
ただし、これは設定で変更することも可能です。 プロジェクト・ツリーは、`settings.gradle`
ファイルで作成します。 デフォルトでは、`settings.gradle`
ファイルがあるプロジェクトがルートプロジェクトになっていますが、ルートプロジェクトの位置もその

56.3.2. プロジェクト・ツリーの構築

設定ファイル内では、プロジェクト・ツリーを構築するためのメソッドを使用できます。これらのメソッ

56.3.2.1. 階層構造のレイアウト

例56.2 階層構造のレイアウト

settings.gradle

```
include 'project1', 'project2:child', 'project3:child1'
```

`include`メソッドは、プロジェクトのパスを引数にとります。
このパスは、ファイルシステム上の物理的な相対パスを表したものです。
例えば、'services:api'というパスは、デフォルトでは'services/api'フォルダ(プロジェクトルートからの相
指定するのは、ツリーの末端となるプロジェクトだけで構いません。
つまり、'services:hotels:api'を指定しただけで3つのプロジェクトが作成されるということです('services

56.3.2.2. フラットなレイアウト

例56.3 フラットなレイアウト

settings.gradle

```
includeFlat 'project3', 'project4'
```

`includeFlat`
メソッドは、ディレクトリ名を引数にとります。これらのディレクトリは、ルートプロジェクトのディレ

56.3.3. プロジェクト・ツリーの属性を変更する

設定ファイルで定義したプロジェクト・ツリーは、プロジェクトディスクリプタ
と呼ばれる属性を持っています。これらの属性は設定ファイルの中でいつでも変更することができます。

例56.4 プロジェクト・ツリーの属性を変更する

settings.gradle

```
println rootProject.name
println project(':projectA').name
```

ディスクリプタを使用して、プロジェクト名やプロジェクトディレクトリ、プロジェクトのビルドスクリ

例56.5 プロジェクトツリーの属性を変更する

settings.gradle

```
rootProject.name = 'main'
project(':projectA').projectDir = new File(settingsDir, '../my-project-a')
project(':projectA').buildFileName = 'projectA.gradle'
```

詳細については`ProjectDescriptor`をご参照ください。

56.4. 初期化

Gradleは、現在ビルドしているものがマルチプロジェクトかシングルプロジェクトか、どうやって知るの
[31]
Gradleは、`settings.gradle`
のないディレクトリで実行された場合、以下のように動作します。

- カレントディレクトリと同階層のディレクトリから`master`という名前のディレクトリを探す。
- 見つからなければ、親階層のディレクトリをたどっていって探す。
- 見つからなければ、シングルプロジェクトとしてビルドする。
- `settings.gradle`
  が見つかれば、Gradleはビルドを実行したプロジェクトがマルチプロジェクトに参加するよう宣言さ

なんのためにこのような振る舞いをするのでしょう。今からビルドするプロジェクトは、マルチプロジェ
57章 マルチプロジェクトのビルド参照)。`-u`
オプションをつけてGradleを実行すれば、親階層をたどって`settings.gradle`
を探しに行くことはありません。つまり、(settings.gradleのない)現在ビルドしているプロジェクトは、
`settings.gradle`がある場合は、`-u`
をつけても意味はありません。settings.gradleをもつプロジェクトは以下のように処理されます。

- `settings.gradle`
  がマルチプロジェクト・ツリーを定義していない場合はシングルプロジェクトとしてビルド
- `settings.gradle`
  がマルチプロジェクト・ツリーを定義している場合は、マルチプロジェクトとしてビルド

この設定ファイルの探索処理は、プロジェクトのディレクトリ構造が前述した階層レイアウトとフラット
「絶対パスによるタスクの実行」
をご参照ください。次のリリースでは、コマンドラインから設定ファイルの場所を指定してやることで、

初期化フェーズでは、ビルドに参加しているすべてのプロジェクトについてProjectオブジェクトが作成さ

Gradle creates a Project object for every project taking part in the build. For a multi-project build these are the projects specified in the Settings object (plus the root project). Each project object has by default a name equal to the name of its top level directory, and every project except the root project has a parent project. Any project may have child projects.

56.5. シングルプロジェクトの設定と実行

シングルプロジェクトの場合、初期化フェーズ完了後
の処理の流れは非常にシンプルです。まず、初期化フェーズで作成されたプロジェクトのビルドスクリプ
57章マルチプロジェクトのビルドを参照してください。

56.6. ライフサイクルからの通知に応答する

ビルドスクリプト内では、ビルドがそのライフサイクルに従って進行していくなかで、さまざまな通知を

56.6.1. プロジェクトの評価

プロジェクトが評価される前またはされた後すぐに通知を受け取ることができます。これは、ビルドスク

以下の例は、testタスクをhasTests
プロパティを持つすべてのプロジェクトに追加するものです。

例56.6 特定プロパティを持つプロジェクトにタスクを追加する

build.gradle

```
allprojects {
    afterEvaluate { project ->
        if (project.hasTests) {
            println "Adding test task to $project"
            project.task('test') << {
                println "Running tests for $project"
            }
        }
    }
}
```

projectA.gradle

```
hasTests = true
```

gradle -q test の出力

```
> gradle -q test
Adding test task to project ':projectA'
Running tests for project ':projectA'
```

この例では、Project.afterEvaluate()
を使用して、プロジェクト評価後に実行されるクロージャを追加しています。

プロジェクト評価時の通知は、どんなプロジェクトに対しても受け取ることができます。次の例では、ナ
afterProject
が、プロジェクトの評価に成功しても例外で失敗しても通知される点に注目してください。

例56.7 通知

build.gradle

```
gradle.afterProject {project, projectState ->
    if (projectState.failure) {
        println "Evaluation of $project FAILED"
    } else {
        println "Evaluation of $project succeeded"
    }
}
```

gradle -q test の出力

```
> gradle -q test
Evaluation of root project 'buildProjectEvaluateEvents' succeeded
Evaluation of project ':projectA' succeeded
Evaluation of project ':projectB' FAILED
```

`ProjectEvaluationListener`を`Gradle`に追加することでも同様の処理を実現できます。

56.6.2. タスク作成

プロジェクトにタスクが作成されたときに通知を受け取ることができます。この通知は、タスクが使用さ

次の例は、作成されたタスクに、随時`srcDir`プロパティを追加していくものです。

例56.8 すべてのタスクにプロパティ値を設定する

build.gradle

```
tasks.whenTaskAdded { task ->
    task.ext.srcDir = 'src/main/java'
}

task a

println "source dir is $a.srcDir"
```

gradle -q a の出力

```
> gradle -q a
source dir is src/main/java
```

`Action`を`TaskContainer`に追加することでも同様の処理を実現できます。

56.6.3. タスク実行グラフの準備

タスクの実行グラフが用意できたときに通知を受け取ることができます。この例については、「DAGによる設定」で紹介しました。

`TaskExecutionGraphListener`を`TaskExecutionGraph`
に追加することでも同様の処理を実現できます。

56.6.4. タスクの実行

タスクが実行される前または実行された後に通知を受け取ることができます。

次の例では、タスクの実行が開始されたときと終わったときにそれぞれロギングを追加しています。aft が、タスクの実行に成功しても例外で失敗しても通知されていることに注目してください。

例56.9 タスク実行の開始時および終了時にロギングを行う

build.gradle

```
task ok

task broken(dependsOn: ok) << {
    throw new RuntimeException('broken')
}

gradle.taskGraph.beforeTask { Task task ->
    println "executing $task ..."
}

gradle.taskGraph.afterTask { Task task, TaskState state ->
    if (state.failure) {
        println "FAILED"
    }
    else {
        println "done"
    }
}
```

gradle -q broken の出力

```
> gradle -q broken
executing task ':ok' ...
done
executing task ':broken' ...
FAILED
```

`TaskExecutionListener`を`TaskExecutionGraph`
に追加することでも同様の処理を実現できます。

---

[31] Gradleは、マルチプロジェクトに含まれる一部のプロジェクトのみ部分的にビルドできる(57章 マルチプロジェクトのビルド参照)

57
マルチプロジェクトのビルド

マルチプロジェクトはビルドツールの知恵の絞りどころであり、とてもやりがいのあるテーマです。Gra

A multi-project build in gradle consists of one root project, and one or more subprojects that may also have subprojects.

57.1. クロスプロジェクト設定

While each subproject could configure itself in complete isolation of the other subprojects, it is common that subprojects share common traits. It is then usually preferable to share configurations among projects, so the same configuration affects several subprojects.

まず、ごく簡単な例から始めましょう。なんと言ってもGradleは汎用のビルドツールですので、扱うプロ
これから例として使用するプロジェクトは、海洋生物に関するものです。

57.1.1. Configuration and execution

「ビルドフェーズ」 describes the phases of every Gradle build. Let's zoom into the configuration and execution phases of a multi-project build. Configuration here means executing the `build.grad` file of a project, which implies e.g. downloading all plugins that were declared using '`apply plugin` '. By default, the configuration of all projects happens before any task is executed. This means that when a single task, from a single project is requested, all projects of multi-project build are configured first. The reason every project needs to be configured is to support the flexibility of accessing and changing any part of the Gradle project model.

57.1.1.1. Configuration on demand

The Configuration injection feature and access to the complete project model are possible because every project is configured before the execution phase. Yet, this approach may not be the most efficient in a very large multi-project build. There are Gradle builds with a hierarchy of hundreds of subprojects. The configuration time of huge multi-project builds may become noticeable. Scalability is an important requirement for Gradle. Hence, starting from version 1.4 a new incubating 'configuration on demand' mode is introduced.

Configuration on demand mode attempts to configure only projects that are relevant for requested tasks, i.e. it only executes the `build.gradle` file of projects that are participating in the build. This way, the configuration time of a large multi-project build can be reduced. In the long term, this mode will become the default mode, possibly the only mode for Gradle build execution. The configuration on demand feature is incubating so not every build is guaranteed

to work correctly. The feature should work very well for multi-project builds that have decoupled projects (「分離されたプロジェクト 」). In “configuration on demand” mode, projects are configured as follows:

- The root project is always configured. This way the typical common configuration is supported (allprojects or subprojects script blocks).
- The project in the directory where the build is executed is also configured, but only when Gradle is executed without any tasks. This way the default tasks behave correctly when projects are configured on demand.
- The standard project dependencies are supported and makes relevant projects configured. If project A has a compile dependency on project B then building A causes configuration of both projects.
- The task dependencies declared via task path are supported and cause relevant projects to be configured. Example: someTask.dependsOn(":someOtherProject:someOtherTask")
- A task requested via task path from the command line (or Tooling API) causes the relevant project to be configured. For example, building 'projectA:projectB:someTask' causes configuration of projectB.

Eager to try out this new feature? To configure on demand with every build run see 「gradle.propertiesを使用したビルド環境の構築」. To configure on demand just for a given build please see 付録D Gradle コマンドライン.

57.1.2. 共通の振る舞いを定義する

次のプロジェクトツリーにそってサンプルを見てみましょう。 これはルートプロジェクト「`water`」と、サブプロジェクトの「`bluewhale`(シロナガスクジラ)」からなるマルチプロジェクトです。

例57.1 マルチプロジェクト・ツリー - warter & bluewhale プロジェクト

Build layout

```
water/
  build.gradle
  settings.gradle
  bluewhale/
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
`samples/userguide/multiproject/firstExample/water`

`settings.gradle`

```
include 'bluewhale'
```

で、bluewhale
プロジェクトのビルドスクリプトは何処にあるのでしょう。Gradleでは、ビルドスクリプトは必須ではあ
`water`のビルドスクリプトを見てみましょう。

例57.2 water(親プロジェクト)のビルドスクリプト

build.gradle

```
Closure cl = { task -> println "I'm $task.project.name" }
task hello << cl
project(':bluewhale') {
    task hello << cl
}
```

gradle -q hello の出力

```
> gradle -q hello
I'm water
I'm bluewhale
```

Gradleでは、マルチプロジェクトを構成するすべてのプロジェクトに、どのビルドスクリプトからでもア
p r o j e c t ( )
というメソッドを使用でき、引数にプロジェクトへのパスを渡せばProjectオブジェクトを取得できます。
クロスプロジェクト設定と呼んでいます。Gradleは、この機能を設定のインジェクション
を利用して実現します。

w a t e r
プロジェクトのビルドスクリプトは、まだ満足できるものではありません。すべてのプロジェクトに手て
krill()プロジェクトをマルチプロジェクトに追加してみましょう。

例57.3 マルチプロジェクトツリー - water, bluewhaleそしてkrill

Build layout

```
water/
  build.gradle
  settings.gradle
  bluewhale/
  krill/
```

ノ ー ト :
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/userguide/multiproject/addKrill/water

settings.gradle

```
include 'bluewhale', 'krill'
```

waterビルドスクリプトを書き直し、一行で記述するようにします。

例57.4 Waterプロジェクトのビルドスクリプト

build.gradle

```
allprojects {
    task hello << { task -> println "I'm $task.project.name" }
}
```

gradle -q hello の出力

```
> gradle -q hello
I'm water
I'm bluewhale
I'm krill
```

素晴らしい。これはどういう原理で動いているのでしょう。プロジェクトAPIは`allprojects`というプロパティを提供しており、そのプロパティには現在のプロジェクトと、そのサブプロジェクトの
`allprojects`をクロージャとともに呼び出せば、クロージャの実行が`allprojects`に格納されたすべてのプロジェクトに委譲されます。もちろん`allprojects`をeachでループすることもできますが、記述が冗長になってしまいます。

他のビルドシステムでは、プロジェクト間の共通設定を行う方法として主に継承を使用します。後ほど

Another possibilty for sharing configuration is to use a common external script. See 「外部のビルドスクリプトをプロジェクトに取り込む」 for more information.

57.2. サブプロジェクトの設定

プロジェクトAPIはサブプロジェクトのみにアクセスする方法も提供します。

57.2.1. 共通の振る舞いを定義する

例57.5 サブプロジェクト共通の振る舞いとすべてのプロジェクト共通の振る舞いをそれぞれ定義する

build.gradle

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
```

gradle -q hello の出力

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
I'm krill
- I depend on water
```

You may notice that there are two code snippets referencing the “`hello`” task. The first one,

which uses the “task” keyword, constructs the task and provides it's base configuration. The second piece doesn't use the “task” keyword, as it is further configuring the existing “hello” task. You may only construct a task once in a project, but you may any number of code blocks providing additional configuration.

57.2.2. 個別の振る舞いを追加する

共通の振る舞いの上に、それぞれのプロジェクト個別の振る舞いを追加することができます。普通は、フ
bluewhaleに個別の振る舞いを追加することもできます。

例57.6 プロジェクト個別の振る舞いを定義する

build.gradle

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
project(':bluewhale').hello << {
    println "- I'm the largest animal that has ever lived on this planet."
}
```

gradle -q hello の出力

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
```

前述のとおり、プロジェクト個別の振る舞いは、普通そのプロジェクトのビルドスクリプトに記述します
krillプロジェクトにも振る舞いを追加してみましょう。

例57.7 krillプロジェクトに個別の振る舞いを定義する

Build layout

```
water/
  build.gradle
  settings.gradle
  bluewhale/
    build.gradle
  krill/
    build.gradle
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/userguide/multiproject/spreadSpecifics/water

settings.gradle

```
include 'bluewhale', 'krill'
```

bluewhale/build.gradle

```
hello.doLast {
  println "- I'm the largest animal that has ever lived on this planet."
}
```

krill/build.gradle

```
hello.doLast {
  println "- The weight of my species in summer is twice as heavy as all human beings
}
```

build.gradle

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
```

gradle -q hello の出力

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
```

57.2.3. プロジェクトのフィルタリング

設定のインジェクションが持つもっと大きな力をお見せするために、`tropicalFish()` プロジェクトを追加します。このプロジェクトに、`water` プロジェクトのビルドスクリプトから振る舞いを定義していきます。

57.2.3.1. 名前によるフィルタリング

例57.8 プロジェクトに振る舞いを追加する(プロジェクト名によるフィルタリング)

Build layout

```
water/
  build.gradle
  settings.gradle
  bluewhale/
    build.gradle
  krill/
    build.gradle
  tropicalFish/
```

ノート:

本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所 **samples/userguide/multiproject/addTropical/water**

settings.gradle

```
include 'bluewhale', 'krill', 'tropicalFish'
```

build.gradle

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
configure(subprojects.findAll {it.name != 'tropicalFish'}) {
    hello << {println '- I love to spend time in the arctic waters.'}
}
```

gradle -q hello の出力

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I love to spend time in the arctic waters.
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
- I love to spend time in the arctic waters.
- The weight of my species in summer is twice as heavy as all human beings.
I'm tropicalFish
- I depend on water
```

configure()は引数にリストを取るメソッドで、そのリストに格納されたプロジェクトに設定を適用します

57.2.3.2. プロパティによるフィルタリング

フィルタリングには、プロジェクト名だけでなく拡張プロパティを使うこともできます。(拡張プロパテ
「拡張プロパティ 」を参照してください。)

例57.9 プロジェクトに振る舞いを追加する(プロパティによるフィルタリング)

Build layout

```
water/
  build.gradle
  settings.gradle
  bluewhale/
    build.gradle
  krill/
    build.gradle
  tropicalFish/
    build.gradle
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/userguide/multiproject/tropicalWithProperties/water

settings.gradle

```
include 'bluewhale', 'krill', 'tropicalFish'
```

bluewhale/build.gradle

```
ext.arctic = true
hello.doLast {
  println "- I'm the largest animal that has ever lived on this planet."
}
```

krill/build.gradle

```
ext.arctic = true
hello.doLast {
    println "- The weight of my species in summer is twice as heavy as all human bein
}
```

tropicalFish/build.gradle

```
ext.arctic = false
```

build.gradle

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello {
        doLast {println "- I depend on water"}
        afterEvaluate { Project project ->
            if (project.arctic) { doLast {
                println '- I love to spend time in the arctic waters.' }
            }
        }
    }
}
```

gradle -q hello の出力

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
- I love to spend time in the arctic waters.
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

waterプロジェクトのビルドスクリプトでは、afterEvaluateという宣言文が使用されています。aft
に引き渡されたクロージャは、そのプロジェクトのビルドスクリプトが評価された後に
実行されます。サブプロジェクトのarctic
プロパティは、それぞれのプロジェクトのビルドスクリプト内でセットされているので、この方法でarc
プロパティにアクセスする必要があるのです。このトピックは「依存関係 - なんの依存関係？」
で詳しく説明します。

57.3. マルチプロジェクトのビルド実行ルール

helloタスクをルートプロジェクトのディレクトリで実行したとき、ビルドは直感に従う、自然な形で実行
bluewhaleディレクトリでGradleを実行した場合、どのようにビルドが実行されるのでしょうか。

例57.10 サブプロジェクトからビルドを実行する

gradle -q hello の出力

```
> gradle -q hello
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
- I love to spend time in the arctic waters.
```

Gradleがビルドを実行するときの基本的なルールは単純なものです。この例では、Gradleはhello
タスクを探して、カレントディレクトリからプロジェクト階層を降りていき、タスクを見つけるとそれを

すべてのプロジェクトを常に評価します。ビルドスクリプトに定義されているタスクオブジェクトは、すべて作成されるのです。その
marineプロジェクトのbluewhaleとkrillに新しいタスクを追加しましょう。

例57.11 プロジェクトの評価と実行

bluewhale/build.gradle

```
ext.arctic = true
hello << { println "- I'm the largest animal that has ever lived on this planet." }

task distanceToIceberg << {
    println '20 nautical miles'
}
```

krill/build.gradle

```
ext.arctic = true
hello << {
    println "- The weight of my species in summer is twice as heavy as all human bein
}

task distanceToIceberg << {
    println '5 nautical miles'
}
```

gradle -q distanceToIceberg の出力

```
> gradle -q distanceToIceberg
20 nautical miles
5 nautical miles
```

-qオプションを外すと、出力結果は以下のようになります。:

例57.12 プロジェクトの評価と実行

gradle distanceToIceberg の出力

```
> gradle distanceToIceberg
:bluewhale:distanceToIceberg
20 nautical miles
:krill:distanceToIceberg
5 nautical miles

BUILD SUCCESSFUL

Total time: 1 secs
```

このビルドはwaterプロジェクトのディレクトリで実行しています。waterプロジェクトもtropicalF
プロジェクトもdistanceToIcebergという名前のタスクは持っていませんが、Gradleはそんなこと気にし

57.4. 絶対パスによるタスクの実行

いままで見てきたように、マルチプロジェクトのサブプロジェクトは、プロジェクトのディレクトリに移
「プロジェクトとタスクのパス」もご参照ください)

例57.13 絶対パスによるタスクの実行

gradle -q :hello :krill:hello hello の出力

```
> gradle -q :hello :krill:hello hello
I'm water
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

ビルドはtropicalFish
プロジェクトのディレクトリで実行されています。上記の例では、ルートプロジェクトwatar、krill
プロジェクト、そしてカレントのサブプロジェクトtropicalFishのhello
がそれぞれ実行されます。

57.5. プロジェクトとタスクのパス

A project path has the following pattern: It starts with an optional colon, which denotes the root project. The root project is the only project in a path that is not specified by its name. The rest of a project path is a colon-separated sequence of project names, where the next project is a subproject of the previous project.

タスクのパスは、例えば :bluewhale:hello
のように、単純にプロジェクトパスにタスク名を加えるだけです。なお、カレントのプロジェクトにある

以前Gradleでは、プロジェクトのパスセパレータとして /
を使用していました。しかし、このセパレータ文字はdirectoryタスク(「ディレクトリの作成　」
参照)が導入されたときに廃止されています。directoryタスクのタスク名に、/
が含まれるためです。

57.6. 依存関係 - なんの依存関係？

先ほどまでの例は特殊なもので、プロジェクトの間で設定情報は依存関係がありましたが、実行に関して

57.6.1. 実行に関する依存関係

57.6.1.1. 依存関係と実行順序

例57.14 依存関係とビルド実行順序

Build layout

```
messages/
  settings.gradle
  consumer/
    build.gradle
  producer/
    build.gradle
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/userguide/multiproject/dependencies/firstMessages/messages

settings.gradle

```
include 'consumer', 'producer'
```

consumer/build.gradle

```
task action << {
    println("Consuming message: ${rootProject.producerMessage}")
}
```

producer/build.gradle

```
task action << {
    println "Producing message:"
    rootProject.producerMessage = 'Watch the order of execution.'
}
```

gradle -q action の出力

```
> gradle -q action
Consuming message: null
Producing message:
```

この例はうまく動作しません。依存関係が定義されていないと、Gradleはタスクをアルファベット順に実
したがって、`:consumer():action`が`:producer():action`より先に実行されてしまうのです。 `pro`
プロジェクトの名前を`aProducer`に修正すればこの問題を解決することはできます。

例57.15 依存関係とビルド実行順序

Build layout

```
messages/
  settings.gradle
  aProducer/
    build.gradle
  consumer/
    build.gradle
```

settings.gradle

```
include 'consumer', 'aProducer'
```

aProducer/build.gradle

```
task action << {
    println "Producing message:"
    rootProject.producerMessage = 'Watch the order of execution.'
}
```

consumer/build.gradle

```
task action << {
    println("Consuming message: ${rootProject.producerMessage}")
}
```

gradle -q action の出力

```
> gradle -q action
Producing message:
Consuming message: Watch the order of execution.
```

ただ、このハックは簡単に破綻してしまいます。consumerディレクトリに移動して、ビルドしてみまし

例57.16 依存関係とビルド実行順序

gradle -q action の出力

```
> gradle -q action
Consuming message: null
```

問題はこの二つの“`action`”タスクが無関係であることです。 確かに、`messages`
プロジェクトをビルドすればGradleはこの二つのタスクを両方とも実行します。
しかし、それは二つのタスクが同じ名前のタスクであり、両方とも実行ディレクトリ以下にあるからです
最後の例では一方のタスクが実行ディレクトリから外れたので、片方の“`action`
”タスクしか実行されませんでした。 このハックよりも良い方法が必要です。

57.6.1.2. 依存関係を宣言する

例57.17 依存関係を宣言する

Build layout

```
messages/
  settings.gradle
  consumer/
    build.gradle
  producer/
    build.gradle
```

ノート：
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/userguide/multiproject/dependencies/messagesWithDependencies/messag

settings.gradle

```
include 'consumer', 'producer'
```

consumer/build.gradle

```
task action(dependsOn: ":producer:action") << {
    println("Consuming message: ${rootProject.producerMessage}")
}
```

producer/build.gradle

```
task action << {
    println "Producing message:"
    rootProject.producerMessage = 'Watch the order of execution.'
}
```

gradle -q action の出力

```
> gradle -q action
Producing message:
Consuming message: Watch the order of execution.
```

consumerディレクトリでビルドすると、以下のようになります。

例57.18 依存関係を宣言する

gradle -q action の出力

```
> gradle -q action
Producing message:
Consuming message: Watch the order of execution.
```

consumerプロジェクトの“action”タスクは producerプロジェクトの“action”タスクに実行時依存関係を持つことを宣言したことで、事態は改善しました。

57.6.1.3. プロジェクトをまたぐタスク間依存関係の性質

もちろん、同じ名前のタスク間でしかプロジェクトをまたぐ依存関係は結べない、ということはありませ
タスクの名前を変えて実行してみたのが、以下の例です。

例57.19 プロジェクトにまたがるタスク間の依存関係

consumer/build.gradle

```
task consume(dependsOn: ':producer:produce') << {
    println("Consuming message: ${rootProject.producerMessage}")
}
```

producer/build.gradle

```
task produce << {
    println "Producing message:"
    rootProject.producerMessage = 'Watch the order of execution.'
}
```

gradle -q consume の出力

```
> gradle -q consume
Producing message:
Consuming message: Watch the order of execution.
```

57.6.2. 評価順序の依存関係

そ ろ そ ろ 実 際 の Java
での例をみていきたいと思いますが、その前に、producer-consumerプロジェクトについてもう一つ例

例57.20 評価順序の依存関係

consumer/build.gradle

```
def message = rootProject.producerMessage

task consume << {
    println("Consuming message: " + message)
}
```

producer/build.gradle

```
rootProject.producerMessage = 'Watch the order of evaluation.'
```

gradle -q consume の出力

```
> gradle -q consume
Consuming message: null
```

デフォルトでは、プロジェクトはプロジェクト名のアルファベット順で評価されます。したがって、con
プロジェクトはproducerプロジェクトの前に評価され、producerプロジェクトのプロパティはconsu
プロジェクトが読みにいった後にセットされてしまうのです。Gradleでは、このような問題に対する解

例57.21 評価順序の依存関係 - evaluationDependsOn

consumer/build.gradle

```
evaluationDependsOn(':producer')

def message = rootProject.producerMessage

task consume << {
    println("Consuming message: " + message)
}
```

gradle -q consume の出力

```
> gradle -q consume
Consuming message: Watch the order of evaluation.
```

このevaluationDependsOnコマンドにより、producerプロジェクトがconsumerの前に評価されるようになります。
なお、この例はメカニズムを説明するためのやや不自然なものです。
この場合は実行時にタスク内でプロパティを読みにいった方が早いでしょう。

例57.22 評価順序の依存関係

consumer/build.gradle

```
task consume << {
    println("Consuming message: ${rootProject.producerMessage}")
}
```

gradle -q consume の出力

```
> gradle -q consume
Consuming message: Watch the order of evaluation.
```

評価順序の依存関係と実行時の依存関係では、大きく異なる点があります。
まず、実行の依存関係は最終的には常にタスク間の依存関係になりますが、評価順序の依存関係はプロシ
さらに、サブプロジェクトを始点にビルドを開始した場合でも、そのマルチプロジェクトに含まれる全て
デフォルトでは、ルートプロジェクトからトップダウンでサブプロジェクトに向かって評価されていきま

このデフォルトを変更して"ボトムアップ"で評価されるようにするには、かわりに"evaluationDepend
"メソッドを使ってください。

同じ階層にあるプロジェクトは、プロジェクト名のアルファベット順で評価されます。よく使われるのに

57.6.3. 実生活での例

Gradleのマルチプロジェクト機能は、開発現場でのユースケースに基づいて作成されています。ここでに
[33] この例では、クロスプロジェクト設定を使って、一つのビルドスクリプトですべてのプロジェクトの設定を行っています。

例57.23 依存関係 - 実生活の例 - クロスプロジェクト設定

Build layout

```
webDist/
  settings.gradle
  build.gradle
  date/
    src/main/java/
      org/gradle/sample/
        DateServlet.java
  hello/
    src/main/java/
      org/gradle/sample/
        HelloServlet.java
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/userguide/multiproject/dependencies/webDist

settings.gradle

```
include 'date', 'hello'
```

build.gradle

```
allprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
}

subprojects {
    apply plugin: 'war'
    repositories {
        mavenCentral()
    }
    dependencies {
        compile "javax.servlet:servlet-api:2.5"
    }
}

task explodedDist(dependsOn: assemble) << {
    File explodedDist = mkdir("$buildDir/explodedDist")
    subprojects.each {project ->
        project.tasks.withType(Jar).each {archiveTask ->
            copy {
                from archiveTask.archivePath
                into explodedDist
            }
        }
    }
}
```

この例は、面白い依存関係を持っています。dateプロジェクトとhello
プロジェクトで定義されているタスクの評価は、明らかにwebDistの評価
に依存しています。Webアプリのビルドロジックは、すべてwebDistによって注入されているからです。

ｗｅｂＤｉｓｔはｈｅｌｌｏとｄａｔｅ
の成果物を使ってビルドされるからです。さらに、もう一つの依存関係があります。webDistプロジェク

この手の依存関係は、マルチプロジェクトのビルドではよく発生します。ビルドシステムがこのような依

57.7. プロジェクト依存関係

あるプロジェクトが、別のプロジェクトの生成するjarをコンパイルするときのクラスパスに必要としてい
これは、Javaのマルチプロジェクトではいかにもよくありそうなパターンです。
「プロジェクト依存関係」
ですでに言及していますが、プロジェクト依存関係はこのようなケースに対応するものです。

例57.24 プロジェクト依存関係

Build layout

```
java/
  settings.gradle
  build.gradle
  api/
    src/main/java/
      org/gradle/sample/
        api/
          Person.java
        apiImpl/
          PersonImpl.java
  services/personService/
    src/
      main/java/
        org/gradle/sample/services/
          PersonService.java
      test/java/
        org/gradle/sample/services/
          PersonServiceTest.java
  shared/
    src/main/java/
      org/gradle/sample/shared/
        Helper.java
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/userguide/multiproject/dependencies/java

shared、api、personServiceの三つのプロジェクトがあります。personService
プロジェクトは他の二つのプロジェクトに依存しています。さらに、apiプロジェクトはshared
プロジェクトに依存しています。[35]

例57.25 プロジェクト依存関係

settings.gradle

```
include 'api', 'shared', 'services:personService'
```

build.gradle

```
subprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
    repositories {
        mavenCentral()
    }
    dependencies {
        testCompile "junit:junit:4.11"
    }
}

project(':api') {
    dependencies {
        compile project(':shared')
    }
}

project(':services:personService') {
    dependencies {
        compile project(':shared'), project(':api')
    }
}
```

ビルドロジックはすべてルートプロジェクトのbuild.gradleに記述されています。 [37]
プロジェクト依存関係は、実行依存の別形態といえます。まずあるプロジェクトのビルドを実行し、生成
gradle compileと入力してみましょう。まずsharedがビルドされ、次にapi
がビルドされるでしょう。プロジェクト依存はマルチプロジェクトの部分ビルドを可能にします。

Mavenを使用されてきた人なら、このことですごく満足できるでしょう。Ivyの経験をお持ちなら、もっ

例57.26 細かい依存関係の制御

build.gradle

```
subprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
}

project(':api') {
    configurations {
        spi
    }
    dependencies {
        compile project(':shared')
    }
    task spiJar(type: Jar) {
        baseName = 'api-spi'
        dependsOn classes
        from sourceSets.main.output
        include('org/gradle/sample/api/**')
    }
    artifacts {
        spi spiJar
    }
}

project(':services:personService') {
    dependencies {
        compile project(':shared')
        compile project(path: ':api', configuration: 'spi')
        testCompile "junit:junit:4.11", project(':api')
    }
}
```

Javaプラグインは、デフォルトではすべてのクラスが含まれたjarをプロジェクトのライブラリとして追カ
そ　れ　に　加　え　て
、apiプロジェクトのインターフェースのみが含まれたライブラリを作成しています。そしてこのライブ
serviceをコンパイルするときはインターフェースしか要りませんが、テストをするにはapiからすべての

57.7.1. 依存プロジェクトをビルドしないようにする

部分ビルドを実行する際、依存しているプロジェクトはビルドしたくない、という場合もあるかもしれま
-aオプションをつけてプロジェクトをビルドしてください。

57.8. Parallel project execution

With more and more CPU cores available on developer desktops and CI servers, it is important that Gradle is able to fully utilise these processing resources. More specifically, the parallel execution attempts to:

- Reduce total build time for a multi-project build where execution is IO bound or otherwise does not consume all available CPU resources.

- Provide faster feedback for execution of small projects without awaiting completion of other projects.

Although Gradle already offers parallel test execution via `Test.setMaxParallelForks()` the feature described in this section is parallel execution at a project level. Parallel execution is an incubating feature. Please use it and let us know how it works for you.

Parallel project execution allows the separate projects in a decoupled multi-project build to be executed in parallel (see also: 「分離されたプロジェクト 」). While parallel execution does not strictly require decoupling at configuration time, the long-term goal is to provide a powerful set of features that will be available for fully decoupled projects. Such features include:

- 「Configuration on demand」.
- Configuration of projects in parallel.
- Re-use of configuration for unchanged projects.
- Project-level up-to-date checks.
- Using pre-built artifacts in the place of building dependent projects.

How does parallel execution work? First, you need to tell Gradle to use the parallel mode. You can use the command line argument (付録D Gradle コマンドライン) or configure your build environment (「gradle.propertiesを使用したビルド環境の構築」). Unless you provide a specific number of parallel threads Gradle attempts to choose the right number based on available CPU cores. Every parallel worker exclusively owns a given project while executing a task. This means that 2 tasks from the same project are never executed in parallel. Therefore only multi-project builds can take advantage of parallel execution. Task dependencies are fully supported and parallel workers will start executing upstream tasks first. Bear in mind that the alphabetical scheduling of decoupled tasks, known from the sequential execution, does not really work in parallel mode. You need to make sure the task dependencies are declared correctly to avoid ordering issues.

57.9. 分離されたプロジェクト

Gradleでは、評価フェーズでも実行フェーズでも、あらゆるプロジェクト間でお互いにアクセスすること
これはビルドの作成者にとっては非常に強力で柔軟性のある仕様ですが、ビルド時にはGradleの方の柔軟
例を挙げると、強固に結合されたプロジェクトは、Gradleに平行ビルドさせることができませんし、依存

お互いのプロジェクト・モデルにアクセスしない二つのプロジェクトは、分離された
プ ロ ジ ェ ク ト と 呼 ば れ ま す 。
分離されたプロジェクトでは、お互いのやりとりに依存関係宣言のみを使用します。つまり、プロジェク
「プロジェクト依存関係 」)やタスク依存(「タスクの依存関係」)です。
その他のあらゆるアクセス、例えば他プロジェクトのオブジェクトを変更したり、設定値を読み込んだり
結合させます。

プロジェクトが結合されるもっとも一般的な状況は、設定のインジェクション(
「クロスプロジェクト設定」)です。 これは直ぐには分かりにくいかもしれませんが、`allprojects`
や `subprojects`
キーワードのような、Gradleの特定の機能を使うと、プロジェクトは自動的に結合されます。
これらのキーワードは、`build.gradle`

ファイルで別のプロジェクトを定義するのに使われるからです。
共通設定の定義しかしていないルートプロジェクトもよくありますが、Gradleがルートプロジェクトを、allprojectsを使用している限り、プロジェクトは他の全てのプロジェクトと結合されます。

つまり、どんな形であれ、ビルドスクリプトのロジック共有、および設定のインジェクション(allproj
やsubprojectsなど)は、プロジェクトを結合させるということです。
私たちは、この分離されたプロジェクトという概念を拡張し、有効活用するような機能を追加する予定て

In order to make good use of cross project configuration without running into issues for parallel and 'configuration on demand' options, follow these recommendations:

- Avoid a subproject's `build.gradle` referencing other subprojects; prefering cross configuration from the root project.
- Avoid changing the configuration of other projects at execution time.

57.10. マルチプロジェクトのビルドとテスティング

Javaプラグインのbuild
タスクは、シングルプロジェクトをコンパイル、テスト、さらに(CodeQualityプラグインを使っていれに
buildNeededとbuildDependentsはこの用途を助けてくれるものです。

例57.25「プロジェクト依存関係」
にあるプロジェクトの構造を使いましょう。この例では:service:personserviceが:apiと:sharedの双方に

シングルプロジェクトの:apiプロジェクトで作業しているとします。:apiに変更を加えましたが、クリー
buildタスクは、まさにこの処理を行います。

例57.27 シングルプロジェクトのビルドとテスト

gradle :api:build の出力

```
> gradle :api:build
:shared:compileJava
:shared:processResources
:shared:classes
:shared:jar
:api:compileJava
:api:processResources
:api:classes
:api:jar
:api:assemble
:api:compileTestJava
:api:processTestResources
:api:testClasses
:api:test
:api:check
:api:build

BUILD SUCCESSFUL

Total time: 1 secs
```

典型的な開発サイクルでは、:apiプロジェクトに何か変更を加えるたびビルドとテストを繰り返すことに
- a
オプションをつけてビルドすると、依存プロジェクトのライブラリを解決する際、キャッシュされたjarた

例57.28 シングルプロジェクトの部分ビルドとテスト

gradle -a :api:build の出力

```
> gradle -a :api:build
:api:compileJava
:api:processResources
:api:classes
:api:jar
:api:assemble
:api:compileTestJava
:api:processTestResources
:api:testClasses
:api:test
:api:check
:api:build

BUILD SUCCESSFUL

Total time: 1 secs
```

今、ちょうど最新のソースをバージョン管理システムから持ってきたとしましょう。:apiが依存している
b u i l d N e e d e d
タスクは、指定したプロジェクトだけでなく、そのプロジェクトのtestRuntime依存関係に設定されてい

例57.29 依存プロジェクトのビルドとテスト

`gradle :api:buildNeeded` の出力

```
> gradle :api:buildNeeded
:shared:compileJava
:shared:processResources
:shared:classes
:shared:jar
:api:compileJava
:api:processResources
:api:classes
:api:jar
:api:assemble
:api:compileTestJava
:api:processTestResources
:api:testClasses
:api:test
:api:check
:api:build
:shared:assemble
:shared:compileTestJava
:shared:processTestResources
:shared:testClasses
:shared:test
:shared:check
:shared:build
:shared:buildNeeded
:api:buildNeeded

BUILD SUCCESSFUL

Total time: 1 secs
```

さらに、:apiプロジェクトをリファクタリングする場合のことを考えてみます。:apiプロジェクトは他の
b u i l d D e p e n d e n t s
タスクは、指定したプロジェクトだけでなく、そのプロジェクトにtestRuntimeのプロジェクト依存関係

例57.30 依存されているプロジェクトのビルドとテスト

gradle :api:buildDependents の出力

```
> gradle :api:buildDependents
:shared:compileJava
:shared:processResources
:shared:classes
:shared:jar
:api:compileJava
:api:processResources
:api:classes
:api:jar
:api:assemble
:api:compileTestJava
:api:processTestResources
:api:testClasses
:api:test
:api:check
:api:build
:services:personService:compileJava
:services:personService:processResources
:services:personService:classes
:services:personService:jar
:services:personService:assemble
:services:personService:compileTestJava
:services:personService:processTestResources
:services:personService:testClasses
:services:personService:test
:services:personService:check
:services:personService:build
:services:personService:buildDependents
:api:buildDependents

BUILD SUCCESSFUL

Total time: 1 secs
```

最後に、すべてのプロジェクトをビルド、テストしたいと思うことがあるかもしれません。ルートプロシ
gradle buildと実行すれば大丈夫です。すべてのプロジェクトがビルドされテストされます。

57.11. Multi Project and buildSrc

「buildSrcプロジェクトのソースをビルドする」 tells us that we can place build logic to be compiled and tested in the special buildSrc directory. In a multi project build, there can only be one buildSrc directory which must be located in the root directory.

57.12. プロパティとメソッドの継承

プロジェクトで宣言されたプロパティとメソッドは、すべての子プロジェクトに継承されます。これは設

なお、Gradleの設定のインジェクション
ではまだメソッドを注入することができません(将来のリリースで対応する予定です)。なので、メソッド

どうみても継承機能を敬遠しているのに、なんでそんなのを実装したのか不思議に思われるかもしれませ

57.13. まとめ

この章を書くのは非常に疲れました。読むのも疲れるでしょう。最後にひとつだけ、GradleでマルチプD
allproject、subprojects、evaluationDependsOn、evaluationDependsOnChildren
、そしてプロジェクト依存関係です。 [39]
この5つ、そしてGradleが設定フェーズと実行フェーズを別個に取り扱うのだということを覚えておけば

---

[33] 実際のケースではhttp://lucene.apache.org/solr
を使っており、warファイルをアクセスするインデックスごとに分割する必要がありました。分散型のウ

[35] services
もまたプロジェクトですが、これはただの入れ物です。このプロジェクトにはビルドスクリプトも別のフ

[37] この例では、ビルドスクリプトの配置を少し簡易化しています。普通は、プロジェクトが指定するものは

[39] マジカルナンバー7±2に収まるナイスな量です

58 カスタムタスクの作成

Gradleは２種類のタスクをサポートしています。
そのひとつは単純タスクで、アクションクロージャでタスクを定義します。　　　　これらは6章 ビルドスクリプトの基本で説明してきたものです。
このタイプのタスクに対しては、タスクのふるまいはアクションクロージャによって決まります。
このタイプのタスクはビルドスクリプトで1回限りのタスクを実装するのに適しています。

もうひとつは拡張タスクで、ふるまいがタスクの中に組み込まれており、ふるまいを制御することのでき
これらは15章タスク詳解で説明してきたものです。
ほとんどのGradleプラグインは拡張タスクを利用します。
拡張タスクでは、単純タスクのようにふるまいを実装する必要はありません。
単にタスクを宣言し、プロパティを利用してタスクを制御するだけです。
この方法により、拡張タスクではふるまいの断片を多くの場所で再利用したり、場合によっては異なるヒ

拡張タスクのふるまいやプロパティはタスクのクラスによって定義されます。
拡張タスクを宣言するときは、タスクのタイプやクラスを指定します。

Gradleで独自のカスタムタスククラスを実装するのは簡単です。
カスタムタスククラスは、最終的にバイトコードにコンパイルされるなら、どんな言語でも好きなものて
このサンプルでは実装言語としてGroovyを使いますが、例えばJavaやScalaを使うこともできます。
Gradle APIはGroovyとの親和性が高いように設計されているので、一般論としてGroovyが一番簡単な選択肢では

58.1. タスククラスのパッケージング

タスククラスのソースを配置できる箇所はいくつかあります。

ビルドスクリプト
: タスククラスをビルドスクリプトに直接含めることができます。
この方法は、特に何もしなくてもタスククラスが自動的にコンパイルされ、ビルドスクリプトのクラ
しかし、タスククラスがビルドスクリプトの外部からは参照できないため、タスククラスを定義して

buildSrcプロジェクト
: タスククラスのソースを　*rootProjectDir*/buildSrc/src/main/groovy ディレクトリに配置できます。
Gradleがタスククラスのコンパイルとテストを実行し、ビルドスクリプトのクラスパス上で有効にな
タスククラスはビルドで利用されるすべてのビルドスクリプトから参照可能です。
しかし、ビルドの外部からは参照できないので、タスクが定義されているビルドの外部でタスククラ

b u i l d S r c
プロジェクトを使うアプローチでは、タスクの宣言「タスクが何を実行すべきか」と、タスクの実装

buildSrcプロジェクトの詳細については、60章ビルドロジックの体系化を参照してください。

スタンドアロンプロジェクト
タスククラスのために独立したプロジェクトを作ることもできます。
このプロジェクトはJARを生成して発行するので、複数のビルドで利用したり、他のユーザーと共有す
一般に、このJARはカスタムプラグインを含むか、関連するいくつかのタスククラスを単一のライブラ
あるいは、その両方の組み合わせです。

サンプルでは、簡単のためにビルドスクリプト内のタスククラスからはじめます。
その後、スタンドアロンプロジェクトについて説明します。

58.2. 単純タスクの作成

カスタムタスククラスを実装するには、DefaultTaskを拡張します。

例58.1 カスタムタスクの定義

build.gradle

```
class GreetingTask extends DefaultTask {
}
```

このタスクは何の役にも立たないので、ふるまいを追加してみましょう。
そのためには、タスクにメソッドを追加し、TaskActionアノテーションでマークします。
タスクが実行されたときに、Gradleがこのメソッドを呼びます。
タスクのふるまいを定義するには、必ずしもメソッドを使う必要はありません。
例えば、タスクのコンストラクタでクロージャを引数としてdoFirst()やdoLast()
を呼ぶことでふるまいを追加することもできます。

例58.2 hello worldタスク

build.gradle

```
task hello(type: GreetingTask)

class GreetingTask extends DefaultTask {
    @TaskAction
    def greet() {
        println 'hello from GreetingTask'
    }
}
```

gradle -q hello の出力

```
> gradle -q hello
hello from GreetingTask
```

タスクにプロパティを追加して、カスタマイズできるようにしてみましょう。

タスクは単純なPOGOsで、タスクを宣言するときにプロパティをセットしたり、タスクオブジェクトの.
ここでは、greetingプロパティを追加して、greeting
タスクを宣言するときにその値をセットしています。

例58.3 カスタマイズ可能なhello worldタスク

build.gradle

```
// Use the default greeting
task hello(type: GreetingTask)

// Customize the greeting
task greeting(type: GreetingTask) {
    greeting = 'greetings from GreetingTask'
}

class GreetingTask extends DefaultTask {
    String greeting = 'hello from GreetingTask'

    @TaskAction
    def greet() {
        println greeting
    }
}
```

gradle -q hello greeting の出力

```
> gradle -q hello greeting
hello from GreetingTask
greetings from GreetingTask
```

58.3. スタンドアロンプロジェクト

それでは、タスクをスタンドアロンプロジェクトに移動して、発行したり他ユーザーと共有したりできる
このプロジェクトは単純なGroovyプロジェクトで、タスククラスを含むJARを生成します。
プロジェクトに対する簡単なビルドスクリプトがこちらです。

例58.4 カスタムタスクのビルド

build.gradle

```
apply plugin: 'groovy'

dependencies {
    compile gradleApi()
    compile localGroovy()
}
```

> ノート:
> 本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
> **samples/customPlugin/plugin**

タスククラスのソースを配置する場所は規約に従いました。

例58.5 カスタムタスク

src/main/groovy/org/gradle/GreetingTask.groovy

```
package org.gradle

import org.gradle.api.DefaultTask
import org.gradle.api.tasks.TaskAction

class GreetingTask extends DefaultTask {
    String greeting = 'hello from GreetingTask'

    @TaskAction
    def greet() {
        println greeting
    }
}
```

58.3.1. タスククラスを別のプロジェクトで使う

ビルドスクリプトでタスククラスを使うためには、ビルドスクリプトのクラスパスにタスククラスを追加
これを実行するには、「ビルドスクリプトで外部ライブラリを使うときの依存関係設定」
で説明されているとおり、buildscript { }ブロックを使います。
次のサンプルでは、タスククラスを含むJARをローカルリポジトリに発行した場合にどのようにすればよ

例58.6 カスタムタスクを別のプロジェクトで使う

build.gradle

```
buildscript {
    repositories {
        maven {
            url uri('../repo')
        }
    }
    dependencies {
        classpath group: 'org.gradle', name: 'customPlugin',
                  version: '1.0-SNAPSHOT'
    }
}

task greeting(type: org.gradle.GreetingTask) {
    greeting = 'howdy!'
}
```

58.3.2. タスククラスのテストの作成

タスククラスのテストのために、ProjectBuilderクラスを利用して Project
のインスタンスを生成することができます。

例58.7 カスタムタスクのテスト

src/test/groovy/org/gradle/GreetingTaskTest.groovy

```
class GreetingTaskTest {
    @Test
    public void canAddTaskToProject() {
        Project project = ProjectBuilder.builder().build()
        def task = project.task('greeting', type: GreetingTask)
        assertTrue(task instanceof GreetingTask)
    }
}
```

58.4. Incremental tasks

Incremental tasks are an incubating feature.

Since the introduction of the implementation described above (early in the Gradle 1.6 release cycle), discussions within the Gradle community have produced superior ideas for exposing the information about changes to task implementors to what is described below. As such, the API for this feature will almost certainly change in upcoming releases. However, please do experiment with the current implementation and share your experiences with the Gradle community.

The feature incubation process, which is part of the Gradle feature lifecycle (see 付録C 機能のライフサイクル), exists for this purpose of ensuring high quality final implementations through incorporation of early user feedback.

With Gradle, it's very simple to implement a task that gets skipped when all of it's inputs and outputs are up to date (see 「更新されていないタスクをスキップする」). However, there are times when only a few input files have changed since the last execution, and you'd like to avoid reprocessing all of the unchanged inputs. This can be particularly useful for a transformer task, that converts input files to output files on a 1:1 basis.

If you'd like to optimise your build so that only out-of-date inputs are processed, you can do so with an incremental task.

58.4.1. Implementing an incremental task

For a task to process inputs incrementally, that task must contain an incremental task action. This is a task action method that contains a single `IncrementalTaskInputs` parameter, which indicates to Gradle that the action will process the changed inputs only.

The incremental task action may supply an `IncrementalTaskInputs.outOfDate()` action for processing any input file that is out-of-date, and a `IncrementalTaskInputs.removed()` action that executes for any input file that has been removed since the previous execution.

例58.8 Defining an incremental task action

build.gradle

```
class IncrementalReverseTask extends DefaultTask {
    @InputDirectory
    def File inputDir

    @OutputDirectory
    def File outputDir

    @Input
    def inputProperty

    @TaskAction
    void execute(IncrementalTaskInputs inputs) {
        println inputs.incremental ? "CHANGED inputs considered out of date"
                                   : "ALL inputs considered out of date"
        inputs.outOfDate { change ->
            println "out of date: ${change.file.name}"
            def targetFile = new File(outputDir, change.file.name)
            targetFile.text = change.file.text.reverse()
        }

        inputs.removed { change ->
            println "removed: ${change.file.name}"
            def targetFile = new File(outputDir, change.file.name)
            targetFile.delete()
        }
    }
}
```

ノート:
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/userguide/tasks/incrementalTask

For a simple transformer task like this, the task action simply needs to generate output files for any out-of-date inputs, and delete output files for any removed inputs.

A task may only contain a single incremental task action.

58.4.2. Which inputs are considered out of date?

When Gradle has history of a previous task execution, and the only changes to the task execution context since that execution are to input files, then Gradle is able to determine which input files need to be reprocessed by the task. In this case, the `IncrementalTaskInputs.outOfDate()` action will be executed for any input file that was added or modified, and the `IncrementalTaskInputs.removed()` action will be executed for any removed input file.

However, there are many cases where Gradle is unable to determine which input files need to be reprocessed. Examples include:

- There is no history available from a previous execution.

- You are building with a different version of Gradle. Currently, Gradle does not use task history from a different version.
- An `upToDateWhen` criteria added to the task returns `false`.
- An input property has changed since the previous execution.
- One or more output files have changed since the previous execution.

In any of these cases, Gradle will consider all of the input files to be `outOfDate`. The `IncrementalTaskInputs.outOfDate()` action will be executed for every input file, and the `IncrementalTaskInputs.removed()` action will not be executed at all.

You can check if Gradle was able to determine the incremental changes to input files with `IncrementalTaskInputs.isIncremental()`.

58.4.3. An incremental task in action

Given the incremental task implementation above, we can explore the various change scenarios by example. Note that the various mutation tasks ('updateInputs', 'removeInput', etc) are only present for demonstration purposes: these would not normally be part of your build script.

First, consider the `IncrementalReverseTask` executed against a set of inputs for the first time. In this case, all inputs will be considered “out of date”:

例58.9 Running the incremental task for the first time

build.gradle

```
task incrementalReverse(type: IncrementalReverseTask) {
    inputDir = file('inputs')
    outputDir = file("$buildDir/outputs")
    inputProperty = project.properties['taskInputProperty'] ?: "original"
}
```

Build layout

```
incrementalTask/
  build.gradle
  inputs/
    1.txt
    2.txt
    3.txt
```

gradle -q incrementalReverse の出力

```
> gradle -q incrementalReverse
ALL inputs considered out of date
out of date: 1.txt
out of date: 2.txt
out of date: 3.txt
```

Naturally when the task is executed again with no changes, then the entire task is up to date and no files are reported to the task action:

例58.10 Running the incremental task with unchanged inputs

gradle -q incrementalReverse の出力

```
> gradle -q incrementalReverse
```

When an input file is modified in some way or a new input file is added, then re-executing the task results in those files being reported to `IncrementalTaskInputs.outOfDate()`:

例58.11 Running the incremental task with updated input files

build.gradle

```
task updateInputs() << {
    file('inputs/1.txt').text = "Changed content for existing file 1."
    file('inputs/4.txt').text = "Content for new file 4."
}
```

gradle -q updateInputs incrementalReverse の出力

```
> gradle -q updateInputs incrementalReverse
CHANGED inputs considered out of date
out of date: 1.txt
out of date: 4.txt
```

When an existing input file is removed, then re-executing the task results in that file being reported to `IncrementalTaskInputs.removed()`:

例58.12 Running the incremental task with an input file removed

build.gradle

```
task removeInput() << {
    file('inputs/3.txt').delete()
}
```

gradle -q removeInput incrementalReverse の出力

```
> gradle -q removeInput incrementalReverse
CHANGED inputs considered out of date
removed: 3.txt
```

When an output file is deleted (or modified), then Gradle is unable to determine which input files are out of date. In this case, all input files are reported to the `IncrementalTaskInputs.outOfDate()` action, and no input files are reported to the `IncrementalTaskInputs.removed()` action:

例58.13 Running the incremental task with an output file removed

build.gradle

```
task removeOutput() << {
    file("$buildDir/outputs/1.txt").delete()
}
```

gradle -q removeOutput incrementalReverse の出力

```
> gradle -q removeOutput incrementalReverse
ALL inputs considered out of date
out of date: 1.txt
out of date: 2.txt
out of date: 3.txt
```

When a task input property is modified, Gradle is unable to determine how this property impacted the task outputs, so all input files are assumed to be out of date. So similar to the changed output file example, all input files are reported to the `IncrementalTaskInputs.outOfDate()` action, and no input files are reported to the `IncrementalTaskInputs.removed()` action:

例58.14 Running the incremental task with an input property changed

gradle -q -PtaskInputProperty=changed incrementalReverse の出力

```
> gradle -q -PtaskInputProperty=changed incrementalReverse
ALL inputs considered out of date
out of date: 1.txt
out of date: 2.txt
out of date: 3.txt
```

59
カスタムプラグインの作成

Gradleプラグインは再利用可能なビルドロジックの断片をパッケージとしてまとめ、異なるプロジェクト
Gradleは独自のカスタムプラグインの実装手段を提供していますので、独自のビルドロジックを再利用し

カスタムプラグインは、最終的にバイトコードへコンパイルできるなら、どんな言語でもお好みの言語て
このサンプルでは実装言語としてGroovyを使います。
かわりにJavaやScalaを使うこともできますので、お好きにどうぞ。

59.1. プラグインのパッケージング

プラグインのソースを配置できる場所はいくつかあります。

ビルドスクリプト
: ビルドスクリプトの中にプラグインのソースを直接含めることができます。
この方法は、特になにもしなくてもプラグインが自動的にコンパイルされ、ビルドスクリプトのクラ
しかし、プラグインはビルドスクリプトの外部から参照できないため、プラグインが定義されている

`buildSrc`プロジェクト
: プラグインのソースを*`rootProjectDir`*`/buildSrc/src/main/groovy`ディレクトリに配置できます。
Gradleがプラグインのコンパイルとテストを行い、ビルドスクリプトのクラスパスで有効になるよう
プラグインは、ビルドで利用されるすべてのビルドスクリプトから参照可能です。
しかし、ビルドの外部からは参照できないので、プラグインが定義されているビルドの外部から再利

`buildSrc`プロジェクトの詳細については、60章ビルドロジックの体系化を参照してください。

スタンドアロンプロジェクト
: プラグイン用に独立したプロジェクトを作ることができます。
このプロジェクトはJARを生成して発行するので、複数のビルドで利用したり、他のユーザーと共有す
一般的に、このJARはカスタムプラグイン、ないしは関連するいくつかのタスククラスのバンドルする
あるいは、その両方の組み合わせです。

このサンプルでは、簡単のためにビルドスクリプト内のプラグインからはじめます。
その後、スタンドアロンプロジェクトの作り方をみてみましょう。

59.2. シンプルなプラグインの作成

カスタムプラグインを作るには、`Plugin`の実装を作成する必要があります。
Gradleはプラグインのインスタンスを生成し、プロジェクトでプラグインが利用されるときにプラグイン
`Plugin.apply()`メソッドを呼びます。
プロジェクトオブジェクトがパラメータとして渡され、必要に応じてプラグインがプロジェクトを構成す
次のサンプルは、プロジェクトに`hello`タスクを追加するgreetingプラグインを含んでいます。

例59.1 カスタムプラグイン

build.gradle

```
apply plugin: GreetingPlugin

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.task('hello') << {
            println "Hello from the GreetingPlugin"
        }
    }
}
```

gradle -q hello の出力

```
> gradle -q hello
Hello from the GreetingPlugin
```

注意すべきは、指定されたプラグインが適用されるプロジェクト毎に、新しいプラグインのインスタンス
note that the `Plugin` class is a generic type. This example has it receiving the `Plugin` type as a type parameter. It's possible to write unusual custom plugins that take different type parameters, but this will be unlikely (until someone figures out more creative things to do here).

59.3. ビルドから入力を得る

プロジェクトにシンプルなextensionオブジェクトを追加してみましょう
ここでは、プロジェクトにextensionオブジェクト`greeting`
を追加し、greetingタスクを構成可能にしています。

例59.2 カスタムプラグインのextension

build.gradle

```
apply plugin: GreetingPlugin

greeting.message = 'Hi from Gradle'

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Add the 'greeting' extension object
        project.extensions.create("greeting", GreetingPluginExtension)
        // Add a task that uses the configuration
        project.task('hello') << {
            println project.greeting.message
        }
    }
}

class GreetingPluginExtension {
    def String message = 'Hello from GreetingPlugin'
}
```

gradle -q hello の出力

```
> gradle -q hello
Hi from Gradle
```

この例では、`GreetingPluginExtension`は`message`フィールドを持つPOGO(plain old Groovy object)です。 extensionオブジェクトは`greeting`という名前でプラグインリストに追加されます。このオブジェクトは、extensionオブジェクトと同じ名前のプロジェクトプロパティとして有効になりま

しばしば、ひとつのプラグインに対していくつかの関連したプロパティを指定する必要がある場合があり Gradleはそれぞれのextensionオブジェクトに対してコンフィグレーションクロージャブロックを追加す グループの設定をまとめて行うことができます。
次のサンプルは、これがどのように働くのかを示しています。

例59.3 コンフィグレーションクロージャ付きのカスタムプラグイン

build.gradle

```
apply plugin: GreetingPlugin

greeting {
    message = 'Hi'
    greeter = 'Gradle'
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.extensions.create("greeting", GreetingPluginExtension)
        project.task('hello') << {
            println "${project.greeting.message} from ${project.greeting.greeter}"
        }
    }
}

class GreetingPluginExtension {
    String message
    String greeter
}
```

gradle -q hello の出力

```
> gradle -q hello
Hi from Gradle
```

この例では、greeting
クロージャ内にいくつかの設定をグループ化してまとめることができています。
ビルドスクリプトにおけるクロージャブロックの名前(greeting
)は、extensionオブジェクトの名前と一致している必要があります。
そして、クロージャが実行されると、Groovyの標準のクロージャ委譲機能に基づいて、extensionオブジ

59.4. カスタムタスクやプラグインでファイルを扱う

カスタムタスクやプラグインを開発するとき、ファイルロケーションの入力コンフィグレーションを柔軟
これを実現するにあたり、ファイルに対する値の解決をなるべく遅らせるためにProject.file()
メソッドを活用できます

例59.4 ファイルプロパティの遅延評価

build.gradle

```
class GreetingToFileTask extends DefaultTask {

    def destination

    File getDestination() {
        project.file(destination)
    }

    @TaskAction
    def greet() {
        def file = getDestination()
        file.parentFile.mkdirs()
        file.write "Hello!"
    }
}

task greet(type: GreetingToFileTask) {
    destination = { project.greetingFile }
}

task sayGreeting(dependsOn: greet) << {
    println file(greetingFile).text
}

ext.greetingFile = "$buildDir/hello.txt"
```

gradle -q sayGreeting の出力

```
> gradle -q sayGreeting
Hello!
```

この例では、greetタスクのdestinationプロパティをクロージャで定義したので、
クロージャの戻り値をfileオブジェクトに変換することが必要になる直前にProject.file()
メソッドによって評価されます。
上記のサンプルでは、タスクで値を利用するコンフィグレーションを行った後でgreetingFile
プロパティの値を指定していることに気づくでしょう。
この種の遅延評価は、ファイルプロパティを設定する際に任意の値を受け入れ、そしてプロパティの読み

59.5. スタンドアロンプロジェクト

それでは、プラグインをスタンドアロンプロジェクトに移動して、発行して他のユーザーと共有できるよ
このプロジェクトは、プラグインクラスを含むJARを発行する単なるGroovyプロジェクトです。
プロジェクトに対するシンプルなビルドスクリプトはこのようになります。
Groovyプラグインを適用して、コンパイル時の依存関係としてGradle APIを追加しています。

例59.5 カスタムプラグインに対するビルド

build.gradle

```
apply plugin: 'groovy'

dependencies {
    compile gradleApi()
    compile localGroovy()
}
```

ノ ー ト :
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/customPlugin/plugin

Gradleはどのようにして`Plugin`の実装を探すのでしょうか？ その答えは、JARの`META-INF/gradle-p`
デ ィ レ ク ト リ に 、
プラグインのIDに対応するプロパティファイルを提供する必要があるということです。

例59.6 カスタムプラグインに対するワイヤリング

src/main/resources/META-INF/gradle-plugins/org.samples.greeting.properties

```
implementation-class=org.gradle.GreetingPlugin
```

プロパティファイルの名前がプラグインIDと一致していてリソースフォルダに配置されていること、
`implementation-class`プロパティに`Plugin`
の実装クラスを指定していることに注意してください。

59.5.1. Creating a plugin id

Plugin ids are fully qualified in a manner similar to Java packages (i.e. a reverse domain name). This helps to avoid collisions and provides a way to group plugins with similar ownership.

Your plugin id should be a combination of components that reflect namespace (a reasonable pointer to you or your organization) and the name of the plugin it provides. For example if you had a Github account named “foo” and your plugin was named “bar”, a suitable plugin id might be `com.github.foo.bar`. Similarly, if the plugin was developed at the baz organization, the plugin id might be `org.baz.bar`.

Plugin ids should conform to the following:

- May contain any alphanumeric character, '.', and '-'.
- Must contain at least one '.' character separating the namespace from the name of the plugin.
- Conventionally use a lowercase reverse domain name convention for the namespace.
- Conventionally use only lowercase characters in the name.
- `org.gradle` and `com.gradleware` namespaces may not be used.
- Cannot start or end with a '.' character.
- Cannot contain consecutive '.' characters (i.e. '..').

Although there are conventional similarities between plugin ids and package names, package names are generally more detailed than is necessary for a plugin id. For instance, it might seem reasonable to add “gradle” as a component of your plugin id, but since plugin ids are only used for Gradle plugins, this would be superfluous. Generally, a namespace that identifies ownership and a name are all that are needed for a good plugin id.

59.5.2. Publishing your plugin

If you are publishing your plugin internally for use within your organization, you can publish it like any other code artifact. See the ivy and maven chapters on publishing artifacts.

If you are interested in publishing your plugin to be used by the wider Gradle community, you can publish it to the Gradle plugin portal. This site provides the ability to search for and gather information about plugins contributed by the Gradle community. See the instructions here on how to make your plugin available on this site.

59.5.3. 別のプロジェクトでプラグインを使う

ビルドスクリプトでプラグインを使うためには、ビルドスクリプトのクラスパスにプラグインクラスを追
このために、「ビルドスクリプトで外部ライブラリを使うときの依存関係設定」で説明したように、bui
ブ ロ ッ ク が 使 え ま す
次のサンプルは、プラグインを含むJARがローカルリポジトリに発行済みのときにこれを行う方法を示し

例59.7 別のプロジェクトでカスタムプラグインを使う

build.gradle

```
buildscript {
    repositories {
        maven {
            url uri('../repo')
        }
    }
    dependencies {
        classpath group: 'org.gradle', name: 'customPlugin',
                  version: '1.0-SNAPSHOT'
    }
}
apply plugin: 'org.samples.greeting'
```

Alternatively, if your plugin is published to the plugin portal, you can use the incubating plugins DSL (see 「Applying plugins with the plugins DSL」) to apply the plugin:

例59.8 Applying a community plugin with the plugins DSL

build.gradle

```
plugins {
    id "com.jfrog.bintray" version "0.4.1"
}
```

59.5.4. プラグインに対するテストの作成

プラグイン実装をテストするときには、`Project`インスタンスを生成するために `ProjectBuilder` クラスが利用できます。

例59.9 カスタムプラグインのテスト

src/test/groovy/org/gradle/GreetingPluginTest.groovy

```
class GreetingPluginTest {
    @Test
    public void greeterPluginAddsGreetingTaskToProject() {
        Project project = ProjectBuilder.builder().build()
        project.apply plugin: 'org.samples.greeting'

        assertTrue(project.tasks.hello instanceof GreetingTask)
    }
}
```

59.5.5. Using the Java Gradle Plugin development plugin

You can use the incubating Java Gradle Plugin development plugin to eliminate some of the boilerplate declarations in your build script and provide some basic validations of plugin metadata. This plugin will automatically apply the Java plugin, add the `gradleApi()` dependency to the compile configuration, and perform plugin metadata validations as part of the `jar` task execution.

例59.10 Using the Java Gradle Plugin Development plugin

build.gradle

```
apply plugin: 'java-gradle-plugin'
```

59.6. 複数のドメインオブジェクトの管理

Gradleはオブジェクトのコレクションを管理するためのユーティリティクラスを提供しており、それらに

例59.11 ドメインオブジェクトの管理

build.gradle

```
apply plugin: DocumentationPlugin

books {
    quickStart {
        sourceFile = file('src/docs/quick-start')
    }
    userGuide {

    }
    developerGuide {

    }
}

task books << {
    books.each { book ->
        println "$book.name -> $book.sourceFile"
    }
}

class DocumentationPlugin implements Plugin<Project> {
    void apply(Project project) {
        def books = project.container(Book)
        books.all {
            sourceFile = project.file("src/docs/$name")
        }
        project.extensions.books = books
    }
}

class Book {
    final String name
    File sourceFile

    Book(String name) {
        this.name = name
    }
}
```

gradle -q books の出力

```
> gradle -q books
developerGuide -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPlugin
quickStart -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWith
userGuide -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWithD
```

Project.container()メソッドは、オブジェクトの管理やコンフィグレーションに便利な多くのメソッドを提供するNamedDomainObjectContainerのインスタンスを生成します。 project.containerのメソッドでタイプを扱うためには、 オブジェクトのユニークかつ固定の名前を提供する“name”プロパティを公開しなければなりません。 project.container(Class)メソッドおよびそのバリエーションは、単一のstringを引数として取るクラスのコンストラクタの実行を試みることで新しいインスタンスを生成このとき、引数はオブジェクトの名前となることが期待されます。

カスタムのインスタンス化戦略を可能にするproject.container
メソッドのバリエーションについては、上記のリンクを参照してください。

60 ビルドロジックの体系化

Gradleでは、様々な方法でビルドロジックを体系化できます。最初は、すべてのビルドロジックがタスク
[4 1]

Gradleでは、決まったディレクトリにクラスを放り込んでおけば自動的にコンパイルしてクラスパスに追

ビルドロジックを体系化する方法について、いくつか例を挙げます。

- POGOs。plain old Groovy Objects (POGOs) をビルドスクリプト内で直接宣言して使用できます。結局のところビルドスクリプトはGroovyで書く
- プロパティとメソッドの継承 。マルチプロジェクトでは、すべてのサブプロジェクトが親プロジェクトからプロパティとメソッド
- 設定のインジェクション 。マルチプロジェクトでは、あるプロジェクト(普通ルートプロジェクトですが)が別のプロジェクトに
- buildSrcプロジェクト 。ビルドスクリプトで使いたいクラスのソースを決まったディレクトリに入れておけば、Gradleは自
- 共有スクリプト 。外部のスクリプトファイルに共通で使う設定を定義して、それをいろいろなプロジェクトのビルド
- カスタムタスク 。ビルドロジックをカスタムタスクにすれば、いろんな場所でそのロジックを再利用できます。
- カスタムプラグイン 。ビルドロジックをカスタムプラグインにすれば、いろんなプロジェクトでそのプラグインを適用す buildSrcプロジェクトや外部ライブラリにそのプラグインを含めれば簡単に追加できます。
- 外部ビルドの実行。現在のビルドから、外部の別のビルドを呼び出すことができます。
- 外部ライブラリ。外部のライブラリを直接ビルドで使うことができます。

60.1. プロパティとメソッドの継承

プロジェクト内で定義されたプロパティとメソッドは、そのプロジェクトのサブプロジェクトでもすべて

例60.1 プロパティとメソッドの継承を使う

build.gradle

```
// Define an extra property
ext.srcDirName = 'src/java'

// Define a method
def getSrcDir(project) {
    return project.file(srcDirName)
}
```

child/build.gradle

```
task show << {
    // Use inherited property
    println 'srcDirName: ' + srcDirName

    // Use inherited method
    File srcDir = getSrcDir(project)
    println 'srcDir: ' + rootProject.relativePath(srcDir)
}
```

gradle -q show の出力

```
> gradle -q show
srcDirName: src/java
srcDir: child/src/java
```

60.2. 設定のインジェクション

「クロスプロジェクト設定」や「サブプロジェクトの設定」
で紹介した設定のインジェクションを使って、複数のプロジェクトにプロパティとメソッドを注入できま

例60.2 プロパティとメソッドのインジェクション

build.gradle

```
subprojects {
    // Define a new property
    ext.srcDirName = 'src/java'

    // Define a method using a closure as the method body
    ext.srcDir = { file(srcDirName) }

    // Define a task
    task show << {
        println 'project: ' + project.path
        println 'srcDirName: ' + srcDirName
        File srcDir = srcDir()
        println 'srcDir: ' + rootProject.relativePath(srcDir)
    }
}

// Inject special case configuration into a particular project
project(':child2') {
    ext.srcDirName = "$srcDirName/legacy"
}
```

child1/build.gradle

```
// Use injected property and method. Here, we override the injected value
srcDirName = 'java'
def dir = srcDir()
```

gradle -q show の出力

```
> gradle -q show
project: :child1
srcDirName: java
srcDir: child1/java
project: :child2
srcDirName: src/java/legacy
srcDir: child2/src/java/legacy
```

60.3. buildSrc プロジェクトのソースをビルドする

Gradleは実行されたときにbuildSrcというディレクトリがないか確認します。
そして、中のコードを自動的にコンパイル、テストして、ビルドスクリプトのクラスパスに追加してくれ
何かを設定する必要は一切ありません。このディレクトリは、カスタムタスクやプラグインを追加するの

マルチプロジェクトのビルドでは、ルートプロジェクトに一つだけbuildSrc
ディレクトリを作成できます。

buildSrcプロジェクトには、次のビルドスクリプトがデフォルトで適用されます。

図60.1 デフォルトのbuildSrcビルドスクリプト

```
apply plugin: 'groovy'
dependencies {
    compile gradleApi()
    compile localGroovy()
}
```

つまり、Java/Groovyプロジェクトのレイアウト規約(表23.4「Javaプラグイン - デ フ ォ ル ト プ ロ ジ ェ ク ト レ イ ア ウ ト」参照)に従って、このディレクトリにソースコードを置くだけでいいということです。

さらに柔軟性が必要なときは、自分で`build.gradle`を用意することもできます。
上記のデフォルトビルドスクリプトは、この場合でも適用されます。つまり、必要なものだけを追加で定
APIへの依存関係は、デフォルトのビルドスクリプトで宣言されているため、ここでは宣言する必要があ

例60.3 カスタムbuildSrcビルドスクリプト

buildSrc/build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    testCompile 'junit:junit:4.11'
}
```

`buildSrc`
プロジェクトは、マルチプロジェクトにすることができます。このプロジェクトは、普通のGradleマルチ
ただし、実際のビルドのクラスパスに置きたい全てのプロジェクトを、buildSrcルートプロジェクトの
`r u n t i m e`
依存関係にしなければなりません。これは、エクスポートしたい全てのプロジェクトの設定に、以下のコ

例60.4 buildSrcルートプロジェクトにサブプロジェクトを追加する

buildSrc/build.gradle

```
rootProject.dependencies {
  runtime project(path)
}
```

ノ ー ト :
本例のソースコードは、Gradleのバイナリ配布物またはソース配布物に含まれています。以下の場所
samples/multiProjectBuildSrc

60.4. 別のGradleビルドを、現在のビルドから呼び出し

G r a d l e B u i l d
で別のビルドを呼び出して実行することができます。そのとき、呼び出すビルドのディレクトリやビルト

例60.5 別のビルドを呼び出す

build.gradle

```
task build(type: GradleBuild) {
    buildFile = 'other.gradle'
    tasks = ['hello']
}
```

other.gradle

```
task hello << {
    println "hello from the other build."
}
```

gradle -q build の出力

```
> gradle -q build
hello from the other build.
```

60.5. ビルドスクリプトで外部ライブラリを使うときの

ビルドスクリプトを実行するときに外部のライブラリを使いたいなら、そのライブラリをビルドスクリプ
b u i l d s c r i p t ( )
メソッドにクロージャを渡し、その中でビルドスクリプトのクラスパスを設定できます。

例60.6 ビルドスクリプトのクラスパスを宣言する

build.gradle

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'commons-codec', name: 'commons-codec', version: '1.2'
    }
}
```

buildscript()メソッドに渡すクロージャは、ScriptHandler
のインスタンスにパラメータなどを設定するものです。ビルドスクリプトのクラスパスを追加するには、
c l a s s p a t h

で依存関係を宣言しましょう。形式はJavaでコンパイル時のクラスパスを宣言するときなどと同じです。「依存関係の定義方法 」
に記載されているタイプの依存関係は、「プロジェクトへの依存」を除けばすべて使えます。

外部ライブラリのクラスをビルドスクリプトのクラスパスを追加すれば、他のクラスと同じようにスクリ

例60.7 外部ライブラリをビルドスクリプトで使用する

build.gradle

```
import org.apache.commons.codec.binary.Base64

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'commons-codec', name: 'commons-codec', version: '1.2'
    }
}

task encode << {
    def byte[] encodedString = new Base64().encode('hello world\n'.getBytes())
    println new String(encodedString)
}
```

gradle -q encode の出力

```
> gradle -q encode
aGVsbG8gd29ybGQK
```

マルチプロジェクトの場合、プロジェクトのビルドスクリプトでクラスパスを宣言すれば、そのサブプロ

60.6. Antオプショナルタスクの依存関係

何でかよく分からないのですが、Antのオプショナルタスクでは先ほどの方法で追加した外部ライブラリ

例60.8 Ant optional dependencies

build.gradle

```
configurations {
    ftpAntTask
}

dependencies {
    ftpAntTask("org.apache.ant:ant-commons-net:1.9.3") {
        module("commons-net:commons-net:1.4.1") {
            dependencies "oro:oro:2.0.8:jar"
        }
    }
}

task ftp << {
    ant {
        taskdef(name: 'ftp',
                classname: 'org.apache.tools.ant.taskdefs.optional.net.FTP',
                classpath: configurations.ftpAntTask.asPath)
        ftp(server: "ftp.apache.org", userid: "anonymous", password: "me@myorg.com"
            fileset(dir: "htdocs/manual")
        }
    }
}
```

This is also a good example for the usage of client modules. The POM file in Maven Central for the ant-commons-net task does not provide the right information for this use case.

これはクライアントモジュールのいい使用例でもあります。mavenのセントラルリポジトリにあるant-c
`pom.xml`は、このユースケースのための正しい情報を提供していません。

60.7. まとめ

Gradleは、ビルドロジックを体系化するための様々な方法を提供しています。自分の分野に合った方法を

---

[41] 一つの単純なクラスから、複雑なクラスまで

61
初期化スクリプト

Gradleは、実行環境に合わせてビルドをカスタマイズするためのパワフルな仕組みを提供しています。ま

Note that this is completely different from the “init” task provided by the “build-init” incubating plugin (see Chapter 47, Build Init Plugin).

61.1. 基本的な使い方

初期化スクリプト(init scripts)は、Gradleで使う他のスクリプトと似たものですが、ビルドの開始前に実行されます。使用する場面と

- 企業内で適用される設定、たとえば、カスタムプラグインのダウンロード先などを設定する。
- 実行環境に依るプロパティをセットする。たとえば、開発者のマシンとインテグレーションサーバー
- ビルド時に要求される、ユーザーの個人情報をセットする。リポジトリやデータベースの認証情報な
- マシン固有の情報を定義する。JDKのインストール場所など。
- ビルドのリスナーを登録する。Gradleのイベントをリスンしたい外部ツールにとって便利かもしれま
- ビルドのロガーを登録する。Gradleがイベントを記録する方法をカスタマイズしたいことがあるかも

初期化スクリプトの主な制限は、buildSrc プロジェクトにアクセスできないことです（この機能の詳細は「buildSrc プロジェクトのソースをビルドする」を参照してください）。

61.2. 初期化スクリプトを使う

初期化スクリプトを使うには、いくつかの方法があります。

- コマンドラインでファイルを指定する。-Iか--init-script オプションに初期化スクリプトのパスを渡します。このオプションを複数並べて、初期化スクリプト
- init.gradleという名前のファイルを*USER_HOME*/.gradle/ディレクトリに置く。
- .gradleという拡張子のファイルを*USER_HOME*/.gradle/init.d/ディレクトリに置く。
- .gradleという拡張子のファイルを*GRADLE_HOME*/init.d/ ディレクトリ、つまりGradleのディストリビューションの中に置く。この機能により、独自のビルド Gradleラッパー 機能を組み合わせれば、企業内のすべてのビルドで独自のビルドロジックを使えるようにもできます。

複数の初期化スクリプトが見つかった場合、そのすべてが上で記載した順番で実行されます。同じディレ

61.3. 初期化スクリプトを記述する

Gradleのビルドスクリプトと同様、初期化スクリプトもGroovyスクリプトです。すべての初期化スクリ
G r a d l e
のインスタンスが割り当てられ、初期化スクリプト内でのプロパティアクセス、メソッドコールはすべて
Gradleインスタンスに委譲されます。

また、すべての初期化スクリプトはScriptインターフェースを実装しています。

61.3.1. 初期化スクリプトでプロジェクトを設定する

初期化スクリプトを使ってプロジェクトの設定を変更することができます。これは、マルチプロジェクト
前 に
、初期化スクリプトで追加的な設定を行うための方法を示すものです。このサンプルでは、ある環境での

例61.1 プロジェクト評価前に初期化スクリプトで追加的な設定を行う

build.gradle

```
repositories {
    mavenCentral()
}

task showRepos << {
    println "All repos:"
    println repositories.collect { it.name }
}
```

init.gradle

```
allprojects {
    repositories {
        mavenLocal()
    }
}
```

gradle --init-script init.gradle -q showRepos の出力

```
> gradle --init-script init.gradle -q showRepos
All repos:
[MavenLocal, MavenRepo]
```

61.4. 初期化スクリプトの外部依存関係

「ビルドスクリプトで外部ライブラリを使うときの依存関係設定」
でビルドスクリプトに依存関係を追加する方法が解説されていますが、初期化スクリプトでも似たような
i n i t s c r i p t ( )
メソッドに初期化スクリプトのクラスパスを宣言したクロージャを引き渡してください。

例61.2 初期化スクリプトの外部依存関係定義

init.gradle

```
initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'org.apache.commons', name: 'commons-math', version: '2.0'
    }
}
```

`initscript()`に渡したクロージャは、`ScriptHandler`
インスタンスを設定します。これは、たとえば、Javaのコンパイルクラスパスを宣言するのと同じ方法て
「依存関係の定義方法」
に載っている依存関係は、プロジェクト依存関係を除いてすべて使うことができます。

このようにして初期化スクリプトのクラスパスを宣言すれば、初期化スクリプト内でどんなクラスでもイ

例61.3 外部依存関係を持つ初期化スクリプト

init.gradle

```
import org.apache.commons.math.fraction.Fraction

initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'org.apache.commons', name: 'commons-math', version: '2.0'
    }
}

println Fraction.ONE_FIFTH.multiply(2)
```

gradle --init-script init.gradle -q doNothing の出力

```
> gradle --init-script init.gradle -q doNothing
2 / 5
```

61.5. Init script plugins

Similar to a Gradle build script or a Gradle settings file, plugins can be applied on init scripts.

例61.4 Using plugins in init scripts

init.gradle

```
apply plugin:EnterpriseRepositoryPlugin

class EnterpriseRepositoryPlugin implements Plugin<Gradle> {

    private static String ENTERPRISE_REPOSITORY_URL = "https://repo.gradle.org/gradle

    void apply(Gradle gradle) {
        // ONLY USE ENTERPRISE REPO FOR DEPENDENCIES
        gradle.allprojects{ project ->
            project.repositories {

                // Remove all repositories not pointing to the enterprise repository 
                all { ArtifactRepository repo ->
                    if (!(repo instanceof MavenArtifactRepository) ||
                          repo.url.toString() != ENTERPRISE_REPOSITORY_URL) {
                        project.logger.lifecycle "Repository ${repo.url} removed. Onl
                        remove repo
                    }
                }

                // add the enterprise repository
                maven {
                    name "STANDARD_ENTERPRISE_REPO"
                    url ENTERPRISE_REPOSITORY_URL
                }
            }
        }
    }
}
```

build.gradle

```
repositories{
    mavenCentral()
}

 task showRepositories << {
    repositories.each{
        println "repository: ${it.name} ('${it.url}')"
    }
}
```

gradle -q -I init.gradle showRepositories の出力

```
> gradle -q -I init.gradle showRepositories
repository: STANDARD_ENTERPRISE_REPO ('https://repo.gradle.org/gradle/repo')
```

The plugin in the init script ensures that only a specified repository is used when running the build.

When applying plugins within the init script, Gradle instantiates the plugin and calls the plugin instance's `Plugin.apply()` method. The `gradle` object is passed as a parameter, which can be used to configure all aspects of a build. Of course, the applied plugin can be resolved as an external dependency as described in 「初期化スクリプトの外部依存関係」

62
Gradleラッパー

Gradleラッパー（以後ラッパーとします）は、Gradleのビルドを実行する方法として推奨されているもの
ラッパーは、バッチスクリプト（Windows用）とシェルスクリプト（その他のOS用）で提供されます。
このラッパーを使ってGradleビルドを実行すると、Gradleが自動的にダウンロードされ、それを使ってb

ラッパーは、バージョン管理システムにチェックインするべき
です。ラッパーと一緒にプロジェクトを配布することで、前もってGradleをインストールすることなく、
さらにいいことに、ラッパーを使ってもらうことで、ビルドに使用されるGradleのバージョンをプロジェ
もちろん、ラッパーは継続的インテグレーション
サーバーで作業するときも非常に便利です。サーバーでビルドの設定を行う必要がないからです。

ラッパーを自分のプロジェクトにインストールするには、`Wrapper`
タスクをビルドスクリプトに追加、設定してから、タスクを実行します。

例62.1 ラッパータスク

build.gradle

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.0'
}
```

このタスクを実行すると、(wrapperタスクのデフォルト設定が使われていれば、)プロジェクトディレク

例62.2 ラッパーにより生成されるファイル

Build layout

```
simple/
  gradlew
  gradlew.bat
  gradle/wrapper/
    gradle-wrapper.jar
    gradle-wrapper.properties
```

これらのファイルは、すべてバージョン管理システムに格納するべき
です。この作業は一回だけ行えば大丈夫です。これらのファイルがプロジェクトに追加されたら、以後の
gradlewコマンドで行ってください。 gradlewコマンドは、gradleコマンドと全く
同じように使うことができます。

Gradleを新しいバージョンに変更する場合でも、wrapperタスクを再実行する必要はありません。 `grad`
ファイルの、関係するエントリを変更するだけです。

ただ、例えばgradle-wrapperの機能が改良された場合などは、wrapperファイルを再作成してください。

62.1. 設定

g r a d l e w
を実行すると、ラッパーはまず、指定されたディストリビューションが利用できるかどうか確認します。
利用できなければディストリビューションを新しくダウンロードし、利用可能ならgradlew
コマンドに指定された引数を、そのディストリビューションのgradle
コマンドにそのまま渡して実行します。

W r a p p e r
タスクを設定する際は、使いたいGradleのバージョンを指定することができます。その場合、作成され
g r a d l e w
コマンドはGradleリポジトリから適切なディストリビューションを選んでダウンロードします。
または、Gradleのディストリビューションをダウンロードするurlを直接指定することもできます。この
gradlewコマンドはディストリビューションのダウンロードにそのURLを使用します。
Gradleのバージョンもダウンロード先も指定しなかった場合は、ラッパーファイルを作成したGradleの

For the details on how to configure the wrapper, see the `Wrapper` class in the API documentation.

ラッパーの詳細な設定方法についてはWrapperを参照してください。

g r a d l e w
コマンドでプロジェクトをビルドする際、ダウンロード処理を一切走らせたくない場合は、ラッパーの設
相対パスもサポートされています。gradle-wrapper.properties
からの相対パスでディストリビューションファイルを指定することができます。

ラッパーを使ってビルドする場合、マシンにインストールされているGradleディストリビューションが

62.2. Unixファイルパーミッション

Wrapperタスクは、生成したgradlew
*NIXコマンドを実行できるよう、適切なファイルパーミッションを追加します。
Subversionはちゃんとこのファイルパーミッションを保存してくれますが、他のバージョン管理システム
確実を期すならsh gradlewで実行してください。

63
Embedding Gradle

63.1. Introduction to the Tooling API

The 1.0 milestone 3 release brought a new API called the tooling API, which you can use for embedding Gradle into your own custom software. This API allows you to execute and monitor builds, and to query Gradle about the details of a build. The main audience for this API will be IDEs, CI servers, other UI authors, or integration testing of your Gradle plugins. However, it is open for anyone who needs to embed Gradle in their application.

A fundamental characteristic of the tooling API is that it operates in a version independent way. This means that you can use the same API to work with different target versions of Gradle. The tooling API is Gradle wrapper aware and, by default, uses the same target Gradle version as that used by the wrapper-powered project.

Some features that the tooling API provides today:

- You can query Gradle for the details of a build, including the project hierarchy and the project dependencies, external dependencies (including source and Javadoc jars), source directories and tasks of each project.
- You can execute a build and listen to stdout and stderr logging and progress (e.g. the stuff shown in the 'status bar' when you run on the command line).
- Tooling API can download and install the appropriate Gradle version, similar to the wrapper. Bear in mind that the tooling API is wrapper aware so you should not need to configure a Gradle distribution directly.
- The implementation is lightweight, with only a small number of dependencies. It is also a well-behaved library, and makes no assumptions about your classloader structure or logging configuration. This makes the API easy to bundle in your application.

In the future we may support other interesting features:

- Performance. The API gives us the opportunity to do lots of caching, static analysis and preemptive work, to make things faster for the user.
- Better progress monitoring and build cancellation. For example, allowing test execution to be monitored.
- Notifications when things in the build change, so that UIs and models can be updated. For example, your Eclipse or IDEA project will update immediately, in the background.
- Validating and prompting for user supplied configuration.
- Prompting for and managing user credentials.

63.2. Tooling API and the Gradle Build Daemon

Please take a look at 19章Gradleデーモン . The Tooling API uses the daemon all the time. In fact, you cannot officially use the Tooling API without the daemon. This means that subsequent calls to the Tooling API, be it model building requests or task executing requests can be executed in the same long-living process. 19章Gradleデーモン contains more details about the daemon, specifically information on situations when new daemons are forked.

63.3. Quickstart

As the tooling API is an interface for developers, the Javadoc is the main documentation for it. This is exactly our intention - we don't expect this chapter to grow very much. Instead we will add more code samples and improve the Javadoc documentation. The main entry point to the tooling API is the `GradleConnector`. You can navigate from there to find code samples and other instructions. Another very important set of resources are the samples that live in “`$gradleHc`” . These samples also specify all of the required dependencies for the Tooling API, along with the suggested repositories to obtain the jars from.

64
Comparing Builds

> Build comparison support is an incubating feature. This means that it is incomplete and not yet at regular Gradle production quality. This also means that this Gradle User Guide chapter is a work in progress.

Gradle provides support for comparing the outcomes (e.g. the produced binary archives) of two builds. There are several reasons why you may want to compare the outcomes of two builds. You may want to compare:

- A build with a newer version of Gradle than it's currently using (i.e. upgrading the Gradle version).
- A Gradle build with a build executed by another tool such as Apache Ant, Apache Maven or something else (i.e. migrating to Gradle).
- The same Gradle build, with the same version, before and after a change to the build (i.e. testing build changes).

By comparing builds in these scenarios you can make an informed decision about the Gradle upgrade, migration to Gradle or build change by understanding the differences in the outcomes. The comparison process produces a HTML report outlining which outcomes were found to be identical and identifying the differences between non-identical outcomes.

64.1. Definition of terms

The following are the terms used for build comparison and their definitions.

"Build"
: In the context of build comparison, a build is not necessarily a Gradle build. It can be any invokable "process" that produces observable "outcomes". At least one of the builds in a comparison will be a Gradle build.

"Build Outcome"
: Something that happens in an observable manner during a build, such as the creation of a zip file or test execution. These are the things that are compared.

"Source Build"
: The build that comparisons are being made against, typically the build in its "current" state. In other words, the left hand side of the comparison.

"Target Build"
: The build that is being compared to the source build, typically the "proposed" build. In other words, the right hand side of the comparison.

"Host Build"
: The Gradle build that executes the comparison process. It may be the same project as either the "target" or "source" build or may be a completely separate project. It does not need to be the same Gradle version as the "source" or "target" builds. The host build must be run with Gradle 1.2 or newer.

"Compared Build Outcome"
: Build outcomes that are intended to be logically equivalent in the "source" and "target" builds, and are therefore meaningfully comparable.

"Uncompared Build Outcome"
: A build outcome is uncompared if a logical equivalent from the other build cannot be found (e.g. a build produces a zip file that the other build does not).

"Unknown Build Outcome"
: A build outcome that cannot be understood by the host build. This can occur when the source or target build is a newer Gradle version than the host build and that Gradle version exposes new outcome types. Unknown build outcomes can be compared in so far as they can be identified to be logically equivalent to an unknown build outcome in the other build, but no meaningful comparison of what the build outcome actually is can be performed. Using the latest Gradle version for the host build will avoid encountering unknown build outcomes.

64.2. Current Capabilities

As this is an incubating feature, a limited set of the eventual functionality has been implemented at this time.

64.2.1. Supported builds

Only support for comparing Gradle builds is available at this time. Both the source and target build must execute with Gradle newer or equal to version `1.0`. The host build must be at least version `1.2`.

Future versions will provide support for executing builds from other build systems such as Apache Ant or Apache Maven, as well as support for executing arbitrary processes (e.g. shell script based builds)

64.2.2. Supported build outcomes

Only support for comparing build outcomes that are `zip` archives is supported at this time. This includes `jar`, `war` and `ear` archives.

Future versions will provide support for comparing outcomes such as test execution (i.e. which tests were executed, which tests failed, etc.)

64.3. Comparing Gradle Builds

The `compare-gradle-builds` plugin can be used to facilitate a comparison between two Gradle builds. The plugin adds a `CompareGradleBuilds` task named “`compareGradleBuilds`” to the project. The configuration of this task specifies what is to be compared. By default, it is configured to compare the current build with itself using the current Gradle version by executing the tasks: “`clean assemble`”.

```
apply plugin: 'compare-gradle-builds'
```

This task can be configured to change what is compared.

```
compareGradleBuilds {
    sourceBuild {
        projectDir "/projects/project-a"
        gradleVersion "1.1"
    }
    targetBuild {
        projectDir "/projects/project-b"
        gradleVersion "1.2"
    }
}
```

The example above specifies a comparison between two different projects using two different Gradle versions.

64.3.1. Trying Gradle upgrades

You can use the build comparison functionality to very quickly try a new Gradle version with your build.

To try your current build with a different Gradle version, simply add the following to the `build.grad` of the root project.

```
apply plugin: 'compare-gradle-builds'

compareGradleBuilds {
    targetBuild.gradleVersion = "«gradle version»"
}
```

Then simply execute the compareGradleBuilds task. You will see the console output of the “source” and “target” builds as they are executing.

64.3.2. The comparison “result”

If there are any differences between the compared outcomes, the task will fail. The location of the HTML report providing insight into the comparison will be given. If all compared outcomes are found to be identical, and there are no uncompared outcomes, and there are no unknown build outcomes, the task will succeed.

You can configure the task to not fail on compared outcome differences by setting the `ignoreFailu` property to true.

```
compareGradleBuilds {
    ignoreFailures = true
}
```

64.3.3. Which archives are compared?

For an archive to be a candidate for comparison, it must be added as an artifact of the archives configuration. Take a look at 52章アーティファクトの公開 for more information on how to configure and add artifacts.

The archive must also have been produced by a `Zip`, `Jar`, `War`, `Ear` task. Future versions of Gradle will support increased flexibility in this area.

65

Ivy Publishing (new)

This chapter describes the new incubating Ivy publishing support provided by the “ivy-publis
” plugin. Eventually this new publishing support will replace publishing via the `Upload` task.

If you are looking for documentation on the original Ivy publishing support using the `Upload` task please see 52章アーティファクトの公開 .

This chapter describes how to publish build artifacts in the Apache Ivy format, usually to a repository for consumption by other builds or projects. What is published is one or more artifacts created by the build, and an Ivy module descriptor (normally `ivy.xml`) that describes the artifacts and the dependencies of the artifacts, if any.

A published Ivy module can be consumed by Gradle (see 51章依存関係の管理) and other tools that understand the Ivy format.

65.1. The “ivy-publish” Plugin

The ability to publish in the Ivy format is provided by the “`ivy-publish`” plugin.

The “`publishing`” plugin creates an extension on the project named “`publishing`” of type `PublishingExtension`. This extension provides a container of named publications and a container of named repositories. The “`ivy-publish`” plugin works with `IvyPublication` publications and `IvyArtifactRepository` repositories.

Example 65.1. Applying the “ivy-publish” plugin

build.gradle

```
apply plugin: 'ivy-publish'
```

Applying the “`ivy-publish`” plugin does the following:

- Applies the “`publishing`” plugin
- Establishes a rule to automatically create a `GenerateIvyDescriptor` task for each `IvyPublication` added (see Section 65.2, “Publications”).
- Establishes a rule to automatically create a `PublishToIvyRepository` task for the

combination of each `IvyPublication` added (see Section 65.2, “Publications”), with each `IvyArtifactRepository` added (see Section 65.3, “Repositories”).

65.2. Publications

> If you are not familiar with project artifacts and configurations, you should read 52章 アーティファクトの公開 , which introduces these concepts. This chapter also describes “publishing artifacts” using a different mechanism than what is described in this chapter. The publishing functionality described here will eventually supersede that functionality.

Publication objects describe the structure/configuration of a publication to be created. Publications are published to repositories via tasks, and the configuration of the publication object determines exactly what is published. All of the publications of a project are defined in the `PublishingExtension.getPublications()` container. Each publication has a unique name within the project.

For the “`ivy-publish`” plugin to have any effect, an `IvyPublication` must be added to the set of publications. This publication determines which artifacts are actually published as well as the details included in the associated Ivy module descriptor file. A publication can be configured by adding components, customizing artifacts, and by modifying the generated module descriptor file directly.

65.2.1. Publishing a Software Component

The simplest way to publish a Gradle project to an Ivy repository is to specify a `SoftwareComponent` to publish. The components presently available for publication are:

Table 65.1. Software Components

Name	Provided By	Artifacts	Dependencies
`java`	Java Plugin	Generated jar file	Dependencies from 'runtime' configuration
`web`	War Plugin	Generated war file	No dependencies

In the following example, artifacts and runtime dependencies are taken from the `java` component, which is added by the `Java Plugin`.

Example 65.2. Publishing a Java module to Ivy

build.gradle

```
publications {
        ivyJava(IvyPublication) {
            from components.java
        }
    }
```

65.2.2. Publishing custom artifacts

It is also possible to explicitly configure artifacts to be included in the publication. Artifacts are commonly supplied as raw files, or as instances of `AbstractArchiveTask` (e.g. Jar, Zip).

For each custom artifact, it is possible to specify the `name`, `extension`, `type`, `classifier` and `conf` values to use for publication. Note that each artifacts must have a unique name/classifier/extension combination.

Configure custom artifacts as follows:

Example 65.3. Publishing additional artifact to Ivy

build.gradle

```
task sourceJar(type: Jar) {
        from sourceSets.main.java
        classifier "source"
    }
    publishing {
        publications {
            ivy(IvyPublication) {
                from components.java
                artifact(sourceJar) {
                    type "source"
                    conf "runtime"
                }
            }
        }
    }
```

See the `IvyPublication` class in the API documentation for more detailed information on how artifacts can be customized.

65.2.3. Identity values for the published project

The generated Ivy module descriptor file contains an `<info>` element that identifies the module. The default identity values are derived from the following:

- `organisation` - `Project.getGroup()`
- `module` - `Project.getName()`
- `revision` - `Project.getVersion()`
- `status` - `Project.getStatus()`
- `branch` - (not set)

Overriding the default identity values is easy: simply specify the `organisation`, `module` or `revision` attributes when configuring the `IvyPublication`. The `status` and `branch` attributes can be set via the `descriptor` property (see `IvyModuleDescriptorSpec`). The `descriptor` property can also be used to add additional custom elements as children of the `<info>` element.

Example 65.4. customizing the publication identity

build.gradle

```
publishing {
        publications {
            ivy(IvyPublication) {
                organisation 'org.gradle.sample'
                module 'project1-sample'
                revision '1.1'
                descriptor.status = 'milestone'
                descriptor.branch = 'testing'
                descriptor.extraInfo 'http://my.namespace', 'myElement', 'Some value'

                from components.java
            }
        }
    }
```

Gradle will handle any valid Unicode character for organisation, module and revision (as well as artifact name, extension and classifier). The only values that are explicitly prohibited are '\', '/' and any ISO control character. The supplied values are validated early during publication.

> Certain repositories are not able to handle all supported characters. For example, the ':' character cannot be used as an identifier when publishing to a filesystem-backed repository on Windows.

65.2.4. Modifying the generated module descriptor

At times, the module descriptor file generated from the project information will need to be tweaked before publishing. The "`ivy-publish`" plugin provides a hook to allow such modification.

Example 65.5. Customizing the module descriptor file

build.gradle

```
publications {
        ivyCustom(IvyPublication) {
            descriptor.withXml {
                asNode().info[0].appendNode('description',
                                            'A demonstration of ivy descriptor custom
            }
        }
    }
```

In this example we are simply adding a 'description' element to the generated Ivy dependency descriptor, but this hook allows you to modify any aspect of the generated descriptor. For example, you could replace the version range for a dependency with the actual version used to produce the build.

See `IvyModuleDescriptorSpec.withXml()` in the API documentation for more information.

It is possible to modify virtually any aspect of the created descriptor should you need to. This

means that it is also possible to modify the descriptor in such a way that it is no longer a valid Ivy module descriptor, so care must be taken when using this feature.

The identifier (organisation, module, revision) of the published module is an exception; these values cannot be modified in the descriptor using the `withXML` hook.

65.2.5. Publishing multiple modules

Sometimes it's useful to publish multiple modules from your Gradle build, without creating a separate Gradle subproject. An example is publishing a separate API and implementation jar for your library. With Gradle this is simple:

Example 65.6. Publishing multiple modules from a single project

build.gradle

```
task apiJar(type: Jar) {
        baseName "publishing-api"
        from sourceSets.main.output
        exclude '**/impl/**'
    }
    publishing {
        publications {
            impl(IvyPublication) {
                organisation 'org.gradle.sample.impl'
                module 'project2-impl'
                revision '2.3'

                from components.java
            }
            api(IvyPublication) {
                organisation 'org.gradle.sample'
                module 'project2-api'
                revision '2'
            }
        }
    }
```

If a project defines multiple publications then Gradle will publish each of these to the defined repositories. Each publication must be given a unique identity as described above.

65.3. Repositories

Publications are published to repositories. The repositories to publish to are defined by the `PublishingExtension.getRepositories()` container.

Example 65.7. Declaring repositories to publish to

build.gradle

```
repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url "$buildDir/repo"
        }
    }
```

The DSL used to declare repositories for publishing is the same DSL that is used to declare repositories for dependencies (`RepositoryHandler`). However, in the context of Ivy publication only the repositories created by the `ivy()` methods can be used as publication destinations. You cannot publish an `IvyPublication` to a Maven repository for example.

65.4. Performing a publish

The "`ivy-publish`" plugin automatically creates a `PublishToIvyRepository` task for each `IvyPublication` and `IvyArtifactRepository` combination in the `publishing.publications` and `publishing.repositories` containers respectively.

The created task is named "`publish«PUBNAME»PublicationTo«REPONAME»Repository`", which is "`publishIvyJavaPublicationToIvyRepository`" for this example. This task is of type `PublishToIvyRepository`.

Example 65.8. Choosing a particular publication to publish

build.gradle

```
apply plugin: 'java'
apply plugin: 'ivy-publish'

group = 'org.gradle.sample'
version = '1.0'

publishing {
    publications {
        ivyJava(IvyPublication) {
            from components.java
        }
    }
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url "$buildDir/repo"
        }
    }
}
```

gradle publishIvyJavaPublicationToIvyRepository の出力

```
> gradle publishIvyJavaPublicationToIvyRepository
:generateDescriptorFileForIvyJavaPublication
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar
:publishIvyJavaPublicationToIvyRepository

BUILD SUCCESSFUL

Total time: 1 secs
```

65.4.1. The “`publish`” lifecycle task

The “`publish`” plugin (that the “`ivy-publish`” plugin implicitly applies) adds a lifecycle task that can be used to publish all publications to all applicable repositories named “`publish`”.

In more concrete terms, executing this task will execute all `PublishToIvyRepository` tasks in the project. This is usually the most convenient way to perform a publish.

Example 65.9. Publishing all publications via the “publish” lifecycle task

gradle publish の出力

```
> gradle publish
:generateDescriptorFileForIvyJavaPublication
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar
:publishIvyJavaPublicationToIvyRepository
:publish

BUILD SUCCESSFUL

Total time: 1 secs
```

65.5. Generating the Ivy module descriptor file without publishing

At times it is useful to generate the Ivy module descriptor file (normally `ivy.xml`) without publishing your module to an Ivy repository. Since descriptor file generation is performed by a separate task, this is very easy to do.

The “`ivy-publish`” plugin creates one `GenerateIvyDescriptor` task for each registered `IvyPublication`, named “`generateDescriptorFileFor«PUBNAME»Publication`”, which will be “`generateDescriptorFileForIvyJavaPublication`” for the previous example of the “`ivyJava`” publication.

You can specify where the generated Ivy file will be located by setting the `destination` property on the generated task. By default this file is written to “`build/publications/«PUBNAME`” .

Example 65.10. Generating the Ivy module descriptor file

build.gradle

```
model {
    tasks.generateDescriptorFileForIvyCustomPublication {
        destination = file("$buildDir/generated-ivy.xml")
    }
}
```

gradle generateDescriptorFileForIvyCustomPublication の出力

```
> gradle generateDescriptorFileForIvyCustomPublication
:generateDescriptorFileForIvyCustomPublication

BUILD SUCCESSFUL

Total time: 1 secs
```

The "`ivy-publish`" plugin leverages some experimental support for late plugin configuration, and the `GenerateIvyDescriptor` task will not be constructed until the publishing extension is configured. The simplest way to ensure that the publishing plugin is configured when you attempt to access the `GenerateIvyDescriptor` task is to place the access inside a `model` block, as the example above demonstrates.

The same applies to any attempt to access publication-specific tasks like `PublishToIvyRepository`. These tasks should be referenced from within a `model` block.

65.6. Complete example

The following example demonstrates publishing with a multi-project build. Each project publishes a Java component and a configured additional source artifact. The descriptor file is customized to include the project description for each project.

Example 65.11. Publishing a Java module

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'ivy-publish'

    version = '1.0'
    group = 'org.gradle.sample'

    repositories {
        mavenCentral()
    }
    task sourceJar(type: Jar) {
        from sourceSets.main.java
        classifier "source"
    }
}

project(":project1") {
    description = "The first project"

    dependencies {
       compile 'junit:junit:4.11', project(':project2')
    }
}

project(":project2") {
    description = "The second project"

    dependencies {
       compile 'commons-collections:commons-collections:3.1'
    }
}

subprojects {
    publishing {
        repositories {
            ivy {
                // change to point to your repo, e.g. http://my.org/repo
                url "${rootProject.buildDir}/repo"
            }
        }
        publications {
            ivy(IvyPublication) {
                from components.java
                artifact(sourceJar) {
                    type "source"
                    conf "runtime"
                }
                descriptor.withXml {
                    asNode().info[0].appendNode('description', description)
                }
            }
        }
    }
}
```

The result is that the following artifacts will be published for each project:

- The Ivy module descriptor file: “`ivy-1.0.xml`” .
- The primary “jar” artifact for the Java component: “`project1-1.0.jar`” .
- The source “jar” artifact that has been explicitly configured: “`project1-1.0-source.jar`” .

When `project1` is published, the module descriptor (i.e. the `ivy.xml` file) that is produced will look like:

Example 65.12. Example generated ivy.xml

output-ivy.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="2.0">
  <info organisation="org.gradle.sample" module="project1" revision="1.0" status="int
    <description>The first project</description>
  </info>
  <configurations>
    <conf name="default" visibility="public" extends="runtime"/>
    <conf name="runtime" visibility="public"/>
  </configurations>
  <publications>
    <artifact name="project1" type="jar" ext="jar" conf="runtime"/>
    <artifact name="project1" type="source" ext="jar" conf="runtime" m:classifier="so
  </publications>
  <dependencies>
    <dependency org="junit" name="junit" rev="4.11" conf="runtime-&gt;default"/>
    <dependency org="org.gradle.sample" name="project2" rev="1.0" conf="runtime-&gt;d
  </dependencies>
</ivy-module>
```

Note that «`PUBLICATION-TIME-ST` in this example Ivy module descriptor will be the timestamp of when the descriptor was generated.

65.7. Future features

The “`ivy-publish`” plugin functionality as described above is incomplete, as the feature is still incubating. In upcoming Gradle releases, the functionality will be expanded to include (but not limited to):

- Convenient customization of module attributes (`module`, `organisation` etc.)
- Convenient customization of dependencies reported in `module descriptor`.
- Multiple discrete publications per project

66

Maven Publishing (new)

> This chapter describes the new incubating Maven publishing support provided by the “maven-publish” plugin. Eventually this new publishing support will replace publishing via the `Upload` task.
>
> If you are looking for documentation on the original Maven publishing support using the `Upload` task please see 52章アーティファクトの公開 .

This chapter describes how to publish build artifacts to an Apache Maven Repository. A module published to a Maven repository can be consumed by Maven, Gradle (see 51章依存関係の管理) and other tools that understand the Maven repository format.

66.1. The “maven-publish” Plugin

The ability to publish in the Maven format is provided by the “`maven-publish`” plugin.

The “`publishing`” plugin creates an extension on the project named “`publishing`” of type `PublishingExtension`. This extension provides a container of named publications and a container of named repositories. The “`maven-publish`” plugin works with `MavenPublication` publications and `MavenArtifactRepository` repositories.

Example 66.1. Applying the 'maven-publish' plugin

build.gradle

```
apply plugin: 'maven-publish'
```

Applying the “`maven-publish`” plugin does the following:

- Applies the “`publishing`” plugin
- Establishes a rule to automatically create a `GenerateMavenPom` task for each `MavenPublication` added (see Section 66.2, “Publications”).
- Establishes a rule to automatically create a `PublishToMavenRepository` task for the combination of each `MavenPublication` added (see Section 66.2, “Publications”), with each `MavenArtifactRepository` added (see Section 66.3, “Repositories”).
- Establishes a rule to automatically create a `PublishToMavenLocal` task for each

MavenPublication added (seeSection 66.2, “Publications”).

66.2. Publications

> If you are not familiar with project artifacts and configurations, you should read the 52章アーティファクトの公開 that introduces these concepts. This chapter also describes “publishing artifacts” using a different mechanism than what is described in this chapter. The publishing functionality described here will eventually supersede that functionality.

Publication objects describe the structure/configuration of a publication to be created. Publications are published to repositories via tasks, and the configuration of the publication object determines exactly what is published. All of the publications of a project are defined in the `PublishingExtension.getPublications()` container. Each publication has a unique name within the project.

For the “`maven-publish`” plugin to have any effect, a `MavenPublication` must be added to the set of publications. This publication determines which artifacts are actually published as well as the details included in the associated POM file. A publication can be configured by adding components, customizing artifacts, and by modifying the generated POM file directly.

66.2.1. Publishing a Software Component

The simplest way to publish a Gradle project to a Maven repository is to specify a `SoftwareComponent` to publish. The components presently available for publication are:

Table 66.1. Software Components

Name	Provided By	Artifacts	Dependencies
`java`	23章Javaプラグイン	Generated jar file	Dependencies from 'runtime' configuration
`web`	26章Warプラグイン	Generated war file	No dependencies

In the following example, artifacts and runtime dependencies are taken from the `java` component, which is added by the `Java Plugin`.

Example 66.2. Adding a MavenPublication for a Java component

build.gradle

```
publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java
        }
    }
}
```

66.2.2. Publishing custom artifacts

It is also possible to explicitly configure artifacts to be included in the publication. Artifacts are commonly supplied as raw files, or as instances of `AbstractArchiveTask` (e.g. Jar, Zip).

For each custom artifact, it is possible to specify the `extension` and `classifier` values to use for publication. Note that only one of the published artifacts can have an empty classifier, and all other artifacts must have a unique classifier/extension combination.

Configure custom artifacts as follows:

Example 66.3. Adding additional artifact to a MavenPublication

build.gradle

```
task sourceJar(type: Jar) {
    from sourceSets.main.allJava
}

publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java

            artifact sourceJar {
                classifier "sources"
            }
        }
    }
}
```

See the `MavenPublication` class in the API documentation for more information about how artifacts can be customized.

66.2.3. Identity values in the generated POM

The attributes of the generated `POM` file will contain identity values derived from the following project properties:

- `groupId` - `Project.getGroup()`
- `artifactId` - `Project.getName()`
- `version` - `Project.getVersion()`

Overriding the default identity values is easy: simply specify the `groupId`, `artifactId` or `version` attributes when configuring the `MavenPublication`.

Example 66.4. customizing the publication identity

build.gradle

```
publishing {
        publications {
            maven(MavenPublication) {
                groupId 'org.gradle.sample'
                artifactId 'project1-sample'
                version '1.1'

                from components.java
            }
        }
    }
```

Maven restricts 'groupId' and 'artifactId' to a limited character set (`[A-Za-z0-9_\\-.]+`) and Gradle enforces this restriction. For 'version' (as well as artifact 'extension' and 'classifier'), Gradle will handle any valid Unicode character.

The only Unicode values that are explicitly prohibited are '\', '/' and any ISO control character. Supplied values are validated early in publication.

Certain repositories will not be able to handle all supported characters. For example, the ':' character cannot be used as an identifier when publishing to a filesystem-backed repository on Windows.

66.2.4. Modifying the generated POM

The generated POM file may need to be tweaked before publishing. The "`maven-publish`" plugin provides a hook to allow such modification.

Example 66.5. Modifying the POM file

build.gradle

```
publications {
        mavenCustom(MavenPublication) {
            pom.withXml {
                asNode().appendNode('description',
                                    'A demonstration of maven POM customization')
            }
        }
    }
```

In this example we are adding a 'description' element for the generated POM. With this hook, you can modify any aspect of the POM. For example, you could replace the version range for a dependency with the actual version used to produce the build.

See `MavenPom.withXml()` in the API documentation for more information.

It is possible to modify virtually any aspect of the created POM should you need to. This means that it is also possible to modify the POM in such a way that it is no longer a valid Maven Pom, so care must be taken when using this feature.

The identifier (groupId, artifactId, version) of the published module is an exception; these values cannot be modified in the POM using the `withXML` hook.

66.2.5. Publishing multiple modules

Sometimes it's useful to publish multiple modules from your Gradle build, without creating a separate Gradle subproject. An example is publishing a separate API and implementation jar for your library. With Gradle this is simple:

Example 66.6. Publishing multiple modules from a single project

build.gradle

```
task apiJar(type: Jar) {
        baseName "publishing-api"
        from sourceSets.main.output
        exclude '**/impl/**'
    }

    publishing {
        publications {
            impl(MavenPublication) {
                groupId 'org.gradle.sample.impl'
                artifactId 'project2-impl'
                version '2.3'

                from components.java
            }
            api(MavenPublication) {
                groupId 'org.gradle.sample'
                artifactId 'project2-api'
                version '2'

                artifact apiJar
            }
        }
    }
```

If a project defines multiple publications then Gradle will publish each of these to the defined repositories. Each publication must be given a unique identity as described above.

66.3. Repositories

Publications are published to repositories. The repositories to publish to are defined by the `PublishingExtension.getRepositories()` container.

Example 66.7. Declaring repositories to publish to

build.gradle

```
publishing {
    repositories {
        maven {
            // change to point to your repo, e.g. http://my.org/repo
            url "$buildDir/repo"
        }
    }
}
```

The DSL used to declare repositories for publication is the same DSL that is used to declare repositories to consume dependencies from, `RepositoryHandler`. However, in the context of Maven publication only `MavenArtifactRepository` repositories can be used for publication.

66.4. Performing a publish

The "`maven-publish`" plugin automatically creates a `PublishToMavenRepository` task for each `MavenPublication` and `MavenArtifactRepository` combination in the `publishing.publ` and `publishing.repositories` containers respectively.

The created task is named "`publish«`*`PUBNAME`*`»PublicationTo«`*`REPONAME`*`»Repository`".

Example 66.8. Publishing a project to a Maven repository

build.gradle

```
apply plugin: 'java'
apply plugin: 'maven-publish'

group = 'org.gradle.sample'
version = '1.0'

publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java
        }
    }
}
publishing {
    repositories {
        maven {
            // change to point to your repo, e.g. http://my.org/repo
            url "$buildDir/repo"
        }
    }
}
```

gradle publish の出力

```
> gradle publish
:generatePomFileForMavenJavaPublication
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:publishMavenJavaPublicationToMavenRepository
:publish

BUILD SUCCESSFUL

Total time: 1 secs
```

In this example, a task named "`publishMavenJavaPublicationToMavenRepository`" is created, which is of type `PublishToMavenRepository`. This task is wired into the `publish` lifecycle task. Executing "`gradle publish`" builds the POM file and all of the artifacts to be published, and transfers them to the repository.

66.5. Publishing to Maven Local

For integration with a local Maven installation, it is sometimes useful to publish the module into the local .m2 repository. In Maven parlance, this is referred to as 'installing' the module. The "`maven`" plugin makes this easy to do by automatically creating a `PublishToMavenLocal` task for each `MavenPublication` in the `publishing.publications` container. Each of these tasks is wired into the `publishToMavenLocal` lifecycle task. You do not need to have `mavenLocal` in your `publishing.repositories` section.

The created task is named "publish«*PUBNAME*»PublicationToMavenLocal".

Example 66.9. Publish a project to the Maven local repository

gradle publishToMavenLocal の出力

```
> gradle publishToMavenLocal
:generatePomFileForMavenJavaPublication
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:publishMavenJavaPublicationToMavenLocal
:publishToMavenLocal

BUILD SUCCESSFUL

Total time: 1 secs
```

The resulting task in this example is named "publishMavenJavaPublicationToMavenLocal". This task is wired into the publishToMavenLocal lifecycle task. Executing "gradle publishT" builds the POM file and all of the artifacts to be published, and "installs" them into the local Maven repository.

66.6. Generating the POM file without publishing

At times it is useful to generate a Maven POM file for a module without actually publishing. Since POM generation is performed by a separate task, it is very easy to do so.

The task for generating the POM file is of type GenerateMavenPom, and it is given a name based on the name of the publication: "generatePomFileFor«*PUBNAME*»Publication". So in the example below, where the publication is named "mavenCustom", the task will be named "genera".

Example 66.10. Generate a POM file without publishing

build.gradle

```
model {
    tasks.generatePomFileForMavenCustomPublication {
        destination = file("$buildDir/generated-pom.xml")
    }
}
```

gradle generatePomFileForMavenCustomPublication の出力

```
> gradle generatePomFileForMavenCustomPublication
:generatePomFileForMavenCustomPublication

BUILD SUCCESSFUL

Total time: 1 secs
```

All details of the publishing model are still considered in POM generation, including `components` `, custom `artifacts`, and any modifications made via `pom.withXml`.

> The "`maven-publish`" plugin leverages some experimental support for late plugin configuration, and any `GenerateMavenPom` tasks will not be constructed until the publishing extension is configured. The simplest way to ensure that the publishing plugin is configured when you attempt to access the `GenerateMavenPom` task is to place the access inside a `model` block, as the example above demonstrates.
>
> The same applies to any attempt to access publication-specific tasks like `PublishToMavenRepository`. These tasks should be referenced from within a `model` block.

A
Gradleサンプル集

Gradleの配布物には、スタンドアローンで動作するサンプルが含まれています。以下にその一覧を掲載しま
GRADLE_HOME/samples ディレクトリに入っています。

表A.1 配布物に入っているサンプル

サンプル	説明
announce	アナウンスプラグインを使うプロジェクト
application	アプリケーションプラグインを使うプロジェクト
buildDashboard	A project which uses the build-dashboard plugin
codeQuality	A project which uses the various code quality plugins.
customBuildLanguage	ビルドDSLへカスタム要素を追加するデモです。カスタ
customDistribution	Gradleのカスタムディストリビューションを作る方法、
customPlugin	カスタムプラグインやカスタムタスクの実装、テスト、
ear/earCustomized/ear	中身をカスタマイズしたWebアプリケーションearプロジ
ear/earWithWar	Webアプリケーションearプロジェクト
groovy/customizedLayout	ソースコードのレイアウトをカスタマイズしたGroovyプ
groovy/mixedJavaAndGroovy	JavaとGroovyのソースが混ざって含まれているプロジェ
groovy/multiproject	複数のGroovyプロジェクトで構成されたビルド。特定の
groovy/quickstart	Groovyクイックスタートサンプル
java/base	基本的なJavaプロジェクト
java/customizedLayout	ソースコードのレイアウトをカスタマイズしたJavaプロ

`java/multiproject`	複数のJavaプロジェクトを使ってアプリケーションを組
`java/quickstart`	Javaクイックスタートプロジェクト
`java/withIntegrationTests`	ソースセットを使って、Javaプロジェクトに結合テスト
`javaGradlePlugin`	This example demonstrates the use of the java gradle the plugin metadata during jar execution.
`maven/pomGeneration`	Mavenリポジトリにデプロイ、インストールするデモ。
`maven/quickstart`	Mavenリポジトリにアーティファクトをデプロイ、イン
`osgi`	OSGiバンドルをビルドするプロジェクト
`scala/customizedLayout`	ソースコードのレイアウトをカスタマイズしたScalaプロ
`scala/fsc`	Fast Scala Compiler (fsc)を使ったScalaプロジェクト
`scala/mixedJavaAndScala`	ScalaとJavaのソースが混ざって含まれているプロジェク
`scala/quickstart`	Scalaクイックスタートプロジェクト
`scala/zinc`	Scala project using the Zinc based Scala compiler.
`testing/testReport`	Generates an HTML test report that includes the test r
`toolingApi/customModel`	A sample of how a plugin can expose its own custom t
`toolingApi/eclipse`	Tooling APIを使ってEclipseのプロジェクトモデルを構築
`toolingApi/idea`	IntelliJ IDEAで必要な情報を、Tooling APIを使って抽出す
`toolingApi/model`	An application that uses the tooling API to build the m
`toolingApi/runBuild`	An application that uses the tooling API to run a Gradl
`userguide/distribution`	A project which uses the distribution plugin
`userguide/javaLibraryDistribution`	A project which uses the Java library distribution plugi
`webApplication/customized`	中身をカスタマイズしたWARをビルドするWebアプリケ

webApplication/quickstart	Webアプリケーション クイックスタートプロジェクト

A.1. サンプル customBuildLanguage

ビルドDSLへカスタム要素を追加するデモです。カスタムプラグインを使ってビルドロジックを体系化する

このビルドは、二種類のプロジェクトから構成されています。「製品」プロジェクトと「製品モジュール」
「製品」は複数の「製品モジュール」から構成されており、ひとつの「製品モジュール」は複数のプロジ
つまり、「製品モジュール」とプロジェクトが、多対多の関係になっているのです。
各「製品」は、「製品モジュール」の実行時クラスパスを固めたZIPファイルを作成します。さらに、ZIP

「製品」プロジェクトのビルドスクリプト（basicEdition/build.gradle
など）では、独自のカスタム要素が使用されています。ビルドスクリプトで、product { }
という要素が使われていることに注目してください。これがカスタム要素です。

プロジェクトのビルドスクリプトには、要素の宣言しか含まれていません。処理の大半はbuildSrc/src/
にある二つのカスタムプラグインで記述されています。

A.2. サンプル customDistribution

Gradleのカスタムディストリビューションを作る方法、およびそのディストリビューションをGradleラッ

このサンプルには次のプロジェクトが含まれています。

- plugin
  ディレクトリに入っているプロジェクトで、カスタムプラグインを実装してディストリビューションに
- consumer
  ディレクトリに入っているプロジェクトで、そのカスタムディストリビューションを使用しています。

A.3. サンプル customPlugin

カスタムプラグインやカスタムタスクの実装、テスト、公開および使用方法を示す、いくつかのサンプル

このサンプルには、以下のプロジェクトが含まれています。

- pluginディレクトリのプロジェクトで、プラグインを実装して公開しています。
- consumerディレクトリのプロジェクトで、そのプラグインを使用しています。

A.4. サンプル java/multiproject

複数のJavaプロジェクトを使ってアプリケーションを組み立てるデモ

このビルドはクライアント・サーバー型のアプリケーションを作成するもので、配布物になるのは二つの
サードパーティ製のアプリケーションをコンパイルするためのAPIを含むJARとクライアントのランタイム
そして、Webサービスを提供するサーバーWARファイルを作成します。

B
陥りがちな罠

B.1. Groovyスクリプトの変数

Gradleユーザーにとって、Groovyがスクリプトの変数をどう扱うか理解しておくことは大事なことです。Groovyには、二種類のスクリプト変数があります。一つがローカルスコープのもので、もう一つがスクリ

例B.1 変数のスコープ：ローカルスコープとスクリプトスコープ

scope.groovy

```
String localScope1 = 'localScope1'
def localScope2 = 'localScope2'
scriptScope = 'scriptScope'

println localScope1
println localScope2
println scriptScope

closure = {
    println localScope1
    println localScope2
    println scriptScope
}

def method() {
    try {
        localScope1
    } catch (MissingPropertyException e) {
        println 'localScope1NotAvailable'
    }
    try {
        localScope2
    } catch(MissingPropertyException e) {
        println 'localScope2NotAvailable'
    }
    println scriptScope
}

closure.call()
method()
```

gradle の出力

```
> gradle
localScope1
localScope2
scriptScope
localScope1
localScope2
scriptScope
localScope1NotAvailable
localScope2NotAvailable
scriptScope
```

型修飾のついた変数は、クロージャ内からは見えていますが、メソッドからは見えていません。これは、(
[44]

B.2. 設定フェーズと実行フェーズ

Gradleが設定フェーズと実行フェーズを区別していることは常に頭に入れておきましょう。とても重要な
56章ビルドのライフサイクル参照)

例B.2 設定フェーズと実行フェーズの区別

build.gradle

```
def classesDir = file('build/classes')
classesDir.mkdirs()
task clean(type: Delete) {
    delete 'build'
}
task compile(dependsOn: 'clean') << {
    if (!classesDir.isDirectory()) {
        println 'The class directory does not exist. I can not operate'
        // do something
    }
    // do something
}
```

gradle -q compile の出力

```
> gradle -q compile
The class directory does not exist. I can not operate
```

ディレクトリの作成は設定フェーズで起こっていて、実行フェーズで`clean`タスクがそのディレクトリを削除してしまっています。

---

[44] これらの議論のうちの一つを、ここで見ることができます。 http://groovy.329449.n5.nabble.com/script-scoping-question-td355887.html

C
機能のライフサイクル

Gradleプロジェクトは現在も活発に開発、改善されており、新しいバージョンが定期的に、また頻繁にリ
新しい機能および修正を頻繁にリリースすることで、それらの新機能が素早くユーザーにより確認され、
それと同時に、APIや機能の安定性もとても重要であり、これもGradleプラットフォームの大事な価値の一
新機能は、設計上の選択により開発プロセスに組み入れられ、自動テストされ、「後方互換性ポリシー
」に従って正式採用されるようになっています。

Gradleの機能ライフサイクルは、これらの目的を達成するために設計されました。
また、機能ライフサイクルは、ある機能が今どんなステータスにあるのか、ユーザーに明確に伝えること
機能
という用語は、たいていの場合APIやDSLのメソッドやプロパティを指しますが、それだけにとどまりませ
コマンドライン引数や実行モード(例えばBuild Deamon)などもその一つです。

C.1. 状態

機能の状態は、以下の四つのうちいずれかとなります。

- 非公開
- 試験的
- 公開
- 非推奨

C.1.1. 非公開

非公開の機能は、広く使われることは想定しておらず、Gradleが内部的に使うためのものです。これらの
従って、これらの機能を使うことは推奨しません。
また、非公開機能はドキュメント化もされません。もしユーザーガイドやDSLリファレンス、APIリファレ

非公開機能は、公開機能に昇格する可能性があります。

C.1.2. 試験的

試験的な
状態にあると書かれた機能は、正式に公開される前に、実際の運用で発生したフィードバックを受け付け
これらの機能は、機能が変更されるリスクを受け入れられるユーザーに向けて公開されているもので、そ

試験的な機能は、試験的でなくなるまでは将来のバージョンのGradleで変更されるかもしれません。変更
新機能の試験期間は、その範囲や複雑性、機能の性質によって様々です。

試験的な機能は、試験的であるとはっきり明示されます。ソースコードでは全ての試験的なメソッド、プ

I n c u b a t i n g
アノテーションで注釈されます。これはDSLリファレンスやAPIリファレンスにマークする際も使用されま

C.1.3. 公開

非公開でない機能のデフォルトのステータスが公開です。
ユーザーガイド、DSLリファレンス、APIリファレンスにおける全ての記載内容は、明示的に試験的とか非
機能は、試験的な状態から公開機能へと言わば昇格
します。試験的だった機能が公開状態に昇格した場合は、そのバージョンのリリースノートで通知します。

公開状態の機能は、非推奨とならない限り決して
削除されたり意図的に変更されたりすることはありません。
全ての公開機能は上位互換性ポリシーの対象となります。

C.1.4. 非推奨

Gradleの発展に伴って、いくつかの機能は別のものに取って代わられたり、不適切になったりすることが
そのような機能は、非推奨とされた後、やがてはGradleから削除されます。
非推奨の機能は、上位互換性ポリシーに従い、削除されるまで決して変更されることはありません。

非推奨の機能は、非推奨であるとはっきり明示されます。ソースコードでは全ての非推奨メソッド、プロ
@java.lang.Deprecated
アノテーションで注釈されます。これはDSLリファレンスやAPIリファレンスにも反映されます。ほとんど

非推奨機能の使用は避けるべきです。リリースノートには、そのリリースで非推奨となった機能が全て記

C.2. 後方互換性ポリシー

Gradleはメジャーバージョン間(1.x、2.xなど)で上位互換性を保持します。
一度Gradleのリリースに導入されたり昇格したりした公開機能は、非推奨にならない限り決して変更され
機能がいったん非推奨になると、次からのメジャーリリースでは削除される可能性があります。
非推奨機能がメジャーバージョンをまたいでサポートされる場合もありますが、必ずそうとは限りません。

D
Gradle コマンドライン

The gradle コマンドには次の使用方法があります:

`gradle` [option...] [task...]

コマンドライン・オプションは下記のgradleコマンドリストに示されている物が利用可能です。

-?, -h, --help
ヘルプのメッセージを参照

-a, --no-rebuild
プロジェクトの依存関係をリビルドしません。

--all
タスク実行時の応答の詳細状況を表示をします。「タスクの一覧」を参照。

-b, --build-file
ビルド対象のgradleファイルを指定します。「ビルドスクリプトを指定して実行する」参照。

-c, --settings-file
設定ファイルを指定します。

--continue
タスクを失敗しても、実行処理を継続します。

--configure-on-demand (incubating)
Only relevant projects are configured in this build run. This means faster builds for large multi-projects. See 「Configuration on demand」.

-D, --system-prop
JVMの環境変数を設定します, 例えば`-Dmyprop=myvalue`。「Gradleプロパティとシステムプロパティ」を参照。

-d, --debug
デバックモードのログ出力(スタックトレースを含みます)。18章ロギング参照。

-g, --gradle-user-home
Gradleのユーザディレクトリを指定します。デフォルトはユーザディレクトリ配下の `.gradle` になります。

--gui
Gradle GUIを起動します。12章GradleのGUIを使う参照。

-I, --init-script
初期化スクリプトを指定します。61章初期化スクリプト参照。

-i, --info
ログレベルをinfoに指定します。18章ロギング参照。

-m, --dry-run
タスクを実行しない形でビルドを走らせます。「空実行」参照。

--no-color
コンソールの出力をカラフルにしない指定。

--offline
ネットワークにアクセスしない形でビルドを実行する指定です。
「キャッシングに関するコマンドラインオプション 」参照。

-P, --project-prop
ルートプロジェクトに、プロジェクトプロパティを設定します。例えば `-Pmyprop=myvalue`。
「Gradleプロパティとシステムプロパティ」参照。

-p, --project-dir
Gradleの実行ディレクトリを指定します。デフォルトはカレントディレクトリです。
「ビルドスクリプトを指定して実行する」参照。

--parallel (incubating)
Build projects in parallel. Gradle will attempt to determine the optimal number of executor threads to use. This option should only be used with decoupled projects (see 「分離されたプロジェクト 」).

--parallel-threads (incubating)
Build projects in parallel, using the specified number of executor threads. For example`--paralle`
. This option should only be used with decoupled projects (see 「分離されたプロジェクト 」).

--profile
Profilesコマンドはビルド実行時に*buildDir*`/reports/profile`
ディレクトリにタスクの実行時間をレポート出力します。 「ビルドのプロファイリング」参照。

--project-cache-dir
プロジェクトのキャッシュディレクトリを指定します。 デフォルトは`.gradle`
ディレクトリ上のルートプロジェクトディレクトリになります。 「キャッシング 」参照。

-q, --quiet
ログ出力をエラーのみにします。18章ロギング参照。

--recompile-scripts
ビルドキャッシュファイルを無視してリコンパイルを強制する指定です。 「キャッシング 」参照。

--refresh-dependencies
依存状態をリフレッシュします。 「キャッシングに関するコマンドラインオプション 」参照。

--rerun-tasks
タスク最適化を無視する指定です。

-S, **--full-stacktrace**
例外発生時にフルスタックトレースを出力します。18章ロギング参照。

-s, **--stacktrace**
ユーザ例外のスタックトレースを出力します。(例えばコンパイルエラー) 18章ロギング参照。

-u, **--no-search-upwards**
親ディレクトリの `settings.gradle` ファイルを探しません。

-v, **--version**
バージョン情報を表示します。

-x, **--exclude-task**
除外タスクを指定します。「タスクを除外してビルドする」参照。

gradle -hコマンドを実行するとき上記の情報は表示されます。

D.1. デーモン コマンドラインオプション:

19章Gradleデーモン のリンクはデーモンに関しての情報を多く含みます。 例えば、デフォルトで`--daem`
指定をつける方法等

--daemon
Gradle デーモンの状態でビルドを実行します。 既存のデーモンが未実行 または 忙しければ 19章
Gradleデーモン 新しいデーモンプロセスを実行します。

--foreground
G r a d l e
daemonをフォアグラウンドの状態で実行します。容易にビルド時の例外をモニターできるので、
デバックやトラブルシューティングに役立ちます。

--no-daemon
対象のビルドを実行させる為にGradleデーモンを使いません。
デーモンモードで実行することをデフォルトにしていれば、時々有用です。

--stop
Gradleデーモンが実行されていれば停止します。 実行しているバージョンのGradleの`--stop`
で停止させてください。

D.2. システムプロパティ

システムプロパティの設定には以下のgradleコマンドが利用可能です。
ただしコマンドラインオプションのシステムプロパティの指定が優先されることに注意

`gradle.user.home`
: Gradleユーザディレクトリの指定

「gradle.propertiesを使用したビルド環境の構築」
にGradleの設定で利用可能なシステムプロパティの情報が記載されています。

D.3. 環境変数

環境変数の設定には以下のgradleコマンドが利用可能です。
ただしコマンドラインオプションの環境変数の指定が優先されることに注意

GRADLE_OPTS
: JVMを始めるためにに使用するコマンドライン引数を指定します。
Gradleの実行のためにに使用させるのに役立つことができます。例えば `GRADLE_OPTS="-Dorg.gradl`
の指定は `--daemon`をコマンドラインで指定する事無く実行させることが出来ます。
「gradle.propertiesを使用したビルド環境の構築」に
環境変数を使う事無しに、デーモンモードでの設定をする情報が記載されています。
例えばより維持可能でより明示的な方法で。

GRADLE_USER_HOME
: Gradleのユーザディレクトリを指定します

JAVA_HOME
: Specifies the JDK installation directory to use.

E IDE対応の現状と、IDEによらない開発支援

E.1. IntelliJ

Gradleは主にIdea IntelliJとそのすばらしいGroovyプラグインを利用して開発されています。
Gradleのビルドスクリプト[46]の開発にも、このIDEのサポートを利用することができます。
IntelliJは任意のファイルパターンをGroovyスクリプトとして解釈することができます。
Gradleの場合は`build.gradle`や`settings.gradle`のようなパターンを定義できます。
これは非常に役に立ちます。
Gradleクラスに対するコンテンツアシストを提示するためのGradleバイナリに対するクラスパスの設定は
プロジェクトのクラスパスに、GradleのJAR(配布物に含まれています)を追加するとよいでしょう。
実際にはそこにあるべきものではないのですが、これを行うことでGradleスクリプトの開発に対するIDEの
もちろん、ビルドスクリプトのための追加のライブラリを使うならば、それはプロジェクトのクラスパスを

われわれは、将来においてIntelliJが`*.gradle`ファイルを特別に扱い、
それらに特化したクラスパスの定義が可能になることを希望しています。

E.2. Eclipse

EclipseにはGroovyプラグインがあります。
我々はその現状や、Gradleをどうサポートできるのかを把握していません。
ユーザーガイドの次の版ではもう少し詳しい記述を追加したいと思っています。

E.3. IDEサポートなしでGradleを使う

我々にできるのは、`throw new org.gradle.api.tasks.StopExecutionException()`のような記述のタイプ量を節約して 代わりに`throw new StopExecutionException()`とだけタイプすればよいようにすることです。
Gradleスクリプトの実行前に一連のimport文を自動的に追加することでこれを実現しています。
以下はスクリプトに追加されるimportの一覧です。

図E.1 Gradleがimportするパッケージ

```
import org.gradle.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
import org.gradle.api.artifacts.cache.*
import org.gradle.api.artifacts.component.*
import org.gradle.api.artifacts.dsl.*
import org.gradle.api.artifacts.ivy.*
import org.gradle.api.artifacts.maven.*
```

```
import org.gradle.api.artifacts.query.*
import org.gradle.api.artifacts.repositories.*
import org.gradle.api.artifacts.result.*
import org.gradle.api.component.*
import org.gradle.api.distribution.*
import org.gradle.api.distribution.plugins.*
import org.gradle.api.dsl.*
import org.gradle.api.execution.*
import org.gradle.api.file.*
import org.gradle.api.initialization.*
import org.gradle.api.initialization.dsl.*
import org.gradle.api.invocation.*
import org.gradle.api.java.archives.*
import org.gradle.api.jvm.*
import org.gradle.api.logging.*
import org.gradle.api.platform.jvm.*
import org.gradle.api.plugins.*
import org.gradle.api.plugins.announce.*
import org.gradle.api.plugins.antlr.*
import org.gradle.api.plugins.buildcomparison.gradle.*
import org.gradle.api.plugins.jetty.*
import org.gradle.api.plugins.osgi.*
import org.gradle.api.plugins.quality.*
import org.gradle.api.plugins.scala.*
import org.gradle.api.plugins.sonar.*
import org.gradle.api.plugins.sonar.model.*
import org.gradle.api.publish.*
import org.gradle.api.publish.ivy.*
import org.gradle.api.publish.ivy.plugins.*
import org.gradle.api.publish.ivy.tasks.*
import org.gradle.api.publish.maven.*
import org.gradle.api.publish.maven.plugins.*
import org.gradle.api.publish.maven.tasks.*
import org.gradle.api.publish.plugins.*
import org.gradle.api.reporting.*
import org.gradle.api.reporting.components.*
import org.gradle.api.reporting.dependencies.*
import org.gradle.api.reporting.plugins.*
import org.gradle.api.resources.*
import org.gradle.api.specs.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.ant.*
import org.gradle.api.tasks.application.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.diagnostics.*
import org.gradle.api.tasks.incremental.*
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.scala.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.testing.junit.*
import org.gradle.api.tasks.testing.testng.*
import org.gradle.api.tasks.util.*
import org.gradle.api.tasks.wrapper.*
import org.gradle.buildinit.plugins.*
import org.gradle.buildinit.tasks.*
import org.gradle.external.javadoc.*
import org.gradle.ide.cdt.*
import org.gradle.ide.cdt.tasks.*
import org.gradle.ide.visualstudio.*
import org.gradle.ide.visualstudio.plugins.*
```

```
import org.gradle.ide.visualstudio.tasks.*
import org.gradle.jvm.*
import org.gradle.jvm.plugins.*
import org.gradle.jvm.toolchain.*
import org.gradle.language.*
import org.gradle.language.assembler.*
import org.gradle.language.assembler.plugins.*
import org.gradle.language.assembler.tasks.*
import org.gradle.language.base.*
import org.gradle.language.base.artifact.*
import org.gradle.language.base.plugins.*
import org.gradle.language.c.*
import org.gradle.language.c.plugins.*
import org.gradle.language.c.tasks.*
import org.gradle.language.cpp.*
import org.gradle.language.cpp.plugins.*
import org.gradle.language.cpp.tasks.*
import org.gradle.language.java.*
import org.gradle.language.java.artifact.*
import org.gradle.language.java.plugins.*
import org.gradle.language.jvm.*
import org.gradle.language.jvm.plugins.*
import org.gradle.language.jvm.tasks.*
import org.gradle.language.nativebase.tasks.*
import org.gradle.language.objectivec.*
import org.gradle.language.objectivec.plugins.*
import org.gradle.language.objectivec.tasks.*
import org.gradle.language.objectivecpp.*
import org.gradle.language.objectivecpp.plugins.*
import org.gradle.language.objectivecpp.tasks.*
import org.gradle.language.rc.*
import org.gradle.language.rc.plugins.*
import org.gradle.language.rc.tasks.*
import org.gradle.nativeplatform.*
import org.gradle.nativeplatform.platform.*
import org.gradle.nativeplatform.plugins.*
import org.gradle.nativeplatform.sourceset.*
import org.gradle.nativeplatform.tasks.*
import org.gradle.nativeplatform.test.*
import org.gradle.nativeplatform.test.cunit.*
import org.gradle.nativeplatform.test.cunit.plugins.*
import org.gradle.nativeplatform.test.cunit.tasks.*
import org.gradle.nativeplatform.test.plugins.*
import org.gradle.nativeplatform.test.tasks.*
import org.gradle.nativeplatform.toolchain.*
import org.gradle.nativeplatform.toolchain.plugins.*
import org.gradle.platform.base.*
import org.gradle.platform.base.binary.*
import org.gradle.platform.base.component.*
import org.gradle.platform.base.test.*
import org.gradle.plugin.use.*
import org.gradle.plugins.ear.*
import org.gradle.plugins.ear.descriptor.*
import org.gradle.plugins.ide.api.*
import org.gradle.plugins.ide.eclipse.*
import org.gradle.plugins.ide.idea.*
import org.gradle.plugins.javascript.base.*
import org.gradle.plugins.javascript.coffeescript.*
import org.gradle.plugins.javascript.envjs.*
import org.gradle.plugins.javascript.envjs.browser.*
import org.gradle.plugins.javascript.envjs.http.*
```

```
import org.gradle.plugins.javascript.envjs.http.simple.*
import org.gradle.plugins.javascript.jshint.*
import org.gradle.plugins.javascript.rhino.*
import org.gradle.plugins.javascript.rhino.worker.*
import org.gradle.plugins.signing.*
import org.gradle.plugins.signing.signatory.*
import org.gradle.plugins.signing.signatory.pgp.*
import org.gradle.plugins.signing.type.*
import org.gradle.plugins.signing.type.pgp.*
import org.gradle.process.*
import org.gradle.sonar.runner.*
import org.gradle.sonar.runner.plugins.*
import org.gradle.sonar.runner.tasks.*
```

```
import org.gradle.testing.jacoco.plugins.*
import org.gradle.testing.jacoco.tasks.*
import org.gradle.util.*
```

---

[46] GradleはGradleでビルドされている

Gradle User Guide

A

アーティファクト / Artifact
    ??

B

ビルドスクリプト / Build Script
    ??

C

コンフィグレーション / Configuration
    See 依存関係の構成 / Dependency Configuration.

コンフィグレーション注入 / Configuration Injection
    ??

D

DAG
    See 無閉路有向グラフ / Directed Acyclic Graph.

依存関係 / Dependency
    See 外部依存関係 / External Dependency.

    See プロジェクト依存関係 / Project Dependency.

    ??

依存関係の構成 / Dependency Configuration
    ??

依存関係の解決 / Dependency Resolution
    ??

無閉路有向グラフ / Directed Acyclic Graph
    無閉路有向グラフとは循環を持たない有向グラフである。
    Gradleでは実行対象となる各タスクはグラフ内のノードとして表現される。
    他のタスクに対するdependsOnによる関連は、(既にグラフに存在していなければ)このタスクをノード
    それら2つのノード間に方向性を伴った辺を作成する。
    あらゆるdependsOn関連は循環を持たないことが検証される。
    あるノードから開始して、辺を順にたどり、元のノードにたどり着く道があってはならない。

ドメイン特化言語 / Domain Specific Language

ドメイン特化言語とは、特定の問題領域や特定の問題表現技法と解決技法に特化したプログラム言語ないしは言語に対する仕様である。　　　　　その概念は新しいものではなく、特定用途のプログラム言語や、あらゆる種類のモデリング・仕様記述言語は常に存在する。しかし、ドメイン特化モデリングの興隆によって、この用語は有名になりつつある。

DSL

See ドメイン特化言語 / Domain Specific Language.

E

外部依存関係 / External Dependency

??

拡張オブジェクト / Extension Object

??

I

初期スクリプト / Init Script

ビルド自体が開始する前に実行されるスクリプト。Gradleおよびビルドそのものをカスタマイズするこ

初期化スクリプト / Initialization Script

See 初期スクリプト / Init Script.

P

プラグイン / Plugin

??

プロジェクト / Project

??

プロジェクト依存関係 / Project Dependency

??

発行 / Publication

??

R

リポジトリ / Repository

??

S

ソースセット / Source Set

??

T

タスク / Task
    ??

推移的依存関係 / Transitive Dependency
    ??