Paincker

```
}
// 拦截所有方法调用,先输出一个log,再调用原始方法
var2.metaClass.invokeMethod = { String name, Object[] args ->
    println "invoke method '$name()'"
    def originMethod = var2.metaClass.getMetaMethod(name, args)
    if (originMethod != null) originMethod.invoke(var2, args)
}
var2.func()
```

执行结果：

```
invoke method 'func()'
hello
```

理解Gradle DSL语法——用Groovy实现自己的DSL

为了较好的理解Gradle DSL语法，本节先给出一段常见的gradle脚本，然后一边对其执行过程进行分析，一边用Groovy自己定义DSL，实现类似的语法效果。

** 注意这里只是简化版的示例，和Gradle的实际实现并不完全相同。**

完整示例工程如下。运行其中的 `DemoDSL.groovy` 即可看到执行结果（如果运行时提示找不到gradle文件，需要在IDEA/Android Studio的RunConfiguration中修改Working directory）。

> https://github.com/jzj1993/GradleStudy

`build.gradle` 脚本

在 `build.gradle` 脚本中，可以给工程添加 `compile` 和 `testCompile` 两个 `Configuration`，然后分别添加若干 `Dependency` (包括远程模块和本地Project等类型依赖)，如下。

```
// 给Project添加两个Configuration
configurations {
    compile
    testCompile
}

// 在不同的Configuration中添加依赖项
dependencies {
    add('compile', 'com.demo:module0')
    compile 'com.demo:module1' // 外部Module依赖
    testCompile project(path: ":library") // Project依赖
    compile('com.demo:module2') { // 支持Closure配置
        transitive = false
    }
    debugCompile 'com.demo:module3' // 没有这个Configuration，会报错
}

// 其他变式写法
dependencies.compile 'com.demo:module4'
project.dependencies.compile 'com.demo:module5'
project.dependencies {
    compile 'com.demo:module6'
}
```

`build.gradle` 脚本的执行

Gradle会在一个Project对象上执行 `build.gradle` 脚本（`Run build.gradle against a Project object`），可以理解成 `build.gradle` 的代理对象是Project。

示例代码中：

1. 创建Project对象。
2. 通过Groovy加装 `build.gradle` 文件，并将其视为Groovy脚本编译生成Script类，创建出一个GroovyObject实例。
3. 利用元编程，设置GroovyObject的代理对象为创建好的Project对象。
4. 执行GroovyObject的 `run()` 方法，即执行 `build.gradle` 脚本。

Paincker


```
        Class groovyClass = new GroovyClassLoader().parseClass(sourceFile);
        return (GroovyObject) groovyClass.newInstance();
    }

    static void setDelegateForGroovyObject(GroovyObject obj, Object delegate) {
        obj.metaClass.getProperty = { String name ->
            def metaProperty = obj.metaClass.getMetaProperty(name)
            metaProperty != null ? metaProperty : delegate.getProperty(name)
        }
        obj.metaClass.invokeMethod = { String name, Object[] args ->
            def metaMethod = obj.metaClass.getMetaMethod(name, args)
            metaMethod != null ? metaMethod.invoke(obj, args) : delegate.invokeMethod(name, args)
        }
    }
}

class Project {
    // ...
}

// 创建Project对象
def project = new Project()
// 加载并实例化Groovy对象
def groovyObject = Utils.loadAndCreateGroovyObject(new File('./build.gradle'))
// 给groovyObject设置代理对象
Utils.setDelegateForGroovyObject(groovyObject, project)
// 执行脚本(Run "build.gradle" against the Project object)
groovyObject.invokeMethod("run", null)
```

Project, configuraitons, dependencies

- `configurations {}` 和 `dependencies {}` 语句其实都是调用Project定义的方法，后面的大括号则是方法的闭包参数。
- `denpendencies.compile` 这种写法，这里的 `dependencies` 则是在调用Project定义的属性。
- `project.xxx`，这里的 `project` 也是Project定义的属性，指向其自身。

```
class Utils {
    /**
     * 指定代理对象,运行闭包
     */
    static void runClosureAgainstObject(Closure closure, Object delegate) {
        Closure c = (Closure) closure.clone()
        c.delegate = delegate
        c.call()
    }
}
class Project {

    ConfigurationContainer configurations = new ConfigurationContainer()
    DependencyHandler dependencies = new DependencyHandler(this)
    Project project = this

    void configurations(Closure closure) {
        Utils.runClosureAgainstObject(closure, configurations)
    }

    void dependencies(Closure closure) {
        Utils.runClosureAgainstObject(closure, dependencies)
    }
}
```

ConfigrationContainer

`configurations(Closure c)` 方法在 `ConfigurationContainer` 对象上执行闭包参数，compile和testCompile都是在调用 `ConfigurationContainer` 的 `propertyMissing()` 。

```
class ConfigurationContainer {
    Map<String, Configuration> configurations = new HashMap<>()

    Object propertyMissing(String name) {
        println "add configuration '$name'"
        configurations.put(name, new Configuration())
    }
}
```

Paincker

`dependencies(Closure c)` 方法在 `DependencyHandler` 上执行闭包参数。

- 闭包中的 `compile xxx` 、 `testCompile xxx` 等都是在调用 `DependencyHandler` 的 `methodMissing()` ，最后被转到调用 `add()` 方法，从而添加依赖。
- 闭包中的 `project(path: 'xxx')` 也是 `DependencyHandler` 定义的一个方法，参数为Map，返回一个 `Dependency` 对象。
- 调用 `compile xxx {}` 时，最后可以传入一个闭包参数，用于配置 `transitive` 属性等操作。当 `DependencyHandler.add` 方法传入了closure，会执行 `Utils.configureObjectWithClosure(dependency, closure)` ，用闭包配置 `Dependency` ，闭包中的 `transitive=false` 会覆盖 `Dependency` 中的对应属性。

```
class Utils {
    static void configureObjectWithClosure(Object object, Closure closure) {
        Closure c = (Closure) closure.clone()
        c.resolveStrategy = Closure.DELEGATE_FIRST;
        c.delegate = object
        c.call()
    }
}
class DependencyHandler {

    Project project;

    DependencyHandler(Project project) {
        this.project = project;
    }

    void add(String configuration, String dependencyNotation) {
        add(configuration, new Dependency(dependencyNotation), null)
    }

    void add(String configuration, String dependencyNotation, Closure closure) {
        add(configuration, new Dependency(dependencyNotation), closure)
    }

    void add(String configuration, Dependency dependency) {
        add(configuration, dependency, null)
    }

    void add(String configuration, Dependency dependency, Closure closure) {
        Configuration cfg = this.project.configurations.configurations.get(configuration)
        if (cfg != null) {
            if (closure != null) {
                Utils.configureObjectWithClosure(dependency, closure)
            }
            cfg.dependencies.add(dependency)
            println "add dependency '${dependency}' to '${configuration}'"
        } else {
            println "configuration '${configuration}' not found, dependency is '${dependency}'"
        }
    }

    Dependency project(Map<String, ?> notation) {
        return new Dependency("project(${notation.get("path")})")
    }

    Object methodMissing(String name, Object args) {
        Object[] arr = (Object[]) args;
        if (arr.length >= 1 && (arr[0] instanceof String || arr[0] instanceof Dependency)
                && this.project.configurations.configurations.get(name) != null) {
            Dependency dependency = arr[0] instanceof String ? new Dependency((String) arr[0]) : (Dependenc
            if (arr.length == 1) {
                add(name, dependency)
            } else if (arr.length == 2 && arr[1] instanceof Closure) {
                add(name, dependency, (Closure) arr[1])
            }
        } else {
            println "method '${name}' with args '${args}' not found!"
        }
        return null
    }
}
```

Gradle构建过程

目录

> https://docs.gradle.org/3.3/userguide/build_lifecycle.html

Paincker

Gradle构建过程通常分为三步。

> A Gradle build has three distinct phases.

1、初始化阶段 (Initialization)

Gradle支持单个和多个工程的编译。在初始化阶段，Gradle判断需要参与编译的工程，为每个工程创建一个Project对象。

在这个阶段，Gradle会创建Settings对象，并在其上执行 `settings.gradle` 脚本，建立工程之间的层次关系。

> Gradle supports single and multi-project builds. During the initialization phase, Gradle determines which projects are going to take part in the build, and creates a Project instance for each of these projects.

2、配置阶段 (Configuration)

在这个阶段，Gradle会分别在每个Project对象上执行对应的 `build.gradle` 脚本，对Project进行配置。

> During this phase the project objects are configured. The build scripts of all projects which are part of the build are executed. Gradle 1.4 introduced an incubating opt-in feature called configuration on demand. In this mode, Gradle configures only relevant projects (see the section called “Configuration on demand”).

3、执行阶段 (Execution)

在执行阶段，Gradle会判断配置阶段创建的哪些Task需要被执行，然后执行选中的每个Task。

> Gradle determines the subset of the tasks, created and configured during the configuration phase, to be executed. The subset is determined by the task name arguments passed to the gradle command and the current directory. Gradle then executes each of the selected tasks.

Gradle源码查看

Gradle是开源的，学习Gradle最好的资料就是Gradle官方文档和Gradle源码。这里介绍查看Gradle源码比较好的方法。

查看gradle脚本调用的API

在Android Studio/IDEA的Gradle工程中，可以光标选中gradle脚本语句，用 `Navigate - Declaration` 菜单或快捷键，跳转到其调用的Gradle API方法。

例如选中 `build.gradle` 中的 `dependencies` ，可跳转到 `Project.dependencies(Closure)` 方法。

这个操作需要IDE支持，有时不一定管用；另外对于动态添加的方法，也不能正常跳转。

查看完整的源码

- Gradle将整个源码分为若干个模块分别发布到了Maven仓库，可参考 https://maven-repository.com/artifact/org.gradle
- 官网 https://gradle.org/releases/ 可下载Gradle的jar包，分两种（ `xx` 表示版本号）：
  - `gradle-xx-bin.zip` ，包含所有class的jar包。
  - `gradle-xx-all.zip` ，包含所有class和源码等内容。

为方便阅读，可用Android Studio或IDEA关联源码（推荐IDEA Ultimate版），具体操作如下。

1. 克隆并用IDEA打开下面的工程。打开时Gradle选 `Use default gradle wrapper` ，同步工程。项目配置了依赖本地gradle库 `compile gradleApi()` ，同步完成后一般会下载并关联gradle的jar包。

   > https://github.com/jzj1993/GradleStudy

   > 如果打开工程没有正确选择Gradle，可以在 `Preferences` - `Build Execution Deployment` - `Build Tools` - `Gradle` 中设置。

Paincker

- 如果IDEA已经关联了 `gradle-xx-all.zip` ，此时就能看到源码。
- 如果关联的是 `gradle-xx-bin.zip` ，此时只能看到class反编译的结果，点击提示栏的 `Choose Source` ，选择 `gradle-3.3-all.zip` 关联源码即可（Mac系统中默认存放在 `~/.gradle/wrapper/dists/` ，如果没有可以自行从官网下载）。

常用API

本节介绍一些Gradle开发最常用的API，并通过实例介绍其使用。

官方DSL Reference

> https://docs.gradle.org/current/dsl/index.html

org.gradle.api.Project

`Project` 对象是Gradle中最核心的API，通过 `Project` 对象可以访问所有Gradle特性。

> This interface is the main API you use to interact with Gradle from your build file. From a Project, you have programmatic access to all of Gradle's features.

Project与build.gradle

Project对象和 `build.gradle` 文件一一对应。在Gradle构建时，会先创建Settings实例并在其上执行 `settings.gradle` ；再通过Settings对象定义的Project层级，创建若干个Project实例，并分别在其上执行对应的 `build.gradle` 。

> **Lifecycle**
> There is a one-to-one relationship between a Project and a build.gradle file. During build initialisation, Gradle assembles a Project object for each project which is to participate in the build, as follows:
>
> - Create a org.gradle.api.initialization.Settings instance for the build.
> - Evaluate the settings.gradle script, if present, against the org.gradle.api.initialization.Settings object to configure it.
> - Use the configured org.gradle.api.initialization.Settings object to create the hierarchy of Project instances.
> - Finally, evaluate each Project by executing its build.gradle file, if present, against the project. The projects are evaluated in breadth-wise order, such that a project is evaluated before its child projects. This order can be overridden by calling evaluationDependsOnChildren() or by adding an explicit evaluation dependency using evaluationDependsOn(String).

Extra属性

Project有一个Extra属性，可通过ext前缀在其中定义属性，定义好后可以不加ext前缀直接访问。

> All extra properties must be defined through the "ext" namespace. Once an extra property has been defined, it is available directly on the owning object (in the below case the Project, Task, and sub-projects respectively) and can be read and updated. Only the initial declaration that needs to be done via the namespace.

示例代码如下。

```
project.ext.prop1 = "foo"
task doStuff {
    ext.prop2 = "bar"
}

ext.isSnapshot = version.endsWith("-SNAPSHOT")
if (isSnapshot) {
    // do snapshot stuff
}
```

Project的属性/方法调用

在 `build.gradle` 中调用属性，或调用 `Project.property(java.lang.String)` 方法时，会按顺序从以下范围查找：

1. Project自身定义的属性
2. Project的Extra属性

Paincker

5. Project中Task的名字
6. 从父Project继承的属性，一直递归到RootProject

在 `build.gradle` 中调用方法时，会按顺序从以下范围查找：

1. Project自身定义的方法
2. build.gradle脚本定义的方法
3. 插件添加类型为Action或Closure的Extension
4. 插件添加的Convension方法
5. Project中Task的名字都会创建一个对应方法
6. 从父Project继承的方法，一直递归到RootProject
7. Project中为Closure类型的属性可以作为方法调用

常用API

`Project` 继承了 `PluginAware` 、 `ExtensionAware` ，分别用于支持Plugin和Extension方法。部分常用API如下。

```
public interface Project extends Comparable<Project>, ExtensionAware, PluginAware {

    Project getRootProject();

    File getRootDir();

    File getBuildDir();

    void allprojects(Closure configureClosure);

    ScriptHandler getBuildscript();

    void buildscript(Closure configureClosure);

    RepositoryHandler getRepositories();

    void repositories(Closure configureClosure);

    ConfigurationContainer getConfigurations();

    void configurations(Closure configureClosure);

    DependencyHandler getDependencies();

    void dependencies(Closure configureClosure);

    ConfigurableFileCollection files(Object... paths);

    ConfigurableFileTree fileTree(Object baseDir);

    Convention getConvention();

    ExtensionContainer getExtensions();

    Task task(String name) throws InvalidUserDataException;

    Task task(String name, Closure configureClosure);

    void afterEvaluate(Closure closure);

    // ...
}
```

常用API示例（以下脚本均写在 `build.gradle` 中）：

```
// 配置Gradle插件，闭包参数会在ScriptHandler上执行
buildscript {
    // ...
}

// 配置所有工程，闭包参数会分别在每个Project上执行
allprojects {
    // ...
}

// 配置使用的仓库，闭包参数会在RepositoryHandler上执行
```

目录

```
// 配置依赖项，闭包参数会在DependencyHandler上执行。
// files和fileTree也是Project提供的API，
// 而project则是DependencyHandler提供的API。
dependencies {
    compile files('hibernate.jar', 'libs/spring.jar')
    compile fileTree('libs')
    compile project(path: ':library')
    // ...
}

// 在当前Project配置完成后，闭包会被执行
afterEvaluate {
    println "Project '$name' has been evaluated!"
}

// 在RootProject配置完成后，闭包会被执行
rootProject.afterEvaluate {
    println "RootProject '$name' has been evaluated!"
}
```

org.gradle.api.invocation.Gradle

`Gradle` 对象表示一次Gradle调用，通过 `Project.getGradle()` 可以获取这个对象。在一次构建过程中只有一个 `Gradle` 对象，可在其上保存一些全局配置参数，包括StartParameter等。

> Represents an invocation of Gradle.
>
> You can obtain a Gradle instance by calling Project.getGradle().

```
public interface Gradle extends PluginAware {

    String getGradleVersion();

    File getGradleUserHomeDir();

    File getGradleHomeDir();

    Gradle getParent();

    Project getRootProject() throws IllegalStateException;

    void rootProject(Action< ? super Project> action);

    void allprojects(Action< ? super Project> action);

    TaskExecutionGraph getTaskGraph();

    StartParameter getStartParameter();

    ProjectEvaluationListener addProjectEvaluationListener(ProjectEvaluationListener listener);

    void removeProjectEvaluationListener(ProjectEvaluationListener listener);

    void beforeProject(Closure closure);

    void afterProject(Closure closure);

    void buildStarted(Closure closure);

    void settingsEvaluated(Closure closure);

    void projectsLoaded(Closure closure);

    void projectsEvaluated(Closure closure);

    void buildFinished(Closure closure);

    void addBuildListener(BuildListener buildListener);

    public void addListener(Object listener);

    public void removeListener(Object listener);

    public void useLogger(Object logger);
```

Paincker

org.gradle.api.initialization.Settings

`Settings` 对象主要用于配置Project的层级结构。

`Settings` 对象和 `settings.gradle` 文件一一对应。Gradle构建的第一步，就是创建 `Settings` 对象并其上执行 `settings.gradle` 脚本。

> Declares the configuration required to instantiate and configure the hierarchy of org.gradle.api.Project instances which are to participate in a build.
>
> There is a one-to-one correspondence between a Settings instance and a settings.gradle settings file. Before Gradle assembles the projects for a build, it creates a Settings instance and executes the settings file against it.

`Settings` 的部分API如下。

```
public interface Settings extends PluginAware {

    String DEFAULT_SETTINGS_FILE = "settings.gradle";

    void include(String[] projectPaths);

    void includeFlat(String[] projectNames);

    Settings getSettings();

    File getSettingsDir();

    File getRootDir();

    ProjectDescriptor getRootProject();

    ProjectDescriptor project(String path) throws UnknownProjectException;

    ProjectDescriptor findProject(String path);

    ProjectDescriptor project(File projectDir) throws UnknownProjectException;

    ProjectDescriptor findProject(File projectDir);

    StartParameter getStartParameter();

    Gradle getGradle();
}
```

常用API示例：

1. `include()` 可以配置包含Project，例如 `include ':app', ':library'`
2. `project()` 可获取ProjectDescriptor从而做一些配置，例如经常会配置Gradle依赖本地Library工程的路径：

```
include ':img:library'
project(':img:library').projectDir = new File('../../img/library')
```

org.gradle.api.Task

Task

`Task` 也是Gradle中很重要的API。Task代表构建过程中的一个原子操作，例如编译classes文件或生成JavaDoc。

每个Task属于一个Project。每个Task都有一个名字。所属Project名+Task名可组成唯一的完整名(fully qualified path)，例如 `:app:assemble` 。

> A Task represents a single atomic piece of work for a build, such as compiling classes or generating javadoc.
>
> Each task belongs to a Project.
>
> Each task has a name, which can be used to refer to the task within its owning project, and a fully qualified path, which is unique across all tasks in all projects. The path is the concatenation of the owning project's path and the task's name.

Paincker

Action

每个Task包含一个Action序列，并在Task执行时按先后顺序执行。通过Task的doFirst/doLast方法可以往Action序列的头部/末尾添加Action，支持Action或闭包（闭包会被转换成Action对象）。

> A Task is made up of a sequence of Action objects. When the task is executed, each of the actions is executed in turn, by calling Action.execute. You can add actions to a task by calling doFirst(Action) or doLast(Action).
>
> Groovy closures can also be used to provide a task action. When the action is executed, the closure is called with the task as parameter. You can add action closures to a task by calling doFirst(Closure) or doLast(Closure).

Task依赖和排序

每个Task可以依赖其他Task，执行Task时会先执行其依赖的Task，通过dependsOn可设置依赖。每个Task还可以设置在其他Task之前、之后执行，一般可通过mustRunAfter设置。

> A task may have dependencies on other tasks or might be scheduled to always run after another task. Gradle ensures that all task dependencies and ordering rules are honored when executing tasks, so that the task is executed after all of its dependencies and any "must run after" tasks have been executed.

例如下面的配置，执行A时一定会先执行B；执行A不一定会执行C；当A、C都要执行时一定先执行C。

```
taskA.dependsOn(taskB)
taskA.mustRunAfter(taskC)
```

> 除了mustRunAfter，还有个shouldRunAfter要求宽松一些，大部分情况下两者效果相同，特殊情况下有差异，具体可参考官方文档：
> https://docs.gradle.org/3.3/userguide/more_about_tasks.html

常用API

Task的部分常用API如下：

```
public interface Task extends Comparable<Task>, ExtensionAware {

    String getName();

    Project getProject();

    TaskDependency getTaskDependencies();

    Task dependsOn(Object... paths);

    String getPath();

    Task doFirst(Action< ? super Task> action);

    Task doFirst(Closure action);

    Task doLast(Action< ? super Task> action);

    Task doLast(Closure action);

    Task configure(Closure configureClosure);

    Task mustRunAfter(Object... paths);

    TaskDependency shouldRunAfter(Object... paths);

    // ...
}
```

Task创建

> 注：Gradle不推荐使用 `task hello << { ... }` 的方式定义Task，并会在后续版本删除，因此这里不做介绍。

在 `build.gradle` 中创建Task，最常见写法如下。`task(xxx)` 是Project提供的API，最终调用了TaskContainer的create方法。可接收参数包括：

- 闭包配置(可选)

```
task hello(dependsOn: clean) {
    doLast {
        println 'hello'
    }
}
```

也可以直接调用TaskContainer创建Task，Project中的tasks属性即为TaskContainer对象。

```
tasks.create('hello')
```

Task创建后会在Project上添加一个同名方法，调用这个方法可以配置Task。

```
task hello

hello {
    doLast {
        println 'hello'
    }
}
```

Task的type属性，带参数的Task

还可以用类实现Task，创建Task时指定type为这个class即可，定义Task的类通常继承自DefaultTask。下列示例代码中给Task定义了一个名为 `name` 的参数。

```
import org.gradle.api.internal.tasks.options.Option

class HelloTask extends DefaultTask {

    String personName = '';

    HelloTask() {
        doLast {
            println "Hello " + personName
        }
    }

    @Option(description = "set person name", option = "name")
    def setMessage(String name) {
        this.personName = name;
    }
}

task hello(type: HelloTask)
```

命令行中执行效果：

```
$ ./gradlew hello --name Tom
:hello
Hello Tom

BUILD SUCCESSFUL

Total time: 0.889 secs
```

org.gradle.api.plugins.PluginAware

前面介绍的 `Gradle` 、 `Settings` 、 `Project` 等接口均继承了 `PluginAware` 接口， `PluginAware` 主要定义了插件相关API。

```
public interface PluginAware {

    PluginContainer getPlugins();

    void apply(Closure closure);

    void apply(Action< ? super ObjectConfigurationAction> action);
```

```
    PluginManager getPluginManager();
}
```

应用插件

`apply plugin: 'java'`，表示应用Java插件。这个语句调用了`apply()`方法，后面的`plugin: 'java'`是一个Map类型参数。

`apply plugin: MyClass`表示应用指定class实现的插件，将在后面的`Plugin`中介绍。

执行其他Gradle脚本

当一个gradle脚本（例如`build.gradle`）中的代码较多时，可以拆分成多个文件。

1. 新写一个gradle文件例如`my_script.gradle`，把拆分出来的代码放在这个文件中。
2. 在`build.gradle`中通过`apply from: 'my_script.gradle'`或`apply from: new File('xxx/my_script.gradle')`，调用当前目录或指定路径的脚本文件。
3. 新的`my_script.gradle`在被执行时，其代理对象和调用它的`build.gradle`一致，即Project对象。
4. 注意，在新的`my_script.gradle`中定义的属性/方法，在`build.gradle`中不能访问。因为每个gradle文件最后都会被编译成单独的Groovy Script，这些属性/方法只是Script类中的成员。
5. 如果要在不同的脚本文件之间传递数据，可以利用Gradle/Settings/Project对象的ext属性实现。

org.gradle.api.Plugin

`Plugin`用于定义插件。Gradle提供了完整的API框架，而很多工作实际是由插件实现的。Gradle内置了Java、Groovy等几种基础插件，也可以自定义插件。

`Plugin`接口很简单，只有一个apply方法。

```
public interface Plugin<T> {
    /**
     * Apply this plugin to the given target object.
     *
     * @param target The target object
     */
    void apply(T target);
}
```

简易插件开发

下面的示例代码实现了HelloPlugin的简易插件，代码可直接写在`build.gradle`中。

插件在`apply(Project)`方法里，给Project创建了一个名为`hello`的Extension和一个名为`welcome`的Task；Task执行时读取Extension并打印字符串。

在`build.gradle`执行到`apply plugin: HelloPlugin`时，`HelloPlugin.apply(Project)`方法被执行，从而Project有了`hello`的Extension，于是后面可以调用`hello {}`对插件进行配置。

```
class HelloExtension {
    Boolean enable = true
    String text = ''
}

class HelloPlugin implements Plugin<Project> {

    @Override
    void apply(Project project) {
        project.extensions.create('hello', HelloExtension)
        project.task('welcome') {
            doLast {
                HelloExtension ext = project.extensions.hello;
                println ext.enable ? "Hello ${ext.text}!" : 'HelloPlugin is disabled.'
            }
        }
    }
}

apply plugin: HelloPlugin
```

```
    text = 'Gradle'
}
```

在命令行中执行结果如下。

```
$ ./gradlew welcome
:welcome
Hello Gradle!

BUILD SUCCESSFUL

Total time: 0.917 secs
```

独立工程开发插件

对于类似前面示例的简单插件，代码可以直接写在工程的gradle脚本中。而对于需要应用到很多工程的插件，或复杂的插件（例如Android插件），在独立的工程中开发是一个更好的选择。

独立工程开发时，可以新建基于Gradle的Groovy工程。

完整代码可参考示例工程中的plugin模块：

> https://github.com/jzj1993/GradleStudy

插件工程的文件结构如下：

```
.
├── build.gradle
└── src
    └── main
        ├── groovy
        │   └── com
        │       └── paincker
        │           └── gradle
        │               ├── GreetingExtension.groovy
        │               └── GreetingPlugin.groovy
        └── resources
            └── META-INF
                └── gradle-plugins
                    └── greeting.properties
```

其中 `build.gradle` 内容如下。

```
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    compile localGroovy() // Groovy支持(本地)
    compile gradleApi() // GradleAPI支持
}
```

在 `src/main/groovy` 目录下，可编写插件源码，java、groovy均可。

在 `src/main/resources/META-INF/gradle-plugins` 目录下，可以创建若干properties文件：

- 文件名即为插件名，例如 `greeting.properties` ，则插件名为 `greeting`
- 文件内容如下，用implementation-class指定插件的实现类

```
implementation-class=com.paincker.gradle.GreetingPlugin
```

通过gradle的assemble命令将插件打包，默认输出到 `build/libs/plugin.jar` 。

```
$ ./gradlew clean :plugin:assemble
```

独立Gradle插件的使用

Paincker

- 可将前面生成的 `plugin.jar` 直接复制到工程根目录，通过 `classpath files('plugin.jar')` 引入。
- 也可以将插件jar包发布到Maven仓库，通过 `classpath 'com.xxx:greeting:1.0'` 的形式引入。

引入插件包后，用 `apply plugin: 'greeting'` 应用插件， `greeting` 即为前面通过 `properties` 文件名指定的插件名字。

```
buildscript {
    dependencies {
        classpath files('plugin.jar')
    }
}

apply plugin: 'greeting'

greet {
    enable = true
    text = 'Plugin'
}
```

org.gradle.api.logging.Logger

Logger用于输出Gradle的Log。在命令行执行gradle任务则Log输出到命令行，在Android Studio中执行则输出到Gradle Console。

Logger提供以下接口。

```
public interface Logger extends org.slf4j.Logger {

    boolean isLifecycleEnabled();

    void debug(String message, Object... objects);

    void lifecycle(String message);

    void lifecycle(String message, Object... objects);

    void lifecycle(String message, Throwable throwable);

    boolean isQuietEnabled();

    void quiet(String message);

    void quiet(String message, Object... objects);

    void info(String message, Object... objects);

    void quiet(String message, Throwable throwable);

    boolean isEnabled(LogLevel level);

    void log(LogLevel level, String message);

    void log(LogLevel level, String message, Object... objects);

    void log(LogLevel level, String message, Throwable throwable);
}
```

其中LogLevel表示Log等级，有以下值。可通过StartParameter控制Gradle要输出的Log等级。

```
public enum LogLevel {
    DEBUG,
    INFO,
    LIFECYCLE,
    WARN,
    QUIET,
    ERROR
}
```

可通过以下方式获取Logger对象：

- org.gradle.api.logging.Logging.getLogger(Class)
- org.gradle.api.logging.Logging.getLogger(String)
- org.gradle.api.Project.getLogger()