Reflection & Annotation

Rémi Forax

java.lang.Class

A l’exécution

Les types sont pour le compilateur, ils ne sont pas présent à l’exécution

A l’exécution, on ne manipule que des valeurs

- Une valeur est un type primitif ou une référence (une adresse)
- Un CPU ne connait que u32, u64, f32, f64
  - boolean/char/short/int → u32, float → f32, long → u64, double → f64
  - Une référence est soit un u32 soit un u64
    (Compressed Pointer)

Où sont les valeurs à l’exécution

Stockage des valeurs

Une **variable locale** est un décalage par rapport à l’addresse de base d’un appel de fonction sur la pile

Un **champ** est un décalage par rapport à l’adresse d’un objet dans le tas

- Un champ statique est un décalage par rapport à l’adresse de la classe dans le metaspace

Une **case** de **tableau** est à l’adresse: adresse_du_tableau + index * taille_de_élement (8, 16, 32 ou 64)

Description des cases mémoires

Le GC (*precise algorithm*)

- doit connaitre si une case mémoire (dans le tas ou la pile) est une référence ou non

La VM utilise des méta-données

- Pour une méthode, on précalcul la pile maximale et on retient le layout des variables locales à chaque appel
  - Pour les boucles longues, on retient le layout au niveau du goto de retour de boucle
- Pour un objet, la classe décrit le type de chaque champ

java.lang.Class

La classe à l'exécution est une structure décrivant la classe mère, les interfaces, les champs, les méthodes, etc

- cette structure est visible comme un objet

On peut demander à chaque objet quel est l'objet correspondant à sa classe

- Class<?> Object.getClass()

Classe et new

La classe à l’exécution correspond à la classe que l’on indique lors d’un new

On peut obtenir l’objet classe avec la notation (dot class) ".class"

Object objet = **new Car**(…);
object.getClass() == **Car**.class

Object object = "hello";
object.getClass() == String.class;

getClass() vs instanceof

Object.getClass() renvoie la classe d'un objet

L'opérateur instanceof demande si c'est "un sous-type de"

```
Object o = new Car();

o instanceof Car    // true
o instanceof Vehicle  // true, avec Car implements Vehicle

o.getClass() == Car.class  // true
o.getClass() == Vehicle.class // false
```

Class et tableau

Les tableaux en Java sont des objets, donc ils ont une classe

- j.l.Class.isArray() est vrai
- j.l.Class.getComponentType() renvoie le type des éléments

Les classes des tableaux n’ont pas de méthode ni de champ

API de reflection

Java == Langage Dynamique

Comme un langage dynamique (Python, JavaScript, Ruby, etc), Java connait la classe des objets à l'exécution

- donc on peut passer outre le système de typage !

On utilise les méthodes de java.lang.Class pour dynamiquement

- Trouver les méthodes
- Appeler les méthodes avec les arguments

On type toutes les références comme Object

- On converti tous les types primitifs en leur Wrapper

class.getMethods()

Permet d’obtenir toutes les méthodes publiques d’un objet

- Renvoie un tableau de java.lang.reflect.Method

Par exemple,

- Method[] methods = String.class.getMethods();
  ...

j.l.r.Method

j.l.r.Method possède les méthodes

- accessFlags(), getName(),
- getParameterTypes(), getReturnType()
- Object invoke(Object instance, Object… args)

et plein d’autres ...

j.l.r.AccessFlag

Enum contenant les modificateurs (PRIVATE, ABSTRACT, VOLATILE, etc)

Class.accessFlags(), Method.accessFlags(), etc renvoie un Set<AccessFlag>

- Tester un flag
  - method.accessFlags().contains(AccessFlags.PRIVATE)
- Tester des flags
  - method.accessFlags().containsAll(Set.of(AccessFlags.PUBLIC, AccessFlags.STATIC))

Builtin java.lang.Class

La classe java.lang.Class sert aussi à représenter le type des paramètres et type de retour d'une méthode qui peuvent être des types primitif ou void

La syntaxe .class marche aussi

- sur les types primitifs: int.class
- sur void: void.class

Ces objets indiquent qu'ils n'ont pas de méthodes :)

Class.getMethod(String,Class...)

On peut demander une méthode publique précise à partir de son nom et de ses paramètres

```
Method concat = String.class.getMethod(
    "concat", String.class);
System.out.println(concat);
```

Invocation de j.l.r.Method

Permet d'appeler une méthode en indiquant l'objet sur lequel on veut appeler la méthode (null pour les méthodes statiques) suivi des arguments

```
Object invoke(Object receiver, Object… args)
```

Par exemple,

```
var method = String.class.getMethod("concat", String.class);
Object result = method.invoke("hello", "o")
System.out.println(result);
```

getConstructors()
getConstructor(parameterClasses)

On peut obtenir les constructeurs publiques

- class.getConstructors()

ou un constructeur publique à partir du type des paramètres

- class.getConstructor(...parameterClasses)

Par exemple

```
Constructor<Point> constructor =
    java.awt.Point.class.getConstructor(int.class, int.class);
System.out.println(constructor);
```

j.l.r.Constructor

j.l.r.Constructor possède les méthodes

- accessFlags(),
- getParameterTypes()
- newInstance(...args) appel le constructeur
- Et plein d’autres

Par exemple

```
Constructor<Point> constructor =
    java.awt.Point.class.getConstructor(int.class,
int.class);
Object result = constructor.newInstance(2, 3);
```

getFields()/getField(name)

On peut aussi obtenir les champs publiques

- class.getFields()

ou un champ publique à partir de son nom

- class.getField(name)

Par exemple

```
Field field = java.awt.Point.class.getField("x");
System.out.println(field);
```

j.l.r.Field

j.l.r.Field possède les méthodes

- accessFlags(), getName(),
- getType()
- Object get(Object instance)
  - et des versions spécialisées getInt(), getBoolean(), etc
- void set(Object instance, Object newValue)
  - Et des versions spécialisées setChar(), setFloat(), etc

et plein d’autres ...

Exceptions et API de reflection

ReflectionOperationException

Lorsque l’on utilise l’API de reflection, on sort du système de typage donc le compilateur ne controle pas

- Si la méthode existe
- Si la méthode est pas abstraite
- Si elle est bien appelé avec le bon objet
- Si on a le droit d’y accéder
- Etc

donc l’API lève des exceptions dans ces cas.

Exception et getMethod()

Class.getMethod(name, parameterClasses)

- NoSuchMethodException si la méthode n’existe pas

Note: getMethod() marche même si la classe contenant la méthode n’est pas publique

- ou le package contenant la classe est pas exporté

Exception et method.invoke()

Method.invoke(receiver, ...arguments)

- IllegalAccessException
  - La classe contenant la méthode ou la méthode n’est pas accessible **de l’endroit de l’appel**
- IllegalArgumentException
  - Le receiver est pas de la bonne classe
- InvocationTargetException
  - Une exception s’est produite dans la méthode appelée
    - le champs cause (getCause()) contient l’exception

InvocationTargetException

On doit propager l'exception "cause"

```
try {
  ...
} catch(InvocationTargetException e) {
  var cause = e.getCause();
  switch(cause) {
    // on propage les runtime exceptions
    case RuntimeException rte -> throw rte;
    // on propage les erreurs
    case Error error - > throw error;
    // la méthode declare une exception
    default - > throw new UndeclaredThrowableException(cause);
  }
}
```

getConstructor() et newInstance()

class.getConstructor(parameterTypes)

- NoSuchMethodException si le constructeur existe pas

constructor.newInstance(...args)

- IllegalAccessException si le constructeur est pas visible
- IllegalArgumentException si un argument est pas de la bonne classe
- InvocationTargetException si une exception se produit dans le constructeur

getField() et field.set()/get()

class.getField(name)

- NoSuchFieldException si le champ existe pas

field.get(receiver)

- IllegalAccessException si le champ est pas visible
- IllegalArgumentException si le receiver est pas de la bonne classe

field.set(receiver, value)

- IllegalAccessException si le champ est pas visible ou final
- IllegalArgumentException si le receiver ou value est pas de la bonne classe

Enum, Record et classes internes/locales

record

j.l.Class possède une méthode isRecord()

class.getRecordComponents()

- Renvoie un tableau des composants du record

j.l.r.RecordComponent

- getName(), getType()
- Method getAccessor()

enum

j.l.Class possède une méthode isEnum()

class.getEnumConstants()

- Renvoie un tableau des valeurs de l’enum

Classes internes/locales

j.l.Class.getClasses() renvoie un tableau des classes internes publiques

- Et j.l.Class.isMemberClass() est vrai

j.l.Class.getEnclosingMethod(), getEnclosingConstructor() si la classe est déclarée dans une méthode ou un constructeur

- Et j.l.Class.isLocalClass() est vrai

Efficacité / Cache

java.lang.ClassValue

Les méthodes getMethods(), getConstriuctors(), getFields(), etc renvoie un tableau

=> ces méthodes doivent faire une copie défensive

donc c'est pas très efficace

La classe ClassValue<V> permet d'implanter un cache

Marche comme une Map<Class<?>, V>

mais supprime automatiquement la classe si elle est déchargée (si son ClassLoader est null)

java.lang.ClassValue<V>

Possède deux méthodes principales

protected **abstract** V computeValue(Class<?>)

- Doit être redéfinie pour indiquer quelle valeur associée à la classe

public V get(Class<?>)

- API publique qui demande la valeur pour la classe

Marche comme un cache

la première fois que get(Class<?>) est appelée pour une Class<?>, computeValue(Class<?>) est appelée et la valeur est renvoyée

les fois suivantes avec la même Class<?>, la valeur retournée précédemment par computeValue(Class<?>) est renvoyée

Passer outre la sécurité
(enfin un peu)

Tous les membres déclarés

Les méthodes

- getDeclaredMethods(), getDeclaredMethod(name, parameterClasses)
- getDeclaredConstructors(), getDeclaredConstructor(parameterClasses)
- getDeclaredFields(), getDeclaredField(name)

permet d’obtenir tous les membres déclarés (donc pas ceux récupéré par héritage) et pas seulements les membres publiques

Sortir du modèle de sécurité

Lorsque l’on appel (invoke/newInstance/get/set) la sécurité est vérifié par rapport à la méthode qui fait l’appel

On peut appeler la méthode

    setAccessible(true)

pour demander que les appels suivant passe outre la sécurité

setAccessible()

La méthode setAccessible() lève InaccessibleObjectException si il n’est pas possible de passer outre la sécurité

- Si il existe un module et que le module est fermé

Un module correspond au fichier module-info.java

- Les classes du JDK sont dans des modules fermés
  - Pareil pour Spring, et autres frameworks JavaEE
- Si on déclare un module, il est fermé par défaut

Annotations

Annotation

En Java, on peut déclarer soit même ses propres annotations ou utiliser des annotations déjà écrites

Pour utiliser une annotation, on indique le nom de la classe précédé d'une '@' puis entre parenthèse des couples clé/valeur

```
@Author(name = "John Doe",
                company = "DoesSoftware")
public class Foo { … }
```

Les valeurs doivent être des constantes

Créer une annotation

Une annotation est une interface spéciale contenant

- des méthodes sans paramètres
  - Les types de retour doit être un type de constante
  - Qui peuvent spécifier une valeur par défaut

- Annoté par des méta-annotations qui indique
  - Où on peut utliser l’annotation
  - Si l’annotation est visible à l’exécution

Créer une annotation

On utilise @interface pour déclarer une annotation

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
  String name();
  String company();
  int year() default 0;  // indique une valeur
                         // par défaut
}
```

Meta-annotations

Permet d’indiquer comment l’annotation est utilisable

- Où peut-on utiliser l’annotation ?
  @Target(ElementType…)
  - CONSTRUCTOR, FIELD, LOCAL_VARIABLE, RECORD_COMPONENT, etc
- Garder dans le .class, accessible par reflection ?
  @Retention(RetentionPolicy)
  - SOURCE, CLASS, RUNTIME
- Propager aux sous-classes ?
  @Inherited
- Apparait dans la javadoc ?
  @Documented

3 sortes d’annotations

Marker annotation, annotation sans méthode

```
public @interface Version0 { }
```

Annotation avec une seule valeur, la méthode doit s’appeler “value”

```
public @interface Version1 {
  String value()
}
```

Annotation à plusieurs valeurs

```
public @interface Version2 {
  int major();
  int minor();
}
```

Utiliser les 3 sortes d’annotations

- Marker annotation

```
@Version0
public class Foo { }
```

- Annotation à une valeur

```
@Version1("14.0")
public record Foo() { }
```

- Annotation avec plusieurs valeurs

```
@Version2({major = 14, minor = 7})
public enum Foo { }
```

Type des constantes

Une valeur d’une annotation peut être de type

- Type primitif: int, float, char, etc
- String
- Class
- Un enum (comme RetentionPolicy)
- Une annotation
- Un tableau des types précédents (comme ElementType[])

en fait, tous les types qui peuvent contenir une constante pour le compilateur

Pas d’héritage

Comme toutes les features récentes (2004 pour les annotations) de Java, les annotations ne permettent pas l’héritage

On utilise la composition à la place

- Une annotation peut contenir une annotation

Implicitement, les annotations hérite de l’interface java.lang.annotation.Annotation

Tableau et Arbre d’annotation

Il n’est pas possible de créer des graphes (pas de boucle), mais aucun problème pour créer des arbres

```
public @interface Version {
  int value();
}

public @interface Product {
  Version[] versions();
}
```

Et à l’utilisation

```
@Product(versions = { @Version(1), @Version(2) })
public class Foo { }
```

Répétition des annotations

Par défaut, une annotation n’est pas repetable, mais on peut définir un container d’annotation, et le meta-annoter avec @Repeatable(class)

- A la déclaration

```
public @interface Hello() { }

@Repeatable(Hello.class)
public @interface HelloContainer {
  Hello[] container();
}
```

- A l’utilisation

```
@Hello @Hello
public record Foo() { }
```

Annotation et Reflection

Accessible par reflection

Pour être accessible, une annotation doit être annoter avec la méta-annotation @Retention avec la RentionPolicy.RUNTIME

Par exemple,

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Hello { }
```

A l’exécution

Le type d’une annotation est une interface (qui hérite de java.lang.annotation.Annotation)

Une annotation est instance d’une classe (créer par le JDK) qui implante l’interface correspondant à l’annotation

- Si on appel une des méthodes de l’interface, la valeur correspondant à l’annotation est renvoyée

AnnotatedElement

Interface commune pour toutes les élements annotable (Class, Method, Constructor, Field, RecordComponent, etc)

- Tester la présence d'une annotation

```
boolean isAnnotationPresent(
        Class<? extends Annotation> annotationType)
```

- Demander la valeur d'une annotation (renvoie null si pas trouvée)

```
<A extends Annotation> A getAnnotation(
        Class<A> annotationType)
```

Exemple

Avec

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Say {
  String message();
}

@Say(message="hello")
public class Foo { }
```

On récupère l'annotation

```
Say say = Foo.class.getAnnotation(Say.class);
System.out.println(say.message()); // appel de la méthode
                                   // de l'interface
```

En conclusion

L’API de reflection permet de

- Passer outre le système de typage
  - Ne permet pas de passer outre la sécurité si un module est définie
  - Demande de gérer correctement les exceptions
- La déclaration d’une annotation est une interface
- L’utilisation d’une annotation est une instance d’une classe qui implante cette interface