

Конспект лекции

Механизмы отражения и проксирования

Цель и задачи лекции

Цель - изучить механизмы отражения и проксирования.

Задачи:

1. Изучить способы получения данных с помощью использования механизмов отражения
2. Дать понятие механизму проксирования

План занятия

1. Reflection API
2. Механизм проксирования

Reflection API

Отражение (reflection) — способность программы анализировать саму себя.

Рефлексия позволяет:

- получать информацию о переменных, методах внутри класса, о самом классе, его конструкторах, реализованных интерфейсах и т.д.;
- получать новый экземпляр класса;
- получать доступ ко всем переменным и методам, в том числе приватным;
- преобразовывать классы одного типа в другой;
- делать все это во время исполнения программы (динамически, в Runtime).

Недостатки рефлексии:

- Худшая производительность в сравнении с классической работой с классами, методами и переменными;
- Ограничения безопасности. Если мы захотим использовать рефлексия на классе, который защищен с помощью специального класса SecurityManager, то ничего у нас не выйдет т.к. этот класс будет выбрасывать исключения каждый раз, как мы попытаемся получить доступ к закрытым членам класса. Такая защита может применяться, например, в Апплетах (Applets);
- Получение доступа к внутренностям класса, что нарушает принцип инкапсуляции. Фактически, мы получаем доступ туда, куда обычному человеку лезть не

желательно. Это как с розеткой, ребёнку лучше к ней не лезть, тогда как опытный электрик запросто с ней поладит.

Тип данных Class

В Java есть специальный класс по имени Class.

Class есть у:

- классов, интерфейсов, перечислений;
- примитивов и обёрток над ними;
- массивов;
- void. Да, ключевое слово void также имеет Class.

Class есть у всех объектов в Java.

Получение имени класса

Чтобы получить инстанс объекта класса Class можно воспользоваться одним из следующих способов.

1. Class.forName("имя.пакета.ИмяКласса")

```
try {
    Class<?> carClass = Class.forName("com.package.MyClass");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

Вызов метода forName() необходимо обернуть в блок try-catch т.к. метод может бросить ClassNotFoundException, в случае если он не найдет класс с таким именем.

2. метод getClass() у экземпляра класса

```
MyClass car = new MyClass();
Class<? extends MyClass> myClass = car.getClass();
```

В этом случае оборачивать метод getClass() в блок try-catch нет необходимости т.к. мы вызываем этот метод у существующего класса, который видит компилятор. Но, к сожалению, компилятор не может знать тип переменной до конца, поэтому мы и имеем "? extends MyClass", как дженерик тип.

3. ИмяКласса.class

```
Class<MyClass> myClass = MyClass.class;
```

Получение информации об иерархической структуре класса

Можно также использовать метод getSuperclass() для объекта Class, чтобы получить объект типа Class, представляющий суперкласс рефлексированного класса. Нужно не забывать учитывать, что в Java отсутствует множественное наследование и класс java.lang.Object является базовым классом для всех классов, вследствие чего если у

класса нет родителя то метод `getSuperclass` вернет `null`. Для того чтобы получить все родительские суперклассы, нужно рекурсивно вызывать метод `getSuperclass()`.

```
Class c = myObj.getClass();
Class superclass = c.getSuperclass();
```

С помощью рефлексии можно также определить, какие интерфейсы реализованы в заданном классе. Метод `getInterfaces()` вернет массив объектов типа `Class`. Каждый объект в массиве представляет один интерфейс, реализованный в заданном классе.

```
Class c = LinkedList.class;
Class[] interfaces = c.getInterfaces();
for(Class cInterface : interfaces) {
    System.out.println( cInterface.getName() );
}
```

Исследование модификаторов доступа класса

```
Class c = obj.getClass();
int mods = c.getModifiers();
if (Modifier.isPublic(mods)) {
    System.out.println("public");
}
if (Modifier.isAbstract(mods)) {
    System.out.println("abstract");
}
if (Modifier.isFinal(mods)) {
    System.out.println("final");
}
```

Чтобы узнать, какие модификаторы были применены к заданному классу, сначала нужно с помощью метода `getClass` получить объект типа `Class`, представляющий данный класс. Затем нужно вызвать метод `getModifiers()` для объекта типа `Class`, чтобы определить значение типа `int`, биты которого представляют модификаторы класса. После этого можно использовать статические методы класса `java.lang.reflect.Modifier`, чтобы определить, какие именно модификаторы были применены к классу.

Получение информации о переменных класса

Получить информацию о переменных класса можно с помощью методов `getDeclaredFields()`, `getDeclaredField()` и `getFields()`, `getField()`.

1. `getDeclaredFields()`. Метод возвращает все объявленные переменные в классе.

```
Class<Car> carClass = Car.class;
Field[] declaredFields = carClass.getDeclaredFields();
for (Field field : declaredFields) {
    System.out.println(field);
}
```

2. `getDeclaredField()`. Метод возвращает переменную по её имени. Если переменной с таким именем нет, то метод выбросит `checked NoSuchFieldException`.

```
Class<Car> carClass = Car.class;
try {
    Field horsepowerField = carClass.getDeclaredField("horsepower");
    System.out.println(horsepowerField);
}
```

```
Field blaBlaField = carClass.getDeclaredField("bla_bla");
} catch (NoSuchFieldException e) {
    e.printStackTrace();
}
```

3. **getFields()**. В отличии от метода **getDeclaredFields()**, метод **getFields()** возвращает **только public** переменные.

```
Class<Car> carClass = Car.class;
Field[] fields = carClass.getFields();
for (Field field : fields) {
    System.out.println(field);
}
```

4. **getField()**. По аналогии с методом **getFields()**, метод **getField()** возвращает только **public** переменные. Даже если поле с таким именем есть, но оно не публичное, метод **getField()** бросит **NoSuchFieldException**

```
Class<Car> carClass = Car.class;
try {
    Field serialNumberField = carClass.getField("serialNumber");
    System.out.println(serialNumberField);
    Field horsepowerField = carClass.getField("horsepower");
} catch (NoSuchFieldException e) {
    e.printStackTrace();
}
```

Получение информации о методах в классе

1. **getDeclaredMethods()**. Метод возвращает все объявленные методы в классе.

```
Class<Car> carClass = Car.class;
Method[] declaredMethods = carClass.getDeclaredMethods();
for (Method method : declaredMethods) {
    System.out.println(method);
}
```

2. **getDeclaredMethod()**. Метод принимает имя и аргументы с типами параметров метода. Если такого метода в классе нет, возникнет исключение **NoSuchMethodException**.
3. **getMethods()**. Метод возвращает все **public** методы класса и **public** методы его родительского класса/интерфейсов
4. **getMethod()**. Как и **getMethods()**, метод **getMethod()** возвращает только публичные методы. Если такого метода нет или он не публичный, мы получим **NoSuchMethodException**.
5. **getEnclosingMethod()**. Если класс является локальным или анонимным, метод возвращает тот метод, в котором этот класс был создан, иначе метод возвращает **null**.

Тип данных Field

Класс **Field** предоставляет возможность:

- получить значение поля, его тип, имя и модификаторы поля

- получить список аннотаций, класс, в котором объявлено поле и другую информацию
- установить новое значение в поле, даже если оно объявлено как private

Рассмотрим класс:

```
class Car {
    private int horsepower;
    public final String serialNumber;

    public Car(int horsepower, String serialNumber) {
        this.horsepower = horsepower;
        this.serialNumber = serialNumber;
    }
}
```

Получение значения переменной

Для того, чтобы получить значение из класса Field существуют методы `getByte()`, `getShort()`, `getInt()`, `getLong()`, `getFloat()`, `getDouble()`, `getChar()`, `getBoolean()` и `get()`. Первые 8 методов существуют для получения примитивов, а последний для получения объектов.

Получение поля с модификатором доступ `public`:

```
Car car = new Car(500, "1233");
Class<? extends Car> carClass = car.getClass();
Field serialNumberField = carClass.getDeclaredField("serialNumber");

//указываем из какого объекта хотим получить значение
String serialNumberValue = (String) serialNumberField.get(car);
System.out.println(serialNumberValue); //output: 1233
```

Получение поля с модификатором доступ `private`:

```
Car car = new Car(500, "1233");
Class<? extends Car> carClass = car.getClass();
Field horsepowerField = carClass.getDeclaredField("horsepower");

horsepowerField.setAccessible(true);
int horsepowerValue = horsepowerField.getInt(car);
System.out.println(horsepowerValue); //output: 500
```

В данном случае необходимо вызвать метод `setAccessible(true)`, в противном случае возникнет исключение `IllegalAccessException`.

Получение имени, типа и модификаторов переменной

Для получения имени, типа, модификатора необходимо воспользоваться методами `getName()`, `getType()`, `getModifiers()`.

```
Car car = new Car(500, "1233");
Class<? extends Car> carClass = car.getClass();
Field horsepowerField = carClass.getDeclaredField("horsepower");

String name = horsepowerField.getName();
System.out.println(name); //output: horsepower
```

```

Class<?> type = horsepowerField.getType();
System.out.println(type); //output: int

int modifiers = horsepowerField.getModifiers();
System.out.println(modifiers); //output: 2

```

Получение конструкторов класса

Чтобы получить информацию об открытых конструкторах класса, нужно вызвать метод `getConstructors()` для объекта `Class`. Этот метод возвращает массив объектов типа `java.lang.reflect.Constructor`. С помощью объекта `Constructor` можно затем получить имя конструктора, модификаторы, типы параметров и генерируемые исключения. Можно также получить по отдельному открытому конструктору, если известны типы его параметров.

```

Class c = obj.getClass();

Constructor[] constructors = c.getConstructors();
for (Constructor constructor : constructors) {
    Class[] paramTypes = constructor.getParameterTypes();
    for (Class paramType : paramTypes) {
        System.out.print(paramType.getName() + " ");
    }
    System.out.println();
}

```

Методы `getConstructor()` и `getConstructors()` возвращают только открытые конструкторы. Если требуется получить все конструкторы класса, включая закрытые можно использовать методы `getDeclaredConstructor()` и `getDeclaredConstructors()` эти методы работают точно также, как их аналоги `getConstructor()` и `getConstructors()`.

Класс Method

Класс `Method` предоставляет возможность:

- получить название метода, его модификаторы, тип возвращаемого значения и входящих параметров
- получить аннотации метода, бросаемые исключения и другую информацию
- вызвать метод, даже приватный

Получение информации о методе и его параметрах

Чтобы получить информацию об открытых методах класса, нужно вызвать метод `getMethods()` для объекта `Class`. Этот метод возвращает массив объектов типа `java.lang.reflect.Method`. Затем с помощью объекта `Method` можно получить имя метода, тип возвращаемого им значения, типы параметров, модификаторы и генерируемые исключения.

```

Class c = obj.getClass();
Method[] methods = c.getMethods();
for (Method method : methods) {
    System.out.println("Имя: " + method.getName());
    System.out.println("Возвращаемый тип: " + method.getReturnType().getName());
}

```

```

Class[] paramTypes = method.getParameterTypes();
System.out.print("Типы параметров: ");
for (Class paramType : paramTypes) {
    System.out.print(" " + paramType.getName());
}
System.out.println();
}

```

Также можно получить информацию по отдельному методу если известны имя метода и типы параметров.

```

Class c = obj.getClass();
Class[] paramTypes = new Class[] { int.class, String.class};
Method m = c.getMethod("methodA", paramTypes);

```

Методы `getMethod()` и `getMethods()` возвращают только открытые методы, для того чтобы получить все методы класса не зависимо от типа доступа, нужно воспользоваться методами `getDeclaredMethod()` и `getDeclaredMethods()`, которые работают точно также как и их аналоги (`getMethod()` и `getMethods()`). Интерфейс Java Reflection Api позволяет динамически вызвать метод, даже если во время компиляции имя этого метода неизвестно.

Загрузка и динамическое создание экземпляра класса

С помощью методов `Class.forName()` и `newInstance()` объекта `Class` можно динамически загружать и создавать экземпляры класса в случае, когда имя класса неизвестно до момента выполнения программы.

```

Class c = Class.forName("Test");
Object obj = c.newInstance();
Test test = (Test) obj;

```

В приведенном коде мы загружаем класс с помощью метода `Class.forName()`, передавая имя этого класса. В результате возвращается объект типа `Class`. Затем мы вызываем метод `newInstance()` для объекта типа `Class`, чтобы создать экземпляры объекта исходного класса. Метод `newInstance()` возвращает объект обобщенного типа `Object`, поэтому в последней строке мы приводим возвращенный объект к тому типу, который нам нужен.

```

class WithPrivateFinalField {

    private int i = 1;

    private final String s = "String S";

    private String s2 = "String S2";

    public String toString() {

        return "i = " + i + ", " + s + ", " + s2;

    }
}

```

```

}

public class ModifyngPrivateFields {

    public static void main(String[] args) throws Exception {

        WithPrivateFinalField pf = new WithPrivateFinalField();

        Field f = pf.getClass().getDeclaredField("i");

        f.setAccessible(true);

        f.setInt(pf, 47);

        System.out.println(pf);

        f = pf.getClass().getDeclaredField("s");

        f.setAccessible(true);

        f.set(pf, "MODIFY S");

        System.out.println(pf);

        f = pf.getClass().getDeclaredField("s2");

        f.setAccessible(true);

        f.set(pf, "MODIFY S2");

        System.out.println(pf);

    }

}

```

Из приведённого кода видно, что private поля можно изменять. Для этого требуется получить объект типа `java.lang.reflect.Field` с помощью метода `getDeclaredField()`, вызвать метод `setAccessible(true)` и с помощью метода `set()`

устанавливаем значение поля. Следует учесть, что поле `final` при выполнении данной процедуры не выдаёт предупреждений, а значение поля остаётся прежним, т.е. `final` поля остаются неизменными.

Механизм проксирования

Предположим, что у нас есть класс `A`, реализующий некоторые интерфейсы. Java-машина во время исполнения может сгенерировать прокси-класс для данного класса `A`, т.е. такой класс, который реализует все интерфейсы класса `A`, но заменяет вызов всех методов этих интерфейсов на вызов метода `InvocationHandler#invoke`, где `InvocationHandler` - интерфейс JVM, для которого можно определять свои реализации.

Создается прокси-класс с помощью вызова метода `Proxy.getProxyClass`, который принимает класс-лоадер и массив интерфейсов (`interfaces`), а возвращает объект класса `java.lang.Class`, который загружен с помощью переданного класс-лоадера и реализует переданный массив интерфейсов.

На передаваемые параметры есть ряд ограничений:

1. Все объекты в массиве `interfaces` должны быть интерфейсами. Они не могут быть классами или примитивами.
2. В массиве `interfaces` не может быть двух одинаковых объектов.
3. Все интерфейсы в массиве `interfaces` должны быть загружены тем класс-лоадером, который передается в метод `getProxyClass`.
4. Все не публичные интерфейсы должны быть определены в одном и том же пакете, иначе генерируемый прокси-класс не сможет их все реализовать.
5. Ни в каких двух интерфейсах не может быть метода с одинаковым названием и сигнатурой параметров, но с разными типами возвращаемого значения.
6. Длина массива `interfaces` ограничена 65535-ю интерфейсами. Никакой Java-класс не может реализовывать более 65535 интерфейсов.

Если какое-либо из вышеперечисленных ограничений нарушено - будет выброшено исключение `IllegalArgumentException`, а если массив интерфейсов `interfaces` равен `null`, то будет выброшено `NullPointerException`.

Свойства динамического прокси-класса

Необходимо сказать пару слов о свойствах класса, создаваемого с помощью `Proxy.getProxyClass`. Данные свойства следующие:

1. Прокси-класс является публичным, снабжен модификатором `final` и не является абстрактным.
2. Имя прокси-класса по-умолчанию не определено, однако начинается на `$Proxy`. Все пространство имен, начинающихся на `$Proxy` зарезервировано для прокси-классов.
3. Прокси-класс наследуется от `java.lang.reflect.Proxy`.

4. Прокси-класс реализует все интерфейсы, переданные при создании, в порядке передачи.
5. Если прокси-класс реализует непубличный интерфейс, то он будет сгенерирован в том пакете, в котором определен этот самый непубличный интерфейс. В общем случае пакет, в котором будет сгенерирован прокси-класс неопределен.
6. Метод `Proxy.isProxyClass` возвращает `true` для классов, созданных с помощью `Proxy.getProxyClass` и для классов объектов, созданных с помощью `Proxy.newProxyInstance` и `false` в противном случае. Данный метод используется подсистемой безопасности Java и нужно понимать, что для класса, просто унаследованного от `java.lang.reflect.Proxy` он вернет `false`.
7. `java.security.ProtectionDomain` для прокси-класса такой же, как и для системных классов, загруженных bootstrap-загрузчиком, например - для `java.lang.Object`. Это логично, потому что код прокси-класса создается самой JVM и у нее нет причин себе не доверять.

Экземпляр динамического прокси-класса и его свойства

Конструктор прокси-класса принимает один аргумент - реализацию интерфейса `InvocationHandler`. Соответственно, объект прокси-класса можно создать с помощью рефлексии, вызвав метод `newInstance` объекта класса `Class`. Однако, существует и другой способ - вызвать метод `Proxy.newProxyInstance`, который принимает на вход загрузчик классов, массив интерфейсов, которые будет реализовывать прокси-класс, и объект, реализующий `InvocationHandler`. Фактически, данный метод комбинирует получение прокси-класса с помощью `Proxy.getProxyClass` и создание экземпляра данного класса через рефлексия.

Свойства созданного экземпляра прокси-класса:

1. Объект прокси-класса приводим ко всем интерфейсам, переданным в массиве `interfaces`. Если `IDemo` - один из переданных интерфейсов, то операция `proxy instanceof IDemo` всегда вернет `true`, а операция `(IDemo) proxy` завершится корректно.
2. Статический метод `Proxy.getInvocationHandler` возвращает обработчик вызовов, переданный при создании экземпляра прокси-класса. Если переданный в данный метод объект не является экземпляром прокси-класса, то будет выброшено `IllegalArgumentException` исключение.
3. Класс-обработчик вызовов реализует интерфейс `InvocationHandler`, в котором определен метод `invoke`, имеющий следующую сигнатуру:

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
```

Здесь `proxy` - экземпляр прокси-класса, который может использоваться при обработке вызова того или иного метода. Второй параметр - `method` является экземпляром класса `java.lang.reflect.Method`. Значение данного параметра - один из методов, определенных в каком-либо из переданных при создании прокси-класса интерфейсов или их супер-интерфейсов. Третий параметр - массив значений аргументов метода. Аргументы примитивных типов будут заменены экземплярами своих классов-

обертки, таких как `java.lang.Boolean` или `java.lang.Integer`. Конкретная реализация метода `invoke` может изменять данный массив.

Значение, возвращаемое методом `invoke` должно иметь тип, совместимый с типом значения, возвращаемого интерфейсным методом, для которого вызывается данная обертка. В частности, если интерфейсный метод возвращает значение примитивного типа - необходимо вернуть экземпляр класса-обертки данного примитивного типа. Если возвращается `null`, а ожидается значение примитивного типа, - будет выброшено `NullPointerException`. В случае непримитивных типов, класс возвращаемого значения метода `invoke` должен быть приводим к классу возвращаемого значения интерфейсного метода, иначе будет выброшено `ClassCastException`.

Внутри метода `invoke` должны бросаться только те проверяемые исключения, которые определены в сигнатуре вызываемого интерфейсного метода либо приводимые к ним. Помимо этих типов исключений разрешается бросать только непроверяемые исключения (такие как `java.lang.RuntimeException`) или ошибки (например, `java.lang.Error`). Если внутри метода `invoke` выброшено проверяемое исключение несопоставимое с описанными в сигнатуре интерфейсного метода - то будет так же выброшено исключение `UndeclaredThrowableException`.

Методы `hashCode`, `equals` и `toString`, определенные в классе `Object`, так же будут вызываться не на прямую, а через метод `invoke` наравне со всеми интерфейсными методами. Другие публичные методы класса `Object` будут вызываться напрямую.

Литература и ссылки

1. Спецификация языка Java <https://docs.oracle.com/javase/specs/jls/se8/html/index.html>
2. Шилдт Г. Java 8. Полное руководство. 9-е издание. — М.: Вильямс, 2012. — 1377 с.
3. Эккель Б. Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.
4. <https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html>

Вопросы для самоконтроля

1. В чем мотивация использования рефлексии?
2. Зачем нам нужен Proxy?