Java для молодых специалистов Банка ВТБ

Урок 9

Reflection API. JDBC. Основы PostgreSQL

Получение информации о классе через Reflection API. Аннотации. Взаимодействие с БД через JDBC, работа с PostgreSQL.

Рефлексия. Общая информация

Java Reflection API позволяет исследовать классы, интерфейсы, поля и методы во время выполнения программы, ничего не зная о них на этапе компиляции. Также с ее помощью можно создавать новые объекты, вызывать у них методы и работать с полями.

Поскольку мы собираемся получать информацию о классах в процессе выполнения программы, то необходимо научиться получать класс в виде Java объекта (объекта типа Class). Для этого есть три возможных варианта.

1) У любого Java объекта можно вызвать метод getClass(), который вернет объект типа Class.

```
public class ReflectionApp {
    public static void main(String[] args) {
        Class stringClass = "Java".getClass();
    }
}
```

2) Запросить объект типа Class напрямую у класса.

```
public class ReflectionApp {
    public static void main(String[] args) {
        Class integerClass = Integer.class;
        Class stringClass = String.class;
        Class intClass = int.class;
        Class voidClass = void.class;
        Class charArrayClass = char[].class;
    }
}
```

3) Вызвать статический метод Class.forName(), и передать ему полное имя класса в качестве аргумента.

```
public class ReflectionApp {
    public static void main(String[] args) {
        try {
            Class jdbcClass = Class.forName("org.sqlite.jdbc");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Заметьте что у любого типа данных Java есть класс: String, int, int[], Integer, char[][], и даже void. Также стоит учесть, что классы int, int[], int[][], int[][]...[] отличаются, это можно легко увидеть, если распечатать их имена в консоль.

Изучаем классы

Имея в распоряжении объект типа Class, мы можем в процессе выполнения программы получить информацию о структуре этого класса (конструкторах, полях, методах, модификаторах доступа, аннотациях, интерфейсах, внутренних классах, и пр.).

Имя класса. Для получения полного имени класса (пакет + имя класса) можно воспользоваться методом getName(), без указания пакета – getSimpleName(). Для класса String эти методы выдадут java.lang.String и String соответственно.

```
public class ReflectionApp {
    public static void main(String[] args) {
        Class s = String.class;
        System.out.println("Полное имя класса: " + s.getName());
        System.out.println("Простое имя класса: " + s.getSimpleName());
    }
}

// Результат:
// Полное имя класса: java.lang.String
// Простое имя класса: String
```

Модификаторы класса. Метод getModifiers() возвращает значение типа int, из которого, с помощью статических методов класса Modifier, можно определить какие именно модификаторы были применены к классу.

```
public class ReflectionApp {
    public static void main(String[] args) {
        Class strClass = String.class;
        int modifiers = strClass.getModifiers();
        if (Modifier.isPublic(modifiers)) {
            System.out.println(strClass.getSimpleName() + " - public");
        }
        if (Modifier.isAbstract(modifiers)) {
            System.out.println(strClass.getSimpleName() + " - abstract");
        }
        if (Modifier.isFinal(modifiers)) {
            System.out.println(strClass.getSimpleName() + " - final");
        }
    }
}

// Результат:
// String - public
// String - final
```

По такому же принципу можно получить модификаторы полей и методов. Для проверки модификаторов используются методы isPublic(), isPrivate(), isAbstract(), isFinal(), isNative(), isInterface(), isSynchronized(), isVolatile(), isStrict(), isTransient(), isProtected(), isStatic().

Суперкласс. Метод getSuperclass() позволяет получить объект типа Class, представляющий суперкласс рефлексированного класса. Для получения всей цепочки родительских классов достаточно рекурсивно вызывать метод getSuperclass() до получения null. Его вернет Object.class.getSuperclass(), так как у него нет родительского класса.

Интерфейсы, реализуемые классом. Метод getInterfaces() возвращает массив объектов типа Class. Каждый из них представляет один интерфейс, реализованный в заданном классе.

Поля класса. Метод getFields() возвращает массив объектов типа Field, соответствующих всем открытым (public) полям класса. Класс Field содержит информацию о полях класса.

```
public class Cat {
    public String name;
    public String color;
    public int age;
}

public class ReflectionApp {
    public static void main(String[] args) {
        Class catClass = Cat.class;
        Field[] publicFields = catClass.getFields();
        for (Field o : publicFields) {
            System.out.println("Тип_поля Имя_поля : " + o.getType().getName() + "
" + o.getName());
        }
    }
}

// Результат:
// Тип_поля Имя_поля : java.lang.String name
// Тип_поля Имя_поля : java.lang.String color
// Тип_поля Имя_поля : int age
```

** В этом примере у всех полей модификатор доступа установлен как public, чтобы можно было получить их список с помощью метода getFields().*

Чтобы получить все поля класса (public, private и protected), применяют метод getDeclaredFields(). Зная имя поля, можно получить ссылку на него через метод getField() или getDeclaredField().

```
public class ReflectionApp {
    public static void main(String[] args) {
        Class catClass = Cat.class;
        Field f = catClass.getDeclaredField("name");
    }
}
```

Получить значение поля можно с помощью метода get(), который принимает входным параметром ссылку на объект класса. Для «чтения» примитивных типов применяют методы getInt(), getFloat(), getByte() и другие. Метод set() предназначен для изменения значения поля. Пример:

```
public static void main(String[] args) {
    try {
        Cat cat = new Cat();
        Field fieldName = cat.getClass().getField("name");
        fieldName.set(cat, "Мурзик");
        Field fieldAge = cat.getClass().getField("age");
        System.out.println(fieldAge.get(cat));
    } catch (NoSuchFieldException | IllegalAccessException e) {
        e.printStackTrace();
    }
}
```

Получение доступа к private полям

Посредством рефлексии можно получать и изменять значения полей с модификатором доступа private.

```
public class ClassWithPrivateField {
    private int field;

    public ClassWithPrivateField(int field) {
        this.field = field;
    }

    public void info() {
        System.out.println("field: " + field);
    }
}

public class ReflectionApp {
    public static void main(String[] args) {
        try {
            ClassWithPrivateField obj = new ClassWithPrivateField(10);
            obj.info();
            Field privateField =
ClassWithPrivateField.class.getDeclaredField("field");
            privateField.setAccessible(true);
            System.out.println("get: " + privateField.get(obj));
            privateField.set(obj, 1000);
            obj.info();
        } catch (NoSuchFieldException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

```
// Результат:
// field: 10
// get: 10
// field: 1000
```

Для этого получаем объект типа **Field** и открываем к нему доступ через **setAccessible(true)**. Затем получаем и изменяем его значение – по аналогии с предыдущим примером. Изменить **final** поле нельзя даже при помощи рефлексии.

Конструкторы класса

Методы **getConstructors()** и **getDeclaredConstructors()** возвращают массив объектов типа **Constructor**. Они содержат в себе информацию о конструкторах класса: имя, модификаторы, типы параметров, генерируемые исключения. Если известен набор параметров конструктора, можно получить ссылку на него с помощью **getConstructor()** или **getDeclaredConstructor()**.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

    public Cat(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Cat(String name) {
        this.name = name;
    }
}

public class ReflectionApp {
    public static void main(String[] args) {
        Constructor[] constructors = Cat.class.getConstructors();
        for (Constructor o : constructors) {
            System.out.println(o);
        }
        System.out.println("---");
        try {
            System.out.println(Cat.class.getConstructor(new Class[] {String.class,
int.class}));
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
    }
```

```
}

// Результат:
// public Cat(java.lang.String,java.lang.String,int)
// public Cat(java.lang.String,int)
// public Cat(java.lang.String)
// ---
// public Cat(java.lang.String,int)
```

Работа с методами

Методы **getMethods()** и **getDeclaredMethods()** возвращают массив объектов типа **Method**, в которых содержится полная информация о методах класса. Если известно имя метода и набор входных параметров, то можно получить ссылку на него с помощью **getMethod()** или **getDeclaredMethod()**.

```
public class MainClass {
    public static void main(String[] args) {
        Method[] methods = Cat.class.getDeclaredMethods();
        for (Method o : methods) {
            System.out.println(o.getReturnType() + " ||| " + o.getName() + " ||| "
+ Arrays.toString(o.getParameterTypes()));
        }
        try {
            Method m1 = Cat.class.getMethod("jump", null);
            Method m2 = Cat.class.getMethod("meow", int.class);
            System.out.println(m1 + " | " + m2);
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
    }
}
Результат:
void ||| jump ||| []
void ||| meow ||| [int]
class java.lang.String ||| getColor ||| []
public void Cat.jump() | public void Cat.meow(int)
```

Java Reflection позволяет динамически вызвать метод, даже если во время компиляции его имя было неизвестно.

```
public class Cat {
    // ...
    public void meow(int dB) {
        System.out.println(name + ": meow - " + dB + " dB");
    }
    // ...
}
public class MainClass {
```

```
    public static void main(String[] args) {
        Cat cat = new Cat("Barsik");
        try {
            Method mMeow = Cat.class.getDeclaredMethod("meow", int.class);
            mMeow.invoke(cat, 5);
        } catch (NoSuchMethodException | IllegalAccessException |
InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
Результат:
Barsik: meow - 5 dB
```

В этом примере сначала в классе **Cat** находим метод **meow**. Затем вызываем у него **invoke()**, который у выбранного объекта вызывает этот метод и принимает два параметра. Первый – это объект класса **Cat**, а второй – набор аргументов, передаваемых методу **meow()**.

Если у метода модификатор доступа **private**, то получить к нему доступ можно по аналогии с нашим примером о **private**-поле.

Создание объектов

```
public class MainClass {
    public static void main(String[] args) {
        try {
            Class someClass = Cat.class;
            Constructor catCounstructor = Cat.class.getConstructor(String.class,
String.class, int.class);
            Cat cat1 = (Cat)someClass.newInstance();
            Cat cat2 = (Cat)catCounstructor.newInstance("Murzik", "Black", 3);
        } catch (InstantiationException | IllegalAccessException |
NoSuchMethodException | InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
```

Метод **newInstance()** позволяет создавать экземпляры класса через объект типа **Class** и возвращает объект типа **Object**. Если этот метод вызван у объекта типа **Class**, то для создания нового объекта используется конструктор по умолчанию. Если он отсутствует – будет брошено исключение. Если вначале получаем объект типа **Constructor** с заданным набором параметров, то **newInstance()** использует этот набор.

Аннотации

Для создания аннотации создаем интерфейс и ставим символ @ перед ключевым словом **interface**. Необходимо указать две аннотации:

- **@Retention** – сообщает, где будет использоваться аннотация:
  - **RetentionPolicy.SOURCE** – используется на этапе компиляции и должна отбрасываться компилятором;
  - **RetentionPolicy.CLASS** – будет записана в .class-файл, но не будет доступна во время выполнения;
  - **RetentionPolicy.RUNTIME** – будет записана в .class-файл и доступна во время выполнения через Reflection.
- **@Target** – к какому типу данных можно подключить эту аннотацию:
  - **ElementType.METHOD** – метод;
  - **ElementType.FIELD** – поле;
  - **ElementType.CONSTRUCTOR** – конструктор;
  - **ElementType.PACKAGE** – пакет;
  - **ElementType.PARAMETER** – параметр;
  - **ElementType.TYPE** – тип;
  - **ElementType.LOCAL_VARIABLE** – локальная переменная и т.д.

Пример простой маркерной аннотации и ее применения:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MarkingAnnotation {
}
```

Получить аннотации полей, методов или классов можно с помощью методов **getAnnotations()** и **getDeclaredAnnotations()** у соответствующего класса – **Field, Method, Class**. Если известно имя нужной аннотации – применяем **getAnnotation()** и **getDeclaredAnnotation()**. Эти методы возвращают объекты типа **Annotation**.

Пример вывода в консоль списка методов с аннотациями **@MarkingAnnotation**.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MarkingAnnotation {
}

public class ReflectionApp {
    @MarkingAnnotation
    public void markedMethod() {
        System.out.println("Java");
    }

    public static void main(String[] args) {
        Method[] methods = MyClass.class.getDeclaredMethods();
        for (Method o : methods) {
            if(o.getAnnotation(MarkingAnnotation.class) != null) {
```

```
                System.out.println(o);
            }
        }
    }
}
```

К аннотациям можно добавлять параметры. Рассмотрим пример работы с такими аннотациями и получения их параметров. Слово **default** в объявлении поля **value** отвечает за установку значения по умолчанию:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AdvancedAnnotation {
    float value() default 5.0f;
}

public class MainClass {
    @AdvancedAnnotation(value = 20.0f)
    public void advAnnotatedMethod() {
        System.out.println("...");
    }

    public static void main(String[] args) {
        try {
            Method m = MainClass.class.getMethod("advAnnotatedMethod", null);
            AdvancedAnnotation annotation =
m.getAnnotation(AdvancedAnnotation.class);
            System.out.println("value: " + annotation.value());
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
    }
}

// Результат:
// value: 20.0
```

Взаимодействие с базами данных

Чтобы начать работу с базой данных, нужно установить систему управления (СУБД). Примеры таких систем – MySQL, Oracle, MS SQL, SQLite. Для тренировки работы с JDBC можно воспользоваться СУБД SQLite, которая имеет следующие особенности: хранит всю базу в одном файле, не требует установки, не поддерживает тип данных Data. Поддерживаемые типы данных: NULL – NULL-значение, INTEGER – целое знаковое, REAL – с плавающей точкой, TEXT – текст, строка (UTF-8), BLOB – бинарные данные.

Простые SQL запросы

Аббревиатура CRUD (Create/Read/Update/Delete) обозначает набор операций, которые можно производить над данными в базе. Они выполняются с помощью языка запросов SQL. Все команды языка регистронезависимы, могут быть разделены любым количеством пробелов и переносов строк.

CREATE

```
CREATE TABLE [имя таблицы] (
[имя колонки] [тип данных],
[имя колонки] [тип данных],
... );
```

Пример запроса:

```
CREATE TABLE IF NOT EXISTS students
(
  id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  name TEXT NOT NULL,
  score INTEGER NOT NULL
);
```

Здесь создается таблица **Students** с полями **id, name, score**.

NOT NULL означает, что поле всегда должно быть проинициализировано. СУБД следит за этим: если поле равно NULL, выдает ошибку.

PRIMARY KEY указывает, что поле имеет уникальное значение в этой таблице. В нашем примере это поле ID – мы хотим, чтобы у каждой записи был уникальный номер.

AUTOINCREMENT означает, что при каждом добавлении записи в таблицу ей автоматически будет присвоен ID на единицу больше предыдущего.

READ

Операция чтения данных из таблицы называется SELECT.

```
SELECT [список полей] FROM [имя таблицы] WHERE [условие];
```

Примеры запросов:

```
SELECT * FROM students;
SELECT * FROM students WHERE id > 3;
SELECT score FROM students WHERE id = 2;
```

Символ «*» означает, что мы хотим получить все поля таблицы. Иначе можно через запятую перечислить необходимые поля. Выражение WHERE не обязательно, но помогает извлекать только интересные для нас данные.

UPDATE

Это операция изменения уже присутствующих в таблице данных или добавления новых.

Добавление новых данных:

```
INSERT INTO [имя таблицы] ([список полей через запятую]) VALUES ([список значений через запятую]);
```

Изменение:

```
UPDATE [имя таблицы] SET [имя колонки]=[новое значение], [имя колонки]=[новое значение],.. WHERE [условие];
```

Примеры запросов:

```
INSERT INTO students (name, score) VALUES ('Bob', 80);
UPDATE students SET score = 90 WHERE name = 'Bob';
```

DELETE

Удаление данных из таблицы:

```
DELETE FROM [имя таблицы] WHERE [условие];
```

Пример запроса:

```
DELETE FROM accounts WHERE id = 0;
```

JDBC

Каждая СУБД разрабатывается конкретной компанией. Чтобы взаимодействовать с базой данных, производитель выпускает специальный драйвер – JDBC. С его помощью устанавливают соединение, изменяют данные, посылают запросы и обрабатывают их результаты.

Все основные сущности в JDBC API – это интерфейсы: Connection, Statement, PreparedStatement, CallableStatement, ResultSet, Driver, DatabaseMetaData. JDBC-драйвер конкретной базы данных предоставляет их реализации.

DriverManager – это синглтон, который содержит информацию о всех зарегистрированных драйверах. Метод **getConnection()** на основании параметра URL находит **java.sql.Driver** соответствующей базы данных и вызывает у него метод **connect()**.

Установка соединения

Драйвер JDBC можно скачать с сайта производителя СУБД. Он распространяется в виде .jar – библиотеки, которую необходимо подключить к проекту. Прежде чем использовать драйвер, его нужно зарегистрировать. Имя драйвера можно найти на сайте разработчиков.

```
// Для SQLite регистрация выглядит следующим образом
Class.forName("org.sqlite.JDBC");
// Для H2 Database - org.h2.Driver
// Для MySQL - com.mysql.jdbc.Driver
```

Исходный код реализации любого драйвера будет содержать статический блок инициализации:

```
static {
    try {
        java.sql.DriverManager.registerDriver(new Driver());
    } catch (SQLException e) {
        throw new RuntimeException("Can't register driver!");
    }
}
```

Вызов **Class.forName()** загружает класс и этим гарантирует выполнение статического блока инициализации, а значит и регистрацию драйвера в **DriverManager**. Чтобы указать, как найти базу данных, используется URL – специальная строка формата **[protocol]:[subprotocol]:[name]**:

```
// protocol: jdbc
// subprotocol: sqlite
// name: test.db
Connection conn = DriverManager.getConnection("jdbc:sqlite:mydatabase.db");
// ... Действия с БД ...
conn.close();
```

Объект **Connection** предоставляет доступ к базе данных. Опционально в него можно передать имя пользователя и пароль, если они установлены. После окончания работы с базой соединение необходимо закрыть методом **close()**.

Запросы в базу

Когда соединение с базой установлено, можно отправлять запросы. Для этого используется объект типа **Statement**, который умеет хранить SQL-команды. В базу можно отправить запрос на получение или изменение данных. В первом случае результатом будет объект **ResultSet**, который хранит результат. Во втором – количество строк таблицы, которые были изменены.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
int result = stmt.executeUpdate("INSERT INTO students (name, score) VALUES
("Bob", 80);");
```

Подготовленный запрос и пакетное выполнение запросов

Для выполнения множества похожих запросов наиболее эффективным и быстрым решением будет **PreparedStatement** – скомпилированная версия SQL-выражения.

В запросах можно использовать параметры: изменять его динамически в зависимости от входных данных. Параметр заменяется символом «?». Каждому параметру в запросе присваивается порядковый номер – индекс, начиная с 1. У объекта **PreparedStatement** есть методы, которые позволяют установить параметры. Нужно указать их позицию и значение:

```
PreparedStatement ps = conn.prepareStatement("SELECT * FROM Students WHERE ID =
?");
ps.setInt(1, 2);
ResultSet rs = ps.executeQuery();
```

PreparedStatement поддерживает пакетную (batch) отправку SQL-запросов, что значительно уменьшает трафик между клиентом и базой данных:

```
PreparedStatement ps = conn.prepareStatement("INSERT INTO students(name, score)
VALUES(?, ?);");
statement.setString(1, "Bob");
statement.setInt(2, 80);
statement.addBatch();
statement.setString(1, "John");
statement.setInt(2, 70);
statement.addBatch();
statement.executeBatch();
```

Обработка результатов

Результатом запроса **SELECT** в базу является таблица, которая сохраняется в объекте **ResultSet**. По строкам можно перемещаться вперед и назад. Для получения значений из определенной колонки текущей строки можно воспользоваться методами **get<Type>(<Param>)**, где **Type** – это тип извлекаемого значения, а **Param** – номер колонки (int) или имя колонки (String).

```
ResultSet rs = stmt.executeQuery();
while (rs.next()) {                    // Пока есть строки
  String name = rs.getString(2);       // Или rs.getString("Name");
}
rs.first();                            // Перейти к первой строке
rs.last();                             // Перейти к последней
rs.next();                             // Перейти к следующей
rs.previous();                         // Перейти к предыдущей
```

Закрытие ресурсов

На каждое соединение СУБД выделяет определенные ресурсы, количество которых ограничено. Поэтому после окончания работы с объектами соединения их нужно закрывать.

Транзакции в JDBC

Транзакция — это совокупность операций над базой данных, которые вместе образуют логически целостную процедуру, и могут быть либо выполнены все вместе, либо не будет выполнена ни одна из них. В простейшем случае транзакция состоит из одной операции. Транзакции являются одним из средств обеспечения согласованности базы данных, наряду с ограничениями целостности (constraints), накладываемыми на таблицы. Транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние.

По умолчанию каждый SQL-запрос выполняется внутри одной транзакции при выполнении **statement.execute()** и подобных методов. Чтобы открыть транзакцию для набора запросов, сначала необходимо установить флаг **autoCommit** у соединения в значение **false**, а затем пользоваться методами **commit()** и **rollback()**.

```
conn.setAutoCommit(false);
Statement st = conn.createStatement();
try {
    st.execute("INSERT INTO students (name, score) VALUES ('Bob', 50)");
    conn.commit();
} catch (SQLException e) {
    conn.rollback();
}
```

Подключение к PostgreSQL

Для начала работы с PostgreSQL необходимо скачать дистрибутив с сайте разработчика и установить его. После чего в pom.xml Maven проекта добавляется зависимость:

```
<dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
   <version>42.2.5</version>
</dependency>
```

Весь остальной код по базовым возможностям работы с базами данных (подключение, выполнение запросов, обработка результатов) остается тем же самым. В качестве url при открытии соединения используется следующий формат.

```
jdbc:postgresql://<database_host>:<port>/<database_name>
```

Порт указывать не обязательно. Подробно об особенностях и возможностях PostgreSQL будет рассказано на следующем занятии.

Практическое задание

1. Реализуйте возможность разметки класса с помощью набора ваших собственных аннотаций (@Table(title), @Column). Напишите обработчик аннотаций, который позволит по размеченному классу построить таблицу в базе данных.
2. * Второй обработчик аннотаций должен уметь добавлять объект размеченного класса в полученную таблицу.

   Замечание: Считаем что в проекте не связанных между собой сущностей, чтобы не продумывать логику их взаимодействия.

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.