



Reflection API и аннотации

Croc Java School



Reflection API

Предоставляет доступ к структуре типа в рантайме:

- определение класса объекта
- получение набора полей, конструкторов, методов, базовых классов
- динамическое конструирование объектов по названию класса
- получение и установка значений полей по названию
- динамический вызов методов по названию



Class<T>

Class<T> предоставляет доступ к метаданным о классе любого объекта.
Экземпляры создаются виртуальной машиной в момент загрузки классов в память.

```
Object value = 5;  
Class<?> type = value.getClass();  
System.out.println(type.getName());
```

```
out: java.lang.Integer
```



Экземпляр `Class<T>` можно получить по названию класса

```
Class<?> type = Class.forName("java.lang.Integer");
```

Если класс с указанным названием виртуальной машине неизвестен, метод выбрасывает исключение `ClassNotFoundException`.

Или через специальный литерал `.class`

```
Class<Integer> type = Integer.class;
```



Название класса

```
Class<Integer> type = Integer.class;  
  
type.getName();           // java.lang.Integer  
type.getSimpleName();     // Integer  
type.getPackageName();    // java.lang
```



Название класса

```
Function<Integer, Integer> mod2 = x -> x % 2;
```

```
Class<?> type = mod2.getClass();
```

```
type.getName();           // ru.croccode.test.Test$$Lambda$14/0x0000000800066840
```

```
type.getSimpleName();     // Test$$Lambda$14/0x0000000800066840
```

```
type.getPackageName();    // ru.croccode.test
```



Категории класса

```
type.isPrimitive();  
type.isArray();  
type.isEnum();  
type.isInterface();  
type.isMemberClass();  
type.isAnonymousClass();  
// ...
```



Поля, конструкторы и методы

```
Field[] getFields()
```

```
Field getField(String name)
```

```
Constructor<?>[] getConstructors()
```


```
Constructor<T> getConstructor(Class<?>... parameterTypes)
```

```
Method[] getMethods()
```

```
Method getMethod(String name, Class<?>... parameterTypes)
```

Методы предоставляют доступ к публичным полям, конструкторам и методам.

На практике



```
public class Parent {  
    private int a;  
  
    public void setA(int a) {  
        this.a = a;  
    }  
}
```

```
public class Child extends Parent {  
    private int b;  
    private int c;  
  
    public void setB(int b) {  
        this.b = b;  
    }  
}
```



Поля класса Child

```
Class<Child> type = Child.class;  
Field[] fields = type.getFields();
```

```
fields:  
<empty>
```



Конструкторы класса Child

```
Class<Child> type = Child.class;  
Constructor<?>[] constructors = type.getConstructors();
```

```
constructors:  
public ru.croccode.test.Child()
```



Методы класса Child

```
Class<Child> type = Child.class;  
Method[] methods = type.getMethods();
```

methods:

```
public void ru.croccode.test.Child.setB(int)  
public void ru.croccode.test.Parent.setA(int)  
+ публичные методы класса Object
```



Иерархия наследования

```
List<Class<?>> inheritancePath = new ArrayList<>();
Class<?> type = Child.class;
while (type != null) {
    inheritancePath.add(type);
    type = type.getSuperclass();
}
String path = inheritancePath.stream()
    .map(Class::getSimpleName)
    .collect(Collectors.joining(" < "));

out:
Child < Parent < Object
```



Реализуемые интерфейсы

```
Class<?> type = LinkedList.class;  
Class<?>[] interfaceTypes = type.getInterfaces();
```

```
interfaceTypes:  
interface java.util.List  
interface java.util.Deque  
interface java.lang.Cloneable  
interface java.io.Serializable
```



Параметры типа

```
Class<LinkedList> type = LinkedList.class;  
TypeVariable<Class<LinkedList>>[] typeVariables  
    = type.getTypeParameters();
```

```
typeVariables:
```

```
E
```

Для чего это все нужно?



Области применения Reflection API

- Фреймворки и библиотеки (ORM, DI)
- Инструменты отладки и тестирования
- Сериализация данных



Динамическое конструирование объектов

```
// получаем описание класса по названию
Class<?> type = Class.forName("ru.croccode.test.Child");

// запрашиваем конструктор без параметров
Constructor<?> constructor = type.getConstructor();

// конструируем новый объект
Object child = constructor.newInstance();
```



Динамический вызов методов

// запрашиваем метод с ожидаемым названием и типами аргументов

```
Method setA = type.getMethod("setA", int.class);
```

// вызываем метод

```
setA.invoke(child, 2021);
```



Многое может пойти не так

В Reflection API определено большое количество типов исключений. В примере с конструированием экземпляра и вызовом метода возможны следующие исключительные ситуации:

- `ClassNotFoundException` JVM неизвестен запрашиваемый класс
- `InstantiationException` экземпляр класса не может быть создан
- `NoSuchMethodException` указанный метод отсутствует в описании класса
- `InvocationTargetException` метод в процессе исполнения выбросил исключение
- `IllegalAccessException` доступ к полю/конструктору/методу ограничен областью видимости




Модификаторы

Информацию о модификаторах элемента языка можно получить с помощью метода

```
int getModifiers()
```

Этот метод определен в классах `Class<T>`, `Field`, `Constructor<T>`, `Method`.

Возвращаемое значение - флаги, упакованные в `int`. Интерпретировать флаги помогает вспомогательный класс `Modifier`.



```
Class<Child> type = Child.class;
Method setA = type.getMethod("setA", int.class);

int modifiers = setA.getModifiers();

Modifier.isPublic(modifiers);    // true
Modifier.isAbstract(modifiers); // false
```

Доступ к закрытым данным



Доступ к закрытым данным

```
public class Child extends Parent {  
    private int b;  
    private int c;  
  
    public void setB(int b) {  
        this.b = b;  
    }  
}
```

Поле c объявлено как закрытое и для него не определен соответствующий сеттер.



DeclaredX

Наряду с методами `getField(s)`, `getConstructor(s)`, `getMethod(s)` в `Class<T>` определены методы `getDeclaredField(s)`, `getDeclaredConstructor(s)`, `getDeclaredMethod(s)`.

Разница на примере методов запроса поля по названию:

`Field getField(String name)`

возвращает публичное поле, определенное в классе или любом его родителе

`Field getDeclaredField(String name)`

возвращает поле, определенное в текущем классе (без учета родителей), без учета ограничений модификаторов доступа

SOMETHING, SOMETHING



DARK SIDE



Запрос информации о private-поле

```
Class<Child> type = Child.class;  
Field c = type.getField("c");
```

Error: java.lang.NoSuchFieldException: c

```
Class<Child> type = Child.class;  
Field c = type.getDeclaredField("c");
```

```
c:  
private int ru.croccode.test.Child.c
```



Попробуем изменить значение private-поля

```
Child child = new Child();  
Class<?> type = child.getClass();
```

```
Field c = type.getDeclaredField("c");  
c.setInt(child, 13);
```

Error: java.lang.IllegalAccessException: class ru.croccode.test.Test cannot access a member of class ru.croccode.test.Child with modifiers "private"

Проверку модификаторов
доступа в Reflection API
МОЖНО ОТКЛЮЧИТЬ 🤯



setAccessible

```
void setAccessible(boolean flag)
```

A value of true indicates that the reflected object should suppress checks for Java language access control when it is used.



Этот код отработает корректно

```
Child child = new Child();  
Class<?> type = child.getClass();  
  
Field c = type.getDeclaredField("c");  
c.setAccessible(true);  
c.setInt(child, 13);
```

Несмотря на то, что поле `c` в классе `Child` закрыто модификатором `private`, его значение можно изменить через Reflection API.

Аннотации



Назначение аннотаций

Механизм предоставления метаданных о типе.

Аннотации служат для разметки кода в целях сопровождения его дополнительной информацией, но сами не определяют никакое поведение и не влияют на исполнение размеченного кода.



@Override

Знакомый пример - аннотация `@Override`. Она помечает метод как переопределяющий базовый и сигнализирует компилятору о необходимости выполнить дополнительные проверки на этапе компиляции.

На логику работы метода аннотация не влияет.

```
@Override  
public void close() throws IOException {  
    resource.close();  
}
```



Определение @Override

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

Ключевое слово `@interface` определяет тип аннотации.

`ElementType.METHOD` задает область применения аннотации - определение метода.

`RetentionPolicy.SOURCE` сигнализирует, что аннотацию нет необходимости записывать в метаданные класса и она может быть удалена на этапе компиляции.



Область применения

Аннотациями можно размечать классы, поля, методы, аргументы методов и не только.

```
public interface GeoService {  
  
    @Streaming  
    @Headers({"Cache-Control: no-cache"})  
    @GET("/{SOURCE_NAME}/terrain/{NORTHING}/{EASTING}")  
    public Response getElevationTerrainTile(  
        @Path(value = "SOURCE_NAME", encode = true) String sourceName,  
        @Path(value = "NORTHING") int northing,  
        @Path(value = "EASTING") int easting,  
        @Query("t") long ts);  
}
```



Неочевидные области применения аннотаций

Вызов конструктора

```
Child child = new @Annotated Child();
```

Приведение типа

```
Parent parent = (@Annotated Parent)child;
```

throws и implements

```
public class Resource implements @Annotated Closeable {  
    private RandomAccessFile f;
```

```
    @Override
```

```
    public void close() throws @Annotated IOException {  
        f.close();
```

```
    }
```

```
}
```



Кастомные аннотации

```
@Target({
    ElementType.TYPE,
    ElementType.FIELD,
    ElementType.METHOD,
    ElementType.PARAMETER
})
@Retention(RetentionPolicy.RUNTIME)
public @interface Beta {

    double completeness() default 0.0;
}
```



ElementType

TYPE

FIELD

METHOD

PARAMETER

CONSTRUCTOR

LOCAL_VARIABLE

ANNOTATION_TYPE

PACKAGE

TYPE_PARAMETER

TYPE_USE

MODULE



RetentionPolicy

SOURCE

CLASS

RUNTIME



Использование аннотации @Beta

```
public class Child extends Parent {  
    private int b;  
    private int c;  
  
    public void setB(int b) {  
        this.b = b;  
    }  
  
    @Beta(completeness = 0.5)  
    public void setC(int c) {  
        this.c = c;  
    }  
}
```



Стандартные аннотации

`@Override`

`@SuppressWarnings`

`@SafeVarargs`

`@Deprecated`

`@FunctionalInterface`



Стандартные аннотации для определения аннотаций

@Target

определяет область применения аннотации

@Retention

определяет время жизни (хранения) аннотации

@Inherited

определяет механизм наследования аннотаций для классов

@Repeatable

определяет возможность одновременного использования нескольких аннотаций одного типа

Reflection API и аннотации



Работа с аннотациями в рантайме (AnnotatedElement)


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

```
Annotation[] getAnnotations()
```

```
<T extends Annotation> T getAnnotation(Class<T> annotationClass)
```

```
Annotation[] getDeclaredAnnotations()
```

```
<T extends Annotation> T getDeclaredAnnotation(Class<T> annotationClass)
```



```
Child child = new Child();
Class<?> type = child.getClass();
Method setC = type.getMethod("setC", int.class);
if (setC.isAnnotationPresent(Beta.class)) {
    Beta beta = setC.getAnnotation(Beta.class);
    if (beta.completeness() <= 0.5) {
        System.out.println("WARNING: experimental beta method used");
    }
}
setC.invoke(child, 2022);
```

**Аннотации в совокупности с
Reflection API - мощный
инструмент для разработки
фреймворков и библиотек**

—



Spring Boot “Hello, World!”

```
@RestController
@EnableAutoConfiguration
public class MyApplication {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```