java.lang.Class

Конструктор java.lang.Class является одной из самых охраняемых сущностей в языке Java. В спецификации чётко сказано, что объекты типа Class может создавать только сама JVM и что нам тут делать нечего, но так ли это на самом деле?

Предлагаю погрузиться в глубины Reflection API (и не только) и выяснить, как там всё устроено и насколько трудно будет обойти имеющиеся ограничения.

Эксперимент я провожу на 64-битной JDK 1.8.0\_151 с дефолтными настройками. Про Java 9 будет в самом конце статьи.

Уровень 1. Простой

Начнём с самых наивных попыток и пойдём по нарастающей. Сперва посмотрим врагу в лицо:

private Class(ClassLoader loader)

{ classLoader = loader; }

Ничего особенного этот конструктор собой не представляет. Компилятор не делает для него никаких исключений, и в байткоде конструктор тоже присутствует. Поэтому попробуем поступить так же, как мы бы поступили с любым другим классом:

Constructor<Class> constructor = Class.class.getDeclaredConstructor(ClassLoader.class); constructor.setAccessible(true); Class<?> clazz = constructor.newInstance(ClassLoader.getSystemClassLoader());

Вполне ожидаемо данный код не будет работать и выдаст следующую ошибку:

Exception in thread "main" java.lang.SecurityException:

Cannot make a java.lang.Class constructor accessible

at java.lang.reflect.AccessibleObject.setAccessible0(...)

at java.lang.reflect.AccessibleObject.setAccessible(...)

at Sample.main(...)

С первой же попытки мы попали на первое предупреждение из метода setAccessible0. Оно захардкожено специально для конструктора класса java.lang.Class:

private static void setAccessible0(AccessibleObject obj, boolean flag)

throws SecurityException

{

if (obj instanceof Constructor && flag == true) {

Constructor<?> c = (Constructor<?>) obj;

if (c.getDeclaringClass() == Class.class) {

throw new SecurityException("Cannot make a java.lang.Class" +

" constructor accessible");

}

}

obj.override = flag;

}

Не проблема, ведь ключевой строкой в этом методе является последняя — установка поля override в значение true. Это легко сделать, используя грубую силу:

Field overrideConstructorField =

AccessibleObject.class.getDeclaredField("override");

overrideConstructorField.setAccessible(true);

overrideConstructorField.set(constructor, true);

Уровень 2. Посложнее

Естественно, установка флага override — это не единственное ограничение, но теперь мы можем хотя бы продвинуться чуть дальше в работе метода newInstance. Достаточно далеко, чтобы спланировать дальнейшие действия. В этот раз ошибка будет следующая:

Exception in thread "main" java.lang.InstantiationException:

Can not instantiate java.lang.Class

at sun.reflect.InstantiationExceptionConstructorAccessorImpl.newInstance(...)

at java.lang.reflect.Constructor.newInstance(...)

at Sample.main(...)

Нас занесло прямиком в класс пакета sun.reflect, а мы знаем, что основная магия должна происходить именно там. Самое время заглянуть в реализацию newInstance класса Constructor и узнать, как мы туда попали:

public T newInstance(Object ... initargs) throws ...

{

...

ConstructorAccessor ca = constructorAccessor; // read volatile

if (ca == null) {

ca = acquireConstructorAccessor();

}

@SuppressWarnings("unchecked")

T inst = (T) ca.newInstance(initargs);

return inst;

}

Из реализации становится понятно, что Constructor делегирует всю работу по инстанцированию другому объекту типа ConstructorAccessor. Он инициализируется ленивым образом и в дальнейшем не меняется. Внутренности метода acquireConstructorAccessor описывать не стану, скажу лишь, что в результате он приводит к вызову метода newConstructorAccessor объекта класса sun.reflect.ReflectionFactory. И именно для конструктора класса java.lang.Class (а ещё для абстрактных классов) данный метод возвращает объект InstantiationExceptionConstructorAccessorImpl. Он не умеет ничего инстанцировать, а только бросается исключениями на каждом обращении к нему. Всё это означает лишь одно: правильный ConstructorAccessor придётся инстанцировать самим.

Уровень 3. Нативный

Время узнать, каких вообще типов бывают объекты ConstructorAccessor (помимо описанного выше):

* BootstrapConstructorAccessorImpl:  
  используется для инстанцирования классов, которые сами являются реализацией ConstructorAccessor. Вероятно, спасает какой-то код от бесконечной рекурсии. Штука узкоспециализированная, трогать я её не буду;
* GeneratedConstructorAccessor:  
  самая интересная реализация, о которой я расскажу подробно, но позже;
* связка NativeConstructorAccessorImpl и DelegatingConstructorAccessorImpl:  
  то, что возвращается по умолчанию, и поэтому рассмотрится мною в первую очередь. DelegatingConstructorAccessorImpl попросту делегирует свою работу другому объекту, хранящемуся у него в поле. Плюс данного подхода в том, что он позволяет подменить реализацию на лету. Именно это на самом деле и происходит — NativeConstructorAccessorImpl для каждого конструктора отрабатывает максимум столько раз, сколько указано в системном свойстве sun.reflect.inflationThreshold (15 по умолчанию), после чего подменяется на GeneratedConstructorAccessor. Справедливости ради стоит добавить, что установка свойства sun.reflect.noInflation в значение "true" по сути сбрасывает inflationThreshhold в ноль, и NativeConstructorAccessorImpl перестаёт создаваться в принципе. По умолчанию это свойство имеет значение "false".

Итак, для самого обычного класса при самых обычных обстоятельствах мы бы получили объект  
NativeConstructorAccessorImpl, а значит, именно его и попробуем создать вручную:

Class<?> nativeCAClass = Class.forName("sun.reflect.NativeConstructorAccessorImpl"); Constructor<?> nativeCAConstructor = nativeCAClass.getDeclaredConstructor(Constructor.class); nativeCAConstructor.setAccessible(true); ConstructorAccessor constructorAccessor = (ConstructorAccessor) nativeCAConstructor.newInstance(constructor);

Здесь нет никаких подвохов: объект создаётся без лишних ограничений, и всё, что нам остаётся, так это с его помощью инстанцировать java.lang.Class:

Class<?> clazz = (Class<?>) constructorAccessor.newInstance(

new Object[]{ClassLoader.getSystemClassLoader()});

Но тут ждёт сюрприз:

#

# A fatal error has been detected by the Java Runtime Environment:

#

# SIGSEGV (0xb) at pc=0x00007f8698589ead, pid=20537, tid=0x00007f8699af3700

#

# JRE version: Java(TM) SE Runtime Environment (8.0\_151-b12) (build 1.8.0\_151-b12)

# ...

Кажется, JVM не ожидает от пользователя столь нелогичных действий, особенно после всех предупреждений. Тем не менее, данный результат можно по праву считать достижением — завалил JVM, ни разу не воспользовавшись классами пакета sun.misc!

Уровень 4. Магический

Нативный вызов не работает — значит, теперь нужно разобраться с GeneratedConstructorAccessor.

На самом деле, это не просто класс, а целое семейство классов. Для каждого конструктора в рантайме генерируется своя уникальная реализация. Именно поэтому в первую очередь используется нативная реализация: генерировать байткод и создавать из него класс дело затратное. Сам процесс генерации класса запрятан в метод generateConstructor класса sun.reflect.MethodAccessorGenerator. Вызвать его вручную не составит труда:

Class<?> methodAccessorGeneratorClass =

Class.forName("sun.reflect.MethodAccessorGenerator");

Constructor<?> methodAccessorGeneratorConstructor =

methodAccessorGeneratorClass.getDeclaredConstructor();

methodAccessorGeneratorConstructor.setAccessible(true);

Object methodAccessorGenerator =

methodAccessorGeneratorConstructor.newInstance();

Method generateConstructor = methodAccessorGeneratorClass

.getDeclaredMethod("generateConstructor",

Class.class, Class[].class, Class[].class, int.class);

generateConstructor.setAccessible(true);

ConstructorAccessor constructorAccessor = (ConstructorAccessor)

generateConstructor.invoke(methodAccessorGenerator,

constructor.getDeclaringClass(), constructor.getParameterTypes(),

constructor.getExceptionTypes(), constructor.getModifiers());

Как и в случае с NativeConstructorAccessorImpl, тут нет подводных камней — данный код отработает и сделает ровно то, что от него ждут. Но давайте задумаемся на минутку: ну сгенерировали мы какой-то класс, откуда у него возьмутся права на вызов приватного конструктора? Такого быть не должно, поэтому мы просто обязаны сдампить сгенерированный класс и изучить его код. Сделать это несложно — встаём отладчиком в метод generateConstructor и в нужный момент дампим нужный нам массив байт в файл. Декомпилированная его версия выглядит следующим образом (после переименования переменных):

public class GeneratedConstructorAccessor1 extends ConstructorAccessorImpl {

public Object newInstance(Object[] args) throws InvocationTargetException {

Class clazz;

ClassLoader classLoader;

try {

clazz = new Class;

if (args.length != 1) {

throw new IllegalArgumentException();

}

classLoader = (ClassLoader) args[0];

} catch (NullPointerException | ClassCastException e) {

throw new IllegalArgumentException(e.toString());

}

try {

clazz.<init>(classLoader);

return clazz;

} catch (Throwable e) {

throw new InvocationTargetException(e);

}

}

}

Такой код, естественно, обратно не скомпилируется, и этому есть две причины:

* вызов new Class без скобочек. Он соответствует инструкции NEW, которая выделяет память под объект, но конструктор у него не вызывает;
* вызов clazz.<init>(classLoader) — это как раз вызов конструктора, который в таком явном виде в языке Java невозможен.

Данные инструкции разнесены для того, чтобы находиться в разных try-блоках. Почему сделано именно так, я не знаю. Вероятно, это был единственный способ обрабатывать исключения так, чтобы они полностью соответствовали спецификации языка.

Если закрыть глаза на нетипичную обработку исключений, то во всём остальном данный класс абсолютно нормален, но всё ещё непонятно, откуда у него вдруг права на вызов приватных конструкторов. Оказывается, всё дело в суперклассе:

abstract class ConstructorAccessorImpl extends MagicAccessorImpl

implements ConstructorAccessor {

public abstract Object newInstance(Object[] args)

throws InstantiationException, IllegalArgumentException,

InvocationTargetException;

}

В JVM есть известный костыль под названием sun.reflect.MagicAccessorImpl. Всякий его наследник обладает неограниченным доступом к любым приватным данным любых классов. Это именно то, что нужно! Раз класс магический, то он поможет получить инстанс java.lang.Class. Проверяем:

Class<?> clazz = (Class<?>) constructorAccessor.newInstance(

new Object[]{ClassLoader.getSystemClassLoader()});

и опять получаем исключение:

Exception in thread "main" java.lang.IllegalAccessError: java.lang.Class at sun.reflect.GeneratedConstructorAccessor1.newInstance(...)

at Sample.main(...)

Вот это уже действительно интересно. Судя по всему, обещанной магии не произошло. Или я ошибаюсь?

Стоит рассмотреть ошибку внимательнее и сравнить её с тем, как должен себя вести метод newInstance. Будь проблема в строке clazz.<init>(classLoader), мы бы получили InvocationTargetException. На деле же имеем IllegalAccessError, то есть до вызова конструктора дело не дошло. С ошибкой отработала инструкция NEW, не позволив выделить память под объект java.lang.Class. Здесь наши полномочия всё, окончены.

Уровень 5. Современный

Reflection не помог решить проблему. Может быть, дело в том, что Reflection старый и слабый, и вместо него стоит использовать молодой и сильный MethodHandles? Думаю, да. Как минимум, стоит попробовать.

И как только я решил, что Reflection не нужен, он тут же пригодился. MethodHandles — это, конечно, хорошо, но с помощью него принято получать лишь те данные, к которым есть доступ. А если понадобился приватный конструктор, то придётся выкручиваться по старинке.

Итак, нам нужен MethodHandles.Lookup с приватным доступом к классу java.lang.Class. На этот случай есть очень подходящий конструктор:

private Lookup(Class<?> lookupClass, int allowedModes)

{. . .}

Вызовем его:

Constructor<MethodHandles.Lookup> lookupConstructor =

MethodHandles.Lookup.class.getDeclaredConstructor(Class.class, int.class);

lookupConstructor.setAccessible(true);

MethodHandles.Lookup lookup = lookupConstructor

.newInstance(Class.class, MethodHandles.Lookup.PRIVATE);

Получив lookup, можно получить объект MethodHandle, соответствующий требуемому нам конструктору:

MethodHandle handle = lookup.findConstructor(Class.class,

MethodType.methodType(Class.class, ClassLoader.class));

После запуска этого метода я был откровенно удивлён — lookup делает вид, что конструктора вообще не существует, хотя он точно присутствует в классе!

Exception in thread "main" java.lang.NoSuchMethodException:

no such constructor:

java.lang.Class.<init>(ClassLoader)Class/newInvokeSpecial

at java.lang.invoke.MemberName.makeAccessException(...)

at java.lang.invoke.MemberName$Factory.resolveOrFail(...)

at java.lang.invoke.MethodHandles$Lookup.resolveOrFail(...)

at java.lang.invoke.MethodHandles$Lookup.findConstructor(...)

at Sample.main(Sample.java:59) Caused by: java.lang.NoSuchFieldError: method resolution failed

at java.lang.invoke.MethodHandleNatives.resolve(...)

at java.lang.invoke.MemberName$Factory.resolve(...)

at java.lang.invoke.MemberName$Factory.resolveOrFail(...)

... 3 more

Странно то, что причина исключения — NoSuchFieldError. Загадочно...

В этот раз ошибся именно я, но далеко не сразу это понял. Спецификация findConstructor требует, чтобы тип возвращаемого значения был void, несмотря на то, что у результата MethodType будет ровно таким, как я описал (всё потому, что метод <init>, отвечающий за конструктор, действительно возвращает void по историческим причинам).  
Так или иначе, путаницы можно избежать, ведь у lookup есть второй метод для получения конструктора, и он называется unreflectConstructor:

MethodHandle handle = lookup.unreflectConstructor(constructor);

Данный метод уж точно корректно отработает и вернёт тот handle, который должен.

Момент истины. Запускаем метод инстанцирования:

Class<?> clazz = (Class<?>) handle.

invoke(ClassLoader.getSystemClassLoader());

Думаю, вы уже догадались, что ничего хорошего не произойдёт, но давайте хоть глянем на ошибку. Сейчас это что-то новенькое:

Exception in thread "main" java.lang.IllegalAccessException: java.lang.Class

at sun.misc.Unsafe.allocateInstance(...)

at java.lang.invoke.DirectMethodHandle.allocateInstance(...)

at java.lang.invoke.LambdaForm$DMH/925858445.newInvokeSpecial\_L\_L(...)

at java.lang.invoke.LambdaForm$MH/523429237.invoke\_MT(...)

at Sample.main(...)

По умолчанию stacktrace отображается укороченным, поэтому я добавил  
-XX:+UnlockDiagnosticVMOptions -XX:+ShowHiddenFrames в параметры запуска. Так становится проще понять, в какое странное место мы попали.

Не буду углубляться в то, какие классы генерирует MethodHandles, да это и не принципиально. Важно совсем другое — мы наконец-то докопались до использования sun.misc.Unsafe, и даже он не в силах создать объект java.lang.Class.

Метод allocaeInstance используется в тех местах, где нужно создать объект, но не вызывать у него конструктор. Такое бывает полезно, например, при десериализации объектов. По сути, это та же инструкция NEW, но не обременённая проверками прав доступа. *Почти* не обременённая, как мы только что увидели.

Раз даже Unsafe не смог, мне остаётся лишь прийти к печальному заключению: аллоцировать новый объект java.lang.Class невозможно. Интересно выходит — думал, что запрещён конструктор, а запрещена аллокация! Попробуем это дело обойти.

Уровень 6. Небезопасный

Предлагаю создать пустой объект и взглянуть, из чего же он состоит. Для этого возьмём Unsafe и аллоцируем новенький java.lang.Object:

Field theUnsafeField = Unsafe.class.getDeclaredField("theUnsafe");

theUnsafeField.setAccessible(true);

Unsafe unsafe = (Unsafe) theUnsafeField.get(null);

Object object = unsafe.allocateInstance(Object.class);

На текущей JVM результатом будет область памяти в 12 байт, выглядящая вот так:

https://habrastorage.org/webt/s6/sn/rm/s6snrm3cgyugvhlntmosw0uamly.png

То, что вы здесь видите, это "заголовок объекта". По большому счёту, он состоит из двух частей — 8 байт markword, которые нас не интересуют, и 4 байта classword, которые важны.

Каким образом JVM узнаёт класс объекта? Она делает это путём чтения области classword, которая хранит указатель на внутреннюю структуру JVM, описывающую класс. Значит если в данное место записать другое значение, то и класс объекта изменится!

Дальнейший код очень, очень плохой, никогда так не делайте:

System.out.println(object.getClass());

unsafe.putInt(object, 8L, unsafe.getInt(Object.class, 8L)); System.out.println(object.getClass());

Мы прочитали classword объекта Object.class и записали его в classword объекта object. Результат работы следующий:

class java.lang.Object

class java.lang.Class

С натяжкой можно считать, что java.lang.Class мы аллоцировали. Мы молодцы! Теперь надо вызвать конструктор. Вы можете смеяться, но сейчас мы будем с помощью ASM генерировать класс, умеющий вызывать нужный конструктор. Естественно, при этом нужно унаследоваться от MagicAccessorImpl.

Так начинается создание класса (константы импортированы статически, так короче):

ClassWriter cw = new ClassWriter(COMPUTE\_FRAMES | COMPUTE\_MAXS);

cw.visit(V1\_8, ACC\_PUBLIC, "sun/reflect/MyConstructorInvocator", null, "sun/reflect/MagicAccessorImpl", null);

Так ему создаётся конструктор:

MethodVisitor init = cw.visitMethod(ACC\_PUBLIC, "<init>", "()V", null, null); init.visitCode();

init.visitVarInsn(ALOAD, 0);

init.visitMethodInsn(INVOKESPECIAL, "sun/reflect/MagicAccessorImpl", "<init>", "()V", false);

init.visitInsn(RETURN);

init.visitMaxs(-1, -1);

init.visitEnd();

А так создаётся метод void construct(Class<?>, ClassLoader), который внутри себя вызывает конструктор у объекта Class<?>:

MethodVisitor construct = cw.visitMethod(ACC\_PUBLIC, "construct", "(Ljava/lang/Class;Ljava/lang/ClassLoader;)V", null, null);

construct.visitCode();

construct.visitVarInsn(ALOAD, 1);

construct.visitVarInsn(ALOAD, 2);

construct.visitMethodInsn(INVOKESPECIAL, "java/lang/Class", "<init>", "(Ljava/lang/ClassLoader;)V", false);

construct.visitInsn(RETURN);

construct.visitMaxs(-1, -1);

construct.visitEnd();

Класс готов. Осталось загрузить, инстанцировать и вызвать нужный метод:

byte[] bytes = cw.toByteArray();

Class<?> myCustomInvocator = unsafe.defineClass(null, bytes, 0, bytes.length, ClassLoader.getSystemClassLoader(), null);

Object ci = myCustomInvocator.newInstance(); Method constructMethod = myCustomInvocator.getDeclaredMethod("construct", Class.class, ClassLoader.class);

Class<?> clazz = (Class<?>) object;

constructMethod.invoke(ci, clazz, ClassLoader.getSystemClassLoader());

И это работает! Точнее так: повезло, что работает. Можно проверить, запустив следующий код:

System.out.println(clazz.getClassLoader());

Вывод будет таким:

sun.misc.Launcher$AppClassLoader@18b4aac2

О том, в какую область памяти записался этот ClassLoader и откуда потом прочитался, я тактично умолчу. И, как ожидалось, вызов практически любого другого метода на данном объекте приводит к немедленному краху JVM. А в остальном — цель выполнена!

Что там в Java 9?

В Java 9 всё почти так же. Можно проделать все те же действия, но с несколькими оговорками:

* в параметры компилятора надо добавить --add-exports java.base/jdk.internal.reflect=sample (где sample — это имя вашего модуля);
* в параметры запуска надо добавить:  
  --add-opens java.base/jdk.internal.reflect=sample  
  --add-opens java.base/java.lang=sample  
  --add-opens java.base/java.lang.reflect=sample  
  --add-opens java.base/java.lang.invoke=sample  
  --add-opens java.base/jdk.internal.reflect=java.base
* в зависимости модуля надо добавить requires jdk.unsupported;
* у конструктора java.lang.Class поменялась сигнатура, надо учесть.

Так же стоит учесть, что sun.reflect перенесли в jdk.internal.reflect и что класс MyConstructorInvocator теперь надо грузить тем же загрузчиком, что у MagicAccessorImpl.  
ClassLoader.getSystemClassLoader() уже не сработает, у него не будет доступа.

Ещё исправили странную багу с NoSuchFieldError: теперь на его месте NoSuchMethodError, который там и должен быть. Мелочь, но приятно.

В целом, в Java 9 нужно намного сильнее постараться, чтобы выстрелить себе в ногу, даже если именно это и является главной целью. Думаю, это и к лучшему.

Выводы:

* при желании в Java можно творить абсолютно безумные вещи, это забавно;
* Reflection API не так уж и сложно устроен;
* MagicAccessorImpl может не всё;
* sun.misc.Unsafe может не всё, но почти;
* Java 9 ещё сильнее старается вас обезопасить.

Не стоит слишком серьёзно воспринимать всё описанное. Сама по себе задача инстанцирования java.lang.Class совершенно бессмысленна. Здесь важны знания, полученные в процессе её решения.