

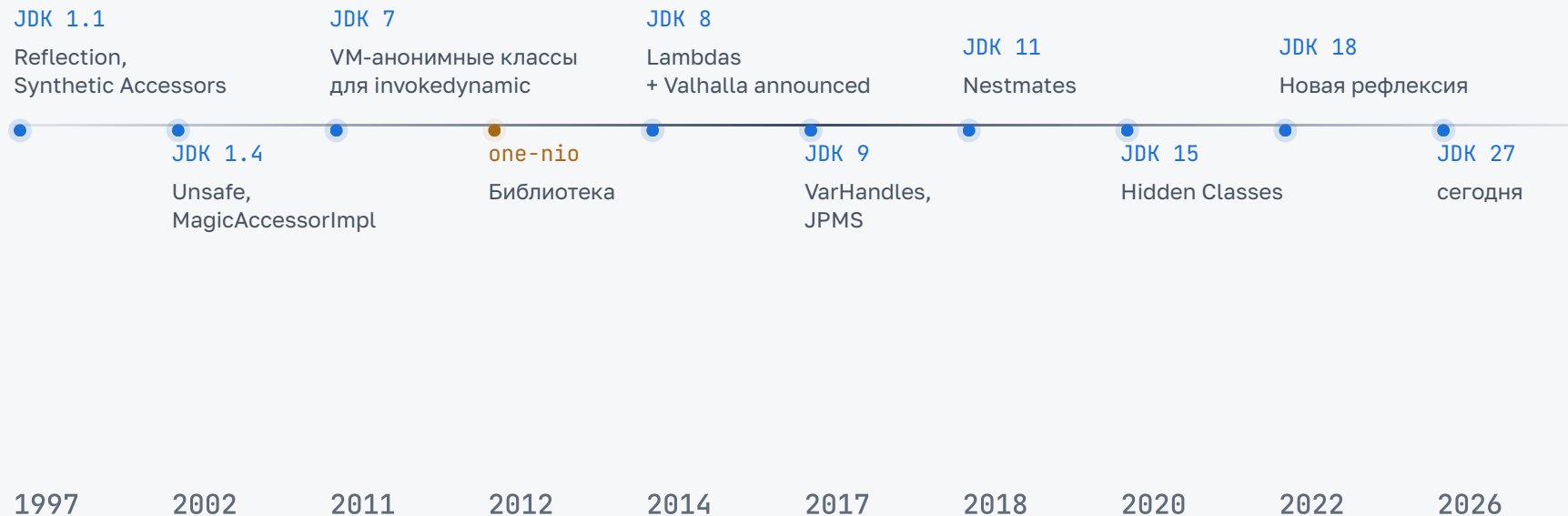
JDK 1.1 → 26

Под капотом JVM: археология внутренних и публичных механизмов

Михаил Богданов

Разработчик, Эксперт

Общий таймлайн появления обсуждаемых фич



Что важно знать о сериализации one-nio

01

Поддерживает автоматический вывод схем по Java-объекту и эволюцию схем

02

Два режима сериализации:

- со схемой объекта
- без схемы: предполагается, что она уже есть на принимающей стороне, либо может быть явно запрошена

03

Сериализаторы классов генерируются на рантайме

04

Два режима работы фреймворка:

- исходный, 2012 г. — работает с JDK 8–23
- новый, 2025 г. — работает с JDK 9–25+

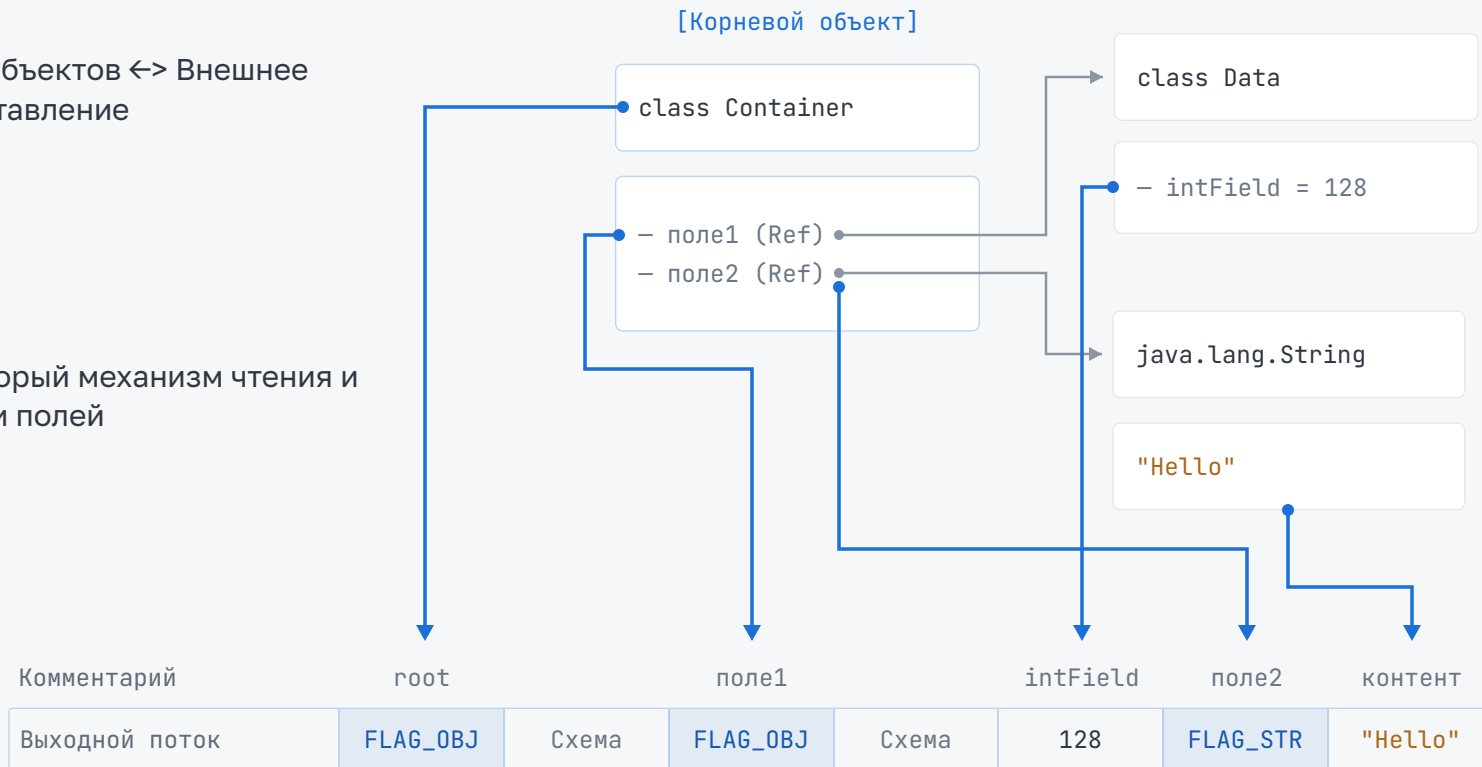
ЧАСТЬ 1

О сериализации в общем и одной важной подзадаче



Сериализация

- 1 Граф объектов ↔ Внешнее представление
- 2 Некоторый механизм чтения и записи полей

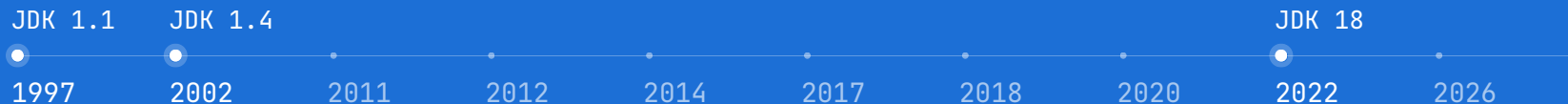


Некоторый механизм чтения и записи полей

**Какие есть варианты для
доступа к состоянию объекта?**

ЧАСТЬ 2

Давайте немного поговорим о рефлексии в JVM



Первая реализация рефлексии в JVM. Методы

Рефлексивный вызов метода идет через JNI вызов `NativeMethodAccessorImpl.invoke0`, что требует:

- переключение контекста выполнения
- маршаллинг данных
- потеря оптимизаций JIT-компилятора
- создание нового Java frame, перенос аргументов из native и переход обратно в песочницу
- много подкапотных проверок

Очень медленно

Подробнее: «Дорог ли native метод? „Секретное“ расширение JNI» — habr.com/ru/articles/222997

Механика ускорения

Что если генерировать Java-байткод для рефлексивного доступа на рантайме, так чтобы этот байткод имел доступ к запрашиваемым методам?

Сразу уходит пачка проблем

Переключение контекста выполнения

Маршаллинг данных

Потеря оптимизаций JIT-компилятора

...

ЭКСПЕРИМЕНТЫ ПОКАЗАЛИ

Генерация класса на рантайме и его загрузка — дорогая операция, но при этом сгенерированный код в десятки раз быстрее нативных вызовов

Механизм инфляции JDK 1.4

01

Первые вызовы делегируются в JNI — дешево стартуем, но медленно работаем.

02

При достижении порога `threshold = 15` вызовов генерируется байт-код.

03

Дальнейшие вызовы идут через сгенерированный код — максимально быстро, плюс дополнительно срабатывают JIT-оптимизации.

Немного археологии в исходниках

ReflectionFactory, JDK 6

"Inflation" mechanism. Loading bytecodes to implement `Method.invoke()` and `Constructor.newInstance()` currently costs **3-4x more** than an invocation via native code for the first invocation (though subsequent invocations have been benchmarked to be **over 20x faster**).

Управление инфляцией

ReflectionFactory.java

```
public class ReflectionFactory {  
    private static boolean noInflation = false;  
    private static int inflationThreshold = 15;  
    ...  
}
```

Можно переопределить свойства, включив или выключив сам механизм и поменяв трешхолд через системные проперти:

`sun.reflect.noInflation`

`sun.reflect.inflationThreshold`

Такая схема рефлексии (JNI + генерация) работала по умолчанию до JDK 17 и была убрана в JDK 22

Бенчмарк сравнения рефлексивных вызовов

JDK 17
Linux x86_64
CPU i7-1185G7

Вызов	ops/ms
Прямой нерефлексивный вызов	3 491 227
Рефлексия: чистая генерация <code>-Djdk.reflect.noInflation=true</code>	407 536
Рефлексия: чистый JNI <code>-Dsun.reflect.inflationThreshold=Integer.MAX_VALUE</code>	13 994

Разница в 30 раз между генерацией и JNI

github.com/max-kammerer/jvm-benchmarks

Стоп, а как сгенерированный код обходит проверки доступа?

В чем его отличие от обычных классов?

Все дело в Магии...

MagicAccessorImpl (MAI) JDK 1.4

JavaDoc for MagicAccessorImpl:

All subclasses of this class are "magically" granted access by the VM to otherwise inaccessible fields and methods of other classes.

+ другие возможности

MAGICACCESSORIMPL

При использовании напрямую (без рефлексии)
получаем **скорость прямого доступа** к полям и
методам

Генерируем сериализаторы на рантайме, наследуя их от MagicAccessorImpl

MAI стал одним из фундаментов работы фреймворка сериализации one-pio

sun.reflect.MagicAccessorImpl

MyClass.java

```
package myPackage;

public class MyClass {
    private String data;

    private void sayHello() {
        System.out.println(data);
    }
}
```

sun.reflect.MagicAccessorImpl

MyClass.java

```
package myPackage;

public class MyClass {
    private String data;

    private void sayHello() {
        System.out.println(data);
    }
}
```

ConfMagic.java

```
package sun.reflect;
import myPackage.MyClass;

public class ConfMagic extends MagicAccessorImpl {

    public void doSomeWork(MyClass myClass) {
        myClass.data = "Hello Conf";
        myClass.sayHello();
    }
}
```

FINAL ПОЛЯ ПРИ ДЕСЕРИАЛИЗАЦИИ

**Всё хорошо, но загвоздка с final-ными полями,
которые не можем инициализировать с помощью
MagicAccessorImpl**

sun.misc.Unsafe JDK 1.4

Unsafe в JVM — это внутренний API, предоставляющий низкоуровневые операции: прямое управление памятью (off-heap), атомарные инструкции, манипуляции с полями классов и создание объектов в обход конструкторов.

Небезопасен, может привести к падению JVM и не рекомендуется для обычного использования.

Используется / использовался в:

Netty

Cassandra

Hazelcast

Kafka

Hadoop

Spark

Lucene

Ignite

TensorFlow

...

Unsafe

MyClass.java

```
public class MyClass {  
    private final String data;  
  
    public MyClass(String data) {  
        this.data = data;  
    }  
  
    public void sayHello() {  
        System.out.println(data);  
    }  
}
```

Unsafe

MyClass.java

```
public class MyClass {  
    private final String data;  
  
    public MyClass(String data) {  
        this.data = data;  
    }  
  
    public void sayHello() {  
        System.out.println(data);  
    }  
}
```

достаём Unsafe

```
public Unsafe getUnsafe() throws ... {  
    Field field = Unsafe.class  
        .getDeclaredField("theUnsafe");  
    field.setAccessible(true);  
    return (sun.misc.Unsafe) field.get(null);  
}
```

Unsafe

MyClass.java

```
public class MyClass {  
    private final String data;  
  
    public MyClass(String data) {  
        this.data = data;  
    }  
  
    public void sayHello() {  
        System.out.println(data);  
    }  
}
```

достаём Unsafe

```
public Unsafe getUnsafe() throws ... {  
    Field field = Unsafe.class  
        .getDeclaredField("theUnsafe");  
    field.setAccessible(true);  
    return (sun.misc.Unsafe) field.get(null);  
}
```

создаём объект без конструктора

```
public MyClass doSomeWork(Class<MyClass> cls) throws ... {  
    MyClass data = (MyClass) unsafe.allocateInstance(cls);  
    ...  
}
```

Unsafe

MyClass.java

```
public class MyClass {
    private final String data;

    public MyClass(String data) {
        this.data = data;
    }

    public void sayHello() {
        System.out.println(data);
    }
}
```

достаём Unsafe

```
public Unsafe getUnsafe() throws ... {
    Field field = Unsafe.class
        .getDeclaredField("theUnsafe");
    field.setAccessible(true);
    return (sun.misc.Unsafe) field.get(null);
}
```

смещение поля

```
public MyClass doSomeWork(Class<MyClass> cls) throws ... {
    MyClass data = (MyClass) unsafe.allocateInstance(cls);
    long fieldOffset = unsafe.objectFieldOffset(
        cls.getDeclaredField("data"));
    ...
}
```

Unsafe

MyClass.java

```
public class MyClass {
    private final String data;

    public MyClass(String data) {
        this.data = data;
    }

    public void sayHello() {
        System.out.println(data);
    }
}
```

достаём Unsafe

```
public Unsafe getUnsafe() throws ... {
    Field field = Unsafe.class
        .getDeclaredField("theUnsafe");
    field.setAccessible(true);
    return (sun.misc.Unsafe) field.get(null);
}
```

пишем в final-поле

```
public MyClass doSomeWork(Class<MyClass> cls) throws ... {
    MyClass data = (MyClass) unsafe.allocateInstance(cls);
    long fieldOffset = unsafe.objectFieldOffset(
        cls.getDeclaredField("data"));

    unsafe.putObject(data, fieldOffset, "New Value");
    return data;
}
```

Unsafe

MyClass.java

```
public class MyClass {  
    private final String data;  
  
    public MyClass(String data) {  
        this.data = data;  
    }  
  
    public void sayHello() {  
        System.out.println(data);  
    }  
}
```

Что если положить что-то другое? Например,
`new Object()`?

пишем в final-поле

```
public MyClass doSomeWork(Class<MyClass> cls) throws ... {  
    MyClass data = (MyClass) unsafe.allocateInstance(cls);  
    long fieldOffset = unsafe.objectFieldOffset(  
        cls.getDeclaredField("data"));  
  
    unsafe.putObject(data, fieldOffset, new Object());  
    return data;  
}
```

MagicAccessorImpl и **Unsafe** стали
фундаментом библиотеки сериализации...

Что принёс JDK 24+

MagicAccessorImpl и его аналоги были убраны в **JDK 23–24**

Также Unsafe постепенно выпиливается из JDK. На замену идут [VarHandles](#) (JDK 9+) и [Foreign Function & Memory API](#) (Java 22+)

Ожидаемые изменения

JDK 18

Появился новый рефлексен по умолчанию

JDK 22

Обе предыдущие реализации были убраны

ЧАСТЬ 3

Каким путем пошла библиотека?

VarHandles



VarHandles

Низкоуровневое API для работы с переменными — полями классов, элементами массивов...

Из документации:

A VarHandle is a dynamically strongly typed reference to a variable, or to a parametrically-defined family of variables, including static fields, non-static fields, array elements, or components of an off-heap data structure. Access to such variables is supported under various access modes, including plain read/write access, volatile read/write access, and compare-and-set.

VarHandles

MyClass.java

```
public class MyClass {  
    private String data;  
    ...  
}
```

VarHandles

MyClass.java

```
public class MyClass {  
    private String data;  
    ...  
}
```

получаем Lookup

```
VarHandle getVarHandle() throws Exception {  
    MethodHandles.Lookup lookup =  
        MethodHandles.privateLookupIn(  
            MyClass.class,  
            MethodHandles.lookup()  
        );  
    return ...;  
}
```

VarHandles

MyClass.java

```
public class MyClass {  
    private String data;  
    ...  
}
```

получаем VarHandle

```
VarHandle getVarHandle() throws Exception {  
    MethodHandles.Lookup lookup =  
        MethodHandles.privateLookupIn(  
            MyClass.class,  
            MethodHandles.lookup()  
        );  
    return lookup.findVarHandle(  
        MyClass.class, "data", String.class);  
}
```

VarHandles

Main.java

```
public static void main(String[] args) throws Exception {  
    MyClass instance = new MyClass("Hello");  
    VarHandle varHandle = getVarHandle();  
  
    → String value = (String) varHandle.get(instance);  
    System.out.println(value);  
  
    → varHandle.set(instance, value + " Updated");  
    System.out.println((String) varHandle.get(instance));  
}
```

VarHandles, производительность

JDK 21
Linux x86_64
CPU i7-1185G7

Benchmark

thrpt, ops/ms

`String field; // прямой доступ`

3 388 772

`static final VarHandle`

3 347 614

`static VarHandle`

389 449

`final VarHandle`

403 657

`VarHandle`

409 669

Делаем статическим `final`-полем, чтобы сработали оптимизации JVM

jvm-benchmarks

Изменения в генерируемых классах

TestData.java

```
class TestData implements Serializable {  
    private String nonFinalField;  
    private final String finalField;  
    ...  
}
```

байткод сериализатора

```
public final read(Lone/nio/serial/DataStream;)  
    Ljava/lang/Object;  
    ...  
ALOAD 1  
INVOKEVIRTUAL DataStream.readObject ()Ljava/lang/Object;  
CHECKCAST java/lang/String
```

Изменения в генерируемых классах

старый вариант · MagicAccessorImpl

```
PUTFIELD TestData.nonFinalField: Ljava/lang/String;
```

новый вариант · VarHandle

```
GETSTATIC .../Delegate0_TestData.$nonFinalField  
    : Ljava/lang/invoke/VarHandle;  
... // переупорядочиваем аргументы на стеке  
INVOKEVIRTUAL .../VarHandle.set (...)V
```

Unsafe для записи final-полей

TestData.java

```
class TestData implements Serializable {  
    private String nonFinalField;  
    private final String finalField;  
    ...  
}
```

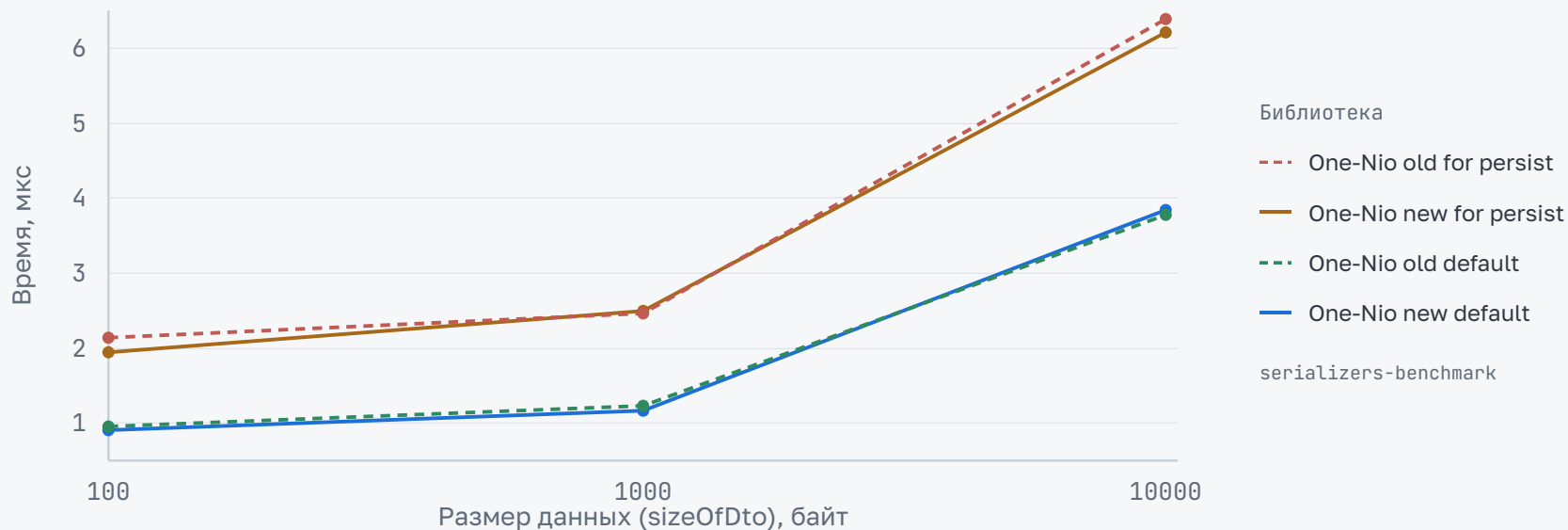
инициализация final по-прежнему через Unsafe

```
ALOAD 1  
INVOKEVIRTUAL DataStream.readObject ()Ljava/lang/Object;  
CHECKCAST java/lang/String  
  
LDC 16L // смещение final поля  
INVOKESTATIC JavaInternals.putObject (... )V
```

Паритет старого и нового режима

Полный цикл: сериализация + десериализация · one-nio 2.2

OpenJDK 21
Linux x86_64
Intel Ultra 5 235H



Tradeoff нового режима на базе VarHandles

Генерируем то же количество классов, что и в исходном режиме, но теперь еще и инициализируем VarHandle'ы в статическом инициализаторе классов `<clinit>`.

Как результат получаем дополнительно время на инициализацию классов + время на прогрев, чтобы достичь паритета с прямым доступом.

Не решает проблему с `final`-полями — необходимо по-прежнему использовать `Unsafe`.

**Какие еще могут
быть альтернативы?**

И еще немного о рефлексии

JEP 416

Reimplement Core Reflection with Method Handles

Рефлексия на базе MethodHandles, ставшая механизмом по умолчанию в JDK 18

Старые варианты были окончательно убраны в JDK 22

JEP 416 • Reimplement Core Reflection

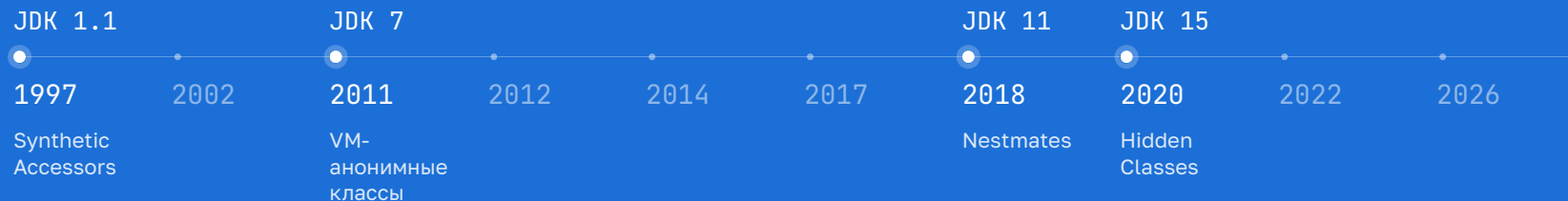
Доступ	Reflection, JDK 17	Reflection, JDK 21	VarHandles, JDK 21
directFieldAccess	3 471 983	3 455 030	3 528 086
final static Field	594 156	1 652 825	3 514 334
static Field	531 546	346 108	410 862
final Field	479 901	338 279	422 750
Field	468 899	339 136	406 005

NESTMATES + HIDDEN CLASSES

Сериализатор — как nestmate для сериализуемого класса

ЧАСТЬ 4

Давайте посмотрим на историю появления Nestmates + Hidden Classes



Язык vs Виртуальная машина: вложенные и анонимные классы

В Java 1.1 появились вложенные и анонимные классы.

Имеют доступ к private-членам outer-класса

MyOuter.java

```
public class MyOuter {  
    private Object state;  
  
    private void test(Object state) {  
        System.out.println(state);  
    };  
  
    public class MyInner {  
        public void doWork() {  
            test(state);  
        }  
    }  
}
```

Язык vs Виртуальная машина: вложенные и анонимные классы

JVM про них ничего не знает

Реализованы по сути на уровне компилятора:

- Плоская структура: классы выносились на top-level с композитным именем
- Генерация дополнительных скрытых методов для доступа к private-членам

MyOuter.java

```
public class MyOuter {  
    private Object state;  
  
    private void test(Object state) {  
        System.out.println(state);  
    };  
  
    public class MyInner {  
        public void doWork() {  
            test(state);  
        }  
    }  
}
```

Синтетические аксессоры

MyOuter.class

```
public class MyOuter {  
    → private LObject; state  
  
    // package local visibility  
    static synthetic access$000(LMyOuter;)L...Object;  
    ...  
    → GETFIELD MyOuter.state : L...Object;  
    ...  
  
    → private test(L...Object;)V  
    ...  
    // package local visibility  
    static synthetic access$100(LMyOuter;L...Object;)V  
    ...  
    → INVOKESPECIAL MyOuter.test (L...Object;)V  
    ...  
}
```

MyOuter\$MyInner.class

```
public class MyOuter$MyInner {  
    // test(state);  
    public doWork()V  
    ...  
    → INVOKESTATIC MyOuter.access$000(...) ...  
    → INVOKESTATIC MyOuter.access$100 (...) ...  
    → RETURN  
}
```

Синтетические аксессуары, tradeoffs

01 Лишний байткод

02 Нарушение инкапсуляции

03 Засорение Reflection API (getDeclaredMethods)

04 Раздувание MetaSpace и ConstantPool

05 Лишняя работа для JIT-а и тулов

JEP 181 • Новый подход Nestmates JDK 11+

- 1 Вводится понятие **гнезда** — контекста контроля доступа на уровне JVM
- 2 Все члены гнезда имеют полноправный доступ к **private**-сущностям друг друга
- 3 Верхний класс — **хозяин гнезда** (Nest Host), вложенные записываются как Nest Members

Introduce nests, an access-control context that aligns with the existing notion of nested types in the Java programming language. Nests allow classes that are logically part of the same code entity, but which are compiled to distinct class files, to access each other's private members without the need for compilers to insert accessibility-broadening bridge methods.

Nestmates в байткоде

исходник

```
public class MyOuter {  
    private Object state;  
  
    private void test(Object state) {  
        System.out.println(state);  
    };  
  
    public class MyInner {  
        public void doWork() {  
            test(state);  
        }  
    }  
}
```

MyOuter.class

хозяин гнезда

```
public class MyOuter {  
    NESTMEMBER MyOuter$MyInner  
    ...  
}
```

MyOuter\$MyInner.class

```
public class MyOuter$MyInner {  
    NESTHOST MyOuter  
    public doWork()V  
        GETFIELD MyOuter.state : L...Object;  
        ...  
        INVOKEVIRTUAL MyOuter.test (L...Object;)V  
}
```

Nestmates не только в Javac

Уходят те же издержки:

~~Лишний байткод~~

~~Нарушение инкапсуляции~~

~~Засорение Reflection API~~

~~Раздувание MetaSpace и ConstantPool~~

~~Лишняя работа для JIT-а и тулов~~

Java [11](#)

2018

Groovy [5](#)

2025

Scala

[Issue 22292](#)

github.com/scala/scala3/issues/22292

Kotlin

[KT-27182](#)

youtrack.jetbrains.com/issue/KT-27182

ГДЕ МЫ

Аксессоры и nestmates разобрали — дальше hidden classes

JDK 1.1



1997

Synthetic
Accessors ✓

JDK 7



2011

VM-
анонимные
классы

JDK 11



2018

Nestmates ✓

JDK 15



2020

Hidden
Classes

Hidden Classes

JEP 371 · JDK 15

- не появились вдруг
- стали расширением внутреннего механизма VM-анонимных классов

VM-анонимные классы JDK 1.7

Классы, появившиеся в JDK 1.7 для поддержки динамических языков — механизм `invokedynamic`.

Особенность: легковесность

Не прибиты к класслоадеру, могут легко быть выгружены как обычные объекты, когда на них больше нет ссылок

-
- нельзя найти через загрузчики классов
 - нет легального имени, на которое можно сослаться в байткоде

Создание

```
Unsafe.defineAnonymousClass(Host.class, byte[] bytecode, ...)
```

[Host.class](#) — класс, через который шли все проверки доступа.

Автоматом получаем доступ к его состоянию.

Поддержка лямбд в OpenJDK 8 через VM-анонимные классы

исходник

```
Runnable lambda = () → {  
    System.out.println("Hello");  
};
```

Лямбды становятся привилегированными компаньонами класса-контейнера благодаря механике анонимных классов

JDK 8 · InnerClassLambdaMetafactory

```
/**  
 * Generate a class file which implements  
 * the functional interface,  
 * define and return the class.  
 */  
private Class<?> spinInnerClass()  
    throws LambdaConversionException {  
    ...  
    return UNSAFE.defineAnonymousClass(  
        targetClass, classBytes, null);  
}
```

Лямбды в OpenJDK

Lambda2Demo.java

```
public class Lambda2Demo {  
    private String name = "Hello";  
  
    public void test() {  
        Runnable lambda = () → {  
            System.out.println(name);  
        };  
        lambda.run();  
    }  
  
    public static void main(  
        String[] a) {  
        new Lambda2Demo().test();  
    }  
}
```

точка вызова

```
→ INVOKEDYNAMIC run(LLambda2Demo;)  
    Ljava/lang/Runnable; [  
        // handle kind 0x6 : INVOKESTATIC  
        java/lang/invoke/LambdaMetafactory  
            .metafactory(...);  
        // arguments:  
        ()V,  
        // handle kind 0x7 : INVOKESPECIAL  
        Lambda2Demo.lambda$test$0()V,  
        ()V  
    ]
```

тело лямбды

```
public class Lambda2Demo {  
    private synthetic lambda$test$0()V  
  
    GETSTATIC java/lang/System.out  
        : Ljava/io/PrintStream;  
    ALOAD 0  
    GETFIELD Lambda2Demo.name  
        : Ljava/lang/String;  
    INVOKEVIRTUAL java/io/PrintStream  
        .println  
        (Ljava/lang/String;)V  
    RETURN  
}
```

Линковка — при первом проходе: генерация класса и CallSite
Один раз на инструкцию

Пример с лямбдой

LambdaDemo.java

```
Runnable lambda = () → {};  
System.out.println(lambda.getClass().getName());
```

Результат: `LambdaDemo$$Lambda$1/0x00000000840060840`

Недостатки VM-анонимных классов

Были плохо интегрированы в экосистему и рантайм JVM: невидимы не только для загрузчиков классов, но и для самой JVM.

Constant pool patching для передачи параметров и контекста выполнения

Hidden Classes

JEP 371 · JDK 15

Introduce hidden classes, which are classes that cannot be used directly by the bytecode of other classes. Hidden classes are intended for use by frameworks that generate classes at run time and use them indirectly, via reflection. A hidden class may be defined as a member of an access control nest, and may be unloaded independently of other classes.

Основные свойства:

- Не привязаны к загрузчику классов, их невозможно найти через `ClassLoader`
- Могут быть выгружены как обычные объекты, как только на них не осталось ссылок
- Нельзя сослаться в байткоде
- Единственный способ использования объектов Hidden Class'ов — косвенно, через объект его Class'a

Hidden Classes: дополнительные свойства JEP 371

`Class::getCanonicalName()`

возвращает null

`Class::getName()`

возвращает «не бинарное» имя

Финальные поля действительно final — нельзя поменять даже через рефлексен и другие методы

Не инструментируются агентами, включая JVM TI

Синергия Hidden Classes и Nestmates

```
lookup.defineHiddenClass(bytecode, true, ClassOption.NESTMATE).lookupClass();
```

Опция **NESTMATE** заставляет JVM внедрить этот свежесозданный скрытый класс в гнездо класса Target. Он динамически становится полноправным «членом семьи».

Расширение Nestmates на рантайме

Будем получать доступ к MyClass

MyClass.java

```
package sample;

public class MyClass {
    private String data;

    public MyClass(String data) {
        this.data = data;
    }

    private void sayHello() {
        System.out.println("Hello, " + data);
    }
}
```

И работать с Hidden-классом через интерфейс аксессора

Accessor.java

```
package sample;

public interface Accessor {
    void invokeSayHello(MyClass target);
}
```

Расширение Nestmates на рантайме

Main.java

```
public static void main(String[] args) throws Throwable {
    MyClass instance = new MyClass("Conference!");

    // 1. Генерируем байткод для нашего скрытого класса
    byte[] bytecode = generateAccessorBytecode(MyClass.class);

    // 2. Создаем Lookup объект
    Lookup lookup = MethodHandles.privateLookupIn(
        MyClass.class, MethodHandles.lookup()
    );

    // 3. Определяем класс
    Lookup hiddenLookup = lookup.defineHiddenClass(
        bytecode,
        true, // Инициализировать класс
        Lookup.ClassOption.NESTMATE
    );
    ...
}
```

Расширение Nestmates на рантайме

Main.java

```
// 3. Определяем и инициализируем класс
Lookup hiddenLookup = lookup.defineHiddenClass(
    bytecode, true, Lookup.ClassOption.NESTMATE
);

// 4. Получаем класс и создаем его экземпляр
Class<?> hiddenClass = hiddenLookup.lookupClass();
Accessor accessor = (Accessor) hiddenClass
    .getConstructor().newInstance();

// 5. Вызываем! Никакого оверхеда на рефлексию,
//    чистый JIT-оптимизированный вызов.
accessor.invokeSayHello(instance);
```

Hello, Conference!

Как выглядит генерация через Class File API

generateAccessorBytecode

```
public static byte[] generateAccessorBytecode(Class<?> targetClass) {  
    ""  
    // 1. Создание билдера  
    return ClassFile.of().build(hiddenClassDesc, classBuilder → {  
        // 2. Указываем базовый интерфейс  
        classBuilder.withInterfaceSymbols(ClassDesc.of("sample.Accessor"));  
        var myClassDesc = ClassDesc.ofDescriptor(targetClass.descriptorString());  
  
        // 3. Генерируем наш метод invokeSayHello  
        classBuilder.withMethodBody("invokeSayHello",  
            MethodTypeDesc.of(ConstantDescs.CD_void, myClassDesc),  
            ClassFile.ACC_PUBLIC,  
            codeBuilder → { // 4. Создание тела метода  
                codeBuilder.aload(1) // 4.1 Загрузка параметра target  
                // 4.2 Вызов метода private sayHello через INVOKEVIRTUAL  
                .invokevirtual(myClassDesc, "sayHello", MethodTypeDesc.ofDescriptor("()V"))  
                .return_();  
            });  
    });  
}
```

Передача контекста в hidden-класс

classData

```
// 1. Контекст передаётся при определении hidden-класса
var lookup = MethodHandles.lookup()
    .defineHiddenClassWithClassData(bytecode, new MyData(), true);

// 2. Внутри hidden-класса достаём его обратно
var myData = MethodHandles.classData(
    MethodHandles.lookup(), ConstantDescs.DEFAULT_NAME, MyData.class);
```

Штатный способ — динамические константы в байткоде

JEP 309

JDK 11

LDC Dynamic #0:_:Lsample/MyData; // инструкция загрузки динамической константы

BootstrapMethods:

0: REF_invokeStatic java/lang/invoke/MethodHandles.classData:

(Ljava/lang/invoke/MethodHandles\$Lookup;Ljava/lang/String;Ljava/lang/Class;)Ljava/lang/Object;

Несовместимость с механизмом инфляции

Нет имени, на которое можно сослаться в байткоде $\Rightarrow$ нет возможности сгенерировать MAI

Проблему решила новая рефлексия на базе MethodHandle, появившаяся в JDK 18

Не только в самой JVM

Google Guice

АОР машинерия и обход рефлексии

Byte Buddy

инструментация лямбд

Apache Lucene

компиляция JS в Bytecode

Elasticsearch

специализация констант в запросах

ЧАСТЬ 5

Не все так гладко. Об ограничениях JPMS



Об ограничениях JPMS. Работа с VarHandles

Мы не можем просто получить доступ к `private/protected` полям в другом модуле — для этого модуль должен открывать свои пакеты для доступа на runtime.

К счастью, все безымянные и автоматические модули открывают все свои типы.

Для именованных модулей есть два варианта:

`opens`

открывать пакеты в декларации модуля

`--add-opens`

флаг запуска JVM

Об ограничениях JPMS. Работа с Hidden Class'ами

Попытка добавить класс в nestmate'ы `BigInteger`

```
Exception in thread "main" java.lang.IllegalAccessException: java.math.BigInteger/sample.HiddenClassDemo does not  
have full privilege access
```

Требуется полный доступ к модулю

Об ограничениях JPMS. Работа с Hidden Class'ами

Это можно обойти, вернув нужный Lookup из целевого модуля, чтобы через него можно было сделать injection.

Снаружи это сделать нельзя.

Об ограничениях JPMS. Работа с Hidden Class'ами

Возможно решение для модулей, которые вы контролируете — передавать лукап из него наружу.

ЧАСТЬ 6

Заключение



Генерация превращает данные в код

КОМПРОМИСС

JVM активно использует генерацию — лямбды, MethodHandle, рефлексия MAI.

За это приходится платить.

Зато потом из коробки на этом коде работают все плюшки JVM.

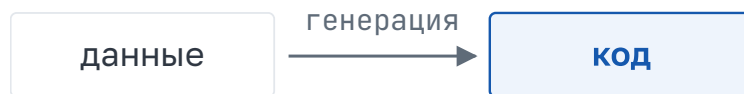
лямбды

MethodHandle

рефлексия MAI

ВЫИГРЫШ

Но это же можно использовать себе на благо: генерация запекает "тяжелые данные" в код, а дальше JIT творит чудеса.

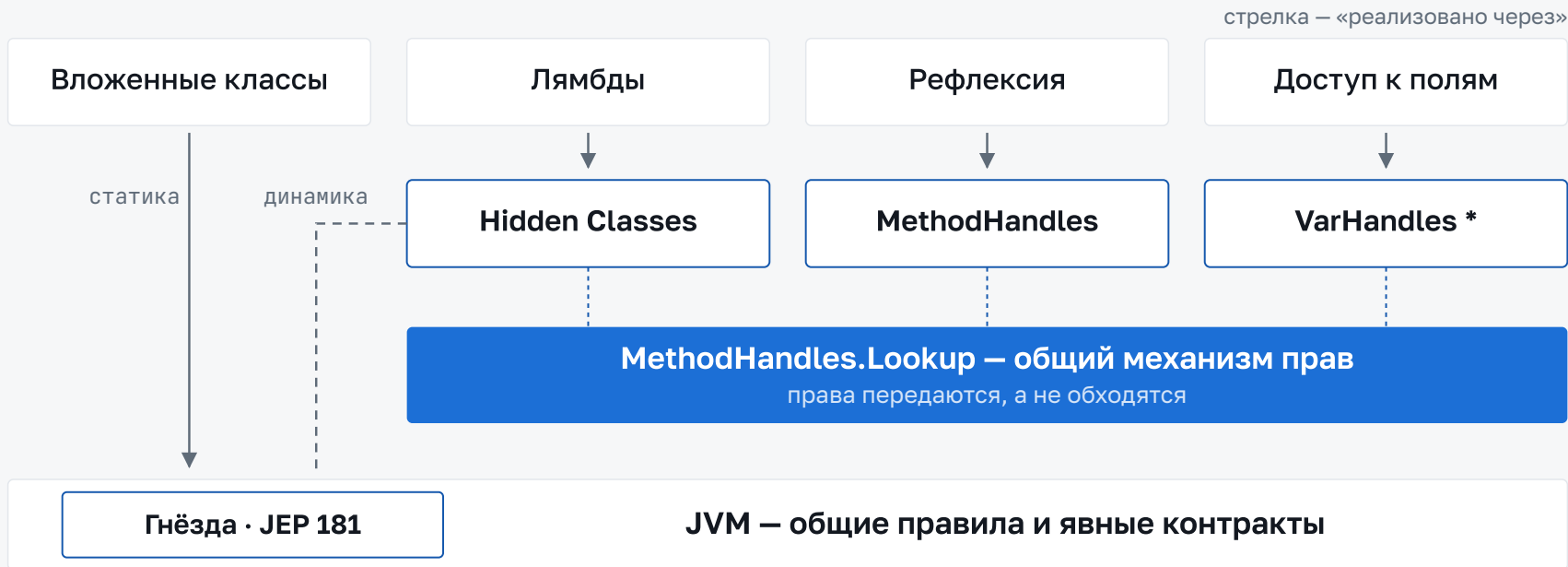


как в примере с one-nio, Elasticsearch, Apache Lucene

Было: разные механизмы со своими ограничениями, плохо интегрировались



Стало: те же задачи — механизмы платформы



Один фундамент, один раздатчик прав

правила держит VM — права раздаёт Lookup

* запись final — Unsafe и ReflectionFactory (JDK 24+)

ИТОГ

30 лет эволюции платформы

от хакерских фиш к выстраиванию устойчивого и предсказуемого фундамента

JDK 1.1	JDK 1.4	JDK 7	JDK 8	JDK 9	JDK 11	JDK 15	JDK 18	JDK 27
•	•	•	•	•	•	•	•	•
1997	2002	2011	2014	2017	2018	2020	2022	2026
рефлексия	MAI	invokedynamic	лямбды	JPMS	nestmates	hidden classes	новая рефлексия	

В поисках баланса, взгляд со стороны

- 01 Большие системы — это набор условностей-компромиссов: возможностей и ограничений
- 02 Системы меняются, сталкиваясь с новыми запросами-вызовами, становясь при этом и более сложными
- 03 Рост требует новых подходов и перебалансировки системы, ограничения становятся опорой для следующих шагов

В какое ограничение уперлась ваша система?

О чем это ограничение для вас?

Спасибо за внимание!

Михаил Богданов

Разработчик, Эксперт

ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ



Хабр

КОНТАКТЫ



LinkedIn