ОТРАЖЕНИЕ, СВЯЗЫВАНИЕ И ДИНАМИКА

как jvm училась оптимизировать непредсказуемое

ПИШИЙ ЕГОР

java разработчик в PVS-Studio

- люблю копаться во **внутрянке java**
- начинал с **плагинов для minecraft**
- продолжаю **в статическом анализе**
- пока ничем **не закончил**



ПЛАН

1. динамический вызов

ПЛАН

1. динамический вызов
2. **рефлексия**

ПЛАН

1. динамический вызов
2. рефлексия
3. **method handle**

ПЛАН

1. динамический вызов
2. рефлексия
3. method handle
4. **как jit оптимизирует method handle**

ПЛАН

1. динамический вызов
2. рефлексия
3. method handle
4. как jit оптимизирует method handle
5. **где method handle используется внутри java**

ДИНАМИЧЕСКИЙ ВЫЗОВ

ДИНАМИЧЕСКИЙ ВЫЗОВ

что?

выбор метода для вызова
во время работы
приложения

ДИНАМИЧЕСКИЙ ВЫЗОВ

что?

выбор метода для вызова во время работы приложения

зачем?

гибкие и расширяемые приложения.
например: di, orm, плагины

ДИНАМИЧЕСКИЙ ВЫЗОВ

что?

выбор метода для вызова
во время работы
приложения

зачем?

гибкие и расширяемые
приложения.
например: di, orm, плагины

как?

узнаем дальше

НА ПРИМЕРЕ

НА ПРИМЕРЕ

события в minecraft

как работают

```
1   class EventListener implements Listener {
2
3
4
5
6
7
8
9
10
11
12
13  }
```

НА ПРИМЕРЕ

события в minecraft
как работают

добавляем метод
принимающий событие

```
class EventListener implements Listener {


  public void onBlockBreak(BlockBreakEvent event) {
    // ....
  }



}
```

НА ПРИМЕРЕ

события в minecraft
как работают

добавляем метод
принимающий событие

вешаем аннотацию
флаг для использования дальше

```
1   class EventListener implements Listener {
2
3     @EventHandler
4     public void onBlockBreak(BlockBreakEvent event) {
5       // ....
6     }
7
8
9
10
11
12
13  }
```

НА ПРИМЕРЕ

регистрируем

просто передавая объект

```
1   class EventListener implements Listener {
2
3     @EventHandler
4     public void onBlockBreak(BlockBreakEvent event) {
5       // ....
6     }
7
8
9
10    void register(PluginManager pluginManager) {
11       pluginManager.register(this);
12    }
13  }
```

НА ПРИМЕРЕ

регистрируем
просто передавая объект

менеджер работает динамически
ищет методы и регистрирует их

```
1   class EventListener implements Listener {
2
3     @EventHandler
4     public void onBlockBreak(BlockBreakEvent event) {
5       // ....
6     }
7
8
9
10    void register(PluginManager pluginManager) {
11       pluginManager.register(this);
12    }
13  }
```

НА ПРИМЕРЕ

регистрируем
просто передавая объект

менеджер работает динамически
ищет методы и регистрирует их

по аннотации
точка входа для менеджера — флаг

```
1   class EventListener implements Listener {
2
3     @EventHandler
4     public void onBlockBreak(BlockBreakEvent event) {
5       // ....
6     }
7
8
9
10    void register(PluginManager pluginManager) {
11       pluginManager.register(this);
12    }
13  }
```

ТО ЕСТЬ

ТО ЕСТЬ

динамический вызов
нужен для гибкости

ИТЕРАЦИЯ ПЕРВАЯ : РЕФЛЕКСИЯ

REFLECTION API

REFLECTION API

встроенный механизм

с 1997 года входит стандартный пакет jdk и развивается по сей день

REFLECTION API

встроенный механизм

с 1997 года входит стандартный пакет jdk и развивается по сей день

позволяет получать метаданные

можно получать и анализировать все члены класса в процессе работы приложения

REFLECTION API

встроенный механизм

с 1997 года входит стандартный пакет jdk и развивается по сей день

позволяет получать метаданные

можно получать и анализировать все члены класса в процессе работы приложения

предоставляет динамический доступ к членам класса

позволяет вызывать, изменять и создавать объекты без статической привязки к ним

```
1   void main() {
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16  }
```

напишем код

больше кода богу кода

```
1   void main() {
2       class Service {
3
4
5       }
6
7
8
9
10
11
12
13
14
15
16  }
```

напишем код

больше кода богу кода

создадим класс

нам понадобится простенький маленький класс

```
1   void main() {
2       class Service {
3           void grind() { }
4           void chill() { }
5       }
6
7
8
9
10
11
12
13
14
15
16  }
```

напишем код

больше кода богу кода

создадим класс

нам понадобится простенький маленький класс

с двумя методами

такие простые, как и сам класс: либо гринд, либо чилл

```
1   void main() {
2       class Service {
3           void grind() { }
4           void chill() { }
5       }
6       var service = new Service();
7
8
9
10
11
12
13
14
15
16  }
```

возьмём экземпляр

неориентированный объект
в объектно-ориентированной java

```
1   void main() {
2       class Service {
3           void grind() { }
4           void chill() { }
5       }
6       var service = new Service();
7
8       String targetMethod =
9               ThreadLocalRandom.current().nextBoolean() ?
10                      "grind" :
11                      "chill";
12
13
14
15
16  }
```

возьмём экземпляр

неориентированный объект
в объектно-ориентированной java

возьмём имя метода

подбросим монетку и выберем имя метода, который нужно вызвать

```
1   void main() {
2       class Service {
3           void grind() { }
4           void chill() { }
5       }
6       var service = new Service();
7
8       String targetMethod =
9               ThreadLocalRandom.current().nextBoolean() ?
10                      "grind" :
11                      "chill";
12
13      Service.class
14
15
16  }
```

берём класс

с помощью рефлексии получаем класс в виде объекта

```
1   void main() {
2       class Service {
3           void grind() { }
4           void chill() { }
5       }
6       var service = new Service();
7
8       String targetMethod =
9               ThreadLocalRandom.current().nextBoolean() ?
10                      "grind" :
11                      "chill";
12
13      Service.class
14              .getDeclaredMethod(targetMethod)
15
16  }
```

берём класс

с помощью рефлексии получаем класс в виде объекта

ищем нужный метод

также через рефлексию получаем метод, который собираемся вызвать

```
1   void main() {
2       class Service {
3           void grind() { }
4           void chill() { }
5       }
6       var service = new Service();
7
8       String targetMethod =
9               ThreadLocalRandom.current().nextBoolean() ?
10                      "grind" :
11                      "chill";
12
13      Service.class
14              .getDeclaredMethod(targetMethod)
15              .invoke(service);
16  }
```

берём класс

с помощью рефлексии получаем класс в виде объекта

ищем метод

также через рефлексию получаем метод, который собираемся вызвать

вызываем метод

успешно нашли метод, теперь вызываем его на объекте

запомнили



REFLECTION API — ПЛЮСЫ

REFLECTION API — ПЛЮСЫ

гибкий

можно загружать классы,
методы и поля по имени,
даже если они приватные

REFLECTION API — ПЛЮСЫ

гибкий

можно загружать классы,
методы и поля по имени,
даже если они приватные

информативный

позволяет узнать структуру
класса во время выполнения

REFLECTION API — ПЛЮСЫ

гибкий

можно загружать классы, методы и поля по имени, даже если они приватные

информативный

позволяет узнать структуру класса во время выполнения

простой

интуитивный api
без лишних слоёв абстракции

REFLECTION API — ПЛЮСЫ

гибкий

можно загружать классы, методы и поля по имени, даже если они приватные

информативный

позволяет узнать структуру класса во время выполнения

простой

интуитивный api без лишних слоёв абстракции

универсальный

подходит для любого класса или объекта

REFLECTION API — МИНУСЫ

REFLECTION API — МИНУСЫ

накладные расходы

REFLECTION API — МИНУСЫ

накладные расходы

проверка доступа

можно загружать классы,
методы и поля по имени,
даже если они приватные

REFLECTION API — МИНУСЫ

накладные расходы

проверка доступа

можно загружать классы, методы и поля по имени, даже если они приватные

проверка аргументов

позволяет узнать структуру класса во время выполнения

REFLECTION API — МИНУСЫ

накладные расходы

проверка доступа

можно загружать классы, методы и поля по имени, даже если они приватные

проверка аргументов

позволяет узнать структуру класса во время выполнения

приведение типов

интуитивный api без лишних слоёв абстракции

REFLECTION API — МИНУСЫ

REFLECTION API — МИНУСЫ

чёрный ящик для jit-компилятора

JIT-КОМПИЛЯТОР

что такое

JIT-КОМПИЛЯТОР

что такое



JIT-КОМПИЛЯТОР

что такое



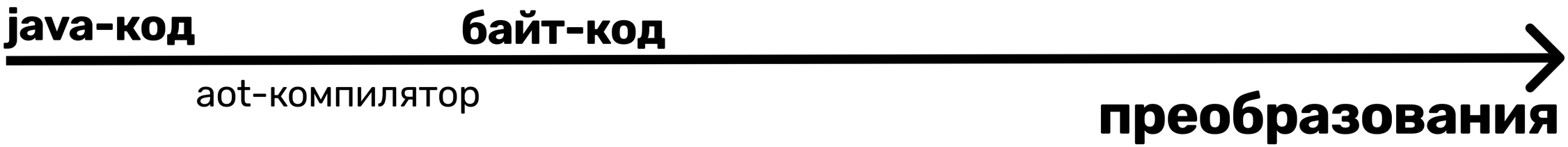
JIT-компилятор

что такое



JIT-компилятор

что такое



JIT-компилятор

что такое



JIT-компилятор

что такое



JIT-КОМПИЛЯТОР

что такое

JIT-КОМПИЛЯТОР

что такое

смотрит на пути выполнения

JIT-КОМПИЛЯТОР

что такое

смотрит на пути выполнения

собирает статистику

JIT-КОМПИЛЯТОР

что такое

смотрит на пути выполнения

собирает статистику

пытается предугадать

JIT-КОМПИЛЯТОР

что такое

смотрит на пути выполнения

собирает статистику

пытается предугадать

создаёт специализированный нативный код

ТО ЕСТЬ

то есть

jit-компилятор нужен,
чтобы динамически создавать
оптимизированный нативный код

REFLECTION API — МИНУСЫ

чёрный ящик для jit-компилятора

REFLECTION API — МИНУСЫ

чёрный ящик для jit-компилятора

```
1   Service.class
2       .getMethod("grind")
3       .invoke(service);
```

REFLECTION API — МИНУСЫ

чёрный ящик для jit-компилятора

```
1   Service.class
2       .getMethod("grind")
3       .invoke(service);
```

REFLECTION API — МИНУСЫ

чёрный ящик для jit-компилятора

```
1   Service.class
2       .getMethod("grind")
3       .invoke(service);
```

.invoke()

вызов вызова

говорим java, что хотим вызвать метод

REFLECTION API — МИНУСЫ

чёрный ящик для jit-компилятора

```
1   Service.class
2       .getMethod("grind")
3       .invoke(service);
```

.invoke()

plain java

уходим в java-код

здесь происходят проверки доступа и первичные преобразования типов

REFLECTION API — МИНУСЫ

чёрный ящик для jit-компилятора

```
1   Service.class
2       .getMethod("grind")
3       .invoke(service);
```

.invoke()

plain java

native magic

лезем в нативный код

здесь ещё раз происходят проверки доступа и окончательные преобразования

REFLECTION API — МИНУСЫ

чёрный ящик для jit-компилятора

```
1   Service.class
2       .getMethod("grind")
3       .invoke(service);
```

.invoke()

plain java

native magic

service.grind()

выполняем нужный метод

наконец происходит вызов требуемого метода

REFLECTION API — МИНУСЫ

чёрный ящик для jit-компилятора

```
1   Service.class
2       .getMethod("grind")
3       .invoke(service);
```

.invoke()

plain java

native magic

service.grind()

здесь jit теряется

ТО ЕСТЬ

то есть

рефлексия даёт нужную гибкость, но с тяжёлой накладной и без jit-оптимизаций

REFLECTION API — МИНУСЫ

как решить

REFLECTION API — МИНУСЫ

как решить

НИКАК.

REFLECTION API — МИНУСЫ

как решить

НИКАК.

только использовать другой api

ИТЕРАЦИЯ ВТОРАЯ: METHOD HANDLE

METHOD HANDLE API

METHOD HANDLE API

появился сильно позже

активно развивается и используется во внутренних механизмах с java 7

METHOD HANDLE API

появился сильно позже

активно развивается и используется во внутренних механизмах с java 7

прозрачен для jit

изначально разрабатывался с целью обеспечения прозрачности для jit-оптимизаций

METHOD HANDLE API

появился сильно позже

активно развивается и используется во внутренних механизмах с java 7

прозрачен для jit

изначально разрабатывался с целью обеспечения прозрачности для jit-оптимизаций

никакой метаинформации

не знает ничего о классах, методах или полях в них. реализует только вызовы

METHOD HANDLE API

появился сильно позже

активно развивается и используется во внутренних механизмах с java 7

прозрачен для jit

изначально разрабатывался с целью обеспечения прозрачности для jit-оптимизаций

никакой метаинформации

не знает ничего о классах, методах или полях в них. реализует только вызовы

требует строгого соответствия типов

нельзя просто взять и передать список object

ЧЕМ ОТЛИЧАЕТСЯ ОТ РЕФЛЕКСИИ

ЧЕМ ОТЛИЧАЕТСЯ ОТ РЕФЛЕКСИИ

reflection

method handle

ЧЕМ ОТЛИЧАЕТСЯ ОТ РЕФЛЕКСИИ

reflection	method handle
гибкий не требует строгой типизации	

ЧЕМ ОТЛИЧАЕТСЯ ОТ РЕФЛЕКСИИ

reflection

гибкий

не требует строгой типизации

method handle

строгий

нужно указывать всё и явно

ЧЕМ ОТЛИЧАЕТСЯ ОТ РЕФЛЕКСИИ

reflection

гибкий
не требует строгой типизации

всенаправленный
можно и в интроспекцию,
и в динамический вызов/доступ

method handle

строгий
нужно указывать всё и явно

ЧЕМ ОТЛИЧАЕТСЯ ОТ РЕФЛЕКСИИ

reflection

гибкий
не требует строгой типизации

всенаправленный
можно и в интроспекцию,
и в динамический вызов/доступ

method handle

строгий
нужно указывать всё и явно

узкий
только динамический вызов/доступ

ЧЕМ ОТЛИЧАЕТСЯ ОТ РЕФЛЕКСИИ

reflection

гибкий
не требует строгой типизации

всенаправленный
можно и в интроспекцию,
и в динамический вызов/доступ

медленный
из оптимизаций только кеширование
на высоком слое абстракции

method handle

строгий
нужно указывать всё и явно

узкий
только динамический вызов/доступ

ЧЕМ ОТЛИЧАЕТСЯ ОТ РЕФЛЕКСИИ

reflection

гибкий
не требует строгой типизации

всенаправленный
можно и в интроспекцию,
и в динамический вызов/доступ

медленный
из оптимизаций только кеширование
на высоком слое абстракции

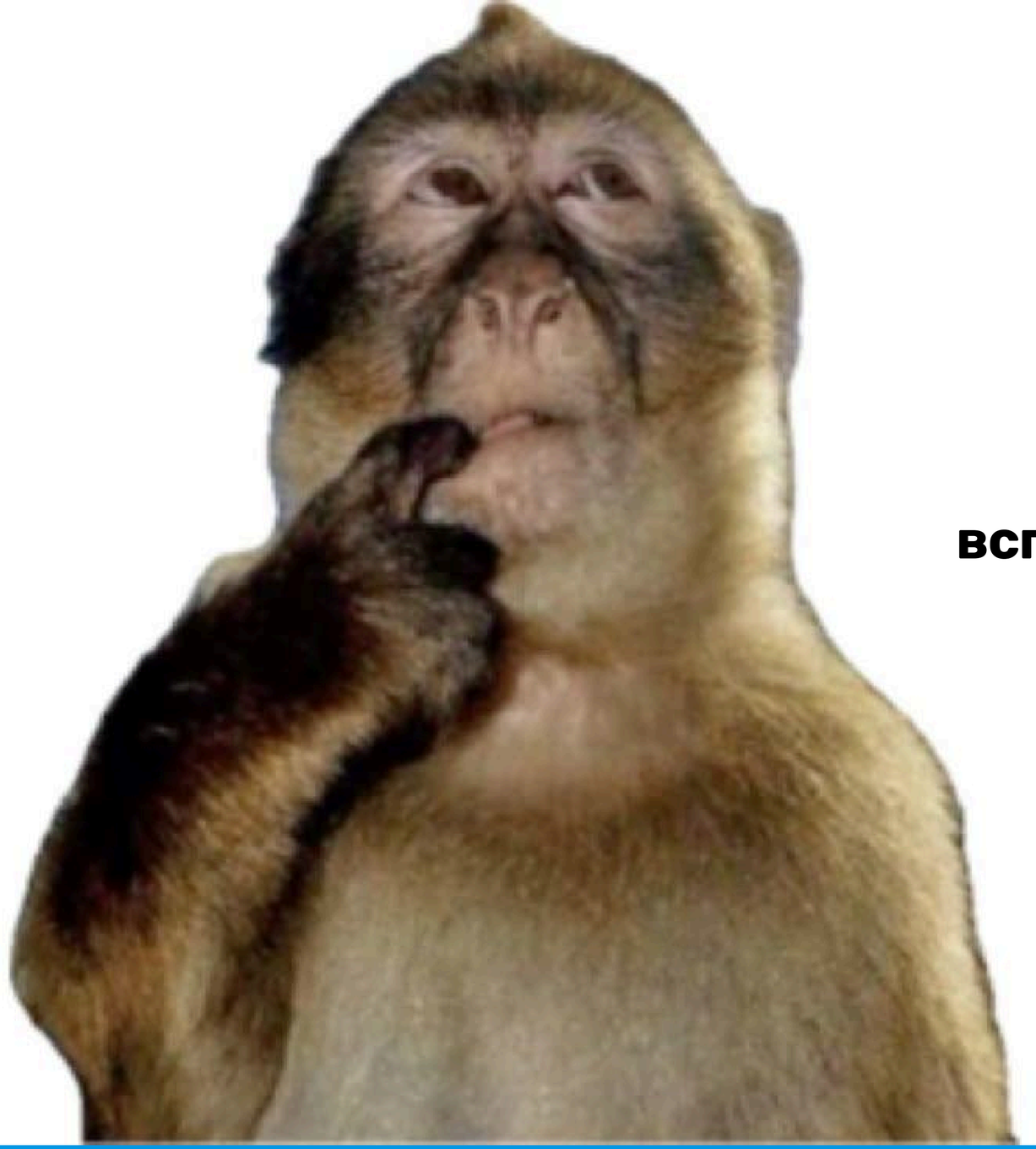
method handle

строгий
нужно указывать всё и явно

узкий
только динамический вызов/доступ

быстрый
прозрачен для всех оптимизаций java-платформы

КАКОВО ЭТО НА ПРАКТИКЕ



вспомнили

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       class Service {
3           void grind() { }
4           void chill() { }
5       }
6       var service = new Service();
7
8       String targetMethod =
9               ThreadLocalRandom.current().nextBoolean() ?
10                      "grind" :
11                      "chill";
12
13      Service.class
14              .getDeclaredMethod(targetMethod)
15              .invoke(service);
16  }
```

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       class Service {
3           void grind() { }
4           void chill() { }
5       }
6       var service = new Service();
7
8       String targetMethod =
9               ThreadLocalRandom.current().nextBoolean() ?
10                      "grind" :
11                      "chill";
12
13      Service.class
14              .getDeclaredMethod(targetMethod)
15              .invoke(service);
16  }
```

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       Service.class
4               .getDeclaredMethod(targetMethod)
5               .invoke(service);
6   }
7
8
9
10
11
12
13
14
15
16
```

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       MethodHandles.lookup()
4           .findVirtual(
5                   Service.class,
6                   targetMethod,
7                   MethodType.methodType(void.class)
8           )
9           .invokeExact(service);
10  }
11
12
13
14
15
16
```

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       MethodHandles.lookup()
4           .findVirtual(
5                   Service.class,
6                   targetMethod,
7                   MethodType.methodType(void.class)
8           )
9           .invokeExact(service);
10  }
11
12
13
14
15
16
```

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       MethodHandles.lookup()
4           .findVirtual(
5                   Service.class,
6                   targetMethod,
7                   MethodType.methodType(void.class)
8           )
9           .invokeExact(service);
10  }
11
12
13
14
15
16
```

создаём лукап

точку входа для работы с method handle и всем, что с ним связано

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       MethodHandles.lookup()
4           .findVirtual(
5                   Service.class,
6                   targetMethod,
7                   MethodType.methodType(void.class)
8           )
9           .invokeExact(service);
10  }
11
12
13
14
15
16
```

создаём лукап

точку входа для работы с method handle и всем, что с ним связано

получаем метод

ищем нестатический метод, принадлежащий классу

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       MethodHandles.lookup()
4           .findVirtual(
5                   Service.class,
6                   targetMethod,
7                   MethodType.methodType(void.class)
8           )
9           .invokeExact(service);
10  }
11
12  class Service {
13      void grind() { }
14      void chill() { }
15  }
16
```

создаём лукап

точку входа для работы с method handle и всем, что с ним связано

получаем метод

ищем нестатический метод, принадлежащий классу

в каком классе

ищем нестатический метод, принадлежащий классу

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       MethodHandles.lookup()
4           .findVirtual(
5                   Service.class,
6                   targetMethod,
7                   MethodType.methodType(void.class)
8           )
9           .invokeExact(service);
10  }
11
12  class Service {
13      void grind() { }
14      void chill() { }
15  }
16
```

с каким именем

ищем нестатический метод,
принадлежащий классу

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       MethodHandles.lookup()
4           .findVirtual(
5                   Service.class,
6                   targetMethod,
7                   MethodType.methodType(void.class)
8           )
9           .invokeExact(service);
10  }
11
12  class Service {
13      void grind() { }
14      void chill() { }
15  }
16
```

с каким именем

ищем нестатический метод,
принадлежащий классу

с каким типом

ищем нестатический метод,
принадлежащий классу

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       MethodHandles.lookup()
4           .findVirtual(
5                   Service.class,
6                   targetMethod,
7                   MethodType.methodType(void.class)
8           )
9           .invokeExact(service);
10  }
11
12
13
14
15
16
```

с каким именем

ищем нестатический метод, принадлежащий классу

с каким типом

ищем нестатический метод, принадлежащий классу

вызываем метод

успешно нашли метод, теперь вызываем его на объекте

КАКОВО ЭТО НА ПРАКТИКЕ


```
1   void main() {
2       // ....
3       MethodHandles.lookup()
4           .findVirtual(
5                   Service.class,
6                   targetMethod,
7                   MethodType.methodType(void.class)
8           )
9           .invokeExact(service);
10  }
11
12
13
14
15
16
```

с каким именем

ищем нестатический метод, принадлежащий классу

с каким типом

ищем нестатический метод, принадлежащий классу

вызываем метод

успешно нашли метод, теперь вызываем его на объекте

почему exact?

если можно вызвать его точно, значит можно и не точно?


почему

METHOD HANDLE "ОСОБЕННЫЙ"

METHOD HANDLE "ОСОБЕННЫЙ"

```
1   class MethodHandle {
2
3       @IntrinsicCandidate
4       public final native @PolymorphicSignature Object invokeExact(Object... args)
5           throws Throwable;
6
7       @IntrinsicCandidate
8       public final native @PolymorphicSignature Object invoke(Object... args)
9           throws Throwable;
10
11  }
```

приведение типов

строгая java не любит неопределённость в типах и ей
нужно преподносить их уже готовенькими

```
1   class MethodHandle {
2
3       @IntrinsicCandidate
4       public final native @PolymorphicSignature Object invokeExact(Object... args)
5           throws Throwable;
6
7       @IntrinsicCandidate
8       public final native @PolymorphicSignature Object invoke(Object... args)
9           throws Throwable;
10
11  }
```

invokeExact ничего за вас не делает

если метод принимает Integer, а вы передаёте int,
invokeExact его не преобразует

```
1   class MethodHandle {
2
3       @IntrinsicCandidate
4       public final native @PolymorphicSignature Object invokeExact(Object... args)
5           throws Throwable;
6
7       @IntrinsicCandidate
8       public final native @PolymorphicSignature Object invoke(Object... args)
9           throws Throwable;
10
11  }
```

invoke чуть-чуть разрешает

теперь можно спокойно передавать Integer в int.

#invoke преобразует в нужный тип

```
1   class MethodHandle {
2
3       @IntrinsicCandidate
4       public final native @PolymorphicSignature Object invokeExact(Object... args)
5           throws Throwable;
6
7       @IntrinsicCandidate
8       public final native @PolymorphicSignature Object invoke(Object... args)
9           throws Throwable;
10
11  }
```

какие строгие типы с object

везде указан object, методы принимают неопределённое количество параметров, а броситься может вообще throwable

```
1   class MethodHandle {
2
3       @IntrinsicCandidate
4       public final native @PolymorphicSignature Object invokeExact(Object... args)
5           throws Throwable;
6
7       @IntrinsicCandidate
8       public final native @PolymorphicSignature Object invoke(Object... args)
9           throws Throwable;
10
11  }
```

дело в полиморфизме

специальный флаг для
компилятора

```
1   class MethodHandle {
2
3       @IntrinsicCandidate
4       public final native @PolymorphicSignature Object invokeExact(Object... args)
5           throws Throwable;
6
7       @IntrinsicCandidate
8       public final native @PolymorphicSignature Object invoke(Object... args)
9           throws Throwable;
10
11  }
```

ЧТО ТАКОЕ ПОЛИМОРФНАЯ СИГНАТУРА

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная

сигнатура

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная

сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная

сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

```
1 int max(int a, int b) {
2     // ....
3 }
```

перешли к декларации

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная
сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

```
1 int max(int a, int b) {
2     // ....
3 }
```

перешли к декларации

```
1 Math.max:(II)I
```

с её типами создали
байт-код инструкцию

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

```
1 int max(int a, int b) {
2     // ....
3 }
```

перешли к декларации

```
1 Math.max:(II)I
```

с её типами создали байт-код инструкцию

полиморфная сигнатура

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

```
1 int max(int a, int b) {
2     // ....
3 }
```

перешли к декларации

```
1 Math.max:(II)I
```

с её типами создали байт-код инструкцию

полиморфная сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

```
1 int max(int a, int b) {
2     // ....
3 }
```

перешли к декларации

```
1 Math.max:(II)I
```

с её типами создали байт-код инструкцию

полиморфная сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

```
1 Object a;
2 long b;
3 int c = ....;
```

посмотрели на переменные

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

```
1 int max(int a, int b) {
2     // ....
3 }
```

перешли к декларации

```
1 Math.max:(II)I
```

с её типами создали байт-код инструкцию

полиморфная сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

```
1 Object a;
2 long b;
3 int c = ....;
```

посмотрели на переменные

```
1 Math.max:(ObjectJ)I
```

с их типами создали байт-код инструкцию

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная сигнатура

```
1 int max(int a, int b) {
2     // ....
3 }
```

типы всегда берутся из декларации метода

полиморфная сигнатура

```
1 int c = Math.max(a, b);
```

увидели вызов

```
1 Object a;
2 long b;
3 int c = ....;
```

посмотрели на переменные

```
1 Math.max:(ObjectJ)I
```

с их типами создали байт-код инструкцию

КАК ПРОИСХОДИТ КОМПИЛЯЦИЯ

обычная сигнатура

```
1 int max(int a, int b) {
2     // ....
3 }
```

типы всегда берутся из декларации метода

полиморфная сигнатура

```
1 Object a;
2 long b;
3 int c = ....;
```

типы определяются по месту вызова метода


зачем?

JIT-КОМПИЛЯТОР ЛЮБИТ ОПРЕДЕЛЁННОСТЬ

приведём пример

возвращаемся к коду

возвращаемся к коду

в отличие от рефлексии, method handle
поддерживает jit-оптимизации

возьмём класс с простым методом

трепещи, Перри-Утконос!

я изобрёл префиксинатор!

```
1   class Prefixinator {
2       static String withPrefix(String str) {
3           return "[OP] " + str;
4       }
5   }
6
7
8
9
10
11
```

создадим method handle

уже описанным раньше

способом через lookup

```
1   class Prefixinator {
2       static String withPrefix(String str) {
3           return "[OP] " + str;
4       }
5   }
6
7   var Prefixinator_withPrefix = lookup.findStatic(
8           Prefixinator.class, "withPrefix", methodType(String.class, String.class)
9   );
10
11
```

и вызовем его

трепещи, Перри-Утконос!

я изобрёл префиксинатор!

```
1   class Prefixinator {
2       static String withPrefix(String str) {
3           return "[OP] " + str;
4       }
5   }
6
7   var Prefixinator_withPrefix = lookup.findStatic(
8           Prefixinator.class, "withPrefix", methodType(String.class, String.class)
9   );
10
11  var prefixed = (String) Prefixinator_withPrefix.invokeExact("Me");
```

даже явно указав возвращаемый тип

помним, что method handle требует

строгого приведения типов

```
1   class Prefixinator {
2       static String withPrefix(String str) {
3           return "[OP] " + str;
4       }
5   }
6
7   var Prefixinator_withPrefix = lookup.findStatic(
8           Prefixinator.class, "withPrefix", methodType(String.class, String.class)
9   );
10
11  var prefixed = (String) Prefixinator_withPrefix.invokeExact("Me");
```

jit смотрит куда вёдет method handle

jit смотрит на метод, к которому
относится method handle

```
1   class Prefixinator {
2       static String withPrefix(String str) {
3           return "[OP] " + str;
4       }
5   }
6
7   var Prefixinator_withPrefix = lookup.findStatic(
8           Prefixinator.class, "withPrefix", methodType(String.class, String.class)
9   );
10
11  var prefixed = (String) Prefixinator_withPrefix.invokeExact("Me");
```

jit решает избавиться от посредника

jit понимает, что вызов method handle можно заменить на обычный вызов метода и делает это

```
1   class Prefixinator {
2       static String withPrefix(String str) {
3           return "[OP] " + str;
4       }
5   }
6
7   var Prefixinator_withPrefix = lookup.findStatic(
8           Prefixinator.class, "withPrefix", methodType(String.class, String.class)
9   );
10
11  var prefixed = withPrefix("Me");
```

ТО ЕСТЬ

method handle реализует динамический вызов без минусов рефлексии

а это точно быстрее?

jit-оптимизации это очень хорошо, но какой прирост на практике?

ПРОИЗВОДИТЕЛЬНОСТЬ

бенчмарк

ПРОИЗВОДИТЕЛЬНОСТЬ

бенчмарк

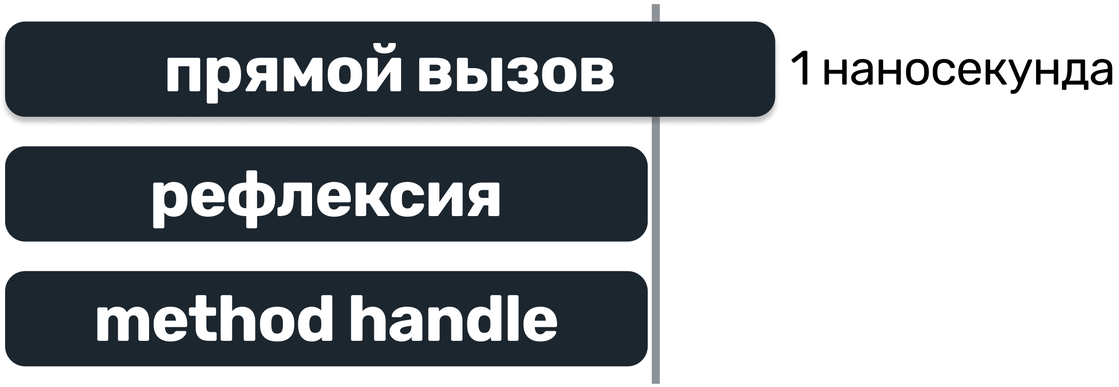
ПРОИЗВОДИТЕЛЬНОСТЬ

бенчмарк

- прямой вызов
- рефлексия
- method handle

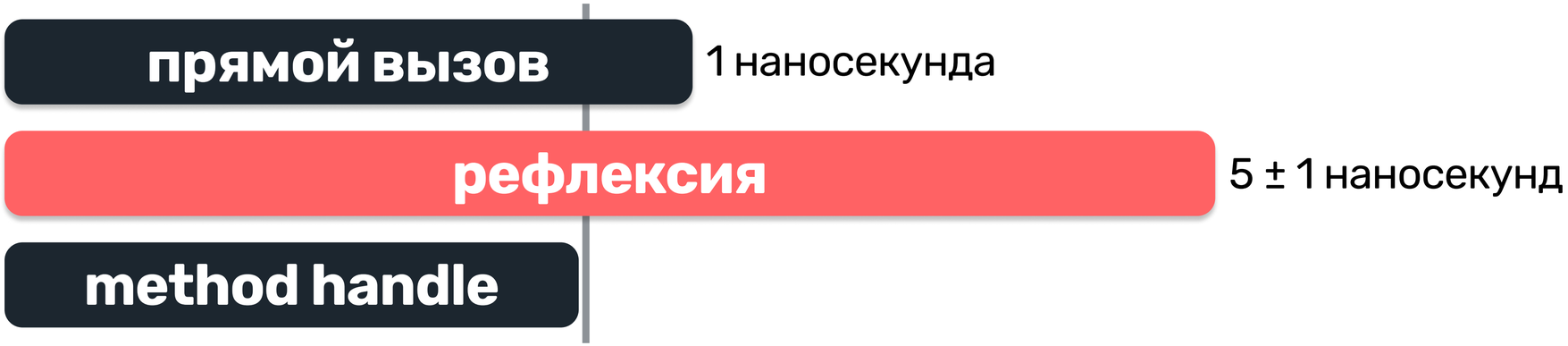
ПРОИЗВОДИТЕЛЬНОСТЬ

бенчмарк



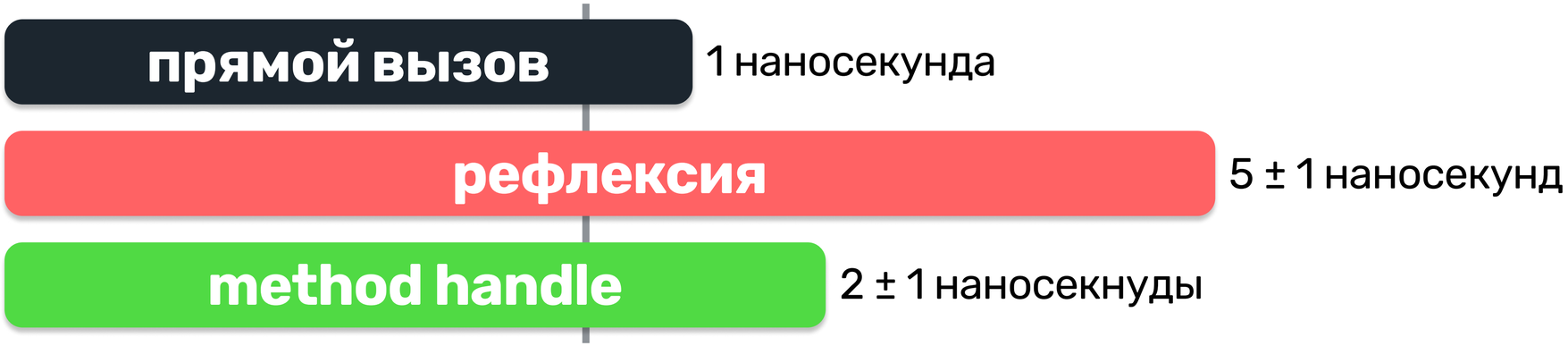
ПРОИЗВОДИТЕЛЬНОСТЬ

бенчмарк



ПРОИЗВОДИТЕЛЬНОСТЬ

бенчмарк



ТО ЕСТЬ

method handle быстрее рефлексии в большинстве случаев

method handle не просто api

сама java использует его в своих механизмах

METHOD HANDLE ВНУТРИ JAVA

METHOD HANDLE ВНУТРИ JAVA

```
1  interface Lambda {
2    void action();
3  }
4
5  Lambda say = () →
6    println("hey!");
7
8  say.action();
9
10
```

лямбды

лямбды изначально работают через method handle, что позволяет одновременно производительными и гибкими

METHOD HANDLE ВНУТРИ JAVA

```
1  interface Lambda {
2    void action();
3  }
4
5  Lambda say = () →
6    println("hey!");
7
8  say.action();
9
10
```

лямбды

лямбды изначально работают через method handle, что позволяет одновременно производительными и гибкими

конкатенация строк

с java 9 конкатенация также работает через method handle, генерируя код прямо на лету

```
1  int a;
2  int b;
3
4  // ....
5
6  String render =
7    a + "/" + b;
8
9  println(render);
10
```

ЧТО МОЖНО СКАЗАТЬ

method handle — не замена рефлексии

рефлексия

получение метаинформации о заргуженных классах

method handle

выполнение методов динамически

КОМБИНИРОВАННЫЙ ПОДХОД

КОМБИНИРОВАННЫЙ ПОДХОД

ищем рефлексией аннотации

method handle тут не сможет

```
1   class EventListener implements Listener {
2
3     @EventHandler
4     public void onBlockBreak(BlockBreakEvent event) {
5       // ....
6     }
7
8
9
10    void register(PluginManager pluginManager) {
11       pluginManager.register(this);
12    }
13  }
```

КОМБИНИРОВАННЫЙ ПОДХОД

ищем рефлексией аннотации

method handle тут не сможет

вызываем через method handle

добиваемся максимальной производительности

```
1   class EventListener implements Listener {
2
3     @EventHandler
4     public void onBlockBreak(BlockBreakEvent event) {
5       // ....
6     }
7
8
9     // ....
10    {
11      Listener_onBlockBreak.invoke(listener, event);
12    }
13  }
```

ВЫВОДЫ

НА ЭТОМ ВСЁ