Тема: Оптимизация байт-кода, генерируемого компилятором Kotlin

Студент: Жарков Д.С.

Руководитель: Бреслав А.А.

Санкт-Петербург, 2015

Kotlin

```
fun main(args: Array<String>) {
    println("Hello World!")
}
```

- *Kotlin* — статически типизированный язык программирования, компилирующийся в JVM байт-код.
- Java-подобный лаконичный синтаксис
- Функции высшего порядка
- Разрабатывается в Jetbrains параллельно с плагином для IntelliJ IDEA

Постановка задачи

Цель: Улучшение производительности байт-кода, генерируемого компилятором Kotlin

- *Изучение подходов к оптимизации в других компиляторах для JVM*
- *Реализация бенчмарков*
- *Анализ результатов измерений*
- *Исправление найденных недостатков*

Подходы к решению

- *Java 6/7* — компиляция «как есть»

Подходы к решению

- *Java 6/7* — компиляция «как есть»
- *Java 8* — генерация анонимных функций с помощью инструкции “INVOKEDYNAMIC”

Подходы к решению

- *Java 6/7* — компиляция «как есть»
- *Java 8* — генерация анонимных функций с помощью инструкции “INVOKEDYNAMIC”
- *Scala* — полный спектр оптимизаций для вызовов функций высшего порядка

Подходы к решению

- *Java 6/7* — компиляция «как есть»
- *Java 8* — генерация анонимных функций с помощью инструкции “INVOKEDYNAMIC”
- *Scala* — полный спектр оптимизаций для вызовов функций высшего порядка
- *ProGuard* — подходы к уменьшению размера байт-кода

Бенчмарки

Макробенчмарки

- Сортировка слиянием
- АВЛ-дерево
- Встраиваемые функции из стандартной библиотеки

Бенчмарки

Макробенчмарки

- Сортировка слиянием
- АВЛ-дерево
- Встраиваемые функции из стандартной библиотеки

Микробенчмарки — отдельные синтаксические конструкции

When

```
val y = when(x) {
    null -> "null"
    in 1..100 -> "1..100"
    is String -> "String"
    else -> "other"
}

val y = when(x) {
    1, 2 -> "1, 2"
    3, 4 -> "3, 4"
    else -> "other"
}
```

When

```
switch(x) {
    case 1:
    case 2:
        System.out.println("1,2");
        break;
    case 3:
    case 4:
        System.out.println("3,4");
        break;
}
```

- Каждый оператор switch в Java компилируется в одну из двух JVM-инструкций
  - *tableswitch* — работает за $O(1)$ времени и требует $O(max - min)$ памяти
  - *lookupswitch* — требует $O(n)$ памяти. По спецификации может работать за $O(n)$

When. Измерения после оптимизаций

	Java (мкс)	Kotlin (мкс)	Opt (мкс)	Уск (раз)
ЦЧ (1..20)	4.28	7.8	4.28	1.82
ЦЧ (1..20,∞)	5.55	7.52	5.55	1.36
Enum	10.5	13.89	10.48	1.33
Строки	21.67	81.93	19.3	4.25

Избыточный боксинг

```
array.count { it % 2 == 0 }
```

Избыточный боксинг

```
array.count { it % 2 == 0 }
```

```
// Function1<Integer, Boolean> predicate = { it % 2 == 0 };
int count = 0;
for (int element : array) {
    if (predicate.invoke(
                Integer.valueOf(element)
            ).booleanValue()
        ) count++;
}
```

Удаление избыточного боксинга

Удаление избыточного боксинга

- Разметка состояния памяти для каждой инструкции $\mathbb{X} = \{\top = \mathit{Object}, \bot = \mathit{Nothing}, \mathit{Boxed}\}$ — является ли конкретное значение «запакованным примитивом»

Удаление избыточного боксинга

- Разметка состояния памяти для каждой инструкции $\mathbb{X} = \{\top = \mathit{Object}, \bot = \mathit{Nothing}, \mathit{Boxed}\}$ — является ли конкретное значение «запакованным примитивом»
- Анализ использования «запакованных» значений

Удаление избыточного боксинга

- Разметка состояния памяти для каждой инструкции $\mathbb{X} = \{\top = \mathit{Object}, \bot = \mathit{Nothing}, \mathit{Boxed}\}$ — является ли конкретное значение «запакованным примитивом»
- Анализ использования «запакованных» значений
- Поиск «зараженных» значений

```
var x = Integer.valueOf(1)
if (p) {
    x = Integer.valueOf(2)
    someMethod(x) // expected boxed value
}
```

После выполнения оператора ‘if’ в x будет одно из двух значений (причем боксинг второго удалять нельзя)

Удаление избыточного боксинга

- Разметка состояния памяти для каждой инструкции $\mathbb{X} = \{\top = \mathit{Object}, \bot = \mathit{Nothing}, \mathit{Boxed}\}$ — является ли конкретное значение «запакованным примитивом»
- Анализ использования «запакованных» значений
- Поиск «зараженных» значений

  ```
  var x = Integer.valueOf(1)
  if (p) {
      x = Integer.valueOf(2)
      someMethod(x) // expected boxed value
  }
  ```

  После выполнения оператора ‘if’ в x будет одно из двух значений (причем боксинг второго удалять нельзя)
- Удаление «безопасных» операций боксинга и распаковки и адаптация байт-кода

Избыточный боксинг. Измерение производительности

array.count { it % 2 == 0 }

Размер	Эталон	До	После	Ускорение (раз)
100	87.97 нс	348.25 нс	88.63 нс	3.93
10000	18.4 мкс	78.05 мкс	18.66 мкс	4.18
1000000	3.71 мс	8.57 мс	3.71 мс	2.31

Избыточный боксинг. Измерение производительности

array.count { it % 2 == 0 }

Размер	Эталон	До	После	Ускорение (раз)
100	87.97 нс	348.25 нс	88.63 нс	3.93
10000	18.4 мкс	78.05 мкс	18.66 мкс	4.18
1000000	3.71 мс	8.57 мс	3.71 мс	2.31

array.filter { it % 2 == 0 }

Размер	Эталон	До	После	Ускорение (раз)
100	598.23 нс	711.11 нс	482.67 нс	1.47
10000	98.65 мкс	121.1 мкс	97.94 мкс	1.24
1000000	11.87 мс	13.68 мс	11.29 мс	1.21

Избыточный боксинг. Измерение производительности

array.count { it % 2 == 0 }

Размер	Эталон	До	После	Ускорение (раз)
100	87.97 нс	348.25 нс	88.63 нс	3.93
10000	18.4 мкс	78.05 мкс	18.66 мкс	4.18
1000000	3.71 мс	8.57 мс	3.71 мс	2.31

array.filter { it % 2 == 0 }

Размер	Эталон	До	После	Ускорение (раз)
100	598.23 нс	711.11 нс	482.67 нс	1.47
10000	98.65 мкс	121.1 мкс	97.94 мкс	1.24
1000000	11.87 мс	13.68 мс	11.29 мс	1.21

array.fold(0) { sum, x -> sum + x }

Размер	Эталон	До	После	Ускорение (раз)
100	27.83 нс	594.34 нс	27.93 нс	21.28
10000	2.77 мкс	100.18 мкс	2.78 мкс	36.09
1000000	301.7 мкс	11.28 мс	304.42 мкс	37.05

Безопасный вызов + Elvis

```
obj?.field // Safe-call
if (obj != null) obj.field else null
```

Безопасный вызов + Elvis

```
obj?.field // Safe-call
if (obj != null) obj.field else null
```

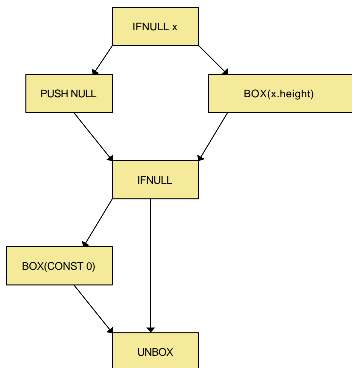
```
mayBeNull ?: 0 // Elvis
if (mayBeNull != null) mayBeNull else 0
```

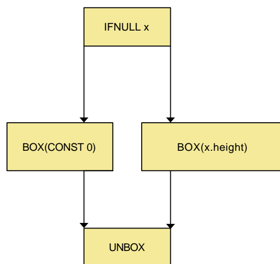
Безопасный вызов + Elvis

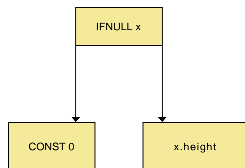
```
x?.height ?: 0
```

Безопасный вызов + Elvis

```
x?.height ?: 0
```







Безопасный вызов + Elvis

АВЛ-Дерево. Добавление элементов

Размер	Java	Kotlin	После	Ускорение (раз)
100	19.41 мкс	30.69 мкс	21.84 мкс	1.41
1000	300.05 мкс	476.81 мкс	335.53 мкс	1.42
100000	78.69 мс	132.77 мс	79.39 мс	1.67

Нарушение условия OSR

```
fun countBenchmark(bh: Blackhole) {
    bh.consume(array.fold { sum, x -> sum + x })
}
```

Нарушение условия OSR

```
fun countBenchmark(bh: Blackhole) {
    bh.consume(array.fold { sum, x -> sum + x })
}

fun countBenchmark(bh: Blackhole) {
    val result = array.fold { sum, x -> sum + x }
    bh.consume(result)
}
```

Нарушение условия OSR

```
fun countBenchmark(bh: Blackhole) {
    bh.consume(array.fold { sum, x -> sum + x })
}
```

Нарушение условия OSR

```
fun countBenchmark(bh: Blackhole) {
    bh.consume(array.fold { sum, x -> sum + x })
}

// after inline
fun countBenchmark(bh: Blackhole) {
    bh.consume(for (element in array) ... )
}
```

Нарушение условия OSR

```
fun countBenchmark(bh: Blackhole) {
    bh.consume(array.fold { sum, x -> sum + x })
}

// after inline
fun countBenchmark(bh: Blackhole) {
    bh.consume(for (element in array) ... )
}

    ALOAD 1 // load bh
    L1: // label
    .... // for
    GOTO L1
    CALL consume
```

Нарушение условия OSR. Решение

```
fun countBenchmark(bh: Blackhole) {
    bh.consume(
        /* inline begin*/
        array.fold { sum, x -> sum + x }
        /* inline end */
    )
}
```

Нарушение условия OSR. Решение

```
fun countBenchmark(bh: Blackhole) {
    bh.consume(
        /* inline begin*/
        array.fold { sum, x -> sum + x }
        /* inline end */
    )
}
```

```
    ALOAD 1 // load bh
    ASTORE 2 // store tmp$0
    .... // fold
    ALOAD 2
    CALL consume
```

Нарушение условия OSR. Измерение производительности

```
bh.consume(array.fold { sum, x -> sum + x })
```

Размер	Эталон	До	После	Ускорение (раз)
100	27.83 нс	595.96 нс	28.06 нс	21.24
10000	2.77 мкс	95.1 мкс	2.8 мкс	34.01
1000000	301.7 мкс	114.62 мс	302.12 мкс	379.39

Спасибо за внимание!

Результаты

- *Реализован набор бенчмарков*
- *Генерация when для целочисленных констант, строк и enum-классов*
  (ускорение от 1.3 до 4.3 раз)
- *Удаление избыточного боксинга*
  (ускорение от 1.2 до 37 раз)
- *Оптимизация для сочетания Elvis + Safe-call*
  (ускорение от 1.4 до 1.7 раз)
- *Решение проблемы с нарушением условия OSR*
  (ускорение до 379.3 раз)
- *Удаление недостижимого кода*