Алексей Андреев | Delightex

konsoletyper@gmail.com

- Алексей Андреев
- Компиляторщик-энтузиаст
- Два года работал в JetBrains, контрибьютил в Kotlin/JS
- До JetBrains работал в энтерпрайзе, в том числе fullstack-разработчиком
- Сейчас работаю в Delightex

TeaVM

- AOT-компилятор байт-кода Java
- Самый известный и стабильный бэкэнд — JavaScript
- Есть WebAssembly и C
- Появился в 2013-м году как хобби-проект
- Опыт с GWT привёл к мысли: можно ли транслировать байт-код вместо исходников
- С весны 2019 production ready

Аналоги

- GWT: самый известный компилятор Java в JavaScript от Google
- J2CL: проект от Google, призванный заменить GWT
- Kotlin/JS: компилятор Kotlin в JS от JetBrains
- Scala.js: компилятор Scala в JS
- CheerpJ: виртуальная машина Java в браузере
- И т.д.

Кем используется

- CoSpaces (хотели начать мигрировать на Kotlin, GWT не давал)
- Codename One (хотели потоки как в Java)
- Greenfoot (хотели потоки как в Java)
- Возможно, кто-то ещё

CoSpaces

https://edu.cospaces.io/

Трудности перевода

- Java и JavaScript не так уж и похожи
- Дьявол — в деталях
- Эмулируем возможности Java в JavaScript
- Получить полностью идиоматический код на JavaScript невозможно

Перегрузка по сигнатуре

```
static void foo(String value) {
  System.out.println("string");
}

static void foo(Integer value) {
  System.out.println("integer");
}
```

Перегрузка по сигнатуре

```
function foo(value) {
  switch (typeof value) {
    case "string":
      console.log("string");
      break;
    case "number":
      console.log("integer");
      break;
  }
}
```

Перегрузка по сигнатуре

```java
static void foo(String value) {
  System.out.println("string");
}

static void foo(Integer value) {
  System.out.println("integer");
}

static void bar() {
  foo((String) null);
  foo((Integer) null);
}
```

Перегрузка по сигнатуре

```
static void foo(String value) {
  System.out.println("string");
}

static void foo(Integer value) {
  System.out.println("integer");
}

static void bar() {
  foo((String) null);
  foo((Integer) null);
}
```

Как различить эти null на рантайме?

Перегрузка по сигнатуре

```
function foo_1(value) {
    return "string";
}

function foo_2(value) {
    return "integer";
}

function bar() {
    foo_1(null);
    foo_2(null);
}
```

Различаем их в компайл-тайме

```java
class A {
  static int FOO = bar();

  static int bar() {
    return 23;
  }
}

class Test {
  public static void main(String[] args) {
    System.out.println(A.FOO);
  }
}
```

```
class A {
    static bar() {
        return 23;
    }
}
A.FOO = A.bar();

console.log(A.FOO);
```

<clinit>

```
class A {
    static int FOO = bar();

    static int bar() {
        System.out.println("bar");
        return 23;
    }
}

class Test {
    public static void main(String[] args) {
        System.out.println("start");
        System.out.println(A.FOO);
    }
}
```

```
start
bar
23
```

<clinit>

- Специальный статический метод, который javac вставляет в классы, где есть статические поля с инициализаторами
- Вызывается при инициализации класса в следующих случаях
  - при вызове статического метода
  - при чтении или изменении статического поля
  - при создании экземпляра
  - при вызове Class.forName
- Разумеется, только один раз
- В JavaScript у классов нет инициализаторов

<clinit>

```
class A {
    static bar() {
        console.log("bar");
        return 23;
    }
    static clinit() {
        A.clinit = () => {};
        A.FOO = bar();
    }
}
console.log("start");
A.clinit();
console.log(A.FOO);
```

equals/hashCode

```
class A {
    int value;
    A(int value) {
        this.value = value;
    }
    public static void main(String[] args) {
        var set = new HashSet<A>();
        set.add(new A(23));
        set.add(new A(23));
        set.add(new A(42));
    }
}
```

```
class A {
    constructor(value) {
        this.value = value;
    }
}
var set = new Set();
set.add(new A(23));
set.add(new A(23));
set.add(new A(42));
```

equals/hashCode

```
class A {
    int value;
    A(int value) {
        this.value = value;
    }
    public boolean equals(Object that) {
        return value == ((A) that).value;
    }
    public int hashCode(Object that) { return value; }
    static void main(String[] args) {
        var set = new HashSet<A>();
        set.add(new A(23));
        set.add(new A(23));
        set.add(new A(42));
    }
}
```

- В JS в классе Object нет equals/hashCode, соответственно коллекции ничего про это не знают
- Выход: определить свой базовый класс для всех Java-классов (например, jl_Object), но тогда имеем проблемы с интеропом
- Альтернатива (GWT, Kotlin/JS): на каждом вызове equals/hashCode генерируем код, который выясняет тип объекта и применяет нужную логику (плохо для производительности)
- Также нужно написать свою реализацию коллекций

ArrayList

```java
class A {
  public static void main(String[] args) {
    var list = new ArrayList<>(List.of(2, 3, 5, 7, 11));
    for (var elem : list) {
      foo(elem, list);
    }
  }
  static void foo(int elem, List<Integer> list) {
    System.out.println(elem + " of " + list);
  }
}
```

ArrayList

```
var list = [2, 3, 5, 7, 11];
for (var elem of list) {
    foo(elem, list);
}

function foo(elem, list) {
  console.log(`${elem} of ${list}`);
}
```

ArrayList

```
class A {
  public static void main(String[] args) {
    var list = new ArrayList<>(List.of(2, 3, 5, 7, 11));
    for (var elem : list) {
      foo(elem, list);
    }
  }
  static void foo(int elem, List<Integer> list) {
    System.out.println(elem + " of " + list);
    if (elem == 2) list.add(13);
  }
}
```

ArrayList

```
var list = [2, 3, 5, 7, 11];
for (var elem of list) {
  foo(elem, list);
}

function foo(elem, list) {
  console.log(`${elem} of ${list}`);
  if (elem === 2) list.push(13);
}
```

ArrayList

```
2 of 2,3,5,7,11
3 of 2,3,5,7,11,13
5 of 2,3,5,7,11,13
7 of 2,3,5,7,11,13
11 of 2,3,5,7,11,13
13 of 2,3,5,7,11,13
```

```
2 of [2, 3, 5, 7, 11]
Exception in thread "main" java.util.ConcurrentModificationException
	at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1042)
	at java.base/java.util.ArrayList$Itr.next(ArrayList.java:996)
	at A.main(A.java:7)
```

Компилятор

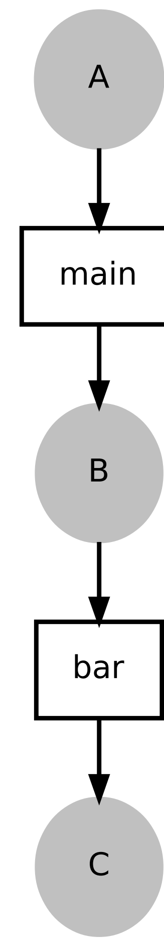
- Что компилировать
- Оптимизации
- Invokedynamic
- Получаем AST
- Green threads

Что компилировать

- JavaScript должен быть как можно меньше
- Компилируем только код, который действительно будет исполняться
- Граф зависимостей классов
- Класс с main-методом достижим
- Пусть A достижим
- Пусть B упоминается в A
- Значит, B тоже достижим

Что компилировать

```
class A {
  public static void main(String[] args) {
    B.foo();
  }
}

class B {
  static void foo() {
    System.out.println("I'm B.foo");
  }
  static void bar() {
    C.baz();
  }
}
```



Что компилировать

- bar не используется, но мы его компилируем
- bar тянет за собой целый класс C
- Значит, обходим граф методов
- Метод main достижим
- Пусть метод a достижим
- Пусть b вызывается в методе a
- Тогда b тоже достижим
- В этом случае достижимы только A.main и B.foo
- Проблема: виртуальные вызовы

Что компилировать

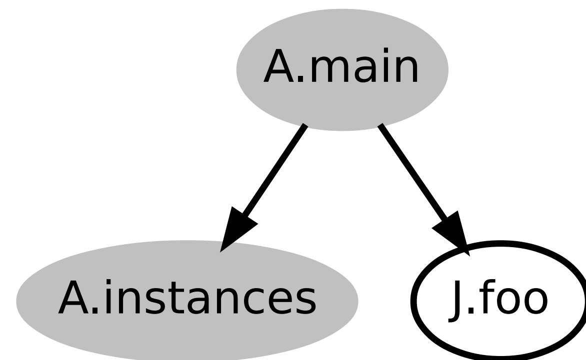
```java
class A {
  public static void main(String[] args) {
    for (J instance : instances()) {
      instance.foo();
    }
  }

  static List<J> instances() {
    return Arrays.asList(new B(), new C());
  }
}
```

Что компилировать

```
class A {
  public static void main(String[] args) {
    for (J instance : instances()) {
      instance.foo();
    }
  }

  static List<J> instances() {
    return Arrays.asList(new B(), new C());
  }
}
```

⟵ *что такое instance.foo?* (указывает на строку `instance.foo();`)

Что компилировать

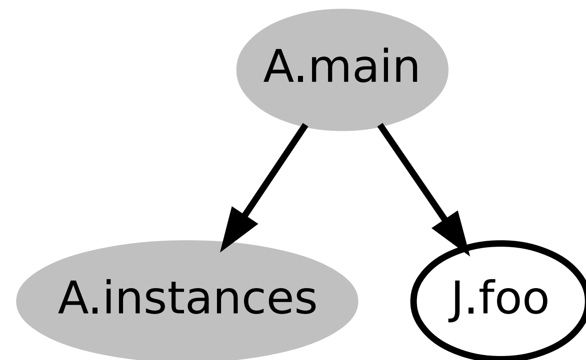



```
class A {
  public static void main(String[] args) {
    for (J instance : instances()) {
      instance.foo();
    }
  }

  static List<J> instances() {
    return Arrays.asList(new B(), new C());
  }
}
```

new: ∅

Что компилировать

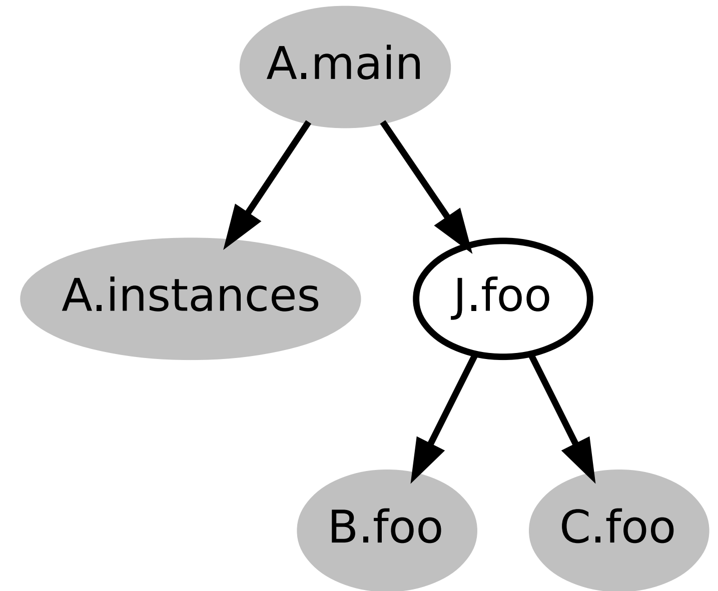
```
class A {
  public static void main(String[] args) {
    for (J instance : instances()) {
      instance.foo();
    }
  }

  static List<J> instances() {
    return Arrays.asList(new B(), new C());
  }
}
```



new: { B, C }

Что компилировать

```
class A {
  public static void main(String[] args) {
    for (J instance : instances()) {
      instance.foo();
    }
  }

  static List<J> instances() {
    return Arrays.asList(new B(), new C());
  }
}
```



new: { B, C }

Что компилировать

- Бонус: девиртуализация
  - Если виртуальный метод резолвится только в одну реализацию, подменяем виртуальный вызов этого метода на прямой вызов реализации
  - Прямые вызовы метода быстрее, чем виртуальные
- Бонус: граф вызовов
  - Полезен для некоторых оптимизаций

Что компилировать

- Проблема: reflection
  - Method.invoke — может быть вызван любой метод любого класса
  - Class.forName — может быть вообще любой класс
  - Выход: не использовать reflection вообще (annotation processors)
  - Выход: явно аннотировать методы, которые можно вызвать через reflection
- Есть более сильные алгоритмы, чем предложенный

Что компилировать

```
class A {
    public static void main(String[] args) {
        for (J instance : instances()) {
            instance.foo();
        }
    }

    static List<J> instances() {
        new B();
        return Arrays.asList(new C());
    }
}
```

- Очевидно, B.foo уже не нужен
- Надо понимать, что возвращает instances
- Это анализ потока данных (data-flow analysis)
- Data-flow analysis пытается понять что-то про значения, которые переменная может принять в каждой точке программы
- Например, какие могут быть типы у этих значений
- Зная это в точке виртуально вызова, можем правильно разрешить метод
- Для data-flow analysis удобен SSA (static single assignment)

SSA

```
var a = 1;
System.out.println(a);
a += 1;
System.out.println(a);
```

- Тут можно подставить константы
- Как это сделать очевидным для машины?
- a изменяется, поэтому это не константа
- Надо работать в терминах определений, а не переменных
- Или переписать код

SSA

```
var a_1 = 1;
System.out.println(a_1);
var a_2 = a_1 + 1;
System.out.println(a_2);
```

- То что получилось — SSA
- SSA — это такое промежуточное представление (intermediate representation, IR)
- В SSA каждая переменная присваивается только один раз
- Теперь протащить константы можно очевидным способом
- SSA используется в любом нормальном компиляторе

- Global value numbering
- Loop invariant code motion
- Dead code elimination
- Redundant null check elimination
- Class initialization elimination
- Scalar replacement
- Inlining
- Devirtualization
- Eager class initialization

- Invokedynamic — это в какой-то степени reflection
- Есть invokedynamic, про которые мы знаем как они работают
- Если в байт-коде неизвестные invokedynamic, то ругаемся на него
- К счастью, javac генерирует только известные invokedynamic
- LambdaMetafactory:
  - генерируем класс во время компиляции
  - заменяем invokedynamic на создание экземпляра этого класса
- StringConcatFactory: заменяем на StringBuilder.append

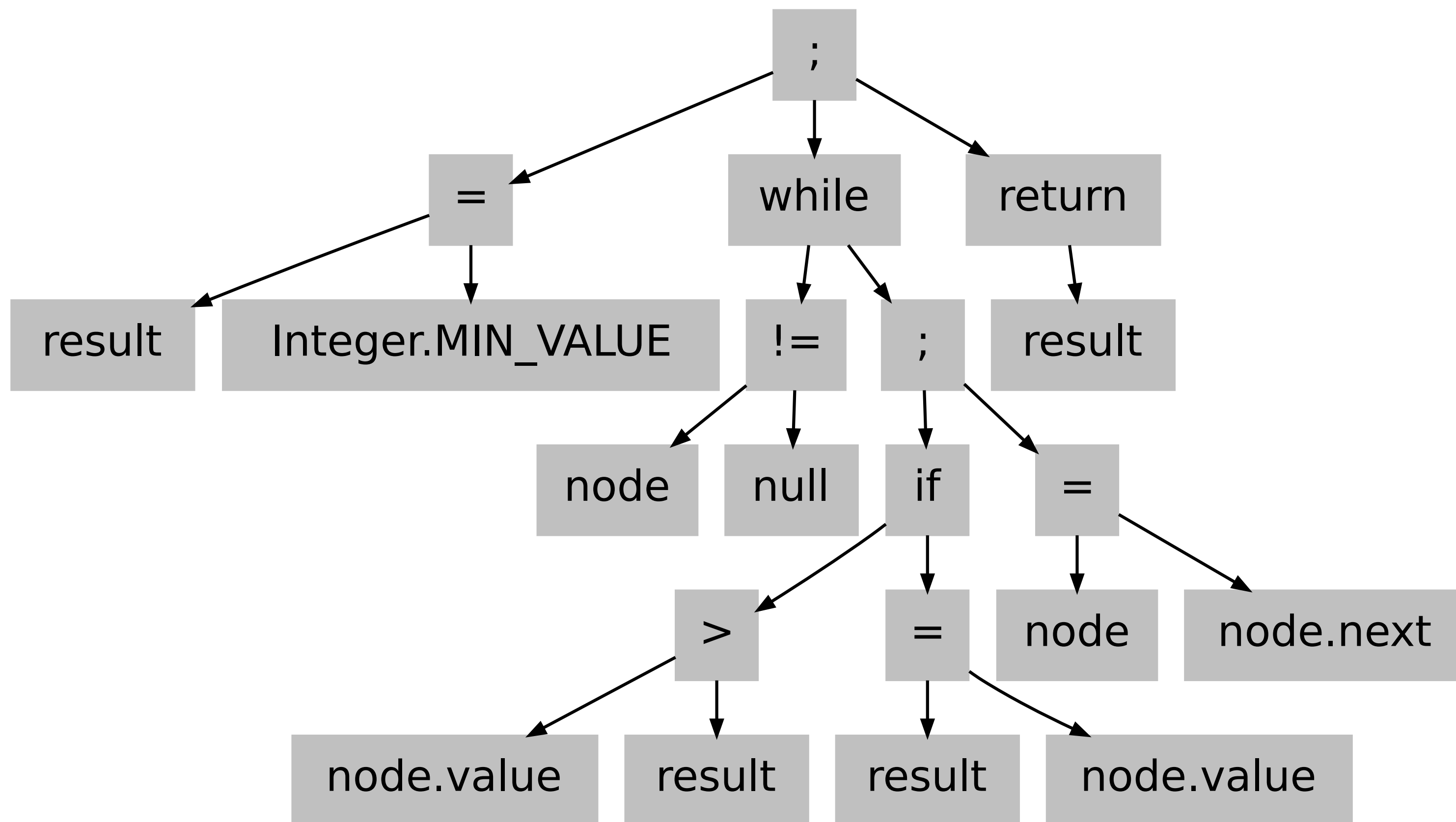
Получаем AST

```
static int max(Node node) {
  var result = Integer.MIN_VALUE;
  while (node != null) {
    if (node.value > result) {
      result = node.value;
    }
    node = node.next;
  }
  return result;
}
```

Получаем AST

- AST, abstract syntax tree, абстрактное синтаксическое дерево
- Компиляторы парсят текст в AST
- Затем анализируют AST
- AST превращается в байт-код
- В TeaVM всё наоборот

Получаем AST

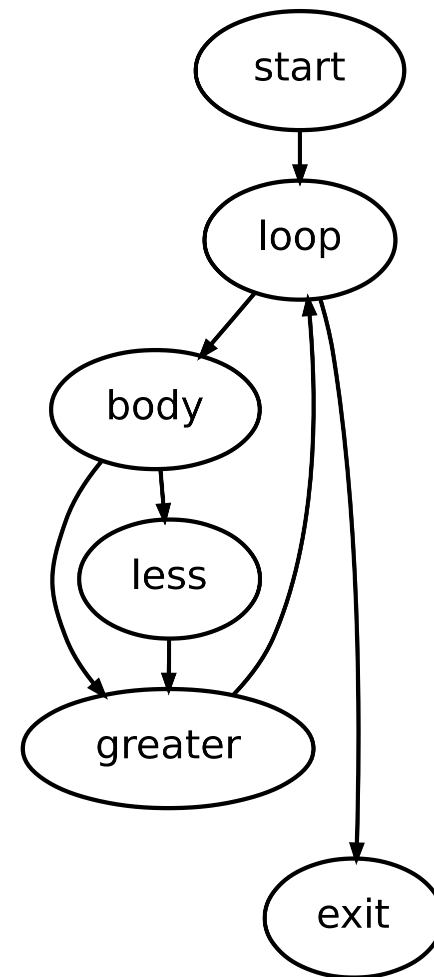


Получаем AST

```
$start
    @2 = -2147483648
    goto $loop
$loop
    if @node == null then goto $exit else goto $body
$body
    @3 = field Node.value @node as I
    @4 = @2 compareTo @3 as int
    if @4 < 0 then goto $less else goto $greater
$less
    @2 = field Node.value @node as I
    goto $greater
$greater
    @node = field Node.next @node as `LNode;`
    goto $loop
$exit
    return @2
```

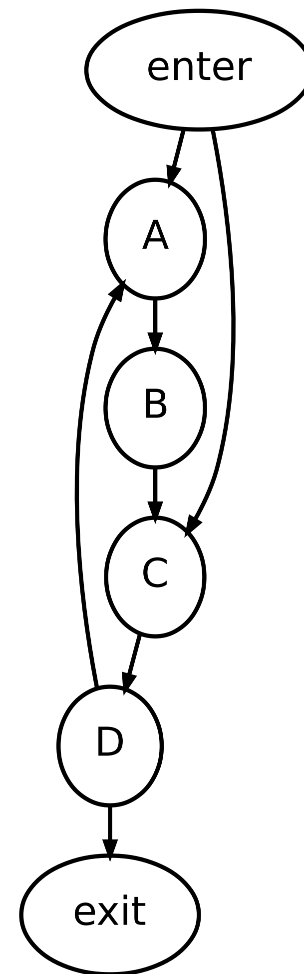
Получаем AST

- Пусть кусок IR между двумя метками будет блоком
- Нарисуем стрелочку из блока A в блок B, если есть переход из блока A в блок B (условный или безусловный)
- Получили CFG (control-flow graph, граф потока управления)
- CFG используется к компиляторах, статических анализаторах, IDE и т.д.
- Задача — CFG преобразовать в AST



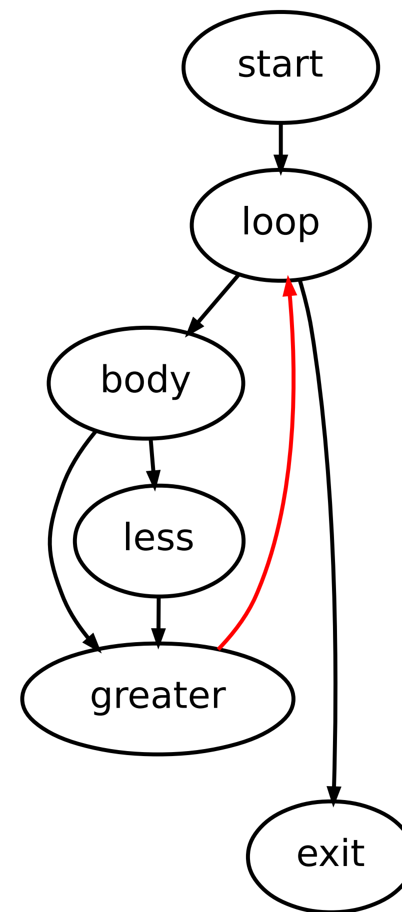
Получаем AST

- CFG: приводимые и неприводимые
- Неприводимые графы — графы с циклами, в которые есть более одного входа
- Неприводимые графы получаются только из async-методов, корутин, генераторов
- Их можно превратить в приводимые (дорогой ценой)
- Для приводимых можно всегда восстановить AST



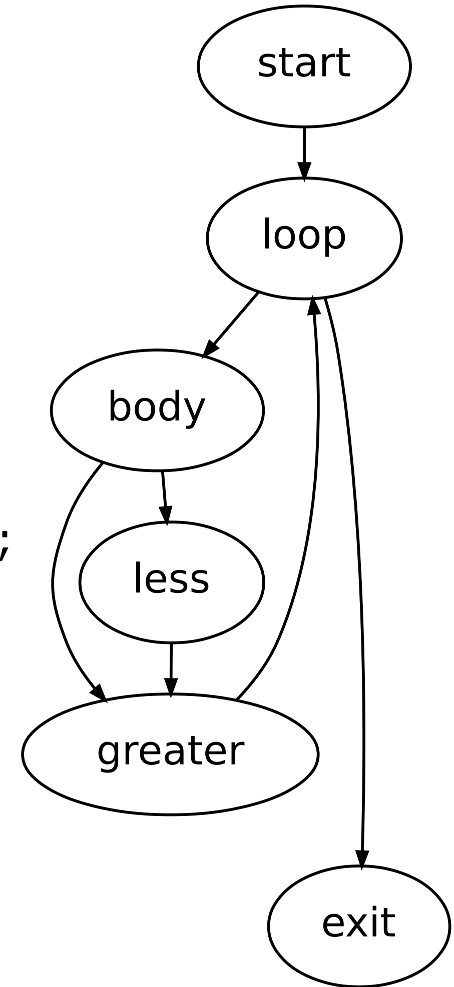
Получаем AST

- На картинке с графом стрелка, ведущая в начало цикла, направлена вверх
- Все остальные — вниз
- Таким граф можно нарисовать всегда
- Стрелки вверх — это while (true)
- Стрелки вниз — это break из блока с меткой
- Оптимизируем



Получаем AST

```
function Node_max($node) {
  var $result;
  $result = (-2147483648);
  loop: while (true) {
    if ($node === null) break loop;
    greater: {
      less: {
        if ($node.$value <= $result) {} else break less;
        break greater;
      } // less
      $result = $node.$value;
      break greater;
    } // greater
    $node = $node.$next;
    continue loop;
  }
  return $result;
}
```



Получаем AST

```
function Node_max($node) {
    var $result;
    $result = (-2147483648);
    while ($node !== null) {
        if ($node.$value0 > $result)
            $result = $node.$value0;
        $node = $node.$next;
    }
    return $result;
}

function Cu(b){var c;c=(-2147483648);while(b!==null)
{if(b.r>c)c=b.r;b=b.L;}return c;}
```

Threads

- WebWorkers — это не потоки, это аналог процессов
- Потоки из UI часто запускают, чтобы тяжёлые вычисления не вешали UI
- Тяжелые вычисления: CPU-bound, IO-bound
- С CPU-bound ничего не поделаешь
- Но IO-bound можно разрулить с помощью green threads
- Green threads — это потоки, которые управляются машиной, а не ОС
- В нашем случае в роли ОС — браузер

Threads

- Coroutine (сопрограмма, корутина) — это процедура, которая может возвращаться в вызывающую процедуру, не доделав работу до конца
- Вызывающая процедура может продолжить выполнение сопрограммы или не делать этого
- Не обязательно продолжать сразу, можно отложить выполнение на потом
- Например, отдавать программы планировщику
- Используем граф вызовов для «заражения» программы async-ами

Threads

```
System.out.println("before");
Thread.sleep(1);
System.out.println("after");
```

Threads

```
$ptr = 0;
if ($rt_resuming()) {
    var $thread = $rt_nativeThread();
    $ptr = $thread.pop();var$1 = $thread.pop();
}
main: while (true) { switch ($ptr) {
    case 0:
        jl_System_out().$println($rt_s(12));
        var$1 = Long_fromInt(1);
        $ptr = 1;
    case 1:
        jl_Thread_sleep(var$1);
        if ($rt_suspending()) break main;
        jl_System_out().$println($rt_s(13));
        return;
}}
$rt_nativeThread().push(var$1, $ptr);
```

Threads

```
$ptr = 0;
if ($rt_resuming()) {
    var $thread = $rt_nativeThread();
    $ptr = $thread.pop();var$1 = $thread.pop();
}
main: while (true) { switch ($ptr) {
    case 0:
        jl_System_out().$println($rt_s(12));
        var$1 = Long_fromInt(1);
        $ptr = 1;
    case 1:
        jl_Thread_sleep(var$1);
        if ($rt_suspending()) break main;
        jl_System_out().$println($rt_s(13));
        return;
}}
$rt_nativeThread().push(var$1, $ptr);
```

Восстанавливаем состояние

Threads

```
$ptr = 0;
if ($rt_resuming()) {
    var $thread = $rt_nativeThread();
    $ptr = $thread.pop();var$1 = $thread.pop();
}
main: while (true) { switch ($ptr) {
    case 0:
        jl_System_out().$println($rt_s(12));
        var$1 = Long_fromInt(1);
        $ptr = 1;
    case 1:
        jl_Thread_sleep(var$1);
        if ($rt_suspending()) break main;
        jl_System_out().$println($rt_s(13));
        return;
}}
$rt_nativeThread().push(var$1, $ptr);
```

Идём до Thread.sleep

Threads

```
$ptr = 0;
if ($rt_resuming()) {
    var $thread = $rt_nativeThread();
    $ptr = $thread.pop();var$1 = $thread.pop();
}
main: while (true) { switch ($ptr) {
    case 0:
        jl_System_out().$println($rt_s(12));
        var$1 = Long_fromInt(1);
        $ptr = 1;
    case 1:
        jl_Thread_sleep(var$1);
        if ($rt_suspending()) break main;
        jl_System_out().$println($rt_s(13));
        return;
}}
$rt_nativeThread().push(var$1, $ptr);
```

Прерываем выполнение и сохраняем состояние, если нужно

Threads

```
$ptr = 0;
if ($rt_resuming()) {
    var $thread = $rt_nativeThread();
    $ptr = $thread.pop();var$1 = $thread.pop();
}
main: while (true) { switch ($ptr) {
    case 0:
        jl_System_out().$println($rt_s(12));
        var$1 = Long_fromInt(1);
        $ptr = 1;
    case 1:
        jl_Thread_sleep(var$1);
        if ($rt_suspending()) break main;
        jl_System_out().$println($rt_s(13));
        return;
}}
$rt_nativeThread().push(var$1, $ptr);
```

Выполняем код после Thread.sleep

- TeaVM — это production-ready
- TeaVM можно использовать, если вашему кроссплатформенному Java-коду необходимо поддержать ещё одну платформу
- TeaVM использует JavaScript как ассемблер
- TeaVM оптимизирует
- Написать свой AOT-компилятор байт-кода не сложно; сложно написать хороший AOT-компилятор байт-кода

Алексей Андреев | Delightex

konsoletyper@gmail.com