Привет! Сегодня я бы хотел поговорить про язык программирования Haxe.

- Поднимите руки те, кто слышал про него
- А есть ли тут флэшеры?
- И ещё немного зарядки для рук :)
    - Поднимите руки те, кто слышал про кросс-компиляторы
    - Теперь те, кто пишет кросс-платформенные приложения (Xamarin, Phonegap, etc.)
    - А как насчёт веб приложений на Node.js?
    - А не на Node.js (PHP, Java, C#)?
    - Мобильные приложения?

Спасибо :) Тогда я начинаю.

Haxe

- Это **не** Hack :)
- Haxe is an open source toolkit based on a modern, high level, strictly typed programming language, a cross-compiler, a complete cross-platform standard library and ways to access each platform's native capabilities

Haxe

- Open source
- Высокоуровневый
- Со строгой типизацией
- Кросс-компилятор
- Кросс-платформенная стандартная библиотека
- Для каждой платформы есть способ использовать её возможности напрямую

Разберём это длинное определение.

Open source — вы можете изучить исходный код языка, чтоб глубже его понять; если чего-то вам не хватает, вы можете это дополнить; есть возможность влиять на развитие языка (почти :) я пробовал протолкнуть одну фишку, но не получилось).

Высокоуровневый — очевидно, что это язык высокого уровня; в принципе и C считается языком высокого уровня (по сравнению с ассемблером), но Haxe более высокоуровневый нежели C (и, наверное, даже C++).

Со строгой типизацией — компилятор строго проверяет типы, сложить строку с числом и умножить на объект не получится :)

Кросс-компилятор — Haxe сам по себе не генерирует бинарных исполняемых файлов, вместо этого он выдаёт либо байт-код (для Flash и Neko VM), либо исходные коды для других языков программирования;

Кросс-платформенная стандартная библиотека — платформ, которые поддерживает Haxe достаточно много, у них всех разные встроенные функции и стандартные библиотеки; чтобы была возможность писать унифицированный код, и при этом не изобретать каждый раз библиотечную функцию с нуля, у Haxe есть своя стандартная библиотека, которая прокидывает вызовы функций в нативную стандартную библиотеку, либо эмулирует эту функцию, если она не доступна нативно;

Для каждой платформы есть способ использовать её напрямую — если не хватает возможностей Haxe, то можно вызывать либо вставлять код на целевой платформе.

Платформы

- Flash, сразу в байткод, минуя ActionScript *(2005)*
- Neko VM, сразу в байткод (*2005)*
- JavaScript *(2006)*
- ActionScript 3 *(2007)*
- PHP *(2008)*
- C++ *(2009)*
- Java *(2012)*
- C# *(2012)*
- Python *(2015)*
- Lua *(2016)*
- PHP7 *(2016)*
- HashLink VM *(2016)*
- C (via HashLink) *(2016)*

Вот список платформ, которые поддерживает Haxe в настоящий момент.

Кто использует Haxe

- BBC
- Disney
- Zynga
- Prezi
- Coca Cola
- Toyota

Хоть Haxe и не мейнстримный язык, его используют довольно крупные и известные компании. Вот небольшой список тех, кто был замечен в использовании Haxe.

Возможно вы знаете некоторых из этих компаний :)

- **1995** — PHP 1.x
- **1998** — PHP 3.x
- **1999** — ES3
- **2000** — C# 1.x, PHP 4.x
- …
- **2004** — PHP 5.x
- **2005** — C# 2.x, **Haxe 1.x**
- …
- **2008** — C# 3.x, **Haxe 2.x** — *generics (только классы), anonymous functions, iterators, properties, ADT (algebraic data type), pattern matching, static extensions, anonymous structures, automatic type inference, macros, DCE*

Прежде мы продолжим, я бы хотел дать небольшую историческую справку.

Haxe зарелизился в 2005 году. В то время PHP дорос до 5й версии, JS был стандарта ES3 и зарелизился C# второй версии. Я затрудняюсь сказать какие возможности были у Haxe в то время, т. к. не смог найти документации по первой версии.

Haxe второй версии зарелизился в 2008 году (тогда же, когда и C# третьей версии). Уже тогда в Haxe были дженерики (правда, только для классов), анонимные функции, итераторы, проперти, алгебраические типы данных, pattern matching, статические расширения, анонимные структуры данных, автоматический вывод типа, макросы и удаление неиспользуемого кода.

- **2009** — Node.js, CofeeScript, ES5
- **2010** — C# 4.x, Rust
- **2011** — Dart
- **2012** — TypeScript, Elm, C# 5.x
- …
- **2015** — ES6, C# 6.x, PHP 7.x, **Haxe 3.x** — *generics (функции), array accessors, array comprehension, abstract types*
- **2016** — ES7
- **2017** — Kotlin, C# 7.x

Только годом позже появился Node.js (это важно, если бы нода появилась на пару лет раньше, возможно, история развития Haxe пошла бы по другому пути).

Haxe 3-й версии зарелизился в 2015 году. Крупных нововведений (заметных внешне) не было, язык стабилизировался. Добавились дженерики для функций, array accessor-ы, такая интересная штука как array comprehension, и, наверное, самая интересная вещь абстрактные типы (в мире Haxe это означает совсем не то, что в мире других языков, я расскажу про них позже)

Для чего используют

- Игры (особенно кросс-платформенные, когда надо зашарить кодовую базу между Web / Mobile / Desktop)
- Web (можно было шарить кодовую базу между сервером и клиентом, ещё до того, как Node.js стал модным)
- Мобильные приложения
- Десктопные приложения
- CLI-приложения
- Кросс-платформенные библиотеки

Игры — почему не Xamarin или Unity? Потому что с ними получаются очень большие билды (и не очень поворотливые). Это всё потому, что код там пишется на C#, а потом, из IL-кода генерируется либо нативный код, либо javascript (и то только в Unity, Xamarin так не умеет). Процесс примерно такой же, как emscripten переводит машинный код в javascript (кстати, может Unity так и делает).

Haxe же генерирует более оптимальный код для целевой платформы, билды небольшие и шустрые.

В 2017 году для веба, мобильных и десктопных приложений есть много других решений, но Haxe это всё тоже может (ещё до того, как Node.js стал модным).

И ещё одна интересная область применения - написание кросс-платформенных библиотек, когда вы пишите либу на Haxe, а затем компилируете её, допустим, в JS и PHP.

Зачем был написан Haxe

- Единый язык (это был 2005 год)
- Было:
  - PHP или Java для server-side
  - JS для dynamic html
  - ActionScript для графической части (на Flash)
- Хотели чтоб стало:
  - Haxe для всего

2005 год - ноды ещё не было, флэш был ещё на коне.

Nicolas Cannasse / The Essential Guide to Open Source Flash Development

- Создать язык более мощный, чем ActionScript 2 или ActionScript 3
- Сделать так, чтобы было легко портировать приложения с ActionScript на Haxe
- Сделать язык, поддерживающий разработку для Flash 6, 7, 8 и 9
- Сделать язык, поддерживающий разработку для JavaScript / AJAX
- Сделать язык, поддерживающий разработку серверно-ориентированных программ, вместо PHP или Java

В книге, соавтором которой является Николас Кеннеси (автор Haxe) приводятся такие первоначальные цели создания Haxe.

О компиляторе

- Написан на OCaml
- Один frontend, множество backend
- Frontend - parsing, AST, type checking, macro, general optimizations, DCE
- Backend - имея на руках AST надо выдать код для целевой платформы (исходный код, байт код, может в будущем и машинный код)
- Есть режим сервера, для поддержки автодополнения кода в IDE (так же в этом режиме поддерживается кэш для для уменьшения времени компиляции)

Компилятор — оптимизирующий

- Подстановка функций (инлайнинг)
- Свёртка констант
- Удаление мёртвого кода (DCE)
- С Haxe 3.3 — static analyzer (дополнительные compile-time оптимизации)

Производительность (vs написанный руками код для целевой платформы)

- Flash — байткод сгенерированный Haxe более оптимизированный (и работает быстрей), чем байткод сгенерированный Action Script Compiler из Flex SDK либо из Flash
- JS — сравнимо с нативным кодом
- C++ — немного медленней, но тоже сравнимо с нативным кодом (только надо учитывать, что в Haxe есть GC)

Рандомный кусок кода

```
var prev = clamp(last - 1);
var next = clamp(last + 1);
var newList : Array<Int> = Random.shuffle([prev, next]);

for (i in index ... (index + list)) {
    var sector = list[i % list];

    if (sector != last
        && sector != prev
        && sector != next
    ) {
        newList.push(sector);
    }
}
```

Возьмём рандомный кусок кода, и посмотрим, что получится после компиляции в JS и в C++.

JavaScript

```
var prev = (this.last - 1 + this.conf.sectors) % this.conf.sectors;
var next = (this.last + 1 + this.conf.sectors) % this.conf.sectors;
var newList = Random.shuffle([prev, next]);
var _g1 = this.index;
var _g = this.index + this.list.length;

while (_g1 < _g) {
    var i = _g1++;
    var sector = this.list[i % this.list.length];

    if (sector != this.last
        && sector != prev
        && sector != next
    ) {
        newList.push(sector);
    }
}
```

JS — вообще говоря нормально :) Сравнимо с тем, что выдаёт Babel.

- Читаемо
- Работает быстро
- Функция clamp() заинлайнена
- Итератор по range преобразован в простой цикл (почему это важно — дело в том, что цикл for в Haxe работает только с итераторами, и если компилировать это в лоб, то будет медленно и хавать память)

C++

```
int prev = hx::Mod(((this->last - (int)1) + ((int)(this->conf-
>__Field(HX_("sectors", 0d, 96, dc, 5d), hx::paccDynamic)))), ((int)(this->conf-
>__Field(HX_("sectors", 0d, 96, dc, 5d), hx::paccDynamic))));
int next = hx::Mod(((this->last + (int)1) + ((int)(this->conf-
>__Field(HX_("sectors", 0d, 96, dc, 5d), hx::paccDynamic)))), ((int)(this->conf-
>__Field(HX_("sectors", 0d, 96, dc, 5d), hx::paccDynamic))));
::Array<int> newList = ::Random_obj::shuffle(::Array_obj<int>::__new(2)->init(0,
prev)->init(1, next));
{
    int _g1 = this->index; int _g = (this->index + this->list->length);
    while ((_g1 < _g)) {
        _g1 = (_g1 + (int)1); int i = (_g1 - (int)1);
        ::Array<int> sector = this->list; int sector1 = sector->__get(hx::Mod(i,
this->list->length));
        bool _hx_tmp1; bool _hx_tmp2;
        if ((sector1 != this->last)) { _hx_tmp2 = (sector1 != prev); } else
{ _hx_tmp2 = false; }
        if (_hx_tmp2) { _hx_tmp1 = (sector1 != next); } else { _hx_tmp1 = false; }
        if (_hx_tmp1) { newList->push(sector1); }
    }
}
```

Наверное видно не очень хорошо :)

- Относительно читаемо (хотя не так хорошо, как в JS)
- Функция заинлайнена (думаю, это происходит во фронтенде Haxe)
- Итератор преобразован в простой цикл

Но есть две не очень оптимальные вещи:

- Для анонимной структуры (this->conf) не был сгенерирован соответствующий класс либо структура, и вместо этого используется динамический объект (наподобие хэш-таблицы); это негативно сказывается на скорости работы; не знаю почему так произошло, ведь информация о том, какие поля есть в анонимной структуре у компилятора есть.
- Какие-то странные преобразования условий, похоже на какую-то самопальную оптимизацию; тоже не уверен зачем такое нужно, может в этом есть смысл, а может надо было оставить оптимизацию условий на откуп компилятору C++

Кратко про язык

- Только про интересные особенности
- Если что, то мануал тут — http://haxe.org/manual/introduction.html
- Классы, интерфейсы, наследование, имплементация, проперти, дженерики и т.д. — всё как везде
- Языки похожие по внешнему виду — Action Script, Type Script

Я не буду пересказывать весь мануал, остановлюсь только на интересных особенностях.

```
class B extend A implements C {
    public static inline var SOME_CONST = 42;
    public var someVar : Int = 0;
    public var propDefGetterNoSetter(default, null) : String = "42";
    public var propGetterSetter(get, set) : Int;

    public function new(someVar : Int) {
        this.someVar = someVar;
    }

    private function get_propGetterSetter() : Int {
        return someVar;
    }

    private function set_propGetterSetter(value : Int) : Int {
        someVar = value;
        return value;
    }
}
```

Небольшой пример кода (наверное, тоже не очень хорошо видно). Тут есть:

- Класс
- Наследование
- Реализация интерфейса
- Константа
- Атрибут класса
- Проперти с геттером по умолчанию и без сеттера
- Проперти с кастомным геттером и сеттером
- И конструктор

Тип переменной указывается через двоеточие после имени переменной (в отличие от C / C++ / C# и Java).

В интерфейсе можно объявлять не только методы, но и переменные и свойства

```
interface Placeable {
    public var x : Float;
    public var y : Float;
}
```

Перейдём к интересным особенностям.

В интерфейсе можно объявлять не только методы, но и переменные и свойства.

Это не уникально для Haxe, но многие другие языки себе такого не позволяют :)

Всё — это выражение

- `var a = if (b > c) { d } else { e }`
- `var a = switch (b) { … }`

Из моднейших мейнстримных языков такое есть в Rust и Kotlin. Из старичков — Haskell и старый добрый Lisp. Наверное, ещё и другие.

Вывод типов

```
function foo() {
    return 42;
}

// тоже самое что и

function foo() : Int {
    return 42;
}
```

Тип вычисляется не в момент декларации, а в момент использования

```
function foo() { return null; }
// тип будет Void -> Unknown<0>

var s : String = foo();
// тип foo() будет Void -> String

// var i : Int = foo();
// будет ошибка компиляции
```

Enum / ADT

```
enum Color {
    Red;
    Green;
    Blue;
    Rgb(r : Int, g : Int, b : Int);
}

var color1 = Color.Red;
var color2 = Color.Rgb(50, 50, 50);
```

Ещё одна интересная особенность — это алгебраические типы данных.

Это как обычный enum, только у значения могут быть параметры.

Enum / ADT

```
switch (color) {
    case Red:
        trace("Red");
    case Green:
        trace("Green");
    case Blue:
        trace("Blue");
    case Rgb(r, g, b):
        trace('Rgb(${r},${g},${b})');
}
```

Для того, чтобы работать с такими енумами в Haxe используется конструкция switch. В последнем case используется variable capture, когда в переменные r, g и b подставятся соответствующие значения, хранимые в переменной color.

При этом, если в switch перечислены не все варианты enum-а (и нет default), то компилятор это не скомпилирует. Это позволяет избежать ошибок, когда в enum добавили (или удалили) какое-то значение, а код, который с ним работает забыли поменять.

Anonymous structure

```
var point = { x : 0.0, y : 12.0 };

// if (point.z > 0.0) { … }
// ошибка компиляции

// point.z = 42.0;
// тоже ошибка компиляции
```

Это это НЕ аналог Object из JS, для компилятора это будет строгий тип.

Typedef

```
typedef UserName = String;

typedef Point = {x:Float, y:Float };
```

Можно создавать алиасы к существующим типам (особенно удобно для алиасинга функций и анонимных структур).

Замечу, что для компилятора что String, что UserName будут одним и тем же типом, т.е. если сделать функцию, которая принимает параметрами UserName и Email (оба — тайпдефы на String), то компилятор никак не защитит от ошибки.

Но начиная с Haxe 3-й версии есть возможность добиться такого поведения (и многого другого).

Abstract type

- В Нахе это несёт несколько иное значение, чем в других языках
- Абстрактный тип — это новый тип, на основе некоторого уже существующего

```
abstract BigDecimal(String) {
    inline function new(value : String) { this =
value; }

    @:from public static function fromInt(value : Int) {
return new BigDecimal(Std.string(value)); }

    @:op(-A) public static inline function neg(value :
BigDecimal) : BigDecimal { … }

    @:op(A + B) public static inline function add(a :
BigDecimal, b : BigDecimal ) : BigDecimal { … }

    @:arrayAccess public inline function arrayRead(k :
Int) { return this.charAt(k); }
}
```

Например, мы хотим сделать тип BigDecimal, на основе строки (ну, по каким-то причинам). В других языках мы бы сделали класс BigDecimal, а в нём приватную переменную типа String со значением. И каждый раз, когда бы мы использовали этот наш класс, в памяти бы был инстанс класса, и инстанс строки.

В Haxe есть возможность избежать лишних аллокаций путём так называемых абстрактных типов. При их использовании память будет выделяться только для основного типа (на базе которого построен абстрактный). Но при этом работать с ними можно так, как будто это настоящие классы.

Так же Haxe не даст автоматически сконвертировать один тип в другой, даже если у них одинаковый базовый тип (это к вопросу, можно ли сделать type-safe функцию, которая принимает username и email, оба - строки — да, можно, и без лишних накладных расходов).

Для абстрактных типов можно перекрывать операторы приведения типа, арифметические и логические операторы, а так же давать этому типу вести себя как массив (аналог в плюсах - перекрыть оператор квадратные скобки).

Abstract enum

```
@:enum
abstract TutorialStep(Int) {
    var None = 0;
    var MsgReachOppositeRow = 1;
    var TryMove = 2;
    var TryHorizontalWall = 3;
    var TryVerticalWall = 4;
    var MsgCantBlock = 5;
}
```

Ещё один интересный наворот — это абстрактный enum. Имеет смысл использовать для минимизации аллокаций памяти (т.к. на настоящий enum память будет аллоцироваться), либо для организации ограниченного количества значений от внешнего API (в строках или интах), например, можно сделать абстрактный enum HTTP-статусов.

В сгенерированном коде будет не настоящий enum, а просто Int, но при этом сохраняются все фишки pattern matching-а (про него чуть позже).

Structural subtyping

```
typedef Iterator<T> = {
    function hasNext() : Bool;
    function next() : T;
}

function isEmpty<T>(it : Iterator<T>) {
    return !it.hasNext();
}
```

Интересный вид казалось бы динамической фичи, но которая будет строго проверяться компилятором.

В функцию isEmpty() можно подать на вход любой класс или структуру, в которой будут методы hasNext() и next().

Это утиная типизация, она не динамическая, всё будет проверено во время компиляции. Примерно такая штука есть в Go, а ещё, кажется, что-то похожее есть в C# для итераторов.

Array comprehension

```
var a = [for (i in 0...10) i];
// [0,1,2,3,4,5,6,7,8,9]

var i = 0;
var b = [while(i < 10) i++];
// [0,1,2,3,4,5,6,7,8,9]
```

Array comprehension (не знаю как правильно сказать по русски) — возможность заполнять массив значениями во время его создания. Такой небольшой синтаксический сахарок.

Похоже как в питоне (а ещё оно есть в CoffeeScript, но он вышел позже чем Haxe).

Array comprehension

```
var a = [
    for (a in 1 ... 11)
        for (b in 2 ... 4)
            if (a % b == 0)
                a + "/" + b
];

// [2/2,3/3,4/2,6/2,6/3,8/2,9/3,10/2]
```

Можно даже так :)

Static extension

```
class StringExt {
    public static function dup(value : String,
count : Int) : String {
        var sb = new StringBuf();
        for (i in 0 ... count) { sb.add(value); }
        return sb.toString();
    }
}

using StringExt;
trace("Abc".dup(3));

// "AbcAbcAbc"
```

Причём функцию можно продолжать использовать, как статическую, т.е. если, по каким-то причинам не хочется делать using, то можно написать StringExt.dup(“Abc”, 3).

Мне кажется это не очень важно, но nice to have; например, тот же Kotlin так не умеет :)

Pattern matching (Basic)

```
enum Tree<T> {
    Leaf(v : T);
    Node(l : Tree<T>, r : Tree<T>);
}

switch (tree) {
    case Leaf(_): "0";
    case Node(_, Leaf(_)): "1";
    case Node(_, Node(Leaf("bar"), _)): "2";
    case _: "3"; // same as "default"
}
```

Перейдём к одной из наиболее интересных особенностей — сопоставление по шаблону. Частично мы рассматривали его в применении к енумам.

Нижнее подчёркивание означает любое значение. case _ - аналогично default.

Pattern matching (Variable capture)

```
switch (node) {
    case Leaf(s): s;
    case Node(Leaf(s), _): s;
    case _: "none";
}

switch (node) {
    case Node(leafNode = Leaf("foo"), _):
        leafNode;
    case x: x;
}
```

При сопоставлении с шаблоном можно нужные значения захватывать в переменные.

Pattern matching (Structure)

```
switch (struct) {
    case { name: "haxe", rating: "poor" }:
        throw false;

    case { rating: "awesome", name: n }:
        n;

    case _:
        "no awesome language found";
}
```

Можно сопоставлять объекты и анонимные структуры.

Pattern matching (Array)

```
switch (arr) {
    case [2, _]: "0";
    case [_, 6]: "1";
    case []: "2";
    case [_, _, _]: "3";
    case _: "4";
}
```

А так же массивы.

Pattern matching (“Or” patterns)

```
switch (num) {
    case 4 | 1: "0";
    case 6 | 7: "1";
    case _: "2";
}
```

В Haxe нет такой штуки как в C / PHP / Java, когда оператор case проваливается в следующий при отсутствии ключевого слова break.

Эта особенность (“проваливание”) часто приводит к ошибкам, но, иногда, бывает полезно на несколько вариантов выполнить один и тот же код.

Для этого в Haxe можно перечислять несколько вариантов через “или”.

Pattern matching (Guards)

```
switch (arr) {
    case [a, b] if (b > a):
        b + ">" + a;

    case [a, b]:
        b + "<=" + a;

    case _:
        "found something else";
}
```

Эту возможность добавили не так давно (я сам о ней узнал в момент подготовки презентации). Выглядит клёво.

Patter matching (Extractors)

```
var e = TString(“fOo");

var success = switch(e) {
    case TString(_.toLowerCase() =>
"foo"): true;

    case _: false;
}
```

Небольшой синтаксический сахарок, дабы не писать лишнего кода — некоторые простые преобразования со значением можно делать сразу во время pattern matching-а.

Macros

- В compile-time можно менять AST программы
- Похоже на annotation processing в Java, только гибче

Самая интересная особенность Haxe — это макросы. В compile-time можно менять AST программы.

Это похоже на annotation processing в Java, только гибче. В Java во время annotation processing-а можно только создавать новые классы, или с помощью определённого хака заменять существующие. В Haxe же можно полностью трансформировать AST.

Кажется, такая же возможность есть в Rust.

Macros

```
macro static function dup(e : Expr) {
    return {
        expr: EBinop(OpAdd, e, e),
        pos: e.pos
    };
}

var x = 0; trace(dup(x++));

// преобразуется в
var x = 0; trace((x++) + (x++));
```

Macro reification

```
macro static function dup(e : Expr) {
    return macro $e + $e;
}

// результат будет аналогичен
// предыдущему слайду
```

Выдавать в макросах нужный AST — это много писанины.

Чтобы упростить написание макросов, в Haxe есть macro reification. Когда вы в макро-функции используете оператор macro, то его результатом будет не вычисленное выражение, а кусок AST.

Примеры

- **continuation**, **async** — async / await на макросах
- **HxSL** — шейдеры, пишутся на DSL, затем компилируются либо в Flash/AGAL, либо в GLSL
- **polygonal-printf** — printf-style форматирование, код генерируется в compile-time
- **json2object** — json-парсер генерится в compile-time под нужный объект, без рефлексии (значит что DCE не будет убирать ничего лишнего)
- Подстановка номера билда в код

На макросах можно делать очень много интересных вещей. Я рандомно выбрал несколько библиотек.

Haxe в настоящем

- Игры - Web / iOS / Android / Windows / macOS / Linux / etc
- Кросс-платформенные библиотеки
- В существующих проектах на любых языках, код для которых может генерировать Haxe

Конечно, я не жду что завтра вы срочно начнёте переводить все свои проекты на Haxe :) Но вот пару примеров того, где мне кажется, что использование Haxe в этом и следующих годах более чем оправдано.

Это кросс-платформенные игры и кросс-платформенные библиотеки.

Ещё можно использовать Haxe в существующий проектах на любых языках, код для которых может генерировать Haxe.

К примеру, знаю одну геймдев-контору, в которой по историческим причинам мобильный движок написан на Xamarin / C#, а Web-движок, на JS. Так вот, они стали использовать Haxe, и унифицировали всю игровую логику.

Ещё, возможно, есть какой-нибудь web-проект не на ноде, а на PHP, C# или Java. И на нём нужна общая логика для бэкэнда и фронтэнда - тут тоже может помочь Haxe.

Спасибо за внимание, всем Нахе. Задавайте ваши вопросы, и я постараюсь на них ответить.