HAXE

мультиплатформенный объектно-ориентированный язык программирования высокого уровня

Языки и платформы

Универсальность Haxe в том, что код на одном языке можно компилировать для нескольких платформ:

- **Flash** - Haxe предлагает очень хорошую производительность и языковые возможности для Flash разработки.
- **JavaScript** - клиентские скрипты .js в т.ч. использующие технологию Ajax.
- **NekoVM** - компилированные программы, исполняемые виртуальной машиной neko, которые могут использоваться как серверные файлы для динамических веб-приложений или как основа программ для командной строки или рабочего стола.

- **PHP** - позволяет использовать высокоуровневый строго-типизированный язык - Haxe, сохраняя при этом полную совместимость с имеющейся серверной платформой и библиотеками.
- **C++** - можно генерировать и C++ код из ваших Haxe программ, с необходимыми make-файлами. Эта целевая платформа активно используется для создания десктопных и мобильных приложений.
- **C#** и **Java** – в их случае также из Haxe генерируется исходный код на этих языках, который затем компилируется.
- **Python** - начиная с версии 3.2.0 добавилась возможность генеровать и код на языке Python.

Для чего используется Haxe?

- Игры, интерактивная графика
- Утилиты командной строки
- Веб разработка (Frontend и Backend)
- Разработка библиотек
- «Клей» между разными платформами/языками

Основная идея Нахе в том, чтобы дать разработчику выбор лучшей платформы для его задачи. Обычно это нелегко сделать, потому что каждая новая платформа идет со своим собственным языком. Нахе же позволяет писать на одном языке код для многих платформ, предоставляя :

- стандартный язык с множеством современных возможностей
- стандартную библиотеку (включая *Date*, *Xml*, *Math*...) которая работает одинаково для всех платформ
- специфичные для платформ библиотеки: весь API каждой конкретной платформы доступен из **Нахе**

С помощью Haxe и связанных с ним технологий (Neko, NME, SysTools, SPOD и т.д.) можно создавать приложения, способные работать под Windows, Mac OS или Linux.

Кроме того язык является открытым
(open source)

ОСНОВЫ HAXE

Синтаксис Haxe подобен Java, ActionScript или C++.

Файл исходного кода состоит из необязательного *названия пакета*, за которым следует несколько описаний импорта (*imports*) и типов.

Для более чёткого разделения, имена пакетов состоят из нескольких *имён-идентификаторов*, начинающихся с символов в нижнем регистре, а *идентификаторы типов* всегда начинаются с заглавных букв.

Система типов

Haxe — статически типизированный язык. Он имеет богатую систему типов, включая классы, интерфейсы, функциональные типы, анонимные типы, алгебраические типы данных, а также абстрактные типы данных.

Основные типы

Существует несколько разновидностей типов. Самые важные из них - *классы (class)* и *перечисления (enum)*.

Вот несколько *основных типов*, определенных в стандартной библиотеке:

enum Void {}

Void определен как *enum*. Enum перечисляет список допустимых *конструкторов*. Пустое перечисление, например *Void*, не может иметь каких-либо значений. Тем не менее, это вполне допустимый тип, который может быть определен и использован.

enum Dynamic<T> {}

Dynamic это перечисление с *параметром типа (type parameter)*.

class Float {}

Float - это класс чисел с плавающей точкой. У него нет никаких методов, так что его реализация может быть существенно оптимизирована на некоторых платформах.

class Int extends Float {}

Int - целое число. У него также нет методов, но он является наследником класса *Float*, поэтому везде, где нужен *Float*, вы можете использовать *Int*, но не наоборот.

```
enum Bool {
    true;
    false;
}
```

Bool - это перечисление, как и *Void*, но у него есть два экземпляра *true* и *false*.

Как видно из этого, даже *стандартные* типы могут быть легко определены с помощью системы типов hAxe. Это также означает, что вы можете определять свои собственные типы.

Классы

В haxe структура классов создаётся следующим образом:

```
package my.pack;
    /*
        так создаётся определение класса
my.pack.MyClass
    */
    class MyClass {
        // ....
    }
</div>
```

У Класса может быть несколько *переменных* и *методов*.

```
package my.pack;

class MyClass {

        var id : Int;

        static var name : String = "MyString";

        function foo() : Void {
        }

        static function bar( s : String, v : Bool ) : Void {
        }
    }
</div>
```

Переменные и методы могут иметь следующие *флаги* :

•**static** : поле принадлежит самому Классу, а не его *экземплярам (объектам)*. Статические идентификаторы могут использоваться напрямую в самом классе. Вне класса, они должны быть использованы с именем, класса, которому принадлежат (например : *my.pack.MyClass.name*).

• **public** : поле открыто для доступа из других классов. **По умолчанию, все поля - *private***.

• **private** : доступ к полю разрешён только из самого класса и из классов, которые от него наследуют.

Все *переменные класса* **должны** быть типизированными, т.е. объявлены с типом (можно использовать тип *Dynamic* если неизвестно, какой тип следует использовать или хочется писать в стиле языков с динамической типизацией).

Аргументы функций и возвращаемые значения необязательны, но их типы также строго проверяются, как мы увидим позже во введении в *выявление типов*.

Статические переменные *могут* получать значение при объявлении, но это не обязательно.

Конструктор класса.

У класса может быть только один конструктор, представляющий из себя нестатическую функцию с именем *new*.

```
class Point {
        public var x : Int;
        public var y : Int;

        public function new() {
            this.x = 0;
            this.y = 0;
        }
    }
</div>
```

Перечисления

Перечисляемые типы — это ключевая особенность языка. Перечисления могут иметь собственные параметры, а также быть рекурсивными.

Строго говоря, это правильные типы-суммы, при условии, что типы-произведения, включенные в них, обязаны быть определены внутри этих типов-сумм.

Это значит, что перечисления не просто именованные «магические числа», как в большинстве языков, ими можно элегантно решать сложные архитектурные проблемы.

Пример перечисления *Color* с тремя *константными конструкторами* для *перечисления*. Кроме этого, конструкторам заданы параметры :

```
enum Color1 {
        red;
        green;
        blue;
        grey( v : Int );
        rgb( r : Int, g : Int, b : Int );
    }
</div>
```

Следующие значения все являются допустимыми примерами значений *Color1* :

```
var c1:Color1 = red;
var c2:Color1 = green;
var c3:Color1 = blue;
var c4:Color1 = grey(0);
var c5:Color1 = grey(128);
var c6:Color1 = rgb( 0x00, 0x12, 0x23 );
var c7:Color1 = rgb( 0xFF, 0xAA, 0xBB );
</div>
```

Также мы можем получить рекурсивный тип, например, чтобы добавить *alpha* :

```
enum Color2 {
        red;
        green;
        blue;
        grey( v : Int );
        rgb( r : Int, g : Int, b : Int );
        alpha( a : Int, col : Color2 );
    }
</div>
```

Допустимые примеры значений *Color2* :

```
    var c1:Color2 = alpha( 127, red );
    var c2:Color2 = alpha( 255, rgb(0,0,0) );
</div>
```

Определения типов (typedef)

Можно создавать определения типов - некие подобия ярлыков типов, которые могут быть использованы для именования анонимных типов или длинных типов, которые вы не хотите повторять повсюду в своей программе.

Это делается с помощью ключевого слова **typedef**.

Определения типов это не классы, они используются только для типизации.

```
typedef User {
    var age : Int;
    var name : String;
}
// ....
var u : User = { age : 26, name : "Tom" };
</div>


// PointCube это трёхмерный массив точек
typedef PointCube = Array<Array<Array<Point>>>
</div>
```

Функции

Функциональные типы являются объектами первого класса в Haxe.

Когда вы хотите определить тип функции, можно сделать это перечислением аргументов и возвращаемого типа в конце, разделёнными стрелками.

Например, **Int -> Void** это тип функции принимающей аргумент *Int* и возвращающей *Void*.

А **Color -> Color -> Int** принимает два аргумента типа *Color* и возвращает *Int*.

```
class C {
        function f(x : String) : Int {
            // ...
        }

        function g() {
            type(f); // выводит String -> Int
            var ftype : String -> String = f;
            // ошибка , должно быть String -> Int
        }
    }
</div>
```

Неизвестный тип (Unknown)

Когда тип не объявлен, используется тип *Unknown*. Но как только первый раз используется другой тип, тип меняется на него. Идентификатор выводящийся с типом *Unknown* используется для различения нескольких неизвестных при переводе в строку сложного типа.

```
function f() {
        var x;
        type(x); // выводит Unknown<0>
        x = 0;
        type(x); // выводит Int
    }
</div>
```

Dynamic

Если вы хотите использовать *динамическую типизизацию* и освободиться от системы типов, можно использовать тип *Dynamic*, который может быть использован вместо *любого* типа данных без каких-либо проверок компилятором.

```
var x : Dynamic = "";
    x = true;
    x = 1.744;
    x = new Array();
</div>
```

Также, *Dynamic* имеет бесконечное количество полей, имеющих тип *Dynamic*, то есть он может быть использован как массив с помощью синтаксиса с квадратными скобками и т.д.

В то время, как это может быть иногда полезным, необходимо соблюдать осторожность и не использовать слишком много динамических переменных, т.к. это может нарушить безопасность программы.

Любой класс может также *реализовывать* Dynamic с параметром типа или без него. В этом случае, существуют поля класса типизированы, а иначе они имеют динамический тип :

```
class C implements Dynamic<Int> {
        public var name : String;
        public var address : String;
    }
    // ...
    var c = new C();
    var n : String = c.name; // ok
    var a : String = c.address; // ok
    var i : Int = c.phone; // ok : используется Dynamic
    var co : String = c.country // ошибка : должен быть Int, так как Dynamic<Int>
</div>
```

АРХИТЕКТУРА

Самым значимым аспектом разработки архитектуры Haxe было решение о поддержке Adobe Flash, JavaScript и серверных приложений единой кодовой базой.

Компилятор

Компилятор Haxe разделен на один фронтенд и множество бэкэндов.

Фронтенд отвечает за парсинг и проверку типов, применение макросов, общую оптимизацию, различные трансформации кода и создания промежуточного представления кода в виде абстрактного синтаксического дерева (АСД).

Каждый из бэкендов отвечает за трансляцию этого АСД в исходный код или байткод целевой платформы.

Компилятор написан на OCaml.

Он может быть запущен в режиме сервера для поддержки автодополнения кода в IDE, также в этом режиме поддерживается кэш для уменьшения времени компиляции.

Производительность

Компилятор Haxe — оптимизирующий компилятор, также использующий подстановку функций, свертку констант, удаление мёртвого кода (DCE) для оптимизации производительности скомпилированных программ.

ActionScript 3:

Программы, скомпилированные на Haxe, обычно быстрее, чем программы, скомпилированные с помощью Flex SDK ActionScript Compiler. Однако, используя ActionScript Compiler 2 (ASC2) и грамотно составленный код, можно добиться сравнимой производительности.

JavaScript:
Программы, скомпилированные на Haxe, сравнимы по скорости с программами, написанными просто на JavaScript. OpenFL — самый распространенный фреймворк, который можно запустить на HTML5/ JavaScript, но приложения, построенные с помощью этого фреймворка, на текущий момент страдают от потерь производительности на мобильных устройствах.

C++:

Программы, скомпилированные на Haxe, работают почти также быстро, как и написанные просто на C++, но приложения, построенные с помощью OpenFL, могут страдать от потерь производительности.

Основные пользователи Haxe — это TiVo, Prezi, Nickelodeon, Disney, Mattel, Hasbro, Coca Cola, Toyota и BBC. OpenFL и Flambe — популярные Haxe фреймворки для создания мультиплатформенного контента и программ из единой кодовой базы.

В связи с всё большим вытеснением технологии Adobe Flash в последние годы в пользу HTML5, Haxe, Unity и другие кроссплатформенные инструменты уделяют последнему всё больше времени, сохраняя обратную поддержку с Adobe Flash Player.

Как попробовать HaXe?

Можно легко и быстро попробовать Haxe, ничего не устанавливая, с помощью он-лайн сервиса TryHaxe! - http://try.haxe.org/, который поможет и написать код с помощью удобного редактора и скомпилирует его и покажет результат.

Загрузить и установить Haxe можно на официальном сайте - http://haxe.org.
Там же есть инструкции по ручной установке и сборке из исходников, если это необходимо.

Спасибо за внимание!