



# Поддержка типа `dynamic` в JVM

Алексей Степанов

Научный руководитель: Андрей Бреслав

САНКТ-ПЕТЕРБУРГСКИЙ АКАДЕМИЧЕСКИЙ УНИВЕРСИТЕТ

10 ноября 2016 г.



## Типизация в языках программирования

- Статическая типизация
- Динамическая типизация
- Постепенная типизация



## Постепенная типизация

Постепенная типизация - система типов, в которой часть переменных и выражений может быть типизированна, и их корректность проверяется в момент компиляции, а часть может быть не типизированна, и об ошибках в них мы узнаем во время исполнения.

## Преимущества

- eval
- DSL (Gradle)
- DOM



## Удобный доступ к полям и методам

```
dynamic x = dom.html.body.tables.main.tr.td;
```

## Почему не Object?

```
Scriptobj.SetProperty("Cnt", ((int)GetProperty("Cnt"))+1);  
scriptobj.Cnt += 1;
```



### Варианты постепенной типизации

От динамической типизации	От статической типизации
Python	C#
JavaScript	Scala
Groovy	Kotlin?



### Python: Type Hints

```
def greeting(name: str) -> str:  
    return 'Hello ' + name
```

```
AnyStr = TypeVar('AnyStr', str, bytes)  
def concat(x: AnyStr, y: AnyStr) -> AnyStr:  
    return x + y
```



### JavaScript: Flow

```
// @flow
function bar(x): string {
  return x.length;
}
bar('Hello, world!');
```

### Flow output

```
3:   return x.length; - number. This type is incompatible
with the expected return type of
2: function bar(x): string { -string
```



### Scala

```
class MyRouter extends Dynamic{
  def selectDynamic(name: String): T = {}
  def updateDynamic(name: String)(value: T): Unit = {}
  def applyDynamic(methodName: String)(args: Any*) {
    println(s"You called '$methodName' method with " +
      s"following arguments: ${args mkString ", "}")
  }
  def applyDynamicNamed(name: String)
    (args: (String, Any)*) {
    println(s"You called '$name' method with " +
      s"following arguments:
      ${args map (a=>a._1+"="+a._2) mkString ", "}")
  }
}
```





### Scala

- + Поддержка в ScalaJS
- + Возможность написать свою логику диспетчеризации
- Нет стандартных реализаций
- Своя логика будет скорее всего медленная



### C#

```
dynamic obj = new MyObject();  
obj.anyMethod(53);
```

### Своё разрешение как в Scala

```
public class MyDynamicImpl : DynamicObject{  
    public override bool TryGetMember(  
        GetMemberBinder binder, out object result){}  
    public override bool TrySetMember(  
        SetMemberBinder binder, object value){}  
}
```



### C# ExpandoObject

```
XElement contactXML =  
    new XElement("Contact",  
        new XElement("Name", "Patrick Hines"),  
        new XElement("Phone", "206-555-0144"),  
        new XElement("Address",  
            new XElement("Street1", "123 Main St"),  
            new XElement("City", "Mercer Island"),  
            new XElement("State", "WA"),  
            new XElement("Postal", "68042")  
        )  
    );
```

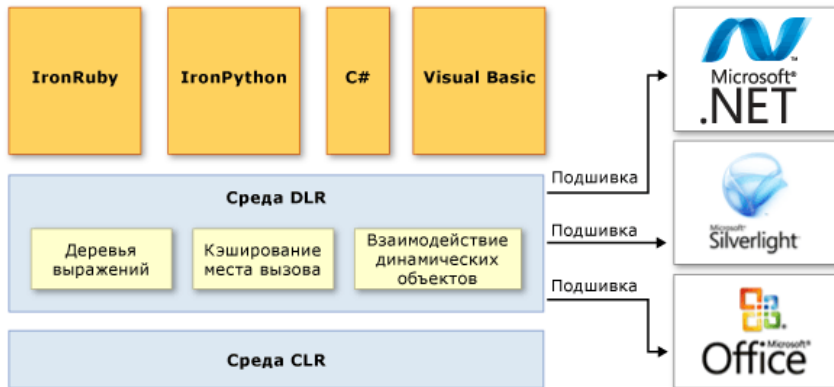


### C# ExpandoObject

```
dynamic contact = new ExpandoObject();  
contact.Name = "Patrick Hines";  
contact.Phone = "206-555-0144";  
contact.Address = new ExpandoObject();  
contact.Address.Street = "123 Main St";  
contact.Address.City = "Mercer Island";  
contact.Address.State = "WA";  
contact.Address.Postal = "8402";
```



### Dynamic Language Runtime





## Возникающие вопросы

- Где хранить информацию о типе?
- Как хранить динамически типизированный объект?
- Как выбирать перегрузки при вызове методов?
- Как сделать всё оптимально и быстро?



- В динамических языках возможна перегрузка только по числу параметров
- В статически типизированных языках у нас доступна перегрузка по типам аргументам

### Возникающий вопрос

- Можем ли мы при постепенной типизации делать во время исполнения всё то же что мы делали во время компиляции?



- В динамических языках возможна перегрузка только по числу параметров
- В статически типизированных языках у нас доступна перегрузка по типам аргументам

### Возникающий вопрос

- Можем ли мы при постепенной типизации делать во время исполнения всё то же что мы делали во время компиляции?
- Ответ: Нет.
  - Это дорого
  - Это невозможно





### Groovy

1. Получить список всех методов с подходящим именем
2. Удалить все методы которые не подходят к данному вызову
3. Если осталось больше одного метода, то вычислить метрику на на методах
4. Метод с наименьшим значением метрики выигрывает
5. Если имеется несколько методов с наименьшей дистанцией, то выдать исключение



### Groovy

```
static foo(Object o, Integer i) { "Integer won" }  
static foo(String s, Object o) { "String won" }  
  
assert foo("potato", new Integer(6)) =~ "Integer"
```

### Полученная дистанция

- $\rho(\text{String}, \text{Object}) + \rho(\text{Integer}, \text{Integer}) == 1 + 0 == 1$
- $\rho(\text{String}, \text{String}) + \rho(\text{Integer}, \text{Object}) == 0 + 2 == 2$   
(Integer  $\rightarrow$  Number  $\rightarrow$  Object)



@CompileStatic разрешает типы только для локальных переменных

```
def foo(Object x){1}
def foo(String x){2}
def create(){ return "" }
```

```
@CompileStatic
def bar() {
    assert foo("") == 2
    assert foo(new Object()) == 1
    assert foo(create()) == 1
}
```

```
bar()
```



### C#

- Во время компиляции мы проверяем что во время исполнения может подойти хотя бы один метод
- По алгоритму похожему на используемый во время компиляции определяем лучший метод
- Разницу между Object и dynamic не разрешаем



### JSmall - динамические теги типа

```
public void XMLWriter::write(int e) { y}
public void XMLWriter::write(Integer e) { y}
public void XMLWriter::write(XMLElement e) { y}
public void XMLWriter::write(StructuredXMLElement e) { y}
public void XMLWriter::write(Serializable e) { y}
w:='XMLWriter' asJavaClass new.
i:= 10 type: 'int'.
obj :='StructuredXMLElement' asJavaClass new
      type: 'java.io.Serializable'
w write: i.
w write: obj.
w write: (obj type: 'StructuredXMLElement').
w write: obj.
```



### JSmall - динамические теги типа

- Мы можем помечать переменные тегами типа
- Тег типа используется только для разрешения вызовов, и никак не связан с реальным типом переменной
- Все значения полученные из Java автоматически маркируются тегом типа



### Clojure

```
public class Base {}  
public class SupplierRouter {  
    public Base getBaseNullSupplier() {return null;}  
    public int method_(String first) {return 0;}  
    public int method_(Integer first) {return 4;}  
    public int method_(Base first) {return 6;}  
}  
  
(ns test (:import (javacl SupplierRouter)))  
(def br (new SupplierRouter))  
(println (. br method_ (. br getBaseNullSupplier)))
```

Что будет выведено на экран?



### JRuby

```
public int method_(String first) {return 0;}  
public int method_(Object first) {return 1;}  
public int method_(int first) {return 2;}  
public int method_(short second) {return 3;}  
public int method_(Integer first) {return 4;}  
public int method_(Base first) {return 5;}  
public int method_(Derived first) {return 6;}
```

```
br = JavaTestCoreUtils::BasicRouter.new  
br.method_(29)  
br.method_(Integer 29)  
br.method_(2.9)
```





```
public interface I1 { int method1(); }  
public interface I2 { int method1(); }  
class InterfaceProvider implements I1, I2 {}  
public class AmbiguousRouter {  
    public int method_I12(I1 a) { return 2;}  
    public int method_I12(I2 a) { return 3;}  
}
```

## Отличия JRuby Nashorn и Clojure



```
public interface I1 { int method1(); }  
public interface I2 { int method1(); }  
class InterfaceProvider implements I1, I2 {}  
public class AmbiguousRouter {  
    public int method_I12(I1 a) { return 2;}  
    public int method_I12(I2 a) { return 3;}  
}
```

## Отличия JRuby Nashorn и Clojure

- Clojure выбирает метод записанный первым в теле класса
- JRuby выбирает метод записанный последним в теле класса



```
public interface I1 { int method1(); }  
public interface I2 { int method1(); }  
class InterfaceProvider implements I1, I2 {}  
public class AmbiguousRouter {  
    public int method_I12(I1 a) { return 2;}  
    public int method_I12(I2 a) { return 3;}  
}
```

## Отличия JRuby Nashorn и Clojure

- Clojure выбирает метод записанный первым в теле класса
- JRuby выбирает метод записанный последним в теле класса
- Nashorn выбирает какой-то метод, а иногда кидает исключение



### Критерии определения правил выбора перегрузок

- Предсказуемость
- Производительность
- Обратная совместимость при стирании типа в статически типизированном коде
- Схожесть работы нетипизированного кода с типизированным



## Kotlin

- Поддержка dynamic в Kotlin для JavaScript
- В JVM dynamic не поддерживается



## Цель

Поддержка типа `dynamic` в Kotlin для JVM

## Задачи

- Разработка правил разрешения перегрузок
- Исследование возможности поддержки совместимости правил с правилами других JVM языков
- Реализация поддержки в компиляторе языка Kotlin под JVM
- Оценка производительности



- Обзор существующих механизмов поддержки динамических типов в статических языках
- Обзор механизмов разрешения перегрузок в связи динамических и статических языков
- Реализация набора тестов для тестирования разрешения перегрузок разных языков
- Начата разработка правил разрешения перегрузок для языка Kotlin



- Доработать правила динамического разрешения перегрузок
- Смотреть в сторону `invoke dynamic`
- `invoke dynamic` не панацея



Вопросы?