

Для чего полезны макросы?

Евгений Бурмако

École Polytechnique Fédérale de Lausanne
<http://scalamacros.org/>

21 августа 2013

Для чего полезны макросы?

- ▶ Генерация кода
- ▶ Статические проверки
- ▶ Предметные языки

Введение в макросы

Что такое макросы?

- ▶ Экспериментальная функциональность Скалы 2.10.x и 2.11.0
- ▶ Программист пишет функции, использующие Scala Reflection API
- ▶ Компилятор вызывает эти функции во время компиляции

Виды макросов

- ▶ Много способов расширения компилятора → много видов макросов
- ▶ Макротипы, макро-аннотации, нетипизированные макросы и т.д.
- ▶ В официальной поставке 2.10.x и 2.11.0 есть только макрометоды

Макрометоды

```
log(Error, "does not compute")
```



```
if (Config.loggingEnabled)  
  Config.logger.log(Error, "does not compute")
```

- ▶ Раскрывают типизированные термы в типизированные термы
- ▶ Результат может содержать произвольные конструкции Скалы
- ▶ Кодогенератор может выполнять произвольные вычисления

Макрометоды

```
def log(severity: Severity, msg: String): Unit = ...
```

- ▶ Сигнатуры макросов выглядят как сигнатуры обычных методов

Макрометоды

```
def log(severity: Severity, msg: String): Unit = macro impl
```

```
def impl(c: Context)  
  (severity: c.Expr[Severity],  
   msg: c.Expr[String]): c.Expr[Unit] = ...
```

- ▶ Сигнатуры макросов выглядят как сигнатуры обычных методов
- ▶ Тела макросов – ссылки на отдельно определяемые реализации

Макрометоды

```
def log(severity: Severity, msg: String): Unit = macro impl
```

```
def impl(c: Context)
  (severity: c.Expr[Severity],
   msg: c.Expr[String]): c.Expr[Unit] = {
  import c.universe._
  reify {
    if (Config.loggingEnabled)
      Config.logger.log(severity.splice, msg.splice)
  }
}
```

- ▶ Сигнатуры макросов выглядят как сигнатуры обычных методов
- ▶ Тела макросов – ссылки на отдельно определяемые реализации
- ▶ Реализации используют Scala Reflection API для анализа и синтеза

Резюме

```
log(Error, "does not compute")
```



```
if (Config.loggingEnabled)  
    Config.logger.log(Error, "does not compute")
```

- ▶ Макрометоды умеют преобразовывать свои вызовы в новый код
- ▶ Преобразования осуществляются только локально
- ▶ Аргументы должны быть статически типизируемы

Генерация кода

Генерация кода

- ▶ Создание нового кода на лету
- ▶ Более удобно и надежно, чем генерировать текст
- ▶ Пока что невозможно создавать глобально видимые классы

Пример №1: улучшение производительности

```
def createArray[T: ClassTag](size: Int, el: T) = {  
  val a = new Array[T](size)  
  for (i <- 0 until size) a(i) = el  
  a  
}
```

- ▶ На Скале легко писать красивый обобщенный код
- ▶ К сожалению, абстракции зачастую приносят накладные расходы
- ▶ Например, в этом случае трансляция полиморфизма в байткод JVM приведет к боксингу, что сильно ухудшит производительность

Пример №1: улучшение производительности

```
def createArray[@specialized T: ClassTag](...) = {  
  val a = new Array[T](size)  
  for (i <- 0 until size) a(i) = el  
  a  
}
```

- ▶ Методы можно специализировать, но это нудно и тяжело
- ▶ Нудно = приходится специализировать всю цепочку вызовов
- ▶ Тяжело = специализация приводит к дубликации байткода

Пример №1: улучшение производительности

```
def createArray[T: ClassTag](size: Int, el: T) = {  
  val a = new Array[T](size)  
  def specBody[@specialized T](el: T) {  
    for (i <- 0 until size) a(i) = el  
  }  
  classTag[T] match {  
    case ClassTag.Int => specBody(el.asInstanceOf[Int])  
    ...  
  }  
  a  
}
```

- ▶ Хотелось бы специализировать ровно столько, сколько нужно
- ▶ Прямо как в недавней работе [Bridging Islands of Specialized Code](#)
- ▶ Также хотелось бы не писать подобный низкоуровневый код руками
- ▶ И вот здесь оказываются очень полезными макросы!

Пример №1: улучшение производительности

```
def specialized[T: ClassTag](code: => T) = macro ...
```

```
def createArray[T: ClassTag](size: Int, el: T) = {  
  val a = new Array[T](size)  
  specialized[T] {  
    for (i <- 0 until size) a(i) = el  
  }  
  a  
}
```

- ▶ Макрос `specialized` берет красивый код и делает его быстрым
- ▶ Это типичный сценарий применения ускоряющих макросов
- ▶ Иногда такие макросы нетривиальны, но прогресс не стоит на месте

Пример №2: материализация

```
trait Reads[T] {  
  def reads(json: JsValue): JsResult[T]  
}
```

```
object Json {  
  def fromJson[T](json: JsValue)  
    (implicit fjs: Reads[T]): JsResult[T]  
}
```

- ▶ Классы типов – красивый подход к написанию расширяемого кода
- ▶ Вот пример сериализатора из Play, основанного на классах типов

Пример №2: материализация

```
def fromJson[T](json: JsValue)
  (implicit fjs: Reads[T]): JsResult[T]

implicit val IntReads = new Reads[Int] {
  def reads(json: JsValue): JsResult[T] = ...
}
```

```
fromJson[Int](json) // пишем
fromJson[Int](json)(IntReads) // получаем
```

- ▶ Классы типов абстрагируют варьирующиеся части алгоритмов
- ▶ Для каждого используемого типа определяются экземпляры
- ▶ Компилятор автоматически подставляет их в нужные вызовы

Пример №2: без макросов (Скала 2.9.x)

```
case class Person(name: String, age: Int)
```

```
implicit val personReads = (  
  (__ \ 'name).reads[String] and  
  (__ \ 'age).reads[Int]  
) (Person)
```

- ▶ Экземпляры классов типов пишутся вручную
- ▶ Много некрасивого, избыточного кода
- ▶ Есть **альтернативы**, но у них есть свои недостатки

Пример №2: макрометоды (Скала 2.10.0)

```
implicit val personReads = Json.reads[Person]
```

- ▶ Как мы уже видели, избыточный код можно генерировать
- ▶ Байткод будет идентичен байткоду предыдущего примера
- ▶ Поэтому производительность останется на отличном уровне

Пример №2: неявные макросы (Скала 2.10.2+)

// не требуется вообще никакого кода

- ▶ Экземпляры классов типов можно генерировать на лету
- ▶ Поэтому объявлять неявные значения нет необходимости
- ▶ Подход применяется в таких библиотеках как [Pickling](#) и [Shapeless](#)

Пример №2: неявные макросы (Скала 2.10.2+)

```
trait Reads[T] { def reads(json: JsValue): JsResult[T] }  
  
object Reads {  
  implicit def materializeReads[T]: Reads[T] = macro ...  
}
```

- ▶ Поиск неявных значений затрагивает не только область видимости
- ▶ Но также и список полей и методов компаньонов
- ▶ Поэтому неявный макрос, объявленный в компаньоне, будет виден всем, кто захочет использовать наш класс типов.

Пример №2: неявные макросы (Скала 2.10.2+)

```
fromJson[Person](json)
```



```
fromJson[Person](json)(materializeReads[Person])
```



```
fromJson[Person](json)(new Reads[Person]{ ... })
```

- ▶ Если экземпляр `Reads[T]` не находится, будет вызываться макрос
- ▶ Макрос посмотрит на значение `T` и сгенерирует нужный экземпляр
- ▶ Подробности можно почитать в [слайдах другого выступления](#)

Пример №3: поставщики типов

```
println(Db.Coffees.all)  
Db.Coffees.insert("Brazilian", 99, 0)
```

- ▶ F# позволяет генерировать обертки вокруг источников данных
- ▶ Будучи статически типизированными, эти обертки приятны и удобны
- ▶ Можно ли что-то похожее реализовать на макрометодах?

Пример №3: поставщики типов

```
def h2db(connString: String): Any = macro ...  
val db = h2db("jdbc:h2:coffees.h2.db")
```



Пример №3: поставщики типов

```
def h2db(connString: String): Any = macro ...  
val db = h2db("jdbc:h2:coffees.h2.db")
```



```
val db = {  
  trait Db {  
    case class Coffee(...)  
    val Coffees: Table[Coffee] = ...  
  }  
  new Db {}  
}
```

- ▶ Макрометоды раскрываются локально
- ▶ Поэтому максимум, чего можно добиться в данном случае, это набор локальных классов, которые не видны извне

Пример №3: поставщики типов

```
scala> val db = h2db("jdbc:h2:coffees.h2.db")
db: AnyRef {
  type Coffee { val name: String; val price: Int; ... }
  val Coffees: Table[this.Coffee]
} = $anon$1...

scala> db.Coffees.all
res1: List[Db$1.this.Coffee] = List(Coffee(Brazilian,99,0))
```

- ▶ К счастью, локальные классы стираются в структурные типы
- ▶ Поэтому все работает как надо (статическая типизация, IDE)
- ▶ Единственная проблема - накладные расходы на рефлексия

Пример №3: поставщики типов

```
class Coffee(row: Row["...".type]) with Dynamic {  
  def selectDynamic = macro ...  
}
```

```
db: AnyRef{type Coffee <: Dynamic; ...}  
coffee.name // преобразуется в: coffee.selectDynamic("name")
```



```
coffee.row["name"].asInstanceOf[String]
```

- ▶ Оказывается, можно обойтись без структурных типов
- ▶ Динамическая типизация + макросы = статическая типизация
- ▶ Этот подход можно улучшать и дальше

Пример №3: честные поставщики типов

```
@H2Db("jdbc:h2:coffees.h2.db") object Db  
println(Db.Coffees.all)  
Db.Coffees.insert("Brazilian", 99, 0)
```

- ▶ Эмуляция поставщиков типов на структурных типах это прикольно
- ▶ Но ее стоит воспринимать только как временное решение
- ▶ **Макро аннотации** должны закрыть этот вопрос раз и навсегда

Статические проверки

Статические проверки

- ▶ Поиск ошибок в программе во время компиляции
- ▶ Можно выдавать собственные ошибки и предупреждения
- ▶ Невозможно выполнять глобальные проверки

Пример №4: статически типизированные строки

```
scala> val x = "42"
```

```
x: String = 42
```

```
scala> "%d".format(x)
```

```
j.u.IllegalArgumentException: d != java.lang.String  
    at java.util.Formatter$FormatSpecifier.failConversion...
```

- Строки обычно считаются небезопасными

Пример №4: статически типизированные строки

```
scala> val x = "42"
```

```
x: String = 42
```

```
scala> "%d".format(x)
```

```
j.u.IllegalArgumentException: d != java.lang.String  
    at java.util.Formatter$FormatSpecifier.failConversion...
```

```
scala> f"$x%d"
```

```
<console>:31: error: type mismatch;  
found   : String  
required: Int
```

- ▶ Строки обычно считаются небезопасными
- ▶ Но, когда за дело берутся макросы, все меняется
- ▶ Особенно если использовать строковую интерполяцию

Пример №4: статически типизированные строки

```
implicit class Formatter(c: StringContext) {  
  def f(args: Any*): String = macro ...  
}
```

```
val x = "42"  
f"$x%d" // преобразуется в: StringContext("", "%d").f(x)
```



```
val arg$1: Int = x // ошибка компиляции  
"%d".format(arg$1)
```

- ▶ Макрос `f` превращает отформатированные строки в блоки кода
- ▶ Явные аннотации типов гарантируют отсутствие ошибок
- ▶ Похожие техники есть для регулярных выражений, двоичных литералов и т.д.

Пример №5: типизированные каналы Акки

```
trait Request
case class Command(msg: String) extends Request

trait Reply
case object CommandSuccess extends Reply
case class CommandFailure(msg: String) extends Reply

val actor = someActor
actor ! Command
```

- ▶ Актеры в Акке динамически типизированы, т.е. ! принимает Any
- ▶ Это делает невозможным статическую проверку сообщений

Пример №5: типизированные каналы Акки

```
trait Request
case class Command(msg: String) extends Request

trait Reply
case object CommandSuccess extends Reply
case class CommandFailure(msg: String) extends Reply

type Spec = (Request, Reply) :+ TNil
val actor = new ChannelRef[Spec](someActor)
actor <-!- Command // ошибка компиляции
```

- ▶ В принципе, сигнатуры актеров можно описать и без макросов
- ▶ Но практичным это становится только на макросах
- ▶ Типизированные каналы Акки разработаны специально для этого

Пример №5: типизированные каналы Акки

```
type Spec = (Request, Reply) :+ TNil  
val actor = new ChannelRef[Spec](someActor)  
actor <-!- Command // ошибка компиляции
```

- ▶ Макрос <-!- берет тип префикса и извлекает спецификацию канала
- ▶ Потом он вычисляет тип аргумента и проверяет его на соответствие
- ▶ Все это можно реализовать при помощи вычислений на типах
- ▶ Но это будет очень сложно и для программиста, и для пользователей

Пример №6: споры

```
def future[T](body: => T) = ...
```

```
def receive = {  
  case Request(data) =>  
    future {  
      val result = transform(data)  
      sender ! Response(result)  
    }  
}
```

- ▶ Пример выше иллюстрирует распространенную ошибку
- ▶ Захват переменной `sender` в замыкание – это плохая идея
- ▶ `sender` это не значение, как может показаться, а изменяемый метод
- ▶ Такого рода ошибки можно ловить макросами: [SIP-21 – Spores](#)

Пример №6: споры

```
def future[T](body: Spore[T]) = ...
```

```
def spore[T](body: => T): Spore[T] = macro ...
```

```
def receive = {  
  case Request(data) =>  
    future(spore {  
      val result = transform(data)  
      sender ! Response(result) // ошибка компиляции  
    })  
}
```

- ▶ Макрос `spore` вычисляет свободные переменные в своем аргументе
- ▶ Если какая-либо из этих переменных изменяемая, выдается ошибка

Пример №6: споры

```
def future[T](body: Spore[T]) = ...
```

```
implicit def anyToSpore[T](body: => T): Spore[T] = macro ...
```

```
def receive = {  
  case Request(data) =>  
    future {  
      val result = transform(data)  
      sender ! Response(result) // ошибка компиляции  
    }  
}
```

- ▶ Преобразование в споры можно сделать неявным
- ▶ В этом случае проверка замыканий будет совершенно незаметной

Предметные языки

Мечта разработчиков предметных языков

- ▶ Пользователь пишет обычную программу
- ▶ Компилятор прозрачно превращает ее в структуру данных

Мечта разработчиков предметных языков

- ▶ Даже из коробки Скала очень хороша
- ▶ **LMS** (также известный как Scala Virtualized) делает Скалу еще лучше
- ▶ Однако LMS тяжеловесен и не работает с официальной Скалой
- ▶ Макросы предоставляют легковесную альтернативу

Пример №7: LINQ

```
val usersMatching = query[String, (Int, String)](  
    "select id, name from users where name = ?")  
usersMatching("John")
```

- ▶ Запросы к базам данных можно писать на SQL

Пример №7: LINQ

```
val usersMatching = query[String, (Int, String)](  
    "select id, name from users where name = ?")  
usersMatching("John")
```

```
case class User(id: Column[Int], name: Column[String])  
users.filter(_.name === "John")
```

- ▶ Запросы к базам данных можно писать на SQL
- ▶ Их также можно закодировать в DSL, правда с некоторым трудом

Пример №7: LINQ

```
val usersMatching = query[String, (Int, String)](  
    "select id, name from users where name = ?")  
usersMatching("John")
```

```
case class User(id: Column[Int], name: Column[String])  
users.filter(_.name === "John")
```

```
case class User(id: Int, name: String)  
users.filter(_.name == "John")
```

- ▶ Запросы к базам данных можно писать на SQL
- ▶ Их также можно закодировать в DSL, правда с некоторым трудом
- ▶ Или же, благодаря макросам, можно все писать на обычной Скале

Пример №7: LINQ

```
trait Query[T] {  
  def filter(p: T => Boolean): Query[T] = macro ...  
}
```

```
val users: Query[User] = ...  
users.filter(_.name == "John")
```



```
Query(Filter(users, Equals(Ref("name"), Literal("John"))))
```

- ▶ Макрос `filter` превращает вызовы методов в структуры данных
- ▶ Дальше эти структуры можно транслировать в SQL
- ▶ Можно делать и по-другому: [An Embedded Query Language in Scala](#)

Пример №8: асинхронные вычисления

```
val futureDOY: Future[Response] =  
    WS.url("http://api.day-of-year/today").get
```

```
val futureDaysLeft: Future[Response] =  
    WS.url("http://api.days-left/today").get
```

```
futureDOY.flatMap { doyResponse =>  
    val dayOfYear = doyResponse.body  
    futureDaysLeft.map { daysLeftResponse =>  
        val daysLeft = daysLeftResponse.body  
        Ok(s"$dayOfYear: $daysLeft days left!")  
    }  
}
```

- ▶ Сделать программу асинхронной совсем непросто
- ▶ По сути, приходится выворачивать поток управления наизнанку

Пример №8: асинхронные вычисления

```
def async[T](body: => T): Future[T] = macro ...  
def await[T](future: Future[T]): T = macro ...
```

```
async {  
  val dayOfYear = await(futureDOY).body  
  val daysLeft = await(futureDaysLeft).body  
  Ok(s"$dayOfYear: $daysLeft days left!")  
}
```

- ▶ Сделать программу асинхронной совсем непросто
- ▶ Но макросы могут сделать это автоматически: [SIP-22 – Async](#)
- ▶ Генераторы из C# и Python эмулируются похожим образом

Пример №9: Datomisca

```
scala> Query("""
```

```
|      [ :find ?e ?n  
|        :in $ ?char  
|        :where [ ?e :person/name ?n ]  
|                [ ?e person/character ?char ]  
|      ]  
|      """)
```

```
res0: TypedQueryAuto2[DatomicData, DatomicData, (DatomicData,  
DatomicData)] = [ :find ?e ?n :in $ ?char :where ... ]
```

- ▶ Благодаря макросам строковая интерполяция становится механизмом встраивания внешних языков
- ▶ В этом примере в Скалу были интегрированы [запросы к Datomic](#)
- ▶ Почти даром получают статические проверки синтаксиса и типов
- ▶ Механизм можно обобщить до [Modular Quasiquote Abstraction](#)

Заключение

Для чего полезны макросы?

- ▶ Генерация кода
- ▶ Статические проверки
- ▶ Предметные языки