

第 14 章 遅延評価val (lazy value)

遅延評価val (lazy value) は、値の初期化を最初にアクセスされるまで遅延させる方法です。これは実行中に必要とならないかもしれない、計算コストが高い値を扱う場合に有用です。最初の例として、従業員のデータベースを考えましょう。各従業員ごとにマネージャーとチームが決まっています。

```
case class Employee(id: Int,
  name: String,
  managerId: Int) {
  val manager: Employee = Db.get(managerId)
  val team: List[Employee] = Db.team(id)
}
```

上記の Employee クラスは、直ちにそのフィールドをすべて初期化し、従業員テーブル全体をメモリにロードします。これは明らかに最善ではなく、フィールドを lazy にすることで簡単に改善できます。このようにして、データベースアクセスを、本当に必要になるまで、また、初めて必要になるまで遅らせます。

```
case class Employee(id: Int,
  name: String,
  managerId: Int) {
  lazy val manager: Employee = Db.get(managerId)
  lazy val team: List[Employee] = Db.team(id)
}
```

実際に何が起きているか、いつレコードがフェッチされるか表示するモックアップのデータベースを使って、見てみましょう。

```
object Db {
  val table = Map(1 -> (1, "Haruki Murakami", 1),
    2 -> (2, "Milan Kundera", 1),
    3 -> (3, "Jeffrey Eugenides", 1),
    4 -> (4, "Mario Vargas Llosa", 1),
    5 -> (5, "Julian Barnes", 2))

  def team(id: Int) = {
    for (rec <- table.values.toList; if rec._3 == id)
      yield recToEmployee(rec)
  }

  def get(id: Int) = recToEmployee(table(id))

  private def recToEmployee(rec: (Int, String, Int)) = {
    println("[db] fetching " + rec._1)
    Employee(rec._1, rec._2, rec._3)
  }
}
```

一人の従業員を取り出すプログラムを実行すると、確かにデータベースは遅延評価Val を参照するときのみアクセスされることが、出力によって確認できます。

ほかの遅延評価val の使い方は、いくつかのモジュールからなるアプリケーションの初期化の順番を解決することです。遅延評価Val が導入される前は、同様のことを object 定義を使用することで実現していました。二つ目の例として、いくつかのモジュールからなるコンパイラを考えてみましょう。最初に、シンボルのためのクラスと2つの事前定義された関数を定義している、単純なシンボルテーブルを見てください。

```
class Symbols(val compiler: Compiler) {
  import compiler.types._

  val Add = new Symbol("+", FunType(List(IntType, IntType), IntType))
  val Sub = new Symbol("-", FunType(List(IntType, IntType), IntType))

  class Symbol(name: String, tpe: Type) {
    override def toString = name + ": " + tpe
  }
}
```

```
}
```

symbols モジュールは、Compiler インスタンスでパラメータ化されています。Compiler インスタンスは、types モジュールなどのほかのサービスへのアクセスを提供します。この例では、事前定義された関数が2つ（加算と減算）だけあり、それらの定義は types モジュールに依存しています。

```
class Types(val compiler: Compiler) {
  import compiler.symtab._

  abstract class Type
  case class FunType(args: List[Type], res: Type) extends Type
  case class NamedType(sym: Symbol) extends Type
  case object IntType extends Type
}
```

2つのコンポーネントをつなぐため、コンパイラオブジェクトを作成して、2つのコンポーネントへ引数として渡します。

```
class Compiler {
  val symtab = new Symbols(this)
  val types = new Types(this)
}
```

残念ながら、この実直的なアプローチは実行時に失敗します。symtab モジュールが types モジュールを必要としているからです。一般的に、モジュール間の依存は複雑になりがちで、正しい順番で初期化するのは難しく、循環があるために不可能なことさえあります。簡単な対処は、そのようなフィールドを lazy にして、正しい順番は compiler に任せてしまうことです。

```
class Compiler {
  lazy val symtab = new Symbols(this)
  lazy val types = new Types(this)
}
```

これで、2つのモジュールは最初のアクセスで初期化され、compiler は期待通りに動くでしょう。

構文

lazy 修飾子は、具体的な値定義でのみ指定できます。値定義におけるすべての型付け規則が遅延評価valにも適用されますが、ひとつだけ制限が取り払われています。それは、再帰的なローカル値が許されることです。

[前ページ](#) [14章](#) [目次](#) [次ページ](#)

- すべての従業員テーブルをメモリにロードします (? 訳微妙) => 「従業員テーブル全体を」の方がいいかもです。 -- pomu0325 (2009-12-29 19:32:07)
- (? 訳微妙, my は ?) => 原文が"のtypo と思われます。 -- pomu0325 (2009-12-29 19:32:39)

名前:	<input type="text"/>
コメント:	<input type="text"/>

投稿