

Scalaの文字列処理

Day 2 リテラル・補間子・特殊な文字

リテラル

プログラム中に値を直接記述する表現

Javaではプリミティブ型（boolean, int, float, etc.）や参照型のStringなどに存在

※Scalaにはプリミティブ型は存在しない

文字に関するリテラル

- Char

Javaが65,536 (16bit)でUnicodeの1文字を表わそうとしたけどサロゲートペア登場のせいで完全には目的を果たせていない悲しいプリミティブ型charを、参照型CharとしてScalaが引き継いだ

- String

Charの配列を扱うラッパークラス

文字コードの歴史

1800年代半ばモールス信号 1900年代初めタイプライタ標準化 1946年ENIAC

70年代 メーカーごとに文字コードを策定

70年代後半～ 言語ごとに文字コードをまとめる動きが活発化

80年代半ば ISOが16bit固定長で世界中の文字を一つの文字集合に収めるUnicodeを構想

90年代前半 Oak言語がUnicodeを採用しchar型を16bit固定長にする

(C/C++言語のchar型は8bit固定長)

90年代半ば 1994年Oak言語がJava言語に名称変更、翌年Java言語公開

2003年 Scala言語公開

文字に関するリテラルの例

リテラル	Java	Scala
空リテラル (Object)	Object n = null;	val n: Object = null
文字リテラル (Char)	char c = 'A';	val c: Char = 'A'
文字列リテラル (String)	String s = "AB¥¥C¥nあいう";	val s: String = "AB¥¥C¥nあいう"
生文字リテラル (String)		val s: String = """AB¥C あいう"""

Scala特有の生文字リテラル

文字列リテラル	生文字リテラル
<pre>val s: String = "AB¥¥C¥nあいう"</pre>	<pre>val s: String = """AB¥C あいう"""</pre>
<pre>val waveDash: String = "¥u301c"</pre>	<pre>val waveDash: String = """¥u301c"""</pre>

生文字リテラル

stripMarginメソッド

生文字リテラルの改行のインデントを揃える方法

BEFORE (普通の生文字リテラルの改行)	AFTER (stripMarginを使用した改行)
<pre>val s: String = ""AB¥C あいう イロハ""</pre>	<pre>val s: String = ""AB¥C あいう イロハ"".stripMargin</pre>
	<pre>val s: String = ""AB¥C %あいう %イロハ"".stripMargin('%')</pre>

Scala特有の補間子

補間子はs補間子、f補間子、raw補間子の3種類

文字列リテラルや生文字リテラルの直前に記述

補間子はStringContextに対する暗黙クラスを定義することで自分で作成可能

s補間子

```
val universe = "宇宙"
```

```
val result = s"生命、$universe、そして${s"万物に  
についての${"究極の"}疑問"}の答えは、${21 + 21}"
```

f補間子

s補間子の拡張

```
val result = f"サイボーグ${9}%03dVSデビルマン"
```

raw補間子

文字列リテラル	文字列リテラル+ raw補間子	生文字リテラル
val s: String = "AB¥C¥nあいう"	val s: String = raw"AB¥C¥nあいう"	val s: String = """AB¥C¥nあいう"""
val waveDash: String = "¥u301c"	val waveDash: String = raw"¥u301c"	val waveDash: String = """¥u301c"""

同じ？？？→NO

文字列リテラル+raw補間子と 生文字リテラルの違い

文字列リテラル+ raw補間子	生文字リテラル
raw"改 行" → ✕	""改 行"" → ○
raw"ダブルクォーテーション 「"」 " → ✕	""ダブルクォーテーション 「"」 "" → ○

補問子の作成 StringContext

```
implicit class JsonHelper(val sc: StringContext) extends AnyVal {  
  def json(args: Any*): JSONObject = {  
    val strings = sc.parts.iterator  
    val expressions = args.iterator  
    val buf = new StringBuffer(strings.next)  
    while (strings.hasNext) {  
      buf.append(expressions.next).append(strings.next)  
    }  
    parseJson(buf)  
  }  
}  
val x: JSONObject = json"{ name: $name, id: $id }"
```


補問子の作成 StringContext

```
implicit class JsonHelper(val sc: StringContext) extends AnyVal {  
  def json(args: Any*): JSONObject = {  
    val strings = sc.parts.iterator  
    val expressions = args.iterator  
    val buf = new StringBuffer(strings.next)  
    while (strings.hasNext) {  
      buf.append(expressions.next).append(strings.next)  
    }  
    parseJson(buf)  
  }  
}
```

```
val x: JSONObject = json"{ name: $name, id: $id }"
```



```
new StringContext("{ name:", ",id: ", "}")  
      .json(name, id)
```


補問子の作成 StringContext

```
implicit class JsonHelper(val sc: StringContext) extends AnyVal {  
  def json(args: Any*): JSONObject = {  
    val strings = sc.parts.iterator  
    val expressions = args.iterator  
    val buf = new StringBuffer(strings.next)  
    while (strings.hasNext) {  
      buf.append(expressions.next).append(strings.next)  
    }  
    parseJson(buf)  
  }  
}
```

```
val x: JSONObject = json"{ name: $name, id: $id }"
```



```
new StringContext("{ name:", ",id: ", "}")
```



```
new JsonHelper(new StringContext("{ name:", ",id: ", "}))
```


補問子の作成 StringContext

```
implicit class JsonHelper(val sc: StringContext) extends AnyVal {  
  def json(args: Any*): JSONObject = {  
    val strings = sc.parts.iterator  
    val expressions = args.iterator  
    val buf = new StringBuffer(strings.next)  
    while (strings.hasNext) {  
      buf.append(expressions.next).append(strings.next)  
    }  
    parseJson(buf)  
  }  
}
```

```
val x: JSONObject = json"{ name: $name, id: $id }"
```

`new StringContext("{ name:", ",id: ", "}")`.json(name, id)

`new JsonHelper(new StringContext("{ name:", ",id: ", "}"))`.json(name, id)

エスケープシーケンス

シーケンス	意味
¥b	バックスペース
¥f	改ページ
¥n	改行
¥r	復帰
¥t	水平タブ
¥¥	バックスラッシュ・半角円記号
¥'	シングルクォーテーション
¥"	ダブルクォーテーション
¥ooo	8進数 $o \in [0, 7], ooo \in [000, 377]$

Unicodeシーケンス

シーケンス	意味
¥uHHHH	HHHHは16進数のUnicodeのコードポイント $h \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

シーケンス例	文字
¥u3003	同上記号「ㄥ」
¥u3005	漢字連続記号「々」
¥u3006	しめ記号「ㄨ」
¥u3007	漢数字ゼロ「〇」
¥u301C	波ダッシュ「〜」
¥u30FB	中点「・」
¥u4EDD	同上記号「仝」
¥u599B	幽霊文字「𐄂」
¥u5F41	幽霊文字「𐄁」
¥uFF5E	全角チルダ「～」
¥uD842¥uDFB7	吉野家の「吉」

OS依存文字

文字	取得方法	Windows	Unix
改行	System.getProperty("line.separator") System.lineSeparator String.format("%n")	¥r¥n	¥n
ディレクトリ やファイルの パスの区切り PATHや CLASSPATH の区切り	System.getProperty("file.separator") File.separator	¥¥	/
	System.getProperty("path.separator") File.pathSeparator	;	: