参考：<https://www.ibm.com/developerworks/cn/java/j-lo-compose/>

DSL 设计基础

我们在用编程语言（比如：Java 语言）实现某项功能时，其实就是在指挥计算机去完成这项功能。但是，语言能够带给我们的并不仅仅局限于此。更重要的是，语言提供了一种用于组织我们的思维，表达计算过程的框架。这个框架的核心就在于，如何能够把简单的概念组合成更为复杂的概念，同时又保持其对于组合方法的封闭，也就是说，组合起来的复杂概念对于组合手段来说和简单的概念别无二致。引用“Structure and Interpretation of Computer Programs”（参见 [参考资源](https://www.ibm.com/developerworks/cn/java/j-lo-compose/#artrelatedtopics)）一书中的话来讲，任何一个强大的语言都是通过如下三种机制来达成这个目标的：

* **原子**：语言中最简单、最基本的实体；
* **组合手段**：把原子组合起来构成更复杂实体的方法；
* **抽象手段**：命名复杂实体的方法，命名后的复杂实体可以和原子一样通过组合手段组合成为更复杂的实体。

像 Java 这种通用的编程语言，由于其所关注的是解决问题的一般方法。因此，其所提供的这三种机制也都是通用层面的。在解决特定问题时，这种通用的手段和特定问题领域中的概念、规则之间是存在着语义鸿沟的，所以某些问题领域中非常清晰、明确的概念和规则，在实现时就可能会变得不那么清晰。作为程序员来说，用干净的代码实现出功能仅仅是初级的要求，更重要的是要提升通用语言的层次，构建出针对特定问题领域的语言（DSL），这个过程中很关键的一点就是寻找并定义出面向问题领域的 **原子概念、组合的方法以及抽象的手段**。这个 DSL 不一定非得像通用语言那样是完备的，其目标在于清晰、直观地表达出问题领域中的概念和规则，其结果就是把通用的编程语言变成解决特定问题的专用语言。

我们曾经在“[基于 Java 的界面布局 DSL 的设计与实现](http://www.ibm.com/developerworks/cn/java/j-lo-uidsl/)”（参见 [参考资源](https://www.ibm.com/developerworks/cn/java/j-lo-compose/#artrelatedtopics)）一文中，构建了一种用于界面布局的 DSL，其中呈现了上述思想。在本文中，我们将以解析器的构造为例，来讲述如何构建一种用于字符串解析的 DSL，这种 DSL 具有强大的扩展能力，读者可以根据自己的需要定义出自己的组合手段。此外，从本文中读者还可以领略到 *函数编程*的优雅之处。

解析器原子

什么是解析器？最简单的解析器是什么？大家通常都会认为解析器就是判断输入的字符串是否满足给定的语法规则，在需要时还可以提取出相应的语法单位实例。从概念上来讲，这种理解没什么问题。不过要想定义出用于解析的 DSL，那么就需要更为精确的定义，也就是说我们要定义出解析器这个概念的确切类型。在 Java 中，我们用 interface 来定义解析器类型，如下：

|  |  |
| --- | --- |
| 1  2  3  4 | interface Parser  {      public Result parse(String target);  } |

其中，target 为要解析的字符串，Result 是解析的结果，只要满足这个接口语义的对象，我们就称其为一个解析器实例。Result 的定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | class Result  {      private String recognized;      private String remaining;      private boolean succeeded;        private Result(String recognized, String remaining,          boolean succeeded) {          this.recognized = recognized;          this.remaining = remaining;          this.succeeded = succeeded;      }        public boolean is\_succeeded() {          return succeeded;      }        public String get\_recognized() {          return recognized;      }        public String get\_remaining() {          return remaining;      }        public static Result succeed(String recognized,          String remaining) {          return new Result(recognized, remaining, true);      }        public static Result fail() {          return new Result("", "", false);      }  } |

其中，recognized 字段表示这个解析器所认识的部分，remaining 表示经过这个解析器解析后所剩余的部分，succeeded 表示解析是否成功，Result 是一个值对象。有了解析器的精确定义，接下来我们就可以定义出最简单的解析器。显然，最简单的解析器就是什么也不解析的解析器，把目标字符串原样返回，我们称其为 Zero，定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Zero implements Parser  {      public Result parse(String target) {          return Result.succeed("", target);      }  } |

Zero 解析器一定会解析成功，不做任何语法单位识别并直接返回目标字符串。下面我们来定义另外一个很简单的解析器 Item，只要目标字符串不为空，Item 就会把目标字符串的第一个字符作为其识别结果，并返回成功，如果目标字符串为空，就返回失败，Item 的定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Item implements Parser  {      public Result parse(String target) {          if(target.length() > 0) {              return Result.succeed(target.substring(0,1),                  target.substring(1));          }          return Result.fail();      }  } |

Zero 和 Item 是我们解析器 DSL 中仅有的两个原子，在下一小节中，我们来定义解析器的组合方法。

解析器组合子

我们在上一小节中定义了 Item 解析器，它无条件的解析出目标字符串中的第一个字符，如果我们希望能够变成有条件的解析，就可以定义出一个 **SAT 组合子**，其接收一个条件谓词（predicate）和一个解析器，并生成一个复合解析器，该复合解析器能否解析成功取决于原始解析器的解析结果是否满足给定的条件谓词，条件谓词和 SAT 的定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | interface Predicate  {      public boolean satisfy(String value);  }    class SAT implements Parser  {      private Predicate pre;      private Parser    parser;        public SAT(Predicate predicate, Parser parser) {          this.pre = predicate;          this.parser = parser;      }        public Result parse(String target) {          Result r = parser.parse(target);          if(r.is\_succeeded() && pre.satisfy(r.get\_recognized())) {              return r;          }          return Result.fail();      }  } |

如果，我们想定义一个解析单个数字的解析器，那么就可以定义一个 IsDigit 条件谓词，并通过 SAT 把该 IsDigit 和 Item 组合起来，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class IsDigit implements Predicate  {      public boolean satisfy(String value) {          char c = value.charAt(0);          return c>='0' && c<='9';      }  } |

解析单位数字的解析器 digit 定义如下：

|  |  |
| --- | --- |
| 1 | Parser digit = new SAT(new IsDigit(), new Item()); |

我们可以采用同样的方法组合出单个字母、单个大写字母、单个小写字母等解析器来。

接下来，我们定义一个 **OR 组合子**，其接收两个解析器，并分别用这两个解析器去解析一个目标串，只要有一个解析成功，就认为解析成功，如果两个都失败，则认为失败，代码定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class OR implements Parser  {      private Parser p1;      private Parser p2;        public OR(Parser p1, Parser p2) {          this.p1 = p1;          this.p2 = p2;      }        public Result parse(String target) {          Result r = p1.parse(target);          return r.is\_succeeded() ? r : p2.parse(target);      }  } |

我们可以定义出一个新的解析器 digit\_or\_alpha，如果目标字符是数字或者字母那么该解析器就解析成功，否则就失败。代码如下：

判断是否是字母的条件谓词：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class IsAlpha implements Predicate  {      public boolean satisfy(String value) {          char c = value.charAt(0);          return (c>='a' && c<='z') || (c>='A' && c<='Z');      }  } |

用于解析单个字母的解析器：

|  |  |
| --- | --- |
| 1 | Parser alpha = new SAT(new IsAlpha(), new Item()); |

digit\_or\_alpha 解析器定义：

|  |  |
| --- | --- |
| 1 | Parser digit\_or\_alpha = new OR(digit, alpha); |

下面我们来定义一个 **顺序组合子 SEQ**，该组合子接收两个解析器，先把第一个解析器应用到目标字符串，如果成功，就把第二个解析器应用到第一个解析器识别后剩余的字符串上，如果这两个解析器都解析成功，那么由 SEQ 组合起来的这个复合解析器就解析成功，只要有一个失败，复合解析器就解析失败。当解析成功时，其识别结果由这两个解析器的识别结果连接而成。

为了能够连接两个 Result 中已经识别出来的解析结果，我们在 Result 类中新增一个静态方法：concat，其定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static Result concat(Result r1, Result r2) {      return new Result(          r1.get\_recognized().concat(r2.get\_recognized()),          r2.get\_remaining(), true);  } |

顺序组合子 SEQ 的定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class SEQ implements Parser  {      private Parser p1;      private Parser p2;        public SEQ(Parser p1, Parser p2) {          this.p1 = p1;          this.p2 = p2;      }      public Result parse(String target) {          Result r1 = p1.parse(target);          if(r1.is\_succeeded()) {              Result r2 = p2.parse(r1.get\_remaining());              if(r2.is\_succeeded()) {                  return Result.concat(r1,r2);             }           }           return Result.fail();       }   } |

现在，如果我们想定义一个解析器用以识别第一个是字母，接下来是一个数字的情况，就可以这样定义：

|  |  |
| --- | --- |
| 1 | Parser alpha\_before\_digit = new SEQ(alpha, digit); |

接下来我们定义本文中的最后一个组合子：**OneOrMany**。该组合子接收一个解析器和一个正整数值，其生成的复合解析器会用原始解析器连续地对目标串进行解析，每一次解析时的输入为上一次解析后剩余的字符串，解析的最大次数由输入的正整数值决定。如果第一次解析就失败，那么该复合解析器就解析失败，否则的话，会一直解析到最大次数或者遇到解析失败为止，并把所有成功的解析的识别结果连接起来作为复合解析器的识别结果，**OneOrMany** 组合子的定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class OneOrMany implements Parser  {      private int max;      private Parser parser;        public OneOrMany(int max, Parser parser) {          this.max = max;          this.parser = parser;      }        public Result parse(String target) {          Result r = parser.parse(target);          return r.is\_succeeded() ? parse2(r,1) : Result.fail();       }        private Result parse2(Result pre, int count) {          if(count >= max) return pre;          Result r = parser.parse(pre.get\_remaining());          return r.is\_succeeded() ?              parse2(Result.concat(pre,r),count+1) : pre;      }   } |

使用该组合子，我们可以容易地定义出用于识别由最少一个，最多 10 个字母组成的串的解析器，如下：

|  |  |
| --- | --- |
| 1 | Parser one\_to\_ten\_alpha = new OneOrMany(10,alpha); |

本文的组合子就定义到此，不过读者可以根据自己的需要，用同样的方法容易地定义出符合自己要求其他组合子来。

抽象的手段

如果在 DSL 的构造中，仅仅提供了一些原子和组合手段，并且组合的结果无法再次参与组合，那么这个 DSL 的扩展能力和适用性就会大大折扣。相反，如果我们还能提供出抽象的手段对组合结果进行命名，命名后的复合实体可以像原子一样参与组合，那么 DSL 的扩展能力就会非常的强大，适用性也会大大增加。因此，抽象的手段在 DSL 的构造过程中是至关重要的。

敏锐的读者可能已经发现，对于我们的解析 DSL 来说，其实在前面的小节中已经使用了抽象的手段。比如，我们在 alpha，digit，digit\_or\_alpha 以及 alpha\_before\_digit 等复合解析器的定义中已经使用了抽象的手段来对其进行命名，然后可以直接使用这个抽象的名字再次参与组合。由于我们的解析器是基于 Java 语言中的 interface 机制定义的，因此，Java 语言中已有的针对 interface 的抽象支持机制完全适用于我们的解析 DSL。因此，我们就无需定义自己的特定抽象手段，直接使用 Java 语言中的即可。

相信读者已经从上一小节中的例子中看到组合、抽象手段的强大威力。在下一小节中，我们将给出一个更为具体的例子：H.248 协议中 NAME 语法解析器的构造。

一个 H.248 实例

在本小节中，我们将基于前面定义的解析器原子和组合子，实现用于识别 H.248 协议中 NAME 语法的解析器的构造。

H.248 是一个通信协议，媒体网关控制器使用该协议来对媒体网关进行控制。H.248 协议是一个基于 ABNF（扩展 BNF）文法描述的基于文本的协议，协议中定义了 H.248 消息的组成部分和具体内容。关于 H.248 协议的具体细节，我们不在本文中讨论，有兴趣的读者可以从 [参考资源](https://www.ibm.com/developerworks/cn/java/j-lo-compose/#artrelatedtopics) 中获取更多内容。我们仅仅关注其中的 NAME 语法定义，如下：

|  |  |
| --- | --- |
| 1  2  3 | NAME = ALPHA \*63(ALPHA / DIGIT / "\_" )  ALPHA = %x41-5A / %x61-7A   ; A-Z, a-z  DIGIT = %x30-39             ; digits 0 through 9 |

我们首先来解释一下其中的一些规则，\*63 其实是 n\*m 修饰规则的一个实例，表示最少 n 个最多 m 个，当 n 等于 0 时，可以简略写成 \*m。因此，\*63 表示最少 0 个，最多 63 个。/ 表示或规则，表示两边的实体可选。（）表示其中的实体必须得有一个。- 表示范围。因此，DIGIT 表示单个数字，ALPHA 表示单个字母（大写或者小写），(ALPHA/ DIGIT/ “\_” )表示要么是个字母，要么是个数字，要么是个下划线。\*63(ALPHA/ DIGIT/ “\_” )表示，最少 0 个，最多 63 个字母或者数字或者下划线。两个实体顺序写在一起，表示一种顺序关系，ALPHA \*63(ALPHA/ DIGIT/ “\_” ) 表示，以字母开始，后面最少 0 个，最多 63 个 字母或者数字或者下划线。更多的规则可以参见 [参考资源](https://www.ibm.com/developerworks/cn/java/j-lo-compose/#artrelatedtopics)。

根据前面的内容可以很容易地直接表达出用于解析这个语法规则的解析器来。如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class H248Parsec  {      public static Parser alpha() {          return new SAT(new IsAlpha(), new Item());      }        public static Parser digit() {          return new SAT(new IsDigit(), new Item());      }        public static Parser underline() {          return new SAT(new IsUnderline(), new Item());      }        public static Parser digit\_or\_alpha\_or\_underline() {          return new OR(alpha(), new OR(digit(), underline()));      }        public static Parser zero\_or\_many(int max, Parser parser){          return new OR(new OneOrMany(max,parser), new Zero());      }        public static Parser name() {          return new SEQ(alpha(),              zero\_or\_many(64,              digit\_or\_alpha\_or\_underline()));      }  } |

可以看出，我们的代码和协议中的语法描述基本上完全一样，我们通过定义自己的面向解析的 DSL，把 Java 这种通用语言变成了用于 ABNF 语法解析的专门语言，符合 Ward Cunningham 关于美的代码的定义。最后，我们用该解析器来做一些关于 NAME 语法识别的实验，如下表所示：

| **输入字符串** | **成功标志** | **识别结果** | **剩余字符串** |
| --- | --- | --- | --- |
| "" | false | "" | "" |
| "\_U" | false | "" | "" |
| "2U" | false | "" | "" |
| "U" | true | "U" | "" |
| "U{" | true | "U" | "{" |
| "U2{" | True | "U2" | "{" |
| "U\_{" | true | "U\_" | "{" |
| "U123\_{" | True | "U123\_" | "{" |
| "USER001" | True | "USER001" | "" |
| "USER001{" | True | "USER001" | "{" |
| "a0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789" | True | "a0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123" | "456789" |