dropincc.java 用户指南

(for dropincc.java v0.2.x)

Index

用途

所面向的需求和场景

Dropincc.java 是一个语法解析器生成器(compiler compiler,后简称“cc 工具”)；

与其它已有的同类产品的(如 javacc, antlr 等)不同之处在于，dropincc.java 的设计目的是为了在 java 平台上实施 DSL 方案而作；这种设计能带来更快的上手速度，更短更敏捷的开发 —— 测试周期循环，以及项目代码量的急剧降低；

简而言之，如果你打算在 java 平台上实现一套 DSL，且又觉得传统的 cc 工具(如 antlr)用起来太过繁琐时，就可以考虑使用 dropincc.java 了。

特点

- 纯 java 代码实现，且无第三方库依赖；也就是说，使用 dropincc.java 时唯一需要做的事情，就是引入 dropincc.java 的 jar 包；其余的就只需要 jdk 的内置库就可以了(需要 jdk1.6 或以上，确保 tools.jar 在 classpath 中，这些条件在你使用 jdk1.6 时都是默认的)
- 纯 java 的用户 api，这是 dropincc.java 区别于其它 cc 工具最明显的地方：传统的 cc 工具，基本

上都要求用户学习一种新的语法，以便用来编写用户想要实现的新语言的语法规则；而 dropincc.java 为用户设计了一套 Fluent Interface 风格的纯 java api 来描述语法规则，这意味着用户完全不需要学习新的语法，只需要使用熟悉的纯 java 语言，就能完成对新语言语法的描述；你能感受到拿起 java 就能轻松上手的滋味

- 动态编译生成可执行的解析器实例并自动管理；这也是区别于传统 cc 工具很明显的地方：传统 cc 工具，一般来讲会将生成的解释器源文件交给用户，让用户将这种源文件放入工程目录中使用；不过，这种自动生成的源文件其实并非需要人力维护的，其可读性、可维护性很低，并且由于它是自动生成的，那么用户其实根本不需要关心它的内容是什么，放入工程源码目录中只是徒然增加了项目代码整体的复杂度以及代码总行数(通常这种自动生成的源文件长达数千行);dropincc.java 借助 jdk1.6 的动态编译技术，将解析器代码的生成、编译、缓存、初始化放在幕后，用户无需对生成的解析器源码做任何操作就能直接动态调用初始化好的解析器对象；这进一步简化了使用，并且有效地防止了你的项目中出现几千行几乎无人能读懂的解析器代码
- 识别 LL(*)语法，对语法的描述、解析能力与 antlr3 基本一致

使用

常见的需求和实现步骤

dropincc.java 最常见的用途当然就是实现自定义的（external）DSL 了；对于这类需求，我们能总结出一个常用的步骤；一般来讲，步骤是：

1. 根据业务需要，设计 DSL 的词法元素和语法元素
2. 使用 EBNF 标记，将词法、语法规则组织成为无二义性的标准形式
3. 根据业务需要，设计出每一种词法、语法元素出现时所要执行的动作，或者说其出现时所要代表的语义
4. 使用 dropincc.java 的 api，将上述设计好的 EBNF 逐行描述，这样就将上述设计转化为了可执行的代码
5. 测试、使用实现好的 DSL

下面将以一个简单例子阐述一个典型的 DSL 设计、实现过程；

“多项式计算器”作为 cc 工具们津津乐道的“HelloWorld!”式的例子，我们不妨也在此考虑，如何用 dropincc.java 实现这样一个多项式计算器;

- 首先第一步，我们需要设计它的词法、语法元素; 作为“多项式计算器”，其业务逻辑是：
    - 能够做数字的加减乘除四则运算
    - 乘除法优先级高于加减法，相同优先级的运算从左到右依次进行
    - 可使用圆括号将一部分式子括起来以提高其运算优先级
- 经过分析，我们应该能够总结出，该 DSL 需要：“+”, “-”, “*”, “/”, “(”, “)”, “数字”这 7 种词法元素
- 然后根据业务逻辑规则，我们应该能为这个 DSL 设计出 4 种语法元素：表达式(expr), 加数(addend), 因数(factor), 并且能总结出下面这样的语法结构：

```
calc ::= expr $
expr ::= addend ((加号|减号) addend)*
addend ::= factor ((乘号|除号) factor)*
factor ::= 左圆括号 expr 右圆括号
    | 数字
```

上述语法描述基本上采用了 EBNF 类似的形式化描述，并且借助了“*”(kleene star)标记来表示“0 个或多个”，这在正则表达式中很常见, 应该不难理解。

除了上面提到的 4 种语法元素之外, 我们还引入了“$”符号和一个新的语法元素“calc”来为整个表达式划定一个词法意义上的“结束”，这都是描述 DSL 语法时的一些通行的做法，表示“若字符串输入结束，则整个表达式的输入也宣告结束”，这里不再赘述。

这段语法描述的意思也很明显和直接：

expr 由 1 个或多个 addend 组成，多个 addend 之间用加号或减号相连;

每个 addend 由 1 个或多个 factor 组成，多个 factor 之间用乘号或除号相连;

每个 factor，要么它是一个数字，要么它是一个被括号括起来的 expr;

这样的一个递归的定义就把我们“多项式计算器”的业务逻辑全部囊括了进去。

另，关于 DSL 的语法结构设计技巧已经超出本指南范围，因此不再赘述，下面将对如何使用 dropincc.java 实现拥有上述规则的 DSL 做出解释。

dropincc.java 用法详解

下载和安装

在 java 项目中使用 dropincc.java 非常简单，只需要在 download 页面下载最新可用的 v0.2.x 版本的 dropincc.java jar 包(本指南针对 v0.2.x 版本描述)放入项目的 classpath 中即可。

若项目使用了 maven 进行依赖管理，还可以直接加上 maven 依赖进行使用：

```
<dependency>
  <groupId>com.github.pfmiles</groupId>
  <artifactId>dropincc.java</artifactId>
  <version>0.2.1</version>
</dependency>
```

另外，请确保项目正在使用 jdk1.6 或以上版本的 java 环境。

token（终结符）定义

这一步，也就是使用 dropincc.java 的方式，将前面用户所设计出的 DSL 的词法元素描述出来;

在 dropincc.java 中，token 是用 TokenDef 这个类型来表示的，他能够通过调用 Lang 对象的方法被构造出来：

```
Lang c = new Lang(“Calculator”);
```

TokenDef plus = c.newToken(“\\+”);

其中 Lang 对象是用作抽象当前正在构造的 DSL 语言本身，它提供一些必要的方法，如创建 token，创建 grule(后面会介绍)，动态编译等；

上面的 plus 对象是表示“加号”这个 token，其参数之所以用“\\+”而不是“+”是因为 dropincc.java 在定义 token 时是以 java 内置正则表达式来描述 token 的规则，而“+”在 java 的正则表达式中是一个特殊字符，因此需要转义；(java 正则表达式的特殊字符共有： <([{\^-=$!|]})?*+.>)

token 除了可以被像上面这样专门通过 Lang 的方法被构造外，还可以在语法描述过程中直接以 String 的形式被“即时”定义，如：

m.or("/")

其中 m 是前面定义的某语法元素，表示乘号，后面的 String：“/”就“即时”定义了一个 token，表示除号，这就省去了 c.newToken(“/')的步骤；上述语句整体表示语法片段: * | / , 即“乘号或除号”

决定某 token 是否应该被 newToken 定义还是只是“即时定义”取决于用户是否需要 TokenDef 提供的一系列方法，如 or, and 表示“与”、“或”等用来表达更多的语法结构，如果不需要，那么清爽的“即时定义”当然是被推荐的。

此外还需要注意的一点是，目前的 dropincc.java 的 v0.2.x 系列版本，生成的解析器的词法规则的匹配是采用“最前匹配”的规律；也就是说，如果同时有多个 token 规则能匹配当前的输入串，dropincc.java 会选择最先被定义的那个 token 规则

因此，有时为了明确 token 的先后顺序，故意会选择 newToken 的形式来定义 token；并且，需要意识到这样的 token 定义是有问题的：

TokenDef t1 = c.newToken(“\\w+”);

TokenDef t2 = c.newToken(“abc”);

上述 token 定义由于 t1 完全涵盖了 t2 且 t1 又被定义在 t2 之前，因此，t2 永远不可能被匹配到，如果希望 t2 也有机会被匹配到，那么 t1,t2 应该交换顺序

最后，这里给出我们的示例“多项式计算器”的 7 种词法元素(token)对应的 java 正则表达式定义：

加号：\\+

减号：\\-

乘号：*

除号：/

正圆括号：\\(

反圆括号：\\)

数字：\\d+(\\.\\d+)?

注：上述转义符由于写在 java 代码中，因此是双反斜杠；“数字”token 的正则表达式，是表示允许带小数点的数字形式

grule(语法元素，即“非终结符”)定义

语法元素在 dropincc.java 中被表示为 Grule(grammar rule 的意思)对象；就好比 TokenDef 用来抽象 token 一样，Grule 用来抽象语法元素。

由于我们已经写出了“多项式计算器”的 EBNF，以及在上一节中，已经写出了所有词法元素(token)的正则表示；因此，目前我们的计算器的 EBNF 看起来应该是这样：

```
calc ::= expr $
expr ::= addend ((“\\+”|”\\-”) addend)*
addend ::= factor ((“\\*”|”/”) factor)*
factor ::= “\\(” expr “\\)”
        | “\\d+(\\.\\d+)?”
```

那么，接下来所要做的事情，就是使用 dropincc.java 将上述 EBNF“逐行翻译”为 dropincc.java 所要求的 api 调用形式：

```
Lang c = new Lang("Calculator");

TokenDef a = c.newToken("\\+");
TokenDef m = c.newToken("\\*");
Grule expr = c.newGrule();
Grule addend = c.newGrule();
Grule factor = c.newGrule();



// 从这里开始
c.defineGrule(expr, CC.EOF);
expr.define(addend, CC.ks(a.or("\\-"), addend));
addend.define(factor, CC.ks(m.or("/"), factor));
factor.define("\\(", expr, "\\)")
        .alt("\\d+(\\.\\d+)?");
```

上面的 java 语句可以仔细对照之前我们写出的 EBNF，其实不难发现就是一个逐行的“翻译”；

这里面需要说明的一些细节是：

“::=”由 lang.defineGrule 或 Grule.define 表示；

结束符“$”被 CC.EOF 常量表示；

“与”、“或”关系由 ele.and 和 ele.or 表示；

右边产生式的多个选择关系(即：“|” 符号)由“alt”调用表示；

“零个或多个”的关系，由 CC.ks 调用表示(ks 即“kleene star”)；

要看懂这段代码，所要了解的概念也就上述这些；并且这些概念已经涵盖了使用 dropincc.java 所需要了解的绝大多数！让用户尽可能少地死记概念也是 dropincc.java 的设计目标之一；

此外，CC.java 工具类中还有其它一些有用的常量或方法，将在下面介绍。

工具类 CC.java

上一节中我们接触过 CC.java 工具类中的 CC.EOF 常量和 CC.ks 方法；出此以外，CC.java 工具类还有其它一些常量和方法，这里我们将 CC.java 中所有的常量和方法放在一起统一说明：

CC.NOTHING：常量值，用来表示“空”产生式，即如果遇到这样的 EBNF：

```
a ::= b$
b ::= c
    |
```

那么，语法元素“b”就可以被表达为：

```
b.define(c)
 .alt(CC.NOTHING);
```

CC.EOF：常量值，表示词法意义上的结束符，这在大多数 DSL 设计中都很常用，表示“输入结束”

CC.ks：方法，意思是 kleene star, 即“克林星闭包”，表示“0 个或多个”重复项

CC.kc：方法，意思是 kleene cross，表示“1 个或多个”重复项

CC.op：方法，意思是 optional，表示“0 个或 1 个”项

总之，CC.java 中的常量和方法，都是用户构造语法时所要用到的非常常用的东西，并且概念很少，熟悉起来应该不困难。

动作(action)定义

经历了上面的过程之后，我们已经把“多项式计算器”的词法、语法解析规则完成了；

不过，只是完成解析是不够的，因为解析的结果仅仅是一个个 token 和语法节点组成的解析树(parse tree)；而我们想要的，是算式的计算结果；这个“计算”的过程才是我们真正的业务逻辑；

所以，和其它 cc 类工具一样，dropincc.java 也提供了一种叫做“action”的机制来让用户在解析语法的过程中插入业务逻辑代码，以完成需要的计算。

具体的 action 插入方法也很简单：在每一个想要插入 action 的产生式后面，调用“`action`”方法并传入一个 action 闭包即可，例如：

```
factor.define("\\(", expr, "\\)").action(new Action<Object>() {
    public Double act(Object matched) {
        return (Double) ((Object[]) matched)[1];
    }
}).alt("\\d+(\\.\\d+)?").action(new Action<Object>() {
    public Double act(Object matched) {
        return Double.parseDouble((String) matched);
    }
});
```

上述代码片段为 factor(因数)的产生式加上了 action，由于 factor 的产生式有 2 个分支，因此对应地也有

2 个 action；

首先来看第二个分支，即对“\\d+(\\.\\d+)?”(数字)的解析结果做操作的 action：它直接将解析到的 String 类型的 token 用 Double.parseDouble 转换成了一个 double 数字返回；这正符合这个分支的业务意义：解析出一个“数字”；

再看第一个分支，"\\(", expr, "\\)"，这个分支的意义是“被括起来的表达式”；跟第二个分支只有一个语法元素“数字”不同，这个分支拥有 3 个语法元素：左括号，表达式，右括号，因此，它的 act 方法的 matched 参数其实是一个长度为 3 的 Object[]，并且由于我们需要关心的只是中间那个语法元素：“表达式”，所以，这个分支的业务逻辑就是将中间这个“表达式”的结果直接返回：

```
return (Double) ((Object[]) matched)[1];
```

可以看到，在编写 action 的过程中，需要掌握的规律就是：matched 参数的类型和长度情况总是和对应分支的语法元素情况有关的，下面列举几种典型的语法元素和对应的 matched 参数情形以供参考从而熟悉其规律：

a, b, c：matched 参数为一个 Object[],长度为 3，其中每一个对应位置的元素的值为 a, b, c 这三种语法元素对应的 action 的返回值

a：matched 参数为一个 Object，其值为语法元素 a 对应的 action 的返回值

a, CC.ks(b), CC.kc(c), CC.op(d)：matched 参数为一个 Object[]，长度为 4，其中：

第一个元素为一个 Object，是更“下层”的语法元素 a 的 action 的返回值；

第二个元素为一个 Object[]，其长度等于克林星闭包(CC.ks)方法所匹配到的“b”元素的个数，每一个 Object 元素值都是对应的“b”元素的 action 的返回值;若此处的“CC.ks”实际运行中没有匹配到任何“b”元素，则该 Object[]的长度为 0

第三个元素也是一个 Object[]，跟第二个元素唯一不同的是，第三个元素的这个 Object[]的长度至少为 1，因为它是经由“CC.kc”匹配而来的；这个 Object[]其中的元素是被匹配到的“c”元素的 action 的返回值

第四个元素是一个 Object，它是被匹配到的“d”元素所对应的 action 的返回值；不过它也有可能为 null，因为这里是“CC.op”在进行匹配

完整的“多项式计算器”的代码示例在：Calculator.java，可仔细对照体会 action 的用法

生成运行时实例并使用

词法、语法规则以及 action 都完成之后，再只需一步，这个“多项式计算器”就可以使用了：

```
calc = c.compile();
```

这一步顾名思义，是将我们的语法规则动态编译为了一个可执行对象“calc”，calc 对象所拥有的“eval”方法即可用来做我们的表达式计算：

```
System.out.println(exe.eval("1 +2+3+(4 +5*6*7*(64/8/2/(2/1 )/1)*8  +9  )+   10"));
```

不妨自己尝试运行得到上述程序的输出结果

ParamedAction

上面的“多项式计算器”能够成功计算各种数字运算，这些参与计算的数字都是以字面量的形式被写在表达式文本中，我们的解析器直接从文本中解析得到这些数字并加以计算；

不过，很多时候我们不仅仅满足于基于表达式本身解析出的信息所做的计算，而是希望在处理 DSL 本身的过程中，与上下文环境进行一些交互来完成任务；比如我们如果开发一种带“变量”的 DSL，那么“变量”的存储显然是需要我们在某个地方维护的，并且这个存储不会直接体现在 DSL 代码中，而是由我们的程序维护在内存中(一般来讲)；这样就需要我们的 DSL 在解析的过程中，与内存中的某个存储或对象进行交互，这种交互的代码显然也应该放在 action 中；

在上面的“多项式计算器”的例子体验过程中，如果稍微留心即可发现，“calc”对象除了有一个只接受一个 String 类型的参数的 eval 方法外，还有一个签名为“eval(String code, Object arg)”的方法；这个方法除了第一个表示“DSL 代码文本”的 String 类型参数外，还有额外一个 Object 类型的参数；这个参数即可用来传入任意外部上下文对象；

在调用 eval 的时候传入了上下文，如何在 action 中使用呢？答案就是“使用 ParamedAction”，ParamedAction 接口签名如下：

```
public interface ParamedAction<P, M> {
    public Object act(P arg, M matched);
}
```

ParamedAction 接口与之前展示过的 Action 接口唯一的区别在于 act 方法多出来一个“arg”参数，这个参数即是 eval 时传入的那个上下文对象；

关于 ParamedAction 的应用场景，可参考 BoolExpr.java，其中有一些简单的使用范例

需要用到 ParamedAction 的应用场景其实有很多，因为我们如果希望创造的 DSL 功能强大那就免不了要和外部上下文做交互，以完成复杂的功能

高级话题

TODO

一些有用的建议

- 上述“多项式计算器”例子讲述了如何使用 dropincc.java 制造一个计算器，其计算过程是伴随着计算器 DSL 的解析过程而进行的；但是很多时候我们并不希望在解析的过程中就做完所有的计算，而是希望先把 DSL 解析成 AST(抽象语法树，即 abstract syntax tree)，然后使用 visitor 模式做一些中间分析或优化，最后再遍历该 AST 执行最终的业务逻辑；其实使用 dropincc.java 做到这一点很简单，就算我没有在此专门指出用户也应该能够想到：直接在 action 中返回 AST 的节点对象就行了，这样你解析器执行的最终返回结果就是一棵 AST 了
- 值得注意的是，Lang.compile()方法是一个很“重”的方法，因此原则上讲它应该只被调用一次；之后可将其返回的 Exe 对象缓存起来并不断重复使用；只要 DSL 的语法规则不调整，就不必再次调用 Lang.compile()；上面提到的 Calculator.java 和 BoolExpr.java 关于如何处理好这个缓存给出了很好的例子，他们将 Lang.compile()的结果直接缓存到了一个 static 变量中，可以作为参考
- Exe 对象是无状态的，因此你可以放心地在多线程环境重复使用它；不过，解析器程序是否真的无状态还取决于用户写的 action 代码，由于 dropincc.java 的 action 实际上是一个闭包(在 java 中体现为一个匿名内部类)，所以用户完全可以在 action 中保持一些状态，dropincc.java 也没有理由禁止用户这么做，只要用户自己清楚地“知道自己在干什么”并且做好必要的线程安全工作就好