第13章 代码生成器实现

- 实现封装元数据的工具类
- 实现代码生成器的代码编写
- 掌握模板创建的

1 构造数据模型

1.1 需求分析

借助Freemarker机制可以方便的根据模板生成文件，同时也是组成代码生成器的核心部分。对于Freemarker而言，其强调 `数据模型 + 模板 = 文件` 的思想，所以代码生成器最重要的一个部分之一就是数据模型。在这里数据模型共有两种形式组成：

- 数据库中表、字段等信息

  针对这部分内容，可以使用元数据读取并封装到java实体类中

- 用户自定义的数据

  为了代码生成器匹配多样的使用环境，可以让用户自定义的数据，并且以key-value的形式配置到properties文件中

接下来我们一起针对这两方面的数据进行处理

1.2 自定义数据

通过PropertiesUtils工具类，统一对properties文件夹下的所有 `.properties` 文件进行加载，并存入内存中

```
/**
 * 需要将自定义的配置信息写入到properties文件中
 *      配置到相对于工程的properties文件夹下
 */
public class PropertiesUtils {

    public static Map<String,String> customMap = new HashMap<>();

    static {
        File dir = new File("properties");
        try {
            List<File> files = FileUtils.searchAllFile(new
File(dir.getAbsolutePath()));
            for (File file : files) {
                if(file.getName().endsWith(".properties")) {
                    Properties prop = new Properties();
                    prop.load(new FileInputStream(file));
                    customMap.putAll((Map) prop);
                }
            }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1.3 元数据处理

加载指定数据库表，将表信息转化为实体类对象（Table）

```
    /**
     * 获取表及字段信息
     */
    public static List<Table> getDbInfo(DataBase db,String tableNamePattern) throws
Exception {
        //创建连接
        Connection connection = getConnection(db.getDriver(),db.getUserName(),
db.getPassWord(), db.getUrl());
        //获取元数据
        DatabaseMetaData metaData = connection.getMetaData();

        //获取所有的数据库表信息
        ResultSet tablers = metaData.getTables(null, null, tableNamePattern, new
String[]{"TABLE"});

        List<Table> list=new ArrayList<Table>();

        //拼装table
        while(tablers.next()) {
            Table table = new Table();

            String tableName=tablers.getString("TABLE_NAME");

            //如果为垃圾表
            if(tableName.indexOf("=")>=0 || tableName.indexOf("$")>=0){
                continue;
            }
            table.setName(tableName);

            table.setComment(tablers.getString("REMARKS"));

            //获得主键
            ResultSet primaryKeys = metaData.getPrimaryKeys(null, null, tableName);

            List<String> keys=new ArrayList<String>();

            while(primaryKeys.next()){
                String keyname=primaryKeys.getString("COLUMN_NAME");
                //判断 表名为全大写 ，则转换为小写
                if(keyname.toUpperCase().equals(keyname)){
                    keyname=keyname.toLowerCase();//转换为小写
                }
```

```
                keys.add(keyname);
            }

            //获得所有列
            ResultSet columnrs = metaData.getColumns(null, null, tableName, null);

            List<Column> columnList=new ArrayList<Column>();

            while(columnrs.next()){

                Column column=new Column();
                //处理字段
                String columnName=  columnrs.getString("COLUMN_NAME");

                //字段名称
                column.setColumnName(columnName);
                column.setColumnName2(StringUtils.toJavaVariableName(columnName));

                //字段类型
                String columnDbType = columnrs.getString("TYPE_NAME");
                column.setColumnDbType(columnDbType);//数据库原始类型

                //java类型
                Map<String, String> convertMap = PropertiesUtils.customMap;
                String typeName = convertMap.get(columnDbType);//获取转换后的类型

                if(typeName==null) {
                    typeName=columnrs.getString("TYPE_NAME");
                }

                column.setColumnType(typeName);

                String remarks = columnrs.getString("REMARKS");//备注

                column.setColumnComment(StringUtils.isBlank(remarks)?
columnName:remarks);

                //如果该列是主键
                if(keys.contains(columnName)){
                    column.setColumnKey("PRI");
                    table.setKey(column.getColumnName());
                }else {
                    column.setColumnKey("");
                }

                columnList.add(column);
            }
            columnrs.close();
            table.setColumns(columnList);
            list.add(table );
        }
        tablers.close();
        connection.close();
```

```
        return list;
    }
```

2 实现代码生成

2.1 需求分析

为了代码更加直观和易于调用，实现代码生成共有两个类组成：

- UI界面统一调用的入口类：GeneratorFacade

  方便多种界面调用，主要完成数据模型获取，调用核心代码处理类完成代码生成

- 代码生成核心处理类：Generator

  根据数据模型和模板文件路径，统一生成文件到指定的输出路径

2.2 模板生成

（1）配置统一调用入口类GeneratorFacade

```
/**
 * 1.根据传入数据库信息构造数据
 * 2.根据模板完成代码生成
 */
public class GeneratorFacade {

    private Generator generator;

    //公共数据Map集合（处理文件路径等公共代码替换）
    private Map<String,Object> commonMap;

    public GeneratorFacade(String templatePath, String outPath, Settings settings) {
        commonMap = settings.getSettingMap();
        commonMap.putAll(PropertiesUtils.customMap);
        try {
            generator = new Generator(templatePath,outPath);
        }catch (Exception e){

        }
    }

    //针对数据库表生成
    public void generatorByTable(DataBase db,String tableName) throws Exception {
        //查询数据库获取所有表信息
        List<Table> tableList = DataBaseUtils.getDbInfo(db, tableName);
        for (Table table : tableList) {
            //根据数据库表信息，构造数据模型并生成代码
            generator.scanTemplatesAndProcess(getTemplateModel(table));
        }
    }
```

```
    //根据数据库对象table构造数据模型
    private Map getTemplateModel(Table table) {
        Map<String,Object> templateMap = new HashMap();
        //table表信息
        templateMap.put("table",table);
        //实体类名称
        String prefixs = (String) commonMap.get("tableRemovePrefixes");

        String className = table.getName();

        for(String prefix : prefixs.split(",")) {
            className = StringUtils.removePrefix(className, prefix,true);
        }
        templateMap.put("ClassName",
StringUtils.makeAllWordFirstLetterUpperCase(className));

        //公共的配置和自定义配置
        templateMap.putAll(commonMap);
        return templateMap;
    }
}
```

（2）处理模板代码生成的核心类Generator

```
public class Generator {

    //模板所在路径
    private String templatePath;
    //代码生成路径
    private String outPath;

    private Configuration conf;

    public Generator(String templatePath, String outPath) throws Exception {
        this.templatePath = templatePath;
        this.outPath = outPath;
        //创建freemarker的核心配置类
        conf = new Configuration();
        //指定模板加载器
        conf.setTemplateLoader(new FileTemplateLoader(new File(templatePath)));
    }

    //扫描所有模板并进行代码生成
    public void scanTemplatesAndProcess(Map dataMap) throws Exception {
        //加载文件夹下的所有模板文件
        List<File> srcFiles = FileUtils.searchAllFile(new File(templatePath));
        //针对每一个模板文件进行代码生成
        for(File srcFile :srcFiles) {
            executeGenerate(dataMap, srcFile);
        }
    }

    //对某个模板生成代码
```

```
    private void executeGenerate(Map dataMap ,File srcFile) throws Exception {
        //获取文件路径
        String templateFile = srcFile.getAbsolutePath().replace(this.templatePath,"");
        //对文件名称进行处理（字符串替换）
        String outputFilepath = processTemplateString(templateFile,dataMap);
        //读取模板
        Template template = conf.getTemplate(templateFile);
        //设置字符集
        template.setOutputEncoding("encode");
        //创建文件
        File outFile = FileUtils.mkdir(outPath,outputFilepath);
        FileWriter fileWriter = new FileWriter(outFile);
        //模板生成
        template.process(dataMap,fileWriter);
        fileWriter.close();
    }
}
```

2.3 路径处理

使用字符串模板对文件生成路径进行统一处理

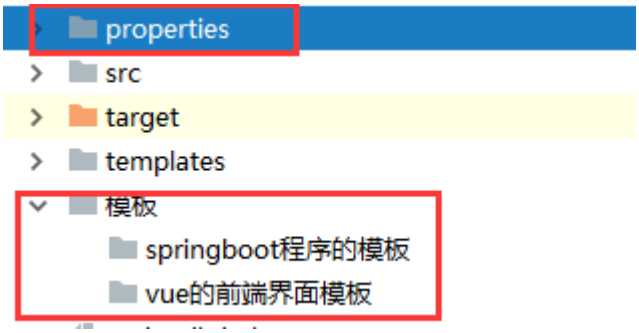
```
    //处理字符串模板
    private String processTemplateString(String templateString,Map dataMap) throws
Exception {
        StringWriter out = new StringWriter();
        Template template = new Template("ts",new StringReader(templateString),conf);
        template.process(dataMap, out);
        return out.toString();
    }
```

3 制作模板

3.1 模板制作的约定

（1）模板位置

模板统一放置到相对于当前路径的`模板`文件夹下



（2）自定义数据

自定义的数据以`.propeties`文件(key-value)的形式存放入相对于当前路径的`properties`文件夹下

（3）数据格式

名称	说明
author	作者
project	工程名
path1	包名1
path2	包名2
path3	包名3
pPackage	完整包名
projectComment	工程描述
ClassName	类名
table	数据库信息

table中数据内容：

name	表名
comment	表注释
key	表主键
columns	所有列信息
columnName	**字段列名**
columnName2	**属性名**
columnType	**java类型**
columnDbType	**数据库类型**
columnComment	**注释**
columnKey	**是否主键**

3.2 需求分析

制作通用的SpringBoot程序的通用模板

- 实体类

  类路径，类名，属性列表（getter，setter方法）

- 持久化层

  类路径，类名，引用实体类

- 业务逻辑层

  类路径，类名，引用实体类，引用持久化层代码

- 视图层

  类路径，类名，引用实体类，引用业务逻辑层代码，请求路径

- 配置文件

  pom文件，springboot配置文件

3.3 SpringBoot通用模板

3.3.1 实体类

```
package ${pPackage}.pojo;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * ${comment!}服务层
 * @author  ${author!"itcast"}
 */
@Entity
@Table(name="${table.name}")
public class ${ClassName} implements Serializable {
    //定义私有属性
    <#list table.columns as column>
    <#if column.columnKey??>
    @Id
    </#if>
    private ${column.columnType} ${column.columnName2};
    </#list>

    //处理getter，setter方法
    <#list table.columns as column>
    public void set${column.columnName2?cap_first}(${column.columnType} value) {
        this.${column.columnName2} = value;
    }

    public ${column.columnType} get${column.columnName2?cap_first}() {
        return this.${column.columnName2};
    }
    </#list>
}
```

3.3.2 持久化层

```
package ${pPackage}.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;

import ${pPackage}.pojo.${ClassName};

/**
 * ${comment!}数据访问接口
 * @author ${author!"itcast"}
 */
public interface ${ClassName}Dao extends
JpaRepository<${ClassName},String>,JpaSpecificationExecutor<${ClassName}>{

}
```

3.3.3 Service层

```
<#assign classNameLower = ClassName ? uncap_first>
package ${pPackage}.service;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Map;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Expression;
import javax.persistence.criteria.Predicate;
import javax.persistence.criteria.Root;
import javax.persistence.criteria.Selection;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.domain.Specification;
import org.springframework.stereotype.Service;
import util.IdWorker;
import ${pPackage}.dao.${ClassName}Dao;
import ${pPackage}.pojo.${ClassName};

/**
 * ${comment!}服务层
 * @author  ${author!"itcast"}
 */
@Service
public class ${ClassName}Service {

    @Autowired
    private ${ClassName}Dao ${classNameLower}Dao;

    @Autowired
```

```
    private IdWorker idWorker;

    /**
     * 查询全部列表
     * @return
     */
    public List<${ClassName}> findAll() {
        return ${classNameLower}Dao.findAll();
    }

    /**
     * 分页查询
     *
     * @param page
     * @param size
     * @return
     */
    public Page<${ClassName}> findPage(int page, int size) {
        PageRequest pageRequest = PageRequest.of(page-1, size);
        return ${classNameLower}Dao.findAll(pageRequest);
    }

    /**
     * 根据ID查询实体
     * @param id
     * @return
     */
    public ${ClassName} findById(String id) {
        return ${classNameLower}Dao.findById(id).get();
    }

    /**
     * 增加
     * @param ${ClassName}
     */
    public void add(${ClassName} ${ClassName}) {
        ${ClassName}.setId( idWorker.nextId()+"" );
        ${classNameLower}Dao.save(${ClassName});
    }

    /**
     * 修改
     * @param ${ClassName}
     */
    public void update(${ClassName} ${ClassName}) {
        ${classNameLower}Dao.save(${ClassName});
    }

    /**
     * 删除
     * @param id
     */
    public void deleteById(String id) {
```

```
        ${classNameLower}Dao.deleteById(id);
    }
}
```

3.3.4 Controller

```
<#assign classNameLower = ClassName ? uncap_first>
package ${pPackage}.controller;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import ${pPackage}.pojo.${ClassName};
import ${pPackage}.service.${ClassName}Service;

import entity.PageResult;
import entity.Result;
/**
 * [comment]控制器层
 * @author Administrator
 *
 */
@RestController
@CrossOrigin
@RequestMapping("/${classNameLower}")
public class ${ClassName}Controller {

    @Autowired
    private ${ClassName}Service ${classNameLower}Service;


    /**
     * 查询全部数据
     * @return
     */
    @RequestMapping(method= RequestMethod.GET)
    public Result findAll(){
        return new Result(ResultCode.SUCCESS,${classNameLower}Service.findAll());
    }

    /**
     * 根据ID查询
     * @param id ID
```

```
     * @return
     */
    @RequestMapping(value="/{id}",method= RequestMethod.GET)
    public Result findById(@PathVariable String id){
        return new Result(ResultCode.SUCCESS,${classNameLower}Service.findById(id));
    }


    /**
     * 分页查询全部数据
     * @param page
     * @param size
     * @return
     */
    @RequestMapping(value="/{page}/{size}",method=RequestMethod.GET)
    public Result findPage(@PathVariable int page,@PathVariable int size){
        Page<${ClassName}> searchPage = ${classNameLower}Service.findPage(page, size);
        PageResult<Role> pr = new
PageResult(searchPage.getTotalElements(),searchPage.getContent());
        return new Result(ResultCode.SUCCESS,pr);
    }


    /**
     * 增加
     * @param ${classNameLower}
     */
    @RequestMapping(method=RequestMethod.POST)
    public Result add(@RequestBody ${ClassName} ${classNameLower}  ){
        ${classNameLower}Service.add(${classNameLower});
        return new Result(ResultCode.SUCCESS);
    }

    /**
     * 修改
     * @param ${classNameLower}
     */
    @RequestMapping(value="/{id}",method= RequestMethod.PUT)
    public Result update(@RequestBody ${ClassName} ${classNameLower}, @PathVariable
String id ){
        ${classNameLower}.setId(id);
        ${classNameLower}Service.update(${classNameLower});
        return new Result(ResultCode.SUCCESS);
    }

    /**
     * 删除
     * @param id
     */
    @RequestMapping(value="/{id}",method= RequestMethod.DELETE)
    public Result delete(@PathVariable String id ){
        ${classNameLower}Service.deleteById(id);
        return new Result(ResultCode.SUCCESS);
```

```
    }
}
```

3.3.5 配置文件

（1）application.yml

```
server:
  port: 9001
spring:
  application:
    name: ${project}-${path3} #指定服务名
  datasource:
    driverClassName: ${driverName}
    url: ${url}
    username: ${dbuser}
    password: ${dbpassword}
  jpa:
    database: MySQL
    show-sql: true
```

（2）pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>${path_1}.${path2}</groupId>
    <artifactId>${project}_parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>${project}_${path3}</artifactId>
  <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
            <groupId>${path1}.${path2}</groupId>
            <artifactId>${project}_common</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
  </dependencies>
</project>
```