Java后台开发常用框架及组件

内部培训资料

本文将介绍一些在Java后台开发中常用的框架及组件，包含项目管理工具、开发集成框架、ORM框架、缓存访问组件等，使用它们能避免重复性劳动、减少编码量，从而提高开发效率。此外，这些都是第三方开源软件，有兴趣的话还可以对其源代码进行研究。

1. 项目管理工具 - Maven

Maven一个用于项目依赖管理与构建管理的工具，它能根据配置自动解析并引入相关的依赖jar，支持依赖关系的多层级解析，而且利用的其构建功能，可以很容易完成项目的单元测试、打包、部署等。

1.1 安装

手工下载安装并比较繁琐，目前主流IDE如Eclipse、IDEA都内置了Maven，无须关注安装，只须简单配置即可使用。

配置Maven 只须新建文件~/.m2/settings.xml，下面介绍相关术语：

- Maven配置文件 Maven配置文件名为settings.xml，负责Maven全局配置，主要内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <localRepository>~/.m2/repository</localRepository>

  <servers>

    <server>
      <id>mfw-public</id>
      <username>username</username>
      <password>password</password>
    </server>
    <server>
      <id>mfw-releases</id>
      <username>username</username>
      <password>password</password>
    </server>
    <server>
      <id>mfw-snapshots</id>
      <username>username</username>
```

```
    <password>password</password>
  </server>

</servers>

<mirrors>

  <mirror>
    <id>company-public-central</id>
    <name>Mirror of central</name>
    <url>https://nexus.mfwdev.com/repository/maven-public/</url>
    <mirrorOf>central</mirrorOf>
  </mirror>

</mirrors>

<profiles>

  <profile>
    <id>jdk-1.6</id>
    <activation>
      <activeByDefault>true</activeByDefault>
      <jdk>1.6</jdk>
    </activation>
    <properties>
      <maven.compiler.source>1.6</maven.compiler.source>
      <maven.compiler.target>1.6</maven.compiler.target>
      <maven.compiler.compilerVersion>1.6</maven.compiler.compilerVersion>
    </properties>
  </profile>

  <profile>
    <id>mfw-profile</id>
    <repositories>
      <repository>
        <id>mfw-snapshots</id>
        <url>https://nexus.mfwdev.com/repository/xxx-snapshots/</url>
        <releases>
          <enabled>false</enabled>
          <updatePolicy>interval:300</updatePolicy>
          <checksumPolicy>warn</checksumPolicy>
        </releases>
        <snapshots>
          <enabled>true</enabled>
          <updatePolicy>interval:300</updatePolicy>
          <checksumPolicy>warn</checksumPolicy>
        </snapshots>
      </repository>
```

```
        <repository>
          <id>mfw-releases</id>
          <url>https://nexus.mfwdev.com/repository/xxx-releases/</url>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>interval:300</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>false</enabled>
            <updatePolicy>interval:300</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </snapshots>
        </repository>

        <repository>
          <id>mfw-public</id>
          <name>Repository for MFW maven</name>
          <url>https://nexus.mfwdev.com/repository/maven-public/</url>
          <layout>default</layout>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>interval:1440</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>interval:1440</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </snapshots>
        </repository>
      </repositories>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>mfw-profile</activeProfile>
  </activeProfiles>

</settings>
```

说明：

1.~/.m2/repository目录表示Maven本地资源库位置。 2.实际使用中将xxx换成相应的业务部门对应的资源库前缀，例如content，如没有，向Maven仓库管理部门申请获得。 3.username和password分别为访问Maven远程资源仓库的用户名和密码，可向Maven仓库管理部门申请获得。

- Maven远程资源库

一个远程集中式的软件仓库，所有使用Maven进行管理的项目对外发布后，都存放于此。它分为三种，官方仓库（包括各种官网镜像）、私有仓库。

- Maven本地资源库

当使用maven进行项目开发时，Maven会把用到的远程资源库中的软件包自动下载到本地一个文件夹，例如~/.m2/repository，以便下次直接使用，这个目录就叫做本地资源库。

- 项目坐标

定义一个项目在Maven资源库的唯一标识，由groupId（组名）、artifactId（构件名）、version（版本）三者唯一标识。

- POM文件

Maven使用一个固定名称的XML配置文件pom.xml来实现配置的集中管理，这个文件叫做 POM（Project Object Model）文件，以下就是一个简单的POM文件。

```
<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>
    <groupId>com.mfw.demo</groupId>
    <artifactId>first_maven_project</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <dependencies>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.9</version>
        </dependency>
    </dependencies>
</project>
```

1.2 工程结构

一个maven项目的典型结构为：

```
|____src                    //源代码根目录
|   |____main               //主体功能代码及资源根目录
|   |   |____resources      //资源文件根目录
|   |   |____java           //java代码根目录
|   |____test               //测试代码及资源根目录
|   |   |____resources      //测试资源文件根目录
|   |   |____java           //测试java代码根目录
|____target                 //编译后的类及资源文件根目录
|   |____test-classes       //编译后的测试类及资源文件根目录
|   |____classes            //编译后的主体功能类及资源文件根目录
|____pom.xml                //工程管理文件
```

1.3 核心功能

1.3.1 依赖管理

Java后台项目开发中不可避免地会需要引入其它软件包（jar包），如果没有自动化管理工具，需要手动获取到目标jar文件，并把它拷贝到指定的引用路径中，才能被正常引用，这当然是重复性地繁琐工作。使用Maven后，只需要在在工程根目录添加POM文件，并在其中使用标签声名所要引入jar包坐标（、、），就能唯一定位这个包，例如

```
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
        <version>3.9</version>
    </dependency>
```

如果这个在本地不存在，将由maven自动下载到本地。至于、、是每个maven项目都必须具有的唯一标识，在上面的pom文件中，也定义了自身的坐标。

1.3.2 构建管理

maven在项目构建管理方面做得很完善，从maven的视角来看，一个项目的生命周期（lifecycle）可以分别编译、测试、打包、安装、部署五个阶段，这些阶段分别使用compile、test、package、install、deploy命令来执行，它们之间是顺序依赖的关系，即后面的命令执行时会自动执行它之前的所有命令，只有前面的命令执行成功才会接着后续执行。

Maven项目生命周期

图1 Maven项目生命周期

compile：编译工程主体功能src/main/目录下代码并将结果输出到target目录。

test：执行src/test/java目录中的测试用例，依赖于主体功能代码编译通过以后才能进行。此步可使用构建参数跳过。

package：将编译后结果按格式（默认为jar）打包至target目录。

install：将package步骤中生成的jar包安装到本地资源库中。

deploy：将package步骤中生成的jar包部署到maven远程资源库。

以上生命周期名称即相应maven执行命令，如compile周期对应的命令即为compile。

此外，利用maven提供的丰富插件（包括很多第三方插件），能够使用简单的命令就可以完成很多复杂的任务，例如包组装、代码检测、文档生成等这里就不一一赘述。

2. 开发集成框架 - Spring

Spring可以说是Java生态第一开发框架了，它经历了近20年的发展，已经形成了涵盖系统集成、控制反转（IoC，也叫依赖注入）容器、AOP管理、Web API、Web Pages、批处理作业、应用启动管理、微服务管理等领域的技术生态，它为企业级应用开发提供了完整解决方案。限于篇幅，本文只介绍Spring IoC容器、Spring Boot和Spring MVC框架三部分。

2.1 Spring IoC 容器

IoC容器是一种实现了对象自动创建并建立相互依赖关系容器，因此它又可以称作依赖注入。IoC容器负责根据配置或者注解定位、解析、实例化应用程序中的用到的对象并建立这些对象间的依赖关系，这样应用程序就无需直接在代码中创建相关的对象，只需直接使用即可。

以下是一个采用XML配置进行依赖关系管理的例子：

添加Maven依赖：

```
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.1.3.RELEASE</version>
    </dependency>
```

编写代码：

```java
package demo.ioc;

public interface IHello {
    void sayHello();
}
```

```java
package demo.ioc;

public class TextHelloImpl implements IHello{

    @Override
    public void sayHello() {
        System.out.println("say hello by text.");
    }

}
```

```java
package demo.ioc;

public class HelloClient {

    private String version;
    private IHello helloService;

    public void showVersion() {
        System.out.println("Version is " + getVersion());
    }

    public void textHello() {
        helloService.sayHello();
    }

   /**setter和getter方法是必不可少的，Spring IoC使用它们对属性字段进行访问*/
    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }

    public IHello getHelloService() {
        return helloService;
    }

    public void setHelloService(IHello helloService) {
        this.helloService = helloService;
    }
```

```
}
```

在src/main/java/demo/ioc目录下创建一个名为applicationContext.xml的文件（文件名可随意），内容如下：

```
<?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/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.1.xsd
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

  <bean id="textHelloService" class="demo.ioc.TextHelloImpl"/>

  <bean id="helloClient" class="demo.ioc.HelloClient">
    <property name="version" value="1.4" />
    <property name="helloService" ref="textHelloService" />
  </bean>

</beans>
```

以上是对Java对象关系的依赖注入进行了配置，配置了一个id为textHelloService的JavaBean，和一个id为helloClient的JavaBean，其中helloClient对象有两个属性，分别使用常量值和引用值进行了注入（赋值），这样我们就可以直接使用注入好的helloClient对象了，下面编写使用代码：

```
package demo.ioc;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Launcher {

    public static void main(String[] args) {

        try (ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext(
            new String[]{"demo/ioc/applicationContext.xml"});
        ) {
            HelloClient helloClient = ac.getBean("helloClient",
HelloClient.class);
            helloClient.showVersion();//输出 Version is 1.4
            helloClient.textHello();//输出 say hello by text.
        }

    }

}
```

2.2 Spring Boot

SpringBoot一个高度集成的Java应用程序开发管理框架，它实现了应用配置集中化管理、源码（包括依赖包）打包、启动、以及集成了许多开箱即用的组件（Servlet容器、数据库连接池、缓存、ES等），使得开发过程变得容易。

在上节的例子中，我们在完成IoC依赖注入后使用手动编程的方式获取并使用了helloClient对象，如果借助Spring Boot的注解式自动扫描与依赖注入功能，是可以自动获得一个注入好的helloClient对象的，示例代码如下：

添加Maven依赖：

```
<parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.1.1.RELEASE</version>
</parent>

<dependencies>
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <version>2.1.1.RELEASE</version>
          <scope>test</scope>
      </dependency>
<dependencies>
```

编写代码：

```
@Component//访注解表示该类在IoC启动后注册成一个Spring Bean组件
public class TextHelloImpl implements IHello{
// ... 同上
}
```

```
@Component
public class HelloClient {
    @Value("1.2")
    private String version;
    @Autowired
    private IHello helloService;
    // ... 同上
}
```

在最上层包目录添加一个SpringBoot启动类：

```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplicationBootStrap {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplicationBootStrap.class, args);
        System.out.println("DemoApplicationBootStrap started");
    }

}
```

在src/test/java/demo/ioc下创建JUnit测试类，

```
package demo.ioc;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBootIoCTest {

    @Autowired
    private HelloClient helloClient;

    @Test
    public void testAnnotationedIoC(){
        System.out.println(helloClient.getVersion());;
        helloClient.textHello();;
```

```
    }

}
```

在IDE中类中右键，Run，如果一切顺利，在控制台将看到以下结果：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.1.RELEASE)

1.2
say hello by text.
```

这样，一个简单的程序借助于Spring框架便运行起来了。

2.3 Spring MVC

Spring MVC框架是Web MVC设计模式的Java版本的实现，它提供一种基于HTTP协议请求/响应交互模型的轻量级Web开发架构，它实现了模型（M）、视图（V）、控制器（C）层的职责分离和系统解耦，与Spring框架无耦合，并能无缝集成到Servlet容器中。这种基于请求驱动类型的MVC框架把一些Web通用处理逻辑（如参数接收、验证、结果返回等）在框架层面进行了抽象封装，使得开发者无须对这些Web层的固化流程进行重复性编码，只需遵照它的规则去开发特定业务逻辑，从而简化了Web层的开发。

下面演示如何使用MVC开发基于HTTP 协议的API，首先来配置MVC，添加如下配置类：

```
package demo;

import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.DispatcherServlet;
import org.springframework.web.servlet.config.annotation.PathMatchConfigurer;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;

@Configuration
public class WebMvcCustomConfig extends WebMvcConfigurationSupport {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer.setUseSuffixPatternMatch(true);
    }
```

```
    @Bean
    public ServletRegistrationBean<DispatcherServlet> dispatcherRegistration(
            DispatcherServlet dispatcherServlet) {
        ServletRegistrationBean<DispatcherServlet> servletRegistration = new
ServletRegistrationBean<DispatcherServlet>(
                dispatcherServlet);
        servletRegistration.addUrlMappings("*.do");
        servletRegistration.addUrlMappings("*.html");
        return servletRegistration;
    }


}
```

然后编写一个仅仅返回“Hello !”文字的API，

```
package demo.controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String hello(HttpServletRequest request,HttpServletResponse
response) {
        return "Hello !";
    }

}
```

创建Spring Boot配置文件src/main/resources/application.properties并添加

```
server.port = 8445
```

在TemplateApplicationBootStrap类右键，Run，如果一切顺利，在控制台将看到以下输出：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.1.RELEASE)
```

这时，在浏览器地址栏输入http://localhost:8445/hello.do，即可看到“Hello !”文字。

Spring MVC在开发Web Pages应用上也非常方便，它能集成JSP（已过时，不推荐使用）、FreeMarker、Thymeleaf等页面模板技术。下面看一个使用Thymeleaf开发Web Pages的示例：

Maven添加以下依赖：

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
```

在application.properties添加以下配置：

```
# 是否启用缓存，建议生产环境开启
spring.thymeleaf.cache=false
spring.thymeleaf.check-template-location=true
# HTTP Header Content-Type 值
spring.thymeleaf.content-type=text/html
spring.thymeleaf.enabled=true
# 模版编码
spring.thymeleaf.encoding=UTF-8
# 应该从解析中排除的视图名称列表（用逗号分隔）
spring.thymeleaf.excluded-view-names=
# 模版模式
spring.thymeleaf.mode=HTML5
# 模版存放路径
#spring.thymeleaf.prefix=classpath:/templates/
# 模版后缀
spring.thymeleaf.suffix=.html
```

编写Controller：

```
package demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
@Controller
@RequestMapping("/thymeleaf")
public class HiPageController {

    @GetMapping(value = "hiPage")
    public String hi(Model model){
        model.addAttribute("userName","xiaoming");
        return "hiPage";
    }


}
```

编写页面src/main/resources/templates/hiPage.html：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>SpringBoot Thymeleaf模版渲染</title>
</head>
<body>
    <h1>Hi,</h1>
    <p style="background-color:#33D589" th:text="'用户名称：' + ${userName}"/>
</body>
</html>
```

在TemplateApplicationBootStrap类右键，Run，这时，在浏览器地址栏输入http://localhost:8445/thymeleaf/hiPage.html，即看到一个欢迎页面。

3. 单元测试框架 - JUnit

单元测试是程序开发中重要的环节，它不仅能在开发过程帮助我们尽早发现错误，确保代码正确性。Java中比较常用的单元测试框架为JUnit。它能通过非常简单的注解方式，快速对已编写的代码进行单元测试。

在SpringBoot中使用JUnit非常简单，只须在POM文件中加入：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>2.1.1.RELEASE</version>
    <scope>test</scope>
</dependency>
```

下面编写一下测试类src/test/java/demo/junit/SpringBootJUnitTest.java：

```java
package demo.junit;

import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBootJUnitTest {

    @Before
    public void setUp() throws Exception {
        System.out.println("before running test");
    }

    @After
    public void tearDown() throws Exception {
        System.out.println("after running test");
    }

    @Test
    public void testIntEq(){
        Assert.assertEquals(129, 128);
        System.out.println("testIntEq pass");
    }

//    @Ignore("not ready yet")
    @Test
    public void testSameString(){
        Assert.assertSame("这两个字符串不相等","abc","abc");
        System.out.println("testSameString pass");
    }

}
```

运行可以看到以下结果。

```
before running test
after running test
before running test
testSameString pass
after running test
```

以下篇节有些例子会使用JUnit来运行。

4. ORM框架 - MyBatis

MyBatis 是管理Java对象与数据库表之间映射关系的优秀的持久层框架。它通过将 Java 接口和 POJOs(Plain Old Java Objects)映射成数据库中的记录的方式来完成对数据库的基本操作，例如CRUD等。MyBatis会自动将接收到的POJOs解析成SQL参数、设置SQL参数、执行SQL语句以及获取结果集并封装成相应对象，使用者不需要对这些过程进行编码，从而大大简化了数据库的访问。MyBatis 支持XML和注解配置方式。

4.1 代码自动生成器

mybatis generator 是一个自动化生成MyBatis所需 Java 接口（Mapper）和 POJOs和工具，如何使用它生成POJOs和Java接口（Mapper）？有命令行手动执行和IDE插件自动生成两种方式，为简便起见，这里介绍与IDE（Eclipse）集成的方式，首先安装插件，

Eclipse菜单栏 -> Help -> Eclipse Marketplace，搜索框输入MyBatis Generator，在列表结果中点击install按钮，直至完成安装，重启Eclipse。

接下来编写配置文件，在Maven工程目录/src/main/resources/下创建mybatis-generator-config.xml文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN" "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>

    <classPathEntry location="/Applications/Eclipse.app/Contents/MacOS/~/.m2/repository//mysql/mysql-connector-java/8.0.13/mysql-connector-java-8.0.13.jar"/> <!--mysql jdbc包位置，按实际情况更改 -->

    <context id="mysql" defaultModelType="flat">

        <commentGenerator>
            <property name="suppressAllComments"  value="true"/>
        </commentGenerator>

        <!-- jdbc连接 -->
        <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"
                        connectionURL="jdbc:mysql://localhost:3306/test"
                        userId="root" password="123456"/><!--mysql jdbc参数，按实际情况更改 -->

        <!-- 生成实体类 -->
        <javaModelGenerator targetPackage="demo.dao.entity"
                            targetProject="first_maven_project"><!--生成的POJOs位置，按实际情况更改 -->
```

```
            <property name="enableSubPackages" value="true"/>
            <property name="trimStrings" value="false"/>
         </javaModelGenerator>

        <!-- 生成dao接口 -->
        <javaClientGenerator targetPackage="demo.dao.mapper"
                             targetProject="first_maven_project"
type="ANNOTATEDMAPPER">
            <property name="enableSubPackages" value="true"/><!--生成的mapper位
置，按实际情况更改 -->
        </javaClientGenerator>

        <!-- 配置需要生成代码的表信息， tableName 和domainObjectName 按实际情况更改,这
里不生成帮助类examples-->
        <table schema="" tableName="user_base_info"
domainObjectName="UserBaseInfoDO"
               enableCountByExample="false" enableDeleteByExample="false"
               enableSelectByExample="false" enableUpdateByExample="false"/>
    </context>

</generatorConfiguration>
```

然后在该配置文件上点击右键选择 Run As -> Run MyBatis Generator，如果一切顺利，将会在控制台输出"MyBatis Generator Finished"成功提示，这时打开配置的目标代码目录就可以看到相应的代码了。

4.2 使用Mapper访问数据库

以与Spring Boot集成的方式，使用mysql数据库为例说明，添加以下maven依赖：

```
  <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>1.3.0</version>
  </dependency>
  <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.13</version>
  </dependency>
```

在Maven工程目录/src/main/resources/application.properties加入以下配置，实际参数值根据实际情况填写：

```
spring.datasource.url: jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=utf8
spring.datasource.username: root
spring.datasource.password: 123456
spring.datasource.driver-class-name: com.mysql.cj.jdbc.Driver
spring.datasource.type = com.zaxxer.hikari.HikariDataSource
spring.datasource.hikari.maximum-pool-size = 300
spring.datasource.hikari.minimum-idle = 5
spring.datasource.hikari.connection-timeout = 3000
spring.datasource.hikari.read-only = false
spring.datasource.hikari.pool-name = templateHikariCP
```

添加Mapper扫描目录，

```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@MapperScan(basePackages = "demo.dao.mapper")
public class DemoApplicationBootStrap {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplicationBootStrap.class, args);
        System.out.println("DemoApplicationBootStrap started");
    }

}
```

现在就可以编写数据库操作代码了，

```
package demo.dao;


import static org.junit.Assert.assertEquals;

import java.util.Date;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import demo.dao.entity.UserBaseInfoDO;
import demo.dao.mapper.UserBaseInfoDOMapper;
```

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserDaoTest {

    @Autowired
    private UserBaseInfoDOMapper userBaseInfoDOMapper;

    @Test
    public void testSelectByPrimaryKey() {
        UserBaseInfoDO userBaseInfoDO =
userBaseInfoDOMapper.selectByPrimaryKey(6L);
        System.out.println(userBaseInfoDO.getId());
    }

    @Test
    public void testInsert() {
        UserBaseInfoDO record = new UserBaseInfoDO();
        record.setCreateTime(new Date());
        record.setGender(1);
        record.setLastModifyTime(new Date());
        record.setNickName("test022-ju");

        record.setUserId(1880002L);
        record.setUserName("arch022@gmail.com");
        record.setUserNameType(1);

        System.out.println(record);
        int effectedRow = userBaseInfoDOMapper.insert(record);
        System.out.println(effectedRow);
        assertEquals(1,effectedRow);
    }

}
```

以上代码中Mapper中的方法是代码生成器自动生成的，无需手工编写，如果想实现其它方法，只需按照规则在Mapper中添加相应的接口注解及SQL即可。

5. 缓存访问组件 - Jedis

在Spring Boot中已经集成好了访问Redis的组件Jedis、Lettuce，本文以Jedis使用为例进行说明。

添加 Maven依赖：

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
            <exclusions>
```

```
                <exclusion>
                    <groupId>io.lettuce</groupId>
                    <artifactId>lettuce-core</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
        </dependency>
```

在Maven工程目录/src/main/resources/application.properties加入以下配置：

```
spring.redis.host = localhost
spring.redis.port = 6379
spring.redis.database = 0

spring.redis.jedis.pool.max-active = 64
spring.redis.jedis.pool.max-wait = 1000ms
spring.redis.jedis.pool.max-idle = 8
spring.redis.jedis.pool.min-idle = 0
spring.redis.timeout = 2000ms
```

编写测试代码：

```
import java.util.ArrayList;
import java.util.List;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBootJedisTest {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Test
    public void testJedis() throws Exception {
        stringRedisTemplate.opsForValue().set("j_s_100", "测试js100");
        List<String> list =new ArrayList<>();
        list.add("e1");
        list.add("e2");
```

```
        list.add("e3");
        stringRedisTemplate.opsForList().leftPushAll("j_s_list100",list);

 stringRedisTemplate.opsForList().range("j_s_list100",0,-1).forEach(value ->{
           System.out.println(value);
        });
    }

}
```

6. 日志组件

日志是程序运行状态的记录，它能为开发者记录关键点运行时数据和异常堆栈信息，这对于开发者来说往往是至关重要的信息。这里介绍Java中常用的日志组件slf4j和logback组合，其中slf4j是一套外观模式（Facade）的日志接口，它后面可以对接多种不同的slf4j实现。如果使用logback，在Spring Boot工程中，只需要在src/main/resources下添加一个名为logback-spring.xml的文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="false" scanPeriod="60 seconds" debug="false">

    <property name="log_history_retain_days" value="100" /><!--定义一个按天分割的日志保留天数-->
    <property name="log_history_retain_hours" value="240" /><!--定义一个按小时分割的日志保留小时数-->
    <property name="log_dir" value="/your/log/path/" /><!--存放日志的路径-->
    <property name="logLevel" value="debug" /><!--日志的默认级别-->
    <property name="log_pattern" value="%d{MM-dd HH:mm:ss} [%thread] %-5level [%logger.%M %line] - %msg%n%ex" /><!--日志的格式-->

    <appender name="all" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>INFO</level>
        </filter>
        <file>${log_dir}/all.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${log_dir}/all.log.%d{yyyy-MM-dd}</fileNamePattern>
            <maxHistory>${log_history_retain_days}</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${log_pattern}</pattern>
        </encoder>
    </appender>
```

```
    <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
        <!-- encoder 默认配置为PatternLayoutEncoder -->
        <encoder>
            <pattern>${log_pattern}</pattern>
        </encoder>
    </appender>

    <appender name="debug"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>DEBUG</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
        <file>${log_dir}/debug.log</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${log_dir}/debug.log.%d{yyyy-MM-dd-HH}
</fileNamePattern>
            <maxHistory>${log_history_retain_hours}</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${log_pattern}</pattern>
        </encoder>
    </appender>

    <appender name="info"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>INFO</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
        <file>${log_dir}/info.log</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${log_dir}/info.log.%d{yyyy-MM-dd}
</fileNamePattern>
            <maxHistory>${log_history_retain_days}</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${log_pattern}</pattern>
        </encoder>
    </appender>

    <appender name="warn"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>WARN</level>
```

```
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
        <file>${log_dir}/warn.log</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${log_dir}/warn.log.%d{yyyy-MM-dd}
</fileNamePattern>
            <maxHistory>${log_history_retain_days}</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${log_pattern}</pattern>
<!--            <immediateFlush>false</immediateFlush> -->
        </encoder>
    </appender>

    <appender name="error"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>ERROR</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
        <file>${log_dir}/error.log</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${log_dir}/error.log.%d{yyyy-MM-dd}
</fileNamePattern>
            <maxHistory>${log_history_retain_days}</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${log_pattern}</pattern>
        </encoder>
    </appender>
    <!-- name根据实际情况填写工程最顶层包的名称，以便所有类均可被该日志配置覆盖到，level定
义了最低应该
输出的日志级别，比如设置level=info，那么info级别以下（debug)的日志将不会打印-->
    <logger name="demo.log" level="${logLevel}" additivity="false">
        <appender-ref ref="debug" />
        <appender-ref ref="info" />
        <appender-ref ref="warn" />
        <appender-ref ref="error" />
    </logger>

    <root level="${logLevel}">
        <appender-ref ref="all" />
    </root>

</configuration>
```

在以上配置中，定义了6个日志输出器，它们种类分两种，一种是RollingFileAppender，一种是ConsoleAppender。RollingFileAppender代表将日志输出到文件，并可根据策略自动切割，而ConsoleAppender是将日志输出的标准输出设备（stdout)。

在logback中将日志按级别主要分别debug, info,warn,error几种，每种日志一般用于输出不同重要程度的信息，分别使用debug(),info(),warn(),error()方法打印。上面配置文件中的配置是设定debug(),info(),warn(),error()方法打印的内容分别保存到debug.log，info.log，warn.log，error.log文件中。实际中一般会根据需要灵活设置日志的最低打印级别，使低于这个级别的日志将不会被打印，即使在程序中调用日志打印方法。一般开发环境设置debug，线上设置info，这样在程序中加入的debug()方法打印代码，在上线后不用清除也不会打印内容。

以下代码演示如何打印日志：

```
package demo.log;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBootLogbackTest {

    private Logger logger =
LoggerFactory.getLogger(SpringBootLogbackTest.class);

    @Test
    public void testLogback() throws Exception {
        logger.debug("This is {} level msg","debug");
        logger.info("This is {} level msg","info");
        logger.warn("This is {} level msg","warn");
        logger.error("This is {} level msg","error");
    }

}
```

运行，现在去/your/log/path/下对应的文件看打印的内容吧，是不是debug.log没有相应的内容？把配置文件改成：

```
    <property name="logLevel" value="info" /><!--日志的默认级别-->
```

然后重新编译一下（Eclipse 下选择Project -> Clean），再次运行，看看deub.log中是不是没有新的内容了。

日志还有一种重要的应用是用于记录程序中出现各种异常和错误，logback支持打印异常堆栈，以方便定位异常位置，下面代码演示了如何使用打印异常堆栈：

```
    @Test
    public void testExcpetion() throws Exception {
        try {
            int m = 0;
            int n = 5/m;
        } catch(Exception e) {
            logger.error("here error",e);
        }
    }
```

运行这个测试方法后，可以在/your/log/path/error.log中看到类似如下异常信息：

```
07-10 19:37:50 [main] ERROR [demo.log.SpringBootLogbackTest.testExcpetion 32]
- here error
java.lang.ArithmeticException: / by zero
  at
demo.log.SpringBootLogbackTest.testExcpetion(SpringBootLogbackTest.java:30)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
  at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.j
ava:43)
  at java.lang.reflect.Method.invoke(Method.java:498)
  at
org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.ja
va:50)
  at
org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.jav
a:12)
  at
org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java
:47)
  at
org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:
17)
  at
org.springframework.test.context.junit4.statements.RunBeforeTestExecutionCallb
acks.evaluate(RunBeforeTestExecutionCallbacks.java:74)
  at
org.springframework.test.context.junit4.statements.RunAfterTestExecutionCallba
cks.evaluate(RunAfterTestExecutionCallbacks.java:84)
  at
org.springframework.test.context.junit4.statements.RunBeforeTestMethodCallback
s.evaluate(RunBeforeTestMethodCallbacks.java:75)
  at
org.springframework.test.context.junit4.statements.RunAfterTestMethodCallbacks
.evaluate(RunAfterTestMethodCallbacks.java:86)
```

```
  at
org.springframework.test.context.junit4.statements.SpringRepeat.evaluate(Sprin
gRepeat.java:84)
  at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)
  at
org.springframework.test.context.junit4.SpringJUnit4ClassRunner.runChild(Sprin
gJUnit4ClassRunner.java:251)
  at
org.springframework.test.context.junit4.SpringJUnit4ClassRunner.runChild(Sprin
gJUnit4ClassRunner.java:97)
  at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
  at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
  at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
  at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
  at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
  at
org.springframework.test.context.junit4.statements.RunBeforeTestClassCallbacks
.evaluate(RunBeforeTestClassCallbacks.java:61)
  at
org.springframework.test.context.junit4.statements.RunAfterTestClassCallbacks.
evaluate(RunAfterTestClassCallbacks.java:70)
  at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
  at
org.springframework.test.context.junit4.SpringJUnit4ClassRunner.run(SpringJUni
t4ClassRunner.java:190)
  at
org.eclipse.jdt.internal.junit4.runner.JUnit4TestReference.run(JUnit4TestRefer
ence.java:89)
  at
org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:41)
  at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunn
er.java:541)
  at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunn
er.java:763)
  at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.ja
va:463)
  at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.j
ava:209)
```

从打印的异常信息我们可以清楚的看到是哪里出现的问题以及运行过程的方法调用栈信息。