相对比较完整的案例，这个案例一旦做完了，其实你基本就掌握了spring boot在web系统中的常见的开发功能了，很多系统直接就可以用了

大家做过很多业务系统的话，其实一些开发框架的整合以及使用，是最最基础的，是一个初级工程师的必备能力\

spring boot + spring mvc + spring + mybatis：4个框架如何整合起来的

1、Spring Boot + Spring Core

Spring Boot默认原生就是支持Spring Core的，只要在main类上加了@ComponantScan，就会自动去扫描各种spring bean，同时基于@Autowired可以完成自动装配

2、Spring Boot + Spring MVC + Spring Core

如果要在Spring Boot中使用Spring MVC，就意味着你肯定是要开发web应用了，那么就在pom.xml中声明对spring-boot-starter-web的依赖，可以让spring boot自动配置好内置tomcat容器，同时支持spring mvc接收http请求

接着可以在包目录中创建你的Controller，对Controller而言，一般会使用@RestController注解，因为现在大多数都是前后端彻底分离的架构，很少再有Controller接收请求之后再去渲染视图的了，只要提供RESTful接口即可

Controller可以基于@Autowired来装配后面的@Service业务逻辑组件

3、Spring Boot + Spring MVC + Spring Core + MyBatis

对于常规的j2ee应用来说，一般都是轻量级的Spring MVC + Spring Core + MyBatis框架架构来进行开发，Spring MVC负责MVC层，Spring Core负责业务逻辑层，MyBatis负责ORM层，那么就需要将MyBatis整合到Spring Boot中来使用

网上有各种乱七八糟的spring boot和mybatis整合的文章，但是都不太标准和靠谱，标准的整合方式，就是基于mybatis提供的mybatis-spring-boot-starter，基于近乎0配置的方式，完成spring boot和mybatis的整合

spring boot 1.5以上的版本，对应的是mybatis-spring-boot-starter的1.3版本

mybatis-spring-boot-starter的工作原理如下

（1）自动发现一个注册好的DataSource：applicationContext.xml手动配置，不用了 -> @Configuration作用替代掉以前的xml配置文件

（2）自动创建一个SqlSessionFactory，并且将DataSource传入SqlSessionFactory中

（3）自动基于SqlSessionFactory创建一个SqlSessionTemplate

（4）扫描所有的Mapper，将SqlSessionTemplate注入其中，然后将Mapper注册到Spring容器上下文中

4、Spring Boot + Spring MVC + Spring Core + MyBatis + Druid整合案例

4.1 在pom.xml中引入需要的依赖

<!-- 引入mybatis-spring-boot-starter依赖，用于mybatis与spring boot整合 -->

<dependency>

   <groupId>org.mybatis.spring.boot</groupId>

   <artifactId>mybatis-spring-boot-starter</artifactId>

   <version>1.3.1</version>

</dependency>

<!-- 引入mysql驱动依赖 -->

<dependency>

   <groupId>mysql</groupId>

   <artifactId>mysql-connector-java</artifactId>

   <scope>runtime</scope>

</dependency>

<!-- 引入spring-boot-starter-data-jpa依赖 -->

<dependency>

   <groupId>org.springframework.boot</groupId>

   <artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<!-- 引入阿里的druid连接池依赖 -->

<dependency>

   <groupId>com.alibaba</groupId>

   <artifactId>druid</artifactId>

   <version>1.1.6</version>

</dependency>

4.2 在application.properties配置文件中配置druid连接池

# 驱动配置信息

spring.datasource.type=com.alibaba.druid.pool.DruidDataSource

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/oa?useUnicode=true&characterEncoding=utf-8

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driverClassName=com.mysql.jdbc.Driver

# 连接池的配置信息

# 连接池初始大小

spring.datasource.initialSize=5

# 连接池最小空闲连接数量

spring.datasource.minIdle=5

# 连接池最大活跃连接数量

spring.datasource.maxActive=20

spring.datasource.maxWait=60000

spring.datasource.timeBetweenEvictionRunsMillis=60000

spring.datasource.minEvictableIdleTimeMillis=300000

spring.datasource.validationQuery=SELECT 1 FROM DUAL

spring.datasource.testWhileIdle=true

spring.datasource.testOnBorrow=false

spring.datasource.testOnReturn=false

spring.datasource.poolPreparedStatements=true

spring.datasource.maxPoolPreparedStatementPerConnectionSize=20

spring.datasource.filters=stat,wall,log4j

spring.datasource.connectionProperties=druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000

4.3 编写Druid连接池配置管理类

// 使用@Configuration标注，表示这是一个配置管理类

@Configuration

public class DruidDBConfig {

   // 因为spring boot是默认开启了资源过滤的

   // 所以这里的配置，都会自动从application.properties配置文件中加载出来，设置到这个@Configuration类中

   @Value("${spring.datasource.url}")

   private String dbUrl;

   @Value("${spring.datasource.username}")

   private String username;

   @Value("${spring.datasource.password}")

   private String password;

   @Value("${spring.datasource.driverClassName}")

   private String driverClassName;

   @Value("${spring.datasource.initialSize}")

   private int initialSize;

   @Value("${spring.datasource.minIdle}")

   private int minIdle;

   @Value("${spring.datasource.maxActive}")

   private int maxActive;

   @Value("${spring.datasource.maxWait}")

   private int maxWait;

   @Value("${spring.datasource.timeBetweenEvictionRunsMillis}")

   private int timeBetweenEvictionRunsMillis;

   @Value("${spring.datasource.minEvictableIdleTimeMillis}")

   private int minEvictableIdleTimeMillis;

   @Value("${spring.datasource.validationQuery}")

   private String validationQuery;

   @Value("${spring.datasource.testWhileIdle}")

   private boolean testWhileIdle;

   @Value("${spring.datasource.testOnBorrow}")

   private boolean testOnBorrow;

   @Value("${spring.datasource.testOnReturn}")

   private boolean testOnReturn;

   @Value("${spring.datasource.poolPreparedStatements}")

   private boolean poolPreparedStatements;

   @Value("${spring.datasource.maxPoolPreparedStatementPerConnectionSize}")

   private int maxPoolPreparedStatementPerConnectionSize;

   @Value("${spring.datasource.filters}")

   private String filters;

   @Value("{spring.datasource.connectionProperties}")

   private String connectionProperties;

   // 在这个配置类中，直接基于配置信息，创建出了一个bean

   // 这个bean就是一个DataSource

   // 这个DataSource bean就会被纳入spring容器的管理范围之内

   @Bean

   @Primary

   public DataSource dataSource(){

       // 这里就是用外部加载进来的配置信息，创建出来一个Druid连接池

       DruidDataSource datasource = new DruidDataSource();

       datasource.setUrl(this.dbUrl);

       datasource.setUsername(username);

       datasource.setPassword(password);

       datasource.setDriverClassName(driverClassName);

       //configuration

       datasource.setInitialSize(initialSize);

       datasource.setMinIdle(minIdle);

       datasource.setMaxActive(maxActive);

       datasource.setMaxWait(maxWait);

       datasource.setTimeBetweenEvictionRunsMillis(timeBetweenEvictionRunsMillis);

       datasource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);

       datasource.setValidationQuery(validationQuery);

       datasource.setTestWhileIdle(testWhileIdle);

       datasource.setTestOnBorrow(testOnBorrow);

       datasource.setTestOnReturn(testOnReturn);

       datasource.setPoolPreparedStatements(poolPreparedStatements);

       datasource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);

       try {

           datasource.setFilters(filters);

      } catch (SQLException e) {

           e.printStackTrace();

      }

       datasource.setConnectionProperties(connectionProperties);

       return datasource;

  }

}

编写这个类之后，还需要在Application类中导入这个配置管理类

@SpringBootApplication

// 使用@Import就可以将其他的配置管理类导入进来

@Import(DruidDBConfig.class)

public class Application {

   public static void main(String[] args) {

       SpringApplication.run(Application.class, args);

  }

}

4.4 编写Domain类

在com.zhss.springboot.domain包下，创建一个Employee类，作为用于测试的领域模型

public class User {

   private Long id;

   private String name;

   private Integer age;

   public Long getId() {

       return id;

  }

   public void setId(Long id) {

       this.id = id;

  }

   public String getName() {

       return name;

  }

   public void setName(String name) {

       this.name = name;

  }

   public Integer getAge() {

       return age;

  }

   public void setAge(Integer age) {

       this.age = age;

  }

}

4.5 编写Mapper类

@Mapper

public interface UserMapper {

   @Select("SELECT \* FROM user")

   List<User> listUsers();

   @Select("SELECT \* FROM user WHERE user\_id = #{userId}")

   User getUserById(@Param("userId") Long userId);

   @Insert("INSERT INTO user(name, age) VALUES(#{name}, #{age})")

   void saveUser(User user);

   @Update("UPDATE user SET name=#{name}, age=#{age} WHERE user\_id=#{userId}")

   void updateUser(User user);

   @Delete("DELETE FROM user WHERE user\_id=#{userId}")

   void removeUser(@Param("userId") Long userId);

}

public interface UserDAO {

   List<User> listUsers();

   User getUserById(Long userId);

   void saveUser(User user);

   void updateUser(User user);

   void removeUser(Long userId);

}

@Repository

public class UserDAOImpl implements UserDao {

   @Autowired

   private UserMapper userMapper;

   public List<User> listUsers() {

       return userMapper.listUsers();

  }

   public User getUserById(Long userId) {

       return userMapper.getUserById();

  }

   public void saveUser(User user) {

       userMapper.saveUser(user);

  }

   public void updateUser(User user) {

       userMapper.updateUser(user);

  }

   public void removeUser(Long userId) {

       userMapper.removeUser(userId);

  }

}

4.6 编写Service类

在com.zhss.springboot.service包下，创建一个EmployeeService接口

在com.zhss.springboot.service.impl包下，创建一个EmployeeServiceImpl实现类

public interface UserService {

   List<User> listUsers();

   User getUserById(Long userId);

   void saveUser(User user);

   void updateUser(User user);

   void removeUser(Long userId);

}

@Service

public class UserServiceImpl implements UserService {

   @Autowired

   private UserDAO userDAO;

   public List<User> listUsers() {

       return userDAO.listUsers();

  }

   public User getUserById(Long userId) {

       return userDAO.getUserById();

  }

   public void saveUser(User user) {

       userDAO.saveUser(user);

  }

   public void updateUser(User user) {

       userDAO.updateUser(user);

  }

   public void removeUser(Long userId) {

       userDAO.removeUser(userId);

  }

}

4.7 编写controller类

@RestController

// 这里直接在类级别给一个/user的映射，就是代表用户管理的请求都会到这里来

@RequestMapping(value="/user")

public class UserController {

   @Autowired

   private UserService userService;

   // GET请求代表着是查询数据

   @RequestMapping(value="/", method=RequestMethod.GET)

   public List<User> listUsers() {

       return userService.listUsers();

  }

   // GET请求+{id}代表的是查询某个用户

   @RequestMapping(value="/{userId}", method=RequestMethod.GET)

   public User getUser(@PathVariable("userId") Long userId) {

       return userService.getUserById(userId);

  }

   // POST请求代表着是新增数据

   @RequestMapping(value="/", method=RequestMethod.POST)

   public String saveUser(User user) {

       userService.saveUser(user);

       return "success";

  }

   // PUT请求代表的是更新

   @RequestMapping(value="/{userId}", method=RequestMethod.PUT)

   public String updateUser(@PathVariable("userId") Long userId, User user) {

       user.setUserId(userId);

       userService.updateUser(user);

       return "success";

  }

   // DELETE请求代表的是删除

   @RequestMapping(value="/{userId}", method=RequestMethod.DELETE)

   public String deleteUser(@PathVariable("userId") Long userId) {

       userService.removeUser(user);

       return "success";

  }

}

4.8 启动程序，然后浏览器访问：http://localhost:8080/employee/1

此时会在浏览器中看到，将id=1的员工信息查询了出来，显示到了浏览器中

curl命令的使用，通过curl命令，可以替代浏览器去进行RESTful接口的一个测试，git命令行里面，就可以直接使用curl命令

curl -XPOST URL -d "param1=value1&param2=value2"

curl -XGET URL

curl -XDELETE URL

curl -XPUT URL

curl -XPUT URL -H 'Content-Type:application/json' -d'

{

   "param1":"value1",

   "param2":"value2"

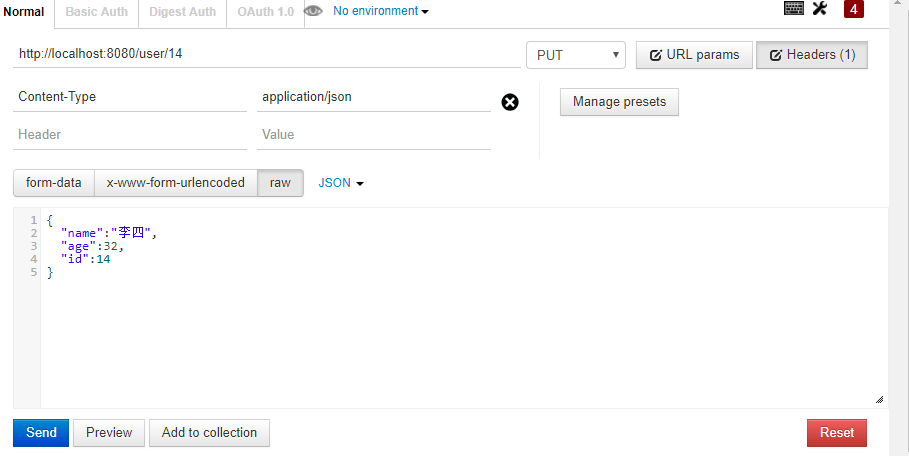
}

'

介绍一下，我平时常用的吧，chrome postman插件，chrome浏览器，postman插件

其实在spring boot做restful接口的时候，PUT请求和DELETE请求是有问题的，因为spring mvc对RESTful接口的支持会有一点问题

所以在做开发的时候，一般对于PUT，POST，请求，他们的请求参数，都不是用表单直接提交的，一般要求是用ajax走content-type是application/json的类型，通过json类型发送请求参数过来



4.9 梳理一下此时的几个框架整合的思路

（1）系统启动的时候，首先会去扫描DruidDBConfig类，这就可以将外部的druid连接池配置加载进来，同时初始化出来一个druid连接池的DataSource bean

（2）mybatis-spring-boot-starter接着开始干活儿，发现了一个DataSource bean，就会将其注入SqlSessionFactory，再创建SqlSessionTemplate，接着扫描Mapper接口，将SqlSessionTemplate注入每个Mapper，然后将Mapper放入spring容器中来管理

（3）spring boot的@ComponantScan注解开始干活儿，自动扫描所有的bean，依次初始化实例以及注入依赖，EmployeeServiceImpl（将EmployeeMapper注入其中），EmployeeCongtroller（将EmployeeServiceImpl注入其中）

（4）此时浏览器发送请求，会先由controlller处理，接着调用service，再调用mapper。mapper底层会基于druid连接池访问到数据库，进行数据操作

至此，最常规的几个框架整合和使用，就完成了

5、mybatis-spring-boot-starter的高级使用

5.1 如果要实现mybatis的完整功能

mybatis的各种高阶功能，原来都是在xml mapper文件中完成的，但是现在都是倾向于无xml化，全部基于java类和注解来完成

那么就直接在mapper接口中，基于mybatis的注解来实现mybatis的所有功能即可

5.2 如果要对mybatis进行配置

mybatis之前都是有一个mybatis-config.xml之类的文件，可以在里面对mybatis本身进行配置的，比如二级缓存之类的，但是在这里，都是在spring boot的application.properties文件中进行配置了。具体要设置什么参数，可以关注一下mybatis的参数配置信息

mybatis.type-aliases-package=com.example.domain.model

mybatis.type-handlers-package=com.example.typehandler

mybatis.configuration.map-underscore-to-camel-case=true

mybatis.configuration.default-fetch-size=100

mybatis.configuration.default-statement-timeout=30

6、目前为止发现的spring boot几大特点总结

（1）对常用的依赖进行统一的版本约束，让各依赖的版本都可以兼容，通过spring-boot-starter-parent中的dependencyManagement实现

（2）简化配置，内置web容器、Auto Configuration、内置打包插件，都是让你写了代码就能立即运行起来

比如mybatis-spring-boot-starter，直接自己按照约定，找到一个数据源就可以完成实现所有的事情，不需要再进行繁琐的xml配置

比如spring-boot-starter-web，也是按照约定，直接完成默认的web程序相关的配置，比如内置web容器的接口、spring mvc的配置，等等

最后发现，整合常用的框架，除了在外部文件里，配置了Druid数据库连接池的一些东西，其他框架几乎没有配置，全部都是基于默认的约定完成

内置的打包插件可以让我们不用做任何配置，直接就完成一个fat jar的打包，可以执行的jar包了就是

然后内置的web容器，直接java -jar就可以启动web容器，并且运行web程序，接着就可以本地调试和测试了

（3）一句话，spring boot，核心目的就是要简化我们的系统开发

简化再简化，开发非常简单，尽量不要配置，基于约定，记住，这是核心思想

同时为了简化系统开发整个流程中的方方面面，后面spring boot还提供了其他很多的东西，到后面我们会再讲解