Web后端框架技术
—Spring Boot + Mybatis

南京大学软件学院

Web后端技术概述

简介

web后端服务主要指在服务器中执行的逻辑运算和数据处理，它为前端提供访问服务。

前后端分离是互联网项目开发的业界标准使用方式。会为以后的大型分布式架构、弹性计算架构、微服务架构、多端化服务（多种客户端例如：浏览器，车载终端，安卓，IOS等等）打下坚实的基础。

核心思想是前端HTML页面通过AJAX调用后端的RESTFUL API接口并使用JSON数据进行交互。

分离模式的优势

- **前后端流量大幅减少**

  前后端流量的大头是HTML/JS/IMG资源，流量的减少在于“前端静态化”这个规则。在第一次获取后，静态资源会被浏览器缓存，即使浏览器继续轮询，服务端也会返回一个非常小Not Modified响应，流量几乎可以忽略不计。

- **表现性能的提高**

  页面的绝大部分内容都是本地缓存直接加载，前端加载和响应速度得到非常大的提高。

分离模式的优势

- **安全性方面的集中优化**

  前端静态化以后，前端页面攻击几无可能，一些注入式攻击在分离模式下被很好的规避； 而后端安全问题集中化了，仅仅只处理一个RESEful接口的安全考虑，安全架设和集中优化变得更明确和便利。

- **开发和构架的分离**

  前端构架能集中考虑性能和优化，而业务、安全和存储等问题就能集中到后端考虑。

- **前后端平台和技术无关**

  不再纠结于平台和技术的选择。

架构

- **API接入层**

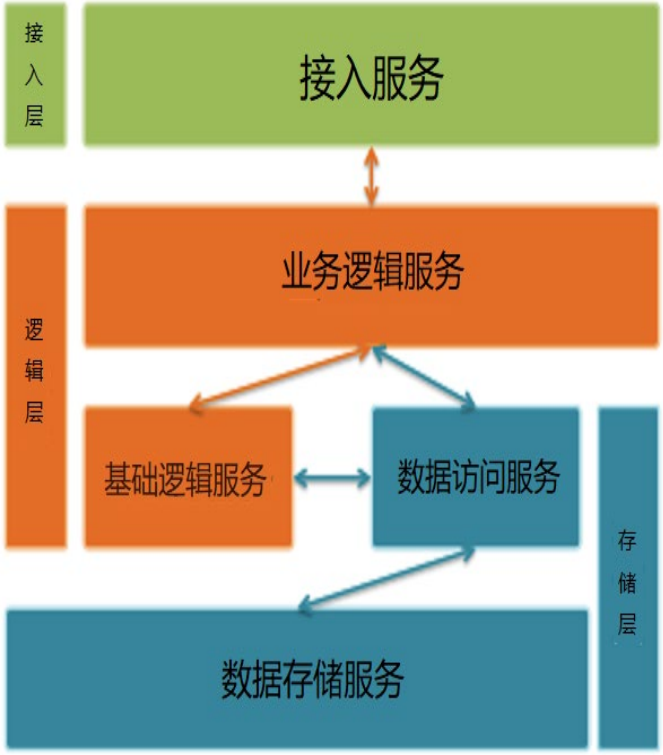
  负责把用户请求分发到业务逻辑层，主要解决跟用户的连接问题。

- **业务逻辑层**

  负责实现具体的业务功能，满足业务需求。从产品的角度看它是整个后台的核心，不论是接入层还是数据存储层都是为它做支撑的。

- **数据存储层**

  负责保存业务所需的数据，提供业务数据的读写支撑。



RESTful API

REST，表示性状态转移（representation state transfer）。简单来说，就是用URI表示资源，用HTTP方法(GET， POST， PUT， DELETE)表征对这些资源的操作。

RESTful API 就是REST风格的API。现在终端平台多样，移动、平板、PC等许多媒介向服务端发送请求后，如果不使用RESTful API，则需要为每个平台的数据请求定义相应的返回格式，以适应前端显示。而RESTful API 要求前端以一种预定义的语法格式发送请求，那么服务端就只需要定义一个统一的响应接口，不必像之前那样解析各色各式的请求。

MVC 框架

MVC框架的核心思想是：解耦，让不同的代码块之间降低耦合，增强代码的可扩展性和可移植性，实现向后兼容。

- M为Model，主要封装对数据库层的访问，内嵌ORM框架，实现面向对象的编程来操作数据库，不用考虑数据库的差异性，简单配置就可以完成数据库切换。
- V为View，用于封装结果，内嵌了模板引擎，实现动态展示数据。
- C为Controller，用于请求，处理业务逻辑，与Model和View交互，返回结果。

IoC 框架

IoC，控制反转（Inversion of Control）。IoC理论提出的观点大体是这样的：借助于“第三方”实现具有依赖关系的对象之间的解耦。

这时候，在实现A时，无须再去考虑A与B、C和D之间的依赖关系。

IoC 框架

实现IoC的方法：依赖注入。

所谓依赖注入，就是由IoC容器在运行期间，动态地将某种依赖关系注入到对象之中。依赖注入(DI)和控制反转(IOC)是从不同的角度描述的同一件事情，就是指通过引入IoC容器，利用依赖注入的方式，实现对象之间的解耦。

ORM 框架

对象关系映射（Object Relational Mapping，简称ORM）模式是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术。简单的说，ORM是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系数据库中。

ORM解决的主要问题是对象关系的映射。域模型和关系模型分别是建立在概念模型的基础上的。域模型是面向对象的，而关系模型是面向关系的。一般情况下，一个持久化类和一个表对应，类的每个实例对应表中的一条记录，类的每个属性对应表的每个字段。

ORM 框架

特点：

- 提高了开发效率。由于ORM可以自动对Entity对象与数据库中的Table进行字段与属性的映射，所以我们实际可能已经不需要一个专用的、庞大的数据访问层。
- ORM提供了对数据库的映射，不用sql直接编码，就能够像操作对象一样从数据库获取数据。

缺点：

- 自动化进行关系数据库的映射需要消耗系统性能。
- 在处理多表联查、where条件复杂之类的查询时，ORM的语法会变得复杂。

Spring Boot

简介

Spring是重量级企业开发框架 Enterprise JavaBean（EJB） 的替代品，Spring为企业级Java开发提供了一种相对简单的方法，通过依赖注入和面向切面编程，用简单的 Java对象（Plain Old Java Object，POJO） 实现了EJB的功能。

Spring Boot是Spring开源组织下的子项目，是Spring组件一站式解决方案，它是为了简化Spring应用的创建、运行、调试、部署等而出现的，使用它可以做到专注于Spring应用的开发，而无需过多关注XML的配置。

官方网站：https://spring.io/projects/spring-boot

GitHub源码：https://github.com/spring-projects/spring-boot

Spring起源

要谈Spring的历史，就要先谈J2EE。J2EE应用程序的广泛实现是在1999年和2000年开始的，它的出现带来了诸如事务管理之类的核心中间层概念的标准化，但是在实践中并没有获得绝对的成功，因为开发效率，开发难度和实际的性能都令人失望。

使用EJB开发JAVA EE应用非常艰苦，很多东西都不能一下子就很容易的理解。EJB要严格地实现各种不同类型的接口，类似的或者重复的代码大量存在。而配置也是复杂和单调，同样使用JNDI进行对象查找的代码也是单调而枯燥。虽然有一些开发工作随着xdoclet的出现，而有所缓解，但是学习EJB的高昂代价，和极低的开发效率，极高的资源消耗，都造成了EJB的使用困难。而Spring出现的初衷就是为了解决类似的这些问题。

Spring起源

Spring的一个最大的目的就是使JAVA EE开发更加容易。同时，Spring之所以与Struts、Hibernate等单层框架不同，是因为Spring致力于提供一个以统一的、高效的方式构造整个应用，并且可以将单层框架以最佳的组合揉和在一起建立一个连贯的体系。可以说Spring是一个提供了更完善开发环境的框架，可以为POJO对象提供企业级的服务。

Spring的形成，最初来自Rod Jahnson所著的一本很有影响力的书籍《Expert One-on-One J2EE Design and Development》，就是在这本书中第一次出现了Spring的一些核心思想，该书出版于2002年。另外一本书《Expert One-on-One J2EE Development without EJB》，更进一步阐述了在不使用EJB开发JAVA EE企业级应用的一些设计思想和具体的做法。

出现原因

虽然Spring的组件代码是轻量级的，但它的配置却是重量级的（需要大量XML配置）。Spring 2.5引入了基于注解的组件扫描，这消除了大量针对应用程序自身组件的显式XML配置。Spring 3.0引入了基于Java的配置，这是一种类型安全的可重构配置方式，可以代替XML。

尽管如此，我们依旧没能逃脱配置的魔爪。开启某些Spring特性时，比如事务管理和Spring MVC，还是需要用XML或Java进行显式配置。启用第三方库时也需要显式配置，比如基于Thymeleaf的Web视图。配置Servlet和过滤器（比如Spring的DispatcherServlet）同样需要在web.xml或Servlet初始化代码里进行显式配置。组件扫描减少了配置量，Java配置让它看上去简洁不少，但Spring还是需要不少配置。

出现原因

配置这些XML文件占用了我们大部分时间和精力。除此之外，相关库的依赖也非常让人头疼，不同库之间的版本冲突也非常常见。

而Spring Boot让这一切成为了过去。从本质上来说，Spring Boot就是Spring，它做了那些没有它你自己也会去做的Spring Bean配置。

Spring 旨在简化J2EE企业应用程序开发，Spring Boot 旨在简化Spring开发。

特性

- **独立运行**

  Spring Boot内嵌了各种servlet容器，Tomcat、Jetty等，不需要打成war包部署到容器中，只要打成一个可执行的jar包就能独立运行，所有的依赖包都在一个jar包内。

- **简化配置**

  spring-boot-starter-web启动器自动依赖其他组件，减少了maven的配置。

- **自动配置**

  Spring Boot能根据当前类路径下的类、jar包来自动配置bean，如添加一个spring-boot-starter-web启动器就能拥有web的功能，无需其他配置。

特性

- **无代码生成和XML配置**

  Spring Boot配置过程中无代码生成，也无需XML配置文件就能完成所有配置工作。

- **应用监控**

  Spring Boot 监控核心是 spring-boot-starter-actuator 依赖，增加依赖后，Spring Boot 会默认配置一些通用的监控，比如 jvm 监控、类加载、健康监控等。

Spring Boot常见依赖项

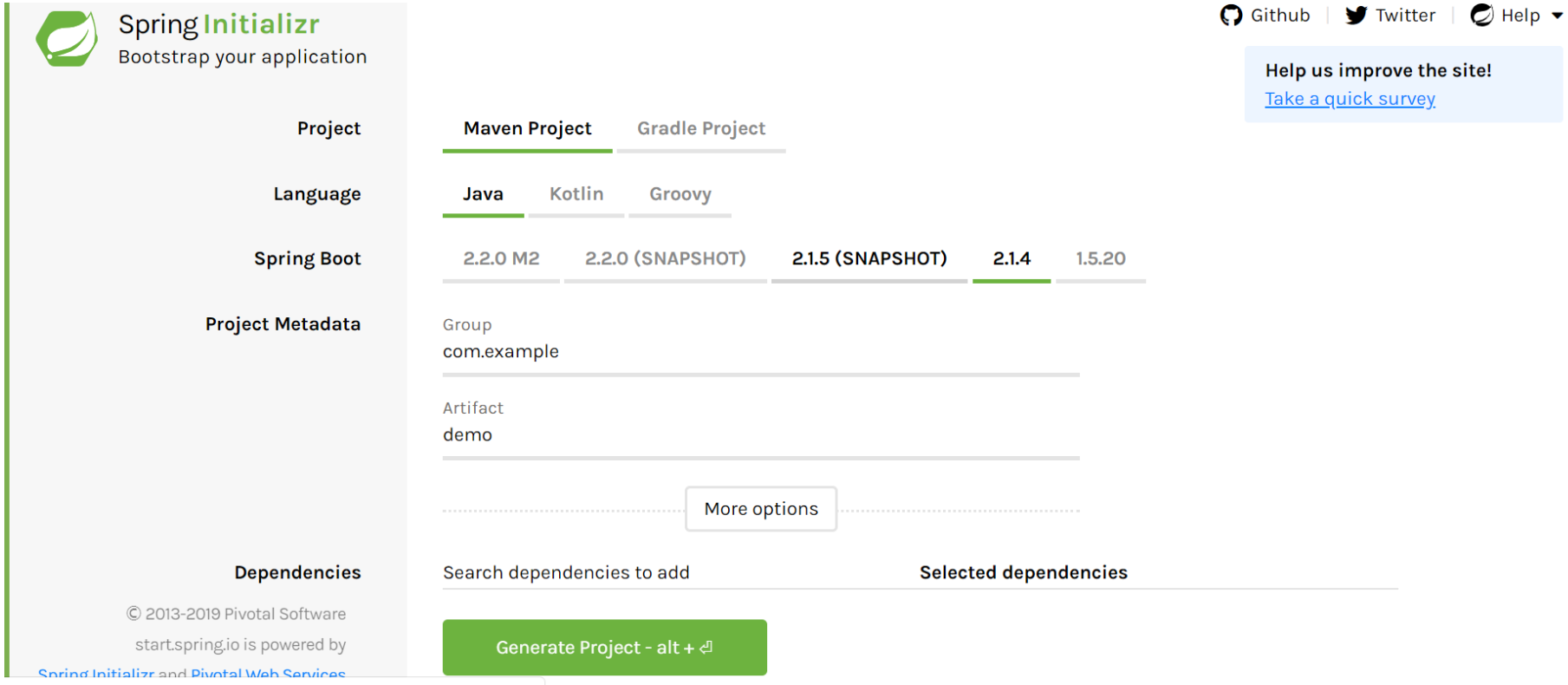
- spring-boot-starter：核心 POM，包含自动配置支持、日志库和对 YAML 配置文件的支持。
- spring-boot-starter-aop：包含 spring-aop 和 AspectJ 来支持面向切面编程（AOP）
- spring-boot-starter-data-jpa：包含 spring-data-jpa、spring-orm 和 Hibernate 来支持 JPA
- spring-boot-starter-datamongodb：对 MongoDB NOSQL 数据库的支持，包括 spring-data-mongodb
- spring-boot-starter-jdbc：支持使用 JDBC 访问数据库
- spring-boot-starter-security：包含 spring-security。

Spring Boot常见依赖项

- spring-boot-starter-test：包含常用的测试所需的依赖
- spring-boot-starter-thymeleaf：对 Thymeleaf 模板引擎的支持
- spring-boot-starter-web：支持 Web 应用开发，包含 Tomcat 和 spring-mvc
- spring-boot-starter-actuator：添加适用于生产环境的功能，如性能指标和监测等功能
- spring-boot-starter-mail：对 javax.mail 的支持
- spring-boot-starter-jetty：导入 Jetty HTTP 引擎（作为 Tomcat 的替代）
- spring-boot-starter-log4j：添加 Log4j 的支持
- spring-boot-starter-logging：导入Spring Boot 默认的日志框架 Logback
- spring-boot-starter-tomcat：导入 Spring Boot 默认的 Tomcat 作为应用服务器

创建Spring Boot项目

1. 使用Spring Initializr的Web界面: https://start.spring.io/

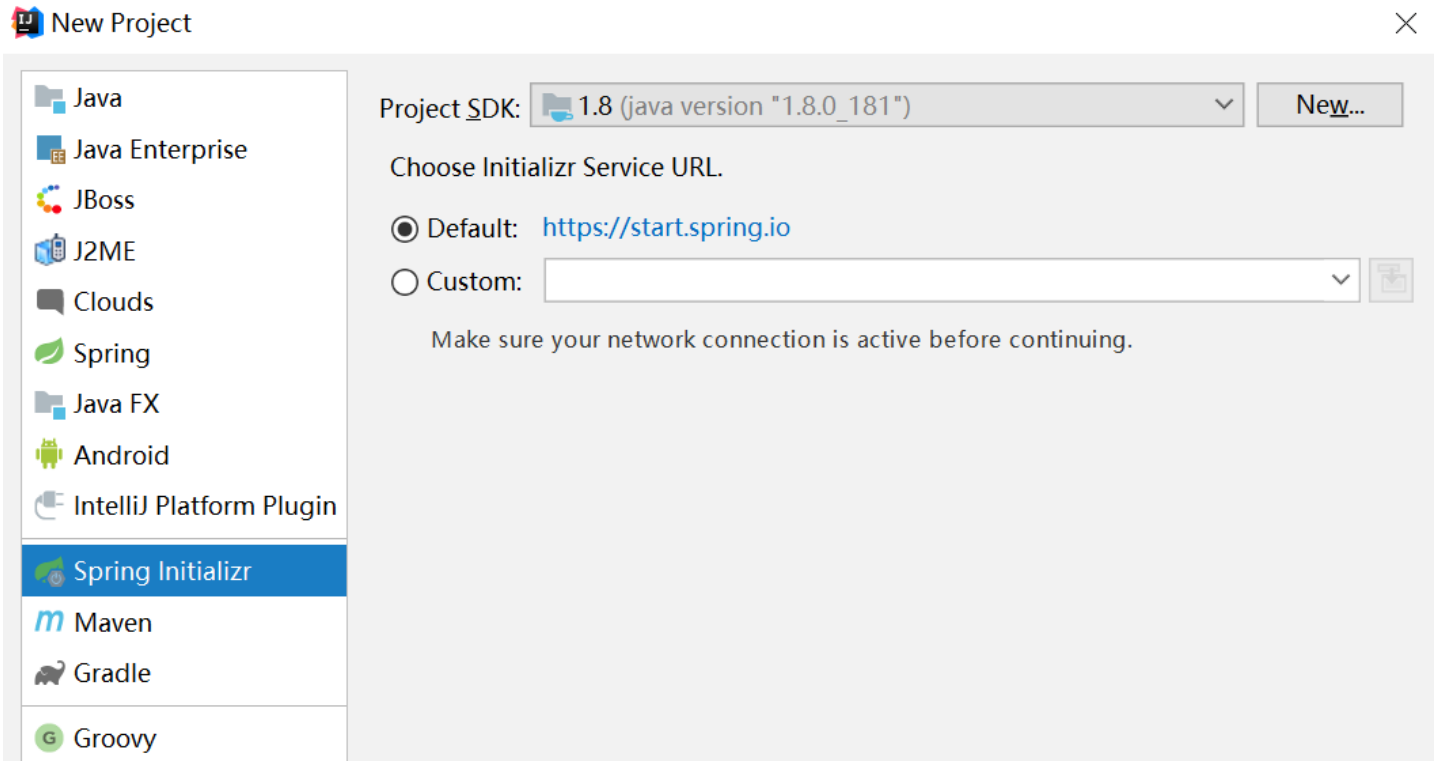


将创建好的项目导入IDE

创建Spring Boot项目

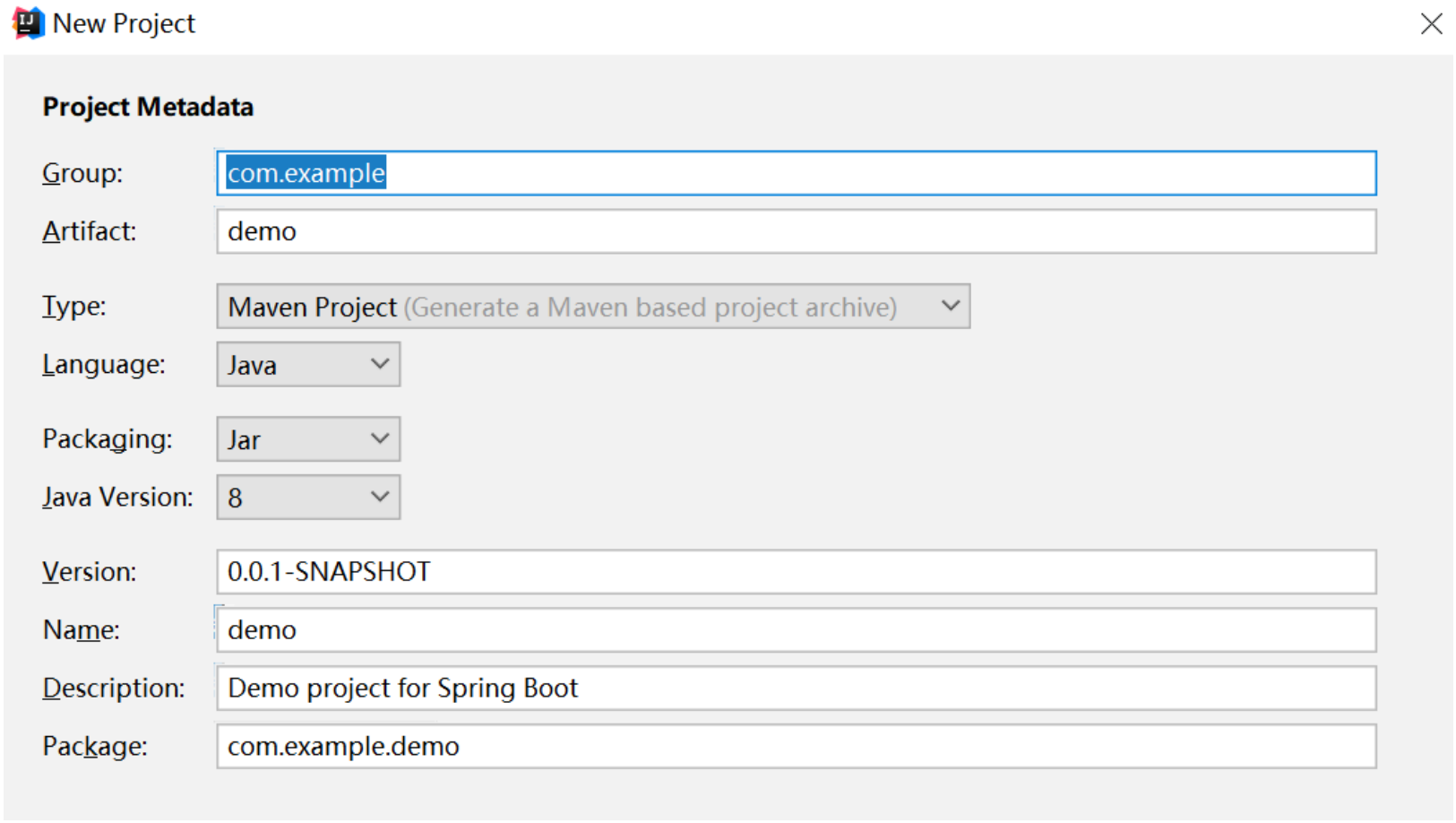
2.在IntelliJ IDEA里创建Spring Boot项目

在File菜单里选择New > Project



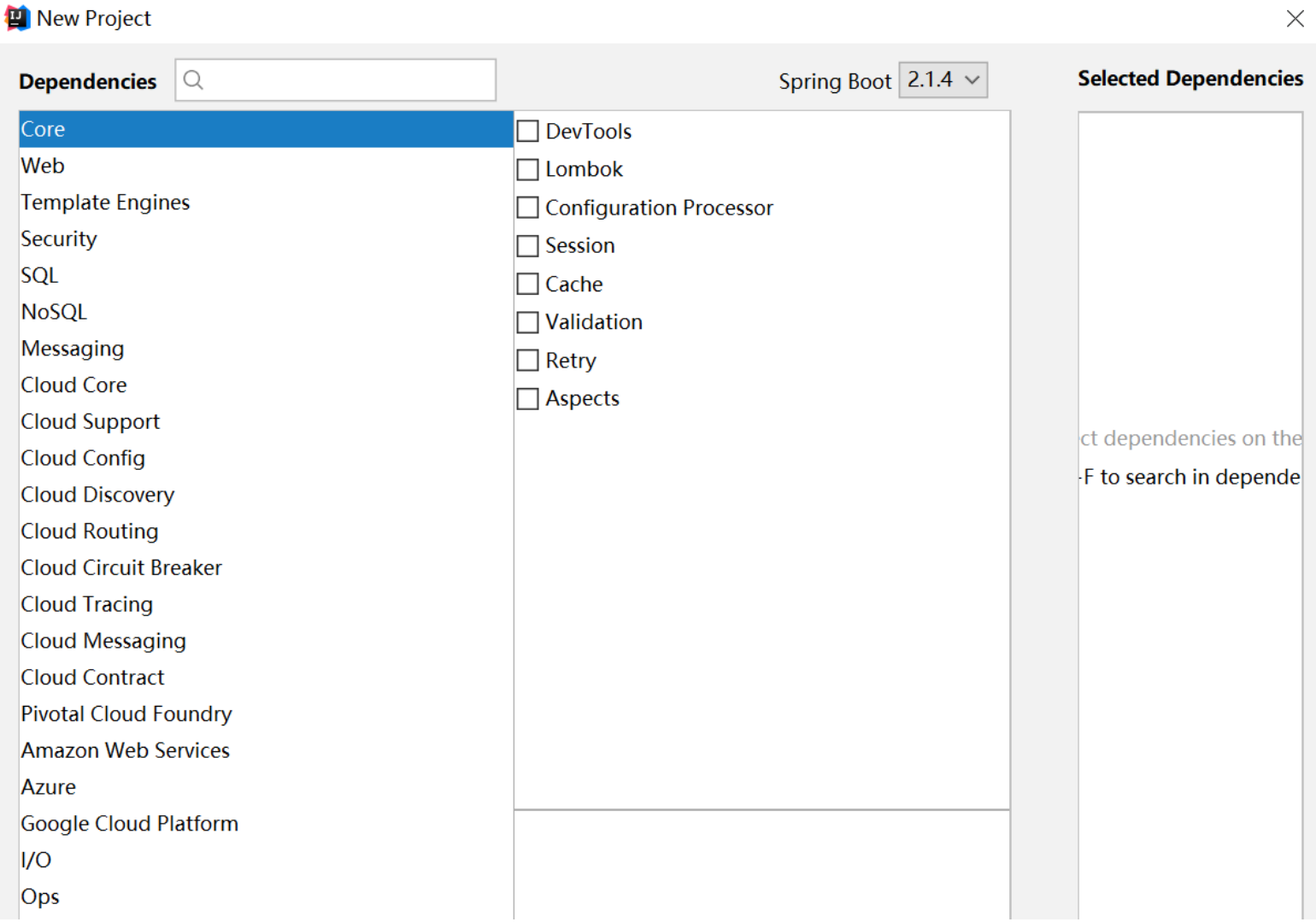
创建Spring Boot项目

填写项目的一些基本信息



创建Spring Boot项目

选择依赖



Spring Boot配置文件

Spring Boot提供了两种常用的配置文件，配置文件名是固定的，分别是application.properties文件和application.yml文件。他们的作用都是修改Spring Boot自动配置的默认值。

- **YAML**

YAML以空格的缩进程度来控制层级关系，只要左边空格对齐则视为同一个层级(不能用tab代替空格，大小写敏感)。YAML支持字面值，对象，数组三种数据结构，也支持复合结构。

✓ 字面值：字符串，布尔类型，数值，日期。字符串默认不加引号，单引号会转义特殊字符。日期格式支持yyyy/MM/dd HH:mm:ss

✓ 对象：由键值对组成，形如 key:(空格)value(空格必须要有)，每组键值对占用一行，且缩进的程度要一致。也可以使用行内写法：{k1: v1, ...kn: vn}

Spring Boot配置文件

- ✓ 数组：由形如 -(空格)value 的数据组成(空格必须要有)，每组数据占用一行，且缩进的程度要一致。也可以使用行内写法： [1,2,...n]
- ✓ 复合结构：上面三种数据结构任意组合

示例

```yaml
yaml:
 str: 字符串可以不加引号
 specialStr: "双引号直接输出\n特殊字符"
 specialStr2: '单引号可以转义\n特殊字符'
 flag: false
 num: 666
 Dnum: 88.88
 list:
  - one
  - two
  - two
 set: [1,2,2,3]
 map: {k1: v1, k2: v2}
 positions:
  - name: ITDragon
    salary: 15000.00
  - name: ITDragonBlog
    salary: 18888.88
```

Spring Boot配置文件

- **properties**

语法结构形如：key=value。

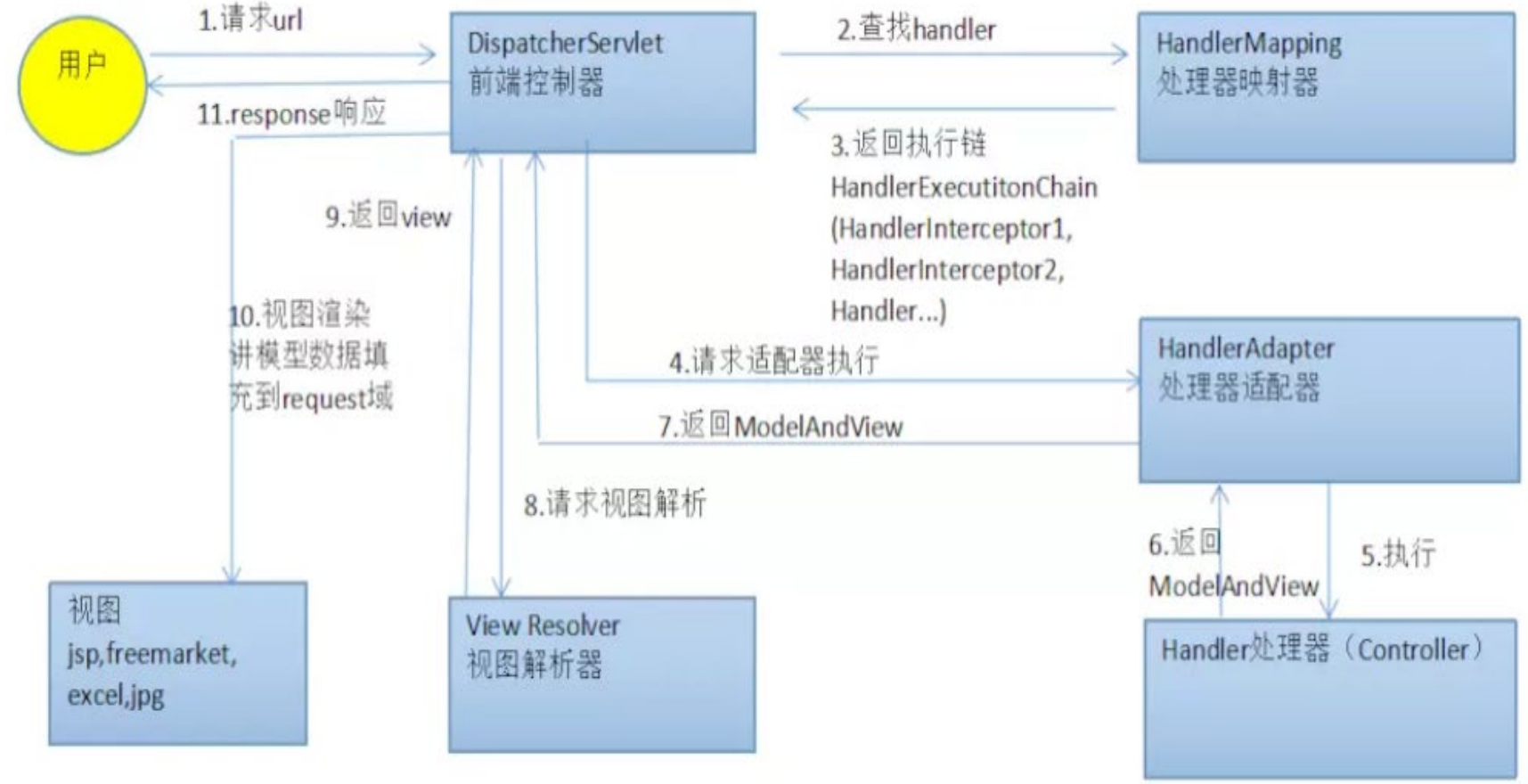
示例

```
1  userinfo.account=itdragonBlog
2  userinfo.age=25
3  userinfo.active=true
4  userinfo.created-date=2018/03/31 16:54:30
5  userinfo.map.k1=v1
6  userinfo.map.k2=v2
7  userinfo.list=one,two,three
```

Spring Boot运行方式

- 使用mvn install/gradle bootRepackage 生成项目的jar包，然后运行jar
- 项目根目录下执行mvn spring-boot:run / gradlew bootRun命令
- 直接执行 main 方法运行

Spring MVC执行流程



Spring Boot核心注解

Spring boot项目的入口类中含有main方法，这是一个标准的Java应用程序的入口方法。核心注解为@SpringBootApplication

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) { SpringApplication.run(DemoApplication.class, args); }

}
```

@SpringBootApplication 为组合注解，主要包含以下几个注解

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
```

Spring Boot核心注解

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。@Configuration是 Spring 3.0 添加的一个注解，用来代替 applicationContext.xml 配置文件，所有这个配置文件里面能做到的事情都可以通过这个注解所在类来进行注册。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan：用来代替配置文件中的 component-scan 配置，开启组件扫描，即自动扫描包路径下的 @Component 注解进行注册 bean 实例到 context 中。

Spring Boot常用注解

@ResponseBody：表示该方法的返回结果直接写入HTTP response body中，一般在异步获取数据时使用，用于构建RESTful的API。

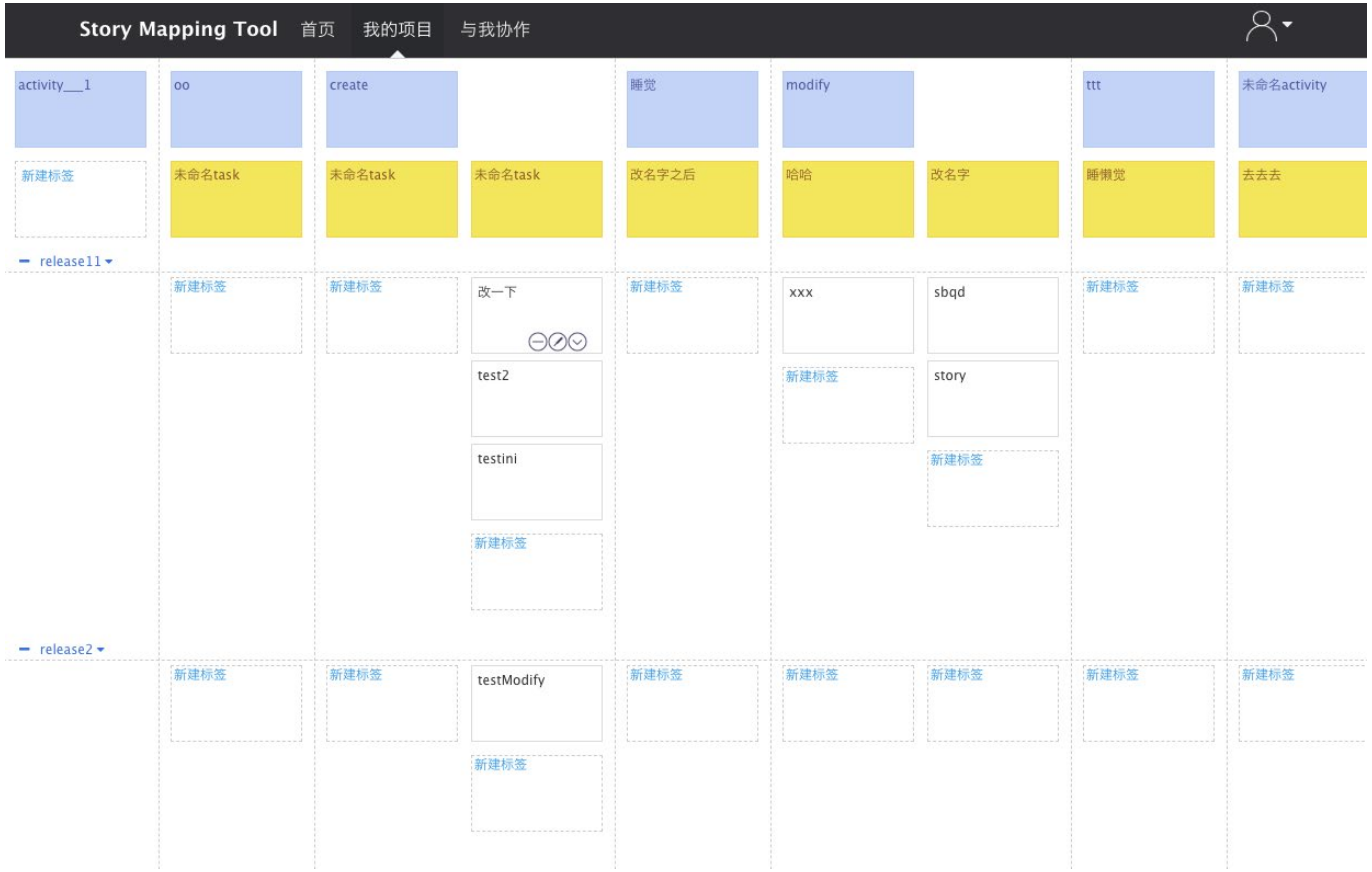
@Controller：用于标注控制层组件。

@Service：用于标注service层的组件。

@Repository：使用@Repository注解可以确保DAO或者repositories提供异常转译，这个注解修饰的DAO或者repositories类会被ComponetScan发现并配置，同时也不需要为它们提供XML配置项。

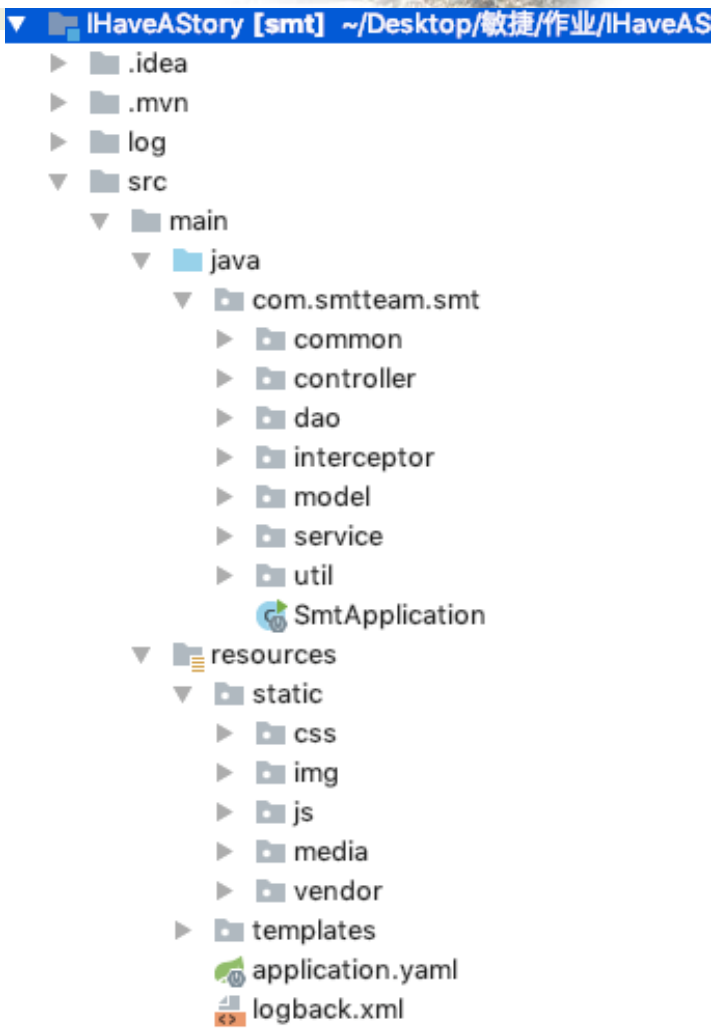
Example

- 如图所示一个web应用的功能是管理敏捷开发流程中的用户故事，即增删改查和拖动story（图中的白色卡片）。



Example

- 整个项目的代码结构如图所示。
- 项目的主类SmtApplication要放在最外层包中，因为springboot会自动扫描主类所在的包路径下的注解，注册 bean 实例到 context 中。
- 一般来说controller、service、dao、model是必要的，model中存放了跟数据库表映射的实体类，其余各层下面会详细讲解。



Controller层(RestController)

```
@RestController
@RequestMapping("/story")
public class StoryController {
    @Autowired
    private StoryService storyService;
}
```

- @Controller对应表现层的Bean。这里用RestController，默认类中的方法都会以json的格式返回给前端。
- 使用@Controller注解标识StoryController之后，就表示要把StoryController交给Spring容器管理，在Spring容器中会存在一个名字为”storyController”的bean。
- 如果@Controller不指定名称，则默认的bean名字为这个类的类名首字母小写，如果指定名称【@Controller(value=“StoryController”)】或者【@Controller(“StoryController”)】，则使用value作为bean的名字。

Controller层(RequestMapping)

```
@RestController
@RequestMapping("/story")
public class StoryController {
    @Autowired
    private StoryService storyService;
}
```

- @RequestMapping注解用于映射url到控制器类或一个特定的处理程序方法。可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。
- @RequestMapping注解常用属性有：
  - value指定请求的地址
  - method指定请求类型， GET、POST、PUT、DELETE等
  - consumes指定请求的接收内容类型，如application/json, text/html
  - produces指定返回内容类型。
  - params指定request中必须包含某些参数值，才让该方法处理请求。
  - headers指定request中必须包含某些指定的header值，才让该方法处理请求。

Controller层 (RequestParam)

```
@RestController
@RequestMapping("/story")
public class StoryController {
    @Autowired
    private StoryService storyService;
    /**
     * 交换两个story
     */
    @PostMapping("/exchange")
    public ResultVO<Story> exchangeById(@RequestParam int src_id,
                                        @RequestParam int tar_id){…}
}
```

- @RequestParam从request里面取参数值。
- 使用http:///IP:port/story/exchange?src_id=num1&tar_id=num2访问， 将num1和num2传递赋值给src_id和tar_id
- @PostMapping("/url") 等同于@RequestMapping(value = "/url",method = RequestMethod.POST)

Controller层(PathVariable)

```
/**根据task获取对应的所有story*/
@GetMapping("/list/{taskId}")
public ResultVO<List<Story>> getByTask(@PathVariable int taskId){…}
```

- @PathVariable映射url片段到java方法的参数。
- 使用http://IP:port/story/list/num访问，num将作为参数传递给taskId
- @GetMapping(“/url”) 等同于@RequestMapping(value = "/url",method = RequestMethod.GET)
- @PathVariable 支持三种参数
  - name 指定绑定参数的名称，要跟URL上面的一样
  - required 指定这个参数是不是必须的
  - value 跟name一样的作用，是name属性的一个别名
  - @RequestParam支持四种参数，加上了defaultValue，表示如果本次请求没有携带这个参数，或者参数为空，那么就会启用默认值

Controller层(RequestBody)

```
/**新增story*/
@PostMapping("/create")
public ResultVO<Story> createStory(@RequestBody StoryVO storyInput){…}
```

- @RequestBody映射请求体到java方法的参数，一般不用于get请求。
- 前端访问需要传递JSON字符串，字符串定义了StoryVO的属性值。
- 前端使用ajax访问方式如右图示例。

```
var data = {
      "taskId": $("#taskId").val(),
      "name": $("#name").val(),
      "storyPoint": $("#storyPoint").val(),
      "priority": $("#priority").val(),
      "description":
$("#description").val(),
      "posId": $("#posId").val(),
      "acceptance":
$("#acceptance").val(),
      "releaseId": $("#iteration").val(),
}
$.ajax({
      type: "POST",
      url: "/story/create",
      dataType:"json",
      contentType : 'application/json',
      data:JSON.stringify(data),
   success: function (data) {…}
})
```

Controller层

```
@GetMapping("/storyMapping")
public ModelAndView getMapping(@RequestParam int proId){
    ModelAndView modelAndView = new ModelAndView("storymap");
    modelAndView.addObject("proId",proId);
    return modelAndView;
}
```

- controller除了下发操作到数据库，还用于跳转界面，返回值为ModelAndView类型。
- ModelAndView的构造参数storymap是指要跳转的目标页面storymap.html的路径。
- ModelAndView可以携带参数，这个方法传递了proId给目标页面，在storymap.html文件中接收方式如下：

```
<script th:inline="javascript">
    var proId = [[${proId}]]
</script>
```

Service层

```
@Service
public class StoryServiceImpl implements StoryService {
    @Autowired
    private StoryDao storyDao;
}
```

- @Service对应业务层的Bean。
- @Service注解是告诉Spring，Spring容器中会存在StoryServiceImpl的bean，当Controller需要使用StoryServiceImpl的的实例时，就可以将Spring创建好的bean注入。在Controller只需要声明一个变量storyService来接收，不用通过new StoryServiceImpl()实例化。接收方式如下

```
@RestController
@RequestMapping("/story")
public class StoryController {
    @Autowired
    private StoryService storyService;
}
```

- @Autowired作用是自动装配bean，而无需再为field设置getter,setter方法。

Repository层

```
@Repository
public interface StoryDao extends JpaRepository<Story, Integer> {
    @Modifying
    @Query("update Story set posId=posId+1 where posId>:posId")
    void updateCreatePosID(@Param("posId") int posId);

    List<Story> findByTaskIdOrderByPosIdDesc(int taskId);

    List<Story> findByTaskId(int taskId);
}
```

- @Repository对应数据访问层的Bean。表示Spring容器中会存在StoryDao的bean，当Service需要使用StoryDao的实例时，就可以将Spring创建好的bean注入：在Service只需要声明一个变量storyDao来接收。接收方式如下：

```
@Service
public class StoryServiceImpl implements StoryService {
    @Autowired
    private StoryDao storyDao;
}
```

JpaRepository

```
@Repository
public interface StoryDao extends JpaRepository<Story, Integer> {
    @Modifying
    @Query("update Story set posId=posId+1 where posId>:posId")
    void updateCreatePosID(@Param("posId") int posId);

    List<Story> findByTaskIdOrderByPosIdDesc(int taskId);

    List<Story> findByTaskId(int taskId);
}
```

- JpaRepository实现一组JPA规范相关的方法， StoryDao接口继承了JpaRepository，就具备了通用的数据访问控制层的能力。
- JpaRepository支持接口规范方法名查询。意思是如果在接口中定义的查询方法符合它的命名规则，就可以不用写实现。比如findByTaskIdOrderByPosIdDesc实现的就是... Where story.taskId = ?1 order by story.posId desc。

JpaRepository

```
@Repository
public interface StoryDao extends JpaRepository<Story, Integer> {
    @Modifying
    @Query("update Story set posId=posId+1 where posId>:posId")
    void updateCreatePosID(@Param("posId") int posId);

    List<Story> findByTaskIdOrderByPosIdDesc(int taskId);

    List<Story> findByTaskId(int taskId);
}
```

- 也可以使用@Query自定义查询。 @Query与 @Modifying 两个annotation一起声明，可定义个性化更新操作，例如只涉及某些字段更新时最为常用。
- 本例的query语句意思是由于各个story有顺序关系，在新插入一个story时需要将它后面的story的位置id加一。

Mybatis

简介

MyBatis是一款优秀的持久层框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生类型、接口和 Java 的 POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

优点

- **简单易学**

  本身就很小且简单。没有任何第三方依赖，易于学习，易于使用，通过文档和源代码，可以比较完全的掌握它的设计思路和实现。

- **灵活**

  不会对应用程序或者数据库的现有设计强加任何影响。 sql写在xml里，便于统一管理和优化。通过sql基本上可以实现我们不使用数据访问框架可以实现的所有功能，或许更多。

- **解除sql与程序代码的耦合**

  通过提供DAL层，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。sql和代码的分离，提高了可维护性。

优点

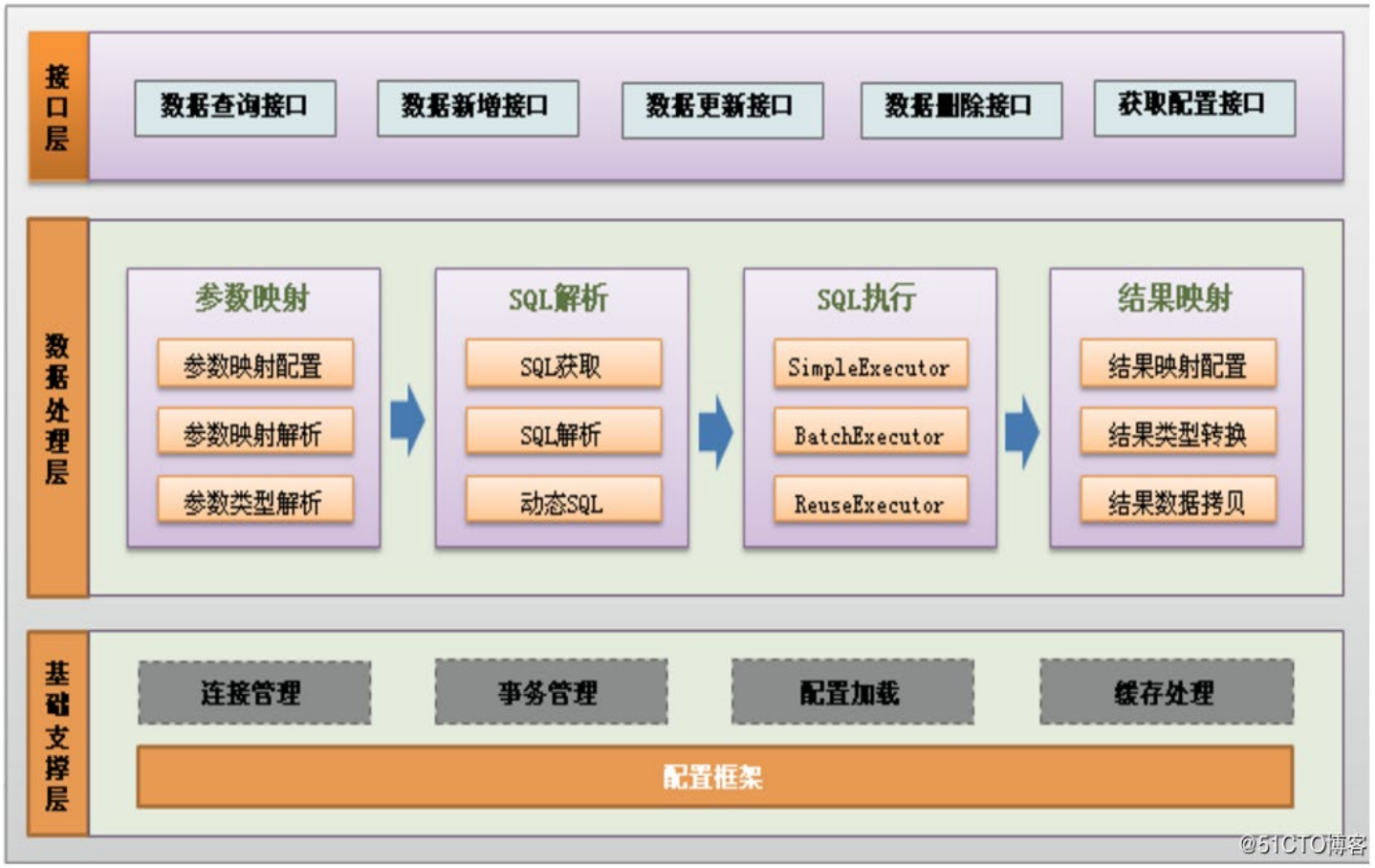
- 提供映射标签，支持对象与数据库的orm字段关系映射。
- 提供对象关系映射标签，支持对象关系组建维护。
- 提供xml标签，支持编写动态sql。

缺点

- 编写SQL语句时工作量很大，尤其是字段多、关联表多时，更是如此。
- SQL语句依赖于数据库，导致数据库移植性差，不能更换数据库。
- 框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际仍然需要自己编写，工作量也比较大，而且不太容易适应快速数据库修改。
- 二级缓存机制不佳。

架构

- API接口层
- 数据处理层
- 基础支撑层



架构

- **API接口层**

  提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。

- **数据处理层**

  负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。

- **基础支撑层**

  负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理。这些都是共用的东西，将他们抽取出来作为最基础的组件，为上层的数据处理层提供最基础的支撑。

第一个Mybatis程序

1. 准备数据库

首先我们创建一个数据库 mybatis，编码方式设置为 UTF-8，然后再创建一个名为 student 的表，插入几行数据：

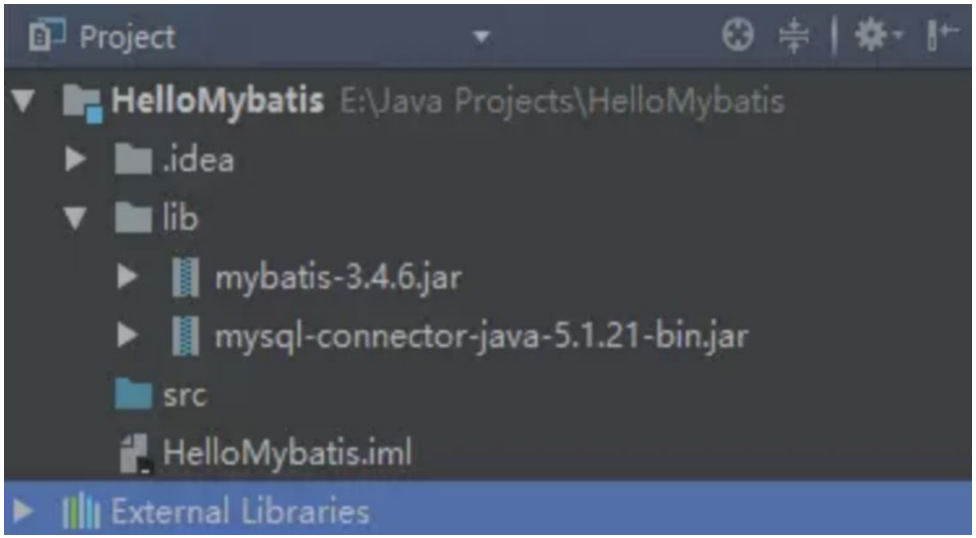
```
DROP DATABASE IF EXISTS mybatis;
CREATE DATABASE mybatis DEFAULT CHARACTER SET utf8;

use mybatis;
CREATE TABLE student(
  id int(11) NOT NULL AUTO_INCREMENT,
  studentID int(11) NOT NULL UNIQUE,
  name varchar(255) NOT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO student VALUES(1,1,'我没有三颗心脏');
INSERT INTO student VALUES(2,2,'我没有三颗心脏');
INSERT INTO student VALUES(3,3,'我没有三颗心脏');
```

第一个Mybatis程序

2. 创建项目

在 IDEA 中新建一个 Java 项目，然后导入必要的 jar 包：

- mybatis-3.4.6.jar
- mysql-connector-java-5.1.21-bin.jar



第一个Mybatis程序

3. 创建实体类

新建实体类 Student，用于映射表 student：

```
package pojo;
public class Student {
    int id;
    int studentID;
    String name;
    /* getter and setter */
}
```

第一个Mybatis程序

4. 配置文件 mybatis-config.xml

创建 MyBaits 的主配置文件 mybatis-config.xml，其主要作用是提供连接数据库用的驱动，数据名称，编码方式，账号密码等：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 别名 -->
    <typeAliases>
        <package name="pojo"/>
    </typeAliases>
    <!-- 数据库环境 -->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
characterEncoding=UTF-8"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>
    <!-- 映射文件 -->
    <mappers>
        <mapper resource="pojo/Student.xml"/>
    </mappers>
</configuration>
```

第一个Mybatis程序

5. 配置文件 Student.xml

新建 Student.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="pojo">
    <select id="listStudent" resultType="Student">
        select * from  student
    </select>
</mapper>
```

由于 mybatis-config.xml 中配置了 <typeAliases> 别名，所以在这里的 resultType 可以直接写 Student，而不用写类的全限定名 pojo.Student。namespace 属性其实就是对 SQL 进行分类管理，实现不同业务的 SQL 隔离。

第一个Mybatis程序

6. 编写测试类

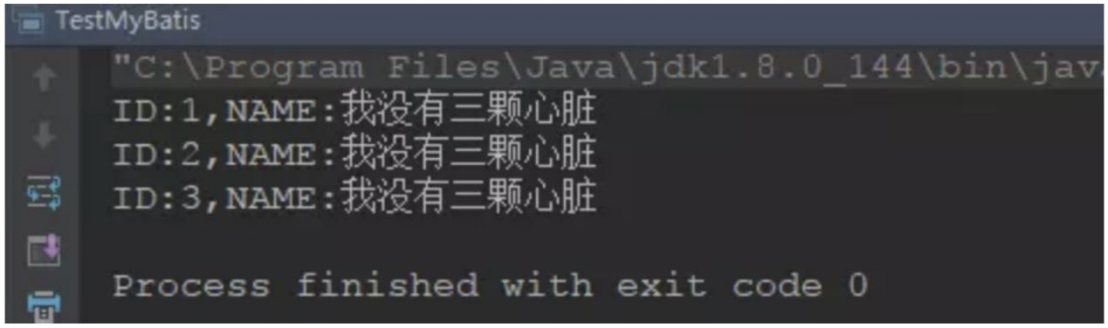
创建测试类 TestMyBatis：

```
public class TestMyBatis {
    public static void main(String[] args) throws IOException {
        // 根据 mybatis-config.xml 配置的信息得到 sqlSessionFactory
        String resource = "mybatis-config.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        // 然后根据 sqlSessionFactory 得到 session
        SqlSession session = sqlSessionFactory.openSession();
        // 最后通过 session 的 selectList() 方法调用 sql 语句 listStudent
        List<Student> listStudent = session.selectList("listStudent");
        for (Student student : listStudent) {
            System.out.println("ID:" + student.getId() + ",NAME:" + student.getName());
        }
    }
}
```

第一个Mybatis程序

7. 运行测试类

运行测试类，控制台输出学生信息：



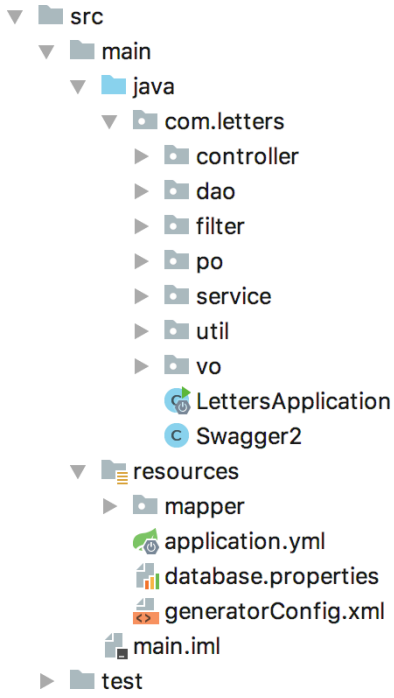
Example

SpringBoot+Mybatis项目实例：

当我们需要展示当前用户收藏的专辑时，我们需要根据已知的用户ID对专辑收藏表进行搜索，查找到当前用户所收藏的所有专辑的ID集合，再根据这些专辑的ID查找到对应专辑的具体信息，最后将专辑具体信息的集合返回到界面展示给用户。

Example

项目整体架构：



Example

yaml配置文件：

```yaml
server:
  port: 8080

spring:
  devtools:
    restart:
      exclude:  static/**,public/**
      enabled:  true
  datasource:
    name: letter
    url: jdbc:mysql://118.25.174.148:3306/letter?characterEncoding=utf-8
    username: root
    password: Letter123
    driver-class-name: com.mysql.cj.jdbc.Driver
mybatis:
  mapper-locations: classpath*:mapper/*Mapper.xml
  type-aliases-package: com.letters.po
pagehelper:
  helperDialect: mysql
  reasonable: true类型为com.letters.dao.ConsultantMapper的变量
  supportMethodsArguments: true
  params: count=countSql
```

Example

1. 创建Controller类

在controller层，创建一个controller类，用以接收和简单处理界面数据。

```
@RestController
@RequestMapping("/api/audio")
public class AudioController {
    private final AudioService audioService;

    @Autowired
    public AudioController(AudioService audioService) { this.audioService = audioService; }

    @GetMapping("/getAudioCollections")
    private ResponseResult<List<Album>> getAudioCollections(@RequestHeader(name = "UserId") int userId) {
        ResponseResult<List<Album>> result = new ResponseResult<>();
        List<Album> albums = audioService.getAlbums(userId);
        result.setData(albums);
        result.setSuccess(true);
        return result;
    }
```

Example

2. 创建Service接口

在service层，创建接口文件并增加查找用户收藏专辑的接口。

```
import java.util.List;

public interface AudioService {

    List<Album> getAlbums(int userId);
```

Example

3. 创建ServiceImpl实现类

在具体实现类中实现查找用户收藏专辑的方法。我们需要根据用户ID信息查找对应的表，找到专辑后返回专辑信息。

```
private final AlbumCollectionMapper albumCollectionMapper;

private final AlbumMapper albumMapper;
@Override
public List<Album> getAlbums(int userId) {
    List<Album> albums = new ArrayList<>();
    List<Integer> albumIds = albumCollectionMapper.selectByUserId(userId);
    for (int albumId: albumIds) {
        albums.add(albumMapper.selectByPrimaryKey(albumId));
    }

    return albums;
}
```

Example

4. 创建DAO层的接口

AlbumCollectionMapper和AlbumMapper都是SpringBoot中的DAO层。

```java
@Repository
public interface AlbumCollectionMapper {
    int deleteByPrimaryKey(AlbumCollectionKey key);

    int insert(AlbumCollection record);

    int insertSelective(AlbumCollection record);

    AlbumCollection selectByPrimaryKey(AlbumCollectionKey key);

    int updateByPrimaryKeySelective(AlbumCollection record);

    int updateByPrimaryKey(AlbumCollection record);

    List<Integer> selectByUserId(int userId);
}
```

```java
package com.letters.dao;

import ...

@Repository
public interface AlbumMapper {
    int deleteByPrimaryKey(Integer id);

    int insert(Album record);

    int insertSelective(Album record);

    Album selectByPrimaryKey(Integer id);

    int updateByPrimaryKeySelective(Album record);

    int updateByPrimaryKey(Album record);

    List<Album> selectByAnchorId(int anchorId);

}
```

Example

5. 创建 Mybatis的映射文件**文件**

具体的数据库查找操作的实现是在Mybatis的映射文件AlbumCollectionMapper.xml和AlbumMapper.xml中实现的。

这里我们需要根据用户的ID信息查找专辑ID，并根据专辑ID查找专辑的具体信息，因此，我们需要在对应表的映射文件中编辑SQL查找语句。

```
<select id="selectByUserId" resultType="java.lang.Integer" parameterType="java.lang.Integer" >
  select album_id
  from album_collection
  where user_id = #{userId,jdbcType=INTEGER}
</select>
```

```
<select id="selectByAnchorId" resultMap="BaseResultMap" parameterType="java.lang.Integer" >
  select
  <include refid="Base_Column_List" />
  from album
  where anchor_id = #{anchorId,jdbcType=INTEGER}
</select>
```

Example

6. 前端页面展示

搜索到的专辑返回到界面，展示给用户。