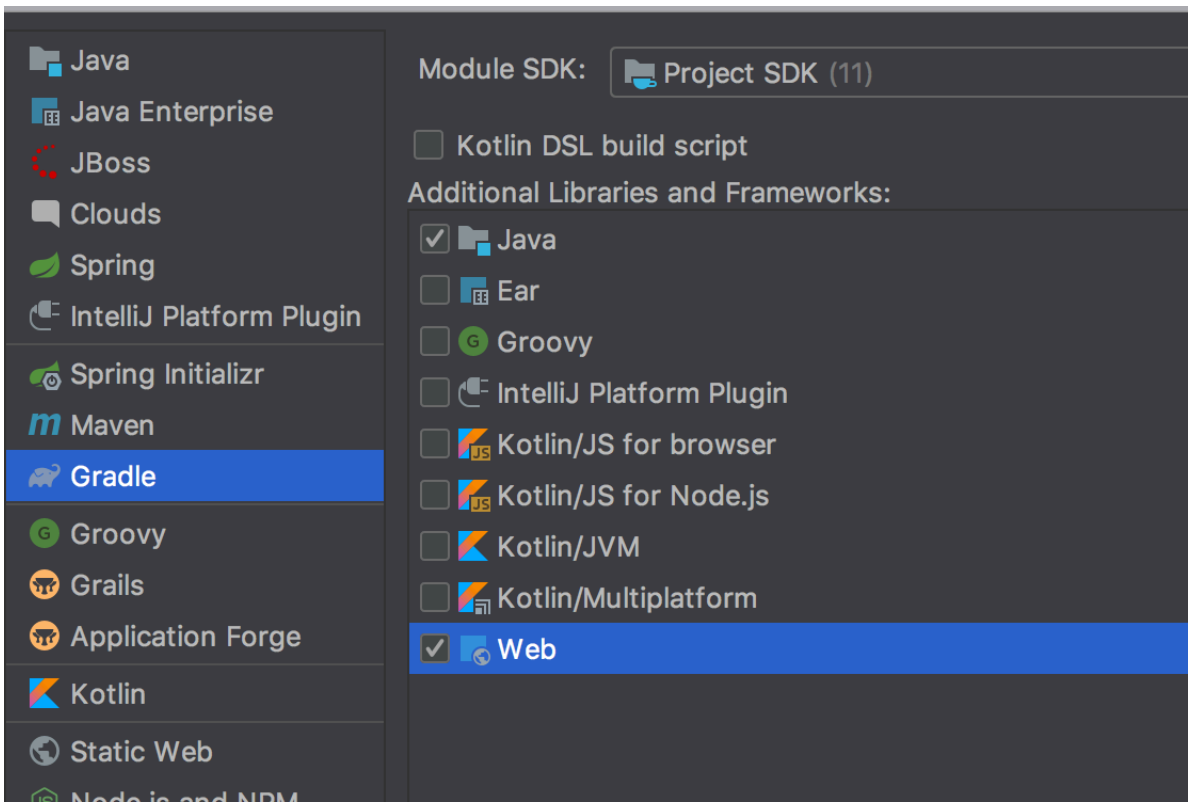
Part IV. SpringMVC 源码深度分析

Code branch "springmvc" - https://github.com/WildCapriccio/SpringSourceCode-ver5.1.x/tree/springmvc

0. 创建新Module 并 配置tomcat 【坑！】

0.1 创建新Module `spring-lagou-yu-mvc`

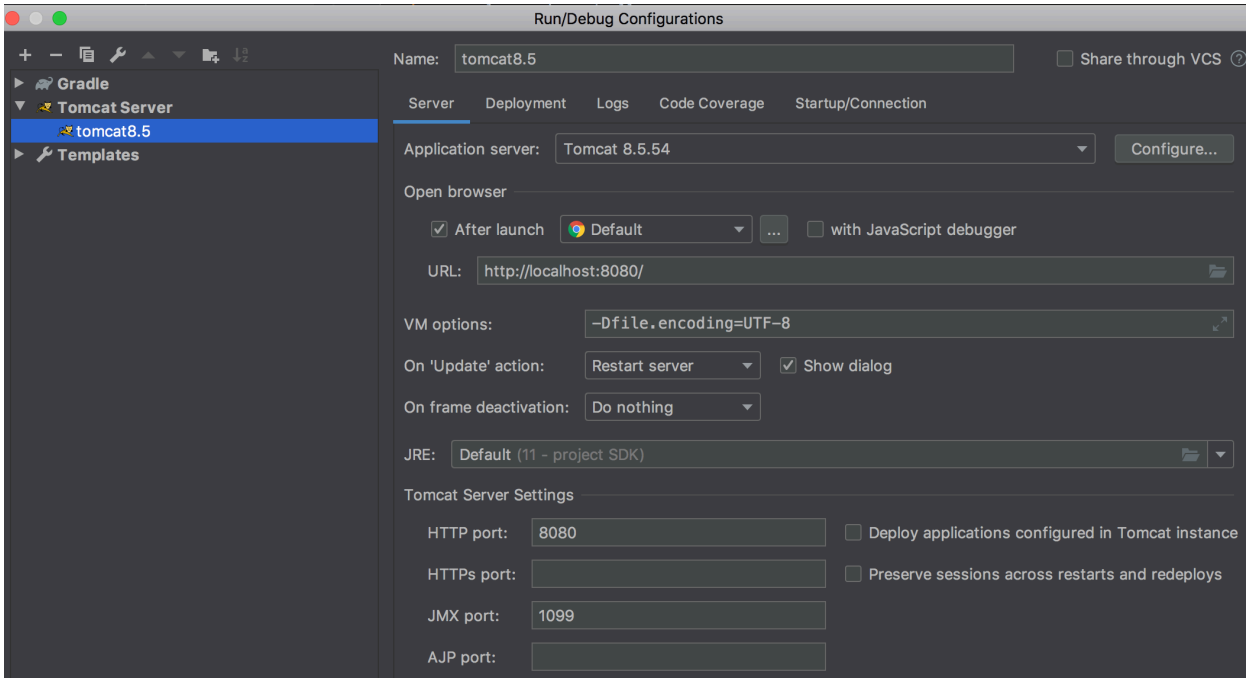
勾选 java 和 web：



0.2 配置tomcat

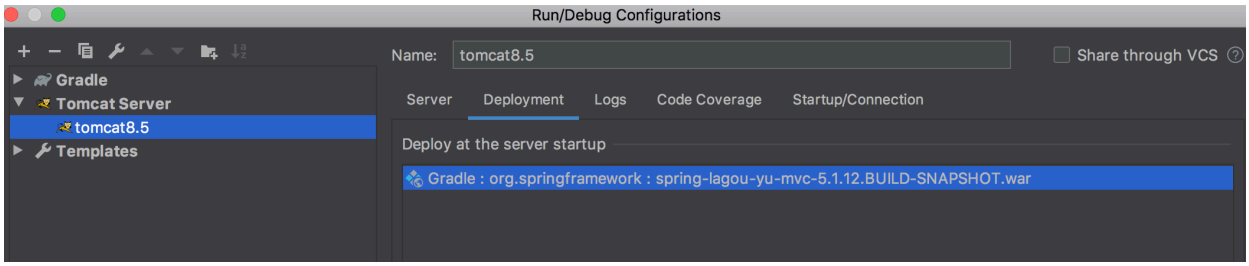
千万不要用tomcat 7.0！不然会报类似错误【link】

1. 官网下载tomcat 8.5.54 的 tar 包，解压后放在 "/Library/" 中
2. 在IntelliJ中，Edit configuration，填好下表（Configure.., URL，VM options）：



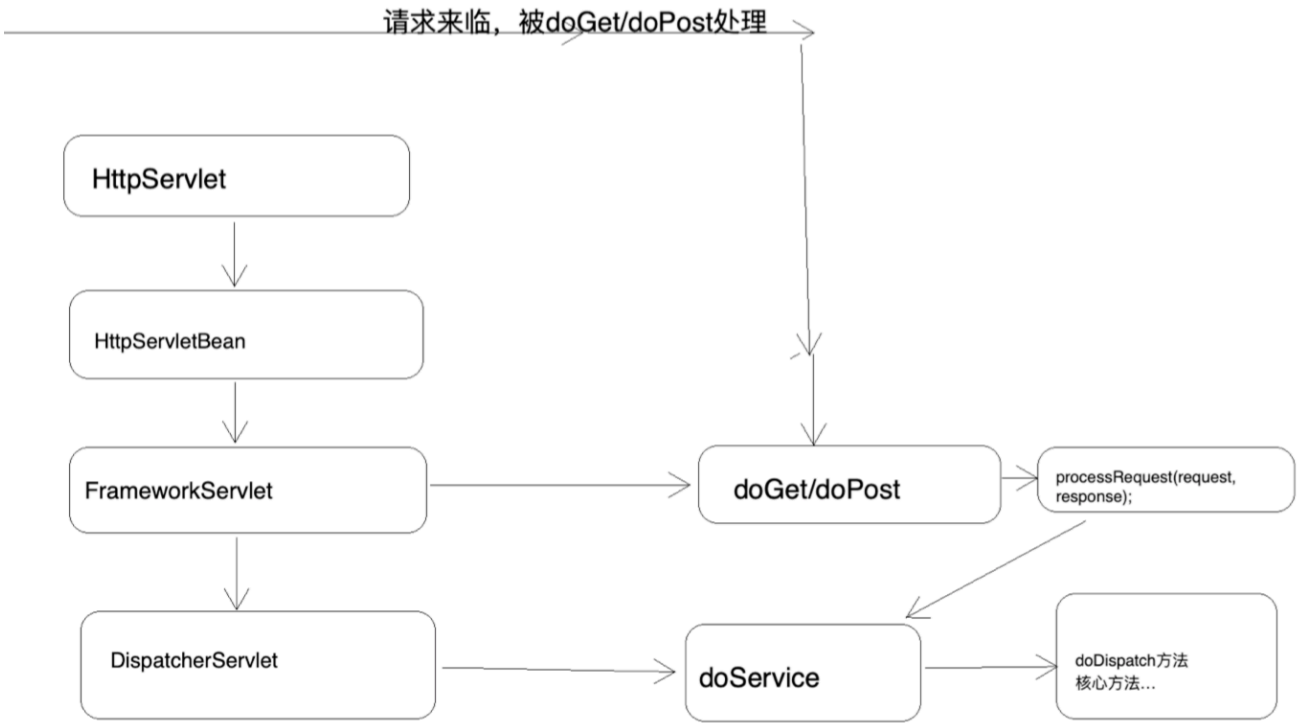
> 注意，tomcat的server会有encoding问题（即server output中的中文会乱码成？），VM options 只有填在上表中才能生效，去文件里面直接加反而不会生效。。。

3. 在 Development tab下填好 artifact（填 .war 而不是 .war (exploded)，不然tomcat会找不到代码里的servlet），注意确认下面的 “Application Context” 是 “/”：



4. 此时要再次确认“Server” tab 下面的 URL 是 “http://localhost:8080/”

1. DispatcherServlet 继承结构



- HttpServlet： 有简单的 doGet, doPost。
- 子类 FrameworkServlet： 真正在Override doGet, doPost 方法，2方法中均包含 processRequest 方法。processRequest 方法中，有个 doService 的 abstract method，这样，具体的怎么do就由其子类 DispatcherServlet 来定义。
- 子类 DispatcherServlet：doService方法 ==> doDispatch方法【核心步骤】

2. 重要时机点分析

放断点：

1. DemoController 的 System.out.println
2. suceess.jsp 的 System.out.println

- URL "/demo/handle01" 会触发：

FrameworkServlet.doGet ==> FrameworkServlet.processRequest ==> DispatcherServlet.doService ==> DispatcherServlet.doDispatch中的 ha.handle 方法：

```
        // 4 实际处理请求，返回结果视图对象
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
```

- 点击首页按钮会触发跳转：

FrameworkServlet.doGet ==> FrameworkServlet.processRequest ==> DispatcherServlet.doService ==> DispatcherServlet.doDispatch中的 processDispatchResult 方法:

```
processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
```

【总结】DispatcherServlet.doDispatch 的核心步骤：

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {

    HandlerExecutionChain mappedHandler = null;
    ...
    try {
      ...
      try {
        // 1 检查是否是文件上传请求
        processedRequest = checkMultipart(request);
        ...

        /*
        * 2 找能处理当前请求的HandlerExecutionChain。
        * HandlerExecutionChain 中封装了Handler 和 Interceptor。
        * */
        mappedHandler = getHandler(processedRequest);

        ...

        // 3 拿到HandlerAdapter（能够正确调用Handler方法的反射工具类）
        HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

        ...

        // 4 HandlerAdapter 实际处理请求，返回结果视图对象
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());


        // 5 结果视图对象的处理
        applyDefaultViewName(processedRequest, mv);

        ...
```

```
      // 6 跳转页面并渲染
      processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
    }
    catch (Exception ex) {
    ...
    }
    finally {
      ...
  }
```

3. getHandler 方法分析

org.springframework.web.servlet.DispatcherServlet#getHandler:

```
@Nullable
  protected HandlerExecutionChain getHandler(HttpServletRequest request) throws
Exception {
    /*
    * handlerMappings 是个list，有2个元素，遍历他们:
    *   [0] = BeanNameUrlHandlerMapping，适用于 xml 配置，早起开发常用
    *   [1] = RequestMappingHandlerMapping，适用于 注解 配置，现在开发常用
    * 映射关系在容器创建时就会创建，这里直接去取即可。
    * */
    if (this.handlerMappings != null) {
      for (HandlerMapping mapping : this.handlerMappings) {
        HandlerExecutionChain handler = mapping.getHandler(request);
        // 根据url “/demo/handle01” 找到了HandlerExecutionChain
        if (handler != null) {
          return handler;
        }
      }
    }
    return null;
  }
```

4. getHandlerAdapter 方法分析

org.springframework.web.servlet.DispatcherServlet#getHandlerAdapter

org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter#supports （return true）

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws
ServletException {
    /*
    * handlerAdapters 是 Interface，有3个实现类，遍历他们，看是否能支持当前handler：
    *   [0] = HttpRequestHandlerAdapter
    *   [1] = SimpleControllerHandlerAdapter
    *   [2] = RequestMappingHandlerAdapter
    * */
    if (this.handlerAdapters != null) {
      for (HandlerAdapter adapter : this.handlerAdapters) {
        if (adapter.supports(handler)) {
          return adapter;
        }
      }
    }
    throw new ServletException("No adapter for handler [" + handler +
        "]: The DispatcherServlet configuration needs to include a HandlerAdapter
that supports this handler");
  }
```

5. ha.handle 方法分析 （匹配参数+invoke目标handler方法）

doDispatch ==>

ha.handle ==>

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter#handleInternal：

```
@Override
  protected ModelAndView handleInternal(HttpServletRequest request,
      HttpServletResponse response, HandlerMethod handlerMethod) throws Exception
{

    ModelAndView mav;
    checkRequest(request);

    // Execute invokeHandlerMethod in synchronized block if required.
```

```
    // 判断当前是否需要支持 在同一个session中只能线性处理请求（因为session本身是线程不安全
的，多个请求过来要考虑如何handle）
    if (this.synchronizeOnSession) {
      HttpSession session = request.getSession(false);
      if (session != null) {
        // 为当前session生成一个唯一的key对象
        Object mutex = WebUtils.getSessionMutex(session);
        // 给该对象加锁
        synchronized (mutex) {
          // 此处才去 invoke handler中的method，继续进行参数等的适配等
          mav = invokeHandlerMethod(request, response, handlerMethod);
        }
      }
      else {
        // No HttpSession available -> no mutex necessary
        mav = invokeHandlerMethod(request, response, handlerMethod);
      }
    }
    else {
      // No synchronization on session demanded at all...
      // 当前若不需要对session进行以上同步处理，就直接invoke
      mav = invokeHandlerMethod(request, response, handlerMethod);
    }
```

==>
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter#invokeHandlerMethod

==>
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod#invokeAndHandle

==> org.springframework.web.method.support.InvocableHandlerMethod#invokeForRequest：

```
@Nullable
public Object invokeForRequest(NativeWebRequest request, @Nullable
ModelAndViewContainer mavContainer,
      Object... providedArgs) throws Exception {

   // Step 1 - 将request中的参数转换为当前handler的参数形式
   Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
   if (logger.isTraceEnabled()) {
      logger.trace("Arguments: " + Arrays.toString(args));
   }
   // Step 2 - 反射调用
   return doInvoke(args);
}
```

其中，Step 1：

org.springframework.web.method.support.InvocableHandlerMethod#getMethodArgumentValues：

针对目标handler method 声明的各个参数：

```
1. 先查后端母方法传来的参数  是否能匹配上
```

2. 若不能，再查 request 中传来的参数，但查之前，要先去确认容器中存在 ArgumentResolver 能够解析当前针对的handler method声明的这个参数。

org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver#resolveArgument

【真正判断request中传来的这个参数是否能匹配上】
org.springframework.web.method.annotation.RequestParamMethodArgumentResolver#resolveName

Step 2：

org.springframework.web.method.support.InvocableHandlerMethod#doInvoke

就用Step 1获得的参数去invoke 目标handler 方法：

```
@Nullable
protected Object doInvoke(Object... args) throws Exception {
   ReflectionUtils.makeAccessible(getBridgedMethod());
   try {
      return getBridgedMethod().invoke(getBean(), args);
   }
```

6. processDispatchResult 方法分析 （跳转+渲染）

org.springframework.web.servlet.DispatcherServlet#processDispatchResult

org.springframework.web.servlet.DispatcherServlet#render

| org.springframework.web.servlet.DispatcherServlet#resolveViewName

| org.springframework.web.servlet.view.AbstractCachingViewResolver#resolveViewName

org.springframework.web.servlet.view.UrlBasedViewResolver#createView

判断是否是重定向、是否是转发等，不同的情况封装的是不同的View的实现。

org.springframework.web.servlet.view.UrlBasedViewResolver#loadView

org.springframework.web.servlet.view.UrlBasedViewResolver#buildView

这里会去拼接 url：

getPrefix(), evaluate expression 后，它= "/WEB-INF/jsp" 即springmvc.xml中设置的前缀

getSuffix(), = ".jsp" 即springmvc.xml中设置的后缀

|

获取到了View之后，再调用View自身的render方法进行渲染：

org.springframework.web.servlet.view.AbstractView#render

org.springframework.web.servlet.view.InternalResourceView#renderMergedOutputModel

把modelmap中的数据暴露到request域中，这也是为什么后台model.add之后，在jsp中也可以从请求域中取出来的原因 【再理解】。

之后获得dispatch 路径 （此处 = "/WEB-INF/jsp/success.jsp"），最终就跳转到这个路径指明的页面。

7. SpringMVC九大组件初始化

7.1 九大组件的定义

九大组件（全是Interface，定义了相应规范）的属性都定义在 org.springframework.web.servlet.DispatcherServlet class 中：

```
/** MultipartResolver used by this servlet. */
  @Nullable
  private MultipartResolver multipartResolver;  // 多部件解析器

  /** LocaleResolver used by this servlet. */
  @Nullable
  private LocaleResolver localeResolver;  // 区域化、国际化解析器

  /** ThemeResolver used by this servlet. */
  @Nullable
  private ThemeResolver themeResolver;  // 主题解析器

  /** List of HandlerMappings used by this servlet. */
  @Nullable
  private List<HandlerMapping> handlerMappings;  // 处理器映射器组件

  /** List of HandlerAdapters used by this servlet. */
  @Nullable
  private List<HandlerAdapter> handlerAdapters;  // 处理器适配器组件

  /** List of HandlerExceptionResolvers used by this servlet. */
  @Nullable
  private List<HandlerExceptionResolver> handlerExceptionResolvers;  // 异常解析器
组件

  /** RequestToViewNameTranslator used by this servlet. */
  @Nullable
  private RequestToViewNameTranslator viewNameTranslator;  // 默认视图名转换器组件
（默认把请求地址当做视图名称）

  /** FlashMapManager used by this servlet. */
  @Nullable
  private FlashMapManager flashMapManager;  // flash属性管理组件

  /** List of ViewResolvers used by this servlet. */
  @Nullable
  private List<ViewResolver> viewResolvers;  // 视图解析器
```

7.2 如何初始化九大组件?

org.springframework.web.servlet.DispatcherServlet#onRefresh

org.springframework.web.servlet.DispatcherServlet#initStrategies

onRefresh的调用栈（谁在用，往下去看）：

FrameworkServlet.

SimpleApplicationEventMulticaster.doInvokeListener （进行一些事件监听）

AbstractApplicationContext.finishRefresh

AbstractApplicationContext.refresh ==> Spring 容器启动的关键方法！

7.3 一些细节

org.springframework.web.servlet.DispatcherServlet#initMultipartResolver：

```
// getBean 是直接从容器中按id "multipartResolver" 去取bean，id是固定值！
this.multipartResolver = context.getBean(MULTIPART_RESOLVER_BEAN_NAME,
MultipartResolver.class);
```

org.springframework.web.servlet.DispatcherServlet#initHandlerMappings：

```
// detectAllHandlerMappings 默认为true，是servlet attribute，可在web.xml中专门设置
if (this.detectAllHandlerMappings) {
   // 找到所有实现了 HandlerMapping Interface 的 Bean
   Map<String, HandlerMapping> matchingBeans =
         BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
HandlerMapping.class, true, false);
   ...
}
else {
   try {
      // 在IoC容器中按照固定id "handlerMapping" 去找 Bean
      HandlerMapping hm = context.getBean(HANDLER_MAPPING_BEAN_NAME,
HandlerMapping.class);
      ...
}

// 以上2分支没找到，就去找默认的HandlerMapping，即读取了默认的
DispatcherServlet.properties文件
if (this.handlerMappings == null) {
   this.handlerMappings = getDefaultStrategies(context, HandlerMapping.class);
   ...
}
```

Part V. SSM整合

1. 整合策略

SSM = Spring + SpringMVC + Mybatis = (Spring + Mybatis) + SpringMVC

先整合 Spring + Mybatis ==> 开发Dao层和Service层

然后再整合 SpringMVC

【需求】

查询Account表中的全部数据 并 显示到页面上。

2. Mybatis整合Spring

2.1 整合目标

【分摊职责】

1. Spring负责：数据库连接池 + 事务管理
2. 把Mybatis中的SqlSessionFactory对象交给Spring容器作为单例对象管理
3. Mybatis中的Mapper动态代理对象交给Spring，直接从Spring容器中获得Mapper的代理对象

【Jar包必备】

1. Junit 测试jar （version 4.12 ）
2. Mybatis jar （version 3.4.5）
3. Spring相关jar （spring-context、spring-test、spring-jdbc、spring-tx、spring-aop、aspectjweaver）
4. 专门粘合Mybatis+Spring的jar （mybatis-spring-xxx.jar）
5. Mysql数据库驱动 jar
6. Druid数据库连接池 jar

2.2 整合实现

2.2.1 准备工作

1. 打开mvc-yu，创建 ssm-yu module （parent= none）
2. 补全 code structure
3. 修改并补全pom.xml

2.2.2 开始

Step 1: Dao层Mapper

Step 2：Service层开发

Step 3：实现整合目标 1.

Step 4：实现整合目标 2.

org.mybatis.spring.SqlSessionFactoryBean 这种 FactoryBean 在与其他框架整合的时候用很多，用来生产特定对象。

Step 5：实现整合目标 3.

注意，在mapper scaner中，要传入sqlSessionFactory的名字，即上面sqlSessionFactory bean的id名（用value而不是ref）：

```
<!--传入sqlSessionFactory的名字，即上面的id名 -->
<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
```

2.3 整合测试

创建测试类：

```
// Spring支持Junit，这里使用Spring专属Junit底层的测试器来驱动
@RunWith(SpringJUnit4ClassRunner.class)
// 不用每次手动new 容器，这里会自动启动容器
@ContextConfiguration(locations = {"classpath*:applicationContext.xml"})
public class MybatisSpringTest {

    // 想测试哪个对象，注入即可
    @Autowired
    private IAccountService accountService;

    @Test
    public void testMybatisSpring() throws Exception {
        List<Account> accounts = accountService.queryAllAccounts();
```

```
        for (int i = 0; i < accounts.size(); i++) {
            System.out.println(accounts.get(i));
        }
    }
}
```

Test result:

Account{name='李大雷', money=19500, cardNo='6029621011000'}
Account{name='韩梅梅', money=20500, cardNo='6029621011001'}

3. 整合SpringMVC

整合思路：在已有工程基础上，开发一个SpringMVC入门案例。

Step 1: pom.xml 中添加所有和SpringMVC相关的dependency。

Step 2: 复制applicationContext.xml为springmvc.xml。springmvc.xml中 能扫描所有controller 并 添加注解驱动。

Step 3: web.xml 中配置servlet + servlet-mapping + 监听器（用于加载Spring）

Step 4: 创建Controller类

Step 5： 为了让配置文件各司其职、便于维护，故拆分配置文件applicationContext.xml 为 applicationContext-dao.xml 和 applicationContext-service.xml。然后修改web.xml中的监听器的class为 applicationContext*.xml

【注意】

在web.xml中，启动servlet的同时，DispatcherServlet会加载springmvc.xml文件，从而启动了SpringMVC。如果还要启动Spring，则要添加一个监听器和一个全局 把Spring的配置文件applicationContext.xml加载进来从而启动Spring，Spring被启动了，其中由于包含关于mybatis的配置，那么mybatis也会被启动。

Spring容器 vs. SpringMVC容器

Spring容器管理Service层对象 + Dao层对象。所以applicationContext.xml中只用扫这两层中的对象即可，不用去扫整个com.lagou.edu pkg。

SpringMVC容器管理Controller对象，这些对象能够引用Spring容器中的对象。所以Controller的扫描只能发生在springMVC的配置文件中。

Test result:

URL = http://localhost:8080/account/queryAll


```
[
{
name: "李大雷",
money: 19500,
cardNo: "6029621011000"
},
{
name: "韩梅梅",
money: 20500,
cardNo: "6029621011001"
}
]
```


随堂练习

Redirect vs. Forward

Refer to - https://www.baeldung.com/servlet-redirect-forward

简言之，forward能够保留请求域参数继续传，redirect不能保留请求域参数值传给后续。

Forward:

- The request will be further processed on the server side
- The client isn't impacted by forward, URL in a browser stays the same
- Request and response objects will remain the same object after forwarding. Request-scope objects will be still available

Redirect:

- The request is redirected to a different resource
- The client will see the URL change after the redirect
- A new request is created, without original request-scope objects unless manual assignments.
- Redirect is normally used within Post/Redirect/Get web development pattern