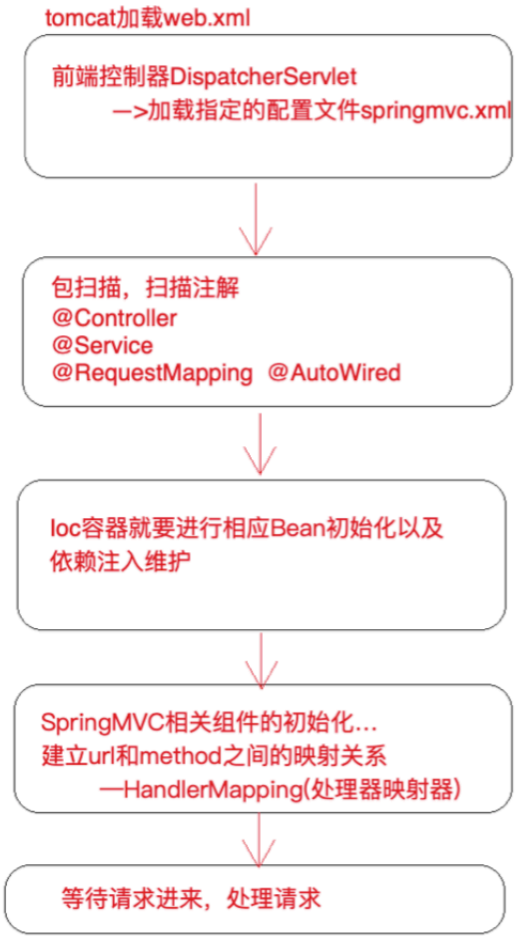
Part III. 手写MVC框架

github commit 详细记录整个过程：

1. 回顾SpringMVC原理



2. 注解

2.1 Preparation

- set up code base
- Create YuDispatcherServlet class （extends HttpServlet）and register it

2.2 Create Annotations pkg

@Documented -- 可以被编辑文档读取

@Retention(RetentionPolicy.RUNTIME) // JVM加载到内存中之后也能生效

3. 流程结构 - servlet.init

3.1 添加springmvc配置文件并在web.xml中注册

3.2 在YuDispatcherServlet class的init方法中，写出整个流程的框架：

```
public class YuDispatcherServlet extends HttpServlet {

    @Override
    public void init(ServletConfig config) throws ServletException {
        // Step 1: load configuration files: springmvc.properties
        // 需在web.xml中<servlet><init-param>中传入该配置文件
        String contextConfigLocation =
config.getInitParameter("contextConfigLocation");
        doLoadConfig(contextConfigLocation);

        // Step 2: scan classes and annotations
        doScan("");

        // Step 3: beans initialization (实现IoC容器, 基于注解)
        doInstance();


        // Step 4: 实现依赖注入
        doAutowired();

        // Step 5: 构造一个HandlerMapping, 将配置好的url和Method建立映射关系
        initHandlerMapping();

        System.out.println("yumvc 初始化完成....");

        // Step 6: 等待请求进入, 处理请求
    }
```

4. doLoadConfig 加载配置文件

先搞成流，再load到Properties中：

```
private Properties properties = new Properties();
...

private void doLoadConfig(String contextConfigLocation) {
    InputStream resourceAsStream =
this.getClass().getClassLoader().getResourceAsStream(contextConfigLocation);

    try {
        properties.load(resourceAsStream);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

5. doScan 扫描类

首先找到pkg 的 diskPath，然后递归扫包，并用一个list来存所有single java classes的名字：

```
// Store all single java classes' names
private List<String> classNames = new LinkedList<>();
...

private void doScan(String scanPackage) {
    // scanPackage = com.lagou.demo --> 包的磁盘路径，即转换成文件夹模式 com/lagou/demo
    String diskPath =
Thread.currentThread().getContextClassLoader().getResource("").getPath()
            + scanPackage.replaceAll("\\.", "/");
    File folder = new File(diskPath);

    File[] files = folder.listFiles();
    for (File file : files) {
        if (file.isDirectory()) {  // sub package
            doScan(scanPackage + "." + file.getName());  //
com.lagou.demo.controller
        }else if (file.getName().endsWith(".class")) {  // a single java class
            String className = scanPackage + "." +
file.getName().replaceAll(".class", "");
            classNames.add(className);
```

```
        }
    }
}
```

6. doInstance 实例化

Scan list of classnames to create id and object into Ioc hashmap. Service的Interface也放一份进入ioc中:

```
// 基于classNames中 类的全限定类名，以及反射技术，完成对象创建和管理
    private void doInstance() {
        if (classNames.size() == 0) return;

        try {
            for (int i = 0; i < classNames.size(); i++) {
                String className = classNames.get(i);  //
com.lagou.demo.controller.DemoController

                // Reflection
                Class<?> aClass = Class.forName(className);

                // 区分 Controller 和 Service 来对id进行处理
                if (aClass.isAnnotationPresent(YuController.class)) {
                    Object o = aClass.newInstance();

                    // controller的id此处不做过多处理，不取value了，就拿类的首字母小写作为
id，保存到ioc中
                    String simpleName = aClass.getSimpleName();  //
DemoController
                    String id = lowerFirstLetter(simpleName);  // id =
demoController

                    ioc.put(id, o);
                } else if (aClass.isAnnotationPresent(YuService.class)) {
                    Object o = aClass.newInstance();

                    YuService annotation = aClass.getAnnotation(YuService.class);
                    String beanName = annotation.value();

                    if (!beanName.trim().isEmpty()) {  // has value, then id =
value
                        ioc.put(beanName, o);
                    }else {
```

```
                    beanName = lowerFirstLetter(aClass.getSimpleName());
                    ioc.put(beanName, o);
                }

                /*
                * Service 层往往是有Interface的（面向Interface开发），此时再以
Interface的全类名为id，存入ioc中，
                * 便于之后根据Interface 类型注入
                * */
                Class<?>[] interfaces = aClass.getInterfaces();
                for (int j = 0; j < interfaces.length; j++) {
                    Class<?> singleInterface = interfaces[j];
                    ioc.put(singleInterface.getName(), o);
                }
            } else {
                continue;
            }

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

7. doAutowired 依赖注入

```
private void doAutowired() {
    if (ioc.isEmpty()) return;

    // 遍历ioc中所有对象，查看对象中的属性是否有autowire注解，若有才维护依赖注入关系
    for (Map.Entry<String, Object> entry : ioc.entrySet()) {
        // 获取当前bean object的所有属性
        Field[] attributes = entry.getValue().getClass().getDeclaredFields();

        for (int i = 0; i < attributes.length; i++) {
            Field attribute = attributes[i];
            if (!attribute.isAnnotationPresent(YuAutowired.class)) {
                continue;
            }

            // 找到了有Autowire注解的attribute
            YuAutowired annotation =
attribute.getAnnotation(YuAutowired.class);
```

```
                String id = annotation.value();
                if (id.trim().isEmpty()) {   // 若没有指定value，则根据attribute的
Interface注入
                    id = attribute.getType().getName();  // IDemoService
                }
                attribute.setAccessible(true);
                try {
                    /*
                     *  field.set(obj, value)
                     *  obj = the object whose field should be modified
                     *  value = the new value for the field of obj
                     * */
                    //                  parentObj ,  childObj
                    attribute.set(entry.getValue(), ioc.get(id));
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

8. initHandlerMapping 【关键】

目的：将url和method建立关联

```
private void initHandlerMapping() {
    if (ioc.isEmpty()) return;

    // 扫描Controller class 和 class中的方法来获得注解中的url
    for (Map.Entry<String, Object> entry : ioc.entrySet()) {
        Class<?> aClass = entry.getValue().getClass();
        if (!aClass.isAnnotationPresent(YuController.class)) {
            continue;
        }

        String baseUrl = "";
        if (aClass.isAnnotationPresent(YuRequestMapping.class)) {
            YuRequestMapping annotation =
aClass.getAnnotation(YuRequestMapping.class);
            baseUrl = annotation.value();   // 获得 "/demo"
        }

        // 遍历Object中的有RequestMapping注解的方法找剩下的url
```

```
        Method[] methods = aClass.getMethods();
        for (int i = 0; i < methods.length; i++) {
            Method method = methods[i];
            if (!method.isAnnotationPresent(YuRequestMapping.class)) {
                continue;
            }

            YuRequestMapping annotation =
method.getAnnotation(YuRequestMapping.class);
            String methodUrl = annotation.value();  // 获得"/query"
            String url = baseUrl + methodUrl;

            // 储存url和method之间的关系
            handlerMapping.put(url, method);
        }
    }
}
```

9. 测试初始化流程

1. 在web.xml中，去掉 `<param-value>classpath*:springmvc.properties</param-value>` 中的 `classpath*:`, 因为 doScan 的input是可以直接从".properties" 文件取的，不需要有完整路径

   ```
   // Step 2: scan classes and annotations
   doScan(properties.getProperty("scanPackage"));
   ```

2. 在pom.xml中，为 javax.servlet-api 添加 provided scope，表明一旦部署到服务器后，就不再使用 javax.servlet-api，而是用tomcat的servlet-api，如果这里不指名scope，会产生冲突从而会报错：

   ```
   <dependency>
     <groupId>javax.servlet</groupId>
     <artifactId>javax.servlet-api</artifactId>
     <version>3.1.0</version>
     <scope>provided</scope>
   </dependency>
   ```

Test:

能正常启动tomcat Server，并能在后台console中打印"yumvc 初始化完成...."

10. handlerMapping 问题分析

在 doPost() 中，为了反射调用方法（即用method.invoke()），需要传入对象、传入参数，故需改造 initHandlerMapping()。

11. Handler封装引入

创建Handler class，其包含4个属性：

```
/*
* 封装Handler方法的各种相关信息
* */
public class Handler {

    private Object controller;  // 即方法所属于的那个类的 bean对象

    private Method method;

    private Pattern pattern;  // spring 中 url是支持正则的

    private Map<String, Integer> paramIndexMapping;  // 参数名:第几个参数（从0开始），方便进行参数绑定

    public Handler(Object controller, Method method, Pattern pattern) {
        this.controller = controller;
        this.method = method;
        this.pattern = pattern;
        this.paramIndexMapping = new HashMap<>();
    }
  // getters and setters
  ...
```

这样，HandlerMapping中装的将不再是 id:Method ， 而是 id:Handler

12. initHandlerMapping 改造

不再需要hashmap，List 足够。

initHandlerMapping 方法中，在获得了最终url后：

```
...
String methodUrl = annotation.value();  // 获得"/query"
String url = baseUrl + methodUrl;   // "/demo/query"

// 封装bean对象，method，url进Handler
Handler handler = new Handler(entry.getValue(), method, Pattern.compile(url));

/*
* 最后，计算当前method的参数位置信息
* 例如 query(HttpServletRequest request, HttpServletResponse response, String
name)
* */
Parameter[] parameters = method.getParameters();
for (int j = 0; j < parameters.length; j++) {
    Parameter parameter = parameters[j];

    // 这里只判断参数类型为 HttpServletRequest 或 HttpServletResponse 或 其他普通类型
    if (parameter.getType() == HttpServletRequest.class
    || parameter.getType() == HttpServletResponse.class) {
        // simpleName 即 HttpServletResponse 或者 HttpServletRequest
        handler.getParamIndexMapping().put(parameter.getType().getSimpleName(),
j);
    } else {
        handler.getParamIndexMapping().put(parameter.getName(), j);  // 即put了
<name, 2>
    }
}

handlerMapping.add(handler);
```

13. 请求处理

```
// 根据url，找到对应的Handler并调用其中的Method
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    // Step 1: 从HandlerMapping中查是否有能匹配上当前 request 里面 uri 的Handler
```

```
    Handler handler = getHandler(req);
    if (handler == null) {
        resp.getWriter().write("404 not found");
        return;
    }

    /*
    * Step 2: 参数绑定:
    *   1. 看method需要哪些参数
    *   2. 创建新array, named paramValues, 准备向其中塞method所需要的参数值
    *   3. 塞值到相应index:
          3.1 遍历request中的所有参数, 这里先只处理普通参数
    *     3.2 再直接处理HttpServletRequest 和 HttpServletResponse
    * */

    Class<?>[] methodParams = handler.getMethod().getParameterTypes();
    int len = methodParams.length;
    Object[] paramValues = new Object[len];

    Map<String, String[]> requestParamMap = req.getParameterMap();  // 同一个参数名
可能是个array,
    for (Map.Entry<String, String[]> entry : requestParamMap.entrySet()) {
        // name=Lisa&name=Jenny --> <name, [Lisa,Jenny]>
        String value = StringUtils.join(entry.getValue(), ",");  // value =
"Lisa,Jenny"

        // 如果参数和方法中的参数匹配上了, 就继续填充参数值进新array
        if (!handler.getParamIndexMapping().containsKey(entry.getKey())) {
            continue;
        }
        String paramName = entry.getKey();  // paramName = "name"
        Integer index = handler.getParamIndexMapping().get(paramName);  // "name"
对应于index为2的位置
        paramValues[index] = value;
    }

    // 处理 HttpServletRequest 和 HttpServletResponse
    int requestIndex =
handler.getParamIndexMapping().get(HttpServletRequest.class.getSimpleName());  //
= 0
    if (0 <= requestIndex && requestIndex < len) {
        paramValues[requestIndex] = req;
    }
    int responseIndex =
handler.getParamIndexMapping().get(HttpServletResponse.class.getSimpleName()); //
= 1
```

```
    if (0 <= responseIndex && responseIndex < len) {
        paramValues[responseIndex] = resp;
    }

    // Step 3: 最终调用Handler中的Method （传入手动塞好的paramValues 作为method的参数）
    try {
        handler.getMethod().invoke(handler.getController(), paramValues);
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}

private Handler getHandler(HttpServletRequest req) {
    if (handlerMapping.isEmpty()) {
        return null;
    }

    String url = req.getRequestURI();  // 拿uri，而不是url

    for (Handler handler : handlerMapping) {
        Matcher matcher = handler.getPattern().matcher(url);
        if (!matcher.matches()) {   // 如果url不匹配Handler的pattern
            continue;
        }
        return handler;
    }
    return null;
}
```

14. 整体测试

- URL = http://localhost:8080/demo/query?name=Jenny
- Test Result：后台打印出name的值是null -- “Service 实现类中的name参数： null”
- Root Cause：在compile时，编译器不会去获取参数名称本身，而是会将用 arg0, arg1, arg2 来替代参数 req, resp, name
- Solution: 在pom.xml中，引入编译插件定义编译细节，即让编译器不要把参数转换成arg0,arg1等，而是直接拿参数名称本身：

```
<!-- 为了让编译器在编译参数时能够获得参数名本身，而不是用arg0,arg1等去替代
（因为存参数名本身也需耗费资源） -->
```

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
      <source>11</source>
      <target>11</target>
      <encoding>utf-8</encoding>
      <!--  告诉编译器，在编译时记录下参数真实名称  -->
      <compilerArgs>
        <arg>-parameters</arg>
      </compilerArgs>
    </configuration>
  </plugin>
```

- 再次 Test， URL = http://localhost:8080/demo/query?name=Jenny
  - Result：“Service 实现类中的name参数： Jenny” 可见后台能够获得参数name的值