1、自己定义拦截类

自己定义拦截类，需要继承HandlerIntercepor接口并实现这个接口的方法

```
if (StringUtils.isEmpty(token)) {
    if (logger.isDebugEnabled()) {
        logger.debug("no login");
    }
    response(response, Constant.ERROR_CODE_NO_LOGIN, Constant.ERROR_MSG_NO_LOGIN);
    return false;
```

preHandle方法：预处理回调方法，实现处理器的预处理（如登录检查），第三个参数为响应的处理器；返回true，映射处理器执行链将继续执行；当返回false时，DispatcherServlet处理器认为拦截器已经处理完了请求，而不继续执行执行链中的其它拦截器和处理器。

false表示流程中断（如登录检查失败），不会继续调用其他的拦截器或处理器，此时我们需要通过response来产生响应；

postHandle****：后处理回调方法，实现处理器的后处理（但在渲染视图之前），此时我们可以通过modelAndView（模型和视图对象）对模型数据进行处理或对视图进行处理，modelAndView也可能为null。

afterCompletion****：**整个请求处理完毕回调方法，即在视图渲染完毕时回调，如性能监控中我们可以在此记录结束时间并输出消耗时间，还可以进行一些资源清理，类似于try-catch-finally中的finally，但仅调用处理器执行链中preHandle返回true的拦截器的afterCompletion。**

```
@Override
    public boolean preHandle(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Object o) throws Exception {
        //方法调用前执行
        return true;//返回为false,拦截器拦截的方法不会调用
    }
    @Override
    public void postHandle(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Object o, ModelAndView modelAndView)
throws Exception {
    //方法执行结束后执行
    }
```

```
@Override
public void afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) throws Exception {
//该方法将在整个请求完成之后，也就是DispatcherServlet渲染了视图执行，这个方法的主要作用是用于清理资源的，
}
```

2、定义配置类（继承WebMvcConfigurerAdapter）

```
@Bean
public LoginInterceptor loginInterceptor() {
    return new LoginInterceptor();
}

@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(loginInterceptor()).addPathPatterns("/**").excludePathPatterns("/public/**");;
}
```

多个拦截器组成一个拦截器链

addPathPatterns 用于添加拦截规则

excludePathPatterns 用户排除拦截

在Adapter中

（1）方式一

由于没有web.xml这个配置文件，所以我们需要继承WebMvcConfigurerAdapter，来替代web..xml配置文件，并在类上加上注解@Configuration

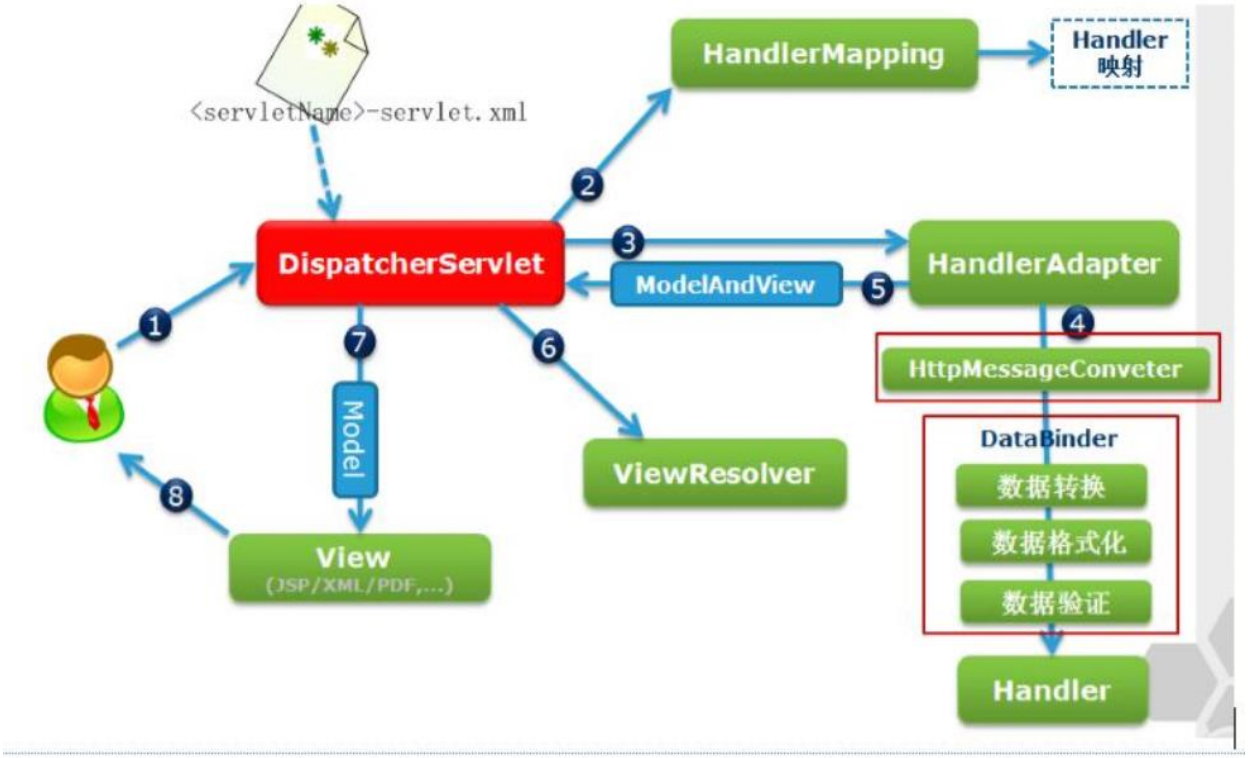
重写addInterceptors方法，加入自定义拦截器，就可以执行自己的代码。

```java
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter{

    /**
     * 注册 拦截器
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new HandlerInterceptorAdapter() {
            @Override
            public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
                    throws Exception {
                System.out.println("222222222222 WebMvcConfigurer.interceptor====");
                return true;
            }
        }).addPathPatterns("/*");
    }

}
```

SpringMvc运行原理：



一旦Http请求到来，DispatcherSevlet将负责将请求分发。DispatcherServlet可以认为是Spring提供的前端控制器，所有的请求都有经过它来统一分发。

在DispatcherServlet将请求分发给Spring Controller之前，需要借助于Spring提供的HandlerMapping定位到具体的Controller。HandlerMapping是这样一种对象，它能够完成客户请求到Controller之间的映射。在Struts中，这种映射是通过struts-config.xml文件完成的。其中，Spring为Controller接口提供了若干实现，例如Spring默认使用的BeanNameUrlHandlerMapping。还有，SimpleUrlHandlerMapping，CommonsPathMapHandlerMapping。

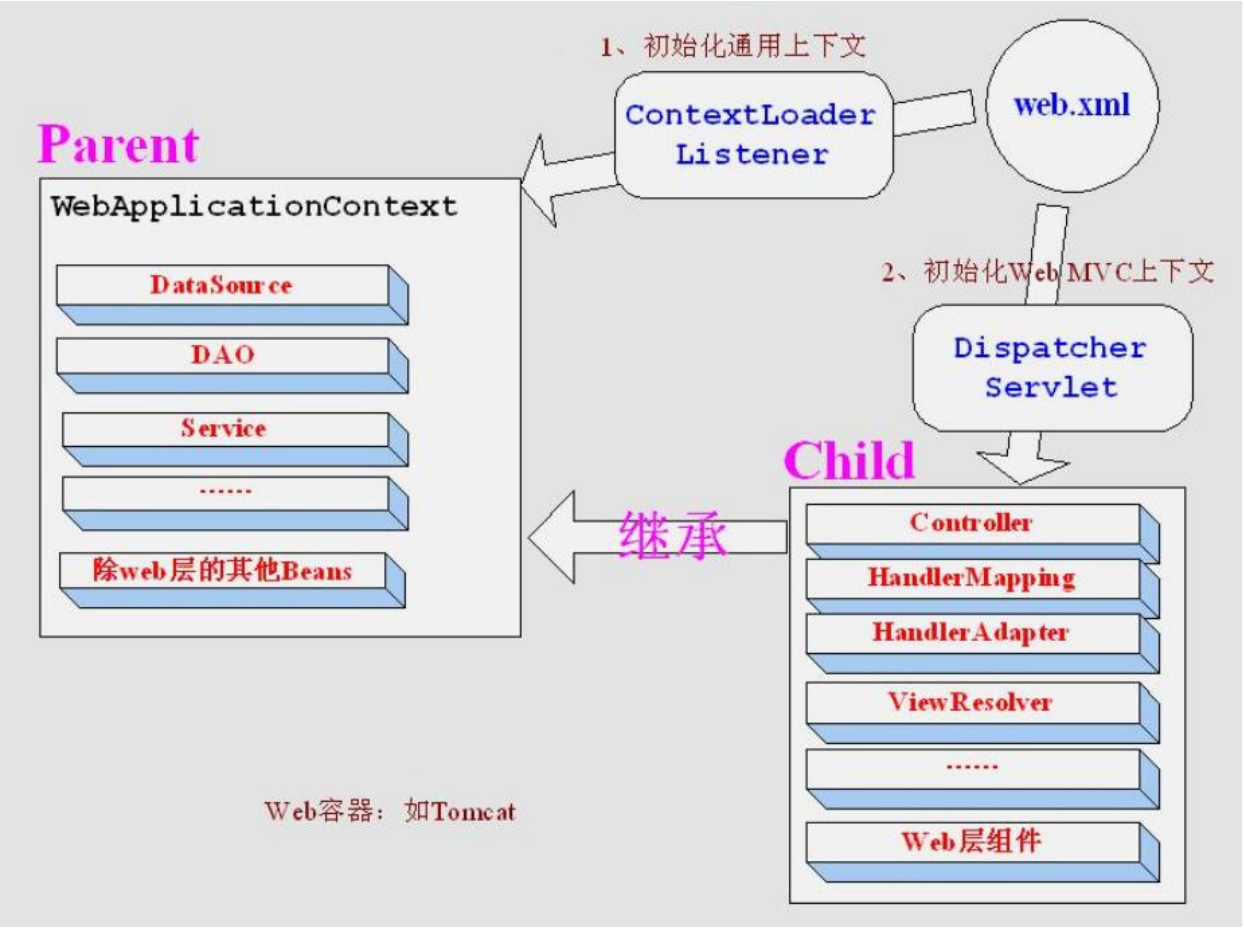
Spring Controller将处理来自DispatcherServlet的请求。Spring的Controller类似于struts的Action，能够接受HttpServletRequest和HttpServletResponse。Spring为Controller接口提供了若干实现类，位于org.springframework.web.servlet.mvc包中。由于Controller需要为并发用户处理上述请求，因此实现Controller接口时，必须保证线程安全并且可重用。Controller将处理客户请求，这和Struts Action 扮演的角色是一致的。

一旦Controller处理完客户请求，则返回ModelAndView对象给DispatcherServlet前端控制器。ModelAndView中包含了模型（Model）和视图（View）。从宏观角度考虑，DispatcherServlet是整个Web应用的控制器；从微观角度考虑，Controller是单个Http请求处理过程中的控制器，而ModelAndView是Http请求过程中返回的模型和视图。前端

控制器返回的视图可以是视图的逻辑名，或者实现了View接口的对象。View对象能够渲染客户响应结果。其中，ModelAndView中的模型能够供渲染View时使用。借助于Map对象能够存储模型。
如果ModelAndView返回的视图只是逻辑名，则需要借助Spring提供的视图解析器（ViewResoler）在Web应用中查找View对象，从而将响应结果渲染给客户。DispatcherServlet将View对象渲染出的结果返回个客户。

二:
（1）用户发送请求，经过前端控制器DispatcherServlet，DispatcherServlet将URL交给处理器映射器HandlerMapping处理HandlerMapping 有点类似于Struts2中的ActionMapper.
（2）HandlerMapping接收到URL，处理完URL，返回了HandlerExecutionChain，HandlerExecutionChain里面包含了当前URL对应的Handler的所有信息，包括Handler本身以及当前Handler对象对应的拦截器对象
（3）HandlerMapping处理返回HandlerExecutionChain之后，DispatcherServlet会根据当前的Handler请求对应的HandlerAdapter。如果当前的Handler有拦截器的话，会先调用拦截器的preHandler方法，然后会调用Handle方法，而HandlerAdapter会调用我们的Handler对象中的执行方法（类似于Action里面的Execute方法）
（4）Handler对象执行方法结束后，返回ModelAndView
（5）ModelAndView会被视图解析器（ViewResolver）解析，然后返回到DispatcherServlet，最后DispatcherServlet将对应的视图返回给客户端扩展

讲解DispatcherServlet:



ContextLoaderListener初始化的上下文加载的Bean是对于整个应用程序共享的，不管是使用什么表现层技术，一般如DAO层、Service层Bean.
DispatcherServlet初始化的上下文加载的Bean是只对Spring Web Mvc 有效的Bean、如Controller、HandlerMapping、HandlerAdapter等等，该初始化上下文应该只加载Web相关组件。

DispatcherServlet继承FrameworkServlet，并实现了onRefresh()方法提供一些前端控制器相关的配置：

```java
public class DispatcherServlet extends FrameworkServlet {
     //实现子类的onRefresh()方法，该方法委托为initStrategies()方法。
    @Override
    protected void onRefresh(ApplicationContext context) {
        initStrategies(context);
    }

    //初始化默认的Spring Web MVC框架使用的策略（如HandlerMapping）
    protected void initStrategies(ApplicationContext context) {
        initMultipartResolver(context);
        initLocaleResolver(context);
        initThemeResolver(context);
        initHandlerMappings(context);
        initHandlerAdapters(context);
        initHandlerExceptionResolvers(context);
        initRequestToViewNameTranslator(context);
        initViewResolvers(context);
        initFlashMapManager(context);
    }
}
```

从如上代码可以看出，DispatcherServlet启动时会进行我们需要的Web层Bean的配置，如HandlerMapping、HandlerAdapter等，而且如果我们没有配置，还会给我们提供默认的配置。

从如上代码我们可以看出，整个DispatcherServlet初始化的过程和做了些什么事情，具体主要做了如下两件事情：

- 初始化Spring Web MVC使用的Web上下文，并且可能指定父容器为（ContextLoaderListener加载了根上下文）；
- 初始化DispatcherServlet使用的策略，如HandlerMapping、HandlerAdapter等。