有道云链接：http://note.youdao.com/noteshare?id=a9ea60905defbf27166e0baf8f5840de&sub=C4A9D5873E364EAB9C6C2E8DDF41CF16

1、Spring整合SpringMVC

2、零配置SpringMVC实现方式：

3、实现基于SPI规范的SpringMVC

4、SPI的方式SpringMVC启动原理

流程图：

源码流程

创建父容器——ContextLoaderListener

创建子容器——DispatcherServlet

4. 初始化ContextLoaderListener

5. 初始化DispatcherServlet

总结

用几道面试题做个总结:

1、Spring整合SpringMVC

特性：

说到Spring整合SpringMVC唯一的体现就是父子容器：

- 通常我们会设置父容器（Spring）管理Service、Dao层的Bean, 子容器(SpringMVC)管理Controller的Bean .
- 子容器可以访问父容器的Bean, 父容器无法访问子容器的Bean。

实现：

相信大家在SSM框架整合的时候都曾在web.xml配置过这段：

```
<!--spring 基于web应用的启动-->
<listener>
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<!--全局参数：spring配置文件-->
<context-param>
```

```
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:spring-core.xml</param-value>
 </context-param>
 <!--前端调度器servlet-->
 <servlet>
  <servlet-name>dispatcherServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <!--设置配置文件的路径-->
  <init-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:spring-mvc.xml</param-value>
 </init-param>
  <!--设置启动即加载-->
  <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
  <servlet-name>dispatcherServlet</servlet-name>
  <url-pattern>/</url-pattern>
 </servlet-mapping>
```

但是它的作用是什么知道吗?

有人可能只知道DispatcherServlet叫前端控制器，是SpringMVC处理前端请求的一个核心调度器

那它为什么能处理请求？处理之前做了什么准备工作呢？又是怎么和Spring结合起来的呢？

为什么有了DispatcherServlet还要个ContextLoaderListener， 配一个不行吗？干嘛要配俩啊?

看完本文你就会有答案!

还有人可能会觉得， 我现在都用SpringBoot开发， 哪还要配这玩意.....

这就是典型的SpringBoot使用后遗症，SpringBoot降低了使用难度，但是从某种程度来说，也让初级的程序员变得更加小白，把实现原理都隐藏起来了而我们只管用，一旦涉及扩展就束手无策。

那当然我们今天不讲SpringBoot,我们今天用贴近SpringBoot的方式来讲SpringMVC。

也就是**零配置（零xml）的放式**来说明SpringMVC的原理！！

> 此方式作为我们本文重点介绍，也是很多人缺失的一种方式， 其实早在Spring3+就已经提供， 只不过我们直到SpringBoot才使用该方式进行自动配置， 这也是很多人从xml调到SpringBoot不适应的原因， 因为你缺失了这个版本。 所以我们以这种方式作为源码切入点既可以理解到XML的方式又能兼顾到SpringBoot的方式 。

2、零配置SpringMVC实现方式：

那没有配置就需要省略掉web.xml 怎么省略呢?

在Servlet3.0提供的规范文档中可以找到2种方式：

1. 注解的方式

a. @WebServlet

b. @WebFilter

c. @WebListener

但是这种方式不利于扩展， 并且如果编写在jar包中tomcat是无法感知到的。

2. SPI的方式

在Serlvet3-1的规范手册中：就提供了一种更加易于扩展可用于共享库可插拔的一种方式，参见8.2.4：

8.2.4 Shared libraries / runtimes pluggability

In addition to supporting fragments and use of annotations one of the requirements is that not only we be able to plug-in things that are bundled in the `WEB-INF/lib` but also plugin shared copies of frameworks - including being able to plug-in to the web container things like JAX-WS, JAX-RS and JSF that build on top of the web container. The `ServletContainerInitializer` allows handling such a use case as described below.

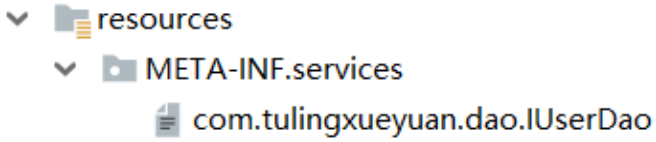
The `ServletContainerInitializer` class is looked up via the jar services API. For each application, an instance of the `ServletContainerInitializer` is created by the container at application startup time. The framework providing an implementation of the `ServletContainerInitializer` MUST bundle in the `META-INF/services` directory of the jar file a file called `javax.servlet.ServletContainerInitializer`, as per the jar services API, that points to the implementation class of the `ServletContainerInitializer`.

也就是让你在应用META-INF/services 路径下 放一个
javax.servlet.ServletContainerInitailizer ——即SPI规范
啥？？ 啥是SPI??

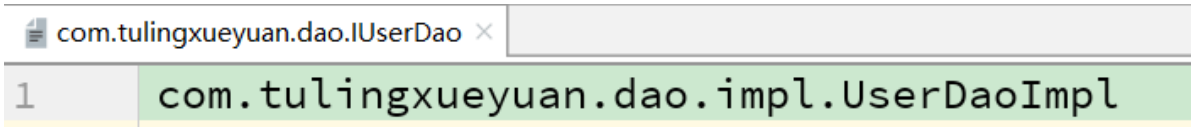
SPI 我们叫他服务接口扩展,(Service Provider Interface) 直译服务提供商接口， 不要被这个名字唬到了， 其实很好理解的一个东西:
其实就是根据Servlet厂商（服务提供商）提供要求的一个接口， 在固定的目录（META-INF/services）放上以接口全类名 为命名的文件， 文件中放入接口的实现的全类名，该类由我们自己实现，按照这种约定的方式（即SPI规范），服务提供商会调用文件中实现类的方法， 从而完成扩展。

SPI演示案例:
假设我们自己是服务提供商： 现在要求的一个接口 IUserDao

1.在固定的目录放上接口的文件名



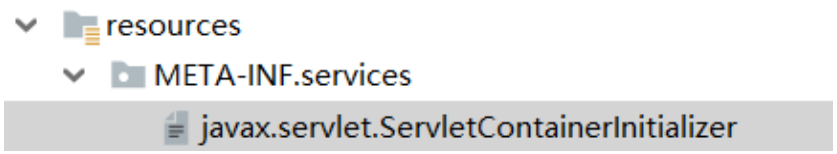
2.文件中放入实现类（该实现类由你实现）：
一行一个实现类。



3.通过java.util.ServiceLoader提供的ServiceLoader就可以完成SPI的实现类加载

```
1 public class App {
2  public static void main(String[] args) {
3  ServiceLoader<IUserDao> daos = ServiceLoader.load(IUserDao.class);
4  for (IUserDao dao : daos) {
5  dao.save();
6  }
7  }
8 }
```

ok 那我们知道了SPI是什么，我们是不是可以在Web应用中，在Servlet的SPI放入对应的接口文件：



放入实现类：

spring-tuling-webmvc\...\javax.servlet.ServletContainerInitializer

```
1  com.tuling.javaweb.spi.TulingSpringServletContainerInitializer
```

通过ServletContext就可以动态注册三大组件：以Servlet注册为例：

```
1
2  public class TulingSpringServletContainerInitializer extends SpringServletCon
tainerInitializer {
3
4   @Override
5   public void onStartup(Set<Class<?>> webAppInitializerClasses, ServletContext
servletContext) throws ServletException {
6
7   // 通过servletContext动态添加Servlet
8   servletContext.addServlet("spiServlet", new HttpServlet() {
9   @Override
10   protected void doGet(HttpServletRequest req, HttpServletResponse resp) thro
ws ServletException, IOException {
11   resp.getWriter().write("spiServlet--doGet");
12   }
13   }).addMapping("/spiServlet.do");
14
15
16   }
17  }
```

当然在SpringMVC中， 这个接口文件和实现类都把我们实现好了，甚至ContextLoaderListener和DispatcherServlet都帮我们注册好了，我们只要让他生效，来，看看他是怎么做的：

3、实现基于SPI规范的SpringMVC

TulingStarterInitializer

- 此类继承AbstractAnnotationConfigDispatcherServletInitializer 这是个啥？ 待会我们讲原理来介绍
- getRootConfigClasses 提供父容器的配置类
- getServletConfigClasses 提供子容器的配置类
- getServletMappings 设置DispatcherServlet的映射

```
1
2 public class TulingStarterInitializer extends AbstractAnnotationConfigDispatc
herServletInitializer {
3
4  /**
5  * 方法实现说明:IOC 父容器的启动类
6  * @author:xsls
7  * @date:2019/7/31 22:12
8  */
9  @Override
10  protected Class<?>[] getRootConfigClasses() {
11  return new Class[]{RootConfig.class};
12  }
13
14  /**
15  * 方法实现说明 IOC子容器配置 web容器配置
16  * @author:xsls
17  * @date:2019/7/31 22:12
18  */
19  @Override
20  protected Class<?>[] getServletConfigClasses() {
21  return new Class[]{WebAppConfig.class};
22  }
23
24  /**
25  * 方法实现说明
26  * @author:xsls
27  * @return: 我们前端控制器DispatcherServlet的拦截路径
28  * @exception:
29  * @date:2019/7/31 22:16
30  */
31  @Override
```

```
32  protected String[] getServletMappings() {
33  return new String[]{"/"};
34  }
```

RootConfig

- 父容器的配置类 =以前的spring.xml
- 扫描的包排除掉@Controller

```
1 @Configuration
2 @ComponentScan(basePackages = "com.tuling",excludeFilters = {
3  @ComponentScan.Filter(type = FilterType.ANNOTATION,value={RestController.cla
ss,Controller.class}),
4  @ComponentScan.Filter(type = ASSIGNABLE_TYPE,value =WebAppConfig.class ),
5 })
6 public class RootConfig {
7 }
```

WebAppConfig

- 子容器的配置类 =以前的spring-mvc.xml
- 扫描的包：包含掉@Controller

```
1
2 @Configuration
3 @ComponentScan(basePackages = {"com.tuling"},includeFilters = {
4  @ComponentScan.Filter(type = FilterType.ANNOTATION,value = {RestController.c
lass, Controller.class})
5 },useDefaultFilters =false)
6 @EnableWebMvc // ≈<mvc:annotation-driven/>
7 public class WebAppConfig implements WebMvcConfigurer{
8
9  /**
10  * 配置拦截器
11  * @return
12  */
13  @Bean
14  public TulingInterceptor tulingInterceptor() {
15  return new TulingInterceptor();
16  }
17
18  /**
19  * 文件上传下载的组件
20  * @return
21  */
22  @Bean
```

```
23   public MultipartResolver multipartResolver() {
24   CommonsMultipartResolver multipartResolver = new
CommonsMultipartResolver();
25   multipartResolver.setDefaultEncoding("UTF-8");
26   multipartResolver.setMaxUploadSize(1024*1024*10);
27   return multipartResolver;
28   }
29
30   /**
31   * 注册处理国际化资源的组件
32   * @return
33   */
34  /* @Bean
35   public AcceptHeaderLocaleResolver localeResolver() {
36   AcceptHeaderLocaleResolver acceptHeaderLocaleResolver = new AcceptHeaderLoc
aleResolver();
37   return acceptHeaderLocaleResolver;
38   }*/
39
40   @Override
41   public void addInterceptors(InterceptorRegistry registry) {
42   registry.addInterceptor(tulingInterceptor()).addPathPatterns("/*");
43   }
44
45
46   /**
47   * 方法实现说明:配置试图解析器
48   * @author:xsls
49   * @exception:
50   * @date:2019/8/6 16:23
51   */
52   @Bean
53   public InternalResourceViewResolver internalResourceViewResolver() {
54   InternalResourceViewResolver viewResolver = new InternalResourceViewResolve
r();
55   viewResolver.setSuffix(".jsp");
56   viewResolver.setPrefix("/WEB-INF/jsp/");
57   return viewResolver;
58   }
59
60
61
62   @Override
```

```
63    public void configureMessageConverters(List<HttpMessageConverter<?>> conver
ters) {
64    converters.add(new MappingJackson2HttpMessageConverter());
65    }
66
67 }
```

自己去添加个Controller进行测试

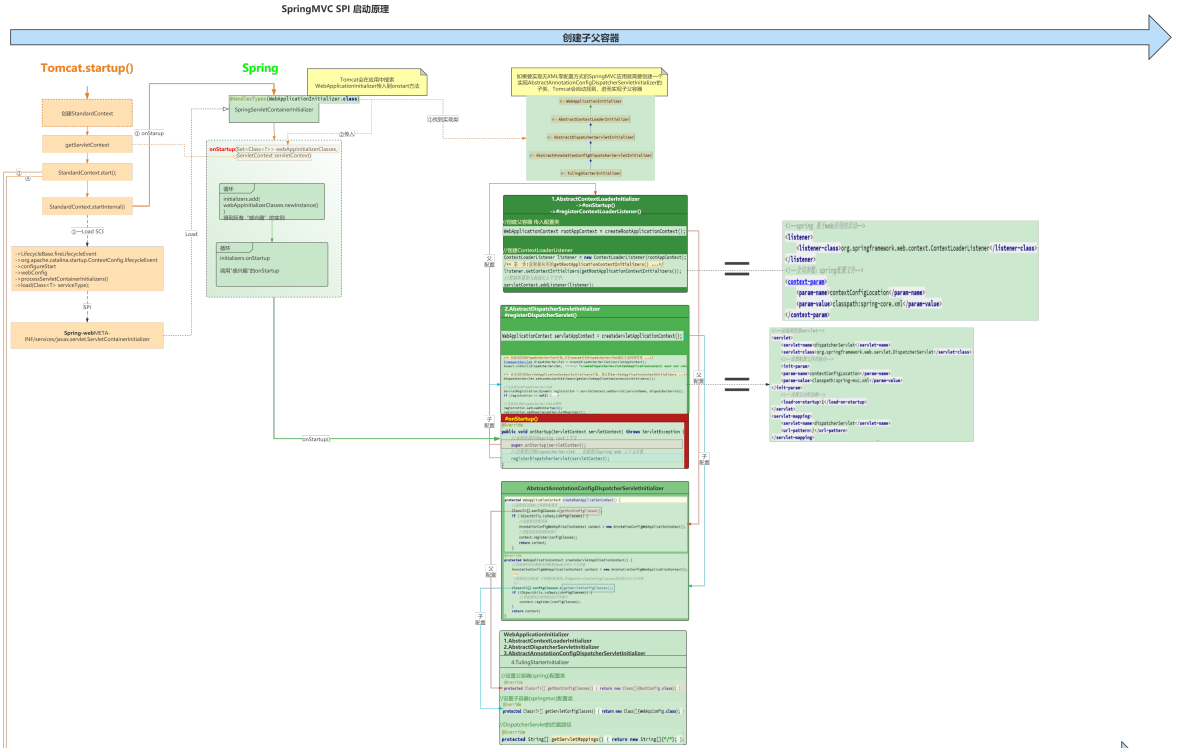
OK， 现在可以访问你的SpringMVC了

4、SPI的方式SpringMVC启动原理

接着我们来看看SPI方式的原理是什么：

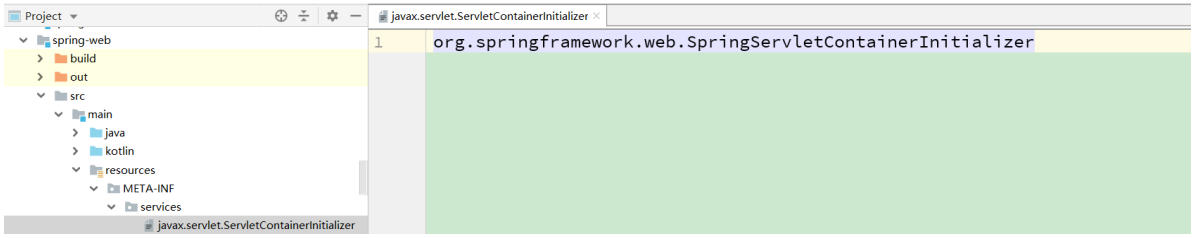
SpringMVC 大致可以分为 启动 和请求 2大部分， 所以我们本文先研究启动部分

流程图：

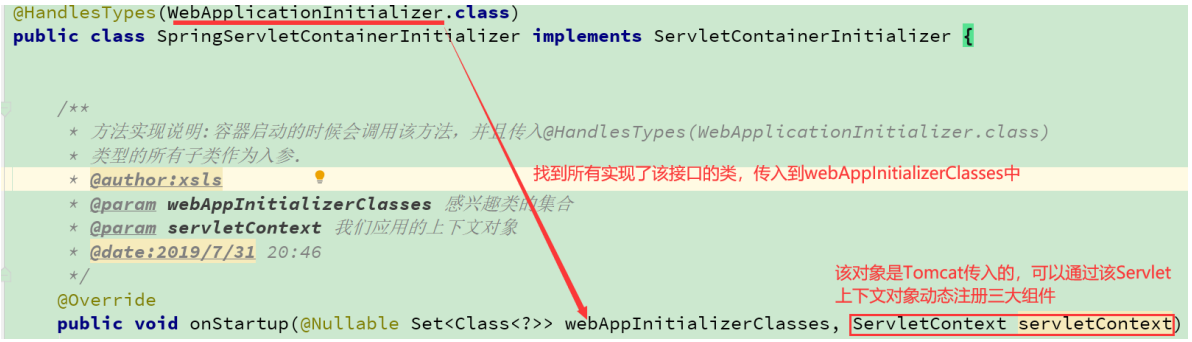


源码流程

1. 外置Tomcat启动的时候通过SPI 找到我们应用中的/META-INF/service/javax.servlet.ServletContainerInitializer

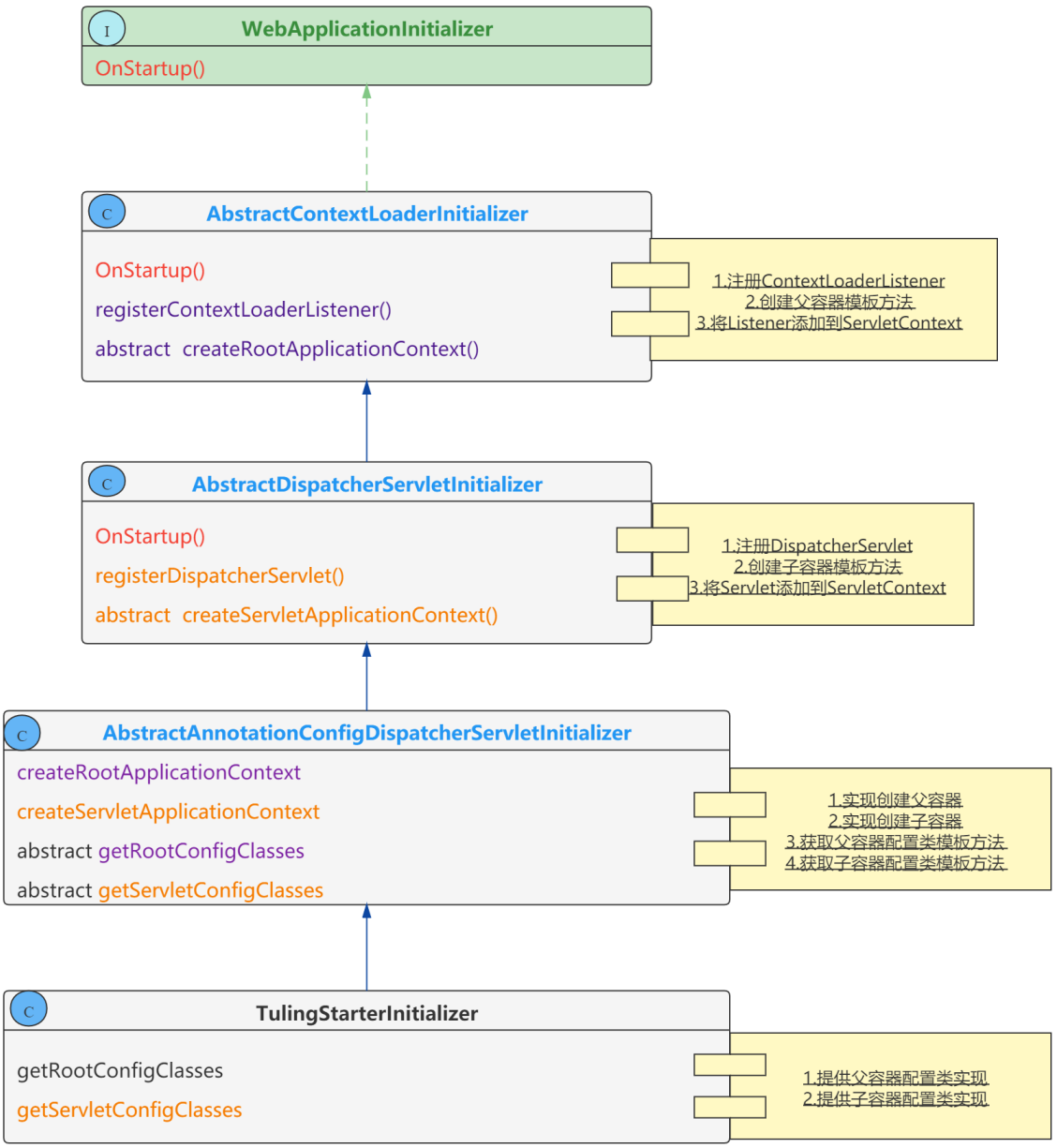


2. 调用SpringServletContainerInitializer.onStartUp()



a. 调用onStartUp()前会先找到@HandlesTypes(WebApplicationInitializer.class) 所有实现了WebApplicationInitializer的类，传入到OnStartup的webAppInitializerClasses参数中，并传入Servlet上下文对象。

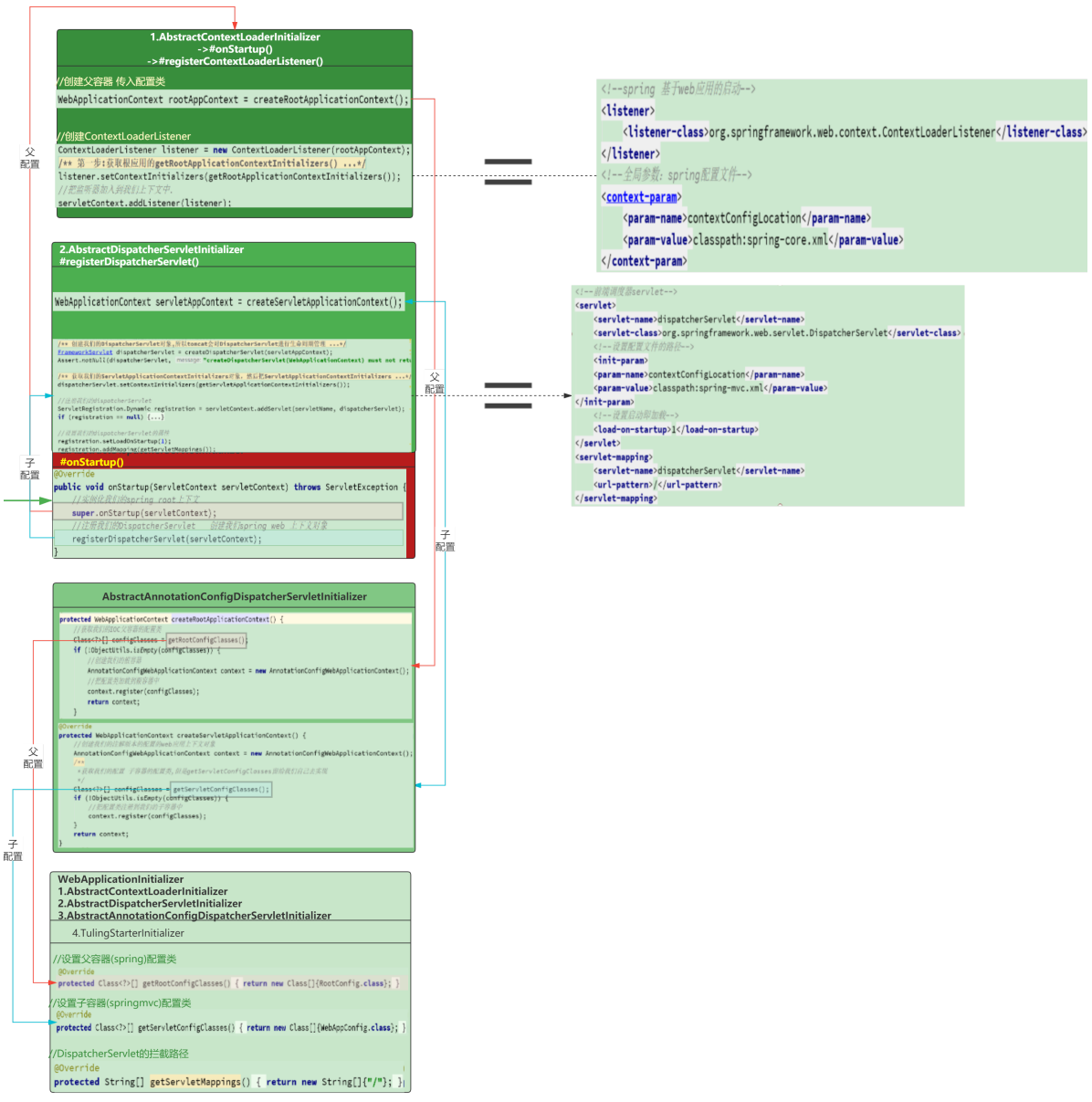
b. 重点关注这组类：他们组成了父子容器



3. 找到所有WebApplicationInitializer的实现类后， 不是接口、不是抽象则通过反射进行实例化（所以，你会发现内部实现类都是抽象的，你想让其起作用我们必须添加一个自定义实现类，在下文提供我的自定义实现类）

4. 调用所有上一步实例化后的对象的onStartup方法

```java
@Override
public void onStartup(@Nullable Set<Class<?>> webAppInitializerClasses, ServletContext servletContext)
        throws ServletException {

    List<WebApplicationInitializer> initializers = new LinkedList<>();
    //传入的兴趣类webAppInitializerClasses的所有子类
    if (webAppInitializerClasses != null) {
        //进行循环取兴趣的类
        for (Class<?> waiClass : webAppInitializerClasses) {
            //判断感兴趣的类不是接口，不是抽象类
            if (!waiClass.isInterface() && !Modifier.isAbstract(waiClass.getModifiers()) &&
                    WebApplicationInitializer.class.isAssignableFrom(waiClass)) {
                try {
                    //通过反射调用创感兴趣类的实例，然后加入到initializers
                    initializers.add((WebApplicationInitializer)
                            ReflectionUtils.accessibleConstructor(waiClass).newInstance());
                }
...

    //依次循环调用我们的感兴趣的类的实例的onStartup方法
    for (WebApplicationInitializer initializer : initializers) {
        initializer.onStartup(servletContext);
    }
}

}
```



1. 首先来到**AbstractDispatcherServletInitializer**#onStartup再执行super.onStartup(servletContext);

```
@Override
public void onStartup(ServletContext servletContext) throws ServletException
{
 //实例化我们的spring root上下文
 super.onStartup(servletContext);
 //注册我们的DispatcherServlet 创建我们spring web 上下文对象
 registerDispatcherServlet(servletContext);
}
```

创建父容器——ContextLoaderListener

2.父类**AbstractContextLoaderInitializer**#onStartup执行registerContextLoaderListener(servletContext);

1. createRootApplicationContext()该方法中会**创建父容器**
    a. 该方法是抽象方法，实现类是AbstractAnnotationConfigDispatcherServletInitializer
        i. 调用getRootConfigClasses();方法获取父容器配置类（此抽象方法在我们自定义的子类中实现提供我们自定义的映射路径 ）
        ii. 创建父容器，注册配置类

```
protected WebApplicationContext createRootApplicationContext() {
    //获取我们的IOC父容器的配置类
    Class<?>[] configClasses = getRootConfigClasses();
    if (!ObjectUtils.isEmpty(configClasses)) {
        //创建我们的根容器
        AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
        //把配置类加载到根容器中
        context.register(configClasses);
        return context;
    }
    else {
        return null;
    }
}
```

1 调用抽象方法获取父容器配置类
该方法可在自定义子类中重写提供

2 创建父容器，并注册配置类

2. 会创建ContextLoaderListener并通过ServletContext注册

```java
protected void registerContextLoaderListener(ServletContext servletContext) {
    /** 创建我们的根的上下文环境WebApplicationContext(AnnotationConfigWebApplicationContext)对象，但是该方在本类中
    WebApplicationContext rootAppContext = createRootApplicationContext();
    //创建的对象WebApplicationContext不为空                    创建父容器
    if (rootAppContext != null) {
        /**
         * <listener>
         *       <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
         *  </listener>
         *
         *  <context-param>
         *      <param-name>contextConfigLocation</param-name>
         *      <param-value>/WEB-INF/app-context.xml</param-value>
         *   </context-param>
         * 创建一个监听器对象
         */
        ContextLoaderListener listener = new ContextLoaderListener(rootAppContext);
        ...
        //把监听器加入到我们上下文中.
        servletContext.addListener(listener);                    创建监听器
    }
    else {...}
}
```

看完大家是不是感觉跟我们XML的配置ContextLoaderListener对上了：

```xml
<!--spring 基于web应用的启动-->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<!--全局参数：spring配置文件-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring-core.xml</param-value>
</context-param>
```

创建子容器——DispatcherServlet

3.回到**AbstractDispatcherServletInitializer**#onStartup再执行registerDispatcherServlet(servletContext);

```java
protected void registerDispatcherServlet(ServletContext servletContext) {
    //获取我们的DispatcherServlet的名称
    String servletName = getServletName();
    ...
    //创建WebApplicationContext对象                    创建子容器
    WebApplicationContext servletAppContext = createServletApplicationContext();
    ...
    /**
     * 创建我们的DispatcherServlet对象,所以tomcat会对DispatcherServlet进行生命周期管理
     */                                    创建DispatcherServlet
    FrameworkServlet dispatcherServlet = createDispatcherServlet(servletAppContext);
    ...
    dispatcherServlet.setContextInitializers(getServletApplicationContextInitializers());

    //注册我们的dispatcherServlet
    ServletRegistration.Dynamic registration = servletContext.addServlet(servletName, dispatcherServlet);
    ...
    //设置我们的dispatcherServlet的属性
    registration.setLoadOnStartup(1);
    registration.addMapping(getServletMappings());          此处是个抽象方法，子类可以扩展
    registration.setAsyncSupported(isAsyncSupported());

    ...
}
```

registerDispatcherServlet方法说明：

1. 调用createServletApplicationContext**创建子容器**
    a. 该方法是抽象方法，实现类是AbstractAnnotationConfigDispatcherServletInitializer

i. 创建子容器（下图很明显不多介绍）

ii. 调用抽象方法：getServletConfigClasses();获得配置类（此抽象方法在我们自定义的子类中实现提供我们自定义的配置类 ）

iii. 配置类除了可以通过ApplicationContext()构造函数的方式传入， 也可以通过这种方式动态添加，不知道了吧~

```
@Override
protected WebApplicationContext createServletApplicationContext() {
    //创建我们的注解版本的配置的web应用上下文对象                    创建子容器
    AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
    /**
     *获取我们的配置 子容器的配置类,但是getServletConfigClasses留给我们自己去实现
     */
    Class<?>[] configClasses = getServletConfigClasses();          调用抽象方法获得子容器的配置类
    if (!ObjectUtils.isEmpty(configClasses)) {
        //把配置类注册到我们的子容器中
        context.register(configClasses);   注册配置类到容器中
    }
    return context;
}
```

2. 调用createDispatcherServlet(servletAppContext);创建DispatcherServlet

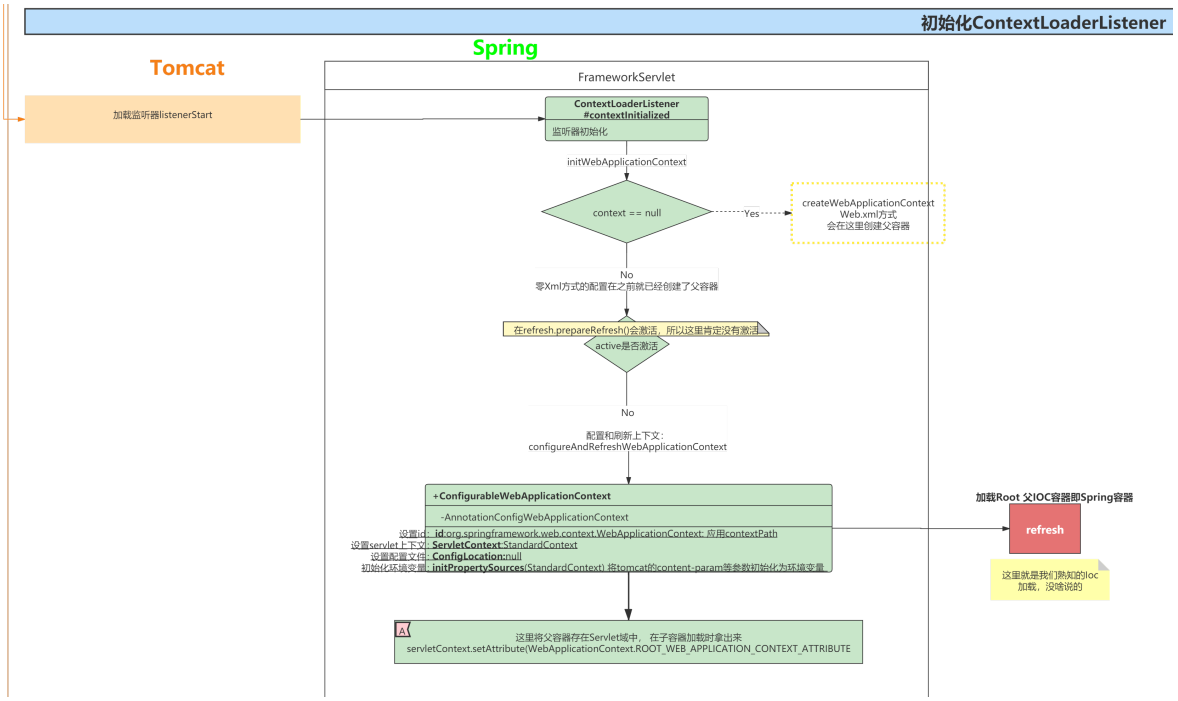
3. 设置启动时加载：registration.setLoadOnStartup(1);

4. 调用抽象方法设置映射路径：getServletMappings()（此抽象方法在我们自定义的子类中实现提供我们自定义的映射路径 ）

> 看完大家是不是感觉跟我们XML的配置DispatcherServlet对上了

```
<!--前端调度器servlet-->
<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!--设置配置文件的路径-->
    <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring-mvc.xml</param-value>
</init-param>
    <!--设置启动即加载-->
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

4. 初始化ContextLoaderListener



ContextLoaderListener加载过程比较简单：

外置tomcat会帮我们调用ContextLoaderListener#contextInitialized 进行初始化

1. xml的方式下会判断容器为空时创建父容器
2. 在里面会调用父容器的refresh方法加载
3. 将父容器存入到Servlet域中供子容器使用

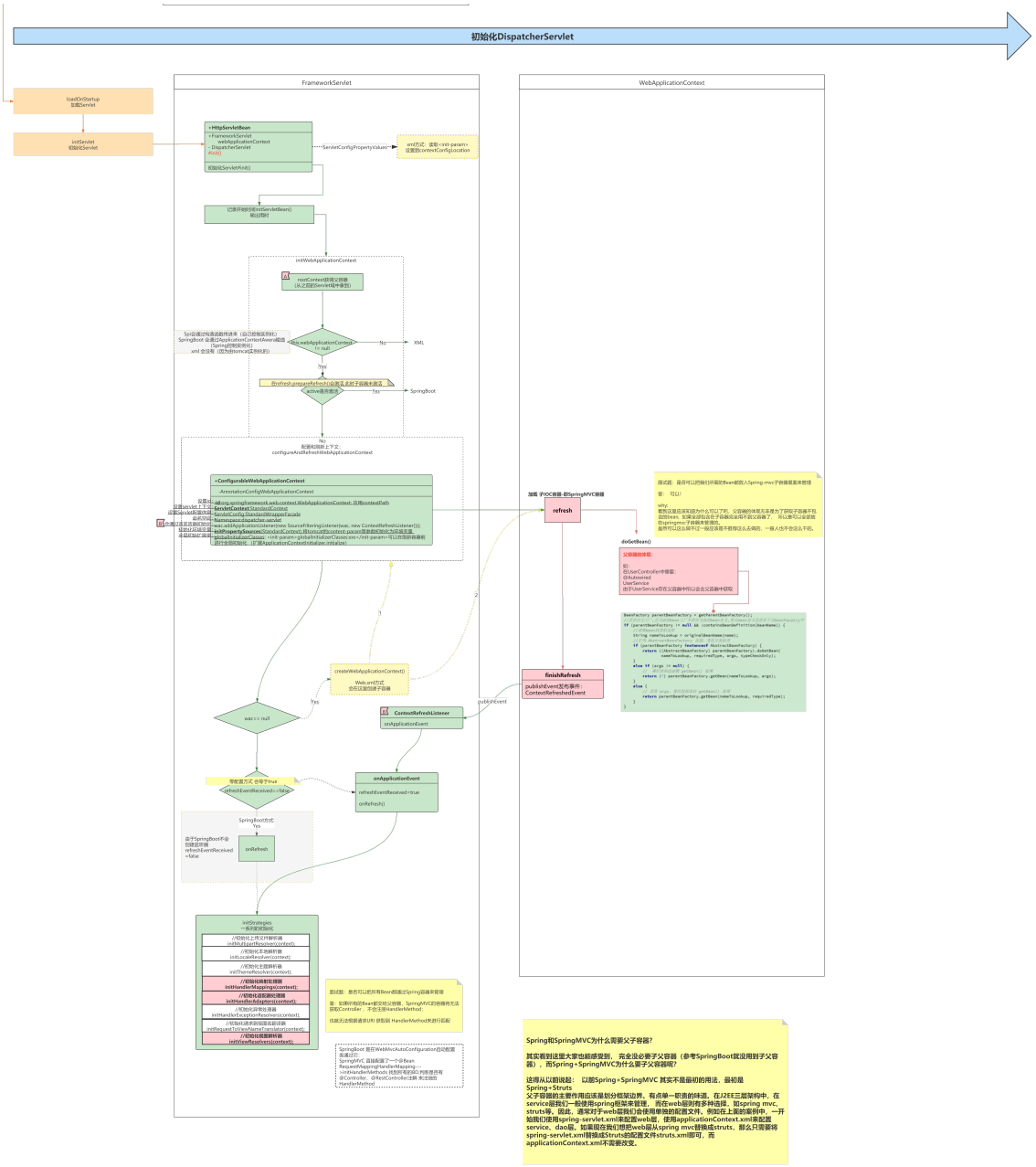
```
public WebApplicationContext initWebApplicationContext(ServletContext servletContext) {
    ...
        /**
         * 在创建xml版本的时候context是空的 所以我们在这里需要在这里创建      根容器对象
         */
        if (this.context == null) {
            /**
             * 注意：由于xml版本中context为空，所以这个if逻辑是用来创建我们的 根容器对象
             */
            this.context = createWebApplicationContext(servletContext);
        }

        if (this.context instanceof ConfigurableWebApplicationContext) {
            //强制转化成ConfigurableWebApplicationContext
            ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) this.context;

            //判断ConfigurableWebApplicationContext 配置上下文版本的是不是激活了
            if (!cwac.isActive()) {//没有激活
                //若此时ConfigurableWebApplicationContext对象的父容器为空 从spring5.0开始这里固定返回null
                if (cwac.getParent() == null) {...}
                //配置和刷新我们的根容器对象
                configureAndRefreshWebApplicationContext(cwac, servletContext);
            }
        }
        /**
         * 把我们的spring上下文保存到 应用上下文对象中
         * 方便我们在Spring web 上下文对象实例化过程会从servletContext取出来
         */

        servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);

        ...

        return this.context;
    }
```

1 xml会在这里创建父容器

2 里面会调用容器的refresh方法

3 放到Servlet域中供子容器使用

5. 初始化DispatcherServlet



可以看到流程比ContextLoaderListener流程更多

外置tomcat会帮我们调用DispatcherServlet#init()　进行初始化--->重点关注：initWebApplicationContext方法

1. *getWebApplicationContext*(getServletContext())获得父容器（从之前的Servlet域中拿到）
2. cwac.setParent(rootContext);给子容器设置父容器
3. 调用configureAndRefreshWebApplicationContext(cwac);

```java
protected WebApplicationContext initWebApplicationContext() {
    /**
     * 从我们的ServletContext对象中获取到Spring root 上下文对象，为啥可以获取，
     * 因为我们在Spring 根容器上下文创建成功后放入到ServletContext对象中
     */
    WebApplicationContext rootContext =
            WebApplicationContextUtils.getWebApplicationContext(getServletContext());
    WebApplicationContext wac = null;                    // 1 从Servlet域中获取父容器

    /**
     *  xml
     *  spi 会通过构造函数将子容器传进来
     *  springboot 将DispatcherServlet配置为Bean 会通过ApplicationContextAware获取唯一容器

     */
    if (this.webApplicationContext != null) {
        wac = this.webApplicationContext;
        //判断webApplicationContext不为空
        if (wac instanceof ConfigurableWebApplicationContext) {
            ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;
            //判断是否激活
            if (!cwac.isActive()) {
                //设置父的上下文对象
                if (cwac.getParent() == null) {
                    // The context instance was injected without an explicit parent -> set
                    // the root application context (if any; may be null) as the parent
                    cwac.setParent(rootContext);         // 2 设置父容器
                }
                /**
                 * 作为SpringMvc 上下文刷新
                 */
                configureAndRefreshWebApplicationContext(cwac);  // 3
            }
        }
    }
```

a. 注册一个监听器（该监听会初始化springmvc所需信息）

i. ContextRefreshedEvent可以看到该监听器监听的是容器refreshed事件， 会在finishRefresh中发布

b. 刷新容器

```java
protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac) {
    ...                                                  // 1 创建监听器，该监听器会在容器的finishedrefresh中发布对应的事件，初始化springmvc所需信息
    wac.addApplicationListener(new SourceFilteringListener(wac, new ContextRefreshListener()));

    ...
    //刷新我们的子容器
    wac.refresh();                                       // 2 刷新容器
}
```

当执行refresh 即加载ioc容器 完了会调用finishRefresh():

1. publishEvent(**new** ContextRefreshedEvent(**this**));发布ContextRefreshedEvent事件
2. 触发上面的ContextRefreshListener监听器：

---->FrameworkServlet.**this**.onApplicationEvent(event);

-------->onRefresh(event.getApplicationContext());

---------->initStrategies(context);

```java
protected void initStrategies(ApplicationContext context) {
 //初始化我们web上下文对象的 用于文件上传下载的解析器对象
 initMultipartResolver(context);
```

```
4   //初始化我们web上下文对象用于处理国际化资源的
5   initLocaleResolver(context);
6   //主题解析器对象初始化
7   initThemeResolver(context);
8   //初始化我们的HandlerMapping
9   initHandlerMappings(context);
10   //实例化我们的HandlerAdapters
11   initHandlerAdapters(context);
12   //实例化我们处理器异常解析器对象
13   initHandlerExceptionResolvers(context);
14   initRequestToViewNameTranslator(context);
15   //给DispatcherSerlvet的ViewResolvers处理器
16   initViewResolvers(context);
17   initFlashMapManager(context);
18  }
```

这里面的每一个方法不用太细看， 就是给SpringMVC准备初始化的数据， 为后续SpringMVC处理请求做准备

基本都是从容器中拿到已经配置的Bean（RequestMappingHandlerMapping、RequestMappingHandlerAdapter、HandlerExceptionResolver ）放到dispatcherServlet中做准备:

```
Map<String, HandlerMapping> matchingBeans =
        BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMapping.class, includeNonSinglet
```

```
Map<String, HandlerAdapter> matchingBeans =
        BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerAdapter.class, includeNonSingleton
```

```
Map<String, HandlerExceptionResolver> matchingBeans = BeanFactoryUtils
        .beansOfTypeIncludingAncestors(context, HandlerExceptionResolver.class, includeNonSingletons: true,
```

...

但是这些Bean又是从哪来的呢？？ 来来来， 回到我们的WebAppConfig我们使用的一个@EnableWebMvc

1. 导入了DelegatingWebMvcConfiguration@Import(DelegatingWebMvcConfiguration.class)
2. DelegatingWebMvcConfiguration的父类就配置了这些Bean

3. 而且我告诉你SpringBoot也是用的这种方式，

```
@Configuration
@ComponentScan(basePackages = {"com.tuling"},includeFilters = {
        @ComponentScan.Filter(type = FilterType.ANNOTATION,value = {RestControl
},useDefaultFilters =false)
@EnableWebMvc    // ≈<mvc:annotation-driven/>
public class WebAppConfig implements WebMvcConfigurer{
```

总结

1. Tomcat在启动时会通过SPI注册 ContextLoaderListener和DispatcherServlet对象
    a. 同时创建父子容器
        i. 分别创建在ContextLoaderListener初始化时创建父容器设置配置类
        ii. 在DispatcherServlet初始化时创建子容器 即2个ApplicationContext实例设置配置类
2. Tomcat在启动时执行ContextLoaderListener和DispatcherServlet对象的初始化方法， 执行容器refresh进行加载
3. 在子容器加载时 创建SpringMVC所需的Bean和预准备的数据：(通过配置类+@EnableWebMvc配置（DelegatingWebMvcConfiguration）——可实现WebMvcConfigurer进行定制扩展)
    a. RequestMappingHandlerMapping，它会处理@RequestMapping 注解
    b. RequestMappingHandlerAdapter，则是处理请求的适配器，确定调用哪个类的哪个方法，并且构造方法参数，返回值。
    c. HandlerExceptionResolver 错误视图解析器
    d. addDefaultHttpMessageConverters 添加默认的消息转换器（解析json、解析xml）
    e. 等....
4. 子容器需要注入父容器的Bean时（比如Controller中需要@Autowired Service的Bean）；会先从子容器中找，没找到会去父容器中找： 详情见AbstractBeanFactory#doGetBean方法

```
1  /**
```

```
 * 一般情况下,只有Spring 和SpringMvc整合的时才会有父子容器的概念,
 * 作用:
 * 比如我们的Controller中注入Service的时候，发现我们依赖的是一个引用对象，那么他就会调用getBean去把service找出来
 * 但是当前所在的容器是web子容器，那么就会在这里的 先去父容器找
 */
BeanFactory parentBeanFactory = getParentBeanFactory();
//若存在父工厂,且当前的bean工厂不存在当前的bean定义,那么bean定义是存在于父beanFactory中
if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
  //获取bean的原始名称
  String nameToLookup = originalBeanName(name);
  //若为 AbstractBeanFactory 类型，委托父类处理
  if (parentBeanFactory instanceof AbstractBeanFactory) {
  return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
  nameToLookup, requiredType, args, typeCheckOnly);
  }
  else if (args != null) {
  // 委托给构造函数 getBean() 处理
  return (T) parentBeanFactory.getBean(nameToLookup, args);
  }
  else {
  // 没有 args，委托给标准的 getBean() 处理
  return parentBeanFactory.getBean(nameToLookup, requiredType);
  }
}
```

用几道面试题做个总结:

Spring和SpringMVC为什么需要父子容器？不要不行吗?

就实现层面来说不用子父容器也可以完成所需功能（参考：`SpringBoot`就没用子父容器）

1. 所以父子容器的主要作用应该是早期Spring为了划分框架边界。有点单一职责的味道。`service`、`dao`层我们一般使用`spring`框架来管理、`controller`层交给springmvc管理
2. 规范整体架构 使 父容器`service`无法访问子容器`controller`、子容器`controller`可以访问父容器 `service`

3. 方便子容器的切换。如果现在我们想把web层从spring mvc替换成struts，那么只需要将spring-mvc.xml替换成Struts的配置文件struts.xml即可，而spring-core.xml不需要改变。

4. 为了节省重复bean创建

是否可以把所有Bean都通过Spring容器来管理？（Spring的applicationContext.xml中配置全局扫描)

不可以，这样会导致我们请求接口的时候产生404。 如果所有的Bean都交给父容器，SpringMVC在初始化HandlerMethods的时候（initHandlerMethods）无法根据Controller的handler方法注册HandlerMethod，并没有去查找父容器的bean;

也就无法根据请求URI 获取到 HandlerMethod来进行匹配.

```
String[] beanNames = (this.detectHandlerMethodsInAncestorContexts ?
        BeanFactoryUtils.beanNamesForTypeIncludingAncestors(obtainApplicationContext(), Object.class) :
        obtainApplicationContext().getBeanNamesForType(Object.class));
                                   实际上是获得当前容器的所有BeanDefinitionNames
for (String beanName : beanNames) {
```

是否可以把我们所需的Bean都放入Spring-mvc子容器里面来管理（springmvc的spring-servlet.xml中配置全局扫描）？

可以， 因为父容器的体现无非是为了获取子容器不包含的bean，如果全部包含在子容器完全用不到父容器了， 所以是可以全部放在springmvc子容器来管理的。

虽然可以这么做不过一般应该是不推荐这么去做的，一般人也不会这么干的。**如果你的项目里有用到事物、或者aop记得也需要把这部分配置需要放到Spring-mvc子容器的配置文件来，不然一部分内容在子容器和一部分内容在父容器,可能就会导致你的事物或者AOP不生效。** 所以如果aop或事物如果不生效也有可能是通过父容器(spring)去增强子容器(Springmvc)，也就无法增强 这也是很多同学会遇到的问题。