

## 45、web实验-抽取公共页面

### 官方文档 - Template Layout

- 公共页面 `/templates/common.html`

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"><!--注意要添加xmlns:th才能添加thymeleaf的标签-->
<head th:fragment="commonheader">
    <!--common-->
    <link href="css/style.css" th:href="@{/css/style.css}" rel="stylesheet">
    <link href="css/style-responsive.css" th:href="@{/css/style-responsive.css}"
rel="stylesheet">
    ...
</head>
<body>
<!-- left side start-->
<div id="leftmenu" class="left-side sticky-left-side">
    ...

    <div class="left-side-inner">
        ...

        <!--sidebar nav start-->
        <ul class="nav nav-pills nav-stacked custom-nav">
            <li><a th:href="@{/main.html}"><i class="fa fa-home"></i> <span>Dashboard</span></a>
</li>
            ...
            <li class="menu-list nav-active"><a href="#"><i class="fa fa-th-list"></i>
<span>Data Tables</span></a>
                <ul class="sub-menu-list">
                    <li><a th:href="@{/basic_table}"> Basic Table</a></li>
                    <li><a th:href="@{/dynamic_table}"> Advanced Table</a></li>
                    <li><a th:href="@{/responsive_table}"> Responsive Table</a></li>
                    <li><a th:href="@{/editable_table}"> Edit Table</a></li>
                </ul>
            </li>
            ...
        </ul>
        <!--sidebar nav end-->
    </div>
</div>
<!-- left side end-->

<!-- header section start-->
<div th:fragment="headermenu" class="header-section">

    <!--toggle button start-->
    <a class="toggle-btn"><i class="fa fa-bars"></i></a>
```

```

        <!--toggle button end-->
        ...

</div>
<!-- header section end-->

<div id="commonsript">
    <!-- Placed js at the end of the document so the pages load faster -->
    <script th:src="@{/js/jquery-1.10.2.min.js}"></script>
    <script th:src="@{/js/jquery-ui-1.9.2.custom.min.js}"></script>
    <script th:src="@{/js/jquery-migrate-1.2.1.min.js}"></script>
    <script th:src="@{/js/bootstrap.min.js}"></script>
    <script th:src="@{/js/modernizr.min.js}"></script>
    <script th:src="@{/js/jquery.nicescroll.js}"></script>
    <!--common scripts for all pages-->
    <script th:src="@{/js/scripts.js}"></script>
</div>
</body>
</html>

```

- `/templates/table/basic_table.html`

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0">
    <meta name="description" content="">
    <meta name="author" content="ThemeBucket">
    <link rel="shortcut icon" href="#" type="image/png">

    <title>Basic Table</title>
    <div th:include="common :: commonheader"> </div><!--将common.html的代码段 插进来-->
</head>

<body class="sticky-header">

<section>
<div th:replace="common :: #leftmenu"></div>

    <!-- main content start-->
    <div class="main-content" >

        <div th:replace="common :: headermenu"></div>
        ...
    </div>
    <!-- main content end-->
</section>

<!-- Placed js at the end of the document so the pages load faster -->
<div th:replace="common :: #commonsript"></div>

```

```
</body>
</html>
```

[Difference between th:insert and th:replace \(and th:include\)](#)

## 46、web实验-遍历数据与页面bug修改

控制层代码：

```
@GetMapping("/dynamic_table")
public String dynamic_table(Model model){
    //表格内容的遍历
    List<User> users = Arrays.asList(new User("zhangsan", "123456"),
                                     new User("lisi", "123444"),
                                     new User("haha", "aaaaa"),
                                     new User("hehe ", "aaddd"));

    model.addAttribute("users",users);

    return "table/dynamic_table";
}
```

页面代码：

```
<table class="display table table-bordered" id="hidden-table-info">
    <thead>
        <tr>
            <th>#</th>
            <th>用户名</th>
            <th>密码</th>
        </tr>
    </thead>
    <tbody>
        <tr class="gradeX" th:each="user,stats:${users}">
            <td th:text="${stats.count}">Trident</td>
            <td th:text="${user.userName}">Internet</td>
            <td >[[${user.password}]]</td>
        </tr>
    </tbody>
</table>
```

## 47、视图解析-【源码分析】-视图解析器与视图

视图解析原理流程：

1. 目标方法处理的过程中（阅读 `DispatcherServlet` 源码），所有数据都会被放在 `ModelAndViewContainer` 里面，其中包括数据和视图地址。
2. 方法的参数是一个自定义类型对象（从请求参数中确定的），把他重新放在 `ModelAndViewContainer`。
3. 任何目标方法执行完成以后都会返回 `ModelAndView`（数据和视图地址）。
4. `processDispatchResult()` 处理派发结果（页面改如何响应）
  - `render(mv, request, response);` 进行页面渲染逻辑
    - 根据方法的 `String` 返回值得到 `View` 对象【定义了页面的渲染逻辑】
      1. 所有的视图解析器尝试是否能根据当前返回值得到 `View` 对象
      2. 得到了 `redirect:/main.html --> Thymeleaf new RedirectView()`。
      3. `ContentNegotiationViewResolver` 里面包含了下面所有的视图解析器，内部还是利用下面所有视图解析器得到视图对象。
      4. `view.render(mv.getModelInternal(), request, response);` 视图对象调用自定义的render进行页面渲染工作。
    - `RedirectView` 如何渲染【重定向到一个页面】
      - 获取目标url地址
    - `response.sendRedirect(encodedURL);`

视图解析：

- 返回值以 ``forward:`` 开始： ``new InternalResourceView(forwardUrl);` --> 转发`  
``request.getRequestDispatcher(path).forward(request, response);``
- 返回值以 ``redirect:`` 开始： ``new RedirectView()` --> render就是重定向`
- 返回值是普通字符串： ``new ThymeleafView()` --->`

阅读源码：最好自己在IDE，打断点，且Debug模式运行实例，这样比较没那么沉闷。

## 48、拦截器-登录检查与静态资源放行

1. 编写一个拦截器实现 `HandlerInterceptor` 接口
2. 拦截器注册到容器中（实现 `WebMvcConfigurer` 的 `addInterceptors()`）
3. 指定拦截规则（**注意**，如果是拦截所有，静态资源也会被拦截）

编写一个实现 `HandlerInterceptor` 接口的拦截器：

```
@Slf4j
public class LoginInterceptor implements HandlerInterceptor {

    /**
     * 目标方法执行之前
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

        String requestURI = request.getRequestURI();
```

```

        log.info("preHandle拦截的请求路径是{}",requestURI);

        //登录检查逻辑
        HttpSession session = request.getSession();

        Object loginUser = session.getAttribute("loginUser");

        if(loginUser != null){
            //放行
            return true;
        }

        //拦截住。未登录。跳转到登录页
        request.setAttribute("msg","请先登录");
        // re.sendRedirect("/");
        request.getRequestDispatcher("/").forward(request,response);
        return false;
    }

    /**
     * 目标方法执行完成以后
     */
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object
handler, ModelAndView modelAndView) throws Exception {
        log.info("postHandle执行{}",modelAndView);
    }

    /**
     * 页面渲染以后
     */
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object
handler, Exception ex) throws Exception {
        log.info("afterCompletion执行异常{}",ex);
    }
}

```

拦截器注册到容器中 && 指定拦截规则：

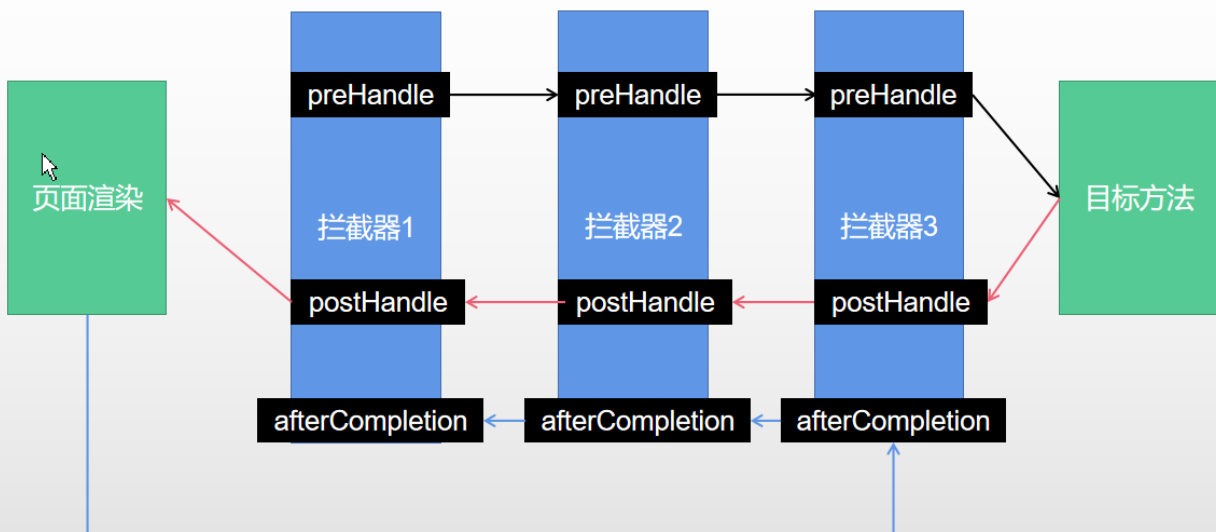
```

@Configuration
public class AdminWebConfig implements WebMvcConfigurer{
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoginInterceptor())//拦截器注册到容器中
            .addPathPatterns("/**") //所有请求都被拦截包括静态资源
            .excludePathPatterns("/", "/login", "/css/**", "/fonts/**", "/images/**",
                "/js/**", "/aa/**"); //放行的请求
    }
}

```

## 49、拦截器-【源码分析】-拦截器的执行时机和原理

1. 根据当前请求，找到 `HandlerExecutionChain`（可以处理请求的handler以及handler的所有 拦截器）
2. 先来顺序执行 所有拦截器的 `preHandle()` 方法。
  - 如果当前拦截器 `preHandle()` 返回为 `true`。则执行下一个拦截器的 `preHandle()`
  - 如果当前拦截器返回为 `false`。直接倒序执行所有已经执行了的拦截器的 `afterCompletion()`。
3. 如果任何一个拦截器返回 `false`，直接跳出不执行目标方法。
4. 所有拦截器都返回 `true`，才执行目标方法。
5. 倒序执行所有拦截器的 `postHandle()` 方法。
6. 前面的步骤有任何异常都会直接倒序触发 `afterCompletion()`。
7. 页面成功渲染完成以后，也会倒序触发 `afterCompletion()`。



[https://blog.csdn.net/qq\\_38262222](https://blog.csdn.net/qq_38262222)

`DispatcherServlet` 中涉及到 `HandlerInterceptor` 的地方：

```
public class DispatcherServlet extends FrameworkServlet {

    ...

    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        boolean multipartRequestParsed = false;

        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

        try {
            ModelAndView mv = null;
            Exception dispatchException = null;
```

```

        ...

        //该方法内调用HandlerInterceptor的preHandle()
        if (!mappedHandler.applyPreHandle(processedRequest, response)) {
            return;
        }

        // Actually invoke the handler.
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

        ...
        //该方法内调用HandlerInterceptor的postHandle()
        mappedHandler.applyPostHandle(processedRequest, response, mv);
    }
    processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
    }
    catch (Exception ex) {
        //该方法内调用HandlerInterceptor接口的afterCompletion方法
        triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
    }
    catch (Throwable err) {
        //该方法内调用HandlerInterceptor接口的afterCompletion方法
        triggerAfterCompletion(processedRequest, response, mappedHandler,
            new NestedServletException("Handler processing failed", err));
    }
    finally {
        ...
    }
}

private void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse
response,
    @Nullable HandlerExecutionChain mappedHandler, Exception ex) throws Exception {

    if (mappedHandler != null) {
        //该方法内调用HandlerInterceptor接口的afterCompletion方法
        mappedHandler.triggerAfterCompletion(request, response, ex);
    }
    throw ex;
}

private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
    @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,
    @Nullable Exception exception) throws Exception {

    ...

    if (mappedHandler != null) {
        //该方法内调用HandlerInterceptor接口的afterCompletion方法
        // Exception (if any) is already handled..
        mappedHandler.triggerAfterCompletion(request, response, null);
    }
}

```

```

    }

}

```

```

public class HandlerExecutionChain {

    ...

    boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response) throws
Exception {
        for (int i = 0; i < this.interceptorList.size(); i++) {
            HandlerInterceptor interceptor = this.interceptorList.get(i);
            //HandlerInterceptor的preHandle方法
            if (!interceptor.preHandle(request, response, this.handler)) {

                triggerAfterCompletion(request, response, null);
                return false;
            }
            this.interceptorIndex = i;
        }
        return true;
    }

    void applyPostHandle(HttpServletRequest request, HttpServletResponse response, @Nullable
ModelAndView mv)
        throws Exception {

        for (int i = this.interceptorList.size() - 1; i >= 0; i--) {
            HandlerInterceptor interceptor = this.interceptorList.get(i);

            //HandlerInterceptor接口的postHandle方法
            interceptor.postHandle(request, response, this.handler, mv);
        }
    }

    void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response,
@Nullable Exception ex) {
        for (int i = this.interceptorIndex; i >= 0; i--) {
            HandlerInterceptor interceptor = this.interceptorList.get(i);
            try {
                //HandlerInterceptor接口的afterCompletion方法
                interceptor.afterCompletion(request, response, this.handler, ex);
            }
            catch (Throwable ex2) {
                logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);
            }
        }
    }
}

```



```
}
```

## 50、文件上传-单文件与多文件上传的使用

- 页面代码 `/static/form/form_layouts.html`

```
<form role="form" th:action="@{/upload}" method="post" enctype="multipart/form-data">
    <div class="form-group">
        <label for="exampleInputEmail1">邮箱</label>
        <input type="email" name="email" class="form-control" id="exampleInputEmail1"
placeholder="Enter email">
    </div>

    <div class="form-group">
        <label for="exampleInputPassword1">名字</label>
        <input type="text" name="username" class="form-control" id="exampleInputPassword1"
placeholder="Password">
    </div>

    <div class="form-group">
        <label for="exampleInputFile">头像</label>
        <input type="file" name="headerImg" id="exampleInputFile">
    </div>

    <div class="form-group">
        <label for="exampleInputFile">生活照</label>
        <input type="file" name="photos" multiple>
    </div>

    <div class="checkbox">
        <label>
            <input type="checkbox"> Check me out
        </label>
    </div>
    <button type="submit" class="btn btn-primary">提交</button>
</form>
```

- 控制层代码

```
@Slf4j
@Controller
public class FormTestController {

    @GetMapping("/form_layouts")
    public String form_layouts(){
        return "form/form_layouts";
    }

    @PostMapping("/upload")
```

```

public String upload(@RequestParam("email") String email,
                    @RequestParam("username") String username,
                    @RequestParam("headerImg") MultipartFile headerImg,
                    @RequestParam("photos") MultipartFile[] photos) throws IOException {

    log.info("上传的信息: email={}, username={}, headerImg={}, photos={}",
            email, username, headerImg.getSize(), photos.length);

    if(!headerImg.isEmpty()){
        //保存到文件服务器, OSS服务器
        String originalFilename = headerImg.getOriginalFilename();
        headerImg.transferTo(new File("H:\\cache\\"+originalFilename));
    }

    if(photos.length > 0){
        for (MultipartFile photo : photos) {
            if(!photo.isEmpty()){
                String originalFilename = photo.getOriginalFilename();
                photo.transferTo(new File("H:\\cache\\"+originalFilename));
            }
        }
    }

    return "main";
}

```

文件上传相关的配置类:

- `org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration`
- `org.springframework.boot.autoconfigure.web.servlet.MultipartProperties`

文件大小相关配置项:

```

spring.servlet.multipart.max-file-size=10MB
spring.servlet.multipart.max-request-size=100MB

```

## 51、文件上传-【源码流程】文件上传参数解析器

文件上传相关的自动配置类 `MultipartAutoConfiguration` 有创建文件上传参数解析器 `StandardServletMultipartResolver`。

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ Servlet.class, StandardServletMultipartResolver.class,
    MultipartConfigElement.class })
@ConditionalOnProperty(prefix = "spring.servlet.multipart", name = "enabled", matchIfMissing =
    true)
@ConditionalOnWebApplication(type = Type.SERVLET)
@EnableConfigurationProperties(MultipartProperties.class)

```

```

public class MultipartAutoConfiguration {

    private final MultipartProperties multipartProperties;

    public MultipartAutoConfiguration(MultipartProperties multipartProperties) {
        this.multipartProperties = multipartProperties;
    }

    @Bean
    @ConditionalOnMissingBean({ MultipartConfigElement.class, CommonsMultipartResolver.class })
    public MultipartConfigElement multipartConfigElement() {
        return this.multipartProperties.createMultipartConfig();
    }

    @Bean(name = DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)
    @ConditionalOnMissingBean(MultipartResolver.class)
    public StandardServletMultipartResolver multipartResolver() {
        //配置好文件上传解析器
        StandardServletMultipartResolver multipartResolver = new
StandardServletMultipartResolver();
        multipartResolver.setResolveLazily(this.multipartProperties.isResolveLazily());
        return multipartResolver;
    }
}

```

//文件上传解析器

```

public class StandardServletMultipartResolver implements MultipartResolver {

    private boolean resolveLazily = false;

    public void setResolveLazily(boolean resolveLazily) {
        this.resolveLazily = resolveLazily;
    }

    @Override
    public boolean isMultipart(HttpServletRequest request) {
        return StringUtils.startsWithIgnoreCase(request.getContentType(), "multipart/");
    }

    @Override
    public MultipartHttpServletRequest resolveMultipart(HttpServletRequest request) throws
MultipartException {
        return new StandardMultipartHttpServletRequest(request, this.resolveLazily);
    }

    @Override
    public void cleanupMultipart(MultipartHttpServletRequest request) {
        if (!(request instanceof AbstractMultipartHttpServletRequest) ||
            ((AbstractMultipartHttpServletRequest) request).isResolved()) {

```

```

        // To be on the safe side: explicitly delete the parts,
        // but only actual file parts (for Resin compatibility)
        try {
            for (Part part : request.getParts()) {
                if (request.getFile(part.getName()) != null) {
                    part.delete();
                }
            }
        }
        catch (Throwable ex) {
            LogFactory.getLog(getClass()).warn("Failed to perform cleanup of multipart
items", ex);
        }
    }
}
}

```

```

public class DispatcherServlet extends FrameworkServlet {

    @Nullable
    private MultipartResolver multipartResolver;

    private void initMultipartResolver(ApplicationContext context) {
        ...

        //这个就是配置类配置的StandardServletMultipartResolver文件上传解析器
        this.multipartResolver = context.getBean(MULTIPART_RESOLVER_BEAN_NAME,
MultipartResolver.class);
        ...
    }

    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        boolean multipartRequestParsed = false; //最后finally的回收flag
        ...
        try {
            ModelAndView mv = null;
            Exception dispatchException = null;

            try {
                //做预处理,如果有上传文件 就new StandardMultipartHttpServletRequest包装类
                processedRequest = checkMultipart(request);
                multipartRequestParsed = (processedRequest != request);
                // Determine handler for the current request.
                mappedHandler = getHandler(processedRequest);

                ...
            }
        }
    }
}

```

```

        // Determine handler adapter for the current request.
        HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

        ...

        // Actually invoke the handler.
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

    }
    ....

    finally {

        ...

        if (multipartRequestParsed) {
            cleanupMultipart(processedRequest);
        }
    }
}

protected HttpServletRequest checkMultipart(HttpServletRequest request) throws
MultipartException {
    if (this.multipartResolver != null && this.multipartResolver.isMultipart(request)) {
        ...
        return this.multipartResolver.resolveMultipart(request);
        ...
    }
}

protected void cleanupMultipart(HttpServletRequest request) {
    if (this.multipartResolver != null) {
        MultipartHttpServletRequest multipartRequest =
            WebUtils.getNativeRequest(request, MultipartHttpServletRequest.class);
        if (multipartRequest != null) {
            this.multipartResolver.cleanupMultipart(multipartRequest);
        }
    }
}
}

```

`mv = ha.handle(processedRequest, response, mappedHandler.getHandler());` 跳到以下的类

```

public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
    implements BeanFactoryAware, InitializingBean {
    @Override
    protected ModelAndView handleInternal(HttpServletRequest request,
        HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
        ModelAndView mav;
        ...

        mav = invokeHandlerMethod(request, response, handlerMethod);
    }
}

```

```

        ...
        return mav;
    }

    @Nullable
    protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
        HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

        ServletWebRequest webRequest = new ServletWebRequest(request, response);
        try {
            WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);
            ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);

            ServletInvocableHandlerMethod invocableMethod =
                createInvocableHandlerMethod(handlerMethod);
            if (this.argumentResolvers != null) { //关注点
                invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
            }
            ...
            invocableMethod.invokeAndHandle(webRequest, mavContainer);
            ...

            return getModelAndView(mavContainer, modelFactory, webRequest);
        }
        finally {
            webRequest.requestCompleted();
        }
    }
}

```

`this.argumentResolvers` 其中主角类 `RequestPartMethodArgumentResolver` 用来生成

```

public class ServletInvocableHandlerMethod extends InvocableHandlerMethod {

    ...
    public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer
        mavContainer,
        Object... providedArgs) throws Exception {
        Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);
        ...
    }

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

        Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
    }
}

```

```

        ...
        return doInvoke(args); //反射调用
    }

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

    //处理得出multipart参数, 准备稍后的反射调用 (@PostMapping标记的上传方法)
    protected Object[] getMethodArgumentValues(NativeWebRequest request, @Nullable
        ModelAndViewContainer mavContainer,
        Object... providedArgs) throws Exception {

        MethodParameter[] parameters = getMethodParameters();
        ...
        Object[] args = new Object[parameters.length];
        for (int i = 0; i < parameters.length; i++) {
            MethodParameter parameter = parameters[i];
            parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);
            args[i] = findProvidedArgument(parameter, providedArgs);
            if (args[i] != null) {
                continue;
            }
            //关注点1
            if (!this.resolvers.supportsParameter(parameter)) {
                throw new IllegalStateException(formatArgumentError(parameter, "No suitable
resolver"));
            }
            try {
                //关注点2
                args[i] = this.resolvers.resolveArgument(parameter, mavContainer, request,
this.dataBinderFactory);
            }
            catch (Exception ex) {
                ...
            }
        }
        return args;
    }
}

```

```

public class RequestPartMethodArgumentResolver extends
    AbstractMessageConverterMethodArgumentResolver {

    //对应上面代码关注点1
    @Override

```

```

    public boolean supportsParameter(MethodParameter parameter) {
        //标注@RequestPart的参数
        if (parameter.hasParameterAnnotation(RequestPart.class)) {
            return true;
        }
        else {
            if (parameter.hasParameterAnnotation(RequestParam.class)) {
                return false;
            }
            return
MultipartResolutionDelegate.isMultipartArgument(parameter.nestedIfOptional());
        }
    }

    //对应上面代码关注点2
    @Override
    @Nullable
    public Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer
mavContainer,
        NativeWebRequest request, @Nullable WebDataBinderFactory binderFactory) throws
Exception {

        HttpServletRequest servletRequest = request.getNativeRequest(HttpServletRequest.class);
        Assert.state(servletRequest != null, "No HttpServletRequest");

        RequestPart requestPart = parameter.getParameterAnnotation(RequestPart.class);
        boolean isRequired = ((requestPart == null || requestPart.required()) &&
!parameter.isOptional());

        String name = getPartName(parameter, requestPart);
        parameter = parameter.nestedIfOptional();
        Object arg = null;

        //封装成MultipartFile类型的对象作参数
        Object mpArg = MultipartResolutionDelegate.resolveMultipartArgument(name, parameter,
servletRequest);
        if (mpArg != MultipartResolutionDelegate.UNRESOLVABLE) {
            arg = mpArg;
        }

        ...

        return adaptArgumentIfNecessary(arg, parameter);
    }
}

```

```

public final class MultipartResolutionDelegate {
    ...

    @Nullable
    public static Object resolveMultipartArgument(String name, MethodParameter parameter,

```



```

HttpServletRequest request)
    throws Exception {

    MultipartHttpServletRequest multipartRequest =
        WebUtils.getNativeRequest(request, MultipartHttpServletRequest.class);
    boolean isMultipart = (multipartRequest != null || isMultipartContent(request));

    if (MultipartFile.class == parameter.getNestedParameterType()) {
        if (!isMultipart) {
            return null;
        }
        if (multipartRequest == null) {
            multipartRequest = new StandardMultipartHttpServletRequest(request);
        }
        return multipartRequest.getFile(name);
    }
    else if (isMultipartFileCollection(parameter)) {
        if (!isMultipart) {
            return null;
        }
        if (multipartRequest == null) {
            multipartRequest = new StandardMultipartHttpServletRequest(request);
        }
        List<MultipartFile> files = multipartRequest.getFiles(name);
        return (!files.isEmpty() ? files : null);
    }
    else if (isMultipartFileArray(parameter)) {
        if (!isMultipart) {
            return null;
        }
        if (multipartRequest == null) {
            multipartRequest = new StandardMultipartHttpServletRequest(request);
        }
        List<MultipartFile> files = multipartRequest.getFiles(name);
        return (!files.isEmpty() ? files.toArray(new MultipartFile[0]) : null);
    }
    else if (Part.class == parameter.getNestedParameterType()) {
        if (!isMultipart) {
            return null;
        }
        return request.getPart(name);
    }
    else if (isPartCollection(parameter)) {
        if (!isMultipart) {
            return null;
        }
        List<Part> parts = resolvePartList(request, name);
        return (!parts.isEmpty() ? parts : null);
    }
    else if (isPartArray(parameter)) {
        if (!isMultipart) {
            return null;
        }
    }
}

```

```

        List<Part> parts = resolvePartList(request, name);
        return (!parts.isEmpty() ? parts.toArray(new Part[0]) : null);
    }
    else {
        return UNRESOLVABLE;
    }
}

...

}

```

## 52、错误处理-SpringBoot默认错误处理机制

[Spring Boot官方文档 - Error Handling](#)

默认规则：

- 默认情况下，Spring Boot提供 `/error` 处理所有错误的映射
- 机器客户端，它将生成JSON响应，其中包含错误，HTTP状态和异常消息的详细信息。对于浏览器客户端，响应一个“whitelabel”错误视图，以HTML格式呈现相同的数据

```

{
  "timestamp": "2020-11-22T05:53:28.416+00:00",
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/asadada"
}

```

- 要对其进行自定义，添加 `View` 解析为 `error`
- 要完全替换默认行为，可以实现 `ExceptionHandler` 并注册该类型的Bean定义，或添加 `ErrorAttributes` 类型的组件 以使用现有机制但替换其内容。
- `/templates/error/` 下的4xx, 5xx页面会被自动解析

## 53、错误处理-【源码分析】底层组件功能分析

- `ErrorMvcAutoConfiguration` 自动配置异常处理规则
- 容器中的组件：类型： `DefaultErrorAttributes` --> id: `errorAttributes`
- `public class DefaultErrorAttributes implements ErrorAttributes, HandlerExceptionResolver`
  - `DefaultErrorAttributes`：定义错误页面中可以包含数据（异常明细，堆栈信息等）。
- 容器中的组件：类型： `BasicExceptionHandler` --> id: `basicExceptionHandler` (json+白页 适配响应)
- 处理默认 `/error` 路径的请求，页面响应 `new ModelAndView("error", model);`
  - 容器中有组件 `View` --> id是error; （响应默认错误页）
  - 容器中放组件 `BeanNameViewResolver` （视图解析器）；按照返回的视图名作为组件的id去容器中找 `View` 对象。

- 容器中的组件：类型： `DefaultErrorViewResolver` -> id: `conventionErrorViewResolver`
- 如果发生异常错误，会以HTTP的状态码 作为视图页地址（viewName），找到真正的页面（主要作用）。
  - error/404、5xx.html
  - 如果想要返回页面，就会找error视图（ `StaticView` 默认是一个白页）。

## 54、错误处理-【源码流程】异常处理流程

譬如写一个会抛出异常的控制层：

```
@Slf4j
@RestController
public class HelloController {

    @RequestMapping("/hello")
    public String handle01(){

        int i = 1 / 0; //将会抛出ArithmeticException

        log.info("Hello, Spring Boot 2!");
        return "Hello, Spring Boot 2!";
    }
}
```

当浏览器发出 `/hello` 请求， `DispatcherServlet` 的 `doDispatch()` 的 `mv = ha.handle(processedRequest, response, mappedHandler.getHandler());` 将会抛出 `ArithmeticException`。

```
public class DispatcherServlet extends FrameworkServlet {
    ...
    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
        ...
        // Actually invoke the handler.
        //将会抛出ArithmeticException
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

        applyDefaultViewName(processedRequest, mv);
        mappedHandler.applyPostHandle(processedRequest, response, mv);
    }
    catch (Exception ex) {
        //将会捕捉ArithmeticException
        dispatchException = ex;
    }
    catch (Throwable err) {
        ...
    }
    //捕捉后，继续运行
    processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
}
    catch (Exception ex) {
        triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
    }
}
```

```

    }
    catch (Throwable err) {
        triggerAfterCompletion(processedRequest, response, mappedHandler,
            new NestedServletException("Handler processing failed", err));
    }
    finally {
        ...
    }
}

private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
    @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,
    @Nullable Exception exception) throws Exception {

    boolean errorView = false;

    if (exception != null) {
        if (exception instanceof ModelAndViewDefiningException) {
            ...
        }
        else {
            Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);
            //ArithmeticException将在这处理
            mv = processHandlerException(request, response, handler, exception);
            errorView = (mv != null);
        }
    }
    ...
}

protected ModelAndView processHandlerException(HttpServletRequest request,
    HttpServletResponse response,
    @Nullable Object handler, Exception ex) throws Exception {

    // Success and error responses may use different content types
    request.removeAttribute(HandlerMapping.PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE);

    // Check registered HandlerExceptionResolvers...
    ModelAndView exMv = null;
    if (this.handlerExceptionResolvers != null) {
        //遍历所有的 handlerExceptionResolvers, 看谁能处理当前异常HandlerExceptionResolver处理
        //器异常解析器
        for (HandlerExceptionResolver resolver : this.handlerExceptionResolvers) {
            exMv = resolver.resolveException(request, response, handler, ex);
            if (exMv != null) {
                break;
            }
        }
    }
    ...

    //若只有系统的自带的异常解析器（没有自定义的），异常还是会抛出

    throw ex;
}

```

```
}  
  
}
```

系统自带的异常解析器：

```
▼ this.handlerExceptionResolvers = {ArrayList@6458} size = 2  
  ▼ 0 = {DefaultErrorAttributes@6777}  
    f includeException = null  
  ▼ 1 = {HandlerExceptionResolverComposite@6778}  
    ▼ f resolvers = {ArrayList@6784} size = 3  
      > 0 = {ExceptionHandlerExceptionResolver@6786}  
      > 1 = {ResponseStatusExceptionHandler@6787}  
      > 2 = {DefaultHandlerExceptionResolver@6788}  
      f order = 0
```

<https://blog.csdn.net/u011863024>

- `DefaultErrorAttributes` 先来处理异常，它主要功能把异常信息保存到request域，并且返回null。

```
public class DefaultErrorAttributes implements ErrorAttributes, HandlerExceptionResolver,  
Ordered {  
    ...  
    public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse  
response, Object handler, Exception ex) {  
        this.storeErrorAttributes(request, ex);  
        return null;  
    }  
  
    private void storeErrorAttributes(HttpServletRequest request, Exception ex) {  
        request.setAttribute(ERROR_ATTRIBUTE, ex); //把异常信息保存到request域  
    }  
    ...  
}
```

- 默认没有任何解析器（上图的 `HandlerExceptionResolverComposite`）能处理异常，所以最后异常会被抛出。
- 最终底层就会转发 `/error` 请求。会被底层的 `BasicErrorController` 处理。

```
@Controller  
@RequestMapping("${server.error.path:${error.path:/error}}")  
public class BasicErrorController extends AbstractErrorController {  
  
    @RequestMapping(produces = MediaType.TEXT_HTML_VALUE)  
    public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response) {  
  
        HttpStatus status = getStatus(request);
```

```

        Map<String, Object> model = Collections
            .unmodifiableMap(getErrorAttributes(request, getErrorAttributeOptions(request,
MediaType.TEXT_HTML)));
        response.setStatus(status.value());
        ModelAndView modelAndView = resolveErrorView(request, response, status, model);
        //如果/template/error内没有4**.html或5**.html,
        //modelAndView为空, 最终还是返回viewName为error的modelAndView
        return (modelAndView != null) ? modelAndView : new ModelAndView("error", model);
    }

    ...
}

```

```

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

    ...

    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
        ...
        // Actually invoke the handler.
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
        ...
        //渲染页面
        processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
        ...
    }

    private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
        @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,
        @Nullable Exception exception) throws Exception {

        boolean errorView = false;
        ...
        // Did the handler return a view to render?
        if (mv != null && !mv.wasCleared()) {
            render(mv, request, response);
            if (errorView) {
                WebUtils.clearErrorRequestAttributes(request);
            }
        }
        ...
    }

    protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse
response) throws Exception {
        ...

        View view;

```

```

String viewName = mv.getViewName();
if (viewName != null) {
    // We need to resolve the view name.
    //找出合适error的View, 如果/template/error内没有4**.html或5**.html,
    //将会返回默认异常页面ErrorMvcAutoConfiguration.StaticView
    //这里按需深究代码吧!
    view = resolveViewName(viewName, mv.getModelInternal(), locale, request);
    ...
}
...
try {
    if (mv.getStatus() != null) {
        response.setStatus(mv.getStatus().value());
    }
    //看下面代码块的StaticView的render块
    view.render(mv.getModelInternal(), request, response);
}
catch (Exception ex) {
    ...
}
}
}

```

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class })
// Load before the main WebMvcAutoConfiguration so that the error View is available
@AutoConfigureBefore(WebMvcAutoConfiguration.class)
@EnableConfigurationProperties({ ServerProperties.class, ResourceProperties.class,
WebMvcProperties.class })
public class ErrorMvcAutoConfiguration {

    ...

    @Configuration(proxyBeanMethods = false)
    @ConditionalOnProperty(prefix = "server.error.whitelabel", name = "enabled", matchIfMissing
= true)
    @Conditional(ErrorTemplateMissingCondition.class)
    protected static class WhitelabelErrorViewConfiguration {

        //将创建一个名为error的系统默认异常页面View的Bean
        private final StaticView defaultErrorView = new StaticView();

        @Bean(name = "error")
        @ConditionalOnMissingBean(name = "error")
        public View defaultErrorView() {
            return this.defaultErrorView;
        }

        // If the user adds @EnableWebMvc then the bean name view resolver from

```

```

// WebMvcAutoConfiguration disappears, so add it back in to avoid disappointment.
@Bean
@ConditionalOnMissingBean
public BeanNameViewResolver beanNameViewResolver() {
    BeanNameViewResolver resolver = new BeanNameViewResolver();
    resolver.setOrder(Ordered.LOWEST_PRECEDENCE - 10);
    return resolver;
}

}

private static class StaticView implements View {

    private static final MediaType TEXT_HTML_UTF8 = new MediaType("text", "html",
StandardCharsets.UTF_8);

    private static final Log logger = LoggerFactory.getLog(StaticView.class);

    @Override
    public void render(Map<String, ?> model, HttpServletRequest request,
HttpServletRequest response)
        throws Exception {
        if (response.isCommitted()) {
            String message = getMessage(model);
            logger.error(message);
            return;
        }
        response.setContentType(TEXT_HTML_UTF8.toString());
        StringBuilder builder = new StringBuilder();
        Object timestamp = model.get("timestamp");
        Object message = model.get("message");
        Object trace = model.get("trace");
        if (response.getContentType() == null) {
            response.setContentType(getContentType());
        }
        //系统默认异常页面html代码
        builder.append("<html><body><h1>Whitelabel Error Page</h1>").append(
            "<p>This application has no explicit mapping for /error, so you are seeing
this as a fallback.</p>")
            .append("<div id='created'>").append(timestamp).append("</div>")
            .append("<div>There was an unexpected error
(type=)").append(htmlEscape(model.get("error")))
            .append(", status=").append(htmlEscape(model.get("status"))).append(").
</div>");
        if (message != null) {
            builder.append("<div>").append(htmlEscape(message)).append("</div>");
        }
        if (trace != null) {
            builder.append("<div style='white-space:pre-
wrap;'>").append(htmlEscape(trace)).append("</div>");
        }

        builder.append("</body></html>");
    }
}

```



```

        response.getWriter().append(builder.toString());
    }

    private String htmlEscape(Object input) {
        return (input != null) ? HtmlUtils.htmlEscape(input.toString()) : null;
    }

    private String getMessage(Map<String, ?> model) {
        Object path = model.get("path");
        String message = "Cannot render error page for request [" + path + "]";
        if (model.get("message") != null) {
            message += " and exception [" + model.get("message") + "]";
        }
        message += " as the response has already been committed.";
        message += " As a result, the response may have the wrong status code.";
        return message;
    }

    @Override
    public String getContentType() {
        return "text/html";
    }
}

```

## 55、错误处理-【源码流程】几种异常处理原理

- 自定义错误页
  - error/404.html error/5xx.html; 有精确的错误状态码页面就匹配精确, 没有就找 4xx.html; 如果都没有就触发白页
- `@ControllerAdvice` + `@ExceptionHandler` 处理全局异常; 底层是 `ExceptionHandlerExceptionResolver` 支持的

```

@Slf4j
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler({ArithmeticException.class, NullPointerException.class}) //处理异常
    public String handleArithException(Exception e){

        log.error("异常是: {}", e);
        return "login"; //视图地址
    }
}

```

- `@ResponseStatus` + 自定义异常; 底层是 `ResponseStatusExceptionResolver`, 把 `ResponseStatus` 注解的信息底层调用 `response.sendError(statusCode, resolvedReason)`, tomcat 发送的 `/error`

```

@ResponseStatus(value= HttpStatus.FORBIDDEN,reason = "用户数量太多")
public class UserTooManyException extends RuntimeException {

    public UserTooManyException(){

    }

    public UserTooManyException(String message){
        super(message);
    }
}

```

```

@Controller
public class TableController {

    @GetMapping("/dynamic_table")
    public String dynamic_table(@RequestParam(value="pn",defaultValue = "1") Integer pn,Model
model){
        //表格内容的遍历
        List<User> users = Arrays.asList(new User("zhangsan", "123456"),
            new User("lisi", "123444"),
            new User("haha", "aaaaa"),
            new User("hehe ", "aaddd"));
        model.addAttribute("users",users);

        if(users.size()>3){
            throw new UserTooManyException();//抛出自定义异常
        }
        return "table/dynamic_table";
    }
}

```

- Spring自家异常如 `org.springframework.web.bind.MissingServletRequestParameterException` , `DefaultHandlerExceptionResolver` 处理Spring自家异常。
- ○ `response.sendError(HttpServletResponse.SC_BAD_REQUEST/*400*/, ex.getMessage());`
- 自定义实现 `HandlerExceptionResolver` 处理异常；可以作为默认的全局异常处理规则

```

@Order(value= Ordered.HIGHEST_PRECEDENCE) //优先级，数字越小优先级越高
@Component
public class CustomerHandlerExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest request,
        HttpServletResponse response,
        Object handler, Exception ex) {

        try {
            response.sendError(511,"我喜欢的错误");
        } catch (IOException e) {
            e.printStackTrace();
        }
        return new ModelAndView();
    }
}

```

```
}  
}
```

- `ErrorViewResolver` 实现自定义处理异常
  - `response.sendError()`, error请求就会转给controller。
  - 你的异常没有任何人能处理, tomcat底层调用 `response.sendError()`, error请求就会转给controller。
  - `basicErrorController` 要去的页面地址是 `ErrorViewResolver` 。

```
@Controller  
@RequestMapping("${server.error.path:${error.path:/error}}")  
public class BasicErrorController extends AbstractErrorController {  
  
    ...  
  
    @RequestMapping(produces = MediaType.TEXT_HTML_VALUE)  
    public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response) {  
        HttpStatus status = getStatus(request);  
        Map<String, Object> model = Collections  
            .unmodifiableMap(getErrorAttributes(request, getErrorAttributeOptions(request,  
MediaType.TEXT_HTML)));  
        response.setStatus(status.value());  
        ModelAndView modelAndView = resolveErrorView(request, response, status, model);  
        return (modelAndView != null) ? modelAndView : new ModelAndView("error", model);  
    }  
  
    protected ModelAndView resolveErrorView(HttpServletRequest request, HttpServletResponse  
response, HttpStatus status,  
        Map<String, Object> model) {  
        //这里用到ErrorViewResolver接口  
        for (ErrorViewResolver resolver : this.errorViewResolvers) {  
            ModelAndView modelAndView = resolver.resolveErrorView(request, status, model);  
            if (modelAndView != null) {  
                return modelAndView;  
            }  
        }  
        return null;  
    }  
  
    ...  
  
}
```

```

@FunctionalInterface
public interface ErrorViewResolver {

    ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String,
    Object> model);

}

```

## 56、原生组件注入-原生注解与Spring方式注入

[官方文档 - Servlets, Filters, and listeners](#)

### 使用原生的注解

```

@WebServlet(urlPatterns = "/my")
public class MyServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        resp.getWriter().write("66666");
    }
}

```

```

@Slf4j
@WebServlet(urlPatterns={"/css/*","/images/*"}) //my
public class MyFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        log.info("MyFilter初始化完成");
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
        log.info("MyFilter工作");
        chain.doFilter(request,response);
    }

    @Override
    public void destroy() {
        log.info("MyFilter销毁");
    }
}

```

```

@Slf4j
@WebListener

public class MyServletContextListener implements ServletContextListener {

```

```

@Override
public void contextInitialized(ServletContextEvent sce) {
    log.info("MySwervletContextListener监听到项目初始化完成");
}

@Override
public void contextDestroyed(ServletContextEvent sce) {
    log.info("MySwervletContextListener监听到项目销毁");
}
}

```

最后还要在主启动类添加注解 `@ServletComponentScan`

```

@WebServletComponentScan(basePackages = "com.lun")//
@SpringBootApplication(exclude = RedisAutoConfiguration.class)
public class Boot05WebAdminApplication {

    public static void main(String[] args) {
        SpringApplication.run(Boot05WebAdminApplication.class, args);
    }
}

```

## Spring方式注入

`ServletRegistrationBean`, `FilterRegistrationBean`, and `ServletListenerRegistrationBean`

```

@Configuration(proxyBeanMethods = true)
public class MyRegistConfig {

    @Bean
    public ServletRegistrationBean myServlet(){
        MyServlet myServlet = new MyServlet();

        return new ServletRegistrationBean(myServlet, "/my", "/my02");
    }

    @Bean
    public FilterRegistrationBean myFilter(){

        MyFilter myFilter = new MyFilter();
        // return new FilterRegistrationBean(myFilter, myServlet());
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(myFilter);
        filterRegistrationBean.setUrlPatterns(Arrays.asList("/my", "/css/*"));
        return filterRegistrationBean;
    }

    @Bean

```

```

    public ServletListenerRegistrationBean myListener(){
        MySwervletContextListener mySwervletContextListener = new MySwervletContextListener();
        return new ServletListenerRegistrationBean(mySwervletContextListener);
    }
}

```

## 57、原生组件注入-【源码分析】DispatcherServlet注入原理

org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration 配置类

```

@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass(DispatcherServlet.class)
@AutoConfigureAfter(ServletWebServerFactoryAutoConfiguration.class)
public class DispatcherServletAutoConfiguration {

    /**
     * The bean name for a DispatcherServlet that will be mapped to the root URL "/"
     */
    public static final String DEFAULT_DISPATCHER_SERVLET_BEAN_NAME = "dispatcherServlet";

    /**
     * The bean name for a ServletRegistrationBean for the DispatcherServlet "/"
     */
    public static final String DEFAULT_DISPATCHER_SERVLET_REGISTRATION_BEAN_NAME =
"dispatcherServletRegistration";

    @Configuration(proxyBeanMethods = false)
    @Conditional(DefaultDispatcherServletCondition.class)
    @ConditionalOnClass(ServletRegistration.class)
    @EnableConfigurationProperties(WebMvcProperties.class)
    protected static class DispatcherServletConfiguration {

        //创建DispatcherServlet类的Bean
        @Bean(name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
        public DispatcherServlet dispatcherServlet(WebMvcProperties webMvcProperties) {
            DispatcherServlet dispatcherServlet = new DispatcherServlet();

            dispatcherServlet.setDispatchOptionsRequest(webMvcProperties.isDispatchOptionsRequest());

            dispatcherServlet.setDispatchTraceRequest(webMvcProperties.isDispatchTraceRequest());

            dispatcherServlet.setThrowExceptionIfNoHandlerFound(webMvcProperties.isThrowExceptionIfNoHandlerFound());

            dispatcherServlet.setPublishEvents(webMvcProperties.isPublishRequestHandledEvents());

            dispatcherServlet.setEnableLoggingRequestDetails(webMvcProperties.isLogRequestDetails());

            return dispatcherServlet;
        }
    }
}

```

```

    }

    @Bean
    @ConditionalOnBean(MultipartResolver.class)
    @ConditionalOnMissingBean(name = DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)
    public MultipartResolver multipartResolver(MultipartResolver resolver) {
        // Detect if the user has created a MultipartResolver but named it incorrectly
        return resolver;
    }

}

@Configuration(proxyBeanMethods = false)
@Conditional(DispatcherServletRegistrationCondition.class)
@ConditionalOnClass(ServletRegistration.class)
@EnableConfigurationProperties(WebMvcProperties.class)
@Import(DispatcherServletConfiguration.class)
protected static class DispatcherServletRegistrationConfiguration {

    //注册DispatcherServlet类
    @Bean(name = DEFAULT_DISPATCHER_SERVLET_REGISTRATION_BEAN_NAME)
    @ConditionalOnBean(value = DispatcherServlet.class, name =
DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
    public DispatcherServletRegistrationBean
dispatcherServletRegistration(DispatcherServlet dispatcherServlet,
        WebMvcProperties webMvcProperties, ObjectProvider<MultipartConfigElement>
multipartConfig) {
        DispatcherServletRegistrationBean registration = new
DispatcherServletRegistrationBean(dispatcherServlet,
            webMvcProperties.getServlet().getPath());
        registration.setName(DEFAULT_DISPATCHER_SERVLET_BEAN_NAME);
        registration.setLoadOnStartup(webMvcProperties.getServlet().getLoadOnStartup());
        multipartConfig.ifAvailable(registration::setMultipartConfig);
        return registration;
    }

}

...

}

```

DispatcherServlet 默认映射的是 / 路径，可以通过在配置文件修改 `spring.mvc.servlet.path=/mvc`。

## 58、嵌入式Servlet容器-【源码分析】切换web服务器与定制化

- 默认支持的WebServer
  - Tomcat, Jetty, Or Undertow。
  - ServletWebServerApplicationContext 容器启动寻找 ServletWebServerFactory 并引导创建服务器。
- 原理

- SpringBoot应用启动发现当前是Web应用，web场景包-导入tomcat。
- web应用会创建一个web版的IOC容器 `ServletWebServerApplicationContext`。
- `ServletWebServerApplicationContext` 启动的时候寻找 `ServletWebServerFactory`（Servlet 的web服务器工厂——>Servlet 的web服务器）。
- SpringBoot底层默认有很多的WebServer工厂（`ServletWebServerFactoryConfiguration` 内创建Bean），如：
  - `TomcatServletWebServerFactory`
  - `JettyServletWebServerFactory`
  - `UndertowServletWebServerFactory`
- 底层直接会有一个自动配置类 `ServletWebServerFactoryAutoConfiguration`。
- `ServletWebServerFactoryAutoConfiguration` 导入了 `ServletWebServerFactoryConfiguration`（配置类）。
- `ServletWebServerFactoryConfiguration` 根据动态判断系统中到底导入了那个Web服务器的包。（默认是web-starter导入tomcat包），容器中就有 `TomcatServletWebServerFactory`
- `TomcatServletWebServerFactory` 创建出Tomcat服务器并启动；`TomcatWebServer` 的构造器拥有初始化方法initialize——`this.tomcat.start();`
- 内嵌服务器，与以前手动把启动服务器相比，改成现在使用代码启动（tomcat核心jar包存在）。

Spring Boot默认使用Tomcat服务器，若需更改其他服务器，则修改工程pom.xml：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

[官方文档 - Use Another Web Server](#)

## 定制Servlet容器

- 实现 `WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>`
  - 把配置文件的值和 `ServletWebServerFactory` 进行绑定
- 修改配置文件 `server.xxx`
- 直接自定义 `ConfigurableServletWebServerFactory`

`xxxxxCustomizer`：定制化工器，可以改变xxxx的默认规则



```
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory server) {
        server.setPort(9000);
    }

}
```

## 59、定制化原理-SpringBoot定制化组件的几种方式（小结）

### 定制化的常见方式

- 修改配置文件
- xxxxxCustomizer
- 编写自定义的配置类 xxxConfiguration + @Bean 替换、增加容器中默认组件，视图解析器
- Web应用 编写一个配置类实现 WebMvcConfigurer 即可定制化web功能 + @Bean 给容器中再扩展一些组件

```
@Configuration
public class AdminWebConfig implements WebMvcConfigurer{
}
```

- @EnableWebMvc + WebMvcConfigurer — @Bean 可以全面接管SpringMVC，所有规则全部自己重新配置；实现定制和扩展功能（高级功能，初学者退避三舍）。
  - 原理：
    1. WebMvcAutoConfiguration 默认的SpringMVC的自动配置功能类，如静态资源、欢迎页等。
    2. 一旦使用 @EnableWebMvc，会 @Import(DelegatingWebMvcConfiguration.class)。
    3. DelegatingWebMvcConfiguration 的作用，只保证SpringMVC最基本的使用
      - 把所有系统中的 WebMvcConfigurer 拿过来，所有功能的定制都是这些 WebMvcConfigurer 合起来一起生效。
      - 自动配置了一些非常底层的组件，如 RequestMappingHandlerMapping，这些组件依赖的组件都是从容器中获取如。
      - public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport。
    4. WebMvcAutoConfiguration 里面的配置要能生效必须 @ConditionalOnMissingBean(WebMvcConfigurationSupport.class)。
    5. @EnableWebMvc 导致了WebMvcAutoConfiguration 没有生效。

### 原理分析套路

场景starter - `xxxxAutoConfiguration` - 导入xxx组件 - 绑定 `xxxProperties` - 绑定配置文件项。

## 60、数据访问-数据库场景的自动配置分析与整合测试

### 导入JDBC场景

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

接着导入数据库驱动包（MySQL为例）。

```
<!--默认版本：-->
<mysql.version>8.0.22</mysql.version>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <!--<version>5.1.49</version>-->
</dependency>

<!--
想要修改版本
1、直接依赖引入具体版本（maven的就近依赖原则）
2、重新声明版本（maven的属性的就近优先原则）
-->
<properties>
  <java.version>1.8</java.version>
  <mysql.version>5.1.49</mysql.version>
</properties>
```

### 相关数据源配置类

- `DataSourceAutoConfiguration`：数据源的自动配置。
  - 修改数据源相关的配置：`spring.datasource`。
  - 数据库连接池的配置，是自己容器中没有`DataSource`才自动配置的。
  - 底层配置好的连接池是：`HikariDataSource`。
- `DataSourceTransactionManagerAutoConfiguration`：事务管理器的自动配置。
- `JdbcTemplateAutoConfiguration`：`JdbcTemplate`的自动配置，可以对数据库进行CRUD。
  - 可以修改前缀为 `spring.jdbc` 的配置项来修改 `JdbcTemplate`。
  - `@Bean @Primary JdbcTemplate`：Spring容器中有这个 `JdbcTemplate` 组件，使用 `@Autowired`。
- `JndiDataSourceAutoConfiguration`：JNDI的自动配置。
- `XADataSourceAutoConfiguration`：分布式事务相关的。

## 修改配置项

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/db_account
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
```

## 单元测试数据源

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.jdbc.core.JdbcTemplate;

@SpringBootTest
class Boot05WebAdminApplicationTests {

    @Autowired
    JdbcTemplate jdbcTemplate;

    @Test//用org.junit.Test会报空指针异常，可能跟JUnit新版本有关
    void contextLoads() {
        //      jdbcTemplate.queryForObject("select * from account_tbl")
        //      jdbcTemplate.queryForList("select * from account_tbl",)
        Long aLong = jdbcTemplate.queryForObject("select count(*) from account_tbl",
Long.class);
        log.info("记录总数: {}",aLong);
    }

}
```

## 61、数据访问-自定义方式整合druid数据源

[Druid官网](#)

### Druid是什么？

它是数据库连接池，它能够提供强大的监控和扩展功能。

[官方文档 - Druid连接池介绍](#)

Spring Boot整合第三方技术的两种方式：

- 自定义

- 找starter场景

## 自定义方式

添加依赖：

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.17</version>
</dependency>
```

配置Druid数据源：

```
@Configuration
public class MyConfig {

    @Bean
    @ConfigurationProperties("spring.datasource")//复用配置文件的数据源配置
    public DataSource dataSource() throws SQLException {
        DruidDataSource druidDataSource = new DruidDataSource();

        //      druidDataSource.setUrl();
        //      druidDataSource.setUsername();
        //      druidDataSource.setPassword();

        return druidDataSource;
    }
}
```

### [更多配置项](#)

配置Druid的监控页功能：

- Druid内置提供了一个 `StatViewServlet` 用于展示Druid的统计信息。 [官方文档 - 配置 StatViewServlet配置](#)。这个 `StatViewServlet` 的用途包括：
  - 提供监控信息展示的html页面
  - 提供监控信息的JSON API
- Druid内置提供一个 `StatFilter` ，用于统计监控信息。 [官方文档 - 配置 StatFilter](#)
- `WebStatFilter` 用于采集web-jdbc关联监控的数据，如SQL监控、URI监控。 [官方文档 - 配置 配置 WebStatFilter](#)
- Druid提供了 `WallFilter` ，它是基于SQL语义分析来实现防御SQL注入攻击的。 [官方文档 - 配置 wallfilter](#)

```
@Configuration
public class MyConfig {

    @Bean
    @ConfigurationProperties("spring.datasource")
    public DataSource dataSource() throws SQLException {

        DruidDataSource druidDataSource = new DruidDataSource();
```

```

        //加入监控和防火墙功能功能
        druidDataSource.setFilters("stat,wall");

        return druidDataSource;
    }

    /**
     * 配置 druid的监控页功能
     * @return
     */
    @Bean
    public ServletRegistrationBean statViewServlet(){
        StatViewServlet statViewServlet = new StatViewServlet();
        ServletRegistrationBean<StatViewServlet> registrationBean =
            new ServletRegistrationBean<>(statViewServlet, "/druid/*");

        //监控页账号密码:
        registrationBean.addInitParameter("loginUsername","admin");
        registrationBean.addInitParameter("loginPassword","123456");

        return registrationBean;
    }

    /**
     * WebStatFilter 用于采集web-jdbc关联监控的数据。
     */
    @Bean
    public FilterRegistrationBean webStatFilter(){
        WebStatFilter webStatFilter = new WebStatFilter();

        FilterRegistrationBean<WebStatFilter> filterRegistrationBean = new
        FilterRegistrationBean<>(webStatFilter);
        filterRegistrationBean.setUrlPatterns(Arrays.asList("/*"));

        filterRegistrationBean.addInitParameter("exclusions","*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*");

        return filterRegistrationBean;
    }
}

```

## 62、数据访问-druid数据源starter整合方式

[官方文档 - Druid Spring Boot Starter](#)

引入依赖:

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid-spring-boot-starter</artifactId>
  <version>1.1.17</version>
</dependency>
```

### 分析自动配置:

- 扩展配置项 `spring.datasource.druid`
- 自动配置类 `DruidDataSourceAutoConfigure`
- `DruidSpringAopConfiguration.class`, 监控SpringBean的; 配置项: `spring.datasource.druid.aop-patterns`
- `DruidStatViewServletConfiguration.class`, 监控页的配置。 `spring.datasource.druid.stat-view-servlet` 默认开启。
- `DruidWebStatFilterConfiguration.class`, web监控配置。 `spring.datasource.druid.web-stat-filter` 默认开启。
- `DruidFilterConfiguration.class` 所有Druid的filter的配置:

```
private static final String FILTER_STAT_PREFIX = "spring.datasource.druid.filter.stat";
private static final String FILTER_CONFIG_PREFIX = "spring.datasource.druid.filter.config";
private static final String FILTER_ENCODING_PREFIX = "spring.datasource.druid.filter.encoding";
private static final String FILTER_SLF4J_PREFIX = "spring.datasource.druid.filter.slf4j";
private static final String FILTER_LOG4J_PREFIX = "spring.datasource.druid.filter.log4j";
private static final String FILTER_LOG4J2_PREFIX = "spring.datasource.druid.filter.log4j2";
private static final String FILTER_COMMONS_LOG_PREFIX = "spring.datasource.druid.filter.common-log";
private static final String FILTER_WALL_PREFIX = "spring.datasource.druid.filter.wall";
```

### 配置示例:

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/db_account
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver

  druid:
    aop-patterns: com.atguigu.admin.* #监控SpringBean
    filters: stat,wall # 底层开启功能, stat (sql监控), wall (防火墙)

    stat-view-servlet: # 配置监控页功能
      enabled: true
      login-username: admin
      login-password: admin
      resetEnable: false

    web-stat-filter: # 监控web
      enabled: true
      urlPattern: /*

      exclusions: '*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*'
```

```

filter:
  stat:      # 对上面filters里面的stat的详细配置
    slow-sql-millis: 1000
    logSlowSql: true
    enabled: true
  wall:
    enabled: true
  config:
    drop-table-allow: false

```

## 63、数据访问-整合MyBatis-配置版

[MyBatis的GitHub仓库](#)

[MyBatis官方](#)

**starter的命名方式：**

1. SpringBoot官方的Starter: `spring-boot-starter-*`
2. 第三方的: `*-spring-boot-starter`

**引入依赖：**

```

<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.1.4</version>
</dependency>

```

**配置模式：**

- 全局配置文件
- `SqlSessionFactory`：自动配置好了
- `SqlSession`：自动配置了 `SqlSessionTemplate` 组合了 `SqlSession`
- `@Import(AutoConfiguredMapperScannerRegistrar.class)`
- `Mapper`：只要我们写的操作MyBatis的接口标准了 `@Mapper` 就会被自动扫描进来

```

@EnableConfigurationProperties(MybatisProperties.class) : MyBatis配置项绑定类。
@AutoConfigureAfter({ DataSourceAutoConfiguration.class,
MybatisLanguageDriverAutoConfiguration.class })
public class MybatisAutoConfiguration{
    ...
}

@ConfigurationProperties(prefix = "mybatis")
public class MybatisProperties{
    ...
}

```

## 配置文件:

```
spring:
  datasource:
    username: root
    password: 1234
    url: jdbc:mysql://localhost:3306/my
    driver-class-name: com.mysql.jdbc.Driver

# 配置mybatis规则
mybatis:
  config-location: classpath:mybatis/mybatis-config.xml #全局配置文件位置
  mapper-locations: classpath:mybatis/*.xml #sql映射文件位置
```

## mybatis-config.xml:

```
<?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>

    <!-- 由于Spring Boot自动配置缘故，此处不必配置，只用来做做样。 -->
</configuration>
```

## Mapper接口:

```
<?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="com.lun.boot.mapper.UserMapper">

    <select id="getUser" resultType="com.lun.boot.bean.User">
        select * from user where id=#{id}
    </select>
</mapper>
```

```
import com.lun.boot.bean.User;
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface UserMapper {
    public User getUser(Integer id);
}
```



## POJO:

```
public class User {  
    private Integer id;  
    private String name;  
  
    //getters and setters...  
}
```

## DB:

```
CREATE TABLE `user` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `name` varchar(45) DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4;
```

## Controller and Service:

```
@Controller  
public class UserController {  
  
    @Autowired  
    private UserService userService;  
  
    @ResponseBody  
    @GetMapping("/user/{id}")  
    public User getUser(@PathVariable("id") Integer id){  
  
        return userService.getUser(id);  
    }  
}
```

```
@Service  
public class UserService {  
  
    @Autowired  
    private UserMapper userMapper; //IDEA下标红线，可忽视这红线  
  
    public User getUser(Integer id){  
        return userMapper.getUser(id);  
    }  
}
```

配置 `private Configuration configuration;` 也就是配置 `mybatis.configuration` 相关的，就是相当于改 mybatis 全局配置文件中的值。（也就是说配置了 `mybatis.configuration`，就不需配置 mybatis 全局配置文件了）

```
# 配置mybatis规则
mybatis:
  mapper-locations: classpath:mybatis/mapper/*.xml
  # 可以不写全局配置文件，所有全局配置文件的配置都放在configuration配置项中了。
  # config-location: classpath:mybatis/mybatis-config.xml
  configuration:
    map-underscore-to-camel-case: true
```

## 小结

- 导入MyBatis官方Starter。
- 编写Mapper接口，需 `@Mapper` 注解。
- 编写SQL映射文件并绑定Mapper接口。
- 在 `application.yaml` 中指定Mapper配置文件的所处位置，以及指定全局配置文件的信息（建议：配置在 `mybatis.configuration`）。

## 64、数据访问-整合MyBatis-注解配置混合版

你可以通过Spring Initializr添加MyBatis的Starter。

注解与配置混合搭配，干活不累：

```
@Mapper
public interface UserMapper {
    public User getUser(Integer id);

    @Select("select * from user where id=#{id}")
    public User getUser2(Integer id);

    public void saveUser(User user);

    @Insert("insert into user(`name`) values(#{name})")
    @Options(useGeneratedKeys = true, keyProperty = "id")
    public void saveUser2(User user);
}
```

```
<?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="com.lun.boot.mapper.UserMapper">

    <select id="getUser" resultType="com.lun.boot.bean.User">
        select * from user where id=#{id}
    </select>

    <insert id="saveUser" useGeneratedKeys="true" keyProperty="id">
        insert into user(`name`) values(#{name})
    </insert>

</mapper>

```

- 简单DAO方法就写在注解上。复杂的就写在配置文件里。
- 使用 `@MapperScan("com.lun.boot.mapper")` 简化，Mapper接口就可以不用标注 `@Mapper` 注解。

```

@MapperScan("com.lun.boot.mapper")
@SpringBootApplication
public class MainApplication {

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

}

```

## 65、数据访问-整合MyBatisPlus操作数据库

[IDEA的MyBatis的插件 - MyBatisX](#)

[MyBatisPlus官网](#)

[MyBatisPlus官方文档](#)

### MyBatisPlus是什么

[MyBatis-Plus](#) (简称 MP) 是一个 [MyBatis](#) 的增强工具，在 MyBatis 的基础上只做增强不做改变，为简化开发、提高效率而生。

添加依赖：

```

<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.1</version>
</dependency>

```

- `MybatisPlusAutoConfiguration` 配置类, `MybatisPlusProperties` 配置项绑定。
- `SqlSessionFactory` 自动配置好, 底层是容器中默认的数据源。
- `mapperLocations` 自动配置好的, 有默认值 `classpath*:mapper/**/*.xml`, 这表示任意包的类路径下的所有mapper文件夹下任意路径下的所有xml都是sql映射文件。 建议以后sql映射文件放在 mapper下。
- 容器中也自动配置好了 `SqlSessionTemplate`。
- `@Mapper` 标注的接口也会被自动扫描, 建议直接 `@MapperScan("com.lun.boot.mapper")` 批量扫描。
- MyBatisPlus**优点之一**: 只需要我们的Mapper继承MyBatisPlus的 `BaseMapper` 就可以拥有CRUD能力, 减轻开发工作。

```
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.lun.hellomybatisplus.model.User;

public interface UserMapper extends BaseMapper<User> {

}
```

## 66、数据访问-CRUD实验-数据列表展示

### [官方文档 - CRUD接口](#)

使用MyBatis Plus提供的 `IService`, `ServiceImpl`, 减轻Service层开发工作。

```
import com.lun.hellomybatisplus.model.User;
import com.baomidou.mybatisplus.extension.service.IService;

import java.util.List;

/**
 * Service 的CRUD也不用写了
 */
public interface UserService extends IService<User> {
    //此处故意为空
}
```

```
import com.lun.hellomybatisplus.model.User;
import com.lun.hellomybatisplus.mapper.UserMapper;
import com.lun.hellomybatisplus.service.UserService;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class UserServiceImpl extends ServiceImpl<UserMapper,User> implements UserService {
    //此处故意为空
}
```

与下一节联合在一起

## 67、数据访问-CRUD实验-分页数据展示

与下一节联合在一起

## 68、数据访问-CRUD实验-删除用户完成

添加分页插件：

```
@Configuration
public class MyBatisConfig {

    /**
     * MybatisPlusInterceptor
     * @return
     */
    @Bean
    public MybatisPlusInterceptor paginationInterceptor() {
        MybatisPlusInterceptor mybatisPlusInterceptor = new MybatisPlusInterceptor();
        // 设置请求的页面大于最大页后操作， true调回到首页， false 继续请求 默认false
        // paginationInterceptor.setOverflow(false);
        // 设置最大单页限制数量，默认 500 条，-1 不受限制
        // paginationInterceptor.setLimit(500);
        // 开启 count 的 join 优化,只针对部分 left join

        //这是分页拦截器
        PaginationInnerInterceptor paginationInnerInterceptor = new
        PaginationInnerInterceptor();
        paginationInnerInterceptor.setOverflow(true);
        paginationInnerInterceptor.setMaxLimit(500L);
        mybatisPlusInterceptor.addInnerInterceptor(paginationInnerInterceptor);

        return mybatisPlusInterceptor;
    }
}
```

```
<table class="display table table-bordered table-striped" id="dynamic-table">
  <thead>
    <tr>
      <th>#</th>
      <th>name</th>
      <th>age</th>
      <th>email</th>
      <th>操作</th>
    </tr>
  </thead>

  <tbody>
```

```

<tr class="gradeX" th:each="user: ${users.records}">
    <td th:text="${user.id}"></td>
    <td>[[${user.name}]]</td>
    <td th:text="${user.age}">Win 95+</td>
    <td th:text="${user.email}">4</td>
    <td>
        <a th:href="@{/user/delete/{id}(id=${user.id},pn=${users.current})}"
            class="btn btn-danger btn-sm" type="button">删除</a>
    </td>
</tr>
</tfoot>
</table>

<div class="row-fluid">
    <div class="span6">
        <div class="dataTables_info" id="dynamic-table_info">
            当前第[[${users.current}]]页 总计 [[${users.pages}]]页 共[[${users.total}]]条记录
        </div>
    </div>
    <div class="span6">
        <div class="dataTables_paginate paging_bootstrap pagination">
            <ul>
                <li class="prev disabled"><a href="#"><- 前一页</a></li>
                <li th:class="${num == users.current?'active':''}"
                    th:each="num:${#numbers.sequence(1,users.pages)}" >
                    <a th:href="@{/dynamic_table(pn=${num})}">[[${num}]]</a>
                </li>
                <li class="next disabled"><a href="#">下一页 ></a></li>
            </ul>
        </div>
    </div>
</div>

```

`#numbers` 表示methods for formatting numeric objects.[link](#)

```

@GetMapping("/user/delete/{id}")
public String deleteUser(@PathVariable("id") Long id,
    @RequestParam(value = "pn",defaultValue = "1")Integer pn,
    RedirectAttributes ra){

    userService.removeById(id);

    ra.addAttribute("pn",pn);
    return "redirect:/dynamic_table";
}

@GetMapping("/dynamic_table")
public String dynamic_table(@RequestParam(value="pn",defaultValue = "1") Integer pn,Model model)
{
    //表格内容的遍历
}

```

```

//从数据库中查出user表中的用户进行展示

//构造分页参数
Page<User> page = new Page<>(pn, 2);
//调用page进行分页
Page<User> userPage = userService.page(page, null);

model.addAttribute("users",userPage);

return "table/dynamic_table";
}

```

## 69、数据访问-准备阿里云Redis环境

添加依赖:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

<!--导入jedis-->
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
</dependency>

```

- `RedisAutoConfiguration` 自动配置类, `RedisProperties` 属性类 --> `spring.redis.xxx`是对redis的配置。
- 连接工厂 `LettuceConnectionConfiguration`、`JedisConnectionConfiguration` 是准备好的。
- 自动注入了 `RedisTemplate<Object, Object>`, `xxxTemplate`。
- 自动注入了 `StringRedisTemplate`, key, value都是String
- 底层只要我们使用 `StringRedisTemplate`、`RedisTemplate` 就可以操作Redis。

外网Redis环境搭建:

1. 阿里云按量付费Redis, 其中选择**经典网络**。
2. 申请Redis的公网连接地址。
3. 修改白名单, 允许 `0.0.0.0/0` 访问。

## 70、数据访问-Redis操作与统计小实验

相关Redis配置:

```

spring:
  redis:
#   url: redis://lfy:Lfy123456@r-bp1nc7reqesxisgxpipd.redis.rds.aliyuncs.com:6379
    host: r-bp1nc7reqesxisgxpipd.redis.rds.aliyuncs.com
    port: 6379
    password: lfy:Lfy123456
    client-type: jedis

```

```

jedis:
  pool:
    max-active: 10
# lettuce:# 另一个用来连接redis的java框架
#   pool:
#     max-active: 10
#     min-idle: 5

```

测试Redis连接:

```

@SpringBootTest
public class Boot05WebAdminApplicationTests {

    @Autowired
    StringRedisTemplate redisTemplate;

    @Autowired
    RedisConnectionFactory redisConnectionFactory;

    @Test
    void testRedis(){
        ValueOperations<String, String> operations = redisTemplate.opsForValue();

        operations.set("hello", "world");

        String hello = operations.get("hello");
        System.out.println(hello);

        System.out.println(redisConnectionFactory.getClass());
    }
}

```

Redis Desktop Manager: 可视化Redis管理软件。

URL统计拦截器:

```

@Component
public class RedisUrlCountInterceptor implements HandlerInterceptor {

    @Autowired
    StringRedisTemplate redisTemplate;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        String uri = request.getRequestURI();
    }
}

```



```

        //默认每次访问当前uri就会计数+1
        redisTemplate.opsForValue().increment(uri);

        return true;
    }
}

```

注册URL统计拦截器:

```

@Configuration
public class AdminWebConfig implements WebMvcConfigurer{

    @Autowired
    RedisUrlCountInterceptor redisUrlCountInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {

        registry.addInterceptor(redisUrlCountInterceptor)
            .addPathPatterns("/**")
            .excludePathPatterns("/", "/login", "/css/**", "/fonts/**", "/images/**",
                "/js/**", "/aa/**");
    }
}

```

Filter、Interceptor 几乎拥有相同的功能?

- Filter是Servlet定义的原生组件，它的好处是脱离Spring应用也能使用。
- Interceptor是Spring定义的接口，可以使用Spring的自动装配等功能。

调用Redis内的统计数据:

```

@Slf4j
@Controller
public class IndexController {

    @Autowired
    StringRedisTemplate redisTemplate;

    @GetMapping("/main.html")
    public String mainPage(HttpSession session, Model model){

        log.info("当前方法是: {}", "mainPage");

        ValueOperations<String, String> opsForValue =
            redisTemplate.opsForValue();

        String s = opsForValue.get("/main.html");
    }
}

```

```

String s1 = opsForValue.get("/sql");

model.addAttribute("mainCount",s);
model.addAttribute("sqlCount",s1);

return "main";
}
}

```

## 71、单元测试-JUnit5简介

Spring Boot 2.2.0 版本开始引入 JUnit 5 作为单元测试默认库

[JUnit 5官方文档](#)

作为最新版本的JUnit框架，JUnit5与之前版本的JUnit框架有很大的不同。由三个不同子项目的几个不同模块组成。

**JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**

- **JUnit Platform:** JUnit Platform是在JVM上启动测试框架的基础，不仅支持JUnit自制的测试引擎，其他测试引擎也都可以接入。
- **JUnit Jupiter:** JUnit Jupiter提供了JUnit5的新的编程模型，是JUnit5新特性的核心。内部包含了一个**测试引擎**，用于在JUnit Platform上运行。
- **JUnit Vintage:** 由于JUnit已经发展多年，为了照顾老的项目，JUnit Vintage提供了兼容JUnit4.x, JUnit3.x的测试引擎。

**注意：**

- SpringBoot 2.4 以上版本移除了默认对 Vintage 的依赖。如果需要兼容JUnit4需要自行引入（不能使用JUnit4的功能 @Test）
- JUnit 5's Vintage已经从 `spring-boot-starter-test` 从移除。如果需要继续兼容JUnit4需要自行引入Vintage依赖：

```

<dependency>
  <groupId>org.junit.vintage</groupId>
  <artifactId>junit-vintage-engine</artifactId>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-core</artifactId>
    </exclusion>
  </exclusions>
</dependency>

```

- 使用添加JUnit 5，添加对应的starter：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

- Spring的JUnit 5的基本单元测试模板（Spring的JUnit4的是  
`@SpringBootTest` + `@RunWith(SpringRunner.class)`）：

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test; //注意不是org.junit.Test (这是JUnit4版本的)
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class SpringBootApplicationTests {

    @Autowired
    private Component component;

    @Test
    // @Transactional 标注后连接数据库有回滚功能
    public void contextLoads() {
        Assertions.assertEquals(5, component.getFive());
    }
}
```

Jupiter

英 ['dʒu:pɪtə(r)] 美 ['dʒu:pɪtər]

n. 木星(太阳系中最大的行星)

vintage

英 ['vɪntɪdʒ] 美 ['vɪntɪdʒ]

n. 特定年份(或地方)酿制的酒;酿造年份;采摘葡萄酿酒的期间(或季节);葡萄收获期(或季节)

adj. (指葡萄酒)优质的, 上等的, 佳酿的;古色古香的(指1917-1930年间制造, 车型和品味受人青睐的);(过去某个时期)典型的, 优质的;(某人的)最佳作品的

## 72、单元测试-常用测试注解

[官方文档 - Annotations](#)

- **@Test**：表示方法是测试方法。但是与JUnit4的@Test不同，他的职责非常单一不能声明任何属性，拓展的测试将会由Jupiter提供额外测试
- **@ParameterizedTest**：表示方法是参数化测试。
- **@RepeatedTest**：表示方法可重复执行。
- **@DisplayName**：为测试类或者测试方法设置展示名称。

- **@BeforeEach**: 表示在**每个**单元测试**之前**执行。
- **@AfterEach**: 表示在**每个**单元测试**之后**执行。
- **@BeforeAll**: 表示在**所有**单元测试**之前**执行。
- **@AfterAll**: 表示在**所有**单元测试**之后**执行。
- **@Tag**: 表示单元测试类别，类似于JUnit4中的@Categories。
- **@Disabled**: 表示测试类或测试方法不执行，类似于JUnit4中的@Ignore。
- **@Timeout**: 表示测试方法运行如果超过了指定时间将会返回错误。
- **@ExtendWith**: 为测试类或测试方法提供扩展类引用。

```
import org.junit.jupiter.api.*;

@DisplayName("junit5功能测试类")
public class Junit5Test {

    @DisplayName("测试displayname注解")
    @Test
    void testDisplayName() {
        System.out.println(1);
        System.out.println(jdbcTemplate);
    }

    @ParameterizedTest
    @ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
    void palindromes(String candidate) {
        assertTrue(StringUtils.isPalindrome(candidate));
    }

    @Disabled
    @DisplayName("测试方法2")
    @Test
    void test2() {
        System.out.println(2);
    }

    @RepeatedTest(5)
    @Test
    void test3() {
        System.out.println(5);
    }

    /**
     * 规定方法超时时间。超出时间测试出异常
     *
     * @throws InterruptedException
     */
    @Timeout(value = 500, unit = TimeUnit.MILLISECONDS)
    @Test
    void testTimeout() throws InterruptedException {
        Thread.sleep(600);
    }
}
```

```
@BeforeEach
void testBeforeEach() {
    System.out.println("测试就要开始了...");
}

@AfterEach
void testAfterEach() {
    System.out.println("测试结束了...");
}

@BeforeAll
static void testBeforeAll() {
    System.out.println("所有测试就要开始了...");
}

@AfterAll
static void testAfterAll() {
    System.out.println("所有测试以及结束了...");
}

}
```

## 73、单元测试-断言机制

断言Assertion是测试方法中的核心部分，用来对测试需要满足的条件进行验证。这些断言方法都是org.junit.jupiter.api.Assertions的静态方法。检查业务逻辑返回的数据是否合理。所有的测试运行结束以后，会有一个详细的测试报告。

JUnit 5 内置的断言可以分成如下几个类别：

### 简单断言

用来对单个值进行简单的验证。如：

方法	说明
assertEquals	判断两个对象或两个原始类型是否相等
assertNotEquals	判断两个对象或两个原始类型是否不相等
assertSame	判断两个对象引用是否指向同一个对象
assertNotSame	判断两个对象引用是否指向不同的对象
assertTrue	判断给定的布尔值是否为 true
assertFalse	判断给定的布尔值是否为 false
assertNull	判断给定的对象引用是否为 null
assertNotNull	判断给定的对象引用是否不为 null

```

@Test
@DisplayName("simple assertion")
public void simple() {
    assertEquals(3, 1 + 2, "simple math");
    assertNotEquals(3, 1 + 1);

    assertNotSame(new Object(), new Object());
    Object obj = new Object();
    assertSame(obj, obj);

    assertFalse(1 > 2);
    assertTrue(1 < 2);

    assertNull(null);
    assertNotNull(new Object());
}

```

## 数组断言

通过 assertEquals 方法来判断两个对象或原始类型的数组是否相等。

```

@Test
@DisplayName("array assertion")
public void array() {
    assertEquals(new int[]{1, 2}, new int[] {1, 2});
}

```

## 组合断言

`assertAll()` 方法接受多个 `org.junit.jupiter.api.Executable` 函数式接口的实例作为要验证的断言，可以通过 lambda 表达式很容易的提供这些断言。

```
@Test
@DisplayName("assert all")
public void all() {
    assertAll("Math",
        () -> assertEquals(2, 1 + 1),
        () -> assertTrue(1 > 0)
    );
}
```

## 异常断言

在JUnit4时期，想要测试方法的异常情况时，需要用 `@Rule` 注解的 `ExpectedException` 变量还是比较麻烦的。而JUnit5提供了一种新的断言方式 `Assertions.assertThrows()`，配合函数式编程就可以进行使用。

```
@Test
@DisplayName("异常测试")
public void exceptionTest() {
    ArithmeticException exception = Assertions.assertThrows(
        //抛出断言异常
        ArithmeticException.class, () -> System.out.println(1 % 0));
}
```

## 超时断言

JUnit5还提供了`Assertions.assertTimeout()`为测试方法设置了超时时间。

```
@Test
@DisplayName("超时测试")
public void timeoutTest() {
    //如果测试方法时间超过1s将会异常
    Assertions.assertTimeout(Duration.ofMillis(1000), () -> Thread.sleep(500));
}
```

## 快速失败

通过 `fail` 方法直接使得测试失败。

```
@Test
@DisplayName("fail")
public void shouldFail() {
    fail("This should fail");
}
```

## 74、单元测试-前置条件

Unit 5 中的前置条件 (assumptions【假设】) 类似于断言, 不同之处在于不满足的**断言assertions**会使得测试方法失败, 而**不满足的前置条件只会使得测试方法的执行终止**。

前置条件可以看成是测试方法执行的前提, 当该前提不满足时, 就没有继续执行的必要。

```
@DisplayName("前置条件")
public class AssumptionsTest {
    private final String environment = "DEV";

    @Test
    @DisplayName("simple")
    public void simpleAssume() {
        assumeTrue(Objects.equals(this.environment, "DEV"));
        assumeFalse(() -> Objects.equals(this.environment, "PROD"));
    }

    @Test
    @DisplayName("assume then do")
    public void assumeThenDo() {
        assumingThat(
            Objects.equals(this.environment, "DEV"),
            () -> System.out.println("In DEV")
        );
    }
}
```

`assumeTrue` 和 `assumeFalse` 确保给定的条件为 `true` 或 `false`, 不满足条件会使得测试执行终止。

`assumingThat` 的参数是表示条件的布尔值和对应的 `Executable` 接口的实现对象。只有条件满足时, `Executable` 对象才会被执行; 当条件不满足时, 测试执行并不会终止。

## 75、单元测试-嵌套测试

[官方文档 - Nested Tests](#)

JUnit 5 可以通过 Java 中的内部类和 `@Nested` 注解实现嵌套测试, 从而可以更好的把相关的测试方法组织在一起。在内部类中可以使用 `@BeforeEach` 和 `@AfterEach` 注解, 而且嵌套的层次没有限制。

```
@DisplayName("A stack")
class TestingAStackDemo {

    Stack<Object> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested
    @DisplayName("when new")
```



```

class WhenNew {

    @BeforeEach
    void createNewStack() {
        stack = new Stack<>();
    }

    @Test
    @DisplayName("is empty")
    void isEmpty() {
        assertTrue(stack.isEmpty());
    }

    @Test
    @DisplayName("throws EmptyStackException when popped")
    void throwsExceptionWhenPopped() {
        assertThrows(EmptyStackException.class, stack::pop);
    }

    @Test
    @DisplayName("throws EmptyStackException when peeked")
    void throwsExceptionWhenPeeked() {
        assertThrows(EmptyStackException.class, stack::peek);
    }

    @Nested
    @DisplayName("after pushing an element")
    class AfterPushing {

        String anElement = "an element";

        @BeforeEach
        void pushAnElement() {
            stack.push(anElement);
        }

        @Test
        @DisplayName("it is no longer empty")
        void isEmpty() {
            assertFalse(stack.isEmpty());
        }

        @Test
        @DisplayName("returns the element when popped and is empty")
        void returnElementWhenPopped() {
            assertEquals(anElement, stack.pop());
            assertTrue(stack.isEmpty());
        }

        @Test
        @DisplayName("returns the element when peeked but remains not empty")
        void returnElementWhenPeeked() {

            assertEquals(anElement, stack.peek());
        }
    }
}

```

```

        assertFalse(stack.isEmpty());
    }
}
}
}

```

## 76、单元测试-参数化测试

### [官方文档 - Parameterized Tests](#)

参数化测试是JUnit5很重要的一个新特性，它使得用不同的参数多次运行测试成为了可能，也为我们的单元测试带来许多便利。

利用@ValueSource等注解，指定入参，我们将可以使用不同的参数进行多次单元测试，而不需要每新增一个参数就新增一个单元测试，省去了很多冗余代码。

利用@ValueSource等注解，指定入参，我们将可以使用不同的参数进行多次单元测试，而不需要每新增一个参数就新增一个单元测试，省去了很多冗余代码。

- **@ValueSource**: 为参数化测试指定入参来源，支持八大基础类以及String类型,Class类型
- **@NullSource**: 表示为参数化测试提供一个null的入参
- **@EnumSource**: 表示为参数化测试提供一个枚举入参
- **@CsvFileSource**: 表示读取指定CSV文件内容作为参数化测试入参
- **@MethodSource**: 表示读取指定方法的返回值作为参数化测试入参(注意方法返回需要是一个流)

当然如果参数化测试仅仅只能做到指定普通的入参还达不到让我觉得惊艳的地步。让我真正感到他的强大之处的地方在于他可以支持外部的各类入参。如:CSV,YML,JSON 文件甚至方法的返回值也可以作为入参。只需要去实现

**ArgumentsProvider** 接口，任何外部文件都可以作为它的入参。

```

@ParameterizedTest
@ValueSource(strings = {"one", "two", "three"})
@DisplayName("参数化测试1")
public void parameterizedTest1(String string) {
    System.out.println(string);
    Assertions.assertTrue(StringUtils.isNotBlank(string));
}

@ParameterizedTest
@MethodSource("method")    //指定方法名
@DisplayName("方法来源参数")
public void testWithExplicitLocalMethodSource(String name) {
    System.out.println(name);
    Assertions.assertNotNull(name);
}

static Stream<String> method() {
    return Stream.of("apple", "banana");
}

```

## 迁移指南

[官方文档 - Migrating from JUnit 4](#)

在进行迁移的时候需要注意如下的变化：

- 注解在 `org.junit.jupiter.api` 包中，断言在 `org.junit.jupiter.api.Assertions` 类中，前置条件在 `org.junit.jupiter.api.Assumptions` 类中。
- 把 `@Before` 和 `@After` 替换成 `@BeforeEach` 和 `@AfterEach`。
- 把 `@BeforeClass` 和 `@AfterClass` 替换成 `@BeforeAll` 和 `@AfterAll`。
- 把 `@Ignore` 替换成 `@Disabled`。
- 把 `@Category` 替换成 `@Tag`。
- 把 `@RunWith`、`@Rule` 和 `@ClassRule` 替换成 `@ExtendWith`。

## 77、指标监控-SpringBoot Actuator与Endpoint

未来每一个微服务在云上部署以后，我们都需要对其进行监控、追踪、审计、控制等。SpringBoot就抽取了Actuator场景，使得我们每个微服务快速引用即可获得生产级别的应用监控、审计等功能。

[官方文档 - Spring Boot Actuator: Production-ready Features](#)

1.x与2.x的不同：

- SpringBoot Actuator 1.x
  - 支持SpringMVC
  - 基于继承方式进行扩展
  - 层级Metrics配置
  - 自定义Metrics收集
  - 默认较少的安全策略
- SpringBoot Actuator 2.x
  - 支持SpringMVC、JAX-RS以及Webflux
  - 注解驱动进行扩展
  - 层级&名称空间Metrics
  - 底层使用MicroMeter，强大、便捷默认丰富的安全策略

## 如何使用

- 添加依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- 访问 `http://localhost:8080/actuator/**`。
- 暴露所有监控信息为HTTP。

```
management:
  endpoints:
    enabled-by-default: true #暴露所有端点信息
  web:
    exposure:
      include: '*' #以web方式暴露
```

- 测试例子

- <http://localhost:8080/actuator/beans>
- <http://localhost:8080/actuator/configprops>
- <http://localhost:8080/actuator/metrics>
- <http://localhost:8080/actuator/metrics/jvm.gc.pause>
- <http://localhost:8080/actuator/metrics/endpointName/detailPath>

actuator

英 ['æktju:ɜ:tə] 美 ['æktju,ɜ:tər]

n. 致动（促动，激励，调节）器；传动（装置，机构）；拖动装置；马达；操作机构；执行机构（元件）；（电磁铁）螺线管；操纵装置（阀门）；调速控制器；往复运动油（气）缸；作动筒

metric

英 ['metrɪk] 美 ['metrɪk]

adj. 米制的;公制的;按公制制作的;用公制测量的

n. 度量标准;[数学]度量;诗体;韵文;诗韵

## 78、指标监控-常使用的端点及开启与禁用

### 常使用的端点

ID	描述
<code>auditevents</code>	暴露当前应用程序的审核事件信息。需要一个 <code>AuditEventRepository</code> 组件。
<code>beans</code>	显示应用程序中所有Spring Bean的完整列表。
<code>caches</code>	暴露可用的缓存。
<code>conditions</code>	显示自动配置的所有条件信息，包括匹配或不匹配的原因。
<code>configprops</code>	显示所有 <code>@ConfigurationProperties</code> 。
<code>env</code>	暴露Spring的属性 <code>ConfigurableEnvironment</code>
<code>flyway</code>	显示已应用的所有Flyway数据库迁移。 需要一个或多个 <code>Flyway</code> 组件。
<code>health</code>	显示应用程序运行状况信息。
<code>httptrace</code>	显示HTTP跟踪信息（默认情况下，最近100个HTTP请求-响应）。需要一个 <code>HttpTraceRepository</code> 组件。
<code>info</code>	显示应用程序信息。
<code>integrationgraph</code>	显示Spring <code>integrationgraph</code> 。需要依赖 <code>spring-integration-core</code> 。
<code>loggers</code>	显示和修改应用程序中日志的配置。
<code>liquibase</code>	显示已应用的所有Liquibase数据库迁移。需要一个或多个 <code>Liquibase</code> 组件。
<code>metrics</code>	显示当前应用程序的“指标”信息。
<code>mappings</code>	显示所有 <code>@RequestMapping</code> 路径列表。
<code>scheduledtasks</code>	显示应用程序中的计划任务。
<code>sessions</code>	允许从Spring Session支持的会话存储中检索和删除用户会话。需要使用Spring Session的基于Servlet的Web应用程序。
<code>shutdown</code>	使应用程序正常关闭。默认禁用。
<code>startup</code>	显示由 <code>ApplicationStartup</code> 收集的启动步骤数据。需要使用 <code>SpringApplication</code> 进行配置 <code>BufferingApplicationStartup</code> 。
<code>threaddump</code>	执行线程转储。

如果您的应用程序是Web应用程序（Spring MVC，Spring WebFlux或Jersey），则可以使用以下附加端点：

ID	描述
heapdump	返回 hprof 堆转储文件。
jolokia	通过HTTP暴露JMX bean（需要引入Jolokia，不适用于WebFlux）。需要引入依赖 jolokia-core。
logfile	返回日志文件的内容（如果已设置 logging.file.name 或 logging.file.path 属性）。支持使用HTTP Range 标头来检索部分日志文件的内容。
prometheus	以Prometheus服务器可以抓取的格式公开指标。需要依赖 micrometer-registry-prometheus。

其中最常用的Endpoint:

- **Health: 监控状况**
- **Metrics: 运行时指标**
- **Loggers: 日志记录**

## Health Endpoint

健康检查端点，我们一般用于在云平台，平台会定时的检查应用的健康状况，我们就需要Health Endpoint可以为平台返回当前应用的一系列组件健康状况的集合。

重要的几点:

- health endpoint返回的结果，应该是一系列健康检查后的一个汇总报告。
- 很多的健康检查默认已经自动配置好了，比如：数据库、redis等。
- 可以很容易的添加自定义的健康检查机制。

## Metrics Endpoint

提供详细的、层级的、空间指标信息，这些信息可以被pull（主动推送）或者push（被动获取）方式得到:

- 通过Metrics对接多种监控系统。
- 简化核心Metrics开发。
- 添加自定义Metrics或者扩展已有Metrics。

## 开启与禁用Endpoints

- 默认所有的Endpoint除过shutdown都是开启的。
- 需要开启或者禁用某个Endpoint。配置模式为 `management.endpoint.<endpointName>.enabled = true`

```
management:
  endpoint:
    beans:
      enabled: true
```

- 或者禁用所有的Endpoint然后手动开启指定的Endpoint。

```
management:
  endpoints:
    enabled-by-default: false
  endpoint:
    beans:
      enabled: true
    health:
      enabled: true
```

## 暴露Endpoints

支持的暴露方式

- HTTP：默认只暴露health和info。
- JMX：默认暴露所有Endpoint。
- 除过health和info，剩下的Endpoint都应该进行保护访问。如果引入Spring Security，则会默认配置安全访问规则。

ID	JMX	Web
auditevents	Yes	No
beans	Yes	No
caches	Yes	No
conditions	Yes	No
configprops	Yes	No
env	Yes	No
flyway	Yes	No
health	Yes	Yes
heapdump	N/A	No
httptrace	Yes	No
info	Yes	Yes
integrationgraph	Yes	No
jolokia	N/A	No
logfile	N/A	No
loggers	Yes	No
liquibase	Yes	No
metrics	Yes	No
mappings	Yes	No
prometheus	N/A	No
scheduledtasks	Yes	No
sessions	Yes	No
shutdown	Yes	No
startup	Yes	No
threaddump	Yes	No

若要更改公开的Endpoint，请配置以下的包含和排除属性：



Property	Default
<code>management.endpoints.jmx.exposure.exclude</code>	
<code>management.endpoints.jmx.exposure.include</code>	<code>*</code>
<code>management.endpoints.web.exposure.exclude</code>	
<code>management.endpoints.web.exposure.include</code>	<code>info, health</code>

[官方文档 - Exposing Endpoints](#)

## 79、指标监控-定制Endpoint

### 定制 Health 信息

```
management:
  health:
    enabled: true
    show-details: always #总是显示详细信息。可显示每个模块的状态信息
```

通过实现 `HealthIndicator` 接口，或继承 `MyComHealthIndicator` 类。

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }

}

/*
构建Health
Health build = Health.down()
    .withDetail("msg", "error service")
    .withDetail("code", "500")
    .withException(new RuntimeException())
    .build();

*/
```

```

@Component
public class MyComHealthIndicator extends AbstractHealthIndicator {

    /**
     * 真实的检查方法
     * @param builder
     * @throws Exception
     */
    @Override
    protected void doHealthCheck(Health.Builder builder) throws Exception {
        //mongodb。 获取连接进行测试
        Map<String, Object> map = new HashMap<>();
        // 检查完成
        if(1 == 2){
            //          builder.up(); //健康
            builder.status(Status.UP);
            map.put("count", 1);
            map.put("ms", 100);
        }else {
            //          builder.down();
            builder.status(Status.OUT_OF_SERVICE);
            map.put("err", "连接超时");
            map.put("ms", 3000);
        }

        builder.withDetail("code", 100)
                .withDetails(map);
    }
}

```

## 定制info信息

常用两种方式：

- 编写配置文件

```

info:
  appName: boot-admin
  version: 2.0.1
  mavenProjectName: @project.artifactId@ #使用@@可以获取maven的pom文件值
  mavenProjectVersion: @project.version@

```

- 编写InfoContributor

```

import java.util.Collections;

```

```
import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class ExampleInfoContributor implements InfoContributor {

    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("example",
            Collections.singletonMap("key", "value"));
    }

}
```

<http://localhost:8080/actuator/info> 会输出以上方式返回的所有info信息

## 定制Metrics信息

[Spring Boot支持的metrics](#)

增加定制Metrics:

```
class MyService{
    Counter counter;
    public MyService(MeterRegistry meterRegistry){
        counter = meterRegistry.counter("myservice.method.running.counter");
    }

    public void hello() {
        counter.increment();
    }
}
```

//也可以使用下面的方式

```
@Bean
MeterBinder queueSize(Queue queue) {
    return (registry) -> Gauge.builder("queueSize", queue::size).register(registry);
}
```

## 定制Endpoint

```
@Component
@Endpoint(id = "container")
public class DockerEndpoint {

    @ReadOperation
```

```

    public Map getDockerInfo(){
        return Collections.singletonMap("info","docker started...");
    }

    @WriteOperation
    private void restartDocker(){
        System.out.println("docker restarted...");
    }
}

```

场景：

- 开发ReadinessEndpoint来管理程序是否就绪。
- 开发LivenessEndpoint来管理程序是否存活。

## 80、指标监控-Boot Admin Server

[官方Github](#)

[官方文档](#)

可视化指标监控

What is Spring Boot Admin?

codecentric's Spring Boot Admin is a community project to manage and monitor your [Spring Boot](#)® applications. The applications register with our Spring Boot Admin Client (via HTTP) or are discovered using Spring Cloud® (e.g. Eureka, Consul). The UI is just a Vue.js application on top of the Spring Boot Actuator endpoints.

[开始使用方法](#)

## 81、高级特性-Profile环境切换

为了方便多环境适配，Spring Boot简化了profile功能。

- 默认配置文件 `application.yaml` 任何时候都会加载。
- 指定环境配置文件 `application-{env}.yaml`，`env` 通常替代为 `test`，
- 激活指定环境
  - 配置文件激活：`spring.profiles.active=prod`
  - 命令行激活：`java -jar xxx.jar --spring.profiles.active=prod --person.name=haha`（修改配置文件的任意值，**命令行优先**）
- 默认配置与环境配置同时生效
- 同名配置项，profile配置优先

### @Profile条件装配功能

```

@Data
@Component
@ConfigurationProperties("person")//在配置文件中配置
public class Person{
    private String name;
    private Integer age;
}

```

application.properties

```

person:
  name: lun
  age: 8

```

```

public interface Person {

    String getName();
    Integer getAge();

}

@Profile("test")//加载application-test.yaml里的
@Component
@ConfigurationProperties("person")
@Data
public class Worker implements Person {

    private String name;
    private Integer age;
}

@Profile(value = {"prod","default"})//加载application-prod.yaml里的
@Component
@ConfigurationProperties("person")
@Data
public class Boss implements Person {

    private String name;
    private Integer age;
}

```

application-test.yaml

```

person:
  name: test-张三

server:
  port: 7000

```

application-prod.yaml

```
person:
  name: prod-张三

server:
  port: 8000
```

application.properties

```
# 激活prod配置文件
spring.profiles.active=prod
```

```
@Autowired
private Person person;

@GetMapping("/")
public String hello(){
    //激活了prod, 则返回Boss; 激活了test, 则返回Worker
    return person.getClass().toString();
}
```

@Profile还可以修饰在方法上:

```
class Color {
}

@Configuration
public class MyConfig {

    @Profile("prod")
    @Bean
    public Color red(){
        return new Color();
    }

    @Profile("test")
    @Bean
    public Color green(){
        return new Color();
    }
}
```

可以激活一组：

```
spring.profiles.active=production

spring.profiles.group.production[0]=proddb
spring.profiles.group.production[1]=prodmq
```

## 82、高级特性-配置加载优先级

### 外部化配置

[官方文档 - Externalized Configuration](#)

Spring Boot uses a very particular `PropertySource` order that is designed to allow sensible overriding of values. Properties are considered in the following order (with values from lower items overriding earlier ones) (1优先级最低，14优先级最高)：

1. Default properties (specified by setting `SpringApplication.setDefaultProperties` ).
2. `@PropertySource` annotations on your `@Configuration` classes. Please note that such property sources are not added to the `Environment` until the application context is being refreshed. This is too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.
3. Config data (such as `application.properties` files)
4. A `RandomValuePropertySource` that has properties only in `random.*` .
5. OS environment variables.
6. Java System properties ( `System.getProperties()` ).
7. JNDI attributes from `java:comp/env` .
8. `ServletContext` init parameters.
9. `ServletConfig` init parameters.
10. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. `properties` attribute on your tests. Available on `@SpringBootTest` and the [test annotations for testing a particular slice of your application](#).
13. `@TestPropertySource` annotations on your tests.
14. [Devtools global settings properties](#) in the `$HOME/.config/spring-boot` directory when devtools is active.

```
import org.springframework.stereotype.*;
import org.springframework.beans.factory.annotation.*;

@Component
public class MyBean {

    @Value("${name}")//以这种方式可以获得配置值
    private String name;

    // ...

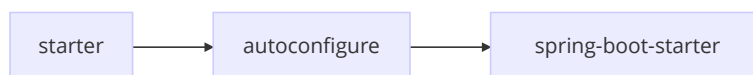
}
```

- 外部配置源
  - Java属性文件。
  - YAML文件。
  - 环境变量。
  - 命令行参数。
- 配置文件查找位置
  1. classpath 根路径。
  2. classpath 根路径下config目录。
  3. jar包当前目录。
  4. jar包当前目录的config目录。
  5. /config子目录的直接子目录。
- 配置文件加载顺序：
  1. 当前jar包内部的 application.properties 和 application.yml。
  2. 当前jar包内部的 application-{profile}.properties 和 application-{profile}.yml。
  3. 引用的外部jar包的 application.properties 和 application.yml。
  4. 引用的外部jar包的 application-{profile}.properties 和 application-{profile}.yml。
- 指定环境优先，外部优先，后面的可以覆盖前面的同名配置项。

## 83、高级特性-自定义starter细节

### starter启动原理

- starter的pom.xml引入autoconfigure依赖



- autoconfigure包中配置使用 META-INF/spring.factories 中 EnableAutoConfiguration 的值，使得项目启动加载指定的自动配置类
- 编写自动配置类 xxxAutoConfiguration -> xxxxProperties



- o @Configuration
  - o @Conditional
  - o @EnableConfigurationProperties
  - o @Bean
  - o .....
- 引入starter --- xxxAutoConfiguration --- 容器中放入组件 --- 绑定xxxProperties ---- 配置项

## 自定义starter

- 目标: 创建 HelloService 的自定义starter。
- 创建两个工程, 分别命名为 hello-spring-boot-starter (普通Maven工程), hello-spring-boot-starter-autoconfigure (需用用到Spring Initializr创建的Maven工程)。
- hello-spring-boot-starter 无需编写什么代码, 只需让该工程引入 hello-spring-boot-starter-autoconfigure 依赖:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.lun</groupId>
    <artifactId>hello-spring-boot-starter</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>com.lun</groupId>
            <artifactId>hello-spring-boot-starter-autoconfigure</artifactId>
            <version>1.0.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

</project>
```

- hello-spring-boot-starter-autoconfigure 的pom.xml如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <groupId>com.lun</groupId>
```

```

<artifactId>hello-spring-boot-starter-autoconfigure</artifactId>
<version>1.0.0-SNAPSHOT</version>
<name>hello-spring-boot-starter-autoconfigure</name>
<description>Demo project for Spring Boot</description>
<properties>
    <java.version>1.8</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
</dependencies>
</project>

```

- 创建4个文件:

- com/lun/hello/auto/HelloServiceAutoConfiguration
- com/lun/hello/bean/HelloProperties
- com/lun/hello/service/HelloService
- src/main/resources/META-INF/spring.factories

```

import com.lun.hello.bean.HelloProperties;
import com.lun.hello.service.HelloService;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@ConditionalOnMissingBean(HelloService.class)
@EnableConfigurationProperties(HelloProperties.class)//默认HelloProperties放在容器中
public class HelloServiceAutoConfiguration {

    @Bean
    public HelloService helloService(){
        return new HelloService();
    }

}

```

```

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("hello")
public class HelloProperties {
    private String prefix;
    private String suffix;

    public String getPrefix() {
        return prefix;
    }
}

```

```

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public String getSuffix() {
        return suffix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
}

```

```

import com.lun.hello.bean.HelloProperties;
import org.springframework.beans.factory.annotation.Autowired;

/**
 * 默认不要放在容器中
 */
public class HelloService {

    @Autowired
    private HelloProperties helloProperties;

    public String sayHello(String userName){
        return helloProperties.getPrefix() + ": " + userName + " > " +
helloProperties.getSuffix();
    }
}

```

```

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.lun.hello.auto.HelloServiceAutoConfiguration

```

- 用maven插件，将两工程install到本地。
- 接下来，测试使用自定义starter，用Spring Initializr创建名为 `hello-spring-boot-starter-test` 工程，引入 `hello-spring-boot-starter` 依赖，其pom.xml如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.2</version>

        <relativePath/> <!-- lookup parent from repository -->
    
```

```

</parent>
<groupId>com.lun</groupId>
<artifactId>hello-spring-boot-starter-test</artifactId>
<version>1.0.0-SNAPSHOT</version>
<name>hello-spring-boot-starter-test</name>
<description>Demo project for Spring Boot</description>
<properties>
    <java.version>1.8</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <!-- 引入`hello-spring-boot-starter`依赖 -->
    <dependency>
        <groupId>com.lun</groupId>
        <artifactId>hello-spring-boot-starter</artifactId>
        <version>1.0.0-SNAPSHOT</version>
    </dependency>

</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>

```

- 添加配置文件 `application.properties` :

```

hello.prefix=hello
hello.suffix=666

```

- 添加单元测试类:

```

import com.lun.hello.service.HelloService; //来自自定义starter
import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.Test;

```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class HelloSpringBootTestApplicationTests {

    @Autowired
    private HelloService helloService;

    @Test
    void contextLoads() {
        // System.out.println(helloService.sayHello("lun"));
        Assertions.assertEquals("hello: lun > 666", helloService.sayHello("lun"));
    }

}
```

## 84、原理解析-SpringApplication创建初始化流程

### SpringBoot启动过程

Spring Boot应用的启动类：

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloSpringBootTestApplication {

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

}
```

```
public class SpringApplication {

    ...

    public static ConfigurableApplicationContext run(Class<?> primarySource, String... args) {
        return run(new Class<?>[] { primarySource }, args);
    }

    public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args) {
        return new SpringApplication(primarySources).run(args);
    }

    //先看看new SpringApplication(primarySources), 下一节再看看run()

    public SpringApplication(Class<?>... primarySources) {
```

```

        this(null, primarySources);
    }

    public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
        this.resourceLoader = resourceLoader;
        Assert.notNull(primarySources, "PrimarySources must not be null");
        this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
        //WebApplicationType是枚举类, 有NONE, SERVLET, REACTIVE, 下行webApplicationType是SERVLET
        this.webApplicationType = WebApplicationType.deduceFromClasspath();

        //初始启动引导器, 去spring.factories文件中找org.springframework.boot.Bootstrapper, 但我找不到实现Bootstrapper接口的类
        this.bootstrappers = new ArrayList<>(getSpringFactoriesInstances(Bootstrapper.class));

        //去spring.factories找 ApplicationContextInitializer
        setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));

        //去spring.factories找 ApplicationListener
        setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));

        this.mainApplicationClass = deduceMainApplicationClass();
    }

    private Class<?> deduceMainApplicationClass() {
        try {
            StackTraceElement[] stackTrace = new RuntimeException().getStackTrace();
            for (StackTraceElement stackTraceElement : stackTrace) {
                if ("main".equals(stackTraceElement.getMethodName())) {
                    return Class.forName(stackTraceElement.getClassName());
                }
            }
        }
        catch (ClassNotFoundException ex) {
            // Swallow and continue
        }
        return null;
    }

    ...
}

```

spring.factories:

```

...

# Application Context Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer,\
org.springframework.boot.context.ContextIdApplicationContextInitializer,\

```

```

org.springframework.boot.context.config.DelegatingApplicationContextInitializer,\
org.springframework.boot.rsocket.context.RSocketPortInfoApplicationContextInitializer,\
org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.ClearCachesApplicationListener,\
org.springframework.boot.builder.ParentContextCloserApplicationListener,\
org.springframework.boot.context.FileEncodingApplicationListener,\
org.springframework.boot.context.config.AnsiOutputApplicationListener,\
org.springframework.boot.context.config.DelegatingApplicationListener,\
org.springframework.boot.context.logging.LoggingApplicationListener,\
org.springframework.boot.env.EnvironmentPostProcessorApplicationListener,\
org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener

...

```

## 85、原理解析-SpringBoot完整启动过程

继续上一节，接着讨论 `return new SpringApplication(primarySources).run(args)` 的 `run` 方法

```

public class SpringApplication {

    ...

    public ConfigurableApplicationContext run(String... args) {
        Stopwatch stopwatch = new Stopwatch();//开始计时器
        stopwatch.start();//开始计时

        //1.
        //创建引导上下文 (Context环境) createBootstrapContext()
        //获取到所有之前的 bootstrappers 挨个执行 initialize() 来完成对引导启动器上下文环境设置
        DefaultBootstrapContext bootstrapContext = createBootstrapContext();

        //2.到最后该方法会返回这context
        ConfigurableApplicationContext context = null;

        //3.让当前应用进入headless模式
        configureHeadlessProperty();

        //4.获取所有 RunListener (运行监听器) ,为了方便所有Listener进行事件感知
        SpringApplicationRunListeners listeners = getRunListeners(args);

        //5. 遍历 SpringApplicationRunListener 调用 starting 方法;
        // 相当于通知所有感兴趣系统正在启动过程的人, 项目正在 starting.
        listeners.starting(bootstrapContext, this.mainApplicationClass);
        try {
            //6.保存命令行参数 ApplicationArguments

            ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);

```

```
//7.准备环境
ConfigurableEnvironment environment = prepareEnvironment(listeners,
bootstrapContext, applicationArguments);
configureIgnoreBeanInfo(environment);

/*打印标志
      .  _ _ _ _ _
     /\ / _ _ ' _ _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \ \
    ( ( ) \ _ _ | ' _ | ' _ | ' _ \ _ _ | \ \ \ \ \
     \ \ _ _ ) | _ | | | | | | | ( _ | | ) ) ) )
      ' | _ _ | . _ | | | _ | _ \ , | / / / /
     =====|_|=====|_|/_/_/_/_/
    :: Spring Boot ::                (v2.4.2)
*/
Banner printedBanner = printBanner(environment);

// 创建IOC容器 (createApplicationContext () )
// 根据项目类型webApplicationType (NONE,SERVLET,REACTIVE) 创建容器,
// 当前会创建 AnnotationConfigServletWebServerApplicationContext
context = createApplicationContext();
context.setApplicationStartup(this.applicationStartup);

//8.准备ApplicationContext IOC容器的基本信息
prepareContext(bootstrapContext, context, environment, listeners,
applicationArguments, printedBanner);
//9.刷新IOC容器,创建容器中的所有组件,Spring框架的内容
refreshContext(context);
//该方法没内容,大概为将来填入
afterRefresh(context, applicationArguments);
stopWatch.stop();//停止计时
if (this.logStartupInfo) { //this.logStartupInfo默认是true
    new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);
}
//10.
listeners.started(context);

//11.调用所有runners
callRunners(context, applicationArguments);
}
catch (Throwable ex) {
    //13.
    handleRunFailure(context, ex, listeners);
    throw new IllegalStateException(ex);
}

try {
    //12.
    listeners.running(context);
}
catch (Throwable ex) {
    //13.
```



```

        handleRunFailure(context, ex, null);
        throw new IllegalStateException(ex);
    }
    return context;
}

//1.
private DefaultBootstrapContext createBootstrapContext() {
    DefaultBootstrapContext bootstrapContext = new DefaultBootstrapContext();
    this.bootstrapers.forEach((initializer) -> initializer.initialize(bootstrapContext));
    return bootstrapContext;
}

//3.
private void configureHeadlessProperty() {
    //this.headless默认为true
    System.setProperty(SYSTEM_PROPERTY_JAVA_AWT_HEADLESS,
        System.getProperty(SYSTEM_PROPERTY_JAVA_AWT_HEADLESS,
Boolean.toString(this.headless)));
}

private static final String SYSTEM_PROPERTY_JAVA_AWT_HEADLESS = "java.awt.headless";

//4.
private SpringApplicationRunListeners getRunListeners(String[] args) {
    Class<?>[] types = new Class<?>[] { SpringApplication.class, String[].class };
    //getSpringFactoriesInstances 去 spring.factories 找 SpringApplicationRunListener
    return new SpringApplicationRunListeners(logger,
        getSpringFactoriesInstances(SpringApplicationRunListener.class, types, this,
args),
        this.applicationStartup);
}

//7.准备环境
private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners listeners,
    DefaultBootstrapContext bootstrapContext, ApplicationArguments
applicationArguments) {
    // Create and configure the environment
    //返回或者创建基础环境信息对象, 如: StandardServletEnvironment,
StandardReactiveWebEnvironment
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    //配置环境信息对象, 读取所有的配置源的配置属性值。
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    //绑定环境信息
    ConfigurationPropertySources.attach(environment);
    //7.1 通知所有的监听器当前环境准备完成
    listeners.environmentPrepared(bootstrapContext, environment);
    DefaultPropertiesPropertySource.moveToEnd(environment);
    configureAdditionalProfiles(environment);
    bindToSpringApplication(environment);
    if (!this.isCustomEnvironment) {
        environment = new

```

```

EnvironmentConverter(getClassLoader()).convertEnvironmentIfNecessary(environment,
    deduceEnvironmentClass());
    }
    ConfigurationPropertySources.attach(environment);
    return environment;
}

//8.
private void prepareContext(DefaultBootstrapContext bootstrapContext,
    ConfigurableApplicationContext context,
        ConfigurableEnvironment environment, SpringApplicationRunListeners listeners,
        ApplicationArguments applicationArguments, Banner printedBanner) {
    //保存环境信息
    context.setEnvironment(environment);
    //IOC容器的后置处理流程
    postProcessApplicationContext(context);
    //应用初始化器
    applyInitializers(context);
    //8.1 遍历所有的 listener 调用 contextPrepared。
    //EventPublishRunListener通知所有的监听器contextPrepared
    listeners.contextPrepared(context);
    bootstrapContext.close(context);
    if (this.logStartupInfo) {
        logStartupInfo(context.getParent() == null);
        logStartupProfileInfo(context);
    }
    // Add boot specific singleton beans
    ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
    beanFactory.registerSingleton("springApplicationArguments", applicationArguments);
    if (printedBanner != null) {
        beanFactory.registerSingleton("springBootBanner", printedBanner);
    }
    if (beanFactory instanceof DefaultListableBeanFactory) {
        ((DefaultListableBeanFactory) beanFactory)
            .setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
    }
    if (this.lazyInitialization) {
        context.addBeanFactoryPostProcessor(new
LazyInitializationBeanFactoryPostProcessor());
    }
    // Load the sources
    Set<Object> sources = getAllSources();
    Assert.notEmpty(sources, "Sources must not be empty");
    load(context, sources.toArray(new Object[0]));
    //8.2
    listeners.contextLoaded(context);
}

//11.调用所有runners
private void callRunners(ApplicationContext context, ApplicationArguments args) {
    List<Object> runners = new ArrayList<>();

    //获取容器中的 ApplicationRunner

```

```

        runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
        //获取容器中的 CommandLineRunner
        runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
        //合并所有runner并且按照@Order进行排序
        AnnotationAwareOrderComparator.sort(runners);
        //遍历所有的runner。调用 run 方法
        for (Object runner : new LinkedHashSet<>(runners)) {
            if (runner instanceof ApplicationRunner) {
                callRunner((ApplicationRunner) runner, args);
            }
            if (runner instanceof CommandLineRunner) {
                callRunner((CommandLineRunner) runner, args);
            }
        }
    }

    //13.
    private void handleRunFailure(ConfigurableApplicationContext context, Throwable exception,
        SpringApplicationRunListeners listeners) {
        try {
            try {
                handleExitCode(context, exception);
                if (listeners != null) {
                    //14.
                    listeners.failed(context, exception);
                }
            }
            finally {
                reportFailure(getExceptionReporters(context), exception);
                if (context != null) {
                    context.close();
                }
            }
        }
        catch (Exception ex) {
            logger.warn("Unable to close ApplicationContext", ex);
        }
        ReflectionUtils.rethrowRuntimeException(exception);
    }

    ...
}

```

```

//2. new SpringApplication(primarySources).run(args) 最后返回的接口类型
public interface ConfigurableApplicationContext extends ApplicationContext, Lifecycle, Closeable {
    String CONFIG_LOCATION_DELIMITERS = ",; \t\n";
    String CONVERSION_SERVICE_BEAN_NAME = "conversionService";
    String LOAD_TIME_WEAVER_BEAN_NAME = "loadTimeWeaver";
    String ENVIRONMENT_BEAN_NAME = "environment";
    String SYSTEM_PROPERTIES_BEAN_NAME = "systemProperties";
}

```

```

String SYSTEM_ENVIRONMENT_BEAN_NAME = "systemEnvironment";
String APPLICATION_STARTUP_BEAN_NAME = "applicationStartup";
String SHUTDOWN_HOOK_THREAD_NAME = "SpringContextShutdownHook";

void setId(String var1);

void setParent(@Nullable ApplicationContext var1);

void setEnvironment(ConfigurableEnvironment var1);

ConfigurableEnvironment getEnvironment();

void setApplicationStartup(ApplicationStartup var1);

ApplicationStartup getApplicationStartup();

void addBeanFactoryPostProcessor(BeanFactoryPostProcessor var1);

void addApplicationListener(ApplicationListener<?> var1);

void setClassLoader(ClassLoader var1);

void addProtocolResolver(ProtocolResolver var1);

void refresh() throws BeansException, IllegalStateException;

void registerShutdownHook();

void close();

boolean isActive();

ConfigurableListableBeanFactory getBeanFactory() throws IllegalStateException;
}

```

```

#4.
#spring.factories
# Run Listeners
org.springframework.boot.SpringApplicationRunListener=\
org.springframework.boot.context.event.EventPublishingRunListener

```

```

class SpringApplicationRunListeners {

    private final Log log;

    private final List<SpringApplicationRunListener> listeners;

    private final ApplicationStartup applicationStartup;

    SpringApplicationRunListeners(Log log, Collection<? extends SpringApplicationRunListener>
listeners,

        ApplicationStartup applicationStartup) {

```

```

        this.log = log;
        this.listeners = new ArrayList<>(listeners);
        this.applicationStartup = applicationStartup;
    }

    //5.遍历 SpringApplicationRunListener 调用 starting 方法;
    //相当于通知所有感兴趣系统正在启动过程的人, 项目正在 starting。
    void starting(ConfigurableBootstrapContext bootstrapContext, Class<?> mainApplicationClass)
    {
        doWithListeners("spring.boot.application.starting", (listener) ->
listener.starting(bootstrapContext),
            (step) -> {
                if (mainApplicationClass != null) {
                    step.tag("mainApplicationClass", mainApplicationClass.getName());
                }
            });
    }

    //7.1
    void environmentPrepared(ConfigurableBootstrapContext bootstrapContext,
ConfigurableEnvironment environment) {
        doWithListeners("spring.boot.application.environment-prepared",
            (listener) -> listener.environmentPrepared(bootstrapContext, environment));
    }

    //8.1
    void contextPrepared(ConfigurableApplicationContext context) {
        doWithListeners("spring.boot.application.context-prepared", (listener) ->
listener.contextPrepared(context));
    }

    //8.2
    void contextLoaded(ConfigurableApplicationContext context) {
        doWithListeners("spring.boot.application.context-loaded", (listener) ->
listener.contextLoaded(context));
    }

    //10.
    void started(ConfigurableApplicationContext context) {
        doWithListeners("spring.boot.application.started", (listener) ->
listener.started(context));
    }

    //12.
    void running(ConfigurableApplicationContext context) {
        doWithListeners("spring.boot.application.running", (listener) ->
listener.running(context));
    }

    //14.
    void failed(ConfigurableApplicationContext context, Throwable exception) {
        doWithListeners("spring.boot.application.failed",

            (listener) -> callFailedListener(listener, context, exception), (step) -> {

```

```

        step.tag("exception", exception.getClass().toString());
        step.tag("message", exception.getMessage());
    });
}

private void doWithListeners(String stepName, Consumer<SpringApplicationRunListener>
listenerAction,
    Consumer<StartupStep> stepAction) {
    StartupStep step = this.applicationStartup.start(stepName);
    this.listeners.forEach(listenerAction);
    if (stepAction != null) {
        stepAction.accept(step);
    }
    step.end();
}

...
}

```

## 86、原理解析-自定义事件监听组件

MyApplicationContextInitializer.java

```

import org.springframework.context.ApplicationContextInitializer;
import org.springframework.context.ConfigurableApplicationContext;

public class MyApplicationContextInitializer implements ApplicationContextInitializer {
    @Override
    public void initialize(ConfigurableApplicationContext applicationContext) {
        System.out.println("MyApplicationContextInitializer ....initialize.... ");
    }
}

```

MyApplicationListener.java

```

import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

public class MyApplicationListener implements ApplicationListener {
    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        System.out.println("MyApplicationListener.....onApplicationEvent...");
    }
}

```

#### MyApplicationRunner.java

```
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Order(1)
@Component//放入容器
public class MyApplicationRunner implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("MyApplicationRunner...run...");
    }
}
```

#### MyCommandLineRunner.java

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
/**
 * 应用启动做一个一次性事情
 */
@Order(2)
@Component//放入容器
public class MyCommandLineRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("MyCommandLineRunner....run....");
    }
}
```

#### MySpringApplicationRunListener.java

```
import org.springframework.boot.ConfigurableBootstrapContext;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.SpringApplicationRunListener;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.core.env.ConfigurableEnvironment;

public class MySpringApplicationRunListener implements SpringApplicationRunListener {

    private SpringApplication application;
    public MySpringApplicationRunListener(SpringApplication application, String[] args){
        this.application = application;
    }
}
```

```

@Override
public void starting(ConfigurableBootstrapContext bootstrapContext) {
    System.out.println("MySpringApplicationRunListener....starting....");
}

@Override
public void environmentPrepared(ConfigurableBootstrapContext bootstrapContext,
ConfigurableEnvironment environment) {
    System.out.println("MySpringApplicationRunListener....environmentPrepared....");
}

@Override
public void contextPrepared(ConfigurableApplicationContext context) {
    System.out.println("MySpringApplicationRunListener....contextPrepared....");
}

@Override
public void contextLoaded(ConfigurableApplicationContext context) {
    System.out.println("MySpringApplicationRunListener....contextLoaded....");
}

@Override
public void started(ConfigurableApplicationContext context) {
    System.out.println("MySpringApplicationRunListener....started....");
}

@Override
public void running(ConfigurableApplicationContext context) {
    System.out.println("MySpringApplicationRunListener....running....");
}

@Override
public void failed(ConfigurableApplicationContext context, Throwable exception) {
    System.out.println("MySpringApplicationRunListener....failed....");
}
}

```

注册 `MyApplicationContextInitializer` , `MyApplicationListener` , `MySpringApplicationRunListener` :

`resources / META-INF / spring.factories` :



```
org.springframework.context.ApplicationContextInitializer=\n    com.lun.boot.listener.MyApplicationContextInitializer\n\norg.springframework.context.ApplicationListener=\n    com.lun.boot.listener.MyApplicationListener\n\norg.springframework.boot.SpringApplicationRunListener=\n    com.lun.boot.listener.MySpringApplicationRunListener
```

## 87、后会有期

路漫漫其修远兮，吾将上下而求索。

纸上得来终觉浅，绝知此事要躬行。

- Spring Boot 2 场景整合篇
  - 虚拟化技术
  - 安全控制
  - 缓存技术
  - 消息中间件
  - 对象存储
  - 定时调度
  - 异步任务
  - 分布式系统
- Spring Boot 2 响应式编程
  - 响应式编程基础
  - Webflux开发Web应用
  - 响应式访问持久化层
  - 响应式安全开发
  - 响应式原理