工程实现文档

张雄彪

Version 1.0, 2016-03-19

Table of Contents

SSO 单点登录

SSO英文全称Single Sign On，单点登录。
SSO是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。它包括可以将这次主要的登录映射到其他应用中用于同一个用户的登录的机制。它是目前比较流行的企业业务整合的解决方案之一。

Chapter 1. 实现机制

当用户第一次访问应用系统1的时候，因为还没有登录，会被引导到认证系统中进行登录；根据用户提供的登录信息，认证系统进行身份校验，如果通过校验，应该返回给用户一个认证的凭据－－ticket；用户再访问别的应用的时候就会将这个ticket带上，作为自己认证的凭据，应用系统接受到请求之后会把ticket送到认证系统进行校验，检查ticket的合法性。如果通过校验，用户就可以在不用再次登录的情况下访问应用系统2和应用系统3了。 要实现SSO，需要以下主要的功能：

系统共享

统一的认证系统是SSO的前提之一。认证系统的主要功能是将用户的登录信息和用户信息库相比较，对用户进行登录认证；认证成功后，认证系统应该生成统一的认证标志（ticket），返还给用户。另外，认证系统还应该对ticket进行校验，判断其有效性。

信息识别

要实现SSO的功能，让用户只登录一次，就必须让应用系统能够识别已经登录过的用户。应用系统应该能对ticket进行识别和提取，通过与认证系统的通讯，能自动判断当前用户是否登录过，从而完成单点登录的功能。

Chapter 2. 主要实现技术

基于*cookies*实现

如果是基于两个域名之间传递sessionid的方法可能在windows中成立，在unix&linux中可能会出现问题;可以基于数据库实现；在安全性方面可能会作更多的考虑。另外，关于跨域问题，虽然cookies本身不跨域，但可以利用它实现跨域的SSO。

Broker-based（基于经纪人）

这种技术的特点就是，有一个集中的认证和用户帐号管理的服务器。经纪人给被用于进一步请求的电子的身份存取。

Agent-based（基于代理人）

在这种解决方案中，有一个自动地为不同的应用程序认证用户身份的代理程序。这个代理程序需要设计有不同的功能。比如，它可以使用口令表或加密密钥来自动地将认证的负担从用户移开。代理人被放在服务器上面，在服务器的认证系统和客户端认证方法之间充当一个"翻译"。例如SSH等。

Token-based

例如SecurID,WebID，现在被广泛使用的口令认证，比如FTP,邮件服务器的登录认证，这是一种简单易用的方式，实现一个口令在多种应用当中使用。

基于安全断言标记语言（*SAML*）实现

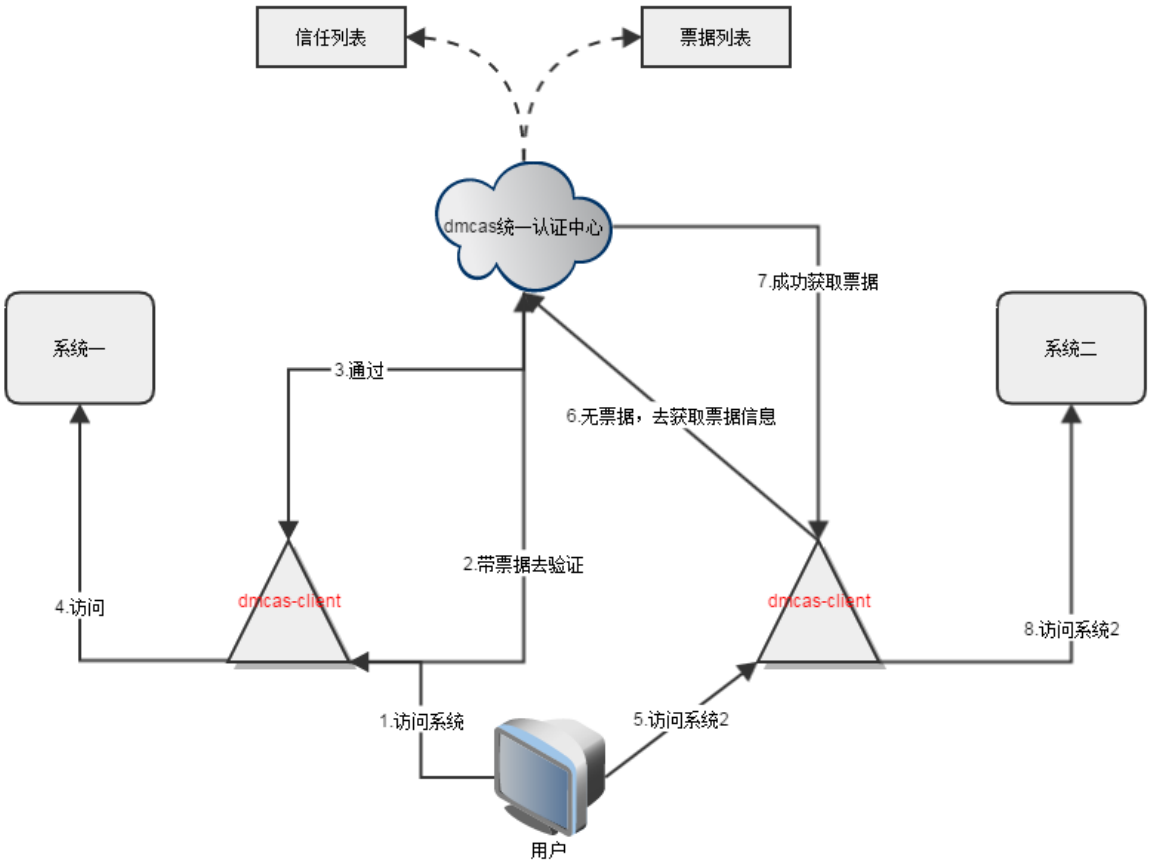
SAML(Security Assertion Markup Language，安全断言标记语言）的出现大大简化了SSO，并被OASIS批准为SSO的执行标准。开源组织OpenSAML 实现了 SAML 规范。

Chapter 3. 现有实现设计

实现方式

目前在现在的系统中，采用了基于 *cookies* 的实现，在多个系统中使用 *cookie* 来传输票据。

单点登录实现的要点在于将认证中心进行统一，在所有系统之上构建一个唯一的认证中心即可。如下图，多系统之间的访问，票据认证应该全部在统一认证中心完成。

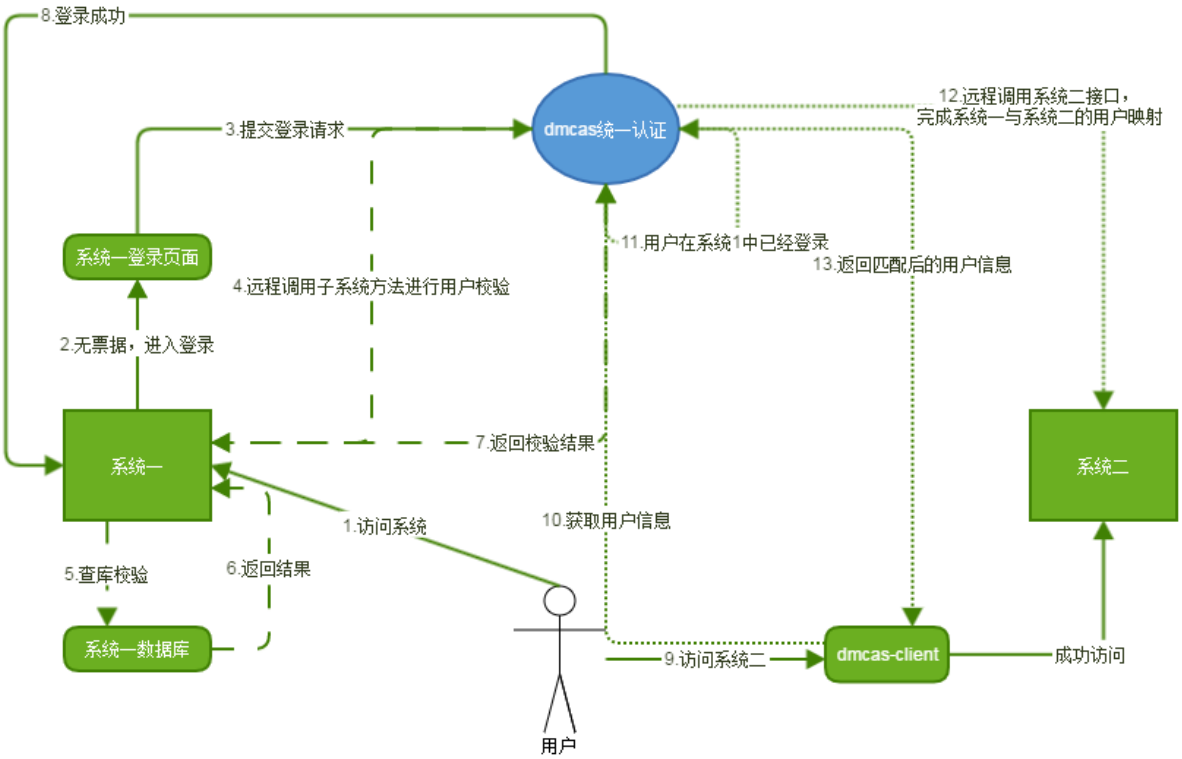


访问示意图

然而在面临多系统之间，不同的用户体系 [1: 指用户信息不一致，角色权限不一致的情况]时，如果还继续将用户的校验放在 *dmcas* 统一认证中心完成，那么将无法完成多系统不同用户之间的集成。

此时的结构应该如下图，各子系统独立完成自己的用户检验，角色权限控制。 ***dmcas* 只应该存储票据与信任列表，进行票据校验即可**。此时 *dmcas* 在中间还应 **协调** 完成各子系统的 **用户匹配**，具体如何 **匹配** 还是由子系统决定，*dmcas* 在中间充当一个 **调度者** 的角色。

这种结构就决定了 *dmcas* 与子系统之间只需要依赖公共接口，通过 *rpc* 调用完成。



多系统之间认证

dmcas

dmcas 作为 *sso* 中的协调者，主要的职责是：

1. 票据的生成与校验
2. 各子系统之间协调访问

Chapter 4. 名词说明

票据中心

这里指后台专门存放用户 *ticket* 信息的 *cache*，目前采用的是 *memcached* 进行存储。

信任列表

这里指各接入的子系统的域名列表。只有加入到了信任列表中的子系统，才可与其它子系统互相访问。

Chapter 5. 实现技术

spring

采用 *spring* 容器管理 *service bean*

mybatis

采用 *mybatis* 完成 *database* 的访问

springmvc

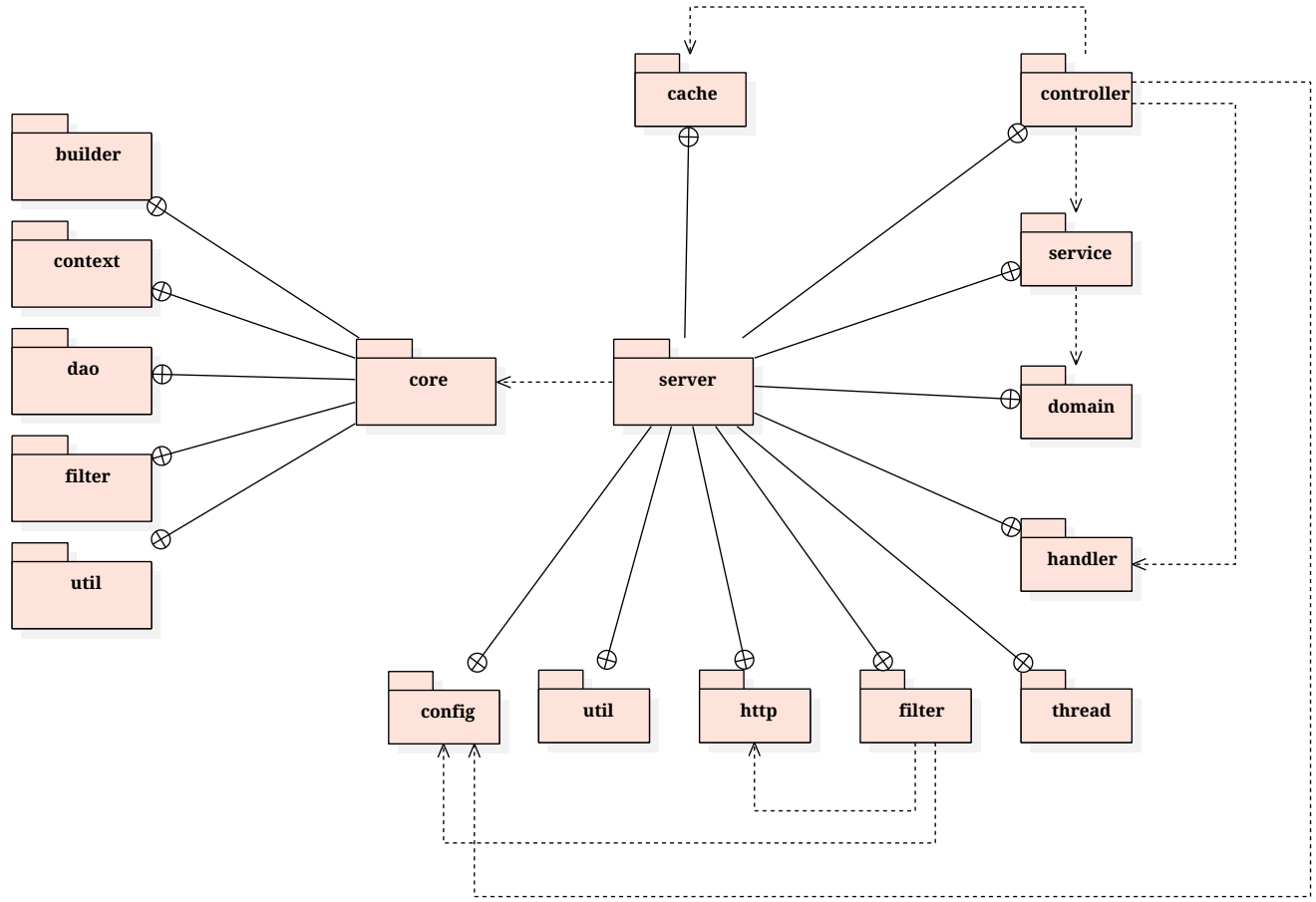
采用 *springmvc* 完成页面控制层的请求处理

memcached

采用 *memcached* 进行后台票据的存储管理

Chapter 6. 实现细节

6.1. 包图



包说明

1. core: 存放一些框架包
    - builder: 存放 *memcached client* 构造内容
    - dao: 存放 *dao* 层
    - filter: 存放框架上的一些 *web filter*
    - util: 存放工具类
2. server: 存放 *dmcas* 对外服务内容
    - cache: 一些缓存类
    - controller: 存放 *controller* 类等
    - service: 接口服务类
    - domain: 存放实体类
    - handler: 存放一些处理器
    - thread: 自定义线程类
    - filter: 存放服务上的 *web filter*

http: 存放 *request* 包装类等

- util: 存放 *web* 相关工具类
- config: 存放服务的一些配置信息

6.2. 功能模块说明

6.2.1. 登入

用户请求登录的逻辑主要在 *AuthController* ，此处主要提供的都是异步请求调用的方法（包括 *jsonp* ）

code

```
/**
 * 用户账号登录
 *
 * @param request  请求
 * @param response 响应
 * @return
 * @throws Exception
 */
@SuppressWarnings({"unchecked", "rawtypes"})
@RequestMapping("/login")
public ModelAndView login(HttpServletRequest request, HttpServletResponse response)
throws Exception {
    return doLogin(request, response, new LoginSucessHandler() {
        @Override
        public void handle(HttpServletRequest request, HttpServletResponse response,
Map<String, Object> params) throws Exception {
            response.sendRedirect(params.get("service").toString());
        }
    }, new LoginFailHandler() {
        @Override
        public void handle(HttpServletRequest request, HttpServletResponse response,
Map<String, Object> params) throws Exception {
            AuthResult authResult = (AuthResult) params.get("authResult");
            Domain domain = (Domain) params.get("domain");
            send2Login(request, response, authResult.getMessage(), domain);
        }
    });
}
```

由上面看出，登录的主要逻辑在于 *doLogin* 方法，此处只是 *doLogin* 方法使用 **策略模式** ，要求调用者传入了两个接口实现。

而在 *doLogin* 中，请求逻辑也相对简单，分为以下几步。

- 获取请求参数并校验
- *rpc* 调用子系统实现，完成 **用户名、密码** 的校验
-

- 调用 **登录成功/失败** 的子系统处理逻辑
- 生成票据信息，存入 **票据中心**
- 返回登录结果及票据

code

```
authResult = this.getAuthResult(request, domain); ①

if (authResult.isValid()) {
  // rpc调用子系统登录成功后处理实现
 ReturnMessage returnMessage = invokeLoginPostProcess(domain, request, authResult,
username, password); ②

 authid = AuthCache.set(username, ip, domain.getDomainName(), idCardNo); ③
 // 调用方自定义登录成功后处理逻辑
 sucessHandler.handle(request, response, params); ④
}else{
  // rpc调用子系统登录失败后处理实现
  ReturnMessage returnMessage = invokeLoginFailProcess(domain, request,
authResult,  username, password); ⑤
  params.put("returnMessage", returnMessage);
  // 调用方自定义登录失败后处理逻辑
  failHandler.handle(request, response, params); ⑥
}
```

① *rpc* 调用子系统完成用户名/密码的校验

② *prc* 调用子系统登录成功后的处理逻辑

③ 存储票据至票据中心

④ 调用自定义登录成功处理逻辑（策略模式）

⑤ *rpc* 调用子系统登录失败后处理实现

⑥ 调用自定义登录失败后处理逻辑（策略模式）

6.2.2. 登出

登出则非常简单，分两个步骤。

- 注销用户票据
- 重定向至登出页面

6.2.3. 票据验证

子系统的访问请求被 *cas-client* 拦截后，会通过模拟 *http* 请求完成校验。具体见 *cas-client* 中的 *AuthFilter* 的 *getAuthUser* 方法。

在 *dmcas* 中则提供了该 *http* 请求的服务方法。

AuthInfoController

```
/**
 * 根据authid获取认证的用户信息
 *
 * @param request
 * @param response
 * @return
 * @throws Exception
 */
@RequestMapping("/getAuthUser")
public ModelAndView getAuthUser(HttpServletRequest request,HttpServletResponse
response) throws Exception {
    // 用户原来的访问地址
    String authid = request.getParameter(CASConf.getCasTicketName());
    String ip = RequestUtil.getRemoteAddr(request);
    String username = AuthCache.get(authid, ip); ①
    if (StringUtils.hasText(username)) {
        response.getWriter().write(username);
    }
    return null;
}
```

① 直接从票据中心获取票据信息

cas-client 中通过此途径获取票据信息后，则只需要判断是否存在信息即可确定票据是否存在。

6.2.4. 子系统及跨系统访问

众所周知，基于 *cookie* 挟带票据的 *sso* 中会受限于 *cookie* 无法跨域访问。此时需要一个公共的 **父域**，而各子系统的域名则作为 **子域** 存在。但是在目前的环境下，通常不能方便地申请到这样的域名。此时需要寻找其它的方法来解决这个问题。

解决思路

1. 在登录的同时，对各子域设计跨域 *cookie* ，然后再访问各子系统时自然就能带 *cookie* 访问。

   优点

   实现简单且有效

   缺点

   - 登录时较慢（由于在设置跨域 *cookie* ）
   - 如果子系统设置 *cookie* 失败（如子系统正好宕机了），稍候访问子系统将会要求重新登录。
   - 不易与其它非子系统的第三方系统（有独立用户体系）集成。

2. 独立 *dmcas* 验证中心，各系统验证信息独自完成。 *dmcas* 作好协调者。此时基于用户来访 *ip* 来确定是同一用户的访问。

   优点

   方便集成其它系统，很容易拓展。

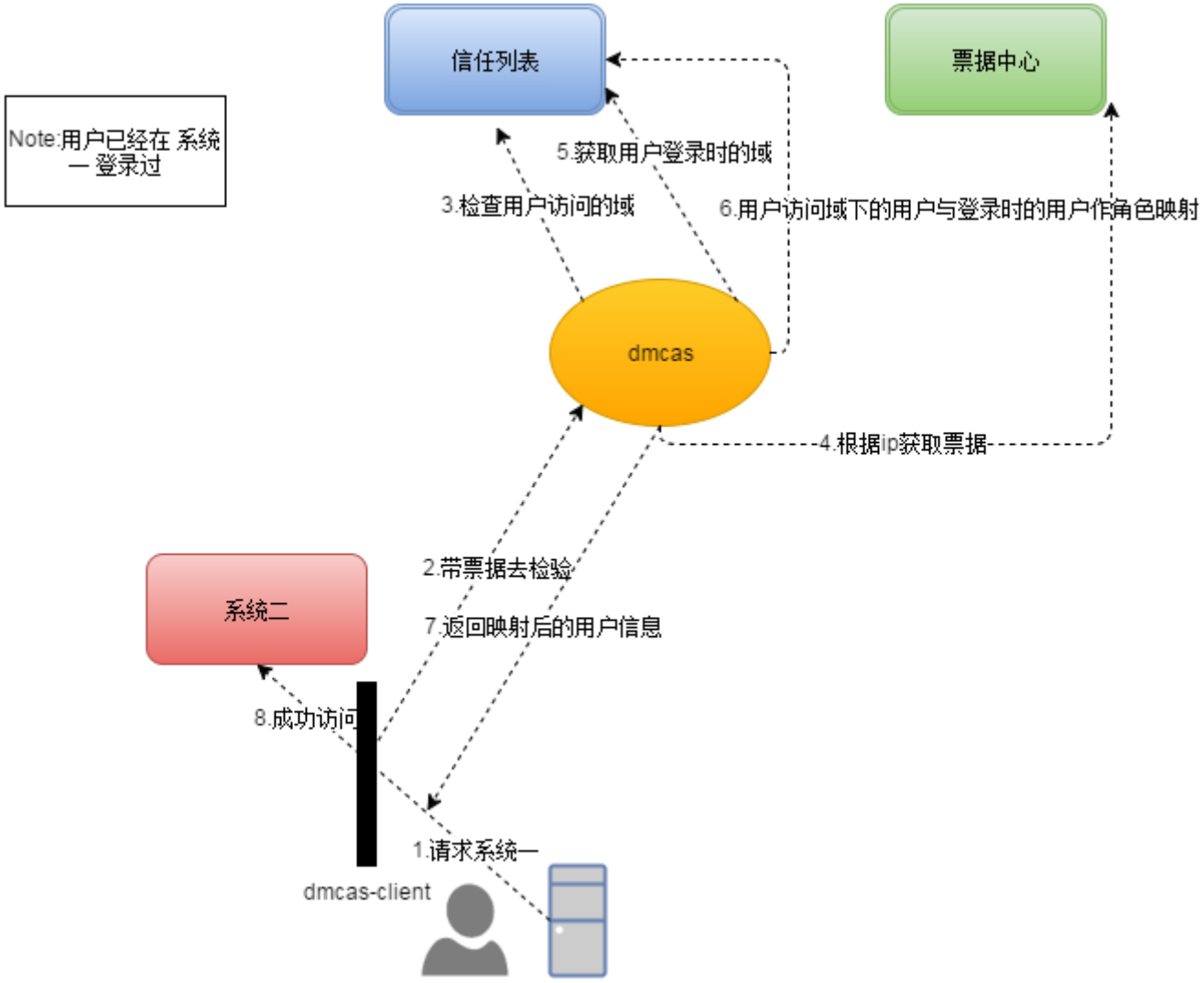
   缺点

   ip可被伪装，且已登录时关闭浏览器后再打开会自动登录。

现有的系统中 *dmcas* 采用的是第一种解决方案。

目前正在做的是升级为第二种解决方案。

实现细节



整个验证流程如上图所示。

实现代码

代码如下： 见 AuthInfoController 中的 getAuthInfo

AuthInfoController.getAuthInfo

```
String clientIp = request.getParameter(CASConf.getIpName()); ①

// 获取ip对应的用户信息
 String authInfo = AuthCache.getByClientIp(clientIp);
 if (!StringUtils.hasText(authInfo)) {
     logger.error("[clientIP:" + clientIp + "] no auth info found by clientIp " +
clientIp);
     return null; ②
 }

String host = request.getRemoteHost();
logger.debug("[clientIP:" + clientIp + "] getAuthInfo,the user request domain: " +
host);

Domain requestDomain = DomainCache.get(host); ③
```

```
String[] authArr = authInfo.split("&");
String userCode = authArr[1];
String loginHost = authArr[2];
Domain loginDomain = DomainCache.get(loginHost); ④

// 获取该用户在登录时的域对应的系统中的用户信息
logger.debug("[clientIP:" + clientIp + ", userCode:" + userCode + "] begin get login
domain(" + loginDomain.getDomainName() + ")'s user info!");
UserMapperService userMapperService = ConsumerCache.get(loginDomain,
UserMapperService.class);
if (userMapperService == null) {
logger.error("can't find login domain's consumer from registry center! domain name:" +
loginDomain.getDomainName() + ", group:" + loginDomain.getGroupName());
return null;
}  ⑤

// 根据该用户信息获取其在被请求的域对应的系统中的用户信息
 logger.debug("[clientIP:" + clientIp + ", userCode:" + userCode + "] begin get " +
user.getUser_code() + " mappingUser from requestDomain " + requestDomain.
getDomainName());
 userMapperService = ConsumerCache.get(requestDomain, UserMapperService.class);
 if (userMapperService == null) {
     logger.error("can't find request domain's consumer from registry center! domain
name:" + requestDomain.getDomainName() + ", group:" + requestDomain.getGroupName());
     return null;
 }

 User mapperUser = userMapperService.getUser(user);

 // 未返回映射用户
 if (mapperUser == null || !StringUtils.hasText(mapperUser.getUser_code())) {
     logger.error("[clientIP:" + clientIp + ", userCode:" + userCode + "] current
login user:" + user.getUser_code() + " can't visit from login domain " + loginDomain
.getDomainName() + " to request domain:" + requestDomain.getDomainName());
     return null;
 }  ⑥

 response.getWriter().write(sb.toString()); ⑦
```

① 获取 *client* 拦截时获取到的用户访问 ip ，这里可能会有问题，如果对方通过 *nginx* 代理的请求，且未提供一个 *header* 保存用户的 *ip* 。

② 校验用户 *authInfo* 信息是否存在

③ 获取用户正要请求的域

④ 从 *authinfo* 中获取用户登陆时存储的域地址

⑤ 获取用户登录时的用户信息。 *rpc* 调用

⑥ 获取用户在目前被访问的系统中的映射后的用户信息。 *rpc* 调用

⑦ 将获取到的映射用户信息回写

get user ip address

```
public static String getClientIpAddr(HttpServletRequest request) {
  String ip = request.getHeader("x-forwarded-for");
  if (ip == null || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {
    ip = request.getHeader("Proxy-Client-IP");
  }
  if (ip == null || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {
    ip = request.getHeader("WL-Proxy-Client-IP");
  }
  if (ip == null || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {
    ip = request.getRemoteAddr();
  }
  return ip;
}
```

dmcas-client

dmcas-client 作为 *sso* 校验中的客户端，用来拦截各子系统发送的请求，在请求抵达子系统前完成访问 *dmcas* 完成用户信息校验工作。

正因为如此， *dmcas-client* 必须部署在各子系统中，通常也是以一个 *jar* [2: 方便集成其它系统] 包的方式存在。

Chapter 7. 部署使用说明

7.1. 获取程序包

第三方需要联系 **湖北公安云** 项目组获取 *dmcas-client* 的程序 *jar* 包。

7.2. 编写配置文件

如新建 *cas-client.properties* 文件。编写如下内容：

cas-client.properties

```
casURL=http://localhost:8080/dmcas ①
loginURL=http://localhost:8080/portal/login.do ②
exceptPathPrefix=/ui/;/services/ ③
publicPathPrefix=/common/;/portal/;/wmf/casAuth ④
```

① 指定 *dmcas* 验证中心的访问地址，请填写实际地址。

② 指定用户未登录后需要跳转到的地址，即配置登陆地址。

③ 希望加入例外的 *url* 匹配规则。 **注意，不会获取用户信息，检测为例外请求后，会直接放行。**

④ 配置为公共请求 *url* 的匹配规则。
注意，会尝试获取用信息，不论是否获取得到，只要是公共请求，都会放行。

7.3. 配置过滤器

在子系统的 *web.xml* 文件中添加如下过滤器，请注意 *filter* 的顺序。具体该 *filter* 放在哪个位置视各系统而定。

web.xml

```
<!-- 单点登录客户端 -->
<filter>
  <filter-name>CASAuthFilter</filter-name>
  <filter-class>com.dm.cas.client.filter.AuthFilter</filter-class>
  <init-param>
    <param-name>configMethod</param-name>
    <param-value>file</param-value> ①
  </init-param>
  <init-param>
    <param-name>configFilePath</param-name>
    <param-value>/WEB-INF/config/cas-client.properties</param-value> ②
  </init-param>
</filter>
<filter-mapping>
  <filter-name>CASAuthFilter</filter-name>
  <url-pattern>*.do</url-pattern> ③
</filter-mapping>
<filter-mapping>
  <filter-name>CASAuthFilter</filter-name>
  <url-pattern>*.html</url-pattern> ④
</filter-mapping>
```

① 指定为 *file* 表示从配置文件中读取配置信息。

② 指定配置文件的路径，此处应该配 *web* 工程的相对路径。

③ 指定请求拦截的 *url* 规则，例如此处为 *.do

④ 指定请求拦截的 *url* 规则，例如此处为 *.html

7.4. 添加域至信任列表

本来完成上述几步后，应该整个过程就结束了。这里最后一步是 *dmcas* 升级方案中需要做的。只有子系统的域在信任列表中时，才允许通过 *dmcas* 进行单点登陆。

所以最后需要联系 **湖北公安云项目组** 将子系统的域添加至信任列表中即可。

Chapter 8. 实现讲解说明

整个 *dmcas-client* 其实内容并不多，只有一个核心的过滤器和几个辅助工具类。

其实 *dmcas-client* 本身并不需要完成多少功能，它只需要拦截用户请求，请求 *dmcas* 完成用户信息校验，转发或重定向用户请求即可。

8.1. AuthFilter核心过滤器

AuthFilter 是 *javax.servlet.Filter* 的一个实现类。所以也就实现了该接口的三个方法。

- init
- doFilter
- destory

在 *init* 方法中，主要就是完成配置参数的加载工作了。此处是分为 **从文件加载** 和 **从数据库加载** 两种方式了。

init

```
public void init(FilterConfig filterConfig) throws ServletException {
// 获取配置方式
String configMethod = filterConfig.getInitParameter("configMethod");
// 获取配置文件路径
String configFilePath = filterConfig.getInitParameter("configFilePath");
// 使用配置文件cas-client.properties
if ("file".equalsIgnoreCase(configMethod)) {
    try {
        String filePath = filterConfig.getServletContext().getRealPath(configFilePath
);
        CASConf.init(filePath); ①
    } catch (Exception e) {
        logger.error("init AuthFilter from file error!", e);
    }
    return;
}
// 使用数据库参数表
if ("db".equalsIgnoreCase(configMethod)) {
    try {
        String filePath = filterConfig.getServletContext().getRealPath(configFilePath
);
        CASConf.initFromDB(filePath); ②
    } catch (Exception e) {
        logger.error("init AuthFilter from database error!", e);
    }
}
}
```

① 一般，第三方集成进来时采用配置文件的方式。

② 自己的子系统集成进来时则采用数据库加载的方式,方便统一管理，多子系统时不需要维护多个配置文件。

doFilter方法中则包含了主要的控制逻辑。大致分为以下几个步骤：

1. 获取用户将要请求的 *url*
2. 检查是否为例个请求，是则放行
3. 从 *cookie* 中获取用户票据，去 *dmcas* 校验用户票据是否合法
4. 根据 *ip* 检验用户是否存在票据信息
5. 检查请求是否为公共请求
6. 无用户信息并且不是公共请求，则重定向至登录页面。否则就放行。

doFilter

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
        throws IOException, ServletException {

    // TODO 设置线程的当前request,user
    HttpServletRequest hrequest = (HttpServletRequest) request;
    HttpServletResponse hresponse = (HttpServletResponse) response;

    String path = hrequest.getServletPath();
    if (StringUtils.hasText(hrequest.getPathInfo())) {
        path += hrequest.getPathInfo();
    }

    // 如果为例外请求，直接通过，不通过cas获取当前登录用户
    if (this.isExceptRequest(path)) {
        chain.doFilter(request, response);
        return;
    }

    // 通过cas获取当前登录用户
    // 基于cookie
    String username = null;
    String authid = CookieUtil.getCookie(hrequest, CASConf.getCasTicketName());

    // 用户浏览器存在cookie信息
    if (StringUtils.hasText(authid)) {
        username = this.getAuthUser(authid); ①
    } else {
        // 不存在cookie信息，根据用户ip获取票据信息
        String clientIp = IPUtil.getClientIpAddr(hrequest);
        if (StringUtils.hasText(clientIp)) {
            // 如果根据clientIp获取到用户信息，则认为请求通过
            String authInfo = this.getAuthInfo(clientIp); ②
            if (StringUtils.hasText(authInfo)) {
                String[] authInfoArr = authInfo.split("&");
                if (authInfoArr.length >= 2) {
```

```
                    authid = authInfoArr[0];
                    username = authInfoArr[1];

                    // 往用户浏览器回写当前域的cookie
                    this.setCookie(hresponse, authid, username);
                } else {
                    logger.error("invalid auth info:" + authInfo);
                }
            }
        }
    }

    // 取消基于请求参数方式, modify by zxb
    // TODO 登录校验
    boolean isPublicRequest = this.isPublicRequest(hrequest, path);

    if (StringUtils.isEmpty(username) && !isPublicRequest) {
        this.redirectView(hrequest, hresponse);
        return;
    }

    chain.doFilter(new CasHttpServletRequest(hrequest, username), response);
}
```

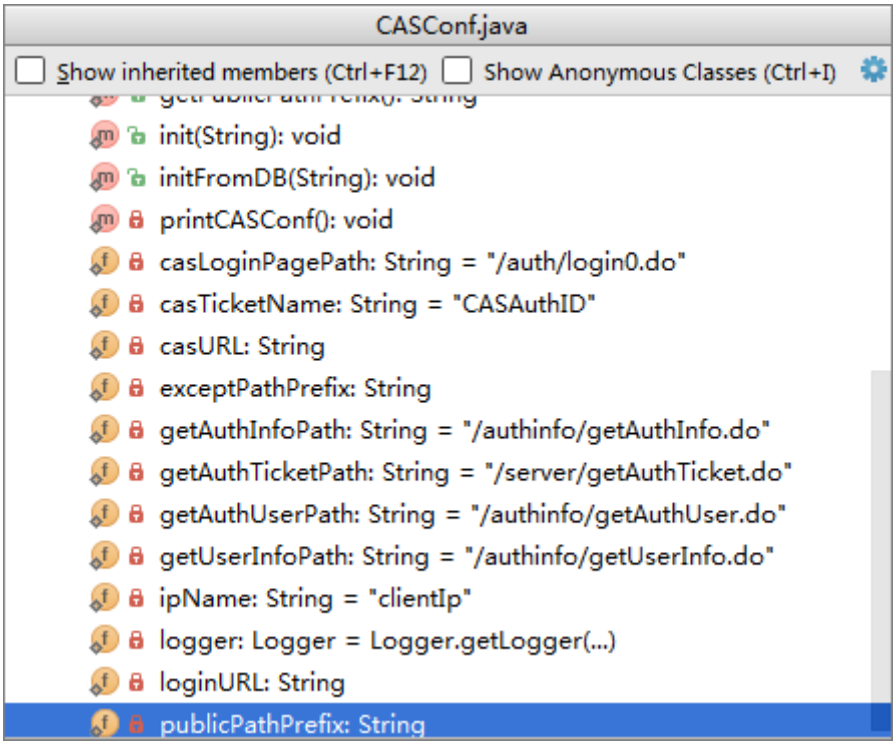
① 模拟发送 *http* 请求至 *dmcas* 校验用户票据有效性

② 模拟发送 *http* 请求至 *dmcas* 根据 *ip* 获取用户信息

destory 方法在 *Filter* 中一般是作为释放资源而存在的，此处则只留了一个空实现。

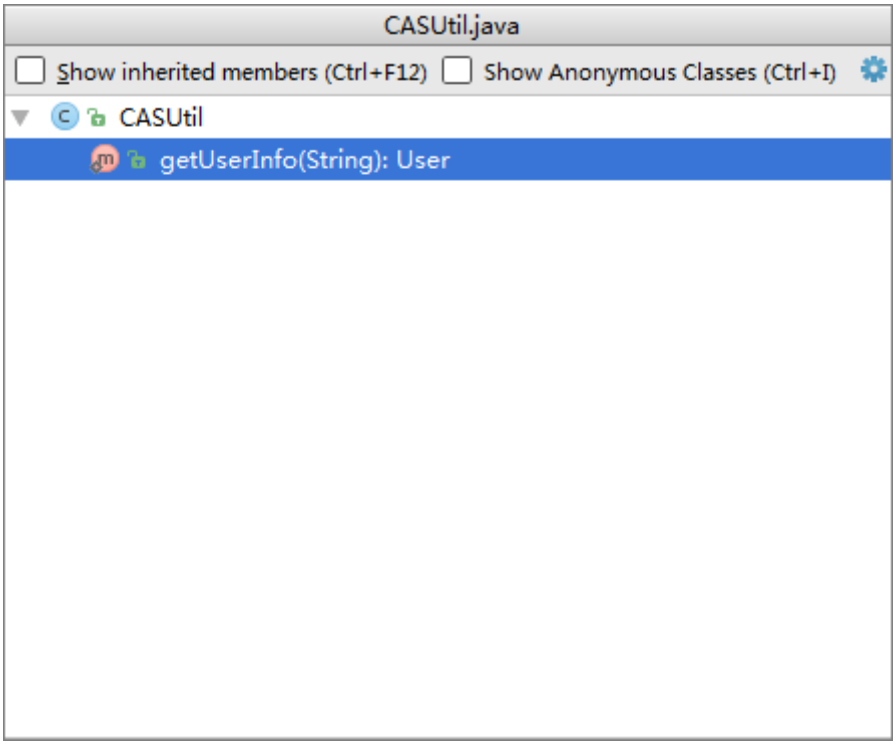
8.2. CASConf配置

CASConf 作为一个配置类，保存了 *AuthFilter* 中使用到的一些配置信息。同时也作为工具类，提供了加载配置信息的方法。



8.3. CASUtil 工具

与第三方系统集成时，通常他们也希望能根据票据获取到具体的用户信息。而在 *CASUtil* 中就提供了该方法。



此处获取用户信息，同样是模拟发送 *http* 请求到 *dmcas* ，通过参数 *casauthid* (即票据)获取用户的具体信息。

CASUtil

```
/**
```

```
 * 用户登录后用户信息（不包含密码）
 *
 * @param authid
 * @preserve
 * @return
 */

public static User getUserInfo(String authid) {
  // 空值直接返回null
  if (StringUtils.isEmpty(authid)) {
    return null;
  }
  // 建立http连接并获取信息
  String userinfo = null;
  User user = new User();
  try {
    // 建立connection
    String urlString = CASConf.getCasURL() + CASConf.getGetUserInfoPath() + "?" +
CASConf.getCasTicketName()
        + "=" + authid + "_userinfo";
    System.out.println("CASAuthID 工具:" + authid);
    URL url = new URL(urlString);
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    // 设置建立连接超时时间
    connection.setConnectTimeout(5000);
    connection.setRequestMethod("POST");
    connection.setDoOutput(true);
    connection.setDoInput(true);
    connection.setUseCaches(false);
    // 获取响应
    connection.connect();
    InputStream inputStream = connection.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
    userinfo = reader.readLine();
    System.out.println("userinfo 工具encode:" + userinfo);
    userinfo = URLDecoder.decode(userinfo, "utf-8");
    JSONObject jsonObject = JSONObject.fromObject(userinfo);
    user = (User) JSONObject.toBean(jsonObject, User.class);

    // 关闭连接
    reader.close();
    inputStream.close();
    connection.disconnect();
  } catch (Exception e) {
    e.printStackTrace();
  }
  return user;
}
```

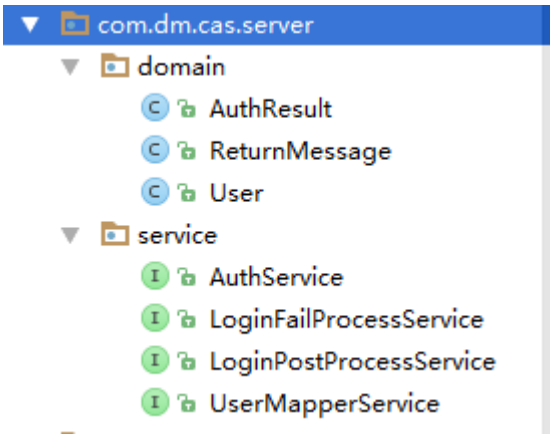
dmcas-validation

在 *dmcas* 升级后，原先所有放在 *dmcas* 中的用户信息校验逻辑及其它一些业务逻辑将全部从 *dmcas* 中剥离。因为这些逻辑不属于 *dmcas* 处理，应该是各子系统自己独有的一些逻辑，即应该子系统自己独立处理。

当这些校验逻辑被剥离出来后，就形成了 *dmcas-validation* 。它作为一个默认的校验实现（仅限于达梦公安团队的项目使用）。当然，各子系统也完全可以实现自己的校验逻辑，只需要遵循 *dmcas* 中指定的那些接口规范即可。

Chapter 9. 接口

在 *dmcas* 独立出来以后，暴露了如下一些接口，子系统实现这些接口后，*dmcas* 将通过 *rpc* 的方式调用子系统的具体实现来完成校验及其它业务逻辑。



interfaces

AuthService

校验服务接口，根据用户登录请求的用户名密码，子系统提供校验服务即可。注意，在子系统自己的登陆页面登陆时才会调用该接口。 必须实现

AuthService

```
/**
 * 检验用户信息服务
 *
 * @author zxb
 * @version 1.0.1
 */
public interface AuthService {

    /**
     * 校验用户名密码是否正确
     *
     * @param username 用户名
     * @param password 用户明文密码
     * @return AuthResult 校验结果
     * @see com.dm.cas.server.domain.AuthResult
     */
    public AuthResult validate(String username, String password);

}
```

LoginPostProcessService

登录成功后的处理接口，子系统实现该接口以便完成一些登陆成功后的处理逻辑。例如，记录登陆日志等。 可选实现

LoginPostProcessService

```
/**
 * 用户登录成功后，子系统的处理服务。
 * <p/>
 * 子系统可以实现该接口，以便在用户登录成功后，计入登录日志等信息。
 * @author zxb
 * @version 1.0.1
 * Created by zxb on 2016/3/13.
 */
public interface LoginPostProcessService {

    /**
     *
cas验证用户登录成功后，会调用该接口。子类实现该接口后可加入自己的一些处理逻辑。
     * @param params 参数列表
     *               <ul>
     *                  <li>user 登录用户信息</li>
     *                  <li>ip 登录用户客户端ip地址</li>
     *               </ul>
     * @throws Exception
     * @return ReturnMessage 返回的消息，子类如果不返回消息及状态码。
cas将使用默认消息及状态码。
     */
    ReturnMessage process(Map<String, Object> params) throws Exception;

}
```

LoginFailProcessService

登陆失败后的处理逻辑，子系统实现该接口后可在用户登陆失败后加入自己的一些处理逻辑。 可选实现

LoginFailProcessService

```
/**
 * @author zxb
 * @version 1.0.1
 * Created by zxb on 2016/3/14.
 */
public interface LoginFailProcessService {

    /**
     *
cas验证用户登录失败后，会调用该接口。子类实现该接口后可加入自己的一些处理逻辑。
     * @param params 参数列表
     *               <ul>
     *                  <li>user 登录用户信息</li>
     *                  <li>ip 登录用户客户端ip地址</li>
     *               </ul>
     * @throws Exception
     * @return ReturnMessage 返回的消息，子类如果不返回消息及状态码。
cas将使用默认消息及状态码。
     */
    ReturnMessage process(Map<String, Object> params) throws Exception;
}
```

UserMapperService

用户映射接口，用户在登陆本系统或第三方系统后，再去访问其它第三方系统时，需要提供一个用户映射的服务。子系统必须实现该接口，且提供用户的匹配映射。 必须实现

UserMapperService

```
/**
 * 用户接口
 * @author zxb
 * @version 1.0.1
 * Created by zxb on 2016/3/16.
 */
public interface UserMapperService {

    /**
     *
根据用户账号获取该用户的信息，子系统实现该接口，便于各系统之间完成用户映射。此处只
会被 dmcas 调用，且只会用于各系统间匹配校验。
     * @param userCode 用户账号
     * @return user 用户信息
     * @throws Exception
     */
    public User getUser(String userCode) throws Exception;

    /**
     * 根据其它系统提供的用户信息，返回本系统对应的用户信息。
     * @param user
     * @return
     * @throws Exception
     */
    public User getUser(User user) throws Exception;
}
```

Chapter 10. 默认实现

在 *dmcas-validtion* 中完成的，只是对上述的几个接口的一个实现而已。

10.1. AuthService

默认的用户名密码校验，采用的是对密码 *md5* 后进行一个比较的校验。实现逻辑很简单，把密码 *md5* 一把后和库中的 *md5* 后的密码进行一个等值比较。

MD5AuthService

```
public AuthResult validate(String username, String password) {
logger.info("begin validate user login info! username:" + username);
AuthResult authResult = new AuthResult();
if (!StringUtils.hasText(username) || !StringUtils.hasText(password)) {
    authResult.setValid(false);
    authResult.setMessage("用户名或密码不能为空！");
    return authResult;
}

Connection conn = null;
PreparedStatement ps = null;
ResultSet rs = null;
try {
    // TODO 此处应使用连接池实现，并发登录情况下此实现会有问题。
    conn = dataSource.getConnection();
    ps = conn.prepareStatement(sql);
    ps.setString(1, username);
    rs = ps.executeQuery();
    String dbPassword = null;
    String flag = null;
    while (rs.next()) {
        dbPassword = rs.getString(1);
        flag = rs.getString(2);
    }
    if (!StringUtils.hasText(dbPassword)) {
        authResult.setValid(false);
        authResult.setMessage("用户不存在！");
        return authResult;
    }
    if (!dbPassword.equalsIgnoreCase(EncryptUtil.md5Degest(password))) {
        authResult.setValid(false);
        authResult.setMessage("用户或密码不正确！");
        return authResult;
    }
    if (!("1".equals(flag))) {
        authResult.setValid(false);
        authResult.setMessage("帐号已被禁用！");
        return authResult;
```

```
    }
    //获取登录用户信息
    User user = userService.queryByCodePassword(username, password);
    authResult.setUser(user);
    //验证是否成功
    authResult.setValid(true);
    return authResult;
} catch (Exception e) {
    logger.error("validate user:" + username + " error!", e);
    // 返回信息
    authResult.setValid(false);
    authResult.setMessage("验证失败！");
    return authResult;
} finally {
    this.release(rs, ps, conn);
    logger.info("begin validate user login info! username:" + username + ", validate:"
+ authResult.isValid() + ",msg:" + authResult.getMessage());
}
}
```

10.2. LoginPostProcessService

在用户登陆成功后，往往需要记录用户登陆行为的日志，刷新用户访问量等等。

LoginPostProcessServiceImpl 实现了该接口，这里则是添加了一些子系统独有的一些逻辑。

接口方法返回的 *ReturnMessage* 可以决定登陆成功后最终返回给客户端的信息，如 *ReturnCode* 和 *ReturnMessage*

LoginPostProcessServiceImpl

```
public class LoginPostProcessServiceImpl implements LoginPostProcessService {

    private Logger logger = Logger.getLogger(this.getClass());

    /**
     * 登录日志类型
     */
    private final String LOGIN_LOG_TYPE = "1001";

    /**
     * 日志服务接口
     */
    private LogService logService;

    /**
     * 参数列表
     */
    private ParamCache paramCache;
```

```
    public void setLogService(LogService logService) {
        this.logService = logService;
    }

    public void setParamCache(ParamCache paramCache) {
        this.paramCache = paramCache;
    }

    @Override
    public ReturnMessage process(Map<String, Object> params) throws Exception {
        logger.info("begin login success process!");
        ReturnMessage returnMessage = null;
        if (params != null && params.size() > 0) {
            // 记录用户登录日志
            User user = (User) params.get("user");
            String ip = params.get("ip").toString();

            logger.debug("create login log, user:" + user.getUser_code() + ", ip:" +
ip);
            this.logService.createLog(user, ip, LOGIN_LOG_TYPE);

            // 更新门户登陆统计信息
            this.logService.update();

            // 验证用户密码是否为初始化密码
            String username = params.get("username").toString();
            String password = params.get("password").toString();
            String initial_password = paramCache.getValue("INITIAL_PASSWORD");
            if (!StringUtils.hasText(initial_password)) {
                logger.error("initial_password is empty,verify password will not
effect!please check the table wmf_param!");
            } else {
                if (EncryptUtil.md5Degest(initial_password).equalsIgnoreCase(password
)) {
                    returnMessage = new ReturnMessage();
                    returnMessage.setReturnCode("1");
                    returnMessage.setReturnMsg("您的密码还是初始化密码，请及时修改！"
);
                    logger.info("verify success!warn, the user " + username + "'s
password is initial_password!");
                }
            }
        }
        logger.info("end login success process!");
        return returnMessage;
    }
}
```

10.3. LoginFailProcessService

LoginFailProcessService 同 *LoginPostProcessService* 接口类似，它在用户登陆失败时会调用。

LoginFailProcessServiceImpl 实现了该接口，添加了一些日志记录的逻辑。

LoginFailProcessServiceImpl

```
/**
 * 登录失败后的处理逻辑
 * Created by zxb on 2016/3/14.
 */
public class LoginFailProcessServiceImpl implements LoginFailProcessService {

    private Logger logger = Logger.getLogger(this.getClass());

    private MistakeLogService mistakeLogService;

    /**
     * 登录日志类型
     */
    private final String LOGIN_LOG_TYPE = "1001";

    public void setMistakeLogService(MistakeLogService mistakeLogService) {
        this.mistakeLogService = mistakeLogService;
    }

    @Override
    public ReturnMessage process(Map<String, Object> params) throws Exception {
        logger.info("begin login fail process!");
        if (params != null && params.size() > 0) {
            // 创建登录失败日志
            User user = (User) params.get("user");
            String ip = params.get("ip").toString();

            logger.debug("login fail user:" + (user == null ? "" : user.getUser_code(
)) + ", ip:" + ip);
            mistakeLogService.createLog(user, ip, LOGIN_LOG_TYPE);
        }
        logger.info("end login fail process!");
        return null;
    }
}
```

10.4. UserMapperService

UserMapperService
作为各子系统间互相访问时最关键的一个服务接口。它要求子系统实现它时，需要提供用户映射的服务。

UserMapperServiceImpl 实现了该接口的两个方法。

UserMapperServiceImpl

```
/**
 * 用户映射服务实现
 *
 * @author zxb
 * @version 1.0.1
 *          Created by zxb on 2016/3/17.
 */
public class UserMapperServiceImpl implements UserMapperService {

    private Logger logger = Logger.getLogger(this.getClass());

    /**
     * 用户操作接口
     */
    private UserService userService;

    /**
     * 设置用户操作接口
     */
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    @Override
    public User getUser(String userCode) throws Exception {
        if (StringUtils.hasText(userCode)) {
            User user = this.userService.queryByCode(userCode);
            return user;
        }
        return null;
    }

    @Override
    public User getUser(User user) throws Exception {
        if (user == null || !StringUtils.hasText(user.getUser_code())) {
            logger.error("user is null or user_code is empty!");
            return null; ①
        }

        String userCode = user.getUser_code();
        String idCardNo = user.getUser_sfzh();
        logger.debug("other system's user, user_code " + userCode + ", idCardNo " +
idCardNo);

        // 非普通用户，如一般的管理用户
        if (!StringUtils.hasText(idCardNo)) {
            logger.error("current user " + userCode + " does'nt has idCardNo!");
            // TODO 后期再看如何处理，或者抛出一个接口，让各个我们做的系统提供实现。
            return null; // 阻止无身份证号的用户进行访问
```

```
            //return user; 测试使用
        }

        // 本系统中映射用户不存在时，则创建用户
        User mappingUser = this.userService.queryByNameSfzh(userCode, idCardNo);
        if (mappingUser == null) {
            mappingUser = this.userService.createPkiUser(userCode, userCode, idCardNo,
"PKI001");
        }

        return mappingUser; ②
    }
}
```

① 没有匹配用户时，直接返回 *null* 则可以。

② 存在用户，则返回匹配的用户即可。

dmga-portal-hb

Chapter 11. pki登录

pki: *Public Key Infrastructure* 即 **公钥基础设施**

在本系统中，通过 **pki** 进行登录，主要是获取 **pki** 中的用户信息。

所以登录也就是分为两步.

1. 获取 **pki** 证书代表的用户信息，警号、身份证号
2. 使用 警号、身份证号 进行登陆

11.1. 引入类库

在集成 **pki** 登录前，需要在 **portal** 工程下引入以下类库。

Commons-codec-1.3.jar	字节编码库；
Commons-httpclient-3.1.jar	Http客户端；
Commons-logging-1.1.jar	日志接口；
IAS_SDK_2.1_JDK13.jar	（可选）
用于兼容2.1及以前版本的支持包，如果使用2.1以上版本，可以不引用。	IAS_SDK_2.2.0_JDK13.jar
开发工具包；	Log4j-1.2.14.jar
日志包；	jit-cinas-saml11-1000-jdk13.jar
SAML 1.1协议支持包；	jit-cinas-commons-1000-jdk13.jar
产品公共类库；	Xmlsec-1.3.0.jar
XML签名加密包；	log4j-1.2.14.jar
第三日志包；	J2EE_Agent_2_2_2_JDK13.jar
Agent功能包	bcprov-jdk13-143.jar

11.2. pki.jsp编写

编写 *pki.jsp* 完成用户信息的获取。

```
<%@ page language="java" pageEncoding="utf-8"%>
<%@ page contentType="text/html; charset=utf-8"%>
<%@ page import="cn.com.jit.assp.ias.sp.saml11.SPUtil"%>
<%@ page import="cn.com.jit.assp.ias.sp.saml11.SPConst"%>
<%@ page import="cn.com.jit.assp.ias.saml.saml11.*"%>
<%@ page
    import="cn.com.jit.assp.ias.saml.saml11.SAMLAttributes.SAMLAttributeName"%>
<%@ page import="cn.com.jit.assp.ias.principal.UserPrincipal"%>
<%@ page import="java.text.SimpleDateFormat"%>
```

```
<%@ page import="java.util.*"%>
<%@ page import="cn.com.jit.cinas.commons.util.StringUtils"%>
<%
    boolean success = false;
    UserPrincipal p = null;
    SimpleDateFormat formatter = new SimpleDateFormat(
            "yyyy    MM    dd     HH:mm:ss");

    p = SPUtil.getUserPrincipal(request);

    success = (p == null ? false : true);
    String user_name = "";
    String user_sfzh = "";

    SAMLAttributes attrs = (SAMLAttributes)p.getAttribute(SPConst
.KEY_SAML_ATTR_STATEMENT_ATTRIBUTES);
    String info = "";
    if (attrs != null) {
        List ls = attrs.getAttributeNames();
        for (int i = 0; i < ls.size(); i++) {
            SAMLAttributeName name = (SAMLAttributeName) ls.get(i);
            //
            String parentValueName = name.getParentName();
            List values = null;
            if (!StringUtils.isBlankOrNull(parentValueName))
                //     "          "
                values = attrs.getAttributeValue(name.getName(),name.getNamespace()
,parentValueName);
            else
                values = attrs.getAttributeValue(name.getName(),name.getNamespace());
            if(values!= null&&values.size()>0) {
                info = values.get(0).toString();
            }
        }
    }
    String pki_info = "";
    if(info != ""){
        pki_info = info.split(",")[0].split("=")[1];
    }
    if(pki_info.split(" ").length>=2){
        pki_info = pki_info.split(" ")[0]+"_"+pki_info.split(" ")[1];
    }

    response.sendRedirect("../information.shtml?status_flag="+success+"&pki_info="
+java.net.URLEncoder.encode(pki_info)); ①
%>
```

① 注意，重定向首页时，此时已经获取到了用户信息，即参数中的 ***pki_info***

11.3. 配置过滤器

上面编写的 *pki.jsp* 中获取到的用户信息，实际上是在 *filter* 中完成后再转给 *pki.jsp* 的。

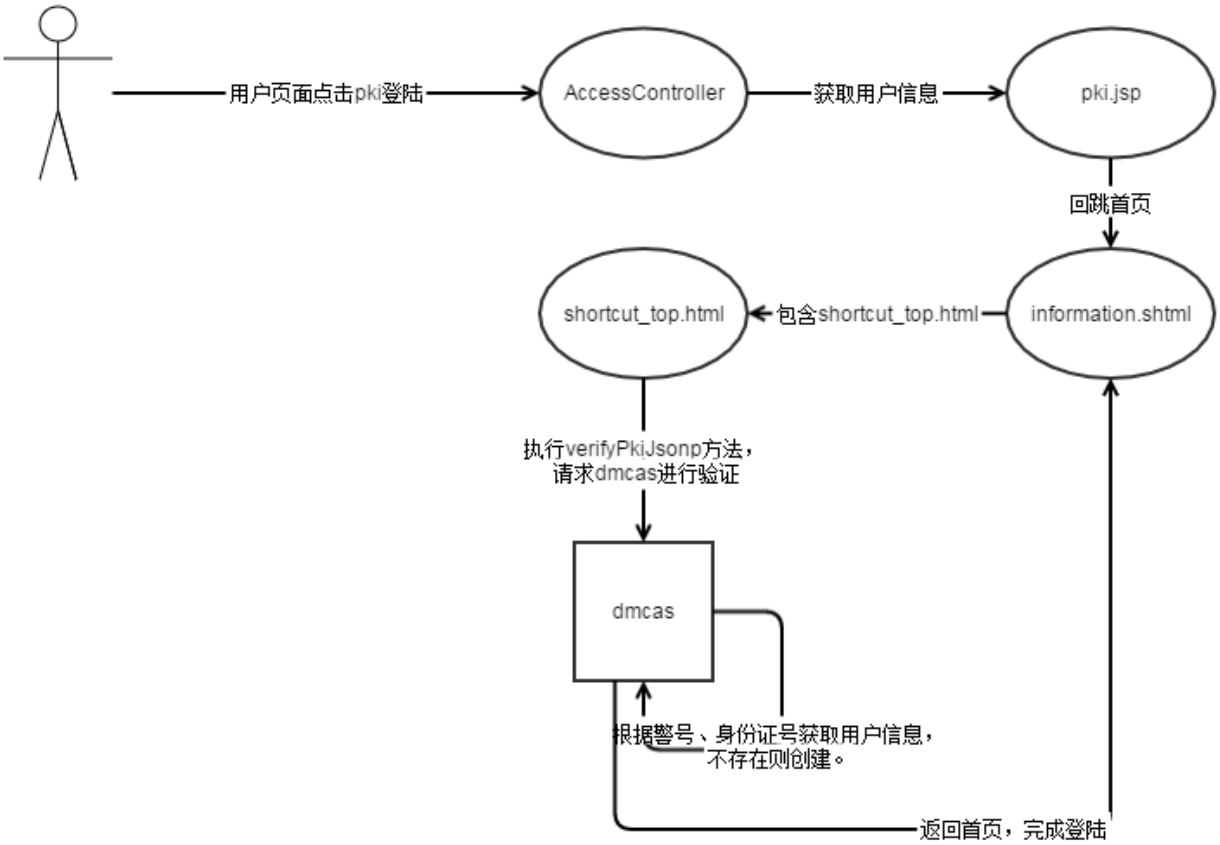
```
<filter>
        <filter-name>AccessController</filter-name>
        <filter-class>
            cn.com.jit.assp.ias.sp.saml11.AccessController
        </filter-class>
        <init-param>
            <param-name>SPConfig</param-name>
            <param-value>
                /WEB-INF/config/JITAgentConfig.xml ①
            </param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>AccessController</filter-name>
        <url-pattern>/html/pki/pki.jsp</url-pattern> ②
    </filter-mapping>
```

① 指向了对应的 **pki** 的配置文件，里面包含了验证服务器地址等信息。

② 拦截了 ***pki.jsp*** 。这里需要注意的是，如果 ***pki.jsp*** 所在的 *uri* 如果不在 **吉大正元信任列表** 中。此时访问会报错提示不在信任列表中。

11.4. 登录请求

请求流程如下图



由图可以看出，*pki* 只是为了获取用户信息，然后再发请求完成用户登陆。

dmga-dubbo-wsdl

Chapter 12. 简述

12.1. 概要说明

在基于 *rpc* 框架 *dubbo* 的基础上构建的 *soa* 并不能直接接入第三方的 *webservice* 服务。而公安内部的接口通讯则主要是以 *webservice* 进行通讯，此时就需要开发一个工具包，完成 *webservice* 服务的接入。

12.2. 实现框架

dubbo

dubbo 是阿里开源出来的一个优秀的 *rpc* 框架，而且该框架被许多大的电商网站使用。本工程主要是基于 *dubbo* 之上，对其进行部分扩展。

cxf

cxf 是一个优秀的 *webservice* 框架。本工程将基于它发布 *webservice* 服务。

axis

axis 同样也是一个 *webservice* 框架。由于公安部使用的是 *axis1.4* 发布的 *webservice* 服务，为了支持这类 *webservice* 服务，所以同样也引入了 *axis1.4* 来完成对该类服务的调用。

12.3. 实现概览

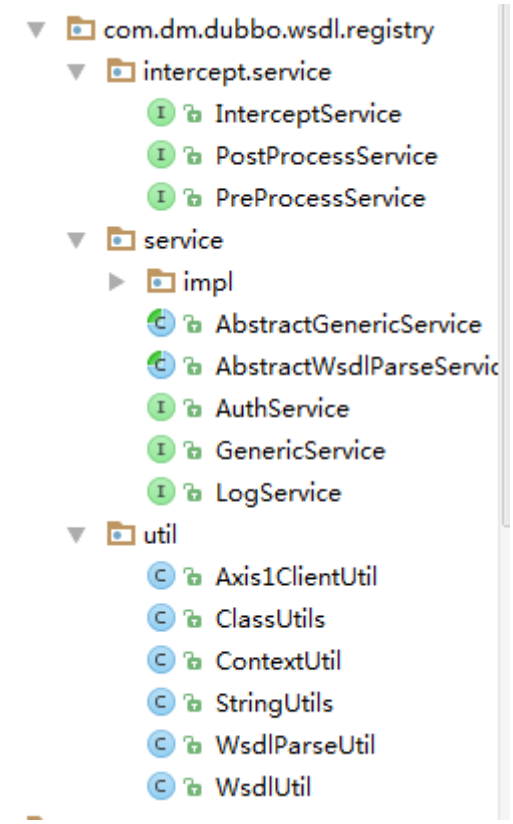
本工程在第三方 *webservice* 服务与 *dubbo* 提供的 *webservice* 服务之间充当了一个代理。

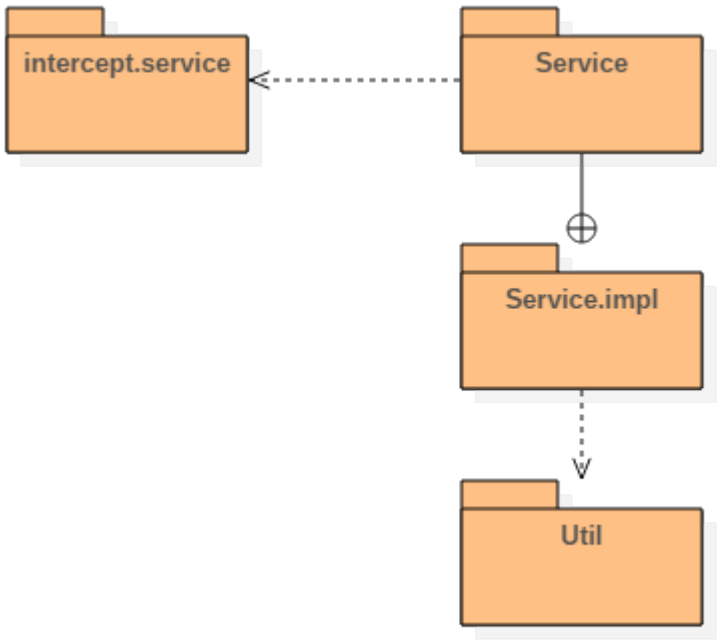
实际上可以理解为， ***消费者*** 调用 ***dubbo*** 提供的 ***webservice*** 服务，而 ***dubbo*** 则调用第三方的 ***webservice*** 服务。如下图：

Chapter 13. 代码说明

本工程依赖于 *dmga-dubbo-service* 工程，部分使用的 *interface* 及 *domain entity* 均在该工程下。

13.1. 包图及描述





包说明

service

抽象的服务接口

service.impl
: 服务实现类

util
: 工具包

intercept.service
: 拦截器接口

Chapter 14. 各功能模块说明

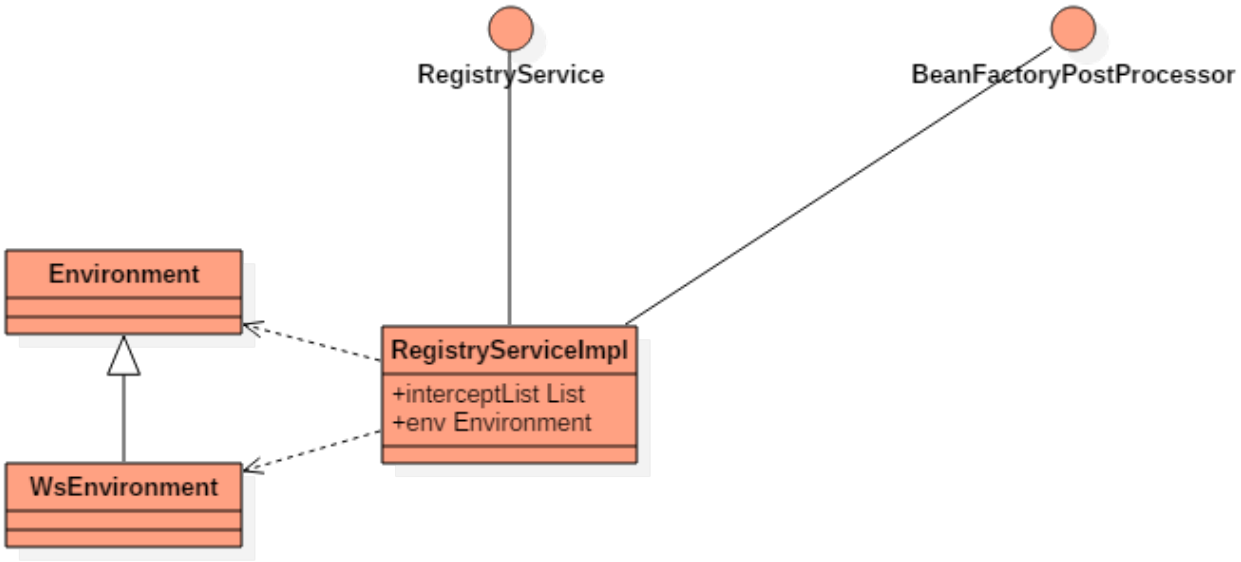
整个工程主要分为三个模块。

1. *ws* 服务的注册
2. *ws* 服务的调用
3. *wsdl* 文档解析

***ws* 服务的注册与调用** 一起完成了 *ws* 服务的代理工作。而 ***wsdl* 文档解析** 则是提供给调用者的一个工具类接口。

14.1. *ws* 服务的注册

14.1.1. 类图



14.1.2. 加载ws配置信息

默认的一些ws配置信息是由 *spring* 配置文件中注入进来的。

```
public class RegistryServiceImpl implements RegistryService, BeanFactoryPostProcessor
{
    /**
     * 服务提供方环境
     */
    private Environment env;
}
```

```
<bean id="defaultEnv" class="com.dm.dubbo.wsdl.registry.domain.Environment">
  <property name="registryAddress" value="zookeeper://localhost:1234"></property> ①
  <property name="protocolName" value="webservice"></property>
  <property name="protocolPort" value="8081"></property> ②
  <property name="protocolServer" value="jetty"></property> ③
  <property name="appName" value="WsGenericService"></property>
</bean>

<!-- 定义注册wsdl服务实现 -->
<bean id="registryService"
  class="com.dm.dubbo.wsdl.registry.service.impl.RegistryServiceImpl">
  <property name="env" ref="defaultEnv" /> ④
</bean>
```

① 定义注册中心的连接地址，新注册的 *ws* 服务将会连接该注册中心

② 定义发布的 *ws* 服务发布到哪个端口。 **不同的 *ws* 服务应该指定不同的端口**

③ 定义发布 *ws* 服务采用的容器，可选 *jetty* 和 *servlet* ，此处 *servlet* 指的是tomcat容器。

④ 此处将上面配置的环境信息注入到 **注册service** 中

14.1.3. 动态生成接口

当发布不同的 *wsdl* 的 *webservice* 服务时，为了在 *dubbo* 及 *cxf* 中显示不同的接口，此时需要动态生成接口。

注意

在 *dubbo* 中，如果 *interface name* 相同，那么需要设置不同的 *group name* 来保证是不同的服务。

而在 *cxf* 中，如果 *interface name* 相同，那么是不能重复发布的，此时就必须要动态生成接口。

动态生成接口思路

使用 *asm* 框架生成 *interface* 的 *bytecode* 写成 *class* 文件，而此 *interface* 只需要是继承现有的 *GenericService* 即可。

GenericService

```
/**
 * dubbo注册wsdl服务代理
 * @author zxb
 * 2016年3月3日 上午11:15:34
 */
@WebService
public interface GenericService {

    /**
     * 通过此接口请求代理方，然后调用提供方的ws服务
     * @param methodName
     * @param condition
     * @return
     * @throws Exception
     */
    public Object query(String methodName, String condition) throws Exception;
}
```

generate interface code

```
/**
 * 为该wsdl生成一个对应的接口
 *
 * @param wsEnv ws环境信息
 * @return 接口类
 * @throws ClassNotFoundException 加载的接口类未找到时抛出
 */
private Class generateInterface(WsEnvironment wsEnv) throws ClassNotFoundException {
    // 获取上下文的类加载器
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    String classpath = classLoader.getResource("").getPath(); // 获取classpath绝对uri
    logger.debug("current classpath:" + classpath);

    // 生成接口文件
    File interfaceClassFile = new File(classpath +
"/com/dm/dubbo/wsdl/registry/service/" + wsEnv.getInterfaceName() + ".class");
    logger.debug("interfaceClassFile:" + interfaceClassFile);

    if (!interfaceClassFile.exists()) {
        logger.info("interfaceClassFile:" + interfaceClassFile.getName() + "does't
exists! begin generate class file!");
        FileOutputStream fos = null;
        try {
            // 生成一个继承于GenericService的接口
            ClassWriter cw = new ClassWriter(0);
            cw.visit(V1_5, ACC_PUBLIC + ACC_ABSTRACT + ACC_INTERFACE,
                    "com/dm/dubbo/wsdl/registry/service/" + wsEnv.getInterfaceName(),
null, "java/lang/Object",
```

```
                    new String[]{"com/dm/dubbo/wsdl/registry/service/GenericService"}
);
            cw.visitEnd();

            // 写成class文件
            interfaceClassFile.getParentFile().mkdirs();
            fos = new FileOutputStream(interfaceClassFile);
            fos.write(cw.toByteArray());
        } catch (IOException e) {
            logger.error("generate class file:" + interfaceClassFile.getName() + "
error!", e);
            return null;
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    logger.error("close file output stream error! class file:" +
interfaceClassFile.getName(), e);
                }
            }
        }
        logger.info("interfaceClassFile:" + interfaceClassFile.getName() + " generate
success! write into " + interfaceClassFile);
    }

    // 加载class入classLoader
    logger.info("begin load interface class " + interfaceClassFile.getName());
    Class dynamicInterfaceClass = classLoader.loadClass(
"com.dm.dubbo.wsdl.registry.service." + wsEnv.getInterfaceName());
    logger.info("end load interface class " + interfaceClassFile.getName());
    return dynamicInterfaceClass;
}
```

从上面代码可以看出，生成接口主要分为以下几步

- 根据 *classloader* 获取 *classpath* 绝对 ***URI***
- 生成一个继承自 *GenericService* 的接口
- 将生成的接口写入 *classpath* 下对应的 *class* 文件
- 使用 *classloader* 将刚才生成的接口对应的 *class* 文件加载到jvm

先前尝试过使用 ***gclib*** 的 *interface maker* 来完成 *interface* 的动态生成。后来在 *dubbo* 框架中生成该 *interface* 的 *wrapper* 时一直报错： javassist : [compare source error] no such class

14.1.4. 动态生成实现类

虽然动态生成了接口，但是已经的 *GenericService* 接口的实现类 *GenericServiceAxis1Impl* 与 *GenericServiceCxfImpl* 是没有实现动态接口的。所以此时还需要这两个实现类也能实现动态接口。

实现思路

此时动态实现类有两种解决思路。

代理实现类

通过 *gclib* 生成现有的实现类的子类，同时实现 *dynamic interface*

持有实现类引用

直接通过 *gclib* 生成 *dynamic interface* 的实现类（ **实际代理类** ），在该 **代理类**中维持一个具体实现类的引用。当调用具体的接口方法时，**代理类**则调用具体实现类的该方法。

code

```
/**
    * 生成动态接口的实现类
    *
    * @param wsEnv            ws环境上下文
    * @param dynamicInterface 动态接口
    * @return 接口实现类
    * @throws ClassNotFoundException
    * @throws InstantiationException
    * @throws IllegalAccessException
    * @throws java.lang.reflect.InvocationTargetException
    */
   private GenericService getGenericService(WsEnvironment wsEnv, final Class
dynamicInterface) throws ClassNotFoundException, InstantiationException,
IllegalAccessException, java.lang.reflect.InvocationTargetException {
       // 获取clientClass实例，即具体的ws调用实例
       logger.info("begin get " + wsEnv.getClientClassName() + "'s instance!");
       Constructor<GenericService> cons = (Constructor<GenericService>) ClassUtils
.getConstructorIfAvailable(
               ClassUtils.forName(wsEnv.getClientClassName(), Thread.currentThread()
.getContextClassLoader()),
               WsEnvironment.class, List.class);
       final GenericService genericService = cons.newInstance(wsEnv, this
.interceptList);
       logger.info("get " + wsEnv.getClientClassName() + "'s instance success!");

       // 生成动态接口的实现类
       logger.info("begin generate " + dynamicInterface.getName() + "'s implementation
class");
       Enhancer enhancer = new Enhancer();
       enhancer.setInterfaces(new Class[]{dynamicInterface});
       enhancer.setCallback(new MethodInterceptor() {
           public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
               if (method.getName().equals("query")) { ①
                   return genericService.query(args[0].toString(), args[1].toString()
); // 调用具体的clientClass实例的方法。
               }
               return method.invoke(obj, args);
           }
       });
       GenericService proxyService = (GenericService) enhancer.create();
       logger.info("generate " + dynamicInterface.getName() + "'s implementation class
success! proxy class name " + proxyService.getClass().getName());
       return proxyService;
   }
```

① 拦截 **代理** 类的 ***query*** 方法,调用 **具体实现类** 的 *query* 方法。

14.1.5. 服务缓存

由于每次发布的服务对象实例非常重，所以需要将该服务缓存起来，以备后用。这里直接通过写了一个本地 *cache* 来实现。

服务对象缓存

code

```
/**
 * 服务缓存
 */
private Map<Environment, ServiceConfig> serviceCache = new ConcurrentHashMap
<Environment, ServiceConfig>();

// 暴露及注册服务
service.setInterface(dynamicInterface);
service.setRef(proxyService); // 真正的服务实现
service.setVersion(wsEnv.getVersion());
service.export();

// 缓存服务
serviceCache.put(wsEnv, service);
```

处理重复发布的服务

通过现有的服务 *cache* ，完成阻止重复服务的发布。

```
ServiceConfig serviceConfig = serviceCache.get(context);
if (serviceConfig != null) {
    logger.info("deploy a duplicate service, contextAppName:" + context.getAppName() +
", contextRegistryAddress:" + context.getRegistryAddress());
    return new RegistryMessage(ReturnStatus.OTHER, "服务已存在，请勿重复发布！");
}
```

同时，在具体的服务发布时，使用了同步代码块防止多线程并发时重复发布服务。

```
synchronized (RegistryServiceImpl.class) {
            serviceConfig = serviceCache.get(context);
            if (serviceConfig != null) {
                logger.info("deploy a duplicate service, contextAppName:" + context
.getAppName() + ", wsdl:" + wsEnv.getWsdlAddress());
                return new RegistryMessage(ReturnStatus.OTHER,
"服务已存在，请勿重复发布！");
            }

            // 发布服务
            serviceConfig.export();
}
```

由代码可以看出，服务是否重复取决于 *cache* 中的 *key* 是否相等。 因此在 *Environment* 与 *WsEnvironment* 中重写了 *hashcode* 与 *equals* 方法。

14.1.6. 服务配置

在线上测试过程中，出现过一些如连接超时等问题。所以在代码中也添加了部分配置。

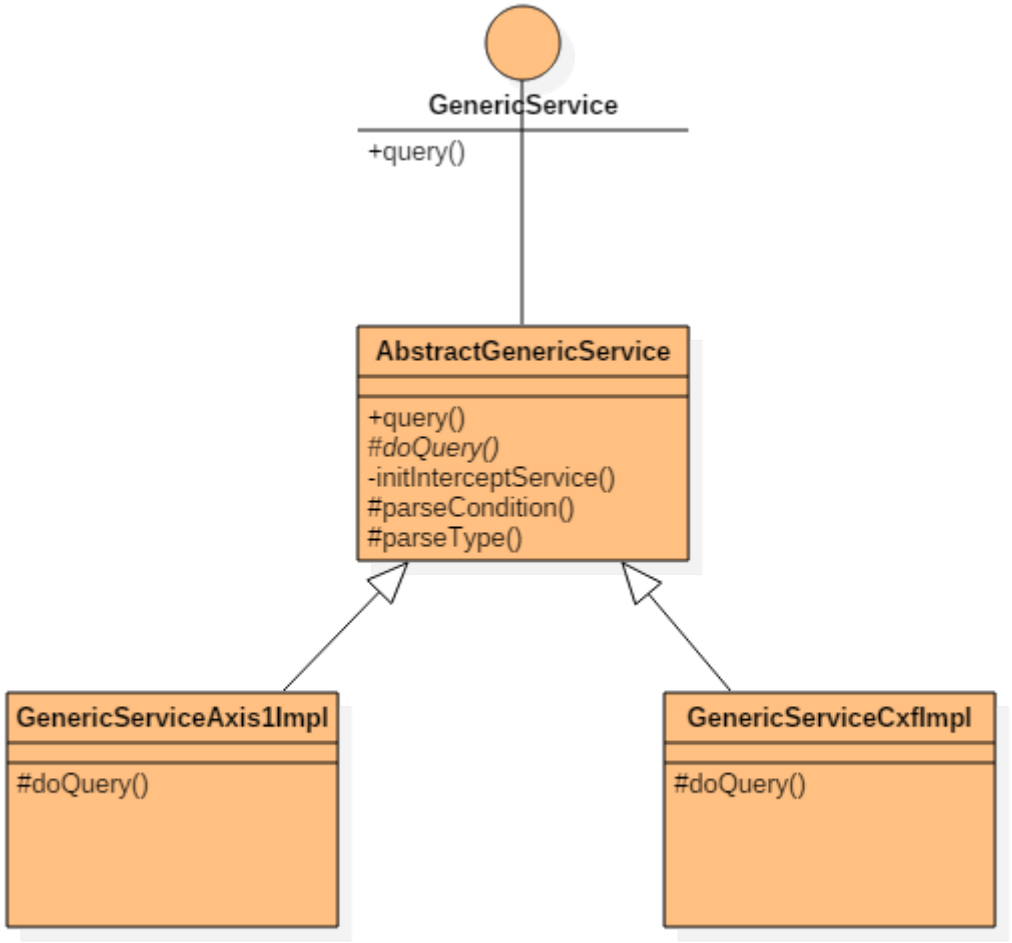
```
// 服务提供者协议配置
ProtocolConfig protocol = new ProtocolConfig();
protocol.setName(env2use.getProtocolName());
protocol.setPort(env2use.getProtocolPort());
protocol.setServer(env2use.getProtocolServer());
protocol.setAccepts(500); // 可接受的长连接为500个

// 服务提供者暴露服务配置
ServiceConfig<GenericService> service = new ServiceConfig<GenericService>(); //
此实例很重，封装了与注册中心的连接，请自行缓存，否则可能造成内存和连接泄漏
service.setApplication(application);
service.setRegistry(registry); // 多个注册中心可以用setRegistries()
service.setProtocol(protocol); // 多个协议可以用setProtocols()
service.setGroup(wsEnv.getGroup());
service.setTimeout(3000); //远程调用超时时间 ①
service.setRetries(0); //远程调用失败重试次数
service.setLoadbalance("roundrobin"); // 设置负载为轮循方式
```

① *timeout* 时间建议设大一些。否则调用方可能会收到一些 *TimeOut* 相关的异常。

14.2. *ws* 服务的调用

本工程中通过发布出去的 *webservice* 服务来反向调用第三方的 *webservice* 服务。

14.2.1. 类图



- ***GenericServiceAxis1Impl*** 完成对 *axis1.4* 提供的 *webservice* 服务的调用
- ***GenericServiceCxfImpl*** 完成对 *cxf* 、 *axis1.6* 提供的 *webservice* 服务的调用

14.2.2. *CXF* 调用

cxf 的调用是直接用 *cxf* 的客户端来完成 *webservcie* 的调用 的。 *cxf* 客户端又分成 静态调用（生成代码）与 动态调用 两种方式，此处采用的是 **动态生成客户端代码** 并调用的方式。

代码实现

```
// 获取Client
Client client = WsdlUtil.getCxfClient(super.wsContext.getWsdlAddress()); ①
Assert.notNull(client);

// 添加消息拦截器
client.getOutInterceptors().add(new LoggingOutInterceptor());
client.getInInterceptors().add(new LoggingInInterceptor());

Object[] result = client.invoke(methodName, params.values().toArray());
return result[0]; ②
```

① 获取client，此处是写地本地 *cache* 来获取客户端的，它实现是通过 *JaxWsDynamicClientFactory* 来创建客户端的。

② 此处目前没有考虑返回结果的类型，这里需要分析 *wsdl* 中对应方法的返回类型。

14.2.3. *axis1.4* 调用

axis1.4 的调用也是直接通过 *axis1.4* 的客户端来完成调用的。当时考虑采用 *soapui* [3: *soapui* 是一个开源的 *webservice* 调用工具] 的实现，出于时间问题，最后放弃了。

代码实现

```
@Override
public Object doQuery(String methodName, Map<String, Object> params) throws Exception
{
  return Axis1ClientUtil.invoke(super.wsContext.getWsdlAddress(), methodName,
      (LinkedHashMap<String, Object>) params);
} ①

public static Object invoke(String wsdl, String methodName, LinkedHashMap<String,
Object> params) throws Exception {
  Call call = Axis1ClientUtil.initCall(wsdl, null, null); ②
  call.setOperationName(methodName); ③

  // 设置参数
  Object[] objArr = new Object[0];
  if (params != null && params.size() > 0) {
    objArr = new Object[params.size()];

    int i = 0;
    for (Entry<String, Object> param : params.entrySet()) {
      call.addParameter(param.getKey(), Constants.XSD_ANY, ParameterMode.IN);
      objArr[i] = param.getValue();
      i++;
    }
  } ④
  return call.invoke(objArr);  ⑤
}
```

① 调用工具类完成 *ws* 调用

② 根据 *wsdl* 地址初始化 *Call*

③ 设置要调用的 *ws* 方法

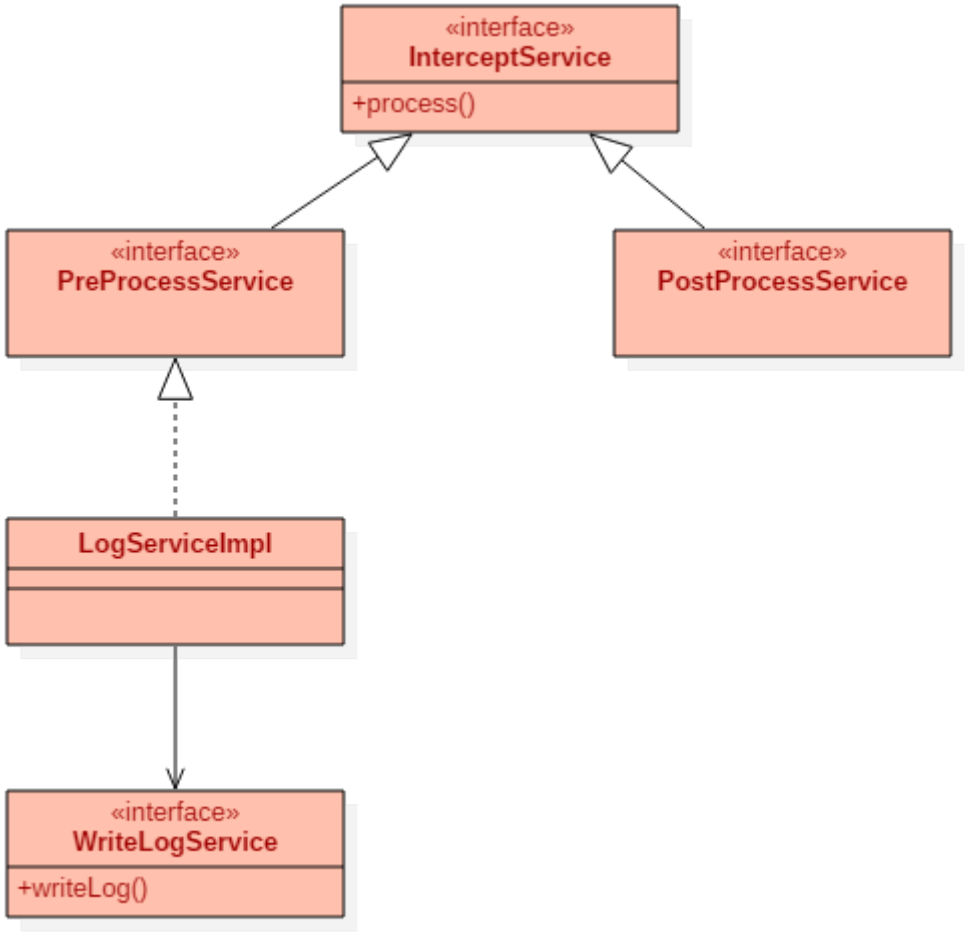
④ 设置 *arguments*

⑤ 完成调用并返回结果

14.3. *ws* 服务拦截

在调用第三方的 *ws* 服务时，通常需要记录请求方的 *ip* 等作为日志信息。此处为了方便后期的扩展，对 *ws* 调用请求的前后添加了额外处理，可以实现请求调用拦截，处理调用结果等。

14.3.1. 类图



14.3.2. 实现思路

在注册 *ws* 服务到 *dubbo* 时，将实现了指定接口的实现类注入到调用 *client* 实现中。在 *client* 调用第三方 *ws* 服务请求的前后添加拦截逻辑。

14.3.3. 实现细节

所有需要定义为拦截服务的类，只需要实现 *InterceptService* 或 *PostProcessService* 或 *PreProcessService* 中的任意一个即可。

手动配置拦截服务

在 *RegistryServiceImpl* 中添加了拦截服务集合。

```
/**
 * 拦截集合
 */
private List<InterceptService> interceptList;
```

所以，用户在 *spring* 配置文件中定义 *RegistryService* 时，可以直接在注入对应的 *InterceptService*

```
<!-- 定义注册wsdl服务实现 -->
<bean id="registryService"
  class="com.dm.dubbo.wsdl.registry.service.impl.RegistryServiceImpl">
  <property name="env" ref="defaultEnv" />
  <property name="interceptList" ref="" /> ①
</bean>
```

① 此处注入拦截服务的集合

自动加载配置为**springBean**的拦截服务

在 *RegistryServiceImpl* 中实现了 *BeanFactoryPostProcessor* 接口， *spring* 会将 *beanFactory* 注入进来。

```
public class RegistryServiceImpl implements RegistryService, BeanFactoryPostProcessor{
①

  public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
throws BeansException {
    if (this.interceptList == null) {
        this.interceptList = new ArrayList<InterceptService>();
    } ②
    String[] beanNames = beanFactory.getBeanNamesForType(InterceptService.class,
false, false); ③
    if (beanNames != null) {
        InterceptService interceptService = null;
        for (String beanName : beanNames) {
            interceptService = (InterceptService) beanFactory.getBean(beanName);
            if (!this.interceptList.contains(interceptService)) {
                this.interceptList.add(interceptService);
            }
        }
    } ④
}
}
```

① 实现 *BeanFactoryPostProcessor* 以获取 *BeanFactory*

② 获取 *interceptList*

③ 获得所有实现了 _InterceptService_ 接口的 _spring bean_

④ 将实现了 *InterceptService* 接口的 *bean* 装载入集合

注入拦截服务集合至 *client* 实现中

```
final GenericService genericService = cons.newInstance(wsEnv, this.interceptList); ①
```

① 通过构造注入 *interceptList*

处理拦截服务集合逻辑

AbstractGenericService 作为 *client* 实现的父类，完成了拦截服务的逻辑，同时也将相关方法声明为 *protected* 方便子类重写。

1. 将注入的 *InterceptService* 分类为前置拦截与后置拦截服务

```
/**
 * 初始化InterceptService
 */
private void initInterceptService() {
  if (this.interceptList != null && this.interceptList.size() > 0) {
    this.preList = new ArrayList<PreProcessService>();
    this.postList = new ArrayList<PostProcessService>();

    for (InterceptService service : this.interceptList) {
      if (service instanceof PreProcessService) {
        this.preList.add((PreProcessService) service);
      }
      if (service instanceof PostProcessService) {
        this.postList.add((PostProcessService) service);
      }
    }
  }
}
```

2. 处理前置逻辑

拦截逻辑都是在请求调用的 *query* 方法中完成的。

```
Object preProcessObj = doPreProcess(methodName, params);
if (preProcessObj != null)
          return preProcessObj;
```

由上可以看出，前置处理时可以直接返回，不进行请求处理。所以此处可以方便地添加一些 **请求参数校验，用户信息校验** 等。

3. 处理后置逻辑

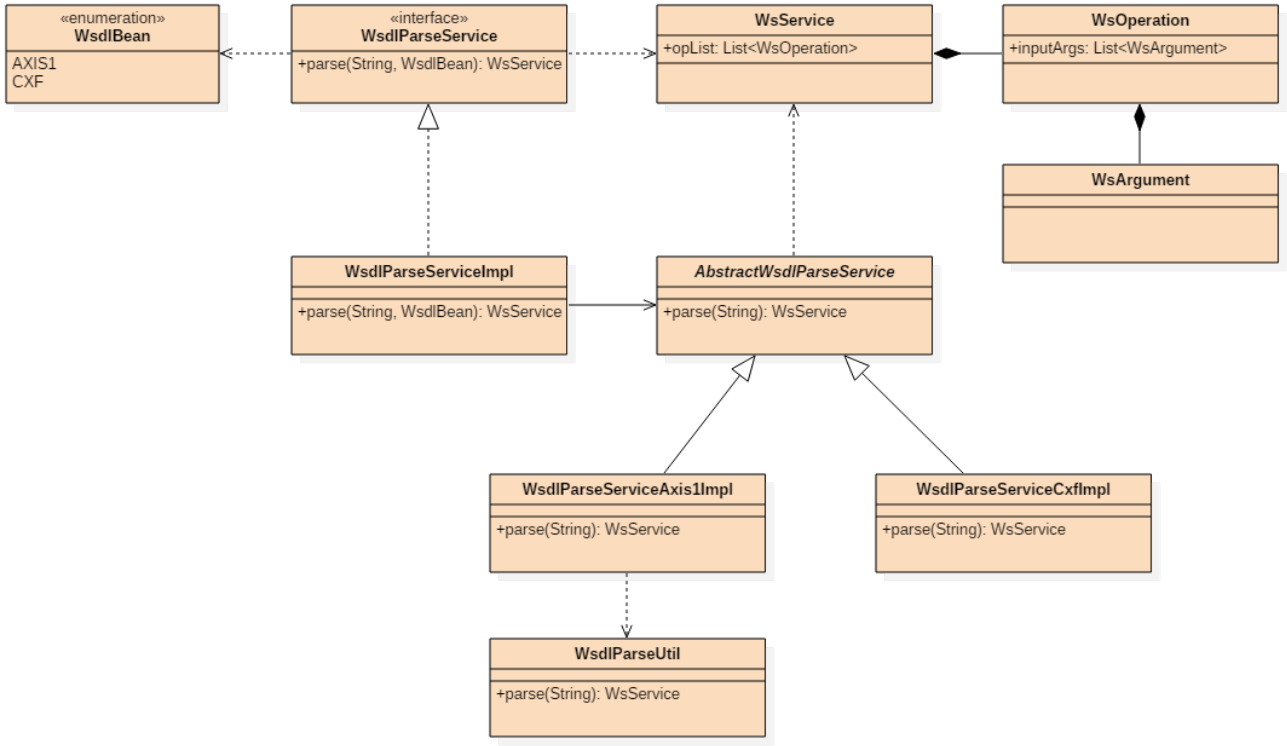
```
// 调用子类具体查询方法
returnObj = doQuery(methodName, realParams);

// 处理postProcess
params.put("returnObj", returnObj);
Object postProcessObj = doPostProcess(params);
if (postProcessObj != null)
          return postProcessObj; // 修改返回结果
```

同上，后置处理，可以添加一些日志记录功能。也同样可以完成一些返回信息过滤功能，此处可以更改返回结果。

14.4. *wsdl* 文档解析

对 *wsdl* 文档的解析，主要是方便对 *webservice* 服务的调用，在调用过程中方便指定请求的 *arguments* 及返回的 *returnType* 等。

14.4.1. 类图



14.4.2. 实现框架

同样，与 *webservice* 调用类似，对 *wsdl* 的解析也分为了 *CXF* 与 *axis1.4* 两种实现。

对 *CXF* 生成的 *wsdl* 解析，则直接使用的 *CXF* 客户端来完成的。

对 *axis1.4* 生成的 *wsdl* 解析，则是使用 *wsdl4j* [4: *wsdl4j* 是一个开源的 *wsdl* 文档解析组件，不过现在快被 *out* 了。] 开源组件完成的。

*cxf*解析与*axis1.4*的不同之处

axis1.4 的解析相对容易，直接使用 *wsdl4j* 即可完成。

而 *cxf* 则对方法的参数使用了嵌入的 *schema* 文档来定义的。而 *wsdl4j* 不支持这种 *schema* 文档的解析 [6: 其官网资料上也明确指出，*wsdl4j* 不支持这种 *schema* 的解析，只支持将其简单地转换为 *w3c dom*]，因为对 *schema* 的解析需要自己手工完成。

soapui 中对这两种 *wsdl* 的解析均有提供，迫于时间关系，未细研究。

14.4.3. *CXF* 的 *wsdl* 解析

对 *CXF* 生成的 *wsdl* 的解析，则由实现类 *WsdlParseServiceCxfImpl* 完成。基本思路都是逐步解析 *wsdl* 各元素来完成的。

code

```
public class WsdlParseServiceCxfImpl extends AbstractWsdlParseService {

    public WsService parse(String url) throws Exception {
        WsService wsService = new WsService();

        // 创建动态客户端
        JaxWsDynamicClientFactory factory = JaxWsDynamicClientFactory.newInstance();
        // 创建客户端连接
        Client client = factory.createClient(url);
        ClientImpl clientImpl = (ClientImpl) client;
        Endpoint endpoint = clientImpl.getEndpoint();

        wsService.setWsdlUrl(url);
        wsService.setServiceName(endpoint.getService().getName().getLocalPart());
        wsService.setTargetNameSpace(endpoint.getService().getName().getNamespaceURI(
));

        BindingInfo bindingInfo = endpoint.getBinding().getBindingInfo();
        Collection<BindingOperationInfo> opInfoCol = bindingInfo.getOperations();
        Iterator<BindingOperationInfo> it = opInfoCol.iterator();

        // 遍历方法
        List<WsOperation> opList = new ArrayList<WsOperation>();
        WsOperation wsOperation = null;
        while (it.hasNext()) {
            wsOperation = new WsOperation();

            BindingOperationInfo boi = it.next();
            String opName = boi.getName().getLocalPart();
            wsOperation.setName(opName); // 设置方法名

            // 通过输入消息找到对应的请求参数列表
            BindingMessageInfo inputMsgInfo = boi.getInput();
            List<MessagePartInfo> parts = inputMsgInfo.getMessageParts();

            List<WsArgument> args = new LinkedList<WsArgument>();
            WsArgument wsArg = null;
            for (MessagePartInfo part : parts) {
                Field[] fields = part.getTypeClass().getDeclaredFields();
                for (Field field : fields) {
                    Type type = field.getGenericType();
                    String typeStr = type.toString();
                    if (typeStr.indexOf("<") != -1 && typeStr.indexOf(">") != -1) {
                        typeStr = typeStr.substring(typeStr.indexOf("<") + 1, typeStr
```

```
.indexOf(">"));
                    }

                    // 设置入参
                    // TODO 目前只支持简单类型的入参，复杂类型需要递归处理
                    wsArg = new WsArgument();
                    wsArg.setName(field.getName());
                    wsArg.setType(typeStr);
                    args.add(wsArg);
                }
            }

            // 通过输出消息拿到返回类型
            BindingMessageInfo outputMsgInfo = boi.getInput();
            parts = outputMsgInfo.getMessageParts();

            for (MessagePartInfo part : parts) {
                Field[] fields = part.getTypeClass().getDeclaredFields();
                for (Field field : fields) {
                    Type type = field.getGenericType();
                    String typeStr = type.toString();
                    if (typeStr.indexOf("<") != -1 && typeStr.indexOf(">") != -1) {
                        typeStr = typeStr.substring(typeStr.indexOf("<") + 1, typeStr
.indexOf(">"));
                    }
                    wsOperation.setReturnType(typeStr);
                    break;
                }
            }

            wsOperation.setInputArgs(args);
            opList.add(wsOperation);
        }
        wsService.setOpList(opList);
        return wsService;
    }
}
```

14.4.4. *axis1.4* 的 *wsdl* 解析

axis1.4 的 *wsdl* 解析，则是由 *wsdl4j* 完成。思路同上，直接读取 *wsdl* 文件，逐一解析各元素即可。

code

```
package com.dm.dubbo.wsdl.registry.util;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
```

```
import javax.wsdl.Binding;
import javax.wsdl.Definition;
import javax.wsdl.Input;
import javax.wsdl.Message;
import javax.wsdl.Operation;
import javax.wsdl.Output;
import javax.wsdl.Part;
import javax.wsdl.Port;
import javax.wsdl.PortType;
import javax.wsdl.Service;
import javax.wsdl.WSDLException;
import javax.wsdl.factory.WSDLFactory;
import javax.wsdl.xml.WSDLReader;
import javax.xml.namespace.QName;

import org.apache.log4j.Logger;
import org.springframework.util.Assert;

import com.dm.dubbo.wsdl.registry.domain.WsArgument;
import com.dm.dubbo.wsdl.registry.domain.WsOperation;
import com.dm.dubbo.wsdl.registry.domain.WsService;

public class WsdlParseUtil {

    private static Logger logger = Logger.getLogger(WsdlParseUtil.class);
    private static WSDLFactory factory;
    private static WSDLReader reader;

    static {
        try {
            factory = WSDLFactory.newInstance();
            reader = factory.newWSDLReader();

            reader.setFeature("javax.wsdl.verbose", true);
            reader.setFeature("javax.wsdl.importDocuments", true);
        } catch (WSDLException e) {
            logger.error("init wsdl factory error!", e);
        }
    }

    @SuppressWarnings("unchecked")
    public static WsService parse(String url) {
        Assert.notNull(url, "wsdl url can't be null or empty!");

        try {
            // 解析wsdl
            Definition def = reader.readWSDL(url);

            // 设置命名空间
            WsService wsService = new WsService();
```

```
            wsService.setWsdlUrl(url);
            wsService.setTargetNameSpace(def.getTargetNamespace());

            // 获取服务
            Map<QName, Service> map = def.getServices();
            if (map == null || map.size() <= 0) {
                logger.error("no service found on wsdl:" + url);
                return null;
            }

            Service service = null;
            Iterator<Service> it = map.values().iterator();
            while (it.hasNext()) {
                service = it.next();
                break; // 只取第一个服务
            }

            if (service != null) {
                wsService.setServiceName(service.getQName().getLocalPart());

                // 获取Ports
                Map<String, Port> ports = service.getPorts();
                if (ports != null && ports.size() > 0) {

                    Port port = null;
                    Iterator<Port> portIt = ports.values().iterator();
                    while (portIt.hasNext()) {
                        port = portIt.next();
                        break;
                    }

                    // 获取所有方法
                    if (port != null) {
                        Binding binding = port.getBinding();
                        PortType portType = binding.getPortType();
                        List<Operation> operations = portType.getOperations();

                        if (operations == null || operations.size() <= 0) {
                            logger.info("no operation found on this wsdl :" + url);
                        } else {
                            WsOperation wsOp = null;
                            List<WsOperation> opList = new ArrayList<WsOperation>();
                            for (Operation op : operations) {
                                if (!op.isUndefined()) {
                                    // 获取输入参数与返回类型
                                    List<WsArgument> args = WsdlParseUtil
.getInputParameters(op);
                                    String returnType = WsdlParseUtil.getReturnType(
op);

                                    // 组装为WsOperation
```

```
                                    wsOp = new WsOperation(op.getName(), args,
returnType);
                                    opList.add(wsOp);
                                }
                            }
                            wsService.setOpList(opList);
                        }
                    }
                }

                return wsService;
            }
        } catch (WSDLException e) {
            logger.error("parse wsdl error! url:" + url, e);
        }
        return null;
    }

    @SuppressWarnings("unchecked")
    private static String getReturnType(Operation op) {
        Output output = op.getOutput();
        Message msgOut = output.getMessage();

        if (!msgOut.isUndefined()) {
            Iterator<Part> partIter = msgOut.getParts().values().iterator();

            while (partIter.hasNext()) {
                Part part = partIter.next();
                if (part.getTypeName() != null) {
                    return part.getTypeName().getLocalPart(); // 获取返回类型
                }
            }
        }
        return null;
    }

    @SuppressWarnings("unchecked")
    private static List<WsArgument> getInputParameters(Operation op) {
        List<WsArgument> list = new ArrayList<WsArgument>();

        WsArgument arg = null;
        Input input = op.getInput();
        Message msg = input.getMessage();

        // 构造请求参数
        if (!msg.isUndefined()) {
            Iterator<Part> partIter = msg.getParts().values().iterator();
            while (partIter.hasNext()) {
                Part part = partIter.next();
                arg = new WsArgument();
                arg.setName(part.getName());
```

```
                if (part.getTypeName() != null) {
                    arg.setType(part.getTypeName().getLocalPart());
                }
                list.add(arg);
            }
        }
        return list;
    }
}
```

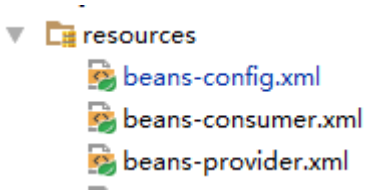
Chapter 15. 启动与打包

由于 *wsdl* 非 *web* 工程，所以只需要直接使用 *java* 命令启动包含 *main* 方法的类即可。

15.1. 启动方式

15.1.1. 编写main方法启动

整个 *wsdl* 工程中的服务的发布与注册均是通过配置文件完成的。



*spring*配置文件

所以从 *main* 方法启动则直接加载这些 *spring* 配置文件即可。

示例

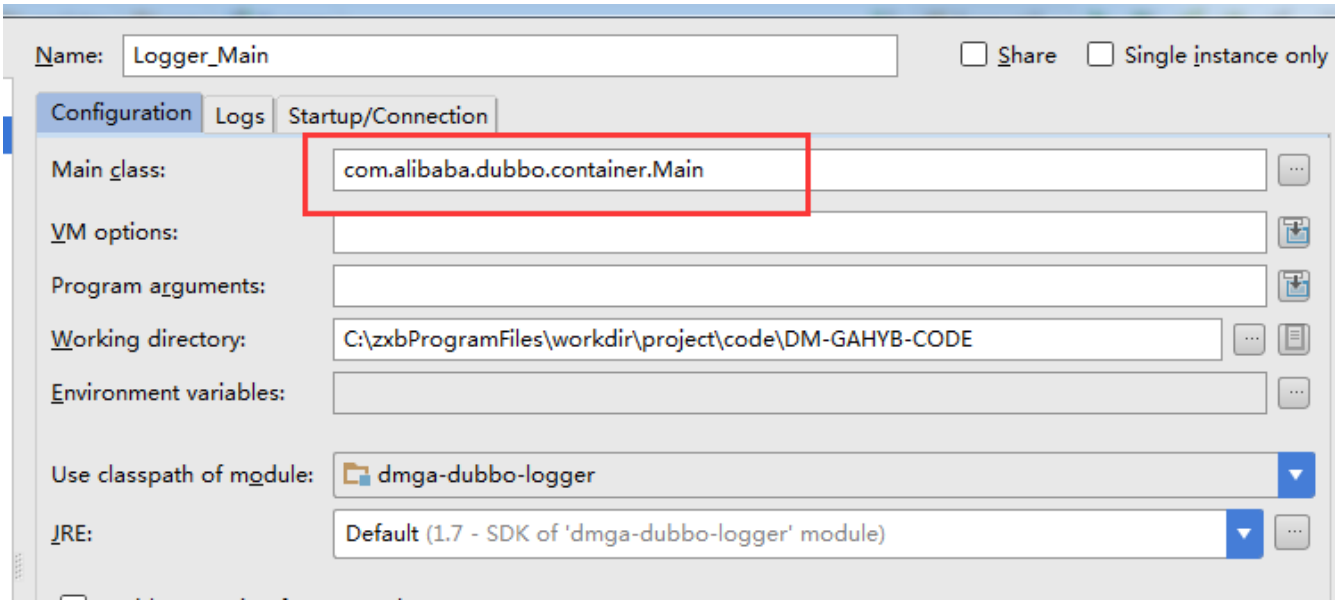
使用 *ClassPathXmlApplicationContext* 加载 *Spring* 配置文件。

```
ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:beans-*.xml");
applicationContext.start();

System.in.read(); ①
```

① 阻止主线程自动运行关闭

15.1.2. 以 *dubbo* 提供的 *Main* 类启动

其实在 *dubbo* 的官网上既不推荐以 *tomcat* 容器方式启动，也不推荐上面的 *main* 方法启动，而是推荐 *dubbo* 自己提供的 *Main* 类启动的，因为它支持 **优雅停机** 。



*idea*下配置示例

- 在 *ide* 下则配置以 *com.alibaba.dubbo.container.Main* 启动运行即可。
- 生产环境下，则应该以 `java com.alibaba.dubbo.container.Main` 方式启动。

15.2. 打包

如果直接将 *dmga-dubbo-wsdl* 工程打成一个 *jar* 包扔到生产环境下去跑的话，那么它的依赖 *jar* 呢？所以在打包的同时，也需要同时将它的依赖 *jar* 也一起打包。

此时需要配置 ***pom.xml*** 中的 *maven-jar-plugin* 以及 *maven-assembly-plugin* . 配置步骤如下。

maven-jar-plugin

在 ***pom.xml*** 中添加 *maven-jar-plugin* 的配置，注意放在 `<bulid><plugins></plugins></build>` 中。

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.4</version>
    <!-- The configuration of the plugin -->
    <configuration>
        <!-- 排队根目录下的配置文件，因为配置文件会使用assembly插件打包到conf目录
-->
        <excludes>
            <exclude>*.xml</exclude>
            <exclude>*.properties</exclude>
        </excludes>
        <!-- Configuration of the archiver -->
        <archive>

            <!--
                生成的jar中，不要包含pom.xml和pom.properties这两个文件
            -->
            <addMavenDescriptor>false</addMavenDescriptor>

            <!-- Manifest specific configuration -->
            <manifest>

                <!--
                    是否要把第三方jar放到manifest的classpath中
                -->
                <!--<addClasspath>true</addClasspath>-->
                <!--
                   生成的manifest中classpath的前缀，因为要把第三方jar放到
lib目录下，所以classpath的前缀是lib/
               -->
                <classpathPrefix>lib/</classpathPrefix>
                <!--<mainClass>com.alibaba.dubbo.container.Main</mainClass>-->
                <!--
                    应用的main class
                -->
            </manifest>
        </archive>
        <!--
            过滤掉不希望包含在jar中的文件
        -->
    </configuration>
</plugin>
```

maven-assembly-plugin

该插件加载具体的打包配置

```
<!-- The configuration of maven-assembly-plugin -->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.4</version>
    <!-- The configuration of the plugin -->
    <configuration>
        <!-- Specifies the configuration file of the assembly plugin -->
        <descriptors>
            <descriptor>
${project.basedir}/src/main/resources/package.xml</descriptor> ①
        </descriptors>
    </configuration>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

① 指定了package.xml的路径

package.xml

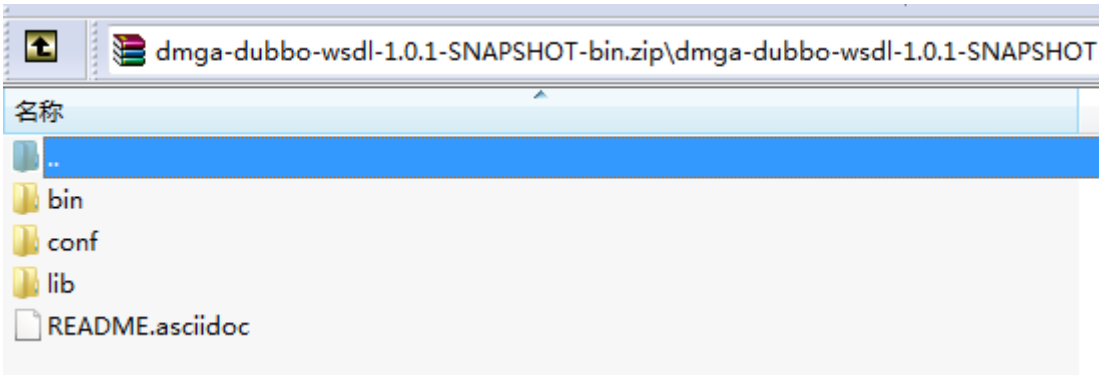
package.xml中配置了具体的打包细节。

```
<assembly>
    <id>bin</id>
    <!-- 最终打包成一个用于发布的zip文件 -->
    <formats>
        <format>zip</format>
    </formats>

    <!-- Adds dependencies to zip package under lib directory -->
    <dependencySets>
        <dependencySet>
            <!--
               不使用项目的artifact，第三方jar不要解压，打包进zip文件的lib目录
            -->
            <useProjectArtifact>false</useProjectArtifact>
            <outputDirectory>lib</outputDirectory>
            <unpack>false</unpack>
        </dependencySet>
    </dependencySets>

    <fileSets>
```

```
        <!-- 把项目相关的说明文件，打包进zip文件的根目录 -->
        <fileSet>
            <directory>${project.basedir}</directory>
            <outputDirectory>/</outputDirectory>
            <includes>
                <include>README*</include>
                <include>LICENSE*</include>
                <include>NOTICE*</include>
            </includes>
        </fileSet>

        <!-- 把项目的配置文件，打包进zip文件的config目录 -->
        <fileSet>
            <directory>${project.build.directory}/classes</directory>
            <outputDirectory>conf</outputDirectory>
            <includes>
                <include>*.xml</include>
                <include>*.properties</include>
            </includes>
        </fileSet>

        <!-- 把项目的脚本文件目录（ src/main/scripts ）中的启动脚本文件，打包进
zip文件的bin目录 -->
        <fileSet>
            <directory>${project.build.scriptSourceDirectory}</directory>
            <outputDirectory>bin</outputDirectory>
            <!--
            <includes>
                <include>startup.*</include>
            </includes>
            -->
        </fileSet>

        <!-- 把项目的脚本文件（除了启动脚本文件），打包进zip文件的script目录 -->
        <!--
        <fileSet>
            <directory>${project.build.scriptSourceDirectory}</directory>
            <outputDirectory></outputDirectory>
            <includes>
                <exclude>startup.*</exclude>
            </includes>
        </fileSet>
        -->

        <!-- 把项目自己编译出来的jar文件，打包进zip文件的lib目录 -->
        <fileSet>
            <directory>${project.build.directory}</directory>
            <outputDirectory>lib</outputDirectory>
            <includes>
                <include>*.jar</include>
            </includes>
```

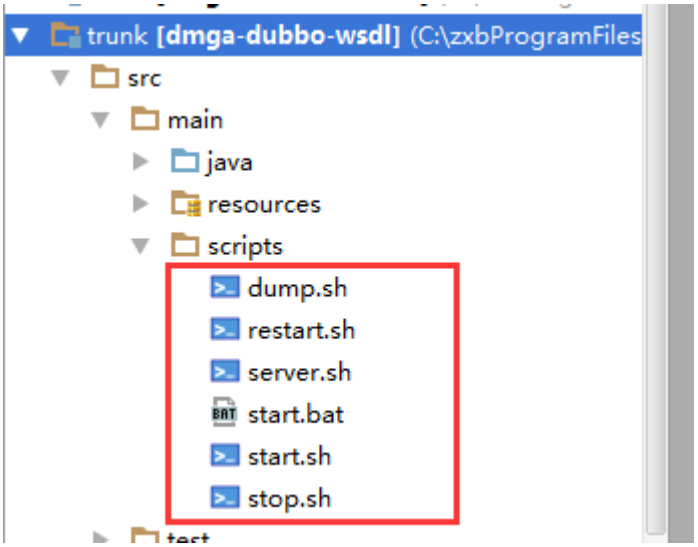
```
        </fileSet>
    </fileSets>
</assembly>
```

按上述配置文件配置完毕后，使用 *maven* 执行命令 `mvn clean package -Dmaven.test.skip=true` 将会在 *target* 目录下产生出 *-bin.zip 文件。该 *zip* 包解压后即是一个可运行包。



15.3. 启动脚本

为了简化生产环境下的一些启动与停止服务操作，则特地将一些命令整成启动脚本存放。



在上述的打包命令中，则是直接将这些启动 *scripts* 打包到了 *bin* 目录。

具体内容可直接查看脚本内容。