REST

【1】简介

REST（Representational State Transfer）表述性状态转移

首先从浏览器发送AJAX请求，然后服务端接受该请求并返回JSON数据返回给浏览器，最后在浏览器中进行界面渲染。

REST是一个无状态的架构模式，因为在任何时候都可以由客户端发出请求到服务端，最终返回自己想要的数据，当前请求不会受到上次请求的影响。也就是说，服务端将内部资源发布REST服务，客户端通过URL来定位这些资源并通过HTTP协议来访问它们。

一个具有REST风格项目的基本特征：

- 具有统一响应结构；
- 前后台数据流转机制(HTTP消息与Java对象的互相转化机制)；
- 统一的异常处理机制；
- 参数验证机制；
- Cors跨域请求机制;
- 安全(鉴权)机制。

【2】请求方式

GET（查）

POST（增）

PUT（改）

DELETE（删）

HEAD

OPTIONS

【3】实现REST框架

1、统一响应结构

每个REST请求将返回相同结构的JSON响应结构。

不妨定义一个相对通用的JSON响应结构，其中包含两部分：元数据与返回值，其中，元数据表示操作是否成功与返回值消息等，返回值对应服务端方法所返回的数据。例如：

```
{
 "meta": {
    "success": true,
    "message": "ok"
 },
 "data": ...
}
```

同时，编写一个Response类来与之对应：

```
public class Response{
    private static final String OK = "ok";
    private static final String ERROR = "error";
    private Meta meta;    // 元数据
    private Object data;  // 响应内容

    public Response success() {
      this.meta = new Meta(true, OK);
      return this;
    }

   public Response success(Object data) {
         this.meta = new Meta(true, OK);
         this.data = data;
         return this;
      }

   public Response failure() {
         this.meta = new Meta(false, ERROR);
         return this;
      }
```

```
    public Response failure(String message) {
            this.meta = new Meta(false, message);
            return this;
        }

    public Meta getMeta() {
            return meta;
        }

    public Object getData() {
            return data;
        }

     /**
     *请求元数据
     */
     public class Meta {

       private boolean success;
       private String message;

       public Meta(boolean success) {
          this.success = success;
       }

       public Meta(boolean success, String message) {
          this.success = success;
          this.message = message;
       }

       public boolean isSuccess() {
          return success;
       }

       public String getMessage() {
          return message;
       }
     }
}
```

以上Response类包括两类通用返回值消息：ok 与 error，还包括两个常用的操作方法：success( )与failure( )，通过一个内部类来展现元数据结构。

2、前后台数据流转

即HTTP消息（带有json格式的参数）与Java对象之间的转化问题。

（1）在SpringMVC中，在Controller中使用@RequestBody注解可以将接收到的HTTP消息转化为Java对象（例如：在参数列表中，@RequestBody User user），使用@ResponseBody可以将 Java 对象转化为特定的HTTP消息（在该方法上一行中添加注解）。

@ResponseBody注解也可以定义在类上，这样所有的方法都继承了该特性。由于经常会使用到@ResponseBody注解，所以Spring提供了一个名为@RestController的注解来取代以上的@Controller注解，这样我们就可以省略返回值前面的@ResponseBody注解了，但参数前面的@RequestBody注解是无法省略的。

（2） 除了使用注解来定义消息转化行为以外，还需要添加Jackson包进行支持，Maven依赖如下：

```
<!-- JSON: jackson -->
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-core-asl</artifactId>
  <version>1.9.12</version>
</dependency>
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-mapper-lgpl</artifactId>
```

```
    <version>1.9.12</version>
  </dependency>
```

在Spring配置文件中添加以下配置即可：

```
<!-- 该配置会自动注册RequestMappingHandlerMapping与RequestMappingHandlerAdapter两个Bean，这是SpringMVC为@Controllers分发请求所必需的，并提供了数据绑定支持、@NumberFormatannotation支持、@DateTimeFormat支持、@Valid支持、读写XML的支持和读写JSON的支持等功能。 -->
<mvc:annotation-driven />
```

3、处理异常行为

在Spring MVC中，可以使用AOP技术，编写一个全局的异常处理切面类，用它来统一处理所有的异常行为，在Spring 3.2中才开始提供。只需定义一个类，并通过@ControllerAdvice注解将其标注即可，同时需要使用@ResponseBody注解表示返回值可序列化为JSON字符串。代码如下：

```
@ControllerAdvice   // 控制器增强
@ResponseBody
public class ExceptionAspect {

  /** Log4j日志处理(@author: rico) */
  private static final Logger log = Logger.getLogger(ExceptionAspect.class);

  /**
   * 400 - Bad Request
   */
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  @ExceptionHandler(HttpMessageNotReadableException.class)
  public Response handleHttpMessageNotReadableException(
      HttpMessageNotReadableException e) {
    log.error("could_not_read_json...", e);
    return new Response().failure("could_not_read_json");
  }

  /**
   * 400 - Bad Request
   */
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  @ExceptionHandler({MethodArgumentNotValidException.class})
  public Response handleValidationException(MethodArgumentNotValidException e) {
    log.error("parameter_validation_exception...", e);
    return new Response().failure("parameter_validation_exception");
  }

  /**
   * 405 - Method Not Allowed。HttpRequestMethodNotSupportedException
   * 是ServletException的子类,需要Servlet API支持
   */
  @ResponseStatus(HttpStatus.METHOD_NOT_ALLOWED)
  @ExceptionHandler(HttpRequestMethodNotSupportedException.class)
  public Response handleHttpRequestMethodNotSupportedException(
      HttpRequestMethodNotSupportedException e) {
    log.error("request_method_not_supported...", e);
    return new Response().failure("request_method_not_supported");
  }

  /**
   * 415 - Unsupported Media Type。HttpMediaTypeNotSupportedException
   * 是ServletException的子类,需要Servlet API支持
   */
  @ResponseStatus(HttpStatus.UNSUPPORTED_MEDIA_TYPE)
  @ExceptionHandler({ HttpMediaTypeNotSupportedException.class })
  public Response handleHttpMediaTypeNotSupportedException(Exception e) {
```

```
        log.error("content_type_not_supported...", e);
        return new Response().failure("content_type_not_supported");
    }

    /**
     * 500 - Internal Server Error
     */
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    @ExceptionHandler(TokenException.class)
    public Response handleTokenException(Exception e) {
        log.error("Token is invaild...", e);
        return new Response().failure("Token is invaild");
    }

    /**
     * 500 - Internal Server Error
     */
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    @ExceptionHandler(Exception.class)
    public Response handleException(Exception e) {
        log.error("Internal Server Error...", e);
        return new Response().failure("Internal Server Error");
    }
}
```

在ExceptionAdvice类中包含一系列的异常处理方法，每个方法都通过@ResponseStatus注解定义了响应状态码，此外还通过@ExceptionHandler注解指定了具体需要拦截的异常类。以上过程只是包含了一部分的异常情况，若需处理其它异常，可添加方法具体的方法。需要注意的是，在运行时从上往下依次调用每个异常处理方法，匹配当前异常类型是否与@ExceptionHandler注解所定义的异常相匹配，若匹配，则执行该方法，同时忽略后续所有的异常处理方法，最终会返回经JSON序列化后的Response对象。

4、支持参数验证

```
@RestController
@RequestMapping("/users")
public class UserController {

    private UserService userService;

    /** Log4j日志处理(@author: rico) */
    private static final Logger log = Logger.getLogger(UserController.class);

    public UserService getUserService() {
        return userService;
    }

    @Resource(name = "userService")
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    @RequestMapping(value = "/user", method = RequestMethod.PUT, produces = "application/json", consumes = "application/json")
    public User addUser(@RequestBody @Valid User user) { // 将接收到的HTTP消息转化为Java对象
        userService.addUser(user);
        log.debug("添加用户 :" + user);
        return user;
    }
    ...
}


public class User implements Serializable{
```

```
    private static final long serialVersionUID = 1L;
    private int id;
    @NotEmpty
    private String uname;
    private String passwd;
    private String gentle;
    private String email;
    private String city;
    public User() {
        super();
    }
    // getter/setter
    // toString
}
```

以上代码将其参数验证行为从Controller中剥离出来，放到另外的类中，这里仅通过@Valid注解来定义uname参数，并通过Bean Validation的参考实现Hibernate Validator的@NotEmpty注解来定义User类中的uname属性。

这里的@Valid注解实际上是Validation Bean规范提供的注解，该规范已由Hibernate Validator框架实现，因此需要添加以下Maven依赖到pom.xml文件中：

```
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-validator</artifactId>
        <version>${hibernate-validator.version}</version>
    </dependency>
```

需要在Spring配置文件中开启该特性，需添加如下配置：

```
    <bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>
```

在全局异常处理类中添加对参数验证异常的处理方法，代码如下：

```
    @ControllerAdvice
    @ResponseBody
    public class ExceptionAdvice {

        /**
         * 400 - Bad Request
         */
        @ResponseStatus(HttpStatus.BAD_REQUEST)
        @ExceptionHandler(ValidationException.class)
        public Response handleValidationException(ValidationException e) {
            logger.error("参数验证失败", e);
            return new Response().failure("validation_exception");
        }
    }
```

至此，REST框架已集成了Bean Validation特性，我们可以使用各种注解来完成所需的参数验证行为了。 整个架构包含两个应用，前端应用提供纯静态的HTML页面，后端应用发布REST API，前端需要通过AJAX调用后端发布的REST API，然而AJAX是不支持跨域访问的，也就是说，前后端两个应用必须在同一个域名下才能访问。

5、解决跨域问题

使前端应用通过AJAX跨域访问后端应用，需要用到CORS（Cross Origin Resource Sharing跨域资源共享）技术来实现，服务端可通过任何编程语言来实现，只需将CORS响应头写入response对象中即可。

首先，需要编写一个Filter，用于过滤所有的HTTP请求，并将CORS响应头写入response对象中，代码如下：

```
    public class CorsFilter implements Filter {

  /** Log4j日志处理(@author: rico) */
  private static final Logger log = Logger.getLogger(UserController.class);

  private String allowOrigin;
  private String allowMethods;
  private String allowCredentials;
  private String allowHeaders;
  private String exposeHeaders;
```

```
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        allowOrigin = filterConfig.getInitParameter("allowOrigin");
        allowMethods = filterConfig.getInitParameter("allowMethods");
        allowCredentials = filterConfig.getInitParameter("allowCredentials");
        allowHeaders = filterConfig.getInitParameter("allowHeaders");
        exposeHeaders = filterConfig.getInitParameter("exposeHeaders");
    }

    @Override
    public void doFilter(ServletRequest req, ServletResponse res,
            FilterChain chain) throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;
        String currentOrigin = request.getHeader("Origin");
        log.debug("currentOrigin : " + currentOrigin);
        if (StringUtil.isNotEmpty(allowOrigin)) {
            List<String> allowOriginList = Arrays
                    .asList(allowOrigin.split(","));
            log.debug("allowOriginList : " + allowOrigin);
            if (CollectionUtil.isNotEmpty(allowOriginList)) {
                if (allowOriginList.contains(currentOrigin)) {
                    response.setHeader("Access-Control-Allow-Origin",
                            currentOrigin);
                }
            }
        }
        if (StringUtil.isNotEmpty(allowMethods)) {
            response.setHeader("Access-Control-Allow-Methods", allowMethods);
        }
        if (StringUtil.isNotEmpty(allowCredentials)) {
            response.setHeader("Access-Control-Allow-Credentials",
                    allowCredentials);
        }
        if (StringUtil.isNotEmpty(allowHeaders)) {
            response.setHeader("Access-Control-Allow-Headers", allowHeaders);
        }
        if (StringUtil.isNotEmpty(exposeHeaders)) {
            response.setHeader("Access-Control-Expose-Headers", exposeHeaders);
        }
        chain.doFilter(req, res);
    }

    @Override
    public void destroy() {
    }
}
```

以上CorsFilter将从web.xml中读取相关Filter初始化参数，并将在处理HTTP请求时将这些参数写入对应的CORS响应头中，下面大致描述一下这些CORS响应头的意义：

（1）Access-Control-Allow-Origin：允许访问的客户端域名，例如：http://web.xxx.com，若为*，则表示从任意域都能访问，即不做任何限制；

（2）Access-Control-Allow-Methods：允许访问的方法名，多个方法名用逗号分割，例如：GET,POST,PUT,DELETE,OPTIONS；

（3）Access-Control-Allow-Credentials：是否允许请求带有验证信息，若要获取客户端域下的cookie时，需要将其设置为true；

（4）Access-Control-Allow-Headers：允许服务端访问的客户端请求头，多个请求头用逗号分割，例如：Content-Type；

（5）Access-Control-Expose-Headers：允许客户端访问的服务端响应头，多个响应头用逗号分割。

需要注意的是，CORS规范中定义Access-Control-Allow-Origin只允许两种取值，要么为*，要么为具体的域名，也就是说，不支持同时配置多

个域名。为了解决跨多个域的问题，需要在代码中做一些处理，这里将Filter初始化参数作为一个域名的集合（用逗号分隔），只需从当前请求中获取Origin请求头，就知道是从哪个域中发出的请求，若该请求在以上允许的域名集合中，则将其放入Access-Control-Allow-Origin响应头，这样跨多个域的问题就轻松解决了。以下是web.xml中配置CorsFilter的方法：

```
    <!-- 通过CORS技术实现AJAX跨域访问 -->
  <filter>
    <filter-name>corsFilter</filter-name>
    <filter-class>cn.edu.tju.rico.filter.CorsFilter</filter-class>
    <init-param>
      <param-name>allowOrigin</param-name>
      <param-value>http://localhost:8020</param-value>
    </init-param>
    <init-param>
      <param-name>allowMethods</param-name>
      <param-value>GET,POST,PUT,DELETE,OPTIONS</param-value>
    </init-param>
    <init-param>
      <param-name>allowCredentials</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>allowHeaders</param-name>
      <param-value>Content-Type,X-Token</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>corsFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

由于REST是无状态的，后端应用发布的REST API可在用户未登录的情况下被任意调用，这显然是不安全的，为了解决这个问题需要为REST请求提供安全机制。

6、提供安全机制

解决REST安全调用问题，可以做得很复杂，也可以做得特简单，可按照以下过程提供REST安全机制：

(1). 当用户登录成功后，在服务端生成一个token，并将其放入内存中（可放入JVM或Redis中），同时将该token返回到客户端；

(2). 在客户端中将返回的token写入cookie中，并且每次请求时都将token随请求头一起发送到服务端；

(3). 提供一个AOP切面，用于拦截所有的Controller方法，在切面中判断token的有效性；

(4). 当登出时，只需清理掉cookie中的token即可，服务端token可设置过期时间，使其自行移除。

首先，我们需要定义一个用于管理token的接口，包括创建token与检查token有效性的功能。代码如下：

```
public interface TokenManager {
  String createToken(String username);
  boolean checkToken(String token);
  void deleteToken(String token);
}
```

然后，我们可提供一个简单的TokenManager实现类，将token存储到JVM内存中。代码如下：

```
public class DefaultTokenManager implements TokenManager {
  /** 将token存储到JVM内存(ConcurrentHashMap)中 */
  private static Map<String, String> tokenMap = new ConcurrentHashMap<String, String>();
  /**
   * @description 利用UUID创建Token(用户登录时，创建Token)
   * @param username
   * @return
   */
  public String createToken(String username) {
    String token = CodecUtil.createUUID();
    tokenMap.put(token, username);
    return token;
  }
```

```
    /**
     * @description Token验证(用户登录验证)
     * @param token
     * @return
     */
    public boolean checkToken(String token) {
        return !StringUtil.isEmpty(token) && tokenMap.containsKey(token);
    }

    /**
     * @description Token删除(用户登出时，删除Token)
     * @param token
     */
    @Override
    public void deleteToken(String token) {
        // TODO Auto-generated method stub
        tokenMap.remove(token);
    }
}
```

需要注意的是，如果需要做到分布式集群，建议基于Redis提供一个实现类，将token存储到Redis中，并利用Redis与生俱来的特性，做到token的分布式一致性。

然后，可以基于Spring AOP写一个切面类，用于拦截Controller类的方法，并从请求头中获取token，最后对token有效性进行判断。代码如下：

```
@Component
@Aspect
public class SecurityAspect {
    /** Log4j日志处理*/
    private static final Logger log = Logger.getLogger(SecurityAspect.class);
    private TokenManager tokenManager;

    @Resource(name = "tokenManager")
    public void setTokenManager(TokenManager tokenManager) {
        this.tokenManager = tokenManager;
    }

    @Around("@annotation(org.springframework.web.bind.annotation.RequestMapping)")
    public Object execute(ProceedingJoinPoint pjp) throws Throwable {
        // 从切点上获取目标方法
        MethodSignature methodSignature = (MethodSignature) pjp.getSignature();
        log.debug("methodSignature : " + methodSignature);
        Method method = methodSignature.getMethod();
        log.debug("Method : " + method.getName() + " : "
                + method.isAnnotationPresent(IgnoreSecurity.class));
        // 若目标方法忽略了安全性检查,则直接调用目标方法
        if (method.isAnnotationPresent(IgnoreSecurity.class)) {
            return pjp.proceed();
        }

        // 从 request header 中获取当前 token
        String token = WebContextUtil.getRequest().getHeader(
                Constants.DEFAULT_TOKEN_NAME);
        // 检查 token 有效性
        if (!tokenManager.checkToken(token)) {
            String message = String.format("token [%s] is invalid", token);
            log.debug("message : " + message);
            throw new TokenException(message);
        }
```

```
        // 调用目标方法
        return pjp.proceed();
    }
}
```

若要使SecurityAspect生效，则需要在SpringMVC配置文件中添加如下Spring 配置：

```
<!-- 启用注解扫描，并定义组件查找规则 ，mvc层只负责扫描@Controller、@ControllerAdvice -->
    <!-- base-package 如果多个，用“,”分隔 -->
    <context:component-scan base-package="cn.edu.tju.rico"
        use-default-filters="false">
        <!-- 扫描 @Controller -->
        <context:include-filter type="annotation"
            expression="org.springframework.stereotype.Controller" />
        <!-- 控制器增强，使一个Contoller成为全局的异常处理类，类中用@ExceptionHandler方法注解的方法可以处理所有Controller发生的异常 -->
        <context:include-filter type="annotation"
            expression="org.springframework.web.bind.annotation.ControllerAdvice" />
    </context:component-scan>

    <!-- 支持Controller的AOP代理 -->
    <aop:aspectj-autoproxy />
```

最后，在web.xml中添加允许的X-Token响应头，配置如下：

```
    <init-param>
        <param-name>allowHeaders</param-name>
        <param-value>Content-Type,X-Token</param-value>
    </init-param>
```

7、项目部署

关于REST服务的调试推荐使用Postman这款工具