后端统一开发框架

黄新房

2017年11月15日

- 使用技术
- 开发工具及环境
- 开发平台及相关约定
- 开发框架及使用技术介绍

使用技术

- 后端采用Java进行开发，JDK版本1.8+
- 使用框架SpringMVC+Hibernate+Spring Data JPA+Hibernate Validator
- 日志采用log4j2

开发工具及环境

- 开发工具使用：Eclipse
- 运行环境：Tomcat 8.5+
- 构建工具：Maven
- 消息总线采用：RabbitMQ ？ ActiveMQ

开发平台及相关约定

- 包名约定：
- Controller约定
- Service约定
- Dao约定
- 其他约定

包名约定

为避免一个相同名字的类出现在不同模块的相同包下，而通常每个产品或项目中每个模块都会配置一个parent

约定包名为${groupId}.${artifactId} 如

现假定parent的

${groupId}为com.nuctech.tcs

${artifactId}为parent

当前模块：

${groupId}为com.nuctech.tcs

${artifactId}为common

则当前模块的包名为**com.nuctech.tcs.common**

若Maven Project出现不同的模块继承不同的parent，请使用如下约定：

现假定parent的

${groupId}为com.nuctech.tcs

${artifactId}为parent(或tcs)

当前模块：

${groupId}为com.nuctech.tcs

${artifactId}包含parent的${artifactId}为parent.common(或tcs.common)，发布的模块不需要包含包含parent的${artifactId}

则当前模块的包名为**com.nuctech.tcs.common**

Controller约定

- controller统一存放于com.nuctech..controller..
- controler统一采用非Collection对象接收前端传递的参数。并且该对象必须加上@Valid注解，并且需要传入BindingResult对象
- controller分页及排序参数统一采用Pageable对象进行接收

Service约定(强制要求)

- 单表对应Service统一继承CommonService
- 单表对应Service统一注入Dao Interface
- Service可以注入多个Dao Interface

  注入的Dao Interface命名规则

  实体类名（首字母小写）+Dao

Dao约定(强制要求)

- Dao Interface 统一继承CommonDao自定义实现类统一继承CommonDaoImpl
- Dao Interface 名称：

  统一采用实体类（首字母小写）+Dao
- Dao自定义实现类命名：

  统一采用Dao Interface名字后加Impl
- Dao自定义实现类不加@Repository 注解

其他约定

- 工具类，优先采用Spring
- 其次采用 apache commons
- **数据提交统一使用form表单方式提交**

强制要求：

Content-Type: application/x-www-form-urlencoded;charset=UTF-8

框架中引用的 Apache commons

```xml
<!-- Apache commons -->
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.3</version>
</dependency>
<dependency>
    <groupId>commons-net</groupId>
    <artifactId>commons-net</artifactId>
    <version>3.6</version>
</dependency>
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.3</version>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-text</artifactId>
    <version>1.1</version>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-compress</artifactId>
    <version>1.14</version>
</dependency>
```

```xml
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.6</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.5</version>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-math3</artifactId>
    <version>3.6.1</version>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-collections4</artifactId>
    <version>4.1</version>
</dependency>
```

注意事项

- activeMQ-all中包含log4j2相关类，与平台中使用的log4j2冲突，请不要使用该jar,建议使用activemq-client.

开发框架及使用技术介绍

- Spring Data JPA 使用
- 动态HQL使用
- JPA CriteriaQuery动态查询
- Hibernate Validator介绍
- Log4J2使用
- XSS使用介绍
- 限流功能
- 配置文件加密及配置路径说明
- **sockjs-web实时通信协议**

Spring Data JPA使用

- 1、通过方法名创建查询
- 2、使用@Query创建查询
- 3、通过调用JPA命名查询语句创建查询

创建查询的顺序

Spring Data JPA 在为接口创建代理对象时，如果发现同时存在多种上述情况可用，它该优先采用哪种策略呢？为此，<jpa:repositories> 提供了 query-lookup-strategy 属性，用以指定查找的顺序。它有如下三个取值：

- create --- 通过解析方法名字来创建查询。即使有符合的命名查询，或者方法通过 @Query 指定的查询语句，都将会被忽略。
- create-if-not-found --- 如果方法通过 @Query 指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则通过解析方法名字来创建查询。这是 query-lookup-strategy 属性的默认值。
- use-declared-query --- 如果方法通过 @Query 指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则抛出异常。

1、通过方法名创建查询

通过解析方法名创建查询

通过前面的例子，读者基本上对解析方法名创建查询的方式有了一个大致的了解，这也是 Spring Data JPA 吸引开发者的一个很重要的因素。该功能其实并非 Spring Data JPA 首创，而是源自一个开源的 JPA 框架 Hades，该框架的作者 Oliver Gierke 本身又是 Spring Data JPA 项目的 Leader，所以把 Hades 的优势引入到 Spring Data JPA 也就是顺理成章的了。

框架在进行方法名解析时，会先把方法名多余的前缀截取掉，比如 find、findBy、read、readBy、get、getBy，然后对剩下部分进行解析。并且如果方法的最后一个参数是 Sort 或者 Pageable 类型，也会提取相关的信息，以便按规则进行排序或者分页查询。

在创建查询时，我们通过在方法名中使用属性名称来表达，比如 findByUserAddressZip ()。框架在解析该方法时，首先剔除 findBy，然后对剩下的属性进行解析，详细规则如下（此处假设该方法针对的域对象为 AccountInfo 类型）：

- 先判断 userAddressZip （根据 POJO 规范，首字母变为小写，下同）是否为 AccountInfo 的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，继续第二步；
- 从右往左截取第一个大写字母开头的字符串（此处为 Zip），然后检查剩下的字符串是否为 AccountInfo 的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，则重复第二步，继续从右往左截取；最后假设 user 为 AccountInfo 的一个属性；
- 接着处理剩下部分（ AddressZip ），先判断 user 所对应的类型是否有 addressZip 属性，如果有，则表示该方法最终是根据 "AccountInfo.user.addressZip" 的取值进行查询；否则继续按照步骤 2 的规则从右往左截取，最终表示根据 "AccountInfo.user.address.zip" 的值进行查询。

可能会存在一种特殊情况，比如 AccountInfo 包含一个 user 的属性，也有一个 userAddress 属性，此时会存在混淆。读者可以明确在属性之间加上 "_" 以显式表达意图，比如 "findByUser_AddressZip()" 或者 "findByUserAddress_Zip()"。

在查询时，通常需要同时根据多个属性进行查询，且查询的条件也格式各样（大于某个值、在某个范围等等），Spring Data JPA 为此提供了一些表达条件查询的关键字，大致如下：

- And --- 等价于 SQL 中的 and 关键字，比如 findByUsernameAndPassword(String user, Striang pwd)；
- Or --- 等价于 SQL 中的 or 关键字，比如 findByUsernameOrAddress(String user, String addr)；
- Between --- 等价于 SQL 中的 between 关键字，比如 findBySalaryBetween(int max, int min)；
- LessThan --- 等价于 SQL 中的 "<"，比如 findBySalaryLessThan(int max)；
- GreaterThan --- 等价于 SQL 中的">"，比如 findBySalaryGreaterThan(int min)；
- IsNull --- 等价于 SQL 中的 "is null"，比如 findByUsernameIsNull()；
- IsNotNull --- 等价于 SQL 中的 "is not null"，比如 findByUsernameIsNotNull()；
- NotNull --- 与 IsNotNull 等价；
- Like --- 等价于 SQL 中的 "like"，比如 findByUsernameLike(String user)；
- NotLike --- 等价于 SQL 中的 "not like"，比如 findByUsernameNotLike(String user)；
- OrderBy --- 等价于 SQL 中的 "order by"，比如 findByUsernameOrderBySalaryAsc(String user)；
- Not --- 等价于 SQL 中的 "！ ="，比如 findByUsernameNot(String user)；
- In --- 等价于 SQL 中的 "in"，比如 findByUsernameIn(Collection<String> userList)，方法的参数可以是 Collection 类型，也可以是数组或者不定长参数；
- NotIn --- 等价于 SQL 中的 "not in"，比如 findByUsernameNotIn(Collection<String> userList)，方法的参数可以是 Collection 类型，也可以是数组或者不定长参数；

3.4.5. Limiting query results

The results of query methods can be limited via the keywords `first` or `top`, which can be used interchangeably. An optional numeric value can be appended to top/first to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed.

Example 18. Limiting the result size of a query with `Top` and `First`

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

The limiting expressions also support the `Distinct` keyword. Also, for the queries limiting the result set to one instance, wrapping the result into an `Optional` is supported.

If pagination or slicing is applied to a limiting query pagination (and the calculation of the number of pages available) then it is applied within the limited result.

2、使用 @Query 创建查询

使用 @Query 创建查询

@Query 注解的使用非常简单，只需在声明的方法上面标注该注解，同时提供一个 JP QL 查询语句即可，如下所示：

清单 16. 使用 @Query 提供自定义查询语句示例

```
public interface UserDao extends Repository<AccountInfo, Long> {

@Query("select a from AccountInfo a where a.accountId = ?1")
public AccountInfo findByAccountId(Long accountId);

   @Query("select a from AccountInfo a where a.balance > ?1")
public Page<AccountInfo> findByBalanceGreaterThan(
Integer balance,Pageable pageable);
}
```

很多开发者在创建 JP QL 时喜欢使用命名参数来代替位置编号，@Query 也对此提供了支持。JP QL 语句中通过":变量"的格式来指定参数，同时在方法的参数前面使用 @Param 将方法参数与 JP QL 中的命名参数对应，示例如下：

清单 17. @Query 支持命名参数示例

```
public interface UserDao extends Repository<AccountInfo, Long> {

public AccountInfo save(AccountInfo accountInfo);

@Query("from AccountInfo a where a.accountId = :id")
public AccountInfo findByAccountId(@Param("id")Long accountId);

  @Query("from AccountInfo a where a.balance > :balance")
  public Page<AccountInfo> findByBalanceGreaterThan(
@Param("balance")Integer balance,Pageable pageable);
}
```

此外，开发者也可以通过使用 @Query 来执行一个更新操作，为此，我们需要在使用 @Query 的同时，用 @Modifying 来将该操作标识为修改查询，这样框架最终会生成一个更新的操作，而非查询。如下所示：

清单 18. 使用 @Modifying 将查询标识为修改查询

```
@Modifying
@Query("update AccountInfo a set a.salary = ?1 where a.salary < ?2")
public int increaseSalary(int after, int before);
```

3、通过调用JPA命名查询语句创建查询

通过调用 JPA 命名查询语句创建查询

命名查询是 JPA 提供的一种将查询语句从方法体中独立出来，以供多个方法共用的功能。Spring Data JPA 对命名查询也提供了很好的支持。用户只需要按照 JPA 规范在 orm.xml 文件或者在代码中使用 @NamedQuery（或 @NamedNativeQuery）定义好查询语句，唯一要做的就是为该语句命名时，需要满足”DomainClass.methodName()”的命名规则。假设定义了如下接口：

清单 19. 使用 JPA 命名查询时，声明接口及方法时不需要什么特殊处理

```
1  public interface UserDao extends Repository<AccountInfo, Long> {
2
3  ......
4
5  public List<AccountInfo> findTop5();
6  }
```

如果希望为 findTop5() 创建命名查询，并与之关联，我们只需要在适当的位置定义命名查询语句，并将其命名为 "AccountInfo.findTop5"，框架在创建代理类的过程中，解析到该方法时，优先查找名为 "AccountInfo.findTop5" 的命名查询定义，如果没有找到，则尝试解析方法名，根据方法名字创建查询。

动态HQL使用

动态构造sql示例:

```
String xsql = "select * from user where 1=1
    /~ and username = {username} ~/
    /~ and password = {password} ~/
    /~ and age = [age] ~/"
    /~ and sex = [sex] ~/"
```

```
Map filters = new HashMap();
filters.put("username", "nuctech");
filters.put("age", "12");
filters.put("sex", "");
```

等同于

```
select * from user where 1=1 and username=:username and age=12
```

相关符号介绍:

/~ segment... ~/ 为一个条件代码块

{key} 过滤器中起标记作用的key,作为后面可以替换为sql的?,或是hql的:username标记

[key] 将直接替换为key value

```
数据类型转换示例:
select * from user where and 1=1 /~ age={age?int} ~/
将会将Map filters中key=age的值转换为int类型
```

JPA CriteriaQuery动态查询

- 类似于Hibernate createQuery查询

```
1.  public InspectVehicle findOne(InspectVehicle info) {
2.      Map<String, Object> eqMap = new HashMap<>();
3.      eqMap.put("passed", info.getPassed());
4.      eqMap.put("driverTag", info.getDriverTag());
5.      eqMap.put("position.site", info.getPosition().getSite());
6.      return inspectVehicleDao.findOne(eqMap, null, null, null);
7.  }
```

Hibernate Validator介绍

Bean Validation 2.0 support

表 1. Bean Validation 中内置的 constraint

Constraint	详细信息
@Null	被注释的元素必须为 null
@NotNull	被注释的元素必须不为 null
@AssertTrue	被注释的元素必须为 true
@AssertFalse	被注释的元素必须为 false
@Min(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@Max(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@DecimalMin(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@DecimalMax(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@Size(max, min)	被注释的元素的大小必须在指定的范围内
@Digits (integer, fraction)	被注释的元素必须是一个数字，其值必须在可接受的范围内
@Past	被注释的元素必须是一个过去的日期
@Future	被注释的元素必须是一个将来的日期
@Pattern(value)	被注释的元素必须符合指定的正则表达式

表 2. Hibernate Validator 附加的 constraint

Constraint	详细信息
@Email	被注释的元素必须是电子邮箱地址
@Length	被注释的字符串的大小必须在指定的范围内
@NotEmpty	被注释的字符串的必须非空
@Range	被注释的元素必须在合适的范围内

一个 constraint 通常由 annotation 和相应的 constraint validator 组成，它们是一对多的关系。也就是说可以有多个 constraint validator 对应一个 annotation。在运行时，Bean Validation 框架本身会根据被注释元素的类型来选择合适的 constraint validator 对数据进行验证。
有些时候，在用户的应用中需要一些更复杂的 constraint。Bean Validation 提供扩展 constraint 的机制。可以通过两种方法去实现，一种是组合现有的 constraint 来生成一个更复杂的 constraint，另外一种是开发一个全新的 constraint。

分组验证及分组顺序

```java
@RequestMapping("/save")
public String save(@Validated({Second.class}) User user, BindingResult result) {
    if(result.hasErrors()) {
        return "error";
    }
    return "success";
}
```

```java
@GroupSequence({First.class, Second.class, User.class})
public class User implements Serializable {
    private Long id;

    @Length(min = 5, max = 20, message = "{user.name.length.illegal}", groups = {First.class})
    @Pattern(regexp = "[a-zA-Z]{5,20}", message = "{user.name.illegal}", groups = {Second.class})
    private String name;

    private String password;
}
```

即通过@Validate注解标识要验证的分组；如果要验证两个的话，可以这样@Validated({First.class, Second.class})。

级联验证

```
1.  public class User {
2.
3.      @Valid
4.      @ConvertGroup(from=First.class, to=Second.class)
5.      private Organization o;
6.
7.  }
```

1、级联验证只要在相应的字段上加@Valid即可，会进行级联验证；@ConvertGroup的作用是当验证o的分组是First时，那么验证o的分组是Second，即分组验证的转换。

Log4J2使用

- 统一使用slf4j接口

```
<Appenders>
  <Appender type="Console" name="STDOUT">
    <Layout type="PatternLayout" pattern="%m MDC%X%n"/>
    <Filters>
      <Filter type="MarkerFilter" marker="FLOW" onMatch="DENY" onMismatch="NEUTRAL"/>
      <Filter type="MarkerFilter" marker="EXCEPTION" onMatch="DENY" onMismatch="ACCEPT"/>
    </Filters>
  </Appender>
  <Appender type="Console" name="FLOW">
    <Layout type="PatternLayout" pattern="%C{1}.%M %m %ex%n"/><!-- class and line number -->
    <Filters>
      <Filter type="MarkerFilter" marker="FLOW" onMatch="ACCEPT" onMismatch="NEUTRAL"/>
      <Filter type="MarkerFilter" marker="EXCEPTION" onMatch="ACCEPT" onMismatch="DENY"/>
    </Filters>
  </Appender>
  <Appender type="File" name="File" fileName="${filename}">
    <Layout type="PatternLayout">
      <Pattern>%d %p %C{1.} [%t] %m%n</Pattern>
    </Layout>
  </Appender>
</Appenders>
```

```
38          logger.info("HTTP request URL:  {}", request.getRequestURL());
39          if (handler.getClass().isAssignableFrom(HandlerMethod.class)) {
40              Map<String, String[]> paramMap = request.getParameterMap();
41              logger.info("HTTP request parameters:  {}", paramMap);
42              logger.info(MarkerFactory.getMarker("FLOW"), "HTTP request parameters:  {}", paramMap);
43              HandlerMethod handlerMethod = (HandlerMethod) handler;
44              OperateLog operateLog = ((HandlerMethod) handler).getMethodAnnotation(OperateLog.class);
45              if (operateLog != null) {
46                  Method method = handlerMethod.getMethod();
47                  logger.info("operateLog: {}, controllerName: {}, methedName: {}", operateLog,
48                          method.getDeclaringClass().getSimpleName(), method.getName());
49                  logger.info("Method call start time: {}", System.currentTimeMillis());
50              }
51          }
52
```

XSS使用介绍

- application/json请求xss为强制转换
- web.xml中需添加

```
<filter>
    <filter-name>xssFilter</filter-name>
    <filter-class>com.nuctech.platform.common.web.xss.XssHttpServletFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>xssFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

限流功能

- dispatcher配置文件中需添加

```
<!-- 请求速率限制拦截器 : IP流量显示 and 注解QPS限制，IP流量限制优先级最高 -->
<bean class="com.nuctech.platform.common.limit.RateLimiterInterceptor">
    <!-- threshold: 同一个IP每分钟最大访问量,
         maximumSize: 同时最大可访问IP数量,
         enable: false/true，true为启用全局ip限制-->
    <constructor-arg name="threshold" value="10" type="java.lang.Integer" />
    <constructor-arg name="maximumSize" value="1024" type="java.lang.Long" />
    <constructor-arg name="enable" value="true" type="java.lang.Boolean" />
</bean>
```

- qps功能启用需加注解

  @QPSAccessRateLimiter(rate = 10.0d)

配置文件加密及路径说明

```
    <!-- 数据库连接池配置 -->
    <bean id="propertyConfigurer"
        class="com.nuctech.platform.common.util.EncryptPropertyPlaceholderConfigurer">
        <property name="randomKey" value="false"/>
        <property name="propertyNames">
            <array value-type="java.lang.String">
                <value>jdbc.username</value>
                <value>jdbc.password</value>
            </array>
        </property>
        <property name="locations">
            <list>
                <value>classpath:etc/db/jdbc_psql.properties</value>
                <!-- <value>file:${catalina.base}/jdbc_psql.properties</value> -->
            </list>
        </property>
    </bean>
</beans>
```

旧版加密为false，新版为true，默认为true
新旧版本以0.5.0为分界线

加密的字段
jdbc.username、jdbc.password为默认内置。可不写

外置配置文件，catalina.base、cataline.home为tomcat目录

使用该类，可以使用工具进行加解密，工具路径：
http://172.16.254.102/svn/DEVP/01-开发工具/PropertiesToolsSetup.exe（新版）
http://172.16.254.102/svn/DEVP/01-开发工具/ PropertiesToolsSetup(before0.5.0).exe（旧版）

sockjs-web实时通信应用解决方案

- **socksjs**
- 客户端和服务器端api尽可能简洁，尽量靠近websocket api
- 支持服务端扩展和负载均衡技术
- 传输层应该全面支持跨域通信
- **如果受到代理服务器的限制**，传输层能优雅地从一种方式回退到另一种方式
- 尽可能快地建立连接
- 客户端只是纯粹的JavaScript，不需要flash
- 客户端JavaScript必须经过严格的测试
- 服务器端代码尽可能简单，降低用另一种语言重写server的代价
- 实际上sockjs的目标也就是sockjs具有的特点。

在SockJS: WebSocket emulation done right一文中对sockjs的特点进行了具体阐述。

sockjs几个特点

- 跨域通信
- 负载均衡
- Prefix-based sticky sessions
- JSESSIONID cookie sticky sessions
- 健壮的传输协议

跨域通信

sockjs服务器端支持跨域通信，意味着我们可以将sockjs server独立出来，把它放在与web主站点不同的域名之上，实际上这是比较合理的部署策略。关于**跨域**，也是一个比较大的话题，其中有一个机制叫cors(跨域资源共享)主要解决JavaScript不能跨域请求的问题。sockjs服务器应该支持这种机制。

负载均衡

无论server端优化得再好，一个sockjs server的处理能力都是有限的，我们更需要的是一种可扩展的解决方案。一种非常简单的方法是把每一个sockjs server放到一个不同的域名之下，如sockjs1.example.com和sockjs2.example.com，允许客户端随机选择一个server。

Prefix-based sticky sessions

在sockjs中，一个典型的url如下：

http://localhost:8000/chat/<serverid>/<sessionid>/

url中的第二个参数sessionid，必须是一个随机字符串，唯一标识一个session。第一个参数serverid,主要应用于负载均衡目的。负载均衡器可以利用这个serverid参数作为一个线索，进行负载均衡分流。具体使用方面，参考HAProxy的一个配置参考文件，其中关键的配置在于

balance uri depth 2

JSESSIONID cookie sticky sessions

另外一种负载均衡配置方案,主要利用含有jsessionid的cookie。这个cookie由socketjs server进行设置，当response到达负载均衡器的时候，jessionid会被加上一个额外的前缀或者后缀，具体原理方面可以参考阅读LOAD BALANCING, AFFINITY, PERSISTENCE, STICKY SESSIONS: WHAT YOU NEED TO KNOW一文。

健壮的传输协议

我们知道html5 的websocket协议定义了websocket api使得网页可以利用websocket协议和远端主机进行全双工通信。websocket协议应该是最快，最好的web通信协议，已被大多数的浏览器所支持。那么为什么还需要sockjs进行封装？

在真实的网络世界中，实际上有着非常复杂的网络拓扑结构，在浏览器和server之间，含有很多的中间节点，包括路由器，代理服务器，反向代理服务器，负载均衡器等等。即使html5 websocket协议已经成为了标准，但是这些中间节点并不一定会遵守这些标准，还有很大可能会阻止websocket handshake的过程，结果无法建立websocket连接。

How HTML5 Web Sockets Interact With Proxy Servers一文中提到了websocket协议和代理服务器的“**不友善关系**”，源于代理服务器对websocket handshake的阻挠和对**长连接,空闲连接**的关闭处理，让我们看到了如果只是直接利用websocket协议，实在是困难重重。

sockjs的出现，实际是为了解决这个问题，使得人们可以建立健壮的web实时通信程序。

sockjs服务器传输协议不仅提供了websocket协议的支持，还提供了流传输**Streaming**和轮询**Polling**，其中又包括多种底层传输方案，如：
xhr，**xhr_streaming**，**jsonp**，**eventsource**，**htmlfile**。每一种传输方案，其实都值得开辟一个章节来大写特写。

如果浏览器客户端js，采用websocket连接不上服务器，它可以回退选择其他传输方案，那么确保总可以利用一种传输协议，连接到服务器。那么开发者就不需要理睬那些**可恶的中间节点**了。