

Smart 2.0 用户手册

2014 年 1 月

目 录

1	项目简介	4
2	开发工具	5
2.1	准备开发工具	5
2.2	搭建开发环境	5
3	快速上手	7
3.1	创建项目	7
3.2	修改 pom.xml 文件	7
3.3	修改 web.xml 文件	9
3.4	新增 config.properties 文件	9
3.5	新增 log4j.properties 文件	9
3.6	新建 index.html 文件	10
3.7	访问应用	10
4	技术架构	12
4.1	技术选型	12
4.2	项目依赖	12
4.3	系统架构	13
5	核心功能	15
5.1	MVC	15
5.2	IOC	17
5.3	AOP	18
5.4	ORM	19
5.5	DAO	21
5.6	事务控制	22
5.7	单元测试	23
5.8	文件上传	24
5.9	DataConext	25
6	相关插件	27
6.1	Cache	27
6.2	WebService	29
6.3	Mail	30
6.4	I18N	32

6.5	Job	34
6.6	Hessian	36
6.7	Template	37
7	代码生成器	38
7.1	安装 Smart SDK	38
7.2	使用 Smart SDK 命令	38
7.3	参考资料	39

1 项目简介

Smart 开源框架可用于快速开发中小规模的 企业应用 或 网站应用

它是一款轻量级 **Java Web** 框架

- ✓ 不到 3000 行代码实现 IOC、AOP、ORM、DAO、MVC 等功能
- ✓ 基于 Servlet 3.0 规范
- ✓ 使用 Java 注解取代 XML 配置

它使应用充分做到“前后端分离”

- ✓ 客户端可使用 HTML 或 JSP 作为视图模板
- ✓ 服务端可发布 REST 服务
- ✓ 通过 Ajax 获取服务端数据并进行界面渲染

源码地址: <http://git.oschina.net/huangyong/smart-framework>

应用示例: <http://git.oschina.net/huangyong/smart-sample>

系列博文: <http://my.oschina.net/huangyong/blog/158380>

2 开发工具

2.1 准备开发工具

Java 虚拟机	JDK 1.6
集成开发环境	Eclipse 或 IntelliJ IDEA
项目构建	Maven
Web 服务器	Tomcat 7.0 、 Apache （可选）
数据库	MySQL （服务器）、 Navicat （客户端）
源码版本控制	Git （服务器）、 SourceTree （客户端）

点击工具名称上的链接可进入下载页面。

2.2 搭建开发环境

2.2.1 搭建 Maven 开发环境

Smart 的相关 jar 包托管在[开源中国](#)（以下简称 OSC）的 Maven 仓库中，若使用 Maven 开发，则需将 OSC 的 Maven 仓库地址添加到 Maven 的 settings.xml 配置文件中，见如下代码片段：

```
...
<mirrors>
  <mirror>
    <id>osc</id>
    <mirrorOf>central</mirrorOf>
    <url>http://maven.oschina.net/content/groups/public/</url>
  </mirror>
  <mirror>
    <id>osc_thirdparty</id>
    <mirrorOf>thirdparty</mirrorOf>
    <url>http://maven.oschina.net/content/repositories/thirdparty/</url>
  </mirror>
</mirrors>
<profiles>
  <profile>
    <id>osc</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <repositories>
      <repository>
        <id>osc</id>
        <url>http://maven.oschina.net/content/groups/public/</url>
      </repository>
      <repository>
        <id>osc_thirdparty</id>
```

```

        <url>http://maven.oschina.net/content/repositories/thirdparty/</url>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>osc</id>
        <url>http://maven.oschina.net/content/groups/public/</url>
    </pluginRepository>
</pluginRepositories>
</profile>
</profiles>

```

随后，可在 pom.xml 中使用如下依赖：

```

...
<properties>
    <smart.version>2.0</smart.version>
</properties>
...
<dependencies>
    <dependency>
        <groupId>com.smart</groupId>
        <artifactId>smart-framework</artifactId>
        <version>${smart.version}</version>
    </dependency>
</dependencies>
...

```

可通过以下地址获取 Smart 的相关 jar 包：

<http://maven.oschina.net/index.html#nexus-search;gav~com.smart~~~~>

2.2.2 搭建源码开发环境

Smart 的相关源码托管在 OSC 的 Git 仓库中，若需搭建源码开发环境，可使用如下 Git 命令下载 Smart Framework 的源码：

```
git clone http://git.oschina.net/huangyong/smart-framework.git
```

随后，可使用以下 Maven 命令将 Smart Framework 的 jar 包安装到 Maven 本地仓库中：

```
mvn install
```

2.2.3 参考资料

✓ Maven 那点事儿：<http://my.oschina.net/huangyong/blog/194583>

3 快速上手

3.1 创建项目

输入以下 Maven 命令：

```
mvn archetype:generate -DinteractiveMode=false -DarchetypeArtifactId=maven-archetype-webapp -DgroupId=com.smart
-DartifactId=smart-demo -Dversion=1.0
```

随后可使用 Eclipse 或 IDEA 直接打开已创建的 Maven 项目。

✓ Eclipse: Import → Maven → Existing Maven Projects → 选择 Root Directory → 点击 Finish 按钮

✓ IDEA: File → Open → 选择 pom.xml → 点击 OK 按钮

3.2 修改 pom.xml 文件

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <smart.version>1.0</smart.version>
  </properties>

  <groupId>com.smart</groupId>
  <artifactId>smart-demo</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>

  <dependencies>
    <!-- JUnit -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <!-- MySQL -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.25</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

```
</dependency>
<!-- Servlet -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
</dependency>
<!-- JSTL -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
    <scope>runtime</scope>
</dependency>
<!-- Smart -->
<dependency>
    <groupId>com.smart</groupId>
    <artifactId>smart-framework</artifactId>
    <version>${smart.version}</version>
</dependency>
</dependencies>

<build>
    <plugins>
        <!-- Compile -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.5.1</version>
            <configuration>
                <source>1.6</source>
                <target>1.6</target>
            </configuration>
        </plugin>
        <!-- Test -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.15</version>
            <configuration>
                <skipTests>true</skipTests>
            </configuration>
        </plugin>
        <!-- Package -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.4</version>
```



```

        <configuration>
            <warName>${project.artifactId}</warName>
        </configuration>
    </plugin>
    <!-- Tomcat -->
    <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
    </plugin>
</plugins>
</build>

</project>

```

3.3 修改 web.xml 文件

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

</web-app>

```

在 web.xml 中无需做任何配置，因为使用了 Servlet 3.0 规范。

3.4 新增 config.properties 文件

在 src/main/resources 目录下新增 config.properties 文件，内容如下：

```

app.name=smart-demo
app.package=com.smart.demo
app.www_path=/www/
app.home_page=/www/index.html

jdbc.type=mysql
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/demo
jdbc.username=root
jdbc.password=root

```

需要在 MySQL 中创建一个名为 demo 的数据库，字符集是 UTF-8，并根据实际情况修改用户名与密码。

3.5 新增 log4j.properties 文件

在 src/main/resources 目录下新增 log4j.properties 文件，内容如下：

```

log4j.rootLogger=INFO,console,file

```

```
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.Target=System.out
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%m%n

log4j.appender.file=org.apache.log4j.DailyRollingFileAppender
log4j.appender.file.File=${catalina.base}/logs/smart-demo/log
log4j.appender.file.DatePattern='_ 'yyyyMMdd
log4j.appender.file.encoding=UTF-8
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{HH:mm:ss,SSS} %p %c (%L) - %m%n

log4j.logger.com.smart.demo=DEBUG
```

3.6 新建 index.html 文件

在 src/main/webapp/www 目录下新增 index.html 文件，内容如下：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Smart</title>
</head>
<body>

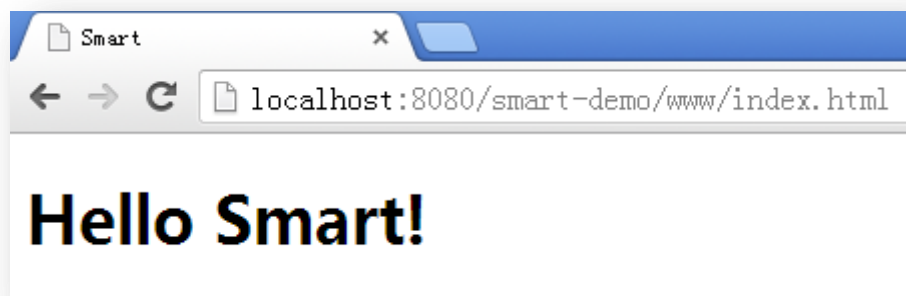
<h1>Hello Smart!</h1>

</body>
</html>
```

3.7 访问应用

在浏览器中输入以下地址：

<http://localhost:8080/smart-demo/www/index.html>



4 技术架构

4.1 技术选型

Smart Framework 在技术选型方面做了多方面的考虑，必须拥有较高的市场占有率，并且具有丰富的参考资料。

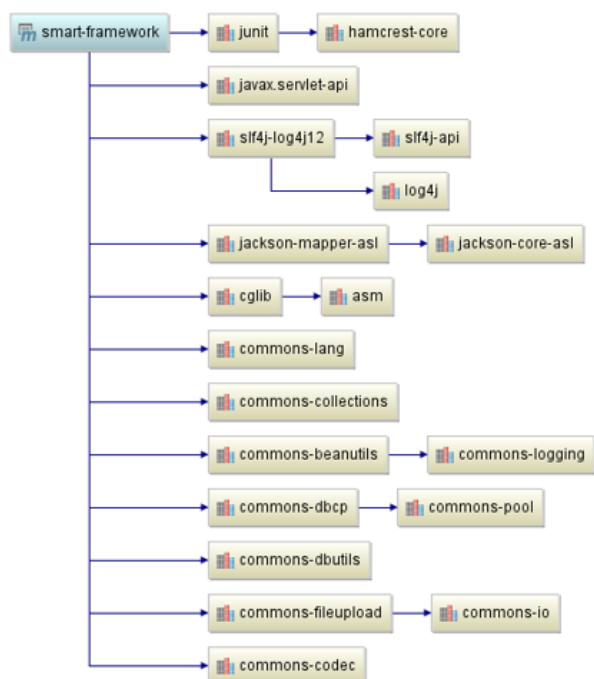
详细的技术选型如下：

Web 框架	使用 Servlet 3.0 规范，可部署在 Tomcat 7.0+ 上
单元测试	使用 JUnit，因为它是 Java 业界市场占有率最高的单元测试框架
数据库连接池	使用 Apache Commons DBCP，因为它稳定而高效
JDBC 封装	使用 Apache Commons DbUtils，因为它足够轻量级，且功能基本够用
文件上传	使用 Apache Commons FileUpload，因为它使用起来非常方便
日志	使用 SLF4J，因为它提供了日志操作的统一接口，可使用 Log4J 或其它工具作为具体实现
JSON 库	使用 Jackson，因为它拥有较高的性能，且市场占有率较高
动态代理	使用 CGLib，因为可以弥补 JDK 动态代理的不足，可在运行时对 class 进行字节码增强

除此以外，也使用了 Apache Commons 的其它知名项目，例如：Lang、Collections、BeanUtils、Codec 等。

4.2 项目依赖

通过以上技术选型，我们可以得知，这些技术都是开源项目，而它们之间存在以下依赖关系：



- 使用 JUnit 进行单元测试
- 基于 Servlet 3.0 规范，可部署在 Tomcat 7.0 上
- 使用 SLF4J 作为日志接口，使用 Log4J 作为日志实现
- 使用 Jackson 作为 JSON 序列化与反序列化
- 使用 CGLib 作为动态代理工具
- 使用 Apache Commons 工具包
- 使用 DBCP 作为数据库连接池
- 使用 DbUtils 封装 JDBC 操作
- 使用 FileUpload 实现文件上传

不难发现，这些项目都拥有较小的体积：

项目	jar 包	文件大小(K)
SLF4J	slf4j-api-1.7.5.jar	26
	slf4j-log4j12-1.7.5.jar	9
Log4J	log4j-1.2.17.jar	479
Jackson	jackson-mapper-asl-1.9.13.jar	763
	jackson-core-asl-1.9.13.jar	227
Cglib	cglib-2.2.2.jar	281
ASM	asm-3.3.1.jar	43
Apache Commons Lang	commons-lang-2.4.jar	256
Apache Commons Collections	commons-collections-3.2.1.jar	562
Apache Commons BeanUtils	commons-beanutils-1.8.3.jar	227
Apache Commons Logging	commons-logging-1.1.1.jar	60
Apache Commons DBCP	commons-dbcp-1.4.jar	157
Apache Commons Pool	commons-pool-1.5.4.jar	94
Apache Commons DbUtils	commons-dbutils-1.5.jar	61
Apache Commons FileUpload	commons-fileupload-1.3.jar	68
Apache Commons IO	commons-io-2.2.jar	170
Apache Commons Codec	commons-codec-1.8.jar	258
		3741

可见，jar 包文件总大小还不到 4 M。

下面再来看看 Smart 框架源码的 LOC（代码行数）统计吧：

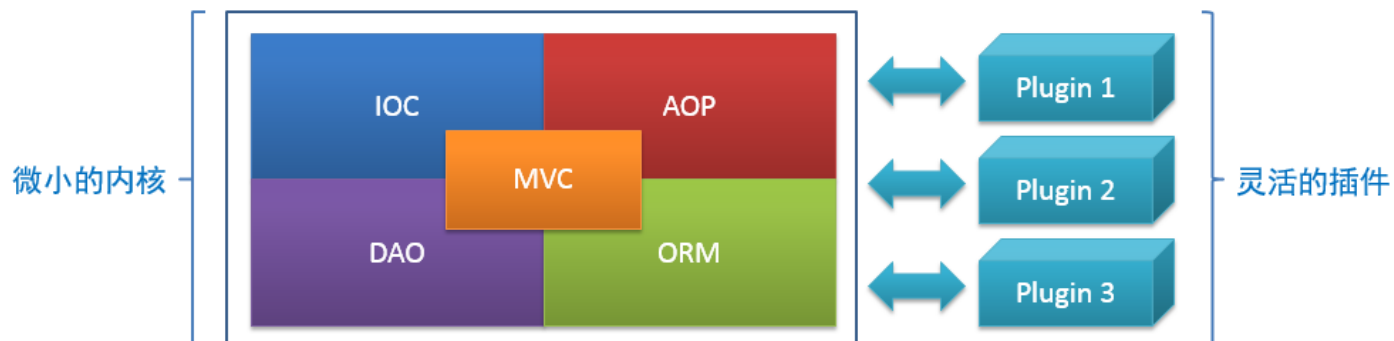
Lines

Total: 3855 lines

Type	Lines	Percentage(%)
Code	2825	73.28%
Code//Comment	41	1.06%
Comment	508	13.18%
Blank	481	12.48%

可见，代码实际总行数（去掉注释与空行）还不到 3000 行。

4.3 系统架构



核心：

MVC	基于 Servlet 3.0 规范
IOC	轻量级 IOC 容器

AOP	轻量级 AOP 框架
ORM	基于 JDBC 规范
DAO	统一的数据访问 API

插件：

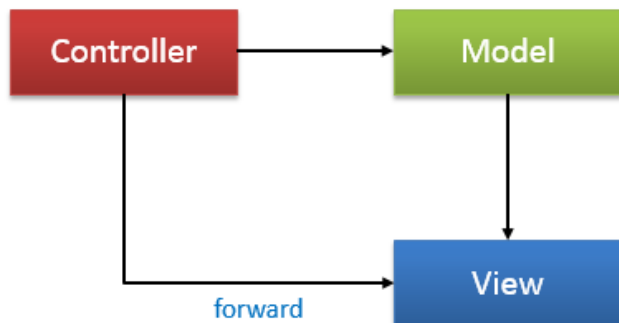
Cache	基于注解或使用 Cache API
WebService	发布与调用 SOAP 服务或 REST 服务
Mail	邮件发送与收取
I18N	国际化多语言包
Job	基于 Quartz 的 cron 表达式的任务调度框架
Hessian	通过 HTTP 传输二进制数据
Template	基于 Velocity 的模板引擎

5 核心功能

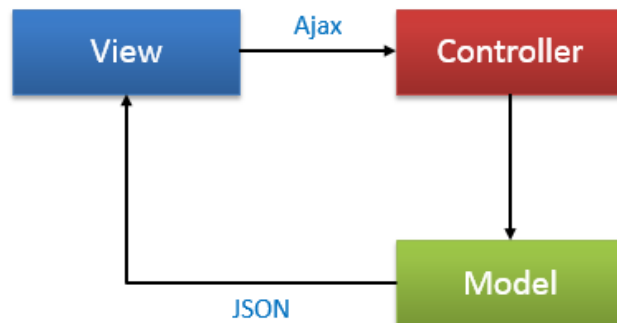
5.1 MVC

5.1.1 MVC 模式

在 Smart 应用中可使用两种 MVC 模式：



正向 MVC
推模式



反向 MVC
拉模式

✓ 正向 MVC

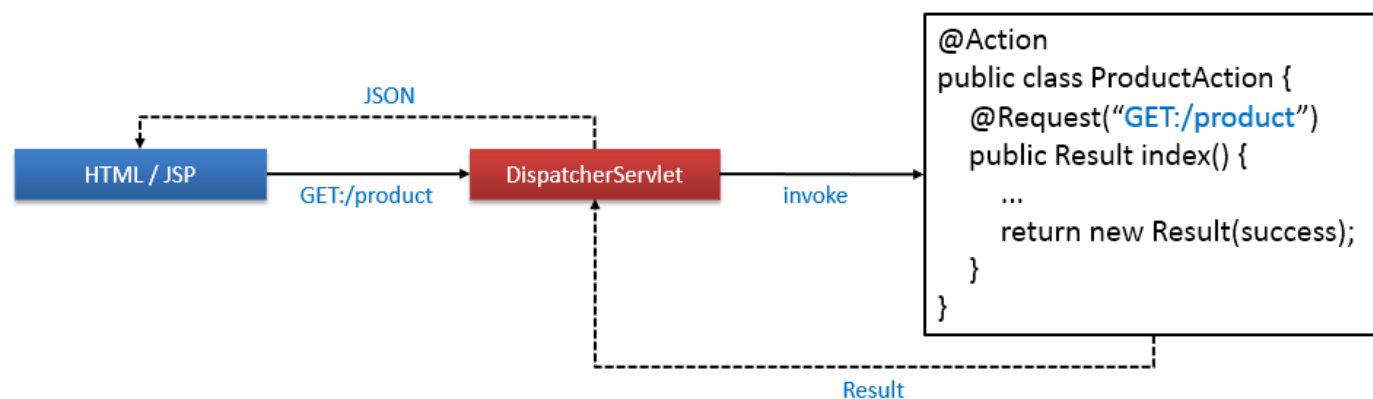
也称为传统 MVC，在 Controller 中获取 Model，并将 Model 转发（forward）到 View 中，这是 推模式，可理解为：服务端将数据推给 View。

✓ 反向 MVC

在 View 上发送 Ajax 请求到 Controller 中，在 Controller 中获取 Model，并将 Model 以 JSON 格式返回给 View，这是 拉模式，可理解为：在 View 中将服务端数据拉回来。

5.1.2 MVC 请求响应过程

以下是一个典型的反向 MVC 请求响应过程：



在 HTML 或 JSP 中发送 GET:/product 请求，该请求被 Smart 的 com.smart.framework.DispatcherServlet 进行处理，根据请求方法与路径调用相应的 Action 方法，最终将结果以 JSON 格式返回。

5.1.3 配置 Action 方法

可使用 `com.smart.framework.annotation.Request` 注解定义请求 URL。

请求 URL 包括两部分：请求方法与 请求路径。

- ✓ 支持 4 中请求方法：GET、POST、PUT、DELETE
- ✓ 请求路径以 “/” 开头
- ✓ 请求路径中可使用占位符，例如：GET:/product/view/{id}

5.1.4 Action 方法参数

Action 方法的参数具有如下规则：

- ✓ 若请求中带有表单数据，则使用 `Map<String, Object>` 参数作为映射。
- ✓ 若请求路径中带有占位符，则使用相应类型的参数作为映射。

示例：

请求路径	Action 方法
POST:/product/search	public Page search(Map<String, Object> fieldMap) { ... }
GET:/product/view/{id}	public Page view(long id) { ... }

5.1.5 Action 方法返回值

可根据具体需求，使用以下两种返回值：

- ✓ 若返回 JSON 数据，则需使用 `com.smart.framework.bean.Result` 对象。
- ✓ 若返回 HTML 或 JSP 页面，则需使用 `com.smart.framework.bean.Page` 对象。

Result 与 Page 的详细 API 如下：

Result	Page
m Result(boolean)	m Page(String)
m error(int) Result	m data(String, Object) Page
m data(Object) Result	p data Map<String, Object>
p error int	p path String
p success boolean	p redirect boolean
p data Object	

5.1.6 参考资料

- ✓ 对 Action 的初步构思：<http://my.oschina.net/huangyong/blog/158470>

- ✓ Action 分发机制实现原理: <http://my.oschina.net/huangyong/blog/158738>
- ✓ 两种 MVC 模式: <http://my.oschina.net/huangyong/blog/169683>
- ✓ 支持“正向 MVC 模式”: <http://my.oschina.net/huangyong/blog/169863>

5.2 IOC

5.2.1 使用 Bean 注解

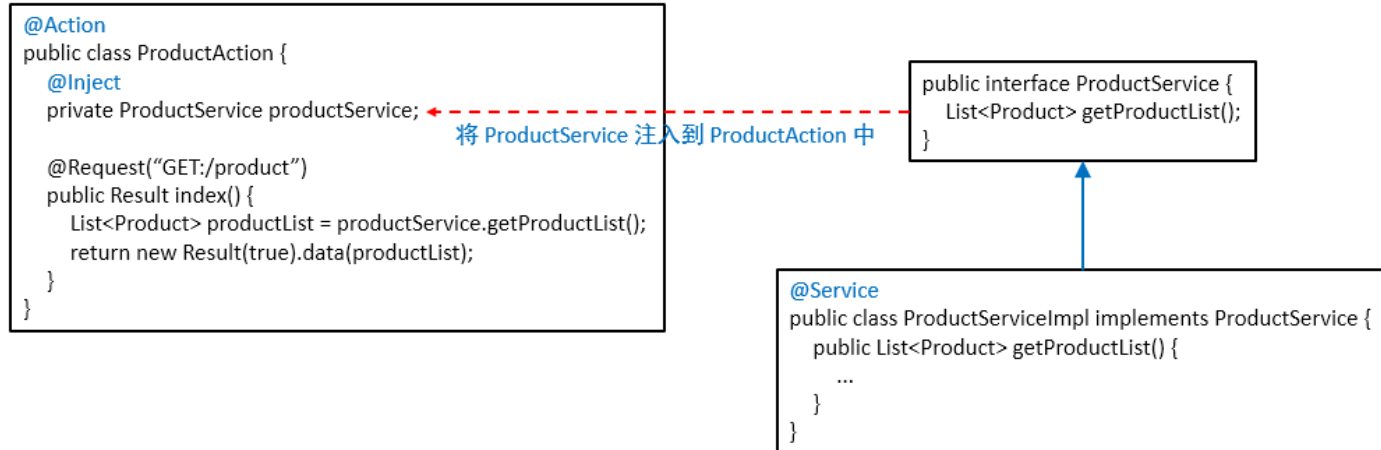
可将 Action、Service 或其它 Bean 可通过注解的方式放入 Smart IOC 容器中进行管理，可使用如下注解：

com.smart.framework.annotation.Bean	用于定义普通 Bean
com.smart.framework.annotation.Action	用于定义在 Action
com.smart.framework.annotation.Service	用于定义 Service
com.smart.framework.annotation.Aspect	用于定义 Aspect（它是 AOP 的切面）

5.2.2 使用 IOC 注解

可使用 com.smart.framework.annotation.Inject 注解进行依赖注入。

在以下代码示例中，定义了一个 Action，并在该 Action 中注入了一个 Service：



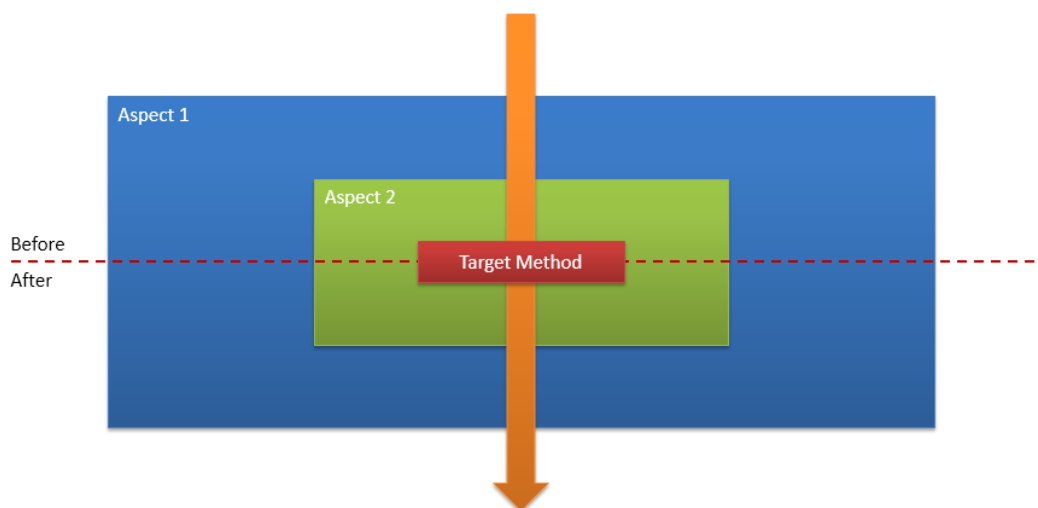
5.2.3 参考资料

- ✓ IOC 实现原理: <http://my.oschina.net/huangyong/blog/158992>
- ✓ 使用“链式代理”实现 AOP: <http://my.oschina.net/huangyong/blog/170494>

5.3 AOP

5.3.1 AOP 原理

可使用 Aspect（切面）拦截特定的目标方法，一个目标方法可被多个 Aspect 拦截，这些 Aspect 构成了一个 AOP Chain，如下图所示：



在调用目标方法的前后都可进行 AOP 拦截，在调用之前，线程首先进入 Aspect 1，然后进入 Aspect 2，最后进入 Target Method。当调用完毕后，线程首先退出 Target Method，然后退出 Aspect 2，最后退出 Aspect 1。

5.3.2 定义 AOP 切面

可使用 `com.smart.framework.annotation.Aspect` 注解定义 AOP 切面：

```
@Aspect(pkg = "com.smart.sample.action")
@Order(0)
public class AccessAspect extends AspectProxy {

    @Override
    public boolean intercept(Class<?> cls, Method method, Object[] params) throws Throwable {
        boolean result = true;
        if (cls == SystemAction.class) {
            result = false;
        }
        return result;
    }

    @Override
    public void before(Class<?> cls, Method method, Object[] params) throws Throwable {
        Long userId = DataContext.Session.get(Constant.USER_ID);
        if (userId == null) {
            WebUtil.setRedirectURL(DataContext.getRequest(), Constant.REDIRECT_URL);
            throw new AccessException();
        }
    }
}
```

```
    }  
  }  
}
```

以上是一个用于访问安全控制的 Aspect，在 intercept 方法里进行拦截条件判断，对于 SystemAction 无需拦截，将返回值设置为 false。可通过 before 方法实现前置拦截，判断 Session 中是否存在 userId，如果不存在则设置重定向 URL，并抛出 AccessException。

在 Aspect 注解中可指定包名（pkg）与类名（cls），在以上示例中仅使用了包名。

5.3.3 配置 AOP 顺序

可使用 com.smart.framework.annotation.Order 注解定义拦截顺序，值越小越先拦截。

5.3.4 AOP 钩子方法

在 AOP 方法拦截时，有一些钩子方法，切面类需继承 com.smart.framework.proxy.AspectProxy 父类，可以有选择性的覆盖父类的方法，这些方法汇总如下：

void begin()	在进入方法时执行
boolean intercept(Class<?> cls, Method method, Object[] params) throws Throwable	设置拦截条件
void before(Class<?> cls, Method method, Object[] params) throws Throwable	在目标方法调用前执行
void after(Class<?> cls, Method method, Object[] params, Object result) throws Throwable	在目标方法调用后执行
void error(Class<?> cls, Method method, Object[] params, Throwable e)	在抛出异常时执行
void end()	在方法执行完毕前执行

5.3.5 参考资料

- ✓ AOP 实现原理：<http://my.oschina.net/huangyong/blog/160769>
- ✓ 访问安全控制解决方案：<http://my.oschina.net/huangyong/blog/173679>
- ✓ Proxy 那点事儿：<http://my.oschina.net/huangyong/blog/159788>
- ✓ AOP 那点事儿：<http://my.oschina.net/huangyong/blog/161338>
- ✓ AOP 那点事儿（续集）：<http://my.oschina.net/huangyong/blog/161402>

5.4 ORM

5.4.1 ORM 映射规则

ORM（Object Relationship Mapping，对象关系映射）即定义 Entity（实体）与 Table（表）之间的映射关系，也就是将面向对象与面向关系进行映射。以下示例中，将 Product 实体映射为 product 表：

```

public class Product extends BaseEntity {

    private long productTypeId;
    private String name;
    private String code;
    private int price;
    private String description;
    private String picture;

    // getter/setter ...
}

```



```

product
id: bigint
product_type_id: bigint
name: varchar(100)
code: char(5)
price: int
description: text
picture: varchar(100)

```

实体类：

- ✓ 必须继承 `com.smart.framework.base.BaseEntity` 父类
- ✓ 实体名与属性名必须为驼峰风格
- ✓ 对于每个属性必须带有 `getter/setter` 方法

表结构：

- ✓ 表名必须均为小写
- ✓ 表名与列名必须为下划线风格

对应规则如下表所示：

实体类	表结构
类名（如：Product）	表名（如：product）
属性名（如：productTypeId）	列名（如：product_type_id）

5.4.2 处理特殊映射

可使用 `com.smart.framework.annotation.Table` 注解映射特殊的表名，可使用 `com.smart.framework.annotation.Column` 注解映射特殊的列名。

5.4.3 参考资料

- ✓ 对 Entity 的初步构思：<http://my.oschina.net/huangyong/blog/158481>
- ✓ Entity 映射机制实现原理：<http://my.oschina.net/huangyong/blog/158916>

5.5 DAO

5.5.1 DAO 的作用

DAO（Data Access Object，数据访问对象）实际上是一种设计模式，它在 Service 层与 JDBC 层之间建立了一座桥梁，可通过 DAO 方便地完成 CRUD（Create/Retrieve/Update/Delete）操作。下图是一个典型的请求调用过程：

















5.5.2 DataSet API

对于单表操作，可使用 `com.smart.framework.DataSet` 工具类，详细的 API 如下：

C DataSet		
	<code>select(Class<T>, String, Object...)</code>	<code>T</code>
	<code>selectList(Class<T>, String, String, Object...)</code>	<code>List<T></code>
	<code>insert(Class<?>, Map<String, Object>)</code>	<code>boolean</code>
	<code>update(Class<?>, Map<String, Object>, String, Object...)</code>	<code>boolean</code>
	<code>delete(Class<?>, String, Object...)</code>	<code>boolean</code>
	<code>selectCount(Class<?>, String, Object...)</code>	<code>long</code>
	<code>selectListForPager(int, int, Class<T>, String, String, Object...)</code>	<code>List<T></code>
	<code>selectMap(Class<T>, String, Object...)</code>	<code>Map<Long, T></code>
	<code>selectColumn(Class<T>, String, String, Object...)</code>	<code>T</code>
	<code>selectColumnList(Class<?>, String, String, String, Object...)</code>	<code>List<T></code>

5.5.3 DataHelper API

对于复杂 SQL 操作，可使用 `com.smart.framework.helper.DBHelper` 工具类，详细的 API 如下：

DBHelper		
	getDataSource()	DataSource
	getConnection()	Connection
	beginTransaction()	void
	commitTransaction()	void
	rollbackTransaction()	void
	getDbType()	String
	queryBean(Class<T>, String, Object...)	T
	queryBeanList(Class<T>, String, Object...)	List<T>
	update(String, Object...)	int
	queryCount(String, Object...)	long
	queryMapList(String, Object...)	List<Map<String, Object>>
	queryColumn(String, String, Object...)	T
	queryColumnList(String, String, Object...)	List<T>
	insertReturnPK(String, Object...)	Serializable

5.6 事务控制

5.6.1 事务控制原理

事务控制（或称为事务管理）实际上也是一种 AOP，也就是说，在执行带有事务的方法前后，分别 **Begin Transaction**（开启事务）与 **Commit Transaction**（提交事务），当遇到异常时需要 **Rollback Transaction**（回滚事务）。

5.6.2 事务控制注解

可使用 `com.smart.framework.annotation.Transaction` 注解定义在需要进行事务控制的方法上，若某方法对数据库进行了写操作（包括：`insert`、`update`、`delete`），则需要考虑事务控制。

以下是一个事务控制的示例：

```
@Service
public class ProductServiceImpl implements ProductService {

    @Override
    @Transaction
    public boolean createProduct(Map<String, Object> fieldMap, Multipart multipart) {
        if (multipart != null) {
            fieldMap.put("picture", multipart.getFileName());
        }
        boolean result = DataSet.insert(Product.class, fieldMap);
        if (result) {
            UploadHelper.uploadFile(Tool.getBasePath(), multipart);
        }
    }
}
```

```
    }  
    return result;  
}  
...
```

5.6.3 参考资料

✓ 事务管理实现原理: <http://my.oschina.net/huangyong/blog/159852>

5.7 单元测试

5.7.1 单元测试使用方法

可使用 JUnit 完成单元测试, 但需要继承 `com.smart.framework.base.BaseTest` 父类, 该父类中提供了一个 `initSQL` 方法, 用于初始化测试数据脚本。

5.7.2 配置单元测试顺序

此外, 可使用 `com.smart.framework.annotation.Order` 注解定义测试方法的执行顺序, 因为 JUnit 不是从上往下的一次执行的。

5.7.3 单元测试示例

```
public class ProductServiceTest extends BaseTest {  
  
    private ProductService productService = BeanHelper.getBean(ProductServiceImpl.class);  
  
    @BeforeClass  
    @AfterClass  
    public static void init() {  
        initSQL("sql/product.sql");  
    }  
  
    @Test  
    @Order(1)  
    public void getProductBeanPagerTest() {  
        int pageNumber = 1;  
        int pageSize = 10;  
        String name = "";  
  
        Pager<ProductBean> productBeanPager = productService.getProductBeanPager(pageNumber,  
        pageSize, name);  
        Assert.assertNotNull(productBeanPager);  
        Assert.assertEquals(productBeanPager.getRecordList().size(), 10);  
        Assert.assertEquals(productBeanPager.getTotalRecord(), 12);  
    }  
}
```

```
}

@Test
@Order(2)
public void getProductBeanTest() {
    long productId = 1;

    ProductBean productBean = productService.getProductBean(productId);
    Assert.assertNotNull(productBean);
    Assert.assertNotNull(productBean.getProduct());
    Assert.assertNotNull(productBean.getProductType());
}
...
```

5.7.4 参考资料

像这样做单元测试: <http://my.oschina.net/huangyong/blog/162325>

5.8 文件上传

5.8.1 文件上传原理

可通过表单实现文件上传, 该表单可包含普通表单字段与文件字段。实际上, 文件将通过表单, 以二进制流的方式通过 HTTP 传输到服务端, 这个流在文件上传中被称为 **Multipart**。

5.8.2 文件上传配置

使用文件上传功能, 需要在 `config.properties` 文件做添加如下配置:

```
app.upload_limit=10
```

以上配置表示文件上传限制为 10M。

5.8.3 对表单的限制

用于文件上传的表单需要满足以下条件:

1. `form` 标签必须带有 `method="post"` 属性 (默认是 `get`)
2. `form` 标签必须带有 `enctype="multipart/form-data"` 属性 (默认是 `application/x-www-form-urlencoded`)
3. 文件上传必须使用 `<input type="file">` 标签

5.8.4 文件上传示例

示例: 一个用于文件上传的表单

```
<form action="/product/create" method="post" enctype="multipart/form-data">
```



```
...
<input type="file" id="picture" name="picture">
...
</form>
```

以上省略了一些普通表单字段与表单提交按钮的代码。

示例：提交一个表单同时上传一份文件

```
@Action
public class ProductAction {
...
    @Request("POST:/product/create")
    public Result create(Map<String, Object> fieldMap, Multipart multipart) {
        boolean success = productService.createProduct(fieldMap, multipart);
        return new Result(success);
    }
...
}
```

若同时上传多份文件，则 `Action` 方法里要使用 `List<Multipart>` 参数做映射。

5.8.5 参考资料

✓ 实现文件上传：<http://my.oschina.net/huangyong/blog/161989>

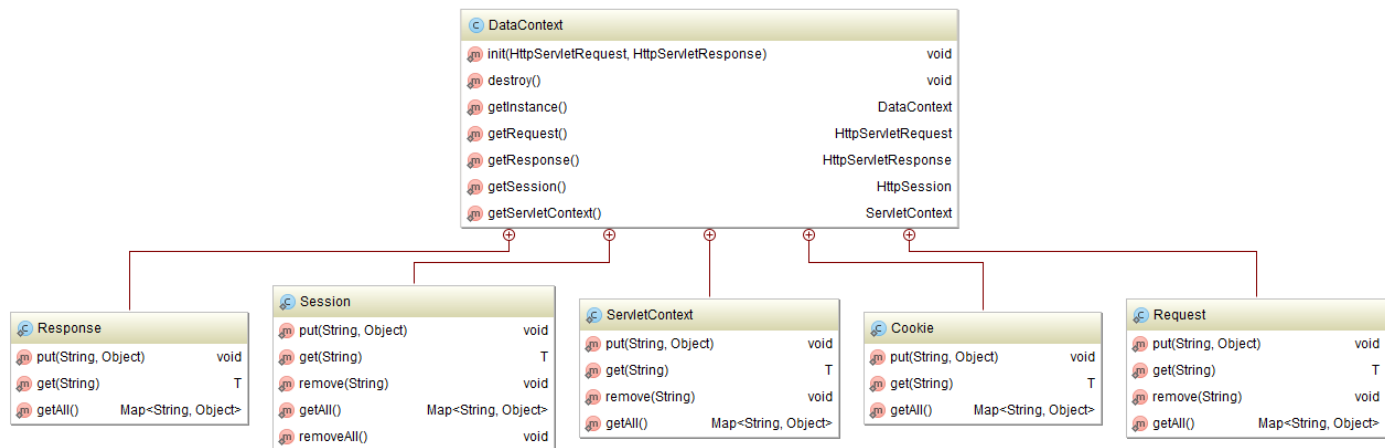
5.9 DataConext

5.9.1 封装 Servlet API

可使用 `com.smart.framework.DataContext` 获取 `Servlet` 相关对象，它们包括：`Request`、`Response`、`Session`、`Cookie`、`ServletContext` 等，也可通过 `DataContext` 操作这些对象的方法。有了 `DataContext`，开发者无需依赖于 `Servlet API` 就能完成同样事情。`DataContext` 内部使用了 `ThreadLocal`，保证了线程安全问题，让多个线程拥有各自的线程本地变量。

5.9.2 DataContext API

`DataContext` 的内部结构如下图所示：



示例：获取 Request 中的数据

```
String value = DataContxt.Request.get("key");
```

示例：将数据放入 Session 中

```
DataContxt.Session.put("key", "value");
```

5.9.3 参考资料

- ✓ 封装 Servlet API: <http://my.oschina.net/huangyong/blog/162773>
- ✓ ThreadLocal 那点事儿: <http://my.oschina.net/huangyong/blog/159489>
- ✓ ThreadLocal 那点事儿（续集）: <http://my.oschina.net/huangyong/blog/159725>

6 相关插件

6.1 Cache

6.1.1 源码地址

<http://git.oschina.net/huangyong/smart-plugin-cache>

6.1.2 Maven 依赖

```
<dependency>
  <groupId>com.smart</groupId>
  <artifactId>smart-plugin-cache</artifactId>
  <version>${smart.version}</version>
</dependency>
```

可使用两种方式使用 Cache 插件：

1. Cache API
2. Cache 注解

6.1.3 Cache API

Cache	CacheManager
<code>get(K)</code> <code>V</code>	<code>getCaches()</code> <code>Iterable<Cache></code>
<code>put(K, V)</code> <code>void</code>	<code>createCache(String)</code> <code>Cache<K, V></code>
<code>put(K, V, long)</code> <code>void</code>	<code>getCache(String)</code> <code>Cache<K, V></code>
<code>remove(K)</code> <code>void</code>	<code>destroyCache(String)</code> <code>void</code>
<code>clear()</code> <code>void</code>	
<code>getDurations() Map<K, Duration></code>	

示例：使用 Cache API 实现缓存控制

```
@Service
public class ProductServiceImpl implements ProductService {

    private Cache<Long, Product> productCache; // 定义一个 Cache

    public ProductServiceImpl() {
        // 创建 CacheManager 及其相关 Cache
        CacheManager cacheManager = new DefaultCacheManager();
        productCache = cacheManager.createCache("product");
    }

    @Override
```

```

public Product getProduct(long id) {
    Product product = productCache.get(id); // 先从 Cache 中获取数据
    if (product == null) {
        // 若数据不存在，则从数据库中获取数据
        product = DataSet.select(Product.class, "id = ?", id);
        // 将数据放入 Cache
        productCache.put(id, product);
    }
    return product;
}
}

```

6.1.4 Cache 注解

若想对方法级别进行缓存控制，则可使用 Cache 注解，将其定义在需要缓存控制的方法上。

相关 Cache 注解包括：

com.smart.plugin.cache.annotation.Cachable	定义在类上，表明该类可进行缓存控制
com.smart.plugin.cache.annotation.CachePut	定义在方法上，可将该方法的返回值放入 Cache
com.smart.plugin.cache.annotation.CacheClear	定义在方法上，执行完该方法后将自动清空 Cache

示例：使用 Cache 注解实现缓存控制

```

@Service
@Cachable
public class ProductServiceImpl implements ProductService {

    @Override
    @CachePut
    public Product getProduct(long id) {
        Product product = productCache.get(id); // 先从 Cache 中获取数据
        if (product == null) {
            // 若数据不存在，则从数据库中获取数据
            product = DataSet.select(Product.class, "id = ?", id);
            // 将数据放入 Cache
            productCache.put(id, product);
        }
        return product;
    }
}

```

6.1.5 参考资料

✓ Smart Plugin —— 从一个简单的 Cache 开始：<http://my.oschina.net/huangyong/blog/173260>

- ✓ 能否让 Cache 变得更加优雅? : <http://my.oschina.net/huangyong/blog/173946>
- ✓ Cache Plugin 实现过程: <http://my.oschina.net/huangyong/blog/174872>
- ✓ 一个简单的 Cache 淘汰策略: <http://my.oschina.net/huangyong/blog/177559>
- ✓ Lock 那点事儿: <http://my.oschina.net/huangyong/blog/172391>

6.2 WebService

6.2.1 源码地址

<http://git.oschina.net/huangyong/smart-plugin-ws>

6.2.2 Maven 依赖

```
<dependency>
    <groupId>com.smart</groupId>
    <artifactId>smart-plugin-ws</artifactId>
    <version>${smart.version}</version>
</dependency>
```

可使用 `com.smart.plugin.ws.WebService.WebService` 注解发布两种 `WebService`:

1. 发布 SOAP 服务
2. 发布 REST 服务

6.2.3 发布 SOAP 服务

需将 `@WebService` 注解定义在接口上, 指定 `WebService` 地址 (即 WSDL 的相对地址) 与 `WebService` 类型。

示例: 发布 SOAP 服务

```
@WebService(value = "/soap/ProductService", type = WebService.Type.SOAP)
public interface ProductService {

    List<Product> getProductList();
}
```

6.2.4 发布 REST 服务

需将 `@WebService` 注解定义在类上, 指定 `WebService` 地址 (即 WADL 的相对地址) 与 `WebService` 类型。

此外, 还需配合使用 `JAX-RS` 注解。

示例: 发布 REST 服务

```
@Service
@WebService(value = "/rest/ProductService", type = WebService.Type.REST)
```

```
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class ProductService {

    @GET
    @Path("/products")
    public List<Product> getProductList() {
        return DataSet.selectList(Product.class, "", "id asc");
    }
}
```

6.2.5 创建 Webservice 客户端

通过代理方式创建 Webservice 客户端：

- ✓ 可使用 SOAPHelper 的 createClient 方法创建 SOAP 客户端。
- ✓ 可使用 RESTHelper 的 createClient 方法创建 REST 客户端。

6.2.6 参考资料

- ✓ 发布与调用 Web 服务还能再简化吗？：<http://my.oschina.net/huangyong/blog/178192>
- ✓ 初步实现 Webservice 插件：<http://my.oschina.net/huangyong/blog/178331>

6.3 Mail

6.3.1 源码地址

<http://git.oschina.net/huangyong/smart-plugin-mail>

6.3.2 Maven 依赖

```
<dependency>
    <groupId>com.smart</groupId>
    <artifactId>smart-plugin-mail</artifactId>
    <version>${smart.version}</version>
</dependency>
```

6.3.3 邮件配置

可在 config.properties 文件中配置邮件服务器相关信息，配置如下：

假设对 foo@163.com 邮箱发送并收取邮件，需在 config.properties 文件中做如下配置：

```
# 调试开关
```

```
mail.is_debug=false

# 发送邮件
mail.sender.protocol=smtp
mail.sender.protocol.ssl=true
mail.sender.protocol.host=smtp.163.com
mail.sender.protocol.port=465
mail.sender.from=foo<foo@163.com>
mail.sender.auth=true
mail.sender.auth.username=foo@163.com
mail.sender.auth.password=xxx

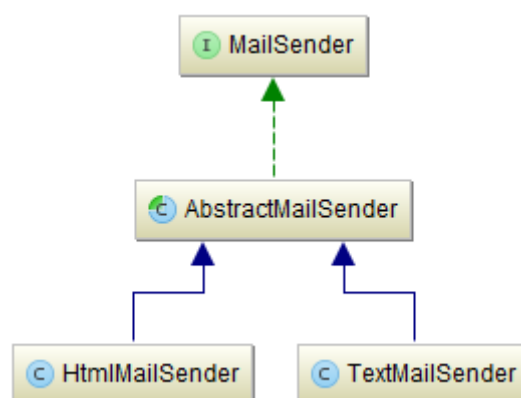
# 收取邮件
mail.fetcher.protocol=pop3
mail.fetcher.protocol.ssl=true
mail.fetcher.protocol.host=pop.163.com
mail.fetcher.protocol.port=995
mail.fetcher.folder=INBOX
mail.fetcher.folder.readonly=true
```

6.3.4 发送邮件

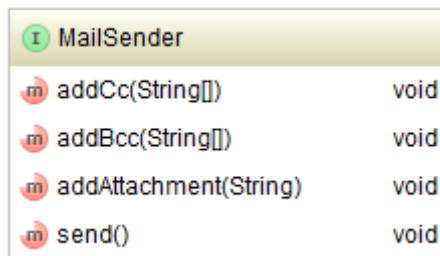
可使用 `com.smart.plugin.mail.send.MailSender` 接口发送邮件，根据不同情况选择相应的实现：

1. 若发送 HTML 格式的邮件，则需使用 `com.smart.plugin.mail.send.impl.HtmlMailSender` 实现类。
2. 若发送纯文本格式的邮件，则需使用 `com.smart.plugin.mail.send.impl.TextMailSender` 实现类。

类图层次结构如下：



MailSender API 如下：

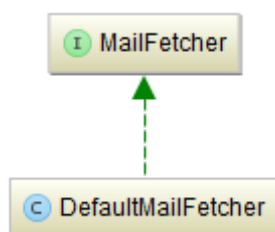


6.3.5 收取邮件

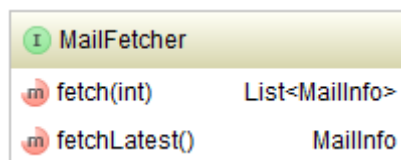
可使用 `com.smart.plugin.mail.fetch.MailFetcher` 接口收取邮件，默认只有一种实现：

`com.smart.plugin.mail.fetch.impl.DefaultMailFetcher`

类图层次结构如下：



MailFetcher API 如下：



6.3.6 参考资料

- ✓ 初步实现 Mail 插件 —— 发送邮件：<http://my.oschina.net/huangyong/blog/178661>
- ✓ 初步实现 Mail 插件 —— 收取邮件：<http://my.oschina.net/huangyong/blog/178915>

6.4 I18N

6.4.1 源码地址

<http://git.oschina.net/huangyong/smart-plugin-i18n>

6.4.2 Maven 依赖

```
<dependency>
```



```
<groupId>com.smart</groupId>
<artifactId>smart-plugin-i18n</artifactId>
<version>${smart.version}</version>
</dependency>
```

6.4.3 I18N 语言包

可在 src/main/resource/i18n 目录下添加 I18N 语言包（properties 文件）。

示例：分别定义英文与中文语言包

i18n_en_US.properties（英文语言包）	i18n_zh_CN.properties（中文语言包）
common.title=Smart Sample common.copyright=Copyright © 2013 ...	common.title=Smart 示例 common.copyright=版权所有 © 2013 ...

中文语言包需使用 ascii2native 编码，以上示例中是编码前的内容。

6.4.4 I18N 配置

可在 config.properties 文件中配置 I18N 语言包是否可重新加载，配置如下：

```
i18n.reloadable=true
```

以上配置项会降低一定的性能，在生产环境中，建议将以上配置项改为 false。

6.4.5 在 JSP 中使用 I18N

在 JSP 中可使用 JSTL 的 fmt 标签实现 I18N 功能，使用方法如下：

1. 引入 fmt 标签库

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

2. 定义默认语言包

```
<f:setBundle basename="i18n.i18n_${system_language}"/>
```

3. 在 JSP 中使用

```
<f:message key="foo.bar"/>
```

6.4.6 在 JS 中使用 I18N

在 JS 中可使用 JS 函数实现 I18N 功能，使用方法如下：

1. 引入 JS 语言包脚本

```
<script type="text/javascript" src="${BASE}/www/i18n/i18n_${system_language}.js"></script>
```

I18N 插件将自动根据 system_language 生成 JS 语言包脚本。

2. 提供一个 Smart.i18n 函数

```
var Smart = {
  i18n: function() {
    var args = arguments;
    var code = args[0];
    var text = window['I18N'][code];
    if (text) {
      if (args.length > 0) {
        text = text.replace(/\{(\d+)\}/g, function(m, i) {
          return args[parseInt(i) + 1];
        });
      }
      return text;
    } else {
      return code;
    }
  }
}
```

3. 在 JS 中使用

```
Smart.i18n('foo.bar')
```

6.4.7 参考资料

✓ 初步实现 I18N 插件: <http://my.oschina.net/huangyong/blog/179171>

6.5 Job

6.5.1 源码地址

<http://git.oschina.net/huangyong/smart-plugin-job>

6.5.2 Maven 依赖

```
<dependency>
  <groupId>com.smart</groupId>
  <artifactId>smart-plugin-job</artifactId>
  <version>${smart.version}</version>
</dependency>
```

6.5.3 自动启动 Job

若想在应用启动时，自动启动某个 Job，则可使用 `com.smart.plugin.job.Job` 注解定义在 Job 类上，该类还需继承 `com.smart.plugin.job.BaseJob` 父类，并覆盖父类的 `execute` 方法。可在 Job 类中使用 `@Inject` 注解实现依赖注入。

示例：每隔 1 秒钟输出当前时间与 Hello Smart! 信息

```
@Bean
@Job("0/1 * * * * ?")
public class SmartHelloJob extends BaseJob {









    private static final SimpleDateFormat format = new SimpleDateFormat("HH:mm:ss");

    @Override
    public void execute() {
        System.out.println(format.format(new Date()) + " - Hello Smart!");
    }
}
```

6.5.4 手工控制 Job

若想手工控制 Job 的启动、暂停、恢复、停止，则可使用 com.smart.plugin.job.JobHelper 类提供的 API 来实现。

JobHelper 的 API 如下：

JobHelper		
	startJob(Class<?>, String)	void
	startJobAll()	void
	stopJob(Class<?>)	void
	stopJobAll()	void
	pauseJob(Class<?>)	void
	resumeJob(Class<?>)	void
	createScheduler(Class<?>, String)	Scheduler
	getScheduler(Class<?>)	Scheduler

示例：定义一个 Job 类，该类不会随应用启动而自动开启

```
@Bean
public class SmartHelloJob extends BaseJob {

    private static final SimpleDateFormat format = new SimpleDateFormat("HH:mm:ss");

    @Override
    public void execute() {
        System.out.println(format.format(new Date()) + " - Hello Smart!");
    }
}
```

示例：通过一个单元测试，手工方式控制 Job 启动与停止

```

public class SmartJobTest extends BaseTest {

    @Test
    public void test() {
        JobHelper.startJob(SmartHelloJob.class, "0/1 * * * * ?");

        sleep(3000);

        JobHelper.stopJob(SmartHelloJob.class);

        sleep(3000);
    }

    private void sleep(long ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

6.5.5 参考资料

✓ 初步实现 Job 插件: <http://my.oschina.net/huangyong/blog/184620>

6.6 Hessian

6.6.1 源码地址

<http://git.oschina.net/huangyong/smart-plugin-hessian>

6.6.2 Maven 依赖

```

<dependency>
    <groupId>com.smart</groupId>
    <artifactId>smart-plugin-hessian</artifactId>
    <version>${smart.version}</version>
</dependency>

```

6.6.3 发布 Hessian 服务

可使用 `com.smart.plugin.hessian.Hessian` 注解发布 Hessian 服务, 只需将该注解定义在 Service 接口上。

示例: 将 Service 接口发布为 Hessian 服务

```
@Hessian("/user_service")
public interface UserService {

    User login(String username, String password);
}
```

6.6.4 创建 Hessian 客户端

可使用 `com.smart.plugin.hessian.HessianHelper` 类的 `createClient` 方法创建 Hessian 客户端。

6.6.5 参考资料

✓ 将 Hessian 集成到 Smart 中: <http://my.oschina.net/huangyong/blog/187561>

6.7 Template

6.7.1 源码地址




<http://git.oschina.net/huangyong/smart-plugin-template>

6.7.2 Maven 依赖

```
<dependency>
    <groupId>com.smart</groupId>
    <artifactId>smart-plugin-template</artifactId>
    <version>${smart.version}</version>
</dependency>
```

6.7.3 TemplateEngine API

可使用 `com.smart.plugin.template.TemplateEngine` 实现合并模板文件或字符串，主要包括以下几个方法：

TemplateEngine		
	<code>mergeTemplateFile(String, Map<String, Object>, String)</code>	<code>void</code>
	<code>mergeTemplateFile(String, Map<String, Object>)</code>	<code>String</code>
	<code>mergeTemplateString(String, Map<String, Object>)</code>	<code>String</code>

7 代码生成器

7.1 安装 Smart SDK

7.1.1 下载 Smart SDK

可下载 Smart SDK 源码，下载地址：<http://git.oschina.net/huangyong/smart-sdk>。

也可直接下载 Smart SDK 的压缩包，下载后自行解压即可。

Smart SDK 依赖于 Smart Framework 与 Smart Generator，可从以下地址下载 Smart Generator：

<http://git.oschina.net/huangyong/smart-generator>

7.1.2 设置环境变量

SMART_HOME = <Smart SDK 的根目录>

PATH = ...;%SMART_HOME%\bin;

7.1.3 验证安装是否成功

打开 cmd 窗口，输入以下命令：

smart

若出现以下信息，则说明 Smart SDK 安装成功：

```
Smart Commands
smart create-app           : Create App
smart create-entity <entity-name> : Create Entity
smart create-service <service-name> : Create Service
smart create-action <action-name> : Create Action
smart create-page <page-name> : Create Page
smart create-crud <crud-name> : Create CRUD
smart load-dict <dict-path> : Load Dict
smart run-test             : Run Test
smart run-app              : Run App
smart build-app            : Bulid App
```

7.2 使用 Smart SDK 命令

命令	说明
smart create-app	创建 Smart 应用，需通过交互式完成 Smart 应用的创建
smart create-entity <entity-name>	创建 Entity 类，需指定 Entity 名称，大小写忽略
smart create-service <service-name>	创建 Service 接口及其实现类，需指定 Service 名称，大小写忽略

smart create-action <action-name>	创建 Action 类，需指定 Action 名称，大小写忽略
smart create-page <page -name>	创建 Page 文件（HTML 文件），需指定 Page 名称，大小写忽略
smart create-crud <crud -name>	创建 CRUD 脚手架模板，需指定 CRUD 名称，大小写忽略
smart load-dict <dict-path>	加载数据字典，需指定文件路径，自动创建 Entity 与建表 SQL 语句
smart run-test	运行单元测试
smart run-app	运行应用程序，将开启内嵌的 Tomcat
smart build-app	应用打包，将应用打为 war 包

7.3 参考资料

- ✓ 对代码生成器的一点想法：<http://my.oschina.net/huangyong/blog/160937>
- ✓ 代码生成器实现过程：<http://my.oschina.net/huangyong/blog/162138>
- ✓ 再论代码生成器：<http://my.oschina.net/huangyong/blog/168218>
- ✓ 使用 Smart SDK 快速开发 Java Web 应用：<http://my.oschina.net/huangyong/blog/169572>