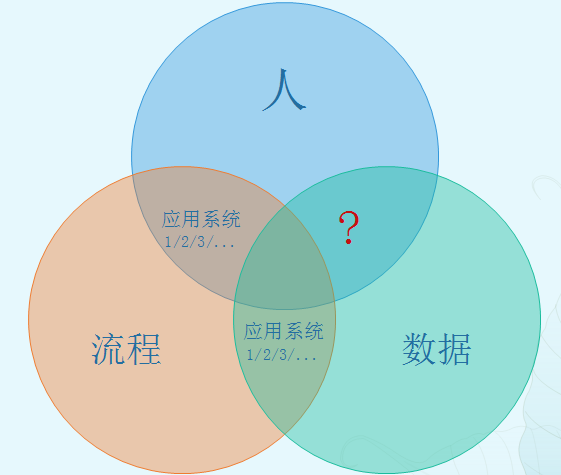
**它山石技术框架说明**

它山石（简称TSS）是一个开源的数据处理引擎，提供在线数据查询、数据分析、数据清洗、数据采集、对外数据服务、数据门户等功能，可全方位管理数据的生成、流转、分析、展示和分享。

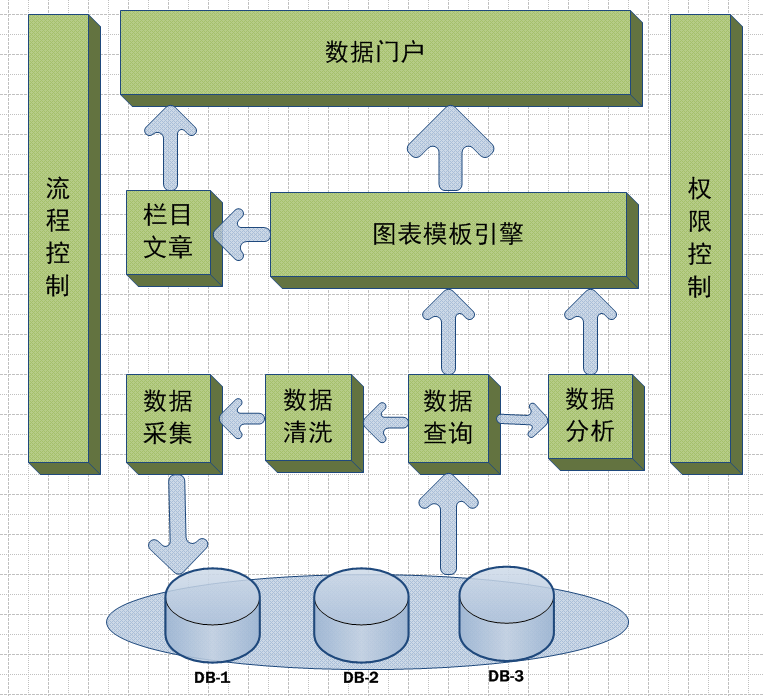
一个信息化的企业日常活动通常包含人、流程、数据这三个要素。人通过各类应用系统（ERP、CRM、各种行业软件等）发起流程，如销售流程、采购流程等，流程再通过应用系统对数据进行读写。数据随着时间不断沉淀增长，在现今的大数据时代正成为各个公司重要的无形资产。而各个应用系统通常只能分析由自身系统产生的数据，很难有哪个业务系统能全方位、综合性的分析企业的全盘数据。TSS的目的就是要打通人和数据之间的联通，降低挖掘数据价值的难度和成本，让数据可视化，让信息流动起来。正如搜索引擎之于Internet，它山石立志做结构化数据领域的数据处理引擎，挖掘无处不在的沉淀数据以焕发其价值。



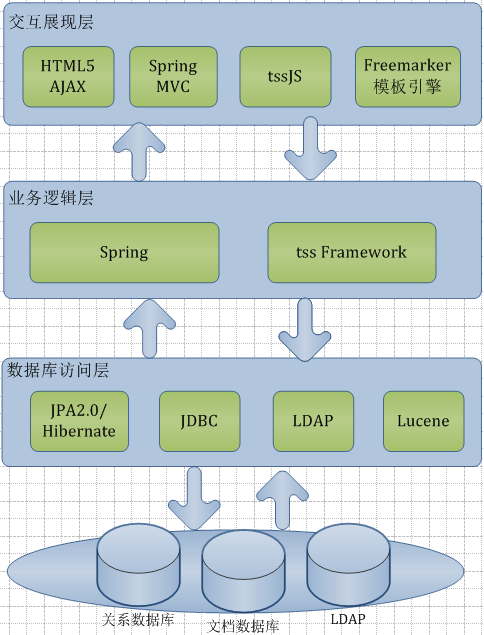
TSS同时也是一个应用基础平台，通过定制数据录入表单及控制流程，快速的开发业务流程。除此还提供专门的技术框架、功能组件及常见问题解决方案，基于此，能快速、高效、低成本搭建具体业务系统，开发者可以专注于业务逻辑的实现，无需为纷杂多变的各种软件技术而分心。降低开发门槛的同时，也减少了不必要的沟通环节，将通常的软件工程：**业务人员-->产品经理-->项目经理-->开发人员-->测试人员**，缩减为**业务人员-->开发人员**，从而最大限度的保证软件的概念完整性和一致性。

**它山石特色：**   
1、设计高度模块化，通用灵活，可任意组合复用   
2、流程、报表、门户、资源，直接在线开发、在线发布   
3、权限模型灵活，对业务对象进行严格的权限控制   
4、Portal聚合一切、提供统一入口（支持单点登录），实现一站式访问   
5、前沿的技术：HTML5 移动访问

**它山石产品架构**



**它山石技术架构**



主要开发语言为Java和JavaScript，后端框架底层采用JPA2.0/JDBC + Spring，前端为HTML5。

* 前后端交换一律采用AJAX，数据格式同时支持XML和JSON
* 前端使用tssJS框架，提供常用DOM操作、事件处理、AJAX交互、数据处理等核心封装，统一的界面样式布局，及菜单、Tree、Grid、Form、Layout、Panel、Message等常用组件
* Spring MVC将后端服务发布成restful风格的web服务
* Spring提供IOC容器及AOP机制，对业务对象的生成、依赖、事务、权限、日志、缓存等进行统一管理
* 使用Freemarker作为模板解析引擎，应用于门户、数据图表等支持模板化二次定制开发
* 使用Lucene作为文档化数据的搜索引擎
* 持久层采用JPA2.0，支持多数据源, 数据库支持MySQL、H2、Oracle等

**框架特色：**   
1、前后端框架为自主原创，精致、实用、高效、易扩展、易维护   
2、前后端充分解耦合，开发时前后独立进行、互不依赖

# 功能划分

|  |  |
| --- | --- |
| 模块 | 主要功能 |
| Framework | 工具包、单点登陆、日志管理、缓存管理、  系统参数、定时任务、异常机制、持久化层封装、  Web容器层封装 |
| UM（统一用户管理） | 用户组织结构管理、导入同步用户  应用资源信息管理  角色权限管理、权限转授 |
| DM （数据管理） | 数据查询、数据分析、数据清洗  数据采集、对外数据服务 |
| CMS（内容管理） | 站点栏目文章管理、全文检索 |
| Portal（门户管理） | 内容聚集和展示、个性化定制  应用集成、数据门户 |

# 常用功能点介绍

### HttpRequest编码过滤器

该过滤器拦截所有servlet请求，将请求的编码设置为配置的编码格式。配置如下：

|  |
| --- |
| @WebFilter(filterName = "EncodingFilter",  urlPatterns = {"/\*"} ,  initParams = {@WebInitParam(name="encoding", value="UTF-8")}  ) |

### HttpRequest上下文过滤器

* 上下文过滤器的作用是将request请求封装成RequsetContext对象设置到现场变量（Context. requestLocal）里，这样在其它的模块里每次需要取request里各类信息（header、cookie、parameters、session）时，只需要通过Context的类方法（即静态方法）即可获取到，而不用将类似登陆用户ID这种常用信息做为参数从控制层（Action）一直传递到业务逻辑层，甚至持久层。 上下文过滤器的配置如下：

|  |
| --- |
| @WebFilter(filterName = "ContextFilter",  urlPatterns = {"/\*"},  initParams = {@WebInitParam(name="ignorePaths", value="/remote/OnlineUserService,js,htm,html,jpg,png,gif,ico,css,xml,swf")}  ) |

### 异常机制

平台所有代码的异常定义都需要实现IBusinessException接口：

|  |
| --- |
| /\*\*  \* 是否需显示错误信息给用户看  public boolean isPopup();  /\*\*  \* 是否需重新登录系统:  \* 0-无需登录；  \* 1-需要重新登录平台；  \* 2-需要重新输入密码，登录其他系统  \*/  public int getRelogin(); |

平台的异常实现类有两个BusinessException 和 BusinessServletException。前者负责处理业务逻辑代码出现的异常，后者负责处理在控制层servlet中出现的异常。另外在单点登陆身份验证时还定义了一个特殊的异常UserIdentificationException，捕获该异常时，系统会自动将其转换为一个 要求 relogin 的BusinessServletException异常。

抛出的异常由CatchExceptionFilter，捕捉到异常后由ExceptionEncoder对象负责将异常输出（控制台、日志文件以及前台）。在ExceptionEncoder中，调用了消息机制中一个ErrorMessageEncoder组织异常输出格式（采用XML格式），另外ExceptionEncoder还允许应用开发者自定义异常转换器（通过实现IExceptionConvertor接口）来定制用户需要的信息内容（比如捕捉到一个数据库插入字段过长的异常信息，本身信息内容对用户来说不友善，难以读懂，这时可以用异常转换器将内容转换成用户能理解的字面意思再输出）。

*异常处理相关的配置如下有：*

|  |
| --- |
| @WebFilter(filterName = "CatchExceptionFilter", urlPatterns = {"/\*"}) |
| 异常转换器（application.properties）：  #Exception Convertor 允许设置多个转换器，以“,”格开  class.name.ExceptionConvertor= com.jinhe.tss.core.exception.convert.impl.SameNodeNameExceptionConvertor,com.jinhe.tss.core.exception.convert.impl.LongerNodeNameExceptionConvertor |

### 在线用户库和用户令牌（Token）机制

在线用户库和用户令牌都是用户单点登陆的重要组成部门。前者记录用户的登陆信息，后者则是表明用户身份的唯一凭证。

用户令牌Token由userId ＋ sessionId ＋ 当前时间 加密生成。在线用户库的工作原理是：

（1）当用户第一次成功登陆平台应用时，将会用登陆生成的Token针对当前应用在在线用户库中注册一条信息。

（2）当访问其它应用时，先检测携带的token是否已经在在线用户库中注册过，如果有，则说明用户已经在其它应用登陆过，同时也用Token为该应用也注册一条信息。

（3）当用户退出某个应用时，删除Token在这个应用的注册信息，同时检测是否还有其它应用在线，如果没有了，则从在线用户库移除本Token。

（4）当session过期时，操作和第三步一样。

在线用户库接口IOnlineUserManager定义了以下方法：

|  |
| --- |
| /\*\*  \* 根据token判断用户是否已经登录其他系统，如果登录则返回True，否则返回false  \*/  boolean isOnline(String token);  /\*\*  \* 注册用户登录当前系统  \* @param token 令牌  \* @param appCode 当前系统Code  \* @param sessionId 当前SessionID  \* @param userId 当前系统用户ID  \*/  void register(String token, String appCode, String sessionId, Long userId);  /\*\*  \* 删除在线用户或访问应用：根据应用Code，SessionId删除相应的记录。  \* SessionDestroyedListener监听器会在session超时时自动销毁在线用户信息。  \* @param appCode  \* @param sessionId  \*/  String delete(String appCode, String sessionId);  /\*\*  \* 根据Token获取当前用户登录的所有系统的相关信息  \*/  Set getAllOnlineInfos4Token(String token); |

在线用户库目前的实现类有：缓存式在线用户库（CacheOnlineUserManager） 和 数据库式在线用户库（DBOnlineUserManager）。

TSS框架里：在线用户库配置在TSS应用里，在 TSS的application.properties里设置如下：

|  |
| --- |
| #Online user manager 在线用户库管理对象，这里指定为“缓存式管理”  class.name.OnlineManager = com.jinhe.tss.core.online.cache.CacheOnlineUserManager |

同时在spring.xml里配置如下：

|  |
| --- |
| <!-- OnlineUserService Config -->  <bean id="online.OnlineUserService" class="com.jinhe.tss.core.online.remote.RemoteOnlineUserManagerService"/> |

再将此online.OnlineUserService 在remote-servlet.xml配置成远程接口（参见[远程调用实现机制](#_远程调用实现机制)）。

其它应用application.properties里还需要如下配置（spring.xml的配置参见[远程调用实现机制](#_远程调用实现机制)）：

|  |
| --- |
| #Online user manager 在线用户库管理对象，这里指定为“远程式管理”  class.name.OnlineManager = com.jinhe.tss.core.online.cache.RemoteOnlineUserManager |

这样其它应用就可以通过远程调用来操作UMS中的“在线用户库”。

### 单点登陆机制

|  |
| --- |
| SSO实现原理：  SSO通常都是应用于WEB系统中。基于Web的身份验证的实质无外乎就是有一个表单，表单里面让用户输入用户名称和密码，然后提交给验证的页面，通过身份验证后，通过Session来储存用户的一些信息，然后每次访问页面时，从session里面读这些信息，如果存在，则进入登录后的界面，否则，就认为没有登录。客户端要支持Cookie来存储sessionid，与服务器上创建的session来对应。  SSO都要有一个单一的登录点，由此登录点将创建的会话token传递给应用系统。SSO需要建立一个统一的认证，权限信息库。但现实中，有的系统无法使用外部的认证，授权信息库，所以就需要在应用系统和Portal Server之间进行认证，同时进行授权信息的数据同步。根据认证，授权实现的位置可以分为两种实现方式：  第一类是通过Agent的方式，即在后端为每个Web应用系统都安装一个Agent，由Agent来接管该系统的身份验证和访问控制，Portal中会存放自己的用户信息，以及这些用户与其他系统的用户对应信息。这些Agent能够通过配置，轻松的接管了后面的系统的身份验证和访问控制。  第二类是通过Proxy的方式，即具有一个Proxy Server，由它来接管对于后端系统的访问，提交请求和读取数据，然后再返回给Portal。同时Portal可以存放用户信息以及用户的对应关系。Proxy Server会通过存储的用户对应关系和用户名和密码，自动完成后端系统的登录，然后就象一个浏览器一样，提取数据，返回数据给后端系统。这样的话后台系统不用做任何修改，身份认证和访问控制仍然由各个系统自己管理。 |

单点登陆（SSO）是整个TSS平台框架的核心之一，负责实现各个基于平台框架的应用之间的相互跳转时的自动登陆。平台单点登陆通过令牌（Token）机制和在线用户库来实现。

AutoLoginFilter拦截所有的请求（除了特殊单独放行，如LoginServet（login.in），/remote/OnlineUserService,/remote/LoginService等），通过各种身份验证器来验证用户是否已经登陆，或者携带的信息是否能完成登陆。目前身份验证器有：

|  |  |
| --- | --- |
| 身份验证器名称 | 作用 |
| UMPasswordIdentifier | 用户名/密码身份验证器，负责第一次登陆时的身份验证 |
| OnlineUserIdentifier | 在线用户库身份验证器，负责验证已经一个应用，跳转到另外一个应用时的身份验证（需要Token） |
| AnonymousUserIdentifier | 匿名用户身份验证，如果不是上面两种验证（既不是第一次登陆、又没有Token，或者Token验证为伪造的），则采用匿名用户身份验证。匿名用户只要*RequestContext*().canAnonymous()就通过，负责抛出UserIdentificationException("系统要求身份认证，请重新登录！") |
| 其它 | 如果需要实现以UsbKey、LTPAToken等其它登陆方式，可以自定义身份验证器 |

如果是第一次登陆(包括匿名)或者是第一次应用跳转，则验证通过过还需要执行用户登陆后的自定义操作。自定义操作在application.perperties设置，名称为class.name.LoginCostomizer。自定义操作执行后整个登陆过程才告结束。需要配置如下：

|  |
| --- |
| @WebFilter(filterName = "AutoLoginFilter",  urlPatterns = {"/auth/\*", "\*.do", "\*.portal"},  initParams = {@WebInitParam(name="ignorePaths", value="login.in,logout.in")  }) |
| @WebServlet(name="LoginServlet", urlPatterns="/auth/login.do")  @WebServlet(name="LogoutServlet", urlPatterns="/logout.in") |
| application.properties里配置：  #登陆后自定义操作配置，本自定义将用户角色信息写入各应用的RoleUserMapping表中  class.name.LoginCostomizer = com.jinhe.tss.um.sso.FetchPermissionAfterLogin  #身份转换器，转换不同应用间的用户身份信息  class.name.IdentityGetter = com.jinhe.tss.um.sso.UMIdentityGetter |
| 前台页面登陆配置：  需要将login.html文件拷贝到各应用前台文件目录下 |
| Session的销毁处理监听：  @WebListener com.jinhe.tss.framework.web.listener.SessionDestroyedListener  注：通过本监听器，平台框架可以在用户session过期销毁的时候，自动将该用户从在线用户库中剔除，并销毁其登陆令牌Token。 |

单点登陆可分成以下几种情况：

登陆TSS：



单一应用自动登陆：



基于TSS开发的应用之间进行SSO：



### 请求代理转发机制（HttpProxyFilter）

请求代理转发大致原理是：**请求头或参数中获取当前请求设置的目标地址所在应用CODE值，如果此CODE值为当前应用CODE，则直接放行；否则根据此CODE值获取对应的应用地址，将访问地址重新改写后（如果是登陆用户则在Header中添加令牌），然后通过HttpClient开源工具包以HTTP方式请求新地址，将返回的结果设置到本次请求的响应。**

**如下图所示：**

请求转发需要配置的地方有：

|  |
| --- |
| appServer.xml：  <?xml version="1.0" encoding="UTF-8"?>  <servers>  <server code="TSS" framework="tss" name="TSS" sessionIdName="JSESSIONID" baseURL="http://localhost:8088/tss"/>  <server code="CWP" framework="tss" name="CWP" sessionIdName="JSESSIONID" baseURL="http://localhost:8088/cwp"/>  </servers> |
| application.perperties：  #Current appliction code  application.code = TSS |

### XMLHttp请求解码过滤器（XmlHttpDecodeFilter）

由于平台框架下大部分请求都是按XMLHttp的形式发送，而提交的参数以及表单数据都是按XML格式拼装好然后放在请求流中发送。XMLHttp请求解码过滤器的作用正是拦截到这些请求，并解析请求中数据流，将它们逐个加入Request的parameter参数池中，从而保证这些值能正确的赋值到Action等定义的属性中。

解析工作主要有com.jinhe.tss.core.parser.http.XmlHttpDncoder对象完成，对应还有一个请求结束把返回结果重新拼装成XML格式的com.jinhe.tss.core.parser.http.XmlHttpEncoder类（详细可参考PTActionSupport类中XmlHttpEncoder的使用）。

### 消息机制 （IMessageEncoder接口）

平台框架里定义了一个IMessageEncoder接口，所以涉及往前台反馈消息的类都需要实现该接口。目前消息机制的信息编码类有：

|  |  |
| --- | --- |
| 类名 | 作用 |
| SuccessMessageEncoder | 负责处理成功信息的编码 |
| ErrorMessageEncoder | 负责处理错误异常信息的编码 |
| XmlHttpEncoder | 负责处理普通返回数据的编码 |

### 远程调用实现机制

通过扩展spring的远程调用（httpInvoke类型：详细可参考spring手册相关章节，其它远程调用类型还有RMI、Hessian等），实现了对远程接口调用的单点登陆控制(即带身份认证的远程调用)。

具体配置如下：

|  |
| --- |
| 服务端Web.xml:  <servlet>  <servlet-name>remote</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>remote</servlet-name>  <url-pattern>/remote/\*</url-pattern>  </servlet-mapping> |
| 服务端spring.xml:  <!-- OnlineUserService Config -->  <bean id="online.OnlineUserService" class="com.jinhe.tss.core.online.remote.RemoteOnlineUserManagerService"/> |
| 服务端WEB-INF/remote-servlet.xml：  <!-- Remote 在线用户管理 -->  <bean name="/OnlineUserService" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">  <property name="service"><ref bean="online.OnlineUserService"/></property>  <propertyname="serviceInterface">  <value>com.jinhe.tss.core.online.IOnlineUserManager</value>  </property>  </bean> |
| 客户端spring.xml:  <!-- Remote Online Service -->  <bean id="RemoteOnlineUserManager"  class="**com.jinhe.tss.core.spring.remoting.HttpInvokerProxyFactory**">  <property name="serviceUrl">  <value>/remote/OnlineUserService</value>  </property>  <property name="appCode">  <value>TSS</value>  </property>  <property name="serviceInterface">  <value>com.jinhe.tss.core.online.IOnlineUserManager</value>  </property>  </bean> |

HttpInvokerProxyFactory 继承了Spring的 HttpInvokerProxyFactoryBean类，增加了两个属性serviceUrl和appCode，这样在客户端配置远程接口时，只需将appCode（如：UMS）以及serviceUrl（如：/remote/OnlineUserService）配置进来进来。如此则可以根据appCode值直接取到该应用的BaseURL（如：http://10.100.1.5/tss）值，然后BaseURL + serviceUrl就是远程接口真实地址。从而避免每次配置远程服务都需要配置其绝对地址，方便了应用的发布过程。HttpInvokerProxyFactory调用了一个执行类AutoLoginHttpInvokerRequestExecutor，该类的主要负责转发和接收远程调用请求和响应中的cookie信息，比如Token Cookie，从而保证在远程调用时能完成单点登陆并对操作和数据可以进行权限过滤。

### 工具类使用说明

通过平台项目多年的开发积累，平台框架现已收集了一批久经考验的工具类方法，大致有：

|  |  |  |
| --- | --- | --- |
| **包名** | **工具类名** | **主要功能** |
| com.jinhe.tss.core.util | BeanUtil |  |
| DateUtil |  |
| DocUtil |  |
| URLUtil |  |
| Encrypt |  |
| FileHelper |  |
| StringUtil |  |
| MathUtil |  |
|  | Grid |  |
| Tree |  |
| XForm |  |

### 部分模块设计简介

|  |  |
| --- | --- |
| **名称** | **说明** |
| **单点登陆机制** | 单点登录对应用集成起到了关键的作用，平台的单点登录机制提供了一个坚实的底部实现的同时也充分考虑了可扩展性，既能非常方便的基于平台框架进行应用开发，又能快速的将第三方应用集成到平台中来。 |
| **AOP机制** | AOP（面向切面编程）的引进使得一些公用模块像权限、缓存、日志、事务等跟应用主业务代码完全解耦开来，同时又只需通过配置文件简单的配置就可以将这些模块功能集成到应用中来，实现无缝集成。无论在开发还是维护上都非常易于操作。 |
| **缓存机制** | 平台的缓存池设计具有很强的可扩展性，除了普通缓存池以外，目前扩展出来的有线程池、数据库连接池、自定义门户池、模板池等各种类型的缓存池实现。另外缓存模块还配备了完善的管理界面，通过管理界面，开发人员可以方便的修改缓存池的各个配置以改善性能；同时还可以对缓存池中的元素进行监视，根据缓存池给出的统计数据进行相应的优化改进。 |
| **日志模块** | 日志可以记下用户的操作信息，从而对一些意外进行回放，对相应人员进行问责；日志也可以对一些访问信息进行统计，从而获取有用信息。平台的日志模块特色在于其灵活性，开发人员不需要在代码中写大量的日志输出语句，而只需在配置文件中对主要的业务类的方法进行配置，通过AOP机制，日志模块就可以记下该方法的调用信息（包括操作人、操作时间、ip、操作参数、操作表、方法执行时间等）。 |
| **参数模块** | 参数模块为平台的应用提供了一个配置参数的地方，参数可以实时进行修改刷新。除了简单的参数外，参数模块还支持列表型（比如下拉框里选项配置）和树型（比如组织结构）参数的配置。 |
| **decode机制** | Decode的引进是为了解决树形结构数据的查询问题，类似用户组织结构，层次结构各异，深度不一，要查找某个枝节点下的数据如果采用递归方式查询的话性能会受到很大影响，同时如果数据有误的话，递归很容易进入死循环。采用Decode机制，在每个节点被创建的时候为其维护一个decode值（生成规则是：取其父节点decode值 + 其本身排序号）。如果查询某枝节点下数据时按该枝节点的decode值进行模糊查询（t.decode like ‘decode%’）即可。从子节点往上查询也是一样（‘decode%’ like t.decode）。 |

# 五、TSS框架优势

框架是一种现有的成功的解决方案，它存在的目标就是：通过提供标准的模式，来提升开发过程和现有组件直接的重复运用性，并为新的需求提供相应的扩展接口，从而提高开发效率和产品质量。

平台框架是由多个独立模块组成，正是这些独立模块为应用程序提供了一个可重用的设计。应用程序代码访问类库从而执行任务，而框架是调用应用程序代码，从而管理程序的流程。这就是经常说道的好莱坞原则：“不要试图联系我们，我们到时候自会通知你。”开发者写的程序在运行时由框架调用。

设计一个在各种未知背景下都可以使用的框架是很有挑战性的，TSS框架适合在复杂多变的J2EE开发中使用，它可以为开发者提供一个简单易用的模型。采用一个经过良好设计的开源框架有很多好处：

* 在好的框架下，开发者只需要写一些必须的代码；他们不需要直接接触底层的API。 这一点很重要。
* 经过良好设计的框架可以为程序提供清晰的结构并且提高程序的内聚性。清晰的结构使得其他人可以更容易加入项目。
* 一个容易使用的框架可以通过一些例子和文档为用户提供最佳实践。
* 采用成功的框架的代码比自己的代码容易测试

框架只有提供了一些值得使用的功能才会变得流行。