该文档以后不会再更新，最新版请移步：**https://hacpai.com/article/1466870492857**

Latke 快速上手指南

作者：B3log Team
版本：1.2.0.7, Jun 25, 2016

概述

Latke（土豆饼）是一个简单易用的 Java Web 应用开发框架，包含 IoC 容器、事件通知、持久化、插件等组件，也包含了一些应用开发时需要的基本服务（例如缓存、定时任务、邮件、HTTP 客户端等）。

在实体模型上使用 JSON 贯穿前后端，使应用开发更加快捷。这是 Latke 不同于其他框架的地方，非常适合小型应用的快速开发。

核心组件

模版引擎

使用 FreeMarker 作为模版引擎，细节参考 FreeMarker 文档。

IoC 容器

实现 JSR-330 规范，默认提供了控制器（@RequestProcessor）、服务（@Service）、DAO（@Repository）的构造型。

事件通知

通过事件管理器接口可进行事件监听器注册、事件发布，实现发布/订阅模式。

ORM

提供了对 JSON 对象的增删改查功能，可以支持关系型数据库（MySQL、H2、SQLServer）以及 Redis 的数据存取。

插件

插件包含完整的前端与后端功能，可以在不修改已有代码的前提下扩展应用功能，并支持运行时的拔插。

国际化

在模版中可以直接使用 ${xxxx} 的形式读取语言配置，后端提供了语言服务来获取不同 Locale 的语言配置。

服务

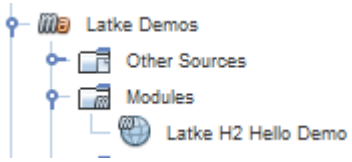
框架内置应用开发时需要的一些基本的常用服务：

- 缓存（CacheFactory，Cache）
- 图片处理（ImageServiceFactory，ImageService）
- 邮件（MailServiceFactory，MailService）
- HTTP 客户端（URLFetchServiceFactory，URLFetchService）
- 用户管理（UserServiceFactory，UserService）
- 日志（Logger）
- 任务队列（TaskQueueServiceFactory，TaskQueueService）
- 多语言（LangPropsService）

Hello World!

项目结构

请迁出官方示例：https://github.com/b3log/latke-demo，迁出后工程结构：



Latke Demos 是父项目，Lakte H2 Hello Demo 是支持 H2 数据库的 Latke 本地版（内嵌 jetty + 内嵌数据库 h2）的示例程序：

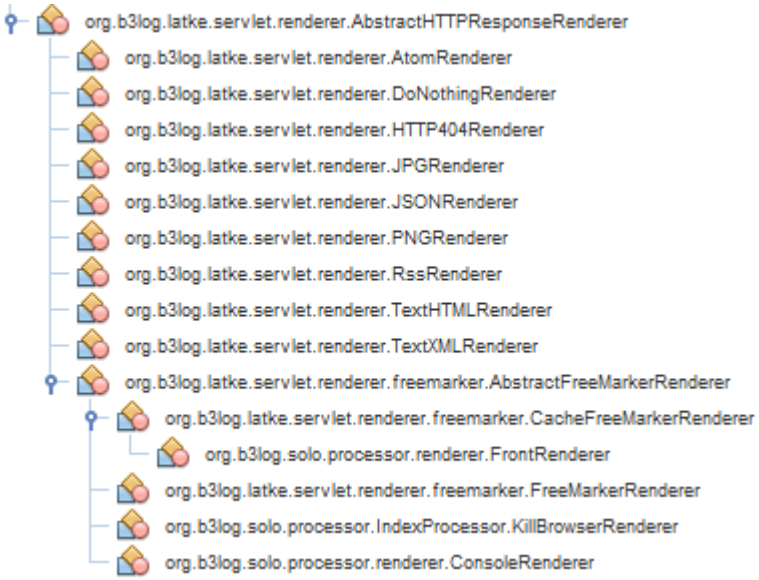
- 静态资源定义（src/main/webapp/WEB-INF/static-resources.xml），用于定义应用中用到的静态资源路径
- 框架通用配置（src/main/resources/latke.properties），定义了服务器访问信息、IoC 扫描包、运行环境、运行模式、部分服务实现（缓存服务、用户服务）、缓存容量、静态资源版本等
- 框架本地实现配置（src/main/resources/local.properties），定义了本地容器相关参数，例如数据库、JDBC 配置等
- 数据库表结构（src/main/resources/repository.json），定义了数据库表结构，用于生成建表语句以及持久化时的校验
- 部署描述符（src/main/webapp/WEB-INF/web.xml），定义了框架启动 Servlet 监听器、请求分发器等

请求处理

客户端的 HTTP 请求会经过 Latke 分发到应用定义的请求处理器上（标注有 @RequestProcessor 的类），一个请求处理器可以包含多个请求处理方法（标注有 @RequestProcessing 的方法），每个方法可以对应多个请求地址。可以看作是 SpringMVC 中控制器的简要实现，略有不同的是在响应的处理上。

```
@RequestProcessor
public final class HelloProcessor {

    private static final Logger LOGGER = Logger.getLogger(HelloProcessor.class.getName());

    @RequestProcessing(value = {"/", "/index", "/**/ant/*/path"}, method = HTTPRequestMethod.GET)
    public void index(final HTTPRequestContext context) {
        LOGGER.entering(HelloProcessor.class.getSimpleName(), "index");

        final AbstractFreeMarkerRenderer render = new FreeMarkerRenderer();
        context.setRenderer(render);

        render.setTemplateName("index.ftl");
        final Map<String, Object> dataModel = render.getDataModel();

        dataModel.put("greeting", "Hello, Latke!");

        LOGGER.exiting(HelloProcessor.class.getSimpleName(), "index");
    }
```

通过使用不同的响应渲染器可以生成不同类型的响应，例如 HTML、Rss、PNG 等：



服务调用

通过 @Inject 注入需要的服务：

```
/**
 * User service.
 */
@Inject
private UserService userService;

@RequestProcessing(value = "/register", method = {HTTPRequestMethod.GET, HTT
public void register(final HTTPRequestContext context, final HttpServletReq
    final AbstractFreeMarkerRenderer render = new FreeMarkerRenderer();
    context.setRenderer(render);

    render.setTemplateName("register.ftl");
    final Map<String, Object> dataModel = render.getDataModel();

    final String name = request.getParameter("name");
    if (!Strings.isEmptyOrNull(name)) {
        LOGGER.log(Level.FINER, "Name[{0}]", name);
        dataModel.put("name", name);

        userService.saveUser(name, 3);
    }
}
```

服务实现

- @Service 标注了该类是一个服务
- @Transactional 标注了该方法是执行在一个事务中。事务隔离为 READ_COMMITTED，传播类型为 REQUIRED

```java
@Service
public class UserService {

    private static final Logger LOGGER = Logger.getLogger(UserService.class.getName());

    @Inject
    private UserRepository userRepository;

    @Transactional
    public void saveUser(final String name, final int age) {
        final JSONObject user = new JSONObject();
        user.put("name", name);
        user.put("age", age);

        String userId;

        try {
            userId = userRepository.add(user);
        } catch (final RepositoryException e) {
            LOGGER.log(Level.ERROR, "Saves user failed", e);

            // Throws an exception to rollback transaction
            throw new IllegalStateException("Saves user failed");
        }

        LOGGER.log(Level.INFO, "Saves a user successfully [userId={0}]", userId);
    }
}
```

DAO

- @Repository 标注了该类是一个 DAO
- DAO 需要继承 AbstractRepository
- 在构造 DAO 时需要指定该 DAO 的名字，该名字对应 repository.json 中的描述

```java
@Repository
public final class UserRepository extends AbstractRepository {

    /**
     * Constructs a user repository.
     */
    public UserRepository() {
        super("user");

        // Generates database tables
        JdbcRepositories.initAllTables();
    }
}
```

repository.json

Source | History

```json
{
    "description": "Description of repository str
    "version": "1.0.0.0, Jul 2, 2013",
    "authors": ["Liang Ding"],
    "repositories": [
        {
            "name": "user",
            "keys": [
                {
                    "name": "oId",
                    "type": "String",
                    "length": 255
                },
                {
                    "name": "name",
                    "type": "String",
                    "length": 255
                },
                {
                    "name": "age",
                    "type": "int"
                }
            ]
        }
    ]
}
```

注意：初始化表 JdbcRepositories.initAllTables() 最好通过单独的初始化程序来做，调用这个方法后会根据 repository.json 的描述生成建表 SQL 并执行。

小结

- 框架使用非常类似 Spring
- 应用基于 JSON，更适合快速开发

限制

IoC

Bean 的生存周期（Scope）默认是单例（Singleton），其他生存周期（例如 Application、Request、Session、Dependent 等）目前 Latke 尚未实现。Latke 提倡的是服务端无状态设计，应用性能可以最优化并降低设计难度。当然，有状态的设计也是可以的，所以这些生存周期后续会逐渐提供支持。

最佳实践

表名前缀

在 local.properties 中有一项配置 jdbc.tablePrefix，如果配置了该项，则初始化表（JdbcRepositories.initAllTables()）时生成的表名就会带有前缀。

建议应用配置该项，以屏蔽不同数据库迁移数据时关键字对表名的影响。

实体模型

Lakte 使用 JSON 作为实体载体，管理 JSON 的键就是对实体的建模。实体的键对应了数据库表列名，实体内嵌的关联对象是服务中组装的。例如对于“用户”实体，键包含了简单类型属性：“name”、“age”，关联类型属性：”books”，构造的对象例如：

```
{
        “name”: “Daniel”,
        “age”: 23,
        “books”: [{
                “name”: “TAO of Life”
        }, ….]
}
```

键管理可以通过 User 类：

```
public class User {
   public static final String USER_NAME = “name”;
   public static final String USER_AGE = “age”;
   public static final String USER_T_BOOKS = “books”;
}
```

T 表示这个属性是非持久化的（User 表中无此列），是通过在服务中组装而来。

repository.json

这个文件可以手工编写，然后使用 JdbcRepositories#initAllTables 方法自动创建数据库；也可以使用 JdbcRepositories#initRepositoryJSON 方法从已有数据库表生成这个文件。

repository.json -> tables：

- 不用对不同数据库编写 SQLs
- 可以运行时建表
- 从业务逻辑实现开始的开发方式

tables -> repository.json：

- 在已有 tables 上继续开发
- 从 DB 开始的开发方式

这两种方式没有什么本质上的区别，可由开发自由决定。

关联查询

实体 JSON 对象中的关联属性是通过组装而来，需要先把这个属性查询出来，再编程组装到这个实体 JSON 对象中。这一点相对于一些 ORM 框架（例如 Hibernate）来说是比较繁琐，但这样做的优势之一就是能够使实体变得更灵活、更容易加入缓存优化性能。

也支持自定义 SELECT SQL，请参考接口 repository#select。