Latke 快速上手指南

作者：88250
原文链接：https://ld246.com/article/1466870492857
来源网站：链滴

该文档适合 Java Web 应用框架初学者以及 Latke 应用开发者，大家在看文档的同时，欢迎提出问题我们一起讨论如何解决，帮助框架演进。

概述

Latke（'l α:tk ，土豆饼）是一个简单易用的 Java Web 应用开发框架，包含 MVC、IoC/AOP、事通知、ORM、插件等组件。

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，非适合小型应用的快速开发，https://hacpai.com/article/1403847528022。

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：单例注入，默认提供了服务（@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
- 缓存：支持进程内存或 Redis，可通过配置切换
- 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

Hello World!

Demo 项目完整代码： https://github.com/88250/latke-demo

项目结构

```
└─src
    └─main
        ├─java
        │  └─latke
        │      └─demo
        │          │  Server.java
        │          │
        │          ├─processor
        │          │      HelloProcessor.java
        │          │      RegisterProcessor.java
        │          │      SimpleFMRenderer.java
        │          │
        │          ├─repository
        │          │      UserRepository.java
        │          │
        │          └─service
        │                  UserService.java
        │
```

```
└─resources
    │  latke.properties
    │  local.properties
    │  log4j.properties
    │  repository.json
    │  static-resources.xml
    │
    └─template
        │  hello.ftl
        │  index.ftl
        │  register.ftl
        │
        └─images
              logo.png
```

- 静态资源定义（src/main/resources/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、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，用于生成建表语以及持久化时的校验

启动服务器

```
public class Server extends BaseServer {

    public static void main(String[] args) {
        Latkes.setScanPath(Server.class.getPackage().getName());
        Latkes.init();
        // 初始化数据库表
        JdbcRepositories.initAllTables();

        final Server server = new Server();
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            server.shutdown();
            Latkes.shutdown();
        }));

        final BeanManager beanManager = BeanManager.getInstance();
        final HelloProcessor helloProcessor = beanManager.getReference(HelloProcessor.class);
        final RegisterProcessor registerProcessor = beanManager.getReference(RegisterProcesso
.class);

        // 配置路由
        final Dispatcher.RouterGroup routeGroup = Dispatcher.group();
        routeGroup.get("/", helloProcessor::index).
                get("/register", registerProcessor::showRegister).
                post("/register", registerProcessor::register).
                get("/var/{pathVar}", registerProcessor::paraPathVar).
                router().get().post().uri("/greeting").handler(helloProcessor::greeting);
        Dispatcher.mapping();
```

```
        server.start(8080);
    }
}
```

请求处理

```
@Singleton
public class HelloProcessor {

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

    public void index(final RequestContext context) {
        context.setRenderer(new SimpleFMRenderer("index.ftl"));

        final Map<String, Object> dataModel = context.getRenderer().getRenderDataModel();
        dataModel.put("greeting", "Hello, Latke!");

        Requests.log(context.getRequest(), Level.DEBUG, LOGGER);
    }

    public void greeting(final RequestContext context) {
        context.setRenderer(new SimpleFMRenderer("hello.ftl"));

        final Map<String, Object> dataModel = context.getRenderer().getRenderDataModel();
        dataModel.put("time", new Date());
        final String name = context.param("name");
        if (StringUtils.isNotBlank(name)) {
            dataModel.put("name", name);
        }
    }
}
```

另外，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

```
@Singleton
public class RegisterProcessor {

    @Inject
    private UserService userService;

    public void showRegister(final RequestContext context) {
        context.setRenderer(new SimpleFMRenderer("register.ftl"));
    }

    public void register(final RequestContext context) { // 函数式路由，在 Server 中配置
        context.setRenderer(new SimpleFMRenderer("register.ftl"));
        final Map<String, Object> dataModel = context.getRenderer().getRenderDataModel();

        final Request request = context.getRequest();
        final String name = request.getParameter("name");
```

```
        if (StringUtils.isNotBlank(name)) {
            dataModel.put("name", name);

            userService.saveUser(name, 3);
        }
    }

    public void paraPathVar(final RequestContext context) {
        final String paraVar = context.param("paraVar");
        final String pathVar = context.pathVar("pathVar");
        context.renderJSON(new JSONObject().put("paraVar", paraVar).put("pathVar", pathVar));
    }
}
```

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 REQUIRED

```
@Service
public class UserService {

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

    @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);

            // 抛出异常后框架将回滚事务
            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

```
@Repository
public class UserRepository extends AbstractRepository {

    public UserRepository() {
        super("user");
    }

    public JSONObject getByName(final String name) throws RepositoryException {
        return getFirst(new Query().setFilter(new PropertyFilter("name", FilterOperator.EQUAL, n
me)));
    }
}
```

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（JdbcRepositori
s.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 User 实体，键包含了简单类型属性：name、age，
联类型属性：books，构造的对象如：

```
{
    "name": "Daniel",
    "age": 23,
    "books": [{
        "name": "TAO of Life"
    }]
}
```

则键管理可以通过 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（Transient） 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的。

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 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
- 支持运行时建表
- 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 SON 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。