Weld 简明教程

宿宝臣 〈subaochen@126.com〉

山东理工大学

December 2, 2016

概述

编程模式的进化

- 面向过程
- 面向对象
- 面向组件
- 面向云计算

编程模式的进化

- 面向过程
- 面向对象
- 面向组件
- 面向云计算

编程模式的进化

- 面向过程
- 面向对象
- 面向组件
- 面向云计算

编程模式的进化

- 面向过程
- 面向对象
- 面向组件
- 面向云计算

补充：注解

可以参考：从 DI 到 CDI，重点是：

- 注解 handler 依赖于 java 的 reflection
- 参考：http://www.infoq.com/cn/articles/cf-java-annotation

DI：Dependency Injection

- DI：组件式编程的基石
- 心中永远有容器

CDI：Contextual Dependency Injection

- CDI: Contexts and Dependency Injection
  - 良好组织的组件生命周期管理：方便性
  - 类型安全的依赖注入：高可靠性
  - 基于事件通知机制的组件通讯：充分解耦
  - 优雅的组件拦截器（Interceptor/Decorator）：业务逻辑处理
  - 扩展容器功能的 SPI：可扩展性
- CDI 的核心思想：类型安全的松耦合
- CDI 是 JavaEE 的重要组成部分

松耦合

- 声明所依赖的组件的类型、状态；
- 无需操心所依赖的组件的生命周期、具体实现（面向接口的编程）；
- 所依赖组件的具体实现、生命周期甚至可配置、可替换（Alternatives）；
- 事件通知机制解耦了事件生产者和消费者；
- 拦截器允许横向解析事务逻辑，而不仅仅是按照时间线的竖向解析，比如?；
- 装饰器允许提炼事务逻辑处理中的共性并任意组合，比如装修房子；

类型安全

- 根据类型而不是名字的依赖注入
- 根据类型而不是名称的事件通知机制
- Interceptor/Decorator 的类型安全？
- 类型可以进行语法检查，而名字/名称仅仅是普通字符串

CDI 和 WELD 的关系

- CDI：JSR-346
- WELD：CDI 的参考实现（RI）
- 版本对照：CDI1.2<—>Weld 2.3.5

基本 weld 开发环境

- 下载 wildfly 10.1 from：http://www.wildfly.org，内置 weld 2.3.5
  - 设置环境变量：export JBOSS_HOME=$HOME/devel/wildfly
  - 启动 wildfly： $JBOSS_HOME/bin/standalone.sh
  - 停止 wildfly： $JBOSS_HOME/bin/jboss-cli.sh -connect :shutdown
- 下载 weld 2.3.4 from： weld 2.3.5 download
  - 解压缩到 $HOME/devel
  - 建立符号链接： ln -s weld-2.3.5.Final weld

Wildfly 进一步探索

- wildfly 的管理
- wildfly 的集群
- wildfly 的社区
- wildfly 的代码结构

先跑个小例子

- 启动 wildfly
- cd weld/examples/jsf/numberguess
- mvn clean package wildfly:deploy
- visit http://localhost:8080/weld-numberguess
- 观察 wildfly 的输出

近观 numberguess 的代码结构

- 有几种实现思路？
  - 纯的 JSF：ManagedBean:@ApplicationScoped numberGenerator
  - SpringMVC how to?
- 使用 netbeans 打开项目（netbeans 是建议的 Java EE IDE）
  - @ApplicationScoped Generator
  - @MaxNumber
  - @Random
- 使用 weld-probe 观察 beans

Wildfly/Tomcat/Jetty/Tomee/Jonas

- Servlet Container: Tomcat/Jetty
- JEE Container:Wildfly/Tomee/Jonas

一切 java 对象都是组件

- 组件的基本要求：定义无参构造方法的 Java 类
- 包括：
  - javabean
  - EJB session bean/Message Bean
  - JSF ManagedBean
- weld 在应用启动时自动扫描所有的 class 文件，符合条件的 java 对象都收入“对象库”

组件的依赖注入

- see: weld tutorial
- @Inject Greeting greeting 背后的故事
- @RequestScoped: web 应用的特殊性

组件的定位（寻址）

- 组件无处不在？如何区分（定位/寻址）组件？
- 组件的名字（字符串）不靠谱
- 组件的数据类型是终极解决方案
  - 数据类型是可语法检查的
  - CDI 类型安全的基础

组件的数据类型

- 每个组件可能存在多种数据类型
  Foo/Bar/FooBar1/FooBar2/Object
  增大了注入组件时分辨的难度：

```
public class Foo extends Bar implements FooBar1,FooBar2 {..
}
```

- @Typed(Foo.class) 明确限定组件的类型为 Foo，不常用的技巧
  - @Inject 的时候，组件类型不存在：unsatisfied dependency
  - @Inject 的时候，存在多个候选组件：ambiguous dependency

qualifier 的引入

- 帮助进一步澄清组件的类型：当依靠组件自身的类型无法唯一确定组件时，需要定义额外的 qualifier 一起联合限定组件：qualifiers **列表**

```
1 @Qualifier
2 @Target({TYPE,METHOD,FIELD,PARAMETER})
3 @Retention(RUNTIME)
4 public @interface Simple {
5 }
```

- see weld-tutorial/qualifiers
  - @Inject @Fancy Greeting greeting;

qualifier 的命名原则

qualifier 是通用的描述，能够复用为最佳设计，
即不要将 qualifier 绑定到特定的场景，比如：

- @Security 比 @SecurityInteceptor 好；
- @Updated 比 @UpdatedDocument 好；
- @Mock 比 @MockDeployment 好；
- @Asynchronous 比 @AyncPaymentProcessor 好；

@Default @Any

- see: @Default/@Any
- @Default：组件的默认 qualifer，如果没有定义其他 qualifier 的话；
- @Any：组件的必然 qualifier，即，任何组件都必然有一个 @Any Qualifier。
- 回顾组件的定位/寻址原理：
  - 定位依据：组件类型 +qualifiers 列表
  - 注入表达式的组件类型是组件定义时类型的子集；
  - 注入表达式的（@Inject）的 qualifiers 列表是组件的 qualifiers 列表的子集。

组件定位实例解析：只有唯一的实现时

- 组件定义：public class SimpleGreeter implements Greeter {...}
- 组件类型：Greeter/SimpleGreeter/Object
- 组件 Qualifier 列表：@Any,@Default
- 注入表达式分析：
  - @Inject Greeter;
  - @Inject SimpleGreeter; // 不好，违反了面向接口编程
  - @Inject @Default Greeter;
  - @Inject @Any Greeter;
  - @Inject @Any @Default Greeter;

组件定位实例解析：只有唯一的实现时

- 组件定义：public class SimpleGreeter implements Greeter {...}
- 组件类型：Greeter/SimpleGreeter/Object
- 组件 Qualifier 列表：@Any,@Default
- 注入表达式分析：
  - @Inject Greeter;
  - @Inject SimpleGreeter; // 不好，违反了面向接口编程
  - @Inject @Default Greeter;
  - @Inject @Any Greeter;
  - @Inject @Any @Default Greeter;

组件定位实例解析：只有唯一的实现时

- 组件定义：public class SimpleGreeter implements Greeter {...}
- 组件类型：Greeter/SimpleGreeter/Object
- 组件 Qualifier 列表：@Any,@Default
- 注入表达式分析：
  - @Inject Greeter;
  - @Inject SimpleGreeter; // 不好，违反了面向接口编程
  - @Inject @Default Greeter;
  - @Inject @Any Greeter;
  - @Inject @Any @Default Greeter;

组件定义实例解析：只有唯一的实现时

- 组件定义：public class SimpleGreeter implements Greeter {...}
- 组件类型：Greeter/SimpleGreeter/Object
- 组件 Qualifier 列表：@Any,@Default
- 注入表达式分析：
  - @Inject Greeter;
  - @Inject SimpleGreeter; // 不好，违反了面向接口编程
  - @Inject @Default Greeter;
  - @Inject @Any Greeter;
  - @Inject @Any @Default Greeter;

组件定义实例解析：只有唯一的实现时

- 组件定义：public class SimpleGreeter implements Greeter {...}
- 组件类型：Greeter/SimpleGreeter/Object
- 组件 Qualifier 列表：@Any,@Default
- 注入表达式分析：
  - @Inject Greeter;
  - @Inject SimpleGreeter; // 不好，违反了面向接口编程
  - @Inject @Default Greeter;
  - @Inject @Any Greeter;
  - @Inject @Any @Default Greeter;

组件定义实例解析：只有唯一的实现时

- 组件定义：public class SimpleGreeter implements Greeter {...}
- 组件类型：Greeter/SimpleGreeter/Object
- 组件 Qualifier 列表：@Any,@Default
- 注入表达式分析：
  - @Inject Greeter;
  - @Inject SimpleGreeter; // 不好，违反了面向接口编程
  - @Inject @Default Greeter;
  - @Inject @Any Greeter;
  - @Inject @Any @Default Greeter;

组件定义实例解析：只有唯一的实现时

- 组件定义：public class SimpleGreeter implements Greeter {...}
- 组件类型：Greeter/SimpleGreeter/Object
- 组件 Qualifier 列表：@Any,@Default
- 注入表达式分析：
  - @Inject Greeter;
  - @Inject SimpleGreeter; // 不好，违反了面向接口编程
  - @Inject @Default Greeter;
  - @Inject @Any Greeter;
  - @Inject @Any @Default Greeter;

组件定义实例解析：只有唯一的实现时

- 组件定义：public class SimpleGreeter implements Greeter {...}
- 组件类型：Greeter/SimpleGreeter/Object
- 组件 Qualifier 列表：@Any,@Default
- 注入表达式分析：
  - @Inject Greeter;
  - @Inject SimpleGreeter; // 不好，违反了面向接口编程
  - @Inject @Default Greeter;
  - @Inject @Any Greeter;
  - @Inject @Any @Default Greeter;

组件定位实例解析：只有唯一的实现时

- 组件定义：public class SimpleGreeter implements Greeter {...}
- 组件类型：Greeter/SimpleGreeter/Object
- 组件 Qualifier 列表：@Any,@Default
- 注入表达式分析：
  - @Inject Greeter;
  - @Inject SimpleGreeter; // 不好，违反了面向接口编程
  - @Inject @Default Greeter;
  - @Inject @Any Greeter;
  - @Inject @Any @Default Greeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter, Greeter, Object
  - FancyGreeter, Greeter, Object
- 组件的 Qualifier 列表
  - @Any, @Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter, Greeter, Object
  - FancyGreeter, Greeter, Object
- 组件的 Qualifier 列表
  - @Any, @Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter, Greeter, Object
  - FancyGreeter, Greeter, Object
- 组件的 Qualifier 列表
  - @Any, @Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件定位实例解析 2：存在多个实现时

- 组件定义
  - public class SimpleGreeter implements Greeter {}
  - @Fancy public class FancyGreeter implments Greeter {}
- 组件类型
  - SimpleGreeter,Greeter,Object
  - FancyGreeter,Greeter,Object
- 组件的 Qualifier 列表
  - @Any,@Default // SimpleGreeter
  - @Any @Fancy // FancyGreeter
- 注入表达式分析
  - @Inject Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Any @Default Greeter greeter; // 注入 SimpleGreeter
  - @Inject @Fancy Greeter greeter; // 注入 FancyGreeter
  - @Inject @Any Greeter greeter; // 错误，Ambigious 解析
  - @Inject @Any @Fancy Greeter greeter; // 注入 FancyGreeter;

组件的构成（成分）

- 一组（非空的）数据类型（本类、父类、接口）
- 一组（非空的）qualifiers（至少一个 @Any）
- 一个 scope
- 可选的 EL name
- 一组可选的拦截器绑定

再看 @Inject

- 属性上的 Inject
- 参数上的 Inject
- 构造方法上的 Inject

组件的生命周期

- 组件的生命周期是“自己”的事情，和 @Inject 没有关系，即和组件的使用者没有关系，组件的使用者也没有办法设置组件的生命周期：组件的使用者和组件是完全解耦的。
- 组件的默认生命周期：@Dependent，随使用者而生而死。
- 内置支持的生命周期：@SessionScoped,@ConversationScoped,@ApplicationScoped,@RequestScoped,@Dependent,@Singleton（不建议）
- 可以通过 SPI 扩展自定义的生命周期，参考 Deltaspike，其 ViewAccessScoped/WindowScoped 比较有特色，see also：deltaspike 的 scope

@Model

Web 开发中的常见组合

- @Named
- @RequestScoped

```
1 @Model
2 public class Blog {
3     private static final int
        PAGE_SIZE = 3;
4     ....
5 }
```

EL 名称

- @Named，主要用于在 JSF 页面中引用组件。
- @Named 默认采用组件的本类名作为名称。

Alternative

- 使用 @Alternative 声明组件是“**备胎**”，默认不启用，
  即在解析组件时屏蔽掉 @Alternative 的组件。
- 在 beans.xml 中明确启用一个 @Alternative 的组件。
- Question: 原来启用的组件是什么状态？
  当 beans.xml 明确声明了一个 Alternative 组件的时候，
  这个组件就成为符合组件解析条件的唯一了，其他符合解析条件的组件，
  无论了有多少个都没有关系。
- 什么场合下使用？
  特殊场景需要特殊组件时。
- 参见 weld-tutorial/alternative

Alternative

- 使用 @Alternative 声明组件是“**备胎**”，默认不启用，
  即在解析组件时屏蔽掉 @Alternative 的组件。
- 在 beans.xml 中明确启用一个 @Alternative 的组件。
- Question: 原来启用的组件是什么状态？
  当 beans.xml 明确声明了一个 Alternative 组件的时候，
  这个组件就成为符合组件解析条件的唯一了，其他符合解析条件的组件，
  无论了有多少个都没有关系。
- 什么场合下使用？
  特殊场景需要特殊组件时。
- 参见 weld-tutorial/alternative

Alternative

- 使用 @Alternative 声明组件是“**备胎**”，默认不启用，
  即在解析组件时屏蔽掉 @Alternative 的组件。
- 在 beans.xml 中明确启用一个 @Alternative 的组件。
- Question: 原来启用的组件是什么状态？
  当 beans.xml 明确声明了一个 Alternative 组件的时候，
  这个组件就成为符合组件解析条件的唯一了，其他符合解析条件的组件，
  无论了有多少个都没有关系。
- 什么场合下使用？
  特殊场景需要特殊组件时。
- 参见 weld-tutorial/alternative

Alternative

- 使用 @Alternative 声明组件是“**备胎**”，默认不启用，
  即在解析组件时屏蔽掉 @Alternative 的组件。
- 在 beans.xml 中明确启用一个 @Alternative 的组件。
- Question: 原来启用的组件是什么状态？
  当 beans.xml 明确声明了一个 Alternative 组件的时候，
  这个组件就成为符合组件解析条件的唯一了，其他符合解析条件的组件，
  无论了有多少个都没有关系。
- 什么场合下使用？
  特殊场景需要特殊组件时。
- 参见 weld-tutorial/alternative

Alternative

- 使用 @Alternative 声明组件是“**备胎**”，默认不启用，
  即在解析组件时屏蔽掉 @Alternative 的组件。
- 在 beans.xml 中明确启用一个 @Alternative 的组件。
- Question: 原来启用的组件是什么状态？
  当 beans.xml 明确声明了一个 Alternative 组件的时候，
  这个组件就成为符合组件解析条件的唯一了，其他符合解析条件的组件，
  无论了有多少个都没有关系。
- 什么场合下使用？
  特殊场景需要特殊组件时。
- 参见 weld-tutorial/alternative

松耦合的总结

- alternative 实现了部署时的组件解耦（多态）
- produces 方法实现了运行时的组件解耦（多态）
- 基于上下文的生命周期管理实现了组件的生命周期解耦：client 无需掌控组件的生命周期，由容器负责。
- 拦截器（Interceptor）：实现了事务逻辑的横向切片
- 装饰器（Decorator）：实现了事务逻辑的纵向切片
- 事件通知机制（Event Notification）：实现了事件产生和事件处理的彻底解耦

松耦合的总结

- alternative 实现了部署时的组件解耦（多态）
- produces 方法实现了运行时的组件解耦（多态）
- 基于上下文的生命周期管理实现了组件的生命周期解耦：client 无需掌控组件的生命周期，由容器负责。
- 拦截器（Interceptor）：实现了事务逻辑的横向切片
- 装饰器（Decorator）：实现了事务逻辑的纵向切片
- 事件通知机制（Event Notification）：实现了事件产生和事件处理的彻底解耦

松耦合的总结

- alternative 实现了部署时的组件解耦（多态）
- produces 方法实现了运行时的组件解耦（多态）
- 基于上下文的生命周期管理实现了组件的生命周期解耦：client 无需掌控组件的生命周期，由容器负责。
- 拦截器（Interceptor）：实现了事务逻辑的横向切片
- 装饰器（Decorator）：实现了事务逻辑的纵向切片
- 事件通知机制（Event Notification）：实现了事件产生和事件处理的彻底解耦

松耦合的总结

- alternative 实现了部署时的组件解耦（多态）
- produces 方法实现了运行时的组件解耦（多态）
- 基于上下文的生命周期管理实现了组件的生命周期解耦：client 无需掌控组件的生命周期，由容器负责。
- 拦截器（Interceptor）：实现了事务逻辑的横向切片
- 装饰器（Decorator）：实现了事务逻辑的纵向切片
- 事件通知机制（Event Notification）：实现了事件产生和事件处理的彻底解耦

松耦合的总结

- alternative 实现了部署时的组件解耦（多态）
- produces 方法实现了运行时的组件解耦（多态）
- 基于上下文的生命周期管理实现了组件的生命周期解耦：client 无需掌控组件的生命周期，由容器负责。
- 拦截器（Interceptor）：实现了事务逻辑的横向切片
- 装饰器（Decorator）：实现了事务逻辑的纵向切片
- 事件通知机制（Event Notification）：实现了事件产生和事件处理的彻底解耦

松耦合的总结

- alternative 实现了部署时的组件解耦（多态）
- produces 方法实现了运行时的组件解耦（多态）
- 基于上下文的生命周期管理实现了组件的生命周期解耦：client 无需掌控组件的生命周期，由容器负责。
- 拦截器（Interceptor）：实现了事务逻辑的横向切片
- 装饰器（Decorator）：实现了事务逻辑的纵向切片
- 事件通知机制（Event Notification）：实现了事件产生和事件处理的彻底解耦

组件的初始化和销毁

- @PostConstruct
- @PreDestroy

```
1  @Model
2  public class GreetingController {
3    @PostConstruct
4    public void init() {
5      System.out.println("
         GreetingController post
         construct......");
6      // do some init work
7    }
8
9    @PreDestroy
10   public void destroy() {
11     System.out.println("
         GreetingController pre
         destroy......");
12     // do some clean work
13   }
14 }
```

导入 Produces

- 解决了什么问题：@Inject 的组件不能由容器直接 new，需要运行时动态产生
- producer method 的 qualifier 列表和 @Inject 时的 qualifier 列表要对应起来。
- 例子：
  - randomNumber
  - EntityManager
  - Logger
  - Resources
- 松耦合的运行时解耦（多态）

导入 Produces

- 解决了什么问题：@Inject 的组件不能由容器直接 new，需要运行时动态产生
- producer method 的 qualifier 列表和 @Inject 时的 qualifier 列表要对应起来。
- 例子：
  - randomNumber
  - EntityManager
  - Logger
  - Resources
- 松耦合的运行时解耦（多态）

导入 Produces

- 解决了什么问题：@Inject 的组件不能由容器直接 new，需要运行时动态产生
- producer method 的 qualifier 列表和 @Inject 时的 qualifier 列表要对应起来。
- 例子：
  - randomNumber
  - EntityManager
  - Logger
  - Resources
- 松耦合的运行时解耦（多态）

导入 Produces

- 解决了什么问题：@Inject 的组件不能由容器直接 new，需要运行时动态产生
- producer method 的 qualifier 列表和 @Inject 时的 qualifier 列表要对应起来。
- 例子：
  - randomNumber
  - EntityManager
  - Logger
  - Resources
- 松耦合的运行时解耦（多态）

导入 Produces

- 解决了什么问题：@Inject 的组件不能由容器直接 new，需要运行时动态产生
- producer method 的 qualifier 列表和 @Inject 时的 qualifier 列表要对应起来。
- 例子：
  - randomNumber
  - EntityManager
  - Logger
  - Resources
- 松耦合的运行时解耦（多态）

导入 Produces

- 解决了什么问题：@Inject 的组件不能由容器直接 new，需要运行时动态产生
- producer method 的 qualifier 列表和 @Inject 时的 qualifier 列表要对应起来。
- 例子：
  - randomNumber
  - EntityManager
  - Logger
  - Resources
- 松耦合的运行时解耦（多态）

导入 Produces

- 解决了什么问题：@Inject 的组件不能由容器直接 new，需要运行时动态产生
- producer method 的 qualifier 列表和 @Inject 时的 qualifier 列表要对应起来。
- 例子：
  - randomNumber
  - EntityManager
  - Logger
  - Resources
- 松耦合的运行时解耦（多态）

导入 Produces

- 解决了什么问题：@Inject 的组件不能由容器直接 new，需要运行时动态产生
- producer method 的 qualifier 列表和 @Inject 时的 qualifier 列表要对应起来。
- 例子：
  - randomNumber
  - EntityManager
  - Logger
  - Resources
- 松耦合的运行时解耦（多态）

Producers 的生命周期

- Produces method 返回的对象默认是 @Dependent 的，可以通过设置其他的生命周期。
- Prouduces method 的生命周期决定了方法的调用频率，一定程度上起到了 Jboss Seam 中 @Factory 的作用。比如，@ApplicationScoped 的 Produces method。
- 包含 Produces method 的组件的生命周期和 Produces method 的生命周期没有多大关系。

Producers 的生命周期

- Produces method 返回的对象默认是 @Dependent 的，可以通过设置其他的生命周期。
- Prouduces method 的生命周期决定了方法的调用频率，一定程度上起到了 Jboss Seam 中 @Factory 的作用。比如，@ApplicationScoped 的 Produces method。
- 包含 Produces method 的组件的生命周期和 Produces method 的生命周期没有多大关系。

Producers 的生命周期

- Produces method 返回的对象默认是 @Dependent 的，可以通过设置其他的生命周期。
- Prouduces method 的生命周期决定了方法的调用频率，一定程度上起到了 Jboss Seam 中 @Factory 的作用。比如，@ApplicationScoped 的 Produces method。
- 包含 Produces method 的组件的生命周期和 Produces method 的生命周期没有多大关系。

Produces 实例：getTask

see: weld-tutorial#produces

```
1   @Produces
2   @Preferred
3   @SessionScoped
4   public Task getTask(AsyncTask asyncTask, SyncTask syncTask)
        {
5     System.out.println("getTask called......");
6     switch (taskType) {
7       case ASYNC:
8         return asyncTask;
9       case SYNC:
10        return syncTask;
11      default:
12        return null;
13    }
14  }
```

再说组件定位

组件拦截器

- 拦截器规范：
  http://docs.oracle.com/javaee/6/tutorial/doc/gkigq.html
- 任何组件均可使用拦截器：

```
public @SessionScoped @Transactional
class ShoppingCart implements Serializable { ... }
```

```
@InterceptorBinding
@Inherited
@Target( { TYPE, METHOD })
@Retention(RUNTIME)
public @interface Transactional {}
```

```
public @Transactional @Interceptor
class TransactionInterceptor { ... }
```

Figure: 拦截器示例

- 拦截器使用场合：
- see also: weld tutorial
- see also: CDI 中的拦截器

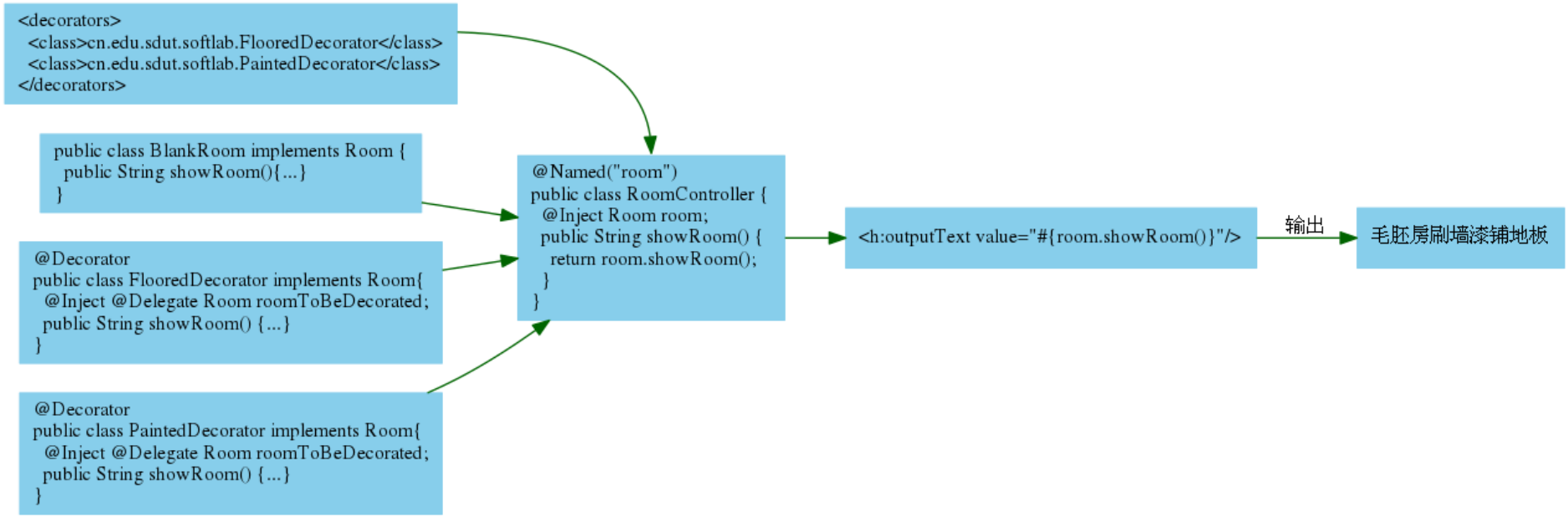
装饰组件

- 区别于 Interceptor，Decorator 干预事务逻辑，或者说，Decorator 是事务逻辑
  处理过程的一个步骤。
- 关于 decorator pattern 参见：浅析 decorator 模式，兼谈 CDI decorator 注解
- Java IO 中的 decorator pattern：

```
1 BufferedReader br = new BufferedReader(new
    InputStreamReader(System.in));
```

- 简单教程：weld-tutorial#decorator

一张图说明 Decorator



Interceptor 和 Decorator 的共同点

表面上看，客户端代码没有任何变化，因此 Interceptor 和 Decorator 都是绝佳的“地下工作者”：

- Interceptor：在合适的位置悄悄的记录对象的状态，但是不干预对象的流转，
  很像监听器（窃听器）。
- Decorator：悄悄的重写对象的方法，因此干预了对象的流转，很像？

Interceptor 和 Decorator 的共同点

表面上看，客户端代码没有任何变化，因此 Interceptor 和 Decorator 都是绝佳的“地下工作者”：

- Interceptor：在合适的位置悄悄的记录对象的状态，但是不干预对象的流转，
  很像监听器（窃听器）。
- Decorator：悄悄的重写对象的方法，因此干预了对象的流转，很像？

事件通知机制

简单、强大的事件通知机制，demo:
weld-tutorial#event

```
@Inject @Any Event<Room> event;
@Inject @CheckIn Event<Room> checkinEvent;
.....
event.select(new AnnotationLiteral<CheckOut>(){ }).fire(room)
checkInEvent.fire(room)
```

```
public void onRoomCheckOut(@Observes @CheckOut Room room)
public void onRoomCheckIn(@Observes @CheckIn Room room)
```

Vetoed

- see: http://docs.jboss.org/cdi/api/1.2/javax/enterprise/inject/Vet
- 排除组件和组件包，但是又不想删除时

stereotype

- qualifier 的模板技术，在逻辑上进一步组织（分组）qualifiers/interceptors。

```
1 @RequestScoped
2 @Named
3 @Security
4 @Stereotype
5 @Retention(RUNTIME)
6 @Target(TYPE)
7 public @interface Action{}
```

```
1  @Alternative
2  @Stereotype
3  @Retention(RUNTIME)
4  @Target(TYPE)
5  public @interface Mock{}
6  // active all Mock class:
7  <beans>
8    <alternatives>
9      <stereotype>org.mycompany
         .testing.Mock</
         stereotype>
10   </alternatives>
11 </beans>
```

why specialization?

Alternative 机制无法完全禁用相应的组件，因为：

- qualifier 列表可能不一定完全对应。
- 希望被禁用的组件可能有 producer method 或者 observer method，
  因此可能被容器激活从而失去了禁用的效果。

why specialization?

Alternative 机制无法完全禁用相应的组件，因为：

- qualifier 列表可能不一定完全对应。
- 希望被禁用的组件可能有 producer method 或者 observer method，
  因此可能被容器激活从而失去了禁用的效果。

如何防止 Alternative 禁用组件失败？

- 仔细检查被禁用组件是否有 producer method or observer method
- 仔细检查被禁用组件的 qualifier 列表和 @Alternative 组件是否一致
- 使用 @Specializes

SPI

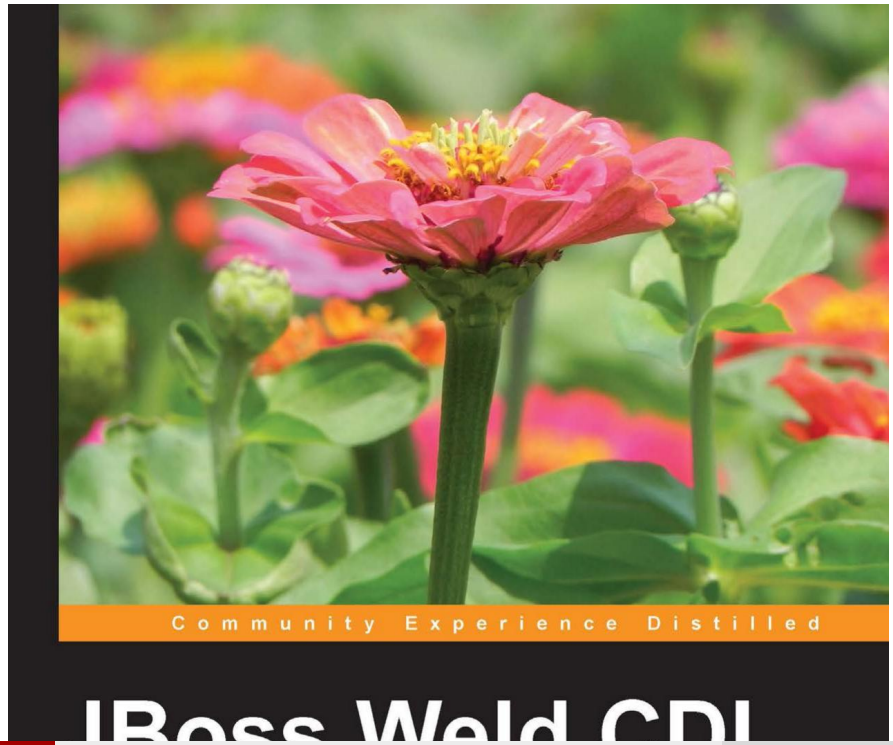
- SomeExtension extends Extension
- META-INF/services/javax.enterprise.inject.spi.Extension 添加 SomeExtension
- observer container lifecycle event
- BeanManager

参考资料

- CDI 1.2 规范
- weld home
- weld docs

进一步阅读

- Weld 自带的 examples
- Wildfly quickstart，较jboss eap quickstart的 bug 更少，推荐 wildfly quickstart
- 本人编写的几个weld tutorial小例子
- deltaspike
- JBoss Weld CDI for Java Platform



致谢

一起学习，共同进步！