Java 依赖注入标准（JSR-330）简介

转载请保留作者信息：

作者：88250 ，Vanessa

时间：2009 年 11 月 19 日

Java 依赖注入标准（JSR-330，Dependency Injection for Java）1.0 规范已于今年 10 月份发布 。该规范主要是面向依赖注入使用者，而对注入器实现、配置并未作详细要求。目前 Spring 、Guice 已经开始兼容该规范，JSR-299（Contexts and Dependency Injection for Java EE platform，参考实现 Weld ）在依赖注入上也使用该规范。JSR-330 规范并未按 JSR 惯例发布规范文档，只发布了规范 API 源码，本文翻译了该规范 API 文档（Javadoc）以作为对 Java 依赖注入标准规范的简介。

javax.inject

包 javax.inject 指定了获取对象的一种方法，该方法与构造器、工厂以及服务定位器（例如 JNDI））这些传统方法相比可以获得更好的可重用性、可测试性以及可维护性。此方法的处理过程就是大家熟知的依赖注入，它对于大多数应用是非常有价值的。

在我们的程序中，很多类型依赖于其他类型。例如，一个 Stopwatch 可能依赖于一个 TimeSource。一些类型被另一个类型依赖，我们就把这些类型叫做这个类型的依赖（物）。在运行时查找一个依赖实例的过程叫做解析 依赖 。如果找不到依赖的实例，那我们称该依赖为不能满足的 ，并导致应用运行失败。

在不使用依赖注入时，对象的依赖解析有几种方式。最常见的就是通过编写直接调用构造器的代码完成：

```
class Stopwatch {
final TimeSource timeSource;
Stopwatch () {
  timeSource = new AtomicClock(...);
}
void start() { ... }
long stop() { ... }
}
```

如果需要更有弹性一点，那么我们可以通过工厂或服务定位器实现：

```
class Stopwatch {
  final TimeSource timeSource;
  Stopwatch () {
    timeSource = DefaultTimeSource.getInstance();
  }
  void start() { ... }
  long stop() { ... }
}
```

在使用这些传统方式进行依赖解析时，程序员必须 做出适当权衡。构造器非常简洁，但却有一些限制（对象生存期，对象复用）。工厂确实解耦了客户与实现，但却需要样本式的代码。 服务定位器更进一步地解耦了客户与实现，但却降低了编译时的类型安全。并且，这三个方式都不适合进行单元测试。例如，当程序员使用工厂时，该工厂的 每一个 产品都必须模拟出来，测试完后还要得记得清理：

```
void testStopwatch() {
    TimeSource original = DefaultTimeSource.getInstance();
  DefaultTimeSource.setInstance(new MockTimeSource());
  try {
    // Now, we can actually test Stopwatch.
    Stopwatch sw = new Stopwatch();
    ...
  } finally {
    DefaultTimeSource.setInstance(original);
  }
}
```

现实中， 要模拟工厂将导致更多的样本式代码。测试模拟出的产品并清理它们在依赖多的情况下很快就控制不了了。更糟的是，程序员必须精确地预测 未来到底需要多少这样的弹性，并为他做的“弹性选择”负责。如果程序员开始时选择了构造器方式，但后来需要一个更有弹性的方式，那他就不得不替换所有调用构造器的代码。 如果程序员一开始过于谨慎地选择了工厂方式，结果可能导致要编写很多额外的样本式代码，引入了不必要的复杂度，潜在的问题比比皆是。

依赖注入 就是为了解决这些问题。代替程序员调用构造器或工厂，一个称作依赖注入器 的工具将把依赖传递给对象：

```
class Stopwatch {
  final TimeSource timeSource;
  @Inject Stopwatch(TimeSource timeSource) {
    this.TimeSource = timeSource;
  }
  void start() { ... }
  long stop() { ... }
}
```

注入器将更进一步地传递依赖给其他的依赖，直到 它 构造出整个 对象图 。 例如，假设一个程序员需要注入器创建一个 StopwatchWidget 实例：

```
/** GUI for a Stopwatch */
class StopwatchWidget {
  @Inject StopwatchWidget(Stopwatch sw) { ... }
  ...
}
```

注入器可能会：

1. 查找一个 TimeSource 实例
2. 使用找到的 TimeSource 实例构造一个 Stopwatch
3. 使用构造的 Stopwatch 实例构造一个 StopwatchWidget

这使得代码保持干净，使得程序员感到使用依赖（物）的基础设施非常容易。

现在，在单元测试中，程序员可以直接构造对象（不使用注入器）并将该对象以模拟依赖的方式直接传入待测对象的构造中。程序员再也不需要为每一次测试都配置并清理工厂或服务定位器。这大大简化了我们的单元测试：

```
void testStopwatch() {
  Stopwatch sw = new Stopwatch(new MockTimeSource());
  ...
}
```

完全降低了单元测试的复杂度，降低的复杂程度与待测对象的数目及其依赖成正比。

包 javax.inject 为使用这样的轻便类提供了依赖注入注解 ，但没有引入依赖配置方式。依赖配置方式取决于注入器的实现。程序员只需要标注了构造器、方法或字段来说明它们的可注入性（上面的例子就是构造器注入）。依赖注入器通过这些注解来识别一个类的依赖，并在运行时注入这些依赖。此外，注入器要能够在构建时 验证所有的依赖是否满足。相比之下，服务定位器在构建时是不能检测到依赖不满足情况的，直到运行时才能发现。

注入器实现有很多形式。一个注入器可以通过 XML、注解、DSL（领域规约语言），或是普通 Java 代码来进行配置。注入器实现可以使用反射或代码生成。使用编译时代码生成的注入器甚至可能没有它自己的运行时描述。而其他注入器实现无论在编译时还是运行时可能都不使用代码生成。一个“容器”，其实可以把它定义为一个注入器，不过包 javax.inject 不涉及非常大概念，旨在最小化注入器实现的限制。

请查阅：

@Inject

@Inject

注解 @Inject 标识了可注入的构造器、方法或字段。可以用于静态或实例成员。一个可注入的成员可以被任何访问修饰符（private、package- private、protected、public）修饰。注入顺序为构造器，字段，最后是方法。超类的字段、方法将优先于子类的字段、方法被注入。对于同一个类的字段是不区分注入顺序的，同一个类的方法亦同。

可注入的构造器指的是标注了 @Inject 并接受 0 个或多个依赖作为实参的构造器。对于每一个类而言，@Inject 最多只允许对一个类的一个构造器进行标注：

```
@Inject
ConstructorModifiers_opt SimpleTypeName(FormalParameterList_opt ) Throws_opt
        ConstructorBody
```

@Inject 对于仅存在默认构造器（访问修饰符为 public 并且无参数）的情况是可选的，注入器将调用默认构造器：

```
@Inject_opt
 Annotations_opt
 public SimpleTypeName() Throws_opt ConstructorBody
```

可注入的字段：

- 被 @Inject 标注。
- 不是 final 的。
- 可以使用任何有效名。

```
@Inject FieldModifiers_opt Type VariableDeclarators;
```

可注入的方法：

- 被 @Inject 标注。
- 不是 abstract 的。
- 没有声明类型参数的方法。
- 可以带返回值。
- 可以使用任何有效名。
- 接受 0 个或多个依赖作为实参。

```
@Inject MethodModifiers_opt ResultType
Identifier(FormalParameterList_opt )
       Throws_opt MethodBody
```

注入器忽略了注入方法的返回值，因为方法的非空返回可能会用于其他上下文（例如 builder-style 的方法链）。

例子：

```
public class Car {
  // Injectable constructor
  @Inject public Car(Engine engine) { ... }

  // Injectable field
  @Inject private Provider<Seat> seatProvider;

  // Injectable package-private method
  @Inject void install(Windshield windshield, Trunk trunk) { ... }
}
```

当一个方法标注了 @Inject 并覆写了其他标注了 @Inject 的方法时，对于每一个实例的每一次注入请求，该方法只会被注入一次。当一个方法没有标注 @Inject 并覆写了其他标注了 @Inject 的方法时，该方法不会被注入。

要进行成员注入就必须标注 @Inject。一个可注入的成员可以使用任何访问修饰符（包括 private）。不过受于平台或注入器限制（例如安全管理或缺乏反射支持），标注了 @Inject 的非公有成员可能将不被注入。

限定器

限定器 注解用于标注可注入的字段或参数，外加该字段或参数的类型，就可以标识出待注入的实现。限定符是可选的，当与 @Inject 一起使用在与注入器无关的类时，对于一个字段或参数，应该最多只有一个限定符被标注。在下面的例子中，限定符被标注了粗体：

```
public class Car {
  @Inject private @Leather Provider<Seat> seatProvider;

  @Inject void install(@Tinted Windshield windshield,
              @Big Trunk trunk) { ... }
}
```

如果一个可注入的方法覆写了其他方法，覆写方法的参数不会自动地从被覆写的方法上继承限定器。

可注入的值

对于一个给定的类型 T 与可选的限定器，注入器必须能够注入用户指定的类：

a. 与 T 是赋值兼容的，并且

b. 有一个可注入的构造器。

例如，用户可能使用外部配置来选择一个 T 的实现。此外，待注入的值取决于注入器实现与它的配置。

循环依赖

本规范并未详细要求探测循环依赖与解析循环依赖。两个构造器间的循环依赖是一个非常明显的问题，另外，对于可注入字段或方法的循环依赖也很常见，例如：

```
class A {
  @Inject B b;
}
class B {
  @Inject A a;
}
```

当构造 A 的一个实例时，一个简单的注入器可能会无限循环构造：B 的一个实例注入给 A 的一个实例，第二个 A 的实例注入给 B 的一个实例，第二个 B 的实例注入给第二个 A 的实例，.....

一个保守的注入器可能会在构建时探测出这个循环依赖，并生成一个错误，指出程序员可以使用 Provider<A> 或 Provider<B> 对应替换 A 或 B 来打破这个循环依赖。从注入的构造器或方法调用该 provider 的 get() 将打破这个循环依赖。对于方法或字段注入的情况，将其依赖的一边放置到某作用域（例如单例作用域 ）也可以使得循环依赖能够被注入器解析。

请查阅：

@Qualifier

@Provider

@Qualifier

注解 @Qualifier 用于标识限定器注解。任何人都可以定义新的限定器注解。一个限定器注解：

- 是被 @Qualifier、@Retention(RUNTIME) 标注的，通常也被 @Documented 标注。
- 可以拥有属性。
- 可能是公共 API 的一部分，就像依赖类型一样，而不像类型实现那样不作为公共 API 的一部分。
- 如果标注了 @Target 可能会有一些用法限制。本规范只是指定了限定器注解可以被使用在字段和参数上，但一些注入器配置可能使用限定器注解在其他一些地方（例如方法或类）上。

例子：

```
@java.lang.annotation.Documented
@java.lang.annotation.Retention(RUNTIME)
@javax.inject.Qualifier
public @interface Leather {
  Color color() default Color.TAN;
  public enum Color { RED, BLACK, TAN }
}
```

请查阅：

@Named

Provider<T>

接口 Provider 用于提供类型 T 的实列。Provider 是一般情况是由注入器实现的。对于任何可注入的 T 而言，您也可以注入 Provider<T>。与直接注入 T 相比，注入 Provider<T> 使得：

- 可以返回多个实例。
- 实例的返回可以延迟化或可选
- 打破循环依赖。
- 可以在一个已知作用域的实例内查询一个更小作用域内的实例。

例子：

```
class Car {
  @Inject Car(Provider<Seat> seatProvider) {
    Seat driver = seatProvider.get();
    Seat passenger = seatProvider.get();
    ...
  }
}
```

get()

用于提供一个完全构造的类型 T 的实例。

异常抛出：RuntimeException —— 当注入器在提供实例时遇到错误将抛出此异常。例如，对于一个可注入的成员 T 抛出了一个异常，注入器将包装此异常并将它抛给 get() 的调用者。调用者不应该尝试处理此类异常，因为不同注入器实现的行为不一样，即使是同一个注入器，也会因为配置不同而表现的行为不同。

@Named

基于 String 的限定器 。

例子：

```
public class Car {
  @Inject @Named("driver") Seat driverSeat;
  @Inject @Named("passenger") Seat passengerSeat;
  ...
}
```

@Scope

注解 @Scope 用于标识作用域注解。一个作用域注解是被标识在包含一个可注入构造器的类上的，用于控制该类型的实例如何被注入器重用。缺省情况下，如果没有标识作用域注解，注入器将为每一次注入都创建（通过注入类型的构造器）新实例，并不重用已有实例。如果多个线程都能够访问一个作用域内的实例，该实例实现应该是线程安全的。作用域实现由注入器完成。

在下面的例子中，作用域注解 @Singleton 确保我们始终只有一个 Log 实例：

```
 @Singleton
class Log {
  void log(String message) { ... }
}
```

当多于一个作用域注解或不被注入器支持的作用域注解被使用在同一个类上时，注入器将生成一个错误。

一个作用域注解：

- 被标注了 @Scope、@Retention(RUNTIME) 标注的，通常也被 @Documented 标注。
- 不应该含有属性。
- 不应该被 @Inherited 标注，因此作用域与继承实现（正交）无关。
- 如果标注了 @Target 可能会有一些用法限制。本规范只是指定了作用域注解可以被使用在类上，但一些注入器配置可能使用作用域注解在其他一些地方（例如工厂方法返回）上。

例子：

```
@java.lang.annotation.Documented
@java.lang.annotation.Retention(RUNTIME)
@javax.inject.Scope
public @interface RequestScoped {}
```

使用 @Scope 来标识一个作用域注解有助于注入器探测程序员使用了作用域注解但却忘了去配置作用域的情况。一个保守的注入器应该生成一个错误而不是去适用该作用域。

请查阅：

@Singleton

@Singleton

注解 @Singleton 标识了注入器只实例化一次的类型。该注解不能被继承。

请查阅：

@Scope