**Ifeng-Common-Manual**

# 介绍

## 概念列表

SoC:Separation of Concerns

POJO:Plain Old Java Object

IoC:Inverse Of Control

## 概念的目的

灵活的架构

代码复用

降低程序的复杂度

降低开发维护成本

提高代码质量

## SoC

关注点分离，是软件设计的普遍原则。

将一个复杂的问题，变成若干个简单问题的组合能力，使得程序开发过程只是面对简单的问题，从而使得问题的思考和描述落入更多人的能力范围内。

关注分离是一个非常泛化的原则，不只是针对程序设计方面，对人员分工同样有效。

## POJO

最初来自于EJB领域

普通的Java对象 vs. 依赖于EJB框架的对象

意义:

1 与框架、容器、运行环境无关。

2 代码直接复用。

3 让实现视图尽量接近逻辑视图，而过程视图、部署视图可配置，由框架和容器提供，编程人员在编写POJO时无需考虑。这样定义一个业务数据的过程变得简单而高效。

EJB排斥POJO，而作为一个轻量级的实践者，POJO显然是首选。

定义数据的过程没有比POJO更简单的了，所以如果能在POJO的前提依然有效的完成过程和部署关系的抽象和开发，简直是一件让人赏心悦目的事儿。

## IoC

思想来自好莱坞原则：

Don’t call us , we’ll call u.

“在好莱坞，演员把简历递交给演艺公司后就只能回家等待。演艺公司对整个娱乐项的完全控制，演员只能被动的接受公司的差使,在需要的时候，完成对应的自己的演出。”

为了保证上层程序的复用，架构人员参考了好莱坞原则，给出了IoC的定义和初衷。

马丁·福勒 Martin Fowler（UML、领域建模、敏捷开发、极限编程） 是IoC概念的先驱，所以我们引用他对IoC的描述做一个开场：

*2004年，Martin Fowler就提出了“哪些方面的控制被反转了？”这个问题。他自己的总结出是依赖对象的获得被反转了。基于这个结论，他为控制反转创造了一个更好的名字：依赖注入。许多非凡的应用（比HelloWorld.java更加优美，更加复杂）都是由两个或是更多的类通过彼此的合作来实现业务逻辑，这使得每个对象都需要，与其合作的对象（也就是它所依赖的对象）的引用。如果这个获取过程要靠自身实现，那么将导致代码高度耦合并且难以测试。*

*IoC 亦称为 “依赖倒置原理”("Dependency Inversion Principle") (Martin 2002)。很多所有框架都使用了这个技巧。*

*应用控制反转，对象在被创建的时候，由调用者将其所依赖的对象的引用，传递给它。也可以说，依赖被注入到对象中。所以，控制反转是，关于一个对象如何获取它所依赖的对象的引用，这个责任的反转。*

上面这段文字对中级程序员来说还是有点坑爹，云里雾里的。实际上的原理比较简单,就是为了保障程序本身不依赖于其他对象或者具体的数据，所以程序本身不应该创建其依赖的对象，而由调用者在构建程序对应的对象时一并传递这些其他的对象和数据。

这个调用者是谁呢？

我们为了完成一个业务框架，通常需要做一些抽象，而这些抽象的边界和扩展性决定了框架的适用范围，为了保障框架的使用范围更加的广泛，往往这些约束都是通过接口或者一些未确定的数据来描述的。而在运行时，业务框架本身是无法去执行一个具体的业务场景的，需要有人使用业务框架去完成相关的一些实现，这些实现是框架需要的业务能力，框架在需要具体业务能力的时候，由框架自己的注入能力将业务能力注入进来。

小一点的范围的应用是类和类之间的依赖关系的去除。

# com.ifeng.common.conf

## 原理

IoC容器，完成依赖注入能力。

类似 Spring ，但是更轻量级，更关注扩展性和易用性。(common.conf 共 1793行代码+注释 ， 563分号行)

## 基本配置方式

example.xml:

<?xml version="1.0" encoding="GB2312"?>

<config>

<config name="example" type="com.ifeng.example.Example">

<set-property name="name" value="a"/>

<set-property name="value" value="aa"/>

</config>

</config>

**public** **class** Example {

**private** String name;

**private** String value;

**public** Example () {

}

**public** String getName() {

**return** name;

}

**public** String getValue() {

**return** value;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** setValue(String value) {

**this**.value = value;

}

}

## Type说明

针对

<config name="example" type="com.ifeng.example.Example">

描述的为该段配置对应的类型。

包含如下几种形式：

1 缺省类型，即不指定type时，默认为java.lang.String

2 支持基本数据类型（primitive）：byte/char/short/boolean/int/long/float 等。

3 com.ifeng.common.conf中的类型，使用“.”开头直接跟类名，可以省略前面的包名如：

com.ifeng.common.conf.MapConfig 可以写为.MapConfig

4 其他类型，使用包含报名的类名来描述如上面的example。

## 对象的配置方式

### 构造函数

例如：

<config name="TestConstructorArg3" type=".ConfigForTest">

<constructor-arg value="strValue1"/>

<constructor-arg type="int" value="999"/>

<constructor-arg type="com.ifeng.common.misc.IpV4Address" value="2.3.4.5"/>

<constructor-arg declared-type=".ConfigForTest" type=".ReferenceConfig"

reference="TestConstructorArg2"/>

</config>

会自动匹配，数量匹配且类型最接近的构造函数。

### setter

<config name="TestSetProperty" type=".ConfigForTest">

<set-property name="strValue" value="str"/>

<set-property name="intValue" value="0x1234abcd"/>

<set-property name="ipV4Value" value="1.2.3.4"/>

<set-property name="objValue" value="stringObj"/>

</config>

会首先调用没有参数的构造函数，然后调用对应的set方法。

需要有一个无参数的构造函数.

适合普通的POJO类。

如果含有多个字符串参数，可以简写为：

<config name="TestSetProperties1" type=".ConfigForTest">

<set-properties strValue="str100" intValue="100"

ipV4Value="128.129.130.131" objValue="stringObj"/>

</config>

调用过程一致。

### 特殊的构造方式

需要待构造的类实现com.ifeng.common.conf.Configurable接口。

/\*\*

\* 配置一个对象.

\*

\* **@param** configRoot 当前的配置上下文对象(在一个子系统中可能只有一个)

\* **@param** parent 上级对象

\* **@param** configEle 配置元素

\* **@return** 配置完成的对象，一般应该是this,但是也可以不是它自己

\* **@throws** ConfigException 配置出错时 (unchecked)

\*/

**public** Object config(ConfigRoot configRoot, Object parent, Element configEle)

**throws** ConfigException /\* unchecked \*/;

实现此方法，即自己解析属于自己的配置，然后构造一个自己并返回。

解析时可以依赖configRoot提供的解析能力。

该方式的例子目的可以参考Configurable接口的javadoc：



好处：代替很多情况下的constructor-arg或set-property，使配置文件更可读 。

可能的弊端：引入了对象对框架的依赖关系

使得这个对象不是POJO了

实现时需要权衡

配置文件的可读性 vs. 配置框架的无关性

一般情况下，我们偏重于前者.

## 容器的装载

ConfigRoot configRoot = new ConfigRoot(“配置文件路径”,System.getProperties());

也可以将路径和文件名分开，并使用URL做参数。参考ConfigRoot的其他构造方法。

然后可以用下面的代码得到配置好的对象：

ConfigForTest obj1 = (ConfigForTest)configRoot

.getValue("TestConstructorArg1");

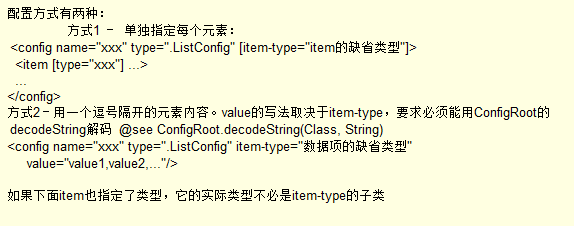
即得到配置文件中下面的对象：

<config name="TestConstructorArg1" type=".ConfigForTest"/>

## Conf包中的扩展类型

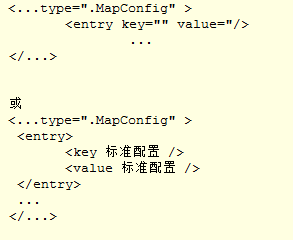
### ListConfig

配置一个java.util.ArrayList

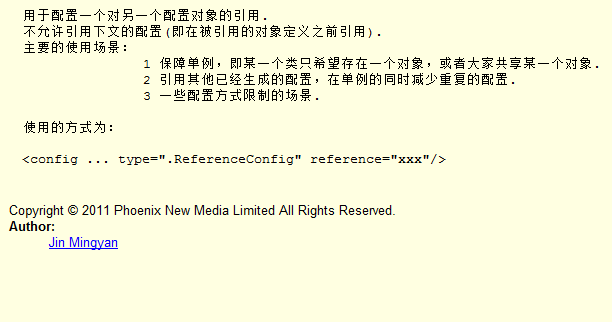


### MapConfig

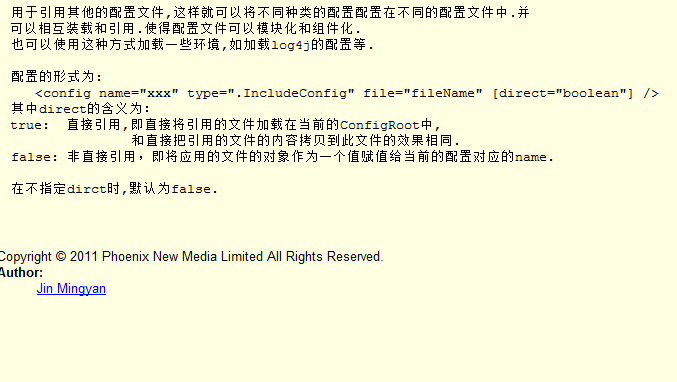
配置一个java.util.HashMap



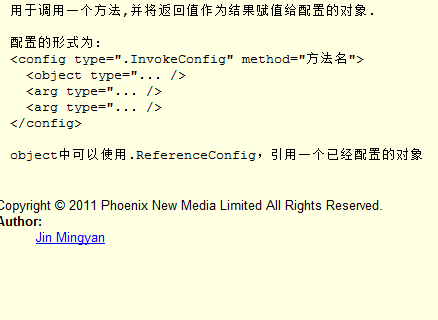
### ReferenceConfig



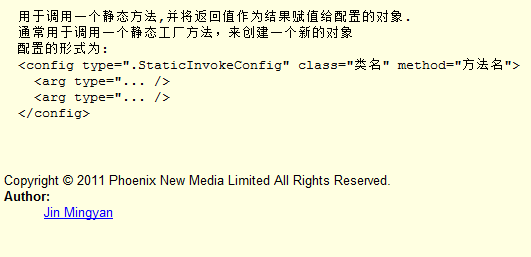
### IncludeConfig



### InvokeConfig



### StaticInvokeConfig



### LogConfig

上文配置完成的log为一个log4j的配置，需要代码中使用com.ifeng.common.misc.Logger类来初始化log。

## 一些建议

IncludeConfig 文件的相对路径决定于最外层的配置文件的路径。

IncludeConfig尽量采用 direct="true" 的形式，以提高配置文件重用时的灵活性。

ReferenceConfig中的reference尽量采用"."开头的本地引用方式，避免Include层次影响引用。

properties 文件只能在文件系统中使用，不能在jar包中使用。

配置文件过于复杂时，建议将复杂而固定不变的配置项用xml形式放在jar包中，在xml中引用放在 conf 目录中的properties文件，以简化用户的配置。

可以将 IncludeConfig 配置项指向的文件名放在 properteis 文件中定义，以便在需要时通过指定文件目录中修改过的配置文件来修改配置文件内容。

Configurable 接口提供了配置格式的灵活性，但是除了非常复杂的配置，还是应该尽量少使用 config 方法，以避免造成框架对业务对象的污染。

不要随意通过装载配置文件来生成ConfigRoot，应该在单一位置装载 xml（在服务器启动时或者在的确无法注入的地方，比如 servlet 引用的工厂中）。

ConfigRoot 只应该出现在两个地方：服务器启动的 main 方法，及 config 方法内部(即应该遵循 IoC 的思想，对象间的依赖应该通过注入的方式（构造方法或者set方法）得到，而不应该用查询的方法得到)。

Configurable 应该不影响类的行为，可以 config 出来的对象，应该也能通过创建对象和 set 方法构造出来。