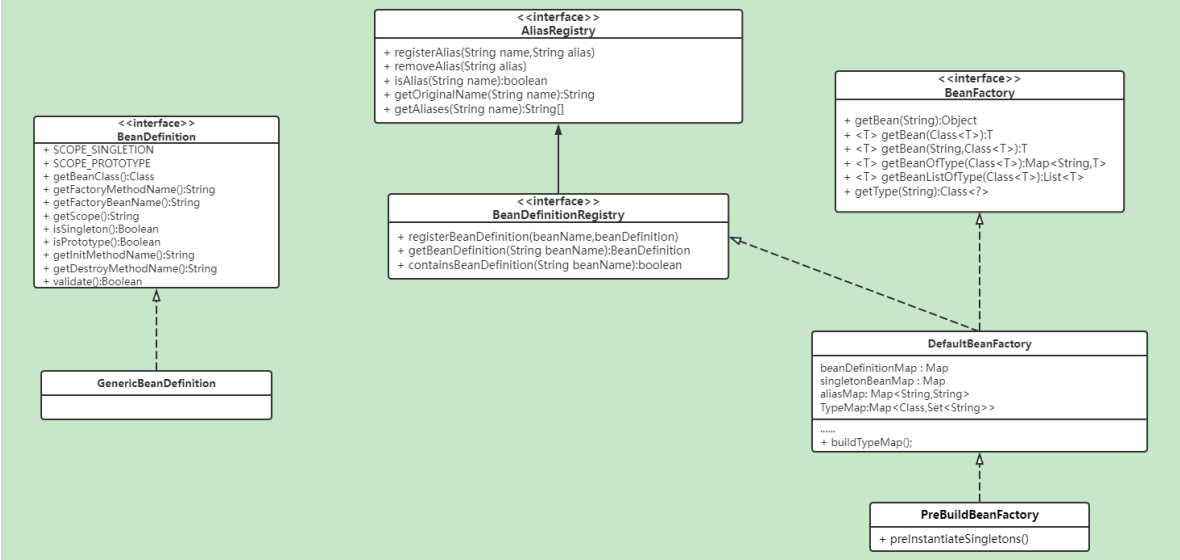
Spring源码手写篇-手写DI

简单回顾前面的手写IoC的内容。



一、DI介绍

DI(Dependency injection)依赖注入。对象之间的依赖由容器在运行期决定，即容器动态的将某个依赖注入到对象之中。说的直白点就是给Bean对象的成员变量赋值。

在这里我们就需要明白几个问题。

1. 哪些地方会有依赖

- 构造参数依赖
- 属性依赖

2. 依赖注入的本质是什么?

依赖注入的本质是`赋值`。赋值有两种情况

1. 给有参构造方法赋值
2. 给属性赋值

3. 参数值、属性值有哪些?

具体赋值有两种情况：直接值和Bean依赖。比如

```
public clsss Girl{
     public Girl(String name,int age,char cup,Boy boyfriend){
         ...
     }
}
```

4. 直接赋值有哪些?

- 基本数据类型：String、int 等
- 数组，集合
- map

二、构造注入

我们先来看看构造参数注入的情况应该要如何解决。

1.构造注入分析

我们应该如何定义构造参数的依赖？也就是我们需要通过构造方法来创建实例，然后对应的构造方法我们需要传入对应的参数。如果不是通过IoC来处理，我们可以直接通过如下的代码实现。

```
    public static void main(String[] args) {
        Boy boy = new Boy();
        Girl girl = new Girl("小丽",20,'C',boy);
    }
```

我们通过直接赋值的方式就可以了。但是在IoC中我们需要通过反射的方式来处理。那么我们在通过反射操作的时候就需要能获取到对应的构造参数的依赖了，这时我们得分析怎么来存储我们的构造参数的依赖了。构造参数的依赖有两个特点：

- 数量
- 顺序

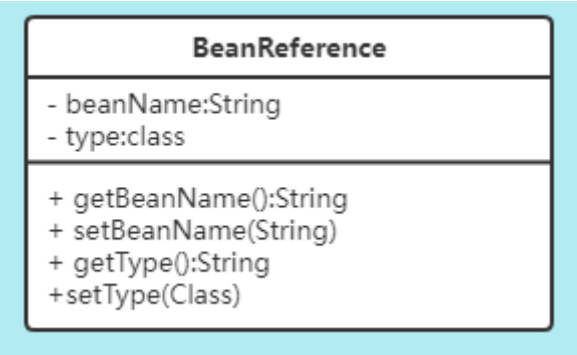
上面的例子中的参数

1. 小丽
2. 20
3. 'C'
4. boy,是一个依赖Bean

参数可以有多个，我们完全可以通过List集合来存储，而且通过添加数据的顺序来决定构造参数的顺序了。但是这里有一个问题，如何表示Bean依赖呢？直接值我们直接添加到集合中就可以了，但是Bean依赖，我们还没有创建对应的对象，这时我们可以维护一个自定义对象，来绑定相关的关系。

2. BeanReference

BeanReference就是用来说明bean依赖的：也就是这个属性依赖哪个类型的Bean



可以根据name来依赖，也可以按照Type来依赖。当然我们的程序中还有一点需要考虑，就是如何来区分是直接值还是Bean依赖呢？有了上面的设计其实就很容易判断了。

```
if ( obj instance BeanReference)
```

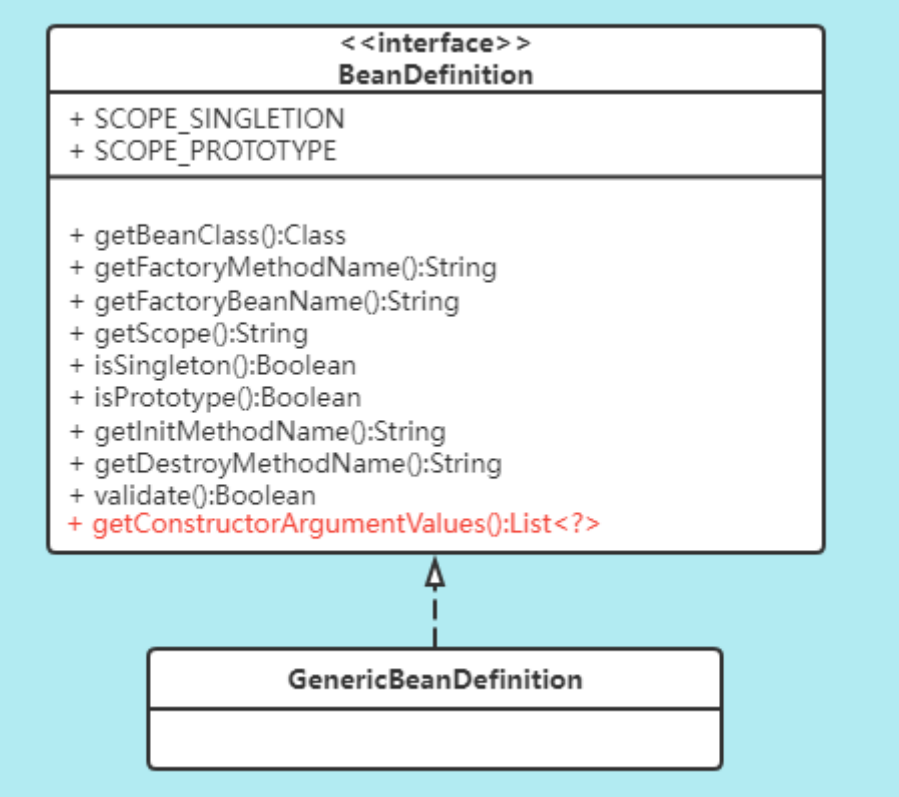
当然还有一种比较复杂的情况如下：

```
public class Girl {

    private String name;
    private int age;
    private char cup;
    private List<Boy> boyFriends;

    public Girl(String name, int age, char cup, List<Boy> boyFriends) {
        this.name = name;
        this.age = age;
        this.cup = cup;
        this.boyFriends = boyFriends;
    }
}
```

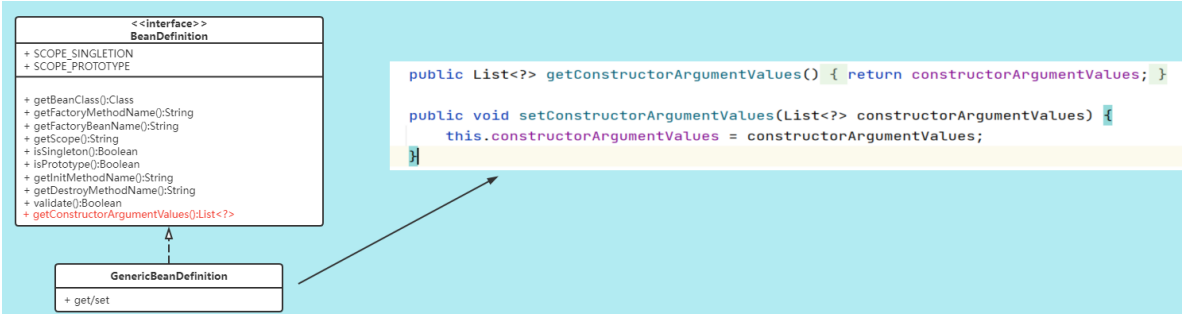
直接值是数组或者集合等，同时容器中的元素是Bean依赖，针对这种情况元素值还是需要用BeanReference来处理的。Bean工厂在处理时需要遍历替换。

3. BeanDefinition实现

接下来我们看看如何具体的来实现DI基于构造参数依赖的相关操作。首先是定义的相关处理了。我们需要在 `BeanDefinition` 中增加构造参数的获取的方法。



然后我们需要在默认的实现GenericBeanDefinition中增加对应的方法来处理。



定义后我们可以测试下对应的应用，定义个ABean，依赖了CBean

```
public class ABean {

    private String name;

    private CBean cb;

    public ABean(String name, CBean cb) {
        super();
        this.name = name;
        this.cb = cb;
        System.out.println("调用了含有CBean参数的构造方法");
    }

    public ABean(String name, CCBean cb) {
        super();
        this.name = name;
        this.cb = cb;
        System.out.println("调用了含有CCBean参数的构造方法");
    }

    public ABean(CBean cb) {
        super();
        this.cb = cb;
    }

    public void doSomthing() {
        System.out.println("Abean.doSomthing(): " + this.name + " cb.name=" +
this.cb.getName());
    }

    public void init() {
        System.out.println("ABean.init() 执行了");
    }

    public void destroy() {
        System.out.println("ABean.destroy() 执行了");
    }
}
```

然后在实例化时我们需要做相关的绑定

```
        GenericBeanDefinition bd = new GenericBeanDefinition();
        bd.setBeanClass(ABean.class);
        // 定义的构造参数的依赖
        List<Object> args = new ArrayList<>();
        args.add("abean01");
        // Bean依赖 通过BeanReference 来处理
        args.add(new BeanReference("cbean"));
        bd.setConstructorArgumentValues(args);
        bf.registerBeanDefinition("abean", bd);
```

构造参数传递后，接下来其实我们就需要要在 `BeanFactory` 中来实现构造参数的注入了

4.BeanFactory实现

前面我们在BeanFactory中实现Bean对象的创建有几种方式

- 构造方法创建
- 工厂静态方法
- 工厂成员方法

我们在通过构造方法创建其实是通过无参构造方法来处理的，这时我们需要改变这块的逻辑，通过有参构造方法来实现。

```
    // 构造方法来构造对象
    private Object createInstanceByConstructor(BeanDefinition bd)
            throws InstantiationException, IllegalAccessException {
        try {
            return bd.getBeanClass().newInstance();
        } catch (SecurityException e1) {
            log.error("创建bean的实例异常,beanDefinition: " + bd, e1);
            throw e1;
        }
    }
```

我们就需要对上面的方法做出改变。

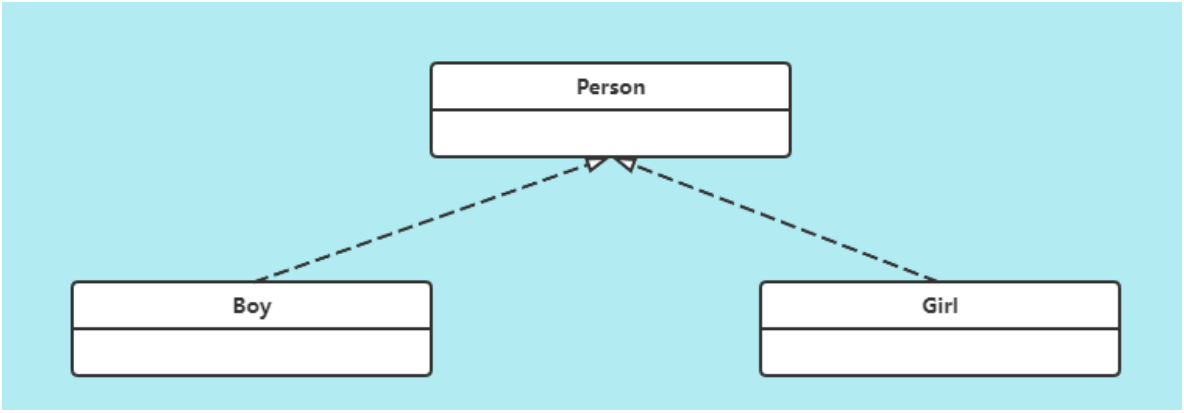
```
    // 构造方法来构造对象
    private Object createInstanceByConstructor(BeanDefinition bd)
            throws InstantiationException, IllegalAccessException {
        // 1. 得到真正的参数值
        List<?> constructorArgumentValues = bd.getConstructorArgumentValues();
        // 2.根据对应的构造参数依赖获取到对应的 Constructor
        Constructor  constructor = 得到对应的构造方法
        // 3.用实际参数值调用构造方法创建对应的对象
        return constructor.newInstance(Object ... 实参值);
    }
```

通过上面的分析我们需要获取对应的构造器。这块我们需要通过反射来获取了。下面是具体的实现逻辑

根据上面的分析，我们实现的逻辑分为两步

1. 先根据参数的类型进行精确匹配查找，如果没有找到，继续执行第二步操作
2. 获得所有的构造方法，遍历构造方法，通过参数数量过滤，再比对形参与实参的类型

因为这里有个情况，实参是Boy，构造方法的形参是Person，第一种精确匹配就没有办法关联了。



具体的实现代码如下：

```
    private Constructor<?> determineConstructor(BeanDefinition bd, Object[]
args) throws Exception {
        /*判定构造方法的逻辑应是怎样的？
        1 先根据参数的类型进行精确匹配查找，如未找到，则进行第2步查找；
        2获得所有的构造方法，遍历，通过参数数量过滤，再比对形参类型与实参类型。
        * */

        Constructor<?> ct = null;

        //没有参数，则用无参构造方法
        if (args == null) {
            return bd.getBeanClass().getConstructor(null);
        }

        // 1 先根据参数的类型进行精确匹配查找
        Class<?>[] paramTypes = new Class[args.length];
        int j = 0;
        for (Object p : args) {
            paramTypes[j++] = p.getClass();
        }
        try {
            ct = bd.getBeanClass().getConstructor(paramTypes);
        } catch (Exception e) {
            // 这个异常不需要处理
        }

        if (ct == null) {
```

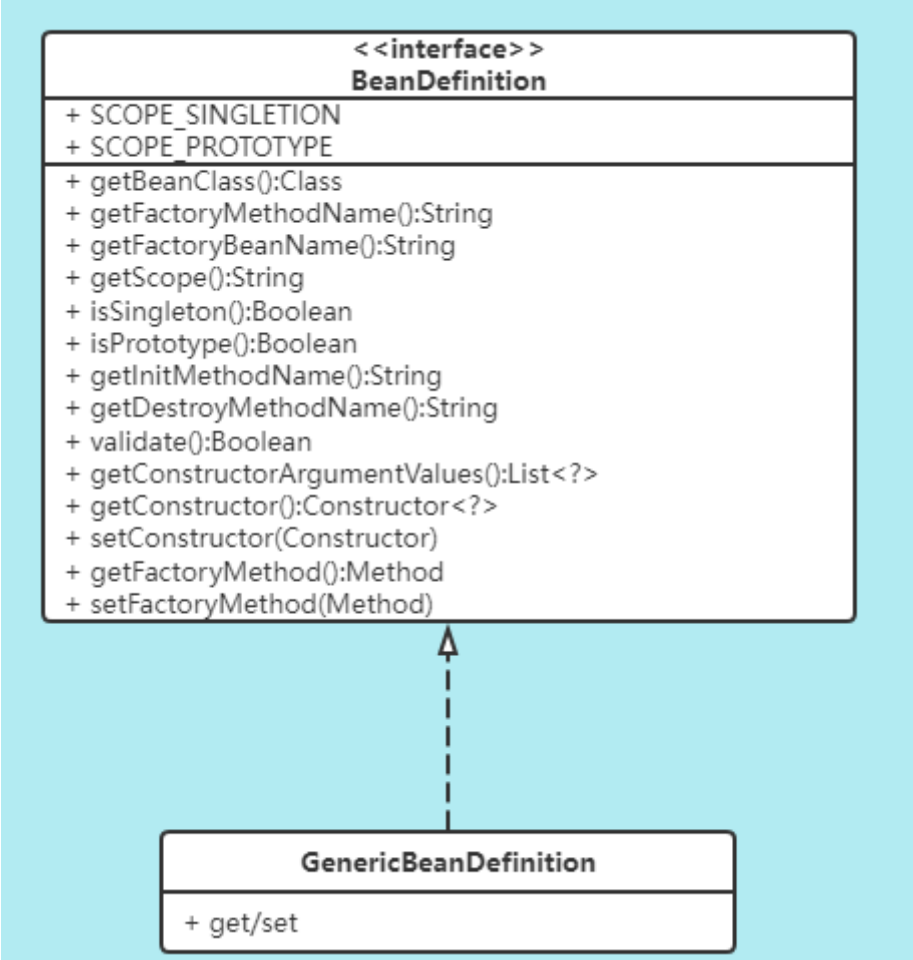
```
            // 2 没有精确参数类型匹配的，获得所有的构造方法，遍历，通过参数数量过滤，再比对形参类型与实参类型。
            // 判断逻辑：先判断参数数量，再依次比对形参类型与实参类型
            outer:
            for (Constructor<?> ct0 : bd.getBeanClass().getConstructors()) {
                Class<?>[] paramterTypes = ct0.getParameterTypes();
                if (paramterTypes.length == args.length) {   //通过参数数量过滤
                    for (int i = 0; i < paramterTypes.length; i++) { //再依次比对形参类型与实参类型是否匹配
                        if (!paramterTypes[i].isAssignableFrom(args[i].getClass())) {
                            continue outer; //参数类型不可赋值（不匹配），跳到外层循环，继续下一个
                        }
                    }

                    ct = ct0;  //匹配上了
                    break outer;
                }
            }
        }

        if (ct != null) {
            return ct;
        } else {
            throw new Exception("不存在对应的构造方法! " + bd);
        }
    }
```

上面我们考虑的是BeanFactory通过构造器来获取对象的逻辑，那如果我们是通过静态工厂方法或者成员工厂方法的方式来处理的，那么构造参数依赖的处理是否和前面的是一样的呢？其实是差不多的，我们需要根据对应的构造参数来推断对应的工厂方法

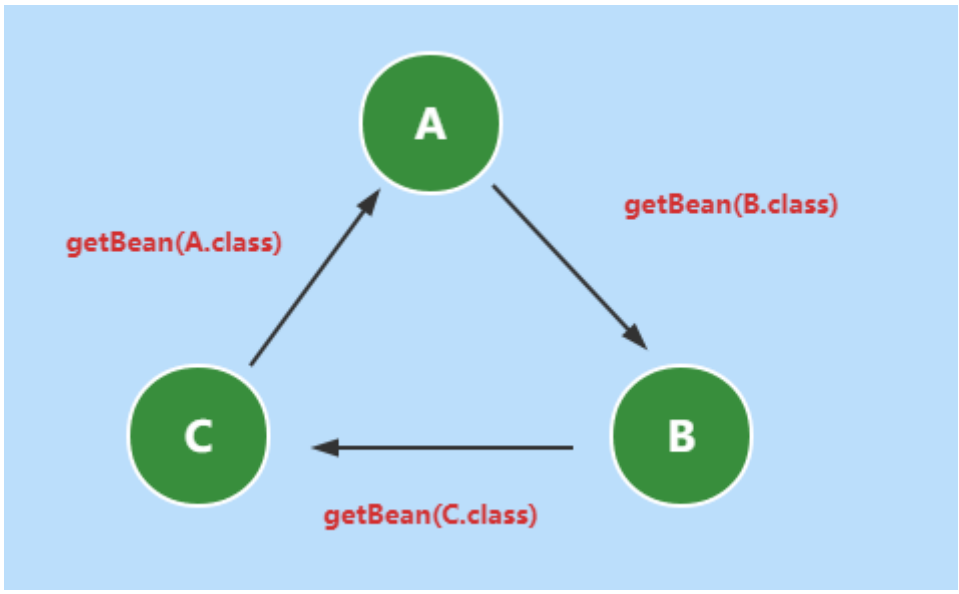
```
    // 静态工厂方法
    private Object createInstanceByStaticFactoryMethod(BeanDefinition bd) throws Exception {

        Object[] realArgs = this.getConstructorArgumentValues(bd);
        Class<?> type = bd.getBeanClass();
        Method m = this.determineFactoryMethod(bd, realArgs, type);
        return m.invoke(type, realArgs);
    }

    // 工厂bean方式来构造对象
    private Object createInstanceByFactoryBean(BeanDefinition bd) throws Exception {

        Object[] realArgs = this.getConstructorArgumentValues(bd);
        Method m = this.determineFactoryMethod(bd, realArgs, this.getType(bd.getFactoryBeanName()));

        Object factoryBean = this.doGetBean(bd.getFactoryBeanName());
        return m.invoke(factoryBean, realArgs);
    }
```

5.缓存功能

对于上面的处理过程相信大家应该清楚了，我们通过推断也得到了对应的构造方法或者对应的工厂方法，那么我们可以不可以在下次需要再次获取的时候省略掉推导的过程呢？显然我们可以在BeanDefinition中增加缓存方法可以实现这个需求。



6. 循环依赖问题



上图是循环依赖的三种情况，虽然方式有点不一样，但是循环依赖的本质是一样的，就你的完整创建要依赖与我，我的完整创建也依赖于你。相互依赖从而没法完整创建造成失败。

我们通过构造参数依赖是完全可能出现上面的情况的，那么这种情况我们能解决吗？构造依赖的情况我们是解决不了的。

```
public class Test01 {
```

```
    public static void main(String[] args) {
        new TestService1();
    }
}

class TestService1{
    private TestService2 testService2 = new TestService2();
}

class TestService2{
    private  TestService1 testService1 = new TestService1();
}
```

既然解决不了，那么我们在程序中如果出现了，应该要怎么来解决呢？其实我们可以在创建一个Bean的时候记录下这个Bean，当这个Bean创建完成后我们在移除这个Bean，然后我们在getBean的时候判断记录中是否有该Bean，如果有就判断为循环依赖，并抛出异常。数据结构我们可以通过Set集合来处理。

```
        BeanDefinition bd = this.getBeanDefinition(beanName);
        Objects.requireNonNull(bd,  message: "beanDefinition不能为空");

        // 检测循环依赖
        Set<String> buildingBeans = this.buildingBeansRecordor.get();
        if (buildingBeans == null) {
            buildingBeans = new HashSet<>();
            this.buildingBeansRecordor.set(buildingBeans);
        }

        // 检测循环依赖
        if (buildingBeans.contains(beanName)) {
            throw new Exception(beanName + " 循环依赖! " + buildingBeans);
        }
        // 记录正在创建的Bean
        buildingBeans.add(beanName);

        if(bd.isSingleton()) { //如果是单例
```

到此构造注入的实现我们就搞定了。

三、属性注入

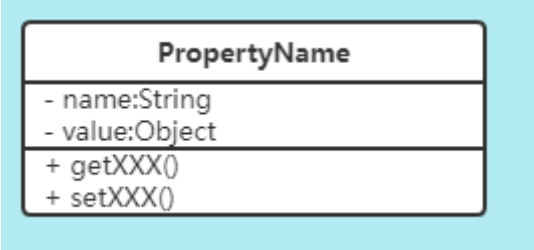
上面搞定了构造注入的方式。接下来我们再看看属性注入的方式有什么需要注意的地方。

1. 属性依赖分析

属性依赖就是某个属性依赖某个值。

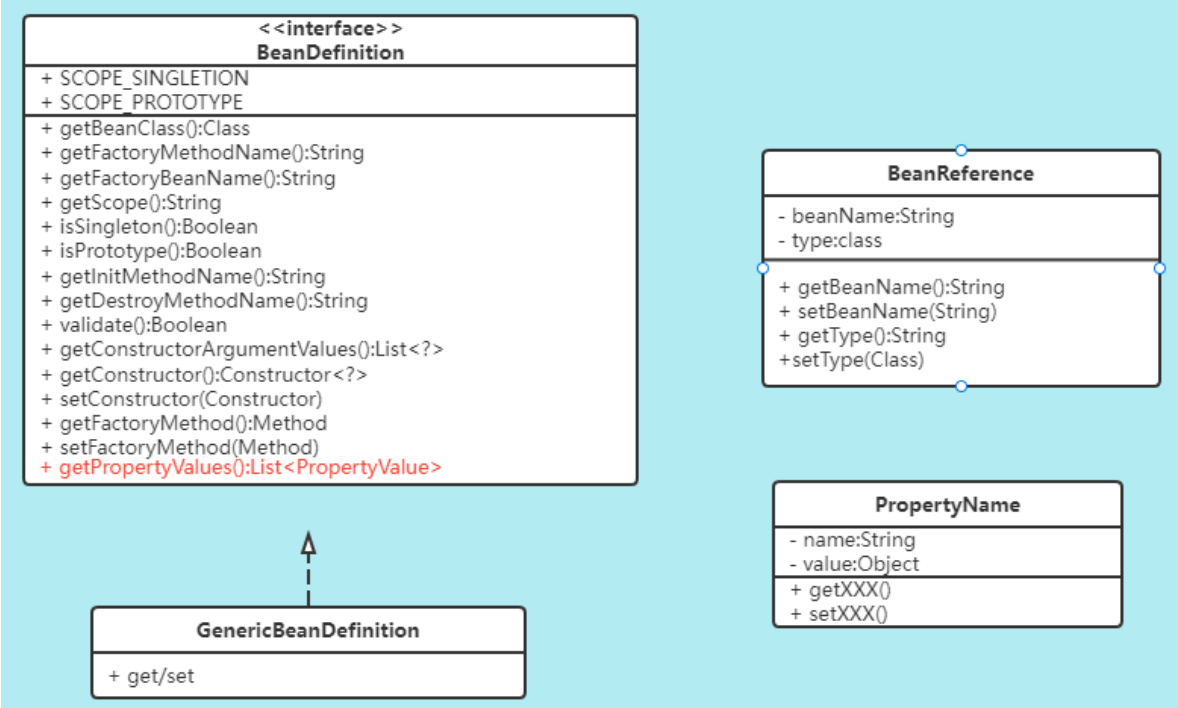
```java
public class Girl {

    private String name;
    private int age;
    private char cup;
    private List<Boy> boyFriends;

    // ....
}
```

那么在获取实例对象后如何根据相关的配置来给对应的属性来赋值呢？这时我们可以定义一个实体类`PropertyValue`来记录相关的属性和值。



2.BeanDefinition实现

这时我们就需要在BeanDefinition中关联相关属性信息了。



3.BeanFactory实现

然后我们在BeanFactory的默认实现DefaultBeanFactory中实现属性值的依赖注入。

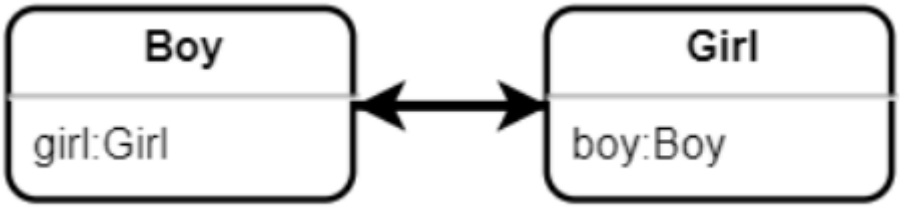
```java
// 创建好实例对象
// 给属性依赖赋值
this.setPropertyDIValues(bd,instance);
// 执行初始化相关方法
this.doInit(bd,instance);
```

具体的实现代码如下：

```
    // 给入属性依赖
    private void setPropertyDIValues(BeanDefinition bd, Object instance) throws
Exception {
        if (CollectionUtils.isEmpty(bd.getPropertyValues())) {
            return;
        }
        for (PropertyValue pv : bd.getPropertyValues()) {
            if (StringUtils.isBlank(pv.getName())) {
                continue;
            }
            Class<?> clazz = instance.getClass();
            Field p = clazz.getDeclaredField(pv.getName());
            //暴力访问  private
            p.setAccessible(true);
            p.set(instance, this.getOneArgumentRealValue(pv.getValue()));

        }
    }
```

4.循环依赖问题

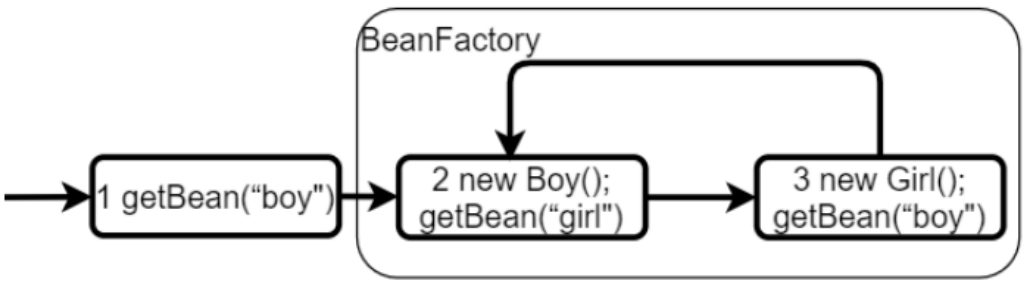
在构造参数依赖中我们发现没有办法解决，在属性依赖中同样会存在循环依赖的问题，这时我们能解决吗?



其实这种情况我们不在IoC场景下非常好解决。如下

```
Boy b = new Boy();
Girl g = new Girl();
b.setGirl(g);
g.setBoy(b);
```

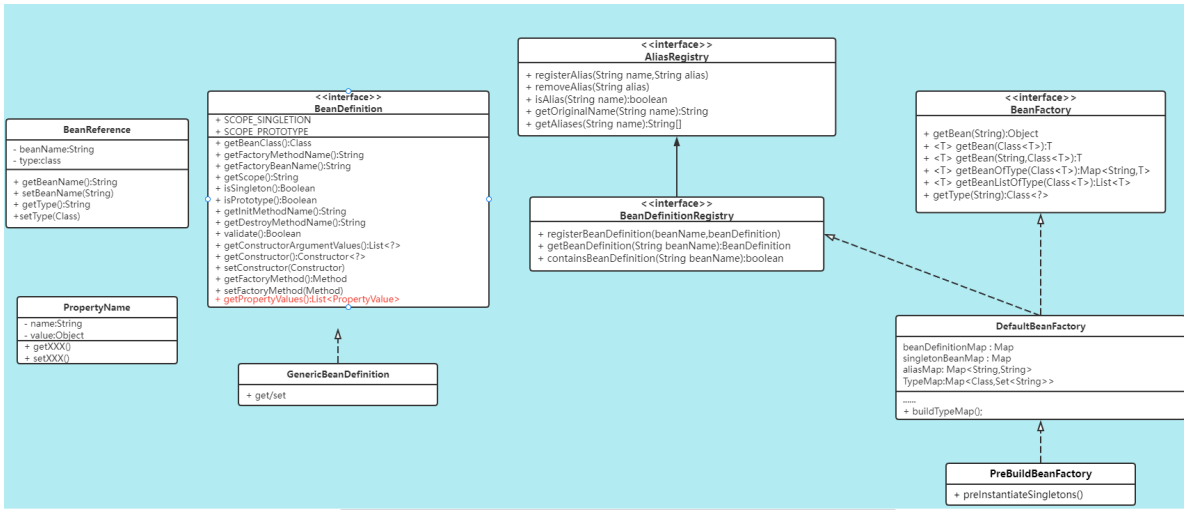
但是在IoC好像不是太好解决：



针对这种情况我们需要通过 `提前暴露` 来解决这个问题，具体看代码!!!

```
    private void doEarlyExposeBuildingBeans(String beanName, Object instance) {
        Map<String,Object> earlyExposeBuildingBeansMap =
earlyExposeBuildingBeans.get();
        if(earlyExposeBuildingBeansMap == null) {
            earlyExposeBuildingBeansMap = new HashMap<>();
            earlyExposeBuildingBeans.set(earlyExposeBuildingBeansMap);
        }
        earlyExposeBuildingBeansMap.put(beanName,instance);
    }
```

最后现阶段已经实现的类图结构



扩展作业：加入Bean配置的条件依赖生效的支持

在Bean定义配置中可以指定它条件依赖某些Bean或类，当这些Bean或类存在时，这个bean的配置才能生效!