从零开始，手写 Spring IOC

作者：fjiang
原文链接：https://ld246.com/article/1624354513057
来源网站：链滴

手写Spring ioc

1. 什么是ioc?

2. ioc工作原理

3. 实现ioc

什么是ioc?

解决对象管理和对象依赖的问题

本来是我们手动new出的对象，现在交给Spring的IOC容器管理，IOC容器可以理解为一个对象工厂我们把该对象交给工厂，工厂管理这些对象的创建以及依赖关系。当我们需要用对象的时候，直接从厂里拿即可。

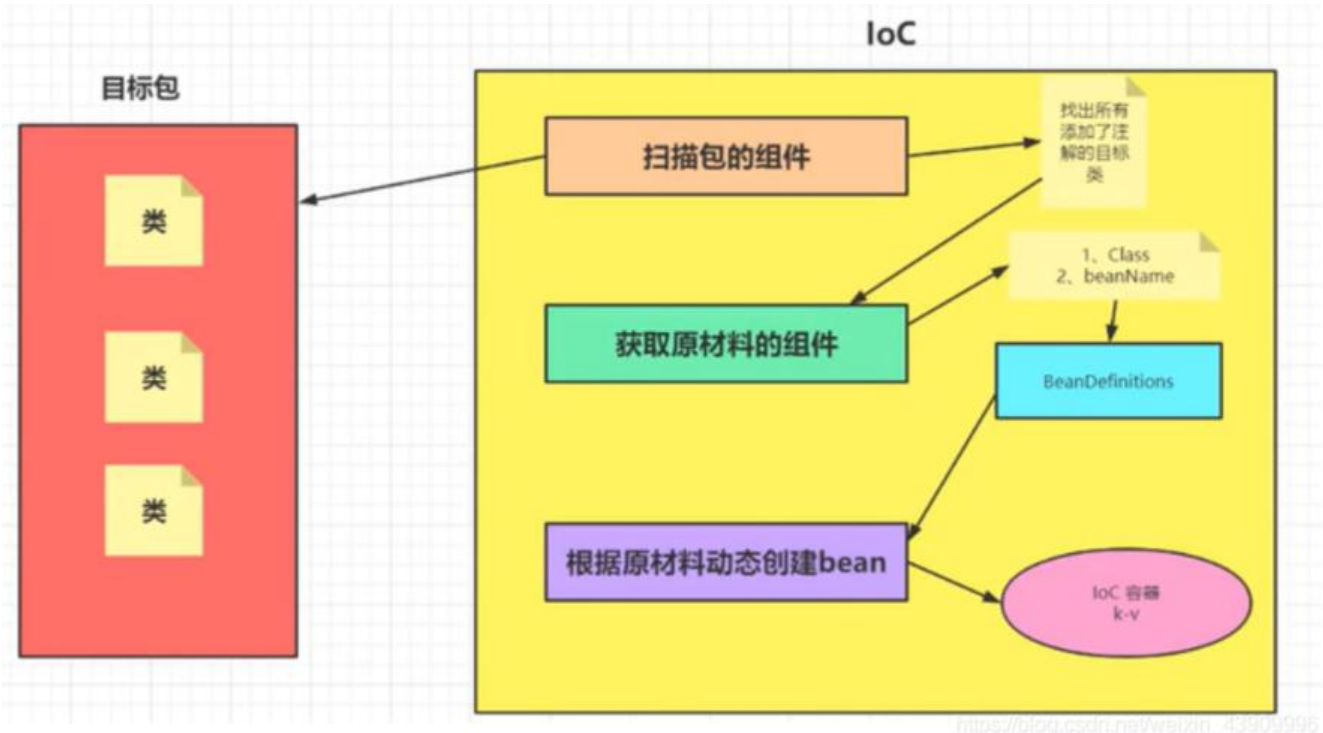
控制反转 依赖注入

控制反转：本来是我们手动new出的对象，现在交给Spring的IOC容器管理（设计）

依赖注入：对象无需创建和管理它的依赖关系，依赖关系将被自动注入到需要对象当中去（实现）

主要好处：**将对象集中统一管理 降低耦合度**

ioc工作原理



主要分为几个步骤

1. 扫描包组件

2. 获得原材料组件
3. 获得BeanDefination集合
4. 根据原材料动态创建Bean
5. 实现Bean的自动装入

实现ioc

新建一个maven项目，因为自己实现Spring ioc，所以什么依赖都不用导入

我们通过注解注入

先创建注解@Component

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Component {

    String value() default "";
}
```

创建注解@Autowired

```
@Target({ElementType.TYPE,ElementType.FIELD,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Autowired {

}
```

由于@Autowired是通过类型注入的，但依赖注入也可能通过name注入，所以创建注解@Qualifier

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Qualifier {
    String value() default "";
}
```

创建@Value注解为字段赋值

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Value {
    String value();
}
```

我们创建两个实体类作为bean

```
@Component("teacher") //声明注解，作为Bean
public class Teacher {
    @Value("001")
    private Integer teacherId;

    @Value("Tom")
```

```
    private String teacherName;

    @Autowired  //自动装配user bean
    private User user;

    public Integer getTeacherId() {
        return teacherId;
    }

    public void setTeacherId(Integer teacherId) {
        this.teacherId = teacherId;
    }

    public String getTeacherName() {
        return teacherName;
    }

    public void setTeacherName(String teacherName) {
        this.teacherName = teacherName;
    }

    @Override
    public String toString() {
        return "Teacher{" +
                "teacherId=" + teacherId +
                ", teacherName='" + teacherName + '\'' +
                ", user=" + user +
                '}';
    }
}

@Component("user")
public class User {

    @Value("001")
    private Integer id;
    @Value("jack")
    private String name;
    @Value("20")
    private Integer age;


    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

创建BeanDefinition类，作为原材料类

```
public class BeanDefinition {

    private String BeanName; //Bean的名字
    private Class BeanClass; //Bean的class

    public String getBeanName() {
        return BeanName;
    }

    public void setBeanName(String beanName) {
        BeanName = beanName;
    }

    public Class getBeanClass() {
        return BeanClass;
    }

    public void setBeanClass(Class beanClass) {
        BeanClass = beanClass;
    }

    public BeanDefinition(String beanName, Class beanClass) {
        BeanName = beanName;
        BeanClass = beanClass;
    }

    public BeanDefinition() {
    }

    @Override
```

```
    public String toString() {
        return "BeanDefinition{" +
                "BeanName='" + BeanName + '\'' +
                ", BeanClass=" + BeanClass +
                '}';
    }
}
```

创建工具类MyTools用于从扫描路径下的所有class(直接复制就行)

```
public class MyTools {

    public static Set<Class<?>> getClasses(String pack) {

        // 第一个class类的集合
        Set<Class<?>> classes = new LinkedHashSet<Class<?>>();
        // 是否循环迭代
        boolean recursive = true;
        // 获取包的名字 并进行替换
        String packageName = pack;
        String packageDirName = packageName.replace('.', '/');
        // 定义一个枚举的集合 并进行循环来处理这个目录下的things
        Enumeration<URL> dirs;
        try {
            dirs = Thread.currentThread().getContextClassLoader().getResources(packageDirName
;
            // 循环迭代下去
            while (dirs.hasMoreElements()) {
                // 获取下一个元素
                URL url = dirs.nextElement();
                // 得到协议的名称
                String protocol = url.getProtocol();
                // 如果是以文件的形式保存在服务器上
                if ("file".equals(protocol)) {
                    // 获取包的物理路径
                    String filePath = URLDecoder.decode(url.getFile(), "UTF-8");
                    // 以文件的方式扫描整个包下的文件 并添加到集合中
                    findClassesInPackageByFile(packageName, filePath, recursive, classes);
                } else if ("jar".equals(protocol)) {
                    // 如果是jar包文件
                    // 定义一个JarFile
                    System.out.println("jar类型的扫描");
                    JarFile jar;
                    try {
                        // 获取jar
                        jar = ((JarURLConnection) url.openConnection()).getJarFile();
                        // 从此jar包 得到一个枚举类
                        Enumeration<JarEntry> entries = jar.entries();
                        findClassesInPackageByJar(packageName, entries, packageDirName, recursive,
lasses);
                    } catch (IOException e) {
                        // log.error("在扫描用户定义视图时从jar包获取文件出错");
                        e.printStackTrace();
                    }
```

```
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return classes;
    }

    private static void findClassesInPackageByJar(String packageName, Enumeration<JarEntry>
entries, String packageDirName, final boolean recursive, Set<Class<?>> classes) {
        // 同样的进行循环迭代
        while (entries.hasMoreElements()) {
            // 获取jar里的一个实体 可以是目录 和一些jar包里的其他文件 如META-INF等文件
            JarEntry entry = entries.nextElement();
            String name = entry.getName();
            // 如果是以/开头的
            if (name.charAt(0) == '/') {
                // 获取后面的字符串
                name = name.substring(1);
            }
            // 如果前半部分和定义的包名相同
            if (name.startsWith(packageDirName)) {
                int idx = name.lastIndexOf('/');
                // 如果以"/"结尾 是一个包
                if (idx != -1) {
                    // 获取包名 把"/"替换成"."
                    packageName = name.substring(0, idx).replace('/', '.');
                }
                // 如果可以迭代下去 并且是一个包
                if ((idx != -1) || recursive) {
                    // 如果是一个.class文件 而且不是目录
                    if (name.endsWith(".class") && !entry.isDirectory()) {
                        // 去掉后面的".class" 获取真正的类名
                        String className = name.substring(packageName.length() + 1, name.length() -
6);
                        try {
                            // 添加到classes
                            classes.add(Class.forName(packageName + '.' + className));
                        } catch (ClassNotFoundException e) {
                            // .error("添加用户自定义视图类错误 找不到此类的.class文件");
                            e.printStackTrace();
                        }
                    }
                }
            }
        }
    }

    private static void findClassesInPackageByFile(String packageName, String packagePath, fi
al boolean recursive, Set<Class<?>> classes) {
        // 获取此包的目录 建立一个File
        File dir = new File(packagePath);
        // 如果不存在或者 也不是目录就直接返回
        if (!dir.exists() || !dir.isDirectory()) {
```

```
            // log.warn("用户定义包名 " + packageName + " 下没有任何文件");
            return;
        }
        // 如果存在 就获取包下的所有文件 包括目录
        File[] dirfiles = dir.listFiles(new FileFilter() {
            // 自定义过滤规则 如果可以循环(包含子目录) 或则是以.class结尾的文件(编译好的java类文件)
            @Override
            public boolean accept(File file) {
                return (recursive && file.isDirectory()) || (file.getName().endsWith(".class"));
            }
        });
        // 循环所有文件
        for (File file : dirfiles) {
            // 如果是目录 则继续扫描
            if (file.isDirectory()) {
                findClassesInPackageByFile(packageName + "." + file.getName(), file.getAbsolutePa
h(), recursive, classes);
            } else {
                // 如果是java类文件 去掉后面的.class 只留下类名
                String className = file.getName().substring(0, file.getName().length() - 6);
                try {
                    classes.add(Thread.currentThread().getContextClassLoader().loadClass(packageN
me + '.' + className));
                } catch (ClassNotFoundException e) {
                    // log.error("添加用户自定义视图类错误 找不到此类的.class文件");
                    e.printStackTrace();
                }
            }
        }
    }

}
```

创建MyAnnotationConfigContext类用于依赖注入

```
public class MyAnnotationConfigApplicationContext {

    public Map<String,Object> ioc = new HashMap<>();  //ioc容器

    public MyAnnotationConfigApplicationContext(String pack) throws InvocationTargetExcept
on, NoSuchMethodException, InstantiationException, IllegalAccessException {
        // 1. 第一步先获取原材料BeanDefinition集合
        Set<BeanDefinition> beanDefinitions = getBeanDefinitions(pack);
        // 2. 创建Bean
        createBean(beanDefinitions);
        // 3. 实现Bean的自动装配
        autowiredBean(beanDefinitions);
    }

    public Set<BeanDefinition> getBeanDefinitions(String pack){
        Set<Class<?>> classes = MyTools.getClasses(pack);
        Set<BeanDefinition> beanDefinitions = new HashSet<>();
        for (Class<?> aClass : classes) {
            //判断是否被@Component注解声明
```

```
            String annotationName = aClass.getAnnotation(Component.class).value();
            String packName = aClass.getName();
            String className = packName.substring(packName.lastIndexOf(".")+1);
            String beanName = "";
            if("".equals(annotationName)){
                //如果未赋初值，则用类名作为beanName(第一个字母大写改为小写)
                beanName = className.substring(0,1).toLowerCase()+className.substring(1);
            }else{
                beanName = annotationName;
            }
            beanDefinitions.add(new BeanDefinition(beanName,aClass));
        }
        return beanDefinitions;
    }

    public void createBean(Set<BeanDefinition> beanDefinitions) throws NoSuchMethodExcep
ion, InvocationTargetException, InstantiationException, IllegalAccessException {
        for (BeanDefinition beanDefinition : beanDefinitions) {
            Class clazz = beanDefinition.getBeanClass();
            String beanName = beanDefinition.getBeanName();
            Object object = clazz.getConstructor().newInstance();
            //通过反射获取所有字段属性
            Field[] declaredFields = clazz.getDeclaredFields();
            for (Field declaredField : declaredFields) {
                Value Annotation = declaredField.getAnnotation(Value.class);
                if(Annotation != null){
                    String value = Annotation.value();
                    String fieldName = declaredField.getName();
                    // 通过setter方法注入的，也可以通过反射注入
                    String methodName = "set"+fieldName.substring(0,1).toUpperCase()+fieldName.
ubstring(1);
                    Method method = clazz.getMethod(methodName,declaredField.getType());
                    //由于数据类型可能不一致
                    //完成数据类型转换
                    Object val = null;
                    switch (declaredField.getType().getName()){
                        case "java.lang.Integer":
                            val = Integer.parseInt(value);
                            break;
                        case "java.lang.String":
                            val = value;
                            break;
                        case "java.lang.Float":
                            val = Float.parseFloat(value);
                            break;
                    }
                    method.invoke(object, val);
                }
            }
            ioc.put(beanName,object);
        }
    }

    public Object getBean(String beanName){
```

```
        //从ioc容器中通过name取bean
        return ioc.get(beanName);
    }

    public void autowiredBean(Set<BeanDefinition> beanDefinitions) throws IllegalAccessExce
tion, NoSuchMethodException, InvocationTargetException, InstantiationException {
        for (BeanDefinition beanDefinition : beanDefinitions) {
            Class clazz = beanDefinition.getBeanClass();
            Field[] declaredFields = clazz.getDeclaredFields();
            for (Field declaredField : declaredFields) {
                //遍历所有字段
                //是否需要自动注入
                Autowired annotation = declaredField.getAnnotation(Autowired.class);
                if (annotation != null) {
                    Qualifier qualifier = declaredField.getAnnotation(Qualifier.class);
                    Object object = getBean(beanDefinition.getBeanName());
                    declaredField.setAccessible(true);
                    if (qualifier != null) {
                        // 通过Name
                        String beanName = qualifier.value();
                        Object bean = getBean(beanName);
                        // 通过反射，直接对成员变量赋值
                        declaredField.set(object, bean);
                    }else{
                        // 通过类型
                        for (String beanName : ioc.keySet()) {
                            if (ioc.get(beanName).getClass() == declaredField.getType()) {
                                declaredField.set(object, ioc.get(beanName));
                            }
                        }
                    }
                }
            }
        }
    }
}
```

一个简单的ioc就创建完成，创建一个test类进行测试

```
public class Test {

    public static void main(String[] args) throws InvocationTargetException, NoSuchMethodExc
ption, InstantiationException, IllegalAccessException {
        MyAnnotationConfigApplicationContext context = new MyAnnotationConfigApplication
ontext("com.fjiang.entity");//路径
        System.out.println(context.getBean("teacher"));//获取Bean
    }
}
```

结果如下，说明创建成功！

写在最后，由于这是简易版的ioc，如若user也依赖了teacher，则会出现循环依赖的问题，该版本暂无法解决

那Spring框架是如何解决的呢?
大致过程：首先A对象实例化，然后对属性进行注入，发现依赖B对象，B对象此时还没创建出来，所转头去实例化B对象，B对象实例化以后，
发现需要依赖A对象，那A对象已经实例化了，所以B对象最终能完成创建，B对象返回A对象的属性注方法上，A对象最终完成创建。

原理：利用了三级缓存

* singletonObjects(一级缓存，日常获取Bean的地方)
* earlySingletonObjects(二级缓存，已实例化，但还没有进行属性注入，由三级缓存放进来)
* singletonFactories(三级缓存，Value是一个对象工厂)

A对象实例化后，在属性注入之前，其实会把A对象放到三级缓存中,key是BeanName,Value是ObjectFactory，等到A对象属性注入时，发现依赖B，
又去实例化B，B属性注入需要A对象，这里就是从三级缓存中拿出ObjectFactory，从ObjectFactory到对应的Bean（对象A），
然后将三级缓存中的A删除，放到二级缓存中。显然，二级缓存存储的key是BeanName,value就是Bean(这里的Bean还没有完成属性注入的相关工作)，
等到完全初始化之后，就会把二级缓存中的remove掉，放入一级缓存中，然后我们getBean的时候实际拿到的是一级缓存的。