IOC 的具体实现

该 IOC 容器采用依赖注册和依赖提供分离的方式，我们可以将程序中需要的依赖在程序运行一开始注入 IOC 容器，在程序运行时动态的获取依赖实例。

这里我们通过**建造者模式**，首先配置 IServiceCollection，向建造者添加依赖描述符，然后 Build()，生成依赖提供者(IServiceProvider)。

1、依赖注册器(IServiceCollection)

既然需要从 IOC 容器中拿去依赖示例，我们就需要先将需要用到的依赖注册到 IOC 容器中。

当我们将依赖注册到 ServiceCollection 时，ServiceCollection 会将注册信息转换成服务描述符(ServiceDescriptor)，存到服务描述符数组中。

```
public class ServiceCollection implements IServiceCollection {
    private List<ServiceDescriptor> serviceCollection = new ArrayList<>();
    // ···
}
```

下面是 IServiceCollection 的依赖注册函数。

```
private IServiceCollection Register(Class<?> clazz, Class<?> iClazz, ServiceLiftTime
liftTime){
        if (iClazz != null)
            clazzList.add(iClazz);
        clazzList.add(clazz);
        serviceCollection.add(new ServiceDescriptor(clazz, iClazz, liftTime));
        return this;
    }
```

这里 IOC 容器并没有实例化依赖，只是将依赖的 Class 类存起来。依赖的实例化将由 IServiceProvider 处理。

2、依赖提供者

ServiceProvider 内置了 依赖存储器工厂(AccessorFactor) , 为依赖提供者生产对应依赖存储器。 当调用 GetService()函数后，ServiceProvider 会从依赖存储器工厂中拿取对应 Class 的依赖存储器，然后通过依赖存储器获取对应的实例。

```
public <T> T GetService(Class<T> clazz) throws Exception {
    if (clazz == IServiceProvider.class)
        return (T) this;

    if (serviceCollection.containsKey(clazz)) {
        var res = serviceCollection.get(clazz);
        res.SetServiceProvider(this);
        return (T) res.Resolve();
    }

    IServiceAccessor accessor = accessorFactory.CreateAccessor(clazz);
    serviceCollection.put(clazz, accessor);
    accessor.SetServiceProvider(this);
    return (T) accessor.Resolve();
}
```

3、依赖存储器

依赖存储器(ServiceAccessor)是 IOC 容器自动装配功能的核心。 在调用依赖存储器的 Resole()获取依赖实例时，使用了 Java 的反射机制，动态的为依赖实例添加它的依赖。

```
protected Object CreateObject() throws Exception {
    Constructor<?> selectConstractor = null;
    Constructor<?>[] constructors = clazz.getConstructors();
    for (Constructor<?> constructor : constructors) {
        // 选择类中最适配的构造器
    }

    Class<?>[] clazzs = selectConstractor.getParameterTypes();
    Object[] parameters = new Object[selectConstractor.getParameterCount()];
    for (int i = 0; i < selectConstractor.getParameterCount(); i++) {
        // 获取选中的构造器的参数列表的依赖
    }
    // 通过构造器创建实例
    Object result = selectConstractor.newInstance(parameters);

    // 注解注入
    Field[] fields = clazz.getDeclaredFields();
    for (Field field : fields) {
        // 将类中所有标记了@Dependence的属性添加依赖
    }
    return result;
}
```

4、依赖的生命周期控制

该 IOC 容器提供了三种生命周期注册(Transient、Scope、Singleton)。 为了实现依赖的不同生命周期，我们为三种生命周期编写了对应的 ServiceAccessor。

- 对于瞬态类型的服务，每次请求都返回新的对象。

```
public class TransientServiceAccessor extends ServiceAccessorBase {
      @Override
      protected Object ResolveObject() throws Exception {
          return CreateObject();
      }
}
```

- 对于单例类型的服务，我们会将单例实例寄存在单例存储器中，确保多次向单例存储器请求实例时，返回同一个对象。

```
public class SingletonServiceAccessor extends ServiceAccessorBase {
      protected Object object = null;

      @Override
      protected Object ResolveObject() throws Exception {
          if (object != null)
              return object;
          object = CreateObject();
          return object;
      }
}
```