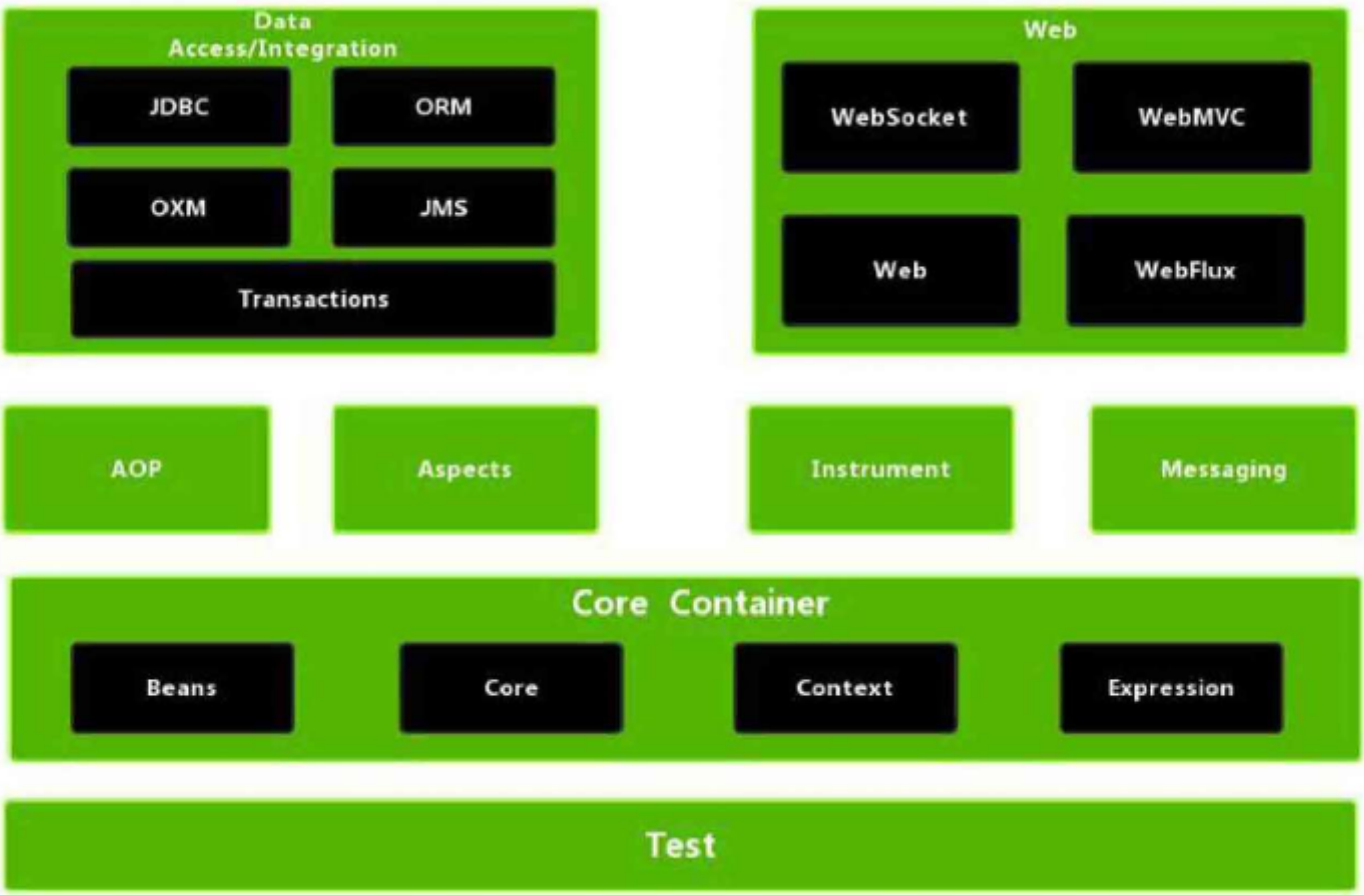
Spring的组成如图所示



这篇笔记主要记录核心容器(Core container)和 Aop切面编程，其中核心容器主要提供依赖注入和控制反转功能

依赖注入和控制反转 (DI AND IOC)

- ## 使用流程

- 新建maven项目，添加Spring核心容器依赖 Spring-core Spring-context spring-bean

```
<!--
https://mvnrepository.com/artifact/org.springframework/spring-core
 -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
```

```
    <version>5.3.20</version>
</dependency>

<!--
https://mvnrepository.com/artifact/org.springframework/spring-context
 -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.20</version>
</dependency>
```

- 编写组件类

```
public interface IAir {
}

@Component("cleanair")
public class CleanAir implements IAir {
    @Override
    public String toString() {
        return "clean air";
    }
}

@Component("dirtyair")
public class DirtyAir implements IAir {
    @Override
    public String toString() {
        return "dirty air";
    }
}

@Component
public class Person {

    public IAir air;
    @Autowired
    public Person(@Qualifier("dirtyair") IAir air){
```

```
        this.air=air;
    }
    public void breath(){
        System.out.println(this.air);
    }
}
```

- 添加注解进容器

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="
http://www.springframework.org/schema/beans
"
       xmlns:context="
http://www.springframework.org/schema/context
"
       xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance
"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd


http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd
">
<!--    注解扫描-->
    <context:component-scan base-package="com.hth.components"/>

</beans>
```

- 测试运行

```
public class Main{
    public static void main(String[] args) throws Throwable{
        ApplicationContext context = new ClassPathXmlApplicationContext(new String[]
{"applicationContext.xml"});
        BeanFactory factory = context; //获取组件容器
        Person p = (Person) factory.getBean(Person.class);
```

```
6           p.breath();
7       }
8   }
```

IOC容器

Spring 提供了两种IOC容器 BeanFactory ApplicationContext

- **BeanFactory**

BeanFactory 会在bean的各个生命周期都对其进行管理，只要我们的bean实现了各个生命周期阶段对应的接口，那么Spring就会在对应的阶段调用接口方法处理bean。

- bean容器的启动

bean容器启动首先会加载配置文件中所有的bean,转化为BeanDefinition(就是描述bean配置的一个类)

如图所示，字段啥意思顾名思义吧，这个了解一下就行

```
public abstract class AbstractBeanDefinition extends
BeanMetadataAttributeAccessor
implements BeanDefinition, Cloneable {
private volatile Object beanClass;
private String scope = SCOPE_DEFAULT;
private boolean abstractFlag = false;
private boolean lazyInit = false;
private int autowireMode = AUTOWIRE_NO;
private int dependencyCheck = DEPENDENCY_CHECK_NONE;
private String[] dependsOn;
private ConstructorArgumentValues constructorArgumentValues;
private MutablePropertyValues propertyValues;
private String factoryBeanName;
private String factoryMethodName;
private String initMethodName;
private String destroyMethodName;
```

- 通过BeanDefinitionRegistry将bean注册到beanFactory中。（没啥软用）

BeanFactory实现类实现了BeanDefinitionRegistry接口的方法，该接口提供了一个方法通过beanName注册对应的BeanDefinition 到一个 Map中

- 容器实例化bean

在实例化过程中，Spring暴露了如下接口，bean对象可实现这些接口

- **ApplicationContext**

在上面的使用流程中，使用ApplicationContext获取了bean的配置，然后直接将ApplicationContext接口对象赋值给了BeanFactory接口对象，为什么可以赋值呢？其实ApplicationContext接口实现了BeanFactory接口。

```
1 public class Main{
2     public static void main(String[] args) throws Throwable{
3         ApplicationContext context = new ClassPathXmlApplicationContext(new String[]
  {"applicationContext.xml"});
4         BeanFactory factory = context; //获取组件容器
5         Person p = (Person) factory.getBean(Person.class);
6         p.breath();
7     }
8 }
```

ApplicationContext实现的接口

```
public interface ApplicationContext extends EnvironmentCapable,
 ListableBeanFactory, HierarchicalBeanFactory,MessageSource,
 ApplicationEventPublisher, ResourcePatternResolver
```

它通过实现其他接口，扩展了BeanFactory接口的功能

● MessageSource：为应用提供国际化访问功能。

● ResourcePattern Resolver：提供资源（如URL和文件系统）的访问支持。

● ApplicationEventPublisher：引入事件机制，包括启动事件、关闭事件等，让容器在上下文中提供了对应用事件的支持。它代表的是一个应用的上下文环境。

ApplicationContext可通过ClassPathXmlApplicationContext 或者FileSystemXmlApplicationContext 对象获取

Bean的注册配置

- **基于XML的bean配置**

在配置文件中编写bean的信息，例如，这是一个最简单的例子

```
<bean id="xx" class="xx"/>
```

- **基于注解的bean配置**

例如上面的@Component注解，添加到Bean类上，并在xml配置文件中指出其所在的包进行扫描

```
<context:component-scan base-package="com.hth.components"/>
```

除了Component，还有另外三个配置Bean的注解

- @Repository：用于对DAO实现类进行标注。
- @Service：用于对Service实现类进行标注。
- @Controller：用于对Controller实现类进行标注。

那问题来了：既然都是bean，为什么还提供4个呢？因为这样能让注解的用途更加清晰，而且不同的注解也有各自特殊的功能。

- **基于java类的bean配置**

在普通的java类上 添加@Configuration注解，在添加了该注解的对象中的类方法标注 @Bean，就相当于该方法定义了一个bean

添加了Configuration注解的对象可以用xml的<bean>标签进行注册，也可使用包扫描<context:component-scan>

```
@Configuration
public class Configurations {
    @Bean
    public Computer computer(){
        return new Computer();
    }
}


public class Main{
    public static void main(String[] args) throws Throwable{
```

```
        ApplicationContext context = new ClassPathXmlApplicationContext(new String[]{"applicationContext.xml"});
        BeanFactory factory = context; //获取组件容器
        Computer p = (Computer) factory.getBean(Computer.class);
        p.switchOn();
    }
}
```

Bean的注入

Bean注入一般分为XMl注入和注解注入

- **XML注入**

XML注入一般有三种方式：属性注入，构造方法注入，工厂方法注入

1. 属性注入

属性注入的情况下，Spring会调用需要注入bean的无参构造函数，然后通过Setter方法进行属性注入，所以需要定义好setter方法

以下新建一个XmlInstance类进行测试，IAir沿用了之前的代码

```
package com.hth.components;

import com.hth.interfaces.IAir;

public class XmlInstance {
    public String name;
    public IAir air;

    public XmlInstance() {
    }

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

    public void setAir(IAir air) {
        this.air = air;
    }

    @Override
```

```
    public String toString() {
        return "XmlInstance{" +
                "name='" + name + '\'' +
                ", air=" + air +
                '}';
    }
}

@Component("cleanair")
public class CleanAir implements IAir {
    @Override
    public String toString() {
        return "clean air";
    }
}

```

在配置文件中注册bean

```
<bean id="xmlinstance" class="com.hth.components.XmlInstance">
    <property name="air" ref="cleanair"/>
    <property name="name" value="hahah"/>
</bean>
```

启动测试

```
public class Main{
    public static void main(String[] args) throws Throwable{
        ApplicationContext context = new ClassPathXmlApplicationContext(new String[]{"applicationContext.xml"});
        BeanFactory factory = context; //获取组件容器
        XmlInstance p = (XmlInstance) factory.getBean("xmlinstance");
        System.out.println(p);
        System.out.println(p.air);
    }
}

输出为:
XmlInstance{name='hahah', air=clean air}
clean air
```

2. 构造方法注入

为XmlInstance类定义一个构造函数如下

```
public XmlInstance(String name, IAir air) {
    this.name = name;
    this.air = air;
}
```

修改xml配置文件

```
<bean id="xmlinstance" class="com.hth.components.XmlInstance">
    <constructor-arg name="air" index="1" ref="cleanair"/>
    <constructor-arg name="name" index="0" value="6666"/>
</bean>
```

运行发现，测试结果相同

3. 工厂方法注入

略，没什么卵用

Tips:<Properties>有一些特殊的属性也有对应的标签方便注入，例如<list> <set> <map>等等

```
<bean id="xmlinstance" class="com.hth.components.XmlInstance" init-method="xxx" destroy-method="xxx"/>
```

有时候实例执行后构造函数之后还需要执行某些初始化逻辑，或者被gc回收前需要执行某些释放资源的代码，可以用如上的配置指定需要执行的方法。

- **注解注入**

常用的注解主要有五种: @Autowired、@Resource、@Required、@Qualifier、@Value

- Autowired:可以按照类型匹配自动注入bean，可以添加在类成员，类方法以及构造函数上面，此注解最好是用在单实例的情况下，如果在一接口多实现类的情况下使用，可能会造成异常。
- Qualifier : 正好弥补Autowired的不足，当在一接口多实现类的情况下，Qualifier可以指定某一个实现类
- Value； 相当于<property>标签的value属性
- Resouce：按照bean注册的名字进行查找，如果不提供名字，则默认按照标志处的变量名或者方法名查找，如果这也找不到。那么它就会出现和Autowired一样的行为按照类型查找
- Required：适用于bean属性setter方法，就是它要保住在setXX()方法上，并且被setXX的XX属性在xml配置文件里写了，如上一节的xml注入的属性注入，个人这个注解多此一举，它的使用需要配合xml配置文件

个人理解: Resouce注解配置 bean注册时表明id ， 基本够平时开发用了

Bean的作用域

作　用　域	描　　述
单例（singleton）	默认，每一个 Spring IoC 容器都拥有唯一的一个实例对象
原型（prototype）	一个 Bean 定义，任意多个对象
请求（request）	一个 HTTP 请求会产生一个 Bean 对象，也就是说，每一个 HTTP 请求都有自己的 Bean 实例，只在基于 Web 的 Spring ApplicationContext 中可用
会话（session）	限定一个 Bean 的作用域为 HTTPSession 的生命周期。同样，只有基于 Web 的 Spring ApplicationContext 才能使用
全局会话（global session）	限定一个 Bean 的作用域为全局 HTTPSession 的生命周期。通常用于门户网站场景，同样，只有基于 Web 的 Spring ApplicationContext 可用

Aop面向切面编程

Aop切面编程基础

aop切面编程一般通过代理实现，代理又分为静态代理，和动态代理

静态代理

静态代理的关键是代理类和目标类实现了相同的接口，并且在静态代理类的内部持有目标类的引用

动态代理

在java中，动态代理需要用到的InvocationHandler接口和Proxy类，其他的不说了，想实现java的动态代理再去了解这两个东西即可

Spring-AOP实现

spring aop的三种实现方式：基于代理的aop，AspectJ基于xml的配置，AspectJ基于注解的配置

- **基于代理的aop**

基于代理的aop主要介绍三个接口的使用：methodbeforeAdvice，afterReturningAdvice，ThrowsAdvice

特别注意ThrowsAdvice这个接口：这个接口在目标方法抛出异常时被执行，但该接口未定义任何方法，但是在实现该接口的类中定义的public void afterThrowing(Exception ex)、public void afterThrowing(Method method, Object[] args, Object target, Exception ex)的方法会在异常抛出时被调用

测试样例如下，

首先引入Spring-aop的依赖

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.3.22</version>
</dependency>
```

定义要被代理的类以及其接口

```java
public interface IAopServ {
    String methodNoAop();
    String methodAop();
}

@Component("aopServ")
public class AopServ implements IAopServ{
    @Value("123456")
    public String name;

    public String getName() {
        return name;
    }

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

    @Override
    public String methodNoAop() {
        System.out.println("mei有AOP的方法");
        return name;
    }

    @Override
    public String methodAop() {
        System.out.println("有Aop的方法");
        return name;
    }
```

```
}
```

定义AOP拦截器类，实现上面所说的三个接口

```
public class AOPInterceptor implements MethodBeforeAdvice, AfterReturningAdvice, ThrowsAdvice {
    @Override
    public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable {
        System.out.println(method.getName()+"的返回值为:  "+returnValue);
    }

    @Override
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println("执行MethodBeforeAdvice "+"执行的方法为:  "+method.getName());
    }

    public void afterThrowing(Exception ex){
        System.out.println("抛出了异常: "+ex.getMessage());
    }

    public void afterThrowing(Method method,Object[] args,Object target,Exception ex){
        System.out.println(method.getName()+"抛出了异常: "+ex.getMessage());
    }
}
```

编写配置文件

代理类和被代理类无法结合起来，因为他们没有持有对方的引用，Spring无法通过bean注入实现结合，这里需要用到两个特殊的bean：NameMatchMethodPointcutAdvisor 和 ProxyFactoryBean
首先将拦截器类注入到NameMatchMethodPointcutAdvisor类中，再将NameMatchMethodPointcutAdvisor注入到ProxyFactoryBean中

```
<bean id="interceptor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="advice" >
        <bean class="com.hth.aop.AOPInterceptor"/>
    </property>
    <property name="mappedName" value="methodAop"/>
</bean>
```

```xml
<bean id="proxyFac" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="interceptorNames">
        <list>
            <value>interceptor</value>
        </list>
    </property>
    <property name="target" ref="aopServ">
    </property>
</bean>
```

- **AspectJ基于xml的配置**

AspectJ是一个专门用于Aop的切面编程框架，需要引入aopalliance和aspectjweaver这两个框架

```xml
<dependency>
    <groupId>aopalliance</groupId>
    <artifactId>aopalliance</artifactId>
    <version>1.0</version>
</dependency>
<!--
https://mvnrepository.com/artifact/org.aspectj/aspectjweaver
 -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.9.1</version>
    <scope>runtime</scope>
</dependency>
```

AspectJ主要提供了下列的配置元素

AOP 配置元素	描　　述
<aop:config>	顶层的 AOP 配置元素，大多数的<aop:*>元素必须包含在<aop:config>元素内
<aop:aspect>	定义切面
<aop:aspect-autoproxy>	启用@AspectJ 注解驱动的切面
<aop:pointcut>	定义切点
<aop:advisor>	定义 AOP 通知器
<aop:before>	定义 AOP 前置通知
<aop:after>	定义 AOP 后置通知（不管被通知的方法是否执行成功）
<aop:after-returning>	定义成功返回后的通知
<aop:after-throwing>	定义抛出异常后的通知
<aop:around>	定义 AOP 环绕通知
<aop:declare-parents>	为被通知的对象引入额外的接口，并透明地实现

示例如下，被代理类沿用上一节的类
首先定义拦截器类

```
@Component("aaAspectJ")
public class AspectJ {
    public void beforeMethod(){
        System.out.println("前置方法执行了");
    }
    public void afterMethod(){
        System.out.println("后置方法执行了");
    }
    public void afterReturning(Object obj){
        System.out.println("方法返回结果 = " + obj);
    }

    public String aroundMethod(ProceedingJoinPoint joinPoint)throws Throwable{
        String result="";
        try{
            System.out.println("前置通知");
            result = (String)joinPoint.proceed();
            System.out.println("后置通知");
        }catch (Exception ex){
            throw ex;
        }
```

```
        return result;
    }

}
```

编写配置文件

```
<aop:config>
    <aop:aspect id="aspect" ref="aaAspectJ">
        <aop:pointcut id="aspect" expression="execution(*
com.hth.aop.AopServ.methodAop(..))"/>
        <aop:before method="beforeMethod" pointcut-ref="aspect"/>
        <aop:after method="afterMethod" pointcut-ref="aspect"/>
        <aop:after-returning method="afterReturning" pointcut-ref="aspect"
returning="obj"/>
        <aop:around method="aroundMethod" pointcut-ref="aspect"/>
    </aop:aspect>
</aop:config>
```

测试

```
public class Main{
    public static void main(String[] args) throws Throwable{
        ApplicationContext context = new ClassPathXmlApplicationContext(new String[]
{"applicationContext.xml"});
        BeanFactory factory = context; //获取组件容器
        IAopServ p = (IAopServ) factory.getBean("aopServ");
        p.methodAop();
        p.methodNoAop();


    }
}
sout:
前置方法执行了
前置通知
有Aop的方法
后置通知
方法返回结果 = 123456
后置方法执行了
```

- **AspectJ基于注解的配置**

添加@Aspect注解在代理方法类上面。再各自定义切面方法

然后在配置文件中开启aop包扫描

```
@Component("aaAspectJ")
@Aspect
public class AspectJ {
    @Before("execution(* com.hth.aop.AopServ.methodAop(..))")
    public void beforeMethod(){
        System.out.println("前置方法执行了");
    }
    @After("execution(* com.hth.aop.AopServ.methodAop(..))")
    public void afterMethod(){
        System.out.println("后置方法执行了");
    }
    @AfterReturning(value = "execution(* com.hth.aop.AopServ.methodAop(..))",returning = "obj")
    public void afterReturning(Object obj){
        System.out.println("方法返回结果 = " + obj);
    }

    @Around("execution(* com.hth.aop.AopServ.methodAop(..))")
    public String aroundMethod(ProceedingJoinPoint joinPoint)throws Throwable{
        String result="";
        try{
            System.out.println("前置通知");
            result = (String)joinPoint.proceed();
            System.out.println("后置通知");
        }catch (Exception ex){
            throw ex;
        }
        return result;
    }
}

```

```
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```