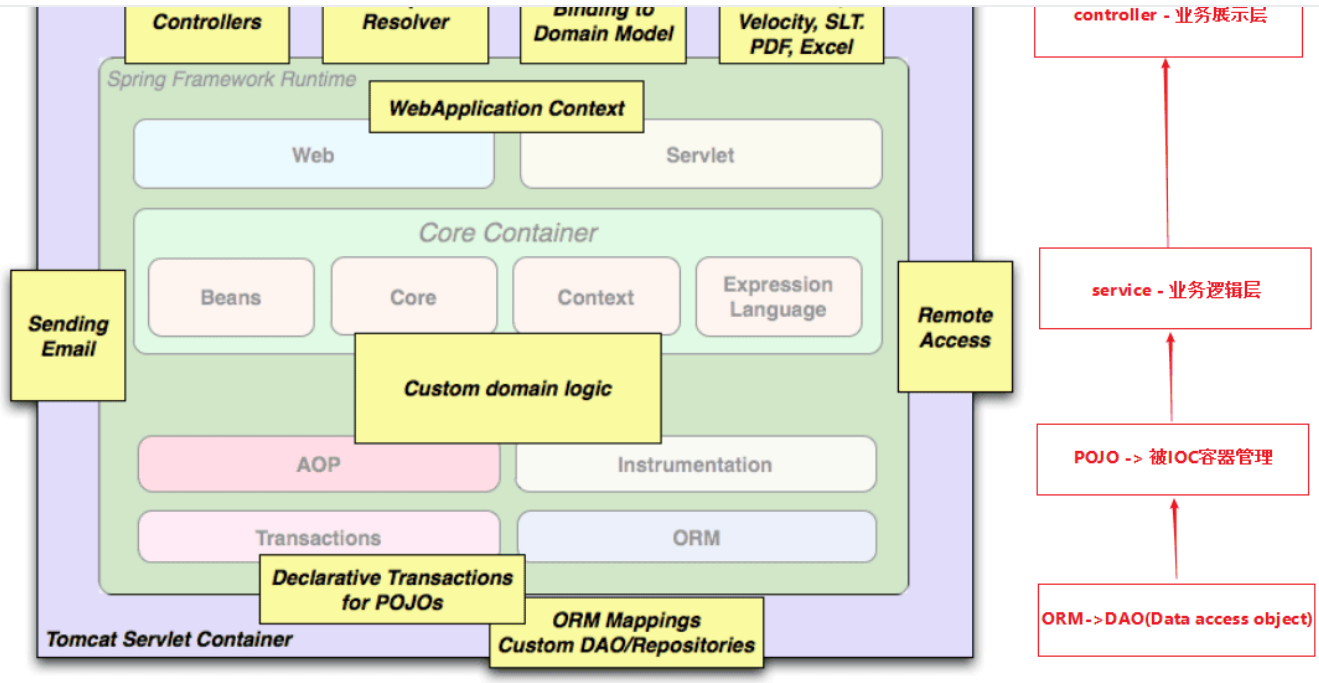
Spring基础 - Spring简单例子引入Spring要点

> 上文中我们简单介绍了Spring和Spring Framework的组件，那么这些Spring Framework组件是如何配合工作的呢？本文主要承接上文，向你展示Spring Framework组件的典型应用场景和基于这个场景设计出的简单案例，并以此引出Spring的核心要点，比如IOC和AOP等；在此基础上还引入了不同的配置方式， 如XML，Java配置和注解方式的差异。@pdai

Spring框架如何应用

> 上文中，我们展示了Spring和Spring Framework的组件, 这里对于开发者来说有几个问题：
>
> 1. 首先，对于Spring进阶，直接去看IOC和AOP，存在一个断层，所以需要整体上构建对Spring框架认知上进一步深入，这样才能构建知识体系。
> 2. 其次，很多开发者入门都是从Spring Boot开始的，他对Spring整体框架底层，以及发展历史不是很了解； 特别是对于一些老旧项目维护和底层bug分析没有全局观。
> 3. 再者，Spring代表的是一种框架设计理念，需要全局上理解Spring Framework组件是如何配合工作的，需要理解它设计的初衷和未来趋势。

如下是官方在解释Spring框架的常用场景的图

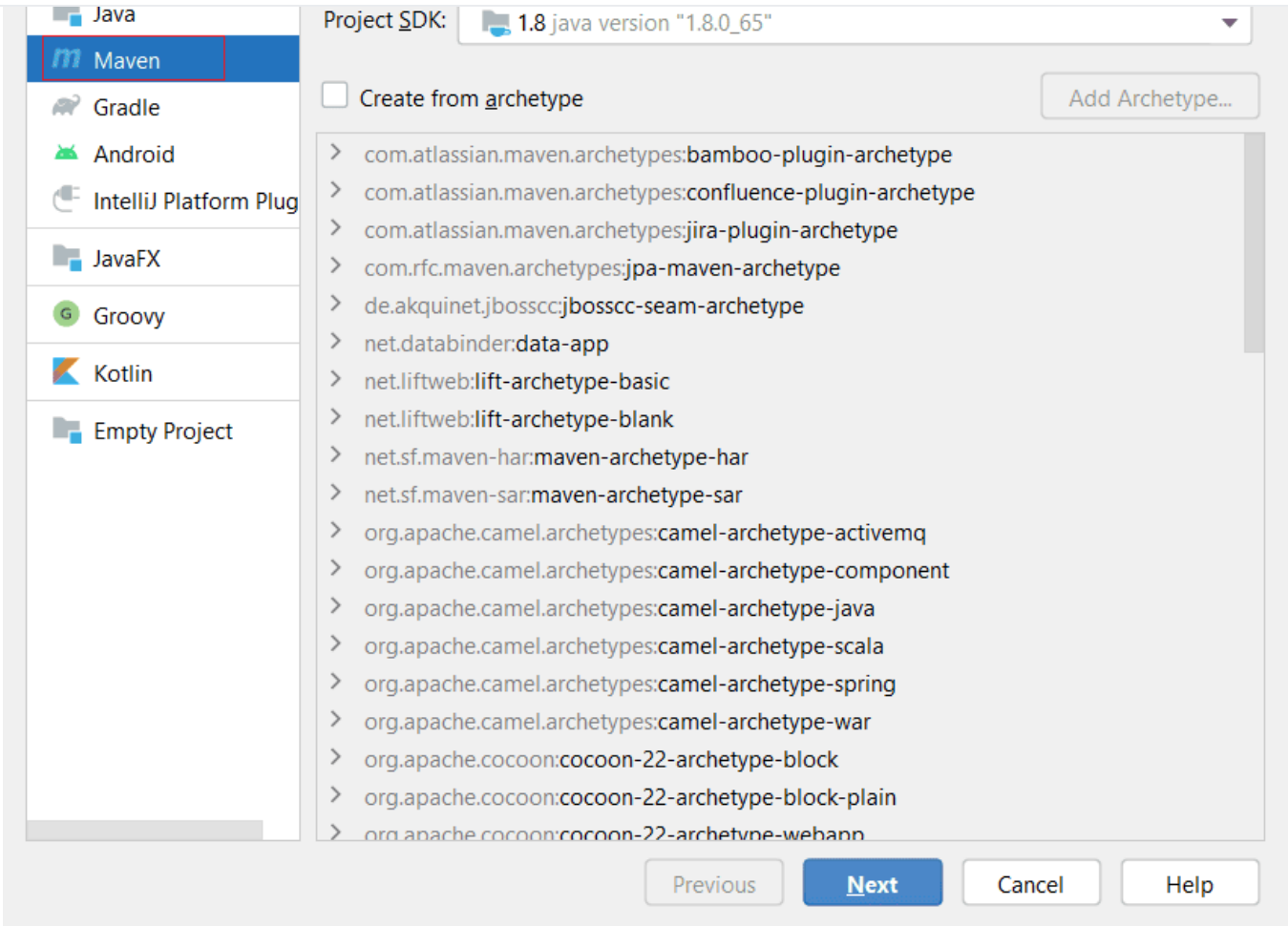


我加上一些注释后，是比较好理解的；引入这个图，重要的原因是为后面设计一个案例帮助你构建认知。

设计一个Spring的Hello World

> 结合上面的使用场景，**设计一个查询用户的案例的两个需求**，来看Spring框架帮我们简化了什么开发工作:
>
> 1. **查询用户数据** - 来看DAO+POJO-> Service 的初始化和装载。
> 2. **给所有Service的查询方法记录日志**

- **创建一个Maven的Java项目**

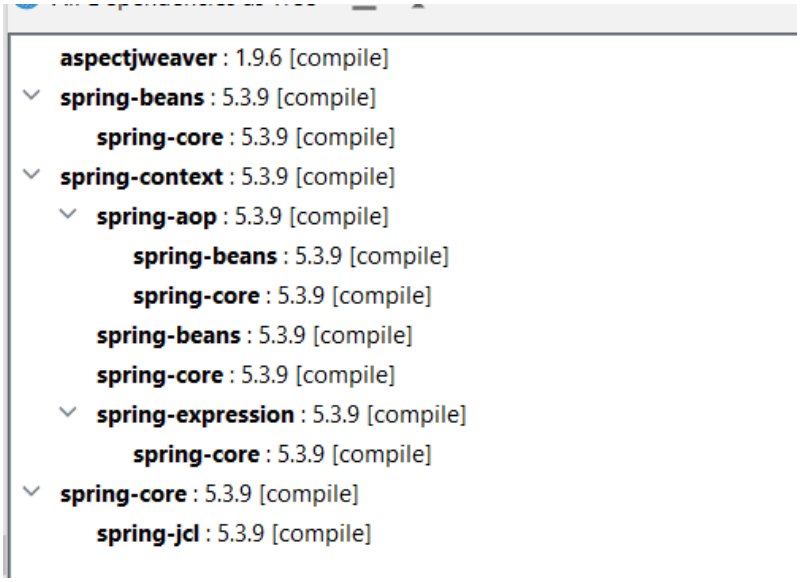


- **引入Spring框架的POM依赖，以及查看这些依赖之间的关系**

xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>tech.pdai</groupId>
    <artifactId>001-spring-framework-demo-helloworld-xml</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
        <spring.version>5.3.9</spring.version>
        <aspectjweaver.version>1.9.6</aspectjweaver.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>

```

```
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>${aspectjweaver.version}</version>
        </dependency>
    </dependencies>

</project>
```



- POJO - User

java

```java
package tech.pdai.springframework.entity;

/**
 * @author pdai
 */
public class User {

    /**
     * user's name.
     */
    private String name;

    /**
     * user's age.
     */
    private int age;

    /**
     * init.
     *
     * @param name name
     * @param age  age
     */
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

```

```
    }

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

    public int getAge() {
        return age;
    }

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

- **DAO 获取 POJO， UserDaoServiceImpl (mock 数据)**

```java
package tech.pdai.springframework.dao;

import java.util.Collections;
import java.util.List;

import tech.pdai.springframework.entity.User;

/**
 * @author pdai
 */
public class UserDaoImpl {

    /**
     * init.
     */
    public UserDaoImpl() {
    }

    /**
     * mocked to find user list.
     *
     * @return user list
     */
    public List<User> findUserList() {
        return Collections.singletonList(new User("pdai", 18));
    }
}
```

并增加daos.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       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">
    <bean id="userDao" class="tech.pdai.springframework.dao.UserDaoImpl">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>
    <!-- more bean definitions for data access objects go here -->
</beans>
```

```java
package tech.pdai.springframework.service;

import java.util.List;

import tech.pdai.springframework.dao.UserDaoImpl;
import tech.pdai.springframework.entity.User;

/**
 * @author pdai
 */
public class UserServiceImpl {

    /**
     * user dao impl.
     */
    private UserDaoImpl userDao;

    /**
     * init.
     */
    public UserServiceImpl() {
    }

    /**
     * find user list.
     *
     * @return user list
     */
    public List<User> findUserList() {
        return this.userDao.findUserList();
    }

    /**
     * set dao.
     *
     * @param userDao user dao
     */
    public void setUserDao(UserDaoImpl userDao) {
        this.userDao = userDao;
    }
}
```

并增加services.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       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">
    <!-- services -->
    <bean id="userService" class="tech.pdai.springframework.service.UserServiceImpl">
        <property name="userDao" ref="userDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>
    <!-- more bean definitions for services go here -->
</beans>
```

- **拦截所有service中的方法，并输出记录**

```java
import java.lang.reflect.Method;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

/**
 * @author pdai
 */
@Aspect
public class LogAspect {

    /**
     * aspect for every methods under service package.
     */
    @Around("execution(* tech.pdai.springframework.service.*.*(..))")
    public Object businessService(ProceedingJoinPoint pjp) throws Throwable {
        // get attribute through annotation
        Method method = ((MethodSignature) pjp.getSignature()).getMethod();
        System.out.println("execute method: " + method.getName());

        // continue to process
        return pjp.proceed();
    }

}
```

并增加aspects.xml

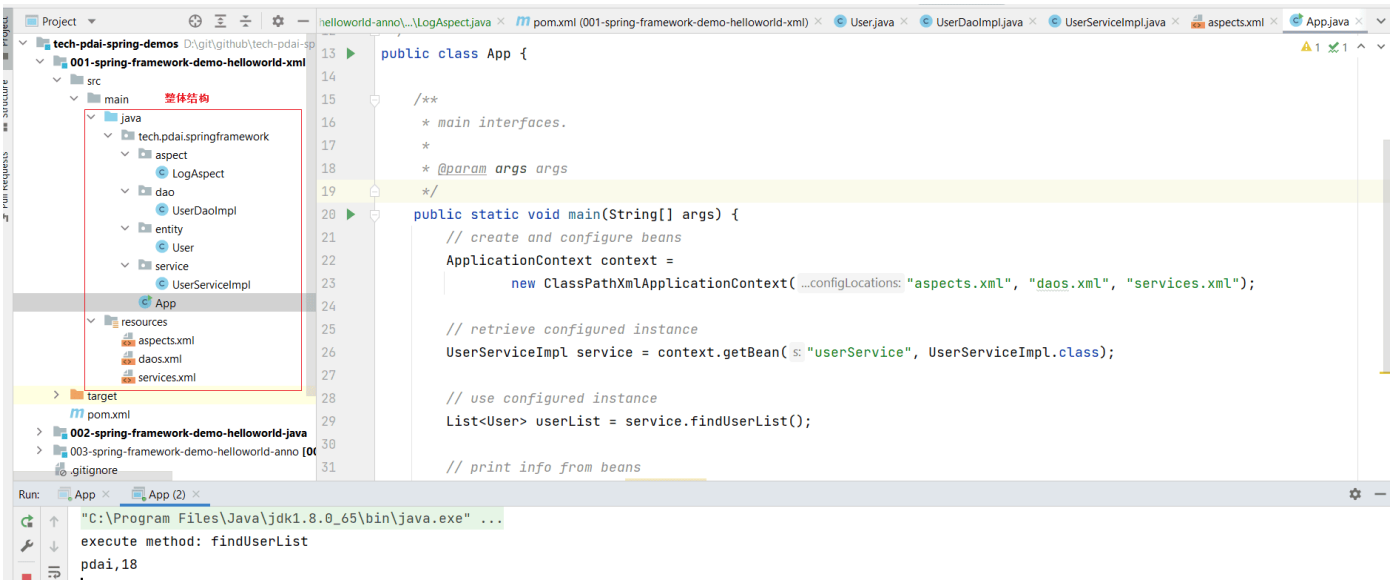
```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
    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

    <context:component-scan base-package="tech.pdai.springframework" />

    <aop:aspectj-autoproxy/>

    <bean id="logAspect" class="tech.pdai.springframework.aspect.LogAspect">
        <!-- configure properties of aspect here as normal -->
    </bean>
    <!-- more bean definitions for data access objects go here -->
</beans>
```

- **组装App**

```java
package tech.pdai.springframework;

import java.util.List;
```

```java
import tech.pdai.springframework.entity.User;
import tech.pdai.springframework.service.UserServiceImpl;

/**
 * @author pdai
 */
public class App {

    /**
     * main interfaces.
     *
     * @param args args
     */
    public static void main(String[] args) {
        // create and configure beans
        ApplicationContext context =
                new ClassPathXmlApplicationContext("aspects.xml", "daos.xml", "services.xml");

        // retrieve configured instance
        UserServiceImpl service = context.getBean("userService", UserServiceImpl.class);

        // use configured instance
        List<User> userList = service.findUserList();

        // print info from beans
        userList.forEach(a -> System.out.println(a.getName() + "," + a.getAge()));
    }
}
```

- **整体结构和运行app**



这个例子体现了Spring的哪些核心要点

> 那么Spring框架帮助我们做什么，它体现了什么哪些要点呢?

控制反转 - IOC

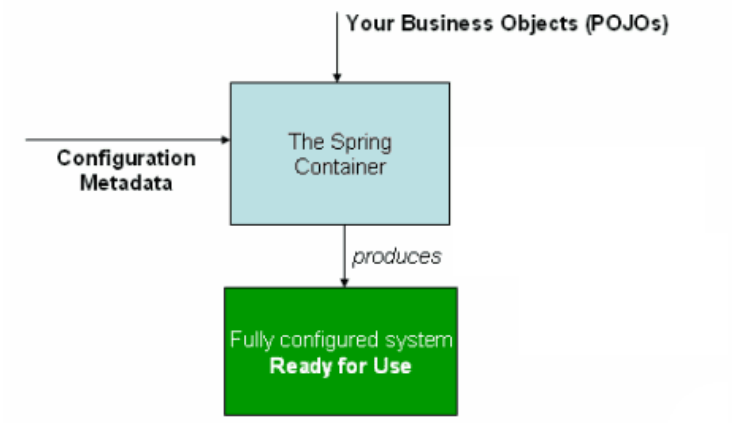
- **如果没有Spring框架，我们需要自己创建User/Dao/Service等**，比如：

```java
UserDaoImpl userDao = new UserDaoImpl();
UserSericeImpl userService = new UserServiceImpl();
userService.setUserDao(userDao);
List<User> userList = userService.findUserList();
```

- **有了Spring框架，可以将原有Bean的创建工作转给框架, 需要用时从Bean的容器中获取即可，这样便简化了开发工作**

Bean的创建和使用分离了。

```java
// create and configure beans
ApplicationContext context =
        new ClassPathXmlApplicationContext("aspects.xml", "daos.xml", "services.xml");

// retrieve configured instance
UserServiceImpl service = context.getBean("userService", UserServiceImpl.class);

// use configured instance
List<User> userList = service.findUserList();
```



更进一步，**你便能理解为何会有如下的知识点了**：

1. Spring框架管理这些Bean的创建工作，即由用户管理Bean转变为框架管理Bean，这个就叫**控制反转 - Inversion of Control (IoC)**
2. Spring 框架托管创建的Bean放在哪里呢？ 这便是**IoC Container**;
3. Spring 框架为了更好让用户配置Bean，必然会引入**不同方式来配置Bean？ 这便是xml配置，Java配置，注解配置**等支持
4. Spring 框架既然接管了Bean的生成，必然需要**管理整个Bean的生命周期**等;
5. 应用程序代码从Ioc Container中获取依赖的Bean，注入到应用程序中，这个过程叫 **依赖注入(Dependency Injection，DI)**； 所以说控制反转是通过依赖注入实现的，其实它们是同一个概念的不同角度描述。通俗来说就是**IoC是设计思想，DI是实现方式**
6. 在依赖注入时，有哪些方式呢？这就是构造器方式，@Autowired, @Resource, @Qualifier... 同时Bean之间存在依赖（可能存在先后顺序问题，以及**循环依赖问题**等）

这边引入我们后续的相关文章：Spring基础 - Spring之控制反转(IOC)

面向切面 - AOP

> 来看第二个需求：**给Service所有方法调用添加日志**（调用方法时），本质上是解耦问题；

```java
/**
* find user list.
*
* @return user list
*/
public List<User> findUserList() {
    System.out.println("execute method findUserList");
    return this.userDao.findUserList();
}
```

- **有了Spring框架，通过@Aspect注解 定义了切面，这个切面中定义了拦截所有service中的方法，并记录日志； 可以明显看到，框架将日志记录和业务需求的代码解耦了，不再是侵入式的了**

```java
/**
* aspect for every methods under service package.
*/
@Around("execution(* tech.pdai.springframework.service.*.*(..))")
public Object businessService(ProceedingJoinPoint pjp) throws Throwable {
    // get attribute through annotation
    Method method = ((MethodSignature) pjp.getSignature()).getMethod();
    System.out.println("execute method: " + method.getName());

    // continue to process
    return pjp.proceed();
}
```

更进一步，**你便能理解为何会有如下的知识点了**：

1. Spring 框架通过定义切面, 通过拦截切点实现了不同业务模块的解耦，这个就叫**面向切面编程 - Aspect Oriented Programming (AOP)**
2. 为什么@Aspect注解使用的是aspectj的jar包呢？这就引出了**Aspect4J和Spring AOP的历史渊源**，只有理解了Aspect4J和Spring的渊源才能理解有些注解上的兼容设计
3. 如何支持**更多拦截方式**来实现解耦， 以满足更多场景需求呢？ 这就是@Around, @Pointcut... 等的设计
4. 那么Spring框架又是如何实现AOP的呢？ 这就引入**代理技术，分静态代理和动态代理**，动态代理又包含JDK代理和CGLIB代理等

这边引入我们后续的相关文章：Spring基础 - Spring之面向切面编程(AOP)

Spring框架设计如何逐步简化开发的

> 通过上述的框架介绍和例子，已经初步知道了Spring设计的两个大的要点：IOC和AOP；从框架的设计角度而言，更为重要的是简化开发，比如提供更为便捷的配置Bean的方式，直至0配置（即约定大于配置）。这里我将通过Spring历史版本的发展，和SpringBoot的推出等，来帮你理解Spring框架是如何逐步简化开发的。

Java 配置方式改造

在前文的例子中， 通过xml配置方式实现的，这种方式实际上比较麻烦； 我通过Java配置进行改造：

- User，UserDaoImpl, UserServiceImpl，LogAspect不用改
- 将原通过.xml配置转换为Java配置

```java
package tech.pdai.springframework.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

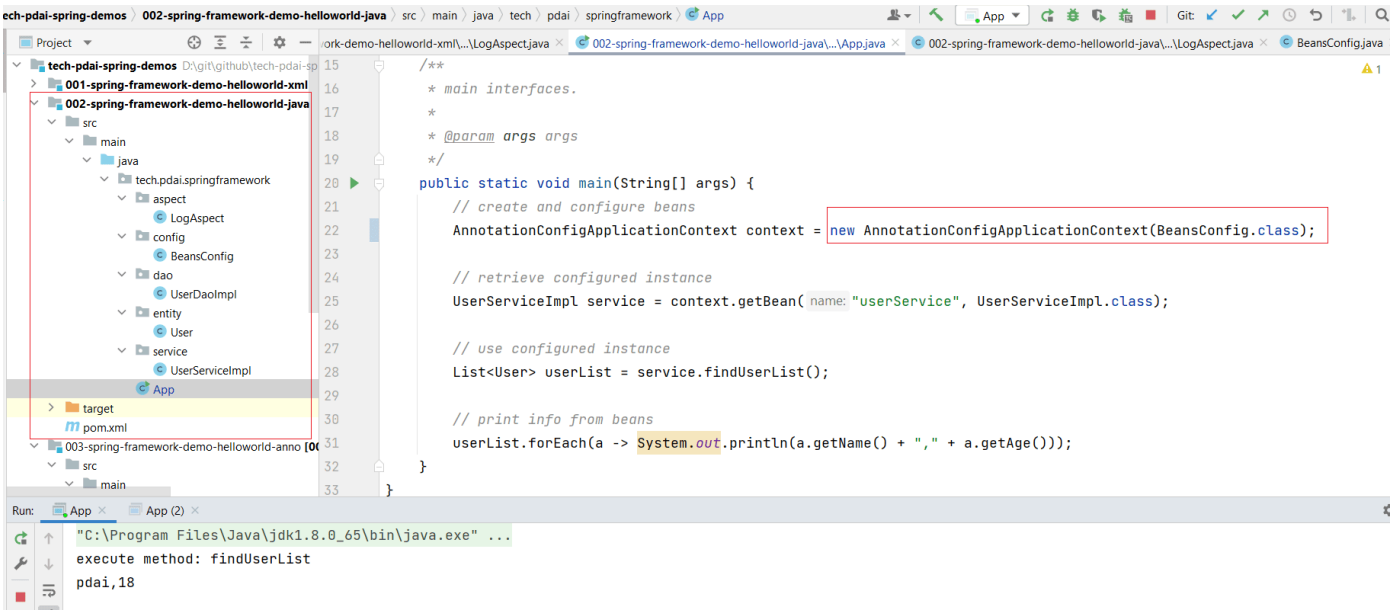
```java
import tech.pdai.springframework.service.UserServiceImpl;

/**
 * @author pdai
 */
@EnableAspectJAutoProxy
@Configuration
public class BeansConfig {

    /**
     * @return user dao
     */
    @Bean("userDao")
    public UserDaoImpl userDao() {
        return new UserDaoImpl();
    }

    /**
     * @return user service
     */
    @Bean("userService")
    public UserServiceImpl userService() {
        UserServiceImpl userService = new UserServiceImpl();
        userService.setUserDao(userDao());
        return userService;
    }

    /**
     * @return log aspect
     */
    @Bean("logAspect")
    public LogAspect logAspect() {
        return new LogAspect();
    }
}
```

- **在App中加载BeansConfig的配置**

```java
package tech.pdai.springframework;

import java.util.List;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import tech.pdai.springframework.config.BeansConfig;
import tech.pdai.springframework.entity.User;
import tech.pdai.springframework.service.UserServiceImpl;

/**
 * @author pdai
 */
public class App {

    /**
     * main interfaces.
     *
     * @param args args
     */
    public static void main(String[] args) {
        // create and configure beans
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(BeansConfig.class);

```

```
27
28          // use configured instance
29          List<User> userList = service.findUserList();
30
31          // print info from beans
32          userList.forEach(a -> System.out.println(a.getName() + "," + a.getAge()));
33      }
    }
```

- **整体结构和运行app**



注解配置方式改造

更进一步，Java 5开始提供注解支持，Spring 2.5 开始完全支持基于注解的配置并且也支持JSR250 注解。在Spring后续的版本发展倾向于通过注解和Java配置结合使用.

- **BeanConfig 不再需要Java配置**

java

```
1   package tech.pdai.springframework.config;
2
3   import org.springframework.context.annotation.ComponentScan;
4   import org.springframework.context.annotation.ComponentScans;
5   import org.springframework.context.annotation.Configuration;
6   import org.springframework.context.annotation.EnableAspectJAutoProxy;
7
8   /**
9    * @author pdai
10   */
11  @Configuration
12  @EnableAspectJAutoProxy
13  public class BeansConfig {
14
15  }
```

- **UserDaoImpl 增加了 @Repository注解**

```
 */
@Repository
public class UserDaoImpl {

    /**
     * mocked to find user list.
     *
     * @return user list
     */
    public List<User> findUserList() {
        return Collections.singletonList(new User("pdai", 18));
    }
}
```

- **UserServiceImpl 增加了@Service 注解，并通过@Autowired注入userDao.**

java

```
/**
 * @author pdai
 */
@Service
public class UserServiceImpl {

    /**
     * user dao impl.
     */
    @Autowired
    private UserDaoImpl userDao;

    /**
     * find user list.
     *
     * @return user list
     */
    public List<User> findUserList() {
        return userDao.findUserList();
    }

}
```

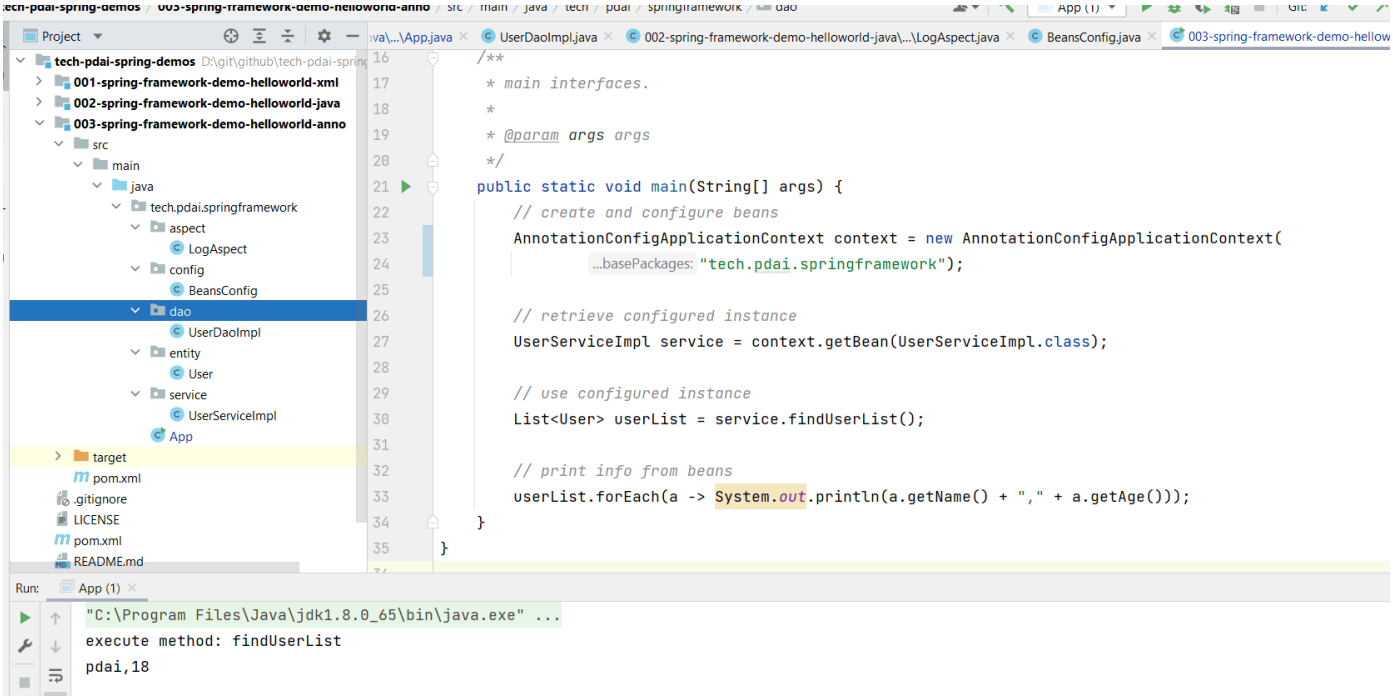
- **在App中扫描tech.pdai.springframework包**

java

```
package tech.pdai.springframework;

import java.util.List;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import tech.pdai.springframework.entity.User;
import tech.pdai.springframework.service.UserServiceImpl;

/**
 * @author pdai
 */
public class App {

    /**
     * main interfaces.
     *
     * @param args args
     */
```

```
                "tech.pdai.springframework");

        // retrieve configured instance
        UserServiceImpl service = context.getBean(UserServiceImpl.class);

        // use configured instance
        List<User> userList = service.findUserList();

        // print info from beans
        userList.forEach(a -> System.out.println(a.getName() + "," + a.getAge()));
    }
}
```

- **整体结构和运行app**



SpringBoot托管配置

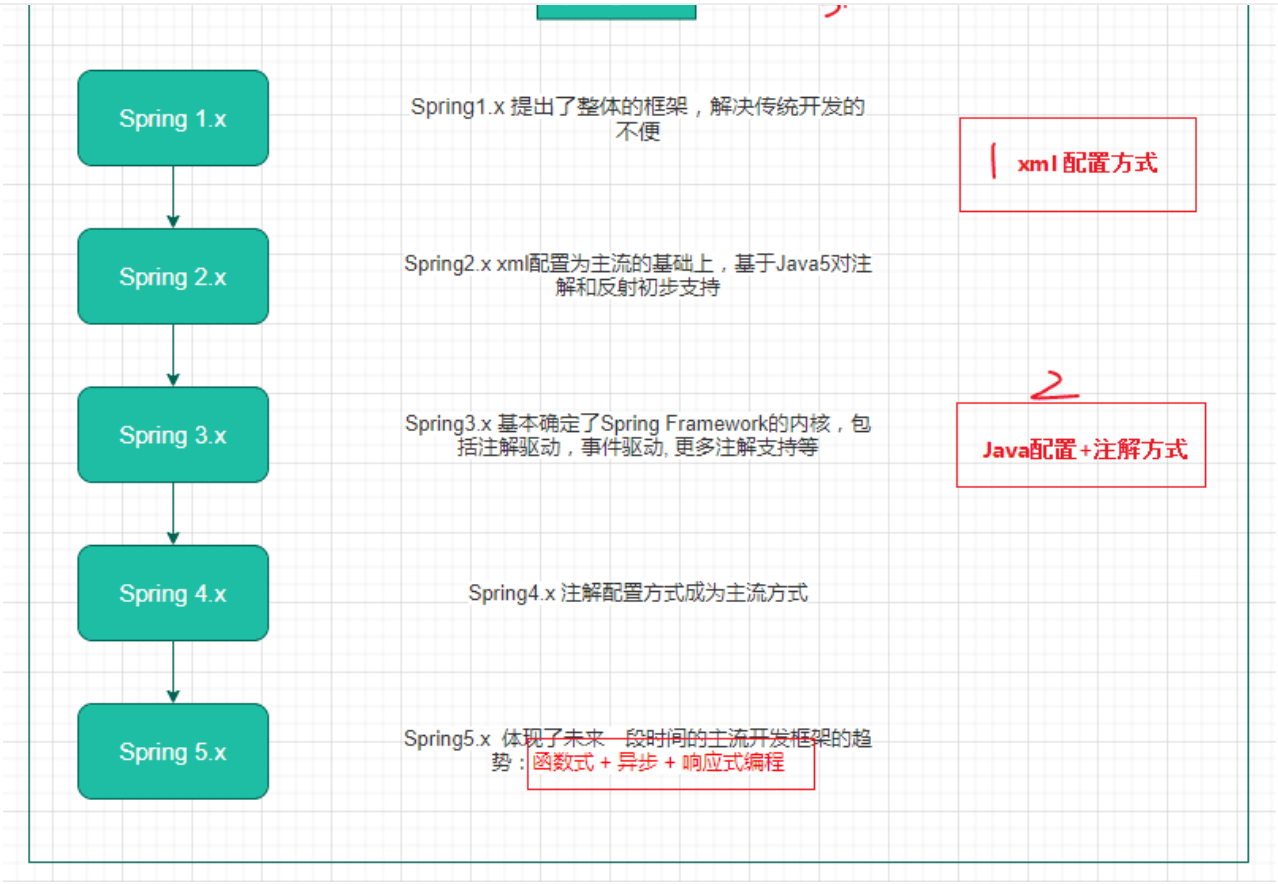
Springboot实际上通过约定大于配置的方式，使用xx-starter统一的对Bean进行默认初始化，用户只需要很少的配置就可以进行开发了。

这个因为很多开发者都是从SpringBoot开始着手开发的，所以这个比较好理解。我们需要的是将知识点都串联起来，构筑认知体系。

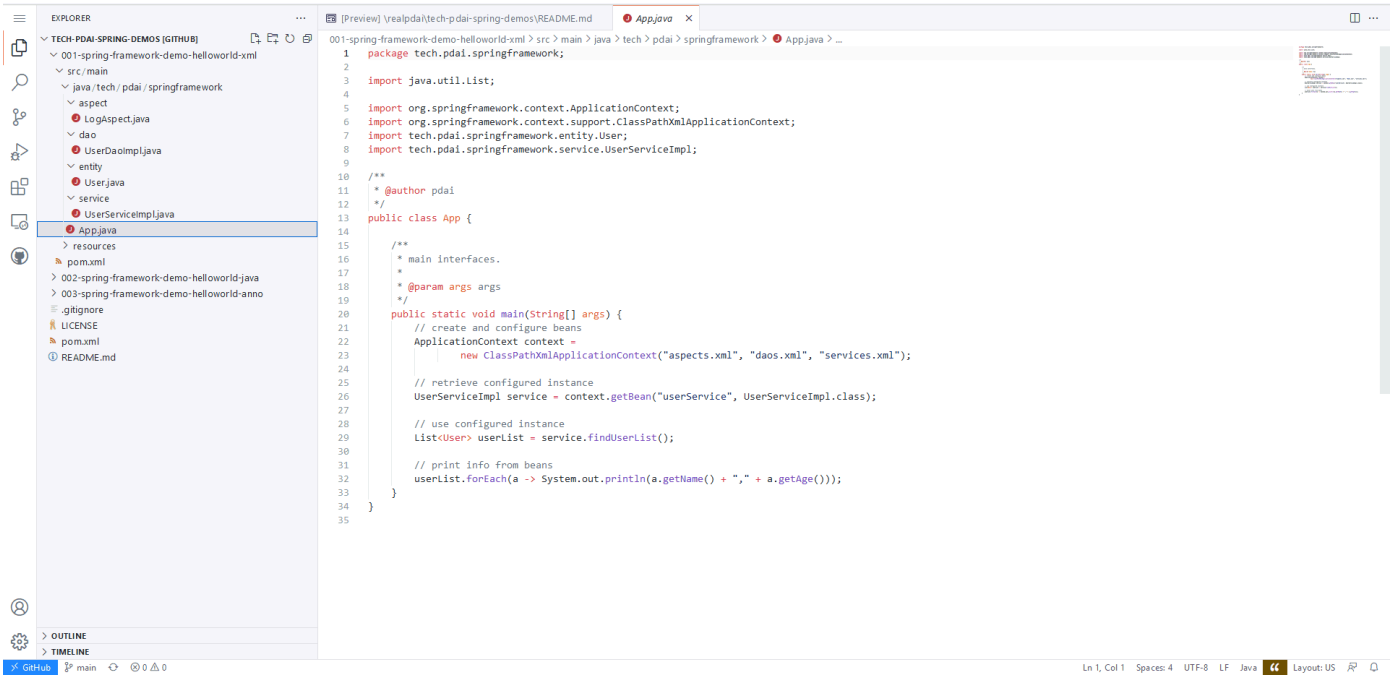
结合Spring历史版本和SpringBoot看发展

最后结合Spring历史版本总结下它的发展：

（这样是不是能够帮助你在整体上构建了知识体系的认知了呢？）



PS：相关代码，可以通过这里直接查看



我要纠错

← Spring基础 - Spring和Spring框架组成

Spring基础 - Spring核心之控制反转(IOC) →