



手写Spring

自己动手，从零开发一个迷你版Spring框架！

廖雪峰

2025-06-16

<https://liaoxuefeng.com/books/summerframework/>

目录

1. 简介
2. 实现IoC容器
 - 2.1. 实现ResourceResolver
 - 2.2. 实现PropertyResolver
 - 2.3. 创建BeanDefinition
 - 2.4. 创建Bean实例
 - 2.5. 初始化Bean
 - 2.6. 实现BeanPostProcessor
 - 2.7. 完成IoC容器
3. 实现AOP
 - 3.1. 实现ProxyResolver
 - 3.2. 实现Around
4. 实现JDBC和事务
 - 4.1. 实现JdbcTemplate
 - 4.2. 实现声明式事务
5. 实现Web MVC
 - 5.1. 启动IoC容器
 - 5.2. 实现MVC
 - 5.3. 开发Web应用
6. 实现Boot
 - 6.1. 启动嵌入式Tomcat
 - 6.2. 开发Boot应用
7. 期末总结

简介

[原文链接](#)



对于Java后端开发的同学来说，Spring框架已经是事实上的标准，如果对Spring和Spring Boot还不熟悉，那需要立刻抓紧时间学习[Spring](#)和[Spring Boot](#)。对于已经能熟练使用Spring框架的同学来说，要进一步理解Spring的设计思想，提升自己的架构能力，不如自己动手，从零开始编写一个Spring框架。

本教程的目标就是以Spring框架为原型，专注于实现Spring的核心功能，编写一个迷你版的Spring框架，我们把它命名为Summer Framework，与Spring主要区别在于，它俩的图标有所不同：

Spring Framework	Summer Framework

Summer Framework设计目标如下：

- context模块：实现ApplicationContext容器与Bean的管理；
- aop模块：实现AOP功能；
- jdbc模块：实现JdbcTemplate，以及声明式事务管理；
- web模块：实现Web MVC和REST API；
- boot模块：实现一个简化版的“Spring Boot”，用于打包运行。

我们会一步一步实现各模块，并在此基础上开发完整的应用程序。

本教程的所有源码均可从[GitHub](#)或[Gitee](#)下载。

[评论](#)

实现IoC容器

[原文链接](#)

Spring的核心就是能管理一组Bean，并能自动配置依赖关系的IoC容器。而我们的Summer Framework的核心context模块就是要实现IoC容器。

设计目标

Spring的IoC容器分为两类：BeanFactory和ApplicationContext，前者总是延迟创建Bean，而后者则在启动时初始化所有Bean。实际使用时，99%都采用ApplicationContext，因此，Summer Framework仅实现ApplicationContext，不支持BeanFactory。

早期的Spring容器采用XML来配置Bean，后期又加入了自动扫描包的功能，即通过 `<context:component-scan base-package="org.example"/>` 的配置。再后来，又加入了Annotation配置，并通过 `@ComponentScan` 注解实现自动扫描。如果使用Spring Boot，则99%都采用 `@ComponentScan` 注解方式配置，因此，Summer Framework也仅实现Annotation配置 + `@ComponentScan` 扫描方式完成容器的配置。

此外，Summer Framework仅支持Singleton类型的Bean，不支持Prototype类型的Bean，因为实际使用中，99%都采用Singleton。依赖注入则与Spring保持一致，支持构造方法、Setter方法与字段注入。支持 `@Configuration` 和 `BeanPostProcessor`。至于Spring的其他功能，例如，层级容器、MessageSource、一个Bean允许多个名字等功能，一概不支持！

下表列出了Spring Framework和Summer Framework在IoC容器方面的异同：

功能	Spring Framework	Summer Framework
IoC容器	支持BeanFactory和ApplicationContext	仅支持ApplicationContext
配置方式	支持XML与Annotation	仅支持Annotation
扫描方式	支持按包名扫描	支持按包名扫描
Bean类型	支持Singleton和Prototype	仅支持Singleton
Bean工厂	支持FactoryBean和@Bean注解	仅支持@Bean注解
定制Bean	支持BeanPostProcessor	支持BeanPostProcessor

功能	Spring Framework	Summer Framework
依赖注入	支持构造方法、Setter方法与字段	支持构造方法、Setter方法与字段
多容器	支持父子容器	不支持

Annotation配置

从使用者的角度看，使用IoC容器时，需要定义一个入口配置，它通常长这样：

```
@ComponentScan
public class AppConfig {
}
```

`AppConfig` 只是一个配置类，它的目的是通过 `@ComponentScan` 来标识要扫描的Bean的包。如果没有明确写出包名，那么将基于 `AppConfig` 所在包进行扫描，如果明确写出了包名，则在指定的包下进行扫描。

在扫描过程中，凡是带有注解 `@Component` 的类，将被添加到IoC容器进行管理：

```
@Component
public class Hello {
}
```

我们用到的许多第三方组件也经常纳入IoC容器管理。这些第三方组件是不可能带有

`@Component` 注解的，引入第三方Bean只能通过工厂模式，即在 `@Configuration` 工厂类中定义带 `@Bean` 的工厂方法：

```
@Configuration
public class DbConfig {
    @Bean
    DataSource createDataSource(...) {
        return new HikariDataSource(...);
    }

    @Bean
    JdbcTemplate createJdbcTemplate(...) {
        return new JdbcTemplate(...);
    }
}
```

```
}  
}
```

基于Annotation配置的IoC容器基本用法就是上面所述。下面，我们就一步一步来实现IoC容器。

[评论](#)

实现ResourceResolver

原文链接

在编写IoC容器之前，我们首先要实现 `@ComponentScan`，即解决“在指定包下扫描所有Class”的问题。

Java的ClassLoader机制可以在指定的Classpath中根据类名加载指定的Class，但遗憾的是，给出一个包名，例如，`org.example`，它并不能获取到该包下的所有Class，也不能获取子包。要在Classpath中扫描指定包名下的所有Class，包括子包，实际上是在Classpath中搜索所有文件，找出文件名匹配的 `.class` 文件。例如，Classpath中搜索的文件

`org/example/Hello.class` 就符合包名 `org.example`，我们需要根据文件路径把它变为 `org.example.Hello`，就相当于获得了类名。因此，搜索Class变成了搜索文件。

我们先定义一个 `Resource` 类型表示文件：

```
public record Resource(String path, String name) {  
}
```

再仿造Spring提供一个 `ResourceResolver`，定义 `scan()` 方法来获取扫描到的 `Resource`：

```
public class ResourceResolver {  
    String basePackage;  
  
    public ResourceResolver(String basePackage) {  
        this.basePackage = basePackage;  
    }  
  
    public <R> List<R> scan(Function<Resource, R> mapper) {  
        ...  
    }  
}
```

这样，我们就可以扫描指定包下的所有文件。有的同学会问，我们的目的是扫描 `.class` 文件，如何过滤出Class？

注意到 `scan()` 方法传入了一个映射函数，我们传入 `Resource` 到Class Name的映射，就可以扫描出Class Name：


```
// 定义一个扫描器：
ResourceResolver rr = new ResourceResolver("org.example");
List<String> classList = rr.scan(res -> {
    String name = res.name(); // 资源名称"org/example/Hello.class"
    if (name.endsWith(".class")) { // 如果以.class结尾
        // 把"org/example/Hello.class"变为"org.example.Hello":
        return name.substring(0, name.length() - 6).replace("/",
            ".").replace("\\", ".");
    }
    // 否则返回null表示不是有效的Class Name:
    return null;
});
```

这样，`ResourceResolver` 只负责扫描并列出现所有文件，由客户端决定是找出 `.class` 文件，还是找出 `.properties` 文件。

在ClassPath中扫描文件的代码是固定模式，可以在网上搜索获得，例如StackOverflow的[这个回答](#)。这里要注意的一点是，Java支持在jar包中搜索文件，所以，不但需要在普通目录中搜索，也需要在Classpath中列出的jar包中搜索，核心代码如下：

```
// 通过ClassLoader获取URL列表：
Enumeration<URL> en = getContextClassLoader().getResources("org/example");
while (en.hasMoreElements()) {
    URL url = en.nextElement();
    URI uri = url.toURI();
    if (uri.toString().startsWith("file:")) {
        // 在目录中搜索
    }
    if (uri.toString().startsWith("jar:")) {
        // 在Jar包中搜索
    }
}
```

几个要点：

1. ClassLoader首先从 `Thread.getContextClassLoader()` 获取，如果获取不到，再从当前Class获取，因为Web应用的ClassLoader不是JVM提供的基于Classpath的ClassLoader，而是Servlet容器提供的ClassLoader，它不在默认的Classpath搜索，而是在 `/WEB-INF/classes` 目录和 `/WEB-INF/lib` 的所有jar包搜索，从 `Thread.getContextClassLoader()` 可以获取到Servlet容器专属的ClassLoader；

2. Windows和Linux/macOS的路径分隔符不同，前者是 `\`，后者是 `/`，需要正确处理；
3. 扫描目录时，返回的路径可能是 `abc/xyz`，也可能是 `abc/xyz/`，需要注意处理末尾的 `/`。

这样我们就完成了能扫描指定包以及子包下所有文件的 `ResourceResolver`。

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

实现PropertyResolver

原文链接

Spring的注入分为 `@Autowired` 和 `@Value` 两种。对于 `@Autowired`，涉及到Bean的依赖，而对于 `@Value`，则仅仅是将对应的配置注入，不涉及Bean的依赖，相对比较简单。为了注入配置，我们用 `PropertyResolver` 保存所有配置项，对外提供查询功能。

本节我们来实现 `PropertyResolver`，它支持3种查询方式：

1. 按配置的key查询，例如：`getProperty("app.title")`；
2. 以 `${abc.xyz}` 形式的查询，例如，`getProperty("${app.title}")`，常用于 `@Value("${app.title}")` 注入；
3. 带默认值的，以 `${abc.xyz:defaultValue}` 形式的查询，例如，`getProperty("${app.title:Summer}")`，常用于 `@Value("${app.title:Summer}")` 注入。

Java本身提供了按key-value查询的 `Properties`，我们先传入 `Properties`，内部按key-value存储：

```
public class PropertyResolver {

    Map<String, String> properties = new HashMap<>();

    public PropertyResolver(Properties props) {
        // 存入环境变量：
        this.properties.putAll(System.getenv());
        // 存入Properties：
        Set<String> names = props.stringPropertyNames();
        for (String name : names) {
            this.properties.put(name, props.getProperty(name));
        }
    }
}
```

这样，我们在 `PropertyResolver` 内部，通过一个 `Map<String, String>` 存储了所有的配置项，包括环境变量。对于按key查询的功能，我们可以简单实现如下：

```
@Nullable
public String getProperty(String key) {
```

```
    return this.properties.get(key);  
}
```

下一步，我们准备解析 `${abc.xyz:defaultValue}` 这样的key，先定义一个 `PropertyExpr`，把解析后的key和defaultValue存储起来：

```
record PropertyExpr(String key, String defaultValue) {  
}
```

然后按 `${...}` 解析：

```
PropertyExpr parsePropertyExpr(String key) {  
    if (key.startsWith("${") && key.endsWith("}")) {  
        // 是否存在defaultValue?  
        int n = key.indexOf(':');  
        if (n == (-1)) {  
            // 没有defaultValue: ${key}  
            String k = key.substring(2, key.length() - 1);  
            return new PropertyExpr(k, null);  
        } else {  
            // 有defaultValue: ${key:default}  
            String k = key.substring(2, n);  
            return new PropertyExpr(k, key.substring(n + 1, key.length() - 1));  
        }  
    }  
    return null;  
}
```

我们把 `getProperty()` 改造一下，即可实现查询 `${abc.xyz:defaultValue}`：

```
@Nullable  
public String getProperty(String key) {  
    // 解析${abc.xyz:defaultValue}:  
    PropertyExpr keyExpr = parsePropertyExpr(key);  
    if (keyExpr != null) {  
        if (keyExpr.defaultValue() != null) {  
            // 带默认值查询:  
            return getProperty(keyExpr.key(), keyExpr.defaultValue());  
        } else {  

```

```
        // 不带默认值查询:
        return getRequiredProperty(keyExpr.key());
    }
}
// 普通key查询:
String value = this.properties.get(key);
if (value != null) {
    return parseValue(value);
}
return value;
}
```

每次查询到value后，我们递归调用 `parseValue()`，这样就可以支持嵌套的key，例如：

```
${app.title:${APP_NAME:Summer}}
```

这样可以先查询 `app.title`，没有找到就再查询 `APP_NAME`，还没有找到就返回默认值 `Summer`。

注意到Spring的 `${...}` 表达式实际上可以做到组合，例如：

```
jdbc.url=jdbc:mysql://${DB_HOST:localhost}:${DB_PORT:3306}/${DB_NAME}
```

而我们实现的 `${...}` 表达式只能嵌套，不能组合，因为要实现Spring的表达式，需要编写一个完整的能解析表达式的复杂功能，而不能仅仅依靠判断 `${` 开头、`}` 结尾。由于解析表达式的功能过于复杂，因此我们决定不予支持。

Spring还支持更复杂的 `# {...}` 表达式，它可以引用Bean、调用方法、计算等：

```
# {appBean.version() + 1}
```

为此Spring专门提供了一个 `spring-expression` 库来支持这种更复杂的功能。按照一切从简的原则，我们不支持 `# {...}` 表达式。

实现类型转换

除了String类型外，@Value注入时，还允许 `boolean`、`int`、`Long` 等基本类型和包装类型。此外，Spring还支持 `Date`、`Duration` 等类型的注入。我们既要实现类型转换，又不能写死，否则，将来支持新的类型时就要改代码。

我们先写类型转换的入口查询：

```
@Nullable
public <T> T getProperty(String key, Class<T> targetType) {
    String value = getProperty(key);
    if (value == null) {
        return null;
    }
    // 转换为指定类型:
    return convert(targetType, value);
}
```

再考虑如何实现 `convert()` 方法。对于类型转换，实际上是从String转换为指定类型，因此，用函数式接口 `Function<String, Object>` 就很合适：

```
public class PropertyResolver {
    // 存储Class -> Function:
    Map<Class<?>, Function<String, Object>> converters = new HashMap<>();

    // 转换到指定Class类型:
    <T> T convert(Class<?> clazz, String value) {
        Function<String, Object> fn = this.converters.get(clazz);
        if (fn == null) {
            throw new IllegalArgumentException("Unsupported value type: " +
clazz.getName());
        }
        return (T) fn.apply(value);
    }
}
```

这样我们就已经实现了类型转换，下一步是把各种要转换的类型放到 `Map` 里。在构造方法中，我们放入常用的基本类型转换器：

```
public PropertyResolver(Properties props) {
    ...
    // String类型:
    converters.put(String.class, s -> s);
    // boolean类型:
    converters.put(boolean.class, s -> Boolean.parseBoolean(s));
    converters.put(Boolean.class, s -> Boolean.valueOf(s));
}
```

```
// int类型:
converters.put(int.class, s -> Integer.parseInt(s));
converters.put(Integer.class, s -> Integer.valueOf(s));
// 其他基本类型...
// Date/Time类型:
converters.put(LocalDate.class, s -> LocalDate.parse(s));
converters.put(LocalTime.class, s -> LocalTime.parse(s));
converters.put(LocalDateTime.class, s -> LocalDateTime.parse(s));
converters.put(ZonedDateTime.class, s -> ZonedDateTime.parse(s));
converters.put(Duration.class, s -> Duration.parse(s));
converters.put(ZoneId.class, s -> ZoneId.of(s));
}
```

如果再加一个 `registerConverter()` 接口，我们就可以对外提供扩展，让用户自己编写自己定制的Converter，这样一来，我们的PropertyResolver就准备就绪，读取配置的初始化代码如下：

```
// Java标准库读取properties文件:
Properties props = new Properties();
props.load(fileInput); // 文件输入流
// 构造PropertyResolver:
PropertyResolver pr = new PropertyResolver(props);
// 后续代码调用...
// pr.getProperty("${app.version:1}", int.class)
```

使用YAML配置

Spring Framework并不支持YAML配置，但Spring Boot支持。因为YAML配置比 `.properties` 要方便，所以我们把对YAML的支持也集成进来。

首先引入依赖 `org.yaml:snakeyaml:2.0`，然后我们写一个 `YamlUtils`，通过 `loadYamlAsPlainMap()` 方法读取一个YAML文件，并返回 `Map`：

```
public class YamlUtils {
    public static Map<String, Object> loadYamlAsPlainMap(String path) {
        return ...
    }
}
```

我们把YAML格式：

```
app:
  title: Summer Framework
  version: ${VER:1.0}
```

读取为 `Map`，其中，每个key都是完整路径，相当于把它变为 `.properties` 格式：

```
app.title=Summer Framework
app.version=${VER:1.0}
```

这样我们无需改动 `PropertyResolver` 的代码，使用YAML时，可以按如下方式读取配置：

```
Map<String, Object> configs =
YamlUtils.loadYamlAsPlainMap("/application.yml");
Properties props = new Properties();
props.putAll(config);
PropertyResolver pr = new PropertyResolver(props);
```

读取YAML的代码比较简单，注意要点如下：

1. SnakeYaml默认读出的结构是树形结构，需要“拍平”成 `abc.xyz` 格式的key；
2. SnakeYaml默认会自动转换 `int`、`boolean` 等value，需要禁用自动转换，把所有value均按 `String` 类型返回。

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

评论

创建BeanDefinition

原文链接

现在，我们可以用 `ResourceResolver` 扫描Class，用 `PropertyResolver` 获取配置，下面，我们开始实现IoC容器。

在IoC容器中，每个Bean都有一个唯一的名字标识。Spring还允许为一个Bean定义多个名字，这里我们简化一下，一个Bean只允许一个名字，因此，很容易想到用一个 `Map<String, Object>` 保存所有的Bean：

```
public class AnnotationConfigApplicationContext {  
    Map<String, Object> beans;  
}
```

这么做不是不可以，但是丢失了大量Bean的定义信息，不便于我们创建Bean以及解析依赖关系。合理的方式是先定义 `BeanDefinition`，它能从Annotation中提取到足够的信息，便于后续创建Bean、设置依赖、调用初始化方法等：

```
public class BeanDefinition {  
    // 全局唯一的Bean Name:  
    String name;  
  
    // Bean的声明类型:  
    Class<?> beanClass;  
  
    // Bean的实例:  
    Object instance = null;  
  
    // 构造方法/null:  
    Constructor<?> constructor;  
  
    // 工厂方法名称/null:  
    String factoryName;  
  
    // 工厂方法/null:  
    Method factoryMethod;  
  
    // Bean的顺序:
```

```
    int order;

    // 是否标识@Primary:
    boolean primary;

    // init/destroy方法名称:
    String initMethodName;
    String destroyMethodName;

    // init/destroy方法:
    Method initMethod;
    Method destroyMethod;
}
```

对于自己定义的带 `@Component` 注解的Bean，我们需要获取Class类型，获取构造方法来创建Bean，然后收集 `@PostConstruct` 和 `@PreDestroy` 标注的初始化与销毁的方法，以及其他信息，如 `@Order` 定义Bean的内部排序顺序，`@Primary` 定义存在多个相同类型时返回哪个“主要”Bean。一个典型的定义如下：

```
@Component
public class Hello {
    @PostConstruct
    void init() {}

    @PreDestroy
    void destroy() {}
}
```

对于 `@Configuration` 定义的 `@Bean` 方法，我们把它看作Bean的工厂方法，我们需要获取方法返回值作为Class类型，方法本身作为创建Bean的 `factoryMethod`，然后收集 `@Bean` 定义的 `initMethod` 和 `destroyMethod` 标识的初始化与销毁的方法名，以及其他 `@Order`、`@Primary` 等信息。一个典型的定义如下：

```
@Configuration
public class AppConfig {
    @Bean(initMethod="init", destroyMethod="close")
    DataSource createDataSource() {
        return new HikariDataSource(...);
    }
}
```

```
}  
}
```

Bean的声明类型

这里我们要特别注意一点，就是Bean的声明类型。对于 `@Component` 定义的Bean，它的声明类型就是其Class本身。然而，对于用 `@Bean` 工厂方法创建的Bean，它的声明类型与实际类型不一定是同一类型。上述 `createDataSource()` 定义的Bean，声明类型是 `DataSource`，实际类型却是某个子类，例如 `HikariDataSource`，因此要特别注意，我们在 `BeanDefinition` 中，存储的 `beanClass` 是**声明类型**，实际类型不必存储，因为可以通过 `instance.getClass()` 获得：

```
public class BeanDefinition {  
    // Bean的声明类型：  
    Class<?> beanClass;  
  
    // Bean的实例：  
    Object instance = null;  
}
```

这也引出了下一个问题：如果我们按照名字查找Bean或BeanDefinition，要么拿到唯一实例，要么不存在，即通过查询 `Map<String, BeanDefinition>` 即可完成：

```
public class AnnotationConfigApplicationContext {  
    Map<String, BeanDefinition> beans;  
  
    // 根据Name查找BeanDefinition, 如果Name不存在, 返回null  
    @Nullable  
    public BeanDefinition findBeanDefinition(String name) {  
        return this.beans.get(name);  
    }  
}
```

但是通过类型查找Bean或BeanDefinition，我们没法定义一个 `Map<Class, BeanDefinition>`，原因就是Bean的声明类型与实际类型不一定相符，举个例子：

```
@Configuration  
public class AppConfig {  
    @Bean
```

```
AtomicInteger counter() {  
    return new AtomicInteger();  
}  
  
@Bean  
Number bigInt() {  
    return new BigInteger("1000000000");  
}  
}
```

当我们调用 `getBean(AtomicInteger.class)` 时，我们会获得 `counter()` 方法创建的唯一实例，但是，当我们调用 `getBean(Number.class)` 时，`counter()` 方法和 `bigInt()` 方法创建的实例均符合要求，此时，如果有且仅有一个标注了 `@Primary`，就返回标注了 `@Primary` 的 Bean，否则，直接报 `NoUniqueBeanDefinitionException` 错误。

因此，对于 `getBean(Class)` 方法，必须遍历找出所有符合类型的 Bean，如果不唯一，再判断 `@Primary`，才能返回唯一 Bean 或报错。

我们编写一个找出所有类型的 `findBeanDefinitions(Class)` 方法如下：

```
// 根据Type查找若干个BeanDefinition，返回0个或多个：  
List<BeanDefinition> findBeanDefinitions(Class<?> type) {  
    return this.beans.values().stream()  
        // 按类型过滤：  
        .filter(def -> type.isAssignableFrom(def.getBeanClass()))  
        // 排序：  
        .sorted().collect(Collectors.toList());  
}
```

我们再编写一个 `findBeanDefinition(Class)` 方法如下：

```
// 根据Type查找某个BeanDefinition，如果不存在返回null，如果存在多个返回@Primary标注的一个：  
@Nullable  
public BeanDefinition findBeanDefinition(Class<?> type) {  
    List<BeanDefinition> defs = findBeanDefinitions(type);  
    if (defs.isEmpty()) { // 没有找到任何BeanDefinition  
        return null;  
    }  
}
```

```
if (defs.size() == 1) { // 找到唯一的一个
    return defs.get(0);
}
// 多于一个时, 查找@Primary:
List<BeanDefinition> primaryDefs = defs.stream().filter(def ->
def.isPrimary()).collect(Collectors.toList());
if (primaryDefs.size() == 1) { // @Primary唯一
    return primaryDefs.get(0);
}
if (primaryDefs.isEmpty()) { // 不存在@Primary
    throw new NoUniqueBeanDefinitionException(String.format("Multiple
bean with type '%s' found, but no @Primary specified.", type.getName()));
} else { // @Primary不唯一
    throw new NoUniqueBeanDefinitionException(String.format("Multiple
bean with type '%s' found, and multiple @Primary specified.",
type.getName()));
}
}
```

现在, 我们已经定义好了数据结构, 下面开始获取所有 `BeanDefinition` 信息, 实际分两步:

```
public class AnnotationConfigApplicationContext {
    Map<String, BeanDefinition> beans;

    public AnnotationConfigApplicationContext(Class<?> configClass,
PropertyResolver propertyResolver) {
        // 扫描获取所有Bean的Class类型:
        Set<String> beanClassNames = scanForClassNames(configClass);

        // 创建Bean的定义:
        this.beans = createBeanDefinitions(beanClassNames);
    }
    ...
}
```

第一步是扫描指定包下的所有Class, 然后返回Class名字, 这一步比较简单:

```
Set<String> scanForClassNames(Class<?> configClass) {
    // 获取@ComponentScan注解:
    ComponentScan scan = ClassUtils.findAnnotation(configClass,
```

```
ComponentScan.class);
// 获取注解配置的package名字,未配置则默认当前类所在包:
String[] scanPackages = scan == null || scan.value().length == 0 ? new
String[] { configClass.getPackage().getName() } : scan.value();

Set<String> classNameSet = new HashSet<>();
// 依次扫描所有包:
for (String pkg : scanPackages) {
    logger.atDebug().log("scan package: {}", pkg);
    // 扫描一个包:
    var rr = new ResourceResolver(pkg);
    List<String> classList = rr.scan(res -> {
        // 遇到以.class结尾的文件,就将其转换为Class全名:
        String name = res.name();
        if (name.endsWith(".class")) {
            return name.substring(0, name.length() - 6).replace("/",
".").replace("\\", ".");
        }
        return null;
    });
    // 扫描结果添加到Set:
    classNameSet.addAll(classList);
}

// 继续查找@Import(Xyz.class)导入的Class配置:
Import importConfig = configClass.getAnnotation(Import.class);
if (importConfig != null) {
    for (Class<?> importConfigClass : importConfig.value()) {
        String importClassName = importConfigClass.getName();
        classNameSet.add(importClassName);
    }
}
return classNameSet;
}
```

注意到扫描结果是指定包的所有Class名称,以及通过 `@Import` 导入的Class名称,下一步才会真正处理各种注解:

```
Map<String, BeanDefinition> createBeanDefinitions(Set<String> classNameSet)
{
```

```
Map<String, BeanDefinition> defs = new HashMap<>();
for (String className : classNameSet) {
    // 获取Class:
    Class<?> clazz = null;
    try {
        clazz = Class.forName(className);
    } catch (ClassNotFoundException e) {
        throw new BeanCreationException(e);
    }
    // 是否标注@Component?
    Component component = ClassUtils.findAnnotation(clazz,
Component.class);
    if (component != null) {
        // 获取Bean的名称:
        String beanName = ClassUtils.getBeanName(clazz);
        var def = new BeanDefinition(
            beanName, clazz, getSuitableConstructor(clazz),
            getOrder(clazz), clazz.isAnnotationPresent(Primary.class),
            // init/destroy方法名称:
            null, null,
            // 查找@PostConstruct方法:
            ClassUtils.findAnnotationMethod(clazz,
PostConstruct.class),
            // 查找@PreDestroy方法:
            ClassUtils.findAnnotationMethod(clazz, PreDestroy.class));
        addBeanDefinitions(defs, def);
        // 查找是否有@Configuration:
        Configuration configuration = ClassUtils.findAnnotation(clazz,
Configuration.class);
        if (configuration != null) {
            // 查找@Bean方法:
            scanFactoryMethods(beanName, clazz, defs);
        }
    }
}
return defs;
}
```

上述代码需要注意的一点是，查找 `@Component` 时，并不是简单地在Class定义查看 `@Component` 注解，因为Spring的 `@Component` 是可以扩展的，例如，标记为 `Controller` 的

Class也符合要求：

```
@Controller
public class MvcController {...}
```

原因就在于，`@Controller` 注解的定义包含了 `@Component`：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {
    String value() default "";
}
```

所以，判断是否存在 `@Component`，不但要在当前类查找 `@Component`，还要在当前类的所有注解上，查找该注解是否有 `@Component`，因此，我们编写了一个能递归查找注解的方法：

```
public class ClassUtils {
    public static <A extends Annotation> A findAnnotation(Class<?> target,
Class<A> annoClass) {
        A a = target.getAnnotation(annoClass);
        for (Annotation anno : target.getAnnotations()) {
            Class<? extends Annotation> annoType = anno.annotationType();
            if (!annoType.getPackageName().equals("java.lang.annotation"))
            {
                A found = findAnnotation(annoType, annoClass);
                if (found != null) {
                    if (a != null) {
                        throw new BeanDefinitionException("Duplicate @" +
annoClass.getSimpleName() + " found on class " + target.getSimpleName());
                    }
                    a = found;
                }
            }
        }
        return a;
    }
}
```


带有 `@Configuration` 注解的Class，视为Bean的工厂，我们需要继续在

`scanFactoryMethods()` 中查找 `@Bean` 标注的方法：

```
void scanFactoryMethods(String factoryBeanName, Class<?> clazz, Map<String,
BeanDefinition> defs) {
    for (Method method : clazz.getDeclaredMethods()) {
        // 是否带有@Bean标注:
        Bean bean = method.getAnnotation(Bean.class);
        if (bean != null) {
            // Bean的声明类型是方法返回类型:
            Class<?> beanClass = method.getReturnType();
            var def = new BeanDefinition(
                ClassUtils.getBeanName(method), beanClass,
                factoryBeanName,
                // 创建Bean的工厂方法:
                method,
                // @Order
                getOrder(method),
                // 是否存在@Primary标注?
                method.isAnnotationPresent(Primary.class),
                // init方法名称:
                bean.initMethod().isEmpty() ? null : bean.initMethod(),
                // destroy方法名称:
                bean.destroyMethod().isEmpty() ? null :
                bean.destroyMethod(),
                // @PostConstruct / @PreDestroy方法:
                null, null);
            addBeanDefinitions(defs, def);
        }
    }
}
```

注意到 `@Configuration` 注解本身又用 `@Component` 注解修饰了，因此，对于一个

`@Configuration` 来说：

```
@Configuration
public class DateTimeConfig {
    @Bean
    LocalDateTime local() { return LocalDateTime.now(); }
```

```
@Bean
ZonedDateTime zoned() { return ZonedDateTime.now(); }
}
```

实际上创建了3个 `BeanDefinition` :

- `DateTimeConfig`本身;
- `LocalDateTime`;
- `ZonedDateTime`。

不创建 `DateTimeConfig` 行不行? 不行, 因为后续没有 `DateTimeConfig` 的实例, 无法调用 `local()` 和 `zoned()` 方法。因为当前我们只创建了 `BeanDefinition` , 所以对于 `LocalDateTime` 和 `ZonedDateTime` 的 `BeanDefinition` 来说, 还必须保存 `DateTimeConfig` 的名字, 将来才能通过名字查找 `DateTimeConfig` 的实例。

有的同学注意到我们同时存储了 `initMethodName` 和 `initMethod` , 以及 `destroyMethodName` 和 `destroyMethod` , 这是因为在 `@Component` 声明的Bean中, 我们可以根据 `@PostConstruct` 和 `@PreDestroy` 直接拿到Method本身, 而在 `@Bean` 声明的Bean中, 我们拿不到Method, 只能从 `@Bean` 注解提取出字符串格式的方法名称, 因此, 存储在 `BeanDefinition` 的方法名称与方法, 其中至少有一个为 `null` 。

最后, 仔细编写 `BeanDefinition` 的 `toString()` 方法, 使之能打印出详细的信息。我们编写测试, 运行, 打印出每个 `BeanDefinition` 如下:

```
define bean: BeanDefinition [name=annotationDestroyBean,
beanClass=com.itranswarp.scan.destroy.AnnotationDestroyBean, factory=null,
init-method=null, destroy-method=destroy, primary=false, instance=null]

define bean: BeanDefinition [name=nestedBean,
beanClass=com.itranswarp.scan.nested.OuterBean$NestedBean, factory=null,
init-method=null, destroy-method=null, primary=false, instance=null]

define bean: BeanDefinition [name=createSpecifyInitBean,
beanClass=com.itranswarp.scan.init.SpecifyInitBean,
factory=SpecifyInitConfiguration.createSpecifyInitBean(String, String),
init-method=null, destroy-method=null, primary=false, instance=null]

...
```

现在，我们已经能扫描并创建所有的 `BeanDefinition`，只是目前每个 `BeanDefinition` 内部的 `instance` 还是 `null`，因为我们后续才会创建真正的Bean。

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

创建Bean实例

原文链接

当我们拿到所有 `BeanDefinition` 之后，就可以开始创建Bean的实例了。

在创建Bean实例之前，我们先看看Spring支持的4种依赖注入模式：

1. 构造方法注入，例如：

```
@Component
public class Hello {
    JdbcTemplate jdbcTemplate;
    public Hello(@Autowired JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

2. 工厂方法注入，例如：

```
@Configuration
public class AppConfig {
    @Bean
    Hello hello(@Autowired JdbcTemplate jdbcTemplate) {
        return new Hello(jdbcTemplate);
    }
}
```

3. Setter方法注入，例如：

```
@Component
public class Hello {
    JdbcTemplate jdbcTemplate;

    @Autowired
    void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

4. 字段注入，例如：

```
@Component
public class Hello {
    @Autowired
    JdbcTemplate jdbcTemplate;
}
```

然而我们仔细分析，发现这4种注入方式实际上是有区别的。

区别就在于，前两种方式，即构造方法注入和工厂方法注入，Bean的创建与注入是一体的，我们无法把它们分成两个阶段，因为无法中断方法内部代码的执行。而后两种方式，即Setter方法注入和属性注入，Bean的创建与注入是可以分开的，即先创建Bean实例，再用反射调用方法或字段，完成注入。

我们再分析一下循环依赖的问题。循环依赖，即A、B互相依赖，或者A依赖B，B依赖C，C依赖A，形成了一个闭环。IoC容器对Bean进行管理，可以解决部分循环依赖问题，但不是所有循环依赖都能解决。

我们先来看不能解决的循环依赖问题，假定下列代码定义的A、B两个Bean：

```
class A {
    final B b;
    A(B b) { this.b = b; }
}

class B {
    final A a;
    B(A a) { this.a = a; }
}
```

这种通过构造方法注入依赖的两个Bean，如果存在循环依赖，是无解的，因为我们不用IoC，自己写Java代码也写不出正确创建两个Bean实例的代码。

因此，我们把构造方法注入和工厂方法注入的依赖称为强依赖，不能有强依赖的循环依赖，否则只能报错。

后两种注入方式形成的依赖则是弱依赖，假定下列代码定义的A、B两个Bean：

```
class A {  
    B b;  
}  
  
class B {  
    A a;  
}
```

这种循环依赖则很容易解决，因为我们可以分两步，先分别实例化Bean，再注入依赖：

```
// 第一步,实例化:  
A a = new A();  
B b = new B();  
// 第二步,注入:  
a.b = b;  
b.a = a;
```

所以，对于IoC容器来说，创建Bean的过程分两步：

1. 创建Bean的实例，此时必须注入强依赖；
2. 对Bean实例进行Setter方法注入和字段注入。

第一步如果遇到循环依赖则直接报错，第二步则不需要关心有没有循环依赖。

我们先实现第一步：创建Bean的实例，同时注入强依赖。

在上一节代码中，我们已经获得了所有的 `BeanDefinition`：

```
public class AnnotationConfigApplicationContext {  
    PropertyResolver propertyResolver;  
    Map<String, BeanDefinition> beans;  
  
    public AnnotationConfigApplicationContext(Class<?> configClass,  
        PropertyResolver propertyResolver) {  
        this.propertyResolver = propertyResolver;  
        // 扫描获取所有Bean的Class类型:  
        Set<String> beanClassNames = scanForClassNames(configClass);  
        // 创建Bean的定义:  
        this.beans = createBeanDefinitions(beanClassNames);  
    }  
}
```

```
}  
}
```

下一步是创建Bean的实例，同时注入强依赖。此阶段必须检测循环依赖。检测循环依赖其实非常简单，就是定义一个 `Set<String>` 跟踪当前正在创建的所有Bean的名称：

```
public class AnnotationConfigApplicationContext {  
    Set<String> creatingBeanNames;  
    ...  
}
```

创建Bean实例我们用方法 `createBeanAsEarlySingleton()` 实现，在方法开始处检测循环依赖：

```
// 创建一个Bean，但不进行字段和方法级别的注入。如果创建的Bean不是Configuration，  
// 则在构造方法/工厂方法中注入的依赖Bean会自动创建  
public Object createBeanAsEarlySingleton(BeanDefinition def) {  
    if (!this.creatingBeanNames.add(def.getName())) {  
        // 检测到重复创建Bean导致的循环依赖：  
        throw new UnsatisfiedDependencyException();  
    }  
    ...  
}
```

由于 `@Configuration` 标识的Bean实际上是工厂，它们必须先实例化，才能实例化其他普通Bean，所以我们先把 `@Configuration` 标识的Bean创建出来，再创建普通Bean：

```
public AnnotationConfigApplicationContext(Class<?> configClass,  
PropertyResolver propertyResolver) {  
    this.propertyResolver = propertyResolver;  
    // 扫描获取所有Bean的Class类型：  
    Set<String> beanClassNames = scanForClassNames(configClass);  
    // 创建Bean的定义：  
    this.beans = createBeanDefinitions(beanClassNames);  
  
    // 创建BeanName检测循环依赖：  
    this.creatingBeanNames = new HashSet<>();  
  
    // 创建@Configuration类型的Bean：  
    this.beans.values().stream()  
        // 过滤出@Configuration:
```

```
        .filter(this::isConfigurationDefinition).sorted().map(def -> {
            // 创建Bean实例:
            createBeanAsEarlySingleton(def);
            return def.getName();
        }).collect(Collectors.toList());

// 创建其他普通Bean:
List<BeanDefinition> defs = this.beans.values().stream()
    // 过滤出instance==null的BeanDefinition:
    .filter(def -> def.getInstance() == null)
    .sorted().collect(Collectors.toList());
// 依次创建Bean实例:
defs.forEach(def -> {
    // 如果Bean未被创建(可能在其他Bean的构造方法注入前被创建):
    if (def.getInstance() == null) {
        // 创建Bean:
        createBeanAsEarlySingleton(def);
    }
});
}
```

剩下的工作就是把 `createBeanAsEarlySingleton()` 补充完整:

```
public Object createBeanAsEarlySingleton(BeanDefinition def) {
    // 检测循环依赖:
    if (!this.creatingBeanNames.add(def.getName())) {
        throw new UnsatisfiedDependencyException();
    }

    // 创建方式: 构造方法或工厂方法:
    Executable createFn = def.getFactoryName() == null ?
        def.getConstructor() : def.getFactoryMethod();

    // 创建参数:
    Parameter[] parameters = createFn.getParameters();
    Object[] args = new Object[parameters.length];
    for (int i = 0; i < parameters.length; i++) {
        // 从参数获取@Value和@Autowired:
        Value value = ...
        Autowired autowired = ...
        // 检查Value和Autowired
    }
}
```



```

    ...
    // 参数类型:
    Class<?> type = param.getType();
    if (value != null) {
        // 参数设置为查询的@Value:
        args[i] =
this.propertyResolver.getRequiredProperty(value.value(), type);
    } else {
        // 参数是@Autowired, 查找依赖的BeanDefinition:
        BeanDefinition dependsOnDef = name.isEmpty() ?
findBeanDefinition(type) : findBeanDefinition(name, type);
        // 获取依赖Bean的实例:
        Object autowiredBeanInstance = dependsOnDef.getInstance();
        if (autowiredBeanInstance == null) {
            // 当前依赖Bean尚未初始化, 递归调用初始化该依赖Bean:
            autowiredBeanInstance =
createBeanAsEarlySingleton(dependsOnDef);
        }
        // 参数设置为依赖的Bean实例:
        args[i] = autowiredBeanInstance;
    }
}
// 已拿到所有方法参数, 创建Bean实例:
Object instance = ...
// 设置实例到BeanDefinition:
def.setInstance(instance);
// 返回实例:
return def.getInstance();
}

```

注意到递归调用:

```

public Object createBeanAsEarlySingleton(BeanDefinition def) {
    ...
    Object[] args = new Object[parameters.length];
    for (int i = 0; i < parameters.length; i++) {
        ...
        // 获取依赖Bean的实例:
        Object autowiredBeanInstance = dependsOnDef.getInstance();
        if (autowiredBeanInstance == null && !isConfiguration) {
            // 当前依赖Bean尚未初始化, 递归调用初始化该依赖Bean:

```

```
        autowiredBeanInstance =  
createBeanAsEarlySingleton(dependsOnDef);  
    }  
    ...  
}  
...  
}
```

假设如下的Bean依赖：

```
@Component  
class A {  
    // 依赖B,C:  
    A(@Autowired B, @Autowired C) {}  
}  
  
@Component  
class B {  
    // 依赖C:  
    B(@Autowired C) {}  
}  
  
@Component  
class C {  
    // 无依赖:  
    C() {}  
}
```

如果按照A、B、C的顺序创建Bean实例，那么系统流程如下：

1. 准备创建A;
2. 检测到依赖B：未就绪;
 1. 准备创建B:
 2. 检测到依赖C：未就绪;
 1. 准备创建C;
 2. 完成创建C;
 3. 完成创建B;
3. 检测到依赖C，已就绪;

4. 完成创建A。

如果按照B、C、A的顺序创建Bean实例，那么系统流程如下：

1. 准备创建B;
2. 检测到依赖C：未就绪;
 1. 准备创建C;
 2. 完成创建C;
3. 完成创建B;
4. 准备创建A;
5. 检测到依赖B，已就绪;
6. 检测到依赖C，已就绪;
7. 完成创建A。

可见无论以什么顺序创建，C总是最先被实例化，A总是最后被实例化。

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

初始化Bean

原文链接

在创建Bean实例的过程中，我们已经完成了强依赖的注入。下一步，是根据Setter方法和字段完成弱依赖注入，接着调用用 `@PostConstruct` 标注的init方法，就完成了所有Bean的初始化。

这一步相对比较简单，因为只涉及到查找依赖的 `@Value` 和 `@Autowired`，然后用反射完成调用即可：

```
public AnnotationConfigApplicationContext(Class<?> configClass,
PropertyResolver propertyResolver) {
    ...

    // 通过字段和set方法注入依赖：
    this.beans.values().forEach(def -> {
        injectBean(def);
    });

    // 调用init方法：
    this.beans.values().forEach(def -> {
        initBean(def);
    });
}
```

使用Setter方法和字段注入时，要注意一点，就是不仅要在当前类查找，还要在父类查找，因为有些 `@Autowired` 写在父类，所有子类都可使用，这样更方便。注入弱依赖代码如下：

```
// 在当前类及父类进行字段和方法注入：
void injectProperties(BeanDefinition def, Class<?> clazz, Object bean) {
    // 在当前类查找Field和Method并注入：
    for (Field f : clazz.getDeclaredFields()) {
        tryInjectProperties(def, clazz, bean, f);
    }
    for (Method m : clazz.getDeclaredMethods()) {
        tryInjectProperties(def, clazz, bean, m);
    }
    // 在父类查找Field和Method并注入：
    Class<?> superClass = clazz.getSuperclass();
}
```

```
    if (superClazz != null) {
        // 递归调用:
        injectProperties(def, superClazz, bean);
    }
}

// 注入单个属性
void tryInjectProperties(BeanDefinition def, Class<?> clazz, Object bean,
    AccessibleObject acc) {
    ...
}
```

弱依赖注入完成后，再循环一遍所有的 `BeanDefinition`，对其调用 `init` 方法，完成最后一步初始化：

```
void initBean(BeanDefinition def) {
    // 调用init方法:
    callMethod(def.getInstance(), def.getInitMethod(),
    def.getInitMethodName());
}
```

处理 `@PreDestroy` 方法更简单，在 `ApplicationContext` 关闭时遍历所有Bean，调用 `destroy` 方法即可。

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

实现BeanPostProcessor

原文链接

现在，我们已经完成了扫描Class名称、创建BeanDefinition、创建Bean实例、初始化Bean，理论上一个可用的IoC容器就已经就绪。

然而，`BeanPostProcessor` 的出现改变了这一切。Spring允许用户自定义一种特殊的Bean，即实现了 `BeanPostProcessor` 接口，它有什么用呢？其实就是替换Bean。我们举个例子，下面的代码是基于Spring代码：

```
@Configuration
@ComponentScan
public class AppConfig {

    public static void main(String[] args) {
        var ctx = new AnnotationConfigApplicationContext(AppConfig.class);
        // 可以获取到ZonedDateTime:
        ZonedDateTime dt = ctx.getBean(ZonedDateTime.class);
        System.out.println(dt);
        // 错误:NoSuchBeanDefinitionException:
        System.out.println(ctx.getBean(LocalDateTime.class));
    }

    // 创建LocalDateTime实例
    @Bean
    public LocalDateTime localDateTime() {
        return LocalDateTime.now();
    }

    // 实现一个BeanPostProcessor
    @Bean
    BeanPostProcessor replaceLocalDateTime() {
        return new BeanPostProcessor() {
            @Override
            public Object postProcessBeforeInitialization(Object bean,
String beanName) throws BeansException {
                // 将LocalDateTime类型实例替换为ZonedDateTime类型实例:
                if (bean instanceof LocalDateTime) {
                    return ZonedDateTime.now();
                }
            }
        };
    }
}
```

```

        }
        return bean;
    }
};
}
}

```

运行可知，我们定义的 `@Bean` 类型明明是 `LocalDateTime` 类型，但却被另一个 `BeanPostProcessor` 替换成了 `ZonedDateTime`，于是，调用 `getBean(ZonedDateTime.class)` 可以拿到替换后的Bean，调用 `getBean(LocalDateTime.class)` 会报错，提示找不到Bean。那么原始的Bean哪去了？答案是被 `BeanPostProcessor` 扔掉了。

可见，`BeanPostProcessor` 是一种特殊Bean，它的作用是根据条件替换某些Bean。上述的例子中，`LocalDateTime` 被替换为 `ZonedDateTime` 其实没啥意义，但实际应用中，把原始Bean替换为代理后的Bean是非常常见的，比如下面的基于Spring的代码：

```

@Configuration
@ComponentScan
public class AppConfig {

    public static void main(String[] args) {
        var ctx = new AnnotationConfigApplicationContext(AppConfig.class);
        UserService u = ctx.getBean(UserService.class);
        System.out.println(u.getClass().getSimpleName()); //
        UserServiceProxy
        u.register("bob@example.com", "bob12345");
    }

    @Bean
    BeanPostProcessor createProxy() {
        return new BeanPostProcessor() {
            @Override
            public Object postProcessBeforeInitialization(Object bean,
String beanName) throws BeansException {
                // 实现事务功能:
                if (bean instanceof UserService u) {
                    return new UserServiceProxy(u);
                }
                return bean;
            }
        }
    }
}

```

```
    }  
}  
  
@Component  
class UserService {  
    public void register(String email, String password) {  
        System.out.println("INSERT INTO ...");  
    }  
}  
  
// 代理类:  
class UserServiceProxy extends UserService {  
    UserService target;  
  
    public UserServiceProxy(UserService target) {  
        this.target = target;  
    }  
  
    @Override  
    public void register(String email, String password) {  
        System.out.println("begin tx");  
        target.register(email, password);  
        System.out.println("commit tx");  
    }  
}
```

如果执行上述代码，打印出的Bean类型不是 `UserService`，而是 `UserServiceProxy`，因此，调用 `register()` 会打印出 `begin tx` 和 `commit tx`，说明“事务”生效了。

迄今为止，创建Proxy似乎没有什么影响。让我们把代码再按实际情况扩展一下，`UserService` 是用户编写的业务代码，需要注入 `JdbcTemplate`：

```
@Component  
class UserService {  
    @Autowired JdbcTemplate jdbcTemplate;  
  
    public void register(String email, String password) {  
        jdbcTemplate.update("INSERT INTO ...");  
    }  
}
```



```
}  
}
```

而 `PostBeanProcessor` 一般由框架本身提供事务功能，所以它会动态创建一个 `UserServiceProxy`：

```
class UserServiceProxy extends UserService {  
    UserService target;  
  
    public UserServiceProxy(UserService target) {  
        this.target = target;  
    }  
  
    @Override  
    public void register(String email, String password) {  
        System.out.println("begin tx");  
        target.register(email, password);  
        System.out.println("commit tx");  
    }  
}
```

调用用户注册的页面由 `MvcController` 控制，因此，将 `UserService` 注入到 `MvcController`：

```
@Controller  
class MvcController {  
    @Autowired UserService userService;  
  
    @PostMapping("/register")  
    void register() {  
        userService.register(...);  
    }  
}
```

一开始，由IoC容器创建的Bean包括：

- JdbcTemplate
- UserService
- MvcController

接着，由于 `BeanPostProcessor` 的介入，原始的 `UserService` 被替换为

`UserServiceProxy`：

- `JdbcTemplate`
- `UserServiceProxy`
- `MvcController`

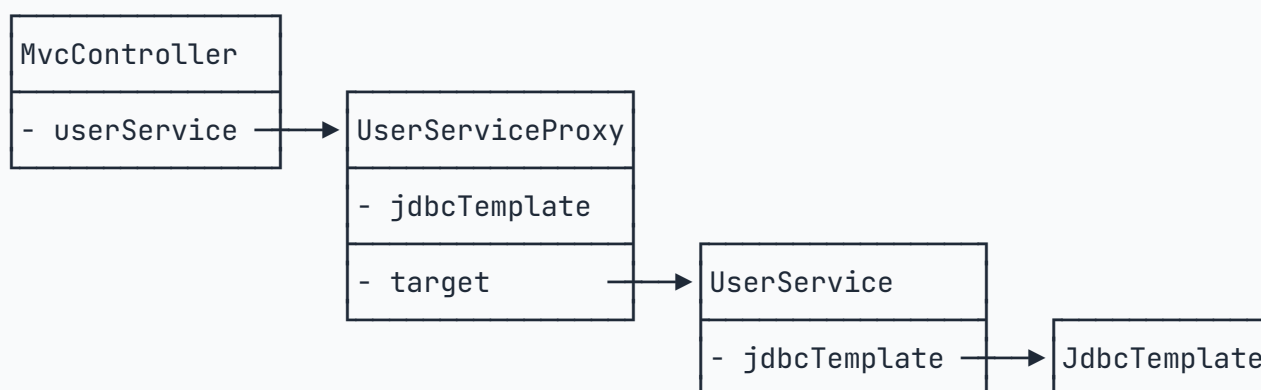
那么问题来了：注意到 `UserServiceProxy` 是从 `UserService` 继承的，它也有一个 `@Autowired JdbcTemplate`，那 `JdbcTemplate` 实例应注入到原始的 `UserService` 还是 `UserServiceProxy`？

从业务逻辑出发，`JdbcTemplate` 实例必须注入到原始的 `UserService`，否则，代理类 `UserServiceProxy` 执行 `target.register()` 时，相当于对原始的 `UserService` 调用 `register()` 方法，如果 `JdbcTemplate` 没有注入，将直接报 `NullPointerException` 错误。

这时第二个问题又来了：`MvcController` 需要注入的 `UserService`，应该是原始的 `UserService` 还是 `UserServiceProxy`？

还是从业务逻辑出发，`MvcController` 需要注入的 `UserService` 必须是 `UserServiceProxy`，否则，事务不起作用。

我们用图描述一下注入关系：



注意到上图的 `UserService` 已经脱离了IoC容器的管理，因为此时 `UserService` 对应的 `BeanDefinition` 中，存放的instance是 `UserServiceProxy`。

可见，引入 `BeanPostProcessor` 可以实现Proxy机制，但也让依赖注入变得更加复杂。

但是我们仔细分析依赖关系，还是可以总结出两条原则：

1. 一个Bean如果被Proxy替换，则依赖它的Bean应注入Proxy，即上图的 `MvcController` 应注入 `UserServiceProxy` ；
2. 一个Bean如果被Proxy替换，如果要注入依赖，则应该注入到原始对象，即上图的 `JdbcTemplate` 应注入到原始的 `UserService` 。

基于这个原则，要满足条件1是很容易的，因为只要创建Bean完成后，立刻调用 `BeanPostProcessor` 就实现了替换，后续其他Bean引用的肯定就是Proxy了。先改造创建Bean的流程，在创建 `@Configuration` 后，接着创建 `BeanPostProcessor` ，再创建其他普通Bean：

```
public AnnotationConfigApplicationContext(Class<?> configClass,
PropertyResolver propertyResolver) {
    ...
    // 创建@Configuration类型的Bean:
    this.beans.values().stream()
        // 过滤出@Configuration:
        .filter(this::isConfigurationDefinition).sorted().map(def -> {
            createBeanAsEarlySingleton(def);
            return def.getName();
        }).collect(Collectors.toList());

    // 创建BeanPostProcessor类型的Bean:
    List<BeanPostProcessor> processors = this.beans.values().stream()
        // 过滤出BeanPostProcessor:
        .filter(this::isBeanPostProcessorDefinition)
        // 排序:
        .sorted()
        // 创建BeanPostProcessor实例:
        .map(def -> {
            return (BeanPostProcessor) createBeanAsEarlySingleton(def);
        }).collect(Collectors.toList());
    this.beanPostProcessors.addAll(processors);

    // 创建其他普通Bean:
    createNormalBeans();
    ...
}
```

再继续修改 `createBeanAsEarlySingleton()` ，创建Bean实例后，调用 `BeanPostProcessor` 处理：

```
public Object createBeanAsEarlySingleton(BeanDefinition def) {  
    ...  
  
    // 创建Bean实例:  
    Object instance = ...;  
    def.setInstance(instance);  
  
    // 调用BeanPostProcessor处理Bean:  
    for (BeanPostProcessor processor : beanPostProcessors) {  
        Object processed =  
processor.postProcessBeforeInitialization(def.getInstance(),  
def.getName());  
        // 如果一个BeanPostProcessor替换了原始Bean, 则更新Bean的引用:  
        if (def.getInstance() != processed) {  
            def.setInstance(processed);  
        }  
    }  
    return def.getInstance();  
}
```

现在, 如果一个Bean被替换为Proxy, 那么 `BeanDefinition` 中的 `instance` 已经是Proxy了, 这时, 对这个Bean进行依赖注入会有问题, 因为注入的是Proxy而不是原始Bean, 怎么办?

这时我们要思考原始Bean去哪了? 原始Bean实际上是被 `BeanPostProcessor` 给丢了! 如果 `BeanPostProcessor` 能保存原始Bean, 那么, 注入前先找到原始Bean, 就可以把依赖正确地注入给原始Bean。我们给 `BeanPostProcessor` 加一个 `postProcessOnSetProperty()` 方法, 让它返回原始Bean:

```
public interface BeanPostProcessor {  
    // 注入依赖时, 应该使用的Bean实例:  
    default Object postProcessOnSetProperty(Object bean, String beanName) {  
        return bean;  
    }  
}
```

再继续把 `injectBean()` 改一下, 不要直接拿 `BeanDefinition.getInstance()`, 而是拿到原始Bean:

```
void injectBean(BeanDefinition def) {  
    // 获取Bean实例，或被代理的原始实例：  
    Object beanInstance = getProxiedInstance(def);  
    try {  
        injectProperties(def, def.getBeanClass(), beanInstance);  
    } catch (ReflectiveOperationException e) {  
        throw new BeanCreationException(e);  
    }  
}
```

`getProxiedInstance()` 就是为了获取原始Bean:

```
Object getProxiedInstance(BeanDefinition def) {  
    Object beanInstance = def.getInstance();  
    // 如果Proxy改变了原始Bean，又希望注入到原始Bean，则由BeanPostProcessor指定  
    // 原始Bean：  
    List<BeanPostProcessor> reversedBeanPostProcessors = new ArrayList<>  
    (this.beanPostProcessors);  
    Collections.reverse(reversedBeanPostProcessors);  
    for (BeanPostProcessor beanPostProcessor : reversedBeanPostProcessors)  
    {  
        Object restoredInstance =  
        beanPostProcessor.postProcessOnSetProperty(beanInstance, def.getName());  
        if (restoredInstance != beanInstance) {  
            beanInstance = restoredInstance;  
        }  
    }  
    return beanInstance;  
}
```

这里我们还能处理多次代理的情况，即一个原始Bean，比如 `UserService`，被一个事务处理的 `BeanPostProcessor` 代理为 `UserServiceTx`，又被一个性能监控的 `BeanPostProcessor` 代理为 `UserServiceMetric`，还原的时候，对 `BeanPostProcessor` 做一个倒序，先还原为 `UserServiceTx`，再还原为 `UserService`。

测试

我们可以写一个测试来验证Bean的注入是否正确。先定义原始Bean：

```
@Component
public class OriginBean {
    @Value("${app.title}")
    public String name;

    @Value("${app.version}")
    public String version;

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

通过 `FirstProxyBeanPostProcessor` 代理为 `FirstProxyBean` :

```
@Order(100)
@Component
public class FirstProxyBeanPostProcessor implements BeanPostProcessor {
    // 保存原始Bean:
    Map<String, Object> originBeans = new HashMap<>();

    @Override
    public Object postProcessBeforeInitialization(Object bean, String
beanName) {
        if (OriginBean.class.isAssignableFrom(bean.getClass())) {
            // 检测到OriginBean,创建FirstProxyBean:
            var proxy = new FirstProxyBean((OriginBean) bean);
            // 保存原始Bean:
            originBeans.put(beanName, bean);
            // 返回Proxy:
            return proxy;
        }
        return bean;
    }

    @Override
    public Object postProcessOnSetProperty(Object bean, String beanName) {
        Object origin = originBeans.get(beanName);
        if (origin != null) {
            // 存在原始Bean时,返回原始Bean:

```

```
        return origin;
    }
    return bean;
}
}

// 代理Bean:
class FirstProxyBean extends OriginBean {
    final OriginBean target;

    public FirstProxyBean(OriginBean target) {
        this.target = target;
    }
}
```

通过 `SecondProxyBeanPostProcessor` 代理为 `SecondProxyBean` :

```
@Order(200)
@Component
public class SecondProxyBeanPostProcessor implements BeanPostProcessor {
    // 保存原始Bean:
    Map<String, Object> originBeans = new HashMap<>();

    @Override
    public Object postProcessBeforeInitialization(Object bean, String
beanName) {
        if (OriginBean.class.isAssignableFrom(bean.getClass())) {
            // 检测到OriginBean,创建SecondProxyBean:
            var proxy = new SecondProxyBean((OriginBean) bean);
            // 保存原始Bean:
            originBeans.put(beanName, bean);
            // 返回Proxy:
            return proxy;
        }
        return bean;
    }

    @Override
    public Object postProcessOnSetProperty(Object bean, String beanName) {
        Object origin = originBeans.get(beanName);
        if (origin != null) {
```

```
        // 存在原始Bean时,返回原始Bean:
        return origin;
    }
    return bean;
}

// 代理Bean:
class SecondProxyBean extends OriginBean {
    final OriginBean target;

    public SecondProxyBean(OriginBean target) {
        this.target = target;
    }
}
```

定义一个Bean，用于检测是否注入了Proxy：

```
@Component
public class InjectProxyOnConstructorBean {
    public final OriginBean injected;

    public InjectProxyOnConstructorBean(@Autowired OriginBean injected) {
        this.injected = injected;
    }
}
```

测试代码如下：

```
var ctx = new AnnotationConfigApplicationContext(ScanApplication.class,
createPropertyResolver());

// 获取OriginBean的实例,此处获取的应该是SendProxyBeanProxy:
OriginBean proxy = ctx.getBean(OriginBean.class);
assertSame(SecondProxyBean.class, proxy.getClass());

// proxy的name和version字段并没有被注入:
assertNull(proxy.name);
assertNull(proxy.version);
```



```
// 但是调用proxy的getName()会最终调用原始Bean的getName(),从而返回正确的值:
assertEquals("Scan App", proxy.getName());

// 获取InjectProxyOnConstructorBean实例:
var inject = ctx.getBean(InjectProxyOnConstructorBean.class);
// 注入的OriginBean应该为Proxy, 而且和前面返回的proxy是同一实例:
assertSame(proxy, inject.injected);
```

从上面的测试代码我们也能看出, 对于使用Proxy模式的Bean来说, 正常的方法调用对用户是透明的, 但是, 直接访问Bean注入的字段, 如果获取的是Proxy, 则字段全部为 `null`, 因为注入并没有发生在Proxy, 而是原始Bean。这也是为什么当我们需要访问某个注入的Bean时, 总是调用方法而不是直接访问字段:

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        // 错误:不要直接访问UserService的字段,因为如果UserService被代理,则返回
        null:
        ZoneId zoneId = userService.zoneId;
        // 正确:通过方法访问UserService的字段,无论是否被代理,返回值均是正确的:
        ZoneId zoneId = userService.getZoneId();
        ...
    }
}
```

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

评论

完成IoC容器

原文链接

现在，我们已经完成了IoC容器的基本功能。最后的收尾工作主要是提取接口。先定义给用户使用的 `ApplicationContext` 接口：

```
public interface ApplicationContext extends AutoCloseable {

    // 是否存在指定name的Bean?
    boolean containsBean(String name);

    // 根据name返回唯一Bean, 未找到抛出NoSuchBeanDefinitionException
    <T> T getBean(String name);

    // 根据name返回唯一Bean, 未找到抛出NoSuchBeanDefinitionException
    <T> T getBean(String name, Class<T> requiredType);

    // 根据type返回唯一Bean, 未找到抛出NoSuchBeanDefinitionException
    <T> T getBean(Class<T> requiredType);

    // 根据type返回一组Bean, 未找到返回空List
    <T> List<T> getBeans(Class<T> requiredType);

    // 关闭并执行所有bean的destroy方法
    void close();
}
```

再定义一个给Framework级别的代码用的 `ConfigurableApplicationContext` 接口：

```
public interface ConfigurableApplicationContext extends ApplicationContext {

    List<BeanDefinition> findBeanDefinitions(Class<?> type);

    @Nullable
    BeanDefinition findBeanDefinition(Class<?> type);

    @Nullable
    BeanDefinition findBeanDefinition(String name);
}
```

```
@Nullable
BeanDefinition findBeanDefinition(String name, Class<?> requiredType);

Object createBeanAsEarlySingleton(BeaDefinition def);
}
```

让 `AnnotationConfigApplicationContext` 实现接口：

```
public class AnnotationConfigApplicationContext implements
ConfigurableApplicationContext {
    ...
}
```

顺便在 `close()` 方法中把Bean的 `destroy` 方法执行了。

最后加一个 `ApplicationUtils` 类，目的是能通过 `getRequiredApplicationContext()` 方法随时获取到 `ApplicationContext` 实例。

搞定 `summer-context` 模块！

有的同学可能会问，为什么我们用了不到1000行核心代码，就实现了 `ApplicationContext` ？如果查看Spring的源码，可以看到，光是层次结构，就令人眼花缭乱：

```
BeanFactory
  HierarchicalBeanFactory
    ConfigurableBeanFactory
      AbstractBeanFactory
        AbstractAutowireCapableBeanFactory
          DefaultListableBeanFactory
ApplicationContext
  ConfigurableApplicationContext
    AbstractApplicationContext
      AbstractRefreshableApplicationContext
        AbstractXmlApplicationContext
          ClassPathXmlApplicationContext
          FileSystemXmlApplicationContext
        GenericApplicationContext
          AnnotationConfigApplicationContext
```

```
GenericXmlApplicationContext  
StaticApplicationContext
```

其实根本原因是我们大幅简化了需求。Spring最早提供了 `BeanFactory` 和 `ApplicationContext` 两种容器，前者是懒加载，后者是立刻初始化所有Bean。懒加载的特性会导致依赖注入变得更加复杂，虽然 `BeanFactory` 在实际项目中并没有什么卵用。然而一旦发布了接口，处于兼容性考虑，就没法再收回去了。再考虑到Spring最早采用XML配置，后来采用Annotation配置，还允许混合配置，这样一来，早期发布的 `XmlApplicationContext` 不能动，新的Annotation配置就必须添加新的实现类，所以，代码的复杂度随着需求增加而增加，保持兼容性又会导致需要更多的代码来实现新功能。

所以，没事不要瞎提需求。

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

实现AOP

原文链接

实现了IoC容器后，我们继续实现AOP功能。

AOP即Aspect Oriented Programming，面向切面编程，它本质上就是一个Proxy模式，只不过可以让IoC容器在运行时再组合起来，而不是事先自己用Proxy模式写死了。而实现Proxy模式的核心是拦截目标Bean的方法调用。

既然原理是方法拦截，那么AOP的实现方式不外乎以下几种：

1. 编译期：在编译时，由编译器把切面调用编译进字节码，这种方式需要定义新的关键字并扩展编译器，AspectJ就扩展了Java编译器，使用关键字aspect来实现织入；
2. 类加载器：在目标类被装载到JVM时，通过一个特殊的类加载器，对目标类的字节码重新“增强”；
3. 运行期：目标对象和切面都是普通Java类，通过JVM的动态代理功能或者第三方库实现运行期动态织入。

从复杂度看，最简单的是方案3，因为不涉及到任何JVM底层。

方案3又有两种实现方式：

1. 使用Java标准库的动态代理机制，不过仅支持对接口代理，无法对具体类实现代理；
2. 使用CGLIB或Javassist这些第三方库，通过动态生成字节码，可以对具体类实现代理。

那么Spring的实现方式是啥？Spring实际上内置了多种代理机制，如果一个Bean声明的类型是接口，那么Spring直接使用Java标准库实现对接口的代理，如果一个Bean声明的类型是Class，那么Spring就使用CGLIB动态生成字节码实现代理。

除了实现代理外，还得有一套机制让用户能定义代理。Spring又提供了多种方式：

1. 用AspectJ的语法来定义AOP，比如 `execution(public * com.itranswarp.service.*.*(..))`；
2. 用注解来定义AOP，比如用 `@Transactional` 表示开启事务。

用表达式匹配，很容易漏掉或者打击面太大。用注解无疑是最简单的，因为这样被装配的Bean自己能清清楚楚地知道自己被安排了。因此，在Summer Framework中，我们只支持Annotation模式的AOP机制，并且采用动态生成字节码的方式实现。

明确了需求，我们来看如何实现动态生成字节码。Spring采用的是CGLIB，因此我们去CGLIB首页看一下，不看不知道，一看吓一跳：

cglib is unmaintained ... migrating to something like ByteBuddy.

原来CGLIB已经不维护了，建议使用ByteBuddy。既然如此，我们就选择ByteBuddy实现AOP吧。

比较一下Spring Framework和Summer Framework对AOP的支持：

	Spring Framework	Summer Framework
AspectJ方式	支持	不支持
Annotation方式	支持	支持
代理接口	支持	不支持
代理类	支持	支持
实现机制	CGLIB	ByteBuddy

下面我们就来准备实现AOP。

评论

实现ProxyResolver

原文链接

为了实现AOP，我们先思考如何在IoC容器中实现一个动态代理。

在IoC容器中，实现动态代理需要用户提供两个Bean：

1. 原始Bean，即需要被代理的Bean；
2. 拦截器，即拦截了目标Bean的方法后，会自动调用拦截器实现代理功能。

拦截器需要定义接口，这里我们直接用Java标准库的 `InvocationHandler`，免去了自定义接口。

假定我们已经从IoC容器中获取了原始Bean与实现了 `InvocationHandler` 的拦截器Bean，那么就可以编写一个 `ProxyResolver` 来实现AOP代理。

从ByteBuddy的官网上搜索很容易找到相关代码，我们整理为 `createProxy()` 方法：

```
public class ProxyResolver {
    // ByteBuddy实例：
    ByteBuddy byteBuddy = new ByteBuddy();

    // 传入原始Bean、拦截器，返回代理后的实例：
    public <T> T createProxy(T bean, InvocationHandler handler) {
        // 目标Bean的Class类型：
        Class<?> targetClass = bean.getClass();
        // 动态创建Proxy的Class：
        Class<?> proxyClass = this.byteBuddy
            // 子类用默认无参数构造方法：
            .subclass(targetClass,
                ConstructorStrategy.Default.DEFAULT_CONSTRUCTOR)
            // 拦截所有public方法：
            .method(ElementMatchers.isPublic()).intercept(InvocationHandlerAdapter.of(
                // 新的拦截器实例：
                new InvocationHandler() {
                    public Object invoke(Object proxy, Method
                        method, Object[] args) throws Throwable {
                        // 将方法调用代理至原始Bean：

```

```
                return handler.invoke(bean, method, args);
            }
        })))
    // 生成字节码:
    .make()
    // 加载字节码:
    .load(targetClass.getClassLoader()).getLoaded();
// 创建Proxy实例:
Object proxy;
try {
    proxy = proxyClass.getConstructor().newInstance();
} catch (RuntimeException e) {
    throw e;
} catch (Exception e) {
    throw new RuntimeException(e);
}
return (T) proxy;
}
}
```

注意 `InvocationHandler` 有两层：外层的 `invoke()` 传入的Object是Proxy实例，内层的 `invoke()` 将调用转发至原始Bean。

一共大约50行代码，我们就实现了AOP功能。有点不敢相信，赶快写个测试看看效果。

先定义一个 `OriginBean`：

```
public class OriginBean {
    public String name;

    @Polite
    public String hello() {
        return "Hello, " + name + ".";
    }

    public String morning() {
        return "Morning, " + name + ".";
    }
}
```


我们要实现的AOP功能是增强带 `@Polite` 注解的方法，把返回值 `Hello, Bob.` 改为 `Hello, Bob!`，让欢迎气氛更强烈一点，因此，编写一个 `InvocationHandler`：

```
public class PoliteInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object bean, Method method, Object[] args) throws
        Throwable {
        // 修改标记了@Polite的方法返回值:
        if (method.getAnnotation(Polite.class) != null) {
            String ret = (String) method.invoke(bean, args);
            if (ret.endsWith(".")) {
                ret = ret.substring(0, ret.length() - 1) + "!";
            }
            return ret;
        }
        return method.invoke(bean, args);
    }
}
```

测试代码：

```
// 原始Bean:
OriginBean origin = new OriginBean();
origin.name = "Bob";
// 调用原始Bean的hello():
assertEquals("Hello, Bob.", origin.hello());

// 创建Proxy:
OriginBean proxy = new ProxyResolver().createProxy(origin, new
    PoliteInvocationHandler());

// Proxy类名,类似OriginBean$ByteBuddy$9hQwRy3T:
System.out.println(proxy.getClass().getName());

// Proxy类与OriginBean.class不同:
assertNotSame(OriginBean.class, proxy.getClass());
// proxy实例的name字段应为null:
assertNull(proxy.name);

// 调用带@Polite的方法:
```

```
assertEquals("Hello, Bob!", proxy.hello());  
// 调用不带@Polite的方法:  
assertEquals("Morning, Bob.", proxy.morning());
```

测试通过，本节到此收工。

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

实现Around

原文链接

现在我们已经实现了ProxyResolver，下一步，实现完整的AOP就很容易了。

我们先从客户端代码入手，看看应当怎么装配AOP。

首先，客户端需要定义一个原始Bean，例如 `OriginBean`，用 `@Around` 注解标注：

```
@Component
@Around("aroundInvocationHandler")
public class OriginBean {

    @Value("${customer.name}")
    public String name;

    @Polite
    public String hello() {
        return "Hello, " + name + ".";
    }

    public String morning() {
        return "Morning, " + name + ".";
    }
}
```

`@Around` 注解的值 `aroundInvocationHandler` 指出应该按什么名字查找拦截器，因此，客户端应再定义一个 `AroundInvocationHandler`：

```
@Component
public class AroundInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
        // 拦截标记了@Polite的方法返回值：
        if (method.getAnnotation(Polite.class) != null) {
            String ret = (String) method.invoke(proxy, args);
            if (ret.endsWith(".")) {
                ret = ret.substring(0, ret.length() - 1) + "!";
            }
        }
        return method.invoke(proxy, args);
    }
}
```

```
        }
        return ret;
    }
    return method.invoke(proxy, args);
}
}
```

有了原始Bean、拦截器，就可以在IoC容器中装配AOP：

```
@Configuration
@ComponentScan
public class AroundApplication {
    @Bean
    AroundProxyBeanPostProcessor createAroundProxyBeanPostProcessor() {
        return new AroundProxyBeanPostProcessor();
    }
}
```

注意到装配AOP是通过 `AroundProxyBeanPostProcessor` 实现的，而这个类是由Framework提供，客户端并不需要自己实现。因此，我们需要开发一个 `AroundProxyBeanPostProcessor`：

```
public class AroundProxyBeanPostProcessor implements BeanPostProcessor {

    Map<String, Object> originBeans = new HashMap<>();

    public Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
        Class<?> beanClass = bean.getClass();
        // 检测@Around注解:
        Around anno = beanClass.getAnnotation(Around.class);
        if (anno != null) {
            String handlerName;
            try {
                handlerName = (String)
anno.annotationType().getMethod("value").invoke(anno);
            } catch (ReflectiveOperationException e) {
                throw new AopConfigException();
            }
            Object proxy = createProxy(beanClass, bean, handlerName);
            originBeans.put(beanName, bean);
        }
    }
}
```

```

        return proxy;
    } else {
        return bean;
    }
}

Object createProxy(Class<?> beanClass, Object bean, String handlerName)
{
    ConfigurableApplicationContext ctx =
    (ConfigurableApplicationContext)
    ApplicationContextUtils.getRequiredApplicationContext();
    BeanDefinition def = ctx.getBeanDefinition(handlerName);
    if (def == null) {
        throw new AopConfigException();
    }
    Object handlerBean = def.getInstance();
    if (handlerBean == null) {
        handlerBean = ctx.createBeanAsEarlySingleton(def);
    }
    if (handlerBean instanceof InvocationHandler handler) {
        return ProxyResolver.getInstance().createProxy(bean, handler);
    } else {
        throw new AopConfigException();
    }
}

@Override
public Object postProcessOnSetProperty(Object bean, String beanName) {
    Object origin = this.originBeans.get(beanName);
    return origin != null ? origin : bean;
}
}

```

上述 `AroundProxyBeanPostProcessor` 的机制非常简单：检测每个Bean实例是否带有 `@Around` 注解，如果有，就根据注解的值查找Bean作为 `InvocationHandler`，最后创建Proxy，返回前保存了原始Bean的引用，因为IoC容器在后续的注入阶段要把相关依赖和值注入到原始Bean。

总结一下，Summer Framework提供的包括：

- `Around` 注解；

- `AroundProxyBeanPostProcessor` 实现AOP。

客户端代码需要提供的包括：

- 带 `@Around` 注解的原始Bean；
- 实现 `InvocationHandler` 的Bean，名字与 `@Around` 注解value保持一致。

没有额外的要求了。

实现Before和After

我们再继续思考，Spring提供的AOP拦截器，有Around、Before和After等好几种。如何实现Before和After拦截？

实际上Around拦截本身就包含了Before和After拦截，我们没必要去修改 `ProxyResolver`，只需要用Adapter模式提供两个拦截器模版，一个是 `BeforeInvocationHandlerAdapter`：

```
public abstract class BeforeInvocationHandlerAdapter implements
InvocationHandler {

    public abstract void before(Object proxy, Method method, Object[]
args);

    @Override
    public final Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        before(proxy, method, args);
        return method.invoke(proxy, args);
    }
}
```

客户端提供的 `InvocationHandler` 只需继承自 `BeforeInvocationHandlerAdapter`，自然就需要覆写 `before()` 方法，实现了Before拦截。

After拦截也是一个拦截器模版：

```
public abstract class AfterInvocationHandlerAdapter implements
InvocationHandler {
    // after允许修改方法返回值：
    public abstract Object after(Object proxy, Object returnValue, Method
```

```
method, Object[] args);

@Override
public final Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    Object ret = method.invoke(proxy, args);
    return after(proxy, ret, method, args);
}
```

扩展Annotation

截止目前，客户端只需要定义带有 `@Around` 注解的Bean，就能自动触发AOP。我们思考下Spring的事务机制，其实也是AOP拦截，不过它的注解是 `@Transactional`。如果要扩展Annotation，即能自定义注解来启动AOP，怎么做？

假设我们后续编写了一个事务模块，提供注解 `@Transactional`，那么，要启动AOP，就必须仿照 `AroundProxyBeanPostProcessor`，提供一个 `TransactionProxyBeanPostProcessor`，不过复制代码太麻烦了，我们可以改造一下 `AroundProxyBeanPostProcessor`，用泛型代码处理Annotation，先抽象出一个 `AnnotationProxyBeanPostProcessor`：

```
public abstract class AnnotationProxyBeanPostProcessor<A extends
Annotation> implements BeanPostProcessor {

    Map<String, Object> originBeans = new HashMap<>();
    Class<A> annotationClass;

    public AnnotationProxyBeanPostProcessor() {
        this.annotationClass = getParameterizedType();
    }
    ...
}
```

实现 `AroundProxyBeanPostProcessor` 就一行定义：

```
public class AroundProxyBeanPostProcessor extends
AnnotationProxyBeanPostProcessor<Around> {
}
```

后续如果我们想实现 `@Transactional` 注解，只需定义：

```
public class TransactionalProxyBeanPostProcessor extends
    AnnotationProxyBeanPostProcessor<Transactional> {
}
```

就能自动根据 `@Transactional` 启动AOP。

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

实现JDBC和事务

[原文链接](#)

我们已经实现了IoC容器和AOP功能，在此基础上增加JDBC和事务的支持就比较容易了。

Spring对JDBC数据库的支持主要包括：

- 1. 提供了一个 `JdbcTemplate` 和 `NamedParameterJdbcTemplate` 模板类，可以方便地操作JDBC；
- 2. 支持流行的ORM框架，如Hibernate、JPA等；
- 3. 支持声明式事务，只需要通过简单的注解配置即可实现事务管理。

在Summer Framework中，我们准备提供一个 `JdbcTemplate` 模板，以及声明式事务的支持。对于ORM，反正手动集成也比较容易，就不管了。

	Spring Framework	Summer Framework
JdbcTemplate	支持	支持
NamedParameterJdbcTemplate	支持	不支持
转换SQL错误码	支持	不支持
ORM	支持	不支持
手动管理事务	支持	不支持
声明式事务	支持	支持

下面开始正式开发Summer Framework的 `JdbcTemplate` 与声明式事务。

[评论](#)

实现JdbcTemplate

原文链接

本节我们来实现JdbcTemplate。在Spring中，通过JdbcTemplate，基本封装了所有JDBC操作，可以覆盖绝大多数数据库操作的场景。

配置DataSource

使用JdbcTemplate之前，我们需要配置JDBC数据源。Spring本身只提供了基础的

`DriverManagerDataSource`，但Spring Boot有一个默认配置的数据源，并采用HikariCP作为连接池。这里我们仿照Spring Boot的方式，先定义默认的数据源配置项：

```
summer:
  datasource:
    url: jdbc:sqlite:test.db
    driver-class-name: org.sqlite.JDBC
    username: sa
    password:
```

再实现一个HikariCP支持的 `DataSource`：

```
@Configuration
public class JdbcConfiguration {

    @Bean(destroyMethod = "close")
    DataSource dataSource(
        // properties:
        @Value("${summer.datasource.url}") String url,
        @Value("${summer.datasource.username}") String username,
        @Value("${summer.datasource.password}") String password,
        @Value("${summer.datasource.driver-class-name:}") String
driver,
        @Value("${summer.datasource.maximum-pool-size:20}") int
maximumPoolSize,
        @Value("${summer.datasource.minimum-pool-size:1}") int
minimumPoolSize,
        @Value("${summer.datasource.connection-timeout:30000}") int
connTimeout
    )
```

```
) {  
    var config = new HikariConfig();  
    config.setAutoCommit(false);  
    config.setJdbcUrl(url);  
    config.setUsername(username);  
    config.setPassword(password);  
    if (driver != null) {  
        config.setDriverClassName(driver);  
    }  
    config.setMaximumPoolSize(maximumPoolSize);  
    config.setMinimumIdle(minimumPoolSize);  
    config.setConnectionTimeout(connTimeout);  
    return new HikariDataSource(config);  
}  
}
```

这样，客户端引入 `JdbcConfiguration` 就自动获得了数据源：

```
@Import(JdbcConfiguration.class)  
@ComponentScan  
@Configuration  
public class AppConfig {  
}
```

定义JdbcTemplate

下一步是定义 `JdbcTemplate`，唯一依赖是注入 `DataSource`：

```
public class JdbcTemplate {  
    final DataSource dataSource;  
  
    public JdbcTemplate(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
}
```

`JdbcTemplate`基于`Template模式`，提供了大量以回调作为参数的模板方法，其中以`execute(ConnectionCallback)`为基础：

```
public <T> T execute(ConnectionCallback<T> action) {
    try (Connection newConn = dataSource.getConnection()) {
        T result = action.doInConnection(newConn);
        return result;
    } catch (SQLException e) {
        throw new DataAccessException(e);
    }
}
```

即由 `JdbcTemplate` 处理获取连接、释放连接、捕获SQLException，上层代码专注于使用 `Connection`：

```
@FunctionalInterface
public interface ConnectionCallback<T> {
    @Nullable
    T doInConnection(Connection con) throws SQLException;
}
```

其他方法其实也是基于 `execute(ConnectionCallback)`，例如：

```
public <T> T execute(PreparedStatementCreator psc,
    PreparedStatementCallback<T> action) {
    return execute((Connection con) -> {
        try (PreparedStatement ps = psc.createPreparedStatement(con)) {
            return action.doInPreparedStatement(ps);
        }
    });
}
```

上述代码实现了 `ConnectionCallback`，内部又调用了传入的 `PreparedStatementCreator` 和 `PreparedStatementCallback`，这样，基于更新操作的 `update` 就可以这么写：

```
public int update(String sql, Object... args) {
    return execute(
        preparedStatementCreator(sql, args),
        (PreparedStatement ps) -> {
            return ps.executeUpdate();
        }
    );
}
```

```
);  
}
```

基于查询操作的 `queryForList()` 就可以这么写：

```
public <T> List<T> queryForList(String sql, RowMapper<T> rowMapper,  
Object... args) {  
    return execute(preparedStatementCreator(sql, args),  
        (PreparedStatement ps) -> {  
            List<T> list = new ArrayList<>();  
            try (ResultSet rs = ps.executeQuery()) {  
                while (rs.next()) {  
                    list.add(rowMapper.mapRow(rs, rs.getRow()));  
                }  
            }  
            return list;  
        }  
    );  
}
```

剩下的一系列查询方法都是基于上述方法的封装，包括：

- `queryForList(String sql, RowMapper rowMapper, Object... args)`
- `queryForList(String sql, Class clazz, Object... args)`
- `queryForNumber(String sql, Object... args)`

总之，就是一个工作量的问题，开发难度基本为0。

测试时，可以使用Sqlite这个轻量级数据库，测试用例覆盖到各种SQL操作，最后把

`JdbcTemplate` 加入到 `JdbcConfiguration` 中，就基本完善了。

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

实现声明式事务

原文链接

Spring提供的声明式事务管理能极大地降低应用程序的事务代码。如果使用基于Annotation配置的声明式事务，则一个与数据库操作相关的类只需加上 `@Transactional` 注解，就实现了事务支持，非常方便：

```
@Transactional
@Component
public class UserService {
}
```

Spring的声明式事务支持JDBC本地事务和JTA分布式事务两种，事务传播模型除了最常用的 `REQUIRED`，还包括Java EE定义的 `SUPPORTS`、`REQUIRED_NEW`、`NESTED` 等多种模式。Summer Framework出于简化目的，仅支持JDBC本地事务，事务传播模型仅支持最常用的 `REQUIRED`，这样可以大大简化代码：

	Spring Framework	Summer Framework
JDBC事务	支持	支持
JTA事务	支持	不支持
REQUIRED传播模式	支持	支持
其他传播模式	支持	不支持
设置隔离级别	支持	不支持

下面我们就来编写声明式事务管理。

首先定义 `@Transactional`，这里就不允许单独在方法处定义，直接在class级别启动所有public方法的事务：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface Transactional {
```

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

默认值 `platformTransactionManager` 表示用名字为 `platformTransactionManager` 的Bean来管理事务。

下一步是定义接口 `PlatformTransactionManager` :

```
public interface PlatformTransactionManager {  
}
```

其实啥也没有，就是一个标识作用。

接着定义 `TransactionStatus` ，表示当前事务状态：

```
public class TransactionStatus {  
    final Connection connection;  
  
    public TransactionStatus(Connection connection) {  
        this.connection = connection;  
    }  
}
```

目前仅封装了一个Connection，将来如果扩展，则可以将事务的传播模式存储在里面。

最后写个 `DataSourceTransactionManager` ，它持有一个`ThreadLocal`存储的 `TransactionStatus` ，以及一个 `DataSource` :

```
public class DataSourceTransactionManager implements  
    PlatformTransactionManager, InvocationHandler  
{  
    static final ThreadLocal<TransactionStatus> transactionStatus = new  
ThreadLocal<>();  
    final DataSource dataSource;  
  
    public DataSourceTransactionManager(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
}
```

因为 `DataSourceTransactionManager` 是真正执行开启、提交、回归事务的地方，在哪执行呢？就在 `invoke()` 内部：

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    TransactionStatus ts = transactionStatus.get();
    if (ts == null) {
        // 当前无事务,开启新事务:
        try (Connection connection = dataSource.getConnection()) {
            final boolean autoCommit = connection.getAutoCommit();
            if (autoCommit) {
                connection.setAutoCommit(false);
            }
            try {
                // 设置ThreadLocal状态:
                transactionStatus.set(new TransactionStatus(connection));
                // 调用业务方法:
                Object r = method.invoke(proxy, args);
                // 提交事务:
                connection.commit();
                // 方法返回:
                return r;
            } catch (InvocationTargetException e) {
                // 回滚事务:
                TransactionException te = new
TransactionException(e.getCause());
                try {
                    connection.rollback();
                } catch (SQLException sqle) {
                    te.addSuppressed(sqle);
                }
                throw te;
            } finally {
                // 删除ThreadLocal状态:
                transactionStatus.remove();
                if (autoCommit) {
                    connection.setAutoCommit(true);
                }
            }
        }
    }
}
```



```
    }  
    } else {  
        // 当前已有事务,加入当前事务执行:  
        return method.invoke(proxy, args);  
    }  
}
```

这样就实现了声明式事务。

有的同学会问, 如果一个方法开启了事务, 那么, 它内部调用其他方法, 是怎么加入当前事务的?

这里我们先需要写一个获取当前事务连接的工具类:

```
public class TransactionalUtils {  
    @Nullable  
    public static Connection getCurrentConnection() {  
        TransactionStatus ts =  
DataSourceTransactionManager.transactionStatus.get();  
        return ts == null ? null : ts.connection;  
    }  
}
```

然后改造下 `JdbcTemplate` 获取连接的代码:

```
public class JdbcTemplate {  
    public <T> T execute(ConnectionCallback<T> action) throws  
DataAccessException {  
        // 尝试获取当前事务连接:  
        Connection current = TransactionalUtils.getCurrentConnection();  
        if (current != null) {  
            try {  
                return action.doInConnection(current);  
            } catch (SQLException e) {  
                throw new DataAccessException(e);  
            }  
        }  
        // 无事务,从DataSource获取新连接:  
        try (Connection newConn = dataSource.getConnection()) {  
            return action.doInConnection(newConn);  
        } catch (SQLException e) {
```

```
        throw new DataAccessException(e);
    }
}
...
}
```

这样，使用 `JdbcTemplate`，如果有事务，自动加入当前事务，否则，按普通SQL执行（数据库隐含事务）。

最后，还需要提供一个 `TransactionalBeanPostProcessor`，使得AOP机制生效，才能拦截

`@Transactional` 标注的Bean的public方法：

```
public class TransactionalBeanPostProcessor extends
AnnotationProxyBeanPostProcessor<Transactional> {
}
```

把它们都整理一下，放到 `JdbcConfiguration` 中：

```
@Configuration
public class JdbcConfiguration {

    @Bean(destroyMethod = "close")
    DataSource dataSource(
        // properties:
        @Value("${summer.datasource.url}") String url,
        @Value("${summer.datasource.username}") String username,
        @Value("${summer.datasource.password}") String password,
        @Value("${summer.datasource.driver-class-name}") String
driver,
        @Value("${summer.datasource.maximum-pool-size:20}") int
maximumPoolSize,
        @Value("${summer.datasource.minimum-pool-size:1}") int
minimumPoolSize,
        @Value("${summer.datasource.connection-timeout:30000}") int
connTimeout
    ) {
        ...
        return new HikariDataSource(config);
    }
}
```

```
@Bean
JdbcTemplate jdbcTemplate(@Autowired DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}

@Bean
TransactionalBeanPostProcessor transactionalBeanPostProcessor() {
    return new TransactionalBeanPostProcessor();
}

@Bean
PlatformTransactionManager platformTransactionManager(@Autowired
DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}
}
```

现在，应用程序只需导入 `JdbcConfiguration`，从连接池到声明式事务全部齐活：

```
@Import(JdbcConfiguration.class)
@ComponentScan
@Configuration
public class AppConfig {
}
```

最后我们总结下各个组件的作用：

1. 由 `JdbcConfiguration` 创建的 `DataSource`，实现了连接池；
2. 由 `JdbcConfiguration` 创建的 `JdbcTemplate`，实现基本SQL操作；
3. 由 `JdbcConfiguration` 创建的 `PlatformTransactionManager`，负责拦截 `@Transactional` 标识的Bean的public方法，自动管理事务；
4. 由 `JdbcConfiguration` 创建的 `TransactionalBeanPostProcessor`，负责给 `@Transactional` 标识的Bean创建AOP代理，拦截器正是 `PlatformTransactionManager`。

应用程序除了导入一个 `JdbcConfiguration`，加上默认配置项，什么也不用干，就可以开始写自动带声明式事务的代码：

```
@Transactional
@Component
public class UserService {
```

```
@Autowired
JdbcTemplate jdbcTemplate;

public User register(String email, String password) {
    jdbcTemplate.update("INSERT INTO ...", ...);
    return ...
}
}
```

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

实现Web MVC

原文链接

现在，我们已经实现了IoC容器、AOP、JdbcTemplate和声明式事务，离一个完整的框架只差一个Web MVC了。

我们先看看Spring的Web MVC主要提供了哪些组件和API支持：

- 1. 一个 `DispatcherServlet` 作为核心处理组件，接收所有URL请求，然后按MVC规则转发；
- 2. 基于 `@Controller` 注解的URL控制器，由应用程序提供，Spring负责解析规则；
- 3. 提供 `ViewResolver` ，将应用程序的Controller处理后的结果进行渲染，给浏览器返回页面；
- 4. 基于 `@RestController` 注解的REST处理机制，由应用程序提供，Spring负责将输入输出变为JSON格式；
- 5. 多种拦截器和异常处理器等。

Spring的Web MVC功能十分强大，涉及到的内容也非常广。相比之下，Summer Framework的Web MVC必然要聚焦在核心组件上：

	Spring Framework	Summer Framework
DispatcherServlet	支持	支持
@Controller注解	支持	支持
@RestController注解	支持	支持
ViewResolver	支持	支持
HandlerInterceptor	支持	不支持
Exception Handler	支持	不支持
CORS	支持	不支持
异步处理	支持	不支持
WebSocket	支持	不支持

不过，Spring Framework的Web MVC模块对 `Filter` 支持有限，要想愉快地使用 `Filter` ，最好通过Spring Boot提供的 `FilterRegistrationBean` ，Summer Framework为了便于应用程

序开发自己的 `Filter` ，直接支持 `FilterRegistrationBean` 。

下面开始正式开发Summer Framework的Web MVC模块。

[评论](#)

启动IoC容器

原文链接

在开发Web MVC模块之前，我们首先回顾下Java Web应用程序到底有几方参与。

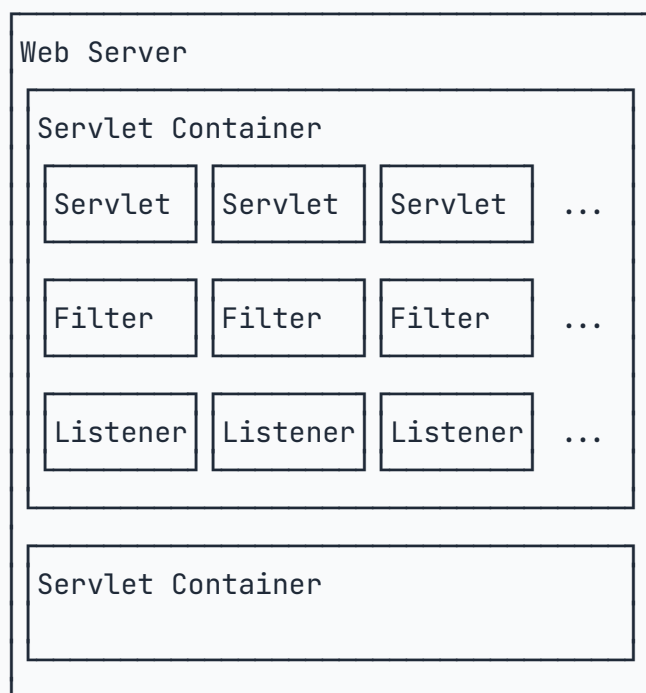
首先，Java Web应用一般遵循Servlet标准，这个标准定义了应用程序可以按接口编写哪些组件：Servlet、Filter和Listener，也规定了一个服务器（如Tomcat、Jetty、JBoss等）应该提供什么样的服务，按什么顺序加载应用程序的组件，最后才能跑起来处理来自用户的HTTP请求。

Servlet规范定义的组件有3类：

1. Servlet：处理HTTP请求，然后输出响应；
2. Filter：对HTTP请求进行过滤，可以有多个Filter形成过滤器链，实现权限检查、限流、缓存等逻辑；
3. Listener：用来监听Web应用程序产生的事件，包括启动、停止、Session有修改等。

这些组件均由应用程序实现。

而服务器为一个应用程序提供一个“容器”，即Servlet Container，一个Server可以同时跑多个Container，不同的Container可以按URL、域名等区分，Container才是用来管理Servlet、Filter、Listener这些组件的：



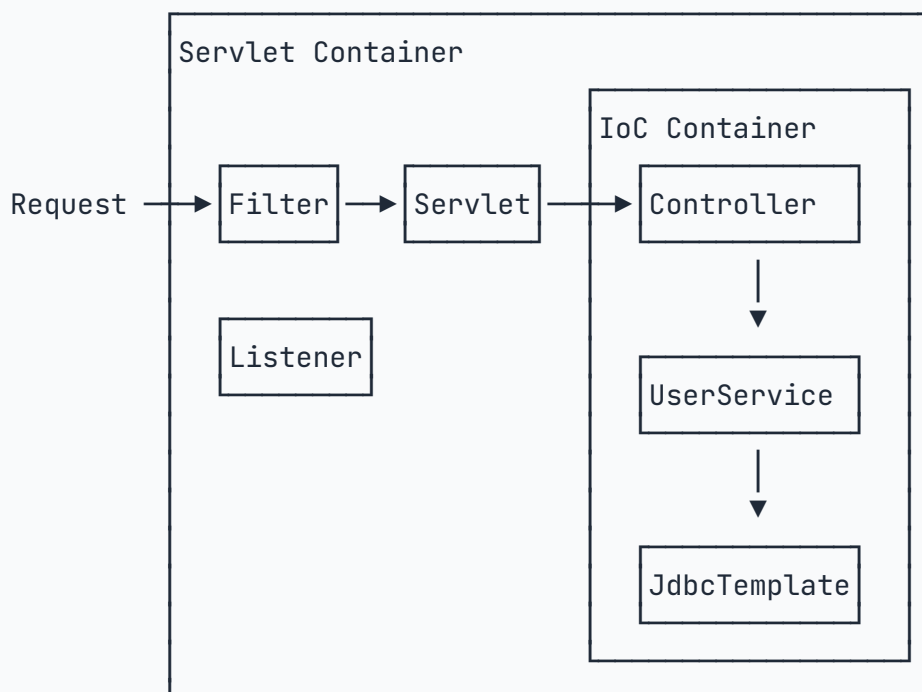
另一个需要特别重要的问题是：组件由谁创建，由谁销毁。

在使用IoC容器时，注意到IoC容器也是一个Java类，IoC容器又管理着很多Bean，因此，创建顺序是：

1. 执行应用程序的入口方法 `main()` ；
2. 在 `main()` 方法中，创建IoC容器的实例；
3. IoC容器在它的内部创建各个Bean的实例。

现在，我们开发的是Web应用程序，它本身就是一堆组件，被Web服务器提供的Servlet“容器”管理，同时，又要加一个IoC容器，到底谁创建谁，谁管理谁，这个问题，必须要搞清楚。

首先，我们不能改变Servlet规范，所以，Servlet、Filter、Listener，以及IoC容器，都必须在Servlet容器内被管理：



所以我们要捋清楚这些组件的创建顺序，以及谁创建谁。

对于一个Web应用程序来说，启动时，应用程序本身只是一个 `war` 包，并没有 `main()` 方法，因此，启动时执行的是Server的 `main()` 方法。以Tomcat服务器为例：

1. 启动服务器，即执行Tomcat的 `main()` 方法；
2. Tomcat根据配置或自动检测到一个 `xyz.war` 包后，为这个 `xyz.war` 应用程序创建Servlet容器；
3. Tomcat继续查找 `xyz.war` 定义的Servlet、Filter和Listener组件，按顺序实例化每个组件（Listener最先被实例化，然后是Filter，最后是Servlet）；

4. 用户发送HTTP请求，Tomcat收到请求后，转发给Servlet容器，容器根据应用程序定义的映射，把请求发送个若干Filter和一个Servlet处理；

5. 处理期间产生的事件则由Servlet容器自动调用Listener。

其中，第3步实例化又有很多方式：

1. 通过在 `web.xml` 配置文件中定义，这也是早期Servlet规范唯一的配置方式；
2. 通过注解 `@WebServlet`、`@WebFilter` 和 `@WebListener` 定义，由Servlet容器自动扫描所有class后创建组件，这和我们用Annotation配置Bean，由IoC容器自动扫描创建Bean非常类似；
3. 先配置一个 `Listener`，由Servlet容器创建 `Listener`，然后，`Listener` 自己调用相关接口，手动创建 `Servlet` 和 `Filter`。

到底用哪种方式，取决于Web应用程序自己如何编写。对于使用Spring框架的Web应用程序来说，Servlet、Filter和Listener数量少，而且是固定的，应用程序自身编写的Controller数量不定，但由IoC容器管理，因此，采用方式3最合适。

具体来说，Tomcat启动一个基于Spring开发的Web应用程序时，按如下步骤初始化：

1. 为Web应用程序准备Servlet容器；
2. 根据配置实例化一个Spring提供的 `Listener`；
 1. Spring提供的 `Listener` 在初始化时启动IoC容器；
 2. Spring提供的 `Listener` 在初始化时向Servlet容器注册Spring内置的一个 `DispatcherServlet`。

当Tomcat把HTTP请求发送给Spring注册的 `Servlet` 后，因为它持有IoC容器的引用，就可以找到 `Controller` 实例，因此，可以把请求继续转发给对应的Controller，这样就完成了HTTP请求的处理。

另外注意到Web应用程序除了提供 `Controller` 外，并不必须与Servlet API打交道，因为被Spring提供的 `DispatcherServlet` 给隔离了。

所以，我们在开发Summer Framework的Web MVC模块时，应该以如下方式初始化：

1. 应用程序必须配置一个Summer Framework提供的Listener；
2. Tomcat完成Servlet容器的创建后，立刻根据配置创建Listener；
 1. Listener初始化时创建IoC容器；
 2. Listener继续创建DispatcherServlet实例，并向Servlet容器注册；

3. DispatcherServlet初始化时获取到IoC容器中的Controller实例，因此可以根据URL调用不同Controller实例的不同处理方法。

我们先写一个只能输出Hello World的Servlet:

```
public class DispatcherServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        PrintWriter pw = resp.getWriter();
        pw.write("<h1>Hello, world!</h1>");
        pw.flush();
    }
}
```

紧接着，编写一个 `ContextLoaderListener`，它实现了 `ServletContextListener` 接口，能监听Servlet容器的启动和销毁，在监听到初始化事件时，完成创建IoC容器和注册

`DispatcherServlet` 两个工作：

```
public class ContextLoaderListener implements ServletContextListener {
    // Servlet容器启动时自动调用：
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        // 创建IoC容器：
        var applicationContext = createApplicationContext(...);
        // 实例化DispatcherServlet：
        var dispatcherServlet = new DispatcherServlet();
        // 注册DispatcherServlet：
        var dispatcherReg = servletContext.addServlet("dispatcherServlet",
            dispatcherServlet);
        dispatcherReg.addMapping("/");
        dispatcherReg.setLoadOnStartup(0);
    }
}
```

这样，我们就完成了Web应用程序的初始化全部流程！

最后两个小问题：

1. 创建IoC容器时，需要的配置文件从哪读？这里我们采用Spring Boot的方式，默认从classpath的 `application.yml` 或 `application.properties` 读。

2. 需要的 `@Configuration` 配置类从哪获取？这是通过 `web.xml` 文件配置的：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  <context-param>
    <!-- 固定名称 -->
    <param-name>configuration</param-name>
    <!-- 配置类的完整类名 -->
    <param-value>com.itranswarp.summer.webapp.WebAppConfig</param-
value>
  </context-param>

  <listener>
    <listener-
class>com.itranswarp.summer.web.ContextLoaderListener</listener-class>
  </listener>
</web-app>
```

在 `ContextLoaderListener` 的 `contextInitialized()` 方法内，先获取 `ServletContext` 引用，再通过 `getInitParameter("configuration")` 拿到完整类名，就可以顺利创建IoC容器了。

用Maven打包后，把生成的 `xyz.war` 改为 `ROOT.war`，复制到Tomcat的 `webapps` 目录下，清除掉其他webapp，启动Tomcat，输入 `http://localhost:8080` 可看到输出 `Hello, world!`。

这样我们就跑通了一个Web应用程序启动的全部流程。

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

实现MVC

原文链接

上一节我们把Web应用程序的流程跑通了，因此，本节重点就在如何继续开发

`DispatcherServlet`，因为整个MVC的处理都是在 `DispatcherServlet` 内部完成的。

要处理MVC，我们先定义 `@Controller` 和 `@RestController`：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {
    String value() default "";
}

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface RestController {
    String value() default "";
}
```

以及 `@GetMapping`、`@PostMapping` 等注解，来标识MVC处理的方法。

`DispatcherServlet` 内部负责从IoC容器找出所有 `@Controller` 和 `@RestController` 定义的Bean，扫描它们的方法，找出 `@GetMapping` 和 `@PostMapping` 标识的方法，这样就有了一个处理特定URL的处理器，我们抽象为 `Dispatcher`：

```
class Dispatcher {
    // 是否返回REST:
    boolean isRest;
    // 是否有@ResponseBody:
    boolean isResponseBody;
    // 是否返回void:
    boolean isVoid;
    // URL正则匹配:
    Pattern urlPattern;
```

```
// Bean实例：
Object controller;
// 处理方法：
Method handlerMethod;
// 方法参数：
Param[] methodParameters;
}
```

方法参数也需要根据 `@RequestParam`、`@RequestBody` 等抽象出 `Param` 类型：

```
class Param {
    // 参数名称：
    String name;
    // 参数类型：
    ParamType paramType;
    // 参数Class类型：
    Class<?> classType;
    // 参数默认值
    String defaultValue;
}
```

一共有4种类型的参数，我们用枚举 `ParamType` 定义：

- `PATH_VARIABLE`：路径参数，从URL中提取；
- `REQUEST_PARAM`：URL参数，从URL Query或Form表单提取；
- `REQUEST_BODY`：REST请求参数，从Post传递的JSON提取；
- `SERVLET_VARIABLE`：`HttpServletRequest` 等Servlet API提供的参数，直接从 `DispatcherServlet` 的方法参数获得。

这样，`DispatcherServlet` 通过反射拿到一组 `Dispatcher` 对象，在 `doGet()` 和 `doPost()` 方法中，依次匹配URL：

```
public class DispatcherServlet extends HttpServlet {

    List<Dispatcher> getDispatchers = new ArrayList<>();
    List<Dispatcher> postDispatchers = new ArrayList<>();

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
```

```
String url = req.getRequestURI();
// 依次匹配每个Dispatcher的URL:
for (Dispatcher dispatcher : getDispatchers) {
    Result result = dispatcher.process(url, req, resp);
    // 匹配成功并处理后:
    if (result.processed()) {
        // 处理结果
        ...
        return;
    }
}
// 未匹配到任何Dispatcher:
resp.sendError(404, "Not Found");
}

@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    ...
}
}
```

这里不能用 `Map<String, Dispatcher>` 的原因在于我们要处理类似 `/hello/{name}` 这样的URL，没法使用精确查找，只能使用正则匹配。

`Dispatcher` 处理后返回类型包括：

- `void` 或 `null`：表示内部已处理完毕；
- `String`：如果以 `redirect:` 开头，则表示一个重定向；
- `String` 或 `byte[]`：如果配合 `@ResponseBody`，则表示返回值直接写入响应；
- `ModelAndView`：表示这是一个MVC响应，包含Model和View名称，后续用模板引擎处理后写入响应；
- 其它类型：如果是 `@RestController`，则序列化为JSON后写入响应。

不符合上述要求的返回类型则报500错误。

这些处理逻辑都十分简单，我们重点看看如何处理 `ModelAndView` 类型，即MVC响应。

为了处理 `ModelAndView`，我们需要一个模板引擎，因此，抽象出 `ViewResolver` 接口：

```
public interface ViewResolver {  
    // 初始化ViewResolver:  
    void init();  
  
    // 渲染:  
    void render(String viewName, Map<String, Object> model,  
        HttpServletRequest req, HttpServletResponse resp);  
}
```

Spring内置FreeMarker引擎，因此我们也把FreeMarker集成进来，写一个

FreeMarkerViewResolver :

```
public class FreeMarkerViewResolver implements ViewResolver {  
  
    final String templatePath;  
    final String templateEncoding;  
    final ServletContext servletContext;  
  
    Configuration config;  
  
    public FreeMarkerViewResolver(ServletContext servletContext, String  
        templatePath, String templateEncoding) {  
        this.servletContext = servletContext;  
        this.templatePath = templatePath;  
        this.templateEncoding = templateEncoding;  
    }  
  
    @Override  
    public void init() {  
        Configuration cfg = new  
Configuration(Configuration.VERSION_2_3_32);  
        cfg.setOutputFormat(HTMLOutputFormat.INSTANCE);  
        cfg.setDefaultEncoding(this.templateEncoding);  
        cfg.setTemplateLoader(new  
ServletTemplateLoader(this.servletContext, this.templatePath));  
  
        cfg.setTemplateExceptionHandler(TemplateExceptionHandler.HTML_DEBUG_HANDLER  
);  
  
        cfg.setAutoEscapingPolicy(Configuration.ENABLE_IF_SUPPORTED_AUTO_ESCAPING_P
```

```
OLICY);
    cfg.setLocalizedLookup(false);
    var ow = new DefaultObjectWrapper(Configuration.VERSION_2_3_32);
    ow.setExposeFields(true);
    cfg.setObjectWrapper(ow);
    this.config = cfg;
}

@Override
public void render(String viewName, Map<String, Object> model,
    HttpServletRequest req, HttpServletResponse resp) throws ServletException,
    IOException {
    Template templ = null;
    try {
        templ = this.config.getTemplate(viewName);
    } catch (Exception e) {
        throw new ServerErrorException("View not found: " + viewName);
    }
    PrintWriter pw = resp.getWriter();
    try {
        templ.process(model, pw);
    } catch (TemplateException e) {
        throw new ServerErrorException(e);
    }
    pw.flush();
}
}
```

这样我们就可以在 `DispatcherServlet` 内部，把处理 `ModelAndView` 和 `ViewResolver` 结合起来，最终向 `HttpServletResponse` 中输出HTML，完成HTTP请求的处理。

为了简化Web应用程序配置，我们提供一个 `WebMvcConfiguration` 配置：

```
@Configuration
public class WebMvcConfiguration {
    private static ServletContext servletContext = null;
    static void setServletContext(ServletContext ctx) {
        servletContext = ctx;
    }

    @Bean(initMethod = "init")
```



```
ViewResolver viewResolver( //
    @Autowired ServletContext servletContext, //
    @Value("${summer.web.freemarker.template-path:/WEB-INF/templates}") String templatePath, //
    @Value("${summer.web.freemarker.template-encoding=UTF-8}")
String templateEncoding) {
    return new FreeMarkerViewResolver(servletContext, templatePath,
templateEncoding);
}

@Bean
ServletContext servletContext() {
    return Objects.requireNonNull(servletContext, "ServletContext is
not set.");
}
}
```

默认创建一个 `ViewResolver` 和 `ServletContext`，注意 `ServletContext` 本身实际上是由Servlet容器提供的，但我们把它放入IoC容器，是因为许多涉及到Web的组件，如 `ViewResolver`，需要注入 `ServletContext`，才能从指定配置加载文件。

最后，整理代码，添加一些能方便用户开发的额外功能，例如处理静态文件等功能，我们的Web MVC模块就开发完毕！

注意事项

在整个HTTP处理流程中，入口是 `DispatcherServlet` 的 `service()` 方法，整个流程如下：

1. Servlet容器调用 `DispatcherServlet` 的 `service()` 方法处理HTTP请求；
2. `service()` 根据GET或POST调用 `doGet()` 或 `doPost()` 方法；
3. 根据URL依次匹配 `Dispatcher`，匹配后调用 `process()` 方法，获得返回值；
4. 根据返回值写入响应：
 1. void或null返回值无需写入响应；
 2. String或byte[]返回值直接写入响应（或重定向）；
 3. REST类型写入JSON序列化结果；
 4. ModelAndView类型调用ViewResolver写入渲染结果。
5. 未匹配到判断是否静态资源：

1. 符合静态目录（默认 `/static/`）则读取文件，写入文件内容；
 2. 网站图标（默认 `/favicon.ico`）则读取 `.ico` 文件，写入文件内容；
6. 其他情况返回404。

由于在处理的每一步都可以向 `HttpServletResponse` 写入响应，因此，后续步骤写入时，应判断前面的步骤是否已经写入并发送了HTTP Header。`isCommitted()` 方法就是干这个用的：

```
if (!resp.isCommitted()) {  
    resp.resetBuffer();  
    writeTo(resp);  
}
```

测试

`DispatcherServlet` 处理HTTP请求时，一些组件是Servlet容器提供的，如：

- `HttpServletRequest`;
- `HttpServletResponse`;
- `HttpSession`;
- `ServletContext`。

要模拟这些对象用Mockito之类的框架代码量也很大，我们可以借用Spring提供的test模块，它实现了完善的MockHttpServletRequest、MockServletContext等对象，便于测试。我们导入：

- `org.springframework:spring-test:6.0.0`
- `org.springframework:spring-web:6.0.0`

注意设置 `<scope>test</scope>`，即仅在测试代码中用到了Spring提供的Mock对象，业务代码并不会用到Spring的任何功能。一个简单的测试用例如下：

```
@Test  
void getGreeting() throws ServletException, IOException {  
    // 创建MockHttpServletRequest:  
    var req = createMockRequest("GET", "/greeting", null, Map.of("name",  
        "Bob"));  
    // 创建MockHttpServletResponse:  
    var resp = createMockResponse();
```

```
// 处理请求:  
this.dispatcherServlet.service(req, resp);  
// 验证200响应:  
assertEquals(200, resp.getStatus());  
// 验证响应内容:  
assertEquals("Hello, Bob", resp.getContentAsString());  
}
```

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

开发Web应用

原文链接

在我们开发完Summer Framework的所有组件后，就可以基于Summer Framework来开发一个真正的Web应用了！

我们来一步一步创建一个 `hello-webapp` 的应用，它基于Maven项目，符合webapp标准。

首先，我们在 `src/main/resources` 下定义配置文件 `application.yml`：

```
app:
  title: Hello Application
  version: 1.0

summer:
  datasource:
    url: jdbc:sqlite:test.db
    driver-class-name: org.sqlite.JDBC
    username: sa
    password:
```

紧接着，定义IoC容器的配置类如下：

```
@ComponentScan
@Configuration
@Import({ JdbcConfiguration.class, WebMvcConfiguration.class })
public class HelloConfiguration {
}
```

以及相关的 `UserService`、`MvcController` 等Bean。

接下来是在 `src/main/webapp/WEB-INF` 目录下创建Servlet容器所需的配置文件 `web.xml`：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  <display-name>Hello Webapp</display-name>

  <context-param>
    <param-name>configuration</param-name>
```

```
<param-value>com.itranswarp.hello.HelloConfiguration</param-value>
</context-param>

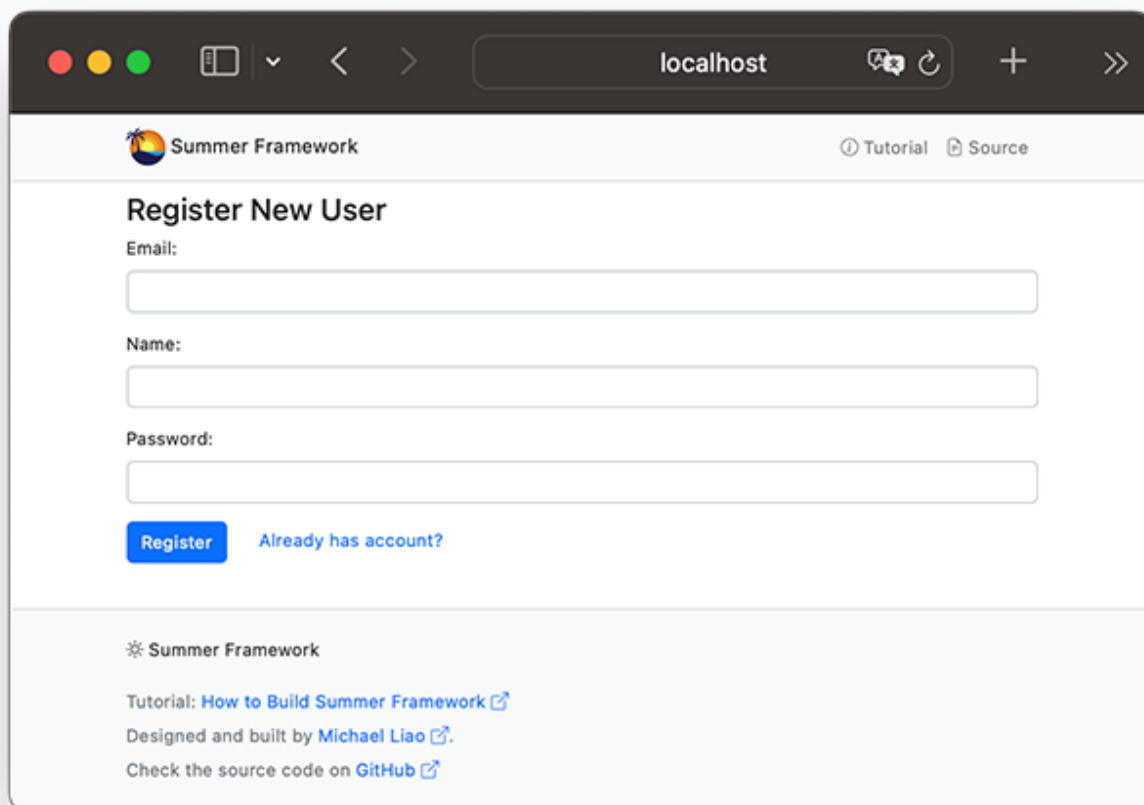
<listener>
  <listener-
class>com.itranswarp.summer.web.ContextLoaderListener</listener-class>
  </listener>
</web-app>
```

Servlet容器会自动读取 `web.xml`，根据配置的Listener启动Summer Framework的web模块的 `ContextLoaderListener`，它又会读取 `web.xml` 配置的 `<context-param>` 获得配置类的全名 `com.itranswarp.hello.HelloConfiguration`，最后用这个配置类完成IoC容器的创建。创建后自动注册Summer Framework的 `DispatcherServlet`，以及Web应用程序定义的 `FilterRegistrationBean`，这样就完成了整个Web应用程序的初始化。

其他用到的资源包括：

- 存储在 `src/main/webapp/static` 目录下的静态资源；
- 存储于 `src/main/webapp/favicon.ico` 的图标文件；
- 存储在 `src/main/webapp/WEB-INF/templates` 目录下的模板。

最后，运行 `mvn clean package` 命令，在 `target` 目录得到最终的war包，改名为 `ROOT.war`，复制到Tomcat的 `webapps` 目录下，启动Tomcat，可以正常访问 `http://localhost:8080`：



参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

实现Boot

原文链接

虽然基于Summer Framework可以开发一个完整的Web应用程序，但是，开发过程还是涉及到先打包，再复制到Tomcat的webapps目录，再启动Tomcat。在开发过程中，经常需要反复来好多次，每次停止、复制、启动，要调试还要接入远程，搞着搞着就会发现，这种开发模式太麻烦。

直接用Spring Framework开发Web应用也是一样，所以才有了Spring Boot，它最让人省心的一点，就是不用装Tomcat，不用复制war包，打个jar包直接就能跑！

所以，为了简化开发流程，我们也仿照Spring Boot，编写一个 `boot` 模块，能直接启动运行！

注意Spring Boot除了直接打包运行外，还提供很多其他功能，而Summer Framework的boot模块只提供打包运行功能，无其他额外功能。

下面开始正式开发Summer Framework的boot模块。

评论

启动嵌入式Tomcat

原文链接

Spring Boot实现一个jar包直接运行的原理其实就是把Tomcat打包进去，自己再写个 `main()` 函数：

```
@SpringBootApplication
public class AppConfig {
    public static void main(String[] args) {
        SpringApplication.run(AppConfig.class, args);
    }
}
```

在 `SpringApplication.run()` 方法内，Spring Boot会启动嵌入式Tomcat，然后再初始化Spring的IoC容器，实际上就是一个jar包内包含了嵌入式Tomcat、Spring IoC容器、Web MVC模块以及应用程序自己开发的Bean。

因此，我们也提供一个 `SummerApplication`，实现 `run()` 方法如下：

```
public class SummerApplication {
    public static void run(String webDir, String baseDir, Class<?>
configClass, String... args) {
        // 读取application.yml配置：
        var propertyResolver = WebUtils.createPropertyResolver();
        // 创建Tomcat服务器：
        var server = startTomcat(webDir, baseDir, configClass,
propertyResolver);
        // 等待服务器结束：
        server.await();
    }
}
```

这里多了两个参数：`webDir` 和 `baseDir`，这是为启动嵌入式Tomcat准备的，启动嵌入式Tomcat的代码如下：

```
Server startTomcat(String webDir, String baseDir, Class<?> configClass,
PropertyResolver propertyResolver) throws Exception {
    int port = propertyResolver.getProperty("${server.port:8080}",
```



```
int.class);
// 实例化Tomcat Server:
Tomcat tomcat = new Tomcat();
tomcat.setPort(port);
// 设置Connector:
tomcat.getConnector().setThrowOnFailure(true);
// 添加一个默认的Webapp, 挂载在'/':
Context ctx = tomcat.addWebapp("", new File(webDir).getAbsolutePath());
// 设置应用程序的目录:
WebResourceRoot resources = new StandardRoot(ctx);
resources.addPreResources(new DirResourceSet(resources, "/WEB-INF/classes", new File(baseDir).getAbsolutePath(), "/"));
ctx.setResources(resources);
// 设置ServletContainerInitializer监听器:
ctx.addServletContainerInitializer(new ContextLoaderInitializer(configClass, propertyResolver), Set.of());
// 启动服务器:
tomcat.start();
return tomcat.getServer();
}
```

那么我们的IoC容器, 以及注册 `Servlet`、`Filter` 是在哪进行的? 答案是在 `startTomcat()` 内注册了一个 `ServletContainerInitializer` 监听器, 这个监听器负责启动IoC容器与注册 `Servlet`、`Filter` :

```
public class ContextLoaderInitializer implements
ServletContainerInitializer {
    final Class<?> configClass;
    final PropertyResolver propertyResolver;

    public ContextLoaderInitializer(Class<?> configClass, PropertyResolver
propertyResolver) {
        this.configClass = configClass;
        this.propertyResolver = propertyResolver;
    }

    @Override
    public void onStartUp(Set<Class<?>> c, ServletContext ctx) throws
ServletException {
        // 设置ServletContext:
```

```
        WebMvcConfiguration.setServletContext(ctx);
        // 启动IoC容器:
        ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(this.configClass,
this.propertyResolver);
        // 注册Filter与DispatcherServlet:
        WebUtils.registerFilters(ctx);
        WebUtils.registerDispatcherServlet(ctx, this.propertyResolver);
    }
}
```

没有复用web模块的 `ContextLoaderListener` 是因为Tomcat不允许没有在 `web.xml` 中声明的 `Listener` 注册 `Filter` 与 `Servlet`，而我们写boot模块原因之一也是要做到不需要 `web.xml`。

这样我们就完成了boot模块的开发，它其实就包含两个组件：

- SummerApplication：负责启动嵌入式Tomcat；
- ContextLoaderInitializer：负责启动IoC容器，注册 `Filter` 与 `DispatcherServlet`。

下面就可以编写一个基于 `boot` 的Web应用程序了。

参考源码

可以从[GitHub](#)或[Gitee](#)下载源码。

[GitHub](#)

[评论](#)

开发Boot应用

原文链接

我们已经准备好boot模块了，下一步是使用boot来开发Web应用程序。

我们还是先定义一个符合Maven结构的Web应用程序 `hello-boot`，先定义配置类

`HelloConfiguration`：

```
@ComponentScan
@Configuration
@Import({ JdbcConfiguration.class, WebMvcConfiguration.class })
public class HelloConfiguration {
}
```

以及 `UserService`、`MvcController` 等业务Bean。

我们直接写个 `main()` 方法启动：

```
public class Main {
    public static void main(String[] args) throws Exception {
        SummerApplication.run("src/main/webapp", "target/classes",
            HelloConfiguration.class, args);
    }
}
```

直接从IDE运行，是没有问题的，能顺利启动Tomcat、创建IoC容器、注册 `Filter` 和 `DispatcherServlet`，可以直接通过浏览器访问。

但是，如果打一个war包，直接运行 `java -jar xyz.war` 是不行的！会直接报错：找不到Main这个class！

这是为什么呢？我们要从JVM的类加载机制说起。

当我们用 `java` 启动一个Java程序时，需要用 `-cp` 参数设置classpath（默认为当前目录 `.`）；当我们用 `java -jar xyz.jar` 启动一个Java程序时，JVM忽略 `-cp` 参数，默认classpath为 `xyz.jar`，这样，如果能在jar包中找到对应的class，就可以正常运行。

要注意的一点是，JVM从jar包加载class，是从jar包的根目录查找的。如果它要加载

`com.itranswarp.hello.Main`，那么，`xyz.jar` 必须按如下目录组织：

```
xyz.jar
└── com
    └── itranswarp
        └── hello
            └── Main.class
```

而我们在用Maven打war包时，结构是这样的：

```
xyz.war
└── WEB-INF
    └── classes
        └── com
            └── itranswarp
                └── hello
                    └── Main.class
```

自然无法加载 `Main`。（注意jar包和war包仅扩展名不同，对JVM来说是完全一样的）

那为什么我们把 `xyz.war` 扔到Tomcat的webapps目录下就能正常运行呢？因为Tomcat启动后，并不使用JVM的ClassLoader加载class，而是为每个webapp创建一个单独的ClassLoader，这个ClassLoader在如下位置搜索class：

- WEB-INF/classes目录；
- WEB-INF/lib目录下的所有jar包。

因此，我们要运行的 `xyz.war` 包必须同时具有Web App的结构，又能在根目录下搜索到应用程序自己编写的 `Main`：

```
xyz.jar
├── com
│   ├── itranswarp
│   │   ├── hello
│   │   └── Main.class
└── WEB-INF
    ├── classes
    └── libs
```

解决方案是在打包时复制所有编译的class到war包根目录，并添加启动类入口。修改

`pom.xml`：

```
<project ...>
    ...

    <build>
        <finalName>${project.name}</finalName>
        <plugins>
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>3.3.2</version>
  <configuration>
    <!-- 复制classes到war包根目录 -->
    <webResources>
      <resource>

<directory>${project.build.directory}/classes</directory>
      </resource>
    </webResources>
    <archiveClasses>true</archiveClasses>
    <archive>
      <manifest>
        <!-- main启动类 -->

<mainClass>com.itranswarp.hello.Main</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
</plugins>
</build>
</project>
```

再次打包，运行，又会得到找不到Class的错误，不过这次是 `SummerApplication` 。

这又是什么原因呢？很明显 `Main` 已经找到了，但是 `SummerApplication` 在哪呢？它其实在 `WEB-INF/lib/summer-boot-1.x.x.jar`，JVM不会在 `WEB-INF/lib` 下搜索Class，也不会在一个jar包内搜索“jar包内的jar包”。

怎么破？

答案是 `Main` 运行时先自解压，再让JVM能搜索到 `WEB-INF/lib/summer-boot-1.x.x.jar` 即可。

需要先修改 `main()` 方法代码：

```
public static void main(String[] args) throws Exception {
    // 判定是否从jar/war启动：
```

```
String jarFile =
Main.class.getProtectionDomain().getCodeSource().getLocation().getFile();
boolean isJarFile = jarFile.endsWith(".war") ||
jarFile.endsWith(".jar");
// 定位webapp根目录:
String webDir = isJarFile ? "tmp-webapp" : "src/main/webapp";
if (isJarFile) {
    // 解压到tmp-webapp:
    Path baseDir = Paths.get(webDir).normalize().toAbsolutePath();
    if (Files.isDirectory(baseDir)) {
        Files.delete(baseDir);
    }
    Files.createDirectories(baseDir);
    System.out.println("extract to: " + baseDir);
    try (JarFile jar = new JarFile(jarFile)) {
        List<JarEntry> entries =
jar.stream().sorted(Comparator.comparing(JarEntry::getName)).collect(Collectors.toList());
        for (JarEntry entry : entries) {
            Path res = baseDir.resolve(entry.getName());
            if (!entry.isDirectory()) {
                System.out.println(res);
                Files.createDirectories(res.getParent());
                Files.copy(jar.getInputStream(entry), res);
            }
        }
    }
    // JVM退出时自动删除tmp-webapp:
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        try {
            Files.walk(baseDir).sorted(Comparator.reverseOrder()).map(Path::toFile).forEach(File::delete);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }));
}
SummerApplication.run(webDir, isJarFile ? "tmp-webapp" :
```

```
"target/classes", HelloConfiguration.class, args);  
}
```

再修改 `pom.xml` , 加上Classpath:

```
<project ...>  
    ...  
  
    <build>  
        <finalName>${project.name}</finalName>  
        <plugins>  
            <plugin>  
                <groupId>org.apache.maven.plugins</groupId>  
                <artifactId>maven-war-plugin</artifactId>  
                <version>3.3.2</version>  
                <configuration>  
                    <!-- 复制classes到war包根目录 -->  
                    <webResources>  
                        <resource>  
  
<directory>${project.build.directory}/classes</directory>  
                        </resource>  
                    </webResources>  
                    <archiveClasses>true</archiveClasses>  
                    <archive>  
                        <manifest>  
                            <!-- 添加Class-Path -->  
                            <addClasspath>true</addClasspath>  
                            <!-- Classpath前缀 -->  
                            <classpathPrefix>tmp-webapp/WEB-  
INF/lib/</classpathPrefix>  
                            <!-- main启动类 -->  
  
<mainClass>com.itranswarp.hello.Main</mainClass>  
                        </manifest>  
                    </archive>  
                </configuration>  
            </plugin>  
        </plugins>
```

```
</build>
</project>
```

当我们打包后，我们来分析启动流程。我们先把war包解压到 `tmp-webapp`，它的结构如下：

```
tmp-webapp
├── META-INF
│   └── MANIFEST.MF
├── WEB-INF
│   ├── classes
│   ├── lib
│   │   ├── summer-boot-1.0.3.jar
│   │   └── ... other jars ...
│   └── templates
│       └── ... templates.html
├── application.yml
├── com
│   └── itranswarp
│       └── hello
│           ├── Main.class
│           └── ... other classes ...
├── favicon.ico
├── logback.xml
└── static
    └── ... static files ...
```

可见，`com/itranswarp/hello/Main.class`、`application.yml`、`logback.xml` 都位于war包的根目录，可以被JVM的ClassLoader直接加载，而想要加载 `WEB-INF/lib/summer-boot-1.x.x.jar`，我们需要给出Classpath。通过 `java -jar xyz.war` 启动时，虽然 `-cp` 参数无效，但JVM会自动从 `META-INF/MANIFEST.MF` 中读取 `Class-Path` 条目，我们用Maven写入后内容如下：

```
Manifest-Version: 1.0
Created-By: Maven WAR Plugin 3.3.2
Build-Jdk-Spec: 17
Main-Class: com.itranswarp.hello.Main
Class-Path: tmp-webapp/WEB-INF/lib/summer-boot-1.0.3.jar tmp-webapp/WEB-
INF/lib/summer-web-1.0.3.jar tmp-webapp/WEB-INF/lib/summer-context-1.0.
3.jar tmp-webapp/WEB-INF/lib/snakeyaml-2.0.jar tmp-webapp/WEB-INF/lib/j
ackson-databind-2.14.2.jar tmp-webapp/WEB-INF/lib/jackson-annotations-2
.14.2.jar tmp-webapp/WEB-INF/lib/jackson-core-2.14.2.jar tmp-webapp/WEB
-INF/lib/jakarta.annotation-api-2.1.1.jar tmp-webapp/WEB-INF/lib/slf4j-
api-2.0.7.jar tmp-webapp/WEB-INF/lib/logback-classic-1.4.6.jar tmp-weba
pp/WEB-INF/lib/logback-core-1.4.6.jar tmp-webapp/WEB-INF/lib/freemarker
-2.3.32.jar tmp-webapp/WEB-INF/lib/summer-jdbc-1.0.3.jar tmp-webapp/WEB
```



```
-INF/lib/summer-aop-1.0.3.jar tmp-webapp/WEB-INF/lib/byte-buddy-1.14.2.jar tmp-webapp/WEB-INF/lib/HikariCP-5.0.1.jar tmp-webapp/WEB-INF/lib/tomcat-embed-core-10.1.7.jar tmp-webapp/WEB-INF/lib/tomcat-annotations-api-10.1.7.jar tmp-webapp/WEB-INF/lib/tomcat-embed-jasper-10.1.7.jar tmp-webapp/WEB-INF/lib/tomcat-embed-el-10.1.7.jar tmp-webapp/WEB-INF/lib/ecj-3.32.0.jar tmp-webapp/WEB-INF/lib/sqlite-jdbc-3.41.2.1.jar
```

JVM会读取到 `Main-Class` 和 `Class-Path`，由于已经解压，就能在 `tmp-webapp` 目录中顺利搜索到 `tmp-webapp/WEB-INF/lib/summer-boot-1.x.x.jar`。后续Tomcat启动后，以 `tmp-webapp` 作为web目录本身就是标准的Web App，Tomcat的ClassLoader也能继续从 `WEB-INF/lib` 加载各种jar包。

我们总结一下，打包时做了哪些工作：

- 复制所有编译的class到war包根目录；
- 修改 `META-INF/MANIFEST.MF`：
 - 添加 `Main-Class` 条目；
 - 添加 `Class-Path` 条目。

运行时的流程如下：

1. JVM从war包加载Main类，执行 `main()` 方法；
2. 立刻自解压war包至 `tmp-webapp` 目录；
3. 后续加载 `SummerApplication` 时，JVM根据Class-Path能找到 `tmp-webapp/WEB-INF/lib/summer-boot-1.x.x.jar`，因此可顺利加载；
4. 启动Tomcat，将 `tmp-webapp` 做为Web目录；
5. 作为Web App使用Tomcat的ClassLoader加载其他组件。

这样我们就实现了一个可以直接用 `java -jar xyz.war` 启动的Web应用程序！

有的同学会问，我们的boot应用，`main()` 方法写了一堆自解压代码，而且，需要在 `pom.xml` 中配置很多额外的设置，对比Spring Boot应用，它对 `main()` 方法没有任何要求，而且，在 `pom.xml` 中也只需配置一个 `spring-boot-maven-plugin`，没有其他额外配置，相比之下简单多了，那么，Spring Boot是如何实现的？

我们找一个Spring Boot打包的jar解压后就明白了，它的jar包结构如下：

```
xyz.jar
├── BOOT-INF
│   ├── classes
│   │   ├── application.yml
│   │   ├── logback-spring.xml
│   │   ├── static
│   │   │   └── ... static files ...
│   │   └── templates
│   │       └── ... templates ...
│   └── lib
│       ├── spring-boot-3.0.0.jar
│       └── ... other jars ...
├── META-INF
│   └── MANIFEST.MF
└── org
    └── springframework
        └── boot
            └── loader
                ├── JarLauncher.class
                └── ... other classes ...
```

Spring Boot并不能修改JVM的ClassLoader机制，因此，Spring Boot的jar包仍然需要在 `META-INF/MANIFEST.MF` 中声明 `Main-Class`，只不过它声明的不是应用程序自己的 `Main`，而是Spring Boot的 `JarLauncher`：

```
Main-Class: org.springframework.boot.loader.JarLauncher
```

在jar包的根目录，JVM可以加载 `JarLauncher`。一旦加载了 `JarLauncher` 后，Spring Boot会用自己的ClassLoader去加载其他的class和jar包，它在 `BOOT-INF/classes` 和 `BOOT-INF/lib` 下搜索。注意Spring Boot自定义的ClassLoader并不需要设置 `Class-Path`，它可以完全自定义搜索路径，包括搜索“jar包中的jar包”。

因此，Spring Boot采用了两种机制来实现可执行jar包：

1. 提供Maven插件，自动设置 `Main-Class`，复制相关启动Class，按 `BOOT-INF` 组织class和jar包；
2. 提供自定义的ClassLoader，可以在jar包中搜索 `BOOT-INF/classes` 和 `BOOT-INF/lib`。

这样就使得编写Web应用程序时能简化打包和启动流程。代价就是编写一个自定义的Maven插件和自定义的ClassLoader工作量很大，有兴趣的同学可以试着实现Spring Boot的机制。

参考源码

可以从[GitHub](https://github.com)或[Gitee](https://gitee.com)下载源码。

GitHub

评论

期末总结

[原文链接](#)

终于到了期末总结的时刻了！

通过开发一个迷你版Spring Framework，相信大家对Spring框架又有了更深刻的理解。通过自己从零开始手写Summer Framework，写完后应该完全可胜任Java架构师这样的高级职位！



[评论](#)



手写Spring

自己动手，从零开发一个迷你版Spring框架！

Author: 廖雪峰

Version: 2025-06-16

Website: <https://liaoxuefeng.com/books/summerframework/>