Chapter 1. Spring概述

Chapter 2. IOC 容器

2.1. 接口详解

2.1.1. 顶层接口

2.1.1.1. BeanFactory

1、功能

1) BeanFactory 是 Spring bean 容器的根接口
2) 提供获取 bean、是否包含 bean、是否单例与原型、获取 bean 类型、bean 别名的 API

2、直系子接口

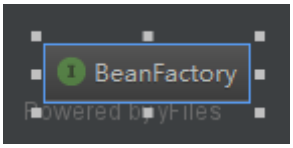
1) HierarchicalBeanFactory
2) AutowireCapableBeanFactory
3) ListableBeanFactory

3、直系抽象子类：无

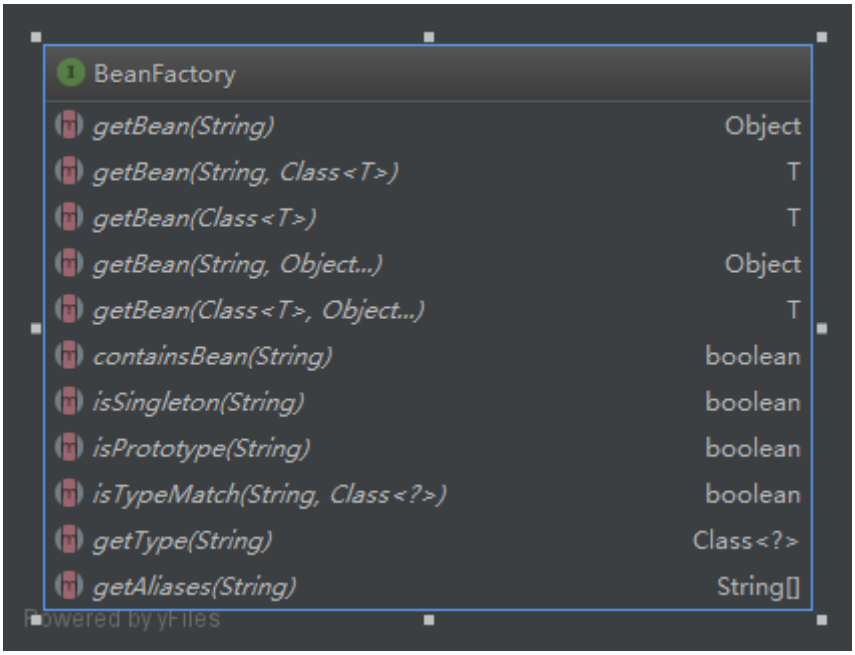
4、直系实现子类

1) SimpleJndiBeanFactory

5、UML 类图



6、接口展示



2.1.1.2. SingletonBeanRegistry

1、功能

2、直系子接口

1) ConfigurableBeanFactory

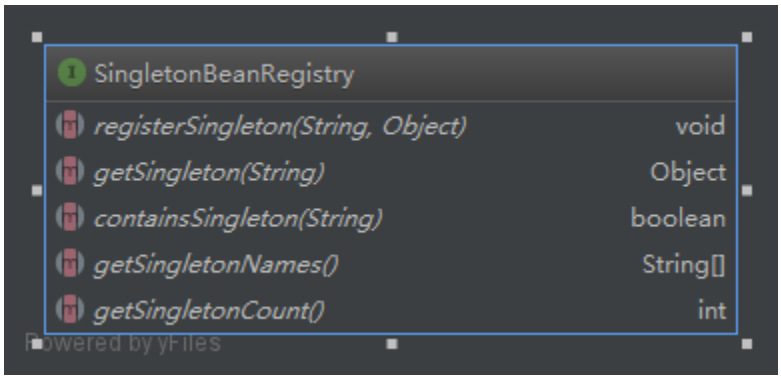
3、直系抽象子类：无

4、直系实现子类

1) DefaultSingletonBeanRegistry

5、UML 类图

6、接口展示



2.1.1.3. ResourceLoader

1、功能

2、直系子接口

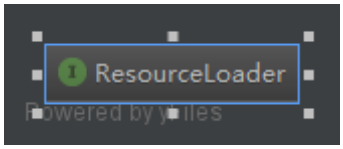
1) ResourcePatternResolver

3、直系抽象子类：无

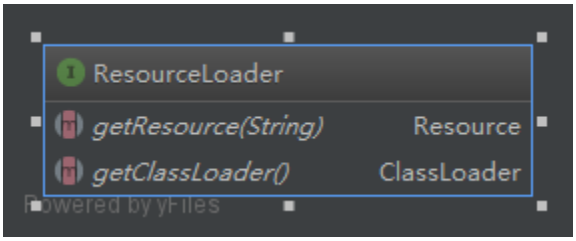
4、直系实现子类

1) DefaultResourceLoader

5、UML 类图



6、接口展示



1、功能

2、直系子接口

3、直系抽象子类

4、直系实现子类

5、UML 类图

6、接口展示

2.1.2. 中层接口

2.1.2.1. HierarchicalBeanFactory

1、功能

1) 提供父容器的访问功能
2) 至于父容器的设置，需要找 ConfigurableBeanFactory 的 setParentBeanFactory(接口把设置跟获取给拆开了)

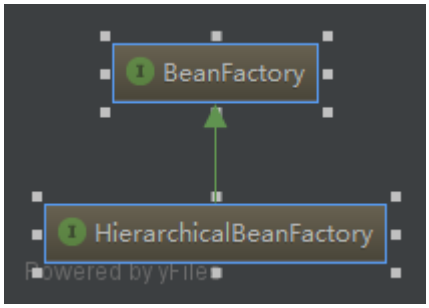
2、直系子接口

1) ConfigurableBeanFactory
2) ApplicationContext

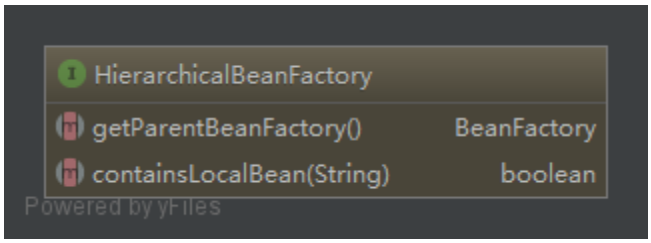
3、直系抽象子类：无

4、直系实现子类：无

5、UML 类图



6、接口展示



2.1.2.2. AutowireCapableBeanFactory

1、功能

1) 在 BeanFactory 基础上实现对已存在实例的管理
2) 可以使用这个接口集成其它框架,捆绑并填充并不由 Spring 管理生命周期并已存在的实例。像集成 WebWork 的 Actions 和 Tapestry Page 就很实用
3) 一般应用开发者不会使用这个接口，所以像 ApplicationContext 这样的外观实现类不会实现这个接口

2、直系子接口

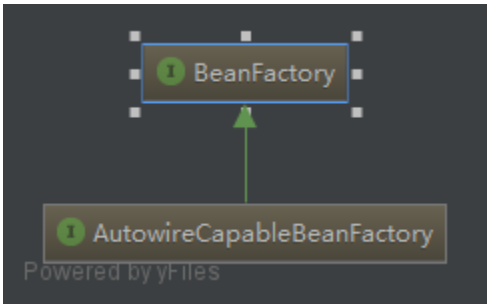
1) ConfigurableListableBeanFactory

3、直系抽象子类

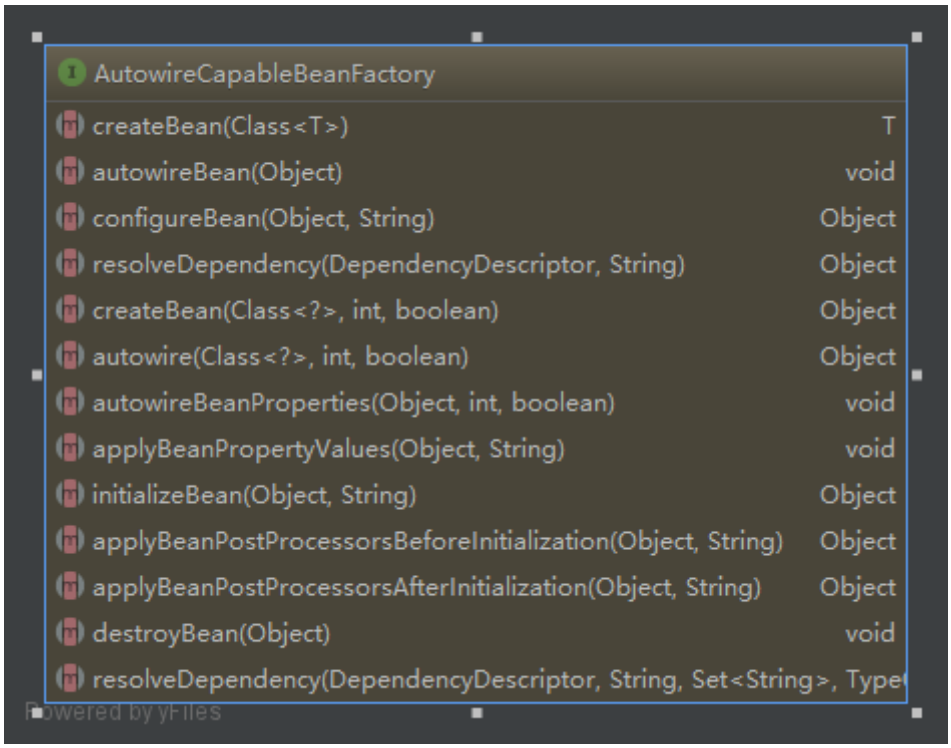
1) AbstractAutowireCapableBeanFactory

4、直系实现子类：无

5、UML 类图



6、接口展示



2.1.2.3. ListableBeanFactory

1、功能

1) 提供容器内 bean 实例的枚举功能。这边不会考虑父容器内的实例
2) 提供容器中 bean 迭代的功能，不再需要一个个 bean 地查找。比如可以一次获取全部的 bean(太暴力了)
3) 根据类型获取 bean。在看 SpringMVC 时，扫描包路径下的具体实现策略就是使用的这种方式(那边使用的是 BeanFactoryUtils 封装的 api)
4) 如果同时实现了 HierarchicalBeanFactory，返回值不会考虑父类 BeanFactory，只考虑当前 factory 定义的类。当然也可以使用 BeanFactoryUtils 辅助类来查找祖先工厂中的类
5) 这个接口中的方法只会考虑本 factory 定义的 bean。这些方法会忽略 ConfigurableBeanFactory 的 registerSingleton 注册的单例 bean(getBeanNamesOfType 和 getBeansOfType 是例外，一样会考虑手动注册的单例).当然 BeanFactory 的 getBean 一样可以透明访问这些特殊 bean. 当然在典型情况下，所有的 bean 都是由 external bean 定义，所以应用不需要顾虑这些差别

2、直系子接口

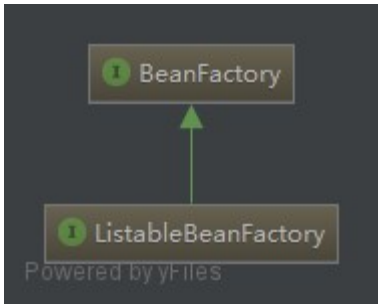
1) ApplicationContext
2) ConfigurableListableBeanFactory

3、直系抽象子类：无

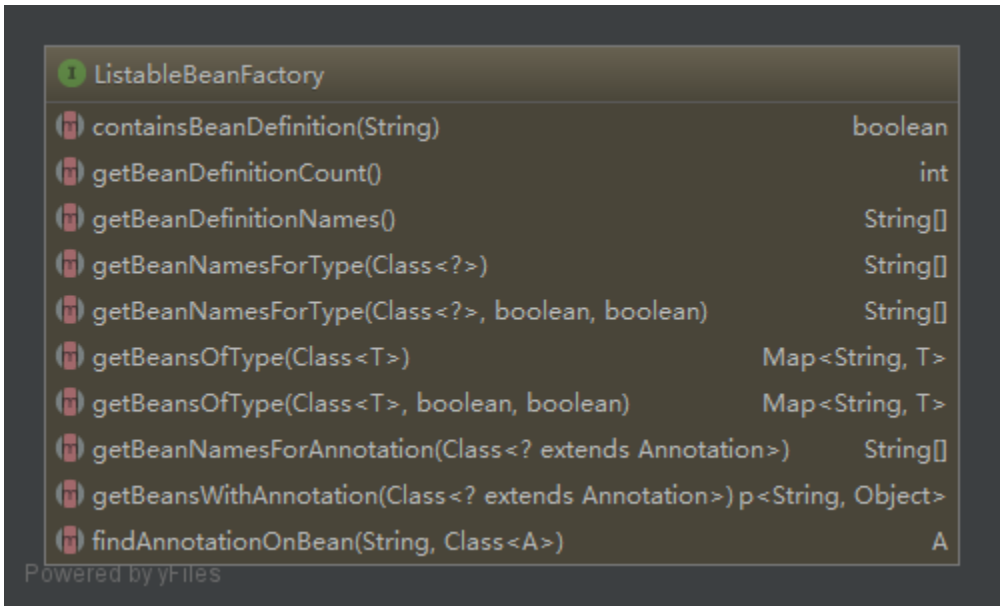
4、直系实现子类

1) StaticListableBeanFactory

5、UML 类图



6、接口展示



2.1.2.4. ConfigurableBeanFactory

1、功能

1) 这边定义了太多太多的 api，比如类加载器，类型转化，属性编辑器，BeanPostProcessor，作用域，bean 定义，处理 bean 依赖关系，合并其他 ConfigurableBeanFactory，bean 如何销毁

2、直系子接口

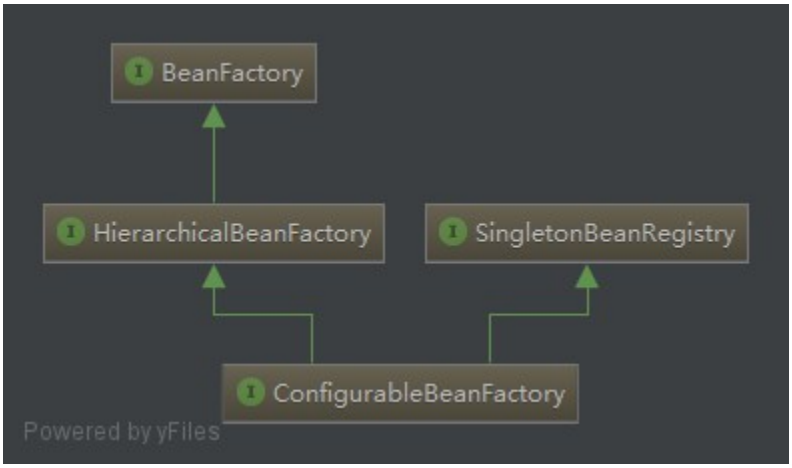
1) ConfigurableListableBeanFactory

3、直系抽象子类

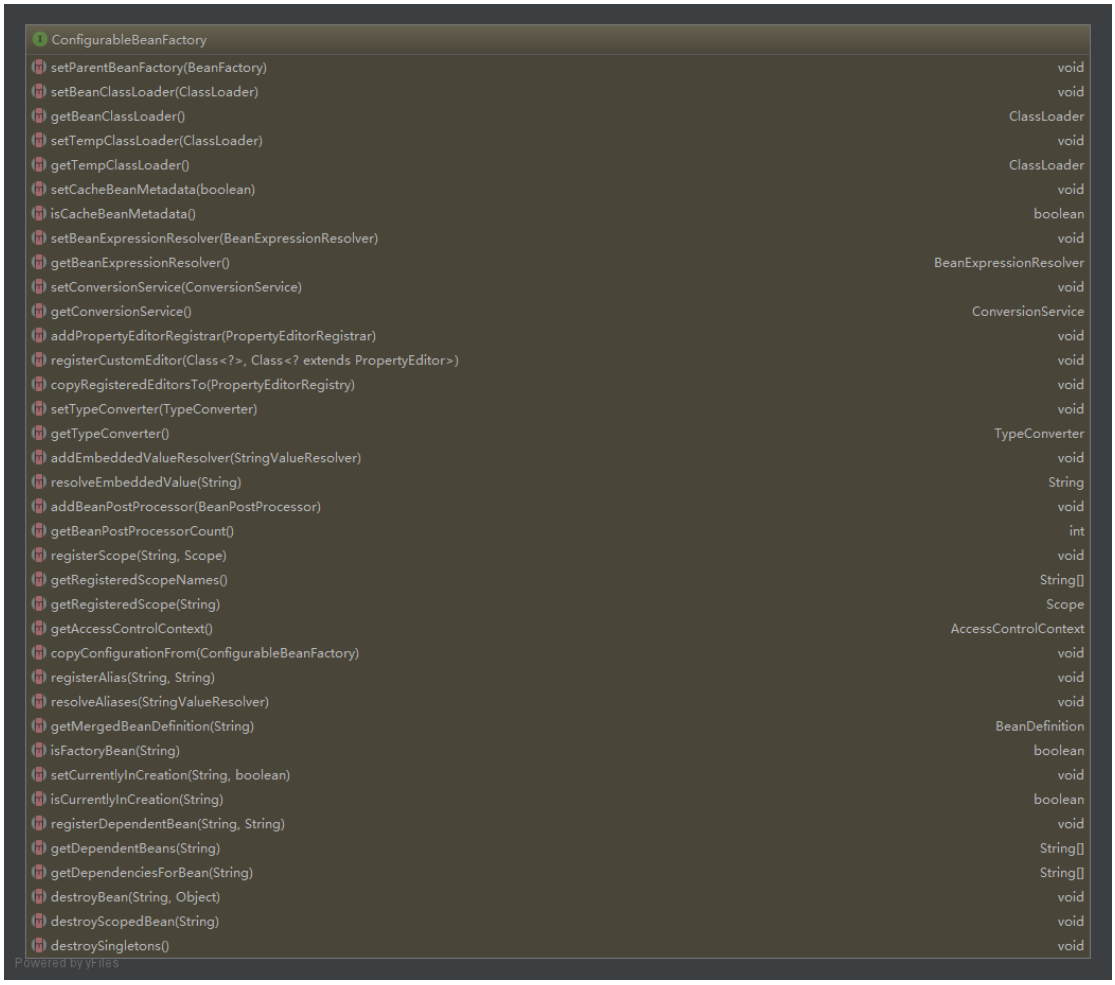
1) AbstractBeanFactory

4、直系实现子类：无

5、UML 类图



6、接口展示



2.1.2.5. ApplicationContext

1、功能

1)

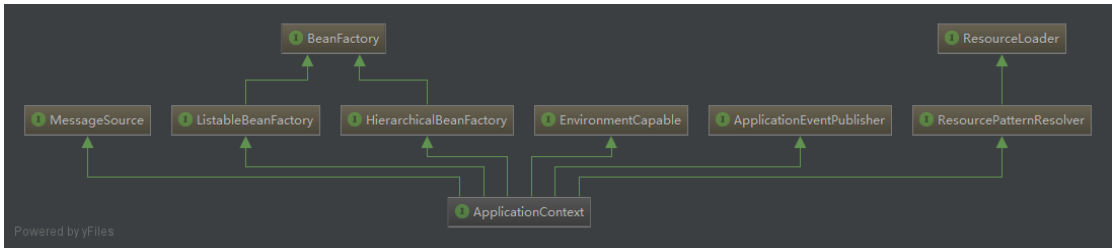
2、直系子接口

1) ConfigurableApplicationContext

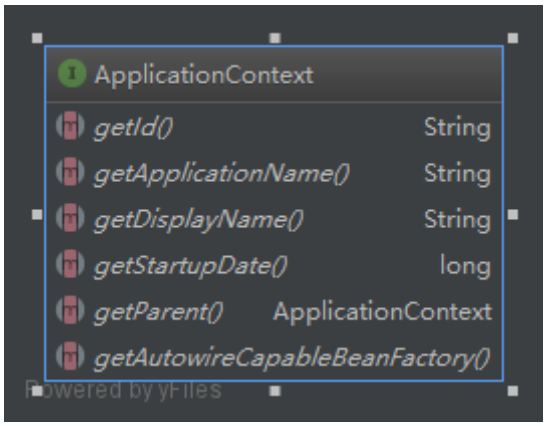
3、直系抽象子类：无

4、直系实现子类：无

5、UML 类图



6、接口展示



2.1.2.6. ConfigurableListableBeanFactory

1、功能

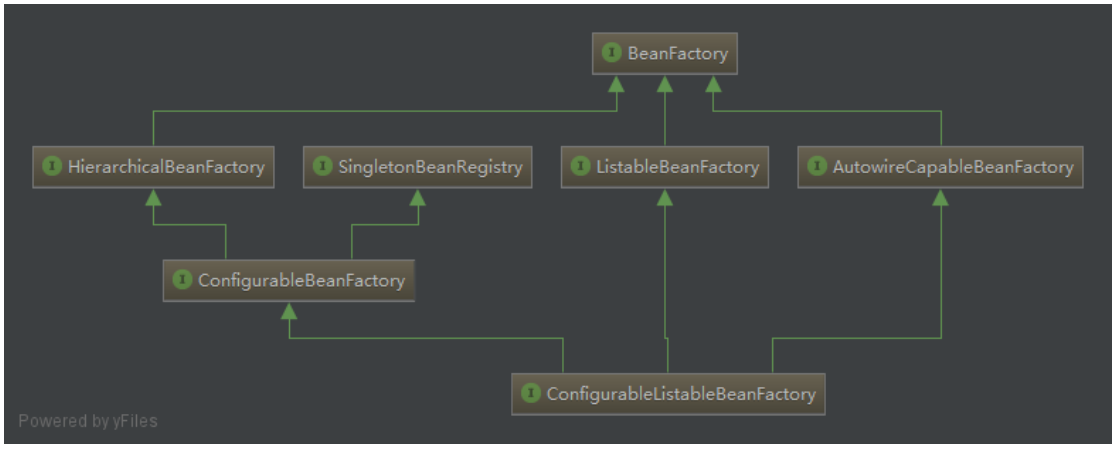
2、直系子接口：无

3、直系抽象子类：无

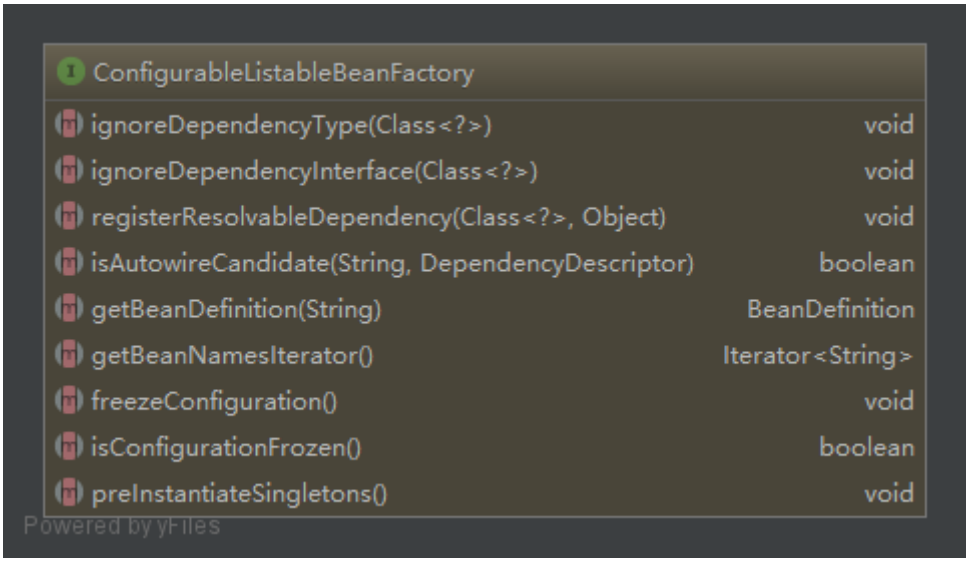
4、直系实现子类

1) DefaultListableBeanFactory

5、UML 类图



6、接口展示



2.1.2.7. ConfigurableApplicationContext

1、功能

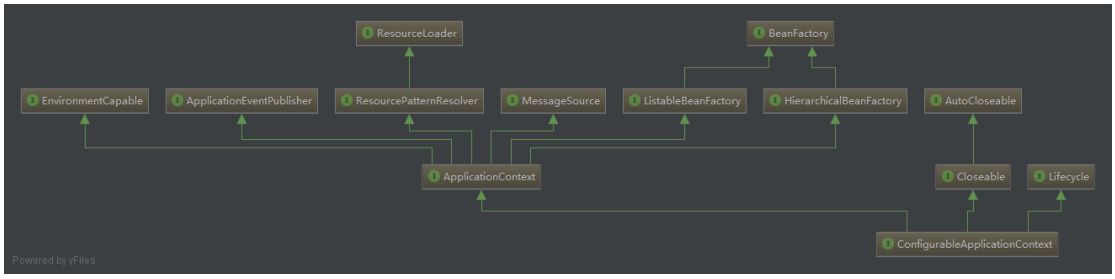
2、直系子接口：无

3、直系抽象子类

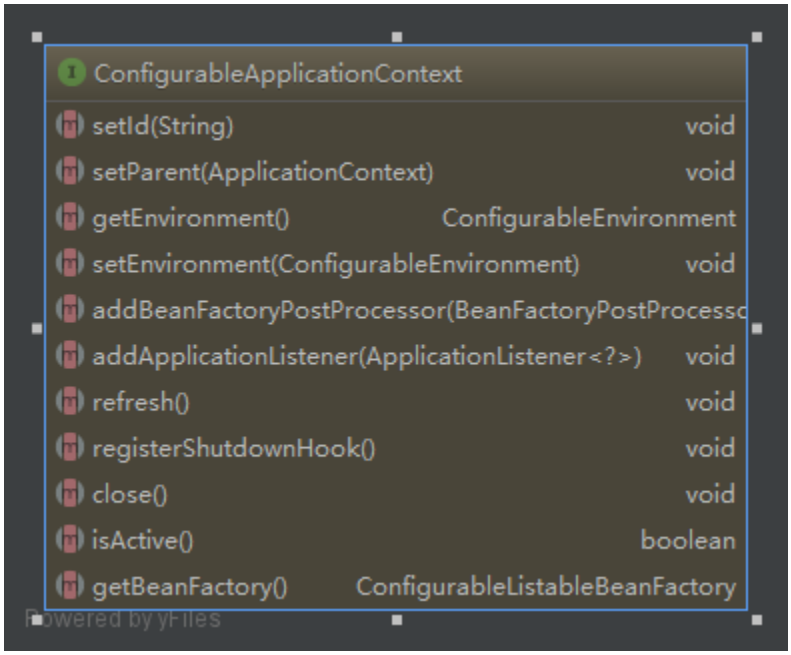
1) AbstractApplicationContext

4、直系实现子类：无

5、UML 类图



6、接口展示



2.1.2.8. ResourcePatternResolver

1、功能

2、直系子接口

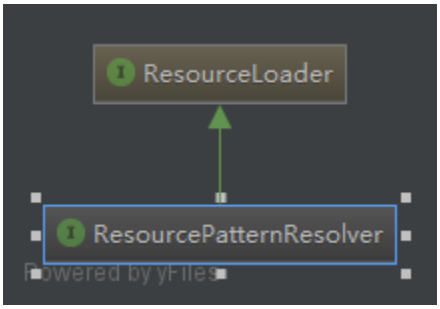
1) ApplicationContext

3、直系抽象子类：无

4、直系实现子类

1) PathMatchingResourcePatternResolver

5、UML 类图



6、接口展示

1、功能

2、直系子接口

3、直系抽象子类

4、直系实现子类

5、UML 类图

6、接口展示

2.2. 抽象类

2.2.1. AbstractAutowireCapableBeanFactory

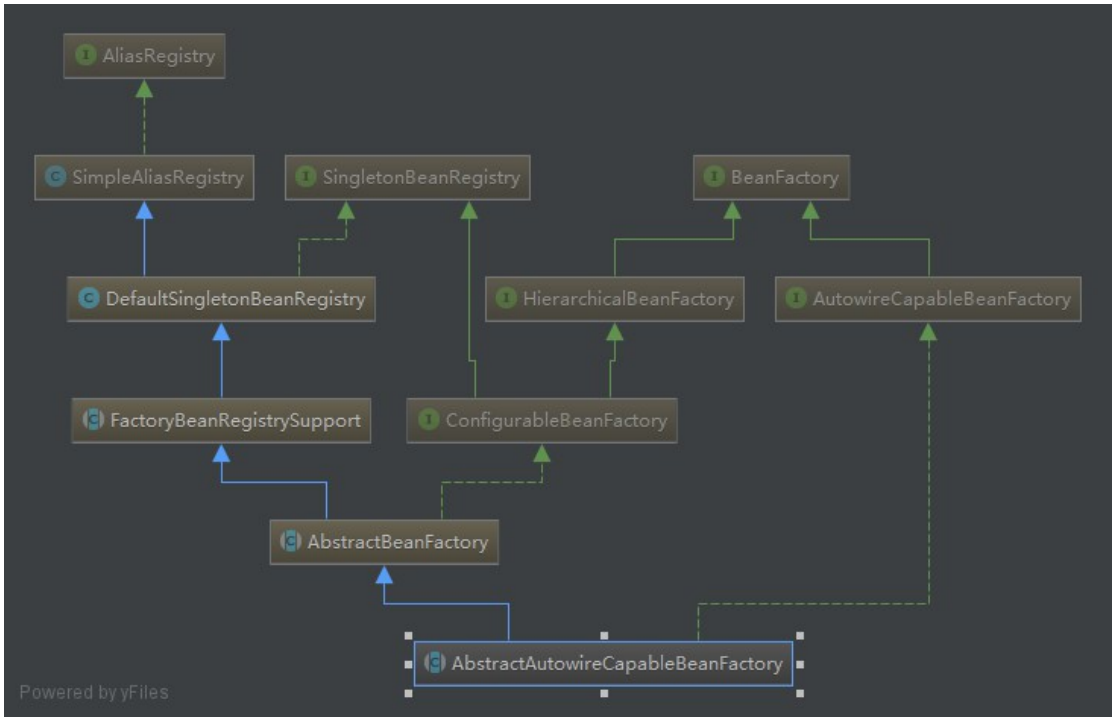
1、功能

2、直系抽象子类：无

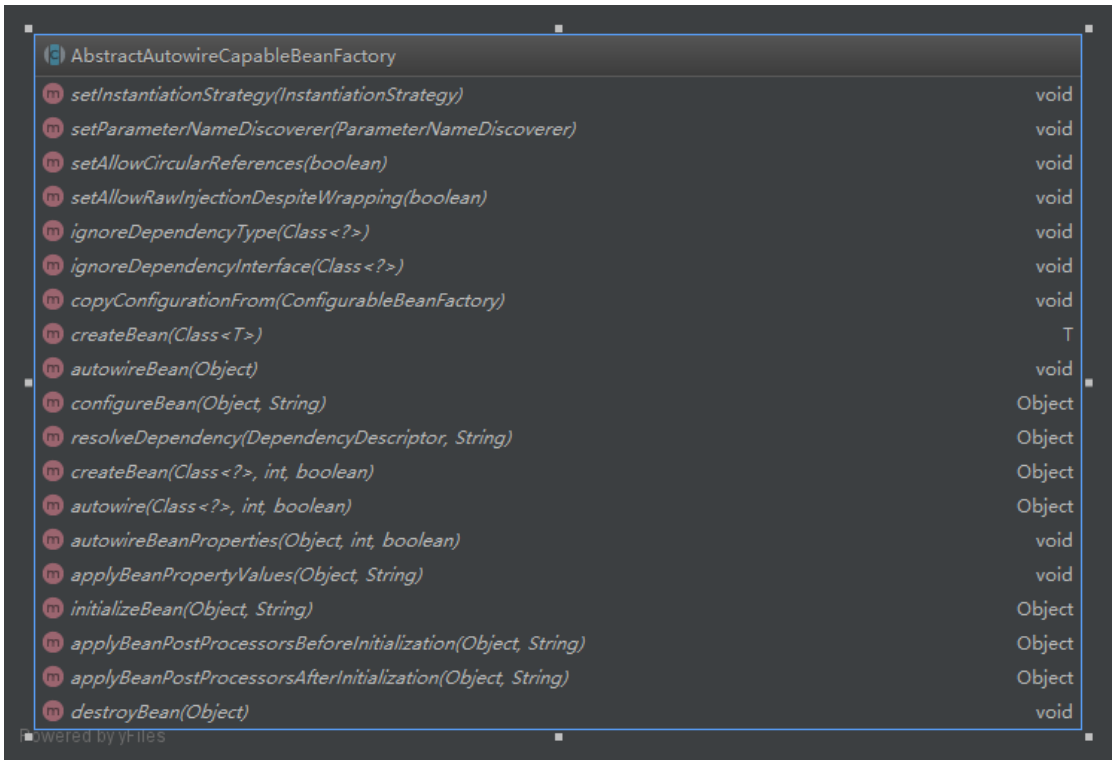
3、直系实现子类

1) DefaultListableBeanFactory

4、UML 类图



5、接口展示



2.2.2. AbstractBeanFactory

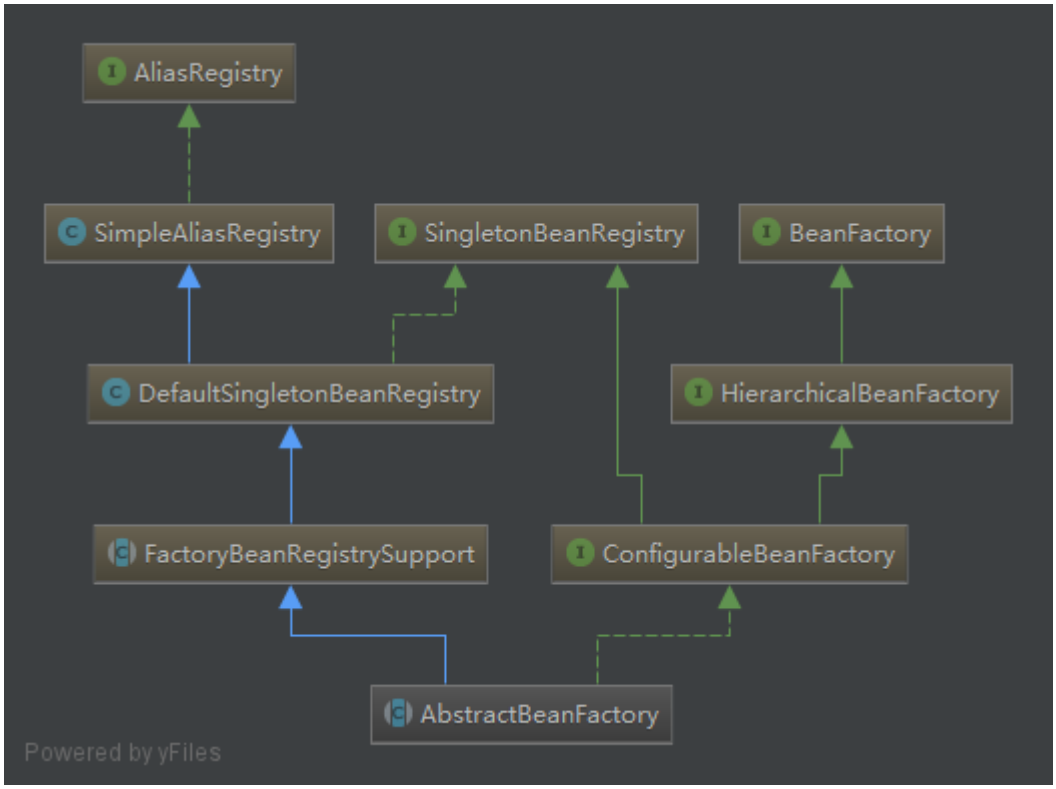
1、功能

2、直系抽象子类

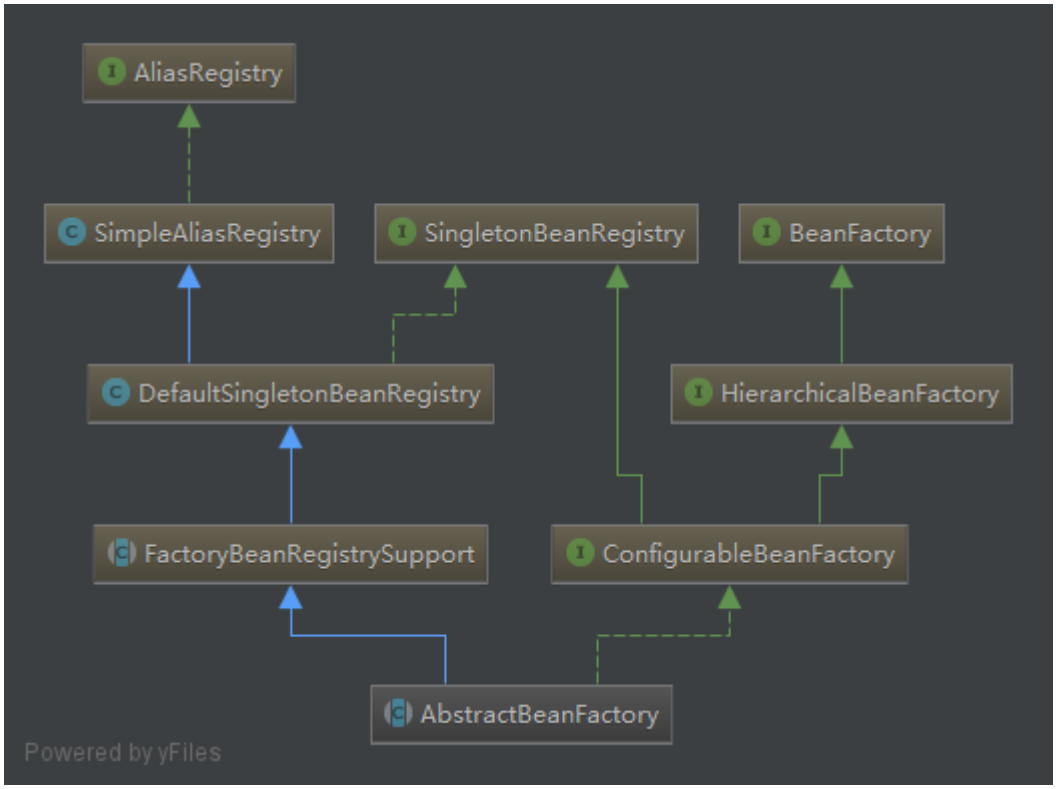
1) AbstractAutowireCapableBeanFactory

3、直系实现子类：无

4、UML 类图



5、接口展示



2.2.3. AbstractApplicationContext

1、功能

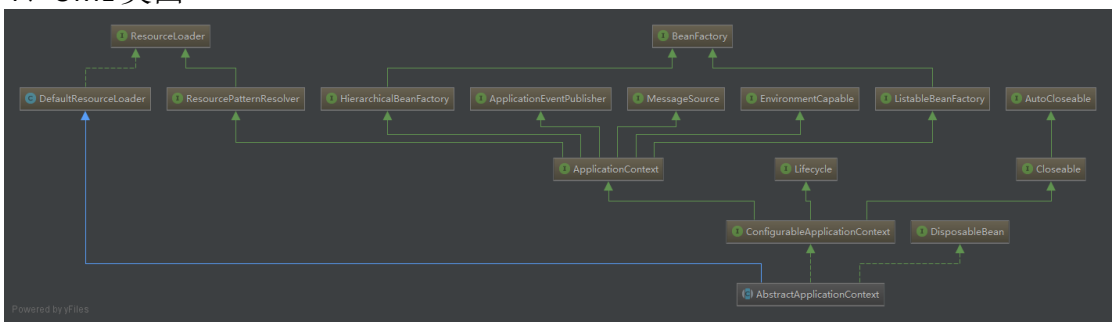
2、直系抽象子类

1) AbstractRefreshableApplicationContext

3、直系实现子类

1) GenericApplicationContext

4、UML 类图



5、接口展示

2.2.4. AbstractRefreshableApplicationContext

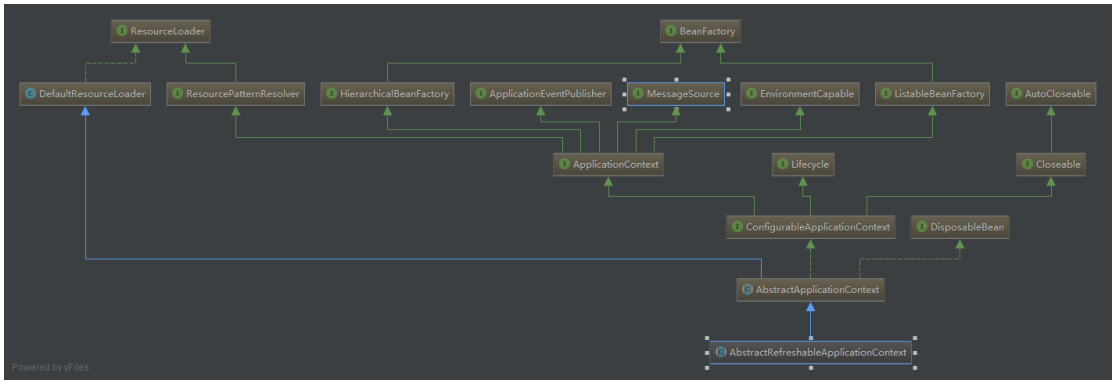
1、功能

2、直系抽象子类

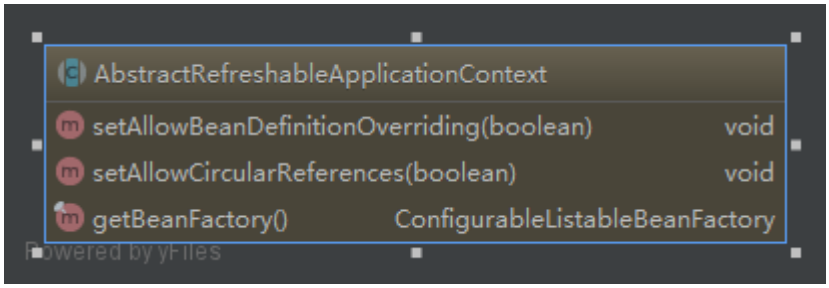
1) AbstractRefreshableConfigApplicationContext

3、直系实现子类：无

4、UML 类图



5、接口展示



2.2.5. AbstractRefreshableConfigApplicationContext

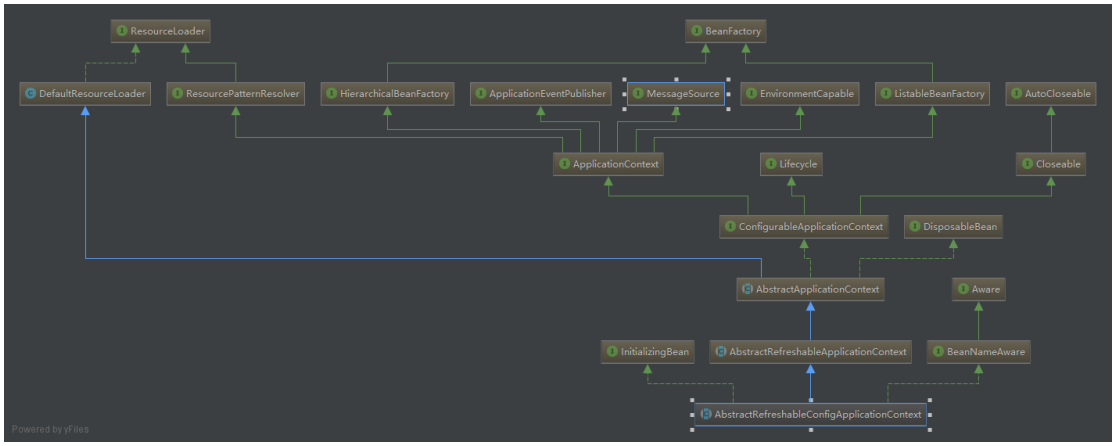
1、功能

2、直系抽象子类

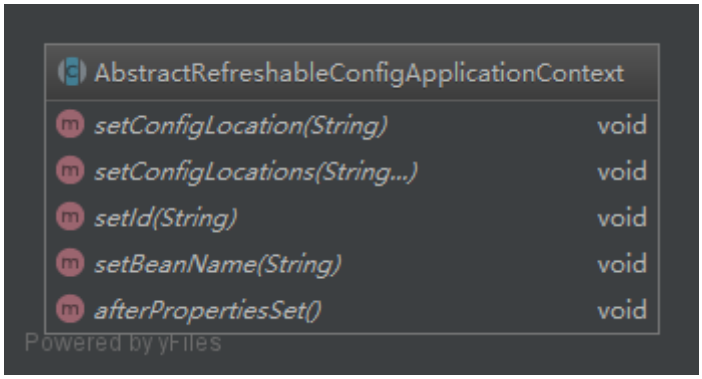
1) AbstractXmlApplicationContext

3、直系实现子类：无

4、UML 类图



5、接口展示



2.2.6. AbstractXmlApplicationContext

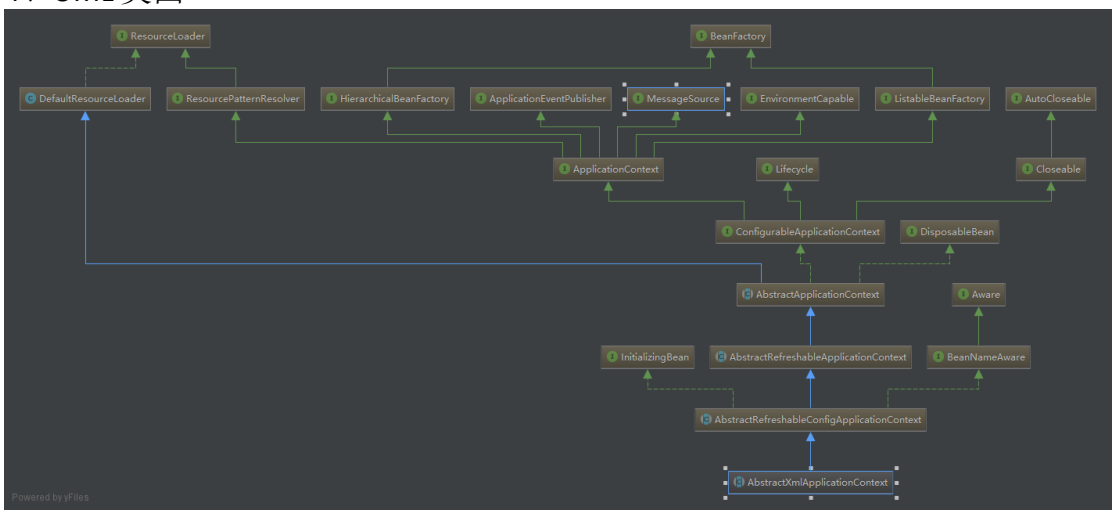
1、功能

2、直系抽象子类：无

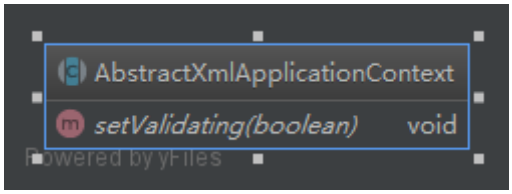
3、直系实现子类

1) FileSystemXmlApplicationContext
2) ClassPathXmlApplicationContext

4、UML 类图



5、接口展示



1、功能

2、直系抽象子类

3、直系实现子类

4、UML 类图

5、接口展示

2.3. 实现类详解

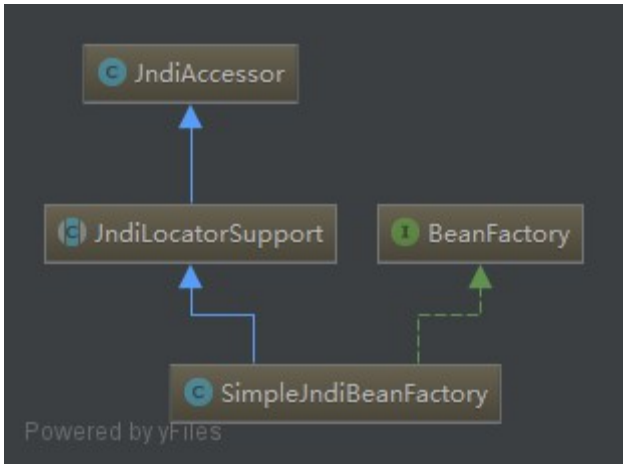
2.3.1. SimpleJndiBeanFactory

1、功能

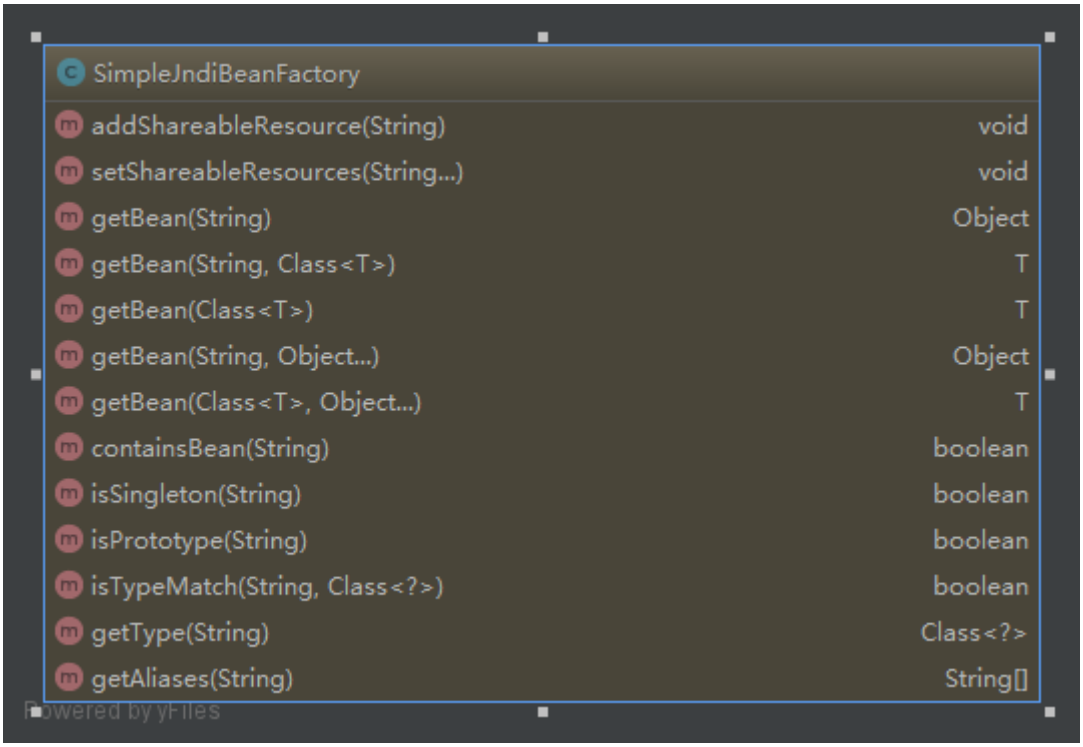
2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示



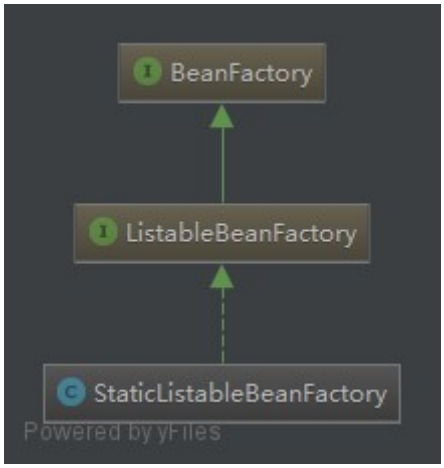
2.3.2. StaticListableBeanFactory

1、功能

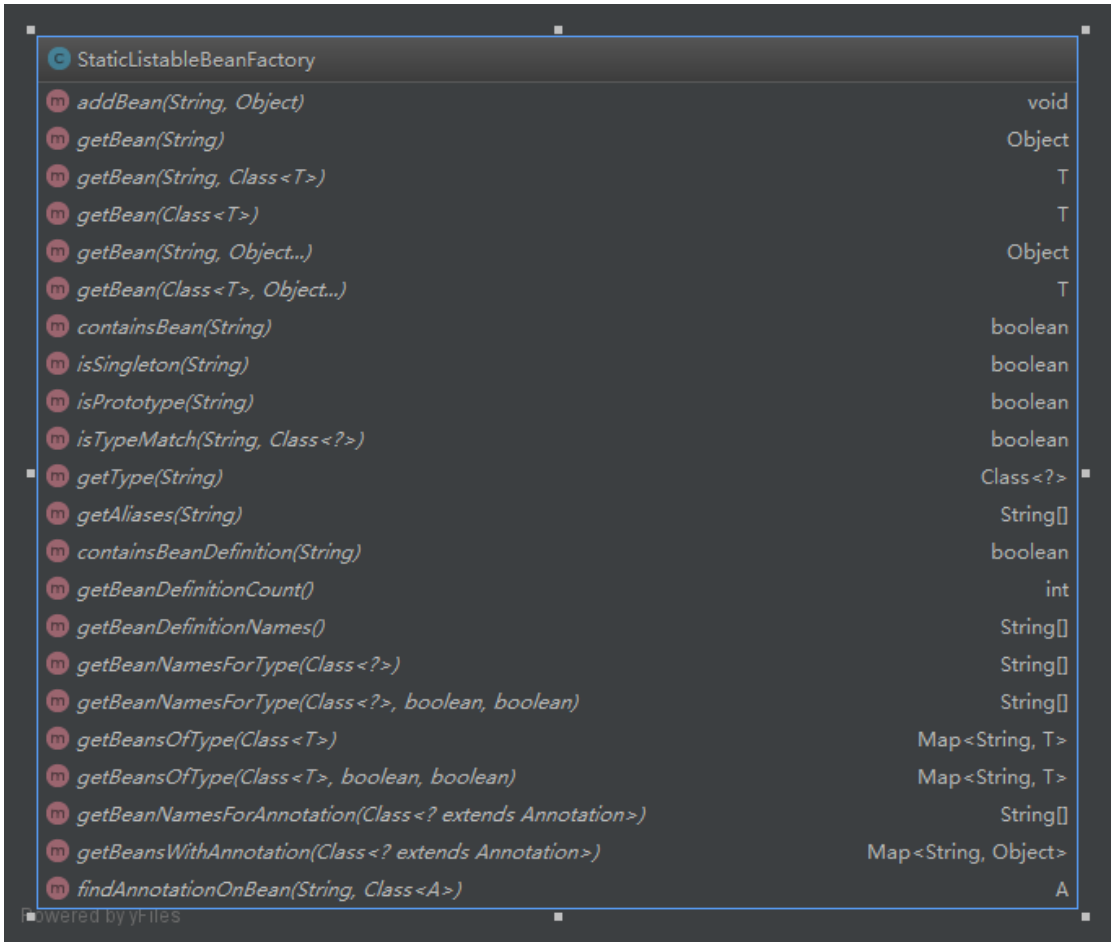
2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示



2.3.3. DefaultListableBeanFactory

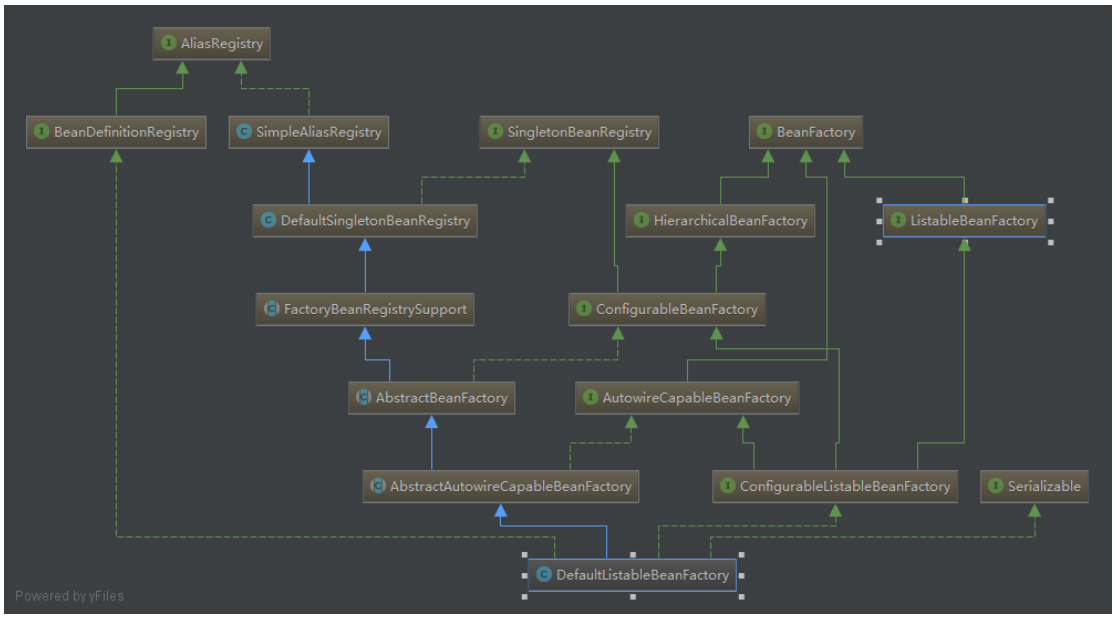
1、功能

2、直系抽象子类

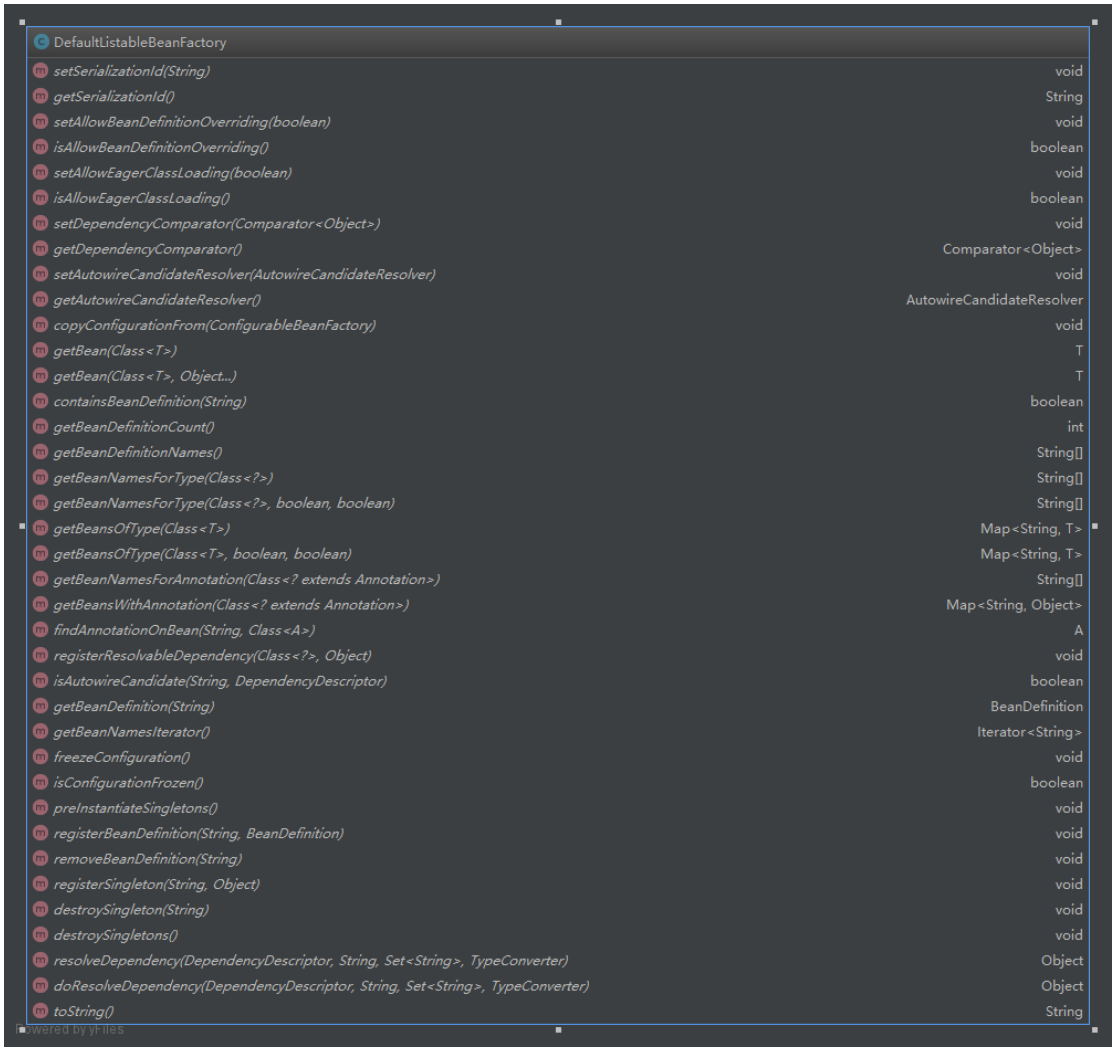
3、直系实现子类

1) XmlBeanFactory

4、UML 类图



5、接口展示



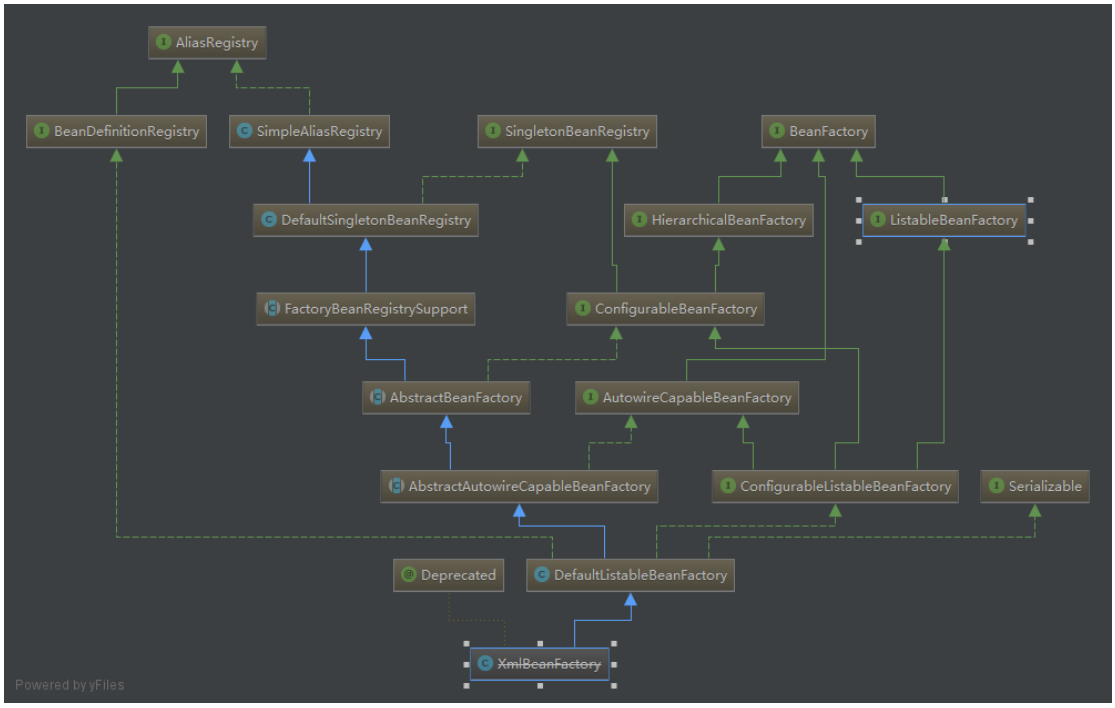
2.3.4. XmlBeanFactory

1、功能

2、直系抽象子类

3、直系实现子类：无

4、UML 类图



5、接口展示：除了构造函方法，没有定义任何方法

2.3.5. GenericApplicationContext

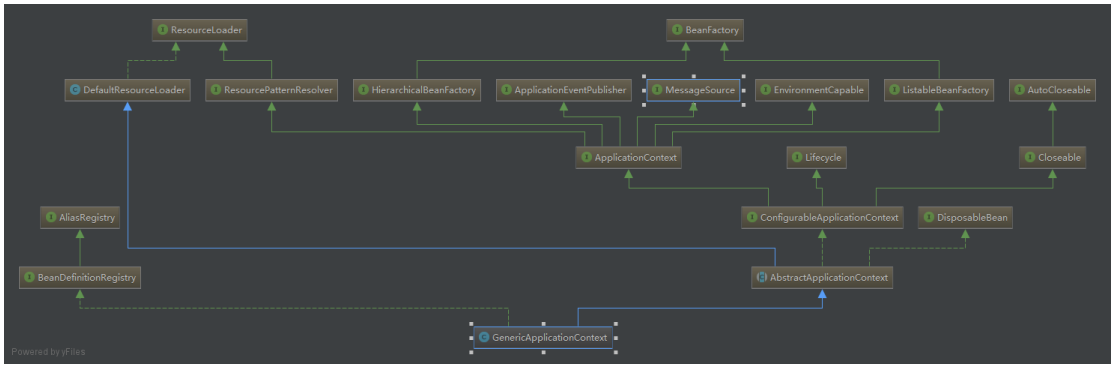
1、功能

2、直系抽象子类：无

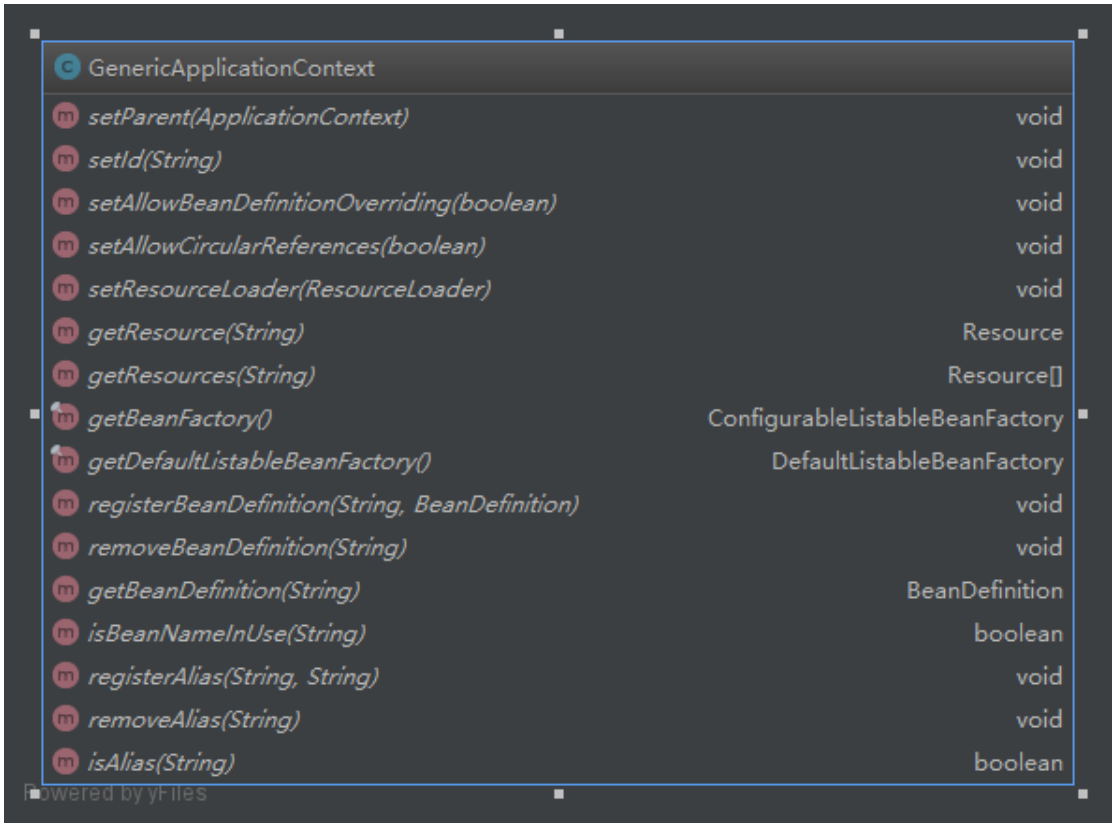
3、直系实现子类

1) GenericXmlApplicationContext
2) StaticApplicationContext
3) GenericGroovyApplicationContext
4) AnnotationConfigApplicationContext

4、UML 类图



5、接口展示



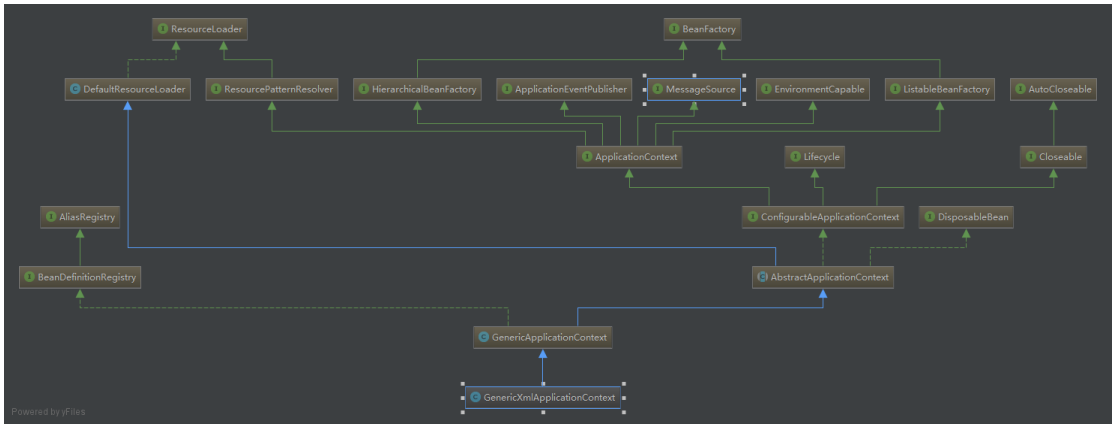
2.3.6. GenericXmlApplicationContext

1、功能

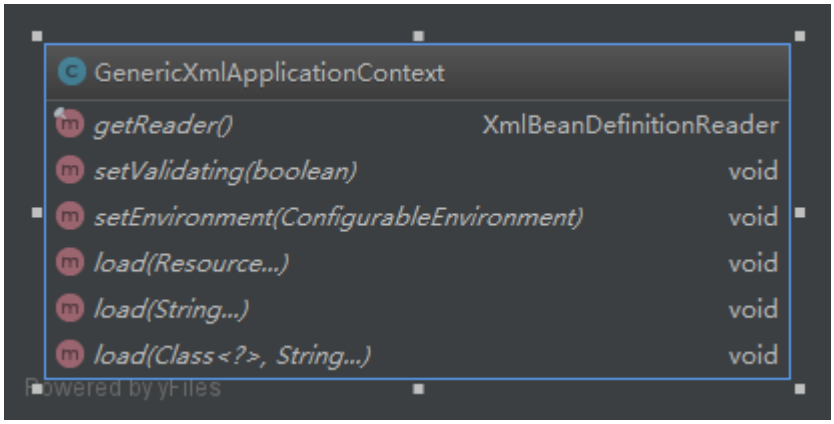
2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示



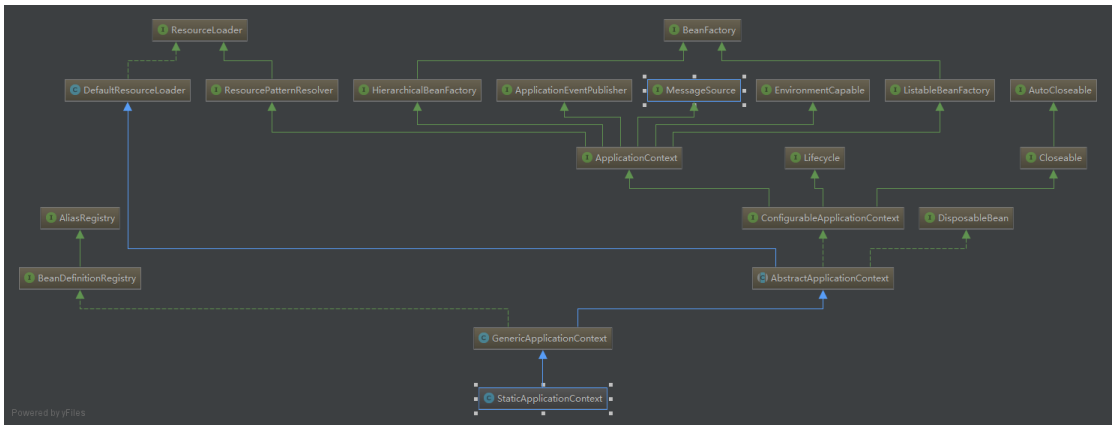
2.3.7. StaticApplicationContext

1、功能

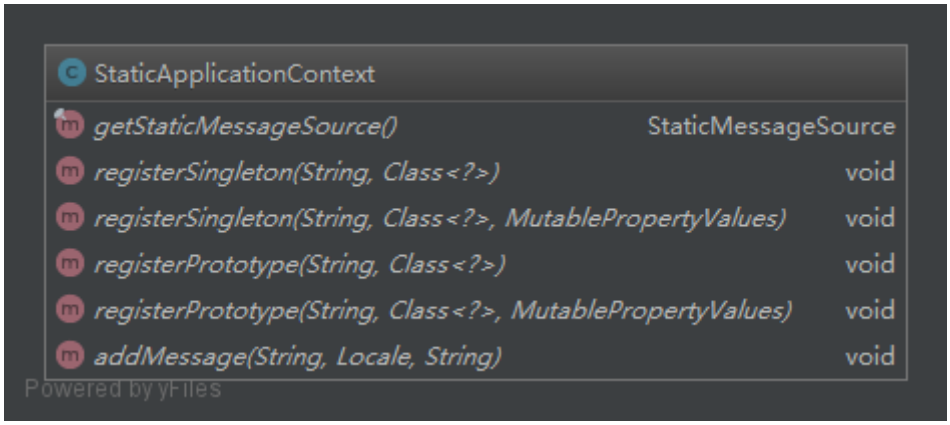
2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示



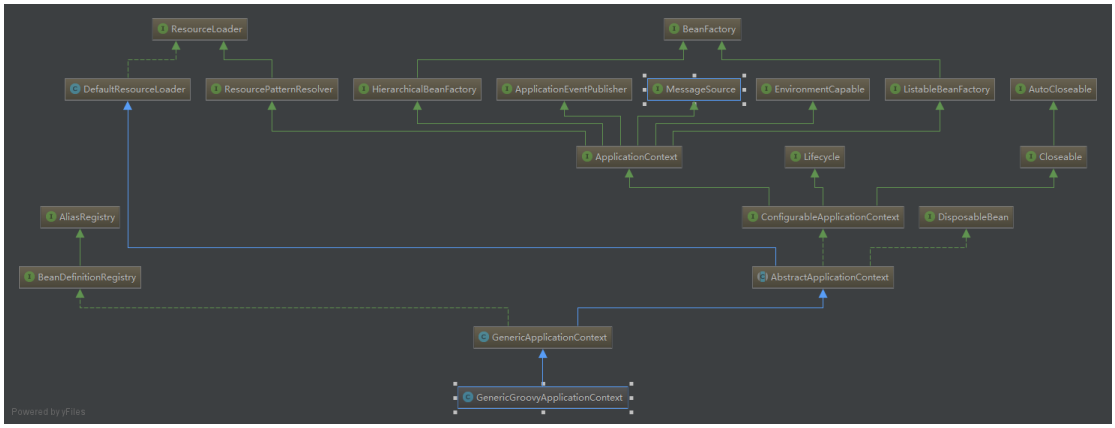
2.3.8. GenericGroovyApplicationContext

1、功能

2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示

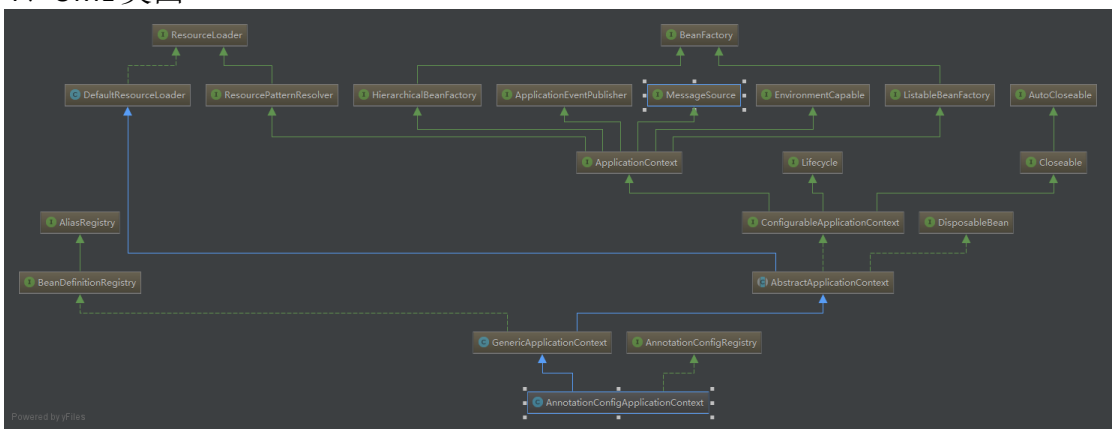
2.3.9. AnnotationConfigApplicationContext

1、功能

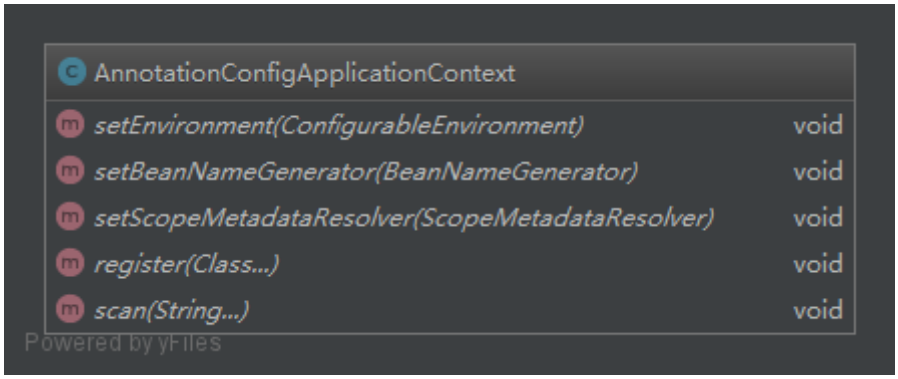
2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示



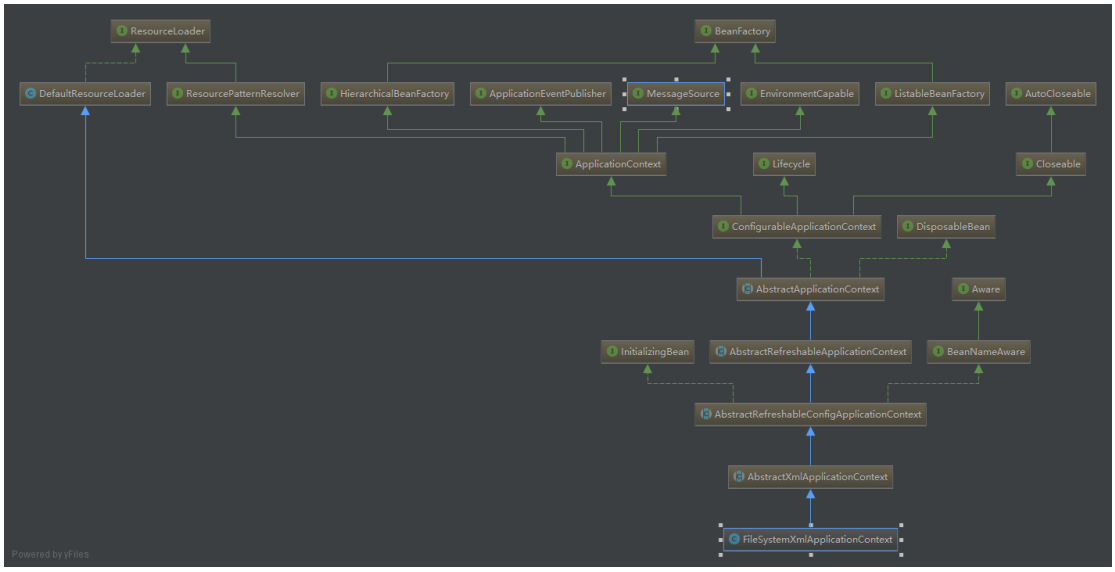
2.3.10. FileSystemXmlApplicationContext

1、功能

2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示：除了构造函方法，没有定义任何方法

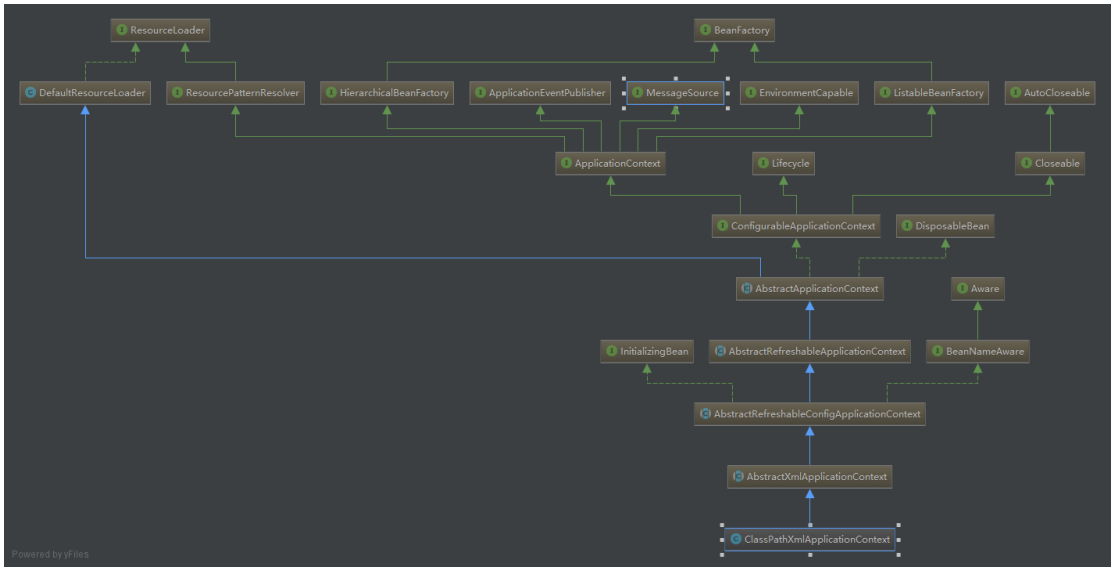
2.3.11. ClassPathXmlApplicationContext

1、功能

2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示：除了构造函方法，没有定义任何方法

2.3.12. DefaultResourceLoader

1、功能

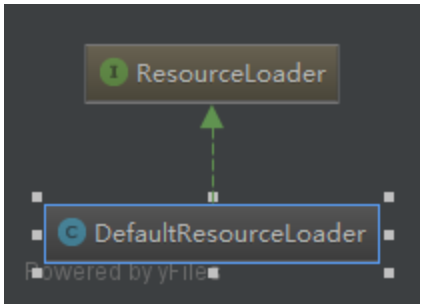
2、直系抽象子类

1) AbstractApplicationContext

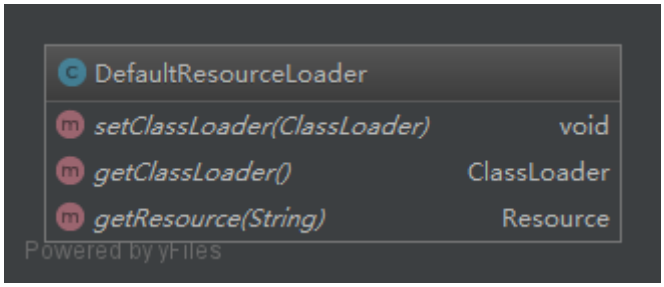
3、直系实现子类

1) ClassRelativeResourceLoader
2) FileSystemResourceLoader

4、UML 类图



5、接口展示



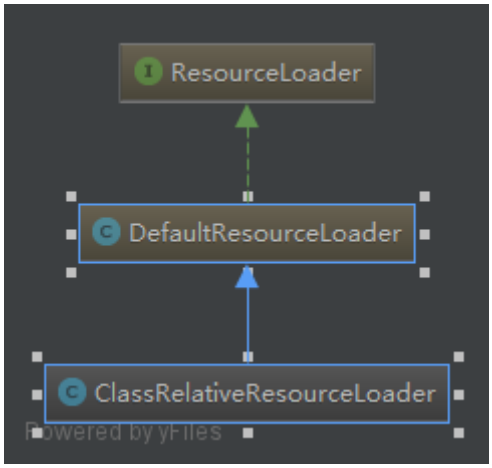
2.3.13. ClassRelativeResourceLoader

1、功能

2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示：除了构造函方法，没有定义任何方法

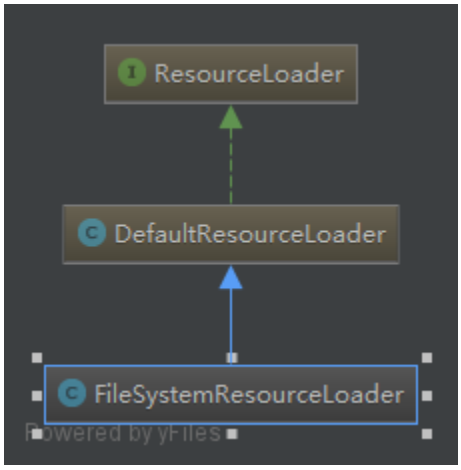
2.3.14. FileSystemResourceLoader

1、功能

2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



5、接口展示：除了构造函方法，没有定义任何方法

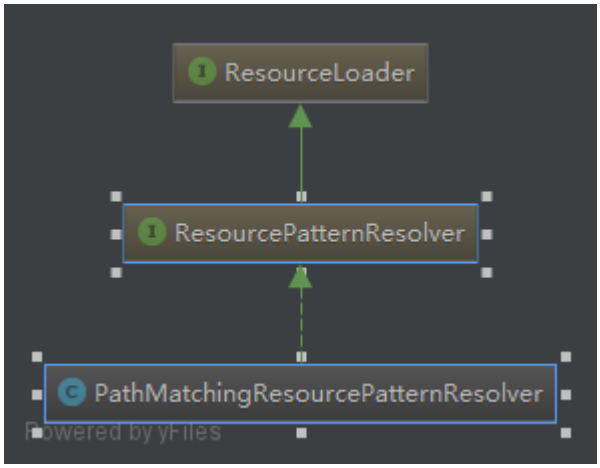
2.3.15. PathMatchingResourcePatternResolver

1、功能

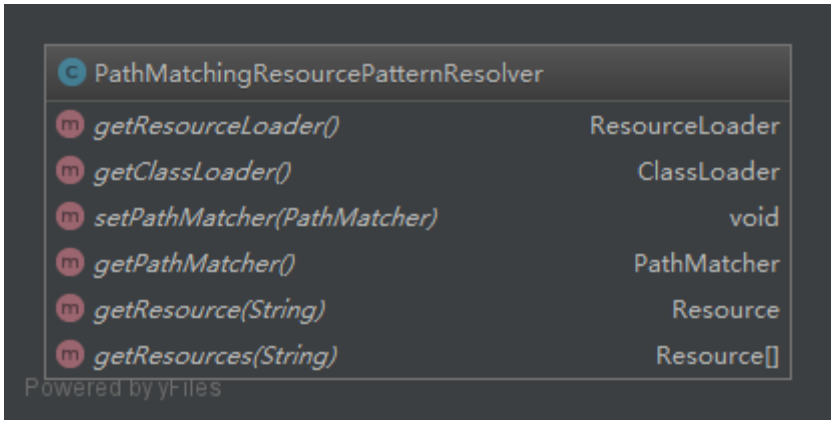
2、直系抽象子类：无

3、直系实现子类：无

4、UML 类图



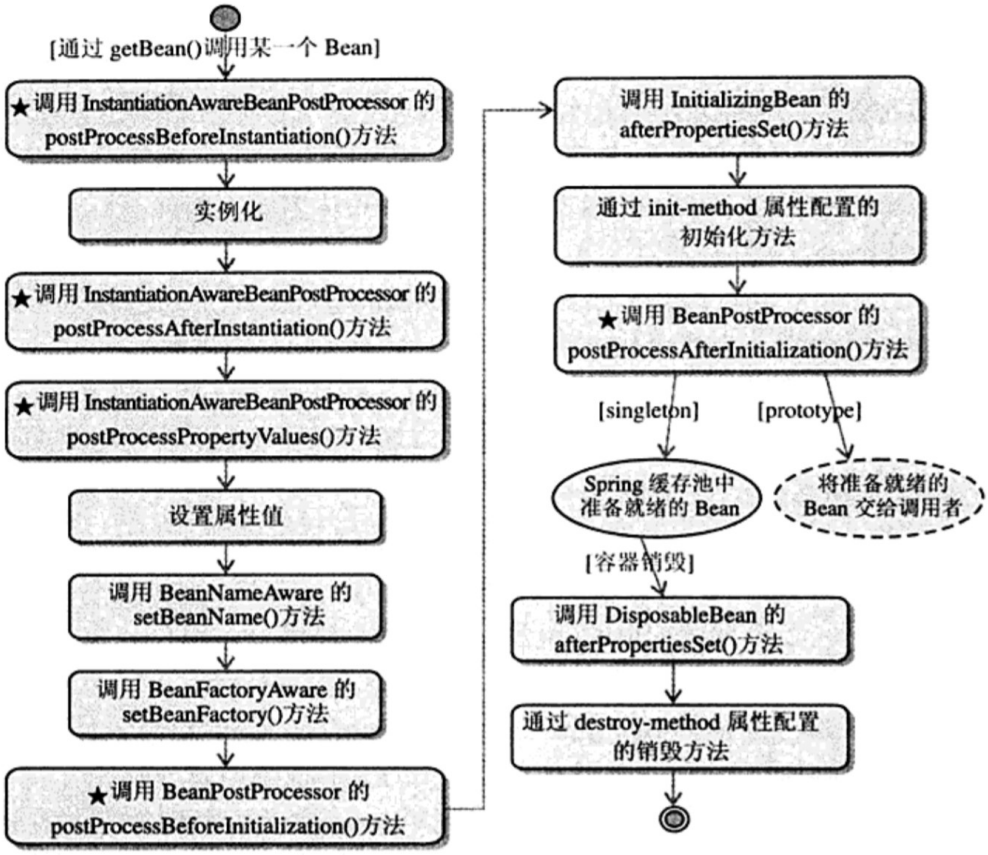
5、接口展示



1、功能
2、直系抽象子类
3、直系实现子类
4、UML 类图
5、接口展示

Chapter 3. Spring Boot

Chapter 4. IoC 容器



1、标星的步骤由 InstantiationAwareBeanPostProcessor 和 BeanPostProcessor 这两个接口实现，一般称它们的实现类为"后处理器"。后处理器接口一般不由 Bean 本身实现，它们独立于 Bean，实现类以容器附加装置的形式注册到 Spring 容器中，并通过接口反射为 Spring 容器扫描识别。当 Spring 容器创建任何 Bean 时，这些后处理器都会发生作用，所以这些后处理器的影响是全局性的。当然用户可以通过合理地编写后处理器，让其仅对感兴趣的 Bean 进行加工处理

Chapter 5. 在 IoC 容器中装配 Bean

Chapter 6. Spring容器高级主题

6.1.Spring容器技术内幕

Chapter 7. Spring AOP基础

Chapter 8. 基于@AspectJ和Schema的AOP

Chapter 9. Spring SpEL

Chapter 10. Spring对DAO的支持

Chapter 11. Spring的事务管理

Chapter 12. Spring的事务管理难点剖析

Chapter 13. 使用 Spring JDBC 访问

Chapter 14. 整合其他 ORM 框架

14.1. 整合 Mybatis

1、Maven 依赖

```
<properties>
    <spring.version>4.1.6.RELEASE</spring.version>
    <mybatis.version>3.2.3</mybatis.version>
    <mybatis.spring.version>1.2.1</mybatis.spring.version>
    <slf4j.version>1.7.5</slf4j.version>
    <log.version>1.2.16</log.version>
    <junit.version>4.11</junit.version>
</properties>

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

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>

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

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <!-- 测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
    </dependency>

    <!-- 日志 -->
    <dependency>
```

```
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>${slf4j.version}</version>
        </dependency>

        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>${log.version}</version>
        </dependency>

        <!-- mybatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>${mybatis.version}</version>
        </dependency>

        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>${mybatis.spring.version}</version>
        </dependency>

        <!-- mysql 驱动包 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.29</version>
        </dependency>

        <dependency>
            <groupId>commons-dbcp</groupId>
            <artifactId>commons-dbcp</artifactId>
            <version>1.4</version>
        </dependency>
    </dependencies>
```

2、Spring 配置文件

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

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">

<!-- 自动扫描包 -->
<context:component-scan base-package="com.sunland.*"/>


<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigure
r">
<property name="location" value="classpath:jdbc.properties" />
</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
<property name="driverClassName" value="${driver}" />
<property name="url" value="${url}" />
<property name="username" value="${username}" />
<property name="password" value="${password}" />
<!-- 初始化连接大小 -->
<property name="initialSize" value="${initialSize}"></property>
<!-- 连接池最大数量 -->
<property name="maxActive" value="${maxActive}"></property>
<!-- 连接池最大空闲 -->
<property name="maxIdle" value="${maxIdle}"></property>
<!-- 连接池最小空闲 -->
<property name="minIdle" value="${minIdle}"></property>
<!-- 获取连接最大等待时间 -->
<property name="maxWait" value="${maxWait}"></property>
</bean>

<!-- spring 和 MyBatis 完美整合，不需要 mybatis 的配置映射文件 -->
<bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
<property name="dataSource" ref="dataSource" />

<!--<property name="configLocation" value="classpath:/mybatis.xml" />-->
</bean>

<!-- DAO 接口所在包名，Spring 会自动查找其下的类 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
<property name="basePackage" value="com.sunland.dao" />
```

```
<property name="sqlSessionFactoryBeanName"
value="sqlSessionFactory"></property>
</bean>



</beans>
```

Chapter 15. Spring Cache

Chapter 16. 任务调度和异步执行器

16.1. 任务调度概述

1、pom 依赖

```
<dependency>
    <groupId>org.quartz-scheduler</groupId>
    <artifactId>quartz</artifactId>
    <version>1.8.6</version>
</dependency>
```

16.2. Quartz 快速进阶

16.2.1. Quartz 基础结构

1、Job：

1) 接口，仅包含一个方法，通过实现该接口来定义需要执行的任务
   void execute(JobExecutionContext context)
2) JobExecutionContext 类提供了调度上下文的各种信息，Job 运行时的信息保存在该类的 JobDateMap 实例中

2、JobDetail：

1) Quartz 在每次执行 Job 时，都重新创建一个 Job 实例，因此它不接受一个 Job 实例，而是接受一个 Job 实现类(Class 对象)，以便运行时通过 newInstance()的反射调用机制实例化 Job
2) JobDetail 描述 Job 的实现类以及其他相关的静态信息，例如 Job 名称、描述、关联监听器等信息

3、Trigger：

1) 描述触发 Job 执行时间触发规则
2) 主要有 SimpleTrigger 和 CronTrigger 两个子类
3) SimpleTrigger 用于描述仅需触发一次或者以固定间隔周期执行的规则
4) CronTrigger 用于描述更复杂的执行规则，例如，每天早晨 9:00 执行，每周 1、3、5 下午 5:00 执行等

4、Calendar：

1) org.quartz.Calendar 和 java.util.Calendar 不同，它是一些日历特定时间点的集合(可以简单地将 org.quartz.Calendar 看做 java.util.Calendar 的集合---一个 java.util.Calendar 代表一个日历时间点，若无特殊说明，后面的 Calendar 即
2) org.quartz.Calendar)
3) 一个 Trigger 可以和多个 Calendar 关联，以便排除某些时间点
4) Quartz 在 org.quartz.impl.calendar 包下提供了若干个 Calendar 的实现类，包括 AnnualCalendar、MonthlyCalendar、WeeklyCalendar 分别针对每年、每月、每周进行定义

5、Scheduler：

1) 代表一个独立运行的容器
2) Trigger 和 JobDetail 可以注册到 Scheduler 中，二者在 Schedule 中拥有各自的组及名称，Trigger 和 JobDetail 在各自的集合中组名和名称的组合必

须唯一，但是 Trigger 和 JobDetail 可以同组名和名称，因为它们处于不同的集合之中)

3) Schedule 定义了多个接口和方法，允许外部通过组以及名称访问和控制容器中的 Trigger 和 JobDetail
4) Schedule 可以将 Trigger 绑定到某一个 JobDetail 中，当 Trigger 被触发时，对应 Job 就能执行
5) 一个 Job 可以对应多个 Trigger，但一个 Trigger 只能对应一个 Job
6) 可以通过 SchedulerFactory 创建一个 Scheduler 实例
7) Scheduler 拥有一个 SchedulerContext，保存着 Scheduler 上下文信息，可以对照 ServletContext 来理解 SchedulerContext。Job 和 Trigger 都可以访问 SchedulerContext 内的信息。SchedulerContext 内部通过一个 Map，以键值对的方式维护这些上下文信息。可以通过 Scheduler#getContext()方法获取对应的 SchedulerContext 实例

6、ThreadPool：

1) Scheduler 使用一个线程池作为任务运行的基础设施，任务通过共享线程池中的线程来提高运行效率

7、JobDataMap

1) JobDetail 含有 JobDataMap
2) Trigger 含有 JobDataMap，通过 JobExecutionContext#getTrigger().getJobDataMap()
3) 通过 JobExecutionContext.getMergedJobDataMap()获取合并后的 JobDataMap
4) JobDataMap 用于在 job 的多次执行中，跟踪 job 的状态

8、Job 有一个 StatefulJob 子接口，代表有状态的任务。该接口是一个没有方法的标签接口，其目的是让 Quartz 知道任务的类型，以便采用不同的执行方案

1) 无状态任务在执行时拥有自己的 JobDataMap 复制，对 JobDataMap 的更改不会影响到下次的执行，无状态 Job 可以并发执行
2) 有状态任务共享同一个 JobDataMap，每次任务执行对 JobDataMap 所做的更改会保存下来，后面的执行可以看到这个更改。有状态的 StatefulJob 不能并发执行，这意味着如果前次 StatefulJob 没有执行完毕，下次的任务将被阻塞

16.2.2. 使用 SimpleTrigger

1、SimpleTrigger 拥有多个重载的构造函数，用于在不同场合下构造出对应的实例

- SimpleTrigger(String name,String group)
- SimpleTrigger(String name,String group,Data startTime)
- SimpleTrigger(String name,String group,Data startTime,Data endTime,int repeatCount,long repeatInterval)
- SimpleTrigger(String name,String group,String jobName,String jobGroup,Data startTime,Data endTime,int repeatCount,long repeatInterval)

16.2.3. 使用 CronTrigger

1、CronTrigger 能够提供比 SimpleTrigger 更具实际意义的调度方案，调度规则基于 Cron 表达式

2、Cron 表达式

位置	时间域名	允许值	允许的特殊字符
1	秒	0-59	,-*/
2	分钟	0-59	,-*/
3	小时	0-23	,-*/
4	日期	1-31	,-*?/L W C
5	月份	1-12	,-*/
6	星期	1-7	,-*?/L C #
7	年(可选)	空值 1970-2099	,-*/

1) *：可用在所有字段中，表示对应时间域的每一个时刻
2) ?：该字符只在日期和星期字段中使用，通常表示无意义的值
3) -：表达一个范围，例如在小时段中使用："10-12"
4) ,：表示一个列表值，例如在星期字段中使用："MON,WED,FRI"
5) /：x/y 表达一个等步长序列，x 为起始值，y 为增量值。*/y 等同于 0/y
6) L：该字符只能在星期和日期字段使用，代表"List"的意思
   - L 在日期字段表示这个月份最后一天，如 1 月 31 日，如年的 2 月 28 日
   - L 在星期字段表示星期六，其值为 7。6L 代表最后一个星期 5(值为 6)
7) W：只能出现在日期字段里，表示离该日期最近的工作日，15W 表示离该月 15 日最近的工作日，若 15 是星期六，那么匹配 14 日星期 5，若 15 是星期日，则匹配 16 日星期一
8) LW：在日期字段里可以使用 LW，它的意思是当月最后一个工作日
9) #：只能在星期字段中使用，表示当月的某个工作日，6#3 表示当月第三个星期 5
10) C：该字符只能在星期和日期字段中使用，代表 Calendar 的意思。5C 在日期字段中相当于 5 日后的那一天，1C 在星期字段中相当于星期日后的第一天

16.2.4. Quartz2.0

1、在 2.x 版本中，Quartz 废弃了很多类的构造方法，而改为采用这些类的建造者类(Builder)来初始化它们

2、目前版本中，简单地，Scheduler 的生成有两种方法，二者区别不大：

```
Scheduler s = new StdSchedulerFactory().getScheduler();
Scheduler s = StdSchedulerFactory.getDefaultScheduler();
```

- Scheduler 对象可以通过许多方法来加载 JobDetail 和 Trigger 对象，scheduleJob 方法是其中之一，此外还有 addJob 等
- 需要注意的是，Scheduler 中允许不存在 JobDetail 和 Trigger，在此情况下调度器将空跑(即死循环)
- 一旦加载了 JobDetail，则至少必须加载一个 Trigger，反之亦然。

3、JobDetail 通过 JobBuilder 生成

- 而 JobBuilder 通过其静态方法 newJob 创建一个 JobBuilder 的实例

- 对于 JobBuilder 对象而言，ofType 是必须调用的方法，通过它来加载实现了 Job 接口的类
- withIndentify 方法将会生成一个 JobKey 的私有成员，它是 JobDetail 对象的唯一标志

```
JobDetail jobDetail=JobBuilder.newJob(SimpleJob.class)
                                .withIdentity("job_id","job_group_id")
                                .build();
```

4、Trigger 通过 TriggerBuilder 生成

- TriggerBuilder 通过其静态方法 newTrigger 创建一个 TriggerBuilder 的实例。与 JobDetail 相同
- TriggerBuilder 也通过 withIndetify 方法为 Trigger 对象创建唯一标志
- 此外，TriggerBuilder 中还有 startNow，startAt，endAt 等方法，来设置任务触发及结束的时间
- 在 1.x 版本的 Quartz 中，触发器还分为 SimpleTrigger、CronTrigger 等，但是在 2.x 版本中，这些具体的 Trigger 类都被废弃了，取而代之的是 TriggerBuilder 中的 withSchedule 方法。该方法需要传入一个 SechduleBuilder 对象，通过该对象来实现触发器的逻辑
- ScheduleBuilder 有三种，分为是

1) SimpleScheduleBuilder：SimpleScheduleBuilder 是简单调用触发器，它只能指定触发的间隔时间和执行次数
2) CronScheduleBuilder：类似于 Linux Cron 的触发器，它通过一个称为 CronExpression 的规则来指定触发规则，通常是每次触发的具体时间
3) CalendarIntervalScheduleBuilder： CalendarIntervalScheduleBuilder 是对 CronScheduleBuilder 的补充，它能指定每隔一段时间触发一次

- withSchedule 最多可能被同一个 TriggerBuilder 对象调用一次

```
Trigger trigger = TriggerBuilder
                .newTrigger()
                withIdentity("trigger1", "group1")
                .startNow()
                .withSchedule(SimpleScheduleBuilder.
                                repeatSecondlyForTotalCount(10, 2))
                .withSchedule(CronScheduleBuilder.
                                cronSchedule("0 30 9 * * ?"))
                .withSchedule(CalendarIntervalScheduleBuilder.
                                calendarIntervalSchedule().withIntervalInHours(2))
                .build();
```

16.3. 在 Spring 中使用 Quartz

1、Spring 提供了两方面的支持

1) 为 Quartz 的重要组件提供了更具 Bean 风格的扩展类
2) 提供创建 Scheduler 的 BeanFactory 类，方便在 Spring 环境下创建对应的组件对象，并结合 Spring 容器生命周期执行启动和停止的动作

16.3.1. 创建 JobDetail

1、可以直接使用 Quartz 的 JobDetail 在 Spring 中配置一个 JobDetail Bean，但是 JobDetail 使用带参数的构造函数，较为不便

2、Spring 通过扩展 JobDetail 提供了一个更具 Bean 风格的 JobDetailFactoryBean

3、此外 Spring 提供了一个 MethodInvokingJobDetailFactoryBean，通过这个 FactoryBean 可以将 Spring 容器中的 Bean 的方法包装成一个 Quartz 任务

16.3.1.1. JobDetailFactoryBean

1、JobDetailFactoryBean 扩展于 Quartz 的 JobDetail，支持以下属性

- jobClass：类型为 Class，实现 Job 接口的任务类
- beanName：默认为 Bean 的 id 名，通过该属性显式指定 Bean 名称，它对应任务的名称
- jobDataAsMap：类型为 Map，为任务所对应的 JobDataMap 提供值。之所以需要提供这个属性，是因为用户无法在 Spring 配置文件中为 JobDataMap 类型的属性提供信息，所以 Spring 通过 jobDataAsMap 设置 JobDataMap 的值
- applicationContextJobDataKey：用户将 Spring ApplicationContext 的引用保存到 JobDataMap 中，以便在 Job 的代码中访问 ApplicationContext。为了达到这个目的，用户需要指定一个键，用于在 jobDataAsMap 中保存 ApplicationContext。如果不设置，就不会将 ApplicationContext 放入 jobDatamap 中
- jobListenerNames：类型为 Stirng[]，指定注册在 Scheduler 中的 JobListeners 名称，以便让这些监听器对本任务的时间进行监听

2、示例如下

```
<bean id="jobDetail"
class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
    <property name="jobClass"
    value="com.sunlands.job.DownloadJob" />
</bean>
```

16.3.1.2. MethodInvokingJobDetailFactoryBean

1、通常情况下，业务都定义在一个业务类的方法中，为满足 Quartz Job 接口的规定，还需要定义个引用业务类方法的实现类。为了避免创建这个只包含遗憾调用代码的 Job 实现类，Spring 提供了 MethodInvokingJobDetailFactoryBean，借由该 FactoryBean，可以将一个 Bean 的某个方法封装成满足 Quartz 要求的 Job

2、示例如下

```
<bean id="jobDetail"
    class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFa
    ctoryBean"
    p:targetObject-ref="myService"
    p:targetMethod="doJob"
    p:concurrent="false"/>

<bean id="myService" class="com.smart.service.MyService"/>
```

- concurrent：该属性指定任务有无状态，无状态(true)的任务可以并发执行，有状态(false)的任务，不能并发执行

16.3.2. 创建 Trigger

1、Quartz 中另一个重要的组件就是 Trigger，Spring 按照相似的思路分别为 SimpleTrigger 和 CronTrigger 提供了更具 Bean 风格的 SimpleTriggerFactoryBean 和 CronTriggerFactoryBean 扩展类，通过这两个扩展类可以更容易地在 Spring 中以 Bean 的方式配置 Trigger

16.3.2.1. SimpleTriggerFactoryBean

1、在默认情况下，通过 SimpleTriggerFactoryBean 配置的 Trigger 名称即为 Bean 的名称，属于默认组。SimpleTriggerFactoryBean 在 SimpleTrigger 的基础上新增了以下属性

1) jobDetail：对应的 JobDetail
2) beanName：默认为 Bean 的 id 名，通过该属性显式指定 Bean 名称，它对应 Trigger 的名称
3) jobDataAsMap：以 Map 类型为 Trigger 关联的 JobDataMap 提供值
4) startDelay：延迟多少时间开始触发，单位为毫秒，默认值为 0
5) triggerListenerNames：类型为 String[]，指定注册在 Scheduler 中的 TriggerListener 名称，以便让这些监听器对本触发器的时间进行监听

2、示例

```
<bean id="simpleTrigger"
      class="org.springframework.scheduling.quartz.SimpleTriggerFactoryBean"
      p:jobDetail-ref="jobDetail"
      p:startDelay="1000"
      p:repeatInterval="2000"
      p:repeatCount="100"
      <property name="jobDataAsMap">
          <map>
              <entry key="count" value="10"/>
          </map>
      </property>
</bean>
```

16.3.2.2. ConTriggerFactoryBean

1、CronTriggerFactoryBean 扩展于 CronTrigger，触发器的名称即为 Bean 的名称，保存在默认组中。在 CronTrigger 的基础上，新增属性和 SimpleTriggerFactoryBean 大致相同，配置方式也相似

2、示例

```
<bean id="checkImagesTrigger"
      class="org.springframework.scheduling.quartz.CronTriggerFactoryBean"
      p:jobDetail-ref="jobDetail"
      p:cronExpression="0/5 * * * * ?"/>
```

16.3.3. 创建 Scheduler

1、Quartz 的 SchedulerFactory 是标准的工厂，不太适合在 Spring 环境下使用。此外为了保证 Scheduler 能够感知 Spring 容器的生命周期，在 Spring 容器启动后，Scheduler 自动开始工作，在 Spring 容器关闭前，自动关闭 Scheduler。为此，Spring 提供了 SchedulerFactoryBean，这个 FactoryBean 大致拥有以下功能

1) 以更具 Bean 风格的方式为 Scheduler 提供配置信息
2) 让 Scheduler 和 Spring 容器的生命周期建立关联
3) 通过属性配置的方式代替 Quartz 自身的配置文件

2、示例

```
<bean id="scheduler"
class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
  <property name="triggers">
    <list>
      <ref bean="simpleTrigger">
    </list>
  </property>

  <property name="schedulerContextAsMap">
    <map>
      <entry key="timeout" value="30"/>
    </map>
  </property>

  <property name="configLocation" value="classpath:com/quartz.properties"/>
</bean>
```

3、除了上述示例中的属性外，还支持以下常规属性

1) calendars：类型为 Map，通过该属性向 Scheduler 注册 Calendar
2) jobDetails：类型为 JobDetail[]，通过该属性向 Scheduler 注册 JobDetail
3) autoStartup：SchedulerFactoryBean 在初始化后是否马上启动 Scheduler，默认为 true，如果设为 false，则需要手动启动 Scheduler
4) startupDelay：在 SchedulerFactoryBean 初始化完毕后，延迟多少秒启动 Scheduler，默认值为 0，可以通过该值让 Scheduler 延迟一小段时间启动，以便让 Spring 能够更快初始化容器中剩余 Bean

16.3.4. 对 Job 示例利用 Spring 注解注入 Service

1、由于 Job 对象在每次执行 Job 时都会通过 JobDetail 通过工厂产生一个新的实例，因此普通的@Autowired 注解无法正常工作

2、需要按如下步骤

1) 定义 Job 工厂类

```
@Component("downloadJobFactory")//配置工厂 bean，作为 scheduler 的工厂
public class DownloadJobFactory extends AdaptableJobFactory {
    @Autowired//获取工厂类的实例
    private AutowireCapableBeanFactory capableBeanFactory;
```

```
        @Override
        protected Object createJobInstance(TriggerFiredBundle bundle) throws
        Exception {
        //调用父类的方法
        Object jobInstance = super.createJobInstance(bundle);
        //进行注入
        capableBeanFactory.autowireBean(jobInstance);
        return jobInstance;
        }
    }
```

2) 配置 schedule bean 时配置 factoryBean 属性

```
    <bean                                   id="scheduler"
    class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
        <property name="triggers">
            <list>
                <ref bean="downloadJobTrigger"/>
            </list>
        </property>
        <property name="jobDetails">
            <list>
                <ref bean="downloadJobDetail" />
            </list>
        </property>
        <property name="jobFactory" ref="downloadJobFactory"/>
    </bean>
```

Chapter 17. Spring MVC

Chapter 18.实战案例开发

Chapter 19. Spring OXM

Chapter 20. 实战型单元测试

1. 注解

1.1.@Component：用于标注 POJO

1、@Repository：用于对 DAO 实现类进行标注
2、@Service：用于对 Service 实现类进行标注
3、@Controller：用于对 Controller 实现类进行标注
4、可以指定 bean 名称

```
@Controller("testBean")
```

1.2.@Autowired：进行自动注入

1、默认按 byType 匹配的方式在容器中查找匹配的 Bean，有且仅有一个才会正确注入
2、@ Autowired(required=false)：即使找不到匹配的 Bean 也不抛出异常
3、使用@Qualifier 指定注入 Bean 的名称

1.3.@RequestMapping：通过请求 URL 进行映射

1、@PathVariable：将 URL 中的占位符参数绑定到控制器处理方法的入参中
2、@RequestParam：绑定请求参数值，可以有以下三个参数

- value：参数名
- required：是否必须，默认 true，表示请求中必须包含对应的参数名，如果不存在，则抛出异常
- defaultValue：默认参数名，自动将 required 设为 false，不推荐使用

3、@CookieValue：绑定请求中的 Cookie 值，同@RequestParam 有三个属性
4、@RequestHeader：绑定请求报文头的属性值，同@RequestParam 有三个属性

1.4.@RequestBody/@ReponseBody

1、使用 HttpMessageConverter<T>将请求信息转换并绑定到处理方法的入参中
2、@RequestBody 标注入参
3、@ReponseBody 标注方法

1.5.@ModelAttribute：

1、模型数据会赋值给该标注了@ModelAttribute 注解的入参
2、在方法定义中使用@ModelAttribute 注解，Spring MVC 在调用目标处理方法前，会先逐个调用在方法级上标注了@ModelAttribute 注解的方法，并将这些方法的返回值添加到模型中
3、处理方法入参最多只能使用一个 Spring MVC 注解，如果使用了@ModelAttribute 注解，就不能再使用@RequestParam 或@CookieValue 等注解了

1.6.@SessionAttributes

1、希望在多个请求之间共用某个模型数据，则可以在控制器类中标注一个@SessionAttributes，Spring MVC 会将模型中对应的属性暂存到 HttpSession 中
2、@ModelAttribute 以及@SessionAttributes 详细处理流程见 P590
3、@SessionAttributes 除了可以通过属性名指定需要放到会话中的属性外，还

可以通过模型属性的对象类型指定哪些模型属性需要放到会话中

4、此外，@SessionAttributes 还可以通过属性名以及 types 一起指定，二者都允许多值，放到会话中的属性是二者的并集

- @SessionAttributes(value={"user1","user2"})：将名为 user1 以及 user2 的模型属性添加到会话中
- @SessionAttributes(types={User.class,Dept.class})：将模型中所有类型为 User 以及 Dept 的属性添加到会话中
- @SessionAttributes(value={"user1","user2"}, types={User.class,Dept.class})：将名为 user1 以及 user2 的模型属性添加到会话中，同时将模型中所有类型为 User 以及 Dept 的属性添加到会话中