1.ComponentScan

```
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
      @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
```

过滤不需要扫描的类

2.SpringBootApplication

能够自定义扫描那些包。但是提供的exclude和excludeName方法时对于其内部i的自动配置类才会生效。为了能够派出其他的类，还可以在加入@ComponentScan以达到目的

```
@SpringBootApplication
@ComponentScan(basePackages = {"com.springboot.chapter3"}, excludeFilters =
{@ComponentScan.Filter(UserService.class)})
public class Chapter3Application {

    public static void main(String[] args) {
        SpringApplication.run(Chapter3Application.class, args);
    }
}
```

3.依赖注入

```
@Autowired
private Animal animal = null;
```

Animal是一个接口，当有多个Animal接口实现的时候，注入的时候会报错。两种修复方式：

- **规则一：首先更具类型找到对应的Bean，如果Bean不唯一，那么会更具属性名称和Bean名称进行匹配。如果匹配的上就会使用该bean，否则抛出异常**

  ```
  @Autowired
  private Animal dog = null;
  ```

  @Autowired是一个默认必须找到对应Bean的注解，如果不能确定其标注属性一定存在并且允许这个被标注的属性为null，则可以配置@Autowired属性required为false。@Autowired(required = false)

- **规则二：消除歧义性 @Primary @Quelifier**

  规则一本来是一个animal的属性，强制改成了dog比较怪异。

  - **@Primary，修改优先权的注解**，假若有猫有狗，而需要猫的时候则可以

```
@Component
@Primary
public class Cat implements Animal {

   @Override
   public void use() {
      System.out.println("猫【" + Cat.class.getSimpleName()+"】是抓老
鼠。");
   }
}
```

- **@Quelifier，与@Autowired组合在一起，通过类型和名称一起找到Bean**

```
@Autowired
@Qualifier("dog")
public void setAnimal(Animal animal) {
    this.animal = animal;
}
```

4. 生命周期

Spring IoC初始化和销毁Bean的过程，便是Bean的生命周期的过程，大致分为**Bean定义、Bean初始化、Bean的生存周期、Bean的销毁4分部分**

- **Spring通过配置，如@ComponentScan定义的扫描路径找到带有@Component的类。**这个过程就是资源定位的过程
- **解析资源，将定义的信息保存起来。**此时没有初始化Bean，仅仅有的是Bean的定义
- **Bean的定义发送到Spring IoC容器。**此时IoC容器中也只有Bean的定义

默认情况，Spring会继续完成Bean的实例化和依赖注入，这样从IoC容器中就可以得到一个依赖注入完成的Bean。但有些Bean会受到变化因素的影响，这时候我们希望取出Bean的时候完成初始化和依赖注入。

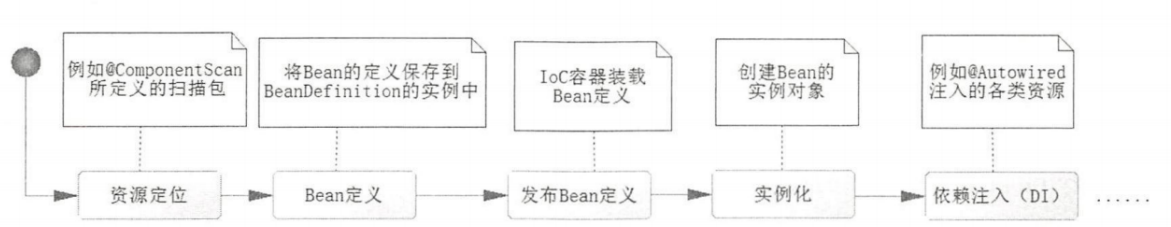

图 3-3　Spring 初始化 Bean

@ComponentScan还有一个配置像lazyInit，默认为false，也就是默认不延迟初始化。因此默认情况下Spring会对Bean进行实例化和依赖注入对应的属性值。

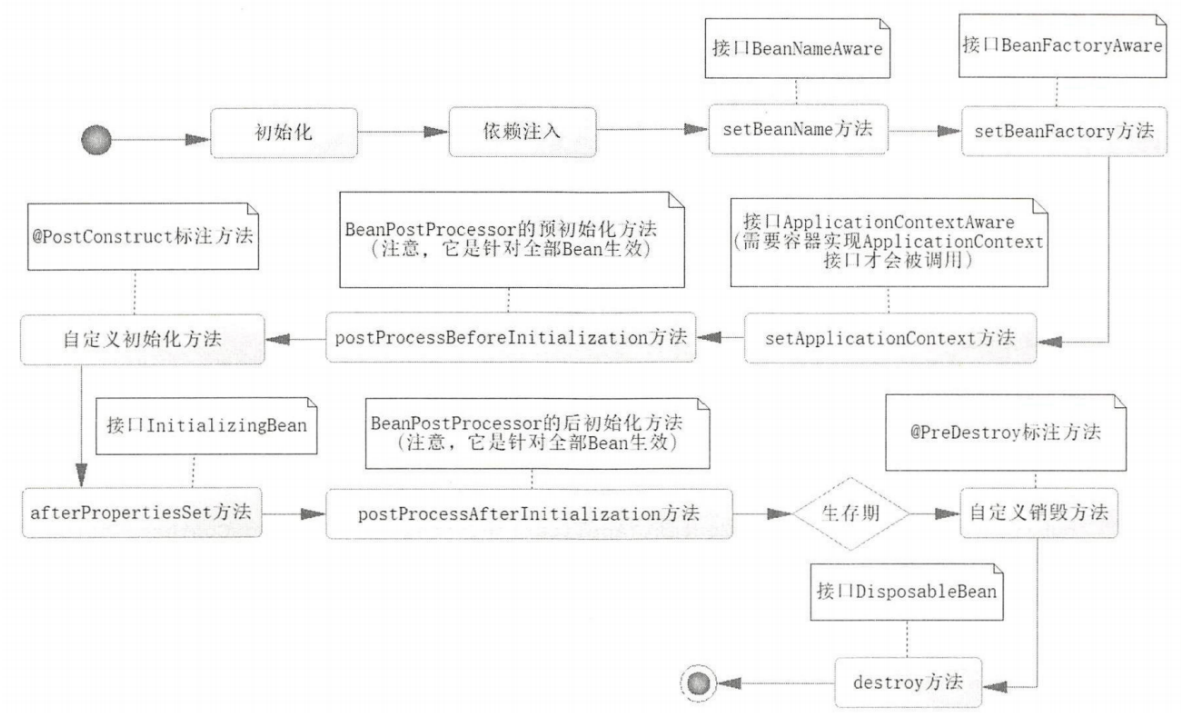

图 3-5　Spring Bean 的生命周期

Spring IoC容器最低的要求是实现BeanFactory接口，而不是实现ApplicationContext接口。只有实现了ApplicationContext接口的容器，才会在生命周期调用ApplicationContextAward所定义的setApplicationContext方法

```
@Component
public class BussinessPerson implements Person, BeanNameAware, BeanFactoryAware,
ApplicationContextAware, InitializingBean, DisposableBean {

    private Animal animal = null;

    @Override
    public void service() {
        this.animal.use();
    }

    @Override
    @Autowired
    @Qualifier("dog")
    public void setAnimal(Animal animal) {
        System.out.println("延迟依赖注入");
        this.animal = animal;
    }

    @Override
    public void setBeanName(String beanName) {
        System.out.println("【" + this.getClass().getSimpleName()
                + "】调用BeanNameAware的setBeanName");
    }

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("【" + this.getClass().getSimpleName()
                + "】调用BeanFactoryAware的setBeanFactory");
    }
```

```
    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
throws BeansException {
        System.out.println("【" + this.getClass().getSimpleName()
                + "】调用ApplicationContextAware的setApplicationContext");

    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("【" + this.getClass().getSimpleName()
                + "】调用InitializingBean的afterPropertiesSet方法");
    }

    @PostConstruct
    public void init() {
        System.out.println("【" + this.getClass().getSimpleName()
                + "】注解@PostConstruct定义的自定义初始化方法");
    }

    @PreDestroy
    public void destroy1() {
        System.out.println("【" + this.getClass().getSimpleName()
                + "】注解@PreDestroy定义的自定义销毁方法");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("【" + this.getClass().getSimpleName()
                + "】 DisposableBean方法");

    }

}
```

这个Bean实现了生命周期中单个Bean可以实现的所有接口，并且通过注解@PostConstruct定义了初始化方法，通过@PreDestroy定义了销毁方法

```
@Component
public class BeanPostProcessorExample implements BeanPostProcessor {

   @Override
   public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
      System.out.println("BeanPostProcessor调用postProcessBeforeInitialization方
法，参数【"
              + bean.getClass().getSimpleName()+ "】【" +beanName+"】 ");
      return bean;
   }

   @Override
   public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
      System.out.println("BeanPostProcessor调用postProcessAfterInitialization方
法，参数【"
          + bean.getClass().getSimpleName()+ "】【" +beanName+"】 ");
      return bean;
```

```
    }
}
```

这个后置处理器会对所有的Bean有效。而其他的接口则是对于单个Bean起作用。

Bean的定义可能使用的是第三方的类，则可以使用@Bean类配置自定义初始化和销毁方法

```
@Bean(name = "dataSource", destroyMethod = "close", initMethod = "init")
```

5. 使用属性文件

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-configuration-processor</artifactId>
   <optional>true</optional>
</dependency>
```

有了依赖，就可以直接使用application.properties配置文件

5.1 配置属性

```
database.driverName=com.mysql.jdbc.Driver
database.url=jdbc:mysql://localhost:3306/chapter3
database.username=root
database.password=123456
```

5.1.1 使用Spring表达式

```
    @Bean(name = "dataSource", destroyMethod = "close")
// @Conditional(DatabaseConditional.class)
   public DataSource getDataSource(
         @Value("${database.driverName}") String driver,
         @Value("${database.url}") String url,
         @Value("${database.username}") String username,
         @Value("${database.password}") String password
         ) {
      Properties props = new Properties();
      props.setProperty("driver", driver);
      props.setProperty("url", url);
      props.setProperty("username", username);
      props.setProperty("password", password);
      DataSource dataSource = null;
      try {
         dataSource = BasicDataSourceFactory.createDataSource(props);
      } catch (Exception e) {
         e.printStackTrace();
      }
      return dataSource;
   }
```

通过@Value注解，使用${...}占位符读取配置在属性文件中的内容

```
@Component
public class DataBaseProperties {
   @Value("${database.driverName}")
   private String driverName = null;
}
```

5.1.2 使用注解@ConfigurationProperties，

```
@Component
@ConfigurationProperties( "database")
public class DataBaseProperties {
   private String driverName = null;

   private String url = null;

   private String username = null;

   private String password = null;
    **set方法省略**
}
```

ConfigurationProperties中配置的字符串database，将与POJO的属性名成组成属性的全限定去配置文件中查找，然后将对应的属性读入到POJO

如果把所有的内容都配置到application.properties，可能会导致配置过多。为了更好的配置，可以使用新的属性文件。例如数据库的属性可以配置在jdbc.properties。然后使用@PropertySource去定义对应的属性文件。把它加载到Spring的上下文中

```
@SpringBootApplication
@ComponentScan(basePackages = {"com.springboot.chapter3"})
@PropertySource(value={"classpath:jdbc.properties"})
@ImportResource(value = {"classpath:spring-other.xml"})
public class Chapter3Application {

   public static void main(String[] args) {
      SpringApplication.run(Chapter3Application.class, args);
   }
}
```

value可以配置多个配置文件，使用classpath前缀，意味着去类文件路径下找到文件，属性ignoreResourceNotFound则是是否忽略配置文件找不到的问题。默认值是false，找不到文件就会报错。

6. 条件装配Bean

```
public class DatabaseConditional implements Condition {

   @Override
   public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
      Environment env = context.getEnvironment();
      return env.containsProperty("database.driverName") &&
env.containsProperty("database.url")
            && env.containsProperty("database.username") &&
env.containsProperty("database.password");
   }
}
```

```
@Bean(name = "dataSource", destroyMethod = "close")
@Conditional(DatabaseConditional.class)
public DataSource getDataSource(
      @Value("${database.driverName}") String driver,
      @Value("${database.url}") String url,
      @Value("${database.username}") String username,
      @Value("${database.password}") String password
      ) {
   Properties props = new Properties();
   props.setProperty("driver", driver);
   props.setProperty("url", url);
   props.setProperty("username", username);
   props.setProperty("password", password);
   DataSource dataSource = null;
   try {
      dataSource = BasicDataSourceFactory.createDataSource(props);
   } catch (Exception e) {
      e.printStackTrace();
   }
   return dataSource;
}
```

matches 方法首先读取其上下文环境 然后 定是否已经配置了对应的数据库信息。这样，当这些都已经配置好后则返回 true 。这个时候 Spring 会装配数据库连接池的 Bean ， 否则是不装配的。

其他内容参考 Spring ->2. 高级装配