- **反射**
- **注解**
- **Module**
- **内部类**
- **接口default方法**
- **函数式接口**
- **Lambda表达式**
- **流式编程**
- **继承相关**
  - 构造函数
- **SpringIOC容器**
  - 容器接口及实现类
  - IOC容器特征
  - IOC容器流程分析
    - AnnotationConfigApplicationContext分析
      - 接口
      - 构造方法
      - 核心Refresh()
- **AOP**
  - 代理模式
    - 静态代理
    - 动态代理
      - JDK动态代理
      - CGLIB代理
  - SpringAOP
    - 流程分析
  - AspectJ
  - SpringAOP&AspectJ区别
- **SpringBoot**
  - 最佳实践
    - 代码结构
    - Configuration Class
    - 自动配置
    - Bean和依赖注入
    - 使用@SpringBootApplication注解
  - SpringBoot Features
      - SpringApplication
        - 启动
        - Lazy初始化
        - 监测
        - Application Events and Listeners
        - ApplicationContext
        - AccessingApplicationArguments
        - ApplicationRunner or CommandLineRunner
    - Externalized Configuration
      - Type-safe Configuration Properties
        - JavaBean properties binding
        - Constructor binding
        - Enabling @ConfigurationProperties
        - Third-party Configuration
        - Relaxed Binding 宽松绑定
        - Map绑定
        - 环境变量绑定
        - 复杂类型合并
        - 属性转换

> 本篇内容为个人笔记，仅供允许学习交流，暂不允许用于任何商业用途
>
> 转载注明出处：h4cknight/spring-analysis: Spring系列源码分析

反射

> 反射机制允许程序借助ReflectionAPI获取并操作类内部信息

- Class.forName，类名.class，obj.getClass

- 注解通过反射判断时，会视为接口，都继承自AnnotatedElement
- enum通过反射判断时，会视为类，都继承自Enum
    - 引申一下，enum是创建单例模式最好的方式
        - 不用处理并发（类加载是唯一的）
        - 不用考虑序列化与反序列化问题（序列化时只序列化name，反序列化时通过name查找枚举值）

注解

- 注解一般通过public @interface 注解名 {定义体} 进行声明
- 定义了注解后，要想注解发挥作用，需要编写相应的注解处理器
    - 编译时注解处理器
        - 一般将代码或字节码和注解做为输入，生成新的文件
    - 运行时注解处理器
        - 通过反射获取注解信息，然后进行其它操作
- AbstractProcessor

> An abstract annotation processor designed to be a convenient superclass for most concrete annotation processors.

- 一个注解可以标注其它注解(意味着@Target(ElementType.TYPE/ANNOTATION_TYPE))，那么被标注的注解称为复合/组合注解(Composed Annotation)

> java中注解不允许使用extends和implements

如@Zero标注在@One上，那么获取注解的注解的方式：One.class.getAnnotation(Zero.class)

Spring4.2之后提供了组合注解的实现方式，就是将多个注解作用于一个注解，用一个注解实现多个注解的功能；另外组合注解可以实现低层次注解属性方法覆盖高层次属性注解属性,实现了类似“继承”关系，但这些都需要依赖于Spring提供的API

API

```
//*AnnotatedElementUtils* defines the public API for Spring's meta-annotation programming model with support for annotation attribute overrides. If you do not need support for annotation attribute overrides, consider using *AnnotationUtils* instead.

@Target({ElementType.TYPE,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface SameAnnotation {
    //同一个注解内的别名，title和name是同一个属性
    @AliasFor("title")
    String name() default "";
    @AliasFor("name")
    String title() default "";
}
@Target({ElementType.TYPE,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface MyAnnotation/2 {
    String name() default "";
}
```

```
@Target({ElementType.TYPE,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@MyAnnotation
public @interface MySubAnnotation {
    @AliasFor(annotation=MyAnnotation.class, attribute="name")//不同注解间的别名
    String name1() default "";
}
@Target({ElementType.TYPE,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@MyAnnotation2(name = "====")
public @interface MySubAnnotation2 {
    String name1() default "";
}
@SameAnnotation(name = "1234")
@MySubAnnotation(name1 = "5678")
public class Java8Test {...}
SameAnnotation annotation = AnnotationUtils.getAnnotation(Java8Test.class,
SameAnnotation.class);
System.out.println(annotation.name());//1234
System.out.println(annotation.title());//1234
MyAnnotation annotation2 =
AnnotatedElementUtils.getMergedAnnotation(Java8Test.class,
MyAnnotation.class);
System.out.println(annotation2.name());//5678
MyAnnotation2 annotation3 =
AnnotatedElementUtils.getMergedAnnotation(Java8Test.class,
MyAnnotation2.class);
System.out.println(annotation3.name());//====，虽然没有定义MyAnnotation2，但通过MySubAnnotation2获取到了MyAnnotation2相关信息
```

Module

> 隔离访问、轻量化、安全

- 模块类型
  - 系统模块，来自JDK和JRE，通过java --list-modules列出
  - 应用程序模块，来自日常开发定义
  - 自动模块，现有的jar文件，不是模块，但是将他们添加到模块路径时，会自动创建具有jar名称的模块
    - 隐式exports所有包类型
    - 隐式requires所有其它的named模块，特别的，还会包含未命名模块
      - 会认为其它模块都可读，因此可以访问所有其它可读的导出类型；
    - 主要场景是自己的包都升级成了module，而别人的包没有升级module，这时就可以将别人的包放在模块路径，变成自动模块；放在类路径不行，因为会被视为unamed module，而这种模块无法被requires
    - 命名：.jar被删除，版本号被删，非字母数字字符替换为.，重复.被替换为单个.，起始/终止.被删除
    - 隐式将META-INF/services resource SPI 作为module 的provides语句
    - 隐式use所有可用的 module service

    - 未命名模块，添加到类路径的jar和类都会被添加到未命名模块
        - 会认为所有其它模块都可读，因此可以访问所有其它可读的导出类型，包括自动模块
        - 会导出自身的所有包，但是在其它named module中并不能访问unamed module，因为其它模块没办法显示的依赖unmaed module，是故意这样设计的
        - 如果同时定义在named module和uname module，后者会被忽略
        - 主要用于将java8以前项目运行在java9，而不用做什么变动
- 模块声明描述文件module-info.java，申明模块
    - 一个项目可以由多个模块组成
    - 包名称唯一，即使在不同的模块，也不同有相同包名
    - 模块名称建议域名反写，且不允许冲突
    - 兼容性
        - 模块jar和普通的jar没有什么区别，除了在根目录有一个module-info.class
            - 模块jar如果在普通的jar环境运行，module-info.class会被忽略
    - java9开始平台包被拆分成多个module，其中base module总是可用，其它模块隐式依赖于它
- 导出包
    - exports
        - exports pkg;
        - exports pkg to module1,module2,...;定向导出
    - 默认情况下，模块下的所有包都是私有的，即使被外部依赖，也无法访问，一个模块内的包互相访不受影响； 使用exports可以公开特定的包
    - 不能导出具体的类
- 依赖，实际表示的是一个可读性，需要同时满足exports一起表示出可访问性

    > 如果一个模块要访问其他模块导出的包，必须用requires关键字导入要访问包所在的模块

    - requires module;
    - requires static module;
        - 尽在编译期间必须有，运行期间可选
    - requires transitive module;
        - a依赖b，b依赖c，如果a要使用c公开的包，那么需要requires模块c; 借助于transitive关键字，只要b中 requires transitive module c，那么a依赖b就自动会requires 模块c，不再需要手动requires 模块c
- 反射
    - exports，静态编译和运行时都会起作用，结果是仅"public"可用，其它不可用；
    - open，仅运行时起作用，默认仅"public"可用，但可以通过setAccessible设置"private"可用
        - 由于静态编译不起作用，所以不能直接使用类型声明来定义对象，会报无法找到对应的类；因此，open也表达了仅反射访问的含义，主要方便于framework的实现
        - 这种可以访问"private"的行为也称为deep reflection
- 服务
    - uses java.sql.Driver; 写在定义接口的module
    - provides java.sql.Driver with com.mysql.jdbc.Driver;
    - 使用了users和provides后可以不使用以前META-INF/services/目录下写一个接口配置文件
    - 代码部分的使用和普通SPI一致
- ModuleLayer/ClassLoader/Configuration参考

内部类

成员内部类/局部内部类/静态内部类产生的效果可以按照对应不同位置的代码或变量进行思考，比如成员内部类对比成员变量，那么必然和外部实例对象有关，必须先实例化外部实例对象，才能再new出成员内部类对象；局部/静态类似考虑

匿名内部类是不能有名字的类，只能在new语句时定义它们。其类声明是在编译时进行的，实例化在运行时进行，这意味着for循环中的一个new语句会创建相同匿名类的几个不同实例，而不是创建几个不同匿名类的单实例。

> new <类或接口> <主体>
>
> 类可以被重写方法；接口可以有多个未实现方法，在主体中需要进行实现，并且主体允许定义成员变量；
>
> 闭包的原理是局部变量通过构造函数传到匿名内部类内部的final变量，从而被记住了；
>
> Lambda 表达式和匿名内部类访问的局部变量必须是 final/effectively final的，它们捕获这些局部变量构成闭包（闭包就是将使用的局部变量包在Lambda表达式和内部类中，关联了强引用，使得这些变量的生命周期延长并可随后被表达式或内部类使用）
>
> 一个非`final`的局部变量或方法参数，其值在初始化后就从未更改，那么该变量就是`effectively final`
>
> final 修饰的变量即成为常量，只能赋值一次，但是 final 所修饰局部变量和成员变量有所不同。
>
> 1. final 修饰的局部变量必须使用之前被赋值一次才能使用。
> 2. final 修饰的成员变量在声明时没有赋值的叫“空白 final 变量”。空白 final 变量必须在构造方法或静态代码块中初始化。
>
> 参考：Java8--匿名类和函数式接口 - 知乎 (zhihu.com)

- 编译后产生的.class文件个数：有多少个类，产生多少个.class文件
- 普通内部类命名：外部类名 + $ + 内部类名 [+ $ + 内部类名 + ...]+ .class
- 匿名内部类命名：外部类名 + $ + 数字 [+ $ + 数字 + ...]+ .class -数字根据在外部类中定义的顺序决定

> 上面命名规则，不完全正确，见过$1Inner等情况，只能说一般可以这样简单理解

接口default方法

- JAVA8开始可以在接口内部提供一些default方法，该方法有方法体，实现类可以重写也可以不重写
- 多个接口有同名的default方法，此时他们的实现类必须重写该default方法来处理编译器报错
- 父类与interface中的default方法有相同的方法签名时，子类继承的是父类的方法，而default会被忽略
- defautl方法中可以使用this，以及一些其它可见的方法；但是实现类中定义的就无法访问，因为不可见

> 引入原因：
>
> 想象一下，你java7的代码写了一个MyIterable实现了java.lang.Iterable接口（此时还没forEach方法），但是当你将MyIterable移植到java8后，**如果forEach不是default的，你的代码会报错，必须重写forEach方法**，有了default以后，能保持向前的兼容

```
public interface Iterable<T> {//java.lang.Iterable
```

```
    Iterator<T> iterator();

    /**
     * @since 1.8
     */
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }

    /**
     * Creates a {@link Spliterator} over the elements described by this
     * {@code Iterable}.
     * @since 1.8
     */
    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
}
```

函数式接口

Definition : a functional interface has exactly one abstract method.

> Since default methods have an implementation, they are not abstract.
>
> If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere.
>
> 可以定义静态方法，默认方法，不视为抽象方法
>
> 接口中可以定义重写Object的方法(没有方法体需要在子类实现)String toString() ;不视为抽象方法

@FunctionalInterface

> If a type is annotated with this annotation type, compilers are required to generate an error message unless:
> The type is an interface type and not an annotation type, enum, or class.
> The annotated type satisfies the requirements of a functional interface.
>
> the compiler will treat any interface meeting the definition of a functional interface as a functional interface regardless of whether or not a FunctionalInterface annotation is present on the interface declaration.
>
> 该注解强制编译器对接口进行检查是否满足函数式接口要求，如果不满足会报错误消息；
>
> 如果一个接口满足函数式接口要求，不用该注解也可以作为函数式接口被使用。

> 函数式接口最大的作用其实是为了**支持行为参数传递**，比如**传递Lambda、方法引用、函数式接口对应的实例对象**，有点像类型定义
>
> `FunctionalInterface`定义的变量允许传入：
>
> - 接口的实现类（传统写法，代码较繁琐）；
> - Lambda表达式（只需列出参数名，由编译器推断类型）；
> - 符合方法签名的静态方法；
>   - Class::Static Method
> - 符合方法签名的实例方法（实例方法所属的实例类型对应接口第一个参数，实例方法的参数部分按顺序对应接口的其它剩余参数）；
>   - e.g. String::compareTo //s1.compareTo(s2); 接口第一个参数对应s1,接口第二个参数对应s2
> - 符合方法签名的构造方法（实例类型被看做返回类型）
>   - e.g. 一般对象String::new，数组对象String[]::new
>
> 目标引用::方法 即方法引用，如果Lambda所要实现的方案，已经有其他方法存在相同方案，那么则可以使用方法引用进行替代。

Lambda表达式

Lambda表达式的结构：

> - Lambda 表达式可以具有零个，一个或多个参数。
> - 可以显式声明参数的类型，也可以由编译器自动从上下文推断参数的类型。例如 `(int a)` 与刚才相同 `(a)`。
> - 参数用小括号括起来，用逗号分隔。例如 `(a, b)` 或 `(int a, int b)` 或 `(String a, int b, float c)`。
> - 空括号用于表示一组空的参数。例如 `() -> 42`。
> - 当有且仅有一个参数时，如果不显式指明类型，则不必使用小括号。例如 `a -> {return a*a;}`。
> - Lambda 表达式的正文可以包含零条，一条或多条语句。
> - 如果 Lambda 表达式的正文只有一条语句，则大括号可不用写，且表达式的返回值类型要与匿名函数的返回类型相同
>   - 接口返回值不为void时
>   - 如果没写大括号，那么返回值不需要写return
>   - 如果有大括号，需要加上return
>   - 接口返回值为void时
>   - 如果没写大括号，那么返回值不需要写return
>   - 如果有大括号，可以不需要return（如果想在大括号中写一个return;也是可以的）
> - 如果 Lambda 表达式的正文有一条以上的语句必须包含在大括号（代码块）中，且表达式的返回值类型要与匿名函数的返回类型相同
>   - 接口返回值不为void时，必须使用return
>   - 接口返回值为void时，可以不需要return（如果想在大括号中写一个return;也是可以的）

> lambda 表达式和内部类可以做到闭包的效果-

Lambda表达式与匿名内部类区别:

> 所需类型不同
> 匿名内部类：可以是接口，也可以是抽象类，还可以是具体类
> Lambda表达式：只能是接口
>
> 使用限制不同
> 如果接口中有且仅有一个抽象方法，可以使用Lambda表达式，也可以使用匿名内部类
> 如果接口中多于一个抽象方法，只能使用匿名内部类，而不能使用Lambda表达式
>
> 实现原理不同
> 匿名内部类：编译之后，产生一个单独的.class字节码文件
> Lambda表达式：编译之后，没有一个单独的.class字节码文件，对应的字节码会在运行的时候动态生成，通过invokedynamic实现
>
> this不同，产生两者区别的原因是实现原理不同
> 匿名内部类：在实现方法中，this为匿名类对象
> Lambda表达式：在static方法中可以使用lambda表达式，但不能在其中用this；在实例方法上下文中，this为调用实例方法的对象，可以理解为闭包中的变量包含了this

有机会可以进一步分析匿名表达式的实现原理

流式编程

```
Stream<Integer> integerStream = Stream.of(1,2,4,5);
Optional<Integer> reduce = integerStream.reduce((x, y) -> x + y);//0元素是empty，1元素值为唯一值，多元素就进行叠加;是Optional，因为未定义初始值
System.out.println(reduce);

Stream<Integer> integerStream2 = Stream.of(1,2,3,4);
Integer reduce1 = integerStream2.reduce(-1, (x, y) -> {
    synchronized (Java8Test.class){System.out.println("**"+x);
                                   System.out.println("**"+y);}
    return x + y;});
System.out.println(reduce1);//有初始值，所以返回不是Optional，而是整数；并行时，会多次使用identity;串行时，只是用一次identity

//并行时会产生多个子流，combiner作用是将子流结果合并；多个子流开始时，都会使用identity，子流结果作为combiner的输入，combiner的结果作为combiner的输入;
//（并行时，接上面）此处和前一个reduce的区别是前一个reduce可以理解为accumulator和combiner是同一个；而此处combiner可以定义成与accumulator不同
//-----------------
//串行时，只会使用accumulator进行累加，只使用一次identity
Stream<Integer> integerStream3 = Stream.of(1,2,3,4);
Integer reduce2 = integerStream3.reduce(0,(x, y) -> {
    System.out.println("i"+x);
    System.out.println("i"+y);
    return x + y;},
                                        (x,y)->{
                                            System.out.println("e"+x);
                                            System.out.println("e"+y);
                                            return x * y;});
```

```
System.out.println(reduce2);
```

[「java8系列」流式编程Stream - SegmentFault 思否]

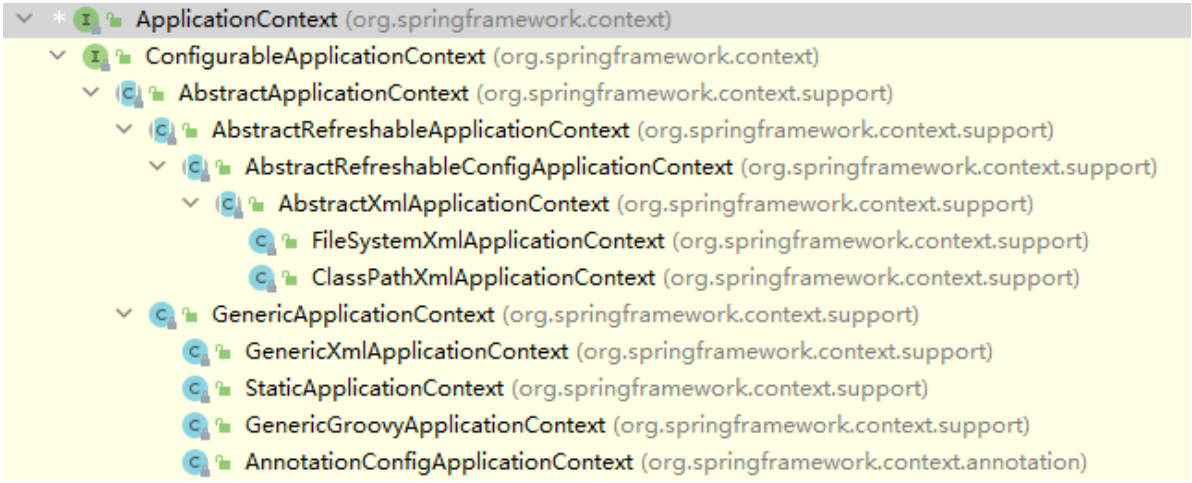
继承相关

构造函数

```
1.子类构造函数可以显示调用父类构造函数
2.子类构造函数可以不显示调用父类构造函数，这是父类默认无参构造函数将会调用，称为隐式调用
3.类中可以不显示定义构造函数，默认有一个默认的无参构造函数
4.如果显示定义了有参构造函数，默认的无参构造函数将不生效
5.父类如果只定义了有参构造，子类将不能显式或隐式调用父类的无参构造，因为无参构造不存在
6.子类构造函数必须调用父类构造函数
```

SpringIOC容器

容器接口及实现类




```
ClassPathXmlApplicationContext是解析XML的容器
AnnotationConfigApplicationContext是解析注解的容器
```

IOC容器特征

- 创建Bean

```
//先通过无参构造函数创建对象；如果没有无参构造函数，只有有参构造的话，会报错
//创建对象后，通过set方法注入属性，没有set会报错
//Spring默认启动时会实例化所有单例Bean,如果想在启动的时候延迟加载bean，可以使用@Lazy注解在使用时才初始化
//BeanDefinition封装了Bean的信息，Spring依据BeanDefinition来实例化Bean;相关的还有一个dependsOn,表示在一个bean创建之前，其它bean都要准备好
```

- 获取Bean

```
//对于一般的Bean可以通过context.getBean("name")获取
//对于实现了org.springframework.beans.factory.FactoryBean的工厂Bean，通过context.getBean("name")获取的是FactroyBean.getObject()得到的对象；如果想获取FactoryBean本身，需要context.getBean("&name")加上&来进行获取
```

- Bean生命周期

```
//@Bean(value = "car",initMethod = "init", destroyMethod = "destroy")可以指定Bean的init方法和destroy方法；在Car类中定义init和destroy方法即可
Car...constructor..
Car...init...
容器创建完成.....
context.close();
Car...destroy...
//Dog实现InitializingBean, DisposableBean
//InitializingBean Interface to be implemented by beans that need to react （做出反应） once all their properties have been set by a BeanFactory；所有属性设置完成后，才会调用afterPropertiesSet做出反应
public void afterPropertiesSet() throws Exception
{System.out.println("Dog...afterPropertiesSet...");}
public void destroy() throws Exception
{System.out.println("Dog...destroy...");}
Dog...constructor...
Dog...afterPropertiesSet...
容器创建完成.....
context.close();
Dog...destroy...
//Mouse实现@PostConstruct、@PreDestroy
//The PostConstruct annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization
@PostConstruct
public void init(){System.out.println("Mouse..@PostConstruct...");}
@PreDestroy
public void destroy(){System.out.println("Mouse..@PreDestroy...");}
Mouse...constructor...
Mouse..@PostConstruct...
容器创建完成.....
context.close();
Mouse..@PreDestroy...
```

- @Configuration定义一个配置类，等同于以前的xml文件

```
@Component
public @interface Configuration {
    /**
     * Explicitly specify the name of the Spring bean definition associated with the
     * {@code @Configuration} class. If left unspecified (the common case), a bean
     * name will be automatically generated.
     * <p>The custom name applies only if the {@code @Configuration} class is picked
     * up via component scanning or supplied directly to an
```

```
     * {@link AnnotationConfigApplicationContext}. If the {@code
@Configuration} class
     * is registered as a traditional XML bean definition, the name/id of
the bean
     * element will take precedence.
     * @return the explicit component name, if any (or empty String
otherwise)
     * @see AnnotationBeanNameGenerator
     */
    @AliasFor(annotation = Component.class)
    String value() default "";//可以指定BeanDefinition的名称
    /**
     * Specify whether {@code @Bean} methods should get proxied in order to
enforce
     * bean lifecycle behavior, e.g. to return shared singleton bean
instances even
     * in case of direct {@code @Bean} method calls in user code. This
feature
     * requires method interception, implemented through a runtime-generated
CGLIB
     * subclass which comes with limitations such as the configuration class
and
     * its methods not being allowed to declare {@code final}.
     * <p>The default is {@code true}, allowing for 'inter-bean references'
via direct
     * method calls within the configuration class as well as for external
calls to
     * this configuration's {@code @Bean} methods, e.g. from another
configuration class.
     * If this is not needed since each of this particular configuration's
{@code @Bean}
     * methods is self-contained and designed as a plain factory method for
container use,
     * switch this flag to {@code false} in order to avoid CGLIB subclass
processing.
     * <p>Turning off bean method interception effectively processes {@code
@Bean}
     * methods individually like when declared on non-{@code @Configuration}
classes,
     * a.k.a. "@Bean Lite Mode" (see {@link Bean @Bean's javadoc}). It is
therefore
     * behaviorally equivalent to removing the {@code @Configuration}
stereotype.
     * @since 5.2
     */
    // 下面是springboot中的一小段代码
    // true 表示的是Full模式， false表示的是Lite（轻量）模式（满足
isConfigurationCandidate的也算Lite配置类，这个方法详见后文分析）
    //if (config != null &&
!Boolean.FALSE.equals(config.get("proxyBeanMethods"))) {
    //  beanDef.setAttribute(CONFIGURATION_CLASS_ATTRIBUTE,
CONFIGURATION_CLASS_FULL);
    //}
    //else if (config != null || isConfigurationCandidate(metadata)) {
    //    beanDef.setAttribute(CONFIGURATION_CLASS_ATTRIBUTE,
CONFIGURATION_CLASS_LITE);
    //}
    // 是否让@Bean标注的方法被代理，true表示代理，false不代理；
```

```
    // true:表示@Bean方法会被代理，也就是配置类会被CGLIB增强，可以直接调用配置类的方法，产生的都是相同的对象（核心是调用BeanFacotry.getBean）
    // false：表示@Bean方法不会被代理，配置类也不会被CGLIB增强，调用@Bean标注的方法和普通方法没区别，每次都会执行一遍
    // 如果配置类内部多个@Bean方法间没有互相调用的需求，也没有外部直接调用需求，仅仅只是做为一个工厂方法被容器调用一次并将实例注册到容器中，
    // 那么设置为false可以避免配置类被CGLIB增强，提高SpringBoot启动速度
    boolean proxyBeanMethods() default true;
}
```

- @Bean 向容器中加入一个bean，无参数时默认以方法名作为bean的id，有参数时@Bean("beanid")以"beanid"作为bean的id
- @PropertySource(value = {"classpath:/iocperson.properties"})

```
//通常和@Configuration一起使用,会将properties文件中的 key/value person.nickname=小明 解析到org.springframework.core.env.Environment中（不仅有解析的文件内容，还包含系统环境变量systemEnvironment和一些虚拟机相关的systemProperties；springboot中自定义属性也会放入Environment中）
//可以通过context.getEnvironment().getProperty("person.nickname")获取值
//可以通过@Value进行数据注入（ #{systemProperties.myProp} style SpEL(Spring EL) or property placeholder such as ${my.app.myProp}）
@Value("张三")
private String name;//name=张三
@Value("#{20-2}")
private int age;//age=18
@Value("${person.nickname}")
private String nickname;//nickname=小明
```

- @ComponentScan("com.example.bean") 可以指定要扫描的包，一般写在配置类上
- @Import 可以通过不同的方式引入别的对象

```
//@Import({Red.class, Blue.class, MyImportSelector.class,MyImportBeanDefinitionRegistrar.class})
//其中，可以引入@Configuration, ImportSelector, ImportBeanDefinitionRegistrar, or regular component classes to import.
//Red/Blue都是regular component，会创建该类对应的对象
//MyImportSelector implements ImportSelector
public String[] selectImports(AnnotationMetadata importingClassMetadata)
{return new String[]{"com.example.bean.Green"};}
//MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar
public void registerBeanDefinitions(AnnotationMetadata
importingClassMetadata, BeanDefinitionRegistry registry)
{RootBeanDefinition rootBeanDefinition = new
RootBeanDefinition("com.example.bean.Yellow");
 registry.registerBeanDefinition("yellow",rootBeanDefinition);}
```

- @Conditional

```
//要是Condition不满足，那么同一位置的所有其它注解都不会生效（比如Import）；相对于其它注解的顺序不影响结果;
//通过实现org.springframework.context.annotation.Condition接口可以自定义条件类
//WindowsCondition implements Condition
@Override
```

```
public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
    ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
    Environment environment = context.getEnvironment();
    ClassLoader classLoader = context.getClassLoader();
    BeanDefinitionRegistry registry = context.getRegistry();
    //        registry.getBeanDefinition()
    String property = environment.getProperty("os.name");
    System.out.println(property);
    if(property.contains("Windows"))
        return true;
    return false;
}
```

- @Autowired

```
//1.先按类型装载，找不到则抛出异常，除非显示加上@Autowired(required = false)属性，那么值为null；找到一个就直接返回；找到多个的话，尝试按变量名加载，找不到唯一的话抛出异常（即使required=false也是如此）；
//2.步骤1中可以改变按照变量名加载的规则，通过加上限定符@Qualifier("beanid")，强制让容器按"beanid"查找，找不到抛出异常;除非@Autowired有required = false，那么结果会赋值为null;
//3.步骤1中可以改变按照变量名加载的规则，通过加上@Primary，那么多个bean时将以@Primary对应的bean为注入bean;仅允许一个@Primary，如果存在多个则报错；
//4.步骤2和步骤3同时使用不会冲突，@Qualifier("beanid")优先级更高，会使用"beanid"让容器进行查找，如果找不到抛出异常（即使存在标注了@Primary的bean还是如此，不会使用@Primary做为补救手段，两种机制只生效一个@Qualifier）;除非@Autowired有required = false，那么结果会赋值为null;

@Autowired是Spring的注解,@Resource是java的标准注解；别的框架可以识别@Resource(扩展性更好)，但不能识别@Autowired（功能更强）;@Resource可以按名称注入也可以按类型注入，不支持required=false;@Inject应用于EJB，基本很少使用；@Resource无参数时，和@Autowired等同，可以和Qualifier以及Primary配合使用，其它更细节的方面，用到再分析；

//默认情况Bean是单例模式；如果配置了@Scope("prototype"),就会变成多实例情形，每次申请Bean都活获取新的Bean
//@Scope单例多例关系
//在单例Bean中注入多例Bean时，如果仅仅对多例Bean使用@Scope("prototype")是不行的，因为单例Bean只创建一次，所以内部的多例Bean也只会创建一次；一种解决办法是通过ApplicationContextAware等感知接口注入到Bean中，然后手动从容器中再次获取多例Bean对象，这样才会产生多例Bean；
//在多例Bean中注入单例Bean时，始终只有一个单例Bean

@Autowired可以使用到构造函数上，当只有一个有参构造函数时，可以省略该注解
```

- @Order, defines the sort order for an annotated component

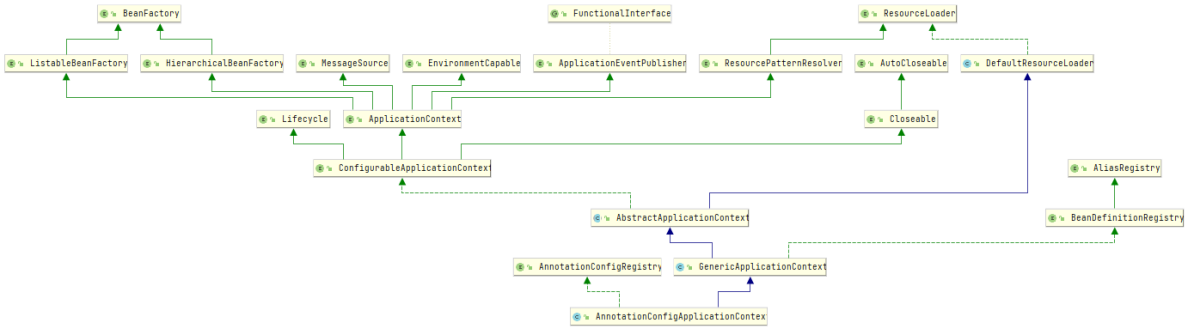
```
//@Order: Lower values have higher priority. The default value is
Ordered.LOWEST_PRECEDENCE, indicating lowest priority
public @interface Order {
    /**
     * The order value.
     * <p>Default is {@link Ordered#LOWEST_PRECEDENCE}.
     * @see Ordered#getOrder()
     */
    int value() default Ordered.LOWEST_PRECEDENCE;
```

```
}

public interface Ordered {
    int HIGHEST_PRECEDENCE = Integer.MIN_VALUE;
    int LOWEST_PRECEDENCE = Integer.MAX_VALUE;
    /**
     * Get the order value of this object.
     * <p>Higher values are interpreted as lower priority. As a consequence,
     * the object with the lowest value has the highest priority
     * <p>Same order values will result in arbitrary sort positions for the
affected objects.
     */
    int getOrder();
}
//Extension of the Ordered interface, expressing a priority ordering:
PriorityOrdered objects are always applied before plain Ordered objects
regardless of their order values.
//When sorting a set of Ordered objects, PriorityOrdered objects and plain
Ordered objects are effectively treated as two separate subsets, with the
set of PriorityOrdered objects preceding the set of plain Ordered objects
and with relative ordering applied within those subsets.
public interface PriorityOrdered extends Ordered {
}
```

IOC容器流程分析

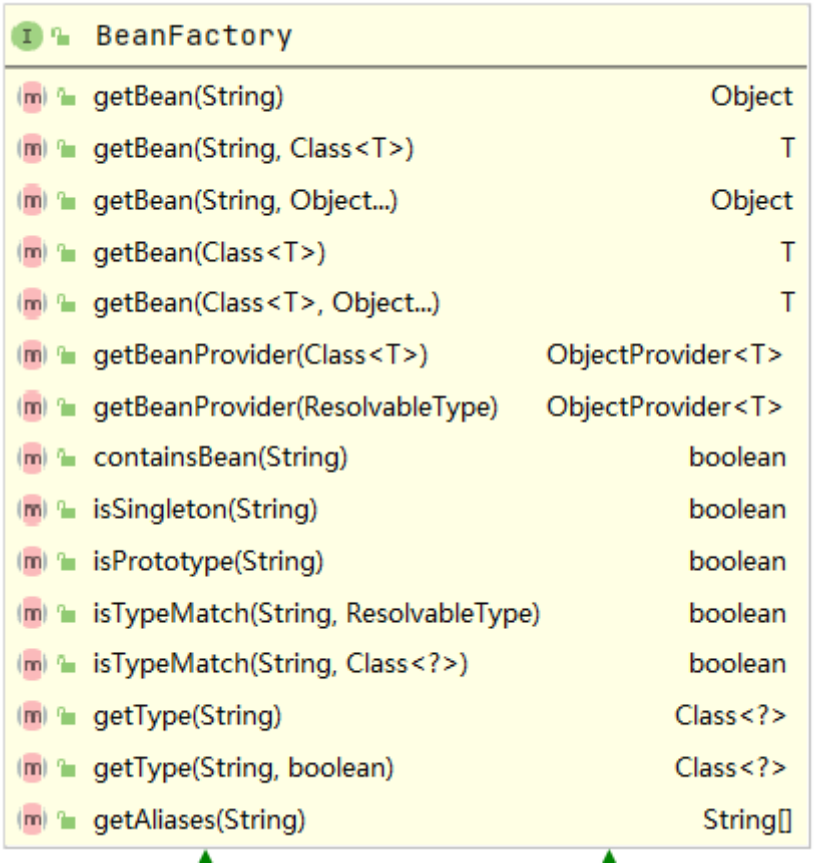
AnnotationConfigApplicationContext分析

接口

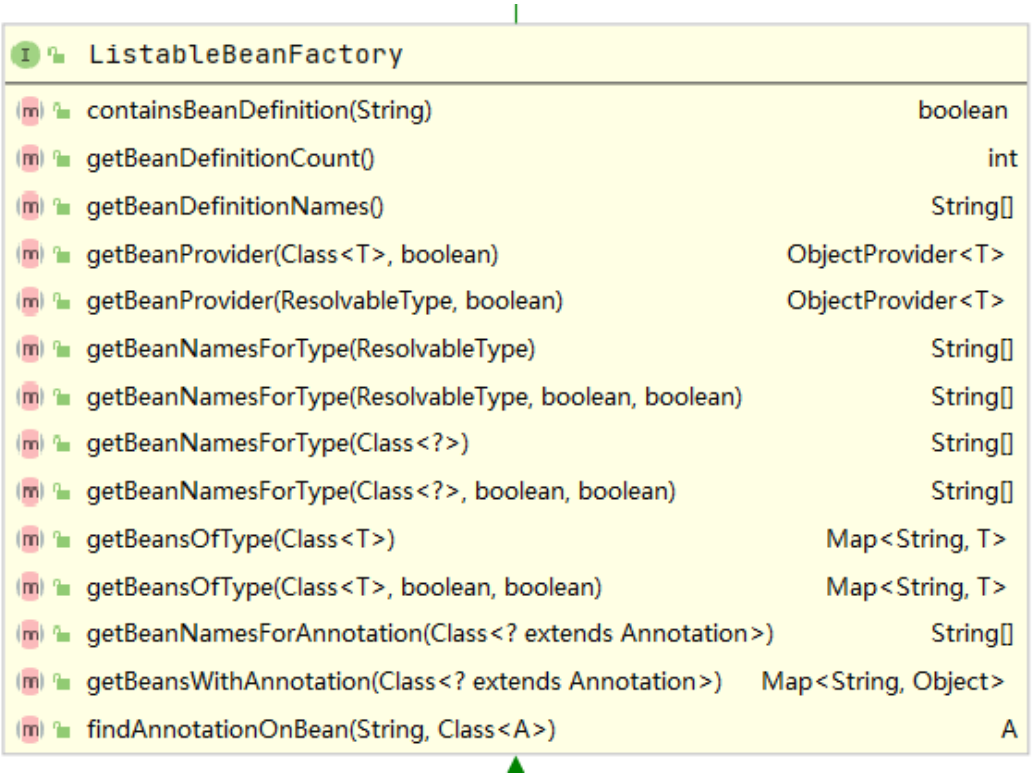


从接口角度看容器ApplicationContext

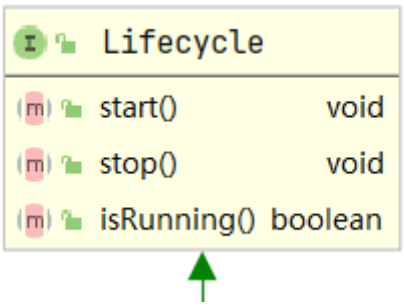
- BeanFactory



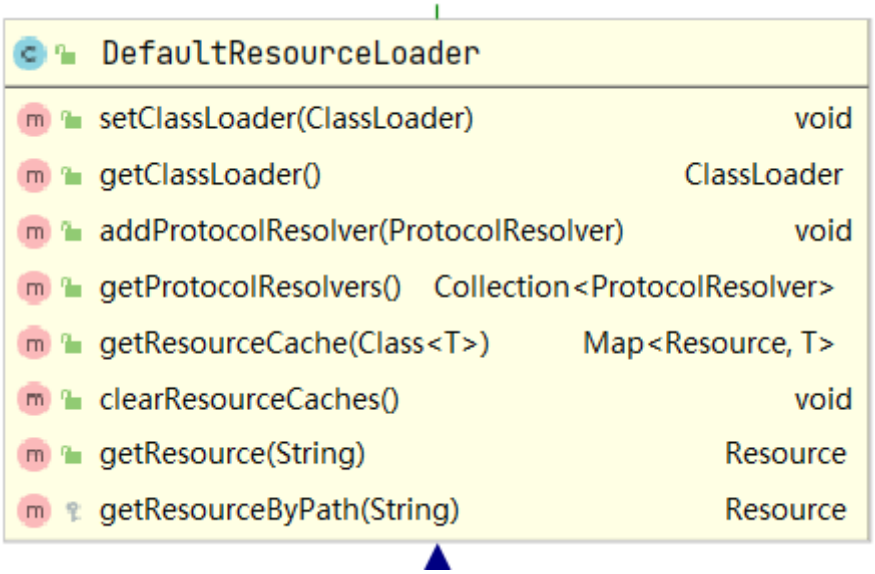
- ListableBeanFactory



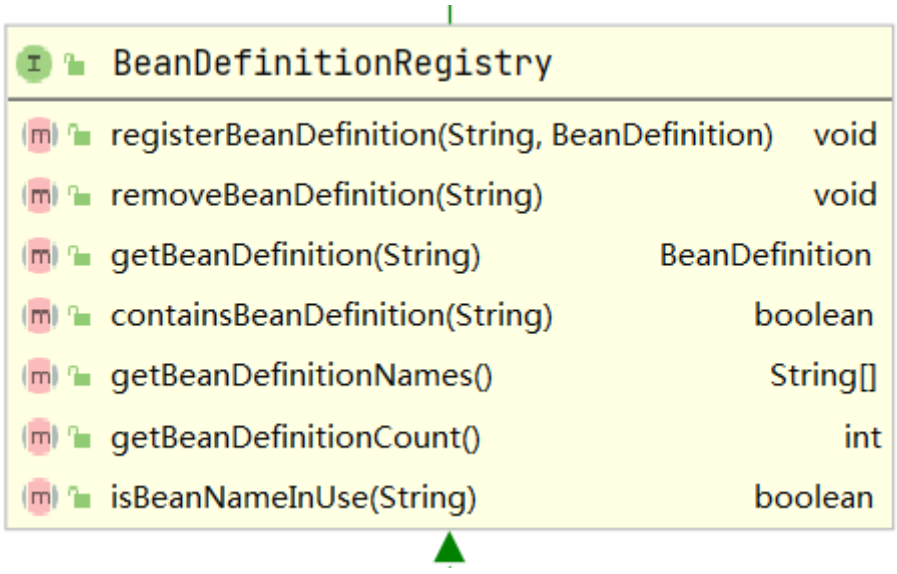
- Lifecycle



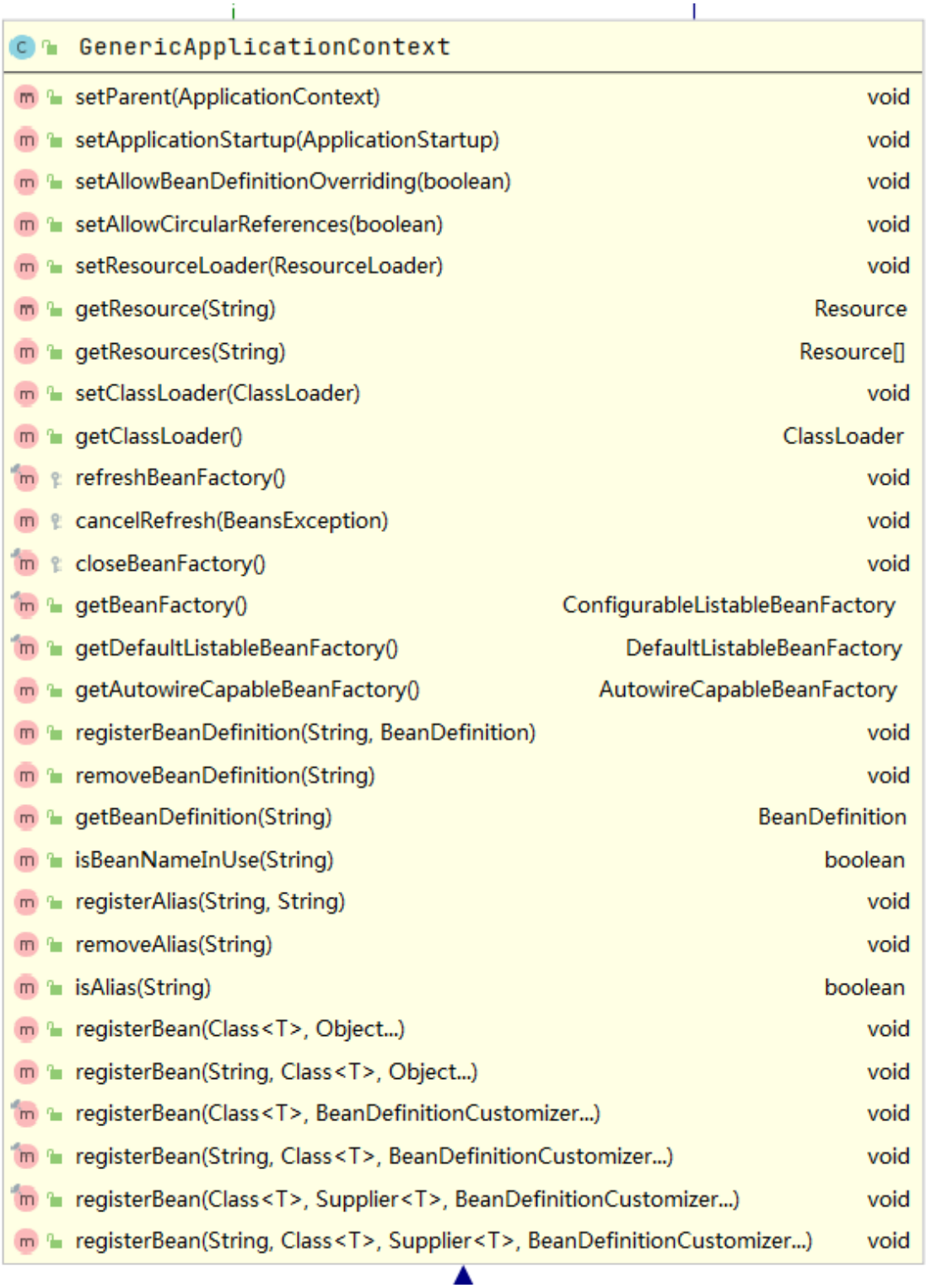
- ResourceLoader
  - DefaultResourceLoader



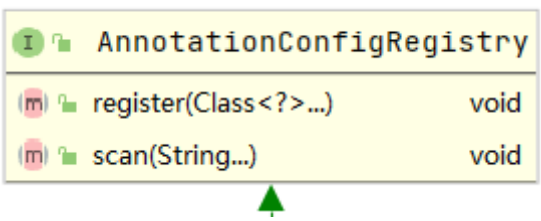
- BeanDefinitionRegistry



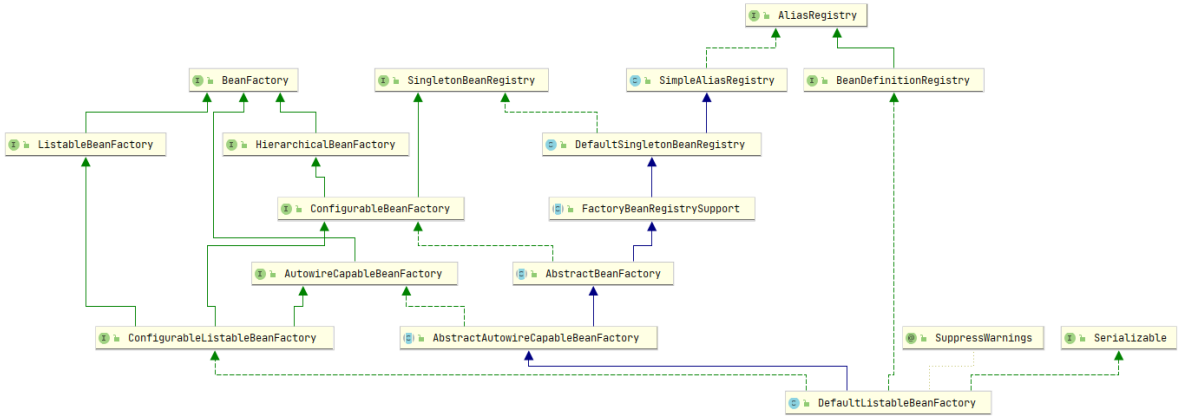
- GenericApplicationContext



- AnnotationConfigRegistry 添加编程方式的类注册以及包扫描形式的注册;该接口通过组合(组合是has-a关系，继承是is-a关系)的设计模式实现，一个内部对象为AnnotatedBeanDefinitionReader(编程注册)，另一个内部对象为ClassPathBeanDefinitionScanner(包扫描)

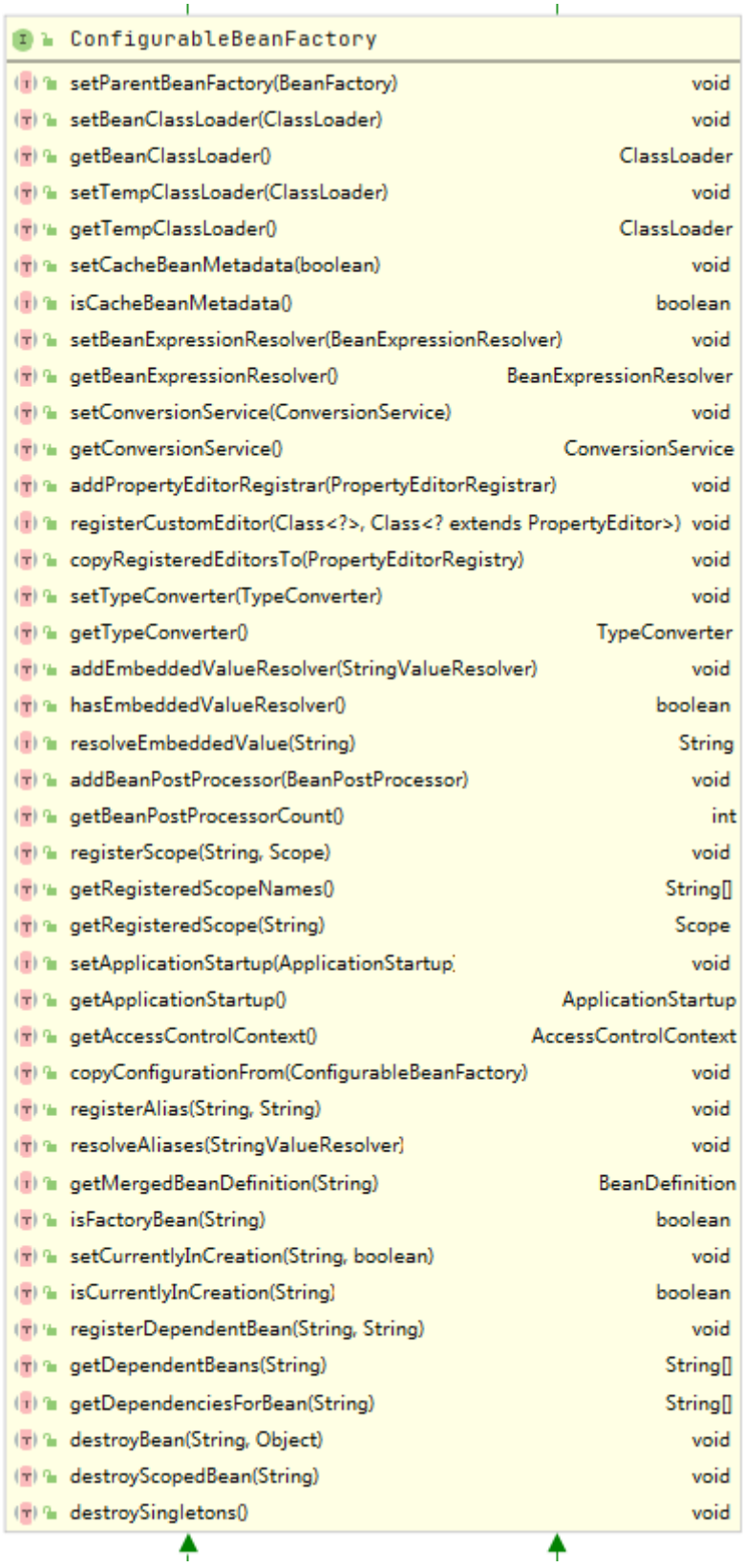


在创建AnnotationConfigApplicationContext时，会默认调用父类GenericApplicationContext的无参构造函数，其中定义了`this.beanFactory = new DefaultListableBeanFactory();`，而这个DefaultListableBeanFactory是真正存放bean等对象的位置，下面是该类的继承关系

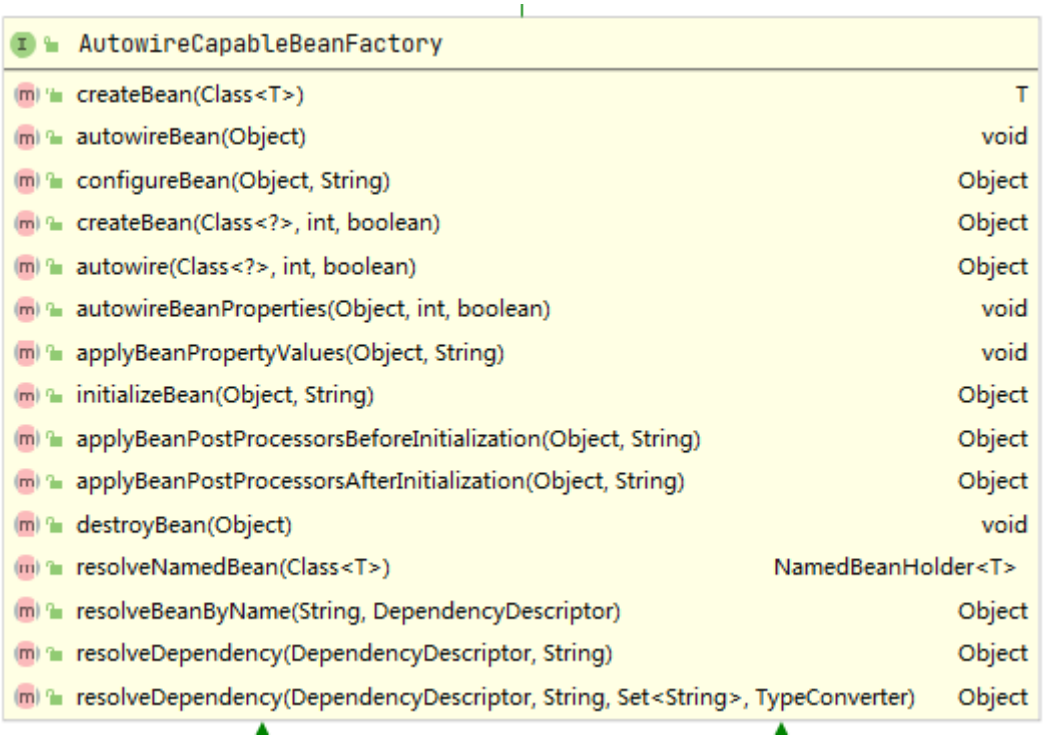


可以看到这里也实现了BeanFactory和ListableFactory，Spring使用了组合(has-a)的设计模式通过DefaultListableBeanFactory()得到beanFactory来实现ApplicationContext的BeanFactoy/ListableBeanFactory等接口功能，具体细节体现在GenericApplicationContext的父类AbstractApplicationContext中(e.g. getBeanFactory().getBean(name)，而getBeanFactory()就是DefaultListableBeanFactory()得到的beanFactory)；另外，又引入了ConfigurableBeanFactory/AutowireCapableBeanFactory/AbstractAutowireCapableBeanFactory等接口和类，增加setParentBeanFactory/addBeanPostProcessor等配置方法，增加autowire/autowireBean/autowireByName/autowireByType等自动装配方法，最终得到一个Listable/Configurable/AutowireCapable/BeanDefinitionRegistry的一个BeanFactory，下面列举一些关键接口和类

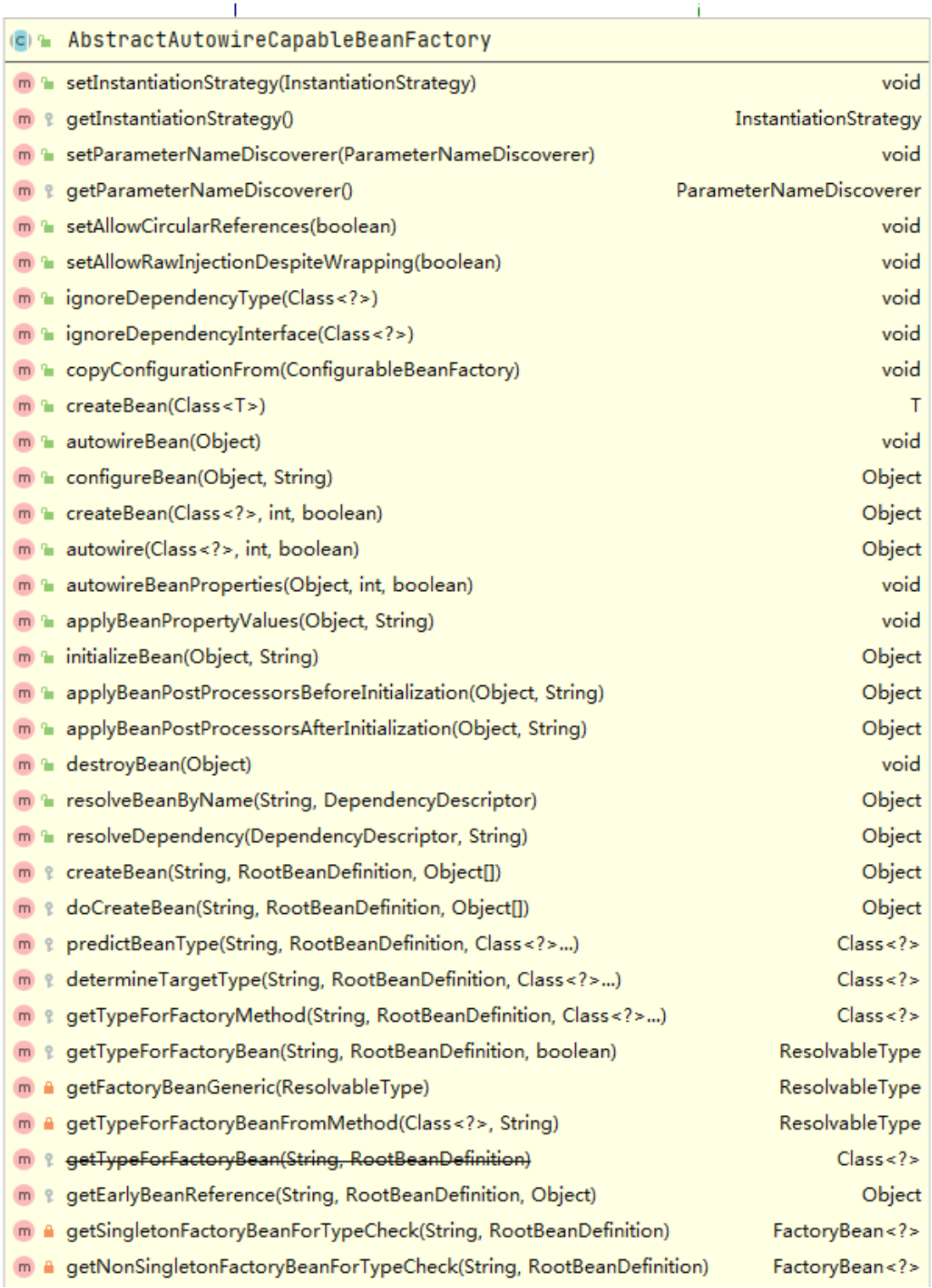
- ConfigurabelBeanFactory

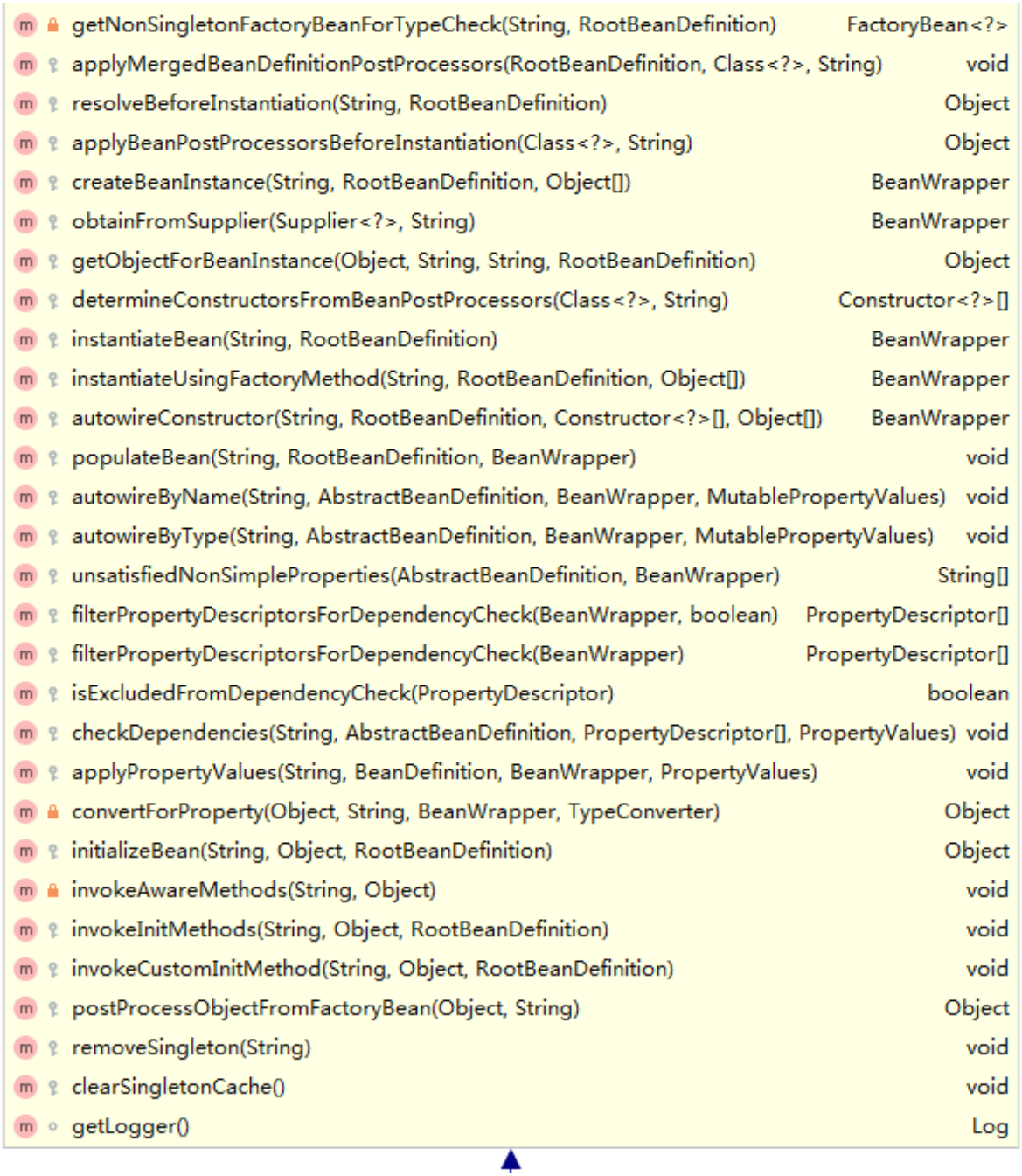


- AutowireCapableBeanFactory



- AbstractAutowireCableBeanFactory


AbstractAutowireCapableBeanFactory
setInstantiationStrategy(InstantiationStrategy) void
getInstantiationStrategy() InstantiationStrategy
setParameterNameDiscoverer(ParameterNameDiscoverer) void
getParameterNameDiscoverer() ParameterNameDiscoverer
setAllowCircularReferences(boolean) void
setAllowRawInjectionDespiteWrapping(boolean) void
ignoreDependencyType(Class<?>) void
ignoreDependencyInterface(Class<?>) void
copyConfigurationFrom(ConfigurableBeanFactory) void
createBean(Class<T>) T
autowireBean(Object) void
configureBean(Object, String) Object
createBean(Class<?>, int, boolean) Object
autowire(Class<?>, int, boolean) Object
autowireBeanProperties(Object, int, boolean) void
applyBeanPropertyValues(Object, String) void
initializeBean(Object, String) Object
applyBeanPostProcessorsBeforeInitialization(Object, String) Object
applyBeanPostProcessorsAfterInitialization(Object, String) Object
destroyBean(Object) void
resolveBeanByName(String, DependencyDescriptor) Object
resolveDependency(DependencyDescriptor, String) Object
createBean(String, RootBeanDefinition, Object[]) Object
doCreateBean(String, RootBeanDefinition, Object[]) Object
predictBeanType(String, RootBeanDefinition, Class<?>...) Class<?>
determineTargetType(String, RootBeanDefinition, Class<?>...) Class<?>
getTypeForFactoryMethod(String, RootBeanDefinition, Class<?>...) Class<?>
getTypeForFactoryBean(String, RootBeanDefinition, boolean) ResolvableType
getFactoryBeanGeneric(ResolvableType) ResolvableType
getTypeForFactoryBeanFromMethod(Class<?>, String) ResolvableType
getTypeForFactoryBean(String, RootBeanDefinition) Class<?>
getEarlyBeanReference(String, RootBeanDefinition, Object) Object
getSingletonFactoryBeanForTypeCheck(String, RootBeanDefinition) FactoryBean<?>
getNonSingletonFactoryBeanForTypeCheck(String, RootBeanDefinition) FactoryBean<?>



上面是从接口的方面了解IOC容器，下面从实现流程进行分析

构造方法

```
//context = new AnnotationConfigApplicationContext(MainConfig.class); MainConfig是@Configuration
//System.out.println("容器创建完成.....");

public AnnotationConfigApplicationContext(Class<?>... componentClasses) {
    //this()之前还会调用父类无参构造public GenericApplicationContext(){this.beanFactory = new DefaultListableBeanFactory();}
    this();
    register(componentClasses);//AnnotationConfigRegistry接口的register功能,编程形式指定注册类
    refresh();//容器的核心方法
}
public AnnotationConfigApplicationContext() {
```

```
        StartupStep createAnnotatedBeanDefReader =
this.getApplicationStartup().start("spring.context.annotated-bean-
reader.create");//应该是设计用来记录启动过程中一些步骤的开始和结束，但是目前的实现类，基本什
么也没做
        this.reader = new AnnotatedBeanDefinitionReader(this);//实现了
AnnotationConfigRegistry接口的register功能，此处还会在beanFactory注册5个
BeanDefintion，其中包含一些BeanFactoryPostProcessor等，比如
ConfigurationClassPostProcessor，AutowiredAnnotationBeanPostProcessor，
CommonAnnotationBeanPostProcessor
        createAnnotatedBeanDefReader.end();
        this.scanner = new ClassPathBeanDefinitionScanner(this);//实现了
AnnotationConfigRegistry接口的scan功能
}
//ApplicationContext通过组合的设计模式来实现主要的功能，this.beanFactory = new
DefaultListableBeanFactory()可以理解为真正的实现对象
```

核心Refresh()

1. **prepareRefresh**
    1. 设置startupDate、closed，active状态
    2. initPropertySources：空方法，可以由子类实现
    3. getEnvironment().validateRequiredProperties()
        - 验证环境中是否包含RequiredProperties（see ConfigurablePropertyResolver#setRequiredProperties）
    4. 设置初始的earlyApplicationListeners
2. **obtainFreshBeanFactory**
    1. refreshBeanFactory
        - 设置refreshed为true，目前仅允许执行一次refresh()方法，如果之前已经为true，那么会抛出异常
        - 设置beanFactory的setSerializationId（序列化ID，目的用于通过该ID进行反序列化）
    2. getBeanFactory：直接返回GenericApplicationContext的构造方法创建的DefaultListableBeanFactory，返回值beanFactory做为后续方法的参数
3. **prepareBeanFactory(beanFactory)**
    1. 设置beanFactory类加载器、设置SpEL(#{exp})解析组件StandardBeanExpressionResolver、设置PropertyEditorRegistrar（其包含ResourceLoader和PropertyResolver(实际对象为StandardEnvironment，可以处理${} placeholder,还涉及system properties、system environment variables))
    2. 设置一个BeanPostProcessor：ApplicationContextAwareProcessor implements BeanPostProcessor，干预以下接口的bean创建

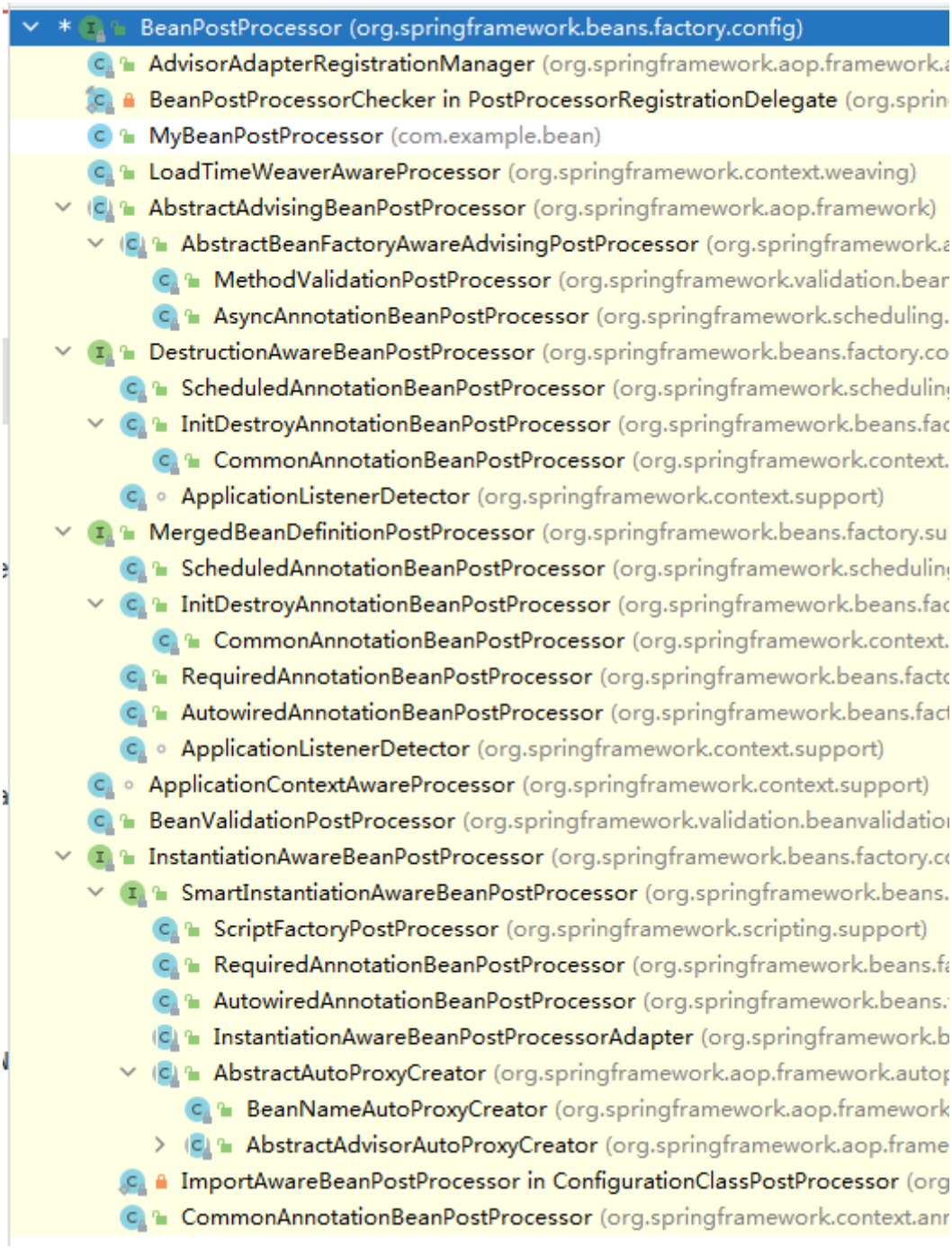
        ```
        bean instanceof EnvironmentAware || bean instanceof
        EmbeddedValueResolverAware ||
                        bean instanceof ResourceLoaderAware || bean instanceof
        ApplicationEventPublisherAware ||
                        bean instanceof MessageSourceAware || bean instanceof
        ApplicationContextAware ||
                        bean instanceof ApplicationStartupAware
        ```

        让Bean感知（Aware）到Spring底层的一些组件

        ```
        //Factory hook that allows for custom modification of new bean instances
        for example, checking for marker interfaces or wrapping beans with
        proxies. 主要用于标记接口及产生代理对象
        ```

```
//Registration:An ApplicationContext can autodetect BeanPostProcessor beans in its bean definitions and apply those post-processors to any beans subsequently created. A plain BeanFactory allows for programmatic registration of post-processors, applying them to all beans created through the bean factory.BeanPostPorcessor注册后会应用于所有其它类型Bean的创建
public interface BeanPostProcessor {
   //post-processors that populate beans via marker interfaces or the like will implement postProcessBeforeInitialization
   //Apply this BeanPostProcessor to the given new bean instance before any bean initialization callbacks (like InitializingBean's afterPropertiesSet or a custom init-method).The bean will already be populated with property values.
    @Nullable
    default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }
    //post-processors that wrap beans with proxies will normally implement postProcessAfterInitialization
    //Apply this BeanPostProcessor to the given new bean instance after any bean initialization callbacks (like InitializingBean's afterPropertiesSet or a custom init-method). The bean will already be populated with property values.
    @Nullable
    default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }
}
```

BeanPostProcessor有很多的子接口和实现类，扩展了各种“时机”下的后置处理，具体情况再具体分析



3. 禁止beanFactory对一些类型进行自动装配，主要和ApplicationContextAwareProcessor对应，让Spring的部分底层的组件只能通过Aware接口进行注入,不能通过Autowire进行装配

```
beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.cla
ss);
beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
beanFactory.ignoreDependencyInterface(ApplicationStartup.class);
基本和ApplicationContextAwareProcessor中的Aware对应
```

4. 设置beanFactory的一些type到value的默认装配值

```
beanFactory.registerResolvableDependency(BeanFactory.class,
beanFactory);
//this指ApplicationContext,ApplicationContext实现了ResourceLoader、
ApplicationEventPublisher、ApplicationContext
beanFactory.registerResolvableDependency(ResourceLoader.class, this);
beanFactory.registerResolvableDependency(ApplicationEventPublisher.class
, this);
beanFactory.registerResolvableDependency(ApplicationContext.class,
this);

registerResolvableDependency：
//Register a special dependency type with corresponding autowired value.
//Map from dependency type to corresponding autowired value.
Map<Class<?>, Object>
//This is intended for factory/context references that are supposed to
be autowirable but are not defined as beans in the factory 这里注册一些可装
配的类型和对应的对象，这些对象没被定义为Bean，但是应该支持自动装配，比如
Factory/context对象
this.resolvableDependencies.put(dependencyType, autowiredValue);
```

5. 注册ApplicationListenerDetector（后置处理器），以便后面创建bean的时候找出ApplicationListener，加入applicationContext

```
            beanFactory.addBeanPostProcessor(new
ApplicationListenerDetector(this));
                postProcessAfterInitialization{... if (bean instanceof
ApplicationListener) {...

 this.applicationContext.addApplicationListener((ApplicationListener<?>)
bean);
                ...}...}
```

6. 在beanFactory中注册beanName为environment、systemProperties、systemEnvironment、applicationStartup组件

4. **postProcessBeanFactory**

```
//Modify the application context's internal bean factory after its standard
initialization
//Allows post-processing of the bean factory in context subclasses
```

5. **invokeBeanFactoryPostProcessors** 这个过程是注册BeanDefintion的核心过程，从会解析用户定义(e.g. @Configuration)的BeanDefinitionRegistryPostProcessor(该接口会新增BeanDefintion，如果新增的也是此类型，那么会被循环解析处理)和BeanFactoryPostProcessor(此接口主要是修改BeanDefintion)

```
invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory,
                                List<BeanFactoryPostProcessor>
beanFactoryPostProcessors（这里数量为0，猜测可能需要子类在
                                postProcessBeanFactory中按需实现，才能让这里的
指默认不为0）) :
     1、首先beanFactoryPostProcessors进行for循环处理
        if (postProcessor instanceof BeanDefinitionRegistryPostProcessor) {
            BeanDefinitionRegistryPostProcessor registryProcessor =
                (BeanDefinitionRegistryPostProcessor) postProcessor;
            registryProcessor.postProcessBeanDefinitionRegistry(registry);
```

```
                registryProcessors.add(registryProcessor);
            }
            else {
                regularPostProcessors.add(postProcessor);
            }
```

由于首先beanFactoryPostProcessors数量为0，所以此处过程相当于什么也没做

2、获取beanFactory中的BeanDefinitionRegistryPostProcessor

获取beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);这一步主要会得到名为org.springframework.context.annotation.internalConfigurationAnnotationProcessor，类为ConfigurationClassPostProcessor的对象，其在第3步会注册@Configuration中定义的类中相关的BeanDefintion

3、对第2步中的结果按照PriorityOrdered、Ordered、未实现前两种排序接口的类 这3种方式分别排序处理（最后一种无需排序），最终调用postProcessBeanDefinitionRegistry，需要注意是此过程会增加BeanDefintion，如果BeanDefintion是BeanDefinitionRegistryPostProcessor，那么还会继续调用它的postProcessBeanDefinitionRegistry继续处理，直到没有新增的BeanDefinitionRegistryPostProcessor

4、继续调用BeanDefinitionRegistryPostProcessor的父接口BeanFactoryPostProcessors方法：

```
invokeBeanFactoryPostProcessors(registryProcessors, beanFactory);//对应PriorityOrdered、Ordered处理
invokeBeanFactoryPostProcessors(regularPostProcessors, beanFactory);//对应未实现前两种排序接口的类
```

这一步的后置处理器，实际上都是BeanDefinitionRegistryPostProcessor（这个类继承BeanFactoryPostProcessor），最终调用postProcessBeanFactory

5、获取所有BeanFactoryPostProcessor，去除BeanDefinitionRegistryPostProcessor之后，将剩下的BeanFactoryPostProcessor按照PriorityOrdered、Ordered、未实现前两种排序接口的类 这3种方式分别排序处理（最后一种无需排序），最终调用postProcessBeanFactory

总结：很多逻辑都是确认PostProcessor的处理顺序，除此之外的核心过程是先从BeanFactory中获取BeanDefinitionRegistryPostProcessor，调用invokeBeanDefinitionRegistryPostProcessors，这个过程会新增BeanDefintion，特别是BeanDefinitionRegistryPostProcessor类型的话，会循环处理知道没有新增的Registry为止，之后调用BeanFactoryPostProcessor的invokeBeanFactoryPostProcessors。再然后，处理不属于BeanDefinitionRegistryPostProcessor但属于BeanFactoryPostProcessor的Bean,通过invokeBeanFactoryPostProcessors调用它们的接口方法

上述过程并未实例化常规使用的 "Regular Bean",因为后续要注册BeanPostProcessors来影响"Regular Bean"过程

注意:此处处理的是

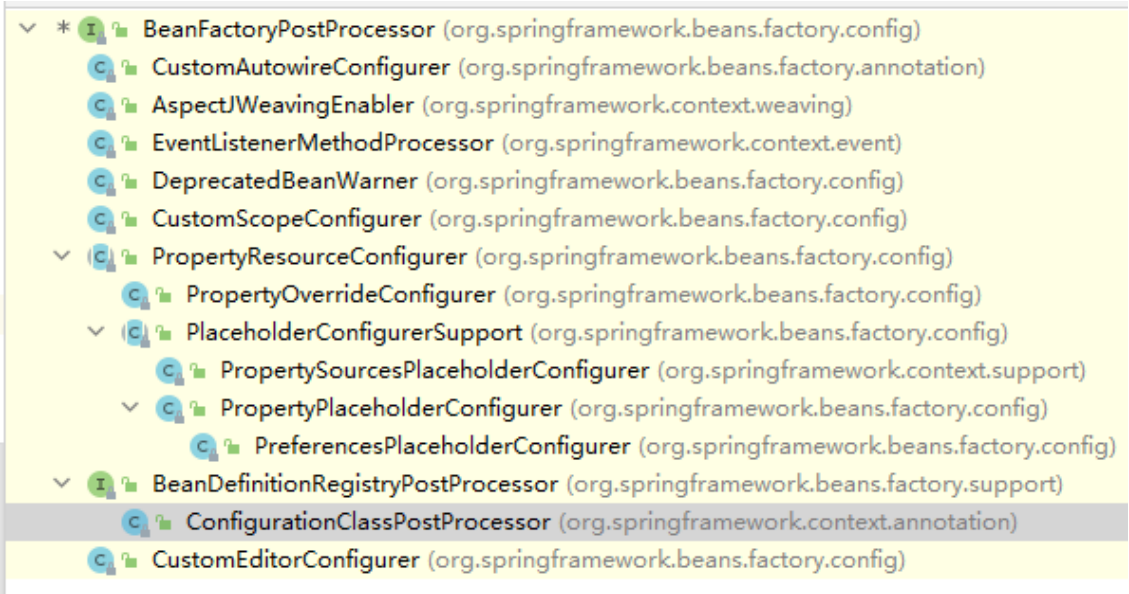
```
public interface BeanDefinitionRegistryPostProcessor extends BeanFactoryPostProcessor {
        /**
         * Modify the application context's internal bean definition registry after its
         * standard initialization.
         **All regular bean definitions will have been loaded 意味着regular的准备好了，但是还可以继续注册BeanDefinition**,
         * but no beans will have been instantiated yet. This allows for adding further
         * bean definitions before the next post-processing phase kicks in.
         */
        void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException;
```

```
    }
    public interface BeanFactoryPostProcessor {
    /**
     * Modify the application context's internal bean factory after its
standard
     * initialization.
     **All bean definitions will have been loaded 意味着执行时所有BeanDefintion
已经准备好了,此时允许添加后和修改BeanDefintion的属性**,
     * but no beans
     * will have been instantiated yet. This allows for overriding or adding
properties even to eager-initializing beans.
     */
        void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException;
    }
```

处理的是BeanFacory的后置处理，而不是一般Bean的后置处理,存在一个ConfigurationClassPostProcessor，是BeanDefinitionRegistryPostProcessor， used for bootstrapping processing of @Configuration classes，会解析由配置类，并产生的其它相关BeanDefinition加入BeanFactory

此时并未实例化“Regular Bean”,但是创建BeanDefinitionRegistryPostProcessor和BeanFactoryPostProcessor时，都使用beanFactory.getBean方法，这种情况下，创建的Bean都基本未经过后置处理器处理(因为还没注册用户的BeanPostProcessors，但是存在ApplicationContextAwareProcessor和ApplicationListenerDetector后置处理器)，所以在BeanFactoryPostProcessor不支持自动装配，因为其对应的自动装配后置处理还未在第6步注册

下面是BeanFactoryPostProcessor相关的接口及类关系图

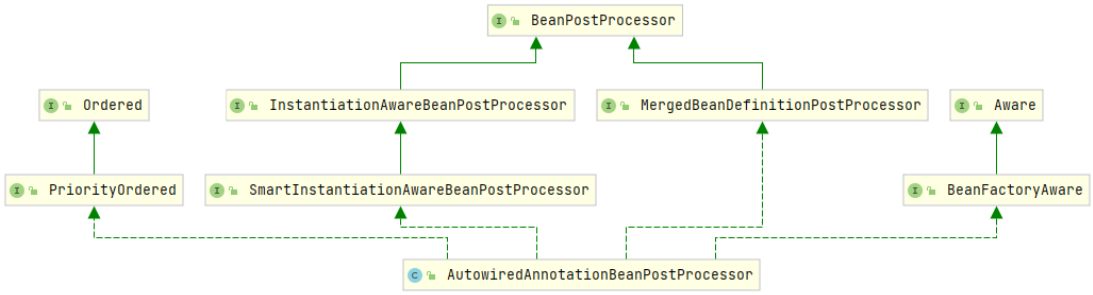


6. **registerBeanPostProcessors**，会解析用户定义(e.g. @Configuration)的BeanPostProcessor，并进行注册

```
registerBeanPostProcessors
    1、获取BeanPostProcessor：postProcessorNames =
beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false),这里基本
是从beanFactory的BeanDefinition中获取类型对应的Bean名称,这里意味着用户定义的Bean也可
能做为PostProcessor
    2、获取BeanPostProcessor目标数量并注册BeanPostProcessorChecker后置处理器：
            //beanFactory.getBeanPostProcessorCount()是之前代码中注册的
ApplicationListenerDetector等后置处理器
            //数量1 指的是后一行代码中注册的BeanPostProcessorChecker
            //postProcessorNames是beanFactory中定义的Bean
```

```
            int beanProcessorTargetCount =
beanFactory.getBeanPostProcessorCount() + 1 + postProcessorNames.length;
            beanFactory.addBeanPostProcessor(new
BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount));
    3、按照PriorityOrdered、Ordered、非排序的regularbeans从postProcessorNames获取并注册BeanPostProcessor，同时
        将MergedBeanDefinitionPostProcessor类型解析保存
    4、将所有的MergedBeanDefinitionPostProcessor(也称internal BeanPostProcessors)重新注册（先删除，再加入到内部BeanPostProcessors列表结尾，默认的自动装配后置处理就在末尾）
    5、将ApplicationListenerDetector重新注册（先删除，再加入到内部BeanPostProcessors列表结尾，在Common/AutowiredAnnotationBeanPostProcessor之后，其作用是发现Bean是ApplicationListener，之后加入内部Listener列表用于multicast event）
    //4/5的先删除再注册，等同于将之前注册过的BeanPostProcessors放到队列尾，使优先级最低
    //postProcessorNames对应两个重要的BeanPostProcessor,涉及自动装配和java JSR-250注解,其BeanDefintion产生于ApplicationContext构造函数定义AnnotatedBeanDefinitionReader的时候，两者都实现了PriorityOrdered接口
    //一个是AutowiredAnnotationBeanPostProcessor（autowires annotated fields, setter methods, and arbitrary config methods. Such members to be injected are detected through annotations: by default, Spring's @Autowired and @Value annotations.Also supports JSR-330's @Inject annotation, if available, as a direct alternative to Spring's own @Autowired.）
    //另一个是CommonAnnotationBeanPostProcessor(supports common Java annotations out of the box, in particular the JSR-250 annotations in the javax.annotation package,e.g.  PostConstruct/PreDestroy/Resource)
    //注意：由于beanFactory会持续注册BeanPostProcessor，而且还会通过beanFactory.getBean来创建新的BeanPostProcessor，这个过程对优先级敏感，细节方面可以查看源码；另外由于此时其它的用户Bean还未实例化，所以想在BeanFacotory中使用Autowired是不可行的
```

下面是AutowiredAnnotationBeanPostProcessor相关的接口和类关系图（BeanPostProcessor的接口和类关系图前文已有描述）



7. **initMessageSource**

```
如果用户定义了自己的名为messageSource的MessageSource，那么使用它注册到beanFactory(会经后置处理器处理)
如果没有就创建默认的DelegatingMessageSource并注册到beanFactory中（new出来的，手动注册，未经后置处理器处理,会记录在manualSingletonNames集合）
//MessageSource主要用于国际化
```

8. **initApplicationEventMulticaster**

如果用户定义了自己的名为applicationEventMulticaster的ApplicationEventMulticaster，那么使用它注册到beanFactory(会经后置处理器处理)
如果没有就创建默认的SimpleApplicationEventMulticaster，并注册到beanFactory中（new出来的，手动注册，未经后置处理器处理,会记录在manualSingletonNames集合）
//MessageSource主要用于国际化

9. **onRefresh**

模板空方法，留给子类按需实现//Initialize other special beans in specific context subclasses.

10. **registerListeners**

向ApplicationEventMulticaster注册用户定义的ApplicationListener,但是这里注册的时候只注册String类型的名称，只有后续使用的时候才会调用getBean创建对应Listener

11. **finishBeanFactoryInitialization** //Finish the initialization,initializing all remaining singleton beans.
    1. 初始化conversionService服务(默认无操作)，设置EmbeddedValueResolver可以Resolve ${...} placeholders，设置LoadTimeWeaverAware（加载时织入AOP相关），设置tempClassLoader为null来停止临时类加载器用于类型匹配，冻结所有已解析的BeanDefintions(目前来看仅仅是在frozenBeanDefinitionNames加入相关的BeanDefintionName)
    2. preInstantiateSingletons
        1. 遍历beanNames（实际就是beanDefinitionNames）
            1. 通过getMergedLocalBeanDefinition(beanName)获取RootBeanDefinition bd；

               //getMergedLocalBeanDefinition(String beanName) 先尝试从本地的mergedBeanDefinitions获取stale（过时）属性为false的beanDefintion，如果获取到则返回；没有获取到则调用getMergedBeanDefinition(beanName, getBeanDefinition(beanName))，其中getBeanDefinition(beanName)是从beanDefinitionMap获取对应的beanDefintion信息
               //getMergedBeanDefinition(beanName, getBeanDefinition(beanName)=bd) 如果mergedBeanDefinitions缓存存在beanName对应的合并后的BeanDefinition,则返回它；否则，先判断bd是否有父BeanDefintion,如果没有，那么就以bd生成一个新的RootBeanDefinition;否则，先获取parentBeanDefinition副本，然后利用parentBeanDefinition.overrideFrom(bd)来合并生成新的RootBeanDefinition(子bd覆盖父bd设置);接着会对BeanDefintion做一些额外的处理（A bean contained in a non-singleton bean cannot be a singleton itself，暂时未遇到这种情况，以后再说）；再然后将生成的RootBeanDefintion加入到mergedBeanDefinitions保存，返回新的RootBeanDefintion；

            2. 如果!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit() 非抽象、单例、非懒加载则进入3，否则什么也不处理继续循环遍历；
            3. 判断isFactoryBean，如果是则调用getBean(&beanName)先实例化FactoryBean对象(内部MAP记录的该对象实际不包含&)，接着判断是否为SmartFactoryBean，如果是调用该接口的isEagerInit()判断是否需要立即初始化FactoryBean暴露的对象(一般只有单例才有意义)，如果是则调用getBean(beanName)（此处相当于外部调用了一次获取Bean操作；前面第一次是实例化FactoryBean，这次是实例化

FactoryBean内部维护的对象（一般是指单例）），否则不处理继续循环;FactoryBean如果是单例的话，它得自己在内部维护这个缓存，Spring容器不管理FactoryBean.getObject对象

4. 非FactoryBean，直接调用getBean(beanName)//3、4步骤都调用了getBean(beanName)，这个方法是创建Bean的核心方法

2. 再次遍历beanNames（实际就是beanDefinitionNames）
    1. 通过getSingleton(beanName)获取单例对象
    2. 如果单例对象singletonInstance instanceof SmartInitializingSingleton，调用afterSingletonsInstantiated；否则什么也不做

```
//调用afterSingletonsInstantiated会保证，所有单例对象都已经被创建，原注释如下：
//with a guarantee that all regular singleton beans have been created already.
//实际上是第一次遍历beanNames，已经将非抽象、单例、非懒加载的对象通过getBean(beanName)初始化了
```

> 前面过程中getBean核心方法分析

getBean(name)

1. 调用doGetBean(name, null, null, false)
    1. String beanName = transformedBeanName(name);//先去除&符号再将name通过aliasMap(一般为空返回自己)转换成规范的beanName
    2. Object sharedInstance = getSingleton(beanName);
        - 依次尝试通过singletonObjects、earlySingletonObjects、singletonFactories获取单例对象，获取到则返回

```
singletonObjects //Cache of singleton objects: bean name to bean instance
earlySingletonObjects //Cache of early singleton objects: bean name to bean instance主要存放半成品的对象，用于解决循环引用的问题A->B,B->A
singletonFactories //Cache of singleton factories: bean name to ObjectFactory，有一个getObject方法，需要注意的是此处是ObjectFactory(函数式接口)，而不是FactoryBean，别弄错
```

    3. if (sharedInstance != null && args == null)

```
if (sharedInstance != null && args == null) {//args在主流程中一般为null
    beanInstance = getObjectForBeanInstance(sharedInstance, name, beanName, null);
}//如果if条件不满足，即getSingleton(beanName)不存在对应单例sharedInstance对象，则进入下面第4点部分

//getObjectForBeanInstance(beanInstance, name, beanName, RootBeanDefinition mbd)分析：
// name是可能带&的原始名称，beanName是规范化后的name，去除了&(如果有的话)
//1. BeanFactoryUtils.isFactoryDereference(name)
if (BeanFactoryUtils.isFactoryDereference(name)) {//name以&开头，说明获取的就是FactoryBean对象
```

```
    if (beanInstance instanceof NullBean) {
    return beanInstance;
    }
    if (!(beanInstance instanceof FactoryBean)) {
        throw new BeanIsNotAFactoryException(beanName,
beanInstance.getClass());
    }
    if (mbd != null) {
        mbd.isFactoryBean = true;
    }
    //不是NullBean，属于FactoryBean,name以&开头，说明是要获取
FactoryBean
    //所以直接返回beanInstance即可
    return beanInstance;
}
//2.如果不是FactoryBean，则直接返回该实例对象
if (!(beanInstance instanceof FactoryBean)) {
    return beanInstance;
}
//3.如果是FactoryBean，最终通过getObjectFromFactoryBean中的
factory.getObject获取对象
```

4. 第一次创建bean的过程

核心部分：

1、如果parent容器存在(默认不存在)且当前容器没有beanName对应的Definition，则去parent容器调用getBean()-like方法，返回

2、mbd = getMergedLocalBeanDefinition(beanName)；检测mbd.isAbstract(),若为false进入3，否则抛出异常

3、通过getBean(dep)先初始化beanName dpendsOn的 dep bean

//mbd.getDependsOn()获取dependsOn的Bean; Autowired的Field不会产生dependsOn的依赖关系；进入下一步前Guarantee initialization of beans that the current bean depends on；官方解释：不常用，当前bean不显式通过属性或者构造参数依赖另一个Bean，但依赖于另一个Bean的初始化时产生的“side effect”，可以使用该属性来强制指定初始化顺序，而且当前Bean会先于另一个Bean销毁；不是指我们平常说的循环引用，Spring通过3级缓存以及singletonsCurrentlyInCreation来实现循环引用的处理（setter不会出现问题，可以使用3级缓存中的半成品对象(允许EarlyReference)，构造函数可能还会出现问题(构造时就要求另一个对象初始化，而另一个对象构造时要求当前对象初始化，这种循环引用是无解的，必须手动处理)）；singletonsCurrentlyInCreation Set表示Names of beans that are currently in creation，构造函数造成的循环依赖最终会通过这个Set在beforeSingletonCreation(String beanName) 方法中检测出来；

//对于beanName Bean和 dep Bean，在通过getBean创建depBean之前，还会先通过registerDependentBean(dep, beanName)来注册dependsOn的依赖间关系，下面是其中一些关键数据结构

//dependentBeanMap 存放的是 Aname->Set<Bname> Aname是Bname的前置要求

//dependenciesForBeanMap 存放的是 Bname->Set<Aname> Aname是Bname的前置要求

//registerDependentBean(String beanName, String dependentBeanName) beanName是dependentBeanName的前置要求，这个会方法更新dependentBeanMap和dependenciesForBeanMap

//isDependent(String beanName, String dependentBeanName) beanName是否为dependentBeanName的前置要求，是返回true,不是则返回false

4、依据单例、原型(多例)还是其它情况(request:一次http请求有效，session:一次session有效)去创建bean，核心方法是createBean

单例情况：

单例创建sharedInstance=getSingleton(String beanName, ObjectFactory<?> singletonFactory)：

beforeSingletonCreation 在singletonsCurrentlyInCreation Set添加beanName,如果beanName已存在于Set，就说明检测到单例Bean的构造函数产生了循环依赖，抛出异常

createBean(beanName, mbd, args)//核心创建

afterSingletonCreation(beanName) 在singletonsCurrentlyInCreation Set移除beanName,如果移除的beanName不存在于Set，那该情况不合理抛出异常

addSingleton(beanName, singletonObject) 已创建完成，更新3级缓存，新建的对象此时才加入singletonObjects

```
beanInstance = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
```

多例情况：

```
beforePrototypeCreation(beanName);
prototypeInstance = createBean(beanName, mbd, args);
afterPrototypeCreation(beanName);
beanInstance = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
```

其它情况：类似多例情况，但多了一个Scope处理

5、registerDisposableBeanIfNecessary（容器销毁时调用）

上述过程中涉及的createBean解析：

1、依据beanDefinition解析相关的class；对beanDefinition执行prepareMethodOverrides（有利于后面加快方法解析速度）

2、resolveBeforeInstantiation：//使用后置处理器处理看是否返回代理对象,如果能就直接返回该对象

```
// Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.
Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
if (bean != null) {
    return bean;
}
```

其中resolveBeforeInstantiation方法内：

满足条件时，将执行下面语句

```
//InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation
bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
if (bean != null) {

//BeanPostProcessor.postProcessAfterInitialization
    bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
```

3、调用doCreateBean

1、createBeanInstance创建实例得到一个BeanWrapper对象，如果构造函数有依赖对象，会先初始化依赖对象

```
            2、
applyMergedBeanDefinitionPostProcessors(RootBeanDefinition mbd,
beanType, beanName)
                //2步骤Allow post-processors to modify bean
definition
            3、populateBean(beanName, RootBeanDefinition
mbd,BeanWrapper bw)：进行属性填充，属性注入的核心方法
                1、如果bw为null，要么放弃填充属性并返回，要么抛出异常；
如果bw不为null，进入下一步
                2、!mbd.isSynthetic() &&
hasInstantiationAwareBeanPostProcessors()时，
postProcessAfterInstantiation
                    //Perform operations after the bean has
been instantiated, via a constructor or factory method,
//but before Spring property population,This is the ideal
callback for performing custom                          //field
injection on the given bean instance
                    for (InstantiationAwareBeanPostProcessor bp
: getBeanPostProcessorCache().instantiationAware) {
                        if
(!bp.postProcessAfterInstantiation(bw.getWrappedInstance(),
beanName)) {
                            return;
                        }
                    }
                3、PropertyValues pvs = (mbd.hasPropertyValues()
? mbd.getPropertyValues() : null);
                  //pvs是将来属性注入的依据；
                  int resolvedAutowireMode =
mbd.getResolvedAutowireMode()
                  //resolvedAutowireMode通常是AUTOWIRE_NO，即此处
不进入if执行
                  //此处个人理解是所有属性都按名称注入或类型注入处理
()pvs，好像是很少会使用的,调试暂未发现经过此处的情形
                  if (resolvedAutowireMode == AUTOWIRE_BY_NAME
|| resolvedAutowireMode == AUTOWIRE_BY_TYPE) {
                      //依据resolvedAutowireMode类型执行
autowireByName，autowireByType
                      //autowireByName核心:Object bean =
getBean(propertyName);   pvs.add(propertyName, bean);
                      //autowireByType核心：
                      //autowiredArgument =
resolveDependency(desc, beanName, autowiredBeanNames,
converter);
                      // if (autowiredArgument != null)
{pvs.add(propertyName, autowiredArgument); }
                  }
                4、
InstantiationAwareBeanPostProcessor.postProcessPropertyValues
//后置处理，注入对象的核心过程
                    for (InstantiationAwareBeanPostProcessor bp
: getBeanPostProcessorCache().instantiationAware) {
                        PropertyValues pvsToUse =
bp.postProcessProperties(pvs, bw.getWrappedInstance(),
beanName);
                        if (pvsToUse == null) {
                            if (filteredPds == null) {
```

```
                                filteredPds =
filterPropertyDescriptorsForDependencyCheck(bw,
mbd.allowCaching);
                            }
                            pvsToUse =
bp.postProcessPropertyValues(pvs, filteredPds,
bw.getWrappedInstance(),

     beanName);
                            if (pvsToUse == null) {
                                return;
                            }
                        }
                        pvs = pvsToUse;
                    }//此处默认有3个相关后置处理器，分别是
ConfigurationClassPostProcessor.ImportAwareBeanPostProcessor、
                    //CommonAnnotationBeanPostProcessor、
AutowiredAnnotationBeanPostProcessor;其中注入过程主要后两个起
                    //作用，对于Spring Autowired注解来说，最后一个后置
处理器才最重要，会注入对象（没有的对象会先创建再注入，递归处理）
                    //上面的pvs即使在A中注入了一个B的情况下调试时仍然数
量为0，所以pvs具体起什么作用暂未发现，其不是一般意义上的属性含义
                5、applyPropertyValues(beanName, mbd, bw, pvs);

                    //pvs中为0时，此处applyPropertyValues通常直接返回
if (pvs.isEmpty()) {return;}，其后续不常见过程暂未分析，
                    //但主要就是和pvs相关
            4、initializeBean(beanName,bean,RootBeanDefinition
mbd)
                1、invokeAwareMethods调用Aware相关的set方法
                2、applyBeanPostProcessorsBeforeInitialization//
调用初始化前的后置处理器方法
                    for (BeanPostProcessor processor :
getBeanPostProcessors()) {
                        Object current =
processor.postProcessBeforeInitialization(result, beanName);
                        if (current == null) {
                            return result;
                        }
                        result = current;
                    }
                3、invokeInitMethods//调用初始化方法
                4、applyBeanPostProcessorsAfterInitialization//
调用初始化之后的后置处理器方法
                    for (BeanPostProcessor processor :
getBeanPostProcessors()) {
                        Object current =
processor.postProcessAfterInitialization(result, beanName);
                        if (current == null) {
                            return result;
                        }
                        result = current;
                    }
            5、registerDisposableBeanIfNecessary（容器销毁时调用）

对上面createBeanInstance创建实例进一步分析
假定One和Two构造函数没有依赖时，默认先实例化One再实例化Two
```

给One加一个构造函数，入参是Two类型对象，并加上Autowired，那么由于有依赖关系，会先实例化Two，再实例化One

```
getBean  （One）
    doGetBean
        getSingleton
            createBean
                doCreateBean
                    createBeanInstance  （One）

 determineConstructorsFromBeanPostProcessors//会用到AutowiredAnnotationBeanPostProcessor
                        autowireConstructor //实际交由ConstructorResolver(beanFactory).autowireConstructor执行

 ConstructorResolver.createArgumentArray

 ConstructorResolver.resolveAutowiredArgument

beanFactory.resolveDependency

beanFactory.doResolveDependency

descriptor.resolveCandidate
                                                    getBean  （此时Two）
```

解析自动装配参数后，通过getBean先来创建其它依赖对象
AutowiredAnnotationBeanPostProcessor干预了构造函数选择的过程，之后由ConstructorResolver来处理之后的参数解析与对象实例化

对上面populateBean注入属性进一步分析
COne中存在Field属性CTwo，并且标注了@Autowired

```
getBean  （COne）
    ....
        populateBean  （COne）
            postProcessProperties
 //AutowiredAnnotationBeanPostProcessor.postProcessProperties
                InjectionMetadata metadata = findAutowiringMetadata(beanName, bean.getClass(), pvs)
                metadata.inject(bean, beanName, pvs)
                    InjectedElement element.inject(target, beanName, pvs)//此处element实际是AutowiredFieldElement
                        AutowiredFieldElement实例方法resolveFieldValue(field, bean, beanName)
                        beanFactory.resolveDependency(desc, beanName, autowiredBeanNames, typeConverter)
                            beanFactory.doResolveDependency
                                descriptor.resolveCandidate
                                    getBean  （此时CTwo)
```

解析自动装配参数后，通过getBean先来创建其它依赖对象
AutowiredAnnotationBeanPostProcessor干预了Field属性注入过程

CCOne中存在方法injectMethod(Three three),且该方法被标注了@Autowired

```
getBean  （CCOne）
    ....
        populateBean  （CCOne）
            postProcessProperties
 //AutowiredAnnotationBeanPostProcessor.postProcessProperties
                InjectionMetadata metadata = findAutowiringMetadata(beanName, bean.getClass(), pvs)
```

```
                metadata.inject(bean, beanName, pvs)
                    InjectedElement element.inject(target,
beanName, pvs)//此处element实际是AutowiredMethodElement
                        arguments =
resolveMethodArguments(method, bean,
beanName)//AutowiredMethodElement中方法

beanFactory.resolveDependency(currDesc, beanName,
autowiredBeans, typeConverter)
                                beanFactory.doResolveDependency
                                    descriptor.resolveCandidate
                                        beanFactory.getBean
(此时是Three)
                        element.member.invoke(bean,
arguments)//member即method对象，进入java反射方法调用阶段
AutowiredAnnotationBeanPostProcessor干预了方法参数注入过程

上面过程中一个InjectionMetadata中可以有多个InjectedElement（比如多个属性注入等），但上面仅仅是单注入处理
解析上面的findAutowiringMetadata(String beanName, Class<?> clazz,
@Nullable PropertyValues pvs)
1、先从injectionMetadataCache获取，能获取到则返回InjectionMetadata;否则进入2
2、通过buildAutowiringMetadata(clazz)构建元数据InjectionMetadata并加入injectionMetadataCache缓存
   //buildAutowiringMetadata过程：
   //判断!AnnotationUtils.isCandidateClass(clazz,
this.autowiredAnnotationTypes)，检测clazz是否含有待处理注解，有就往下执行
   //先解析本地doWithLocalFields  currElements.add(new
AutowiredFieldElement(field, required))
   //再解析本地doWithLocalMethods currElements.add(new
AutowiredMethodElement(method, required, pd));
   //将上面解析加入集合elements.addAll(0, currElements); //
   //targetClass = targetClass.getSuperclass(); 循环处理父类
   //由于elements中每次都加入在位置0，所以最终父类注入数据在最前面，子类注入数据在后面，且每一层类的注入数据都是属性注入在前，方法注入在后
   //这里将注解解析成了元数据并缓存，后续处理只需要基于元数据进行处理了

DependencyDescriptor是对Field或者MethodParameter的封装，应该是为了方便使用
上面过程可以发现
resolveAutowiredArgument/resolveFieldValue/resolveMethodArguments最终将解析依赖的委托给了beanFactory来做
beanFactory.resolveDependency进一步解析
    //multipleBeans = resolveMultipleBeans(descriptor,
beanName, autowiredBeanNames, typeConverter)
    //  先通过resolveMultipleBeans尝试按照
StreamDependencyDescriptor/isArray/Collection/Map来进行“集合”注入，满足就返回
    //  这步失败后，后续操作都是单Bean注入尝试
    //matchingBeans = findAutowireCandidates(beanName, type,
descriptor)    找到所有候选Bean
    //matchingBeans数量大于1个，则
determineAutowireCandidate(matchingBeans, descriptor)依据
@Primary和@Priority确定候选
    //matchingBeans数量等于1，采用matchBean信息，通过
descriptor.resolveCandidate获取实例
```

```
    //返回确定的候选Bean
```

5. adaptBeanInstance(name, beanInstance, requiredType)

```
adaptBeanInstance(String name, Object bean, @Nullable Class<?>
requiredType) :
    //检查bean是否可转换为指定的requiredType，如果可以直接返回；
    //如果不可以，尝试通过typeConverter进行转换，转换失败抛出异常;成功则
返回；
    // Check if required type matches the type of the actual
bean instance.
    if (requiredType != null && !requiredType.isInstance(bean))
{
        try {
            Object convertedBean =
getTypeConverter().convertIfNecessary(bean, requiredType);
            if (convertedBean == null) {
                throw new BeanNotOfRequiredTypeException(name,
requiredType, bean.getClass());
            }
            return (T) convertedBean;
        }
        catch (TypeMismatchException ex) {
            if (logger.isTraceEnabled()) {
                logger.trace("Failed to convert bean '" + name
+ "' to required type '" +

ClassUtils.getQualifiedName(requiredType) + "'", ex);
            }
            throw new BeanNotOfRequiredTypeException(name,
requiredType, bean.getClass());
        }
    }
    return (T) bean;
```

12. **finishRefresh**

```
protected void finishRefresh() {
        // Clear context-level resource caches (such as ASM metadata from
scanning).
        clearResourceCaches();
        // Initialize lifecycle processor for this context.
        initLifecycleProcessor();
        // Propagate refresh to lifecycle processor first.
        getLifecycleProcessor().onRefresh();
        // Publish the final event.
        publishEvent(new ContextRefreshedEvent(this));
        // Participate in LiveBeansView MBean, if active.
        if (!NativeDetector.inNativeImage()) {
            LiveBeansView.registerApplicationContext(this);
        }
    }
```

AOP

代理模式

静态代理

代理类在程序运行前就存在，这种代理方式称为静态代理；通常代理类和增强类都**实现同一个接口或是继承自相同的父类**；该方式简单直接，但维护麻烦

```
//此处仅演示增强类和实现类都是同一个接口的代理方式；继承的实现方式类似
new PSerProxy(new PSerImpl()).service();

public interface IPSer {
    void service();
}
public class PSerImpl implements IPSer{
    @Override
    public void service() {
        System.out.println("PSerImpl service");
    }
}
public class PSerProxy implements IPSer{
    IPSer ipSer;
    public PSerProxy(IPSer ipSer){
        this.ipSer = ipSer;
    }
    @Override
    public void service() {
        System.out.println("PSerProxy step1");
        ipSer.service();
        System.out.println("PserProxy step2");
    }
}
Output:
PSerProxy step1
PSerImpl service
PserProxy step2
```

动态代理

代理类在程序运行时创建的方式称为动态代理，在运行前并未在Java代码中定义，而是根据JAVA代码的“指示”动态生成

JDK动态代理

> 该代理方式要求被代理类必须实现接口，有一定的局限性;生成的类继承了Proxy，并实现了相关的接口

```
public class JDKProxy {
    public static void proxytest(){
        PSerImpl pSer = new PSerImpl();
        IPSer proxyInstance = (IPSer)
Proxy.newProxyInstance(pSer.getClass().getClassLoader(),
pSer.getClass().getInterfaces(), new InvocationHandler() {
            @Override
```

```
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                System.out.println("Jdkproxy invoke step1");
                System.out.println(proxy.getClass());
                System.out.println(method);
                System.out.println(args);
                method.invoke(pSer,args);
                System.out.println("Jdkproxy invoke step2");
                return null;
            }
        });
        proxyInstance.service();
    }
}
Output:
Jdkproxy invoke step1
class com.sun.proxy.$Proxy0
public abstract void com.example.aop.IPSer.service()
null
PSerImpl service
Jdkproxy invoke step2
```

CGLIB代理

> CGLIB底层使用ASM(字节码操作框架)来操作字节码生成新的类，让生成的代理类继承被代理类，并在代理类中对代理方法进行强化处理，因此CGLIB可以支持没有实现接口的类(但被代理类被final关键字所修饰时会代理失败，因为不能在子类被重写)

```
public class CglibProxy {
    public static void cglibproxytest(){
        PSerImpl cglibProxyObj = (PSerImpl)
CglibProxyFactory.createCglibProxyObj(new PSerImpl());
        cglibProxyObj.service();
    }
    static class CglibProxyFactory {
        public static Object createCglibProxyObj(Object obj){
            Enhancer enhancer = new Enhancer();
            //为生成的代理类指定父类
            enhancer.setSuperclass(obj.getClass());
            //设置回调：代理类上的所有方法，都会调用ProxyMethodInterceptor.interpcept方
法
            enhancer.setCallback(new ProxyMethodInterceptor(obj));
            return enhancer.create();
        }
        static class ProxyMethodInterceptor implements MethodInterceptor{
            private Object obj;
            public ProxyMethodInterceptor(Object obj){
                this.obj = obj;
            }
            /**
             * @param proxyObj  代理对象
             * @param method    被代理原始方法方法,使用它必须时必须维护一个原始对象的引用
             * @param args      方法入参
             * @param methodProxy   CGLIB增强后的方法
             * 从理解上来说，method可以理解为原始方法，希望接受的参数自然是原始对象，因此需要
维护一个原始对象的引用
```

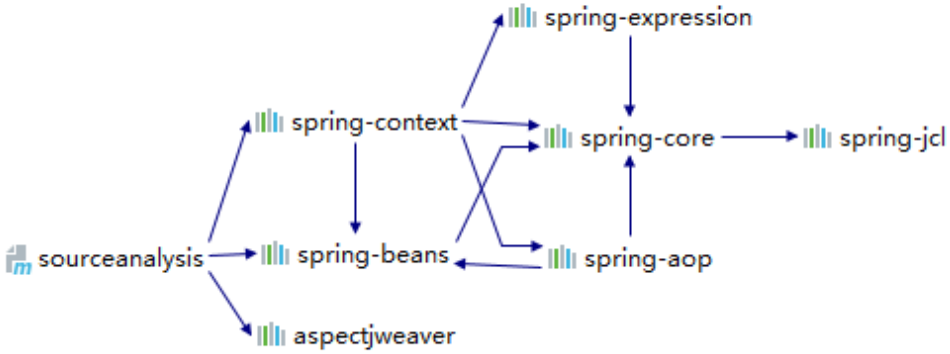
```
             *            methodProxy可以理解为增强类的方法，它期望接受的对象自然是增强对象，通过methodProxy.invokeSuper(proxyObj,args)也能达到调用原始方法的目的
             *                        注意：methodProxy不能调用methodProxy.invoke(proxyObj,args)，会造成再次调用intercept，形成递归
             */
            @Override
            public Object intercept(Object proxyObj, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
                System.out.println("MethodInterceptor intercept step1");
                System.out.println(proxyObj.getClass());
                System.out.println(method);
                System.out.println(args.length);
                System.out.println(methodProxy);
                method.invoke(this.obj,args);//正确的调用原始method    方法1
                //method.invoke(proxyObj,args);//错误调用，会形成递归
                methodProxy.invokeSuper(proxyObj,args);//正确的调用原始method    方法2  使用它可以不用维护原始对象的引用
                //methodProxy.invoke(proxyObj,args);//错误调用，会形成递归
                System.out.println("MethodInterceptor intercept step2");
                return null;
            }
        }
    }
}
Output:
MethodInterceptor intercept step1
class com.example.aop.PSerImpl$$EnhancerByCGLIB$$16a9ce5a
public void com.example.aop.PSerImpl.service()
0
org.springframework.cglib.proxy.MethodProxy@51521cc1
PSerImpl service
PSerImpl service
MethodInterceptor intercept step2
```

SpringAOP

> writing custom aspects by using @AspectJ annotation style, and use of the AspectJ pointcut language, 但是依旧使用SpringAOP进行weaving织入

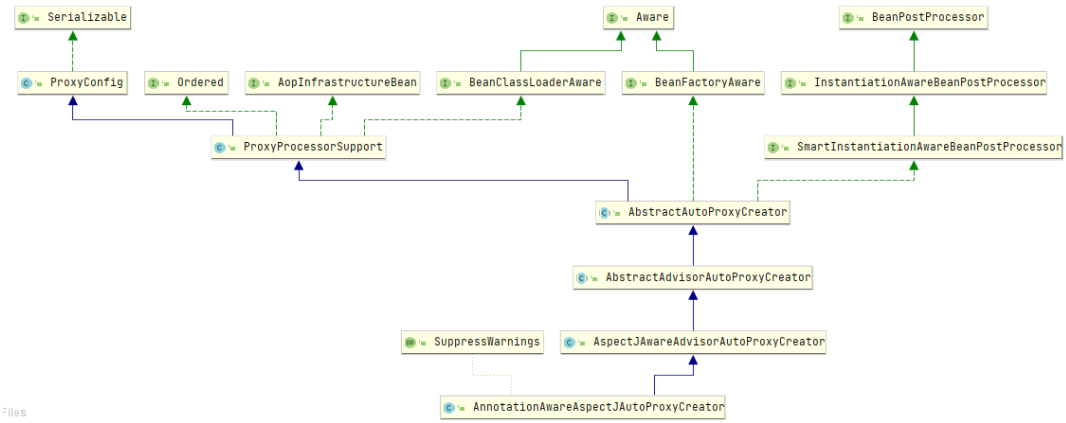
- 依赖：基于SpringBoot使用SpringAOP，仅需引入spring-context和aspectjweaver即可，就能使用基于@AspectJ注解风格的SpringAOP
    - 不引入aspectjweaver，就无法使用@AspectJ/@Pointcut/@Before/@After/@AfterThrowing/@Around等注解

      > 经过测试即使不引入aspecjweaver，只引入spring-context，mainConfig配置类也被CGLIB增强，说明aspecjweaver仅仅让框架支持了注解形式的AOP，但底层AOP机制并不来自aspectjweaver

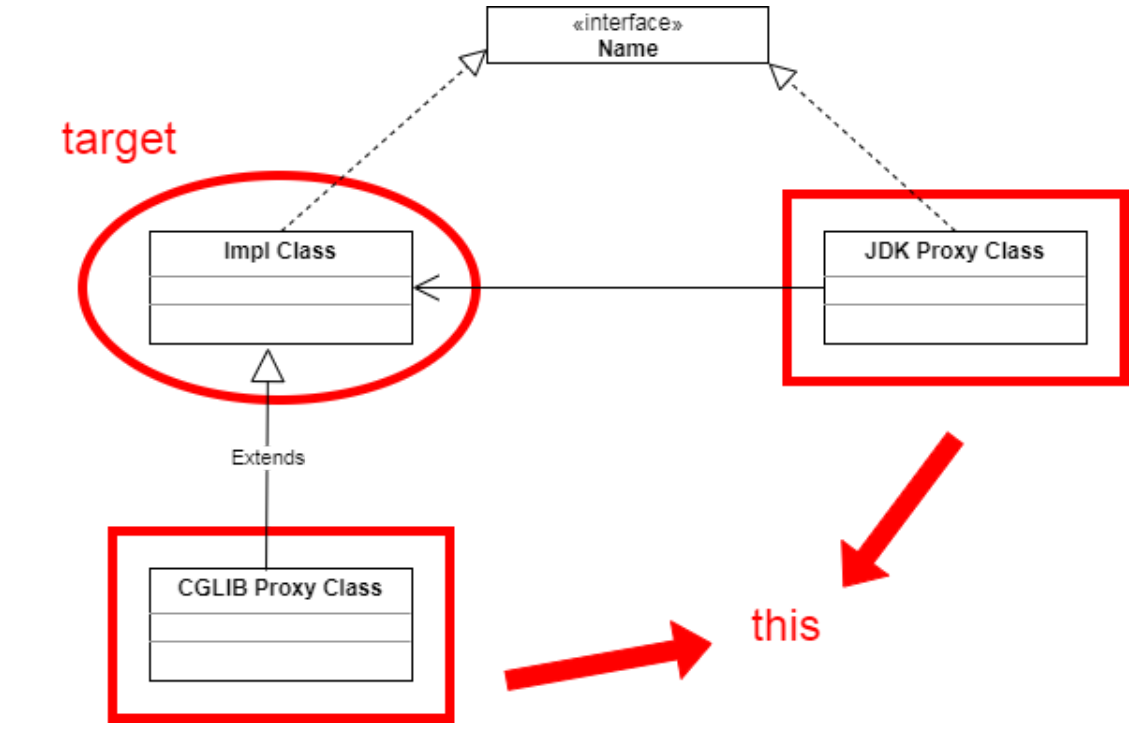


- 为了能够处理前面引入的@AspectJ风格注解，需要在@Configuration标注的类上，同时加上@EnableAspectJAutoProxy

```
@Configuration
//proxyTargetClass=true表示在Spring框架中强制使用CGLIB实现代理；否则，JDK/CGLIB代理都可能使用
//注意SpringBoot2.x中，默认使用CGLIB，使用的代理类为AopAutoConfiguration进行配置；此处spring框架中默认还是使用的AnnotationAwareAspectJAutoProxyCreator；
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class AppConfig {
    // ...
}
@EnableAspectJAutoProxy applies to its local application context only, allowing for selective proxying of beans at different levels. //表明父子容器可以分别设置是否代理
//@AspectJ风格的注解标注的类，必须被Spring管理，否则无法代理（即必须被同时标注了@Component等注解）
@EnableAspectJAutoProxy引入了org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator,是AOP核心类
```



- 切入点表达式this和target的区别



切入例子（来源：http://www.bianchengquan.com/article/615900.html）

PointCut	JDK Proxy	CGLIB Proxy
this(aop.service.AnimalService)	成功切入	成功切入
this(aop.serviceimpl.DogService)	失败	成功切入
target(aop.service.AnimalService)	成功切入	成功切入
target(aop.serviceimpl.DogService)	成功切入	成功切入

其中DogService 实现了AnimalService接口；JDKProxyClass extends Proxy implements AnimalService；this是AnimalService，但是在JDKProxy时，this是Proxy不是DogService因此切入失败；target是AnimalService也是DogService，所以都能切入成功；this/target(Type)提供了两种视角去过滤，表示this/target instanceof Type 为true的连接点；Aspect切面默认是“”单例，另外支持两种多例切面@Aspect(perthis)/@Aspect(pertarget)，他们会为每一个符合this/target条件的类都创建一个切面实例(主要用于维持某种状态，比如计数，每个代理对象调用方法时，不同的切面维护不同的计数)，同时要在切面类上指明Scope为prototype，否则Spring启动会报错；

- 机制
  - 底层使用JDK动态代理（有个问题就是@Autowired只能标注在接口而不能在实现类型上，因为动态代理出来的对象无法赋值，这也是SpringBoot改用默认CGLIB的一部分原因）或者CGLIB代理
  - 如果被代理的对象实现了接口可以通过JDK代理实现，那么JDK代理会被使用，如果被代理的对象没有实现任何接口，那么CGLIB代理会被使用；这个过程可以通过定义配置强制都使用CGLIB代理
  - 从spring4.0开始，被代理对象的构造函数不会再调用两次；但如果你的JVM不允许构造函数bypassing绕过，你可能还会看到两次调用
- Example1

```
@Component
public class Calc {

    public Calc(){
```

```
        System.out.println("Calc created");
    }
    public int add(int a, int b) {
        System.out.println(a+b);
        return a+b;
    }
    public int div(int a, int b) {
        System.out.println(a/b);
        return a/b;
    }
}
@Aspect//在一个切面中允许多次使用@Before等注解，存在优先顺序;默认为“”单例，其它的还有perthis（SpringAOP通常指代理对象）/pertarget(SpringAOP中通常指被代理对象)用于多例有状态的情况
@Component
public class LogAspect {
    public LogAspect() {
        System.out.println("LogAspect created");
    }
    @Pointcut("execution(* com.example.aop.Calc.*(..))")
    public void pointcut(){}
    @Before("pointcut()")//不能控制是否进入被代理的方法执行，默认始终执行
    public void BeforeLog(){
        System.out.println("LogAspect......Before");
    }
    @After("pointcut()")
    public void AfterLog(){
        System.out.println("LogAspect......After");
    }
    @AfterReturning("pointcut()")
    public void AfterReturingLog(){
        System.out.println("LogAspect......AfterReturning");
    }
    @AfterThrowing("pointcut()")
    public void AfterThrowingLog(){
        System.out.println("LogAspect......AfterThrowing");
    }
//    @Around("pointcut()")//可以控制是否进入被代理的方法执行；可以代理前四种;Around要求advice的返回值和被代理方法返回值一致；
//    public int AroundLog(ProceedingJoinPoint proceedingJoinPoint){
//        Object obj = null;
//        try {
//            System.out.println("LogAspect......AroundBefore");
//            obj=proceedingJoinPoint.proceed(proceedingJoinPoint.getArgs());//proceed时会依旧触发前面几种切入点
//            System.out.println("LogAspect......AroundAfter");
//        } catch (Throwable throwable) {
//            //throwable.printStackTrace();
//        }finally {
//            System.out.println("LogAspect......Aroundfinally");
//        }
//        System.out.println("LogAspect......AroundfinallyAfter");
//        return ((Integer) obj).intValue();
//    }
}
Output:
calc.add(1,2):
```

```
LogAspect......Before
3
LogAspect......AfterReturning
LogAspect......After

calc.div(1,0):
LogAspect......Before
LogAspect......AfterThrowing
LogAspect......After

当启用@Around时:
calc.add(1,2):
LogAspect......AroundBefore
LogAspect......Before
3
LogAspect......AfterReturning
LogAspect......After
LogAspect......AroundAfter
LogAspect......Aroundfinally
LogAspect......AroundfinallyAfter

calc.div(1,0):
LogAspect......AroundBefore
LogAspect......Before
LogAspect......AfterThrowing
LogAspect......After
LogAspect......Aroundfinally
LogAspect......AroundfinallyAfter
```

- Example2 (pertarget)

```
public class CSuper {
}
@Component
public class Calc2 extends CSuper{
    public Calc2(){
        System.out.println("Calc2 created");
    }
    public int add(int a, int b) {
        System.out.println(a+b);
        return a+b;
    }
    public int minus(int a, int b) {
        System.out.println(a-b);
        return a-b;
    }
}
@Component
public class Calc3 extends CSuper{
    public Calc3(){
        System.out.println("Calc3 created");
    }
    public int add(int a, int b) {
        System.out.println(a+b);
        return a+b;
    }
    public int multiple(int a, int b) {
```

```
        System.out.println(a*b);
        return a*b;
    }
}
@Aspect("pertarget(target(com.example.aop.CSuper))")
@Component
@Scope("prototype")
public class LogAspect2 {
    int count = 0;
    public LogAspect2() {
        System.out.println("LogAspect2 created");
    }
    @Pointcut("target(com.example.aop.CSuper)")
    public void pointcut(){}
    @Before("pointcut()")
    public void BeforeLog(){
        System.out.println("LogAspect......Before:"+count++);
    }
    @After("pointcut()")
    public void AfterLog(){
        System.out.println("LogAspect......After:"+count++);
    }
}
Calc2 calc2 = context.getBean("calc2", Calc2.class);
calc2.add(1,1);
System.out.println("------------------");
Calc3 calc3 = context.getBean("calc3", Calc3.class);
calc3.add(1,1);
calc3.multiple(1,1);
System.out.println("------------------");
Output:
LogAspect2 created
LogAspect......Before:0
2
LogAspect......After:1
------------------
LogAspect2 created
LogAspect......Before:0
2
LogAspect......After:1
LogAspect......Before:2
1
LogAspect......After:3
------------------
```

流程分析

- AnnotationAwareAspectJAutoProxyCreator是AOP过程中的核心类

```
//Any AspectJ annotated classes will automatically be recognized, and their advice applied if Spring AOP's proxy-based model is capable of applying it. This covers method execution joinpoints.
```

  - SpringAOP代理时机

```
getBean  （Calc）
```

```
    doGetBean
        getSingleton
            createBean//此处实际是getSingleton中执行singletonObject =
singletonFactory.getObject()内部执行
                doCreateBean
                    instanceWrapper = createBeanInstance(beanName, mbd,
args)
                    bean = instanceWrapper.getWrappedInstance()
                    exposedObject = bean
                    populateBean(beanName, mbd, instanceWrapper) //填充的
是代理前的Bean对象
                    exposedObject = initializeBean(beanName,
exposedObject, mbd) //exposedObject最后会变成代理对象
                        wrappedBean = exposedObject
                        wrappedBean =
applyBeanPostProcessorsBeforeInitialization(wrappedBean,
beanName)//wrappedBean不变
                        invokeInitMethods(beanName, wrappedBean, mbd)//
调用初始化方法
                        //下面产生效果的PostProcessor是
AnnotationAwareAspectJAutoProxyCreator.postProcessAfterInitialization
                        wrappedBean =
applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName)
                            return wrapIfNecessary(bean, beanName,
cacheKey)
                                specificInterceptors =
getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null)
                                proxy = createProxy(bean.getClass(),
beanName, specificInterceptors,
                                                    new
SingletonTargetSource(bean))//返回代理对象
                                return proxy
                        return wrappedBean
            addSingleton(beanName, singletonObject)//将singletonObject注
册到工厂，此时singletonObject已是代理对象


目前调试基本上都是此种方式产生AOP代理


下面是另外一种产生AOP代理的情况，和自定义TargetSource有关，暂时未在实践中遇到过
getBean  （Calc）
    doGetBean
        getSingleton
            createBean
                bean = resolveBeforeInstantiation(beanName, mbdToUse)
                    bean =
applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
                        result =
bp.postProcessBeforeInstantiation(beanClass, beanName)
                            targetSource =
getCustomTargetSource(beanClass, beanName)
                            specificInterceptors =
getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource)
                            proxy = createProxy(beanClass, beanName,
specificInterceptors, targetSource)//返回代理对象
                            return proxy
                        if (result != null) {return result;}
```

```
                    if (bean != null) {
                        bean =
applyBeanPostProcessorsAfterInitialization(bean, beanName);
                    }
                if (bean != null) {return bean;}
```

@Autowied Field时的AOP过程和分析Spring时属性注入没什么区别，都是在populateBean时发生，因为都是通过getBean进行递归处理；其它位置应该也类似（因为都是通过getBean获取的，因此推测都如此）
在循环依赖时，有一个关键方法SmartInstantiationAwareBeanPostProcessor.getEarlyBeanReference(Object bean, String beanName)（主要发生在getBean调用getSingleton尝试从缓存获取对象的时候，通过ObjectFactory缓存获取），默认有两个实现类，一个是AnnotationAwareAspectJAutoProxyCreator，其主要执行wrapIfNecessary(bean, beanName, cacheKey)，另外还会通过getSingleton对于earlySingletonExposure进行额外处理，修正exposedObject引用；另一个是AutowiredAnnotationBeanPostProcessor，它直接返回参数bean（这个参数来自ObjectFactory,这是前面匿名表达式形成的一个闭包中记录的bean）；
然后这里有个看起来很奇怪的地方（但是是正常的），就是整个代理过程中会有两种对象，一种是原始的target，一种是proxy对象，假设分1,2两个对象并循环引用（这里用循环引用举例，非循环引用时效果是一样的），那么就会产生target1(引用了proxy2),proxy1(继承target1且对应存放target2的字段为null)，target2(引用了proxy1),proxy2(继承target2且对应存放target1的字段为null)，会有四个对象，proxy = createProxy(bean.getClass(),beanName,specificInterceptors, new SingletonTargetSource(bean))通过bean(即target)和proxy进行关联，直接看容器中的代理对象对应的target字段是为null，这点特别需要注意；虽然如此，但是proxy对象还是间接持有了target对象，从调试角度看的话有CGLIB$CALLBACK_num大部分都能间接访问target

- Advice之间存在PartialOrder(偏序)排序关系，默认排序器为AspectJPrecedenceComparator

```
//Orders AspectJ advice/advisors by precedence (not invocation order).
//Given two pieces of advice, A and B:
//If A and B are defined in different aspects, then the advice in the aspect
with the lowest order value has the highest precedence.
//If A and B are defined in the same aspect, if one of A or B is a form of
after advice, then the advice declared last in the aspect has the highest
precedence. If neither A nor B is a form of after advice, then the advice
declared first in the aspect has the highest precedence.
```

- BeanFactoryAspectJAdvisorsBuilder.buildAspectJAdvisors()

```
//Look for AspectJ-annotated aspect beans in the current bean factory, and return to a list of Spring AOP Advisors representing them
其使用了
ReflectiveAspectJAdvisorFactory.getAdvisors(MetadataAwareAspectInstanceFactory aspectInstanceFactory)
另外BeanFactoryAspectInstanceFactory implements MetadataAwareAspectInstanceFactory是getAdvisors的参数//创建Aspect的工厂，public Object getAspectInstance() ，其本身仅代表一种类型的切面；但是配合循环所有beanName时，就能获取到所有Advisors
AspectMetadata  //Metadata for an AspectJ aspect class, with an additional Spring AOP pointcut for the per clause.
new InstantiationModelAwarePointcutAdvisorImpl(expressionPointcut（切入点）, candidateAdviceMethod（增强方法）,
                this（advisorFactory）, aspectInstanceFactory, declarationOrderInAspect, aspectName)//advisor实际是切入点+增强方法
```

- Pointcut实际是类过滤器和方法过滤器的组合

```
public interface Pointcut {
    /**
     * Return the ClassFilter for this pointcut.
     * @return the ClassFilter (never {@code null})
     */
    ClassFilter getClassFilter();
    /**
     * Return the MethodMatcher for this pointcut.
     * @return the MethodMatcher (never {@code null})
     */
    MethodMatcher getMethodMatcher();
}
```

AspectJ

- 通常需要编写aop.xml
- 通常需要运行时指定aspectjweaver.jar为javaagent
- 需要时再详细分析

SpringAOP&AspectJ区别

1. 目标：SpringAOP只提供一个不完备的实现，只被用于SpringIOC容器管理的Bean，处理常见情况; AspectJ提供一个完备的AOP实现，除了spring也可用于别的地方
2. 织入时机：
    - SpringAOP动态时织入（使用JDK/CGLIB进行动态代理）
    - Aspectj发生在编译时或者类加载时织入
        - Compile-time weaving:编译期织入，编译器将切面和应用源码便已在一个文件中
        - Post-compile weaving:编译后织入，也称二进制织入，将已有的字节码文件与切面编制在一起
        - Load-time weaving:加载时织入，与编译后织入一样，只是织入时间推迟到类加载到JVM时期

- SpringAOP仅发生在运行期间；AspectJ在运行期间不做任何事情，仅在编译和加载时发生，引入了AspectJ compiler(ajc)，无需任何设计模式
  - SpringAOP编译和织入过程都很简单，无需在常规编译过程中引入额外的东西
  - AspectJ需要引入ajc然后重编译我们的库（除非切换到post-compile or load-time weaving），这个过程比SpringAOP更复杂，我们要引入AspectJ的Java工具，包括a compiler(ajc)，a debugger(ajdb)，a documentation generator(ajdoc)，a program structure browser(ajbrowser)等，我们需要集成在IDE或者构建工具中

3. 连接点：
   - SpringAOP基于动态代理，不能处理final修饰的方法（运行时报错）
   - AspectJ直接在真实代码中织入了代理代码，它不需要子类实现，也能支持许多其它连接点

Joinpoint	SpringAOP Supported	AspectJ Supported
Method Call	No	Yes
Method Execution	Yes	Yes
Constructor Call	No	Yes
Constructor Execution	No	Yes
Static initializer execution	No	Yes
Object initialization	No	Yes
Field reference	No	Yes
Field assignment	No	Yes
Handler execution	No	Yes
Advice execution	No	Yes

4. 性能：一般而言编译时织入比运行时织入更快，基准测试中，AspectJ一般比SpringAOP快8-35倍

SpringBoot

- SpringBoot可以以jar包形式独立运行
- 可以内嵌Servlet容器，如Tomcat、Jetty或者Undertow
- **提供starter简化配置**
- 提供大量默认的**自动配置**，简化开发
- 提供产品级功能，如可集成**应用监控Actuator**等
- 无代码生成过程以及可无需xml配置

整体基于约定由于配置的思想

最佳实践

代码结构

- 不要使用“default package”

```
//When a class does not include a package declaration, it is considered to be in the “default package”.The use of the “default package” is generally discouraged and should be avoided. It can causeparticular problems for Spring Boot applications that use the @ComponentScan, @ConfigurationPropertiesScan, @EntityScan, or @SpringBootApplication annotations, since every classfrom every jar is read.
```

- 将main class放置在root package

```
//We generally recommend that you locate your main application class in a root package above otherclasses. The @SpringBootApplication annotation is often placed on your main class, and it implicitlydefines a base “search package” for certain items.

The main class file would declare the main method, along with the basic@SpringBootApplication, as follows
   @SpringBootApplicationpublic
   class Application {
       public static void main(String[] args) {
           SpringApplication.run(Application.class, args);
       }
   }
```

Configuration Class

Spring Boot favors Java-based configuration. Although it is possible to use SpringApplication withXML sources, we generally recommend that your primary source be a single @Configuration class.Searching for **Enable* annotations** can be a good starting point.//SpringBoot支持Spring应用的XML配置，但推荐使用@Configuration配置类，搜索使用Enable*注解是开启某种注解配置的好的开始

- 导入额外的@Configuration配置类

```
//You need not put all your @Configuration into a single class. The @Import annotation can be used toimport additional configuration classes. Alternatively, you can use @ComponentScan to automaticallypick up all Spring components, including @Configuration classes.
```

- 导入XML配置

```
//If you absolutely must use XML based configuration, we recommend that you still start with a@Configuration class. You can then use an @ImportResource annotation to load XML configurationfiles
```

自动配置

Spring Boot auto-configuration attempts to automatically configure your Spring application basedon the jar dependencies that you have added. ou need to opt-in(选择性加入) to auto-configuration by adding the @EnableAutoConfiguration or@SpringBootApplication annotations to one of your @Configuration classes.//SpringBoot依据所添加的jar进行自动配置(判断有无某种依赖进而是否启用某种功能)，需要手动加入 @EnableAutoConfiguration or @SpringBootApplication注解

- 替代自动配置组件

```
//Auto-configuration is non-invasive(非侵入的). At any point, you can start to define your own configuration to replace specific parts of the auto-configuration. For example, if you add your own DataSource bean,the default embedded database support backs away.//自动配置是非侵入的，你可以定义自己的配置组件来替代自动配置的某一部分组件
//If you need to find out what auto-configuration is currently being applied, and why, start yourapplication with the --debug switch. Doing so enables debug logs for a selection of core loggers and logs a conditions report to the console.//可以通过--debug选项启动springboot，这样可以观察到哪些自动配置的组件被应用及其原因
```

- 禁用自动配置组件

```
//If you find that specific auto-configuration classes that you do not want are being applied, you canuse the exclude attribute of @SpringBootApplication to disable them:@SpringBootApplication(exclude= {DataSourceAutoConfiguration.class})
//If the class is not on the classpath, you can use the excludeName attribute of the annotation andspecify the fully qualified name instead. If you prefer to use @EnableAutoConfiguration rather than @SpringBootApplication, exclude and excludeName are also available. Finally, you can also control thelist of auto-configuration classes to exclude by using the spring.autoconfigure.exclude property
//三种方式禁用组件@SpringBootApplication(exclude/excludeName)、@EnableAutoConfiguration(exclude/excludeName)、属性spring.autoconfigure.exclude
//Even though auto-configuration classes are public, the only aspect of the class thatis considered public API is the name of the class which can be used for disablingthe auto-configuration. 仅推荐使用类的名称，而不建议使用类内部的配置类或bean等，即使是公开的
```

Bean和依赖注入

You are free to use any of the standard Spring Framework techniques to define your beans andtheir injected dependencies.按照最佳实践的方式建立代码结构后， All of your application components (@Component, @Service, @Repository, @Controller etc.) are automatically registered as Spring Beans

使用@SpringBootApplication注解

Many Spring Boot developers like their apps to use auto-configuration, component scan and be ableto define extra configuration on their "application class". A single @SpringBootApplicationannotation can be used to enable those three features, that is:

- @EnableAutoConfiguration: enable Spring Boot's auto-configuration mechanism
- @ComponentScan: enable @Component scan on the package where the application is located
- @Configuration: allow to register extra beans in the context or import additional configurationclasses

None of these features are mandatory(强制的) and you may choose to replace this singleannotation by any of the features that it enables

example:

```
@Configuration(proxyBeanMethods = false)
@EnableAutoConfiguration
@Import({ MyConfig.class, MyAnotherConfig.class })
public class Application {...}
//In this example, Application is just like any other Spring Boot application exceptthat @Component-annotated classes and @ConfigurationProperties-annotated classesare not detected automatically and the user-defined beans are imported explicitly(see @Import).
```

SpringBoot Features

SpringApplication

启动

The SpringApplication class provides a convenient way to bootstrap a Spring application that is started from a main() method.

```
public static void main(String[] args) {
 SpringApplication.run(MySpringConfiguration.class, args);}
//By default, INFO logging messages are shown, Startup information logging can be turned off by settingspring.main.log-startup-info to false.
//If your application fails to start, registered FailureAnalyzers get a chance to provide a dedicatederror message and a concrete action to fix the problem. Spring Boot provides numerous FailureAnalyzer implementations.
//If no failure analyzers are able to handle the exception, you can still display the full conditionsreport to better understand what went wrong. To do so, you need to enable the debug property orenable DEBUG logging for org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener
下面是一种流式写法
new SpringApplicationBuilder().sources(Parent.class).child(Application.class).bannerMode(Banner.Mode.OFF).run(args);//此处关闭了banner
```

Lazy初始化

```
//If a misconfigured bean is initialized lazily, a failure will no longer occur during startup and the problem will only become apparent when the bean is initialized. Care must also be taken to ensurethat the JVM has sufficient memory to accommodate all of the application's beans and not just thosethat are initialized during startup. For these reasons, lazy initialization is not enabled by default;默认非懒加载
it can be enabled using the spring.main.lazy-initialization property :
spring.main.lazy-initialization=true
```

监测

```
//Spring Boot provides Kubernetes HTTP probes for "Liveness" and "Readiness" with Actuator HealthEndpoints.//需要再分析
```

Application Events and Listeners

存在很多的发布事件时机，通常不会使用它们，但得知道它们存在，下面是一部分事件发生的时机

```
1.An ApplicationStartingEvent is sent at the start of a run but
 before any processing, except for
the registration of listeners and initializers.
2.An ApplicationEnvironmentPreparedEvent is sent when the Environment to be used
in the context isknown but before the context is created.
3.An ApplicationContextInitializedEvent is sent when the
 ApplicationContext is prepared and ApplicationContextInitializers have been
called but before any bean definitions are loaded.
4.An ApplicationPreparedEvent is sent just before the refresh is started but
after bean definitionshave been loaded.
5.An ApplicationStartedEvent is sent after the context has been
  refreshed but before anyapplication and command-line runners have been
called.
6.An AvailabilityChangeEvent is sent right after with
 LivenessState.CORRECT to indicate that theapplication is considered as
live.
7.An ApplicationReadyEvent is sent after any application and command-
line runners have beencalled.
8.An AvailabilityChangeEvent is sent right after with
ReadinessState.ACCEPTING_TRAFFIC to indicatethat the application is ready to
service requests.
9.An ApplicationFailedEvent is sent if there is an exception on startup.
...
//Some events are actually triggered before the ApplicationContext is created, so you cannot register a listener on those as a @Bean. You can register them with theSpringApplication.addListeners(...) method or the SpringApplicationBuilder.listeners(...) method.If you want those listeners to be registered automatically, regardless of the way theapplication is created, you can add a META-INF/spring.factories file to your projectand reference your listener(s) by using the org.springframework.context.ApplicationListener key, as shown in the following example:
org.springframework.context.ApplicationListener=com.example.project.MyListener
```

ApplicationContext

A SpringApplication attempts to create the right type of ApplicationContext on your behalf. The algorithm used to determine a WebApplicationType is the following:

- If Spring MVC is present, an **AnnotationConfigServletWebServerApplicationContext** is used
- If Spring MVC is not present and Spring WebFlux is present, an **AnnotationConfigReactiveWebServerApplicationContext** is used
- Otherwise, **AnnotationConfigApplicationContext** is used

setWebApplicationType(WebApplicationType)/setApplicationContextClass(...) 可进行调整

AccessingApplicationArguments

```
If you need to access the application arguments that were passed to
SpringApplication.run(...), youcan    inject    a
 org.springframework.boot.ApplicationArguments    bean.    The
 ApplicationArgumentsinterface  provides  access  to  both  the  raw  String[]
 arguments  as  well  as  parsed  option  and  non-option arguments
@Autowired
public MyBean(ApplicationArguments args) {
    boolean debug = args.containsOption("debug");
    List<String> files = args.getNonOptionArgs();
    // if run with "--debug logfile.txt" debug=true, files=["logfile.txt"]
}
Spring Boot also registers a CommandLinePropertySource with the Spring
Environment.This   lets   you   also   inject   single   application   arguments
  by   using   the   @Value annotation
```

ApplicationRunner or CommandLineRunner

```
If  you  need  to  run  some  specific  code  once  the  SpringApplication  has
 started,  you  can  implementthe  ApplicationRunner  or  CommandLineRunner
 interfaces.  Both  interfaces  work  in  the  same  way  andoffer a single run
method, which is called just before SpringApplication.run(...) completes.This
 contract  is  well  suited  for  tasks  that  should  run  after  application
 startup  butbefore it starts accepting traffic//适用于应用程序启动后但在其开始接受流量
之前运行的任务
代码仅执行一次时，可以使用这里的接口在容器启动时执行一次，可以用Order接口进行排序
```

Externalized Configuration

Spring Boot lets you externalize your configuration so that you can work with the same applicationcode in different environments. You can use a variety of external configuration sources, include **Java properties files, YAML files, environment variables, and command-line arguments**.

Property values can be injected directly into your beans by using the **@Value** annotation, accessed through Spring's Environment abstraction, or be bound to **structured objects** through **@ConfigurationProperties**

Spring Boot uses a very particular PropertySource order that is designed to allow sensible overriding of values. Properties are considered in the following order (with values from lower items overriding earlier ones)://后面条目的属性会覆盖前面条目的属性

1. Default properties (specified by setting SpringApplication.setDefaultProperties).

2. @PropertySource annotations on your @Configuration classes. Please note that such property sources are not added to the Environment until the application context is being refreshed. This is too late to configure certain properties such as logging.* and spring.main.* which are readbefore refresh begins.
    - Annotation providing a convenient and declarative mechanism for adding a PropertySource to Spring's Environment. To be used in conjunction with @Configuration classes.

      e.g.

        - @PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")
3. Config data (such as application.properties files) //后面的条目属性会覆盖前面条目的属性，即后面条目优先级更高，覆盖前面优先级低的属性
    1. Application properties packaged inside your jar (application.properties and YAML variants).
    2. Profile-specific application properties packaged inside your jar (application-{profile}.properties and YAML variants).
    3. Application properties outside of your packaged jar (application.properties and YAMLvariants).
    4. Profile-specific application properties outside of your packaged jar (application-{profile}.properties and YAML variants)

```
Spring Boot will automatically find and load application.properties and
application.yaml files fromthe following locations when your application
starts:
1.The classpath root
2.The classpath /config package
3.The current
4.The /config subdirectory in the current directory
5.Immediate child directories of the /config subdirectory
后面的优先级高于前面的优先级，后面的属性会覆盖前面的属性
If you do not like application as the configuration file name, you can
switch to another file name byspecifying a spring.config.name environment
property. You can also refer to an explicit location byusing the
spring.config.location environment property (which is a comma-
separated list of directory locations or file paths//可以指定location和配置文
件名称
java -jar myproject.jar --spring.config.name=myproject//指定名称
java -jar myproject.jar --
spring.config.location=optional:classpath:/default.properties,optional:class
path:/override.properties//指定位置和名称
//Use the prefix optional: if the locations are optional and you
don’t mind if theydon’t exist.
As well as application property files, Spring Boot will also attempt to load
profile-specific files usingthe naming convention application-
{profile}.you can use spring.profiles.active(允许通过,指定多个文件) property to
specify. if no profiles are explicitly activated, then properties
from application-default are considered.//默认也会加载application-default形式
的配置文件
可以通过spring.config.import进一步引入新的配置文件，比如可以在
application.properties file中定义
spring.config.import=optional:file:./dev.properties 引入额外的配置，对于无扩展名
文件，可以通过spring.config.import=file:/etc/config/myconfig[.yaml]这种方式给
springboot一个提示，按照yaml形式导入文件
```

相对于classpath和file来说，还有一种configtree的方式，将文件名作为key，将文件内容做为value，比如spring.config.import=optional:configtree:/etc/config/，且存在/etc/config/myapp/username和/etc/config/myapp/password文件，那么在springboot中可以使用myapp.username和myapp.password
可以通过Property Plachholder引用属性:app.name=MyAppapp
app.description=${app.name}
一个单独的配置文件可以拆分成多个逻辑文件，在yaml中通过---，在properties中通过#---，再配合spring.config.activate.*进行条件激活逻辑文件
The following specifies that the second document is only active when running on Kubernetes, and only when either the “prod” or “staging” profiles are active:
myprop=always-set
#---
spring.config.activate.on-cloud-platform=kubernetes
spring.config.activate.on-profile=prod | staging
myotherprop=sometimes-set
属性加密：Spring Boot does not provide any built in support for encrypting property values, however, it doesprovide the hook points necessary to modify values contained in the Spring Environment.
TheEnvironmentPostProcessor interface allows you to manipulate the Environment before the applicationstarts.另外，一个可选的参考the Spring Cloud Vault project
YAML files cannot be loaded by using the @PropertySource annotation. So, in the case that you need to load values that way, you need to use a properties file.//yaml的缺点，不能适用@PropertySource注解
不建议混合使用spring.profiles.active与spring.config.activate多逻辑文件

4. A RandomValuePropertySource that has properties only in random.*.

```
允许在配置文件中使用随机值
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.uuid=${random.uuid}
my.number-less-than-ten=${random.int(10)}
my.number-in-range=${random.int[1024,65536]}
```

5. OS environment variables.
6. Java System properties (System.getProperties()).
    1. java -Dprokey=value 这种方式附带启动的参数为系统属性
    2. springboot对于一些特定application.properties中的属性也会转换成系统属性，可以查看后面日志部分，有一些这样的情况
    3. 另外其还包含：information about the current user,the current version of the java runtime等，debuge还能看到一些虚拟机信息等
    4. 实际就是System类中的一个Properties类型的静态数据
7. JNDI attributes from java:comp/env.
8. ServletContext init parameters.
9. ServletConfig init parameters.
10. Properties from SPRING_APPLICATION_JSON (inline JSON embedded in an environment variable or system property).

```
When your application starts, any spring.application.json or
 SPRING_APPLICATION_JSON propertieswill be parsed and added to the
Environment
  环境变量 SPRING_APPLICATION_JSON='{"acme":{"name":"test"}' //you end up
with acme.name=test in the Spring Environment
  system property java -Dspring.application.json='{"acme":{"name":"test"}}'
-jar myapp.jar
  满足下一条目的命令行参数形式也是可以的 --spring.application.json='{"acme":
{"name":"test"}}'
  JNDI的形式java:comp/env/spring.application.json 也是可以的
```

11. Command line arguments.

```
By default, SpringApplication converts any command line option
 arguments (that is, arguments starting with --, such as --
server.port=9000) to a property and adds them to the Spring Environment.If
 you do not want command line properties to be added to the
 Environment, you can disable them by using
SpringApplication.setAddCommandLineProperties(false).
```

12. properties attribute on your tests. Available on @SpringBootTest and the test annotations for testing a particular slice of your application.
13. @TestPropertySource annotations on your tests.
14. Devtools global settings properties in the $HOME/.config/spring-boot directory when devtools isactive.

Type-safe Configuration Properties

对于上述第3点PropertySource涉及的配置文件，此处进一步总结：使用@Value("${property}") 注入配置属性有时会显得笨重，Spring Boot provides an alternative method of working with properties that lets strongly typed beans govern and validate the configuration of your application.使用强类型Bean管理和验证配置

JavaBean properties binding

```
It is possible to bind a bean declaring standard JavaBean properties
 as shown in the following example:
@ConfigurationProperties("acme")
public class AcmeProperties {
    private boolean enabled;
    private InetAddress remoteAddress;
    private final Security security = new Security();
    public boolean isEnabled() { ... }
    public void setEnabled(boolean enabled) { ... }
    public InetAddress getRemoteAddress() { ... }
    public void setRemoteAddress(InetAddress remoteAddress) { ... }
    public Security getSecurity() { ... }
    public static class Security {
        private String username;
        private String password;
        private List<String> roles = new ArrayList<>
(Collections.singleton("USER"));
        public String getUsername() { ... }
        public void setUsername(String username) { ... }
        public String getPassword() { ... }
```

```
        public void setPassword(String password) { ... }
        public List<String> getRoles() { ... }
        public void setRoles(List<String> roles) { ... }
    }
}
```

The preceding POJO defines the following properties:

- acme.enabled, with a value of false by default.
- acme.remote-address, with a type that can be coerced from String.
- acme.security.username, with a nested "security" object whose name is determined by the nameof the property. In particular, the return type is not used at all there and could have beenSecurityProperties.//可以嵌套对象使用
- acme.security.password.
- acme.security.roles, with a collection of String that defaults to USER.

The properties that map to @ConfigurationProperties classes available in SpringBoot, which are configured via properties files, YAML files, environment variables

A setter may be omitted in the following cases://可以省略setter的情形

- Maps, as long as they are initialized, need a getter but not necessarily a setter,since they can be mutated by the binder.
- Collections and arrays can be accessed either through an index (typically withYAML) or by using a single comma-separated value (properties). In the lattercase, a setter is mandatory. We recommend to always add a setter for suchtypes. If you initialize a collection, make sure it is not immutable (as in thepreceding example).
- If nested POJO properties are initialized (like the Security field in the precedingexample), a setter is not required. If you want the binder to create the instanceon the fly by using its default constructor, you need a setter.

Only standard Java Bean properties are considered and binding on static properties is not supported.

Constructor binding

The example in the previous section can be rewritten in an immutable fashion as shown in the following example:

```
@ConstructorBinding
@ConfigurationProperties("acme")
public class AcmeProperties {
    private final boolean enabled;
    private final InetAddress remoteAddress;
    private final Security security;
    public AcmeProperties(boolean enabled, InetAddress remoteAddress,
Securitysecurity) {
        this.enabled = enabled;
        this.remoteAddress = remoteAddress;
        this.security = security;
    }
    public boolean isEnabled() { ... }
    public InetAddress getRemoteAddress() { ... }
    public Security getSecurity() { ... }
    public static class Security {
        private final String username;
        private final String password;
```

```
        private final List<String> roles;
        public Security(String username, String password,@DefaultValue("USER")
List<String> roles) {
            this.username = username;
            this.password = password;
            this.roles = roles;
        }
        public String getUsername() { ... }
        public String getPassword() { ... }
        public List<String> getRoles() { ... }
    }
}
```

In this setup, the @ConstructorBinding annotation is used to indicate that constructor binding shouldbe used. Nested members of a @ConstructorBinding class (such as Security in the example above) will also bebound via their constructor.

Default values can be specified using @DefaultValue and the same conversion service will be applied to coerce the String value to the target type of a missing property.//如果在属性缺失的情况下，@DefaultValue中的值会转换成目标类型对象使用

By default, if no properties are bound to Security, the AcmeProperties instance will contain a null value for security. If you wish you return a non-null instance of Security even when no properties are bound to it, you can use anempty @DefaultValue annotation to do so:

```
@ConstructorBinding
@ConfigurationProperties("acme")
public class AcmeProperties {
    private final boolean enabled;
    private final InetAddress remoteAddress;
    private final Security security;
    public AcmeProperties(boolean enabled, InetAddress remoteAddress,
@DefaultValue Security security) {
        this.enabled = enabled;
        this.remoteAddress = remoteAddress;
        this.security = security;
    }
}
```

To use constructor binding the class must be enabled using @EnableConfigurationProperties or configuration property scanning. You cannot use constructor binding with beans that are created by the regular Spring mechanisms (e.g. @Component beans, beans created via @Bean methods or beans loaded using @Import)//使用构造函数绑定时必须启用@EnableConfigurationProperties 或者配置属性扫描功能，常规的Spring机制（@Component beans, beans created via @Bean methods or beans loaded using @Import）构建的Bean不能使用构造函数绑定

If you have more than one constructor for your class you can also use@ConstructorBinding directly on the constructor that should be bound.//超过一个构造函数时的处理

The use of java.util.Optional with @ConfigurationProperties is not recommended

Enabling @ConfigurationProperties

Spring Boot provides infrastructure to bind @ConfigurationProperties types and register them as beans. You can either enable configuration properties on a class-by-class basis or enable configuration property scanning that works in a similar manner to component scanning.激活 @ConfigurationProperties做为Bean注入容器有两种方式

Sometimes, classes annotated with @ConfigurationProperties might not be suitable for scanning, for example, if you're developing your own auto-configuration or you want to enable them conditionally. In these cases, specify the list of types to process using the @EnableConfigurationProperties annotation. This can be done on any @Configuration class, as shownin the following example:有时候不适合报扫描形式加入ConfigurationProperties，这里是指定类形式

```
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(AcmeProperties.class)
public class MyConfiguration {}
```

To use configuration property scanning, add the @ConfigurationPropertiesScan annotation to yourapplication. Typically, it is added to the main application class that is annotated with@SpringBootApplication but it can be added to any @Configuration class. By default, scanning willoccur from the package of the class that declares the annotation. If you want to define specificpackages to scan, you can do so as shown in the following example:扫描形式

```
@SpringBootApplication
//底层使用ConfigurationPropertiesScanRegistrar进行注册，自定义了扫描；看着不算难，有需要应该可以仿着这里实现一个自定义注解扫描功能；
@ConfigurationPropertiesScan({ "com.example.app", "org.acme.another" })
public class MyApplication {}
```

We recommend that @ConfigurationProperties only deal with the environment and, in particular,does not inject other beans from the context. For corner cases, setter injection can be used or any of the *Aware interfaces provided by the framework (such as EnvironmentAware if you need access to the Environment). If you still want to inject other beans using the constructor, the configuration properties bean must be annotated with @Component and use JavaBean-based property binding. 推荐@ConfigurationProperties 仅仅和Environment进行交互，不在其中注入其它Bean，在特殊情况下setter注入和*Aware注入也是可以的；如果使用构造函数注入其它Bean，那么@ConfigurationProperties标注的类还必须还被标注为@Component等要求；@ConfigurationProperties标注的对象，可以@Autowired装配给其它对象

Third-party Configuration

可以在@Bean使用时同时使用@ConfigurationProperties，那么Bean在创建后可以根据ConfigurationProperties绑定属性

```
@ConfigurationProperties(prefix = "another")//所有another开头的属性会尝试绑定到AnotherComponent的属性上
@Bean
public AnotherComponent anotherComponent() {...}
```

Relaxed Binding 宽松绑定

Spring Boot uses some relaxed rules for binding Environment properties to @ConfigurationProperties beans, so there does not need to be an exact match between the Environment property name and thebean property name.不要求名称完全匹配，只要满足特定的命名规则

```
@ConfigurationProperties(prefix="acme.my-project.person")
public class OwnerProperties {
    private String firstName;
    public String getFirstName() {
        return this.firstName;
    }
    public void setFirstName(String f irstName) {
        this.firstName = firstName;
    }
}
```

下面是使用情形描述

Property	Note
`acme.my-project.person.first-name`	Kebab case, which is recommended for use in `.properties` and `.yml` files.
`acme.myProject.person.firstName`	Standard camel case syntax.
`acme.my_project.person.first_name`	Underscore notation, which is an alternative format for use in `.properties` and `.yml` files.
`ACME_MYPROJECT_PERSON_FIRSTNAME`	Upper case format, which is recommended when using system environment variables.

The `prefix` value for the annotation *must* be in kebab case (lowercase and separated by `-`, such as `acme.my-project.person`).

Table 7. relaxed binding rules per property source

Property Source	Simple	List
Properties Files	Camel case, kebab case, or underscore notation	Standard list syntax using `[ ]` or comma-separated values
YAML Files	Camel case, kebab case, or underscore notation	Standard YAML list syntax or comma-separated values
Environment Variables	Upper case format with underscore as the delimiter (see Binding from Environment Variables).	Numeric values surrounded by underscores (see Binding from Environment Variables)
System properties	Camel case, kebab case, or underscore notation	Standard list syntax using `[ ]` or comma-separated values

We recommend that, when possible, properties are stored in lower-case kebab format, such as `my.property-name=acme`.

Map绑定

当绑定Map属性时，key如果只包含小写字母或者-，那么无需注意什么；否则，必须使用[]将key包围起来，这样其它符号才能得以保留，如果不是用[]包围key，那么其它符号将会被直接移除（效果上等同移除效果）

```
acme.map.[/key1]=value1
acme.map.[/key2]=value2
acme.map./key3=value3
or
acme:
    map:
        "[/key1]": value1
        "[/key2]": value2
        "/key3": value3
//The properties above will bind to a Map with /key1, /key2 and key3 as the keys in the map
```

环境变量绑定

环境变量的名称通常有严格的限定，为了将属性写在环境变量中，可以将配置文件的名称按照如下规则进行转换后再写入环境变量

- Replace dots (.) with underscores (_).
- Remove any dashes (-).
- Convert to uppercase.

For example, the configuration property spring.main.log-startup-info would be an environmentvariable named SPRING_MAIN_LOGSTARTUPINFO，the configuration property my.acme[0].other would use an environment variablenamed MY_ACME_0_OTHER.

复杂类型合并

When lists are configured in more than one place, overriding works by replacing the entire list.When a List is specified in multiple profiles, the one with the highest priority (and only that one) isused.// List内容不会发生单Entry替换，只会整个List都被重建(所有Entry使用最高有限级别的数据进行重建，比如高有限级只有一条数据，低优先级的地方有两条数据，最后也只会使用高优先级的一条数据进行List构建)

For Map properties, you can bind with property values drawn from multiple sources. However, forthe same property in multiple sources, the one with the highest priority is used.//Map内容会发生单Entry的属性替换，不会整个Map或者Entry替换，只会使用优先级最高的属性进行覆盖（比如低优先级key为key1且EntryValue.name=A EntryValue.desc=D，高优先级key为key1且EntryValue.name=B，那么最后key1对应的EntryValue.name=B, EntryValue.desc=D;此外，key只要出现过，就是map的一条记录，不会出现只使用高优先级数据的情况，所有数据都会使用，只是高优先会覆盖低优先级，和List很不一样)

The preceding merging rules apply to properties from all property sources, and not just files.

属性转换

Spring Boot attempts to coerce the external application properties to the right type when it binds tothe @ConfigurationProperties beans. If you need custom type conversion, you can provide aConversionService bean (with a bean named conversionService) or custom property editors (througha CustomEditorConfigurer bean) or custom Converters (with bean definitions annotated as@ConfigurationPropertiesBinding)

@ConfigurationProperties Validation 校验

Spring Boot attempts to validate @ConfigurationProperties classes whenever they are annotatedwith Spring's @Validated annotation. You can use JSR-303 javax.validation constraint annotationsdirectly on your configuration class. To do so, ensure that a compliant JSR-303 implementation is on your classpath and then add constraint annotations to your fields, as shown in the following example:

```
@ConfigurationProperties(prefix="acme")
@Validatedpublic
class AcmeProperties {
    @NotNull
    private InetAddress remoteAddress;
    @Valid//To ensure that validation is always triggered for nested
properties, even when no properties arefound, the associated field must
be annotated with @Valid.
    private final Security security = new Security();
    // ... getters and setters
    public static class Security {
        @NotEmpty
        public String username;
        // ... getters and setters
    }
}
you can also trigger validation by annotating the @Bean method that
 creates the configuration properties with @Validated
```

You can also add a custom Spring Validator by creating a bean definition calledconfigurationPropertiesValidator. The @Bean method should be declared static. The configurationproperties validator is created very early in the application's lifecycle, and declaring the @Beanmethod as static lets the bean be created without having to instantiate the @Configuration class.Doing so avoids any problems that may be caused by early instantiation.

@ConfigurationProperties vs. @Value

Feature	@ConfigurationProperties	@Value
Relaxed binding	Yes	Limited (see note below)
Meta-data support	Yes	No
SpEL evaluation	No	Yes

If you do want to use @Value, we recommend that you refer to property names using their canonical form (kebab-case using only lowercase letters). This will allow Spring Boot to use the same logic as it does when relaxed binding @ConfigurationProperties. For example, @Value("{demo.item-price}") will pick up demo.item-price and demo.itemPrice forms from the application.properties file, as well as DEMO_ITEMPRICE from the system environment. If you used @Value("{demo.itemPrice}") instead, demo.item-price and DEMO_ITEMPRICE would not be considered.

Profiles 环境切换

Spring Profiles provide a way to segregate parts of your application configuration and make it beavailable only in certain environments. Any @Component, @Configuration or @ConfigurationPropertiescan be marked with @Profile to limit when it is loaded, as shown in the following example:

```
@Configuration(proxyBeanMethods = false)
@Profile("production")
public class ProductionConfiguration {
    // ...
}
You can use a spring.profiles.activeEnvironment property to specify which
profiles are active
spring.profiles.active=dev,hsqldb
除了上述方式，we can create a production group that consists of our proddb and
prodmq profiles
spring.profiles.group.production[0]=proddb
spring.profiles.group.production[1]=prodmq
Our application can now be started using --spring.profiles.active=production to
active the production, proddb and prodmq profiles in one hit
```

Logging

Spring Boot uses Commons Logging for all internal logging but leaves the underlying logimplementation open. Default configurations are provided for Java Util Logging, Log4J2, andLogback. In each case, loggers are pre-configured to use console output with optional file outputalso available.//使用面向接口编程，将底层实现开放

By default, if you use the “Starters”, Logback is used for logging. Appropriate Logback routing isalso included to ensure that dependent libraries that use Java Util Logging, Commons Logging,Log4J, or SLF4J all work correctly.//starter默认使用logback

LogFormat

The defualt log output:

- Date and Time: Millisecond precision and easily sortable.
- Log Level: ERROR, WARN, INFO, DEBUG, or TRACE.
- Process ID.
- A --- separator to distinguish the start of actual log messages.
- Thread name: Enclosed in square brackets (may be truncated for console output).
- Logger name: This is usually the source class name (often abbreviated).
- The log message.

  默认springboot Logback does not have a FATAL level. It is mapped to ERROR.

Color-coded Output

If your terminal supports ANSI, color output is used to aid readability. You can set spring.output.ansi.enabled to a supported value to override the auto-detection.如果终端支持的话，用户可以控制终端输出的字体颜色，不展开说，需要时查资料

File Output

By default, Spring Boot logs only to the console and does not write log files. If you want to write log files in addition to the console output, you need to set a **logging.file.name** or **logging.file.path** property(for example, in your application.properties) 默认SpringBoot日志只打印在Console，要写文件必须手动设置相关属性

The following table shows how the `logging.*` properties can be used together:

Table 8. Logging properties

logging.file.name	logging.file.path	Example	Description
(none)	*(none)*		Console only logging.
Specific file	*(none)*	`my.log`	Writes to the specified log file. Names can be an exact location or relative to the current directory.
(none)	Specific directory	`/var/log`	Writes `spring.log` to the specified directory. Names can be an exact location or relative to the current directory.

Log files rotate when they reach 10 MB and, as with console output, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged by default.

Logging properties are independent of the actual logging infrastructure. As a result, specific configuration keys (such as `logback.configurationFile` for Logback) are not managed by spring Boot.

默认日志等级INFO，默认日志循环覆盖大小10M;**日志的特定属性和具体日志框架相关，SpringBoot配置文件不管理这部分配置**;个人理解properties这些文件管理一些通用简单场景，对于复杂的日志需求还是需要通过xml配置；可以简单认为除了笔记记录的属性，其它属性基本上需要通过日志框架的xml文件进行配置；

File Rotation

If you are using the Logback, it's possible to fine-tune log rotation settings using your application.properties or application.yaml file. For all other logging system, you'll need to configure rotation settings directly yourself (for example, if you use Log4J2 then you could add a log4j.xml file).springboot由于默认使用logback，对其支持比较好，可以在application.properties中进行配置日志循环覆盖的相关属性，下面是Springboot配置文件支持的logback属性

The following rotation policy properties are supported:

Name	Description
`logging.logback.rollingpolicy.file-name-pattern`	The filename pattern used to create log archives.
`logging.logback.rollingpolicy.clean-history-on-start`	If log archive cleanup should occur when the application starts.
`logging.logback.rollingpolicy.max-file-size`	The maximum size of log file before it's archived.
`logging.logback.rollingpolicy.total-size-cap`	The maximum amount of size log archives can take before being deleted.
`logging.logback.rollingpolicy.max-history`	The number of days to keep log archives (defaults to 7)

Log Level

```
logging.group.tomcat=org.apache.catalina,org.apache.coyote,org.apache.tomcat//分组 Log Group
logging.level.tomcat=trace
logging.level.root=warn
logging.level.org.springframework.web=debug
```

Custom Log Configuration

The various logging systems can be activated by including the appropriate libraries on the classpath and can be further customized by providing a suitable configuration file in the root of the classpath or in a location specified by the following Spring `Environment` property: `logging.config`.

You can force Spring Boot to use a particular logging system by using the `org.springframework.boot.logging.LoggingSystem` system property. The value should be the fully qualified class name of a `LoggingSystem` implementation. You can also disable Spring Boot's logging configuration entirely by using a value of `none`.

Since logging is initialized **before** the `ApplicationContext` is created, it is not possible to control logging from `@PropertySources` in Spring `@Configuration` files. The only way to change the logging system or disable it entirely is via System properties.

Depending on your logging system, the following files are loaded:

Logging System	Customization
Logback	`logback-spring.xml`, `logback-spring.groovy`, `logback.xml`, or `logback.groovy`
Log4j2	`log4j2-spring.xml` or `log4j2.xml`
JDK (Java Util Logging)	`logging.properties`

To help with the customization, some other properties are transferred from the Spring `Environment` to System properties, as described in the following table:

Spring Environment	System Property	Comments
`logging.exception-conversion-word`	`LOG_EXCEPTION_CONVERSION_WORD`	The conversion word used when logging exceptions.
`logging.file.name`	`LOG_FILE`	If defined, it is used in the default log configuration.
`logging.file.path`	`LOG_PATH`	If defined, it is used in the default log configuration.
`logging.pattern.console`	`CONSOLE_LOG_PATTERN`	The log pattern to use on the console (stdout).
`logging.pattern.dateformat`	`LOG_DATEFORMAT_PATTERN`	Appender pattern for log date format.
`logging.charset.console`	`CONSOLE_LOG_CHARSET`	The charset to use for console logging.
`logging.pattern.file`	`FILE_LOG_PATTERN`	The log pattern to use in a file (if `LOG_FILE` is enabled).
`logging.charset.file`	`FILE_LOG_CHARSET`	The charset to use for file logging (if `LOG_FILE` is enabled).
`logging.pattern.level`	`LOG_LEVEL_PATTERN`	The format to use when rendering the log level (default `%5p`).
`PID`	`PID`	The current process ID (discovered if possible and when not already defined as an OS environment variable).

- 可以通过logging.config定义配置文件位置
- 日志系统初始化于ApplicationContext创建之前，所以只能通过System Properties(调试测试logging.file.name在前文说的第6点Java System properties (System.getProperties()))设置LoggingSystem
- 推荐使用带-spring的配置文件（功能更强），尽量不要在以excutable jar时使用JDK Log，存在类加载问题
    - Spring Boot includes a number of extensions to Logback that can help with advanced configuration.You can use these extensions in your logback-spring.xml configuration file.

- The <springProfile> tag lets you optionally include or exclude sections of configuration based onthe active Spring profiles

```
<springProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
</springProfile>
<springProfile name="dev | staging">
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active-->
</springProfile>
<springProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is not active -->
</springProfile>
```

- The <springProperty> tag lets you expose properties from the Spring Environment for use withinLogback.

```
<springProperty scope="context" name="fluentHost"
source="myapp.fluentd.host" defaultValue="localhost"/>
<appender name="FLUENT"
class="ch.qos.logback.more.appenders.DataFluentAppender">
    <remoteHost>${fluentHost}</remoteHost>
    ...
</appender>
```

- LoggingApplicationListener

> An ApplicationListener that configures the LoggingSystem. If the environment contains a logging.config property it will be used to bootstrap the logging system, otherwise a default configuration is used.

Using Logging - slf4j logback

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
//一种标准的使用方法
public class Wombat {
    final static Logger logger = LoggerFactory.getLogger(Wombat.class);//如果未引入实现类，默认返回一个NOPLogger，其日志相关方法都是空方法体
    Integer t;
    Integer oldT;

    public void setTemperature(Integer temperature) {
        oldT = t;
        t = temperature;
        logger.debug("Temperature set to {}. Old temperature was {}.", t, oldT);
        if(temperature.intValue() > 50) {
            logger.info("Temperature has risen above 50 degrees.");
        }
    }
}
```

- **logback架构**

  日志API和普通println的重要优势是可以禁用某些日志语句并通知允许其他语句不受阻碍的打印

  - Logger
    - logger名字区分大小写，并遵循分层命名规则（如com.foo是com.foo.Bar的logger的父），相同的名称将获取相同的logger
      - 以日志记录器所在类命名日志记录器是已知的最佳通用策略
      - 调试可以看到每个.都会创建一个logger，即使你只使用了com.foo的Logger，程序还是会创建叫com的logger
    - 根logger比较特殊，名称为ROOT，且默认级别DEBUG，是所有logger的顶层父节点
    - 每个logger都可以指定LEVEL级别，如果未指定的话，将从最近的父logger尝试获取父logger的LEVEL；如果也没有就继续往上找，直到找到为止并使用它
      - logger.setLevel，一个改变logger级别的方法
      - 最后logger.setLevel或继承的级别称为有效级别
    - 日志记录请求级别大于等于其日志记录程序logger的有效级别，则启用日志记录请求；否则，该请求被禁用
      - TRACE < DEBUG < INFO < WARN < ERROR
      - 每个启用的日志请求都将被转发给该logger的所有appender以及层次结构中更高的logger(如果Additivity Flag为false，就不会向高层次转发（L开始，P为false，那么L-P之间的appender可用，更高层次的appender则不可用）；

```
//Invoke all the appenders of this logger.
//Params:event - The event to log
public void callAppenders(ILoggingEvent event) {
    int writes = 0;
    for (Logger l = this; l != null; l = l.parent) {//这里先处理当前logger，再处理父logger
        writes += l.appendLoopOnAppenders(event);//所有Logger都处理的同一event，所以event.loggerName都是一致的
        if (!l.additive) {//默认l.additive为true，写成false后，这个递归处理过程就停止了
            break;//前面说的叫com的logger不存在关联的appender,additive为true,在处理时相当于无事发生
        }
    }
    // No appenders in hierarchy
    if (writes == 0) {
        loggerContext.noAppenderDefinedWarning(this);
    }
}
private int appendLoopOnAppenders(ILoggingEvent event) {
    if (aai != null) {
        return aai.appendLoopOnAppenders(event);//aai是AppenderAttachableImpl的实现，是一个logger关联的多个输出目的target
    } else {
        return 0;
    }
}
public int appendLoopOnAppenders(E e) {
    int size = 0;
    final Appender<E>[] appenderArray = appenderList.asTypedArray();
    final int len = appenderArray.length;
```

```
    for (int i = 0; i < len; i++) {//处理当前层次的logger的所有关联
的appender
        appenderArray[i].doAppend(e);
        size++;
    }
    return size;
}
doAppend(e){
    ...
    byte[] byteArray = this.encoder.encode(event);//格式化串，
txt=layout.doLayout(event);convertToBytes(txt);
    writeBytes(byteArray);//写target,一种实现
this.outputStream.write(byteArray);
    ...
}
```

- LoggerContext
  - 每一个logger都附加到一个LoggerContext，这些LoggerContext负责生成logger
  - 调试可以看到默认都在一个默认的LoggerContext中
- Appender
  - 在logback中，输出目的地称为appender，可以设置为控制台、文件、远程套接字服务器、常用数据库、JMS等
  - 一个logger可以附加到多个Appender
    - logger.addAppender，一个可增加Appender的方法
- Layout
  - 用于格式化日志请求
- 详细配置参考官网，一个简单的示例也可参考https://developer.aliyun.com/article/902043

SpringBoot流程分析

启动流程

```
SpringApplication.run(Application.class, args)
    run(new Class<?>[] { primarySource }, args)
        new SpringApplication(primarySources).run(args)//前3步调用栈

new SpringApplication(primarySources)//此处时第3步构造函数部分
    this(null, primarySources)//此处即是下面的方法
public SpringApplication(ResourceLoader resourceLoader, Class<?>...
primarySources) {
    this.resourceLoader = resourceLoader;//null
    Assert.notNull(primarySources, "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));//即
为Application.class
    //依据类路径下特定类存在情况判断容器环境，WebApplicationType.REACTIVE(流式编程
web)/SERVLET（web）/NONE(非web)
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    //此处是查找所有jar下面的META-INF/spring.factories，并从中找出
    //key为org.springframework.boot.BootstrapRegistryInitializer中的值（值为该key对
应的实现类）
    //其内部使用SpringFactoriesLoader的机制进行加载相关的实现类，主要方法是
SpringFactoriesLoader.loadSpringFactories
```

```
    //SpringFactoriesLoader loads and instantiates factories of a given type from
    //"META-INF/spring.factories" files which may be present in multiple JAR
files in the classpath.
    //The spring.factories file must be in Properties format, where the key is
the fully qualified name of
    //the interface or abstract class, and the value is a comma-separated list of
implementation class
    //names. For
example:example.MyService=example.MyServiceImpl1,example.MyServiceImpl2
    //where example.MyService is the name of the interface, and MyServiceImpl1
and MyServiceImpl2 are two
    //implementations//代码中有Assert.isAssignable(type, instanceClass);会有检测，右
边必须是左侧key的实现类
    //抛开当前的bootstrapRegistry来说，
SpringFactoriesLoader.getSpringFactoriesInstances有多种重载方法，
    //影响着实现类的构造函数的调用；此处虽然名字叫做factory，但并不是需要实现某种固定factory
接口；
    //第一次获取数据时会获取所有spring.factories中数据，并缓存起来，下次调用时直接从缓存查找
    //整个过程有点等同于一般的SPI的过程，只不过这里是spring自己实现的，两者可以进行类比
    //查找出来的实现类会进行sort排序，和Ordered接口以及@Order，@Priority有关
    //由于这部分实现机制本身是比较简单的，直接看源码即可，这里只进行总结性概述
    this.bootstrapRegistryInitializers = new ArrayList<>(
        getSpringFactoriesInstances(BootstrapRegistryInitializer.class));//调试
时，在webstarter情况下默认为0
    //此处依旧使用了SpringFactoriesLoader加载key为
org.springframework.context.ApplicationContextInitializer
    //的实现类，调试时在webstarter情况下默认有7个，并设置到SpringApplication的
initializers顺序中
    setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
    //同前两个类似，此处要求key为ApplicationListener.class.getName() 前面所说的key都是
如此形式来查找的
    //在SpringApplication中注册ApplicationListener；不仅仅此处还有后面的启动流程，如果想
要干预springboot的启动过程
    //可以自行实现相关的spring.factories以及实现类，从而影响springboot启动流程的效果，web
环境调试时，默认有8个
    setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
    //获取主类，实际上就是main方法入口所在类，内部通过new
RuntimeException().getStackTrace()获取异常堆栈
    //在异常堆栈中找main()，然后通过Class.forName(stackTraceElement.getClassName())获
取main方法所在的类
    this.mainApplicationClass = deduceMainApplicationClass();
}

public ConfigurableApplicationContext run(String... args) {//此处时第3步run方法部分
    long startTime = System.nanoTime();
    DefaultBootstrapContext bootstrapContext = createBootstrapContext();//用于启动
期间的启动上下文
    ConfigurableApplicationContext context = null;-
        configureHeadlessProperty();//System.setProperty设置java.awt.headless属
性，调试时为true
    //这里内部让然使用spring.factories的机制加载了SpringApplicationRunListener.class
    //SpringApplicationRunListeners是对所有SpringApplicationRunListener的封装，调用
SpringApplicationRunListeners
    //事件方法相当于调用其内部所有SpringApplicationRunListener对应的事件方法，在
webstarter环境调试时，只有一个
```

```
    //EventPublishingRunListener，但是该listener内部的initialMulticaster又包装了之前的8个ApplicationListener
    //于是EventPublishingRunListener的事件方法调用时，会采取如下的形式发送事件
initialMulticaster
    //  .multicastEvent(new ApplicationStartingEvent(bootstrapContext, this.application, this.args))
    //每种不同类型的事件发生时，其multicastEvent(ApplicationEvent event)的event实现类就不一样，比如
    //ApplicationEnvironmentPreparedEvent/ApplicationContextInitializedEvent等，这些过程中有的会向容器注入bean
    //所以这样看来，用户基本没必要实现SpringApplicationRunListener，实现ApplicationListener基本可满足需要
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting(bootstrapContext, this.mainApplicationClass);
    try {
        //对命令行程序输入参数进行封装，内部使用了Source，而Source使用了SimpleCommandLinePropertySource
        //其可以方便的解析option arguments(--foo=bar)和non-option arguments(没有--前缀的参数)
        ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
        //主要过程
        //1.内部创建了environment以及相关的PropertySources
        //2.执行了listeners.environmentPrepared，触发listener的相关事件（调试时其注册了两个PropertySource，其中一个
        //是RandomValuePropertySource，提供随机数的数据源
        ConfigurableEnvironment environment = prepareEnvironment(listeners, bootstrapContext, applicationArguments);
        //System.setProperties配置spring.beaninfo.ignore系统属性，web环境调试时默认设置为true;效果暂未分析，需要再说
        configureIgnoreBeanInfo(environment);
        //打印banner,此处跟踪后，发现Banner及版本信息是通过System.out进行打印的
        //同时发现SpringApplication中的logger=LogFactory.getLog(SpringApplication.class)其在webstarter环境中为
        //org.apache.commons.logging.LogAdapter.Slf4jLocationAwareLog,位于spring-jcl包中，其用的并不是
        //默认的logback的实现类（该logger暂不清楚使用在哪，此处并未使用，只是顺便分析了一下；
        //但可以直到早期logger使用的并不是logback，但容器启动后用户通过slf4j接口得到的logger确实是logback实现类对象）
        Banner printedBanner = printBanner(environment);
        //依据WebApplicationType创建容器，在webstarter下是AnnotationConfigServletWebServerApplicationContext
        //构造函数中主要创建了AnnotatedBeanDefinitionReader ClassPathBeanDefinitionScanner
        //特别的AnnotatedBeanDefinitionReader通过

 //AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry)注册了
        //ConfigurationClassPostProcessor（后续加载BeanDefintion的核心类），AutowiredAnnotationBeanPostProcessor
        //（后续自动装配的核心类）等
        //SERVLET：AnnotationConfigServletWebServerApplicationContext
        //REACTIVE：AnnotationConfigReactiveWebServerApplicationContext
        //default：AnnotationConfigApplicationContext   WebApplicationType的枚举值就还剩一个NONE，实际这里就是NONE
        //这部分源码分析后面跟着该类的类图
        context = createApplicationContext();
```

```
        context.setApplicationStartup(this.applicationStartup);//设置
applicationStartup，不重要
        //此段过程单独放在后面分析
        prepareContext(bootstrapContext, context, environment, listeners,
applicationArguments, printedBanner);
        //在webstarter环境中，该方法一方面向容器中注册
ApplicationContextClosedListener，并且context被到
        //shutdownHook.contexts中，shutdownHook是个静态变量，相当于context被强引用了
        //其次执行AbstractApplicationContext.refresh()方法，这部分完全就是spring容器的
refresh()，但是需要注意
        //refresh()中调用的一些方法会被子类
AnnotationConfigServletWebServerApplicationContext重写，后面单独分析
        refreshContext(context);//在webstarter环境中，这个地方执行完后，内嵌的tomcat就
已经启动了
        afterRefresh(context, applicationArguments);//目前是个空方法
        Duration timeTakenToStartup = Duration.ofNanos(System.nanoTime() -
startTime);
        if (this.logStartupInfo) {
            //调试时，可以看到此处内部的logger已经是logback的实例了
            //打印Started Application in 358.274 seconds (JVM running for
374.834)，默认颜色效果和
            //jcl包中的Slf4jLocationAwareLog是一样的
            new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
timeTakenToStartup);
        }
        //触发监听器事件
        listeners.started(context, timeTakenToStartup);
        //从context中获取ApplicationRunner，CommandLineRunner类型的Bean
        //将获取的Bean进行优先级排序，然后循环执行run方法，默认webstarter数量为0，相当于什
么也没做
        callRunners(context, applicationArguments);
    }
    catch (Throwable ex) {
        handleRunFailure(context, ex, listeners);
        throw new IllegalStateException   (ex);
    }
    try {
        Duration timeTakenToReady = Duration.ofNanos(System.nanoTime() -
startTime);
        listeners.ready(context, timeTakenToReady);//触发监听器事件
    }
    catch (Throwable ex) {
        handleRunFailure(context, ex, null);
        throw new IllegalStateException(ex);
    }
    return context;
}

private void prepareContext(DefaultBootstrapContext bootstrapContext,
ConfigurableApplicationContext context,ConfigurableEnvironment environment,
SpringApplicationRunListeners listeners,
          ApplicationArguments applicationArguments, Banner printedBanner) {
    //给容器设置environment,细节方面还包括AnnotatedBeanDefinitionReader和
    //ClassPathBeanDefinitionScanner中的environment
    context.setEnvironment(environment);
    //从调试webstarter调试来看，主要是设置ConversionService
```

```
 //context.getBeanFactory().setConversionService(context.getEnvironment().getConversionService())
    //内部默认有137个转换器，可应用于大多数的情况：String->Number Short->String String->YearMonth等
    postProcessApplicationContext(context);
    //底层利用了GenericTypeResolver/ResolvableType解析泛型信息进行校验context是否是Initializer实例可接受的泛型实例
    //校验成功后执行ApplicationContextInitializer.initialize(context)
    applyInitializers(context);
    //触发监听器事件
    listeners.contextPrepared(context);
    //触发bootstrapContext的事件，但webstarter调试时内部没有listener，实际相当于什么也没做
    bootstrapContext.close(context);
    if (this.logStartupInfo) {//在webstarter下此处打印使用的是Slf4jLocationAwareLog
        //打印...INFO...Starting Webenv84jarApplication using Java 1.8.0_311 on ..
        logStartupInfo(context.getParent() == null);
        //打印...INFO...No active profile set, falling back to 1 default profile: "default"
        logStartupProfileInfo(context);
    }
    // Add boot specific singleton beans
    ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
    //向容器注册一个Bean(封装了命令行参数)
    beanFactory.registerSingleton("springApplicationArguments", applicationArguments);
    if (printedBanner != null) {
        //向容器注册一个Bean(封装了Banner)
        beanFactory.registerSingleton("springBootBanner", printedBanner);
    }
    if (beanFactory instanceof AbstractAutowireCapableBeanFactory) {
        //webstarter调试时，allowCircularReferences=false;高版本springboot默认禁用了循环引用
        ((AbstractAutowireCapableBeanFactory) beanFactory).setAllowCircularReferences(this.allowCircularReferences);
        //webstarter调试时，allowBeanDefinitionOverriding值为false
        if (beanFactory instanceof DefaultListableBeanFactory) {
            ((DefaultListableBeanFactory) beanFactory)

.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
        }
    }
    if (this.lazyInitialization) {//懒加载相关，默认为false
        context.addBeanFactoryPostProcessor(new LazyInitializationBeanFactoryPostProcessor());
    }
    // Load the sources
    Set<Object> sources = getAllSources();//Sources存放的就是主类
    Assert.notEmpty(sources, "Sources must not be empty");
    //底层中：Loads bean definitions from underlying sources, including XML and JavaConfig.
    //Acts as a simple facade over AnnotatedBeanDefinitionReader, XmlBeanDefinitionReader and
    //ClassPathBeanDefinitionScanner.
    //在webstarter环境下调试结果来看，只是将主类注册到容器的beanDefintionMap和beanDefintionNames，并未注册实例Bean
```

```
    load(context, sources.toArray(new Object[0]));
    //触发监听器事件
    listeners.contextLoaded(context);
}
```

AnnotationConfigServletWebServerApplicationContext refresh分析（在webstarter环境下）：

AnnotationConfigServletWebServerApplicationContext.refresh()调用的实际是spring中AbstractApplicationContext的refresh(),虽然子类在refresh()时都是调用的父类AbstractApplicationContext的该方法，但是该方法中调用的一些其它方法会被子类重写，这里和AnnotationConfigApplicationContext进行比较，主要分析重大区别

区别1.refresh方法中的postProcessBeanFactory(beanFactory)

```
    protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
        super.postProcessBeanFactory(beanFactory);//调用的是ServletWebServerApplicationContext中的实现
        //下面的basePackages和annotatedClasses在webstarter环境下调试时都不执行，暂不分析了
        if (this.basePackages != null && this.basePackages.length > 0) {
            this.scanner.scan(this.basePackages);
        }
        if (!this.annotatedClasses.isEmpty()) {

this.reader.register(ClassUtils.toClassArray(this.annotatedClasses));
        }
    }
    //上面super.postProcessBeanFactory(beanFactory)的内容
    protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
        //添加WebApplicationContextServletContextAwareProcessor，可让实现ServletContextAware接口的Bean感知到
        //ServletContext
        beanFactory.addBeanPostProcessor(new WebApplicationContextServletContextAwareProcessor(this));
        beanFactory.ignoreDependencyInterface(ServletContextAware.class);//该类不允许自动装配
        registerWebApplicationScopes();//向beanFactory的scopes域中显示加入request/session scope
    }
```

区别2.refresh方法中的onRefresh()// Initialize other special beans in specific context subclasses.

```
    //AnnotationConfigApplicationContext在该方法中什么也没做，
    //而AnnotationConfigServletWebServerApplicationContext一方面设置了ThemeSource，另一方面createWebServer
    protected void onRefresh() {
        //protected void onRefresh() {
        //  实际为ResourceBundleThemeSource，核心有一个Theme getTheme(String themeName)方法
        //  可以解析Theme特定的消息、代码、路径，国际化等，未深入分析
        //  this.themeSource = UiApplicationContextUtils.initThemeSource(this);
        //}
        super.onRefresh();
        try {
            createWebServer();//创建web服务器，这里是内置tomcat
        }
        catch (Throwable ex) {
            throw new ApplicationContextException("Unable to start web server", ex);
        }
```

```
    }
    private void createWebServer() {
        WebServer webServer = this.webServer;
        ServletContext servletContext = getServletContext();
        if (webServer == null && servletContext == null) {//默认就全为null
            StartupStep createWebServer =
this.getApplicationStartup().start("spring.boot.webserver.create");
            //protected ServletWebServerFactory getWebServerFactory() {
            //    // Use bean names so that we don't consider the hierarchy
            //    //从beanFactory中获取类型为ServletWebServerFactory的beanNames(从
beanDefinitionNames和manualSingletonNames中获取)
            //    //这里的实现类在调试时为TomcatServletWebServerFactory，查找引用可以看
到其所在配置类为ServletWebServerFactoryConfiguration
            //    //查找配置类引用时可以看到在
ServletWebServerFactoryAutoConfiguration自动配置类的注解上通过@Import引入了
            //    //ServletWebServerFactoryConfiguration，下面会贴出这里的相关代码
            //    String[] beanNames =
getBeanFactory().getBeanNamesForType(ServletWebServerFactory.class);
            //    if (beanNames.length == 0) {//异常检测
            //        throw new ApplicationContextException("Unable to start
ServletWebServerApplicationContext due to missing "
            //                + "ServletWebServerFactory bean.");
            //    }
            //    if (beanNames.length > 1) {//异常检测
            //      throw new ApplicationContextException("Unable to start
ServletWebServerApplicationContext due to multiple "
            //                + "ServletWebServerFactory beans : " +
StringUtils.arrayToCommaDelimitedString(beanNames));
            //    }
            //    //调用BeanFacotry的getBean方法提前创建对象，并放入BeanFactory
            //    //创建fatctory过程中还会创建WebServerFactoryCustomizer类型的对象，实
际为ServletWebServerFactoryCustomizer
            //    return getBeanFactory().getBean(beanNames[0],
ServletWebServerFactory.class);
            //}
            ServletWebServerFactory factory = getWebServerFactory();
            createWebServer.tag("factory", factory.getClass().toString());
            //getSelfInitializer():retrun this::selfInitialize;等同于
selfInitialize
            //private void selfInitialize(ServletContext servletContext) throws
ServletException {
            //    prepareWebApplicationContext(servletContext);
            //    registerApplicationScope(servletContext);
            //
WebApplicationContextUtils.registerEnvironmentBeans(getBeanFactory(),
servletContext);
            //    for (ServletContextInitializer beans :
getServletContextInitializerBeans()) {
            //        beans.onStartup(servletContext);
            //    }
            //}
            //创建WebServer的核心过程，暂不准备进行深入分析
            //public WebServer getWebServer(ServletContextInitializer...
initializers) {
            //    if (this.disableMBeanRegistry) {
            //        Registry.disableRegistry();
            //    }
            //    Tomcat tomcat = new Tomcat();
```

```
            //    File baseDir = (this.baseDirectory != null) ?
this.baseDirectory : createTempDir("tomcat");
            //    tomcat.setBaseDir(baseDir.getAbsolutePath());
            //    Connector connector = new Connector(this.protocol);
            //    connector.setThrowOnFailure(true);
            //    tomcat.getService().addConnector(connector);
            //    customizeConnector(connector);
            //    tomcat.setConnector(connector);
            //    tomcat.getHost().setAutoDeploy(false);
            //    configureEngine(tomcat.getEngine());
            //    for (Connector additionalConnector :
this.additionalTomcatConnectors) {
            //        tomcat.getService().addConnector(additionalConnector);
            //    }
            //    prepareContext(tomcat.getHost(), initializers);
            //    return getTomcatWebServer(tomcat);
            //}
            this.webServer = factory.getWebServer(getSelfInitializer());
            createWebServer.end();
            //注册单例Bean，这里的两个单例都是和Lifecycle相关
            getBeanFactory().registerSingleton("webServerGracefulShutdown",
                    new WebServerGracefulShutdownLifecycle(this.webServer));
            //特别的，这个WebServerStartStopLifecycle单例有一个start方法
            //public void start() {
            //    this.webServer.start();
            //    this.running = true;
            //    this.applicationContext
            //            .publishEvent(new
ServletWebServerInitializedEvent(this.webServer, this.applicationContext));
            //}
            //在refresh()方法末尾会调用finishRefresh()方法，该方法又会调用一个
getLifecycleProcessor().onRefresh()，这个方法又会最终执行
            //Lifecycle的start方法，其中一个就是WebServerStartStopLifecycle.start方
法：
            //public void start() {
            //    //真正启动web服务器；此处执行之前所有Bean实例已经有BeanFactory创建好了
            //    //在这个webServer.start()内部，调用了
addPreviouslyRemovedConnectors()：该方法执行完，用netstat就能看到tomcat监听了服务端口
            //    this.webServer.start();
            //    this.running = true;
            //    this.applicationContext
            //            .publishEvent(new
ServletWebServerInitializedEvent(this.webServer, this.applicationContext));
            //}
            getBeanFactory().registerSingleton("webServerStartStop",
                    new WebServerStartStopLifecycle(this, this.webServer));
        }
        else if (servletContext != null) {
            try {
                getSelfInitializer().onStartup(servletContext);
            }
            catch (ServletException ex) {
                throw new ApplicationContextException("Cannot initialize servlet
context", ex);
            }
        }
        initPropertySources();//依据Environment处理属性呢
    }
```

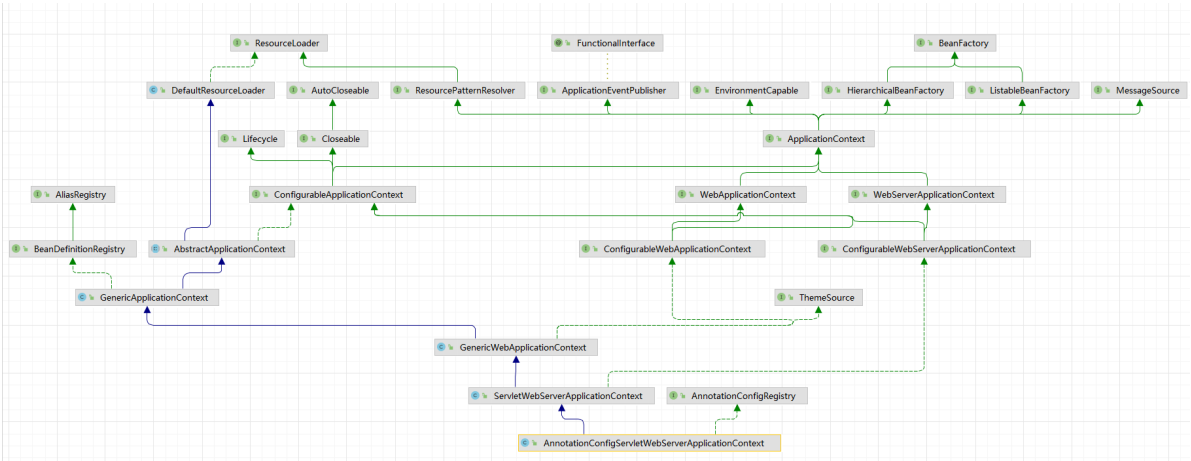
```
    @Configuration(proxyBeanMethods = false)
    @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
    @ConditionalOnClass(ServletRequest.class)
    @ConditionalOnWebApplication(type = Type.SERVLET)
    @EnableConfigurationProperties(ServerProperties.class)
    @Import({
ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class,
            ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,//引入
Tomcat相关Factory
            ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
            ServletWebServerFactoryConfiguration.EmbeddedUndertow.class })
    //此类是自动配置类，在spring.factories中有
org.springframework.boot.autoconfigure.EnableAutoConfiguration=...,

//org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAuto
Configuration,...
    public class ServletWebServerFactoryAutoConfiguration {...}

    @Configuration(proxyBeanMethods = false)
    class ServletWebServerFactoryConfiguration {
        @Configuration(proxyBeanMethods = false)
        @ConditionalOnClass({ Servlet.class, Tomcat.class, UpgradeProtocol.class
})
        @ConditionalOnMissingBean(value = ServletWebServerFactory.class, search
= SearchStrategy.CURRENT)
        static class EmbeddedTomcat {//ServletWebServerFactoryAutoConfiguration上
@Import引入的TomcatFactory
            @Bean
            TomcatServletWebServerFactory tomcatServletWebServerFactory(
                    ObjectProvider<TomcatConnectorCustomizer>
connectorCustomizers,
                    ObjectProvider<TomcatContextCustomizer> contextCustomizers,
                    ObjectProvider<TomcatProtocolHandlerCustomizer<?>>
protocolHandlerCustomizers) {
                TomcatServletWebServerFactory factory = new
TomcatServletWebServerFactory();
                factory.getTomcatConnectorCustomizers()

.addAll(connectorCustomizers.orderedStream().collect(Collectors.toList()));
                factory.getTomcatContextCustomizers()

.addAll(contextCustomizers.orderedStream().collect(Collectors.toList()));
                factory.getTomcatProtocolHandlerCustomizers()

.addAll(protocolHandlerCustomizers.orderedStream().collect(Collectors.toList()))
;
                return factory;
            }
            ...
        }
    }


补充.refresh方法中的invokeBeanFactoryPostProcessors
    //内部最重要的是ConfigurationClassPostProcessor类的
postProcessBeanDefinitionRegistry后置处理，
```

```
    //导入了所有的BeanDefintion，下面其核心调用方法;由于在Spring容器部分未深入，此处进一步分析理解
    public void processConfigBeanDefinitions(BeanDefinitionRegistry registry) {
        List<BeanDefinitionHolder> configCandidates = new ArrayList<>();
        String[] candidateNames = registry.getBeanDefinitionNames();
        //从registry(就是beanFactory)已注册的BeanDefintion中获取配置类（@Configuration标记的类），
        //并加入configCandidates，此处在webstarter调试时，获取的就是主类
        for (String beanName : candidateNames) {
            BeanDefinition beanDef = registry.getBeanDefinition(beanName);
            if (beanDef.getAttribute(ConfigurationClassUtils.CONFIGURATION_CLASS_ATTRIBUTE) != null) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Bean definition has already been processed as a configuration class: " + beanDef);
                }
            }
            else if (ConfigurationClassUtils.checkConfigurationClassCandidate(beanDef, this.metadataReaderFactory)) {
                configCandidates.add(new BeanDefinitionHolder(beanDef, beanName));
            }
        }
        // Return immediately if no @Configuration classes were found
        if (configCandidates.isEmpty()) {
            return;
        }
        // Sort by previously determined @Order value, if applicable
        configCandidates.sort((bd1, bd2) -> {
            int i1 = ConfigurationClassUtils.getOrder(bd1.getBeanDefinition());
            int i2 = ConfigurationClassUtils.getOrder(bd2.getBeanDefinition());
            return Integer.compare(i1, i2);
        });
        // Detect any custom bean name generation strategy supplied through the enclosing application
        // context  此处到下一个注释之间在webstarter调试时相当于什么也没做,但这部内容应该是和自定义beanName生成相关
        SingletonBeanRegistry sbr = null;
        if (registry instanceof SingletonBeanRegistry) {
            sbr = (SingletonBeanRegistry) registry;
            if (!this.localBeanNameGeneratorSet) {
                BeanNameGenerator generator = (BeanNameGenerator) sbr.getSingleton(

AnnotationConfigUtils.CONFIGURATION_BEAN_NAME_GENERATOR);
                if (generator != null) { //调试为null
                    this.componentScanBeanNameGenerator = generator;
                    this.importBeanNameGenerator = generator;
                }
            }
        }
        if (this.environment == null) {
            this.environment = new StandardEnvironment();
        }
        //Parse each @Configuration class
```

```
        //ConfigurationClassParser:Parses a Configuration class definition,
populating a collection of
        //ConfigurationClass objects (parsing a single Configuration class may
result in any number of
        //ConfigurationClass objects because one Configuration class may import
another using the
        //Import annotation).
        ConfigurationClassParser parser = new ConfigurationClassParser(
                this.metadataReaderFactory, this.problemReporter,
this.environment,
                this.resourceLoader, this.componentScanBeanNameGenerator,
registry);

        Set<BeanDefinitionHolder> candidates = new LinkedHashSet<>
(configCandidates);
        Set<ConfigurationClass> alreadyParsed = new HashSet<>
(configCandidates.size());
        do {
            StartupStep processConfig =
this.applicationStartup.start("spring.context.config-classes.parse");
            //解析配置类，之后的parser.getConfigurationClasses 才会获取到相关配置类，见
后文详细分析
            //执行完后parser解析出来的配置类信息存放在parser中，此时context中还没加载所有
相关BeanDefintion，但有一部分已经加载了
            parser.parse(candidates);
            //遍历parse出来的所有配置类，进行校验：关键字：proxyBeanMethods、final、
@Bean标注的方法isOverridable、isStatic
            parser.validate();
            Set<ConfigurationClass> configClasses = new LinkedHashSet<>
(parser.getConfigurationClasses());
            configClasses.removeAll(alreadyParsed);
            // Read the model and create bean definitions based on its content
            if (this.reader == null) {//构建bean defintion reader
                this.reader = new ConfigurationClassBeanDefinitionReader(
                        registry, this.sourceExtractor, this.resourceLoader,
this.environment,
                        this.importBeanNameGenerator,
parser.getImportRegistry());
            }
            //依据配置类加载BeanDefintion,见后文详细分析
            this.reader.loadBeanDefinitions(configClasses);
            alreadyParsed.addAll(configClasses);
            processConfig.tag("classCount", () ->
String.valueOf(configClasses.size())).end();
            candidates.clear();
            if (registry.getBeanDefinitionCount() > candidateNames.length) {
                String[] newCandidateNames = registry.getBeanDefinitionNames();
                Set<String> oldCandidateNames = new HashSet<>
(Arrays.asList(candidateNames));
                Set<String> alreadyParsedClasses = new HashSet<>();
                for (ConfigurationClass configurationClass : alreadyParsed) {

alreadyParsedClasses.add(configurationClass.getMetadata().getClassName());
                }
                for (String candidateName : newCandidateNames) {
                    if (!oldCandidateNames.contains(candidateName)) {
                        BeanDefinition bd =
registry.getBeanDefinition(candidateName);
```

```
                        if
(ConfigurationClassUtils.checkConfigurationClassCandidate(bd,
this.metadataReaderFactory) &&

!alreadyParsedClasses.contains(bd.getBeanClassName())) {
                            candidates.add(new BeanDefinitionHolder(bd,
candidateName));
                        }
                    }
                }
                candidateNames = newCandidateNames;
            }
        }
        while (!candidates.isEmpty());//在webstarter环境中只执行了一次，没有多次循环

        // Register the ImportRegistry as a bean in order to support ImportAware
@Configuration classes
        if (sbr != null && !sbr.containsSingleton(IMPORT_REGISTRY_BEAN_NAME))
{//webstarter此处执行
            sbr.registerSingleton(IMPORT_REGISTRY_BEAN_NAME,
parser.getImportRegistry());
        }
        if (this.metadataReaderFactory instanceof CachingMetadataReaderFactory)
{//webstarter此处不执行
            // Clear cache in externally provided MetadataReaderFactory; this is
a no-op
            // for a shared cache since it'll be cleared by the
ApplicationContext.
            ((CachingMetadataReaderFactory)
this.metadataReaderFactory).clearCache();
        }
    }
```

AnnotationConfigServletWebServerApplicationContext



解析配置类

此处是解析配置类前文parser.parse(candidates)的核心过程
ConfigurationClassPostProcessor中
processConfigBeanDefinitions(BeanDefinitionRegistry registry)方法调用了
...
//这个执行过程只有一个ConfigurationClassParser，但递归处理时存在多个ConfigurationClass

```
//执行完后parser解析出来的配置类信息存放在parser中，此时context中还没加载所有相关BeanDefintion(例外：ComponentScan的组件会立刻注册BeanDefintion）
parser.parse(candidates) //parser是ConfigurationClassParser，candidates是Set<BeanDefinitionHolder>，这里是其调用栈
    parse(((AnnotatedBeanDefinition) bd).getMetadata(), holder.getBeanName())
        processConfigurationClass(new ConfigurationClass(metadata, beanName), DEFAULT_EXCLUSION_FILTER)
            //此处涉及的条件评估体系，后文详细分析，配置类在解析的时候都是PARSE_CONFIGURATION阶段，只有标注了@ComponentScan的类才会是
            //ConfigurationPhase.REGISTER_BEAN
            this.conditionEvaluator.shouldSkip(configClass.getMetadata(), ConfigurationPhase.PARSE_CONFIGURATION)为true就return
            ...
            do {
                sourceClass = doProcessConfigurationClass(configClass, sourceClass, filter);
            }
            while (sourceClass != null)
            ...
    this.deferredImportSelectorHandler.process()//此处处理DeferredImportSelector，这里可以结合后文processImports部分理解
parser.validate()//遍历parse出来的所有配置类，进行校验；关键字：proxyBeanMethods、final、@Bean标注的方法isOverridable、isStatic
configClasses = new LinkedHashSet<>(parser.getConfigurationClasses())
configClasses.removeAll(alreadyParsed)
if (this.reader == null) {
    this.reader = new ConfigurationClassBeanDefinitionReader(
        registry, this.sourceExtractor, this.resourceLoader, this.environment,
        this.importBeanNameGenerator, parser.getImportRegistry());
}
//依据配置类加载所有剩余的BeanDefitnion
this.reader.loadBeanDefinitions(configClasses)
...

protected final SourceClass doProcessConfigurationClass(
            ConfigurationClass configClass, SourceClass sourceClass,
Predicate<String> filter)
            throws IOException {

    if (configClass.getMetadata().isAnnotated(Component.class.getName())) {
        // Recursively process any member (nested) classes first
        //首先对内嵌的@Configuration进行递归处理，总是先解析内部的配置类，再解析外部配置类
        processMemberClasses(configClass, sourceClass, filter);
    }
    // Process any @PropertySource annotations
    // @PropertySource:Annotation providing a convenient and declarative mechanism for adding a PropertySource to Spring's
    // Environment. To be used in conjunction with @Configuration classes.
    //此处处理注解类上的@PropertySource,效果是在environment的PropertySources加入新的propertySource
    for (AnnotationAttributes propertySource : AnnotationConfigUtils.attributesForRepeatable(
        sourceClass.getMetadata(), PropertySources.class,
        org.springframework.context.annotation.PropertySource.class)) {
        if (this.environment instanceof ConfigurableEnvironment) {
            processPropertySource(propertySource);
        }
        else {
```

```
            logger.info("Ignoring @PropertySource annotation on [" +
sourceClass.getMetadata().getClassName() +
                        "]. Reason: Environment must implement
ConfigurableEnvironment");
        }
    }
    // Process any @ComponentScan annotations
    Set<AnnotationAttributes> componentScans =
AnnotationConfigUtils.attributesForRepeatable(
        sourceClass.getMetadata(), ComponentScans.class, ComponentScan.class);//
所有ComponentScan注解信息
    if (!componentScans.isEmpty() &&
        !this.conditionEvaluator.shouldSkip(sourceClass.getMetadata(),
ConfigurationPhase.REGISTER_BEAN)) {
        for (AnnotationAttributes componentScan : componentScans) {
            // The config class is annotated with @ComponentScan -> perform the
scan immediately
            //看调试时实际数据，此处只有用户定义的包下的组件（个人调试时就是自定义的
@Controller Hello组件）
            Set<BeanDefinitionHolder> scannedBeanDefinitions =
                this.componentScanParser.parse(componentScan,
sourceClass.getMetadata().getClassName());
            // Check the set of scanned definitions for any further config
classes and parse recursively if needed
            //扫描出来的BeanDefintion如果存在配置类，那么进一步递归解析这些配置类
            for (BeanDefinitionHolder holder : scannedBeanDefinitions) {
                //在AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata,
definitionHolder, registry)产生这种
                //代理行为时，在一定条件下会设置其返回的proxyBeanDefintion的
OriginatingBeanDefinition为一开始的
                //definitionHolder中的BeanDefintion，所以下面的行为时为了获取一开始的原
始BeanDefintion
                BeanDefinition bdCand =
holder.getBeanDefinition().getOriginatingBeanDefinition();
                if (bdCand == null) {
                    bdCand = holder.getBeanDefinition();
                }
                //如果BeanDefintion是配置类，递归解析
                if
(ConfigurationClassUtils.checkConfigurationClassCandidate(bdCand,
this.metadataReaderFactory)) {
                    //个人编写的@Controller Hello组件也会走这里,其满足上面的检测，会进入
parse；说明这些组件也会被视为潜在的配置类
                    //在这些类上也可以使用@PropertySource @ComponentScan @Import
@ImportResource @Bean等，但更好方式是写在@Configuration
                    //此处方法签名和前面的不一样 parse(@Nullable String className,
String beanName) 逻辑也略微不一样
                    //内部使用了SimpleMetadataReader：MetadataReader implementation
based on an ASM ClassReader.其由
                    //SimpleMetadataReaderFactory.getMetadataReader(className)获
取，而这最终又会通过org.springframework.asm包中的
                    //ClassVisitor、ClassReader对类进行解析，最后一个resource(类path
是主要区别)对应一个MetadataReader（外部类和内部类
                    //会分别创建一个对应的MetadataReader,内部类例子：
org.example.controller.Hello$NestedClass2会创建一个
                    //对应MetadataReader,通常发生在processMemberClasses方法中），并存
入cache
                    //下面两个注释代码是紧挨着的parse的核心逻辑
```

```
                    //MetadataReader reader =
this.metadataReaderFactory.getMetadataReader(className); 获取类文件对应的reader
                    //processConfigurationClass(new ConfigurationClass(reader,
beanName), DEFAULT_EXCLUSION_FILTER); 进行递归
                    parse(bdCand.getBeanClassName(), holder.getBeanName());//这里
使用原始的BeanName
                }
            }
        }
    }
    // Process any @Import annotations
    //@Import:Indicates one or more component classes to import — typically
@Configuration classes.
    //getImports解析并获取导入的组件类，这些组件类可能仍然注解了@Import，此时，就会递归处理
@Import导入
    processImports(configClass, sourceClass, getImports(sourceClass), filter,
true);
    // Process any @ImportResource annotations  该注解主要用于导入xml/groovy形式的
spring配置文件
    //@ImportResource : By default, arguments to the value attribute will be
processed using a GroovyBeanDefinitionReader
    //if ending in ".groovy"; otherwise, an XmlBeanDefinitionReader will be used
to parse Spring <beans/> XML files.
    AnnotationAttributes importResource =
        AnnotationConfigUtils.attributesFor(sourceClass.getMetadata(),
ImportResource.class);
    if (importResource != null) {
        String[] resources = importResource.getStringArray("locations");
        Class<? extends BeanDefinitionReader> readerClass =
importResource.getClass("reader");
        for (String resource : resources) {
            String resolvedResource =
this.environment.resolveRequiredPlaceholders(resource);
            configClass.addImportedResource(resolvedResource, readerClass);
        }
    }
    // Process individual @Bean methods
    //这个过程中会Try reading the class file via ASM for deterministic declaration
order，即尝试通过ASM获取确定性的声明顺序，未递归
    Set<MethodMetadata> beanMethods = retrieveBeanMethodMetadata(sourceClass);
    for (MethodMetadata methodMetadata : beanMethods) {
        configClass.addBeanMethod(new BeanMethod(methodMetadata, configClass));
    }
    // Process default methods on interfaces
    //递归处理接口中的默认方法， A default method or other concrete method on a Java
8+ interface
    //内部调用了retrieveBeanMethodMetadata(ifc)，说明这些方法还要被标注@Bean才会被处理；
另外还判断了，不允许为抽象方法
    processInterfaces(configClass, sourceClass);
    // Process superclass, if any
    //结合该方法外部的do while循环，递归处理父类；说明先处理自身，再处理父类
    if (sourceClass.getMetadata().hasSuperClass()) {
        String superclass = sourceClass.getMetadata().getSuperClassName();
        if (superclass != null && !superclass.startsWith("java") &&
            !this.knownSuperclasses.containsKey(superclass)) {
            this.knownSuperclasses.put(superclass, configClass);
            // Superclass found, return its annotation metadata and recurse
            return sourceClass.getSuperClass();
```

```
        }
    }
    // No superclass -> processing is complete
    return null;
}
```

processMemberClasses

```
//递归处理内部配置类，再处理外部类;
private void processMemberClasses(ConfigurationClass configClass, SourceClass
sourceClass,
            Predicate<String> filter) throws IOException {
    Collection<SourceClass> memberClasses = sourceClass.getMemberClasses();//获取
所有内部类
    if (!memberClasses.isEmpty()) {
        List<SourceClass> candidates = new ArrayList<>(memberClasses.size());
        for (SourceClass memberClass : memberClasses) {
            //检测内部类是否为候选配置类：类上标注Component、ComponentScan、Import、
ImportResource，或类内含有标注@Bean的方法
            if
(ConfigurationClassUtils.isConfigurationCandidate(memberClass.getMetadata()) &&
                //个人理解是检测内部类和外部类类名是否一致，candidates只在不一致时添加

 !memberClass.getMetadata().getClassName().equals(configClass.getMetadata().getC
lassName())) {
                candidates.add(memberClass);
            }
        }
        OrderComparator.sort(candidates);//排序
        for (SourceClass candidate : candidates) {
            if (this.importStack.contains(configClass)) {//importStack用于检测循环
引入配置类的情况
                this.problemReporter.error(new
CircularImportProblem(configClass, this.importStack));
            }
            else {
                this.importStack.push(configClass);
                try {//将内部类进行递归解析，此处是parser内部方法，条件评估时是
ConfigurationPhase.PARSE_CONFIGURATION
                    //只要外部类被parse处理了，那么内部类只要有可能是候选配置类，那么就一定
会被处理（处理过程中会处理Conditional等条件注解）

 processConfigurationClass(candidate.asConfigClass(configClass), filter);
                }
                finally {
                    this.importStack.pop();
                }
            }
        }
    }
}
//检测到类上标注有Component/ComponentScan/Import/ImportResource注解、或类内部标注了
@Bean的方法 就视其为配置类候选
public static boolean isConfigurationCandidate(AnnotationMetadata metadata) {
    // Do not consider an interface or an annotation... 注解判断时也是接口，所以这里检
测接口和注解，不允许是接口和注解
```

```
    if (metadata.isInterface()) {
        return false;
    }
    // Any of the typical annotations found?
    //检测到Component、ComponentScan、Import、ImportResource 就视为配置类候选
    for (String indicator : candidateIndicators) {
        if (metadata.isAnnotated(indicator)) {
            return true;
        }
    }
    // Finally, let's look for @Bean methods...
    try {
        return metadata.hasAnnotatedMethods(Bean.class.getName());//如果存在标注了
@Bean的方法，就是为配置类候选
    }
    catch (Throwable ex) {
        if (logger.isDebugEnabled()) {
            logger.debug("Failed to introspect @Bean methods on class [" +
metadata.getClassName() + "]: " + ex);
        }
        return false;
    }
}
```

processPropertySource

```
//向environment中注入PropertySource
private void processPropertySource(AnnotationAttributes propertySource) throws
IOException {
    //一开始部分是获取当前处理的PropertySource注解属性信息
    String name = propertySource.getString("name");
    if (!StringUtils.hasLength(name)) {
        name = null;
    }
    String encoding = propertySource.getString("encoding");
    if (!StringUtils.hasLength(encoding)) {
        encoding = null;
    }
    String[] locations = propertySource.getStringArray("value");
    Assert.isTrue(locations.length > 0, "At least one @PropertySource(value)
location is required");
    //这个表明每一个PropertySource都可以设置如果资源文件未找到时，是否可以忽略资源未找到错误
    boolean ignoreResourceNotFound =
propertySource.getBoolean("ignoreResourceNotFound");
    //用户可以在PropertySource注解中指定创建PropertySource类对象用的工厂；未指定时默认为
DefaultPropertySourceFactory
    Class<? extends PropertySourceFactory> factoryClass =
propertySource.getClass("factory");
    PropertySourceFactory factory = (factoryClass == PropertySourceFactory.class
?
                                     DEFAULT_PROPERTY_SOURCE_FACTORY :
BeanUtils.instantiateClass(factoryClass));
    for (String location : locations) {
        try {
            //解析location中的${...} placeholder
            String resolvedLocation =
this.environment.resolveRequiredPlaceholders(location);
```

```
            Resource resource =
this.resourceLoader.getResource(resolvedLocation);//依据位置信息获取资源
            //向environment中注入PropertySource(可能会和已有的名称相同的ProperySource
合并)，后文详细分析
            addPropertySource(factory.createPropertySource(name, new
EncodedResource(resource, encoding)));
        }
        catch (IllegalArgumentException | FileNotFoundException |
UnknownHostException | SocketException ex) {
            // Placeholders not resolvable or resource not found when trying to
open it
            if (ignoreResourceNotFound) {
                if (logger.isInfoEnabled()) {
                    logger.info("Properties location [" + location + "] not
resolvable: " + ex.getMessage());
                }
            }
            else {
                throw ex;
            }
        }
    }
}
private void addPropertySource(PropertySource<?> propertySource) {
    String name = propertySource.getName();//获取待处理propertySource的名称
    MutablePropertySources propertySources = ((ConfigurableEnvironment)
this.environment).getPropertySources();//获取environment中所有PropertySources
    if (this.propertySourceNames.contains(name)) {//已存在名称相同的propertySource;
这里属于ConfigurationClassParser的检查
        // We've already added a version, we need to extend it
        //查看是否已存在同名的，这里属于environment的propertySources检查
        PropertySource<?> existing = propertySources.get(name);
        if (existing != null) {
            //withResourceName尝试加上或覆盖一个名字
            PropertySource<?> newSource = (propertySource instanceof
ResourcePropertySource ?
                                           ((ResourcePropertySource)
propertySource).withResourceName() : propertySource);
            //CompositePropertySource内部也有个proeprtySources列表
            //Composite PropertySource implementation that iterates over a set
of PropertySource instances.
            if (existing instanceof CompositePropertySource) {
                ((CompositePropertySource)
existing).addFirstPropertySource(newSource);
            }
            else {
                if (existing instanceof ResourcePropertySource) {
                    existing = ((ResourcePropertySource)
existing).withResourceName();
                }
                CompositePropertySource composite = new
CompositePropertySource(name);
                composite.addPropertySource(newSource);
                composite.addPropertySource(existing);
                propertySources.replace(name, composite);
            }
            return;
        }
```

```
    }
    if (this.propertySourceNames.isEmpty()) {//不存在任何PropertySource时，直接加入
envrionment中的propertySource
        propertySources.addLast(propertySource);
    }
    else {
        //末尾的名称是最后加入的
        String firstProcessed =
this.propertySourceNames.get(this.propertySourceNames.size() - 1);
        //每次新增都插入在最后的PropertySource之前，即后来的PropertySource会插入在
environment的PropertySources最前面
        propertySources.addBefore(firstProcessed, propertySource);
    }
    this.propertySourceNames.add(name);//List列表末尾加入name
}
```

componentScan

```
//Process any @ComponentScan annotations 处理组件扫描注解；先通过excludeFilter进行过
滤，再用includeFilter进行适配,再进行条件适配；
//******特别注意，在解析配置类的ComponentScan阶段，这些扫描出来的BeanDefintion就立马被注入
容器；并在是配置类候选时会递归解析配置类候选*******
Set<AnnotationAttributes> componentScans =
AnnotationConfigUtils.attributesForRepeatable(
    sourceClass.getMetadata(), ComponentScans.class, ComponentScan.class);
if (!componentScans.isEmpty() &&
    //在解析阶段此处只有标注@ComponentScans的这个类型是REGISTER_BEAN阶段，其后续
componentScanParser.parse扫描时，shouldSkip第二个参数为null即
    //依据扫描出来的正在处理的类的元数据信息自动推断递归调用shouldSkip第二个参数阶段的值,一般
是PARSE_CONFIGURATION，虽然也有可能为REGISTER_BEAN
    //但是目前调试未遇见过
    //if (phase == null) {
    //    if (metadata instanceof AnnotationMetadata &&
    //
ConfigurationClassUtils.isConfigurationCandidate((AnnotationMetadata) metadata))
{
    //        return shouldSkip(metadata,
ConfigurationPhase.PARSE_CONFIGURATION);
    //    }
    //    return shouldSkip(metadata, ConfigurationPhase.REGISTER_BEAN);
    //}
    !this.conditionEvaluator.shouldSkip(sourceClass.getMetadata(),
ConfigurationPhase.REGISTER_BEAN)) {
    for (AnnotationAttributes componentScan : componentScans) {
        // The config class is annotated with @ComponentScan -> perform the scan
immediately
        Set<BeanDefinitionHolder> scannedBeanDefinitions =
            this.componentScanParser.parse(componentScan,
sourceClass.getMetadata().getClassName());
        // Check the set of scanned definitions for any further config classes
and parse recursively if needed
        for (BeanDefinitionHolder holder : scannedBeanDefinitions) {
            BeanDefinition bdCand =
holder.getBeanDefinition().getOriginatingBeanDefinition();
            if (bdCand == null) {
                bdCand = holder.getBeanDefinition();
```

```
            }
            if (ConfigurationClassUtils.checkConfigurationClassCandidate(bdCand,
this.metadataReaderFactory)) {//检测是否配置类候选
                parse(bdCand.getBeanClassName(), holder.getBeanName());//这里也是
使用的原始的BeanName，递归解析
            }
        }
    }
}
//上文this.componentScanParser.parse(componentScan,
sourceClass.getMetadata().getClassName());的parse方法分析：
//一个ComponentScan执行一次
public Set<BeanDefinitionHolder> parse(AnnotationAttributes componentScan,
String declaringClass) {
        ClassPathBeanDefinitionScanner scanner = new
ClassPathBeanDefinitionScanner(this.registry,
                componentScan.getBoolean("useDefaultFilters"), this.environment,
this.resourceLoader);
    //大部分方法内容只是依据ComponentScan注解的属性进行一些设置
    Class<? extends BeanNameGenerator> generatorClass =
componentScan.getClass("nameGenerator");
    boolean useInheritedGenerator = (BeanNameGenerator.class == generatorClass);
    scanner.setBeanNameGenerator(useInheritedGenerator ? this.beanNameGenerator
:
                                 BeanUtils.instantiateClass(generatorClass));
    //除了ComponentScan有ScopedProxyMode属性，@Scoped也有这个相关的叫proxyMode属性
    ScopedProxyMode scopedProxyMode = componentScan.getEnum("scopedProxy");
    if (scopedProxyMode != ScopedProxyMode.DEFAULT) {
        scanner.setScopedProxyMode(scopedProxyMode);
    }
    else {
        Class<? extends ScopeMetadataResolver> resolverClass =
componentScan.getClass("scopeResolver");

 scanner.setScopeMetadataResolver(BeanUtils.instantiateClass(resolverClass));
    }
    //主类上的默认值为**/*.class
    scanner.setResourcePattern(componentScan.getString("resourcePattern"));
    //设置includeFilters
    for (AnnotationAttributes includeFilterAttributes :
componentScan.getAnnotationArray("includeFilters")) {
        List<TypeFilter> typeFilters =
TypeFilterUtils.createTypeFiltersFor(includeFilterAttributes,
this.environment,this.resourceLoader, this.registry);
        for (TypeFilter typeFilter : typeFilters) {
            scanner.addIncludeFilter(typeFilter);
        }
    }
    //设置excludeFilters
    for (AnnotationAttributes excludeFilterAttributes :
componentScan.getAnnotationArray("excludeFilters")) {
        List<TypeFilter> typeFilters =
TypeFilterUtils.createTypeFiltersFor(excludeFilterAttributes,
this.environment,this.resourceLoader, this.registry);
        for (TypeFilter typeFilter : typeFilters) {
            scanner.addExcludeFilter(typeFilter);
        }
    }
```

```
    //设置lazyInit
    boolean lazyInit = componentScan.getBoolean("lazyInit");
    if (lazyInit) {
        scanner.getBeanDefinitionDefaults().setLazyInit(true);
    }
    Set<String> basePackages = new LinkedHashSet<>();
    String[] basePackagesArray = componentScan.getStringArray("basePackages");
    for (String pkg : basePackagesArray) {
        //依据分隔符",; \t\n"解析得到路径，说明一个String也可以表示多个路径
        String[] tokenized =
StringUtils.tokenizeToStringArray(this.environment.resolvePlaceholders(pkg),
                    ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS);
        Collections.addAll(basePackages, tokenized);
    }
    for (Class<?> clazz : componentScan.getClassArray("basePackageClasses")) {
        basePackages.add(ClassUtils.getPackageName(clazz));
    }
    if (basePackages.isEmpty()) {//在未显示指定扫描路径时，以声明该注解类所在包路径为默认
路径
        basePackages.add(ClassUtils.getPackageName(declaringClass));
    }
    scanner.addExcludeFilter(new AbstractTypeHierarchyTraversingFilter(false,
false) {
        @Override
        protected boolean matchClassName(String className) {
            return declaringClass.equals(className);//这个就是在扫描时排除声明类
        }
    });
    return scanner.doScan(StringUtils.toStringArray(basePackages));//真正执行扫描
}

protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
    Assert.notEmpty(basePackages, "At least one base package must be
specified");
    Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
    for (String basePackage : basePackages) {
        //findCandidateComponents一般内部用scanCandidateComponents(basePackage)，后
文分析
        //利用ASM技术分析basePackage包下类的信息，生成BeanDefintion
        Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
        for (BeanDefinition candidate : candidates) {
            //处理Scope注解中的proxyMode
            ScopeMetadata scopeMetadata =
this.scopeMetadataResolver.resolveScopeMetadata(candidate);
            candidate.setScope(scopeMetadata.getScopeName());
            //此处beanNmaeGenerator是AnnotationBeanNameGenerator，If the
annotation's value doesn't indicate a bean name, an
            //appropriate name will be built based on the short name of the
class (with the first letter lower-cased).
            //For example:com.xyz.FooServiceImpl -> fooServiceImpl
            String beanName = this.beanNameGenerator.generateBeanName(candidate,
this.registry);
            if (candidate instanceof AbstractBeanDefinition) {
                //protected void postProcessBeanDefinition(AbstractBeanDefinition
beanDefinition, String beanName) {
                //
beanDefinition.applyDefaults(this.beanDefinitionDefaults);//应用默认的
BeanDefintion属性配置
```

```
                //    if (this.autowireCandidatePatterns != null) {//此处是设置
autowireCandidate的值相关
                //
beanDefinition.setAutowireCandidate(PatternMatchUtils.simpleMatch(this.autowireC
andidatePatterns,
                //
            beanName));
                //    }
                //}
                postProcessBeanDefinition((AbstractBeanDefinition) candidate,
beanName);
            }
            if (candidate instanceof AnnotatedBeanDefinition) {
                //处理Lazy Primary DependsOn Role Description 注解信息

 AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefiniti
on) candidate);
            }
            if (checkCandidate(beanName, candidate)) {//此判断分析，见后文
                //进入此处一般表示beanFacotry不存在beanNanme对应的BeanDefintion，准备将
candidate注入beanFactory
                BeanDefinitionHolder definitionHolder = new
BeanDefinitionHolder(candidate, beanName);
                definitionHolder =
                    AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata,
definitionHolder, this.registry);
                beanDefinitions.add(definitionHolder);
                //向beanFactory中注册candidate，内部最核心的方法调用是
                //registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());
                //此处特别注意，在解析配置类的ComponentScan阶段，这些扫描出来的
BeanDefintion就立马被注入容器
                registerBeanDefinition(definitionHolder, this.registry);
            }
        }
    }
    return beanDefinitions;
}

private Set<BeanDefinition> scanCandidateComponents(String basePackage) {
    Set<BeanDefinition> candidates = new LinkedHashSet<>();
    try {
        //调试的一个值：classpath*:org/example/**/*.class
        String packageSearchPath =
ResourcePatternResolver.CLASSPATH_ALL_URL_PREFIX +
            resolveBasePackage(basePackage) + '/' + this.resourcePattern;
        //按照包名获取其下面的所有类，每一个类都对应一个Resource
        //一个调试结果：表明内部类被视为一个单独的Resource
        //file [D:\IDEA
Workspace\logtest\target\classes\org\example\Applogtest.class]
        //file [D:\IDEA
Workspace\logtest\target\classes\org\example\Applogtest$NestedClass.class]
        //....
        Resource[] resources =
getResourcePatternResolver().getResources(packageSearchPath);
        boolean traceEnabled = logger.isTraceEnabled();
        boolean debugEnabled = logger.isDebugEnabled();
        for (Resource resource : resources) {
```

```
            if (traceEnabled) {
                logger.trace("Scanning " + resource);
            }
            if (resource.isReadable()) {
                try {
                    //实例为SimpleMetadataReader，底层使用ASM技术读取元数据
                    MetadataReader metadataReader =
getMetadataReaderFactory().getMetadataReader(resource);
                    //先通过excludeFilter进行过滤，再用includeFilter进行适配,再进行条件
适配
                    //protected boolean isCandidateComponent(MetadataReader
metadataReader) throws IOException {
                    //    for (TypeFilter tf : this.excludeFilters) {
                    //        if (tf.match(metadataReader,
getMetadataReaderFactory())) {
                    //            return false;
                    //        }
                    //    }
                    //    for (TypeFilter tf : this.includeFilters) {
                    //        if (tf.match(metadataReader,
getMetadataReaderFactory())) {
                    //            //这里面会触发条件条件判断比如
ConditionalOnClass/Bean
                    //            //ConditionalOnClass: 基本就
是!isPresent(className, classLoader)，而这个底层又是通过Class.forName
                    //            //进行判断，如果类存在isPresent就返回True,否则返回
false;
                    //
                    //            return isConditionMatch(metadataReader);
                    //        }
                    //    }
                    //    return false;
                    //}
                    if (isCandidateComponent(metadataReader)) {
                        //扫描到的组件类型都为ScannedGenericBeanDefinition
                        ScannedGenericBeanDefinition sbd = new
ScannedGenericBeanDefinition(metadataReader);
                        sbd.setSource(resource);
                        //The default implementation checks whether the class is
not an interface and not dependent
                        //on an enclosing class. isIndependent():Determine
whether the underlying class is independent, i.e.
                        //whether it is a top-level class or a nested class
(static inner class) that can be constructed
                        //independently from an enclosing class 不是特别理解，但影响
不大，这里一般是具体的实现类应该都没问题
                        //protected boolean
isCandidateComponent(AnnotatedBeanDefinition beanDefinition) {
                        //  AnnotationMetadata metadata =
beanDefinition.getMetadata();
                        //  return (metadata.isIndependent() &&
(metadata.isConcrete() ||
                        //          (metadata.isAbstract() &&
metadata.hasAnnotatedMethods(Lookup.class.getName()))));
                        //}
                        if (isCandidateComponent(sbd)) {
                            if (debugEnabled) {
```

```
                                logger.debug("Identified candidate component
class: " + resource);
                            }
                            candidates.add(sbd);//视为扫描到的BeanDefinition
                        }
                        else {
                            if (debugEnabled) {logger.debug("Ignored because not
a concrete top-level class: " + resource);}
                        }
                    }
                    else {
                        if (traceEnabled) {logger.trace("Ignored because not
matching any filter: " + resource); }
                    }
                }
                catch (Throwable ex) {
                    throw new BeanDefinitionStoreException("Failed to read
candidate component class: " + resource, ex);
                }
            }
            else { if (traceEnabled) {logger.trace("Ignored because not
readable: " + resource);}}
        }
    }
    catch (IOException ex) { throw new BeanDefinitionStoreException("I/O failure
during classpath scanning", ex); }
    return candidates;
}

protected boolean checkCandidate(String beanName, BeanDefinition beanDefinition)
throws IllegalStateException {
    if (!this.registry.containsBeanDefinition(beanName)) {//如果beanFactory不存在已
有BeanDefintion，返回true
        return true;
    }
    BeanDefinition existingDef = this.registry.getBeanDefinition(beanName);
    BeanDefinition originatingDef = existingDef.getOriginatingBeanDefinition();
    if (originatingDef != null) {
        existingDef = originatingDef;
    }
    //检测新计划注册的beanDeftintion和已有的BeanDefinton是否“兼容”，兼容则表示新的
BeanDefitnition不进行注册；否则抛出异常
    //protected boolean isCompatible(BeanDefinition newDefinition, BeanDefinition
existingDefinition) {
    //  return (!(existingDefinition instanceof ScannedGenericBeanDefinition) ||
// explicitly registered overriding bean
    //          (newDefinition.getSource() != null &&
newDefinition.getSource().equals(existingDefinition.getSource())) ||  //
    //scanned same file twice
    //          newDefinition.equals(existingDefinition));  // scanned
equivalent class twice
    //}
    //不是特别理解“兼容”的含义，但是问题不大
    if (isCompatible(beanDefinition, existingDef)) {
        return false;
    }
    throw new ConflictingBeanDefinitionException("Annotation-specified bean name
'" + beanName +
```

```
     "' for bean class [" + beanDefinition.getBeanClassName() + "] conflicts
with existing, " + "non-compatible bean definition of same name and class [" +
existingDef.getBeanClassName() + "]");
}
```

processImports

```
//Process any @Import annotations  @Configuration, ImportSelector,
ImportBeanDefinitionRegistrar, or regular component classes to import.
processImports(configClass, sourceClass, getImports(sourceClass), filter, true);

//Returns @Import class, considering all meta-annotations 获取sourceClass上标注的
Import数据（包括元注解上的import）
//会考虑所有注解元数据信息：比如@EnableAutoConfiguration注解在定义时的注解元数据就包含
@Import(AutoConfigurationImportSelector.class)
//SourceClass:Simple wrapper that allows annotated source classes to be dealt
with in a uniform manner, regardless of how they
//are loaded. 其中定义了private final Object source;  // 既可能是Class，也可能是
MetadataReader（这种形式通常意味着以ASM读取处理类）
private Set<SourceClass> getImports(SourceClass sourceClass) throws IOException
{
    Set<SourceClass> imports = new LinkedHashSet<>();
    Set<SourceClass> visited = new LinkedHashSet<>();
    collectImports(sourceClass, imports, visited);
    return imports;
}
private void collectImports(SourceClass sourceClass, Set<SourceClass> imports,
Set<SourceClass> visited)
            throws IOException {
    if (visited.add(sourceClass)) {
        for (SourceClass annotation : sourceClass.getAnnotations()) {
            String annName = annotation.getMetadata().getClassName();
            if (!annName.equals(Import.class.getName())) {
                collectImports(annotation, imports, visited);//递归处理非Import注
解；因为某些注解的元注解会进行Import
            }
        }
        //将当前处理的sourceClass上的所有Import的class加入到imports中

 imports.addAll(sourceClass.getAnnotationAttributes(Import.class.getName(),
"value"));
    }
}

private void processImports(ConfigurationClass configClass, SourceClass
currentSourceClass,
            Collection<SourceClass> importCandidates, Predicate<String>
exclusionFilter,
            boolean checkForCircularImports) {
    if (importCandidates.isEmpty()) {//如果没有需要Import的候选成员，则直接返回
        return;
    }
    //checkForCircularImports为true,isChainedImportOnStack(configClass)用于检测循环
导入
    //private boolean isChainedImportOnStack(ConfigurationClass configClass) {
    //  if (this.importStack.contains(configClass)) {
    //      String configClassName = configClass.getMetadata().getClassName();
```

```
    //      //此处getImportingClassFor相当于调用
CollectionUtils.lastElement(this.imports.get(importedClass))
    //      //可以结合后文this.importStack.registerImport来对比理解，这里维护了被导入类
与导入ing类的关系
    //      AnnotationMetadata importingClass =
this.importStack.getImportingClassFor(configClassName);
    //      while (importingClass != null) {
    //          if (configClassName.equals(importingClass.getClassName())) {
    //              return true;
    //          }
    //          importingClass =
this.importStack.getImportingClassFor(importingClass.getClassName());
    //      }
    //  }
    //  return false;
    //}
    if (checkForCircularImports && isChainedImportOnStack(configClass)) {
        this.problemReporter.error(new CircularImportProblem(configClass,
this.importStack));//错误report
    }
    else {
        this.importStack.push(configClass);
        try {
            for (SourceClass candidate : importCandidates) {
                if (candidate.isAssignable(ImportSelector.class)) {
                    // Candidate class is an ImportSelector -> delegate to it to
determine imports
                    Class<?> candidateClass = candidate.loadClass();
                    //实例化ImportSelector，这种类型的实现类只能通过Aware接口/以及有参构
造函数（仅唯一一个构造函数）注入一些特定数据
                    //包括BeanClassLoaderAware BeanFactoryAware EnvironmentAware
ResourceLoaderAware感知器（唯一构造函数只能接收感知器
                    //对应的类型实例，其它都不行）,实例化时要么是带有这些特定数据的唯一构
造，要么是无参构造
                    //ParserStrategyUtils: Common delegate code for the handling
of parser strategies, e.g. TypeFilter,
                    //ImportSelector, ImportBeanDefinitionRegistrar
                    ImportSelector selector =
ParserStrategyUtils.instantiateClass(candidateClass, ImportSelector.class,
                                                this.environment,
this.resourceLoader, this.registry);
                    Predicate<String> selectorFilter =
selector.getExclusionFilter();
                    if (selectorFilter != null) {
                        exclusionFilter = exclusionFilter.or(selectorFilter);//此
处产生了一个新的exclusionFilter
                    }
                    //是否是延迟导入DeferredImportSelector的实例
                    //延迟不会立马执行；非延迟则会递归立马执行
                    if (selector instanceof DeferredImportSelector) {
                        //将configClass和selector封装后放入
ConfigurationClassParser.deferredImportSelectorHandler
                        //在顶层parse方法即将执行结束前，会执行
this.deferredImportSelectorHandler.process();
                        //DeferredImportSelector:A variation of ImportSelector
that runs after all @Configuration beans have
                        //been processed.This type of selector can be
particularly useful when the selected imports are
```

```
                        //@Conditional.详细见后文分析
                        this.deferredImportSelectorHandler.handle(configClass,
(DeferredImportSelector) selector);
                    }
                    else {
                        String[] importClassNames =
selector.selectImports(currentSourceClass.getMetadata());
                        Collection<SourceClass> importSourceClasses =
asSourceClasses(importClassNames, exclusionFilter);
                        //这种方式不检测循环引入,且新的exclusionFilter只在这里的递归导入
中被传递使用
                        processImports(configClass, currentSourceClass,
importSourceClasses, exclusionFilter, false);
                    }
                }
                else if
(candidate.isAssignable(ImportBeanDefinitionRegistrar.class)) {
                    // Candidate class is an ImportBeanDefinitionRegistrar ->
                    // delegate to it to register additional bean definitions
                    Class<?> candidateClass = candidate.loadClass();
                    //实例化ImportBeanDefinitionRegistrar
                    ImportBeanDefinitionRegistrar registrar =
                        ParserStrategyUtils.instantiateClass(candidateClass,
ImportBeanDefinitionRegistrar.class,
                                                             this.environment,
this.resourceLoader, this.registry);
                    //此处只是放入配置类存储:
this.importBeanDefinitionRegistrars.put(registrar, importingClassMetadata);
                    //并未实际执行注册BeanDefintion,只能延迟执行
                    configClass.addImportBeanDefinitionRegistrar(registrar,
currentSourceClass.getMetadata());
                }
                else {
                    // Candidate class not an ImportSelector or
ImportBeanDefinitionRegistrar ->
                    // process it as an @Configuration class
                    //在ImportStack中有MultiValueMap<String, AnnotationMetadata>
imports
                    //此处相当于调用
ImportStack.imports.add(importedClass[candidate],
importingClass[currentSourceClass])
                    this.importStack.registerImport(
                        currentSourceClass.getMetadata(),
candidate.getMetadata().getClassName());
                    //类似一般的配置类候选一样进行递归处理，这里是面的条件评估时是
ConfigurationPhase.PARSE_CONFIGURATION

 processConfigurationClass(candidate.asConfigClass(configClass),
exclusionFilter);
                }
            }
        }
        catch (BeanDefinitionStoreException ex) {
            throw ex;
        }
        catch (Throwable ex) {
            throw new BeanDefinitionStoreException(
```

```
                "Failed to process import candidates for configuration class ["
+
                configClass.getMetadata().getClassName() + "]", ex);
        }
        finally {
            this.importStack.pop();
        }
    }
}

private class DeferredImportSelectorHandler {
    @Nullable
    private List<DeferredImportSelectorHolder> deferredImportSelectors = new
ArrayList<>();
    public void handle(ConfigurationClass configClass, DeferredImportSelector
importSelector) {
        DeferredImportSelectorHolder holder = new
DeferredImportSelectorHolder(configClass, importSelector);
        if (this.deferredImportSelectors == null) {
            DeferredImportSelectorGroupingHandler handler = new
DeferredImportSelectorGroupingHandler();
            handler.register(holder);
            handler.processGroupImports();
        }
        else {
            this.deferredImportSelectors.add(holder);//默认情况下应该是只执行这个添加
操作
        }
    }

    public void process() {
        List<DeferredImportSelectorHolder> deferredImports =
this.deferredImportSelectors;
        this.deferredImportSelectors = null;
        try {
            if (deferredImports != null) {
                //an import group which can provide additional sorting and
filtering logic across different selectors.
                //从spring5.0开始在DeferredImportSelector引入group概念，可以跟细分的排
序与处理导入过程
                DeferredImportSelectorGroupingHandler handler = new
DeferredImportSelectorGroupingHandler();
                deferredImports.sort(DEFERRED_IMPORT_COMPARATOR);//排序
deferredImports
                deferredImports.forEach(handler::register);//利用deferedImports以
及hanlder::register进行组相关的注册
                //依据之前的注册结果，按组处理导入；其会利用所有importSelector的
ExclusionFilter，之后调用processImports逻辑
                //
                handler.processGroupImports();
            }
        }
        finally {
            this.deferredImportSelectors = new ArrayList<>();
        }
    }
}
```

```
private class DeferredImportSelectorGroupingHandler {//此处配合
DeferredImportSelectorHandler.process()阅读，仅作为信息补充不进行详细分析
    private final Map<Object, DeferredImportSelectorGrouping> groupings = new
LinkedHashMap<>();
    private final Map<AnnotationMetadata, ConfigurationClass>
configurationClasses = new HashMap<>();
    public void register(DeferredImportSelectorHolder deferredImport) {
        Class<? extends Group> group =
deferredImport.getImportSelector().getImportGroup();
        DeferredImportSelectorGrouping grouping =
this.groupings.computeIfAbsent(
            (group != null ? group : deferredImport),
            key -> new DeferredImportSelectorGrouping(createGroup(group)));
        grouping.add(deferredImport);

 this.configurationClasses.put(deferredImport.getConfigurationClass().getMetadat
a(),
                                      deferredImport.getConfigurationClass());
    }
    public void processGroupImports() {
        for (DeferredImportSelectorGrouping grouping : this.groupings.values())
{
            Predicate<String> exclusionFilter = grouping.getCandidateFilter();
            grouping.getImports().forEach(entry -> {
                ConfigurationClass configurationClass =
this.configurationClasses.get(entry.getMetadata());
                try {
                    processImports(configurationClass,
asSourceClass(configurationClass, exclusionFilter),

Collections.singleton(asSourceClass(entry.getImportClassName(),
exclusionFilter)),
                                   exclusionFilter, false);
                }
                catch (BeanDefinitionStoreException ex) {
                    throw ex;
                }
                catch (Throwable ex) {
                    throw new BeanDefinitionStoreException(
                        "Failed to process import candidates for configuration
class [" +
                        configurationClass.getMetadata().getClassName() + "]",
ex);
                }
            });
        }
    }
    private Group createGroup(@Nullable Class<? extends Group> type) {
        Class<? extends Group> effectiveType = (type != null ? type :
DefaultDeferredImportSelectorGroup.class);
        return ParserStrategyUtils.instantiateClass(effectiveType, Group.class,

 ConfigurationClassParser.this.environment,

 ConfigurationClassParser.this.resourceLoader,

 ConfigurationClassParser.this.registry);
    }
```

```
}
```

importResource

```
// Process any @ImportResource annotations  该注解主要用于导入xml/groovy形式的spring配置文件
AnnotationAttributes importResource =
    AnnotationConfigUtils.attributesFor(sourceClass.getMetadata(),
ImportResource.class);
if (importResource != null) {
    String[] resources = importResource.getStringArray("locations");
    Class<? extends BeanDefinitionReader> readerClass =
importResource.getClass("reader");
    for (String resource : resources) {
        String resolvedResource =
this.environment.resolveRequiredPlaceholders(resource);
        //此处只是将要导入的资源相关信息存入configClass，并未真正执行ImportResource注册BeanDefinitnio
        //this.importedResources.put(importedResource, readerClass)
        configClass.addImportedResource(resolvedResource, readerClass);
    }
}
```

processBean

```
// Process individual @Bean methods
//这个过程中会Try reading the class file via ASM for deterministic declaration order，即尝试通过ASM获取确定性的声明顺序，没有递归行为
Set<MethodMetadata> beanMethods = retrieveBeanMethodMetadata(sourceClass);
for (MethodMetadata methodMetadata : beanMethods) {
    //此处仅仅是将相关元数据信息存入config,this.beanMethods.add(method); 并未真正注入BeanDefintion
    configClass.addBeanMethod(new BeanMethod(methodMetadata, configClass));
}
```

processInterfaces

```
// Process default methods on interfaces
processInterfaces(configClass, sourceClass);

private void processInterfaces(ConfigurationClass configClass, SourceClass
sourceClass) throws IOException {
    for (SourceClass ifc : sourceClass.getInterfaces()) {
        //获取标注了@Bean的方法
        Set<MethodMetadata> beanMethods = retrieveBeanMethodMetadata(ifc);
        for (MethodMetadata methodMetadata : beanMethods) {
            if (!methodMetadata.isAbstract()) {//只有有实体的方法才能进一步处理，只能是default方法
                // A default method or other concrete method on a Java 8+
interface...
                //只是将相关元数据信息存入config,this.beanMethods.add(method); 并未真正注入BeanDefintion
                configClass.addBeanMethod(new BeanMethod(methodMetadata,
configClass));
            }
        }
```

```
            processInterfaces(configClass, ifc);//递归处理接口
        }
    }
```

processSuperClass

```
// Process superclass, if any
//结合所在方法外部的do while循环，递归处理父类；说明先处理自身，再处理父类
if (sourceClass.getMetadata().hasSuperClass()) {
    String superclass = sourceClass.getMetadata().getSuperClassName();
    if (superclass != null && !superclass.startsWith("java") &&
        !this.knownSuperclasses.containsKey(superclass)) {
        this.knownSuperclasses.put(superclass, configClass);
        // Superclass found, return its annotation metadata and recurse
        return sourceClass.getSuperClass();
    }
}
// No superclass -> processing is complete
return null;
```

加载BeanDefintion

```
此处是依据配置类加载BeanDefinition的核心过程,对应前文中
this.reader.loadBeanDefinitions(configClasses)
public void loadBeanDefinitions(Set<ConfigurationClass> configurationModel) {
    TrackedConditionEvaluator trackedConditionEvaluator = new
TrackedConditionEvaluator();//见条件Evaluator分析
    for (ConfigurationClass configClass : configurationModel) {
        loadBeanDefinitionsForConfigurationClass(configClass,
trackedConditionEvaluator);
    }
}
private void loadBeanDefinitionsForConfigurationClass(
            ConfigurationClass configClass, TrackedConditionEvaluator
trackedConditionEvaluator) {
    if (trackedConditionEvaluator.shouldSkip(configClass)) {//评估是否跳过导入
configClass的BeanDefintion
        String beanName = configClass.getBeanName();
        if (StringUtils.hasLength(beanName) &&
this.registry.containsBeanDefinition(beanName)) {
            this.registry.removeBeanDefinition(beanName);//registry是beanFactory
        }

 this.importRegistry.removeImportingClass(configClass.getMetadata().getClassName
());
        return;
    }
    if (configClass.isImported()) {
        //Register the Configuration class itself as a bean definition.
        //走到这说明configClass过了trackedConditionEvaluator.shouldSkip，内部使用了
ConfigurationPhase.REGISTER_BEAN
        //表示当前类以及“导入当前类的类”都是使用ConfigurationPhase.REGISTER_BEAN，表示条
件评估会时@ConditionalOnBean有效
        //此处将配置类的BeanDefintion注入beanFactory,细节参考后面对该方法的详细分析
```

```
            registerBeanDefinitionForImportedConfigurationClass(configClass);
        }
        for (BeanMethod beanMethod : configClass.getBeanMethods()) {
            //Read the given BeanMethod, registering bean definitions with the
BeanDefinitionRegistry就是注册@Bean标识的BeanDefintion
            //内部使用了ConfigurationPhase.REGISTER_BEAN，表示条件评估会时
@ConditionalOnBean有效
            //细节参考后面对该方法的详细分析
            loadBeanDefinitionsForBeanMethod(beanMethod);
        }
        //尝试从xml/grovvy等配置文件中加载BeanDefinition, 未特别深入，细节参考后文分析

     loadBeanDefinitionsFromImportedResources(configClass.getImportedResources());
        //尝试从Registrars注册BeanDefinition，没有进行条件评估，细节参考后文分析

     loadBeanDefinitionsFromRegistrars(configClass.getImportBeanDefinitionRegistrars
());
    }
    //向BeanFactory注册confiClass对应的BeanDefintion
    private void
registerBeanDefinitionForImportedConfigurationClass(ConfigurationClass
configClass) {
        AnnotationMetadata metadata = configClass.getMetadata();
        //可以看到默认是AnnotatedGenericBeanDefinition类型的Bean的定义
        AnnotatedGenericBeanDefinition configBeanDef = new
AnnotatedGenericBeanDefinition(metadata);
        ScopeMetadata scopeMetadata =
scopeMetadataResolver.resolveScopeMetadata(configBeanDef);
        configBeanDef.setScope(scopeMetadata.getScopeName());
        //通过BeanNameGenerator生成configBeanName，通常是全类名小写
        String configBeanName =
this.importBeanNameGenerator.generateBeanName(configBeanDef, this.registry);
        //主要处理配置类上的Lazy Primary DependsOn Role Description的注解属性
        AnnotationConfigUtils.processCommonDefinitionAnnotations(configBeanDef,
metadata);
        BeanDefinitionHolder definitionHolder = new
BeanDefinitionHolder(configBeanDef, configBeanName);
        //ScopedProxyMode.NO就直接返回definitionHolder
        //否则调用ScopedProxyCreator.createScopedProxy(definition, registry,
proxyTargetClass) 核心
        //    targetDefinition = definition.getBeanDefinition();//原始的beanDefintion
        //    targetBeanName = getTargetBeanName(originalBeanName);//"scopedTarget." +
originalBeanName
        //    proxyDefinition = new RootBeanDefinition(ScopedProxyFactoryBean.class)
        //    proxyDefinition.setDecoratedDefinition(new
BeanDefinitionHolder(targetDefinition, targetBeanName))
        //    registry.registerBeanDefinition(targetBeanName, targetDefinition)//将加上
scopedTarget.的原始beanDefinition注册beanFactory
        //    new
BeanDefinitionHolder(proxyDefinition,originalBeanName,definition.getAliases())使
用proxyDefinition替代原beanDefinition
        //
        definitionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata,
definitionHolder, this.registry);
        //以原始configBeanName注册BeanDeftintion(可能是原始的，也可能是proxyDefinition)
        this.registry.registerBeanDefinition(definitionHolder.getBeanName(),
definitionHolder.getBeanDefinition());
        configClass.setBeanName(configBeanName);
```

```
    if (logger.isTraceEnabled()) {
        logger.trace("Registered bean definition for imported class '" +
configBeanName + "'");
    }
}

private void loadBeanDefinitionsForBeanMethod(BeanMethod beanMethod) {
    ConfigurationClass configClass = beanMethod.getConfigurationClass();
    MethodMetadata metadata = beanMethod.getMetadata();
    String methodName = metadata.getMethodName();
    // Do we need to mark the bean as skipped by its condition? 依据beanMethod上的
条件注解是否满足要求
    if (this.conditionEvaluator.shouldSkip(metadata,
ConfigurationPhase.REGISTER_BEAN)) {
        configClass.skippedBeanMethods.add(methodName);
        return;
    }
    if (configClass.skippedBeanMethods.contains(methodName)) {
        return;
    }
    //获取Bean注解的属性信息
    AnnotationAttributes bean = AnnotationConfigUtils.attributesFor(metadata,
Bean.class);
    Assert.state(bean != null, "No @Bean annotation attributes");
    // Consider name and any aliases
    List<String> names = new ArrayList<>
(Arrays.asList(bean.getStringArray("name")));
    String beanName = (!names.isEmpty() ? names.remove(0) : methodName);
    // Register aliases even when overridden
    for (String alias : names) {
        this.registry.registerAlias(beanName, alias);//向beanFactory注册别名
    }
    // Has this effectively been overridden before (e.g. via XML)?
    // 目前个人理解是如果是不同配置类产生的相同名称的BeanDefintion，那么当前的
BeanDefinition覆盖之前的BeanDefinition;如果是相同配置类通过重载
    // 产生的BeanDefintion,那么保留之前的BeanDefintion
    if (isOverriddenByExistingDefinition(beanMethod, beanName)) {
        if (beanName.equals(beanMethod.getConfigurationClass().getBeanName())) {
            throw new
BeanDefinitionStoreException(beanMethod.getConfigurationClass().getResource().ge
tDescription(),
                                                    beanName, "Bean name derived
from @Bean method '" + beanMethod.getMetadata().getMethodName() +
                                                    "' clashes with bean name for
containing configuration class; please make those names unique!");
        }
        return;
    }
    //ConfigurationClassBeanDefinition: RootBeanDefinition marker subclass used
to signify that a bean definition was created
    //from a configuration class as opposed to any other configuration source 这
种类型的BeanDefinition表示其信息产生于某个配置类
    ConfigurationClassBeanDefinition beanDef = new
ConfigurationClassBeanDefinition(configClass, metadata, beanName);
    beanDef.setSource(this.sourceExtractor.extractSource(metadata,
configClass.getResource()));
    //静态@Bean和非静态的处理
    if (metadata.isStatic()) {
```

```
        // static @Bean method
        if (configClass.getMetadata() instanceof StandardAnnotationMetadata) {
            beanDef.setBeanClass(((StandardAnnotationMetadata)
configClass.getMetadata()).getIntrospectedClass());
        }
        else {
            beanDef.setBeanClassName(configClass.getMetadata().getClassName());
        }
        beanDef.setUniqueFactoryMethodName(methodName);
    }
    else {
        // instance @Bean method
        beanDef.setFactoryBeanName(configClass.getBeanName());
        beanDef.setUniqueFactoryMethodName(methodName);
    }
    //下面基本都是依据注解等元数据信息设置BeanDefintion
    if (metadata instanceof StandardMethodMetadata) {
        beanDef.setResolvedFactoryMethod(((StandardMethodMetadata)
metadata).getIntrospectedMethod());
    }
    beanDef.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_CONSTRUCTOR);

 beanDef.setAttribute(org.springframework.beans.factory.annotation.RequiredAnnot
ationBeanPostProcessor.
                         SKIP_REQUIRED_CHECK_ATTRIBUTE, Boolean.TRUE);
    AnnotationConfigUtils.processCommonDefinitionAnnotations(beanDef, metadata);
    Autowire autowire = bean.getEnum("autowire");
    if (autowire.isAutowire()) {
        beanDef.setAutowireMode(autowire.value());
    }
    boolean autowireCandidate = bean.getBoolean("autowireCandidate");
    if (!autowireCandidate) {
        beanDef.setAutowireCandidate(false);
    }
    String initMethodName = bean.getString("initMethod");
    if (StringUtils.hasText(initMethodName)) {
        beanDef.setInitMethodName(initMethodName);
    }
    String destroyMethodName = bean.getString("destroyMethod");
    beanDef.setDestroyMethodName(destroyMethodName);

    // Consider scoping 此处先获取代理模式信息
    ScopedProxyMode proxyMode = ScopedProxyMode.NO;
    AnnotationAttributes attributes =
AnnotationConfigUtils.attributesFor(metadata, Scope.class);
    if (attributes != null) {
        beanDef.setScope(attributes.getString("value"));
        proxyMode = attributes.getEnum("proxyMode");
        if (proxyMode == ScopedProxyMode.DEFAULT) {
            proxyMode = ScopedProxyMode.NO;
        }
    }
    // Replace the original bean definition with the target one, if necessary
    BeanDefinition beanDefToRegister = beanDef;
    if (proxyMode != ScopedProxyMode.NO) {
        BeanDefinitionHolder proxyDef = ScopedProxyCreator.createScopedProxy(
            //此处代理模式处理过程和
registerBeanDefinitionForImportedConfigurationClass中的代理模式一样，就不再重复分析
```

```
            new BeanDefinitionHolder(beanDef, beanName), this.registry,
            proxyMode == ScopedProxyMode.TARGET_CLASS);
        beanDefToRegister = new ConfigurationClassBeanDefinition(
            (RootBeanDefinition) proxyDef.getBeanDefinition(), configClass,
metadata, beanName);
    }
    if (logger.isTraceEnabled()) {
        logger.trace(String.format("Registering bean definition for @Bean method
%s.%s()",
                                   configClass.getMetadata().getClassName(),
beanName));
    }
    this.registry.registerBeanDefinition(beanName, beanDefToRegister);//注册
BeanDefintion
}

private void loadBeanDefinitionsFromImportedResources(//从xml/groovy等文件中加载
BeanDefintion
            Map<String, Class<? extends BeanDefinitionReader>>
importedResources) {

    Map<Class<?>, BeanDefinitionReader> readerInstanceCache = new HashMap<>();

    importedResources.forEach((resource, readerClass) -> {
        // Default reader selection necessary?
        if (BeanDefinitionReader.class == readerClass) {//目前主要就是groovy和xml资
源加载reader
            if (StringUtils.endsWithIgnoreCase(resource, ".groovy")) {
                // When clearly asking for Groovy, that's what they'll get...
                readerClass = GroovyBeanDefinitionReader.class;
            }
            else if (shouldIgnoreXml) {
                throw new UnsupportedOperationException("XML support disabled");
            }
            else {
                // Primarily ".xml" files but for any other extension as well
                readerClass = XmlBeanDefinitionReader.class;
            }
        }
        BeanDefinitionReader reader = readerInstanceCache.get(readerClass);
        if (reader == null) {
            try {
                // Instantiate the specified BeanDefinitionReader 创建实例
                reader =
readerClass.getConstructor(BeanDefinitionRegistry.class).newInstance(this.regist
ry);
                // Delegate the current ResourceLoader to it if possible  处理属性
                if (reader instanceof AbstractBeanDefinitionReader) {
                    AbstractBeanDefinitionReader abdr =
((AbstractBeanDefinitionReader) reader);
                    abdr.setResourceLoader(this.resourceLoader);
                    abdr.setEnvironment(this.environment);
                }
                readerInstanceCache.put(readerClass, reader);
            }
            catch (Throwable ex) {
                throw new IllegalStateException(
```

```
                    "Could not instantiate BeanDefinitionReader class [" +
readerClass.getName() + "]");
            }
        }
        // TODO SPR-6310: qualify relative path locations as done in
AbstractContextLoader.modifyLocations
        reader.loadBeanDefinitions(resource);
    });
}
//遍历registrars和对应的Metadata，向registry(beanFactory)中注入BeanDefinition
private void
loadBeanDefinitionsFromRegistrars(Map<ImportBeanDefinitionRegistrar,
AnnotationMetadata> registrars) {
    registrars.forEach((registrar, metadata) ->
                       registrar.registerBeanDefinitions(   metadata,
this.registry, this.importBeanNameGenerator));
}
```

条件Evaluator

ConditionEvaluator

```
//ConditionEvaluator类中的核心处理方法
//Determine if an item should be skipped based on @Conditional annotations. The
ConfigurationCondition.ConfigurationPhase will be deduced from the type of item
(i.e. a @Configuration class will be
ConfigurationCondition.ConfigurationPhase.PARSE_CONFIGURATION)
//shouldSkip第二个参数为null时会根据metadata自动推测phase
public boolean shouldSkip(AnnotatedTypeMetadata metadata) {
    return shouldSkip(metadata, null);
}

public boolean shouldSkip(@Nullable AnnotatedTypeMetadata metadata, @Nullable
ConfigurationPhase phase) {
    //不存在注解或者注解不包含Conditional，那么返回false,表示跳过
    if (metadata == null || !metadata.isAnnotated(Conditional.class.getName()))
{
        return false;
    }
    //在解析阶段@ComponentScans进行componentScanParser.parse扫描时，shouldSkip第二个参
数为null即
    //依据扫描出来的正在处理的类的元数据信息自动推断递归调用shouldSkip第二个参数阶段的值,一般
是PARSE_CONFIGURATION，虽然也有可能为REGISTER_BEAN
    //但是目前调试未遇见过
    if (phase == null) {//自动推测phase
        //AnnotationMetadata extends ClassMetadata, AnnotatedTypeMetadata继承
ClassMetadata个人理解它必是类上信息
        //Interface that defines abstract access to the annotations of a
specific class,
        //in a form that does not require that class to be loaded yet.
        if (metadata instanceof AnnotationMetadata &&
            //public static boolean isConfigurationCandidate(AnnotationMetadata
metadata) {
                // Do not consider an interface or an annotation...
```

```
                //if (metadata.isInterface()) {//whether the underlying class
represents an interface.
                //    return false;
                //}
                // Any of the typical annotations found?
                // 判断是否标注了如下注解
Component/ComponentScan/Import/ImportResource，是就返回true
                //for (String indicator : candidateIndicators) {
                //    if (metadata.isAnnotated(indicator)) {
                //        return true;
                //    }
                //}
                // Finally, let's look for @Bean methods...
                // 此处相当于metadata.hasAnnotatedMethods(Bean.class.getName())，判
断是否有@Bean
                //return hasBeanMethods(metadata);
            //} 这里整个方法用于判断配置类

 ConfigurationClassUtils.isConfigurationCandidate((AnnotationMetadata)
metadata)) {
            return shouldSkip(metadata, ConfigurationPhase.PARSE_CONFIGURATION);
        }
        return shouldSkip(metadata, ConfigurationPhase.REGISTER_BEAN);//这个调试目
前还未遇见过
    }
    //依据元数据获取条件类，接着执行排序
    List<Condition> conditions = new ArrayList<>();
    //依据元数据信息获取类上Conditional注解中Class[] value属性，但是Class属性会转成
string[]；
    //e.g.:@Conditional(OnClassCondition.class)就是@ConditionalOnClass；
@Conditional(OnBeanCondition.class)就是@ConditionalOnBean
    for (String[] conditionClasses : getConditionClasses(metadata)) {
        for (String conditionClass : conditionClasses) {
            ////这里获取的就是OnClassCondition、OnBeanCondition等
            Condition condition = getCondition(conditionClass,
this.context.getClassLoader());
            conditions.add(condition);
        }
    }
    AnnotationAwareOrderComparator.sort(conditions);
    //执行条件匹配
    for (Condition condition : conditions) {
        ConfigurationPhase requiredPhase = null;
        // ConfigurationCondition extends Condition ,可以提供更细粒度的条件控制
        if (condition instanceof ConfigurationCondition) {
            requiredPhase = ((ConfigurationCondition)
condition).getConfigurationPhase();
        }
        // ConfigurationCondition类可以根据其使用时机返回不同的requiredPhase，只有满足时
机的条件类才会去尝试执行匹配算法
        // 只要有一个条件不满足，那么shouldSkip就会返回true
        // OnClassCondition不是ConfigurationCondition，它的requiredPhase永远为null即
总会进入condition.matches
        // OnBeanCondition是ConfigurationCondition，并且它的requiredPhase永远为
ConfigurationPhase.REGISTER_BEAN即在REGISTER_BEAN
        // 阶段才会进入condition.matches匹配，否则相当于@ConditionalOnBean（元注解为
@Conditional(OnBeanCondition.class)）注解不存在（实际
        // 上，条件对象还是会创建，只是不参与匹配）
```

```
        if ((requiredPhase == null || requiredPhase == phase) &&
!condition.matches(this.context, metadata)) {
            return true;
        }
    }
    //表示条件满足，shouldSkip表示false
    return false;
}
```

TrackedConditionEvaluator

```
//这个类在评估条件前，会先考虑导入当前类的类是否至少存在一个，如果存在，则判断当前类的评估条件；如果不存在，说明当前类不是被Imported by，那么跳过，主要是确保导入“被导入类”的类需要满足条件注解要求才行；利用了缓存加速评估
/**
* Evaluate {@code @Conditional} annotations, tracking results and taking into
* account 'imported by'.
*/
private class TrackedConditionEvaluator {
    private final Map<ConfigurationClass, Boolean> skipped = new HashMap<>();
    public boolean shouldSkip(ConfigurationClass configClass) {
        Boolean skip = this.skipped.get(configClass);
        if (skip == null) {
            if (configClass.isImported()) {
                boolean allSkipped = true;
                for (ConfigurationClass importedBy :
configClass.getImportedBy()) {
                    if (!shouldSkip(importedBy)) {//评估导入当前类的类的条件
                        allSkipped = false;
                        break;
                    }
                }
                if (allSkipped) {
                    // The config classes that imported this one were all
skipped, therefore we are skipped...
                    skip = true;
                }
            }
            if (skip == null) {//评估当前类的条件
                skip = conditionEvaluator.shouldSkip(configClass.getMetadata(),
ConfigurationPhase.REGISTER_BEAN);
            }
            this.skipped.put(configClass, skip);
        }
        return skip;
    }
}
```

自动配置

流程分析

```
invokeBeanFactoryPostProcessors
    PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors
        invokeBeanDefinitionRegistryPostProcessors

BeanDefinitionRegistryPostProcessor.postProcessBeanDefinitionRegistry
                processConfigBeanDefinitions
                    candidateNames = registry.getBeanDefinitionNames()//一开始只有
少量的已载入BeanDefinition
                    for (String beanName : candidateNames) {
                        ...
                        if
(ConfigurationClassUtils.checkConfigurationClassCandidate(beanDef,

 this.metadataReaderFactory)) {
                            //此处只有主类是候选配置类
                            configCandidates.add(new
BeanDefinitionHolder(beanDef, beanName));
                        }
                    }
                    configCandidates.sort
                    candidates = new LinkedHashSet<>(configCandidates)
                    parser.parse(candidates)//这里最开始，只有主类，整个处理过程都是从着
开始
```

- SpringBootApplication由SpringBootConfiguration/EnableAutoConfiguration/ComponentScan三个重要元注解组成
    - SpringBootConfiguration基本等同于@Configuration，这也是checkConfigurationClassCandidate能通过的原因
- 首先对主类进行parse
    - 这个阶段会先触发@ComponenetScan处理，将用户定义的组件扫描
        - 会经过excludeFilters/includeFilters过滤
        - 会经过isConditionMatch：conditionEvaluator.shouldSkip(metadataReader.getAnnotationMetadata()) 条件注解判断
        - 经过上面判断且符合条件的组件会将BeanDefintion注册到BeanFacotry（除了此阶段其它阶段都不会注册BeanDefinition）
- 之后触发@Import处理
    - @EnableAutoConfiguration注解的元注解引入的@Import(AutoConfigurationImportSelector.class)
        - 由于AutoConfigurationImportSelector是DeferredImportSelector，其会存放入deferredImportSelectorHandler，在主类解析完后，顶层parse方法最后一步执行；这是自动配置的的核心类；
        - EnableAutoConfiguration还有一个元注解@AutoConfigurationPackage
            - @AutoConfigurationPackage的元注解引入@Import(AutoConfigurationPackages.Registrar.class)
                - 这部分属于ImportBeanDefinitionRegistrar，仅仅在配置类中添加了Registrar对象，但未真正注册BeanDefintion
    - 其它阶段的解析对于主类来说无意义

- 所有执行过doProcessConfigurationClass(一个parse的子方法，基本等同于parse)方法的配置类都会存放入parser.configurationClasses一个MAP中，在后续loadBeanDefinitions使用
- parse结尾阶段：this.deferredImportSelectorHandler.process() 这个是自动配置的核心过程，此时主类以及Component扫描的组件类已注册BeanDefintion，但是其它形式比如Import的组件还没注册BeanDefintion，但是相关信息已记录在parser中，唯一例外的就是DeferredImportSelector，此时处理这个自动配置类的导入；再次强调一点，DeferredImportSelector只有一次执行机会，就是在parse结尾阶段，之后DeferredImportSelector即使再次引入了DeferredImportSelector,对应代码：this.deferredImportSelectorHandler.handle(configClass, (DeferredImportSelector) selector)，只会临时加入到内部变量中，但是这个变量最后会被直接清空，什么也不会执行;核心处理过程见后文分析
- 解析完配置类后，进行loadBeanDefintions；核心处理过程见前文分析
    - parse阶段得到的配置类自身会走条件Evaluator体系，评估失败直接返回，成功往下走
    - registerBeanDefinitionForImportedConfigurationClass注册配置类自己
    - loadBeanDefinitionsForBeanMethod 注册配置类中@Bean标注的方法并注册Bean,会走独立的conditionEvaluator.shouldSkip
    - loadBeanDefinitionsFromImportedResources 从xml/groovy等文件中加载，暂未进一步深入分析
    - loadBeanDefinitionsFromRegistrars 处理配置类导入的Registrars，不会走独立的conditionEvaluator.shouldSkip（但本身其实经历过导入该Registrars的配置类的条件评估）

  补充parse结尾阶段分析：this.deferredImportSelectorHandler.process()

```
public void process() {
    //在springboot默认配置下，此处只有一个类型为AutoConfigurationImportSelector的
DeferredImportSelector
    List<DeferredImportSelectorHolder> deferredImports =
this.deferredImportSelectors;
    //此处被设置为null；主要用于处理再次引入DeferredImportSelector，暂时来说只是让程
序不报错，但是再次引入的
    //DeferredImportSelector是不会执行的
    this.deferredImportSelectors = null;
    try {
        if (deferredImports != null) {
            DeferredImportSelectorGroupingHandler handler = new
                DeferredImportSelectorGroupingHandler();
            deferredImports.sort(DEFERRED_IMPORT_COMPARATOR);
            deferredImports.forEach(handler::register);
            //封装了DeferredImportSelector核心处理过程，实际只处理
AutoConfigurationImportSelector
            //与自动配置有关
            handler.processGroupImports();
        }
    }
    finally {
        //执行完延迟导入后被清空；此时里面可能有processGroupImports再次引入的
DeferredImportSelector
        //但是会立马清空，什么也不会执行
        this.deferredImportSelectors = new ArrayList<>();
    }
}

public void processGroupImports() {
    //此处SpringBoot默认只有一个封装了AutoConfigurationImportSelector的grouping
```

```
    for (DeferredImportSelectorGrouping grouping : this.groupings.values())
{
        //此处一是A:
        //  Predicate<String> DEFAULT_EXCLUSION_FILTER = className ->
        //  (className.startsWith("java.lang.annotation.") ||
        //     className.startsWith("org.springframework.stereotype."))
        //二是AutoConfigurationImportSelector中的shouldExclude方法B构建的
Predicate:
        //private boolean shouldExclude(String configurationClassName) {
        //    //此处后文详细分析，这里filter包括
OnClassCondition/OnWebApplicationCondition/OnBeanCondition
        //    return getConfigurationClassFilter().
        //
filter(Collections.singletonList(configurationClassName)).isEmpty();
        //}
        //得到test(t) -> A.test(t) || B.test(t) 作为exclusionFilter
        Predicate<String> exclusionFilter = grouping.getCandidateFilter();
        //grouping.getImports()见后文分析
        grouping.getImports().forEach(entry -> {
            ConfigurationClass configurationClass =
                this.configurationClasses.get(entry.getMetadata());
            try {
                //此处是延迟导入的核心逻辑（前文已分析），和普通Import导入基本没什么太
大区别
                processImports(configurationClass,
asSourceClass(configurationClass,exclusionFilter),

Collections.singleton(asSourceClass(entry.getImportClassName(),
exclusionFilter)),
                     exclusionFilter, false);
            }
            catch (BeanDefinitionStoreException ex) {
                throw ex;
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                    "Failed to process import candidates for configuration
class [" +
                    configurationClass.getMetadata().getClassName() + "]",
ex);
            }
        });
    }
}
```

分析前文：

```
getConfigurationClassFilter().
   filter(Collections.singletonList(configurationClassName)).isEmpty();
private ConfigurationClassFilter getConfigurationClassFilter() {
    if (this.configurationClassFilter == null) {
        //此处见下面进一步分析
        List<AutoConfigurationImportFilter> filters =
getAutoConfigurationImportFilters();
        for (AutoConfigurationImportFilter filter : filters) {//处理Aware感知
器接口
            invokeAwareMethods(filter);
        }
        //封装filters并创建configurationClassFilter
```

```
            this.configurationClassFilter = new
ConfigurationClassFilter(this.beanClassLoader, filters);
        }
        return this.configurationClassFilter;
    }
    protected List<AutoConfigurationImportFilter>
    getAutoConfigurationImportFilters() {
        //在META-INF/spring.factories搜索key为
        //org.springframework.boot.autoconfigure.AutoConfigurationImportFilter的
    value值，
        //并依据value值中的实现类实例化并返回，见下面进一步分析
        return
    SpringFactoriesLoader.loadFactories(AutoConfigurationImportFilter.class,
                    this.beanClassLoader);
    }
    //此处是SpringFactoriesLoader.loadFactories方法详细
    public static <T> List<T> loadFactories(Class<T> factoryType, @Nullable
    ClassLoader classLoader) {
        Assert.notNull(factoryType, "'factoryType' must not be null");
        ClassLoader classLoaderToUse = classLoader;
        if (classLoaderToUse == null) {
            classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();
        }
        //该方法从META-INF/spring.factories加载所有的key=value1,value2对，并解析成
    Map<String, List<String>>
        //只有第一次调用该方法需要解析，后续只需要从缓存取出Map即可，使用
    factoryType.getName()作为key进行查找
        //不同文件相同的key的value会进行合并到一个key对应的List<String>下面，List中一般为
    实现为key的实现类
        List<String> factoryImplementationNames = loadFactoryNames(factoryType,
    classLoaderToUse);
        if (logger.isTraceEnabled()) {
            logger.trace("Loaded [" + factoryType.getName() + "] names: " +
    factoryImplementationNames);
        }
        List<T> result = new ArrayList<>(factoryImplementationNames.size());
        for (String factoryImplementationName : factoryImplementationNames) {
            //只使用无参构造函数实例化对象并放入result，注意实例化过程会进行校验，要求value
    中的实现类必须
            //factoryType.isAssignableFrom(value中的实现类)
            result.add(instantiateFactory(factoryImplementationName,
    factoryType, classLoaderToUse));
        }
        AnnotationAwareOrderComparator.sort(result);//结果排序
        return result;
    }
    @FunctionalInterface
    public interface AutoConfigurationImportFilter {
        /**
         * Apply the filter to the given auto-configuration class candidates.
         * @param autoConfigurationClasses the auto-configuration classes being
    considered.
         * This array may contain {@code null} elements. Implementations should
    not change the
         * values in this array.
         * @param autoConfigurationMetadata access to the meta-data generated by
    the
         * auto-configure annotation processor
```

```
     * @return a boolean array indicating which of the auto-configuration
classes should
     * be imported. The returned array must be the same size as the incoming
     * {@code autoConfigurationClasses} parameter. Entries containing {@code
false} will
     * not be imported.
     */
    boolean[] match(String[] autoConfigurationClasses,
AutoConfigurationMetadata,
                    autoConfigurationMetadata);
}

分析前文：
grouping.getImports()
public Iterable<Group.Entry> getImports() {
    //此处只有封装了AutoConfigurationImportSelector的
DeferredImportSelectorHolder
    for (DeferredImportSelectorHolder deferredImport : this.deferredImports)
{
        //获取的Metadata是主类的Metadata,剩下一个参数是
AutoConfigurationImportSelector

 this.group.process(deferredImport.getConfigurationClass().getMetadata(),
                            deferredImport.getImportSelector());
    }
   //此处见后文分析，涉及到了排序
    return this.group.selectImports();
}
//this.group.process即为AutoConfigurationImportSelector.process，此处是详细过程
public void process(AnnotationMetadata annotationMetadata,
DeferredImportSelector deferredImportSelector) {
    Assert.state(deferredImportSelector instanceof
AutoConfigurationImportSelector,
                 () -> String.format("Only %s implementations are supported,
got %s",

AutoConfigurationImportSelector.class.getSimpleName(),

deferredImportSelector.getClass().getName()));
    AutoConfigurationEntry autoConfigurationEntry =
        ((AutoConfigurationImportSelector) deferredImportSelector)
        //见下面详细分析；此处的内部的过滤只通过META-INF/spring-autoconfigure-
metadata.properties里的元数据条件过滤
        .getAutoConfigurationEntry(annotationMetadata);
    this.autoConfigurationEntries.add(autoConfigurationEntry);
    for (String importClassName :
autoConfigurationEntry.getConfigurations()) {
        //存入entries，annotationMetadata总是主类元数据;importClassName是经过过滤
后将进行processImports的类
        this.entries.putIfAbsent(importClassName, annotationMetadata);
    }
}
//((AutoConfigurationImportSelector) deferredImportSelector)
//  .getAutoConfigurationEntry(annotationMetadata)分析：
protected AutoConfigurationEntry
getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
    //检测Environment中是否有key为spring.boot.enableautoconfiguration的属性；如果
没有，默认值为true
```

```
    //可以看出自动配置功能可以通过属性进行关闭
    if (!isEnabled(annotationMetadata)) {//这里的metadata是主类的元数据
        return EMPTY_ENTRY;
    }
    //获取的属性中只有exclude和excludeName，默认都为空字符串数组
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    //protected List<String> getCandidateConfigurations(AnnotationMetadata
metadata,
    //                                   AnnotationAttributes attributes) {
    //  List<String> configurations =
    //        //这里第一个参数为EnableAutoConfiguration.class
    //        //意味着在spring.factories中寻找key为
    //
org.springframework.boot.autoconfigure.EnableAutoConfiguration的属性对并将解析
值返回
    //
SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(
),
    //          getBeanClassLoader());
    //  Assert.notEmpty(configurations, "...");
    //  return configurations;//因此就是查找出来的未经过滤的所有预定义的自动配置类名
称列表，webstarter下有133
    //}
    List<String> configurations =
getCandidateConfigurations(annotationMetadata, attributes);
    //内部先转换成Set，再转换成List实现去重
    configurations = removeDuplicates(configurations);
    //protected Set<String> getExclusions(AnnotationMetadata metadata,
    //                                    AnnotationAttributes attributes) {
    //    Set<String> excluded = new LinkedHashSet<>();//集合，用于去重复
    //    excluded.addAll(asList(attributes, "exclude"));//将exclude内容加入集
合
    //    //将excludeName内容加入集合
    //
excluded.addAll(Arrays.asList(attributes.getStringArray("excludeName")));
    //    //getExcludeAutoConfigurationsProperty基本就是从Environment中获取key
为
    //    //spring.autoconfigure.exclude的数据（会视为需要排除的自动装配类）；然后返
回结果放入excluded
    //    excluded.addAll(getExcludeAutoConfigurationsProperty());
    //    return excluded;//默认springboot返回结果为空，即不存在排除类
    //}
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    //private void checkExcludedClasses(List<String> configurations,
Set<String> exclusions) {
    //  List<String> invalidExcludes = new ArrayList<>(exclusions.size());
    //  for (String exclusion : exclusions) {
    //      if (ClassUtils.isPresent(exclusion, getClass().getClassLoader())
&&//检测类是否可加载
    //              !configurations.contains(exclusion)) {//进一步判断排除类是否属
于自动装配类
    //          invalidExcludes.add(exclusion);
    //      }
    //  }
    //  if (!invalidExcludes.isEmpty()) {//如果排除了非自动装配的类，那么就会抛出异
常
    //      handleInvalidExcludes(invalidExcludes);//提示并抛出异常
    //  }
```

```
    //}
    checkExcludedClasses(configurations, exclusions);
    //从未经过滤的所有预定义的自动配置类名称列表排除exclusions（默认无排除类）
    configurations.removeAll(exclusions);
    //private ConfigurationClassFilter getConfigurationClassFilter() {
    //  if (this.configurationClassFilter == null) {
    //      //此处是从spring.factories中获取key为
    //
//org.springframework.boot.autoconfigure.AutoConfigurationImportFilter的数据，
    //      //解析并调用无参构造实例化过滤器，返回为为filters；调试时可以看到在
webstarter下有三个默认过滤器
    //      //按顺序分别为
OnClassCondition/OnWebApplicationCondition/OnBeanCondition
    //      List<AutoConfigurationImportFilter> filters =
getAutoConfigurationImportFilters();
    //      for (AutoConfigurationImportFilter filter : filters) {
    //          invokeAwareMethods(filter);
    //      }
    //      //ConfigurationClassFilter(ClassLoader classLoader,
    //      //                         List<AutoConfigurationImportFilter>
filters) {
    //      //      this.autoConfigurationMetadata =
    //      //
AutoConfigurationMetadataLoader.loadMetadata(classLoader);
    //      //      this.filters = filters;
    //      //}这个构造方法还需要特别注意autoConfigurationMetadata，会从
    //      //META-INF/spring-autoconfigure-metadata.properties中加载属性并放入
autoConfigurationMetadata
    //      //在webStarter下调试观测到700多的数据，注意多文件时相同key的话后面的值会覆
盖前面的值
    //      this.configurationClassFilter = new
ConfigurationClassFilter(this.beanClassLoader,
    //                                               filters);//此处将filters封装
一下
    //  }
    //  return this.configurationClassFilter;
    //} filter方法见下面分析
    configurations = getConfigurationClassFilter().filter(configurations);//
调试时此处过滤完后从133个变24个
    //从spring.factories中获取key为
org.springframework.boot.autoconfigure.AutoConfigurationImportListener的
Listener
    //用无参构造实例化，之后将当前环境封装成一个Event:event = new
AutoConfigurationImportEvent(this, configurations, exclusions)
    //然后循环遍历所有Listener，调用
listener.onAutoConfigurationImportEvent(event)
    //调试时就1个（写笔记的时候由于外部环境原因两个不同版本来回切换看，但版本差别不是特别
大），是
    //ConditionEvaluationReportAutoConfigurationImportListener 主要是record作
用
    fireAutoConfigurationImportEvents(configurations, exclusions);
    //将过滤后且经过Listener处理后的configurations和exclusions封装成
AutoConfigurationEntry并返回
    return new AutoConfigurationEntry(configurations, exclusions);
}
//对应前文getConfigurationClassFilter().filter(configurations)
List<String> filter(List<String> configurations) {//configurations是未经过滤的
所有预定义的自动配置类名称列表，webstarter133个
```

```
    long startTime = System.nanoTime();
    String[] candidates = StringUtils.toStringArray(configurations);
    boolean skipped = false;
    //此处是OnClassCondition/OnWebApplicationCondition/OnBeanCondition，注意调
用的是AutoConfigurationImportFilter的match方法
    for (AutoConfigurationImportFilter filter : this.filters) {
        //Entries containing false will not be imported,说明为false的项不考虑导
入，匹配方法下文分析
        boolean[] match = filter.match(candidates,
this.autoConfigurationMetadata);
        for (int i = 0; i < match.length; i++) {
            if (!match[i]) {//match[i] false表示不导入
                candidates[i] = null;//设置candidates[i]为null表示不导入
                skipped = true;
            }
        }
    }
    if (!skipped) {
        return configurations;
    }
    List<String> result = new ArrayList<>(candidates.length);
    for (String candidate : candidates) {//过滤不导入项，将可导入项放入result，随
后返回
        if (candidate != null) {
            result.add(candidate);
        }
    }
    if (logger.isTraceEnabled()) {
        int numberFiltered = configurations.size() - result.size();
        logger.trace("Filtered " + numberFiltered + " auto configuration
class in "
                     + TimeUnit.NANOSECONDS.toMillis(System.nanoTime() -
startTime) + " ms");
    }
    return result;
}
```

OnClassConditoin:

（另外两个条件OnWebApplicationCondition/OnBeanCondition和这类似，都是通过SpringFactoriesLoader.loadFactories
(AutoConfigurationImportFilter.class, this.beanClassLoader)一起实例化且只在此处使用（不像一般条件处理过程，依据处理对象的元数据来决定实例化条件对象），都是使用autoConfigurationMetadata进行过滤

OnWebApplicationCondition（关键代码）：

```
type=autoConfigurationMetadata.get(autoConfigurationClass,
"ConditionalOnWebApplication")
                                if (type == null) {return null;} 表示无
ConditionalOnWebApplication元数据信息
                                !isPresent(SERVLET_WEB_APPLICATION_CLASS,
getBeanClassLoader())
                                !isPresent(REACTIVE_WEB_APPLICATION_CLASS,
getBeanClassLoader())
```

OnBeanCondition(关键代码):

```
autoConfigurationMetadata.getSet(autoConfigurationClass,
"ConditionalOnBean")
                                !isPresent(className, classLoader)
```

```
    都是通过!isPresent(className, classLoader)进行判断，而其底层则是通过
Class.forName判断，及都是通过相关类是否存在进行判断，但要注意
    调用的都是AutoConfigurationImportFilter的match方法）
class OnClassCondition extends FilteringSpringBootCondition;
FilteringSpringBootCondition extends SpringBootCondition
    implements AutoConfigurationImportFilter, BeanFactoryAware,
BeanClassLoaderAware;
public boolean[] match(String[] autoConfigurationClasses,
AutoConfigurationMetadata autoConfigurationMetadata) {
    ConditionEvaluationReport report =
ConditionEvaluationReport.find(this.beanFactory);
    //OnClassCondition.getOutcomes->resolveOutcomesThreaded-
>resolveOutcomes->StandardOutcomesResolver.getOutcomes
    //private ConditionOutcome[] getOutcomes(String[]
autoConfigurationClasses, int start, int end,
    //          AutoConfigurationMetadata autoConfigurationMetadata)
{//StandardOutcomesResolver.getOutcomes
    //    ConditionOutcome[] outcomes = new ConditionOutcome[end - start];//
默认全为null
    //    for (int i = start; i < end; i++) {
    //        String autoConfigurationClass =
autoConfigurationClasses[i];//autoConfigurationClasses就是那133个自动配置类
    //        //做null判断的原因是之前的过滤类已经设置null表示不导入，多个过滤类共用一
个String[],因此为null直接跳过
    //        //由于默认值为null,所以outcomes[i]为null，在外层中match[i]视为true，
表示匹配，而再外层在match[i]匹配条件下什么也没做
    //        //也就是不会影响autoConfigurationClasses[i]的值，其为null（表示不考
虑导入）后，后续的过滤都保持null
    //        if (autoConfigurationClass != null) {
    //            //autoConfigurationMetadata是META-INF/spring-autoconfigure-
metadata.properties中那700多的数据，用于
    //            //快速过滤自动配置类，这里面存放的基本是载入条件：类名+"."+key(比如
ConditionalOnClass)，这些条件与类上条件注解保持一致
    //            //猜测该文件可能是自动生成的（人工维护可能也有但是springboot维护人员
应该不傻），用于快速过滤而不用一个一个解析类文件
    //            //get在内部实际调用为：properties.getProperty(className + "."
+ key);
    //            String candidates =
autoConfigurationMetadata.get(autoConfigurationClass, "ConditionalOnClass");
    //            if (candidates != null) {
    //              //private ConditionOutcome getOutcome(String candidates)
{
    //              //      try {
    //              //          if (!candidates.contains(",")) {//单值无，分隔
    //              //              return getOutcome(candidates,
this.beanClassLoader);
    //              //          }
    //              //          for (String candidate :
StringUtils.commaDelimitedListToStringArray(candidates)) {
    //              //              //单值但，分隔，需要拆分后再处理
    //              //              //private ConditionOutcome
getOutcome(String className, ClassLoader classLoader) {
    //              //              //  //matches相当
于//!isPresent(className, classLoader);底层通过Class.forName判断是否存在
    //              //              //        if
(ClassNameFilter.MISSING.matches(className, classLoader)) {
    //              //              //            return
```

```
    //              //              //
ConditionOutcome.noMatch(ConditionMessage.forCondition(ConditionalOnClass.cl
ass)
    //              //              //            .didNotFind("required
class").items(Style.QUOTE, className));
    //              //              //        }//noMatch:返回
ConditionOutcome(false, message)，isMatch必为false
    //              //              //        return null;//返回null说明存在
    //              //              //    }
    //              //              ConditionOutcome outcome =
getOutcome(candidate, this.beanClassLoader);
    //              //              if (outcome != null) {
    //              //                  return outcome;//不为Null说明一个条件不
匹配了
    //              //              }
    //              //          }
    //              //      }
    //              //      catch (Exception ex) {
    //              //          // We'll get another chance later
    //              //      }
    //              //      return null;//null表示条件匹配
    //              //}
    //                outcomes[i - start] = getOutcome(candidates);
    //            }
    //        }
    //    }
    //    return outcomes;//null表示匹配，ConditionOutcome.isMatch()必然为false
    //}
    ConditionOutcome[] outcomes = getOutcomes(autoConfigurationClasses,
autoConfigurationMetadata);
    boolean[] match = new boolean[outcomes.length];
    for (int i = 0; i < outcomes.length; i++) {
    //null表示匹配，match[i]为true;如果调用了outcomes[i].isMatch()，其结果必然为
false;表示不匹配
        match[i] = (outcomes[i] == null || outcomes[i].isMatch());
        if (!match[i] && outcomes[i] != null) {
            logOutcome(autoConfigurationClasses[i], outcomes[i]);
            if (report != null) {

 report.recordConditionEvaluation(autoConfigurationClasses[i], this,
outcomes[i]);
            }
        }
    }
    return match;//返回结果中true表示匹配，false表示不匹配
}
```

分析前文：

```
this.group.selectImports();
public Iterable<Entry> selectImports() {
    if (this.autoConfigurationEntries.isEmpty()) {
        return Collections.emptyList();
    }
    //获取所有exclusions排除类；因为每一个autoConfigurationEntry可能不一样，所以这里
是获取总的合并后的排除类
    Set<String> allExclusions = this.autoConfigurationEntries.stream()
```

```
.map(AutoConfigurationEntry::getExclusions).flatMap(Collection::stream).collect(Collectors.toSet());
    //获取所有autoConfigurationEntry中的configurations进行合并；每一个autoConfigurationEntry可能不一样
    Set<String> processedConfigurations = this.autoConfigurationEntries.stream()

.map(AutoConfigurationEntry::getConfigurations).flatMap(Collection::stream)
        .collect(Collectors.toCollection(LinkedHashSet::new));
    //事实上由于只有一个autoConfigurationEntry，实际这里相当于未排除（之前已经排除过了）；
    //只有真正出现多个autoConfiguraitonEntry时，才有意义
    processedConfigurations.removeAll(allExclusions);
    //getAutoConfigurationMetadata()是那700多条元数据
    //此处将排序后的结果收集成list<Entry>返回,排序见下面分析
    return sortAutoConfigurations(processedConfigurations, getAutoConfigurationMetadata()).stream()
        .map((importClassName) -> new Entry(this.entries.get(importClassName), importClassName))
        .collect(Collectors.toList());
}
private List<String> sortAutoConfigurations(Set<String> configurations,
                AutoConfigurationMetadata autoConfigurationMetadata) {
    return new AutoConfigurationSorter(getMetadataReaderFactory(), autoConfigurationMetadata)
        .getInPriorityOrder(configurations);
}
List<String> getInPriorityOrder(Collection<String> classNames) {
    //构造方法内部直接调用addToClasses，这个方法除了将自动配置类加到了this.classes中，也将自动配置类上的AutoConfigureBefore/After也放入
    //this.classes中，this为AutoConfigurationClasses
    //private void addToClasses(MetadataReaderFactory metadataReaderFactory,
    //          AutoConfigurationMetadata autoConfigurationMetadata, Collection<String> classNames, boolean required) {
    //    for (String className : classNames) {
    //        if (!this.classes.containsKey(className)) {//classes默认为空HashMap<String, AutoConfigurationClass>
    //            AutoConfigurationClass autoConfigurationClass = new AutoConfigurationClass(className,
    //                    metadataReaderFactory, autoConfigurationMetadata);
    //            boolean isAvailable() {
    //                try {//wasProcessed核心处理会判断：properties.containsKey(className)，properties是那700多条数据
    //                    if (!wasProcessed()) {//autoConfigurationMetadata700多数据中是否包含key=className是返回true否则false
    //                        //700多数据中不包含时通过ASM获取一下元数据；异常的话就返回false
    //                        //说明元数据信息既可以从元数据文件获取也可以通过ASM解析类文件获取并存入autoConfigurationClass内部数据
    //                        getAnnotationMetadata();
    //                    }
    //                    return true;
    //                }
    //                catch (Exception ex) {
    //                    return false;
    //                }
```

```
    //             }
    //           boolean available =
autoConfigurationClass.isAvailable();//available表示是否可以读取到元数据信息
    //            if (required || available) {//构造函数调用addToClasses
时,required为true；递归时，其为false;
    //                this.classes.put(className, autoConfigurationClass);//
缓存中放入此类
    //            }
    //            if (available) {//true表示有元数据信息（即可来自元数据文件，也可以
来自ASM解析）
    //                //autoConfigurationClass.getBefore()：内部判断元数据文件是
否有当前key为className对应的AutoConfigureBefore
    //                //条件，如果有就从元数据文件中获取；如果没有就通过ASM解析的元数
据获取;上述获取的数据可能为空集合；
    //                //注意递归处理时required变成了false
    //                addToClasses(metadataReaderFactory,
autoConfigurationMetadata,
    //                             autoConfigurationClass.getBefore(),
false);
    //                //此处和上面的getBefore类似，只是变成了AutoConfigureAfter而
已
    //                addToClasses(metadataReaderFactory,
autoConfigurationMetadata,
    //                             autoConfigurationClass.getAfter(),
false);
    //            }
    //        }
    //    }
    //}
    AutoConfigurationClasses classes = new
AutoConfigurationClasses(this.metadataReaderFactory,

 this.autoConfigurationMetadata, classNames);
    List<String> orderedClassNames = new ArrayList<>(classNames);//此处是过滤
后的自动配置类
    // Initially sort alphabetically 即字符串比较，按照字母顺序排序自动配置类
    Collections.sort(orderedClassNames);
    // Then sort by order  按照AutoConfigureOrder进行排序
    orderedClassNames.sort((o1, o2) -> {
        //private int getOrder() {
        //wasProcessed()：autoConfigurationMetadata700多数据中是否包含
key=this.className是返回true否则false；
        //其中this为o1/o2 className对应的AutoConfigurationClass,而
this.className实际就是o1/o2字符串类名
        //  if (wasProcessed()) {
        //      //从元数据文件中获取this.className对应的AutoConfigureOrder信息；如
果存在则返回元数据文件中的值，不存在则返回
        //      //默认值AutoConfigureOrder.DEFAULT_ORDER(值为0)
        //      return
this.autoConfigurationMetadata.getInteger(this.className,
"AutoConfigureOrder",
        //              AutoConfigureOrder.DEFAULT_ORDER);
        //  }
        //  Map<String, Object> attributes = getAnnotationMetadata()//底层通过
ASM技术解析出元数据，得到SimpleAnnotationMetadata
        //
.getAnnotationAttributes(AutoConfigureOrder.class.getName());//获取
AutoConfigureOrder注解属性
```

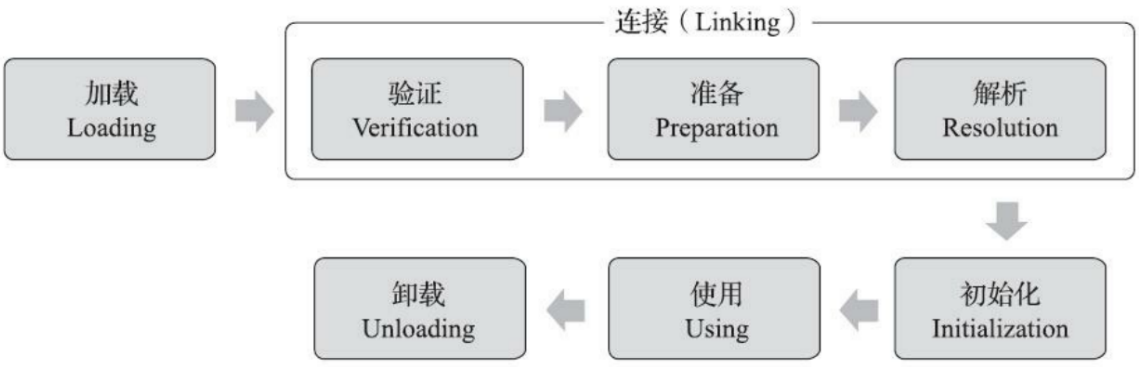
```
        //  //如果注解存在就返回注解信息，如果注解不存在，就使用
AutoConfigureOrder.DEFAULT_ORDER（值为0）
        //  return (attributes != null) ? (Integer) attributes.get("value")
: AutoConfigureOrder.DEFAULT_ORDER;
        //}
        int i1 = classes.get(o1).getOrder();
        int i2 = classes.get(o2).getOrder();
        return Integer.compare(i1, i2);
    });
    // Then respect @AutoConfigureBefore @AutoConfigureAfter
    //private List<String> sortByAnnotation(AutoConfigurationClasses
classes, List<String> classNames) {
    //  List<String> toSort = new ArrayList<>(classNames);
    //  toSort.addAll(classes.getAllNames());
    //  Set<String> sorted = new LinkedHashSet<>();//记录了插入顺序的Set(重复插
入不会影响顺序已存在的顺序)
    //  Set<String> processing = new LinkedHashSet<>();
    //  while (!toSort.isEmpty()) {//下面进一步分析
    //      doSortByAfterAnnotation(classes, toSort, sorted, processing,
null);
    //  }
    //  sorted.retainAll(classNames);
    //  return new ArrayList<>(sorted);
    //}
    //private void doSortByAfterAnnotation(AutoConfigurationClasses classes,
List<String> toSort, Set<String> sorted,
    //      Set<String> processing, String current) {
    //  if (current == null) {
    //      current = toSort.remove(0);//外层while循环每次进入时都会移除首元素
    //  }
    //  processing.add(current);//服务于循环检测
    //  //autoconfigure classes should have been applied即在current之前必须
applied的自动配置;同时处理了AutoConfigureBefore/After
    //  for (String after : classes.getClassesRequestedAfter(current)) {
    //      checkForCycles(processing, current, after);//processing不能包含
after，否则表示出现循环，抛出异常;
    //      //sorted不包含after（包含意味着偏序关系已满足且已处理）；只有toSort包含的
数据才可能放入sorted
    //      if (!sorted.contains(after) && toSort.contains(after)) {
    //          doSortByAfterAnnotation(classes, toSort, sorted, processing,
after);//递归处理偏序关系
    //      }
    //  }
    //  processing.remove(current);
    //  sorted.add(current);
    //}
    //这里本质上是依据AutoConfigureBefore/After构建偏序关系排序；如果不存在偏序关系，
那么上一步的排序保持不变
    orderedClassNames = sortByAnnotation(classes, orderedClassNames);
    return orderedClassNames;
}
```

> 总是根据处理对象的元数据信息来创建条件对象，存在什么条件判断就创建什么条件对象(e.g 标注ConditionalOnBean则会创建OnBeanCondition对象)

> 自己项目的包（一般指主类同级别的包）下的文件在 doScan时就会创建条件对象，扫描到的组件有什么元数据创建创建什么对象，走的是条件Evaluator体系(shouldSkip相关)
>
> 自动配置过滤类时会创建条件对象，包含三个OnClassCondition/OnWebApplicationCondition/OnBeanCondition条件对象，但走的是独立的AutoConfigurationImportFilter.match方法，都是通过isPresent（底层是Class.forName）进行判断的，和BeanFacotory没有关系，底层利用了元数据文件加速分析条件
>
> 自动配置类的对象会在processImports处理，待导入对象的元数据有什么条件创建什么对象，走的是条件Evaluator体系(shouldSkip相关)
>
> 用户可以自定义条件类
>
> OnClassCondition不是ConfigurationCondition，它的requiredPhase永远为null，在Evaluator体系评估时即总会进入condition.matches
> OnBeanCondition是ConfigurationCondition，并且它的requiredPhase永远为ConfigurationPhase.REGISTER_BEAN即在Evaluator体系评估时只有REGISTER_BEAN阶段才会进入condition.matches匹配，否则相当于@ConditionalOnBean（元注解为@Conditional(OnBeanCondition.class)）注解不存在（实际 上，条件对象还是会创建，只是不参与匹配）；走条件Evaluator体系评估时,OnClassCondition总是依据Class.forName类似的原理检测相关类是否存在，而OnBeanCondition是通过BeanFactory的接口方法检测在工厂的Bean（相应的类型的Bean是否存在,是否有标注了指定注解的Bean,是否有指定名称的Bean,是否搜索父Factory,是否存在包含指定泛型参数的指定类型的类..）
>
> doScan扫描组件时，只有@ComponentScan标注的组件自身是REGISTER阶段，其扫描出来的类一般是PARSE_CONFIGURATION阶段，虽然也有可能在自动推测的时候让扫描出来的类为REGISTER，但调试时暂未遇到过；loadBeanDefinitions时目前调试来看都属于REGISTER阶段
>
> 基于源码分析，注册Bean可以按顺序分为几个时期
>
> 1、解析阶段时@ComponentScan阶段的类在条件许可时就立马创建BeanDefintition并注入BeanFacotry，解析阶段的其它解析过程并未注册；在解析阶段@ConditionalOnBean不会生效；解析后的配置类放入configurationClasses；（在后续loadBeanDefinitions时会使用trackedConditionEvaluator. shouldSkip(configClass) 进一步判断配置类，这时底层shouldSkip是REGISTER_BEAN阶段，此时@ConditionalOnBean就又会生效产生效果）
>
> 2、在解析阶段的结尾，会进行自动配置类的解析，和Import导入处理类似，解析后的数据存储在parser中，还未真正loadBeanDefintions
>
> 3、在loadBeanDefintions阶段进行载入BeanDefintions

@ConditionalOnClass

- 类加载



一个类型从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期将会经历加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）七个阶段，其中验证、准备、解析三个部分统称 为连接（Linking）

关于在什么情况下需要开始类加载过程的第一个阶段"加载"，《Java虚拟机规范》中并没有进行 强制约束，这点可以交给虚拟机的具体实现来自由把握。但是对于初始化阶段，《Java虚拟机规范》 则是严格规定了有且只有六种情况必须立即对类进行"初始化"（而加载、验证、准备自然需要在此之 前开始）：（**即初始化时必然已经进行加载**，一般而言只有主动引用时才尝试加载）

```
1）遇到new、getstatic、putstatic或invokestatic这四条字节码指令时，如果类型没有进行过初始化，则需要先触发其初始化阶段。能够生成这四条指令的典型Java代码场景有：
·使用new关键字实例化对象的时候。
·读取或设置一个类型的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）的时候。
·调用一个类型的静态方法的时候。
2）使用java.lang.reflect包的方法对类型进行反射调用的时候，如果类型没有进行过初始化，则需要先触发其初始化。
3）当初始化类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
4）当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。
5）当使用JDK 7新加入的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果为REF_getStatic、REF_putStatic、REF_invokeStatic、REF_newInvokeSpecial四种类型的方法句
柄，并且这个方法句柄对应的类没有进行过初始化，则需要先触发其初始化。
6）当一个接口中定义了JDK 8新加入的默认方法（被default关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化，那该接口要在其之前被初始化。
```

对于这六种会触发类型进行初始化的场景，《Java虚拟机规范》中使用了一个非常强烈的限定语 ——"有且只有"，这六种场景中的行为称为对一个类型进行**主动引用**。除此之外，所有引用类型的方 式都不会触发初始化，称为被动引用，下面几个例子解释了被动引用

```
1.通过子类引用父类的静态字段，不会导致子类初始化//SubClass.value  value实际是父类字段
2.通过数组定义来引用类，不会触发此类的初始化//SuperClass[] sca = new SuperClass[10];
3.常量在编译阶段会存入调用类的常量池中，本质上没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化//public static final String HELLOWORLD = "hello world",
System.out.println(ConstClass.HELLOWORLD)
```

- 类加载的过程中初始化顺序没有严格的规定，就极有可能出现相同类每次初始化值不相等的情况,所以类的初始化一定是有顺序的

  ```
  //一个简单的过程描述
  1.父类的静态变量
  2.父类的静态初始化块
  3.子类的静态变量
  4.子类的静态初始化块
  5.父类的变量
  6.父类的初始化块
  7.父类的构造函数
  8.子类的变量
  9.子类的初始化块
  10.子类的构造函数
  ```

- java文件头的import

> import的存在纯粹是为了方便写代码，在编译后的class文件中没有import区。在Java源码编译器进行编译时，每个名字都会经过解析（resolution）找到其全名（canonical form）

- ClassNotFoundException

> 当程序试图使用Class.forname()、Classloader#findSystemClass()、Classloader#loadClass()方法通过字符串名的形式加载此类时，会抛出ClassNotFoundException

- 情景问题

```
@Configuration
class C {
    @Bean
    @ConditionalOnClass(A.class)
    A getBeanA(){
        return new A();
    }
    @Bean
    @ConditionalOnMissingClass("A")
    B getBeanB(){
        return new B();
    }
}
```

  - 开发时如果class A不存在，编译器会提示编译错误
    - 因此，开发时需要提供相关的包
  - 运行时当我们遇到class类不存在的时候会报ClassNotFoundException，但spring正确的选择了具体的实现类，这是为什么
    - 运行时只要不主动引用不存在的类，那么这些类就不会被加载，就不会产生ClassNotFoundException
      - 如果不主动引用的化，怎么分析注解上的类呢，这些类一开始并不直到是否存在
        - springboot 通过ConfigurationClassBeanDefinitionReader读取@Configuration注解标识的类。在扫描候选资源时，spring并没有通过Classloader#loadClass()来加载class文件，而是通过Classloader.getResource()获得class二进制文件，通过ClassReader对二进制文件进行ASM语法解析，从而得到候选类和注解的元数据。也就是说在spring扫描需要加载的Bean时，所有候选类都没有加载初始化

          > ASM是一个通用的JAVA字节码操纵和分析框架

        - 当spring获得候选bean的元信息时，需要判断这个bean是否真的需要被加载。这个就是spring的ConditionEvaluator体系，核心方法是ConditionEvaluator.shouldSkip(metadata)
        - 在获得ConditionalOnClass注解的元数据时还有个特殊处理，我们声明@ConditionalOnClass(A.class)时，使用的是Class类。为了后续操作时不引起class加载，AnnotatedElementUtils#getMergedAnnotationAttributes(element，annotationName，classValuesAsString，nestedAnnotationsAsMap)将class类转换成了class的名称字符串,在最终判断class是否存在时，jvm其实还是抛出了ClassNotFoundException，只是异常被吞没了

```
@Conditional(OnClassCondition.class)
public @interface ConditionalOnClass {
    /**
     * The classes that must be present. Since this
annotation is parsed by loading class
     * bytecode, it is safe to specify classes here that
may ultimately not be on the
     * classpath, only if this annotation is directly on
the affected component and
     * <b>not</b> if this annotation is used as a composed,
meta-annotation. In order to
     * use this annotation as a meta-annotation, only use
the {@link #name} attribute.
     * @return the classes that must be present
     */
    Class<?>[] value() default {};
/**
     * The classes names that must be present.
 * @return the class names that must be present.
     */
    String[] name() default {};
}
@Order(Ordered.HIGHEST_PRECEDENCE)
class OnClassCondition extends FilteringSpringBootCondition
```

类关系

```
abstract class FilteringSpringBootCondition extends
SpringBootCondition
     implements AutoConfigurationImportFilter,
BeanFactoryAware, BeanClassLoaderAware
```

核心匹配方法（matchs调用当前方法,matchs不同于AutoConfigurationImportFilter用于过滤的匹配方法match，方法签名不一样，调用时机不一样,AutoConfigurationImportFilter不用于shouldSkip时条件判断，而此处的会）

```
public ConditionOutcome getMatchOutcome(ConditionContext
context, AnnotatedTypeMetadata metadata) {
    ClassLoader classLoader = context.getClassLoader();
    ConditionMessage matchMessage =
ConditionMessage.empty();
    List<String> onClasses = getCandidates(metadata,
ConditionalOnClass.class);
    if (onClasses != null) {
        //最终内部调用isPresent进行判断
        List<String> missing = filter(onClasses,
ClassNameFilter.MISSING, classLoader);
        if (!missing.isEmpty()) {
            return
ConditionOutcome.noMatch(ConditionMessage.forCondition(Cond
itionalOnClass.class)
                                .didNotFind("required class",
"required classes").items(Style.QUOTE, missing));
        }
        matchMessage =
matchMessage.andCondition(ConditionalOnClass.class)
            .found("required class", "required classes")
            .items(Style.QUOTE, filter(onClasses,
ClassNameFilter.PRESENT, classLoader));
    }
```

```
    List<String> onMissingClasses = getCandidates(metadata,
ConditionalOnMissingClass.class);
    if (onMissingClasses != null) {
        List<String> present = filter(onMissingClasses,
ClassNameFilter.PRESENT, classLoader);
        if (!present.isEmpty()) {
            return
ConditionOutcome.noMatch(ConditionMessage.forCondition(Cond
itionalOnMissingClass.class)
                                .found("unwanted class",
"unwanted classes").items(Style.QUOTE, present));
        }
        matchMessage =
matchMessage.andCondition(ConditionalOnMissingClass.class)
            .didNotFind("unwanted class", "unwanted
classes")
            .items(Style.QUOTE, filter(onMissingClasses,
ClassNameFilter.MISSING, classLoader));
    }
    return ConditionOutcome.match(matchMessage);
}

static boolean isPresent(String className, ClassLoader
classLoader) { //ConditionalOnClass就是依据类是否存在进行匹配
    if (classLoader == null) {
        classLoader = ClassUtils.getDefaultClassLoader();
    }
    try {
        resolve(className, classLoader);
        return true;
    }
    catch (Throwable ex) {
        return false;
    }
}
protected static Class<?> resolve(String className,
ClassLoader classLoader) throws ClassNotFoundException {
    if (classLoader != null) {
        return Class.forName(className, false,
classLoader);
    }
    return Class.forName(className);
}
```

- 因此，前一点说的不主动引用不存在的类更强调的是容器初始化后，应用程序就不会再主动引用不存在的类了；但是在初始化过程中，容器自身还是进行了主动引用并处理了相关的ClassNotFoundException

> 参考文章：@ConditionalOnClass(A.class)为什么不报错

@ConditionalOnBean

```
@Conditional that only matches when beans meeting all the specified requirements
are already contained in the BeanFactory. All the requirements must be met for
the condition to match, but they do not have to be met by the same bean.
When placed on a @Bean method, the bean class defaults to the return type of the
factory method:
```

```
    @Configuration
    public class MyAutoConfiguration {
        @ConditionalOnBean
        @Bean
        public MyService myService() {
            ...
        }
    }
In the sample above the condition will match if a bean of type MyService is
already contained in the BeanFactory.
The condition can only match the bean definitions that have been processed by
the application context so far and, as such, it is strongly recommended to use
this condition on auto-configuration classes only. If a candidate bean may be
created by another auto-configuration, make sure that the one using this
condition runs after.仅推荐在自动配置类中使用
@Conditional(OnBeanCondition.class)
public @interface ConditionalOnBean {
    /**
     * The class types of beans that should be checked. The condition matches
when beans
     * of all classes specified are contained in the {@link BeanFactory}.
     * @return the class types of beans to check
     */
    Class<?>[] value() default {};//@ConditionalOnBean(DatabaseClient.class)
    /**
     * The class type names of beans that should be checked. The condition
matches when
     * beans of all classes specified are contained in the {@link BeanFactory}.
     * @return the class type names of beans to check
     */
    String[] type() default {};//@ConditionalOnBean(type =
"org.glassfish.jersey.server.ResourceConfig")
    /**
     * The annotation type decorating a bean that should be checked. The
condition matches
     * when all of the annotations specified are defined on beans in the
     * {@link BeanFactory}.
     * @return the class-level annotation types to check
     */
    Class<? extends Annotation>[] annotation() default {};
    /**
     * The names of beans to check. The condition matches when all of the bean
names
     * specified are contained in the {@link BeanFactory}.
     * @return the names of beans to check
     */
    String[] name() default {};
    /**
     * Strategy to decide if the application context hierarchy (parent contexts)
should be
     * considered.
     * @return the search strategy
     */
    SearchStrategy search() default SearchStrategy.ALL;
    /**
     * Additional classes that may contain the specified bean types within their
generic
```

```
     * parameters. For example, an annotation declaring {@code value=Name.class}
and
     * {@code parameterizedContainer=NameRegistration.class} would detect both
     * {@code Name} and {@code NameRegistration<Name>}.
     * @return the container types
     * @since 2.1.0
     */
    Class<?>[] parameterizedContainer() default {};//比如在ignored的type是
parameterizedContainer中类的泛型参数，那么type要忽略，parameterizedContainer也会考虑忽
略；ignored此处没有，但是ConditionalOnMissingBean中有；ignored换成type也是可以的，也有对
type类似处理
}
@Order(Ordered.LOWEST_PRECEDENCE)
class OnBeanCondition extends FilteringSpringBootCondition implements
ConfigurationCondition 类关系
abstract class FilteringSpringBootCondition extends SpringBootCondition
     implements AutoConfigurationImportFilter, BeanFactoryAware,
BeanClassLoaderAware

@Override
public ConfigurationPhase getConfigurationPhase() {//来自ConfigurationCondition接
口，OnClassCondition未实现该接口
    //shouldSkip只有在REGISTER_BEAN阶段，才会使用@ConditionalOnBean的匹配方法，其它时候
shouldSkip视该注解如不存在一样
    return ConfigurationPhase.REGISTER_BEAN;
}
核心匹配方法（matchs调用当前方法,matchs不同于AutoConfigurationImportFilter用于过滤的匹配
方法match，方法签名不一样，调用时机不一样,AutoConfigurationImportFilter不用于shouldSkip
时条件判断，而此处的会）
public ConditionOutcome getMatchOutcome(ConditionContext context,
AnnotatedTypeMetadata metadata) {
    ConditionMessage matchMessage = ConditionMessage.empty();
    MergedAnnotations annotations = metadata.getAnnotations();
    if (annotations.isPresent(ConditionalOnBean.class)) {
        //构建用于进行搜索的对象spec：context是
ConditionEvaluator.ConditionContextImpl类
        //Spec(ConditionContext context, AnnotatedTypeMetadata metadata,
MergedAnnotations annotations,
        //      Class<A> annotationType) {
        //  //解析出ConditionalOnBean注解上的所有属性
        //  MultiValueMap<String, Object> attributes =
annotations.stream(annotationType)
        //
.filter(MergedAnnotationPredicates.unique(MergedAnnotation::getMetaTypes))
        //
.collect(MergedAnnotationCollectors.toMultiValueMap(Adapt.CLASS_TO_STRING));
        //  MergedAnnotation<A> annotation = annotations.get(annotationType);
        //  this.classLoader = context.getClassLoader();
        //  this.annotationType = annotationType;
        //  this.names = extract(attributes, "name");
        //  this.annotations = extract(attributes, "annotation");
        //  this.ignoredTypes = extract(attributes, "ignored", "ignoredType");
        //  this.parameterizedContainers =
resolveWhenPossible(extract(attributes, "parameterizedContainer"));
        //public enum SearchStrategy {
        //    /** Search only the current context. */
        //    CURRENT,
        //    /**Search all ancestors, but not the current context. */
```

```
        //    ANCESTORS,
        //    /** Search the entire hierarchy.  */
        //    ALL
        //} //默认是ALL
        //  this.strategy = annotation.getValue("search",
SearchStrategy.class).orElse(null);
        //  从调试结果来看types就是@ConditionalOnBean注解携带的类信息
        //  Set<String> types = extractTypes(attributes);
        //  BeanTypeDeductionException deductionException = null;
        //  if (types.isEmpty() && this.names.isEmpty()) {
        //      try {
        //          private Set<String> deducedBeanType(ConditionContext
context, AnnotatedTypeMetadata metadata) {
        //                  if (metadata instanceof MethodMetadata &&
metadata.isAnnotated(Bean.class.getName())) {
        //                      //如果标注在@Bean的方法上，那么返回的是方法的返回类型
        //                      return deducedBeanTypeForBeanMethod(context,
(MethodMetadata) metadata);
        //                  }
        //                  return Collections.emptySet();//非标注了@Bean的方法直接返
回空
        //              }
        //          types = deducedBeanType(context, metadata);
        //      }
        //      catch (BeanTypeDeductionException ex) {
        //          deductionException = ex;
        //      }
        //  }
        //  this.types = types;
        //  //校验this.types, this.names, this.annotations 必须至少有一个不为空；否则
抛出异常
        //  validate(deductionException);
        //}
        Spec<ConditionalOnBean> spec = new Spec<>(context, metadata,
annotations, ConditionalOnBean.class);
        MatchResult matchResult = getMatchingBeans(context, spec);//该方法见下面详
细分析
        if (!matchResult.isAllMatched()) {
            String reason = createOnBeanNoMatchReason(matchResult);
            return ConditionOutcome.noMatch(spec.message().because(reason));
        }
        matchMessage = spec.message(matchMessage).found("bean",
"beans").items(Style.QUOTE,

matchResult.getNamesOfAllMatches());
    }
    if (metadata.isAnnotated(ConditionalOnSingleCandidate.class.getName())) {
        Spec<ConditionalOnSingleCandidate> spec = new
SingleCandidateSpec(context, metadata, annotations);
        MatchResult matchResult = getMatchingBeans(context, spec);
        if (!matchResult.isAllMatched()) {
            return ConditionOutcome.noMatch(spec.message().didNotFind("any
beans").atAll());
        }
        else if (!hasSingleAutowireCandidate(context.getBeanFactory(),
matchResult.getNamesOfAllMatches(),
                                             spec.getStrategy() ==
SearchStrategy.ALL)) {
```

```
            return ConditionOutcome.noMatch(spec.message().didNotFind("a primary
bean from beans")
                                                .items(Style.QUOTE,
matchResult.getNamesOfAllMatches()));
        }
        matchMessage = spec.message(matchMessage).found("a primary bean from
beans").items(Style.QUOTE,

          matchResult.getNamesOfAllMatches());
    }
    if (metadata.isAnnotated(ConditionalOnMissingBean.class.getName())) {
        Spec<ConditionalOnMissingBean> spec = new Spec<>(context, metadata,
annotations,

ConditionalOnMissingBean.class);
        MatchResult matchResult = getMatchingBeans(context, spec);
        if (matchResult.isAnyMatched()) {
            String reason = createOnMissingBeanNoMatchReason(matchResult);
            return ConditionOutcome.noMatch(spec.message().because(reason));
        }
        matchMessage = spec.message(matchMessage).didNotFind("any
beans").atAll();
    }
    return ConditionOutcome.match(matchMessage);
}
//这个方法会被上面
ConditionalOnBean/ConditionalOnSingleCandidate/ConditionalOnMissingBean多处使用
protected final MatchResult getMatchingBeans(ConditionContext context, Spec<?>
spec) {
    ClassLoader classLoader = context.getClassLoader();
    ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
    boolean considerHierarchy = spec.getStrategy() != SearchStrategy.CURRENT;
    Set<Class<?>> parameterizedContainers = spec.getParameterizedContainers();
    if (spec.getStrategy() == SearchStrategy.ANCESTORS) {
        BeanFactory parent = beanFactory.getParentBeanFactory();
        Assert.isInstanceOf(ConfigurableListableBeanFactory.class, parent,
                            "Unable to use SearchStrategy.ANCESTORS");
        beanFactory = (ConfigurableListableBeanFactory) parent;
    }
    MatchResult result = new MatchResult();
    //此处底层通过beanFactory.getBeanNamesForType搜索,该搜索包括beanDefinitionNames以
及手动注册的manualSingletonNames;
    //ignored虽然ConditionalOnBean没有但ConditionalOnMissingBean有
    Set<String> beansIgnoredByType = getNamesOfBeansIgnoredByType(classLoader,
beanFactory, considerHierarchy,

 spec.getIgnoredTypes(), parameterizedContainers);
    for (String type : spec.getTypes()) {
        //此处底层通过beanFactory.getBeanNamesForType搜索,该搜索包括
beanDefinitionNames以及手动注册的manualSingletonNames
        Collection<String> typeMatches = getBeanNamesForType(classLoader,
considerHierarchy, beanFactory, type,

parameterizedContainers);
        Iterator<String> iterator = typeMatches.iterator();
        while (iterator.hasNext()) {
            String match = iterator.next();
            //match属于忽略类型或者match名称以scopedTarget.开头
```

```
            if (beansIgnoredByType.contains(match) ||
ScopedProxyUtils.isScopedTarget(match)) {
                iterator.remove();//从typeMaatches中移出match
            }
        }
        if (typeMatches.isEmpty()) {
            result.recordUnmatchedType(type);//记录未匹配类型
        }
        else {
            result.recordMatchedType(type, typeMatches);//记录匹配类型
        }
    }
    for (String annotation : spec.getAnnotations()) {
        //底层通过beanFactory.getBeanNamesForAnnotation(annotationType)获取:
        //Find all names of beans which are annotated with the supplied
Annotation type,
        //without creating corresponding bean instances yet
        //更底层是通过beanDefinitionNames manualSingletonNames 遍历搜索
        Set<String> annotationMatches = getBeanNamesForAnnotation(classLoader,
beanFactory, annotation,

 considerHierarchy);
        annotationMatches.removeAll(beansIgnoredByType);
        if (annotationMatches.isEmpty()) {
            result.recordUnmatchedAnnotation(annotation);//记录未匹配
        }
        else {
            result.recordMatchedAnnotation(annotation, annotationMatches);//记录已
匹配
        }
    }
    for (String beanName : spec.getNames()) {
        //containsBean底层使用
beanFactory.containsBean(beanName)/beanFactory.containsLocalBean(beanName)进行判
断:
        //更底层则是containsSingleton(beanName) ||
containsBeanDefinition(beanName)进行判断
        if (!beansIgnoredByType.contains(beanName) && containsBean(beanFactory,
beanName, considerHierarchy)) {
            result.recordMatchedName(beanName);//记录未匹配
        }
        else {
            result.recordUnmatchedName(beanName);//记录匹配
        }
    }
    return result;
}
```

@ConditionalOnProperty

```
Property Value  havingValue=""  havingValue="true"  havingValue="false"
havingValue="foo"
"true"              yes             yes                 no                  no
"false"             no              no                  yes                 no
"foo"               yes             no                  no                  yes
```

```
If the property is not contained in the *Environment* at all, the
matchIfMissing() attribute is consulted. By default missing attributes do not
match.

@Conditional(OnPropertyCondition.class)
public @interface ConditionalOnProperty {

    /**
     * Alias for {@link #name()}.
     * @return the names
     */
    String[] value() default {};
    /**
     * A prefix that should be applied to each property. The prefix
automatically ends
     * with a dot if not specified. A valid prefix is defined by one or more
words
     * separated with dots (e.g. {@code "acme.system.feature"}).
     * @return the prefix
     */
    String prefix() default "";
    /**
     * The name of the properties to test. If a prefix has been defined, it is
applied to
     * compute the full key of each property. For instance if the prefix is
     * {@code app.config} and one value is {@code my-value}, the full key would
be
     * {@code app.config.my-value}
     * <p>
     * Use the dashed notation to specify each property, that is all lower case
with a "-"
     * to separate words (e.g. {@code my-long-property}).
     * @return the names
     */
    String[] name() default {};
    /**
     * The string representation of the expected value for the properties. If
not
     * specified, the property must <strong>not</strong> be equal to {@code
false}.
     * @return the expected value
     */
    String havingValue() default "";
    /**
     * Specify if the condition should match if the property is not set.
Defaults to
     * {@code false}.
     * @return if should match if the property is missing
     */
    boolean matchIfMissing() default false;
}
//e.g. 下面例子表示只有Environment中包含swagger.config.enabled=true属性，条件才匹
配,Swagger才可用
@ConditionalOnProperty(prefix="swagger.config", value="enabled",
havingValue="true")
@EnableSwagger
```

@AutoConfigureBefore

```
//Hint that an auto-configuration should be applied before other specified auto-configuration classes.
//As with standard @Configuration classes, the order in which auto-configuration classes are applied only affects the order in which their beans are defined. The order in which those beans are subsequently created is unaffected and is determined by each bean's dependencies and any @DependsOn relationships.
这里的顺序只会影响BeanDefinition定义的顺序，不会影响Bean实例创建的顺序，Bean实例创建顺序只和Bean本身的Bean依赖（构造、自动装配等依赖）和@DependsOn有关
public @interface AutoConfigureBefore {
    /**
     * The auto-configure classes that should have not yet been applied.
     * @return the classes
     */
    Class<?>[] value() default {};
    /**
     * The names of the auto-configure classes that should have not yet been applied.
     * @return the class names
     * @since 1.2.2
     */
    String[] name() default {};
}
```

@AutoConfigureAfter

```
//Hint for that an auto-configuration should be applied after other specified auto-configuration classes.在
//As with standard @Configuration classes, the order in which auto-configuration classes are applied only affects the order in which their beans are defined. The order in which those beans are subsequently created is unaffected and is determined by each bean's dependencies and any @DependsOn relationships.
这里的顺序只会影响BeanDefinition定义的顺序，不会影响Bean实例创建的顺序，Bean实例创建顺序只和Bean本身的Bean依赖（构造、自动装配等依赖）和@DependsOn有关
public @interface AutoConfigureAfter {
    /**
     * The auto-configure classes that should have already been applied.
     * @return the classes
     */
    Class<?>[] value() default {};
    /**
     * The names of the auto-configure classes that should have already been applied.
     * @return the class names
     * @since 1.2.2
     */
    String[] name() default {};
}
```

@DependsOn

```
//Beans on which the current bean depends. Any beans specified are guaranteed to be created by the container before this bean. Used infrequently in cases where a bean does not explicitly depend on another through properties or constructor arguments, but rather depends on the side effects of another bean's initialization 很少使用，仅在无法显示地通过属性和构造参数表示依赖关系时，但Bean加载的存在一些“副作用”需要指定先后顺序时，那么可以使用DependsOn
//A depends-on declaration can specify both an initialization-time dependency and, in the case of singleton beans only, a corresponding destruction-time dependency. Dependent beans that define a depends-on relationship with a given bean are destroyed first, prior to the given bean itself being destroyed. Thus, a depends-on declaration can also control shutdown order.还影响关闭销毁顺序
public @interface DependsOn {
    String[] value() default {};
}
```

Spring事务

> 事务的原子性/隔离性/持久性，都是为了服务于一致性，让数据库的数据从一个一致性状态达到另外一个一致性状态；原子性是解决单线程下的一致性问题；隔离性是让并发的事务像单线程事务一样有序执行，解决并发的一致性问题；持久性是让一致性状态得到永久保存

自动配置

> 在Spring Boot中，当我们使用了`spring-boot-starter-jdbc`或`spring-boot-starter-data-jpa`依赖的时候，框架会自动默认分别注入`DataSourceTransactionManager`或`JpaTransactionManager`
>
> ```
> <dependency>
> <groupId>org.springframework.boot</groupId>
> <artifactId>spring-boot-starter-jdbc</artifactId>
> </dependency>
> ```
>
> `spring-boot-starter-jdbc`提供了数据源配置、事务管理、数据访问等等功能，而对于不同类型的数据库，需要提供不同的驱动实现，比如`mysql`：
>
> ```
> <dependency>
> <groupId>mysql</groupId>
> <artifactId>mysql-connector-java</artifactId>
> </dependency>
> ```
>
> `spring-boot-starter-jdbc`引入了`spring-jdbc`和`HikariCP`(性能极好的一个数据库连接池)，而`spring-jdbc`又引入了`spring-tx`(事务相关，仅仅使用web-starter是不会引入事务的)
>
> 一个可供参考的配置文章：SpringBoot2.x：引入jdbc模块与JdbcTemplate简单使用
>
> 一个简单的搭建mysql的docker镜像：Docker 安装 MySQL
>
> 如果不参考文章的话，可以直接开启debug=true，查看错误，定位必填的属性；另外debug还能查看具体哪些配置类被加载

```
@Bean
```

```
@ConfigurationProperties(prefix = "spring.datasource.hikari")
HikariDataSource dataSource(DataSourceProperties properties) {
    //@ConfigurationProperties(prefix = "spring.datasource")
    //public class DataSourceProperties implements BeanClassLoaderAware,
InitializingBean
    //properties 来源于 spring.datasource，可以用它初始化HikariDataSource一部分信息
    //返回对象又会被spring.datasource.hikari的属性进行设置
    HikariDataSource dataSource = createDataSource(properties,
HikariDataSource.class);
    if (StringUtils.hasText(properties.getName())) {
        dataSource.setPoolName(properties.getName());
    }
    return dataSource;
}
```

数据库连接池的几个核心设置为：

```
spring.datasource.url=jdbc:mysql://localhost:3306
spring.datasource.username=user
spring.datasource.password=pwd

@Bean
@ConditionalOnMissingBean(TransactionManager.class)
DataSourceTransactionManager transactionManager(Environment environment,
DataSource dataSource,
                        ObjectProvider<TransactionManagerCustomizers>
transactionManagerCustomizers) {
    DataSourceTransactionManager transactionManager =
createTransactionManager(environment, dataSource);
    transactionManagerCustomizers.ifAvailable((customizers) ->
customizers.customize(transactionManager));
    return transactionManager;//调试时最终默认返回JdbcTransactionManager
}
private DataSourceTransactionManager createTransactionManager(Environment
environment, DataSource dataSource) {
    //调试时，此处默认返回JdbcTransactionManager
    //public class JdbcTransactionManager extends DataSourceTransactionManager
    //public class DataSourceTransactionManager extends
AbstractPlatformTransactionManager implements
    //ResourceTransactionManager, InitializingBean
    //public abstract class AbstractPlatformTransactionManager implements
    //PlatformTransactionManager, Serializable
    //public interface PlatformTransactionManager extends TransactionManager {
    //    TransactionStatus getTransaction(@Nullable TransactionDefinition
definition) throws
    //        TransactionException;
    //    void commit(TransactionStatus status) throws TransactionException;
    //    void rollback(TransactionStatus status) throws TransactionException;
    //}
    return environment.getProperty("spring.dao.exceptiontranslation.enabled",
Boolean.class, Boolean.TRUE)
        ? new JdbcTransactionManager(dataSource) : new
DataSourceTransactionManager(dataSource);
}

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ DataSource.class, JdbcTemplate.class })
@ConditionalOnSingleCandidate(DataSource.class)
@AutoConfigureAfter(DataSourceAutoConfiguration.class)
@EnableConfigurationProperties(JdbcProperties.class)
```

```
@Import({ DatabaseInitializationDependencyConfigurer.class,
JdbcTemplateConfiguration.class,
        NamedParameterJdbcTemplateConfiguration.class })//关注
JdbcTemplateConfiguration
public class JdbcTemplateAutoConfiguration {
}
@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(JdbcOperations.class)
class JdbcTemplateConfiguration {
    @Bean
    @Primary//创建JdbcTemplate，其是对JDBC的轻度封装
    JdbcTemplate jdbcTemplate(DataSource dataSource, JdbcProperties properties)
{
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        JdbcProperties.Template template = properties.getTemplate();
        jdbcTemplate.setFetchSize(template.getFetchSize());
        jdbcTemplate.setMaxRows(template.getMaxRows());
        if (template.getQueryTimeout() != null) {
            jdbcTemplate.setQueryTimeout((int)
template.getQueryTimeout().getSeconds());
        }
        return jdbcTemplate;
    }
}

@Configuration(proxyBeanMethods = false)
@EnableTransactionManagement(proxyTargetClass = true)//默认的自动配置就激活了事务管理
@ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class",
havingValue = "true",
                       matchIfMissing = true)
public static class CglibAutoProxyConfiguration {
}
```

使用以及注意事项

@Transactional、Propagation、Isolation

```
public @interface Transactional {
    @AliasFor("transactionManager")
    String value() default "";
    @AliasFor("value")
    String transactionManager() default "";//当应用程序需要多个事务管理器时可以使用
    /**
     * Defines zero (0) or more transaction labels.
     * <p>Labels may be used to describe a transaction, and they can be
evaluated
     * by individual transaction managers.
     * @since 5.3
     */
    String[] label() default {};
    /**
     * The transaction propagation type.
     */
    Propagation propagation() default Propagation.REQUIRED;
    /**
     * The transaction isolation level.
```

```
     * Defaults to {@link Isolation#DEFAULT}.
     *
     */
    Isolation isolation() default Isolation.DEFAULT;
    /**
     * The timeout for this transaction (in seconds).
     * Defaults to the default timeout of the underlying transaction system.
     * mysql 通过innodb_rollback_on_timeout控制锁超时时间，默认50s
     * innodb_rollback_on_timeout：用来设定是否在等待超时时对进行中的事务进行回滚操作。默认是OFF，不回滚,主要由
     * 用户决定是否进行回滚（如果用户网络断开，那么该事务后面必然会回滚）
     */
    int timeout() default TransactionDefinition.TIMEOUT_DEFAULT;
    /**
     * The timeout for this transaction (in seconds).
     * Defaults to the default timeout of the underlying transaction system.
     * @return the timeout in seconds as a String value, e.g. a placeholder
     * @since 5.3
     */
    String timeoutString() default "";
    /**
     * A boolean flag that can be set to {@code true} if the transaction is
     * effectively read-only, allowing for corresponding optimizations at runtime.
     * Defaults to {@code false}.
     * This just serves as a hint for the actual transaction subsystem;
     */
    boolean readOnly() default false;
    /**
     * Defines zero (0) or more exception {@linkplain Class classes}, which must be
     * subclasses of {@link Throwable}, indicating which exception types must cause
     * a transaction rollback.
     * By default, a transaction will be rolled back on RuntimeException
     * and Error but not on checked exceptions (business exceptions).
     */
    Class<? extends Throwable>[] rollbackFor() default {};
    /**
     * Defines zero (0) or more exception name patterns (for exceptions which must be a
     * subclass of {@link Throwable}), indicating which exception types must cause
     * a transaction rollback.
     */
    String[] rollbackForClassName() default {};
    /**
     * Defines zero (0) or more exception {@link Class Classes}, which must be
     * subclasses of {@link Throwable}, indicating which exception types must
     * not cause a transaction rollback.
     */
    Class<? extends Throwable>[] noRollbackFor() default {};
    /**
     * Defines zero (0) or more exception name patterns (for exceptions which must be a
     * subclass of {@link Throwable}) indicating which exception types must not
     * cause a transaction rollback.
     */
```

```
    String[] noRollbackForClassName() default {};
}
public enum Propagation {//事务传播
    /**
     * Support a current transaction, create a new one if none exists.
     * Spring默认的事务传播类型
     * 如果当前没有事务，则自己新建一个事务，如果当前存在事务，则加入这个事务
     */
    REQUIRED(TransactionDefinition.PROPAGATION_REQUIRED),
    /**
     * Support a current transaction, execute non-transactionally if none
exists.
     * 如果当前没有事务，则自己新建一个事务，如果当前存在事务，则加入这个事务
     */
    SUPPORTS(TransactionDefinition.PROPAGATION_SUPPORTS),
    /**
     * Support a current transaction, throw an exception if none exists.
     * 当前存在事务，则加入当前事务，如果当前事务不存在，则抛出异常
     */
    MANDATORY(TransactionDefinition.PROPAGATION_MANDATORY),
    /**
     * Create a new transaction, and suspend the current transaction if one
exists.
     * 创建一个新事务；如果存在当前事务，则挂起该事务
     */
    REQUIRES_NEW(TransactionDefinition.PROPAGATION_REQUIRES_NEW),
    /**
     * Execute non-transactionally, suspend the current transaction if one
exists.
     * 始终以非事务方式执行,如果当前存在事务，则挂起当前事务
     */
    NOT_SUPPORTED(TransactionDefinition.PROPAGATION_NOT_SUPPORTED),
    /**
     * Execute non-transactionally, throw an exception if a transaction exists.
     * 不使用事务，如果当前事务存在，则抛出异常
     */
    NEVER(TransactionDefinition.PROPAGATION_NEVER),
    /**
     * Execute within a nested transaction if a current transaction exists,
     * behave like {@code REQUIRED} otherwise.
     * 如果当前事务存在，则在嵌套事务中执行，否则REQUIRED的操作一样（开启一个事务）
     */
    NESTED(TransactionDefinition.PROPAGATION_NESTED);
    private final int value;
    Propagation(int value) {
        this.value = value;
    }
    public int value() {
        return this.value;
    }
}
public enum Isolation {
    /**
     * Use the default isolation level of the underlying datastore.
     * 使用底层数据库的默认隔离级别，MYSQL：默认为REPEATABLE_READ级别，其它的一般为
READ_COMMITTED
     * All other levels correspond to the JDBC isolation levels.
     */
```

```
    DEFAULT(TransactionDefinition.ISOLATION_DEFAULT),
    /**
     * A constant indicating that dirty reads, non-repeatable reads and phantom
reads
     * can occur. This level allows a row changed by one transaction to be read
by
     * another transaction before any changes in that row have been committed
     * (a "dirty read"). If any of the changes are rolled back, the second
     * transaction will have retrieved an invalid row.读未提交
     */
    READ_UNCOMMITTED(TransactionDefinition.ISOLATION_READ_UNCOMMITTED),
    /**
     * A constant indicating that dirty reads are prevented; non-repeatable
reads
     * and phantom reads can occur. This level only prohibits a transaction
     * from reading a row with uncommitted changes in it.读已提交
     */
    READ_COMMITTED(TransactionDefinition.ISOLATION_READ_COMMITTED),
    /**
     * A constant indicating that dirty reads and non-repeatable reads are
     * prevented; phantom幻读 reads can occur. This level prohibits a transaction
     * from reading a row with uncommitted changes in it, and it also prohibits
     * the situation where one transaction reads a row, a second transaction
     * alters the row, and the first transaction rereads the row, getting
     * different values the second time (a "non-repeatable read").//可重复读
     */
    REPEATABLE_READ(TransactionDefinition.ISOLATION_REPEATABLE_READ),
    /**
     * A constant indicating that dirty reads, non-repeatable reads and phantom
     * reads are prevented. This level includes the prohibitions in
     * {@code ISOLATION_REPEATABLE_READ} and further prohibits the situation
     * where one transaction reads all rows that satisfy a {@code WHERE}
     * condition, a second transaction inserts a row that satisfies that
     * {@code WHERE} condition, and the first transaction rereads for the
     * same condition, retrieving the additional "phantom" row in the second
read.
     * @see java.sql.Connection#TRANSACTION_SERIALIZABLE
     */
    SERIALIZABLE(TransactionDefinition.ISOLATION_SERIALIZABLE);//串行化
    private final int value;
    Isolation(int value) {
        this.value = value;
    }
    public int value() {
        return this.value;
    }
}
```

@Transactional不生效的9种情况

> 不生效的原因一部分是因为数据库引擎不支持，另一部分原因是”AOP“以及动态代理产生的问题；并且鉴于这么多的特别情况，如果自己无法保证都能预想到，可以针对事务方法写test()测试进行检测

1. **数据库引擎不支持事务**，以 MySQL 为例，其 MyISAM 引擎是不支持事务操作的，InnoDB 才是支持事务的引擎
2. **没有被 Spring 管理**，只有被Spring管理的Bean才能进行AOP以及动态代理

```
You can also use the Spring Framework's @Transactional support outside of a
Spring container by
means of an AspectJ aspect. To do so, first annotate your classes (and
optionally your classes'
methods) with the @Transactional annotation, and then link (weave) your
application with the
org.springframework.transaction.aspectj.AnnotationTransactionAspect defined
in the spring aspects.jar file. You must also configure the aspect with a
transaction manager.  实际在spring容器外部是可用的，只是需要用到AspectJ,暂不深入分析
```

3. **方法不是 public**

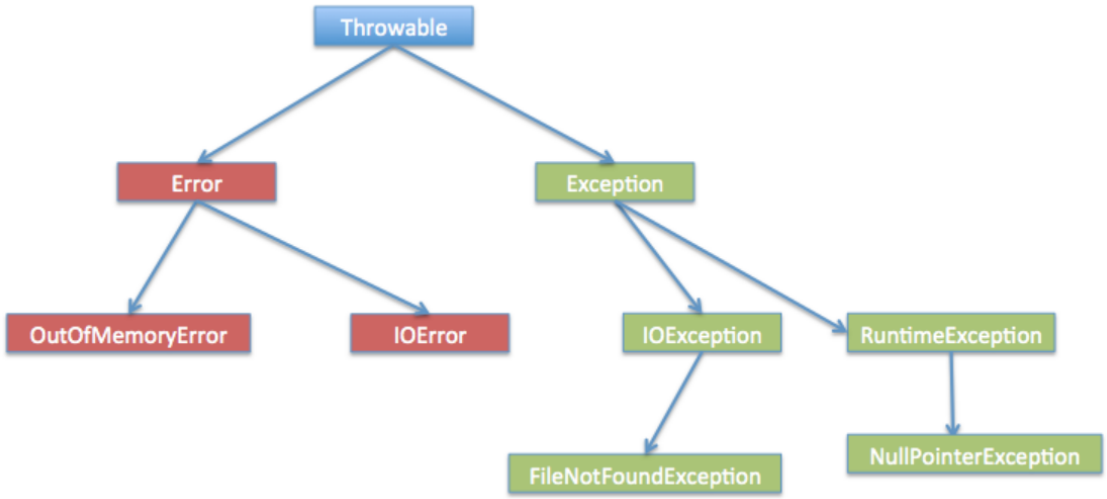
```
When you use transactional proxies with Spring's standard configuration, you
should apply the @Transactional annotation only to methods with public
visibility.If you do annotate protected, private, or package-visible methods
with the @Transactional annotation, no error is raised, but the annotated
method does not exhibit the configured transactional settings. If you need
to annotate non-public
methods, consider the tip in the following paragraph for class-based proxies
or consider using AspectJ compile-time or load-time weaving (described
later).说明非公共方法也是可行的，但是必须是静态织入代理
```

4. **自身调用问题**

```
In proxy mode (which is the default), only external method calls coming in
through the proxy are intercepted. This means that self-invocation (in
effect, a method within the target object calling another method of the
target object) does not lead to an actual transaction at runtime even if the
invoked method is marked with @Transactional. 这个问题产生的原因和动态代理的有关，
当CGLIB增强时，会产生两个对象，代理对象和实际对象，所有的数据都在实际对象那，代理对象仅仅起
到拦截方法的作用，在实际对象中直接使用this时，this指向实际对象而不是代理对象；为了实现目
标，需要把代理对象通过自动装配引入，一种不是很优雅的方式是自己自动装配自己的类型对象，使用时
不用this而是使用自动装配的对象，可以达到效果
```

5. **数据源没有配置事务管理器**，默认情况下SpringBoot会自动配置一个事务管理器，但多事务管理时需要自行配置
6. **@Transactional使用了不支持的属性**，比如Propagation.NOT_SUPPORTED
7. **异常被吃了**，方法内部trycatch把异常吃了，然后又不抛出来，事务捕捉不到异常也就无法回滚
8. **异常类型错误**，默认回滚的是RuntimeException and Error，如果你想触发其他异常的回滚，需要在注解上配置一下比如@Transactional(rollbackFor = Throwable.class)，这个配置仅限于 Throwable 异常类及其子类

简单补充一下：
检查性异常：The class Exception and any subclasses that are not also subclasses of RuntimeException are checked exceptions.Checked exceptions need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.编译时就会被检查，比如文件不存在异常
运行时异常：RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.比如ArithmeticException extends RuntimeException就产生于除0异常
错误(Error)：错误不是异常，而是脱离程序员控制的问题，比如OOM；A method is not required to declare in its throws clause any subclasses of Error that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur，其内在含义是需要捕获RuntimeException和Checked Exception，但是不要捕获Error;如果发生Error，通常意味着程序已经不可能再做任何恢复；虽然如此，但确实有些时候还是需要用Throwable来捕获Error（不到万不得已不要用）,比如尝试优雅的关闭或打印日志等，但不要所有地方都使用Throwable



9. **多线程调用**，Spring通过ThreadLocal把数据库连接和当前线程绑定,不同的线程数据库连接不一致的话，事务也就不可能是同一个

```
@Transactional
public void doSomething(){
    new Thread(()->{doOtherthing();}) .start();//这里面doOtherthing事务就失效了
}
```

SpringMVC

自动配置

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class,
WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class,
TaskExecutionAutoConfiguration.class,
```

```
        ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {

}

@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass(DispatcherServlet.class)
@AutoConfigureAfter(ServletWebServerFactoryAutoConfiguration.class)
public class DispatcherServletAutoConfiguration {//涉及dispatcherServlet创建
}

@Configuration(proxyBeanMethods = false)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@ConditionalOnClass(ServletRequest.class)
@ConditionalOnWebApplication(type = Type.SERVLET)
@EnableConfigurationProperties(ServerProperties.class)
@Import({
ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class,
        ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,//Tomcat工厂
        ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
        ServletWebServerFactoryConfiguration.EmbeddedUndertow.class })
public class ServletWebServerFactoryAutoConfiguration {
}

@Configuration(proxyBeanMethods = false)
class ServletWebServerFactoryConfiguration {
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass({ Servlet.class, Tomcat.class, UpgradeProtocol.class })
    @ConditionalOnMissingBean(value = ServletWebServerFactory.class, search =
SearchStrategy.CURRENT)
    static class EmbeddedTomcat {
        @Bean
        TomcatServletWebServerFactory tomcatServletWebServerFactory(
                ObjectProvider<TomcatConnectorCustomizer> connectorCustomizers,
                ObjectProvider<TomcatContextCustomizer> contextCustomizers,
                ObjectProvider<TomcatProtocolHandlerCustomizer<?>>
protocolHandlerCustomizers) {
            TomcatServletWebServerFactory factory = new
TomcatServletWebServerFactory();
            factory.getTomcatConnectorCustomizers()

.addAll(connectorCustomizers.orderedStream().collect(Collectors.toList()));
            factory.getTomcatContextCustomizers()

.addAll(contextCustomizers.orderedStream().collect(Collectors.toList()));
            factory.getTomcatProtocolHandlerCustomizers()

.addAll(protocolHandlerCustomizers.orderedStream().collect(Collectors.toList()))
;
            return factory;
        }//TomcatServletWebServerFactory.getWebServer得到WebServer
    }
}
//更多细节可以在使用和调试时分析
```

流程分析

调用栈

Tomcat三大组件（或者成为JavaWeb三大组件），Servlet（运行在服务端的小程序）、Filter（过滤器）、Listener（监听特定事件）

一次请求的调用栈(hello部分是栈顶，run部分是栈底，调用顺序需要从下往上看)

```
 at com.example.controller.Hello.hello(Hello.java:49)
 at
sun.reflect.NativeMethodAccessorImpl.invoke0(NativeMethodAccessorImpl.java:-1)
 at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
 at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav
a:43)
 at java.lang.reflect.Method.invoke(Method.java:498)
 at
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(Invocable
HandlerMethod.java:197)
 at
org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(I
nvocableHandlerMethod.java:141)
 at
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMet
hod.invokeAndHandle(ServletInvocableHandlerMethod.java:106)
 at
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapt
er.invokeHandlerMethod(RequestMappingHandlerAdapter.java:894)
 at
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapt
er.handleInternal(RequestMappingHandlerAdapter.java:808)
 at
org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(A
bstractHandlerMethodAdapter.java:87)
 at
org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.j
ava:1060)
 at
org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.ja
va:962)
 //doService会向request中注入一些属性，然后调用doDispatch--真正的核心处理方法，将在下面展开
分析
 at
org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet
.java:1006)
 at
org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:898
)
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:626)
 at
org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:8
83)
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:733)
```

```
 //从这往上是servlet.service(request, response)处理，servlet即DispatcherServlet，是javaWeb三大组件之一
 at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:227)
 at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:162)
 at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
 at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:189)
 at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:162)
 at org.springframework.web.filter.RequestContextFilter.doFilterInternal(RequestContextFilter.java:100)
 at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:119)
 at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:189)
 at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:162)
 at org.springframework.web.filter.FormContentFilter.doFilterInternal(FormContentFilter.java:93)
 at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:119)
 at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:189)
 at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:162)
 at org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:201)
 at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:119)
 at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:189)
 at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:162)
 //从这往上是FilterChian处理（Filter组件处理）,Filter按链式处理，是javaWeb三大组件之一
 at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:202)
 at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:97)
```

```
 at
org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.jav
a:542)
 at
org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:143)
 at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:92)
 at
org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:78)
 //---从这往上是Tomcat内部Valve处理
 at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:346)
 at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:374)
 at
org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:65)
 at
org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.ja
va:887)
 at
org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:16
84)
 at
org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
 - locked <0x1f9c> (a org.apache.tomcat.util.net.NioEndpoint$NioSocketWrapper)
 at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
 at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
 at
org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:6
1)
 at java.lang.Thread.run(Thread.java:748)
```

doDispatch

```
public class DispatcherServlet extends FrameworkServlet {
    //这下面是DispatcherSevlet九大组件名称，对应到DispatcherSevlet中initStrategies方法
初始化
    /** Well-known name for the MultipartResolver object in the bean factory for
this namespace. */
    public static final String MULTIPART_RESOLVER_BEAN_NAME =
"multipartResolver";//文件上传相关
    /** Well-known name for the LocaleResolver object in the bean factory for
this namespace. */
    public static final String LOCALE_RESOLVER_BEAN_NAME = "localeResolver";
    /** Well-known name for the ThemeResolver object in the bean factory for this
namespace. */
    public static final String THEME_RESOLVER_BEAN_NAME = "themeResolver";
    /**
     * Well-known name for the HandlerMapping object in the bean factory for
this namespace.
     * Only used when "detectAllHandlerMappings" is turned off.
     * @see #setDetectAllHandlerMappings
     */
    public static final String HANDLER_MAPPING_BEAN_NAME = "handlerMapping";
    /**
     * Well-known name for the HandlerAdapter object in the bean factory for
this namespace.
     * Only used when "detectAllHandlerAdapters" is turned off.
```

```
     * @see #setDetectAllHandlerAdapters
     */
    public static final String HANDLER_ADAPTER_BEAN_NAME = "handlerAdapter";
    /**
     * Well-known name for the HandlerExceptionResolver object in the bean
factory for this namespace.
     * Only used when "detectAllHandlerExceptionResolvers" is turned off.
     * @see #setDetectAllHandlerExceptionResolvers
     */
    public static final String HANDLER_EXCEPTION_RESOLVER_BEAN_NAME =
"handlerExceptionResolver";
    /**
     * Well-known name for the RequestToViewNameTranslator object in the bean
factory for this namespace.
     */
    public static final String REQUEST_TO_VIEW_NAME_TRANSLATOR_BEAN_NAME =
"viewNameTranslator";
    /**
     * Well-known name for the ViewResolver object in the bean factory for this
namespace.
     * Only used when "detectAllViewResolvers" is turned off.
     * @see #setDetectAllViewResolvers
     */
    public static final String VIEW_RESOLVER_BEAN_NAME = "viewResolver";
    /**
     * Well-known name for the FlashMapManager object in the bean factory for
this namespace.
     */
    public static final String FLASH_MAP_MANAGER_BEAN_NAME = "flashMapManager";

    protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        boolean multipartRequestParsed = false;
        //The central class for managing asynchronous request processing;request
中有则使用，无则创建并放入request
        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
        try {
            ModelAndView mv = null;
            Exception dispatchException = null;
            try {
                //protected HttpServletRequest checkMultipart(HttpServletRequest
request) throws MultipartException {
                //    //检测是否为文件上传请求；PartMart表示传的是二进制流，通常就是指文件
                //    if (this.multipartResolver != null &&
this.multipartResolver.isMultipart(request)) {
                //        //如果request是一个封装类型，那么尝试解出内部封装的类型，直到其
为MultipartHttpServletRequest，那么就返回它
                //        //否则，解封不出指定类型就返回null
                //        if (WebUtils.getNativeRequest(request,
MultipartHttpServletRequest.class) != null) {
                //            if
(request.getDispatcherType().equals(DispatcherType.REQUEST)) {
                //                logger.trace("Request already resolved to
MultipartHttpServletRequest, e.g. by“ +
                //                                "MultipartFilter");
                //            }
```

```
                //        }
                //        else if (hasMultipartException(request)) {
                //            logger.debug("Multipart resolution previously
failed for current request - " +
                //                    "skipping re-resolution for undisturbed
error rendering");
                //        }
                //        else {
                //            try {
                //              //Parse the given HTTP request into multipart
files and parameters, and wrap the request inside
                //              //a MultipartHttpServletRequest object that
provides access to file descriptors and makes
                //              //contained parameters accessible via the
standard ServletRequest methods.
                //              //返回值是MultipartHttpServletRequest，提供了对文件描
述符的访问以及包含参数的访问
                //              return
this.multipartResolver.resolveMultipart(request);
                //            }
                //            catch (MultipartException ex) {
                //                if
(request.getAttribute(WebUtils.ERROR_EXCEPTION_ATTRIBUTE) != null) {
                //                    logger.debug("Multipart resolution failed
for error dispatch", ex);
                //                    // Keep processing error dispatch with
regular request handle below
                //                }
                //                else {
                //                    throw ex;
                //                }
                //            }
                //        }
                //    }
                //    // If not returned before: return original request.
                //    return request;
                //}
                //这里就是如果request是一个文件类型，那么返回的是
MultipartHttpServletRequest；否则，就是原本的request
                processedRequest = checkMultipart(request);
                multipartRequestParsed = (processedRequest != request);//不相等表
示一定经过转换解析得到MultipartHttpServletRequest
                // Return the HandlerExecutionChain for this request.
                // Tries all handler mappings in order.
                //protected HandlerExecutionChain getHandler(HttpServletRequest
request) throws Exception {
                //    if (this.handlerMappings != null) {
                //        for (HandlerMapping mapping : this.handlerMappings) {
                //            //这里调试时有5个，第一个为
RequestMappingHandlerMapping，通常在它这就返回了，下文会对它展开详细分析
                //            HandlerExecutionChain handler =
mapping.getHandler(request);
                //            if (handler != null) {
                //                return handler;
                //            }
                //        }
                //    }
                //    return null;
```

```
                //}
                // Determine handler for the current request.
                mappedHandler = getHandler(processedRequest);
                if (mappedHandler == null) {
                    noHandlerFound(processedRequest, response);
                    return;
                }
                // Determine handler adapter for the current request.
                //protected HandlerAdapter getHandlerAdapter(Object handler)
throws ServletException {
                //    if (this.handlerAdapters != null) {
                //        //调试时默认有4个，返回的是RequestMappingHandlerAdapter
                //        for (HandlerAdapter adapter : this.handlerAdapters) {
                //            if (adapter.supports(handler)) {
                //                return adapter;
                //            }
                //        }
                //    }
                //    throw new ServletException("No adapter for handler [" +
handler +
                //            "]: The DispatcherServlet configuration needs to
include a HandlerAdapter that supports\
                //                                    this handler");
                //}HandlerAdapter:MVC framework SPI, allowing parameterization of
the core MVC workflow.自定义一部分MVC工作流程
                HandlerAdapter ha =
getHandlerAdapter(mappedHandler.getHandler());
                // Process last-modified header, if supported by the handler.
                String method = request.getMethod();
                boolean isGet = "GET".equals(method);
                if (isGet || "HEAD".equals(method)) {
                    long lastModified = ha.getLastModified(request,
mappedHandler.getHandler());
                    if (new ServletWebRequest(request,
response).checkNotModified(lastModified) && isGet) {//调试时一般的请求为false
                        return;
                    }
                }
                //boolean applyPreHandle(HttpServletRequest request,
HttpServletResponse response) throws Exception {
                //    for (int i = 0; i < this.interceptorList.size(); i++) {
                //        HandlerInterceptor interceptor =
this.interceptorList.get(i);
                //        //循环遍历所有的HandleInterceptor，一般而言必须所有的
preHandle都返回true才行，当返回false，那么这个请求就直接
                //        //逆向执行所有已经执行过preHandle方法的HandlerInterceptor的
afterCompletion
                //        if (!interceptor.preHandle(request, response,
this.handler)) {
                //            //void triggerAfterCompletion(HttpServletRequest
request, HttpServletResponse response,
                //            //                            @Nullable Exception
ex) {
                //            //    for (int i = this.interceptorIndex; i >= 0;
i--) {
                //            //        HandlerInterceptor interceptor =
this.interceptorList.get(i);
                //            //        try {
```

```
                //            //            interceptor.afterCompletion(request,
response, this.handler, ex);
                //            //        }
                //            //        catch (Throwable ex2) {
                //            //
logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);
                //            //        }
                //            //    }
                //           //}
                //            triggerAfterCompletion(request, response, null);
                //            return false;
                //        }
                //        this.interceptorIndex = i;
                //    }
                //    return true;
                //}
                if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                    return;
                }
                // Actually invoke the handler. 真正执行Controller中的处理方法，
reolve返回值，并将处理结果封装成ModelAndView返回
                // 详细见后文分析
                mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());
                if (asyncManager.isConcurrentHandlingStarted()) {
                    return;
                }
                applyDefaultViewName(processedRequest, mv);//Do we need view
name translation 在调试时的普通get请求相当于什么也没执行
                //void applyPostHandle(HttpServletRequest request,
HttpServletResponse response, @Nullable ModelAndView mv)
                //          throws Exception {
                //    for (int i = this.interceptorList.size() - 1; i >= 0; i--)
{//从后往前执行HandlerInterceptor的postHandle方法
                //        HandlerInterceptor interceptor =
this.interceptorList.get(i);
                //        interceptor.postHandle(request, response, this.handler,
mv);
                //    }
                //}
                mappedHandler.applyPostHandle(processedRequest, response, mv);
            }
            catch (Exception ex) {
                dispatchException = ex;
            }
            catch (Throwable err) {
                // As of 4.3, we're processing Errors thrown from handler methods
as well,
                // making them available for @ExceptionHandler methods and other
scenarios.
                dispatchException = new NestedServletException("Handler dispatch
failed", err);
            }
            //详细见后文分析，主要是渲染工作
            processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
        }
        catch (Exception ex) {
```

```
            triggerAfterCompletion(processedRequest, response, mappedHandler,
ex);//异常时触发AfterCompletion
        }
        catch (Throwable err) {
            triggerAfterCompletion(processedRequest, response, mappedHandler,//异
常时触发AfterCompletion
                                   new NestedServletException("Handler
processing failed", err));
        }
        finally {
            if (asyncManager.isConcurrentHandlingStarted()) {
                // Instead of postHandle and afterCompletion
                if (mappedHandler != null) {

 mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
                }
            }
            else {
                // Clean up any resources used by a multipart request.
                if (multipartRequestParsed) {
                    cleanupMultipart(processedRequest);
                }
            }
        }
    }
}

//前文中RequestMappingHandlerMapping.getHandler的分析：（实际调用是其父类
AbstractHandlerMapping的方法）
public final HandlerExecutionChain getHandler(HttpServletRequest request) throws
Exception {
    //见后文分析，核心是根据request以及@RequestMapping信息（即RequestMappingInfo）获取最
合适的处理方法
    Object handler = getHandlerInternal(request);
    if (handler == null) {//调试时未走此分支
        handler = getDefaultHandler();//此处返回Object，可能为String
    }
    if (handler == null) {
        return null;
    }
    // Bean name or resolved handler?
    if (handler instanceof String) {
        String handlerName = (String) handler;
        handler = obtainApplicationContext().getBean(handlerName);//从Spring容器中
通过getBean获取（不存在就还会创建）
    }
    // Ensure presence of cached lookupPath for interceptors and others：检测
request中是否有相关属性
    if (!ServletRequestPathUtils.hasCachedPath(request)) {
        initLookupPath(request);
    }
    //整个方法相当于创建一个HandlerExecutionChain，然后加入daptedInterceptors中所有的
HandlerInterceptor
    //protected HandlerExecutionChain getHandlerExecutionChain(Object handler,
HttpServletRequest request) {
    //  HandlerExecutionChain chain = (handler instanceof HandlerExecutionChain
?
```

```
    //          (HandlerExecutionChain) handler : new
HandlerExecutionChain(handler));
    //  //调试时，此处有两个HandlerInterceptor,一个是
ConversionServiceExposingInterceptor，一个是ResourceUrlProviderExposingInterceptor
    //  for (HandlerInterceptor interceptor : this.adaptedInterceptors) {
    //      if (interceptor instanceof MappedInterceptor) {
    //          MappedInterceptor mappedInterceptor = (MappedInterceptor)
interceptor;
    //          if (mappedInterceptor.matches(request)) {
    //              chain.addInterceptor(mappedInterceptor.getInterceptor());
    //          }
    //      }
    //      else {
    //          chain.addInterceptor(interceptor);//调试时都走这
    //      }
    //  }
    //  return chain;
    //}
    HandlerExecutionChain executionChain = getHandlerExecutionChain(handler,
request);
    if (logger.isTraceEnabled()) {
        logger.trace("Mapped to " + handler);
    }
    else if (logger.isDebugEnabled() &&
!request.getDispatcherType().equals(DispatcherType.ASYNC)) {
        logger.debug("Mapped to " + executionChain.getHandler());
    }
    //和CORS Cross Origin Resource Share 跨域资源共享相关，一般的访问在调试时未走if语句
    if (hasCorsConfigurationSource(handler) ||
CorsUtils.isPreFlightRequest(request)) {
        CorsConfiguration config = getCorsConfiguration(handler, request);
        if (getCorsConfigurationSource() != null) {
            CorsConfiguration globalConfig =
getCorsConfigurationSource().getCorsConfiguration(request);
            config = (globalConfig != null ? globalConfig.combine(config) :
config);
        }
        if (config != null) {
            config.validateAllowCredentials();
        }
        executionChain = getCorsHandlerExecutionChain(request, executionChain,
config);
    }
    return executionChain;
}
//基本等同于后面lookupHandlerMethod的方法，根据request以及@RequestMapping信息（即
RequestMappingInfo）获取最合适的处理方法
protected HandlerMethod getHandlerInternal(HttpServletRequest request) throws
Exception {
    //移除request中
org.springframework.web.servlet.HandlerMapping.producibleMediaTypes属性
    request.removeAttribute(PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE);
    try {
        //super对应的方法分析
        //protected HandlerMethod getHandlerInternal(HttpServletRequest request)
throws Exception {
        //  //protected String initLookupPath(HttpServletRequest request) {
        //  //    if (usesPathPatterns()) {//patternParser为null，走false中的逻辑
```

```
        //  //        request.removeAttribute(UrlPathHelper.PATH_ATTRIBUTE);
        //  //        RequestPath requestPath =
ServletRequestPathUtils.getParsedRequestPath(request);
        //  //        String lookupPath =
requestPath.pathWithinApplication().value();
        //  //        return
UrlPathHelper.defaultInstance.removeSemicolonContent(lookupPath);
        //  //    }
        //  //    else {//调试时执行这里的逻辑；UrlPathHelper：Helper class for URL
path matching
        //  //      //public String resolveAndCacheLookupPath(HttpServletRequest
request) {
        //  //      //  //调用层次比较多，只说过程
        //  //      //  //0.内部通过contextPath = getContextPath(request)获取上下文
路径（server.servlet.context-path可配置，默认""）
        //  //      //  //1.内部通过uri = request.getRequestURI()获取uri
        //  //      //  //2.uri = removeSemicolonContent(uri);移除url中所有;至/之间
的内容
        //  //      //  //  uri = decodeRequestString(request, uri);//尝试解码
        //  //      //  //  uri = getSanitizedPath(uri);//去除路径中的双//
        //  //      //  //3.path = getRemainingPath(requestUri, contextPath,
true); path实际为uri-contextPath部分
        //  //      //  //4.path要是全为空白则相当于"/"，lookupPath基本就是path
        //  //      //  String lookupPath = getLookupPathForRequest(request);
        //  //      //  //request中设置属性
org.springframework.web.util.UrlPathHelper.PATH
        //  //      //  request.setAttribute(PATH_ATTRIBUTE, lookupPath);
        //  //      //  return lookupPath;
        //  //      //}
        //  //      return
getUrlPathHelper().resolveAndCacheLookupPath(request);
        //  //  }
        //  //}//基本上等同于获取request中除去上下文路径后的路径
        //  String lookupPath = initLookupPath(request);
        //  this.mappingRegistry.acquireReadLock();//上读锁
        //  try {
        //      //Look up the best-matching handler method for the current
request. If multiple matches are found,
        //      //the best match is selected.
        //     //整个过程根据request以及@RequestMapping信息（即RequestMappingInfo）获
取最合适的处理方法，也即HandlerMethod,详细见后文分析
        //      HandlerMethod handlerMethod = lookupHandlerMethod(lookupPath,
request);
        //      //public HandlerMethod createWithResolvedBean() {
        //      //    Object handler = this.bean;
        //      //    if (this.bean instanceof String) {
        //      //        Assert.state(this.beanFactory != null, "Cannot resolve
bean name without BeanFactory");
        //      //        String beanName = (String) this.bean;
        //      //        handler = this.beanFactory.getBean(beanName);
        //      //    }
        //      //    return new HandlerMethod(this, handler);//依据已有的
handlerMethod复制一个新的HandlerMethod，但是附带上解析后的bean
        //      //}
        //      return (handlerMethod != null ?
handlerMethod.createWithResolvedBean() : null);
        //  }
        //  finally {
```

```
        //      this.mappingRegistry.releaseReadLock();//释放读锁
        //  }
        //}
        return super.getHandlerInternal(request);
    }
    finally {
        //移除request中
org.springframework.web.servlet.mvc.condition.ProducesRequestCondition.MEDIA_TYP
ES属性
        ProducesRequestCondition.clearMediaTypesAttribute(request);
    }
}
//根据request以及@RequestMapping信息（即RequestMappingInfo）获取最合适的处理方法
protected HandlerMethod lookupHandlerMethod(String lookupPath,
HttpServletRequest request) throws Exception{
    List<Match> matches = new ArrayList<>();
    //此处的T是RequestMappingInfo；getMappingsByDirectPath是从MultiValueMap中获取
lookupPath对应的List
    List<T> directPathMatches =
this.mappingRegistry.getMappingsByDirectPath(lookupPath);
    if (directPathMatches != null) {
        //private void addMatchingMappings(Collection<T> mappings, List<Match>
matches,
        //                                 HttpServletRequest request) {
        //    for (T mapping : mappings) {//遍历List<RequestMappingInfo>
        //      //Request mapping information. A composite（组合） for the the
following conditions:
        //      //1PathPatternsRequestCondition with parsed PathPatterns or
PatternsRequestCondition with
        //      //String patterns via PathMatcher匹配URL,可用||组合
        //      //2RequestMethodsRequestCondition匹配
GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS,TRACE，可用||组合
        //      //3ParamsRequestCondition:Supported at the type level as well as
at the method level!
        //      //when used at the type level, all method-level mappings inherit
this parameter restriction.
        //      //匹配参数，可用&&组合
        //      //4HeadersRequestCondition支持类型及方法层次，匹配headers，可用&&组合
        //      //5ConsumesRequestCondition匹配Content-Type，可用||组合
        //      //6ProducesRequestCondition匹配Accept，可用||组合
        //      //7RequestCondition (optional, custom request condition) 可自定义
条件
        //      //RequestMappingInfo实际就是封装了@RequestMapping注解的信息
        //      //此处getMatchingMapping底层利用RequestMappingInfo内部的条件信息判断
request是否匹配
        //      //a new instance(基于mapping new一个新的RequestMappingInfo)in case
of a match;
        //      //or null otherwise
        //      T match = getMatchingMapping(mapping, request);
        //      if (match != null) {//匹配的话，封装成Match放入matches
        //          //getRegistrations():Map<T, MappingRegistration<T>> T为
RequestMappingInfo
        //          //MappingRegistration目前调试来看是封装了RequestMappingInfo、
HandlerMethod等信息的一个对象
        //          //Match真就仅仅只有RequestMappingInfo和HandlerMethod两个信息，仅
仅是封装作用
        //          matches.add(new Match(match,
this.mappingRegistry.getRegistrations().get(mapping)));
```

```
        //      }
        //}
        addMatchingMappings(directPathMatches, matches, request);
    }
    if (matches.isEmpty()) {
        //通过lookupPath处理没找到合适的Match,那么尝试mappingRegistry所有已有的
RequestMappingInfo
        addMatchingMappings(this.mappingRegistry.getRegistrations().keySet(),
matches, request);
    }
    if (!matches.isEmpty()) {
        Match bestMatch = matches.get(0);//初始化bestMatch,默认为第一条
        if (matches.size() > 1) {
            //创建Match的比较器，这里每种条件都有自己的比较规则，比较多也不是很重要，暂不深入
分析
            Comparator<Match> comparator = new
MatchComparator(getMappingComparator(request));
            matches.sort(comparator);//排序
            bestMatch = matches.get(0);//设置为排序后的第一条
            if (logger.isTraceEnabled()) {
                logger.trace(matches.size() + " matching mappings: " + matches);
            }
            //借助Access-Control-Allow-Origin响应头可以允许跨域请求（比如ajax）
            //对于非简单请求（比如自定义了头部或者非简单方法如GET HEAD POST）会先进行一个
preflight（一个OPTIONS请求）
            //请求成功后才会发送真正的请求
            if (CorsUtils.isPreFlightRequest(request)) {
                for (Match match : matches) {
                    if (match.hasCorsConfig()) {//跨域资源共享Cross-orgin resource
sharing
                        return PREFLIGHT_AMBIGUOUS_MATCH;//里面hander会抛出
UnsupportedOperationException异常
                    }
                }
            }
            else {
                Match secondBestMatch = matches.get(1);
                //第一个和第二个优先级相等，则抛出异常
                if (comparator.compare(bestMatch, secondBestMatch) == 0) {
                    Method m1 = bestMatch.getHandlerMethod().getMethod();
                    Method m2 = secondBestMatch.getHandlerMethod().getMethod();
                    String uri = request.getRequestURI();
                    throw new IllegalStateException(
                        "Ambiguous handler methods mapped for '" + uri + "': {"
+ m1 + ", " + m2 + "}");
                }
            }
        }
        //设置org.springframework.web.servlet.HandlerMapping.bestMatchingHandler属
性为bestMatch.getHandlerMethod()
        request.setAttribute(BEST_MATCHING_HANDLER_ATTRIBUTE,
bestMatch.getHandlerMethod());
        //设置
org.springframework.web.servlet.HandlerMapping.pathWithinHandlerMapping属性为
lookupPath
        //设置org.springframework.web.servlet.HandlerMapping.bestMatchingPattern
```

```
        //设置
org.springframework.web.servlet.HandlerMapping.uriTemplateVariables,和模板变量相关
(@PathVariable使用)
        //设置org.springframework.web.servlet.HandlerMapping.producibleMediaTypes
和Accept相关
        //Expose URI template variables（/abc/{id} id就是模板变量）,
        //matrix variables（./123;q=2/.q就是martrix variable，@MatrixVariable使
用）,
        //and producible media types in the request
        handleMatch(bestMatch.mapping, lookupPath, request);
        return bestMatch.getHandlerMethod();//返回handlerMethod
    }
    else {//遍历所有的RequestMapInfo，分析mismatch原因，抛出异常；某些特殊场景可能返回
null，不过主要意义应该还是抛出异常
        return handleNoMatch(this.mappingRegistry.getRegistrations().keySet(),
lookupPath, request);
    }
}

//前文ha.handle(processedRequest, response, mappedHandler.getHandler())：
doDispatch->RequestMappingHandlerAdapter.handle-
>RequestMappingHandlerAdapter.handleInternal-
>RequestMappingHandlerAdapter.invokeHandlerMethod
protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws
Exception {
    //将request和response封装成一个ServletWebRequest
    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    try {
        //private WebDataBinderFactory getDataBinderFactory(HandlerMethod
handlerMethod) throws Exception {
        //    Class<?> handlerType = handlerMethod.getBeanType();//处理的请求的方法
的所在的Controller的Bean类型
        //    Set<Method> methods = this.initBinderCache.get(handlerType);
        //    if (methods == null) {//先从缓存获取Controller中定义@InitBinder方法，没
有缓存就解析创建并缓存
        //        methods = MethodIntrospector.selectMethods(handlerType,
INIT_BINDER_METHODS);
        //        this.initBinderCache.put(handlerType, methods);
        //    }
        //   List<InvocableHandlerMethod> initBinderMethods = new ArrayList<>();
        //    // Global methods first：先从@ControllerAdvice标注的组件中处理标注了
@InitBinder的方法
        //    this.initBinderAdviceCache.forEach((controllerAdviceBean,
methodSet) -> {
        //        if (controllerAdviceBean.isApplicableToBeanType(handlerType))
{
        //            Object bean = controllerAdviceBean.resolveBean();
        //            for (Method method : methodSet) {
        //                initBinderMethods.add(createInitBinderMethod(bean,
method));
        //            }
        //        }
        //    });
        //    for (Method method : methods) {//再处理求的方法的所在的Controller中
@InitBinder方法
        //        Object bean = handlerMethod.getBean();
        //        initBinderMethods.add(createInitBinderMethod(bean, method));
```

```
        //    }//创建ServletRequestDataBinderFactory工厂;
        //    //webDataBinder是用于表单到方法的数据绑定的，所谓的属性编辑器可以理解就是帮助
我们完成参数绑定
        //    return createDataBinderFactory(initBinderMethods);
        //}//一个简单的参考文章https://www.cnblogs.com/better-farther-
world2099/articles/10971897.html
        WebDataBinderFactory binderFactory =
getDataBinderFactory(handlerMethod);
        //private ModelFactory getModelFactory(HandlerMethod handlerMethod,
WebDataBinderFactory
        //                                     binderFactory) {
        //    //有则获取，无则创建，处理@SessionAttributes相关
        //    SessionAttributesHandler sessionAttrHandler =
getSessionAttributesHandler(handlerMethod);
        //    Class<?> handlerType = handlerMethod.getBeanType();
        //    Set<Method> methods = this.modelAttributeCache.get(handlerType);
        //    if (methods == null) {//先从缓存获取Controller中定义 @ModelAttribute方
法，没有缓存就解析创建并缓存
        //        methods = MethodIntrospector.selectMethods(handlerType,
MODEL_ATTRIBUTE_METHODS);
        //        this.modelAttributeCache.put(handlerType, methods);
        //    }
        //    List<InvocableHandlerMethod> attrMethods = new ArrayList<>();
        //    // Global methods first 先从@ControllerAdvice标注的组件中处理标注了
@ModelAttribute的方法
        //    this.modelAttributeAdviceCache.forEach((controllerAdviceBean,
methodSet) -> {
        //        if (controllerAdviceBean.isApplicableToBeanType(handlerType))
{
        //            Object bean = controllerAdviceBean.resolveBean();
        //            for (Method method : methodSet) {
        //
attrMethods.add(createModelAttributeMethod(binderFactory, bean, method));
        //            }
        //        }
        //    });
        //    for (Method method : methods) {//再处理求的方法的所在的Controller中
@ModelAttribute方法
        //        Object bean = handlerMethod.getBean();
        //        attrMethods.add(createModelAttributeMethod(binderFactory,
bean, method));
        //    }//Create a ModelFactory with the given @ModelAttribute methods
        //    return new ModelFactory(attrMethods, binderFactory,
sessionAttrHandler);
        //}
        ModelFactory modelFactory = getModelFactory(handlerMethod,
binderFactory);
        //将handlerMethod封装成ServletInvocableHandlerMethod
        ServletInvocableHandlerMethod invocableMethod =
createInvocableHandlerMethod(handlerMethod);
        //设置方法参数解析器，用于处理方法参数的绑定
        if (this.argumentResolvers != null) {

 invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
        }
        //设置方法返回值解析器，用于处理方法返回参数的处理
        if (this.returnValueHandlers != null) {
```

```
 invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
        }
        //设置数据绑定工厂
        invocableMethod.setDataBinderFactory(binderFactory);
        //设置方法参数发现组件

 invocableMethod.setParameterNameDiscoverer(this.parameterNameDiscoverer);
        //建一个容器，用于记录 HandlerMethodArgumentResolvers and
HandlerMethodReturnValueHandlers
        //处理后的Model和View，以及View相关的决策（比如@ResponseBody要求返回JSON，那么可
以在处理过程
        //中设置requestHandled标识为true，那么后续就不需要进行View解析）
        //A default Model is automatically created at instantiation. 在重定向场景可
以手动设置Model
        //An alternate model instance may be provided via setRedirectModel for
use in a redirect scenario.
        ModelAndViewContainer mavContainer = new ModelAndViewContainer();
        //FlashMap主要用于Redirect转发时的参数传递，我们只需要在redirect之前将需要传递的参
数写入
        //DispatcherServlet.OUTPUT_FLASH_MAP_ATTRIBUTE中,这样在redirect之后的
handler中Spring会自动的设置到Model中

 mavContainer.addAllAttributes(RequestContextUtils.getInputFlashMap(request));
        //public void initModel(NativeWebRequest request, ModelAndViewContainer
container, HandlerMethod
        //                      handlerMethod) throws Exception {
        //    //当前方法用于填充模型数据
        //    //1.Retrieve "known" session attributes listed as
@SessionAttributes. 填充模型
        //    Map<String, ?> sessionAttributes =
this.sessionAttributesHandler.retrieveAttributes(request);
        //    container.mergeAttributes(sessionAttributes);
        //    //2.Invoke @ModelAttribute methods 填充模型
        //    invokeModelAttributeMethods(request, container);
        //  //3.Find @ModelAttribute method arguments also listed as
@SessionAttributes and ensure
        //    //they're present in the model raising an exception if necessary.
        //    for (String name : findSessionAttributeArguments(handlerMethod)) {
        //        if (!container.containsAttribute(name)) {
        //            Object value =
this.sessionAttributesHandler.retrieveAttribute(request, name);
        //            if (value == null) {
        //                throw new HttpSessionRequiredException("Expected
session attribute '" +
        //                                                       name + "'",
name);
        //            }
        //            container.addAttribute(name, value);
        //        }
        //    }
        //}
        modelFactory.initModel(webRequest, mavContainer, invocableMethod);
        //设置ignoreDefaultModelOnRedirect标识

 mavContainer.setIgnoreDefaultModelOnRedirect(this.ignoreDefaultModelOnRedirect)
;
        //初始化一些参数，并设置一些回调处理拦截器
```

```
        AsyncWebRequest asyncWebRequest =
WebAsyncUtils.createAsyncWebRequest(request, response);
        asyncWebRequest.setTimeout(this.asyncRequestTimeout);
        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
        asyncManager.setTaskExecutor(this.taskExecutor);
        asyncManager.setAsyncWebRequest(asyncWebRequest);
        //希迈纳两个回调拦截器在调试时为0
        asyncManager.registerCallableInterceptors(this.callableInterceptors);

 asyncManager.registerDeferredResultInterceptors(this.deferredResultInterceptors
);
        if (asyncManager.hasConcurrentResult()) {//调试未执行
            Object result = asyncManager.getConcurrentResult();
            mavContainer = (ModelAndViewContainer)
asyncManager.getConcurrentResultContext()[0];
            asyncManager.clearConcurrentResult();
            LogFormatUtils.traceDebug(logger, traceOn -> {
                String formatted = LogFormatUtils.formatValue(result, !traceOn);
                return "Resume with async result [" + formatted + "]";
            });
            invocableMethod = invocableMethod.wrapConcurrentResult(result);
        }
        //Invoke the method and handle the return value through one of the
configured
        //HandlerMethodReturnValueHandlers. 见后文分析
        invocableMethod.invokeAndHandle(webRequest, mavContainer);
        if (asyncManager.isConcurrentHandlingStarted()) {
            return null;
        }
        //见后文分析
        return getModelAndView(mavContainer, modelFactory, webRequest);
    }
    finally {
        webRequest.requestCompleted();
    }
}
//前文中invocableMethod.invokeAndHandle(webRequest, mavContainer):
public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer
mavContainer,
                            Object... providedArgs) throws Exception {
    //public Object invokeForRequest(NativeWebRequest request, @Nullable
ModelAndViewContainer
    //                               mavContainer,Object... providedArgs) throws
Exception {
    //    //getMethodArgumentValues解析出方法的请求参数，具体细节暂不展开，简单说有如下几
方面
    //    //1.args[i] = findProvidedArgument(parameter, providedArgs) 通过
providedArgs来尝试获取参数
    //    //2.1失败的话，args[i] = this.resolvers.resolveArgument(parameter,
mavContainer, request,
    //    //this.dataBinderFactory)，通过HandlerMethodArgumentResolverComposite解
析参数，其中有过程:
    //    //private HandlerMethodArgumentResolver
getArgumentResolver(MethodParameter parameter) {
    //    //HandlerMethodArgumentResolver result =
this.argumentResolverCache.get(parameter);
    //    //  if (result == null) {
```

```
    //    //      //调试时这里有27个resolvers，从中选择第一个满足要求能处理的resolver，缓
存并返回
    //    //      for (HandlerMethodArgumentResolver resolver :
this.argumentResolvers) {
    //    //          if (resolver.supportsParameter(parameter)) {
    //    //              result = resolver;
    //    //              this.argumentResolverCache.put(parameter, result);
    //    //              break;
    //    //          }
    //    //      }
    //    //  }
    //    //  return result;
    //    //}
    //    Object[] args = getMethodArgumentValues(request, mavContainer,
providedArgs);
    //    if (logger.isTraceEnabled()) {
    //        logger.trace("Arguments: " + Arrays.toString(args));
    //    }
    //    //调用Controller对应的处理方法，底层调用method.invoke(getBean(), args)
    //    return doInvoke(args);
    //}
    Object returnValue = invokeForRequest(webRequest, mavContainer,
providedArgs);
    setResponseStatus(webRequest);//设置返回状态
    //下面if调试时都未执行
    if (returnValue == null) {
        if (isRequestNotModified(webRequest) || getResponseStatus() != null ||
            mavContainer.isRequestHandled()) {
            disableContentCachingIfNecessary(webRequest);
            mavContainer.setRequestHandled(true);
            return;
        }
    }
    else if (StringUtils.hasText(getResponseStatusReason())) {
        mavContainer.setRequestHandled(true);
        return;
    }
    //先初始化requestedHandled为false
    mavContainer.setRequestHandled(false);
    Assert.state(this.returnValueHandlers != null, "No return value handlers");
    try {
        //public void handleReturnValue(@Nullable Object returnValue,
MethodParameter returnType,
        //                              ModelAndViewContainer mavContainer,
NativeWebRequest webRequest) throws Exception {
        //    // private HandlerMethodReturnValueHandler selectHandler(@Nullable
Object value,
        //    //
MethodParameter returnType) {
        //    //     boolean isAsyncValue = isAsyncReturnValue(value,
returnType);
        //    //     //调试时有15个returnValueHandlers，从中选择第一个能支持返回类型处理
的hander
        //    //     for (HandlerMethodReturnValueHandler handler :
this.returnValueHandlers) {
        //    //         if (isAsyncValue && !(handler instanceof
AsyncHandlerMethodReturnValueHandler)) {
        //    //             continue;
```

```
        //    //        }
        //    //         if (handler.supportsReturnType(returnType)) {
        //    //             return handler;
        //    //         }
        //    //     }
        //    //     return null;
        //    // }
        //    HandlerMethodReturnValueHandler handler =
selectHandler(returnValue, returnType);
        //    if (handler == null) {
        //        throw new IllegalArgumentException("Unknown return value type:
" +
        //
returnType.getParameterType().getName());
        //    }
        //    //e.g.如果时@ResponseBody标注的处理方法，那么handler会是
RequestResponseBodyMethodProcessor
        //    //其中的一些关键代码：1.mavContainer.setRequestHandled(true); 说明后续不
需要进行View解析
        //    //2.this.contentNegotiationManager.resolveMediaTypes(new
ServletWebRequest(request))内容协商，
        //    //获取客户端可接受的acceptableTypes，ContentNegotiationManager：
Central class to determine
        //    //requested media types for a request. This is done by delegating
to a list of configured
        //    //ContentNegotiationStrategy instances.内部默认只有一个
HeaderContentNegotiationStrategy：checks
        //    //the 'Accept' request header
        //    //3.producibleTypes = getProducibleMediaTypes(request, valueType,
targetType) 获取服务器可用
        //    //MediaType，其先从request的
HandlerMapping.PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE属性中尝试获取
        //    //若失败则尝试从RequestResponseBodyMethodProcessor.messageConverters
中获取可以处理的converter及其
        //    //能处理的MediaType，合并所有结果返回
        //    //4.通过对比服务端和客户端的MediaType，获取可用的mediaTypesToUse
        //    //5.ediaType.sortBySpecificityAndQuality(mediaTypesToUse) 排序
mediaTypesToUse
        //    //6.底层使用objectWriter.writeValue(generator, value); 转换成JSON然后
写
        //    handler.handleReturnValue(returnValue, returnType, mavContainer,
webRequest);
        //}
        this.returnValueHandlers.handleReturnValue(
            returnValue, getReturnValueType(returnValue), mavContainer,
webRequest);
    }
    catch (Exception ex) {
        if (logger.isTraceEnabled()) {
            logger.trace(formatErrorForReturnValue(returnValue), ex);
        }
        throw ex;
    }
}

//前文getModelAndView:
private ModelAndView getModelAndView(ModelAndViewContainer mavContainer,
```

```
            ModelFactory modelFactory, NativeWebRequest webRequest) throws
Exception {
    //此处在普通请求调试时，相当于仅执行
this.sessionAttributesHandler.storeAttributes(request, defaultModel);保存model到
    //sessionAttributesHandler
    modelFactory.updateModel(webRequest, mavContainer);
    if (mavContainer.isRequestHandled()) {//如果requestHandled=true（比如
@ResponseBody产生的效果），直接返回不进行视图解析处理
        return null;
    }
    ModelMap model = mavContainer.getModel();//如果不是重定向，就返回默认模型
defaultModel；否则返回redirectModel
    //将之前已有的信息封装成ModelAndView
    ModelAndView mav = new ModelAndView(mavContainer.getViewName(), model,
mavContainer.getStatus());
    if (!mavContainer.isViewReference()) {//isViewReference():判断this.view
instanceof String
        mav.setView((View) mavContainer.getView());//在mav中设置View对象
    }
    if (model instanceof RedirectAttributes) {//重定向相关处理
        Map<String, ?> flashAttributes = ((RedirectAttributes)
model).getFlashAttributes();
        HttpServletRequest request =
webRequest.getNativeRequest(HttpServletRequest.class);
        if (request != null) {

 RequestContextUtils.getOutputFlashMap(request).putAll(flashAttributes);
        }
    }
    return mav;//返回ModelAndView
}
//对应前文processDispatchResult
private void processDispatchResult(HttpServletRequest request,
HttpServletResponse response,
            @Nullable HandlerExecutionChain mappedHandler, @Nullable
ModelAndView mv,
            @Nullable Exception exception) throws Exception {
    boolean errorView = false;
    if (exception != null) {//有异常，则尝试获取处理异常后的ModelAndView
        if (exception instanceof ModelAndViewDefiningException) {
            logger.debug("ModelAndViewDefiningException encountered",
exception);
            mv = ((ModelAndViewDefiningException) exception).getModelAndView();
        }
        else {
            Object handler = (mappedHandler != null ? mappedHandler.getHandler()
: null);
            mv = processHandlerException(request, response, handler, exception);
            errorView = (mv != null);
        }
    }
    // Did the handler return a view to render?
    if (mv != null && !mv.wasCleared()) {
        //protected void render(ModelAndView mv, HttpServletRequest request,
HttpServletResponse response)
        //      throws Exception {
        //    // Determine locale for request and apply it to the response.
        //    Locale locale =
```

```
        //            (this.localeResolver != null ?
this.localeResolver.resolveLocale(request) :
        //                    request.getLocale());
        //    response.setLocale(locale);//和本地化相关
        //    View view;
        //    String viewName = mv.getViewName();
        //    if (viewName != null) {//viewName存在则依据viewName并通过
viewResolvers解析出View对象；否则可直接使用mv的View对象
        //        // We need to resolve the view name.
        //        protected View resolveViewName(String viewName, @Nullable
Map<String, Object> model,
        //                                       Locale locale,
HttpServletRequest request) throws Exception {
        //            if (this.viewResolvers != null) {
        //                //调试时有4个ViewResolver,返回第一个解析出来的View
        //                for (ViewResolver viewResolver : this.viewResolvers) {
        //                    View view = viewResolver.resolveViewName(viewName,
locale);
        //                    if (view != null) {
        //                        return view;
        //                    }
        //                }
        //            }
        //            return null;
        //        }
        //        view = resolveViewName(viewName, mv.getModelInternal(),
locale, request);
        //        if (view == null) {
        //            throw new ServletException("Could not resolve view with
name '" + mv.getViewName() +
        //                    "' in servlet with name '" + getServletName() +
"'");
        //        }
        //    }
        //    else {
        //        // No need to lookup: the ModelAndView object contains the
actual View object.
        //        view = mv.getView();
        //        if (view == null) {
        //            throw new ServletException("ModelAndView [" + mv + "]
neither contains a view name nor a " +
        //                    "View object in servlet with name '" +
getServletName() + "'");
        //        }
        //    }
        //
        //    // Delegate to the View object for rendering.
        //    if (logger.isTraceEnabled()) {
        //        logger.trace("Rendering view [" + view + "] ");
        //    }
        //    try {
        //        if (mv.getStatus() != null) {
        //            response.setStatus(mv.getStatus().value());//设置Status
        //        }
        //        //使用view进行真正的渲染
        //        //Render the view given the specified model.
        //        //The first step will be preparing the request: In the JSP
case, this would mean setting model
```

```
        //        //objects as request attributes. The second step will be the
actual rendering of the view, for
        //        //example including the JSP via a RequestDispatcher.
        //        //个人理解渲染的实质就是依据模型数据，解析模板，生成一个html数据流，最后写
入response，响应给用户
        //        //viewResolver和view搭配实现自定义渲染
        //        view.render(mv.getModelInternal(), request, response);
        //    }
        //    catch (Exception ex) {
        //        if (logger.isDebugEnabled()) {
        //            logger.debug("Error rendering view [" + view + "]", ex);
        //        }
        //        throw ex;
        //    }
        //}
        render(mv, request, response);//Render the given ModelAndView.
        if (errorView) {
            WebUtils.clearErrorRequestAttributes(request);
        }
    }
    else {
        if (logger.isTraceEnabled()) {
            logger.trace("No view rendering, null ModelAndView returned.");
        }
    }
    if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
        // Concurrent handling started during a forward
        return;
    }
    if (mappedHandler != null) {
        // Exception (if any) is already handled..  处理完异常或正常执行完后触发
HandlerInterceptor.afterCompletion的执行
        mappedHandler.triggerAfterCompletion(request, response, null);
    }
}
```

父子容器

Springboot webstarter场景下，不存在父子容器；但在spring集成springMVC时可以通过配置实现父子容器效果，父容器无法访问子容器Bean，子容器可以访问父容器Bean，此处不进一步展开

```
//在第一次进行web请求时，dispatcherServlet执行如下方法初始化内部组件
public class DispatcherServlet extends FrameworkServlet
FrameworkServlet.initWebApplicationContext
protected WebApplicationContext initWebApplicationContext() {
    //getServletContext(): ApplicationContextFacade implements ServletContext，内
部有ApplicationContext
    //ApplicationContextFacade中的org.apache.catalina.core.ApplicationContext存在
    //org.springframework.web.context.WebApplicationContext.ROOT,其值为
    //AnnotationConfigServletWebServerApplicationContext，就是rootContext
    WebApplicationContext rootContext =

 WebApplicationContextUtils.getWebApplicationContext(getServletContext());
    WebApplicationContext wac = null;
```

```
    if (this.webApplicationContext != null) {
        // A context instance was injected at construction time -> use it
        wac = this.webApplicationContext;//这个也是
AnnotationConfigServletWebServerApplicationContext，同一个
        if (wac instanceof ConfigurableWebApplicationContext) {
            ConfigurableWebApplicationContext cwac =
(ConfigurableWebApplicationContext) wac;
            if (!cwac.isActive()) {//未进入此条件执行，即springboot没设置parent容器
                // The context has not yet been refreshed -> provide services
such as
                // setting the parent context, setting the application context
id, etc
                if (cwac.getParent() == null) {
                    // The context instance was injected without an explicit
parent -> set
                    // the root application context (if any; may be null) as the
parent
                    cwac.setParent(rootContext);
                }
                configureAndRefreshWebApplicationContext(cwac);
            }
        }
    }
    if (wac == null) {
        // No context instance was injected at construction time -> see if one
        // has been registered in the servlet context. If one exists, it is
assumed
        // that the parent context (if any) has already been set and that the
        // user has performed any initialization such as setting the context id
        wac = findWebApplicationContext();
    }
    if (wac == null) {
        // No context instance is defined for this servlet -> create a local one
        wac = createWebApplicationContext(rootContext);
    }

    if (!this.refreshEventReceived) {
        // Either the context is not a ConfigurableApplicationContext with
refresh
        // support or the context injected at construction time had already been
        // refreshed -> trigger initial onRefresh manually here.
        synchronized (this.onRefreshMonitor) {
            //protected void onRefresh(ApplicationContext context) {
            //    initStrategies(context);
            //}
            onRefresh(wac);
        }
    }
    if (this.publishContext) {
        // Publish the context as a servlet context attribute.
        String attrName = getServletContextAttributeName();
        getServletContext().setAttribute(attrName, wac);
    }
    return wac;
}
//在第一次进行web请求时，dispatcherServlet执行如下方法初始化内部组件，context为
AnnotationConfigServletWebServerApplicationContext
protected void initStrategies(ApplicationContext context) {
```

```
    initMultipartResolver(context);
    initLocaleResolver(context);
    initThemeResolver(context);
    //此处会先从当前context中查找，然后会从parent容器查找，但parent容器此处不存在
    //Map<String, HandlerMapping> matchingBeans =
    //  BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
HandlerMapping.class, true, false);
    initHandlerMappings(context);
    initHandlerAdapters(context);
    initHandlerExceptionResolvers(context);
    initRequestToViewNameTranslator(context);
    initViewResolvers(context);
    initFlashMapManager(context);
}
```