

专注Java后端技术

≤	2020年11月							≥
日	一	二	三	四	五	六		
1	2	3	4	5	6	7		
8	9	10	11	12	13	14		
15	16	17	18	19	20	21		
22	23	24	25	26	27	28		
29	30	1	2	3	4	5		
6	7	8	9	10	11	12		

 QQ交谈

昵称: 聂晨
园龄: 3年6个月
粉丝: 191
关注: 3
[+加关注](#)

随笔分类

- [docker\(1\)](#)
- [gradle\(6\)](#)
- [java基础\(9\)](#)
- [jsp/servlet/filter\(2\)](#)
- [kafka\(3\)](#)
- [kubernetes\(4\)](#)
- [Reactive\(2\)](#)
- [redis\(2\)](#)
- [solr\(5\)](#)
- [spring\(10\)](#)
- [SpringBoot\(14\)](#)
- [SpringCloud\(19\)](#)
- [SpringSecurity\(5\)](#)
- [zookeeper\(1\)](#)
- [项目实战\(2\)](#)

随笔档案

- [2020年8月\(1\)](#)
- [2020年6月\(1\)](#)
- [2019年10月\(1\)](#)
- [2019年6月\(1\)](#)
- [2019年5月\(2\)](#)
- [2019年3月\(1\)](#)
- [2019年2月\(2\)](#)
- [2019年1月\(2\)](#)
- [2018年12月\(5\)](#)
- [2018年7月\(3\)](#)
- [2018年6月\(4\)](#)
- [2018年5月\(6\)](#)
- [2018年4月\(11\)](#)
- [2018年3月\(9\)](#)
- [2018年1月\(6\)](#)
- [2017年12月\(6\)](#)
- [2017年11月\(2\)](#)
- [2017年10月\(5\)](#)
- [2017年9月\(2\)](#)

[新随笔](#) [新文章](#) [管理](#)

posts - 81,comments - 61,trackbacks - 0


深入理解SpringBoot之自动装配

SpringBoot的自动装配是拆箱即用的基础，也是微服务化的前提。其实它并不那么神秘，我在这之前已经写过最基本的实现了，大家可以参考[这篇文章](#)。这次主要的议题是，来看看它是怎么样实现的，我们透过源代码来把握自动装配的来龙去脉。

一、自动装配过程分析

1.1、关于@SpringBootApplication

我们在编写SpringBoot项目时，@SpringBootApplication是最常见的注解了，我们可以看一下源代码：



```
/*
 * Copyright 2012-2017 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.boot.autoconfigure;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.boot.SpringBootConfiguration;
import org.springframework.boot.context.TypeExcludeFilter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.ComponentScan.Filter;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.core.annotation.AliasFor;
```

[2017年8月\(1\)](#)[2017年7月\(4\)](#)[2017年6月\(1\)](#)[2017年5月\(5\)](#)

最新评论

[1. Re:SpringBoot+kafka+ELK分布式日志收集](#)

您好, 可以加个QQ联系一下吗

--二十二success

[2. Re:Webflux快速入门](#)

看看

--~hello🌞,ukyo🌞~

[3. Re:Gradle学习之闭包](#)

66

--颜王s

[4. Re:Webflux快速入门](#)

@Jx_ForeverYoung 一台服务器, 一秒钟请求3000次服务器崩溃了。但是增加吞吐后, 一秒接受请求5000次。...

--bbird2018

[5. Re:Webflux快速入门](#)

@hanxingruo 异步不仅仅是抖机灵。同步会阻塞, 然后依靠系统多线程来切换。而异步的话, 递交, 然后忙别的事情去了, 不占用这部分通信的资源, 等待处理完成再返回。会提升性能滴...

--bbird2018

```
/**
 * Indicates a {@link Configuration} class that declares one or more
 * {@link Bean} methods and also triggers {@link EnableAutoConfiguration}
 * auto-configuration and {@link ComponentScan} component scanning. This is a convenient
 * annotation that is equivalent to declaring {@code @Configuration},
 * {@code @EnableAutoConfiguration} and {@code @ComponentScan}.
 *
 * @author Phillip Webb
 * @author Stephane Nicoll
 * @since 1.2.0
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class)
})
public @interface SpringBootApplication {

    /**
     * Exclude specific auto-configuration classes such that they will never be applied.
     * @return the classes to exclude
     */
    @AliasFor(annotation = EnableAutoConfiguration.class, attribute = "exclude")
    Class<?>[] exclude() default {};

    /**
     * Exclude specific auto-configuration class names such that they will never be
     * applied.
     * @return the class names to exclude
     * @since 1.3.0
     */
    @AliasFor(annotation = EnableAutoConfiguration.class, attribute = "excludeName")
    String[] excludeName() default {};


    /**
     * Base packages to scan for annotated components. Use {@link #scanBasePackageClasses}
     * for a type-safe alternative to String-based package names.
     * @return base packages to scan
     * @since 1.3.0
     */
    @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
    String[] scanBasePackages() default {};

    /**
     * Type-safe alternative to {@link #scanBasePackages} for specifying the packages to
     * scan for annotated components. The package of each class specified will be scanned.
     * <p>
     * Consider creating a special no-op marker class or interface in each package that
     * serves no purpose other than being referenced by this attribute.
     * @return base packages to scan
     * @since 1.3.0
     */
    @AliasFor(annotation = ComponentScan.class, attribute = "basePackageClasses")
    Class<?>[] scanBasePackageClasses() default {};
}
```



这里面包含了@SpringBootConfiguration, @EnableAutoConfiguration, @ComponentScan, 此处@ComponentScan由于没有指定扫描包, 因此它默认扫描的是与该类同级的类或者同级包下的所有类, 另外@SpringBootConfiguration, 通过源码得知它是一个@Configuration:





```

/*
 * Copyright 2012-2016 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.boot;


import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.context.annotation.Configuration;

/**
 * Indicates that a class provides Spring Boot application
 * {@link Configuration @Configuration}. Can be used as an alternative to the Spring's
 * standard {@code @Configuration} annotation so that configuration can be found
 * automatically (for example in tests).
 *
 * <p>
 * Application should only ever include <em>one</em> {@code @SpringBootApplication} as
 * most idiomatic Spring Boot applications will inherit it from
 * {@code @SpringBootApplication}.
 *
 * @author Phillip Webb
 * @since 1.4.0
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootApplication {

}



```



由此我们可以推断出@SpringBootApplication等同于@Configuration @ComponentScan @EnableAutoConfiguration

1.2、@EnableAutoConfiguration

一旦加上此注解，那么将会开启自动装配功能，简单点讲，Spring会试图在你的classpath下找到所有配置的Bean然后进行装配。当然装配Bean时，会根据若干个(Conditional)定制规则来进行初始化。我们看一下它的源码：

```

/*
 * Copyright 2012-2017 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0

```

```

*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/

package org.springframework.boot.autoconfigure;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.boot.autoconfigure.condition.ConditionalOnBean;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.context.embedded.EmbeddedServletContainerFactory;
import org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletContainerFactory;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.core.io.support.SpringFactoriesLoader;

/**
 * Enable auto-configuration of the Spring Application Context, attempting to guess and
 * configure beans that you are likely to need. Auto-configuration classes are usually
 * applied based on your classpath and what beans you have defined. For example, If you
 * have tomcat-embedded.jar on your classpath you are likely to want a
 * TomcatEmbeddedServletContainerFactory (unless you have defined your own
 * EmbeddedServletContainerFactory bean).
 *
 * <p>
 * When using SpringBootApplication, the auto-configuration of the context is
 * automatically enabled and adding this annotation has therefore no additional effect.
 *
 * <p>
 * Auto-configuration tries to be as intelligent as possible and will back-away as you
 * define more of your own configuration. You can always manually #exclude() any
 * configuration that you never want to apply (use #excludeName() if you don't
 * have access to them). You can also exclude them via the
 * spring.autoconfigure.exclude property. Auto-configuration is always applied
 * after user-defined beans have been registered.
 *
 * <p>
 * The package of the class that is annotated with @EnableAutoConfiguration,
 * usually via @SpringBootApplication, has specific significance and is often used
 * as a 'default'. For example, it will be used when scanning for @Entity classes.
 * It is generally recommended that you place @EnableAutoConfiguration (if you're
 * not using @SpringBootApplication) in a root package so that all sub-packages
 * and classes can be searched.
 *
 * <p>
 * Auto-configuration classes are regular Spring Configuration beans. They are
 * located using the SpringFactoriesLoader mechanism (keyed against this class).
 * Generally auto-configuration beans are Conditional beans (most
 * often using ConditionalOnClass and
 * ConditionalOnMissingBean annotations).
 *
 * @author Phillip Webb
 * @author Stephane Nicoll
 * @see ConditionalOnBean
 * @see ConditionalOnMissingBean
 * @see ConditionalOnClass
 * @see AutoConfigureAfter
 * @see SpringBootApplication
 */
@SuppressWarnings("deprecation")
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented

```

```

@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    /**
     * Exclude specific auto-configuration classes such that they will never be applied
     * @return the classes to exclude
     */
    Class<?>[] exclude() default {};

    /**
     * Exclude specific auto-configuration class names such that they will never be
     * applied.
     * @return the class names to exclude
     * @since 1.3.0
     */
    String[] excludeName() default {};
}

```

虽然根据文档注释的说明它指点我们去看EnableAutoConfigurationImportSelector。但是该类在SpringBoot1.5.X版本已经过时了，因此我们看一下它的父类AutoConfigurationImportSelector：

```

/*
 * Copyright 2012-2017 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.boot.autoconfigure;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.TimeUnit;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.Aware;
import org.springframework.beans.factory.BeanClassLoaderAware;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.NoSuchBeanDefinitionException;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;

```

```

import org.springframework.boot.bind.RelaxedPropertyResolver;
import org.springframework.context.EnvironmentAware;
import org.springframework.context.ResourceLoaderAware;
import org.springframework.context.annotation.DeferredImportSelector;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.AnnotationAttributes;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.Environment;
import org.springframework.core.io.ResourceLoader;
import org.springframework.core.io.support.SpringFactoriesLoader;
import org.springframework.core.type.AnnotationMetadata;
import org.springframework.core.type.classreading.CachingMetadataReaderFactory;
import org.springframework.core.type.classreading.MetadataReaderFactory;
import org.springframework.util.Assert;
import org.springframework.util.ClassUtils;
import org.springframework.util.StringUtils;

/**
 * {@link DeferredImportSelector} to handle {@link EnableAutoConfiguration
 * auto-configuration}. This class can also be subclassed if a custom variant of
 * {@link EnableAutoConfiguration @EnableAutoConfiguration}. is needed.
 *
 * @author Phillip Webb
 * @author Andy Wilkinson
 * @author Stephane Nicoll
 * @author Madhura Bhawe
 * @since 1.3.0
 * @see EnableAutoConfiguration
 */
public class AutoConfigurationImportSelector
    implements DeferredImportSelector, BeanClassLoaderAware, ResourceLoaderAware,
        BeanFactoryAware, EnvironmentAware, Ordered {

    private static final String[] NO_IMPORTS = {};

    private static final Log logger = LoggerFactory
        .getLog(AutoConfigurationImportSelector.class);

    private ConfigurableListableBeanFactory beanFactory;

    private Environment environment;

    private ClassLoader beanClassLoader;

    private ResourceLoader resourceLoader;

    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        if (!isEnabled(annotationMetadata)) {
            return NO_IMPORTS;
        }
        try {
            AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadata
                .loadMetadata(this.beanClassLoader);
            AnnotationAttributes attributes = getAttributes(annotationMetadata);
            List<String> configurations = getCandidateConfigurations(annotationMetadata,
                attributes);
            configurations = removeDuplicates(configurations);
            configurations = sort(configurations, autoConfigurationMetadata);
            Set<String> exclusions = getExclusions(annotationMetadata, attributes);
            checkExcludedClasses(configurations, exclusions);
            configurations.removeAll(exclusions);
            configurations = filter(configurations, autoConfigurationMetadata);
            fireAutoConfigurationImportEvents(configurations, exclusions);
            return configurations.toArray(new String[configurations.size()]);
        }
        catch (IOException ex) {
            throw new IllegalStateException(ex);
        }
    }
}

```

```

protected boolean isEnabled(AnnotationMetadata metadata) {
    return true;
}

/**
 * Return the appropriate {@link AnnotationAttributes} from the
 * {@link AnnotationMetadata}. By default this method will return attributes for
 * {@link #getAnnotationClass()}.
 * @param metadata the annotation metadata
 * @return annotation attributes
 */
protected AnnotationAttributes getAttributes(AnnotationMetadata metadata) {
    String name = getAnnotationClass().getName();
    AnnotationAttributes attributes = AnnotationAttributes
        .fromMap(metadata.getAnnotationAttributes(name, true));
    Assert.notNull(attributes,
        "No auto-configuration attributes found. Is " + metadata.getClassName()
        + " annotated with " + ClassUtils.getShortName(name) + "?");
    return attributes;
}

/**
 * Return the source annotation class used by the selector.
 * @return the annotation class
 */
protected Class<?> getAnnotationClass() {
    return EnableAutoConfiguration.class;
}

/**
 * Return the auto-configuration class names that should be considered. By default
 * this method will load candidates using {@link SpringFactoriesLoader} with
 * {@link #getSpringFactoriesLoaderFactoryClass()}.
 * @param metadata the source metadata
 * @param attributes the {@link #getAttributes(AnnotationMetadata) annotation
 * attributes}
 * @return a list of candidate configurations
 */
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
    AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    Assert.notEmpty(configurations,
        "No auto configuration classes found in META-INF/spring.factories. If you
        are using a custom packaging, make sure that file is correct");
    return configurations;
}

/**
 * Return the class used by {@link SpringFactoriesLoader} to load configuration
 * candidates.
 * @return the factory class
 */
protected Class<?> getSpringFactoriesLoaderFactoryClass() {
    return EnableAutoConfiguration.class;
}

private void checkExcludedClasses(List<String> configurations,
    Set<String> exclusions) {
    List<String> invalidExcludes = new ArrayList<String>(exclusions.size());
    for (String exclusion : exclusions) {
        if (ClassUtils.isPresent(exclusion, getClass().getClassLoader())
            && !configurations.contains(exclusion)) {
            invalidExcludes.add(exclusion);
        }
    }
    if (!invalidExcludes.isEmpty()) {
        handleInvalidExcludes(invalidExcludes);
    }
}

```

```

    }

    /**
     * Handle any invalid excludes that have been specified.
     * @param invalidExcludes the list of invalid excludes (will always have at least one
     * element)
     */
    protected void handleInvalidExcludes(List<String> invalidExcludes) {
        StringBuilder message = new StringBuilder();
        for (String exclude : invalidExcludes) {
            message.append("\t- ").append(exclude).append(String.format("%n"));
        }
        throw new IllegalStateException(String
            .format("The following classes could not be excluded because they are"
                + " not auto-configuration classes:%n%s", message));
    }

    /**
     * Return any exclusions that limit the candidate configurations.
     * @param metadata the source metadata
     * @param attributes the {@link #getAttributes(AnnotationMetadata) annotation
     * attributes}
     * @return exclusions or an empty set
     */
    protected Set<String> getExclusions(AnnotationMetadata metadata,
        AnnotationAttributes attributes) {
        Set<String> excluded = new LinkedHashSet<String>();
        excluded.addAll(asList(attributes, "exclude"));
        excluded.addAll(Arrays.asList(attributes.getStringArray("excludeName")));
        excluded.addAll(getExcludeAutoConfigurationsProperty());
        return excluded;
    }

    private List<String> getExcludeAutoConfigurationsProperty() {
        if (getEnvironment() instanceof ConfigurableEnvironment) {
            RelaxedPropertyResolver resolver = new RelaxedPropertyResolver(
                this.environment, "spring.autoconfigure.");
            Map<String, Object> properties = resolver.getSubProperties("exclude");
            if (properties.isEmpty()) {
                return Collections.emptyList();
            }
            List<String> excludes = new ArrayList<String>();
            for (Map.Entry<String, Object> entry : properties.entrySet()) {
                String name = entry.getKey();
                Object value = entry.getValue();
                if (name.isEmpty() || name.startsWith("[") && value != null) {
                    excludes.addAll(new HashSet<String>(Arrays.asList(StringUtils
                        .tokenizeToStringArray(String.valueOf(value), ","))));
                }
            }
            return excludes;
        }
        RelaxedPropertyResolver resolver = new RelaxedPropertyResolver(getEnvironment(),
            "spring.autoconfigure.");
        String[] exclude = resolver.getProperty("exclude", String[].class);
        return (Arrays.asList(exclude == null ? new String[0] : exclude));
    }

    private List<String> sort(List<String> configurations,
        AutoConfigurationMetadata autoConfigurationMetadata) throws IOException {
        configurations = new AutoConfigurationSorter(getMetadataReaderFactory(),
            autoConfigurationMetadata).getInPriorityOrder(configurations);
        return configurations;
    }

    private List<String> filter(List<String> configurations,
        AutoConfigurationMetadata autoConfigurationMetadata) {
        long startTime = System.nanoTime();
        String[] candidates = configurations.toArray(new String[configurations.size()]);
        boolean[] skip = new boolean[candidates.length];
    }

```



```

        boolean skipped = false;
        for (AutoConfigurationImportFilter filter : getAutoConfigurationImportFilters())
            invokeAwareMethods(filter);
        boolean[] match = filter.match(candidates, autoConfigurationMetadata);
        for (int i = 0; i < match.length; i++) {
            if (!match[i]) {
                skip[i] = true;
                skipped = true;
            }
        }
    }
    if (!skipped) {
        return configurations;
    }
    List<String> result = new ArrayList<String>(candidates.length);
    for (int i = 0; i < candidates.length; i++) {
        if (!skip[i]) {
            result.add(candidates[i]);
        }
    }
    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 new ArrayList<String>(result);
}

protected List<AutoConfigurationImportFilter> getAutoConfigurationImportFilters() {
    return SpringFactoriesLoader.loadFactories(AutoConfigurationImportFilter.class,
        this.beanClassLoader);
}

private MetadataReaderFactory getMetadataReaderFactory() {
    try {
        return getBeanFactory().getBean(
            SharedMetadataReaderFactoryContextInitializer.BEAN_NAME,
            MetadataReaderFactory.class);
    }
    catch (NoSuchBeanDefinitionException ex) {
        return new CachingMetadataReaderFactory(this.resourceLoader);
    }
}

protected final <T> List<T> removeDuplicates(List<T> list) {
    return new ArrayList<T>(new LinkedHashSet<T>(list));
}

protected final List<String> asList(AnnotationAttributes attributes, String name) {
    String[] value = attributes.getStringArray(name);
    return Arrays.asList(value == null ? new String[0] : value);
}

private void fireAutoConfigurationImportEvents(List<String> configurations,
    Set<String> exclusions) {
    List<AutoConfigurationImportListener> listeners = getAutoConfigurationImportListeners();
    if (!listeners.isEmpty()) {
        AutoConfigurationImportEvent event = new AutoConfigurationImportEvent(this,
            configurations, exclusions);
        for (AutoConfigurationImportListener listener : listeners) {
            invokeAwareMethods(listener);
            listener.onAutoConfigurationImportEvent(event);
        }
    }
}

protected List<AutoConfigurationImportListener> getAutoConfigurationImportListeners() {
    return SpringFactoriesLoader.loadFactories(AutoConfigurationImportListener.class,
        this.beanClassLoader);
}

```

```

    }

    private void invokeAwareMethods(Object instance) {
        if (instance instanceof Aware) {
            if (instance instanceof BeanClassLoaderAware) {
                ((BeanClassLoaderAware) instance)
                    .setBeanClassLoader(this.beanClassLoader);
            }
            if (instance instanceof BeanFactoryAware) {
                ((BeanFactoryAware) instance).setBeanFactory(this.beanFactory);
            }
            if (instance instanceof EnvironmentAware) {
                ((EnvironmentAware) instance).setEnvironment(this.environment);
            }
            if (instance instanceof ResourceLoaderAware) {
                ((ResourceLoaderAware) instance).setResourceLoader(this.resourceLoader);
            }
        }
    }

    @Override
    public void setBeanFactory(BeansException {
        Assert.isInstanceOf(ConfigurableListableBeanFactory.class, beanFactory);
        this.beanFactory = (ConfigurableListableBeanFactory) beanFactory;
    }

    protected final ConfigurableListableBeanFactory getBeanFactory() {
        return this.beanFactory;
    }

    @Override
    public void setBeanClassLoader(ClassLoader classLoader) {
        this.beanClassLoader = classLoader;
    }

    protected ClassLoader getBeanClassLoader() {
        return this.beanClassLoader;
    }

    @Override
    public void setEnvironment(Environment environment) {
        this.environment = environment;
    }

    protected final Environment getEnvironment() {
        return this.environment;
    }

    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }

    protected final ResourceLoader getResourceLoader() {
        return this.resourceLoader;
    }

    @Override
    public int getOrder() {
        return Ordered.LOWEST_PRECEDENCE - 1;
    }
}

```

首先该类实现了DeferredImportSelector接口，这个接口继承了ImportSelector:



```


/*
 * Copyright 2002-2013 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.context.annotation;

import org.springframework.core.type.AnnotationMetadata;

/**
 * Interface to be implemented by types that determine which {@link Configuration}
 * class(es) should be imported based on a given selection criteria, usually one or more
 * annotation attributes.
 *
 * 

An {@link ImportSelector} may implement any of the following
 * {@link org.springframework.beans.factory.Aware Aware} interfaces, and their respective
 * methods will be called prior to {@link #selectImports}:


 * 


 * - {@link org.springframework.context.EnvironmentAware EnvironmentAware}

 * - {@link org.springframework.beans.factory.BeanFactoryAware BeanFactoryAware}

 * - {@link org.springframework.beans.factory.BeanClassLoaderAware BeanClassLoaderAware}

 * - {@link org.springframework.context.ResourceLoaderAware ResourceLoaderAware}


 *
 * 

ImportSelectors are usually processed in the same way as regular {@code @Import}
 * annotations, however, it is also possible to defer selection of imports until all
 * {@code @Configuration} classes have been processed (see {@link DeferredImportSelector}
 * for details).


 *
 * @author Chris Beams
 * @since 3.1
 * @see DeferredImportSelector
 * @see Import
 * @see ImportBeanDefinitionRegistrar
 * @see Configuration
 */
public interface ImportSelector {

    /**
     * Select and return the names of which class(es) should be imported based on
     * the {@link AnnotationMetadata} of the importing {@link Configuration} class.
     */
    String[] selectImports(AnnotationMetadata importingClassMetadata);


}


```

该接口主要是为了导入@Configuration的配置项，而DeferredImportSelector是延期导入，当所有的@Configuration都处理过后才会执行。

回过头来我们看一下AutoConfigurationImportSelector的selectImport方法：

```


@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {

```

```

        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadata
            .loadMetadata(this.getBeanClassLoader());
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        List<String> configurations = getCandidateConfigurations(annotationMetadata,
            attributes);
        configurations = removeDuplicates(configurations);
        configurations = sort(configurations, autoConfigurationMetadata);
        Set<String> exclusions = getExclusions(annotationMetadata, attributes);
        checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = filter(configurations, autoConfigurationMetadata);
        fireAutoConfigurationImportEvents(configurations, exclusions);
        return configurations.toArray(new String[configurations.size()]);
    }
    catch (IOException ex) {
        throw new IllegalStateException(ex);
    }
}

```

该方法刚开始会先判断是否进行自动装配，而后会从META-INF/spring-autoconfigure-metadata.properties读取元数据与元数据的相关属性，紧接着会调用getCandidateConfigurations方法：

```

/**
 * Return the auto-configuration class names that should be considered. By default
 * this method will load candidates using {@link SpringFactoriesLoader} with
 * {@link #getSpringFactoriesLoaderFactoryClass()}.
 * @param metadata the source metadata
 * @param attributes the {@link #getAttributes(AnnotationMetadata) annotation
 * attributes}
 * @return a list of candidate configurations
 */
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
    AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    Assert.notEmpty(configurations,
        "No auto configuration classes found in META-INF/spring.factories. If you
        are using a custom packaging, make sure that file is correct");
    return configurations;
}

/**
 * Return the class used by {@link SpringFactoriesLoader} to load configuration
 * candidates.
 * @return the factory class
 */
protected Class<?> getSpringFactoriesLoaderFactoryClass() {
    return EnableAutoConfiguration.class;
}

```

在这里又遇到我们的老熟人了--SpringFactoriesLoader，它会读取META-INF/spring.factories下的EnableAutoConfiguration的配置，紧接着在进行排除与过滤，进而得到需要装配的类。最后让所有配置在META-INF/spring.factories下的AutoConfigurationImportListener执行AutoConfigurationImportEvent事件，代码如下：

```

private void fireAutoConfigurationImportEvents(List<String> configurations,
    Set<String> exclusions) {
    List<AutoConfigurationImportListener> listeners = getAutoConfigurationImportListeners();
    if (!listeners.isEmpty()) {

```

```

        AutoConfigurationImportEvent event = new AutoConfigurationImportEvent(this,
            configurations, exclusions);
        for (AutoConfigurationImportListener listener : listeners) {
            invokeAwareMethods(listener);
            listener.onAutoConfigurationImportEvent(event);
        }
    }

    protected List<AutoConfigurationImportListener> getAutoConfigurationImportListeners() {
        return SpringFactoriesLoader.loadFactories(AutoConfigurationImportListener.class,
            this.beanClassLoader);
    }
}

```

二、何时进行自动装配

在前面的环节里只是最终要确定哪些类需要被装配，在SpringBoot时何时处理这些自动装配的类呢？下面我们简要的分析一下：

2.1、AbstractApplicationContext的refresh方法：

这个方法老生常谈了其中请大家关注一下这个方法：

```

// Invoke factory processors registered as beans in the context.
invokeBeanFactoryPostProcessors(beanFactory);

```

在这里是处理BeanFactoryPostProcessor的，那么我们在来看一下这个接口BeanDefinitionRegistryPostProcessor:

```

/*
 * Copyright 2002-2010 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.beans.factory.support;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;

/**
 * Extension to the standard {@link BeanFactoryPostProcessor} SPI, allowing for
 * the registration of further bean definitions <i>before</i> regular
 * BeanFactoryPostProcessor detection kicks in. In particular,
 * BeanDefinitionRegistryPostProcessor may register further bean definitions
 * which in turn define BeanFactoryPostProcessor instances.
 *
 * @author Juergen Hoeller
 * @since 3.0.1
 * @see org.springframework.context.annotation.ConfigurationClassPostProcessor
 */
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,
 * but no beans will have been instantiated yet. This allows for adding further
 * bean definitions before the next post-processing phase kicks in.
 * @param registry the bean definition registry used by the application context
 * @throws org.springframework.beans.BeansException in case of errors
 */
void postProcessBeanDefinitionRegistry(BeansException registry) throws BeansException
}

```

该接口继承了BeanFactoryPostProcessor。

2.2、ConfigurationClassPostProcessor 类

该类主要处理@Configuration注解的，它实现了BeanDefinitionRegistryPostProcessor，那么也间接实现了BeanFactoryPostProcessor，关键代码如下：

```

@Override
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
    int factoryId = System.identityHashCode(beanFactory);
    if (this.factoriesPostProcessed.contains(factoryId)) {
        throw new IllegalStateException(
            "postProcessBeanFactory already called on this post-processor again"
        );
    }
    this.factoriesPostProcessed.add(factoryId);
    if (!this.registriesPostProcessed.contains(factoryId)) {
        // BeanDefinitionRegistryPostProcessor hook apparently not supported...
        // Simply call processConfigurationClasses lazily at this point then.
        processConfigBeanDefinitions((BeanDefinitionRegistry) beanFactory);
    }

    enhanceConfigurationClasses(beanFactory);
    beanFactory.addBeanPostProcessor(new ImportAwareBeanPostProcessor(beanFactory));
}

/**
 * Build and validate a configuration model based on the registry of
 * {@link Configuration} classes.
 */
public void processConfigBeanDefinitions(BeansException registry) {
    //.....省略部分代码

    // Parse each @Configuration class
    ConfigurationClassParser parser = new ConfigurationClassParser(
        this.metadataReaderFactory, this.problemReporter, this.environment,
        this.resourceLoader, this.componentScanBeanNameGenerator, registry);

    Set<BeanDefinitionHolder> candidates = new LinkedHashSet<BeanDefinitionHolder>();
    Set<ConfigurationClass> alreadyParsed = new HashSet<ConfigurationClass>();
    do {
        parser.parse(candidates);
        parser.validate();

        Set<ConfigurationClass> configClasses = new LinkedHashSet<ConfigurationClass>();
        configClasses.removeAll(alreadyParsed);

        // Read the model and create bean definitions based on its content
        if (this.reader == null) {
            this.reader = new ConfigurationClassBeanDefinitionReader(
                registry, this.sourceExtractor, this.resourceLoader, this.environment,
                this.importBeanNameGenerator, parser.getImportRegistry());
        }
    }
}

```

```

this.reader.loadBeanDefinitions(configClasses);
alreadyParsed.addAll(configClasses);

candidates.clear();
if (registry.getBeanDefinitionCount() > candidateNames.length) {
    String[] newCandidateNames = registry.getBeanDefinitionNames();
    Set<String> oldCandidateNames = new HashSet<String>(Arrays.asList(candidateNames));
    Set<String> alreadyParsedClasses = new HashSet<String>();
    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, alreadyParsedClasses)) {
                candidates.add(new BeanDefinitionHolder(bd, candidateName));
            }
        }
    }
    candidateNames = newCandidateNames;
}
while (!candidates.isEmpty());
// ....省略部分代码
}

```

其实这里注释已经很清楚了，我们可以清楚的看到解析每一个@ConfigurationClass的关键类是：ConfigurationClassParser，那么我们继续看一看这个类的parse方法：

```

public void parse(Set<BeanDefinitionHolder> configCandidates) {
    this.deferredImportSelectors = new LinkedList<DeferredImportSelectorHolder>();

    for (BeanDefinitionHolder holder : configCandidates) {
        BeanDefinition bd = holder.getBeanDefinition();
        try {
            if (bd instanceof AnnotatedBeanDefinition) {
                parse(((AnnotatedBeanDefinition) bd).getMetadata(), holder.getBeanName());
            }
            else if (bd instanceof AbstractBeanDefinition && ((AbstractBeanDefinition) bd).isLazyInit()) {
                parse(((AbstractBeanDefinition) bd).getBeanClass(), holder.getBeanName());
            }
            else {
                parse(bd.getBeanClassName(), holder.getBeanName());
            }
        }
        catch (BeanDefinitionStoreException ex) {
            throw ex;
        }
        catch (Throwable ex) {
            throw new BeanDefinitionStoreException(
                "Failed to parse configuration class [" + bd.getBeanClassName() + "]", ex);
        }
    }

    processDeferredImportSelectors();
}

```

在这里大家留意一下最后一句processDeferredImportSelectors方法，在这里将会对DeferredImportSelector进行处理，这样我们就和AutoConfigurationSelector结合到一起了：



```
private void processDeferredImportSelectors() {
    List<DeferredImportSelectorHolder> deferredImports = this.deferredImportSelectors;
    this.deferredImportSelectors = null;
    Collections.sort(deferredImports, DEFERRED_IMPORT_COMPARATOR);

    for (DeferredImportSelectorHolder deferredImport : deferredImports) {
        ConfigurationClass configClass = deferredImport.getConfigurationClass();
        try {
            String[] imports = deferredImport.getImportSelector().selectImports(configClass.getMetadata());
            processImports(configClass, asSourceClass(configClass), asSourceClasses(imports));
        } catch (BeanDefinitionStoreException ex) {
            throw ex;
        } catch (Throwable ex) {
            throw new BeanDefinitionStoreException(
                "Failed to process import candidates for configuration class [" +
                configClass.getMetadata().getClassName() + "]", ex);
        }
    }
}
```

请大家关注这句代码：String[] imports = deferredImport.getImportSelector().selectImports(configClass.getMetadata());在这里deferredImport的类型为DeferredImportSelectorHolder:

```
private static class DeferredImportSelectorHolder {

    private final ConfigurationClass configurationClass;

    private final DeferredImportSelector importSelector;

    public DeferredImportSelectorHolder(ConfigurationClass configClass, DeferredImportSelector selector) {
        this.configurationClass = configClass;
        this.importSelector = selector;
    }

    public ConfigurationClass getConfigurationClass() {
        return this.configurationClass;
    }

    public DeferredImportSelector getImportSelector() {
        return this.importSelector;
    }
}
```

在这个内部类里持有了一个DeferredImportSelector的引用，至此将会执行自动装配的所有操作

三、总结

- 1) 自动装配还是利用了SpringFactoriesLoader来加载META-INF/spring.factories文件里所有配置的EnableAutoConfiguration，它会经过exclude和filter等操作，最终确定要装配的类
- 2) 处理@Configuration的核心还是ConfigurationClassPostProcessor，这个类实现了BeanFactoryPostProcessor，因此当AbstractApplicationContext执行refresh方法里的invokeBeanFactoryPostProcessors(beanFactory)方法时会执行自动装配

分类: [SpringBoot](#)

标签: [SpringBoot](#) [自动装配](#)

好文要顶

关注我

收藏该文





聂晨

关注 - 3

粉丝 - 191

+加关注

11

0

« 上一篇: [深入理解SpringAOP之代理对象](#)

» 下一篇: [深入理解SpringBoot之装配条件](#)

posted on 2018-05-12 17:36 [聂晨](#) 阅读(29196) 评论(1) [编辑](#) [收藏](#)

FeedBack:

#1楼

2020-03-13 09:51 | [混沌初开](#)

我看源码的时候也能找到这段代码，但是理解不了，一个看不懂英文，另一个可能水平还不够。你是怎么看懂的？

支持(0)

反对(0)

[回复](#)





[引用](#)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

发表评论 [【福利】注册AWS账号，立享12个月免费套餐](#)

编辑

预览

B    

支持 Markdown

提交评论

[退出](#) [订阅评论](#) [我的博客](#)

[Ctrl+Enter快捷键提交]

[首页](#) [新闻](#) [博问](#) [专区](#) [闪存](#) [班级](#)

相关博文:

- [SpringBoot](#)
 - [SpringBoot](#)
 - [SpringBoot](#)
 - [Springboot](#)
 - [【SpringBoot】SpringBoot国际化 \(七\)](#)
- » [更多推荐...](#)