springboot 自动装配原理详解

1)传统ssm整合redis的时候需要在xml的配置文件中进行大量的配置Bean

我们在这里使用springboot来代替ssm的整合,只是通过xml的形式来整合redis

第一步:加入配置

第二步: 配置xml的bean的配置

```
//配置连接池
<bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">
  cproperty name="minIdle" value="10">
  cproperty name="maxTotal" value="20"></property>
</bean>
//配置连接工厂
<br/><bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">
  cproperty name="hostName" value="47.104.128.12">
  cproperty name="password" value="123456">
  cproperty name="database" value="0"></property>
  cproperty name="poolConfig" ref="poolConfig">
</bean>
//配置 redisTemplate 模版类
<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
  connectionFactory" ref="jedisConnectionFactory"/>
  <!--如果不配置Serializer,那么存储的时候缺省使用String,如果用User类型存储,那么会提示错误User can't cast to String!
  cproperty name="keySerializer">
    <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>
  cproperty name="valueSerializer">
    <bean class="org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer"/>
  </property>
  cproperty name="hashKeySerializer">
    <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>
  </property>
  cproperty name="hashValueSerializer">
   <bean class="org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer"/>
  </property>
</bean>
```

第三步:导入配置

```
@SpringBootApplication
@ImportResource(locations = "classpath:beans.xml")
@RestController
public class TulingOpenAutoconfigPrincipleApplication {

          @Autowired
          private RedisTemplate redisTemplate;

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

          @RequestMapping("/testRedis")
          public String testRedis() {
                redisTemplate.opsForValue().set("smlz","smlz");
                return "OK";
          }
}
```

2)综上所述 我们发现,若整合redis**的时候通过传统的整合,进行了大量的配置,那么我们来看下通过springboot自动装** 配整合的对比

导入依赖:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

修改yml配置文件

```
spring.redis.host=47.104.128.12
spring.redis.port=6379
spring.redis.password=123456
```

直接使用(下述代码可以不要配置,为了解决保存使用jdk的序列方式才配置的)

```
@Bean
public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory) {
    RedisTemplate<Object, Object> template = new RedisTemplate<>();
    template.setDefaultSerializer(new Jackson2JsonRedisSerializer<Object>(Object.class));
    template.setConnectionFactory(redisConnectionFactory);
    return template;
}
```

- 3) 传统整合和springboot自动装配 优劣势分析。。。。。。。。。。。。
- 4) 自动装配原理前的不得不说的几个注解
- 4.1)通过@Import注解来导入ImportSelector组件

①:写一个配置类在配置类上标注一个@Import的注解,

```
@Configuration
@Import(value = {TulingSelector.class})
public class TulingConfig {
}
```

②: 在@Import注解的value值 写自己需要导入的组件

在selectImports方法中 就是你需要导入组件的全类名

```
public class TulingSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        return new String[]{"com.tuling.service.TulingServiceImpl"};
    }
}
```

核心代码:

```
@RestController
public class TulingController {
  //自动注入 tulingServiceImpl
  @Autowired
  private TulingServiceImpl tulingServiceImpl;
  @RequestMapping("testTuling")
  public String testTuling() {
     tuling Service Impl. test Service ();\\
     return "tulingOk";
  }
}
这里是没有标注其他注解提供给spring包扫描的
public class TulingServiceImpl {
  public void testService() {
     System.out.println("我是通过importSelector导入进来的service");
  }
}
```

```
▼ 📄 java
           ▼ 🛅 com
                                                        6
                                                                public class TulingController {
              ▼ 🛅 tuling
                 ▼ 🛅 config
                        C TulingConfig
                                                                  private TulingServiceImpl tulingServiceImpl
                        © 🖫 TulingSelector

▼ Image: controller
                      © % TulingController
                                                                     blic String testTuling() {
  tulingServiceImpl.testService();

▼ lim service

                        © • TulingServiceImpl
                    💰 🖟 TulingvipSpringannoApplicati
        ▶ ■ resources
     ▶ itest
   target
TulingvipSpringannoApplication 2019=03=20 15:16:26. 520 1NRU 13984 --
                                                                o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped UKL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.re
      2019-03-20 15:16:26.645 INFO 13984 --- [
                                                         main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.web.servlet.context.AnnotationConfigServ
                                                         main] s.w.s.m.m.a.RequestMappingiandlerMapping: Mapped "{[/testTuling]}" onto public java.lang.String com.tuling.controller.TulingController.tes main] s.w.s.m.m.a.RequestMappingiandlerMapping: Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity(java.util.Map(java.lang.t
     2019-03-20 15:16:26.676 INFO 13984 ---
2019-03-20 15:16:26.679 INFO 13984 ---
      2019-03-20 15:16:26.680 INFO 13984 ---
                                                         main] s.w.s.m.m.a.RequestMappingHandlerMapping: Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servlet.ModelAndViet
     2019-03-20 15:16:26.693 INFO 13984 ---
                                                         main] o.s. w.s. handler. SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org. springframework. web. servlet.resource
                                                         main] o. s. w. s. handler. SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org. springframework. web. servlet.resource. Resour
     2019-03-20 15:16:26.693 INFO 13984 ---
      2019-03-20 15:16:26.766 INFO 13984 ---
                                                         main] o.s.j.e.a.AnnotationMBeanExporter
                                                                                                       : Registering beans for JMX exposure on startup
                                                         main] o. s. b. w. embedded. tomcat. TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path
      2019-03-20 15:16:26.791 INFO 13984 --- [
      2019-03-20 15:16:26.793 INFO 13984 ---
                                                         main] c.tuling.TulingvipSpringannoApplication : Started TulingvipSpringannoApplication in 1.559 seconds (JVM running for 1.89)
      2019-03-20 15:16:46.974 INFO 13984 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
                                                                                                     : Initializing Spring FrameworkServlet 'dispatcherServlet'
      erminal Spring 0: Messages 4: Run 5: Debug 6: TODO
```

1.2) 通过@Import导入ImportBeanDefinitionRegistrar 从而进来导入组件

```
configuration
@Import(value = {TulingSelector.class, TulingImportBeanDefinitionRegistrar.class})
public class TulingConfig {
}
```

核心代码:

```
public class TulingImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
  @Override
  public void registerBeanDefinitions(AnnotationMetadata annotationMetadata, BeanDefinitionRegistry beanDefinitionReg
     //定义一个BeanDefinition
     RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(TulingDao.class);
     //把自定义的bean定义导入到容器中
     be an Definition Registry. register Be an Definition ("tuling Dao", root Be an Definition);\\
  }
}
通过ImportSelector功能导入进来的
public class TulingServiceImpl {
  @Autowired
  private TulingDao tulingDao;
  public void testService() {
     tulingDao.testTulingDao();
     System.out.println("我是通过importSelector导入进来的service");
  }
}
通过ImportBeanDefinitionRegistar导入进来的
public class TulingDao {
  public void testTulingDao() {
```

```
System.out.println("我是通过ImportBeanDefinitionRegistrar导入进来tulingDao组件");
}
```

测试结果:

```
pom.xml
 Run 🍕 TulingvipSpringannoApplication
                     2019-03-20 lb:35:24.741 INWO 13192 --- [ost-startStop-1] o.s.b.w.servlet.FilterKegistrationBean : Mapping filter: requestContextWilter to: [/*]
                  2019-03-20 15:35:24.814 INFO 13192 --- [ main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [cla:
 2019-03-20 15:35:24.934 INFO 13192 --- [
                                                                                                                     main] s. w. s. m. m. a. RequestMappingHandlerAdapter: Looking for @ControllerAdvice: org. springframework. boot. web.
 2019-03-20 15:35:24.966 INFO 13192 --- [ main] s.w.s.m.m.a.RequestMappingflandlerMapping: Mapped "[[/testTuling]]" onto public java.lang.String com.tu main] s.w.s.m.m.a.RequestMappingflandlerMapping: Mapped "[[/error]]" onto public org.springframework.http.Re:
 2019-03-20 15:35:24.969 INFO 13192 --- [
                                                                                                                     main] s.w.s.m.m.a.RequestMappingHandlerMapping: Mapped "{[/error], produces=[text/html]}" onto public org.sp:
                                                                                                                 main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/≠≠] onto handler of type [class o:
 1 = 2019-03-20 15:35:24.983 INFO 13192 --- [
         2019-03-20 15:35:24.983 INFO 13192 --- [
                                                                                                              main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.sprin;
                  180
 X
                   2019-03-20 15:35:31.497 INFO 13192 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'
                    2019-03-20 15:35:31.497 INFO 13192 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started to the control of the control of
                     我是通过ImportBeanDefinitionRegistrar导入进来tulingDao组件
                    我是诵讨importSelector导入讲来的service
■ Terminal  Spring  © 0: Messages  4: Run  5: Debug  6: TODO
```

1.3)spring底层条件装配的原理@Conditional

应用要求:比如我有二个组件,一个是TulingLog 一个是TulingAspect

而TulingLog 是依赖TulingAspect的 只有容器中有TulingAspect组件才会加载TulingLog

```
tulingLog组件 依赖TulingAspect组件
public class TulingLog {
}

tulingAspect组件
public class TulingAspect {
}
```

①:自定义条件组件条件

```
public class TulingConditional implements Condition {
  public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata) {
    //容器中包含tulingAspect组件才返回Ture
    if(conditionContext.getBeanFactory().containsBean("tulingAspect")){
      return true;
    }else{
      return false;
    }
 }
}
              -----该情况下会加载二个组件--------
  @Bean
  public TulingAspect tulingAspect() {
    System.out.println("TulingAspect组件自动装配到容器中");
    return new TulingAspect();
  }
```

```
@Bean
@Conditional(value = TulingConditional.class)
public TulingLog tulingLog() {
  System.out.println("TulingLog组件自动装配到容器中");
  return new TulingLog();
}
                 /*@Bean**/
public TulingAspect tulingAspect() {
  System.out.println("TulingAspect组件自动装配到容器中");
  return new TulingAspect();
}
@Bean
@Conditional(value = TulingConditional.class)
public TulingLog tulingLog() {
  System.out.println("TulingLog组件自动装配到容器中");
  return new TulingLog();
}
```

自动装配原理分析 从@SpringbootApplication入手分析

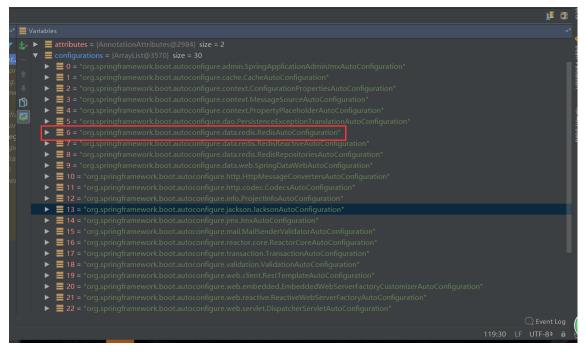


那我们仔细分析

org. spring framework. boot. autoconfigure. Auto Configuration Import Selector # select Imports and the property of the prop

```
public class AutoConfigurationImportSelector
                                   implements\ Deferred Import Selector,\ Bean Class Loader Aware,\ Resource Loader Aware,
                                   BeanFactoryAware, EnvironmentAware, Ordered {
                  @Override
                  public String[] selectImports(AnnotationMetadata annotationMetadata) {
                                   if (!isEnabled(annotationMetadata)) {
                                                     return NO_IMPORTS;
                                  }
                                   Auto Configuration Metadata\ auto Configuration Metadata\ =\ Auto Configuration Metadata\ Loader
                                                                      .loadMetadata(this.beanClassLoader);
                                   AnnotationAttributes attributes = getAttributes(annotationMetadata);
                                   //去mata-info/spring.factories文件中 查询 EnableAutoConfiguration对于值
                                   List < String > configurations = get Candidate Configurations (annotation Metadata, and the configuration of the configuration) and the configuration of t
                                                                      attributes);
                                   //去除重复的配置类,若我们自己写的starter 可能存主重复的
                                   configurations = removeDuplicates(configurations);
```

```
Set<String> exclusions = getExclusions(annotationMetadata, attributes);
          checkExcludedClasses(configurations, exclusions);
          configurations.removeAll(exclusions);
          //根据maven 导入的启动器过滤出 需要导入的配置类
          configurations = filter(configurations, autoConfigurationMetadata);
          fireAutoConfigurationImportEvents(configurations, exclusions);
          return\ StringUtils. to StringArray (configurations);
     }
}
//去spring.factories 中去查询EnableAutoConfirution类
private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader) {
          MultiValueMap<String, String> result = cache.get(classLoader);
          if (result != null) {
               return result;
          }
          try {
               Enumeration < URL > urls = (classLoader != null?
                         classLoader.getResources(FACTORIES_RESOURCE_LOCATION):
                         ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
               result = new LinkedMultiValueMap<>();
               while (urls.hasMoreElements()) {
                    URL url = urls.nextElement();
                    UrlResource resource = new UrlResource(url);
                    Properties properties = PropertiesLoaderUtils.loadProperties(resource);
                    for (Map.Entry<?, ?> entry : properties.entrySet()) {
                         List<String> factoryClassNames = Arrays.asList(
                                   StringUtils.commaDelimitedListToStringArray((String) entry.getValue()));
                         result.addAll((String) entry.getKey(), factoryClassNames);
                    }
               }
               cache.put(classLoader, result);
               return result;
          }
          catch (IOException ex) {
               throw new IllegalArgumentException("Unable to load factories from location [" +
                         FACTORIES_RESOURCE_LOCATION + "]", ex);
          }
     }
```



导入了三个组件 RedisTemplate StringRedisTemplate

JedisConnectionConfiguration

```
@Configuration
@ConditionalOnClass(RedisOperations.class)
@Enable Configuration Properties (Redis Properties. class)\\
@Import({ LettuceConnectionConfiguration.class, JedisConnectionConfiguration.class })
public class RedisAutoConfiguration {
   //导入redisTemplate
       @Bean
       @ConditionalOnMissingBean(name = "redisTemplate")
       public RedisTemplate < Object, Object > redisTemplate(
                     RedisConnectionFactory redisConnectionFactory) throws UnknownHostException {
              RedisTemplate<Object, Object> template = new RedisTemplate<>();
              template.setConnectionFactory(redisConnectionFactory);
              return template;
       }
       @Bean
       @ConditionalOnMissingBean
       public StringRedisTemplate stringRedisTemplate(
                     RedisConnectionFactory redisConnectionFactory) throws UnknownHostException {
              StringRedisTemplate template = new StringRedisTemplate();
              template.set Connection Factory (red is Connection Factory);\\
              return template;
      }
}
@ConditionalOnClass({ GenericObjectPool.class, JedisConnection.class, Jedis.class })
class JedisConnectionConfiguration extends RedisConnectionConfiguration {
       private final RedisProperties properties;
       private final List<JedisClientConfigurationBuilderCustomizer> builderCustomizers;
      JedisConnectionConfiguration(RedisProperties properties,
                     ObjectProvider<RedisSentinelConfiguration> sentinelConfiguration,
                     Object Provider < RedisCluster Configuration > cluster Configuration,\\
                     Object Provider < List < Jedis Client Configuration Builder Customizer >> builder Customizers) \ \{ configuration Builder Customizer <> builder <> builder Customizer <> builder <> builder <> builder <> builder << builder << builder << builder << builder <</br>
              super(properties, sentinelConfiguration, clusterConfiguration);
              this.properties = properties;
              this.builderCustomizers = builderCustomizers
                            .getIfAvailable(Collections::emptyList);
       }
       @ConditionalOnMissingBean(RedisConnectionFactory.class)
       public JedisConnectionFactory redisConnectionFactory() throws UnknownHostException {
              return createJedisConnectionFactory();
       }
       private JedisConnectionFactory createJedisConnectionFactory() {
             JedisClientConfiguration clientConfiguration = getJedisClientConfiguration();
```

```
if (getSentinelConfig() != null) {
                return new JedisConnectionFactory(getSentinelConfig(), clientConfiguration);
          }
          if (getClusterConfiguration() != null) {
                return new JedisConnectionFactory(getClusterConfiguration(),
                           clientConfiguration);
          return new JedisConnectionFactory(getStandaloneConfig(), clientConfiguration);
     }
     private JedisClientConfiguration getJedisClientConfiguration() {
          JedisClientConfigurationBuilder builder = applyProperties(
                     JedisClientConfiguration.builder());
          RedisProperties.Pool pool = this.properties.getJedis().getPool();
          if (pool != null) {
                applyPooling(pool, builder);
          }
          if (StringUtils.hasText(this.properties.getUrl())) {
                customize Configuration From Url (builder);\\
          }
          customize(builder);
          return builder.build();
     }
     private JedisClientConfigurationBuilder applyProperties(
                JedisClientConfigurationBuilder builder) {
          if (this.properties.isSsl()) {
                builder.useSsl();
          }
          if (this.properties.getTimeout() != null) {
                Duration timeout = this.properties.getTimeout();
                builder.readTimeout(timeout).connectTimeout(timeout);
          }
          return builder;
     }
     private void applyPooling(RedisProperties.Pool pool,
                JedisClientConfiguration.JedisClientConfigurationBuilder builder) {
          builder.use Pooling ().pool Config (jed is Pool Config (pool));\\
     }
     private JedisPoolConfig jedisPoolConfig(RedisProperties.Pool pool) {
          JedisPoolConfig config = new JedisPoolConfig();
          config.setMaxTotal(pool.getMaxActive());
          config.setMaxIdle(pool.getMaxIdle());
          config.setMinIdle(pool.getMinIdle());
          if (pool.getMaxWait() != null) {
                config.setMaxWaitMillis(pool.getMaxWait().toMillis());
          }
          return config;
     }
}
```