1、springboot 工程搭建

一个 springboot 工程的搭建其实很简单
1、导入必要的 jar 包，如：

```
<!-- 继承父工程 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Web 启动器

```
<!-- web 启动器 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!--排除 Tomcat 启动器，如果用 jetty 需要排除，如果要打包(war)

部署到服务器需要排除内置 Tomcat -->
    <!--          <exclusions>
                <exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
            </exclusions>-->
</dependency>
```

启动类：

```
@SpringBootApplication(scanBasePackages =
{"com.xiangxue.jack"})
@MapperScan("com.xiangxue.jack.dao")
@ServletComponentScan(basePackages = {"com.xiangxue.jack"})
@EnableConfigurationProperties(DruidConfig.class)
public class SpringbootTest extends
SpringBootServletInitializer {

    public static void main(String[] args) {
```

```
        ConfigurableApplicationContext applicationContext =
SpringApplication.run(SpringbootTest.class,
                args);
    }

    @Override
    protected SpringApplicationBuilder
configure(SpringApplicationBuilder builder) {
        return builder.sources(SpringBootTest.class);
    }
}
```

注解解释：

```
@SpringBootApplication(scanBasePackages                    =
{"com.xiangxue.jack"})
```

扫描包下面的注解并把类实例化加入到 spring 容器中

```
@MapperScan("com.xiangxue.jack.dao")
```

扫描该包，把 dao 包下的接口生成代理实例

```
@ServletComponentScan(basePackages                         =
{"com.xiangxue.jack"})
```

扫描*@WebFilter，@WebListener，@WebServlet*

注解，把 Servlet、Filter、Listener 加入到 spring 容器中、

```
@EnableConfigurationProperties(DruidConfig.class)
```

开启配置文件读取功能，DruidConfig 类必须要有

```
@ConfigurationProperties(prefix =
"spring.druid",ignoreInvalidFields = true)
```

注解，该注解会读取默认的配置文件 application.properties 配置，然后

会读取 **spring.druid** 作为前缀的配置属性。

2、springboot 整合 Servlet、Filter、Listener

有的时候需要在 springboot 工程里面加入 Servlet、Filter 和 Listener，这时候只需要再启动类上加入：

```
@ServletComponentScan(basePackages = {"com.xiangxue.jack"})
```

注解，然后在 Servlet、Filter、Listener 上面加入响应注解即可。如：

```
@WebServlet(urlPatterns = "/jack/*")
public class JackServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
```

```
@WebFilter(urlPatterns = "/*",filterName = "myFilter")
public class MyFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request, ServletResponse
        System.out.println("--------MyFilter----------");
        chain.doFilter(request,response);
    }
}
```

```
@WebListener
public class MyListener implements ServletContextListener {

    @Override
    public void contextDestroyed(ServletContextEvent contextEvent) {
        System.out.println("contextDestroyed"); }
    @Override
    public void contextInitialized(ServletContextEvent contextEvent) {
        System.out.println("contextInitialized"); }
}
```

3、springboot 整合 druid

Druid 是一个非常优秀的连接池，非常好的管理了数据库连接，可以实时监控数据库连接对象和应用程序的数据库操作记录

1、jar 包导入

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.26</version>
</dependency>
```

2、druid 配置

数据库连接配置加载

```java
@Data
@Configuration
@ConfigurationProperties(prefix = "spring.druid",ignoreInvalidFields = true)
public class DruidConfig {

    private String driverClassName;
    private String jdbcUrl;
    private String jdbcUrl1;
    private String username;
    private String password;
    private int maxActive;
    private int minIdle;
    private int initialSize;
    private Long timeBetweenEvictionRunsMillis;
    private Long minEvictableIdleTimeMillis;
    private String validationQuery;
    private boolean testWhileIdle;
    private boolean testOnBorrow;
    private boolean testOnReturn;
    private boolean poolPreparedStatements;
    private Integer maxPoolPreparedStatementPerConnectionSize;
    private String filters;
    private String connectionProperties;
```

读取application.properties配置文件中以spring.druid作为前缀的配置属性值。

数据源对象创建并加入到 spring 容器中

```java
@Bean(destroyMethod = "close",initMethod = "init")
public DataSource getDs1(){
    DruidDataSource druidDataSource = new DruidDataSource();
    druidDataSource.setDriverClassName(driverClassName);
    druidDataSource.setUrl(jdbcUrl);
    druidDataSource.setUsername(username);
    druidDataSource.setPassword(password);
    druidDataSource.setMaxActive(maxActive);
    druidDataSource.setInitialSize(initialSize);
    druidDataSource.setTimeBetweenConnectErrorMillis(timeBetweenEvictionRunsMillis);
    druidDataSource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);
    druidDataSource.setValidationQuery(validationQuery);
    druidDataSource.setTestWhileIdle(testWhileIdle);
    druidDataSource.setTestOnBorrow(testOnBorrow);
    druidDataSource.setTestOnReturn(testOnReturn);
    druidDataSource.setPoolPreparedStatements(poolPreparedStatements);
    druidDataSource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);

    try {
        druidDataSource.setFilters(filters);
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return druidDataSource;
}
```

Druid 数据监控配置

```java
@Bean
public ServletRegistrationBean druidStateViewServlet(){
    ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(new StatViewServlet(), ...urlMa
    //初始化参数initParams
    //添加白名单
    servletRegistrationBean.addInitParameter( name: "allow", value: "");
    //添加ip黑名单
    servletRegistrationBean.addInitParameter( name: "deny", value: "192.168.16.111");
    //登录查看信息的账号密码
    servletRegistrationBean.addInitParameter( name: "loginUsername", value: "admin");
    servletRegistrationBean.addInitParameter( name: "loginPassword", value: "123");
    //是否能够重置数据
    servletRegistrationBean.addInitParameter( name: "resetEnable", value: "false");
    return servletRegistrationBean;
}
```

```java
/**
 * 过滤不需要监控的后缀
 * @return
 */
@Bean
public FilterRegistrationBean druidStatFilter(){
    FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(new WebStatFilter());
    //添加过滤规则
    filterRegistrationBean.addUrlPatterns("/*");
    //添加不需要忽略的格式信息
    filterRegistrationBean.addInitParameter( name: "exclusions", value: "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/
    return filterRegistrationBean;
}
```

4、springboot 整合 mybatis

Mybatis 是一个非常优秀的 ORM 框架，公司里面用得也比较多
1、jar 包导入

```
<!-- 把 mybatis 的启动器引入 -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>RELEASE</version>
</dependency>
```

2、application.properties 配置
把该包下的 bean 生成别名

```
mybatis.typeAliasesPackage=com.xiangxue.jack.bean
```

Mybatis 会把这个路径下的 xml 解析出来建立接口的映射关系

```
mybatis.mapperLocations=classpath:com/xiangxue/jack/xml/*Mapper.xml
```

3、启动了添加扫描注解

```java
@SpringBootApplication(scanBasePackages = {"com.xiangxue.jack"})
@MapperScan("com.xiangxue.jack.dao")
@ServletComponentScan(basePackages = {"com.xiangxue.jack"})
@EnableConfigurationProperties(DruidConfig.class)
public class SpringbootTest extends SpringBootServletInitializer {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext = SpringApp
                args);
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBui
        return builder.sources(SpringBootTest.class);
    }
}
```

该注解会扫描 dao 包下的接口，把接口生成代理对象并加入到 spring 容器中，在业务代码里面就可以按照类型注入了。如图：

```
@Service
public class AreaServiceImpl implements AreaService {

    @Autowired
    CommonMapper mapper;
```

5、springboot 整合 JPA

JPA 也是一个非常优秀的 ORM 框架，用起来也非常简单

1、jar 包导入

```
<!-- Spring Boot JPA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

2、application.properties 配置

自动创建表

```
spring.jpa.hibernate.ddl-auto:update
```

打印 sql 语句

```
spring.jpa.show-sql:true
```

3、实例 bean

```
@Data
@Entity
@Table(name = "xx_student")
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "name")
    private String name;

    @Column(name = "cardNum")
    private String cardNum;
}
```

当启动的时候会自动生成 xx_student 表

4、dao 配置

```java
/*
* Student操作对象
* Integer主键类型
*
* */
public interface StudentDao extends JpaRepository<Student,Integer> {

}
```

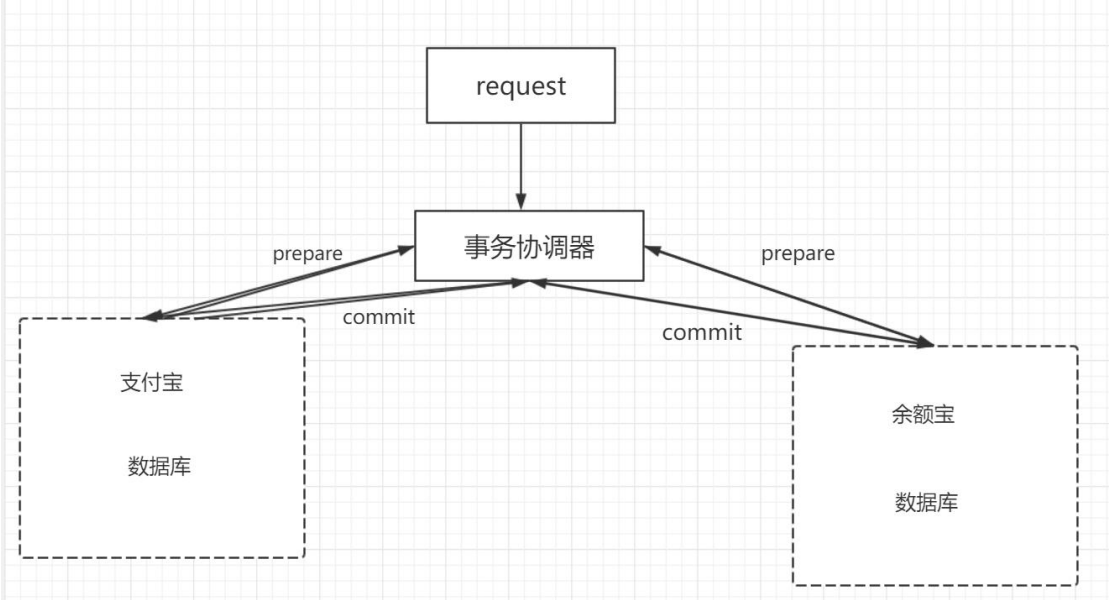
泛型中指定了操作的实体 bean 是哪一个，JpaRepository 中定义了该实体的增删改查的所有方法，用 studentDao 对象调用即可

5、业务代码中使用

```java
@Service
public class StudentServiceImpl implements StudentService {

    @Autowired
    private StudentDao studentDao;

    @Override
    public List<Student> findAll() { return studentDao.findAll(); }

    @Override
    public Student findById(Integer id) {

        //
        Optional<Student> byId = studentDao.findById(id);

        if(byId.isPresent()) {
            return byId.get();
        }
        return null;
    }
```

6、springboot 整合 atomikos

Atomikos 是一个基于 XA 协议的分布式事务解决管理框架，其核心思想就是两段提交，一般使用在在一个业务方法里面涉及到的对多个数据源的操作，要保证在这个业务方法操作中，保持对两个数据源的同时提交和同时回滚。如图所示：



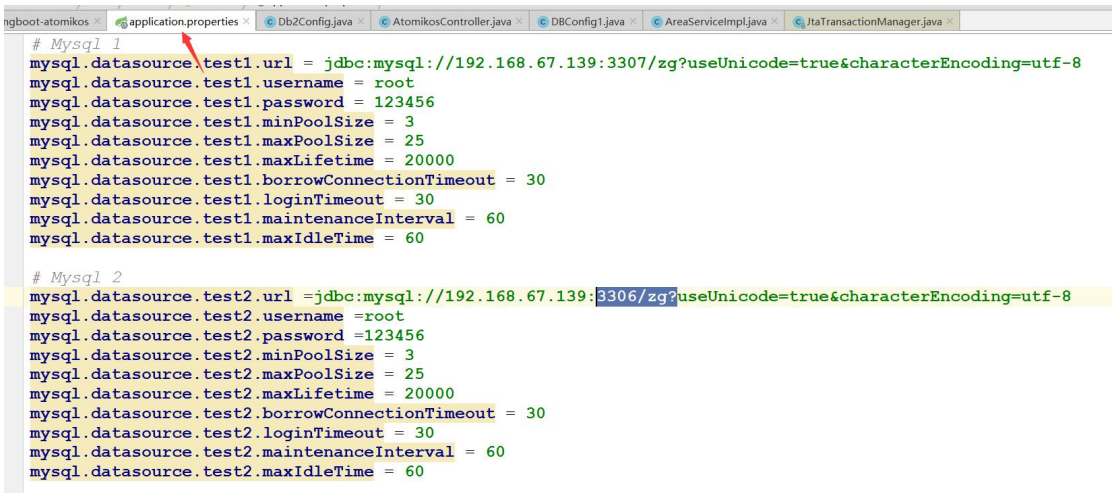
Atomikos 就是图中的事务协调器的角色，负责对两个数据源的管理，同时提交或同时回滚。

就是在真正提交之前有一个预提交的过程，就是检测两个数据源是否能够提交，如果有一个返回 No，那么这个事务就不能提交。具体配置：

1、jar 包导入

```
<dependency>
    <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jta-atomikos</artifactId>
</dependency>
```

2、两个数据源连接信息配置



3、数据源创建

```
@Bean(name = "test1DataSource")
public DataSource testDataSource() {
    MysqlXADataSource mysqlXaDataSource = new MysqlXADataSource();
    mysqlXaDataSource.setUrl(testConfig.getUrl());
    mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(true);
    mysqlXaDataSource.setPassword(testConfig.getPassword());
    mysqlXaDataSource.setUser(testConfig.getUsername());
    mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(true);

    AtomikosDataSourceBean xaDataSource = new AtomikosDataSourceBean();
    xaDataSource.setXaDataSource(mysqlXaDataSource);
    xaDataSource.setUniqueResourceName("test1DataSource");
    xaDataSource.setMinPoolSize(testConfig.getMinPoolSize());
    xaDataSource.setMaxPoolSize(testConfig.getMaxPoolSize());
    xaDataSource.setMaxLifetime(testConfig.getMaxLifetime());
    xaDataSource.setBorrowConnectionTimeout(testConfig.getBorrowConnectionTimeout());
    try {
        xaDataSource.setLoginTimeout(testConfig.getLoginTimeout());
    } catch (SQLException e) {
        e.printStackTrace();
    }
    xaDataSource.setMaintenanceInterval(testConfig.getMaintenanceInterval());
    xaDataSource.setMaxIdleTime(testConfig.getMaxIdleTime());
    xaDataSource.setTestQuery(testConfig.getTestQuery());
    return xaDataSource;
}

@Bean(name = "test1SqlSessionFactory")
public SqlSessionFactory testSqlSessionFactory(@Qualifier("test1DataSource") DataSource dataSource)
        throws Exception {
    SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
    bean.setDataSource(dataSource);
    return bean.getObject();
}

@Bean(name = "test1SqlSessionTemplate")
public SqlSessionTemplate testSqlSessionTemplate(
        @Qualifier("test1SqlSessionFactory") SqlSessionFactory sqlSessionFactory) throws Exception {
    return new SqlSessionTemplate(sqlSessionFactory);
}
```

4、具体代码案例

```
@Service
public class AreaServiceImpl implements AreaService {

    @Autowired
    private CommonMapper1 commonMapper1;

    @Autowired
    private CommonMapper2 commonMapper2;

    @Autowired
    TransactionManager transactionManager;

    @Transactional
    public int saveArea(ConsultConfigArea area) {
        System.out.println(transactionManager);
        JtaTransactionManager jtaTransactionManager = (JtaTransactionManager)transactionManager;
        System.out.println(jtaTransactionManager.getUserTransaction());
        UserTransaction userTransaction = jtaTransactionManager.getUserTransaction();
        int count = commonMapper1.addArea(area);
        int count1 = commonMapper2.addArea(area);
        if(true) throw new RuntimeException("xx");
        return count;
    }
}
```

这里 spring 事务已经交给 atomikos 来管理了，其实就是由 atomikos 中的 JtaTransationManager 来管理事务，重写了 commit 和 Rollback 和 getTransaction 方法，只是 commit 和 Rollback 是对两个数据源的操作而已。

7、springboot 整合 redis

1、jar 包导入

```
<dependency>
    <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2、application.properties 配置

```
# Redis 数据库索引（默认为 0）
spring.redis.database=0
# Redis 服务器地址
spring.redis.host=192.168.67.139
# Redis 服务器连接端口
spring.redis.port=6379
# Redis 服务器连接密码（默认为空）
spring.redis.password=
# 连接池最大连接数（使用负值表示没有限制）
spring.redis.pool.max-active=8
# 连接池最大阻塞等待时间（使用负值表示没有限制）
spring.redis.pool.max-wait=-1
# 连接池中的最大空闲连接
spring.redis.pool.max-idle=8
# 连接池中的最小空闲连接
spring.redis.pool.min-idle=0
# 连接超时时间（毫秒）
spring.redis.timeout=0
```

3、把 redis 整合到 spring 的缓存体系中
CacheManager 对象创建

```
//缓存管理器
@Bean
public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {
    RedisCacheConfiguration redisCacheConfiguration = RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofHours(1)); // 设置缓存有效期一小时
    return RedisCacheManager
            .builder(RedisCacheWriter.nonLockingRedisCacheWriter(redisConnectionFactory))
            .cacheDefaults(redisCacheConfiguration).build();
}
```

创建 redis 连接对象

```java
@Bean
public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory) {
    RedisTemplate<String, Object> template = new RedisTemplate<>();
    // 配置连接工厂
    template.setConnectionFactory(factory);
    //使用Jackson2JsonRedisSerializer来序列化和反序列化redis的value值（默认使用JDK的序列化方式）
    Jackson2JsonRedisSerializer jacksonSeial = new Jackson2JsonRedisSerializer(Object.class);
    ObjectMapper om = new ObjectMapper();
    // 指定要序列化的域，field,get和set,以及修饰符范围，ANY是都有包括private和public
    om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
    // 指定序列化输入的类型，类必须是非final修饰的，final修饰的类，比如String,Integer等会跑出异常
    om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
    jacksonSeial.setObjectMapper(om);
    // 值采用json序列化
    template.setValueSerializer(jacksonSeial);
    //使用StringRedisSerializer来序列化和反序列化redis的key值
    template.setKeySerializer(new StringRedisSerializer());
    // 设置hash key 和value序列化模式
    template.setHashKeySerializer(new StringRedisSerializer());
    template.setHashValueSerializer(jacksonSeial);
    template.afterPropertiesSet();
    return template;
}
```

4、在代码中的使用

```java
@Cacheable(cacheNames = "redisCache",key = "'jack' + #id")
@Override
public String queryData(String id) {
    System.out.println("======CacheServiceImpl.queryData");
    List<ConsultConfigArea> areas = commonMapper.queryAreaById(id);
    return JSONObject.toJSONString(areas);
}

@CachePut(cacheNames = "redisCache",key = "'jack' + #id")
@Override
public String putCache(String id) {
    System.out.println("======CacheServiceImpl.queryData");
    List<ConsultConfigArea> areas = commonMapper.queryAreaById(id);
    return JSONObject.toJSONString(areas);
}
```

在业务代码中我们只要在方法上面加上@Cacheable@CachePut 注解就可以了，加上这个注解再配置 RedisCacheManager 的整合，spring 就会把业务代码的返回结果存到 redis 中。

8、springboot 整合 mongodb

1、jar 包导入

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

2、application.properties 配置

```
spring.data.mongodb.uri=mongodb://192.168.67.139:27017/aa
_db
```

3、代码案例

```
@Service
public class MongoServiceImpl implements MongoService<User> {
    /*
    * 这个实例是从哪里来的？？
    * */
    @Autowired
    private MongoTemplate mongoTemplate;

    @Override
    public String save(User obj) {
        mongoTemplate.save(obj);
        return "1";
    }

    @Override
    public List<User> findAll() {
        List<User> all = mongoTemplate.findAll(User.class);
        return all;
    }

    @Override
    public User getById(String id) {
        Query query = new Query(Criteria.where("_id").is(id));
        return mongoTemplate.findOne(query,User.class);
    }
```

mongoTemplate 对象是可以直接依赖注入进来的，由启动器创建了该类的对象。

9、springboot 整合 JAX-RS 规范

JAX-RS 是 JAVA EE6 引入的一个新技术。 JAX-RS 即 Java API for RESTful Web Services，是一个 Java 编程语言的应用程序接口，支持按照表述性状态转移（REST）架构风格创建 Web 服务。JAX-RS 使用了 Java SE5 引入的 Java 注解来简化 Web 服务的客户端和服务端的开发和部署。
其实就是类似于 Servlet 规范来接收用户请求的一个规范。

1、jar 包导入

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

2、代码配置

```
@Configuration
public class JerseyConfig {
    @Bean
    public ServletRegistrationBean jerseyServlet() {

        //手动注册 servlet

        ServletRegistrationBean registrationBean = new
ServletRegistrationBean(new ServletContainer(), "/rest/*");
```

```
registrationBean.addInitParameter(ServletProperties.JAXRS
_APPLICATION_CLASS,JerseyResourceConfig.class.getName());
        return registrationBean;
    }
}
```

手动注册了一个 Servlet，拦截的路径是/rest/*

定义 Jersey 的资源加载类 JerseyResourceConfig，如图：

```
public class JerseyResourceConfig extends ResourceConfig {

    public JerseyResourceConfig() {

        register(RequestContextFilter.class);

        // 加载资源文件,这里直接扫描com.xiangxue.jack.jersey下的所有api  JAX-RS
        packages("com.xiangxue.jack.jersey");
    }
}
```

Packages 方法是扫描包的路径，扫描 jersey 规范中相应的注解。如图：

```
/*
* 可以接受http请求
* */
@Path("/jersey/")
public class JerseyController {

    @Path("{id}")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public String hello(@PathParam("id") Long id) { return "hello"; }
}
```

10、springboot 整合 JSP

1、jar 包导入

```
<!-- 这个两个 jar 包是 springboot 支持 jsp 的 jar -->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

2、application.properties 配置

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
```

11、springboot 整合 freemarker

freemarker 是一款比较优秀的模板引擎
1、jar 包导入

```
<!-- 引入 freeMarker 的依赖包. -->
<dependency>
    <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-freemarker</artifactId>
</dependency>
```

2、application.properties 配置

```
spring.freemarker.allow-request-override=false
spring.freemarker.cache=true
spring.freemarker.check-template-location=true
spring.freemarker.charset=UTF-8
spring.freemarker.content-type=text/html
spring.freemarker.expose-request-attributes=false
spring.freemarker.expose-session-attributes=false
spring.freemarker.expose-spring-macro-helpers=false
```

12、springboot 整合 swagger2

swagger2 是一个可以根据接口定义自动生成接口 API 文档的框架
1、jar 包导入

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

2、代码配置

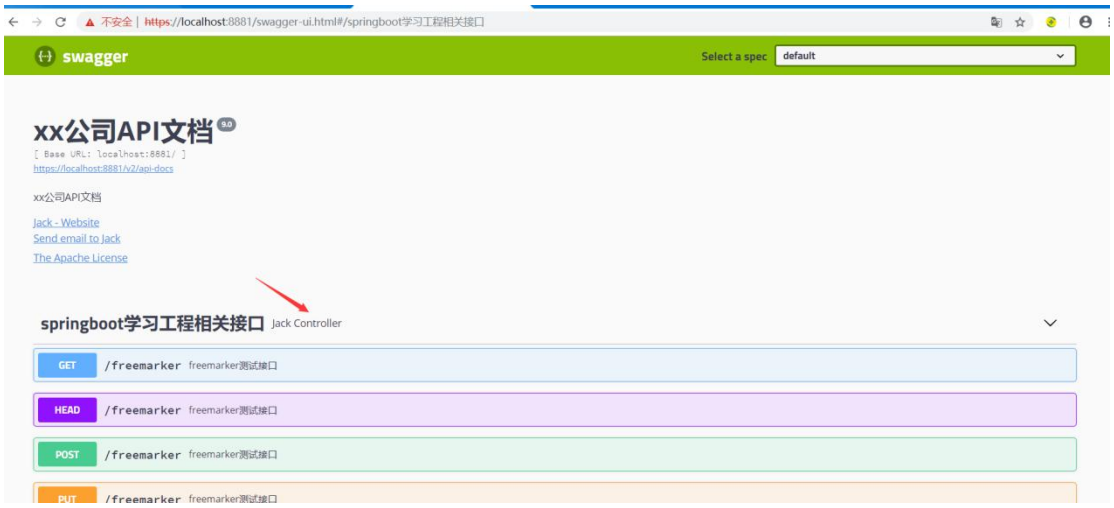
```java
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .pathMapping("/")
                .select()
                .apis(RequestHandlerSelectors.basePackage("com.xiangxue.jack.controller"))
                .paths(PathSelectors.any())
                .build().apiInfo(new ApiInfoBuilder()
                        .title("xx公司API文档")
                        .description("xx公司API文档")
                        .version("9.0")
                        .contact(new Contact( name: "Jack",  url: "blog.csdn.net",  email: "aaa@gmail.com"))
                        .license("The Apache License")
                        .licenseUrl("http://www.baidu.com")
                        .build());
    }
}
```

3、具体使用

```java
@Controller
@Api(tags = "springboot学习工程相关接口")
public class JackController {

    private static final Logger logger = LoggerFactory.getLogger(JackController.class);

    @Autowired
    AreaService areaService;

    @Value("${application.field:default value jack}")
    private String zhuguangField = "";

    @ApiOperation("jsp测试接口")
    @ApiImplicitParams({
            @ApiImplicitParam(name = "username", value = "用户名", defaultValue = "jack"),
            @ApiImplicitParam(name = "address", value = "用户地址", defaultValue = "长沙")
    })
    @RequestMapping("/index")
    public ModelAndView index() {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("index");
        mv.addObject( attributeName: "time", new Date());
        mv.addObject( attributeName: "message", zhuguangField);
        return mv;
    }
```

4、ui 界面

界面请求 url：*https://localhost:8881/swagger-ui.html*



13、springboot 整合 Actuator 监控管理

Actuator 监控是一个用于监控 springboot 健康状况的工具，可以实时的工程的健康和调用情况

1、jar 包导入

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

2、application.properties 配置

actuator 默认只开放两个接口分别是：

http://localhost:8881/actuator/health
http://localhost:8881/actuator/info

如果想开放监控的所有界面，需要在 application.properties 进行配置

```
#默认只有 info/health
management.endpoints.web.exposure.include=*
```

这个监控不需要使用，是自动统计消息的。

1 /health/{component}/{instance} GET

报告程序的健康指标，这些数据由 HealthIndicator 实现类提供

2 /info GET

获取程序指定发布的信息，这些信息由配置文件中 info 打头的属性提供

3 /configprops GET

描述配置属性（包含默认值）如何注入到 bean

4 /beans GET

描述程序中的 bean，及之间的依赖关系

5 /env GET

获取全部环境属性

6 /env/{name} GET

根据名称获取指定的环境属性值

7 /mappings GET

描述全部的 URI 路径，及和控制器的映射关系

8 /metrics/{requiredMetricName} GET

统计程序的各种度量信息，如内存用量和请求数

9 /httptrace GET

提供基本的 http 请求跟踪信息，如请求头等

10 /threaddump GET

获取线程活动的快照

11 /conditions GET

提供自动配置报告，记录哪些自动配置通过，哪些没有通过

12 /loggers/{name} GET

查看日志配置信息

13 /auditevents GET

查看系统发布的事件信息

14 /caches/{cache} GET/DELETE

查看系统的缓存管理器，另可根据缓存管理器名称查询；另 DELETE 操作可清除缓存

15 /scheduledtasks GET

查看系统发布的定时任务信息

16 /features GET

查看 Springcloud 全家桶组件信息

17 /refresh POST

重启应用程序，慎用

18 /shutdown POST

关闭应用程序，慎用

14、springboot 整合 https

1、生成 https 证书

cd 到 jdk 的 bin 目录，执行生成证书的指令：

keytool -genkey -alias spring -keypass 123456 -keyalg RSA -keysize 1024 -validity 365 -keystore E:/springboot.keystore -storepass 123456

genkey 表示要创建一个新的密钥。

alias 表示 keystore 的别名。

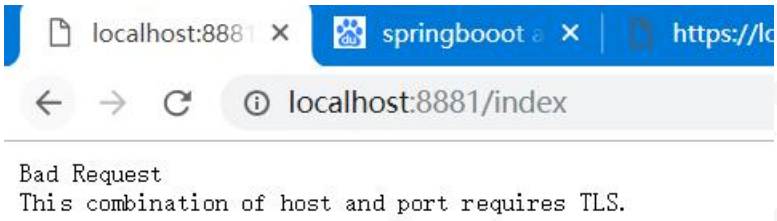
keyalg 表示使用的加密算法是 RSA ，一种非对称加密算法。

keysize 表示密钥的长度。
keystore 表示生成的密钥存放位置。
validity 表示密钥的有效时间，单位为天。

正式开发过程中，需要申请正式的被浏览器信任的证书

2、application.properties 配置
server.port=8881
server.ssl.key-password=123456
server.ssl.key-store=classpath:springboot.keystore
server.ssl.key-alias=spring

如果是 http 的方式访问则会报错



15、SpringBoot 工程 docker 化

微服务项目用 docker 来部署和管理是非常方便和节省资源的，所有 springboot 项目也有 docker 话的需求，springboot 项目 docker 是通过 maven 插件来实现的：
1、docker 化插件

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.4.13</version>
    <configuration>
        <imageName>jack/micro</imageName>

<dockerDirectory>${project.basedir}/src/main/docker</dock
erDirectory>
        <resources>
            <resource>
                <targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>
            </resource>
```

```
        </resources>
    </configuration>
</plugin>
```

2、Dockerfile 文件

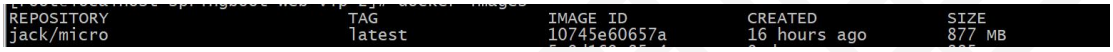
```
FROM docker.io/relateiq/oracle-java8
VOLUME /tmp
ADD springboot-web.jar app.jar
#RUN bash -c 'touch /app.jar'
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
EXPOSE 8881
```

3、把工程上传到 linux 环境并打包生成工程镜像

在 pom.xml 文件目录下执行指令

mvn package docker:build

这个指令就会根据 Dockerfile 的内容来生成镜像



然后根据镜像来启动容器

docker run -ti -d -p 8881:8881 --name springboot jack/micro

查看容器启动日志

docker logs -f springboot

这样 springboot 工程镜像化就完成了。

16、SpringBoot 整合 rabbitmq

1、jar 包导入

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

2、application 配置

```
spring.rabbitmq.host=192.168.88.139
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=admin
```

3、代码配置类

```java
@Configuration
public class RabbitmqConfig {

    @Bean(name = "message")
    public Queue queueMessage() {
        /*
         * 这个步骤就是往rabbitmq的broker里面创建一个队列
         * */
        return new Queue( name: "jack.message");
    }
    /*
     * 创建交换器
     * */
    @Bean
    public TopicExchange exchange() {
        return new TopicExchange( name: "exchange.message");
    }

    @Bean
    Binding bindingExchangeMessage(@Qualifier("message") Queue queueMessage,
                                   TopicExchange exchange) {
        return BindingBuilder.bind(queueMessage).to(exchange).
                with( routingKey: "jack.message.routeKey");
    }
}
```

4、消息发送和消费

消息发送

```java
@Slf4j
@Component
public class RabbitmqSender {

    @Autowired
    private AmqpTemplate amqpTemplate;

    public void sendMessage(String exchange,String roukekey,Message content) {
        try {
            log.info("============发送消息给余额宝==========" + content);
            amqpTemplate.convertAndSend(exchange, roukekey,
                    JSONObject.toJSONString(content));
        }catch (Exception e) {

        }
    }
}
```

消息消费

```java
@Slf4j
@Component
public class MessageListener {

    @Autowired
    OrderService orderService;

    @RabbitListener(queues = "jack.message.response")
    public void process(final String result) {
        log.info("============接收到余额宝转账成功的应答消息============" + result);
        orderService.updateMessage(result);
    }
}
```

分布式事务解决方案，请看 jack 老师写的博客：

https://blog.csdn.net/luoyang_java/article/details/84953241

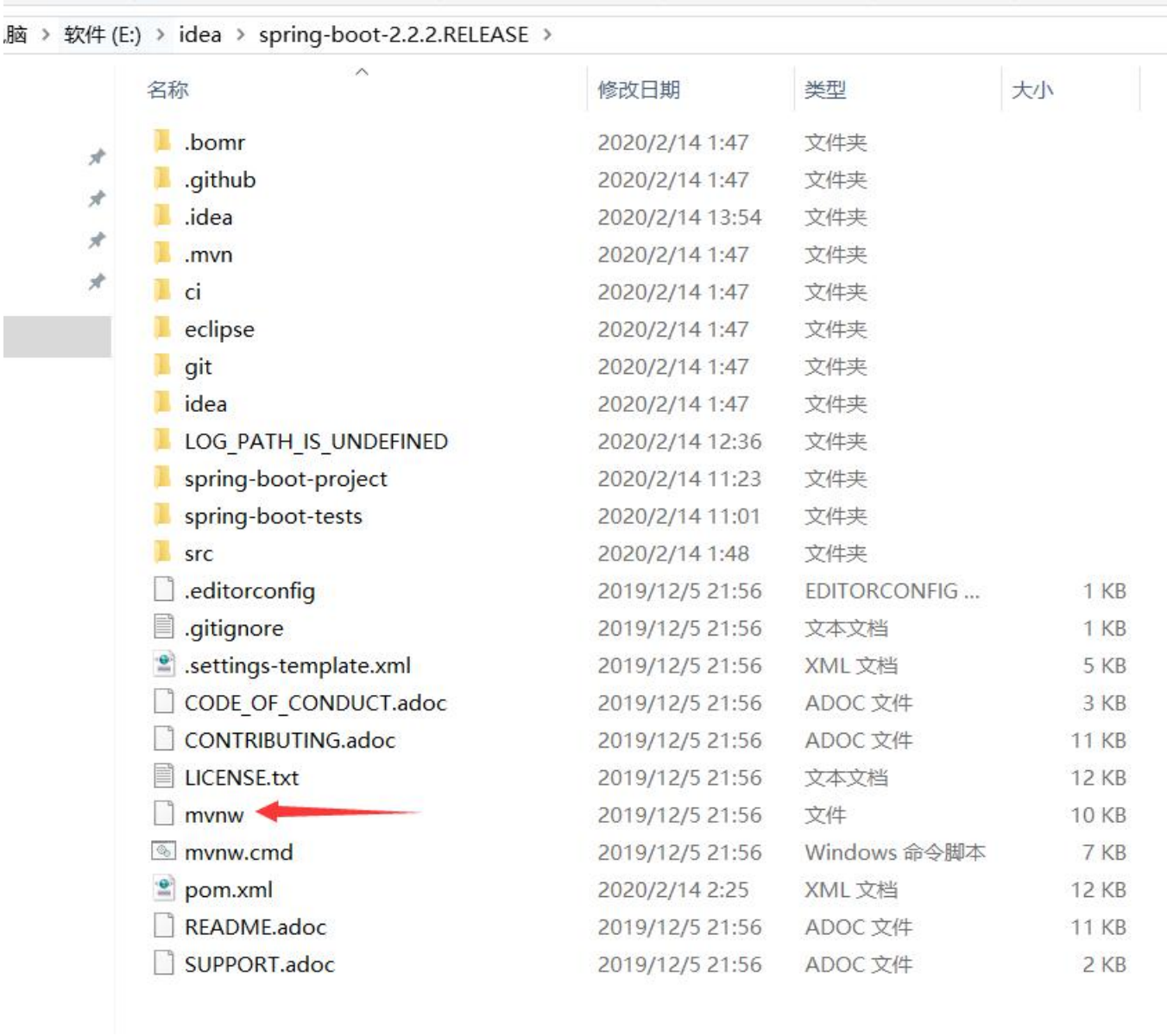
17、SpringBoot 源码构建

spring-boot-2.2.2.RELEASE.zip

1、进入到下载的源码目录执行命令

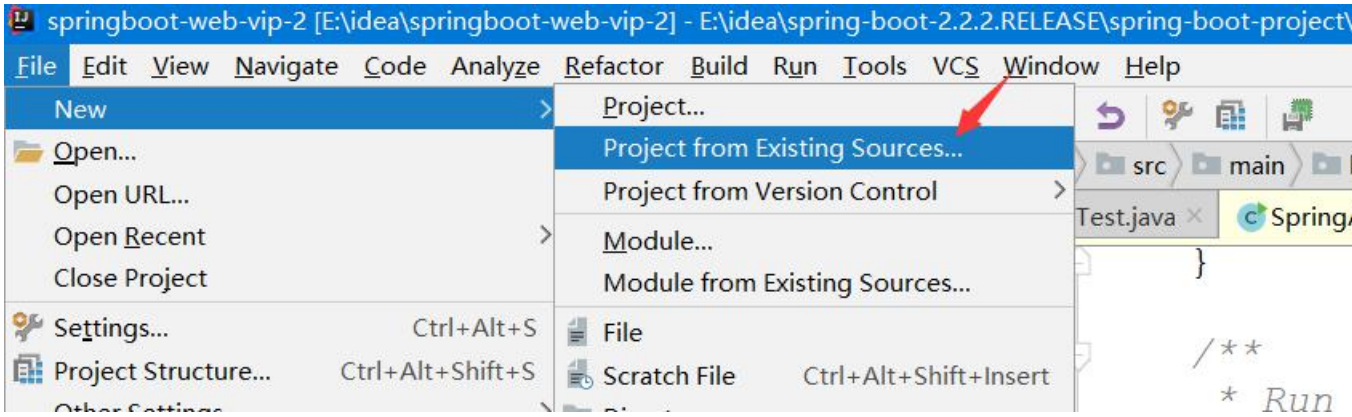
```
E:\idea\spring-boot-2.2.2.RELEASE>mvnw clean install -DskipTests -Pfast
Found "E:\idea\spring-boot-2.2.2.RELEASE\.mvn\wrapper\maven-wrapper.jar"
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] Spring Boot Build                                                  [pom]
```

mvnw clean install -DskipTests -Pfast

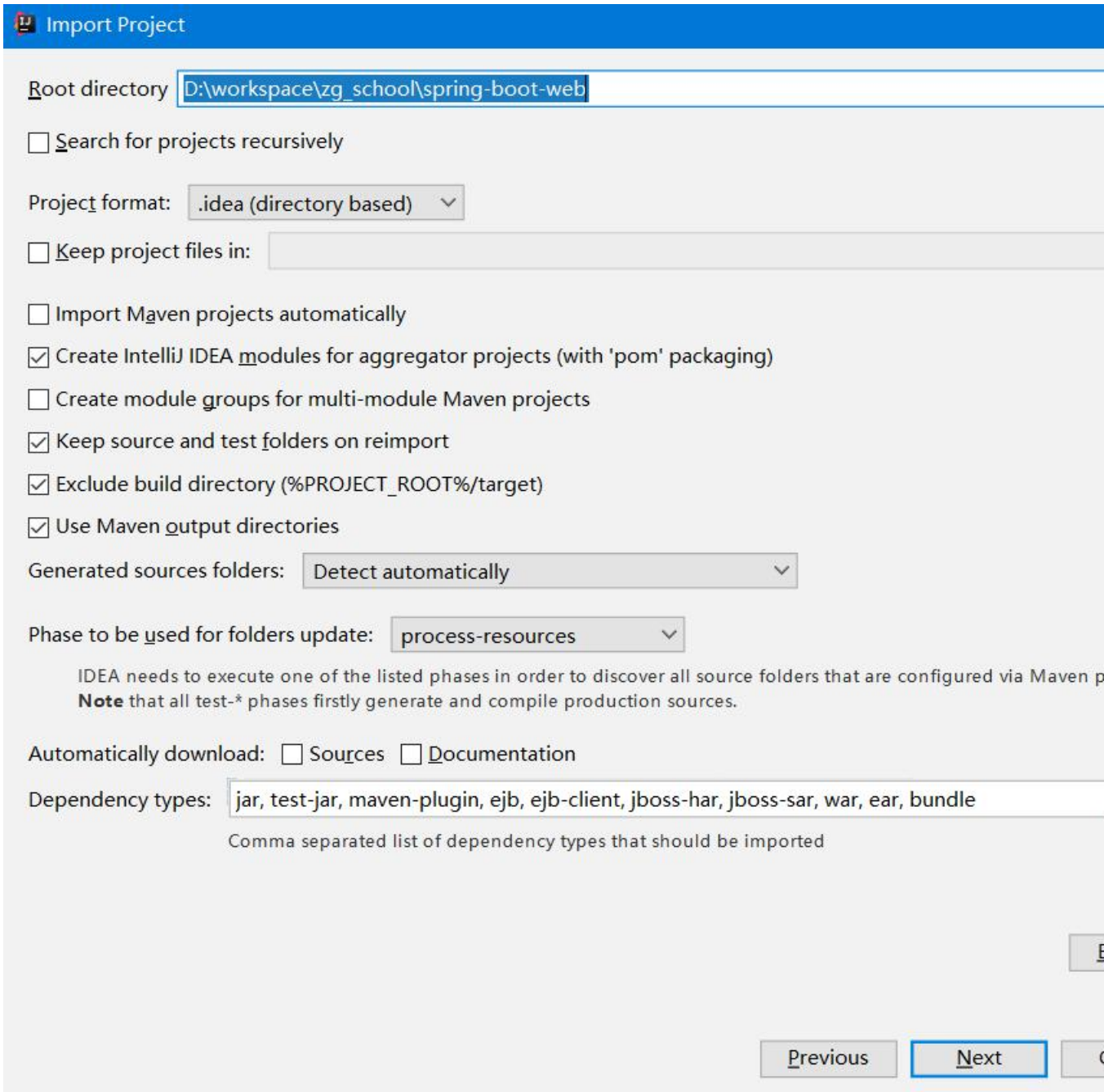


2、如果出现报错找不到 spring-javaformat 插件 执行 mvn spring-javaformat:apply 命令就可以了
命令执行成功后，再次执行指令：mvnw clean install -DskipTests -Pfast

3、把下载好的项目导入到 idea

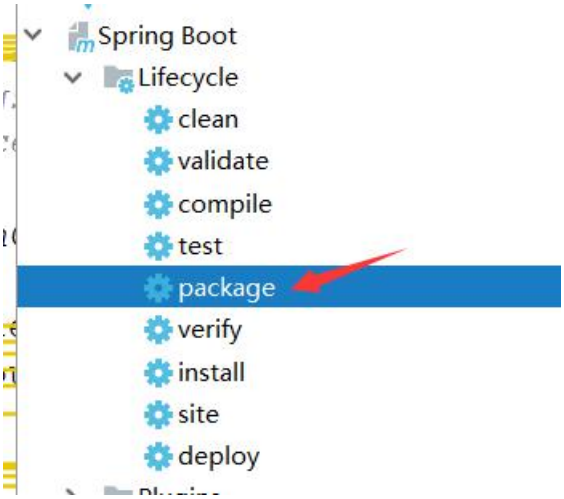


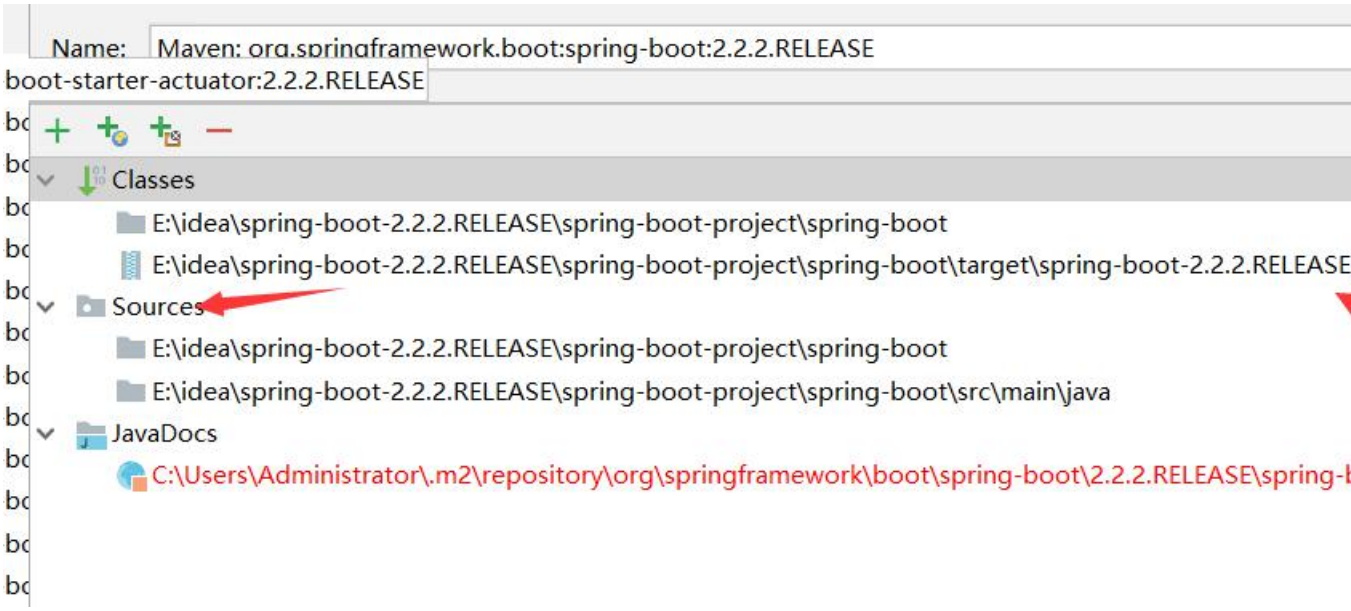

Import Project
Create project from existing sources
Import project from external model
Eclipse
Flash Builder
Gradle
Maven


Import Project
Root directory
D:\workspace\zg_school\spring-boot-web
Search for projects recursively
Project format:
.idea (directory based)
Keep project files in:
Import Maven projects automatically
Create IntelliJ IDEA modules for aggregator projects (with 'pom' packaging)
Create module groups for multi-module Maven projects
Keep source and test folders on reimport
Exclude build directory (%PROJECT_ROOT%/target)
Use Maven output directories
Generated sources folders:
Detect automatically
Phase to be used for folders update:
process-resources
IDEA needs to execute one of the listed phases in order to discover all source folders that are configured via Maven p
Note that all test-* phases firstly generate and compile production sources.
Automatically download:
Sources
Documentation
Dependency types:
jar, test-jar, maven-plugin, ejb, ejb-client, jboss-har, jboss-sar, war, ear, bundle
Comma separated list of dependency types that should be imported
Previous
Next

这里选择 maven 版本，maven 版本必须是 3.5 以上的版本，要不然 springboot 项目导入会有问题

springboot 打 jar 包
打包的时候一定要把 test 包删掉，要不然会执行测试用例，然后把打成的 jar 包导入 springboot demo 工程即可





如果编译源码时`<disable.checks>true</disable.checks>`

标签报错，这里把标签内容修改为 true 就可以了。

18、SpringBoot 项目启动

把 springboot 项目打成 jar 包启动方式：
java -jar springboot-web.jar
把 springboot 项目打成 war 启动方式：

要修改一下启动类

```java
@SpringBootApplication(scanBasePackages = {"com.xiangxue.jack"})
@MapperScan("com.xiangxue.jack.dao")
@ServletComponentScan(basePackages = {"com.xiangxue.jack"})
@EnableConfigurationProperties(DruidConfig.class)
public class SpringbootTest extends SpringBootServletInitializer {

    /*
    * 1、要完成Spring容器的启动
    * 2、把项目部署到tomcat
    * */
    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext = SpringApplication.run(SpringbootTest.class,
                args);
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(SpringBootTest.class);
    }
}
```

启动类要继承 SpringBootServletInitializer

实现 configure 方法

war 包的启动也是一样的：java -jar springboot-web.jar

当然 war 包启动，也可以部署到 tomcat 的 webapps 下面，启动外部的 tomcat 来部署，但是需要把 springboot 内置的 tomcat 去掉，如图：

```xml
<!-- web 启动器 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!--排除Tomcat启动器，如果用jetty需要排除，如果要打包(war)部署到服务器需要排除内
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

排除掉 web 启动器中的 tomcat 就可以了。

19、SpringBoot 启动源码

为什么 springboot 工程能够在 mian 方法中完成启动呢？

需要大家掌握的有几个点：

1、SPI

SPI 在 springboot 中是去读取 META-INF/spring.factories 目录的配置文件内容，把配置文件中的类加载到 spring 容器中。这样如果你想把一个类加载到 spring 容器中，也可以采用这种方式来做。把类配置到 spring.factories 配置文件中即可。

那么我们总结一下，如果你想把一个类加载到 spring 容器中管理有几种方式：

1、通过 xml 的 bean 标签

2、通过加@Component 注解被@ComponentScan 扫描

3、通过在 spring.factories 配置该类

前两者是加载本工程的 bean，扫描本工程的 bean，第三点可以加载第三方定义的 jar 包中的 bean，毕竟第三方 jar 包的包名跟本工程包名可能不一样，所以前两个方式扫描不到。

2、创建 springboot 的上下文对象

```
// 创建上下文对象
context = createApplicationContext();
```

```
*/
public static final String DEFAULT_SERVLET_WEB_CONTEXT_CLASS = "org.springframework.boot."
        + "web.servlet.context.AnnotationConfigServletWebServerApplicationContext";
```

在这个上下文对象构造函数中把 ConfigurationClassPostProcessor 变成 beanDefinition 对象。

3、容器的启动

```
// 核心方法，启动spring容器
refreshContext(context);
```

其实很简单就是调用了上下文对象的 refresh 核心方法：

```
private void refreshContext(ConfigurableApplicationContext context) {
    refresh(context);
    if (this.registerShutdownHook) {
        try {
            context.registerShutdownHook();
        }
        catch (AccessControlException ex) {
            // Not allowed in some environments.
        }
    }
}
```

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

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

            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            initMessageSource();

            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();
```

这里就不再赘述，spring 源码讲得很清楚。其中 springboot 要关注了是一个钩子方法 onRefresh();

4、内置 tomcat 的启动和部署
Tomcat 的启动在 onRefresh()中：

```
@Override
public WebServer getWebServer(ServletContextInitializer... initializers) {
    if (this.disableMBeanRegistry) {
        Registry.disableRegistry();
    }
    Tomcat tomcat = new Tomcat();
    File baseDir = (this.baseDirectory != null) ? this.baseDirectory : createTempDir( prefix: "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);
}
```

从源码我们看到，springboot 的启动就是核心就是完成了两件事，一个是 spring 容器的启动调用了 refresh 核心方法，一个是 tomcat 的启动，new 出了一个内置的 tomcat。

20、SpringBoot 自动配置源码

为什么要有 springboot 自动配置功能？
在 springboot 项目中，我们可以在业务代码里面用事务注解，用缓存注解，用 mvn 相关的功能等等，但是我们并没有在 springboot 项目把这些功能开启添加进来，那么为什么我们可以在业务代码中使用这些功能呢？也就是说这些功能如何跑到 springboot 项目中来的呢？这就是 springboot 的自动配置功能

自动配置功能开启，我们看看启动类：

```
@SpringBootApplication(scanBasePackages = {"com.xiangxue.jack"})
@MapperScan("com.xiangxue.jack.dao")
@ServletComponentScan(basePackages = {"com.xiangxue.jack"})
@EnableConfigurationProperties(DruidConfig.class)
public class SpringbootTest extends SpringBootServletInitializer {

    /*
```

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFi
public @interface SpringBootApplication {

    /**
```

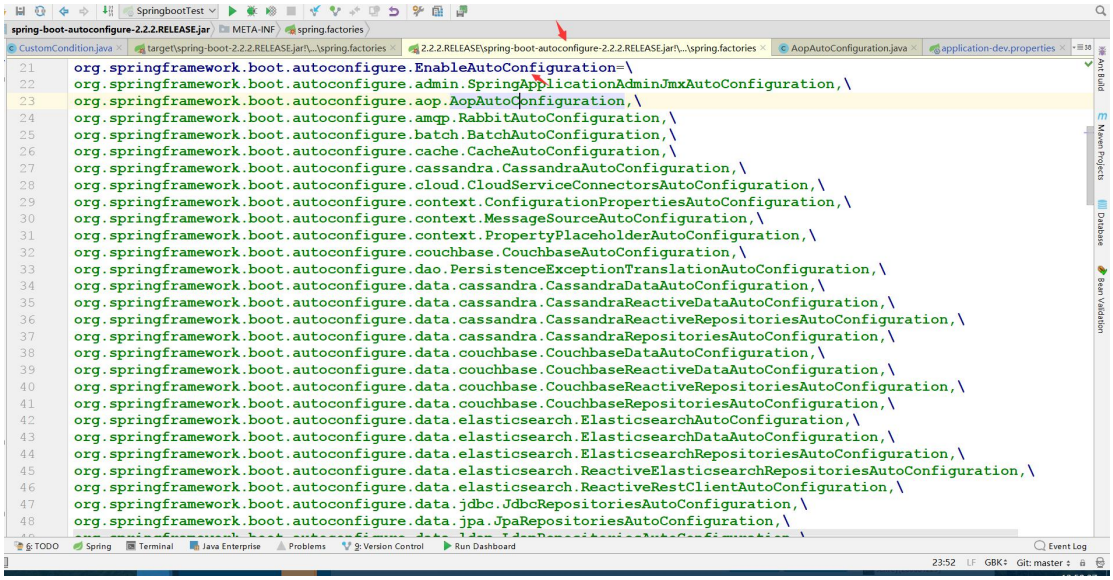
```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

我们看看这个类

```java
public class AutoConfigurationImportSelector implements DeferredImportSelector, BeanClassLoaderAware,
        ResourceLoaderAware, BeanFactoryAware, EnvironmentAware, Ordered {

    private static final AutoConfigurationEntry EMPTY_ENTRY = new AutoConfigurationEntry();
```

实现了 DeferredImportSelector 接口。
这个类的核心功能是通过 SPI 机制收集 EnableAutoConfiguration 为 key 的所有类，然后通过 ConfigurationClassPostProcessor 这个类调用到该类中的方法，把收集到的类变成 beanDefinition 对象最终实例化加入到 spring 容器。

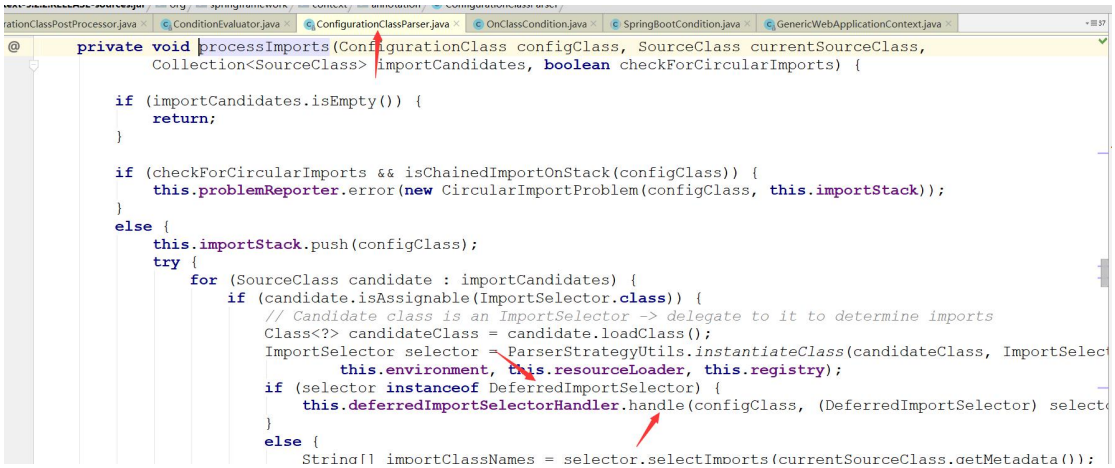
EnableAutoConfiguration 为 key 的所有类：该配置文件在 spring-boot-autoconfigure jar 包中。



AutoConfigurationImportSelector 类中两个方法会被 ConfigurationClassPostProcessor 调到：

```java
@Override
public void process(AnnotationMetadata annotationMetadata, DeferredImportSelector deferredImportSelector) {

@Override
public Iterable<Entry> selectImports() {...}
```

在这两个方法中完成了 SPI 类的收集。
ConfigurationClassPostProcessor 类只是把收集到的类变成 beanDefinition 并加入到 spring 容器。
ConfigurationClassPostProcessor 类调用的地方。



```java
private void processImports(ConfigurationClass configClass, SourceClass currentSourceClass,
        Collection<SourceClass> importCandidates, boolean checkForCircularImports) {

    if (importCandidates.isEmpty()) {
        return;
    }

    if (checkForCircularImports && isChainedImportOnStack(configClass)) {
        this.problemReporter.error(new CircularImportProblem(configClass, this.importStack));
    }
    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();
                    ImportSelector selector = ParserStrategyUtils.instantiateClass(candidateClass, ImportSelect
                            this.environment, this.resourceLoader, this.registry);
                    if (selector instanceof DeferredImportSelector) {
                        this.deferredImportSelectorHandler.handle(configClass, (DeferredImportSelector) selecto
                    }
                    else {
                        String[] importClassNames = selector.selectImports(currentSourceClass.getMetadata());
```

```java
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);
    }
}
```

```java
public void processGroupImports() {
    for (DeferredImportSelectorGrouping grouping : this.groupings.values()) {
        grouping.getImports().forEach(entry -> {
            ConfigurationClass configurationClass = this.configurationClasses.get(
                    entry.getMetadata());
            try {
                processImports(configurationClass, asSourceClass(configurationClass),
                        asSourceClasses(entry.getImportClassName()), checkForCircularImports: false);
            }
            catch (BeanDefinitionStoreException ex) {
                throw ex;
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                        "Failed to process import candidates for configuration class [" +
                                configurationClass.getMetadata().getClassName() + "]", ex);
            }
        });
    }
}
```

```java
public Iterable<Group.Entry> getImports() {
    for (DeferredImportSelectorHolder deferredImport : this.deferredImports) {
        this.group.process(deferredImport.getConfigurationClass().getMetadata(),
                deferredImport.getImportSelector());
    }
    return this.group.selectImports();
}
```

这就是这两个方法的调用地方。
上述就是 EnableAutoConfiguration 为 key 自动配置类的收集过程。有自动配置类的收集并加入到 spring 容器，前面提到的 aop，事务，缓存，mvc 功能就已经导入到 springboot 工程了。

21、AOP 功能的自动配置类

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
```

这个就是 AOP 功能的自动配置类，我们在讲 spring 源码的时候知道，如果我们要开启 aop 功能，必须要在 xml 里面加配置或者用注解的方式手动开启 aop 功能、注解方式开启 aop 功能如图：

```
/*
* @Service会被@ComponentScan扫描到
* */
@Service
/*
* 开启注解AOP
* 替代了: <aop:aspectj-autoproxy/>
* */
@EnableAspectJAutoProxy(proxyTargetClass = false,exposeProxy = true)
public class EnableAspectJAutoProxyBean {

}
```

我们读源码的时候就知道，手动开启 aop 功能的这个注解其实就是往 spring 容器里面加入了一个支持 aop 功能的入口类。

我们看看 springboot 中 aop 的自动配置类

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnProperty(prefix = "spring.aop", name = "auto", havingValue = "true", matchIfMissing = true)
public class AopAutoConfiguration {

    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(Advice.class)
    static class AspectJAutoProxyingConfiguration {

        @Configuration(proxyBeanMethods = false)
        @EnableAspectJAutoProxy(proxyTargetClass = false)
        @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "false",
                matchIfMissing = false)
        static class JdkDynamicAutoProxyConfiguration {

        }

        @Configuration(proxyBeanMethods = false)
        @EnableAspectJAutoProxy(proxyTargetClass = true)
        @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "true",
                matchIfMissing = true)
        static class CglibAutoProxyConfiguration {

        }

    }
```

看到这个代码其实很明显，默认是开启 aop 自动配置功能的
默认是开启了 cglib 的动态代理功能的，并且也加上了
`@EnableAspectJAutoProxy(proxyTargetClass = false)`
注解了，加上这个注解就会开启 aop 功能。

22、Condition 功能的实现

Condition 功能的使用很简单
使用的需求是：有的时候我们需要当某一个条件满足的时候才把一些类实例化并加入到 spring 容器中。

当 spring 容器中存在 jack 这个 bean 时，才调用该方法

```
@Bean
@ConditionalOnBean(name = "jack")
public Famliy getFamliy(People people) {
    JackFamliy jackFamliy = new JackFamliy();
    jackFamliy.setPeople(people);
    people.toString();
    return jackFamliy;
}
```

当工程上下文中存在这个类是才调用该方法

```
/*
* 当类路径上存在该类时才会实例化Role
* 类路径存在该类时，才会调用该方法
* */
@Bean
@ConditionalOnClass(name = "com.xiangxue.jack.controller.JackController1")
public Role getRole() {
    System.out.println("======ConfigFamliy.getRole====");
    return new Role();
}
```

这两个跟前面两个刚刚相反

```
/*
* 当类路径不存在该类是，调用方法
* */
@Bean
@ConditionalOnMissingClass(value = "com.xiangxue.jack.controller.JackController1")
public ZgGoods zgGoods() {
    System.out.println("======ConfigFamliy.zgGoods====");
    return new ZgGoods();
}

/*
* 当不存在某对象时调用该方法
* */
@Bean
@ConditionalOnMissingBean(name = "jack")
public User getUser() {
    System.out.println("======ConfigFamliy.getUser====");
    return new User();
}
```

当表达式成立时调用该方法

```java
/*
* 当表达式成立时调用该方法
* */
@Bean
@ConditionalOnExpression("${spring.datasource.max-idle}==10")
public void conditionalOnExpressionTest() {
    System.out.println("=======ConfigFamliy.conditionalOnExpressionTest");
}

/*
```

当配置文件的值相等时调用该方法

```java
/*
* 当存在配置时调用该方法
* */
@Bean
@ConditionalOnProperty(prefix = "spring.redis", name = "host", havingValue = "192.168.67.139")
public void conditionalOnPropertyTest() {
    System.out.println("======ConfigFamliy.conditionalOnPropertyTest");
}
```

自定义 condition

1、实现 Condition 接口

```java
public class CustomCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        System.out.println("=====CustomCondition.matches======");
        String property = context.getEnvironment().getProperty("spring.redis.jedis.pool.max-active");

        if("8".equals(property)) {
            return false;
        } else {
            return true;
        }
    }
}
```

2、自定义 condition 的使用

```java
@Bean
@Conditional(value = CustomCondition.class)
public User conditionalTest() {
    System.out.println("=======ConfigFamliy.conditionalTest");
    return new User();
}
```

当自定义 CustomCondition 中的 matches 方法返回 true 时才会调用到该方法。

@Conditional 注解作用在类上面也是一样的使用方式

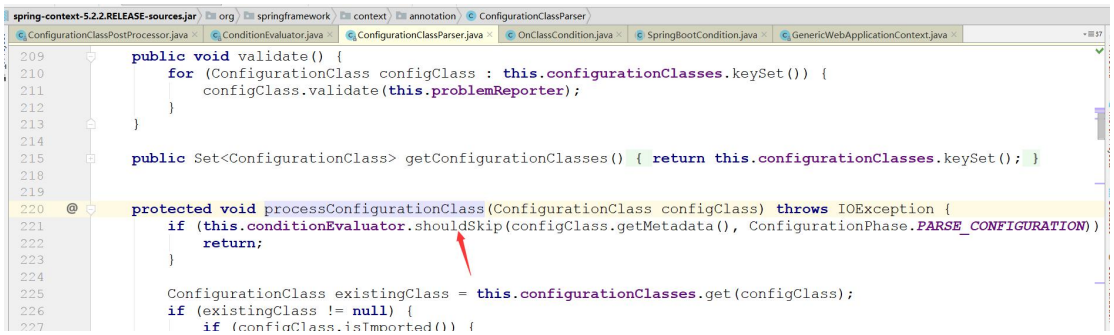
```java
@Component
@Conditional(value = CustomCondition.class)
public class CustomConditionBean {
}
```

Condition 的原理和源码

从 condition 的使用需求我们知道，这个是单条件满足的时候才实例化 bean 和加入到 spring 容器，而在 spring 中一个类的实例化必须要变成 beanDefinition 对象。而 ConfigurationClassPostProcessor 是所有 beanDefinition 对象的集

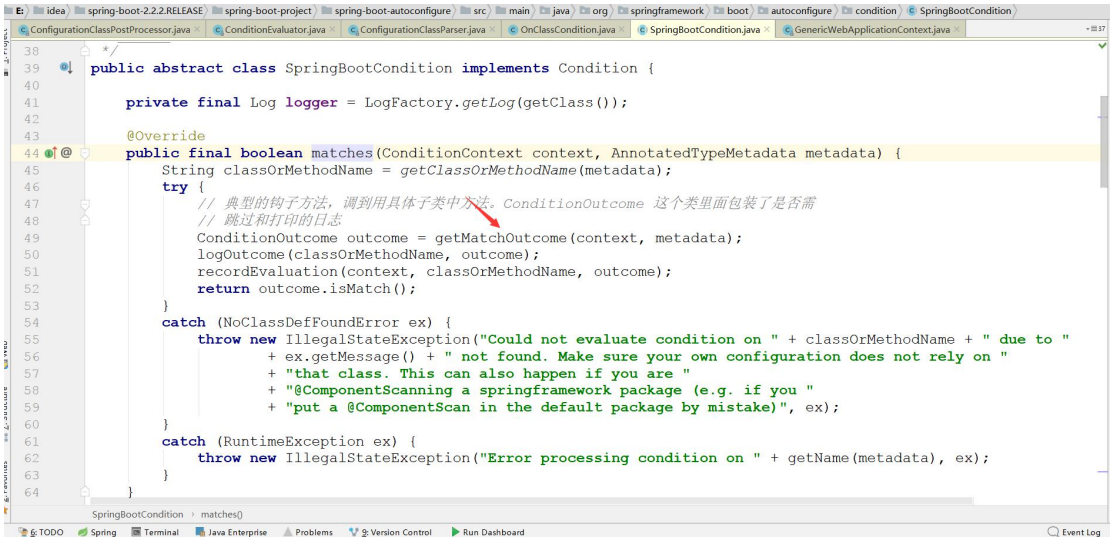
散地，所有的 beanDefinition 都会在这个类里面处理。那么我们要完成 Condition 功能也必定在这个类里面。

ConfigurationClassPostProcessor 类中的 shouldSkip 方法就是做 bean 过滤的。



```
public void validate() {
    for (ConfigurationClass configClass : this.configurationClasses.keySet()) {
        configClass.validate(this.problemReporter);
    }
}

public Set<ConfigurationClass> getConfigurationClasses() { return this.configurationClasses.keySet(); }

protected void processConfigurationClass(ConfigurationClass configClass) throws IOException {
    if (this.conditionEvaluator.shouldSkip(configClass.getMetadata(), ConfigurationPhase.PARSE_CONFIGURATION)) {
        return;
    }

    ConfigurationClass existingClass = this.configurationClasses.get(configClass);
    if (existingClass != null) {
        if (configClass.isImported()) {
```

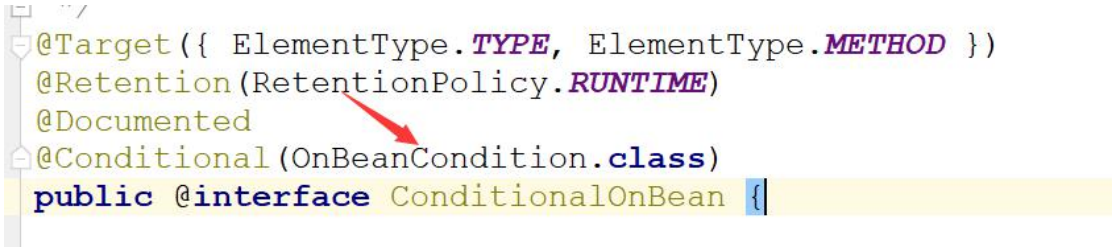
```
for (Condition condition : conditions) {
    ConfigurationPhase requiredPhase = null;
    if (condition instanceof ConfigurationCondition) {
        requiredPhase = ((ConfigurationCondition) condition).getConfigurationPhase();
    }
    if ((requiredPhase == null || requiredPhase == phase) && !condition.matches(this.context, metadata)) {
        return true;
    }
}
```



```
*/
public abstract class SpringBootCondition implements Condition {

    private final Log logger = LogFactory.getLog(getClass());

    @Override
    public final boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        String classOrMethodName = getClassOrMethodName(metadata);
        try {
            // 典型的钩子方法，调到用具体子类中方法。ConditionOutcome 这个类里面包装了是否需
            // 跳过和打印的日志
            ConditionOutcome outcome = getMatchOutcome(context, metadata);
            logOutcome(classOrMethodName, outcome);
            recordEvaluation(context, classOrMethodName, outcome);
            return outcome.isMatch();
        }
        catch (NoClassDefFoundError ex) {
            throw new IllegalStateException("Could not evaluate condition on " + classOrMethodName + " due to "
                    + ex.getMessage() + " not found. Make sure your own configuration does not rely on "
                    + "that class. This can also happen if you are "
                    + "@ComponentScanning a springframework package (e.g. if you "
                    + "put a @ComponentScan in the default package by mistake)", ex);
        }
        catch (RuntimeException ex) {
            throw new IllegalStateException("Error processing condition on " + getName(metadata), ex);
        }
    }
```

这个 getMatchOutcome 是一个钩子方法，不同的注解调用的实现类不一样。这里看两个注解的实现

1



```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(OnBeanCondition.class)
public @interface ConditionalOnBean {
```

Bean 存在时才掉用方法，这个其实很好理解，判断 bean 是否存在其实就只要从 BeanFactory 中找就行了，源码里面就是从 BeanFactory 中找。

```java
public ConditionOutcome getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata) {
    ConditionMessage matchMessage = ConditionMessage.empty();
    // 获取类上面的注解
    MergedAnnotations annotations = metadata.getAnnotations();
    if (annotations.isPresent(ConditionalOnBean.class)) {
        // 包装一下
        Spec<ConditionalOnBean> spec = new Spec<>(context, metadata, annotations, ConditionalOnBean.class);
        // 核心方法，其实就是判断spring容器中是否有ConditionalOnBean注解中的bean
        MatchResult matchResult = getMatchingBeans(context, spec);
        if (!matchResult.isAllMatched()) {
            String reason = createOnBeanNoMatchReason(matchResult);
            return ConditionOutcome.noMatch(spec.message().because(reason));
        }
        matchMessage = spec.message(matchMessage).found( singular: "bean",  plural: "beans").items(Style.QUOTE,
                matchResult.getNamesOfAllMatches());
    }
```

```java
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,
                message: "Unable to use SearchStrategy.ANCESTORS");
        beanFactory = (ConfigurableListableBeanFactory) parent;
    }
    MatchResult result = new MatchResult();
    Set<String> beansIgnoredByType = getNamesOfBeansIgnoredByType(classLoader, beanFactory, considerHierarchy,
            spec.getIgnoredTypes(), parameterizedContainers);
    for (String type : spec.getTypes()) {
        // 这里明显是根据注解中的类型从spring容器中获取bean，如果能获取到说明是匹配的
        Collection<String> typeMatches = getBeanNamesForType(classLoader, considerHierarchy, beanFactory, type,
                parameterizedContainers);
        typeMatches.removeAll(beansIgnoredByType);
        if (typeMatches.isEmpty()) {
            result.recordUnmatchedType(type);
        }
        else {
            result.recordMatchedType(type, typeMatches);
        }
    }
    for (String annotation : spec.getAnnotations()) {
```

从 BeanFactory 中获取对应的实例，如果有则匹配。

2

```java
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(OnClassCondition.class)
public @interface ConditionalOnClass {
```

当工程上下文中存在该类时才调用方法
实现原理，就是通过反射的方式，如果反射有异常则返回 false，如果反射没异常返回 true

```java
    }

    @Override
    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) {
            // 核心方法，过滤一下，其实就是Class.forname一下ConditionalOnClass注解中的类，如果有异常就说明
            // 上下文中没这个类，没有就过滤掉
            List<String> missing = filter(onClasses, ClassNameFilter.MISSING, classLoader);
            if (!missing.isEmpty()) {
                return ConditionOutcome.noMatch(ConditionMessage.forCondition(ConditionalOnClass.class)
                        .didNotFind( singular: "required class",  plural: "required classes").items(Style.QUOTE, missing));
            }
            matchMessage = matchMessage.andCondition(ConditionalOnClass.class)
                    .found( singular: "required class",  plural: "required classes")
                    .items(Style.QUOTE, filter(onClasses, ClassNameFilter.PRESENT, classLoader));
        }
        List<String> onMissingClasses = getCandidates(metadata, ConditionalOnMissingClass.class);
        if (onMissingClasses != null) {
```

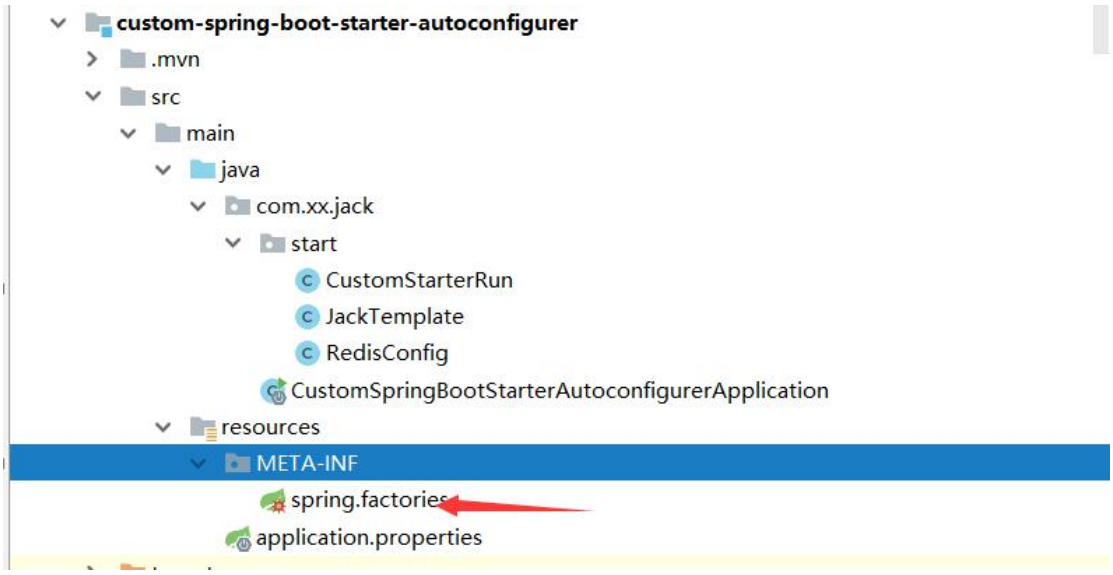
```
    MISSING {

        @Override
        public boolean matches(String className, ClassLoader classLoader) {
            return !isPresent(className, classLoader);
        }

    };

    abstract boolean matches(String className, ClassLoader classLoader);

    static boolean isPresent(String className, ClassLoader classLoader) {
        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 classLoader.loadClass(className);
    }
    return Class.forName(className);
}
```

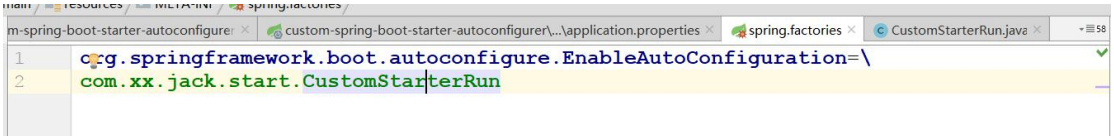
通过 Class.forName 反射，有异常返回 false，没异常返回 true。

23、自定义启动器

当公司里面需要把一些共用的 api 封装成 jar 包的时候，就可以尝试自定义启动器来做。
自定义启动器用到的就是 springboot 中的 SPI 原理，springboot 会去加载 META-INF/spring.factories 配置文件。加载 EnableAutoConfiguration 为 key 的所有类。
利用这一点，我们也可以定义一个工程也会有这个文件。
1、定义启动器核心工程
工程结构



spring.factories 配置内容

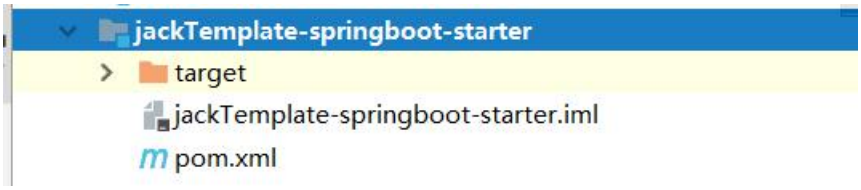


被 springboot SPI 加载的类

```
@Configuration
@ConditionalOnClass(JackTemplate.class)
@EnableConfigurationProperties(RedisConfig.class)
public class CustomStarterRun {

    @Autowired
    private RedisConfig redisConfig;

    @Bean
    public JackTemplate jackTemplate() {
        JackTemplate jackTemplate = new JackTemplate(redisConfig);
        return jackTemplate;
    }
}
```

这个 JackTemplate 实例就是我们封装的通用 API，其他工程可以直接导入 jar 使用的。

2、自定义 starter
我们还会定义一个没代码的工程，在这个工程里面没有任何代码，只有一个 pom 文件。Pom 里面就是对前面核心工程 jar 包的导入
工程结构：



Pom 文件内容：

```
  <dependencies>
    <dependency>
      <groupId>com.xx.jack</groupId>
      <artifactId>custom-spring-boot-starter-autoconfigurer</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```

3、自定义启动器使用
其实就只有在 springboot 工程 pom 文件里面导入依赖就可以了

```
<dependency>
    <groupId>com.xx.jack</groupId>
    <artifactId>jackTemplate-spring-boot-starter</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

这个依赖就是自定义 starter 哪个工程的 maven 坐标。

24、Redis 自动配置

自动配置类
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration
为什么可以在程序里面直接依赖注入 RedisTemplate，在自动配置类创建了该实例

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(RedisOperations.class)
@EnableConfigurationProperties(RedisProperties.class)
@Import({ LettuceConnectionConfiguration.class, JedisConnectionConfiguration.class })
public class RedisAutoConfiguration {

    @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.setConnectionFactory(redisConnectionFactory);
        return template;
    }
```

25、数据源自动配置

自动配置类
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
在 springboot 工程中，在不配置数据源对象的情况下，默认是有数据源对象的。

```
    /**
     * Hikari DataSource configuration.
     */
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(HikariDataSource.class)
    @ConditionalOnMissingBean(DataSource.class)
    @ConditionalOnProperty(name = "spring.datasource.type", havingValue = "com.zaxxer.hikari.HikariDa
            matchIfMissing = true)
    static class Hikari {

        @Bean
        @ConfigurationProperties(prefix = "spring.datasource.hikari")
        HikariDataSource dataSource(DataSourceProperties properties) {
            HikariDataSource dataSource = createDataSource(properties, HikariDataSource.class);
            if (StringUtils.hasText(properties.getName())) {
                dataSource.setPoolName(properties.getName());
            }
            return dataSource;
        }

    }
```

默认是有 Hikari 数据源对象的。

26、JdbcTemplate 自动配置

org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(JdbcOperations.class)
class JdbcTemplateConfiguration {

    @Bean
    @Primary
    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;
    }

}
```

27、事务管理器自动配置

org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration

```
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnSingleCandidate(DataSource.class)
    static class DataSourceTransactionManagerConfiguration {

        @Bean
        @ConditionalOnMissingBean(PlatformTransactionManager.class)
        DataSourceTransactionManager transactionManager(DataSource dataSource,
                ObjectProvider<TransactionManagerCustomizers> transactionManagerCustomizers) {
            DataSourceTransactionManager transactionManager = new DataSourceTransactionManager(dataSou
            transactionManagerCustomizers.ifAvailable((customizers) -> customizers.customize(transactio
            return transactionManager;
        }

    }
```

28、事务自动配置

org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration

编程式事务对象

```
@Bean
@ConditionalOnMissingBean(TransactionOperations.class)
public TransactionTemplate transactionTemplate(PlatformTransactionManager transactionManager)
    return new TransactionTemplate(transactionManager);
}
```

事务功能导入注解

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnBean(TransactionManager.class)
@ConditionalOnMissingBean(AbstractTransactionManagementConfiguration.class)
public static class EnableTransactionManagementConfiguration {

    @Configuration(proxyBeanMethods = false)
    @EnableTransactionManagement(proxyTargetClass = false)
    @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "false",
            matchIfMissing = false)
    public static class JdkDynamicAutoProxyConfiguration {

    }

    @Configuration(proxyBeanMethods = false)
    @EnableTransactionManagement(proxyTargetClass = true)
    @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "true",
            matchIfMissing = true)
    public static class CglibAutoProxyConfiguration {

    }

}
```

29、DispatcherServlet 自动配置

org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration

```
@Bean(name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
public DispatcherServlet dispatcherServlet(HttpProperties httpProperties, WebMvcProperties webMvcProperties
    DispatcherServlet dispatcherServlet = new DispatcherServlet();
    dispatcherServlet.setDispatchOptionsRequest(webMvcProperties.isDispatchOptionsRequest());
    dispatcherServlet.setDispatchTraceRequest(webMvcProperties.isDispatchTraceRequest());
    dispatcherServlet.setThrowExceptionIfNoHandlerFound(webMvcProperties.isThrowExceptionIfNoHandlerFound()
    dispatcherServlet.setPublishEvents(webMvcProperties.isPublishRequestHandledEvents());
    dispatcherServlet.setEnableLoggingRequestDetails(httpProperties.isLogRequestDetails());
    return dispatcherServlet;
}
```

```
@Bean(name = DEFAULT_DISPATCHER_SERVLET_REGISTRATION_BEAN_NAME)
@ConditionalOnBean(value = DispatcherServlet.class, name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
public DispatcherServletRegistrationBean dispatcherServletRegistration(DispatcherServlet dispatcherServlet,
        WebMvcProperties webMvcProperties, ObjectProvider<MultipartConfigElement> multipartConfig) {
    DispatcherServletRegistrationBean registration = new DispatcherServletRegistrationBean(dispatcherServle
            webMvcProperties.getServlet().getPath());
    registration.setName(DEFAULT_DISPATCHER_SERVLET_BEAN_NAME);
    registration.setLoadOnStartup(webMvcProperties.getServlet().getLoadOnStartup());
    multipartConfig.ifAvailable(registration::setMultipartConfig);
    return registration;
}
```

30、MVC 自动配置

org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration

其实 mvc 的自动配置就是把一些支持 mvc 功能的类创建出来

比如：handlerMapping，HandlerAdapter，ViewResolver 等等实例

```
@Bean
@Override
public RequestMappingHandlerAdapter requestMappingHandlerAdapter(
        @Qualifier("mvcContentNegotiationManager") ContentNegotiationManager contentNe
        @Qualifier("mvcConversionService") FormattingConversionService conversionServi
        @Qualifier("mvcValidator") Validator validator) {
    RequestMappingHandlerAdapter adapter = super.requestMappingHandlerAdapter(contentN
            conversionService, validator);
    adapter.setIgnoreDefaultModelOnRedirect(
            this.mvcProperties == null || this.mvcProperties.isIgnoreDefaultModelOnRec
    return adapter;
}
```

```java
@Bean
@Primary
@Override
public RequestMappingHandlerMapping requestMappingHandlerMapping(
        @Qualifier("mvcContentNegotiationManager") ContentNegotiationManager contentNegotia
        @Qualifier("mvcConversionService") FormattingConversionService conversionService,
        @Qualifier("mvcResourceUrlProvider") ResourceUrlProvider resourceUrlProvider) {
    // Must be @Primary for MvcUriComponentsBuilder to work
    return super.requestMappingHandlerMapping(contentNegotiationManager, conversionService,
            resourceUrlProvider);
}


@Bean
@ConditionalOnMissingBean
public InternalResourceViewResolver defaultViewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setPrefix(this.mvcProperties.getView().getPrefix());
    resolver.setSuffix(this.mvcProperties.getView().getSuffix());
    return resolver;
}

@Bean
@Override
public FormattingConversionService mvcConversionService() {
    WebConversionService conversionService = new WebConversionService(this.mvcPropert
    addFormatters(conversionService);
    return conversionService;
}
```