1、推出springboot的初衷是为了简化spring的配置，使得开发中集成新功能时更快。

1、SpringBoot图标是通过SpringBootScanner打印的，是在SpringApplication.run()中调起的：

```
                applicationArguments);
        configureIgnoreBeanInfo(environment);
        Banner printedBanner = printBanner(environment);
        context = createApplicationContext();
*/
class SpringBootBanner implements Banner {

    private static final String[] BANNER = { "",
            "  .   ____          _            __ _ _",
            " /\\\\ / ___'_ __ _ _(_)_ __  __ _ \\ \\ \\ \\",
            "( ( )\\___ | '_ | '_| | '_ \\/ _` | \\ \\ \\ \\",
            " \\\\/  ___)| |_)| | | | | || (_| |  ) ) ) )",
            "  '  |____| .__|_| |_|_| |_\\__, | / / / /",
            " =========|_|==============|___/=/_/_/_/" };
```

1、替换Banner只需要在resource下边加一个banner.txt就可以，图标样式可以找其他三方工具生成。也可以将其替换成图片。

2、@EnableAutoConfiguration注解的作用是引入第三方组件（即starter）。如果用@EnableAutoConfiguration注解的话那我们还需要额外使用@ComponentScan注解去扫其他包下bean。

3、@SpringBootApplication注解引入了@EnableAutoConfiguration注解，并且它还引入@ComponentScan注解去扫描启动类所在包以及它的子包下的bean（@ComponentScan注解在spring第一课的笔记中有详细讲解）：

```
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

SpringBootApplication注解中引入了@EnableAutoConfiguration注解和@ComponentScan注解

4、一般项目中@SpringBootApplication注解和@EnableAutoConfiguration注解都会用到，因为我们需要排除一些自动加载的组件：

```
@SpringBootApplication

@EnableAutoConfiguration(exclude = { DataSourceAutoConfiguration.class })
```

启动时排除DataSourceAutoConfiguration组件

5、server.servlet.context-path配置的作用：

定义： server.servlet.context-path= # Context path of the application. 应用的上下文路径，也可以称为项目路径，是构成url地址的一部分。

server.servlet.context-path不配置时，默认为 / ，如：localhost:8080/xxxxxx

当server.servlet.context-path有配置时，比如 /demo，此时的访问方式为localhost:8080/demo/xxxxxx

6、事务支持：方法加@Transactional注解。（2.0以前的版本需要在启动类加@EnableTransactionalManagement注解，2.0以后的不需要）

7、统一异常处理：@ControllerAdvice和@ExceptionHandler搭配使用，当将异常抛到controller时,可以对异常进行统一处理,规定返回的json格式或是跳转到一个错误页面：

```
@ControllerAdvice                                   对Controller中的异常进行统一处理
public class GlobalExceptionHandler {

    @ExceptionHandler(value = RuntimeException.class)   拦截RuntimeException，此处也可以改成Exception
    @ResponseBody
    public Object defaultErrorHandler(HttpServletRequest req, Exception e) throws Exception {
        e.printStackTrace();
        return "我是个异常处理类";
    }
```

8、静态资源访问：静态资源是指js, css, html, 图片，音视频等，Spring Boot默认提供静态资源目录位置需置于classpath下，目录名需符合如下规则：/static、/public、/resources、/META-INF/resources。例如在resources目录下面建立static文件夹，在文件夹里面任意放张图片。命名为：enjoy.jpg，在地址栏上输入localhost:8080/enjoy.jpg，可以看到图片。

9、springboot中不建议使用jsp，而是推荐使用模板引擎thymeleaf来渲染html。

10、springboot集成Swagger2用来生成api文档。

11、在springboot中默认使用的日志工具是logback，默认日志级别为info。

12、logging.path 和logging.file都配置了，只会有logging.file生效，所以，如果要指定日志生成的具体位置使用logging.file 配置就好。

13、Spring Boot 热加载/部署：spring-boot-devtools 是一个为开发者服务的一个模块，其中最重要的功能就是自动应用代码更改到最新的App上面去。原理是在发现代码有更改之后，重新启动应用，但是速度比手动停止后再启动还要更快，更快指的不是节省出来的手工操作的时间。其深层原理是使用了两个ClassLoader，一个Classloader加载那些不

会改变的类（第三方Jar包），另一个ClassLoader加载会更改的类，称为restart ClassLoader ,这样在有代码更改的时候，原来的restart ClassLoader 被丢弃，重新创建一个restart ClassLoader，由于需要加载的类相比较少，所以实现了较快的重启时间。

14、Actuator监控管理，缺点是没有可视化界面。通过actuator/+端点名就可以获取相应的信息（默认只加载了info/health，加载全部要配置management.endpoints.web.exposure.include=*）：

- /actuator/beans：显示应用程序中所有Spring bean的完整列表。
- /actuator/configprops：显示所有配置信息。
- /actuator/env：陈列所有的环境变量。
- /actuator/mappings：显示所有@RequestMapping的url整理列表。
- /actuator/health：显示应用程序运行状况信息 up表示成功 down失败
- /actuator/info：查看自定义应用信息

15、springBoot后台线程是否会自动退出和引入的starter有关系：

- 引入spring-boot-starter-web时，启动成功后后台线程会阻塞，即不会自动退出。具体原因我觉得是spring-boot-starter-web中做了阻塞操作：

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

- 引入spring-boot-starter时，启动成功后后台线程会自动退出，需要我们手动阻塞：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
@SpringBootApplication
public class LjqTestApplication {

    public static void main(String[] args) throws Exception{
        SpringApplication.run(LjqTestApplication.class,args);
        new CountDownLatch(1).await();
    }
```

手动干预阻塞后台线程

16、多数据源与jta+atomikos分布式事务，此处的分布式事务不是指多个应用之间的，而是控制同一个应用的多个数据源的事务。详细配置过程看word。

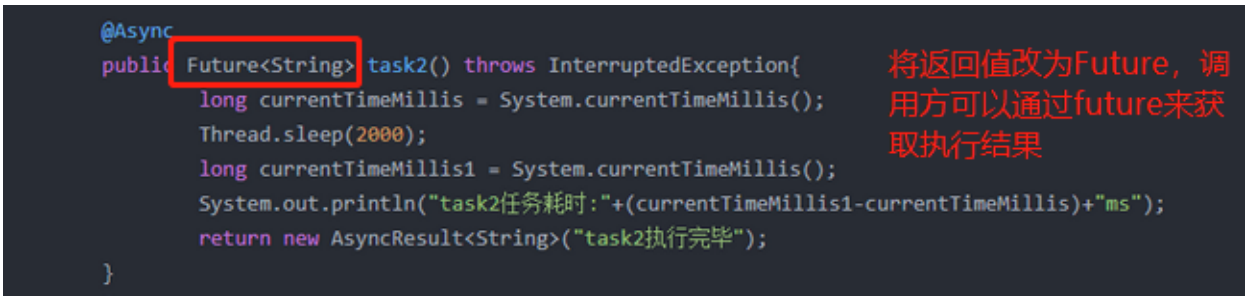
17、spring-boot热部署只需要引入spring-boot-devtools，我的idea版本不需要加maven自动编译插件，自动编译或者手动编译根据word中的配置就行，按需选择。

18、yml文件中注入map、list等的方式
(https://www.jianshu.com/p/f536b10dd9e7)

19、application.yml和bootstrap.yml文件：

- bootstrap.yml：bootstrap.yml 用来程序引导时执行，应用于更加早期配置信息读取。可以理解成系统级别的一些参数配置，这些参数一般是不会变动的。一旦bootStrap.yml 被加载，则内容不会被覆盖。
- application.yml：application.yml 可以用来定义应用级别的， 应用程序特有配置信息，可以用来配置后续各个模块中需使用的公共参数等。
- 两者区别：bootstrap.yml 和 application.yml 都可以用来配置参数，bootstrap.yml 先加载 application.yml后加载，如果加载的 application.yml 的内容标签与 bootstrap 的标签一致，application 也不会覆盖 bootstrap，而 application.yml 里面的内容可以动态替换。
- bootstrap.yml典型应用场景：
  - 当使用 Spring Cloud Config Server 配置中心时，这时需要在 bootstrap.yml 配置文件中指定 spring.application.name 和 spring.cloud.config.server.git.uri，添加连接到配置中心的配置属性来加载外部配置中心的配置信息；
  - 一些固定的不能被覆盖的属性；
  - 一些加密/解密的场景；

20、SpringBoot使用@Async实现异步调用（自定义线程池 https://www.jianshu.com/p/a13ac0d774c6），如果想要获取异步执行结果则可以将异步执行的方法返回值改为Future，如下图：



21、Spring Factories详解（SPI）：https://www.jianshu.com/p/00e49c607fa1。

----------自定义starter----------

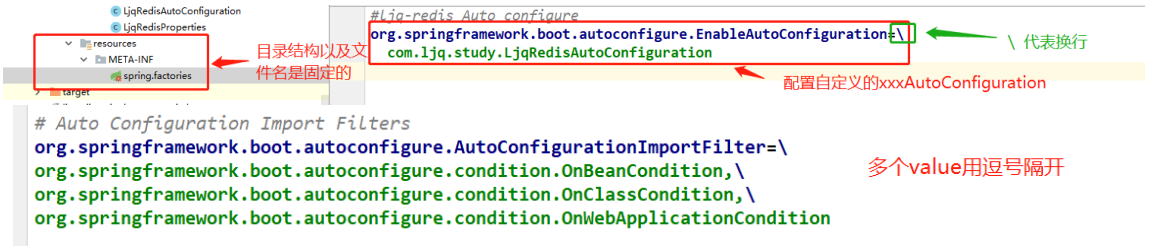
关于SpringBoot中starter的思考：

1. 为什么要定义starter？------>所谓starter就是SpringBoot为我们提供的对一些组件的封装（例如mybatis-starter、redis-starter等），这些starter能非常方便快捷的增加功能（当我们需要使用redis的时候引入redis-starter依赖，并在配置文件中配置host、port等参数就ok），配置很简便（只需要在application.properties中稍微配置下就可以使用了）。
2. 以使用redis为例，通过starter配置的优秀之处？------->当我们项目中不通过starter集成redis时，我们需要将JedisPool实例手动加载到容器中或者通过工具类提供单例的JedisPool对象供程序员使用，这样在微服务架构下每个项目都需要做这些不必要的工作。因为我们可以自定义一个redis-starter，通过这个starter将JedisPool实例加载到容器中，然后将它打成jar包，各个项目引用这个starter依赖就直接可以使用了（这也是springboot的初衷，springboot替我们做了这些工作，然后项目中直接引starter就可以了）。
3. 自定义redis-starter时的感悟：其实整个starter最终只做了了一件事，就是将JedisPool对象加载到容器中，然后我们在项目中可以通过@Resource、@Autowired等注解引用这个对象去操作redis。（但是starter的功能不止于此，我们可以在starter项目中扩展很多功能【即加很多工具类】供引用者使用）。

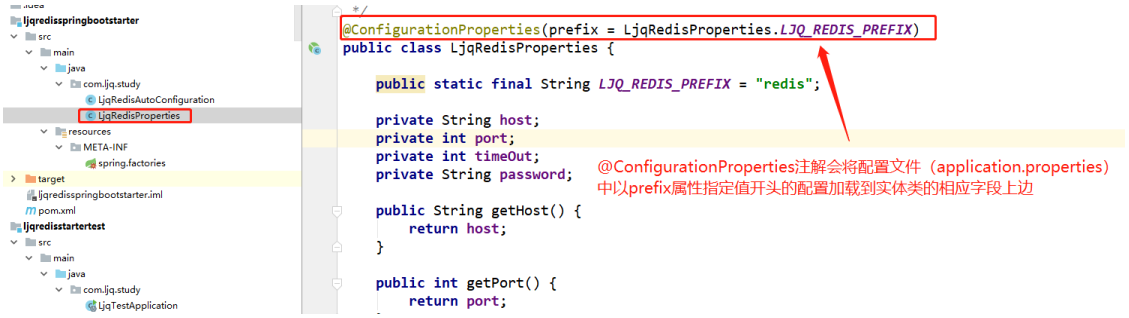
自定义startter分为3三步（SPI设计思想，可以将它称为springboot spi）：

1. 在src/main/resources目录下创建META-INF目录，并在目录内添加文件spring.factories，spring.factories文件的配置结构是key--->value形式，多个

value时用逗号隔开，如下图：



2. 创建一个xxxProperties用于加载自定义starter所需要的配置：



3. 创建一个配置类，这个配置类用于加载配置，并实例化相应的客户端（例如自定义redis-starter时，这个配置类可以实例化Jedis客户端或者JedisPool）：


```
/**
 *该属性为true时，配置文件中缺少对应的value或name的对应的属性值，也会注入成功
 */
boolean matchIfMissing() default false;
```

4. 自定义starter可参考自己写的ljq-redis-spring-boot-starter或者参考lision老师写的redis-starter。

5. @EnableConfigurationProperties通过EnableConfigurationPropertiesImportSelector将配置类加载到容器中（https://www.jianshu.com/p/7f54da1cb2eb）：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(EnableConfigurationPropertiesImportSelector.class)
public @interface EnableConfigurationProperties {
```

starter自动化运作原理：

- **@SpringBootApplication引入了开启自动化配置的注解@EnableAutoConfiguration：**

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

@SpringBootApplication注解通过引入@EnableAutoConfiguration注解来实现自动加载starter

- **@EnableAutoConfiguration注解内使用@import注解通过自定义的ImportSelector即EnableAutoConfigurationImportSelector将starter加载到容器中（关于@Import注解以及自定义ImportSelector的使用可查看spring-test工程）：**

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

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    /**
```

通过它将各种autoConfiguration加载到容器中

- **EnableAutoConfigurationImportSelector内部则是使用了SpringFactoriesLoader.loadFactoryNames方法进行扫描具有META-INF/spring.factories文件的jar包：**

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
        AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
            getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    Assert.notEmpty(configurations,
            message: "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;
}
```

扫描具有META-INF\spring.factories文件的jar包

----------springboot中的注解----------

1、SpringBoot提供的这些注解我们在项目中也可以使用（可以用在类上边也可以用在方法上边），前提是我们得为属性提供set方法，例如使用@ConfigurationProperties：

```java
@Configuration
public class DataSourceConfiguration {
    @Bean
    @ConfigurationProperties(prefix = "spring.datasource")
    public DataSource dataSource() {
        DruidDataSource druidDataSource = new DruidDataSource();
        return druidDataSource;
    }
```

配置德鲁伊数据源

2、关于SpringBoot中的条件注解（以Conditional开头的）：@Conditional是Spring4新提供的注解，它的作用是根据某个条件创建特定的Bean，通过实现Condition接口，并重写matches方法来构造判断条件（参考spring-test工程中的使用方式）。总的来说，就是根据特定条件来控制Bean的创建行为，这样我们可以利用这个特性进行一些自动的配置。SpringBoot中的条件注解就是@Conditional注解的扩展注解。

（https://www.cnblogs.com/haha12/p/11304738.html），以@ConditionalOnClass为例如下图：

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

3、SpringBoot中提供的@Conditional注解的扩展注解：

- **@ConditionalOnBean：仅仅在当前上下文中存在某个对象时，才会实例化一个Bean。**
- **@ConditionalOnClass：某个class位于类路径上，才会实例化一个Bean。**
- **@ConditionalOnExpression：当表达式为true的时候，才会实例化一个Bean。**
- **@ConditionalOnMissingBean：仅仅在当前上下文中不存在某个对象时，才会实例化一个Bean。**
- **@ConditionalOnMissingClass：某个class类路径上不存在的时候，才会实例化一个Bean。**
- **@ConditionalOnNotWebApplication：不是web应用，才会实例化一个Bean。**
- **@ConditionalOnBean：当容器中有指定Bean的条件下进行实例化。**
- **@ConditionalOnMissingBean：当容器里没有指定Bean的条件下进行实例化。**

- **@ConditionalOnClass：当classpath类路径下有指定类的条件下进行实例化。**
- **@ConditionalOnMissingClass：当类路径下没有指定类的条件下进行实例化。**
- **@ConditionalOnWebApplication：当项目是一个Web项目时进行实例化。**
- **@ConditionalOnNotWebApplication：当项目不是一个Web项目时进行实例化。**
- **@ConditionalOnProperty：当指定的属性有指定的值时进行实例化。**
- **@ConditionalOnExpression：基于SpEL表达式的条件判断。**
- **@ConditionalOnJava：当JVM版本为指定的版本范围时触发实例化。**
- **@ConditionalOnResource：当类路径下有指定的资源时触发实例化。**
- **@ConditionalOnJndi：在JNDI存在的条件下触发实例化。**
- **@ConditionalOnSingleCandidate：当指定的Bean在容器中只有一个，或者有多个但是指定了首选的Bean时触发实例化。**

----------SpringBoot性能优化----------

1、SpringBoot项目启动慢的原因：一般情况下我们会使用@SpringBootApplication注解来自动获取应用的配置信息，但这样也会带来一些副作用。使用这个注解后，会触发自动配置（auto-configuration）和组件扫描（component scanning），这跟使用@Configuration、@EnableAutoConfiguration和@ComponentScan三个注解的作用一样。这样做给开发带来方便的同时，会有以下的一些影响：

- **会导致项目启动时间变长（原因：扫描了很多被忽略的组件（Negative matches）,这些组件被扫描之后加载失败，白白浪费了cpu资源和内存资源）。当启动一个大的应用程序，或将做大量的集成测试启动应用程序时，影响会特别明显。**
- **会加载一些不需要的多余的实例（beans），虽然有些组件被加载成功了（Positive matches中的一部分），但是这些组件我们的项目中用不到，所以可以将它排除。**
- **会增加CPU消耗和内存的占用。**

2、当我们将项目日志级别调成debug启动项目时会在控制台看到以下内容：

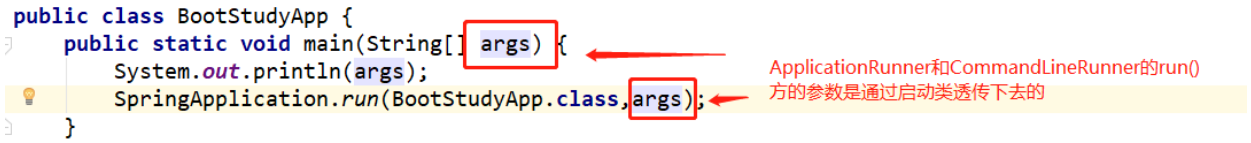
3、扫描优化：根据上面的理论知识，我们只需要在启动的时候通过@Import注解显示的引入Positive matches、Unconditional classes中的组件，这样就避免了无用的扫描工作。

4、容器优化：默认情况下，SpringBoot使用Tomcat来作为内嵌的Servlet容器，可以将用Undertow替换Tomcat（在starter中排除tomcat依赖，然后另外引入Undertow依赖）。

5、JVM参数调优。

----------可扩展接口----------

1、ApplicationRunner、CommandLineRunner这两个接口的作用几乎是一样的，只不过它们run()方法的参数不一样而已，run()方法的参数都是通过启动类透传下去的：



2、ApplicationRunner的run方法形参为ApplicationArguments（它对main方法的args数组进行了封装解析），同时我们还可以以键值对的方式配置main方法参数，当配置键值对参数的时候需要在键前面加--，ApplicationArguments提供了相应的方法去获取键、值、原生参数（详细看代码注释，注释很详细），因此ApplicationRunner相对于

CommandLineRunner在参数处理这块功能比较丰富一点，CommandLIneRunner只不过原封不动的将args参数传过来，没做特殊处理：

3、ApplicationContextInitializer是在容器实例化之后、刷新之前调用的（即调用spring的refresh()方法）。ApplicationContextInitializer是Spring框架原有的概念（即这个接口是spring提供的，但是spring中并没有利用这个接口）。ApplicationContextInitializer是Spring留出来允许我们在上下文刷新之前做自定义操作的钩子，这个钩子被SpringBoot发扬光大了：

4、SpringApplicationRunListener 接口的作用主要就是在Spring Boot 启动初始化的过程中可以通过SpringApplicationRunListener接口回调来让用户在启动的各个流程中可以加入自己的逻辑。对于开发者来说，基本没有什么常见的场景要求我们必须实现一个自定义的SpringApplicationRunListener（如果需要实现那必须的提供一个特定的构造方

法，并且需要在spring.factories文件中配置）。因为springboot已经提供了一个SpringApplicationRunListener的实现类EventPublishingRunListener，它里边维护了Spring广播器SimpleApplicationEventMulticaster，在EventPublishingRunListener的各个方法中通过广播器发布了不同的事件，因此我们只需要实现ApplicationListener去监听我们感兴趣的事件而不需要实现SpringApplicationRunListener 的所有方法。

- https://www.jianshu.com/p/b86a7c8b3442
- https://www.cnblogs.com/duanxz/p/11243271.html

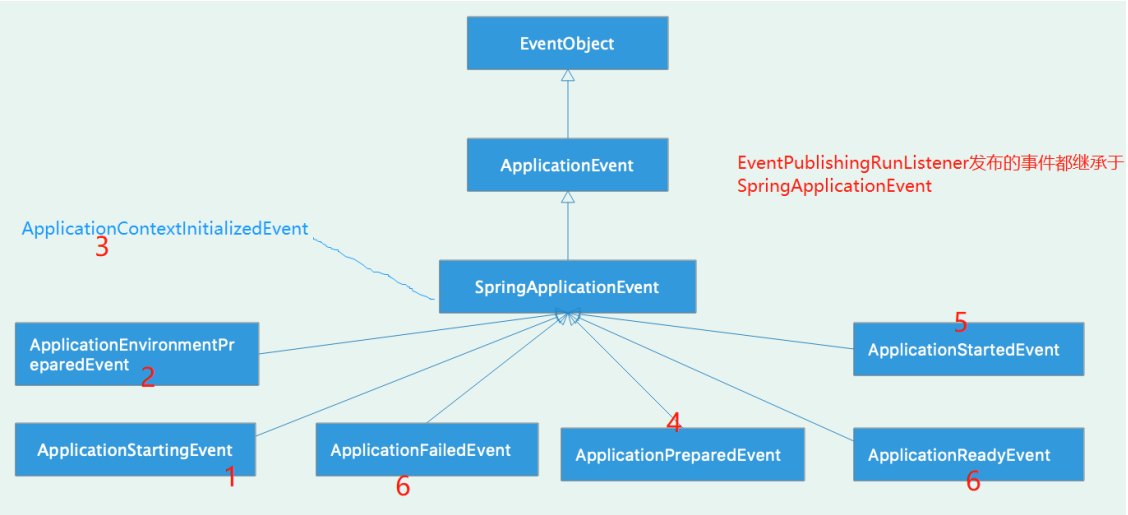
5、直接实现SpringApplicationRunListener接口的方式看word中的例子，需要实现以下方法，一般工作用只需要关心启动过程中的某个点，所以这种方式有点杀鸡用牛刀的感觉，没必要：

```
package org.springframework.boot;
public interface SpringApplicationRunListener {

    // 在run()方法开始执行时，该方法就立即被调用，可用于在初始化最早期时做一些工作
    void starting();
    // 当environment构建完成，ApplicationContext创建之前，该方法被调用
    void environmentPrepared(ConfigurableEnvironment environment);
    // 当ApplicationContext构建完成时，该方法被调用
    void contextPrepared(ConfigurableApplicationContext context);
    // 在ApplicationContext完成加载，但没有被刷新前，该方法被调用
    void contextLoaded(ConfigurableApplicationContext context);
    // 在ApplicationContext刷新并启动后，CommandLineRunners和ApplicationRunner未被调用前，该方法
    void started(ConfigurableApplicationContext context);
    // 在run()方法执行完成前该方法被调用
    void running(ConfigurableApplicationContext context);
    // 当应用运行出错时该方法被调用
    void failed(ConfigurableApplicationContext context, Throwable exception);
}
```

6、通过监听EventPublishingRunListener发布的各个事件来影响我们感兴趣的某个点：

- EventPublishingRunListener在各个点发布的事件如下，具体看源码可知不难：



- **实践举例：如果想在项目刚起动的时候做一些事情则可以写一个ApplicationListener实现类去监听ApplicationStartingEvent。**

7、SpringApplicationRunListener是springboot对spring监听机制的扩展，springboot定义了不同的事件来标识启动过程的各个周期，并且还提供了一个实现类EventPublishingRunListener在启动的不同周期去发布这些事件，开发者则可以去监听这些事件对springboot的整个启动流程进行干预，做一些业务处理。EventPublishingRunListener最终还是通过spring广播器去广播这些事件的，从这而也可看出springboot和spring的完美结合：

```
*/
public class EventPublishingRunListener implements SpringApplicationRunListener, Ordered {

    private final SpringApplication application;

    private final String[] args;

    private final SimpleApplicationEventMulticaster initialMulticaster;

    public EventPublishingRunListener(SpringApplication application, String[] args) {
        this.application = application;
        this.args = args;
        this.initialMulticaster = new SimpleApplicationEventMulticaster();
        for (ApplicationListener<?> listener : application.getListeners()) {
            this.initialMulticaster.addApplicationListener(listener);
        }
    }
```

EventPublishingRunListener提供了一个属性用来维护spring广播器，通过这个广播器去发布事件

8、关于这四类扩展接口如何才能在springboot启动的时候调到的问题：

- **ApplicationRunner和CommandLineRunner是在容器刷新之后调用的，这意味着只需要再实现类加上相应的注解（例如@Componet）或者通过配置xml文件将他们加载到容器中，则容器刷新以后容器中已经存在实现类对象了，所以可以调用到：**

```java
*/
@Component
public class MyCommandLineRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("执行CommandLineRunner接口！！！");
    }
*/
@Component
public class MyApplicationRunner implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("getSourceArgs():" + args.getSourceArgs());
        System.out.println("getOptionNames():" + args.getOptionNames());
```

- **ApplicationContextInitializer是在容器刷新之前调用的，所以要配置在META-INF/spring.factories目录下，以便在创建SpringApplication对象时能扫描到并将实现类对象存储到当前SpringApplication对象的initializers属性中，当调用的时候才能拿到这些对象。当然还有其他方式参考链接博客 https://www.cnblogs.com/duanxz/p/11239291.html：**

```java
*/
@Component
public class MyApplcationContextInitializer implements ApplicationContextInitializer {
    @Override
    public void initialize(ConfigurableApplicationContext applicationContext) {
        System.out.println("回调MyApplcationContextInitializer实现类！！！");
    }
}
```

在这个实现类加@Componet注解只是将它加载容器中，而在启动的时候并不会回调它，因为ApplicationContextInitializer接口的实现类是在容器刷新之前调用的，所以需要额外早META-INF目录配置

resources
META-INF
spring.factories
application.yml

```
cn.ljq.listener.StartingApplicationListener
org.springframework.context.ApplicationContextInitializer=\
cn.ljq.listener.MyApplcationContextInitializer
```

- **SpringApplicationRunListener接口中提供的方法贯穿了整启动流程，有些是在容器刷新之前调用的，有些是在容器刷新之后调用的，它的初始化还需要特定的构造函数（参数用到了SpringApplication对象），因此它的实现类必须通过在META-INF配置的这种方式才能调用到：**

```java
public class MySpringApplicationRunListener implements SpringApplicationRunListener {

    //必须提供此构造方法
    public MySpringApplicationRunListener(SpringApplication application,String args[]){}
```

META-INF
spring.factories
application.yml
test

```
org.springframework.boot.SpringApplicationRunListener=\
cn.ljq.listener.MySpringApplicationRunListener
```

- **一般情况下我们没必要实现SpringApplicationRunListener接口，可以通过监听EventPublishingRunListener发布的事件类参与springboot启动流程，这些在**

上文说过，详细看上文，此处需要说的是监听EventPublishingRunListener发布事件的监听器有些只需注入到容器中就可以，而有些需要在META-INF/spring.factories文件中配置的问题，因为部分事件（ApplicationStartingEvent、ApplicationEnvironmentPreparedEvent、ApplicationContextInitializedEvent、ApplicationPreparedEvent）是在容器刷新之前发布的，所以我们仅仅将监听器注册到容器中springboot启动时不会回调的，所以要想在springboot启动的过程中回调这些事件的监听器那么就需要在META-INF/spring.factories文件中配置以便创建SpringApplication对象时扫描并存储到listeners属性中供调用，而对于（ApplicationStartedEvent、ApplicationReadyEvent）事件是在容器刷新之后调用的，所以只需要将它们的实现类注入到容器就能在springboot启动时调用到。

----------springboot启动流程----------

详细看word中的注释

1、创建SpringApplication对象，创建SpringApplication对象时会去扫描META-INF/spring.factories目录下配置的ApplicationContextInitializer和ApplicationListener并将它们分别存储到当前SpringApplication对象的initializers、listeners属性中：

```
*/
public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args) {
    return new SpringApplication(primarySources).run(args);
}
```

先创建SpringApplication对象　　在调用run()方法

```
*/
@SuppressWarnings({"rawtypes", "unchecked"})
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, message: "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

此处底层是通过SpringFactoriesLoader去扫描META-INF/spring.factories文件中配置的ApplicationContextInitializer和ApplicationListener类并创建相应对象存储到当前SpringApplication对象的initializers、listeners属性中

```
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type, Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = getClassLoader();
    // Use names and ensure unique to protect against duplicates
    Set<String> names = new LinkedHashSet<>(SpringFactoriesLoader.loadFactoryNames(type, classLoader));
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes, classLoader, args, names);
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}
```

```
private List<ApplicationContextInitializer<?>> initializers;
```

存储META-INF/spring.factories文件中配置的ApplicationContextInitializer接口实现类的对象

```
private List<ApplicationListener<?>> listeners;
```

存储META-INF/spring.factories文件中配置的ApplicationListener接口的实现类的对象

2、执行run方法：

----------MVC注解----------

@RequestParam和

@PathVariable（https://blog.csdn.net/a15028596338/article/details/84976223）