# 1. spring boot 基础回顾

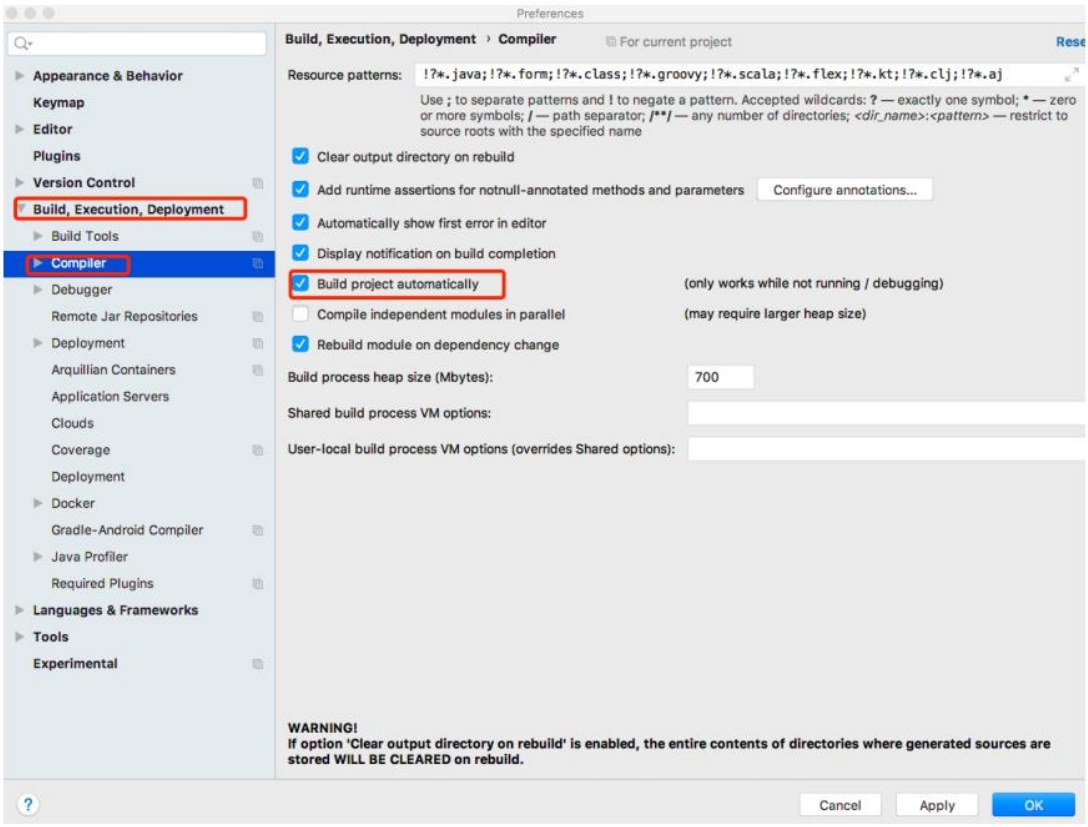
## spring-boot-devtools 项目热部署

Pom 文件中添加依赖

```
<--引入热部署依赖--!>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
</dependency>
```

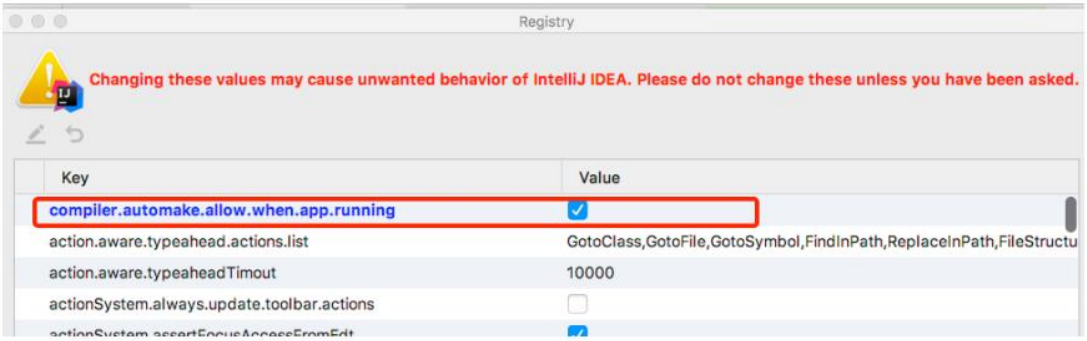
## Idea 上热部署设置

选择IDEA工具界面的【File】->【Settings】选项，打开Compiler面板设置页面




选择Build下的Compiler选项，在右侧勾选“Build project automatically”选项将项目设置为自动编译，单击【Apply】→【OK】按钮保存设置

在项目任意页面中使用组合快捷键“Ctrl+Shift+Alt+/”打开Maintenance选项框，选中并打开Registry页面，具体如图1-17所示




列表中找到“compiler.automake.allow.when.app.running”，将该选项后的Value值勾选，用于指定IDEA工具在程序运行过程中自动编译，最后单击【Close】按钮完成设置

# Spring boo 全局配置文件

application.properties
application.yaml
当 properties 和 yaml 同时配置时，优先加载 properties 配置文件。

YAML 扩展名为 yaml 或者 yml 都可以。

加载顺序为 yml,yaml,properties

```
<build>
  <resources>
    <resource>
      <filtering>true</filtering>
      <directory>${basedir}/src/main/resources</directory>
      <includes>
        <include>**/application*.yml</include>
        <include>**/application*.yaml</include>
        <include>**/application*.properties</include>
      </includes>
    </resource>
    <resource>
```

# 配置文件属性值注入

如果配置属性是 spring boot 已有属性，则 spring boot 内部会自动扫描并读取配置文件中的配置属性，用于覆盖默认值。

# 使用@ConfigurationProperties 注入属性

将配置文件中的属性批量注入到 Bean 对象的对应属性中。

```
@Component
@ConfigurationProperties(prefix = "person") //将配置文件中以 person 开头的属性，注入到该类中
public class Person {
private int id;
 public void setId(int id) {
this.id = id;
}
}
```

### 1.6.1 使用@ConfigurationProperties注入属性

Spring Boot提供的@ConfigurationProperties注解用来快速、方便地将配置文件中的自定义属性值批量注入到某个Bean对象的多个对应属性中。假设现在有一个配置文件，如果使用@ConfigurationProperties注入配置文件的属性，示例代码如下：

```
@Component
@ConfigurationProperties(prefix = "person")
public class Person {
    private int id;
    // 属性的setXX()方法
    public void setId(int id) {
        this.id = id;
    }
}
```

```
@Component    //用于将Person类作为Bean注入到Spring容器中
@ConfigurationProperties(prefix = "person") //将配置文件中以person开头的属性注入到该类中
public class Person {

  private int id;             //id
  private String name;       //名称
  private List hobby;        //爱好
  private String[] family; //家庭成员
  private Map map;
  private Pet pet;           //宠物
      // 省略属性getXX()和setXX()方法
    // 省略toString()方法
```

---

```
}
```

@ConfigurationProperties(prefix = "person")注解的作用是将配置文件中以person开头的属性值通过setXX()方法注入到实体类对应属性中

@Component注解的作用是将当前注入属性值的Person类对象作为Bean组件放到Spring容器中，只有这样才能被@ConfigurationProperties注解进行赋值

# 使用@value 注入属性

```
@Component
public class Student {
@Value("${person.id}")
private int id;
@Value("${person.name}")
private String name;
}
```

# Application 启动文件放置位置

Spring boot 项目的 Application 启动文件应该放到主目录下，spring 启动时，会自动扫描启动文件所在的同级目录，及子目录下的所有文件。并加载到 ioc 容器中。

Spring boot 内置 tomcat，默认 8080 端口，路径默认为空。

## 自定义配置文件

## @PropertySource 指定自定义的配置文件和类

- @Configuration注解表示当前类是一个自定义配置类，并添加为Spring容器的组件，这里也可以使用传统的@Component注解；
- @PropertySource("classpath:test.properties")注解指定了自定义配置文件的位置和名称，此示例表示自定义配置文件为classpath类路径下的test.properties文件；
- @ConfigurationProperties(prefix = "test")注解将上述自定义配置文件test.properties中以test开头的属性值注入到该配置类属性中。
- 如果配置类上使用的是@Component注解而非@Configuration注解，那么@EnableConfigurationProperties注解还可以省略

## @Confifiguration 编写自定义配置类

## 随机数设置及参数间引用

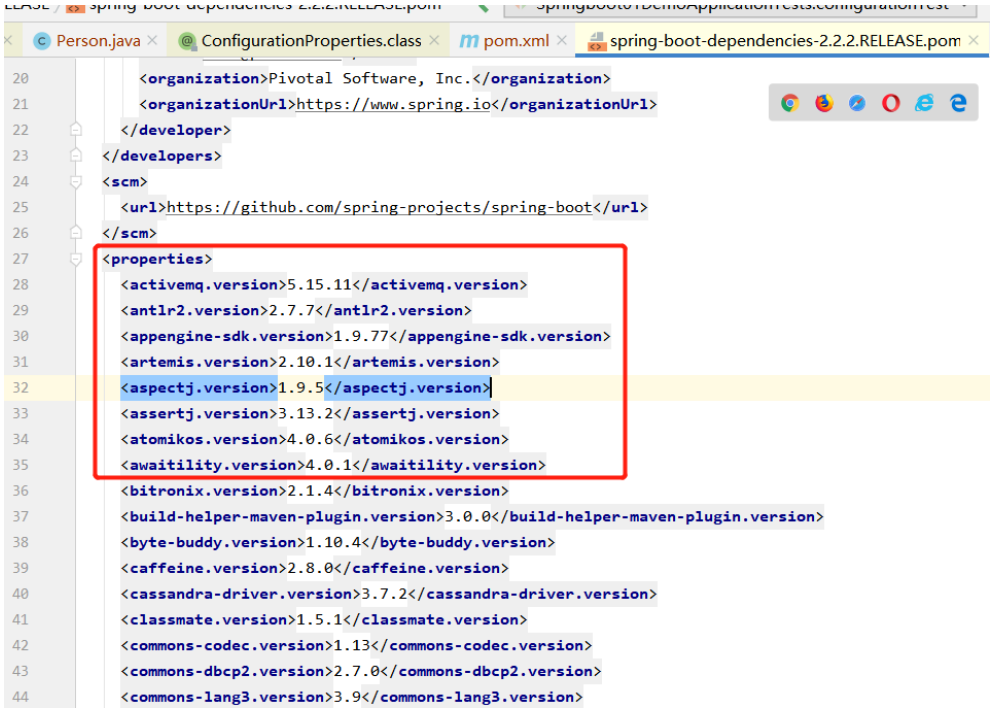
# Spring boot 源码剖析

## 依赖管理

### 1. 为什么 spring boot 项目的 pom 中不需要引入版本？

`spring-boot-starter-parent` 统一版本号管理。

因为 pom 默认引入了父依赖启动器 `spring-boot-starter-parent`，`<artifactId>spring-boot-starter-parent</artifactId>`，而该父级引用标签又有父级 `<artifactId>spring-boot-dependencies</artifactId>`，该 pom 中设定了常用的技术框架依赖的版本，进行版本号统一管理。




## 2. Spring boot 项目运行的依赖 jar 包从何而来？

Maven 的依赖传递。

**创建 spring boot 项目时添加了 spring-boot-starter-web 依赖，而 spring-boot-starter-web 又依赖 spring-webmvc 等 web 场景下所需的底层所有依赖，进而间接依赖了 spring mvc 和 tomcat 等运行必须的依赖包。**

所以不需要额外导入 tomcat 及其他 web 依赖文件等。当然依赖所需的版本号由 spring-boot-starter-parent 统一管理。

通俗的说是将依赖的所有文件进行打包，打包到 **spring-boot-starter-web** 启动文件中。

# 自动配置

```
@SpringBootApplication
public class Springboot01DemoApplication {

    public static void main(String[] args) { SpringApplication.run(Springboot01DemoApplication.class, args); }

}
```

1. Spring boot 到底如何进行自动配置的，都把那些组件进行了自动配置。

`@SpringBootConfiguration` 标明该类为配置类，spring boot 启动时自动加载该类。

`@Configuration` 标明为配置类

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

@EnableAutoConfiguration 自动配置功能

@AutoConfigurationPackage 自动配置包，会把@SpringBootApplication 所在类的包名拿到，并扫描该包及子包，将组件添加到容器中。

@Import(AutoConfigurationPackages.Registrar.class)默认将主配置类(@SpringBootApplication)所在的包及其子包里面的所有组件扫描到 Spring 容器中

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

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

@Import(AutoConfigurationImportSelector.class) 帮助 springboot 应用将**所有符合条件的@Configuration 配置都加载到**当前 SpringBoot 创建并使用的 **IoC 容器**(ApplicationContext)中

# 自定义 Starter

## Spring boot starter 机制

Spring boot 由众多 starter 组成（一系列自动化配置的 starter 插件），可以理解为可插拔式

的插件。想要使用某组件需要在 pom 文件中，配置相应的**依赖启动器**。

## 为什么要自定义 starter？

一些独立于业务模块之外的配置模块，可以将这些配置模块封装成一个个 starter，复用时只需要将其在 pom 中引用即可，spring boot 为我们自动装配。

## 自定义 starter 命名规则

官方使用 spring-boot-starter-xxx
自定义 xxx-spring-boot-starter
用于区分官方和自定义 starter。

```
@Configuration
@ConditionalOnClass(SimpleBean.class)  //注解：当类路径classpath下有指定的类的情况，就会进行自动配置
public class MyAutoConfiguration {

    static {
        System.out.println("MyAutoConfiguration init...");
    }
    @Autowired
    private SimpleBean simpleBean;

    public void test(){
        simpleBean.getId();
    }

}
```

`@ConditionalOnClass(SimpleBean.class)`
该注解表示：当类路径 classpath 下有指定类的情况，springboot 才会去自动配置。否则在该配置类中使用 `SimpleBean` 对象会报错。

也可以直接使用`@Bean` 方式注入 `SimpleBean`

```
@Configuration
@ConditionalOnClass  //注解：当类路径classpath下有指定的类的情况，就会进行自动配置
public class MyAutoConfiguration {

    static {
        System.out.println("MyAutoConfiguration init...");
    }

    @Bean
    public SimpleBean simpleBean(){
        return new SimpleBean();
    }
```

# 执行原理

1. 先创建 new SpringApplication(primarySources)对象
   (1) 设置基本属性
   (2) 项目启动类 SpringbootDemoApplication.class 设置为属性存储起来
   (3) 设置应用类型是 SERVLET 应用（Spring 5 之前的传统 MVC 应用）还是 REACTIVE 应用（Spring 5 开始出现的 WebFlux 交互式应用）

   (4) 设置初始化器(Initializer),最后会调用这些初始化器
   所谓的初始化器就是 org.springframework.context.ApplicationContextInitializer 的实现类,在 Spring 上下文被刷新之前进行初始化的操作

   (5) 设置监听器(Listener)
   (6) 初始化 mainApplicationClass 属性:用于推断并设置项目 main()方法启动的主程序启动类

2. 运行 run 方法
   (1) 获取并启动监听器
   (2) 项目运行环境 Environment 的预配置
   (3) 创建 Spring 容器
   (4) Spring 容器前置处理
   (5) 刷新容器
   (6) Spring 容器后置处理
   (7) 发出结束执行的事件通知
   (8) 执行 Runners
   (9) 发布应用上下文就绪事件

# Spring boot 缓存管理

缓存管理的实体类数据默认使用的是 JDK 序列化方式管理。

## 自定义 RedisTemplate

#### Redis API 默认序列化机制

基于 api 方式的 redis 管用

```java
@Service
public class ApiCommentService {

    @Autowired
    private CommentRepository commentRepository;

    @Autowired
    private RedisTemplate redisTemplate;

    // 使用API方式进行缓存：先去缓存中查找，缓存中有，直接返回，没有，查询数据库
    public Comment findCommentById(Integer id){
        Object o = redisTemplate.opsForValue().get("comment_" + id);
        if(o!=null){
            //查询到了数据，直接返回
            return (Comment) o;
        }else {
            //缓存中没有，从数据库查询
            Optional<Comment> byId = commentRepository.findById(id);
            if(byId.isPresent()){
                Comment comment = byId.get();
                //将查询结果存到缓存中，同时还可以设置有效期为1天
                redisTemplate.opsForValue().set( k: "comment_" + id,comment, l: 1, TimeUnit.DAYS);
                return  comment;
            }
        }
        return  null;
    }
    //更新方法
    public Comment updateComment(Comment comment){
        commentRepository.updateComment(comment.getAuthor(),comment.getId());
        //将更新数据进行缓存更新
        redisTemplate.opsForValue().set("comment_" + comment.getId(),comment);
        return comment;
    }

    //删除方法
    public void deleteComment(Integer id){
        commentRepository.deleteById(id);
        redisTemplate.delete( key: "comment_" + id);
    }
}
```

```java
@Bean
public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory) {
    RedisTemplate<Object, Object> template = new RedisTemplate<>();
    template.setConnectionFactory(redisConnectionFactory);

    // 创建 JSON 格式序列化对象，对缓存数据的 key 和 value 进行转换
    Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new
Jackson2JsonRedisSerializer(Object.class);

    // 解决查询缓存转换异常的问题
    ObjectMapper om = new ObjectMapper();
    om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
```

```
    om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
    jackson2JsonRedisSerializer.setObjectMapper(om);

    //设置 redisTemplate 模板 API 的序列化方式为 json
    template.setDefaultSerializer(jackson2JsonRedisSerializer);
    return template;
}
```

# 自定义 RedisCacheManager

对注解方式生效

```
@Service
public class CommentService {

    @Autowired
    private CommentRepository commentRepository;

    //@Cacheable:  将该方法查询结果comment存放在springboot默认缓存中
     //cacheNames: 起一个缓存命名空间  对应缓存唯一标识

    // value: 缓存结果  key:默认在只有一个参数的情况下，key值默认就是方法参数值 如果没有参数或者多个参数的情况：simpleKeyGenerate
    // 查询方法
    @Cacheable(cacheNames = "comment",unless = "#result==null")
    public Comment findCommentById(Integer id){
        Optional<Comment> byId = commentRepository.findById(id);
        if(byId.isPresent()){
            Comment comment = byId.get();
            return  comment;
        }
        return  null;
    }

    //更新方法
    @CachePut(cacheNames = "comment",key = "#result.id")
    public Comment updateComment(Comment comment){
        commentRepository.updateComment(comment.getAuthor(),comment.getId());
        return comment;
    }

    //删除方法
    @CacheEvict(cacheNames = "comment")
    public void deleteComment(Integer id){
        commentRepository.deleteById(id);
    }

```

# 数据库脚本

#创建数据库
drop DATABASE springbootdata;
CREATE DATABASE springbootdata;

```
#选择使用数据库
USE springbootdata;
#创建表 t_article 并插入相关数据
DROP TABLE IF EXISTS t_article;
CREATE TABLE t_article (
id int(20) NOT NULL AUTO_INCREMENT COMMENT '文章 id',

title varchar(200) DEFAULT NULL COMMENT '文章标题',
content longtext COMMENT '文章内容', PRIMARY KEY (id)
)

ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;


INSERT  INTO t_article VALUES ('1', 'Spring Boot 基础入门','从入门到精通讲解・・・');

INSERT  INTO t_article VALUES ('2', 'Spring Cloud 基础入门','从入门至 U 精通讲解・・・');

#创建表 t_comment 并插入相关数据
DROP TABLE IF EXISTS t_comment;
CREATE TABLE t_comment (
id int(20) NOT NULL AUTO_INCREMENT COMMENT '评论 id',
content longtext COMMENT '评论内容',
author varchar(200) DEFAULT NULL COMMENT '评论作者',
a_id int(20) DEFAULT NULL COMMENT '关联的文章 id', PRIMARY KEY (id)
)ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

INSERT  INTO t_comment VALUES ('1','很全、很详细','luccy' , '1');
INSERT  INTO t_comment VALUES ('2','赞一个','tom' , '1');
INSERT  INTO t_comment VALUES ('3','很详纟 田','eric', '1');
INSERT  INTO t_comment VALUES ('4','很好，非常详细','张三','1');
INSERT  INTO t_comment VALUES ('5','很不错','李四','2');
```