

6.5 服务器开发补充

- Spring Security

- 日志

- Lombok

- 配置文件

目 录

1

Spring Security

2

日志

3

Lombok

4

配置文件

信息安全概述

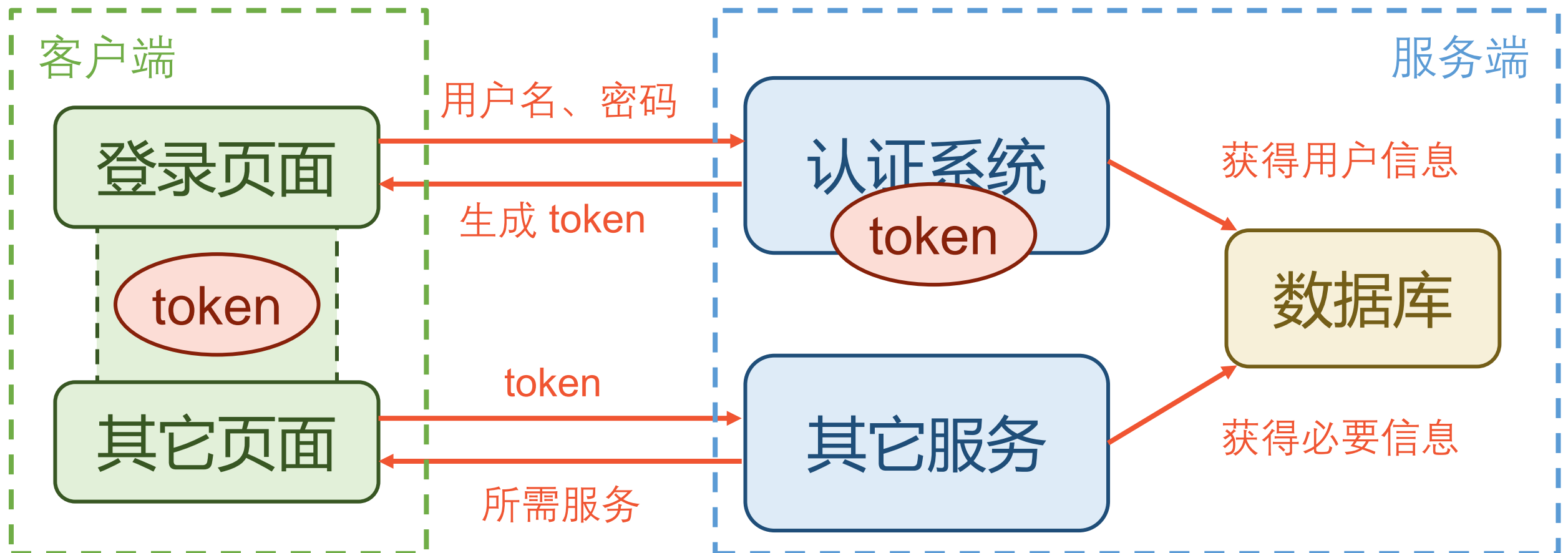
- 在实际的服务器开发中，我们不仅需要给用户提供各种各样的服务，还需要保证这些服务的（信息）安全性。要保证服务安全，一般的做法是给每一个用户分配一些**权限**[権限]，使每一个用户只能使用它被允许的服务，即**访问控制**[アクセス制御]。
- 比如，Alice 只被允许登录自己的账号并发布信息，不应该能登录 Bob 的；外部人员只允许使用公开的服务，而修改服务器中核心数据的服务只有网站管理人员被允许使用，等等。

权限管理的要素

- 管理权限的过程可以分为三个步骤，也被称为“3A”：
 - **认证**[認證] (Authentication)：识别用户的身份，使系统能够确认使用该服务的用户“是谁”。比如 **Alice** 在登录时使用正确的用户名和密码使网站确认是她在提供服务。
 - **授权**[承認] (Authorization)：给每一个用户分配合适的权限（可以使用哪些服务）。比如 **Alice** 是个普通用户，她只能发帖或回复；**Bob** 是个管理员，他可以删除别人的帖子或查看别人的操作纪录。
 - **记录** (Accounting)：把用户操作记录下来以便管理。
- 我们一般讨论的信息安全可能指两种概念：一种是设计系统使其包含对上述步骤的实现；另一种是设计算法保证上述步骤的隐秘性（比如加密算法）。本节课只讨论前者，即 **Spring** 中是如何实现权限管理的。

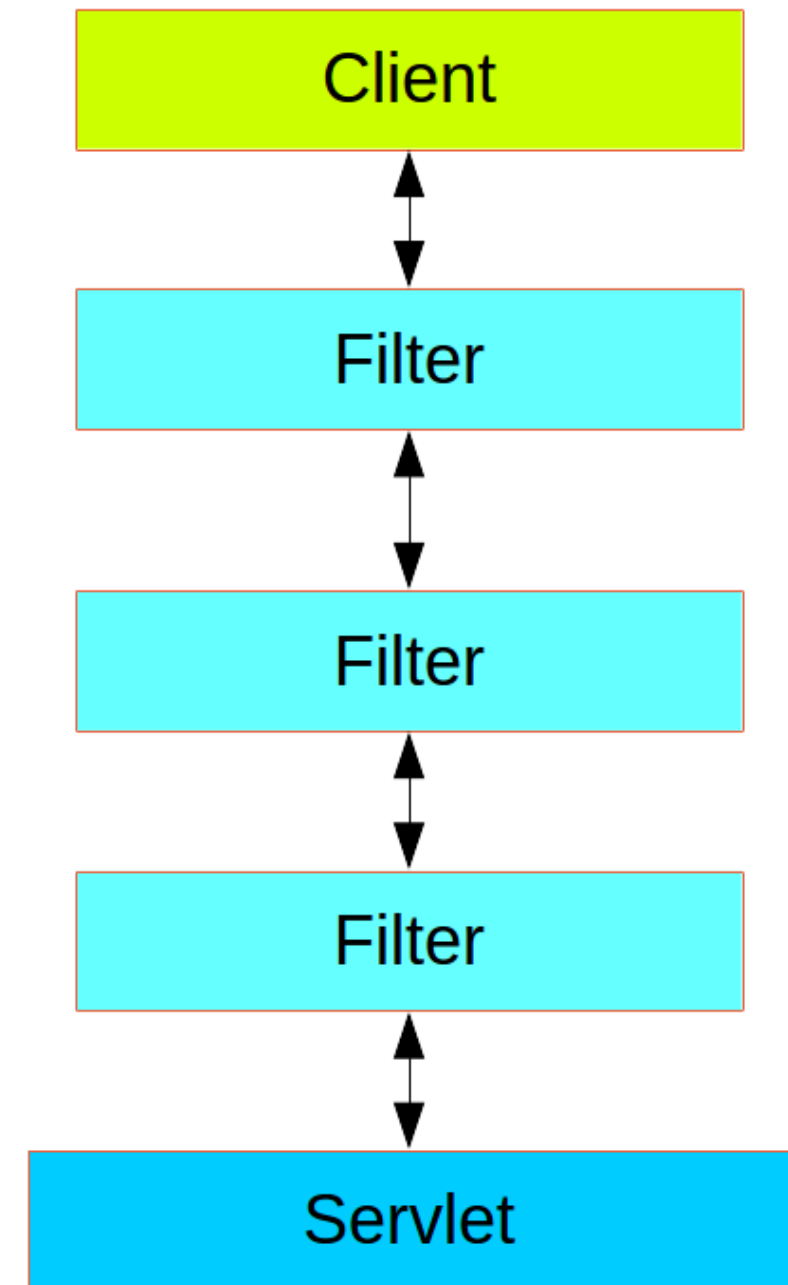
Spring Security 基本原理

- Spring 提供了 **Spring Security** 这一框架处理和认证、授权等一系列安全相关操作。认证模式有很多种，我们这里介绍最主流的一种：**Token 模式**。
- 我们可以简单把使用 **token** 进行认证的过程整理如下：



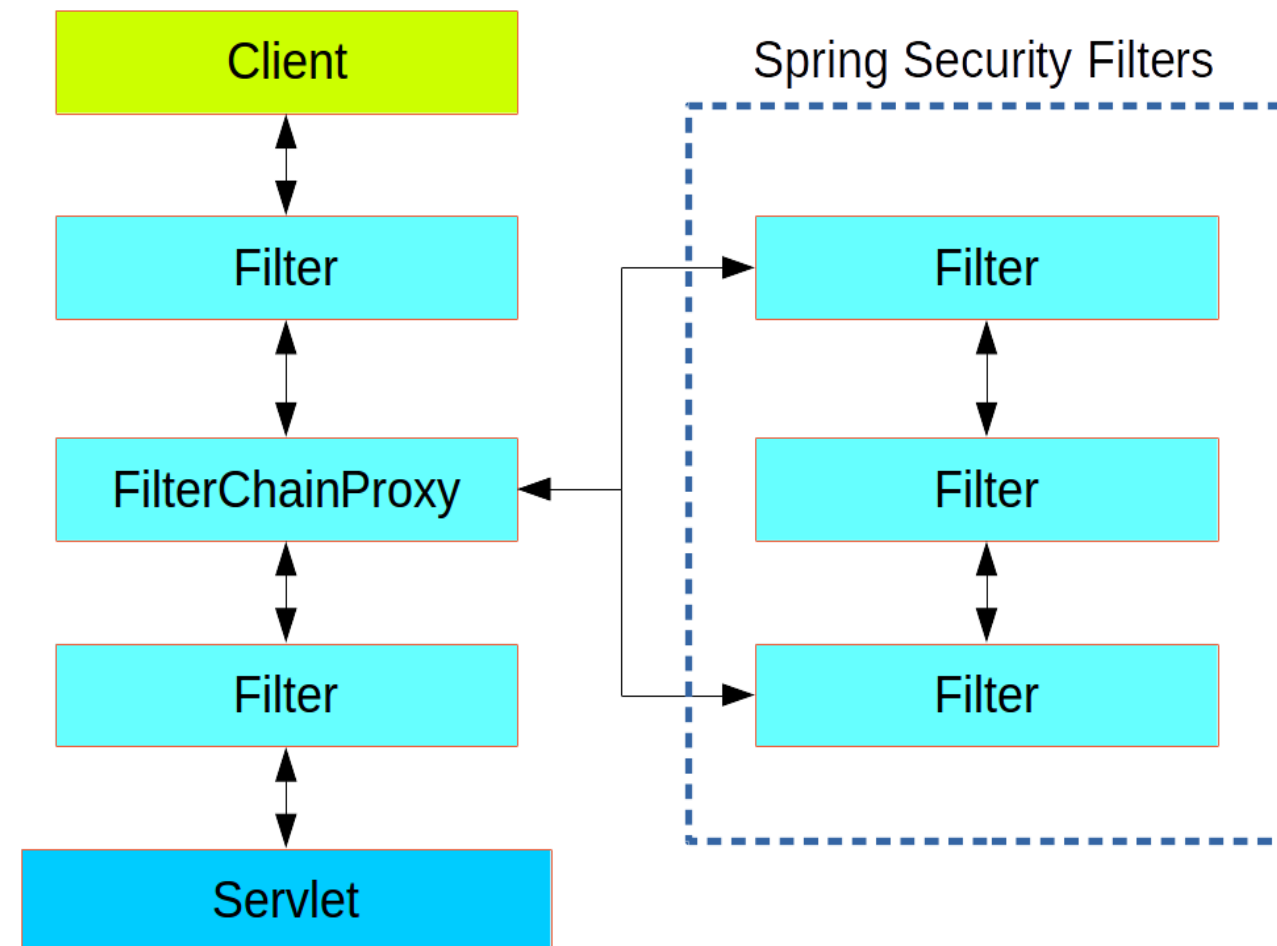
过滤器

- 在授权时，Spring Security 采取一种被称为**过滤器链**[フィルタチェン]的模式。
- **过滤器**[フィルター]是 Servlet 中一种处理 HTTP 请求和响应的对象。客户端发送的请求不会直接发送至服务器，而是需要先通过某些过滤器。每个过滤器会对请求进行一些审核和修改，再发给下一级过滤器。而从下一级过滤器发回的响应也可以通过此过滤器审核或修改。




Spring 过滤器链

- Spring Security 在 Servlet 容器中加入了一个特殊的过滤器，名为 **FilterChainProxy**。和用户认证、授权相关的操作都通过这个过滤器完成。
- 这个过滤器本身又包含一些 Spring Security 提供的过滤器，构成一个过滤器链。这些过滤器各司其职，比如有些负责认证 token，有些负责确认用户权限。在实际使用的过程中，我们可以根据自己的需求选择合适的过滤器加入过滤器链中。

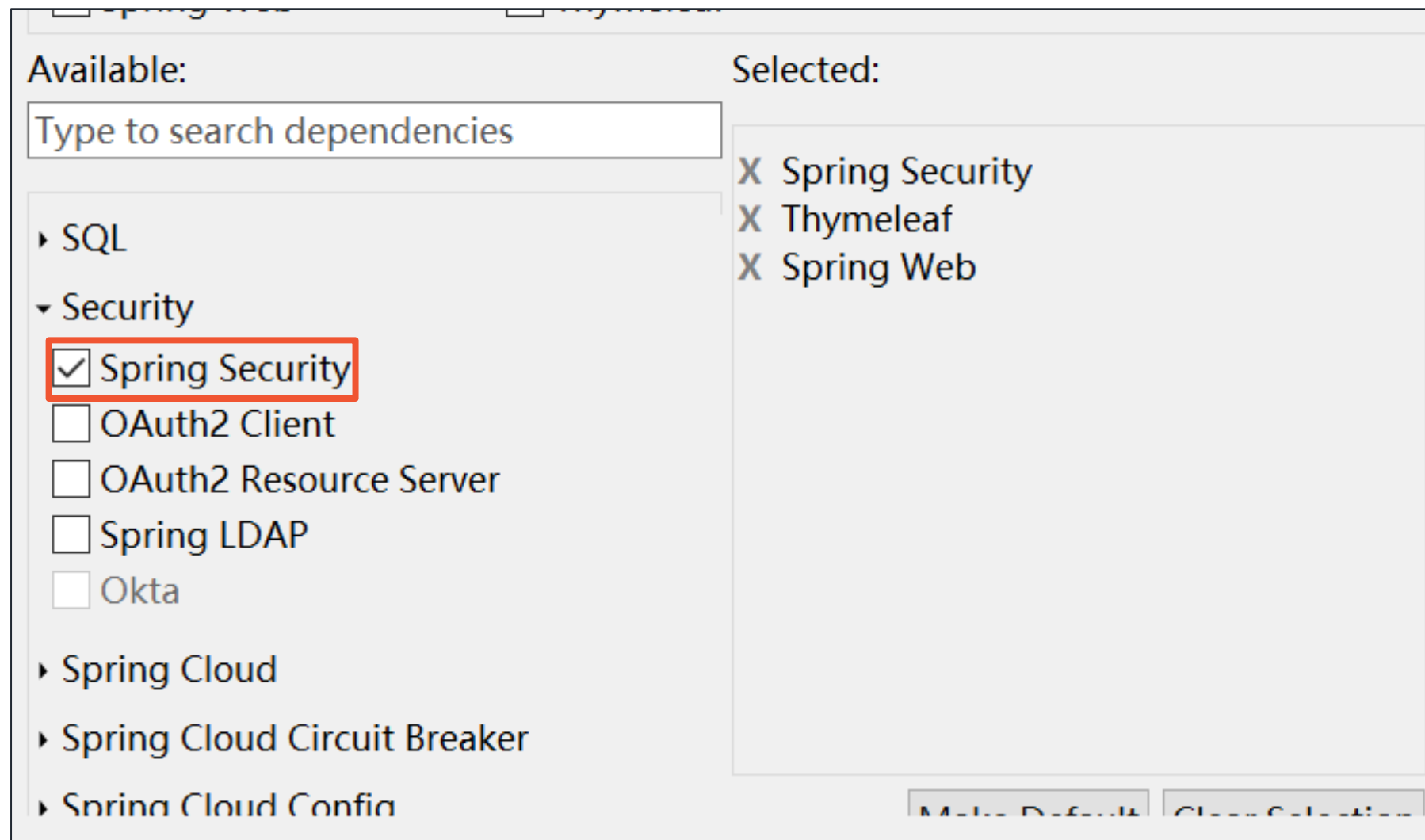


简单的 Security 项目

- Spring Security 提供了大量认证和授权相关的算法、API 和各种过滤器，学习起来是非常困难的。这里提供一个 Spring Security 的官方教程链接：
 <https://spring.io/guides/topicals/spring-security-architecture/>
- 不过如果只是实现一个基于登录和注册页面最简单的登录系统，Spring Boot 提供了一个相对简单的 API 帮助我们创建网站。尽管它使用的模式因为过于简单，存在安全隐患，**不适合在实际产品中使用**，但我们还是可以通过它体验一下 Spring Security 的基本模式。
- 接下来就让我们一起实现这个简单的登录系统。

创建项目

- 要使用 Spring Security 的功能，需要在创建项目时（或创建后使用 Maven）添加对 Spring Security 的依赖。

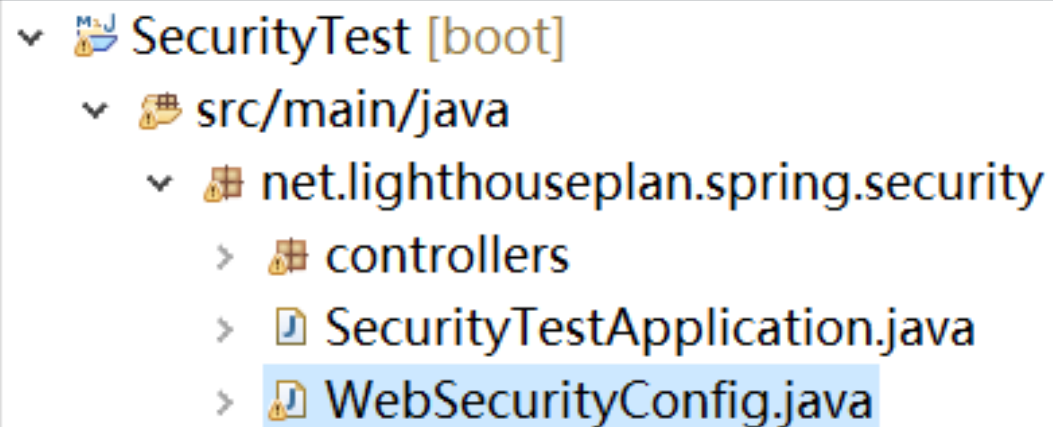


创建配置类

- Spring Security 的认证和授权功能需要在 Java 代码中设置，创建一个继承了 **WebSecurityConfigurerAdapter** 的 Java 类，并加上相关注解：

```
1 @Configuration
2 @EnableWebSecurity
3 public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
4     //...
5 }
```

- 在核心包内的任意地方创建都可以，比如 Application 类旁：



```
SecurityTest [boot]
├── src/main/java
│   └── net.lighthouseplan.spring.security
│       ├── controllers
│       ├── SecurityTestApplication.java
│       └── WebSecurityConfig.java
```

重写 userDetailsService() 方法

- 通过重写配置类中的 **userDetailsService()** 方法，我们可以设置网站认证的方法：

```
1 @Bean
2 @Override
3 public UserDetailsService userDetailsService() {
4     UserDetails user =
5         User.withDefaultPasswordEncoder()
6             .username("Alice")
7             .password("123456")
8             .roles("USER")
9             .build();
10
11     return new InMemoryUserDetailsManager(user);
12 }
```

创建一个用户的信息

用户名为 Alice
密码为 123456

用户的角色为“USER” (一般用户)

创建一个包含此用户的认证系统

重写 `configure()` 方法

- 通过重写 `configure()` 方法，我们可以添加一些和网站授权有关的配置信息：

```
1 @Override
2 protected void configure(HttpSecurity http) throws Exception {
3     http
4         .authorizeRequests()
5             .antMatchers("/register").permitAll()
6             .anyRequest().authenticated()
7             .and()
8         .formLogin()
9             .loginPage("/login")
10            .permitAll()
11            .and()
12            .logout()
13                .permitAll();
14 }
```

设置任何用户都能访问注册页面

设置其它请求都需要用户登录

设置自定义登录界面

和登出有关的设置

配置 login 页面

- 如果你不添加相关设置，Spring 将会使用一个默认的 login 页面。添加配置后，你可以自己实现控制器和 HTML 模板以实现自定义登录页面。

```
.and()  
.formLogin()  
.loginPage("/login")  
.permitAll()  
.and()
```

```
1 @GetMapping("/login")  
2 public String getLoginPage() {  
3     return "login.html";  
4 }
```

- 在 login.html 中，需要向 **/login** 发送一个 **POST** 请求，并且包含对应的 **username** 和 **password** 属性。你不需要编写该请求的控制器。
- 在登录成功后，用户会返回他访问 login 页面前尝试访问的链接。如果没有，则会访问 **/**。

获得用户信息

- 你可以随时使用以下方法在 Java 中获得用户相关信息：

```
1 UserDetails user = (UserDetails) SecurityContextHolder  
2     .getContext().getAuthentication().getPrincipal();  
3  
4 mav.addObject("name", user.getUsername());
```

- 如果只是想在 HTML 中显示用户名，也可以使用 Thymeleaf 的便利语法：

```
<h1>Hello, [[${#httpServletRequest.remoteUser}]]!</h1>
```


Q & A

Question and answer

目 录

1

Spring Security

2

日志

3

Lombok

4

配置文件

日志

- 到目前为止，我们都是使用 `System.out.println()` 等方法输出测试或 Debug 信息的。在实际的开发过程中，我们通常会把这样的测试信息或系统的运行纪录保存在外部文件中，这些文件被称为 **日志**`[ログ]`。
- 使用日志而不是只在控制台中输出有几个好处：
 - **每一次**运行信息都被自动保存，可以随时查看之前的纪录；
 - 当发生**错误**导致程序意外终止时也能留下纪录；
 - 开发者可以根据用户给的日志信息进行 **Debug**。

日志等级

- 实际输出日志的过程中，经常会把所有输出信息根据重要程度分成几个**等级**。一种常用的等级结构如右表所示：
- 这样，在程序不同的执行场景下，比如在开发时、测试时、或者发布的产品被用户使用时，我们可以选择在控制台以及外部日志文件里输出什么等级以上的信息。

| 等级 | 含义 |
|-------|---------------|
| TRACE | 细节信息 |
| DEBUG | 对 Debug 有用的信息 |
| INFO | 一般信息 |
| WARN | 可能导致问题的警告信息 |
| ERROR | 错误或异常的相关信息 |

SLF4J

- 有非常多的 Java 类库提供了和输出日志有关的功能，包括 Log4J、Logback、SLF4J 等。其用法大多大同小异。本节简单介绍用法较为简单的 **SLF4J**。
- 由于 Spring Boot 的依赖已经包含了 SLF4J，我们不需要额外导入依赖。相关方法位于 **org.slf4j** 包内。

获得 Logger

- 要输出日志，我们首先要获得一个 **Logger** 类的对象：

```
Logger logger = LoggerFactory.getLogger("Logger Name");
```

- 其中，Logger Name 是这个 Logger 的名字。我们一般会用当前类名作为 Logger 的名字：

```
Logger logger = LoggerFactory.getLogger(MainController.class);
```


输出日志

- 有了 Logger 后，使用其同名方法即可输出相应等级的日志。例如，以下代码输出一个 WARN 等级的日志：

```
logger.warn("Caution!");
```

- 和 System.out.println() 类似，我们可以在控制台看到各个等级日志的输出：

```
2022-08-13 07:01:47.692 INFO 17308 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring Dispa
2022-08-13 07:01:47.692 INFO 17308 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dis
2022-08-13 07:01:47.693 INFO 17308 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization
2022-08-13 07:01:47.720 INFO 17308 --- [nio-8080-exec-1] n.l.s.s.controllers.MainController : Hello!
2022-08-13 07:01:47.720 WARN 17308 --- [nio-8080-exec-1] n.l.s.s.controllers.MainController : Caution!
2022-08-13 07:01:47.720 ERROR 17308 --- [nio-8080-exec-1] n.l.s.s.controllers.MainController : Something went wrong...
```

- 可以注意到，日志的输出日期、时间、线程等也都会被记录下来。

配置日志输出

- 我们注意到目前只有 INFO 等级以上的信息被输出至控制台。我们可以在 `application.properties` 中添加以下配置信息以改变输出日志的等级限制：

```
logging.level.[logger 的类名或包名]=[最低输出等级]
```

- 比如要把这个 Security 项目里所有自定义的 logger 的输出等级都设为 DEBUG 以上，只要书写以下配置：

```
logging.level.net.lighthouseplan.spring.security=DEBUG
```

- 还可以使用以下配置将日志输出至外部文件：

```
logging.file.name=[日志文件路径]
```

Q & A

Question and answer

目 录

1

Spring Security

2

日志

3

Lombok


4

配置文件

JavaBeans 规范

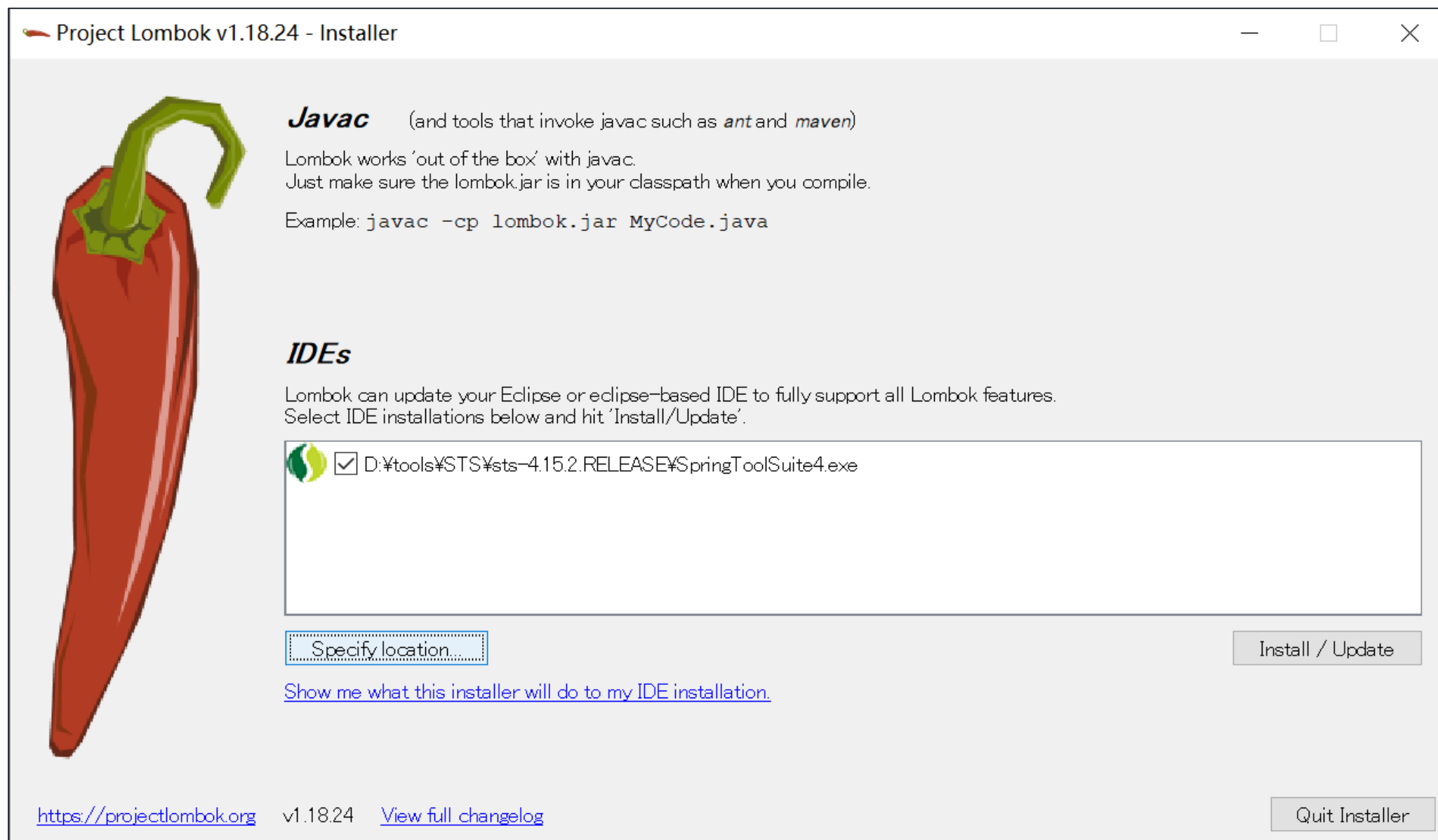
- 一个 Java 类满足以下条件，则被称为一个 **JavaBean**:
 - 所有的属性均为 **private**;
 - 所有属性都有对应的 **getter**、**setter**;
 - 有一个 **public** 的无参构造方法;
 - 实现 **Serializable** 接口（可被序列化，用于存储或传输）。
- 在开发过程中，经常使用类似 JavaBean 的规范作为 **DTO** 或 **参数类** 的标准以管理数据。有些规范（比如 **Plain Old Java Object**, **POJO**）会略微有些不同，比如构造方法不一定无参、不需要实现 **Serializable** 等。

Lombok

- 可以注意到，不同的 JavaBean 类定义方式大同小异，重复度很高。除此之外，像是包含所有参数的构造器、toString() 方法等代码的书写也都是类似的。
- Lombok 就是一个用于帮助我们实现这些代码的自动生成，提高开发效率的工具包。
- **Lombok 需要额外安装。**从以下链接下载安装包：
 <https://projectlombok.org/download>

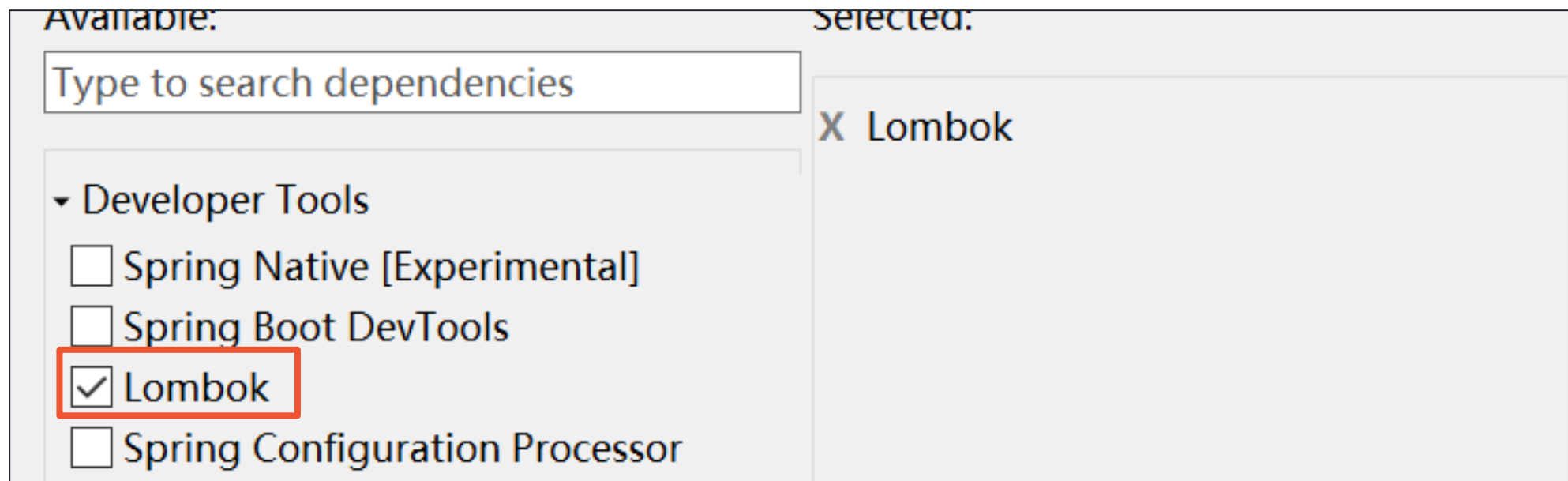
安装 Lombok

- 如果没有显示 STS，点击 Specify Location，找到 STS 文件。选择 STS 文件，点击 Install / Update 安装：



添加 Lombok 依赖

- 重启 STS，你现在可以在新建项目时加入 Lombok 的依赖（当然也可以之后通过 Maven 添加）：



@Getter 与 @Setter

- Lombok 的主要使用方法是在某些定义前添加一些注解。
- 比如，在变量的定义前添加 **@Getter** 或 **@Setter** 注解，可以自动为其生成对应的 `getter` 和 `setter`:

```
1 public class Student {  
2     @Getter  
3     @Setter  
4     private String name;  
5 }
```

```
1 Student student = new Student();  
2 student.setName("Alice");  
3 System.out.println(student.getName()); // => Alice
```


常用注解

- 下表列出了一些其它的 Lombok 中的常用注解（都写在**类前**）：

| 注解 | 功能 |
|--------------------------|---|
| @NoArgsConstructor | 生成无参构造方法 |
| @RequiredArgsConstructor | 生成用参数初始化必须变量的构造方法 |
| @AllArgsConstructor | 生成用参数初始化所有变量的构造方法 |
| @ToString | 生成 toString() 方法 |
| @EqualsAndHashCode | 生成 equals() 和 hashCode() 方法 |
| @Data | 给类加上 @ToString、@EqualsAndHashCode和 @RequiredArgsConstructor；给所有变量加上 @Getter 和 @Setter |
| @Log (@Slf4j) | 生成一个 (SLF4J) Logger 变量 log |

@Builder

- **@Builder** 注解可以生成一个用于创建对象的静态 **builder()** 方法。它可以让我们的创建对象的代码可读性更高，方便扩展：

```
1 @Data
2 @Builder
3 public class Student {
4     private Long id;
5     private String name;
6     private int score;
7 }
```

```
1 Student student = Student.builder()
2     .id(1L)
3     .name("Alice")
4     .score(90)
5     .build();
6 System.out.println(student); // => Student(id=1, name=Alice, score=90)
```

Q & A

Question and answer



目次

1

Spring Security

2

日志

3

Lombok

4

配置文件

application.properties

- Spring Boot 使用了一个全局的配置文件 **application.properties** 用于存储整个项目的通用配置信息。
- 它应该被放在 `src/main/resources` 目录下或者项目路径的 `/config` 目录下。
- 我们可以使用该 `Properties` 文件修改某些默认配置的值，比如默认被设为 8080 的端口号，可以通过 `server.port` 设置：

```
server.port=8090
```

可以配置的参数

- 除了我们之前使用过的参数外，还有很多其它配置可以通过 `application.properties` 进行设置。下表列出了其中一些：

| 参数 | 功能 |
|--------------------------------------|--|
| <code>spring.application.name</code> | 应用的名称 |
| <code>server.address</code> | 服务器的地址 |
| <code>spring.mail.host</code> | 服务器使用的邮箱主机地址 |
| <code>logging.file.path</code> | 日志文件的地址 |
| <code>spring.config.name</code> | 配置文件的地址（默认是 <code>application</code> ） |

- 完整的可配置参数列表可以在以下文档查询：

 <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html>

Properties 文件的问题

- 实际使用的配置通常都满足一定的**层次结构**。但传统 .properties 文件没有利用这种结构，在配置参数较多、参数名较长时可读性会很低：

```
1 spring.datasource.url=jdbc:postgresql://localhost:5432/security
2 spring.datasource.username=postgres
3 spring.datasource.password=123456
4
5 logging.file.name=application.log
6 logging.level.root=INFO
7 logging.level.net.lighthouseplan.spring.security=DEBUG
```

YAML 格式

- Spring Boot 也支持使用 **YAML** 格式保存配置。YAML 以一种层次化的书写方式表示信息，因此很适合描述配置文件：

```
1 spring:
2   datasource:
3     url: jdbc:postgresql://localhost:5432/security
4     username: postgres
5     password: 123456
6
7   logging:
8     file:
9       name: application.log
10    level:
11      root: INFO
12    net.lighthouseplan.spring.security: DEBUG
```

- 使用 YAML 格式书写的配置可以写入 **application.yml** 中，替换掉原本的 **application.properties** 即可使用。

Q & A

Question and answer

总结

Sum Up

1. Security:

- ① 网站权限系统的一般概念：认证、授权、使用 token 进行认证的方法。
- ② Spring Security 的核心概念和最基本的使用方法。

2. 日志的概念和日志工具包的使用方法。

3. Lombok 的使用方法。

4. 配置文件的多种书写方法。

THANK YOU!