下图模块参见讲义。

配置国际化页面：

第一步：

编写多语言国际化配置文件

login_zh_CN.properties

```
login.tip=请登录
login.username=用户名
login.password=密码
login.rememberme=记住我
login.button=登录
```

login_en_US.properties

```
login.tip=Please sign in
login.username=Username
login.password=Password
login.rememberme=Remember me
login.button=Login
```

login.properties为自定义默认语言配置文件，login_zh_CN.properties为自定义中文国际化文件，login_en_US.properties为自定义英文国际化文件

需要说明的是，Spring Boot默认识别的语言配置文件为类路径resources下的messages.properties；其他语言国际化文件的名称必须严格按照“文件前缀名*语言代码*国家代码.properties”的形式命名

本示例中，在项目类路径resources下自定义了一个i18n包用于统一配置管理多语言配置文件，并将项目默认语言配置文件名自定义为login.properties，因此，后续还必须在项目全局配置文件中进行国际化文件基础名配置，才能引用自定义国际化文件

2. 编写配置文件

打开项目的application.properties全局配置文件，在该文件中添加国际化文件基础名设置，内容如文件

```
# 配置国际化文件基础名
spring.messages.basename=i18n.login
```

spring.messages.basename=i18n.login”设置了自定义国际化文件的基础名。其中，i18n表示国际化文件相对项目类路径resources的位置，login表示多语言文件的前缀名。如果开发者完全按照Spring Boot默认识别机制，在项目类路径resources下编写messages.properties等国际化文件，可以省略国际化文件基础名的配置

第二步：

自定义区域信息解析器

```
@Configuration
public class MyLocaleResovel implements LocaleResolver {

    //自定义区域解析方式
     @Override
    public Locale resolveLocale(HttpServletRequest httpServletRequest) {
        // 1.获取页面手动切换的语言参数
         String l = httpServletRequest.getParameter("l");
         // 2.获取请求头自动传递的语言参数Accept-Language
         String header = httpServletRequest.getHeader("Accept-Language");
         Locale locale=null;
```

```
        // 如果手动切换的语言参数l为空则使用请求头自动传递的参数header
        if(!StringUtils.isEmpty(l)){
               String[] split = l.split("_");
                locale=new Locale(split[0],split[1]);
        }else {
           // Accept-Language: en-US,en;q=0.9;zh-CN;q=0.8,zh;q=0.7
                String[] splits = header.split(",");
                String[] split = splits[0].split("-");
                locale=new Locale(split[0],split[1]);
        }
       return locale;
    }
 @Override
 public void setLocale(HttpServletRequest httpServletRequest, @Nullable
     HttpServletResponse httpServletResponse, @Nullable Locale locale){
 }
 // 将MyLocalResovl类注册为LocaleResolver的Bean组件
 @Bean
 public LocaleResolver localeResolver(){
    return new MyLocalResovel();
    }
}
```

第三步：

前端使用

```
<a class="btn btn-sm" th:href="@{/toLoginPage(l='zh_CN')}">中文</a>
<a class="btn btn-sm" th:href="@{/toLoginPage(l='en_US')}">English</a>
```

springboot缓存管理：

使用默认缓存：

springboot继承了spring的缓存管理功能，通过使用@EnableCaching注解开启缓存支持，springboot就可以启动缓存管理自动化配置。

```
@EnableCaching //开启spring boot基于注解的缓存管理支持
@SpringBootApplication
public class Springboot05CacheApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot05CacheApplication.class, args);
    }

}
```

Service接口层添加@Cacheable注解

```java
public class CommentService {
    //unless：如果返回结果为空，不加入缓存
    @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;
    }
}
```

cache生成key策略：

1. springboot默认缓存ConcurrentMapCacheManager就是Map集合（如下）：
   ConcurrentMap<String, Cache> cacheMap;
2. Cache保存的就是上面的comment对象，每一个Cache有多个k-v键值对，key默认在只有一个参数的情况下，如上面只有一个id，那么key的值就是id;
   如果没有参数或者多个参数的情况，使用simpleKeyGenerate对象生成key,多个参数会把参数放进去生成key,如下图1.5;
3. 也可以指定key，如图1.1
4. key的SpEL表达式:如图1.2

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

1.1

常用的SPEL表达式

描述	示例
当前被调用的方法名	#root.mathodName
当前被调用的方法	#root.mathod
当前被调用的目标对象	#root.target
当前被调用的目标对象类	#root.targetClass
当前被调用的方法的参数列表	#root.args[0] 第一个参数, #root.args[1] 第二个参数...
根据参数名字取出值	#参数名, 也可以使用 #p0 #a0 0是参数的下标索引
当前方法的返回值	#result

1.2

@Cacheable注解提供多个属性，对缓存存储进行相关配置，可以和SPEL表达式组合使用如图1.1
key="#result.id"//指定返回结果的id为key

属性名	说明
value/cacheNames	指定缓存空间的名称，必配属性。这两个属性二选一使用
key	指定缓存数据的key，默认使用方法参数值，可以使用SpEL表达式
keyGenerator	指定缓存数据的key的生成器，与key属性二选一使用
cacheManager	指定缓存管理器
cacheResolver	指定缓存解析器，与cacheManager属性二选一使用
condition	指定在符合某条件下，进行数据缓存
unless	指定在符合某条件下，不进行数据缓存
sync	指定是否使用异步缓存。默认false

1.3

cache其他注解介绍：

修改建议使用该注解：

> **3. @CachePut注解**
>
> 目标方法执行完之后生效, @CachePut被使用于修改操作比较多,哪怕缓存中已经存在目标值了,但是这个注解保证这个方法**依然会执行**,执行之后的结果被保存在缓存中
>
> @CachePut注解也提供了多个属性，这些属性与@Cacheable注解的属性完全相同。
>
> 更新操作,前端会把id+实体传递到后端使用,我们就直接指定方法的返回值从新存进缓存时的 `key="#id"`, 如果前端只是给了实体,我们就使用 `key="#实体.id"` 获取key. 同时,他的执行时机是目标方法结束后执行, 所以也可以使用 `key="#result.id"`, 拿出返回值的id

删除建议使用该注解：

> **4. @CacheEvict注解**
>
> @CacheEvict注解是由Spring框架提供的，可以作用于类或方法（通常用在数据删除方法上），该注解的作用是删除缓存数据。@CacheEvict注解的默认执行顺序是，先进行方法调用，然后将缓存进行清除。

基于注解的Redis缓存实现：

1.导入依赖：

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

当我们添加进redis相关的启动器之后, SpringBoot会使用`RedisCacheConfigratioin`当做生效的自动配置类进行缓存相关的自动装配,容器中使用的缓存管理器是`RedisCacheManager`, 这个缓存管理器创建的Cache为`RedisCache`, 进而操控redis进行数据的缓存

RedisCacheManager是spring-boot-starter-data-redis里面的，RedisCacheConfiguration是springboot的。

2.配置properties：

```
# Redis服务地址
spring.redis.host=127.0.0.1
# 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=
```

3.对CommentService类中的方法进行修改使用@Cacheable、@CachePut、@CacheEvict三个注解定制缓存管理，分别进行缓存存储、缓存更新、缓存删除的演示

```
@Service
public class CommentService {
    @Autowired
    private CommentRepository commentRepository;
    // 查询方法
    @Cacheable(cacheNames = "comment",unless = "#result==null")//当结果为空，不进行缓存
    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);
    }
}

```

```
//Comment需要实现序列化Serializable
@Entity
@Table(name = "t_comment")
public class Comment implements Serializable {}
```

```
#设置基于注解的Redis缓存数据统一设置有效期为1分钟，单位毫秒（一般不建议这么做）
spring.cache.redis.time-to-live=60000
```

该方法放进缓存中value是HEX格式存储，不方便可视化管理，使用自定义Redis缓存序列化机制即可

还可以打开Redis客户端可视化管理工具Redis Desktop Manager连接本地启用的Redis服务，查看具体的数据缓存效果

执行findById()方法查询出的用户评论信息Comment正确存储到了Redis缓存库中名为comment的名称空间下。其中缓存数据的唯一标识key值是以“名称空间comment::+参数值（comment::1）”的字符串形式体现的，而value值则是经过JDK默认序列格式化后的HEX格式存储。这种JDK默认序列格式化后的数据显然不方便缓存数据的可视化查看和管理，所以在实际开发中，通常会自定义数据的序列化格式

基于API的Redis缓存实现：

```
@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("comment_" + id,comment,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);
```

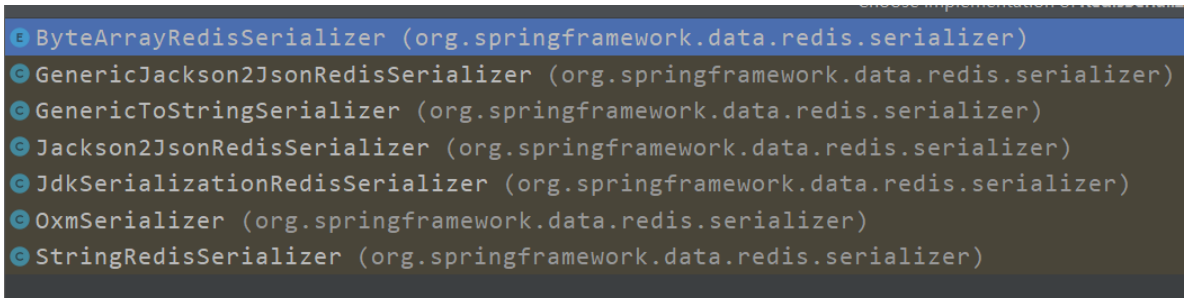
```
        redisTemplate.delete("comment_" + id);
    }
}
RedisTemplate在spring-boot-starter-data-redis里的spring-data-redis.jar包里
```

Redis API默认序列化机制：

1. 使用RedisTemplate进行Redis数据缓存操作时，序列化方式为空时默认使用JdkSerializationRedisSerializer方式，所以进行数据缓存的实体类必须实现JDK自带的序列化接口（例如Serializable，这种方式不方便可视化查看和管理，如下图）

2. 使用RedisTemplate进行Redis数据缓存操作时，如果自定义缓存序列化方式，defaultSerializer那么将使用自定义的序列化方式。（下面会介绍这种方式）

RedisSerializer是一个Redis序列化接口，默认有6个实现类，6个实现类代表6种不同的数据序列化方式。开发者可以根据需要选择序列化方式。（例如JSON方式）




```
public class RedisAutoConfiguration {
    @Bean//建立一个名称为方法名redisTemplate，value是返回值template的bean
    @ConditionalOnMissingBean(name = "redisTemplate")//如果有bean的名字是
redisTemplate则下面方法不生效，否则默认使用下面的返回值。
    public RedisTemplate<Object, Object> redisTemplate(
            RedisConnectionFactory redisConnectionFactory) throws
UnknownHostException {
        RedisTemplate<Object, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }
    ...
}
```

Redis API自定义序列化机制：

项目引入Redis依赖后，Spring Boot提供的RedisAutoConfiguration自动配置会生效，打开RedisAutoConfiguration源码，里面有关于RedisTemplate的定义方式，使用自定义序列化方式的RedisTemplate进行数据缓存操作，需创建一个名为redisTemplate的Bean组件，并在该组件中设置对应的序列化方式即可（会覆盖原有默认序列化方式）。

```
@Configuration
public class RedisConfig {
```

```
    @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（工具类，不用理解具体方式）
        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;
    }
}
```

Redis 注解默认序列化机制：

和api方式类似，只是默认的bean是cacheManager

```
public class RedisCacheConfiguration {
    @Bean
    public RedisCacheManager cacheManager(RedisConnectionFactory redisConnectionFactory, ResourceLoader resourceLoader) {
        RedisCacheManagerBuilder builder =
                RedisCacheManager.builder(redisConnectionFactory)

.cacheDefaults(this.determineConfiguration(resourceLoader.getClassLoader()));
        List<String> cacheNames = this.cacheProperties.getCacheNames();
        if (!cacheNames.isEmpty()) {
            builder.initialCacheNames(new LinkedHashSet(cacheNames));
        }
        return
                (RedisCacheManager)
this.customizerInvoker.customize(builder.build());
    }

    private org.springframework.data.redis.cache.RedisCacheConfiguration determineConfiguration(ClassLoader classLoader) {
        if (this.redisCacheConfiguration != null) {
            return this.redisCacheConfiguration;
        } else {
            Redis redisProperties = this.cacheProperties.getRedis();
            org.springframework.data.redis.cache.RedisCacheConfiguration
                    config =

 org.springframework.data.redis.cache.RedisCacheConfiguration.defaultCacheConf
            ig();
            config = config.serializeValuesWith(SerializationPair.fromSerializer(
```

```
                                new JdkSerializationRedisSerializer(classLoader)));
            return config;
        }
    }
}
```

Redis 注解自定义序列化机制：

和api方式类似，只是覆盖的bean是cacheManager

```
//自定义RedisCacheManager（该方式针对基于注解方式Redis缓存实现）
    @Bean
    public RedisCacheManager cacheManager(RedisConnectionFactory
redisConnectionFactory) {
        // 分别创建String和JSON格式序列化对象，对缓存数据key和value进行转换
        RedisSerializer<String> strSerializer = new StringRedisSerializer();
        Jackson2JsonRedisSerializer jacksonSeial =
                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);
        jacksonSeial.setObjectMapper(om);

        // 定制缓存数据序列化方式及时效
        RedisCacheConfiguration config =
RedisCacheConfiguration.defaultCacheConfig()
                .entryTtl(Duration.ofDays(1))//设置缓存时长1天
                .serializeKeysWith(RedisSerializationContext.SerializationPair
                        .fromSerializer(strSerializer))//设置缓存的key值为
strSerializer
                .serializeValuesWith(RedisSerializationContext.SerializationPair
                        .fromSerializer(jacksonSeial))//设置缓存的value为JSON格式序
列化对象
                .disableCachingNullValues();
        RedisCacheManager cacheManager = RedisCacheManager
                .builder(redisConnectionFactory).cacheDefaults(config).build();
        return cacheManager;
    }
```

自定义Redis注解和api方式的序列化方式总结：

两个都是有默认的序列化方式，api默认序列化的bean为redisTemplate，注解默认序列化的bean为cacheManager，我们自定义序列化方式只需要和它们创建一样的bean即可覆盖默认序列化方式（默认序列化使用@ConditionalOnMissingBean注解来实现以上的功能）。