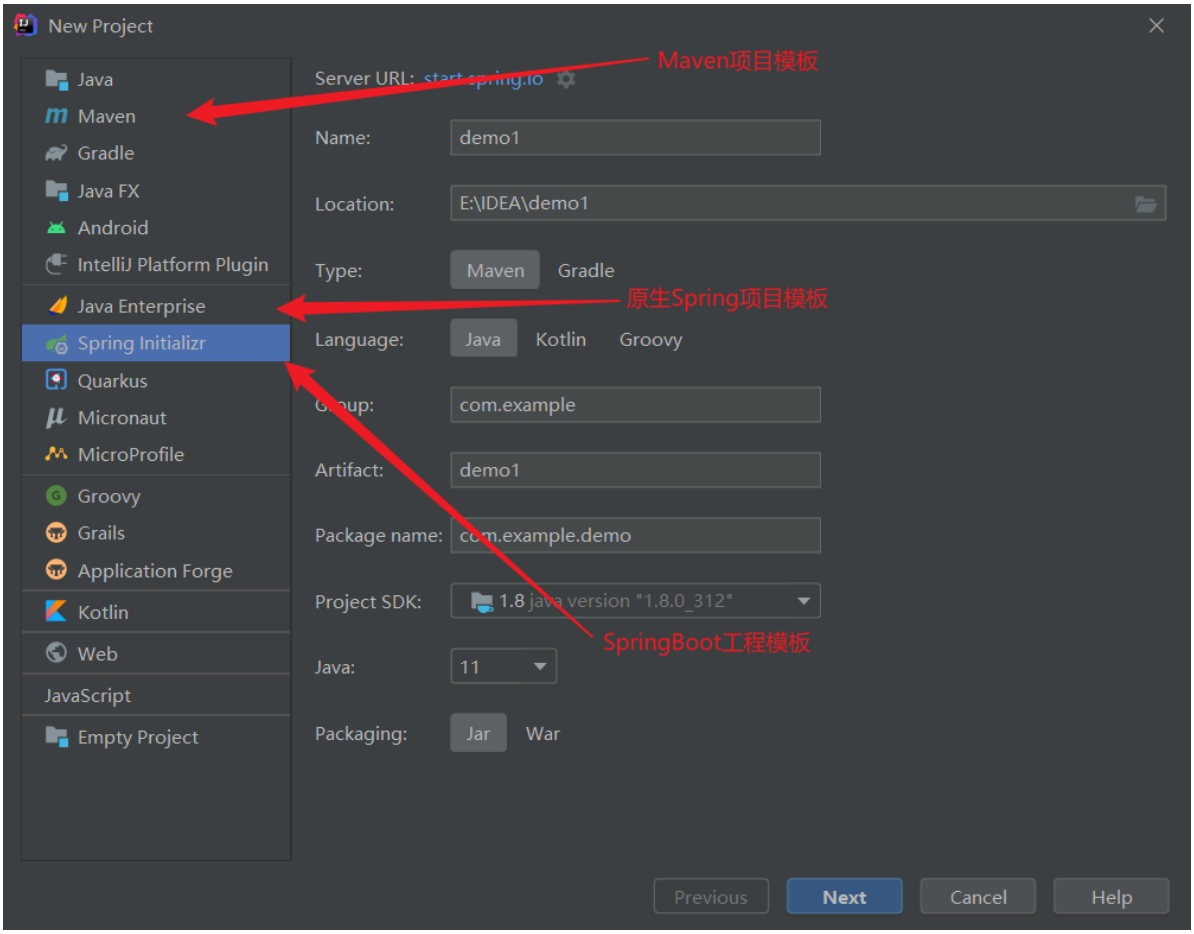
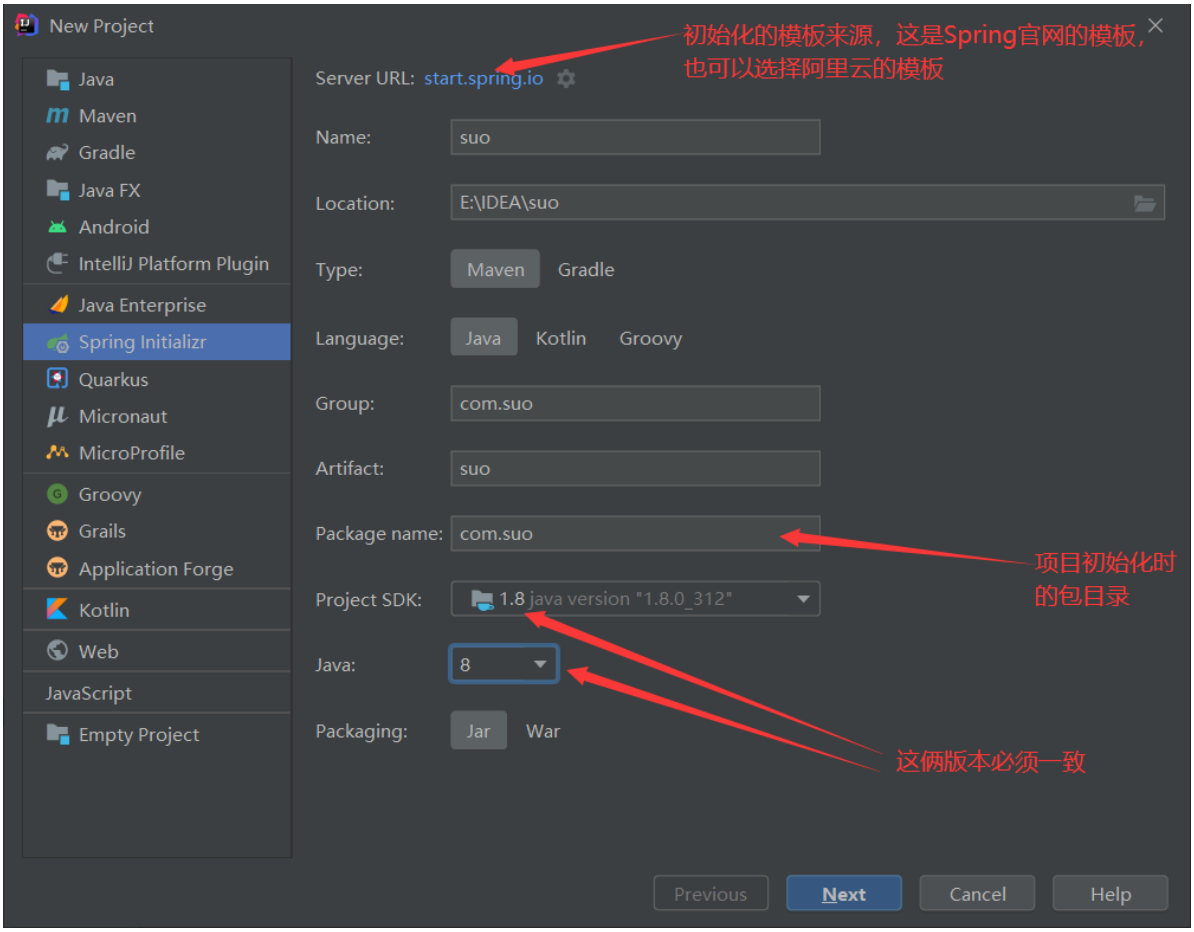
环境搭建

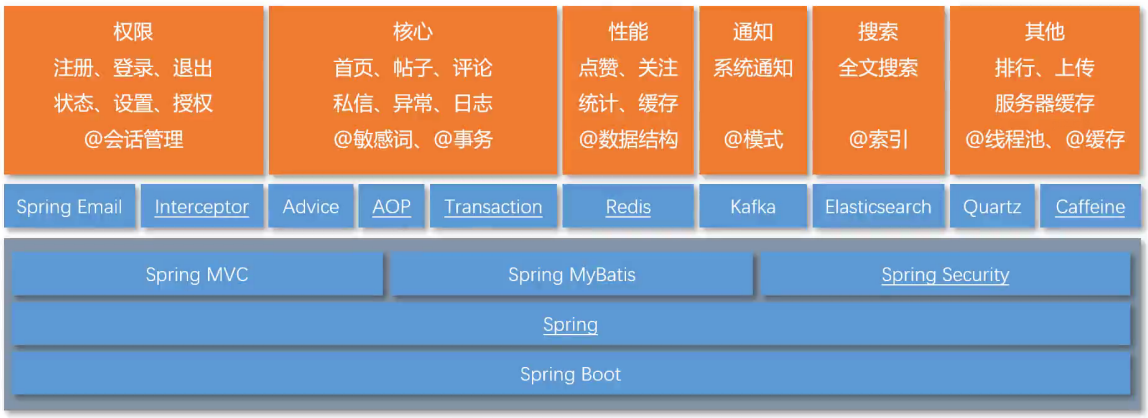
直接使用idea提供的模板进行创建工程



值得注意的是初始化模板的时候有几个自定义配置



架构



Spring介绍

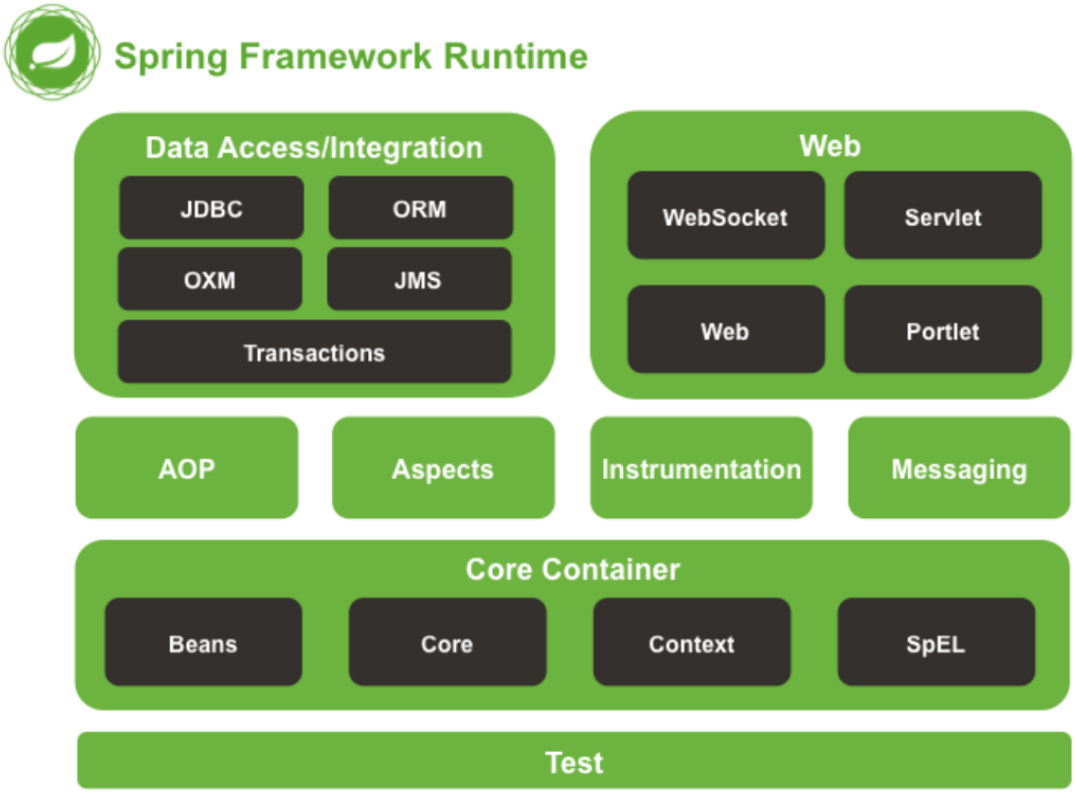
简介

Spring 是一款开源的轻量级 Java 开发框架，旨在提高开发人员的开发效率以及系统的可维护性。

其中 IoC（Inverse of Control:控制反转） 和 AOP(Aspect-Oriented Programming:面向切面编程)是Spring的两个核心思想

Spring的一些重要模块

下图对应的是 Spring4.x 版本。目前最新的 5.x 版本中 Web 模块的 Portlet 组件已经被废弃掉，同时增加了用于异步响应式处理的 WebFlux 组件。



Spring Core

核心模块， Spring 其他所有的功能基本都需要依赖于该类库，主要提供 IoC 依赖注入功能的支持。Spring以bean的方式组织和管理Java应用中的各个组件及其关系。Spring使用BeanFactory来产生和管理Bean，它是工厂模式的实现。BeanFactory使用控制反转(IoC)模式将应用的配置和依赖性规范与实际的应用程序代码分开

Spring Aspects

该模块为与 AspectJ 的集成提供支持。

Spring AOP

提供了面向切面的编程实现。

Spring Data Access/Integration：

Spring Data Access/Integration 由 5 个模块组成：

- spring-jdbc : 提供了对数据库访问的抽象 JDBC。不同的数据库都有自己独立的 API 用于操作数据库，而 Java 程序只需要和 JDBC API 交互，这样就屏蔽了数据库的影响。
- spring-tx : 提供对事务的支持。
- spring-orm : 对象关系映射，用来把对象模型表示的对象映射到基于SQL的关系模型数据库结构中去，程序中的对象与数据库通过这个桥梁进行相互转换。
- spring-oxm： O 代表 Object，X代表XML，M代表Mapping，目的是在Java对象和XML之间进行转换操作。
- spring-jms : Java Message Service Java 消息服务。

Spring Web

Spring Web 由 4 个模块组成：

- spring-web： 对 Web 功能的实现提供一些最基础的支持。
- spring-webmvc： 提供对 Spring MVC 的实现。

- spring-websocket： 提供了对 WebSocket 的支持，WebSocket 可以让客户端和服务端进行双向通信。
- spring-webflux： WebFlux 是 Spring Framework 5.0 中引入的新的响应式框架。

Spring Test

Spring 团队提倡测试驱动开发（TDD）。有了控制反转 (IoC)的帮助，单元测试和集成测试变得更简单。

Spring 的测试模块对 JUnit（单元测试框架）等常用的测试框架支持的都比较好。

优点

序号	好处	说明
1	轻量	Spring轻量的，基本的版本大约2MB。
2	控制反转	Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，不是创建或查找依赖的对象们。
3	面向切面编程	Spring支持面向切面的编程,把应业务逻辑和系统服务分开。
4	容器	Spring包含并管理应用中对象的生命周期和配置。
5	集成各种优秀框架	Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如MyBatis等）的直接支持。
6	支持声明式事务处理	只需要通过配置就可以完成对事物的管理，而无须手动编程。
7	方便程序的测试	Spring提供了对Junit4的支持，可以通过注解方便的测试Spring程序。

IoC（Inverse of Control:控制反转）

- 控制：对与JavaBean的创建（实例化、管理）的权力
- 反转：将控制权转交给Spring框架来管理而非程序员本身

IoC是一种设计思想而非一种技术，实现IoC思想的方法是依赖注入（Dependency Injection，简称 DI）

使用 IoC 的思想，我们将对象的控制权（创建、管理）交有 IoC 容器去管理，我们在使用的时候直接向 IoC 容器 “要” 就可以了

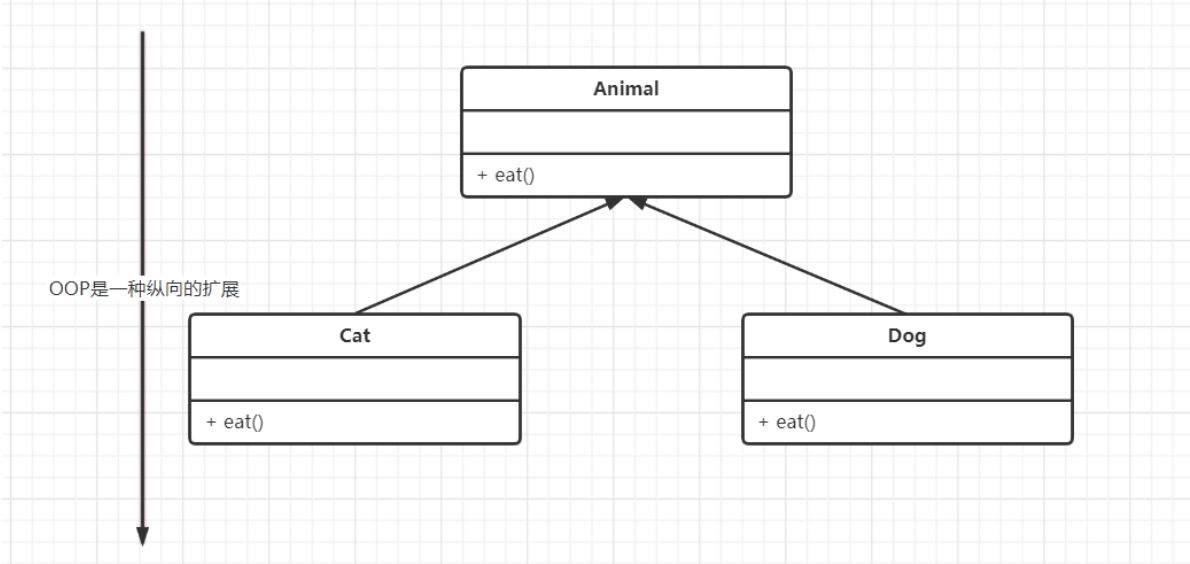
AOP(Aspect-Oriented Programming:面向切面编程)

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

AOP是对与OOP的一种延伸

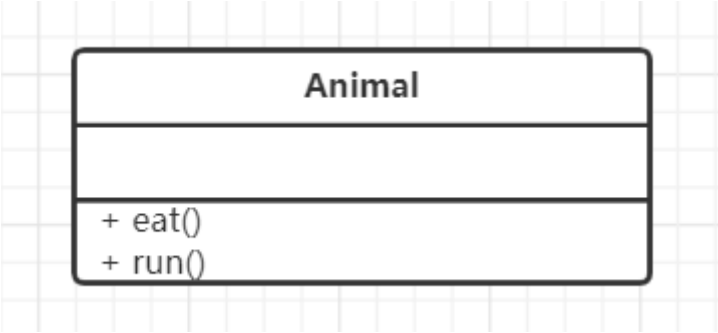
先以OOP举例：

假如有一个父类Animal有一个eat方法而其子类Cat、Dog继承了Animal那么Cat和Dog也会有eat方法，Java通过继承的方式使子类扩展了父类的方法，由下图可知这是一种纵向的扩展



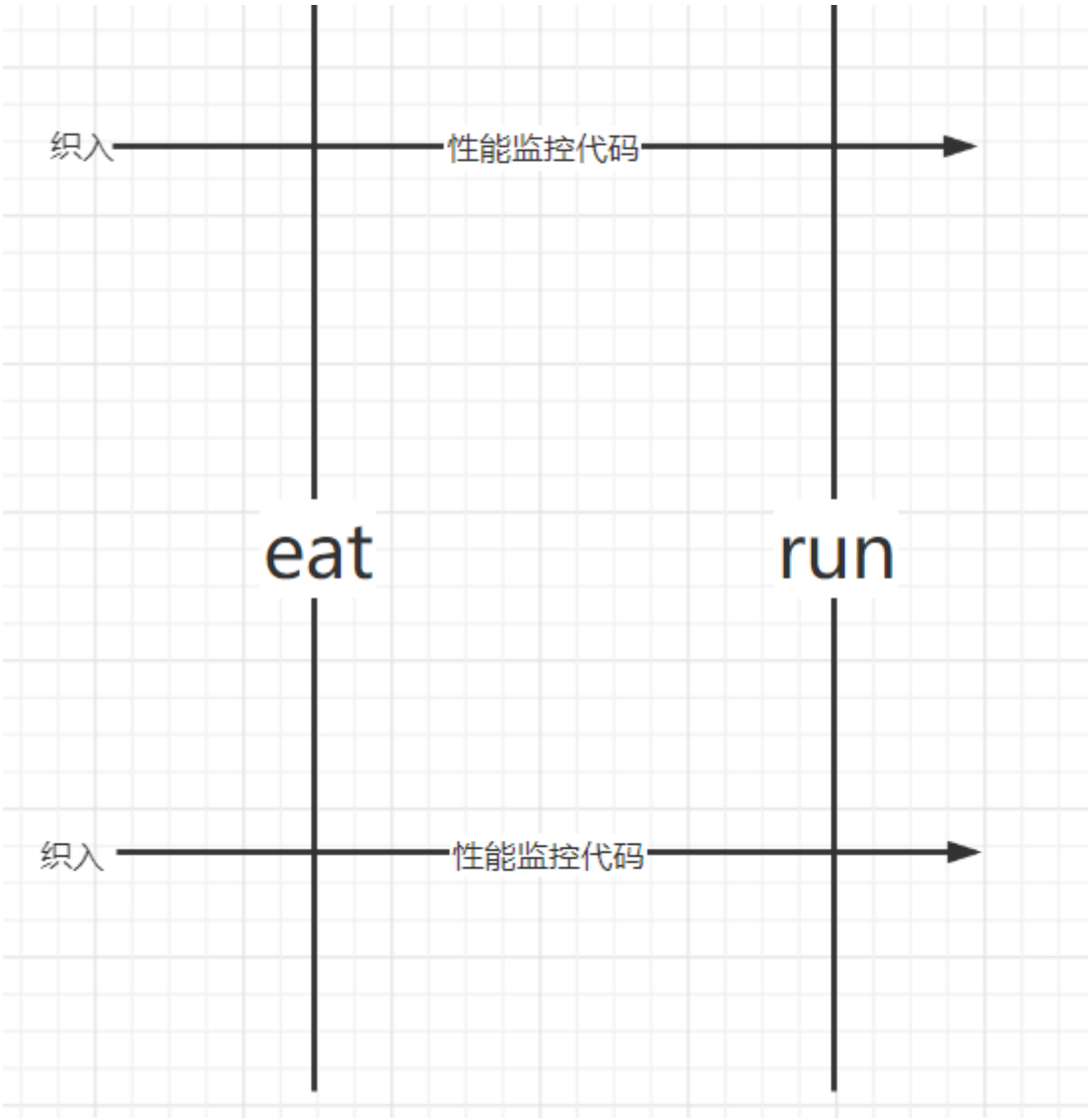
然后以AOP举例：

假如我想让Cat的eat方法和run方法同时加上性能监控代码、日志代码等这显然会产生大量相同的代码，这部分重复的代码，一般统称为 **横切逻辑代码**。AOP就是为了解决这个问题




```
public class Animal {

    public void eat() {
        // 性能监控代码
        long start = System.currentTimeMillis();

        // 业务逻辑代码
        System.out.println("I can eat...");

        // 性能监控代码
        System.out.println("执行时长：" + (System.currentTimeMillis() -
start)/1000f + "s");
    }

    public void run() {
        // 性能监控代码
        long start = System.currentTimeMillis();

        // 业务逻辑代码
        System.out.println("I can run...");

        // 性能监控代码
        System.out.println("执行时长：" + (System.currentTimeMillis() -
start)/1000f + "s");
```

```
    }
}
```



切： 指的是横切逻辑，原有业务逻辑代码不动，只能操作横切逻辑代码，所以面向横切逻辑

面： 横切逻辑代码往往要影响的是很多个方法，每个方法如同一个点，多个点构成一个面。这里有一个面的概念

这就是AOP为什么被称为面向切面编程

Spring AOP 就是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用**JDK Proxy**，去创建代理对象（基于Java多态的特性所以成立），而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候Spring AOP会使用**Cglib** ，这时候Spring AOP会使用 **Cglib** 生成一个被代理对象的子类来作为代理（基于Java继承的特性）。

这是代理模式的一种体现，当要调用需要被增强的方法时调用一个代理对象，而代理对象的该方法是已经被增强了，所以就达到了增强方法的作用。

Spring AOP 和 AspectJ AOP 的区别

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。 Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)。

Spring AOP 已经集成了 AspectJ，AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。AspectJ 相比于 Spring AOP 功能更加强大，但是 Spring AOP 相对来说更简单，

如果我们的切面比较少，那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ，它比 Spring AOP 快很多。

Spring Bean

JavaBean就是那些只有私有属性，和Getter和Setter方法的类

Spring Bean可以理解为由IoC容器管理的类

bean 的作用域

Spring 中 Bean 的作用域通常有下面几种：

- **singleton** : 唯一 bean 实例，Spring 中的 bean 默认都是单例的，对单例设计模式的应用。
- **prototype** : 每次请求都会创建一个新的 bean 实例。
- **request** : 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP request 内有效。
- **session** : 每一次来自新 session 的 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有效。
- **global-session** ： 全局 session 作用域，仅仅在基于 portlet 的 web 应用中才有意义，Spring5 已经没有了。Portlet 是能够生成语义代码(例如：HTML)片段的小型 Java Web 插件。它们基于 portlet 容器，可以像 servlet 一样处理 HTTP 请求。但是，与 servlet 不同，每个 portlet 都有不同的会话。

单例 bean 的线程安全问题

大部分时候我们并没有在项目中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为当多个线程操作同一个对象的时候是存在资源竞争的。

常见的有两种解决办法：

1. 在 bean 中尽量避免定义可变的成员变量。
2. **在类中定义一个 `ThreadLocal` 成员变量，将需要的可变成员变量保存在 `ThreadLocal` 中（推荐的一种方式）。**

不过，大部分 bean 实际都是无状态（没有实例变量）的（比如 Dao、Service），这种情况下， bean 是线程安全的。

例如我们可以用方法2来检测登录状态，显示登录信息，你可能有疑惑为啥不用session来存储User信息呢？因为服务器是要同时处理多个浏览器的请求的所有服务器会对不同的浏览器创建不同的线程，假如每个线程也就是浏览器都使用同一个bean那么就会产生问题。所以我门应该为一个线程创建一个bean

```
/**
 * 用于替换session
 */
@Component
public class HostHolder {

    private ThreadLocal<User> users = new ThreadLocal<>();

    public void setUser(User user) {
        users.set(user);
    }
```

```
    public User getUser(){
        return users.get();
    }

    public void clear() {
        users.remove();
    }
}
```

我们将User对象封装到ThreadLocal中，ThreadLocal为我们提供set和get方法可以将bean放入ThreadLocal和取出，我们来看一下源码

```
    public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
    }
```

其实很好理解，set方法就是先获取当前线程然后以以当前线程为key，要保存的bean为value存储到一个map当中。

```
    public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }
```

get方法就是以当前线程为key从map中取数据

@Component 和 @Bean 的区别是什么?

1. `@Component` 注解作用于类，而`@Bean`注解作用于方法。
2. `@Component`通常是通过类路径扫描来自动侦测以及自动装配到 Spring 容器中（我们可以使用`@ComponentScan` 注解定义要扫描的路径从中找出标识了需要装配的类自动装配到 Spring 的 bean 容器中）。`@Bean` 注解通常是我们在标有该注解的方法中定义产生这个 bean,`@Bean`告诉了Spring 这是某个类的实例，当我需要用它的时候还给我。
3. `@Bean` 注解比 `@Component` 注解的自定义性更强，而且很多地方我们只能通过 `@Bean` 注解来注册 bean。比如当我们引用第三方库中的类需要装配到 `Spring`容器时，则只能通过 `@Bean`来实现。

```
@Bean
public OneService getService(status) {
    case (status)  {
        when 1:
                return new serviceImpl1();
        when 2:
                return new serviceImpl2();
        when 3:
                return new serviceImpl3();
    }
}
```

Spring 框架中用到了哪些设计模式?

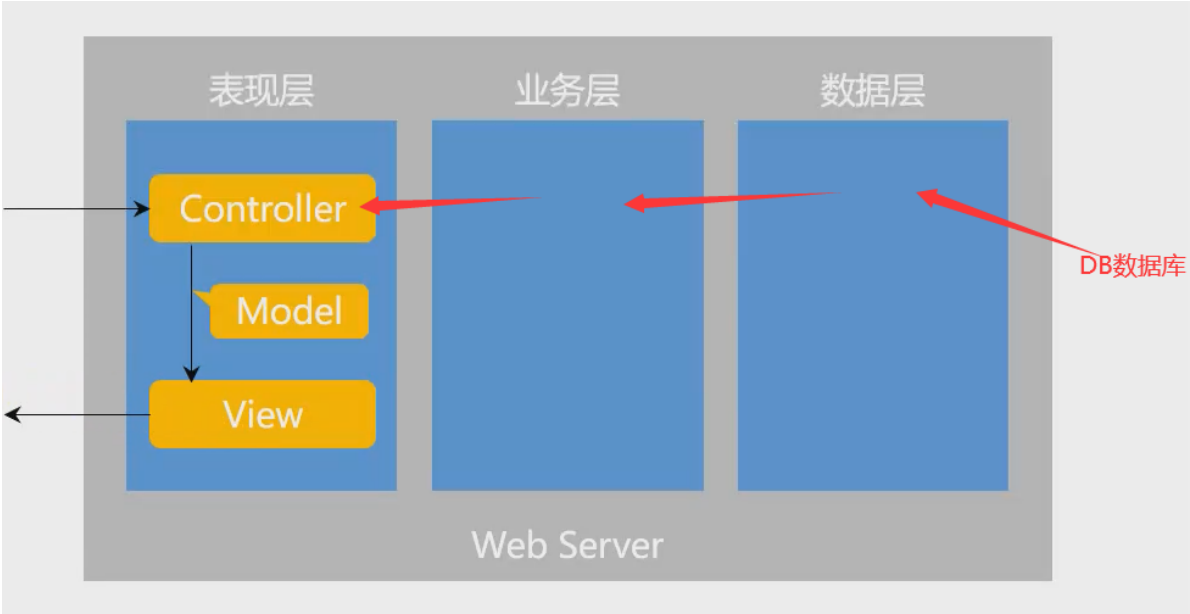
- **工厂设计模式** : Spring 使用工厂模式通过 `BeanFactory`、`ApplicationContext` 创建 bean 对象。
- **代理设计模式** : Spring AOP 功能的实现。
- **单例设计模式** : Spring 中的 Bean 默认都是单例的。
- **模板方法模式** : Spring 中 `jdbcTemplate`、`hibernateTemplate` 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。
- **包装器设计模式** : 我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。
- **观察者模式:** Spring 事件驱动模型就是观察者模式很经典的一个应用。
- **适配器模式** : Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配`Controller`。
- ……

SpringMVC的介绍

对于MVC的理解：

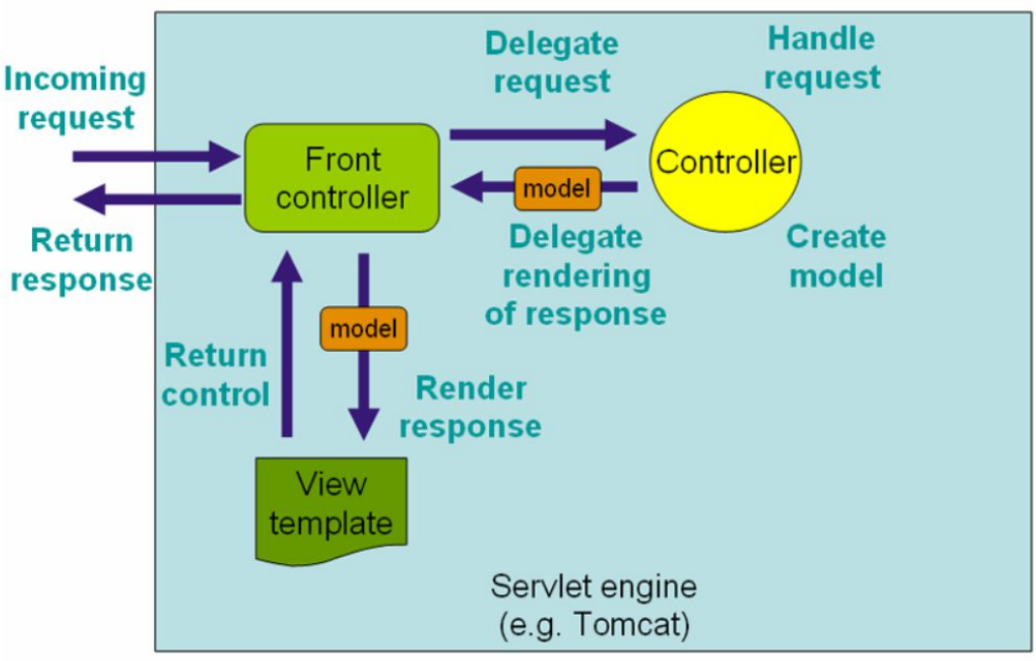
M代表Model，V代表View，C代表Control

MVC与三层架构是两个不一样的东西，一个是设计思想，一个是架构设计，所以其实两者是不能一一对应的SpringMVC主要是处理三层架构中View层的功能。



流程说明（重要）：

1. 客户端（浏览器）发送请求，直接请求到 `DispatcherServlet`。
2. `DispatcherServlet` 根据请求信息调用 `HandlerMapping`，解析请求对应的 `Handler`。
3. 解析到对应的 `Handler`（也就是我们平常说的 `Controller` 控制器）后，开始由 `HandlerAdapter` 适配器处理。
4. `HandlerAdapter` 会根据 `Handler` 来调用真正的处理器开处理请求，并处理相应的业务逻辑。
5. 处理器处理完业务后，会返回一个 `ModelAndView` 对象，`Model` 是返回的数据对象，`View` 是个逻辑上的 `View`。
6. `ViewResolver` 会根据逻辑 `View` 查找实际的 `View`。
7. `DispaterServlet` 把返回的 `Model` 传给 `View`（视图渲染）。
8. 把 `View` 返回给请求者（浏览器）



Spring 事务

事务是逻辑上的一组操作，要么都执行，要么都不执行。

事务的特性（ACID）

- **原子性（Atomicity）：** 一个事务（transaction）中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。即，事务不可分割、不可约简。
- **一致性（Consistency）：** 在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设约束、触发器、级联回滚等。
- **隔离性（Isolation）：** 数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括未提交读（Read uncommitted）、提交读（read committed）、可重复读（repeatable read）和串行化（Serializable）。
- **持久性（Durability）：**事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

Spring 管理事务的方式

- **编程式事务**： 在代码中硬编码(不推荐使用)：通过 `TransactionTemplate`或者 `TransactionManager` 手动管理事务，实际应用中很少使用，但是对于你理解 Spring 事务管理原理有帮助。
- **声明式事务**： 在 XML 配置文件中配置或者直接基于注解（推荐使用）：实际是通过 AOP 实现（基于`@Transactional` 的全注解方式使用最多）

Spring 事务中事务传播行为

事务传播行为是为了解决业务层方法之间互相调用的事务问题。

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

正确的事务传播行为可能的值如下:

1.`TransactionDefinition.PROPAGATION_REQUIRED`

使用的最多的一个事务传播行为，我们平时经常使用的`@Transactional`注解默认使用就是这个事务传播行为。如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。

`2.TransactionDefinition.PROPAGATION_REQUIRES_NEW`

创建一个新的事务，如果当前存在事务，则把当前事务挂起。也就是说不管外部方法是否开启事务，`Propagation.REQUIRES_NEW`修饰的内部方法会新开启自己的事务，且开启的事务相互独立，互不干扰。

3.`TransactionDefinition.PROPAGATION_NESTED`

如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于`TransactionDefinition.PROPAGATION_REQUIRED`。

4.`TransactionDefinition.PROPAGATION_MANDATORY`

如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）

这个使用的很少。

若是错误的配置以下 3 种事务传播行为，事务将不会发生回滚：

- `TransactionDefinition.PROPAGATION_SUPPORTS`: 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- `TransactionDefinition.PROPAGATION_NOT_SUPPORTED`: 以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- `TransactionDefinition.PROPAGATION_NEVER`: 以非事务方式运行，如果当前存在事务，则抛出异常。

Spring 事务中的隔离级别

和事务传播行为这块一样，为了方便使用，Spring 也相应地定义了一个枚举类：`Isolation`

```
public enum Isolation {

    DEFAULT(TransactionDefinition.ISOLATION_DEFAULT),

    READ_UNCOMMITTED(TransactionDefinition.ISOLATION_READ_UNCOMMITTED),

    READ_COMMITTED(TransactionDefinition.ISOLATION_READ_COMMITTED),

    REPEATABLE_READ(TransactionDefinition.ISOLATION_REPEATABLE_READ),

    SERIALIZABLE(TransactionDefinition.ISOLATION_SERIALIZABLE);

    private final int value;

    Isolation(int value) {
        this.value = value;
    }

    public int value() {
        return this.value;
    }

}
```

下面我依次对每一种事务隔离级别进行介绍：

- `TransactionDefinition.ISOLATION_DEFAULT` :使用后端数据库默认的隔离级别，MySQL 默认采用的 `REPEATABLE_READ` 隔离级别 Oracle 默认采用的 `READ_COMMITTED` 隔离级别.

- `TransactionDefinition.ISOLATION_READ_UNCOMMITTED`:最低的隔离级别，使用这个隔离级别很少，因为它允许读取尚未提交的数据变更，**可能会导致脏读、幻读或不可重复读**
- `TransactionDefinition.ISOLATION_READ_COMMITTED`：允许读取并发事务已经提交的数据，**可以阻止脏读，但是幻读或不可重复读仍有可能发生**
- `TransactionDefinition.ISOLATION_REPEATABLE_READ`：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生。**
- `TransactionDefinition.ISOLATION_SERIALIZABLE`：最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读**。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

@Transactional(rollbackFor = Exception.class)注解

`Exception` 分为运行时异常 `RuntimeException` 和非运行时异常。事务管理对于企业应用来说是至关重要的，即使出现异常情况，它也可以保证数据的一致性。

当 `@Transactional` 注解作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。如果类或者方法加了这个注解，那么这个类里面的方法抛出异常，就会回滚，数据库里面的数据也会回滚。

在 `@Transactional` 注解中如果不配置`rollbackFor`属性,那么事务只会在遇到`RuntimeException`的时候才会回滚，加上 `rollbackFor=Exception.class`,可以让事务在遇到非运行时异常时也回滚。

数据库表的梳理

总共5张表，

- comment：

```
CREATE TABLE `comment` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) DEFAULT NULL,
  `entity_type` int(11) DEFAULT NULL,
  `entity_id` int(11) DEFAULT NULL,
  `target_id` int(11) DEFAULT NULL,
  `content` text,
  `status` int(11) DEFAULT NULL,
  `create_time` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `index_user_id` (`user_id`) /*!80000 INVISIBLE */,
  KEY `index_entity_id` (`entity_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

评论表，其中user_id用于表示发这条评论的用户id

entity_type表示评论的类型 1表示回复帖子的评论　2表示回复评论的评论

entity_id表示该评论的帖子id

target_id表示回复时回复对象的id，如果id=0说明这是一条回复帖子的评论，如果id!=0说明这是回复target_id用户的评论

content评论内容

status评论的状态0表示有用的评论 1表示以删除的评论

create_time评论的发表时间

- discuss_post:

```
CREATE TABLE `discuss_post` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` varchar(45) DEFAULT NULL,
  `title` varchar(100) DEFAULT NULL,
  `content` text,
  `type` int(11) DEFAULT NULL COMMENT '0-普通; 1-置顶;',
  `status` int(11) DEFAULT NULL COMMENT '0-正常; 1-精华; 2-拉黑;',
  `create_time` timestamp NULL DEFAULT NULL,
  `comment_count` int(11) DEFAULT NULL,
  `score` double DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `index_user_id` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

帖子表，

user_id表示发帖人的id

title表示标题

content表示帖子内容

type表示帖子类型 0表示普通帖子 1表示置顶帖子

status表示帖子的状态 0表示正常帖子 1表示精华帖 2表示拉黑帖

create_time表示发帖日期

comment_count表示帖子的评论数，是一个冗余参数，目的是为了提供查询效率

score

- login_ticket:

```
CREATE TABLE `login_ticket` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) NOT NULL,
  `ticket` varchar(45) NOT NULL,
  `status` int(11) DEFAULT '0' COMMENT '0-有效; 1-无效;',
  `expired` timestamp NOT NULL,
  PRIMARY KEY (`id`),
  KEY `index_ticket` (`ticket`(20))
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

登录凭证表，目的是为了保护用户数据的安全，用于检测用户是否处于登录状态，然后在启动服务器时通过拦截器获取登录凭证创建User对象

user_id用户id

ticket凭证编号

status凭证状态 0表示有效凭证 1表示无效凭证

expired用于检测这个凭证是否已经过期

- message:

```
CREATE TABLE `message` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `from_id` int(11) DEFAULT NULL,
  `to_id` int(11) DEFAULT NULL,
  `conversation_id` varchar(45) NOT NULL,
  `content` text,
  `status` int(11) DEFAULT NULL COMMENT '0-未读;1-已读;2-删除;',
  `create_time` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `index_from_id` (`from_id`),
  KEY `index_to_id` (`to_id`),
  KEY `index_conversation_id` (`conversation_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

消息表

from_id是发消息的人的id

to_id是发送目标的人的id

conversation_id也是一个冗余的字段目的是便于查询，规则以from_id和to_id用_拼接且小的在前大的在后

content消息内容

status消息的状态 0表示未读 1表示已读 2表示删除

create_time消息的发送时间

- user：

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(50) DEFAULT NULL,
  `password` varchar(50) DEFAULT NULL,
  `salt` varchar(50) DEFAULT NULL,
  `email` varchar(100) DEFAULT NULL,
  `type` int(11) DEFAULT NULL COMMENT '0-普通用户; 1-超级管理员; 2-版主;',
  `status` int(11) DEFAULT NULL COMMENT '0-未激活; 1-已激活;',
  `activation_code` varchar(100) DEFAULT NULL,
  `header_url` varchar(200) DEFAULT NULL,
  `create_time` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `index_username` (`username`(20)),
  KEY `index_email` (`email`(20))
) ENGINE=InnoDB AUTO_INCREMENT=101 DEFAULT CHARSET=utf8;
```

用户表

username用户名称

password用户密码

salt盐用来加密

email用户的邮箱

type用户的类型 0表示普通用户 1表示超级管理员 2表示版主

status 0表示未激活 1表示激活

activation_code激活码

header_url头像地址

create_time用户注册时间

业务梳理

首先业务就是先思考前端需要展示什么什么数据，然后在考虑后端需要什么数据进行处理才能得到前端所需要的数据，最后才是考虑如何处理才能得到目标数据（**也就是考虑**`条件`**是什么，**`目标`**是什么**）

后端一定要有一个意识就是程序的完整性，上来二话不说先判断前端返回的数据是否为空，如果为空要进行什么样的处理

开发首页的业务梳理

> 开发首页需要返回给前端什么数据?

1. 帖子的标题
2. 发帖人的头像
3. 发帖人的名称
4. 发帖的时间

仅此而已

> 后端需要什么样的数据进行处理才能得到这些数据呢?

显然这些数据都与帖子有关所以我们只需要查询帖子表然后将帖子表数据一一展示即可

> 为了减小服务器和浏览器的压力还得要进行分页处理

分页其实很简单有个叫做pageHelper的插件能轻松解决问题

注册功能业务梳理

> 前端需要什么数据?

注册功能前端其实不需要什么数据，或者说需要一个反馈，比如注册成功，注册失败，用户名已经存在，邮箱名已经存在等等

> 后端需要什么数据?

这是一个表单，后端需要的数据就是前端表单里的所有数据咯，非常的无脑简单

首先要考虑的业务功能是当鼠标离开账号框时就进行检测是否账号已存在，这个技术可以采用ajax的技术来实现，前端当触发离开文本框时间时，调用后端查数据库的操作，后端将bool值异步返回给前端进行展示

然后就是邮箱的问题了，先进行和上述一样的操作，然后全部没问题之后进行后端service的操作，将表单数据封装成一个User，然后将Userinsert到数据库中，注意此时的user的激活状态处于未激活状态，并且提供一个激活码，然后发送一封邮件给用户用于激活账号，这封邮件中包含一个超链接，这个超链接中跳转的地址携带一个激活码的参数和用户id，后端获取这个参数通过数据库进行比对如果激活码一致则激活账号，返回相应页面，否则直接返回相应页面并提供错误信息

发送邮件的功能实现

发送邮件的功能是为注册功能做准备的，发送邮件用到了Spring Email的技术

来先简单介绍一下该技术

首先进行配置文件，配置文件的目的是告诉Spring需要用哪个邮箱来发送邮件

```yaml
spring:
  mail:
    host: smtp.qq.com
    port: 465
    username: ***@qq.com
    password: ***
    protocol: smtps
    properties:
      mail.smtp.auth: true
      mail.smtp.starttls.enable: true
      mail.smtp.starttls.required: true
      mail.smtp.socketFactory.port: 465
      mail.smtp.socketFactory.class: javax.net.ssl.SSLSocketFactory
      mail.smtp.socketFactory.fallback: false
```

首先是JavaMailSender提供了一个发送邮件send的方法

send方法的参数需要一个MimeMessage进行封装，封装的参数是邮件的发送方，邮件的接受方，邮件的主题，邮件的内容

而MimeMessage不方便进行封装所以Spring又提供了一个类MimeMessageHelper进行封装

MimeMessage通过JavaMailSender的createMimeMessage方法得到

MimeMessageHelper通过MimeMessageHelper的有参构造得到参数就是MimeMessage

再将MimeMessage通过MimeMessageHelper的getMimeMessage方法得到

最后将MimeMessage作为参数传递给send

```java
@Component
public class MailClient {
    private static final Logger logger =
LoggerFactory.getLogger(MailClient.class);

    @Autowired
    private JavaMailSender mailSender;

    @Value("${spring.mail.username}")
    private String from;

    public void sendMail(String to, String subject, String content) {
        try {
            MimeMessage mimeMessage = mailSender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(mimeMessage);
            helper.setFrom(from);
            helper.setTo(to);
            helper.setText(content,true);//表示可以发送HTML文件
            helper.setSubject(subject);
            mailSender.send(helper.getMimeMessage());
        } catch (MessagingException e) {
```

```
            logger.error("发送邮件失败：" + e.getMessage());
        }

    }
}
```

如果想要发送HTML文件的话 只需要将helper.setText(content,true)方法参数设置为true即可

如果想要返回一个动态渲染后的HTML的话thymeleaf提供了一个模板引擎templateEngine

```
Context context = new Context();
context.setVariable("username","test");

String process = templateEngine.process("/mail/demo", context);
```

我们只需要new一个Context对象然后将需要动态生成的参数以key：value的形式通过setVariable方法传参即可

最后指定你要发送的HTML路径templateEngine的process方法会自动帮你完成任务

登录与退出功能的业务梳理

登录和退出功能就太简单了

> 前端需要什么数据?

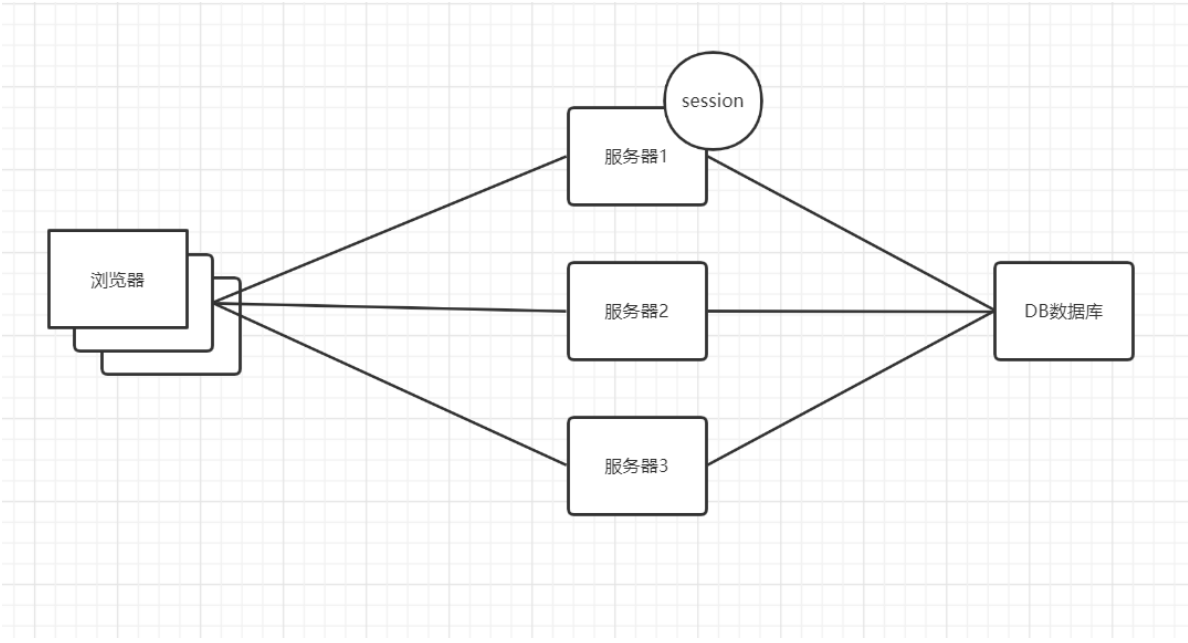
nothing 或者提供一些交互信息

> 后端需要什么数据

就是表单数据

登录这个功能还是值得玩味的，传统的做法是直接查询user表获取user对象然后将user对象放入session中，然后之后的一切操作全部基于这个session。

好我们现在来思考一下这个方案有什么问题



如果现在是这么一个状态，现在有多个服务器来处理浏览器请求，那么就会出现这么一个现象一开始是服务器1处理浏览器请求然后下一个请求用服务器2来处理数据此时服务器2没有session对象那么此时服务器就会报错或者重新创建一个新的session对象，这样不仅效率低还浪费资源。不要跟我讲可以用cookie来处理，用cookie的数据是十分不安全的！！！

> 解决方案

我们新建了一个表叫做凭证表，诶！！！把凭证放到cookie中是不是就安全啦，因为我只放了一个凭证的号码到cookie中其他私密信息都没放到浏览器中

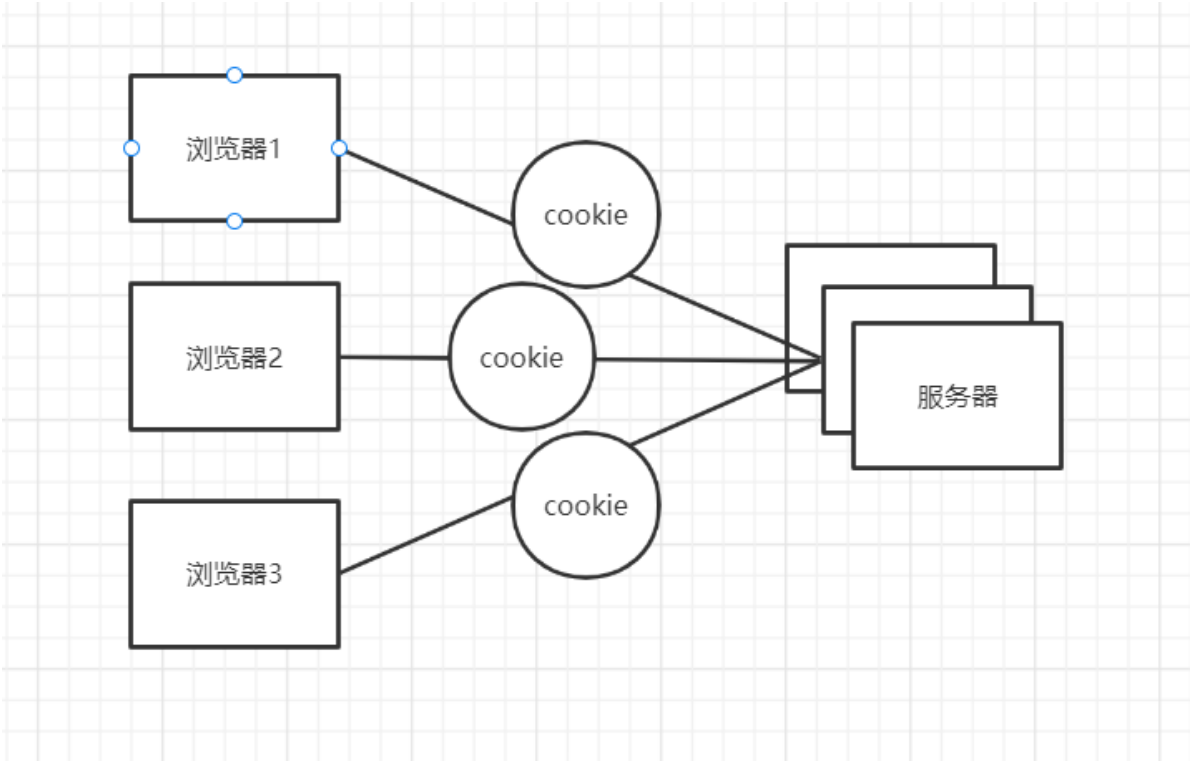
好此时你可能会想这么操作是不是有点浪费资源了，并且效率还不高所以我们采用Redis进行优化

Redis最重要就是对于Key的设计不过这个业务比较无脑直接以凭证为key再加个前缀就好了

```
    private static final String SPLIT = ":";
    private static final String PREFIX_TICKET = "ticket";
    // 登录的凭证
    public static String getTicketKey(String ticket) {
        return PREFIX_TICKET + SPLIT + ticket;
    }
```

现在的登录操作就很有趣了，首先第一次登录的时候服务器会发给浏览器一个cookie里面放了个凭证，然后之后登录都是通过拦截器获取这个凭证，再通过这个凭证查询Redis数据库获取userid再通过这个userid查询数据库获得user

好接下来我们再考虑一种情况



此时有多个浏览器同时发给服务器cookie那么此时服务器创建的user对象到底是哪个cookie呢？？？

这就涉及到单例 bean 的线程安全问题

大部分时候我们并没有在项目中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为当多个线程操作同一个对象的时候是存在资源竞争的。

常见的有两种解决办法：

1. 在 bean 中尽量避免定义可变的成员变量。

2. **在类中定义一个 `ThreadLocal` 成员变量，将需要的可变成员变量保存在 `ThreadLocal` 中（推荐的一种方式）。**

不过，大部分 bean 实际都是无状态（没有实例变量）的（比如 Dao、Service），这种情况下， bean 是线程安全的。

例如我们可以用方法2来检测登录状态，显示登录信息，你可能有疑惑为啥不用session来存储User信息呢？因为服务器是要同时处理多个浏览器的请求的所有服务器会对不同的浏览器创建不同的线程，假如每个线程也就是浏览器都使用同一个bean那么就会产生问题。所以我门应该为一个线程创建一个bean

```
/**
 * 用于替换session
 */
@Component
public class HostHolder {

    private ThreadLocal<User> users = new ThreadLocal<>();

    public void setUser(User user) {
        users.set(user);
    }

    public User getUser(){
        return users.get();
    }

    public void clear() {
        users.remove();
    }
}
```

我们将User对象封装到ThreadLocal中，ThreadLocal为我们提供set和get方法可以将bean放入ThreadLocal和取出，我们来看一下源码

```
    public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
    }
```

其实很好理解，set方法就是先获取当前线程然后以以当前线程为key，要保存的bean为value存储到一个map当中。

```
    public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }
```

get方法就是以当前线程为key从map中取数据

这样就可以解决这个问题了，每个线程一个user，然后我们再创建一个拦截器，当访问服务器时就将user对象set到ThreadLocal中，需要的时候get一下就好了

SpringMVC拦截器

我们来复习一下SpringMVC的拦截器技术

步骤：

1. 创建拦截器

```
@Component
public class LoginTicketInterceptor implements HandlerInterceptor {

    @Autowired
    private UserService userService;

    @Autowired
    private HostHolder hostHolder;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        //从cookie中获取凭证
        String ticket = CookieUtil.getValue(request, "ticket");

        if(ticket != null) {
            LoginTicket loginTicket = userService.findLoginTicket(ticket);
            if(loginTicket != null && loginTicket.getStatus() == 0 &&
loginTicket.getExpired().after(new Date())) {
                User user = userService.findUserById(loginTicket.getUserId());
                hostHolder.setUser(user);
            }
        }

        return true;
    }

    @Override
```

```
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        User user = hostHolder.getUser();
        if(user != null && modelAndView!= null) {
            modelAndView.addObject("loginUser",user);
        }
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        hostHolder.clear();
    }
}
```

拦截器必须继承HandlerInterceptor然后重写三个方法

- preHandle　　在目标方法执行前执行
- postHandle　　在目标方法执行之后，视图未返回前执行
- afterCompletion　在整个流程执行完毕后执行

2. 创建配置类

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
    @Autowired
    private LoginTicketInterceptor loginTicketInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {

 registry.addInterceptor(loginTicketInterceptor).excludePathPatterns("/**/*.css"
, "/**/*.js", "/**/*.png", "/**/*.jpg", "/**/*.jpeg");

    }
}
```

- addInterceptor：添加拦截器
- excludePathPatterns：将一些资源进行除外，不进行拦截

回显登录信息的业务梳理

> 前端需要什么数据?

user对象

> 后端需要什么数据?

cookie对象

所以其实我们只需要通过cookie来创建user对象就好了，怎么得到呢？上面已经解决了直接ThreadLocal中get一下就好了

修改个人信息的业务梳理

修改信息分为两部分1. 修改头像 2. 修改密码

当然修改前要先判断用户是否已经登录了，其实也有简单粗暴的方法就是，给每个需要检验用户是否登录的方法都进行判断，但这太low了，上面提到了一个拦截器的方法可以用拦截器来实现此功能。用于拦截未登录状态时防止用户进行操作修改数据库

这时又出现了一些问题，那就是拦截器的配置，当你每需要创建一个要被拦截的类时，就要修改拦截器源码，或者些不需要拦截的类时也要修改源码，排除这些类，这时你会想这岂不是比暴力法还麻烦。这里提供一个注解的方法，自己写一个注解，当被这个注解标注时，就会被这个拦截器拦截

元注解一共有四个

- @Target注解 Target注解的作用是：描述注解的使用范围（即：被修饰的注解可以用在什么地方）。
- @Retention注解　Reteniton注解的作用是：描述注解保留的时间范围（即：被描述的注解在它所修饰的类中可以被保留到何时）。
- @Documented注解 Documented注解的作用是：描述在使用 javadoc 工具为类生成帮助文档时是否要保留其注解信息。
- @Inherited注解　Inherited注解的作用是：使被它修饰的注解具有继承性（如果某个类使用了被@Inherited修饰的注解，则其子类将自动具有该注解）。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LoginRequired {
}
```

然后我们只需要在需要拦截的方法上添加这个注解就好了

接下来就是编写拦截器了

```
@Component
public class LoginRequiredInterceptor implements HandlerInterceptor {

    @Autowired
    private HostHolder hostHolder;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        if (handler instanceof HandlerMethod) {
            HandlerMethod handlerMethod = (HandlerMethod) handler;
            Method method = handlerMethod.getMethod();
            LoginRequired annotation =
method.getAnnotation(LoginRequired.class);
            if (annotation != null && hostHolder.getUser() == null) {
                response.sendRedirect(request.getContextPath() + "/login");
                return false;
            }
        }
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        HandlerInterceptor.super.postHandle(request, response, handler,
modelAndView);
```

```
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        HandlerInterceptor.super.afterCompletion(request, response, handler,
ex);
    }
}
```

最后配置类修改一下

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
    @Autowired
    private LoginTicketInterceptor loginTicketInterceptor;

    @Autowired
    private LoginRequiredInterceptor loginRequiredInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {

 registry.addInterceptor(loginTicketInterceptor).excludePathPatterns("/**/*.css"
, "/**/*.js", "/**/*.png", "/**/*.jpg", "/**/*.jpeg");

 registry.addInterceptor(loginRequiredInterceptor).excludePathPatterns("/**/*.cs
s", "/**/*.js", "/**/*.png", "/**/*.jpg", "/**/*.jpeg");
    }
}
```

过滤完后可以正式进入修改信息的主题了

首先是修改头像也就是上传图片的功能

这里涉及SpringMVC文件上传的功能

SpringMVC文件上传

文件上传三要素

1. 表单项type="file"
2. 表单的提交方式是post
3. 表单的enctype="multipart/form-data"

当满足这三个条件时上传文件SpringMVC为我们提供了一个参数MultipartFile，然后这个对象为我们提供了一个方法transferTo（File（url））

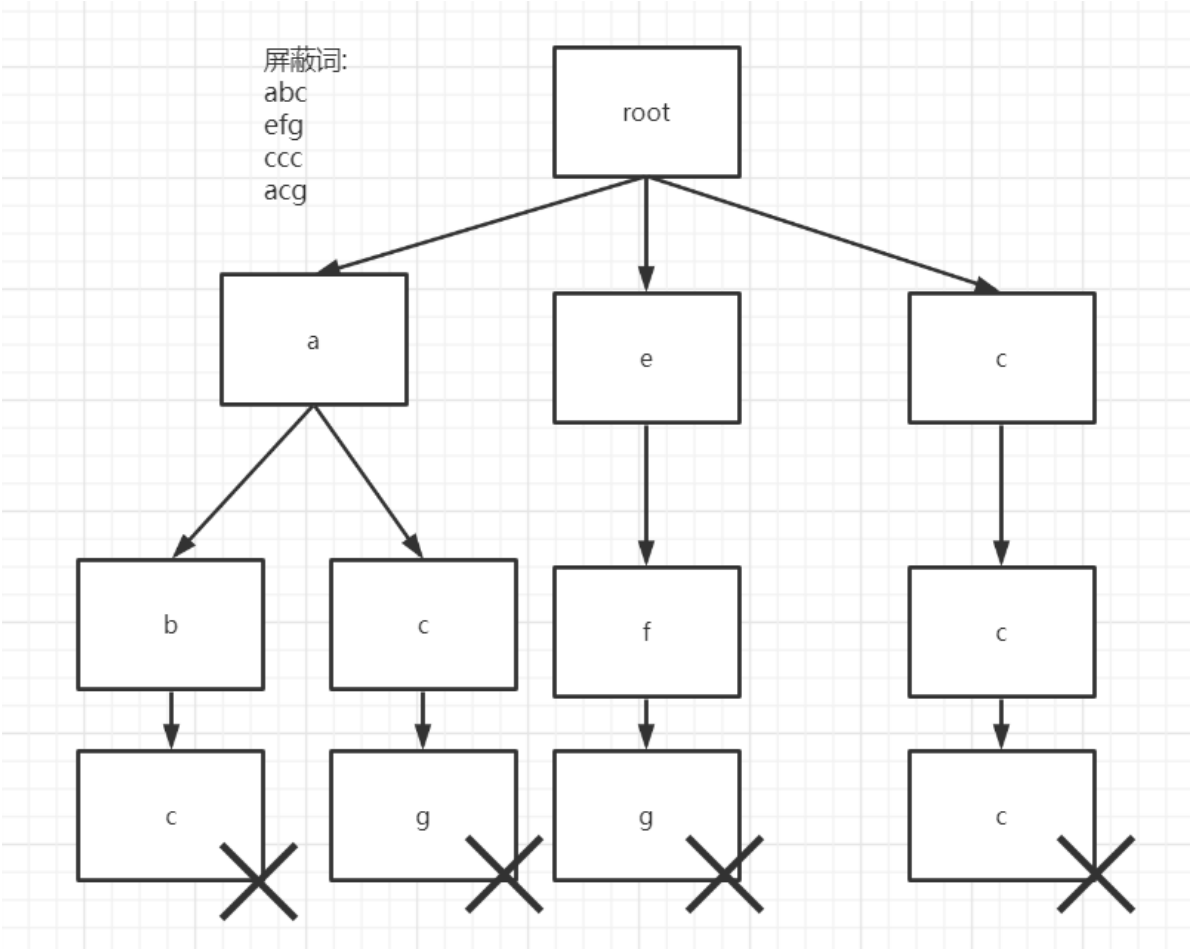
```
@LoginRequired
@RequestMapping(path = "/upload", method = RequestMethod.POST)
public String uploadHeader(MultipartFile headerImage, Model model) {
    //判断上传文件是否为空
    if (headerImage == null) {
        model.addAttribute("error", "您还没有选择图片!");
        return "/site/setting";
```

```
    }

    //获取文件名的后缀
    String fileName = headerImage.getOriginalFilename();
    String suffix = fileName.substring(fileName.lastIndexOf("."));
    if (StringUtils.isBlank(suffix)) {
        model.addAttribute("error", "文件的格式不正确!");
        return "/site/setting";
    }

    // 生成随机文件名
    fileName = CommunityUtil.generateUUID() + suffix;
    // 确定文件存放的路径
    File dest = new File(uploadPath + "/" + fileName);

    try {
        //转存文件
        headerImage.transferTo(dest);
    } catch (IOException e) {
        logger.error("上传文件失败: " + e.getMessage());
        throw new RuntimeException("上传文件失败,服务器发生异常!", e);
    }

    // 更新当前用户的头像的路径(web访问路径)
    User user = hostHolder.getUser();
    String headerUrl = domain + contextPath + "/user/header/" + fileName;
    userService.updateHeader(user.getId(), headerUrl);

    return "redirect:/index";
}
```

现在我们来梳理业务逻辑:

为了防止上传重复名的文件导致文件被覆盖所以我们采用生成随机字符串的方式来解决这个问题，所以我们要获得文件的后缀然后拼上随机字符串。最后调用transferTo方法进行转存就好了

过滤敏感词



过滤铭感词这个业务分为两部分（众所周知程序=数据结构+算法）

1. 敏感词应该用什么数据结构来存储?
2. 如果辨别一个字符串时敏感词，也就是说采用什么算法来过滤敏感词?

第一个问题：我对于敏感词采用树的结构来存储，值得注意的时这个树还得要一个boolean来记录这个节点是否时最后一个节点，树有常见的表示发有双亲表示法，孩子兄弟表示法，孩子表示法，我这里采用的是孩子表示法

```
// 前缀树
private class TrieNode {

    // 关键词结束标识
    private boolean isKeywordEnd = false;

    // 子节点(key是下级字符,value是下级节点)
    private Map<Character, TrieNode> subNodes = new HashMap<>();

    public boolean isKeywordEnd() {
        return isKeywordEnd;
    }

    public void setKeywordEnd(boolean keywordEnd) {
        isKeywordEnd = keywordEnd;
    }

    // 添加子节点
    public void addSubNode(Character c, TrieNode node) {
        subNodes.put(c, node);
    }
```

```
    // 获取子节点
    public TrieNode getSubNode(Character c) {
        return subNodes.get(c);
    }

}
```

然后树的初始化操作

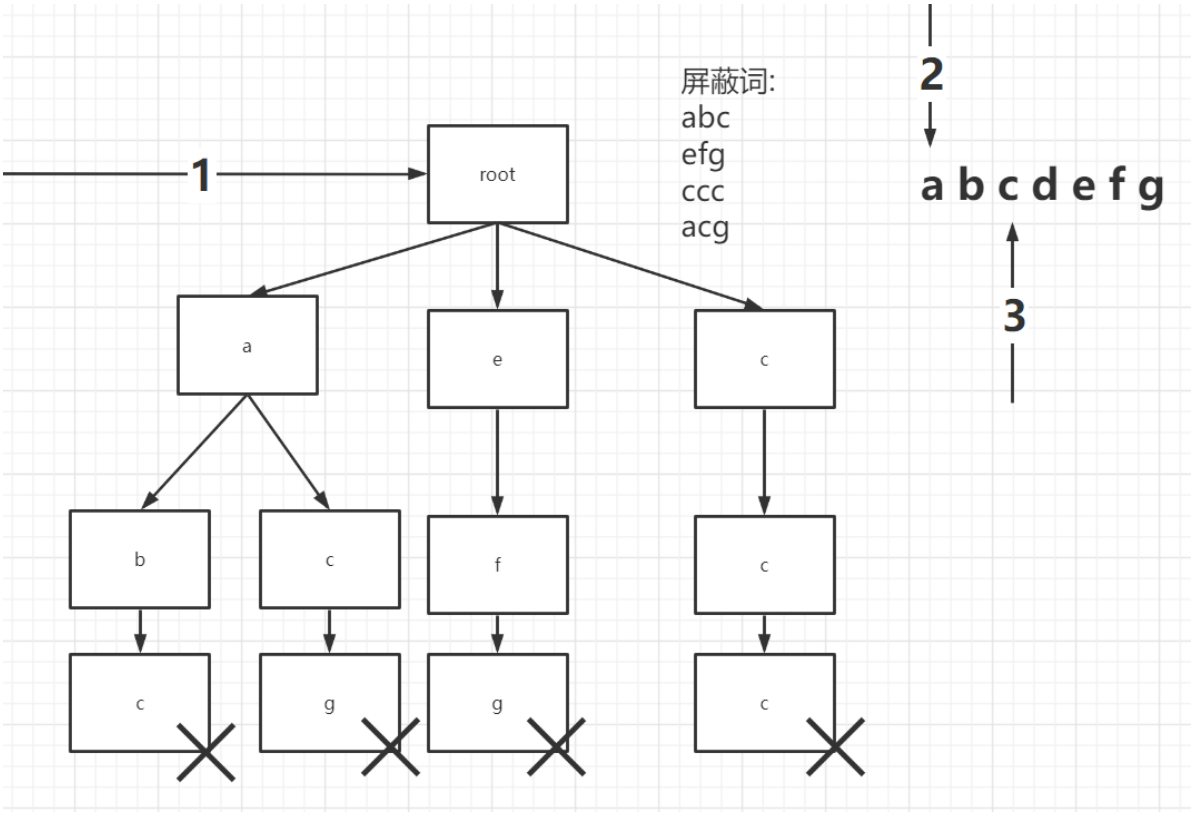
```
@PostConstruct
public void init() {
    try (
            InputStream is =
this.getClass().getClassLoader().getResourceAsStream("sensitive-words.txt");
            BufferedReader reader = new BufferedReader(new
InputStreamReader(is));
    ) {
        String keyword;
        while ((keyword = reader.readLine()) != null) {
            // 添加到前缀树
            this.addKeyword(keyword);
        }
    } catch (IOException e) {
        logger.error("加载敏感词文件失败: " + e.getMessage());
    }
}

// 将一个敏感词添加到前缀树中
private void addKeyword(String keyword) {
    TrieNode tempNode = rootNode;
    for (int i = 0; i < keyword.length(); i++) {
        char c = keyword.charAt(i);
        TrieNode subNode = tempNode.getSubNode(c);

        if (subNode == null) {
            // 初始化子节点
            subNode = new TrieNode();
            tempNode.addSubNode(c, subNode);
        }

        // 指向子节点,进入下一轮循环
        tempNode = subNode;

        // 设置结束标识
        if (i == keyword.length() - 1) {
            tempNode.setKeywordEnd(true);
        }
    }
}
```

第二个问题采用什么算法?

我这里使用三个指针的方式，第一个指针指向树，第二个指针指向要判断字符串的头，第三个指针指向这个字符串的结尾。

> 思路:

> 首先起始指针与树进行比对观察是否有匹配的值如果没有则2，3两个指针向后移动一位，如果有则1， 3指针向后移动一位，然后进行判断3指针是否有匹配的节点如果有则1，3向后移动一位并且判断此节点是否为最后一位节点如果是则屏蔽。然后1节点指回root，3++，2=3；如果中途发现没有匹配的节点了说明这不是敏感词，2++，3=2；




```
/**
 * 过滤敏感词
 *
 * @param text 待过滤的文本
 * @return 过滤后的文本
 */
public String filter(String text){
    if(StringUtils.isBlank(text)){
        return null;
    }
    // 指针1
    TrieNode tempNode = rootNode;
    // 指针2
    int begin = 0;
    // 指针3
    int position = 0;
    // 结果
    StringBuilder sb = new StringBuilder();

    while(begin < text.length()){
        if(position < text.length()) {
            Character c = text.charAt(position);

            // 跳过符号
            if (isSymbol(c)) {
                if (tempNode == rootNode) {
                    begin++;
                    sb.append(c);
                }
```

```
                position++;
                continue;
            }

            // 检查下级节点
            tempNode = tempNode.getSubNode(c);
            if (tempNode == null) {
                // 以begin开头的字符串不是敏感词
                sb.append(text.charAt(begin));
                // 进入下一个位置
                position = ++begin;
                // 重新指向根节点
                tempNode = rootNode;
            }
            // 发现敏感词
            else if (tempNode.isKeywordEnd()) {
                sb.append(REPLACEMENT);
                begin = ++position;
                // 重新指向根节点
                tempNode = rootNode;
            }
            // 检查下一个字符
            else {
                position++;
            }
        }
        // position遍历越界仍未匹配到敏感词
        else{
            sb.append(text.charAt(begin));
            position = ++begin;
            tempNode = rootNode;
        }
    }
    return sb.toString();
}

// 判断是否为符号
private boolean isSymbol(Character c) {
    // 0x2E80~0x9FFF 是东亚文字范围
    return !CharUtils.isAsciiAlphanumeric(c) && (c < 0x2E80 || c > 0x9FFF);
}
```

发布帖子功能的业务梳理

这个功能就很无脑了，还是老样子先思考条件是什么目标是什么

> 前端需要什么样的数据?

前端不需要数据顶多一个信息交互的提示

> 后端需要什么数据?

一个是表单一个用户，用户可以直接从ThreadLocal中get，所以其实只需要一个表单数据就好。

> 梳理

后端从前端获取表单数据然后从ThreadLocal中得到User然后调用插入数据库的方法即可

帖子详情

这个功能也很无脑

> 前端需要什么样的数据?

前端需要帖子的详细信息和用户的头像和名称

> 后端需要什么数据?

这个帖子的id

> 梳理

后端根据帖子的id查询数据库然后将数据返回给前端就好了

事务管理

这里介绍一下Spring的事务管理操作

SpringMVC的声明式事务还是很简单的

直接使用注解就好了@Transactional

在要使用事务的方法上加上这个注解@Transactional然后注解里写上事务的隔离级别和传播行为就好了

```
@Transactional(isolation = Isolation.READ_COMMITTED, propagation =
Propagation.REQUIRED)
```

显示评论

这块应该也属于帖子详情的一部分，因为评论是线上在帖子下面的，还是那样

> 前端需要什么数据

需要评论者头像姓名、评论的内容时间以及类型，因为要展示这是个评论还是回复，如果是回复还得要显示回复对象名称

> 后端需要什么

帖子id就够了！！！因为有帖子id我就能得到发帖人以及通过查询评论表得到评论的相关信息（评论人，评论类型，评论内容，如果类型是评论还会有回复对象）

> 业务梳理

后端从前端获取帖子id通过查询评论表然后首先判断评论类型如果是评论则直接显示如果回复则获取回复人的id和被回复的人的id然后查数据库返回给前端

添加评论

> 前端需要什么?

前端需要有一个来自后端的提示

> 后端需要什么

帖子id，时间，目标id，目标类型，回复内容，回复人的id

> 业务梳理

根据回复的地方不同，可以获取回复类型，用户id，帖子id时间回复内容很容易获取，回复目标人id如果是回复帖子则设置为0否则直接从前端页面获取

私信列表

> 前端需要啥

前端需要私信人信息，以及跟他的第一条聊天记录，以及聊天记录数，未读数据，首页还需要未读数据

> 后端需要啥

用户id就好了

> 业务梳理

通过用户id查询消息表找到所有有聊天记录的根据聊天人id进行分类然后将第一条数据返回，点击查看将数据全部标记已读

关键这些信息可以在用户登录时就进行获取因为首页可以显示未读数量

所以我们采用一个拦截器

```
@Component
public class MessageInterceptor implements HandlerInterceptor {

    @Autowired
    private HostHolder hostHolder;

    @Autowired
    private MessageService messageService;

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        User user = hostHolder.getUser();
        if (user != null && modelAndView != null) {
            int letterUnreadCount =
messageService.findLetterUnreadCount(user.getId(), null);
            int noticeUnreadCount =
messageService.findNoticeUnreadCount(user.getId(), null);
            modelAndView.addObject("allUnreadCount", letterUnreadCount +
noticeUnreadCount);
        }
    }
}
```

发送私信

这个so easy 啦

> 前端

需要交互信息

> 后端

需要数据库所需要的数据

发送者id，接收者id，标题，内容，时间

统一处理异常

Spring提供了提供了两个注解

@ControllerAdvice(annotations = Controller.class)写在类上告诉Spring这是个Advice

@ExceptionHandler({Exception.class})写在方法上告诉如果出现异常统统调用这个方法来处理异常

```
@ControllerAdvice(annotations = Controller.class)
public class ExceptionAdvice {

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

    @ExceptionHandler({Exception.class})
    public void handleException(Exception e, HttpServletRequest request,
HttpServletResponse response) throws IOException {
        logger.error("服务器发生异常: " + e.getMessage());
        for (StackTraceElement element : e.getStackTrace()) {
            logger.error(element.toString());
        }

        String xRequestedWith = request.getHeader("x-requested-with");
        if ("XMLHttpRequest".equals(xRequestedWith)) {
            response.setContentType("application/json;charset=utf-8");
            PrintWriter writer = response.getWriter();
            VO vo = new VO();
            vo.setCode(1);
            vo.setObj("服务器异常!");
            writer.write(JSON.toJSONString(vo));
        } else {
            response.sendRedirect(request.getContextPath() + "/error");
        }
    }
}
```

统一日志管理

统一日志管理采用AOP的思想在内一个方法内都织入记录日志的行为

很简单Spring提供了@Aspect注解只要在一个类上加上这个注解就可以实现该功能

```
@Component
@Aspect
public class ServiceLogAspect {

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

    @Pointcut("execution(* com.suo.service.*.*(..))")
    public void pointCut(){}

    @Before("pointCut()")
    public void before(JoinPoint joinPoint){
        // 用户[1.2.3.4],在[xxx],访问了[com.suo.service.xxx()].
        //利用request获取用户ip
        ServletRequestAttributes attributes = (ServletRequestAttributes)
RequestContextHolder.getRequestAttributes();
        HttpServletRequest request = attributes.getRequest();
        String ip = request.getRemoteHost();
        //获取用户访问时间
        String now = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new
Date());
```

```
        //获取用户访问方法
        String target = joinPoint.getSignature().getDeclaringTypeName() + "." +
joinPoint.getSignature().getName();
        logger.info(String.format("用户[%s],在[%s],访问了[%s].", ip, now, target));
    }

}
```

点赞

点赞功能如果单从业务逻辑的角度讲其实没啥难的，无非就是给每个用户设计一张表来记录该用户的被点赞数，该用户id，以及点赞者id

这个时候我们就要思考了点赞这个操作是不是一个数据量非常大操作特别频繁的呀，所以这个时候我们应该采用一个更加高效的数据库来存储这些数据，而这个数据库就是Redis数据库

Redis

Redis是一款基于键值对的NoSQL 数据库，它的值支持多种数据结构:字符串(strings)、哈希(hashes)、列表(lists)、 集合(sets)、 有序集合(sorted sets)等。

Redis将所有的数据都存放在内存中，所以它的读写性能十分惊人。同时，Redis还可以将内存中的数据以快照或日志的形式保存到硬盘上，以保证数据的安全性。

Redis典型的应用场景包括:缓存、排行榜、计数器、社交网络、消息队列等。

感觉redis有点像全局的Map以Key：value的形式存储数据，只不过他是一个全局的map并且效率更高，提供了更高效的CRUD，更多的类型

> Redis的事务机制

redis的事务是将操作放入一个队列中，所以不要在事务中进行查找操作

```
    //事务
    @Test
    public void tx(){
        Object obj = redisTemplate.execute(new SessionCallback() {
            @Override
            public Object execute(RedisOperations operations) throws
DataAccessException {
                String key = "test:tx";
                //开启事务
                operations.multi();
                ValueOperations valueOperations = operations.opsForValue();
                valueOperations.set(key,1);
                valueOperations.set(key,2);
                System.out.println(valueOperations.get(key));
                //提交事务
                return operations.exec();
            }
        });
        System.out.println(obj);
    }
```

Spring 整合 Redis

> 步骤

1. 引入依赖
2. 配置数据库参数
3. 编写配置类
4. 构造RedisTemplate

> 访问Redis

- redisTemplate. opsForValue()　　strings类型
- redisTemplate. opsForHash()　　hashset类型
- redisTemplate. opsForList ()　　lists类型
- redisTemplate. opsForSet ()　　sets类型
- redisTemplate. opsForzSet ()　　sorted set类型

> 配置类

```
@Configuration
public class RedisConfig {

    @Bean
    public RedisTemplate<String,Object> redisTemplate(RedisConnectionFactory
factory) {
        RedisTemplate<String,Object> template = new RedisTemplate<>();

        // 设置key的序列化方式
        template.setKeySerializer(RedisSerializer.string());

        // 设置value的序列化方式
        template.setValueSerializer(RedisSerializer.json());

        // 设置hash的key序列化方式
        template.setHashKeySerializer(RedisSerializer.string());

        // 设置hash的value序列化方式
        template.setHashValueSerializer(RedisSerializer.json());

        // 在设置完参数后生效
        template.afterPropertiesSet();

        return template;
    }

}
```

Redis的高级数据类型

1. HyperLogLog

- 采用一种基数算法，用于完成独立总数的统计。
- 占据空间小，无论统计多少个数据，只占12K的内存空间。
- 不精确的统计算法，标准误差为0.81%。

- 相当于set，用于去重，统计数据

```
    //统计20w个重复数据的独立总数
    @Test
    public void HyperLogLogTest() {
        String key = "test:hll:01";

        for (int i = 0; i < 100000; i++) {
            HyperLogLogOperations hyperLogLogOperations =
redisTemplate.opsForHyperLogLog();
            hyperLogLogOperations.add(key,i);
        }
        for (int i = 0; i < 100000; i++) {
            HyperLogLogOperations hyperLogLogOperations =
redisTemplate.opsForHyperLogLog();
            hyperLogLogOperations.add(key,(int)(Math.random() * 10000 + 1));
        }
        System.out.println(redisTemplate.opsForHyperLogLog().size(key));
    }

    //将三组数据合并，再统计合并后重复数据的独立总数
    @Test
    public void HyperLogLogUnionTest() {
        String key = "test:hll:02";
        for (int i = 0; i < 10000; i++) {
            HyperLogLogOperations hyperLogLogOperations =
redisTemplate.opsForHyperLogLog();
            hyperLogLogOperations.add(key,i);
        }

        String key2 = "test:hll:03";
        for (int i = 0; i < 10000; i++) {
            HyperLogLogOperations hyperLogLogOperations =
redisTemplate.opsForHyperLogLog();
            hyperLogLogOperations.add(key2,i + 5000);
        }

        String key3 = "test:hll:04";
        for (int i = 0; i < 10000; i++) {
            HyperLogLogOperations hyperLogLogOperations =
redisTemplate.opsForHyperLogLog();
            hyperLogLogOperations.add(key3,i + 10000);
        }

        String unionKey = "test:hll:union";


 System.out.println(redisTemplate.opsForHyperLogLog().union(unionKey,key,key2,ke
y3));
    }
```

2. Bitmap

- 不是一种独立的数据结构，实际上就是字符串。
- 支持按位存取数据，可以将其看成是byte数组。
- 适合存储索大量的连续的数据的布尔值。
- 相当于string byte[] 内容只能有0 1

```
    @Test
    public void BitmapTest() {
        String key = "test:bm:01";

        // 记录
        redisTemplate.opsForValue().setBit(key,1,true);
        redisTemplate.opsForValue().setBit(key,4,false);
        redisTemplate.opsForValue().setBit(key,7,true);

        // 查询
        System.out.println(redisTemplate.opsForValue().getBit(key,1));
        System.out.println(redisTemplate.opsForValue().getBit(key,2));
        System.out.println(redisTemplate.opsForValue().getBit(key,3));

        // 统计
        Object obj = redisTemplate.execute(new RedisCallback() {
            @Override
            public Object doInRedis(RedisConnection connection) throws
DataAccessException {
                return connection.bitCount(key.getBytes());
            }
        });

        System.out.println(obj);
    }

    // 或运算OR
    @Test
    public void BitmapOperationTest() {
        String key2 = "test:bm:02";
        // 记录
        redisTemplate.opsForValue().setBit(key2,0,true);
        redisTemplate.opsForValue().setBit(key2,1,true);
        redisTemplate.opsForValue().setBit(key2,2,true);

        String key3 = "test:bm:03";
        // 记录
        redisTemplate.opsForValue().setBit(key3,2,true);
        redisTemplate.opsForValue().setBit(key3,3,true);
        redisTemplate.opsForValue().setBit(key3,4,true);

        String key4 = "test:bm:04";
        // 记录
        redisTemplate.opsForValue().setBit(key4,4,true);
        redisTemplate.opsForValue().setBit(key4,5,true);
        redisTemplate.opsForValue().setBit(key4,6,true);

        String key = "test:bm:or";
        Object obj = redisTemplate.execute(new RedisCallback() {
            @Override
            public Object doInRedis(RedisConnection connection) throws
DataAccessException {

 connection.bitOp(RedisStringCommands.BitOperation.OR,key.getBytes(),key2.getByt
es(),key3.getBytes(),key4.getBytes());
                return connection.bitCount(key.getBytes());
            }
```

```
        });

        System.out.println(obj);

        for (int i = 0; i < 7; i++) {
            System.out.println(redisTemplate.opsForValue().getBit(key,i));
        }
    }
```

Redis的技术讲解完了接下来就是开始业务逻辑的处理了

使用Redis最难的并不是操作而是如何设计一个复用性高的表也就是key！！！

要思考怎么样设计一个key可以表示所有要表示的数据

以点赞为例：

点赞操作要分为两种一种是实体的点赞一种是用户的点赞

首先这是个实体点赞操作部署关注操作对吧所以key里面要有like字段，然后点赞操作我们需要知道这是个帖子啊还是评论啊，还是回复啊所以我们还需要实体的类型是吧，再者就是这个实体的id了。然后这个点赞是我点的吧需要我的id对吧。所以key为like:entity:entityType:entityId这样设计还能提供拓展性如果以后提供了更多的点赞种类都能使用比如我还想增加一个给用户直接点赞的需求那么只需要type+1就好了

然后巧妙的地方来了value我选用的是set（userid）因为我们需要考虑到这个用户有可能是点错了啦要取消点赞哒所以存储set可以判断这个用户有没有点过赞然后才能进行操作，而且set还能获得点赞的数量这样岂不美哉。

接着就是用户点赞key的设计了这个没啥就是一个用户id就好了存储的value也是数量就好了
like:user:userId -> int这个操作的设计是为了后面发送系统消息的需求提供的之后再讲

> 业务逻辑

后端需要从前端获取这个点赞的操作是给帖子还是回复还是评论也就是实体类型，然后是实体id然后是作为被点赞的人的id，至于点赞者我的id可以直接从ThreadLocal中获取，然后将我的id存到实体点赞的set中，然后用户点赞++反之移出再--。如果是点赞的操作那么还需要将这条消息添加到消息队列中

关注

关注和点赞操作一模一样所以我就只讲一下key的设计好了

key设计要分为两个一个是某个实体（用户）关注的实体（用户）列表，一个是一个实体（用户）被关注的实体（用户）列表，说白了就是我的关注和我的粉丝

```
// 某个用户关注的实体
// followee:userId:entityType -> zset(entityId,now)
// userId -> 某个关注别人的关注者id
// entityType -> 关注的类型，可以是帖子可以是人等待
// value是zset(entityId,now) 是一个有序集合以时间为score，以实体id为值
```

```
// 某个实体拥有的粉丝
// follower:entityType:entityId -> zset(userId,now)
// entityType -> 实体类型可以是用户可以是帖子等等
// entityId -> 这个实体的id
// 有了这两个参数可以唯一的表示某个实体，比如说如果这个实体是人那么entityType就是人entityId就是userId如果这个实体是帖子那么entityType就是帖子entityId就是帖子id
// value是zset(userId,now)是一个有序集合，集合中值是userId表示是哪个用户关注了这个实体
```

使用Redis缓存优化登录页面

这个思路很简单，就是通过用户id查询数据库这个操作是不是很频繁啊，比如进入首页后我们是不是要吧所以用户都查询一遍啊，所以我们采用Redis缓存来提高性能。

思路：

1. 先从缓存中取值
2. 如果取不到则查数据库然后将数据放入Redis缓存
3. 如果进行修改操作则直接删除缓存

```
public User findUserById(int id) {
    //return userMapper.selectById(id);
    User user = getCache(id);
    if(user == null) {
        user = initCache(id);
    }
    return user;
}

// 1.优先从缓存中取值
private User getCache(int userId) {
    String redisKey = RedisKeyUtil.getUserKey(userId);
    return (User) redisTemplate.opsForValue().get(redisKey);
}

// 2.取不到时初始化缓存数据
private User initCache(int userId) {
    User user = userMapper.selectById(userId);
    String redisKey = RedisKeyUtil.getUserKey(userId);
    redisTemplate.opsForValue().set(redisKey, user, 3600, TimeUnit.SECONDS);
    return user;
}

// 3.数据变更时清除缓存数据
private void clearCache(int userId) {
    String redisKey = RedisKeyUtil.getUserKey(userId);
    redisTemplate.delete(redisKey);
}
```

经jmeter压力测试后发现在2000线程并发的条件下使用Reids缓存性能提高了3倍

发送系统私信

发送系统私信有人会说这和发普通私信有什么区别啊，这不是一样的吗。

确实业务逻辑就是一样的但是这个可是系统通知啊设计到一个点就是并发量会很大，程序会吃不消所以我们采用Kafka消息队列的技术提高程序并发性

了解Kafka我们得先从阻塞队列说起

> 阻塞队列

阻塞队列本质上就是接口

- BlockingQueue
  - 解决线程通信的问题。
  - 阻塞方法: put、take。
- 生产者消费者模式
  - 生产者:产生数据的线程。
  - 消费者:使用数据的线程。
- 实现类
  - ArrayBlockingQueue
  - LinkedBlockingQueue
  - PriorityBlockingQueue、SynchronousQueue、 DelayQueue等。

```
package com.suo;

import org.springframework.boot.test.context.SpringBootTest;

import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

@SpringBootTest
public class BlockingQueueTest {

    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(10);
        new Thread(new Producer(queue)).start();
        new Thread(new Consumer(queue)).start();
        new Thread(new Consumer(queue)).start();
        new Thread(new Consumer(queue)).start();
```

```
    }

}

class Producer implements Runnable {

    private BlockingQueue<Integer> queue;

    public Producer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 100; i++) {
                Thread.sleep(20);
                queue.put(i);
                System.out.println(Thread.currentThread().getName() + "生产者:" +
queue.size());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}

class Consumer implements Runnable {

    private BlockingQueue<Integer> queue;

    public Consumer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        try {
            while (true) {
                Thread.sleep(new Random().nextInt(1000));
                queue.take();
                System.out.println(Thread.currentThread().getName() + "消费者:" +
queue.size());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

这个时候生产者速度快那么生产者会一直生产直到队列爆满了这个时候生产者线程被阻塞，其他线程就会更容易获取CPU资源所以此时消费者抢到CPU资源，进行消费，反之亦然。

阻塞队列提供了两个方法一个是put方法，一个是take方法，这两个方法都是具有阻塞性的，当队列满了之后这两个方法会处于一个等待状态这样可以空出资源来给其他进程工程，提供效率

> Kafka

- Kafka简介
  - Kafka是一个分布式的流媒体平台。
  - 应用:消息系统、日志收集、用户行为追踪、流式处理。
- Kafka特点
  - 高吞吐量、消息持久化、高可靠性、高扩展性。
- Kafka术语
  - Broker、Zookeeper
  - Topic、Partition、 Offset
  - Leader Replica、Follower Replica

> Spring整合Kafka

1. 引入依赖

spring-kafka

2. 配置Kafka

- 配置server、consumer

3. 访问Kafka

- 生产者
  kafkaTemplate. send(topic, data);
- 消费者
  @KafkaListener(topics = { "test"})
  public void handleMessage (ConsumerRecord record){}

生产者发消息是主动的，消费者接受消息是被动的是自动的

生产者要使用KafkaTemplate 提供的`send(topic,content)`方法

消费者要添加一个`@KafkaListener(topics = {"topic"})`的注解，那么这个方法就会接受content封装到ConsumerRecord record对象当中

使用前一定要开启zookeeper和kafka！！！

```
@SpringBootTest
public class KafkaTest {

    @Autowired
    private KafkaProducer kafkaProducer;

    @Test
    public void testKafka() {
        kafkaProducer.sendMessage("test", "你好");
        kafkaProducer.sendMessage("test", "在吗");
```

```
        try {
            Thread.sleep(3000 * 10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}

@Component
class KafkaProducer {

    @Autowired
    private KafkaTemplate<String,String> kafkaTemplate;

    public void sendMessage(String topic, String content) {
        kafkaTemplate.send(topic, content);
    }

}

@Component
class KafkaConsumer {

    @KafkaListener(topics = {"test"})
    public void handleMessage(ConsumerRecord record) {
        System.out.println(record.value());
    }

}
```

> 业务梳理

技术介绍完了现在正式开始业务梳理了

首先我们先来确定topic和content；

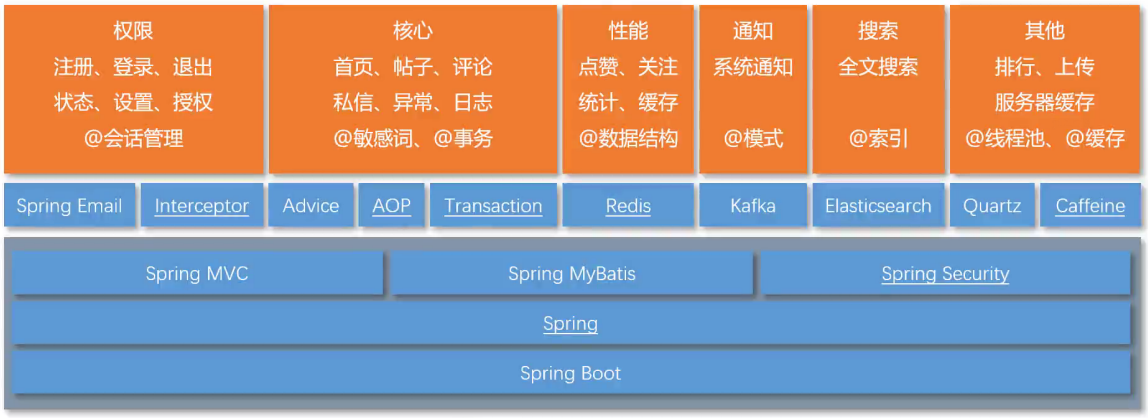
如果是点赞那么topic就是like，如果是关注那么topic就是follow

content都一样就是一个JSON格式的字符串，我们将其封装成Event对象这样便于JSON的转化

```
public class Event {
    private String topic;
    private int userId;
    private int entityType;
    private int entityId;
    private int entityUserId;
    private Map<String, Object> data = new HashMap<>();
}
```

收藏也是一样的业务逻辑

最后来个项目总结



本项目的基础是SpringBoot核心是**Spring**，然后由Spring整合SpringMVC、SpringMyBatis、**SpringSecunrity**、SpringEmail、**Redis**、Kafka、Elasticsearch 等技术其中权限部分主要交给SpringSecunrity和SpirngMVC的拦截器Interceptor；

核心部分由Spring负责包括其强大的事务机制，IoC和AOP思想

性能优化主要是靠Reids来实现，一方面Redis是存储在内存中而不是硬盘所以读写速度会比MySQL快，另一方面Reids可以作为缓存提高性能

高并发是通过Kafka来实现，Kafka重点关注其生产者和消费者模式