# 单点登录（SSO,Single Sign On）

## SSO定义

### SSO简介

SSO（ Single Sign-On ），中文意即单点登录，翻译得比较精简，个人觉得 Wiki 上的解释更细腻点—— SSO, is a property of access control of multiple related, but independent software systems. With this property a user logs with a single ID and password to gain access to connected system or systems without using different usernames or passwords, or in some configurations seamlessly sign on at each system. ( 单点登录是一种控制多个相关但彼此独立的系统的访问权限, 拥有这一权限的用户可以使用单一的ID和密码访问某个或多个系统从而避免使用不同的用户名或密码，或者通过某种配置无缝地登录每个系统 ). 注：系统，在本文特指WEB 应用或者WEB 服务；用户，下文也会称之为User；ID，用户标识；密码，本文也称其为口令，Password, Passcode 或者 Pin。

OK，从上面的定义中我们总结出 与 SSO 交互的2个元素：1. 用户，2. 系统，它的特点是：一次登录，全部访问。上面提到SSO是访问控制的一种，控制用户能否登录，即验证用户身份，而且是所有其它系统的身份验证都在它这里进行，那么我们是不是可以认为SSO还是一个验证中心。那么从整个系统层面来看SSO，它的核心就是这3个元素了：1. 用户，2. 系统，3. 验证中心。可能扯了那么多还是不足以形象地描述我们萌萌的SSO，呐，有图有真相：



好，经过我上面一大段废话，基本上对SSO要解决什么问题有一个清晰的认识。现在我们自行脑（yi）补（yin）下SSO 的原理是什么样的。

一个账户：嗯，规定所有系统统一使用相同账户，就能保证一个账户了；

一次登录全部访问：通过SSO登录后，让其告知其它各个系统保存该用户的信息，用户就不用重复多次的登录了；

嗯，问题解决了，没错，就这样。

### sso优点

1、方便用户

用户使用应用系统时，能够一次登录，多次使用。用户不再需要每次输入用户名称和用户密码，也不需要牢记多套用户名称和用户密码。单点登录平台能够改善用户使用应用系统的体验。

2、方便管理员

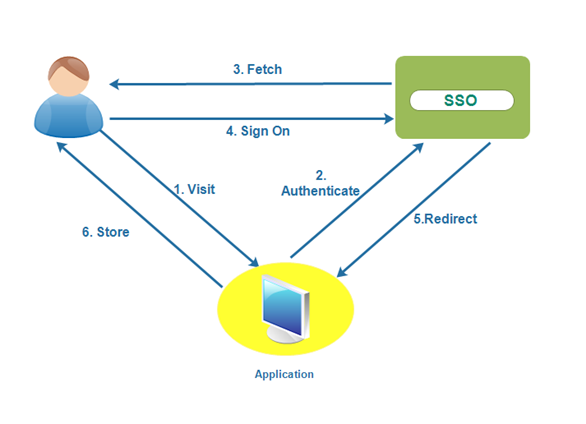
系统管理员只需要维护一套统一的用户账号，方便、简单。相比之下，系统管理员以前需要管理很多套的用户账号。每一个应用系统就有一套用户账号，不仅给管理上带来不方便，而且，也容易出现管理漏洞。

3、简化应用系统开发

开发新的应用系统时，可以直接使用单点登录平台的用户认证服务，简化开发流程。单点登录平台通过提供统一的认证平台，实现单点登录。因此，应用系统并不需要开发用户认证程序。

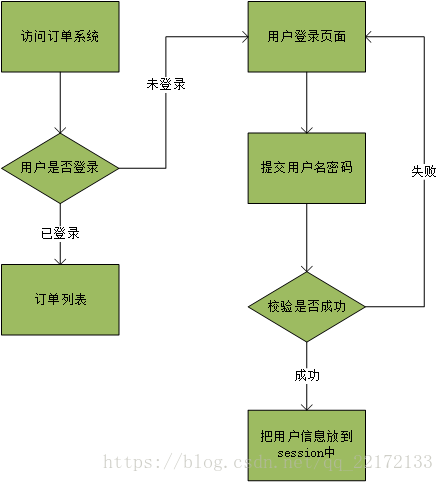
### 总结

SSO一句话总结：完成登录逻辑并使各系统共享验证信息和验证逻辑

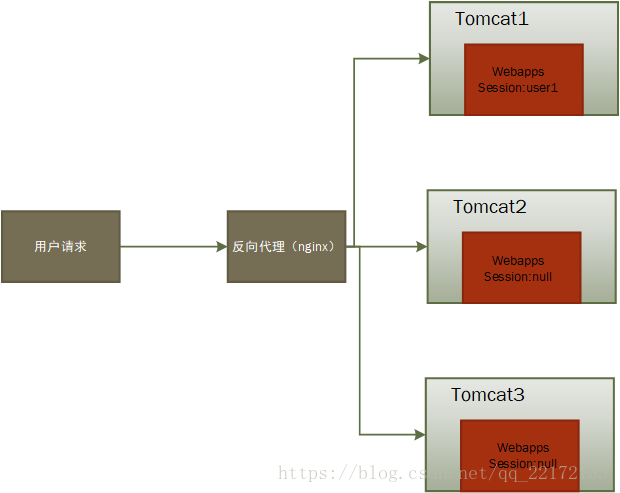


## 问题引入

我们在做SSO之前首先要明白为什么要有单点登录，即SSO在解决什么问题？那么我们先来看一下传统的登录实现方式：



以上就是传统的登录实现方式，但是在并发量高的情况下呢？比如现在有 2000~3000 的并发，这时一个tomcat 不能满足业务需求，需要做集群。如下：



那么现在就会出现Session共享的问题（tomcat做集群配置session复制。如果集群中节点很多，会形成网络风暴。推荐节点数量不要超过5个）。此外在分布式架构中，我们会把系统拆分成多个子系统，在这些子系统之间进行跳转时也会出现session不能共享的问题（和上述类似）。

上述这些情况正是SSO要解决的问题！

## 方案1

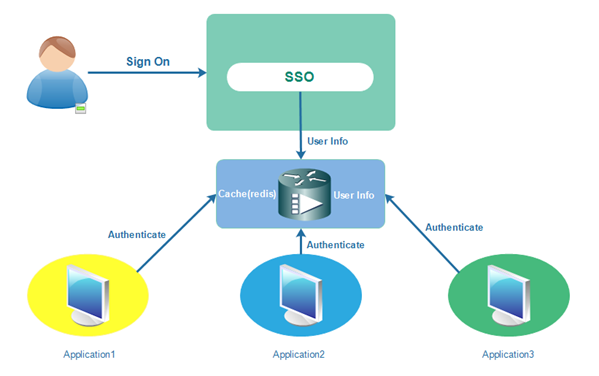
由上面的猜想可以得到第1个解决方案，记为方案1。这里对这个猜想做一点小小的优化，猜想中第2点 "各个系统保存" 好让人闹心，同一份数据保存多份，太浪费，这里我们把每个已登录的用户信息保存到公共缓存中。好，我们再来描述下这个方案：

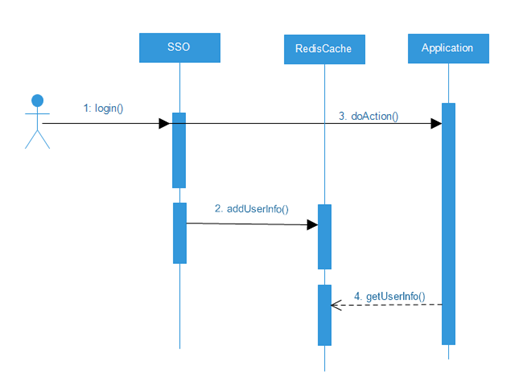
1、User 发送登录请求给SSO，附上自己的 ID 和 password；

2、SSO验证成功将用户信息保存在公共缓存 Cache 中；

3、User每次发送请求给系统 Ai 时，将 ID 作为请求参数；

3、系统 Aj 通过 请求中传过来的 User ID从公共缓存 Cache 中验证 User 是否登录，完成后续动作；





### 方案2

SSO\_V1 貌似解决了问题，但是深入思考，细思极恐！因为这个设计有Bug：每次传 ID 给服务Ai，但是这个ID 每次怎么获取来呢？登录SSO的时候，这倒没有问题，可以让用户填！但第2次请求是发给Ai中的某一个 Aj 时，ID 要怎么来（ 假设百度和新浪是相关但彼此独立的系统，登录百度后，再访问新浪时怎么让新浪取到与登录百度时一样的ID吧）？总不至于每次发请求时都要求用户填一遍ID 吧？

### 方案三

# 消息系统

## 消息系统

作用提醒用户

### 方式

短信

微信

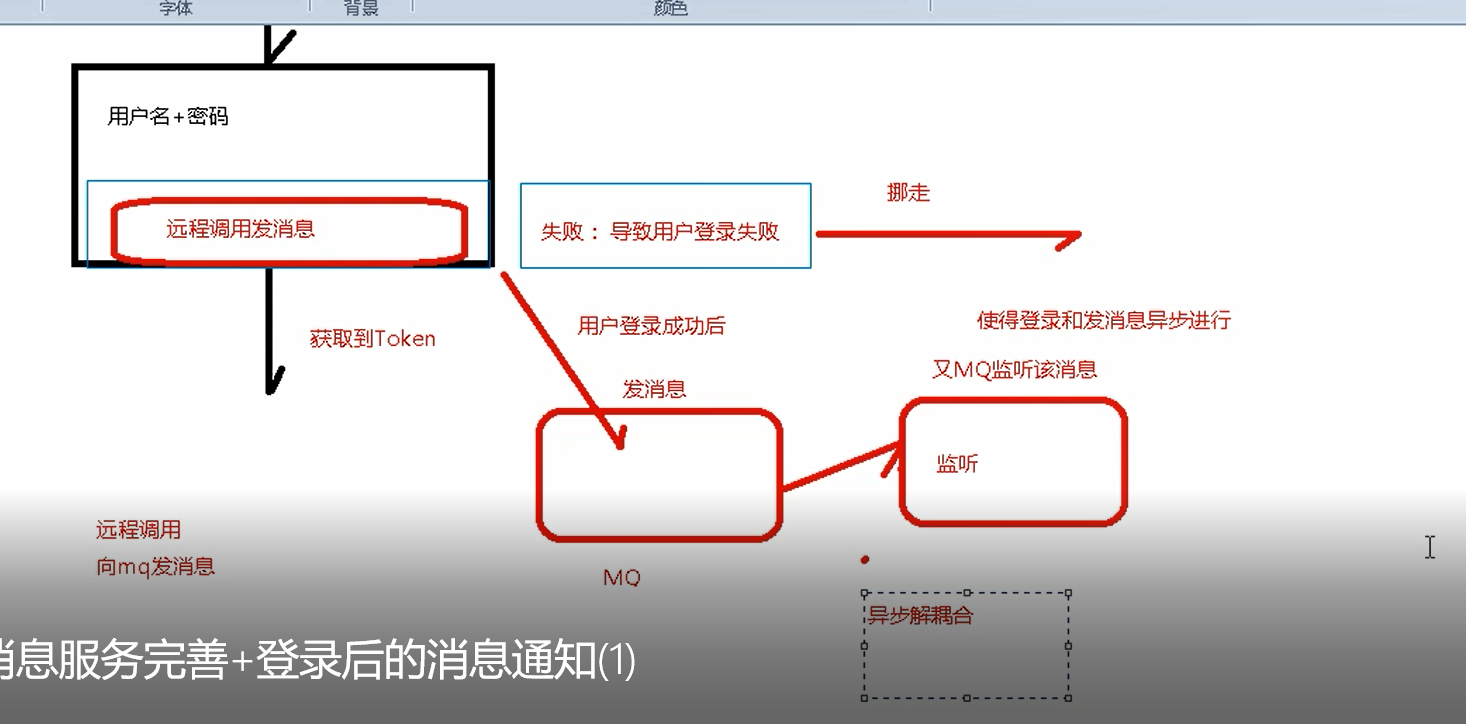
邮件

值用service即可，其他远程调用

## 登录发消息

### dubbo方案

### mq方案



# 集成微信

## 详情

### 访问网址

<http://mp.weixin.qq.com/debug/cgi-bin/sandbox?t=sandbox/login>

## 定义模板

### 模板格式

模板消息调用时主要需要模板ID和模板中各参数的赋值内容。请注意：=

1.模板中参数内容必须以".DATA"结尾，否则视为保留字;

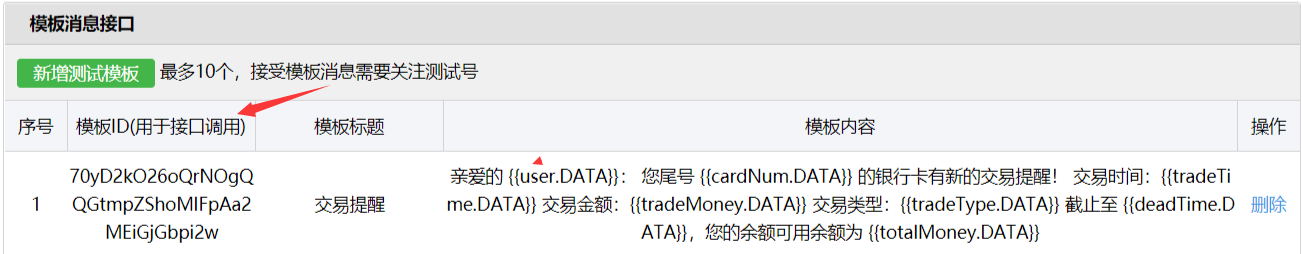
2.模板保留符号"{{ }}"

|  |
| --- |
| 亲爱的{{user.DATA}}：  您尾号{{cardNum.DATA}}的银行卡有新的交易提醒！  交易时间：{{tradeTime.DATA}}  交易金额：{{tradeMoney.DATA}}  交易类型：{{tradeType.DATA}}  截止至{{deadTime.DATA}}，您的余额可用余额为{{totalMoney.DATA}} |



生成的模板ID

为了保证用户不受到骚扰，在开发者出现需要主动提醒、通知用户时，才允许开发者在公众平台网站中模板消息库中选择模板，选择后获得模板ID，再根据模板ID向用户主动推送提醒、通知消息。



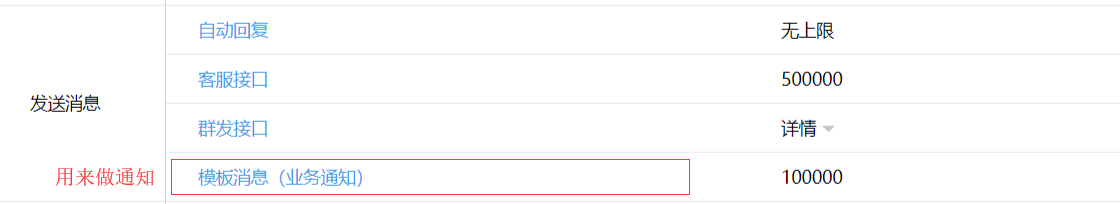
模板id：

70yD2kO26oQrNOgQQGtmpZShoMIFpAa2MEiGjGbpi2w

### 记录测试号

测试号的appID和appsecret是要他们获取访问微信接口的凭证。也就是只有通过验证，只有在该网站注册过的才能微信的接口。

|  |
| --- |
| appID wxe533af97c1113515  appsecret 2a4c03c4bd9aea718b77b84d542202ce |



## 模板消息

### 获取template\_id

### 请求接口

请求方法：POST

请求的URL地址：

<https://api.weixin.qq.com/cgi-bin/message/template/send?access_token=ACCESS_TOKEN>

ACCESS\_TOKE：access\_token是公众号的全局唯一接口调用凭据，公众号调用各接口时都需使用access\_token。开发者需要进行妥善保存。access\_token的存储至少要保留512个字符空间。access\_token的有效期目前为2个小时，需定时刷新，重复获取将导致上次获取的access\_token失效。

获取ACCESS\_TOKE的方法：通过访问接口来获取

https请求方式: GET

<https://api.weixin.qq.com/cgi-bin/token?grant_type=client_credential&appid=APPID&secret=APPSECRET>

参数：



appid：用户名

secret: 密码

例如获取token

|  |
| --- |
| **public** **static** **void** main(String[] args) {  //d  RestTemplate restTemplate = **new** RestTemplate();  String url = "https://api.weixin.qq.com/cgi-bin/token?grant\_type=client\_credential&appid=APPID&secret=APPSECRET";  String url1 = url.replaceAll("APPID", "wxe533af97c1113515");  String url2 = url1.replaceAll("APPSECRET", "2a4c03c4bd9aea718b77b84d542202ce");  String result = restTemplate.getForObject(url2, String.**class**);  System.***out***.println(result);  } |

返回的数据

{"access\_token":"18\_PceQaRwDCkoaRzNjvCJhWPq6xJ6pWJQ7hxHgNBD1gecPkssgVbdYkx\_aciOXFjF\_NB27MQiw9LuXioYWBBNVqGnf3dTgV2Yb7cy\_Hd6Rx5lpcT6dVbMoeOkILOgRTBgAEALNV","expires\_in":7200}

### 请求的数据格式：

|  |
| --- |
| {     **"data"**:{          **"User"**:{             **"color"**:**"#000000"**, //显示在模板上面的字的颜色             **"value"**:**"sx" //显示在模板上的字段名**         }     },     **"template\_id"**:**"1111"**, 微信接口定义的模板id     **"topcolor"**:**"topcolor"**,      **"touser"**:**"user"**, 向谁发送信息     **"url"**:**"url" 请求的微信接口。** } |

java对应的数据格式

|  |
| --- |
| **public** **class** MessageTemplate {  **private** String touser;  **private** String template\_id;  **private** String url;  **private** String topcolor;  //封装消息内容  **private** Map<String,Map<String,String>> data;  } |

### 请求实例

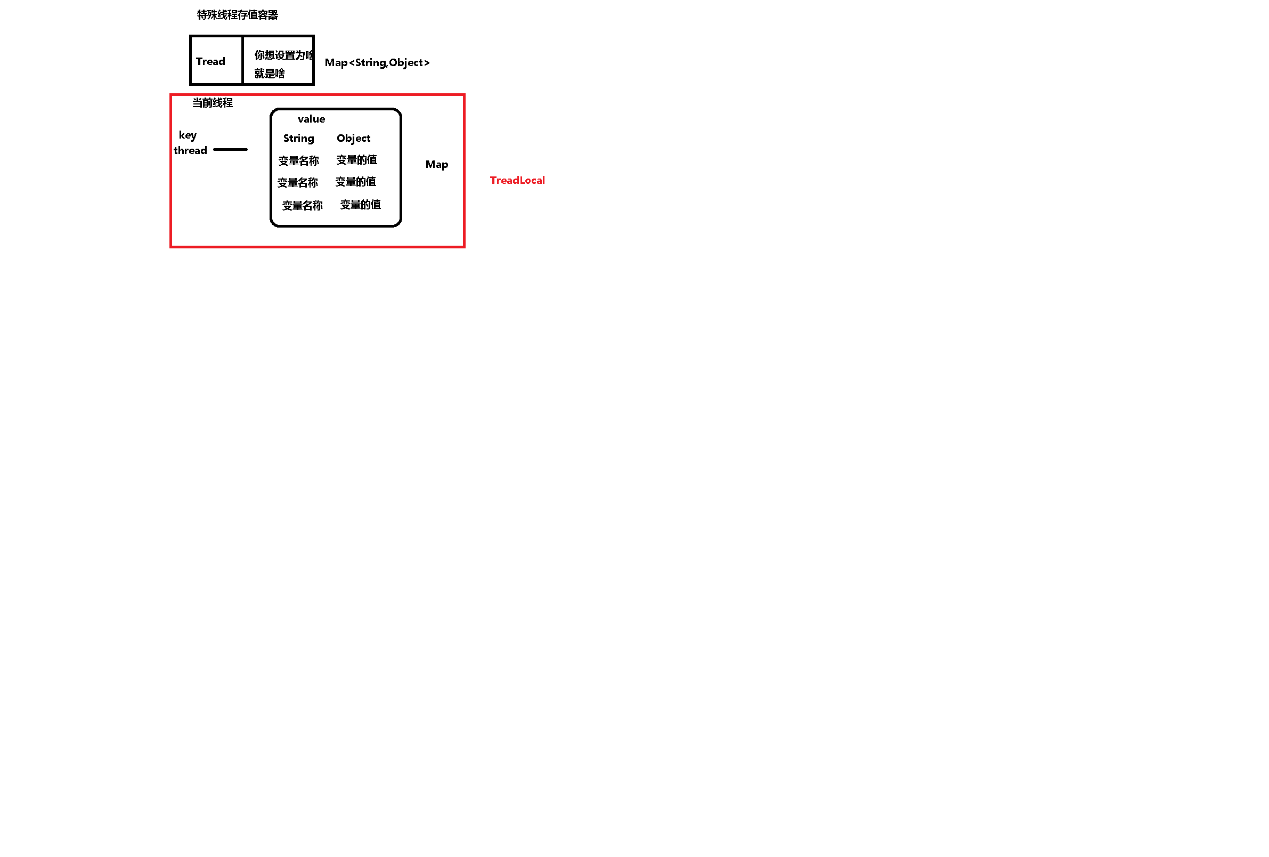
|  |
| --- |
| //给用户发送信息  **public** **static** **void** sendMessage() {  MessageTemplate messageTemplate = **new** MessageTemplate();  //设置模板id  messageTemplate.setTemplate\_id("70yD2kO26oQrNOgQQGtmpZShoMIFpAa2MEiGjGbpi2w");  //发送给那个用户,关注微信公众号自动生成一个id  messageTemplate.setTouser("oAeI56LW6Upx45KzxnB5BuyGkaX8");  //点击公众公众号会跳转的地址  messageTemplate.setUrl("https://www.baidu.com/");  HashMap<String,Map<String, String>> data = **new** HashMap<>();  data.put("user", MessageTemplate.*initChiMap*("吴伟凡", ""));  data.put("cardNum", MessageTemplate.*initChiMap*("9296", ""));  data.put("tradeTime", MessageTemplate.*initChiMap*("2019-2-18 17:00", ""));  data.put("tradeMoney", MessageTemplate.*initChiMap*("RMB：1000000元", ""));  data.put("tradeType", MessageTemplate.*initChiMap*("消费", ""));  data.put("deadTime", MessageTemplate.*initChiMap*("2019-2-19 17:01", ""));  data.put("totalMoney", MessageTemplate.*initChiMap*("RMB：10000000.000 元", ""));  messageTemplate.setData(data);  String url = "https://api.weixin.qq.com/cgi-bin/message/template/send?access\_token=ACCESS\_TOKEN";  //获取的token  url = url.replaceAll("ACCESS\_TOKEN","18\_h08ziaO1YegKrxs2h4n\_iLcPo1\_emkG6NqkELpVYbQ0zY80rpU1kiEOTEBxJUkAtRS2JqY\_SImakbL6sV38T1\_REFNm\_XumPv910fuoYugW3nPoUbjp7kUm-JGR\_5A3ZBKs\_sdPES\_vDYqxoBSXdAIAULQ");  RestTemplate restTemplate = **new** RestTemplate();  //发送post请求  String result = restTemplate.postForObject(url, messageTemplate, String.**class**);  System.***out***.println(result);  } |

返回的结果：{"errcode":0,"errmsg":"ok","msgid":687737191010861056}

# 订单系统

## json跨域请求

## ThreadLocal



## 分页

## 订单编号的生成

# 分页

## 原始分页

### controller

|  |
| --- |
| @GetMapping(**"/page"**) **public** Result page(@RequestParam(**"page"**) Integer page,  @RequestParam(**"pageSize"**) Integer pageSize){  PageExtend pageExtend = **new** PageExtend();  pageExtend.setPageNo(page);  pageExtend.setPageSize(pageSize);  **return** Result.*create*(**memberService**.page(pageExtend)); } |

### Service

|  |
| --- |
| @Override **public** PageInfo<PageExtend> page(PageExtend pageExtend) {  *//查询总的条数* Integer count = **classMemberDao**.count(pageExtend);  PagingQuery<PageExtend> pagingQuery = **new** PagingQuery<>(pageExtend, count);  *//计算首页参数* PageExtend condition = pagingQuery.condition();  List<PageExtend> pageExtends = **classMemberDao**.page(condition);  **return** pagingQuery.pageInfo(pageExtends); } |

### 附件类

PageExtend

|  |
| --- |
| **public class** PageExtend **extends** PageLimit {   Integer **id**;   Integer **name**;   Integer **desc**; } |

PageLimit

|  |
| --- |
| **public class** PageLimit **extends** BaseDomain {  */\*\*  \* 当前页的编号  \*/* @JSONField(serialize = **false**)  **private** Integer **page**;  */\*\*  \* 每页的数量  \*/* @JSONField(serialize = **false**)  **private** Integer **pageSize**;  */\*\*  \* 当前页起始的条目  \*/* @JSONField(serialize = **false**)  **private** Integer **pageFirst**;   **public** Integer getPageNo() {  **return page**;  }   **public void** setPageNo(Integer pageNo) {  **this**.**page** = pageNo;  }   **public** Integer getPageSize() {  **return pageSize**;  }   **public void** setPageSize(Integer pageSize) {  **this**.**pageSize** = pageSize;  }   **public** Integer getPageFirst() {  **return pageFirst**;  }   **public void** setPageFirst(Integer pageFirst) {  **this**.**pageFirst** = pageFirst;  } } |

PagingQuery

|  |
| --- |
| **public class** PagingQuery<T **extends** PageLimit> {  **private** T **t**;  **private** Integer **count**;  **private static final int *PAGE\_SIZE*** = 20;  **private static final int *PAGE\_NO*** = 1;    **public** PagingQuery(T t, Integer count) {  **this**.**t** = t;  **this**.**count** = count;  }   **public** T condition() {  **if** (**t**.getPageNo() == **null** || **t**.getPageSize() == **null**) {  **t**.setPageNo(***PAGE\_NO***);  **t**.setPageSize(***PAGE\_SIZE***);  }   *// 计算实际页数* Integer actualPageCount = **count** % **t**.getPageSize() > 0 ? **count** / **t**.getPageSize() + 1 : **count** / **t**.getPageSize();   *// 计算实际查询页* Integer actualPage = **t**.getPageNo() > actualPageCount ? actualPageCount : **t**.getPageNo();   *// 计算最后一页的内容总数* Integer lastPageCount = **count** - (actualPage - 1) \* **t**.getPageSize();   *// 计算PageFirst* **t**.setPageFirst(PageInfo.*getPageFirst*(actualPage, **t**.getPageSize()));   **return t**;  }   **public** PageInfo pageInfo(List list) {  PageInfo pageInfo;  **if** (**t**.getPageNo() != **null** && **t**.getPageSize() != **null**) {  Integer actualMaxPage = **count** / **t**.getPageSize() + 1;  pageInfo = **new** PageInfo<>(**t**.getPageNo() > actualMaxPage ? actualMaxPage.intValue() : **t**.getPageNo(), **t**.getPageSize(), **count**);  } **else** {  pageInfo = **new** PageInfo<>(0, 0, **count**);  }  pageInfo.setList(list);  **return** pageInfo;  } } |

pageInfo

|  |
| --- |
| **public class** PageInfo<T> {   */\*\*  \* 每页显示集合  \*/* **private** List<T> **list** = **new** ArrayList<>();   */\*\*  \* 总条数  \*/* **private int total**;   */\*\*  \* 总页数  \*/* **private int totalPage**;   */\*\*  \* 当前页  \*/* **private int page**;   */\*\*  \* 显示条数  \*/* **private int pageSize**;   */\*\*  \* 起始数目  \*/* **private int pageFirst** = 0;   **public** PageInfo(**int** page, **int** pageSize, **int** total) {  **this**.**page** = page;  **this**.**pageSize** = pageSize;   **this**.**total** = total;  }   **public** PageInfo() {  **this**.setList(Collections.*emptyList*());  **this**.**total** = 0;  }   **public int** getTotalPage() {  **if** (**total** <= 0) {  **return** 0;  }  **totalPage** = **total** / **pageSize**;  **if** (**total** % **pageSize** > 0) {  **totalPage**++;  }  **return totalPage**;  }   **public static int** getPageFirst(**int** pageNo, **int** pageSize) {  *//如果第一页，则从0开始* **if** (1 >= pageNo) {  **return** 0;  } **else** {  **return** (pageNo - 1) \* pageSize;  }  }  **public int** getPagefirst() {  **return pageFirst**;  }  **public void** setPagefirst(**int** pageFirst) {  **this**.**pageFirst** = pageFirst;  }  **public void** setTotalPage(**int** totalpage) {  **this**.**totalPage** = totalpage;  }  **public** List<T> getList() {  **return list**;  }  **public void** setList(List<T> list) {  **this**.**list** = list;  }  **public int** getTotal() {  **return total**;  }  **public void** setTotal(**int** total) {  **this**.**total** = total;  }  **public int** getPage() {  **return page**;  }  **public void** setPage(**int** page) {  **this**.**page** = page;  }  **public int** getPageSize() {  **return pageSize**;  }  **public void** setPageSize(**int** pageSize) {  **this**.**pageSize** = pageSize;  }  **public int** getFirst() {  **return** (**page** - 1) \* **pageSize**;  } } |

# 项目优化

## 静态map的本地缓存

### 描述

数据量少，并且数据不变的表可以在项目启动时,读取到内存的一个map中,使用不需要读取读取数据库。

### AttributeCache缓存map

使用一个Map<Integer, Object>存储，key为对象的id(或者是能够标志这个对象的字段1)，value为对象。

|  |
| --- |
| **public class** AttributeCache {  **public static** Map<Integer, Object> *attributeMap* = Maps.*newHashMap*();  **public static** Map<Integer, Object> *attributeMap1* = Maps.*newHashMap*();  **public static** Map<Integer, Object> *attributeMap2* = Maps.*newHashMap*(); } |

### AttributeListener(初始化缓存map)

项目启动时，讲数据库中的记录放到内存中map中。

|  |
| --- |
| *//初始化bean时,调用afterPropertiesSet()方法* @Component **public class** AttributeListener **implements** InitializingBean {  @Override  **public void** afterPropertiesSet() {  *//加载到内存中的方法* loadAttribute();  loadAttribute1();  loadAttribute2();  }  **private void** loadAttribute() {  *//从数据库中加载* List<Object> list = attributeService.listObject();  *//放入对应的map中* list.forEach(c -> AttributeCache.*classTypeMap*.put(c.getId(), c));  }  **private void** loadAttribute1() {  List<Object> list = attributeService.listObject();  list.forEach(c -> AttributeCache.*lessonGroupMap*.put(c.getId(), c));  }  **private void** loadAttribute2() {  List<Object> list = attributeService.listObject();  list.forEach(c -> AttributeCache.*lessonGroupMap*.put(c.getId(), c));  } } |