社交网络系统

软件构架文档

版本 <4.0>

修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **说明** | **作者** |
| 2015.3.26 | 1.0 | 迭代一 | 张子辰 |
| 2015.4.22 | 2.0 | 迭代二 | 张子辰 |
| 2015.5.22 | 3.0 | 迭代三 | 张子辰 |
| 2015.6.20 | 4.0 | 迭代四 | 张子辰 |

目录

1. 简介 5

1.1 目的 5

1.2 功能 5

1.3 开发平台和环境 5

2. 架构风格 5

3. 逻辑视图 6

3.1 类图 6

3.2 包图 7

4. 进程视图 8

5. 部署视图 9

6. 开发视图 9

7. 监控与恢复 9

8. 可靠性 10

9. 安全 10

10. 可修改性 10

11. 易用性 10

12. 可测试性 10

13. 设计模式 11

13.1 工厂模式 11

13.2 装饰器模式 11

13.3 构造器模式 11

13.4 桥接模式 11

13.5 迭代器模式 11

13.6 适配器模式 11

13.7 抽象工厂模式、单例模式 11

13.8 模板方法模式 12

13.9 解释器模式 12

13.10 中介者模式 12

13.11 备忘录模式 12

14. 架构模式 12

14.1 数据库设计模式 12

14.2 数据库访问模式 12

14.3 会话管理模式 12

14.4 插件机制 13

14.5 分离接口 13

14.6 服务设计模式 13

15. SaaS 13

15.1 工作流程 13

15.2 数据库 13

15.3 个性化 14

16. 大数据分析 14

16.1 策略 14

16.2 数据模型 14

16.3 局限性 14

软件构架文档

# 简介

## 目的

本文档将从构架方面对社交网络系统（即SNS）进行综合概述，其中会使用4+1视图来描述系统的各个方面。接着，它会从安全、故障监控与备份、易用性、可靠性来详细描述该系统的设计。

## 功能

该社交网络系统具有如下功能：发微博、写博客和即时聊天。

## 1.3 开发平台和环境

开发语言：Java。

组件或框架：servlet+jsp。

建模工具：PowerDesign。

数据库：mysql。

# 架构风格

该网站共使用了以下架构：

（1）浏览器和服务端间用了B/S架构。

（2）服务端的程序整体上用了MVC三层架构。

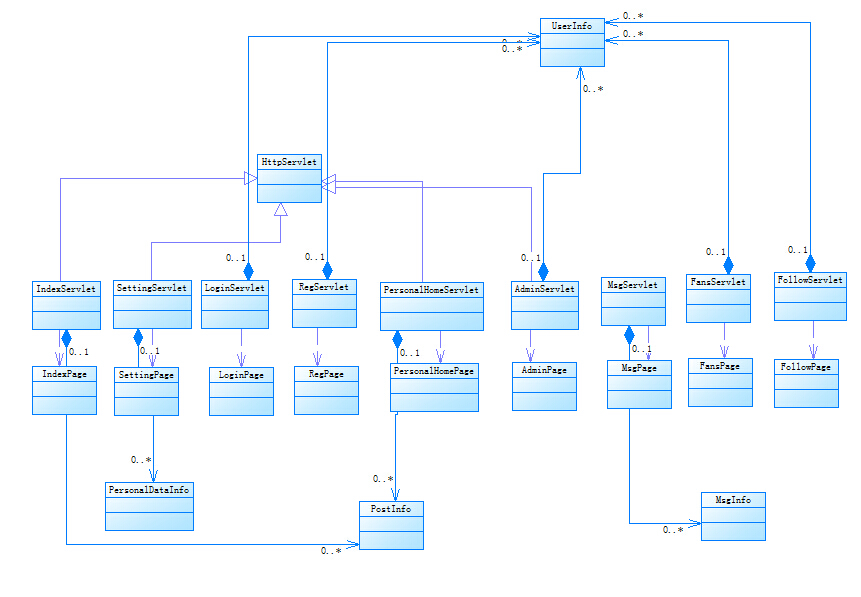
（3）控制层和数据层间用了黑板报模型。

（4）控制层的输入校检处用了管道过滤器模型。

在web开发方面，基本都是套现有的框架或者中间件，框架是哪种模型就用哪种模型，没有太多选择的余地。

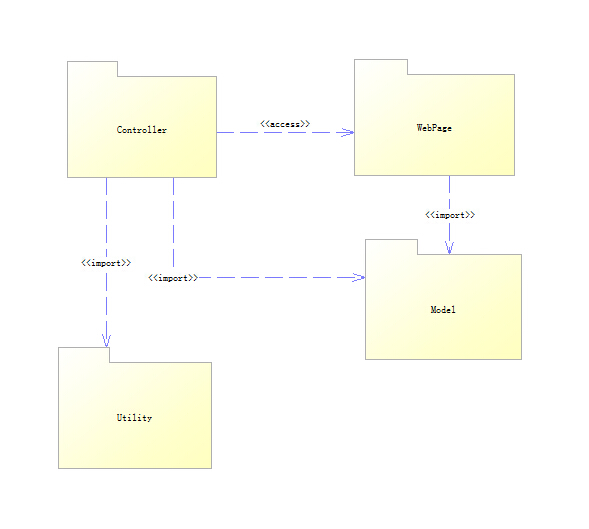
# 逻辑视图

## 类图



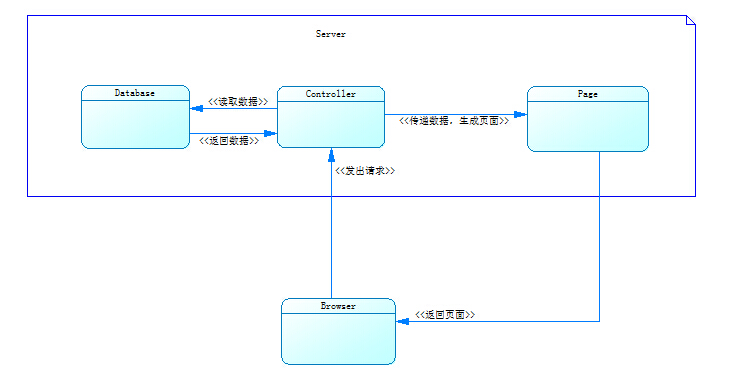
首先，按照功能划分为如下几个页面：主页、登录、注册、个人页、设置、粉丝、关注、消息、后台管理。每个页面配有一个Servelet类和一个Page。Info后缀的是存放数据实体的类。

## 包图

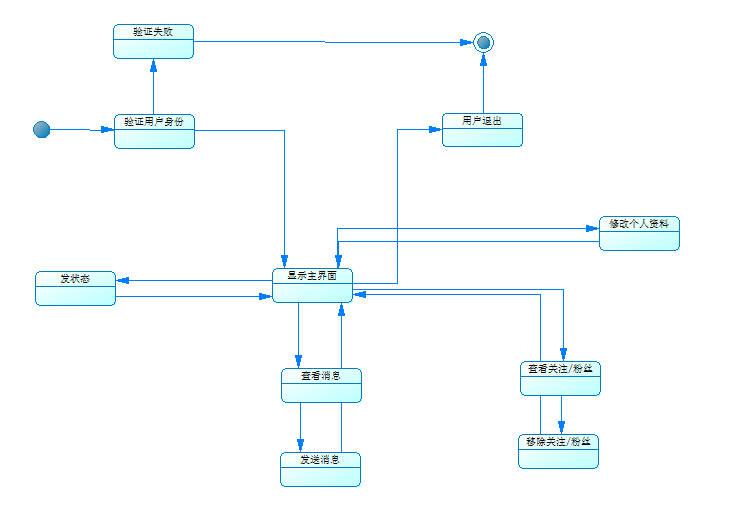


其中，Sevlet存放于Controller中，Page存放于WebPage中，Info存放于Model中。Utility中是一些功能类，比如编码解码。

# 进程视图

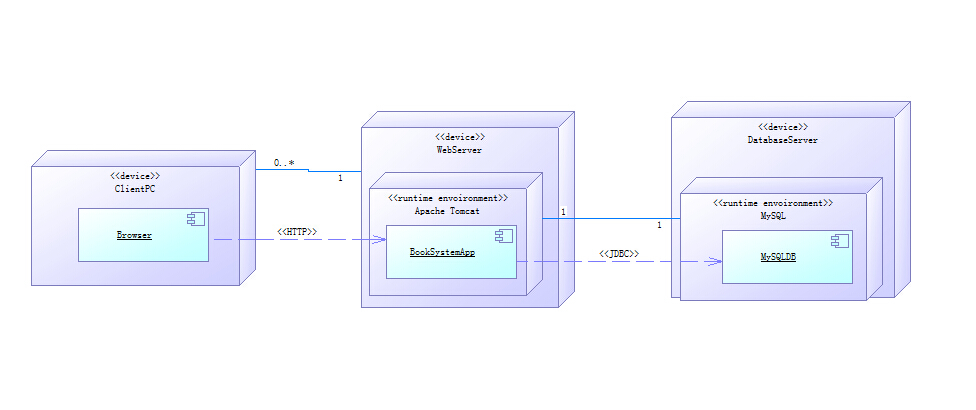


这是在一次请求之内的视图。当用户访问一个页面时，服务器会创建相应Controller类的对象，这个对象从数据库读取数据后，封装为一些数据结构，通过request传递给相应的Page。Page生成页面后返回给浏览器。



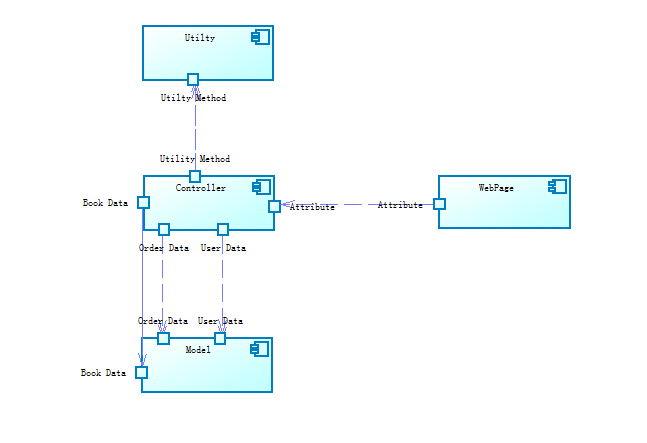
这是用户在线上的操作轨迹。由于http请求不是长连接，其中每一个操作包含一次完整的请求。

# 部署视图



其中，WebServer和DatabaseServe可以是一台服务器，也可以是一个集群，视用户数量而定。

# 开发视图



各个名字的含义和包图中相同。

# 监控与恢复

对于集群中的每台主机，采用心跳的方式进行监控。即被监测的服务器每隔一段时间，向监控服务器发心跳包，内容带上时间即可。如果没有收到某个服务器的消息，可以推算出大致是什么时候挂的。

监控一共有三种方式，除了心跳之外，有一种方案是被监控的主机在发生故障时向监控服务器发消息，但是故障可能很严重而导致消息并不能够发出。还有一种方案叫做ping或者echo，即监控用的主机定时向被监控者发请求，若收到请求则说明正常。但这种方案需要正在工作的服务器监听额外的请求，无疑增加了它的负载。综合各种因素，最后选择了心跳的方式。

该系统的数据采用被动冗余的方式进行备份。被动冗余指的是，服务器各节点之间不主动进行主备切换，而是由客户端来试，直到找到一个发送成功的节点，这个节点就自动成为主要节点。由于SNS从本质上来说可靠性要求不是很强，故采用这种策略。

# 可靠性

我们采用如下方式来提升它的可靠性：

（1）集群之间共享资源。

（2）限制单位时间请求次数，限制单位时间网络流量，限制单位时间内数据库连接次数，限制单位时间并发连接数。

（3）设置优先级。优先级相同的任务先创建先处理。

# 安全

经过我们在web开发方面的经验，采用如下方式来提升安全性：

（1）对所有具有格式的输入参数进行检验。

（2）对于所有可能带有html实体的输出参数，在page层进行转义。

（3）使用orm框架来防止sql注入。

（4）关键操作，比如添加删除等，设置并检验token来避免csrf。

（5）在所有页面的controller层，首先检验用户的权限。

（6）用户的密码用sha1或sha256加密后在数据库中存储。

（7）限制单ip24小时内的最大注册用户数。

# 可修改性

一般情况下，我们只是改动接口里面的逻辑，如果一定要改动接口，会采用增加适配器的方式来实现可修改性。适配器方式，实际上是在一个类上面实现两个接口，一新一旧，新的接口函数的实现去调用旧的。我们在开放新的接口同时，并不关闭旧的接口，以保证使用旧接口的客户端正常运行，当然旧接口只是没新接口强大罢了。

# 易用性

在易用性方面，我们会从一下几点来入手：

（1）页面使用合理的布局和颜色搭配。

（2）常使用的链接放置在突出位置，其他链接也不会放置过深。设置一个区域，用户可以自定义他们常使用的链接。

（3）顺应用户的使用习惯，通过参考其他网站，使用页面交互上的一些惯例。

此外，网站的易用性不仅仅是架构方面的事情。用户界面，包括布局和配色和交互方式，这种设计上的东西也影响者易用性。

# 可测试性

我们会考虑为测试人员开放特殊访问接口来增加可测试性。比如，重要操作前后都要记录日志，然后开放接口让测试人员可以按需访问日志的某个部分。也可以开放数据库的读权限给测试人员，便于检测数据的增删改查情况。

# 设计模式

## 工厂模式

工厂模式提供了一个接口，可以根据传进去的参数返回相应的对象。

假设我的社交系统打算面向世界，就要有一个国际化的策略。我的策略是这样，所有要进行语言转换的词，都有一个id编号，读取用户的语言设置后，加载相应的字典，用id在字典中取到相应的词后，写到页面上。获取字典的接口可以设计成工厂模式，传进去表示语言的字符串，返回相应的字典。

## 装饰器模式

装饰器模式相当于给被装饰的对象做外包装，同一个对象可以表现出不同的效果。

我的社交系统可能会根据访问者权限的不同，展示用户资料的完整度不同，比如，管理员可以看到全部资料，包括邮箱和手机等敏感信息，好友可以看到大部分资料，陌生人只能看到必要的信息。再比如，博客的文章，需要根据情景的不同（手机或者PC端，首页或者内页）加载不同详细程度的正文内容。这种情况会用到装饰器模式。

## 构造器模式

构造器模式用于简化对象的构造操作，当一个对象的构造器中，参数特别多的时候，就应该改为构造器模式进行构造。

我的社交系统带有按条件搜索指定用户的功能。搜索条件包括昵称、年龄、城市、工作教育经历，等等一系列用户设置的信息。搜索的时候需要构造一个SearchCondition对象，如果直接在构造器中传入所有参数，会显得很繁琐，所以这里采用构造器模式。

## 桥接模式

桥接模式将类的继承关系简化为组合关系。这样的好处是，当类有多个属性时，所需要创建类的数目由相乘的数量，减少为相加的数量。比如，手机有品牌和功能两个属性，假设有n个品牌，m中功能，若是按照继承关系来构建类，需要有mn个类；但采用桥接模式后，将品牌和功能变为手机的属性，只需要有（m + n）个类，编码量减少，代码复用程度提高，而且每个属性的更换和组合会更加灵活。

在社交系统中，用户的属性之一是权限，当然它还有很多属性。权限这个属性将封装成一个类，和用户类组合，而不是由用户类派生出管理员和版主类。这个地方应用了桥接模式。

## 迭代器模式

迭代器模式将各种奇奇怪怪的数据结构（比如说树和图）统一为线性操作。这样调用者可以在不知道具体数据结构及其相关知识的情况下，遍历整个结构。

为了便于操作，数据库要做or映射，返回值是一个装有查询结果的容器，这个容器将采用迭代器模式，可以用迭代器进行访问。

## 适配器模式

适配器模式可以将一组接口转化为另一组接口，常常用于不同版本的接口间的适配。

我的社交系统为了方便各种客户端（PC、移动端）的访问，设置了不同协议的HTTP接口。而在控制层内部，为了代码复用会把业务逻辑与控制逻辑分离，单独写在一些类里面。这些业务逻辑类，会提供一些接口，比如传入用户信息、标题和内容调用AddArticle来发布博文。假设我的系统开始没有移动端，后来新加上去，然而移动端需要上传一个值来表示它的种类（ios、安卓、wp），我就需要开一个新的接口，但是我应用了适配器模式之后，就不需要完全重写业务逻辑。我只需要把新的接口适配到旧接口上面，之后额外处理用户端种类就好。

## 抽象工厂模式、单例模式

在一个web系统里面，经常要连接数据库。连接数据库的语句不只一步，而且连接的时候要传很多参数，比如用户名、密码、主机。这些参数是从配置文件里面获得的，每次都要把get方法写一遍很麻烦。这些东西可以包装在一个工厂类里面，工厂类提供一个getConn的静态方法用来获取数据库连接。还可以应用单例模式，在每次调用的时候，检查数据库连接的状态，如果当前进程已有一个连接，就直接把内置的连接返回，不额外新建连接。

## 模板方法模式

在我的社交系统中，专门有一个类，根据加载进来的参数，渲染html页面。但是用户常常有更换不同皮肤的需求，不同皮肤实际上只是css和一些装饰用的部件的不同，大体上还是一致的。于是我把整个页面切成若干个部分，分别由若干个函数进行渲染。如果要更换皮肤，一个新的皮肤对应一个派生类，这个类可以覆盖掉原始皮肤类的方法，实现自己的逻辑；也可以新增加一些方法，渲染出新的部件，最后按自己的方式组合。

## 解释器模式

解释器模式用于解释某一种脚本语言。解释器模式的难点不在于创建类本身，而在于词法解析的算法部分。由于技术难度太大，而且我觉得没有必要为这个社交网站单独设计一门语言，所以自己编写的类里面没有用到解释器模式。不过，社交网站还是大量调用了带有解释器模式的库，比如正则表达式。在控制层的最开始，我们读进参数后要验证参数的合法性，这个位置大量用到了正则表达式。

## 中介者模式

中介者模式常常用于GUI程序的绑定回调中。比如，一个按钮的回调函数是设置一个文本框的内容，我们通常的做法是在窗体类里面写回调，然后在这个按钮的点击事件上面绑定个带有窗体对象的闭包，闭包里面调用回调函数。而不是直接把文本框这个对象传给按钮，然后按钮继承出一个类来改写回调内容。这样减少了控件间的耦合程度。

由于web应用并不处理控件，中介者模式在这个web应用本身不适用。但是如果我为我的社交系统编写一个移动端的app，里面绑定回调的操作会大量应用中介者模式。

## 备忘录模式

备忘录模式是指创建一个类来保存历史操作。这个模式在游戏的存档读档撤销功能里非常常见。

现在很多网站，比如知乎、wordpress都拥有查看历史记录的功能，我的社交网站也不例外。在用户请求查看历史记录的时候，从数据库加载历史记录，并存到备忘录的类里面，刷到页面上给用户展示。之后，用户可以根据历史记录的id请求恢复到某个版本。

# 架构模式

## 数据库设计模式

在主键的选取方面，我的社交系统的模式是，实体表选择一个整数的id作为主键，关系表选择多个相关id作为共同主键。

在主键的获取方面，我的社交系统选择让数据库计数器来管理，返回自增主键。

这种方法简单直接，web应用上面没有复杂的逻辑。

## 数据库访问模式

我的社交系统选择orm作为访问模式。Orm解决了数据库中的表和应用中的对象不匹配的问题，而且能够以编程语言而不是sql的方式获取数据，更加灵活，便于操作。

另外，整个应用会做到尽量避免长连接。在设计上，每个事务都很简短，没有什么需要用到长连接的地方，再有一点就是长连接非常耗性能，不利于大量用户的并发访问。

## 会话管理模式

会话管理模式共有三种，一种是放在数据库中，一种是放在server端，最后一种是放在浏览器端。

通过对各大电商网站的观察，本人发现，他们的策略是，未登录用户的购物车信息放在浏览器端，已登录用户的信息放在数据库中。

我的社交系统也会采用这种模式。首先服务器端是不能放的，原因很简单，大多数人还是习惯将购物车当做收藏夹来用，谁也不希望自己保存的购物车信息几分钟就没了。其次，服务端的会话存储是序列化之后存到文件中，也需要做本地IO，性能上和数据库访问应该差别不大。对于未登录的用户，他们的访问一般是短时期，临时性的，但是这种临时性的访问也有一定的数量，放在浏览器端有助于缓解服务端的压力。如果这些临时性的用户想要在不同终端操作，他们一定会去注册用户的。

## 插件机制

插件其实是一个应用，提供了一组确定的接口，以便于主程序能够正确加载。这就好比javase的程序要声明main函数，以便jvm能够正确加载。

插件的优点无需多说。我的社交系统在设计上有微博（状态）、博客、im三个功能。但是互联网是飞速发展的，我们并不知道未来会发明，或者需要一些什么新的功能。比如我的网站上面要加个电商平台买东西，或者要加个论坛，用户要与主程序保持一致，诸如此类。但是我的主程序已经打好jar包了，改动起来要重新编译测试部署，十分费劲，这个时候以插件的形式增加功能是最合适的。像一些成熟的网站平台，比如discuz、wordpress都有非常完善的插件机制和生态。我想，我的社交网站也应该这么做。

## 分离接口

前面我也提到过，我的社交网站可访问的终端，起码有web端和移动端。为了区分他们做流量统计，这两个终端访问的HTTP接口是不一样的。然而，这些不同的接口，所做的事情有可能一样，比如都是发状态或者写博客。这就需要把业务逻辑，即具体的功能实现部分单独放到一个类中，提供编程语言上的接口。然后，控制逻辑部分，即Servlet或者WebService来调用它们。

在各种框架或者组件中，都能找到这种模式。在ssh中，存放业务逻辑的类叫做DAO，在javaee中，它们叫做EJB。

## 服务设计模式

在有状态和无状态之间，我的社交网站选择无状态服务。个人认为，在没有特定需求的情况下，都应该选择无状态模式。有状态模式适合于需要维持会话的情况，而纵观我这个网站的设计，所有状态都是靠cookie来传递，在主要功能上面没有什么东西需要用到会话。

在同步和异步上面，二者没有哪种会更好，只能具体问题具体分析。异步的主要应用范围，一是不需要返回消息的请求，比如后台处理的事务。一般后台会有个独立的server执行消息队列，其他server直接向它发消息，结果写在日志里面，不返回。第二个是不需要立即返回的情况，比如找回密码，一般的设计是发送一封验证邮件。但是面向用户的大多数服务，都是应该立即返回结果的。

说一下所用的组件。Javaee中ejb有三种，SessionBean是同步的，MDB是异步的。SessionBean里面又分为StatefulBean和StatelessBean，前者有状态，后者无状态。

# SaaS

我的服务器将来有可能会向用户提供SaaS服务，即把服务器划分出来虚拟的一部分，供用户部署自己的web应用。这项服务可以作为一个收费服务，提高公司的利润。

## 工作流程

用户首先到一个指定的页面提交申请，审批通过后会在一个专用的表中记录用户的信息，在一个特定的目录下面分配子目录，并分配子域名给用户。然后，用户需要选择他们所需的应用，由于不是PaaS，不能够自己上传，只能在我们提供的选项当中选。部署成功之后可以通过域名来访问，用户当然也可以绑定自己的域名来访问。

## 数据库

即使不同用户部署了同一款应用，他们的数据也应该是隔离的。这里我们采用了共享数据表但隔离记录的方法。这种方法最省主机的性能，但麻烦的是，应用的数据表和数据库的操作都需要一定改变。原先数据表中所有的主键，都要和用户id构成联合主键。用户id作为外键，引用上面所说的用户的记录。另外，应用内会保存用户id，进行数据库操作的时候会带上，确保用户只能操作自己的数据。

## 个性化

用户可以编辑html小工具和css来更改页面的样子，就像旧版的百度空间那样。为了安全起见，系统可能会对提交的内容做一些过滤。

# 大数据分析

对于一个社交平台来说，用户的行为记录本身是最有价值的数据，埋藏的无穷无尽的宝藏。

## 策略

经过一些考虑，我觉得可以从以下几点来分析：

1. 通过用户之间的关注操作，可以分析出用户间的关系图，可以找出一个用户的所有二度人脉，并向他推荐。
2. 通过用户所发的状态，可以提炼出一些标签来描述用户的特征。用户之间的标签可以做余弦相似性匹配，相似度比较高的人可以互相推荐。
3. 可以把博文提炼成关键字，再根据点击量，统计出当下最流行什么，或者用户最关注什么。
4. 可以对某一关键字的博文评论进行统计，判断用户对它的态度是正面的或者负面的。

## 数据模型

为了便于在分布式系统上存储和处理，记录采用mongodb存成一个个对象的形式。

一个对象大概是下面的样子：

{

“\_id”, //此条记录的id

“user\_id”, //操作人的id

“type”, //操作类型，比如发博客、发状态、关注之类

“target\_id”, //博客id、状态的id或者被关注人id

“target\_content”, //博客或者状态关键词，其他情况为空

“time” //操作时间

}

用mapreduce处理时，先根据type筛选出相应的操作类型，再挑选一个或几个属性进行处理。比如说，想知道被访问博客的关键词词频，就可以：

log.filter(o => o.type == VISIT\_BLOG)

.map(o => o.target\_content.split(‘ ’))

.reduce((res, li) => {

for(var e of li)

res[e] += 1;

return res;

}, {});

## 局限性

然而，大数据分析也有它的局限性。大数据分析只是静态的阐释表面现象，并不能够很好地解释背后的本质。举个例子，通过大数据分析，我们会得出冰淇淋的销量和溺水死亡的人数成正相关。然而我们都知道天气热是这两个现象的因，而不是他们互为因果。如果我们不知道背后得因果逻辑，而盲目的减少冰淇淋店的数量，效果不会有什么改善。

另外，大数据分析只能够为基于用户或者历史信息统计的领域提供决策帮助，比如销售、产品这种数据密集型的领域。但对于技术密集型的科研研发型领域，则帮助不大。