Pyneo4jet 项目说明

基于 Neo4j 图形数据库的微博网站

扈　煊　　1201214133

潘婉琼　　1201214139

周　志　　1201214118

2012 年 12 月 24 日

目录

1 项目简介

本项目是北京大学《海量图数据的管理与挖掘》的课程作业之一，主要是利用开源的图数据库 Neo4j 来构建一个基本的社交网站。

1.1 Neo4j

Neo4j 是一个用 Java 实现、完全兼容 ACID 的图形数据库。数据以一种针对图形网络进行过优化的格式保存在磁盘上。Neo4j 的内核是一种极快的图形引擎，具有数据库产品期望的所有特性，如恢复、两阶段提交、符合 XA 等。

开发者可以通过 Java-API 直接与图形模型交互，这个 API 暴露了非常灵活的数据结构。至于象 JRuby/Ruby、Scala、Python、Clojure 等其他语言，社区也贡献了优秀的绑定库。本项目使用的是 Neo4j 基于 Python 的 API。

Neo4j 的典型数据特征:

- 数据结构不是必须的，甚至可以完全没有，这可以简化模式变更和延迟数据迁移。
- 可以方便建模常见的复杂领域数据集，如 CMS 里的访问控制可被建模成细粒度的访问控制表，类对象数据库的用例、TripleStores 以及其他例子。
- 典型使用的领域如语义网和 RDF、LinkedData、GIS、基因分析、社交网络数据建模、深度推荐算法以及其他领域。

1.2 Pyneo4jet

线上部署 http://pyneo4jet.huxuan.org

代码托管 https://github.com/huxuan/pyneo4jet

主要功能

- 用户的注册、登录
- 用户资料/密码更新
- 好友的关注和取消关注
- 查看自己和好友的新鲜事
- 随便看看

2 项目分工

本项目使用的是 MVC（Model-View-Controller）模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

2.1 模型层

负责人 潘婉琼

编程语言 Python

代码文件 model.py

主要功能 封装与 Neo4j 数据库的读写交互，实现主要设计的对象类及其方法

数据结构 User 是用户类，提供用户与数据库的接口，包括用户的注册和登录验证、资料更新、关注和取消关注好友、得到自己的时间线、得到好友的新鲜事以及随便看看等；Tweet 是消息类，提供消息与数据库的接口，包括新鲜事的新建和删除等。

2.2 视图层

负责人 周志

语言 Bottle 的模板语法（主要是 HTML 和 Python 的结合）

代码文件 views 目录下的所有文件，主要是 tpl 模板和 css

主要功能 对用户信息友好的展示，提供简洁方便的交互。

文件说明 Bottle 是一种基于 Python 的快速、简单和轻量级的微 web 框架，通过对后缀为 tpl 的文件可实现模板的调用，通过 route 和控制回调实现对 css 文件的识别。其中 base.tpl 是基本模板，nav.tpl 是导航栏模板，signin.tpl 是登陆模板，signup.tpl 是注册模板，tweet_form.tpl 是状态发布模板，profile.tpl 是个人主页模板，password_update.tpl 是修改密码模板，profile_update.tpl 是更改个人信息模板，user.tpl 是好友列表模板，style.css 是网页的样式和布局。

2.3 控制器层

负责人 扈煊

编程语言 Python

代码文件 pyneo4jet.py

主要功能 服务器端的指令处理以及和模型层、视图层的交互。

2.4 数据解析脚本

负责人 扈煊

编程语言 Python

代码文件 data/parser.py

主要功能 解析原始数据，批量新建用户并利用用户信息模拟生成相应的消息。

2.5 服务器相关

负责人 扈煊

框架 Bottle [1]

服务器 Gevent [2]

反向代理 Nginx[3]

说明 Bottle 和 Gevent 都是基于 Python 并且非常轻量级，非常适合小项目使用，由于服务器上不止 Pyneo4jet 一个项目在运行，所以使用 Nginx 作为 80 端口的统一负载通过反向代理来实现 Pyneo4jet 的调用。

2.6 工作量统计

具体统计数据：https://github.com/huxuan/pyneo4jet/graphs/contributors

具体修改记录：https://github.com/huxuan/pyneo4jet/commits/master

[1] http://bottlepy.org/

[2] http://gevent.org/

[3] http://nginx.org/

3 页面功能

3.1 /

网站首页

功能说明

GET 默认跳转显示登录页面，如已登录则跳转到 timeline，如果带有参数 action=signup，则显示注册页面。

POST 处理登录和注册请求，如出错则返回原页面并显示错误信息，否则跳转到 timeline 页面。

3.2 /<username>/

用户信息页面

参数说明

username 当前浏览的用户名称

index 消息列表开始编号

功能说明

GET 默认显示用户信息页面，如果是当前登录用户则判断是否带有可能的 action 参数，如果参数 action=profile_update，则显示用户信息更新页面，如果参数 action=password_update，则显示用户密码更新页面。

POST 如果是当前登录用户，则处理用户信息更新和密码更新的请求，根据 action 参数的不同进行不同的修改。如果不是当前用户，则处理关注和取消关注的请求，根据 action 参数等于 follow 还是 unfollow 进行用户之间关注关系的修改。

3.3 /<username>/timeline/<index:int>

用户新鲜事页面

参数说明

username 当前浏览的用户名称

index 消息列表开始编号

功能说明

GET 获取当前浏览用户的信息列表，包括他和他好友最新的消息更新。

3.4 /<username>/tweets/<index:int>

用户消息页面

参数说明

username 当前浏览的用户名称

index 消息列表开始编号

功能说明

GET 获取当前浏览用户发送的最新消息列表

3.5 /<username>/followers/<index:int>

关注用户者页面

参数说明

username 当前浏览的用户名称

index 用户列表开始编号

功能说明

GET 显示关注当前浏览用户的用户列表

3.6 /<username>/following/<index:int>

用户关注者页面

参数说明

username 当前浏览的用户名称

index 用户列表开始编号

GET 显示当前浏览用户关注的用户列表

3.7 /<username>/random/

随便看看页面

参数说明

username 当前浏览的用户名称

GET 获取非当前浏览用户所发的随机消息列表页面

4 Neo4j 交互

4.1 User

4.1.1 添加新用户到数据库

函数名 `add(username, password, password_confirm, invitation)`

参数 **username** 用户名

password 用户密码

password_confirm 用户密码确认

invitation 邀请码

步骤

1. 判断该用户信息是否合法，如果不合法就返回 False 和错误信息。
2. 新建一个 user_node，将参数中的的信息拷贝入 user_node 中，并将该 user_node 加入到用户索引 user_idx 中。
3. 创建成功，返回 True。

代码

```
1  if not username or not password or not password_confirm:
2      return False, 'The username/password should not be
           empty!'
3  if password != password_confirm:
4      return False, 'The password you input twice is not the
           same!'
5  if invitation != INVITATION_CODE:
6      return False, 'The invitation code is invalid!'
7  user_node = user_idx['username'][username].single
8  if user_node:
9      return False, 'The username %s has been used!' %
           username
10 with db.transaction:
11     user_node = db.node()
12     user_node['name'] = username
13     user_node['username'] = username
14     user_node['password'] = password
```

```
15      user_idx['username'][username] = user_node
16  user = User.get(username)
17  return True, user
```

4.1.2 根据用户名找到对应的用户

函数名 `get(username)`

参数 **username** 该用户的名称

步骤
1. 根据用户索引 user_idx 找到所对应用户的 user_node。
2. 新建一个 User 类，将 user_node 的信息拷贝入 user 对象中。
3. 返回 User 对象

代码

```
1  user = None
2  user_node = user_idx['username'][username].single
3  if user_node:
4      user = User(username, user_node['password'])
5      user.name = user_node['name']
6      user.gender = user_node['gender']
7      user.hometown = user_node['hometown']
8  return user
```

4.1.3 关注好友

函数名 `follow(self, username)`

参数 **username** 好友的用户名

步骤
1. 根据用户索引 user_idx 找到 self 所对应用户的 user_node。
2. 判断 self 用户是否已经 follow 了 username 用户，如果是的话，就返回 False 和错误信息。

3. 根据用户索引 user_idx 找到用户名为 username 所对应用户的 follow_user。
4. 在 user_node 和 follow_user 之间建立 FOLLOW 关系，返回 True。

代码

```
user_node = user_idx['username'][self.username].single
for rel in user_node.FOLLOW.outgoing:
    f_node = rel.end
    if f_node['username'] == username:
        return False, 'You have followed %s !' % username
follow_user = user_idx['username'][username].single
with db.transaction:
    user_node.FOLLOW(follow_user)
return True, 'Follow user %s successfully!' % username
```

4.1.4 取消关注好友

函数名 `unfollow(self, username)`

参数 **username** 好友的用户名

步骤
1. 根据用户索引 user_idx 找到 self 所对应用户的 user_node。
2. 判断 self 用户是否已经 follow 了 username 用户，如果是的话，就删除该关系，并返回 True。
3. 如果没有返回 True 的话，就在函数最后返回 False。

代码

```
user_node = user_idx['username'][self.username].single
with db.transaction:
    for rel in user_node.FOLLOW.outgoing:
        f_node = rel.end
        if f_node['username'] == username:
            rel.delete()
            return True, 'Unfollowed user %s!' % username
return False, 'You haven\'t follow %s yet !' % username
```

4.1.5 找到关注该用户的好友

函数名 `get_followers(self, index=0, amount=10)`

参数 **index** 起始下标

amount 需要返回的好友个数

步骤

1. 根据用户索引 user_idx 找到 self 所对应的 user_from。
2. 新建一个 users_list，将所有关注 user_from 的好友加入该 users_list 中。该步骤中主要用到了 Neo4j 中 relationship 的方向。
3. 返回 users_list 中满足下标条件的元素。

代码

```
1 user_from = user_idx['username'][self.username].single
2 users_list = []
3 for relationship in user_from.FOLLOW.incoming:
4     user_to = relationship.start
5     user = User.get(user_to['username'])
6     users_list.append(user)
7 return users_list[index : min(index+amount, len(users_list
      ))]
```

4.1.6 找到该用户关注的好友

函数名 `get_following(self, index=0, amount=10)`

参数 **index** 起始下标

amount 需要返回的好友个数

步骤

1. 根据用户索引 user_idx 找到 self 所对应的 user_from。
2. 新建一个 users_list，将 user_from 关注的所有好友加入该 users_list 中。该步骤中主要用到了 Neo4j 中 relationship 的方向。

3. 返回 users_list 中满足下标条件的元素。

代码

```
user_from = user_idx['username'][self.username].single
users_list = []
for relationship in user_from.FOLLOW.outgoing:
    user_to = relationship.end
    user = User.get(user_to['username'])
    users_list.append(user)
return users_list[index : min(index+amount, len(users_list
    ))]
```

4.1.7 找到该用户发表的所有新鲜事

函数名 `get_tweets(self, index=0, amount=10)`

参数 **index** 起始下标

amount 需要返回的新鲜事个数

步骤

1. 根据用户索引 user_idx 找到 self 所对应的 user_from。
2. 新建一个 tweets_list，将 user_from 发表的所有新鲜事都加入该 tweets_list 中。该步骤中主要用到了 Neo4j 中 relationship 的方向。
3. 返回 tweets_list 中满足下标条件的元素。

代码

```
user_from = user_idx['username'][self.username].single
tweets_list = []
for relationship in user_from.SEND.incoming:
    tweet_node = relationship.start
    tweet = Tweet()
    tweet.text = tweet_node['text']
    tweet.username= tweet_node.SEND.outgoing.single.end['
        username']
    tweet.created_at = tweet_node['created_at']
    tweet.tid = tweet_node['tid']
```

```
10      tweets_list.append(tweet)
11  tweets_list.sort(key=lambda tweet: tweet.created_at,
        reverse=True)
12  return tweets_list[index : min(index + amount, len(
        tweets_list))]
```

4.1.8 返回该用户及其好友的新鲜事列表

函数名 `get_timeline(self, index=0, amount=10)`

参数 **index** 起始下标

amount 需要返回的新鲜事个数

步骤

1. 根据用户索引 user_idx 找到 self 所对应的 user_node。
2. 新建一个 tweets_list，将 user_node 及其好友发表的所有新鲜事都加入该 tweets_list 中。该步骤中主要用到了 Neo4j 中 relationship 的方向。
3. 对 tweets_list 按创建时间进行排序
4. 返回 tweets_list 中满足下标条件的元素。

代码

```
1  tweets_list = []
2  user_node = user_idx['username'][self.username].single
3  for follow_rel in user_node.FOLLOW.outgoing:
4      follow_node = follow_rel.end
5      for send_rel in follow_node.SEND.incoming:
6          tweet_node = send_rel.start
7          tweet = Tweet.get(tweet_node['tid'])
8          tweets_list.append(tweet)
9  for rel in user_node.SEND.incoming:
10     tweet_node = rel.start
11     tweet = Tweet.get(tweet_node['tid'])
12     tweets_list.append(tweet)
13 tweets_list.sort(key=lambda tweet: tweet.created_at,
       reverse=True)
```

```
14 return tweets_list[index : min(index + amount, len(
       tweets_list))]
```

4.1.9 随便看看，返回非该用户的随机新鲜事

函数名 `get_random_tweets(self, index=0, amount=10)`

参数 **index** 起始下标

amount 需要返回的新鲜事个数

步骤

1. 从 tweet_ref 处获得当前的 tweet 总个数 tot。
2. 新建一个 random_list，和变量 count 初始化为 0。
3. 在 (0,tot) 中随机生成一个 tid，判断该 tid 对应的 tweet 的用户是否为 self，如果不是的话，就判断 rancom_list 中是否已经加入该 tweet，如果没有加入过的话，就把该 tweet 加入 random_list 中。Count 记录的是循环的次数。
4. 重复上一步，直到 random_list 中的元素大于等于 amount 或者 count 大于 10 倍的 amount。
5. 返回 random_list。

代码

```
1  tot = tweet_ref['tot_tweet']
2  random_list = []
3  tids = set([])
4  count = 0
5  while(len(random_list) < amount):
6      count += 1
7      tid = random.randrange(0, tot)
8      tweet = Tweet.get(tid)
9      if tweet.username != self.username and tid not in tids:
10         random_list.append(tweet)
11         tids.add(tid)
12     if count > 10 * amount:
13         break
```

```
14 random_list.sort(key=lambda tweet: tweet.created_at,
       reverse=True)
15 return random_list
```

4.2 Tweet

4.2.1 添加新鲜事到数据库

函数名 `add(username, text, created_at)`

参数 **username** 发表该 tweet 的用户

text tweet 内容

created_at tweet 创建时间

步骤

1. 根据用户索引 user_idx 找到 username 所对应的用户的 user_node，新建一个 tweet_node，将该 tweet_node 与 user_node 之间建立一个 SEND 关系。
2. 根据数据库中的 tweet_ref 点得到目前 tweet 的总个数 N，然后给该 tweet 分配一个新的 tid（N+1）。
3. 将 tweet_node 加入到 tweet 索引 tweet_idx 中。

代码

```
1  user_node = user_idx['username'][username].single
2  if not user_node:
3      return False, 'User not found!'
4  if text:
5      with db.transaction:
6          tweet_node = db.node()
7          tweet_node['text'] = text
8          tweet_node['created_at'] = created_at
9          tweet_node.SEND(user_node)
10         tid = tweet_ref['tot_tweet']
11         tweet_node['tid'] = tid
12         tweet_ref['tot_tweet'] = tweet_ref['tot_tweet'] + 1
```

```
13          tweet_idx['tid'][tid] = tweet_node
14          return True, ''
15  else:
16  return False, 'Tweet should not be empty!'
```

4.2.2 根据 tid 找到对应的 tweet

函数名 get(tid)

参数 tid 该 tweet 的编号

步骤

1. 根据 tweet 索引 tweet_idx 找到 tid 所对应的 tweet 的 tweet_node。
2. 新建一个 Tweet() tweet，将 tweet_node 的信息拷贝入 tweet 对象中。
3. 返回 tweet

代码

```
1  tweet = Tweet()
2  tweet_node = tweet_idx['tid'][tid].single
3  if tweet_node:
4      tweet.username = tweet_node.SEND.outgoing.single.end['
           username']
5      tweet.text = tweet_node['text']
6      tweet.created_at = tweet_node['created_at']
7      tweet.tid = tweet_node['tid']
8      return tweet
```

5 心得体会

5.1 周志

通过本次实验，我学习了 python 的有关知识，尤其是基于 python 的 web 框架 bottle，对 git，github 版本管理有了简单的认识，进一步学习了 html 和 css 的相关知识，对网页布局和美化有了更深的了解。

在本次实验中，我对团队协调的有了更深的认识，刚开始考虑事情不够周全，在项目规范和开发工具使用上犯了一些错误，经过和组员交流我意识到了问题的严重性，对此在以后的项目开发上这个问题值得重视。

在本次实验中我要感谢两位组员在我遇到困难时给予的帮助和支持以及在我给项目增添麻烦时给予的理解和包涵。

5.2 潘婉琼

经过本次实验，我对图数据库有了更深入的理解，发现了图数据库与关系数据库的根本性不同，图数据库比关系数据库要灵活很多。

在编程的过程中，我学习了在 Linux 环境下使用 python 脚本语言进行编程和调试，并学会了 GIT 版本控制软件的简单使用方法。

在此要特别感谢扈煊同学对我耐心的帮助和指导，从他身上，我学到了很多程序员应该具备的专业素质。

5.3 扈煊

Pyneo4jet 实现了一个简单的微博网站所必须的基本功能，主要特色是用 Neo4j 图数据库代替了常规的关系型数据库如 MySQL 用于数据的存取，整个网站的控制层和视图层几乎没有涉及 Neo4j 的交互，所有与 Neo4j 的交互都由模型层封装成了相应的类和方法，这也是为什么本项目使用 MVC 三层架构模式的主要原因。

在使用图数据库的过程中，比较重要的一点是抛弃已经熟悉的关系型数据库的思维，尽可能的理解不同的 NoSQL 系统特有的特点和优势，例如 Neo4j 的索引和 id 概念，neo4j 的索引本质上就是一个 key-value 的 hash，id 在 neo4j 中也不是必要的元素，在 Pyneo4jet 中 User 的存取就直接将 username 作为 key 而且没有存储任何额外的类似于 id 的元素。此外在保存 user 及其所发的 tweet

以及 user 之间的 follow 关系时，使用的是 Neo4j 的 relati onship 而不必再使用额外的表来记录这些关系，这不仅简化了数据之间的逻辑，而且让代码也更加具有易读性，应该说正是 Neo4j 的精华所在。

与此同时，Neo4j 也有很多可以改进的地方，比如需要额外人工维护的索引是一个非常大的硬伤，应该至少支持某种规则的索引，这个规则可以是特定 type 的 relationship，例如 friends 关系，也可以是特定 type 的 node 的 properties 的 key，例如 user 的 username，然后将这一类规则的索引由系统自身维护，从而进一步增加 Neo4j 的集成度并减少 Neo4j 的使用者也就是开发者的工作。

在团队交流中，主要由于我的个人习惯，让另外两位队友都使用了 git/github 版本管理、python 语言以及 bottle 模块，可能是由于对这些工具接触不久，所以不少时间都用在了项目本身以外问题的沟通上，我切实感受到了两位队友短时间内在这些方面都有了很大的提升，在此也为在交流过程中由于自身性格原因有时过于生硬的言辞表示歉意，很高兴与你们一起顺利完成本次大作业。

当然，Pyneo4jet 作为本次课程的大作业，离真正的实用还有很大的差距，由于事先未能进行足够完备的设计，项目实现过程也过于仓促，尤其是控制器部分的内容，基本没有 Code Review，差不多都是以可以运行作为标准，难免很多缺漏的地方，甚至可能有跨站攻击的潜在危险，我们尽可能的将代码尤其是安全性方面进行了完善，并且部署在线上供老师、助教和其他同学亲身体验，如发现任何 bug 或者是任何建议，还望不吝指出。

参考文献

[1] Neo4j 说明文档，http://docs.neo4j.org/chunked/stable/index.html

[2] Neo4j python wrapper 简明文档，http://docs.neo4j.org/chunked/stable/tutorials-python-embedded.html

[3] Neo4j python wrapper 文档，http://docs.neo4j.org/chunked/stable/python-embedded.html

[4] Swaroop, C. H. "A Byte of Python.", 2003.

[5] Python 官方文档 (Python 2.7)，http://docs.python.org/2.7/

[6] Bottle 简明文档，http://bottlepy.org/docs/stable/tutorial.html

[7] Bottle 模板语法说明文档，http://bottlepy.org/docs/stable/stpl.html

[8] W3School HTML 教程，http://www.w3school.com.cn/html/

[9] W3School CSS 教程，http://www.w3school.com.cn/css/