CS209 · Computer System Design and Applications

KDD About Java Based on GitHub and StackOverflow

何泽安 12011323
王贵正 12011425
May. 21, 2022

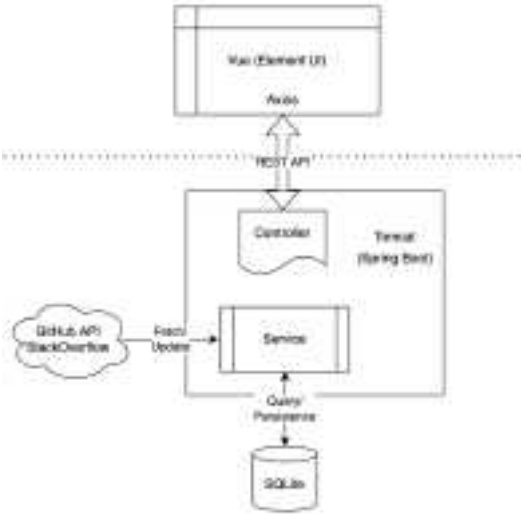
Contents

1 Abstract

In this project, we have designed a system that collects data from GitHub specific repositories and StackOverflow top questions and then analyze them, trying to find out ways to explain the thread of Java or the popularity of an OSSP. However, since we are not actually taking CS306, this project does not include compact analysis, but every knowledge points mentioned in this course are involved in this project.

2 System Design

The project is under the B-S (browser-server) architecture. We have the backend running in the Tomcat server (the service code are built base on Spring Boot), the service may uses the REST API provided by GitHub, or fetch (crawl) the webpage of StackOverflow, and save the data into database, it also provides controller for the REST API that used by the frontend. The frontend is based on Vue, it uses Axios as the client to send requests to the backend, we won't go deep in this report (in a word, it send requests carrying data as URL params to the proper REST API endpoint and parse the JSON response).



As for the code structure, we follow the design with low level dependency. Datum are stored in *entity*s that will be stored into database; to interreact with database, we have the *dao (data access object)* for help; for some analysis datum that will not be stored in the database but want to transfered to the frontend, we have *dto (data transfer object)* that only contains data. We also have *util* classes, including a *factory class* and a web proxy designed under *singleton pattern*. Complex business logics are placed inside *service*s, and *controller*s take hand on the REST API.

2.1 Data Collection

To collect the raw data of a specified repository, we choose to send requests to GitHub API. For each repository, the first time we query it in our project, it checks the database and cannot find the data, thus several calls to the GitHub API will be sent *asynchronized* to collect the metadata about the repo, issues, info of contributors (we also call another API that parse the location of user to the standard country name). Once the repository has been searched, its data will be saved in the database, in any later queries about the save repository, the data can be just selected from the database, letting one query can be respond within 200ms. As for the StackOverflow questions, we just want to show another technique, web crawler. It just mimic the human's behavior and fetch several pages with the filter set. Also, we use the database to cache the pages fetched.

Performance The baseline of data collection was using a loop containing synchronized methods, therefore, each loop need at least 1 RTT for data fetching, and the procession of next page has to wait for the former one. Then, after making the data fetching async, a.e. adding the processing of each page as a task, without waiting for it, we can continuously add other tasks, finally, collect the tasks' result. The performance enhancement of this could be considerable: assume there are n pages of issues we need to process, the synchronized old way takes $(t_{\text{fetch}} + t_{\text{proc}}) \cdot n$ while the new way only takes about $(t_{\text{fetch}} + t_{\text{proc}})$. By test, fetching a repository with 800 issues would take about 6s, with async data fetching, while the baseline cost 27s. Vice versa for the StackOverflow data requests.

2.2 IO

All the data (except the requests send by the frontend) flow into the backend in text. The GitHub part (api.github.com and api.positionstack.com) use JSON to contain the response. They can be convert into objects we defined by mapping the values into Java fields, here we chose to use *ObjectMapper* provided by *Jackson*. As for the StackOverflow webpages downloaded by the crawler, we *Jsoup* to parse the elements in the page and pick proper values into Java objects. As for the response, we just return them in the methods in the controllers, and they will be automatically encoded into Json by Spring.

2.3 Persistence

Data collected are saved into database (we chose *SQLite* because its ease of use) and can be override when a request with param *forceUpdate* was sent by the frontend. We designed models that can store the necessary data both in Java objects, and in database. *JPA* is the magic that makes this possible. We define objects that contains data, and use *OneToOne / OneToMany / ManyToMany* annotations that will instruct JPA to create proper connection tables and store the objects cascade into the database.

① Some Foreign Keys are not properly built, but it was JPA that handles them, still they can work;

② Some isolated tables are @Deprecated entities that not removed from the code.

To interreact with the database (CRUD), we just need to use the *JpaRepository* interface, there's even no single line of SQL in our code. Things like below are elegant and beautiful.

```java
@Repository
public interface GithubRepoDAO extends JpaRepository<GithubRepo, Long> {
    Optional<GithubRepo> findByNameAndOwner_UsernameAllIgnoreCase(String name, String username);
}
```

2.4 Specific Implementations

Take the most core class, GithubRepo, as an example. There are not only data fields like Long, that stored directly in the table of this class in database, but objects in other class, here we need to define the relationship between them by annotations to help JPA understand them and build the correct connection tables.

```java
public class GithubRepo implements Serializable {
    @Id
    private Long id;
    private String name;

    @ManyToOne(cascade = CascadeType.MERGE)
    private GithubUser owner;

    private Long stars;
    @JsonProperty("subscribers_count")
    private Long watchers;
    @JsonProperty("forks_count")
    private Long forks;
```

```
14
15      @OrderColumn
16      private String[] topics;
17      private String language;
18      private String description;
19
20      @OneToMany(cascade = CascadeType.MERGE)
21      @ToString.Exclude
22      private Set<StarLog> starHistOverlook;
23
24      private Date infoUpdatedAt;
25
26      private Integer issueCnt;
27      @OneToMany(cascade = CascadeType.MERGE)
28      @ToString.Exclude
29      private List<RepoIssue> issues;
30
31      @OneToMany
32      private List<RepoIssueLabel> labels;
33
34      @ManyToMany(cascade = CascadeType.MERGE)
35      @JsonIgnoreProperties
36      private List<RepoContributor> contributors;
37  }
```

Most datum stored in the database are the raw data, some data that can be analyzed extreamly fast are not stored (e.g. catagray the number of issue, group by year) since they are redundant. While some time costing analysis (e.g. NLP chunking words) are saved once finished.

Business logics, that analyze the data and interreact with DAO, are placed in service.

REST API are handled by controller. After the spring application is start, spring will start listening the port set in the conf, and requests to this port carries the URL including path and params.

```
1   @RestController
2   @RequestMapping("/api/gh")
```

```java
public class GithubRepoController {
    @CrossOrigin
    @GetMapping(value = "/fuzzy")
    @ResponseBody
    @SuppressWarnings("unchecked")
    public List<Map<String, String>> fuzzySearchRepoByName(@RequestParam String q)
    /* ... */
}
```

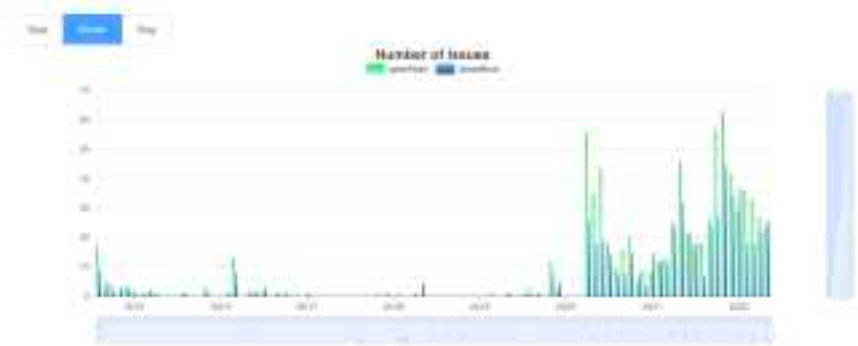
3 Data Analysis

For a specific repository on GitHub, there are different dimensions of data.

Repository Information We provide the basic information of the repository for reference in the form of table, including owner name, stars, description, language, forks, last update time, and the avatar of the owner. From which the user can access the information like if the main language of this repository is Java, if this repository is popular or widely praised by people.

Issues of Repository The following for aspects are applied.

Open and Close of Issues The issues' open can reflect the activity level or usage of the repository to some degree, while the issues' close change can reflect the variation of developers' reply or problem solving in frequency. Thus, we provide a bar chart to show the issues activity. For different repositories, the issues number or scale of change are quite different, then we provide three scales on time axis, with the time units can be one of year, month, or day.



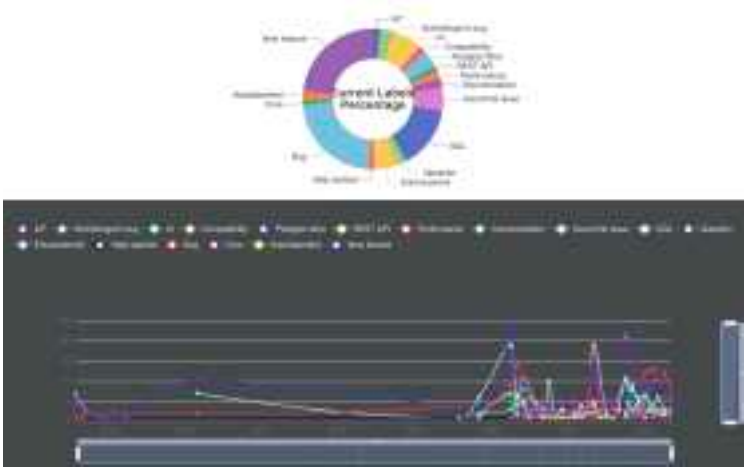
Feed-Forward Network to Predict Issues' Open Since the issues open can reflect the activity, we can roughly predict the future variation of that using the Feed-Forward Network with a linear model training on the actual data. For some of the repositories, the issues' open might present as a ladder pattern with respect to time (because of the amassed activities occur in a short time), which will make the prediction inaccurate, however, most of the repositories have an approximate straight line on the open issues-time chart, which means the approach is effective.

Note that dissipates the model is not good enough, we should focus more on the Java techniques applied. We use deepnetts' API to calculate the params of the model, more over, we need to first normalize the data. To do this, one may use Apache's math library, but that's time wasting since we already have one in the FeedForwardNetwork, but they are private. Here we use the reflection to access private fields in the model object!

```
FeedForwardNetwork model = FeedForwardNetwork.builder()
        .addInputLayer(1)
        .addOutputLayer(1, ActivationType.LINEAR)
        .lossFunction(LossType.MEAN_SQUARED_ERROR)
        .build();
try {
    //! Use java reflection to avoid calculate the params for normalization twice
    Field trainerField = NeuralNetwork.class.getDeclaredField("trainer");
    trainerField.setAccessible(true);
    Trainer mt = (Trainer) trainerField.get(model);

    Field trainMaxEpoch = BackpropagationTrainer.class.getDeclaredField("maxEpochs");
    trainMaxEpoch.setAccessible(true);
    trainMaxEpoch.set(mt, 100);
} catch (Exception e) {
    e.printStackTrace();
}
model.train(ds);
```

Labels of issues We found that there are lots of types of issues, which are represented as label, like 'bugs', 'questions', 'new features', etc. Thus, we count the number of issues separately with the labels. For labels like 'bug' which are sometimes paid more attention by users, we provide the legend in the chart, which enables user to select some labels and set only them visible, to make their variation more clear.

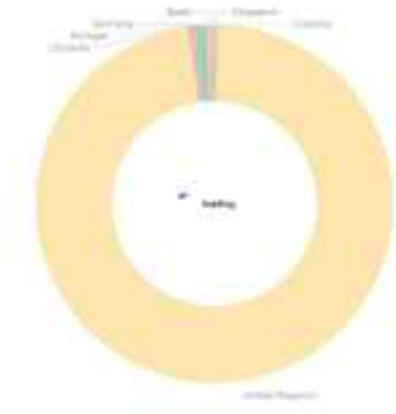


Word-cloud (NLP) With counting the word frequency of issues, we can obtain not only the technologies that the repository involved, but also the altitude of other developers to this repository. Since nouns or verbs are often in different categories (like most names of language or extensions are noun, here we used NLP word chunker to analyze), we divide them into two parts and provide two word-clouds.

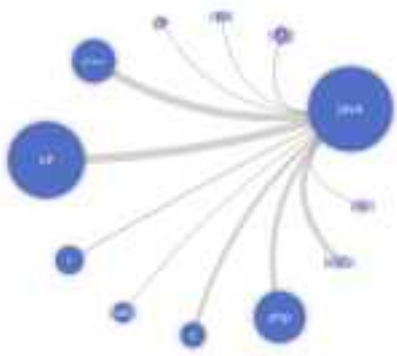
Besides, if the users double click on a word in the word cloud, our program will link to the specific Github issues searching result page with the word as the search keyword.

Contributions in the World For a famous or important repository, the user might consider where the contributors from. We accessed the location of contributors that they signed in their profile and analyzed the countries by program, and then calculated the total commits of a country, representing the amount of contributions from that country. The data was shown in the form of a pie chart, and an integer which is the contributors without a valid location in their personal profile.

For the questions on Stack Overflow, we mainly used cluster analysis. Beside a basic word cloud for all the questions tagged java with most frequency, we analyzed the degree of correlation between Java and other languages, by counting all the questions with both java tag and one other language tag (like C#, JavaScript, Python). This can represent how often some language is compared with java or used together with java, in other words, their relationship. The data in this part are shown in the form of relation graph, the closer the relationship is, the wider the connecting line will be.



Similarly, we analyzed the different kinds of exceptions of java which are asked about in Stack Overflow, and the size of circles in the relation graph represents the times a type of exception is tagged in StackOverflow questions.



4 Enhancing User Experience

For supporting users' interaction with the application, we provide an input search component which enables users to type in a string, and then we provide input suggestions (the string is used as keyword to make a fuzzy search on repository names) and auto complete functions. Thus, the users don't have to know the exact repository names and get the result they want in a short time.