Something about toyEngine

WEI YAN

wyglauk@gmail.com

Introduction

The toyEngine is a implementation of the search engine invert-index, along with basic and advanced functionalities. Specifically this program is currently providing the functionalities as services including: **a)** persisting and loading of lexicon / last posting unit id / term associated information; **b)** posting list persisting and lazily loading; **c)** double layers lock service for terms; **d)** posting unit lazily deleting and posting list cleaning; **e)** document adding and deleting; **f)** posting list accessing / visiting status recording and automatically deactivating; **g)** reloading inverted index for reallocating post unit IDs; **h)** posting list scanning and two simple document scoring models; **i)** three searching algorithms including plain, maxScore, WAND;

From the perspective of design, the program mainly consists of three parts, **a)** inverted-index and associated operations; **b)** entities supporting the implementations of the operations; **c)** helper classes like commonly used basic data structures and various utils. The design of entities are mostly applying the schema of “mainstay and plugins”, in which specific functionalities and even data structures are provided and maintained by the plugins, this is for the convenience of developing additional functionalities based on the current backbone.

Architecture Overview

The following figure shows the overall architecture of the program.

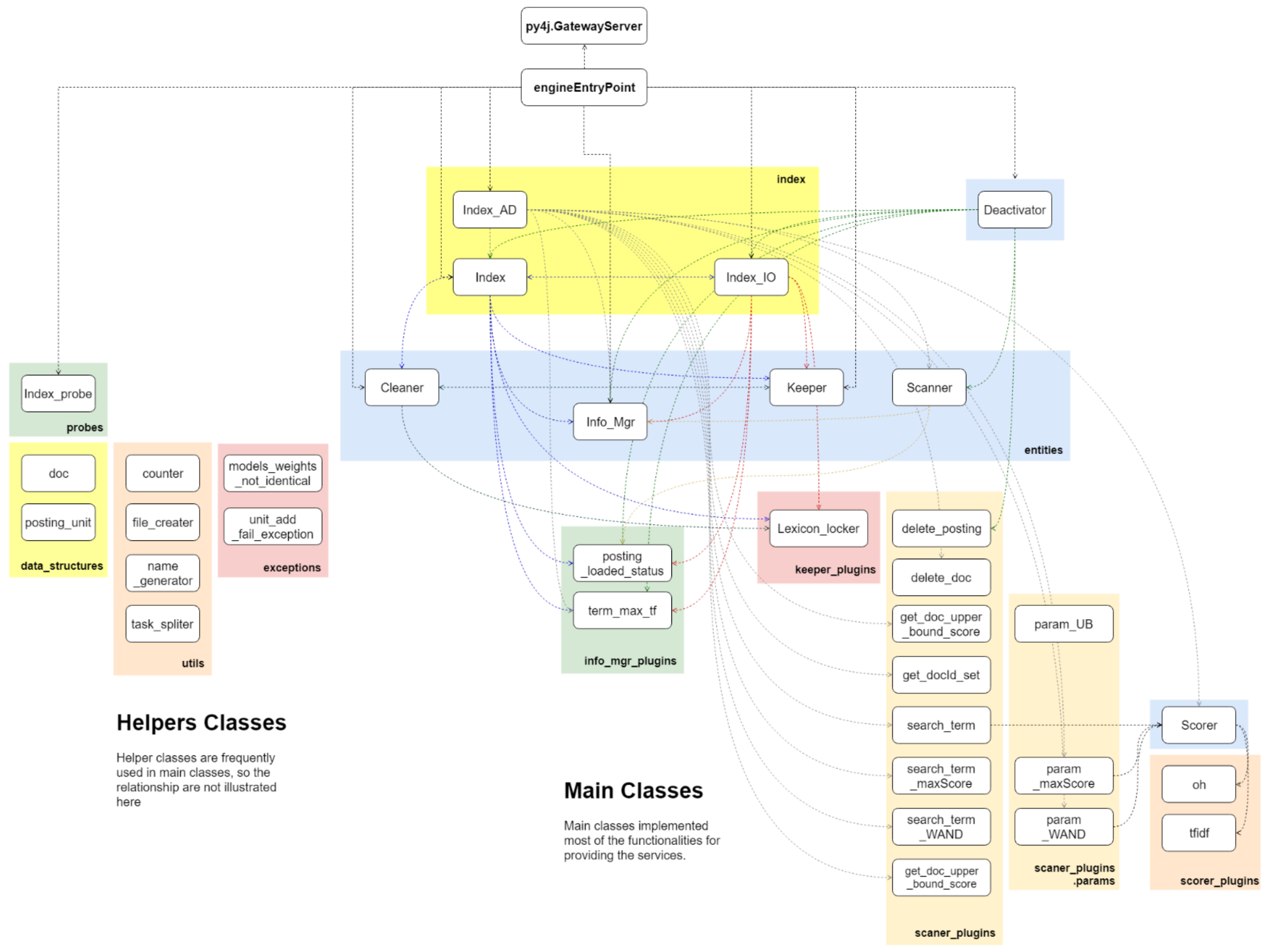
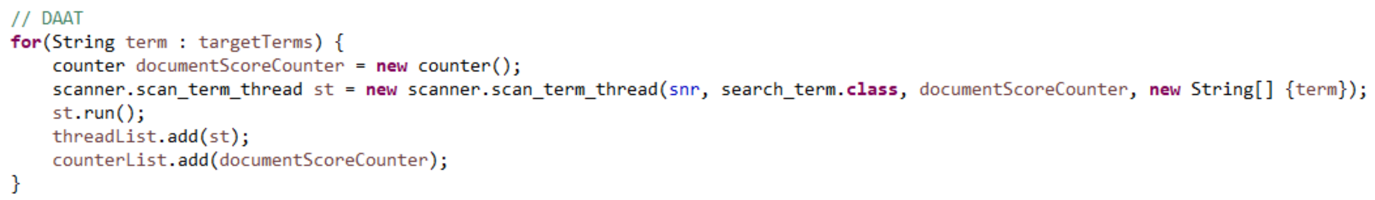


Fig 1. Overall architecture of toyEngine

On the left of figure 1, there are 9 helper classes belong to 4 packages, which are wildly used in other parts of the program; on the right, the rest classes belong to 6 packages (count to depth 2) consist the main classes which provide most of the functionalities provided as services.

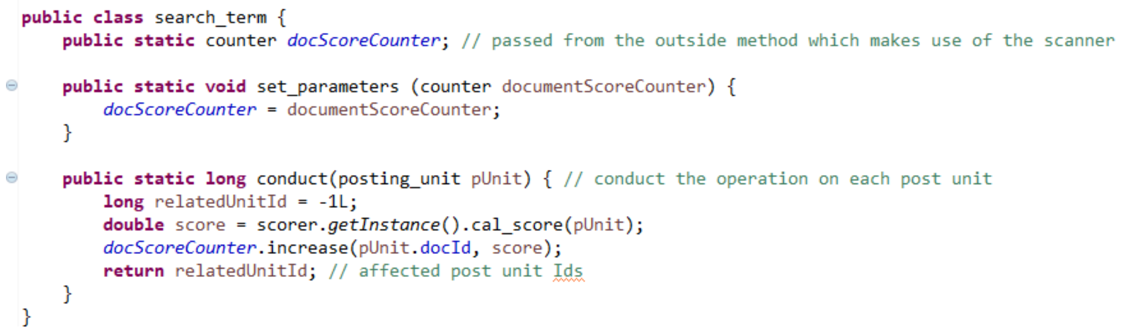
To be more specific, on the right of figure 1, classes are in fact belong to 3 categories: **a)** classes in packages *index* (with yellow background); **b)** classes belong to *entities* package (with blue background); **c)** The rest classes which are plugins or parameters of plugins of corresponding entity classes.

To explain why the structure is designed as this, let us use the plain search functionality, which is defined as *index_advanced_operations.search*, as an example:



```
// DAAT
for(String term : targetTerms) {
    counter documentScoreCounter = new counter();
    scanner.scan_term_thread st = new scanner.scan_term_thread(snr, search_term.class, documentScoreCounter, new String[] {term});
    st.run();
    threadList.add(st);
    counterList.add(documentScoreCounter);
}
```

Fig 2. Code snippet of index_advanced_operations.search.



```
public class search_term {
    public static counter docScoreCounter; // passed from the outside method which makes use of the scanner

    public static void set_parameters (counter documentScoreCounter) {
        docScoreCounter = documentScoreCounter;
    }

    public static long conduct(posting_unit pUnit) { // conduct the operation on each post unit
        long relatedUnitId = -1L;
        double score = scorer.getInstance().cal_score(pUnit);
        docScoreCounter.increase(pUnit.docId, score);
        return relatedUnitId; // affected post unit Ids
    }
}
```

Fig 3. Detail in scanner_plugins.search_term.

It can be seen from figure 2 that the function instantiated the *scanner.scan_term_thread* by pass the *scanner_plugins.search_term.class* into it, *documentScoreCounter* is a counter object worked as the parameter of *search_term* for collecting the score of document contributed by posting units.

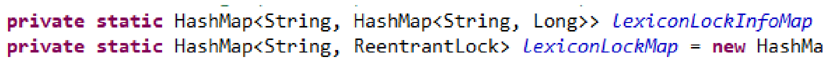
By defining the *search_term* as plugin of *scanner*, split the common functionality of scanning through the posting list with the specific service functionality of calculating the contribution score from one posting unit to the document. **This allows the developer does not need to care about the implementation details in scanner when need to develop some functionalities involve the scanning of posting list, as long as one follows the common structure of plugin classes - plugin needs to implement the *set_parameters* method and *conduct* method**.

For the similar reason, entities *information_manager*, *keeper*, *scorer* and *scanner* are defined with plugins to provide specific functionalities.

Interesting Things

Double Layers Term Lock

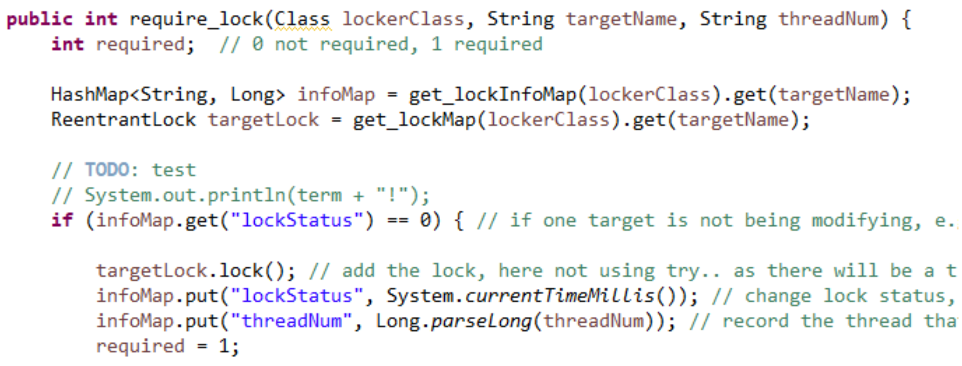
When adding or deleting posting units during the serving of engine, we need to make sure the synchronization of modifying inverted-index, mainly *index.lexicon* and *index.postUnitMap*, so we need to add locks on terms. In this program the locks are manipulated through the methods defined in *entities.keeper*, but the actual lock pool is provided by the plugin of keeper *entities.keeper_plugins.lexicon_locker*:



```
private static HashMap<String, HashMap<String, Long>> LexiconLockInfoMap
private static HashMap<String, ReentrantLock> LexiconLockMap = new HashMa
```

Fig 4. Maps in lexicon_keeper.

As can be seen in figure 4, the *lexicon_locker* provides two maps, *lexiconLockInfoMap* and *lexiconLockMap*. The first one maps from term string to the information map which contains the *lockStatus* and *threadName*. *lockStatus* could set as 0 or a timestamp, and threadName denotes the thread successfully required the lock. lexiconLockMap contains the actual *ReentrantLock* associated with the term, which is created when new term is added into the inverted-index.



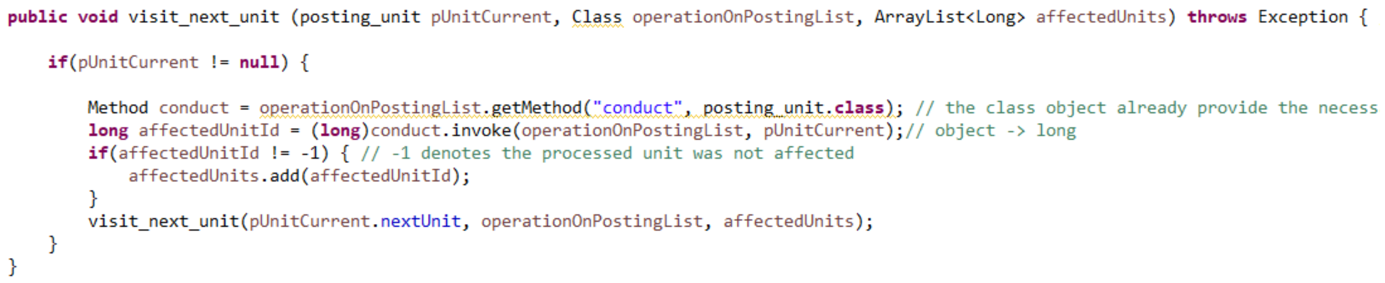
```
public int require_lock(Class lockerClass, String targetName, String threadNum) {
    int required;  // 0 not required, 1 required

    HashMap<String, Long> infoMap = get_lockInfoMap(lockerClass).get(targetName);
    ReentrantLock targetLock = get_lockMap(lockerClass).get(targetName);

    // TODO: test
    // System.out.println(term + "!");
    if (infoMap.get("lockStatus") == 0) { // if one target is not being modifying, e.

        targetLock.lock(); // add the lock, here not using try.. as there will be a t
        infoMap.put("lockStatus", System.currentTimeMillis()); // change lock status,
        infoMap.put("threadNum", Long.parseLong(threadNum)); // record the thread tha
        required = 1;
```

Fig 5. Snippet of entities.keeper.require_lock.

Figure 5 shows the snippet of *entities.keeper.require_lock*, **it can be seen that the *lockStatus* is checked firstly, then the lock is added, and then status and information are recorded, this is why it is called "double layer"**. This design is useful when there are multiple threads try to add new documents into inverted-index, each threads needs to add many posting units so that needs to require lock on the corresponding terms, if without the first layer checking *lockStatus*, all the threads try to add units of the same term will be holding up and wait for the lock, however, **with such *lockStatus* checking, only threads getting status as 0 at exactly same time will be waiting, other threads will just return a failed requirement flag and skip this unit, as a result the overall efficiency is boosted**. To ensure all the units are eventually added, a retry logic is implemented in the *index.add_doc*.

Scanner and Plugins

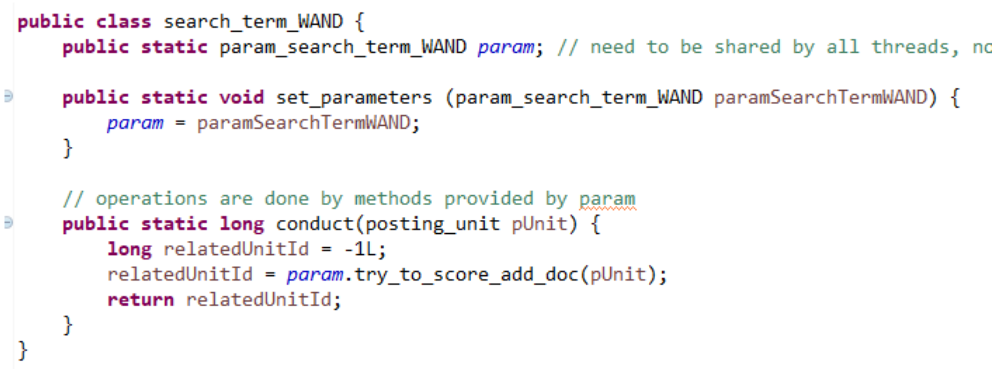
In architecture overview section, the usage of scanner is discussed as an example of schema "mainstay and plugins", here go deeper into the implementation of scanner.



```
public void visit_next_unit (posting_unit pUnitCurrent, Class operationOnPostingList, ArrayList<Long> affectedUnits) throws Exception {

    if(pUnitCurrent != null) {

        Method conduct = operationOnPostingList.getMethod("conduct", posting_unit.class); // the class object already provide the necess
        long affectedUnitId = (long)conduct.invoke(operationOnPostingList, pUnitCurrent);// object -> long
        if(affectedUnitId != -1) { // -1 denotes the processed unit was not affected
            affectedUnits.add(affectedUnitId);
        }
        visit_next_unit(pUnitCurrent.nextUnit, operationOnPostingList, affectedUnits);
    }
}
```

Fig 6. entities.scanner.visit_next_unit.

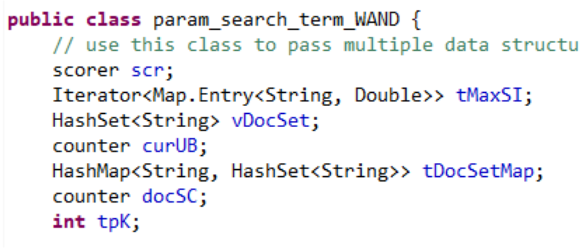
Figure 6 shows the core method if scanner, which recursively accesses the *nextUnit* field of *posting_uint* object which points to the next posting unit, and invokes the *operationOnPostingList.conduct* method on each posting_unit, for *index_advanced_operations.search*, it is *scanner_plugins.search_term*; for *index_advanced_operations.search_maxScore*, it is *scanner_plugins.search_term_maxScore*; for *index_advanced_operations.search_WAND*, it is *scanner_plugins.search_term_WAND*, etc. All these plugins implements two methods *set_parameters* and *conduct*. Use *search_term_WAND* as an example:



```
public class search_term_WAND {
    public static param_search_term_WAND param; // need to be shared by all threads, nc

    public static void set_parameters (param_search_term_WAND paramSearchTermWAND) {
        param = paramSearchTermWAND;
    }

    // operations are done by methods provided by param
    public static long conduct(posting_unit pUnit) {
        long relatedUnitId = -1L;
        relatedUnitId = param.try_to_score_add_doc(pUnit);
        return relatedUnitId;
    }
}
```

Fig 7. scanner_plugins.search_term_WAND.

As shown in figure 7, *search_term_WAND* implements the two common methods of plugins. However, what need to be noticed is, in *search_term_WAND* the parameter is not simple data structure like *search_term* has, instead it is a new param data structure *param_search_term_WAND*.



```
public class param_search_term_WAND {
    // use this class to pass multiple data structu
    scorer scr;
    Iterator<Map.Entry<String, Double>> tMaxSI;
    HashSet<String> vDocSet;
    counter curUB;
    HashMap<String, HashSet<String>> tDocSetMap;
    counter docSC;
    int tpK;
```

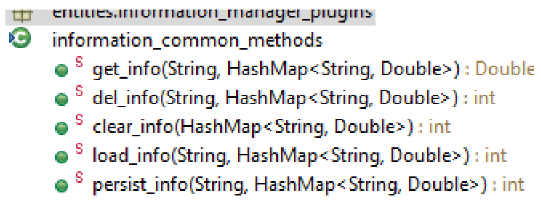
Fig 8. Snippet of scanner_plugins.parameters.param_search_term_WAND.

Figure 8 shows a snippet of *param_search_term_WAND*, which wraps up the necessary data structures passed from *index_advanced_operations*, also implements the logic of scoring the document in *param_seawrch_term_WAND.try_to_score_add_doc*. The reason of implementing the scoring process in the param instead of in plugin is for the convenience of making use of the wrapped data structures.

Scanner provides two types of threads *scan_term_thread* and *scan_term_thread_with_lock* for making use of the scanner with multi-threads, the second one require term locks before the scanning, which is used in deactivator for deleting posting units of the deactivated terms.

Information_manager and Plugins

It is very much like the idea of *keeper*, *information_manager* works as a unified interface to the information maps maintained in its plugins. Among *information_manager_plugins*, there is a class *information_common_methods* provides the shared common methods among plugins.

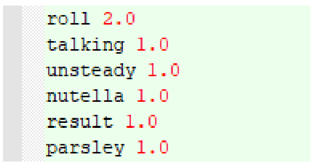


```
entities.information_manager_plugins
information_common_methods
    get_info(String, HashMap<String, Double>) : Double
    del_info(String, HashMap<String, Double>) : int
    clear_info(HashMap<String, Double>) : int
    load_info(String, HashMap<String, Double>) : int
    persist_info(String, HashMap<String, Double>) : int
```

Fig 9. Common methods of information_manager_plugins.

Figure 9 shows the methods provided by *information_common_methods*, in fact a more natural implementation approach is making a parent class provides all the common methods and set several abstract methods, it is for the convenience of using reflections that most of the data structures and methods provided by plugins are made static, in the next version of toyEngine, this could be changed.

Additionally the current existing two plugins are *posting_loaded_status* and *term_max_tf*. The first one record information of when is a term loaded / access last time, so that this information is set when posting units are created / loaded / scanned. Deactivator also rely on the posting_loaded_status to check if a posting list expired, which means a posting list is not accessed for a pre-set period. The *term_max_tf* is for recording the max term frequency of a term, such that is frequently updated when new posting units are added, currently this information is persisted to a *persistence/information/term_max_tf*, which looks like:



```
roll 2.0
talking 1.0
unsteady 1.0
nutella 1.0
result 1.0
parsley 1.0
```

Fig 10. Persisted term_max_tf.

In next version, a new basic data structure *term* should be extract out like *posting_unit*, and information like max term frequency could be persisted along with the lexicon then.

Deactivator and Posting List Automatically Deactivation

..

Cleaner and Lazily Deleting of Posting Units

..

Persistence

..

toyEngine_operator

..