Technical Details

胡雨軒

November 30, 2014

2.

Contents

Chapter 1

Méthodes d'accès et de stockage des données

Analyser un fichier et construire une représentation abstraite de l'information qu'il contient à partir des informations que l'on en obtient ligne par ligne est un problème d'analyse syntaxique[1] dont il faut ensuite enregistrer les résultats pour pouvoir les utiliser plus tard. Plusieurs solutions sont envisagées dans le cadre de ce micro-projet.

1.1 Analyse syntaxique d'un fichier texte

Programmation impérative ou tentative de programmation fonctionnelle.

1.2 Stockage des données

L'idée la plus naturelle est d'obtenir en sortie du programme deux fichiers textes qui contiennent une liste de nœuds et une liste d'arêtes mais cette idée mène très rapidement à utiliser une représentation temporaire de la structure de composition des sinogrammes. On peut donc également construire une représentation temporaire des objets, l'afficher puis éteindre le programme sans en garder trace. Il est enfin envisageable d'enregistrer le résultat obtenu dans une base de données pour faciliter son exploitation ultérieure.

[1] Un analyseur syntaxique est un *parser* en anglais.

Nous avons bénéficié des lumières de Messieurs ROBERT et TALEL, professeurs respectivement d'architecture des ordinateurs et de bases de données que nous remercions, pour explorer ces trois voies : représentation temporaire, génération de listes et utilisation de base de données. Ces trois voies posent chacune des questions intéressantes en optimisation et modèle relationnel.

Plusieurs choix s'offrent à nous pour le stockage des sinogrammes. La solution la plus pérenne est de passer par une base de données et donc d'utiliser le disque dur. Cela pose des contraintes de , mais quelle est la solution la plus rapide ? Tous les caractères ont une IDS mais seulement les caractères explicites ont un point de code.

1.2.1 Listes d'arêtes et de nœuds

1.2.2 Base de données MariaDB

Notule Je remercie M. TALEL, qui donne le cours `inf225`, de bien avoir accepté de répondre à mes questions.

1.2.3 La carte de données `PairMap`

Notule Je remercie M. ROBERT, qui donne le cours `inf227`, de bien avoir accepté de répondre à mes questions.

Statégies d'optimisation

La structure de données dans laquelle on stocke les caractères est un dictionnaire qui a pour clef l'IDS d'un caractère (ou son hash). Il faut une structure qui évite complètement les redondances, c'est-à-dire que la forêt des sinogrammes n'ait pas de doublon quel que soit l'ordre dans lequel les caractères sont entrés et quel que soit le détail des IDS. `PairMap` n'a pas vocation a être générique : des structures existent déjà pour cela mais à offrir la solution la plus adaptée au stockage des sinogrammes.

Trois stratégies d'optimisation des accès mémoires :

- Un tas ;
- Un accès aléatoire ;
- Un arbre couvrant.

Stratégie du tas

- Chaque caractère possède ses propres composants ;
- La taille d'un tas est 3 ;
- Solution tributaire des IDS ;
- Explosion combinatoire pour mettre à jour ;
- Analyse de caractère rapide.

Stratégie de l'accès aléatoire

- Lecture et écriture rapide si vraiment aléatoire ;
- Chaque sinogramme contient un tableau de référence de composants.

Stratégie de l'arbre couvrant

- Hypothèse : faiblement connecté ;
- Euh… mais ce n'est vraiment pratique à parcourir !

Un sinogramme comporte trois références vers ses composants et pour l'immense majeure partie deux références. Si le sinogramme est une entrée de la table on a un caractère, un point de code et une IDS. Si c'est un sinogramme induit on n'a qu'une IDS. Certains sinogrammes d'un pan Unicode (*unicode block*) sont décomposés en sinogrammes qui n'ont pas de décomposition dans ce pan mais se décomposent dans un autre pan. L'ordre de décomposition des sinogrammes n'est pas respecté globalement mais seulement à l'intérieur d'un pan. Bien qu'un point de code et une IDS renvoient à une même réalité, il faut les voir comme deux sous-clefs, facette d'une même clef.

Tous les caractères ont une IDS mais seulement les caractères explicites ont un point de code. Les caractères implicites n'ont pas de point de code, seulement une IDS. L'IDS peut donc être vue comme une clef principale à côté de laquelle on place lorsque c'est possible un point de code. L'organisation interne de `PairMap` devra donc être un dictionnaire `(K1, V)` avec `K1` une IDS et `V` une référence vers un objet `Node`. Il y a également un dictionnaire `(K2, K1)` avec `K2` un point de code. Le dictionnaire réciproque `(K1, K2)` n'est nécessaire que pour économiser la résolution

d'une référence : on peut en effet accéder à `K2` très facilement à partir de `K1 : K1 → V.K2`.

L'interface de `PairMap` est finalement un dictionnaire `(K1, K2, V)`.

La classe `TreeMap` peut-elle apporter quelque chose de plus que `HashMap` ?

1.3 Pourquoi `PairMap` n'est pas générique

Ajouter la généricité en Java a été un *tour de force*[2]. Cf. l'article chinese-huawen/gj-oopsla java genericity.pdf

http://stackoverflow.com/questions/3403909/get-generic-type-of-class-at-ru

> Generics are not reified at run-time. This means the information is not present at run-time.
>
> Adding generics to Java while mantaining backward compatibility was a tour-de-force (you can see the seminal paper about it: Making the future safe for the past: adding genericity to the Java programming language).
>
> There is a rich literature on the subject, and some people are dissatisfied with the current state, some says that actually it's a lure and there is no real need for it. You can read both links, I found them quite interesting.

http://stackoverflow.com/questions/1004022/java-generic-class-determine-typ

> In contrast to .NET Java generics are implemented by a technique called "type erasure".
>
> What this means is that the compiler will use the type information when generating the class files, but not transfer this information to the byte code. If you look at the compiled classes with javap or similar tools, you will find that a List<String> is a simple List (of Object) in the class file, just as it was in pre-Java-5 code.
>
> Code accessing the generic List will be "rewritten" by the compiler to include the casts you would have to write yourself in earlier versions. In effect the following two code fragments are identical from a byte code perspective once the compiler is done with them:
>
> Java 5:

[2] En français dans un texte anglais.

> List<String> stringList = new ArrayList<String>(); stringList.add("Hello World"); String hw = stringList.get(0);
>
> Java 1.4 and before:
>
> List stringList = new ArrayList(); stringList.add("Hello World"); String hw = (String)stringList.get(0);
>
> When reading values from a generic class in Java 5 the necessary cast to the declared type parameter is automatically inserted. When inserting, the compiler will check the value you try to put in and abort with an error if it is not a String.
>
> The whole thing was done to keep old libraries and new generified code interoperable without any need to recompile the existing libs. This is a major advantage over the .NET way where generic classes and non-generic ones live side-by-side but cannot be interchanged freely.
>
> Both approaches have their pros and cons, but that's the way it is in Java.
>
> To get back to your original question: You will not be able to get at the type information at runtime, because it simply is not there anymore, once the compiler has done its job. This is surely limiting in some ways and there are some cranky ways around it which are usually based on storing a class-instance somewhere, but this is not a standard feature.

https://github.com/megamattron/collections-generic/blob/master/src/java/org/apache/commons/collections15/MultiMap.java Des gens ont créé le symétrique : une clef, plusieurs valeurs.

https://github.com/megamattron/collections-generic/blob/master/src/java/org/apache/commons/collections15/keyvalue/MultiKey.java On a aussi une solution pour *agréger* des clefs je n'ai pas trouvé plusieurs clefs de types différents pour une valeur.

Je veux absolument de la généricité par exemple la fonction get : si on a K1 = Integer et K2 = String, comment faire si on a get([Integer]) pour savoir dans quel dictionnaire on doit chercher ? On pourrait tester dans un dictionnaire puis dans l'autre mais ce n'est pas optimisé. D'autre part, pour la fonction put(Pair<K1, K2> key, V value) avec une paire

Si on fait un champ de type K1 on ne peut pas récupérer son type. Une solution communément conseillée (parameterizedtype) ne marche que pour les classes abstraites... type erasure est très fort ah ah

1.4 Evolution du paradigme de programmation

Introduction du paradigme fonctionnel.

On remarque dans le premier listing que l'exécution est purement séquentielle. On pourrait cependant parcourir séquentiellement le tableau et lancer un `Thread` pour analyser chaque ligne mais on se heurterait à des problèmes de cohérence. Ici, `AliasMap` est remplacé par deux dictionnaires : le dictionnaire principal `dictionary` et le dictionnaire des alias `alias`.

Listing 1.1: Impératif pur. Premier jet sans `AliasMap`

```
Parser<Node, RowChise> parser;
parser = new Parser<>(files, 25000);
Iterator<RowChise> iterator = parser.iterator();

while (iterator.hasNext()) {
⟶RowChise row = iterator.next();

⟶if (row.getCharacter().contains("灣") || row.getCharacter().
    contains("縲")) {
⟶⟶System.out.print("");
⟶}

⟶Node node = new Node(row.getCharacter(), row.getSequence());

⟶alias.put(node.getCharacter(), node.getId());
⟶dictionary.put(node.getId(), node);

⟶main++;
}
```

On utilise maintenant `AliasMap`. Le fonctionnel est sous-jacent mais pas encore directement visible puisque caché dans `AliasMap`.

Listing 1.2: Impératif pur. Avec `AliasMap`

```
Parser<Node, RowChise> parser;
parser = new Parser<>(files, 25000);
Iterator<RowChise> iterator = parser.iterator();

while (iterator.hasNext()) {
⟶RowChise row = iterator.next();

⟶if (row.getCharacter().contains("灣") || row.getCharacter().
    contains("縲")) {
⟶⟶System.out.print("");
```

```
10 ⟶}
11
12 ⟶Node node = new Node(row.getCharacter(), row.getSequence());
13
14 ⟶try {
15 ⟶⟶aliasMap.put(new Alias<Integer, String>(node.getId(), node
       .getCharacter()), node);
16 ⟶} catch (UndefinedAliasException e) {
17 ⟶⟶e.printStackTrace();
18 ⟶}
19
20 ⟶main++;
21 }
```

L'extrait de code suivant correspond à changer la forme sans toucher au fond. On utilise bien un itérateur mais on applique une action sur chacun de ses éléments en utilisant une syntaxe fonctionnelle. C'est un premier pas qui ne révolutionne cependant pas grand'chose.

Listing 1.3: Premier pas de fonctionnel

```
1  Parser<Node, RowChise> parser;
2  parser = new Parser<>(files, 25000);
3  Iterator<RowChise> iterator = parser.iterator();
4
5  iterator.forEachRemaining(x -> {
6
7  ⟶if (x.getCharacter().contains("灣") || x.getCharacter().
       contains("縲")) {
8  ⟶⟶System.out.print("");
9  ⟶}
10
11 ⟶Node node = new Node(x.getCharacter(), x.getSequence());
12
13 ⟶try {
14 ⟶⟶aliasMap.put(new Alias<Integer, String>(node.getId(), node
       .getCharacter()), node);
15 ⟶} catch (UndefinedAliasException e) {
16 ⟶⟶e.printStackTrace();
17 ⟶}
18 ⟶main++;
19 });
```