

分散木構造データベース Jungle による 企業向け許認可システム



琉球大学工学部情報工学科

金川竜己

指導教員 河野真治

目次

第 1 章	知的構造を格納するためのデータベース	1
1.1	研究目的	1
第 2 章	分散木構造データベース Jungle	3
2.1	Jungle のデータ構造	3
2.2	分散機能	5
第 3 章	組織中の許認可管理	
	アプリケーション maTrix	6
3.1	maTrix の保持するデータ構造	6
3.2	Jungle 上での maTrix のデータ構造の表現	9
3.3	maTrix のデータの版管理	13
3.4	Jungle での maTrix の版管理の表現	14
3.5	申請の許認可	15
3.6	XACML	16
3.7	Jungle 上の許認可	17
第 4 章	Jungle に新しく追加した API	19
4.1	検索 API の実装	19
4.2	Jungle 上の Index の設計	21
4.3	過去の Tree に対するアクセス	23
4.4	JungleXMLReader	24
4.5	XACMLInterpreter	24
第 5 章	検索 API の評価	26
5.1	測定方法	26
5.2	測定環境	26
5.3	検索の API の測定	27
第 6 章	結論	28
6.1	まとめ	28
6.2	今後の課題	28
6.2.1	push/pop	28

6.2.2	index \mathcal{O} IncrementalUpdate	29
6.2.3	differentialList	29
6.2.4	exponential backoff	29

目 次

2.1	破壊的木構造の編集	3
2.2	非破壊的木構造の編集	4
2.3	非破壊的木構造の編集	4
2.4	Jungle-network のトポロジの形成例とデータの伝搬順序	5
3.1	組織モデル表現例	7
3.2	Jungle 上での人物 Tree の表現例 (1)	10
3.3	Jungle 上での人物 Tree の表現例 (2)	11
3.4	maTrix の構成情報例 1	13
3.5	maTrix の構成情報例 1	13
3.6	構成情報モデル Tree 表現例	14
3.7	XACML のデータ構造	16
4.1	find の検索順序	20
4.2	FindInSubTree の検索順序	21
4.3	ParentIndexExample	22
4.4	XACMLInterpreter で用いられている stack	25
5.1	inActive の実行時間	27
6.1	TreeMap の Get	30

表 目 次

3.1	maTrix の保持する木構造データの例	6
3.2	Person.xml の要素	9
3.3	図 3.2 に対応した XML	10
3.4	図 3.3 に対応した XML	12
3.5	Jungle に対応した XML 文例	12
3.6	構成情報 Tree の TreeNode が保持している Attribute	14
3.7	Rule の評価	17
3.8	XACML のルール結合アルゴリズム	17
4.1	ParentIndex の戻り値	22
4.2	sax の主要なイベント一覧	24
5.1	実験環境	26
6.1	データアクセス関数	31

第1章 知的構造を格納するためのデータベース

1.1 研究目的

我々があつまっている知識は木構造であることが多い。例えば書籍や組織情報などである。RDB上に木構造を格納する際、nodeとnodeの関係を定義したテーブルを用意し、表構造に変換するが、木構造のままデータベースに格納することで、直接的な操作や効率的なサービスが可能になる考えられる。

また、データベースを使用するウェブサービスの規模も年々大きなものとなり、それに比例して負荷も増大し、その結果サービスが停止する自体が多々見られるようになった。そのため、データベースの処理性能はそのままサービスの質につながっている重要な項目となっている。

データベースの処理性能を向上させる代表的な方法として、ハードウェア的に高性能なマシンを用意することで処理性能を上げるスケールアップと、汎用的なマシンをいくつも用意し、処理を分散させることで処理性能を上げるスケールアウトの2つがある。単純に処理能力を上げたいのなら、スケールアップは有効ではあるが、単一のマシンを高性能にするのにも限界があり、いずれはそのマシンの限界を超える負荷がかかる可能性もある。それに対しスケールアウトは、処理が重くなるに連れて汎用的なマシンを順次追加していくことで性能を上げるため、ハードウェア的に高性能なマシンを要求せずすみ、柔軟な対応を取ることが出来るため、データベースの性能を上げる方法としてはスケールアウトが求められている。本研究で扱うスケラビリティとはスケールアウトのことをさす。

分散データシステムにおいて、データの整合性(一貫性)、常にアクセスが可能であること(可用性)、データを分散させやすいかどうか(分割耐性)、この3つを同時に保証することは出来ない。この定義をCAP定理と言う。

今、最も使われているデータベースであるRDBは、一貫性と可用性を重視している。そのため、データを分割し、複数のノードにデータを分散させると、整合性のチェック等の問題があり、スケールアウトを行うことが困難である。

当研究室では、これらの問題を解決した、煩雑なデータ設計が必要ないスケラビリティのあるデータベースを目指して、非破壊的木構造データベースJungleを開発している。Jungleは、全体の整合性ではなく、木ごとに閉じた局所的な整合性を保証している。また、整合性のある木同士をマージすることで新しい整合性のある木をす繰り出すことも可能であるため、データの伝搬も容易である。そのため、Jungleは可用性と分割耐性を重視しつつ、整合性も持つ。

Jungle は、これまでの開発によって木構造を格納する機能をもっている。そこで、当研究では、Jungle 上に組織の許認可管理アプリケーション maTrix を実装し、データベースの表現力、機能の充分性、実用的な性能実証実験を行う。

第2章 分散木構造データベースJungle

Jungle とは、当研究室で開発を行っている、スケーラビリティのある、世の中の知識構造を煩雑なデータ設計を行うことなく格納できることを目標としたデータベースである。本章では、Jungle の基本的な特徴についての解説を行う。

2.1 Jungle のデータ構造

研究目的でも述べたが、我々が扱う知識は主に木構造である。Jungle はそれらの知識を直接格納するため、データ形式は非破壊的木構造を採用している。はじめに、非破壊的木構造と通常破壊的木構造の違いについて説明を行う。

通常破壊的木構造は、データの編集を行う際に、データを上書き更新する(図 2.1)ため、編集を行っている間ずっと木にロックをかける必要がある。また、閲覧者がいる場合検索途中でデータが変わることを避けるために、データの検索が終わるまで書き換えを待つ必要がある。しかし、これではロックによりスケーラビリティが損なわれてしまう。

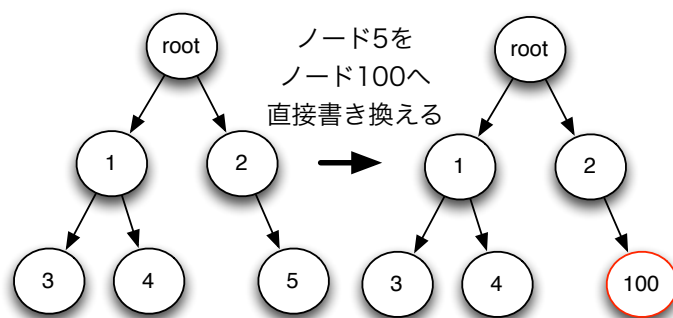


図 2.1: 破壊的木構造の編集

それに比べ非破壊的木構造は、一度生成した木を上書きすることはない。データの編集は、ルートから編集を行うノードまでコピーを行い新しく木構造を構築することで行う(図 2.2)。

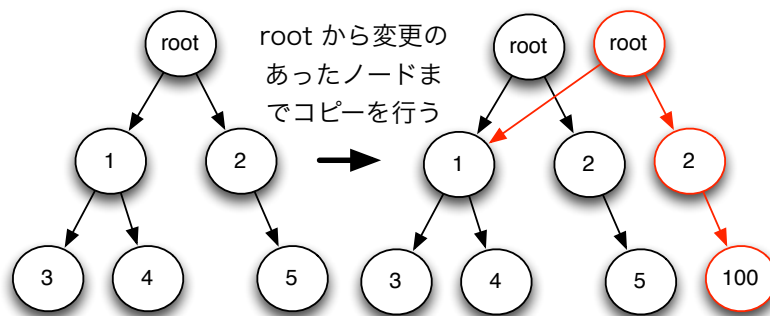


図 2.2: 非破壊的木構造の編集

非破壊的木構造においてデータのロックが必要になる部分は、木のコピーを作った後に、ルートノードを更新するときだけである。また、データ編集を行っている間ロックが必要な破壊的木構造に比べ、非破壊的木構造は検索中の木が変更されないことが保証されているため、編集集中においてもデータの読み込みが可能である。(図 2.3) そのため、破壊的木構造に比べてスケールアウトがしやすくなっている。

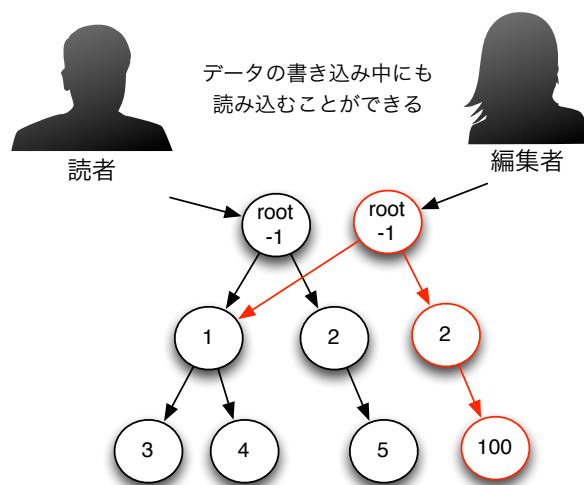


図 2.3: 非破壊的木構造の編集

また、Jungle は過去の version の Tree を全て保持しているため、いつでもアクセスすることが可能である。

2.2 分散機能

Jungleの分散機能は、当研究室で開発を行っている並列分散フレームワークである Alice を使用している。Alice はユーザーが望んだマシンへの接続や、必要なデータへのアクセスを行う機構等、ネットワークトポロジー形成機能を提供している。

Jungle は、ネットワークトポロジーを構築する際に、木構造を想定したネットワークトポロジーを形成しサーバー同士を接続することで通信を行っている。木構造なら、一度 RootNode までデータを伝搬させることで整合性を取ることが出来る (図 2.4) からである。データの伝搬中に衝突が発生した場合、Merge を行い結果を改めて伝搬すれば良い。

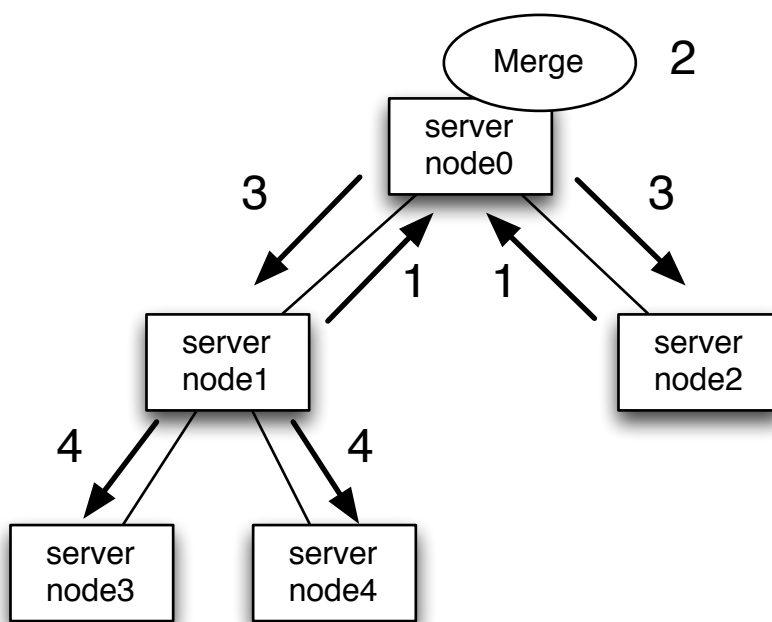


図 2.4: Jungle-network のトポロジーの形成例とデータの伝搬順序

第3章 組織中の許認可管理

アプリケーション maTrix

maTrix とは Symphonies 社が開発しているアカウント管理、許諾判定システムのことである。人、組織、役割等の情報などを木構造として持っている、それらの情報を関連付けることで組織の構造を表現し、許認可管理を行う。本章では maTrix の説明と、Jungle 上における maTrix の実装方法の説明を行う。

3.1 maTrix の保持するデータ構造

matrix は人、役職、役割、役割記述要素といったデータ (表 6.1) を用いた組織構造と、許認可の判断に用いるポリシーファイルの 2 つのデータを持っている。

表 3.1: maTrix の保持する木構造データの例

人物	人のデータが入っている。personId を持つ
組織	組織のデータが入っている。orgId を持つ
役割	人や組織の役割データが入っている。roleId を持つ
役割記述要素	役割が持っている要素が記述されている。rdeId を持つ

これらのデータは Id を用いて他の木の参照を行い、組織構造を構築している。組織構造の例として、人物と組織の組み合わせで組織運用モデルを表現しており、役割は自身のみで役割モデルを表現している。このように様々な木構造が参照しあうことで多数のデータモデルを表現している。以下に maTrix の組織情報モデルを記す (図 3.1)。

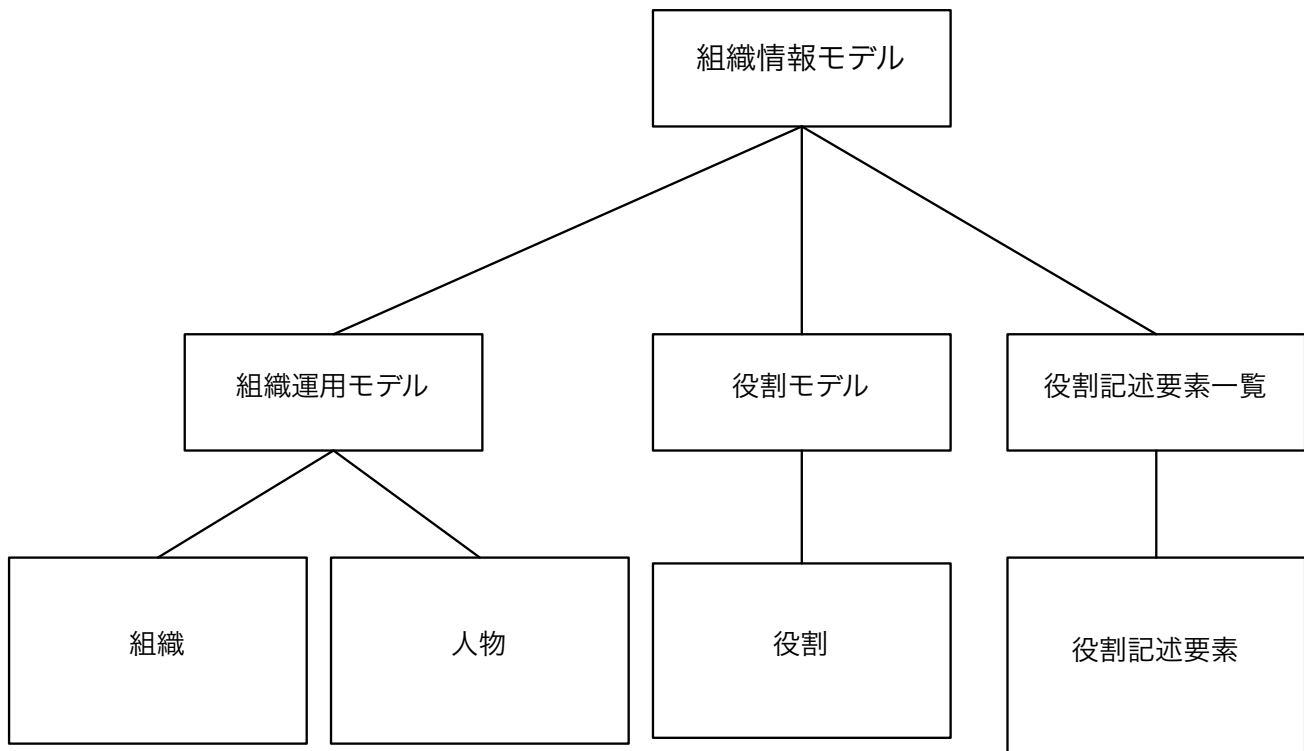


図 3.1: 組織モデル表現例

maTrix のデータ構造は、保持している組織情報を xml や json 形式で出力することができる。以下に人物 Tree を xml 形式で出力したデータの一部として、1 人分のデータを記述する

```
<Persons>
  <Person id="p:1" type="Person">
    <accountId>a:26</accountId>
    <lastName>東</lastName>
    <name>東俊一</name>
    <nameReading>あずましゅんいちくん</nameReading>
    <roleRefIds>r:10 r:34</roleRefIds>
    <parentOrganizations type="OrganizationMappedByRole">
      <OrganizationMappedByRole type="OrganizationMappedByRole">
        <organizationRefId>o:2</organizationRefId>
        <roleRefId>r:10</roleRefId>
      </OrganizationMappedByRole>
      <OrganizationMappedByRole type="OrganizationMappedByRole">
        <organizationRefId>o:11</organizationRefId>
        <roleRefId>r:34</roleRefId>
      </OrganizationMappedByRole>
    </parentOrganizations>
    <priorities type="PriorityMappedByRole">
      <PriorityMappedByRole type="PriorityMappedByRole">
        <priority>0</priority>
        <roleRefId>r:10</roleRefId>
      </PriorityMappedByRole>
      <PriorityMappedByRole type="PriorityMappedByRole">
        <priority>1</priority>
        <roleRefId>r:34</roleRefId>
      </PriorityMappedByRole>
    </priorities>
  </Person>
</Persons>
```

表 3.2: Person.xml の要素

Persons	この要素以下に Person の情報があることを意味する
Person	人の情報が以下にあることを示す。PersonId が割り振られている
accountId	その Person のアカウント Id。Tree 同士の参照に用いる。
lastName	苗字
name	フルネーム
nameReading	名前のふりがな
roleRefIds	役割の Id を記述する
parentOrganizations	この要素以下にその人が所属している組織の Id を記述する
OrganizationMappedByRole	この要素以下に組織と、その組織の役割を記述する
organizationRefId	所属している組織の Id
priorities	人物に割り振られている役割の優先順位を以下に記述する
PriorityMappedByRole	この要素以下に役割と優先順位をペアで記述する
priority	役割の優先順位を記述する

Person.xml を例で上げたが、役職や役割も同じような構造でデータを保持しており、maTrix には組織情報からデータを取得する Function が 15 種類実装されている。

3.2 Jungle 上での maTrix のデータ構造の表現

maTrix が保持している、人物や役職は、木構造のデータであるため直接 Jungle に格納できる。実際に人物のデータを格納した Jungle の構造を (図 3.2、図 3.2) に示した。

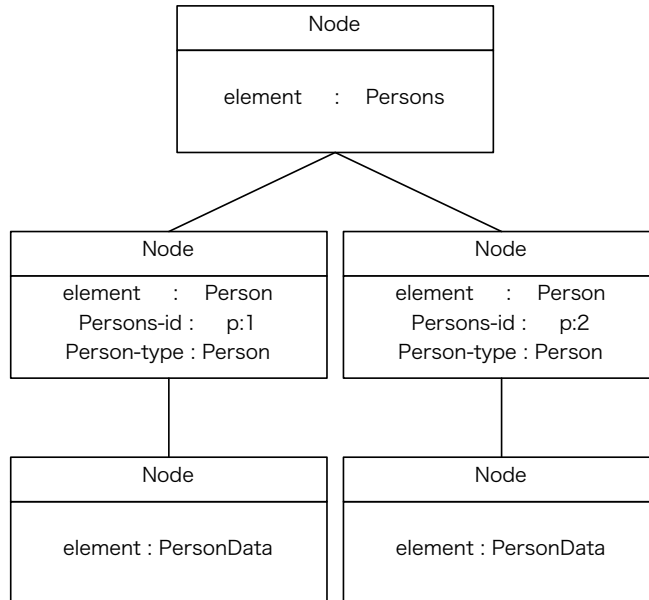


図 3.2: Jungle 上での人物 Tree の表現例 (1)

表 3.3: 図 3.2 に対応した XML

```

<Persons>
  <Person id="p:1" type="Person">
    <PersonData> </PersonData>
  </Person>
  <Person id="p:2" type="Person">
    <PersonData> </PersonData>
  </Person>
</Persons>
  
```

図 3.2 は、表 3.3 を Jungle に格納したものである。Jungle は、TreeNode に<Key String, Value Attribute>の組み合わせでデータを保持する。そのため、XML のデータを TreeNode に格納する際、要素の格納は< Key "element", Value "要素名">、属性の格納は<Key "要素名-属性名", Value "属性値">で格納する。属性格納時の Key を、"要素名-属性名"にした理由は、Index で検索を行う際に Key の重複を避けるためである。

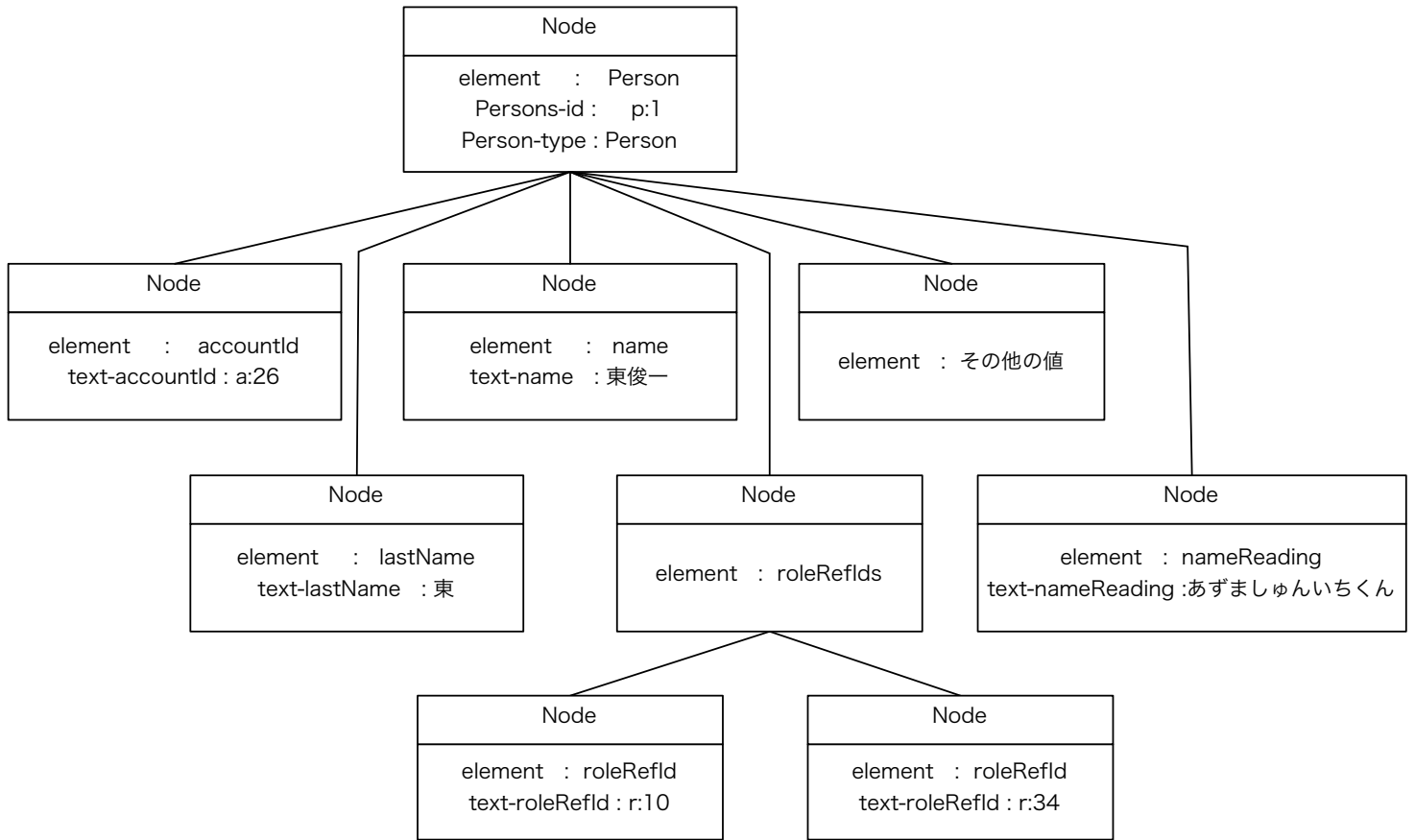


図 3.3: Jungle 上での人物 Tree の表現例 (2)

表 3.4: 図 3.3 に対応した XML

```
<Person id="p:1" type="Person">
  <accountId>a:26</accountId>
  <lastName>東</lastName>
  <name>東俊一</name>
  <nameReading>あずましゅんいちくん</nameReading>
  <roleRefIds>r:10 r:34</roleRefIds>
  <その他の値> </その他の値>
</Person>
```

図 3.3 と表 3.4 も同じく PersonTree の一部を Jungle に格納した時の図である。基本的な構造は図 3.2 と変わらないが、1 箇所 XML の構造を変えて Jungle に格納した箇所があるのでその解説を行う。Jungle はノードにデータを格納する際に、List 構造を持つことが出来ないため、key に対応する value は 1 つしか格納できない。よって、表 3.4 の <roleRefIds>r:10 r:34</roleRefId> の様に、2 つの値をテキスト部分が保持している場合どちらか片方しか格納することが出来ない。そこで、読み込む際にデータ構造を、表 3.5 の様に書き換えて Jungle に格納する。

表 3.5: Jungle に対応した XML 文例

```
<roleRefIds>
  <roleRefId>r:10</roleRefId>
  <roleRefId>r:34</roleRefId>
</roleRefIds>
```

この様に maTriX のデータ構造を Jungle にマッピングしていく。また木構造同士の参照を用いて行う組織構造の表現は、Jungle では Tree に対する検索で表現する。

3.3 maTrix のデータの版管理

maTrix は、組織構造を構成情報モデルとして表現し、版管理している。構成情報モデルは version 毎に存在し、各 version に対応した組織構造を構成する木構造の集合を保持している。

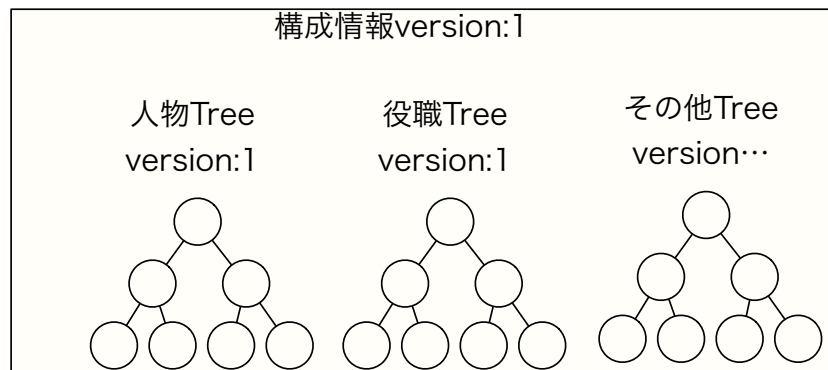


図 3.4: maTrix の構成情報例 1

人物と役職の Tree の version が共に 1 の時、構成情報モデルの version も 1 とする。

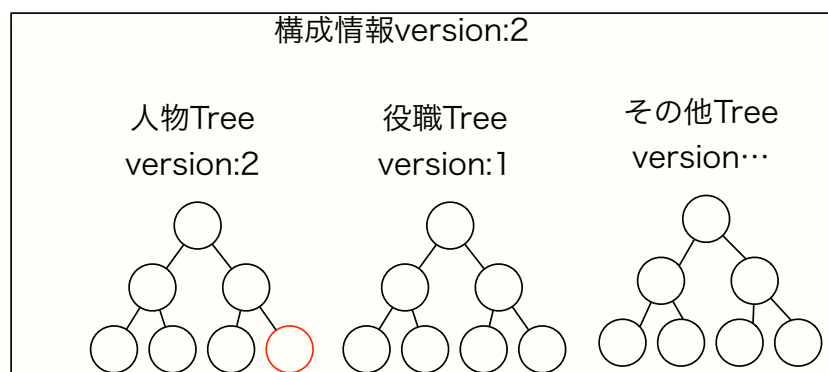


図 3.5: maTrix の構成情報例 1

構成情報モデルの version:1 の人物 Tree を更新した場合、version:2 の構成情報が構築され、maTrix は version:1 の構成情報と version:2 の構成情報の両方を保持し、両方の構成情報にアクセスすることが可能である。

3.4 Jungle での maTriX の版管理の表現

Jungle の Tree は、全ての version の Tree で単一な Id を保持しているため、Id を指定すれば、過去の Tree にアクセスすることが可能である。それを利用し、Jungle 上での過去の変更履歴を保持する構成情報モデルの表現は、構成情報モデルの version と、各 Tree の version を保持し、関連付ける構成情報モデル Tree を作成し表現した。(図 3.6)。

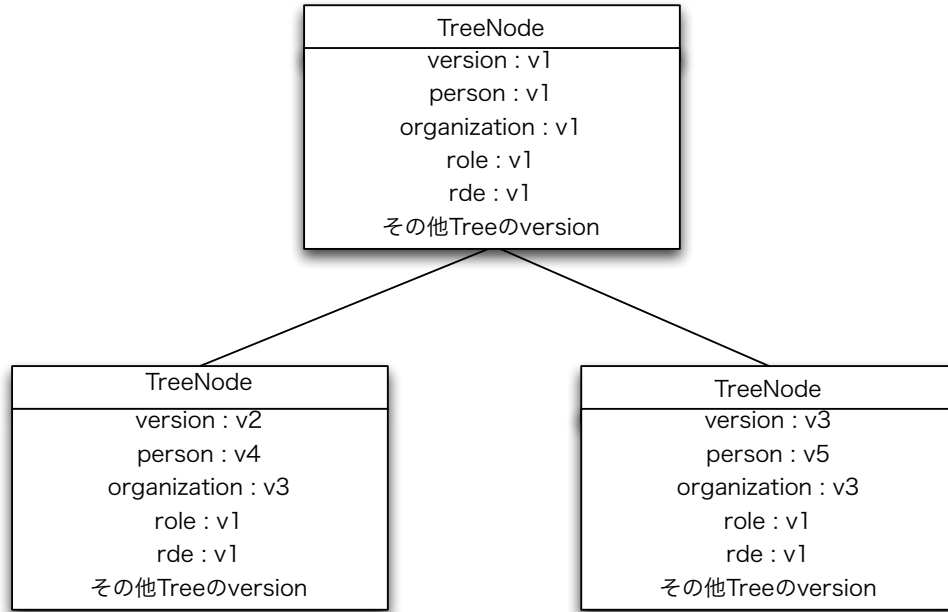


図 3.6: 構成情報モデル Tree 表現例

表 3.6: 構成情報 Tree の TreeNode が保持している Attribute

version	構成情報モデルの version
person	構成情報モデルの version に対応する人物 Tree の version
organization	構成情報モデルの version に対応する組織 Tree の version
role	構成情報モデルの version に対応する役割 Tree の version
rde	構成情報モデルの version に対応する役割記述要素 Tree の version

実際にどのように Jungle 上で過去の構成情報モデルにアクセスするか、例題を用いて説明する。構成情報モデル version:3 に対応する人物 Tree にアクセスする手順を以下に示す。

1. 構成情報 Tree からアクセスしたい version 情報を保持している Node を取得する (今回の例題では version3)
2. 取得した TreeNode には、構成情報:version:3 に対応した人物 Tree などの version が記述されているため、人物 Tree の version(v:5) を取得する

3. 2 で取得した version の人物 Tree にアクセスする。

といった手順で Jungle では maTrix の構成情報モデルを表現する。

3.5 申請の許認可

maTrix を用いた許認可は、組織構造だけで判断されるわけではない。アクセス可能な時間や、貸出の場合は同時貸出数の上限など、組織構造以外のアクセスルールは組織構造だけでは表現できない。そのため、maTrix はアクセス管理のルールを表現方法を定義する XACML という言語で記述された、ポリシーファイルを用いて許認可判断を行う。ポリシーファイルは、リポジトリで管理されており、アクセス要求にあったポリシーファイルが利用される。以下に maTrix での申請の許認可の流れを記述する。

1. A さんが、学科のノート PC の借りるために、maTrix に貸出許可を求める。
2. maTrix はリポジトリから、貸出許可を与えるかを判断するためのポリシーを取得する。
3. ポリシーファイルを元に、データにアクセスを行い権限を与えるかどうかを判断する。
4. maTrix は権限に応じて、ノート PC を A さんに貸出許可を与える。

といった流れになる。

maTrix の許認可を使用するメリットとして、許認可のログをとっておくことで、いつ、誰が、どのポリシーを元に、どんなことをしたか、の情報がいつでも取得可能であるため、データの不正改竄等の問題発生時の解決等役立つ。といったメリットもある。

3.6 XACML

本節では maTrix のポリシーファイルの記述に用いられている XACML についての説明を行う。XACML は、データに関するアクセス要求について、その要求元の情報と要求の内容、アクセス対象の組み合わせから、そのアクセス要求が許可されるか否認されるかを判断するためのルールを記述できる。実際に使用する際は、XACML 自体がアクセス制御を行うのではなく、アクセス管理アプリケーションが XACML を参照しアクセス制御を行う。XACML は以下の様なデータ構造を持つ (図 3.7)

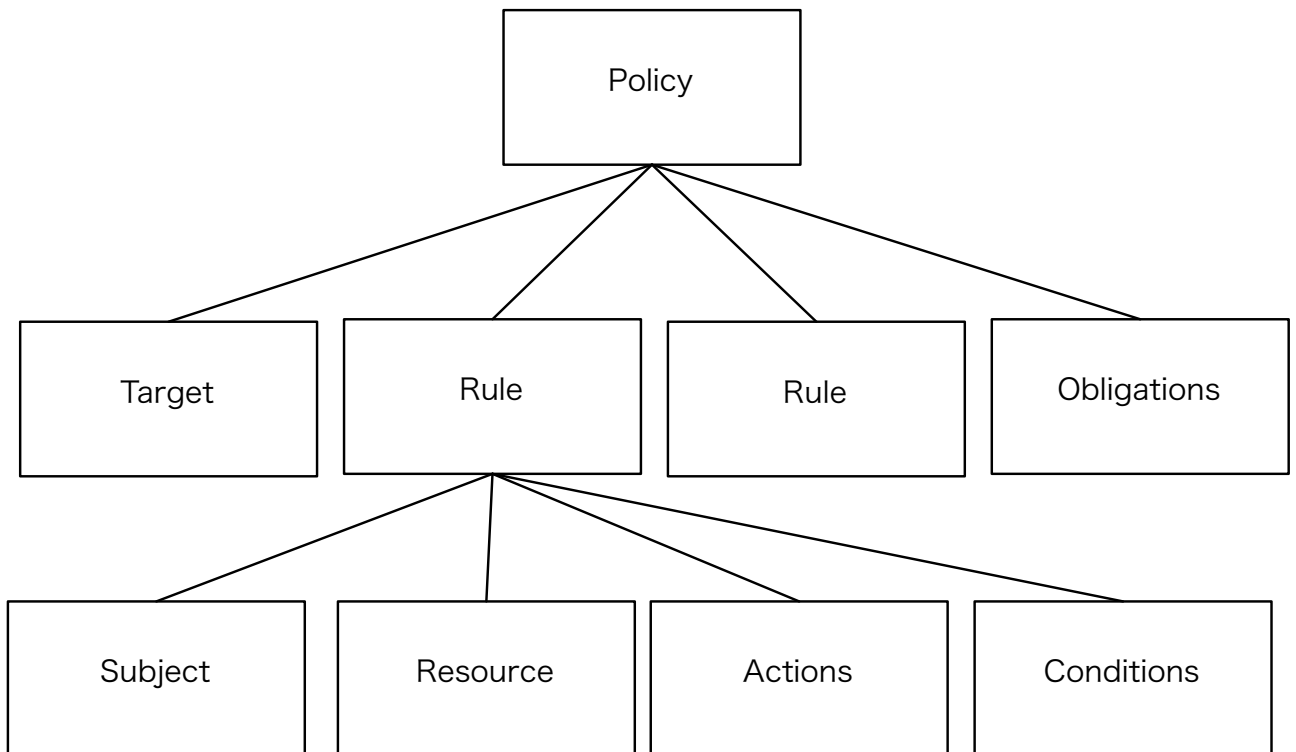


図 3.7: XACML のデータ構造

Target は、誰に対して (Subject)、どのような資源を (Resource)、どのように扱うか (Action) の3つの要素を持ち、それぞれの要素で、評価関数を用いて評価を行う。評価の結果は、Match、No-Match、Indeterminate(評価不能)の3つである。

Rule は、Target に対する規則を定めるもので、ルールの方針にそってアクセス要求がこのルールに適合した場合に決定する値 (Permit、Deny) を、Effect として設定しておく。また、Target とは別にオプションとして条件 (Conditions) を付けることも可能である (条件の例としては、アクセスを許可する時間の指定などがある)。ルールの評価は、Target の評価関数の結果と Condition の、ルール結合アルゴリズム (表 3.7) にそって結合する。

表 3.7: Rule の評価

Target	Condition	評価
Match	true	Efferct(Permit or Deny)
Match	false	NotApplicable
Match	Indeteminate	Indeteminate
No-Macth	-	NotApplicate
Indeterminate	-	Indeteminate

Policy

複数の Rule をまとめたものを Policy という。Policy の子要素には、Target、Rule、Obligations(責務)がある。もしも、Policy に Obligations が規定された場合は、たとえ、ルール結合後の結果が permit であったとしても、Obligations に記述されていることを同時に実施することが出来なかった場合、承認を拒否する必要がある。また Policy が複数の Rule を評価するときは、ルール結合アルゴリズム (表 3.8) を用いる。

表 3.8: XACML のルール結合アルゴリズム

Deny-overrides	どれか 1 つのルールの Effect が Deny であれば結果は Deny とする
Permit-overrides	どれか 1 つのルールの Effect が Permit であれば結果は Permit とする
First-applicable	ポリシー内のすべてのルールについて順番に評価して、対象がマッチした場合、対象の評価結果を Match とし、次に Condition を評価し、これが True なら結果は Effect で指定された Permit または Deny とする。

XACML の評価が終わり、最終的に Permit だった場合その申請は許可される。

3.7 Jungle 上の許認可

Jungle 上で maTriX の許認可を行う際は、ポリシーファイルを読み込み、各 Tree に対して検索関数を用いて、その結果から許認可を行う。

maTriX と同じ例題で、A さんが、学科のノート PC の借りるために貸出申請を行った際の許認可の流れを以下に記す。

1. 人物 Tree から、A さんのデータを取得し、A さんに割り振られている役割の ID を取得する。
2. 役割 Tree から、1 で取得した役割 ID のデータを取得し、その役割 ID に割り振られている役割記述要素 ID を取得する

3. 役割記述要素 Tree から、2 で取得した役割記述要素 ID のデータを取得し、役割記述要素名を取得する
4. 3 で取得した役割記述要素名が、琉球大学工学部学生かどうかを調べ、一致した場合は貸出を許可し、一致しなかった場合は貸出を許可しない。

といった流れになる。Jungle が提供しているのはあくまで組織構造であるため、アクセス可能な時間等の条件に関して Jungle は一切関与しない。

第4章 Jungleに新しく追加したAPI

前章では maTrix のデータ構造と許認可について説明を行った後、Jungle 上にどのように構築するか記述した。本章では、Jungle 上に maTrix を構築するために新しく追加した機能について解説する。

4.1 検索 API の実装

Jungle の Tree に対する検索は、java8 の新機能である lambda 式を用いて find 関数を実装した。lambda 式を使用することで、匿名クラスを使う時より簡潔にコードを記述できるようになった。

find 関数の定義

```
public Iterator<TreeNode>
    find(final Query query, final String key, String searchValue);
```

find 関数は引数に Query、String key、String value の 3 つの引数を取り、条件に一致した Node の Iterator を返す。第一引数には、探索の条件を記述する関数 boolean condition(TreeNode) を定義した InterfaceQuery を。第二、第三引数の、String key、String value は Index の取得を行うために使用する。

QueryInterface の定義

```
public interface Query {
    boolean condition(TreeNode _node);
}
```

以下に実際に lambda 式を用いた find 関数の使用例と実際の処理の流れを記す。

Jungle の Query 部分のソースコード

```
Iterator<TreeNode> pairPersonIterator =
traverser.find((TreeNode node) -> {
    String element = node.getAttributes().getString("element");
    if (element == null)
        return false;
    if (element.equals("Person"))
        return true;
    return false;
}, "element", "Person");
```

1. find 関数の処理の流れは、まず、第 2、第 3 引数の String key,String value を用いて、これらの値に対応した Index が登録されているかを調べる、Index がある場合は Index を使用し値を返す。Index がない場合は、Tree から深さ優先で TreeNode を取得していく。(図 4.1)
2. TreeNode を取得したら、boolean Query.condition(TreeNode) を実行する。この condition は実際に find を使用する際に lambda 式で記述する。
3. condition の中では、TreeNode の Attribute に対して get を行い探索対象のデータを TreeNode が保持しているかを調べる
4. データを持っていた場合は True を、持っていなかった場合は False を返す
5. condition の戻り値が、True だった場合取得した TreeNode を返す。False だった場合は次の TreeNode を取得し、2 に戻る。

find 関数は、上記の処理を探索が終わるまで繰り返す。

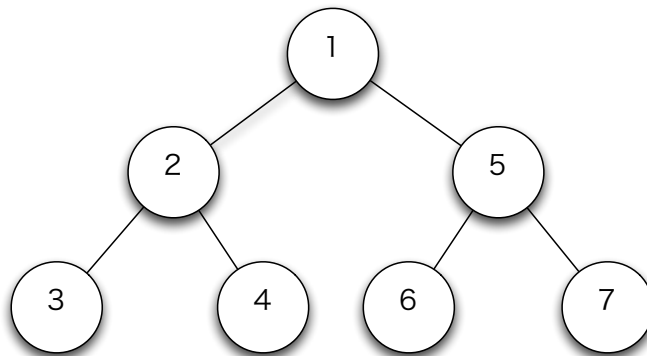


図 4.1: find の検索順序

検索 API は、他に特定の Node 以下に対して検索を行う findInSubTree(Query,node,key,value) も実装した。(図 4.2)

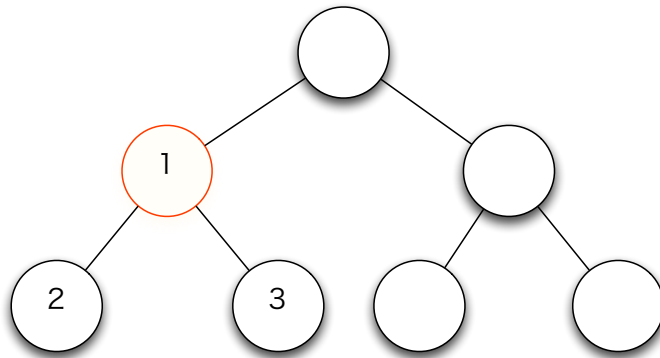


図 4.2: FindInSubTree の検索順序

findInSubTree を使用することで、更に限定的な探索が行えるようになった。この関数も基本的な処理の流れは find 関数と同じである。また、maTrix 上に実装されていた、構成情報からデータを取得する Function も全て実装して、実際の maTrix と同じようにデータのアクセスを行えるようにした。

4.2 Jungle 上の Index の設計

Jungle の探索は Tree を全探索するので、探索の計算量は $O(n)$ である。そこで、Index を使用することで効率よく探索を行えるようにする。Jungle は過去の Tree を全て保持しているため、Tree の version 毎に Index を持っていることが望ましい。そこで、メモリの消費量を抑え、各 version の Tree に Index をもたせる方法として、FunctionalJava の TreeMap を使用した Index の実装を提案する。

TreeMap は、Key と Value のペアを用いて赤黒木を構築する。赤黒木の長所として、ソート済み二分木の探索なので計算量が $O(\log N)$ であることがあげられる。

それに加え、FunctionalJava の TreeMap は、データの更新が行われた際に、一度作られた Tree に対して更新を行わず過去の Tree を再利用し、更新後の TreeMap 新しく返すため、メモリの使用量を抑えつつ複数の version で Index を保持できる。

Index の型を以下に記す。

Index の型

```
TreeMap<String key, TreeMap<String value, List<TreeNode>>>
```

最初の `TreeMap<String key, TreeMap>` は Index を格納する TreeMap である。この TreeMap に対し key で get を行うと、key に対応する Index が登録されている場合、Index を取得でき、取得した Index に対し value で get を行うと、value の値を持つ Node の List が返ってくる。

ParentIndex

TreeNode で get を行うと、親 Node を返す ParentIndex を実装した。ParentIndex の型は、TreeMap<TreeNode,TreeNode> で定義されている。

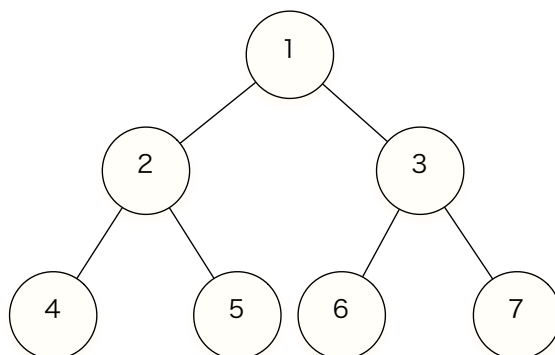


図 4.3: ParentIndexExample

上記の図 4.3 を用いた ParentIndex の例を表 4.1 にまとめた。

表 4.1: ParentIndex の返り値

返ってくる Node	渡す Node
Node2	Node4、 Node5
Node3	Node6、 Node7
Node1	Node2、 Node3

Jungle はノードの編集に、そのノードまでの NodePath を必要としたため、ParentIndex を実装するまでは、Index の型を、TreeMap<String key, TreeMap<String value, List<Pair<TreeNode,NodePath>>>> と定義し、返り値を Pair<TreeNode,NodePath> とし、TreeNode とその TreeNode への NodePath のペアにすることで Node の編集を可能にしていた。しかし、Tree の編集を行った際に、TreeNode への Path は常に変動する為、Index 内の NodePath の更新コストが編集時のネックであった。しかし、ParentIndex を実装することで、TreeNode から NodePath を取得できるようになったため、Index に NodePath を入れる必要がなくなり、Index は今の型で定義できるようになった。ParentIndex も Index と同じように FunctionalJava の TreeMap を使用しているため、メモリの消費を抑えて、全ての version の Tree で保持することが可能である。

以下に ParentIndex を使用して、NodePath を取得する方法を示す。

1. ParentIndex を用いて親 Node を取得する。
2. 親 Node から子ノードのリストを取得する。

3. 子ノードのリストから、親 Node を get するのに使用した Node がある位置のインデックスを取得する。
4. 3 で取得した Index を Path に追加する
5. rootNode に辿り着くまで 1 - 4 を繰り返すと、対象 Node への NodePath が取得できる。

4.3 過去の Tree に対するアクセス

Jungle 上で maTriX の構成情報モデルの表現を行う際に、過去の Tree にアクセスする必要がある。Jungle は、クラス ChangeSet 内に Tree のデータを保持している。ChangeSet からは、今の revisionId と、1 つ前の Tree のデータを取得できるため、再帰的に過去の Tree のデータを取得できる。アクセスしたい Tree の revisionId を引数に取り、過去の Tree のデータの取得と revisionId の比較を繰り返すことで過去の Tree にアクセスする、getOldTree(long revisionId) を DefaultJungleTree 内に実装した。

以下に getOldTree の実装部分のコードを示す。

getOldTree の実装部分

```
@Override
public Either<Error, JungleTree> getOldTree(long revision) {
    TreeContext tc = repository.get();
    ChangeSet cs = tc.getChangeSet();

    for (; cs.revision() != revision;) {
        cs = cs.prev();
        if (cs == null)
            return DefaultEither.newA(GetOldTreeError.OLD_TREE_NOT_FOUND);
    }

    TreeNode root = cs.getRoot();

    TreeContext oldTc = new DefaultTreeContext(root, cs);
    String oldTreeUuid = uuid + revision;
    JungleTree oldTree =
        new DefaultJungleTree(oldTc, oldTreeUuid, writer, treeEditor);
    return DefaultEither.newB(oldTree);
}
```

for 文の中で、過去の Tree の取得と、revision の比較を繰り返している。revision が一致した場合は、その version の Tree を返し、アクセスしたい revision を持つ Tree が見つからなかった場合は、エラーを返す。

4.4 JungleXMLReader

maTrix から XML 形式で書き出されたデータを Jungle に格納するための API として、XMLReader を実装した。JungleXMLReader の実装には sax(Simple API for XML) を用いた。

sax は、XML のタグやテキストデータを読んだ際にイベントを発生させながら、構文解析を行う。プログラマは、ContentHandler という特殊なイベントリスナを sax の parser に登録すると、発生するイベントを取得できる。イベントには、読み込んだタグの名前やテキストデータが含まれている。

表 4.2: sax の主要なイベント一覧

イベント名	呼び出されるタイミング
startDocument	XML の Parse が始まる時
endDocument	XML の Parse が終わった時
startElement	要素を読み込んだ時
endElement	要素が閉じられた時
Charactor	テキストが読み込まれた時

sax では、org.xml.sax.helpers.DefaultHandler という形で、ContentHandler のデフォルト機能が提供されているため、プログラマはこれを継承することで必要なイベント処理のみを override して記述できるようになっている。

XMLReader で使用している ReadXmlHandler は、startElement、charactor、endElement、endDocument、の4つのイベントを使用しており、XML を読み込む際に、Tree を構築しながら Parse を行う。

startElement が呼ばれた時は、今いる地点の下に新しく Node を作り、Attribute の値を格納する。charactor が呼ばれた時は、今いる Node にテキストデータを格納する。endElement が呼ばれたら、今いる Node の親ノードに移動する。

以上の3つの関数を用いて、XML のデータを Tree に格納していき、endDocument が呼ばれた時に木の Commit を行っている。データの格納は、Jungle 上での maTrix のデータ構造の表現で述べたように、多少 Jungle に適合した形に書き換えてから行う。

4.5 XACMLInterpreter

XMLReader と同じよう sax にを用いて実装している。XACMLInterpreter は、引数に、使用する policyFile 名、どの Resource にアクセスするか、どんな処理を行うか、許認可を求める人の UserID 等を与える。

XACMLInterpreter で Handler が使用しているイベントは、XMLReader と同じで、startElement、charactor、endElement、endDocument の4つを使用している。

startElement では、主に評価関数や、評価関数の引数等を取得する。また評価関数が入れ子になっていることがあるため、functionStack と attributeStack を用いて評価関数と引数を関連付けている。図 (4.4) の例だと、function3 を実行した後、function3 の戻り値と、function2 の引数の 2 つを使って function2 を実行する。

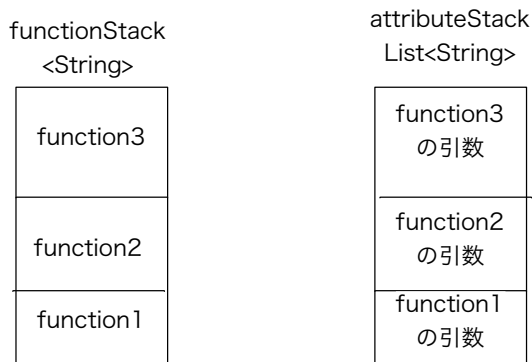


図 4.4: XACMLInterpreter で用いられている stack

character では、評価関数の引数を取得し、AttributeStack に push する endElement では、主に評価関数の実行を行う。endDocument では、これまでに実行した評価関数等の結果から今回の許認可を判断する。

XACMLInterpreter を用いることで Jungle 単体で XACML で記述されたポリシーファイルのテストが行えるようになった。

第5章 検索APIの評価

5.1 測定方法

測定は maTriX が書き出した人物、組織等の XML ファイルを XMLReader で Jungle に読み込み構成情報モデルを構築する。その構成情報モデルに対して、実装したデータに対するアクセス関数の実行時間を測定し、実用的な早さで動作するかを調べる。

5.2 測定環境

プログラムの測定には並列信頼権が所有している firefly を使用した。firefly のスペックを表 5.1 に記す

表 5.1: 実験環境

名前	概要
CPU	24
Memory	16GB
OS	OSX yosemite 10.10.2
JavaVM	1.8.0-25

5.3 検索の API の測定

Jungle に対する検索 API の測定を行う。測定には、maTriX が保持しているデータにアクセスする際に用いる関数のうちの 1 つである、boolean `isActive(String personId, String version)` という関数を用いる。`isActive` は、第一引数で与えた `PersonId` を持つ人物が、第二引数で与えられた構成情報モデルの `version` に対応した人物 `Tree` にいるかどうかを調べる関数である。実験の結果は図 5.1 となる。横軸は人物 `Tree` にいる人の数を表しており、縦軸は探索にかかった時間を表している。

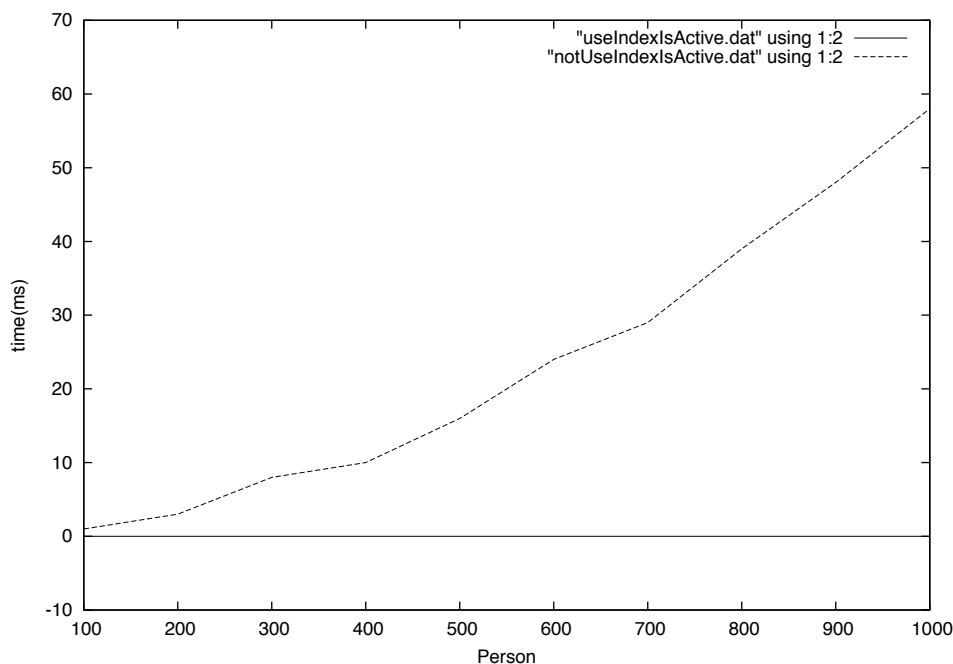


図 5.1: `isActive` の実行時間

`isActive` の処理時間は、Index を使用しない場合 `Person` の数が増えると比例して増えていくのに対し、Index を使用すると `Person` の数に関係なく一定の処理時間だった。

この結果より、Jungle に実装した検索 API は Index を使用することで高速に動作することが確認できた。

第6章 結論

6.1 まとめ

本研究では、初めに当研究室が開発している分散データベース Jungle で使われている非破壊的木構造について述べ、破壊的木構造に比べてロックが少ないというメリットがあること、当研究室で開発している並列分散フレームワーク Alice を用いて分散実装を行っていることを述べた。

次に、実際に Jungle の上に構築する、組織の中の許認可管理を行うアプリケーションである maTrix が、どのようなデータ構造を保持し、申請の許認可を行っているかの説明を行い、Jungle 上で maTrix をどのように実装するかの説明を行った。

maTrix を表現する上で、Jungle に必要な機能がなかったため機能拡張を行った。Java8 の新機能である lambda 式を用いて Tree に対して特定の値の検索を行えるようにし、functionalJava の TreeMap を用いて Index を実装することで検索の高速化に成功した。また、全ての Version の Tree が持っている固有の revisionId を使用し、過去の Tree に対するアクセスも可能にした。

さらに、sax を用いて XMLReader を実装したことで Jungle に XML 形式で記述されたデータを格納できるようになり、maTrix が書き出したデータを取り込めるようになった。また、XACMLInterpreter を実装することでポリシーファイルを読み込み、実際に Jungle を用いて許認可判断を行えるようになり Jungle 上で許認可判断を行えるようになった。

最後に、検索 API の性能評価を行った結果、Jungle の検索は Index を用いることで実用的な速度が出ることを確認した。

6.2 今後の課題

6.2.1 push/pop

Jungle の新しい子の Insert 処理の計算量は木の深さに依存するため、最悪計算量は $O(n)$ となる。しかし、Tree の根の部分に子を追加する push/pop を実装することで Insert の計算量が、非破壊の性質を維持しながら $O(1)$ で行えるようになる

6.2.2 index の IncrementalUpdate

今 Jungle の Index は木の更新が行われる度に新しく作りなおされているため、メモリの消費が大きいしかし新しく作り直さず、木の更新時に差分更新を行うことでメモリの消費を抑えて各 version の Tree に Index を保持することが可能になる。

6.2.3 differencialList

Tree の葉部分に、更新可能な未定義ノードを付加しておくことで、ルートまでのコピーを行わずにノードの追加を行えるようになるので、更新処理が短くなる。

6.2.4 exponential backoff

Jungle は書き込みが競合し、書き込みに失敗した場合すぐに再度書き込みを行うため、書き込みが集中した際失敗を繰り返すことがある。しかし、書き込みが失敗する度に一定時間待機してから再度書き込みを行うことで、競合を避ける事が出来る。

付録

functionalJava のバグについて

Jungle の Index の実装に使用した FunctionalJava の TreeMap.get の計算量が $O(n)$ という致命的なバグがあった。そのため本研究では FunctionalJava のデバッグを行い TreeMap.get の修正を行った。以下に修正前と修正後の測定結果のグラフを記す。

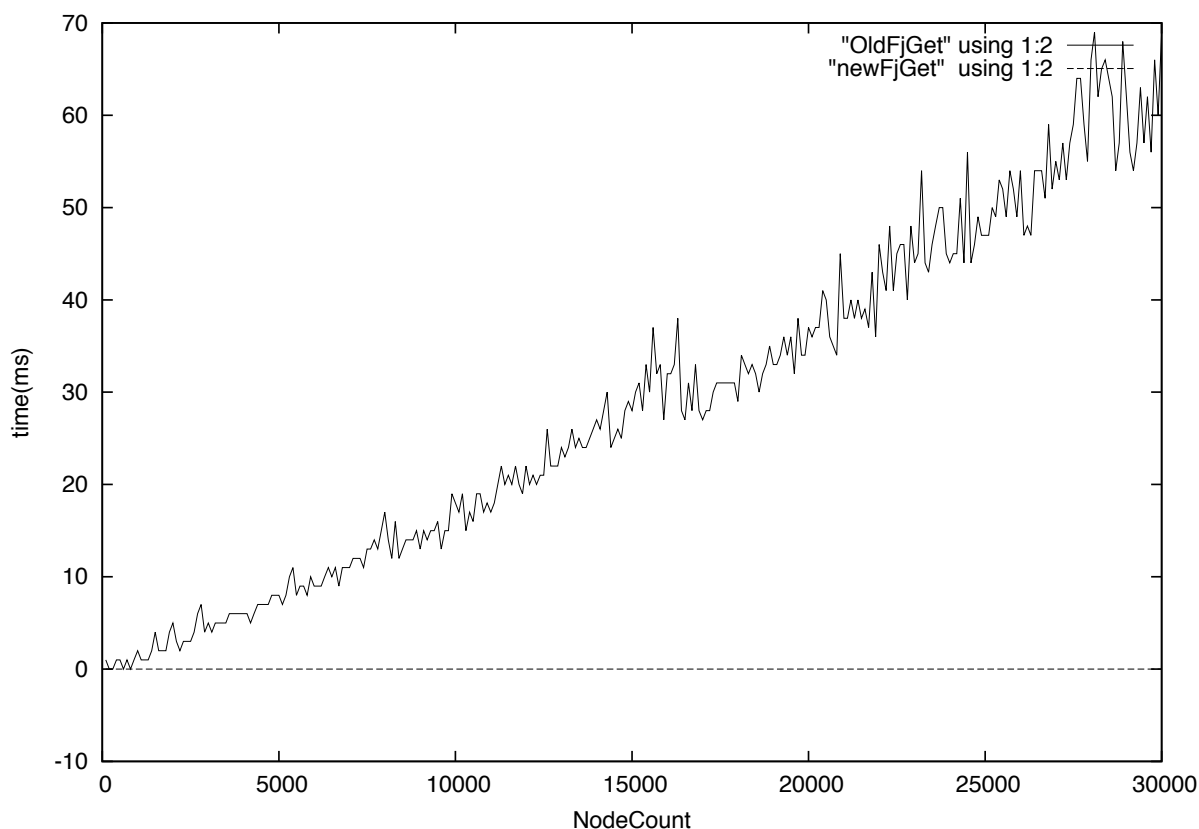


図 6.1: TreeMap の Get

修正前の TreeMap.get は、attribute を探す際に Tree を再構築しながら探索を行っていたため、その部分がネックになっていた。なので、Tree の再構築を行わず探索するよう修正を行った。

maTriX のデータに対するアクセス関数

maTriX がデータにアクセスする際に使用する関数の一覧を以下に記す。

表 6.1: データアクセス関数

isActive	引数に与えられた人に対応するアカウントが有効であるかを調べる
rolePriority	引数に与えられた人が特定の役割を持つ場合に、役割の優先順位を返す
competentRoleId	引数に与えられた人に割り当てられている主務を返す
concurrentRoleIds	引数に与えられた人に割り当てられている兼務をすべて返す
roleDescriptionElementIds	引数に与えられた人、組織の役割記述要素をすべて返す
roleIds	引数に与えられた人、または組織に割り当てられている役割を全て返す
childOrganizationIds	引数に与えられた組織の子組織をすべて返す
deepChildOrganizationIds	引数に与えられた組織の小組織を再帰的に辿って全て返す
personIds	引数に与えられた組織に所属する全ての人を返す
deepPersonIds	引数に与えられた組織を再帰的に深く辿りながらそれらに所属する全ての人を返す
parentOrganizationIds	引数に与えられた組織または人が所属する親組織を返す
deepParentOrganizationIds	引数に与えられた組織または人が所属する親組織を再帰的に全て返す
deepChildOrganizationIdsPlus	deepChildOrganizationIds と基本的に同じ処理をするが、自身も処理対象に含める
deepParentOrganizationIdsPlus	deepParentOrganizationIdsPlus と基本的に同じ処理をするが自身も処理対象に含める
stringPathToId	パスで特定できる要素の Id を返す。 パスで指定可能な要素は役割と役割記述要素の 2 つがある

上記の関数の Jungle 版を本研究では実装した。

参考文献

- [1] 玉城将士 非破壊的木構造を用いた分散 CMS の設計と実装
- [2] 大城信康 分散 Database Jungle に関する研究
- [3] 玉城将士、河野真治. Cassandra を使ったスケーラビリティのある cms の設計. 情報処理学会, March 2011
- [4] Eric Redmond and Jim R. Wilson 7つのデータベース7つの世界
- [5] The MongoDB 2.6 Manual <http://docs.mongodb.org/manual/>
- [6] XACML References and Products
<http://docs.oasis-open.org/xacml/xacmlRefs.html>

謝辞

本研究の遂行，また本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜りました河野真治助教授に深く感謝いたします。

また、様々な研究や勉強の機会を与えてくださった、株式会社Symphonyの永山辰巳さん、同じく様々な助言を頂いた合木茂さん、株式会社PCIホールディングスの皆さまに深く感謝いたします。様々な研究に関わることで自身の研究にも役立てることが出来ました。

研究を行うにあたり、数々の貴重な御助言と細かな御配慮を戴いた、大城信康さん、杉本優さん、並びに並列信頼研究室の全てのメンバーに深く感謝致します。

最後に、有意義な時間を共に過ごした情報工学科の学友、並びに物心両面で支えてくれた両親に深く感謝致します。

2015年3月
金川竜己