Jena入門書

Java アプリケーション内のRDFモデルをJena Semantic Web Frameworkで活用

Philip McCarthy
開発者
SmartStream Technologies Ltd

2004年 6月 23日

半構造化データ(semi-structured data)の表記そして処理に適しているとして、RDFはますます注目を集めています。この記事では、Javaアプリケーションの中のRDFデータ・モデルを活用するためにJena Semantic Web Toolkitを使用する方法をWeb開発者Philip McCarthyは紹介します。

最近ではResource Description Framework (RDF) はW3Cに推薦され、XMLとSOAPのような他のWeb標準と肩を並べられるようになりました。特別な受信データに対処するCRMのような分野に応用され、LiveJournal と TypePadのようなソーシャル・ネットワーキング(social networking) そして自費出版ソフトウェアにて既に広く採用されています。

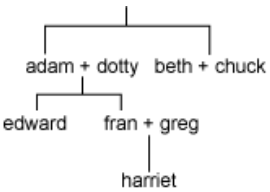
RDF モデルを活用する能力を所持することから、Javaプログラマーはこれからもますます多くのメリットを得ることができます。この記事では、HP Labのオープン・ソースJena Semantic Web Frameworkの機能の一部を紹介します（参考文献を参照してください）。どのようにしてRDFモデルを作成して取り込み、どのようにしてそれらをデータベースに存続させるか、そしてRDQL（照会言語）を使用してプログラム上で照会するかをここで学びます。最後に、ontology(オントロジー)の観点からモデルの知識を推測するのに、Jenaの推理力をどのようにして活用するかをデモンストレーションします。

この記事では、Javaプログラミングの基礎知識のみならず、読者が（グラフ(graphs)、トリプル(triples)、そしてスキーマ(schemas)の概念を基にする）RDFにも慣れ親しんでいると言う前提で考察は進みます。

単純なRDFモデルを作成

まずは基礎から始めましょう。モデルをゼロから作り上げ、それにRDFの記述を追加します。この章では、図1に示されるとおり、架空の親族関係を記述するモデルをどのようにして作成するかを披露します。

図1. 架空の家系図



「Relationship」の語彙（参考文献を参照）から引用される`siblingOf`（兄弟、姉妹）、`spouseOf`（配偶者）、`parentOf`（親）、そして`childOf`（子供）のプロパティーで、それぞれ異なる関係の種類を記述できます。単純な方法として、作られる名前空間（`http://family/`）からのURIで、家族を構成するメンバーを識別できます。語彙のURIはJenaコードにて頻繁に活用されるので、（綴り間違いの減少のためにも）それらをJava定数として宣言するのが良いでしょう。

Schemagen

JenaのAPIを使用するモデルに関わるのであれば、モデルの語彙でのそれぞれのプロパティーの定数を定義するのが便利です。RDFS、DAML、またはOWLの語彙表記が可能であれば、JenaのSchemagen ツールはこれらの定数を自動的に生成することにより作業そのものを楽にしてくれます。

出力を実行するために、Schemagen はコマンド行で実行され、クラスのJavaパッケージ、そして（スキーマまたはontologyファイルの）ロケーションと名前を含むパラメーターを受け取ります。生成されたJavaクラスはインポートされ、モデルにアクセスするためにそのProperty定数を使えます。

進化し続ける語彙にJavaの定数クラスが追い付くように、Antと共に構築プロセスの一部としてSchemagen を実行させるのも可能です。

Jenaの`ModelFactory` クラスは、異なるタイプのモデルを作成するにはより好ましい方法です。この場合、メモリ内の空モデルを求めていますので、`ModelFactory.createDefaultModel()`メソッドを呼び出すべきです。このメソッドは、家系内のそれぞれのメンバーを代表する`Resource` を作成するのに使用する`Model`のインスタンスを戻します。リソースが作成されれば、ステートメントは出来あがりモデルを追加できます。

Jenaにおいては、ステートメントの主部は常に`Resource`、`Property`が述部を表記し、そして目的語はまた別の`Resource` かリテラル値です。Jena では述部は`Literal`タイプで表記されます。これら全ては共通のインターフェース（`RDFNode`）を共有します。この家系図内の関係を表記するには、4つの異なる`Property` インスタンスを必要とします。これらのインスタンスは`Model.createProperty()`を活用し作成されます。

モデルにステートメントを追加する最も簡単な方法は、`Resource.addProperty()`の呼び出しです。`Resource` をその主部とするステートメントを、このメソッドはモデル内で作成します。メソッドは2つのパラメーター（ステートメントの述部を表記する`Property`、そしてステートメントの目的語）を含みます。`addProperty()`メソッドはオーバーロードです。1つのオーバーロードは`RDFNode` を目的語として受け取るので、`Resource` または`Literal` を使用できます。Java プリミティブまたは`String`に表記されるリテラルを受け取る便利なオーバーロードもあります。この例では、ステートメントの目的語は、他の家系のメンバーを表記する`Resources`です。

トリプル(Triple)である主部、述部、目的語と共に`Model.createStatement()` を呼び出すことにより、モデル上にてステートメントを直接作成可能です。`Statement` をこの方法で作成しても、モデルにStatementを追加しません。Statementをモデルに追加したいのであれば、作成された`Statement`で`Model.add()`を呼び出しましょう (リスト1参照) 。

リスト1. 架空の家系を表記するためにモデルを作成

```
// URI declarations
String familyUri = "http://family/";
String relationshipUri = "http://purl.org/vocab/relationship/";
// Create an empty Model
Model model = ModelFactory.createDefaultModel();
// Create a Resource for each family member, identified by their URI
Resource adam = model.createResource(familyUri+"adam");
Resource beth = model.createResource(familyUri+"beth");
Resource chuck = model.createResource(familyUri+"chuck");
Resource dotty = model.createResource(familyUri+"dotty");
// and so on for other family members
// Create properties for the different types of relationship to represent
Property childOf = model.createProperty(relationshipUri,"childOf");
Property parentOf = model.createProperty(relationshipUri,"parentOf");
Property siblingOf = model.createProperty(relationshipUri,"siblingOf");
Property spouseOf = model.createProperty(relationshipUri,"spouseOf");
// Add properties to adam describing relationships to other family members
adam.addProperty(siblingOf,beth);
adam.addProperty(spouseOf,dotty);
adam.addProperty(parentOf,edward);
// Can also create statements directly . . .
Statement statement = model.createStatement(adam,parentOf,fran);
// but remember to add the created statement to the model
model.add(statement);
```

完全コードの例であるFamilyModel.javaは、ステートメントの一群がどのようにしてモデルに同時に (配列または`java.util.List`として) 追加されるのかをデモンストレーションします。

家系図モデルが完成しましたので、どうすればJenaの照会APIを使用して情報を摘出できるかを観察してください。

RDFモデルを問い合わせる

プログラム上でJenaモデルを問い合わせる行為は、主に`Model` と`Resource` インターフェース上のlist() メソッドを通して実行されます。とある状態と一致する`Statements`、目的語、そして主部を得るためにこれらのメソッドを使用します。それらは (特定の目的語・タイプを戻すおまけのメソッドを持つ) `java.util.Iterator` のスペシャライゼイション(specialisations)を戻したりもします。

リスト1の家系モデルの例に焦点を戻し、リスト2にて示されるような別の方法での問い合わせに注目しましょう。

リスト2. 家系モデルの照会

```
// List everyone in the model who has a child:
ResIterator parents = model.listSubjectsWithProperty(parentOf);
// Because subjects of statements are Resources, the method returned a ResIterator
while (parents.hasNext()) {
  // ResIterator has a typed nextResource() method
  Resource person = parents.nextResource();
  // Print the URI of the resource
  System.out.println(person.getURI());
}
// Can also find all the parents by getting the objects of all "childOf" statements
// Objects of statements could be Resources or literals, so the Iterator returned
// contains RDFNodes
NodeIterator moreParents = model.listObjectsOfProperty(childOf);
// To find all the siblings of a specific person, the model itself can be queried
NodeIterator siblings = model.listObjectsOfProperty(edward, siblingOf);

// But it's more elegant to ask the Resource directly
// This method yields an iterator over Statements
StmtIterator moreSiblings = edward.listProperties(siblingOf);
```

ここで紹介されているコンビニエンス・メソッドの根源にある最も一般的な照会メソッドは`Model.listStatements(Resource ##, Property ##, RDFNode ###)`です。これらのパラメーターのどれもがnullとして設定可能であり、（その場合には）ワイルドカードとしての機能を果たして何にでも対処が可能となります。`Model.listStatements()`の使用例はリスト3にて示されています。

リスト3. モデル照会にSelectorsを使用

```
// Find the exact statement "adam is a spouse of dotty"
model.listStatements(adam,spouseOf,dotty);

// Find all statements with adam as the subject and dotty as the object
model.listStatements(adam,null,dotty);

// Find any statements made about adam
model.listStatements(adam,null,null);
// Find any statement with the siblingOf property
model.listStatements(null,siblingOf,null);
```

インポートし永続するモデル

全てのアプリケーションが空のモデルから始まるわけではありません。より一般的には、モデルは開始の時点で現存のデータを取り込みます。この状況でメモリ内のモデルを使用することの欠点は、アプリケーションを起動するたびに一からモデルを再取り込みしなくてはならないことです。さらに、メモリ内のモデルへの変更は、アプリケーションがシャットダウンされるたびに失われます

一つの解決法は、起動時に`Model.write()`でファイル・システムにモデルを直列化させ、`Model.read()`で直列化を解除します。しかしながら、Jenaは（バッキング・ストアに透明かつ継続的に存続された）パーシスタントなモデルを提供します。Jenaはファイル・システムまたはリレーショナル・データベースにてそのモデルを存続できます。現在サポートされるデータベース・エンジンは、PostgreSQL、Oracle、そしてMySQLです。

WordNet

WordNet とは、「英語用の字句データベース」です。ここで採用しているRDF表記は、Sergey Melnik と Stefan Deckerによるものです。それは4つの別個のモデルとして表記され、そのうちの3つをこの記事の用例で使用します。

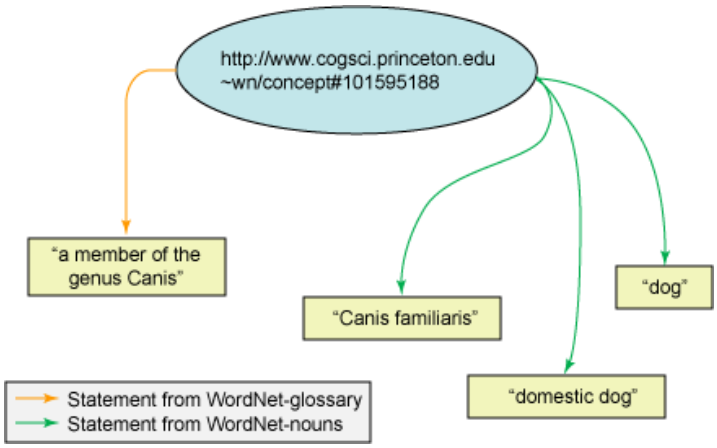
WordNet名詞のモデルは全てのWordNetに表記される「lexical concepts (語彙コンセプト) 」そしてそれぞれを表記する「word form (ワード・フォーム) 」を含みます。例えば、それは「domestic dog」、「dog」、そして「Canis familiaris」と言うワード・フォームが表記する語彙コンセプトを含みます。

第2のモデルはWordNet用語集です。それはモデル内にあるそれぞれの語彙コンセプトの簡単な定義を提供します。「dog」の語彙コンセプトには、「有史以前に人間により飼養化された (普通の狼の子孫と思われる) イヌ属の一種」と記される用語集入力があります。

WordNet下位語は第3のモデルです。それはモデル内にある概念の階層を定義します。「dog (イヌ) 」のコンセプトは「canine (イヌ科) 」のコンセプトの下位語であり、それは「carnivore (肉食動物) 」のコンセプトの下位語です。

どのようにしてモデルをインポートそして存続させるかをデモンストレーションするために、MySQL にWordNet 1.6 データベースのRDF表記をインポートします。ここで使用するWordNet表記が複数の別個のRDF ドキュメントとして存在するため、1つのJenaモデルにそれらをインポートする行為はステートメントを組み合わせます。(Nouns とGlossary のモデルを組み合わせた後の) WordNet モデルのフラグメントの構造を、図2でデモンストレーションします。

図2. 組み合わせられたWordNetの名詞モデルと用語集モデルの構造



データベースを処理するモデルを作成する第一歩は、MySQL ドライバー・クラスをインスタンス化して`DBConnection` インスタンスを作成することです。`DBConnection` コンストラクターは、ユーザーがデータベースにログインする時に使用するIDとパスワードを取得します。Jena が「`jdbc:mysql://localhost/dbname`」として使用する、MySQL データベースの名前を含むデータベースURLパラメーターをも取得します。Jena は1つのデータベースで複数のモデルを作成できます。最後の`DBConnection` パラメーターはデータベース・タイプであり、MySQLの場合それは「`MySQL`」です。

`DBConnection` インスタンスをJenaの`ModelFactory<` と共に使用し、データベースを処理するモデルを作成できます。

一度モデルが作成されれば、WordNet RDF ドキュメントをファイル・システムから読み込めます。様々な種類の`Model.read()`メソッドが`Reader`、`InputStream`、または`URL`からモデルを取り込むことができます。モデルをNotation3、N-Triples、または (デフォルトとして) RDF/XML 構文から構文解析できます。WordNet モデルはRDF/XMLとしてシリアライズ化されていますので、構文を指定する必要はありません。モデルを読み込むとき、ベースURIは供給されます。ベースURIを使用して、モデル内のどの相対URIをも絶対URIに変換できます。WordNetは相対URIを含まないため、このパラメーターは`null`に設定できます。

MySQL パーシスタントモデル(MySQL-persisted model)にWordNetのRDF/XMLファイルをインポートする完全なプロセスをリスト4に示します。

リスト4. WordNet モデルをインポートそして永続

```
// Instantiate the MySQL driver
Class.forName("com.mysql.jdbc.Driver");
// Create a database connection object
DBConnection connection = new DBConnection(DB_URL, DB_USER, DB_PASSWORD, DB_TYPE);

// Get a ModelMaker for database-backed models
ModelMaker maker = ModelFactory.createModelRDBMaker(connection);
// Create a new model named "wordnet." Setting the second parameter to "true" causes an
// AlreadyExistsException to be thrown if the db already has a model with this name
Model wordnetModel = maker.createModel("wordnet",true);
// Start a database transaction. Without one, each statement will be auto-committed
// as it is added, which slows down the model import significantly.
model.begin();
// For each wordnet model . . .
InputStream in = this.getClass().getClassLoader().getResourceAsStream(filename);
model.read(in,null);
// Commit the database transaction
model.commit();
```

wordnet モデルを取り込みましたので、`ModelMaker.openModel("wordnet",true);`への呼び出しで後にアクセス可能となります。

実行されるそれぞれの種類の照会は数行ものあつらえのコード書き出しを必要としますので、Jena APIのみを使用してWordNet 並みに内容が巨大で濃いモデルを照会する行為には限界があります。運の良いことに、一般的な照会を表現するメカニズムをJenaはRDQLと言う形で提供します。

RDF Data Query Language (RDQL)

RDQL はRDFの照会言語です。まだ正式な標準ではありませんが、RDF フレームワークはRDQL を広範囲に渡り実装します。照会エンジンがデータ・モデルへのアクセスと言う大変な役割を担当し、RDQL は複雑な照会の簡潔な表現を可能にします。RDQLの構文は表面上ではSQLの構文を彷彿とさせ、実際にその概念の一部はリレーショナル・データベースの照会に関わったことのある人には親しみを覚えやすいでしょう。JenaのWebサイトで良質のRDQL チュートリアルを見付けられますが、基本を奥深く図解するには幾つかの簡単な用例で十分です。

Jena モデルに対して`jena.rdfquery` ツールを使用しコマンド行でRDQL 照会を実行できます。RDFQuery はRDQL 照会をテキスト・ファイルから取り込み、指定されたモデルに対してそれを実行します。データベースを処理するモデルに対して実行するのに、結構な数のパラメーター

を必要とします。この後に続く幾つかの用例の実行に必要とされる完全なコマンド行を、リスト5に示します。

リスト5. コマンド行からRDQL 照会を実行

```
$java jena.rdfquery --data jdbc:mysql://localhost/jena --user dbuser --password dbpass
--driver com.mysql.jdbc.Driver --dbType MySQL --dbName wordnet --query example_query.rdql
```

ご覧のとおり、MySQLへの接続を作成するために必要な詳細をそれらのパラメーターの大半が提供します。重要な部分は、RDQL ファイルのロケーションである「`query example_query.rdql`」です。Jenaのライブラリー・ディレクトリーにある全てのJARファイルを`jena.rdfquery`実行時に必要とすることに注意してください。

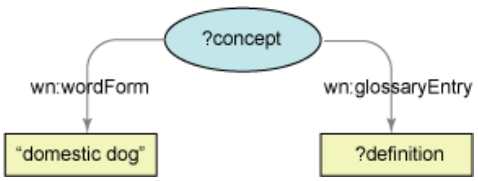
最初に検査する照会をリスト6に示します。

リスト6. 「domestic dog」のWordNet用語集入力を検出するRDQL 照会

```
SELECT
    ?definition
WHERE
    (?concept, <wn:wordForm>, "domestic dog"),
    (?concept, <wn:glossaryEntry>, ?definition)
USING
    wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>
```

照会により出力される変数（この場合、変数は`definition`）を、`SELECT`の部分が宣言します。`WHERE`の文節は第2の変数conceptを導き、グラフに照合するトリプル(triple)を定義します。（`WHERE`の文節にある全てのtripleが効力を有する）グラフにあるステートメントを、照会は検出します。図3に示されるとおり、英語では`WHERE`の文節は「domestic dogと言うwordformを持つコンセプトを探し、それらのコンセプトに対する用語集入力を探す」のようになります。`USING`文節は、名前空間の接頭部を宣言するのに使われるコンビニエンスです。

図3. リスト6のWHERE文節に照合されるグラフ



照会を実行すれば、以下のとおりになります。

```
definition
================================================================================
"a member of the genus Canis (probably descended from the common wolf) that has
been domesticated by man since prehistoric times; occurs in many breeds; "the
dog barked all night""
```

ここで得られる結果は一つだけです。リスト7にて示される次の照会では、「find concepts represented by the word 'bear' and find the glossary entries for those concepts」（bear（熊）と言う単語に表記されるコンセプトを探し、それらのコンセプトに対する用語集入力を探す）と記述されています。

リスト7. 「bear」のWordNet用語集入力を検出するRDQL 照会

```
SELECT
    ?definition
WHERE
    (?concept, <wn:wordForm>, "bear"),
    (?concept, <wn:glossaryEntry>, ?definition)
USING
    wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>
```

このwordform「bear」は複数の別個のコンセプトを表記しますので、この照会は15個の結果を戻します。結果の冒頭は次のとおりで、ご丁寧にも「bear」の同綴異義語の用語も連ねます。

```
definition
================================================================================
"massive plantigrade carnivorous or omnivorous mammals with long shaggy coats
and strong claws"
"an investor with a pessimistic market outlook"
"have on one's person; "He wore a red ribbon"; "bear a scar""
"give birth (to a newborn); "My wife had twins yesterday!""
```

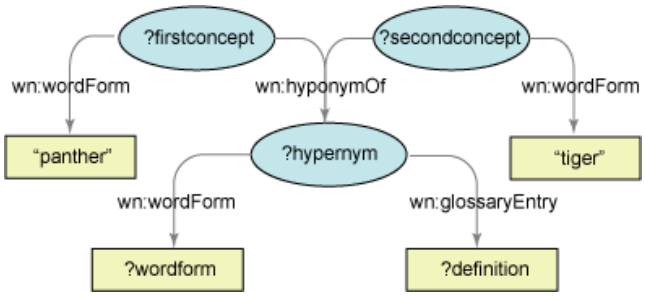
リスト8に示されるより手ごたえのある用例は、2つの別々の単語の上位語を探します。

リスト8. 「panther（ヒョウ）」と「tiger（トラ）」のWordNet 用語集入力を検出するRDQL 照会

```
SELECT
      ?wordform, ?definition
WHERE
      (?firstconcept, <wn:wordForm>, "panther"),
      (?secondconcept, <wn:wordForm>, "tiger"),
      (?firstconcept, <wn:hyponymOf>, ?hypernym),
      (?secondconcept, <wn:hyponymOf>, ?hypernym),
      (?hypernym, <wn:wordForm>, ?wordform),
      (?hypernym, <wn:glossaryEntry>, ?definition)
USING
      wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>
```

図4に示されるとおり、「panther（ヒョウ）とtiger（トラ）の単語に参照されるコンセプトを探し、最初の2つのコンセプトを下位語として扱う第3のコンセプトを探し、そしてそのコンセプトのでありそうな単語と用語集入力を探す」と照会は述べております。

図4. リスト8のWHERE文節に適合するグラフ



`wordform` と`definition`は両方ともに`SELECT`文節にて宣言されますので、両方とも出力です。この照会は1つのWordNet コンセプトと適合するのみですが、コンセプトには2つの別個のwordformsがありますので照会のグラフは2つの方法で照合されます。

```
wordform  | definition
================================================================================
"big cat" | "any of several large cats typically able to roar and living in the wild"
"cat"     | "any of several large cats typically able to roar and living in the wild"
```

JenaにてRDQL を使用

Jenaのパッケージ`com.hp.hpl.jena.rdql`はJavaコードでRDQLを使用するうえで必要な全てのクラスとインターフェースを含みます。RDQL 照会を作成するには、`String`にRDQL を入れ、`Query`のコンストラクターに受け渡します。RDQL そのものにて`FROM`文節に特別の規定がなければ、(照会へのソースとして使用するように) モデルを明確に設定するのが普通です。`Query` が一度準備されれば、`QueryEngine` をそれから作成し照会を実行できます。リスト9はこのプロセスのデモンストレーションです。

リスト9. RDQL照会を作成そして実行

```
// Create a new query passing a String containing the RDQL to execute
Query query = new Query(queryString);
// Set the model to run the query against
query.setSource(model);

// Use the query to create a query engine
QueryEngine qe = new QueryEngine(query);
// Use the query engine to execute the query
QueryResults results = qe.exec();
```

`Query` と一緒に使う役に立つテクニックは、実行する前に一部の変数を固定された値に設定することです。この使用パターンは`javax.sql.PreparedStatement`のそれに類似します。実行時に`QueryEngine` に受け渡される`ResultBinding` 目的語を介し、任意の値に変数は設定されます。Jena`Resources` またはリテラル値に変数を設定することが可能です。それを変数に設定する前に、リテラルを`Model.createLiteral` への呼び出しでラッピングしてください。この事前に設定するアプローチをリスト10に示します。

リスト10. 照会変数を任意の値に設定

```
// Create a query that has variables x and y
Query query = new Query(queryString);
// A ResultBinding specifies mappings between query variables and values
ResultBinding initialBinding = new ResultBinding() ;
// Bind the query's first variable to a resource
Resource someResource = getSomeResource();
initialBinding.add("x", someResource);
// Bind the query's second variable to a literal value
RDFNode foo = model.createLiteral("bar");
initialBinding.add("y", foo);
// Execute the query with the specified values for x and y
QueryEngine qe = new QueryEngine(query);
QueryResults results = qe.exec(initialBinding);
```

`QueryEngine.exec()`に戻される`QueryResults` 目的語は、`java.util.Iterator`を実装します。その`next()` メソッドは`ResultBinding` 目的語を戻します。`SELECT` 文節の一部かどうかに関係無く、照会にて使われる全ての変数を`ResultBinding` から名前を使って取得できます。ここでもリスト6のRDQL照会を使い、どのようにそれを実行するかをリスト11にて示します。

リスト11. 「domestic dog」のWordNet用語集入力を検出するRDQL照会

```
SELECT ?definition
WHERE (?concept, <wn:wordForm>, "domestic dog"), (?concept, <wn:glossaryEntry>, ?definition)
USING wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>";
```

予想通り、この照会の実行から取得される`ResultBinding`はその単語のリテラル用語集入力を含みます。さらに、変数`concept`にアクセスが可能です。`ResultBinding.get()`を呼び出すことにより変数を名前で取得できます。このメソッドで戻される全ての変数を（さらなるRDQL 照会にその変数を設定したい場合に便利な）`RDFNode`にキャストできます。

この場合、変数`concept`はRDFのリソースを表記しますので、`ResultBinding.get()` から得た`Object`を`Resource`にキャストできます。リスト12に示されるとおり、モデルのその部分をさらに探求するのに`Resource`の照会メソッドを呼び出すことができます。

リスト12. 照会の結果への働きかけ

```
// Execute a query
QueryResults results = qe.exec();
// Loop over the results
while (results.hasNext()) {
  ResultBinding binding = (ResultBinding)results.next();
  // Print the literal value of the "definition" variable
  RDFNode definition = (RDFNode) binding.get("definition");
  System.out.println(definition.toString());
  // Get the RDF resource used in the query
  Resource concept = (Resource)binding.get("concept");
  // Query the concept directly to find other wordforms it has
  List wordforms = concept.listObjectsOfProperty(wordForm);
}
```

ソースのダウンロード（参考文献を参照）に含まれるFindHypernym.java と言う名のプログラムはここで考察された領域を要約します。それはリスト13に示される照会を駆使し、コマンド行に与えられた単語の上位語を探します。

リスト13. コンセプトの上位語の用語集入力とwordformsを探すRDQL 照会

```
SELECT ?hypernym, ?definition
WHERE (?firstconcept, <wn:wordForm>, ?hyponym),
    (?firstconcept, <wn:hyponymOf>, ?secondconcept),
    (?secondconcept, <wn:wordForm>, ?hypernym), (?secondconcept, <wn:glossaryEntry>, ?definition)
USING wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>
```

コマンド行に与えられた単語は下位語に設定され、照会はその単語が表記するコンセプトを探し、最初のコンセプトを下位語とする第2のコンセプトを探し、そしてそのコンセプトのwordformと定義を出力します。その出力がどのようなものかを、リスト14にて示します。

リスト14. FindHypernym プログラムの例を実行

```
$ java FindHypernym "wisteria"
Hypernyms found for 'wisteria':
vine:   weak-stemmed plant that derives support from climbing, twining,
or creeping along a surface
```

OWLで意味を追加

なぜ「wisteria（藤）」の上位語を検索しても直ぐ上の上位語である「vine（つる植物）」しか戻さないのか、不思議に思われていることでしょう。植物学にこだわる人でしたら、上位語として「tracheophyte（維管束植物）」そして当然のことながら「plant（植物）」を期待します。WordNet モデルは確かに「wisteria（藤）」が「vine（つる植物）」の下位語だと示し、「vine（つる植物）」が「tracheophyte（維管束植物）」の下位語だと述べます。『下位語』や『上位語』の関係は推移的なので、「wisteria（藤）」が「tracheophyte（維管束植物）」の下位語であると直感的に解ります。FindHypernym プログラムにその知識を盛り込む方法が必要となり、そこでOWLの出番となります。

推移的な関係

3つの要素（a、b、c）の間の関係が推移的であれば、aとcの関係そしてbとcの関係はaとcの関係を浮き彫りにします。

推移的な関係の一例として、「～より大きい (>)」が挙げられます。aがbより大きく（a>b）、bがcよりも大きければ（b>c）、当然のことですがaはcより大きくなくてはなりません（a>c）。

OWL（Web Ontology Language）は「語彙の用語の意味そしてそれらの用語同士の関係を明確に表記すること」を目的とするW3C Recommendationです。RDFスキーマと共に、OWL はRDF モデルの内容を形式的に記述する仕組みを提供します。リソースが属することができる階層的なクラスを定義することに加え、OWL はリソースのプロパティーの特徴を表現することを許可します。例えば、リスト1で使用されたRelationship の語彙では、プロパティー`childOf` がプロパティー`parentOf` の反転であることを述べるのにOWLを使うことが可能です。別の用例では、WordNet の語彙のプロパティー`hyponymOf` が推移的だと述べます。

Jenaにおいては、ontologyはRDFモデルの特殊タイプ（`OntModel`）として扱われます。このインターフェースはプログラム上でontologyを操作することを可能にし、プロパティーの制限やクラスを作成するコンビニエンス・メソッドもあります。別の方法として、ontologyを普通のRDFモデルとして扱い、そのセマンティック・ルールを定義する記述を単に追加します。これらのテクニックは両方ともリスト15にてデモンストレーションされています。現存するデータ・モデルにontology的記述を追加すること、または`Model.union()`を使用してデータ・モデルにontologyモデルを組み合わせることが可能である事実に注目してください。

リスト15. WordNetのOWLontologyモデルを作成

```
// Make a new model to act as an OWL ontology for WordNet
OntModel wnOntology = ModelFactory.createOntologyModel();
// Use OntModel's convenience method to describe
// WordNet's hyponymOf property as transitive
wnOntology.createTransitiveProperty(WordnetVocab.hyponymOf.getURI());
// Alternatively, just add a statement to the underlying model to express that
// hyponymOf is of type TransitiveProperty
wnOntology.add(WordnetVocab.hyponymOf, RDF.type, OWL.TransitiveProperty);
```

Jenaで推論

ontologyとモデルを与えられれば、Jenaの推論エンジンはモデルが明確に表現しない追加の記述を推理できます。様々な種類のontologyに働きかける複数の`Reasoner` タイプをJenaは提供します。WordNet モデルと共にOWL ontologyを使用したいはずですので、`OWLReasoner` が必要です。

次の例は推論モデルを作成するためにWordNetモデルそのものにOWL WordNet ontologyを応用する方法を披露します。ここでは実際に（下位語の階層で言えば「plant life（植物）」より下の名詞しか含まない）WordNet のサブセットを使用します。サブセットしか使わない理由は、推論モデルはメモリにて保留されなくてはならず、メモリ内モデルが実用性を保つにはWordNetが大きすぎるからです。完全なWordNet モデルから植物モデルを抽出するのに使用したコードはExtractPlants.java と名付けられ、記事のソースに含まれています（参考文献を参照してください）。

まず、`ReasonerRegistry`から`OWLReasoner` を取得します。`ReasonerRegistry.getOWLReasoner()` は（この単純な例では十分な）標準構成のOWL reasonerを戻します。次のステップでは、そのreasonerをWordNet ontologyに設定します。この操作はontologyのルールを応用する準備の出来たreasonerを戻します。その次に、設定されたreasonerを使用してWordNetモデルから`InfModel`を作成します。

基のデータそしてOWL ontologyから推論モデルを作成した後では、それは他のどの`Model`インスタンスのごとく扱われます。それゆえに、リスト16で示されるように、FindHypernym.javaに通常のJenaモデルと共に使用されるRDQL照会とJavaコードは、変更無しで推論モデルに再度応用されます。

リスト16. 推論モデルを作成そして照会

```
// Get a reference to the WordNet plants model
ModelMaker maker = ModelFactory.createModelRDBMaker(connection);
Model model = maker.openModel("wordnet-plants",true);
// Create an OWL reasoner
Reasoner owlReasoner = ReasonerRegistry.getOWLReasoner();
// Bind the reasoner to the WordNet ontology model
Reasoner wnReasoner = owlReasoner.bindSchema(wnOntology);
// Use the reasoner to create an inference model
InfModel infModel = ModelFactory.createInfModel(wnReasoner, model);
// Set the inference model as the source of the query
query.setSource(infModel);
// Execute the query as normal
QueryEngine qe = new QueryEngine(query);
QueryResults results = qe.exec(initialBinding);
```

FindInferredHypernyms.javaと名付けられた記事ソースにて、完全なリストを入手できます。推論モデルで「wisteria（藤）」の上位語の照会をするとどうなるかを、リスト17にて示します

リスト17. FindInferredHypernyms プログラムの例を実行

```
$ java FindInferredHypernyms wisteria
Hypernyms found for 'wisteria':
vine:   weak-stemmed plant that derives support from climbing, twining, or creeping along a surface
tracheophyte:   green plant having a vascular system: ferns, gymnosperms, angiosperms
vascular plant: green plant having a vascular system: ferns, gymnosperms, angiosperms
plant life:     a living organism lacking the power of locomotion
flora:  a living organism lacking the power of locomotion
plant:  a living organism lacking the power of locomotion
```

OWL ontologyに含まれる情報のおかげで、「wisteria（藤）」には上位語があることをJenaはモデルを通して推論できます。

まとめ

この記事は、どのようにしてRDFモデルを作成、インポート、そして存続させるかを示す例で、Jena Semantic Web Toolkitで最も重要な機能をいくつか具体的に説明しました。モデルを照会する様々な方法、そして任意の照会を簡潔に表現するのにRDQLをどのように使用するかを観察しました。さらに、Jenaの推論エンジンがどのようにしてontologyを基にしたモデルの推論を行なうかをも説明しました。

この記事の用例は、RDFモデルとしてデータを表記する能力そしてデータをそれらから抽出するRDQLの柔軟性をも具体的に説明しました。Javaアプリケーションの RDFモデルを採用する場合に、ここで説明した基本的なアプローチは役に立つ出発点となります。

Jenaは包括的なRDFツール・セットであり、ここで説明された能力をも遥かに凌駕します。Jenaのさらなる能力を学ぶには、Jena プロジェクトのホームページから入るのが良いでしょう。

著者について

Philip McCarthy

Philip McCarthy は J2EE そしてフロントエンドの技術を専門とする Web Developer (Web 開発者) です。Orange での Consumer Internet Applications での実績を含む 4 年間の Java プログラミングの経験を誇ります。現在、彼は SmartStream Technologies にて金融関連の Web アプリケーションを開発中です。