Chapter 1

Apache Jena

Contributed by Tong Zhao

1.1 Background

Apache Jena [3] is a free and open source Java specifically for building Semantic Web and Linked Data applications. Apache Jena provides an Java API that supports reading, writing and manipulating Resource Description Framework (RDF) graphs, where RDF is a set of World Wide Web Consortium (W3C) specifications [1] designed as a metadata data model.

Jena was created by HP Labs[1] in the year of 2000. In 2009, HP decided to stop direct support of development of Jena, so the project team applied to have Jena adopted by the Apache Software Foundation in November 2010 and the vote result is still publicly available[2].

1.2 Expressing Graphs

In Jena, an RDF graph is represented by a data structure called `Model`. A `Model` contains a collection of RDF resources (vertices), attached to each other by labelled relations (edges). Each relationship goes only in one direction, so all RDF graphs are directed and hence Jena does not support undirected graphs. Each resource can have any number of `VCARD`, which are the properties of that resource. The vCard Ontology [2] is predefined and contains a large number of property types.

1.3 Syntax

Since Jena is a Java API, the syntax of Jena is just like any Java programs. Below is a small piece of code from Jena official documentations, the detailed meanings of the methods will be talked in the next section.

```
// list the statements in the Model
StmtIterator iter = model.listStatements();

// print out the predicate, subject and object of each statement
```

[1]http://www8.hp.com/us/en/hp-labs/index.html

[2]http://mail-archives.apache.org/mod_mbox/incubator-general/201011.mbox/<4CEC31E4.9080401@apache.org>

```
while (iter.hasNext()) {
    Statement stmt      = iter.nextStatement();  // get next statement
    Resource  subject   = stmt.getSubject();     // get the subject
    Property  predicate = stmt.getPredicate();   // get the predicate
    RDFNode   object    = stmt.getObject();      // get the object

    System.out.print(subject.toString());
    System.out.print(" " + predicate.toString() + " ");
    if (object instanceof Resource) {
       System.out.print(object.toString());
    } else {
        // object is a literal
        System.out.print(" \"" + object.toString() + "\"");
    }

    System.out.println(" .");
}
```

1.4 Key Graph Primitives

- `Model` is the class that represents a graph in Jena API. To create a model, one can use a predefined model type in `ModelFactory` as:
  `Model model = ModelFactory.createDefaultModel();`

- `Resource` is the class that represents a vertec in Jena API. Each resource usually have a URI as its unique id, so a resouce can be created as:
  `static String personURI = "http://somewhere/JohnSmith";`
  `Resource johnSmith = model.createResource(personURI);`

- `.addProperty()` is the method that can be used to add a property or a statement (edge) to a resourse. Taking the above resource `johnSmith` as an example, to add a property to it:
  `static String fullName = "John Smith";`
  `johnSmith.addProperty(VCARD.FN, fullName);`
  To make a statement between `johnSmith` with a new resource `davidSmith`:
  `Resource davidSmith = model.createResource("http://somewhere/davidSmith");`
  `johnSmith.addProperty(hasSon, davidSmith);`

- `.listStatements()` is the method to get a iterator which contains all the statements within the model. The example in Section 1.3 is using it to print out all the statements within a given model.

- `SimpleSelector` is a class that can be used to make basic queries on the model. Following is a basic example of it:

```
// select all the resources with a VCARD.FN property
// whose value ends with "Smith"
StmtIterator iter = model.listStatements(
    new SimpleSelector(null, VCARD.FN, (RDFNode) null) {
        public boolean selects(Statement s)
            {return s.getString().endsWith("Smith");}
    });
```

- `JenaARQTest`, `QueryFactory` and `QueryExecutionFactory` are the classes that can be together used to execute any SPARQL queries on RDF graphs within the Java API, where SPARQL is a query language that was specifically designed for RDF databases.

1.5 Execution Model

I did not find any description of Jena's execution model.

1.6 Examples

Below is a basic BFS implementation[3] by Jena API in Java.

```
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.Resource;
import com.hp.hpl.jena.rdf.model.StmtIterator;
import com.hp.hpl.jena.vocabulary.OWL;
import com.hp.hpl.jena.vocabulary.RDFS;

public class BFSInRDFWithJena {

    public static List<List<Resource>> BFS( final Model model,
                final Queue<List<Resource>> queue, final int depth ) {
        final List<List<Resource>> results = new ArrayList<>();
        while ( !queue.isEmpty() ) {
            final List<Resource> path = queue.poll();
            results.add( path );
            if ( path.size() < depth ) {
                final Resource last = path.get( path.size() - 1 );
                final StmtIterator stmt = model.listStatements(null,
                                        RDFS.subClassOf, last);
                while ( stmt.hasNext() ) {
                    final List<Resource> extPath = new ArrayList<>( path );
                    extPath.add( stmt.next().getSubject().asResource() );
                    queue.offer( extPath );
                }
            }
        }
```

[3]https://stackoverflow.com/questions/17750421/retrieving-all-paths-in-an-owl-class-hierarchy-with-sparql-and-jena

```
        return results;
    }

    public static void main( final String[] args ) throws IOException {
        final Model model = ModelFactory.createDefaultModel();
        try (final InputStream in = BFSInRDFWithJena.class.getClassLoader().
                                        getResourceAsStream("camera.owl")) {
            model.read( in, null );
        }

        // setup the initial queue
        final Queue<List<Resource>> queue = new LinkedList<>();
        final List<Resource> thingPath = new ArrayList<>();
        thingPath.add( OWL.Thing );
        queue.offer( thingPath );

        // Get the paths, and display them
        final List<List<Resource>> paths = BFS( model, queue, 4 );
        for ( List<Resource> path : paths ) {
            System.out.println( path );
        }
    }
}
```

1.7 Conclusion

Jena API is not like other graph paradigms like `NetworkX`[4] or `SNAP`[5], which are designed for graph algorithms. Jena was developed very specifically for building Semantic Web and Linked Data applications. In such applications, data are usually saved in RDF formatted databases and the resources usually have URI or URL. Jena can then take the advantage of working with or even querying RDF graphs within Java.

[4]https://networkx.github.io/
[5]http://snap.stanford.edu/index.html

Bibliography

[1] Steve Bratt. Semantic web and other w3c technologies to watch. *Talks at W3C, January*, 2007.

[2] Renato Iannella and James McKinney. vcard ontology-for describing people and organizations. *W3C Group Note NOTE-vcard-rdf-20140522*, 2014.

[3] Apache Jena. Apache jena. *jena. apache. org [Online]. Available: http://jena. apache. org [Accessed: Mar. 20, 2014]*, page 14, 2013.