图形数据库**Neo4J**简介

最近我在用图形数据库来完成对一个初创项目的支持。在使用过程中觉得这种图形数据库实际上挺有意思的。因此在这里给大家做一个简单的介绍。

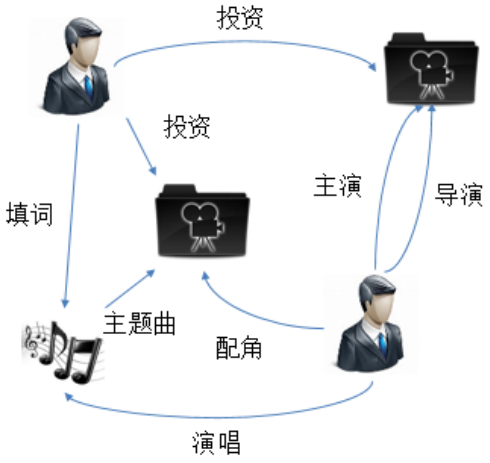
NoSQL数据库相信大家都听说过。它们常常可以用来处理传统的关系型数据库所难以解决的一系列问题。通常情况下，这些NoSQL数据库分为Graph，Document，Column Family以及Key-Value Store等四种。这四种类型的数据库分别使用了不同的数据结构来记录数据。因此它们所适用的场景也不尽相同。

其中最为特别的便是图形数据库了。可以说，它和其它的一系列NoSQL数据库非常不同：丰富的关系表示，完整的事务支持，却没有一个纯正的横向扩展解决方案。

在本文中，我们就将对业界非常流行的图形数据库Neo4J进行简单的介绍。

图形数据库简介

相信您和我一样，在使用关系型数据库时常常会遇到一系列非常复杂的设计问题。例如一部电影中的各个演员常常有主角配角之分，还要有导演，特效等人员的参与。通常情况下这些人员常常都被抽象为Person类型，对应着同一个数据库表。同时一位导演本身也可以是其它电影或者电视剧的演员，更可能是歌手，甚至是某些影视公司的投资者（没错，我这个例子的确是以赵薇为模板的）。而这些影视公司则常常是一系列电影，电视剧的资方。这种彼此关联的关系常常会非常复杂，而且在两个实体之间常常同时存在着多个不同的关系：



在尝试使用关系型数据库对这些关系进行建模时，我们首先需要建立表示各种实体的一系列表：表示人的表，表示电影的表，表示电视剧的表，表示影视公司的表等等。这些表常常需要通过一系列关联表将它们关联起来：通过这些关联表来记录一个人到底参演过哪些电影，参演过哪些电视剧，唱过哪些歌，同时又是哪些公司的投资方。同时我们还需要创建一系列关联表来记录一部电影中哪些人是主角，哪些人是配角，哪个人是导演，哪些人是特效等。可以看到，我们需要大量的关联表来记录这一系列复杂的关系。在更多实体引入之后，我们将需要越来越多的关联表，从而使得基于关系型数据库的解决方案繁琐易错。

这一切的症结主要在于关系型数据库是以为实体建模这一基础理念设计的。该设计理念并没有提供对这些实体间关系的直接支持。在需要描述这些实体之间的关系时，我们常常需要创建一个关联表以记录这些数据之间的关联关系，而且这些关联表常常不用来记录除外键之外的其它数据。也就是说，这些关联表也仅仅是通过关系型数据库所已有的功能来模拟实体之间的关系。这种模拟导致了两个非常糟糕的结果：数据库需要通过关联表间接地维护实体间的关系，导致数据库的执行效能低下；同时关联表的数量急剧上升。

这种执行效能到底低下到什么程度呢？就以建立人和电影之间的投资关系为例。一个使用关联表的设计常常如下所示：

昵称：loveis715
园龄：7年8个月
荣誉：推荐博客
粉丝：394
关注：0
+加关注

< 2017年12月 >

日	一	二	三	四	五	六
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

随笔分类
AWS(1)
C++ 基本功(4)
ExtJS(1)
HTTP(2)
Java基本功(4)
Web Service(19)
WPF(18)
Writing Patterns Line By Line(3)
安全(4)
数据库设计(5)

随笔档案
2017年1月 (1)
2016年3月 (6)
2016年2月 (3)
2016年1月 (2)
2015年7月 (3)
2015年6月 (4)
2015年5月 (5)
2015年4月 (3)
2015年3月 (5)
2013年1月 (1)
2012年7月 (2)
2012年4月 (3)
2012年3月 (1)
2012年2月 (3)
2012年1月 (2)
2011年12月 (4)

积分与排名
积分 - 107764
排名 - 2685

最新评论
1. Re:Cassandra简介

学习到了很多东西，对于后续操作数据库以及为什么项目中这么用变得清晰了

--LC大侠

2. Re:Cassandra简介

感谢博主，作为Cassandra的入门文章，学到很多~

--Winston_bear

3. Re:通过Spring Data Neo4J操作您的图形数据库

楼主，非常感谢

--JungLee

4. Re:Cross-Site Scripting（XSS）简介

<Script>alert("XSS attack availabl e!");</Script>

--一介匹夫

5. Re:通过Spring Data Neo4J操作您的图形数据库

@JungLee您好，我重新下了代码，用的S TS3.6.3运行所有的单元测试是好的。。。 mvn clean install也能运行所有测试。新环境，因为写这个文章是在2年多以前，那个电脑现在都英勇就义.....

--loveis715

6. Re:通过Spring Data Neo4J操作您的图形数据库

楼主，以下是我的测试错误信息，对象初始化错误，是不是缺失代理配置，楼主能帮忙看看吗org.springframework.beans.fact ory.BeanCreationException: Er.....

--JungLee

7. Re:通过Spring Data Neo4J操作您的图形数据库

@JungLee您好，这里Spring Data会自动创建一个Proxy，比如org.springframewo rk.aop.framework.JdkDynamicAopProx y...

--loveis715

8. Re:通过Spring Data Neo4J操作您的图形数据库

楼主， public interface PersonRepositor y extends GraphRepository { } @Auto wired private PersonRepos.....

--JungLee

9. Re:DTO – 服务实现中的核心数据

nice， 概念性的东西很重要

--天空海阔

10. Re:WPF控件编程

mark

--其实我不笨

阅读排行榜

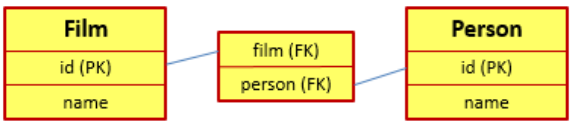
1. REST简介(77218)
2. Cassandra简介(37840)
3. 图形数据库Neo4J简介(23305)
4. WPF拖放功能实现(21807)
5. Microservice架构模式简介(21713)
6. 面试中的Singleton(20802)
7. WPF - 善用路由事件(17775)
8. WPF - 绑定及惯用法(一)(14870)
9. WPF - Adorner(14582)
10. DTO – 服务实现中的核心数据(1143 0)

评论排行榜

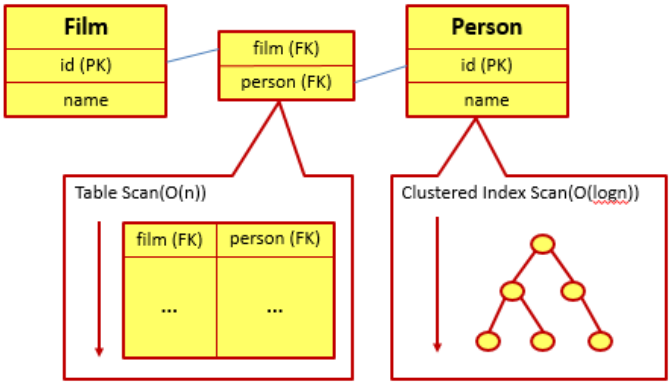
1. REST简介(68)
2. 面试中的Singleton(62)
3. 服务的扩展性(21)
4. Microservice架构模式简介(20)
5. 企业级负载平衡简介(12)
6. Cassandra简介(11)
7. 从Dispatcher.PushFrame()说起(11)
8. WPF - 属性系统 （1 of 4）(10)
9. Cross-Site Scripting（XSS）简介(9)
10. WPF - 在子线程中显示窗口(7)

推荐排行榜

1. REST简介(90)



如果现在我们想要通过该关系找到一部电影的所有投资人，关系型数据库常常会执行哪些操作呢？首先，在关联表中执行一个Table Scan操作（假设没有得到索引支持），以找到所有film域的值与目标电影id相匹配的记录。接下来，通过这些记录中的person域所记录的Person的主键值来从Person表中找到相应的记录。如果记录较少，那么这步就会使用Clustered Index Seek操作（假设是使用该运算符）。整个操作的时间复杂度将变为O(nlogn)：



可以看到，通过关联表组织的关系在运行时的性能并不是很好。如果我们所需要操作的数据集包含了非常多的关系，而且主要是在对这些关系进行操作，那么可以想象到关系数据库的性能将变得有多差。

除了性能之外，关联表数量的管理也是一个非常让人头疼的问题。刚刚我们仅仅是举了一个具有四个实体的例子：人，电影，电视剧，影视公司。现实生活中的例子可不是这么简单。在一些场景下，我们常常需要对更多的实体进行建模，从而完整地描述某一领域内的关联关系。这种关联关系所涵盖的可能包含影视公司的控股关系，各控股公司之间复杂的持股关系以及各公司之间的借贷款情况及担保关系等，更可能是人之间的关系，人与各个品牌之间的代言关系，各个品牌与所属公司之间的关系等。

可以看到，在需要描述大量关系时，传统的关系型数据库已经不堪重负。它所能承担的是较多实体但是实体间关系略显简单的情况。而对于这种实体间关系非常复杂，常常需要在关系之中记录数据，而且大部分对数据的操作都与关系有关的情况，原生支持了关系的图形数据库才是正确的选择。它不仅仅可以为我们带来运行性能的提升，更可以大大提高系统开发效率，减少维护成本。

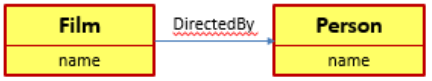
在一个图形数据库中，数据库的最主要组成主要有两种，结点集和连接结点的关系。结点集就是图中一系列结点的集合，比较接近于关系数据库中所最常使用的表。而关系则是图形数据库所特有的组成。因此对于一个习惯于使用关系型数据库开发的人而言，如何正确地理解关系则是正确使用图形数据库的关键。

注：这里的结点集是我自己的翻译。在Neo4J官方文档中，其被称为label。原文为：A label is a named graph construct that is used to group nodes into sets; all nodes labeled with the same label belongs to the same set。我个人觉得生硬地取名为标签反而容易让别人混淆，所以选取了"group nodes into sets"的意译，也好让label和node，即结点集和结点之间的关系能够更好地对应。

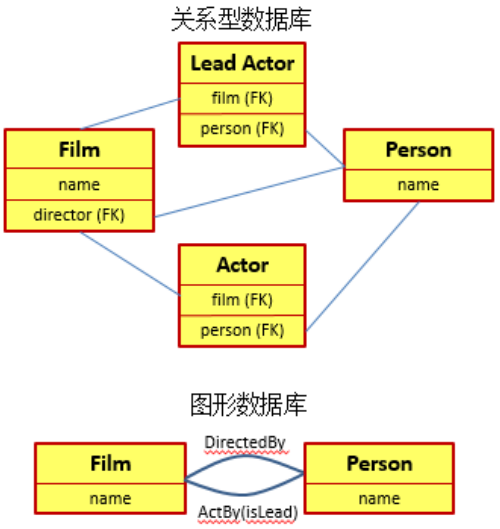
但是不用担心，在了解了图形数据库对数据进行抽象的方式之后，您就会觉得这些数据抽象方式实际上和关系型数据库还是非常接近的。简单地说，每个结点仍具有标示自己所属实体类型的标签，也既是其所属的结点集，并记录一系列描述该结点特性的属性。除此之外，我们还可以通过关系来连接各个结点。因此各个结点集的抽象实际上与关系型数据库中的各个表的抽象还是有些类似的：

关系型数据库

图形数据库

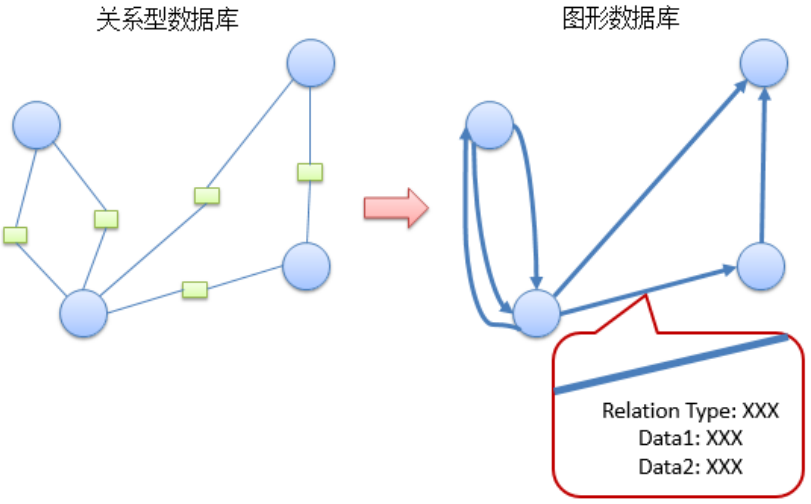


但是在表示关系的时候，关系型数据库和图形数据库就有很大的不同了：



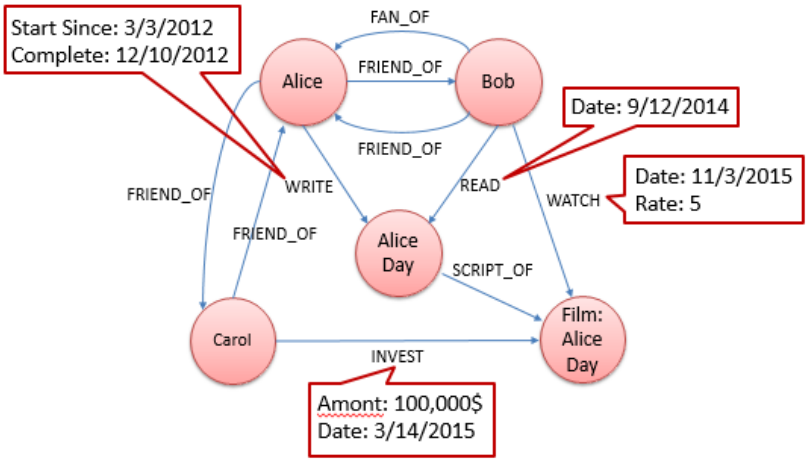
从上图中可以看到，在需要表示多对多关系时，我们常常需要创建一个关联表来记录不同实体的多对多关系，而且这些关联表常常不用来记录信息。如果两个实体之间拥有多种关系，那么我们就需要在它们之间创建多个关联表。而在一个图形数据库中，我们只需要标明两者之间存在着不同的关系，例如用DirectBy关系指向电影的导演，或用ActBy关系来指定参与电影拍摄的各个演员。同时在ActBy关系中，我们更可以通过关系中的属性来表示其是否是该电影的主演。而且从上面所展示的关系的名称上可以看出，关系是有向的。如果希望在两个结点集间建立双向关系，我们就需要为每个方向定义一个关系。

也就是说，相对于关系数据库中的各种关联表，图形数据库中的关系可以通过关系能够包含属性这一功能来提供更为丰富的关系展现方式。因此相较于关系型数据库，图形数据库的用户在对事物进行抽象时将拥有一个额外的武器，那就是丰富的关系：



因此在为图形数据库定义数据展现时，我们应该以一种更为自然的方式来对这些需要展现的事物进行抽象：首先为这些事物定义其所对应的结点集，并定义该结点集所具有的各个属性。接下来辨识出它们之间的关系并创建这些关系的相应抽象。

因此一个图形数据库中所承载的数据最终将有类似于下图所示的结构：



设计一个优质的图

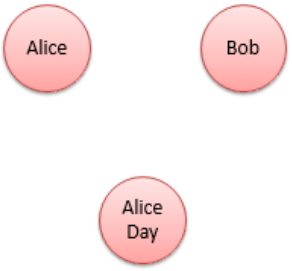
在了解了图形数据库的基础知识之后，我们就要开始尝试使用图形数据库了。首先我们要搞清楚一个问题，那就是如何为我们的图形数据库定义一个设计良好的图？实际上这并不困难，您只需要了解图数据库设计时所使用的一系列要点即可。

首先就是，分清图中结点集，结点以及关系之间的相互联系。在以往的基于关系型数据库的设计中，我们常常会使用一个表来抽象一类事物。如对于人这个概念，我们常常会抽象出一个表，并在表中添加表示两个人的记录，Alice和Bob：

id(PK)	name
1	Alice
2	Bob
...	...

而在图数据库中，这里对应着两个概念：结点集和结点。在通常情况下，图形数据库中的数据展示并不使用结点集，而是独立的结点：

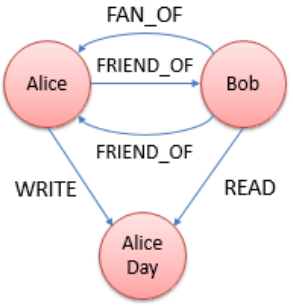
而如果需要在图中添加对书籍的支持，那么这些书籍将仍然被表示为一个结点：



也就是说，虽然在一个图数据库中常常拥有结点集的概念，但是它已经不再作为图数据库的最重要抽象方式了。甚至从某些图形数据库已经允许软件开发人员使用Schemaless结点这一点上来看，它们已经将结点集的概念弱化了。反过来，我们思考的角度就应该是结点个体，以及这些个体之间所存在的一系列关系。

那么我们是不是可以随便定义各个结点所具有的数据呢？不是的。这里最为常用的一个准则就是：Schemaless这种灵活度能为你带来好处。例如相较于强类型语言，弱类型语言可以为软件开发人员带来更大的开发灵活度，但是其维护性和严谨性常常不如强类型语言。同样地，在使用Schemaless结点时也要兼顾灵活性和维护性。

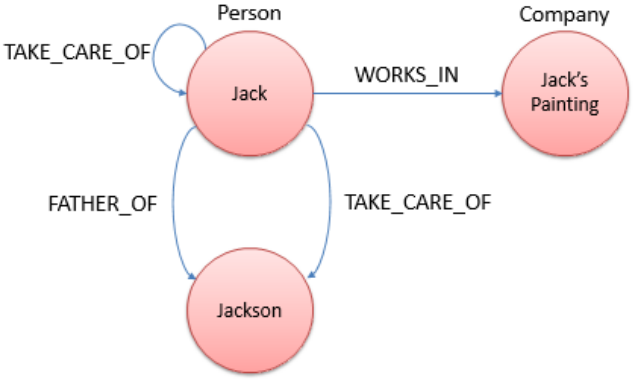
这样我们就可以在结点中添加多种多样的关系，而不用像在关系型数据库中那样需要担心是否需要通过更改数据库的Schema来记录一些外键。这进而允许软件开发人员在各结点间添加多种多样的关系：



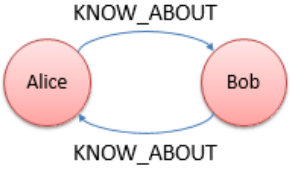
因此在一个图形数据库中，结点集这个概念已经不是最重要的那一类概念了。例如在某些图形数据库中，各个结点的ID并不是按照结点集来组织的，而是根据结点的创建顺序来赋予的。在调试时您可能会发现，某个结点集内的第一个结点的ID是1，第二个结点的ID就是3了。而具有2这个ID的结点则处于另一个结点集中。

那么我们应该如何为业务逻辑定义一个合适的图呢？简单地说，单一事物应该被抽象为一个结点，而同一类型的结点被记录在同一个结点集中。结点集内各结点所包含的数据可能有一些不同，如一个人可能有不同的职责并由此通过不同的关系和其它结点关联。例如一个人既可能是演员，可能是导演，也可能是演员兼导演。在关系型数据库中，我们可能需要为演员和导演建立不同的表。而在图形数据库中，这三种类型的人都是人这个结点集内的数据，而不同的仅仅是它们通过不同的关系连接到不同的结点上了而已。也就是说，在图形数据库中，结点集并不会像关系型数据库中的表一样粒度那么小。

一旦抽象出了各个结点集，我们就需要找出这些结点之间所可能拥有的关系。这些关系不仅仅是跨结点集的。有时候，这些关系是同一结点集内的结点之间的关系，甚至是同一结点指向自身的关系：



这些关系通常都具有一个起点和终点。也就是说，图形数据库中的关系常常是有向的。如果希望在两个结点之间创建一个相互关系，如Alice和Bob彼此相识，我们就需要在他们之间创建两个KNOW_ABOUT关系。其中一个关系由Alice指向Bob，而另一个关系则由Bob指向Alice：



需要注意的一点就是，虽然说图形数据库中的关系是单向的，但是在一些图形数据库的实现中，如Neo4J，我们不仅仅可以查找到从某个结点所发出的关系，也可以找到指向某个结点的各个关系。也就是说，虽然图中的关系是单向的，但是关系在起点和终点都可以被查找到。

在项目中使用**Neo4J**

OK，在大概了解了图形数据库的一些基础知识之后，我们就将以Neo4J为例讲解如何使用一个图形数据库了。Neo4J是Neo Technology所提供的开源图形数据库。其按照上面所介绍的结点/关系模型组织数据，并拥有以下一系列特性：

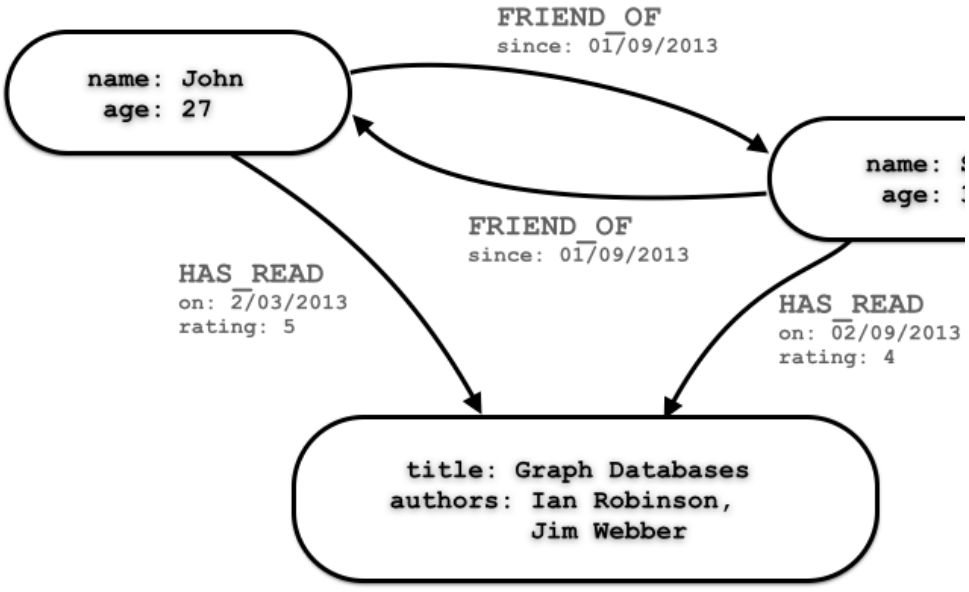
- 对事务的支持。Neo4J强制要求每个对数据的更改都需要在一个事务之内完成，以保证数据的一致性。
- 强大的图形搜索能力。Neo4J允许用户通过Cypher语言来操作数据库。该语言是特意为操作图形数据库设计的，因此其可以非常高效地操作图形数据库。同时Neo4J也提供了面向当前市场一系列流行语言的客户端，以供使用这些语言的开发人员能够快速地对Neo4J进行操作。除此之外，一些项目，如Spring Data Neo4J，也提供了一系列非常简单明了的数据操作方式，使得用户上手变得更为容易。

- 具有一定的横向扩展能力。由于图中的一个结点常常具有和其它结点相关联的关系，因此像一系列Sharding解决方案那样对图进行切割常常并不现实。因此Neo4J当前所提供的横向扩展方案主要是通过Read Replica进行的读写分割。反过来，由于单个Neo4J实例可以存储几十亿个结点及关系，因此对于一般的企业级应用，这种横向扩展能力已经足够了。

好，现在我们就来看一个通过Cypher来创建并操作图形数据库的例子（来自http://neo4j.com/developer/guide-data-modeling/）：

```
// 创建Sally这个Person类型的结点，该结点的name属性为Sally，age属性为32
CREATE (sally:Person { name: 'Sally', age: 32 })

// 创建John结点
CREATE (john:Person { name: 'John', age: 27 })

// 创建Graph Databases一书所对应的结点
CREATE (gdb:Book { title: 'Graph Databases',
                   authors: ['Ian Robinson', 'Jim Webber'] })

// 在Sally和John之间建立朋友关系，这里的since值应该是timestamp。请自行回忆各位的日期是如何在关系数据库中记录的~~~
CREATE (sally)-[:FRIEND_OF { since: 1357718400 }]->(john)

// 在Sally和Graph Databases一书之间建立已读关系
CREATE (sally)-[:HAS_READ { rating: 4, on: 1360396800 }]->(gdb)

// 在John和Graph Databases一书之间建立已读关系
CREATE (john)-[:HAS_READ { rating: 5, on: 1359878400 }]->(gdb)
```

该段语句创建了三个结点：Person结点Sally和John，以及Book结点gdb。同时还指定了它们之间的关系：



注：原图来自http://neo4j.com/developer/guide-data-modeling/

想节省时间花在有用的地方，但是为了完整性不得不写

这里有一点需要注意的地方，那就是关系是单向的。如果希望建立一个双向的关系，就像上面Sally和John互为朋友关系那样，我们按理来说应该需要重复执行创建关系的过程。由于我没有试过最新版本的Neo4J（因为它最近有一个破坏了后向兼容性的更改，我们暂时没有办法升级Neo4J，也就没有办法确认上面的代码是不是少创建了一次FRIEND_OF），因此请读者注意。如果有谁实验了，请将结果添加到Comment中，感激不尽。

有了数据，我们就可以对数据进行操作了。虽然Cypher和SQL操作的是不同的数据结构，但是他们的语法结构还是非常相似的。例如下面的语句就用来获得Sally和John是什么时候成为朋友的（来自http://neo4j.com/developer/guide-data-modeling/）：

```
MATCH (sally:Person { name: 'Sally' })
MATCH (john:Person { name: 'John' })
MATCH (sally)-[r:FRIEND_OF]-(john)
```

```
4 RETURN r.since as friends_since
```

而且还有一些更复杂的语法。如下面的操作就用来判断Sally和John谁先读了《Graph Databases》一书：

```
1 MATCH (people:Person)
2 WHERE people.name = 'John' OR people.name = 'Sally'
3 MATCH (people)-[r:HAS_READ]->(gdb:Book { title: 'Graph Databases' })
4 RETURN people.name as first_reader
5 ORDER BY r.on
6 LIMIT 1
```

当然，谁都不愿意写SQL，否则Hibernate也发展不起来。一个当前较为流行的解决方案就是Spring Data Neo4J。通过定义一系列Java类并在其上使用一系列标记，我们就能在系统中使用Neo4J了。现在我们就以3.4.4版本的Spring Data Neo4J为例讲解如何对其进行使用。

首先，我们需要为将要存入到Neo4J中的数据定义相应的数据类型（来自于http://projects.spring.io/spring-data-neo4j/）：

```
1 // 通过NodeEntity标记来创建一个需要被存入到Neo4J的数据类型
2 @NodeEntity
3 public class Movie {
4     // 通过GraphId标记来指定作为ID的域。如果是新建一个结点，那么我们需要将该域置空（null）。不知道4.0.0是否还有这个限制
5     @GraphId Long id;
6
7     // 创建针对该域的索引
8     @Indexed(type = FULLTEXT, indexName = "search")
9     String title;
10
11    // 对Person类的直接引用。在保存时，其将会被自动保存到Person结点集中并保持Movie类对该实例的引用
12    Person director;
13
14    // 通过RelatedTo标记来标示当前集合所引用的各个实体对应于当前Movie实例的关系是ACTS_IN。注意这里标明了方向是INCOMING，也就是说，其方向是从Person指向Movie，也既是Person ACTS_IN Movie。而在Person中，我们同样可以拥有一个Movie的集合，同样使用RelatedTo标记使用ACTS_IN关系，而direction为OUTGOING
15    // 另，RelatedTo和RelatedToVia标记按理来说在4.0.0里已经被丢弃了，但是在官方示例中依然被使用
16    @RelatedTo(type="ACTS_IN", direction = INCOMING)
17    Set<Person> actors;
18
19    @RelatedToVia(type = "RATED")
20    Iterable<Rating> ratings;
21
22    // 使用自定义Query读取数据
23    @Query("start movie=node({self}) match
24            movie-->genre<--similar return similar")
25    Iterable<Movie> similarMovies;
26 }
```

接下来，您就可以创建一个用来对刚刚所定义的类型进行CRUD操作的Repository了：

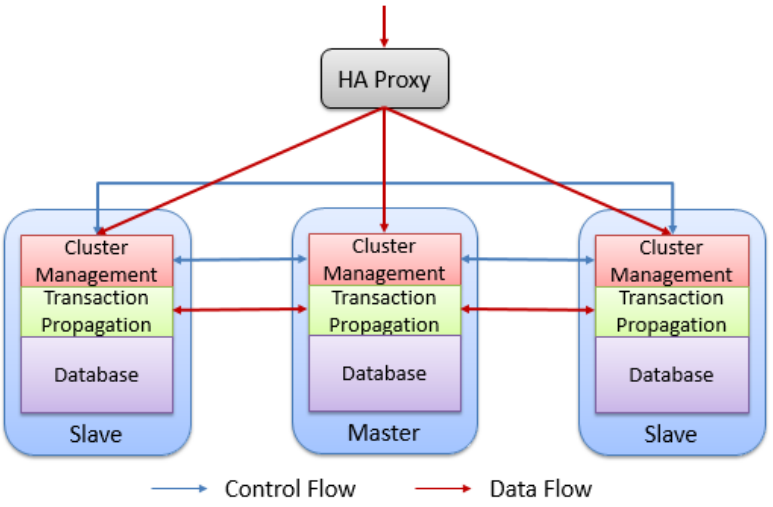
```
1 // 从GraphRepository接口直接得到对Movie类进行CRUD的功能
2 interface MovieRepository extends GraphRepository<Movie> {
3     // 通过Cypher语句来执行特定的操作
4     @Query("start movie={0} match m<-[rating:RATED]-user
5             return rating")
6     Iterable<Rating> getRatings(Movie movie);
7
8     // 和Spring Data JPA一样，可以通过特定的规则组合函数名来添加筛选条件
9     Iterable<Person> findByActorsMoviesActorName(name)
10 }
```

最后我们需要在Spring的配置文件中指定这些组成所在的位置：

```
1 <neo4j:config storeDirectory="target/graph.db" base-package="com.example.neo4j.entity"/>
2 <neo4j:repositories base-package="com.example.neo4j.repository" />
```

Neo4J集群

OK，在了解了如何使用Neo4J之后，下一步要考虑的就是如何通过搭建一个Neo4J集群来提供一个具有高可用性，高吞吐量的解决方案了。首先您要知道的是，和其它NoSQL数据库所提供的近乎无限的横向扩展能力相比，Neo4J集群实际上是有一定限制的。为了能更好地理解这些限制，就让我们首先看一看Neo4J集群的架构以及它到底是如何工作的：



上图展示了一个由三个Neo4J结点所组成的Master-Slave集群。通常情况下，每个Neo4J集群都包含一个Master和多个Slave。该集群中的每个Neo4J实例都包含了图中的所有数据。这样任何一个Neo4J实例的失效都不会导致数据的丢失。集群中的Master主要负责数据的写入，接下来Slave则会将Master中的数据更改同步到自身。如果一个写入请求到达了Slave，那么该Slave也将会就该请求与Master通信。此时该写入请求将首先被Master执行，再异步地将数据更新到各个Slave中。所以在上图中，您可以看到表示数据写入方式的红线有从Master到Slave，也有从Slave到Master，但是并没有从Slave到Slave。而所有这一切都是通过Transaction Propagation组成来协调完成的。

有些读者可能已经注意到了：Neo4J集群中数据的写入是通过Master来完成的，那是不是Master会变成系统的写入瓶颈呢？答案是几乎不会。首先是图数据修改的复杂性导致其并不会像栈，数组等数据类型那样容易被修改。在修改一个图的时候，我们不但需要修改图结点本身，还要维护各个关系，本身就是一个比较复杂的过程，对用户而言也是较难理解的。因此对图所进行的操作也常常是读比写多很多。同时Neo4J内部还有一个写队列，可以用来暂时缓存向Neo4J实例的写入操作，从而使得Neo4J能够处理突然到来的大量写入操作。而在最坏的情况就是Neo4J集群需要面对持续的大量的写入操作。在这种情况下，我们就需要考虑Neo4J集群的纵向扩展了，因为此时横向扩展无益于解决这个问题。

反过来，由于数据的读取可以通过集群中的任意一个Neo4J实例来完成，因此Neo4J集群的读吞吐量可以在理论上做到随集群中Neo4J实例的个数线性增长。例如如果一个拥有5个结点的Neo4J集群可以每秒响应500个读请求，那么再添加一个结点就可以将其扩容为每秒响应600个读请求。

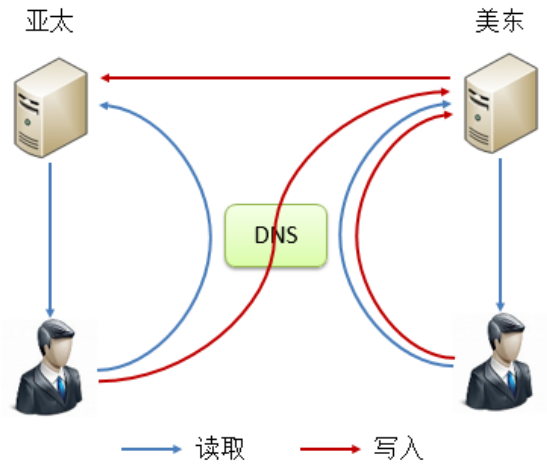
但在请求量非常巨大而且访问的数据分布非常随机的情况下，另一个问题就可能发生了，那就是Cache-Miss。Neo4J内部使用一个缓存记录最近所访问的数据。这些缓存数据会保存在内存中以便快速地响应数据读取请求。但是在请求量非常巨大而且所访问数据分布随机的情况下，Cache-Miss将会持续地发生，使得每次对数据的读取都要经过磁盘查找来完成，从而大大地降低了Neo4J实例的运行效率。而Neo4J所提供的解决方案被称为Cache-based Sharding。简单地说，就是使用同一个Neo4J实例来响应一个用户所发送的所有需求。其背后的原理也非常简单，那就是同一个用户在一段时间内所访问的数据常常是类似的。因此将这个用户的一系列数据请求发送到同一个Neo4J服务器实例上可以很大程度上降低发生Cache-Miss的概率。

Neo4J数据服务器中的另一个组成Cluster Management则用来负责同步集群中各个实例的状态，并监控其它Neo4J结点的加入和离开。同时其还负责维护领导选举结果的一致性。如果Neo4J集群中失效的结点个数超过了集群中结点个数的一半，那么该集群将只接受读取操作，直到有效结点重新超过集群结点数量的一半。

在启动时，一个Neo4J数据库实例将首先尝试着加入由配置文件所标明的集群。如果该集群存在，那么它将作为一个Slave加入。否则该集群将被创建，并且其将被作为该集群的Master。

如果Neo4J集群中的一个Neo4J实例失效了，那么其它Neo4J实例会在短时间内探测到该情况并将其标示为失效，直到其重新恢复到正常状态并将数据同步到最新。这其中有一个特殊情况，那就是Master失效的情况。在该情况下，Neo4J集群将会通过内置的Leader选举功能选举出新的Master。

在Cluster Management组成的帮助下，我们还可以创建一个Global Cluster。其拥有一个Master Cluster以及多个Slave Cluster。该集群组建方式允许Master Cluster和Slave Cluster处于不同区域的服务集群中。这样就可以允许服务的用户访问距离自己最近的服务。和Neo4J集群中的Master及Slave实例的关系类似，数据的写入通常都是在Master Cluster中进行，而Slave Cluster将只负责提供数据读取服务。



提高**Neo4J**的性能

相信您在上面对Neo4J集群的讲解中已经看出，Neo4J集群实际上还是有一些限制的。这些限制将可能导致Neo4J集群在总的系统容量，如存储结点的数目或写吞吐量等众多方面存在着一定的瓶颈。在《服务的扩展性》一文中我们曾经介绍过，通过纵向扩展，我们同样可以提高服务的整体性能。除了通过为Neo4J提供具有更高容量的硬件之外，更有效地使用Neo4J也是纵向扩展的一个重要方法。

和SQL Server等数据库所提供的Execution Plan类似，Neo4J也提供了Execution Plan。在执行一个请求时，Neo4J将会把这个请求拆解为一系列较小的操作符（Operator）。每个操作符都将执行一部分工作，并彼此相互协作完成对请求的响应。与SQL Server的Execution Plan类似，Neo4J的Execution Plan同样拥有Scan，Seek，Merge，Filter等多种类型的操作。我们可以通过EXPLAIN或PROFILE命令得到一个请求将被如何执行的树形表示。通过查看这些树形表示，软件开发人员能够了解一个请求在Neo4J中是如何运行的：

```
ColumnFilter
  |
  +ExtractPath
    |
    +Filter
      |
      +TraversalMatcher

+------------------+------+---------+-------------+-----------------------------------------------------------
|         Operator | Rows |  DbHits | Identifiers |
+------------------+------+---------+-------------+-----------------------------------------------------------
|     ColumnFilter |   17 |       0 |             |
|      ExtractPath |   17 |       0 |           p |
|           Filter |   17 | 1147092 |             | (((((any(-_-INNER-_- in Collection(List({  AUTOSTRING0
                                                    where Property(m,URI(0)) == -_-INNER-_-)
                                                    AND nonEmpty(FilterFunction(r,x,RelationshipTypeFunctio
                                                    AND nonEmpty(FilterFunction(r,y,RelationshipTypeFunctio
                                                    AND nonEmpty(FilterFunction(r,z,RelationshipTypeFunctio
                                                    AND nonEmpty(FilterFunction(r,u,RelationshipTypeFunctio
                                                    AND nonEmpty(FilterFunction(r,v,RelationshipTypeFunctio
| TraversalMatcher | 5312 |   11810 |             |
+------------------+------+---------+-------------+-----------------------------------------------------------

Total database accesses: 1158902
```

以下是当前版本的Neo4J所支持的所有操作符：http://neo4j.com/docs/stable/execution-plans.html。

有通过Execution Plan调优经验的读者可能第一眼就看到了一个操作符：Node Index Seek。它的名字直接透露了Neo4J中的另一个调优利器：索引。我们知道，只要查找出的结果集中记录的数据并不是很多，那么SQL Server中的Clusted Index Seek常常是具有最优效率的操作。因此在Neo4J中，我们同样需要尽量合理地使用索引，从而使得Neo4J所生成的Execution Plan能使用基于索引的一系列操作符。让我们回忆一下之前展示给大家的按照Spring Data Neo4J方式抽象出的Movie类：

```
@NodeEntity
public class Movie {
    @GraphId Long id;

    @Indexed(type = FULLTEXT, indexName = "search")
    String title;

    ……
}
```

上面的代码展示了我们应该如何通过@Indexed标记来创建一个索引。如果您是直接使用Cypher来操

作Neo4J的，那么您可以通过以下语句来创建一个索引：

```
1 CREATE INDEX ON :Movie(title)
```

而这里有一个和SQL Server略有不一致的地方，那就是对@GraphId标记的理解。在SQL Server中，Primary Key实际上是与索引没有任何关联的，只是在默认情况下，其常常会被自动地添加一个索引。而在Neo4J中，由@GraphId标记所修饰的域则更像是Neo4J的内部实现。Neo4J通过该域所记录的值执行对结点的访问，而不是在其上自动地添加了一个索引。

还有一个可能非常容易影响Neo4J性能的可能，那就是尝试使用Neo4J记录不适合它记录的数据。在本文的一开始我们就已经介绍了Neo4J所适合的领域，那就是记录图形数据，及结点集和结点之间的关系。而对于其它一些类型的数据，如用户的用户名/密码对，那就不是图形数据库所擅长的领域了。在这些情况下，我们应该选取合适的数据库来记录这些数据。在一个大型系统中，多种不同类型的数据库相互协作是经常有的事情，所以没有必要非要将一些本来应该由其它类型的数据库所记录的数据硬生生地记录在Neo4J中。

和其它数据库合作

在上面我们刚刚提到了不应该由Neo4J记录不适合它记录的数据，以保证Neo4J不被不合理的使用方式拉低其执行效率。那么这些数据应该记录在哪里呢？答案非常简单：适合记录这些数据的其它类型的数据库。

可能你觉得我这句话是废话。其实我也这么觉得。而我想在这里介绍的是，如何完成Neo4J和其它数据库之间的集成，从而使它们协同工作，向用户提供完整的服务。对于某些系统，我们可以允许这些数据库之间拥有一定程度的不一致；而对于另外一些系统，我们则需要时刻保证数据的一致性。

Neo4J所提出支持的技术方案主要有三种：Event-based Synchronization，Periodic Synchronization以及Periodic Full Export/Import of Data。Event-based Synchronization实际上就是通过同时向基于Neo4J的后台和基于其它数据库的后台发送相同的消息，并由这些后台完成对数据的写入。Periodic Synchronization则是定时地将一个数据库中对数据的更改同步到另一个数据库中。而Periodic Full Export/Import of Data则是通过将一个数据库中的所有数据导入到另外一个数据库中的方式来完成的。

这三种解决方案都是用来处理Neo4J所记录的数据与其它数据库相同的情况。而更为常见的情况则是，Neo4J记录实体关系比较复杂的图，其它数据库则用来记录具有其它类型表现形式的数据。Neo4J和这些数据库之间的数据只有一部分交集，而每个数据库都拥有自己所特有的数据。针对这种情况的处理方法则常常是多步提交。例如在一个交友网站中，用户可以在页面上完成自身账户的设置，如用户名，密码等，并可以在下一步添加好友界面中添加一系列好友以及有关于该好友的注释。那么在该系统中，用户自身的账户设置就可能记录在关系型数据库中，而有关好友的相关信息则记录在图形数据库中。如果将这两步中的所有信息作为一个请求发送到后台，那么就可能出现在某个数据库上成功保存而在另一个数据库上保存失败的情况。为了避免这种情况，我们就需要将填充这两部分资料的信息分为两个页面，而在每个页面下部提供一个”保存并进行下一步”的按钮。这样如果第一步设置账户的步骤无法正常保存，那么用户就没有办法进行下一步添加朋友的操作。而在添加朋友这步中，如果图形数据库无法正常保存，那么我们将可以明确地告诉用户添加朋友失败，从而允许用户重试。

其实很多时候，跨不同数据库保存数据的问题都可以通过调整设计的方式来解决，况且这些数据库所记录的数据常常具有非常不同的数据结构。因此就用户来说，分成多步提交常常是一个非常自然的使用方式。

分类：Web Service,数据库设计

标签：Neo4J Graph database NoSQL 图形数据库

loveis715
关注 - 0
粉丝 - 394
10 0
荣誉：推荐博客
+加关注

« 上一篇：.NET - 基于事件的异步模型
» 下一篇：Cassandra简介

posted @ 2016-03-16 01:19 loveis715 阅读(23306) 评论(4) 编辑 收藏

发表评论

#1楼 2016-05-31 16:45 | 黑色卷纸

非常好的文章，深受启发，感谢博主。

支持(0) 反对(

#2楼 2016-10-26 16:39 | 净静云

Mark

支持(0) 反对(

#3楼 2016-11-21 10:52 | 骨折仔

ding

支持(0) 反对(

#4楼 2017-05-08 15:47 | 悦光阴

学习了，谢谢分享

支持(0) 反对(

刷新评论 刷新页面 返回顶

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。