Spark GraphX基本操作

```
import org.apache.spark.SparkContext
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.graphx.Graph
import org.apache.spark.graphx.Edge
import org.apache.spark.graphx.VertexRDD
import org.apache.spark.graphx.util.GraphGenerators
import org.apache.spark.graphx.GraphLoader
import org.apache.spark.storage.StorageLevel
import org.apache.spark.rdd.RDD

object SparkGraphx1 {

  def main(args: Array[String]) {

    val sc = new SparkContext("spark://centos.host1:7077", "Spark Graphx")

    //创建点RDD
    val users: RDD[(VertexId, (String, String))] = sc.parallelize(Array(
      (3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
      (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
    //创建边RDD
    val relationships: RDD[Edge[String]] = sc.parallelize(Array(
      Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),
      Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
    //定义一个默认用户，避免有不存在用户的关系
    val defaultUser = ("John Doe", "Missing")
    //构造Graph
    val graph = Graph(users, relationships, defaultUser)

    //点RDD、边RDD过滤
    val fcount1 = graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
    println("postdocs users count: " + fcount1)
    val fcount2 = graph.edges.filter(edge => edge.srcId > edge.dstId).count
    println("srcId > dstId edges count: " + fcount2)
    val fcount3 = graph.edges.filter { case Edge(src, dst, prop) => src > dst }.count
    println("srcId > dstId edges count: " + fcount3)

    //Triplets(三元组)，包含源点、源点属性、目标点、目标点属性、边属性
    val triplets: RDD[String] = graph.triplets.map(triplet => triplet.srcId + "-" +
      triplet.srcAttr._1 + "-" + triplet.attr + "-" + triplet.dstId + "-" + triplet.dstAttr._1)
    triplets.collect().foreach(println(_))

    //度、入度、出度
    val degrees: VertexRDD[Int] = graph.degrees;
    degrees.collect().foreach(println)
    val inDegrees: VertexRDD[Int] = graph.inDegrees
    inDegrees.collect().foreach(println)
    val outDegrees: VertexRDD[Int] = graph.outDegrees
    outDegrees.collect().foreach(println)

    //构建子图
    val subGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
    subGraph.vertices.collect().foreach(println(_))
    subGraph.triplets.map(triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
      .collect().foreach(println(_))

    //Map操作，根据原图的一些特性得到新图，原图结构是不变的，下面两个逻辑是等价的，但是第一个不会被graphx系统优化
    val newVertices = graph.vertices.map { case (id, attr) => (id, (attr._1 + "-1", attr._2 + "-2")) }
    val newGraph1 = Graph(newVertices, graph.edges)
    val newGraph2 = graph.mapVertices((id, attr) => (id, (attr._1 + "-1", attr._2 + "-2")))

    //构造一个新图，顶点属性是出度
```

```
    val inputGraph: Graph[Int, String] =
      graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) => degOpt.getOrElse(0))
    //根据顶点属性为出度的图构造一个新图，依据PageRank算法初始化边与点
    val outputGraph: Graph[Double, Double] =
      inputGraph.mapTriplets(triplet => 1.0 / triplet.srcAttr).mapVertices((id, _) => 1.0)

    //图的反向操作，新的图形的所有边的方向相反，不修改顶点或边性属性、不改变的边的数目，它可以有效地实现不必要的数据移动或
复制
    var rGraph = graph.reverse

    //Mask操作也是根据输入图构造一个新图，达到一个限制制约的效果
    val ccGraph = graph.connectedComponents()
    val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
    val validCCGraph = ccGraph.mask(validGraph)

    //Join操作，原图外连出度点构造一个新图    ，出度为顶点属性
    val degreeGraph2 = graph.outerJoinVertices(outDegrees) { (id, attr, outDegreeOpt) =>
      outDegreeOpt match {
        case Some(outDeg) => outDeg
        case None => 0 //没有出度标识为零
      }
    }

    //缓存。默认情况下,缓存在内存的图会在内存紧张的时候被强制清理，采用的是LRU算法
    graph.cache()
    graph.persist(StorageLevel.MEMORY_ONLY)
    graph.unpersistVertices(true)

    //GraphLoader构建Graph
    var path = "/user/hadoop/data/temp/graph/graph.txt"
    var minEdgePartitions = 1
    var canonicalOrientation = false // if sourceId < destId this value is true
    val graph1 = GraphLoader.edgeListFile(sc, path, canonicalOrientation, minEdgePartitions,
      StorageLevel.MEMORY_ONLY, StorageLevel.MEMORY_ONLY)

    val verticesCount = graph1.vertices.count
    println(s"verticesCount: $verticesCount")
    graph1.vertices.collect().foreach(println)

    val edgesCount = graph1.edges.count
    println(s"edgesCount: $edgesCount")
    graph1.edges.collect().foreach(println)

    //PageRank
    val pageRankGraph = graph1.pageRank(0.001)
    pageRankGraph.vertices.sortBy(_._2, false).saveAsTextFile("/user/hadoop/data/temp/graph/graph.pr")
    pageRankGraph.vertices.top(5)(Ordering.by(_._2)).foreach(println)

    //Connected Components
    val connectedComponentsGraph = graph1.connectedComponents()
    connectedComponentsGraph.vertices.sortBy(_._2, false).saveAsTextFile("/user/hadoop/data/temp/graph/graph.cc")
    connectedComponentsGraph.vertices.top(5)(Ordering.by(_._2)).foreach(println)

    //TriangleCount主要用途之一是用于社区发现 保持sourceId小于destId
    val graph2 = GraphLoader.edgeListFile(sc, path, true)
    val triangleCountGraph = graph2.triangleCount()
    triangleCountGraph.vertices.sortBy(_._2, false).saveAsTextFile("/user/hadoop/data/temp/graph/graph.tc")
    triangleCountGraph.vertices.top(5)(Ordering.by(_._2)).foreach(println)

    sc.stop()
  }

}
```