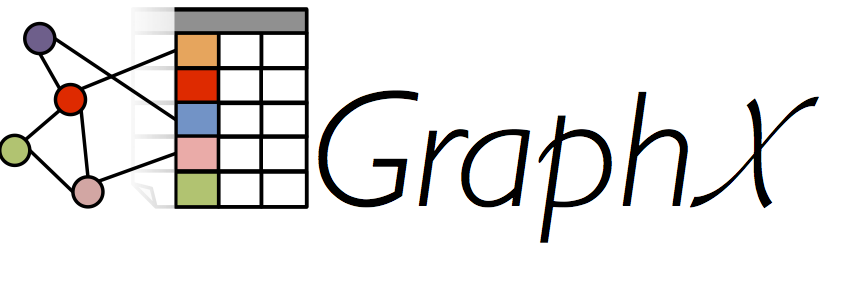
# 一.GraphX简介



   GraphX是一个新的Spark API，它用于图和分布式图(graph-parallel)的计算。GraphX通过引入弹性分布式属性图（[Resilient Distributed Property Graph](https://endymecy.gitbooks.io/spark-graphx-source-analysis/content/property-graph.md)）： 顶点和边均有属性的有向多重图，来扩展Spark RDD。为了支持图计算，GraphX开发了一组基本的功能操作以及一个优化过的Pregel API。另外，GraphX包含了一个快速增长的图算法和图builders的 集合，用以简化图分析任务。

  从社交网络到语言建模，不断增长的规模以及图形数据的重要性已经推动了许多新的分布式图系统（如[Giraph](http://giraph.apache.org/" \t "https://endymecy.gitbooks.io/spark-graphx-source-analysis/content/_blank)和[GraphLab](http://graphlab.org/" \t "https://endymecy.gitbooks.io/spark-graphx-source-analysis/content/_blank)）的发展。 通过限制计算类型以及引入新的技术来切分和分配图，这些系统可以高效地执行复杂的图形算法，比一般的分布式数据计算（data-parallel，如spark、MapReduce）快很多。

分布式图（graph-parallel）计算和分布式数据（data-parallel）计算类似，分布式数据计算采用了一种record-centric的集合视图，而分布式图计算采用了一种vertex-centric的图视图。 分布式数据计算通过同时处理独立的数据来获得并发的目的，分布式图计算则是通过对图数据进行分区（即切分）来获得并发的目的。更准确的说，分布式图计算递归地定义特征的转换函数（这种转换函数作用于邻居特征），通过并发地执行这些转换函数来获得并发的目的。

  分布式图计算比分布式数据计算更适合图的处理，但是在典型的图处理流水线中，它并不能很好地处理所有操作。例如，虽然分布式图系统可以很好的计算PageRank以及label diffusion，但是它们不适合从不同的数据源构建图或者跨过多个图计算特征。 更准确的说，分布式图系统提供的更窄的计算视图无法处理那些构建和转换图结构以及跨越多个图的需求。分布式图系统中无法提供的这些操作需要数据在图本体之上移动并且需要一个图层面而不是单独的顶点或边层面的计算视图。例如，我们可能想限制我们的分析到几个子图上，然后比较结果。 这不仅需要改变图结构，还需要跨多个图计算。

发展历程

 早在 0.5 版本，Spark 就带了一个小型的Bagel 模块，提供了类似 Pregel 的功能。这个版本还非常原始，性能和功能都比较弱，属于实验型产品。

 到 0.8 版本时，鉴于业界对分布式图计算的需求日益见涨，Spark 开始独立一个分支Graphx-Branch，作为独立的图计算模块，借鉴 GraphLab，开始设计开发 GraphX。

 在 0.9 版本中，这个模块被正式集成到主干，虽然是 Alpha 版本，但已可以试用，小面包圈Bagel 告别舞台。1.0 版本，GraphX 正式投入生产使用。现在spark的版本已更新到2.4.0

# 图计算优势

1.基于内存实现了数据的复用与快速读取

具有较多迭代次数是图计算算法的一个重要特点。在海量数据背景下，如何保证图计算算法的执行效率是所有图计算模型面对的一个难题。基于MapReduce的图计算模型在进行迭代计算过程中，中间数据的操作都是基于磁盘展开的。这使得数据的转换和复制开销非常大，其中包括序列化开销等。除此之外，许多与图结构信息相关的数据无法进行重用，这使得系统不得不反复读取一些相同的数据对图进行重构。相对于传统的图计算模型，GraphX得益于Spark中的RDD和任务调度策略，能够对图数据进行缓存和Pipline操作，实现了图的复用与快速运算。

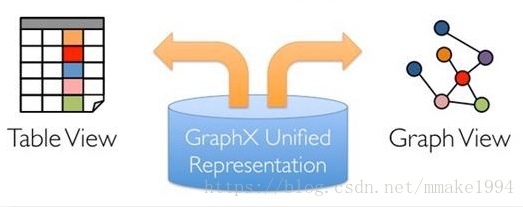
2.统一了图视图和表视图

传统图计算模型都是将表视图和图视图分别进行实现的，这意味着图计算模型要针对不同的视图分别进行维护，而且视图间的转换也比较烦琐。GraphX通过弹性分布式属性图统一了表视图和图视图，即两种视图对应同一物理存储但是各自具有独立的操作，这使得操作更具有灵活性和高效性，如下图，基于GraphX执行图计算任务，一方面用户不必再对不同的组件进行学习、部署、维护和管理，降低运维成本；另一方面更有利于实现基于内存的Pipeline操作。

此外，在GraphX中还提出了一种路由表（routing table）。该表描述了图存储时的一些元数据信息，记录了顶点所在的物理位置以及顶点间的关系。利用该路由表能够极大地提高图计算的执行效率。

3.能与Spark框架上的组件无缝集成

仅从图计算性能方面对比，目前性能最好的模型仍然是GraphLab。但是单一组件或单一性能无法决定整个系统的综合处理能力。尤其是在大数据背景下，任何数据处理业务都需要同一平台上的多个组件通过相互协作来完成，例如海量数据的获取、表示、分析、查询、可视化以及数据通信等各环节对应着一系列专用的组件。然而不同组件之间在集成性方面存在着很大差异。由于GraphX是Spark上的一个组件，能与Spark Streaming、Spark SQL和SparkMLlib等进行无缝衔接，例如可以利用SparkSQL进行ETL，然后将处理后的数据传给GraphX进行计算；或者GraphX与MLlib结合对图数据进行深度挖掘，这些都是Spark一栈式解决方案的具体应用，而Garph、GraphLab等则不具备这一特点。因此，在Spark平台上进行图计算，首选GraphX.



# Graphx的存储原理

巨型图的存储总体上有边分割和点分割两种存储方式。2013 年，GraphLab2.0 将其存储方式由边分割变为点分割，在性能上取得重大提升，目前基本上被业界广泛接受并使用。

边分割（Edge-Cut）：每个顶点都存储一次，但有的边会被打断分到两台机器上。这样做的好处是节省存储空间；坏处是对图进行基于边的计算时，对于一条两个顶点被分到不同机器上的边来说，要跨机器通信传输数据，内网通信流量大。

点分割（Vertex-Cut）：每条边只存储一次，都只会出现在一台机器上。邻居多的点会被复制到多台机器上，增加了存储开销，同时会引发数据同步问题。好处是可以大幅减少内网通信量。

图计算使用的是点分割方式存储图。这种存储方式特点是任何一条边只会出现在一台机器上，每个点有可能分布到不同的机器上。 当点被分割到不同机器上时，是相同的镜像，但是有一个点作为主点,其他的点作为虚点，当点的数据发生变化时,先更新主点的数据，然后将所有更新好的数据发送到虚点所在的所有机器，更新虚点。 这样做的好处是在边的存储上是没有冗余的，而且对于某个点与它的邻居的交互操作，只要满足交换律和结合律，就可以在不同的机器上面执行，网络开销较小。但是这种分割方式会存储多份点数据，更新点时， 会发生网络传输，并且有可能出现同步问题。

GraphX在进行图分割时，有几种不同的分区(partition)策略，它通过PartitionStrategy专门定义这些策略。在PartitionStrategy中，总共定义了EdgePartition2D、EdgePartition1D、RandomVertexCut以及 CanonicalRandomVertexCut这四种不同的分区策略。

## 1 RandomVertexCut

**case object** RandomVertexCut **extends** PartitionStrategy {  
 **override def** getPartition(src: VertexId, dst: VertexId, numParts: PartitionID): PartitionID = {  
 math.*abs*((src, dst).hashCode()) % numParts  
 }  
}

 这个方法比较简单，通过取源顶点和目标顶点id的哈希值来将边分配到不同的分区。这个方法会产生一个随机的边分割，两个顶点之间相同方向的边会分配到同一个分区。

## 2 CanonicalRandomVertexCut

**case object** CanonicalRandomVertexCut **extends** PartitionStrategy {  
 **override def** getPartition(src: VertexId, dst: VertexId, numParts: PartitionID): PartitionID = {  
 **if** (src < dst) {  
 math.*abs*((src, dst).hashCode()) % numParts  
 } **else** {  
 math.*abs*((dst, src).hashCode()) % numParts  
 }  
 }  
}

这种分割方法和前一种方法没有本质的不同。不同的是，哈希值的产生带有确定的方向（即两个顶点中较小id的顶点在前）。两个顶点之间所有的边都会分配到同一个分区，而不管方向如何。

## 3 EdgePartition1D

**case object** EdgePartition1D **extends** PartitionStrategy {  
 **override def** getPartition(src: VertexId, dst: VertexId, numParts: PartitionID): PartitionID = {  
 **val** mixingPrime: VertexId = 1125899906842597L  
 (math.*abs*(src \* mixingPrime) % numParts).toInt  
 }  
}

这种方法仅仅根据源顶点id来将边分配到不同的分区。有相同源顶点的边会分配到同一分区。

## 4 EdgePartition2D

**case object** EdgePartition2D **extends** PartitionStrategy {  
 **override def** getPartition(src: VertexId, dst: VertexId, numParts: PartitionID): PartitionID = {  
 **val** ceilSqrtNumParts: PartitionID = math.*ceil*(math.*sqrt*(numParts)).toInt  
 **val** mixingPrime: VertexId = 1125899906842597L  
 **if** (numParts == ceilSqrtNumParts \* ceilSqrtNumParts) {  
 // Use old method for perfect squared to ensure we get same results  
 **val** col: PartitionID = (math.*abs*(src \* mixingPrime) % ceilSqrtNumParts).toInt  
 **val** row: PartitionID = (math.*abs*(dst \* mixingPrime) % ceilSqrtNumParts).toInt  
 (col \* ceilSqrtNumParts + row) % numParts  
  
 } **else** {  
 // Otherwise use new method  
 **val** cols = ceilSqrtNumParts  
 **val** rows = (numParts + cols - 1) / cols  
 **val** lastColRows = numParts - rows \* (cols - 1)  
 **val** col = (math.*abs*(src \* mixingPrime) % numParts / rows).toInt  
 **val** row = (math.*abs*(dst \* mixingPrime) % (**if** (col < cols - 1) rows **else** lastColRows)).toInt  
 col \* rows + row  
  
 }  
 }  
}

这种分割方法同时使用到了源顶点id和目的顶点id。它使用稀疏边连接矩阵的2维区分来将边分配到不同的分区，从而保证顶点的备份数不大于2 \* sqrt(numParts)的限制。这里numParts表示分区数。 这个方法的实现分两种情况，即分区数能完全开方和不能完全开方两种情况。当分区数能完全开方时，采用第一种方式，当分区数不能完全开方时，采用下面的方法。这个方法的最后一列允许拥有不同的行数。

# Graphx中的vertices、edges以及triplets

## 1 vertices

在GraphX中，vertices对应着名称为VertexRDD的RDD。这个RDD有顶点id和顶点属性两个成员变量。

**abstract class** VertexRDD[VD](  
 sc: SparkContext,  
 deps: Seq[Dependency[\_]]) **extends** RDD[(VertexId, VD)](sc, deps)

从源码中我们可以看到，VertexRDD继承自RDD[(VertexId, VD)]，这里VertexId表示顶点id，VD表示顶点所带的属性的类别。这从另一个角度也说明VertexRDD拥有顶点id和顶点属性。

## 2 edges

在GraphX中，edges对应着EdgeRDD。这个RDD拥有三个成员变量，分别是源顶点id、目标顶点id以及边属性。

**abstract class** EdgeRDD[ED](  
 sc: SparkContext,  
 deps: Seq[Dependency[\_]]) **extends** RDD[Edge[ED]](sc, deps)

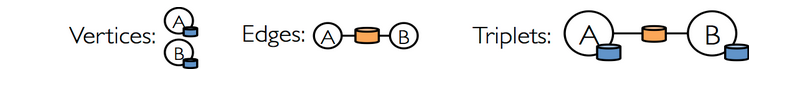
从源码中我们可以看到，EdgeRDD继承自RDD[Edge[ED]]，即类型为Edge[ED]的RDD。Edge[ED]在后文会讲到。

## 3 triplets

在GraphX中，triplets对应着EdgeTriplet。它是一个三元组视图，这个视图逻辑上将顶点和边的属性保存为一个RDD[EdgeTriplet[VD, ED]]。可以通过下面的Sql表达式表示这个三元视图的含义:

SELECT src.id, dst.id, src.attr, e.attr, dst.attr FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst ON e.srcId = src.Id AND e.dstId = dst.Id

同样，也可以通过下面图解的形式来表示它的含义：



# 图的构建

## 1 构建图的方法

构建图的入口方法有两种，分别是根据边构建和根据边的两个顶点构建。

**根据边构建图(Graph.fromEdges)**

**def** fromEdges[VD: ClassTag, ED: ClassTag](  
 edges: RDD[Edge[ED]],  
 defaultValue: VD,  
 edgeStorageLevel: StorageLevel = StorageLevel.*MEMORY\_ONLY*,  
 vertexStorageLevel: StorageLevel = StorageLevel.*MEMORY\_ONLY*): Graph[VD, ED] = {  
 *GraphImpl*(edges, defaultValue, edgeStorageLevel, vertexStorageLevel)  
}

**根据边的两个顶点数据构建(Graph.fromEdgeTuples)**

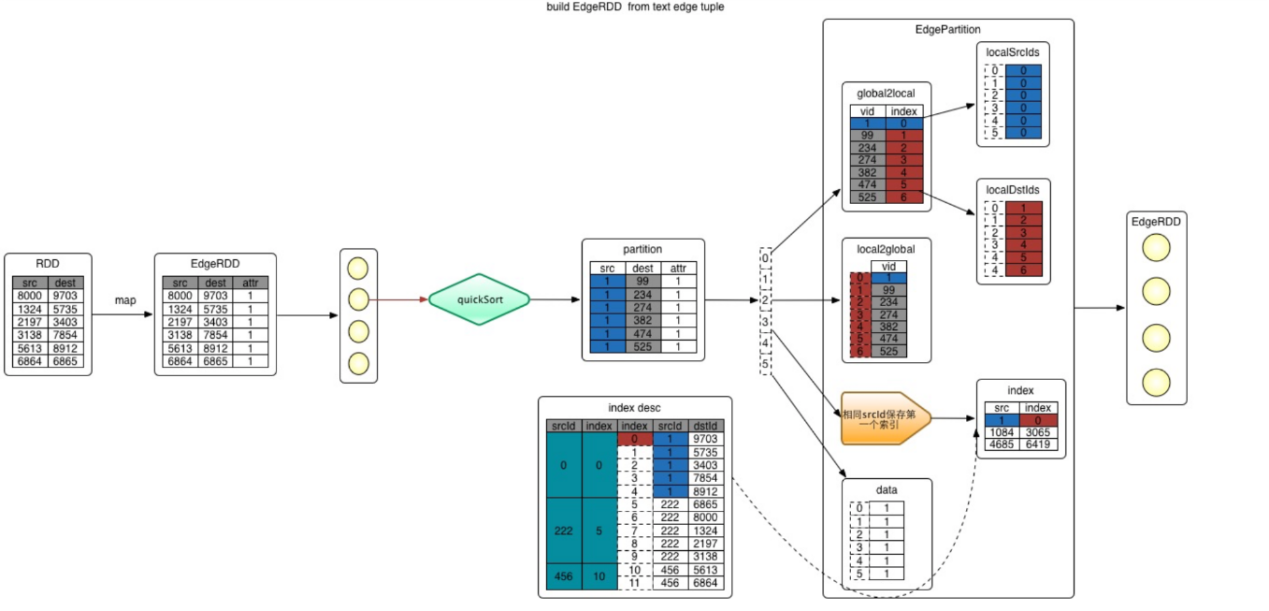
**def** fromEdgeTuples[VD: ClassTag](  
 rawEdges: RDD[(VertexId, VertexId)],  
 defaultValue: VD,  
 uniqueEdges: Option[PartitionStrategy] = None,  
 edgeStorageLevel: StorageLevel = StorageLevel.*MEMORY\_ONLY*,  
 vertexStorageLevel: StorageLevel = StorageLevel.*MEMORY\_ONLY*): Graph[VD, Int] =  
{  
 **val** edges = rawEdges.map(p => *Edge*(p.\_1, p.\_2, 1))  
 **val** graph = *GraphImpl*(edges, defaultValue, edgeStorageLevel, vertexStorageLevel)  
 uniqueEdges **match** {  
 **case** *Some*(p) => graph.partitionBy(p).groupEdges((a, b) => a + b)  
 **case** None => graph  
 }  
}

从上面的代码我们知道，不管是根据边构建图还是根据边的两个顶点数据构建，最终都是使用GraphImpl来构建的，即调用了GraphImpl的apply方法。

## 2 构建图的过程

 构建图的过程很简单，分为三步，它们分别是构建边EdgeRDD、构建顶点VertexRDD、生成Graph对象。下面分别介绍这三个步骤。

2.1 构建边的EdgeRDD



* **1** 从文件中加载信息，转换成tuple的形式,即(srcId, dstId)



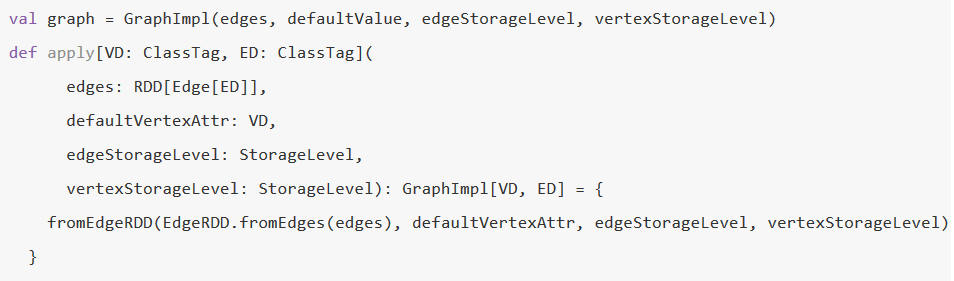
* **2** 入口，调用Graph.fromEdgeTuples(rawEdgesRdd)

源数据为分割的两个点ID，把源数据映射成Edge(srcId, dstId, attr)对象, attr默认为1。这样元数据就构建成了RDD[Edge[ED]],如下面的代码



* **3** 将RDD[Edge[ED]]进一步转化成EdgeRDDImpl[ED, VD]

第二步构建完RDD[Edge[ED]]之后，GraphX通过调用GraphImpl的apply方法来构建Graph。



在apply调用fromEdgeRDD之前，代码会调用EdgeRDD.fromEdges(edges)将RDD[Edge[ED]]转化成EdgeRDDImpl[ED, VD]。

## 3 构建图的点集合（例）

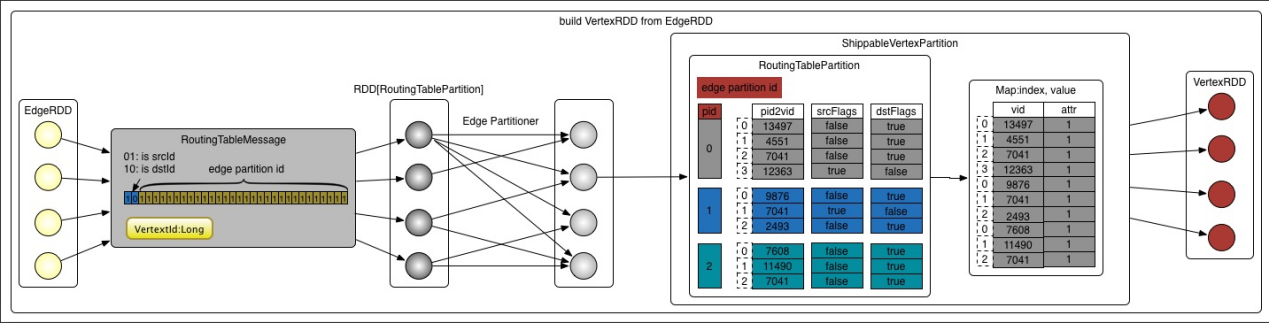
// 读取本地文件  
**val** conf = **new** SparkConf().setAppName(**this**.getClass.getName).setMaster("local[\*]")  
 // 采用Kryo序列化方式  
 .set("spark.serializer","org.apache.spark.serializer.KryoSerializer")  
**val** sc = **new** SparkContext(conf)  
  
// 构造出点的集合  
**val** rdd1 = sc.makeRDD(*Seq*(  
 (1L, ("李连杰", 56)),  
 (2L, ("王祖贤", 53)),  
 (6L, ("成龙", 62)),  
 (9L, ("周星驰", 57)),  
 (133L, ("周润发", 62)),  
 (138L, ("岳云鹏", 33)),  
 (16L, ("沈腾", 36)),  
 (21L, ("黄渤", 45)),  
 (44L, ("杨幂", 33)),  
 (158L, ("张云雷", 28)),  
 (5L, ("高圆圆", 42)),  
 (7L, ("张卫健", 55))  
))

## 4 构建图的边集合（例）

// 构造边的集合  
**val** edgeRDD: RDD[Edge[Int]] = sc.makeRDD(*Seq*(  
 *Edge*(1L, 133L,0),  
 *Edge*(2L, 133L,0),  
 *Edge*(6L, 133L,0),  
 *Edge*(9L, 133L,0),  
 *Edge*(6L, 138L,0),  
 *Edge*(21L, 138L,0),  
 *Edge*(44L, 138L,0),  
 *Edge*(16L, 138L,0),  
 *Edge*(5L, 158L,0),  
 *Edge*(7L, 158L,0)  
))

## 5 构建图

// 构造图  
**val** graph: Graph[(String, Int), Int] = *Graph*(rdd1,edgeRDD)  
// 取出顶点 (6,1),(2,1)  
**val** common = graph.connectedComponents().vertices  
//common.foreach(println)  
common.join(rdd1).map{  
 **case** (userId,(cmId,(name,age))) =>(cmId,*List*((name,age)))  
}.reduceByKey(\_++\_).foreach(*println*)



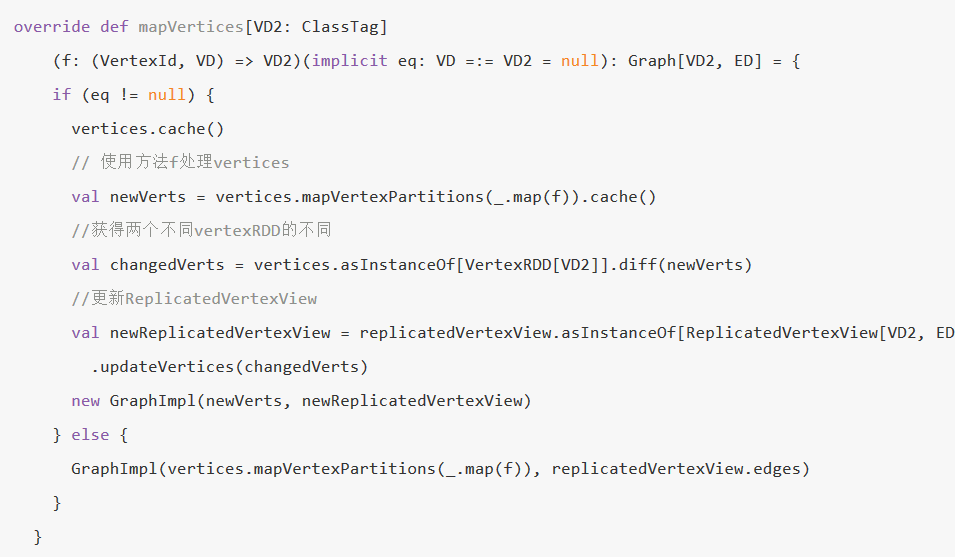
# GraphX的图运算操作

## 1 转换操作

GraphX中的转换操作主要有mapVertices,mapEdges和mapTriplets三个，它们在Graph文件中定义，在GraphImpl文件中实现。下面分别介绍这三个方法。

### 1 mapVertices

mapVertices用来更新顶点属性。从图的构建那章我们知道，顶点属性保存在边分区中，所以我们需要改变的是边分区中的属性。



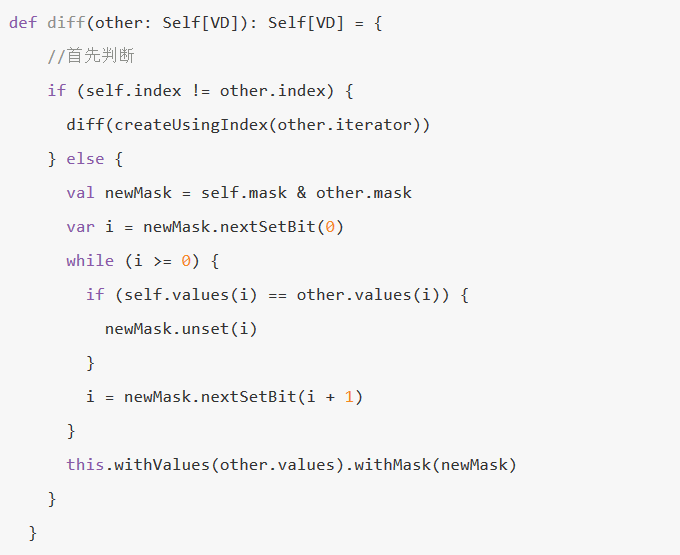
上面的代码中，当VD和VD2类型相同时，我们可以重用没有发生变化的点，否则需要重新创建所有的点。我们分析VD和VD2相同的情况，分四步处理。

1 使用方法f处理vertices,获得新的VertexRDD

2 使用在VertexRDD中定义的diff方法求出新VertexRDD和源VertexRDD的不同

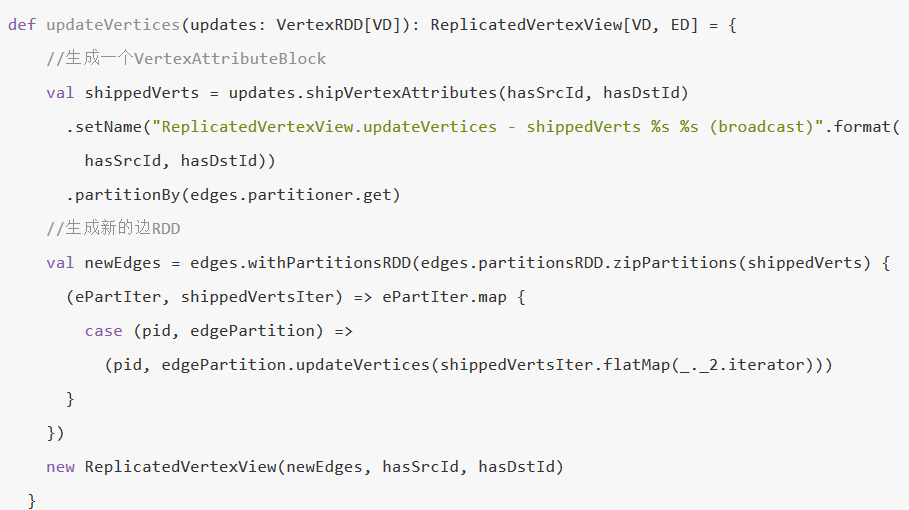


这个方法首先处理新生成的VertexRDD的分区，如果它的分区和源VertexRDD的分区一致，那么直接取出它的partitionsRDD,否则重新分区后取出它的partitionsRDD。 针对新旧两个VertexRDD的所有分区，调用VertexPartitionBaseOps中的diff方法求得分区的不同。



该方法隐藏两个VertexRDD中相同的顶点信息，得到一个新的VertexRDD

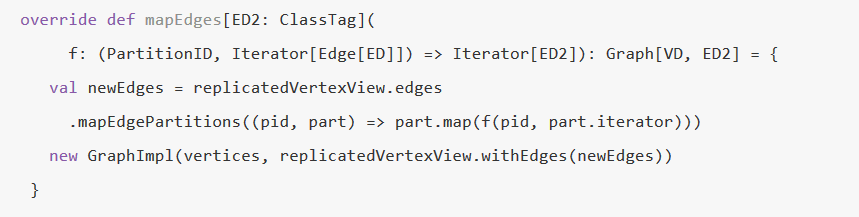
**3** 更新ReplicatedVertexView



updateVertices方法返回一个新的ReplicatedVertexView,它更新了边分区中包含的顶点属性。我们看看它的实现过程。首先看shipVertexAttributes方法的调用。 调用shipVertexAttributes方法会生成一个VertexAttributeBlock，VertexAttributeBlock包含当前分区的顶点属性，这些属性可以在特定的边分区使用。

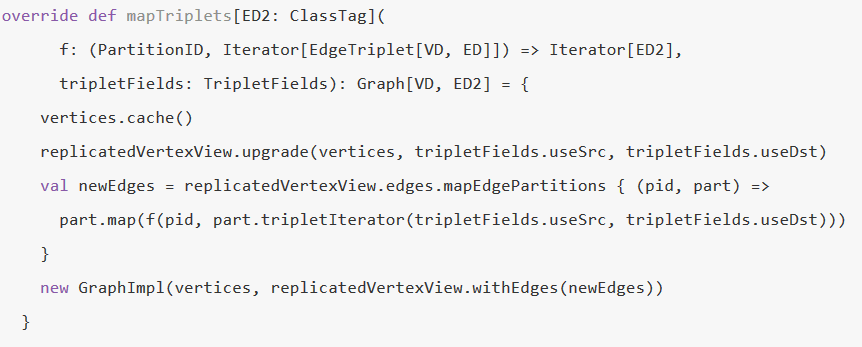
### 2 mapEdges

mapEdges用来更新边属性。



相比于mapVertices，mapEdges显然要简单得多，它只需要根据方法f生成新的EdgeRDD,然后再初始化即可。

### 3 mapTriplets：用来更新边属性



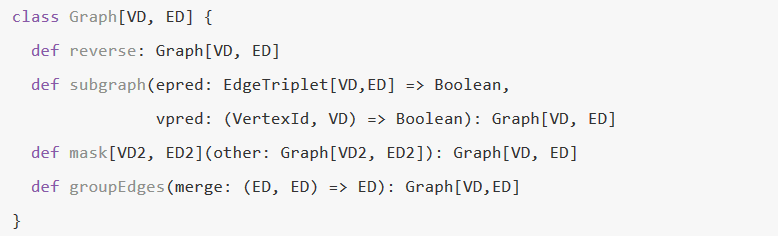
这段代码中，replicatedVertexView调用upgrade方法修改当前的ReplicatedVertexView，使调用者可以访问到指定级别的边信息（如仅仅可以读源顶点的属性）。



最后，用f处理边，生成新的RDD，最后用新的数据初始化图。

## 2 结构操作

当前的GraphX仅仅支持一组简单的常用结构性操作。下面是基本的结构性操作列表。

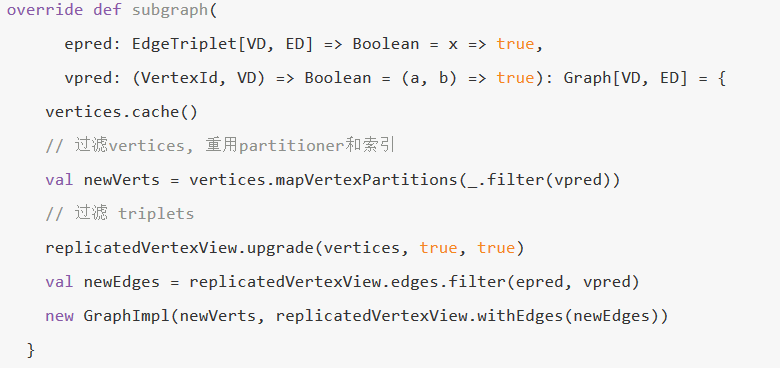


### 1 reverse reverse操作返回一个新的图，这个图的边的方向都是反转的。例如，这个操作可以用来计算反转的PageRank。因为反转操作没有修改顶点或者边的属性或者改变边的数量，所以我们可以 在不移动或者复制数据的情况下有效地实现它



### 2 subgraph

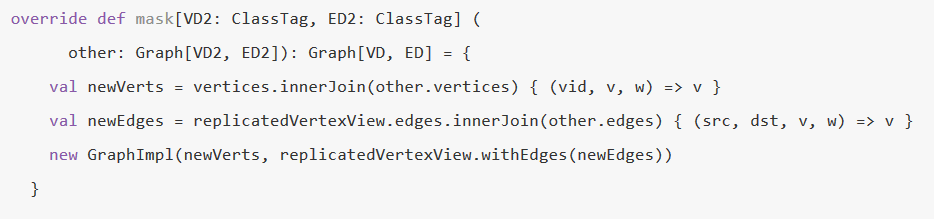
subgraph操作利用顶点和边的判断式（predicates），返回的图仅仅包含满足顶点判断式的顶点、满足边判断式的边以及满足顶点判断式的triple。subgraph操作可以用于很多场景，如获取 感兴趣的顶点和边组成的图或者获取清除断开连接后的图。



该代码显示，subgraph方法的实现分两步：先过滤VertexRDD，然后再过滤EdgeRDD

### 3 mask

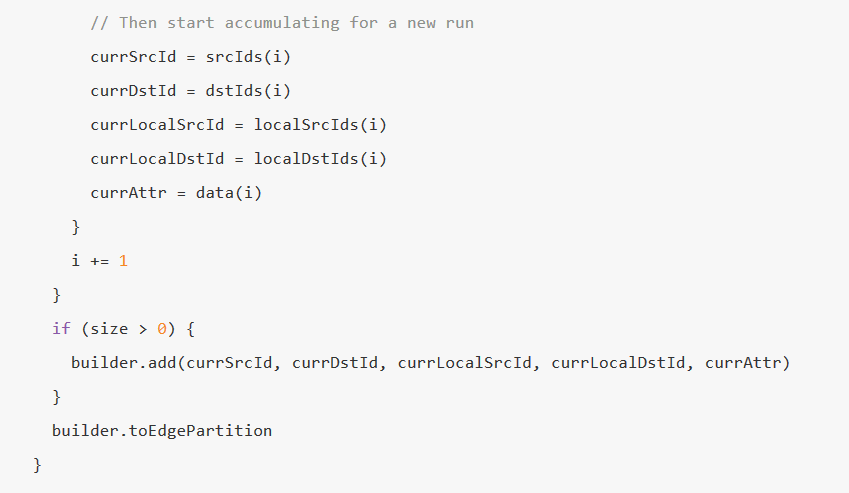
mask操作构造一个子图，这个子图包含输入图中包含的顶点和边。它的实现很简单，顶点和边均做inner join操作即可。这个操作可以和subgraph操作相结合，基于另外一个相关图的特征去约束一个图。



### 4 groupEdges

groupEdges操作合并多重图中的并行边(如顶点对之间重复的边)。在大量的应用程序中，并行的边可以合并（它们的权重合并）为一条边从而降低图的大小。





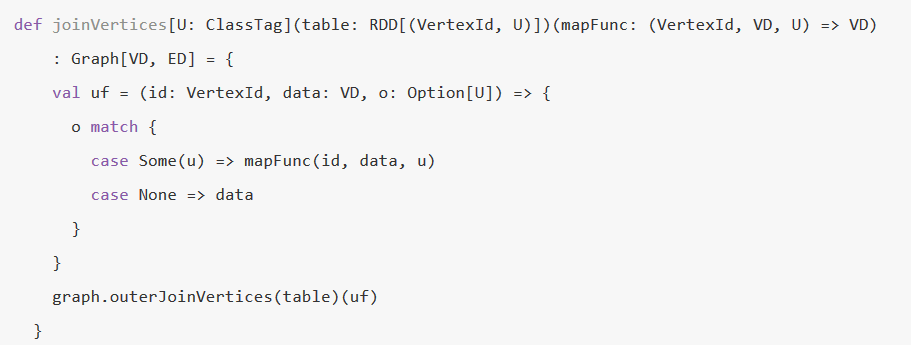
## 3 关联操作

在许多情况下，有必要将外部数据加入到图中。例如，我们可能有额外的用户属性需要合并到已有的图中或者我们可能想从一个图中取出顶点特征加入到另外一个图中。这些任务可以用join操作完成。 主要的join操作如下所示。



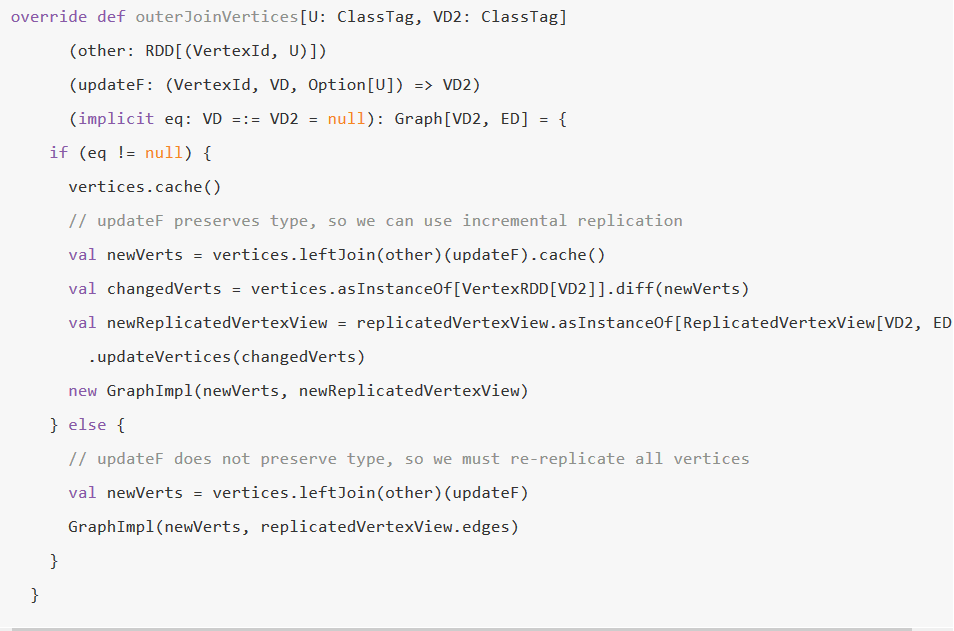
joinVertices操作join输入RDD和顶点，返回一个新的带有顶点特征的图。这些特征是通过在连接顶点的结果上使用用户定义的map函数获得的。没有匹配的顶点保留其原始值。 下面详细地来分析这两个函数。

### **1 joinVertices**



我们可以看到，joinVertices的实现是通过outerJoinVertices来实现的。这是因为join本来就是outer join的一种特例。

### **2 outerJoinVertices**



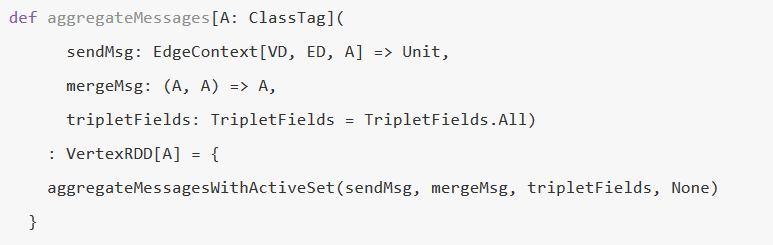
通过以上的代码我们可以看到，如果updateF不改变类型，我们只需要创建改变的顶点即可，否则我们要重新创建所有的顶点。

## 4 聚合操作

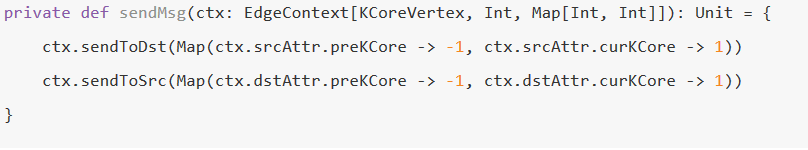
 GraphX中提供的聚合操作有aggregateMessages、collectNeighborIds和collectNeighbors三个，其中aggregateMessages在GraphImpl中实现，collectNeighborIds和collectNeighbors在 GraphOps中实现。下面分别介绍这几个方法。

### 1 aggregateMessages

aggregateMessages是GraphX最重要的API，用于替换mapReduceTriplets。目前mapReduceTriplets最终也是通过aggregateMessages来实现的。它主要功能是向邻边发消息，合并邻边收到的消息，返回messageRDD。



该接口有三个参数，分别为发消息函数，合并消息函数以及发消息的方向。

sendMsg： 发消息函数  


mergeMsg：合并消息函数

该函数用于在Map阶段每个edge分区中每个点收到的消息合并，并且它还用于reduce阶段，合并不同分区的消息。合并vertexId相同的消息。

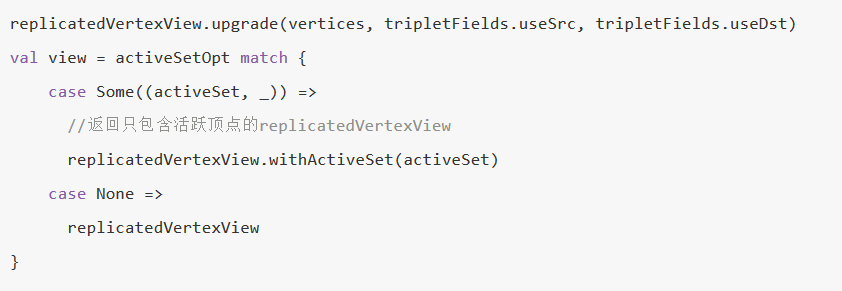
tripletFields：定义发消息的方向

1.1aggregateMessages处理流程

aggregateMessages方法分为Map和Reduce两个阶段

**Map阶段**

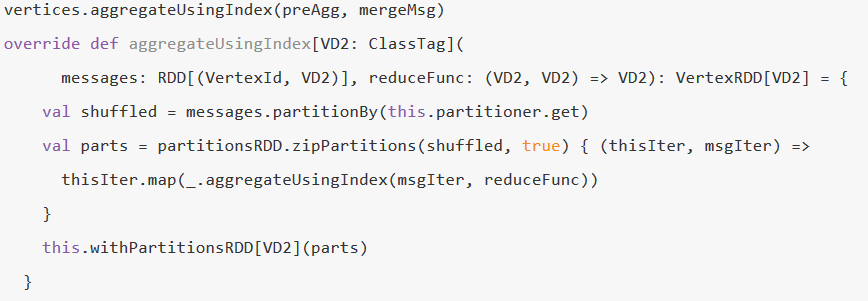
从入口函数进入aggregateMessagesWithActiveSet函数，该函数首先使用VertexRDD[VD]更新replicatedVertexView, 只更新其中vertexRDD中attr对象。如[构建图](https://endymecy.gitbooks.io/spark-graphx-source-analysis/content/build-graph.html)中介绍的， replicatedVertexView是点和边的视图，点的属性有变化，要更新边中包含的点的attr。



程序然后会对replicatedVertexView的edgeRDD做mapPartitions操作，所有的操作都在每个边分区的迭代中完成，如下面的代码：



**Reduce阶段**



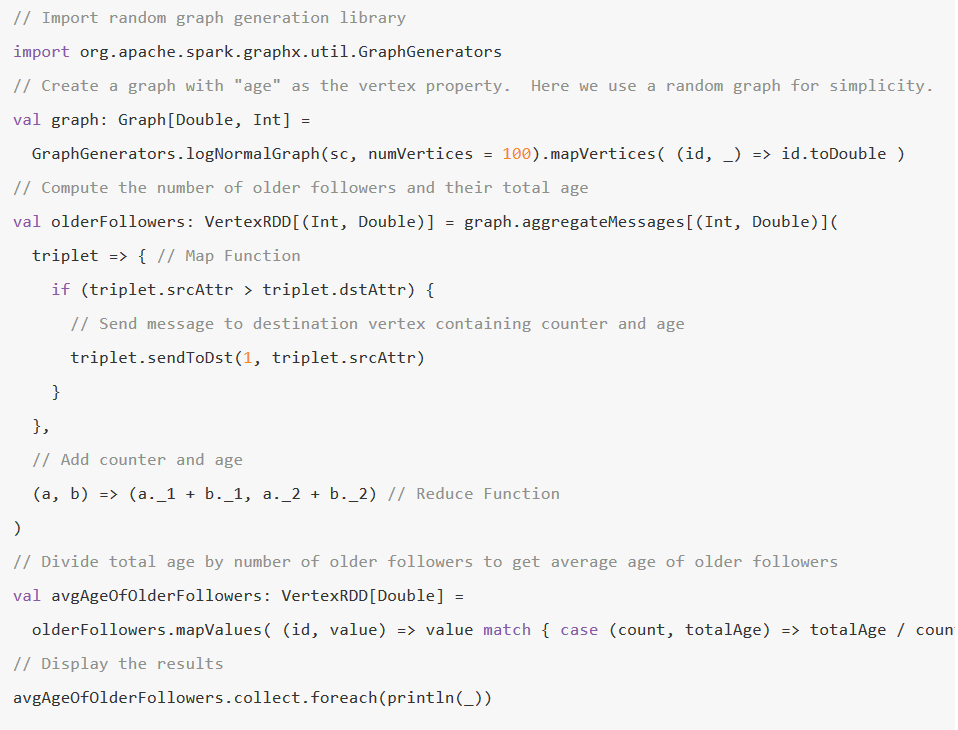
上面的代码通过两步实现。

1 对messages重新分区，分区器使用VertexRDD的partitioner。然后使用zipPartitions合并两个分区。

2 对等合并attr, 聚合函数使用传入的mergeMsg函数

**举例：**

下面的例子计算比用户年龄大的追随者（即followers）的平均年龄。

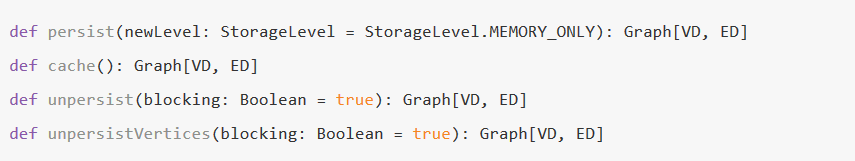


## 5 缓存操作

在Spark中，RDD默认是不缓存的。为了避免重复计算，当需要多次利用它们时，我们必须显示地缓存它们。GraphX中的图也有相同的方式。当利用到图多次时，确保首先访问Graph.cache()方法。

  在迭代计算中，为了获得最佳的性能，不缓存可能是必须的。默认情况下，缓存的RDD和图会一直保留在内存中直到因为内存压力迫使它们以LRU的顺序删除。对于迭代计算，先前的迭代的中间结果将填充到缓存 中。虽然它们最终会被删除，但是保存在内存中的不需要的数据将会减慢垃圾回收。只有中间结果不需要，不缓存它们是更高效的。然而，因为图是由多个RDD组成的，正确的不持久化它们是困难的。对于迭代计算，我们建议使用Pregel API，它可以正确的不持久化中间结果。

  GraphX中的缓存操作有cache,persist,unpersist和unpersistVertices。它们的接口分别是：



## 6 程序代码

import org.apache.log4j.{Level, Logger}

import org.apache.spark.{SparkContext, SparkConf}

import org.apache.spark.graphx.\_

import org.apache.spark.rdd.RDD

object GraphXExample {

  def main(args: Array[String]) {

    //屏蔽日志

    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

    Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

    //设置运行环境

    val conf = new SparkConf().setAppName("SimpleGraphX").setMaster("local")

    val sc = new SparkContext(conf)

    //设置顶点和边，注意顶点和边都是用元组定义的Array

    //顶点的数据类型是VD:(String,Int)

    val vertexArray = Array(

      (1L, ("Alice", 28)),

      (2L, ("Bob", 27)),

      (3L, ("Charlie", 65)),

      (4L, ("David", 42)),

      (5L, ("Ed", 55)),

      (6L, ("Fran", 50))

    )

    //边的数据类型ED:Int

    val edgeArray = Array(

      Edge(2L, 1L, 7),

      Edge(2L, 4L, 2),

      Edge(3L, 2L, 4),

      Edge(3L, 6L, 3),

      Edge(4L, 1L, 1),

      Edge(5L, 2L, 2),

      Edge(5L, 3L, 8),

      Edge(5L, 6L, 3)

    )

    //构造vertexRDD和edgeRDD

    val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)

    val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)

    //构造图Graph[VD,ED]

    val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  图的属性    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*         println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("属性演示")

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("找出图中年龄大于30的顶点：")

    graph.vertices.filter { case (id, (name, age)) => age > 30}.collect.foreach {

      case (id, (name, age)) => println(s"$name is $age")

    }

    //边操作：找出图中属性大于5的边

    println("找出图中属性大于5的边：")

graph.edges.filter(e => e.attr > 5).collect.foreach(e => println(s"${e.srcId} to ${e.dstId} att ${e.attr}"))

    println

    //triplets操作，((srcId, srcAttr), (dstId, dstAttr), attr)

    println("列出边属性>5的tripltes：")

    for (triplet <- graph.triplets.filter(t => t.attr > 5).collect) {

      println(s"${triplet.srcAttr.\_1} likes ${triplet.dstAttr.\_1}")

    }

    println

    //Degrees操作

    println("找出图中最大的出度、入度、度数：")

    def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {

      if (a.\_2 > b.\_2) a else b

    }

    println("max of outDegrees:" + graph.outDegrees.reduce(max) + " max of inDegrees:" + graph.inDegrees.reduce(max) + " max of Degrees:" + graph.degrees.reduce(max))

    println

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  转换操作    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("转换操作")

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("顶点的转换操作，顶点age + 10：")

    graph.mapVertices{ case (id, (name, age)) => (id, (name, age+10))}.vertices.collect.foreach(v => println(s"${v.\_2.\_1} is ${v.\_2.\_2}"))

    println

    println("边的转换操作，边的属性\*2：")

    graph.mapEdges(e=>e.attr\*2).edges.collect.foreach(e => println(s"${e.srcId} to ${e.dstId} att ${e.attr}"))

    println

      //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  结构操作    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("结构操作")

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("顶点年纪>30的子图：")

    val subGraph = graph.subgraph(vpred = (id, vd) => vd.\_2 >= 30)

    println("子图所有顶点：")

    subGraph.vertices.collect.foreach(v => println(s"${v.\_2.\_1} is ${v.\_2.\_2}"))

    println

    println("子图所有边：")

    subGraph.edges.collect.foreach(e => println(s"${e.srcId} to ${e.dstId} att ${e.attr}"))

    println

      //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  连接操作    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("连接操作")

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    val inDegrees: VertexRDD[Int] = graph.inDegrees

    case class User(name: String, age: Int, inDeg: Int, outDeg: Int)

    //创建一个新图，顶点VD的数据类型为User，并从graph做类型转换

    val initialUserGraph: Graph[User, Int] = graph.mapVertices { case (id, (name, age)) => User(name, age, 0, 0)}

    //initialUserGraph与inDegrees、outDegrees（RDD）进行连接，并修改initialUserGraph中inDeg值、outDeg值

    val userGraph = initialUserGraph.outerJoinVertices(initialUserGraph.inDegrees) {

      case (id, u, inDegOpt) => User(u.name, u.age, inDegOpt.getOrElse(0), u.outDeg)

    }.outerJoinVertices(initialUserGraph.outDegrees) {

      case (id, u, outDegOpt) => User(u.name, u.age, u.inDeg,outDegOpt.getOrElse(0))

    }

    println("连接图的属性：")

userGraph.vertices.collect.foreach(v => println(s"${v.\_2.name} inDeg: ${v.\_2.inDeg}  outDeg: ${v.\_2.outDeg}"))

    println

    println("出度和入读相同的人员：")

    userGraph.vertices.filter {

      case (id, u) => u.inDeg == u.outDeg

    }.collect.foreach {

      case (id, property) => println(property.name)

    }

    println

      //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  聚合操作    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("聚合操作")

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("找出年纪最大的追求者：")

    val oldestFollower: VertexRDD[(String, Int)] = userGraph.mapReduceTriplets[(String, Int)](

      // 将源顶点的属性发送给目标顶点，map过程

      edge => Iterator((edge.dstId, (edge.srcAttr.name, edge.srcAttr.age))),

      // 得到最大追求者，reduce过程

      (a, b) => if (a.\_2 > b.\_2) a else b

    )

    userGraph.vertices.leftJoin(oldestFollower) { (id, user, optOldestFollower) =>

      optOldestFollower match {

        case None => s"${user.name} does not have any followers."

        case Some((name, age)) => s"${name} is the oldest follower of ${user.name}."

      }

    }.collect.foreach { case (id, str) => println(str)}

    println

     //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  实用操作    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("聚合操作")

    println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    println("找出5到各顶点的最短：")

    val sourceId: VertexId = 5L // 定义源点

    val initialGraph = graph.mapVertices((id, \_) => if (id == sourceId) 0.0 else Double.PositiveInfinity)

    val sssp = initialGraph.pregel(Double.PositiveInfinity)(

      (id, dist, newDist) => math.min(dist, newDist),

      triplet => {  // 计算权重

        if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {

          Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))

        } else {

          Iterator.empty

        }

      },

      (a,b) => math.min(a,b) // 最短距离

    )

    println(sssp.vertices.collect.mkString("\n"))

    sc.stop()

  }

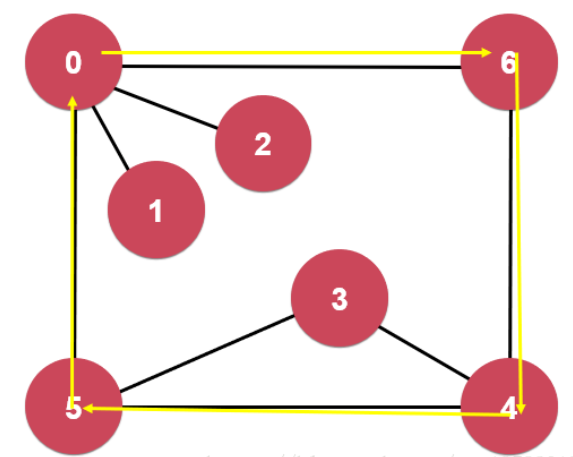
}

# 深度优先算法和广度优先算法

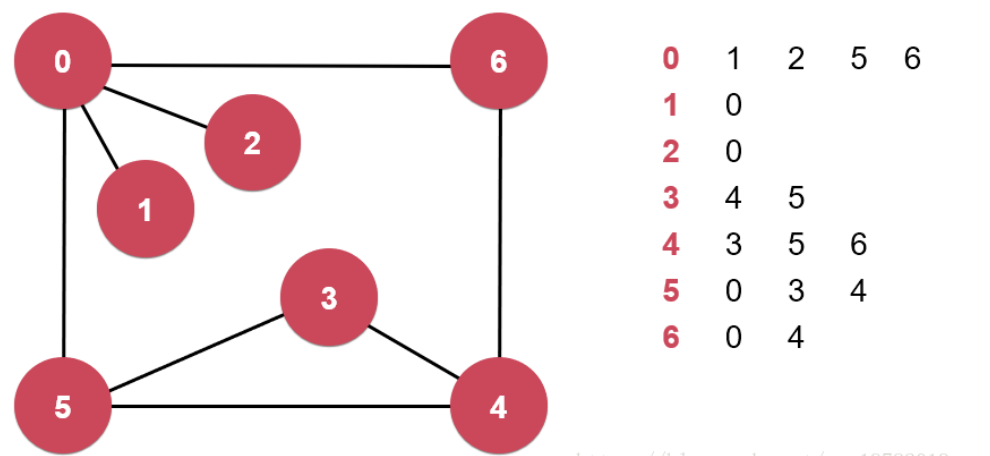
## 1 深度优先算法

深度优先搜索算法（英语：Depth-First-Search，DFS）是一种用于遍历或搜索树或图的算法。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点v的所在边都己被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。

先来个图，大家体会一下，有没有发现深度优先算法是转圈圈？这会一直将会导致深度搜索一直在“转圈圈”。



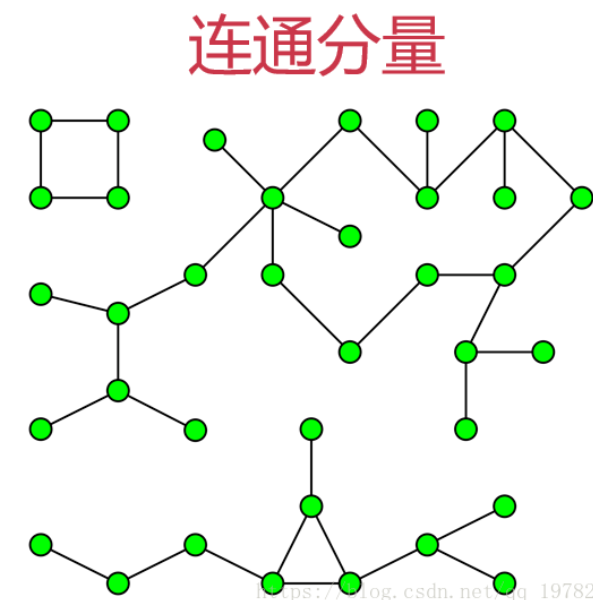
那么，该如何去解决这问题呢？答案当然是有的，既然图的深度搜索可能会发生“转圈圈”，那么我们所需要解决的问题就是如何改进深度搜索算法，使其能够停止下来。这很容易实现，我们只需要在使用额外的一个辅助数组visited[ ]即可。假设当前访问了某一个节点i，我们就标记其状态为已经被访问状态，在辅助数组中表示为visited[i]=true。这样，当深度搜索在搜索前，先参考visited[ ]数组，判断要访问的节点是否已经被访问过了，如果被访问过了则跳过该节点继续进行深度搜索。



首先，我们从0号节点进行进行搜索，查找辅助表visited发现visited[0]=false,即0号节点从未被搜索过，所以我们搜索完0号节点标记其visit状态为真，即修改辅助表visited【0】=true。接下来的任务是要遍历0号节点的所有的相邻节点，所以我们遍历1号节点，发现其visited状态为false，说明这个节点从未被遍历过，接下来我们就修改其visited状态为true,即visited[1]=true。之后，有些同学可能会想当然的遍历0号节点的第2个邻接点2号节点，这可是大错特错了！何为深度优先遍历算法，我认为其精髓就是这句话：尽可能深的搜索图的所有邻节点。（搜索条件：深度>广度）既然1号节点也存在着邻节点，那么我们就应该“钻牛角尖”，永不放弃，尽可能深的去搜索邻节点，所以搜索完1号节点，不能返回去搜索0号节点的其他邻节点，而是继续往深处去探索1号节点的邻节点。

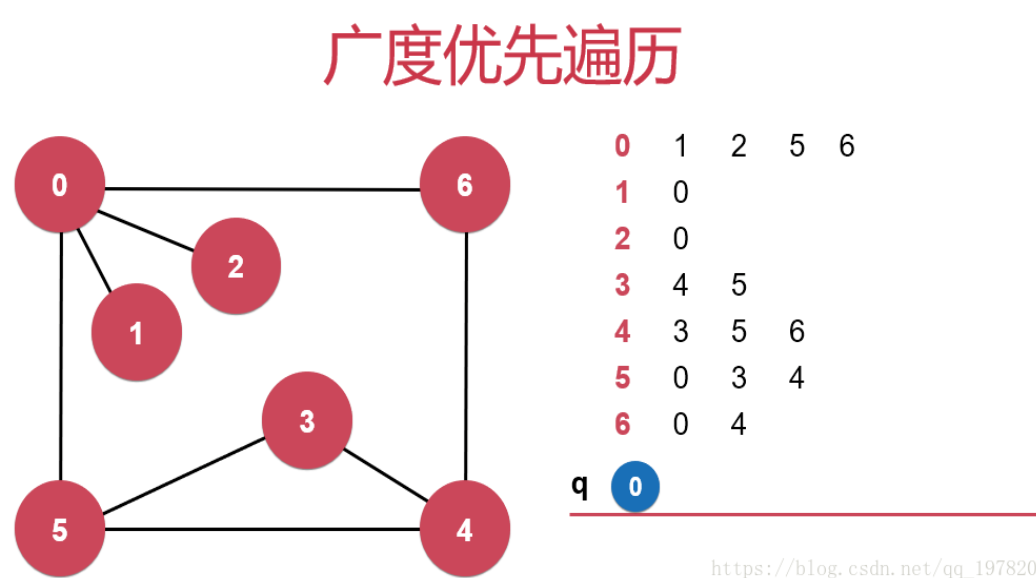
而1节点再往深处搜，其只有一个邻节点：0号节点。显然，0号节点已经被搜索过了，即1号节点的所有邻节点已经都被搜索过了，那么我们在1号节点就不能往下“钻”了。已经满足了“深“这个条件，接下来就应该满足“广度”这个条件了。””然后我们返回上一层的0号节点，再来看0号节点其他的邻节点，2号节点未被访问，则访问2号节点，访问完后发现2号节点也存在邻节点，则应该先满足“深”的条件，继续对2号节点的邻节点搜索，发现2号节点的所有邻节点（其实只有一个0号节点已经被访问过了），那么我们完成了“广度”的条件。在返回上一层继续......总而言之，对于每个节点来说，能深度往下搜下去就尽量往深处搜，实在深不下去了，在返回上一层完成“广”的搜索，就这样一直递归下去，图中所有的节点一定会全部被访问到的。

**联通分量的计算**

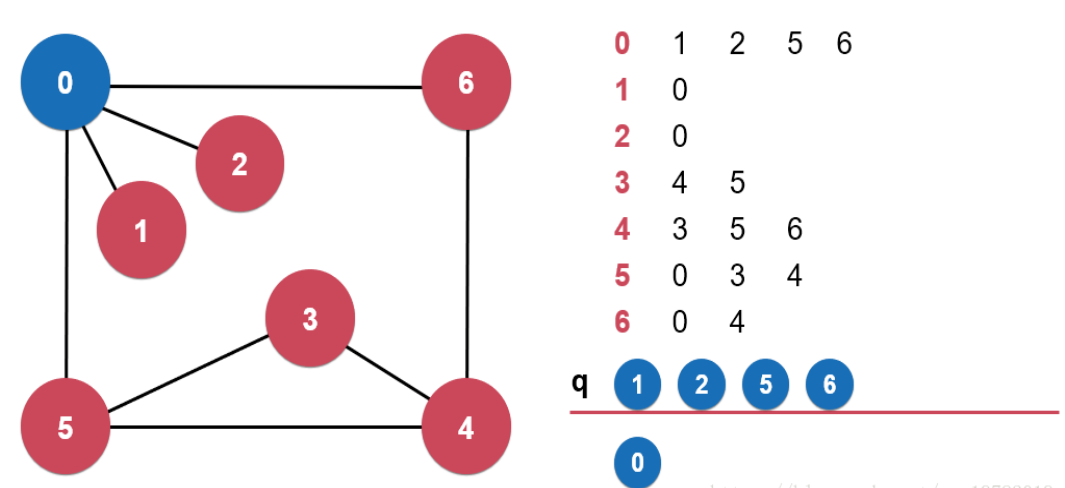


其中，在一个图中，存在3个部分，在图论中，这三个小的部分我们就称作为3个联通分量。联通分量和联通分量之间没有任何边相连。那么如何通过深度遍历算法来计算一个图中的联通分量个数呢？首先，先定义一个ccount变量用来记录联通分量的数目和联通分量所属的序号，接下来我们从左上角的第一个点开始进行深度遍历，我们会发现，遍历4个点之后将会结束遍历，这说明这4个点属于同一个联通分量之中，因为是第一个联通分量，所以该联通分量的序号（该联通分量所有节点的序号）即为cccount的值：为0，既然找到了一个联通分量，则ccount++变成1,。接下来，我们在从所有的点中找出下一个未被访问过的点，在执行一深度遍历算法，直到算法停止，说明又找到一个联通分量,，这这个联通分量的序号为1（该联通分量所有节点的序号）。接下来又继续找一个未被访问过的点进行深度遍历操作，直到停止，联通分量数加一，联通分量序号加一.......。有了上面的基础工作，则判断量节点是否处于同一个联通分量就很简单了，只需要比较两个节点的分别对应的联通序号是不是一样的，就能得出两个节点是不是处于同一个联通分量中。

## 2 广度优先算法



首先，上图左边的是一个简单的无向图，右上方是一个表格，描述了每个节点与之相连的所有邻节点。右下方是一个队列q。还有新增一个ord数组用来计算节点与起始节点之间的距离。首先把起始节点（假设0为起始节点）压入队列中，设置ord[0]=0。标记0节点的visited状态为true，那么此时0节点就已经被遍历了，接下来不能立马让0节点出队列，因为0节点还有一群“跟随”他的邻节点还没入队列呢！所谓的广度优先，就是尽量把节点与之相邻的所有临节点都遍历完再往“深”出去遍历，因此，0节点不仅不能立马出队，还要把他的所有的“邻居节点”都拉进队列里来，他才能安心的离开队列。因为他的邻节点不仅仅是他一个人的邻节点，也有可能是其他节点的邻节点，因此，有可能其他的邻节点已经先把他的部分邻居节点拉入队列中了，所以“拉邻居节点入队列”要排除已经入队列的“邻居节点”，那又如何判断一个节点是否已经进入队列了呢？别忘记了！我们还有一个visited数组记录着呢，任意一个节点只要进入了队列，其visited状态就会置为true。所以，按照这个思路，0节点在出队列之前要把他所有的“邻居节点”中符合条件的节点一起拉入队列中，其中0节点的邻居节点是1,2,5,6，这四个节点的visited状态都为false，所以把他们拉进队伍，并把visited状态置为true，把from[1],from[2],from[5],from[6]都置为0，表示这4个节点都是从0节点遍历过来的。最后，设置ord[1]=ord[0]+1,ord[2]=ord[0]+1,ord[5]=ord[0]+1,ord[6]=ord[0]+1。这样，0节点才完成了他应该完成的所有任务，可以允许出列了



接下来，队列中的首节点是1号节点，轮到1号节点完成任务了，visited[0]=true,但0号节点已经入队过了，所以没有邻居节点可以被1节点拉进队列了，没有新的节点入队，因此此时不用维护from，ord这两个表了，1节点可以允许出列了。

只要队列不为空，那么队列就可以按照上面的规则一直执行下去，直到队列为空，此时图中所有的节点就都已经入队，并完成了相应的数据维护工作，图的“广度优先遍历”就已经完成了。

# PageRank

一：算法介绍

PageRank是Google专有的算法，用于衡量特定网页相对于搜索引擎索引中的其他网页而言的重要程度。

一个页面的“得票数”由所有链向它的页面的重要性来决定，到一个页面的超链接相当于对该页投一票。一个页面的PageRank是由所有链向它的页面（“链入页面”）的重要性经过递归算法得到的。一个有较多链入的页面会有较高的得分，相反如果一个页面没有任何链入页面，那么它没有得分。

二：源码分析

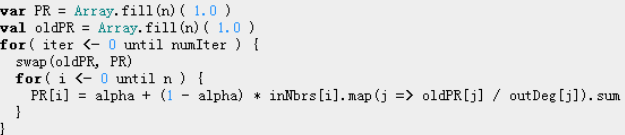
I：PregelPageRank

文件位置：spark\graphx\src\main\scala\org\apache\spark\graphx\lib\PageRank.scala

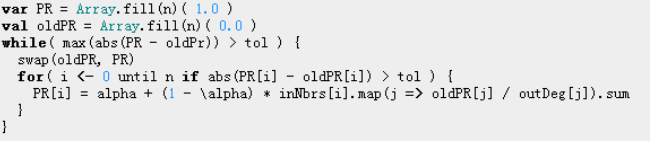
1.1：代码简介

该PageRank模型提供了两种调用方式：

第一种：（静态）在调用时提供一个参数number，用于指定迭代次数，即无论结果如何，该算法在迭代number次后停止计算，返回图结果。



第二种：（动态）在调用时提供一个参数tol，用于指定前后两次迭代的结果差值应小于tol，以达到最终收敛的效果时才停止计算，返回图结果。



这是GraphX提供的用Pregel的模型改进后产生的图算法，通常我们在进行使用PageRank的代码编写时并不涉及去改动这份源码，而是直接调用：



1.2：源码解析

首先解释下列代码中run()的几个参数:

graph:进行PageRank计算的图模型

numIter:固定的PageRank计算的迭代次数

resetProb:随机重置的概率,通常都是0.15

Graph:返回值,以图的形式包括最终的顶点值(pagerank值)和边值(权重值),进而得到最终的排名结果

def run[VD: ClassTag, ED: ClassTag](

graph: Graph[VD, ED],numIter: Int, resetProb: Double = 0.15): Graph[Double, Double] =

{

//下列这段代码用于初始化PageRank图模型，具体内容是赋予每个顶点属性为值1.0，赋予每条边属性为值“1/该边的出发顶点的出度数”。

val pagerankGraph: Graph[Double, Double] = graph

//将每个顶点进行连接（度的传递）得到顶点属性值为出度数

.outerJoinVertices(graph.outDegrees) { (vid, vdata, deg) =>deg.getOrElse(0) }

//通过顶点的出度数为每条边设置权重值；这里是Triplet型的迭代器不停地执行一个map函数来遍历得到每条边的权重值，值为1.0/顶点出度数

.mapTriplets( e => 1.0 / e.srcAttr )

//设置每个顶点的初始属性值为1.0

.mapVertices( (id,attr) => 1.0 )

.cache() //将完成初始化的图缓存操作

//以下将定义三个所需函数来完成GraphX对PageRank的算法实现

//用作 Pregel的message

//第一个函数用于返回一个考虑“随机事件”发生后的计算结果

def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double=

resetProb + (1.0 - resetProb) \* msgSum

//第二个函数用于得到一个迭代器，里面包含了两个信息：该边的目的ID、该边的源属性值和权重的乘积（该边传递的实际PR值）

def sendMessage(edge: EdgeTriplet[Double, Double]) =

Iterator((edge.dstId, edge.srcAttr\* edge.attr))

//第三个函数用于将顶点属性值和传递的值进行累加

def messageCombiner(a: Double, b: Double): Double = a + b

//在该PageRank模型中每个顶点接受到的初始传递信息都是0.0

val initialMessage = 0.0

// 执行 pregel 模型算法（固定的迭代次数）

Pregel(pagerankGraph, initialMessage, numIter, activeDirection= EdgeDirection.Out)(

vertexProgram,sendMessage, messageCombiner)

}

至此第一种（静态）PageRank模型计算结束

以下是第二种（动态）PageRank模型计算，相同代码就不再累赘解释

初始化参数和上面不同的是少了numIter（迭代次数），多了tol（比较两次迭代的结果差）

Pregel相当于图计算的引擎，用于图计算的大框架（对顶点的消息计算、消息发送、消息合并），它是图迭代的执行者。lib中的所有算法模型最后都会调用Pregel。

GraohOps 则相当于一个可以快速调用方法的清单，里面给出了很多类或方法的入口；在此例中的PageRank（）、Pregel（）方法都是从这启动的。

II：PageRank简例

2.1：代码简介

这段代码是官方GraphX guide 提供的，由于使用的就是spark包中的自带数据，所以用于测试非常简单。另外，这个例子确实非常简短，因为自带数据量非常非常小（但这并不意味我们不可以修改原始数据）！

下面是全部的源码，除开注释可看出代码非常的少，那是因为GraphX为了让开发者方便直接提供了多个算法模型（如上述的PageRank.scala文件），只需代入数据直接调用就行。

2.2：源码解析

// 从特定的边列表文件中读取数据生成图框架

val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")

// 用上面的图框架来调用pageRank(动态)算法

//特别注意：静态调用的方法名是staticPageRank(Int)

// vertices将返回顶点属性

val ranks = graph.pageRank(0.0001).vertices

// 将上面得到的ranks（顶点属性）和用户进行关系连接

// 首先也是读取一个包含了用户信息的文件，然后调用了一个map函数，即将文件里的每行数据按 ”,” 切开并返回存储了处理后数据的RDD

val users = sc.textFile("graphx/data/users.txt").map { line =>

val fields = line.split(",")

(fields(0).toLong, fields(1))

}

// 这里具体实现了将ranks和用户列表一一对应起来

// 从map函数的内容可以看出是按id来进行连接，但返回的结果只含用户名和它的相应rank值

val ranksByUsername = users.join(ranks).map {

case (id, (username, rank)) => (username, rank)

}

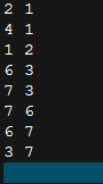
// 收集上面RDD里的数据并打印出来

println(ranksByUsername.collect().mkString("\n"))

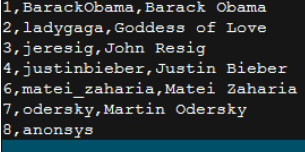
2.3：输入数据

在spark-1.0.1\graphx\data 目录下有两份数据文件：

followers.txt



Users.txt



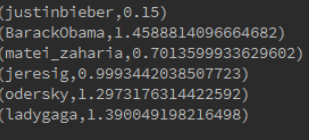
这两份数据的数据量非常小，但是作为测试可以更好的分析其算法原理。

2.4：输出结果

㈠动态调用



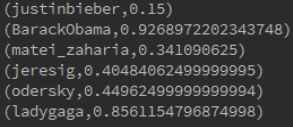
结果1：



下面变更下参数（tol值）



结果2：

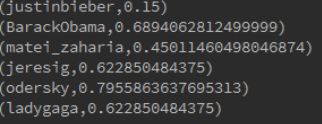


可以看出和上面的结果相差较大，事实上，参数值越小得到的结果越有说服力。

⑵静态调用



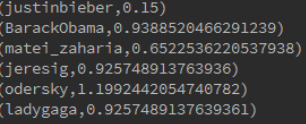
结果1：



增加迭代次数：



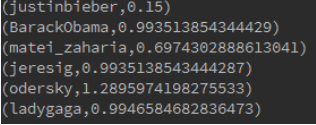
结果2：



结果依然不够准确，再次增加迭代次数：



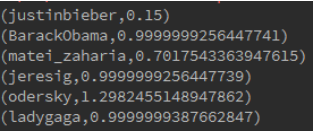
结果3：



结果依然不够准确，再次增加迭代次数：



结果4：



可以看出对于该数据，用静态算法很难得到准确的结果

三：应用场景

Google的网页排名并非如此单纯的PageRank算法，它考虑的综合因素至少有10点以上。

但该算法仍然可以为排名计算（网页排名、用户排名等）提供其中一个可靠的依据。在目前，我们应该不会去改动模型代码，而是合理的安排需要处理的数据。该算法处理的场景原型很容易看出：有相互联系的事物网中，评选出最受“欢迎”的事物。什么叫受欢迎？——被其他事物选择、依赖、信任、消费等等。