



Community Experience Distilled

Apache Spark Graph Processing

Build, process, and analyze large-scale graphs with Spark

*Foreword by Denny Lee, Technology Evangelist, Databricks
Advisor, WearHacks*

Rindra Ramamonjison

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

Apache Spark Graph Processing

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2015

Production reference: 1040915

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-180-5

www.packtpub.com

译序

嗯，继上一本《Learning Spark》之后，又一个译序，脸皮又厚了一些了。

上一本《Learning Spark》计划是翻译第三章到第九章，实际上完成到第八章。我在译序中写道：“因为没有讲我最喜欢的 GraphX，所以这几章我也不想再继续翻了”。后来，看到了这本《Apache Spark Graph Processing》，全书总共有七章。看标题就知道我蛮喜欢的哈，于是给自己订了个小目标，就是翻译前四章后上传。已经忘了是哪天开始的，大概是过年那阵子吧，年后来北京出差都已经好几个月了，反正有好久啦。在来北京出差之前，就已经边看边翻译完了前三章。出差还是挺忙的，有一阵没一阵的，总算是把第四章看完了，写了下来。

不啰嗦了，跟上一本一样，翻译的时候，我尽量保持内容和原作所在的页码一致，方便各位对照原文纠错。没翻译的章节是第5章 创建自定义的图聚合操作，第6章 用 Pregel 进行图的并行迭代处理，以及第7章 学习图的结构。有兴趣的可以看看原书吧。或者我以后也会更新，不着急的可以关注下我的 blog。但是不保证哈~~~

另外，本书算是 Spark 的入门内容，看本书的应该也是初学者。推荐 Databricks 提供的一个压缩包：<http://training.databricks.com/workshop/usb.zip>。这是个 Spark 的练习环境，有代码样例，测试数据。只需要你有个单机的 Linux 环境，然后下载这个压缩包，解压后就能用了。

顺便也广告一下 Spark 的入门书籍《Learning Spark》的中文版 PDF 版本还可以在 CSDN 下载：http://download.csdn.net/detail/coding_hello/9161615。放上去以后，看到下载次数也有几百次了，还有人评论说翻得不错。我也还蛮开心的，哈哈~ 大家好才是真的好哈~

很明显英语不是我的母语，所以翻译的不正确请原谅，我只能说我尽力了，谢谢！当然，如果你愿意告诉我哪里错了，那真是太感谢了，好人一生平安~！

如果有啥要指教我的，可以 email: coding_hello@126.com
有更新的话，会放在我的 CSDN 博客: http://blog.csdn.net/coding_hello/article/
或者直接在 CSDN 资源里搜书名吧，那里也会放一份方便查找下载的。

祝各位阅读愉快，欢迎交流！

coding
2016-07-31

1

初识 Spark 和 GraphX

Apache Spark 是处理大型分布式数据集的集群计算平台。在 Spark 中进行数据又快又简单，这归功于其优化的并行计算引擎和灵活而统一的 API。Spark 的核心抽象是基于**弹性分布式数据集(RDD)**的概念。通过扩展 MapReduce 框架，Spark 的核心 API 使得分析任务更易于编写。在核心 API 的顶层，Spark 提供了一组综合的高级库，可用于一些特定任务，如图计算和机器学习。其中，GraphX 就是 Spark 处理图并行计算的库。

本章中将通过构建一个社交网络和探索该网络中人与人之间的关系来介绍 Spark 和 GraphX。另外，你也将学到用 Scala Build Tool(SBT)来构建和运行 Spark 程序。在本站最后，你将了解如何：

- 成功安装 Spark 到你的计算机
- 用 Shark Shell 实验，回顾 Spark 的数据抽象
- 创建图并基于 RDD 和图操作来探索这些联系
- 用 SBT 构建和提交独立的 Spark 应用

下载和安装 Spark 1.4.1

接下来的章节，我们将整理 Spark 安装的详细过程。Spark 是构建于 Scala，运行在 **Java 虚拟机(JVM)**里。在安装 Spark 前，你应该先安装 **Java Development Kit(JDK)** 在你电脑上。

确定你安装的是 JDK，而不是 **Java Runtime Environment(JRE)**。你可以从 <http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html> 下载。

下一步，从 Spark 项目网站 <https://spark.apache.org/downloads.html> 下载最新发布的 Spark 版本。执行下面三步来安装 Spark 到你的电脑里：

1. 选择包类型：**Pre-built for Hadoop 2.6 and later**，然后选择下载类型：**Direct Download**。确定你选择的是 Hadoop 预编译版本，而不是源代码
2. 下载压缩的 TAR 文件 `spark-1.4.1-bin-hadoop2.6.tgz`，放到你电脑上的一个目录中。

3. 打开一个终端，进入到刚才的目录。用下面的命令解压 TAR 文件，重命名 Spark 根目录为 `spark-1.4.1`，然后列示安装的文件和子目录：

```
tar -xf spark-1.4.1-bin-hadoop2.6.tgz
mv spark-1.4.1-bin-hadoop2.6 spark-1.4.1
cd spark-1.4.1
ls
```

就是这样！现在你已经安装了 Spark 和它的库到你的电脑上了。注意在 `spark-1.4.1` 的主目录中的下列文件和子目录：

- `core`: 该目录包含了 Spark 核心组件和 API 的源代码
- `bin`: 该目录包含了一些可执行文件，用于提交和分发 Spark 应用程序或者在 Spark shell 中与 Spark 交互
- `graphx, mlib, sql 和 streaming`: 这些是 Spark 的库，提供了做不同类型的数据处理的统一的接口，即图处理，机器学习，查询和流处理。
- `examples`: 该目录包含了 demo 和 Spark 应用的示例。

创建快捷方式到 Spark 主目录和示例目录通常会很方便。在 Linux 或者 Mac 中，打开或创建 `~/.bash_profile` 文件，插入如下几行

```
export SPARKHOME="/[Where you put Spark]/spark-1.4.1/"
export SPARKSCALAEX="ls ../spark-
1.4.1/examples/src/main/scala/org/apache/spark/examples/"
```


要来个完整性检查，你可以输入一些 Scala 表达式或声明并求值。现在就在 shell 中输入一些命令吧：

```
scala> sc

res1: org.apache.spark.SparkContext = org.apache.spark.SparkContext@52e52233
scala> val myRDD = sc.parallelize(List(1,2,3,4,5))

myRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:12

scala> sc.textFile("README.md").filter(line => line contains "Spark").count()
res2: Long = 21
```

以下就是上述代码表达的内容：首先，我们显示了被定义为变量 `sc` 的 Spark Context，该变量在你启动 Spark shell 时被自动创建。SparkContext 是 Spark API 的入口点。其次，我们创建了名为 `myRDD` 的 RDD 对象，是通过对一个 5 个数字的 list 调用 `parallelize` 函数获得的。最后，我们加载了 README.md 文件到 RDD 对象，过滤出了包含单词“Spark”的行，最后对过滤后的 RDD 调用 `count` 动作来计算这些行数。



这里期待你已经熟悉基本的 RDD 变换和动作，如 `map`, `reduce` 和 `filter`。如果不熟，我建议你先学习那些。可以通过阅读编程向导 <https://spark.apache.org/docs/latest/programming-guide.html> 或阅读入门书籍，比如 Packt 出版的 Fast Data Processing with Spark 或 O'Reilly Media 出版的 Learning Spark¹。

即使你完全不了解 RDD 背后的机制也不要慌，下面的补习会帮你记住重点。RDD 是 Spark 中的核心数据抽象，用来表达大型分布式数据集可以被分区并且被跨集群的机器并行处理。Spark API 提供了一组统一的操作来变换和规约 RDD 中的数据。在这些抽象和操作上层，GraphX 库也提供了灵活的 API 允许我们创建图并轻松地处理之。

也许当你在 shell 中运行过之前那些命令，你就被那些 INFO 开头的长长的日志语句清单吓坏了。这里有个办法可以减少 shell 中 Spark 输出的信息量。

¹ 译者注，《Learning Spark》的主要章节的中文版可以在 CSDN 资源下载，或者在我的 CSDN Blog 里找到



你可以像下面这样来降低 Spark shell 的日志级别：

- 首先，进入到 `$SCALAHOME/conf` 目录
- 然后，创建一个新文件 `log4j.properties`
- 在 `conf` 目录里，打开模板文件 `log4j.properties.template` 并复制其内容到 `log4j.properties` 中
- 查找 `log4j.rootCategory=INFO,console` 这一行，并用下面两行中的一行替换：
 - `log4j.rootCategory=WARN, console`
 - `log4j.rootCategory=ERROR, console`
- 最后，重新启动 Spark shell，可以看到 shell 的输出较少的日志消息。

GraphX 入门

现在我们已经安装了 Spark 并使用了 Spark shell，让我们通过在 shell 中编写代码来创建我们在 Spark 中的第一个图，并依靠这些代码开发和运行一个独立程序。本节中我们三个学习目标：

1. 首先，你将通过一个具体的例子来学习如何用 Spark Core 和 GraphX API 来构建和研究图。
2. 然后，你将复习一些 Scala 编程的重要特性，这些特性在 Spark 中进行图处理时是需要重点掌握的。
3. 最后，你将学习如何开发和运行一个独立的 Spark 应用。

构建小型社交网络

让我们来创建一个小型社交网络并研究网络中不同人之间的关系吧。再次提醒，学习 Spark 最好的方式是使用 shell。所以我们的工作流是先在 shell 中试验，然后迁移我们的代码到独立 Spark 应用。启动 shell 前确认当前目录是 `$SPARKHOME`。

首先我们需要如下所示的导入 GraphX 和 RDD 模块，这样我们就能用更短的名字调用 API：

```
scala> import org.apache.spark.graphx._  
scala> import org.apache.spark.rdd.RDD
```


之前说过 `SparkContext` 是 Spark 程序的主入口点，并且是在 Spark shell 启动时自动创建的。它还提供了有用的方法来从本地集合创建 RDD，从本地或 Hadoop 文件系统加载数据到 rdd，以及保存输出数据到磁盘。

加载数据

在本例中，我们将处理两个文件 `people.csv` 和 `links.csv`，这两文件包含在目录 `$SPARKHOME/data` 中。我们输入以下的命令来加载这些文件到 Spark：

```
scala> val people = sc.textFile("./data/people.csv")
people: org.apache.spark.rdd.RDD[String] = ./data/people.csv
MappedRDD[81] at textFile at <console>:33
```

```
scala> val links = sc.textFile("./data/links.csv")
links: org.apache.spark.rdd.RDD[String] = ./data/links.csv MappedRDD[83]
at textFile at <console>:33
```

加载这些 CSV 文件返回给我们两个字符串 RDD。要创建图，我们需要解析这些字符串到两个定点和边的适当的集合。




Shell 中的当前目录是 `$SPARKHOME` 十分重要。否则的话，你后面会遇到错误。因为 Spark 找不到那俩文件。

属性图

在深入之前，先介绍一些关键定义和图抽象。在 Spark 中，图被表示为属性图，定义在 `Graph` 类，如下：

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED, VD]
}
```

这表示 `Graph` 类提供了访问它的顶点和边的 `get` 方法。这些是后来被抽象的 `RDD` 子类 `VertexRDD[VD]` 和 `EdgeRDD[ED,VD]`。注意,这里的 `VD` 和 `ED` 指的是 `VertexRDD`, `EdgeRDD` 和 `Graph` 类的一些 `Scala` 类型参数。这些参数类型可以说基本类型,如 `String`,或者是用户自定义类型,如我们的社交图示例中的 `Person` 类。这里要重点指出的是 `Spark` 中的属性图是有向多重图。这表示在任意一组顶点间允许有多条边。同时,每条边都是有向的,定义了一个单向关系。这很容易理解,例如,在一个 `Twitter` 图中,一个用户可以粉另一个用户,但是反过来不需要一定是这样。对于双向连接模型,比如 `Facebook` 的朋友关系,我们需要在节点间定义两条边,并且这两条边指向相对的方向。关于关系的额外属性可以保存为边的属性。

 所谓属性图就是有用用户定义对象附加到顶点和边的图。这些对象类描述了图的属性。在实践中是通过参数化 `Graph`, `VertexRDD`, `EdgeRDD` 类来实现。并且图中的每条边定义了一个单向关系,但是任意一对顶点间可以存在多条边。

转换 `RDD` 到 `VertexRDD` 和 `EdgeRDD`

回到我们的例子,用三个步骤构建图,如下所示:

1. 我们定义一个 `case class Person`,带一个 `age` 和一个 `name` 的参数。`Case` 类在我们后面需要对 `Person` 对象做模式匹配时非常有用。

```
case class Person(name: String, age: Int)
```

2. 下一步,我们要解析 `people` 和 `link` 中的每一行 `CSV` 文本到对应的类型为 `Person` 和 `Edge` 的新对象,并收集结果到 `RDD[(VertexId, Person)]` 和 `RDD[Edge[String]]` 中。

```
val peopleRDD: RDD[(VertexId, Person)] = people map { line
=>
    val row = line split ','
    (row(0).toInt, Person(row(1), row(2).toInt))
}
```

```
scala> type Connection = String
scala> val linksRDD: RDD[Edge[Connection]] = links map {line =>
  val row = line split ' '
  Edge(row(0).toInt, row(1).toInt, row(2))
}
```

要在 shell 中粘贴或编写多行代码：

- 输入命令：paste
- 粘贴或编写需要的代码
- 在 Mac 上按 Cmd+D 或 Windows 上按 Ctrl+D 对代码求值

VertexId 在 Graph 中就是被定义为 Long 类型的别名。另外，Edge 类是定义在 org.apache.spark.graphx.Edge 中，如下所示：



```
class Edge(srcId: VertexId, dstId: VertexId, attr:ED)
```

类参数 srcId 和 dstId 是源和目的顶点 ID，被一条边连接。在我们的社交网络示例中，两个人之间的连接是单向的，并且属性 attr 被描述为 Connection 类型。我们在这里定义 Connection 为 String 类型的别名。为了清晰，为 Edge 的类型参数给定一个有意义的参数名通常会有帮助。

3. 现在，我们可以创建我们的社交图并命名为 tinySocial，使用工厂方法

Graph(...):

```
scala> val tinySocial: Graph[Person, Connection] =
  Graph(peopleRDD, linksRDD)
tinySocial: org.apache.spark.graphx.Graph[Person,Connection] =
  org.apache.spark.graphx.impl.GraphImpl@128cd92a
```

关于这个构造函数，有两点要说明。之前我告诉过你图的成员 vertices 和 edges 是 VertexRDD[VD] 和 EdgeRDD[ED,VD] 的实例。然而，我们给上面的 Graph 工厂方法传入的是 RDD[(VertexID, Person)] 和 RDD[Edge[Connection]]。这是如何工作的？这是因为 VertexRDD[VD] 和 EdgeRDD[ED,VD] 类分别是 RDD[(VertexId, Person)] 和 RDD[Edge[Connection]] 的子类。另外，VertexRDD[VD] 加了约束，VertexID 只能出现一次。根本上，我们社交网络中的两个人不会有相同的 VertexID。此外，VertexRDD[VD] 和 EdgeRDD[ED,VD] 提供了一些操作来变换顶点和边的属性。在后面的章节中我们会了解更多。

图操作介绍

最后我们来看一下网络中的点和边，通过访问和收集它们的方式：

```
scala> tinySocial.vertices.collect()
res: Array[(org.apache.spark.graphx.VertexId, Person)] =
Array((4,Person(Dave,25)), (6,Person(Faith,21)), (8,Person(Harvey,47)),
(2,Person(Bob,18)), (1,Person(Alice,20)), (3,Person(Charlie,30)),
(7,Person(George,34)), (9,Person(Ivy,21)), (5,Person(Eve,30)))
scala> tinySocial.edges.collect()
res: Array[org.apache.spark.graphx.Edge[Connection]] =
Array(Edge(1,2,friend), Edge(1,3,sister), Edge(2,4,brother),
Edge(3,2,boss), Edge(4,5,client), Edge(1,9,friend), Edge(6,7,cousin),
Edge(7,9,coworker), Edge(8,9,father))
```

我们使用 Graph 类的 edges 和 vertices 的 get 方法和 RDD 的动作 collect 来将结果放到本地集合中。

现在，假设我们想仅打印下面 profLinks 列表中列出的职业连接：

```
val profLinks: List[Connection] = List("Coworker", "Boss",
"Employee", "Client", "Supplier")
```

获得这个结果的一种糟糕的方式是过滤出对应于这些职业连接的边，然后循环所有过滤出的边，提取对应顶点的名字，并打印出源和目标顶点的连接。这个方法的代码如下：

```
val profNetwork =
tinySocial.edges.filter{ case Edge(_,_,link) =>
profLinks.contains(link)}
for {
  Edge(src, dst, link) <- profNetwork.collect()
  srcName = (peopleRDD.filter{case (id, person) => id == src}
first)._2.name
  dstName = (peopleRDD.filter{case (id, person) => id == dst}
first)._2.name
} println(srcName + " is a " + link + " of " + dstName)
```

```
Charlie is a boss of Bob
Dave is a client of Eve
George is a coworker of Ivy
```

前面的代码有两个问题。首先，它可以更简洁更富于表达；其次，在循环内的过滤操作太没效率。

幸运的是，还有更好的选择。**GraphX** 库提供了两种不同的方式查看数据：既可以是图，也可以是边，顶点或三元组的表格。对于每一种视图，**GraphX** 库都提供了一组丰富的操作，其实现也是为执行优化过的。这意味着我们通常可以用预定义的图操作和算法轻松地处理图。例如，我们可以简化之前的代码使它更高效，如下所示：

```
tinySocial.subgraph(profLinks contains _.attr).  
  triplets.foreach(t => println(t.srcAttr.name + " is a " +  
    t.attr + " of " + t.dstAttr.name))  
Charlie is a boss of Bob  
Dave is a client of Eve  
George is a coworker of Ivy
```

我们仅用 `subgraph` 操作来过滤职业连接。然后，用 **triplet 视图** 并行访问边和顶点的属性。简而言之，`triplet` 操作返回了 `EdgeTriplet[Person, Connection]` 的 RDD。注意 `EdgeTriplet` 只是三元组 `((VertexId, Person), (VertexId, Person), Connection)` 的参数化类型的别名，包含了关于源节点，目标节点和边属性的所有信息。

构建和提交一个独立应用

让我们为我们的社交网络示例开发和运行一个独立程序来结束本章吧。

编写和配置 Spark 程序

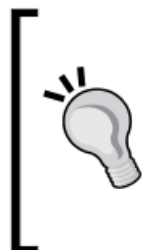
在 shell 中试验满意后，现在我们来编写我们的第一个 **Spark** 程序。打开你喜欢的文本行编辑器，并创建一个名叫 `simpleGraph.scala` 的新文件，并将它放到目录 `$$SPARKHOME/exercises/chap1` 下。**Spark** 程序的模板看起来像是下面的代码：

```
import org.apache.spark.SparkContext  
import org.apache.spark.SparkContext._  
import org.apache.spark.SparkConf
```

```
import org.apache.spark.rdd.RDD
import org.apache.spark.graphx._
object SimpleGraphApp {
  def main(args: Array[String]){
    // Configure the program
    val conf = new SparkConf()
    .setAppName("Tiny Social")
    .setMaster("local")
    .set("spark.driver.memory", "2G")
    val sc = new SparkContext(conf)
    // Load some data into RDDs
    val people = sc.textFile("./data/people.csv")
    val links = sc.textFile("./data/links.csv")
    // After that, we use the Spark API as in the shell
    // ...
  }
}
```

你能在示例文件中看到 SimpleGraph.scala 的完整代码，可以在 Packt 网站下载。

下载示例代码



你可以在 <http://www.packtpub.com> 从你的账号下载所有你付费过的 Packt 出版的书的示例代码文件。如果你是从其他地方付费的，你可以访问 <http://www.packtpub.com/support> 并注册来通过 email 直接获得。

让我们重温一下代码来理解创建和配置一个独立的 Spark 应用有什么要求。

作为一个 Scala 程序，我们的 Spark 程序应当被构建在一个顶级 Scala 对象内，它必须有一个 main 函数，签名为 `def main(args: Array[String]): Unit`。换句话说，主程序接受一个字符串数组为参数，没有返回值。在我们的示例中，顶级对象是 SimpleGraphApp。

在 `simpleGraph.scala` 文件在开始处，我们写了下面这些的 `import` 语句：

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

前两行导入了 `SparkContext` 类，此外还有一些定义在它的伴生对象里的隐式转换。了解这些隐式转换是什么不是非常重要。只要确认你导入了 `SparkContext` 和 `SparkContext._` 就好了。

 当你使用 Spark shell 时，`SparkContext` 和 `SparkContext._` 已经为我们使用被自动导入了。

第三行导入的 `SparkConf` 类，这是个封装类，包含了 `Spark` 应用的配置，如应用名称，每个 `executor` 的内存大小，`master` 或集群管理器的地址等。

接下来的几行，我们导入了一些 `RDD` 和 `GraphX` 相关的类构造器和操作：

```
import org.apache.spark.rdd.RDD
import org.apache.spark.graphx._
```

`Org.apache.spark.graphx` 后面的下划线确保所有 `GraphX` 的公共 API 都被导入了。

在 `main` 函数中我们必须先配置 `Spark` 程序。为此我们创建了一个名为 `SparkConf` 的对象，并通过 `SparkConf` 对象的 `set` 方法链设定了应用配置。`SparkConf` 提供了一些常用属性的特定 `set` 方法，比如应用名和 `master`。或者，也可以用一般的 `set` 方法传入一系列键值对来一起设置多个属性。最常见的配置参数已列出在下表，包括默认值和用法。更多的列表见 <https://spark.apache.org/docs/latest/configuration.html>。

Spark 属性名称	用法及默认值
<code>spark.app.name</code>	这是你应用的名字。这个名字会显示在 UI 以及日志数据中。

Spark 属性名称	用法及默认值
spark.master	这是要连接的集群管理器。例如，spark://host:port, mesos://host:port, yarn 或者 local
spark.executor.memory	这是每个 executor 进程使用的内存数量，和 JVM 的内存字符串格式相同（比如 512M, 2G）。默认值 1G。
spark.driver.memory	当你在本地运行 Spark，spark.master=local 时，你的 executor 就成了 driver，你需要设置该参数而不是 spark.executor.memory。默认值是 512M。
spark.storage.memoryFraction	这是 Java 堆内存可用于 Spark 内存缓存的因子。默认值是 0.6。
spark.serializer	这是用于序列化对象在网络上发送或者被以序列化形式缓存的类。它是默认类 org.apache.spark.serializer.JavaSerializer 的子类。

在我们的例子中，我们是这样初始化的：

```
val conf = new SparkConf()
    .setAppName("Tiny Social")
    .setMaster("local")
    .set("spark.driver.memory", "2G")
val sc = new SparkContext(conf)
```

严格的说，我们设置了应用名为“Tiny Social”，master 是提交应用所在的机器本地。另外 driver 内存被设置为 2G。如果我们要设置 master 到集群，而不是本地，我们应当通过 spark.executor.memory 来指定每个 executor 使用的内存，而不是指定 spark.driver.memory。



原则上，driver 和 executor 有不同的角色，一般运行在不同的进程。除非设置了 master 为 local。Driver 进程编辑我们的程序为任务，并调度到一个或多个 executor 上，并维护每个 RDD 的物理位置。每个 executor 进程的职责是执行任务，在内存中保存和缓存 RDD。

在你的程序内设定 Spark 应用配置到 SparkConf 对象不是强制的。也可以在提交我们的应用时，用 spark-submit 工具的命令行选项设置这些参数。在下面的章节会讲到这部分内容。现在我们就这样创建我们的 SparkConf 对象：

```
val sc = new SparkContext(new SparkConf())
```

配置完程序后，接下来就是要加载我们要处理的数据，通过 SparkConf 对象 sc 的类似 sc.textFile 的工具方法：

```
val people = sc.textFile("./data/people.csv")  
val links = sc.textFile("./data/links.csv")
```

最后，剩下的程序由和我们在 shell 里使用的 RDD 和图的相同操作组成。



为避免混淆，当传递相对文件路径到类似 sc.textFile() 这样的 I/O 动作时，本书的习惯用法是总是设置命令行的当前目录为项目根目录。例如，如果我们的 Tiny Social 应用的根目录是 \$SPARKHOME/exercises/chap1，那么 Spark 将会到文件路径 \$SPARKHOME/exercises/chap1/data 下去查找要加载的数据。这里假定我们将文件放到了 data 目录下。

用 Scala Build Tool 构建程序

写完整个程序后，我们将用 **Scala Build Tool(SBT)** 构建它。如果你的电脑里还没有安装 SBT，你需要先装一个。对于大多数操作系统，如何安装 SBT 的详细操作指南见 <http://www.scala-sbt.org/0.13/tutorial/index.html>。还有些不同的方法安装 SBT，我建议使用包管理器而不是手动安装。装完之后，执行下面的命令将追加 SBT 的安装目录到 PATH 环境变量中：

```
$ export PATH=$PATH:/usr/local/bin/sbt1
```

一旦我们安装好 SBT，就可以用它来构建我们的程序和所有的依赖到一个单独的 JAR 包文件，称为 **uber jar**。实际上，当在多个 worker 机器上运行 Spark 程序时，如果有些机器上没有正确的依赖 JAR，就会出错。

¹ 译者注，这里貌似原文笔误，没有 sbt1。另外，不一定是 /usr/local/bin/sbt，看情况

通过用 SBT 打包到 uber jar，应用的代码和它的依赖都被分发到 worker 上。具体来说，我们需要创建一个构建定义文件，在里面设置好项目配置。同时，我们还要指定依赖和解析器，帮助 SBT 找到程序需要的包。解析器指出所需的 JAR 包所在的软件库的名字和位置。让我们在项目根目录\$SPARKHOME/exercises/chap1 下创建一个 build.sbt 的文件，并插入如下几行：

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.10.4"
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.4.1",
  "org.apache.spark" %% "spark-graphx" % "1.4.1"
)
resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

按惯例，配置是用 Scala 表达式定义，需要被空行分开。之前，我们设置了项目名称，版本号，Scala 版本号，还有 Spark 库依赖。要构建程序，输入下面命令：

```
$ sbt package
```

这会创建一个 JAR 文件在\$SPARKHOME/exercises/chap1/target/scala-2.10/simple-project_2.10-1.0.jar。

用 Spark-submit 分发和运行

最终，我们在终端中调用\$SPARKHOME/bin/目录下的 spark-submit 脚本来运行\$SPARKHOME/exercises/chap1 下的程序：

```
$ ../../bin/spark-submit --class \
"SimpleGraphApp" \
./target/scala-2.10/simple-project_2.10-1.0.jar
```

Spark assembly has been built with Hive, including Datanucleus jars on

classpath

Charlie is a boss of Bob

Dave is a client of Eve

George is a coworker of Ivy

Spark-submit 命令需要的参数是 Spark 应用对象名字和之前用 SBT 构建的 JAR 包。如果我们没有在创建 SparkConf 中设定 master 配置，也可以在 spark-submit 指定 --master 参数。



你可以用下面的命令列出 spark-submit 脚本的所有可用参数：

spark-submit --help

关于如何提交一个 Spark 应用到远程集群的更多细节详见
<http://spark.apache.org/docs/latest/submitting-applications.html>

总结

在本章，我们来了个 Spark 中图处理的旋风之旅。具体的说，我们安装了 JDK，预编译的 Spark 版本以及 SBT 工具。此外，通过在 Spark shell 以及一个独立应用中创建一个社交网络的方式介绍了 Spark 中的图抽象和操作。

在下一章，你将学习更多关于在 Spark 中如何构建和探索图。

构造和探索 图

本章的目标是教会我们在 Spark 和 GraphX 中如何表示各种类型的网络和复杂系统为属性图。在我们能描述这些系统的行为，分析这些系统的内在结构之前，我们首先需要映射它们的组件到顶点或者节点，映射这些独立个体之间的交互为边或者连接。以我们之前章节学到的为基础，我们将钻研关于 GraphX 中如何存储和表达图的细节。另外，本章介绍了图论，以及图的基本特征。贯穿本章，我们将使用真实世界的数据集，映射到不同类型的图。样例包括 email 通讯网络，食物风味网络，社交自我网络等。在结束本章时，你将理解如何：

- 通过多种方式加载数据并构建 Spark 图
- 使用 join 操作符混合外部数据到已存在的图
- 构建二分图和多重图
- 探索图，计算它们的基本统计量

网络数据集

在前一章，我们构造过一个小的社交网络图示例，没实用价值。从本章开始，我们将开始处理现实世界的数据集，取材于各种应用。实际上，图被用来表达任意复杂的系统，它描述了系统中各组件间的交互作用。尽管不同系统在形式，大小，性质和粒度上均有差异，图论提供了通用的语言以及一组工具来表达和分析复杂系统。



简单来说，图是由一组边连接的顶点集合组成。每条边表示连接的一组顶点的关系。本书中，有时候我们用简短的术语网络节点来表示顶点，用连接表示边。注意，Spark 支持多重图，也就是说，它允许一对节点之间有多条边。

让我们来预览下本章中我们将要构建的网络。

通信网络

我们将遇到的第一种通信网络是 **Email 通信图**。一个组织的 email 交互历史可以映射为一个通信图以便了解组织背后的非正式结构。这样的图也可以用来确定组织中有更大影响力的人，或者是组织的中心，也许未必是某个级别高的。Email 通信网络是有向图的典型例子，每封 email 连接着一个源节点和一个目标节点。我们将使用安然语料库(**Enron Corpus**)，这是安然公司的 158 名雇员生成的 email 数据库。这是唯一的网上公开的被大量采集的团体 email。安然语料库特别有意思，它捕获了在导致公司破产的丑闻之前的公司内部所有的通信。原始的数据集是由 William Cohem 发布在 CMU，可以从 <https://www.cs.cmu.edu/~.enron/> 下载。这个完整数据集的详细描述是由 Klimmt 和 Yang 在 2004 年完成的。在这里，我们使用到的数据集是一个更精炼的版本，由 Leskovec 在 2009 年提供，你可以从 <https://snap.stanford.edu/data/email-Enron.html> 下载。


调味网络

我们要借用的另一个例子来自烹饪世界的原料复合网络，是 Ahn et al 于 2011 年提出的。这是个二分图，也就是说这些节点被分成两个不相交的集合：原料节点和化合物节点。当一种化合物出现在食物原料中时，就有一个原料到化合物的连结。依据这个原料-化合物网络，当然有可能创建一个叫做调味网络的东西。不同于连结原料到化合物，每当一对原料共有至少一种化合物时，调味网络就连结这一对原料。

在本章我们会构造出原料-化合物网络，然后，在第四章 *改变和塑造图来满足你的需要*，我们将根据原料-化合物网络来构建调味网络。分析这样的图极有趣，因为这会帮助我们了解更多的食物搭配和饮食文化。调味网络也能帮助食品科学家或者业余厨师创造新的食谱。我们将用到的数据库包括原料-化合物和食谱，采集自 <http://www.epicurious.com/>, allrecipes.com 和 <http://www.menupan.com/>。数据库可以在 <http://yongyeol.com/2011/12/15/paper-flavor-network.html> 下载。

社交自我网络

本章我们最后要研究的数据库是来源于 Google+ 的社交自我网络的集合。这些数据由 McAuley 和 Leskovec 在 2012 年采集自通过**共享圈子**功能手动共享了他们自己的社交圈子的用户。本数据集包括用户资料，他们的圈子以及他们的自我网络，可以在斯坦福的 SNAP 项目网站下载 <http://snap.stanford.edu/data/egonets-Gplus.html>

 这些数据库没有随着 Spark 的安装提供。必须先从它们的网站下载后复制到 \$SPARKHOME/data 目录下。当有不同大小的数据库可用时，我们选择这些数据库中的较小的版本以便快速演示本书讲授的概念。

图构造器

在 GraphX 中有四个函数用于构造属性图。其中的每个函数都要求用来构造图的数据有其特定的数据结构。

图工厂方法

第一个方法就是之前章节已经看到过的图工厂方法。定义在 Graph 伴生对象的 apply 方法中，如下：

```
def apply[VD, ED](  
    vertices: RDD[(VertexId, VD)],
```

```
edges: RDD[Edge[ED]],
defaultVertexAttr: VD = null)

: Graph[VD, ED]
```

同之前看到的一样，函数带两个 RDD 集合：RDD[(VertexID, VD)]和 RDD[Edge[ED]] 作为顶点和边对应的参数来构造 Graph[VD, ED]的参数。参数 defaultVertexAttr 是用于给那些出现在边的 RDD 而没有在顶点 RDD 中的点赋一个默认值。图工厂方法适用于已有边和点的 RDD 集合的情况。

edgeListFile

更常见的情况是你的数据集仅仅表示出了边的数据。这样的话，GraphX 提供了下面的 GraphLoader.edgeListFile 方法，定义在 GraphLoader 中：

```
def edgeListFile(
  sc: SparkContext,
  path: String,
  canonicalOrientation: Boolean = false,
  minEdgePartitions: Int = 1)

: Graph[Int, Int]
```

它带了一个路径参数指向包含边列表的文件。文件中每一行表示图中的一条边，用两个整数以如下形式表示：源 ID 目标 ID。当读取文件中的边列表时，会忽略以#开头的任何注释行。从这些边和对应的顶点就构造出了图。

参数 minEdgePartitions 是要生成的边分区的最小值。如果邻接表被分区为比 minEdgePartitions 更多的块，那么更多的分区就会被创建。

fromEdges

类似于 GraphLoader.edgeListFile 方法，第三个函数为 Graph.fromEdges，允许你从 RDD[Edge[ED]]集合创建图。另外，它还可以用边的 RDD 中指定的顶点 ID 参数自动创建顶点，而 defaultValue 参数就是默认的顶点属性：

```
def fromEdges[VD, ED](
  edges: RDD[Edge[ED]],
  defaultValue: VD)

: Graph[VD, ED]
```

fromEdgeTuples

最后一个图构造器函数是 `Graph.fromEdgeTuples`。它仅仅是从边的二元组的 RDD 来创建图，也就是 `RDD[(VertexID, VertexID)]` 类型集合。其默认赋值边属性为 1。

```
def fromEdgeTuples[VD](  
  rawEdges: RDD[(VertexId, VertexId)],  
  defaultValue: VD,  
  uniqueEdges: Option[PartitionStrategy] = None)  
  : Graph[VD, Int]
```

构建图

现在让我们打开 Spark Shell，使用前面的图构造器来构造三种类型的图：有向 Email 通信网络，原料-化合物连接的两分图，以及多重图。



除非另行说明，否则的话，我们总是假定 Spark Shell 是从 `$SPARKHOME` 目录下启动的。该目录就是本书中任何用到相对路径的当前目录。

构建有向图

第一个要构建的图是安然 Email 通信网络。如果你已经重新启动了 Spark Shell，你需要再次导入 GraphX 库。首先，在 `$SPARKHOME` 目录创建一个新目录 `data`，然后复制数据集到里面。这些文件包含了员工间 email 通信的邻接表。假定当前目录是 `$SPARKHOME`，我们将文件路径传入 `GraphLoader.edgeListFile` 方法：

```
scala> import org.apache.spark.graphx._  
import org.apache.spark.graphx._
```

```
scala> import org.apache.spark.rdd._  
import org.apache.spark.rdd._
```

```
scala> val emailGraph = GraphLoader.edgeListFile(sc, "./data/emailEnron.txt")

emailGraph: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@609db0e
```

注意 GraphLoader.edgeListFile 方法总是返回一个 graph 对象，其顶点和边的属性类型为 Int，默认值是 1。可以通过查看前 5 个点和边来确认一下：

```
scala> emailGraph.vertices.take(5)

res: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((19021,1),
(28730,1), (23776,1), (31037,1), (34207,1))
```

```
scala> emailGraph.edges.take(5)

res: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(0,1,1),
Edge(1,0,1), Edge(1,2,1), Edge(1,3,1), Edge(1,4,1))
```

第一个顶点(19021,1)的顶点 ID 是 19021，其属性值确实是设置的 1。类似的，第一条边(0,1,1)捕获的是在源 ID 0 和目标 ID 1 之间的通信。

在 GraphX 中，所有的边必须有方向。要表示无方向或双向的图，可以在点对的两个方向都建立连接。在我们的通信网络中，可以验证例如 19021 节点，既有进入的连接，也有发出的连接。首先，我们来收集节点 19021 通信的目标节点：

```
scala> emailGraph.edges.filter(_._srcId == 19021).map(_._dstId).collect()

res: Array[org.apache.spark.graphx.VertexId] = Array(696, 4232, 6811,
8315, 26007)
```

结果显示这些都是和进入 19021 的边的源节点相同的节点。

```
scala> emailGraph.edges.filter(_._dstId == 19021).map(_._srcId).collect()

res: Array[org.apache.spark.graphx.VertexId] = Array(696, 4232, 6811,
8315, 26007)
```

构建二分图

有些应用中，将系统的视图展现为二分图会更好。二分图由两个节点集合组成，在相同集合中的节点不会连接，只有不同集合中的节点间有连接。这种图的一个例子就是食物原料-化合物网络。

这里要处理的文件有 `ingr_info.tsv`, `comp_info.tsv` 和 `ingr_comp.tsv`, 复制其到 `$$SPARKHOME` 目录中。前两个文件分别包含了食物原料和化合物的信息。

让我们用 `scala.io.Source` 包中的 `Source.fromFile` 方法快速瞄一眼这两文件的第一行。对这个方法我们唯一的要求就是简单看下文件的开头部分内容:

```
scala> import scala.io.Source
```

```
import scala.io.Source
```

```
scala> Source.fromFile("./data/ingr_info.tsv").getLines().take(7).foreach(println)
```

```
# id ingredient name category
```

```
0 magnolia_tripetala flower
```

```
1 calyptranthes_parriculata plant
```

```
2 chamaecyparis_pisifera_oil plant derivative
```

```
3 mackerel fish/seafood
```

```
4 mimusops_elengi_flower flower
```

```
5 hyssop herb
```

```
scala> Source.fromFile("./data/comp_info.tsv").getLines().take(7).foreach(println)
```

```
# id Compound name CAS number
```

```
0 jasmone 488-10-8
```

```
1 5-methylhexanoic_acid 628-46-6
```

```
2 l-glutamine 56-85-9
```

```
3 1-methyl-3-methoxy-4-isopropylbenzene 1076-56-8
```

```
4 methyl-3-phenylpropionate 103-25-3
```

```
5 3-mercapto-2-methylpentan-1-ol_(racemic) 227456-27-1
```

第三个文件包含了原料和化合物之间的邻接表:

```
scala> Source.fromFile("./data/ingr_comp.tsv").getLines().take(7).foreach(println)
```

```
# ingredient id compound id
```

```
1392 906
```

```
1259 861
```


1079 673

22 906

103 906

1005 906

实际上,我们用来构造图的数据集并不会是以 **Spark** 的图构造器期待的形式存在。例如,在食物网络的示例中,数据集有两个问题。首先,我们不能简单的从邻接表直接创建图,因为原料和化合物的索引都从 **0** 开始,并且互有重叠。因此如果发生节点 **id** 相同的情况会无法区分。其次,我们有两种节点: 原料和化合物:



为了创建两分图,我们首先要创建 **Ingredient** 类和 **Compound** 类。并且使用 **scala** 的继承,这两个类都继承自 **FNNode** 类

```
scala> class FNNode(val name: String)
defined class FNNode
```

```
scala> case class Ingredient(override val name: String, category: String)
extends FNNode(name)
defined class Ingredient
```

```
scala> case class Compound(override val name: String, cas: String)
extends FNNode(name)
defined class Compound
```

然后,我们要加载所有的 **Compound** 和 **Ingredient** 对象到 **RDD[FNNode]**集合中。这部分需要一些数据整理:

```
val ingredients: RDD[(VertexId, FNNode)] =
  sc.textFile("./data/ingr_info.tsv").
    filter(! _.startsWith("#")).
    map {line =>
      val row = line split '\t'
      (row(0).toInt, Ingredient(row(1), row(2)))
    }
ingredients:
org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
FNNode)] = MappedRDD[32] at map at <console>:26
```

在前面的代码中，我们首先加载 `comp_info.tsv` 文件中的文本到 `RDD[String]` 中，并过滤了以 `#` 开头的注释行。然后我们解析了 `tab` 分隔的行到 `RDD[Ingredient]` 顶点集合。接下来就可以对 `comp_info.tsv` 做些类似的处理，并创建 `RDD[Compound]` 顶点集：

```
val compounds: RDD[(VertexId, FNNode)] =
  sc.textFile("./data/comp_info.tsv").
    filter(!_._startsWith("#")).
    map {line =>
      val row = line split '\t'
      (10000L + row(0).toInt, Compound(row(1), row(2)))
    }
compounds:
org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,FNNode)] =
MappedRDD[28] at map at <console>:26
```

然而这里有个关键问题我们之前提过，因为每个节点要有唯一索引，我们不得不改变化合物的索引，增加 10000。这样就不会同时有索引对应原料和化合物了。

接下来我们从 `ingr_comp.tsv` 数据集创建一个 `RDD[Edge[Int]]` 集合：

```
val links: RDD[Edge[Int]] =
  sc.textFile("./data/ingr_comp.tsv").
    filter(!_._startsWith("#")).
    map {line =>
      val row = line split '\t'
      Edge(row(0).toInt, 10000L + row(1).toInt, 1)
    }
}
```

当解析 `ingr_comp.tsv` 中邻接表的每一行数据，也都需要对化合物的索引增加 10000。这个改进很完美是因为我们从数据集的描述中预先知道在数据集中有多少原料和化合物。处理现实中凌乱的数据集时需要更小心一些。接下来，由于原料和化合物之间的连接并没有权重或者有意义的属性值，我们就将 `Edge` 类的模板参数设为 `Int` 类型，并设置默认值 1 作为每个连接的属性值就好了。

最后，我们将两个节点集合串起来到一个单独的 `RDD` 中，然后和连接的 `RDD` 一起传给 `Graph()` 工厂方法：

```
scala> val nodes = ingredients ++ compounds
```

```
nodes: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
FNNode)] = UnionRDD[61] at $plus$plus at <console>:30
```

```
scala> val foodNetwork = Graph(nodes, links)
```

```
foodNetwork: org.apache.spark.graphx.Graph[FNNode,Int] = org.apache.
spark.graphx.impl.GraphImpl@635933c1
```

现在我们来研究下原料-化合物图：

```
scala> def showTriplet(t: EdgeTriplet[FNNode,Int]): String = "The
ingredient " ++ t.srcAttr.name ++ " contains " ++ t.dstAttr.name
```

```
showTriplet: (t: EdgeTriplet[FNNode,Int])String
```

```
scala> foodNetwork.triplets.take(5).
  foreach(showTriplet _ andThen println _)
The ingredient calypttranthes_parriculata contains citral_(neral)
The ingredient chamaecyparis_pisifera_oil contains undecanoic_acid
The ingredient hyssop contains myrtenyl_acetate
The ingredient hyssop contains 4-(2,6,6-trimethyl-cyclohexa-1,3-dienyl)
but-2-en-4-one
The ingredient buchu contains menthol
```

最开始我们定义了一个辅助函数 ShowTriplet，它返回一个原料-化合物连接的 Triplet 的 String 类型描述。然后我们取了前 5 个 Triplet 并打印输出到控制台。在上面的例子中，我们在参数中用到了 Scala 的复合函数 showTriplet _ andThen println _ 并将它传给了 foreach 方法。

构建有权重的社交自我网络

最后一个例子，我们来构建一个在前面章节提到过的来自 Google+ 数据集的自我网络。自我网络是展示一个人的连接关系的图。更精确地说，它关注的是一个被称作焦点的单节点，并且仅仅表现这个节点和其邻居节点间的连接。尽管 SNAP 网站的整个数据集包含了 133 个用户的自我网络，我们只打算构建一个人的自我网络来说明。我们将要处理的文件已经被放置在 \$SPARKHOME/data 目录下。

它们的描述如下：

- **ego.edges**: 自我网络中的有向边。焦点并未出现在列表中，但是假定了它跟随每一个出现在本文件中的节点 ID。
- **ego.feats**: edge 文件中出现的每个节点的特征。
- **ego.featsnames**: 这是每个特征维度的命名。如果用户有这个属性，则特征值为 1，否则为 0。

首先，我们从 Breeze 库中导入绝对值函数和 `SparseVector` 类，接下来会用到：

```
import scala.math.abs
import breeze.linalg.SparseVector
```

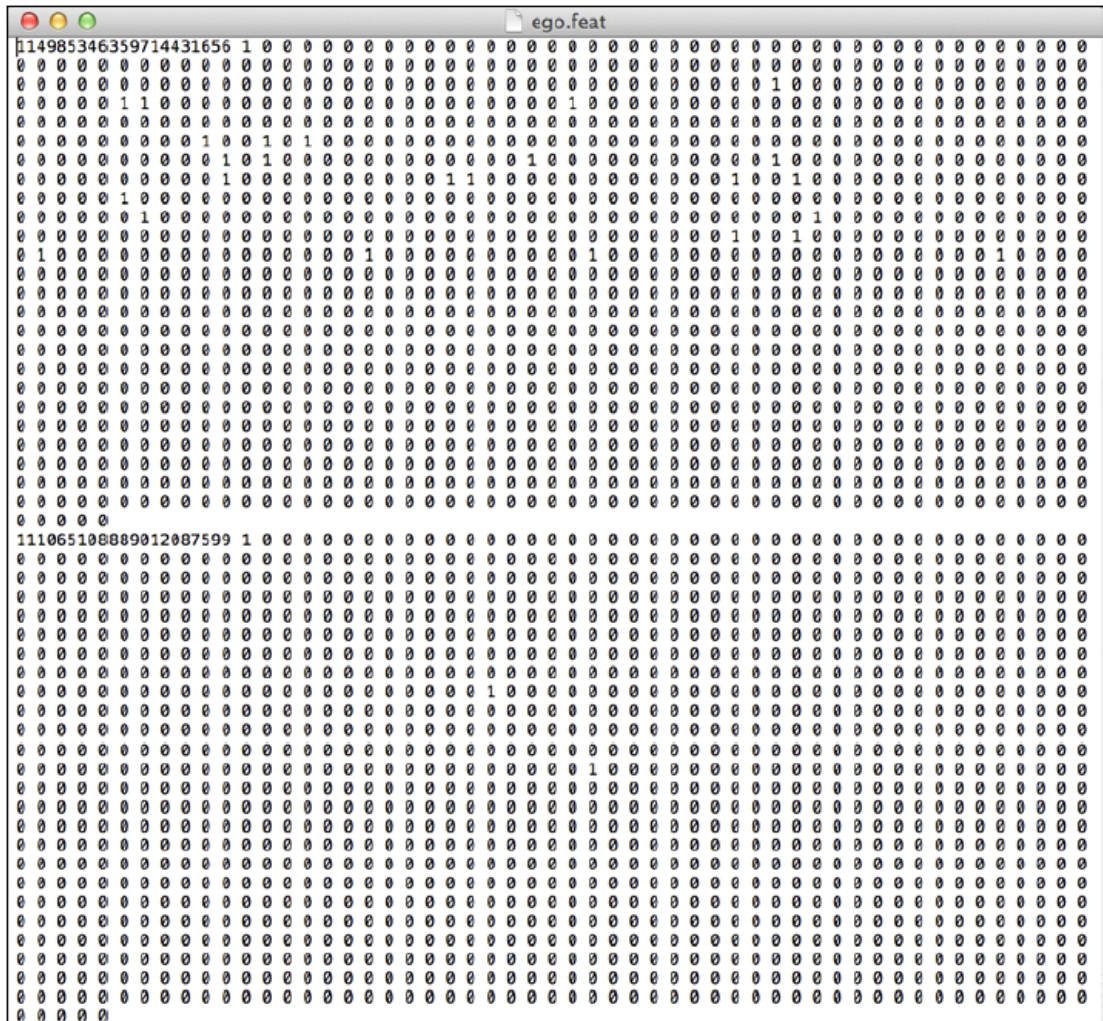
然后，我们为 `SparseVector[Int]` 定义一个类型别名为 `Feature`：

```
type Feature = breeze.linalg.SparseVector[Int]
```

使用下面的代码我们就可以读入 `ego.feats` 文件中的特征值，并收集到 `map` 里，其 `key` 和 `value` 分别是 `Long` 类型和 `Feature` 类型：

```
val featureMap: Map[Long, Feature] =
  Source.fromFile("./data/ego.feats").
    getLines().
    map{line =>
      val row = line split ' '
      val key = abs(row.head.hashCode.toLong)
      val feat = SparseVector(row.tail.map(_toInt))
      (key, feat)
    }.toMap
```

我们后退一步，回头快速查看下 `ego.feats` 文件，理解一下前面的一串 RDD 变换是在做些什么，并且为何需要这么做。`Ego.feats` 中的每一行有如下的形式：



每一行的第一个数字对应自我网络中的节点 ID。

剩下的 0 和 1 的数字串指示这个特定节点有哪个特征。例如，节点 ID 后的第一个 1 对应 `gender:1` 这个特征。事实上，每个特征被设计为 `description:value` 的形式。实际上，我们通常对我们要处理的数据集的格式只有有限的控制。就像本例中，总是有些数据整理还需要我们去做。首先，自我网络中的每个顶点都应该有个 `Long` 类型的顶点 ID。然而，数据集中的节点 ID，比如 114985346359714431656，就超过了 `Long` 类型的允许范围。

因此，我们不得不为节点创建新的索引。其次，我们需要解析数据中的 0,1 字符串来创建特征向量，这种形式更实用。

幸运的是，这些问题都很好处理。要转换原始 ID 到顶点 ID，我们简单的将节点 ID 对应的字符串做 hash，如下：

```
val key = abs(row.head.hashCode.toLong)
```

然后，我们利用 Breeze 库中的 SparseVector 库的优点高效的存储特征索引。

接下来，我们读取 ego.edges 文件来创建自我网络中的连接的 RDD[Edge[Int]]集合。相对于前面的图的例子，这次我们建模自我网络为一个权重图。准确的说，每个连接的属性对应连接这对节点的一个特征值。这可以通过下面的变换完成：

```
val edges: RDD[Edge[Int]] =  
  sc.textFile("./data/ego.edges").  
    map {line =>  
      val row = line split ' '  
      val srcId = abs(row(0).hashCode.toLong)  
      val dstId = abs(row(1).hashCode.toLong)  
      val srcFeat = featureMap(srcId)  
      val dstFeat = featureMap(dstId)  
      val numCommonFeats = srcFeat dot dstFeat  
      Edge(srcId, dstId, numCommonFeats)  
    }
```



要找到源和目的节点间某个特征的值，我们只需要用 Breeze 库中 SparseVector 类的点积操作。此外，我们同样不得不通过数据集中节点 ID 的 hashCode 属性计算新的顶点 ID。

终于，我们现在可以用 Graph.fromEdges 函数来创建一个自我网络了。该函数使用 RDD[Edge[Int]]和默认值 1 这两个参数。

```
val egoNetwork: Graph[Int,Int] = Graph.fromEdges(edges, 1)
```


接下来, 我们就能查看焦点的连接中有多少个节点和其邻接点有一些共同的特征了:

```
scala> egoNetwork.edges.filter(_attr == 3).count()
```

```
res: Long = 1852
```

```
scala> egoNetwork.edges.filter(_attr == 2).count()
```

```
res: Long = 9353
```

```
scala> egoNetwork.edges.filter(_attr == 1).count()
```

```
res: Long = 107934
```

计算网络节点的度

现在, 我们将要研究这三个图, 并介绍一个网络节点中的重要属性, 即节点的度。

节点的度表示表示它和其他节点的连接的个数。在有向图中, 我们可以将度区分为入度(in-degree) 和出度(out-degree)。入度即是指向该节点的连接数; 出度则是该节点指向的连接数。接下来几节内容, 我们将研究这三个示例网络的度分布。

安然 Email 网络的入度和出度

对于安然 Email 网络, 我们可以确定连接数大概是节点数的十倍以上。

```
scala> emailGraph.numEdges
```

```
res: Long = 367662
```

```
scala> emailGraph.numVertices
```

```
res: Long = 36692
```

实际上，本例中员工的入度和出度是完全相同的，因为这个 email 网络是个双向网络。这可以通过查看平均度数来确认：

```
scala> emailGraph.inDegrees.map(_._2).sum / emailGraph.numVertices  
res: Double = 10.020222391802028
```

```
scala> emailGraph.outDegrees.map(_._2).sum / emailGraph.numVertices  
res: Double = 10.020222391802028
```

如果我们想找到给最多的人发 email 的人，可以定义和使用下面的 max 函数：

```
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {  
    if (a._2 > b._2) a else b  
}
```

我们来看看输出：

```
scala> emailGraph.outDegrees.reduce(max)  
res: (org.apache.spark.graphx.VertexId, Int) = (5038,1383)
```

这个人可能是经理或者是充当组织枢纽的员工。类似的，我们可以定义一个 min 函数来找人。现在我们来检查一下在安然是否有孤立的员工组，代码如下：

```
scala> emailGraph.outDegrees.filter(_._2 <= 1).count  
res83: Long = 11211
```

看起来似乎有很多员工仅从一个员工那接收邮件。这个员工可能是老板，或者是来自人力资源部。

二分食物网络中的度

对于原料-化合物二分图，我们也可以研究下有最多化合物的食物，或者哪种化合物在我们的原料列表中最常见：

```
scala> foodNetwork.outDegrees.reduce(max)  
res: (org.apache.spark.graphx.VertexId, Int) = (908,239)  
scala> foodNetwork.vertices.filter(_._1 == 908).collect()
```

```
res: Array[(org.apache.spark.graphx.VertexId, FNNode)] =  
Array((908,Ingredient(black_tea,plant derivative)))
```

```
scala> foodNetwork.inDegrees.reduce(max)  
res: (org.apache.spark.graphx.VertexId, Int) = (10292,299)
```

```
scala> foodNetwork.vertices.filter(_._1 == 10292).collect()  
res: Array[(org.apache.spark.graphx.VertexId, FNNode)] =  
Array((10292,Compound(1-octanol,111-87-5)))
```

前两个问题的答案清楚了，就是黑茶和化合物 1-octanol

社交自我网络的度数直方图

类似的，我们可以计算自我网络中连接的度数。我们来看看网络中的最大值和最小值：

```
scala> egoNetwork.degrees.reduce(max)  
res91: (org.apache.spark.graphx.VertexId, Int) = (1643293729,1084)
```

```
scala> egoNetwork.degrees.reduce(min)  
res92: (org.apache.spark.graphx.VertexId, Int) = (550756674,1)
```

假设我们现在想要的到度的直方图数据，那么可以写下列代码来完成：

```
egoNetwork.degrees.  
  map(t => (t._2,t._1)).  
  groupByKey.map(t => (t._1,t._2.size)).  
  sortBy(_._1).collect()  
  
res: Array[(Int, Int)] = Array((1,15), (2,19), (3,12), (4,17),  
(5,11), (6,19), (7,14), (8,9), (9,8), (10,10), (11,1), (12,9),  
(13,6), (14,7), (15,8), (16,6), (17,5), (18,5), (19,7), (20,6),  
(21,8), (22,5), (23,8), (24,1), (25,2), (26,5), (27,8), (28,4),  
(29,6), (30,7), (31,5), (32,10), (33,6), (34,10), (35,5), (36,9),  
(37,7), (38,8), (39,5), (40,4), (41,3), (42,1), (43,3), (44,5),  
(45,7), (46,6), (47,3), (48,6), (49,1), (50,9), (51,5),...
```

总结

在本章中，我们通过处理来自社交网络、食品科学和 email 通信网络的具体例子学习了 Spark 中构建图的三种不同方法。也看到了构建图需要对数据进行准备和整理。虽然如此，GraphX 还是提供了多种方法供我们选择，这依赖于我们需要创建的图的表示，可用数据集的情况。这样的功能是 GraphX 相对于其他类似图处理框架的优势。此外，我们还看到了一些图的基本统计和属性。这对于表征其结构和理解其表现形式会有帮助。

在下一章中，我们将深度分析图，使用数据可视化工具和新的图论工具和算法，比如连通性，三角形计数和 PageRank。

3

图分析和 可视化

在本章，我们将学习如何用可视化工具和图算法来分析图的特征。比如，我们将使用一些 **GraphX** 中可用的算法来看看图是如何连接的。另外，我们还将计算一些常见指标，例如三角形计数和聚合系数。除此之外，我们还将通过一个具体例子学习 **PageRank** 算法如何被用于计算网络中节点的重要度。在此过程中，我们将介绍在这里和后面章节被证明有用的新 **RDD** 操作。最后，本章还提供了一个小窍门用于依赖第三方库构建 **Spark** 应用。完整本章学习后，你将学会的工具和概念是：

- 可视化大规模图数据
- 计算网络的连通组件
- 使用 **PageRank** 算法计算网络中的节点重要性
- 使用第三方库 **SBT** 构建 **Spark** 应用

网络数据集

我们将使用第二章构造和探索图中引入的相同的数据集，包括社交自我网络，email 图以及食品复合网络。

图可视化

Spark 和 GraphX 没有提供任何内置的数据可视化功能，因为其焦点是数据处理。然而对数据分析来说，一图胜千言。接下来的章节我们会构造 Spark 应用用于可视化以及图的连通性分析。我们将靠第三方库 **GraphStream** 画出网络，使用 **BreezeViz** 来绘制图的结构属性，比如度分布。这些库都不完美，有限制，但是相对稳定，简单易用。因此我们就用它们来研究本章中的示例图。



目前，不请求大量的计算资源来绘制大规模的网络的图可视化引擎和库仍然有瓶颈。例如，流行的网络分析软件 SNAP 当前依赖于 GraphViz 引擎来画网络，但只能画小到中等大小的网络。Gephi 是另一个做交互式网络可视化的工具。尽管它有些很赞的特性，比如多层级和内置的 3D 渲染引擎，但是仍然有很高的 CPU 和内存需求。为了绘制标准图，有个新项目 Apache Zeppelin，提供了基于 web 的笔记本用于交互式数据分析和可视化。同时它内置了和 Spark 的集成。更多信息可访问其官方网站。

安装 GraphStream 和 BreezeViz 库

我们来开始安装第三方库及其依赖到 \$SPARKHOME/lib 目录里。GraphStream 是个超棒的 Java 库，能提供动态网络的可视化，可以随着时间演变。而我们的目标，只是要显示静态网络，所以我们只需要下载两个 jar 包：核心库 gs-core-1.2.jar 和 UI 库 gs-ui-1.2.jar。可以从下面的库中下载：

- <https://oss.sonatype.org/content/repositories/releases/org/graphstream/gstream/gstream-core/1.2/>

- <https://oss.sonatype.org/content/repositories/releases/org/graphstream/gstream/gstream-gs-ui/1.2/>

放置这两个 jar 包到项目根目录下的 lib 目录里。接下来，再从下面的库中下载 breeze_2.10-0.9.jar 和 breeze-viz_2.10-0.9.jar:

- http://repo.spring.io/libs-release-remote/org/scalanlp/breeze_2.10/0.9/
- http://repo1.maven.org/maven2/org/scalanlp/breeze-viz_2.10/0.9/

由于 BreezeViz 是个 Scala 库，依赖于另一个库 JFreeChart，所以你也需要安装 jcomm-1.0.16.jar 和 jfreechart-1.0.13.jar。这些 jar 包可以从下面库中找到:

- <https://repository.jboss.org/nexus/content/repositories/thirdparty-releases/jfree/jcommon/1.0.16/>
- <http://repo1.maven.org/maven2/jfree/jfreechart/1.0.13/>

下载完所有这些 jar 包以后，全都复制到项目根目录的 lib 目录下。现在就可以从 Spark shell 画你的第一个图啦。

可视化图数据

打开终端，设置当前目录为\$SPARKHOME，启动 Spark shell。这次你需要用—jar 选项指定第三方库:

```
$ ./bin/spark-shell --jars \
lib/breeze-viz_2.10-0.9.jar,\
lib/breeze_2.10-0.9.jar,\
lib/gstream-core-1.2.jar,\
lib/gstream-ui-1.2.jar,\
lib/jcommon-1.0.16.jar,\
lib/jfreechart-1.0.13.jar
```

或者，你也可以用下面的短命令少敲几下:

```
$. ./bin/spark-shell --jars \
$(find "." -name '*.jar' | xargs echo | tr ' ' ',')
```

不同于一次指定一个 jar 包，上面的命令会加载所有的 jar 包。

第一个例子，我们将可视化前一章见到过的社交自我网络。首先，需要导入 `GraphStream` 类，如下：

```
scala> import org.graphstream.graph.{Graph => GraphStream}
import org.graphstream.graph.{Graph=>GraphStream}
scala> import org.graphstream.graph.implementations._
import org.graphstream.graph.implementations._
```

重命名 `org.graphstream.graph.Graph` 到 `GraphStream` 很重要，可以避免和 `GraphX` 中的 `Graph` 类的名字空间冲突。

接下来，用 `Graph.fromEdges` 加载社交自我网络的数据，跟前一章一样。之后，创建一个 `SingleGraph` 对象：

```
// Create a SingleGraph class for GraphStream visualization
val graph: SingleGraph = new SingleGraph("EgoSocial")
```

`SingleGraph` 是 `GraphStream` 的抽象，能提供图数据的维护和视觉化能力。具体来说，我们可以调用 `SingleGraph` 对象的 `addNode` 和 `addEdge` 方法来增加网络节点和连接。我们也可以对图或者每个独立的边和节点调用 `addAttribute` 方法设置属性值。`GraphStream` 最酷的一点是它可以用类似 `CSS` 的样式表控制图中元素的显示，将图的结构和可视化清晰的分离。实践起来也超简单噢。所以，我们创建一个名为 `stylesheet` 的文件放到一个新的目录 `./style/` 里。插入下面几行到样式表中：

```
node {
  fill-color: #a1d99b;
  size: 20px;
  text-size: 12;
  text-alignment: at-right;
  text-padding: 2;
  text-background-color: #fff7bc;
}
edge {
  shape: cubic-curve;
  fill-color: #dd1c77;
  z-index: 0;
  text-background-mode: rounded-box;
  text-background-color: #fff7bc;
  text-alignment: above;
  text-padding: 2;
}
```

上面的样式表用选择器 `node` 和 `edge` 描述了图元素的视觉风格,用键值对指定了它们的视觉属性。本例中,我们设置了节点和边的颜色和形状以及文本属性。

GraphStream 中用到的样式表属性的详尽参考见 http://graphstream-project.org/doc/Tutorials/Graph-Visualisation_1.1/。

样式表一准备好,我们就可以连接到 SingleGraph 对象的图了:

```
// Set up the visual attributes for graph visualization
graph.addAttribute("ui.stylesheet","url(file:./style/stylesheet)")
graph.addAttribute("ui.quality")
graph.addAttribute("ui.antialias")
```

最后两行只是通知渲染引擎偏好质量而不是速度。接下来,我们不得不重新加载之前章节构建的图。为避免重复,我们忽略图构建的部分。然后,加载社交网络的 VertexRDD 和 EdgeRDD 到 GraphStream 对象中,代码如下:

```
// Given the egoNetwork, load the graphX vertices into GraphStream
for ((id,_) <- egoNetwork.vertices.collect()) {
  val node = graph.addNode(id.toString).asInstanceOf[SingleNode]
}

// Load the graphX edges into GraphStream edges
for (Edge(x, y, _) <- egoNetwork.edges.collect()) {
  val edge = graph.addEdge(x.toString ++ y.toString, x.toString, y.toString,
    true).
    asInstanceOf[AbstractEdge]
}
```

要添加顶点,只需要以字符串参数传入顶点 ID。对于边,需要传入四个参数给 `addEdge` 方法。第一个是每条边的字符串标识。由于这个标识在原始数据集或者 GraphX 的图对象中不可用,我们必须创建一个。那么,最简单的方案就是拼接每条边上连着的顶点 ID。



在前面的代码中，我们不得不用一个小技巧来避免我们的 Scala 代码和 GraphStream Java 库的互用性问题。就像 GraphStream 的 API `org.graphstream.graph.`

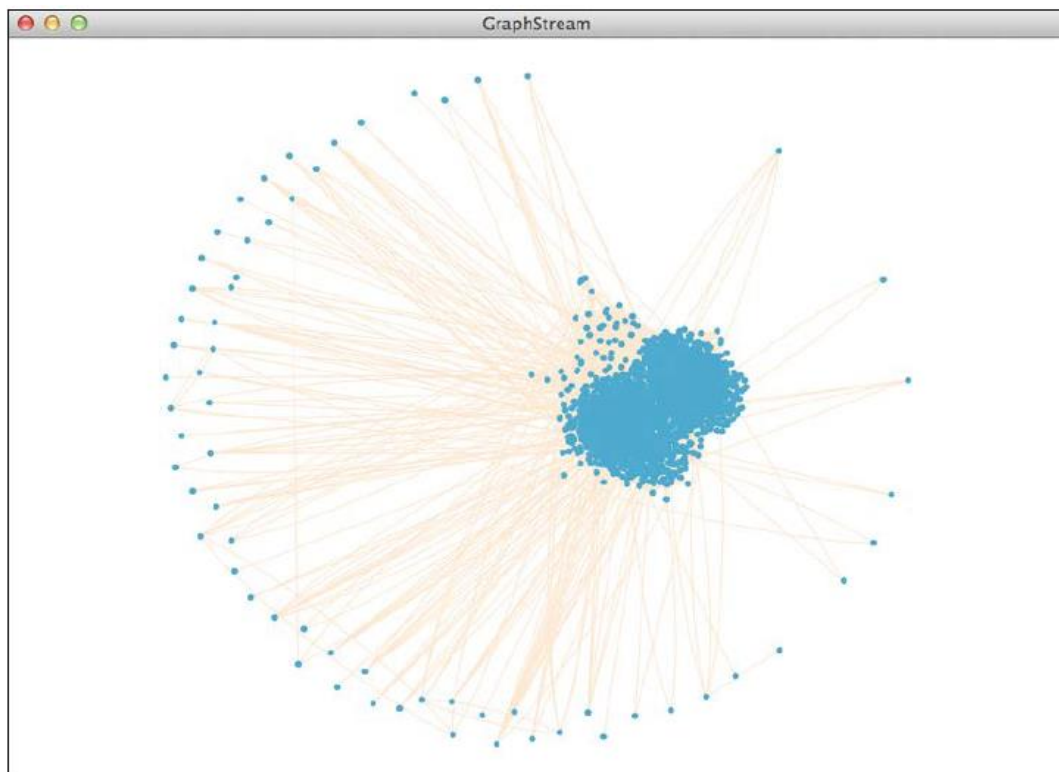
`implementations.AbstractGraph` 所描述的一样，`addNode` 和 `addEdge` 方法分别返回顶点和边。然而，GraphStream 是第三方 Java 库，我们不得不用 `asInstanceOf[T]` 方法迫使 `addNode` 和 `addEdge` 的返回类型做转换，`T` 分别是 `SingleNode` 和 `AbstractEdge`。那么，如果我们忽略这个显式类型转换会发生什么呢？你会得到一个奇怪的异常：

```
java.lang.ClassCastException:  
org.graphstream.graph.implementations.SingleNode  
cannot be cast to scala.runtime.Nothing$
```

现在干嘛？唯一要做的就是显示出这个社交自我网络图。只要对图调用 `display` 方法：

```
graph.display()
```

瞧~，你现在应该看到新窗口中的网络图了，就像这样：





如果没有显示颜色在上面，你应该检查当设置图的属性 `ui.stylesheet` 时，样式表文件的路径是否正确。

绘制度的分布

从这个显示的画面中，每个人在自我网络中看起来都是一个孤立的点或者连接到一大群可变的朋友。我们可以通过绘制网络的度分布来进一步分析这个问题。有 `spark-shell` 的帮助要完成这个太容易了。确保你已经从 `JFreeChart` 和 `Breeze` 中导出了一些类：

```
import org.jfree.chart.axis.ValueAxis
import breeze.linalg._
import breeze.plot._
```

接下来，我们将利用第二章 构建和探索图中创建的 `degreeHistogram` 函数。为方便阅读，下面是它的定义：

```
def degreeHistogram(net: Graph[Int, Int]): Array[(Int, Int)] =
  net.degrees.map(t => (t._2, t._1)).
    groupByKey.map(t => (t._1, t._2.size)).
    sortBy(_._1).collect()
```

从度的直方图上，我们可以看到度的分布，这是整个网络中节点的度的概率分布。为此，我们就用所有节点的总数来标准化，这样一来，度的概率加起来就是 1：

```
val nn = egoNetwork.numVertices
val egoDegreeDistribution = degreeHistogram(egoNetwork).map({case (d,n) =>
  (d,n.toDouble/nn)})
```

要显示度的分布图，我们首先要创建一个 `Figure` 对象 `f` 和两个绘图对象 `p1` 和 `p2`。下面代码中，`p1 = f.subplot(2,1,0)`，`p2 = f.subplot(2,1,1)`，指定了 `f` 的两个子绘图区域且 `p1` 在 `p2` 之上。实际上，子区域前两个参数是图中的行和列，而第三个参数是指明子区域的索引，从 0 开始：

```
val f = Figure()
val p1 = f.subplot(2,1,0)
val x = new DenseVector(egoDegreeDistribution map (_._1.toDouble))
val y = new DenseVector(egoDegreeDistribution map (_._2))
```

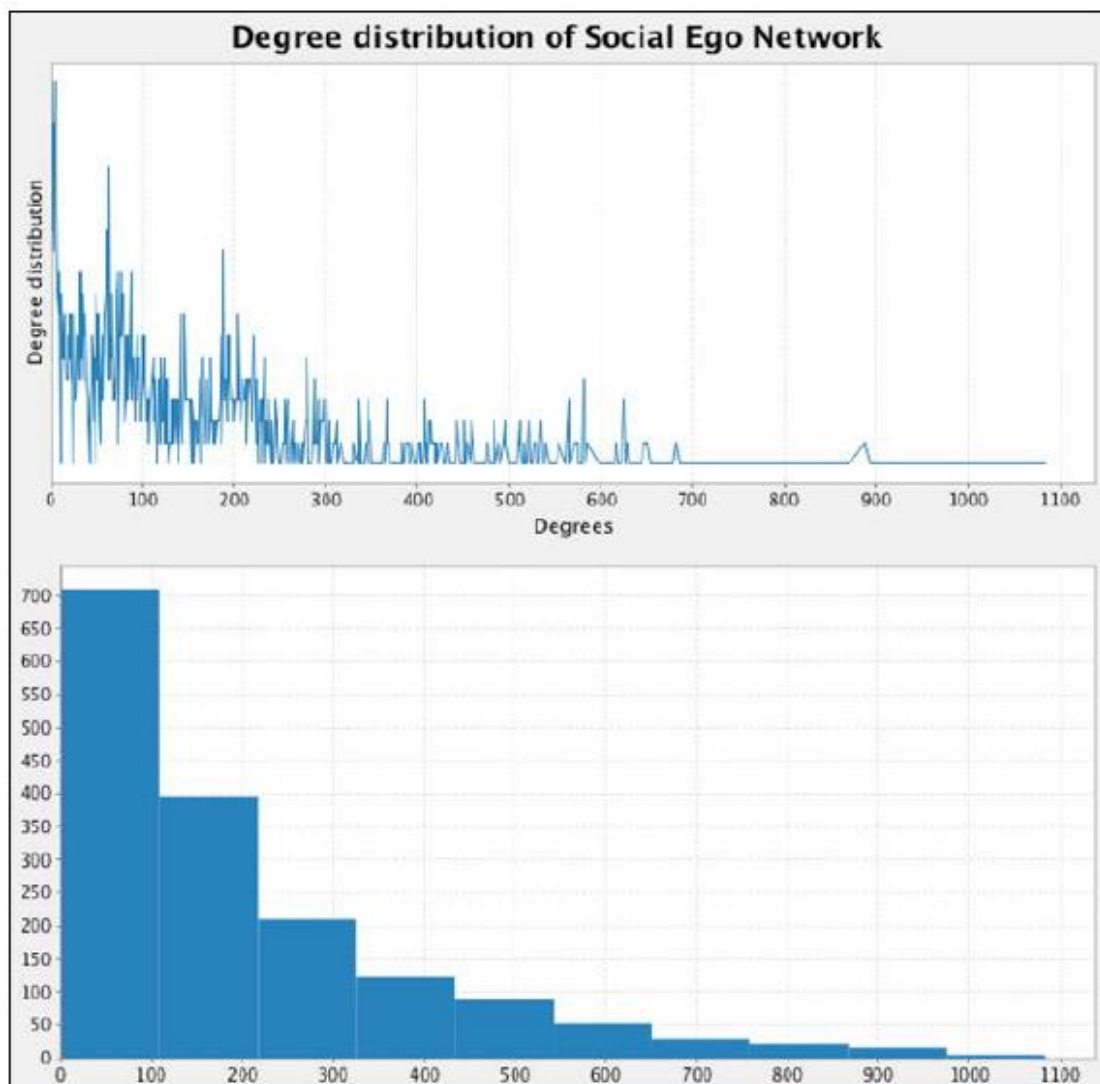
```

p1.xlabel = "Degrees"
p1.ylabel = "Distribution"
p1 += plot(x, y)
p1.title = "Degree distribution of social ego network"
val p2 = f.subplot(2,1,1)
val egoDegrees = egoNetwork.degrees.map(_._2).collect()

p1.xlabel = "Degrees"
p1.ylabel = "Histogram of node degrees"
p2 += hist(egoDegrees, 10)

```

接着，这些代码就会显示自我网络的度分布和度的频度。



分析网络的连通性

接下来，我们将从视觉上来考察和分析食品网络的连通性。用前一章中的步骤重新加载原料和化合物数据集。之后，创建一个 `GraphStream` 对象：

```
// Create a SingleGraph class for GraphStream visualization
val graph: SingleGraph = new SingleGraph("FoodNetwork")
```

然后，设置图的 `ui.stylesheet` 属性。由于食品网络是个二分图，那么用两个不同颜色来显示节点效果棒棒哒。我们来创建一个新的样式表，同时，减小节点的尺寸并隐藏文本属性：

```
node {
  size: 5px;
  text-mode: hidden;
  z-index: 1;
  fill-mode: dyn-plain;
  fill-color: "#e7298a", "#43a2ca";
}
edge {
  shape: line;
  fill-color: #fee6ce;
  arrow-size: 2px, 1px;
  z-index: 0;
}
```



样式表中颜色的值是用#设置成十六进制的。你可以从这个超棒的 **ColorBrewer** 调色板中选择你喜欢的颜色，网址是 <http://colorbrewer2.org/>

现在，我们再次从 `foodNetwork` 加载顶点和边到 `GraphStream` 对象里，使用 `addNode` 和 `addEdge` 方法。这次，我们将根据它是原料还是化合物来动态设置节点的颜色：

```
// Load the graphX vertices into GraphStream nodes
for ((id:VertexId, fnn:FNNNode) <- foodNetwork.vertices.collect())
{
  val node = graph.addNode(id.toString).asInstanceOf[SingleNode]
  node.addAttribute("name", fnn.name)
  node.addAttribute("ui.label", fnn.name)
  if (fnn.isInstanceOf[Compound])
    node.addAttribute("ui.color", 1: java.lang.Double)
```

```

else if(fnn.isInstanceOf[Compound])
  node.addAttribute("ui.color",0: java.lang.Double)
}

```



你也许会问问自己在用 `addNode` 加载节点时为什么要用 `isInstanceOf[T]` 来确定节点的类型。为什么我们不用 Scala 超棒的模式匹配特性？我们可以在单独的 Spark 程序里这么用，但是当前在 Spark shell 里不能使用对 case class 的模式匹配。这就是为什么要用 `isInstanceOf[T]` 的原因。

加载食品网络的节点跟加载自我网络几乎一样。唯一的区别是设置不同的颜色到节点。以类似的方式，加载边到 `GraphStream` 对象中：

```

// Load the graphX edges into GraphStream edges
for (Edge(x,y,_) <- foodNetwork.edges.collect()) {
  val edge = graph.addEdge(x.toString ++ y.toString,\
    x.toString, y.toString,
    true).
  asInstanceOf[AbstractEdge]
}

```

要显示食品网络，就调用 `graph.display()`。你会看到类似这样的图：



从这幅图中，我们看到许多原料共享相同的化合物，而有些化合物只在某些原料中。类似于社交自我网络，这个网络由一些孤立点和一些庞大的连通节点组成。这就引出了我们的下一个主题，如何度量图的连通性。

寻找连通组件

在网络中，如果图中两个节点之间有一条路径，这两个节点就是连通的。如果所有的节点之间都是连通的，那么这个网络被称为连通的。否则的话，一个不连通的图就会有许多的组件，而它们每一个都是连通的。要找到图中的连通组件在 GraphX 里很容易，使用 `connectedComponents` 方法：

以食品网络为例，我们可以验证它有 27 个组件：

```
// Connected Components
scala> val cc = foodNetwork.connectedComponents()
cc: org.apache.spark.graphx.Graph[VertexId,Int]

// Number of components
scala> cc.vertices.map(_._2).collect.distinct.size
res: Int = 27
```

上面给出了 `cc` 的类型，我们看到它返回了另一个有相同节点数的图。属于同一个组件的顶点有相同的属性，其值是组件中最小的顶点 ID。换句话说，每个节点的属性标识了它归属的组件。我们来看看这些组件标识：

```
scala> cc.vertices.map(_._2).distinct.collect
res6: Array[org.apache.spark.graphx.VertexId] = Array(892, 0, 1344, 528, 468, 392, 960, 409, 557, 529, 585, 1105, 233, 181, 481, 1146, 970, 622, 1186, 514, 1150, 511, 47, 711, 1211, 1511, 363)
```

现在，假设我们想以降序列出组件和其节点的数量。为此，我们可以调用 Spark 的 PairedRDD 的 `groupBy` 和 `sortBy`：

```
scala> cc.vertices.groupBy(_._2).
  map((p => (p._1,p._2.size))).
  sortBy(x => x._2, false).collect()
res: Array[(VertexId, Int)] = Array((0,2580), (528,8), (1344,3), (392,3), (585,3), (481,3), (892,2), (960,2), (409,2), (557,2), (529,2), (1105,2), (181,2), (970,2), (622,2), (1186,2), (1150,2), (511,2), (47,2), (711,2), (1511,2), (363,2), (468,1), (233,1), (1146,1), (514,1), (1211,1))
```


这个巨大的组件有 2580 种原料和化合物节点，它们中最小的顶点 ID 是 0。一般来说，我们可以定义一个函数，参数是一个连通组件的图，返回最大连通图中最小的顶点 ID 和节点数

```
def largestComponent(cc: Graph[VertexId, Int]): (VertexId, Int) =  
  cc.vertices.map(x => (x._2,x._1)).  
  groupBy(_._1).  
  map(p => (p._1,p._2.size)).  
  max()(Ordering.by(_._2))
```

在函数中，我们对组件图的顶点按顶点 ID 分组。然后，映射每个组件为键值对，其中键为组件 ID，值为组件的节点数。最后，使用 reduce 操作调用 max 函数返回一个键值对，对应最大的那个组件。在前面例子中，我们不得不传给 max 方法两个列表参数。第一个总是空，而第二个则是暗示排序。为了对值进行排序，我们必须传右边的给 max 排序为 Ordering.by(_._2)。



除了 GraphX 的图相关操作意外，Spark 的通用 RDD 和键值 RDD 操作对某些任务也会有用。上面这个函数就是个标准的示例，一连串数据操作都完全是 Spark 的 RDD 和键值 RDD 操作。更多的细节请看 Spark 和 GraphX 的 API 手册：<http://spark.apache.org/docs/1.1.0/api/scala/index.html#org.apache.spark.package>

三角关系计数以及聚合系数计算

接下来，我们要用安然 Email 数据集来讲解用三角关系和聚合系数分析图的连通性。三角关系是指连接了三个节点的子图。计算通过每个节点的三角关系的数量有助于量化图的连通性。特别是，计算聚合系数要求计算三角关系数量来度量网络中每个节点的邻居的局部密度。

目前，在 Spark 中对输入的图进行三角关系计数的实现有个限制。特别是，输入图的边应该有个标准化的方向；这里的标准化是指源 ID 参数必须要小于目标 ID 参数。对于这个 Email 图，这暗示着两个人之间应该最多只有一个方向的连接。这个限制其实没多那么严重，因为我们仍然可以假定 Email 图中的每个连接都表示两人之间的双向通信。我们可以通过过滤掉源 ID 大于目标 ID 的边来加以约束。除了这个限制外，输入图还必须被使用 `partitionBy` 进行过分区。因此，我们可以这么来加载 Email 图：

```
val emailGraph =  
  GraphLoader.edgeListFile(sc, "./data/emailEnron.txt").  
  subgraph(epred = t => t.srcId < t.dstId).  
  partitionBy(PartitionStrategy.RandomVertexCut)
```

一旦安然 Email 图加载了，就可以计算三角关系数了。

```
scala> emailGraph.triangleCount()  
res: Graph[Int,Int]  
scala> val triangleCounts = emailGraph.triangleCount().vertices  
triangleCounts:VertexRDD[Int]
```

和 `connectedComponent` 类似，`triangleCount` 算法返回一个有相同节点数的新图。不过，三角关系数量变成了顶点的属性。

挺容易的吧？现在，我们来计算安然 Email 图的局部聚合系数。首先，定义一个函数来计算特定点的聚合系数。对于给定点，聚合系数获取的是该点的网络局部密度。邻居节点越密集，局部聚合系数就越接近 1。可以通过下面的函数计算：


```
def clusterCoeff(tup: (VertexId, (Int,Int))): (VertexId, Double) =  
  tup match {case (vid, (t, d)) =>  
    (vid, (2*t.toDouble/(d*(d-1))))  
  }
```

`clusterCoeff` 的参数是个二元组，由我们要计算的节点的顶点 ID 和另一个由节点的三角关系数量和度数组成的二元组构成。返回的是聚合系数和顶点 ID 的二元组。实际上，给定点的局部聚合系数就是该点的每一对邻居节点连接的概率估计。因此，该系数可以计算为节点的邻居间的总连接数，也就等于通过该节点的三角关系数，与邻居间所有可能的连接数的比值。

由此，我们可以计算所有节点的聚合系数：

```
def clusterCoefficients(graph: Graph[Int,Int]):  
  RDD[(VertexId, Double)] = {  
    val gRDD: RDD[(VertexId, (Int, Int))] =  
      graph.triangleCount().vertices join graph.degrees  
    gRDD map clusterCoeff  
  }
```

最后这一个函数带了一个图参数作为输入，然后返回了一个二元的 RDD 集合，其元素包含顶点标识和对应的局部聚合系数。

 这个对于给定点的局部聚合系数的公式仅当它的度数，也就是邻居数，大于 1 时可用。如果节点的度数是 1 或者 0，clusterCoeff 函数将返回 NaN 值。因此，当我们想计算网络的平均或者全局聚合系数时，我们必须先检查网络中有些点是孤立的。我们不但要过滤掉叶子和孤立节点，而且还要调整全局聚合系数公式来避免邻近聚合的偏差。

现在我们用前面的函数来计算 email 图的聚合系数：

```
scala> val coeffs = clusterCoefficients(emailGraph)  
scala> coeffs.take(10)  
res: Array[(VertexId, Double)] = Array((19021,0.9), (28730,1.0),  
(23776,1.0), (31037,0.0), (34207,NaN), (29127,1.0), (9831,NaN),  
(5354,0.0380952380952381), (32676,0.46153846153846156), (4926,1.0))
```

我们看到，对于有些节点，返回的聚合系数是 NaN 值。实际上，这种情况在这 36692 个节点中有 25481 个：

```
// Check the NaN values.  
scala> coeffs.filter (x => !x._2.isNaN).count  
res: Long = 25481
```

要纠正这个情况，在计算平均聚合系数时我们要过滤掉这些节点：

```
// Calculate the adjusted global cluster coefficient  
scala> val nonIsolatedNodes = coeffs.filter(x => !x._2.isNaN)  
nonIsolatedNodes: RDD[(VertexId, Double)]  
scala> val globalCoeff =  
  nonIsolatedNodes.map(_._2).sum / nonIsolatedNodes.count  
globalCoeff:  
Double = 0.7156424032347621
```

网络中心性和 PageRank

之前，我们已经用到了网络的度的分布和聚合系数来理解网络是如何连接的。特别是，我们学会了如何找到最大的连通组件和度数最高的节点。然后，我们使网络可视化，看到了一些节点由于被许多节点连接而更有机会成为网络的中心。在某种意义上，节点的度数可以解释为一种中心性测度已确定该节点相对于网络中其余节点的重要性。在本节，我们将介绍一种不同的中心性测度，即 **PageRank** 算法，对网络中的节点排名很有用。



存在许多对图的中心性的其它测度值。例如，中介中心性对于信息流问题就很有用。给定一个节点，其中介值就是从所有的节点到所有其它节点的最短路径中经过该点的数量。不同于 **PageRank**，中介中心性对于战略性的处在最短路径上，连接着一对其它节点的那些节点赋予较高的得分。其它的一些测度值还有连接中心性和 **Katz** 中心性。**GraphX** 中没有预定义的算法来计算这些值。原因之一是精确计算中介中心性太过于复杂。因此，仍然需要开发近似算法，并且这也是对当前 **GraphX** 库进行扩展的极好的开源贡献哈。

PageRank 如何运转

PageRank 是 Google 的 web 搜索引擎获得难以置信的成功之后广为人知的算法。为了响应每一个查询请求，Google 想要首先显示重要的网页。简单的说，**PageRank** 就是给每个网页赋一个概率值。从长远来看，得分越高的节点，用户就越有可能访问该网页。

为了得到最终的 **PageRank** 分值，算法模拟了对网页图的随机访问的行为。在每一步，上网者可以访问当前链接的页面或者是随机跳到另一个页面（不必是邻近的页面）。这是通过图结构指定的转换概率来完成。例如，一个有 1000 个节点的网页图将对应一个 1000 乘以 1000 的转换概率矩阵。矩阵第 i 行第 j 列的元素值为 $1/k$ ，表示 j 页面有 k 个链出页面，其中有一个是 i 页面。否则的话，值为 0。**PageRank** 算法从一个随机节点出发，在每一步更新 **PageRank** 分值。

该算法的简略实现如下：

```
var PR = Array.fill(n)(1.0)
val oldPR = Array.fill(n)(1.0)
while( iter <= maxIter || max(abs(PR - oldPr)) > tol) {
  swap(oldPR, PR)
  for(i <- 0 until n) {
    PR[i] = d + (1 - d) * inNbrs[i].map(j => oldPR[j] / outDeg[j]).sum
  }
}
```

上面代码中， d ¹就是个随机的重置概率，默认值为 0.15。后面的 `inNbrs[i]` 是链接到 `i` 的邻居的集合，而 `outDeg[j]` 则是顶点 `j` 的出度。

更新的第一项是由于上网者会以概率 d 略过邻居页面而跳到一个随机的页面。更新每个页面重要度的第二项是基于前一轮所有链接到它的邻居页面的得分。这个过程不断重复，直到 PageRank 收敛到一个固定值，或者达到迭代的最大次数。

在 GraphX 中，有两个 PageRank 算法的实现。第一个实现使用 Pregel 接口，运行 PageRank 一个固定的迭代次数 `numIter`。第二个实现用的是标准的 Graph 接口，运行 PageRank 直到 PageRank 值的变化小于指定的误差 `tol`。



这些 PageRank 算法利用了关于顶点的数据并行性。尤其是，Pregel 实现依赖于本地消息传递来更新 PageRank 值。另一点要指出的是返回的 PageRank 值没有标准化。因此，它们没有描述概率分布。此外，没有输入链接的页面也会有 PageRank 值为 α 。尽管如此，顶级页面仍然可以靠对返回的 PageRank 图的顶点的得分属性值排序来发现。

网页排名

这里，我们将使用一个新的数据集来示范 PageRank。第一个是来自 Notre Dame 大学的网页图。直连的边表示它们之间有超链接。使用 PageRank，我们将排名并找到最重要的页面。

¹ 译者注，这里原文 α 貌似笔误，应该就是指的上面对代码中的 d 。

数据集可以下载自 <http://snap.stanford.edu/data/web-NotreDame.html>。这个数据集率先使用于（Albert, Jeong & Barabasi, 1999）：

```
// Load web graph
val webGraph = GraphLoader.edgeListFile(sc,"./data/web-NotreDame.txt")

// Run PageRank with an error tolerance of 0.0001
val ranks = webGraph.pageRank(0.001).vertices

// Find the top 10 pages of University of Notre Dame
val topPages = ranks.sortBy(_._2, false).take(10)
```

Scala 构建工具回顾

(译注: *sbt* 和 *maven* 的用法, 这个就懒得翻了, 需要用的百度吧~)

4

转换和塑造图 达到你的需要

在本章，我们要学习用不同的操作集来转换图形。特别是，我们将提到些图的专用操作，可以改变图形元素的属性或者是图的结构。也就是说，这里我们用到的所有的操作都是对图进行调用并返回一张新图的方法。另外，我们也将用 `join` 方法合并图数据和其他数据集。使用真实的数据集，你将理解何时以及如何：

- 用属性操作修改顶点或边的属性
- 用结构化操作修改图的形状
- 连接另外的 RDD 到属性图

转换顶点和边的属性

`Map` 操作是分布式数据库或者说 `Spark` 中的 `RDD` 的核心方法。类似的，属性图有三个 `map` 操作，定义如下：

```
class Graph[VD, ED] {  
    def mapVertices[VD2](mapFun: (VertexId, VD) => VD2): Graph[VD2, ED]  
    def mapEdges[ED2](mapFun: Edge[ED] => ED2): Graph[VD, ED2]  
    def mapTriplets[ED2](mapFun: EdgeTriplet[VD, ED] => ED2): Graph  
      [VD, ED2]  
}
```


被调用在属性图上的这些方法中的每一个都带着顶点属性类型 **VD** 和边的属性类型 **ED**。同时，它们也还都带了一个用户定义的映射函数 **mapFun**，用于执行下面的动作之一：

- 对于 **mapVertex** 函数，**mapFun** 以二元组(VertexID, ED)为输入，返回一个转换过的顶点属性类型 **VD2**。
- 对于 **mapEdge** 函数，**mapFun** 带了一个 **Edge** 对象为输入，返回一个转换过的边属性类型 **ED2**。
- 对于 **mapTriplets** 函数，**mapFun** 带了一个 **EdgeTriplet** 对象为输入，返回一个转换过的边属性类型 **ED2**。



每种情况下，图的结构都保持原封不动，这表示这些映射操作绝不会改变顶点间的连接或者顶点的索引。这是这些操作相比起基本 **RDD** 操作的一个关键优点。尽管后者也可以被用来达到同样效果，但是前者还是要更高效些，感谢 **GraphX** 系统的优化。因此，如果你只是想改变图的属性而不是结构，你应该总是使用这三个映射操作。**mapEdge** 和 **mapTriplet** 的区别是，对于后者，边和顶点属性都可用于 **mapFun** 的三元组输入中来创建新的边属性。相对于此，**mapEdge** 的 **mapFun** 函数仅能访问边属性。

现在，让我们通过一些简单例子来实际看看吧。

mapVertices

考虑一个人与人之间的社交图，顶点有个类型 **Person**，边的类型为 **Link**。首先，我们来创建这些 **scala** 类型，如下：

```
case class Person(first: String, last: String, age: Int)
case class Link(relationship: String, duration: Float)
```

假设我们从一个叫 **people** 的 **VertexRDD** 和叫 **Links** 的 **EdgeRDD** 构造了图：

```
val inputGraph: Graph[Person, Link] = Graph(people, links)
```

如果我们需要，我们可以用 **mapVertices** 来转换 **people** 的属性仅包含名字：

```
val outputGraph: Graph[String, Link] =
  inputGraph.mapVertices(_._1, person => person.first + person.last)
```

现在，这个新图 `outputGraph` 有一个 `String` 类型的顶点属性，替代了 `Person`。而人们之间的连接保持不变。

mapEdges

类似的，假设我们只对关系类型感兴趣而不在乎持续时间。这一次，我们可以用 `mapEdges` 来改变边的属性，如下：

```
val outputGraph: Graph[Person, String] =  
    inputGraph.mapEdges(link => link.relationship)
```

mapTriplets

最后，假设我们想跟踪人们相识时的年龄，并将该信息加入到边的属性里。我们可以用 `mapTriplets` 函数来处理，如下：

```
val outputGraph: Graph[Person, (Int, Int)] =  
    inputGraph.mapTriplets(t => (t.srcAttr.age - t.attr.duration,  
    t.dstAttr.age - t.attr.duration))
```

如果我们想同时改变图的边和顶点属性，只要将 `mapEdges` 或者 `mapTriplets` 和 `mapVertices` 串起来就行了，因为这每个方法返回的都是属性图。

修改图的结构

`GraphX` 同样也提供了 4 个可用的方法来修改图的结构。这些方法签名如下：

```
class Graph[VD, ED] {  
    def reverse: Graph[VD, ED]  
  
    def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,  
        vpred: (VertexId, VD) => Boolean): Graph[VD, ED]  
  
    def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]  
  
    def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]  
}
```

reverse 操作

跟名字暗示的一样，`reverse` 操作符返回一个新图，所有边的方向都逆向了。没有修改任何顶点和边的属性，或者边的数量。而且，它的实现 没有产生任何数据移动或者复制。

subgraph 操作

列表中接下来是 `subgraph` 操作，用于图的过滤。它带了两个判定函数做参数，返回一个 `Boolean` 值。第一个判定函数 `epred` 带了一个 `EdgeTriplet` 参数，当三元组满足判定时返回 `true`。类似的，`vpred` 函数带了一个 `(VertexID, ED)` 二元组参数，当顶点满足判定时返回 `true`。

通过这些判定函数，`subgraph` 返回的图仅包含了满足顶点判定的节点和这些顶点间满足边判定的边。默认情况下，当不提供这俩判定函数时，顶点和边的判定都返回 `true`。这意味着我们可以仅传入一个顶点判定，或者一个边判定，或者两者都指定。如果只传入了顶点判定到 `subgraph`，那么两个连接着的顶点如果被过滤出去了，那它们连接的那条边自然也就被过滤出去了。

`subgraph` 操作对于不用计数的情况很好用。例如，一个实际中常见的情况是图中有孤立的点或者丢失了顶点属性的边。我们就可以用 `subgraph` 排除掉这些元素。另一种 `subgraph` 有用的情况是，当我们想去掉图中的一个中心时，比如，某个节点连接了太多节点。

看个实例，用 `subgraph` 来回答下面的社交网络中常见的问题：“我的朋友的朋友列表中的哪个人还不是我的朋友？”：

```
// Given a social network
type Name = String
class Person(name: Name, friends: List[Name])
val socialNetwork: Graph[Person, Int] = ...

// that I am part of
val me = Person(myName, myFriends)

// I want know my friends' friends that are not yet my friends
val potentialFriends = socialNetwork.subgraph(vpred =
  (_, p: Person) => !(me.friends contains p.name))
```



注意，我们没传入边的判定函数作为给 `subgraph` 的参数。因此，`scala` 会用 `epred` 的默认值，这个函数总是返回 `true`。另一方面，我们应该讲 `vpred` 以命名参数来传递，这样的话 `Scala` 就能知道哪个判定函数传入了或者漏了。

mask 操作

`mask` 操作同样是过滤被调用的图。相对于 `subgraph`，`mask` 没带判定函数的参数。但它带了另一个图参数。这样的话，表达式 `graph.mask(anotherGraph)` 通过返回一个图的方式构造了 `graph` 的一个子图，这个子图包含了同时在 `anotherGraph` 中发现的顶点和边。这可以和 `subgraph` 操作一起使用，基于另一个相关图的属性来过滤图。

考虑下面的这种情况，当我们想找到一个图中的连通子图，但是还想要去掉结果图中丢失了属性信息的顶点。那么，我们可以运行我们之前已经看到过的 `connectedComponent` 算法，并使用 `subgraph` 和 `mask` 操作一起得到期望的结果。代码如下：

```
// Run Connected Components
val ccGraph = graph.connectedComponents()

// Remove vertices with missing attribute values and the edges connected to
// them
val validGraph = graph.subgraph(vpred =
  (_, attr) => attr.info != "NA")

// Restrict the resulting components to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

GroupEdge 操作

`Spark` 的属性图允许任何连接的节点间可以有多条边。`GroupEdge` 操作是另一个结构化操作，合并了每一组节点间的重复的边变为单条边。为此，`groupEdge` 需要一个名为 `merge` 的函数参数，它带了一对类型为 `ED` 的边属性，合并为同类型的单个属性值。结果就是 `groupEdge` 返回的图和原来的图类型相同。本章后面会有个详细的实例来看 `groupEdge` 操作。

连接图数据集

除了前面的映射和过滤操作之外，GraphX 还提供了 API 用来连接 RDD 数据集和图。这可以用于当我们要为图中的顶点属性添加额外的信息或者是当我们想合并两个相关图的顶点属性的时候。这些情况都可以用下面的 `join` 操作完成。

joinVertices

下面是第一个操作 `joinVertices` 的方法签名：

```
def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD):  
  Graph[VD, ED]
```

它调用于 `Graph[VD, ED]` 对象，并且需要两个输入作为柯林化参数传入。第一个参数，`joinVertices` 连接一个图的顶点属性与一个类型为 `RDD[(VertexID, U)]` 的输入顶点表。第二个参数是用户自定义的 `map` 函数。这个 `map` 函数连接每个顶点原始的和传入的属性到一个新属性。新属性的返回类型必须和原属性相同。另外，在传入的 RDD 中没有匹配值的顶点保持原值。

outerJoinVertices

第二个连接操作是 `outerJoinVertices`，这是一个比 `joinVertices` 更通用的方法。其方法签名如下：

```
def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD,  
  Option[U]) => VD2): Graph[VD2, ED]
```

虽然 `outerJoinVertices` 同样期待一个顶点 RDD 参数和一个 `map` 函数的参数，但是这个 `map` 函数允许改变顶点属性类型。同时，原始图中所有的顶点都改变了，即使它们不在传入的 RDD 表中。

结果就是，`map` 函数带了一个 `Option` 类型参数 `Option[U]` 代替了 `joinVertices` 中简单的 `U` 类型。

实例 - 好莱坞电影的图

实例会有助于说明这些差异。为此，我们走进好莱坞，构建一个电影男女演员的小图。

```
scala> val actors: RDD[(VertexId, String)] = sc.parallelize(List(
  (1L, "George Clooney"),(2L, "Julia Stiles"),
  (3L, "Will Smith"), (4L, "Matt Damon"),
  (5L, "Salma Hayek")))
actors: RDD[(VertexId, String)]
```

图中的两个人如果在同一部电影出现过就会连接起来。每条边将包含电影名字。来加载信息到边的 RDD 中，就叫 movie:

```
scala> val movies: RDD[Edge[String]] = sc.parallelize(List(
  Edge(1L,4L,"Ocean's Eleven"),
  Edge(2L, 4L, "Bourne Ultimatum"),
  Edge(3L, 5L, "Wild Wild West"),
  Edge(1L, 5L, "From Dusk Till Dawn"),
  Edge(3L, 4L, "The Legend of Bagger Vance"))
)
movies: RDD[Edge[String]]
```

现在，我们可以构建电影的图了，看看里面是些什么：

```
scala> val movieGraph = Graph(actors, movies)
movieGraph: Graph[String,String]
```

```
scala> movieGraph.triplets.foreach(t => println(
  t.srcAttr + " & " + t.dstAttr + " appeared in " + t.attr))
```

George Clooney & Matt Damon appeared in Ocean's Eleven

Julia Stiles & Matt Damon appeared in Bourne Ultimatum

George Clooney & Salma Hayek appeared in From Dusk Till Dawn

Will Smith & Matt Damon appeared in The Legend of Bagger Vance

Will Smith & Salma Hayek appeared in Wild Wild West

目前，我们的顶点只包含了每个男女演员的名字：

```
scala> movieGraph.vertices.foreach(println)
```

```
(2,Julia Stiles)
```

```
(1,George Clooney)
```

```
(5,Salma Hayek)
```

```
(4,Matt Damon)
```

```
(3,Will Smith)
```

假设我们能访问一个演员个人档案的数据库。对于本例，我们迅速加载这样一个数据库到 RDD：

```
scala> case class Biography(birthname: String, hometown: String)
defined class Biography
```

```
scala> val bio: RDD[(VertexId, Biography)] = sc.parallelize(List(
  (2, Biography("Julia O'Hara Stiles", "NY City, NY, USA")),
  (3, Biography("Willard Christopher Smith Jr.", "Philadelphia, PA, USA")),
  (4, Biography("Matthew Paige Damon", "Boston, MA, USA")),
  (5, Biography("Salma Valgarma Hayek-Jimenez", "Coatzacoalcos, Veracruz,
    Mexico")),
  (6, Biography("José Antonio Domínguez Banderas", "Málaga, Andalucía, Spain")),
  (7, Biography("Paul William Walker IV", "Glendale, CA, USA"))
))
bio: RDD[(VertexId, Biography)]
```

我们要使用 `joinVertices` 来连接这些信息到我们的电影图中。为此，我们创建一个自定义函数追加家乡到男女演员的名字后面：

```
scala> def appendHometown(id: VertexId, name: String, bio: Biography): String =
  name + ":" + bio.hometown
```

```
appendHometown: (id: VertexId, name: String, bio: Biography)String
```

对于 `joinVertices`，记住映射函数要返回和原图的顶点属性相同的类型，这里是 `String`。现在可以连接个人档案到我们好莱坞电影图的顶点属性了：

```
scala> val movieJoinedGraph =
  movieGraph.joinVertices(bio)(appendHometown)
movieJoinedGraph: Graph[String,String]
```



```
scala> movieJoinedGraph.vertices.foreach(println)
```

```
(1,George Clooney)
(5,Salma Hayek:Coatzacoalcos, Veracruz, Mexico)
(2,Julia Stiles:NY City, NY, USA)
(4,Matt Damon:Boston, MA, USA)
(3,Will Smith:Philadelphia, PA, USA)
```

接下来，我们使用 `outerJoinVertices` 来看看差别。这一次我们将直接传入一个匿名的 `map` 函数连接名字和个人档案，那么返回的一对就是：

```
scala> val movieOuterJoinedGraph =
movieGraph.outerJoinVertices(bio)((_,name, bio) => (name,bio))

movieOuterJoinedGraph: Graph[(String, Option[Biography]), String]
```

注意，`outerJoinVertices` 是如何改变顶点属性类型从 `String` 到元组(`String`, `Option[Biography]`)的。现在，打印出这些顶点：

```
scala> movieOuterJoinedGraph.vertices.foreach(println)
(1,(George Clooney,None))
(4,(Matt Damon,Some(Biography(Matthew Paige Damon,Boston, MA, USA))))
(5,(Salma Hayek,Some(Biography(Salma Valgarma Hayek- Jimenez,Coatzacoalcos,
Veracruz, Mexico))))
(2,(Julia Stiles,Some(Biography(Julia O'Hara Stiles,NY City, NY, USA))))
(3,(Will Smith,Some(Biography(Willard Christopher Smith Jr.,Philadelphia, PA,
USA))))
```

跟前面提到的一样，即使传入给 `outerJoinVertices` 的 `bio` 数据集里没有 `George Clooney` 的个人档案，它的新属性也变成了 `None`，这对于 `Option[Biography]` 类型是个有效值。

某些情况下，这对于要提取可选值之外的其他信息会很方便。为此，我们可以用定义于 `Option[T]` 的方法 `getOrElse` 并为顶点提供一个默认的不出现在传入的顶点 RDD 中的新属性值：

```
scala> val movieOuterJoinedGraph = movieGraph.outerJoinVertices(bio)((_, name,
bio) =>
(name,bio.getOrElse(Biography("NA","NA"))))
movieOuterJoinedGraph: Graph[(String, Biography),String]
```

```
scala> movieOuterJoinedGraph.vertices.foreach(println)
(1,(George Clooney,Biography(NA,NA)))
(2,(Julia Stiles,Biography(Julia O'Hara Stiles,NY City, NY, USA)))
(5,(Salma Hayek,Biography(Salma Valgarma Hayek-Jimenez,Coatzacoalcos,
Veracruz, Mexico)))
(4,(Matt Damon,Biography(Matthew Paige Damon,Boston, MA, USA)))
(3,(Will Smith,Biography(Willard Christopher Smith Jr.,Philadelphia, PA, USA)))
```

当然，要为连接的顶点创建一个新的返回类型也是可能的。例如，我们可以创建一个新的类型 `Actor` 来生成新的图类型 `Graph[Actor, String]`，如下：

```
scala> case class Actor(name: String, birthname: String, hometown: String)
defined class Actor

scala> val movieOuterJoinedGraph = movieGraph.outerJoinVertices(bio)((_, name,
b) => b match {
    case Some(bio) => Actor(name, bio.birthname, bio.hometown)
    case None => Actor(name, "", "")
})
movieOuterJoinedGraph: Graph[Actor,String]
```

列示一下新图的顶点，我们得到了预期的结果：

```
scala> movieOuterJoinedGraph.vertices.foreach(println)
(4,Actor(Matt Damon,Matthew Paige Damon,Boston, MA, USA))
(1,Actor(George Clooney,,))
(5,Actor(Salma Hayek,Salma Valgarma Hayek-Jimenez,Coatzacoalcos, Veracruz,
Mexico))
(2,Actor(Julia Stiles,Julia O'Hara Stiles,NY City, NY, USA))
(3,Actor(Will Smith,Willard Christopher Smith Jr.,Philadelphia, PA, USA))
```

注意，没有为 Antonio Banderas 和 Paul Walker 创建新顶点，尽管他们出现在 `bio` 数据集的 RDD 中。因为他们不属于原图。



当调用前面提到的 `outerJoinVertices` 时,我们传入 `map` 函数,没有带类型声明。这是可选的,只要 `map` 函数的定义符合期待的输入和输出类型。

尽管对传入给 `joinVertices` 和 `outerJoinVertices` 的 RDD 数据集中的一个顶点有可能会不止一个值,但是只有一个会被用到。因此,建议 RDD 中的值唯一。

对于 `joinVertices` 和 `outerJoinVertices`,输出图的顶点是相同的。只是顶点属性会不一样。不会有新的顶点被创建,因为它们的角色只是从传入的 RDD 中连接信息到已存在的顶点。

VertexRDD 和 EdgeRDD 上的数据操作

所有之前我们见过的操作都是图操作。它们对图进行调用并返回一个新的图对象。在本节,我们将介绍转换 `VertexRDD` 和 `EdgeRDD` 集合的操作。这些集合的类型分别是 `RDD[(VertexID, ED)]` 和 `RDD[Edge[ED]]` 对应的子类型。

VertexRDD 和 EdgeRDD 的映射

首先要说的是, `mapValues` 函数带了一个 `map` 函数做为输入参数,转换 `VertexRDD` 中每个顶点属性值。然后,它返回一个新的 `VertexRDD` 对象,但是保存了原顶点的索引。`mapValues` 方法被重载过,所以 `map` 函数能接受类型为 `VD` 或者 `(VertexID, VD)` 的参数。新顶点的属性类型可以和 `VD` 不同:

```
def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]
def mapValues[VD2](map: (VertexID, VD) => VD2): VertexRDD[VD2]
```

为了演示,我们取出之前好莱坞明星的个人档案的 `VertexRDD` 集合:

```
scala> val actorsBio = movieJoinedGraph.vertices
actorsBio: VertexRDD[String]
```

```
scala> actorsBio.foreach(println)
(4,Matt Damon:Boston, Massachusetts, USA)
(1,George Clooney)
```

(5,Salma Hayek:Coatzacoalcos, Veracruz, Mexico)

(3,Will Smith:Philadelphia, Pennsylvania, USA)

(2,Julia Stiles:New York City, New York, USA)

现在，我们可以用 `mapValues` 来提取他们的名字到新的 `VertexRDD` 集合：

```
scala> actorsBio.mapValues(s => s.split(':')[0]).foreach(println)
```

(2,Julia Stiles)

(1,George Clooney)

(5,Salma Hayek)

(4,Matt Damon)

(3,Will Smith)

使用重载的 `mapValues` 方法，我们可以包括顶点 ID 到 `map` 函数的输入中，仍然得到类似结果：

```
scala> actorsBio.mapValues((vid,s) => s.split(':')[0]).foreach(println)
```

(1,George Clooney)

(5,Salma Hayek)

(3,Will Smith)

(4,Matt Damon)

(2,Julia Stiles)

同样的，也有一个转换 `EdgeRDD` 的 `mapValues` 方法：

```
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
```

类似的，`mapValues` 仅改变边的属性值。不会去掉或者增加边，也不会修改边的方向。


VertexRDD 的过滤

使用 `filter` 方法，我们同样可以过滤 `VertexRDD`。在不改变顶点索引的同时，`filter` 去掉不满足传入给 `filter` 的用户自定义判定条件的顶点。和 `mapValues` 相反，`filter` 没有被重载，所以判定函数类型必须是 `(VertexID, VD) => Boolean`。总结如下：

```
def filter(pred: (VertexId, VD) => Boolean): VertexRDD[VD]
```

除了 filter 之外，diff 操作也能过滤 VertexRDD 集合里的顶点。它以另一个 VertexRDD 集合为输入，从原集合去掉同时出现在输入集合中的顶点：

```
def diff(other: VertexRDD[VD]): VertexRDD[VD]
```

 GraphX 没有为 EdgeRDD 集合提供类似 filter 的操作，为过滤边可以用图操作 subgraph 直接而高效的达到目的。见前一节的 *修改图的结构*。

VertexRDD 的连接

下面的连接操作是为 VertexRDD 集合优化过的：

```
def innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2):  
VertexRDD[VD2]
```

```
def leftJoin[U, VD2](other: RDD[(VertexId, VD2)])(f: (VertexId, VD, Option[U]) =>  
VD2): VertexRDD[VD2]
```

第一个操作是 innerJoin，它带了 VertexRDD 和一个用户自定义函数为输入参数。用该函数，会连接在原 VertexRDD 和输入的 VertexRDD 中都出现的顶点。换句话说，innerJoin 返回顶点的交集，并通过 f 函数合并属性。

所以，给定来自 movieGraph 的顶点 RDD，将其与个人档案的 RDD 做 innerJoin 的结果将不包含 George Clooney，Paul Walker 和 José Antonio Domínguez Banderas。

```
scala> val actors = movieGraph.vertices  
actors: VertexRDD[String]
```

```
scala> actors.innerJoin(bio)((vid, name, b) => name + " is from " +  
b.hometown).foreach(println)  
(4,Matt Damon is from Boston, Massachusetts, USA)  
(5,Salma Hayek is from Coatzacoalcas, Veracruz, Mexico)  
(2,Julia Stiles is from New York City, New York, USA)  
(3,Will Smith is from Philadelphia, Pennsylvania, USA)
```

第二个操作 `leftJoin` 和定义在 `Graph[VD, ED]` 中的 `outerJoinVertices` 操作类似。除了输入 `VertexRDD` 集合意外，它同样带了类型为 `(VertexID, VD, Option[U]) => VD2` 的用户自定义函数 `f`。其结果集 `VertexRDD` 将同样包含和原 `VertexRDD` 相同的顶点。因为函数 `f` 的第三个输入参数是 `Option[U]`，所以它能够处理在原集合中的顶点没出现在输入结合中的情况。接着用之前的例子，我们可以类似的处理：

```
scala> actors.leftJoin(bio)((vid, name, b) => b match {  
  case Some(bio) => name + " is from " + bio.hometown  
  case None => name + "'s hometown is unknown"  
}).foreach(println)
```

```
(4,Matt Damon is from Boston, Massachusetts, USA)  
(1,George Clooney's hometown is unknown)  
(5,Salma Hayek is from Coatzacoalcos, Veracruz, Mexico)  
(2,Julia Stiles is from New York City, New York, USA)  
(3,Will Smith is from Philadelphia, Pennsylvania, USA)
```

EdgeRDD 的连接

在 `GraphX` 中存在一个连接操作 `innerJoin` 来连接两个 `EdgeRDD`：

```
def innerJoin[ED2, ED3](other: EdgeRDD[ED2])(f: (VertexId, VertexId, ED, ED2) =>  
  ED3): EdgeRDD[ED3]
```

和 `VertexRDD` 的 `innerJoin` 方法类似，除了输入函数的类型 `f: (VertexID, VertexID, ED, ED2) => ED3`。而且，`innerJoin` 和原 `EdgeRDD` 使用相同的分区策略。

逆转边的方向

在之前，我们已经看到 `reverse` 操作逆转图中的所有的边。当我们只想逆转图中的边的一个子集时，下面的 `EdgeRDD` 中定义的 `reverse` 方法就有用了：

```
def reverse: EdgeRDD[ED]
```

比如，我们都知道 `Spark` 中属性图必须有方向。唯一建模无向图的方法就是每条边添加一个逆向连接。这可以和下面的代码一样很轻松的办到。首先我们提取电影图中的边到名为 `movie` 的 `EdgeRDD` 中：

```
scala> val movies = movieGraph.edges  
movies: EdgeRDD[String,String]
```

```
scala> movies.foreach(println)
Edge(1,4,Ocean's Eleven)
Edge(3,5,Wild Wild West)
Edge(2,4,Bourne Ultimatum)
Edge(1,5,From Dusk Till Dawn)
Edge(3,4,The Legend of Bagger Vance)
```

然后，我们创建了有逆向连接的新的 EdgeRDD。接着，我们得到了两个 EdgeRDD 集合的并集的双向图：

```
scala> val bidirectedGraph = Graph(actors, movies union
movies.reverse)
```

通过打印新的边集合，我们看到这起作用了：

```
scala> bidirectedGraph.edges.foreach(println)
Edge(1,5,From Dusk Till Dawn)
Edge(3,4,The Legend of Bagger Vance)
Edge(3,5,Wild Wild West)
Edge(1,4,Ocean's Eleven)
Edge(2,4,Bourne Ultimatum)
Edge(4,1,Ocean's Eleven)
Edge(4,2,Bourne Ultimatum)
Edge(5,3,Wild Wild West)
Edge(4,3,The Legend of Bagger Vance)
Edge(5,1,From Dusk Till Dawn)
```



EdgeRDD[ED]是 RDD[Edge[ED]]的子类型，它使用 PartitionStrategy 中定义的某个分区策略将边组织到分区块中。边的属性和邻接结构都分开保存在每个分区里，所以当仅有边属性被改变时结构可以被重用。

在 Spark1.0 和 1.1 中，EdgeRDD 的类型签名处于优化的目的被改为 EdgeRDD[ED, VD]。从 Spark1.2 开始，在以不同的方式实现缓存优化时，签名又被切换回原来更简单的 EdgeRDD[ED]类型定义。

收集邻居信息

在做图计算时，我们也许想使用邻居的信息，比如邻居顶点的属性。

`collectNeighborIds` 和 `collectNeighbors` 这两个操作明显允许我们能这么干。

`collectNeighborIds` 仅收集每个邻居节点的顶点 ID 到 `VertexRDD` 中，而

`collectNeighbor` 还采集它们的属性：

```
def collectNeighborIds(edgeDirection: EdgeDirection):  
  VertexRDD[Array[VertexId]]  
def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexId,  
  VD)]]
```

这两个方法都是调用到属性图上，并传入一个 `EdgeDirection` 作为输入参数。

`EdgeDirection` 属性有四个可能的值：

- `Edge.Direction.In`：当指定这个选项，每个顶点只采集进入的连接邻居的属性。
- `Edge.Direction.Out`：每个顶点只采集从自己连出去的连接邻居的属性。
- `Edge.Direction.Either`：每个顶点采集所有邻居的属性。
- `Edge.Direction.Both`：每个顶点采集连出或连入到自己的邻居的属性。



为性能优化，最好是避免用这两个操作，而是重写计算代码使用下一章将要出现的更通用和更高效的 `aggregateMessages` 操作。其效率增益很大，尤其是在实现迭代的图并行算法时。但是对于简单图的一次性转换，使用 `collectNeighbor` 和 `collectNeighborIds` 也还好。

从食品网络到调料配对

在第二章 *构建和探索图* 中，我们展示了食品原料数据集并构建了一个连接每种食品原料和化合物的二分图。接下来我们要构建另一个仅由食品原料组成的图。在新图中，一对食品原料连接仅当它们共享至少一种化合物。这张新图我们称之为调料网络。之后我们能用这张图来根据经验用新的食物组合创建新的食谱。

让我们从第二章 *构建和探索图* 中构建的二分食品网络开始吧：


```
scala> val nodes = ingredients ++ compounds
scala> val foodNetwork = Graph(nodes, links)
foodNetwork: Graph[Node,Int]
```

要创建新的调料网络，我们需要知道哪些原料共享了化合物。这可以做到，首先在 `foodNetwork` 图中为每个化合物节点收集原料 ID。具体来说，我们收集并分组有相同化合物的原料 ID 到 RDD 集合，集合元素类型为二元组(化合物 ID, `Array`[原料 ID])。如下所示：

```
scala> val similarIngr: RDD[(VertexId, Array[VertexId])] =
foodNetwork.collectNeighborIds(EdgeDirection.In)
similarIngr: RDD[(VertexId, Array[VertexId])]
```

接下来，我们创建一个函数 `pairIngredients`，带一个同样的二元组参数(化合物 ID, `Array`[原料 ID])，并对数组 中每一对原料之间创建一条边。

```
def pairIngredients(ingPerComp: (VertexId, Array[VertexId])):
Seq[Edge[Int]] =
  for {
    x <- ingPerComp._2
    y <- ingPerComp._2
    if x != y
  } yield Edge(x,y,1)
pairIngredients:
(ingPerComp:(VertexId,Array[VertexId]))Seq[Edge[Int]]
```

完成以后，我们就可以为食品网络中共享化合物的每一对原料创建 `EdgeRDD` 集合，代码如下：

```
scala> val flavorPairsRDD: RDD[Edge[Int]] = similarIngr flatMap pairIngredients
flavorPairsRDD: RDD[Edge[Int]]
```

最后，我们创建了新的调料网络：


```
scala> val flavorNetwork = Graph(ingredients, flavorPairsRDD).cache
flavorNetwork: Graph[Node,Int]
```

我们打印 flavorNetwork 中的前 20 个三元组看看：

```
scala> flavorNetwork.triplets.take(20).foreach(println)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(9,Ingredient(peanut_butter,plant
derivative)),1)
((3,Ingredient(mackerel,fish/seafood)),(17,Ingredient(red_bean,vegetable)),1)
((3,Ingredient(mackerel,fish/seafood)),(17,Ingredient(red_bean,vegetable)),1)
((3,Ingredient(mackerel,fish/seafood)),(17,Ingredient(red_bean,vegetable)),1)
((3,Ingredient(mackerel,fish/seafood)),(17,Ingredient(red_bean,vegetable)),1)
((3,Ingredient(mackerel,fish/seafood)),(17,Ingredient(red_bean,vegetable)),1)
((3,Ingredient(mackerel,fish/seafood)),(17,Ingredient(red_bean,vegetable)),1)
((3,Ingredient(mackerel,fish/seafood)),(17,Ingredient(red_bean,vegetable)),1)
```

看起来鲭鱼，花生酱和红豆之间有些共同点。在我们尝试新的食谱前，我们来稍微调整下网络。注意，当一组原料共享不止一种化合物时，可能有多条重复的边。假设我们想将每组原料间的多条边分组合并成一条边，其中包含了每组原料之间共享化合物的个数。我们可以用 `groupEdges` 办到：

```
val flavorWeightedNetwork =  
  flavorNetwork.partitionBy(PartitionStrategy.EdgePartition2D).groupEdges((x,y)  
    => x+y)  
flavorWeightedNetwork: Graph[Node,Int]
```

 `GroupEdges` 要求图被重新分区，因为它假定相同的边在同一个分区里。因此，你必须在分组之前先调用 `partitionBy`。

现在，我们来打印 20 组共享了最多化合物的原料：

```
scala> flavorWeightedNetwork.triplets.  
  sortBy(t => t.attr, false).take(20).  
  foreach(t => println(t.srcAttr.name + " and " + t.dstAttr.name + " share " + t.attr + "  
  compounds."))
```

```
bantu_beer and beer share 227 compounds.  
beer and bantu_beer share 227 compounds.  
roasted_beef and grilled_beef share 207 compounds.  
grilled_beef and roasted_beef share 207 compounds.  
grilled_beef and fried_beef share 200 compounds.  
fried_beef and grilled_beef share 200 compounds.  
beef and roasted_beef share 199 compounds.  
beef and grilled_beef share 199 compounds.  
beef and raw_beef share 199 compounds.  
beef and fried_beef share 199 compounds.  
roasted_beef and beef share 199 compounds.  
roasted_beef and raw_beef share 199 compounds.  
roasted_beef and fried_beef share 199 compounds.  
grilled_beef and beef share 199 compounds.  
grilled_beef and raw_beef share 199 compounds.  
raw_beef and beef share 199 compounds.
```

raw_beef and roasted_beef share 199 compounds.

raw_beef and grilled_beef share 199 compounds.

raw_beef and fried_beef share 199 compounds.

fried_beef and beef share 199 compounds.

这没什么好惊讶的，烤箱的烤牛肉和烧烤的烤牛肉当然有非常多的共同成分。虽然这个例子并没有教我们更多的烹饪技巧，不过它说明了我们可以混合多个操作来改变图到我们想要的形式。

总结

大概来说，GraphX 提供了一些方法和操作来转换图的元素和结构。我们可以用图特定的操作转换一个图到新图。另外，我们还可以用特殊的方法处理 VertexRDD 和 EdgeRDD。除此之外，我们用连接方法合并了图数据和其他数据集。你可以用所有这些方法来整理新图的数据对其塑形来满足你特定的需要。

在下一章中，你将学习如何使用优化的方法来创建你自己的自定义的图操作，比如 aggregateMessage 和 mapReduceTriplets。