

# 《大数据技术原理与应用（第2版）》

<http://dbllab.xmu.edu.cn/post/bigdata>

温馨提示：编辑幻灯片母版，可以修改每页PPT的厦大校徽和底部文字

## 第11章 图计算

(PPT版本号：2017年2月版本)

林子雨

厦门大学计算机科学系

E-mail: [ziyulin@xmu.edu.cn](mailto:ziyulin@xmu.edu.cn) ▶▶

主页: <http://www.cs.xmu.edu.cn/linziyu>





# 本章配套教学视频

《大数据技术原理与应用（第2版）》  
《第11章 图计算》在线视频观看地址

<http://dblab.xmu.edu.cn/post/bigdata-online-course/#lesson12>

## 大数据技术原理与应用

**BIGDATA TECHNOLOGY AND APPLICATION**

打开大数据之门，遨游大数据世界





# 提纲

- 11.1 图计算简介
- 11.2 Pregel简介
- 11.3 Pregel图计算模型
- 11.4 Pregel的C++ API
- 11.5 Pregel的体系结构
- 11.6 Pregel的应用实例
- 11.7 Pregel和MapReduce实现PageRank算法的对比
- 11.8 Hama的安装和使用

本PPT是如下教材的配套讲义：

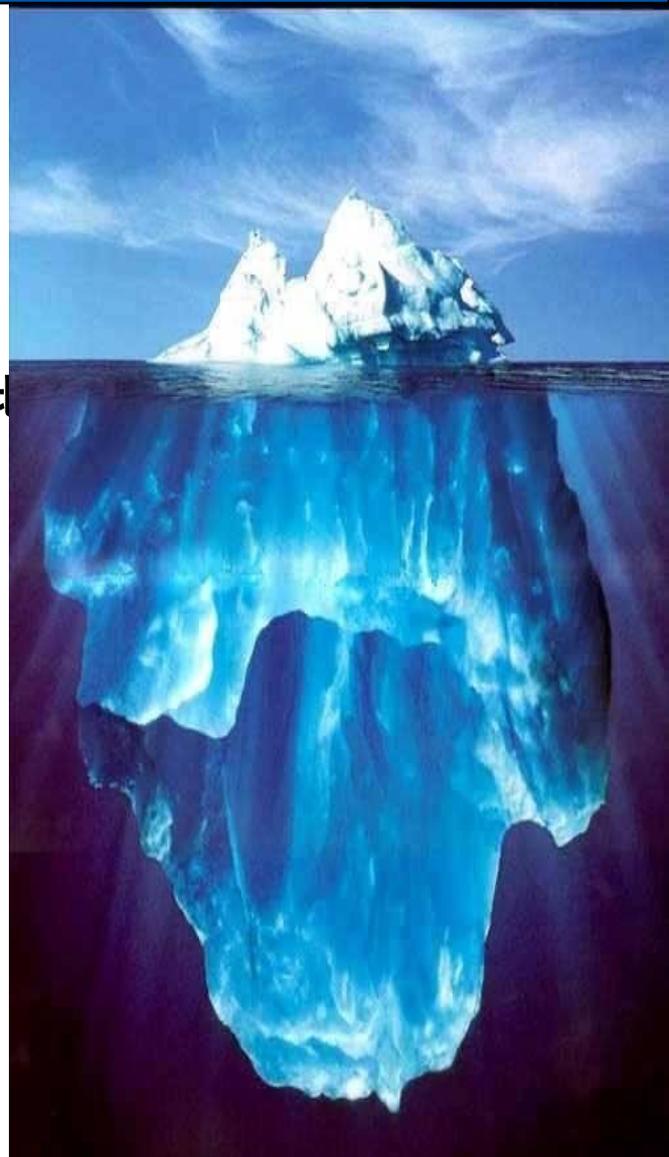
《大数据技术原理与应用  
——概念、存储、处理、分析与应用》  
(2017年2月第2版)

ISBN:978-7-115-44330-4

厦门大学 林子雨 编著，人民邮电出版社

欢迎访问《大数据技术原理与应用》教材官方网站：

<http://dbl原因lab.xmu.edu.cn/post/bigdata>





# 11.1 图计算简介

- 11.1.1 图结构数据
- 11.1.2 传统图计算解决方案的不足之处
- 11.1.3 图计算通用软件



## 11.1.1 图结构数据

- 许多大数据都是以大规模图或网络的形式呈现，如社交网络、传染病传播途径、交通事故对路网的影响
- 许多非图结构的大数据，也常常会被转换为图模型后进行分析
- 图数据结构很好地表达了数据之间的关联性
- 关联性计算是大数据计算的核心——通过获得数据的关联性，可以从噪音很多的海量数据中抽取有用的信息
  - 比如，通过为购物者之间的关系建模，就能很快找到口味相似的用户，并为之推荐商品
  - 或者在社交网络中，通过传播关系发现意见领袖



## 11.1.2

# 传统图计算解决方案的不足之处

很多传统的图计算算法都存在以下几个典型问题：

- (1) 常常表现出比较差的内存访问局部性
- (2) 针对单个顶点的处理工作过少
- (3) 计算过程中伴随着并行度的改变



## 11.1.2 传统图计算解决方案的不足之处

针对大型图（比如社交网络和网络图）的计算问题，可能的解决方案及其不足之处具体如下：

- **(1) 为特定的图应用定制相应的分布式实现：**通用性不好
- **(2) 基于现有的分布式计算平台进行图计算：**在性能和易用性方面往往无法达到最优
  - 现有的并行计算框架像MapReduce还无法满足复杂的关联性计算
  - MapReduce作为单输入、两阶段、粗粒度数据并行的分布式计算框架，在表达多迭代、稀疏结构和细粒度数据时，力不从心
  - 比如，有公司利用MapReduce进行社交用户推荐，对于5000万注册用户，50亿关系对，利用10台机器的集群，需要超过10个小时的计算
- **(3) 使用单机的图算法库：**比如BGL、LEAD、NetworkX、JDSL、Stanford GraphBase和FGL等，但是，在可以解决的问题的规模方面具有很大的局限性
- **(4) 使用已有的并行图计算系统：**比如，Parallel BGL和CGM Graph，实现了很多并行图算法，但是，对大规模分布式系统非常重要的一些方面（比如容错），无法提供较好的支持



## 11.1.3 图计算通用软件

- 传统的图计算解决方案无法解决大型图的计算问题，因此，就需要设计能够用来解决这些问题的通用图计算软件
- 针对大型图的计算，目前通用的图计算软件主要包括两种：
  - 第一种主要是基于遍历算法的、实时的图数据库，如 Neo4j、OrientDB、DEX和 Infinite Graph
  - 第二种则是以图顶点为中心的、基于消息传递批处理的并行引擎，如GoldenOrb、Giraph、Pregel和Hama，这些图处理软件主要是基于BSP模型实现的并行图处理系统



# 11.1.3 图计算通用软件

一次BSP(Bulk Synchronous Parallel Computing Model, 又称“大同步”模型)计算过程包括一系列全局超步（所谓的超步就是计算中的一次迭代），每个超步主要包括三个组件：

- 局部计算**：每个参与的**处理器**都有自身的计算任务，它们只读取存储在本机内存中的值，不同处理器的计算任务都是异步并且独立的
- 通讯**：处理器群相互交换数据，交换的形式是，由一方发起推送(put)和获取(get)操作
- 栅栏同步(Barrier Synchronization)**：当一个处理器遇到“路障”（或栅栏），会等到其他所有处理器完成它们的计算步骤；每一次同步也是一个超步的完成和下一个超步的开始

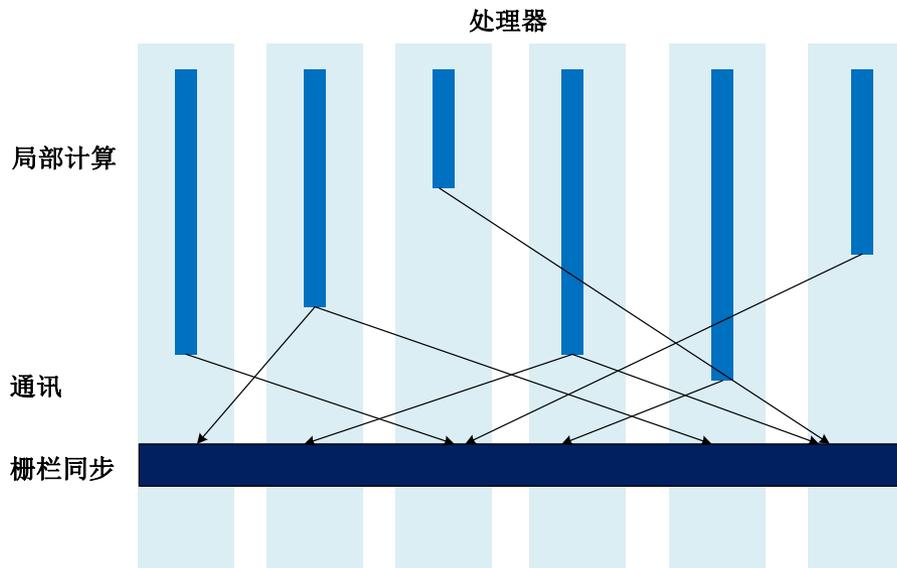
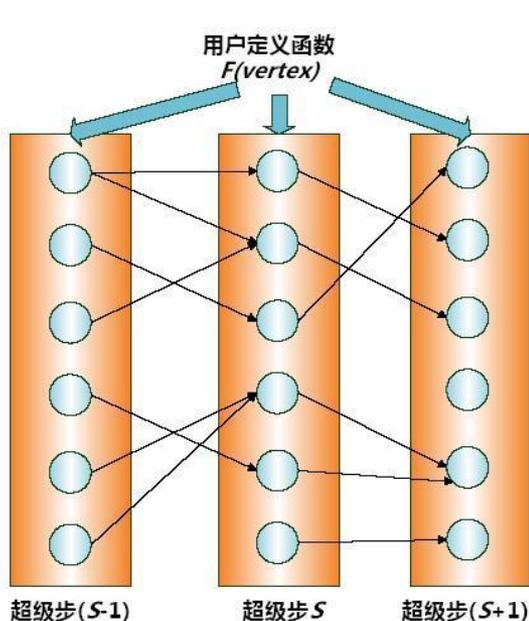


图9-1 一个超步的垂直结构图



## 11.2 Pregel简介

- 谷歌公司在2003年到2004年公布了GFS、MapReduce和BigTable，成为后来云计算和Hadoop项目的重要基石
- 谷歌在后Hadoop时代的新“三驾马车”——Caffeine、Dremel和Pregel，再一次影响着圈子与大数据技术的发展潮流
- Pregel是一种基于BSP模型实现的并行图处理系统
- 为了解决大型图的分布式计算问题，Pregel搭建了一套可扩展的、有容错机制的平台，该平台提供了一套非常灵活的API，可以描述各种各样的图计算
- Pregel作为分布式图计算的计算框架，主要用于图遍历、最短路径、PageRank计算等等



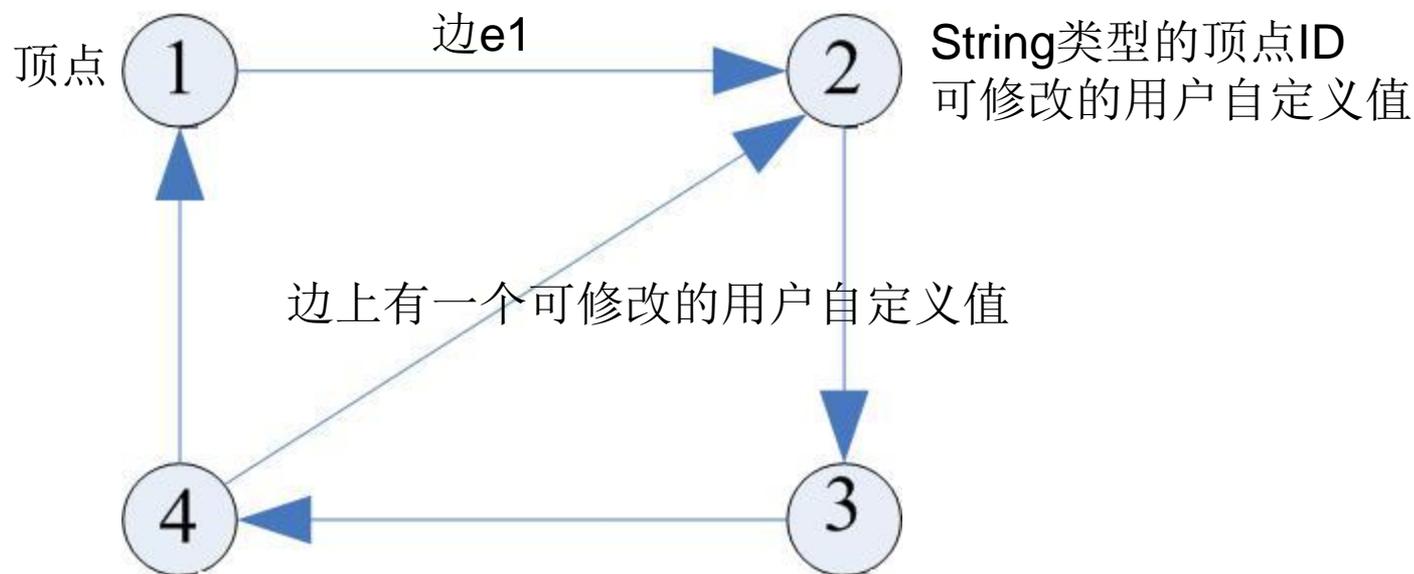
# 11.3 Pregel图计算模型

- 11.3.1 有向图和顶点
- 11.3.2 顶点之间的消息传递
- 11.3.3 Pregel的计算过程
- 11.3.4 实例



## 11.3.1 有向图和顶点

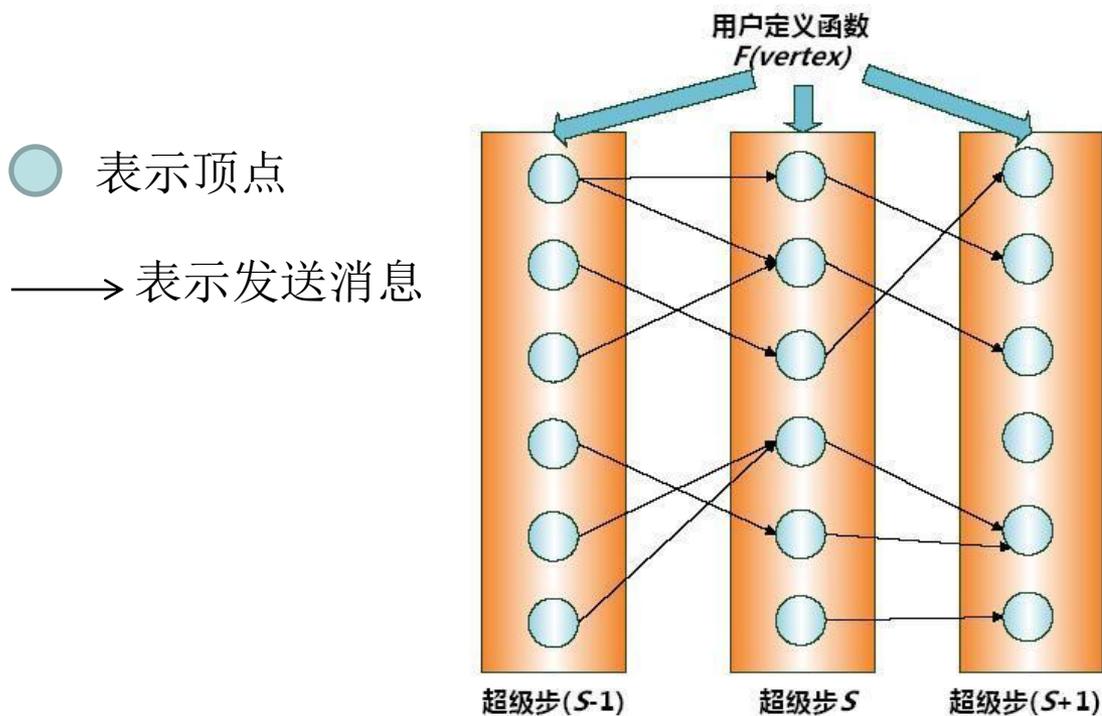
- Pregel计算模型以有向图作为输入
- 有向图的每个顶点都有一个**String**类型的顶点ID
- 每个顶点都有一个可修改的用户自定义值与之关联
- 每条有向边都和其源顶点关联，并记录了其目标顶点ID
- 边上有一个可修改的用户自定义值





# 11.3.1 有向图和顶点

- 在每个超步  $S$  中，图中的所有顶点都会并行执行相同的用户自定义函数
- 每个顶点可以接收前一个超步  $(S-1)$  中发送给它的消息，修改其自身及其出射边的状态，并发送消息给其他顶点，甚至是修改整个图的拓扑结构
- 在这种计算模式中，“边”并不是核心对象，在边上面不会运行相应的计算，只有顶点才会执行用户自定义函数进行相应计算





## 11.3.2 顶点之间的消息传递

采用消息传递模型主要基于以下两个原因：

- (1) 消息传递具有足够的表达能力，没有必要使用远程读取或共享内存的方式
- (2) 有助于提升系统整体性能。大型图计算通常是由一个集群完成的，集群环境中执行远程数据读取会有较高的延迟；Pregel的消息模式采用异步和批量的方式传递消息，因此可以缓解远程读取的延迟

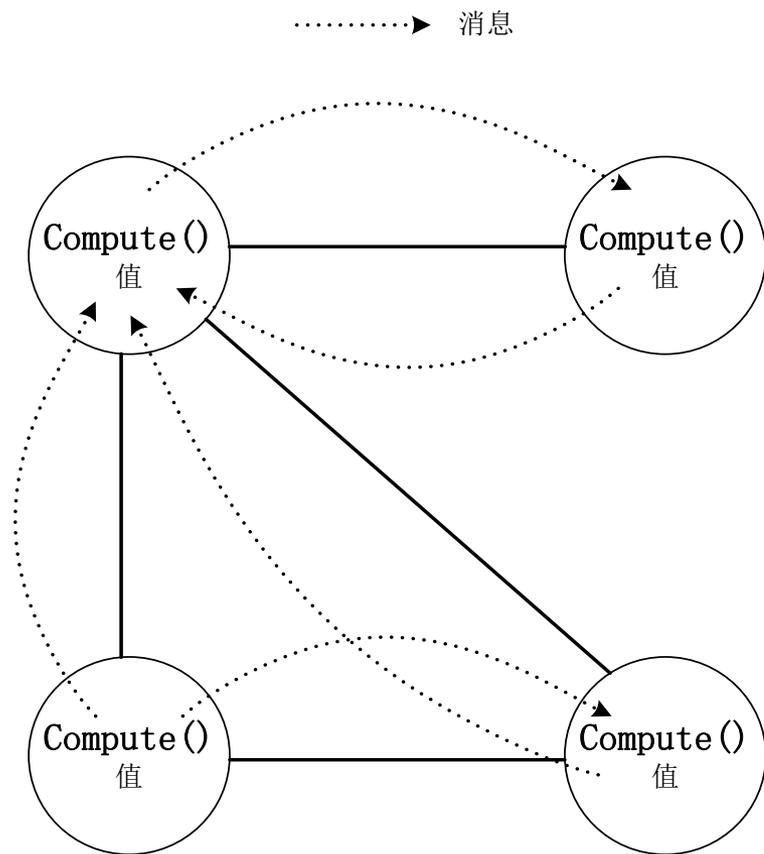
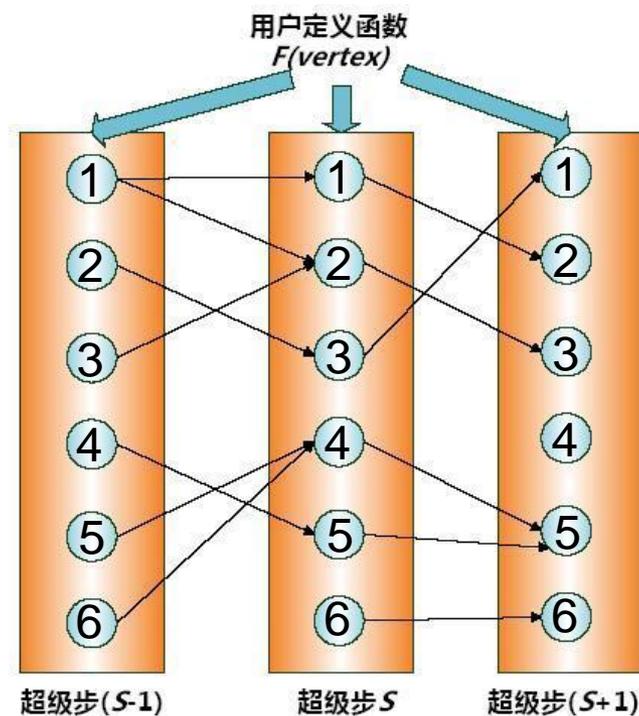


图9-2 纯消息传递模型图



# 11.3.3 Pregel的计算过程

- Pregel的计算过程是由一系列被称为“超步”的迭代组成的
- 在每个超步中，每个顶点上面都会并行执行用户自定义的函数，该函数描述了一个顶点 $V$ 在一个超步 $S$ 中需要执行的操作
- 该函数可以读取前一个超步 $(S-1)$ 中其他顶点发送给顶点 $V$ 的消息，执行相应计算后，修改顶点 $V$ 及其出射边的状态，然后沿着顶点 $V$ 的出射边发送消息给其他顶点，而且，一个消息可能经过多条边的传递后被发送到任意已知ID的目标顶点上去
- 这些消息将会在下一个超步 $(S+1)$ 中被目标顶点接收，然后象上述过程一样开始下一个超步 $(S+1)$ 的迭代过程



● 表示顶点

→ 表示发送消息



## 11.3.3 Pregel的计算过程

- 在Pregel计算过程中，一个算法什么时候可以结束，是由所有顶点的状态决定的
- 在第0个超步，所有顶点处于活跃状态，都会参与该超步的计算过程
- 当一个顶点不需要继续执行进一步的计算时，就会把自己的状态设置为“停机”，进入非活跃状态
- 一旦一个顶点进入非活跃状态，后续超步中就不会再在该顶点上执行计算，除非其他顶点给该顶点发送消息把它再次激活
- 当一个处于非活跃状态的顶点收到来自其他顶点的消息时，Pregel计算框架必须根据条件判断来决定是否将其显式唤醒进入活跃状态
- 当图中所有的顶点都已经标识其自身达到“非活跃（inactive）”状态，并且没有消息在传送的时候，算法就可以停止运行

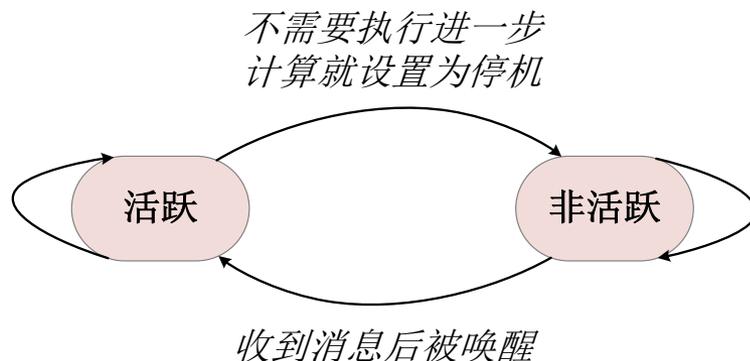


图9-3 一个简单的状态机图



# 11.3.4 实例

○ 活跃  
● 非活跃

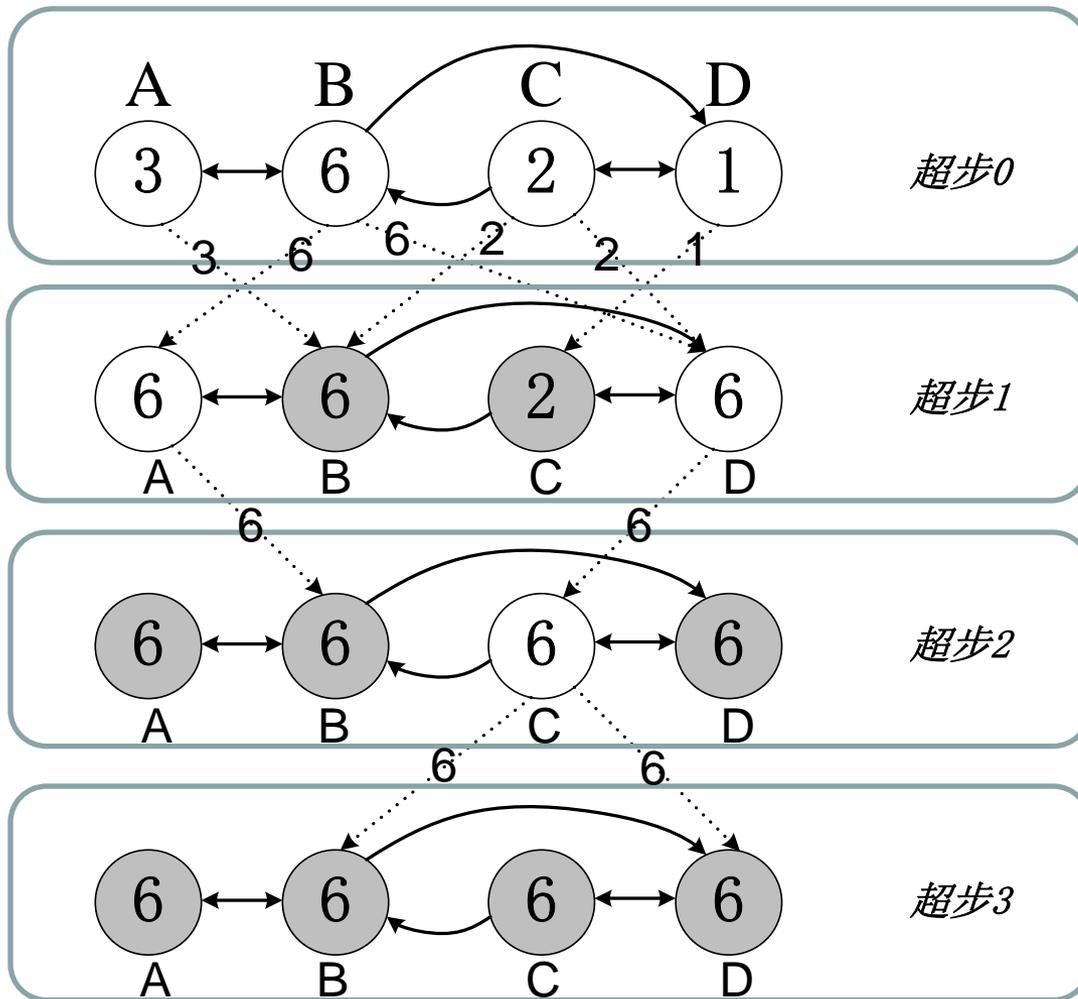


图9-4 一个求最大值的Pregel计算过程图



## 11.4 Pregel的C++ API

Pregel已经预先定义好一个基类——Vertex类:

```
template <typename VertexValue, typename EdgeValue, typename MessageValue>
class Vertex {
public:
    virtual void Compute(MessageIterator* msgs) = 0;
    const string& vertex_id() const;
    int64 superstep() const;
    const VertexValue& GetValue();
    VertexValue* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();
    void SendMessageTo(const string& dest_vertex,          const MessageValue& message);
    void VoteToHalt();
};
```

- 在Vertex类中，定义了三个值类型参数，分别表示顶点、边和消息。每一个顶点都有一个给定类型的值与之对应
- 编写Pregel程序时，需要继承Vertex类，并且覆写Vertex类的虚函数Compute()



## 11.4 Pregel的C++ API

- 在Pregel执行计算过程时，在每个超步中都会并行调用每个顶点上定义的Compute()函数
- 允许Compute()方法查询当前顶点及其边的信息，以及发送消息到其他的顶点
  - Compute()方法可以调用GetValue()方法来获取当前顶点的值
  - 调用MutableValue()方法来修改当前顶点的值
  - 通过由出射边的迭代器提供的方法来查看、修改出射边对应的值
- 对状态的修改，对于被修改的顶点而言是可以立即被看见的，但是，对于其他顶点而言是不可见的，因此，不同顶点并发进行的数据访问是不存在竞争关系的

整个过程中，唯一需要在超步之间持久化的顶点级状态，是顶点和其对应的边所关联的值，因而，Pregel计算框架所需要管理的图状态就只包括顶点和边所关联的值，这种做法大大简化了计算流程，同时，也有利于图的分布和故障恢复



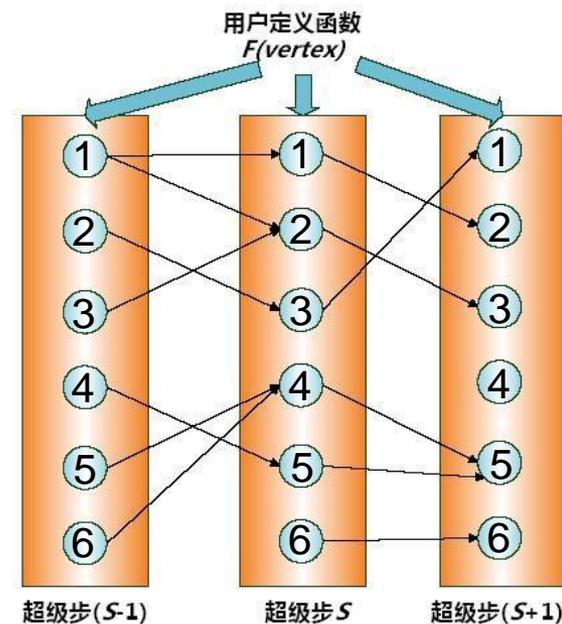
# 11.4 Pregel的C++ API

- 11.4.1 消息传递机制
- 11.4.2 Combiner
- 11.4.3 Aggregator
- 11.4.4 拓扑改变
- 11.4.5 输入和输出



# 11.4.1 消息传递机制

- 顶点之间的通讯是借助于消息传递机制来实现的，每条消息都包含了消息值和需要到达的目标顶点ID。用户可以通过Vertex类的模板参数来设定消息值的数据类型
- 在一个超步S中，一个顶点可以发送任意数量的消息，这些消息将在下一个超步（S+1）中被其他顶点接收
- 也就是说，在超步（S+1）中，当Pregel计算框架在顶点V上执行用户自定义的Compute()方法时，所有在前一个超步S中发送给顶点V的消息，都可以通过一个迭代器来访问到。迭代器不能保证消息的顺序，不过可以保证消息一定会被传送并且不会被重复传送
- 一个顶点V通过与之关联的出射边向外发送消息，并且，消息要到达的目标顶点并不一定是与顶点V相邻的顶点，一个消息可以连续经过多条连通的边到达某个与顶点V不相邻的顶点U，U可以从接收的消息中获取到与其不相邻的顶点V的ID





## 11.4.2 Combiner

- **Pregel**计算框架在消息发出去之前，**Combiner**可以将发往同一个顶点的多个整型值进行求和得到一个值，只需向外发送这个“求和结果”，从而实现了由多个消息合并成一个消息，大大减少了传输和缓存的开销
- 在默认情况下，**Pregel**计算框架并不会开启**Combiner**功能，因为，通常很难找到一种对所有顶点的**Compute()**函数都合适的**Combiner**
- 当用户打算开启**Combiner**功能时，可以继承**Combiner**类并覆写虚函数**Combine()**
- 此外，通常只对那些满足交换律和结合律的操作才可以去开启**Combiner**功能，因为，**Pregel**计算框架无法保证哪些消息会被合并，也无法保证消息传递给 **Combine()**的顺序和合并操作执行的顺序

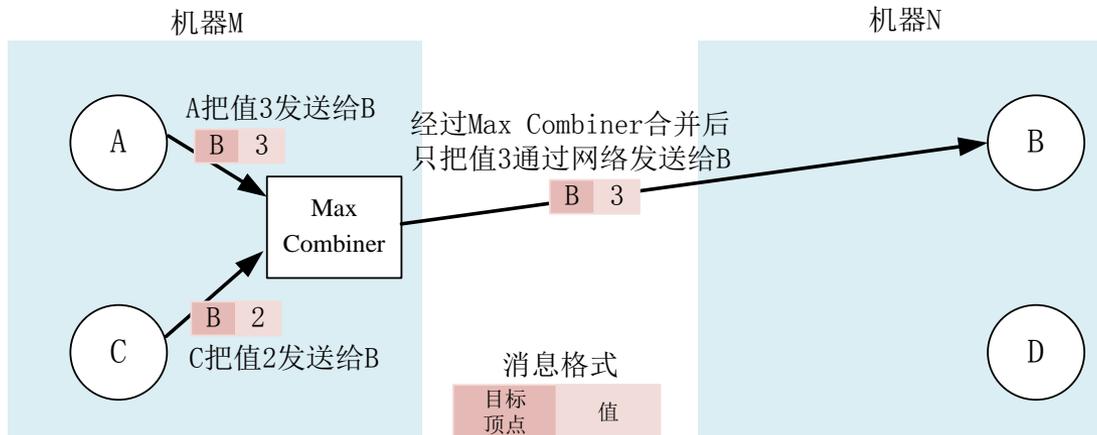


图9-5 Combiner应用的例子



## 11.4.3 Aggregator

- Aggregator提供了一种全局通信、监控和数据查看的机制
- 在一个超步 $S$ 中，每一个顶点都可以向一个Aggregator提供一个数据，Pregel计算框架会对这些值进行聚合操作产生一个值，在下一个超步( $S+1$ )中，图中的所有顶点都可以看见这个值
- Aggregator的聚合功能，允许在整型和字符串类型上执行最大值、最小值、求和操作，比如，可以定义一个“Sum”Aggregator来统计每个顶点的出射边数量，最后相加可以得到整个图的边的数量
- Aggregator还可以实现全局协同的功能，比如，可以设计“and”Aggregator来决定在某个超步中Compute()函数是否执行某些逻辑分支，只有当“and”Aggregator显示所有顶点都满足了某条件时，才去执行这些逻辑分支



## 11.4.4 拓扑改变

- **Pregel**计算框架允许用户在自定义函数**Compute()**中定义操作，修改图的拓扑结构，比如在图中增加（或删除）边或顶点
- 对于全局拓扑改变，**Pregel**采用了惰性协调机制，在改变请求发出时，**Pregel**不会对这些操作进行协调，只有当这些改变请求的消息到达目标顶点并被执行时，**Pregel**才会对这些操作进行协调，这样，所有针对某个顶点**V**的拓扑修改操作所引发的冲突，都会由**V**自己来处理
- 对于本地的局部拓扑改变，是不会引发冲突的，顶点或边的本地增减能够立即生效，很大程度上简化了分布式编程



## 11.4.5 输入和输出

- 在Pregel计算框架中，图的保存格式多种多样，包括文本文件、关系数据库或键值数据库等
- 在Pregel中，“从输入文件生成得到图结构”和“执行图计算”这两个过程是分离的，从而不会限制输入文件的格式
- 对于输出，Pregel也采用了灵活的方式，可以以多种方式进行输出



# 11.5 Pregel的体系结构

- 11.5.1 Pregel的执行过程
- 11.5.2 容错性
- 11.5.3 Worker
- 11.5.4 Master
- 11.5.5 Aggregator



# 11.5.1 Pregel的执行过程

- 在Pregel计算框架中，一个大型图会被划分成许多个分区，每个分区都包含了一部分顶点以及以其为起点的边
- 一个顶点应该被分配到哪个分区上，是由一个函数决定的，系统默认函数为 $\text{hash}(\text{ID}) \bmod N$ ，其中， $N$ 为所有分区总数， $\text{ID}$ 是这个顶点的标识符；当然，用户也可以自己定义这个函数
- 这样，无论在哪台机器上，都可以简单根据顶点 $\text{ID}$ 判断出该顶点属于哪个分区，即使该顶点可能已经不存在了

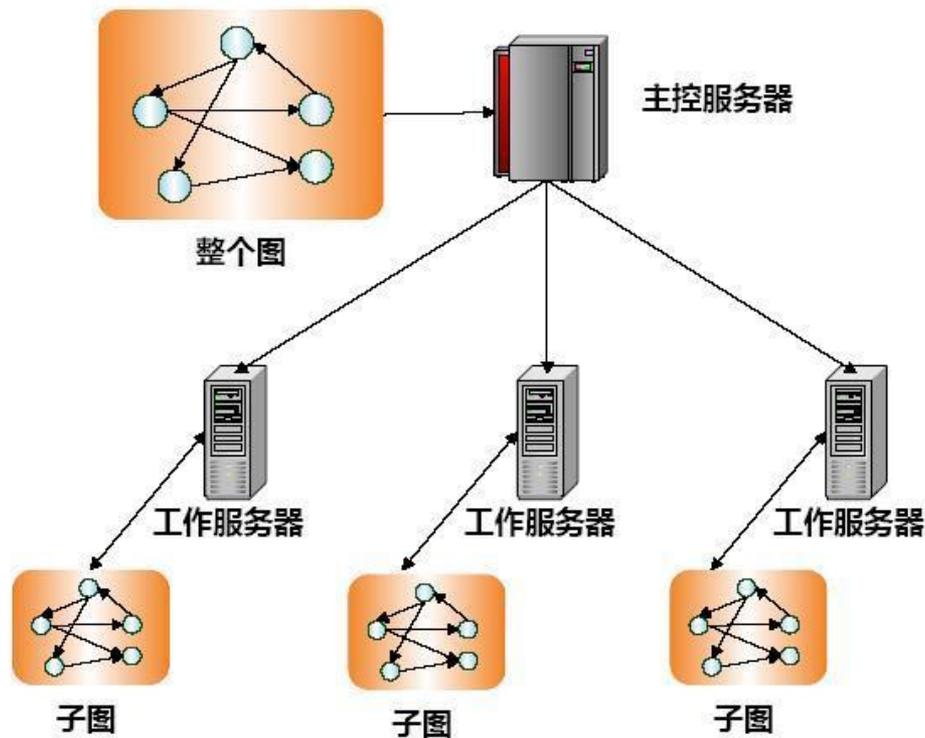


图9-6图的划分图



# 11.5.1 Pregel的执行过程

在理想的情况下（不发生任何错误），一个Pregel用户程序的执行过程如下：

(1) 选择集群中的多台机器执行图计算任务，每台机器上运行用户程序的一个副本，其中，有一台机器会被选为Master，其他机器作为Worker。Master只负责协调多个Worker执行任务，系统不会把图的任何分区分配给它。Worker借助于名称服务系统可以定位到Master的位置，并向Master发送自己的注册信息。

(2) Master把一个图分成多个分区，并把分区分配到多个Worker。一个Worker会领到一个或多个分区，每个Worker知道所有其他Worker所分配到的分区情况。每个Worker负责维护分配给自己的那些分区的状态(顶点及边的增删)，对分配给自己的分区中的顶点执行Compute()函数，向外发送消息，并管理接收到的消息。

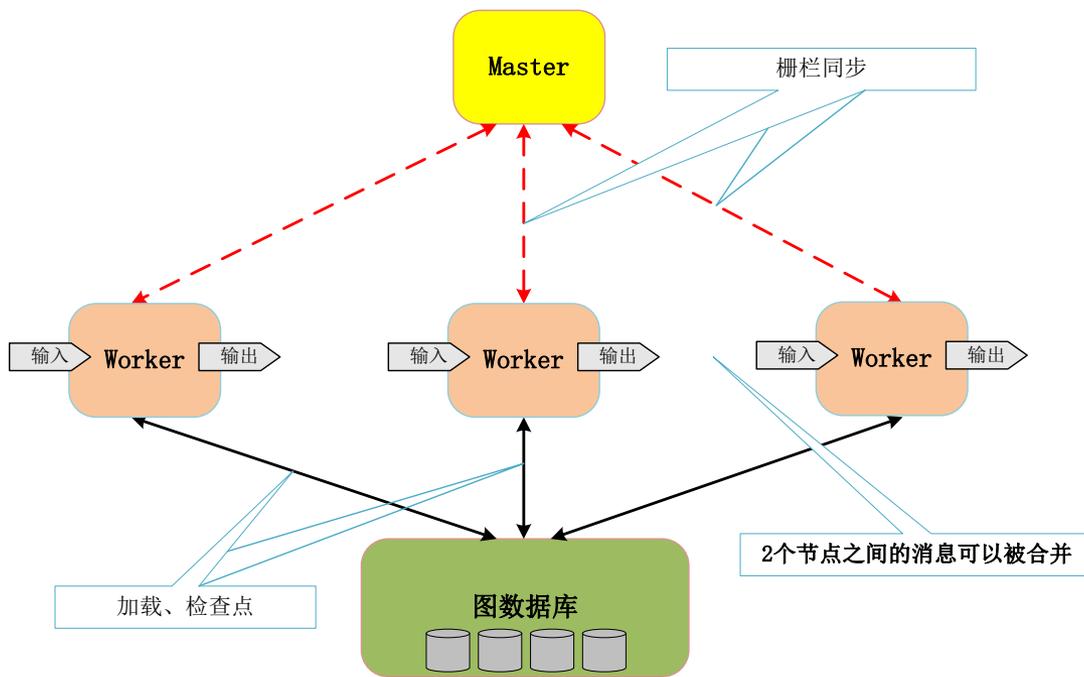


图9-7 Pregel的执行过程图



# 11.5.1 Pregel的执行过程

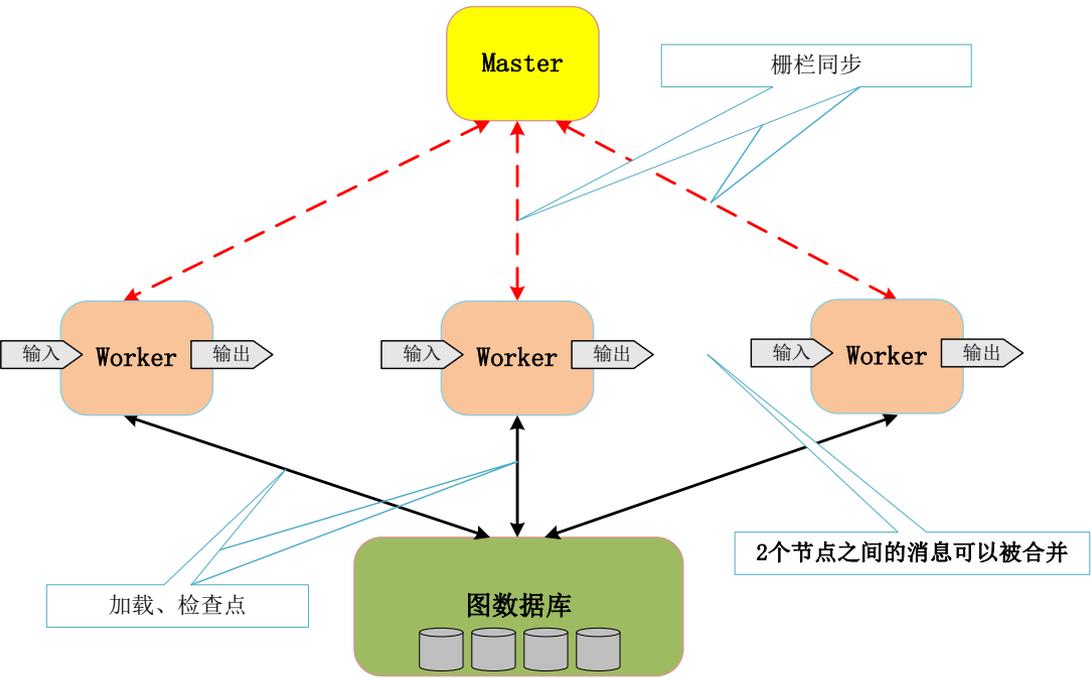


图9-7 Pregel的执行过程图

(3) Master会把用户输入划分成多个部分，通常是基于文件边界进行划分。划分后，每个部分都是一系列记录的集合，每条记录都包含一定数量的顶点和边。然后，Master会为每个Worker分配用户输入的一部分。如果一个Worker从输入内容中加载到的顶点，刚好是自己所分配到的分区中的顶点，就会立即更新相应的数据结构。否则，该Worker会根据加载到的顶点的ID，把它发送到其所属的分区所在的Worker上。当所有的输入都被加载后，图中的所有顶点都会被标记为“活跃”状态。



# 11.5.1 Pregel的执行过程

(4) Master向每个Worker发送指令，Worker收到指令后，开始运行一个超步。Worker会为自己管辖的每个分区分配一个线程，对于分区中的每个顶点，Worker会把来自上一个超步的、发给该顶点的消息传递给它，并调用处于“活跃”状态的顶点上的Compute()函数，在执行计算过程中，顶点可以对外发送消息，但是，所有消息的发送工作必须在本超步结束之前完成。当所有这些工作都完成以后，Worker会通知Master，并把自己在下一个超步还处于“活跃”状态的顶点的数量报告给Master。上述步骤会被不断重复，直到所有顶点都不再活跃并且系统中不会有任何消息在传输，这时，执行过程才会结束。

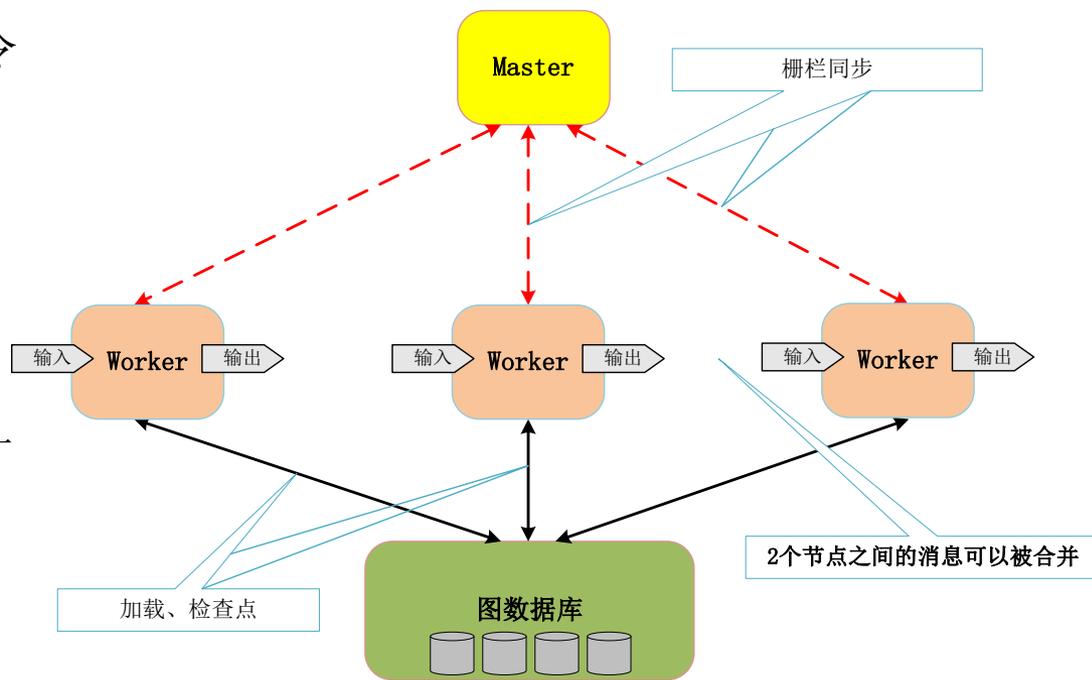


图9-7 Pregel的执行过程图

(5) 计算过程结束后，Master会给所有的Worker发送指令，通知每个Worker对自己的计算结果进行持久化存储。



## 11.5.2 容错性

- **Pregel**采用检查点机制来实现容错。在每个超步的开始，**Master**会通知所有的**Worker**把自己管辖的分区的状态（包括顶点值、边值以及接收到的消息），写入到持久化存储设备
- **Master**会周期性地向每个**Worker**发送ping消息，**Worker**收到ping消息后会给**Master**发送反馈消息。如果**Master**在指定时间间隔内没有收到某个**Worker**的反馈消息，就会把该**Worker**标记为“失效”。同样地，如果一个**Worker**在指定的时间间隔内没有收到来自**Master**的ping消息，该**Worker**也会停止工作
- 每个**Worker**上都保存了一个或多个分区的状态信息，当一个**Worker**发生故障时，它所负责维护的分区的当前状态信息就会丢失。**Master**监测到一个**Worker**发生故障“失效”后，会把失效**Worker**所分配到的分区，重新分配到其他处于正常工作状态的**Worker**集合上，然后，所有这些分区会从最近的某超步**S**开始时写出的检查点中，重新加载状态信息



## 11.5.3 Worker

在一个**Worker**中，它所管辖的分区的状态信息是保存在内存中的。

分区中的顶点的状态信息包括：

- 顶点的当前值
- 以该顶点为起点的出射边列表，每条出射边包含了目标顶点ID和边的值
- 消息队列，包含了所有接收到的、发送给该顶点的消息
- 标志位，用来标记顶点是否处于活跃状态

在每个超步中，**Worker**会对自己所管辖的分区中的每个顶点进行遍历，并调用顶点上的**Compute()**函数，在调用时，会把以下三个参数传递进去：

- 该顶点的当前值
- 一个接收到的消息的迭代器
- 一个出射边的迭代器



## 11.5.3 Worker

- 在Pregel中，为了获得更好的性能，“标志位”和输入消息队列是分开保存的
- 对于每个顶点而言，Pregel只保存一份顶点值和边值，但是，会保存两份“标志位”和输入消息队列，分别用于当前超步和下一个超步
- 在超步 $S$ 中，当一个Worker在进行顶点处理时，用于当前超步的消息会被处理，同时，它在处理过程中还会接收到来自其他Worker的消息，这些消息会在下一个超步 $S+1$ 中被处理，因此，需要两个消息队列用于存放作用于当前超步 $S$ 的消息和作用于下一个超步 $S+1$ 的消息
- 如果一个顶点 $V$ 在超步 $S$ 接收到消息，那么，它表示 $V$ 将会在下一个超步 $S+1$ 中（而不是当前超步 $S$ 中）处于“活跃”状态



## 11.5.3 Worker

- 当一个Worker上的一个顶点V需要发送消息到其他顶点U时，该Worker会首先判断目标顶点U是否位于自己机器上
- 如果目标顶点U在自己的机器上，就直接把消息放入到与目标顶点U对应的输入消息队列中
- 如果发现目标顶点U在远程机器上，这个消息就会被暂时缓存到本地，当缓存中的消息数目达到一个事先设定的阈值时，这些缓存消息会被批量异步发送出去，传输到目标顶点所在的Worker上
- 如果存在用户自定义的Combiner操作，那么，当消息被加入到输出队列或者到达输入队列时，就可以对消息执行合并操作，这样可以节省存储空间和网络传输开销



## 11.5.4 Master

- **Master**主要负责协调各个**Worker**执行任务，每个**Worker**会借助于名称服务系统定位到**Master**的位置，并向**Master**发送自己的注册信息，**Master**会为每个**Worker**分配一个唯一的ID
- **Master**维护着关于当前处于“有效”状态的所有**Worker**的各种信息，包括每个**Worker**的ID和地址信息，以及每个**Worker**被分配到的分区信息
- 虽然在集群中只有一个**Master**，但是，它仍然能够承担起一个大规模图计算的协调任务，这是因为**Master**中保存这些信息的数据结构的大小，只与分区的数量有关，而与顶点和边的数量无关



## 11.5.4 Master

- 一个大规模图计算任务会被**Master**分解到多个**Worker**去执行，在每个超步开始时，**Master**都会向所有处于“有效”状态的**Worker**发送相同的指令，然后等待这些**Worker**的回应
- 如果在指定时间内收不到某个**Worker**的反馈，**Master**就认为这个**Worker**失效
- 如果参与任务执行的多个**Worker**中的任意一个发生了故障失效，**Master**就会进入恢复模式
- 在每个超步中，图计算的各种工作，比如输入、输出、计算、保存和从检查点中恢复，都会在“路障（barrier）”之前结束
- 如果路障同步成功，说明一个超步顺利结束，**Master**就会进入下一个处理阶段，图计算进入下一个超步的执行



## 11.5.4 Master

- **Master**在内部运行了一个HTTP服务器来显示图计算过程的各种信息
- 用户可以通过网页随时监控图计算执行过程各个细节
  - 图的大小
  - 关于出度分布的柱状图
  - 处于活跃状态的顶点数量
  - 在当前超步的时间信息和消息流量
  - 所有用户自定义Aggregator的值



## 11.5.5 Aggregator

- 每个用户自定义的**Aggregator**都会采用聚合函数对一个值集合进行聚合计算得到一个全局值
- 每个**Worker**都保存了一个**Aggregator**的实例集，其中的每个实例都是由类型名称和实例名称来标识的
- 在执行图计算过程的某个超步**S**中，每个**Worker**会利用一个**Aggregator**对当前本地分区中包含的所有顶点的值进行归约，得到一个本地的局部归约值
- 在超步**S**结束时，所有**Worker**会将所有包含局部归约值的**Aggregator**的值进行最后的汇总，得到全局值，然后提交给**Master**
- 在下一个超步**S+1**开始时，**Master**就会将**Aggregator**的全局值发送给每个**Worker**

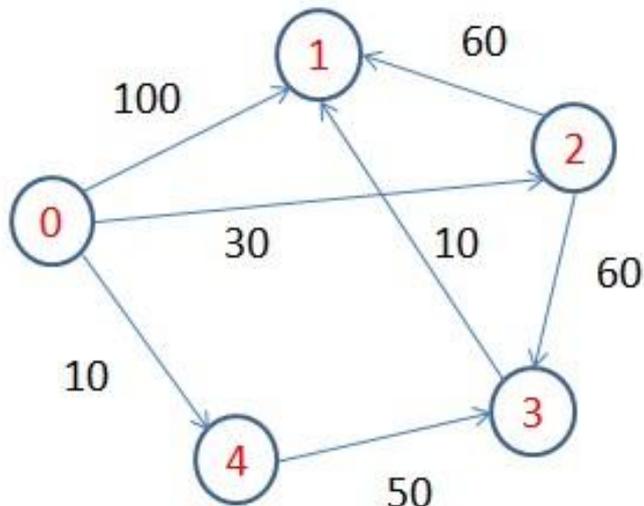


# 11.6 Pregel的应用实例

- 11.6.1 单源最短路径
- 11.6.2 二分匹配



# 11.6.1 单源最短路径



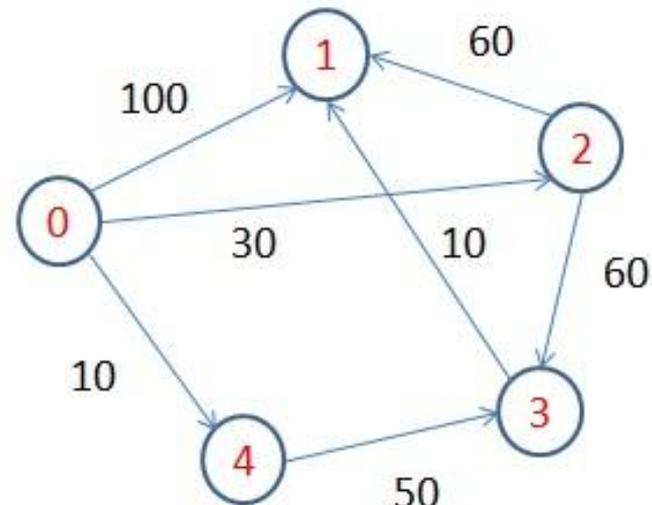
Dijkstra算法是解决单源最短路径问题的贪婪算法



# 11.6.1 单源最短路径

Pregel非常适合用来解决单源最短路径问题，实现代码如下：

```
1 class ShortestPathVertex
2   : public Vertex<int, int, int> {
3   void Compute(MessageIterator* msgs) {
4     int mindist = IsSource(vertex_id()) ? 0 : INF;
5     for (; !msgs->Done(); msgs->Next())
6       mindist = min(mindist, msgs->Value());
7     if (mindist < GetValue()) {
8       *MutableValue() = mindist;
9     OutEdgeIterator iter = GetOutEdgeIterator();
10    for (; !iter.Done(); iter.Next())
11      SendMessageTo(iter.Target(),
12                    mindist + iter.GetValue());
13  }
14  VoteToHalt();
15  }
16  };
```



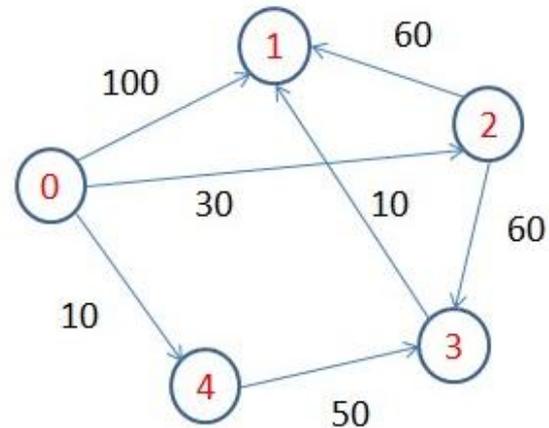


# 11.6.1 单源最短路径

```

1 class ShortestPathVertex
2   : public Vertex<int, int, int> {
3   void Compute(MessageIterator* msgs) {
4     int mindist = IsSource(vertex_id()) ? 0 : INF;
5     for (; !msgs->Done(); msgs->Next())
6       mindist = min(mindist, msgs->Value());
7     if (mindist < GetValue()) {
8       *MutableValue() = mindist;
9       OutEdgeIterator iter = GetOutEdgeIterator();
10      for (; !iter.Done(); iter.Next())
11        SendMessageTo(iter.Target(),
12                      mindist + iter.GetValue());
13    }
14    VoteToHalt();
15  }
16 };

```



每个顶点并行执行Compute()函数

表1 超步0开始时的顶点值

	0	1	2	3	4
0	0	INF	INF	INF	INF

表2 超步0结束时的顶点值

	0	1	2	3	4
0	0	INF	INF	INF	INF

超步0结束时，所有顶点非活跃

表3 顶点0向其他顶点发送消息

	1	2	3	4
0	100	30	无	10



# 11.6.1 单源最短路径

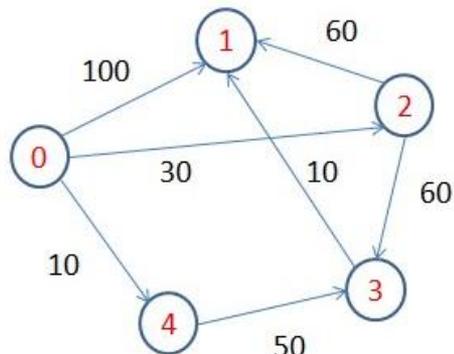


表4 上一步（超步0）中发出的消息

	1	2	3	4
0	100	30	无	10

表5 超步1开始时的顶点值

	0	1	2	3	4
0	0	INF	INF	INF	INF

表6 超步1结束时的顶点值

	0	1	2	3	4
0	0	100	30	INF	10

超步1:

- 顶点0: 没有收到消息, 依然非活跃
- 顶点1: 收到消息100 (唯一消息), 被显式唤醒, 执行计算, mindist变为100, 小于顶点值INF, 顶点值修改为100, 没有出射边, 不需要发送消息, 最后变为非活跃
- 顶点2: 收到消息30, 被显式唤醒, 执行计算, mindist变为30, 小于顶点值INF, 顶点值修改为30, 有两条出射边, 向顶点3发送消息90 (即: 30+60), 向顶点1发送消息90 (即: 30+60), 最后变为非活跃
- 顶点3: 没有收到消息, 依然非活跃
- 顶点4: 收到消息10, 被显式唤醒, 执行计算, mindist变为10, 小于顶点值INF, 顶点值修改为10, 向顶点3发送消息60 (即: 10+50), 最后变为非活跃

剩余超步省略.....

当所有顶点非活跃, 并且没有消息传递, 就结束



## 11.6.2 二分匹配

程序的执行过程是由四个阶段组成的多个循环组成的，当程序执行到超步 $S$ 时， $S \bmod 4$ 就可以得到当前超步处于循环的哪个阶段。每个循环的四个阶段如下：

**(1) 阶段0:** 对于左集合中的任意顶点 $V$ ，如果 $V$ 还没有被匹配，就发送消息给它的每个邻居顶点请求匹配，然后，顶点 $V$ 会调用`VoteToHalt()`进入“非活跃”状态。如果顶点 $V$ 已经找到了匹配，或者 $V$ 没有找到匹配但是没有出射边，那么，顶点 $V$ 就不会发送消息。当顶点 $V$ 没有发送消息，或者顶点 $V$ 发送了消息但是所有的消息接收者都已经被匹配，那么，该顶点就不会再变为“活跃 (**active**)”状态

**(2) 阶段1:** 对于右集合中的任意顶点 $U$ ，如果它还没有被匹配，则会随机选择它接收到的消息中的其中一个，并向左集合中的消息发送者发送消息表示接受该匹配请求，然后给左集合中的其他请求者发送拒绝消息；然后，顶点 $U$ 会调用`VoteToHalt()`进入“非活跃”状态

**(3) 阶段2:** 左集合中那些还未被匹配的顶点，会从它所收到的、右集合发送过来的接受请求中，选择其中一个给予确认，并发送一个确认消息。对于左集合中已经匹配的顶点而言，因为它们在阶段0不会向右集合发送任何匹配请求消息，因而也不会接收到任何来自右集合的匹配接受消息，因此，是不会执行阶段2的

**(4) 阶段3:** 右集合中还未被匹配的任意顶点 $U$ ，会收到来自左集合的匹配确认消息，但是，每个未匹配的顶点 $U$ ，最多会收到一个确认消息。然后，顶点 $U$ 会调用`VoteToHalt()`进入“非活跃”状态，完成它自身的匹配工作



## 11.7 Pregel和MapReduce实现PageRank算法的对比

- **11.7.1 PageRank算法**
- **11.7.2 PageRank算法在Pregel中的实现**
- **11.7.3 PageRank算法在MapReduce中的实现**
- **11.7.4 PageRank算法在Pregel和MapReduce中实现的比较**



# 11.7.1 PageRank算法

- PageRank是一个函数，它为网络中每个网页赋一个权值。通过该权值来判断该网页的重要性
- 该权值分配的方法并不是固定的，对PageRank算法的一些简单变形都会改变网页的相对PageRank值（PR值）
- PageRank作为谷歌的网页链接排名算法，基本公式如下：

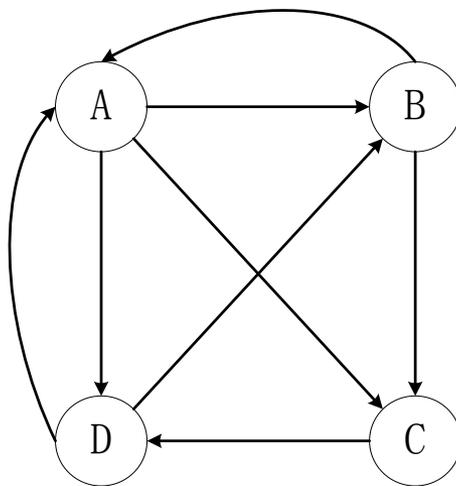
$$PR = \beta \sum_{i=1}^n \frac{PR_i}{N_i} + (1 - \beta) \frac{1}{N}$$

- 对于任意一个网页链接，其PR值为链入到该链接的源链接的PR值对该链接的贡献和，其中， $N$ 表示该网络中所有网页的数量， $N_i$ 为第 $i$ 个源链接的链出度， $PR_i$ 表示第 $i$ 个源链接的PR值



# 11.7.1 PageRank算法

- 网络链接之间的关系可以用一个连通图来表示，下图就是四个网页（ $A, B, C, D$ ）互相链入链出组成的连通图，从中可以看出，网页 $A$ 中包含指向网页 $B$ 、 $C$ 和 $D$ 的外链，网页 $B$ 和 $D$ 是网页 $A$ 的源链接





## 11.7.2 PageRank算法在Pregel中的实现

- 在Pregel计算模型中，图中的每个顶点会对应一个计算单元，每个计算单元包含三个成员变量：
  - 顶点值（Vertex value）：顶点对应的PR值
  - 出射边（Out edge）：只需要表示一条边，可以不取值
  - 消息（Message）：传递的消息，因为需要将本顶点对其它顶点的PR贡献值，传递给目标顶点
- 每个计算单元包含一个成员函数Compute()，该函数定义了顶点上的运算，包括该顶点的PR值计算，以及从该顶点发送消息到其链出顶点



## 11.7.2 PageRank算法在Pregel中的实现

```
class PageRankVertex: public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (;!msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *MutableValue() =
                0.15 / NumVertices() + 0.85 * sum;
        }
        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue()/ n);
        } else {
            VoteToHalt();
        }
    }
};
```



## 11.7.2 PageRank算法在Pregel中的实现

- PageRankVertex继承自Vertex类，顶点值类型是double，用来保存PageRank中间值，消息类型也是double，用来传输PageRank值，边的value类型是void，因为不需要存储任何信息
- 这里假设在第0个超步时，图中各顶点值被初始化为 $1/\text{NumVertices}()$ ，其中， $\text{NumVertices}()$ 表示顶点数目
- 在前30个超步中，每个顶点都会沿着它的出射边，发送它的PageRank值除以出射边数目以后的结果值。从第1个超步开始，每个顶点会将到达的消息中的值加到sum值中，同时将它的PageRank值设为 $0.15/\text{NumVertices}()+0.85*\text{sum}$
- 到了第30个超步后，就没有需要发送的消息了，同时所有的顶点停止计算，得到最终结果



## 11.7.3 PageRank算法在MapReduce中的实现

- MapReduce也是谷歌公司提出的一种计算模型，它是为全量计算而设计
- 采用MapReduce实现PageRank的计算过程包括三个阶段：
  - 第一阶段：解析网页
  - 第二阶段：**PageRank分配**
  - 第三阶段：收敛阶段



# 11.7.3 PageRank算法在MapReduce中的实现

## 1. 阶段1: 解析网页

- 该阶段的任务就是分析一个页面的链接数并赋初值。
- 一个网页可以表示为由网址和内容构成的键值对 $\langle \text{URL}, \text{page content} \rangle$ ，作为Map任务的输入。阶段1的Map任务把 $\langle \text{URL}, \text{page content} \rangle$ 映射为 $\langle \text{URL}, \langle \text{PR}_{\text{init}}, \text{url\_list} \rangle \rangle$ 后进行输出，其中， $\text{PR}_{\text{init}}$ 是该URL页面对应的PageRank初始值， $\text{url\_list}$ 包含了该URL页面中的外链所指向的所有URL。Reduce任务只是恒等函数，输入和输出相同。
- 对右图，每个网页的初始PageRank值为 $1/4$ 。它在该阶段中：

Map任务的输入为：

$\langle A_{\text{URL}}, A_{\text{content}} \rangle$

$\langle B_{\text{URL}}, B_{\text{content}} \rangle$

$\langle C_{\text{URL}}, C_{\text{content}} \rangle$

$\langle D_{\text{URL}}, D_{\text{content}} \rangle$

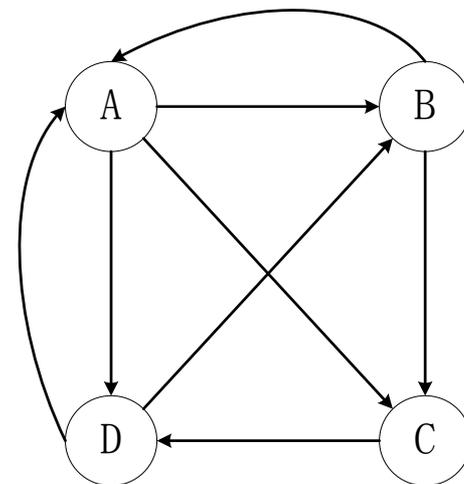
Map任务的输出为：

$\langle A_{\text{URL}}, \langle 1/4, \langle B_{\text{URL}}, C_{\text{URL}}, D_{\text{URL}} \rangle \rangle \rangle$

$\langle B_{\text{URL}}, \langle 1/4, \langle A_{\text{URL}}, C_{\text{URL}} \rangle \rangle \rangle$

$\langle C_{\text{URL}}, \langle 1/4, D_{\text{URL}} \rangle \rangle$

$\langle D_{\text{URL}}, \langle 1/4, \langle A_{\text{URL}}, B_{\text{URL}} \rangle \rangle \rangle$





# 11.7.3 PageRank算法在MapReduce中的实现

## 2. 阶段2: PageRank分配

- 该阶段的任务就是多次迭代计算页面的PageRank值。
- 在该阶段中，Map任务的输入是 $\langle \text{URL}, \langle \text{cur\_rank}, \text{url\_list} \rangle \rangle$ ，其中， $\text{cur\_rank}$ 是该URL页面对应的PageRank当前值， $\text{url\_list}$ 包含了该URL页面中的外链所指向的所有URL。
- 对于 $\text{url\_list}$ 中的每个元素 $u$ ，Map任务输出 $\langle u, \langle \text{URL}, \text{cur\_rank}/|\text{url\_list}| \rangle \rangle$ （其中， $|\text{url\_list}|$ 表示外链的个数），并输出链接关系 $\langle \text{URL}, \text{url\_list} \rangle$ 。
- 每个页面的PageRank当前值被平均分配给了它们的每个外链。Map任务的输出会作为下面Reduce任务的输入。对下图第一次迭代Map任务的输入输出如下：

输入为:

$\langle A_{\text{URL}}, A_{\text{content}} \rangle$

$\langle B_{\text{URL}}, B_{\text{content}} \rangle$

$\langle C_{\text{URL}}, C_{\text{content}} \rangle$

$\langle D_{\text{URL}}, D_{\text{content}} \rangle$

输出为:

$\langle B_{\text{URL}}, \langle A_{\text{URL}}, 1/12 \rangle \rangle$

$\langle C_{\text{URL}}, \langle A_{\text{URL}}, 1/12 \rangle \rangle$

$\langle D_{\text{URL}}, \langle A_{\text{URL}}, 1/12 \rangle \rangle$

$\langle A_{\text{URL}}, \langle B_{\text{URL}}, C_{\text{URL}}, D_{\text{URL}} \rangle \rangle$

$\langle A_{\text{URL}}, \langle B_{\text{URL}}, 1/8 \rangle \rangle$

$\langle C_{\text{URL}}, \langle B_{\text{URL}}, 1/8 \rangle \rangle$

$\langle B_{\text{URL}}, \langle A_{\text{URL}}, C_{\text{URL}} \rangle \rangle$

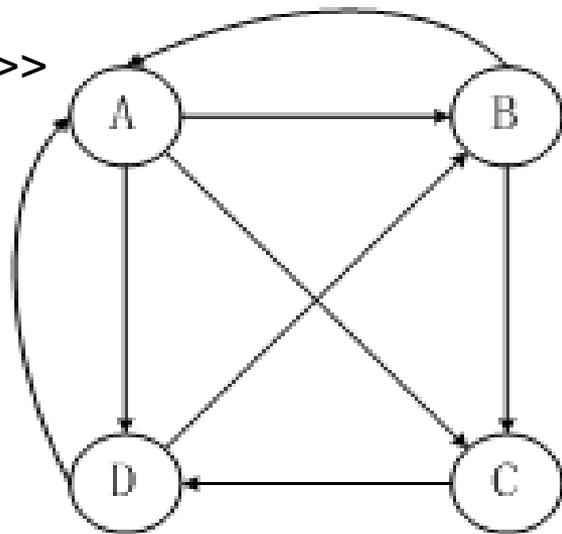
$\langle D_{\text{URL}}, \langle C_{\text{URL}}, 1/4 \rangle \rangle$

$\langle C_{\text{URL}}, D_{\text{URL}} \rangle$

$\langle A_{\text{URL}}, \langle D_{\text{URL}}, 1/8 \rangle \rangle$

$\langle B_{\text{URL}}, \langle D_{\text{URL}}, 1/8 \rangle \rangle$

$\langle D_{\text{URL}}, \langle A_{\text{URL}}, B_{\text{URL}} \rangle \rangle$





# 11.7.3 PageRank算法在MapReduce中的实现

## 2. 阶段2: PageRank分配 (Reduce阶段)

- 然后, 在该阶段的Reduce阶段, Reduce任务会获得 $\langle \text{URL}, \text{url\_list} \rangle$ 和 $\langle u, \langle \text{URL}, \text{cur\_rank}/|\text{url\_list}| \rangle \rangle$ , Reduce任务对于具有相同key值的value进行汇总, 并把汇总结果乘以d, 得到每个网页的新的PageRank值new\_rank, 然后输出 $\langle \text{URL}, \langle \text{new\_rank}, \text{url\_list} \rangle \rangle$ , 作为下一次迭代过程的输入。

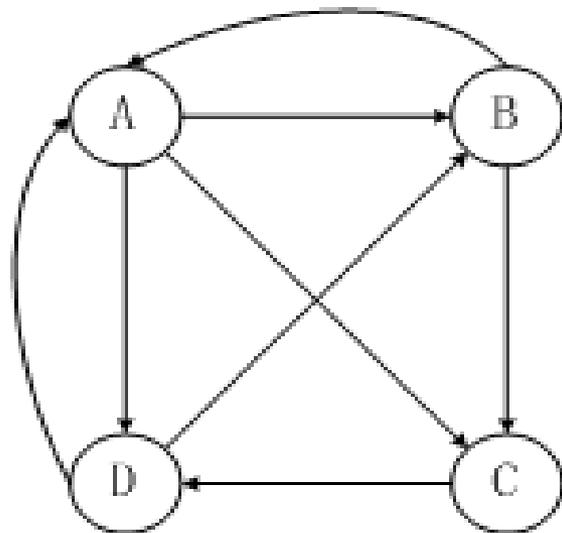
Reduce任务把第一次迭代后Map任务的输出作为自己的输入, 经过处理后, 阶段2的Reduce输出为:

```
<AURL, <0.2500, <BURL, CURL, DURL>>>  
<BURL, <0.2147, <AURL, CURL>>>  
<CURL, <0.2147, DURL>>  
<DURL, <0.3206, <AURL, BURL>>>
```

经过本轮迭代, 每个网页都计算得到了新的PageRank值。

下次迭代阶段2的Reduce输出为:

```
<AURL, <0.2200, <BURL, CURL, DURL>>>  
<BURL, <0.1996, <AURL, CURL>>>  
<CURL, <0.1996, DURL>>  
<DURL, <0.3808, <AURL, BURL>>>
```





## 11.7.3 PageRank算法在MapReduce中的实现

**Mapper**函数的伪码:

```
input <PageN, RankN> -> PageA,PageB,PageC
... // PageN外链指向PageA,PageB,PageC ...
begin
  Nn := the number of outlinks for PageN;
  for each outlink PageK
    output PageK -> <PageN, RankN/Nn>
  output PageN -> PageA, PageB, PageC ... //
  同时输出链接关系，用于迭代
end
/*****
```

Mapper输出如下（已经排序，所以PageK的数据排在一起，最后一行则是链接关系对）：

```
PageK -> <PageN1, RankN1/Nn1>
PageK -> <PageN2, RankN2/Nn2>
...
PageK -> <PageAk, PageBk, PageCk>
```

**Reducer**函数的伪码:

```
input mapper's output
begin
  RankK :=(1-beta)/N; //N为整个网络的网
  页总数
  for each inlink PageNi
    RankK += RankNi/Nni * beta
  //输出PageK及其新的PageRank值用于
  下次迭代
  output <PageK, RankK> -> <PageAk,
  PageBk, PageCk...>
end
```

该阶段是一个多次迭代过程，迭代多次后，当PageRank值趋于稳定时，就得出较为精确的PageRank值。



## 11.7.3 PageRank算法在MapReduce中的实现

### 3. 阶段3：收敛阶段

- 该阶段的任务就是由一个非并行组件决定是否达到收敛，如果达到收敛，就写出PageRank生成的列表。否则，回退到PageRank分配阶段的输出，作为新一轮迭代的输入，开始新一轮PageRank分配阶段的迭代
- 一般判断是否收敛的条件是所有网页的PageRank值不再变化，或者运行30次以后我们就认为已经收敛了



## 11.7.4 PageRank算法在Pregel和MapReduce中实现的比较

- PageRank算法在Pregel和MapReduce中实现方式的区别主要表现在以下几个方面：
  - (1) Pregel将PageRank处理对象看成是连通图，而MapReduce则将其看成是键值对
  - (2) Pregel将计算细化到顶点，同时在顶点内控制循环迭代次数，而MapReduce则将计算批量化处理，按任务进行循环迭代控制
  - (3) 图算法如果用MapReduce实现，需要一系列的MapReduce的调用。从一个阶段到下一个阶段，它需要传递整个图的状态，会产生大量不必要的序列化和反序列化开销。而Pregel使用超步简化了这个过程



# 11.8 Hama的安装和使用

11.8.1 Hama介绍

11.8.2 安装Hama的基本过程

11.8.3 运行Hama实例PageRank



## 11.8.1 Hama介绍

- Hama是Google Pregel的开源实现
- 与Hadoop适合于分布式大数据处理不同，Hama主要用于分布式的矩阵、graph、网络算法的计算
- Hama是在HDFS上实现的BSP(Bulk Synchronous Parallel)计算框架，弥补Hadoop在计算能力上的不足



## 11.8.2 安装Hama的基本过程

本实例中Hama具体运行环境如下：

- Ubuntu 14.04
- Java JDK 1.7
- Hadoop 2.6.0



## 11.8.2 安装Hama的基本过程

Hama（单机）安装步骤如下：

- (1) 安装好合适版本的JDK和Hadoop
- (2) 从官网下载Hama安装文件，比如Hama 0.7.0版本
- (3) 下载文件后，运用下面命令

```
sudo tar -zxf ~/下载/hama-dist-0.7.0.tar.gz -C /usr/local
```

解压至 /usr/local/hama，再运用下面命令

```
sudo mv ./hama-0.7.0/ ./hama
```

修改目录名称方便使用

- (4) 进入hama中的conf文件夹，修改hama-env.sh文件，在其中加入java的home路径，即加入：

```
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
```



## 11.8.2 安装Hama的基本过程

(5) 修改 hama-site.xml文件，这是hama配置的核心文件，具体内容如下：

```
<configuration>
```

```
  <property>
```

```
    <name>bsp.master.address</name>
```

```
    <value>local</value>
```

```
    <description>The address of the bsp master server. Either the  
    literal string "local" or a host:port for distributed mode
```

```
  </description>
```

```
</property>
```

```
  <property>
```

```
    <name>fs.default.name</name>
```

```
    <value>local</value>
```

```
    <description>
```

```
    The name of the default file system. Either the literal string  
    "local" or a host:port for HDFS.
```

```
  </description>
```

```
</property>
```



## 11.8.2 安装Hama的基本过程

```
<property>
```

```
  <name>hama.zookeeper.quorum</name>
```

```
  <value>localhost</value>
```

```
  <description>Comma separated list of servers in the ZooKeeper Quorum.  
  For example,
```

```
  "host1.mydomain.com,host2.mydomain.com,host3.mydomain.com".
```

```
  By default this is set to localhost for local and pseudo-distributed modes  
  of operation. For a fully-distributed setup, this should be set to a full  
  list of ZooKeeper quorum servers. If HAMA_MANAGES_ZK is set in hama-  
  env.sh
```

```
  this is the list of servers which we will start/stop zookeeper on.
```

```
  </description>
```

```
</property>
```

```
</configuration>
```



## 11.8.3 运行Hama实例PageRank

(1) 生成 randomgraph, 运行如下命令:

```
./bin/hama jar hama-examples-0.7.0.jar gen fastgen -v 100 -e 10 -o  
randomgraph -t 2
```

- 生成的文件位于 /usr/local/hama 下的 randomgraph。它表示100个节点，1000条边的数据，存储在两个文件中（part-00000,part-00001）。

```
hadoop@dsj-Lenovo:/usr/local/hama$ ls ./randomgraph  
part-00000  part-00001
```



## 11.8.3 运行Hama实例PageRank

(2)执行pagerank

```
./bin/hama jar hama-examples-0.7.0.jar pagerank -i randomgraph -o pagerankresult -t 4
```

运行结果保存在pagerankresult文件中

单机模式下，数据读取都是在本地文件系统，不需要读取HDFS中的文件。



# 本章小结

- 本章内容介绍了图计算框架Pregel的相关知识。传统的图计算解决方案无法解决大型的图计算问题，包括Pregel在内的各种图计算框架脱颖而出。
- Pregel并没有采用远程数据读取或者共享内存的方式，而是采用了纯消息传递模型，来实现不同顶点之间的信息交换。Pregel的计算过程是由一系列被称为“超步”的迭代组成的，每次迭代对应了BSP模型中的一个超步。
- Pregel已经预先定义好一个基类——Vertex类，编写Pregel程序时，需要继承Vertex类，并且覆写Vertex类的虚函数Compute()。在Pregel执行计算过程时，在每个超步中都会并行调用每个顶点上定义的Compute()函数。
- Pregel是为执行大规模图计算而设计的，通常运行在由多台廉价服务器构成的集群上。一个图计算任务会被分解到多台机器上同时执行，Pregel采用检查点机制来实现容错。
- Pregel作为分布式图计算的计算框架，主要用于图遍历、最短路径、PageRank计算等等。
- 本章最后通过对PageRank算法在MapReduce和Pregel上执行方式的不同进行比较，说明了Pregel解决图计算问题的优势。



# 附录：主讲教师林子雨简介



## 主讲教师：林子雨

单位：厦门大学计算机科学系

E-mail: ziyulin@xmu.edu.cn

个人网页: <http://www.cs.xmu.edu.cn/linziyu>

数据库实验室网站: <http://dblab.xmu.edu.cn>



扫一扫访问个人主页

林子雨，男，1978年出生，博士（毕业于北京大学），现为厦门大学计算机科学系助理教授（讲师），曾任厦门大学信息科学与技术学院院长助理、晋江市发展和改革局副局长。中国计算机学会数据库专业委员会委员，中国计算机学会信息系统专业委员会委员，荣获“2016中国大数据创新百人”称号。中国高校首个“数字教师”提出者和建设者，厦门大学数据库实验室负责人，厦门大学云计算与大数据研究中心主要建设者和骨干成员，2013年度厦门大学奖教金获得者。主要研究方向为数据库、数据仓库、数据挖掘、大数据、云计算和物联网，并以第一作者身份在《软件学报》《计算机学报》和《计算机研究与发展》等国家重点期刊以及国际学术会议上发表多篇学术论文。作为项目负责人主持的科研项目包括1项国家自然科学基金青年基金项目(No.61303004)、1项福建省自然科学基金项目(No.2013J05099)和1项中央高校基本科研业务费项目(No.2011121049)，同时，作为课题负责人完成了国家发改委城市信息化重大课题、国家物联网重大应用示范工程区域试点泉州市工作方案、2015泉州市互联网经济调研等课题。中国高校首个“数字教师”提出者和建设者，2009年至今，“数字教师”大平台累计向网络免费发布超过100万字高价值的研究和教学资料，累计网络访问量超过100万次。打造了中国高校大数据教学知名品牌，编著出版了中国高校第一本系统介绍大数据知识的专业教材《大数据技术原理与应用》，并成为京东、当当网等网店畅销书籍；建设了国内高校首个大数据课程公共服务平台，为教师教学和学生学习大数据课程提供全方位、一站式服务，年访问量超过50万次。具有丰富的政府和企业信息化培训经验，厦门大学管理学院EDP中心、浙江大学管理学院EDP中心、厦门大学继续教育学院、泉州市科技培训中心特邀培训讲师，曾给中国移动通信集团公司、福州马尾区政府、福建龙岩卷烟厂、福建省物联网科学研究院、石狮市物流协会、厦门市物流协会、浙江省中小企业家、四川泸州企业家、江苏沛县企业家等开展信息化培训，累计培训人数达3000人以上。



# 附录：《大数据技术原理与应用》教材



扫一扫访问教材官网

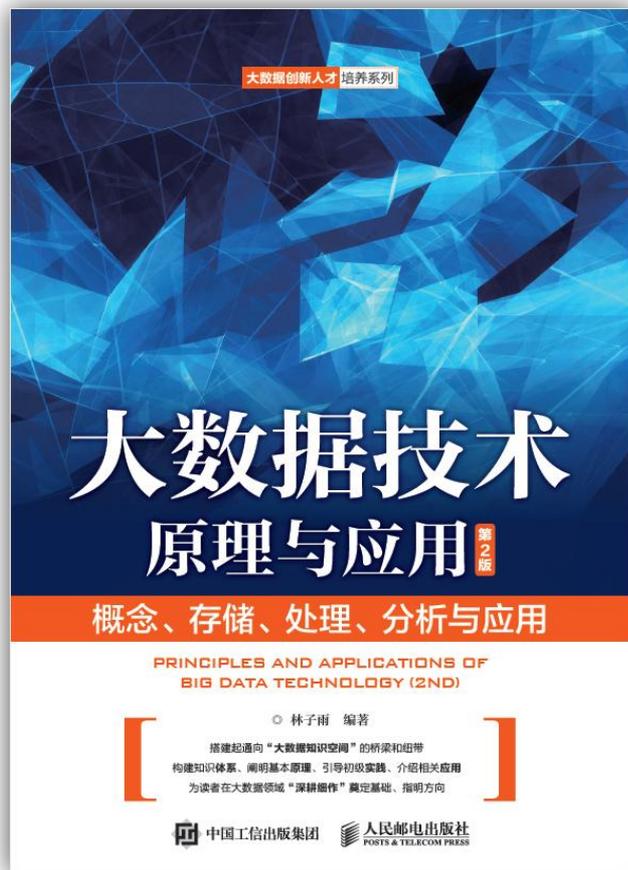
《大数据技术原理与应用——概念、存储、处理、分析与应用（第2版）》，由厦门大学计算机科学系林子雨博士编著，是中国高校第一本系统介绍大数据知识的专业教材。

全书共有15章，系统地论述了大数据的基本概念、大数据处理架构Hadoop、分布式文件系统HDFS、分布式数据库HBase、NoSQL数据库、云数据库、分布式并行编程模型MapReduce、Spark、流计算、图计算、数据可视化以及大数据在互联网、生物医学和物流等各个领域的应用。在Hadoop、HDFS、HBase和MapReduce等重要章节，安排了入门级的实践操作，让读者更好地学习和掌握大数据关键技术。

本书可以作为高等院校计算机专业、信息管理等相关专业的大数据课程教材，也可供相关技术人员参考、学习、培训之用。

欢迎访问《大数据技术原理与应用——概念、存储、处理、分析与应用》教材官方网站：

<http://dblab.xmu.edu.cn/post/bigdata>





# 附录：中国高校大数据课程公共服务平台



## 中国高校大数据课程 公共服务平台

<http://dblab.xmu.edu.cn/post/bigdata-teaching-platform/>



扫一扫访问平台主页



扫一扫观看3分钟FLASH动画宣传片

A group of silhouettes of people standing in a circle, holding hands, positioned at the top of the slide.

**Thank You!**

A group of silhouettes of people standing in a circle, holding hands, positioned at the bottom of the slide.

Department of Computer Science, Xiamen University, 2017