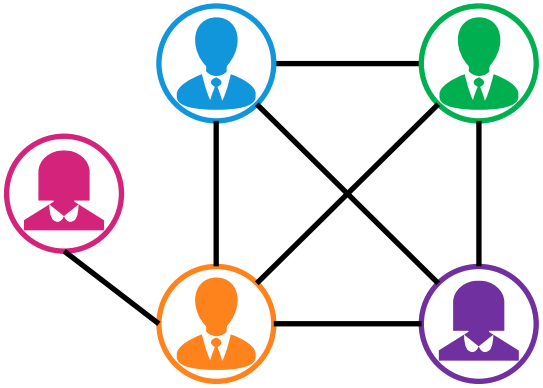
第十一章 图处理系统Giraph

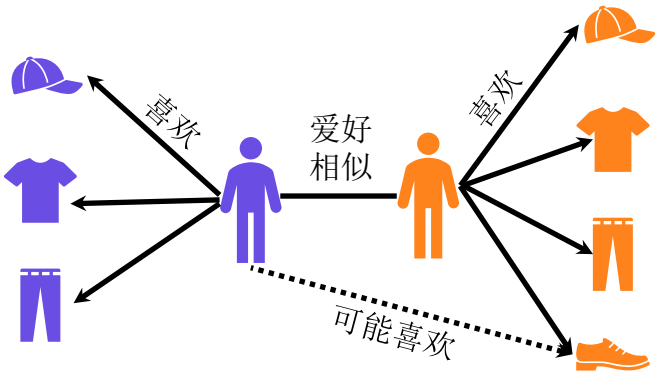
徐 辰
cxu@dase.ecnu.edu.cn

華東師範大學

图数据广泛存在

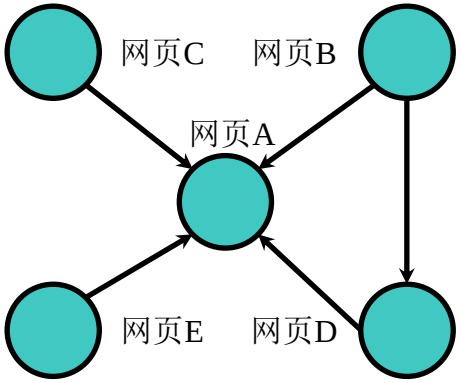
- 某些数据本身以图结构的形式呈现
  - 社交网络
  - 网购
  - 传染病传播途径
  - 交通路网





图数据广泛存在

- 某些非图结构的数据，也可以转换为图模型后进行处理
  - 网页链接（将网页视为顶点，链接视为边）
  - 机器学习训练数据（数据项看作顶点，顶点之间没有边）

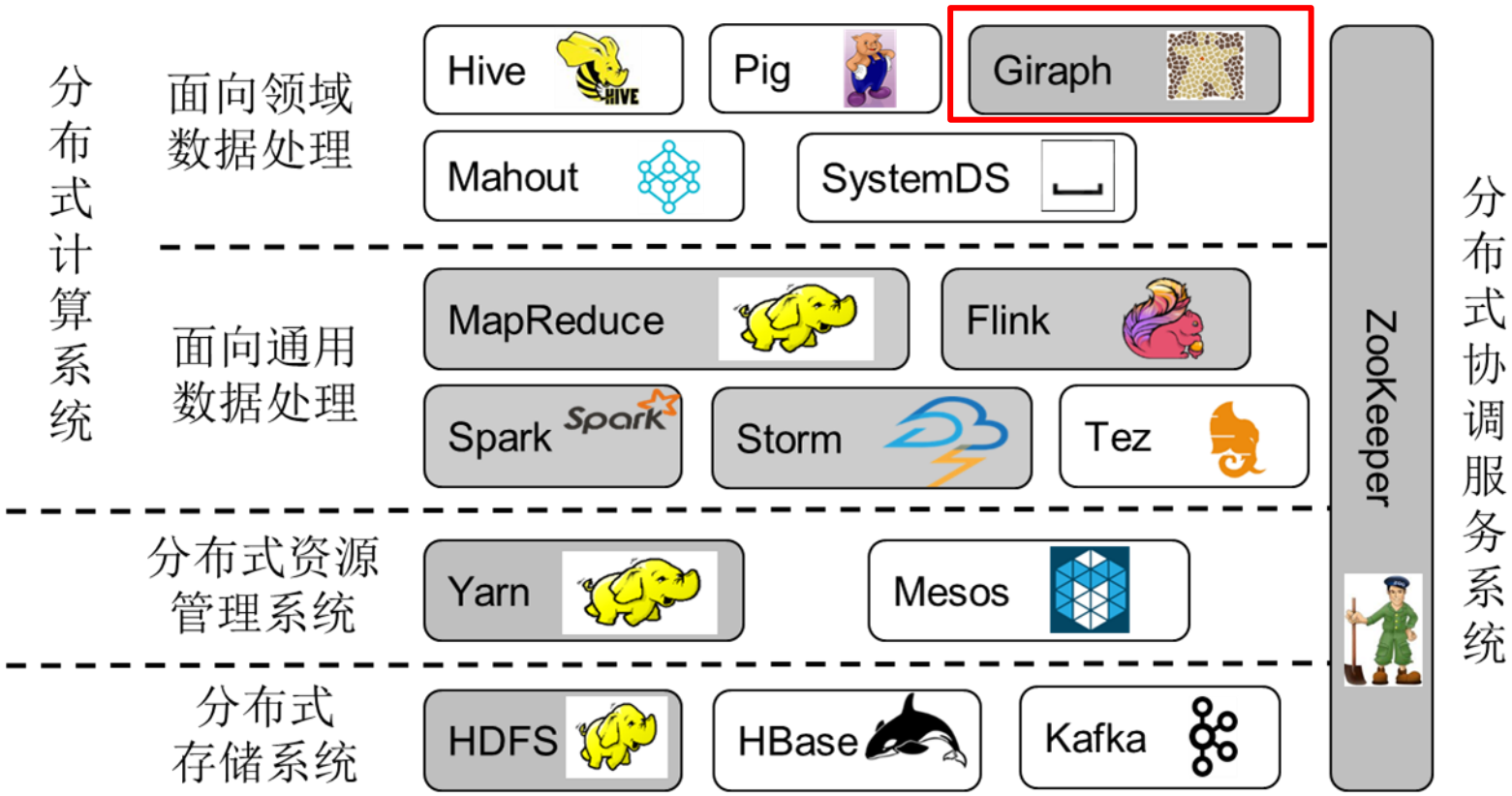


Giraph简介

- 2010年，谷歌发表Pregel论文
- 2012年，雅虎借鉴谷歌论文中的思想开发实现了Giraph，捐赠给Apache软件基金会
  - 利用MapReduce框架
  - 不是基于MapReduce API计算
- 创新转化
  - Open Graph

Meta

分布式计算系统生态圈


分布式计算系统
面向领域数据处理
Hive
Pig
Giraph
Mahout
SystemDS
面向通用数据处理
MapReduce
Flink
Spark
Storm
Tez
分布式资源管理系统
Yarn
Mesos
分布式存储系统
HDFS
HBase
Kafka
ZooKeeper
分布式协调服务系统

大纲

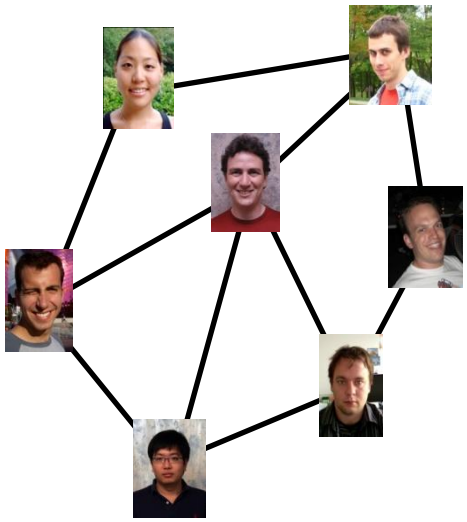
- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

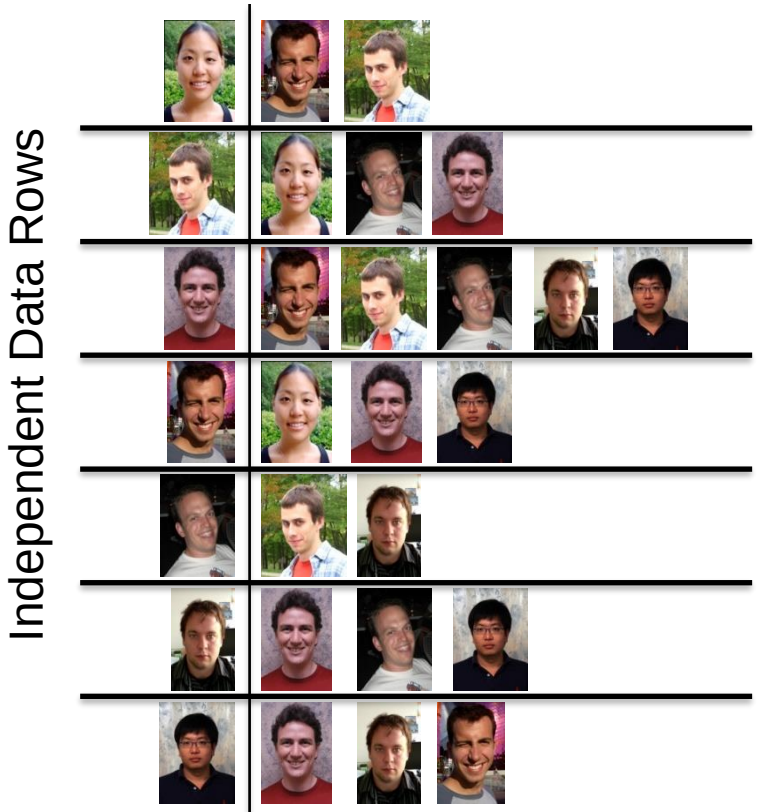
需求分析

- 对图处理算法通用
- 支持大规模图处理
- 自身具备容错能力
- 为图处理进行优化

MapReduce中图的表示

- 用户编写代码将图转变为key-value格式的文件，MapReduce系统并不认为这个文件是一张图

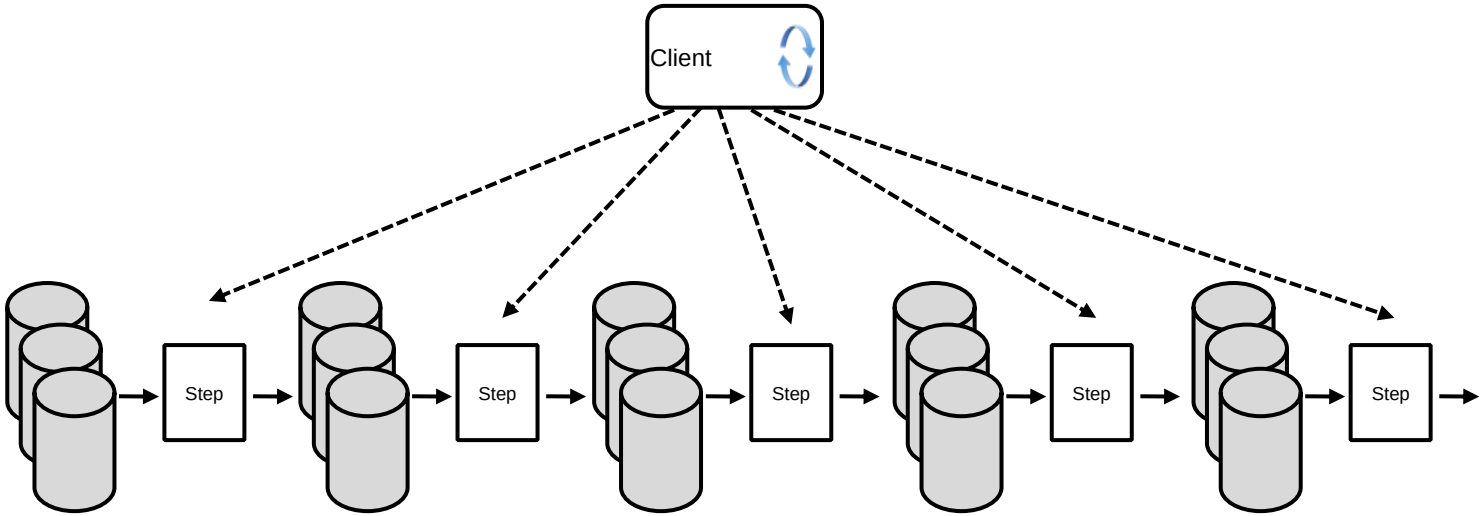




图片来自于 GraphLab: A New Parallel Framework for Machine Learning

MapReduce实现

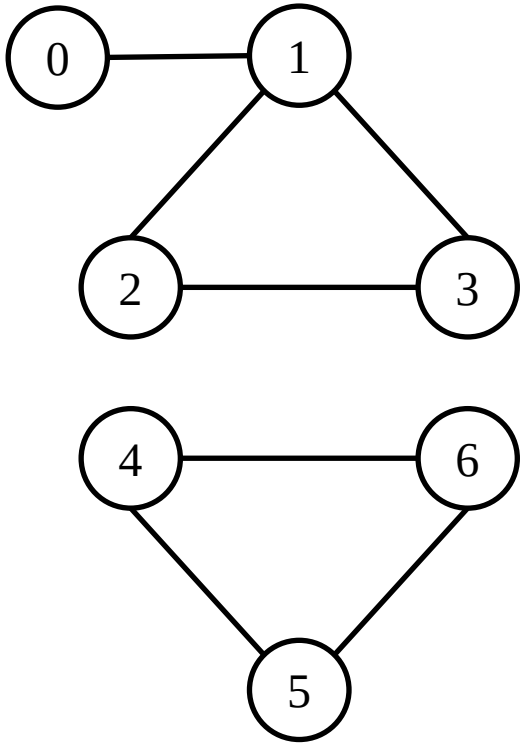
- 多个作业：迭代计算过程中每一迭代步结束时将结果写入HDFS，下一步将该结果再次从HDFS读出



大纲

- 设计思想
  - 数据模型
  - 计算模型
  - 迭代模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

邻接矩阵与邻接表

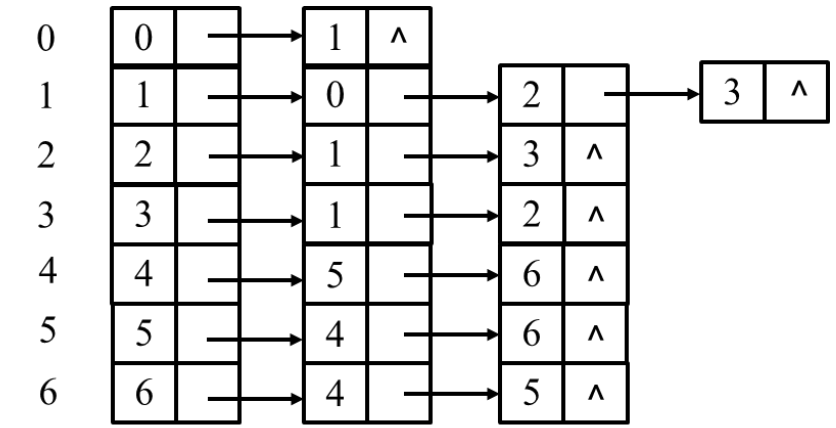


顶点数组：

0	1	2	3	4	5	6

边数组：

0	1	0	0	0	0	0
1	0	1	1	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	0	0	0	0	1	1
0	0	0	0	1	0	1
0	0	0	0	1	1	0



图数据模型

- 图是由顶点(Vertex)和边(Edge)组成的
  - vertexID：顶点具有的唯一的标识
  - value：与顶点关联的一个自定义的计算值，用于存储计算过程中所需保持的数值
    - 网页链接排名算法中顶点的Rank值
    - 单源最短路径算法中顶点与起始点间的距离等
  - weight：指图中由某一顶点与其邻居顶点确定的边上用户自定义的权值

图数据模型

- 图中的某一顶点及以其为起点的边可以抽象地表示如下：

  [vertexID, value, {[neighborID,weight], …}]

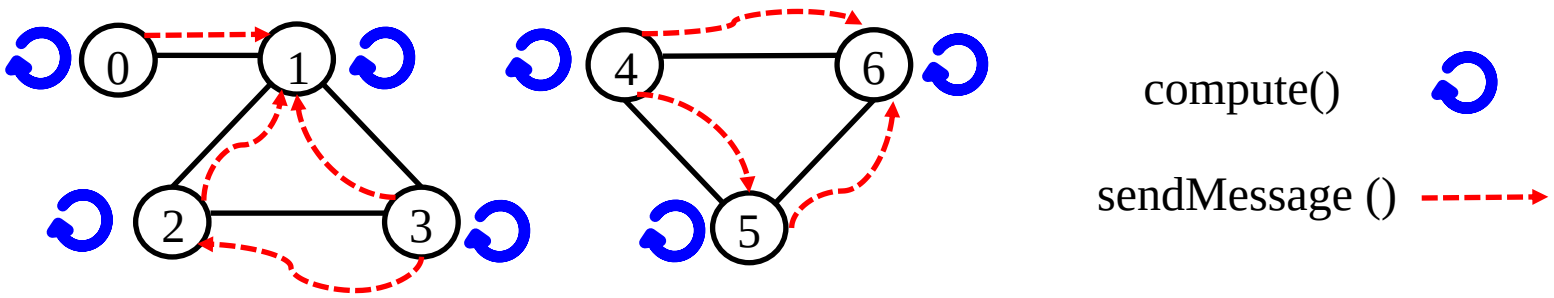
- <vetextID, neighborID>表示一条边，该边的权值为weight
  - 对于很多算法来说，边的权值固定不变

大纲

- 设计思想
  - 数据模型
  - 计算模型
  - 迭代模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Vertex-centric

- Giraph以顶点为主进行数据表示，采用以顶点为中心（Vertex-centric）的计算模型
- 顶点执行用户自定义函数进行相应计算
  - compute( )：计算并更新顶点的计算值(value)
  - sendMessage ( )：将消息传递给指定顶点



大纲

- 设计思想
  - 数据模型
  - 计算模型
  - 迭代模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

图算法的迭代

- 某一顶点的计算值在本轮迭代更新后，而其它顶点的计算值可能尚未完成该轮计算

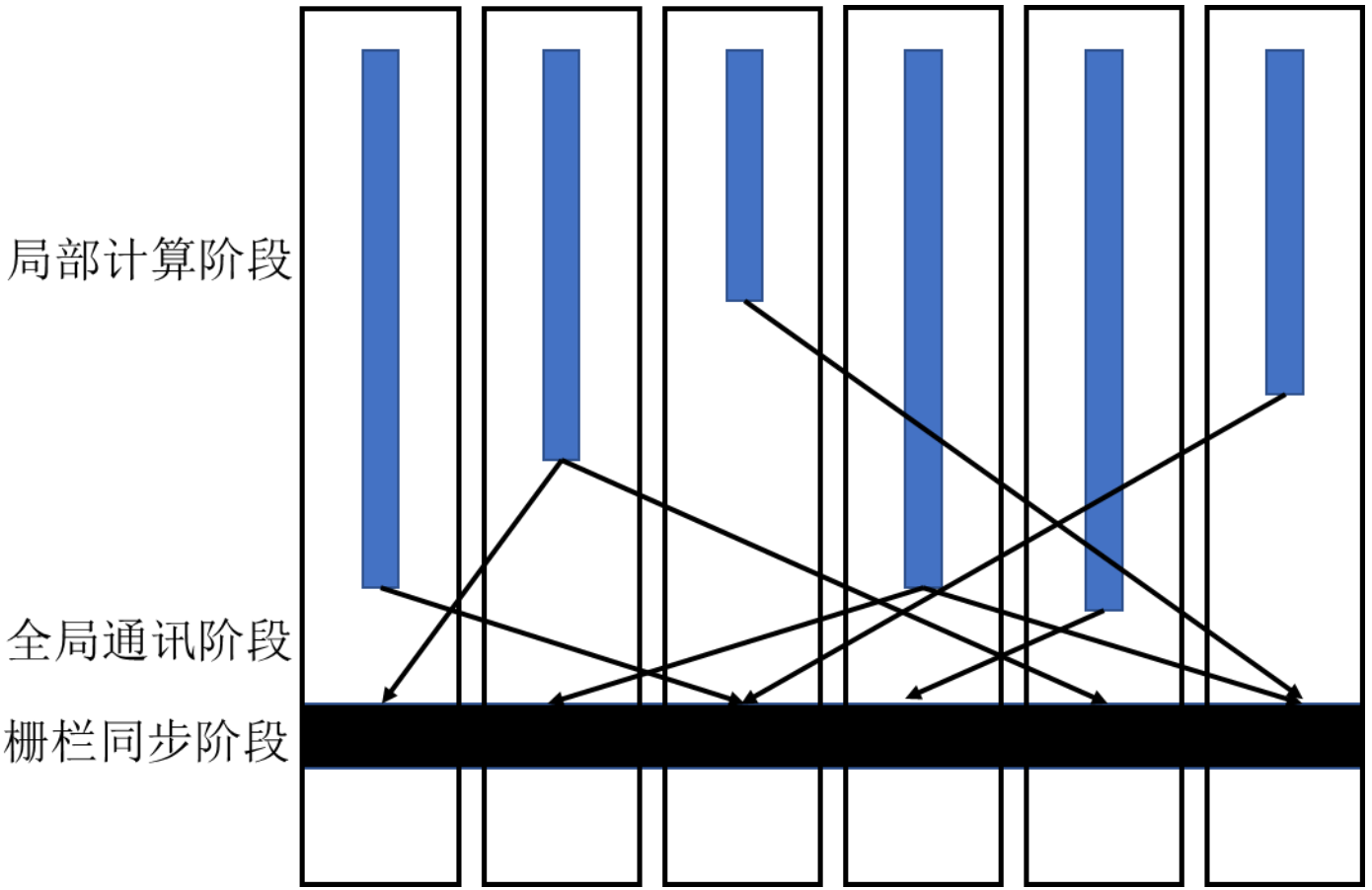
- 该顶点是否可立即进入下一轮迭代计算?
  - 不可以，BSP模型（Bulk Synchronous Parallel)
  - 可以，非BSP模型（本课程不讨论）

BSP迭代模型

- 一个BSP程序由一系列串行的超步(superstep)组成，超步内并行

- 一个超步分为三个阶段：
  - 局部计算阶段，每个处理器只对存储本地内存中的数据进行本地计算
  - 全局通信阶段，对任何非本地数据进行操作
  - 栅栏同步(Barrier Synchronization)阶段，等待所有通信行为的结束

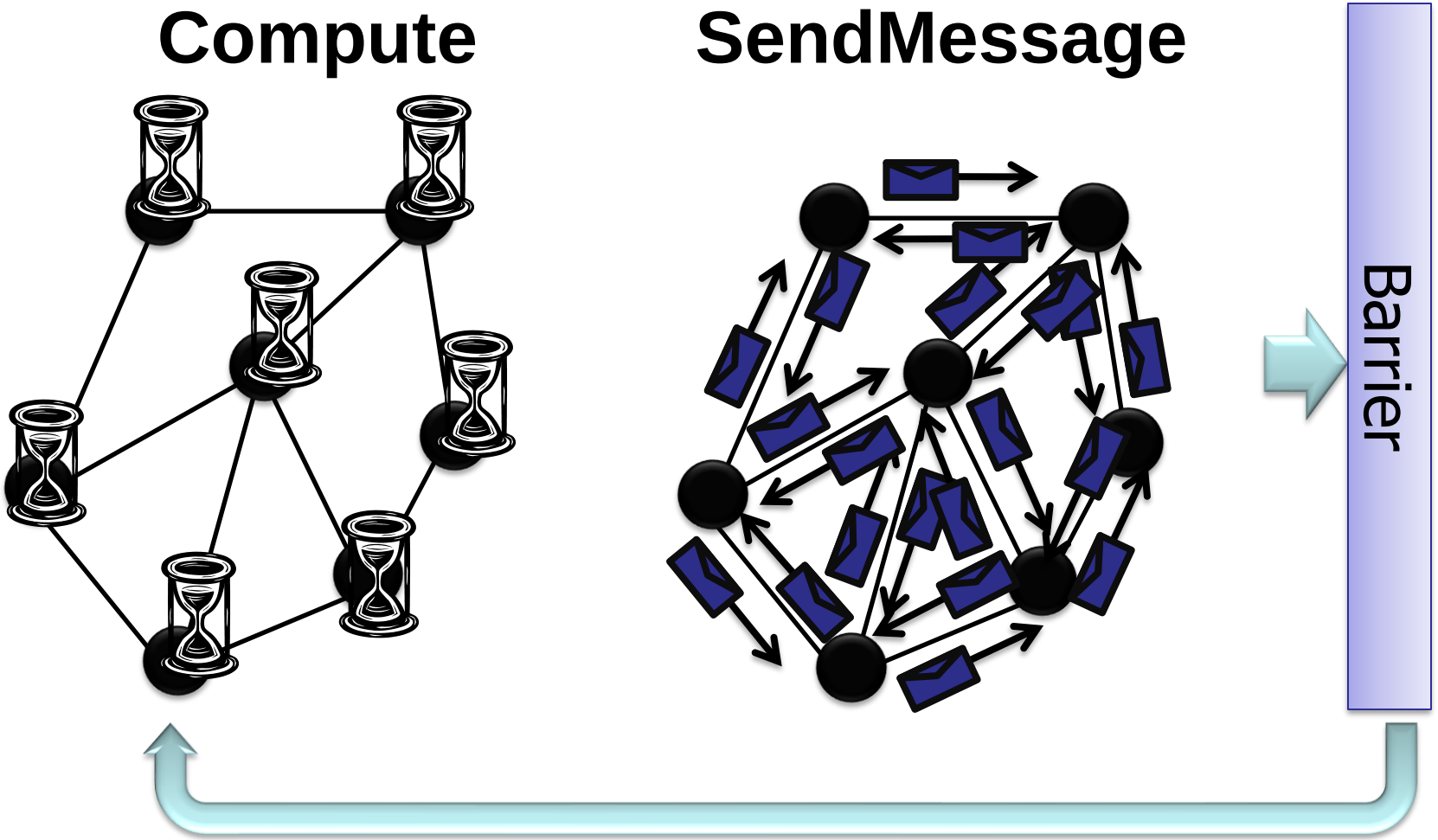
BSP迭代模型



图处理中的BSP迭代

- 在局部计算阶段，各个处理器对每个顶点调用用户自定义的compute()方法，完成顶点计算值的更新
- 在全局通信阶段，针对每个顶点调用的SendMessage ()方法，完成顶点之间的信息的交换
- 在栅栏同步阶段，每个顶点都要等待其它所有顶点完成计算并发出消息

图处理中的BSP迭代

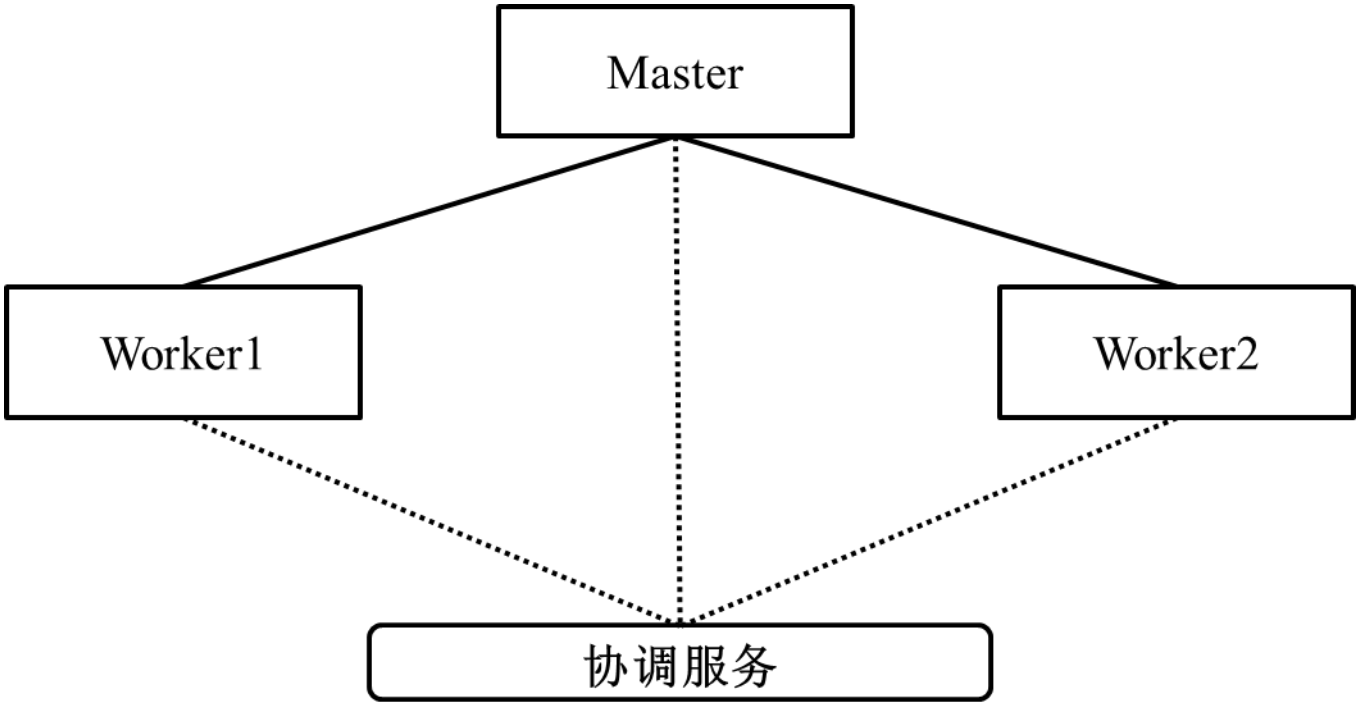


图片来自于 GraphLab: A New Parallel Framework for Machine Learning

大纲

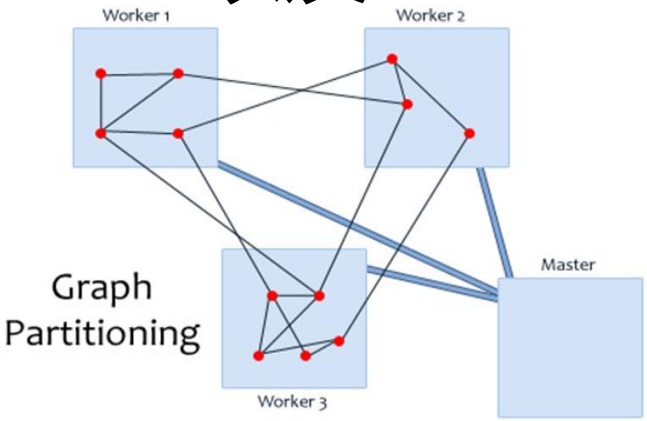
- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

抽象架构图



系统角色

- Master：负责给各个Worker分配任务
- Worker：系统将图进行了划分，形成若干个分区，每个Worker负责一个或多个分区并负责针对该分区的计算任务
- 协调服务(Coordination Service)：协调Master与worker以及worker之间



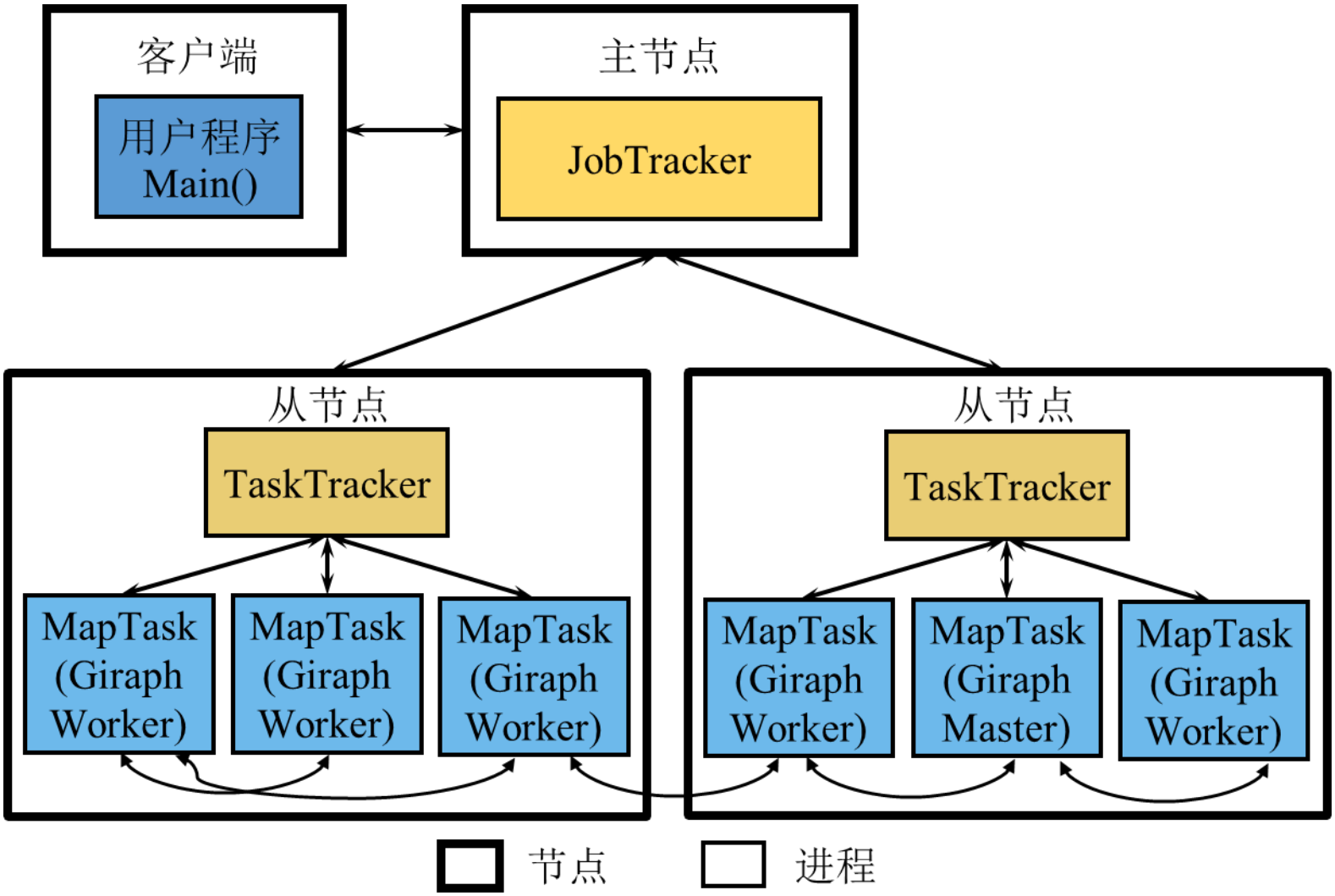
Worker

- 所管辖分区的顶点描述信息保存在内存中
  - 顶点与边的值：
    - 顶点的当前计算值
    - 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：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
    - 如果一个顶点V在超步S接收到消息，那么V将参与下一个超步的计算，意味着在下一个超步S+1中(而不是当前超步S中)处于“活跃”状态

Master

- Master负责协调各Worker执行任务，每个Worker均向Master发送自己的注册信息，Master会为Worker分配一个唯一的ID
- Master维护所有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
- Master维护的数据信息的大小，只与分区的数量有关，而与顶点和边的数量无关

Giraph与MapReduce



Giraph与MapReduce

- Giraph参考了Pregel体系架构，但在实现时借用了MapReduce框架启动Master和Worker这些进程
  - Giraph并没有像普通MapReduce程序那样编写Map和Reduce方法，而是将所有的图处理逻辑都在启动Map任务的过程中实现
  - 从MapReduce框架的角度来看，执行Giraph作业仅启动了Map任务

ZooKeeper in Giraph

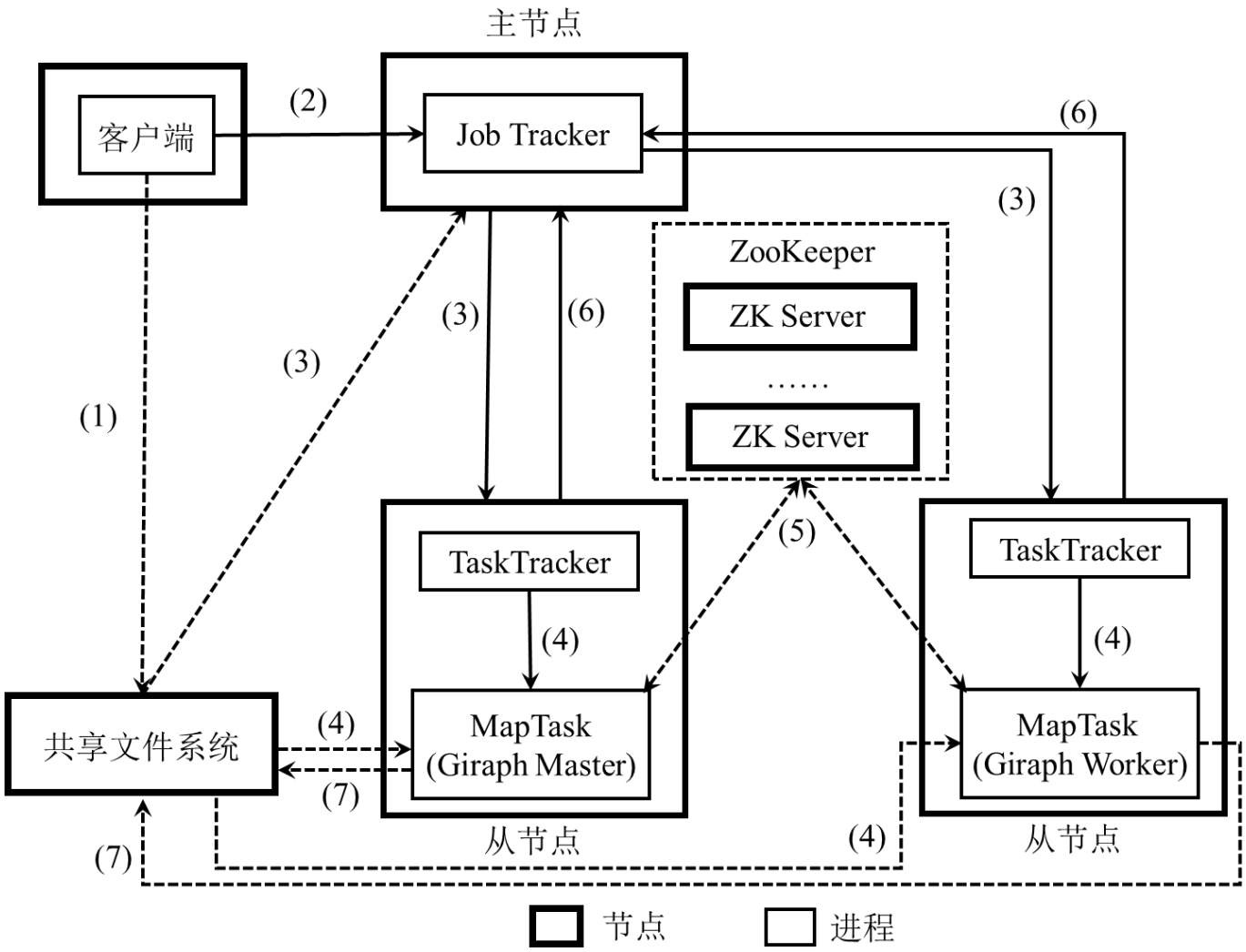
- 分布式选主：启动的Map任务中，有一个作为Giraph的Master，其余作为Worker。如何决定哪个Map任务作为Master?

- 栅栏同步：Zookeeper还可以实现BSP模型中栅栏同步的功能

大纲

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程示例

应用程序执行流程


主节点
客户端
(2)
Job Tracker
(6)
(3)
ZooKeeper
ZK Server
.....
ZK Server
(3)
(6)
(3)
(1)
(5)
TaskTracker
(4)
MapTask
(Giraph Master)
TaskTracker
(4)
MapTask
(Giraph Worker)
共享文件系统
(4)
(7)
从节点
从节点
(7)
(4)
节点
进程

应用程序执行流程

1. Client将用户编写的Giraph作业配置信息、jar包等上传到共享的文件系统(例如：HDFS)
2. Client提交作业给JobTracker，告诉JobTracker作业信息的位置
3. JobTracker读取作业的信息，生成一系列Map任务，调度给有空闲slot的TaskTracker
4. TaskTracker根据JobTacker的指令启动Child进程执行Map任务，Map任务将从诸如HDFS等共享文件系统或其它存储图数据的存储系统读取输入数据

应用程序执行流程

5. 多个Map任务通过ZooKeeper进行选主，其中一个Map任务作为Giraph的Master，其它作为Giraph的Worker，并且Master和Worker通过Zookeeper协调执行Giraph作业
6. JobTracker从TaskTracker处获得Giraph的Master和Worker进度信息
7. 任务完成计算后将结果写入共享文件系统，则意味着整个作业执行完毕

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Giraph工作过程

- 读入数据经过一系列超步计算后将结果输出，大体划分为三个阶段：
  - 数据输入阶段：如何针对输入的图数据进行划分？
  - 迭代计算阶段：如何进行超步计算以及同步控制？
  - 数据输出阶段：各Worker对自己负责分区的计算结果写入HDFS等持久化存储系统

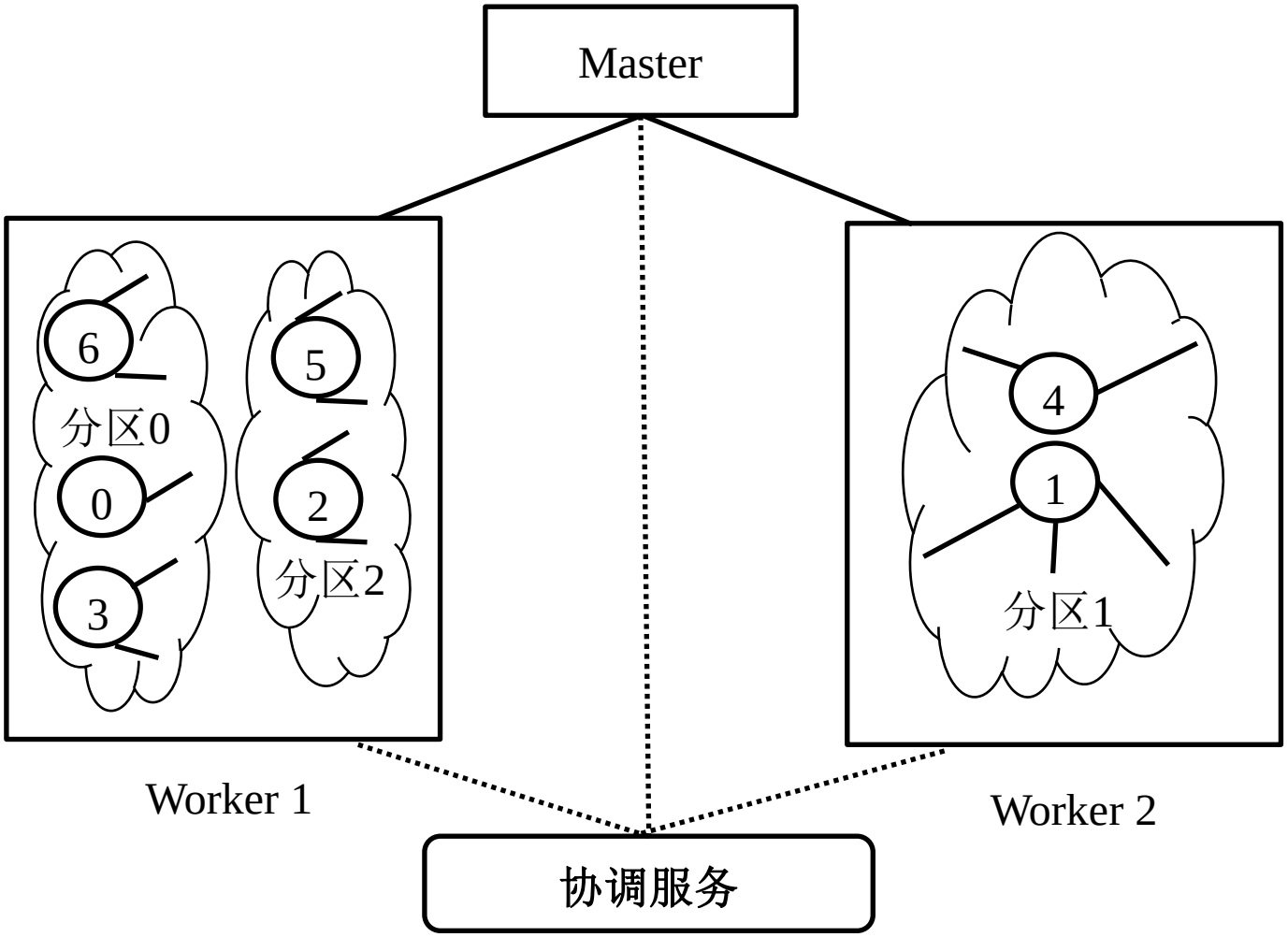
大纲

- 设计思想
- 体系架构
- 工作原理
  - 数据划分
  - 超步计算
  - 同步控制
- 容错机制
- 编程示例

图数据的分区

- 分区：一组顶点和以这些顶点为起点的边
  - 按照顶点的标识（vertexID）决定该顶点属于哪一个分区
  - 默认的划分方法：hash(vertexID) mod N
    - N为所有分区总数
- 无论哪一个Worker，只要将顶点的ID通过划分方法计算就可以知道该顶点属于哪个分区

图数据的分区



输入数据的分区

- 输入数据的分区与Giraph需要的分区往往是不一致的
  - 例如，存储在HDFS上的图数据以一个大文件的形式存在并按照顶点的ID排序，而Giraph系统希望顶点按照ID模取5进行划分

- 数据划分实际上是要完成输入数据到Giraph期望分区的调整，这一过程是由Master和Worker共同完成的

边加载边划分

- Master
  - 将输入的图数据根据Worker数量初始分解为多个部分，为每个Worker分配一部分数据
- Worker
  - 读取初始分配的部分，并根据划分方法计算该顶点是否属于当前Worker负责的分区
    - 若不属于，那么当前Worker将该顶点发给其所属的分区所在的Worker上

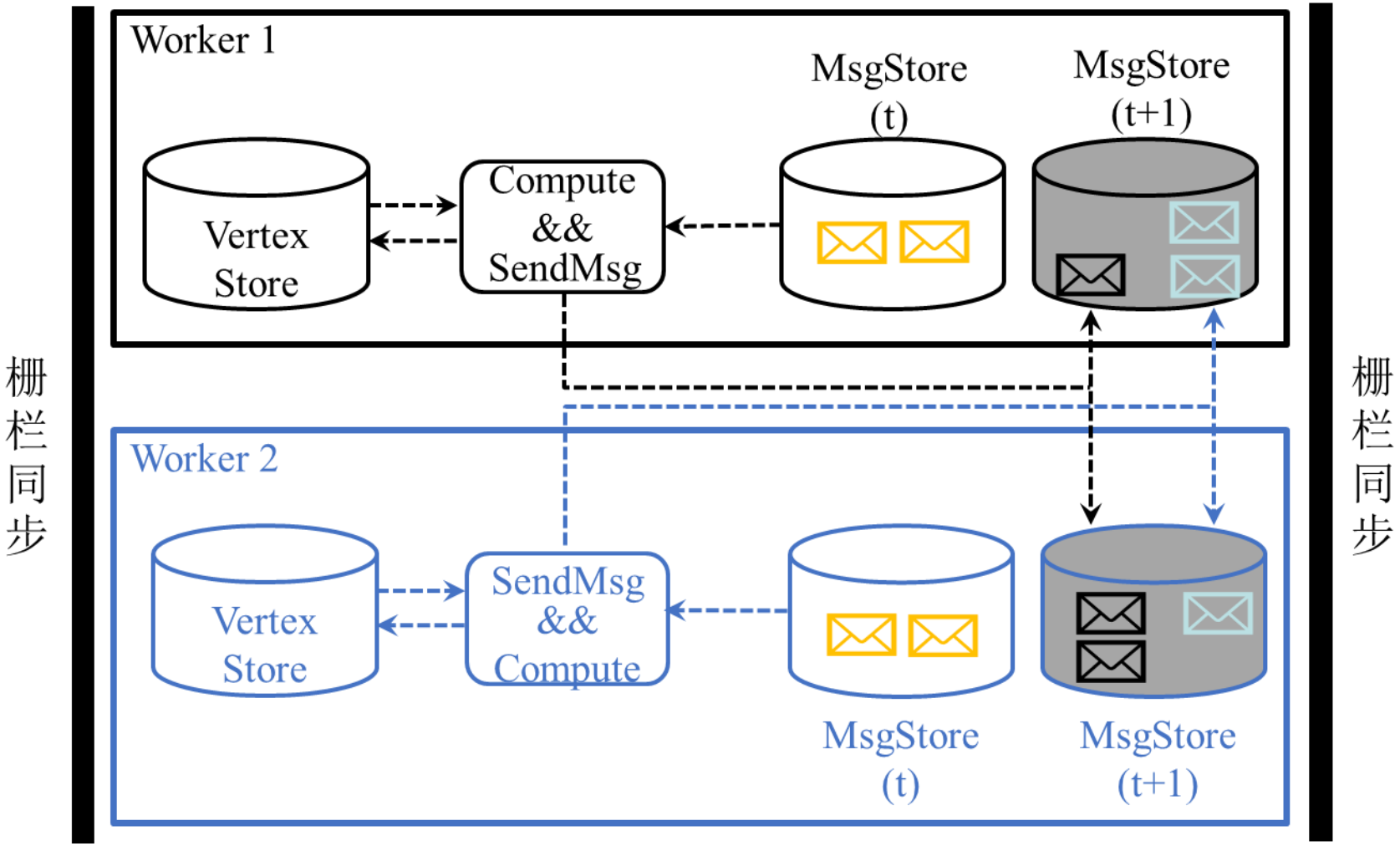
大纲

- 设计思想
- 体系架构
- 工作原理
  - 数据划分
  - 超步计算
  - 同步控制
- 容错机制
- 编程示例

超步计算

- 当超步开始时，Worker针对每个顶点调用compute方法，并根据获取的前一个超步的消息更新该顶点的计算值
  - 在计算过程中， Worker将更新后的计算值以消息形式发送给邻居顶点
- 在超步结束的时候，Worker需要进行同步控制，保证所有的消息都已经发送并接收成功

超步t的执行过程


Worker 1
MsgStore
(t)
MsgStore
(t+1)
Vertex
Store
Compute
&&
SendMsg
栅栏同步
Worker 2
Vertex
Store
SendMsg
&&
Compute
MsgStore
(t)
MsgStore
(t+1)
栅栏同步

VertexStore和MsgStore

- VertexStore：存储顶点标识、计算值以及边的信息
- MsgStore：存储两份消息
  - MsgStore (t) 存储在超步t-1全局通信阶段各顶点获取的消息，这些消息用于超步t中顶点的计算
  - MsgStore (t+1)存储在超步t全局通信阶段各顶点获取的消息，意味着将用于超步t+1中顶点的计算

StatStore

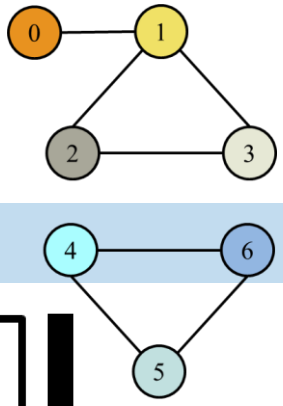
- 在超步t中，Giraph也需要存储两份标志位信息
  - StatStore (t) 指示了当前超步t中处于活跃状态的顶点
  - StatStore (t+1)指示了超步t+1中处于活跃状态的顶点

- 顶点的状态
  - 活跃：该顶点参与计算
  - 非活跃：该顶点不执行计算

超步t中某一顶点的处理过程

1. 对于StatStore(t)中的某一活跃顶点，从MsgStore(t)读取属于该顶点的消息，并调用compute方法更新VertexStore
2. 将更新后的计算值以消息形式发送给邻居顶点
3. 若无其它顶点发来的消息，则将StatStore(t+1)中该顶点的状态设置为非活跃顶点，意味着该顶点不参与下一个超步。否则，将消息存入MsgStore(t+1)，用于下一个超步，并将顶点的状态设置为活跃顶点

无向图连通分量超步1执行过程



栅栏同步

Worker 1

Vertex Store

Compute && SendMsg

MsgStore (1)

MsgStore (2)

Worker 2

Vertex Store

SendMsg && Compute

MsgStore (1)

MsgStore (2)

栅栏同步

大纲

- 设计思想
- 体系架构
- 工作原理
  - 数据划分
  - 超步计算
  - 同步控制
- 容错机制
- 编程示例

Worker间的同步

- Giraph中多个Worker同时进行超步计算，而多个Worker之间需要进行同步
- 根据BSP模型，同步控制需要确保即所有Worker都完成超步t后再进入超步t+1
- Giraph中的Master和Worker通过ZooKeeper来完成同步

迭代的结束（收敛）

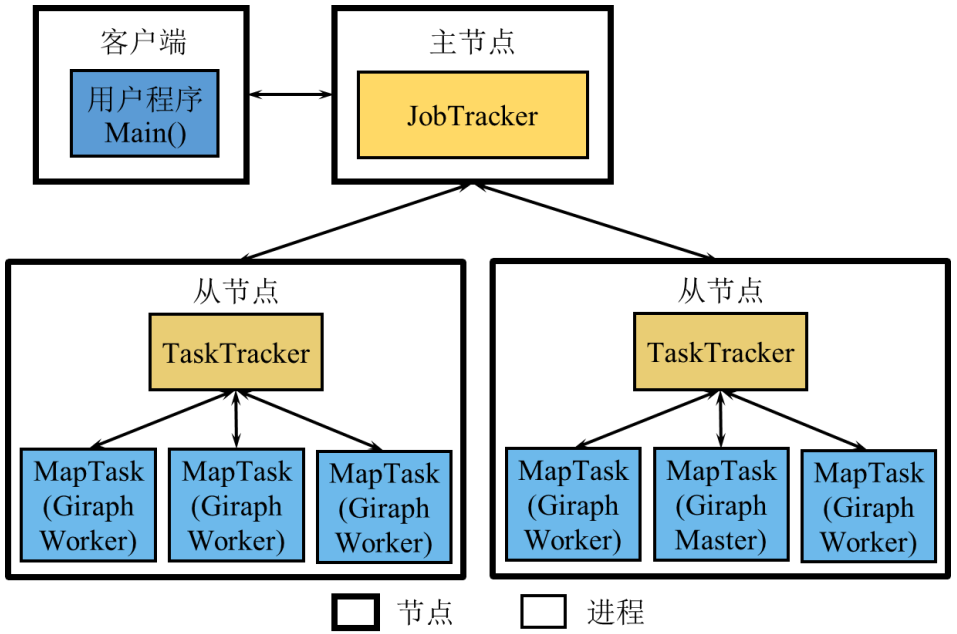
- 超步同步完成后，Worker根据StatStore(t+1)信息把在下一个超步还处于活跃状态的顶点数量报告给Master
- 如果Master收集到所有Worker中活跃顶点数量之和为0，意味着迭代过程可以结束
- Master给所有Worker发送指令，通知每个Worker对自己的计算结果进行持久化存储

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

故障类型：MapReduce

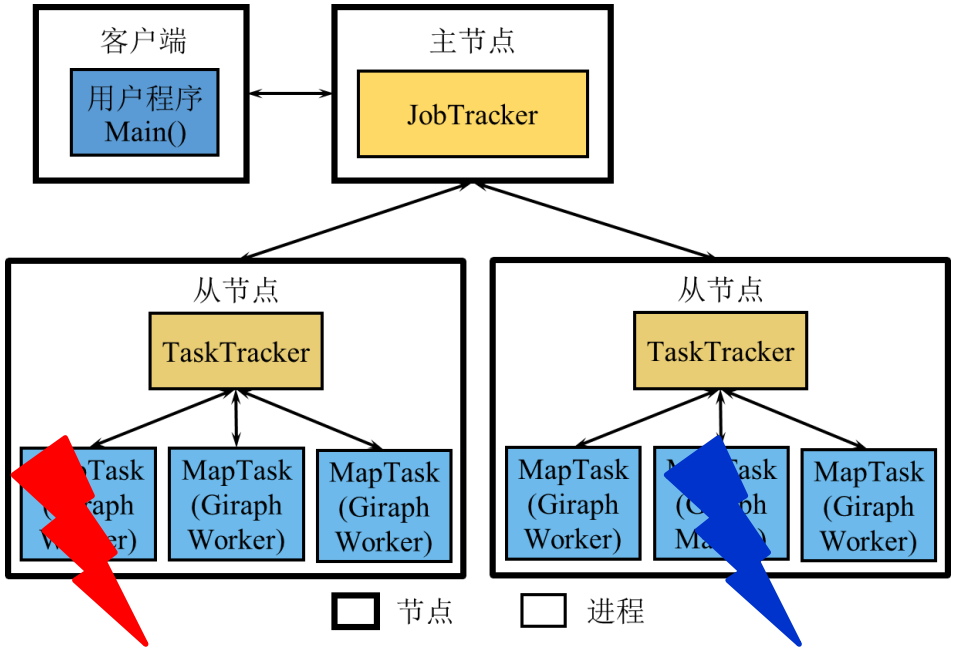
- 主节点故障
  - JobTracker故障：如宕机引起
- 从节点故障
  - TaskTracker故障：如节点宕机引起
  - Task故障: 如JVM内存不够退出



思考：是否可以直接利用MapReduce的容错机制?

故障类型：Giraph

- Master故障
  - 意味着主控节点丢失，整个作业将失败
  - 怎么办?
- Worker故障
  - 该Worker维护的计算信息丢失
  - 需设置检查点



大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 检查点
  - 故障恢复
- 编程示例

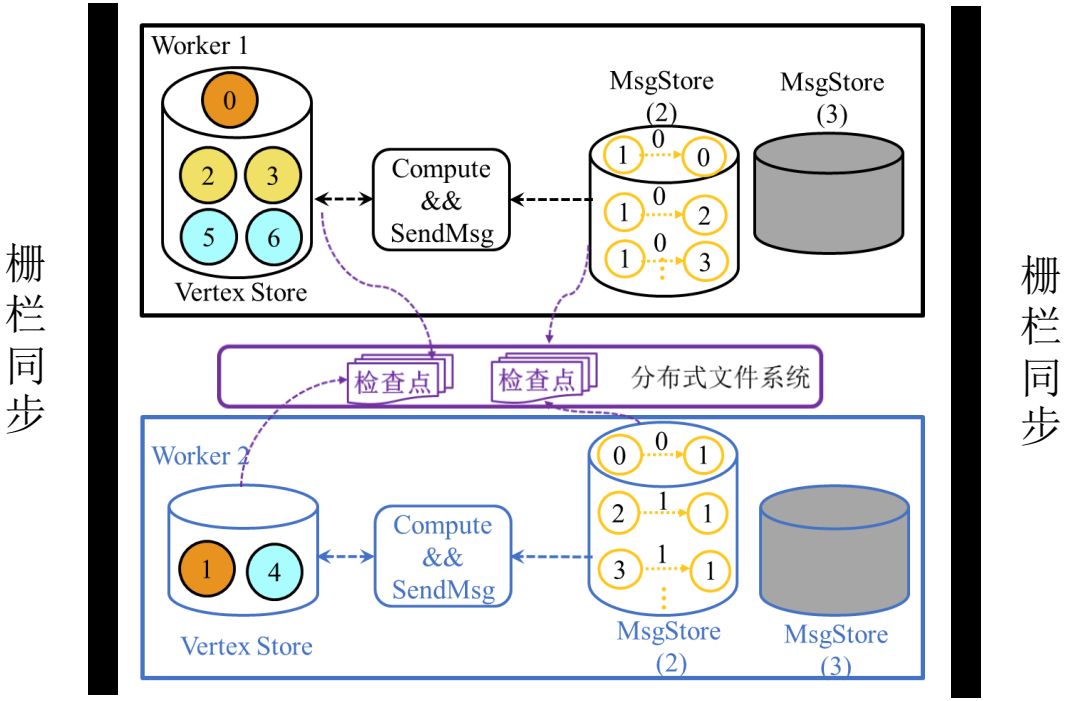
检查点

- Giraph允许用户设置写检查点的间隔
  - 默认情况下，该间隔的值为0，即不写检查点
  - 如果该间隔值不为0，那么每隔的一定超步将进行写检查点

- 在需要写检查点的超步开始时
  - Master通知所有Worker把管辖的分区信息（顶点、边、接收到的消息、标志位等）写入到持久化存储系统
  - Master保存Aggregator的值

检查点

- 在计算无向图连接分量的例子中，假设设置检查点的间隔为2，那么该计算过程在超步0和超步2开始时，系统保存检查点

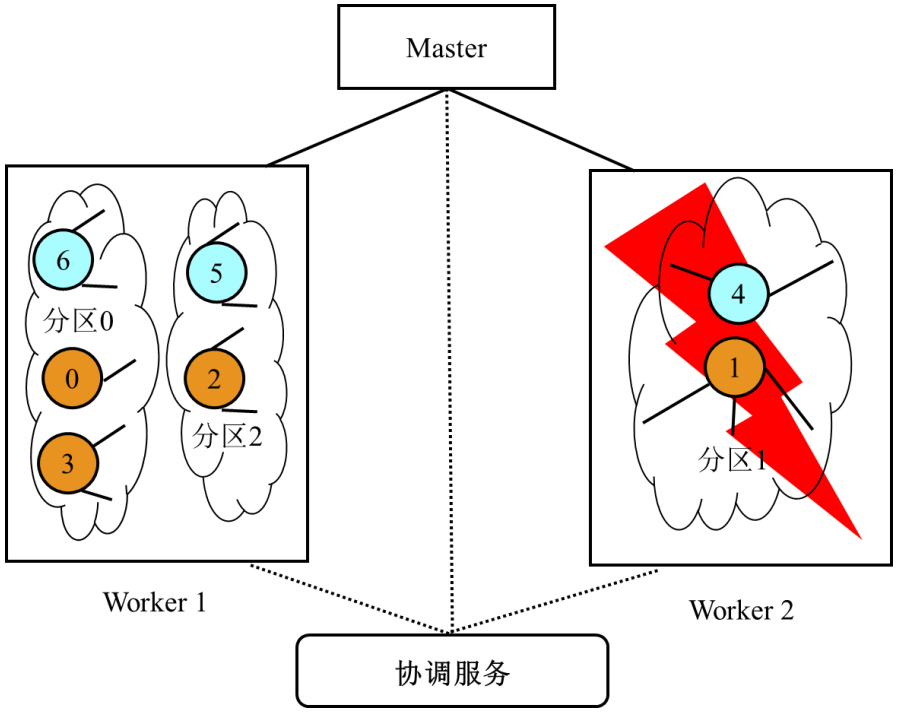


大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 检查点
  - 故障恢复
- 编程示例

故障

- 在Giraph的实现中，当一个或多个Worker发生故障，被分配到这些Worker的分区信息就丢失了



恢复

- MapReduce的容错机制将重启新的Map任务，因而Giraph中有新的Worker产生
- Giraph中的Master将给所有的Worker重新分配分区
  - 如果用户没有设置检查点：Worker重新载入输入数据，从头开始计算
  - 如果用户设置了检查点：Worker回滚到最近的检查点所在的超步S，在超步S开始时，从检查点中重新加载信息并继续执行计算

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Compute方法框架


```
import org.apache.giraph.graph.*;
import org.apache.hadoop.io.*;
......

/* 步骤1：确定顶点标识I、顶点的计算值V、边的权值E以及消息值M的数据类型 */
public class CustomComputation extends BasicComputation<I的数据类型, V的数据类型, E的数据
    类型, M的数据类型> {

  @Override
  public void compute(Vertex<I的数据类型, V的数据类型, E的数据类型> vertex, Iterable<M的
      数据类型> messages) {
    /* 步骤2：实现顶点计算值的更新以及发送消息 */
    ......
  }
}
```

主方法框架

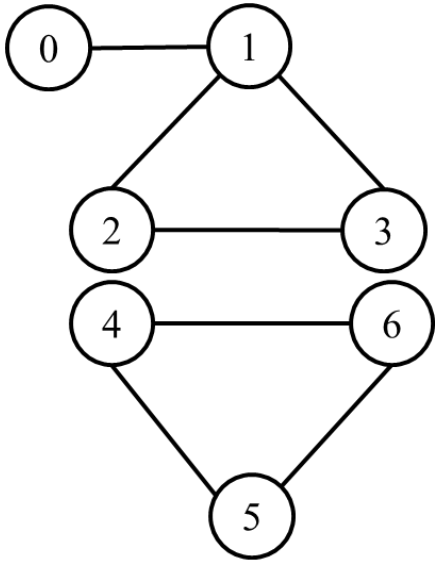

```java
public class CustomRunner extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    /* 步骤1:  设置作业的信息 */
    GiraphConfiguration giraphConf = new GiraphConfiguration(getConf());

    // 设置compute方法
    giraphConf.setComputationClass(CustomComputation.class);
    // 设置图数据的输入格式
    giraphConf.setVertexInputFormatClass(图数据的输入格式类);
    //设置图数据的输出格式
    giraphConf.setVertexOutputFormatClass(图数据的输出格式类);

    // 启用本地调试模式
    giraphConf.setLocalTestMode(true);
    // 最小和最大的Worker数量均为1，Master协调超步时所需Worker响应的百分比为100
    giraphConf.setWorkerConfiguration(1, 1, 100);
    // Master和Worker位于同一进程
    GiraphConstants.SPLIT_MASTER_WORKER.set(giraphConf, false);

    // 创建Giraph作业
    GiraphJob giraphJob = new GiraphJob(giraphConf, getClass().getSimpleName());

    // 设置图数据的输入路径
    GiraphTextInputFormat.setVertexInputPath(giraphConf, new Path(args[0]));
    // 设置图数据的输出路径
    GiraphTextOutputFormat.setOutputPath(giraphJob.getInternalJob(), new Path(args[1]));

    return giraphJob.run(true) ? 0 : -1;
  }

  public static void main(String[] args) throws Exception {
    /* 步骤2:  运行作业 */
    int exitCode = ToolRunner.run(new CustomRunner(), args);
    System.exit(exitCode);
  }
}
```

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 连通分量
  - 单源最短路径
  - 网页链接排名
  - K均值聚类

连通分量

- 输入：保存在文本文件中，每行由顶点标识及该顶点出边指向的邻居顶点标识组成
- 输出：保存在文本文件中，每行由顶点标识以及保存了连通分量编号的计算值组成



输入	输出
0 1 1 0 2 3 2 1 3 3 1 2 4 5 6 5 4 6 6 4 5	0 0 1 0 2 0 3 0 4 4 5 4 6 4

解决方案

- 顶点计算值的更新过程
  - 超步0时，以顶点的标识将各顶点的计算值初始化为连通分量编号并向所有的邻居顶点发送消息
  - 其余超步从接受到的消息中挑选出最小的连通分量的编号并更新计算值，且每个顶点仅在计算值发生更新时才向邻居顶点发送消息，对应的消息内容为计算值中保存的连通分量编号

编写compute方法


```
/* 步骤1：确定顶点标识I、顶点的计算值V、边的权值E以及消息值M的数据类型 */
public class ConnectedComponentsComputation
    extends BasicComputation<IntWritable, IntWritable, NullWritable, IntWritable> {

  @Override
  public void compute(Vertex<IntWritable, IntWritable, NullWritable> vertex,
      Iterable<IntWritable> messages) throws IOException {
    /* 步骤2：编写与顶点计算、更新相关的处理逻辑以及发送消息 */
    // 超步0时向所有邻居顶点发送消息
    if (getSuperstep() == 0) {
      sendMessageToAllEdges(vertex, vertex.getValue());
      vertex.voteToHalt();
      return;
    }

    boolean changed = false;
    int currentComponent = vertex.getValue().get();
    // 从消息中挑选出最小的连通分量编号
    for (IntWritable message : messages) {
        int candidateComponent = message.get();
        if (candidateComponent < currentComponent) {
          currentComponent = candidateComponent;
          changed = true;
        }
      }

      if (changed) {
        // 更新计算值并向邻居顶点发送消息
        vertex.setValue(new IntWritable(currentComponent));
        sendMessageToAllEdges(vertex, vertex.getValue());
      }
      vertex.voteToHalt();
    }
  }
```

编写主方法


```
public class ConnectedComponentsRunner extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    /* 步骤1:  设置作业的信息 */
    GiraphConfiguration giraphConf = new GiraphConfiguration(getConf());

    // 设置compute方法
    giraphConf.setComputationClass(ConnectedComponentsComputation.class);
    // 设置图数据的输入格式
    giraphConf.setVertexInputFormatClass(IntIntNullTextInputFormat.class);
    // 设置图数据的输出格式
    giraphConf.setVertexOutputFormatClass(IdWithValueTextOutputFormat.class);

    // 启用本地调试模式
    giraphConf.setLocalTestMode(true);
    // 最小和最大的Worker数量均为1，Master协调超步时所需Worker响应的百分比为100
    giraphConf.setWorkerConfiguration(1, 1, 100);
    // Master和Worker位于同一进程
    GiraphConstants.SPLIT_MASTER_WORKER.set(giraphConf, false);

    // 创建Giraph作业
    GiraphJob giraphJob = new GiraphJob(giraphConf, getClass().getSimpleName());
    // 设置图数据的输入路径
    GiraphTextInputFormat.addVertexInputPath(giraphConf, new Path(args[0]));
    // 设置图数据的输出路径
    GiraphTextOutputFormat.setOutputPath(giraphJob.getInternalJob(), new Path(args[1]));

    return giraphJob.run(true) ? 0 : -1;
  }

  public static void main(String[] args) throws Exception {
    /* 步骤2:  运行作业 */
    int exitCode = ToolRunner.run(new ShortestPathRunner(), args);
    System.exit(exitCode);
      }
    }
```

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 连通分量
  - 单源最短路径
  - 网页链接排名
  - K均值聚类

单源最短路径

- 输入：保存在文本文件中，每行表示有向图的一个顶点距离源点的距离以及该顶点距离邻居顶点的距离
  - [标识, 计算值,[[邻居顶点,出边的权值],…]]
- 输出：保存在文本文件中，每行由顶点标识和最短路径的长度组成

输入	输出
[0,-1, [[1,1.0], [3,3.0]]] [1,-1, [[2,1.0]]] [2,-1, [[0,1.0], [1,1.0]]] [3,-1, [[1,1.0], [2,1.0]]]	0 0.0 1 1.0 2 2.0 3 3.0

解决方案

- 顶点计算值的更新过程
  - 超步0将顶点计算值初始化为无穷大
  - 其余超步从接收的路径值消息中挑选出最小路径值，并在该值小于计算值的情况时对计算值进行更新，顶点计算值更新时，将计算值与到邻居顶点的出边权值之和作为消息发送

编写compute方法


```java
/* 步骤1：确定顶点标识I、顶点的计算值V、边的权值E以及消息值M的数据类型 */
public class ShortestPathComputation extends BasicComputation<LongWritable,
    DoubleWritable, FloatWritable, DoubleWritable> {

  // 源点
  protected static final int SOURCE_VERTEX = 0;
  // 表示无穷大
  protected static final Double INF = Double.MAX_VALUE;

  @Override
  public void compute(Vertex<LongWritable, DoubleWritable, FloatWritable> vertex,
      Iterable<DoubleWritable> messages) {
    /* 步骤2：编写与顶点计算、更新相关的处理逻辑以及发送消息 */
    // 超步0时将顶点初始化为表示无穷大的INF
    if (getSuperstep() == 0) {
      vertex.setValue(new DoubleWritable(INF));
    }

    // 根据接收到的消息计算当前距离源点的最短路径值
    double minDist = vertex.getId().get() == SOURCE_VERTEX ? 0d : INF;
    for (DoubleWritable message : messages) {
      minDist = Math.min(minDist, message.get());
    }

    // 当minDist小于顶点的计算值时将计算值更新为minDist
    if (minDist < vertex.getValue().get()) {
      vertex.setValue(new DoubleWritable(minDist));
      for (Edge<LongWritable, FloatWritable> edge : vertex.getEdges()) {
        double distance = minDist + edge.getValue().get();
        sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));
      }
    }

    vertex.voteToHalt();
  }
}
```

编写主方法


```
public class ShortestPathRunner extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    /* 步骤1:  设置作业的信息 */
    GiraphConfiguration giraphConf = new GiraphConfiguration(getConf());

    // 设置compute方法
    giraphConf.setComputationClass(ShortestPathComputation.class);
    // 设置图数据的输入格式
    giraphConf.setVertexInputFormatClass(JsonLongDoubleFloatDoubleVertexInputFormat.class);
    // 设置图数据的输出格式
    giraphConf.setVertexOutputFormatClass(IdWithValueTextOutputFormat.class);

    // 启用本地调试模式
    giraphConf.setLocalTestMode(true);
    // 最小和最大的Worker数量均为1，Master协调超步时所需Worker响应的百分比为100
    giraphConf.setWorkerConfiguration(1, 1, 100);
    // Master和Worker位于同一进程
    GiraphConstants.SPLIT_MASTER_WORKER.set(giraphConf, false);

    // 创建Giraph作业
    GiraphJob giraphJob = new GiraphJob(giraphConf, getClass().getSimpleName());

    // 设置图数据的输入路径
    GiraphTextInputFormat.addVertexInputPath(giraphConf, new Path(args[0]));
    // 设置图数据的输出路径
    GiraphTextOutputFormat.setOutputPath(giraphJob.getInternalJob(), new Path(args[1]));
      return giraphJob.run(true) ? 0 : -1;
    }

    public static void main(String[] args) throws Exception {
      /* 步骤2:  运行作业 */
      int exitCode = ToolRunner.run(new ShortestPathRunner(), args);
      System.exit(exitCode);
    }
  }
```

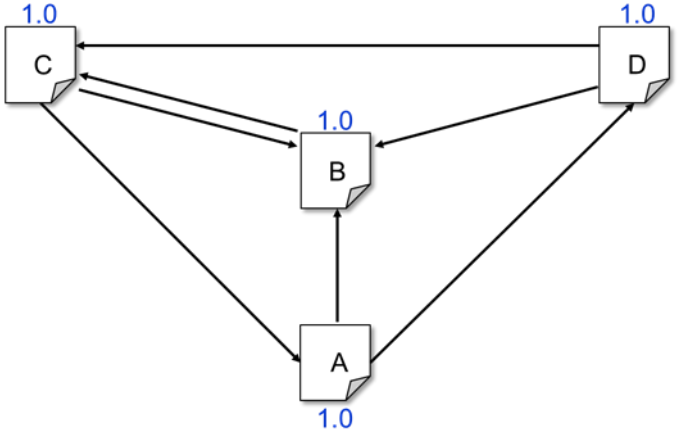
大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 连通分量
  - 单源最短路径
  - 网页链接排名
  - K均值聚类

网页链接排名

- 输入：保存在文本文件中，一行为一项网页信息
  - 网页信息：(网页名 网页排名值 (出站链接 出站链接的权重...))
- 输出：网页名及其排名值

输入	输出
A 1.0 B 1.0 D 1.0 B 1.0 C 1.0 C 1.0 A 1.0 B 1.0 D 1.0 B 1.0 C 1.0	A 0.21436 B 0.36332 C 0.40833 D 0.13027



解决方案

- 顶点计算值的更新过程
  - 超步0时，设置每个网页的排名值为1，网页p_j计算对其链向的网页的排名值贡献$PR(p_j)/L(p_j)$，并发送给链向的网页
  - 其余超步，每个网页p_i累加接收到的所有链向p_i的网页$M(p_i)$的贡献值，然后更新排名值为$0.15/N + 0.85 * \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$，并将新的贡献值发送给链向的网页

编写compute方法


```java
/* 步骤1：确定顶点标识I、顶点的计算值V、边的权值E以及消息值M的数据类型 */
public class PageRankComputation extends BasicComputation<Text, DoubleWritable,
    DoubleWritable, DoubleWritable> {

  // 阻尼系数
  private static final double D = 0.85;
  // 最大超步数
  private static final int MAX_SUPERSTEP = 20;

  @Override
  public void compute(Vertex<Text, DoubleWritable, DoubleWritable> vertex,
      Iterable<DoubleWritable> messages) {
    /* 步骤2：编写与顶点计算、更新相关的处理逻辑以及发送消息 */
    if (getSuperstep() > 0) {
      // 对接收到的贡献值进行累加
      double sum = 0;
      for (DoubleWritable message : messages) {
        sum += message.get();
      }
      //计算并更新排名值
      double rankValue = (1 - D) / getTotalNumVertices() + D * sum;
      vertex.setValue(new DoubleWritable(rankValue));
    }

    // 小于设定的最大超步数则发送消息，否则使得顶点进入非活跃状态
    if (getSuperstep() < MAX_SUPERSTEP) {
      // 存在出站链接时，各网页将网页的贡献值发送给链向的网页
      if (vertex.getNumEdges() != 0) {
        sendMessageToAllEdges(vertex, new DoubleWritable(
            vertex.getValue().get() / vertex.getNumEdges()));
      }
    } else {
      // 将当前网页的排名值四舍五入保留5位小数
      double rankValue = vertex.getValue().get();
      rankValue = Double.parseDouble(String.format("%.5f", rankValue));
      vertex.setValue(new DoubleWritable(rankValue));
      vertex.voteToHalt();
    }
  }
}
```

编写主方法


```
public class PageRankRunner extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    /* 步骤1:  设置作业的信息 */
    GiraphConfiguration giraphConf = new GiraphConfiguration(getConf());
    // 设置compute方法
    giraphConf.setComputationClass(PageRankComputation.class);
    // 设置图数据的输入格式
    giraphConf.setVertexInputFormatClass(TextDoubleDoubleAdjacencyListVertexInputFormat.class);
    // 设置图数据的输出格式
    giraphConf.setVertexOutputFormatClass(IdWithValueTextOutputFormat.class);

    // 启用本地调试模式
    giraphConf.setLocalTestMode(true);
    // 最小和最大的Worker数量均为1，Master协调超步时所需Worker响应的百分比为100
    giraphConf.setWorkerConfiguration(1, 1, 100);
    // Master和Worker位于同一进程
    GiraphConstants.SPLIT_MASTER_WORKER.set(giraphConf, false);
    // 创建Giraph作业
    GiraphJob giraphJob = new GiraphJob(giraphConf, getClass().getSimpleName());
    // 设置图数据的输入路径
    GiraphTextInputFormat.addVertexInputPath(giraphConf, new Path(args[0]));
    // 设置图数据的输出路径
    GiraphTextOutputFormat.setOutputPath(giraphJob.getInternalJob(), new Path(args[1]));

    return giraphJob.run(true) ? 0 : -1;
  }

  public static void main(String[] args) throws Exception {
    /* 步骤2:  运行作业 */
    int exitCode = ToolRunner.run(new PageRankRunner(), args);
    System.exit(exitCode);
  }
}
```

大纲

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 连通分量
  - 单源最短路径
  - 网页链接排名
  - K均值聚类

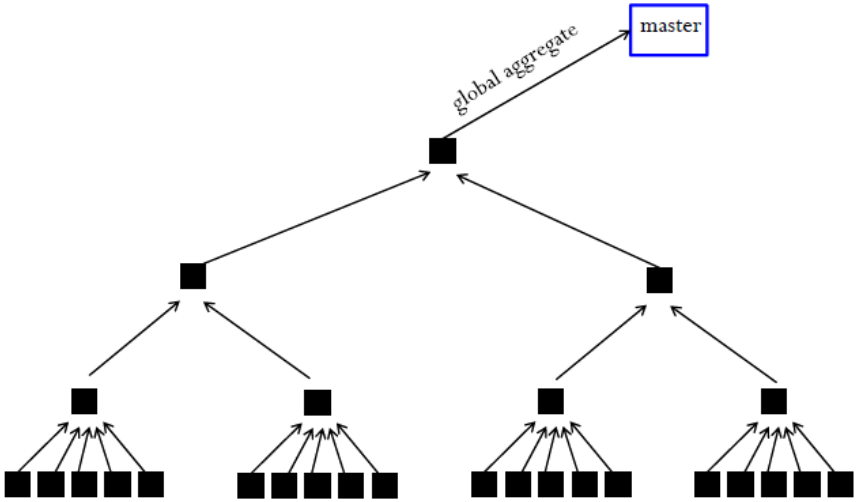
K均值聚类

- 输入：两个文本文件，分别保存数据集和聚类中心集
  - 数据集：每行为一个二维数据点及其类别标签
  - 聚类中心集：每行为一个二维数据点
- 输出：数据点及其类别标签

数据集	聚类中心集	聚类结果
0,0 -1 1,2 -1 3,1 -1 8,8 -1 9,10 -1 10,7 -1	1,2 3,1	0,0 1.0 1,2 1.0 10,7 2.0 3,1 1.0 8,8 2.0 9,10 2.0

补充：Aggregator

- 一种全局通信、监控和数据查看的机制
  - 每个顶点都可以向某个Aggregator提供数据，系统可以对这些值进行聚合操作产生一个值，之后所有顶点都可以获取这个值



解决方案

- 顶点计算值的更新过程
  - 超步计算时，顶点通过聚类中心集的Aggregator获取所有的聚类中心，计算后更新类别标签
  - 每个顶点将保存的数据点提供给所属聚类中心对应的Aggregator，对于同一Aggregator汇总的数据点，Master计算得到新的聚类中心，并替换原聚类中心

编写compute方法


```
/* 步骤1：确定顶点标识I、顶点的计算值V、边的权值E以及消息值M的数据类型 */
public class KMeansComputation extends BasicComputation<Text, DoubleWritable,
    DoubleWritable, NullWritable> {

  // 最大超步数
  private static final int MAX_SUPERSTEP = 20;
  // 聚类中心集的名称
  private static final String CENTERS = "centers";
  // 聚类中心1和2的名称前缀
  private static final String CENTER_PREFIX = "center";

  @Override
  public void compute(Vertex<Text, DoubleWritable, DoubleWritable> vertex,
      Iterable<NullWritable> iterable) {
    /* 步骤2：编写与顶点计算、更新相关的处理逻辑以及发送消息 */
    if (getSuperstep() < MAX_SUPERSTEP) {
      // 通过Aggregator获取聚类中心集并进行解析
      String centersStr = getAggregatedValue(CENTERS).toString();
      List<List<Double>> centers = PointsOperation.parse(centersStr);

      // 解析顶点中保存的数据点
      List<Double> point = new ArrayList<>();
      for (String dimension : vertex.getId().toString().split(",")) {
        point.add(Double.parseDouble(dimension));
      }
```

编写compute方法（续）


```
      // 遍历聚类中心集并计算与数据点的距离
      double minDistance = Double.MAX_VALUE;
      int centerIndex = -1;
      for (int i = 0; i < centers.size(); i++) {
        double distance = 0;
        List<Double> center = centers.get(i);
        for (int j = 0; j < center.size(); j++) {
          distance += Math.pow(point.get(j) - center.get(j), 2);
        }

        distance = Math.sqrt(distance);
        if (distance < minDistance) {
          minDistance = distance;
          centerIndex = i + 1;
        }
      }
      vertex.setValue(new DoubleWritable(centerIndex));
      // 将数据点提供给所属聚类中心的Aggregator
      aggregate(CENTER_PREFIX + centerIndex,
          new Text(StringUtils.join(",", point) + "\t"));
      } else {
        vertex.voteToHalt();
      }
    }
  }
```

使用Aggregator机制的框架


```
public class CustomMasterCompute extends DefaultMasterCompute {

  @Override
  public void initialize() throws InstantiationException, IllegalAccessException {
    /* 步骤1：注册Aggregator*/
    ......
  }

  @Override
  public void compute() {
    /* 步骤2：对汇总的数据进行处理*/
    ......
  }
}
```

编写使用Aggregator的过程


```
@Override
public void initialize() throws InstantiationException, IllegalAccessException {
  /* 步骤1：注册Aggregator*/
  // 注册聚类中心集的Aggregator
  registerAggregator(CENTERS, TextAppendAggregator.class);
  // 注册聚类中心1和2的Aggregator，用于收集属于同一个聚类中心的数据点
  for (int i = 1; i <= CENTER_SIZE; i++) {
    registerAggregator(CENTER_PREFIX + i, TextAppendAggregator.class);
  }
}
@Override
public void compute() {
  /* 步骤2：对汇总的数据进行处理*/
  StringBuilder centers = new StringBuilder();
  // 超步0时从文件中读取聚类中心集
  if (getSuperstep() == 0) {
    String centersPath = getConf().get(CENTERS);
    for (String center : PointsOperation.getCenters(centersPath)) {
      centers.append(center).append("\t");
    }
  } else if (getSuperstep() < MAX_SUPERSTEP) {
    // 依次处理聚类中心1和2的Aggregator汇总的数据点
    for (int i = 1; i <= CENTER_SIZE; i++) {
      List<Double> newCenter = new ArrayList<>();
      // 获取并解析出属于同一聚类中心的数据点
      String datas = getAggregatedValue(CENTER_PREFIX + i).toString();
      List<List<Double>> points = PointsOperation.parse(datas);
      // 计算每个维度的平均值从而得到新的聚类中心
    for (int j = 0; j < points.get(0).size(); j++) {
      double sum = 0;
      for (List<Double> point : points) {
        sum += point.get(j);
      }
      newCenter.add(sum / points.size());
    }
    centers.append(StringUtils.join(",", newCenter)).append("\t");
  }
}
  // 汇总聚类中心到Aggregator
  setAggregatedValue(CENTERS, new Text(centers.toString()));
}
```

编写主方法


```java
public class KMeansRunner extends Configured implements Tool {

  // 聚类中心集的名称
  private static final String CENTERS = "centers";

  @Override
  public int run(String[] args) throws Exception {
    /* 步骤1:  设置作业的信息 */
    GiraphConfiguration giraphConf = new GiraphConfiguration(getConf());

    // 设置compute方法
    giraphConf.setComputationClass(KMeansComputation.class);
    // 设置图数据的输入格式
    giraphConf.setVertexInputFormatClass(TextDoubleDoubleAdjacencyListVertexInputFormat.class);
    // 设置图数据的输出格式
    giraphConf.setVertexOutputFormatClass(IdWithValueTextOutputFormat.class);
    // 设置MasterCompute，启用Aggregator机制
    giraphConf.setMasterComputeClass(KMeansMasterCompute.class);
    // 设置初始聚类中心集的文件路径的配置项
    giraphConf.set(CENTERS, args[2]);

    // 启用本地调试模式
    giraphConf.setLocalTestMode(true);
    // 最小的Worker数量和最大的Worker数量均为1，Master协调超步时所需Worker响应的百分比为100
    giraphConf.setWorkerConfiguration(1, 1, 100);
    // Master和Worker位于同一进程
    GiraphConstants.SPLIT_MASTER_WORKER.set(giraphConf, false);

    // 创建Giraph作业
    GiraphJob giraphJob = new GiraphJob(giraphConf, getClass().getSimpleName());

    // 设置图数据的输入路径
    GiraphTextOutputFormat.setOutputPath(giraphJob.getInternalJob(), new Path(args[1]));

    return giraphJob.run(true) ? 0 : -1;
  }

  public static void main(String[] args) throws Exception {
    /* 步骤2:  运行作业 */
    int exitCode = ToolRunner.run(new KMeansRunner(), args);
    System.exit(exitCode);
  }
}
```

课后阅读

- 学术论文
  - Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel : A System for Large-Scale Graph Processing. In SIGMOD Conference (pp. 135–145).

课后阅读

- 企业应用
  - 阿里图计算平台GraphScope
    https://blog.csdn.net/yunqiinsight/article/details/113932857
  - 腾讯高性能图计算框架Plato
    https://page.om.qq.com/page/Ot7BiIMqQAoMLRMi9rk-PeZw0
  - 京东图计算系统 JoyGraph
    https://toutiao.io/posts/0dry2s/preview

本章小结

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例