Graph 接口

所在的包：org.jgrapht

Graph 是最基本的接口

一些其他的接口：

UndirectedGraph 和 DirectedGraph 以及 WeightedGraph 这三个接口分别继承了 Graph 接口，并拓展了相应的接口，是三个最基本的接口。

EdgeFactory 是生产边类的抽象工厂类的接口

VertexFactory 是生产顶点类的抽象工厂类的接口

ClassBasedEdgeFactory 类实现了 EdgeFactory 这个抽象工厂接口，同样 ClassBasedVertexFactory 类也实现了相应的 VertexFactory 接口。

这里说明一下 ClassBasedEdgeFactory 类的构造函数

public ClassBasedEdgeFactory(Class<? extends E> edgeClass)

我们需要传递的参数是用户自定义的边类的 Class 类。同理 ClassBasedVertexFactory 类也是一样。

AbstractGraph 类是一个抽象类，它实现了 Graph 接口。

边类的设计

在 org.jgrapht.graph 包中有两个边类，DefaultWeightedEdge 类和 DefaultEdge 类，它们是无权图和加权图边类的默认实现。DefaultEdge 类是 IntrusiveEdge 类的子类，下面是两个 IntrusiveEdge 类的两个变量，它们是代表边的两个顶点

```
Object source;
Object target;
```

DefaultWeightedEdge 类是默认的加权图中边的实现，它有一个重要的实例字段

它最重要的方法是

protected double getWeight()，注意这个方法是 protected，只供他的子类使用。

注意：因为边的权重要在某些遍历算法(如 **ClosestFirstIterator** 类，在该类的 **private double calculatePathLength(V vertex, E edge)**方法时候要使用边的权重来计算路径的权重和)时候使用，这时候需要获取边的权重，在这些算法中它们是通过 **Graph** 接口的 **public double getEdgeWeight(E e);**来获取边的权重,Graph 接口中的这个方法的具体实现在 AbstractBaseGraph 类中实现，实现如下：

```
public double getEdgeWeight(E e)
    {
        if (e instanceof DefaultWeightedEdge) {
            return ((DefaultWeightedEdge) e).weight;
        } else {
            return WeightedGraph.DEFAULT_EDGE_WEIGHT;
        }
    }
```

注意：WeightedGraph.DEFAULT_EDGE_WEIGHT=1.0；这就要求设计的有权图的边类时，应该继承 DefaultWeightedEdge。

加权图都实现了 interface WeightedGraph<V, E>这个接口，该接口一个重要的方法是 public

void setEdgeWeight(E e, double weight);，

这个方法主要在 AbstractBaseGraph<V, E>类中实现，实现如下：

```
public void setEdgeWeight(E e, double weight)
    {
        assert (e instanceof DefaultWeightedEdge) : e.getClass();
//这个要求加权图的边类需要是 DefaultWeightedEdge 类或其子类，如果边类不是DefaultWeightedEdge 类或其子类。
        ((DefaultWeightedEdge) e).weight = weight;
    }
```

我们主要使用这个方法给边赋予权重。如果没有用这个方法给边 e 赋值，则边的权重就默认是 1.0。这样自己设计的边类不需要权重字段了。

设计完边类之后，我们要添加边，Graph 接口提供了两种添加边的方法

（1） public E addEdge(V sourceVertex, V targetVertex)

（2） public boolean addEdge(V sourceVertex, V targetVertex, E e)

使用第一种方法创建边 e，我们在为边 e 赋予权重时，需要先调用 Graph 接口的 public E getEdge(V sourceVertex, V targetVertex) 方法，然后调用 public void setEdgeWeight(E e, double weight)方法设置边的权重，这个方法的实现用到了 EdgeFactory 类，它要求边类必须有默认的构造函数（即没有参数的构造函数）。

若使用第二种方法创建边 e，则可以立即使用 public void setEdgeWeight(E e, double weight) 方法来设置边的权重。

下面是一个自己定义的边类：

```
public class Edge  extends DefaultWeightedEdge{
    public Edge(){
    }
}
```

注意：我们不需要在自己定义的边类中添加边的顶点字段，如果我们想获取边的两个顶点，我们可以通过 Graph 接口的 public V getEdgeSource(E e)和 public V getEdgeTarget(E e)方法来获取边的起始顶点和结束顶点，例如 graph.getEdgeSource(e),其中 graph 是某个图类的实例。

顶点类的设计

我们需要获取某条边的顶点时，在 JGraphT 的算法中主要是通过 Graph 接口的 public V getEdgeTarget(E e); public V getEdgeSource(E e);这两个方法，它们的实现也在 AbstractBaseGraph 类中实现，举例

```
public V getEdgeSource(E e)
    {
        return TypeUtil.uncheckedCast(
            getIntrusiveEdge(e).source,
            vertexTypeDecl);
    }
```

该方法调用了 getIntrusiveEdge 方法返回了 IntrusiveEdge 类的实例，IntrusiveEdge 的实例有两个变量 source 和 target，它们就是我们自己的定义的边类，只不过它们是 object 类，需要转型为我们定义的 V 类型，通过调用 TypeUtil.uncheckedCast 方法（这个方法只有一句代码，很简单）。

AbstractBaseGraph 类

AbstractBaseGraph 类继承了 AbstractGraph 类并实现了 Graph 接口，是最为重要的一个类。其中重要的成员有：

(1) Specifics specifics　(2)Map<E, IntrusiveEdge> edgeMap

(1)第一个重要成员

Specifics 类是 AbstractBaseGraph 类的一个内部类，它包括了图的具体的实现细节，它是一个接口，它的具体实现分别在

DirectedSpecifics 类和 UndirectedSpecifics 类中。

（**1**）**DirectedSpecifics** 类

DirectedSpecifics 类最为重要的结构是 Map<V, DirectedEdgeContainer<V, E>>，其中 V 表示顶点类，DirectedEdgeContainer 类是存放边类的类。

DirectedEdgeContainer 类是相当于存放边类的容器，对于有向图来说，每个顶点都有出边和入边。因此对应于实现来说，DirectedEdgeContainer 类中有两个重要的成员

Set<EE> incoming;　Set<EE> outgoing;

incoming 存放的是顶点 V 的入边类，而 **outgoing** 存放的是顶点的出边类。此处使用存放边类的容器时 Set,因为 Set 具有重要的性质是存放的元素都是唯一的。

（**2**）**UndirectedSpecifics** 类

UndirectedSpecifics 类中最为重要的结构是 Map<V, UndirectedEdgeContainer<V, E>> ，不同的是 UndirectedEdgeContainer 类只需要一个存放边类的容器，因为在无向图中，每一个顶点所关联的边不存在方向，所以 UndirectedEdgeContainer 类中只有成员 Set<EE> vertexEdges;

(2)第二个重要成员

AbstractBaseGraph 类中第二个重要成员是 Map<E, IntrusiveEdge> edgeMap。

IntrusiveEdge 是一个 JGraphT 内部的边类，其中 DefaultEdge 类继承了该类。IntrusiveEdge 类有两个成员：Object source; Object target;分别记录一条边的开始顶点和结束顶点，即记录一条边的两个顶点。

当向图结构添加一条边 e（e 是用户自定义边类 E 的实例），JGraphT 的实现是 Specifics 执行一些操作，然后在 Map<E, IntrusiveEdge> edgeMap 中添加用户自定义的边类与 JGraphT 内部实现的 IntrusiveEdge

类的映射，其中 IntrusiveEdge 类记录了该边的两个顶点，也就是用户自定义类的实例 e 和内部 IntrusiveEdge 类的实例的映射。

Specifics 类的方法

（1） public abstract void addVertex(V vertex);
向图中添加顶点

（2） public abstract Set<V> getVertexSet();
获取图的顶点集合

（3） public abstract Set<E> getAllEdges(V sourceVertex,V targetVertex);
获取源顶点和目标顶点之间的边的集合

(4) public abstract E getEdge(V sourceVertex, V targetVertex);
获取源顶点是 sourceVertex 和目标顶点是 targetVertex 的边。

(5) public abstract void addEdgeToTouchingVertices(E e);
添加边 e 到相关的顶点

(6) public abstract int degreeOf(V vertex);
顶点 V 的度（无向图中的方法）

(7) public abstract Set<E> edgesOf(V vertex);
获取顶点 V 的边的集合

(8) public abstract int inDegreeOf(V vertex);
顶点 V 的入度（有向图的方法）

(9) public abstract Set<E> incomingEdgesOf(V vertex);
获取顶点 V 的入边的集合（有向图的方法）

(10) public abstract int outDegreeOf(V vertex);
顶点 V 的出度（有向图的方法）

(11) public abstract Set<E> outgoingEdgesOf(V vertex);
获取顶点 V 的出边的集合（有向图的方法）

(12) public abstract void removeEdgeFromTouchingVertices(E e);
从相关的顶点取出边 e （touching 相关的）

AbstractBaseGraph 类的方法：

public Set<E> getAllEdges(V sourceVertex, V targetVertex)

public E getEdge(V sourceVertex, V targetVertex)

public E addEdge(V sourceVertex, V targetVertex)

public boolean addEdge(V sourceVertex, V targetVertex, E e)

public boolean addVertex(V v)

AbstractBaseGraph 类中方法的实现都依赖于 Specifics 类，不同的 AbstractBaseGraph 类的子类都依赖于具体的 Specifics 类的子类。

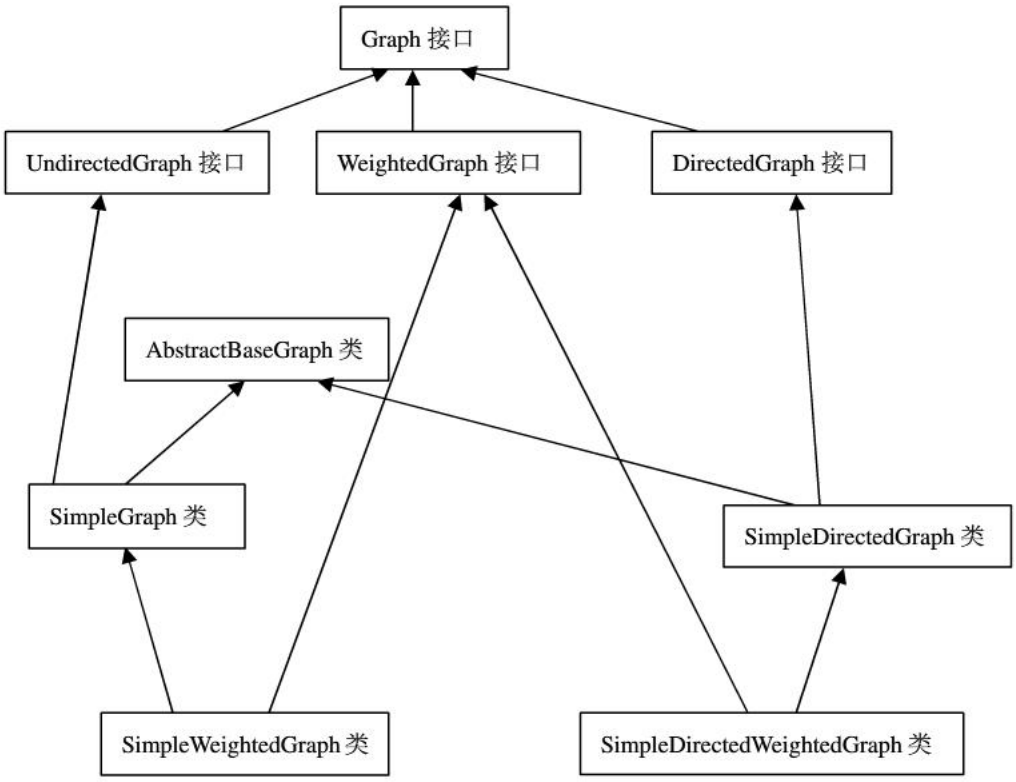
图类的继承关系图

SimpleGraph 类继承了 AbstractBaseGraph 类并实现了 UndirectedGraph 接口。（无向简单图）

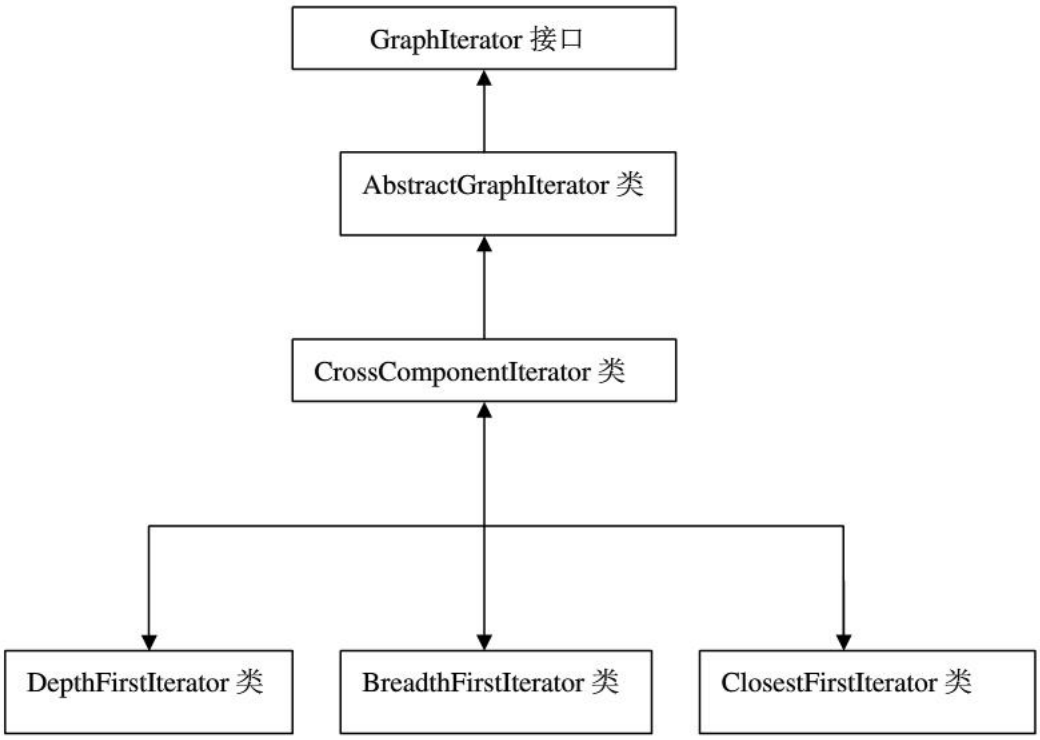
SimpleDirectedGraph 类继承了 AbstractBaseGraph 类并实现了 DirectedGraph 接口。（有向简单图）

SimpleWeightedGraph 类是简单无向加权图，

SimpleDirectedWeightedGraph 类是简单有向加权图。



遍历器类（**org.jgrapht.traverse** 包）
遍历器类的继承关系图：



（Ⅰ）**GraphIterator** 接口

(1) public boolean isCrossComponentTraversal();

//Test whether this iterator is set to traverse the grpah across connected components.

判断遍历器是否是跨连通分支遍历图

(2) public void setReuseEvents(boolean reuseEvents);

//Sets a value the <code>reuseEvents</code> flag. If the <code>

* reuseEvents</code> flag is set to <code>true</code> this class will reuse

* previously fired events and will not create a new object for each event.

* This option increases performance but should be used with care,

* especially in multithreaded environment.

设置 reuseEvents 标志的值，如果 reuseEvents 标志的值被置为真，这个类将复用先前激活的事件，将不会为每个事件创建新的对象；这个选项可以增加性能但是应该小心使用，尤其是在多线程环境下。

(3) public boolean isReuseEvents();

(4) public void addTraversalListener(TraversalListener<V, E> l);

//Adds the specified traversal listener to this iterator.

向遍历器添加具体的遍历监听器（traversal listener）

(5) public void remove();

(6) public void removeTraversalListener(TraversalListener<V, E> l);

// Removes the specified traversal listener from this iterator.

（Ⅱ）**AbstractGraphIterator** 类

AbstractGraphIterator 抽象类实现了 GraphIterator 接口。

类成员：

（1） boolean crossComponentTraversal

这个变量用来设置是否跨连通分支遍历，设置为 true 表示跨连通分支遍历。

重要方法：

（1） public void setCrossComponentTraversal(boolean crossComponentTraversal)

//用于设置是否 CrossComponentTraversal（跨连通分支遍历）

（2）public boolean isCrossComponentTraversal()

//用于判断是否是 CrossComponentTraversal

（Ⅲ）**CrossComponentIterator** 类（跨连通分支遍历）

CrossComponentIterator<V, E, D>

<V> vertex type

<E> edge type

<D> type of data associated to seen vertices

CrossComponentIterator 类继承自 AbstractGraphIterator 类，它的构造函数：

public CrossComponentIterator(Graph<V, E> g, V startVertex)

构造函数

public CrossComponentIterator(Graph<V, E> g, V startVertex)

其中参数 g 是要遍历的图，startVertex 是遍历的开始顶点。

实现代码片段:

```
if (startVertex == null) {
        // pick a start vertex if graph not empty
        if (vertexIterator.hasNext()) {
            this.startVertex = vertexIterator.next();
        } else {
            this.startVertex = null;
        }
    } else if (g.containsVertex(startVertex)) {
        this.startVertex = startVertex;
    } else {
        throw new IllegalArgumentException(
            "graph must contain the start vertex");
    }
```

它的重要成员：

（1） Graph<V, E> graph

这个成员是要遍历的图实例。

（2） Specifics<V, E> specifics

这里的 Specifics 类是 CrossComponentIterator 类的内部类，它只有一个方法如下：

public abstract Set<? extends EE> edgesOf(VV vertex);

上面的方法用来获取顶点的边集，对于无向图（即在 UndirectedSpecifics 类），该方法用来获取顶点 V 邻接的边集，而对于有向图（即在 DirectedSpecifics 类）该方法用来获取顶点 V 的出边的集合。

Specifics 类有两种实现分别是 DirectedSpecifics 类和 UndirectedSpecifics 类，这两个类都只有一个类成员 Graph<VV, EE> graph,即要遍历的图实例。

（3） **Map<V, D> seen**

用来存储遍历时访问过的顶点 V 和一些关于顶点 V 的格外的信息 D（当然 D 是可选的，如果不用，可以用 null 代替）。

（4） V startVertex

遍历访问的起始顶点

（5） Iterator<V> vertexIterator

遍历图中所有顶点的遍历器，在类的构造函数中代码如下

vertexIterator = g.vertexSet().iterator(); g 为 Graph 类的子类的实例。

类的重要方法：

（1） public boolean hasNext()

方法介绍：判断是否仍有顶点还没有被访问，如果所有顶点都已经被访问，返回 false，否则返回 true，表示仍有顶点需要访问。

方法实现代码：先调用 isConnectedComponentExhausted()判断当前连通分支的顶点是否都已经访问，如果返回 true，说明当前连通分支的所有顶点都已经访问完。如果上述方法返回 true，然后再调用 isCrossComponentTraversal()判断当前的遍历是否要跨越连通分支，如果返回 true，使用 vertexIterator（顶点集合遍历器）查找还没有访问到的顶点（没有找到返回 false），并调用 encounterVertex(v, null)方法对顶点进行相应处理（在宽度优先遍历方法中进行的处理时添加顶点到已访问的顶点集合中，并将该顶点添加到队列）。

（2）public V next()

方法介绍：获取下一个要访问的顶点。

方法解释：调用 hasNext()判断是否有连通分支可以访问，如果有连通分支可以访问，则调用 V nextVertex = provideNextVertex() 获取下一个可以访问的节点，并调用 addUnseenChildrenOf(nextVertex)添加 nextVertex 顶点的邻接顶点中没有访问的顶点。

（2） protected abstract boolean isConnectedComponentExhausted();

方法介绍：判断当前的连通分支内的顶点是否都已经访问（穷尽），如果没有访问完，则该方法返回 true，否则返回 false。

（3） private void addUnseenChildrenOf(V vertex)

方法解释：添加顶点 vertex 所有的没有访问的邻接顶点

方法代码：

调用 specifics.edgesOf(vertex)方法返回顶点 vertex 的边集，然后循环处理每一条边。对每一条边，调用 Graphs 类的 getOppositeVertex(graph, edge, vertex)获取顶点 vertex 顶点相对的顶点 oppositeV。判断顶点 oppositeV 是否已经访问过（即调用 isSeenVertex(oppositeV)方法），如果已经访问过，调用 encounterVertexAgain(oppositeV, edge) 方法，否则调用 encounterVertex(oppositeV, edge)。

（4） private void encounterStartVertex()

方法介绍：遇到起始顶点时处理起始顶点的方法。

方法代码实现:

encounterVertex(startVertex, null);

startVertex = null;

代码介绍：

encounterStartVertex()方法先调用 encounterVertex 方法对 startVertex 顶点进行处理，并标记 startVertex 为 null（相当于作标记）。

（5）protected D putSeenData(V vertex, D data)

//Stores iterator-dependent data for a vertex that has been seen.

存储与遍历器（iterator）相关的已经到达的节点的数据(存储在该类的 Map<V, D> seen 字段里)，即每一个 iterator 都有自己的单独的 seen 字段。

重要的事件方法：

①private EdgeTraversalEvent<V, E> createEdgeTraversalEvent(E edge)

②private VertexTraversalEvent<V> createVertexTraversalEvent(V vertex)

③注意：

public V next()：该方法调用的有关事件的方法

fireConnectedComponentStarted(ccStartedEvent);//有关连通分支开始遍历的事件

fireVertexTraversed(createVertexTraversalEvent(nextVertex));//有关顶点遍历的事件

public boolean hasNext()：该方法调用的有关事件的方法

fireConnectedComponentFinished(ccFinishedEvent); //有关连通分支遍历结束的事件

private VertexTraversalEvent<V> createVertexTraversalEvent(V vertex)

//创建顶点遍历事件

private EdgeTraversalEvent<V, E> createEdgeTraversalEvent(E edge)//创建边遍历事件

注意：在 **CrossComponentIterator<V, E, D>**类中，在创建事件对象，即调用对象的构造函数时，它初始化 **eventSource**（即事件源）为 **this**（即当前的 **iterator** 实例）。

（Ⅳ）**BreadthFirstIterator** 类

BreadthFirstIterator 类继承自 CrossComponentIterator 类。

类成员：Deque<V> queue

宽度优先遍历算法要使用队列，这个成员就是算法中使用的队列。

类方法：

(1) protected boolean isConnectedComponentExhausted()

方法的实现代码：

queue.isEmpty();

代码解释： 判断队列是否为空。

(2) protected void encounterVertex(V vertex, E edge)

方法介绍：遇到顶点 vertex 时（一般是第一次），处理顶点 vertex 的方法（encounter 英语的意思是遇到）

方法的实现代码

putSeenData(vertex, null);

queue.add(vertex);

代码解释：添加顶点 vertex 到已经访问的 seen,并将该顶点添加到队列中。

（3）protected void encounterVertexAgain(V vertex, E edge)

方法介绍：再一次遇到顶点 vertex 时进行处理的方法。

方法的实现代码：此处方法的实现体为空。

代码解释：在宽度优先遍历时遇到已经访问过的顶点不做任何处理，即实现体为空。

（5） protected V provideNextVertex()

方法的实现代码：

queue.removeFirst();

代码解释：从队列中取出下一个要访问的顶点。

（Ⅴ）**ClosestFirstIterator** 类

重要字段：

①private FibonacciHeap<QueueEntry<V, E>> heap

// Priority queue of fringe vertices.，用于选择下一个节点

②private double radius = Double.POSITIVE_INFINITY; //搜索到达的最大距离

③静态内部类 QueueEntry<V, E>

字段介绍：

E spanningTreeEdge;

//Best spanning tree edge to vertex seen so far. 到达该顶点的目前为止最好的生成树的边

V vertex; // The vertex reached. 到达的顶点

boolean frozen; // True once spanningTreeEdge is guaranteed to be the true minimum.

一旦 spanningTreeEdge（生成树边）确保是最小的，frozen 置为 true

构造函数

public ClosestFirstIterator(Graph<V, E> g)

//默认设置 startVertex=null radius=Double.POSITIVE_INFINITY

public ClosestFirstIterator(Graph<V, E> g, V startVertex)

// 默认设置 radius=Double.POSITIVE_INFINITY

public ClosestFirstIterator(Graph<V, E> g, V startVertex, double radius)

重要方法

① private void checkRadiusTraversal(boolean crossComponentTraversal)

代码如下：

```
{
        if (crossComponentTraversal && (radius != Double.POSITIVE_INFINITY)) {
            throw new IllegalArgumentException(
                "radius may not be specified for cross-component traversal");
        }
    }
```

② protected boolean isConnectedComponentExhausted()

//判断连通分支是否遍历完毕

③ private FibonacciHeapNode<QueueEntry<V, E>> createSeenData

// The first time we see a vertex, make up a new heap node for it

如果我们第一次到达该顶点，我们为该顶点创建一个堆节点（其他方法将该节点插入堆）

④ protected void encounterVertex(V vertex, E edge)

//第一次到达顶点 vertex

⑤ protected void encounterVertexAgain(V vertex, E edge)

//再次见到该顶点

⑥ public double getShortestPathLength(V vertex)

// Get the length of the shortest path known to the given vertex. If thevertex has already been visited, then it is truly the shortest path length; otherwise, it is the best known upper bound.

获取到达从 startVertex 顶点到给定顶点 vertex 的最短路径长度，如果顶点已经到达了，那么返回的距离就是最短路径长度，否则它是目前已知的最好的上界。

Dijkstra 单源最短路径

所在的包：org.jgrapht.alg 包

介绍:用于求 startVertex 和 endVertex 之间的最短路径

重要字段

private GraphPath<V, E> path;

构造函数：

（1）public DijkstraShortestPath(Graph<V, E> graph,V startVertex,V endVertex)

（2）public DijkstraShortestPath(Graph<V, E> graph,V startVertex,

V endVertex, double radius)

GraphPath 接口（**org.jgrapht.***包）

（1） public Graph<V, E> getGraph();

获取包含的图。

（2） public V getStartVertex();

返回路径的起始顶点

（3） public V getEndVertex();

返回路径的结束顶点

（4） public List<E> getEdgeList();

返回路径所经过的边类列表。

（5）public double getWeight();

获取路径的总权重长度。

GraphPathImpl 类（实现 **GraphPath** 接口）

类的实例变量：

private Graph<V, E> graph;

private List<E> edgeList;

private V startVertex;

private V endVertex;

private double weight;

ClosestFirstIterator 类（**org.jgrapht.traverse** 类）

ClosestFirstIterator 类继承自 CrossComponentIterator 类。

ClosestFirstIterator<V, E>

（1） 构造函数

该类共有三个构造函数:

(a) public ClosestFirstIterator(Graph<V, E> g)

(b) public ClosestFirstIterator(Graph<V, E> g, V startVertex)

(c) public ClosestFirstIterator(Graph<V, E> g, V startVertex, double radius)

（2） 实例变量

private FibonacciHeap<QueueEntry<V, E>> heap;

（3） 方法

ClosestFirstIterator 类重载了 CrossComponentIterator 类的几个方法，如下

protected boolean isConnectedComponentExhausted();

protected void encounterVertex(V vertex, E edge)

protected void encounterVertexAgain(V vertex, E edge)

protected V provideNextVertex()

Fibonacci 堆（**org.jgrapht.util** 包的 **FibonacciHeap** 类和 **FibonacciHeapNode** 类）

FibonacciHeapNode<T>类

（**1**）实例变量

T data　　节点数据

FibonacciHeapNode<T> child;　第一个孩子节点

FibonacciHeapNode<T> left;　　左兄弟节点

FibonacciHeapNode<T> parent;　父节点

FibonacciHeapNode<T> right;　右兄弟节点

boolean mark;　　从该节点添加到它的父节点，如果该节点有孩子节点被去除，该值为 true

double key;　　节点的 key 值

int degree;　　该节点孩子节点的数目

（**2**）　构造函数

public FibonacciHeapNode(T data, double key)

GraphPath 接口

（1）　public Graph<V, E> getGraph(); //获取路径所在的图

（2）　public V getStartVertex();　//获取路径的起始节点

（3）　public V getEndVertex();　//获取路径的结束节点

（4）　public List<E> getEdgeList(); //获取路径所有的边

（5）　public double getWeight(); //获取路径的权重之和

GraphPathImpl 类（实现 GraphPath 接口）

所在的包: org.jgrapht.graph 包

实例字段：

```
private Graph<V, E> graph;
private List<E> edgeList;
private V startVertex;
private V endVertex;
private double weight;
```

GraphDelegator 类

介绍：GraphDelegatror 是图，它把所有的操作都委托给它的 backing graph（构造函数内部指定的图）

所在的包: org.jgrapht.graph 包

重要字段

private Graph<V, E> delegate;

// backing graph(delegate)，所有的操作都在 delegate（代理图）上执行，delegate 可以看作是最底层的图，所有的对 GraphDelegater 类实例的操作最终都将在底层的 delegate 图上执行。

构造函数：

public GraphDelegator(Graph<V, E> g)

注意：param g the backing graph (the delegate).
对 GraphDelegator 类实例（即图实例）的修改也影响 delegate 图
重要方法举例：
（1）public Set<E> getAllEdges(V sourceVertex, V targetVertex)
//获取 delegate 图的所有边
（2）public E addEdge(V sourceVertex, V targetVertex)
//在 delegate 图上添加边

EdgeReversedGraph 类
//GraphDelegate 图的子类
介绍：
This class allows you to use a directed graph algorithm in reverse. For
* example, suppose you have a directed graph representing a tree, with edges
* from parent to child, and you want to find all of the parents of a node. To
* do this, simply create an edge-reversed graph and pass that as input to
* {@link org.jgrapht.traverse.DepthFirstIterator}.

JGraphT 事件
JGraphT 事件的所在的包：org.jgrapht.event
包的介绍：
Event classes and listener interfaces, used to provide a change notification mechanism on graph modification events.
事件相关
(1) class EdgeTraversalEvent<V, E>类
解释: A traversal event for a graph edge. 图边的遍历事件
重要字段
protected E edge; // The traversed edge. 遍历的边
重要方法
public E getEdge() // Returns the traversed edge. 获取遍历的边
(2) class VertexTraversalEvent<V>类
重要字段
protected V vertex; // * The traversed vertex.
重要方法
public V getVertex() // Returns the traversed vertex.

(3) class ConnectedComponentTraversalEvent 类

重要字段

private int type; // The type of this event. 事件的类型：主要有两种

CONNECTED_COMPONENT_STARTED

CONNECTED_COMPONENT_FINISHED

构造函数:

public ConnectedComponentTraversalEvent(Object eventSource, int type)

//注意 eventSource 是事件源（the source of the event.）

重要方法

public int getType() // Returns the event type.

(4) class GraphChangeEvent 类

解释：An event which indicates that a graph has changed. This class is a root for graph change events.

指示图已经发生改变的事件，这个类是图改变事件的根类。

构造函数：

public GraphChangeEvent(Object eventSource, int type)

(5) class **GraphVertexChangeEvent<V>**类

class **GraphEdgeChangeEvent<V,E>**类

注意：这两个类是 GraphChangeEvent 类的子类

监听者相关

(5) interface VertexSetListener<V>

解释：

A listener that is notified when the graph's vertex set changes. It should be used when *only* notifications on vertex-set changes are of interest. If all graph notifications are of interest better use GraphListener.

当图的顶点集发生改变时，被通知的监听者。如果仅对顶点集改变的通知感兴趣，它可以被使用。如果对所有图改变的通知感兴趣，使用GraphListener更好

①public void vertexAdded(GraphVertexChangeEvent<V> e);

// Notifies that a vertex has been added to the graph.

②public void vertexRemoved(GraphVertexChangeEvent<V> e);

// Notifies that a vertex has been removed from the graph.

(6) interface GraphListener<V, E>

注意**:**这个接口扩展了 VertexSetListener<V>接口

解释：listener that is notified when the graph changes. If only notifications on vertex set changes are required it is more efficient to use the VertexSetListener

当图发生改变时，被通知的监听者。如果仅仅需要发生在顶点集改变的通知，使用 VertexSetListener 更有效

①public void edgeAdded(GraphEdgeChangeEvent<V, E> e);

//Notifies that an edge has been added to the graph.

通知一条边已经添加到图中。

②public void edgeRemoved(GraphEdgeChangeEvent<V, E> e);

//Notifies that an edge has been removed from the graph.

(7) interface TraversalListener<V, E>

重要方法介绍

①public void connectedComponentFinished(ConnectedComponentTraversalEvent e);

// Called to inform listeners that the traversal of the current connected

* component has finished.

用来通知监听者当前连通分支的遍历已经结束

②public void connectedComponentStarted(ConnectedComponentTraversalEvent e);

// // Called to inform listeners that a traversal of a new connected component

* has started.

用来通知监听者一个新的连通分支的遍历开始

③public void edgeTraversed(EdgeTraversalEvent<V, E> e);

// Called to inform the listener that the specified edge have been visited

* during the graph traversal. Depending on the traversal algorithm, edge

* might be visited more than once.

用来在图遍历的时候通知监听者具体的边已经被访问,取决于具体的遍历算法，一条边可以被访问多次。

④ public void vertexTraversed(VertexTraversalEvent<V> e);

/ / Called to inform the listener that the specified vertex have been visited

* during the graph traversal. Depending on the traversal algorithm, vertex

* might be visited more than once.

用来在图遍历的时候通知监听者具体的顶点已经被访问,取决于具体的遍历算法，一个顶点可以被访问多次。

⑤ public void vertexFinished(VertexTraversalEvent<V> e);

//Called to inform the listener that the specified vertex have been

* finished during the graph traversal. Exact meaning of "finish" is

* algorithm-dependent; e.g. for DFS, it means that all vertices reachable

* via the vertex have been visited as well.

用来在图遍历的时候通知监听者具体的顶点已经结束,准确的结束的定义取决于具体的遍历算法， 例如 DFS，”结束”的定义是所有的顶点都已经被访问过。