# Storm的数据模型

storm应用处理的基本数据单元叫做**tuple（元组）。**每个元组包含了一个预定义的fields列表。每个field的数据类型可以是byte,char,integer,long,float,double,

boolean,byte[]。storm同时提供了API定义自己的数据类型，用于在tuple中进行序列化。tuple是动态适应的，意味着你只需要定义fields的名称而不是数据类型，由于storm的一个进程单元可以持有多种数据类型，所以定义field的数据类型也不现实。

# Storm中的topology定义

在storm的术语中，topology抽象定义了计算图。你可以创建一个storm topology并部署在storm集群上运行。一个topology可以通过有向无环图表示，每个节点都运行一部分任务然后转发到下一个节点。



下列是storm topology中的组件：

* **Stream**：一个流指的是一组无边界的tuple序列，可以被storm并行的操作。每个流可以被一种或多种类型的bolts处理。在storm应用中每个流都有一个ID，bolt可以根据流ID从这些流中产生和消费元组。
* **Spout：**一个spout就是topology多种tuples的源头。它负责从外部数据源读取或监听数据**。org.apache.storm.topology.IrichSpout**用于定义spouts，在java中编写spout类需要继承IrichSpout接口。任何时候一个spout分发tuple，storm会处理这个tuple，并且追踪所有的tuple是生成情况，当源头中的所有tuple执行（发送）完毕，storm会返回一个ack给spout。这个追踪仅当发送tuple时提供messageId才发生。我们可以设定tuple处理超时时间，也需要提供messageId。

**spout中的重要方法：**

* nextTuple():storm通过调用此方法从输入源中获取下一个元组。在此方法中，可以定义读取数据逻辑，并将他们发送到SpoutOutputCollector实例中。如果一个spout希望发送数据到不止数据流中，可以使用declareStream()，并且指定streamID。
* ack(Object msgId):当tuple携带着message ID，并且完成了处理后会调用。
* open()：这个方法仅会调用一次，仅当spout初始化时调用。如果需要连接外部数据源输入数据，在open方法中定义连接的逻辑，然后再nextTuple方法定义读取和分发逻辑。
* **Bolt**：bolt是topology中处理tuple的powerhouse，负责转换流。每个bolt需要对tuple是进行一个简单的转换，多个bolts需要相互协调完成一个复杂的转换。org.apache.storm.topology.IrichBolt是Java中定义Bolt的接口。一个bolt可以订阅多个流中的组件--spouts或者bolts，也可以发送输出到多个流中。输出流应该使用OutputFieldsDeclarer.declare()方法定义输出的fields名称格式。

**bolt的重要方法：**

* execute(Tuple tuple):每个tuple从订阅的输入流中来到时执行此方法。在这个方法中，可以对tuple进行任何需要的处理，然后，可以按照一定的输出格式进行分发，发送到定义好的流中，或者存储到数据库中等等操作。

你不需要在此方法执行时立刻处理tuple，可以需要时再作处理。

* prepare(Map stormConf,Topologycontext,context,OutputCollector collector):一个bolt可以被多个workers执行。一个bolt实例在客户端机器上创建，然后序列化并提交到nimbus，当nimbus为topology创建worker实例后，它发送序列化的bolt到workers，worker会进行反序列化，调用prepare方法。

# Worker Process

一个storm topology在storm集群的多个节点上执行。每个节点运行一个或多个叫做**worker processes**的**JVMs**，它们负责处理**topology**的一部分（一个topo的子集）。一个Storm cluster同时可以处理多个topologies。一个worker process一定要与这些topologies中的一个进行关联，并且执行与之关联的topology中的多个组件。如果同时有多个topologies运行，**它们之间时不会共享workers的**，因此相互之间是隔离的。一个supervisor节点最多启动4个worker进程。默认情况下，一个worker对应一个acker

# The executor

在每个workder process中，会存在多个**进程**执行topology的各部分。每个线程（Thread）叫做一个**executor**，**每个executor只能执行一个topology中相同的组件(spout/bolt)的多个task**。**storm会为每个acker单独开启一个excutor中的一个task用于确保tuple的正确接收。**

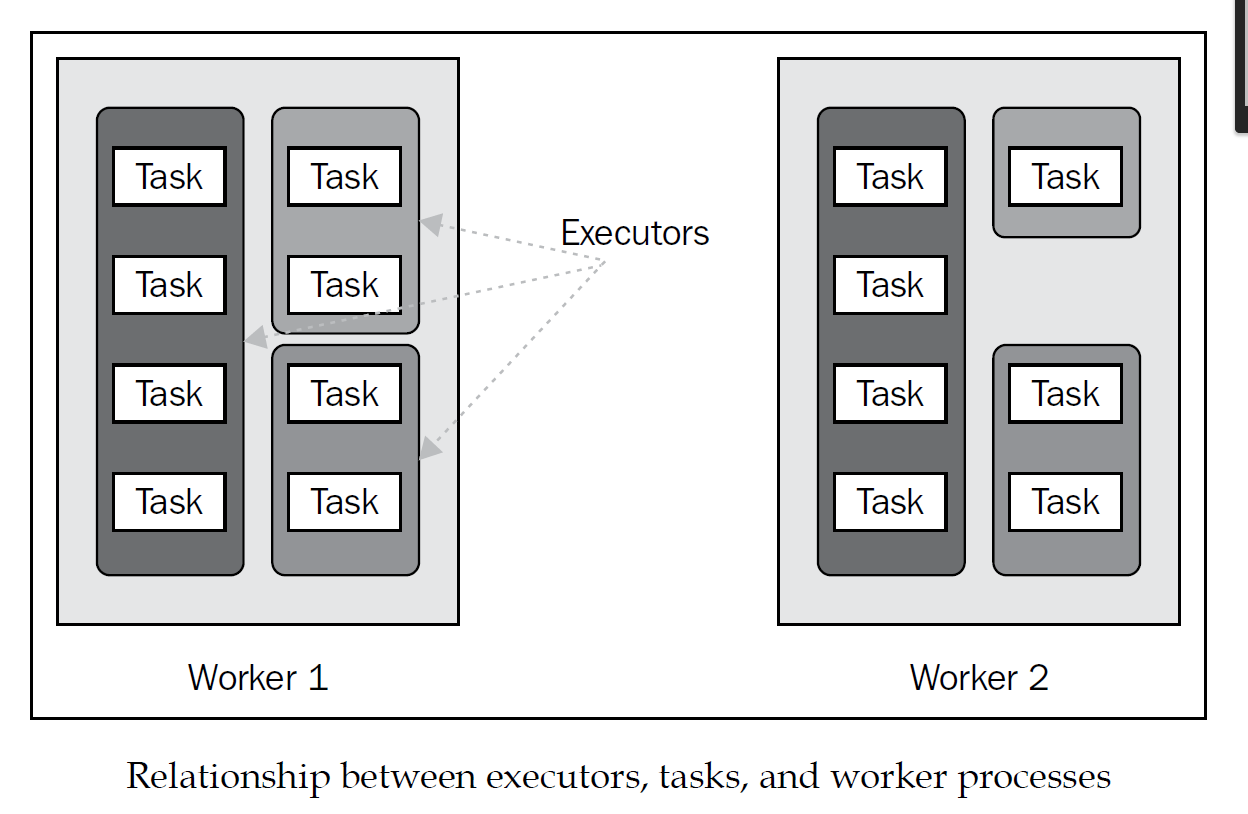
每个executor，作为一个独立线程，仅能有序的执行被分发的任务。topology运行时，executors的数量可以动态地改变。意味着可以容易的控制topology中组件的并发度(**degree of parallelism**)。

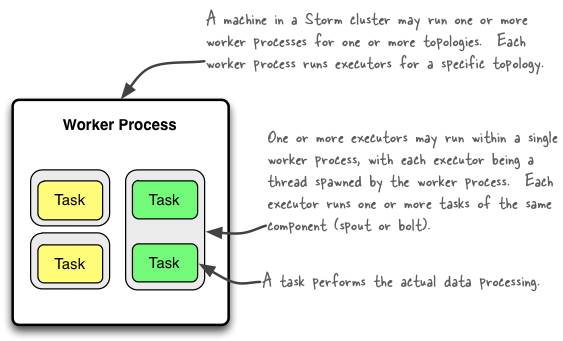
# Tasks

**task**是storm中最小的任务执行单元。**每个task是一个spout或者bolt实例**。当定义一个Storm Tolopogy，你可以设置具体每个spout和bolt的具体任务数。一旦定义了在运行期间就不可以改变，每个task可以独立运行或者与同类型的的其他task同时运行。

下图展示了**worker process , executors , tasks**的关系。

有两个workers，每个worker都开启了3个进程(executor)。worker1中executors分别运行了4、2、2个task（也就是spout/bolt实例）。worker2中executors分别运行了4、1、2个task（也就是spout/bolt实例）。整个topology的任务数是15，因此并发度(degree of parallelism)是15.





从代码级别上配置并发度

**1.设置worker processes个数。**

Config conf = new Config();

conf.setNumWorkers(3);

storm会自动分配worker在每个supervisor中的个数，应该平均分配到每个节点上。

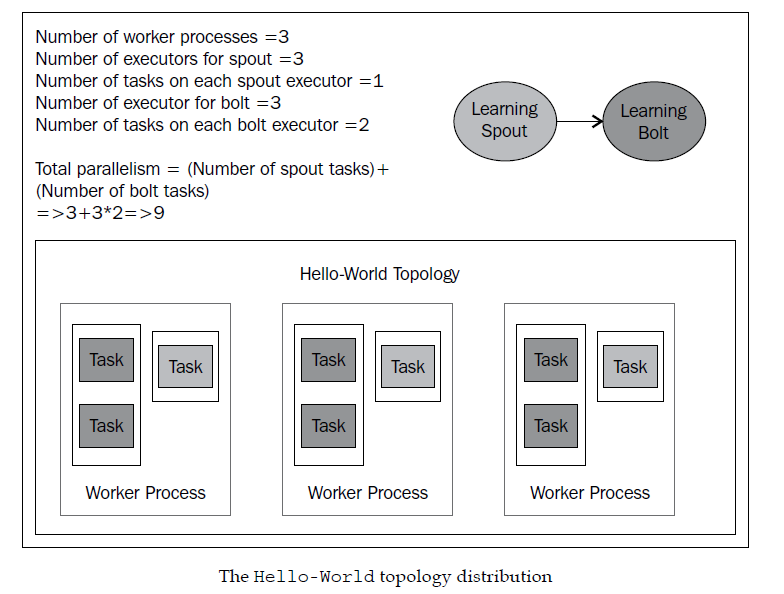
**2.设置executors的个数。**

通过setSpout(args,args,parallelism\_ hint) or setBolt(args,args,parallelism\_hint)方法的**parallelism\_hint**参数设定

**3.设置task个数**

builder.setSpout("LearningStormSpout", new LearningStormSpout(), 2).setNumTasks(4);

# 分发worker processes,executors,tasks举例



总并发度 = spout tasks数目 + bolt tasks数目。

如果总的并发度不是worker的倍数，storm会尽量均匀的分发tasks。

***默认情况下：***

一个supervisor节点最多启动4个worker进程

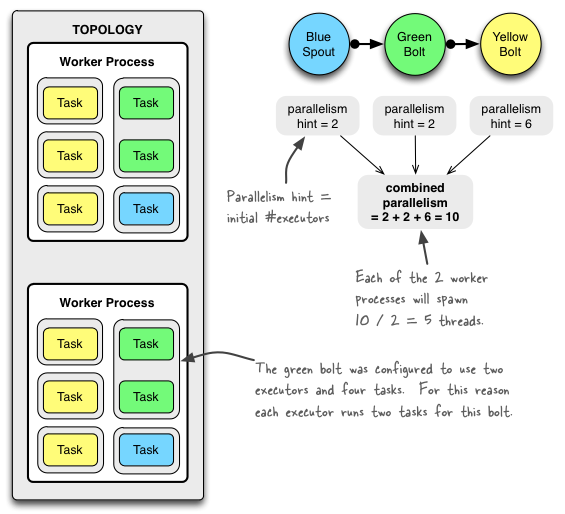
一个topology默认占用一个worker进程

一个worker进程对应一个acker，一个acker对应一个executor中的一个task(这个task独立占有一个executor)

一个spout/bolt对应一个executor

一个executor启动一个task

例子2



The GreenBolt was configured as per the code snippet above whereas BlueSpout and YellowBolt only set the parallelism hint (number of executors). Here is the relevant code:

Config conf **=** **new** Config**();**

conf**.**setNumWorkers**(**2**);** *// use two worker processes*

topologyBuilder**.**setSpout**(**"blue-spout"**,** **new** BlueSpout**(),** 2**);** *// set parallelism hint to 2*

topologyBuilder**.**setBolt**(**"green-bolt"**,** **new** GreenBolt**(),** 2**)**

**.**setNumTasks**(**4**)**

**.**shuffleGrouping**(**"blue-spout"**);**

topologyBuilder**.**setBolt**(**"yellow-bolt"**,** **new** YellowBolt**(),** 6**)**

**.**shuffleGrouping**(**"green-bolt"**);**

StormSubmitter**.**submitTopology**(**

"mytopology"**,**

conf**,**

topologyBuilder**.**createTopology**()**

**);**

And of course Storm comes with additional configuration settings to control the parallelism of a topology, including:

* [TOPOLOGY\_MAX\_TASK\_PARALLELISM](http://storm.apache.org/releases/1.2.2/javadocs/org/apache/storm/Config.html#TOPOLOGY_MAX_TASK_PARALLELISM): This setting puts a ceiling on the number of executors that can be spawned for a single component. It is typically used during testing to limit the number of threads spawned when running a topology in local mode. You can set this option via e.g. [Config#setMaxTaskParallelism()](http://storm.apache.org/releases/1.2.2/javadocs/org/apache/storm/Config.html#setMaxTaskParallelism(int)).

## How to change the parallelism of a running topology

A nifty feature of Storm is that you can increase or decrease the number of worker processes and/or executors without being required to restart the cluster or the topology. The act of doing so is called rebalancing.

You have two options to rebalance a topology:

1. Use the Storm web UI to rebalance the topology.
2. Use the CLI tool storm rebalance as described below.

Here is an example of using the CLI tool:

## Reconfigure the topology "mytopology" to use 5 worker processes,

## the spout "blue-spout" to use 3 executors and

## the bolt "yellow-bolt" to use 10 executors.

$ storm rebalance mytopology -n 5 -e blue-spout=3 -e yellow-bolt=10

# 流分组

1. **Shuffle grouping**: Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples.
2. **Fields grouping**: The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, but tuples with different "user-id"'s may go to different tasks.
3. **Partial Key grouping**: The stream is partitioned by the fields specified in the grouping, like the Fields grouping, but are load balanced between two downstream bolts, which provides better utilization of resources when the incoming data is skewed. [This paper](https://melmeric.files.wordpress.com/2014/11/the-power-of-both-choices-practical-load-balancing-for-distributed-stream-processing-engines.pdf) provides a good explanation of how it works and the advantages it provides.
4. **All grouping**: The stream is replicated across all the bolt's tasks. Use this grouping with care.
5. **Global grouping**: The entire stream goes to a single one of the bolt's tasks. Specifically, it goes to the task with the lowest id.
6. **None grouping**: This grouping specifies that you don't care how the stream is grouped. Currently, none groupings are equivalent to shuffle groupings. Eventually though, Storm will push down bolts with none groupings to execute in the same thread as the bolt or spout they subscribe from (when possible).
7. **Direct grouping**: This is a special kind of grouping. A stream grouped this way means that the **producer** of the tuple decides which task of the consumer will receive this tuple. Direct groupings can only be declared on streams that have been declared as direct streams. Tuples emitted to a direct stream must be emitted using one of the [emitDirect](javadocs/org/apache/storm/task/OutputCollector.html#emitDirect(int, int, java.util.List) methods. A bolt can get the task ids of its consumers by either using the provided [TopologyContext](http://storm.apache.org/releases/1.2.2/javadocs/org/apache/storm/task/TopologyContext.html) or by keeping track of the output of the emit method in [OutputCollector](http://storm.apache.org/releases/1.2.2/javadocs/org/apache/storm/task/OutputCollector.html) (which returns the task ids that the tuple was sent to).
8. **Local or shuffle grouping**: If the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise, this acts like a normal shuffle grouping.

## shuffle grouping

shuffle grouping使用统一的、随机的方法分发tuples给每个任务。每个task会处理相同个数的tuple，这种分组方法对于没有数据驱动的分区是非常理想的。

## Fields grouping

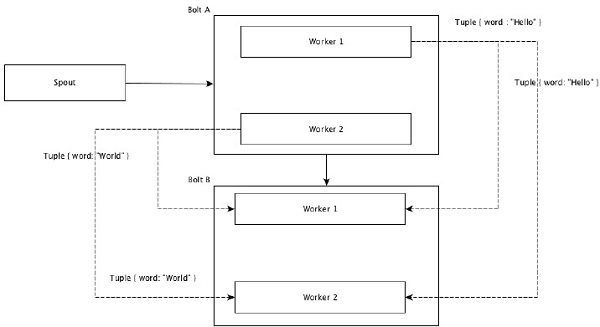
这是一种基于tuples的field的分组。给定一个field，所有具有相同的field值得tuple都会被分配到相同的task中。

builder.setBolt("2", new TweetCounter()).fieldsGrouping("1", new Fields("username"))

fields grouping使用以下函数进行计算：hash (fields) % (no. of tasks)

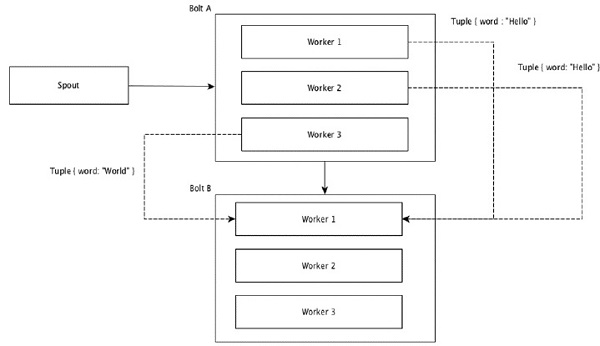
## All grouping

这个分组不进行分区，而是把所有的tuple复制到所有的task中，即每个task都拥有相同的tuple。



## Global grouping

所有的流会被分组和转发到同一个bolt实例（task）中，特别地，发送到id最小的worker中。



## Direct grouping

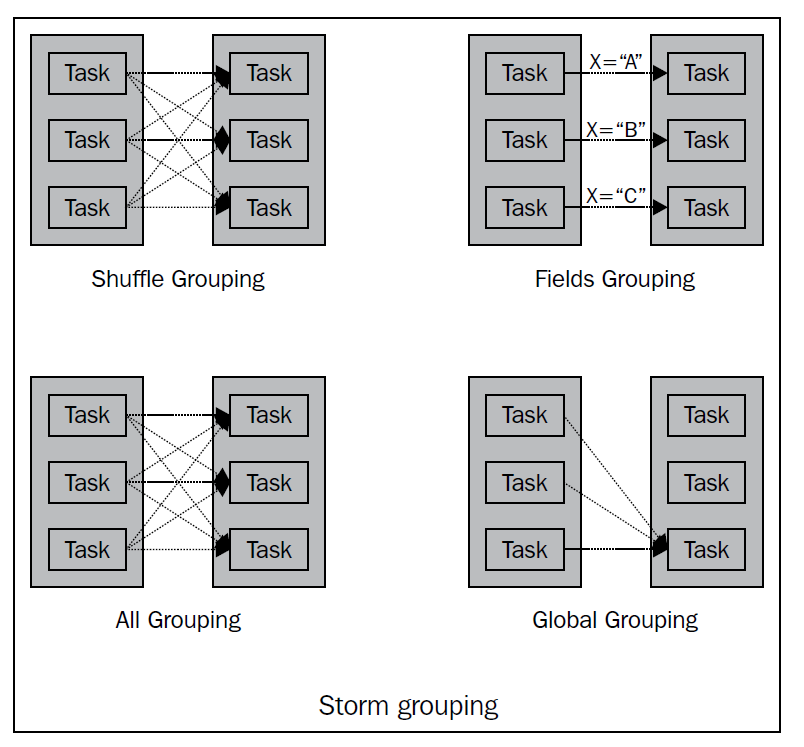
发送给一个指定的bolt，需要在分发时使用emitDirect()方法，并指定taskId。

## None grouping

开发者无需关心分组情况，storm自动完成。

## Custom grouping

自定义的grouping，需要实现org.apache.storm.grouping.CustomStreamGrouping接口



# 容错

**What happens when a worker dies?**

当一个worker停止了，supervisor会重启，如果持续重启失败并无法和nimbus进行心跳连接，numbus会重新调度该worker。

**What happens when a node dies?**

该节点上tasks将超时，然后nimbus会重新分配tasks到其他节点。

**What happens when** **Nimbus or Supervisor daemons die?**

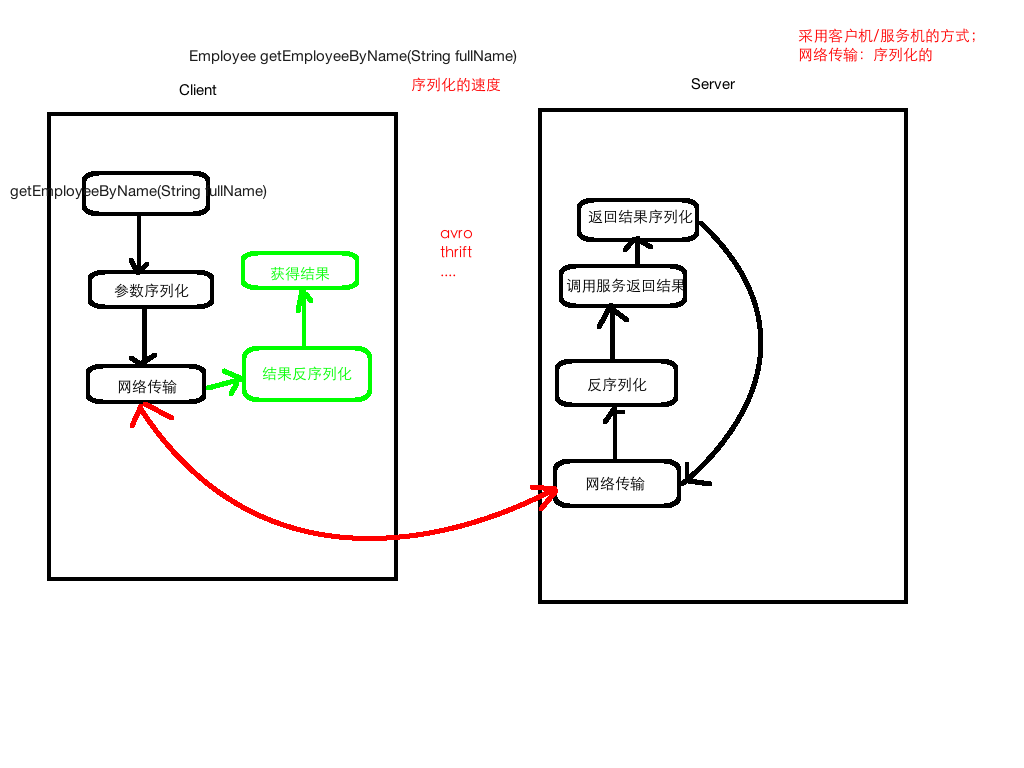
Nimbus和Supervisor都可以进行fail-fast(面临异常情况下的自销毁)，并且是无状态的（所有的状态保存在zk或者磁盘上）Nimbus和Supervisor必须运行在工具的监测下，像daemontools or monit。所以如果**Nimbus or Supervisor daemons**停止了，它们会像没有任何事发生一样重启。

**Is Nimbus a single point of failure?**

如果nimbus停止了，workers仍然会运行。如果workers停止了supervisor会重新启动它们。但是没有nimbus，workers不会重新分配的其他节点（必要情况下，就像你丢失了一个worker机器）。从1.0.0后nimbus支持HA。

# 分布式RPC

**什么是RPC？**

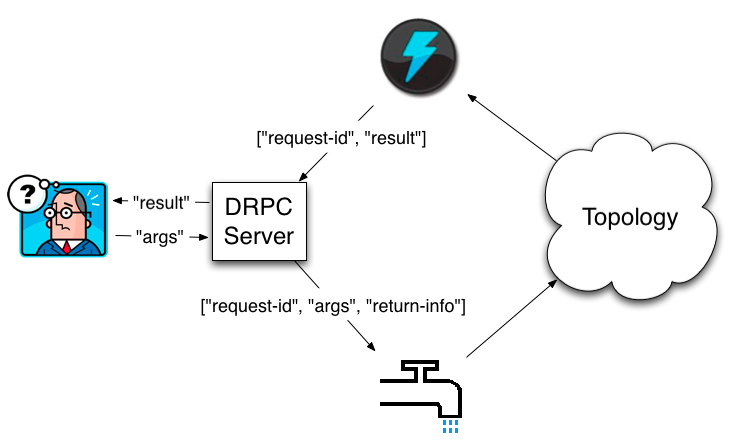


**Storm中的Dritribute RPC（DRPC）**

<http://storm.apache.org/releases/1.2.2/Distributed-RPC.html>

DRPC的目标是利用storm进行并行计算。

DRPC Server接收RPC请求，发送到Storm topology，然后从topology中获取结果，返回到等待的客户端。从客户端的角度看，DRPC和普通的RPC没有区别。



客户端发送方法名和参数到DRPC Server中执行，实现了该方法的topology使用DRPCSpout从DPRC Server中接收调用方法的流。每个调用的方法都有一个由DRPC Server标记的唯一的ID，然后topology计算结果，并且在最后一个叫bolt中调用ReturnResults类连接DPRC Server,把结果传递给它，DPRC Server使用id匹配客户端等待的结果。

# 与JDBC整合

<http://storm.apache.org/releases/1.2.2/storm-jdbc.html>

## 插入数据库

### **ConnectionProvider**

不同的连接池需要实现此接口，storm提供了实现类

 org.apache.storm.jdbc.common.HikariCPConnectionProvider，通过此类设置连接池、用户名、密码、连接地址等等。

### JdbcMapper

**public** **interface** **JdbcMapper** **extends** Serializable **{**

List**<**Column**>** **getColumns(**ITuple tuple**);**

**}**

此方法定义了一个tuple如何插入一行到数据库中。如果没有重写此方法，那么从上一级发送过来的tuple格式和对应名称应该与对应的sql语句的字段名称相同，否则应该重写此方法进行手动映射。此方法返回的List<Column>的元素顺序很重要，它的顺序就是sql语句中需要插入的参数的顺序。

SimpleJdbcMapper 是JdbcMapper的实现类。

在构造SimpleJdbcMapper时，可以显式指定需要插入的列，这个字段需要跟上一级spout或bolt所declarer的字段名称相同。

List**<**Column**>** columnSchema **=** Lists**.**newArrayList**(**

**new** **Column(**"user\_id"**,** java**.**sql**.**Types**.**INTEGER**),**

**new** **Column(**"user\_name"**,** java**.**sql**.**Types**.**VARCHAR**));**

JdbcMapper simpleJdbcMapper **=** **new** SimpleJdbcMapper**(**columnSchema**);**

### JdbcInsertBolt

JdbcInsertBolt userPersistanceBolt **=**

**new** JdbcInsertBolt**(**connectionProvider**,** simpleJdbcMapper**)**

**.**withInsertQuery**(**"insert into user values (?,?)"**)**

**.**withQueryTimeoutSecs**(**30**);**



## 读取数据库

通过select查询数据库，然后发送到topology中。

### SimpleJdbcLookupMapper

Fields outputFields **=** **new** Fields**(**"user\_id"**,** "user\_name"**,** "create\_date"**);**

List**<**Column**>** queryParamColumns **=** Lists**.**newArrayList**(new** Column**(**"user\_id"**,** Types**.**INTEGER**));**

**this.**jdbcLookupMapper **=** **new** SimpleJdbcLookupMapper**(**outputFields**,** queryParamColumns**);**

outputFields定义了输出格式，queryParamColumns定义了作为查询条件的列。

### JdbcLookupBolt

String selectSql **=** "select user\_name from user\_details where user\_id = ?"**;**

SimpleJdbcLookupMapper lookupMapper **=** **new** SimpleJdbcLookupMapper**(**outputFields**,** queryParamColumns**)**

JdbcLookupBolt userNameLookupBolt **=** **new** JdbcLookupBolt**(**connectionProvider**,** selectSql**,** lookupMapper**).**withQueryTimeoutSecs**(**30**);**

JdbcLookupBolt执行了查询操作，并输出查询结果到下一级bolt。

# 与Kafka整合

<http://storm.apache.org/releases/1.2.2/storm-kafka-client.html>

使用新式的API --- maven: storm-kafka-client

## Reading From kafka (Spouts)

作为kafka的消费者：**KafkaSpout**

## Writing to Kafka as part of your topology

作为kafka的生产者：**KafkaBolt**

# 与HBase整合

<http://storm.apache.org/releases/1.2.2/storm-hbase.html>

The main API for interacting with HBase is

the org.apache.storm.hbase.bolt.mapper.HBaseMapper interface:

## 插入数据

**SimpleHBaseMapper：**

HBaseMapper implementation called SimpleHBaseMapper that can map Storm tuples to both regular HBase columns as well as counter columns.

**HbaseBolt**

The HBaseBolt will delegate to the mapper instance to figure out how to persist tuple data to HBase.

## 查询数据

**HbaseValueMapper：**

This class allows you to transform the HBase lookup result into storm Values that will be emitted by the HbaseLookupBolt

**HbaseLookupBolt**

To use the HBaseLookupBolt, Construct it with the name of the table to write to, an implementation of HBaseMapper and an implementation of HBaseRowToStormValueMapper. You can optionally specify a HBaseProjectionCriteria.

The HBaseLookupBolt will use the mapper to get rowKey to lookup for. It will use the HBaseProjectionCriteria to figure out which columns to include in the result and it will leverage the HBaseRowToStormValueMapper to get the values to be emitted by the bolt.