Storm

OutLine

- Storm基础
- Storm架构
- Storm容错
- Storm开发

MapReduce数据处理特点

- 海量数据集
  - G ->T -> P 级都能处理
- 全量数据集同时处理
  - 一次性同时处理整个数据集
- 批处理方式
  - 大数据输入，大批数据输出

其他数据处理类型

- 实时数据分析需求
  - 实时报表动态展现
  - 数据流量波动状态
  - 反馈系统
- 时效性
  - 秒级处理完成数据
- 增量式处理
  - 数据来一条，处理一条

流式处理

- 时效性高
- 逐条处理数据
- 低延时
- 不是一个新概念
  - 管道（PIPE）
    - Cat input | grep pattern | sort | uniq > output
    - Cat input | python map.py | sort | python reduce.py > output

分布式流处理

- 单机处理不了
  - 内存
  - CPU
  - 存储
- 多机流式系统
  - 流量控制
  - 容灾冗余
  - 路径选择
  - 扩展

Storm

- 开源分布式实时计算系统
- Twitter出品
- 托管在GitHub上
- 目前互联网中应用最广泛
  - 相对稳定，有高容错性
  - 开源

Storm

- 没有持久化层
- 保证消息得到处理
- 支持多种编程语言
- 高效，用ZeroMQ作为底层消息队列
- 支持本地模式，可模拟集群所有功能
- 使用原语
  - 类同MapReduce中的Map、Reduce

Storm vs Hadoop

Hadoop

Storm

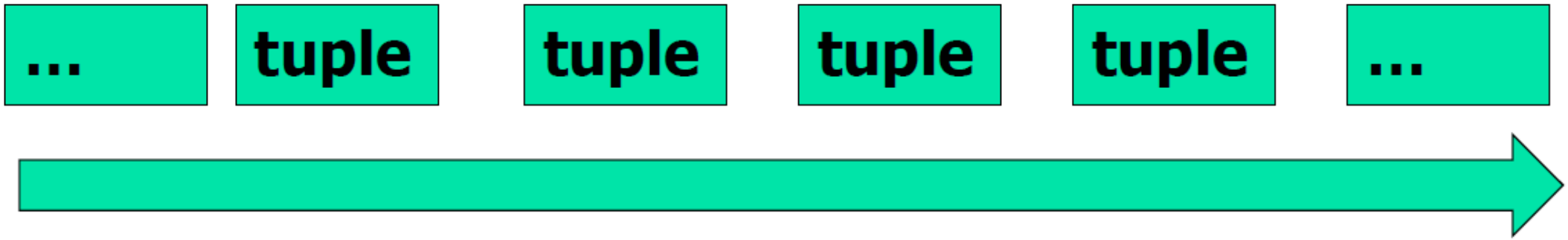
Storm vs Hadoop

- Storm任务没有结束，Hadoop任务执行完结束
- Storm延时更低，得益于网络直传、内存计算，省去了批处理的收集数据的时间
- Hadoop使用磁盘作为中间交换的介质，而storm的数据是一直在内存中流转的
- Storm的吞吐能力不及Hadoop，所以不适合批处理计算模型

1. 延时，指数据从产生到运算产生结果的时间
2. 吞吐，指系统单位时间处理的数据量

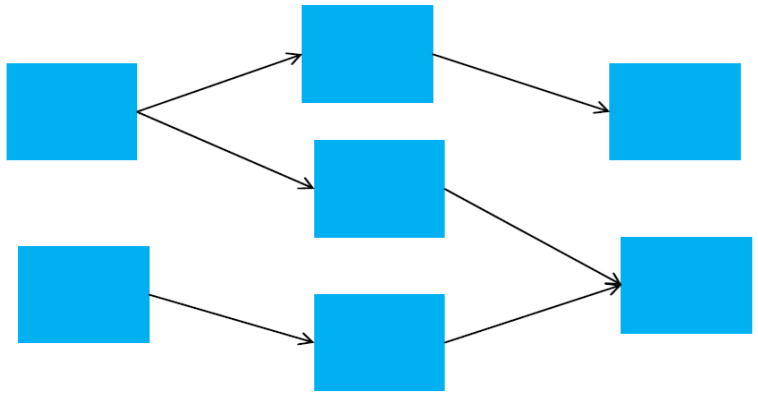
基本概念

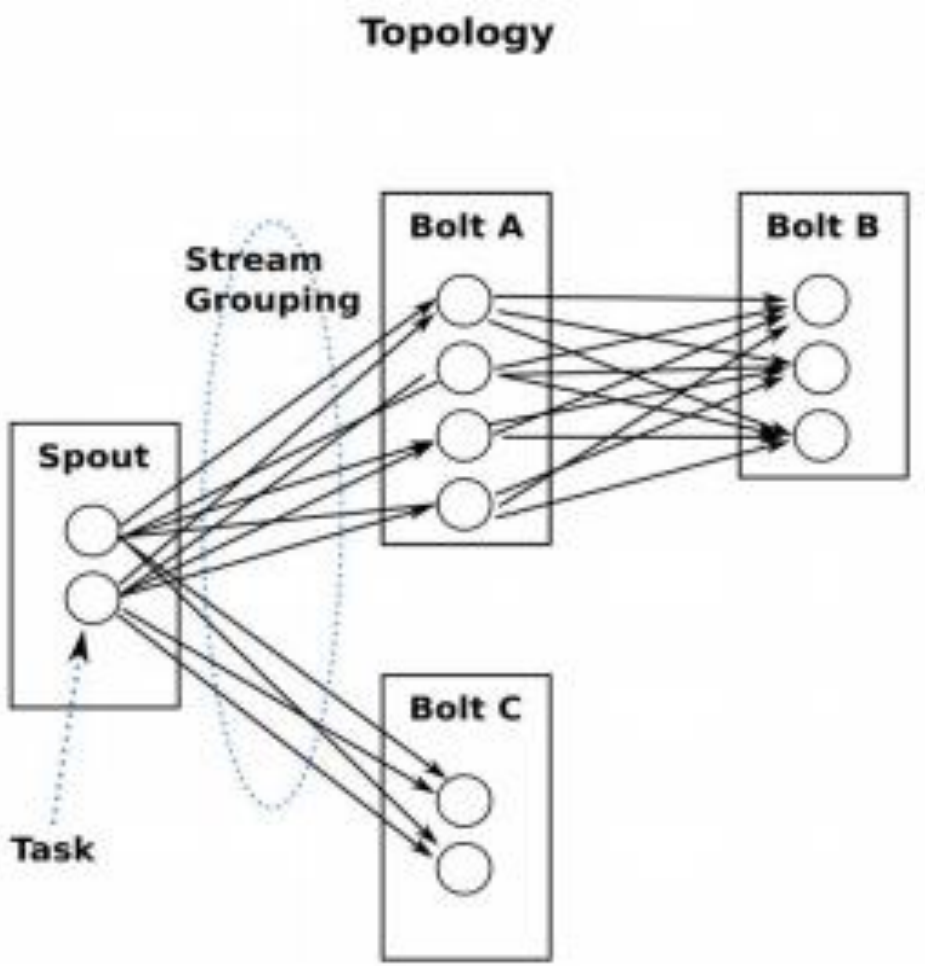
- Stream
  - 以Tuple为基本单位组成的一条有向无界的数据流
- Tuple
  - Integer,long,short,byte,string,double,float,boolean和byte array，包括自定义类型



基本概念

- Topology
  - 计算逻辑的封装
  - 由spouts和bolts组成的图，通过stream grouping将图中的spouts和bolts连接起来
  - 类同MapReduce中的job
    - 不会结束，除非主动kill





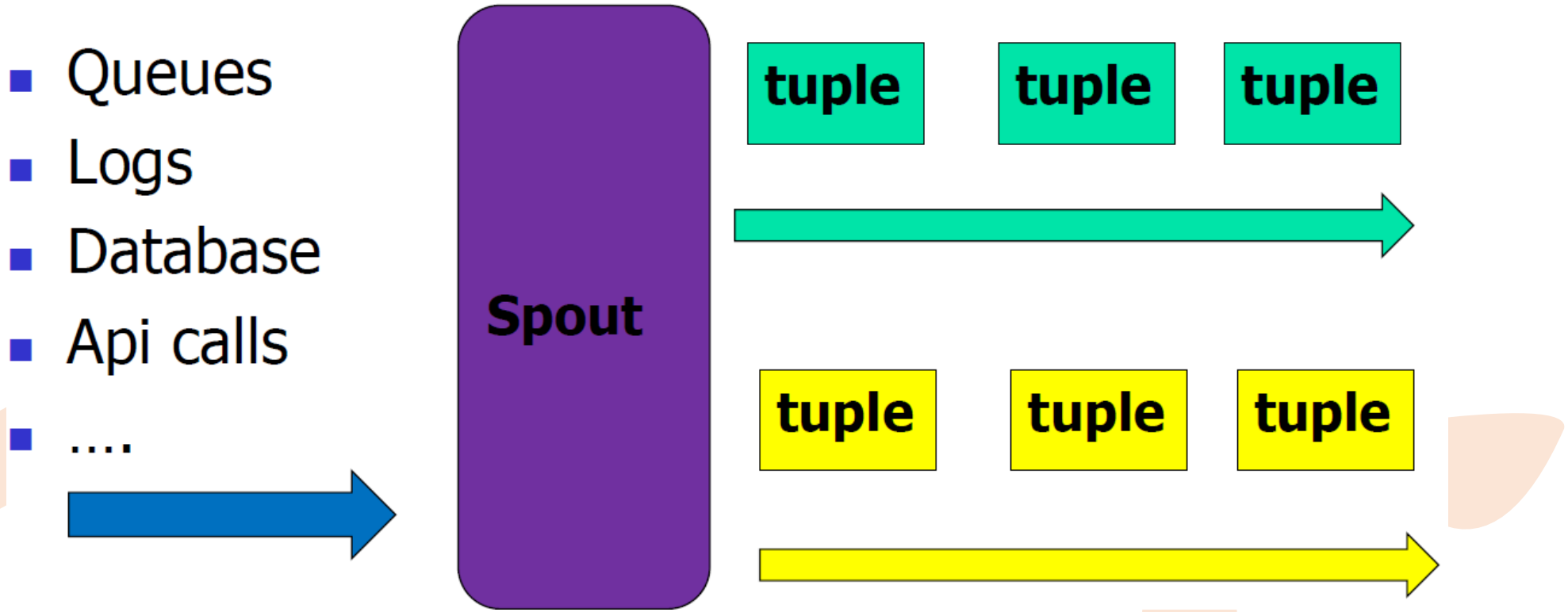
基本概念

- Topology任务执行
  - Storm jar code.jar MyTopology arg1 arg2
    - storm jar负责连接到Nimbus并且上传jar包
    - 运行主类 MyTopology, 参数是arg1, arg2；这个类的main函数定义这个topology并且把它提交给Nimbus
    - Topology的定义是一个Thrift结构，并且Nimbus就是一个Thrift服务， 你可以提交由任何语言创建的topology；

基本概念

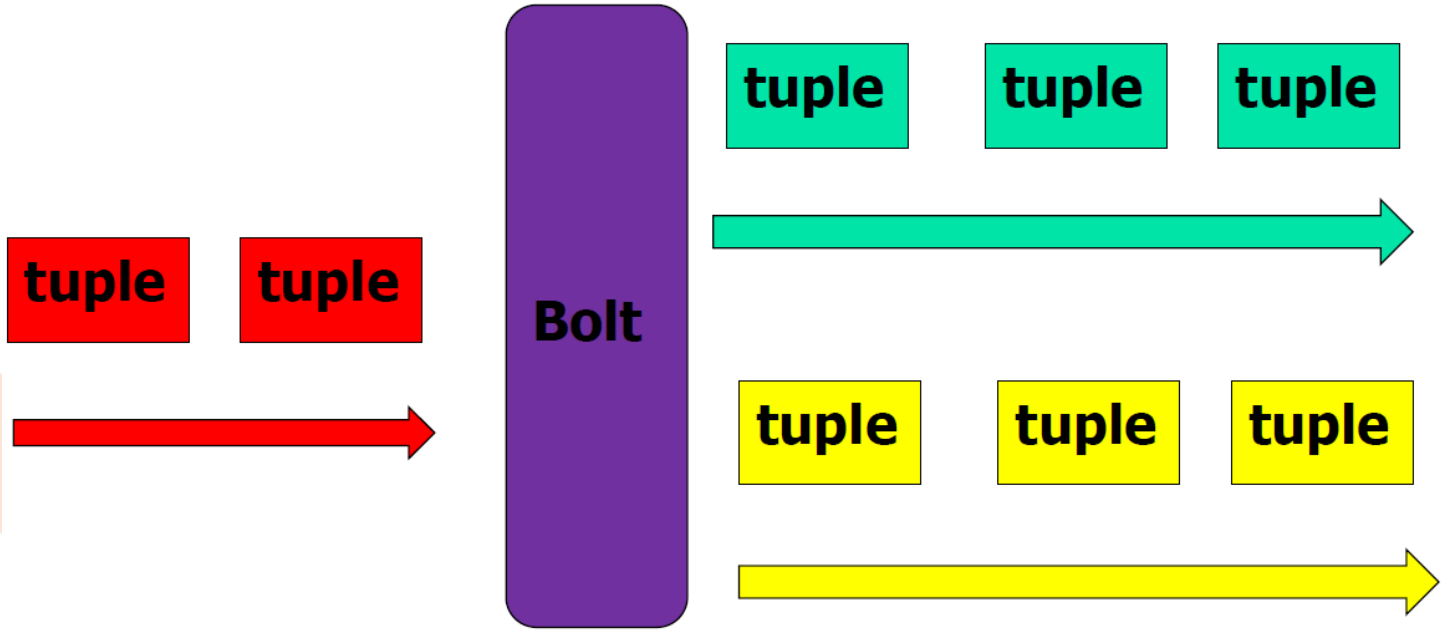
- Spout
  - 消息来源，消息生产者
  - 可靠的，不可靠的
    - 可靠的，如果没有被成功处理，可重新emit一个tuple
  - 可指定emit多个Stream流
    - OutFieldsDeclarer.declareStream定义
    - SpoutOutputCollector指定
  - nextTuple

基本概念

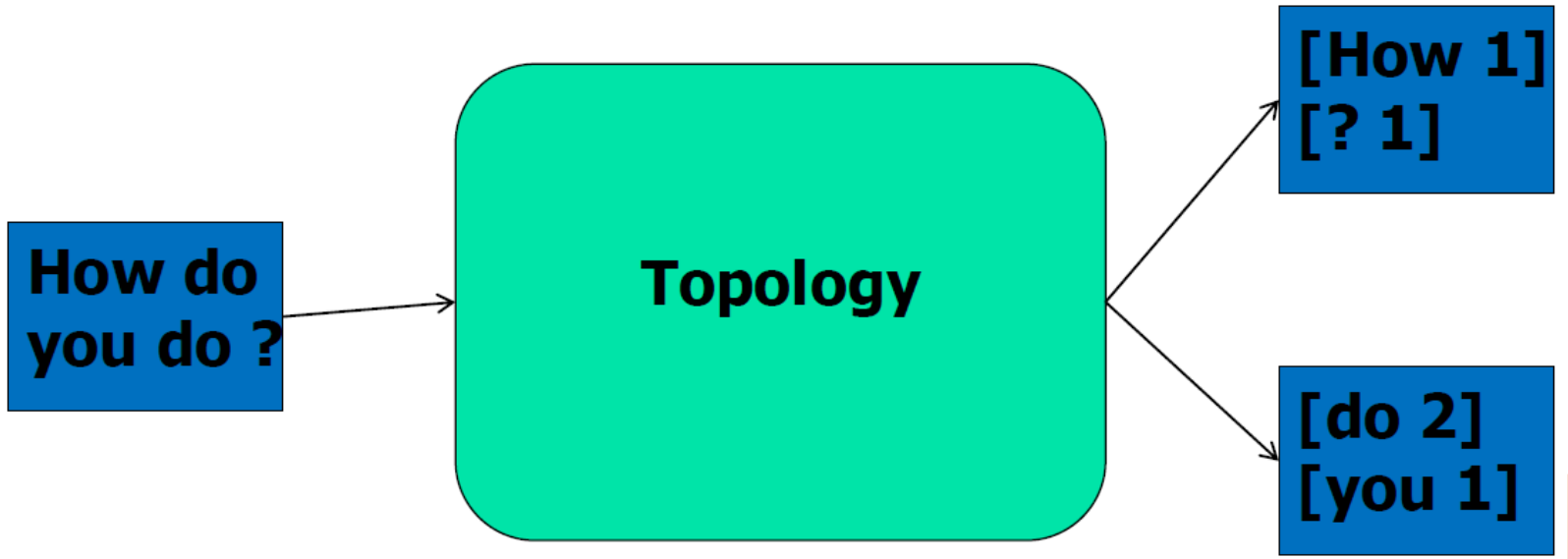


基本概念

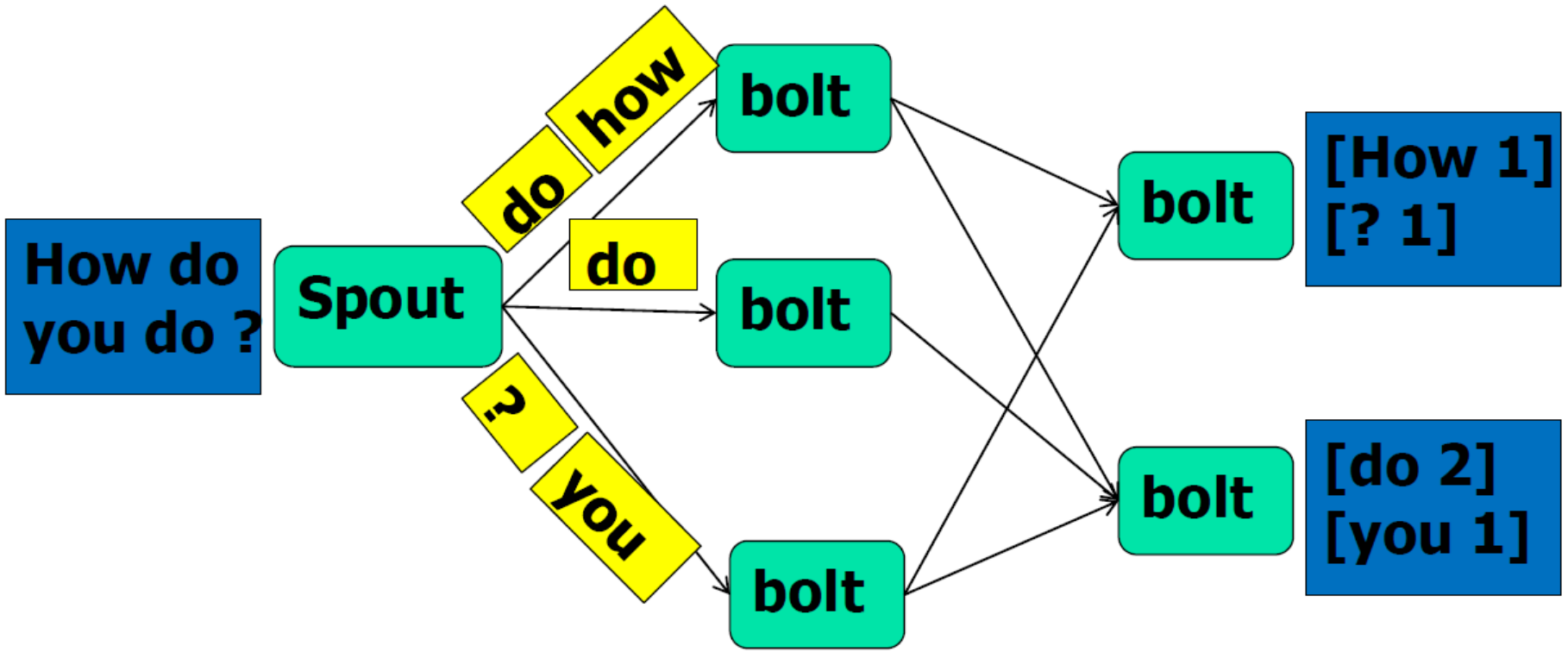
- Bolt
  - 消息处理逻辑
    - 如过滤，访问数据库，聚合
  - 多个bolt处理负责步骤
  - 可以发射多个数据流
  - 主方法为execute
    - 以tuple为输入
    - 处理具体的tuple
    - 发射0或多个tuple
    - OutputCollector的ack，确认
    - IBasicBolt，会自动调节



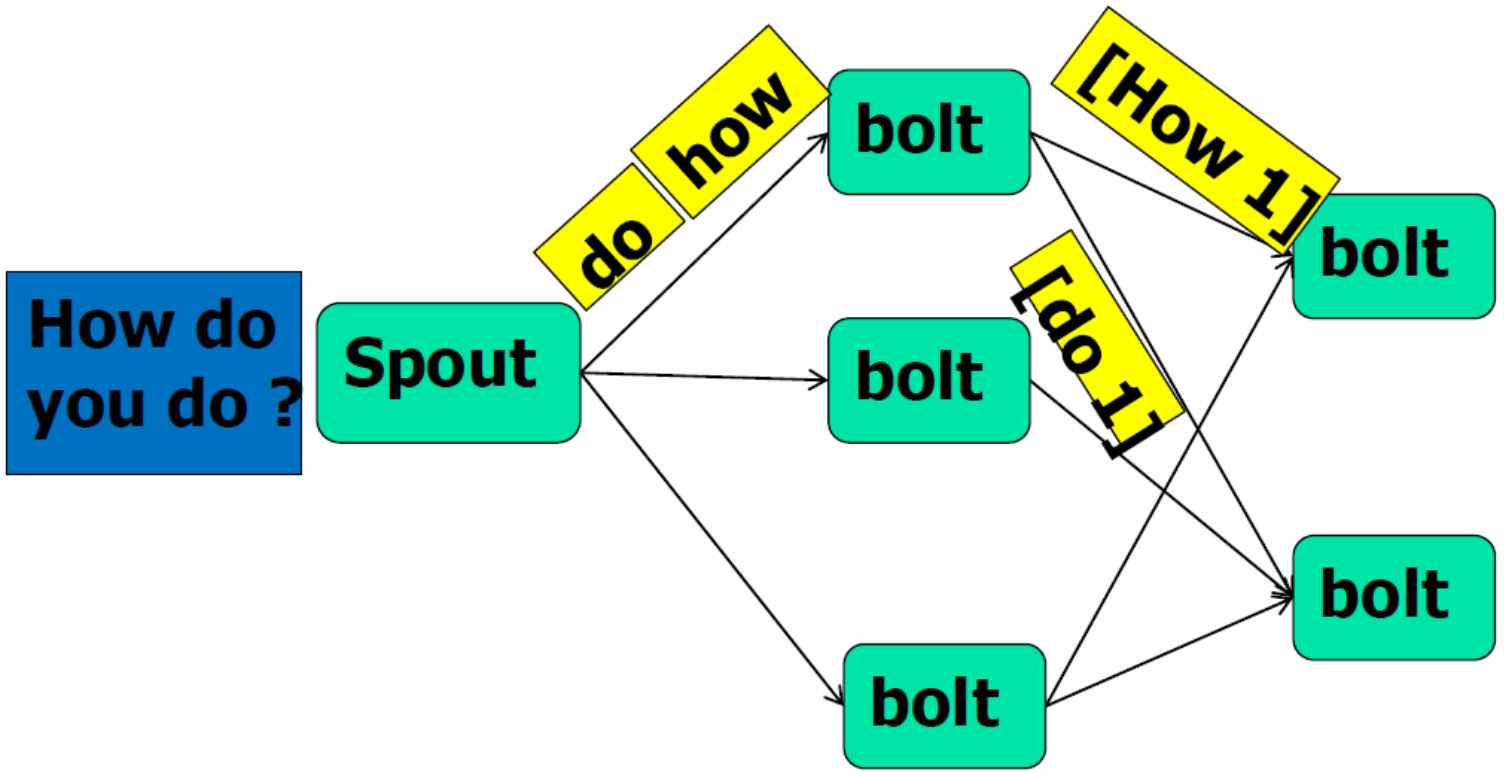
基本概念



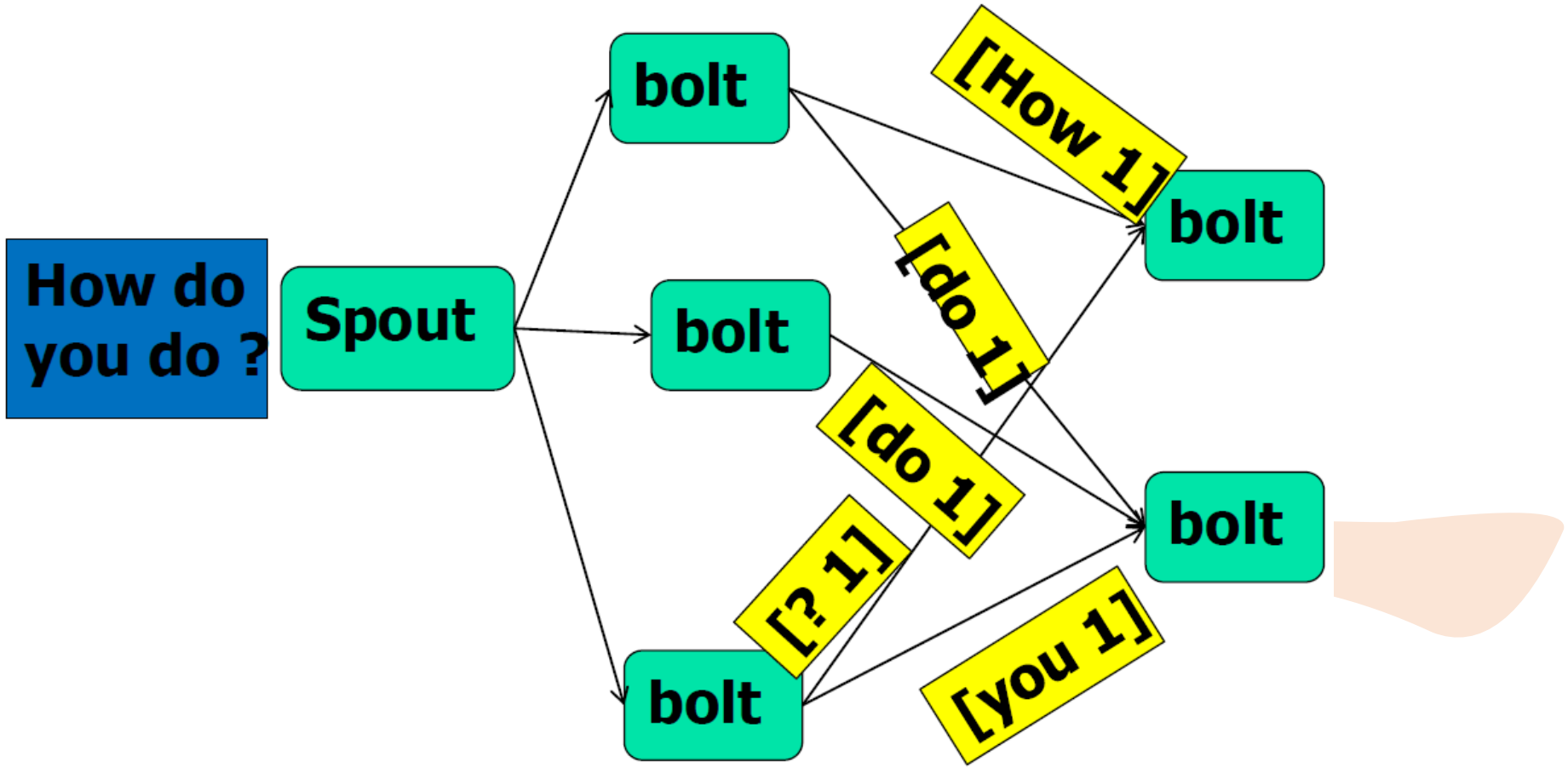
基本概念



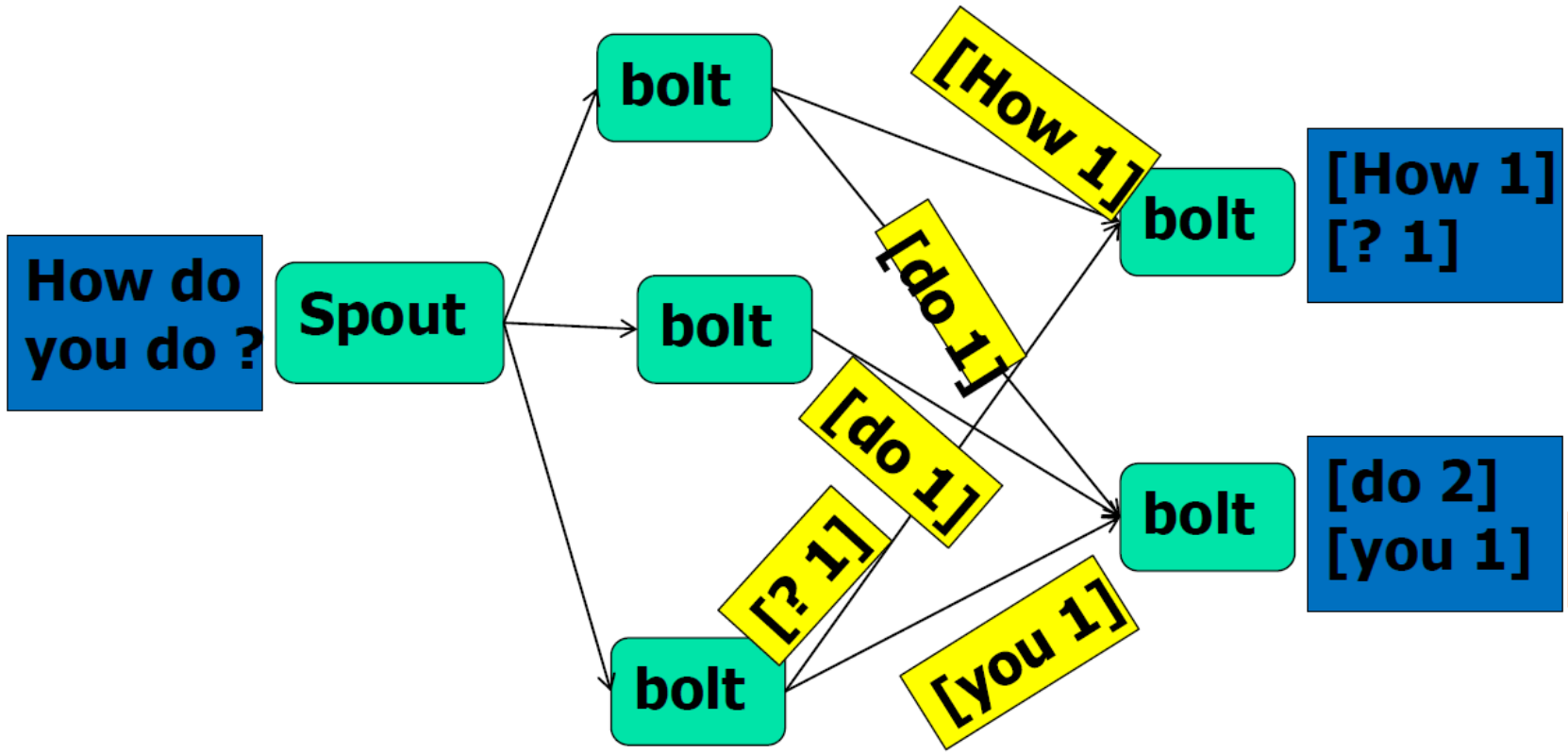
基本概念



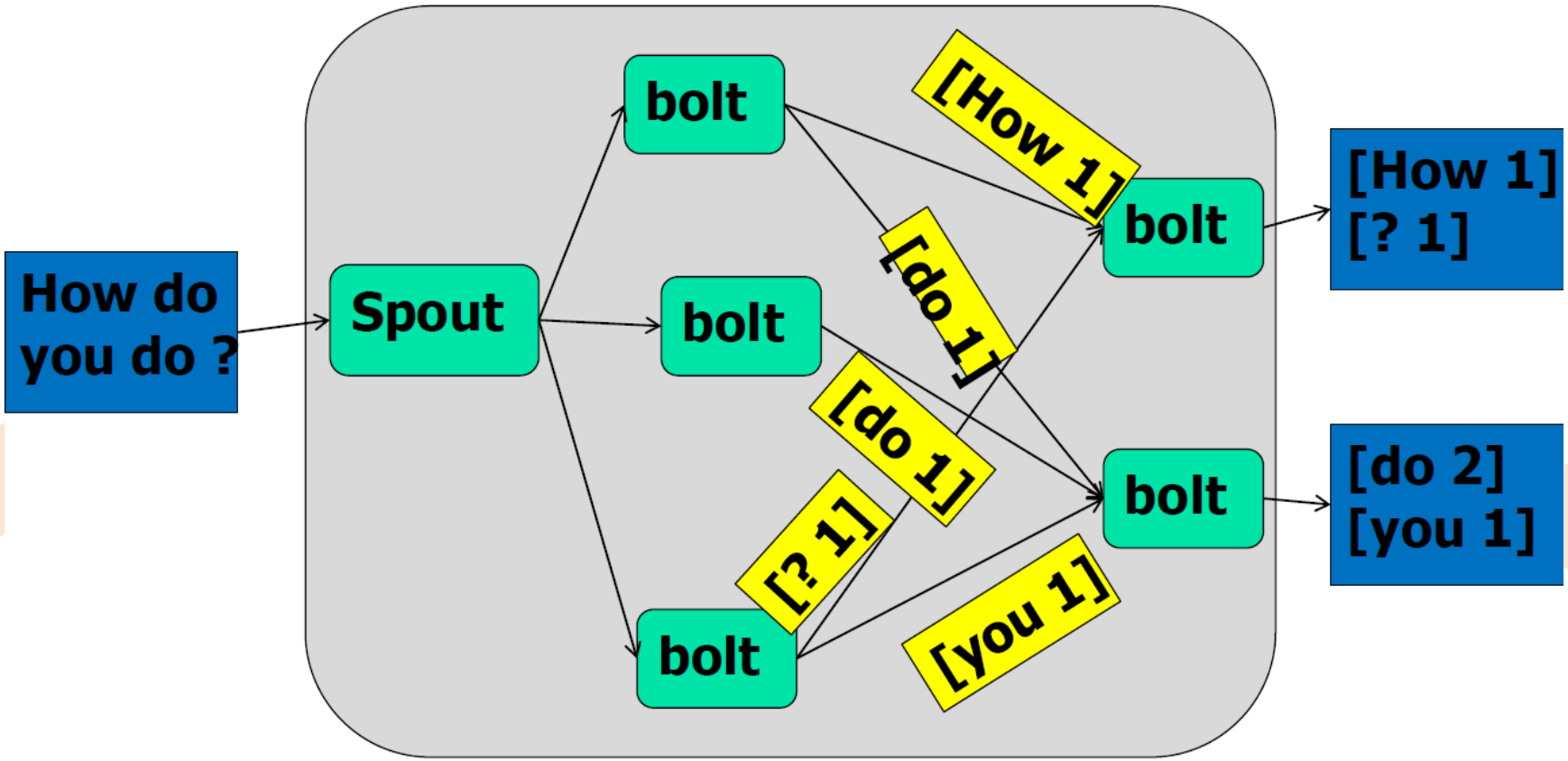
基本概念



基本概念

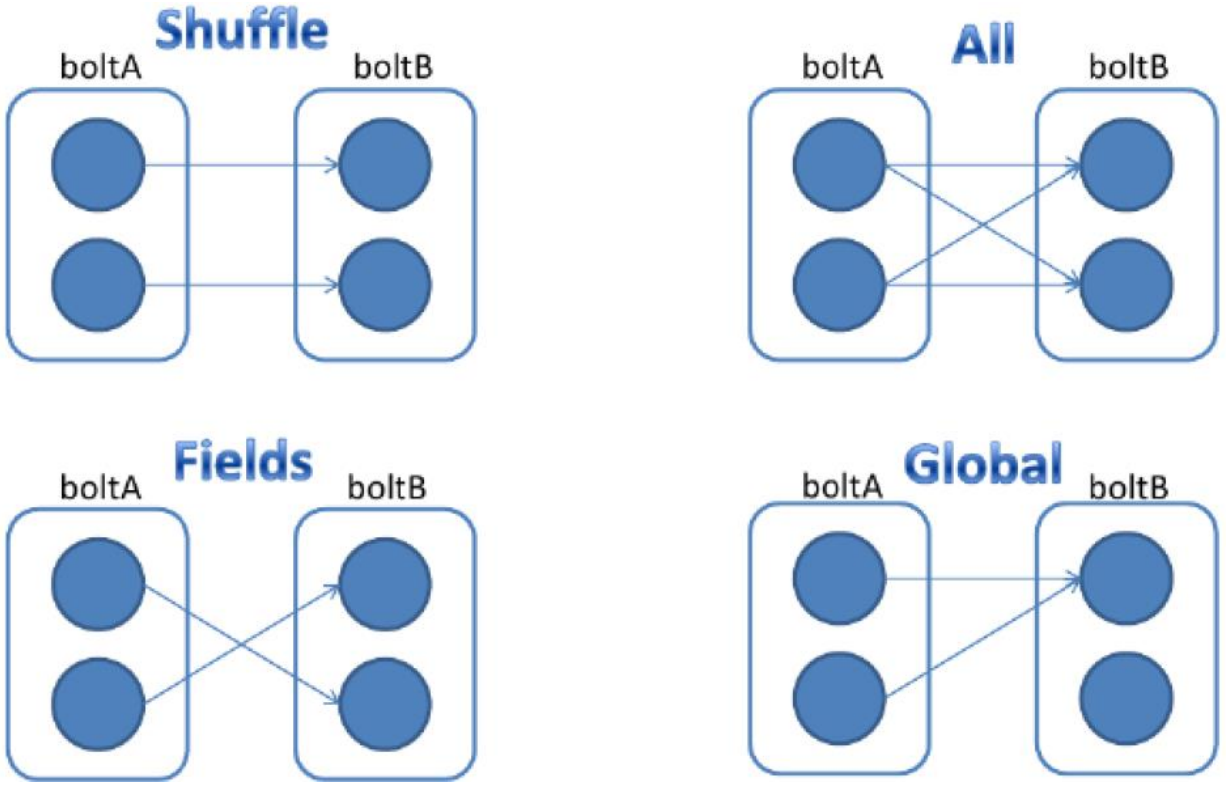


基本概念



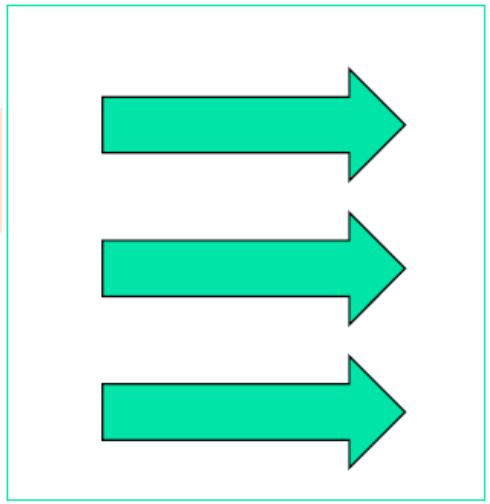
Grouping

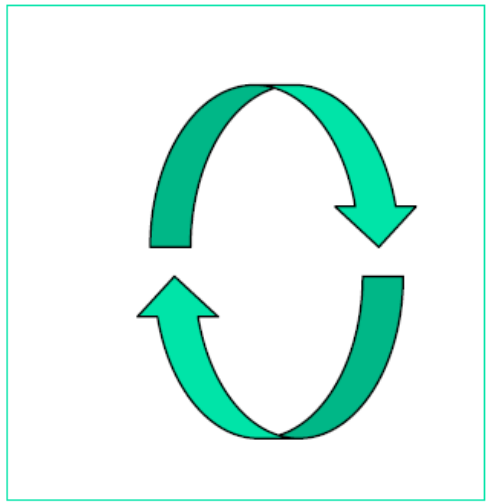
- Stream Grouping
  - Shuffle Grouping：随机分组
  - Fields Grouping:按指定的field分组
  - All Grouping：广播分组
  - Global Grouping：全局分组

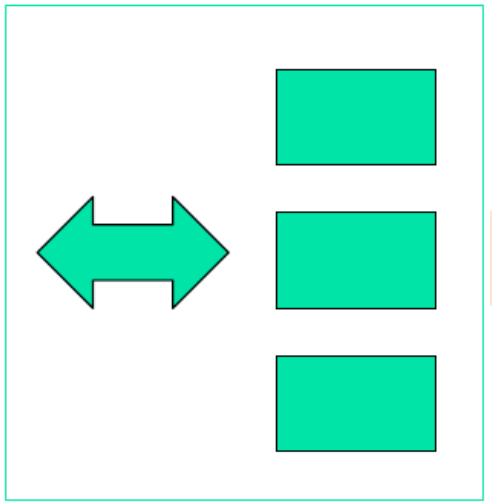


常见模式

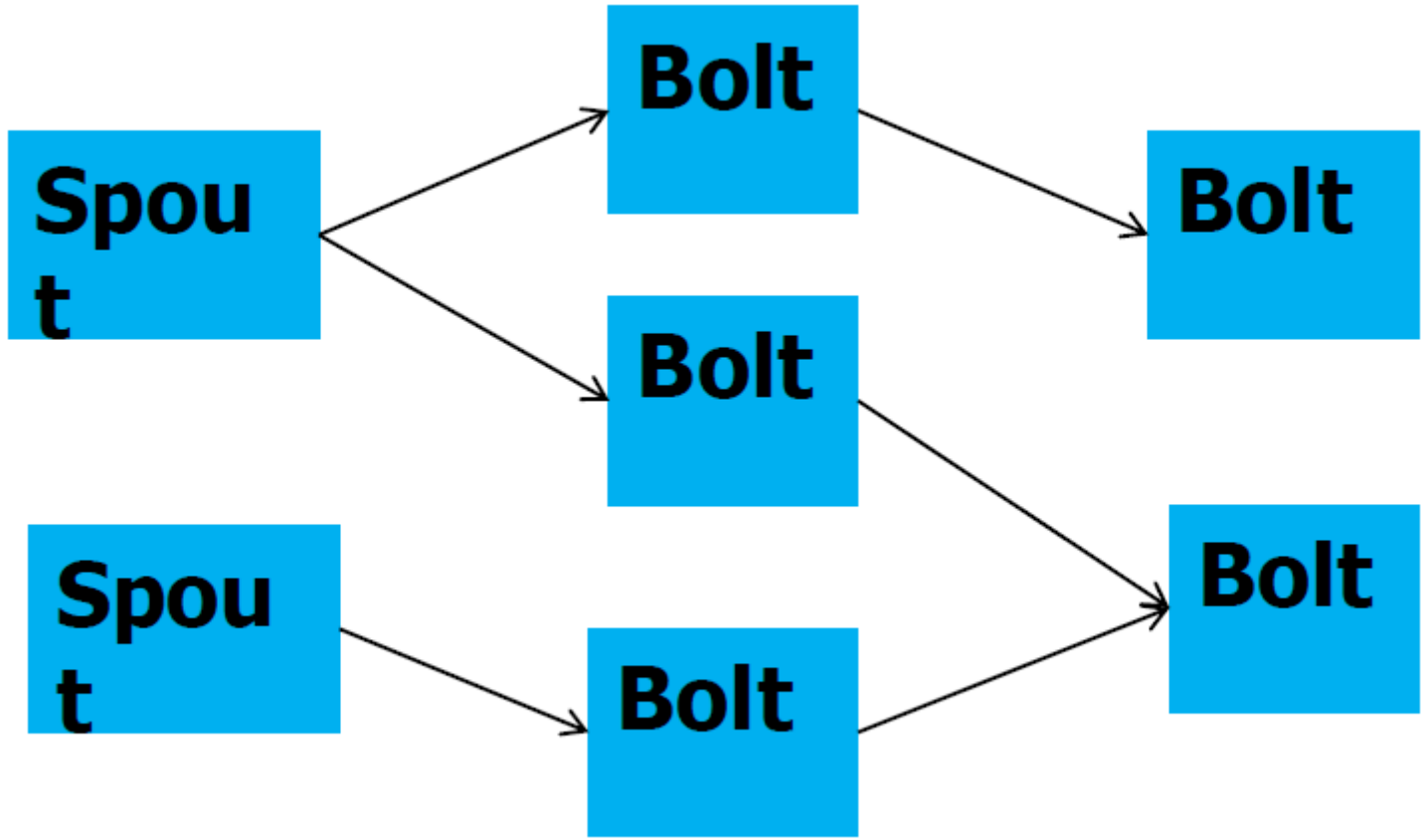
- 流式计算
- 持续计算
- 分布式RPC



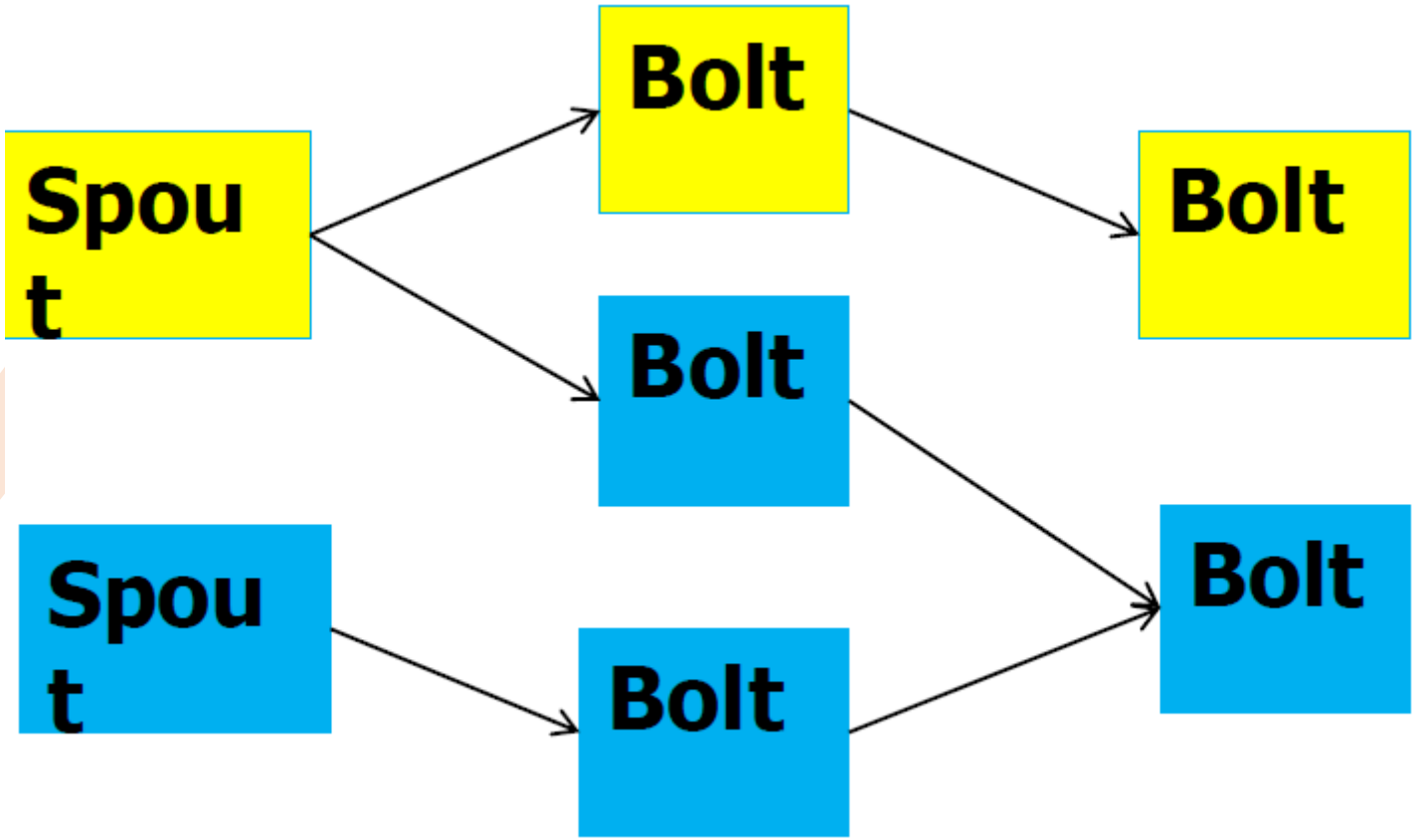




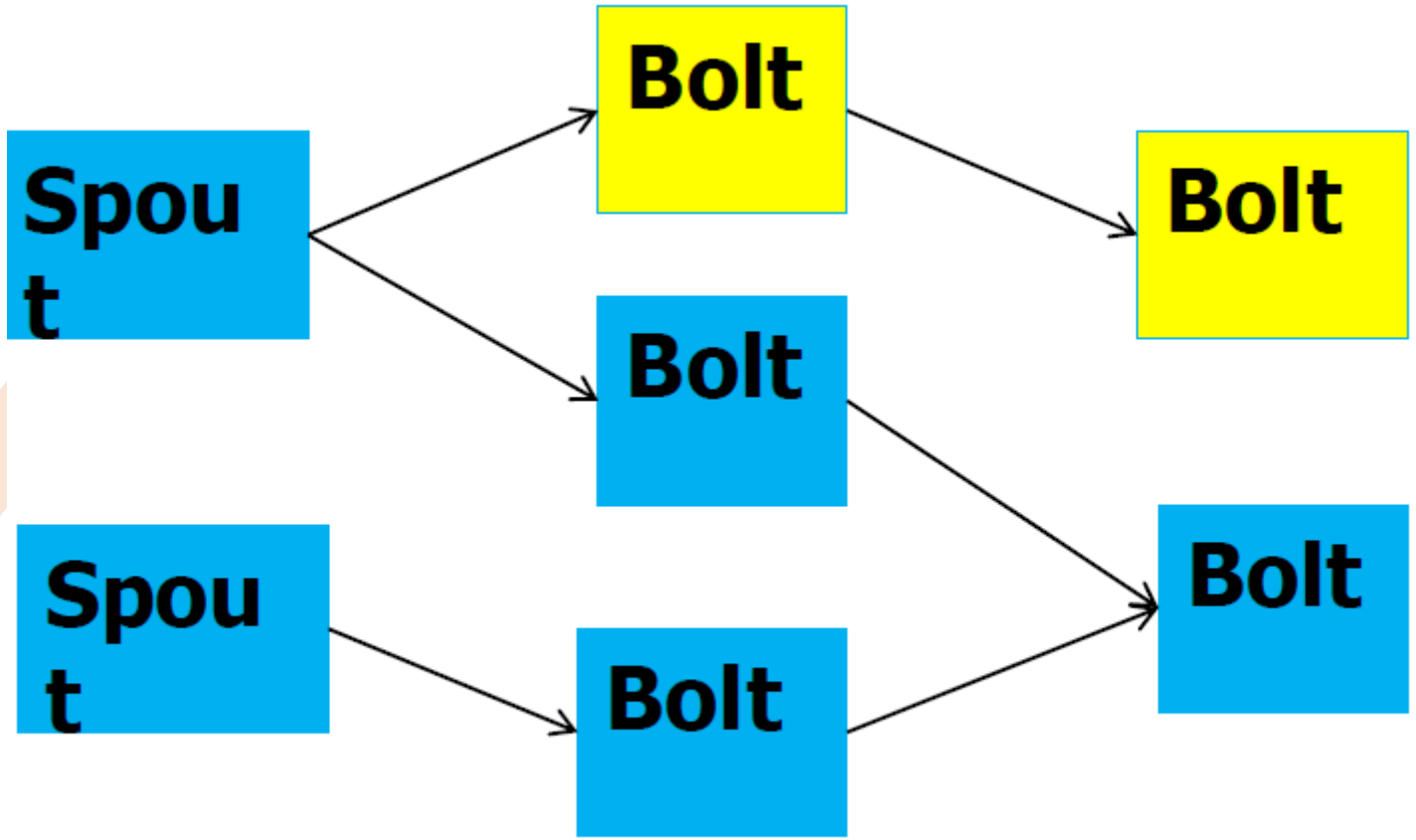
流式模式



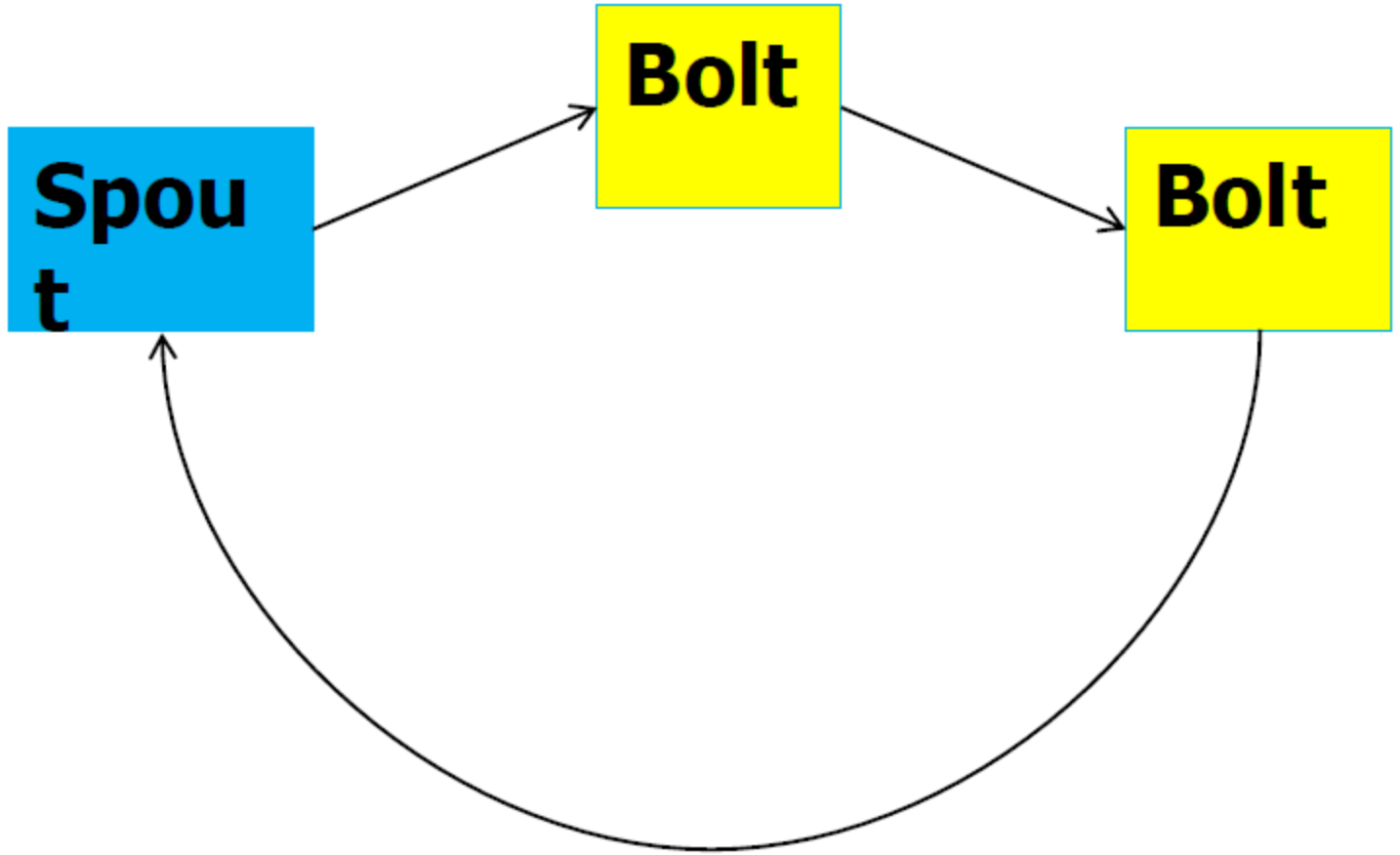
流式模式——过滤，组装



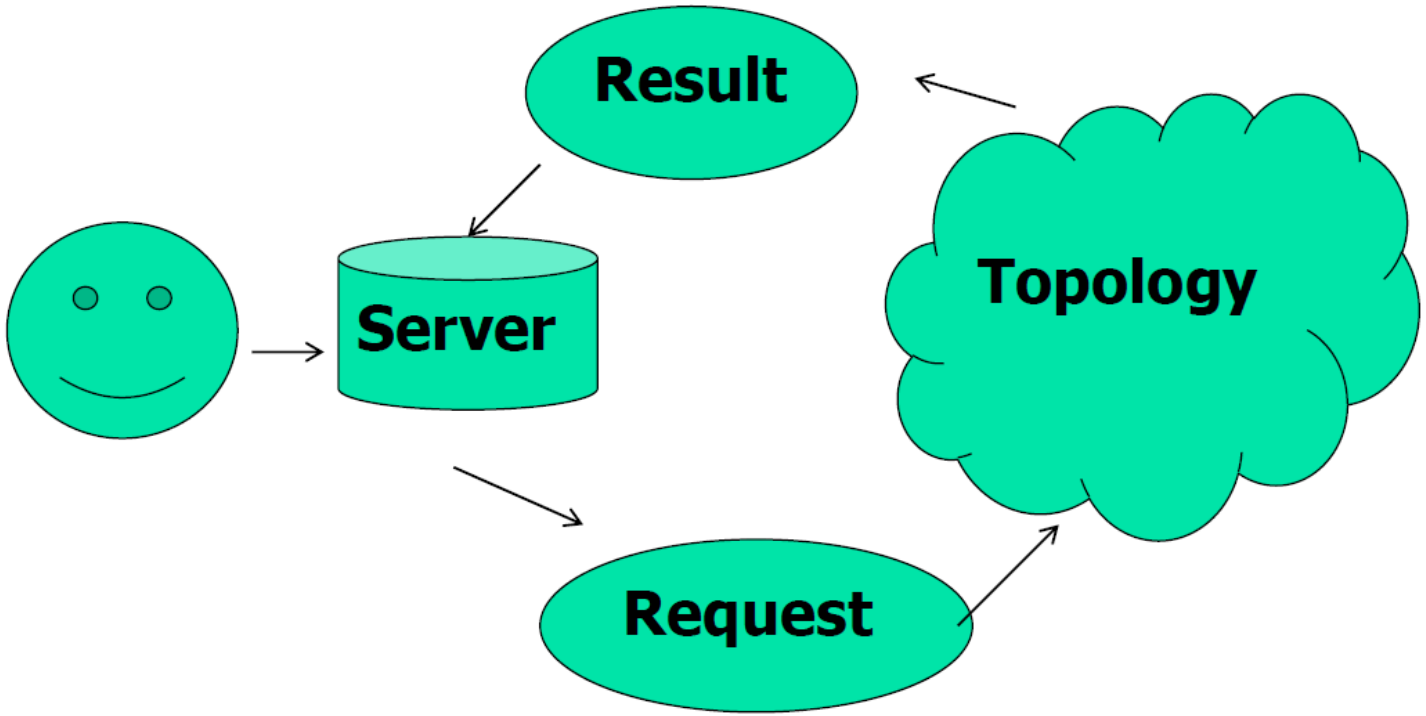
流式模式——Join



流式模式——持续计算



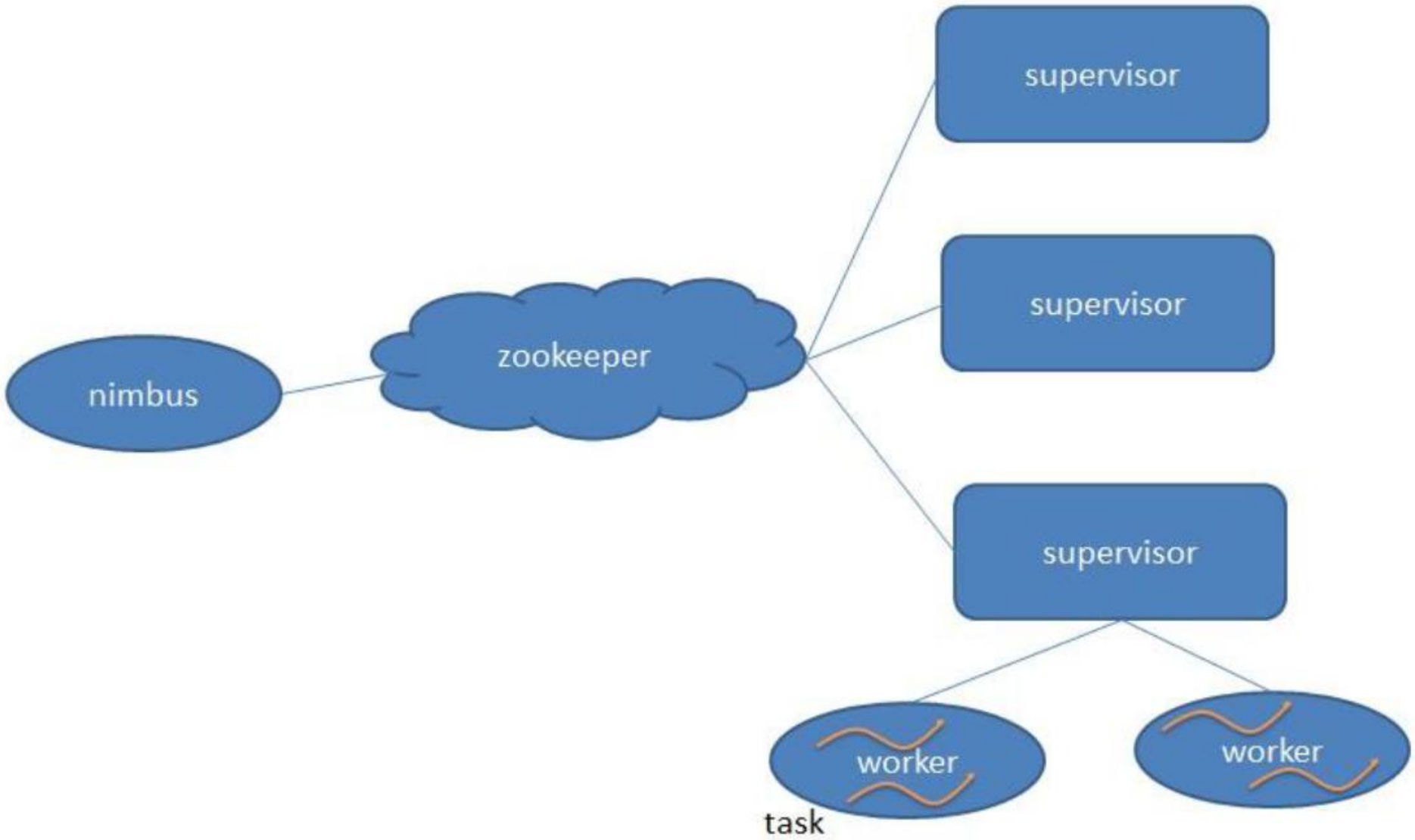
流式模式——分布式RPC



OutLine

- Storm基础
- Storm架构
- Storm容错
- Storm开发

Storm架构


supervisor
supervisor
zookeeper
nimbus
supervisor
worker
worker
task

Storm架构

	Hadoop	Storm
系统角色	JobTracker	Nimbus
	TaskTracker	Supervisor
	Child	Worker
应用名称	Job	Topology
组件接口	Mapper/Reducer	Spout/Bolt

Storm架构

- Nimbus
  - Master Node
  - 负责资源分配和任务调度
  - 类似Hadoop里的JobTracker，负责在集群里面分发代码，分配计算任务给Supervisor，并且监控状态

Storm架构

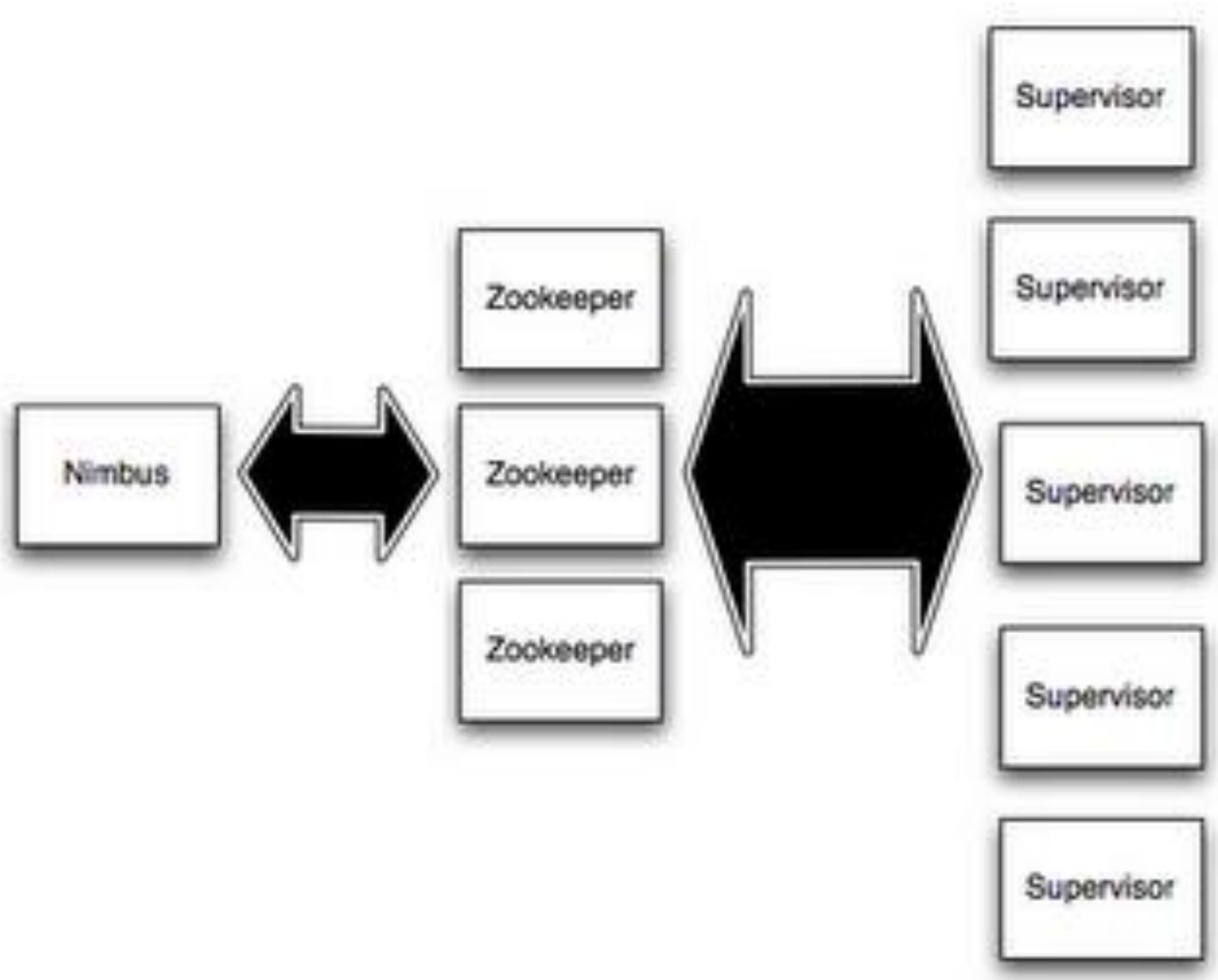
- Supervisor
  - Worker Node
  - 负责接收nimbus分配的任务
  - 每个工作节点存在一个
  - 启动和停止属于自己管理的worker进程（每一个工作进程执行一个Topology的一个子集，一个Topology由运行在很多机器上的很多worker工作进程组成）

Storm架构

- Nimbus和Supervisor之间的所有协调工作都是通过Zookeeper集群完成
- Nimbus进程和Supervisor进程都是**快速失败（fail-fast）和无状态的**。所有状态要么在Zookeeper里面，要么在本地磁盘上

- 这也就意味着你可以用kill -9来杀死Nimbus和Supervisor进程， 然后再重启它们，就好像什么都没有发生过。这个设计使得Storm异常的稳定。

Storm架构


Supervisor
Supervisor
Supervisor
Supervisor
Supervisor
Zookeeper
Zookeeper
Zookeeper
Nimbus

Storm架构

- Worker
  - 运行具体处理组件逻辑的进程
  - 一个Topology可能会在一个或者多个worker里面执行
  - 每个worker是一个物理JVM并且执行整个Topology的一部分
  - 采取JDK的Executor

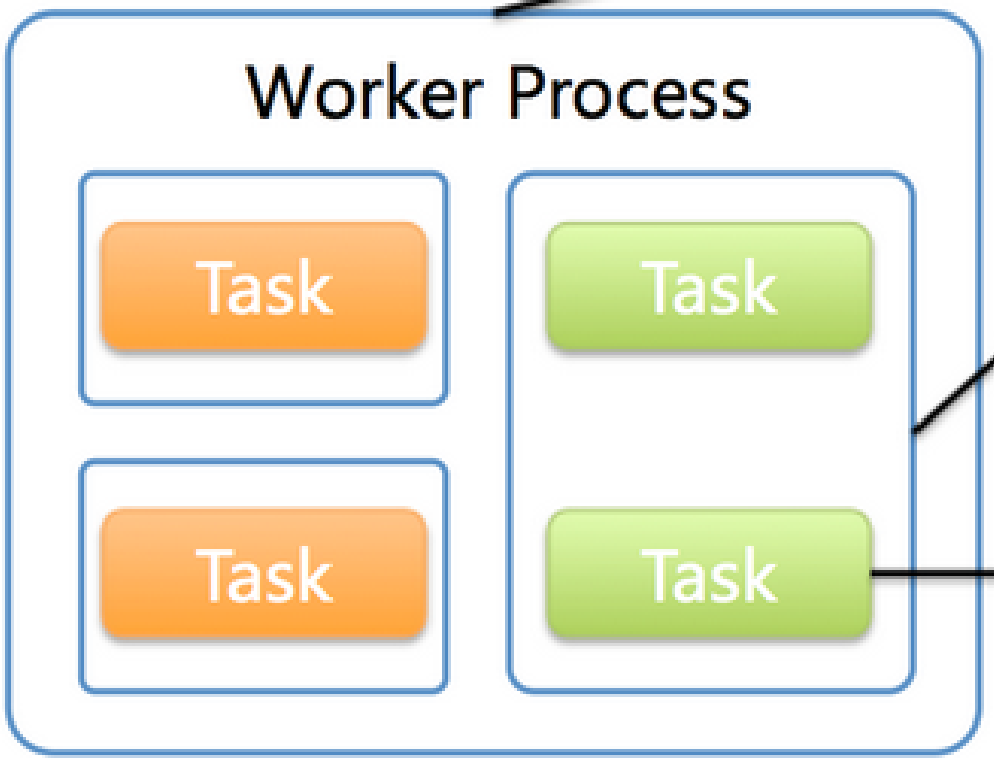
比如，对于并行度是300的topology来说，如果我们使用50个工作进程来执行，那么每个工作进程会处理其中的6个tasks，Storm会尽量均匀的工作分配给所有的worker

Storm架构

- Task
  - Worker中的每一个spout/bolt的线程称为一个task
  - 每一个spout和bolt会被当作很多task在整个集群里执行
  - 每一个executor对应到一个线程，在这个线程上运行多个task
  - stream grouping则是定义怎么从一堆task发射tuple到另外一堆task
  - 可以调用TopologyBuilder类的setSpout和setBolt来设置并行度（也就是有多少个task）

Storm架构

- Worker与Task关系



storm集群里的1台物理机会启动1个或多个worker进程（即：jvm进程），所有的topology将在这些worker进程里被运行。

在1个单独的worker进程里会运行1个或多个executor线程。每个executor只会运行1个topology的1个component(spout或bolt)的task实例

1个task是最终完成数据处理的实体单元。

Storm架构

- Worker和Task关系
  - 1个worker进程执行的是1个topology的子集（注：不会出现1个worker为多个topology服务）。1个worker进程会启动1个或多个executor线程来执行1个topology的component(spout或bolt)。因此，1个运行中的topology就是由集群中多台物理机上的多个worker进程组成的。
  - executor是1个被worker进程启动的单独线程。每个executor只会运行1个topology的1个component(spout或bolt)的task（注：task可以是1个或多个，storm默认是1个component只生成1个task，executor线程里会在每次循环里顺序调用所有task实例）。
  - task是最终运行spout或bolt中代码的单元（注：1个task即为spout或bolt的1个实例，executor线程在执行期间会调用该task的nextTuple或execute方法）。topology启动后，1个component(spout或bolt)的task数目是固定不变的，但该component使用的executor线程数可以动态调整（例如：1个executor线程可以执行该component的1个或多个task实例）。这意味着，对于1个component存在这样的条件：#threads<=#tasks（即：线程数小于等于task数目）。默认情况下task的数目等于executor线程数目，即1个executor线程只运行1个task。

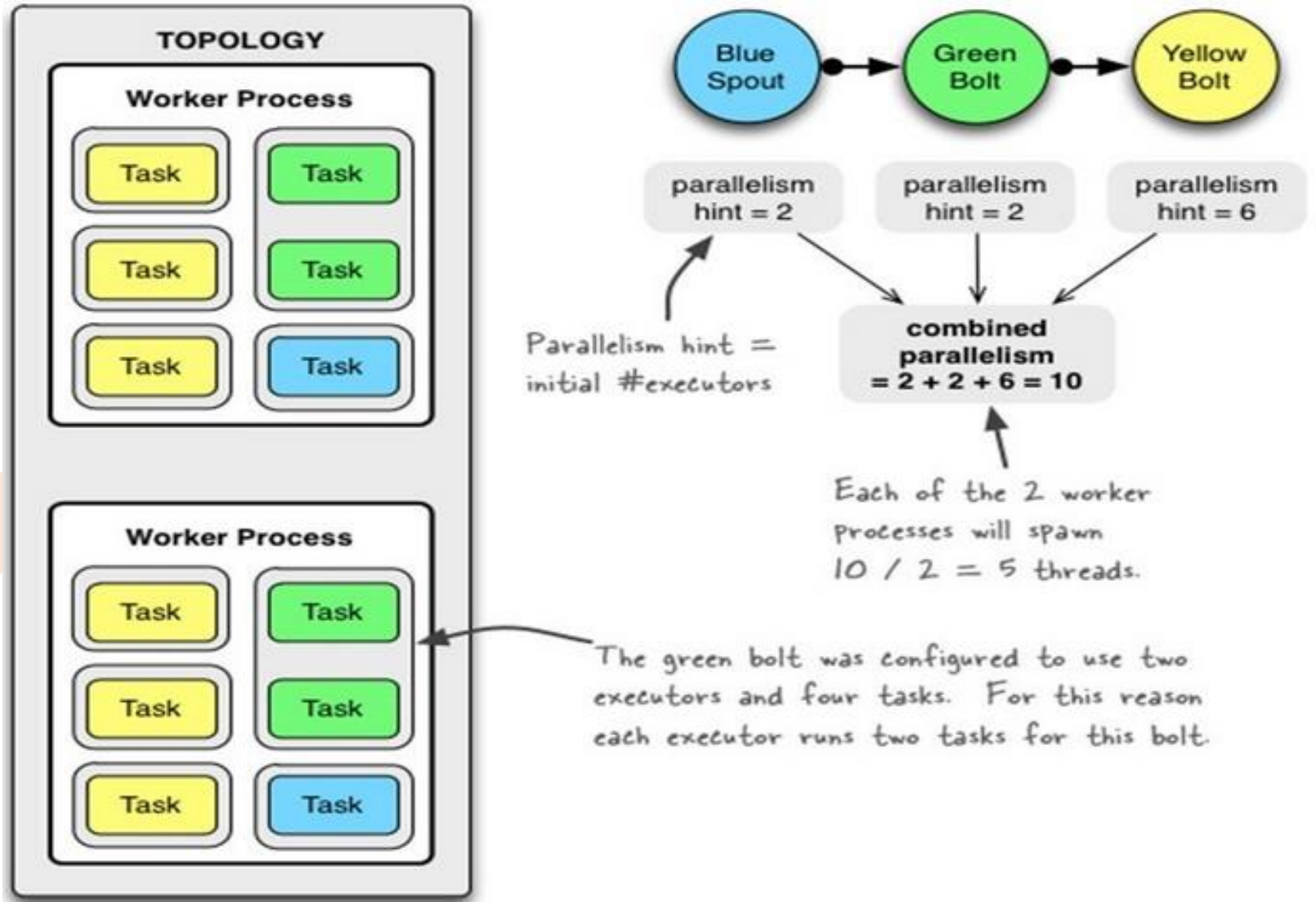
Storm架构

- Config conf = new Config();

- //设置Worker数量
- conf.setNumWorkers(2);

- // 设置Executor数量
- topolopyBuilder.setSpout("BlueSpout", new BlueSpout(), 2);
- topolopyBuilder.setBolt("GreenBolt", new GreenBolt(), 2)
- .setNumTasks(4) // 设置Task数量
- .shuffleGrouping("BlueSpout");
- topolopyBuilder.setBolt("YellowBolt", new YellowBolt(), 6)
- .shuffleGrouping("GreenBolt");

Storm架构


TOPOLOGY
Worker Process
Task
Task
Task
Task
Task
Task
Worker Process
Task
Task
Task
Task
Task
Task
Blue Spout
Green Bolt
Yellow Bolt
parallelism hint = 2
parallelism hint = 2
parallelism hint = 6
Parallelism hint = initial #executors
combined parallelism = 2 + 2 + 6 = 10
Each of the 2 worker processes will spawn 10 / 2 = 5 threads.
The green bolt was configured to use two executors and four tasks. For this reason each executor runs two tasks for this bolt.

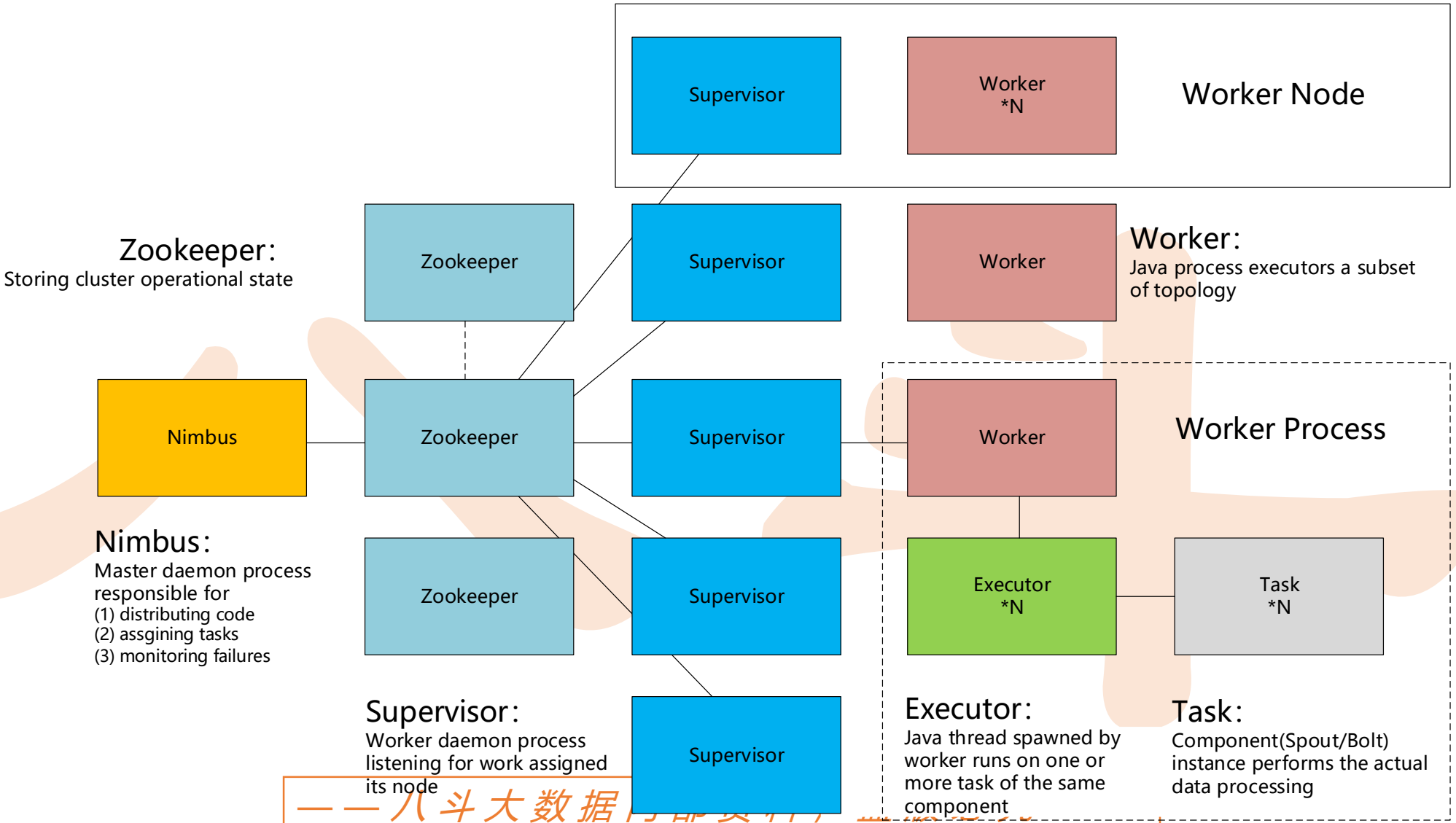
Storm架构

- 重新配置Topology "myTopology"使用5个Workers
- BlueSpout使用3个Executors
- YellowSpout使用10个Executors

-]# storm rebalance myTopology -n 5 -e BlueSpout=3 -e YellowBolt=10

总结：一个topology可以通过setNumWorkers来设置worker的数量，通过设置parallelism来规定executor的数量（一个component（spout/bolt）可以由多个executor来执行），通过setNumTasks来设置每个executor跑多少个task（默认为一对一）。
task是spout和bolt执行的最小单元。

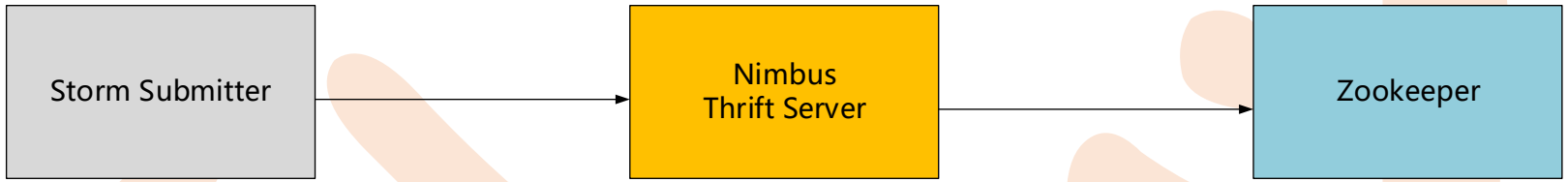
Storm架构



Storm架构

> **bin/storm jar**

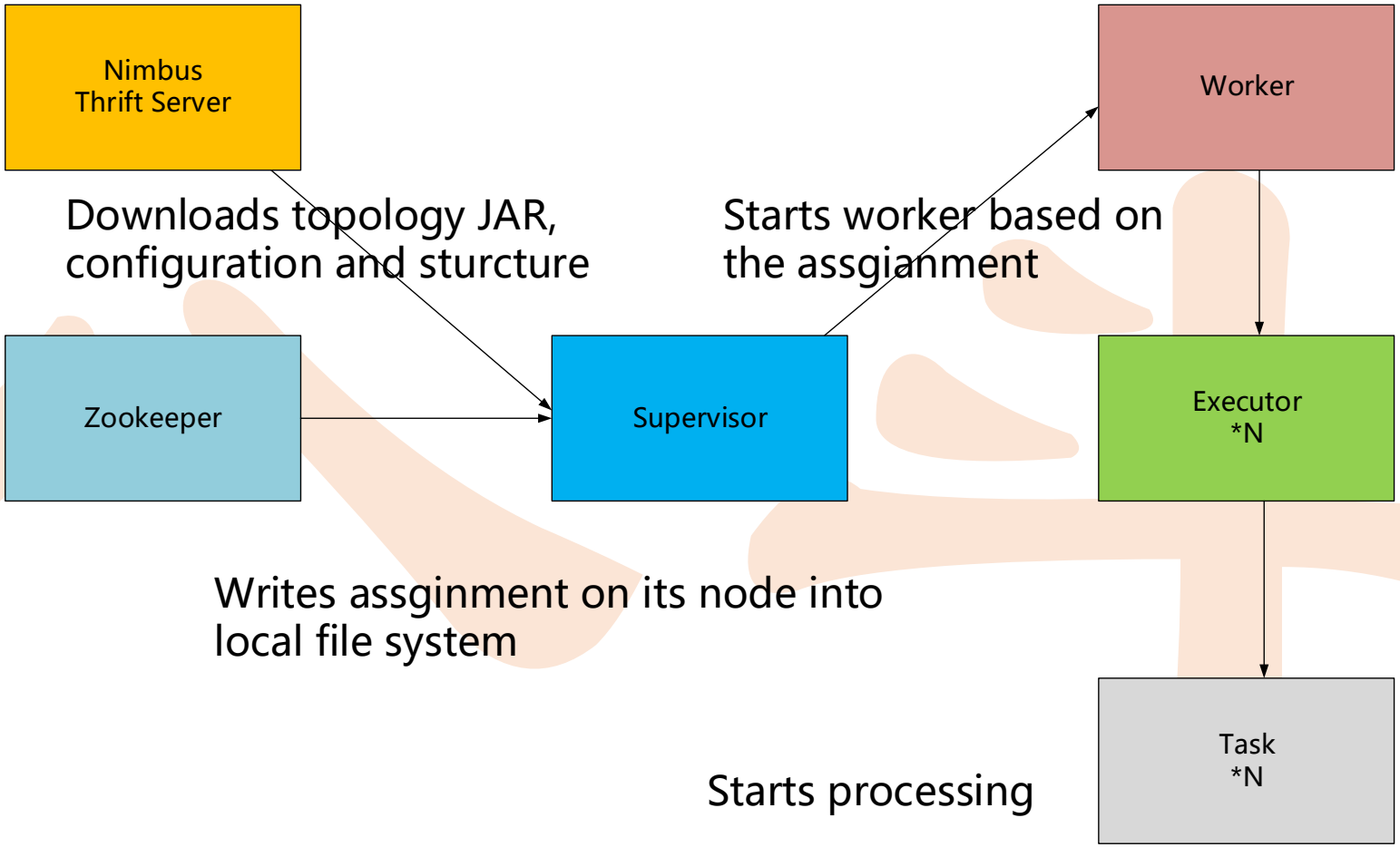
Copies topology JAR.
Configuration and structure into local file system



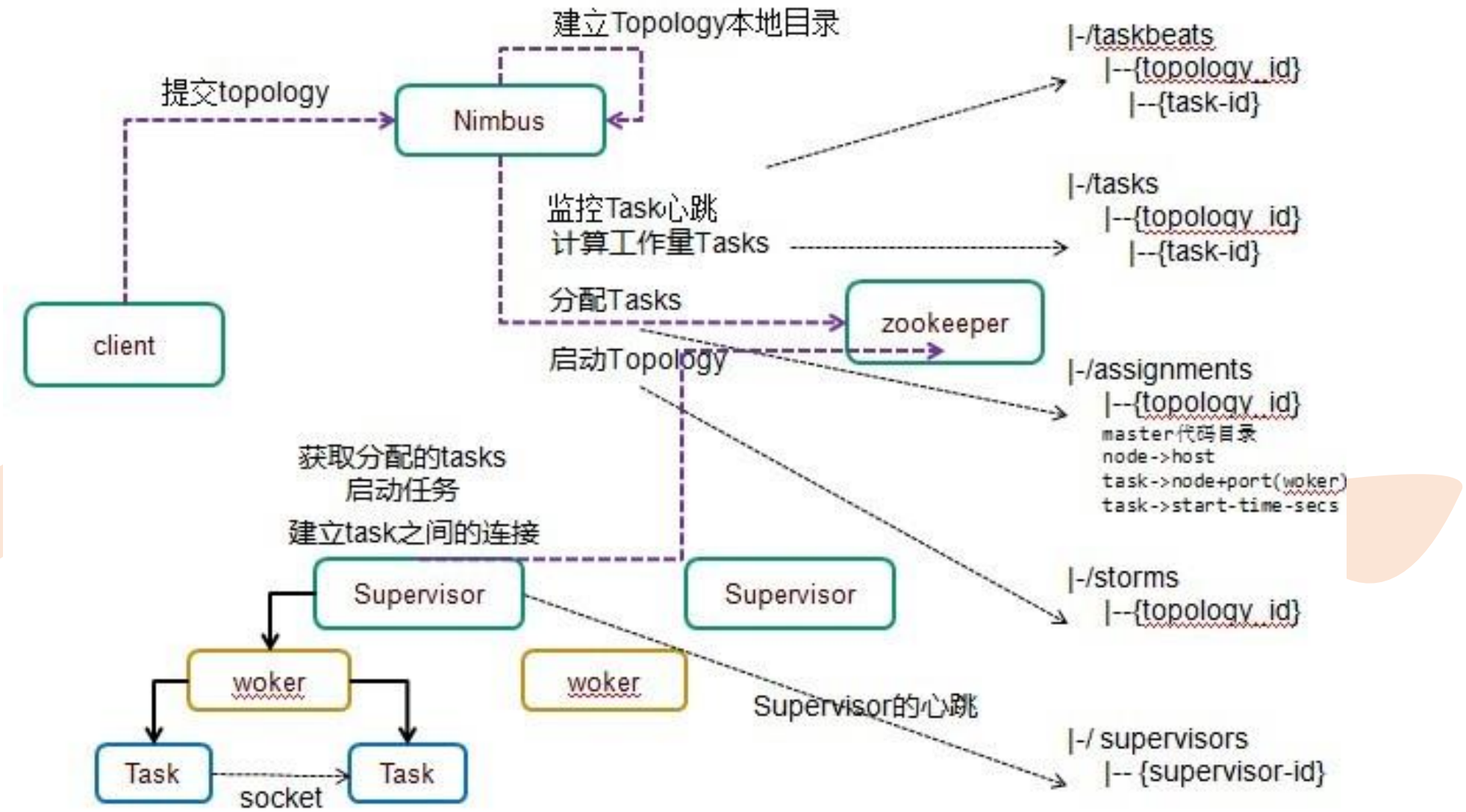
Upload topology JAR to
Nimbus' inbox with dependencies
Submits topoplogy configurations
as JSON and sturcture as Thrift

Set up static information for topology
Makes assgianment Starts topology

Storm架构



Storm架构


建立Topology本地目录
提交topology
Nimbus
client
|-/taskbeats
|--{topology_id}
|--{task-id}
监控Task心跳
计算工作量Tasks
|-/tasks
|--{topology_id}
|--{task-id}
分配Tasks
zookeeper
启动Topology
|-/assignments
|--{topology_id}
master代码目录
node->host
task->node+port(woker)
task->start-time-secs
获取分配的tasks
启动任务
建立task之间的连接
Supervisor
Supervisor
|-/storms
|--{topology_id}
woker
woker
Supervisor的心跳
|-/ supervisors
|-- {supervisor-id}
Task
Task
socket

OutLine

- Storm基础
- Storm架构
- Storm容错
- Storm开发

Storm容错——架构容错

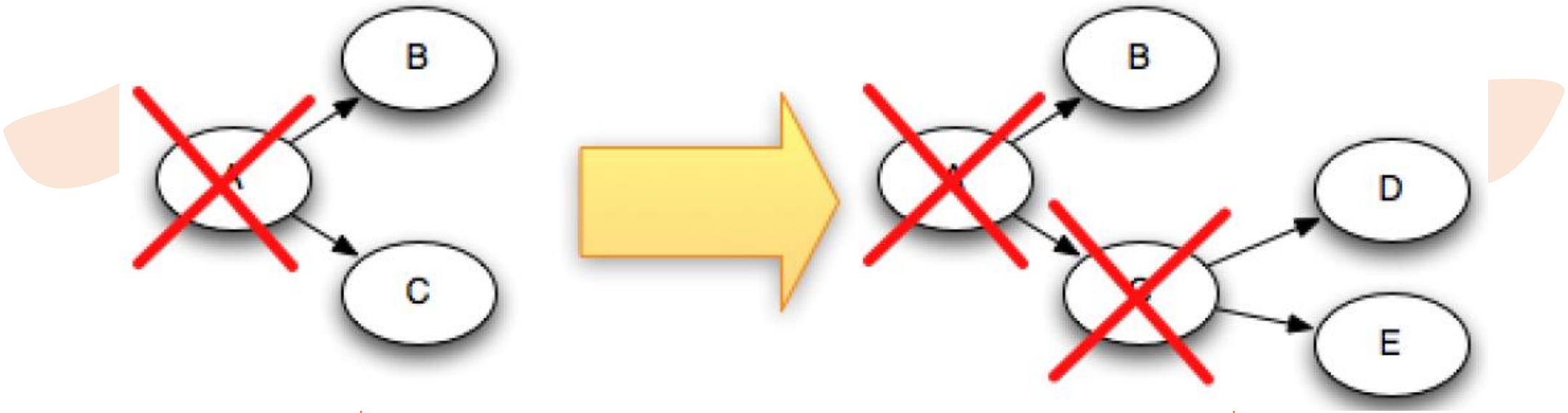
- Zookeeper
  - 存储Nimbus与Supervisor数据
- 节点宕机
  - Heartbeat
  - Nimbus
- Nimbus/Supervisor宕机
  - Worker继续工作
  - 任务失败
- Worker出错Worker失败，
  - Supervisor重启Worker

Storm容错——数据容错

- Storm的可靠性是指Storm会告知用户每一个消息单元是否在一个指定的时间(timeout)内被完全处理。
- Ack机制（Storm中的每一个Topology中都包含有一个Acker组件）
- 所有的节点ack成功，任务成功

Storm容错——数据容错

- 特殊的Task（Acker Bolt）
  - Acker，跟踪每一个spout发出的tuple树
  - 一个tuple树完成时，发送消息给tuple的创造者
  - Acker的数量， 默认值是1
    - 如果你的topology里面的tuple比较多的话， 那么把acker的数量设置多一点，效率会高一点。



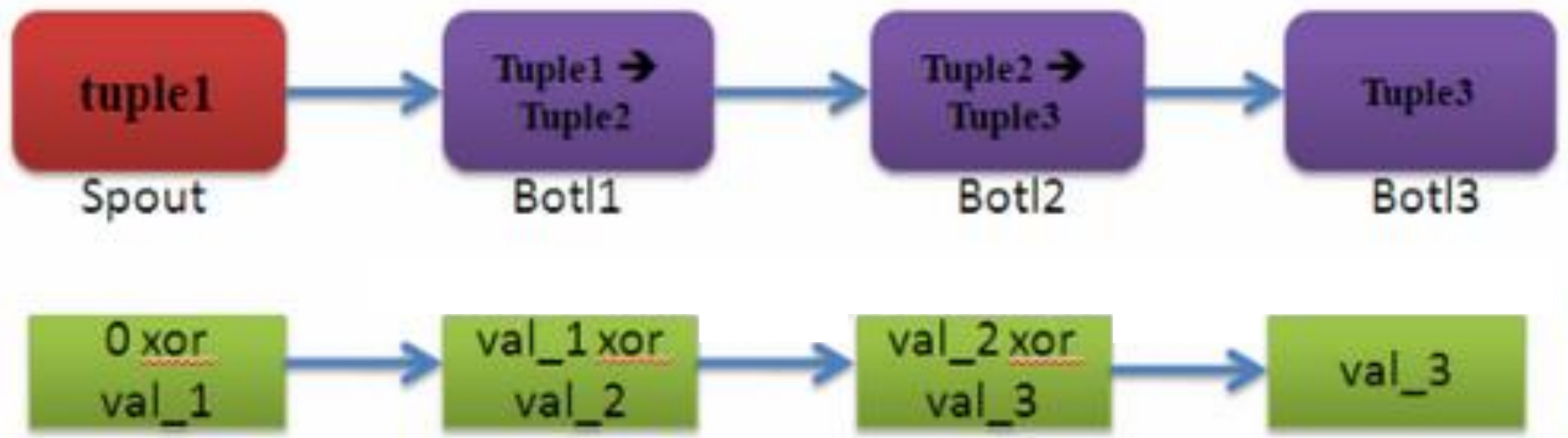
Storm容错——数据容错

- 实现
  - 内存超级大
    - Acker Task并不显式的跟踪tuple树。 对于那些有成千上万个节点的tuple树，把这么多的tuple信息都跟踪起来会耗费太多的内存。

Storm容错——数据容错

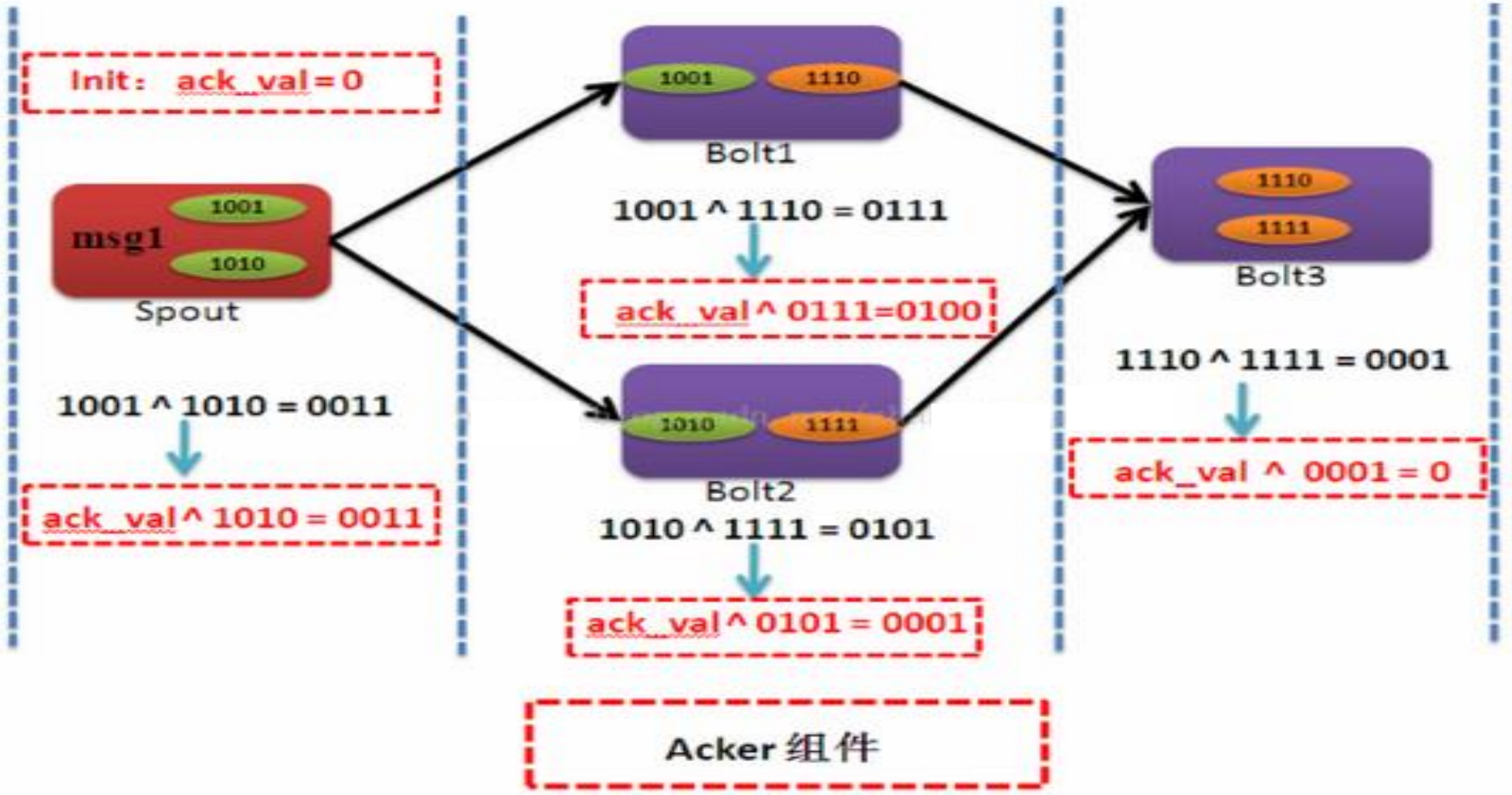
- 真实实现
  - 内存量是恒定的（20bytes）
  - 对于100万tuple，也才20M 左右
  - Taskid：ackval
  - Ackval所有创建的tupleid/ack的tuple一起异或
    - 一个acker task存储了一个spout-tuple-id到一对值的一个mapping。这个对子的第一个值是创建这个tuple的taskid， 这个是用来在完成处理tuple的时候发送消息用的。 第二个值是一个64位的数字称作：ack val，ack val是整个tuple树的状态的一个表示，不管这棵树多大。它只是简单地把这棵树上的所有创建的tupleid/ack的tupleid一起异或(XOR)。

Storm容错——数据容错



ark_val = ark_val xor val_1 xor val_1 xor val_2 xor val_2 xor val_3 xor val_3 = 0

Storm容错——数据容错



Storm容错——数据容错

- 一个tuple没有ack
  - 处理的task挂掉了，超时，重新处理
- Ack挂掉了
  - 一致性hash
  - 全挂了，超时，重新处理
- Spout挂掉了
  - 重新处理

Storm容错——数据容错

- Bolt
  - Anchoring
    - 将tuple作为一个锚点添加到原tuple上
  - Multi-anchoring
    - 如果tuple有两个原tuple，则为每个tuple添加一个锚点
  - Ack
    - 通知ack task，该tuple已被当前bolt成功消费
  - Fail
    - 通知ack task，该tuple已被消费失败

OutLine

- Storm基础
- Storm架构
- Storm容错
- Storm开发

一个简单的Topology

- 代码
  - TopologyBuilder builder = new TopologyBuilder();
  - builder.setSpout("words", new TestWordSpout(), 10);
  - builder.setBolt("exclaim1", new ExclamationBolt(), 3)
  - .shuffleGrouping("words");
  - builder.setBolt("exclaim2", new ExclamationBolt(), 2)
  - .shuffleGrouping("exclaim1");
- 这个拓扑包括一个spout和两个bolt。Spout发送单词。每个bolt在输入数据的尾部追加字符串“!!!”。三个节点排成一条线：spout发射给首个bolt，然后，这个bolt再发射给第二个bolt。如果spout发射元组“bob”和“john”，然后，第二个bolt将发射元组“bob!!!!!!”和“john!!!!!!”。

Spout实现

- 代码
  - public void nextTuple() {
  - Utils.sleep(100);
  - final String[] words = new String[] {"nathan", "mike", "jackson",
  - "golda", "bertels"};
  - final Random rand = new Random();
  - final String word = words[rand.nextInt(words.length)];
  - _collector.emit(new Values(word));
  - }
- Spout负责发送新消息到拓扑。拓扑中的TestWordSpout每隔0.1秒就从["nathan", "mike", "jackson", "golda", "bertels"]列表中随机选择一个单词，并把该单词作为一元元组发射出去。

Bolt实现

- 代码

```
public static class ExclamationBolt implements IRichBolt {
        OutputCollector _collector;
        public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
                _collector = collector;
        }
        public void execute(Tuple tuple) {
                _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
                _collector.ack(tuple);
        }
        public void cleanup() {
        }
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
                declarer.declare(new Fields("word"));
        }
}
```

Bolt实现

- prepare方法提供给bolt一个Outputcollector用来发射tuple。Bolt可以在任意时候发射tuple – 可以在prepare、execute、cleanup方法中发射, 或者甚至在另一个线程中异步发射。prepare方法只是简单地把OutputCollector作为一个类成员变量保存，以供execute方法以后使用。
- execute方法从bolt的一个输入流接收tuple。ExclamationBolt获取tuple的第一个字段（field），然后在值的尾部追加“!!!”作为一个新元组发射出去。如果一个bolt有多个输入源，你可以通过调用Tuple类的getSourceComponent方法找出tuple来自哪个输入源。
- 在execute方法中还有其它一些事情，即输入的tuple作为emit方法的第一个参数，在最后一行，输入的tuple被ack。这些是用于保证数据不会丢失的Storm可靠性API的一部分。

Bolt实现

- 当bolt关闭时，cleanup方法将被调用，它将清理所有已打开的资源。在集群中并不保证cleanup方法一定被调用。例如，如果正在运行task的机器突然down机，那么就没办法调用cleanup方法。Cleanup方法当初是为了在本地模式运行拓扑而设计（本地模式：在一个进程内模拟storm集群），你可以运行和杀掉一些topology，且不会有资源泄漏方面的问题。

- declareOutputFields方法声明ExclamationBolt发射只有一个“word”字段的一元元组

本地模式运行

- 本地模式运行ExclamationTopology的代码：
- Config conf = new Config();
- conf.setDebug(true);
- //conf.setNumWorkers(2);
- LocalCluster cluster = new LocalCluster();
- cluster.submitTopology("test", conf, builder.createTopology());

- 代码通过创建一个LocalCluster对象定义一个进程内集群（在进程内模拟集群）。提交拓扑到虚拟集群和提交拓扑到真正的分布式集群相同。通过调用submitTopology方法提交拓扑到LocalCluster，该方法需要三个参数：拓扑名称、拓扑的配置、拓扑自身。

- 拓扑名称用于标识拓扑，以便你以后你能kill它。拓扑将一直运行，直到你kill它。

常用配置

- **Config.TOPOLOGY_WORKERS:**
  - 这个设置用多少个工作进程来执行这个topology。比如，如果你把它设置成25， 那么集群里面一共会有25个java进程来执行这个topology的所有task。如果你的这个topology里面所有组件加起来一共有150的并行度，那么每个进程里面会有6个线程(150 / 25 = 6)。
- **Config.TOPOLOGY_ACKERS:**
  - 这个配置设置acker任务的并行度。默认的acker任务并行度为1，当系统中有大量的消息时，应该适当提高acker任务的并发度。设置为0，通过此方法，当Spout发送一个消息的时候，它的ack方法将立刻被调用；
- **Config.TOPOLOGY_MAX_SPOUT_PENDING:**
  - 这个设置一个spout task上面最多有多少个没有处理的tuple（没有ack/failed）回复， 我们推荐你设置这个配置，以防止tuple队列爆掉。
- **Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS:**
  - 这个配置storm的tuple的超时时间 – 超过这个时间的tuple被认为处理失败了。这个设置的默认设置是30秒

Storm环境

wget http://www.python.org/ftp/python/2.6.6/Python-2.6.6.tar.bz2

- Conf/storm.yaml
  - storm.zookeeper.servers
    - 多个Zookeeper服务器
  - Storm.local.dir
    - Storm用于存储jar包和临时文件的本地存储目录
  - Java.library.path
  - Nimbus.host
  - Supervisor.slots.ports
    - 几个port，就几个worker
  - Ui.port

Storm环境

- bin/storm nimbus >/dev/null 2>&1 &
- bin/storm supervisor >/dev/null 2>&1 &
- bin/storm ui >/dev/null 2>&1 &
  - http://{nimbus host}:port
- ./logs
  - 查看错误日志

Q & A

@八斗学院