



第七章

分布式批处理系统

第七章、分布式批处理系统

01

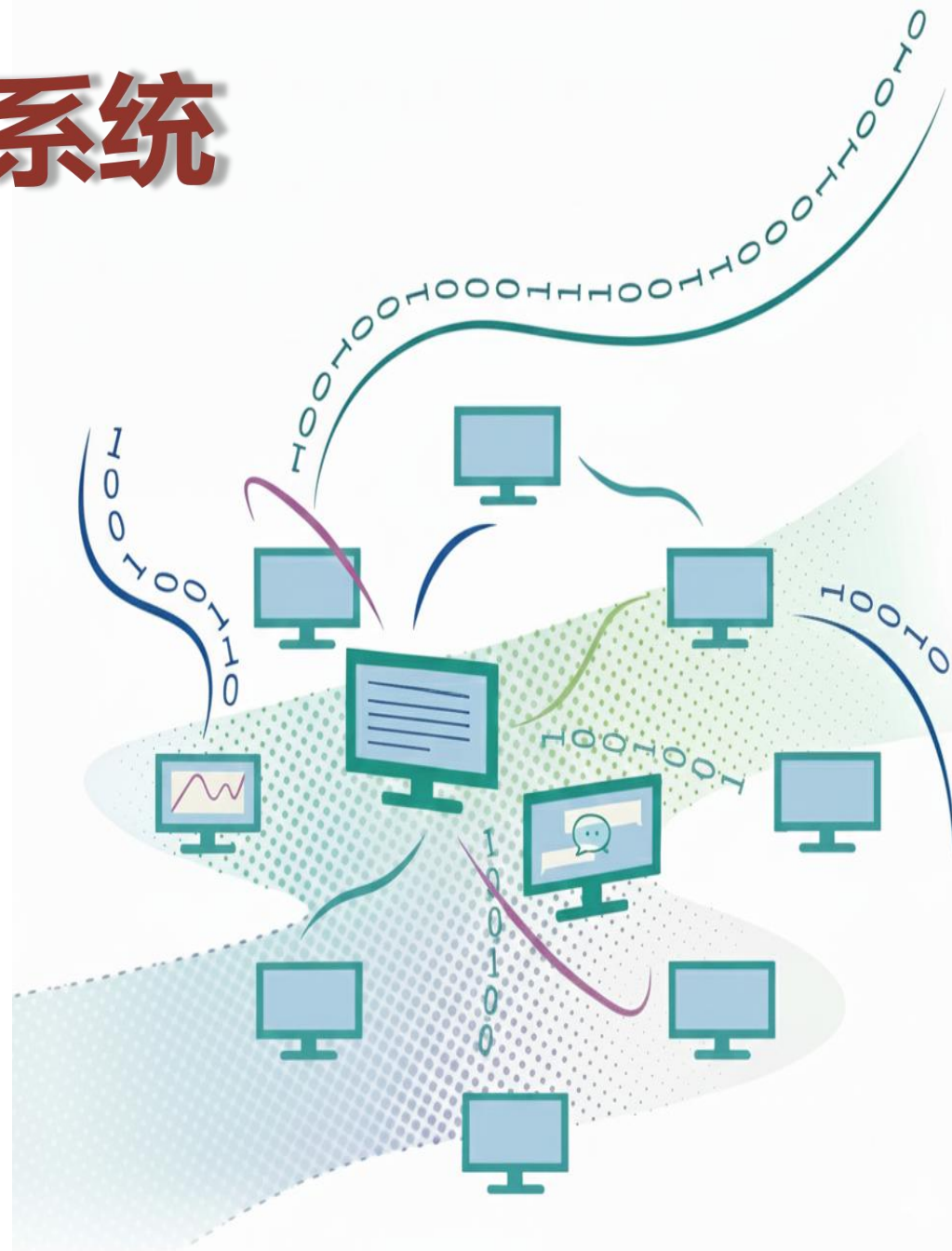
7.1 任务分配与协调

02

7.2 计算故障检测与恢复

03

7.3 批处理系统-MapReduce



第7章 分布式批处理系统

- 利用分布式集群中的计算节点**处理数据量或计算量大、耗时长**的计算任务
 - 整个批处理任务分解为**多个子任务**，分配给**不同的计算节点**执行
 - 根据任务之间的依赖关系、系统负载均衡等进行任务协调
 - 汇总各个节点的结果



7.1 任务分配与协调

任务分配 task allocation

- 如何有效地将任务下发到各个节点
- 在任务开始前，分解整个批处理任务为较小的子任务，并分布到多个计算节点



7.1 任务分配与协调

任务协调 task coordination

- 如何管理这些任务的执行，以保持整个系统的高效运作
- 在任务执行过程中，管理和调节各个节点之间的工作关系和数据流



7.1.1 任务分配

如何将计算任务分配给不同的计算节点



负载均衡

系统资源得到充分利用

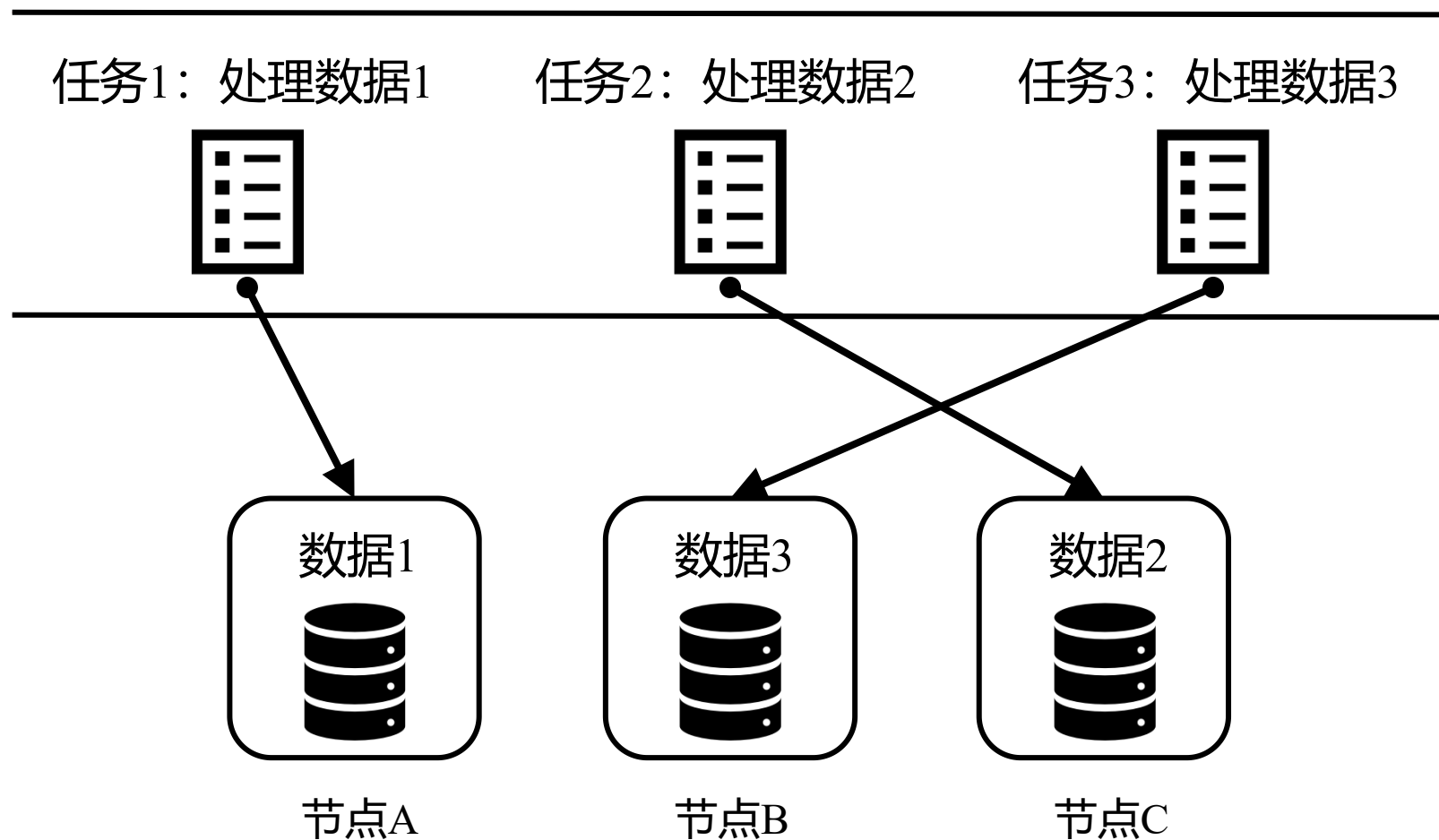
避免某些节点过载而导致性能下降

- ✓ 近数据任务分配
- ✓ 轮询分配
- ✓ 主导资源公平（Dominant Resource Fair, DRF）算法
- ✓ 最大最小公平（Max-Min Fairness）算法

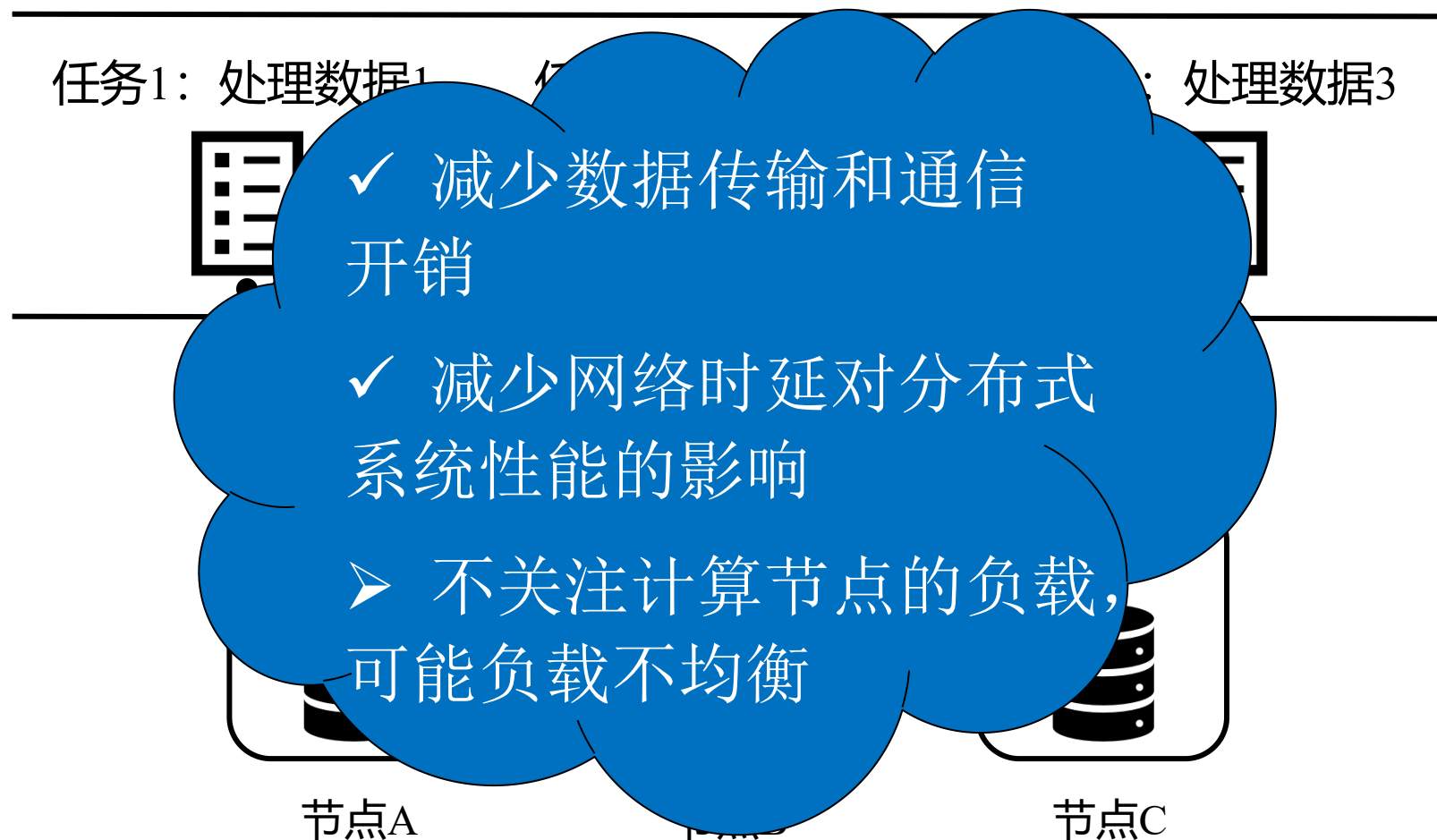
近数据任务分配

- 根据任务所需要数据的位置和分布情况进行任务分配
- 将任务分配给与相关数据位置相近或者已经缓存了相关数据的计算节点
- 目的：减少数据传输和通信开销
- 典型系统：MapReduce

近数据任务分配



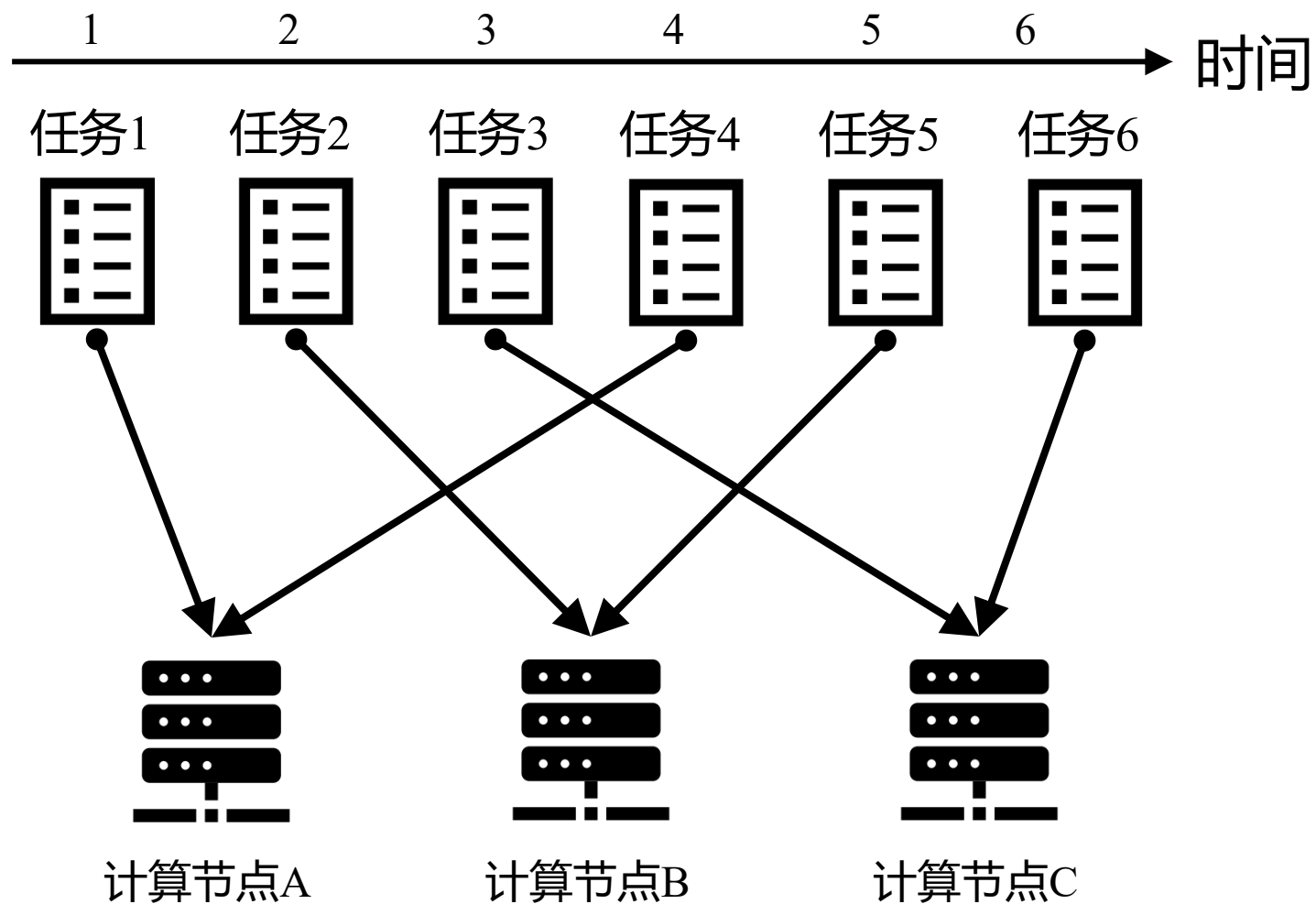
近数据任务分配



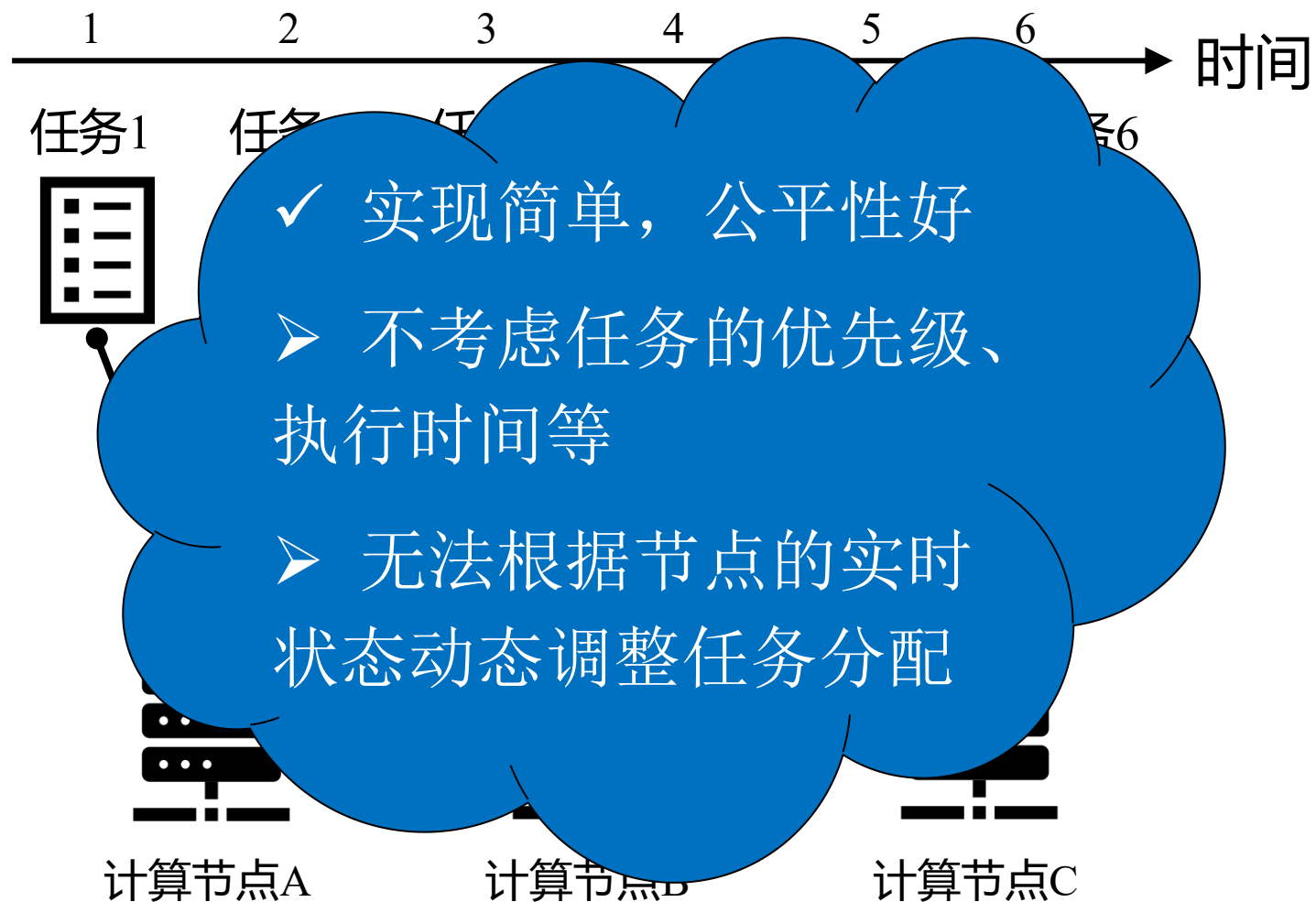
轮询分配

- 按照某个固定的顺序依次将任务分配给可用的节点
- 公平性，均匀分配
 - 确定可用于任务执行的节点列表（物理机器、虚拟机或容器）
 - 有新的任务需要分配时，分配器按照轮询顺序将任务分配给下一个计算节点
 - 每个节点依次接收任务，直到轮询一遍所有节点

轮询分配

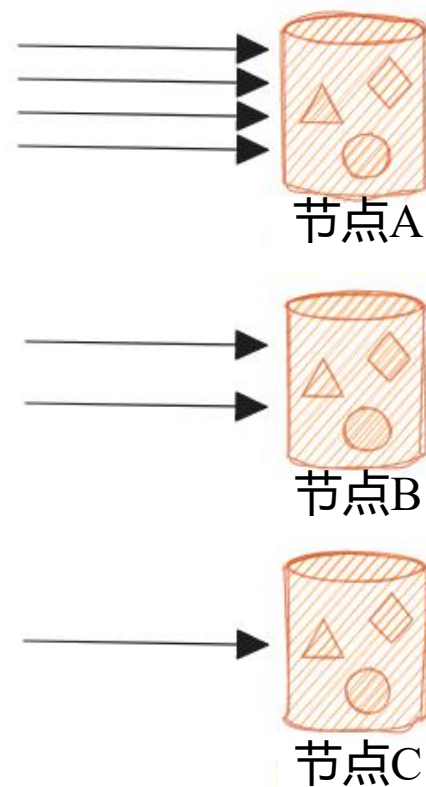
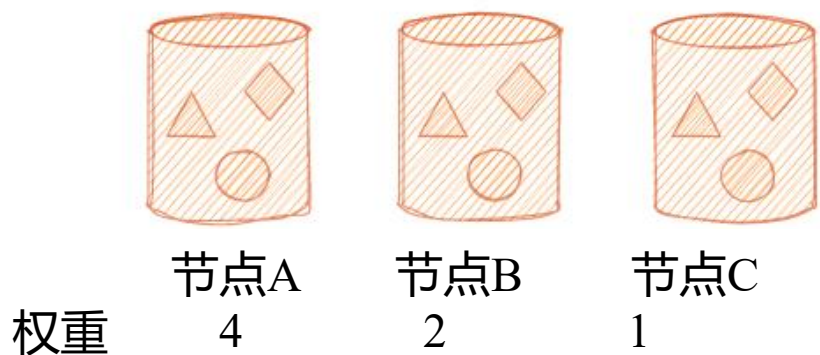


轮询分配



加权轮询分配

- 每个计算节点被分配一个权重值，代表该节点处理能力的相对大小
- 较高权重值的节点将被分配更多的任务



加权轮询分配



算法思想：

- 轮询所有节点，寻找权重最大节点
- 选中节点，然后将权重减1
- 当所有节点权重都为0时，重置权重

序号	轮前权重	转移节点	轮后权重
1	A(4), B(2), C(1)	A	A(3), B(2), C(1)
2	A(3), B(2), C(1)	A	A(2), B(2), C(1)
3	A(2), B(2), C(1)	A	A(1), B(2), C(1)
4	A(1), B(2), C(1)	B	A(1), B(1), C(1)
5	A(1), B(1), C(1)	A	A(0), B(1), C(1)
6	A(0), B(1), C(1)	B	A(0), B(0), C(1)
7	A(0), B(0), C(1)	C	A(0), B(0), C(0)

加权轮询分配



算法思想：

- 轮询所有节点，寻找权重最大的节点
- 选中节点，然后按照权重分配请求
- 当所有节点权重都为0时，重新轮询

- 节点A权重为4，那么前2个请求一定会落在节点A上
- 权重大的机器压力过大，权重小的机器C一直在空闲（能力小，不代表一个请求都不能处理）
- 假如权重为{A:10, B:1, C:1}，会放大这一问题

序号	轮前	轮后	轮后
1	A(4), B(1), C(1)	A(4), B(1), C(1)	A(4), B(1), C(1)
2	A(3), B(2), C(1)	A(3), B(2), C(1)	A(3), B(2), C(1)
3	A(2), B(2), C(1)	A(2), B(2), C(1)	A(2), B(2), C(1)
4	A(1), B(2), C(1)	A(1), B(2), C(1)	A(1), B(2), C(1)
5	A(1), B(1), C(1)	A(1), B(1), C(1)	A(1), B(1), C(1)
6	A(0), B(1), C(1)	B(1), A(0), C(1)	A(0), B(0), C(1)
7	A(0), B(0), C(1)	C(1), A(0), B(0)	A(0), B(0), C(0)

平滑加权轮询分配

- 计算总权重 $W=W_A+W_B+W_C$
- 选择权重最大的节点作为转移节点，将其权重变为：轮前权重+原始权重- W ；
- 其余节点：轮前权重+原始权重

序号	轮前权重	转移节点	轮后权重
1	A(4), B(2), C(1)	A	A(1), B(4), C(2)
2	A(1), B(4), C(2)	B	A(5), B(-1), C(3)
3	A(5), B(-1), C(3)	A	A(2), B(1), C(4)
4	A(2), B(1), C(4)	C	A(6), B(3), C(-2)
5	A(6), B(3), C(-2)	A	A(3), B(5), C(-1)
6	A(3), B(5), C(-1)	B	A(7), B(0), C(0)
7	A(7), B(0), C(0)	A	A(4), B(2), C(1)

平滑加权轮询分配

- 计算**总权重** $W=W_A+W_B+W_C$
- 选择权重最大的节点作为转移节点，将其权重变为：**轮前权重+原始权重- W** ；
- 其余节点：**轮前权重+原始权重**

序号	轮前权重	转移节点	轮后权重
1	A(4), B(2), C(1)	A	A(1), B(4), C(2)
2	A(1), B(4), C(2)	B	A(2), B(1), C(4)
3	A(5), B(-1), C(3)	A	A(6), B(3), C(-2)
4	A(2), B(1), C(4)	B	A(3), B(5), C(-1)
5	A(6), B(3), C(-2)	A	A(7), B(0), C(0)
6	A(3), B(5), C(-1)	B	A(4), B(2), C(1)
7	A(7), B(0), C(0)	A	

每次选择后权重的
动态调整，保证了
服务分配的平滑性，
实现了相对公平

主导资源公平 DRF 算法

- DRF (Dominant Resource Fair)
- 在多租户的集群系统中实现多种资源类型的公平共享
- 一个系统共9个CPU和18GB内存，任务A需要分配1个CPU和4GB内存，因此其CPU占比为 $\frac{1}{9}$ ，内存占比为 $\frac{2}{9}$ 。因此A的**主导资源**是内存，**主导资源占比**是 $\frac{2}{9}$ 。
- DRF算法将资源按照主导资源占比大小进行排序，并将资源分配给具有**最小主导资源占比**的任务



DRF 算法

	CPU (个)	内存 (GB)
系统资源	9	18
用户A每个任务需要	1	4
用户B每个任务需要	3	1

- 首先，随机选择一个任务进行资源分配，于是B的主导资源占比变为 $\frac{1}{3}$
- 此时A的主导资源占比为0，更小，于是将资源分配给A。
- 此时A的主导资源占比为 $\frac{2}{9}$ ，B的主导资源占比为 $\frac{1}{3}$ ，A更小，于是继续将资源分配给A

■

调度	用户A		用户B		CPU分配占比	RAM分配占比
	资源占比	主导资源占比	资源占比	主导资源占比		
用户B	(0, 0)	0	(3/9, 1/18)	1/3	3/9	1/18
用户A	(1/9, 4/18)	2/9	(3/9, 1/18)	1/3	4/9	5/18
用户A	(2/9, 8/18)	4/9	(3/9, 1/18)	1/3	5/9	9/18
用户B	(2/9, 8/18)	4/9	(6/9, 2/18)	2/3	8/9	10/18
用户A	(3/9, 12/18)	2/3	(6/9, 2/18)	2/3	1	14/18

- 最终，A和B的主导资源占比都为 $\frac{2}{3}$ ，CPU资源饱和

局限性

- **复杂性**：需要考虑各个资源的需求和剩余量，并对任务进行多维度的比较和排序。
- **通信开销大**：需要获取和比较各个任务的资源需求和剩余量信息，这可能需要在不同的计算节点之间进行通信和信息交换。
- **不考虑任务优先级**：DRF算法主要关注资源的公平分配，而不考虑任务的优先级或紧迫程度。这可能导致某些紧急任务或高优先级任务无法及时获得足够的资源，影响系统的响应性和性能。

最大最小公平算法

每个任务至少获得其**最低需求**的资源量，并尽可能地使资源分配达到最大化。

首先把资源平均分给各个任务。如果有资源超过任务需求，则没用到的部分在其他任务之间再平均划分

	A	B	C	D	
资源需求	2	3	4	5	
平分资源	2.5	2.5	2.5	2.5	$10/4=2.5$
平分A多余资源	2	2.66	2.66	2.66	$(2.5-2)/3+2.5=2.66$

最大最小公平算法

每个任务至少获得其**最低需求**，资源分配达到最大化。

首先把资源平均分给各任务之间再平均划分

- ✓ 每个参与者都至少能获得其所需的最小份额
- ✓ 主要目标：公平性
- 可能会无法满足任何单项任务的最低要求，从而阻止所有任务的执行，导致资源使用效率低下。
- 适用于单一类型的资源分配

资源需求

平分资源

2.5	2.5	2.5
-----	-----	-----

$10/4=2.5$

平分A多余资源

2	2.66	2.66	2.66
---	------	------	------

$(2.5-2)/3+2.5=2.66$

最大最小公平算法

每个任务至少获得其**最低需求**，资源分配达到最大化。

首先把资源平均分给各任务之间再平均划分

- ✓ 每个参与者都至少能获得其所需的最小份额
- ✓ 主要目标：公平性
- 可能会无法满足任何单项任务的最低要求，从而阻止所有任务的执行，导致资源使用效率低下。

➤ 适用于单一类型的资源分配

资源需求

平分资源

2.5	2.5	2.5
-----	-----	-----

$10/4=2.5$

平分A多余资源

2	2.66	2.66	2.66
---	------	------	------

$(2.5-2)/3+2.5=2.66$

加权最大最小公平算法

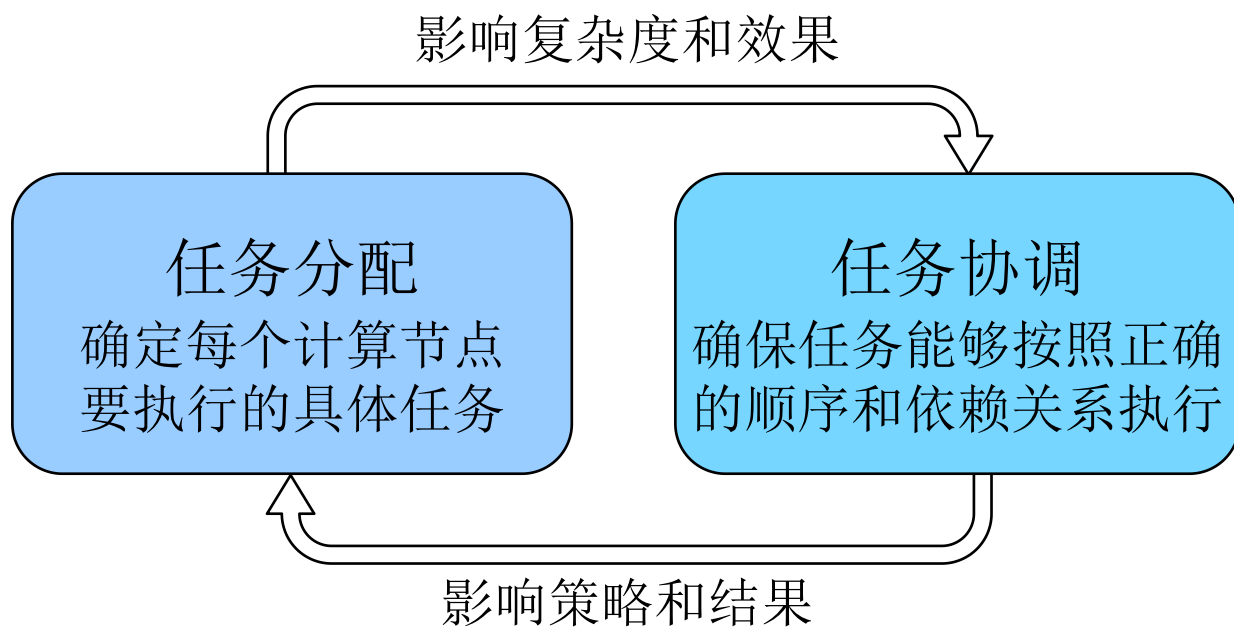
7.1.1 任务分配

总结

- ✓ 近数据任务分配
- ✓ 轮询分配
- ✓ DRF (Dominant Resource Fair) 算法
- ✓ 最大最小公平 (Max-Min Fairness) 算法
- 任务分配的目的:
 - 每个计算节点处理的任务量相对均衡
- 任务分配的过程
 - 任务分割
 - 资源评估和选择
 - 任务的实际分配

7.1.2 任务协调

- 任务协调：在分布式计算环境中，协调不同计算节点上的任务执行和数据传输，确保任务按照预定的顺序和依赖关系进行执行。



7.1.2 任务协调

■ 目标

- 保证任务的正确性和一致性
- 避免冲突和数据竞争

■ 过程

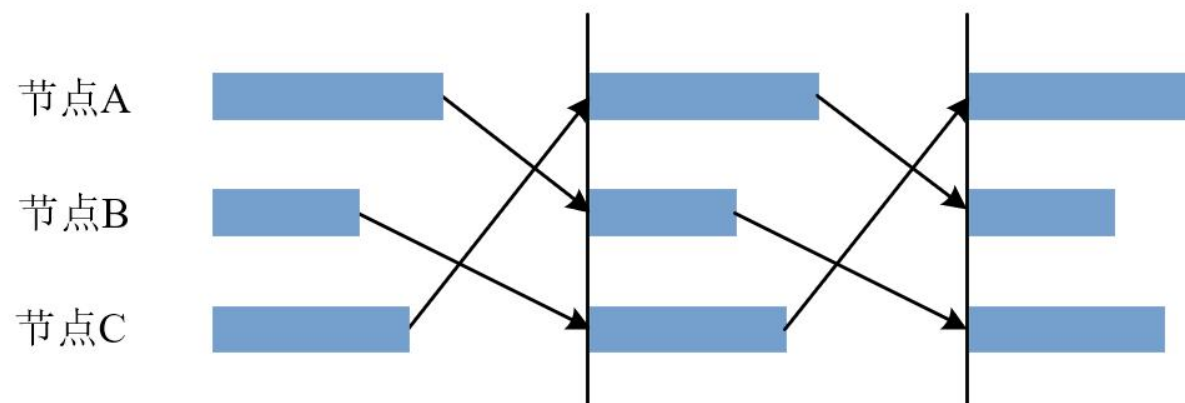
- 任务状态的监控
- 任务之间的通信和同步
- 数据共享和传输

- ✓ 同步计算和异步计算
- ✓ 计算图机制（静态计算图和动态计算图）
- ✓ 预测执行机制

同步计算与异步计算

同步计算 Synchronous Computation

- 在计算过程中，各个计算节点必须按照同步的步调进行计算，并在每个步骤结束后进行数据交换和同步。
 - 将整个计算过程划分为一系列同步步骤
 - 同步通信机制
 - 各个节点按照同样的顺序执行计算



同步计算与异步计算

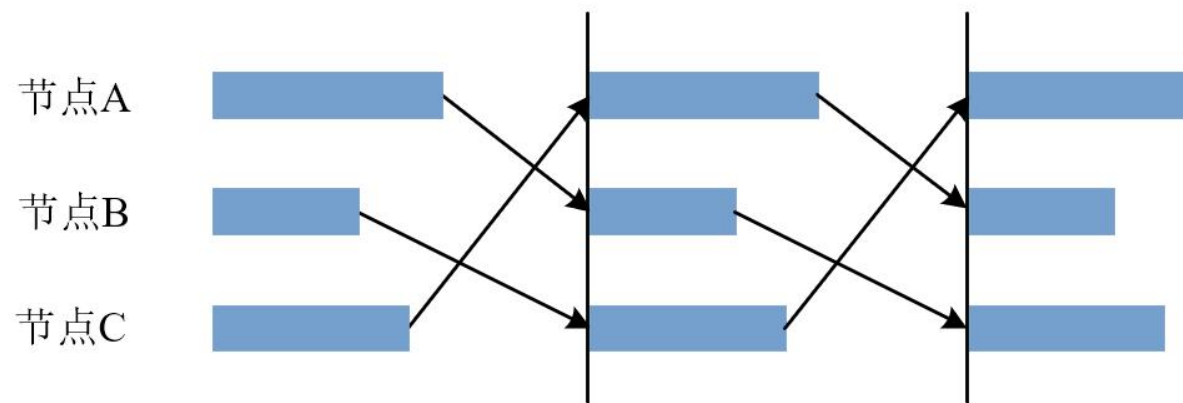
同步计算 Synchronous Computation

■ 优势

- 易于理解和调试
- 可预测性
- 数据一致性

■ 不足

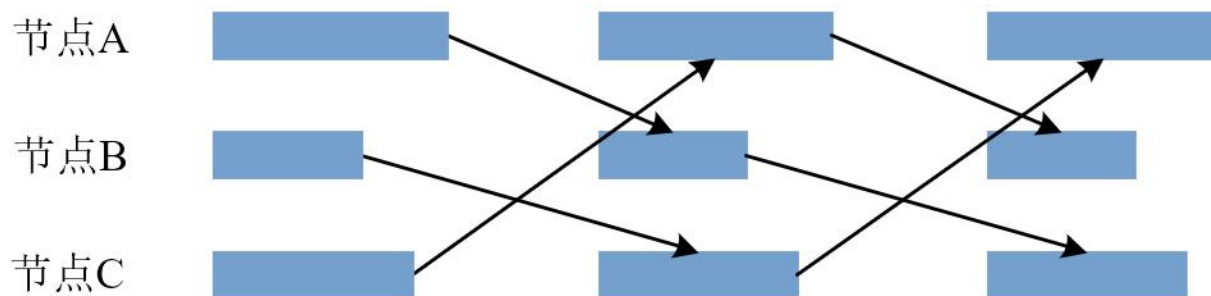
- 计算效率低
- 短板效应
- 扩展性差



同步计算与异步计算

异步计算 Asynchronous Computation

- 任务以非固定的顺序和时间步骤进行执行，任务之间不需要相互等待和协调，而是根据自己的进度和可用资源来决定执行顺序。
 - 计算节点接收到任务，独立地开始执行计算
 - 非阻塞通信
 - 可以设置同步点



同步计算与异步计算

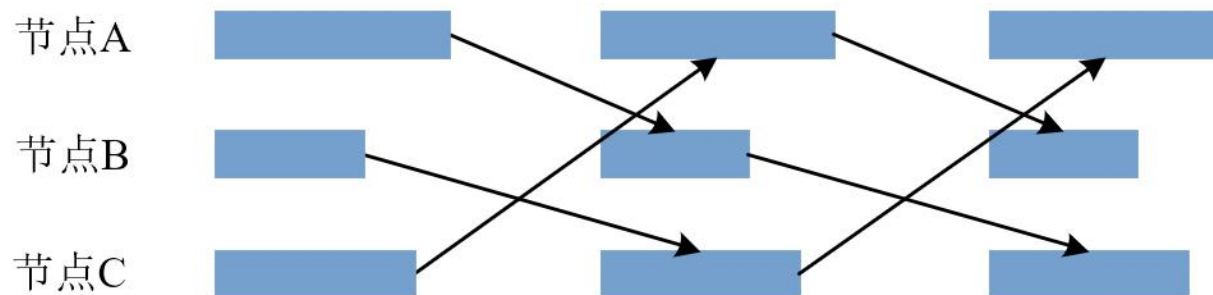
异步计算 Asynchronous Computation

■ 优势

- 易高计算效率
- 无空闲等待

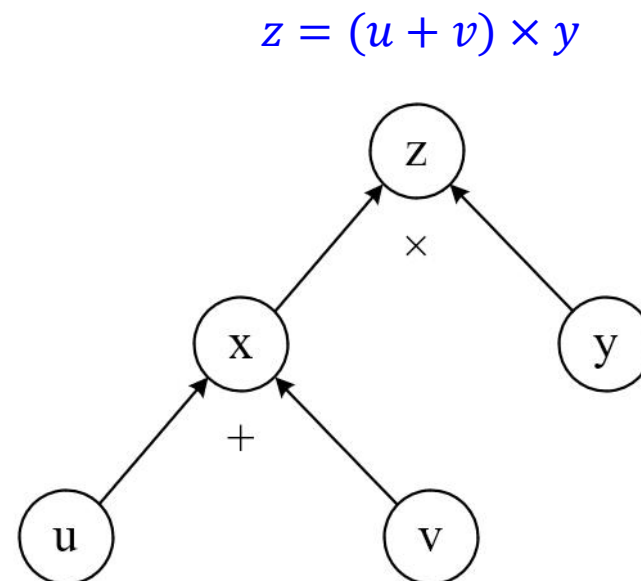
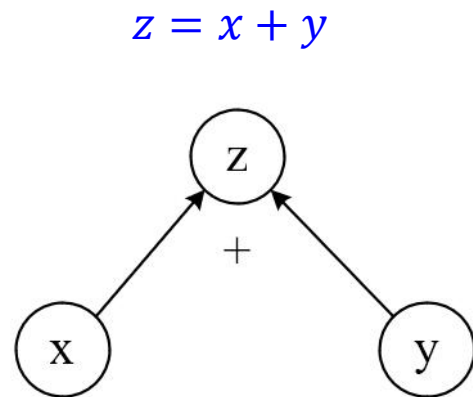
■ 不足

- 数据冲突
- 结果正确性



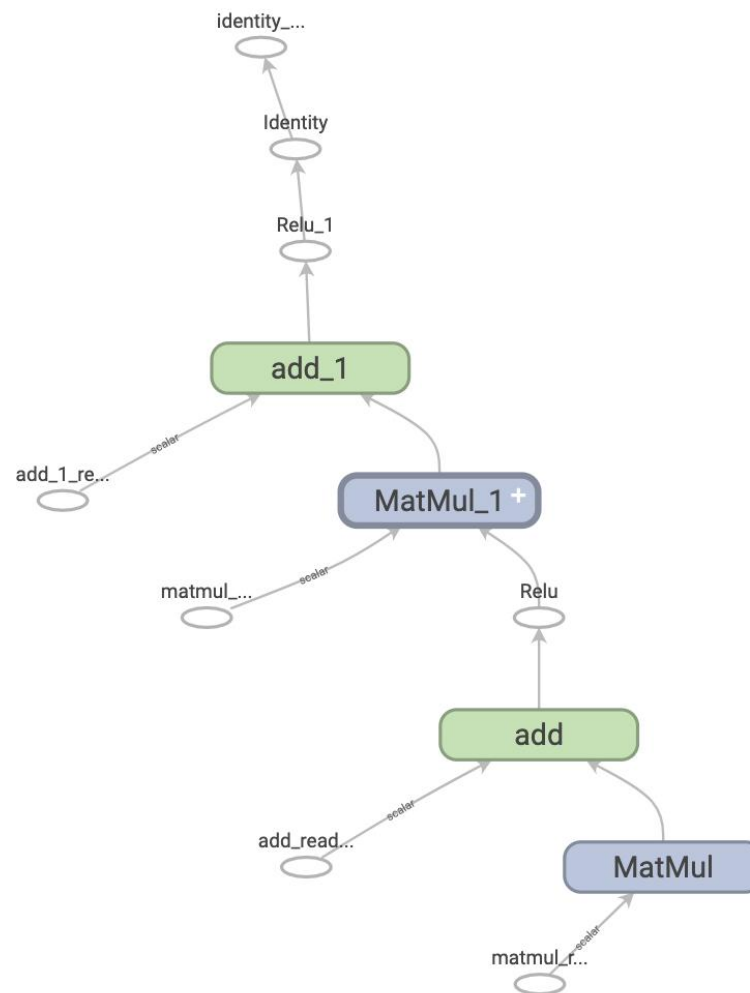
计算图 Computational Graph

- 表示计算任务和它们之间的依赖关系
- 将计算任务分解为更小的可并行执行单元
- DAG



静态计算图

- 在程序运行之前就被定义并优化的计算图。一旦定义，在整个程序运行期间不会改变。
- 先构建计算图，然后进行计算
- TensorFlow（新版本支持动态图）、Theano



表示两层神经网络的TensorFlow计算图
在TensorBoard中呈现的样子

静态计算图

■ 优势

- 在模型训练之前，可以对计算图进行深度优化
- 计算过程中执行效率更高
- 减少运行时的不确定性和错误，适用于大规模部署

■ 不足

- 对于需要条件执行或循环结构的模型，静态图可能难以实现或不够灵活
- 调试和修改图结构可能较为复杂

动态计算图

- 也称为**即时执行模型**，是在程序运行时构建和修改的。图的结构可以根据输入数据动态变化，每次执行可能都会创建新的计算图
- 计算图的构建与计算同时进行
- PyTorch、JAX（支持动态图特性）、TensorFlow的Eager Execution

例子：Pytorch动态生成计算图

A graph is created on the fly

W_h h W_x x

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```



动态计算图

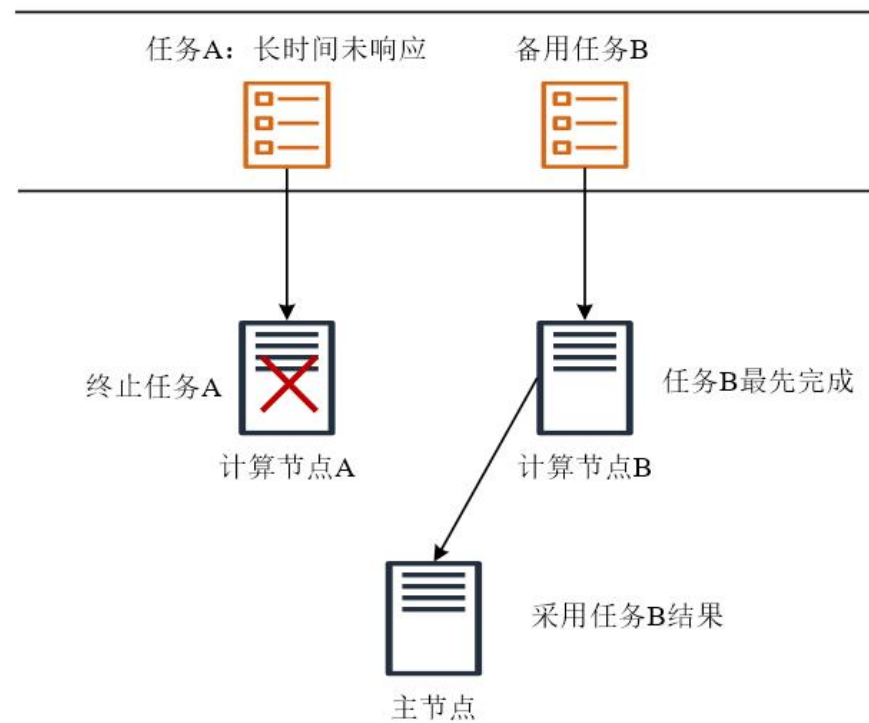
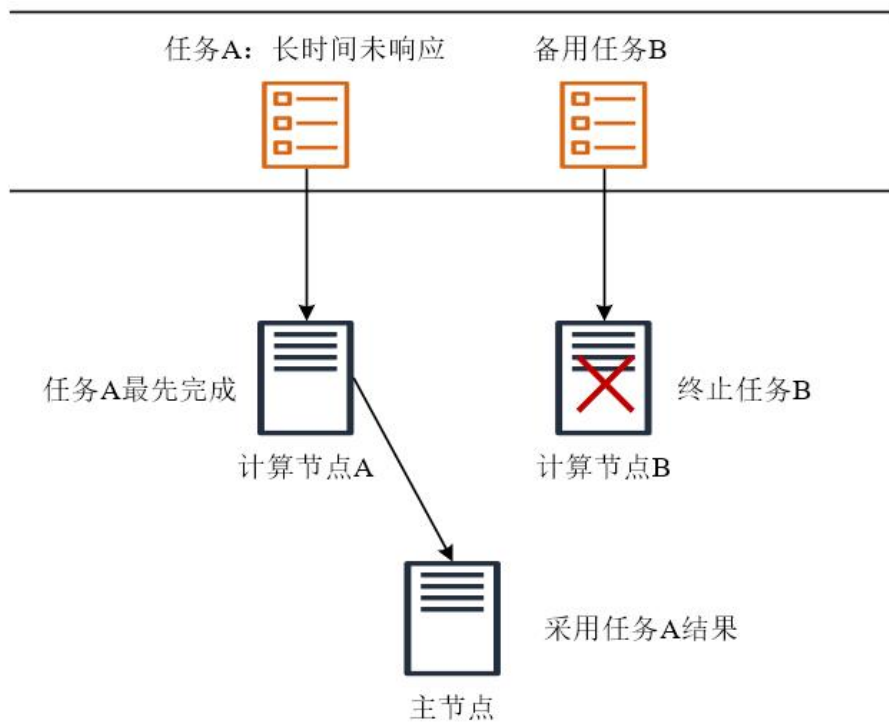
- 优势
 - 可以根据需要动态改变计算图（适合循环等逻辑）
 - 可以像普通Python代码一样进行逐行调试
 - 更直观，代码更易于编写和理解
- 不足
 - 缺乏深度优化，可能在执行效率上有所牺牲
 - 每次执行可能需要重新构建图，会增加计算资源消耗

7.1.3 预测执行机制

- 系统预测任务的执行情况，当一个任务执行较慢时，系统预测到任务可能无法成功完成，就在另一个节点上再启动一个相同的备用任务，并使用相同的数据，最后采纳最先完成的任务。

- 在Spark中任务会以计算图的方式并行执行，每个节点并行运行在不同的Executor中。有的任务执行很快，有的任务执行很慢（由于网络抖动、性能不同、数据倾斜等）。
- 很慢的任务会成为整个任务的瓶颈，此时可以触发预测执行，为长时间的task重新启动一个task，哪个先完成就使用哪个的结果，并kill掉另一个task。

7.1.3 预测执行机制



7.1.3 预测执行机制

■ 优势

- 提高系统的容错性，发生节点故障时可以进行转移；
- 当某个任务进度较慢时，备用任务可以在空闲的资源上执行，提高资源利用率；
- 提高作业的执行速度。

■ 不足

- 启动和执行备用任务需要一定的计算和存储资源；
- 不完全适用于所有类型的作业和任务，例如当某些任务具有严格的依赖关系时，或者网络延迟较高时；
- 可能引入不确定性。

第七章、分布式批处理系统

01

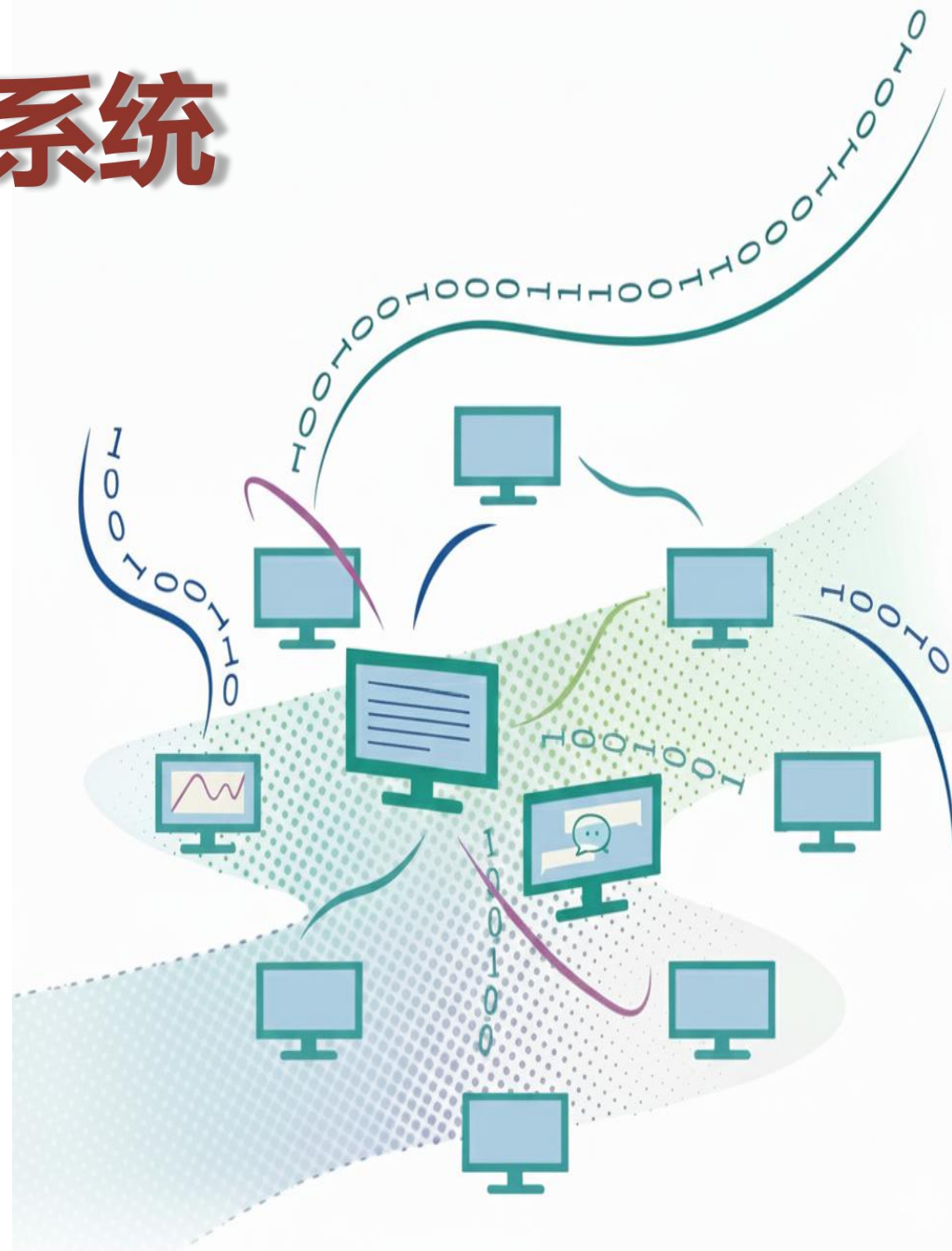
7.1 任务分配与协调

02

7.2 计算故障检测与恢复

03

7.3 批处理系统-MapReduce



7.2 计算故障检测与恢复

- 计算故障检测：分布式批处理系统监测子任务的运行状态，并在发现异常时触发相应的故障处理机制。

周期性的健康检查、心跳检测

- 故障恢复：发现系统组件或子任务执行失败后，系统采取一系列措施使系统尽快恢复到正常运行状态。

重新分配任务、切换到备用节点、重新启动故障节点、使用冗余数据

7.2.1 故障检测

- 分布批处理由大量计算节点及其连接计算节点的网络构成，分布式系统的规模越大，出现故障的概率就越大。
- 为了使系统能够以适当的方式解决各种故障，分布式系统需要及时地检测出这些故障。
 - ✓ 心跳检测
 - ✓ Gossip风格的故障检测

心跳检测

- 心跳：一种特殊的消息，用于表示节点的存活状态和运行情况。
- 心跳的发送者和接受者取决于故障检测的目的



目的1：让管理节点知道系统中哪些节点活跃

- 发送方-系统中的每个节点，接收方-管理节点；
- 每个节点定期向管理节点发送心跳消息，管理节点根据收到的心跳消息更新节点列表。

目的2：让每个节点知道自己的邻居节点是否可用

- 发送方、接收方-都是系统中的每个节点；
- 每个节点根据收发的心跳消息判断邻居节点的状态。

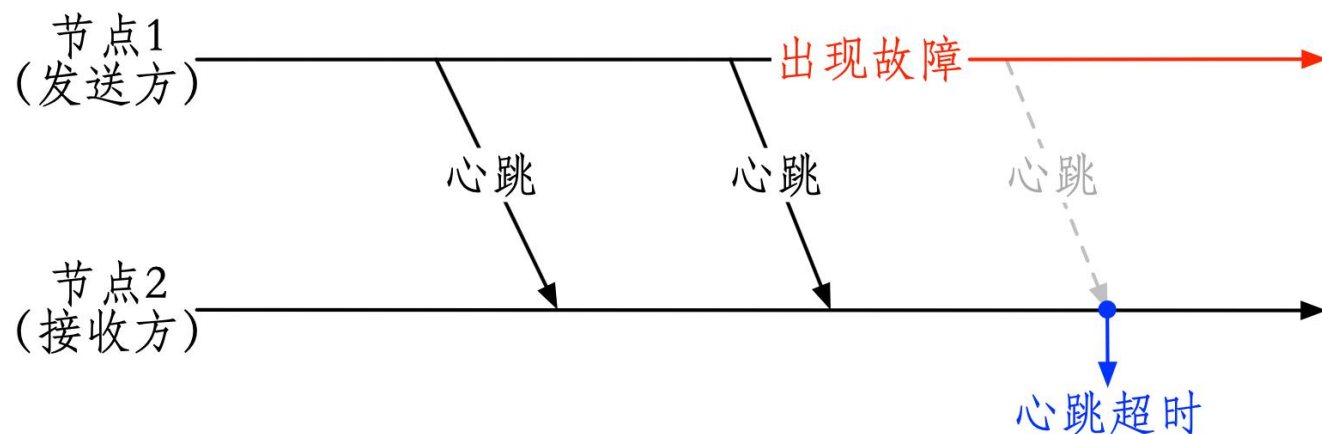
心跳检测

- 心跳：一种特殊的消息，用于表示节点的存活状态和运行情况。
- 心跳的发送者和接受者取决于故障检测的目的
- 心跳包括什么？
 - 通常是轻量级、低开销的信号或数据包，传达发送方的存在状态
 - 可能是周期性的“我还活着”信号
 - 可能是包含了发送方状态或能力的数据包，例如CPU使用率、内存占用、网络延迟、服务负载等

心跳检测

固定心跳检测

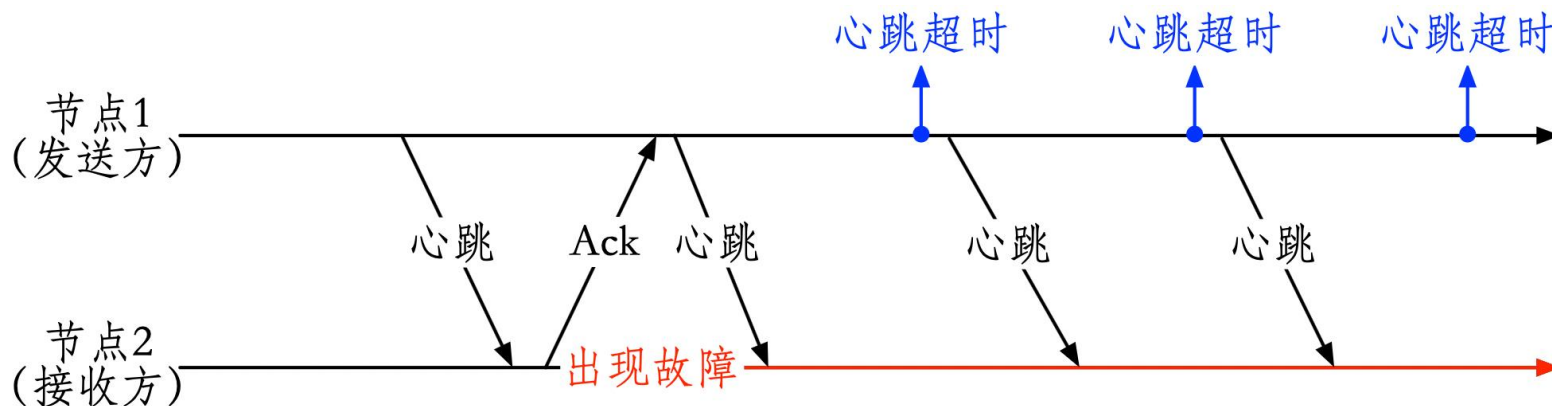
- 在规定的时间内，每个心跳发送方给接收方发送心跳消息
- **单向检测**：接收方收到来自发送方的心跳消息时，它会更新发送方的最后一次心跳时间，以确认发送方仍然存活



心跳检测

固定心跳检测

- 在规定的时间内，每个心跳发送方给接收方发送心跳消息
- **双向检测**：接收方收到来自发送方的心跳消息时，它会回复一个心跳确认消息（Ack），以确认发送方和自己都仍然存活



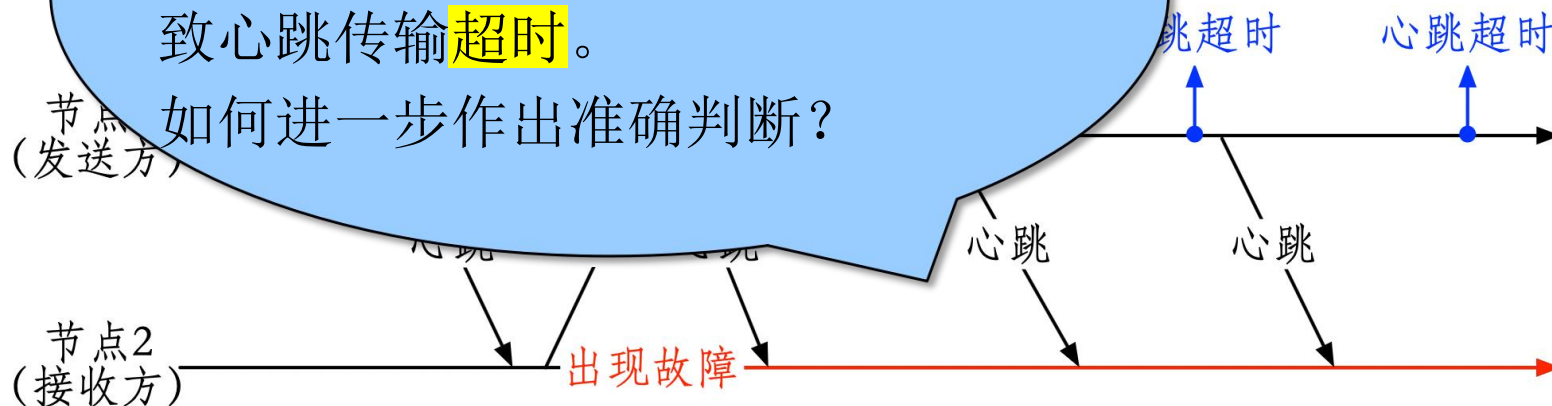
心跳检测

固定心跳检测

- 在规定的时间内，每个心跳发送方给接收方发送心跳消息
- **双向检测**：接收方收到心跳消息时，它会回复一个心跳确认消息

(Ack)，但并不一定意味着节点2本身发生了故障，也可能是节点2服务繁忙导致心跳传输**超时**。

如何进一步作出准确判断？



心跳检测

固定心跳检测

周期性心跳检测机制

- 假设心跳发送方固定周期 T 秒发送心跳，如果接收方连续 k 次没有收到心跳，或者发送方连续 k 次没有收到确认消息，那么就发送方或接收方说明已经发生了故障，需要采取相应的故障恢复措施。
- 对于要求秒级故障检测的场景（时延敏感性场景），则 $k * T \leq 1s$ ，将 T 设置为ms级，例如200ms， k 设置为 $1000/200=5$ 次。

心跳检测

固定心跳检测

累计失效检测机制

- 在周期性心跳检测机制的基础上，统计一定周期内节点的返回情况，包括超时返回和正确返回。
- 计算 失效率 = $\frac{\text{超时返回次数}}{\text{总返回次数}}$
- 判定为可能失效的节点，可以发起有限次的重试以进一步判断节点是否发生故障。如果超过重试次数仍然没有收到确认消息，则断定该节点已发生故障。

心跳检测

根据历史心跳信息的故障预测

- 又称 ϕ 值故障检测， ϕ 值用于评估心跳超时的概率。如果 ϕ 值很高，意味着心跳比正常情况延迟得多。
 1. **数据收集**：收集窗口期内每次心跳到达时间；
 2. **建模**：假设心跳到达时间间隔的分布遵循正态分布
 3. **实时监测**：继续监测新的心跳到达时间；
 4. **计算 ϕ 值**：利用统计方法计算新数据与模型期望之间的偏差，用 ϕ 值量化偏差；
 5. **故障识别**：当 ϕ 值超过预设的阈值时，判定为系统出现故障；

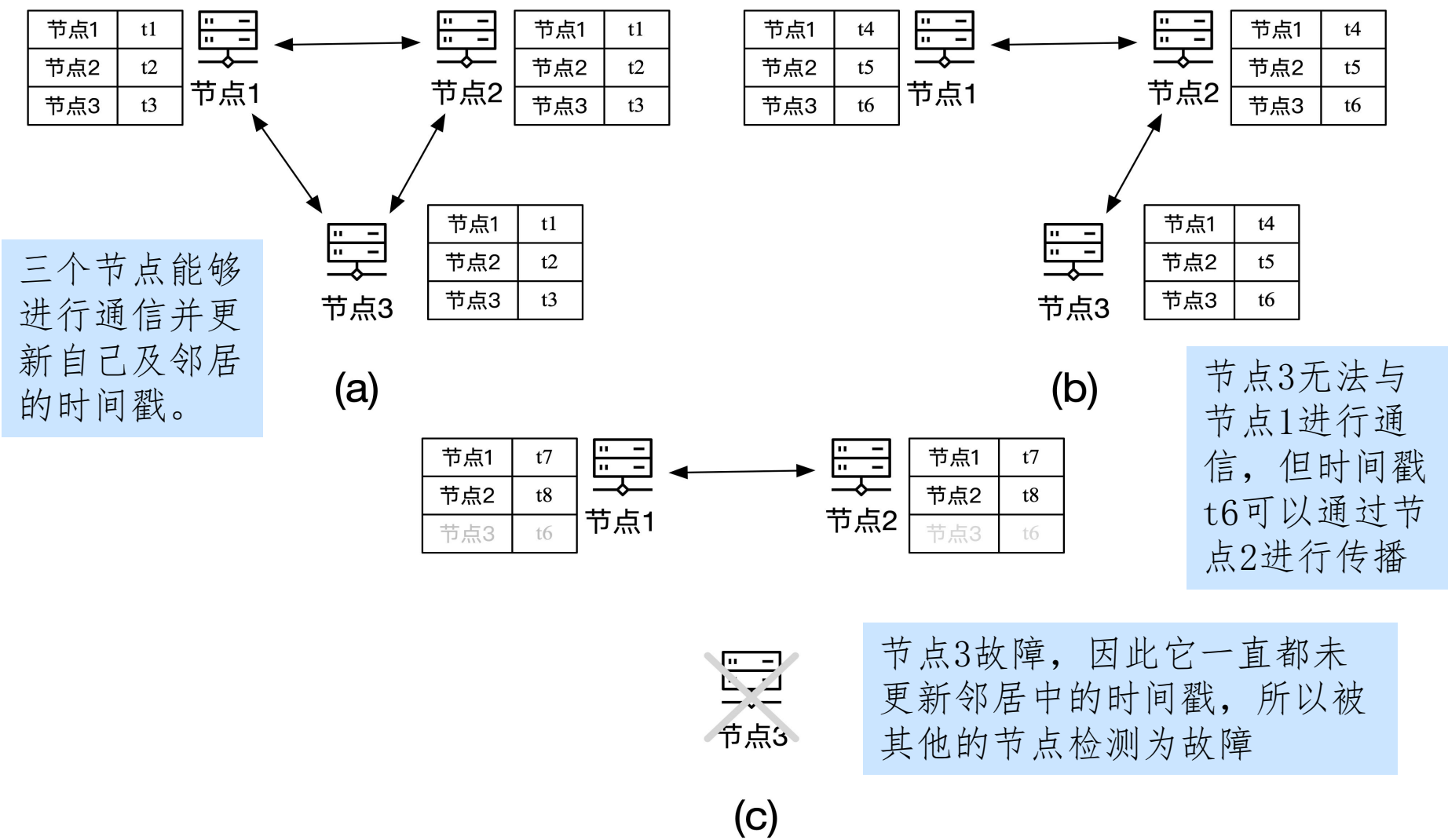
根据历史心跳信息的故障预测

- 又称 ϕ 值故障检测， ϕ 值用于评估心跳超时的概率。如果 ϕ 值很高，意味着心跳比正常情况延迟得多。
 - ϕ 值故障检测策略适用于心跳间隔比较平稳或符合规律的情况，对于具有突发情况或心跳间隔无规律的场景误判率比较高。
 - ϕ 值故障检测在需要高可靠性的分布式系统中非常有用，例如，在大规模分布式数据库、云计算平台和微服务架构中监测服务的健康状况。

Gossip风格的故障检测

- Gossip算法也称为“流言算法” “传染病模型”，通过节点之间简单且频繁的消息交换来传播和更新关于系统状态的信息。
 1. **随机通信**：每个节点周期性地与其他随机选定的节点交换信息（类似于人们在社交中随机交谈）；
 2. **状态信息的交换**：每次通信时，节点会交换它们所知道的关于系统中每个节点的最新状态信息，比如哪些节点是活跃的、哪些可能已经故障；
 3. **状态更新**：接收信息的节点会根据收到的信息更新自己的状态表。
 4. **传播与收敛**：随着时间的推移和信息的不断交换，系统中的状态信息会逐渐传播至所有节点，并最终达到一致。

Gossip风格的故障检测



Gossip风格的故障检测

- **可扩展性**: Gossip算法不需要中央控制节点, 适合大规模分布式系统。
- **容错性**: 算法可以容忍网络中的部分失败, 因为状态信息是通过多条路径在节点间传播的。
- **最终一致性**: 虽然节点间的信息可能会暂时不一致, 但最终所有节点的视图会收敛成一致状态。
- **网络友好**: 由于每次只与少数几个节点交换小量信息, Gossip算法对网络带宽的占用相对较低。

- 适用于需要高度可扩展和自我管理的系统, 如P2P网络、大规模云计算服务和微服务架构中的服务发现。
- 通过分散的方式有效地管理和监控系统状态, 尤其适合动态变化的网络环境。

7.2.2 故障恢复

- 修复分布式系统中出现的故障，使系统恢复到正常的状态。

- 回退恢复 Backward

- 在系统正常运行时周期性记录系统当前状态，称为检查点(checkpoint)。
- 发生故障时，将系统状态回滚至最近的检查点。
- 如有需要就执行一些修复操作，随后继续执行任务。

- 前向恢复 Forward

- 发生故障时，假定可以完全准确地得到系统中的故障和损失的详细信息。于是尝试直接从当前的错误状态恢复，继续向前推进到正确的状态。

检查点机制

- 周期性地将应用程序的状态保存到磁盘上，以便在发生故障时进行恢复。

- 独立检查点

- 每个组件都独立地定时记录本地状态

- 协调检查点

- 一个进程负责协调所有组件的检查点，以确保所有组件的检查点是一致的

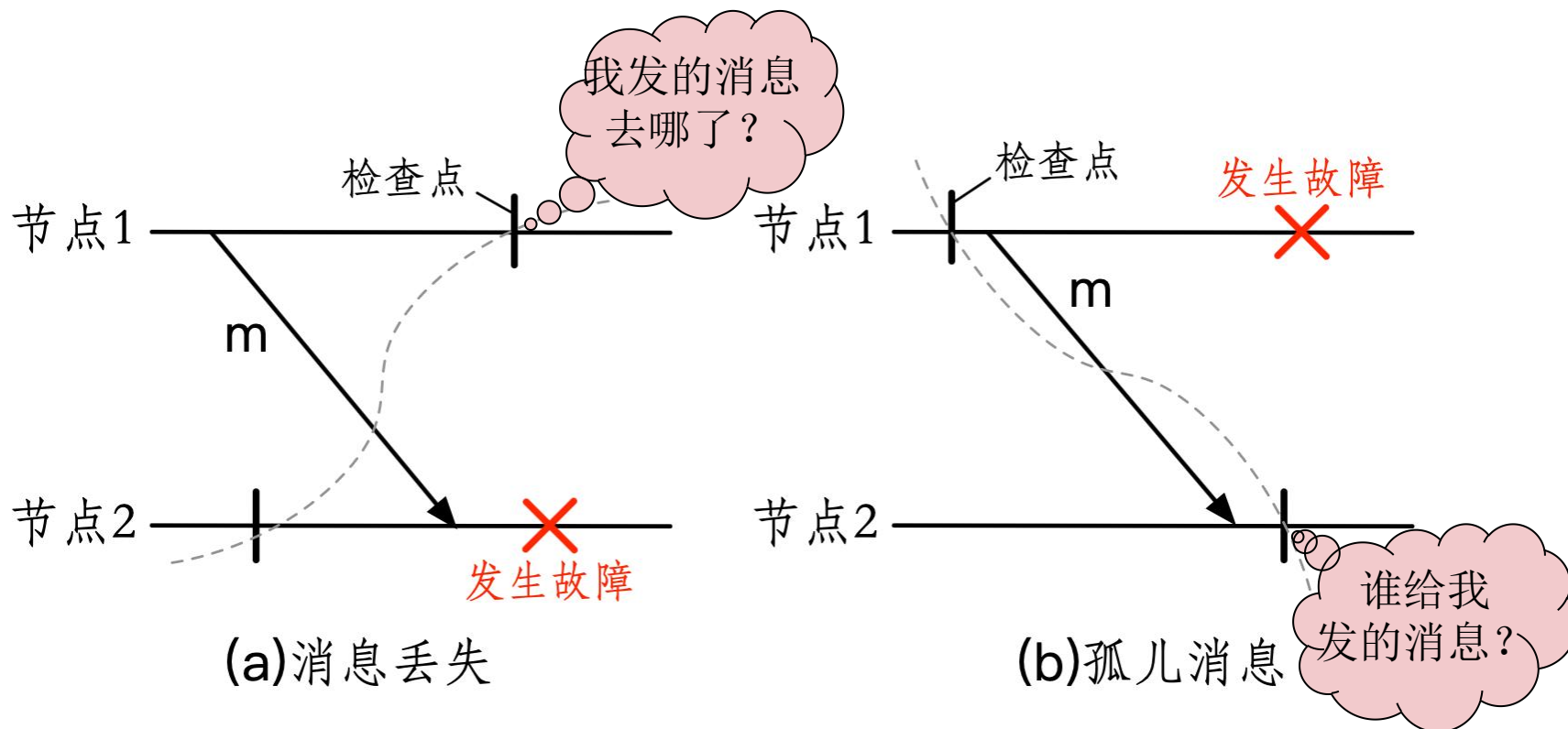
“状态”通常指系统中各个组件、节点或任务的当前运行情况或属性。

- 节点状态（可用性、负载情况、资源利用率等信息）
- 任务状态（任务的启动、执行进度、完成状态等）
- 通信状态（消息的发送、接收情况，以及网络通信的延迟、稳定性等）
- 资源状态（计算资源、存储资源、网络带宽等）

检查点机制

独立检查点存在的问题

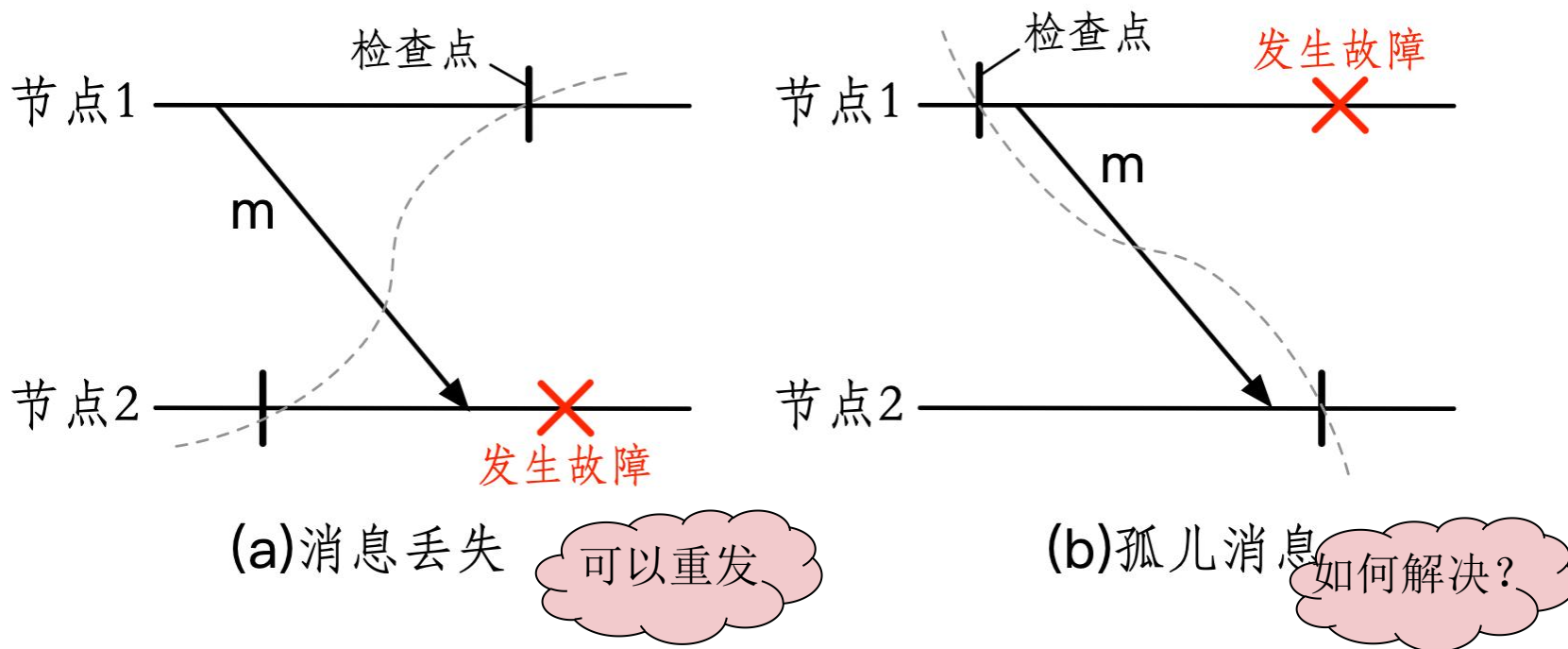
- 由于分布式系统的网络状态可能不稳定，导致消息的传播与执行速度不一致，因此每个组件的快照状态是不一致的。



检查点机制

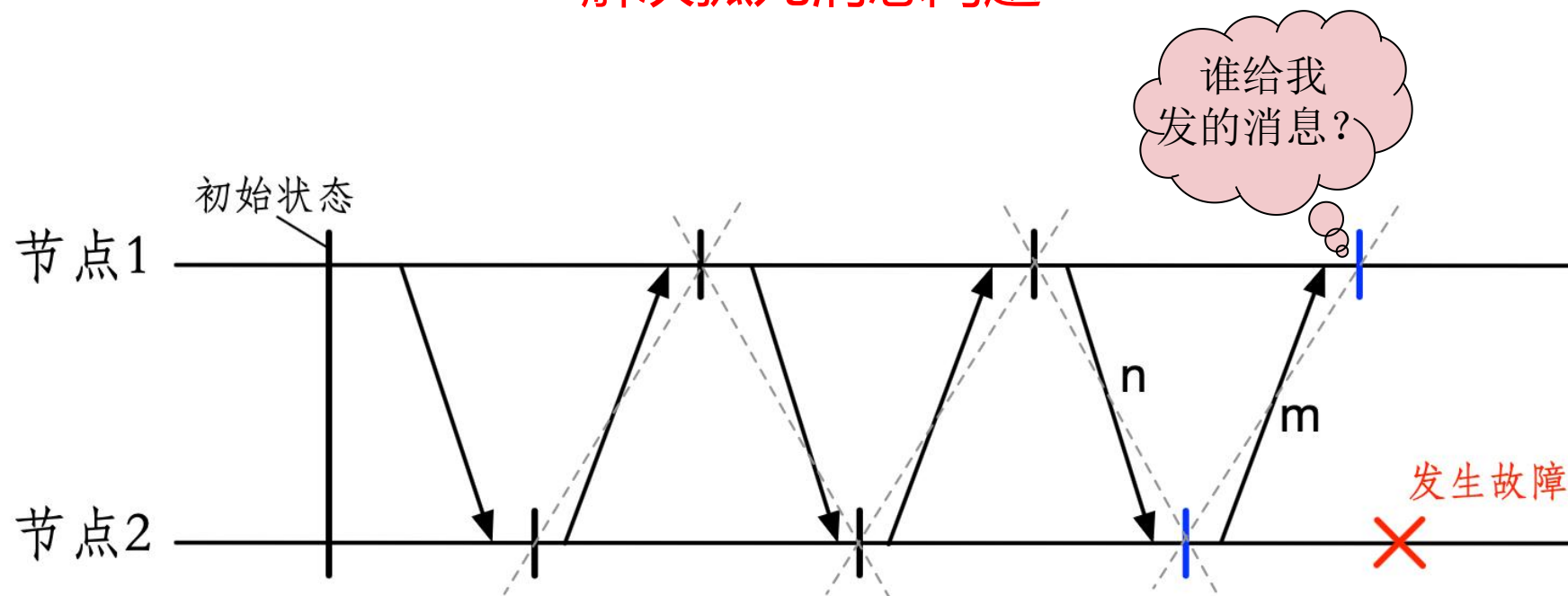
独立检查点存在的问题

- 由于分布式系统的网络状态可能不稳定，导致消息的传播与执行速度不一致，因此每个组件的快照状态是不一致的。



检查点机制

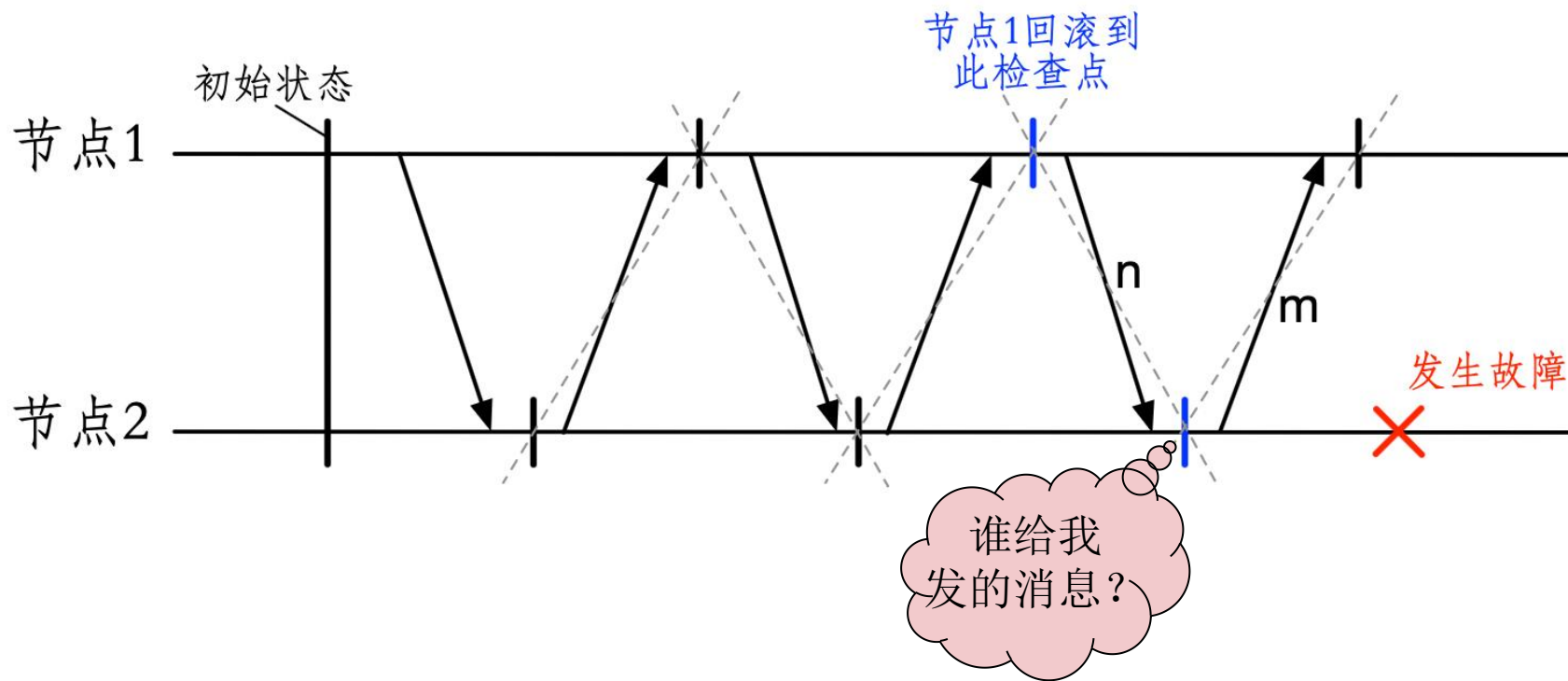
解决孤儿消息问题



- 节点1出现孤儿消息问题时，最直观的方法：回滚到上一个检查点，清除对孤儿消息的纪录；

检查点机制

解决孤儿消息问题

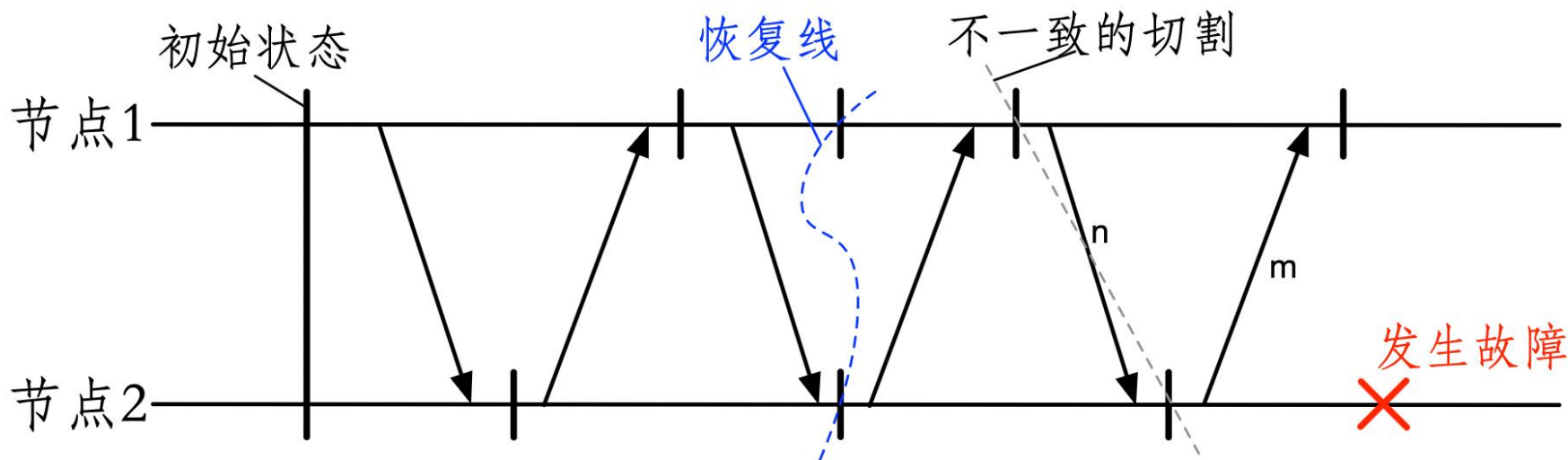


- 此时消息 n 成为了孤儿消息，节点2需要进一步回滚。
- 多米诺效应，需要回滚到初始状态.....

检查点机制

解决孤儿消息问题，同时需要防止多米诺效应

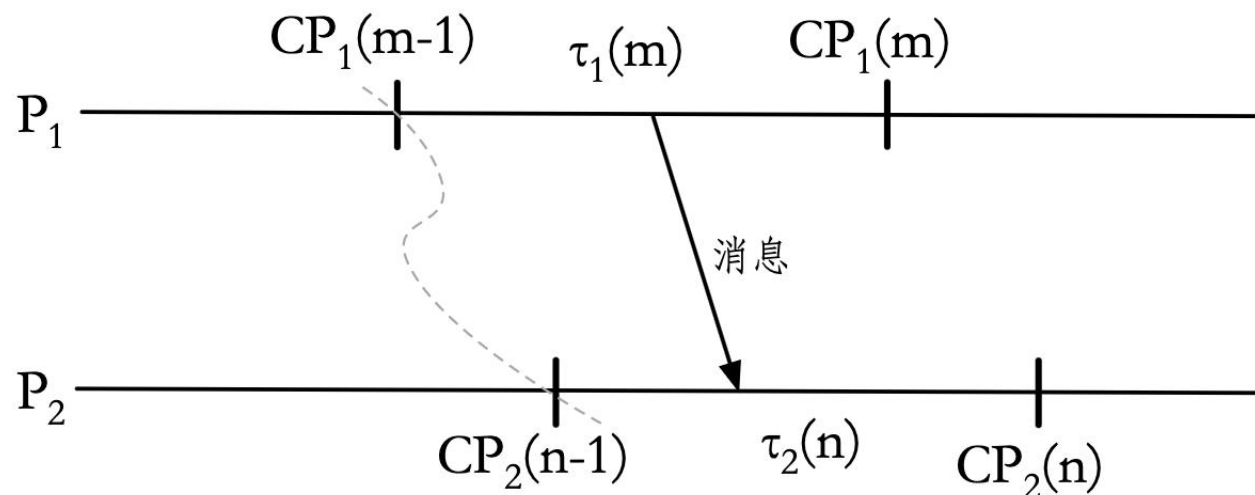
- 明确目标：通过检查点机制进行故障恢复，是为了使得系统回滚到一致的全局状态。
- 记录每个组件在哪一个检查点时系统达到了全局一致性，设立恢复线（recovery line）用于故障恢复。



如何设立恢复线?

检查点机制

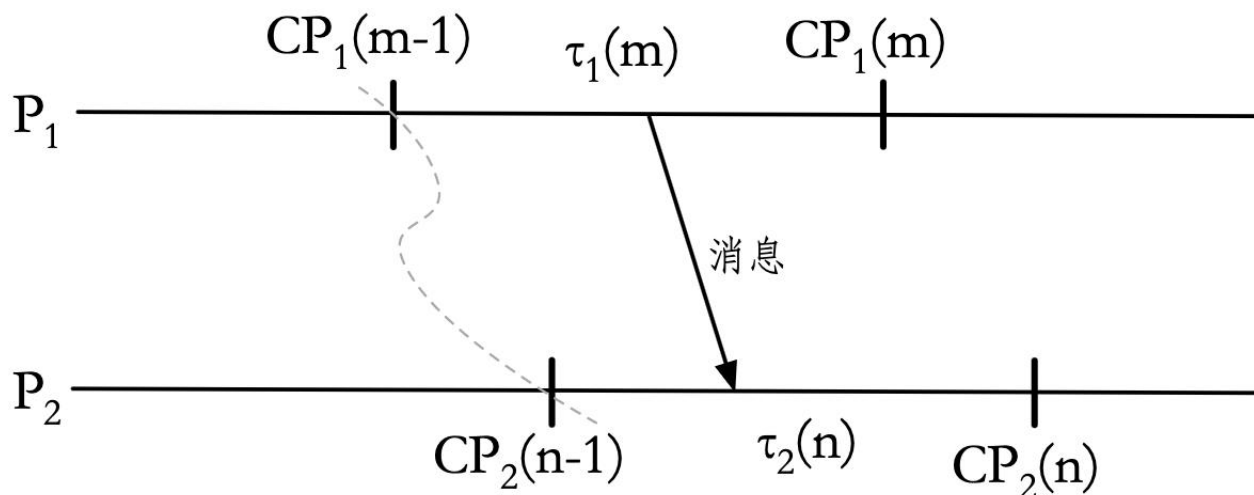
解决孤儿消息问题，同时需要防止多米诺效应



- $CP_i(m)$: 进程 P_i 的第 i 个检查点
- $\tau_i(m)$: 在两个检查点 $CP_i(m-1)$ 与 $CP_i(m)$ 之间的时间间隔
- 进程 P_1 记录: 在 $\tau_1(m)$ 期间, 发送了消息给哪些进程
- 进程 P_2 记录: 时间间隔的依赖关系 $\tau_1(m) \rightarrow \tau_2(n)$, 并保存到本地存储

检查点机制

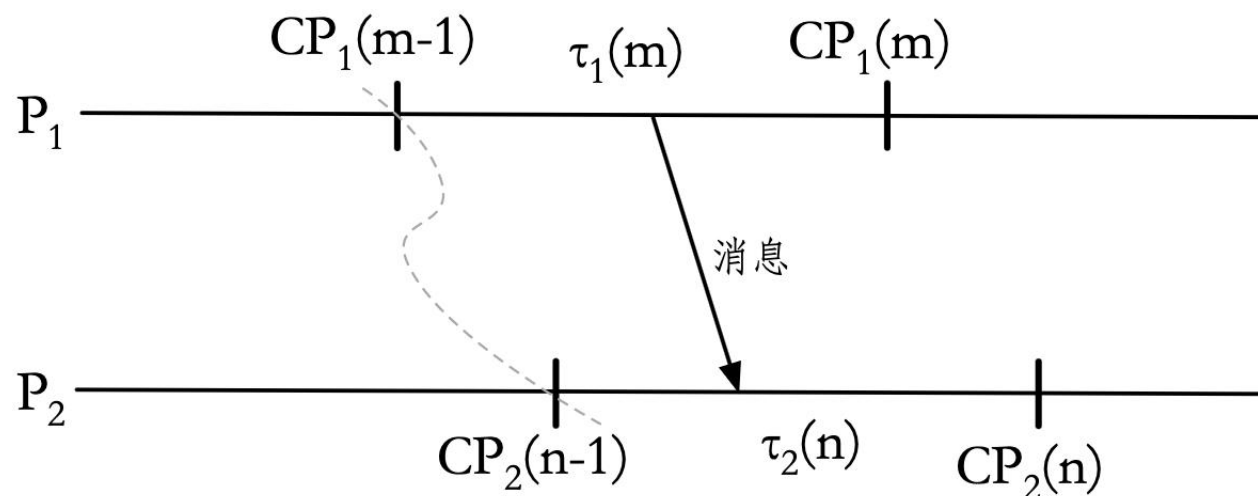
解决孤儿消息问题，同时需要防止多米诺效应



- 当进程 P_1 回滚到 $CP_1(m-1)$ 时，为了获得一致的切割，需要根据 P_2 记录的依赖关系，令进程回退到 $\tau_2(n)$ 之前的检查点状态。在这个例子中需要保证 P_2 回滚到 $CP_2(n-1)$ 检查点。
- 可以比较发送的和接收的消息数量来检测孤儿消息的存在

检查点机制

解决孤儿消息问题，同时需要防止多米诺效应



- 计算恢复线需要对每个时间间隔进行依赖关系分析
- 将状态保存到本地存储中需要额外的I/O开销
- 独立检查点可能会丢失数据

检查点机制

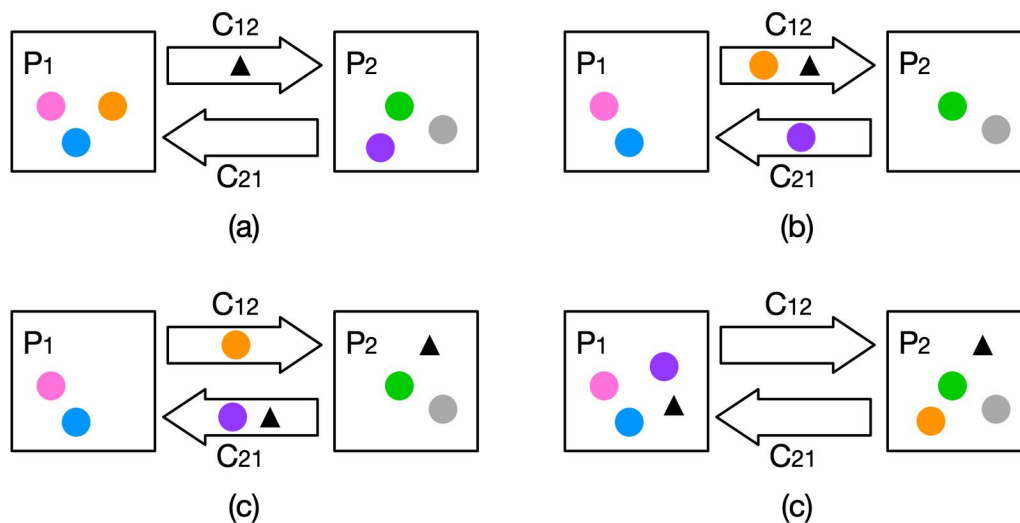
协调检查点

- 所有的进程同步将状态写到本地持久性存储中。
- 这些状态自动保持全局一致，可以避免多米诺效应导致的持续回退。
- 优点是可以尽可能地保留数据，缺点是恢复速度较慢。

检查点机制

Chandy-Lamport (CL) 算法

- 通过记录系统的全局快照来实现故障恢复。



- Step 1

假设系统中包含两个进程 P_1 和 P_2 ，它们各包含三个变量

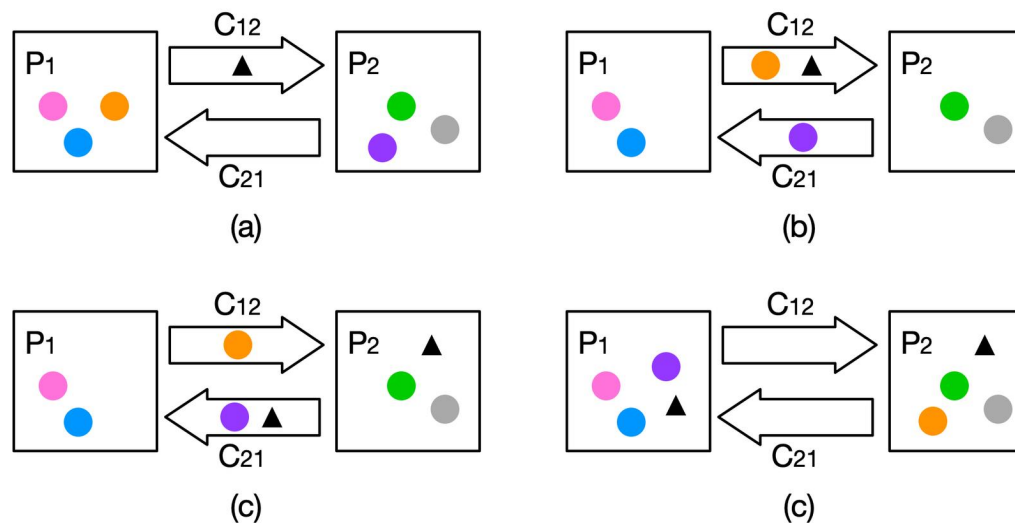
初始状态为 $P_1\{\text{pink}, \text{blue}, \text{orange}\}$, $P_2\{\text{green}, \text{purple}, \text{grey}\}$

此时由 P_1 发起全局快照， P_1 先记录本身的快照 $P_1\{\text{pink}, \text{blue}, \text{orange}\}$ ，然后向 P_2 发送一个标记▲

检查点机制

Chandy-Lamport (CL) 算法

- 通过记录系统的全局快照来实现故障恢复。



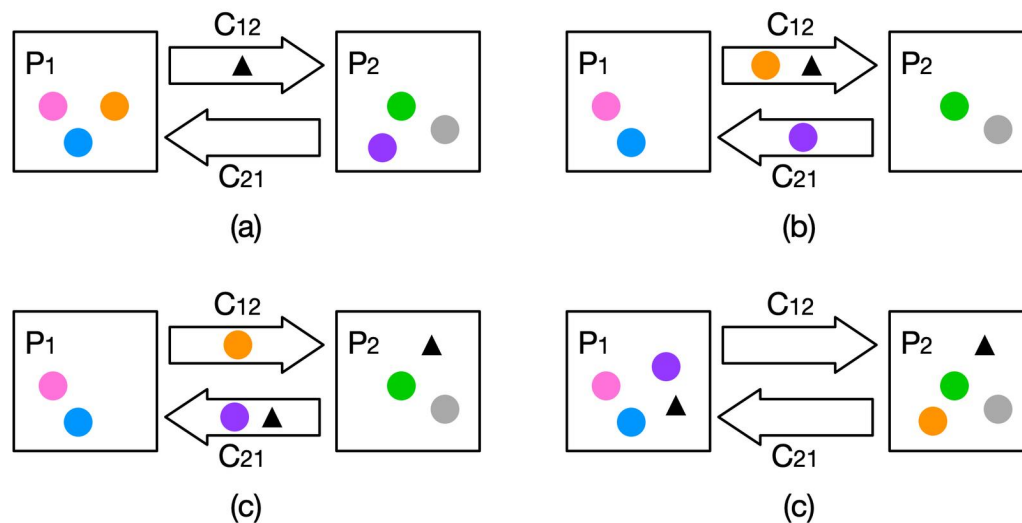
- Step 2

在标记到达 P_2 之前, P_1 向 P_2 发送消息●, P_2 向 P_1 发送消息●

检查点机制

Chandy-Lamport (CL) 算法

- 通过记录系统的全局快照来实现故障恢复。



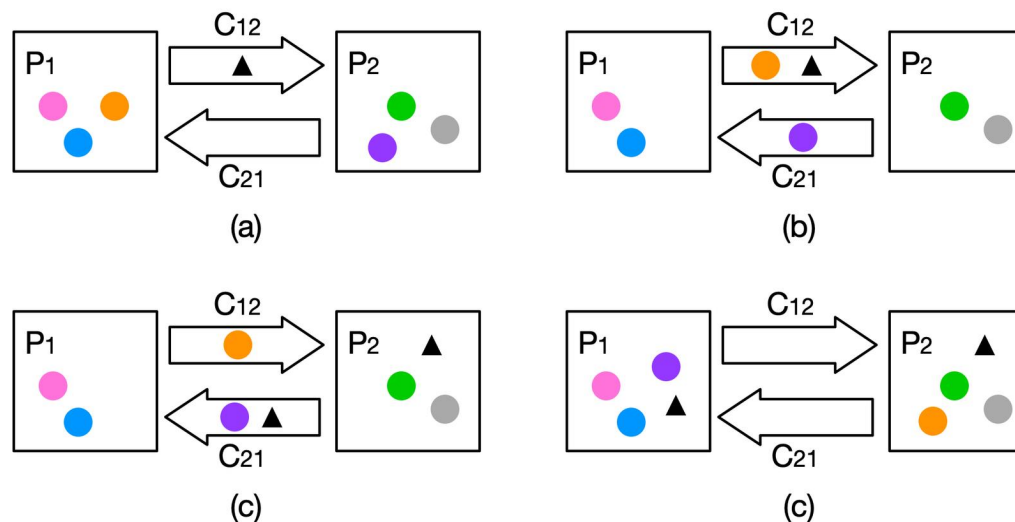
Step 3

P_2 收到 P_1 发送过来的标记之后，由于本身已有进程状态，就记录自己的本地快照： $P_2\{\bullet, \bullet\}$ ，并将 C_{12} 的状态记录为空，即 $C_{12}\{\emptyset\}$ ，并向输出通道 C_{21} 发送标记。

检查点机制

Chandy-Lamport (CL) 算法

- 通过记录系统的全局快照来实现故障恢复。



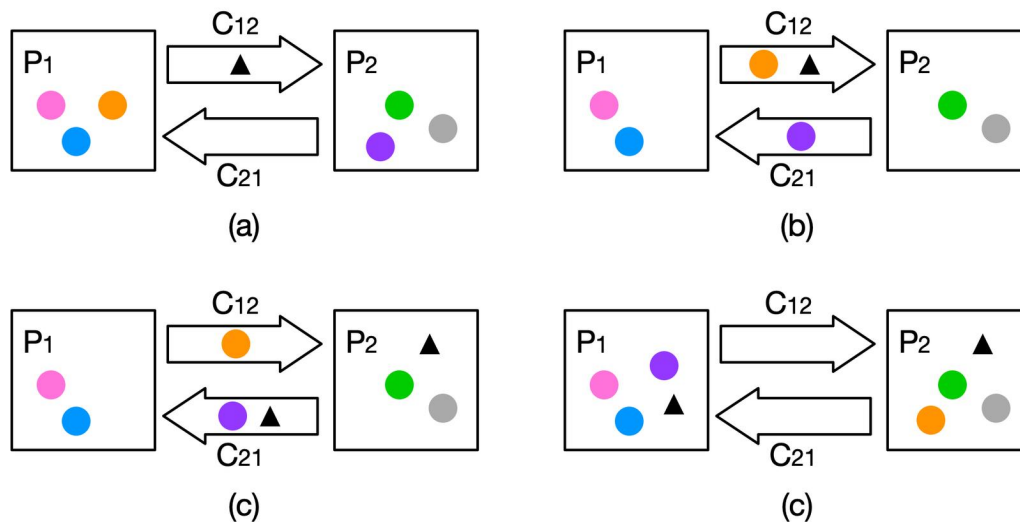
Step 4

P_1 收到 P_2 之前发送过来的消息●以及标记▲

由于 P_1 已经做了本地快照，所以 P_1 将通道 C_{21} 的状态记录为 $C_{21}\{\bullet\}$

检查点机制

Chandy-Lamport (CL) 算法



- 经过以上的操作，系统完整的快照为： $P_1\{\text{pink}, \text{blue}, \text{orange}\}$, $C_{12}\{\emptyset\}$, $P_2\{\text{green}, \text{grey}\}$, $C_{21}\{\text{purple}\}$
- 分布式系统没有**全局时钟**
- 如果 P_1 记录完，等很久 P_2 才记录，中间产生的消息就会丢失或重复
- CL 算法通过标记确保了记录的一致性，就像在流水线上放了一个隔板，隔板前的属于旧快照，隔板后的属于新周期



设立检查点的开销很大，是否可以找到一种可以减少检查点数目，但同时也能高效进行故障恢复的技术？

■ 消息日志（Message Log）技术

- 所有接收和发送消息的详细信息（如操作类型、时间戳、参与实体等）都记录下来（接收者日志/发送者日志）并持久化；
- 在发生故障时，消息日志使用一个检查点状态作为开始点，把从该点之后发送的所有消息都进行重发操作，从而避免回滚到初始状态。

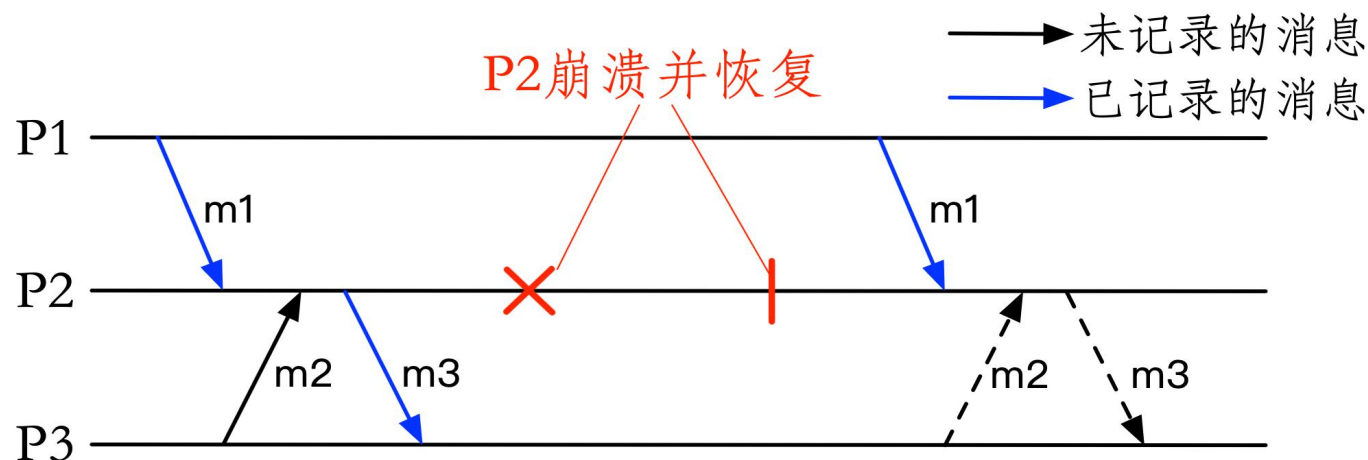
消息日志

- 日志持久化
 - 日志文件必须存储在持久化存储设备上，如硬盘或分布式文件系统。
- 使用消息日志进行故障恢复
 - **检测到故障后进行状态回滚**：使用日志文件中记录的信息回滚到最后一次一致的状态。撤销在故障点之后执行的所有操作。
 - **重新执行业务**：从最后一次已知的一致状态开始，利用日志中记录的操作信息重新执行这些操作，恢复到故障发生前的状态。这个过程称为“重放”或“重做”。
- 日志修剪
 - 删除已经被成功重放且不再需要的日志条目
 - 定期将多个日志条目合并成一个状态快照

消息日志

消息日志技术如何处理孤儿进程？

孤儿进程：在其他进程发生故障后仍然存在的进程，它的状态与从故障恢复后的其他进程状态会不一致。



- 上图例子使用接收日志。进程P2记录了m1，但没有记录m2，因此不会重放“接收m2”这个事件，当然也就不会再重新发送m3了。
- 但是，m3却被进程P3记录在了日志里。因此m3是孤儿消息，P3是孤儿进程。

消息日志

- 每个消息都具有一个头部
 - 该消息的发送者和接收者
 - 消息独一无二的序列号
 - 决定何时该消息需要传递给接收进程的传输号
- 当一个消息已经被写入到稳定存储中，就可以说这个消息是稳定的
- 通过重发稳定的消息来重新恢复故障的进程

Alvisi L, Marzullo K. **Message logging: Pessimistic, optimistic, causal, and optimal**[J]. IEEE Transactions on Software Engineering, 1998.

消息日志

- 两个进程集合
 - $DEP(m)$: 已经传送了消息 m 的进程集合
 - $COPY(m)$: 持有消息 m 的备份的那些进程集合
- 孤儿进程更准确的定义
 - 假定一些进程发生了故障， Q 是幸存的进程之一。如果 Q 包含在 $DEP(m)$ 中，而集合 $COPY(m)$ 中的所有进程都失效了，那么 Q 就是一个孤儿进程。
 - 也就是说，当一个进程依赖于消息 m ，但是无法向该进程重发消息 m ，该进程就是一个孤儿进程。

Alvisi L, Marzullo K. **Message logging: Pessimistic, optimistic, causal, and optimal**[J]. IEEE Transactions on Software Engineering, 1998.

- 避免孤儿进程的出现
 - 当集合COPY(m)中的进程都失效的时候，确保DEP(m)中没有存活的进程（强制把它们全部失效）。
 - 当一个进程成为DEP(m)中的一个成员的时候，强制它也成为COPY(m)的一个成员。
 - 可以使用两种方法实现：悲观的日志协议和乐观的日志协议。

Alvisi L, Marzullo K. **Message logging: Pessimistic, optimistic, causal, and optimal**[J]. IEEE Transactions on Software Engineering, 1998.

消息日志

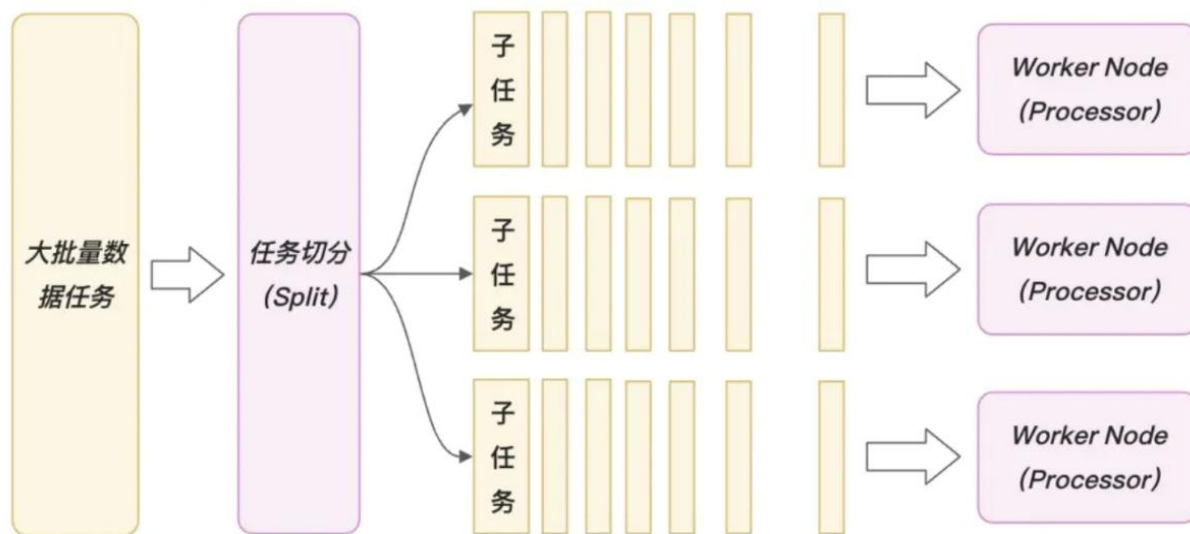
■ 悲观的日志协议

- 假定：进程P收到消息m，在写入稳定存储之前会发生故障。
- 一个进程在被传递了消息m后，除非已经把m写入日志，否则不允许发送任何消息。
- 此时就没有其他进程依赖于消息m，避免了孤儿进程的问题。

■ 乐观的日志协议

- 假定：故障发生的频率较低，因此不会频繁写入稳定存储。
- 如果集合COPY(m)中的每个进程都发生了故障，DEP(m)中的每个孤儿进程一直要回滚到以前的某个状态，直到它们不再依赖已经失败的进程。
- 需要保持追踪依赖关系。

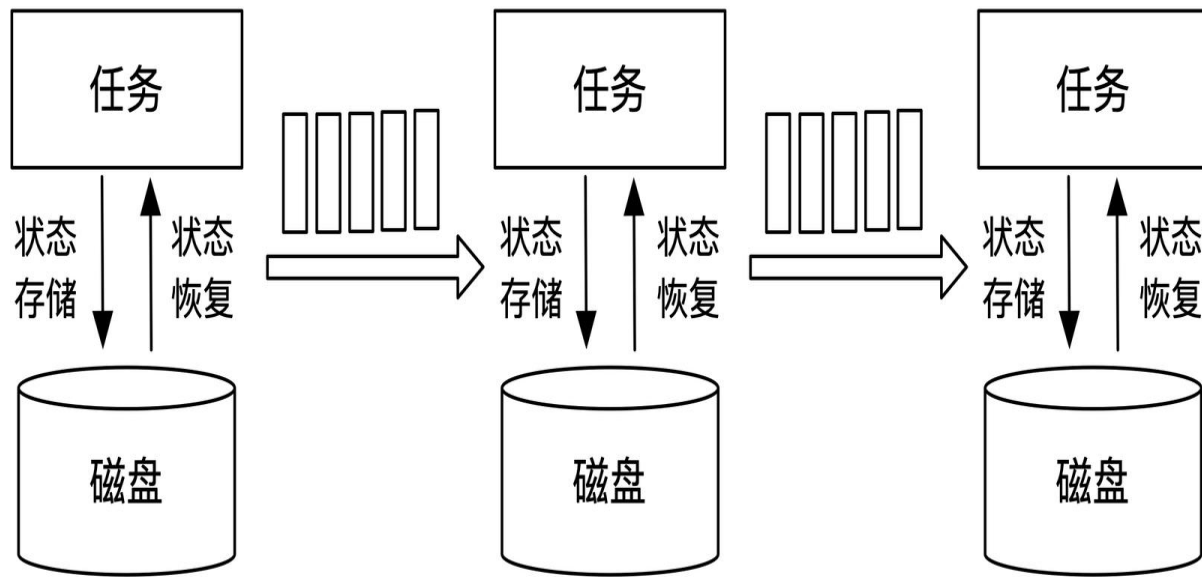
中间状态持久化



- 在分布式批处理系统中，计算任务分解为多个子任务，由不同节点并行处理。
- 子任务间存在依赖关系，需按顺序执行。
- 为了保障故障时的状态恢复，系统将执行过程中的中间状态保存至磁盘或其他存储介质，使得节点故障时能通过读取这些状态来恢复任务。

中间状态持久化

- 中间状态
 - 在程序运行中，输入数据经过一系列操作后的状态，既非输入也非输出数据，而是中间过渡状态。
- 可以被持久化的对象
 - 处理数据后的结果
 - 数据本身
 - 数据位置



故障转移

- 在发生故障时，将系统负载从故障节点转移到其他健康节点，以保证系统的可用性。
- 重启策略 (Restart)
 - 立即重启
 - 延迟重启
- 迁移策略 (Migration)
 - 全量迁移
 - 逐步迁移
- 负载均衡策略 (Load Balancing)
 - 轮询策略 (Round-Robin)
 - 随机策略 (Random)
 - 权重策略 (Weighted)

前向恢复技术

- 前向恢复也称为前馈恢复或纠错恢复，它不依赖于将系统状态回滚到以前的正确状态，而是尝试直接从当前的错误状态恢复，继续向前推进到正确的状态。
- ✓ 错误掩蔽 (Error Masking)
 - 通过软件或硬件冗余来掩蔽故障的影响，允许系统继续运行，无需回滚
 - 应用：错误更正码 (ECC内存)、RAID磁盘阵列
- ✓ 异常处理 (Exception Handling)
 - 在软件层面检测异常条件并执行预定义的异常处理程序，以纠正错误或以其他方式响应，而无需回滚到以前的状态。
 - 应用：软件中的try-catch异常处理机制

第七章、分布式批处理系统

01

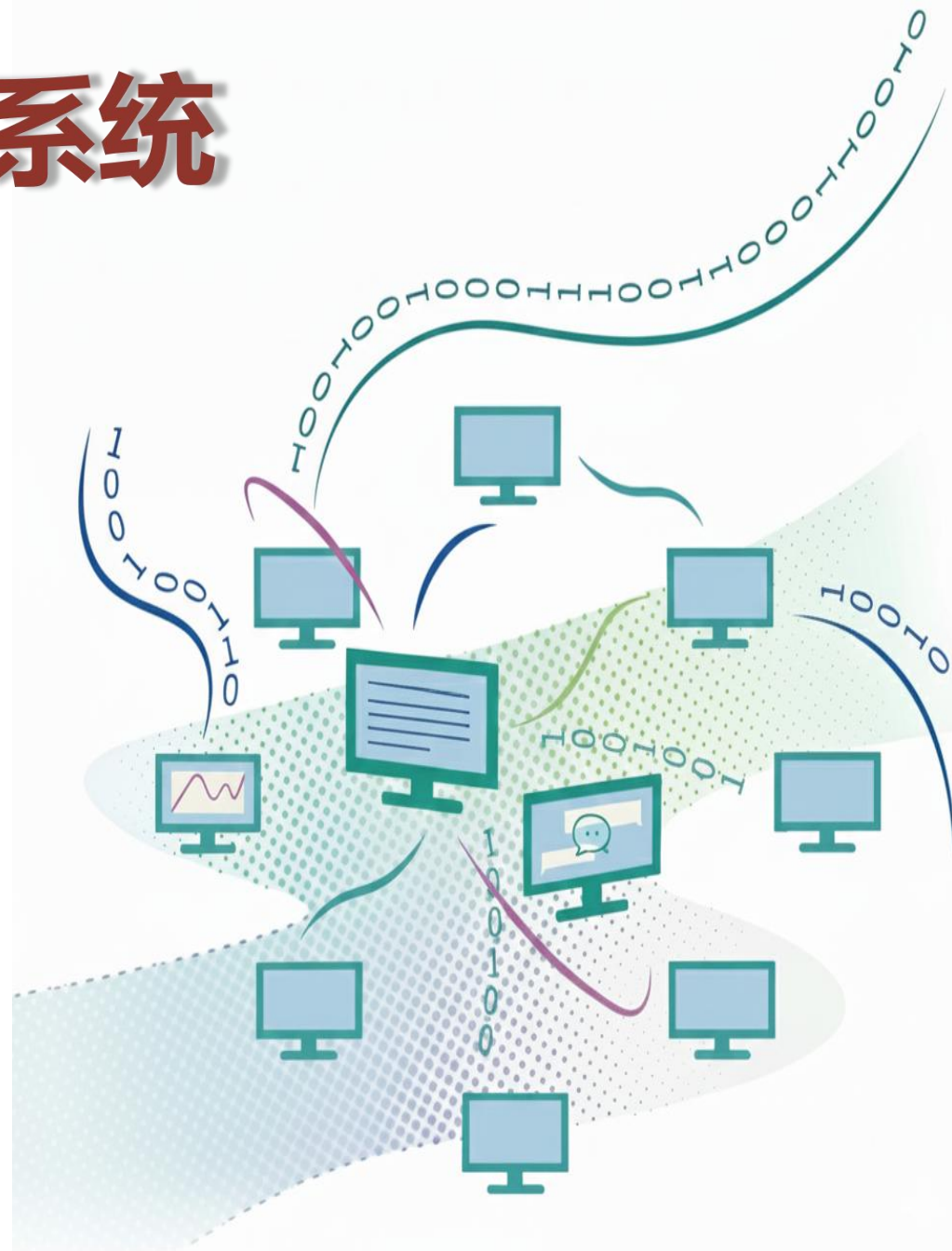
7.1 任务分配与协调

02

7.2 计算故障检测与恢复

03

7.3 批处理系统-MapReduce



7.3 批处理系统 MapReduce

- MapReduce是一种大数据计算模型和编程模型，用于在分布式环境中并行处理大量数据。
- Google于2004年提出，用于解决其搜索引擎中大规模网页数据的并行化处理。
- MapReduce简化了分布式计算的开发和执行过程，提供了自动的任务划分、数据分布、任务调度和高效容错机制。

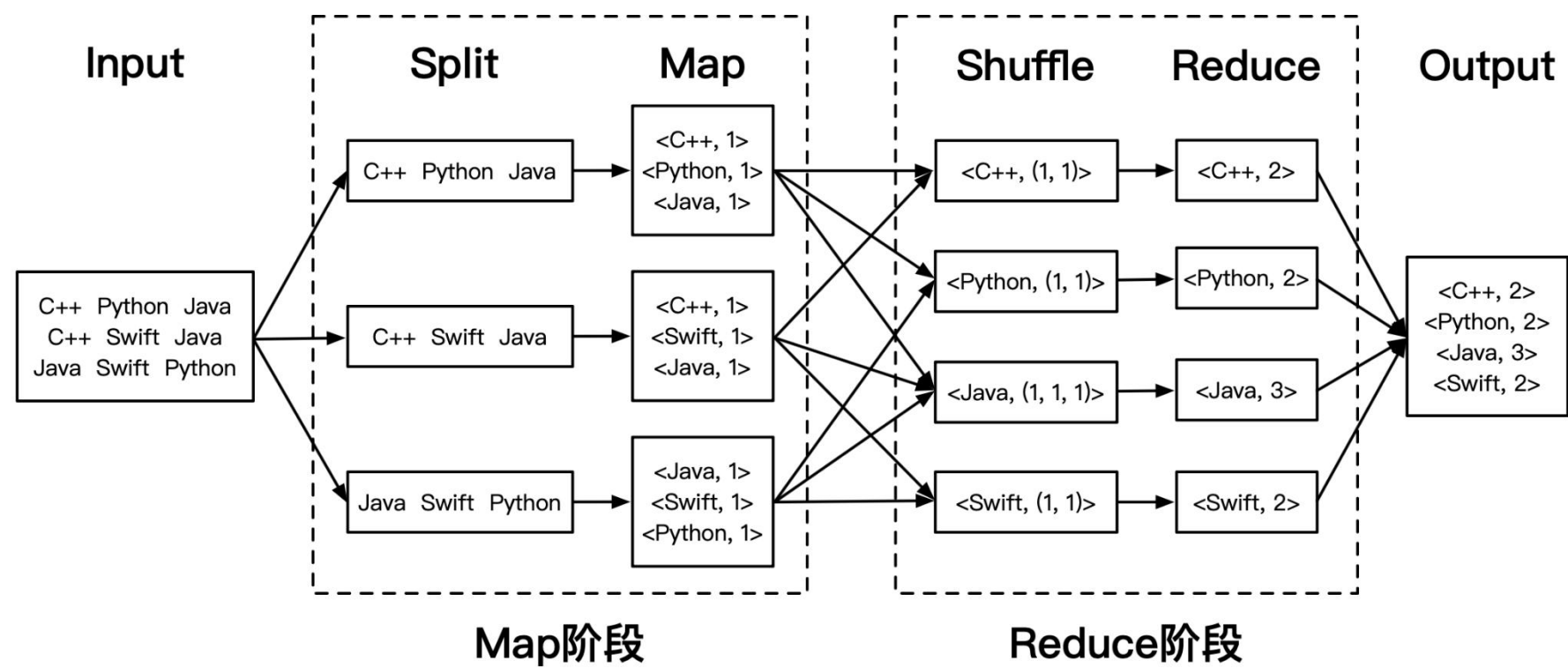
7.3 批处理系统 MapReduce

- Map（映射）和Reduce（归约）的概念源自函数式编程语言

Map：接受一个函数和一个列表，将这个函数应用于列表中的每个元素，返回一个新的列表	Reduce：将一个二元操作函数应用于序列中的元素，将其归约为单一的结果。
<pre>function square(x): return x * x list = [1, 2, 3, 4, 5] squared_list = map(square, list) // squared_list 现在是 [1, 4, 9, 16, 25]</pre>	<pre>function add(x, y): return x + y list = [1, 2, 3, 4, 5] sum = reduce(add, list, 0) // 初始值为 0 // sum 现在是 15</pre>
MapReduce中：Map阶段处理输入数据，对每个数据项执行相同的操作（如分析或转换），并输出一系列中间键值对。	MapReduce中：Reduce阶段对所有具有相同“键”的值执行归约操作（如合并或求和），生成最终的结果。

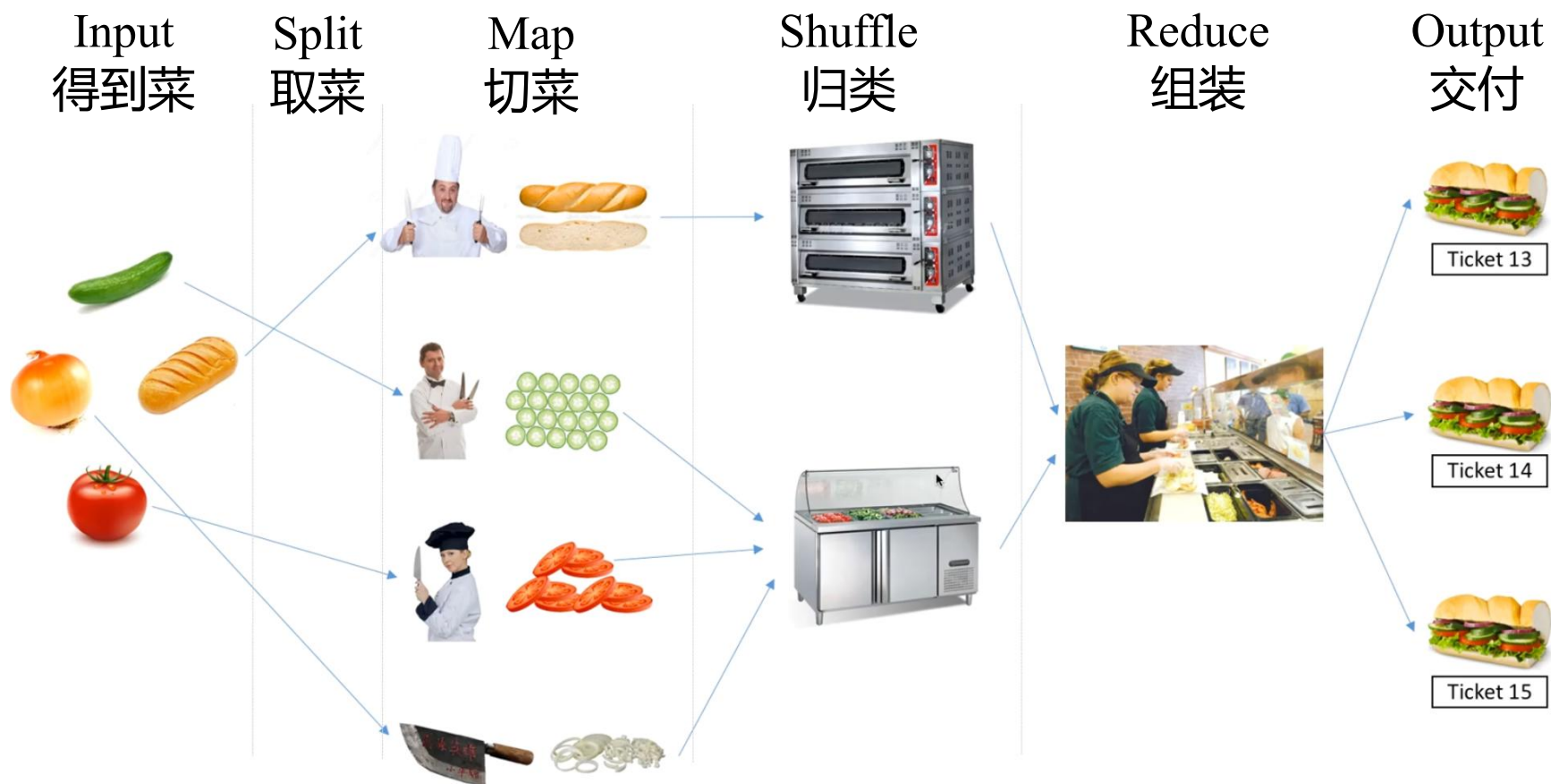
7.3 批处理系统 MapReduce

MapReduce的工作流程-以统计单词出现次数的应用为例



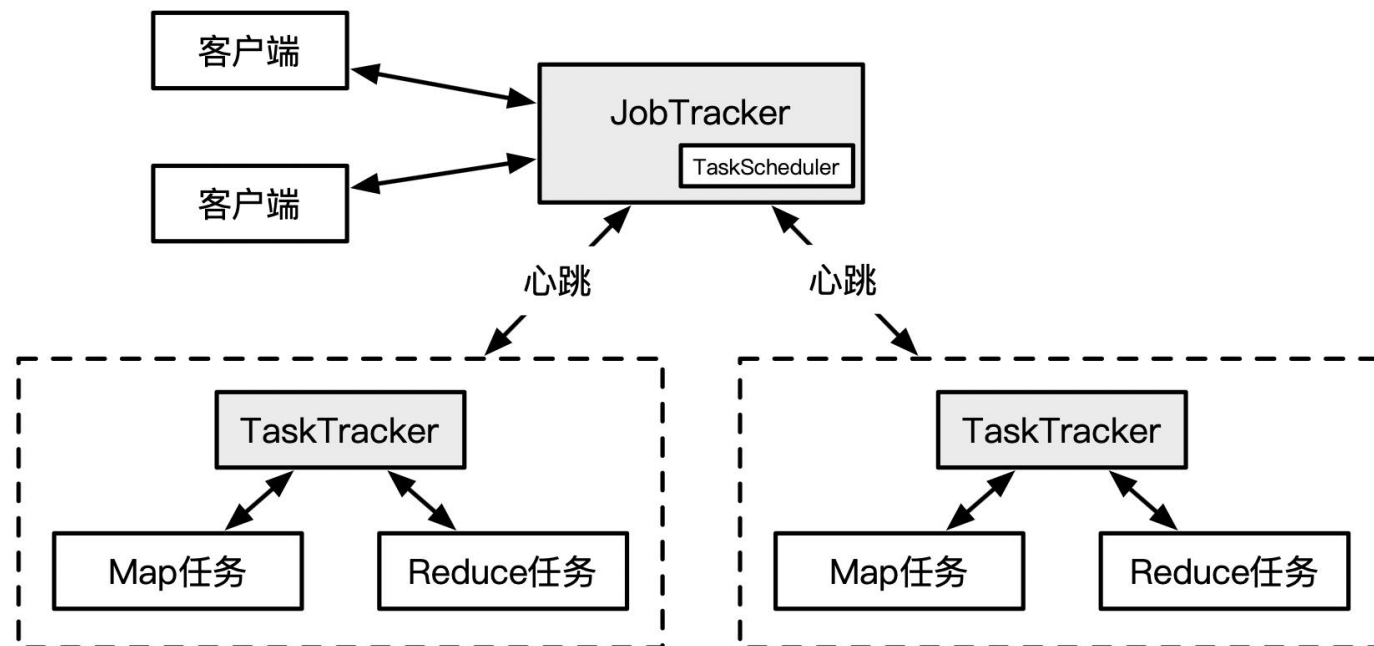
7.3 批处理系统 MapReduce

MapReduce的工作流程-抽象成后厨做菜的过程

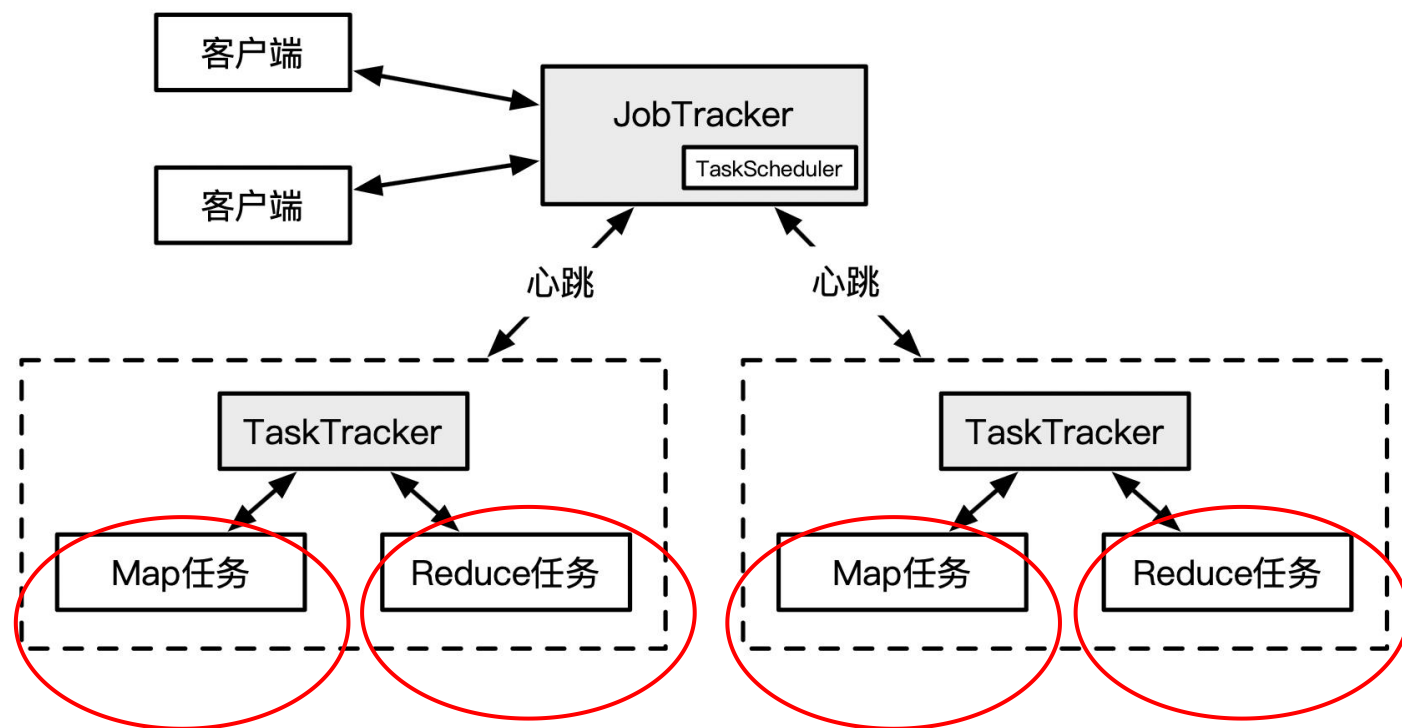


7.3.1 MapReduce架构

- MapReduce采用主从（Master/Slave）架构，包括一个主节点（JobTracker）和多个从节点（TaskTracker）

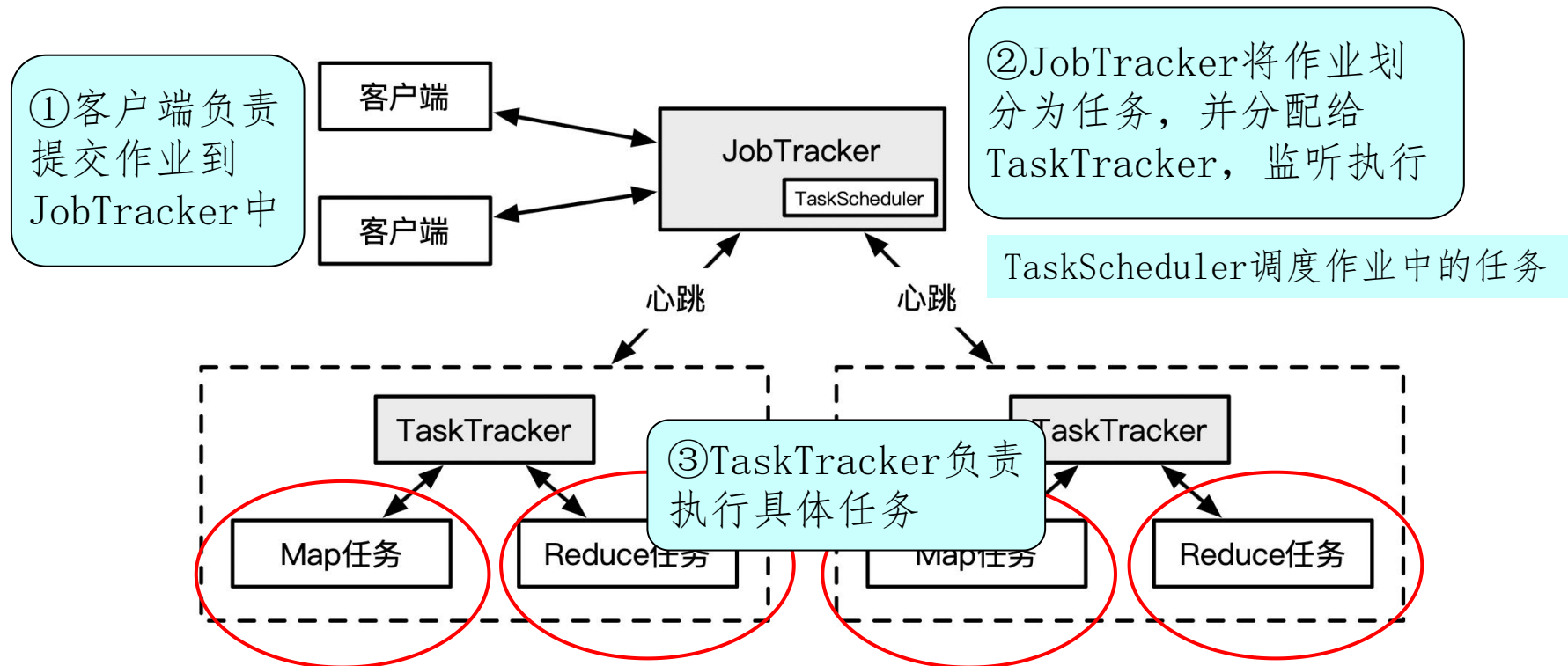


7.3.1 MapReduce架构



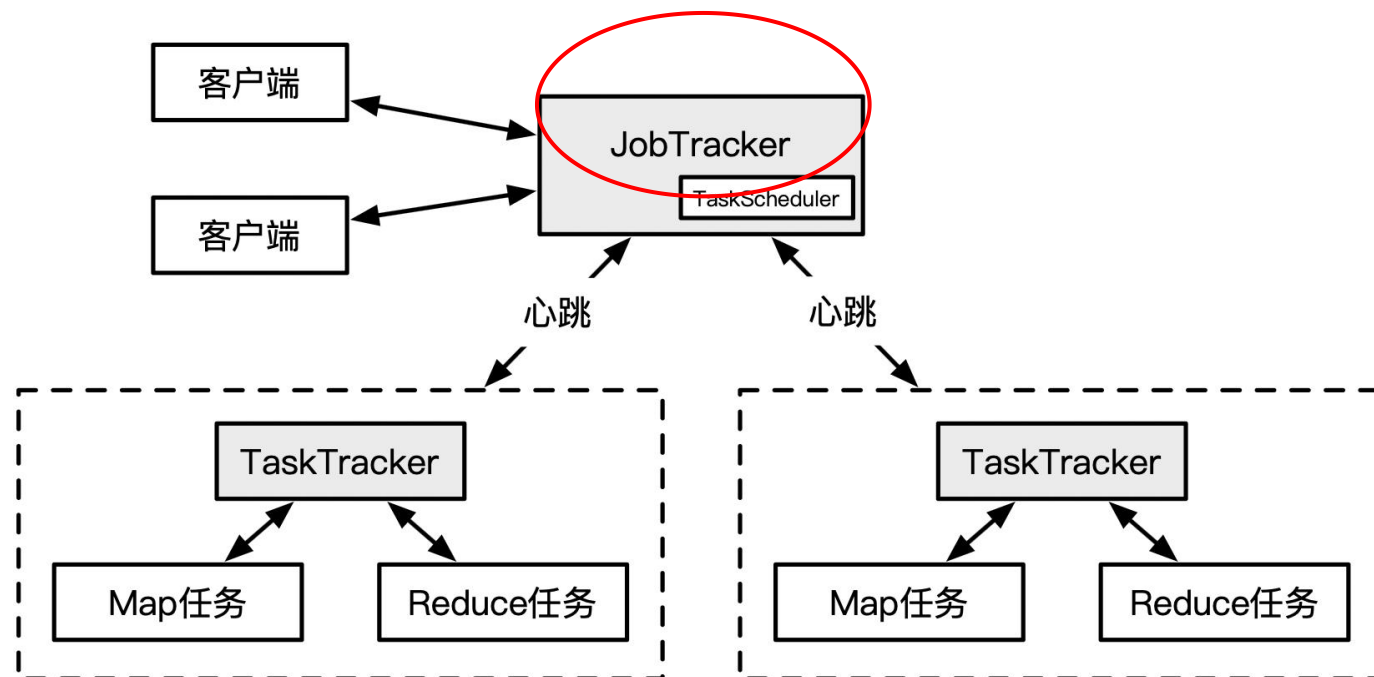
- 一个完整的Map和Reduce过程被称为一个**作业 (Job)**。完成一个批处理任务往往需要**多个作业**。
- 启动一个作业时，MapReduce会将**这个作业划分为多个Map任务和多个Reduce任务**。这些任务会在多个节点上并行执行。

7.3.1 MapReduce架构



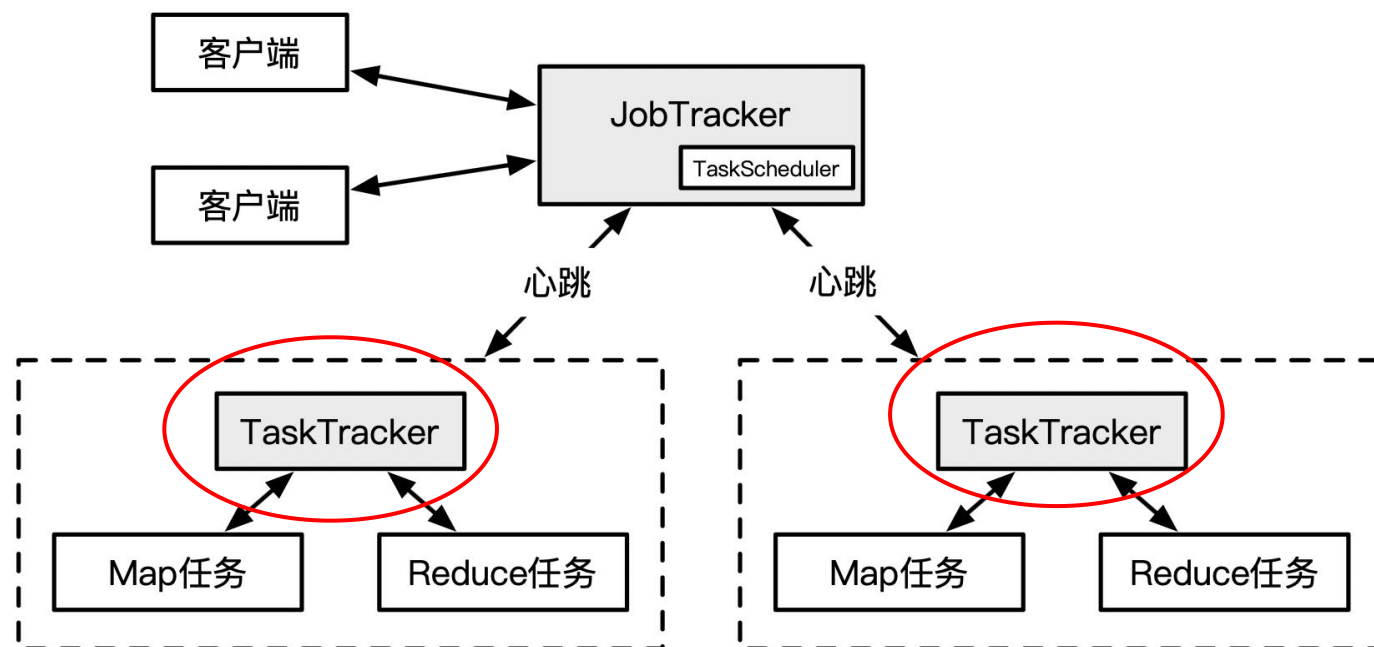
- 一个完整的Map和Reduce过程被称为一个**作业 (Job)**。完成一个批处理任务往往需要**多个作业**。
- 启动一个作业时，MapReduce会将这个作业划分为多个**Map任务**和多个**Reduce任务**。这些任务会在多个节点上并行执行。

7.3.1 MapReduce架构



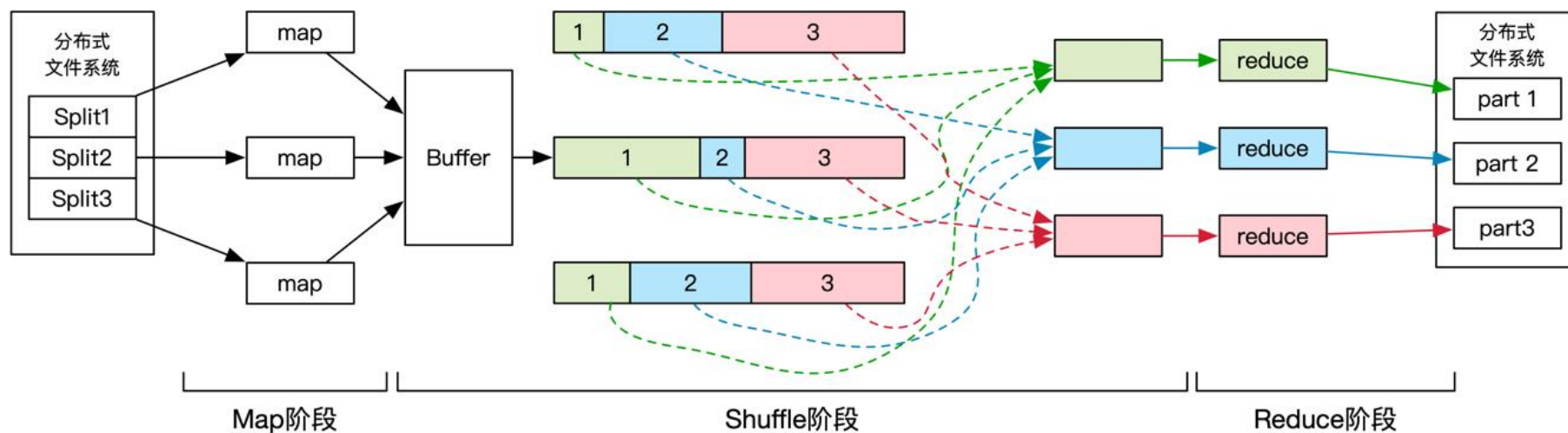
- JobTracker是MapReduce作业的主节点，负责作业调度、资源管理、任务监控和协调整个作业的执行。
 - 资源监控和作业调度
 - 监控整个作业的执行

7.3.1 MapReduce架构



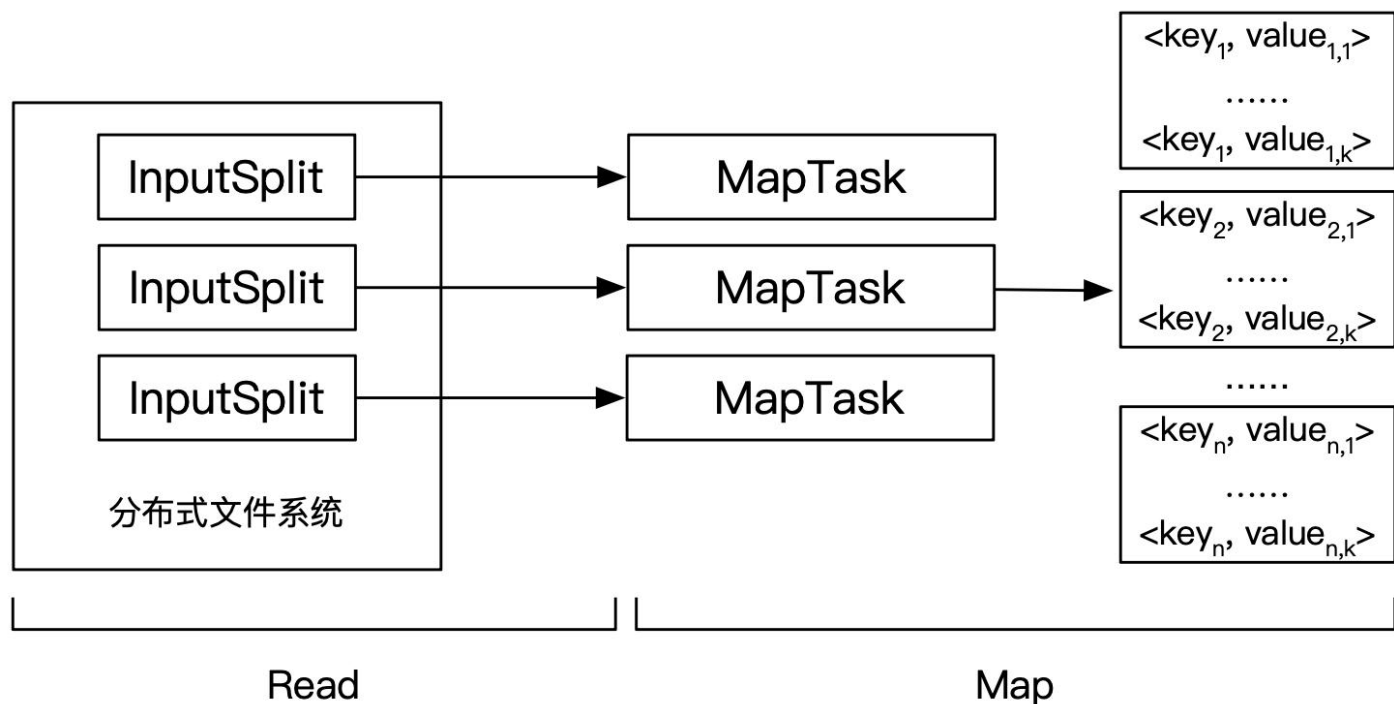
- TaskTracker是MapReduce作业的从节点，接收并执行来自JobTracker的**任务**分配。
- 通过心跳周期性地将本节点上资源的使用情况和任务的运行进度汇报给JobTracker；
- 接收JobTracker发送过来的**命令**并执行相应的操作。

7.3.2 MapReduce工作流程



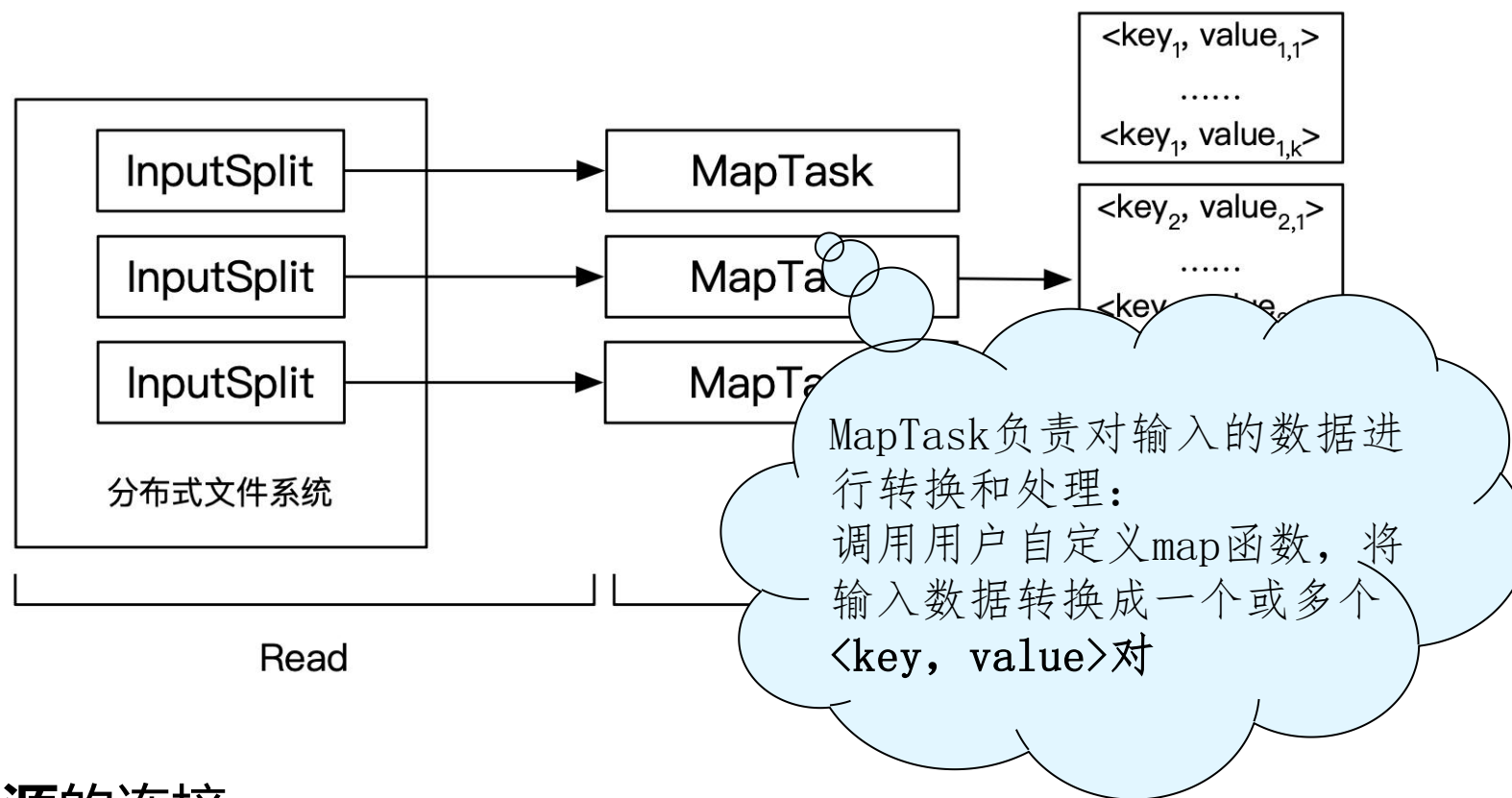
- Map阶段：从数据源读取数据，执行map函数，以<key, value>的格式输出map函数处理结果。
- Shuffle阶段：从Map阶段输出到Reduce阶段输入的整个过程，它将Map阶段处理的数据传递给Reduce阶段。
- Reduce阶段：将Shuffle阶段传递过来的数据调用reduce函数，输出本作业的结果。

Map阶段



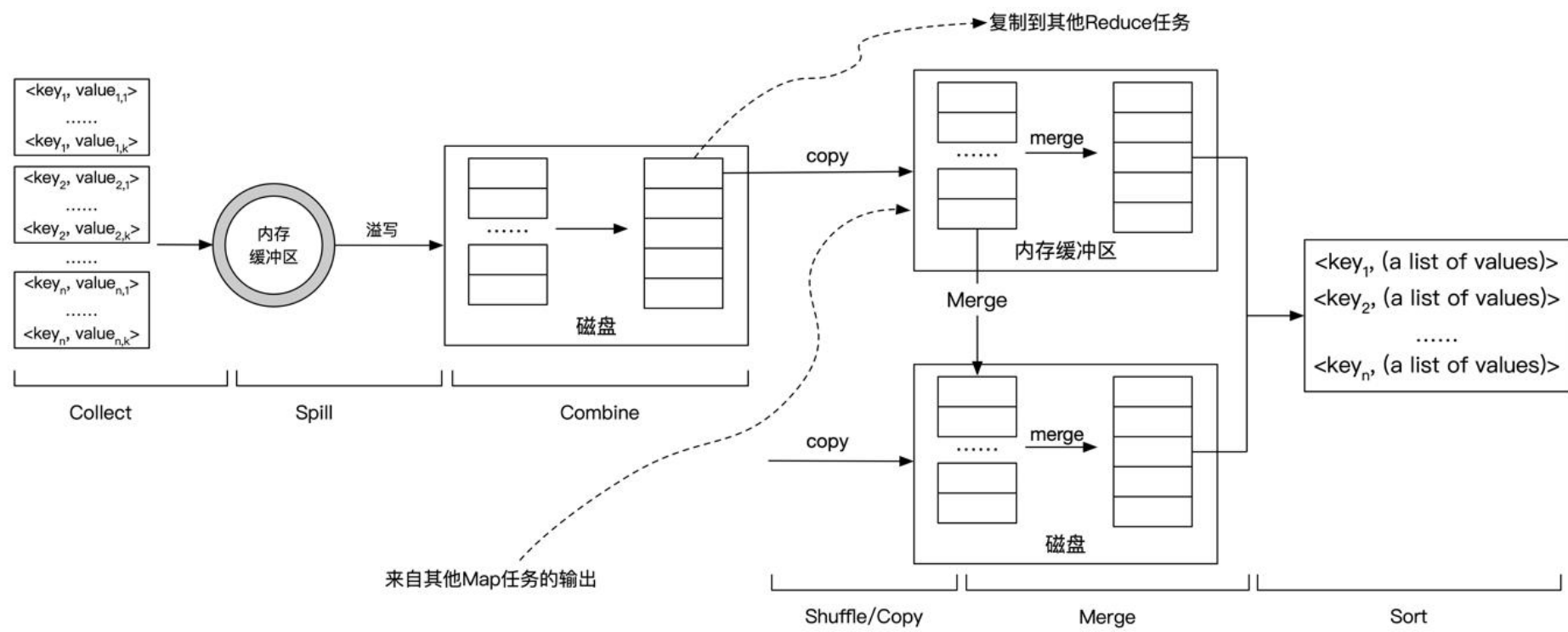
1. 建立与**数据源**的连接;
2. Map任务将原始数据切割成**逻辑上**的数据块InputSplit, 通常为64MB或128MB;
3. 分块被分配给不同的**MapTask**处理, 默认一个InputSplit对应一个MapTask。

Map阶段

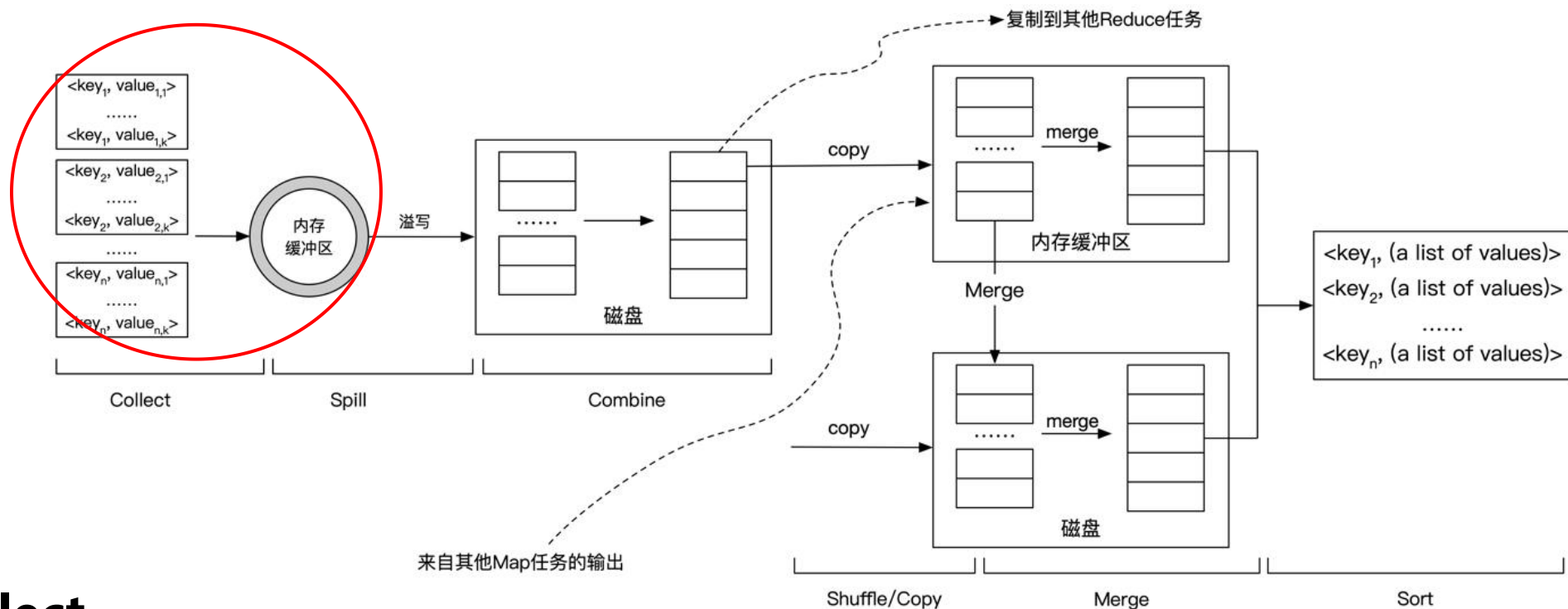


1. 建立与**数据源**的连接；
2. Map任务将原始数据切割成**逻辑上**的数据块InputSplit，通常为64MB或128MB；
3. 分块被分配给不同的**MapTask**处理，默认一个InputSplit对应一个MapTask。

Shuffle阶段



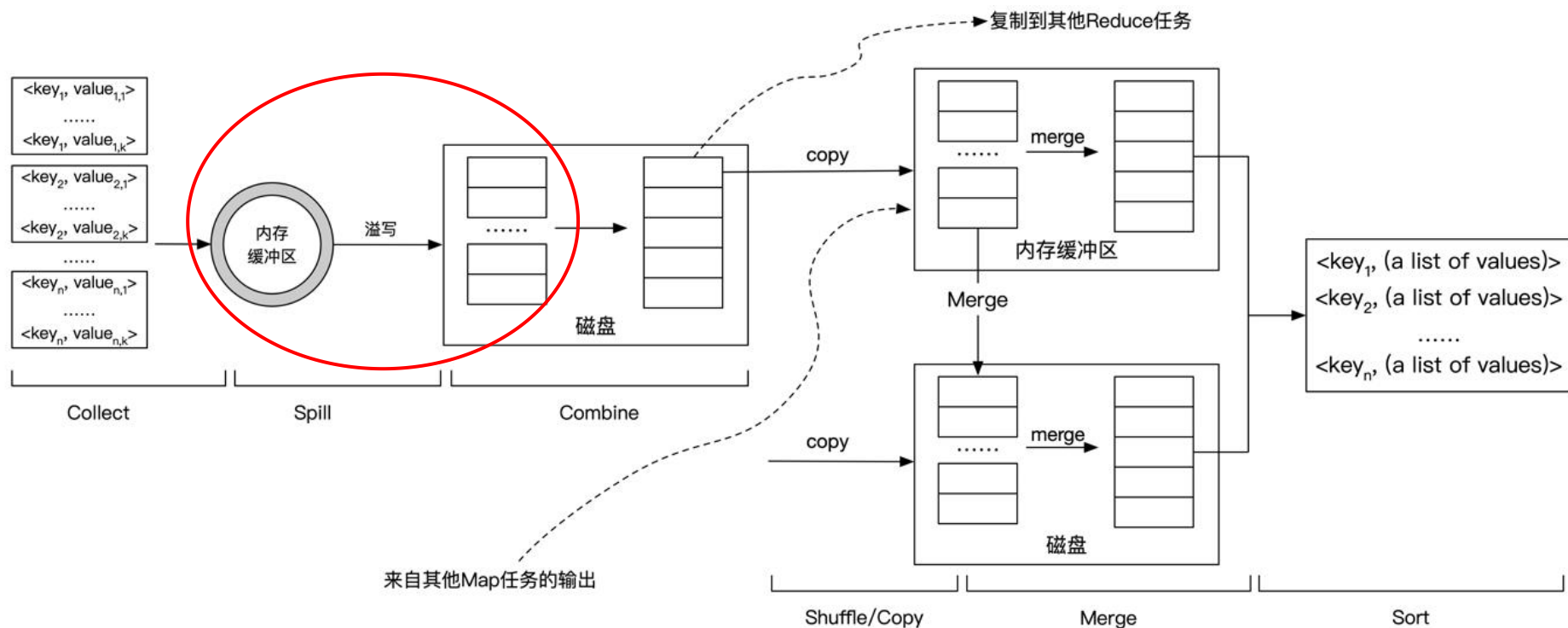
Shuffle阶段



■ Collect

- 将MapTask的输出结果收集到环形内存缓冲区（默认大小为100M）
- 存入缓冲区之前根据key对 $\langle \text{key}, \text{value} \rangle$ 对进行哈希划分，缓冲区中数据的格式为 $\langle \text{partition}, \text{key}, \text{value} \rangle$
- 选择环形缓冲区的原因：并发读写与循环读写

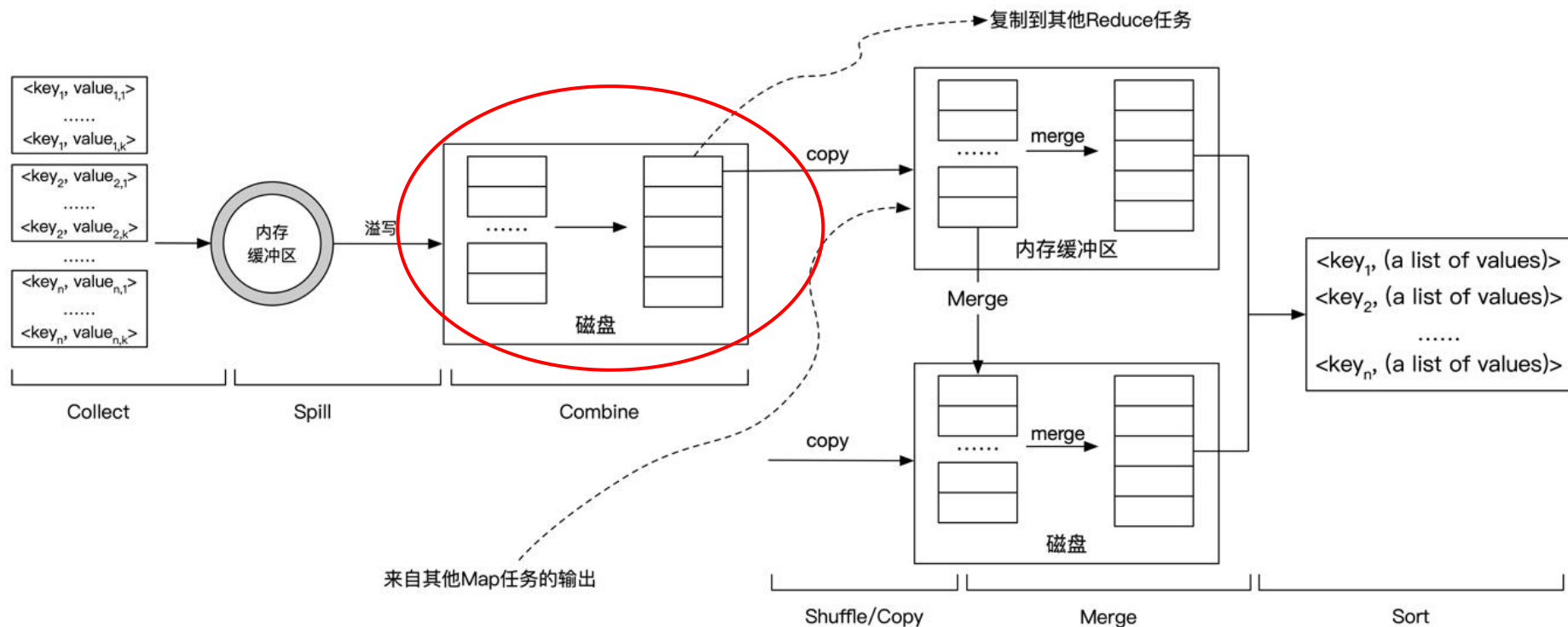
Shuffle阶段



■ Spill

- 缓冲区中的数据量达到一定阈值时（例如80MB），系统会启动一个线程将缓冲区中的溢出数据写入磁盘
- 溢写的同时，会对数据进行合并、压缩，防止过多溢写文件

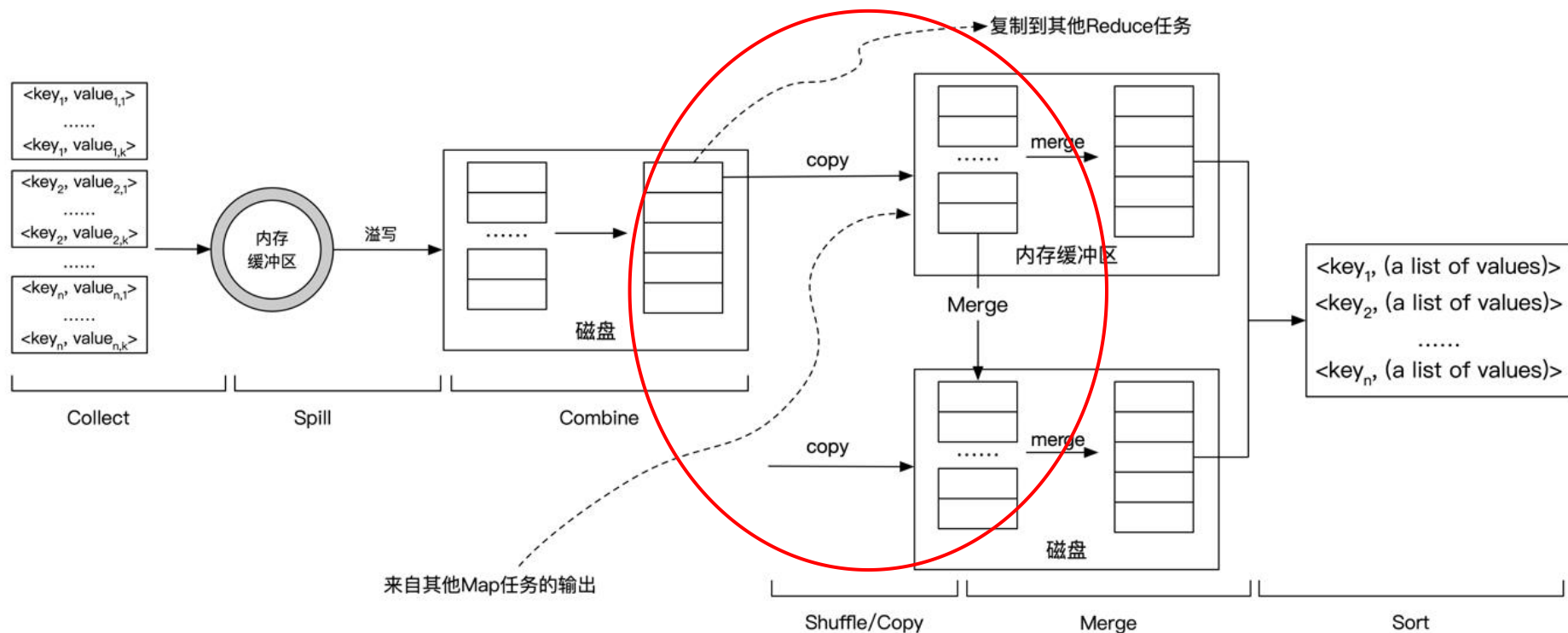
Shuffle阶段



Combine

- 把所有溢出的临时文件根据键进行合并操作，以确保一个MapTask最终只产生一个中间数据文件

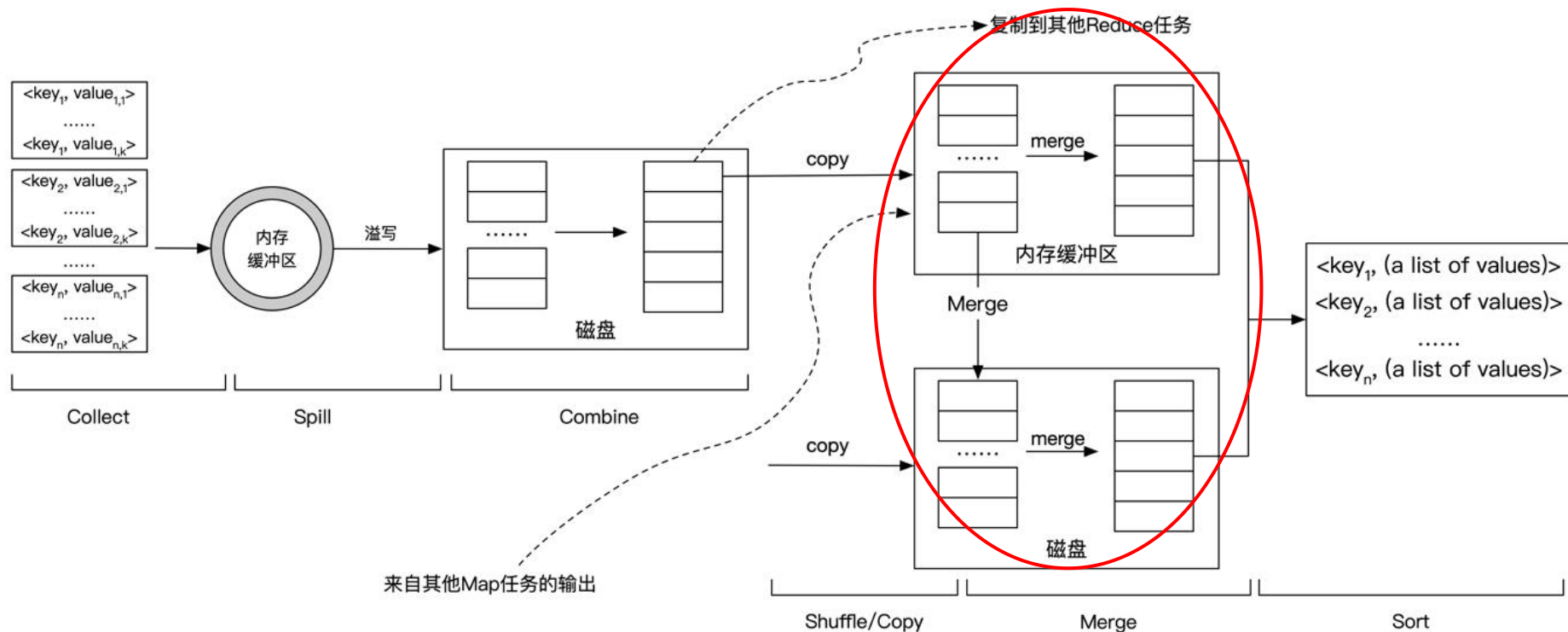
Shuffle阶段



■ Copy

- 将MapTask的输出数据复制到对应ReduceTask所在节点的本地磁盘上，使得Reduce任务可以从本地磁盘直接读取数据
- “移动计算而不移动数据”

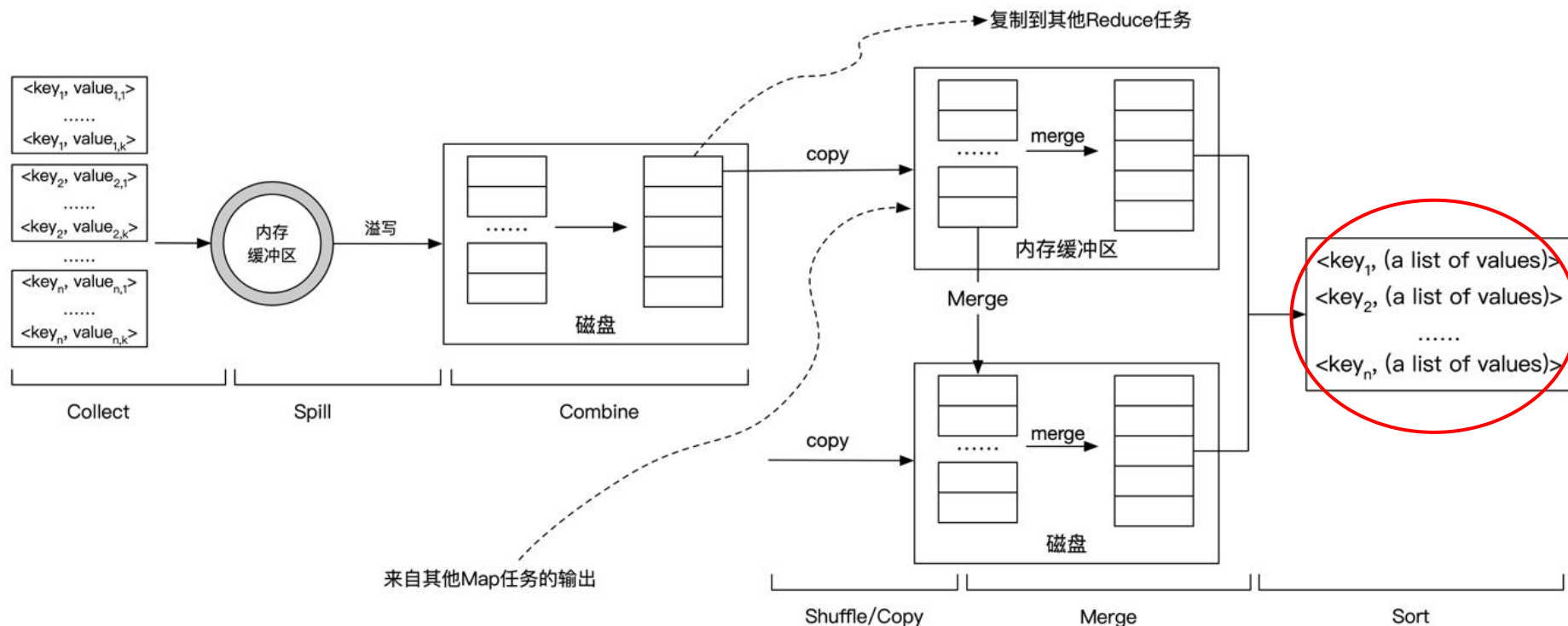
Shuffle阶段



■ Merge

- 在复制数据的同时，会在后台开启两个线程对内存到本地的数据文件进行合并操作
- 将来自不同MapTask输出的相同key的 $\langle \text{key}, \text{value} \rangle$ 对合并到一个文件中

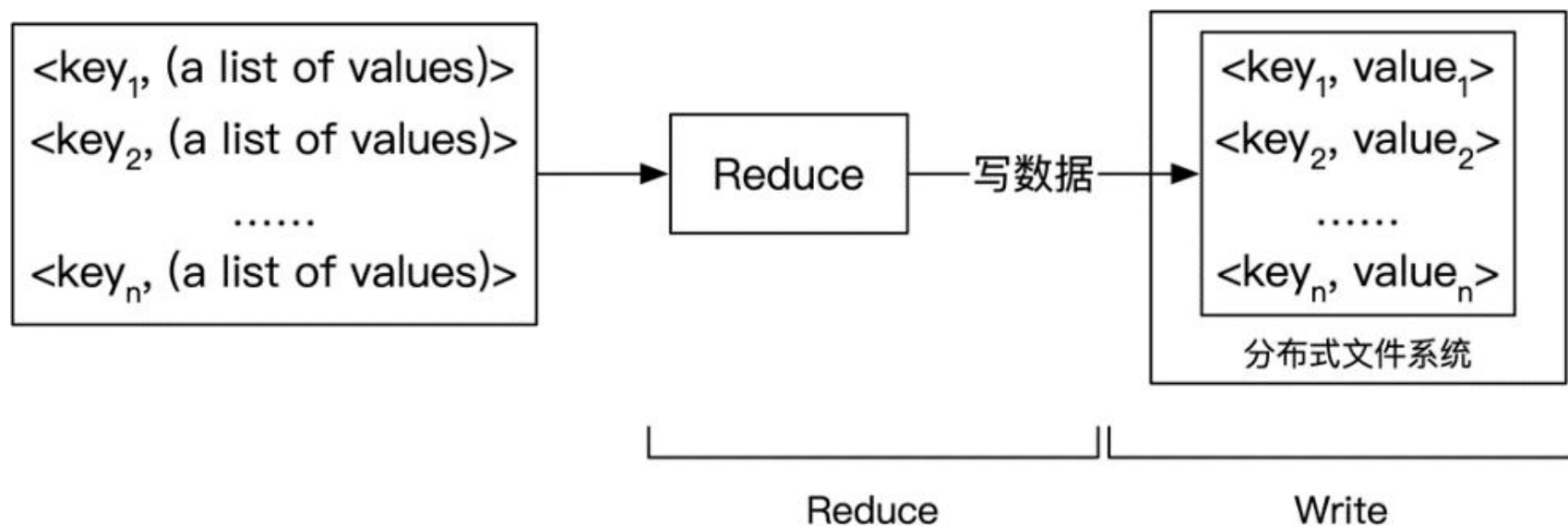
Shuffle阶段



■ Sort

- 在对数据进行合并的同时，会进行排序操作
- MapTask阶段已经对数据进行了局部的排序，ReduceTask只需保证复制的数据的最终整体有效性即可

Reduce阶段



- 整个作业的汇总和输出阶段
- 调用用户自定义的reduce函数，处理Shuffle后的 $\langle \text{key}, \text{list}(\text{values}) \rangle$ 对
- 相同键的数据根据用户自定义的逻辑合并起来
- ReduceTask以 $\langle \text{key}, \text{value} \rangle$ 的形式输出最终结果到分布式文件系统中

7.3.3 故障恢复

- MapReduce系统包含数百甚至数千台机器，每台机器都有发生故障的可能
- 采取故障恢复机制，确保在某些机器或网络故障的情况下仍能够继续执行整个任务
- 从硬件故障与任务失败两个角度分析故障恢复技术

硬件故障

■ JobTracker 故障

- 集群中只有一个 JobTracker，因此此故障非常严重
- 最直接的方法：人工介入并重启
- 实际上，一些企业会创建多个备用 JobTracker 节点。在主 JobTracker 失败之后採用领导选举算法来又一次确定 JobTracker 节点

硬件故障

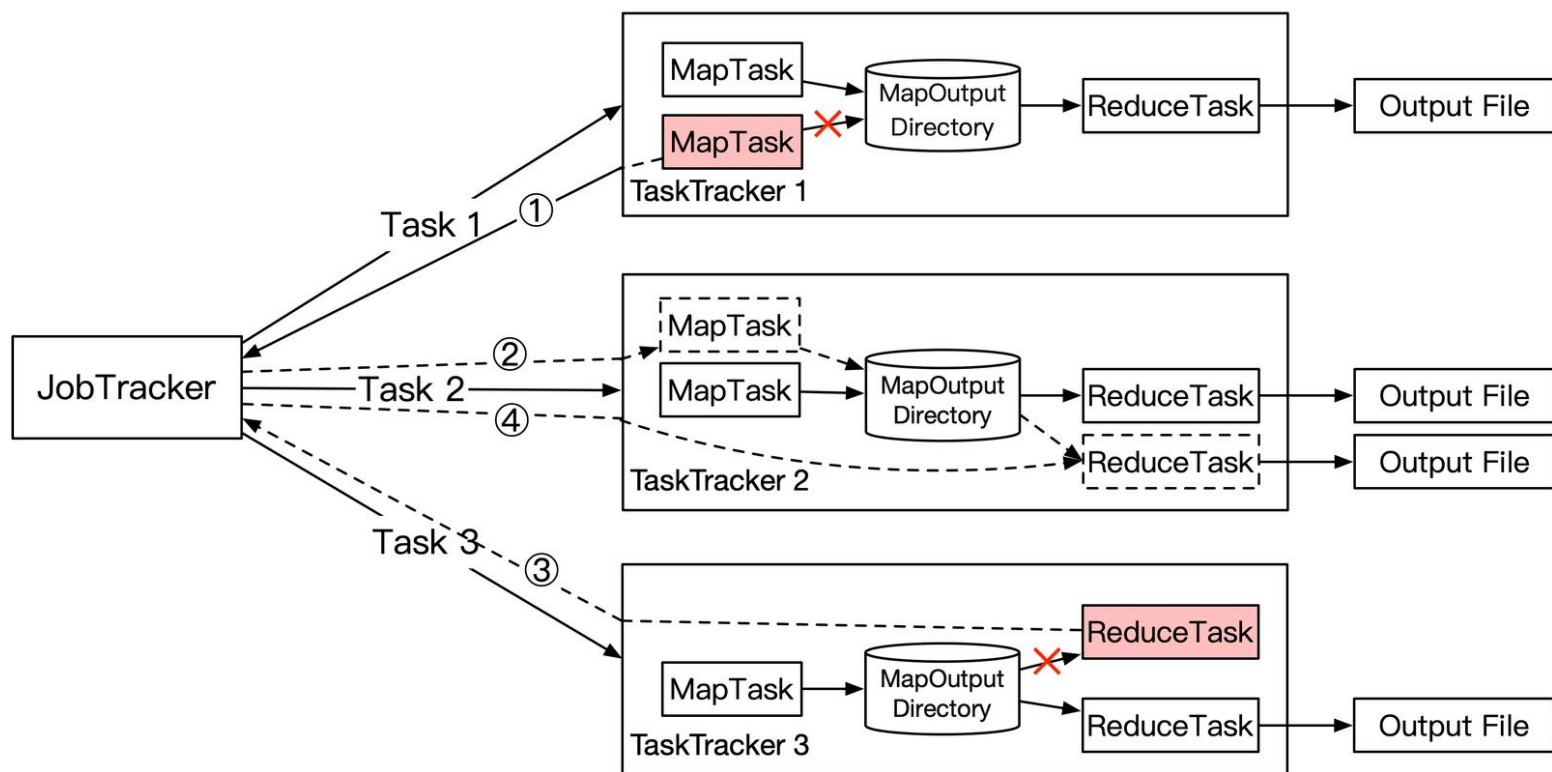
■ TaskTracker 故障

- 正常情况下，TaskTracker 通过不断的心跳机制与 JobTracker 保持通信
- 一旦某个 TaskTracker 出现故障，它会停止或减少向 JobTracker 发送心跳
- 通信中断时，存在两个关键条件用于决定哪些任务需要被终止并重新调度到其他 TaskTracker 上执行：

- 任务尚未完成，包括正在执行的Map或Reduce任务；
- 任务的输出未被其他任务使用，即Map任务的输出还未被任何Reduce任务使用，或者这些输出数据丢失了。

硬件故障

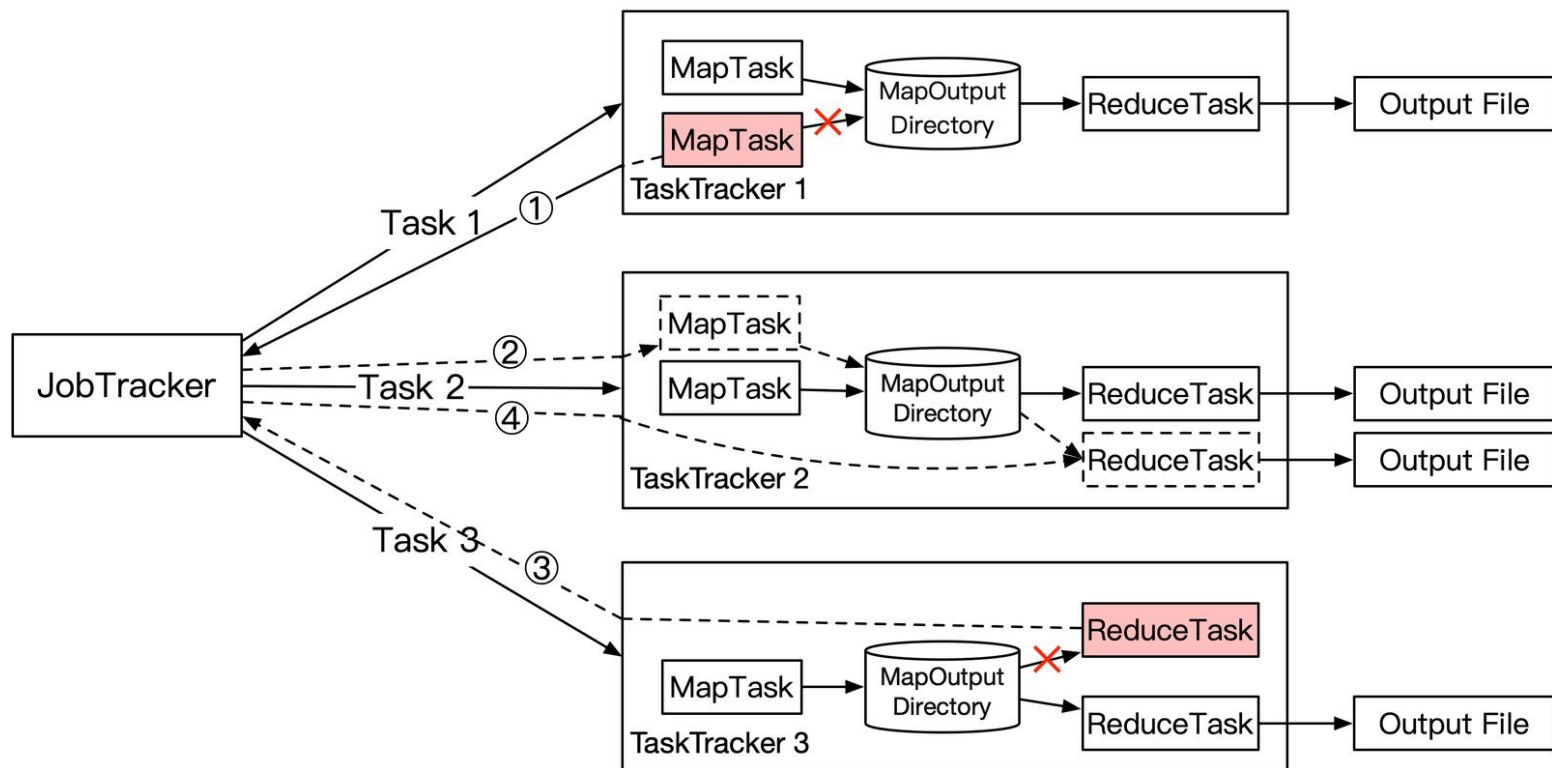
- 当故障发生在不同阶段时的恢复机制：



- ①、②：TaskTracker1在运行一个MapTask时出现故障，MapReduce框架会尝试重新启动失败的MapTask，重新分配任务到可用的节点。

硬件故障

- 当故障发生在不同阶段时的恢复机制：



- ③、④：在TaskTracker3上运行一个ReduceTask时出现故障，MapReduce框架会尝试重新启动失败的ReduceTask，将失败的任务分配给另一个可用的TaskTracker2执行，或在集群中的其他节点上启动新的Reduce任务。

任务失败

- 当用户代码缺陷导致任务在运行过程中抛出异常时，任务进程会主动退出，并向TaskTracker父进程发送错误消息，TaskTracker将该任务尝试标记为失败。
- 对于进程崩溃故障，TaskTracker的监听程序会察觉到进程的退出。
- 当程序陷入死循环时，由于TaskTracker没有接收到进度更新，该任务尝试同样会被标记为失败。

任务失败

- TaskTracker标记任务尝试失败后，会将其自身的任务计数器减1，以便向JobTracker请求新的任务。
- 通过心跳机制，TaskTracker向JobTracker报告本地任务尝试的失败情况。
- JobTracker接收到任务失败的通知后，重置任务状态，将其重新加入调度队列以再次分配给其他TaskTracker运行。
- JobTracker会尽力避免将失败的任务再次分配给已经运行失败的TaskTracker。
- 假设某个任务尝试了一定次数（可以根据需求进行设置），但仍未能成功完成，整个作业就会被标记为失败。

小节：分布式批处理系统

- 任务分配与协调
 - 近数据任务分配，轮询分配，加权轮询分配，DRF，Max-Min
 - 同步/异步计算，计算图机制和预测执行机制
- 故障检测与恢复
 - 心跳检测，Gossip风格故障检测
 - 检查点机制，消息日志，中间状态持久化，故障转移
- 批处理系统——MapReduce
 - 编程模型，基本架构，故障恢复

第七章 完