

└ 第1章 实例	2	└ 第16讲 图数据库neo4j	85
└ 第2章 Message和Kafka	6	└ 第17讲 LSM-VectorDB	90
└ 第3章 websocket	10	└ 第18讲 influxDB	96
└ 第4章 transaction事务	13	└ 第19讲 gaussDB	101
└ 第5讲 事务管理	20	└ 第20讲 datalake	107
└ 第6章 多线程	26	└ 第21讲 cluster	109
└ 第7章 缓存	35	└ 第22讲 cloud computing	112
└ 第8章 全文搜索	42	└ 第23讲 graphQL	117
└ 第9章 web服务	47	└ 第24讲 docker	119
└ 第10章 微服务	52	└ 第25讲 hadoop	125
└ 第10章 总结summary	61	└ 第26讲 spark	129
└ 第11章 mysql优化1	63	└ 第27讲 storm	136
└ 第12章 mysql优化2	71	└ 第28讲 HDFS	141
└ 第13章 mysql备份与恢复	74	└ 第29讲 HBase	147
└ 第14讲 mysql分区	78	└ 第30讲 Hive	151
└ 第15讲 mongoDB	81	└ 第31讲 flink	156

# 应用系统体系架构-2023年秋季

## 1. 实例

### 1.1 填坑：Cookie和Session

核心问题是什么呢？

**Http协议是一种无状态的协议！**也就是说，每次请求都是独立的，服务器并不知道你是谁，你上次请求的信息是什么。

所以，怎么解决这个问题呢？

- **Cookie：**客户端浏览器用来保存用户信息的一种机制；当我们通过浏览器进行网页访问的时候，服务器会将一些数据以cookie的形式保存在客户端浏览器上，当下次客户端浏览器再次访问该网站时，会将cookie数据发送给服务器，服务器通过cookie数据来辨别用户身份。（cookie存的是key-value键值对）
- **Session：**表示一个会话，是属于服务器端的一种容器对象；默认情况下，针对每个浏览器的请求，server都会创建一个session对象，生成一个sessionId，用于标识该session对象，同时将sessionId以cookie的形式发送给客户端浏览器，客户端浏览器再次访问该网站时，会将cookie数据发送给服务器，服务器通过cookie数据来辨别用户身份，从而找到对应的session对象，如果找不到，就会创建一个新的session对象。简单来说，**Session的作用就是帮助我们实现一个有状态的Http协议。**

### 1.2 实例池（对象池）

"对象池"（Object Pool）是一种设计模式，它是一种用于**管理和重用对象实例**的机制，以提高性能和资源利用率的方式。对象池通常用于减少创建和销毁对象的开销，特别是在对象的创建成本较高或频繁创建和销毁对象可能导致性能下降的情况下。

在Java中，对象池通常是一个集合，用于存储和管理多个对象实例。当需要使用对象时，可以从对象池中获取一个可用的对象，而不是每次都创建新的对象。一旦使用完成，可以将对象返回到对象池中，以便稍后重用，而不是立即销毁它。

- 实例池的数量一定有上限的，不可能运行每一个用户上来都能创建一个对象，否则DDos攻击的时候内存直接爆炸。
- 假设我们只能创建两个对象，这样内存就不会爆炸了。那是怎么服务于多个用户呢？

- 假设A用户来了，我们创建一个实例A，然后B来了创建一个B，现在实例池满了。这些实例包括数据现在所在位置都是内存里面！
- C来了之后，根据最近最少使用的算法，把A从内存里面换出，落到硬盘上面，然后创建一个C的实例。
- 这样就哪怕限制了实例的数量，也可以服务多个用户。

因此，**系统尽量要无状态的，避免有状态的服务。**

1. 比如说我们要统计一个网站在线用户的数量，这样所有用户公用的一个域就是count：反应用户数量，这样就是无状态的。（这个状态是公用的，不是每个用户都有一个）
2. 如果是每个人来了以后，每个人的对象都不一样，那么就是有状态的。但是有状态的就需要**针对每个用户单独存储**，占用空间，所以尽可能的少或者避免。

所以，我们在spring中如何控制实例数量呢？

## 1.3 Scope：实例对象的数量控制

在Spring框架中，“scope”（作用域）用于定义Spring容器如何管理bean实例的生命周期和可见性。Spring支持不同的bean作用域；可以根据应用程序的需求选择适当的作用域。以下是一些常见的Spring作用域：

1. **Singleton (单例)**：这是Spring默认的作用域。在单例作用域下，Spring容器只创建一个bean实例，并在应用程序的整个生命周期内重用该实例。这意味着每次请求该bean时，都会返回相同的实例。
2. **Prototype (原型)**：在原型作用域下，每次请求bean时，Spring容器都会创建一个新的bean实例。这意味着每次请求都会返回一个不同的实例。
3. **Request (请求)**：这个作用域适用于Web应用程序，每个HTTP请求都会创建一个新的bean实例，每个请求之间的实例不共享。
4. **Session (会话)**：类似于请求作用域，但在HTTP会话的整个生命周期内创建和维护一个bean实例。不同用户的会话之间的实例不共享。
5. **Global Session (全局会话)**：仅在分布式Web应用程序中使用，它与会话作用域类似，但在集群环境中，全局会话在多个节点之间共享。
6. **Custom (自定义)**：您还可以定义自定义的作用域，以满足特定需求。要使用自定义作用域，您需要实现Spring的 `org.springframework.beans.factory.config.Scope` 接口，并将其配置到Spring容器中。

关于scope的理解，详见作业1-计时器。

**tips：**两个浏览器就是两个进程；两个进程是相互隔离的；所以我们用两个浏览器来模拟两个客户端。

## 1.4 数据库连接的数量

数据库的连接也是池化的！（刚才的实例池是对象池，和这里的连接池不是一个东西！）

连接池（Connection Pool）是一种用于管理和重用数据库连接、网络连接或其他资源连接的技术，旨在提高应用程序性能和资源利用率。**连接池通过维护一组已创建的连接实例，并在需要时分配这些连接，以减少创建和销毁连接的开销。**

这里的连接池本质是一个线程池，里面是一大堆线程。

以下是连接池的工作原理和好处：

### 工作原理：

1. **初始化连接池：** 在应用程序启动时，连接池会初始化一定数量的连接实例，这些连接可以立即使用。
2. **连接分配：** 当应用程序需要使用连接时，它向连接池请求一个连接。连接池会检查是否有可用的连接实例，如果有，则分配一个给应用程序。
3. **连接使用：** 应用程序使用连接执行数据库查询、网络通信或其他操作。
4. **连接释放：** 当应用程序完成连接的使用时，它将连接释放回连接池，而不是立即关闭连接。连接池会重新标记这个连接为可用状态。

假如要支持10万用户，需要在连接池里面配置多少数据库连接？

连接池的数量：

$$connections = ((core\_count * 2) + effective\_spindle\_count)$$

注：

1. *core\_count*: CPU核心数
2. *effective\_spindle\_count*: 有效磁盘数

为什么：

- 线程来回切换过程需要时间，假如配很多的数据库连接，会导致很多的线程切换的时间，反而降低效率
- 让数据库连接的数量和CPU核心的数量差不多或者两倍（超线程）的时候，这样线程上下文切换的时候就会少，比较好
- 因为CPU处理完之后，在把内存的东西写硬盘的时候写的过程，CPU是空闲的时候，所以还加上有效的硬盘的数量。

所以假如4核心2硬盘的服务器，就只需要十个数据库连接，就可以为十万的用户服务了！

为什么呢：

- 连接池看的不是客户端的数量，而是机器硬件的配置，让需要获取连接的线程在那里等待，然后逐一处理即可。

那如果有100万用户的时候该怎么办？

- 这时候不应该增加数据库连接的数量，而是应该增加机器的数量，每个机器上有一定数量的数据库的连接！

# 第2章 Message和Kafka

## 1. 同步客户端-服务器模型

- 最早的通信全部都是同步的的通信，比如系统A和系统B之间通信，需要两个系统提供API
- 同步的通信需要客户端一直等待相应，但是可能需要等很久
- 当需要通讯的模块多了之后，不然四五个通讯系统的时候都要交互的时候，比如A系统要对B、C、D都开发一套API，这很麻烦。API的数量会大量增加
- 紧耦合，一旦有一个地方的函数参数或者返回类型要修改，大量的代码都需要修改，非常麻烦。
- 假如有一个系统暂时挂掉了，能不能反复多尝试几次呢——传统的方法不行
- 通讯不可靠，过于依赖请求响应的模式。
- 没有缓冲区。

## 2. 异步客户端-服务器模型

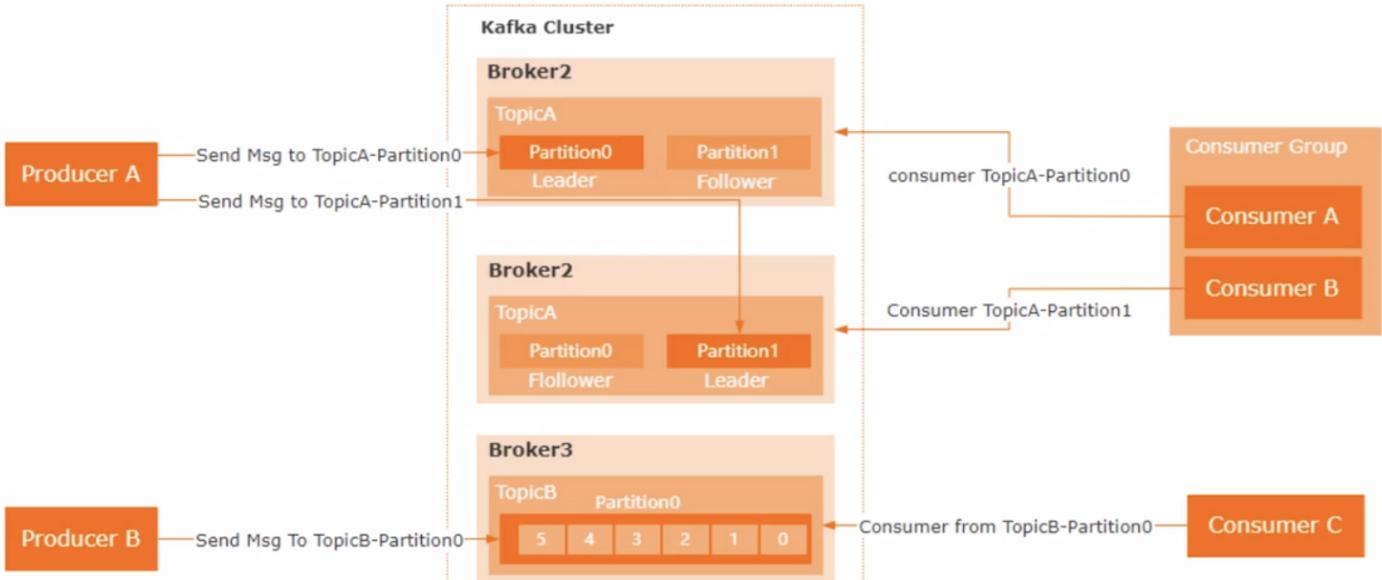
- 在所有的系统里面有一个通讯的媒介，系统不直接交互，而是通过中介来通讯。
- 系统之间约定统一的通讯模式，topic相当于信箱，每次通讯的时候就把信息往信箱里面发，然后另外一个系统可能会监听这个信箱，一旦有新的内容接受，就开始处理
- 站在Kafka消息中间件的角度，所有系统都是对等的，不存在客户端-服务器的模型，所有的系统都可以收发信息。
- 实现解耦，哪怕有一个系统崩了，消息中间件会多次发送

## 3. Kafka

Kafka是一种消息队列；主要用来处理大量数据状态下的消息队列，一般用来做日志的处理。

Kafka的作用有：

- 解耦合
- 异步处理
- 流量削峰



然后来看一些关于kafka的基本概念：

1. broker: kafka集群包含一个或多个服务器，这种服务器被称为broker
2. producer: 负责发布消息到kafka broker
3. consumer: 消息消费者，向kafka broker读取消息的客户端

然后往broker里面细看，我们能看到topic, topic里面有很多partition；我们来一个一个解释：

1. topic: 可以理解为一个队列，生产者和消费者都是面向一个topic
2. partition: 为了实现扩展性，一个非常大的topic可以分布到多个broker上，一个topic可以分为多个partition，每个partition是一个有序的队列
3. leader: 每个partition多个副本的主角色，生产者发送数据的对象，以及消费者消费数据的对象都是leader
4. follower: 每个partition多个副本的从角色，实时的从leader中同步数据，保持和leader数据的同步，leader发生故障的时候，某个follower会成为新的leader

### 概念：Topic

在Apache Kafka中，**Topic (主题)** 是一种用于组织和分类消息的基本单元。Topic是Kafka消息系统中的重要概念，它可以看作是一个消息类别或通道，用于将消息进行逻辑上的划分和分类。

以下是关于Kafka Topic的一些关键特点和作用：

1. **消息分类**: Topic允许你将消息分组成不同的类别或主题，每个主题都有一个唯一的名称。这有助于组织和管理消息，使得消息可以根据其内容或用途进行逻辑上的分类。
2. **发布与订阅**: 生产者 (Producer) 将消息发布到一个或多个Topic，而消费者 (Consumer) 可以订阅一个或多个Topic以接收相关的消息。这种发布-订阅模式使得消息的发送和接收可以相互独立，生产者和消费者不需要直接通信。

3. **多订阅者**: 一个Topic可以有多个订阅者，这意味着多个消费者可以同时订阅相同的Topic并独立地处理消息。这种多订阅者模式支持并行和分布式处理。
4. **消息保留**: Kafka支持消息保留策略，允许你配置消息在Topic中保留的时间。这样，消费者可以消费最新的消息，而旧的消息会根据策略自动删除。
5. **分区**: 每个Topic可以分为多个分区，每个分区可以视为该主题的一个子集。分区允许Kafka在不同的节点上并行处理消息，从而提高了可伸缩性和性能。
6. **消息顺序性**: Kafka保证在单个分区内的消息是有序的，这意味着如果消息按照一定的顺序发送到同一个分区，它们将以相同的顺序被消费。

总之，Kafka中的Topic是一种用于组织和管理消息的重要机制，它允许将消息分类、分发、保留和处理。每个Topic都有一个唯一的名称，允许多个生产者和消费者在分布式环境中高效地交换数据。Kafka的分区机制使得消息处理具有高度的可伸缩性和性能。

## 3.1 Kafka的原理

- kafka的数据存储在log里面，数据往文件后面不断追加，追加的速度更快，如果是随机访问反而效率低
- log里面都是有序的事件，每个事件都有位移，偏移量
- 当然有很多个消费者，把消费者分成不同的组。当一个消息被所有的消费者读走了之后，这个消息才会被删除掉；多个用户可以读取同一个log，并且维护他们各自的文件位置（都到哪里了）
- 这样就能保证log文件不可能无限制的增长

## 3.2 Kafka的数据类型

- Topic里面存储的都是同构的数据类型，数据类型统一（访问效率高，推导出偏移量为X的时候直接就可以推出）
- Topic里面存储的都是异构的数据类型，数据类型不一样（访问效率低，不好推出某个偏移量的消息）
- Topic可能会分成几个Partition，假设某个Topic非常大，就可以分区。最终会使用某种hash算法，kafka自己会决定放到哪一个具体的区域里面。
- 分的区域越多，管理的复杂性增加，但是并行处理，也增加了处理的效率。
- 卡夫卡的消息（event）是一个带有时间戳的key-value对，他会记录一些发生的事情。key可以作为分类的依据，value最好不要作为分类的依据。此外，key也不是必须的，value

## 3.3 集群的容灾

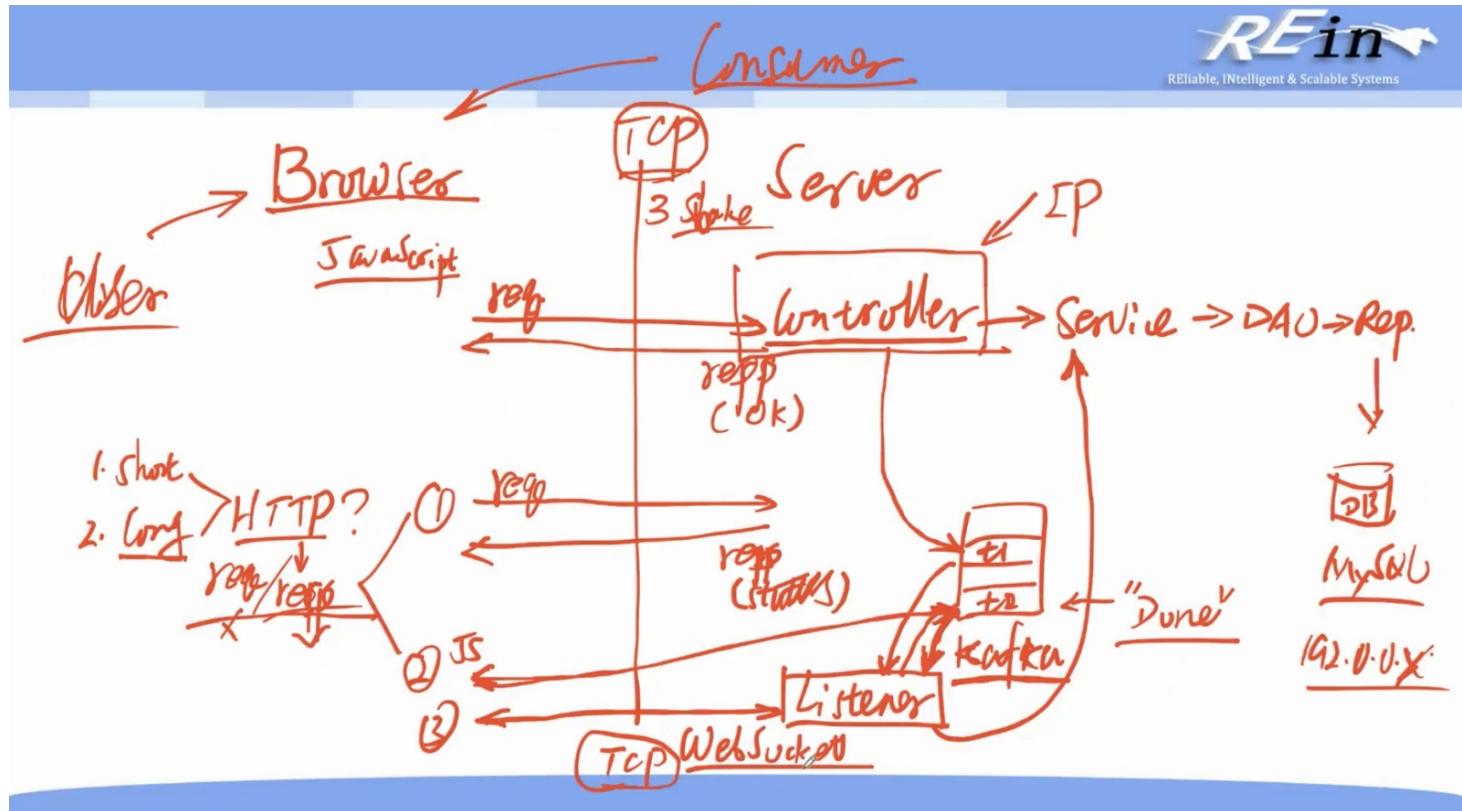
- 用空间换可靠性，比如每个数据存储两个副本，这样比如有一个卡夫卡服务器崩溃了，还有一个备份能够使用
- 需要一个协调器在卡夫卡集群里面，比如有多个消费者订阅了集群的topic，他们都在处理topic里面的内容。consume会不断的给协调器发心跳包，表明自己正在处理，如果某个consumer不发了说

明故障了，协调器就不会把消息发给故障的。

# 第3章 WebSocket

2023.09.17

我们先来看一下目前的一整套流程逻辑：



## 3.1 前端获得数据的方式

### 3.1.1 Basic ways

首先我们来看两种最基本的方式：

- 在前端工程中，使用JavaScript脚本监听后端kafka的存储订单处理结果消息的Topic；
  - 优点就是直接交互，简洁明了，降低了后端Spring的压力(不需要单独写一个接口，单独运行相关的Kafka查询组件的信息)。
  - 缺点也是直接交互，使用JS监听，相当于用户客户端直接和信箱交互，这就会导致我的**卡夫卡 Topic信箱直接暴露**。卡夫卡信箱里面涉及到用户的订单数据，所以使用后端直接交互，然后后端Spring暴露访问接口，更加安全。
- 在前端发送Ajax请求获取订单的最新状态，后端接收到请求后将订单状态返回给前端去显示；
  - 如果前端轮询，消耗资源和时间。

前面两种都是所谓的“请求-响应”模式，因为我们走的都是Http协议。  
websocket就是要突破这种模式，产生“双工”——谁来发起都可以。

**websocket是一种应用层的协议，和http协议一样，都是基于TCP（传输层）的协议。**

### 3.1.2 WebSocket

后端的消息监听器类监听到消息处理结果Topic中的消息后，通过WebSocket发送给前端。

- 优点：
  - 推送功能：支持服务器端向客户端推送功能。服务器可以直接发送数据而不用等待客户端的请求。
  - 减少通信量：只要建立起websocket连接，就一直保持连接，在此期间可以源源不断的传送消息，直到关闭请求。也就避免了HTTP的非状态性。和http相比，不但每次连接时的总开销减少了，而且websocket的头部信息量也小，通信量也减少了。
  - 减少资源消耗：如果用AJax轮询的话，我们需要专门设置一个接口，运行相关的查询代码，而且由于前端是定时的不断的发请求来查询，相关的查询结果的代码要运行很多次，这浪费了资源，反而如果只用WebSocket，推送代码只用运行一次。
- 缺点
  - 比如需要浏览器支持WebSocket，例如我的Safari浏览器在WebSocket的测试中出现了一些问题（由于一些安全性的原因，但是Edge浏览器就可以正常保证WebSocket的连接），同时如果只是单页面涉及到WebSocket还好，涉及到多页面，定时推送，复杂的推送，就非常容易出问题了，不管是前端，还是后端都会遇到一些问题。

## 3.2 WebSocket的具体使用

### 3.2.1 一些概念

#### 1. endpoint

在WebSocket中，**endpoint** 是指WebSocket通信的目标或终点。它是WebSocket连接的另一端，可以是服务器或客户端，用于接收或发送WebSocket消息。

在WebSocket通信中，有两种类型的endpoint：

1. **WebSocket服务器端(endpoint)**：这是WebSocket通信的服务器端，它等待客户端的连接请求并处理它们。一旦连接建立，服务器端就会监听来自客户端的消息，并可以向客户端发送消息。  
WebSocket服务器通常用于实时通信、在线游戏、聊天应用等场景。
2. **WebSocket客户端(endpoint)**：这是WebSocket通信的客户端，它与WebSocket服务器建立连接，并可以向服务器发送消息，同时接收服务器发送的消息。WebSocket客户端通常用于与服务器

进行双向通信，以获取实时数据或与服务器进行交互。

无论是服务器端还是客户端，endpoint都是WebSocket通信的重要组成部分，它们通过WebSocket协议进行通信，实现了实时、双向的数据交换。WebSocket协议允许这些endpoint之间建立持久性连接，以便在不断开连接的情况下传输数据。这使得WebSocket在实时应用程序中非常有用，因为它减少了与HTTP请求/响应模型相关的开销，并提供了更快的数据传输和更低的延迟。

## 2. 消息代理 (Message Broker)

在WebSocket通信中，**消息代理** 是指WebSocket服务器和客户端之间的中间件，用于转发WebSocket消息。它充当了WebSocket服务器和客户端之间的桥梁，使它们可以相互通信。消息代理可以是一个独立的服务器，也可以是WebSocket服务器本身。

# 第4章 Transaction Management 事务管理

2023.09.16

- What is a transaction? 事务的概念
- container-managed transactions 容器管理的事务
- isolation and database locking 隔离和数据库锁
- updating multiple databases 更新多个数据库

## 1. 事务的概念

转账：

1. 把钱从第一个人的账户里面扣除
2. 把钱加到第二个人的账户里面

如果第一个动作成功，第二个动作失败，我们就需要回滚第一个动作；这两个动作在我看来是**原子性的**。

本质上就是你要告诉我，事务从哪里开始，到哪里结束？

转账：

1. A和B同时转账给C，转10元；C原本有100元
2. 如果是同时转账，结果应该是120，但实际可能是110——事务的**隔离性**

### 定义

在java EE应用程序中，事务是一系列必须全部成功完成的操作，否则每个操作中的所有更改都将被撤消。事务以提交或回滚结束。

### 事务的四个特性（ACID）

1. 原子性（Atomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。
2. 一致性（Consistency）：在事务开始和完成时，数据都必须保持一致状态。
3. 隔离性（Isolation）：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。
4. 持久性（Durability）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

从编程的角度来说，我们要考虑的就是**原子性**和**隔离性**。剩下两件事交给数据库就好。

在spring中，我们都希望以 annotation (声明) 的方式来实现事务管理（而不是自己编码）。

## 2. 容器管理的事务

在Spring中，容器管理事务是指Spring容器负责管理应用程序的事务生命周期，以确保事务的正确执行和一致性。它提供了一种声明式的、基于配置的方式来管理事务，使得开发人员可以将精力集中在业务逻辑上，而不必关心事务的细节。

### 2.1 事务传播行为

- `@Transaction(propagation = Propagation.REQUIRED)`：如果当前线程上面有事务就使用当前的事务，如果没有就创建一个事务；注意！在进入和退出方法的时候都会做检查！
- `@Transaction(propagation = Propagation.NOT_SUPPORT)`：容器不为这个方法开启事务；你有事务就挂起，你没事务就算了
- `@Transaction(propagation = Propagation.REQUIRED_NEW)`：不管是否存在事务，都创建一个新的事务，原来的挂起，新的执行完毕之后，继续执行老的事务
- `@Transaction(propagation = Propagation.MANDATORY)`：必须在一个已有的事务里面执行，否则抛出异常
- `@Transaction(propagation = Propagation.NEVER)`：必须在一个没有的事务里面执行，否则抛出异常
- `@Transaction(propagation = Propagation.SUPPORT)`：如果其他的bean调用这个方法，在其他的bean里面声明了事务就使用事务，如果其他的bean里面没有声明事务，那就不用事务（主打一个佛系）

#### 举例

1. 在下面的例子中，如果 `method1` 和 `method2` 都是 REQUIRED，那么 `method2` 会加入到 `method1` 的事务中，如果 `method2` 抛出异常，那么 `method1` 也会回滚。
2. 当程序执行完 `method2` 的时候，还会做一次检查，发现这个事务就不是你开启的，所以就不会提交。当 `method1` 执行完毕的时候，才会提交。
3. 如果 `method2` 上面的注解是 REQUIRES\_NEW，那么 `method2` 就会开启一个新的事务，`method1` 和 `method2` 就是两个事务了。`method2` 结束的时候，它就直接把自己的事务提交掉了。**如果 `method2` 抛出异常，那么 `method1` 的事务不会回滚。**

```
// bean1
@Transactional(propagation = Propagation.REQUIRED)
public void method1() {
    // do something
    bean2.method2();
    // do something
}

// bean2
@Transactional(propagation = Propagation.REQUIRED)
public void method2() {
    // do something
}
```

## 2.1.1 回滚

spring如何回滚呢？

- 抛出 `RuntimeException` 或者 `Error` 的时候，spring会自动回滚

```

// BankServiceImpl.java
@Service
public class BankServiceImpl implements BankService {
    @Autowired
    private BankDao bankDao;

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void transfer(String from, String to, int money) {
        bankDao.withdraw(from, money);

        // int result = 10 / 0;      // 1. 在deposit之前

        try {
            bankDao.deposit(to, money);
        } catch (Exception e) {
            e.printStackTrace();      // 打印异常
        }

        // int result = 10 / 0;      // 2. 在deposit之后
    }

    // BankDaoImpl.java
    @Repository
    public class BankDaoImpl implements BankDao {
        @Autowired
        private BankRepository bankRepository;

        @Transactional(propagation = Propagation.REQUIRED)
        public void withdraw(String from, int money) {
            Bank bank = bankRepository.findById(from).get();
            bank.setMoney(bank.getMoney() - money);
            bankRepository.save(bank);

            // int result = 10 / 0;
        }

        @Transactional(propagation = Propagation.REQUIRED)
        public void deposit(String to, int money) {
            Bank bank = bankRepository.findById(to).get();
            bank.setMoney(bank.getMoney() + money);
            bankRepository.save(bank);
        }
    }
}

```

```

    // int result = 10 / 0;
}

}

```

<b>id</b>	<b>transfer</b>	<b>withdraw</b>	<b>deposit</b>	<b>result</b>
1	result = 10/0 在deposit之前	正常	正常 REQUIRES_NEW	整个事务回滚
2	result = 10/0 在deposit之后	正常	正常 REQUIRES_NEW	deposit成功, withdraw和transfer回滚
3	正常	result = 10/0	正常 REQUIRES_NEW	整个事务回滚 (因为withdraw抛出异常, 一路抛上去没人接收, 所以整个事务回滚)
4	正常	正常	result = 10/0 REQUIRES_NEW	withdraw和transfer成功, deposit回滚 (因为deposit扔出去的错误被catch了, 所以deposit回滚, 剩下两个没事)

## 2.2 事务隔离级别

要理解事务隔离级别，我们需要先了解我们会碰到的一些问题。

### 2.2.1 一些问题

- 脏读：**事务T1将某一值修改，然后事务T2读取该值，此后T1因为某种原因撤销对该值的修改，这就导致了T2所读取到的数据是无效的。这就是脏读。
- 不可重复读：**事务T1读取某一数据，事务T2读取并修改了该数据，T1为了对读取值进行检验而再次读取该数据，发现得到了不同的结果。（我把数据读走之后锁死了，不让你动）
- 幻读：**事务A读取值，把读取走的数据锁死了，但是事务B往里面添加了几行数据，这就导致事务A再次读取的时候发现可能多了几行符合条件的。就像幻影一样。（这个时候要锁的不仅仅是你读走的数据，要锁的是整个表格；应该在这个事务的执行期间，我们按照同一个逻辑，读到的东西是相同的）

这三个问题都是事务没有隔离好导致的；综上需要权衡，在性能和隔离程度直接选择。

### 2.2.2 锁！

- 读锁：**读取锁防止其他事务在事务结束之前更改事务期间读取的数据，从而防止不可重复的读取。  
**其他事务可以读取数据，但不能写入数据。当前事务也被禁止进行更改。**

- **写锁**: 写入锁用于更新。写锁防止其他事务在当前事务完成之前更改数据，但**允许其他事务和当前事务本身进行脏读取**。换句话说，事务可以读取自己未提交的更改。（第一种事务隔离就是解决这个问题）
- **独占写锁**: 独占写入锁用于更新。防止其他事务读取或更改数据，直到当前事务完成。它还可以防止其他事务的脏读取。

还有什么处理concurrency的方法呢？

离线锁：所谓“离线”指的是flush之前，数据读出来之后，放在了tomcat的内存里；这个时候是离线的；所谓“在线”，就是这个对象实时反映了数据库的数据；但显然OR映射/JDBC都是离线的。

- **乐观离线锁**: 保持乐观的状态，默认不会出现冲突（所以数据库就不加锁）。用户写入数据的时候，检查一下读取的数据是否发生了改变，如果发现不一致，那就是说明用户是基于陈旧的数据在进行修改，就会抛出异常。（实现方法：增加一个版本号，表里面加一列，更新之后版本号码加1，校验的时候检查版本号码是否有问题，当然也可以用时间戳）
- **悲观离线锁**: 对于冲突这个事情保持悲观太多，也就是认为冲突的概率非常高。当有用户来获取Session的时候就会获取锁，一旦有人在读取或者修改，就无法获取锁。
- **粗粒锁Coarse-Grained Lock** : 一次性用一个锁，把根数据相关的，比如外键关联的所有数据全部锁死。实现原理：两组数据通过引用共享同一个版本号码，**通过引用的方法，两个数据引用的是同一个版本号码**（所谓的版本号不是一个数字，而是一个**对象**，这个对象里面有一个数字属性，这个数字就是版本号码）。

## 2.2.2 隔离级别

- `@Transaction(isolation = Isolation.READ_UNCOMMITTED)` : 读取未提交的数据（说穿了就是**啥也不干**，上面三个问题都有）（用的少，因为实在是太垃圾了）
- `@Transaction(isolation = Isolation.READ_COMMITTED)` : 读取已经提交的数据（**解决了脏读**，会出现不可重复读还有幻读）SQLSERVER默认。因为读取已提交的数据，未提交的数据不能读，所以避免了脏读的问题。
- `@Transaction(isolation = Isolation.REPEATABLE_READ)` : 可重复性读（会出现幻读）MYSQL默认。因为**读走的数据加了锁（行锁）**，这样别的数据就不能改，避免了不可重复性读的问题。但是还是会出现幻读，因为可能会有别的事务添加几行新数据，这就导致可能两次相同的读取在后一次读取的时候发现多出来几行
- `@Transaction(isolation = Isolation.SERIALIZABLE)` : 串行化执行，最严格的级别。**把整个表格锁死**，幻读也可以避免了！（用的少，因为这样几乎就没有并发了）

## 2.2.3 更新多个数据库——分布式事务

只要不是同一个数据库，就是分布式事务。

**多数据源：两阶段提交协议**

1. prepare阶段：去问！（数据库A能不能提交？数据库B能不能提交？）
2. commit/rollback阶段：根据prepare阶段的结果，决定commit还是rollback

## 实现

通过 @Transactional 注解的 transactionManager 属性指定事务管理器，从而实现多数据源的事务管理。

# 第5讲 事务管理

2023.12.30

不开心就不开心  
也别勉强的慰问  
但求随着我的心  
洒脱地尊重我的伤感  
别要不开心便找开心  
去避过我的良心  
消化忧郁后  
才令我拾回自信心  
——《开不了心》陈奕迅

上节课我们关注的点在于应用的角度（比如你要写一个spring），你如何管理事务；但是事务最终我们还是要落实到数据库层面，因为数据库是事务的最终实现者。

## 1. 可串行化调度

调度：事务的并发过程中，决定事务中每个操作的执行顺序。

□ 串行调度(serial schedule): 事务串行执行

□ 调度定义下的并发控制：

- 给定一个并发调度  $S$ ，存在一个串行调度  $S'$ ，在任何数据库状态下，**按照调度  $S$  和 调度  $S'$  执行后所产生的结果都是相同的**。
- 此时调度  $S$  被称之为**可串行化调度** (serializable schedule)。



# 冲突可串行化调度



□ 定义：冲突可串行化调度，是从冲突的角度出发，针对一个调度  $S$  去发现其等价的串行调度  $S'$  来确定  $S$  是一个可串行化调度。

- 一次操作交换 定义为交换事务调度序列中相邻的两个操作，一个交换操作可以将一个调度A变成另外一个调度B。
- 当交换不会影响两个调度的一致性时，定义该交换得到的两个调度是等价的，该交换为等价交换。
  - 等价操作
    - 交换连续两个相同数据读取操作的顺序:  $R_{T1}(A) R_{T2}(A)$
    - 交换连续两个不同数据读写操作的顺序:  $R_{T1}(A) W_{T2}(B); W_{T1}(A) R_{T2}(B); W_{T1}(A) W_{T2}(B)$
  - 非等价操作
    - 交换连续两个相同数据的读写、写写操作:  $R_{T1}(A) W_{T2}(A); W_{T1}(A) R_{T2}(A); W_{T1}(A) W_{T2}(A)$

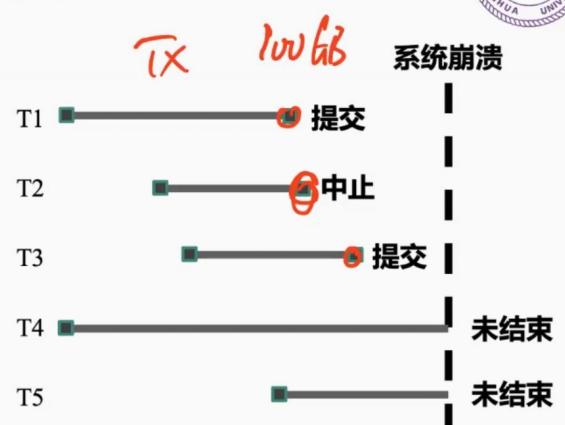
## 2. 系统崩溃的恢复



### 系统崩溃恢复



- 脏页：内存页面已更新，磁盘页面未更新
- 刷脏：将内存脏页刷到磁盘
- T1、T3在崩溃时刻前已经提交
  - 如果未刷脏，则影响持久性，需要重做
  - 如果已刷脏，则不影响持久性
- T2在崩溃时刻前已经中止（已经完成了回滚）
  - 如果期间刷过脏，但是abort时未刷脏，则影响持久性，需要重做
  - 如果已刷脏，则不影响持久性
- T4、T5在崩溃时刻前未结束
  - 如果已刷脏，则影响原子性，需要回滚
  - 如果未刷脏，则不影响原子性



	已刷脏	未刷脏
提交Commit	😊	重做 redo
中止Abort	😊	重做 undo
未完成	回滚	😊

事务很大，在过程中会不断刷到硬盘，因此中止崩溃会有undo。

来点CSE：其实这就是cse讲的all or nothing atomicity。



# 系统崩溃下的事务



## 已完成的事务

- 已提交的事务 (Commit标记) : 它们对数据库的修改可能没有写回磁盘, 缓冲区数据丢失后这些修改无法找回。事务的持久性受到了影响, 需要重做这些事务。
- 已中止的事务 (Abort标记) : 系统对这些事务的撤销可能没有写回磁盘, 因此在重启之后这些撤销内容会丢失。事务的持久性受到了影响, 需要重做这些事务。

**未完成的事务 (只有Start, 没有Commit、Abort标记) :** 它们可能已经对数据库造成了修改, 但是没有被系统提交, 重启之后的数据库也没有撤销这些修改。事务的原子性受到了影响, 需要回滚这些事务



# 崩溃恢复策略设计



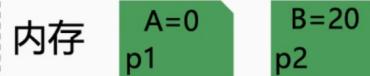
## 原子性保证

- 选择①: NO-STEAL (非窃取)
  - 未结束事务不能将脏页写入磁盘, 不存在原子性问题
- 选择②: STEAL (窃取)
  - 未结束事务能将脏页写入磁盘, 影响原子性, 需要回滚

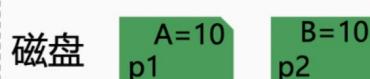
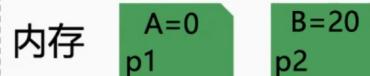
## 持久性保证

- 选择①: Force (强制)
  - 已完成事务强制将脏页写入磁盘, 不存在持久性问题
- 选择②: No-Force (非强制)
  - 已完成事务不强制将脏页写入磁盘, 影响持久性, 需要重做

初始: A=10; B=10  
T: A=A-10, B=B+10



初始: A=10; B=10  
T: A=A-10, B=B+10





# 保证持久性和原子性的方案选择



## □ FORCE条件：事务完成强制刷脏 vs 不强制刷脏

- Force缺点：每次事务提交都必须刷新脏页，消耗大量IO读写资源
- 解决方法：使用NO-FORCE模式，利用Redo日志重做事务
- Redo日志：记录事务对数据库的所有影响

## □ NO-STEAL条件：事务中间可以刷脏 vs 不可以刷脏

- No-Steal缺点：事务执行过程中不能刷新磁盘，必须占有较大的缓冲区空间，不利于多个事务的并发执行
- 解决方法：使用STEAL模式，利用Undo日志撤销事务
- Undo日志：记录撤销事务所需的内容

## 3. 日志记录方案

### 逻辑日志：

- 记录事务中高层抽象的逻辑操作
- 举例：记录日志中UPDATE,DELETE,INSERT的文本信息；例如小明的年龄由20改为21

### 物理日志：

- 记录数据库具体物理变化
- 距离：记录一个被查询影响的数据前后的值
  - 例如第10个页面第100偏移量的值由20改为21

### 物理逻辑日志

- 一种结合了物理日志和逻辑日志的混合方法
- 日志记录中包含了数据页面的物理信息，但是页面以内目标数据项的修改信息则是以逻辑方式记录
  - 例如第100个页面（物理）的小明年龄值由20改为21（逻辑）



# 三种日志对比



**UPDATE Student SET Sname= "Mike" WHERE Sno = "1" ;**

物理日志

```
< T1,  
Table= Student,  
Page=99,  
Offset=4,  
Before=James,  
After=Mike >  
  
< T1,  
Index=X_PKEY,  
Page=45,  
Offset=9,  
Key=(1,Record1) >
```

逻辑日志

```
< T1,  
Query= "UPDATE  
Student SET Sname  
= Mike WHERE Sno =  
1" >
```

物理逻辑日志

```
< T1,  
Table= Student,  
Page=99,  
ObjectId=1,  
Before=James,  
After=Mike >  
  
< T1,  
Index=X_PKEY,  
IndexPage=45,  
Key=(1,Record1) >
```



## 物理/逻辑日志性质比较



### 口有关数据库日志的三个重要性质：

- 幂等性：一条日志记录无论执行一次或多次，得到的结果都是一致的。
  - 例如： $x=x+1$ 不幂等； $x=0$ 幂等
  - 物理日志满足幂等性；逻辑日志不满足
- 失败可重做性：一条日志执行失败后，是否可以重做一遍达成恢复目的。
  - 例如：插入一条记录失败，再次插入成功。
  - 物理日志满足失败可重做性；逻辑日志不满足：例如插入数据页面成功，而插入索引失败，重做插入这个逻辑日志失败。
- 操作可逆性：逆向执行日志记录的操作，可以恢复原来状态（未执行这批操作时的状态）
  - 例如第10个页面第100偏移量的值由20改成21，逆操作由21改成20
  - 物理日志不可逆（页面偏移量位置可能被后续记录修改），逻辑日志可逆。

redo日志应该使用物理日志：因为要求幂等性！

undo日志应该使用逻辑日志：为什么不用另一个？因为操作是不可逆的

	解析速度	日志量	可重做性	幂等性	可逆性	应用场景
物理日志	快	大	是	是	否	重做日志
逻辑日志	慢	小	否	否	是	撤销日志
物理逻辑日志	较快	中	是	否	否	撤销日志

**注：物理逻辑日志用于回滚时，特别是索引页面分裂，可通过页面前后指针来完成回滚。**

# 第6章 多线程

2023.09.20

No matter where you go, no matter what you do, I know you will be ambitious. You wouldn't be here today if you weren't. Match that ambition with humility – a humility of purpose.

- introduction to thread
- thread in Java

## 1. 定义

- **进程**: 有独立的内存空间；两个进程的环境是完全隔离开来的，互相访问不到对方的内存空间（需要进程间通信）；
- **线程**: 轻量级的进程；进程和线程都提供了一个执行环境，但创建新线程所需的资源比创建新进程少。线程存在于一个进程中-每个进程至少有一个线程。线程共享进程的资源，包括内存和打开的文件。这有助于通讯高效但可能存在潜在的问题。

## 2. ICS限时返场：基础概念一览

### 2.1 interrupt

中断是对线程的一种指示，告诉线程应该停止正在做的事情，做其他事情。线程如何响应中断取决于程序员，但是线程终止是很常见的。

### 2.2 join

join方法允许一个线程等待另一个线程的完成。

- 如果t是一个正在执行的线程，t.join()会导致当前线程暂停执行，直到t的线程终止。
- join的重载允许程序员指定一个等待时间。然而，和sleep一样，join依赖于操作系统的时间，所以你不应该假设join会等待你指定的时间。和sleep一样，join响应中断会抛出InterruptedException。

我们来看一个例子：

```
package CSDN;
public class TestJoin {

    public static void main(String[] args) throws InterruptedException {
        ThreadTest t1=new ThreadTest("A");
        ThreadTest t2=new ThreadTest("B");
        t1.start();
        t2.start();
    }

}

class ThreadTest extends Thread {
    private String name;
    public ThreadTest(String name){
        this.name=name;
    }
    public void run(){
        for(int i=1;i<=5;i++){
            System.out.println(name+"-"+i);
        }
    }
}
```

输出结果：

```
A-1
B-1
B-2
B-3
A-2
B-4
A-3
B-5
A-4
A-5
```

可以看出，两个线程是交替执行的。如果我们想要让线程A执行完之后再执行线程B，我们可以使用join方法：

```
package CSDN;
public class TestJoin {

    public static void main(String[] args) throws InterruptedException {
        ThreadTest t1=new ThreadTest("A");
        ThreadTest t2=new ThreadTest("B");
        t1.start();
        t1.join();
        t2.start();
    }
}
```

输出结果：

```
A-1
A-2
A-3
A-4
A-5
B-1
B-2
B-3
B-4
B-5
```

### 3. 创建线程

- **方法一：手动创建一个线程**，这个线程执行完就结束了。 （人为的管理、控制线程）
  - 一种方法是继承Thread类（Java是单根继承，和cpp不一样，所以这种方法不推荐）
  - 一种方法是实现Runnable这个接口（推荐使用这个，因为Java里面只能继承一个类，但是实现可以实现多个接口）
- **方法二：通过线程池**，如果有需要获取线程就在池子里面找一个，用完之后收回，还是放在在线程池里面。节约资源，因为创建线程的时候需要消耗资源。（不需要考虑线程的管理，线程是新创建的还是已有的使用者不需要知道，由线程池管理）

### 4. 多线程问题

sharing access to fields会造成两个问题：

## 4.1 线程干涉

(两个线程都在写) A线程写入之后, B线程再次写入, 然后导致A写的丢失了, 这就是线程干涉。

If the initial value of c is 0, their interleaved actions might follow this sequence:

- Thread A: Retrieve c.
- Thread B: Retrieve c.
- Thread A: Increment retrieved value; result is 1.
- Thread B: Decrement retrieved value; result is -1.
- Thread A: Store result in c; c is now 1.
- Thread B: Store result in c; c is now -1.

## 4.2 内存一致性错误

因为一个线程在写, 一个线程在读。当**不同线程对应该是相同数据的看到的不一致时**, 就会发生内存一致性错误。避免内存一致性错误的关键是理解**先发生后发生的关系**。这种关系只是保证一个特定语句的内存写入对另一个特定的语句可见。(总之就是**不能线程交替读和写**, 要么专门读, 要么专门写)

Suppose a simple int field is defined and initialized:

- int counter = 0;
- The counter field is shared between two threads, A and B.
- Suppose thread A increments counter:  
`counter++;`
- Then, shortly afterwards, thread B prints out counter:  
`System.out.println(counter);`

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1".

But if the two statements are executed in separate threads, the value printed out might well be "0", because **there's no guarantee that thread A's change to counter will be visible to thread B** — unless the programmer has established a happens-before relationship between these two statements.

再来看一个例子:

```

public class UnSafeMultipleThreads<Static, C> {
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n",
            threadName,
            message);
    }
}

Counter c = new Counter();

private class CounterLoop implements Runnable {
    public void run() {
        try {
            for (int i = 0; i < 100; i++) {
                // Pause for 1 seconds
                Thread.sleep(1000);
                // Print a message
                threadMessage(String.valueOf(c.value()));
                c.increment();
            }
        } catch (InterruptedException e) {
            threadMessage("I wasn't done!");
        }
    }
}

public static void main(String args[]) throws InterruptedException {
    // Delay, in milliseconds before
    // we interrupt MessageLoop
    // thread (default one hour).
    long patience = 1000; // * 60 * 60;

    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();

    UnSafeMultipleThreads s = new UnSafeMultipleThreads();
    Thread t1 = new Thread(s.new CounterLoop());
    t1.start();
    Thread t2 = new Thread(s.new CounterLoop());
    t2.start();
    threadMessage("Waiting for MessageLoop thread to finish");
}

```

```
}
```

输出可能是：

```
Run: [UnSafeMultipleThreads] ×
▶ ↑ main: Finally!
[...] ↓ Thread-0: 0
[...] ⏺ Thread-1: 0
[...] ⏺ Thread-1: 2
[...] ⏺ Thread-0: 2
[...] ⏺ Thread-1: 4
[...] ⏺ Thread-0: 4
[...] ⏺ Thread-1: 6
[...] ⏺ Thread-0: 6
[...] ⏺ Thread-0: 8
[...] ⏺ Thread-1: 8
[...] ⏺ Thread-0: 10
[...] ⏺ Thread-1: 10
[...] ⏺ Thread-1: 12
[...] ⏺ Thread-0: 12
[...] ⏺ Thread-1: 14
[...] ⏺ Thread-0: 14
[...] ⏺ Thread-1: 16
```

## 5. 解决办法

### 5.1 Synchronized 同步修饰函数名

假设有一个线程在调用第一个方法，其他所有的线程都不能调用同一个对象的同步方法！注意是**三个函数只要有一个在被调用，剩下的包括自己就都不允许其他的线程调用**；注意是同一个对象，两个对象就没事。

- 原理：任何一个对象有一个“内部锁”，而每一个 `synchronized` 方法执行的前提是要获取这个锁；
- 问题：`static` 静态方法能不能使用同步方法呢？不会，因为每一个静态方法和类关联而不是一个对象，其实每一个class都对应关联着一个class Object，所以这个class Object也有锁，所以即使是静态方法也适用。

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment() { c++; }  
    public synchronized void decrement() { c--; }  
    public synchronized int value() { return c; }  
}
```

## 5.2 同步语句——细粒度的控制

5.1中，我们把同步放在了方法上面；我们是否可以把同步放在语句上面呢？答案是可以的。

```
// 注意看这个方法里面，有三个语句，我们只同步了其中的两个语句
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

再来看一个例子：

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;

    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        // 这里获得了lock1的锁（lock1是一个对象！）
        synchronized(lock1)
        { c1++; }

    }

    public void inc2() {
        // 这里获得了lock2的锁
        synchronized(lock2)
        { c2++; }

    }
}
```

## 5.3 可重入的锁： ReentrantLock

我们有一个类 `c`，它有一个Synchronized方法 `m`，`m` 里面调用了另外一个Synchronized方法 `n`，`n` 也在 `c` 里面；这个时候我们发现按照我们前面的逻辑就蚌埠住了。调用 `n` 的时候，发现这个对象的锁在 `m` 手里呢！

也有可能 `m` 是一个递归的方法……

引入概念：**可重入的锁**。可重入的锁意味着：

- 一个线程不能获得一个被其他线程占据的lock；但是它能够获得一个它已经拥有的锁。

## 5.4 原子变量

对于Java来说，除了 `long` 和 `double` 之外，其他的基本类型都是原子的。如果我们想要把一个变量变成原子变量，只需要在变量定义的前面加一个 `volatile` 修饰就可以了。

前提是这个变量很容易出现内存一致性错误的问题，就需要加一个原子变量，这样编译器就会识别，保证这个变量的线程安全。但是不要把所有的变量都变成原子变量，这样会降低性能，矫枉过正。

## 5.5 Liveness：多线程碰到的各种问题

- 死锁**：A线程等待B，B等待A，这样就会陷入死锁的循环。
- 饿死**：A想要访问一个共享资源，但是一直获取不到，就像被饿死了一样。（需要消除那些贪婪的线程，避免一个线程长期贪婪的占用资源；所以tomcat里面用了连接池，避免一个连接长期占用资源）
- 活锁**：假设有AB两个线程，A产生数据发给B，如果A产生的数据特别多，导致B无法处理，传来的数据太多了导致B忙不来，也就是A得不到响应或者回复答复。

## 6. Immutable Objects 不可变对象

为什么需要不可变对象：

- 线程干涉是因为（两个线程都在写W/W）
- 内存不一致是因为一个在写一个在读（W/R）
- 如果我们禁止掉写操作，就避免了这个问题。**
- 缺点：创建新对象的代价很大，但是其实Java的JVM让这个代价不是很大。

定义：如果对象在构造后其状态不能改变，则该对象被认为是不可变的。最大程度地依赖不可变对象被广泛接受为创建简单、可靠代码的可靠策略。不可变对象在并发应用程序中特别有用。因为它们不能改变状态，所以它们不能被线程干扰破坏或在不一致的状态下观察到。

- 想要修改不可变对象？**那就创建一个新对象！**

为什么要选择线程安全的集合类型来维护客户端 Session？

如果有大量的用户（或者说客户端进入到websocket连接），那么这个时候要做的动作就是把Session放入一个集合中【我用的集合是ConcurrentHashMap】，也就是在往集合里面写入内容，如果不能保证线程安全，由于在往集合中写入内容这一个操作不是原子操作，甚至涉及到很多复杂的过程，也就是说这些线程直接会相互干扰影响（我们可以举一个最简单的例子，哪怕i++这一行代码从汇编来看都涉及到三个操作，从内存拷贝、自加、然后复制回去内存），最终导致的结果就是：可能同时有100个用户涌进来，但是由于线程之间的干扰，最终能够维持的Session就只有80或者90多个，会有遗漏。为此，我们需要使用线程安全的集合来维护Sessions集合。

## 7. 线程池

- 执行器 Executor：你也不要自己创建线程；我创建一个执行器，它就是一个线程池；线程池来管理对象的生命周期，客户端只需要把任务交给线程池就可以了。
- 使用工作线程可以最大限度地减少线程创建带来的开销。一种常见的线程池类型是**固定数量线程池**（没有创建线程的开销）。这种类型的池始终有指定数量的线程在运行。固定线程池的一个重要优点是，使用它的应用程序可以优雅地降级（我写的代码崩溃了但是不会影响线程池，独立的）。
- 创建使用固定线程池的执行器的简单方法是调用 `java.util.concurrent.Executors` 中的 `newFixedThreadPool` 工厂方法；此类还提供以下工厂方法：`newCachedThreadPool` 方法创建具有可扩展线程池的执行器。此执行器适用于启动许多短期任务的应用程序。`newSingleThreadExecutor` 方法创建一个执行器，一次执行一个任务。

补充：java新特性

- `virtual threads`：更加轻量化，比如在被IO阻塞时，那么就把线程解绑；等IO结束时再绑定线程，但不一定是原先的线程。但对于长时间占用cpu的任务不友好。

# 第7章 缓存

2023.10.02

目标：根据业务需求，设计实现基于分布式缓存的数据访问方案。

完全明白是放纵  
但是只得这刻可相信  
未来又怕会  
终于都扑空  
——《拥抱这分钟》 陈奕迅

## 1. 为什么缓存

Recall: Servlet and Tomcat

### 1. Servlet

- Servlet是Java编写的**服务器端程序**，用于**处理Web请求和生成Web响应**。它是基于Java的标准，可以通过Java的API来创建和扩展。Servlet通常用于处理HTTP请求，但也可以用于处理其他协议。
- Servlet的主要功能是接收来自客户端（通常是浏览器）的HTTP请求，执行相应的处理逻辑，然后生成HTTP响应返回给客户端。这可以包括动态生成HTML页面、处理表单提交、管理会话等任务。

### 2. Tomcat

- Tomcat是一个开源的Java Servlet容器，也可以称为**Web服务器**。它是Apache软件基金会的项目，用于**运行和管理Servlet和JavaServer Pages (JSP) 应用程序**。
- Tomcat提供了一个运行环境，可以部署和托管Servlet和JSP应用程序。它处理了与HTTP通信相关的底层细节，包括接收HTTP请求、解析请求、调用适当的Servlet来处理请求，并将响应返回给客户端。

总结一下，Servlet是一种用于**处理Web请求和生成Web响应的Java组件**，而Tomcat是一个容器，用于**托管和运行Servlet和JSP应用程序**。在学习Spring时，你将经常与这两个概念打交道，因为Spring的Web模块可以用来开发基于Servlet的Web应用程序，而Tomcat可以用于部署和运行这些应用程序。希望这个简要的介绍有助于你更好地理解它们的作用和关系。

## 1.1 网络开销

前端 → Tomcat → 数据库

Tomcat和数据库都不一定是在一台机器上面，可能是分布式的，可能是多台机器，可能是多个Tomcat，可能是多个数据库。

如果绝大多数情况都是只读的，后端和数据库之间的交互，**网络开销也很大啊！**

——上缓存！直接把数据缓存到Tomcat那里！

## 1.2 似乎已经有缓存了？……

还有一个问题是：

Q：数据库里面有buffer、ORM映射里面也有buffer作为缓冲区，那为什么要缓存呢？

A：上面说的这两个缓存都不是你可以控制的，完全取决于他们自身，我们开发者管不了。而使用redis，我就可以自己的控制缓存！

事实上，我们后台的数据库未必是关系型数据库！我们可能还搞了文件系统、mongo……最终它们还是表示为对象，理论上来说它们也应该进缓存！但是像**spring jpa里面的缓存，或者hanibatis里面的缓存**，把数据库里面抓取过来的数据已经弄成对象缓存了，它也**仅仅针对关系型数据库！**（还有，我把东西写到redis里面，写入的是已组装好的对象；原始数据组装为对象也需要消耗呢！所以这也是redis节省的地方）

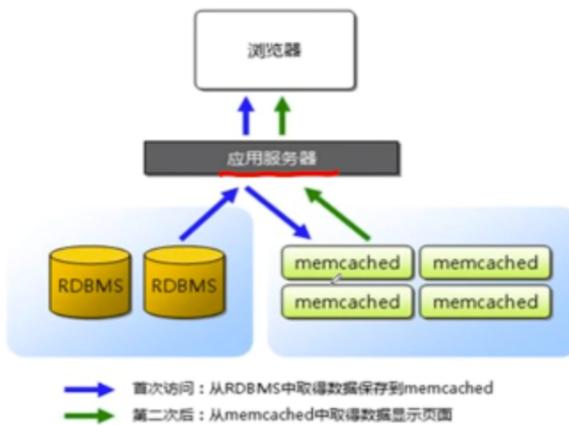
假如我还有从文件系统里面读取的，nosql数据库里面的读取的东西、动态生成的网页、图片都可以，所以使用redis就都可以存储，redis完全在开发者的控制下。比如双十一的时候，我可以趁用户访问量少的时候提前把可能卖爆的网页缓存出来，这样提高访问的速度。

redis甚至还可以作为消息中间件，类似kafka的topic，他和卡夫卡的差别是redis里面的东西在内存里面。

**数据库服务器对于内存要求比较高，而Tomcat服务器主要是CPU密集型**（因为要处理一堆请求）；所以，对于一些只读的数据，我从数据库拿到之后就缓存在Tomcat本地！

## 1.3 Memcache

memcached：存储在内存中的KV

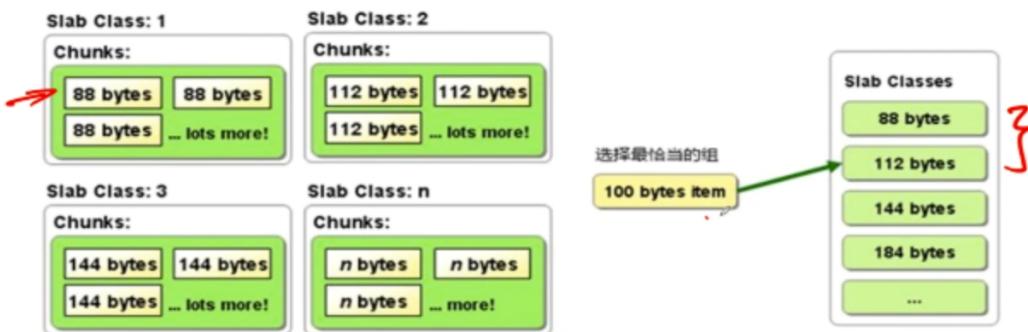


memcached该如何设计，才能让内存高效的利用呢？



varchar就两端放，因为不知道header和数据有多长

memcache定义了标准容器（不同大小的chunk）



## 2. 一致性哈希

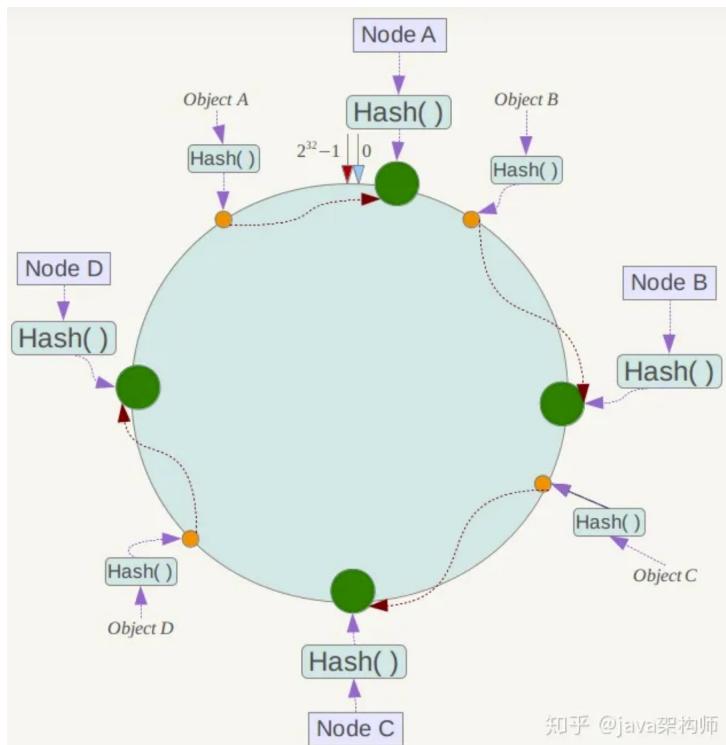
参考资料：<https://zhuanlan.zhihu.com/p/98030096>

## 2.1 好的哈希算法？

一致性哈希提出了在动态变化的Cache环境中，哈希算法应该满足的4个适应条件：

1. 均衡性 (Balance)：哈希的结果应该尽可能分布均匀，这样可以使得每个缓存节点负载均衡。  
(如果一个用完了另一个还没用，就很难蚌)
2. 单调性 (Monotonicity)：当缓存节点增加或者减少时，应该尽量减少缓存数据的重新分配。  
(尽量保护已经分配的内容不会被重新映射到新的缓冲区)
3. 分散性 (Spread)：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。
4. 负载 (Load)：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

## 2.2 一致性哈希算法



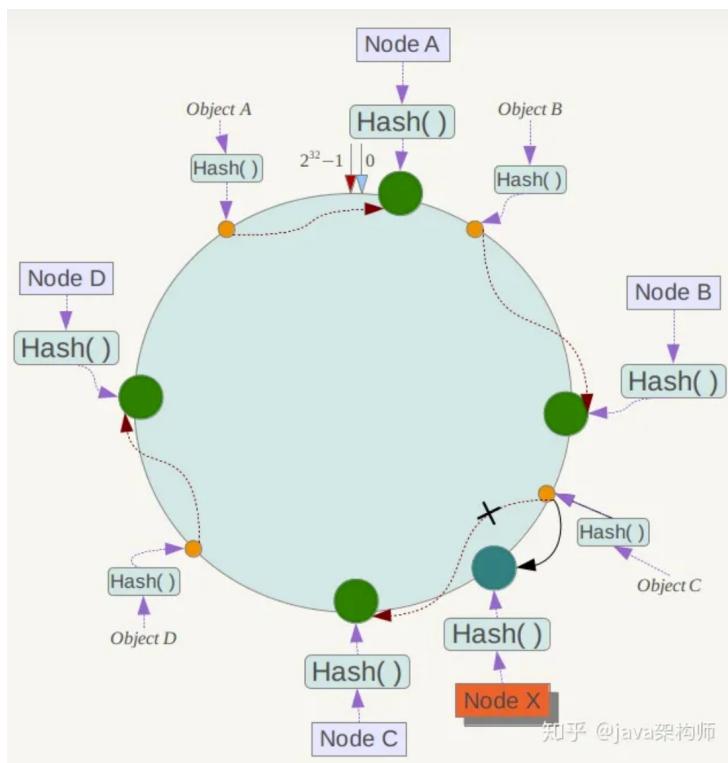
按照常用的hash算法，来将对应的key哈希到一个具有 $2^32$ 个节点的空间中；现在我们将这些数字头尾相连，结合成一个闭合环形。

然后我们映射服务器节点。具体可以选择服务器的ip或唯一主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置。假设我们将四台服务器使用ip地址哈希后在环空间上。

然后我们将objectA、objectB、objectC、objectD四个对象通过特定的Hash函数计算出对应的key值，然后散列到Hash环上，然后从数据所在位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器。

## 服务器的删除与添加

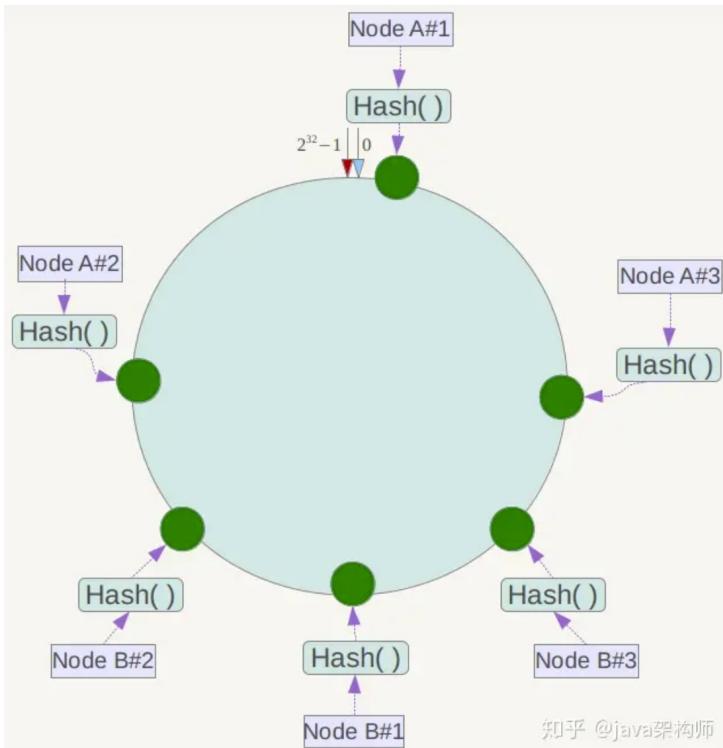
1. 如果此时NodeC宕机了，此时Object A、B、D不会受到影响，只有Object C会重新分配到Node D上面去，而其他数据对象不会发生变化；
2. 如果在环境中新增一台服务器Node X，通过hash算法将Node X映射到环中，通过对节点的添加和删除的分析，一致性哈希算法在保持了单调性的同时，还是数据的迁移达到了最小，这样的算法对分布式集群来说是非常合适的，避免了大量数据迁移，减小了服务器的压力。



## 2.3 虚拟节点

当服务器节点比较少的时候，会出现一个问题，就是此时必然造成大量数据集中到一个节点上面，极少数数据集中到另外的节点上面。

为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以先确定每个物理节点关联的虚拟节点数量，然后在ip或者主机名后面增加编号。例如上面的情况，可以为每台服务器计算三个虚拟节点，于是可以分别计算“Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：



同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到Node A上。这样就解决了服务节点少时数据倾斜的问题。每个物理节点关联的虚拟节点数量就根据具体的生产环境情况再确定。

## 3. Redis!

### 3.1 基本概念

- key-value store
- **NoSQL** database

可以有主从备份（缓存怎么会有主从备份？因为有人把它当NoSQL数据库用……）、集群化部署

什么东西放缓存：按照上课讲的，那就是：

1. 经常用到的东西
2. 经常涉及到读取操作的，而很少涉及到写的操作的
3. 中间计算的结果
4. 经常要写的东西不要放入缓存

### 3.2 spring!

#### 概念：什么是序列化？

两个进程之间通信（夏老师讲的cse!），实际上是一件很麻烦的事情！

Tomcat在一台服务器跑我们的应用，redis也在另一台跑着，它们是两个进程。

Tomcat写了无论哪个地址，我想访问你redis那边的地址，我访问不到啊！进程间所谓的通信就是传输数据；比如我要传输一个object，我把它转化为一个字节数组 byte[]，这就是序列化；然后你收到了这个字节数组，你就把它反序列化，就变成了一个object。

写一个 RedisConfig.java 的配置类，然后写个 yaml 文件，用！

## 4. 分布式缓存

- 为什么需要分布式缓存：因为一个机器的缓存可能不够，**物理上多个机器，实际逻辑上是一个大的缓存**，可以充分利用多个机器的内存；
- 稳定可以备份，可靠性增加。

# 第8章 全文搜索

2023.09.23

## 1. 概览

- **Lucene** (重点；读音：鲁-sing) : Java开发的全文搜索引擎工具包，提供了完整的查询引擎和索引引擎，部分功能如分词、排序、过滤等。

后面两部分简单过一下：

- Solr: 基于Lucene的全文搜索服务器，提供了完善的web界面，方便用户使用。
- Elasticsearch: 基于Lucene的全文搜索服务器，提供了完善的web界面，方便用户使用。

## 2. 什么是全文搜索

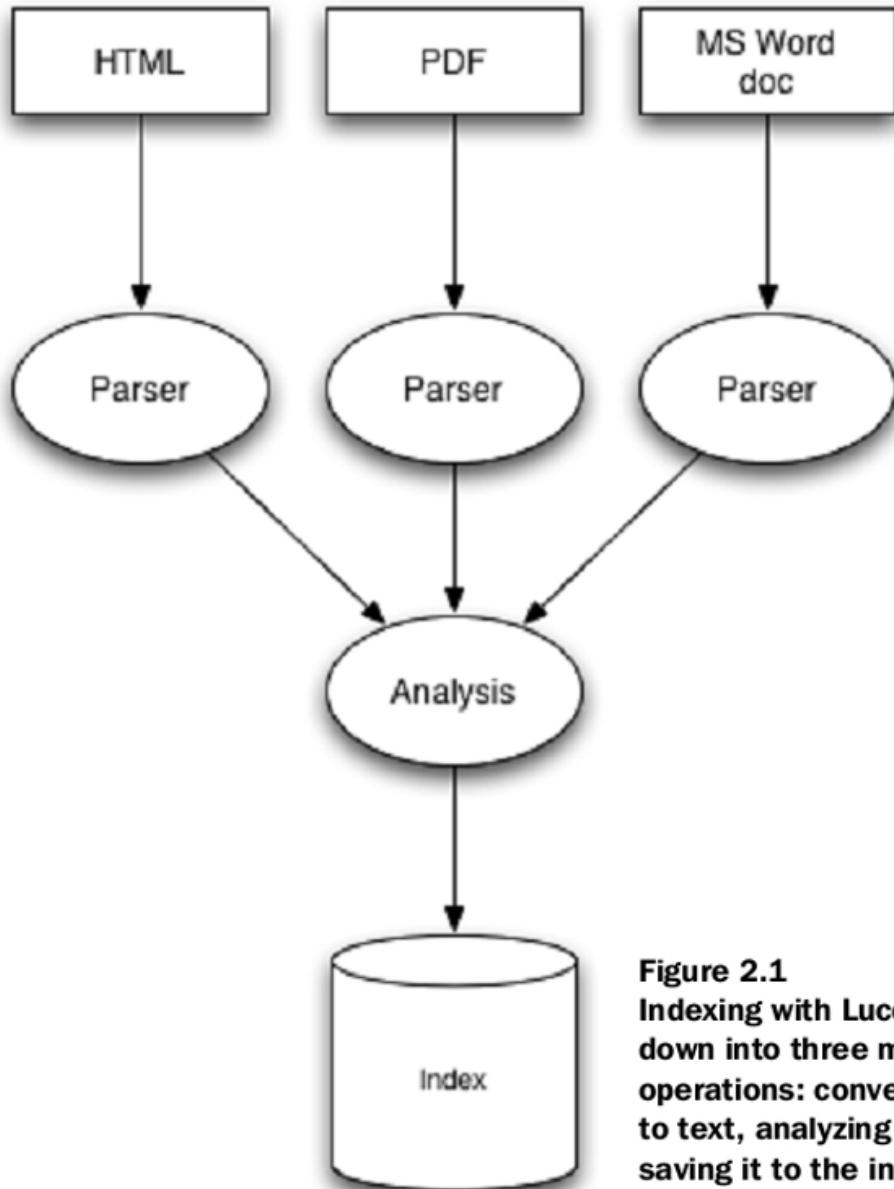
Lucene做的事情：

1. 建立索引 (如果不建立索引，每次都要扫描所有的文件，效率太低)
2. 搜索

类似百度的网站，用户搜索一个关键词，根据建立的索引，查找包括关键词的相关的**非结构化的数据**里面有哪些包含目标关键词。

所谓的结构化，比如数据库，所有数据是schema的，数据有字段，像表格，是有结构的；但这里比如我给你一个html，或者一段txt，这种东西是没有结构可寻的。

- 结构化数据建立索引的方式：B+树就可以
- 那非结构化数据怎么办呢？比如有一个dir目录，里面有一大堆.txt，和.html文件，我要问这个目录里面包含关键词java的文件有哪些？



**Figure 2.1**  
**Indexing with Lucene breaks down into three main operations: converting data to text, analyzing it, and saving it to the index.**

### 3. 建立索引

- 如果我搜索java，你要告诉我java出现在哪些行，以及具体的位置？
1. 用某一种tokenizer处理源文件
  2. preprocess，比如把所有的大写字母都变成小写，相同的单词都合并到一起（比如有些单词是take、took、taken都是take的不同形式）
  3. 反向索引：为什么叫反向索引呢，因为不是每一行映射到几个单词，而是某一个关键词映射到哪几行的哪几个位置。

**正向索引 (Forward Index) 和反向索引 (Inverted Index)** 是两种不同的文本索引技术，用于管理和加速文本数据的检索。它们在搜索引擎和信息检索系统中起着关键作用，并在不同的方面具有

不同的优势和用途。

## 1. 正向索引 (Forward Index) :

- 正向索引是一种按文档来组织和存储文本数据的索引方式。每个文档都有一个对应的索引项，这个索引项包含了文档中的所有信息，通常以文档的标识符（如文档ID）为索引的键。
- 正向索引适合于需要按文档进行检索的场景，例如在文档管理系统中查找特定文档或根据文档属性进行过滤和排序。
- 缺点是在处理大量文本数据时，正向索引可能需要大量的存储空间，因为每个文档都需要一个完整的索引项。

## 2. 反向索引 (Inverted Index) :

- 反向索引是一种按单词或词组来组织和存储文本数据的索引方式。它将文本数据中的每个单词或词组与包含它们的文档关联起来，以及它们在文档中的位置。
- 反向索引适用于全文搜索和信息检索系统，因为它允许根据关键词快速查找包含这些关键词的文档。这种索引方式在搜索引擎中得到广泛应用。
- 反向索引通常占用相对较少的存储空间，因为它不需要存储完整的文档内容，只需存储单词或词组的位置信息和文档标识符。

比较：

- 正向索引适用于需要按文档检索的应用，例如文档管理系统或内容展示。
- 反向索引适用于需要全文搜索和关键词检索的应用，例如搜索引擎和信息检索系统。
- 正向索引需要更多的存储空间，但在访问特定文档时速度较快。
- 反向索引占用较少的存储空间，但在全文搜索和关键词检索时速度更快。

通常，搜索引擎会结合使用正向索引和反向索引，以满足不同的检索需求，并提供高效的搜索体验。正向索引用于快速获取文档的详细信息，而反向索引用于高效地找到包含查询关键词的文档。

# 4. Core Indexing Classes 核心索引类

## 4.1 Brief Overview 简要概述

### • IndexWriter:

- `IndexWriter` 是索引创建和维护的核心类。它负责将文档添加到索引、更新索引、删除文档以及优化索引等操作。`IndexWriter` 是在索引建立和更新过程中的主要接口之一。

### • Directory:

- `Directory` 是索引文件的存储和管理抽象。它定义了索引文件的位置和访问方式，可以是基于文件系统的目录，也可以是内存中的数据结构。`Directory` 提供了对索引文件的读写操作，使得索引可以被持久化存储和检索。

- **Analyzer:**
  - Analyzer 是文本分析的关键组件。它定义了如何将文本数据分割成单词或词组，进行词干化、去除停用词等文本处理操作。正确选择和配置适当的分析器对于索引的质量和性能至关重要。
- **Document:**
  - Document 表示索引中的一个文档。文档通常由一组字段（Field）组成，每个字段包含了文档的一部分信息，如标题、正文、作者等。Document 用于将文本数据添加到索引。
- **Field:**
  - Field 是文档中的一个字段或属性。它包含了字段的名称、值以及用于指定如何处理该字段的配置选项。字段可以是文本、数字、日期等不同类型的数据，根据需要进行索引和检索。

这些核心索引类是构建文本搜索引擎和信息检索系统的基础，它们协同工作以创建、管理和查询索引，以便用户能够高效地检索和获取相关文档。

## 4.2 Field

每个字段对应于在搜索期间根据索引查询或从索引检索的一段数据。

```
public abstract class BaseIndexTestCase extends TestCase {
    protected String[] keywords = {"1", "2"};
    protected String[] unindexed = {"Netherlands", "Italy"};
    protected String[] unstored = {
        "Amsterdam has lots of bridges",
        "Venice has lots of canals"
    };
    protected String[] text = {"Amsterdam", "Venice"};
    protected Directory dir;

    ...
}
```

- **Keyword**：关键字不被分析（就是不会被拆开），而是被索引并逐字存储在索引中。
- **UnIndexed**：既不分析也不索引，但其值按原样存储在索引中。也就是说一旦假如通过keyword查询到了，要把unindex的部分的内容要带出来，但是这部分不索引。（你不需要把荷兰和意大利做索引，我希望的是我查出来1和2，你把荷兰和意大利的内容带出来）
- **UnStored**：此字段类型被分析和索引，**要放到索引里去建，但不存储在索引中**。当然有一些停用词列表，比如一些非常常用的词，这些东西就被忽略。一般是正文，非常长不应该存在索引里面，非常浪费。（比如这里我们把Amsterdam,bridge,Venice,canal来放入索引，你按照bridge查是可以查到这个文件的，但是你索引里面没有存储整个完整的内容，因为完整的内容可能很大，所以我们真的要取要读硬盘）

- `Text` : 被分析并被索引，存储到索引里面。这意味着可以针对这种类型的字段进行搜索，但要注意字段大小。可以类比`keyword`，区别是**这个text会被分析一下，而keyword不会**（也就是text会被断开）

综上，以上field保证索引的大小！

再举一个例子：

写一个 `javaDoc` 文档，比如一个类一个 `javaDoc`，应该怎么做呢？

- `keywords` : 类名，这个就没必要断开了
- `unindexed` : 带出来的内容，比如类的作者、编写日期、版本（单搜3.0版本没意义，很多类都是3.0版本的）
- `unstored` : `code`和`comment`，这个东西太大了，不应该存储在索引里面
- `text` : 类的描述，类的功能描述，一段话，那我就希望断开来扔进索引。比如写的是“这个类是计算哈希值的”，那么我希望搜索“哈希”和“计算”的时候都能搜得到。

真正查找的时候，都是`vector`: 向量化，据此可以找到相似的单词。

打分逻辑：一般按相似度打，当然也可以自己boost（比如给广告费了就涨一点）。

# 第9章 web服务

2023.09.24

## 一、WebService概念

Web Services:

- present the opportunity for real interoperability across hardware, operating systems, programming languages, and applications 提供跨硬件、操作系统、编程语言和应用程序实现真正互操作性的机会

Web:

- **access with web protocols**, such as HTTP, FTP, SMTP
- 相对于IPC（进程间通信），比如java中的RMI（remote method invocation）远程方法调用，或者其他语言中的RPC（远程过程调用），**web的访问能力会强得多**——只有web才能做到说比如美国大学有个服务器，我在中国也可以访问得到。

Services:

- independent of implementation **独立于具体的实现**
- Service解决的是异构的问题（操作系统、编程语言的差异），比如客户端是C#开发的，服务端是java开发的，那为了方便他们之间的交互，我们就要走**纯文本**的路线，大家都能认识。既然是纯文本，那需要一定的格式，SOAP传的数据里面包括了一系列的operation、参数的名称类型，但是数据驱动型的。

## 二、SOAP：简单对象访问协议

SOAP：指的是简单对象访问协议，针对API纯文本化传递，例如**规定传递XML的格式**。

WSDL：一个自描述的文件，把接口描述成xml的形式（只能XML），这个文件不需要自己写，通过工具生成。会根据接口、参数类型、参数顺序、接口返回的参数类型，全部用xml来表述出来。此外会把operation、服务所在的URL接口的名字写出来。

binding是什么呢？比如一个webService可以通过http和ftp，那就有两个 binding；也就是 binding 是说明怎么访问的。类比老师接受答疑，可以通过腾讯会议也可以通过电话。

举例：

一个Java程序访问一个c#扣款的服务：

那么这个java程序就需要把c#扣款的服务的wsdl文件给拿到本地，无论你什么框架（比如spring），你要做两件事：

1. 根据wsdl文件生成java语言的接口
2. java程序调用接口，里面有一些操作可以调用，调用之后就可以经过一个proxy代理，翻译成一个soap消息发送到c#扣款的服务

c#扣款的服务接收到soap消息，也是先被一个proxy代理拦截，把soap消息翻译成c#然后调用具体的业务逻辑。产生结果后，又返回给proxy代理，组装成一个soap消息，作为调用结果返回java端的proxy接收到soap消息，翻译成java的返回结果。

java端成功的获取到返回结果

## 三、SOAP的缺点

**性能不好！**

- 需要一个WSDL的文件，里面需要描述消息传递的格式，有一个外挂文件的方式总的来说是不好，需要额外的东西（自包含是最好的）
- 反复组装和解析SOAP需要消耗大量的时间，需要经过很多的翻译
- 客户端、服务端和API耦合，一旦API发生变化（比如下单参数从四个变成五个），那么WSDL文件就要发生变化，服务端客户端代码都需要重新生成，需要消耗时间资源（代码可维护性变差）

因此我们需要**数据驱动型**的方式来，**Restful Web Service**应运而生。

## 四、Restful Web Service

含义：Representational表达性，State状态，Transfer转移，全名**表述性状态转移**。

ajax + json事实上就是web2.0所谓强调动态内容生成去加载；所以现在我们来看restful web service就比较好理解了：

- 表达性：
  - 所有的数据都是资源（既然是资源，就可以通过统一资源定位符来描述）；至于数据怎么描述，这件事情是客户端做的（不是客户端自己负责，客户端的js脚本当然是从服务器下载下来的，是你开发的）
  - 每一个资源都可以有不同的表现方式（同样的数据，可以呈现为饼图、柱状图、折线图）
  - 每个资源都有自己的独有的URI

URL: Uniform Resource Locator 统一资源定位符

URI: Uniform Resource Identifier 统一资源标识符

URL是URI的特例：URL一定要有页面，但是URI不一定要有页面，比如我要查询一个学生的信息，那么我可以通过 /student/1 来查询，这个URI就是一个资源的表述，但是这个URI不一定要有页面。

- 状态：

- 指的是客户端的状态，客户端维护自己的状态，服务器是无状态的（这就回到了第一节课说的，如果服务器是有状态的，不同的客户端，我服务器就要维护不同的对象！），客户端自己保存cookie或者localStorage
- 资源的表示就是客户端的状态（数据在前端表示为表格或者图？）

- 转移：

- 客户端从一个页面跳到另一个页面，一种表示跳到另一种表示，**客户端的状态/表示可以发生迁移**。
- 迁移的过程中，客户端只把数据请求回来，然后呈现出来，客户端的页面然后就发生了变化，这就是所谓的表达性状态转移

### **Restful Web Service特点：**

1. 典型的客户端、服务端架构，但是是无状态的
2. 客户端和服务端之间传递的都是**数据**（数据！纯的！）
3. 服务端只处理数据，数据的展示完全依赖于客户端

### **How to design REST**

1. 通过URL设计

- developers just need to abstract resources according to URLs
- we just need to design suitable URLs which directly represent user interface

2. Http based:

- GET方法对应读、POST代表创建、PUT代表更新、DELETE代表删除，四个HTTP方法严格对应增删改查
- 通过Http的返回码表示返回的结果，比如201成功，404代表不存在，500代表服务器内部的错误。

## **五、Web Service的优缺点**

【这一板块如果有任何问题，请直接回去看web service的定义】

优点：

- 跨平台，基于XML, json等等
- 自描述：
  - i. WSDL就是一个自描述文件，里面包括了一系列的操作
  - ii. Restful的话就是完全基于URL
- 模块化好，封装的好，不需要关心具体的实现
- 区域访问性质广，可以穿透防火墙

缺点：

- 写代码的效率低，不适合stand-alone应用，soap显然比之间rpc效率低，rest因为传递的是纯数据，数据的解析需要消耗一定的资源
- 性能有一定的降低了：需要把java对象转化为一个纯文本的，或者解析文本作为一个java对象
- 安全性，因为webService的地域性广，就比较容易受到攻击，或者中间人截获了。

## 六、什么时候用web Service

支持跨防火墙的通讯、支持跨防火墙的通信、支持应用集成、支持B2B集成、鼓励重用软件；

不应该使用WS时：独立应用程序、如MS Office（访问范围小）、局域网中的同构应用（都是java接口，直接上java就完事了）：例如COM+或EJB之间的通信（再比如spring访问mysql、redis、kafka、elasticSearch）

## 七、SOA：面向服务的架构

SOA指的是Service-oriented architecture（面向服务的架构）。我们关注的问题是：**我们在系统开发的过程中，异构性和需求的变化是我们永远都会遇到的，那么我们如何让自己的系统去快速响应这些问题呢？就是网上去找别人写好的东西，如果别人写的是和语言无关的服务，我们就直接拿来集成！**

假如我要构建一个电子书店、里面包括了认证登录系统、财务系统、统计系统、订单系统，这么多系统开发下来很复杂，而且假如我是卖家，卖给了1000个新华书店的用户，然后突然发现某个系统存在漏洞，就要告诉一千个用户，这很麻烦。

所以就有一个新方法、登录系统用第三方的比如百度的，订单系统用第三方的，然后我自己就只做一个集成。如果发现登录系统有bug，就告诉百度的问题，百度只要更新自己的就可以了，我们不用把更新包分发给1000个用户，因为它们都是直接调用百度的服务（**比如你调百度地图，你当然不是把百度的副本放到本地运行，而是调用人家的api**）。这样就可以快速构建一个应用，节约开发成本。

特点：

1. 服务之间松散耦合，也就是是登录系统、财务系统、统计系统、订单系统这些系统通过我开发的中间件联系，而不是producer&consumer之间直接互相通信，这样的好处就是假如我发现登录系统不好或者太贵了，我可以换一下，换之后别的系统也不会受到影响（当然松耦合的代价就是性能会差一点）
2. 位置透明，中介者负责路由
3. 协议独立，从http切换到ftp，但是不要让客户端来实现

# 第10章 微服务

2023.10.02

难离难舍想抱紧些  
茫茫人生好像荒野  
如孩儿能伏于爸爸的肩膊  
哪怕遥遥长路多斜  
——《单车》陈奕迅

## 1. 什么是微服务

- 我们构建的不是一个完整的系统，而是多个小的、自包含（比如有自己的数据库、缓存……）、互不干扰的（比如一个崩溃了不干扰另外一个系统）小模块构成。每个小模块都可以单独的独立的运行。这样就使得我们的代码有弹性。

总之微服务就是把一个大型的应用切分成几个小的应用模块，每个块都独立与其他的服务。

为什么用微服务：

- 便于维护（单个服务器出问题了针对性解决，而不需要把所有的服务停机）、彼此隔离（微服务的每个小模块之间彼此隔离不影响）
- 提高生产力（彼此之间没有直接交互，开发的时候效率++）
- 更高的容错性（一个坏了别的不会坏）
- 更好的业务提供（容错性好也保证服务的稳定性）

前端不需要硬编码端口（或者机器ip，因为也不一定是同一台机器呀）；我在微服务里面加一个Gateway网关来转发；这样我微服务集群里面哪怕一个机器崩了，把相关服务迁移到另一个机器，IP地址、端口什么的都改变了，前端代码不需要改变，因为GateWay会处理转发。

缺点：

1. 如果GateWay崩溃了那么就有些困难，什么事情都做不了了；
2. 成本增加，运维的复杂度增加，原先所有的都在一个机器上，但是现在可能变成了多个机器。

## 2. 各部分的作用解释

- **GateWay**: 作为一个统一访问的网关。应用程序（前端）不需要知道具体的是哪个服务器处理对应的服务，只需要对GateWay发送请求就可以。此外，GateWay还可以起到负载均衡的作用，比如登录服务有三个服务器，他们的key名字都是完全一样的，GateWay就会采用相关的负载均衡的策略，提高并发性能。
- **注册中心**: (Eureka为例) 相当于一个注册表 (key-value存储, key代表服务名字, value代表服务器地址, 这可能会发生改变)，当一个微服务的服务器启动的时候 (包括GateWay)，他会到Service Registry这个注册表上注册自己的服务，这样GateWay在面对客户端请求到来的时候，就会根据这个注册表来把请求转发给相关的微服务的服务器。

## 3. 无服务 (Serverless Application)

无服务器应用程序利用现代云计算功能和抽象，让人专注于逻辑而不是基础设施。在无服务器环境中，您可以专注于编写应用程序代码，而底层平台负责扩展、运行时、资源分配、安全性和其他“服务器”细节。

### 什么是无服务：

- 无服务器工作负载是“与服务器基础架构通常处理的方面无关的事件驱动工作负载”，**诸如“要运行多少实例”和“要使用什么操作系统”之类的问题都由函数即服务平台（或FaaS）管理**，让开发人员可以自由地专注于业务逻辑。

无服务器应用程序具有许多特定特性，包括：带有触发器的事件驱动代码执行；平台处理所有启动、停止和缩放事务；可扩展到零，闲置时成本低到零；无状态

### 函数式服务：

- 类比 $y = f(x)$ ，给定一个输入，得到一个输出结果。此外两次或者多次的函数计算之间没有关系，上次请求计算的结果计算完返回就结束，跟后面的没有任何关系，体现出一个无状态。

## 4. 实际部署

- **首先需要启动注册表服务器**，因为其他的服务器启动的时候都要到服务注册这里来注册一下，所以必须要最先启动Eureka的这个服务。
- **然后启动微服务具体的服务器**，在这个服务器启动的时候会自动到注册表服务器里面注册一下，便于后面的GateWay能够找得到。
- **最后启动GateWay**，因为具体的业务服务器已经启动了，启动GateWay之后，用户的请求就可以转发到对应的微服务，如果先启动GateWay而没有启动具体的业务服务器，如果有请求发来就会导致

GateWay找不到对应的微服务的服务器，然后出现报错。

## 5. 微服务状态

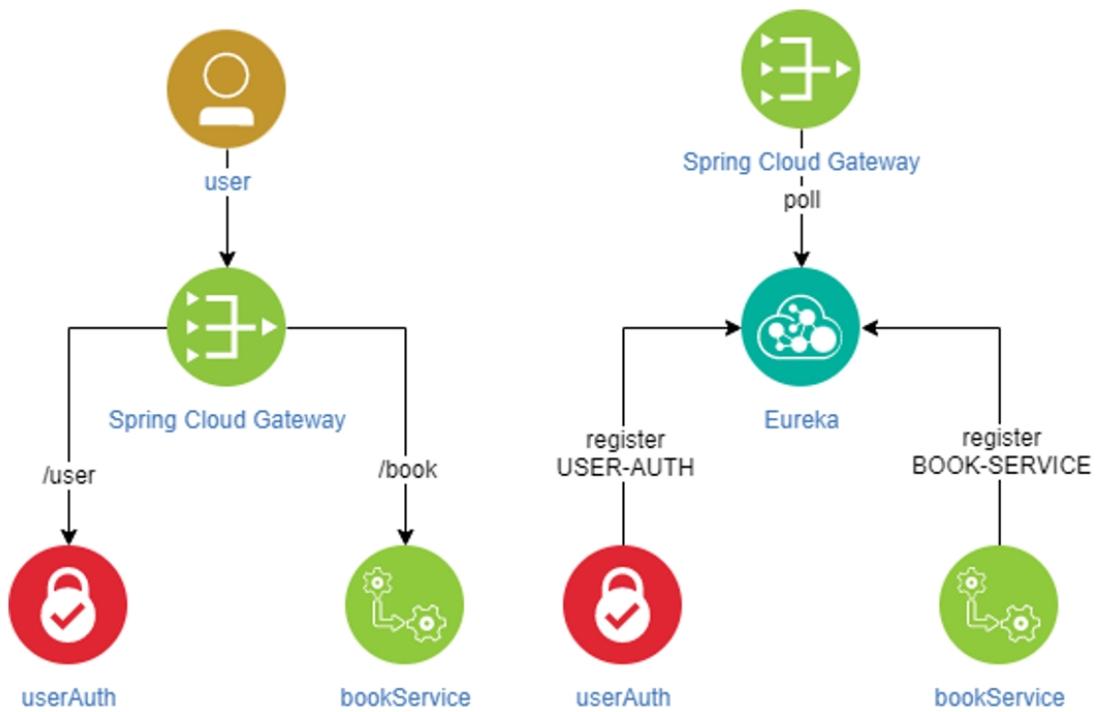
微服务怎么记录用户的登录状态？

**肯定不能用session**，比如微服务集群里面一个服务器用来登录，如果用这个服务器存储session，那么他下次下单的时候，可能是另外一个服务器处理的，而另外的服务器不知道用户的登录状态，因此解决方案就是jwt。

用户使用一个jwt token传入，服务器的GateWay对于需要登录授权的请求加一个过滤器，然后解析和这个jwt，再把解析获取到的用户信息放入请求的header里面，如果解析失败就直接拦截不转发给后面的，这样也增加了安全性。

## 6. 举一只栗子

这是一个书籍服务的微服务架构。



### 6.1 配置文件一览

下面是几个服务的配置文件。

Eureka 的配置文件：

```
server:  
  port:  
    8040  
spring:  
  application:  
    name: Eureka  
eureka:  
  instance:  
    prefer-ip-address: true  
  client:  
    fetch-registry: false  
    register-with-eureka: false  
    serviceUrl:  
      defaultZone: http://localhost:8040/eureka
```

Book 的配置文件：

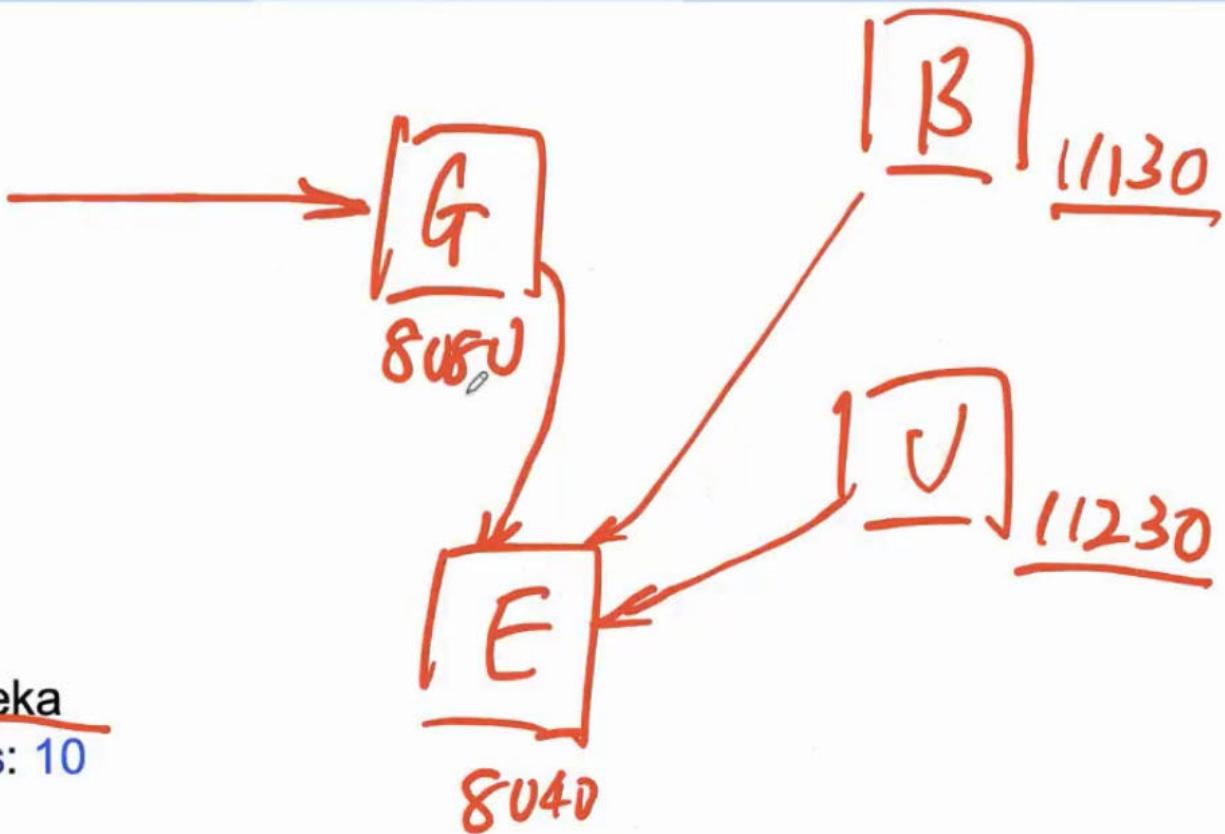
```
# 服务名字  
spring:  
  application:  
    name: book-service  
  
# 我要注册到Eureka上面去， Eureka的信息  
eureka:  
  instance:  
    prefer-ip-address: true  
    ip-address: localhost  
  client:  
    registerWithEureka: true  
    fetchRegistry: true  
    serviceUrl:  
      defaultZone: http://localhost:8040/eureka  
server:  
  port: 11130
```

auth 服务的配置文件：

```
#mybatis:  
# config-location: classpath:mybatis/mybatis-config.xml #全局配置文件位置  
# mapper-locations: classpath:mybatis/mapper/*.xml #sql映射文件位置  
spring:  
    datasource:  
        url: jdbc:mysql://localhost:3306/bookstore  
        username: root  
        password: reins2011!  
        driver-class-name: com.mysql.cj.jdbc.Driver  
    application:  
        name: user-auth  
    jpa:  
        hibernate:  
            ddl-auto: update  
  
eureka:  
    instance:  
        prefer-ip-address: true  
        ip-address: localhost  
    client:  
        registerWithEureka: true  
        fetchRegistry: true  
        serviceUrl:  
            defaultZone: http://localhost:8040/eureka  
server:  
    port: 11230
```

Gateway 的配置文件:

```
server:
  port: 8080
  error:
    include-message: always
spring:
  cloud:
    gateway:
      globalcors:
        cors-configurations:
          '/**':
            allowedOrigins: "*"
            allowedMethods:
              - GET
              - POST
application:
  name: gateway
eureka:
  instance:
    prefer-ip-address: true
    ip-address: localhost
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8040/eureka
      eureka-service-url-poll-interval-seconds: 10
```



Eureka  
ids: 10

Gateway , auth , book 都注册到了 Eureka 上面去了。

## 6.2 Eureka

Eureka 其实是很被动的，它啥也不知道，都是别人来找他注册。

代码就没啥需要自己写的：

```
package org.reins.se3353.eureka;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer
@SpringBootApplication
public class EurekaApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }

}
```

## 6.3 Gateway

Gateway 就核心得多了，下面仅仅展示 GatewayApplication 的代码：

```

package org.reins.se3353.gateway;

import org.reins.se3353.gateway.filters.JwtCheckFilter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableDiscoveryClient
public class GatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }

    @Autowired
    JwtCheckFilter jwtCheckFilter;
    @Bean
    public RouteLocator myRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
            .route(r -> r.path("/book/**"))
                .filters(f -> f.rewritePath("/book", "").filter(
                    .uri("lb://BOOK-SERVICE"))
            .route(r->r.path("/user/**"))
                .filters(f->f.rewritePath("/user", ""))
                    .uri("lb://USER-AUTH")
            )
            .build();
    }
}

```

Gateway 做的事情是：

客户端发过来的请求，比如 8080/user/login，或者 8080/book/buyBook，Gateway 会知道根据 /user 或者 /book 来转发到对应的微服务上面去，至于那个微服务的ip和端口，Gateway 不需要知道，因为 Gateway 会去 Eureka 上面去找，然后转发过去。

# 总结

2023.10.02

由这一分钟开始计起 春风秋雨间  
限我对你以半年时间 慢慢的心淡  
付清 账单 平静的对你热度退减  
一天一点伤心过 这一百数十晚  
大概也够我 送我来回地狱又折返人间  
春天分手 秋天会习惯  
苦冲开了便淡  
说什么再平反  
只怕被迫一起 更碍眼  
往后这半年间  
只爱自己 虽说不太习惯  
毕竟有限 就当 过关  
——《心淡》 容祖儿

No!	Yes!
Stateful	Stateless
serialized	parallelized
syn	asyn
HDD vs SSD	Cache
sequence	full-text
mono	micro

1. 上来就说了，我们写后台的时候，不要写有状态的，要写无状态的！这样后台的压力就会变低；
2. 然后我们说，不要串行，要并发！（事务就是最典型的例子）；在并行的时候，最重要的就是我们在os里面提到过的happen before的概念。在分布式系统里面，大家的时间很可能都不是一致的（操作需要精确到毫秒级！）不能单靠时间，要靠一些其他方法（当然这是后话）。
3. 不要同步通信！这会导致压力很大，阻塞！异步通信可以做流量的削峰；当然异步通信还包括了ajax前后端通信异步刷新页面，还有后端主动推消息的websocket。

4. 不要老是去读硬盘！要用缓存！而且我们说了，缓存的目的是因为redis这种事百分百由你控制的，而mysql里面这种缓存其实你的控制力非常有限。
5. 全文搜索！不要按顺序来搜索，做反向索引！这里我们处理的是没有结构化的非常大坨的数据。
6. 不要做单体的应用！做解耦的、大家基础设施不是共享的应用！这样保证了基础设施层面大家不竞争了，一个崩溃了其他人也都还能好好的！

# 第11讲 MySQL优化1

没有资格说感性  
但最高兴那刻 会现形  
常在怀疑着 以后 是重重逆境  
所以这样拼  
这派对后 难道你知 在哪里有星  
明年保了寿命 无法肯定  
有力气有憧憬  
明年的签证未过 温馨小店  
或游人过剩  
有风吕 跟暖酒  
你就无谓 等雪溶才尽兴  
完美在这夜擦肩 难道你可补领  
还说下次做更好 谁料这刻是尾声  
——《东京人寿》容祖儿

## 1. 优化

- 表结构是否合理，范式化？数据类型？
- 拆表？对于一张表中的数据，我们更希望是整张表中的数据全部被访问或者全部不被访问；若不然，就需要考虑拆表。
- varchar（可变长，便于压缩）和char还有text（存储一个指针，执行外部存储的原文）；char(N)占用的字节得看字符集，占用的是字符集中最长的字节数；
- 读取记录的时候，考虑到page的大小有限，如果一条记录的大小就占用了好几个page显然是不合理的，所以特别长的文本需要使用text类型，这样读取出来的text部分就是一个指针，性能更好。
- 什么样的索引科学？
- 什么样的存储引擎好？（注：MySQL其实分为两部分：Server层和存储引擎层，Server层的作用是跟客户端交互，解析SQL，进行查询优化，存储引擎层负责数据的存储和提取）
- 锁、隔离机制？所有的内存用到缓存合适吗？
- 当硬件赶不上业务的时候，该怎么调整服务器性能？
- 列最好直接设置为not null；否则，在存储的时候，在这一列上多占用一个bit，来表示是否为null，这样会浪费空间；同时，SQL在执行的时候还会多执行一句来判断你这一个字段到底是不是空的，浪费时间。解决方案：即使是空的，我也写一个特殊值进去，就不为空了。
- 能选数字就选数字，数字类型少很多字节，而不是选择字符类型，这样性能会更好。

- join的时候要注意：两个表里面的那两个列，首先数据类型最好一致；其次两个列的名称最好也是  
一致的，不要一个是“student-id”，一个是“student-number”。

下面介绍的优化方式有：

1. 索引
2. 聚簇索引
3. 复合索引
4. 前缀索引
5. Hash索引
6. Row format
7. 全文索引
8. 不常用的数据处理
9. blob优化
10. table\_open\_cache
11. innodb\_page\_size

## 2. 索引的建立优缺点

提高select性能最好的办法就是在经常查询的列上面建立索引。

索引条目充当指向表行的指针，允许查询快速确定哪些行符合WHERE子句中的条件，并检索这些行的其他列值。

索引尽可能建立在取值为**不为空**（否则请用一个特殊的值填充空值），而且值唯一（或者很少有重复的）列上面。

并不是索引建立的越多越好，不必要的索引建立会浪费时间来决定用什么索引搜索；同时索引本身占据空间！（我每建立一个索引，就相当于用这个key来建立了一棵B+树）

- 如果建立了一堆索引，对于数据更新的时候，索引改变需要调整，那么由于大量的索引建立，导致调整b+ tree需要消耗大量的时间（所以需要管理者权衡考虑！）
- 系统会自动在自增的主键建立索引（是聚簇索引，也即索引在叶子节点的指针指向的位置在磁盘上是连续的），这样最大的好处就是在join操作的时候，可以快速定位到目标。
- 当表格比较小的时候，建立索引很浪费，不如全表扫描。
- 获取要find all的时候，索引也很浪费，本来就是要把整个表数据读取出来，索引失去意义。

选择自增主键做索引的好处：

- 如果表内的某几列都很重要，但是拿不准哪几列做索引的时候，不妨直接使用自增主键
- 自增主键的唯一性由数据库保证，更加可靠

- 同时这个主键作为外键也比较方便
- 因为索引是要把这列复制出来的，因此结构简单的话，生成的索引比较小，同时生成的B+树的效率也比较高

## 3. 聚簇索引

定义：

- **索引的顺序和数据存储的顺序完全一致的索引叫做聚簇索引。** InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，聚簇索引一般就是按照每张表的主键构造一颗B+树，同时叶子节点中存放的就是整张表的行记录数据，也将聚集索引的叶子节点称为数据页。

为什么快：

- 索引记录的位置和存储的位置在顺序上是一样的，这样找起来就非常快。例如：某个节点有多个子节点，从第一个节点到最后一个节点排出来顺序是依次递增的时候，这样显然在查找一个范围的时候就非常快，可以按照**顺序来读取磁盘**。反之如果存储的地址顺序是乱的，在查找一个范围的时候就不能一次读取一片区域出来，效率就会变低，要读取多次。

## 4. 复合索引

- 根据多个列建立索引，会有不同列的优先级，比如姓名相同按照年龄排序，年龄相同安装学号排序。因此，**索引的顺序非常重要！**
- 最多16列，有数量限制。

建立三个单独的索引比建立一个复合的索引要浪费空间，B+树的叶子节点存储要索引的值还有一个指向硬盘的位置，而**建立三个单独的索引，就需要三个树**，叶子结点存储的同理，也就是说建立复合索引相对来说更好。而且调整一棵树的速度比调整三棵树的效率显然要快的。

所以复合索引的顺序非常的重要！

## 5. 前缀索引

索引长度一定是越短越好。这样一个block里面能读进来更多的索引项。

假如某些国家的人名非常长，达到了300个字符，在建立索引的时候可以只使用前N个字符；这样可以减少IO操作，单条记录的大小，然后增加每个page能够读取到记录的条目数量。

## 6. Hash索引

首先来看看B-Tree index的特点：

- A B-tree index can be used for column comparisons in expressions that use the =, >, >=, <, <=, or BETWEEN operators.
- The index also can be used for LIKE comparisons if the argument to LIKE is a constant string that does not start with a wildcard character.

在内存里面我们使用的是哈希索引；哈希查找只能做单值查找，和范围小于、大于、顺序这种都不行。

通过Hash算法（常见的Hash算法有直接定址法、平方取中法、折叠法、除数取余法、随机数法），将数据库字段数据转换成定长的Hash值，与这条数据的行指针一并存入Hash表的对应位置；如果发生Hash碰撞（两个不同关键字的Hash值相同）可以通过reHash一下，全部重新打乱，增加Hash值的长度，或者通过链表的方式往后面增加。

## 7. Row format

InnoDB存储引擎支持的row format有（默认是 DYNAMIC）：

- REDUNDANT：有冗余，假如varchar(100)，实际写入20个字符，实际存储的时候占用100个字符（前缀索引长度可以达到767）。好处：所有行全部都是等长的，这样可以快速定位到某一行，不需要遍历。
- COMPACT：紧凑的，假如varchar(100)，实际写入20个字符，实际存储的时候占用20个字符（前缀索引长度可以达到767）
- DYNAMIC：动态的，假如varchar(100)，实际写入20个字符，20个字符都是完全一样的，实际存储的时候不压缩（前缀索引长度可以达到3072）
- COMPRESSED：压缩的，假如varchar(100)，实际写入20个字符，20个字符都是完全一样的，实际存储的时候会压缩一下（前缀索引长度可以达到3072）

## 8. 全文索引

参考elasticsearch那一章节。

## 9. 不常用的数据处理

如果一个表格里面的很多查询对于这个表都不包含一些列，怎么办？

- 考虑切开这个表，然后使用join连接，拆两个表，常用的放一个，不常用的放一个。
- 总结：对于一张表中的数据，我们更希望是整张表中的数据全部被访问或者全部不被访问；若不然，就需要考虑拆表。

数据库的block按照列存储，好处就是统计一列的总和，可能效率非常高，缺点是多次插入一个行的时候非常不适合。

## 10. blob优化——巨大的数据不直接存储在表格里

blob：big large object。

是一个二进制数组；不存储在表格里面，表格里面只存指针，通过指针关联到外部存储（比如你有一个巨大的文本，你存储成为varchar，超过一个block了，那你赶紧用blob优化）。

比如，书的简介，你就全部单独放在一个表里面，然后用blob来做优化。

如果要判断两个blob相等，没有必要把两个特别大的blob读取出来，可以通过计算hash值，如果hash不等数值，那肯定不相等。

## 11. table\_open\_cache

数值越大，可以同时打开的表格的数量就越多，这个cache存储的是打开文件的文件标识符。

- 假如有三个数据库连接，三个连接同时都在操作TableA，那么缓存中就会有三个这样的表，而不是一个表（显然你开的越多，性能是越好的）；（当然redo-undo log还是只有一份，所以不会有并发错误）
- 如果有200个并发的连接，table\_open\_cache的值应该设置为： $200 * N$ , N就是执行查询的时候join表的最大数量。

1. 这个值并不是越大越好，服务器内存有限，此外到底能同时打开多少个表，还受到操作系统的限制（文件标识符受到限制）
2. 缓存满了怎么办？使用least recently used的方式，就会把这个文件描述符关掉，flush之后再加载其他表进来，也会触发落硬盘的操作！

问题：查看mysql的时候发现系统打开的表格数量好几百为什么呢？

1. 一个是因为系统表，有些执行操作的时候，需要用到系统表，所以打开了。
2. 假如有三个数据库连接，三个连接同时都在操作TableA，那么缓存中就会有三个这样的表，而不是一个表，这样也会增加一些open table数量。

3. 一些sql语句执行的时候，系统在优化之后执行可能会创建一些view或者临时表，因此就会发现打开的表数量很大。

## 12. innodb\_page\_size

数据库默认按照行存储的，如果存满了一个1 page就把这一页load到内存里面

行的尺寸和页的大小相关，每行的数据的大小应该是innodb\_page\_size的一半而且必须要小一点，因为有一些元数据在里面，需要占用空间

这就是mysql存储行的最大上限

**规则：一行里面最多放置65535个字节**

- **Row Size Limits**

- The maximum row size for a given table is determined by several factors:
  - The internal representation of a MySQL table has a maximum row size limit of **65,535 bytes**, even if the storage engine is capable of supporting larger rows. **BLOB** and **TEXT** columns only contribute **9 to 12 bytes** toward the row size limit because **their contents are stored separately from the rest of the row**.
  - The maximum row size for an **InnoDB** table, which applies to data stored locally within a database page, is **slightly less than half a page** for 4KB, 8KB, 16KB, and 32KB **innodb page size** settings.
  - **Different storage formats** use different amounts of page header and trailer data, which affects the amount of storage available for rows.

- **Row Size Limits Examples**

```
mysql> CREATE TABLE t1
      (c1 VARCHAR(32765) NOT NULL, c2 VARCHAR(32766) NOT NULL)
      ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> CREATE TABLE t2
      (c1 VARCHAR(65535) NOT NULL)
      ENGINE = InnoDB CHARACTER SET latin1;
```

ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBS, is 65535. This includes storage overhead, check the manual. You have to change some columns to TEXT or BLOBS

```
mysql> CREATE TABLE t2
      (c1 VARCHAR(65533) NOT NULL)
      ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.01 sec)
```

那么上图中，第二个插入为什么失败了？

因为一行的大小确实是65535上限，但是我们存储的是varchar，所以还需要存储一个长度，这个长度是2个字节，所以实际上是65533（因为65535十六进制是ffff，所以只能存储65533个字节），所以第三个插入时是成功的。

In contrast to CHAR,VARCHAR values are stored as a 1-byte or 2-byte length prefix plus data. The length prefix indicates the number of bytes in the value column uses one length byte if values require no more than 255 bytes, two length bytes if values may require more than 255 bytes. —— 来自mysql官方

- **Row Size Limits Examples**

```
mysql> CREATE TABLE t4 (
    c1 CHAR(255),c2 CHAR(255),c3 CHAR(255),
    c4 CHAR(255),c5 CHAR(255),c6 CHAR(255),
    c7 CHAR(255),c8 CHAR(255),c9 CHAR(255),
    c10 CHAR(255),c11 CHAR(255),c12 CHAR(255),
    c13 CHAR(255),c14 CHAR(255),c15 CHAR(255),
    c16 CHAR(255),c17 CHAR(255),c18 CHAR(255),
    c19 CHAR(255),c20 CHAR(255),c21 CHAR(255),
    c22 CHAR(255),c23 CHAR(255),c24 CHAR(255),
    c25 CHAR(255),c26 CHAR(255),c27 CHAR(255),
    c28 CHAR(255),c29 CHAR(255),c30 CHAR(255),
    c31 CHAR(255),c32 CHAR(255),c33 CHAR(255) )
    ENGINE=InnoDB ROW_FORMAT=DYNAMIC DEFAULT CHARSET latin1;
```

ERROR 1118 (42000): Row size too large (> 8126). Changing some columns to TEXT or BLOB may help. In current row format, BLOB prefix of 0 bytes is stored inline.

但是插入下面的表是成功的：

```
CREATE TABLE t (
    c1 VARCHAR(65530))
    ROW_FORMAT=DYNAMIC DEFAULT CHARSET=latin1;
```

如何理解60000+的成功了，8000+的失败？

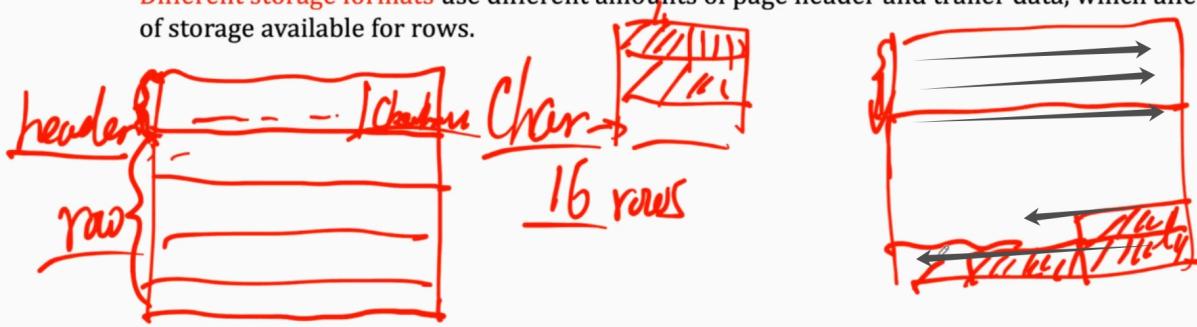
我们在做的时候，innodb的page size是16kb，那么这一行的大小就是比一半稍微小一点，就是8192，这里它告诉我们说不能大于8126；那存varchar的时候，它是这样存的：存前缀，剩下的部分存到其他地方去了。当然如果存的下，就不用一个指针指到外面去了，直接存表里面完事。

所以这也就引出了一个新的问题：**varchar和char各自是如何在表里面存储的？**

首先：CHAR最多到255，VARCHAR可以更大。

然后我们来看结构：

- Different storage formats use different amounts of page header and trailer data, which affects the amount of storage available for rows.



51

2024-10-30 16:08:44

- char: 开始一个头header，后面一行一行因为每一行长度是定长的 (char是，其他定长的数据类型也是) , 所以能推算出来长度，直接挨个放
- varchar: 这个比较tricky, header从头往后写, varchar从后往前加, 直到两个遇上为止

# 第12章 mysql优化2

寂寞攀附在等过的门  
地板裂缝在时间的河  
爱与恨总是一线之隔  
这样下去也不是不可  
我们一路停停走走  
越来越不知道自己要的是什么  
我们我们 谁也不肯承认  
捂住了耳朵 听见的笑声 是假的  
——《可以了》 陈奕迅

## 1. AutoCommit

- 含义为是否自动提交事务。
- 默认的AutoCommit=1，也就是假如建立一个数据库链接，我想数据库发送sql语句，发10条语句，这每一条语句都是会开启一个事务，也就是说开启了10个事务，这浪费性能，所以最好建议关掉，然后手动声明开启事务，这样就可以提高性能。

## 2. buffer

- buffer\_pool：缓冲池，放数据的缓冲池，这个适度增大可以让数据读写硬盘的次数减少
- innodb\_change\_buffering=all，以便除了插入操作外，还有更新和删除操作都会用到缓冲buffering
- 把大型的事务分成几个小型的事务，不要把一个执行插入特大的大量数据的事务运行，这样会特别消耗性能。
- READ\_ONLY事务：在只读事务里面出现write就会报错；只读的事务效率就会变高，数据load到buffer里；但是问题在于，别人如果改写了数据，就会出现不同步的问题；对于这种事务，如果我们希望它长期执行，可能的解决方案是提交之前刷新一下数据。

## 3. Optimize

每隔一段时间，将表在硬盘上连续分布，重建，数据和索引都发生改变，类似windows磁盘碎片整理的感觉。

整个索引在这个过程中也是会变的。

```
OPTIMIZE TABLE table_name;
```

## 4. 数据的导入导出加载

批量导入如何加快性能？

- 导入数据（导入本质就是很多insert）时，可以关闭自动提交（不然一句话一个事务很低效），还有例如UNIQUE检查（每次插入一条，和以前的比较是不是unique）、外键检查（导入订单表，关联了另外一张表book，那个表里面有没有这本书？没有就是错误的！）等；可以使用多行INSERT。当然因为关闭了这些检查，因此出错还是有可能的。

## 5. 自增模式

when doing bulk inserts into tables with auto-increment columns:

- set innodb autoinc lock mode to 2(interleaved) instead of 1(consecutive)

(不一定是主键自增，每一列都可以设置为自增的)

假设说有两个线程都在执行insert，第一个线程插入10条，第二个线程插入20条；假设说现在数据库里面key现在最大是25，那我就直接分配一批26-35给thread1，36-55分配给thread2。这就是interleaved（交错）。整体性能就会提高。

- Bulk Data Loading for InnoDB Tables

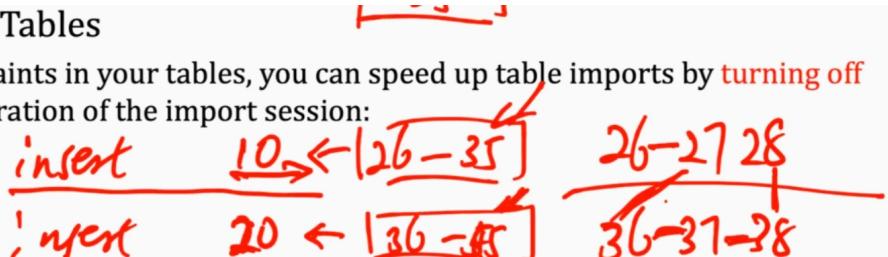
- If you have FOREIGN KEY constraints in your tables, you can speed up table imports by turning off the foreign key checks for the duration of the import session:

```
SET foreign_key_checks=0;
```

... SQL import statements ...

```
SET foreign_key_checks=1;
```

For big tables, this can save a lot of disk I/O.



- Use the multiple-row INSERT syntax to reduce communication overhead between the client and the server if you need to insert many rows:

```
INSERT INTO yourtable VALUES (1,2), (5,5), ...;
```

This tip is valid for inserts into any table, not just InnoDB tables.

- When doing bulk inserts into tables with auto-increment columns, set innodb autoinc lock mode to 2 (interleaved) instead of 1 (consecutive).

## 6. 优化磁盘IO

- buffer\_pool的大小开的太小的话，频繁落盘；开的太大的话，对内存占用变多，同时脏页也会变多
- 多个线程负责buffer的落盘（当buffer中脏页达到了一定比例时，会触发落盘）

## 7. other tricks

- newly read blocks are inserted into the middle of lru list，这样只用一次的数据能更快的被替换出去
- 把表删成空表用TRUNCATE TABLE快
- 10G才是 $16 * 128M$ 的整数倍

```
shell> mysqld --innodb-buffer-pool-size=9G --innodb-buffer-pool-instances=16
mysql> SELECT @@innodb_buffer_pool_size/1024/1024/1024;
+-----+
| @@innodb_buffer_pool_size/1024/1024/1024 |
+-----+
| 10.000000000000000 |
+-----+
```

- 实例最多开64个；最少开1G的内存（上面就开了16个实例）

# 第13讲 MySQL的备份与恢复

2023.10.03

Memories beyond the line  
선 너머에 기억이  
calling me  
나를 부르고 있어  
for a very long time  
아주 오랜 시간 동안  
In a voice I had forgotten  
잊고 있던 목소리에  
I go back against the wave  
물결을 거슬러 나 돌아가  
To the place where the sea inside me was born  
내 안의 바다가 태어난 곳으로  
Even if you are swept away and lost, you are free  
휩쓸려 길을 잃어도 자유로와  
I no longer close my eyes to the darkness that traps me  
더이상 날 가두는 어둠에 눈 감지 않아  
Never pretend not to know me again  
두 번 다시 날 모른 척 하지 않아  
Still sometimes  
그럼에도 여전히 가끔은  
There will be days when you lose to life  
삶에게 지는 날들도 있겠지  
Even if I get lost again, I know the way back  
또다시 혼매일지라도 돌아오는 길을 알아  
——《My Sea》 IU

## 0. Intro

提高数据库容灾能力。既然是备份容灾，那我读的时候也能去备份的节点上面读呀（master上read/write, slave上只read）！所以访问的性能也能有所提高。

# 1. 备份的类型

- **逻辑备份**（记住用户做了哪些操作，sql语句全部记录下来）、**物理备份**（把数据文件全部拷贝）
- **全量备份、增量备份**（所有数据全部备份还是记录从上一个时刻到现在增量）  
恢复的方法、备份数据的加密？

## 2. 物理、逻辑备份

### 2.1 物理备份

**物理备份**：拷贝整个表格，索引文件、数据文件、元数据文件，当出现故障的时候，把所有的文件拷贝回去就好了。

优点：

- 适合大的数据库。
- 恢复速度快，出现问题的时候直接把文件拷贝回去就好了，方便快捷恢复便利。
- 相对来说安全一些。

缺点：

- 比如从Mysql备份的内容就只能用于Mysql，不能像逻辑备份一样导入到别的类型的数据库。

### 2.2 逻辑备份

**逻辑备份**：先要备份表的结构，然后记录所有的SQL语句的操作，比如所有的增删改查的操作。当一个表崩溃的时候，在另外一个表里面按照顺序执行一遍这些操作，就可以实现备份的恢复。

优点：

- 因为备份的是SQL语句，所以换一个机器也能执行，换一个数据库系统也能执行。比如从MySQL导出的SQL脚本备份，换到oracle也一般可以执行，兼容性好。（你标准SQL，都可以执行的）反之你物理备份就不行了，比如MySQL->oracle, linux->windows，只做物理备份，导出来的文件肯定是不能用的。

缺点：

- 空间占用比较大，SQL的脚本文件因为还包含了SQL的关键字，所以比只存储数据的表格文件是要大的，牺牲了一部分的空间，在数据量比较大的时候，导出的SQL备份脚本文件会比较浪费空间。
- SQL文件如果泄露了的话，数据库就泄露了，安全性差。

### 3. 在线备份和离线备份

- **在线备份**: 不停机地做（热备份）；在系统在运行的时候备份。缺点管理很复杂，涉及到各种加锁，优点是用户还可以使用；
- **离线备份**: 数据库暂停服务，然后专门备份（冷备份）。缺点是用户不能使用了。
- 组合：暖备份。数据库还在跑，但是锁住不让你修改数据。

### 4. 快照

**增量式备份**的一种，一些文件系统实现允许拍摄“快照”。它们在给定时间点提供文件系统的逻辑拷贝，而不需要整个文件系统的物理拷贝。（例如，实现可以使用copy-on-write技术，以便只复制快照时间后修改的文件系统的部分。）

MySQL本身不提供获取文件系统快照的功能。它可以通过Veritas、LVM或ZFS等第三方解决方案获得。

实际上，由于copy-on-write技术，假如在某个时间我们做了一个全量备份，那么之后由于数据库的增删改，然后出现了文件数据的变化，文件相对于旧有的文件会有一部分变化，**这部分变化就叫做快照！** 所以快照是增量式备份。

**增量备份记录在bin-log里面。**

### 5. 恢复方法

#### 5.1 补充：背景知识

我们来细致地看一下MySQL的结构。

其实是分两层的，一层是上面的Server，这是面向用户的，用户发过来的SQL请求，你来做处理；第二层是引擎，InnoDB or MyISAM，默认现在用的是InnoDB。引擎在这里负责的是和文件系统交互。有一个很大的buffer缓存。有很多log：undo log，redo log，整个这一层操作的时候它看不到SQL，只看到文件。

上面的SQL是在处理SQL的，它对应的log就是**bin-log**，**它记住你执行的所有的SQL操作。**

这两层之间是可以解耦的。解耦的好处就是，我在linux或者windows上面安装MySQL，我只需要改engine这一层就可以了。

## 5.2 恢复方法

假设我们的备份策略是每个周日下午一点进行一次全量备份，也就是把数据库里面的所有数据全备份一次。那么，在两次全量备份之间，数据库每次执行写入操作的时候，都会对bin-log文件进行操作，写入增量备份。也就是记录用户对数据库的所有的操作变更。现在假如在周三的中午，数据库主机突然崩了，那么我们首先要恢复到最近一次做的全量备份：好了，现在就已经恢复到最近一次全量备份的状态里，然后要从上次全量备份到崩溃期间的binlog文件全部读取出来，然后恢复。如果每次磁盘坏死或者磁盘错误，基本上就能恢复到原来的状态。但是如果有一个binlog文件崩溃了或者丢了，那就可能恢复不到周三崩溃的时候的状态了，可能只能恢复到周二的下午或者某个稍微更早的时间点。

# 第14讲 MySQL分区

2023.10.09 听完了分区1

忘掉谁是你 记住我亦有自己见地  
无论你几高 身价亦低过青花瓷器  
评核我自己 只顾投资于爱情  
困在你小宇宙损失对大世界的好奇  
——《搜神记》容祖儿

参考文章：<https://blog.csdn.net/frostlulu/article/details/122304238>

## 1. 分区的好处

当一个表里面的数据太多了的时候，占用的空间太大，可能就接近磁盘的存储空间，这时候就需要分区，把表的数据拆开（分区），然后让不同分区的数据存储到不同的磁盘上面。

这里需要思考的一个问题就是，如果我只是简单地在两个磁盘上都有两张结构完全相同的表，只是把数据分散存在两个节点上，那在逻辑上面它们就不是一张表！它们只是碰巧拥有相同的结构而已。所以我们要做的事情就是，让这些多个节点上存储的东西在逻辑上是一张表！

分区表的优点：

- **提高性能**：由于满足给定WHERE子句的数据只被你存储在一个或多个分区上，这将自动从搜索中排除任何剩余分区，因此某些查询可以得到极大的优化。失去有用性的数据通常可以通过删除仅包含该数据的分区（或多个分区）轻松地从分区表中删除。相反，在某些情况下，通过添加一个或多个新分区来专门存储数据，可以大大简化添加新数据的过程。
- 数据可以跨磁盘/文件系统存储，适合存储大量数据。
- 数据的管理非常方便，以分区为单位操作数据，不会影响其他分区的正常运行。

对于应用来说，**表依然是一个逻辑整体，但数据库可以针对不同的数据分区独立执行管理操作**，不影响其他分区的运行。而数据划分的规则即称为分区函数，数据写入表时，会根据运算结果决定写入哪个分区。

此外，**MySQL不支持垂直分区，仅仅支持水平分区**，因为垂直分区不太符合MySQL的设计理念。

## 2. 分区的类型

### 2.1 范围Range分区

指定了某一列或者某几列，根据他们值的范围，然后来分区。

- 区域应该是连续而且不能重复的，通常都是用的LESS THEN (都是小于啊！没有等于)，比如<0的分个区，<50的分个区，< MAX\_VALUE (这玩意就是保证最后所有最大值都给你包容进去) 分个区。
- 不允许用float浮点数。臧老师的ICS告诉我们，浮点数的问题是很多的！因为浮点数是近似表示。
- 如果制定了多列，比如a、b、c，能不能把a的列做一个平方或者函数计算一下处理？不行！**不接受表达式**。
- 多列比较的是向量，比如要比较 (a,b,c) (d,e,f)，就比较a和d，比较出来小于的话就小于，比较不出来就是比较b和e，然后同理比较c和f

举例：

例：定义一个员工表，根据员工ID分区，1~10号员工一个分区，11~20号员工一个分区，依次类推，共建立4个分区。

### 2.2 列表List分区

列表分区和范围分区类似，主要区别是list partition的分区范围是预先定义好的一系列值，而不是连续的范围。

- 比如交大有闵行、七宝校区等等就可以根据这些校区的值来确定分区。

举例：创建一张员工表按照ID进行分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30)
)
PARTITION BY LIST(id) (
    PARTITION p0 VALUES IN (1,3,5,7,9),
    PARTITION p1 VALUES IN (2,4,6,8,10)
);
```

定义某个列的取值范围，根据取值范围来分区。

不同于Switch语句里面的default，采用list分区假如插入一个不符合分区list的元组，系统会直接报错。如果采用Ignore标识符，插入多个行，有的合法有的不合法，系统会插入合法的，丢弃不合法。

## 2.3 Hash分区

根据用户指定的表达式来处理，比如把学号的数字模3，然后决定是哪一台服务器存储，一般在非负的值上面处理。

特殊情况：线性哈希。

线性哈希参考文档：<https://www.docs4dev.com/docs/zh/mysql/5.7/reference/partitioning-linear-hash.html>

## 2.4 Key分区

Mysql自动根据主键散列。

## 3. 问题处理

NULL处理：如果遇到了NULL数值，这种插入的东西会被放到最小的一个分区里面（范围最低的一个分区）而对于List分区，允许in Null专门创建一个分区；对于Hash分区，null都会被当做0然后送到hash函数里面

调整分区（重新组织分区）：比较耗时间，调整涉及到整个表格

删除分区：如果用的分区规则全是小于，那么删除一个分区就会导致原来存储的数据删除，但是后面新的输入插入进来的时候就会进入正确的分区。

特别提示：不允许drop 分区（通过Hash分区或者key散列的）！一旦drop，那么整个分区的数量就发生变化，那么假如之前是取模运算，现在分区数量改变，所有的存储都要发生大变化！除非重新调整，而且不是用reorginaze，用的语法是coalesce！但是追加可以直接进行！具体原因没有讲，有自己的实现！

# 第15讲 MongoDB

2023.12.21 冬至前夕

差不多冬至一早一晚还是有雨  
当初的坚持现已令你很怀疑  
很怀疑 你最尾等到 只有这枯枝  
苦恋几多次悉心栽种 全力灌注  
所得竟不如别个后辈 收成时  
这一次 你真的很介意  
但见旁人谈情何引诱  
问到何时葡萄先熟透  
你要静候 再静候  
就算失收始终要守  
——《葡萄成熟时》陈奕迅

## 1. NoSQL

- Big Data: NoSQL
  - 只要数据分布式存储，需要增加可靠性，就需要副本replication
  - 分布式存储，使用关系型行不行？我们在分区的时候讲到了，如果一张表数据量很大，我就要拆，然后这种操作可能又需要加锁。

NoSQL是指**Not Only** SQL。

举例：

- **Neo4j**: 存储的是图数据，图数据库比较适合查找一个节点附近的相关的节点，比如查找距离为2（条边）的节点；（如果用SQL的话，就需要连两次表，效率比较低）
- **Influxdb**: 存储时序数据，比如有一个温度传感器，他就会根据传感器数据进行记录。数据非常简单由时间戳和数据组成；

MongoDB数据的结构化程度不太好，在SQL的数据里面可以画出二维的表格；当数据结构性不好的时候，就需要MongoDB类似的非结构化数据库。

## 2. DAO和Repository差异

- 一般来说，一个Repository对应的一个类型的数据库里面的一个表
- **DAO的任务是需要屏蔽不同数据库差异，让服务层的眼里只有一个一个的对象。** 比如对于“用户”，用户的头像可能存在MongoDB，其他的用户名邮箱存在SQL里面，DAO就要完成组装。

## 3. MongoDB

### 3.1 基本概念

- document：文档，MongoDB的基本单元，类似于关系型数据库的一行
- collection：集合，类似于关系型数据库的一张表
- database：数据库，类似于关系型数据库的一个数据库

所有的内容都是json的对象，key-value对。

特点：

- 支持Map Reduce
- collection是模式自由（schema free）的
- 权限控制（就不同的人可以访问不同的数据库），每个数据库存储在单独的文件
- 支持地理空间索引

这个一个 collection 里面可以放不同的document，它是 schema free 的，那是不是理论上我所有的数据扔到一个 collection？当然是可以的，我们在关系型数据库里为什么会有不同的表？那是因为不同的表有不同的结构，你往里面放的那记录去表示人和表示这个订单的或者书的数据，它就在不同的表里，它们的结构是不一样的，所以你把它拆开了。但是你这边是说document，也就是行，可以有不同的字段，那理论上确实可以把所有数据放到一个collection里面。

所以这个问题不是这么显然的，就在关系型数据库里，你可能觉得这很正常，但是在非关系数据库，就像 MongoDB 这样的，他约束这么松的话，你为什么还需要多个表？

- Keeping different kinds of documents in the same collection can be a nightmare for developers and admins
- It is much faster to get a list of collections than to extract a list of the types in a collection.
- Grouping documents of the same kind together in the same collection allows for data locality.
- We begin to impose some structure on our documents when we create indexes.

## 3.2 Auto Sharding 自动分片

- Sharding指的是分片，MongoDB支持自动分片；假如一个collection比较大，他就会把collection切成几个shard，shard其实还有更小的单位就是chunk，把这些chunk分布到多台机器上面存储
- 此外MongoDB会自动保证不同的机器之间，chunk的数量差异小于等于2
- 分区之后，还有一些配置的元数据需要记录，在config配置服务器存储，它相当于一个路由器，针对访问的请求结合元数据，把请求转发到对应的分片存储的数据服务器来获得结果
- 这样的好处就是存储的压力比较平衡，当然也支持人工的管理，因为有时候数据的冷热有差别，数据访问的频率有差别，这种情况就会手动分片，把一些经常访问的数据均衡放置，因此这种情况下，可能某一台机器存储的大小比较大，不同机器之间chunk数量的差距可能超过了2（为什么有时候也需要人工管理sharding分片）

## 3.3 shard的依据

需要选择一列，作为一个key，这个key就叫做shard key

- 例如A-F开头学生姓名作为一个分片
- 例如G-P开头学生姓名作为一个分片
- 例如Q-Z开头学生姓名作为一个分片

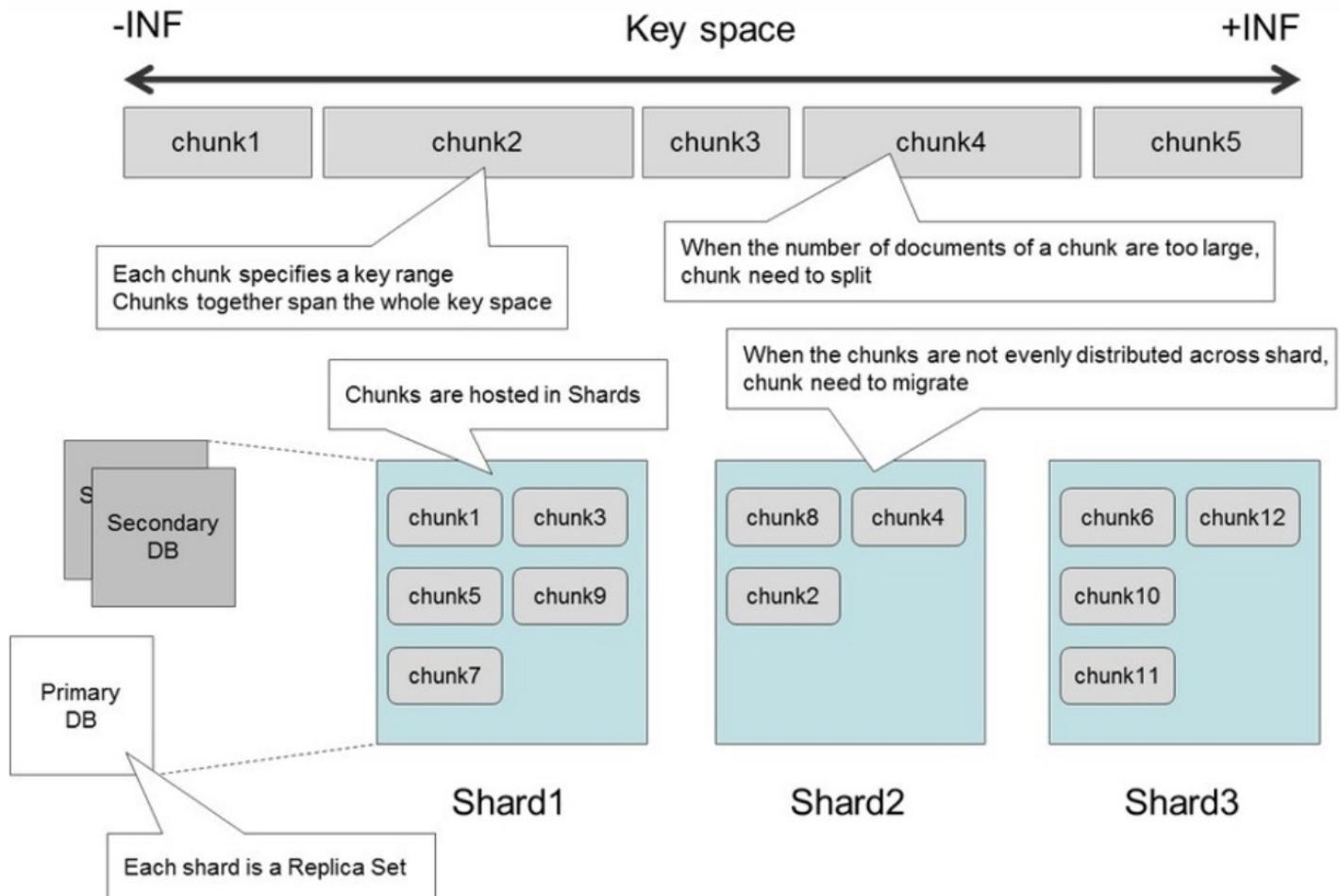
## 3.4 什么时候sharding

通常情况下默认不会是sharding的

- 当前计算机上的磁盘空间已用完。
- 希望写入数据的速度快于单个mongodb所能处理的速度。（比如大量的传感器往数据库1s可能要写入1GB的数据，如果来不及写，就需要多个MongoDB来写入）
- 希望在内存中保留更大比例的数据以提高性能，分布式缓存

## 3.5 超容量处理

如下图所示：



- 假设开始的时候分片，每个chunk代表的是某一个范围的数据，chunk就会被分布式的存储在shard服务器里面。
- 当往一个chunk里面不断插入数据的时候，这个chunk可能就会分裂，分裂成多个chunk。
- 考虑到前面所说的不同的shard服务器之间chunk的数量差异不能超过2个，那么一旦分裂之后数量违反了这个规定，MongoDB就会自动的调整chunk在服务器之间的位置，保证均衡。

# 第16讲 图数据库

2023.12.23

雨点不清楚 你已抛低我  
仍共疾风东奔西走地找你  
仿佛不知不再会有结果  
你永不清楚 你那天经过  
留下万千追忆一生封锁我  
今天可否会想起我  
——《蓝雨》张学友

## 1. 基本概念

图数据库是啥？它就是用一种特定的方法去存储图数据，就不再是我们在关系型数据库里看到的这样的一张一张的表，然后表和表之间有外键的关联，也不是 MongoDB 里边说它是一个的 JSON 文档。它既能表示一种节点和边的拓扑，然后它又能够在上面做一些特定的搜索。所以它提供的这种搜索引擎跟我们前面看到的是不一样的，它也提供了一种查询语言，它就不叫 SQL，它名字叫做 Cypher。

说穿了，它就在解决关系型数据库，必须要建立在结构化非常良好的这种数据上，因而通过外界关联去找数据或者存储数据的这样一个方式——由于这种“关系”太强，导致不能直接表达数据的自然结构，这就是图数据库想要解决的问题。

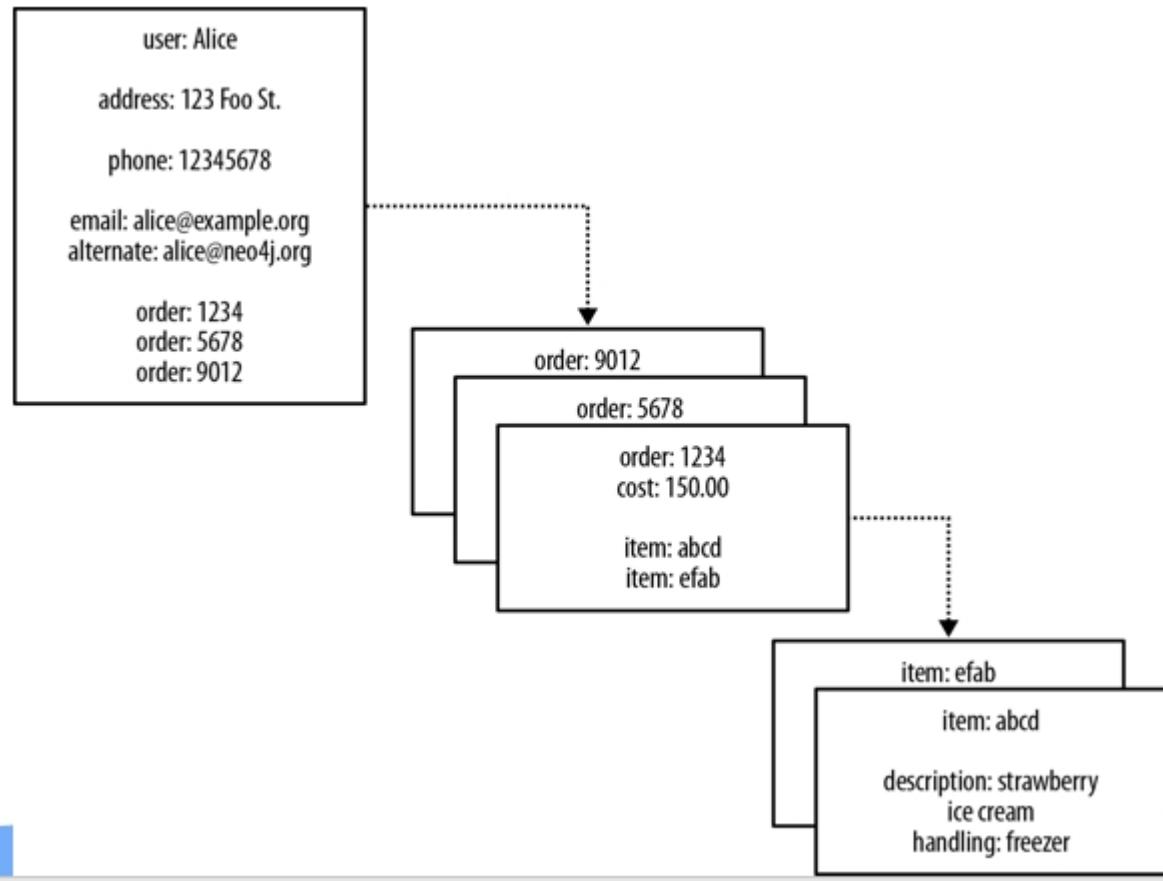
### 1.1 Pros 优点

**使用图数据库的理由：**

- 比如存储一个好友的关系，如果用 mysql 存储的话，在查找一个人好友的好友的好友需要经过多次 join 连接操作（要先做笛卡尔积，然后根据某个条件进行进一步筛选），这种在复杂的查询的情况下，mysql 的查询的效率就非常低。
- 但是用图数据库，就可以直接用一个图来存储好友的关系，根据图找到一个用户的关系网，代价比 mysql 低得多，适合距离在一定范围的好友，非常时候做用户推荐。

即便是 MongoDB 那样的文档型数据库，在那种关系比较复杂的关系存储的时候，查询的代价也会非常大。比如 **查询谁买的东西有哪些很容易，但是反过来查询就很困难**。在关系型数据库里面这个地方是要有一个外键关联的，而在 MongoDB 里面这其实只是存了他的 ID，那你说这个 1234 在这个里面能不能具体找到？他不做这个强制性的约束，或者说 MongoDB 不去做这个检查，那所以就是看起来它比较简单。

单了，那就是说我没有这个 join 这个动作，我搜出来这些之后到 order 里面去遍历，不需要做笛卡尔积，但遍历这件事情开销还是不小的。

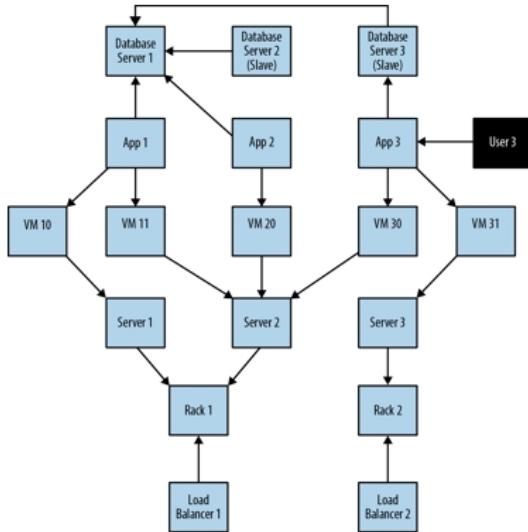


所以选择图数据就很好。

## 1.2 使用图进行数据建模

### • A Comparison of Relational and Graph Modeling

- To facilitate this comparison, we'll examine a simple data center management domain.

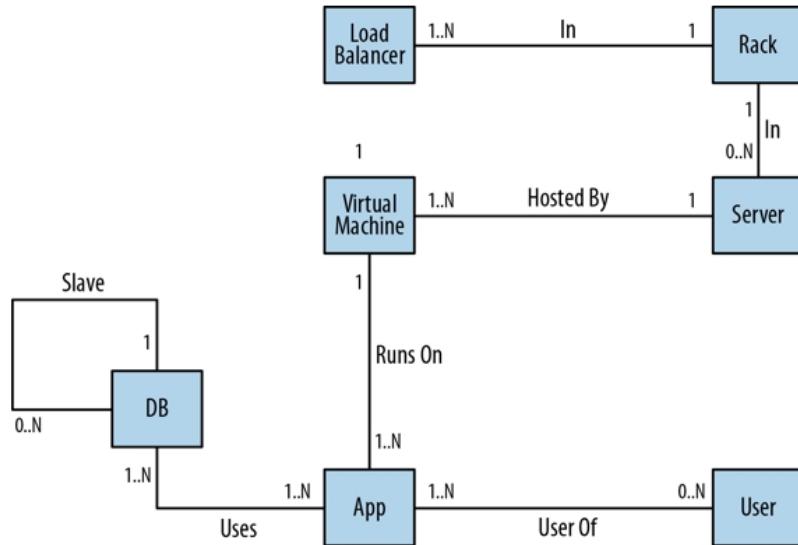


比如说有一个互联网的公司，然后我对里面所有的资产做了一个描述。用户会去用我们开发的APP，这些APP会用到一些数据库服务器，数据库服务器他们互相之间还是主从备份。

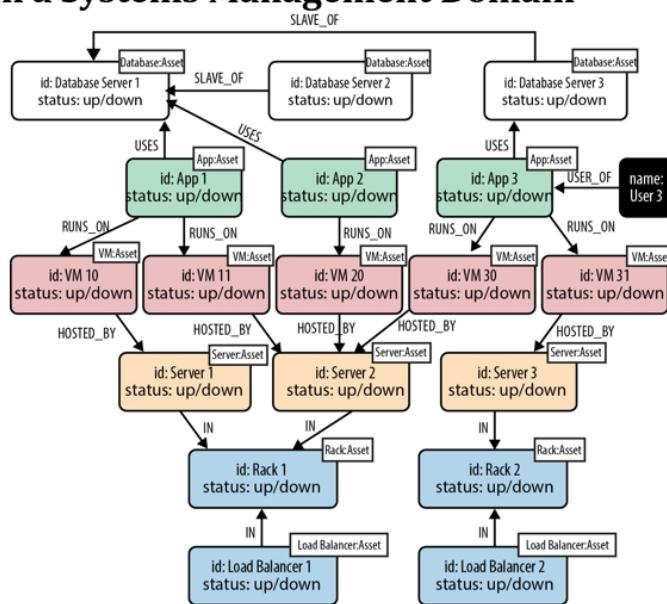
这些应用它是要跑在一些虚拟机里，这些虚拟机会运行在一些物理机上，然后这个机架它上面就会有一个load balance，就在做负载均衡，要把这个机架上所有的服务器的压力要给它均衡。然后这时候有一个人上来说他去用了这个APP3，然后他发现出了问题，能不能把所有可能出故障的资产找出？

使用图数据库建模，理论上一个user经过5跳，它能到达的这些节点就是你这张图里有可能包含的所有出错的节点。

- Relational Modeling in a Systems Management Domain



- Graph Modeling in a Systems Management Domain



```

MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User 3' AND asset.status = 'down' RETURN DISTINCT asset
    
```

上面的语句实际上就能找出下面的关系：

```
(user)-[:USER_OF]->(app)
(user)-[:USER_OF]->(app)-[:USES]->(database)
(user)-[:USER_OF]->(app)-[:USES]->(database)-[:SLAVE_OF]->(another-database)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)-[:IN]->(rack)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)-[:IN]->
(rack) <-[IN]-(load-balancer)
```

## 2. 运行

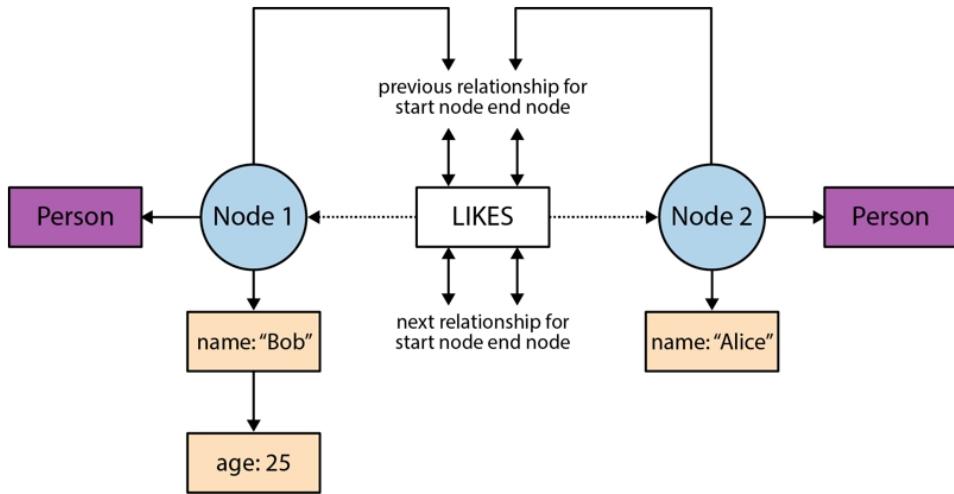
- 运行可以以单独运行，就像我们的数据库单独作为一个进程跑；
- 运行也可以用嵌入，也就是说比如和Tomcat服务器嵌入，一起作为一个进程运行；
- 运行也可以集群化部署，会重复的存储一些边缘的节点（因而把子图拼接起来的时候，边缘能够重新连接起来）。每个机器存储图的一个部分，然后最终构成一个完整的大图。

## 3. 存储原理

节点和边是分开存储的：

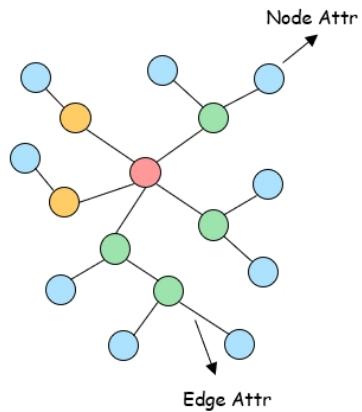
- 节点的存储占用15byte：第一个位是一个标志位，是否有关联的边是否正在使用，然后（偏移量  
1）是存储下一个边的ID关系，然后存储下一个属性的id，然后存储labels，最后存储其他的内容
- 边的存储占用34byte，第一个位是一个标志位，存储是否正在使用；之后存储这个关系的起始节点是谁，终止节点是谁、关系的类型；然后存储第一个节点（边的起始节点）的前一个关系是谁，后一个关系是谁，第二个节点的前一个关系是谁，后一个关系是谁，后面就是保留字段
- 是通过链表连起来的

- Neo4j stores graph data in a number of different *store files*.
  - Doubly Linked Lists in the Relationship Store



## 4. 图计算概论

- 图数据上的分类问题
  - 图粒度 vs. 节点粒度
  - 节点特征 vs. 边特征
- 应用场景：
  - 应用对象：社交网络、交易网络（边特征敏感）
  - 应用任务：用户分类、异常行为检测…
- 图神经网络：
  - 传统图神经网络未考虑边特征对于分类结果的影响
    - 谱图理论：Graph Convolutional Network (GCN)
    - 图注意力：Graph Attention Network (GAT)
  - 仅有少数研究考虑了GNN在边特征上的拓展
    - 仅可处理标签化的边特征 (R-GCN)
    - 仅将边特征视为权重进行处理 (EGNN)



# 第17讲 Log Structured Database & Vector Database

2023.12.21

嫌弃你想再会 被丢低想反悔  
谁叫我要靠别人待薄才配  
熟悉的想讲再会 陌生的都很匹配  
难怪我永远怀念飞灰  
如果一呼气一吸气代表相爱  
或者淹死我会更发现你存在  
——《黑夜不再来》陈奕迅

## 1. 作业讲评

### 1.1 Redis

第一个问题就是，Redis，我看了大家都做了有增删改查的4个动作，你都做，那为什么会去做呢？因为你考虑到我在下订单的时候，我会去把这个book的它的库存给改一下。那从这个角度去说，我们上课就讲了啥样的数据应该放到Redis，如果你频繁的去改在Redis里一个东西，那你还不如不要用它，因为你用它的话，你的前提就是我要先改Redis，再去改数据库，或者先改数据库，再改Redis，那我性能肯定不如我，压根就不放到里面，我就直接就到数据库里去改就算了，因为反正你每一次都要改数据库，为什么要额外多出这一个动作了？那你说放在Redis里是什么呢？我们说应该大量的只是读的数据，你修改的次数是很少的。那么从这个意义上来说，如果真想把book放进去，那么这个数据它不应该在book这张表里，你的book里面应该放的是有关它的，不会去修改的。

就我们说的一些静态数据，它所有的动态数据就像我们看到的stock，或者它有一些书评，这些东西反正会根据它的还有这个价格，或者根据它的这个，不同的时候，**它会变化的这些东西就我们可以管它叫一个动态的，那它应该在另外一张表里，然后两个表之间做一个外键的关联**，然后你把静态的东西放进去，它应该是只读的，它不应该是经常写的，如果经常写的东西放到Redis里一定不合理。

Redis和你的数据库里头，你要把它们是同步的，那怎么做？绝大多数同学就说那我先去改写Redis，再去改写数据库，然后就结束了。

那如果是这样做，那你会看到这样一个问题，就是首先这两个改写动作它肯定要在一个**transaction**里，就是你要保证他一个成功了，另外一个也得成功，否则你状态就不一致了。那么在transaction里面问题就又出来了，transaction到底在管理啥？那我上课时候说过，他要管理的是那种资源性的，这

种管理器能够管的东西包括数据库、邮件服务器、GMS 服务器，那么你这个东西在内存里行不行？那你要考虑了，你即使用了 transaction，你也可能这个 Redis 里的这种事物的这种处理需要你去自己去处理的，要去写的，那就是说在事务开始之前，你要先去记住 Redis 状态结束之后，把它之前的状态就更新掉了，**如果没有成功回滚，你要把之前状态要恢复出来**，这是要自己去做的，这是一个问题，就是你必须要保证在一个事物里去执行。

## 1.2 Web Socket

Web SOCKET 很多同学写的代码是对的，就是我在这个时刻才建立连接，当拿到这个结束的这个结果之后，这个 Web SOCKET 我就给它断掉，就这样的话不会让服务器端一直维护着大量的 Web SOCKET，就是把它断掉。嗯，先说就是在目前大家写的这个例子里面，这么做是可以的，但是你仔细去想一下，**建立 Web SOCKET 是需要双方要握手，是有一个开销的**。那如果你这个消息推送实际上是非常的频繁，而且不是说像这种应答式的这种方式，那么你这种实现就有点问题，你看你的实现是啥？是说我要发订单的时候，我 Web SOCKET 去跟服务器端接一下，通了等我，当我得到消息之后，我就把它断掉。

那你说你是不是失去了 Web SOCKET 的一半的功能？Web SOCKET 我们当时讲的是说在你客户端不发任何请求的时候，我也可以把数据推给你，你收到了你就断掉了，那我想给你推出，我怎么推？所以你貌似很正确，就是说我节约资源嘛，所以我收到之后我就把它关掉了。

那我只能说在这一个请求里面，在这个下订单

请求里面，也许看起来是合理的，但实际上你就没有用到 Web socket 的它的一个本质的一个特点。那你如果只是这样 HTTP 就可以了，因为你看整个的流程是你客户端发起，最后关掉。你本来是说我要在服务器端，在你没有客户端，没有发起任何请求的时候，我服务器端就能推一个来，那像这种设计里面你是实现不了的。所以**在绝大多数的情况下，你做 Websocket 的时候，你不应该把它给断掉**，否则的话你就用 HTTP 行了。所以**Web SOCKET 的本质上是说我要全双工，所谓全双工就是说你给我主动发也行，我给你主动发也行，你把它关掉就不存在主动发了**。那你说我们上课举的例子是说你要看这个股票，比如说啊，有个走势，那这个走势数据是服务器端不断的往客户端推的，你都把它给关掉了，我怎么推给你？所以就是说你有些东西你要反复思考一下，就是关掉，貌似节约了资源，但实际上那你就不要用 Web SOCKET 了，你就直接 HTTP 就行了。

## 2. Log Structured Database

### 2.1 LSM-Tree

- 是一种分层、有序、面向磁盘的数据结构，其核心思想是充分利用磁盘批量顺序写性能远高于随机写性能的特点。

Log:

- Log以Append的模式追加写入，不存在删除和修改
- 这种结构虽然大大提升了数据的写入能力，却是以牺牲部分读取性能为代价，故此这种结构通常适合于写多读少的场景

## 优点

- 大幅度提高插入（修改、删除）性能
- 空间放大率降低
- 访问新数据更快，适合时序、实时存储
- 数据热度分布和level相关；显然读新数据会更快

那你说它适合于什么场景？双11就是一个非常典型的场景，就是订单。但是订单里面我们看你又有两种分类。

第一个你在双十一那天不断的下单，这就是一个所谓的**OLTP (online transaction processing, 在线事务处理)**，OLTP本质上是什么呢？你在按行操作，我是在不断的插记录插这个订单进去。新插入的行就在上层，老的就在下面。

但还有一种情况是我想统计一下我双11总共卖了多少钱，所谓的**OLAP (online analytical processing, 在线分析处理)**。那这个是在做分析，那它和上面的区别是什么呢？就是我是冲着一列去的，就是你这个表里面订单里面你有很多的字段，其中有一个字段可能是最后这个总价，其实我就是要把所有的字段的这一列拿出来做了个sum，我就知道今天卖了多少钱，我只关心这一列的所有值，我并不关心这一行里面其他的值。

正常的数据库我们看到的都是按行去存储，那他对这个OLTP就是我插入一行，删除一行。但是对这个OLAP就不太友好，我要定位到每一行，还得定位到我要找的那一列，其实我读了很多根本就没有意义的这个列出来了。解决方法是数据不能按行存，要按列存。

但是你可能按行存和按列存都需要，因为你既有事务性的操作，也有分析类型的操作，这时候就是出现了一个问题，就是我们这是个混合类型的负载，我要求你能不能对混合类型的负载做支撑，就是既是在线事务处理，又能做在线分析。

我们来看一下两边：

- **在线分析处理**：对列操作，而且一般是对大量订单的列操作，需要读大量数据。只有管理员做这种操作，高延迟、低并发，大量数据。
- **在线事务处理**：对一行操作，大量做插入一行，删除一行，读取一行的这种操作。大量客户产生订单，要求高并发，低延迟，订单本身数据量是小的。

## 解决方案

1. 干脆我让每一张表有这两种存储，然后你来了请求之后，我分析一下那个请求应该是在这一边还是在这一边，这样不挺好吗？缺陷是保持数据一致性，额外开销。
2. 就一种存储，但是我会去分析一下你这些操作的特性，我看他们在数据上做什么操作，我在两个存储之间切换，并且我也不是说所有的数据都要做，比如说0-1000我按列存储，1001-

2000 我按行存，这就是你看到的现在的数据库里面它都有这样的技术。

## 缺点

- 牺牲了读性能（一次可能访问多个层）
- 读、写放大率提升

## 2.2 SSTable(Sorted String Table)

### SSTable的特点

- 存储的是<键,值>格式的字节数据
- 字节数据的长度随意，没有限制
- 数据顺序写入
- 键可以重复，键值对不需要对齐
- 随机读取操作非常高效

### SSTable的限制

- 一旦SSTable写入硬盘后，就是不可变的，因为插入或者删除需要对SSTable文件进行大量的I/O操作
- 不适合随机读取和写入，因为效率很低，原因同上一条

## 2.3 写阻塞问题

写入的时候可能有个阻塞问题，为啥？因为它写入的时候它有可能需要去做这个合并，比如 L0 层满了之后，它要往 L1 层去落，那么我刚才讲了一个极端情况是 L1 层你落下来之后也满了，又要往 L2 层落，以此类推，每一层可能都会满。

碰到这种情况怎么办？他就说这样在做 compaction 的时候，把内存里的落下来这个动作直接做了以后，如果触发了从 L0 到 L1 层的这个动作，就转移到后台去做异步的写入硬盘就不要在前台，就是说它只要落下来马上就给用，就可以去变成一个可写的一个内存表，马上就可以接收新的请求进去，后面就放到这里面，这就所谓的写阻塞，就是说你在写的过程当中就很容易不断的往下落，然后就看到这个放大就很严重，就这一次其操作其实执行了很多写得动作，然后还有可能成为这个瓶颈，这样就**阻碍了这个事物的处理的可用性**。

## 2.4 读放大问题

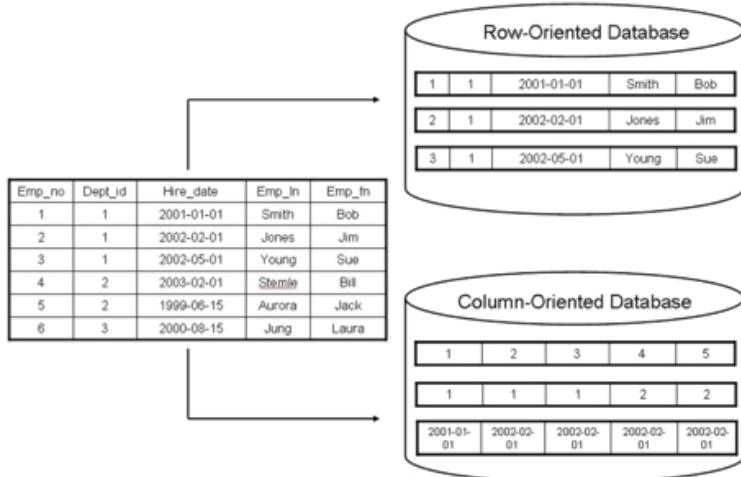
那读呢？是存在这么一个问题，就是我在找一个数据的时候，我先到内存表里；找不到，我再到这个 L0 层找，找不到我也不能说这数据不存在，有可能它太老，它落到底下去了，于是我一层层找，极有可能就是最后你在很找了很多层你才找到。而且你在不同的层里面可能还存着不同的版本。因为我们刚才说了，它的改写是通过追加得到的，那也就是说你如果有老版本的话，它会在更低的层里面，如果没

有经过compaction，这个数据还是在的，只有 compact 才有一次机会可能把它给删掉。**限制了AP查询的性能。**

## 解决方案

### 在RocksDB中增加列式存储

列式存储在访问少量列时磁盘读取量更小，可以减少读放大的开销



### 行式存储(原)

- 适合事务处理(TP)
- 数据写入不需要拆分属性

### 列式存储(新增)

- 适合分析处理(AP)
- 便于分类压缩
- 对内存友好

## 提出混合存储策略

=>常做事务的数据以行式存储,

=>常做查询的数据以列式存储

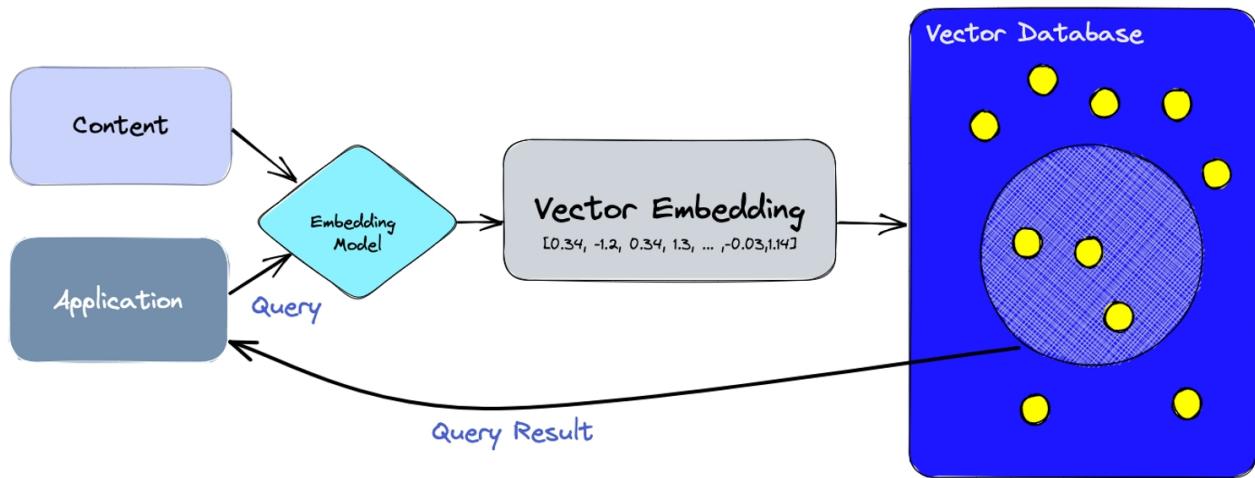
## 以磁盘读写开销为格式转换的指标

- 以文件为格式调整粒度
- 额外记录每个文件的历史操作
- 计算每个文件在行式存储和列式存储下重做历史操作时的磁盘读写次数

## 兼容原有的后台逻辑

- 新增主动的conversion后台过程  
监控所有文件，当满足【原格式读代价 > 新格式读代价 + 新格式写代价】条件，就触发原地格式转换
- 结合原有的compaction过程  
新文件产生时，根据祖先文件，选取读写代价总和最小的格式

### 3. Vector Database: Pinecone



我会有一个，比如说训练的样本，我给了这个一个 embedding 的模型，embedding 的 model 就是刚才我们看到的一张图，本来是  $200 * 300$  再乘以 3 这样的数据构成的，那它是怎么变成这个模型的输入呢？我是把它全部拆开来变成一维的，那这个模型的作用就是在把它产生对这个神经网络的这个输入，他要做这样的一个事情。

那这个不一定像我刚才说这么简单，我就直接把它拆开了。比如说我还可以说一个你，你是这个颜色 RGB 是 0 到 255 的吗？那我也可以把它多放一些，缩到 0 到 1 之间，就还可以做一些预处理。那我还可以说我是不是在里面做一些差值，或者说过滤一下，反正就是总的来说你会有这么一个模型，就产生了这个 content 的一个向量的一个表达方式，然后它就在数据库里存储的时候，向量就是  $n$  维空间，它你就可以想象为是在  $n$  维空间里面实际上就是一个点。然后当你的应用来了之后，你会查给他一个 query，这个 query 就会在这个数据库里面，这个  $n$  维的向量空间里面去找最相似的。比如说你这一个 query 表达的那个概念就是在这里的，像刚才我输入的 0.10.20.3 就是在这里面的某一个点，那他就会在一定范围内按照你的要求，比如说 TOP50，他去找出来离他最近的。相似度的计算方式允许你自己设定。

找一个最接近你描述的这个场景出来，或者说我在设计一个下棋的这样一个模型，当你下了一步棋之后，我用一个向量去表示当前的棋盘，我就在我收集到的所有棋谱里面去找跟它最接近的那个棋，然后我就知道该怎么下，因为那个棋盘也是按时序的，第一步谁第二步谁放到一起，我就到里面去找所有的棋盘里面所有的步数下完以后的样子，看他跟谁最接近，然后看那个棋盘里面要想赢下一步应该在哪里下，我就下那一步棋。

# 第18讲 influxDB 时序数据库

2023.12.21

看我脸上的苍白 看到记忆慢下来  
过去甜蜜在倒带 只是感觉已经不在  
过去你给的期待 被我一次次摔坏  
已经碎成太多块 要怎么拼凑跟重来  
——《倒带》周杰伦

## 1. 概念

时间序列数据库（TSDB）是针对时间戳或时间序列数据而优化的数据库。

时序数据：比如说一个地图，我这是一个，比如说一个花园，我在里面布了很多的传感器，这些传感器就是它的温度和湿度，我来做这个自动浇灌，然后他们就会源源不断的产生数据，我要把它存起来，那可以看到它们源源不断产生数据，于是我们看到这数据有几个特征。

- 第一个：这个数据只要你往上发，它一定**带一个时间戳**，我要知道你这个数据是什么时间发上来？这是第一个问题。
- 第二个：你一定会有大量的这种传感器在传数据，所以它必须要支持这种**高并发**，就**吞吐量要大**，这种吞吐量必须要大才能支持高并发。
- 第三个：你说传感器如果他在报数据，他偶尔丢掉一两个或者一两个不太准，它会不会影响你的这个数据整体的效果？  
就是他如果说每一秒钟都报一个数据，在某一秒他突然丢了一个数据，它会不会影响你做统计的效果？就理论来说，它里面的**数据允许少量的不准确或者是缺失**，也就是说它的数据和那种 transaction 的数据就形成了鲜明的对比。
- 第四个：就是其实我们**对单点的数据是不感兴趣的**，你说我要知道某一个时间点，这里面每一秒都要发数据，我要知道在某一秒的数据，这个肯定你不太关心，你关心的是**一个时间区间内**，比如说在一小时内它的数据什么样的，我要去求和。所以在持续数据库里，他对单点数据的访问不会特别多，他经常做的是对一块数据，尤其是时间片划分出来的一段数据，他要做一个访问。

1. 整体的存放是用了类似Ism日志结构合并树的结构；时序数据库的生命周期一般比较短，比如一个传感器，可能我们只关心最近一周的或者一个月的数据，超过的时间的数据就可以删除（比如当掉到L0以下，我们就将其压缩，甚至删除）。（类比关系型数据库的订单数据，订单数据必须要持久化保存，但是这种时序数据库可以**通过摘要压缩一下，或者直接删掉**）

- 不太可能会建立索引，例如记录温度随时间的变化，一般不会有必要建立索引（比如建个温度在时间的索引，这没有意义）
- 对于时序数据库接受的数据的特点：更多的数据点，更多的数据源，更多的监控，更多的控制

### Telegraf & InfluxDB

InfluxDB是数据库； Telegraf是数据采集器，可以采集各种各样的数据，然后把数据写入到InfluxDB里面去。

## 2. 优化

- 可以不记录时间戳：如果传输来的数据比较稳定的话，例如每秒1个数据，我可以记录某一个特定数据（比如开始的数据的时间戳）然后后面每秒记录一个数据就好。
- 可以记录数据的增量：比如记录温度，变化比较小。我就记录相比上一个数据增加或者减少了多少度。这样存储的数据的空间得以节省了，所以可以存更多的数据。

## 3. InfluxDB

### 3.1 基本概念

**bucket:** my\_bucket

_time	_measurement	location	scientist	_field	_value
2019-08-18T00:00:00Z	census	klamath	anderson	bees	23
2019-08-18T00:00:00Z	census	portland	mullen	ants	30
2019-08-18T00:06:00Z	census	klamath	anderson	bees	28
2019-08-18T00:06:00Z	census	portland	mullen	ants	32

The diagram illustrates the structure of InfluxDB data. It shows a table with six columns: \_time, \_measurement, location, scientist, \_field, and \_value. Below the table, six labels are aligned with arrows pointing to specific columns: 'timestamps' points to the \_time column, 'measurement' points to the \_measurement column, 'Tag value' points to both the location and scientist columns, 'Filed value' points to the \_value column, and 'Field key' points to the \_field column.

- Bucket就是类似于关系型数据库中“表”的概念；没有库的概念，因为不像关系型数据库中有外键关联的概念。
- 带有\_开头的，都是系统保留的字段，也就是一定会有的一个列，反之都是用户自定义的字段。
- \_time：时间戳，数据对应的时间，因为可能同一个时间会接收到很多数据，所以很可能同一个时间接收到很多数据，所以时间戳不能唯一的标识。时间戳非常准确，精确到纳秒级别。

- `_measurement`: 起一个名字，一个统称，这个表格在干嘛。census就是调查种群数量。这里我们发现它是共享的，所以存储的时候会优化，只存储一次。
- `_field`: 存储的是key，比如下表里面存储的就是某个物种的名字；和`_value`构成键值对。Field可以作为筛选的依据，得到一个Field Set。
- `_value`: 存储的是value，类型可以是strings, floats, integers, or booleans，之所以不能是别的，是因为如果是复杂的数据类型转换会耽误时间，效率降低，所以就只能存这些基础的数据类型。

## 3.2 Schema优化

上图中，location和scientist都是tag，`_field`和`_value`都是field。

- **Fields不参与索引**，必须扫描全表；如果经常被访问，不适合以这种形式存储，就应该以tag的形式存储。
- **Tags参与索引**，当然查询起来就会更快。当然索引本身是有开销的，并且如果你进行写操作，索引也会更新，带来更多开销。

If our sample census data grew to millions of rows, to optimize your query, you could rearrange your schema so the fields (bees and ants) becomes tags and the tags (location and scientist) become fields:

_time	_measurement	bees	_field	_value
2019-08-18T00:00:00Z	census	23	location	klamath
2019-08-18T00:00:00Z	census	23	scientist	anderson
2019-08-18T00:06:00Z	census	<b>28</b>	location	klamath
2019-08-18T00:06:00Z	census	28	scientist	anderson

_time	_measurement	ants	_field	_value
2019-08-18T00:00:00Z	census	30	location	portland
2019-08-18T00:00:00Z	census	30	scientist	mullen
2019-08-18T00:06:00Z	census	<b>32</b>	location	portland
2019-08-18T00:06:00Z	census	32	scientist	mullen

influx的数据都是以append的方式往里加（类比Ism，但是实际上叫做时间序列合并树）。

### 3.3 influx中的其他基本概念

- Series: measurement, tag set, 和field key都相同的点集合。
- Point: 一个数据点，带有时间戳的数据点。  
E.x. 2019-08-18T00:00:00Z census ants 30 portland mullen
- Bucket: 存储桶，归属于一个组织，存储相对应的数据点集合。
- Organization: 组织，里面有一组用户，里面有若干个bucket。

### 3.4 InfluxDB设计原则：何以支持时序数据？

- **严格按照时间组织**，按照时间顺序递增追加append，不会出现说我突然插入一个以前的时间戳
- **严格限制update和delete**。首先本来就是传感器送过来的数据，你改它干吗？而且，你明明一直在追加，你说我往里面去改写，那它就不是顺序读了，它要做一次随机读，它效率就降低了。改一下有可能尺寸会发生变化，要么留下空洞，要么它存不下后面东西所有都要往后挪，所以它就严格限制你不能这么做。所以他认为时序数据主要就是新数据，他们是从来不会发生变更的。

- **数据的读写优先。** 你说这个是什么意思？你数据库里就是读和写，然后你说他们优先，那还有啥东西？就是我们的数据库 InfluxDB，那它也许会是分布式处存储的，那它可能一主两从，那你这些中间是不是就会有同步的问题？这是一个动作，这是他内部要去实现的，但是他就告诉你，如果你现在有这个读和写，就指的是用户发送过来的读和写的要求的话，必须优先处理，哪怕你没同步好也要处理。它实现的是最终一致性，而不是实时一致性。因为它认为系统的实时性比较重要——开始我们讲过了，我们是允许时序数据中一些小小的错误的。
- **幂等性。** 应用场景里面就是大量的传感器，会把大量的数据通过不可靠的或者不那么可靠的网络传递过来，那我就不能保证它没有被发送多次，但是它发送多次它也不能对我的数据库产生影响，我只存一个（根据时间戳）。比如除了value之外的部分数据都是相同的，这样的数据提交了三次，不会像mysql一样插入三行相同的数据，InfluxDB会用最新的一个数据存储。
- **没有传统意义上的id。** 因为数据集比单个点更重要，所以InfluxDB实现了强大的工具来聚合数据和处理大型数据集。点通过时间戳和序列来区分，因此没有传统意义上的ID。

## 3.5 Storage Engine 存储引擎

整体是TSM (Time Structured Merge Tree) 结构，类似于LSM的结构。

落到比较下面的层就按照列压缩。为什么按照列存？我们关心的是给定时间段内列的value，按照列存，这些讯息就可以连续存储了。

## 3.6 Shard

数据量大了，mongoDB中就是把数据压成很多shard，neo4j也是。时序数据库是最简单的线性表，因为并不存在什么诡异的外键关联这种，直接切就完事了。

1个 bucket 可以分成若干个 shard group，这个 shard group 里面又分成若干个shard。意义？首先是我们可以做分布式存储。还有，类比我们在关系型数据库里看到那个partition，就是我如果要查某一天的数据，你不用去做全表扫描，或者建一个全表的索引去做处理，我就定位到这个 shard group里去查找。如果跨越几天，我就查符合的两个三个shard group。



# 第19讲 GaussDB

2023.11.20-11.22

若是我记得你 亦是无须紧记  
习惯一个人没有伤悲  
而无论旧时说爱多美  
再过半天你便记不起  
若是要等你 亦是无须等你  
遗留下这个世界向着前飞  
纵爱理不理纵隔千里  
谁预知将来或再一起再恋上你  
——《一个人飞》李克勤

## 1. 课前review： MongoDB什么时候使用？

为什么我们需要MongoDB？这个一定要从它是**schema free**的这一点来出发。

三星和苹果的手机，就它俩的属性不完全一样，也就是说大家是 schema free 的，就你有你的属性，我有我的属性，在这种情况下，那我把它存到一起都在这个 mobile 这个 collection 里。

你如果用三星里面它特有的，比如说它价格比较便宜，用它特有的价格去搜索，尽管 apple 里面没有 price 这一个属性，你这个搜索的语句一定能搜出来符合要求的东西。所以你是在 schema free 的这样一个数据库上去存了大家不一样的数据，然后我的搜索还可以按照不一样的属性去给它找出来，不是说大家都要有相同的这个属性，所以这个场合才是MongoDB很有用的地方。

还有什么是 schema free 的？其实有一个东西是，就是如果我对书有一些评论。那是不是就是树形的？就是说这本书还有一个评论，它还可以有跟帖，它的跟帖还可以有跟帖，它的跟帖可能有好几个，跟帖的跟帖也可以继续下去。然后另外一本书它也有，但是它到底有几层？然后每一层的跟帖有多少个？这是不一定的。

就大家这个是没有什么说，大家的跟帖最多到三层，每一层最多 5 个，然后我就做了个表，没有的，那这时候你看到你要从这种书评的话，那它确实就是 schema free 的，然后你就可以把这些东西就存成一本书的某一个属性往里面去追加。那如果你要在 MySQL 里去存，那它最带来的最大问题就是，那你要去找这个书评的时候，你这么这么多层，你怎么去处理？你是把列转成行存呢？还是说你怎么用其他的方法？反正就比较麻烦，所以这个是想给大家解释一下，就是你要用 MongoDB，你千万别觉得他只是拿来存图片用的。

## 2. 云数据库

GaussDB是一个分布式的数据库。

- 华为的openGauss不是云数据库，搬上云GaussDB才是云数据库。
- openGauss是一个高性能、高安全、高可靠性的企业级开源关系型数据库
- openGauss是一个单进程多线程的架构，但是MySQL是一个多进程的架构，很可能一个数据库连接就对应一个进程。使用单进程的话可以降低通信的开销。多进程需要使用到进程间通讯，而单进程的话可以使用共享内存的方法。
- 单表大小32TB，单行数据量支持1GB

分布式的困难？一个关系型数据库，要想做分布式就很恶心了，就困难在你把数据怎么存储才能让这个数据库它的性能会比较高，比如说要做一个查询，正好这个查询涉及到数据全部存在不同的节点上，那你要把数据全部的汇聚到一个地方，这就很恶心。

### 2.1 云数据库的概念和特点

什么是云原生？和本地数据库的差异是什么呢？

云数据库的一个最大的特点是什么？比如说在华为的机房里有很多很多的服务器拿出来去给你用了，然后你也别管它有多少物理服务器，你就说我需要一台虚拟机，那理论上就是这个机房里所有的机器全部可以给你拿来用（比如支持在线弹性伸缩），你只要有钱就可以——对你来说就是你不要管这个机房里有多少机器，**你眼睛里看到的是一台超级计算机**。我把所有的硬盘合到一起，形成了一个网络文件系统，我再把机器上运行的这些CPU通过网络合在一起，形成了一个超级CPU。

然后我所有的任务来了之后，就在这些CPU上去调度，利用底下所有的硬盘合起来构成的网络文件系统去存储数据，这就是**存储和计算的分离**。

这就和大家看到的本地的计算机不一样，甚至跟你本地搞了5台计算机，大家建一个集群也不一样。集群的目的是做主从备份——一主四从，做读写均负载均衡，但是你部署为一个云的话，在我眼里看到的不是五台机器，是一台机器，所有的任务可以在这上面去进行调度，这跟你五台机器彼此之间隔离，然后大家互相之间去做备份是不一样的。所以他这里就讲了我们是一个分布式的共享存储。

为什么是共享存储？就刚才讲它是一个超大的一个存储，一个网盘，然后在这个基础之上要做高可用的容灾，就是我所有的服务器有可能会出错，一旦出错马上就有其他的顶上来，然后我任务和数据就做一次迁移。

## 2.2 GaussDB的分布式优化器



你写了一个 SQL 的查询，他就先做语法和解析，就看你要干啥，然后生成一棵查询计划树。它的执行效率就可能不高，所以就根据我预先定义好的一些规则去优化。

优化的东西有包括什么呢？比如在分布式场景下，比如说我有两张表去连接，那你可以 T1 连接 T2， T2 可以连接 T1。

所以经过这一系列的东西，最后你会发现你写的这个 SQL 语句，你可能压根都没想过数据是分布的，然后还有什么将来能并行的。但是经过这一系列的处理，最后它产生的是在多个机器上，而且是并行去执行的，这样的逻辑就跟你写的不一样了，中间经过了一系列的优化，就在防止你写的东西很烂。

## 2.3 GaussDB多租户

如果存的时候每一个数据库都是所有人各自独占，完全不一样，那么带来的一个问题是我这个很难去维护。微信小程序它的一个基本的原理是：它要用到微信本身自己提供的那些服务去开发他自己的东西。那么如果我要是看到所有的这些小程序，他们都有各自完全不同的数据结构去存储，然后还要我对他们去进行支撑，我就一个数据库，我如何为所有的小程序服务？

换句话说，你有一个数据库服务器的实例，你的东西去给不同的租户，他们每一个人都在上面可以存他们的数据，但是他们彼此之间都认为自己是有一个独占的数据库。



# GaussDB多租户



## 口数据库

- 资源管控与隔离
  - CPU、内存、磁盘
- 数据隔离
  - 数据多租
  - 租户索引
- 日志隔离
- 弹性伸缩

?

WX



13

设我现在有两个应用，一个应用说我这个数据库里面这个表，它就一张表，他说我的字段分别是a、b、c、d、e。那有另外一个用户，他说我的字段是这样的：a、b、c、d、f。如何服务这两个用户？

- 第一种方法：single table，我只有一张表，没有个性化的东西，最后表有个表示oid (owner id)，用这个标识来区分数据是谁的，我们就可以直接隔离了。缺点：schema要求是一样的，不支持个性化了，事务的管理也会比较复杂，并发量高，访问性能就会下降。
- 第二种方法：不是一张表。第一张表比如我存abcd+oid，第二张表存oid+主键（关联到第一张表）+扩展key+扩展value。这里其实就是把自己特有的列转为了行来存储。
- 第三种方法：算了，大家还是别存在一起了。第一个应用有自己的表，第二个应用有自己的表，但是大家还是放在一个db里面。
- 第四种方法：干脆一个人一个db，彻底分开。（最不好，最浪费资源的方法）。

站在数据库的角度来看：

1. schema 如果有一些可以共享，那么我这个数据库就好管理，我既可以支持你有这个共同的东西，也可以支持你个性化东西。
2. 硬件的资源要隔离开，某一个租户挂了不会影响其他人。

## 2.4 openGauss的查询优化

因为数据量巨大，所以优化是值得的，否则数据量小，比如别做优化，优化反而花时间。

## 2.5 行存储和列存储

- 行存储：适合业务数据的频繁更新，比如插入更新一条数据，类比双十一开卖的时候订单很多条目要插入进来
- 列存储：支持业务数据的追加和分析场景，类比双十一结束的时候需要统计销售额【注意：在批量加载的时候为什么更适合列存储呢？因为列存储的时候压缩率高，比如某一列连续下来存的是5个连续的1，我就可以压缩一下，或者我时间戳的话发现可以记录一个增量，这样就大大的优化了存储的空间，使用列存储的话更加的高效】
- openGauss的引擎会根据执行的SQL语句判断是在行还是列存储上进行查找
- 同时做行存储和列存储的缺点是浪费了空间，但是好处是适合的SQL会发到对应的存储上，速度会明显提高。
- 内存表：数据整个在内存里面，大量的数据一次性加载到内存里面，以后的所有操作全部都在内存里面，可能只要我不关机，我可能数据一直都不用落硬盘，每天可能为了备份，把数据落一次硬盘。（那和我们之前说的mysql有什么区别呢？之前所说的buffer比较小，我们只能把正在操作或需要操作的数据加载到buffer里面，但是内存表要求很大的缓存，开机的一刻就把数据加载到内存中）
- 应用场景：比如存储一系列风力发电机的厂家、型号、风场信息用行存储，但是如果风力发电机不断的发时序数据的话，这些数据应该更适合列存储。同时需要行、列存储的场景就是这样

## 2.6 ORM效率

有人可能有疑问：ORM映射是通过把对象的操作通过JPA工具生成SQL语句，然后发送给数据库进行操作的，那如果我自己直接写SQL语句的话不是避免了这个转换的过程，一定会有更高的效率吗？

回答：错误的。JPA工具里面有大量积累的优化的SQL的经验，会把对象的操作转换为基本上说是最优效率的SQL操作语句。但是如果你自己写SQL语句可能效率远远比不上。但是有人说数据库不是有查询优化吗？实际上数据库的查询优化很有限（课上的举例就是底板不行怎么打扮都不好看）所以说通过ORM映射产生的SQL基本就是最优解。此外还有一种情况，比如数据库里面有一些函数需要被调用，他们产生的是一个结果返回我，数据的处理就是在数据库里面实现的，如果用ORM的话可能会写一个遍历，那这样的情况可能ORM就比较差。

## 2.7 GaussDB的分布式事务

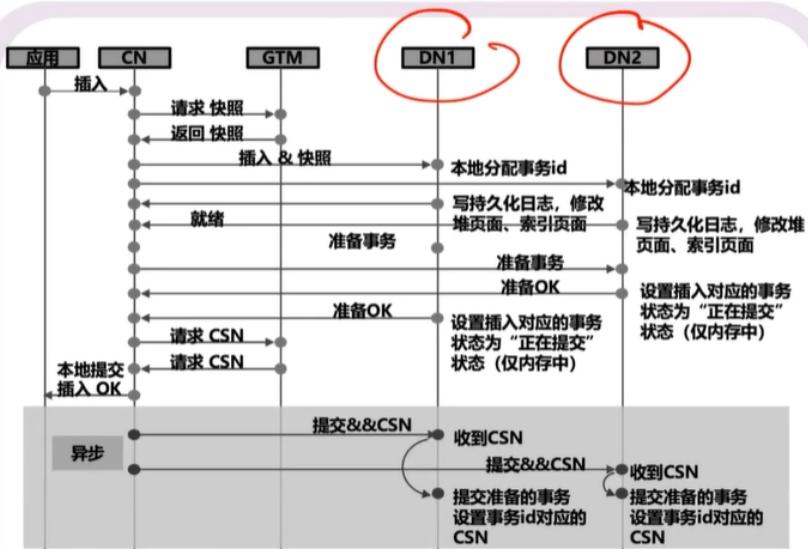


# GaussDB分布式事务



➤ 在分布式事务（含有多个DN）中，第二阶段事务提交改为异步方式，只同步做两阶段提交的准备阶段

- DN处于准备状态的事务依赖对应CN上的事务是否提交，如已提交，且CSN比快照CSN小，即为可见；
- DN上处于准备状态的事务，
- CN上的事务不处于提交状态，
- 则必须判断是否是残留状态，
- 如果是则进行回滚。



如果你只涉及到一个数据节点，那这是一种情况，如果你涉及到两个数据节点，那就会有一个两阶段提交协议。

优化是说在第二个阶段它是异步的；他发请求出去，然后他也没有去等着他们；回没有回来的这个线。也就是说这个灰色的部分相当于在后台慢慢去处理，前端这端他马上就到OK了，就此处后面这端他会想办法去让他一定能提交的，那么这是他做的一个优化。

为什么要有异步？举个例子：

比如说有一个数据采集器，它每隔一段时间就会去采集一下数据，然后把数据写到数据库里面去。但可能数据发过来非常的快，第一个数据还没处理完，第二个数据就发过来了。同步的情况下，必须是前端发过来一个数据，后端server去处理，处理完了和前端说哦我处理完了，前端再来下一个。这就会出现问题。

什么意思？就是在现在这个场景之下，是我们不断的来事件，事件在没有被处理完之前会有新的事件进来，我们要依次要去处理，它不能丢。但是我每一个处理之后不是同步在处理，不是说第一个处理完了，第二个才能来，那这种东西叫啥呢？叫流式数据。

# 第20讲 DataLake 数据湖

2023.11.22

应该早已没期待  
应该心死为何仍未放开  
应该不要回来任你伤害  
恋什么爱 你精彩 我悲哀  
——《我应该》 张学友

## 1. Data Warehouse 数据仓库

数据仓库：

- 多元的数据如何融合处理？数据仓库：把很多来源的数据加载之后，把他们安装同一个方式存储。
- 比如交大的数据、复旦、同济的数据，可能分别有10、11、12个字段的数据，需要把这些数据“抽取、转换、存储”，通常来说把不同的文件转存储到Parquet/RC File的文件格式
- RC File的文件：把原始的关系抽取某一列或者几列、甚至把整个表存储起来。同时文件最前面还有有一些元数据表达的行号是哪几行、或者是否有压缩，压缩后key长度。然后这些文件被建成HDFS的块。
- Parquet：类似RC File。
- 存在的问题：需要**ETL（抽取、转换、存储）然后写入到Parquet的中间文件格式，这个的代价是很大的**。而这些文件在之后如果直接做SQL的时候效率很低。所以最终还是要转到SQL的数据库中（直接对数据库处理）那我们就想：如果数据来了，我们可不可以先把数据按照原始的格式存储，当我真正需要的时候，再把数据导入进来。所以这就是数据湖。

## 2. Data Lake 数据湖

数据湖：

- 以**自然/原始格式存储数据**的系统或存储库，通常是对象blob或文件
- 数据湖通常是一个单一的数据存储，包括源系统数据、传感器数据、社交数据等的原始副本，以及用于报告、可视化、高级分析和机器学习等任务的转换数据。
- 数据湖可以包括来自关系数据库的结构化数据（行和列）、半结构化数据（CSV、日志、XML、JSON）、非结构化数据（电子邮件、文档、pdf）和二进制数据（图像、音频、视频）。数据湖可以建立在“本地”（在组织的数据中心内）或“云中”（使用亚马逊、微软、甲骨文云或谷歌等供应商的云服务）。

1. 在必要的时候会通过ETL的方法，把数据导入到数据仓库的里面
2. 数据湖要解决的问题就是：面临大量的数据要导入的时候怎么把原始数据快速接受下来

### 3. Data Lake vs Data Warehouse

数据湖和数据仓库的四个不同点：

	Data Lake	Data Warehouse
Data Structure	Raw	Processed
Purpose of Data	Not yet defined	Currently in use
Users	Data Scientists: 针对原始数据进行处理	Business Professionals: 主要针对商业金融的分析、统计
Accessibility	Highly accessible and quick to update：因为不需要去做etl这样的动作	More complicated and costly to make changes： 可以理解为主要是数据的一次导入， 以后只是做分析的

特点	数据仓库	数据湖
数据	关系数据	各种数据（因为它是直接存储的原始数据： 结构化半结构化等等都可以）
schema 模式	比较严格的模式， 数据在读取的时候或者写入的时候做一个校验 (schema-on-write或者read)	schema-on-read（只有在数据分析的时候， 也就是从读走数据的时候校验数据的模式是否合法， 因为它是直接存储的原始数据，存储的时候不校验）
性能	本地存储的时候非常快	用低成本的存储把大量的数据存储进来， 当搜索的时候可能比较慢
数据质量	schema-on-write，过滤了不合规定的数据	因为所有数据直接接受，质量无保证

- 湖仓一体：不管是谁要来查数据，都到底层去抓数据，如果是关系型的就去关系型的找数据，非关系型的就到非关系型的去找就好了，不严格区分数据湖和数据仓库。

# 第21讲 Cluster 集群

就让我们虚伪 有感情 别浪费  
不能相爱的一对 亲爱像两兄妹  
爱让我们虚伪  
我得到 于事无补的安慰  
你也得到 模仿爱上一个人的机会  
残忍也不失慈悲  
这样的关系你说 多完美  
——《兄妹》 陈奕迅

## 1. 为什么需要集群

- 有许多用户，可能在许多不同的地方（高性能）
- 系统是长时间运行的，不能终端服务（可靠性）
- 每秒处理大量事务
- 用户数量和系统负载可能会增加（比如open ai）
- 代表可观的商业价值(比如支付系统是一个很关键的系统，但是类比电子书店的浏览功能就是个很随便的功能，因此为了保证支付的可靠性，很可能需要把常规的业务服务和支付服务分开，保证安全和可靠性)
- 由多人操作和管理

概念：

- 节点：每个服务器都是一个节点

对于用户来说，看到的就是单一系统（类比微服务中的gateway）

## 2. 负载均衡

负载平衡意味着在集群节点之间分配请求，以优化整个系统的性能。

- 方法一：负载均衡器用来决定为请求选择哪个目标节点的算法可以是系统的或随机的。（优点：负载均衡的效果比较好，不同的机器轮流来处理，一般来说比方法三的效果好；缺点：session需要单独维护，以及没有方法二的优点）（**round robin**）
- 方法二：负载均衡器可以尝试监视集群中不同节点上的负载，并选择负载低于其他节点的节点。（优点：相比较上面的多个机器按照顺序轮流来处理用户请求，比如假设有个用户的请求需要的处理时

间比较长，夸张点说假如需要处理一分钟，那么负载均衡器在后面接收到别的用户的请求，就不会向刚刚那个正在处理长请求的服务器发送，而会选择空闲或者说是最少链接的机器来处理请求，缺点：session需要单独维护，下面第三种方法就来解决session的问题！）（**least connections**）

- 方法三：ip-hash Web负载平衡器的一个重要特性是会话粘性，这意味着客户端会话中的所有请求都指向同一服务器。（优点：保证了会话的粘性，只要用户的IP不变，集群中处理用户请求的机器就不会变，缺点：这是三种方法里面负载均衡效果最差的一种，如果hash函数选择的不好的话，最终的结果很可能是导致集群里面的经常是一台机器在处理请求，而另外的几个机器可能没有什么请求被分配；当然还有一个缺点就是如果用户改变了IP，比如上网途中开启了或者关闭了VPN或者代理服务器，导致用户ip改变，这时候session就可能丢失）（**ip-hash**）

## 3. session维护

由于我们使用负载均衡，第一次可能第一次登录转发给了A服务器，第二次订单可能转发给了B服务器，那我们知道服务器session是存储在内存里面的，B服务器怎么知道A服务器的session？

- 方法一：用ip-hash，保证对于同一个用户的访问，用同一台机器处理。
- 方法二：用一台单独的服务器，例如redis，存储用户的session，这样的话所有的机器只需要在集群里面的redis服务器里面拿就可以找到用户的session。此外，这种方法后端应用服务器重启之后，用户的session不会丢失。（这个bug在于单点故障，如果redis挂了麻烦就大了）
- 方法三（不推荐）：不同的后端服务器之间建立通讯，获取session方法。

## 4. Mysql集群

组成：一个类似GateWay的MySQL Router，负责转发请求。

数据库组成可能是一个主MySQL服务器，两个从MySQL服务器，主服务器负责读写操作，后面两个从服务器只能读，然后同步主服务器的内容。

MySQL Router，负责转发请求也会保证负载均衡，对于读的操作均衡分配（当然主服务器可能会少一点）对于写的操作分配给主服务器。

## 5. Nginx

Nginx是开源、高性能、高可靠的Web和反向代理服务器，而且支持热部署，几乎可以做到7\*24小时不间断运行，即使运行几个月也不需要重新启动，还能在不间断服务的情况下对软件版本进行热更新。性能是Nginx最重要的考量，其占用内存少、并发能力强、能支持高达5w个并发连接数，最重要的是，Nginx是免费的并可以商业化，配置使用也比较简单。

Nginx的最重要的几个使用场景：

- 静态资源服务，通过本地文件系统提供服务；
- 反向代理服务，延伸出包括缓存、负载均衡等；
- API 服务，OpenResty。

对于前端来说 Node.js 并不陌生，Nginx 和 Node.js 的很多理念类似，HTTP 服务器、事件驱动、异步非阻塞等，且 Nginx 的大部分功能使用 Node.js 也可以实现，但 Nginx 和 Node.js 并不冲突，都有自己擅长的领域。Nginx 擅长于底层服务器端资源的处理（静态资源处理转发、反向代理，负载均衡等），Node.js 更擅长上层具体业务逻辑的处理，两者可以完美组合。

## 5.1 反向代理

代理和反向代理：

- [Dropbox](#), [Netflix](#), [Wordpress.com](#), [FastMail.FM](#).



反向代理：(reverse proxy)，指的是**代理外网用户的请求到内部的指定的服务器，并将数据返回给用户的一种方式**；客户端不直接与后端服务器进行通信，而是与反向代理服务器进行通信，隐藏了后端服务器的 IP 地址。

反向代理的主要作用是提供负载均衡和高可用性。

- 负载均衡：Nginx可以将传入的请求分发给多个后端服务器，以平衡服务器的负载，提高系统性能和可靠性。
- 缓存功能：Nginx可以缓存静态文件或动态页面，减轻服务器的负载，提高响应速度。
- 动静分离：将动态生成的内容（如 PHP、Python、Node.js 等）和静态资源（如 HTML、CSS、JavaScript、图片、视频等）分别存放在不同的服务器或路径上。
- 多站点代理：Nginx可以代理多个域名或虚拟主机，将不同的请求转发到不同的后端服务器上，实现多个站点的共享端口。

# 第22讲 Cloud Computing

2023.11.30

倒不如这样  
我们回到拥抱的现场  
证明感情总是善良  
残忍的是人会成长  
——《不如这样》陈奕迅

## 1. overview

- cloud computing 云计算
- edge computing 边缘计算

## 2. cloud computing 云计算

### 2.1 基本概念

云计算：把所有东西变成互联网上面能够访问的服务，暴露出来。

云这个词至少代表了两个概念：

- 第一是它很远
- 第二是它是不透明的，比如你使用阿里云，你是否关心它在华东、华北跑你的虚拟机？你不关心。

云里面就强调了虚拟化，因为不同的人的需求是不同的，有人想要windows，有人想要linux，所以用虚拟机而不是物理机。

云的一些特征：

1. 定价灵活：短期租用、长期租用……
2. 弹性：当你觉得容量不够，可以较为轻松地实现扩容
3. 虚拟化：如果不虚拟化，共享的粒度就变成了物理机了，但是物理机本身一般就比较强了，很多人用不到，虚拟化就可以避免资源的浪费；另外就是前面提到的，用户使用环境差异非常大。

## 2.2 云计算的核心技术

在讲数据湖的时候，我们提到了说数据分为两类：批处理和流处理（batch processing & stream processing）。

下面这些玩意都是批处理的。Storm是流处理的。Spark是内存处理，也是批处理的。Spark Streaming是流处理的。

### 2.2.1 MapReduce

MapReduce就是一个典型的批处理的方式：每台机器现在就像原来的单机中的一个core一样（整个MapReduce其实有点点类似单机的Job Scheduling）。

我们写一个用户程序，比如我们要统计红楼梦里面，出现过多少个人物？假设我们有120回，太多了，我将其分成很多split（最适合是split大小是多少？就是你分布式文件系统的block大小！参考cse里面的gfs）；假设我们切出来100个split。

一个人统计太慢了，我找了3个人，每个人处理一个部分（当然分东西是动态的，master会做好调度）。如何处理呢？我会写好Map的逻辑，我读到了一次贾宝玉，我就产生一个键值对：{贾宝玉:1}。

master在这去做调度，那么大家一起做的时候，就会可能会出现最后一个情况，就是说这个worker前两个worker都做完了，最后这个worker处理第100块一直没处理完，那这时候我怎么办？我有一个逻辑，就是我说这样吧，上面这个同学既然做完了没事干，来我把100给你，你也去处理。因为我在判断这个100始终做不完，两种情况，一种是你挂了，一种是这个100相当难处理。反正我就让另外空闲的人也去做，只要有人做完就好了；假设你没死，那也不耽误事儿。

然后它们写完的东西就会发到intermediate files中间文件，这些东西存放在本地的硬盘上；然后reduce再去做处理——我最后就要把相同名字的键值对加起来。然后我们发现reduce phase还有两个worker：最后统计的时候，人物可能会很多，我们说：以A-N结尾的，放到某些文件里面，后面给到1号worker，剩余的放到另一个地方，后面给到2号worker。然后每个worker都会统计一下，最后把结果汇总起来，生成了output file，将所有人生生成的输出文件汇总，就得到了最后的输出文件。

注：

- 由于频繁读写硬盘，这里效率是偏低的；所以解决方案是写到内存里面，我们就需要一些很不错的数据结构来支持内存中的存储，这就是spark在做的事情。

总结：

- 如何理解“批”的概念呢？注意到，reduce开展工作之前，所有的统计工作是都必须完成的，否则就会出错。而不是说来一点处理一点，所有的数据在这里是成批流动的。流式处理就是数据来了我就处理，甚至他没处理完，新的数据就来了，源源不断的来。

- Map Reduce中的Map定义是：把输入映射成输出，每个机器不会管别的输入，只会管自己的输入部分，把输入结果产生中间结果。Reduce负责合并的部分，把所有的中间结果合并，得到最终的输出。

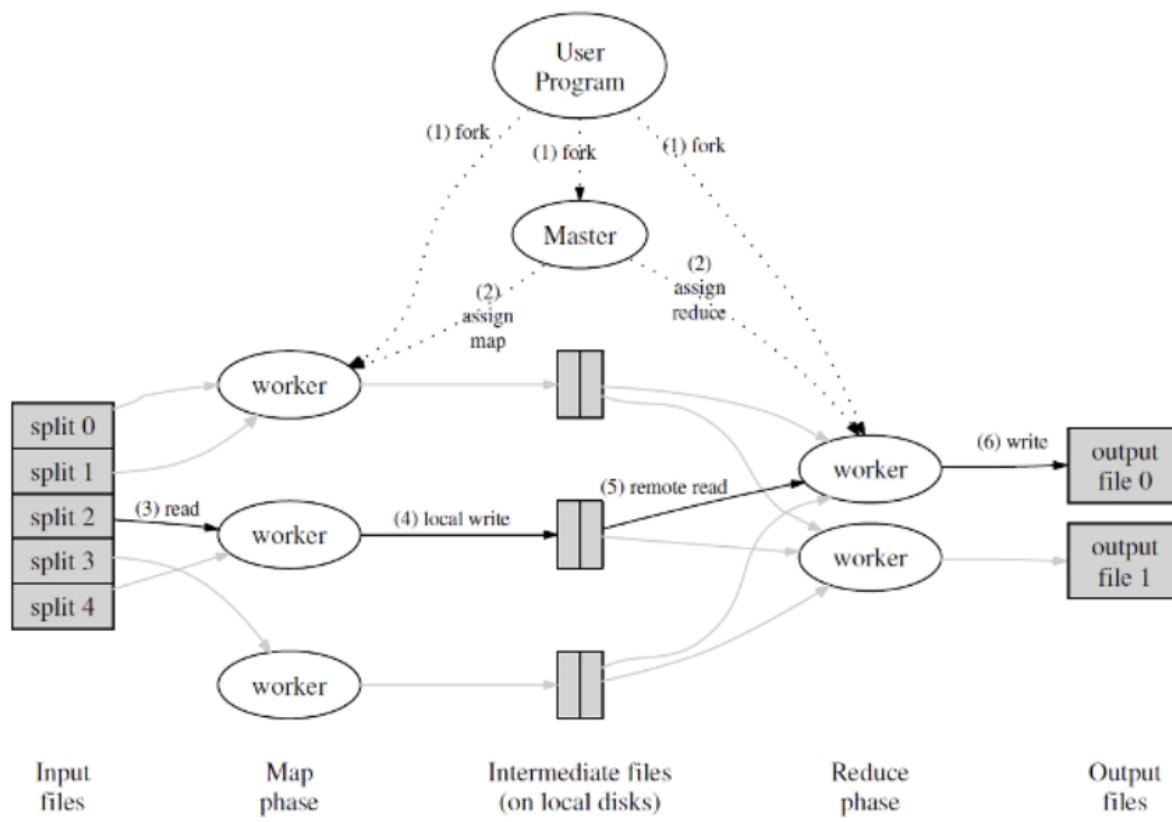
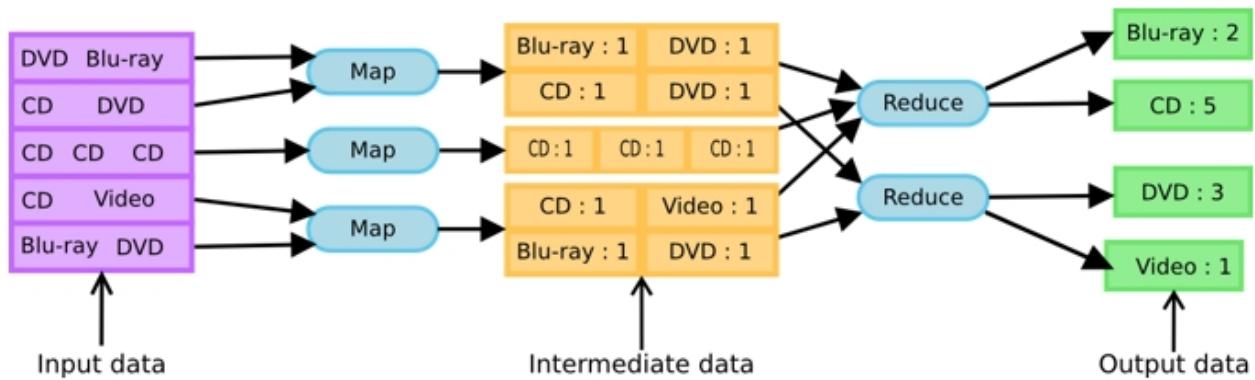


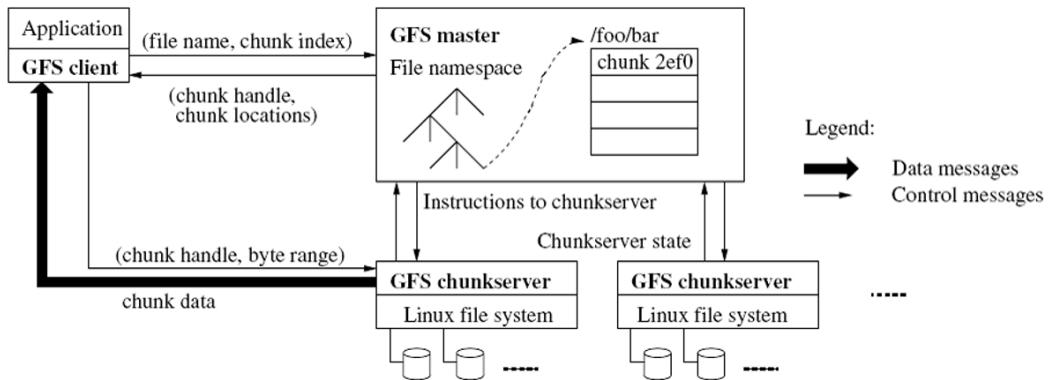
Figure 1: Execution overview



## 2.2.2 Distributed Google File System

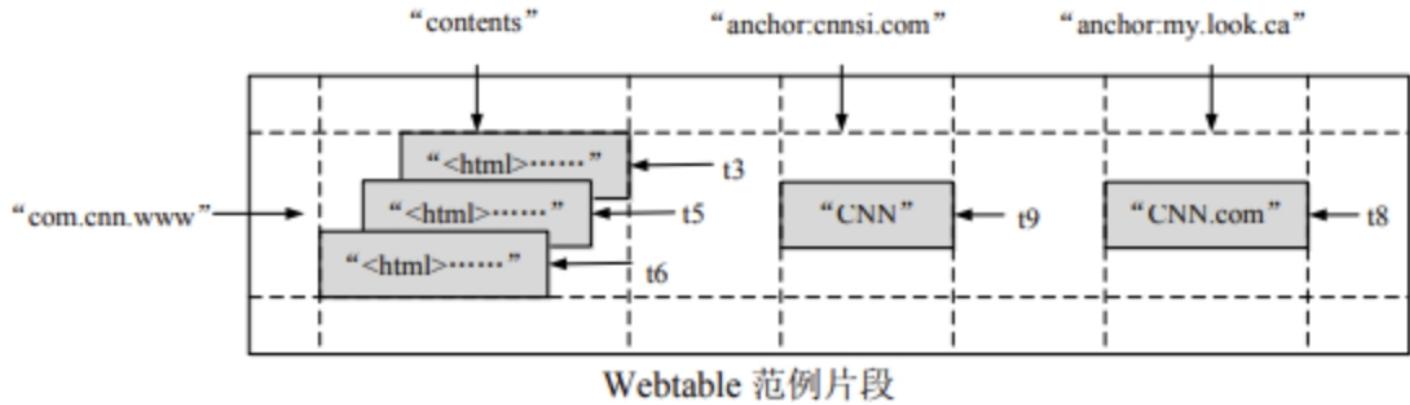
利用现有的文件系统实现文件管理，在上面再加一层，实现分布式。

- Google File System(GFS) to meet the rapidly growing demands of Google's data processing needs.



把文件切成很多的chunk。

### 2.2.3 Google BigTable: KV数据库的鼻祖



1. 首先和MongoDB一样，不存在什么乱七八糟的外键关联。
2. Column Family 列族：像二级分类一样，比如你有十个字段，我们分为3+3+4，3，3，4就分别成为列族。后面结构也可以比较自由地变化，没有那么严格的schema。
3. TimeStamp：基于时间戳的数据存储。上图中展示出了一个立体的结构。

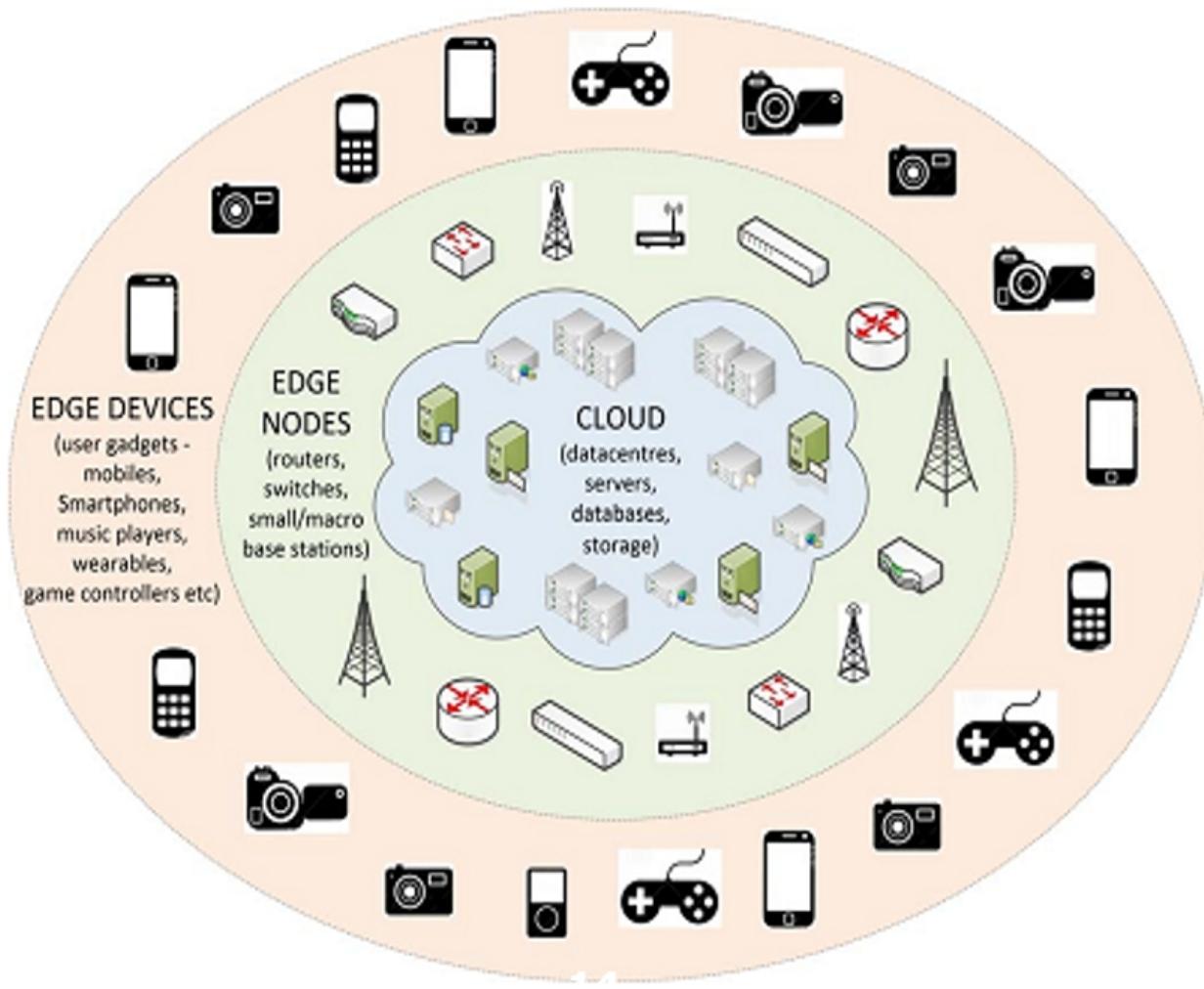
总结：数据量非常大、schema没那么严格、希望存储多个版本。

### 2.2.4 Hadoop

Hadoop就是前面那么一大坨玩意的开源实现。

## 3. edge computing 边缘计算

我们现在的设备都是在网络的边缘。



定义：在网络的边缘，临近数据源的地方，来进行数据的处理，是一种优化云计算系统的方式。

主要考虑到的就是：

1. 网络连接可能不是持续稳定的，你的数据未必都能发到云里面去，到达核心的云服务器
2. 通信的带宽可能是受限的，如果大家同时发送，可能很快都占满了

计算迁移：把计算从云端迁移到边缘端。 (cloud offloading)

# 第23讲 GraphQL

2023.12.01

总要在雨天 逃避某段从前  
但雨点偏偏促使这样遇见  
总要在雨天 人便挂念从前  
在痛哭告别后从没再见  
——《分手总要在雨天》张学友

前面我们说的客户端和服务端的交互方式：

1. 大二的时候我们只知道发http请求：get, post, put, delete，充分利用这些方法，产生restful的交互性，实现增删改查（但是没有复杂的交互逻辑，我如果想要一个很复杂的逻辑，就不好说了）。
2. 后来我们又看到了websocket。

recall：什么是RESTful API？

RESTful API (Representational State Transfer API) 是一种设计风格，用于构建网络服务和应用程序之间的通信。它基于 REST 架构原则，这些原则提供了一组规范和约定，使得系统能够在分布式环境中更加灵活、可扩展和易于维护。

以下是 RESTful API 的一些关键特征和原则：

1. **资源 (Resources)**：在 REST 中，所有的事物都被视为资源。每个资源都有一个唯一的标识符（通常是 URI）。
2. **表现层 (Representation)**：资源的表现层定义了如何表示和传输资源的状态。通常，这可以是 JSON、XML 或其他格式。
3. **状态无关 (Stateless)**：每个请求从客户端到服务器都必须包含所有信息，服务器不应存储客户端的状态。这使得系统更容易扩展，因为不依赖于特定的会话状态。
4. **统一接口 (Uniform Interface)**：统一接口是 RESTful API 的核心。它包括以下几个原则：
  - **标识资源**：使用唯一的 URI 标识每个资源。
  - **通过资源操作**：使用标准化的 HTTP 方法 (GET、POST、PUT、DELETE) 对资源执行操作。
  - **自描述消息**：通过资源的表现层传递的消息应该包含足够的信息，使客户端能够理解并处理资源。
5. **无状态通信 (Stateless Communication)**：每个请求从客户端到服务器都必须包含所有必要的信息，服务器不应存储客户端的状态。

RESTful API 被广泛用于构建 Web 服务、移动应用后端和其他分布式系统，因为它提供了一

种简单而灵活的方式来设计和管理 API。在使用 RESTful API 时，客户端可以通过简单的 HTTP 请求与服务器进行通信，并使用资源标识符来操作和获取数据。

GraphQL本身是一种查询语言，我们这里运用的是Http协议，request body里面的内容是GraphQL的查询语句。服务器端处理之后，返回给你一个带着查询结果的response。

查询和返回的结果全部都是json格式的数据。

## 区别：

### 1. 查询语言和端点数量：

- **GraphQL**: 使用单一端点，客户端可以精确指定所需数据，而无需多次请求不同的端点。GraphQL 使用一种强大的查询语言，允许客户端请求自己需要的字段。
- **RESTful API**: 通常使用多个端点，每个端点代表一个资源或一组相关资源。客户端需要根据 RESTful API 的设计来请求不同的端点，每个端点返回预定义的数据结构。

### 2. 数据获取方式：

- **GraphQL**: 客户端可以按需获取需要的数据，而不会获取过多或过少的数据。客户端决定所需数据的结构。
- **RESTful API**: 服务器端决定返回的数据结构，客户端需要根据服务器端提供的资源路径获取数据。

### 3. 版本管理：

- **GraphQL**: 由于客户端可以精确指定所需数据，因此不需要对 API 版本进行严格管理。新增或修改字段不会影响现有客户端。
- **RESTful API**: 如果对资源进行了修改，可能需要增加 API 版本或通过其他方式进行版本管理，以确保对现有客户端的兼容性。

### 4. 实时数据：

- **GraphQL**: 支持实时数据的推送，可以通过订阅机制实现。
- **RESTful API**: 通常需要通过轮询或使用 WebSocket 等机制来实现实时性。

## 联系：

1. **HTTP 协议**: GraphQL 和 RESTful API 都基于 HTTP 协议，可以使用相同的底层传输机制。
2. **资源**: 两者都处理资源，但 RESTful API 以资源为中心，每个资源有一个唯一的 URI，而 GraphQL 允许客户端根据需要组合资源。
3. **用途**: 根据具体的需求和团队偏好，选择 GraphQL 或 RESTful API。RESTful API 在很多项目中仍然是非常流行和有效的选择，而 GraphQL 提供了更灵活的数据获取方式。

总体来说，选择 GraphQL 还是 RESTful API 取决于项目的特定需求和团队的背景，每种架构风格都有其适用的场景。

# 第24讲 Docker

2023.12.04

早知解散后 各自有际遇作导游  
奇就奇在 接受了 各自有路走  
却没人像你让我 眼泪背着流  
严重似情侣 讲分手  
——《最佳损友》陈奕迅

## 1. Docker

### 1.1 Docker 是什么？

Containerization（容器化）是一种虚拟化技术，**它将应用程序及其依赖项打包到容器中，以便可以在任何环境中运行**（也就是为你的应用提供了一个环境）。容器化软件的一个重要好处是，它们与其运行的环境隔离，因此可以在任何地方运行。

Docker 是一种容器化技术，它使用操作系统级虚拟化来提供独立的软件容器。与虚拟机不同，容器不需要虚拟化硬件，因为它们直接在操作系统内核上运行。这使得容器更轻量级、更易于部署和更快速。

跑 100 个Docker，那他们的机器每一个看起来都像是个完整的 Linux 的文件系统，那实际上你知道文件系统在这个宿主机上它也有一个文件系统，它就是用 **c group 和 name space 这种方式去把你这些文件系统一个隔开。**

在硬件上面是做了一个资源的分片，容器和容器之间的资源是完全隔离的，互相之间就不会产生干扰。

### 1.2 Image

Docker 镜像是一个只读的模板，它是用来创建 Docker 容器的。Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。

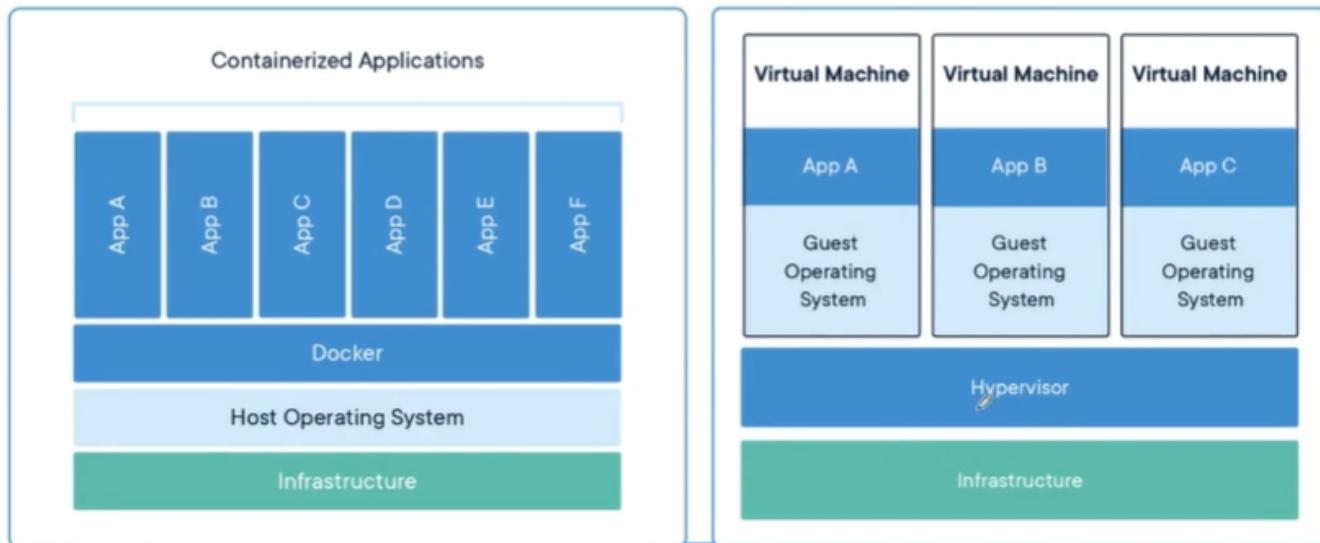
镜像不包含任何动态数据，其内容在构建之后也不会被改变。

**layer 的概念：**镜像是由多层文件系统联合组成，每一层镜像都是在前一层镜像的基础上进行的修改，每一层镜像都可以看作是一个 Docker 镜像。

- 可重用性和分发：由于层的存在，可以轻松地重用和共享部分镜像。如果两个镜像共享相同的层，它们只需要存储这些层的一个副本，而不是两个。
- 容器之间的共享层：当多个容器共享相同的基础镜像时，它们会共享那些相同的层，这节省了存储空间。

## 1.3 virtual machine vs docker

虚拟机：自身是一个进程，但是包含了一个完整的os；比如我跑了三个linux虚拟机，我们觉得这太浪费了（当然如果跑了一个windows，一个mac os，一个linux那当我没说）。如果大家都是linux的，我们就把你们各自不同的特有的东西给你放到你的自己的这个 image 里，就你的运行环境，其他凡是依赖于操作系统的，那么我就通过 Docker 去给你转入。

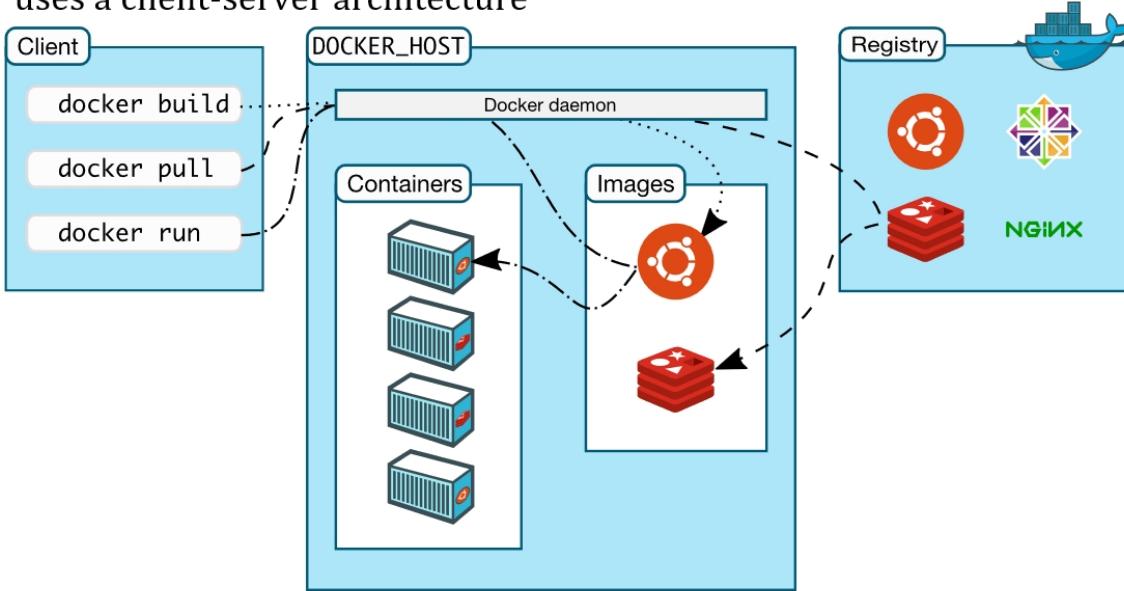


左图中，最上层的app全都是对linux系统产生调用，docker负责把这些linux系统调用转为windows的系统调用。在这种情况下，因为所有人都用linux，我们就不用每个人都装一个完整的操作系统。

更具备可移植性、更高效：我迁移底层系统，windows换为mac os，我只需要把docker这一层换掉就行了。

## 2. Docker Architecture

- Docker uses a client-server architecture



image在registry中， registry是一个集中的存储和分发image的地方，可以理解为一个image的仓库。 registry中的image可以通过pull命令下载到本地，也可以通过push命令上传到registry中。

build是说我们可以通过Dockerfile来build一个image，这个image可以是我们自己的，也可以是别人的，比如我们可以build一个nginx的image，然后把我们自己的网站放到这个image中，这样我们就可以通过这个image来部署我们的网站了。

run是说我们可以通过image来run一个container，这个container就是一个运行的实例，我们可以通过这个container来访问我们的网站。

image又是分层的，我们下载的时候也未必是下载完整的image，是下你本地没有的。

### what is kubernetes?

所谓 K8S 就是你要去观察你所有的物理机上面的一些docker的运行状态，然后去调度它。比如说我要去做负载均衡，这个物理机比较多，上面跑的容器比较多，我就要迁移一些到另外一个地方，但是整个这件事情对用户要屏蔽。

## 3. Dockerfile

要实现容器化部署 (build an image)，你需要写一个dockerfile。

```
# syntax=docker/dockerfile:1

FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

注：实际上在写spring boot的时候，你其实是不需要写dockerfile的。

第一行是说我写的这个应用本身是不能跑的，大家可以看到它这个里面就是有一堆的 js 的脚本。那我们之前就知道了，你 js 脚本一定要在某一个 Web 服务器里面，就是一个 HTTP 的 Server 里面才能跑，就相当于我们当时在开发这个前端的时候，实际上你跑了一个 HTTP 前端的Server，它在 3000 这个端口，然后它跑起来之后你的应用才能够被服务。那所以你这个 Docker 上跑起来必须要用到 node js，他用的是 18 这个版本。

所以他就说在这个容器里要跑一下 NODE 去运行这个 source 这个目录，就我们把它下载下来，这个 source，这个目录里面那个 index.js，这是我们看到的这个js。然后跑起来之后你要暴露在 3000 这个端口上。

build的含义就是把你的代码打包成一个镜像，然后这个镜像就可以被部署到任何一个地方。注意科学上网。

```
docker build -t getting-started .
```

image打好了，我要改里面的内容怎么办？重新打包、编译。所以有没有简便的方法？就是volume。

我们先来再看另外一个问题，就是持久化问题。

1. When a container runs, it uses the various layers from an image for its filesystem.
2. Each container also gets its own “scratch space” to create/update/remove files.
3. Any changes won’t be seen in another container, even if they are using the same image.

那一个容器我起来以后，我写了一些东西，以后我把这个容器销毁了，我希望这些修改保存到image中去了，应该如何做到这种持久化？

## 4. Volume 容器的卷

这个说穿了这次cse里面我们已经用过了，就是把主机上的一个目录映射到docker里面去。

这时候就要用到容器的卷，那么容器的卷分两种，一种是由 Docker 统一管理的卷，就是 name 的 value。

还有一种是你可以自己去指定的一个卷，就它脱离 Docker 的管理，我指定它在硬盘的某一个位置。

volume就是在你的这个机器启动以后，它可以连接到一个特定的目录里头。我把硬盘上的一个目录直接连到你这个容器里面的一个文件系统的某一个位置上，就好像是我给你外挂了一个这样的一个volume一样。所以将来你在容器里去操作这个特殊的路径的时候，实际上就在操作我在宿主机上的那个文件系统里面的某一个文件，那个文件在存的时候，在宿主机上来说就是一个文件，这个文件放到这个容器里就变成了它的一个外挂卷。

这样的话就相当于我一个 image 起来的时候，我要到机器上去捞这个外挂的卷，捞出来之后挂到我们的这个系统里，那所以 image 不动，image 里对文件系统的写字、写入这些动作我全部都存在volume 里，volume 是存在宿主机上的，所以即使你这个容器没有了，这个卷本身还在，下次你根据这个 image 重启一个容器，再把这个卷挂回来就可以了。

## 4.1 Named volumes

volume 实际上就是一个存放数据的一个桶，它的物理位置是存在硬盘上的，这个物理位置是你不用管的，是 Docker 它会去维护的。你只需要记住它的 name 是啥就行了。

然后我启动这个 container 的时候挂一个卷。每一次新启动这个容器的时候，你只要指定这个 volume 到，就能帮你把它找到挂进去。

说穿了就是我只给了它一个名字，其余它具体存在哪里，都不用你管，docker 来负责。

## 4.2 Bind mounts

这次cse的chfs就是这么干的。也就是我们可以自己决定这个卷在宿主机上的哪个位置。

所以volume的意义就是，我经常需要去改那个代码，我就不需要每次去撸一个新的容器出来，我的项目代码挂载进去，每次我改了代码，直接就能反映出来。这就是所谓的“开发用的容器”。

# 5. Multi-container apps 多容器应用

我经常可能跑起来需要两个容器。我们在面向对象编程的时候，就知道职责要单一，那每一个容器尽量它的职责也单一，这样有助于万一哪一个崩了，另外一个还好。就是我们的应用在一个容器里面，我们的 MySQL 应该在另外一个容器里面跑。

先看第一点，就是这两个容器怎么通信？就是要靠网络，他们必须在同一个网络里。所以就两步：创建网络、运行。

但是这么跑还是两个容器分别在跑；有没有一步到位的方法？

## 5.1 Docker Compose 一次性启动多个容器

说白了就是一个脚本，把你的这些容器的启动命令都写在这个脚本里面，然后你一次性启动这个脚本，它就会把这些容器都启动起来。

# 第25讲 Hadoop

2023.12.06

和你也许不会再拥抱 待你我都苍老  
散半里的步 前尘就似轻于鸿毛  
提及心底苦恼 如像自言自语说他人是非 多么好  
从来未爱你 绵绵  
可惜我爱怀念 尤其是代我伤心的唱片  
——《绵绵》 陈奕迅

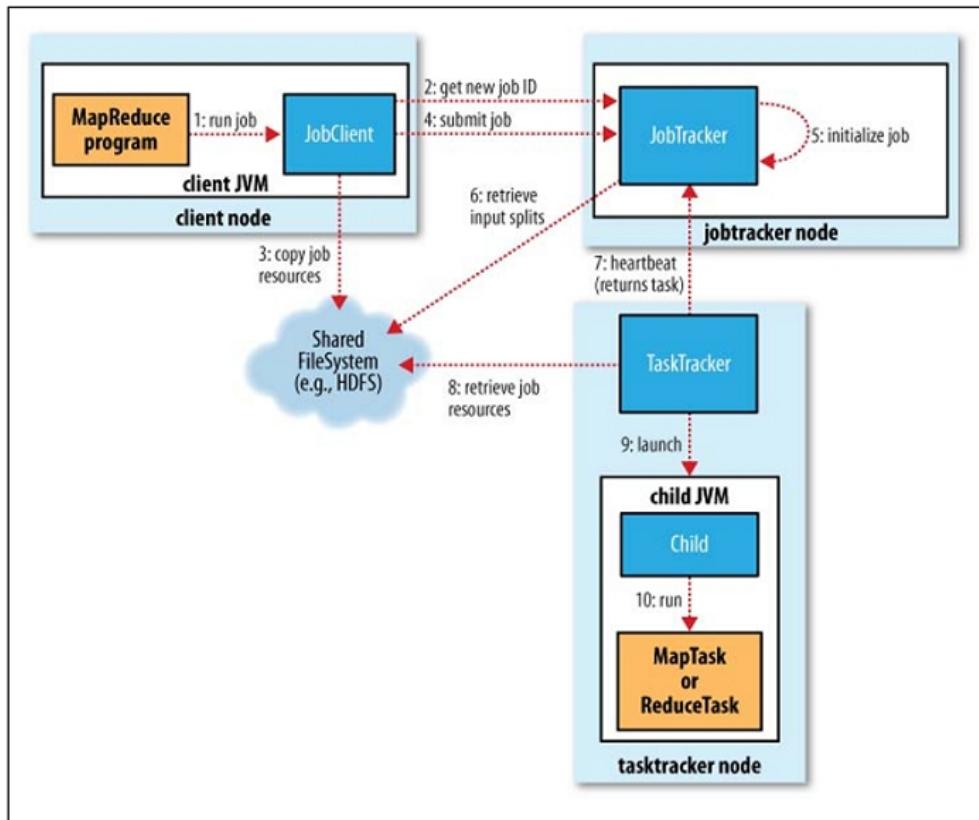
## 1. Contents

- Hadoop: Basic Concepts
  - Hadoop Common: 内核
  - HDFS: 分布式文件系统
  - YARN: 资源管理
  - MapReduce: 分布式计算；MapReduce并行处理的时候会出现一个问题，就是单一故障节点，可能导致整个系统崩了，或者成为bottleneck，所以要有YARN。

针对大数据批处理请求，设计基于MapReduce/YARN的并行处理方案。

## 2. 从MapReduce到YARN

### 2.1 Deeper into MapReduce



我们有一堆的能够执行任务的节点——task Tracker NODE，构成了一个集群。

- client NODE是你的客户端；比如你写的统计红楼梦的代码就是这个黄颜色的（MapReduce Program）。
- 程序在跑的时候，JobClient会从JobTracker那里得到一个JobID，然后JobClient指定输入输出目录（和HDFS交互），然后提交这个作业。
- JobTracker就开始做调度；到文件系统把你的输入文件拿出来（已经被切成了split），就启动若干个任务节点来操作；每个任务节点都有一个TaskTracker，会去文件系统获得任务所需要的资源。然后我就在我这个节点上创建一个新的JVM，是它的子进程，在这个子进程里就执行黄色的MapReduce Program。
- 执行的过程中，TaskTracker不断发心跳给JobTracker。

问题：

- 为什么需要TaskTracker？因为一台机器上可能执行了多个MapReduce Program。
- JobTracker的职责是什么？
  - 第一个分配资源，就是我怎么会知道要让哪一些节点去执行这个 task 呢？
  - 第二个，他在跟踪这个 tracker 的执行情况。

- 那你说是不是他就很重量级，他要管的事情非常多，而且一旦他崩溃，整个系统崩溃。你再重启，你会发现所有的job全没了，这些 tracker 都不知道向谁汇报，这不是我们希望得到的。
- 所以我们就有了YARN。它就是希望将这两个职责分开。

## Reduce的三个阶段

### Shuffle & Sort



- **Shuffle**

- Input to the Reducer is the sorted output of the mappers.
- In this phase the framework fetches the **relevant partition** of the output of all the mappers, via **HTTP**.

- **Sort**

- The framework groups Reducer inputs by **keys** (since different mappers may have output the same key) in this stage.
- The shuffle and sort phases occur **simultaneously**; while map-outputs are being fetched they are merged.

### Secondary Sort & Reduce



- **Secondary Sort**

- If equivalence rules for grouping the intermediate keys are required to be different from those for grouping keys before reduction,
- then one may specify a **Comparator** via [`Job.setSortComparatorClass\(Class\)`](#).
- Since [`Job.setGroupingComparatorClass\(Class\)`](#) can be used to control how intermediate keys are grouped, these can be used in conjunction to simulate **secondary sort on values**.

- **Reduce**

- In this phase the **reduce(WritableComparable, Iterable<Writable>, Context)** method is called for each **<key, (list of values)>** pair in the grouped inputs.
- The output of the reduce task is typically written to the [\*\*FileSystem\*\*](#) via [`Context.write\(WritableComparable, Writable\)`](#).
- The output of the Reducer is **not sorted**.

## Reducer的数目？

- $number.of.nodes * number.of.max.containers.per.node * 0.95 or 1.75$ , 注意不是范围，只有两个值

- 如果是0.95，比如说有 5 台机器，每个上面可以跑 5 个容器，那  $5 * 5$  是 25。如果我乘 0.95，比如说是 24，那我起 24 个 reduce，因为你能跑 25 个容器，所有的 reduce 就一起跑起来。那如果乘 1.75，就是说我可能会有 35 个这样的 reduce，那我在 25 上不能同时跑，我先跑 25 个，中间有跑完的，就让他从这里面继续拿。就像我刚才说的，三个同学处理 12 分，怎么处理每一个先跑一个，跑完告诉我再给他分一个。
- 那你说这两个有什么优点和缺点？先看上面的，这个是说你的 reduce 的数量一定小于你运行的这个能力，那所有的 reduce 就同时并发的就开始跑。缺点，万一某一个 reduce 特别慢，就是因为数据的原因，比如它那个集合里面的值特别多，那于是你就变成了有 0.05% 的容器在那闲着，在有一个巨忙的。然后底下这个就相当于我把这个任务切碎一点，我先跑 25 个，大家都很少有一些在那等，但是因为每一个任务都很小，所以他们很快就执行完了，在这里面再去捞，速度会快一点，那所以你就会看到上面那个负载均衡不好，就有的可能很忙，有的就闲着，但是它任务调度就会变得简单，因为我一次性的就把所有的 reduce 就在一起跑，底下这个负载一定是均衡的，大家都挺忙的，同时忙，大家一起玩完，就结束。但是需要做任务的来回的调度，带来额外的开销。
- 那到底选哪个？我觉得其实就是你的一个权衡，你觉得这些任务本身已经很平均了，那我们就用 0.95 就可以，如果你认为确实存在比较大的倾斜，有的这个及格里面可能就 10 个，有的有 10K，那你还是用底下这种方式可能更好一点。

## 2.2 YARN

资源管理器还是只有一个，是全局的；但是那个 Job Tracker 分开一个任务/一个应用一个。

# 第26讲 Spark

我把幸福看得太简单了点  
你有多用心我却没有发觉  
谢谢你的照顾你的离开  
让我对爱有更多了解  
现在才说也许太晚了一点  
失去过的人会更珍惜一点  
——《谢谢》 陈奕迅

参考资料：

1. <https://www.hadoopdoc.com/spark/spark-intro>
2. [https://www.cntofu.com/book/198/notes/Spark\\_RDD.md](https://www.cntofu.com/book/198/notes/Spark_RDD.md)

## 1. Spark基本概念

### 1.1 简介

- 相对于 MapReduce 的批处理计算，Spark 可以带来上百倍的性能提升，因此它成为继 MapReduce 之后，最为广泛使用的分布式计算框架。
- 和Hadoop的差异是，是用内存做计算的，可以构建于Hadoop之上。
- 能做数据科学、机器学习等任务
- 可以做批处理，也可以做流处理（数据是源源不断的）
- 可以在单点做，也可以在集群做。

### 再说批和流

比如这个教室里，放置一个传感器sensor，然后它就可以不断的去往服务器server去发数据。那这个数据就是只要这个传感器不坏，它就一直在发，是为“流”。这是第一点。

第二点，他发出去，你去处理。例如，他不断过来发的是这个温度，那这个温度过来之后，服务器在这一端他就要去判断这个温度，来决定这个空调它到底要不要开。

但是你可以看到它这个接口走什么，如果你走http就有问题了。http是什么呢？是请求-响应模式的。那是不是意味着我在这一边发过来一个数据，你就必须得响应？你得告诉我开还是不开？然后在响应没有回来之前，我可能这边是不能发了，这个设计显然就不对。

HTTP 这个协议你要不断的发，你不要管我处理没处理过来。比如说你一秒钟发一次，连着发了 10 秒就有 10 个数据了，也许我服务器端呢？还没有那么强的这个处理能力，当你发第 11 个数据

之后，第一个数据才处理完。

如果是这样的话，就意味着你在这个客户端就像这个sensor一样，它实际上在发这个数据的时候压根就没有得到响应的情况下，他就发了第二个数据，这是你看和批处理存在一个很大的差异，那你说那怎么办？他发快了，处理的慢怎么办？那我们在讲消息中间件的时候就在处理这个事——kafka！就是你要通过一个消息队列把它给收下来，然后你到后面慢慢去处理，你也可以来了 10 个，我忽略掉其中 9 个，每 10 个我处理一个，这都是我服务器端的它的一个逻辑。但是我就不能是请求响应式了，不能说你发 10 个数据过来，我就要产生 10 个响应，我可能就产生一个响应。而中间做这件事情必须有一个像kafka这样的东西，在中间做一个像过渡一样的东西，这就我们看到流式数据和批式数据的一个非常大的一个差异，除了这个数据它源源不断，而且它是来一点处理一点；和批次处理，说我一批处理完它能进入下一个阶段不一样之外，更重要的是他没有说一定是个请求响应模式，这是很大的一个区别。

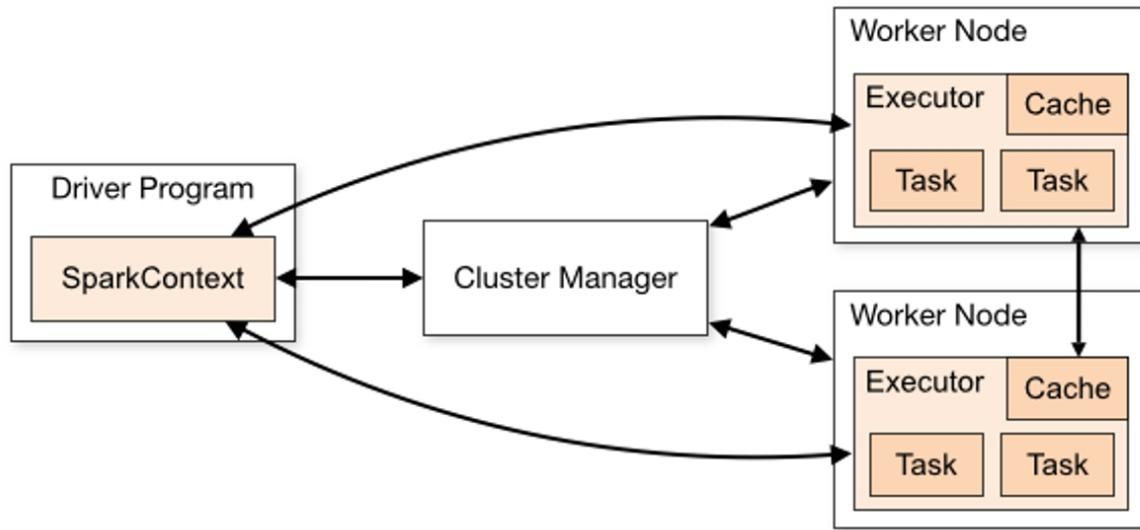
## 1.2 内存！

Hadoop做mapreduce是利用分布式文件系统，磁盘来做中间结果的汇聚，很多的磁盘io操作，所以慢。你读了一个block进来，你把结果spill到硬盘上 (io+1) ；所有spills全部出来之后，要做shuffle, sort, 放到不同的地方 (io+1) ，然后reduce去读，处理完再写 (io+2) 。

Spark的逻辑是，基本上还是在HDFS来做处理，但是我一旦从分布式文件系统里面把数据读出来，我就做内存计算了。内存不够怎么办？本来应该是要换到硬盘上，所以我用cache，让它在内存里占住，不让它被换出swap到硬盘上。

Spark 里面告诉你 cache 这件事情，就跟你在 cpp 里面你定义那个变量和 register 一样，你定义一个变量是register，它真的是在寄存器里面吗？取决于编译器他认不认对不对？你定义 cpp 里面你定义一个变量说它是register，那编译器完全有可能会把它放到内存里，它不是在寄存器里，它要根据实际情况去判断的，那就跟这道理类似的，开始只是你一种表达，你一个愿望， Spark 到底做不做这件事？不一定，那至少你他有足够的内存空间的情况下，会优先考虑你这个愿望。

## 2. Spark Components



Term 术语	Meaning 含义
Application	Spark 应用程序，由集群上的一个 Driver 节点和多个 Executor 节点组成。
Driver program	主应用程序，该进程运行应用的 main() 方法并且创建 SparkContext
Cluster manager	集群资源管理器，Spark 可以运行在多种集群管理器上，包括 Hadoop YARN、Apache Mesos、Standalone。
Worker node	执行计算任务的工作节点
Executor	位于工作节点上的应用进程， 负责执行计算任务并且将输出数据保存到内存或者磁盘中
Task	被发送到 Executor 中的工作单元
Job	多个并行执行的Task，合起来就叫做一个Job

执行过程：

- 用户程序创建 `SparkContext` 后，它会连接到集群资源管理器，集群资源管理器会为用户程序分配计算资源，并启动 `Executor`；
  - `SparkContext` 是 Spark 应用程序执行的入口，任何 Spark 应用程序最重要的一个步骤就是生成 `SparkContext` 对象。 `SparkContext` 允许 Spark 应用程序通过资源管理器访问 Spark 集群。
- Driver 将计算程序划分为不同的执行阶段和多个 Task，之后将 Task 发送给 `Executor`；

- Executor 负责执行 Task，并将执行状态汇报给 Driver，同时也会将当前节点资源的使用情况汇报给集群资源管理器。

## 3. Spark RDD(Resilient Distributed Dataset) 弹性分布式数据集

Spark的基础数据结构，RDD具有**不可修改的特性**（就不会涉及到复杂的内存操作，比如删除，所有后面的元素要往前移动），并且会在集群的不同节点运行计算。Spark RDD里面的数据集会被逻辑分成若干个分区，这些分区是分布在集群的不同节点的，基于这样的特性，RDD才能在集群不同节点并行计算。

- Resilient (弹性)：RDD之间会形成有向无环图 (DAG)，如果RDD丢失了或者失效了，可以从父RDD重新计算得到。即容错性。
- Distributed (分布式)：RDD的数据是以逻辑分区的形式分布在集群的不同节点的。
- Dataset (数据集)：即RDD存储的数据记录，可以从外部数据生成RDD，例如Json文件，CSV文件，文本文件，数据库等。

### 3.1 RDD操作

RDD操作分为两类：**transformation**和**action**。

- transformation：对RDD进行转换，返回一个新的RDD，但是不会立即执行，只有遇到action操作时才会执行。举例：map就是典型的transformation操作。
- action：对RDD进行计算，返回一个结果或者将结果保存到外部存储系统中。举例：reduce就是典型的action操作。

spark中，**所有的transformation操作都是lazy的**，也就是说，只有当action操作发生时，才会触发transformation操作的执行。

why？我们刚才说 RDD 是不能修改的，是只读的。transformation 的结果是一个新的RDD，那就是说你在执行这个 transformation 之后，一定会生成新的RDD，它就会占内存。但是这个 RDD 什么时候被会被用到？不知道，因为你的 driver 最后一定是想得到一个值，那既然不知道，你为什么先让他把内存先占着，你应该一直到最后他要做一个action，要得到一个值的时候，这时候才要迫不得已往前去推，他是经历了哪些 transformation 得到了，再把前面的 transformation 都做一遍，这样的话我可以最节省的去使用内存。所以他就想了这么一个办法。

这里的方法还是构建有向无环图DAG。

## 3.2 分区 Partition

一旦外部文件读进来，或者说像你在内存里一个数组给你做了一个这样的一个并行化，得到了一个 RDD 之后，紧跟着他就要去做分区，就是他要把你这个 data set 要切成若干块放到不同的机器上去。因为如果只在一台机器上就不能并行了；他要看你 worker node 的数量来做这个切分，然后 RDD 已经切分好了，现在可以在上面就可以去做这个相应的操作了。

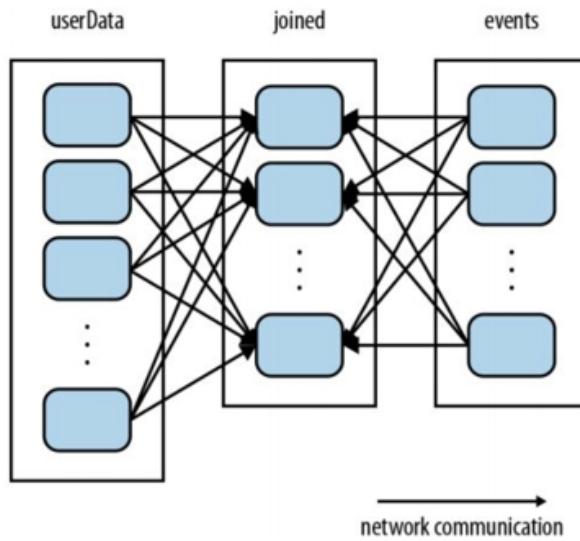
分区好处？

- 假设程序在内存中持有一个非常大的用户信息表 (UserID, UserInfo)，其中 UserInfo 包含用户订阅的主题列表
- 有一个很小的表，记录过去5分钟里在网页上点击过链接的事件，键值对为 (UserID, LinkInfo)
- 程序周期性地将这两个表合并、查询

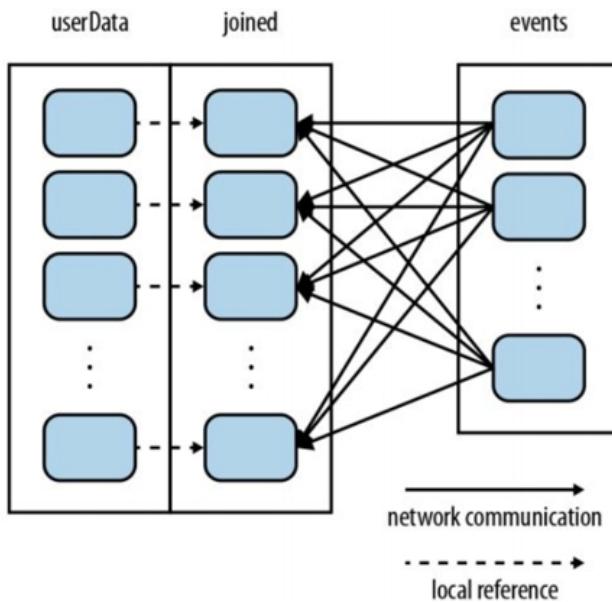
中间生成的joined是分好块的，比如a-g一块， h-m一块， n-z一块。

如果没有partition，你其实是不知道数据集中的主键是如何分布的；一块在一台机器上面存，我们这儿是分布式存储。如果我们知道A的userData只存在某台机器上面，A的events只存在某台机器上，我们只要join这两部分数据就行了。但是实际上我们现在做不到这一点。

这里userData是几乎不发生变化的，events是及时更新的数据；但是我们还是需要周期性地做hash和shuffle userData。这里网络通信开销就很大，有点像机器学习中的全连接层。



所以我们提出来，加载用户信息的时候，把用户分解为100块。event是按照时间顺序不断增加的，这是乱的没办法，但我们userData已经分好了。



### 3.3 Dependency 依赖

`joined`这些RDD是依赖于`userData`和`events`的。

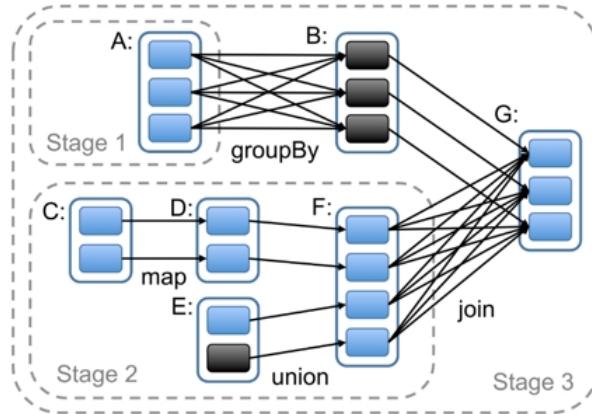
- Narrow Dependency: 每个父RDD的partition最多被子RDD的一个partition使用，这种依赖关系称为窄依赖。窄依赖的RDD可以并行计算，不需要shuffle。
- Wide Dependency: 每个父RDD的partition可能被子RDD的多个partition使用，这种依赖关系称为宽依赖。宽依赖的RDD需要shuffle操作。

#### 窄依赖的好处

- 窄依赖只需计算丢失RDD的父分区，不同节点间可以并行计算，能更有效地进行节点的恢复。
- 宽依赖中，重算的子RDD分区往往来源自多个父RDD分区，其中只有一部分数据用于恢复，造成了不必要的冗余，甚至需要整体重新计算。
- 宽依赖做完了一般需要缓存，因为比较复杂，好不容易做完了，存下来。

#### 概念：Stage

Spark怎样划分任务阶段 (stage) 的例子：实线方框表示RDD，实心矩形表示分区（黑色表示该分区被缓存）。要在RDD G上执行一个动作，调度器根据宽依赖创建一组stage，并在每个stage内部将具有窄依赖的转换流水线化 (pipeline)。本例不用再执行stage 1，因为B已经存在于缓存中了，所以只需要运行stage2和3。



- 每个阶段stage内部尽可能多地包含一组具有窄依赖关系的transformations操作；以便将它们流水线并行化（pipeline）（到最后一次性并行地做，不用占据内存空间，效率高）
- 边界有两种情况：一是宽依赖上的Shuffle操作；二是已缓存分区（碰到边界，这个stage就要真正执行了，执行完了这个stage结束）

这样的好处就是只有在 f 要被使用的那一刻，而且需要被宽依赖的使用的那一刻，我们才去创建f。在此之前无论是cdef四个当中哪一个都在内存里是不存在，那所以你就省空间。

# 第27讲 Storm

2023.12.15

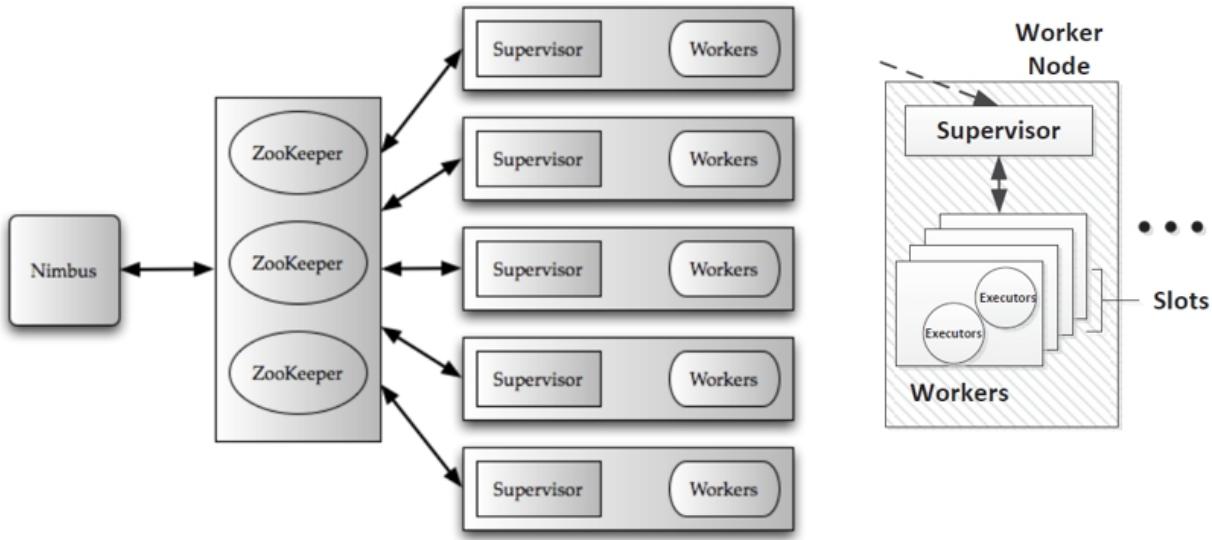
短暂的相遇 却念念不忘  
多少恍惚的的时候 仿佛看见你在人海川流  
隐约中你已浮现  
一转眼又不见  
当天边那颗星出现 你可知我又开始想念  
有多少爱恋 今生无处安放  
冥冥中什么已改变  
月光如春风拂面  
——《假如爱有天意》 李健

参考资料：

1. [https://www.w3schools.cn/apache\\_storm/apache\\_storm\\_introduction.html](https://www.w3schools.cn/apache_storm/apache_storm_introduction.html)

## 1. Storm 简介

做流数据处理。推特发明的。比如说如果有人发一张照片——恶意地发一组 20 MB的高清照片上去，你如果不做处理，页面的加载就非常慢。推特做的事情，比如说对推文理照片，它叫分成不同尺寸，也就压缩成200K，然后当你点它看原图的时候，它再就把 20 MB拿出来。这就是两个动作，它不用在一开始加载整个推文的时候就把这个数据加载，那所以他要做类似很多这样的一个处理。而这些推文都是源源不断的在不断的发，是为“流处理”。

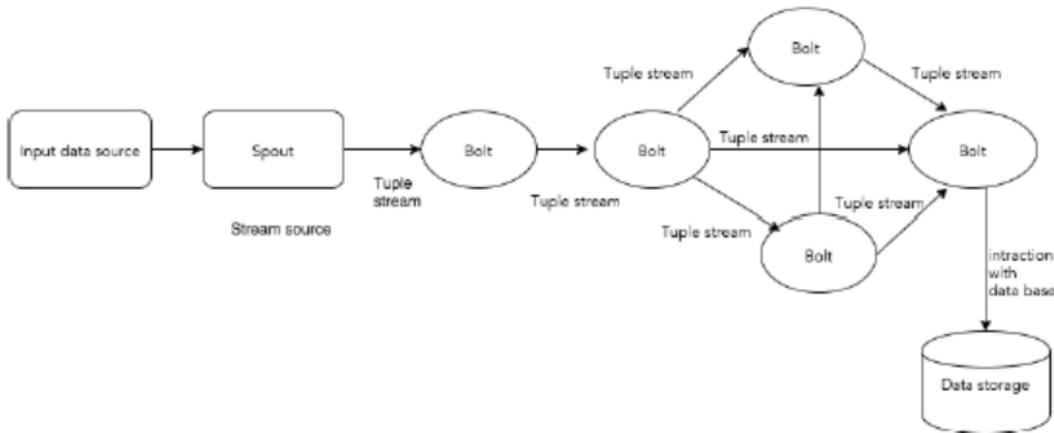


Hadoop 和 Storm 框架基本上用于分析大数据。两者相辅相成，在某些方面有所不同。Apache Storm 完成了除持久性之外的所有操作，而 Hadoop 擅长一切，但在实时计算方面滞后。下表比较了 Storm 和 Hadoop 的属性。

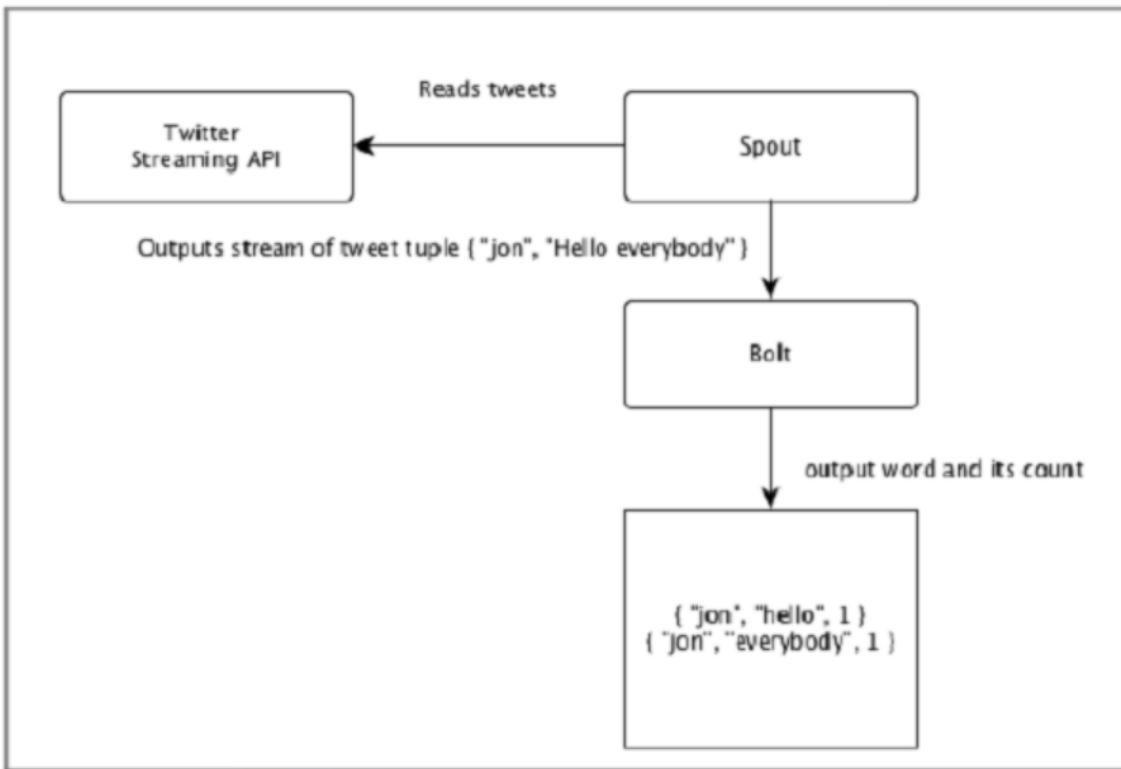
Storm	Hadoop
实时流处理	批处理
无状态	有状态
具有基于 ZooKeeper 协调的主/从架构。 主节点称为nimbus，从节点称为supervisors。	具有/不具有基于 ZooKeeper 的协调的主从架构。 主节点是job tracker, 从节点是task tracker。
Storm 流式处理可以在集群上每秒访问数万条消息。	Hadoop 分布式文件系统 (HDFS) 使用 MapReduce 框架来处理需要数分钟或数小时的海量数据。
Storm topology 会一直运行， 直到用户关闭或发生不可恢复的意外故障。	MapReduce 作业按顺序执行并最终完成。
<b>两者都是分布式和容错的。</b>	
如果 nimbus / supervisor 死了， 重新启动会使其从停止的地方继续， 因此不会受到任何影响。	如果 JobTracker 死掉， 所有正在运行的作业都将丢失。

## 2. Architecture

### 2.1 Basic Concepts



组件	说明
Tuple	元组是 Storm 中的主要数据结构。它是有序元素的列表。默认情况下，元组支持所有数据类型。通常，它被建模为一组逗号分隔值并传递给 Storm 集群。
Stream	Stream 是一个无序、unbounded的元组序列。
Spouts	流的来源。一般来说，Storm 接受来自原始数据源的输入数据，如 Twitter Streaming API、Apache Kafka 队列、Kestrel 队列等。否则，您可以编写 spout 从数据源读取数据。"ISpout"是实现spout的核心接口，具体接口有IRichSpout、BaseRichSpout、KafkaSpout等。
Bolts	Bolts 是逻辑处理单元。Spout 将数据传递给 bolts 和 bolts 进程并产生一个新的输出流。Bolts 可以执行过滤、聚合、连接、与数据源和数据库交互的操作。Bolt 接收数据并发送到一个或多个 Bolt。"IBolt"是实现bolt的核心接口。一些常用的接口有 IRichBolt、IBasicBolt 等。



"Twitter 分析"的输入来自 Twitter 流 API。 Spout 将使用 Twitter 流 API 读取用户的推文，并以元组流的形式输出。来自 spout 的单个元组将具有 twitter 用户名和单个 tweet 作为逗号分隔值。然后，这组元组将被转发到 Bolt， Bolt 会将推文拆分为单个单词，计算字数，并将信息保存到配置的数据源中。现在，我们可以通过查询数据源轻松获得结果。

### 微观上是批处理，宏观上是流处理。

什么意思呢？我发过来的是一个一个的tuple，但tuple本身发过来是源源不断的。

## 2.2 topology

Spout 和 bolts 连接在一起并形成 topology 。 实时应用程序逻辑在 Storm topology 中指定。简单来说，topology 是一个有向图，其中顶点是计算，边是数据流。

一个简单的 topology 结构从 spout 开始。 Spout 将数据发送到一个或多个bolt。 Bolt 表示 topology 中具有最小处理逻辑的节点，并且可以将一个 Bolt 的输出作为输入发送到另一个 Bolt。

Storm 保持 topology 始终运行，直到人为终止 topology 。 Apache Storm 的主要工作是运行 topology ，并将在给定时间运行任意数量的 topology 。

## 2.3 任务

现在你对 spout 和 bolts 有了一个基本的了解。它们是 topology 的最小逻辑单元，topology 是使用单个 spout 和一组 bolt 构建的。它们应该以特定顺序正确执行，以使 topology 成功运行。 Storm 对每个

spout 和 bolt 的执行称为"任务"。 简单来说，一个任务要么是执行一个 spout，要么是一个 bolt。 在给定的时间，每个 spout 和 bolt 可以有多个实例在多个单独的线程中运行。

### 3. Apache ZooKeeper

zookeeper 在帮你在管理一个集群，它做了一个中间的一个协调服务。他就是说他要在一个集群里面，达成这种大家的这种共识这样一致的看法。

然后整个这个集群的管理，当这个集群里面的master死了之后怎么选取新的？就像是那个docker 里的 Kubernetes。

# 第28讲 HDFS

2023.12.20

你记起吗  
任性固执的那幕 无聊事互相反驳  
不知道 不知道 你已不知道  
开心过 争执过 温馨过 挑剔过  
糊涂地陪着你 追忆 追忆 追忆  
模糊地追忆 虽则我 只得我  
——《失魂记》 李克勤

参考资料：

1. HDFS的机架感知与副本放置策略：<https://zhuanlan.zhihu.com/p/341505568>

## 1. HDFS使用场景

适合于：

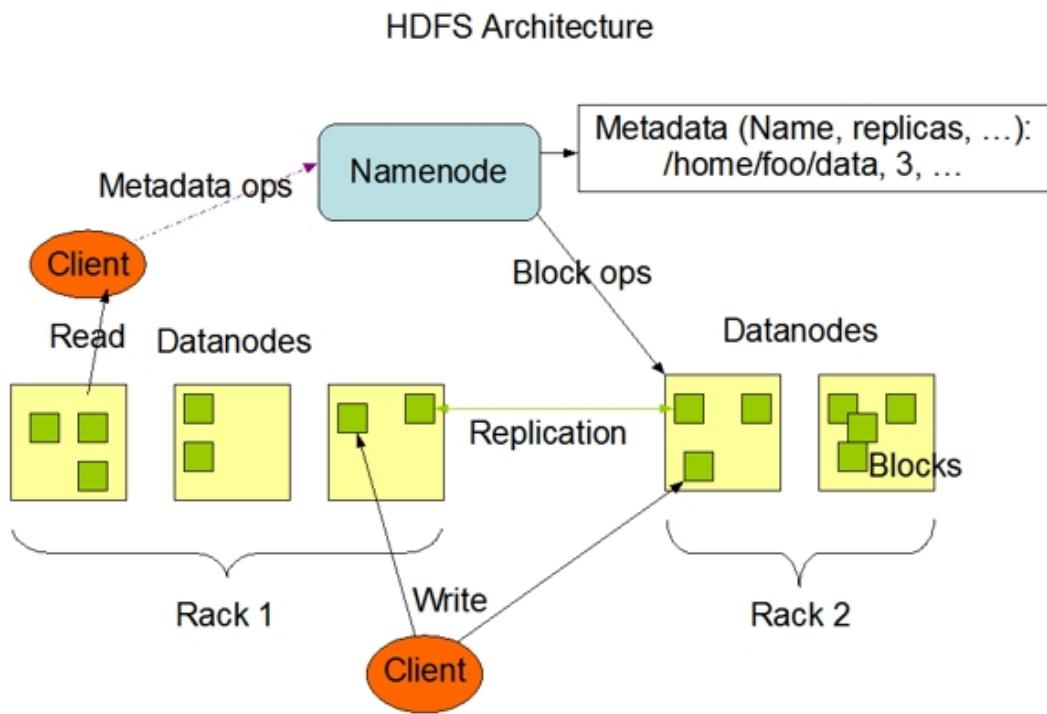
- **适用于非常大的文件**；小于100MB的这种文件，效率反而非常差
- **流式数据访问**；一般来说，数据往里面写只写一次，读要读很多次；需要频繁修改的数据，不适合放在HDFS里面。时序数据，比如influxDB，就比较符合我们这里的设定。
- Commodity Hardware：**用一大堆廉价的硬件**，并不需要非常昂贵的、高可靠的硬件。所以你要拿一堆小的拼一个大的，本来小的机器就没那么可靠，机器多了挂的概率又变大，所以要用空间换可靠性——比如三个副本。为了保证Consistency，如果write开销又很大，所以又回到了第二条。

不适合：

- **低延时的数据访问**：主要是做高吞吐，而不是低延时；就是有大量用户来给我一次性可以从 100 台机器里面同时读数据出来，但是你单独去考察我一个数据访问，发请求到 HDFS 系统，再到得到响应，数据读回来，他没有你现在用的 linux 文件系统效率高。比如我一个巨大的文件，分为1000个 block，存放在1000台dataserver上，我同时去读，效率就很高。但是如果一个小文件，就一个 block，在一个dataserver上，读起来效率就.....
- **很多的小文件**。文件很小，不需要分布到多台dataserver存储，就没意义了。meta data server的文件分区表本身也就会非常大，难以维护。
- **多个写者，任意的、随心所欲的修改方式（比如我们原来定义只能做append，现在说中间任意的修改也可以了）**。consistency是一个挑战。如果有多个写操作进来，我们就让它们串行——single

writer。写也只在文件的最后append。

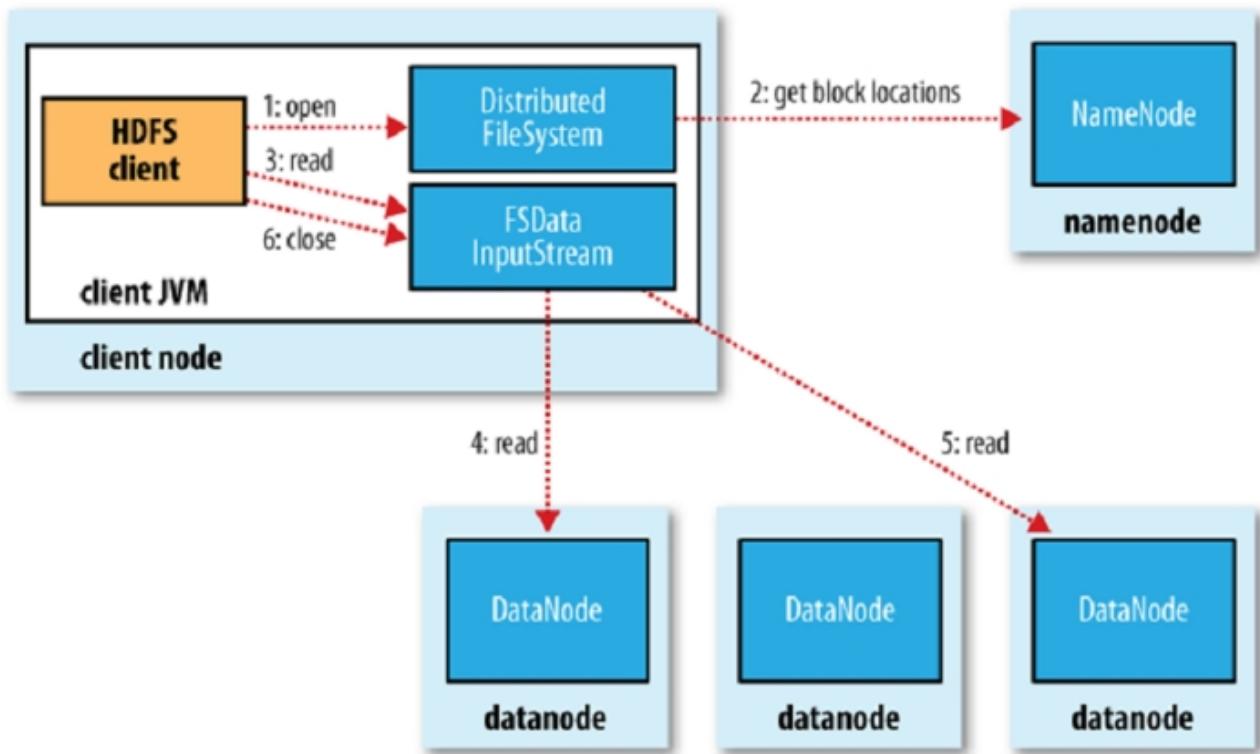
## 2. HDFS架构



整个的逻辑还是master-slave。master也就是namenode/meta data server, slave也就是datanode。

master里面只存储元数据，也就是存储的文件名，有多少副本，每个副本都在哪个dataserver上？它有很多个block，每个block都在哪个dataserver上？这些信息都在master里面。

读数据：

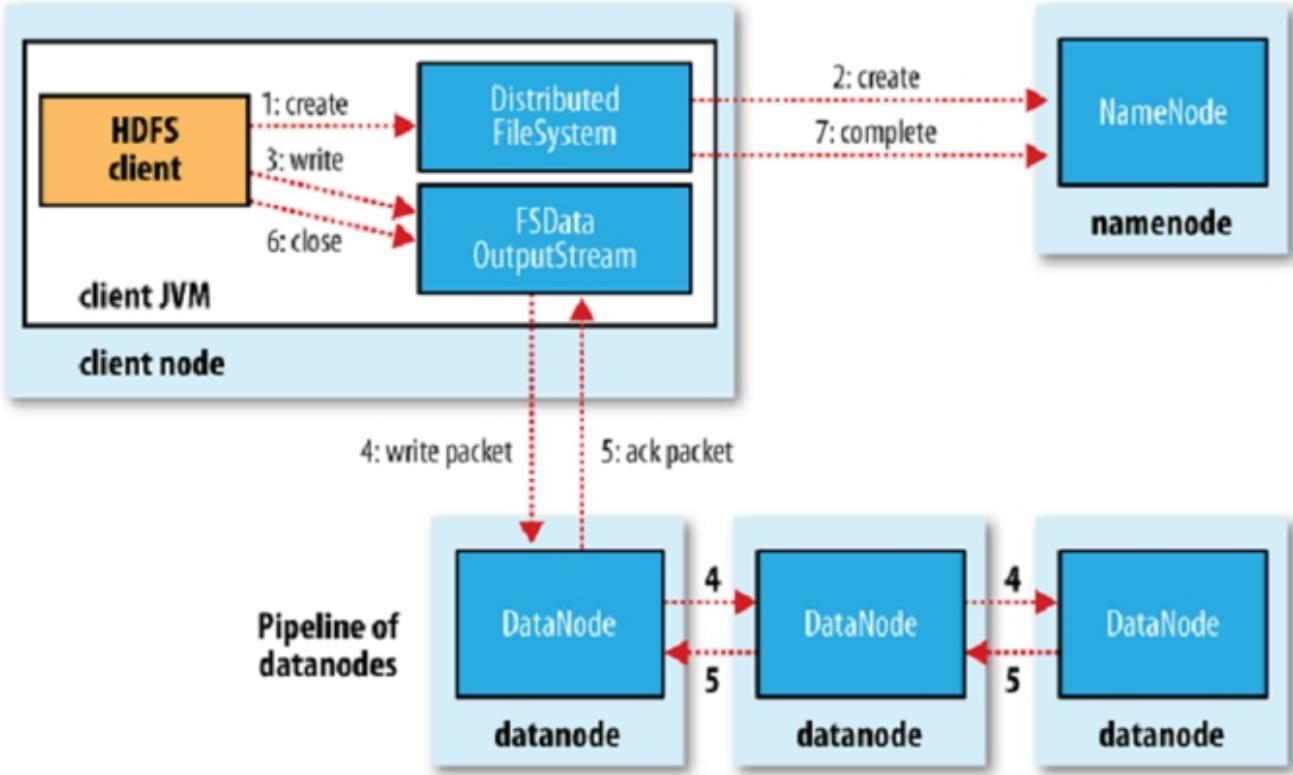


所以客户端client (hdfs的客户端程序，不是你写的程序) 每次都是先和master打交道，我要读什么文件？然后master告诉我这个文件有多少个block，每个block在哪个dataserver上，然后我client就可以直接去dataserver上读了——注意，meta只是告诉你数据在哪儿，client去相应的data server把数据读出来；而不是meta dataserver直接帮你去找data server，这样它就累死了。这个逻辑和nginx是很像的。

再说多一点：比如数据库里面，就是在分布式的系统里面，你所有的东西如果都能做下推，你的效率就会提高很多，因为他们就会充分利用这些是各自节点的计算的能力，而且在这些节点和主节点之间传递数据量会明显减少，这就我们看到的下推的意义。

client的读写都是直接和datanode打交道。namenode对datanode的要求就是，你要不断给我发送心跳告诉我你还活着，我可以往你这里存储数据。datanode本身其实并不需要知道任何的额外信息。这样来看，整个系统可扩展性是很好的。

写数据：



4

当然，这里namenode就成为了单一故障节点——挂了整个系统就挂了。再看看，重要的其实是namenode上的数据；所以这个数据也可以做replica。

### 3. Master

#### 3.1 Log

第一个就是他要管的是你对整个文件系统这个名字空间，比如说我增加了文件或者增加目录，删除了等等，所有的变化或者产生变化的操作全部要记录。它记在哪？它有一个log，就像我们讲 MySQL 所有操作的 log 一样，它也有log；为什么会有log？因为这些操作他也在考能提高性能，从你设计一个系统角度来说，你提高性能，是不是说那我未必，你做的 change 马上就要落盘，可以体现在内存里，隔一段时间再去做硬盘，那所以log就很有用。

当你在内存里还没有去落硬盘的时候，系统崩了，我就把你记下来，下次系统恢复的时候就可以把落的内容重新执行一遍，所以所有的变化都会记到一个 log 里。

#### 3.2 Safemode 安全模式

就是在整个你的 HDFS 在启动的时候，这个 name node 是进入一个叫做安全模式的阶段。这个阶段它是不发生任何写操作的。现在大家都别动，安全模式起来，我先去根据你们这些 data node 给我汇报心

跳和这个 block report 数据，我 name node 先去做一个统计。因为你这个 block report 里面已经列出来所有每个这个数据节点 data block 的这个列表，所以他拿出来之后他就去统计了一下，看看每一个 block 是不是都已经达到了要求的 replica 的数量，如果达到了，我们认为这个检查就通过了。

如果不够，我就要执行复制，一直要复制到这些 replica 数量满足要求。

之前 block 第一次被写的时候不是已经写了满足要求的副本数目了，为什么现在启动了还要做这个检查？

你在上一次启动的时候也许是有 10 个 data note，然后你现在你看是你起了 data note 之后，或许只有 9 个，它不能保证那 10 个 slave 都起来；甚至还有可能是有一些离开的或者加入一些新的节点。因此根本不能够保证相同的 slave 起来了。

## 4. Replica

每个文件的 replica 的数量都是可以指定的，那他有一个极大的问题，就是他告诉你在任何时刻只能有一个 writer；replica 它的数量如果变多，就意味着我这个 write 在执行一次写操作的时候花的时间会越长，就意味着我失败的概率可能越大，而且会造成其他的写操作的堵塞，所以这个 replica 的数量不能设太大，你设太大了，一次写操作要花费太长时间；而它又严格要求只有一个 writer，那你这个效率就太低了。

那所以你要分两个方法来解决问题：

- 第一，replica 它数量不要太大；
- 第二，数据最好只写一次，以后不要频繁的写了。

HDFS 里面，存订单行不行？存 book 行不行？

订单实际上是可以的，订单是你生成之后不会改，book 行不行取决于你的设计。有些同学在里面是设计了一个叫做 stock 的字段，是说这是库存，那你说你把它放到 HDFS 里面去存，是不是他要经常变？那你说我如果把它剥离出来，这 stock 放内存里，这是可以的，所以行不行不能一概而论。

### 4.1 Heart Beat 心跳包的设计

第一个，心跳到底是个啥？心跳就是我周期性发的一些信息，告诉别人我还活着。但问题是心跳里面包含啥？你说心跳就是我 ping 一下，然后那个说收到 pong，这没任何意义，白占带宽，你去想想心跳你要发是不是通过 TCP IP 协议再跟在一个网络里发？那你至少应该填满一个 TCP 的包，让他发成比较合适，对吧？那 TCP 的包也不是那么小，你就发个 ping 我还活着，太浪费了。所以你应该去设计这个心跳里包含的数据包，最好正好能填满一个 TCP 的数据包。那你填啥呢？就填我们这里看到这个 block report（注：A Blockreport contains a list of all blocks on a DataNode），那是一个 TCP 的包的整数

倍，比如一倍、两倍、三倍，这样发心跳是最合适的。所以这地方就说明了第一个问题是到底啥是心跳？**心跳实际上里面是可以带一些你的状态数据的**，状态数据是你来定的。

第二个问题就是，我多长时间发个心跳？这个东西取决于你这个系统，你想多长时间去发现一个节点崩了？比如说我 10 分钟之内，只要发现崩了就行，那所以我定为 3 分钟发一次，你说不是 10 分钟吗，你为什么定 3 分钟？那我要考虑是不是因为网络的原因所以到不了？我给他宽限一点，十分钟我收不到一次心跳我才会认为它挂了。

第三个问题就是心跳怎么发出来？心跳不能是**专门的一个线程**，固定时期，到点以后这个线程发一个心跳出去，这是假的心跳。**真正的心跳应该是在你的主线程里面**，主线程是一个不断的循环，在它的某一步发心跳，这样才能保证你的主线程是一直活着的。如果是单独的一个心跳线程的话，主线程崩了，这个线程一直好，就出事情了。

## 4.2 机架感知与副本存放策略

### 副本存放策略

简要概况起来3点，以3个副本为例：

- 1st replica. 如果写请求方所在机器是其中一个datanode,则直接存放在本地,否则随机在集群中选择一个datanode.
- 2nd replica. 第二个副本存放于不同第一个副本的所在的机架.
- 3rd replica.第三个副本存放于第二个副本所在的机架,但是属于不同的节点.

HDFS采用一种称为机架感知的策略来改进数据的可靠性、可用性和网络带宽的利用率。通过一个机架感知的过程，NameNode可以确定每一个DataNode所属的机架id。

### 机架感知：网络拓扑结构

在海量数据处理中，主要限制原因之一是节点之间数据的传输速率，即带宽。因此，将两个节点之间的带宽作为两个节点之间的距离的衡量标准。

hadoop为此采用了一个简单的方法：把网络看作一棵树，两个节点之间的距离是他们到最近共同祖先的距离总和。该树中的层次是没有预先设定的，但是相对与数据中心，机架和正在运行的节点，通常可以设定等级。具体想法是针对以下每个常见，可用带宽依次递减：

1. 同一节点上的进程；
2. 同一机架上的不同节点；
3. 同一数据中心中不同机架上的节点；
4. 不同数据中心的节点。

# 第29讲 HBase (Google Big Table)

仍然能拥有梦想跟前途  
仍然能拥有自尊跟自豪  
仍然明知许多女伴一转身会遇到  
为何感到 这不算最好  
明明从不信天荒跟地老  
明明从不会后悔得不到  
明明从新掌握去做 我总可以做到  
为何今晚我不懂如何 告别烦恼  
——《寂寞的男人》张学友

参考资料：

1. <http://arthurchiao.art/blog/google-bigtable-zh/>

## 1. HBase

如果我们把Hadoop当作一个云上的操作系统：

1. map reduce就是作业调度
  2. hdfs就是文件系统
  3. hbase就是**数据库** (google的big table的开源版本)
- 数据库的特点：大数据来的时候，结构化的数据是少量的，大量的是半结构化和非结构化的数据。还有一个特点是“大尺寸”，所以我们必然想到的是分布式。

关系型做分布式的问题是，有两个巨大的表，如果我们没有外键关联，那无所谓；但是如果存在了外键关联，如何划分就是一个大问题——否则如果切分得不好，你在做join的时候，某台机器可能要和很多其他机器通信，就很慢。

- 尽管数据量很大，HBase可以做近乎实时的、随机访问的读写。
- 现在我们只有一张表，只有一张大表，这张表里面不存在跟其他表的关联，所以我可以**任意的在某个位置水平切割**，它存到不同的节点上就可以，所以它是一个线性的模块化的这个可扩展性。

## 2. 数据模型

### 2.1 基本概念

- 按列存；建立在HDFS之上。

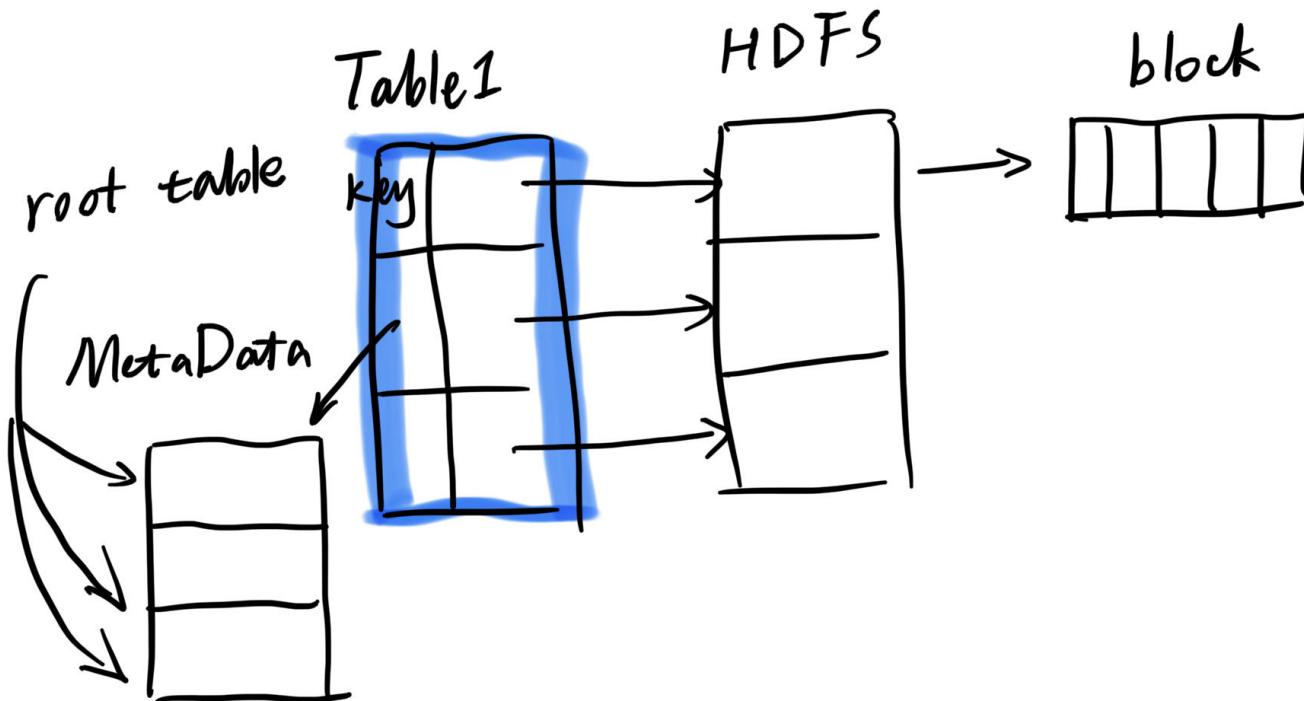
如果我有一个巨大的表，在HBase眼里，我先将其切成三块，比如每块100G，这些信息会记录到meta元数据那边去。

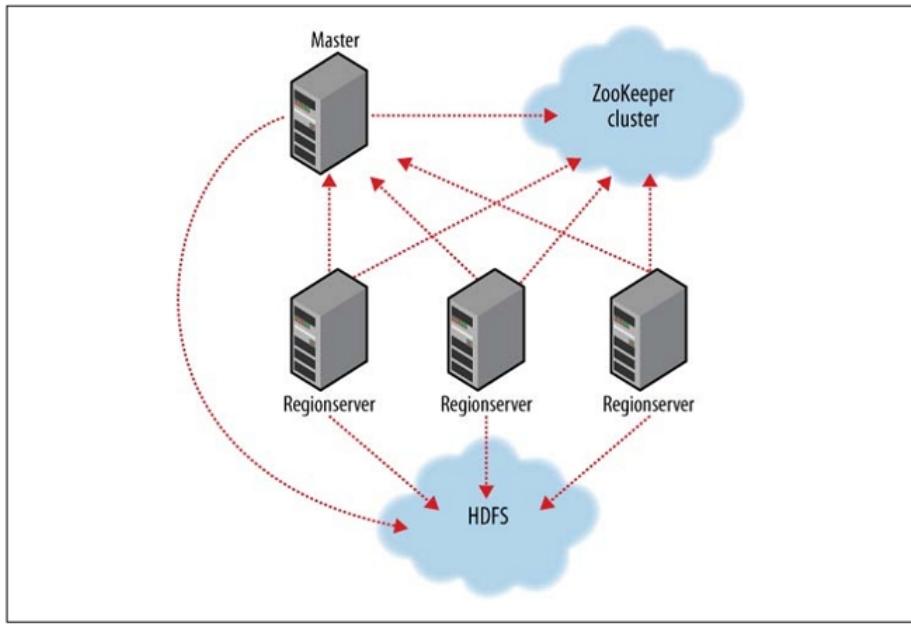
然后它们落到HDFS里面，每块继续切，切成block。

所以实际上我们可以看到这里是两层；第一层的原因是为了加速搜索，比如我有一个key，我按照key的范围做了一个分解：1-1000, 1001-2000, 2001-inf，这样我就可以快速定位到这个key在哪个块里面，然后再去找这个块里面的数据。

当然元数据的表可能也会变得很大，所以也需要切。所以用户查找就变成了：

- 先去root表里面找到这个表的元数据在哪个region里面（region就是你对表水平分区产生的结果，开始的时候一个表只有一个region，后来会慢慢变多）
- 然后去这个region里面找到这个key在哪个region里面
- 然后去这个region里面找到这个key在哪个block里面





- **带版本的。**就刚才讲到如果我把这个数据表只是把外键关联去掉，那我带来了一个问题，是它的数据表达能力会变弱，那于是怎么办？ hbase 就说我让你有几点可以变得跟关系型数据库不一样，你的表达能力会变强。其中有一点就是我在这个维度上就像三维的一样，我在这个维度上允许你做多个版本的存储，比如有一行。对一个字段，可以记 5 个不同时间版本的值，就相当于是一个三维立体的结构。这个就是以关系型数据库不能去表达的。那而且它支持这个 cell 可以说有五个版本，旁边的这个 cell 就只有两个版本。而且大家的版本是靠时间戳来的，所以是可以对齐的。也就是说我想去看一下某一个时间戳下这个数据是什么样，我们就去找每一个 cell 在靠近这个时间戳的时候它的值是什么。尽管你又5个时间戳，你有2个时间戳，我找最靠近你这个时间戳，我就只能知道整个这张表所有的 cell 在这一个时间戳上它的值是什么。
- **所有操作必须全部用主键来。**这里的键必须十个有意义的键，而不是随便的一个东西。比如下面的主键是url。
- 所有的行都是按照主键排序的（存的是字节码，按照字节排序）。
- **列族：**多个列键（column keys）可以组织成 column families（列族）。column family 是访问控制（access control）的基本单位。列键的格式：family:qualifier，其中 family 必须为可打印的（printable）字符串，但 qualifier（修饰符）可以为任意字符串。例如，Webtable 中有一个 column family 是语言（language），用来标记每个网页分别是用什么语言写的。在这个 column family 中我们只用了一个列键，其中存储的是每种语言的 ID。Webtable 中的另一个 column family 是 anchor，在这个 family 中每一个列键都表示一个独立的 anchor，如图 1 所示，其中的修饰符（qualifier）是引用这个网页的 anchor 名字，对应的数据项内容是链接的文本（link text）。列族也满足连续存储，一个列族里面的列放在一起。

这样写那分为列族，它的好处是啥？就是列族那个就相当于把这些列就做了个群组，然后就告诉你在 Hbase 里面，首先你建立一张表，就先要说有哪些列族，你得说我要有一个temperature，然后我就开始记录这个温度，比如我现在又两个列键最高温度、最低温度，我记录了 10 天之后说不对，这个列组里面要新增加一例是他的这个湿度，那么在关系型数据库里面这个操作几乎是不行的，但

是在这个 h base 里就是很容易，那关系数据库为什么不可以呢？那你已经记录了 10 天了，那你说对这十天的数据，应该是多少呢？你说我就让他默认值是0，那也就是意味着你这个湿度“就是”0，湿度是 0 的话，这是有可能是一个正常的一个湿度值，那你就不好处理了，要么你就是让这个值变成是可以为空的。但在 Hbase 里根本就没有这么复杂，你想做你就做吧，你就插入，之前的值没有就是没有。为什么？就还是我们刚才讲到它是列存的，列存的就这一列存在一起，你前面的没有，没有就算了呗，我就没有。将来你在取这一行数据的时候，你是在不同的列里面取，那在这一列里取不到这一列里，它没有这一列的值，他就这么做了。

所以我们看到的分列族，里面包含列，这可以动态的增加，这在关联数据库里是不可能实现的。

## 2.2 Web Table

一个 Bigtable 就是一个稀疏、分布式、持久的多维有序映射表（map），数据通过行键、列键和一个时间戳进行索引，表中的每个数据项都是不作理解的字节数组。

A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

映射：`(row:string, column:string, time:int64) -> string`

我们首先评估了类似 Bigtable 这样的系统有哪些潜在的使用场景，然后才确定了数据模型。举个具体例子，这个例子也影响了 Bigtable 的一些设计：我们想保存大量的网页 和网页相关的元数据，这些数据会被不同的项目使用，这里将这张表称为 Webtable。

在 Webtable 中，我们用网页的 URL 作为行键，网页某些信息作为列键，将网页内容存储在 `contents:` 列，并记录抓取网页时对应的时间戳，最终存储布局如图 1 所示。

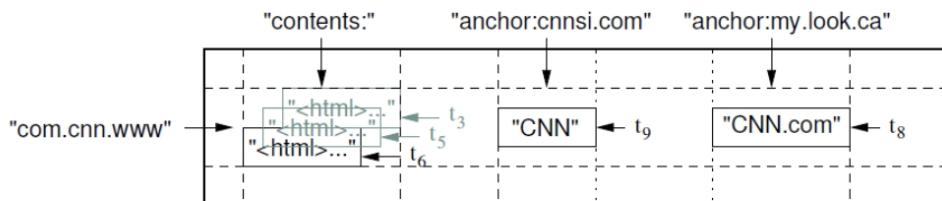


图 1 存储网页的 bigtable 的一个切片 (slice)

- 行索引：`URL`
- `contents:` 列：存储页面内容 (page content)
- `anchor:` 开头的列：存储引用了这个页面的 anchor (HTML 锚点) 的文本 (text of the anchors that reference this page)

图中可以看出，CNN 主页被 Sports Illustrated (`cnnsi.com`) 和 MY-look 主页 (`my.look.ca`) 引用了，因此会有 `anchor:cnnsi.com` 和 `anchor:my.look.ca` 两列，其中每列一个版本；`contents:` 列有三个版本，时间戳分别为 `t3`、`t5` 和 `t6`。

为什么把url倒过来存？因为这样可以让同一个域名的网页存储在一起，这样可以提高访问效率。

# 第30讲 Hive

2023.12.25

明日灯饰必须拆下 换到欢呼声不过一刹  
明晨遇到 亦记不到  
和谁在醉酒中偷偷拥抱  
仍然在傻笑 但你哪知道我想哭  
和谁撞到 亦怕生保  
宁愿在醉酒中辛苦呕吐  
仍然在头痛 合唱的诗歌听不到  
——《Lonely Christmas》 陈奕迅

## Reference

- 林舒怀学姐的笔记

## 1. Hive简介

Hive是基于Hadoop的一个数据仓库工具，可以**将结构化的数据文件映射为一张数据库表**，并提供类SQL查询功能。

其本质是将SQL转换为MapReduce/Spark的任务进行运算，底层由HDFS来提供数据的存储，说白了hive可以理解为一个将SQL转换为MapReduce/Spark的任务的工具，甚至更进一步可以说hive就是一个MapReduce/Spark Sql的客户端

为什么要使用hive？

主要的原因有以下几点：

- 学习MapReduce的成本比较高，项目周期要求太短，MapReduce如果要实现复杂的查询逻辑开发的难度是比较大的。
- 而如果使用hive，hive采用操作接口类似SQL语法，提高快速开发的能力。避免去书写MapReduce，减少学习成本，而且提供了功能的扩展

hive的特点：

- 可扩展：Hive可以自由的扩展集群的规模，一般情况下不需要重启服务。
- 延展性：Hive支持用户自定义函数，用户可以根据自己的需求来实现自己的函数。

- 容错：良好的容错性，节点出现问题SQL仍可完成执行。

## 2. 基本概念

### 2.1 ETL

当使用Hive时，ETL指的是数据的抽取（Extract）、转换（Transform）和加载（Load）过程。这是一种用于从源系统中提取数据、进行必要的转换，然后加载到目标数据仓库或数据库中的流程。ETL是数据仓库和大数据处理中常见的一种数据集成过程。

具体在Hive中，ETL过程可以包括以下步骤：

#### 1. 数据抽取（Extract）：

从不同的数据源（如关系型数据库、日志文件、其他数据存储系统）中提取数据。这可能涉及到连接到外部系统、读取文件、或使用Sqoop等工具来获取数据。

#### 2. 数据转换（Transform）：

在Hive中，数据转换通常指的是对原始数据进行处理、清洗、筛选、聚合或其他变换操作，以使其符合目标数据仓库的格式和结构。这可能包括使用Hive的SQL查询语言执行一系列转换操作。

#### 3. 数据加载（Load）：

将经过转换的数据加载到Hive表中。这可以通过将数据插入到Hive表中，也可以通过外部表或分区表等机制来加载数据。加载的数据通常是经过转换和处理的，使其适应于目标分析或查询。

在Hive中，用户可以使用HiveQL（类似于SQL的查询语言）来执行这些操作。Hive提供了强大的查询引擎，允许用户定义和执行复杂的ETL逻辑。ETL过程在Hive中通常用于准备数据，以便进行分析、报告和其他数据驱动的任务。

## 2.2 Metadata Store 到底存了啥

A Hive table is logically made up of the data being stored and the associated metadata describing the layout of the data in the table.

一般我们用Hive时候，肯定不是创建了表之后，把数据一条一条发进去，都是我有一个很大的文件，然后我把这个文件导入到了某一个表，就是要这样去做的，那比如说这个文件里就有你记录下来的大量的这种传感器传过来的温度数据或者是湿度数据，那就会很多，那比如说这里面就会有这个10万条数据，那我一次性就可以把它给导入到这张表里。

不同格式的文件都可以导入到Hive里头来，于是他就需要一个Meta data store去存储。比如说他会说这个CSV文件里包含了5个列，每个列的类型都是什么？包括我有哪些文件？就这种元数据它要存起来。

从文件中向Hive中加载数据：

```
hive> LOAD DATA LOCAL INPATH '/home/hadoop/data/emp.txt'  
OVERWRITE INTO TABLE emp;
```

注意，这里的意思是，从txt文件里面读进来了所有的东西，放到表格里了吗？并不是的！**这个地方只是在告诉 Hive focus 的内容来自于这个文件，并没有做ETL的动作。**这样想，如果你让我加载这个文件，我就把它读进来，然后去做解析，就一行一行做解析，插入到表里面，这个文件如果巨大，花了二十分钟才搞定ETL，但是说不定我隔了几个小时或者好几天都用不到这些数据！当我真正去在表上面执行HQL操作的时候，我再把数据拉进来也不迟。好处是你这个load你会发生发现非常快就结束，比如说我复制过来就结束，但是你会发现以后再执行 HQL 的时候就比较慢，但是他觉得这又不是一个大问题，首先来说我没有写操作，刚才我们一上来就说数据仓库其实没有写操作（**数据仓库一次写入多次读，因为不是原始数据，改没有意义**），都是分析操作，而且你是大量数据，你本来就需要大量时间去进行这个处理。

## 2.3 File Formats and Compression 支持的文件格式

- TextFile
- SequenceFile
- RCFile: row columnar file
- ORCFile: optimised row columnar file
- Avro File
- **Parquet**: 它现在事实上是一种标准，这已经成为一个阿帕奇的单独的一个项目，那么他就在告诉你所有的数据，比如说我们现在有 a 类型的数据库和 b 类型数据库，大家如果想交互，可以全部都转成它是一种格式。

# RCFile

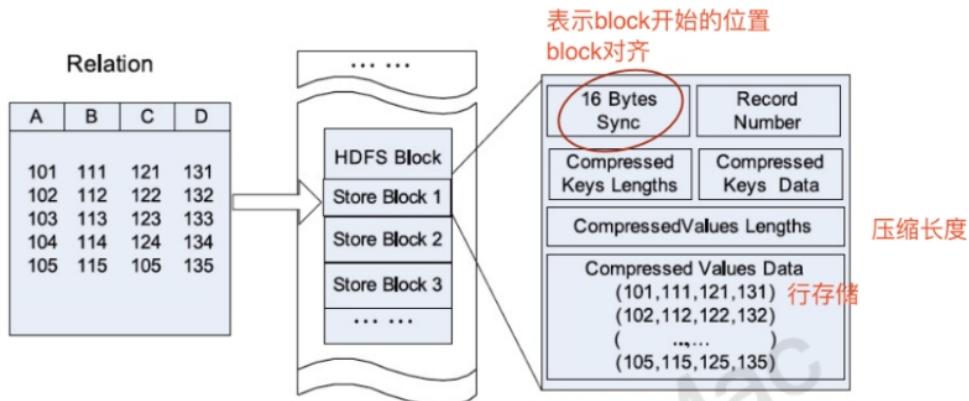


Fig. 1: An example of row-store in an HDFS block.

上图中，我们可以发现是按照行来存储并且压缩的——其实这样做压缩是不合理的，比如第一行101, 111, 121, 131，你看他就是公差为10的等差数列，好像也可以啊，真实情况下，比如说 a 是数据， b 是字符串， c 是可能表示奇葩的东西，那可能就寄了。

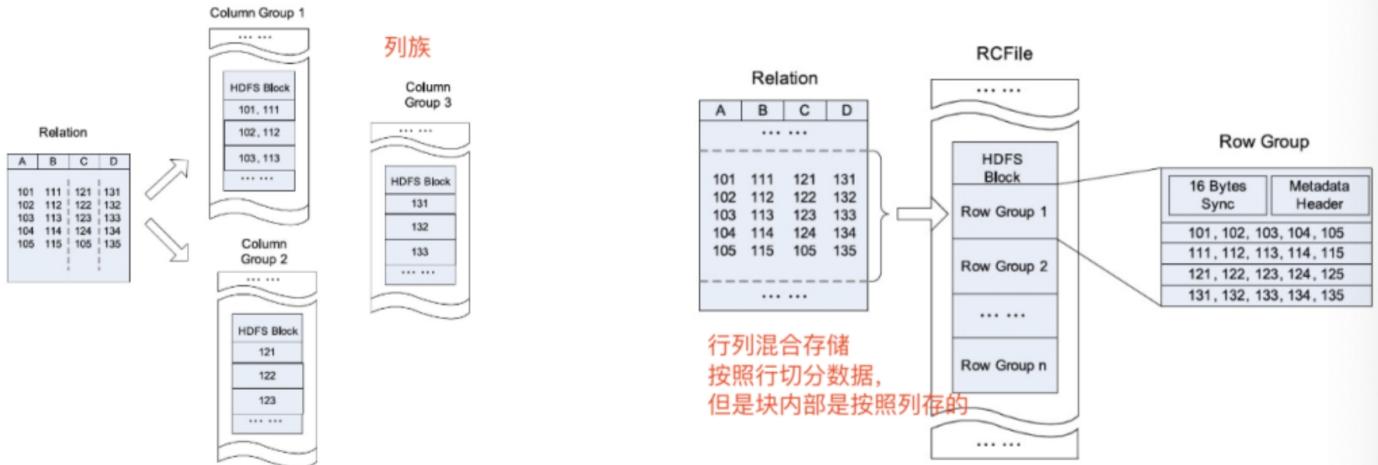


Fig. 2: An example of column-group in an HDFS block. The four columns are stored into three column groups, since column A and B are grouped in the first column group.

Fig. 3: An example to demonstrate the data layout of RCFile in an HDFS block.

2011 ICDE conference paper "[RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems](#)" by Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu

## 恶趣味：考试无关向

细致分析一下这篇论文。

为什么要进一步压缩文件让文件变得很小呢？注意看这篇论文的题目——是在map reduce的数据仓库里面搞事情！map reduce的问题是什么？就是大量的磁盘io，这也是spark出现的原因。那么既然频繁disk io，我就让文件小一点，速度就能提高。

### 3. 和传统数据库的对比

#### Schema on read

- 就是这个数据在 load 的时候（我把一个文件要load到一张表里）没有去直接做 ETL 这种，它是等到当你要去执行一个 HQL 的时候，它才去做这个文本的校验，去做ETL，去做这个文本的抽取、转换、加载等等。

#### Schema on write

- 对于MySQL，输入在插入/load的时候就必须要做校验（schema on write），因为在传统数据库里面，它可能数据量没有像 Hive 这么大。Hive 实际上它是个 data Warehouse，它是一个这个数据仓库，它不是一个数据表。也就是，本质上你在MySQL里面，你存储了订单、书籍信息，还有用户信息，然后你把订单的数据导入到data warehouse里面，然后我要去做在线分析，比如我有很多家电子书店，统计总共卖书的情况？

换句话说：

- MySQL这种是**面向事务的**，一定要求数据的一致性、完整性；
- Hive 为什么要 schema on read？因为大量的数据过来，如果我一开始就要去做这个校验，大量的时间就在前面就花掉了，那么我这个花掉这个时间值不值也不知道，因为我还不知道什么时候会去做这一步分析这个动作。

所以，严禁把Hive拿过来当作电子书店的数据库来用！数据仓库总的来说就是他要把各种各样的数据，比如说我们班上 100 个同学开发各自的电子书店，我要把大家的数据全部都拿过来去做汇总，我就拷贝 100 个文件在那里。至于你要在上面去做订单总数的统计，还是订单价格总额的这个汇总，那是等到你执行一个 HQL 的时候我再去做分析；这个时候load就非常快。大家用的数据库也各不相同，MySQL，MongoDB，都无所谓，我支持你用统一的一条语句查询，而且我支持用mapreduce方式在多个机器上面同时对这100个文件来做查询。

其实它已经具备了数据湖的能力；数据湖里面强调，数据未必都要按把它们转成某种特殊的格式，就按他们原始格式放在这里就行。一些数据湖事实上就是在Hive的基础上改造的。

#### 目前数据湖和数据仓库的三种关系

- 数据湖存所有的数据原始文件，作为数据仓库的数据源，在数据仓库做高级操作时直接从数据湖找数据
- 数据仓库里的数据是文件格式区别的，数据湖可以翻译不同格式的数据，就是直接把数据湖作为数据仓库，都可以用SQL方式访问数据湖
- 介于前两种之间，当数据的使用高于一定频率时移动到数据仓库里，即热数据存在数据仓库里，可以提高处理这些数据时的效率，这种关系也称湖仓一体 (lakehouse)，是hive的前身

# 第31讲 flink

2023.12.27

出发啦不要问那路在哪  
迎风向前 是唯一的方法  
出发啦不想问那路在哪  
运命哎呀 什么关卡  
当车声隆隆 梦开始阵痛  
它卷起了风重新雕塑每个面孔  
夜雾那么浓 开阔也汹涌  
有一种预感路的终点是迷宫  
——《亡命之徒》 纵贯线

## 1. 基本概念

Flink要处理流式数据。所谓的流式数据，就是我们看到有很多的事件，然后试卷它在发生的时候，它会一个一个的产生，然后他们就进入到了这个流失数据的这个处理系统里面。

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams.

### Streams:

- bounded and unbounded streams
- real-time and recorded streams 所谓recorded，比如我处理速度没有那么快，我就在flink前面加一层kafka消息队列，存在topic里面。

Flink要做的事情：

1. 如果多个处理器，如何分区？也即数据如何按照一定的规则，放到不同的节点上面去执行；
2. 一旦这样做，可能产生乱序。如果不是按照时间产生的顺序进入系统，何如？例如每到整点处理一次，九点钟开始处理八点到九点数据，但九点十五还有八点到九点的数据被发过来；
3. 我们现在有两个产生事件的源；它们要隔离开来，也即这两个源在处理的时候，我要知道它们的上一个事件，要与之关联。也即所有操作必须时有状态的，类似session的那套逻辑。要记住状态这件事情是我们前面在讲Hadoop、Spark、storm 里面都没有了，我们都是给这三个东西提交了一个作业，这个作业对于Hadoop和这种批处理来说，那就这一块数据，你现在让我执行的这一块跟

上一块之间是没有任何关系，反正我就每一块来一块处理一块，那 storm 的这个流的处理也是一样的，你让我处理我就处理，至于他跟之前的事件什么关系，我也不知道，我也不用去管理他。

## 1.1. State?

所谓状态，就是你一个对象，你里面会有很多的数据成员，不同的对象的这些数据成员可能是不一样的——例如购物车，里面有很多kv pairs，那么不同的人的购物车里面的键值对都是不同的。

比如我想要知道一个人在网上的访问模式，必须把这个人的所有的点击动作要把它关联起来，那么它当前已点击到哪里，它上一次点击到哪里，这就是他的一个状态了。

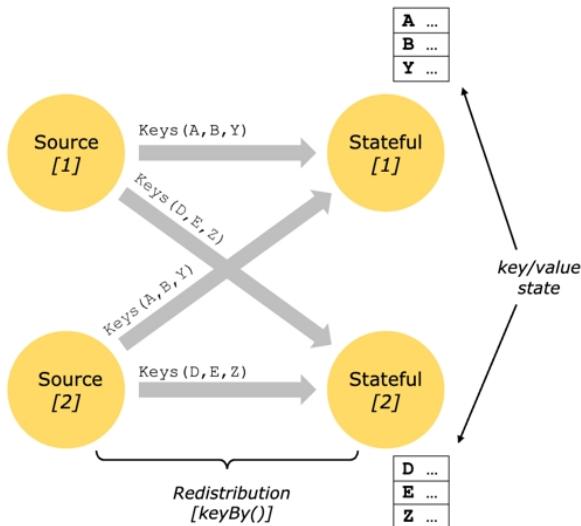
Flink 里头，他说我在处理这个事件流的时候，总会产生一些中间结果，我把它存到一些变量，这些就是状态，这些状态必须在这一堆的事件前后，我们必须去做关联。

我们第一节课讲的状态是讲的是，实际上每一个对象（每一个用户）都维护了单独的状态，所以每个对象的状态都要保存，当我在内存里不够的时候，我要把它写到硬盘上去，然后当用到它的时候我再把它给拉回来，就防止这个状态太多之后把内存给消耗完了，所以它有换进换出这样的动作。这里的话，状态也要考虑这个问题，可以写在rocksdb里面，做到可插拔，就直接重启以后去数据库恢复就好，经常去做checkpoint，来容错。

- 周期性的去做这个snapshot，然后他这个周期性的做他还会考虑到几点，就第一个不能影响到现在的这个事件的处理的这个流，所以他要去异步的做，就是比如说隔一段时间做一下，而且就是你前端正在改修改，后端慢慢去往里写，不一定说我要前端停一下，让他先到后端，写完之后再说。你前端继续执行前端一端执行他的状态，一边去做这个备份。
- 然后它是增量，肯定不能做全量，而是增量的，我每一次做 snapshot 时候就把这一段时间内变化的值给写过来，这样的话可以靠最小的这个开销就可以来实现这个做这个状态的这个持久化。

Keyed state is maintained in what can be thought of as an embedded key/value store.

- The state is partitioned and distributed strictly together with the streams that are read by the stateful operators.



这里kv，保证的是key一样的事件，都会到某台机器上处理，所以它的状态得以维护，这里engine X里面的ip哈希是类似的逻辑。

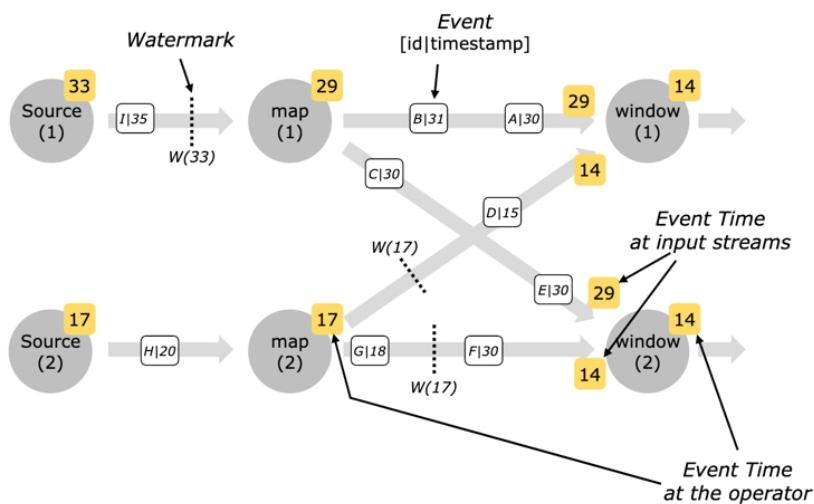
## 1.2 Watermarks 水位线：解决少量数据的延迟到达问题

我们真正关心的是按照事件产生的时间去处理，但是一旦乱序，怎么办？所以引入了“**水位线**”的概念（watermarks）。

如果我是8点到9点的数据，一个时间窗去处理，9点到10点一个时间窗去处理，但是在这个9点零二分的时候来了一个9点的数据，因为网络传输延迟。最简单的解决方案就是**提供一个你可以接受的延迟时间**，比如我定义九点零五分为ddl，他在他九点零二分来的，那他仍然可以当成这个时间窗户数据处理。你说那不对，那如果他要是九点零六分来这种都属于确实超出了我的容忍范围，那我就几种不同的选择，第一种就把它放到下一个时间窗处理，第二就把它扔了，那反正事件不断的拦，就是这是你一个选择的问题而已。

- Watermarks in Parallel Streams

- The figure below shows an example of events and watermarks flowing through parallel streams, and operators tracking event time.



我们来看一下这个比较复杂的计算图。这里面灰色的就是若干算子，黄色的就是每一个算子的时间戳。注意分布式的时钟是不同步的。

举个例子，这个source上面当前的时间它走到了33，然后它就会说它会发出一个他已经走过33的一个水位线的一个数据出来，水位线数据会和所有的事件混在一起往前走。map2的时间戳应该是14，写错了。

在接收多个流的数据过来的时候，取较小的这个流作为我当前这个算子的时间，这算自以这个时间为基准，在它产生的事件上去打时间戳，并且发这种我们看到这个水位线的数据出来。

### 窗口机制

- 窗口机制有两种，一种是上面这个时间窗口，比如说每 30 秒我处理一下这个窗口里的所有的数据，还有一个是数据驱动，就是我数个数，每 3 个事件我就处理一下。
- 总的来说就是我们拿窗口里的数据去做批处理，每一个批处理完之后紧跟着处理下一批。从宏观上看，于是它就是流式的数据处理了。

总结：我们是要靠这种时间戳加水位线的这种方式在告诉每一个算子过来的事件，它们的先后顺序什么样，你应该怎么把它组织到你这样的一个时间窗里面，在这个时间窗里激发对所有的时间的处理。

## 2. Architecture

它的状态是要频繁的使用内存的，而且他这个就是我们刚刚看到的，就是所谓的状态，是指你在每一次你的逻辑在处理这个事件的时候，在拿到事件的信息之后，你这个逻辑程序还要去读一下状态，才能知道怎么处理这个事件，而这个状态就在内存或硬盘上。

