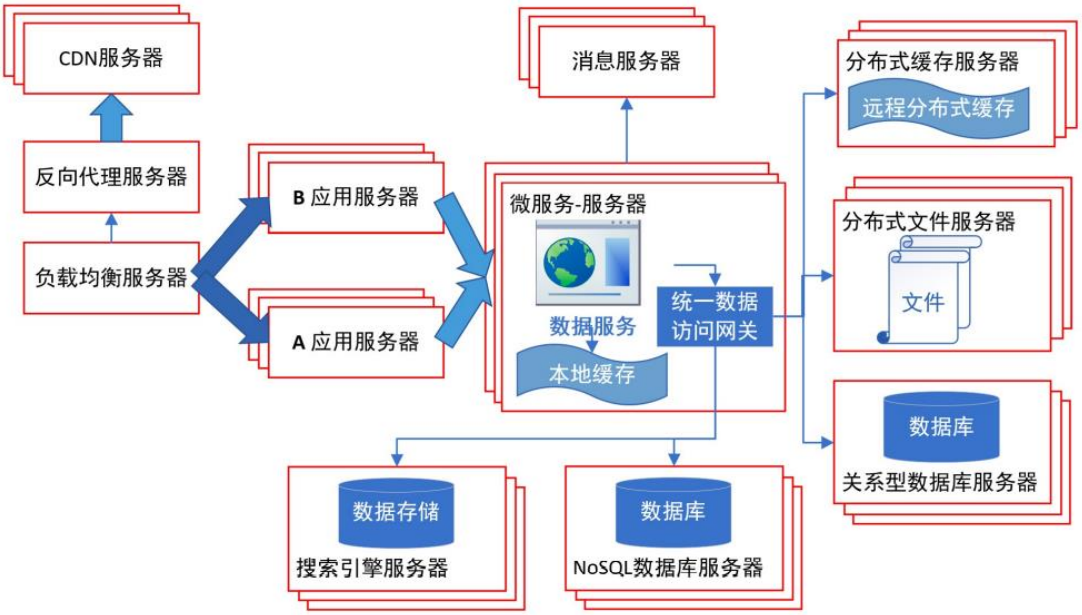
# 应用系统体系架构

*Wang Haotian*

## 目录

## Architecture & Service Component




为什么要将文件服务器、数据库服务器单独分出来？为了防止别人可以通过 app 的服务器攻击到数据库和文件，所以这两个用内网 IP 进行访问。不过分 server 也带来一个问题，那就是数据的不一致

数据库主从备份，但从不做事情会有点浪费，所以承担一部分读的功能，既然这样，现在数据库也是集群了。

使用 CDN 将视频等大资源转移出去，也加快了用户获取的速度

数据库未必用关系型，不同类型的数据库适合于不同的情景

异步通信会有消息中间件

系统中的应用会有多个，应用服务器、消息服务器还有各种公用的服务，其他应用都会用到的，可以抽离出来做微服务

### 关于 Scope

- singleton：全局有且仅有一个实例
- prototype：每次获取 bean 时会有一个新的实例
- request：针对每一次 HTTP request 都会产生一个新的 bean，同时该 bean 仅在当前 HTTP request 内有效

- session：针对每一个 HTTP session 会话都会产生一个新的 bean，同时该 bean 仅在当前 HTTP session 内有效

### 关于状态

考虑这种情景：

无状态：service 里有一个 uid，我希望写入 order 表中。但若无状态，只有一个 service，那么不同用户过来之后，我只会记录下最后一个人的 uid。

有状态：每个人来都有一个单独的 service 和一个单独的 uid。

但有状态实质却会带来内存消耗。

tomcat 中有一个 instance pool，每个 service 实例会装在里面，新用户来了会 new 一个新实例，装不下后会将老实例写到硬盘里（LRU），后面再用的时候从硬盘读出来。

这带来两个问题：

1. 硬盘重复 IO 开销大
2. instance pool 中若分配 instance 太多，则浪费空间，太小了了重复 IO

但若没有状态，则无法维护这样的变量，会带来问题。

有无状态需要进行权衡，建议多数为无状态，少数为有状态

## Messaging

JMS

Point-to-Point 和发布订阅模式

JMS 天生就是异步的，客户端获取消息的时候，不需要主动发送请求，消息会自动发送给可用的客户端。

一些要点：

1. 实现：前端消息发到 controller，controller 把东西放进消息中间件特定的消息队列里，然后立即返回告诉客户端 OK。有一个 service，在空闲的时候再对消息进行处理。
2. 优点：

- 异步，不需要一直等结果才能返回，性能会好很多
- 可信，保证一定能被执行，收到结果。虽然暂时可能不会处理，但总有一天会处理

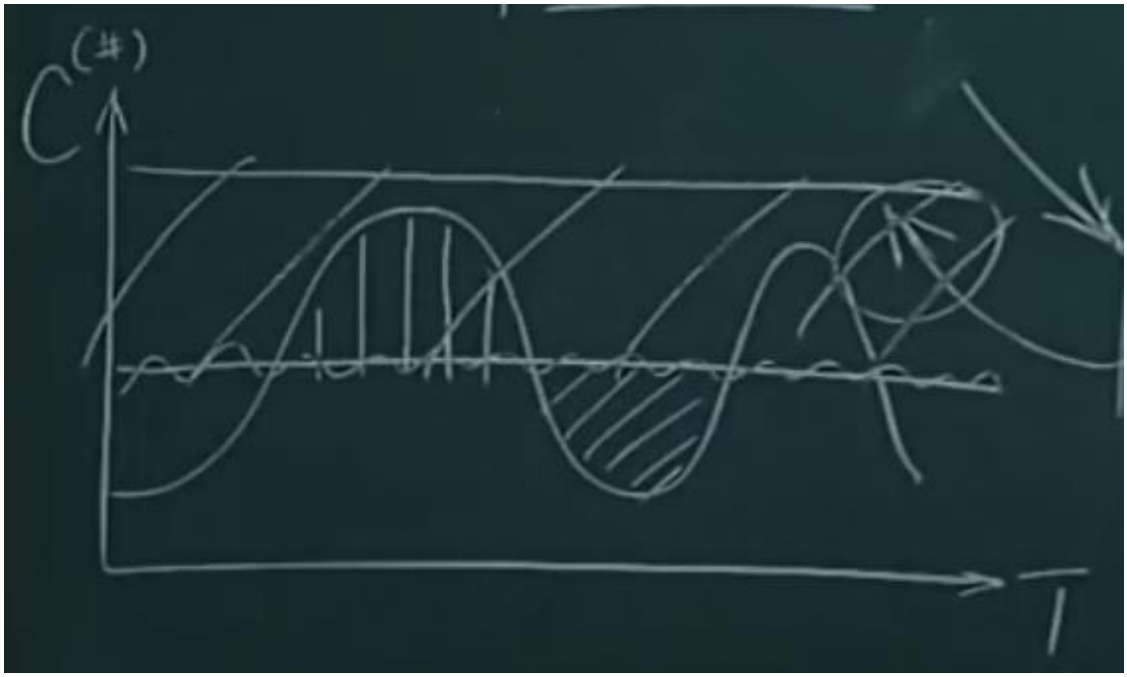
3. 对消息双方而言：哪怕消费者的代码重构了，都不会对生产者产生影响。所有的通信全部都是走消息，两边互不影响。消息队列中存储的不再是 java 类，而是 json 等纯文本对象。交互的时候生产者将 java 类转为 json，消费者将 json parse 为 java 类然后进行处理。
4. 消息也会带来麻烦：
    - 消息是弱类型，编译器没法发现传过来的消息有问题，消费者可能处理的时候可能有异常
    - 消息的传递，以及 java 对象和用于传递的纯文本（json）等类型之间的转换也会带来很大的时间开销
    - 消息结果需要专门的方法返回，异常也需要有额外的处理方式

综上所述，优先使用同步，只有性能真的跟不上的时候再用消息队列，比如双十一；还需要考虑处理能力和负载量的关系

如下图：

- 负载量远超处理能力
- 负载量存在周期性较大波动
- 负载量一直是较小波动

当负载量周期性波动较大时，是比较适合使用消息模型的



### Point-to-Point：

应用程序由消息队列，发送者，接收者组成。每一个消息发送给一个特殊的消息队列，该队列保存了所有发送给它的消息(除了被接收者消费掉的和过期的消息)。点对点消息模型有一些特性

- 每个消息只有一个接收者
- 消息发送者和接收者并没有时间依赖性
- 当消息发送者发送消息的时候，无论接收者程序在不在运行，都能获取到消息
- 当接收者收到消息的时候，会发送确认收到通知（acknowledgement）

## 发布订阅模型

在发布/订阅消息模型中，发布者发布一个消息，该消息通过 topic 传递给所有的客户端。在这种模型中，发布者和订阅者彼此不知道对方，是匿名的且可以动态发布和订阅 topic。topic 主要用于保存和传递消息，且会一直保存消息直到消息被传递给客户端。发布/订阅消息模型特性如下：

- 一个消息可以传递给多个订阅者

- 发布者和订阅者有时间依赖性，只有当客户端创建订阅后才能接受消息，且订阅者需一直保持活动状态以接收消息
- 为了缓和这样严格的时间相关性，JMS 允许订阅者创建一个**可持久化**的订阅。这样，即使订阅者没有被激活（运行），它也能接收到发布者的消息

### 接收消息

在 JMS 中，消息的接收可以使用以下两种方式：

- 同步：使用同步方式接收消息的话，消息订阅者调用 receive()方法。在 receive()中，消息未到达或在到达指定时间之前，方法会阻塞，直到消息可用。
- 异步：使用异步方式接收消息的话，消息订阅者需注册一个消息监听者，类似于事件监听器，只要消息到达，JMS 服务提供者会通过调用监听器的 onMessage()递送消息。

### Browser

允许你浏览队列中的消息，并且展示消息头

Browser 可以实现只看消息而不消费

## Websocket

WebSocket 是基于 TCP/IP 协议，独立于 HTTP 协议的通信协议。

WebSocket 是双向通讯，有状态，客户端与服务端双向实时响应（客户端 ⇄ 服务端）

建立 Websocket 连接需要三次握手

## Transaction

### 事务的传播属性

Propagation：

- PROPAGATION_REQUIRED：支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择

- PROPAGATION_SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行
- PROPAGATION_MANDATORY：支持当前事务，如果当前没有事务，就抛出异常
- PROPAGATION_REQUIRES_NEW：新建事务，如果当前存在事务，把当前事务挂起
- PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
- PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

## 事务的隔离级别

### *第一种隔离级别：Read uncommitted(读未提交)*

如果一个事务已经开始写数据，则另外一个事务不允许同时进行写操作，但允许其他事务读此行数据，该隔离级别可以通过“排他写锁”，但是不排斥读线程实现。这样就避免了更新丢失，却可能出现脏读，也就是说事务 B 读取到了事务 A 未提交的数据

解决了更新丢失，但还是可能会出现脏读

### *第二种隔离级别：Read committed(读提交)*

如果是一个读事务(线程)，则允许其他事务读写，如果是写事务将会禁止其他事务访问该行数据，该隔离级别避免了脏读，但是可能出现不可重复读。事务 A 事先读取了数据，事务 B 紧接着更新了数据，并提交了事务，而事务 A 再次读取该数据时，数据已经发生了改变。

解决了更新丢失和脏读问题

### *第三种隔离级别：Repeatable read(可重复读取)*

可重复读取是指在一个事务内，多次读同一个数据，在这个事务还没结束时，其他事务不能访问该数据(包括了读写)，这样就可以在同一个事务内两次读到的数据是一样的，因此称为是可重复读隔离级别，读取数据的事务将会禁止写事务(但允许读事务)，写事务则禁止任何其他事务(包括了读写)，这样避免了不可重复读和脏读，但是有时可能会出现幻读。(读取数据的事务)可以通过“共享读镜”和“排他写锁”实现。

解决了更新丢失、脏读、不可重复读、但是还会出现幻读

*第四种隔离级别：Serializable(可序化)*

提供严格的事务隔离，它要求事务序列化执行，事务只能一个接着一个地执行，但不能并发执行，如果仅仅通过“行级锁”是无法实现序列化的，必须通过其他机制保证新插入的数据不会被执行查询操作的事务访问到。序列化是最高的事务隔离级别，同时代价也是最高的，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻读

解决了更新丢失、脏读、不可重复读、幻读(虚读)

## Multithreading

明确一些线程和进程的概念：

- 进程：每个进程有自己的运行资源、运行环境和内存空间
- 每个进程内可以有多个线程，进程内的线程共享内存、运行环境、打开的文件等等，他们可以直接互相访问，效率很高。

### Interrupt

如果有一个 long running thread，那么应该经常判断是否被 Interrupted 了，这个要靠写代码来检查

但是如果是 `Thread.sleep(10000)`期间被 interrupt 掉，是会直接抛出 InterruptedException 的

```
public class MyThread implements Runnable{
    @Override
    public void run() {
        try{
            for(int i = 0;i < 100000000000;++i){
                System.out.println("Still running!");
                if (Thread.interrupted()){
                    throw new InterruptedException("Yor are interrupted!
");
                }
            }
        }
        catch (InterruptedException e){
            System.out.println(e.getMessage());
            System.out.println("Oh no, I wasn't done!");
        }
    }
}
```

多线程锁：

Java 的 Synchronize 关键词，使用对象的锁进行的，而且一个对象只有一把锁

但是 Java 中的锁是可重入的！一个线程可以获得他已经得到的锁

比如下列语句

```
public class Score {
    int pts = 0;

    public synchronized int getPts(){ return pts; }
    public synchronized void incrementPts() { pts++; }
    public synchronized void decrementPts() { pts--; }
    public void incrementPts2() {
        synchronized (this) { pts++; }
    }
    public void decrementPts2() {
        synchronized (this) { pts--; }
    }
}
```

### Live Lock

活锁指的是任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。 活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的“活”， 而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。活锁可以认为是一种特殊的饥饿。

### Starvation

饥饿，与死锁和活锁非常相似。是指一个可运行的进程尽管能继续执行，但被调度器无限期地忽视，而不能被调度执行的情况。饥饿可以通过先来先服务资源分配策略来避免。

### Guarded Blocks

其实就是 wait+notify

### Thread Pool

如果对于每个 http 请求都创建一个线程，那很容易被攻击，通过大量 http 请求就让服务器爆掉了。所以我们用线程池，在请求到来的时候会分配一个线程对象给他，当请求完成后会把这个线程对象收回到线程池中保留。

当线程对象都被占用，而新的请求到来时，是创建新的线程对象，还是将一个正在被占用的线程对象分配给他进行关联，都是具体有线程池决定的。

所有对线程的调度都由线程池来决定。

## Caching

### MemCached

是一种 key-value store。使用的是一致性哈希来确定 key 的位置，使得移除增加结点开销也很小，在集群中也不需要大规模扫描，所有性能很好

### Redis

redis 可以直接把 java object 写到硬盘，省去了解析的时间开销。看似 redis 把缓存的东西落在硬盘上很傻，但是考虑到落在的是 SSD 上，读取速度很快，并且没有解析的时间开销，更重要的是，为了安全起见，数据库放在单独的机器上，而 redis 和 tomcat 虽然是不同进程，但运行在同一台机器上，从另一台机器的 SSD 读取也比网络传输更快。（当然其他的数据库不会存在 SSD 里面，redis 存 SSD 是因为比较小，且是存缓存）

### Searching

1. 反向索引：是指根据具体词语去查询在文档中出现的位置，这相对于我们之前根据索引位置查单词是相反的
2. Fields 分类：
    - keyword：不会被分词器断开，但会用来生成索引
    - unindexed： 既不会被断开，也不会用来生成索引影响索引的位置，但会存在索引当中
    - unstored：会被分析和用来生成索引，但不会存在索引当中（因为太长了）
    - text：会被分析（断开）和用来生成索引

例如，对于一个项目而言，文件名 a.txt 是 keyword（并不会将文件的后缀进行专门的断开），文件位置是 unindexed，源码是 unstored，注释是 text（会逐词断开参与到索引生成）

3. Heterogeneous Documents 异构文档

   允许有不同字段的文档在同一个索引中共存，也就是说，一个 Document 有 Field（name，price），下一个 Document 可以有 Field（name，age），两个 Document 可以代表完全不同的记录

4. 每层加新的东西不必删除索引，索引可以增量着建

5. 可以设定一个 booster，booster 越大就越在上面，更容易被找到

6. lucene 开的内存越大，速度越快；还能支持并行操作

## Web Services

希望能为不同语言提供调用接口（用纯文本），还要用能走足够远的协议

这种纯文本可以用 json，也可以用 SOAP（用 XML 写的）

### SOAP

SOAP 简单对象访问协议

- 必须的 Envelope 元素
- 可选的 Header 元素，包含头部信息
- 必须的 Body 元素，包含所有的调用和相应信息
- 可选的 Fault 元素，提供有关在处理此消息所发生的错误的信息

语法规则：

- SOAP 消息必须用 XML 来编码
- SOAP 消息必须使用 SOAP Envelope 命名空间
- SOAP 消息必须使用 SOAP Encoding 命名空间
- SOAP 消息不能包含 DTD 引用
- SOAP 消息不能包含 XML 处理指令

### WSDL

WSDL（网络服务描述语言，Web Services Description Language）是一门基于 XML 的语言，用于描述 Web Services 以及如何对它们进行访问。

### SOAP 和 WSDL 的关系

wsdl 与 soap 的关系在于：wsdl 绑定服务的时候可以设定使用的协议，协议可以是 soap、http、smtp、ftp 等任何一种传输协议，除此以外 wsdl 还可以绑定 jms、ejb 及 local java 等等，不过都是需要对 binding 和 service 元素做扩展的，而且需要扩展服务器的功能以支持这种扩展

### WSDL 例子

例如 java 程序想调用 C#的程序：

C#生成一个.wsdl，大公司像支付宝这样的，会把这玩意放在自己特定的网站上；小公司放在阿里云这样大家都知道的地方让 java 那边下；

java 这边下载下来之后，用 axios 等来根据.wsdl 来生成一个 java 的接口类，同时还会生成这个接口类的一个实现（并非实现业务，而是实现能够将请求传递到 C#那一边）

C#这边用 WCF 等东西，同样根据.wsdl 生成代理和接口，可以来调用 C#中对函数真正的业务实现，然后通过代理将结果返回给 java 那边。

### Restful Web Service

rest 是无状态的服务器，所有的状态都保持在客户端和服务器之间的消息中，服务器端只处理数据需求，显示完全依赖于客户端。rest 是幂等的，因此结果可以缓存到客户端或者服务。

restful 是以数据资源为中心的，好处如下：

比如如果你 url 叫/getPrice，/updatePrice，如果以后后端的 url 改了，那么前端也要改，就很麻烦。如果以数据为中心，url 统一叫/price，只用 get，post 等方法来区别，就很省事。

### Webservice 优缺点

优点:

1. 采用 xml 支持跨平台远程调用

2. 基于 http 的 soap 协议，可跨越防火墙
3. 支持面向对象开发
4. 有利于软件和数据的重用，实现松耦合.

缺点:

1. 效率不高（Poor performance）：由于 soap 是基于 xml 传输，本身使用 xml 传输一些无关的东西从而效率不高。随着 soap 协议的完善，soap 协议增加了许多内容，这样就导致了使用 soap 协议去完成简单的数据传输的效率更低了
2. 生产力低下（Low productivity）：不适用于 stand-alone aoolication，其实对于简单的接口如果直接使用 http 传输自定义数据内容比 webservice 开发更快捷。例如第三方支付公司的支持接口，例如支付宝的接口
3. 安全性。需要基于其他的机制，比如 HTTP 和 SSL

## Microservices & Serverless

微服务：

gateway 提供统一入口，负责把请求调用导过去；还有一个注册中心，把服务都注册在里面，用的时候在里面找。服务会将自己主动注册到注册中心里面，并且会定期请求以告诉它自己还活着。注册中心还会定期检查服务的情况，将有问题的及时移除。

具体实现可以在 ppt 里边看

## Security

微服务本身不在乎请求从哪里过来的，通过 GateWay 隔离之后请求没有差异

### Message Digest 消息摘要

为了避免信息被篡改，我们可以将原始消息的摘要和获取信息的摘要进行比较，若相同说明未曾改变；但即使不同也无法反推原始信息到底是什么

它有以下性质：

1. 长度固定
2. 一旦有任何一位发生改变，最终生成的内容会发生很大变化，因而无法伪造

## 消息签名

RSA 非对称加密算法，公钥加密只能用私钥解密，私钥加密只能用公钥解密

Authentication Problem：

- A 想把自己的公钥交给 B 进行交流，为了确保 A 是可信的，他们通过共同信任的 C 来进行担保
- A 把自己的公钥发给公共信任人 C，C 拿自己的私钥对 A 的公钥签名，然后把签名生成的摘要和 A 的公钥一起发给 B
- B 拿 C 的公钥对摘要进行解密，再拿解密的结果和收到的 A 的公钥进行比较，如果相同，则可信

私钥公钥这个体系的作用：

1. 用自己的私钥加密，表明自己的身份
2. 用对方的公钥加密，做数据保护

## https

建立了安全的 socket 层，s 表示通道是 SSL

浏览器生成一个随机对称秘钥，用证书加密后发送给服务器，服务器用自己私钥解密后就能用这个对称秘钥和浏览器直接进行加密通信

## Kerberos 协议

交大财务调用认证中心，认证中心有两个功能：

1. 认证：你是谁（AS）
2. 授权：你能干什么（TGS）Ticker Granting Server

认证授权部署在同一个服务器上，只做单点认证

Kerbos 协议的流程（PPT security 第 62 张的图）

1. 用户想要访问交大财务，现在有一个认证中心的服务器
2. 用户在客户端输入 UserId 和 Pwd
3. UserId 发送给 AS，Pwd 留在本地经过 hash 产生一个 key_user
4. 服务器段根据 UserId 找到一个 user，经过相同的 hash 算法产生一个 key_as
5. key 不能在网络上传输的，若 AS 用 key_as 加密一个东西（session key，一段时间内有效），user 可以拿自己的 key_user 解密，说明密码对了
6. AS 认证之后要进行授权，需要和 TGS 交互，不能明文交互。AS 产生一个 session key，通过刚才 AS 生成的 key 进行加密，用户拿到 session key 之后和 TGS 交互，但是是否相互信任？看下变的流程
7. TGS 用 TGS 的 pub_key 加密一个 session key 产生 message_b
8. user 不能解开 message_b，则原封不动传回去，把 session key 和 message_b 发回去
9. session key 用 TGS_pubkey 加密之后产生 message_c，把 UserId 和 timestamp 用 session key 加密之后产生 message_d
10. TGS 拿自己的 private key 解密得到 session key，再用 session key 解密 message_d 得到信息，知道你在什么时候想要访问
11. 接着 TGS 进入 TGS 的数据库查权限信息，有权限才接着进行下一步
12. TGS 会生成一个新的 key，用于 user 和交大财务交互
13. TGS 用 session key 加密上边说的 user 和交大财务交互的 key，称 message_f，传给 user，user 就可以用 session key 解开然后拿到 key
14. TGS 生成 message_e，用交大财务的 pub_key 加密 user 和交大财务交互的 key，同时附带 user client 的 IP 等信息
15. user 解密 message_f 之后，没有交大财务的 private key，所以 message_e 解不开，把 message_e 原封不动的发回来，交大财务解开之后能拿到 client 的信息和密钥
16. 用户再发给交大财务 client 一个 message_g，包含 client 信息，用 user 和交大财务交互的 key 加密。交大财务已经有了这个 key，因此可以拿到信息
17. 由于 TGS 是用交大财务的 pub_key 加密的，所以这个肯定是一个真的网站，不是一个钓鱼网站

18. 交大财务用 user 和交大财务交互的 key 加密每次 user 发来的时间戳+1，称为 message_h，client 可以检查这个时间戳对不对，以验证交大财务 server 对不对。

为什么叫单点认证？

AS 认证只做一次

局限性：

多跑几个 AS 和 TGS 分布式实例，不然就裂开

Kerberos 三头狗：

Client、AS/TGS、Service Server 相互制约

## 安全

常见攻击手段：

- 跨站脚本攻击：比如一个文本输入框里边输入脚本
- 注入式攻击：密码框输入必定为真的 sql 脚本
- 分布式拒绝服务攻击（DDOS）：脚本同时动作，发大量请求搞坏服务器

抵御用户的跨站脚本攻击：

1. 把脚本的符号转义掉
2. 富文本编辑器，当文本展示，不当脚本运行

防止注入式攻击：

1. 不要用 statement 拼 sql 字符串方式进行 JDBC 的访问

防止分布式拒绝服务攻击：

1. 同一个 IP 多少时间内限定访问（但是不同机器就不行了）
2. 做好访问检测

安全的手段：

1. 密文存储
2. 链路加密
3. 维护数据完整性（hash，checksum）
4. 代码有限暴露（尽量所有都是 private），封装接口，多个接口合为一个
5. 配置防火墙
6. 入侵检测系统（抓所有的包进行检测）
7. 恢复，打 ckpt
8. 日志工具

## MySQL Opt

**首先记住一点，数据库的东西在硬盘上存，访问时一定要 load 到 memory，内存比硬盘小，且存在数据不一致，这是一切复杂性的根源**

索引，数据库结构，表的结构，字段设置等

OLAP，在线分析处理，列存比较合适

OLTP，在线事务处理，行存比较合适

### 索引

索引建树的类型

- 索引优化：B+树
- 地理位置索引：R-tree，空间索引是一个地理位置的值，经纬度、海拔、投影值等等很多信息
- MEMORY table 支持哈希索引（hash 会占用空间）
- InnoDB 反向索引实现搜索引擎

索引的注意事项

- 多列索引的排序很重要
- 在查找时，索引文件 load 到内存里，再内存中搜索，如果索引文件很大，还得把当前这一页换出去，会有额外空间、时间的开销
- 数据量很小的时候建索引，效果提升不明显
- 索引太长，可以做前缀索引。
- 在批量操作，几乎操作所有数据时，索引没有太大意义
- InnoDB 在 disk 上存数据时是按照主键索引的顺序存储的
- 主键要保证绝对唯一，如果是多列联合主键，会增加复杂性

关于全局唯一标识符：

在服务器集群时，可以用全局唯一标识符例如 UUID，防止自增主键的重复，UUID 是 32 位 16 进制数，一共 16 字节。但是 UUID 占用空间比较大，而且不能知道插入的先后顺序，若需要时间信息，需要加入额外时间戳。

关于外键：

1. 给不常用的列可以进行分表，外键关联，便于每次向内存加载更多数据，每次不需要从硬盘加载大量数据。否则对常用列的操作性能会降低。
2. 主键要尽量简单，会用其他表的主键当外键

关于前缀索引：

MySQL 会对数据做压缩，根据存储方式的不同，前缀索引的长度可以不同

关于多列复合索引：

1. 最多支持 16 个列，没什么特别限制
2. 查询时匹配的顺序时建索引时的顺序，跳过第一列直接搜索第二列，索引就废了

3. JPA 使用数据库的速度可能没那么高，JPA 翻译的 sql 语句可能和预想的不同

关于 Hash 索引：

不能做范围查找，但是单个查找非常快，order by 是不行的

关于索引顺序：

- B+树索引，默认升序，也可以设置为倒序
- 在进行 Order By 操作时，如果和索引的顺序是相同的，就不需要排序，节省一点时间

## 数据库设计

数据尺寸：

- 数据量尽量小一点=>数据压缩
- 表的列
- 行的格式
- 索引
- Join 操作
- 范式化（衡量数据的冗余度，范式化越高数据冗余越小，但是性能可能会变差，因为外键操作特别多，join 操作也很多）

表的列：

- 适合它的最小类型
- 尽量不让列为 null，可能的话生成 not null，null 不利于索引，同时占用额外空间，还不利于编程

行结构：

- 可以对行进行一点压缩/Compact（Compact 不一定要用压缩算法去压缩，而是一种 Compact 的表示，在时序数据库中也有体现）

索引：

- 主键索引越短越好，因为外键要存
- 组合索引比多个单独索引要好，因为组合索引是一整个！插入操作不用调整很多个树
- 总的索引越少越好，否则增加插入开销

Join：

- 拆表做 join，减少冗余

范式化：

- 去冗余
- 数据同步问题

### 数据类型选择

- 能用数字，不用字符串
- 文本内容很多，不如存成 Text 或 Blob
- 行的尺寸有限制，小于 8k 可以用 varchar，否则一定要用 text 或者 blob，因为一页 16k（假设是 16k），带上其他数据之后，一行可能存不下
- 主键选择，当一行数据很多时，UUID 的空间占用是可以接受的，但是 UUID 之间无法比较先后顺序
- blob 可以设置懒加载，有关 blob 的表可以专门设置一台机器或者硬盘，和其他经常存取的数据区分开
- blob 的比较可以，可以比较摘要，长字符串比较效率很低

## MySQL 的设置相关

- table open cache
- 建表、建库的数量和文件系统有关，但是同时打开的表的数量与 cache 有关，超出上限会告诉你打开表的数量达到上限，暂时不能打开
- 打开的表是在后台统计的，通常比想象的多
- Tomcat 启动时会创建数据连接池，MySQL 为了使外部连接看起来统一，会为每个连接打开一张表，比如有四个并发会话连接，都打开 book table 时，就会打开四张表
- MySQL 可以设置最大连接数，最大打开表的数量。不能开太多，会占用内存，提升复杂性
- 表只有在 evict cache 或者 flush 时才会被认为关闭，类似于内存页的换进换出
- 内存满了会换出最近最少使用的表（替换策略可以换）
- 而在流量很大时，可能会突破内存限制，用完之后立刻关掉（比如，双十一时最近最少使用的表也只是在 100ms 以内，evict 掉很不合理）
- 当 Union 时，在内存中建一张临时表进行两张表的合并，合并之后在放到 disk，这也是**打开表数量超出预想的值**的原因之一
- 表的尺寸不能太大，受到文件系统文件大小的限制
- 表尺寸过大，可以分表，逻辑上可以 partition，或者物理上把文件数据分开放在分布式文件系统中
- MySQL 列数量有上限
- 行的尺寸也有限制，65535 bytes = 64k，但也不一定，每行的尺寸必须小于页尺寸的一半上限 65535 是设置 page size=128k 的情况
- varchar 有 2byte 表示占用长度，nullable 需要额外部分尺寸表示是否为空，这两种可能导致行尺寸超出容量限制

## InnoDB

- Optimize storage layout，类似文件碎片整理，还能增加局部性
- 事务 AutoCommit，可以多个写操作放在一个事务，关掉 AutoCommit，不必每次 flush
- long running tx 的 roll back，部分已经落盘，回滚有额外开销

- long running tx 应该拆分成小的 tx
- Cache 放大一些
- 当 tx 里边全部是 select，MySQL 会进行优化（开启只读事务，隔离等级不一样，no-locking select statement）
- rollback 要 redo、undo，buffer 开的大，可以记录操作结果，恢复很容易，比记录操作容易

导入大量数据：

- 关掉 AutoCommit
- data loading 有大量有 insert，若开启 AutoCommit，每次都要 flush，开销大
- 唯一索引约束插入时会检查唯一性，关掉 AutoCommit，会 load 在内存，内存中检查唯一性比硬盘检查更快
- 使用多行插入
- 整数递增主键间隔设置为 2，多线程插入防止冲突

优化 Query：

- 主键默认做索引，不要指定太多、太长的列
- 经常访问的列，建其他索引，而且是复合索引而不是多个单独索引
- 尽量声明 not null

优化 Disk I/O

- 内存不够必然在内存和硬盘换进换出，CPU 占用率就高；CPU 不忙时，没有频繁的换进换出，那这时怀疑 Disk I/O 是瓶颈
- 解决方案之一是扩大内存
- 还可以优化 flush 策略，减少 flush
- 还可以指定脏页超过阈值就 flush
- DMA 直接硬盘访问，跳过内存

- 换硬盘！整一个非旋转性的（比如 SSD）！
- 随机读取用 SSD 很快，而 HDD 就不太行，但是顺序读取时 SSD 和 HDD 性能其实差不多的。HDD 保存数据时间比 SSD 更长。
- 当有 500G 的 SSD 和 2TB 的 HDD，fusion storage 融合存储，设计一些维度，把不同访问模式的数据放在不同的存储介质上

预抓取 Read ahead：

- 缓存在从硬盘读一个表时，会把某一行数据所在的一小块 load 到缓存
- 后边也有可能接着在下一次读取，这个想法时 make sense 的
- 因此会预抓取，把下边的也读进来
- 不能一次预抓取太多了，要控制预抓取的数量，因为预抓取是基于判断的，猜错一次开销很大
- 预抓取的策略可以优化

删数据：

truncate table 而不是 delete，truncate 直接抹除

### InnoDB Buffer

table open cache：存的是 table handler

InnoDB buffer：存的是数据

buffer 很多块，每个之间是独立的，就是很多 buffer 实例

每一个实例会有多少内存

buffer 实例数 x 每个 buffer 的大小是总的大小，但是每块缓存的大小需要是 128M 的整数倍

为什么要有多个实例？提高并发性，不同线程不在同一个实例操作

必须有很大的内存才会分实例，否则不会分实例

访问数据时，首先去 table-open-cache 看能不能打开一个表，取 table handler

然后再去 InnoDB-buffer 找 data

buffer 如果放不下了需要清除缓存，就会使用 LRU 的方法换出一块数据

但是如果是请求 count 或者 sum，需要扫描全表，而且被换到内存之后，短时间又不会被访问，因此设置最热是不合理的

因此新的进来的数据不会是 hottest，而是 LRU 策略排序的 3/8 的位置

缓存读取数据时，会进行 optimize，减少碎片、增加有序性

既然数据已经是有序性的了，为什么不多读取一些？可以多读一点

随机的读一些数据，防止需要的时候才去硬盘找，减少开销

数据全部加载到 buffer，会有写操作的，经常把数据 write back 到硬盘上去

一旦 dirty 页的数量超过阈值，就触发 flush 的动作，整个 buffer 的脏页写到硬盘上去

写完之后，数据仍然保存在内存里，只是说把 bitmap 的 dirty bit 清除掉

多线程进行 flush，线程数可以设置，一般设置为 instance 的数量

可以将 memory 的数据周期性的写到硬盘，这样进行 warm-up 的时间就会减少，不用重新根据用户的读写重新慢慢 load，而是一次性直接 load 8G 进来（不一定是 8G，看 buffer 大小）

把 cache 分成多个区，指定哪些表只能缓存在哪些区（hot、warm、cold 等区）

hot 和 cold 都比较小，warm 最大

都是缓存，hot 和 cold 的 cache 有什么区别？

- 管理内存的方式应该是有差异的
- 存在 hot => warm => cold 的逐渐流动

加载时还要注意，索引加载到 cache 是加载所有的索引还是不加载叶子节点？

- 可以设置的
- 如果是 UUID，加载叶子节点，会占用很大空间
- 可以不加载叶子节点，这样加载的索引就非常多，省一半空间，虽然读取时需要再加载，但是缓存会省很多，可以放更多的索引

如果要 enlarge cache：

- 先写 dirty page
- 然后把所有的缓存都删掉
- 重新分配一次内存
- 这样会导致重新构建缓存，和重启没区别

Prepare Statement：

- 把已经解析编译好的 sql 语句存下来
- 以后再来一个相似的语句，直接用，比较快

trigger：

如果我在 user 表有一个 balance，但是在这一列上边有一个数据约束

当 insert 时会触发 trigger 检查是否符合 constraint

存储过程：

- client 通过 tomcat 访问 mysql
- 如果没有存储过程，全都要在 tomcat 里边 load 然后计算，这样会很占用内存
- 但是如果在 mysql 写存储过程，靠近数据源，传输数据量少

## Backup& Recovery

### *逻辑备份& 物理备份*

逻辑备份还是物理备份？

全量备份还是增量备份？

逻辑 & 物理：

物理备份：

拷贝文件出来，好处是数据库很大时备份出来的东西小，而且不用转换成脚本，速度快，还可以备份 log。但是可迁移性不好，必须是同一个版本的 mysql，还会对硬件有要求，而且导入数据时，必须关掉 mysql

逻辑备份：

生成脚本，可迁移性强，而且可以在运行时 recover，粒度可以细粒度控制，删掉一些行也是可以的。但是导出的文件大，而且生成脚本，导出速度慢

备份是 remote 还是 local？

如果 mysqldump 的方式生成脚本，是可以返回给服务器的

但是如果是生成特定格式分开的文件（比如逗号分割），是在数据库机器本地的

snapshot：

copy on write 的方式，存储的很少，没有一次性直接 copy

没有修改的数据是没有 copy 的，在以后恢复，是拿初始内容加上 snapshot 的内容拼接起来的数据

少量数据就可以记住某一个时间点的快照

快照是一个逻辑的副本，没有直接 copy，不然要存储爆炸了

### *全量备份& 增量备份*

全量 & 增量：

全量备份：

直接进行简单的拷贝

在做完全量备份的时候，之前的 bin log 都可以删掉了，不然太占用空间

做全量备份时，要先锁表，flush，文件全部一致才可以进行拷贝

增量备份：

写入 bin-log，存储的是 sql 语句

bin-log 可以进行压缩和加密，就可以减少空间，防止被别人看见

bin-log 也是可以导出的

### *Recovery*

在恢复时，找到某一个时刻的全量备份，load 到数据库中，然后重新执行 bin-log，就可以恢复到之前的状态

备份时可以在 slave 上边进行备份，这样 master 不用关

如果表坏了，可以用 repair 命令

设计全量备份的策略：

- 定期 dump 全部的数据到一个文件并且标时间
- dump 出来的是脚本而不是文件
- dump 是一个事务，all or nothing
- dump 要加锁

只要做了 dump，就可以删掉全部的 bin-log 了

只有单个数据库的 dump，不会带有 create 和 use 的命令，需要先手动建数据库

如果导出 tab 分割的文本文件，是可以用可视化工具打开的，比较直观

dump 也不单单是数据，还会把存储过程、函数、trigger 等 dump 出来

你可以选择是不是要把业务逻辑也 dump 出来

### *选择性地 Recover*

如何有选择性的执行 bin-log？假如有人运行了 drop database，我们可以跳过这条命令，执行这条命令之前的全部 bin-log，然后重新在执行这条 bin-log 之后的全部命令

但是如果是一条其他的命令，比如 update，即使跳过也不一定能恢复正常逻辑

### 关于存储过程和函数

如果存在业务逻辑，可以把业务逻辑放在 mysql，tomcat 只需要把参数发过来然后运行，这样减少 mysql 和 tomcat 之间数据发送的总量；但是这样对数据库迁移不是很友好，还有可能产生兼容性的问题

如果把业务逻辑存在 MySQL 里边，性能会比 Tomcat 高

但是在 MySQL 中编写业务逻辑，对于程序的编写不友好

如何选择？需要做 trade-off！

如果是大量的呈现式的业务，存在大量的数据传输，可以把逻辑写在 MySQL 中

否则尽量写在 Tomcat 服务器中

### Partition

如果一个表太大，大到放不下，需要进行分割，防止出现一个特别大的文件

MyISAM 不支持，只能用 InnoDB

sql 语句只支持分表，至于说是不是存在不同的机器上边，这个需要数据库管理系统实现

vertical 的切分需要用户自己做，水平的切割叫做分区

进行表的分区，一张表被分成了物理上不同的几个部分，删除某一个分区就会很快，甚至还可以指定对某一个分区进行查询

分区最重要的是分区的逻辑：

- 离散型，数据取值是离散的
- 连续型分区

按照条件分区，每个区之间不能有重叠

分区也不一定是要平衡的，分区的条件可以自己写

甚至可以写函数，按照函数计算的值进行分区

那么能不能在两列上边分区？（多个列）

可以！

多列分区，会按照每一列的条件进行判断，是按照 tuple 的比较

```
(1, 5) < (2, 3) # True
(2, 8) < (2, 6) # False
(1, 2) < (1, 2) # False
```

按索引依次比较对应元素
如不相等，则结果为元组比较的结果(如上第 1 行代码)
如相等，则比较下一对元素，直至有结果(如上第 2 行代码)
如所有对应元素都相等，则判为相等(如上第 3 行代码)

less than (5,12)

那么比较 (5,10) (5,11) (5,12) 前两个都是 true，第三个是 false

- 多列分区，会按照每一列的条件进行判断，是按照 tuple 的比较
- 还可以按照 LIST 分区，每一个 LIST 中的值在一个区。但是 LIST 分区，只承认在 LSIT 中出现的数据，如果找不到对应的，就会报错
- 还有哈希分区（hash partitioning）和线性哈希分区（linear hash partitioning），还支持自定义 hash 函数
- key partitioning 键分区

- subpartitioning 可以进一步细化

如果是 range 分区，null 的值对应的是 minvalue，一定会被分到最小的分区里边

如果是 list 分区，null 需要单独处理，任何一个 list 都没有 null 就会报错

如果是 hash 和 key，null 会把它的值当作 0 计算

把一个分区删掉之后，数据也就没有了

删除分区之后再去插入数据，插入的位置可能会改变

分区会做搜索剪枝的操作，搜索限制在小的范围内，筛选掉了大量无关数据

如果需要加新的 range 或 list 分区：

- 在后边追加分区，不能在中间加新的分区
- reorganize 才能在中间搞新的分区，其实就是整个重新组织了一遍

对于 hash 和 key 分区：

- 不能 drop
- 可以 merge
- 所有的数据都要留下来，但是分区的数量发生了变化，需要重新组织

T2 复制表 T1 的结构，但是删掉分区

T1 是分区的，可以把 T1 的某些分区交换到 T2，这样 T2 就有了数据

对 T2 在加入一些数据，再交换回去，就不行了

## NoSQL

NoSQL：not only sql

支持 schema 不严格的存储

大数据场景下，数据量大的问题：

- 数据能不能存的下
- 如果存的下，能不能快速读出来=>多个盘

100G 的 10 个硬盘，可以每个人的 100G 数据存到 10 个硬盘上边，每个 10G，这样等于说每个人的读写速率提升了 10 倍

数据组织成分布式存储，利用率没有降低，但是速度提高了

但是这种组织方式使得故障率提高，需要冗余来提高可靠性

当我们引入分布式存储，数据分布在多个硬盘上，每一部分处理完，合起来结果可能不是最终的结果

为什么在分布式场景下，关系型数据库不太行了？

- 数据太多，表太大的话性能不好
- 如果进行分表，或者分区，都会引起在多台机器、多个表之间的连接问题

NoSQL 支持非结构化、半结构化的数据，有没有一个字段和字段可以为空时完全不同的概念

### MongoDB

MongoDB 支持 auto-sharding，因为它都是键值对，不存在连接问题

document 是 MongoDB 的基本数据单位

document 的集合就是 collection，对应的是表

每一个 document 有一个 special key "_id"，在 collection 中是独一无二的

但是一个 collection 的 documents 可以有不同的 shape 和 type

document 是许多键值对，可以放很多键值对，但是键值对是有序的

document 的中的 key 大小写敏感，类型敏感，一个 document 中不能有相同的键

MongoDB 是键值对，可以是异构的，没有相同的 schema

可以查询某些 document 不存在而另一些 document 存在的的键

### *Sharding*

- MongoDB 拿很多服务器存储，每一个服务器叫做 shard server
- 有一个 config 记住 collection 被切成多少块，在哪些 server 上边
- MondoDB 的 collection 被分成了 chunk，每一个 collection 都可以被分布式地存在不同的 shard server 上边
- 可以根据 ID 分成不同的 chunk，chunk 存储在不同的 shard server 上边
- 分好范围之后，生成 document 是会生成 ID，就知道应该放在哪一个 chunk 上边
- 当 chunk 超过限制的大小，chunk 会分裂成两个
- 会保证所有 shard server 上边的 chunk 数量相差小于 2，一旦超过，就会对 chunk 做迁移
- 因此保证负载均衡，如果指定 chunk size 64MB，那么每一个 shard server 的负载差距不超过 128MB
- 但是可以关掉这个 chunk 的自动迁移，在特殊场景下是很有用的

## 图数据库

在拿节点和边描述数据的结构

关系型数据库的多对多很难真是描述，而且需要做很多表和表之间的连接

怎么存？查询语言？

节点上打标签，就表明了类型

节点之间的边表示了节点之间的关系

Querying Graphs：Cypher 语法

很复杂，摆烂了

搜索的过程就是按照边的结构特征进行遍历

关系型数据库是在做笛卡尔积，join 操作，巨慢无比

图数据库可以做协同过滤：Alice 喜欢一本书，看看喜欢这本书的人还喜欢什么，推荐给 Alice

Neo4j 可以变成 jar 跑在 Tomcat 里边，也可以单独跑一个服务器内嵌在 server 里边，嵌入式的。

每天启动一个，把数据存下来，不希望数据在一台服务器上，

### *图数据库的 Data Model*

第一个节点，下一个节点是谁，下一个属性是谁

类似于链表，用双向链表存储关系，比关系型数据库的 join 操作要快

### *Neo4j 应用*

数据清洗

Neo4j 切图，每一张图负责一部分

把图塞道神经网络里边

寻找洗钱网络是不能直接写 sql 语句的

需要构建集群，使用机器学习

### 混合存储的问题（行存储？列存储？）

行存适合 OLTP 场景：

- 保证加载一次就可以把数据 load 到内存里

列存适合 OLAP 场景：

- 经常需要拿到所有数据的某一个字段，这种情况下行存储需要把所有数据 load 到内存但是只取一个字段，有很大的 overhead
- 如果数据太大，可以分块存储，这在 HBase 中可以体现

当需要同时支持 OLTP 和 OLAP 时怎么搞？有两种策略：

1. 数据存储两份，一份行存，一份列存
2. 数据只存储一份，如果事务型操作多就行存储，如果分析型数据多就列存储

### B 树和 B+树

B Tree 和 B+ Tree 也是数据库的一种存储方式

B Tree 中间节点也可以存放值

B+ Tree 把数据全部放在叶子节点，只存 key 不存 value。而且 B+树的叶子节点有指向兄弟节点的指针，支持 Range Query

但是插入操作可能导致连续的分裂，有空间放大率

而且存在很多空洞，对内存造成浪费

### LSMT

是 kv-store，有增删改查的接口

分层结构，每一层是上一层的几倍大小

内存中有一层，读写都在是对内存的直接操作

为了防止崩溃，会有 WAL

内存有两块，一块写满之后，立刻切换到另一块，同时满的那一块进行 flush 到 L0

L0 的 key 是有重叠的，但是在进行 compaction 时进行多路归并排序

L1 之后的层都不能有 key 的重叠

bloom filter 可以加速查找过程

由于 LSMT 新数据在上层，因此读热数据性能比较好

空间放大率：

- 基本没有，数据禁止排列的

时间放大率：

- 读放大：读的时候，时间放大率会比较高，如果找的时不存在的数据，会一直读到最后一层
- 写放大：写的时候，可能每一层都满了，就需要进行连续的 compaction

有索引块，找到 offset，因此字符串可以任意长

为了节省空间，不是对任意一个 kv 都存储，它会进行优化，可能是找出 key 额公共前缀进行分组存储

CRC 循环校验码，如果按行存，每一行进行一个 CRC 的计算就可以

但是如果按列存，怎么知道每一行的数据是否是对的？

按列存，追加一些数据，CRC 是不是会改变？

按行存，key 有公共部分，可以存成一个，但是如果按列存，会比较复杂

### RocksDB

RocksDB 采用 LSMT 作为 Data Model

rocksDB 默认按照行存

大量写操作进来，高并发，低延迟

数据统计，读操作，低并发、高延迟

#### *rocksDB 写阻塞问题*

写流程：

优点：

- 低延迟插入

- 写入内存直接返回
- 后台进行异步的写入硬盘

缺点：

- L0 写满时将阻塞内存到磁盘的 flush 的过程
- L0 下沉，compaction 过程无法多任务执行
- 异步写，写放大严重，容易磁盘变成瓶颈
- 降低了 transaction processing 的可用性

优化：

- 中间层缓存数据（多加内存，满了立刻切换？）
- 负载均衡

*读放大问题*

需要访问所有可能的数据文件

解决方案：

- 新列存结构
- 在线混合存储决策

## 常见的列式存储

Row Group 格式

行列折中格式

常见的行列混合存储决策算法：

- 行列对等存储
- 行列差异存储
- 基于查询的分析

## 时序数据库

时间序列数据库：数据上带有时间戳

如果用关系型数据库

| id | timestamp | price | account |
|---|---|---|---|
| | xxxxxxxxxxxxxxx | | |

时间戳占用空间太大

时间戳不能当主键，可能会有同一时刻很多条记录

而且时间戳当主键导致索引太长

如果数据库内的数据过多，老旧的数据还要不要？存活多长时间？

如果是时序数据库，像是一个环形队列，满了之后会删掉最旧的数据

时序数据库的特点：

- 数据压缩之后，相比于关系型数据库可以存放更多的数据
- 数据都比较简单，没有 relation

### *InfluxDB*

- InfluxDB 内有很多 Bucket，每一个 bucket 是同一个结构
- bucket 第一列是时间戳，这是必须有的，时间非常精确
- measurement，即对数据做一下分类，对 bucket 中的数据分组之后描述一下是干嘛的
- tag，tag 上边会建索引。tag 也有 tag key 和 tag value
- field 有 key 和 value，field 上边不建索引。从 field set 是同一个时间点上采集到的所有数据的集合。采集到的数据一定是以 field 的方式存储，但是 field 不做索引，如果一个 field 经常使用，需要把它转成 tag

为什么 field 不能建立索引？

和数据的存储方式有关，field 这种存储方式没法建立索引

| 时序数据库中的 field | 关系型数据库 |
| --- | --- |
| 100 | 100 |
| +1 | 101 |
| -1 | 100 |

tag 和 field 之间是可以相互转换的

能不能有多个 tag？最好不要

比如既有 bees 又有 ants，但是某一时刻来的数据只有 bees，没有 ants，就没法放了。

他们其实不能放在一个 bucket 里边，放在两个 bucket 中是比较好的

所谓时间序列是指针对某一个相同的东西，只是时间上不同，他们构成一个序列。而把 ants 和 bees 放在一起是没有意义的

InfluxDB 和 RocksDB 类似，插入操作也是后台进行的，严格限制更新和删除

如果你进行删除和更新，就会停掉数据的插入，执行完 query 再接着插入数据（采样）

InfluxDB 如果数据向下落，就说明是老旧数据，会进行压缩；而内存中的数据是不会进行压缩的

为了有比较好的性能，最好 batch process

接收到数据后，组成一个 batch 再存入数据库，不要一条一条插入

一旦写入硬盘，旧的内存就被删掉了

InfluxDB 是以列存储的方式进行存储：

- 利于数据压缩（看上边的例子）

序列的 key 要做索引，key 是 measurement + tag + field，索引不需要那么多，只有这一种

时序数据库的数据特别多，数据量巨大：

- 支持分布式，可以切成 shard
- shard 针对的是落在硬盘上的数据
- measuerment 是逻辑上的东西，而 shard 是针对的物理上存储，分成不同的文件

默认情况下，只有一个 shard，可以自行指定 shard 的存活时间

可以指定新的 shard 不压缩，旧的数据 shard 进行压缩

不能为过去的数据做 shard

由于 InfluxDB 的数据时间戳是单调递增，因此数据不存在时间上的 overlap

### 总结一下 NoSQL

MongoDB

Neo4j

RocksDB & LevelDB

InfluxDB

Lucene & Solr & ElasticSearch

Dao 把多个 Repository 组合起来拿到数据，Entity，本质还是 TP（事务型处理）

如果要做 AP，那么我们需要 DataWarehouse

但是这么多数据库，怎么进行统一便捷的管理？DataLake

## 云原生数据库 & DataLake

### 云原生数据库

云原生数据库：未必是关系型数据库，还可以是非关系型数据库

云原生数据库，依托于云平台，结合了一些硬件设施进行优化

数据库在云上的特点：

- 虚拟化，资源的池化：目的是提高资源的利用率
- 资源耦合：将 CPU、内存、硬盘等资源解耦合每一层单独调用，crash 可以立刻从资源池寻找新的替代
- 弹性：即插即用，便于扩展
- 规模化
- 执行计划、分配
- 数据读写、共享存储

share nothing，通过网络传输

带来的问题：

TCP/IP 传输，会进行 packet 的封装，导致 CPU 负荷过大

现在出现了新的技术，不经过 OS、kernel，直接把包给到网卡，然后发出去，很快

网卡、路由器、交换机等网络设备需要支持 RDMA，通过高性能网络访问其他设备上的数据

## 计算下推

云原生还有一个特点：计算下推

云原生的计算节点和存储节点是分离的，缺点是存储节点的计算资源被浪费了

而且存储节点会把大量数据发送给计算节点，导致内部的带宽占用很大

如何利用存储节点的计算资源？

计算下推！

在进一步思考，能不能下推到存储介质上边？

总之没最终的目的就是，通过不断地计算下推，减少内部的数据流动，分摊压力，充分利用计算资源

可惜的是，上述的设计是没有统一的标准的，需要结合硬件资源和特殊的设计才可以完成，这也是云原生强大的特点

视图：

物化视图在 cache 里边，存一下元数据和哪一个物化视图最匹配

物化视图存在重复数据，可以进行一些集合操作，不用加载那么多

## 数据仓库

当业务变大，会产生其他系统

如果进行数据分析，可能对多个主题进行操作

那么如何对不同数据库的数据做一个拼接耦合？

1.
   - ETL：数据的清洗、转化、加载
   - 多个数据源的数据首先进行一次转换
   - 数据在加载时，可能需要进行某些语义、数值、类型的处理
   - 数据清洗，删除 dirty data
2.
   - 数据建模，按照 OLAP 的需求进行建模
3.
   - 进行分析、可视化

在数据仓库中，主要针对的时 OLAP 的处理，就是为了做分析的。数据仓库的数据源是非常多的

shard server 的模式是 share nothing，并行处理性能很好，但是一旦涉及到互相之间的交互，就寄了

### 数据湖

数据湖中的数据时存在各种格式的，不需要做 ETL

进行元数据管理

进行分析时在把数据从数据湖抽取出来，导入到数据仓库

### 湖仓一体

当所有数据都被数据仓库用到时，就等于说数据湖全部数据被复制了一遍，等于多做了一次复制

因此有人提出湖仓一体的概念

而且根据数据的类型，在进入时就可以决定放在数据湖还是数据仓库

### 对比数据仓库和数据湖

| 数据仓库 | 数据湖 |
| --- | --- |
| 数据体系严格，提前建模（OLAP） | 数据体系松散，事后建模 |
| 灵活性较低 | 灵活性较高 |
| 数据治理容易 | 数据治理困难 |
| 数据种类单一（结构化、半结构化） | 数据种类丰富（结构化、半结构化、非结构化） |
| 面向成熟数据的企业级分析与处理 | 面向异构数据的科学探查与价值挖掘 |
| 向特定引擎开放，易获得高度优化 | 向所有引擎开放，各个引擎有限优化 |

## Clustering

### 会话粘滞性

load-balance 策略：

考虑连接数

最简单的是 round-robin 轮询

可以改变 weight

如果是按照 IP 的 hash，可以解决会话粘滞性的问题，但是不能做到负载均衡

解决会话粘滞性的问题还可以使用第三方工具存储 session

比如，用 redis 存储 session，这样集群服务器都会向 redis 取 session

- 缺陷是 redis 单点故障：可以备份
- 向 redis 取 session 会稍微慢一点

### 数据库集群

MySQL 主从备份

至少有三个节点

MySQL 集群的作用：

- 可以做数据备份，如果主节点炸了，不至于数据全都没有
- 从节点可以分担读请求，但是写请求还是只能在主节点上边响应
- 设置好种子，集群可以动态增加

主从备份：

- cold：只写一个，每隔一段时间进行数据同步
- warm：每隔一小时更换主从节点的角色，进行切换
- hot：拿到请求，各处理各的，同时进行

还可以是多个 primary

## 虚拟化 Virtualization

JVM 把 bytecode 翻译成机器可以听懂的东西，Java 可移植性比较好

VM：对 CPU 和内存做直接的虚拟化，对 IO 操作经过特殊的虚拟机转化为对物理机的操作

Image 文件切分，如果 Image 文件有相同的部分，则可以对相同的部分进行复用

Docker+K8S

KVM+Open Stack

监控的数据都放在时间序列数据库中

虚拟机运行的是完整的操作系统。虚拟机的目的是做隔离，但是跑完整的操作系统占用空间太大了

docker 是把对 linux 的的调用转换为对物理机操作系统的调用

containr 是一个运行时。容器就是把你写的代码和需要的东西全部打包运行

docker 是分层打包的

docker 的 image 没有包含完整的操作系统

拿一个 Image 可以运行多个实例，App 的系统调用发送给 docker 引擎，转换为 host machine 的系统调用

容器互相之间是隔离的，每个容器都有自己独立的 IP

可以通过 IP 加端口访问 container，也可以通过端口映射访问 container

一个 docker 不能直接访问另一个 docker 的内容，必须通过网络进行访问

*（这一段是在讲什么啊我焯）*

*java 代码需要 JVM 来加载，JVM 是用 java 写的*

*最初的 ClassLoader 是用 C 写的*

*在打包时，不应该把 MySQL 驱动放在/lib/ext 中，不要直接打包*

*ClassLoader 也是一种隔离*

docker run 的一些参数

- -d 后台运行，守护进程

- -p 端口映射 当前机器的 xxxx 端口映射道 docker container 的 yyyy 端口

内容持久化 container 之间的内容是完全隔离的

写入 container 的文件，重新启动一个 ubuntu 的实例，是没有之前写入的文件的，不会保存在 Image 中

能否持久化？Volume！

可以将 volume mount 到 container 中（文件映射）

通常 volumn 是一个文件，但是：

- 挂载数据库？
- 远程文件？

是可以挂载远程目录的

同一个 Image 在跑不同的实例可以挂载不同的 volume，就不会相互干扰

## Docker & K8S

docker 的 image 打包了一些依赖的库，将自己的应用打包进去，变成一个容器，交给 Docker Engine 运行

container 的调用全部交给 docker engine，docker engine 转换为对 host OS 的调用

是否可以把 NodeJs 和 MySQL 搞成一个缝合怪？

不太好，一个崩了另一个也崩了

而且，Image 是分层的，即使是不同的 Image，也有可能是某些层是完全相同的，在 pull 的时候已经有了分层，再缝合不太好

挂一个卷进去，当一个容器销毁的时候，它的数据可以在本地存着，不会顺带销毁

### Docker Compose

每次启动两个容器，好复杂，能不能有方便的方法？

Compose！

需要写 Compose 文件，放在目录里边

### K8S

docker 有一个运行时，可以把 docker 的底层运行时换掉，换成更加轻量级的

K8S 是用 GO 语言写的，如果想魔改，需要用 GO 语言编写

在一个集群里边，如果每一个都可以跑后端，如果发现 container 崩了，需要人为进行重启。

有没有什么自动化的东西，可以进行自动部署和迁移？

这一系列的东西使用 K8S 可以搞！

K8S：

- 服务发现和负载均衡
  - 所有的容器在 K8S 注册一下，还可以进行负载均衡
  - 因为位置是主动注册的，K8S 知道容器在什么位置，无论 IP、端口怎么变化，都可以进行负载均衡（因为是按照名字找的）
  - 一个崩了，可以进行迁移，这个过程用户是无感的
- 还能进行存储编排
  - 不同的容器可以挂载不同的卷，进行存储编排
- 状态回滚
  - 可以检查状态是不是正常
- 部署，背包问题
- 自愈
  - 可以关掉，再启动一个

- 安全问题

控制面：要完成的所有功能

控制面包含的构建很多，建议把控制面装在一台机器上，不要和容器节点放在一起

- API-server：类似 gateway，因为它可能是系统瓶颈，可以集群化部署
- 调度器：判断容器如何分配节点，调度器可以跑多个实例
- ETCD：状态存储库，不能是集群，只能是一个地方完整地存储
- 容器管理器：
- 云相关的容器管理器

Node：容器节点

- 在每一个节点上都要安装一个 kubelet 小程序，监控节点的状态
- 节点上还要有 k-proxy，使节点在同一个网络内互通

不能直接房屋内 node，需要访问 api-server

pod 是一个 server+mysql，这是部署的一个实例

pod 是部署的最小粒度，容器的最小单元，其中的 server 和 mysql 不能拆开

## 云计算

云计算对应了边缘计算，所以有人想把边缘计算叫做雾计算

云计算需要进行

- 操作系统的调度
- 分布式文件系统
- 数据库
- 数据库上的事务管理

这些和单机版本有什么区别？

云计算之前有一种叫做网格计算：

- 所有人计算资源共享
- 没有得到人们的响应

云计算：

- IBM 看见了商机：厂商有大量资源，拿出来给别人用，这些机器看不见，就好像在云上
- 其实就是把集群共享出来，供别人使用
- 暴露的是服务，web 服务，这些服务都不在本地，而是通过互联网去访问的
- 云服务是按需计费、弹性拓展的
- 云服务的所有管理上的复杂性都是云服务厂商管理的

需求不同，如何应对复杂的、不同的需求？

- 需要进行**虚拟化**，服务器全部装同样操作系统，操作系统上边运行不同的虚拟机
- 哪怕规定只能用 linux 操作系统，也必须虚拟化
- 原因是，为了满足需求，云服务厂商购买了 16 核 100G 内存的高性能服务器，而如果用户只要 2 核 8G 内存，需要进行资源分配，这也需要通过虚拟化实现
- 虚拟机之间是隔离的，用户看不到其他的虚拟机
- 像 JDBC 一样，少量的资源能干很多事情，原因是用户 A 租了一台云服务器，但是这台服务器在不忙的时候会被资源回收，资源分给其他的服务器使用

云服务的特点：

- 弹性定价策略

- 弹性拓展，能用 eureka 进行服务注册
- 快速供给，我买了之后一分钟给我
- 虚拟化

云服务的必要支持：

- 负载优化
  - 监控机器的占用情况，把服务全部迁移到一部分机器上边
  - CPU 调频，节约电
- 整体的服务控制、调度
- 部署选择
  - 共有云还是私有云

## 云计算的核心技术

核心技术

- 调度系统：代表性的是 MapReduce
- 分布式文件系统：代表性的 GFS
- 分布式存储系统：代表性的 BigTable
- 内存管理：有 Xen 等

MapReduce：

- mapper 处理数据，达到阈值之后写文件
- shuffle 把 mapper 产生的数据进行排序和分组（把每个 mapper 的文件变为每个 reducer 一个，便于 reducer 处理）
- MapReduce 是有大量硬盘读写的系统

Spark：

- 由于 MapReduce 磁盘 IO 很多，速度很慢，Spark 就想能不能把操作都放在内存里边运行
- Spark 的要求：首先需要内存大一点
- 其次 Java 就不行了（Java 太吃内存），换成 Scala，而且进行函数式编程

边缘计算：

- 雾计算，在身边的
- 云计算，放在远端、看不见的
- 软件学院摄像头的例子，通过摄像头的计算资源进行匹配
- 把计算推到网络的边缘，从终端到云，在任何一个节点都可以做处理
  - 数据永远在靠近数据的地方处理是最快的
  - 移动网络是不可靠的网络，不保证一直是持续的

下述讨论的是面临移动网络下的场景，最大的问题是数据传输的速度，移动网络（主要设备是手机）带宽是非常不稳定的。而且在移动，可能刚刚连接上第一个基站，这个基站开始运行任务；移动至第二个基站时，第一个基站就没法连接上了。运营商需要做虚拟机的迁移，把第一个基站的虚拟机迁移到第二个上边。

边缘计算解决的问题：

- 数据的处理要靠近数据的源头，就近存储、就近处理
- 移动网络，不可靠的网络连接，可能不能连接到云端，只能连接到基站

任务的本地执行、部分卸载、完全卸载（offloading）

- 本地能不能做？不能做就丢到上一层
- 能做一部分，可以把一部分丢到上边
- 不至于从小基站到宏基站一条路带宽全部占满

## Hadoop

核心是要跑一个分布式文件系统，后台有一个 yarn 调度这个分布式文件系统

combiner 只处理一个 mapper 上边的输出，将一个 mapper 里边的数据进行初步的合并。

那么还要不要 reducer 吗？需要的，因为 reducer 可能收集来自很多个 mapper 的数据，combiner 只是减少数据冗余，做一个初步的处理

map 的数量和文件数量、大小有关，是通过 Hadoop 的框架进行调度的

combiner 是复用的 reducer 的代码，相当于是在本地做的 reduce，combiner 的数量和 mapper 有关

reducer 的数量是可以人为控制的，如果不设置，会根据 CPU 核数和任务的情况进行设置

如何扩展数据：

- 如果输入文件太大，MapReduce 会把文件切分成固定尺寸的 split，一般是文件系统 block size 的大小，HDFS 中的 split size 默认是 64MB
- 每一个 split 会分配一个 map 任务
- spilt 尺寸要合适，如果太小了，要做 map 太多次；如果太大了， 读的效率不一定搞
- Reducer 也可以是多个，每一个 Reducer 只负责一部分

### 关于 Job 和 Task 的重要概念

Job 和 Task：

hadoop 的一个作业称为 job，job 里面分为 map task 和 reduce task，每个 task 都是在自己的进程中运行的，当 task 结束时，进程也会结束

Job tracker 的任务

1. 接受所有提交上来的任务

2. 看集群里边哪些机器是空的，然后给空的机器启动 task，起到任务管理器的作用，管理哪些程序应该跑在哪些机器上
3. 还要跟踪每一个 task 是不是崩了，如果崩了要在另外的地方另外启动一个
4. Job Tracker 在系统中只有一个，同时承担的任务过于重要，如果 Job Tracker 崩了，系统就完蛋了

Task Tracker：

1. Task Tracker 是每个机器上都有的一个部分，他做的事情主要是监视自己所在机器的资源情况
2. TaskTracker 同时监视当前机器的 tasks 运行状况，TaskTracker 需要把这些信息通过 heartbeat 发送给 JobTracker，JobTracker 会搜集这些信息以给新提交的 job 分配运行在哪些机器上

- 一台机器上一般可以同时启动 10-100 个 Mapper，不建议更多是因为机器上边的进程太多会导致进程调度困难
- MapReduce 需要做一些任务的决策，考虑如何切分文件、分配任务，因此在数据量少的时候，额外开销是比较昂贵的
- reducer 的数量：$number\ of\ nodes \times number\ of\ containers\ per\ node \times weight$，weight 在 0.95 到 1.75 之间
- map 主要设置是 spill 文件，内存满了就 spill 写到硬盘
- map 把中间结果通过网络发送给 reduce，为了减少尺寸，还需要进行压缩
- 还可以设置每 spill 多少应该进行一次 spill 文件的合并等
- Job Tracker

### Yarn

Yarn 把资源管理器和任务管理器分开

每一个 Job 都有一个 master，管理自己的 mapper 和 reducer，就把应用的管理和资源管理分开。

client 请求 ResourceManager，每一个机器上都有 NodeManager 管理资源，并且将硬件资源的情况告诉 Resource

而 MapReduce 的任务管理会针对每一个单独的 Job 启动一个 ApplicationMaster，每一个 AppMaster 管理自己的 Mapper 的 Reducer，当 AppMaster 收到情况之后，将消息传回给 ResourceManager

ResourceManager 也会根据硬件资源的占用情况，选择在适当的机器上启动 AppMaster 和 Container（container 就是 mapper 或者 reducer）

## Spark

MapReduce 频繁地读写硬盘

读一个 split，写满内存将 spill 文件写到硬盘。处理完要将小文件合并、排序，再传到 Reducer 上边

Spark 就是为了解决 MapReduce 的这个缺点

几个主要的功能：

- 建立在 HDFS 之上，可以处理批数据/流数据
  - 流数据也是通过切分时间片的方式进行处理的
- 支持 Spark SQL，但是底层不一定是结构化数据
- 大数据科学
- 机器学习库

特点：

- 速度快，所有操作都在内存中完成
- 可以集成分布式文件系统

RDD：弹性分布式数据集

每读一行，都是 RDD 的一个元素，但是 RDD 可以在几个机器上分布（在内存里），逻辑上是同一个

Spark 的几个特点：

- 惰性操作，用到才会执行具体的 transformation 语句，产生新的 RDD，非万不得已不要执行，否则占用内存
- cache 操作：由于内存满了会根据 LRU 进行内存替换，t 掉旧的内存，如果用了 cache，说明这个 message 很重要，不会参与内存的替换
- cache 只是一个想法，告诉 Spark 这部分数据需要 cache 住。如果 cache 过多，会进行一个压缩，如果压缩都不行，就放在硬盘上

在 Spark 上做的所有操作分为几类：

- transformation：变换，输入时 RDD 输出也是 RDD，对 RDD 进行变换生成一个新的 RDD，但是不会修改原来的 RDD，而且不会立刻运行
  - map filter 等
  - transformation 可以在本地做，每一块只依赖于之前的一块的 RDD，不需要和其他的进行交互
- action：立刻执行，不会等待，返回值不是 RDD，会形成 DAG 图提交 Spark 集群运行，并立即返回结果
  - 依赖于多个块，不能在本地做

Spark 的工作流程

- 所有的操作都是从 Driver Program 中的 SparkContext 开始的
- 将任务提交给 Driver Program 之后，SparkContext 和 Cluster Manager 沟通，分给合适的 Worker Node 执行
- 每一个 Worker Node 都有一个 Executor 的进程，执行一些 task，并将执行完的将结果给 SparkContext
- Cluster Manager 负责监控的作用，监控 Worker Node 的情况等。

文件是分布在不同的机器上，WorkerNode 最好靠的比较近，而且要求 Worker Node 和 Driver Program 的网络是相连的

写好的代码不是在 IDE 里边直接跑，而是用过 Spark 的 submit 命令，把写好的东西扔到一个节点上跑

### RDD

- 可以并行处理的数据集
- 可以存在多个机器上，逻辑上是同一个，物理上可以分布式存储在内存中
- 容错性好，可以恢复
- RDD 可以从现成的 RDD 产生或者从文件中读取
- RDD 加载之后就需要进行分区
- RDD 可以进行 transformation 和 action，区别在上边说过
- 可以持久化（指的不是存到硬盘上，而是在 cache 里边持久存储）

partition：

- 如果没有人为控制，就需要问每一个机器这块数据你有没有，是很大的开销
- 如果做了 partition，只需要找特定的机器就可以拿到数据

### 宽依赖/窄依赖

宽依赖以及窄依赖：

- ppt 有例子
- 窄依赖：父 RDD 的每个分区只被子 RDD 的一个分区所使用
- 宽依赖：父 RDD 的每个分区都可能被多个子 RDD 分区所使用

宽依赖可能带来很多网络开销，如果一个 RDD 崩了，需要前边父 RDD 的所有分区才能生成，代价很高；如果是窄依赖，不需要很大开销

宽依赖往往对应着 shuffle 操作，需要在运行过程中将同一个父 RDD 的分区传入到不同的子 RDD 分区中，中间可能涉及多个节点之间的数据传输；而窄依赖的每个父 RDD 的分区只会传入到一个子 RDD 分区中，通常可以在一个节点内完成转换。

stage：

stage 的切割规则：从后往前，遇到宽依赖就切割 stage

调度时按照 stage 执行的，可以将 stage 的结果缓存下来，不用一直回推到最开始。遇到宽依赖不得不运行时，才去做一个 stage

on heap 和 off heap

- on-heap：由 Garbage Collector 管理的 Memory，会频繁地进行 GC
- off-heap：被 OS 管理的 Memory

使用 OFF_HEAP 的优点：在内存有限时，可以减少频繁 GC 及不必要的内存消耗（减少内存的使用），提升程序性能

Spark SQL：

专门处理结构化数据，可以看作一个内存数据库，默认按照列存储

流式数据处理：

来了之后就插入到表中，像是一个没有边界的表，通过划分细粒度的时间片，把批处理变成类似流处理，每个细分的时间片内仍然是批处理

## Storm

专门拿来做流式数据处理

由 spout 和 bolt 构成一张有向无环图 DAG

处理的是没有边界的流数据

可拓展的（storm 本身做不了，需要 zookeeper 支持拓展性、容错性）

zookeeper 管理集群中的每一个节点（资源控制）

master 在 Storm 中被叫做 Nimbus

zookeeper 怎么直到 worker 可用？和 worker 机器上的 supervisor 通过网络连接，supervisor 将 worker 的信息告诉 zookeeper

分布式环境最好把资源控制和任务调度分开，参考 Hadoop 的 Yarn

Topology：

Storm 中运行的一个实时应用程序的名称，将 Spout、 Bolt 整合起来的拓扑图。定义了 Spout 和 Bolt 的结合关系、并发数量、配置等等

Spout：

在一个 topology 中获取源数据流的组件。通常情况下 spout 会从外部数据源中读取数据，然后转换为 topology 内部的源数据。

Bolt：

接受数据然后执行处理的组件,用户可以在其中执行自己想要的操作。

Tuple：

一次消息传递的基本单元，理解为一组消息就是一个 Tuple。

Stream：

Tuple 的集合。表示数据的流向。

构建 DAG 的过程，有点类似于 PyTorch 里边构建神经网络的例子（就像搭积木）

## HDFS

### 一些基本概念

HDFS 不是从头开始搭建的，底层是 linux 文件系统

面向尺寸特别大的文件，一个大文件切分成 HDFS 的 block size 放在不同的机器上，不同机器并行读，就相当于增加了带宽；如果是小文件，只能在一个机器上，而且需要 HDFS 这一层直到文件位置再去向 linux 文件系统取文件，导致速度降低。

HDFS 面向的应用场景是写入一次，多次读取的方式访问。如果是插入、更新，导致每一个 block 都要修改，开销巨大，而后边追加的操作时可以的。这种文件系统的应用场景很适合时序数据库

由于底层的机器是廉价机器，随时会崩溃，因此为了容错，数据需要有 replica

Name Node 和 client 交互，告诉 client 文件具体在哪一个 Data Node 上边，client 直接和 Data Node 交互，client 和 Name Node 只交互一次

当有大量的 block 存到一个 Name Node 上边，文件的路径是/hdfs/1.txt，但是当它向 Data Node 存储时，可能分块存储成为/hdfs/1-1.txt 和/hdfs/1-2.txt，Data Node 并不知道自己存的什么，也不知道自己存的文件的名字

Data Node 会自己建一个目录，再去找具体的文件，这个目录和 HDFS 的目录没有任何关系，是内部自己组织数据的一种方式，是 user-invisible 的

一般来说 Name Node 单独用一台机器来跑，因为 Name Node 上边存了元数据，需要它的负载轻一些

HDFS 也支持用户配额，限定用户最多使用多少数据

不支持链接和软链接

Name Node 中记录的数据：

- 文件的名字
- 文件 replica 数量，不一定是全部相同的，可以指定副本因子
- 多少的 block，id 是什么，分布在哪些机器上

HDFS 删除文件是先放到垃圾箱，过一定时间才彻底删除，因此刚刚删除文件时占用的空间并不会减

- Data Node 发心告诉 Name Node 自己还或者
- 如果 Data Node 死了，就不在向他转发请求
- 会将死亡的 Data Node 上边的 block 进行拷贝，满足副本因子的要求

### 心跳

Block Report 和心跳要分开，因为 Block Report 没必要太频繁

心跳不能太严格，如果在一定时间崩了才会判定死亡

为什么不能太严格：

- 如果是定时器，Timer 是多线程，Timer 线程正常不代表读写线程是正常的
- 需要用 while 循环，和业务在一个线程，业务死了，心跳肯定发不出去。但是如果这样，时间就不一定准，因此心跳不能太严格

### 机架感知

如何防止副本？一定要在不同的机架防止，提高容错

同时，如果距离太远，在 Data Node 写时，网络开销很大，需要做一些平衡

每一个机架的 replica 数量也有上限

### 安全模式

HDFS 在启动时会检查 block 是否满足副本因子的要求，如果不满足会进行 replica

### 元数据

edit log 不能存在 HDFS 里边，因为是要靠 edit log 保证数据的原子性，因此 edit log 放在 OS 的 file system 中

FSImage 也在 OS 的 local File System 中，FSImage 存的是元数据

来了写操作，首先写 edit log，写完 edit log 再写 metadata

checkpoint 在 FSImage 更新，edit log 清空之后

### Failure

如果是 Data Nodeheartbeat 断了，过了指定时间，这个 Data Node 就被标记为死亡，在这上边的 replica 都无效了，这上边的副本因子就清空

如果说某些 replica 的 checksum 不符合标准，则 Name Node 就会提高副本因子，再做一遍 replica，保证 replica 数量足够

数据完整性是通过 checksum 进行检验的，checksum 不能和文件数据存放在一起，而是单独放在隐藏文件中

如果 edit log 和 FSImage 坏了，那么整个文件系统就不行了，因此它们也需要多个副本。这些副本不能放在 HDFS 中，而是应该放在本地文件系统中

### 文件删除

把文件放在 trash 目录中

目录移动也比较简单，因为 Name Node 不存放数据，改目录也只是改元数据，而且目录也只是一个逻辑概念，不是真正移动

因此文件删除这个操作非常快

调副本因子时，也只是把一些 block 放到 trash 中，这是由 Name Node 进行选择的

## HBase

关键词：

- 大数据存储
- 分布式
- 带版本
- 面向列的
- 非关系型数据库，schema 不严格

### 基本概念

HBase 面向列的，有列族的概念

每一个 cell 都有时间戳，带有版本的概念，如果查找是没有特殊要求，默认返回最新版本的值

在硬盘上按照 region 的模式存，一张表的一些行作为一个 region，但是存储模式是按照列存储，由于 region 的模式，可扩展性很好，一个 region 满了再开一个 region，一个 region server 满了再开一个 region server

按列存储，可以更改编码机制，更加省空间，压缩机制上，可以进一步压缩

按照 id 进行排序，然后将排序后的数据存放在不同的 region 中

HBase 没有库的概念，取而代之的是 Namespace

其次就是行和列的概念

列被组织成了列族，存储的是相关的列

在设计 HBase 的表时，需要改变关系型数据库的设计思路，把原来的整个关系型数据库数据库存成一张表，把原来觉得比较内聚、在关系型数据库总是一张表的数据放在一个列族中。

列族内部的数据是存在一起的，列族之间是分开的

HBase 会自动进行表的切分，当一张表大到一定程度会自动切开，水平切分产生另外一个 region

HBase 表的删除需要先 disable 掉，否则不能删除

建议列族不要超过三个，一旦列族太多，需要记住在一个 region 中列族在什么位置，列族多了 index 就很长

```
put   'test', 'row1', 'cf:a', 'value1'
put   'test', 'row2', 'cf:b', 'value1'
put   'test', 'row3', 'cf:c', 'value1'
```

| | cf:a | cf:b | cf:c |
|---|---|---|---|
| row1 | value1 | | |
| row2 | | value2 | |
| row3 | | | value3 |

稀疏的表，而且是动态插入，一开始并没有这三行

HBase 物理上的存储：

row key + time stamp + content

版本号可以指定最多存储几个，在取数据时，如果不指定，默认取最新的数据

有一些关键点：

- region 尺寸，不要太小，不到一个 block 很浪费，太大一台 server 放不下
- cell 不要太大
- 列族最好一到三个
- region 的数量最好在 50-100
- 列族名字短一点，因为列族要参与索引，长了占用空间
- 如果是时序数据，主键设计比较重要
- 操作一张表的时候只允许有一个列族正在运行

## Hive

Hive 数据仓库

用了很多数据库：

- MySQL
- MongoDB
- Neo4j
- InfluxDB
- RocksDB
- ……

从业务需求，可能需要把所有数据放在一起做分析，需要导入数据仓库

Hive 里边用 Ctrl+A 把数据断开，或者指定几类格式文件，只要复合规定就可以导入数据仓库

导入数据仓库之后，变成一张一张的表，导入之后对 DataWarehouse 就没有任何区别了

导入数据仓库要经过数据的过滤、清洗、转换、加载的过程

如果是 schema on write，在写入的时候校验数据是否符合格式，这样要过很久才能告诉 client 是否加载成功，而且这张表甚至不会使用，因此数据仓库采用 Schema on write 开销比较大

## 数据仓库的一些讨论

但是什么时候做数据过滤清洗、转换、加载的过程？

- 数据仓库是 Schema on read，只有在使用数据时才会进行数据的校验
  - 数据仓库在 load 数据之后，原始数据还在原来的库里，处于分离的状态，就没有原始的数据了（原始的格式没了，在数据仓库上边是去掉了原始格式的一张一张的表），而数据湖存储的时原始的数据（原始格式）
- 如果只使用数据湖：可以将数据湖的各种原始数据搭配上 SQL 的转换器，这样就将数据湖当作数据仓库使用了，不需要数据仓库了，但是性能必然没有数据仓库好
- 湖仓一体（LakeHouse）：
  - 热数据在数据仓库，冷数据在数据湖，如果数据湖中的数据被多次访问，就放到数据仓库中，这种设想是 make sense 的

## Hive

目标是做大数据的分析，是 Hadoop 的一个组件，底层是 HDFS

数据加载进 Hive 中之后，让熟悉 SQL 的人可以用类 SQL 的语句操控这些数据进行大数据分析

类 SQL 语句是用 MapReduce 运行的

/user/hive/warehouse 用于存储元数据（表的结构等……）

/tmp 目录用于存储临时文件，因为 load 数据要进行一些过滤、清洗、转换，会有中间数据生成

Hive 里边按某一列做 Partition

Hive 加载数据，采用 MapReduce 的方案，同样地 Hive 几乎所有的 SQL 都是 MapReduce 的模式执行的

频繁初始化 HDFS 会造成 Name Node 和 Data Node 不一致

Hive 支持的数据格式包括纯文本文件、Parquet、ORC 等格式，因此大家可以把数据都导入进来

RCFile：

先水平切成很多块，而对于每一块，是按列存储的。数据从数据库中转换为 RCFile，从而 load 到数据仓库中

还有 Parquet，也是数据库转成 Parquet 在导入 Hive 中

这就是为什么 Hive 使用 schema on read，不然 laod 太慢了

Hive 还可以直接加载压缩文件到某一个表中，等待以后实际用到才会解压缩，这样子空间占用会比较小，但是由于原始数据没有还原出来，不能进行数据的切分，就只能有一个 Map 和一个 Reduce，不能并行操作，时间占用比较大

## Put them all together

A 问 B 请求，不想要数据的结构，想要的是高级信息，比如今年的业绩 blabla

烟囱：各个部门、公司之间孤立的数据系统

因此要构建数据中台：

- 聚合、治理跨域的数据，封装成服务，供前台业务调用
- 企业内外部异构的数据采集、治理、建模、分析、应用，使数据对内优化体改业务，对外可以使数据合作价值释放，成为企业资产管理中枢

层次划分：

1. 数据层：高压数据流，数据接入接口
2. data hub 数据集成
3. 计算层：数据计算引擎（比如 AI 模型等），大数据平台（允许拿到集成完的数据），云平台数据库
4. 服务层：搞搞服务什么的

数据中心建设思路：

1. 数据清洗与转换
2. 数据弹性存储
3. 数据融合（基于并行处理基础设施）
4. 面向应用的数据服务（最终给用户暴露的东西）

key points：

1. **统一时空**基准下的数据融合（时间：统一授时，空间：空间基准相同，就好像是对空间的描述）
2. 面向 xxxx 的数据架构（物理实体：船，虚拟实体：航线），搞搞“数字孪生”
3. 精细化的数据治理（统一数据表征）
4. 弹性存储与处理计算基础设施

- 数字化（把东西存在电脑里），数据化（做过处理，能够进行一些操作）
- 物理世界的数字化
- 数字资产化、资产数字化赋能平台

## 总结一下！

*服务器端开发*

- 大二的时候，tomcat 访问 mysql，解决了结构化数据的存储
- tomcat 是 java 的一个进程，要控制实例数量

- Scope 控制实例数量，实质是节约内存资源
- tomcat 访问 mysql，一般是同步调用，导致阻塞。大量请求，可以改用异步的通信方式：发消息。记下来之后说我收到了，事后处理。但是事后处理的前提是，有空闲时候，可以做到削峰填谷。如果一直很忙，全是峰，就寄了。

*事务*

事务用描述性的方式

requires_new 不是嵌套事务，根本不是一回事

同步，用锁

wait-notify

cache

异构，WebService，SOAP

WebService 里边为什么要用微服务

微服务的各个服务之间比较独立，一个崩了其他的没事，而且可以独立地加资源

所有服务可以启动多个实例，但是写好的代码不可能知道各个实例在哪里，因此有一个注册中心

用户发请求到 GateWay，然后 GateWay 在注册表上边找一下，进行调度

好了，你的应用现在很牛逼了！

*数据库*

关系型数据库四节课

MySQL 缓存-打开的表有哪些，索引有哪些

缓存大小有限制

备份：

- 集群主从备份
- 逻辑备份 bin log，可以跨版本甚至跨数据库
- 直接拷贝文件做物理备份，比 bin log 省空间，但是会有版本问题
- partition，判断做 partition 有没有好处

讲完 MySQL 拓展出了一堆其他东西

MongoDB

- 面向 document
- 表叫做 collection
- 同一个 collection 的 schema 可以不一样
- MongoDB 的 sharding 根据 key 进行 sharding

Neo4j：

- 图数据库
- 好友关系，社交网络

LSMT InfluxDB：

- WAL
- LSMT 没有空间放大，但是有读放大：全部读完才能判断有没有，写放大：有可能很多 compaction

Datalake：

- 数据库很多，怎么放在 datalake 里边

当太多的时候，做分布式集群，集群的会话状态维护：

- gateway+register
- nginx

docker：

- 每一个 docker 有独立的 ip

- 容器化部署
- 要有一个容器管理的功能，K8S

如果在云里边：

- cloud computing
- edge computing

hadoop：

- MapReduce 并行计算，但是效率不高，读写硬盘太多

Spark：

- 不读写硬盘，进行内存操作
- RDD，分布式内存

storm：

- 上边的都是批数据
- 流式处理用 storm

HDFS：

- 本地文件系统上边构建出 HDFS
- 在 HDFS 上构建出 HBase 数据库和 Hive 数据仓库
- HBase 可以数据切分
- Hive 在 load 时候不做校验，而是在做 sql 式校验