

# 期末提纲

lec1: Overview of Enterprise Applications

State

如何选定scope

lect2: Messageing

Java Message Service

Message

Messaging Styles

Get a Connection and a Session

Create a Producer or a Consumer

Durable subscription 持久化预定

JMS Browser

lec3: WebSocket

WebSocket protocol

handshake

Endpoint

The process for creating and deploying a WebSocket endpoint

lec4: Transaction Management

Def transaction

事务特性

container-managed transaction demarcation 6个属性

@Transactional isolation

读写锁

Transaction Isolation Levels

Updating Multiple Databases

Concurrency

Optimistic Offline Lock

Pessimistic Offline Lock 悲观锁真的要加锁

lect5 : Multithreading

两种创立试线程的方式

Interrupts

Join

多线程交互

happens- before relationship

synchronized 关键字

Deadlock & livelock & starvation

Producer-Consumer

活锁

Executor线程池

Atomic Variables

lect6: Memory Caching & redis

Memory Management

Redis

适合放的元素

什么时候放到redis

作业思考

lect7: Full-text Searching

索引的概念

Lucene

lect8: Web Services

web services

restful web services

lect9: Microservices & Serverless

microservices

Eureka & routing

serverless

function service

lect10: security

Message digest 消息摘要

非对称加密

性质：

Symmetric Ciphers 对称加密

https 三次握手

单点认证 Single sign-on (SSO)

Kerberos

详细分析

Security Characteristics

安全策略Security Tactics

lect11: MySQL Optimization

硬件层优化：行存or列存？

索引的性质

mysql 使用index

索引设计原则

lect12: MySQL Optimization 2

Optimizing for innodb tables

Buffering and Caching

Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)

Configuring Buffer Pool Flushing

Caching of Prepared Statements and Stored Programs

MyISAM

lect 13: MySQL Backup & Recovery

物理备份和逻辑备份

online vs offline backup

local VS remote backup

snapshot

binary log

mysql enterprise backup

物理备份

逻辑备份

崩溃的情况

Using mysqldump for Backups

Q1

Q2

Q3

lect14 : MySQL Partitioning

背景：需要做分区&分区优势

分区策略

按照多列分区

Subpartitioning

How MySQL Partitioning Handles NULL

改变partition

manage hash and key partition

交换分区 exchange partition and subpartition

Q1

Q2

Q3

Q4

Q5

Mysql中InnoDB和MyISAM的比较

数据库更新方案

book

lect15: NoSQL & MongoDB

Mongodb

autosharding in MongoDB

lect16: Neo4J & Graph Computing

Graph database

lect 17: Log-Structured Database

LSTM tree

SSTABLE

lect18: Timeseries Database

InfluxDB

InfluxDB design principle

InfluxDB storage engine

InfluxDB shards and shard groups

lect20: Data Lake

lect21: Clustering

nginx

MySQL InnoDB cluster

primary和secondary的备份

改变拓扑结构

lect 22: Virtualization & Container

Container Volume

Multi container apps

多个容器一起启动：compose

lect23:Kubernetes

为什么需要 Kubernetes，它能做什么？

Kubernetes 组件

控制平面组件 (Control Plane Components)

Node 组件

lect 24: Cloud Computing

作业调度 → mapreduce

文件系统 → GFS

Bigtable

hadoop

two phase commit —— google percolator

edge computing

lect25: Hadoop

Yarn

map reduce优缺点

lect26: Spark

lect 27: storm

Apache storm

zookeeper

Storm Cluster的组成部分

Storm与Spark、Hadoop三种框架对比

lect28: HDFS

文件管理 (文件夹)

file system namespace

The Persistence of File System Metadata

snapshot

re-replication

删除文件

删除replica

lec29: HBase

数据存储

Namespace

Version管理

HBase和RDBMS的对比

lec30: hive

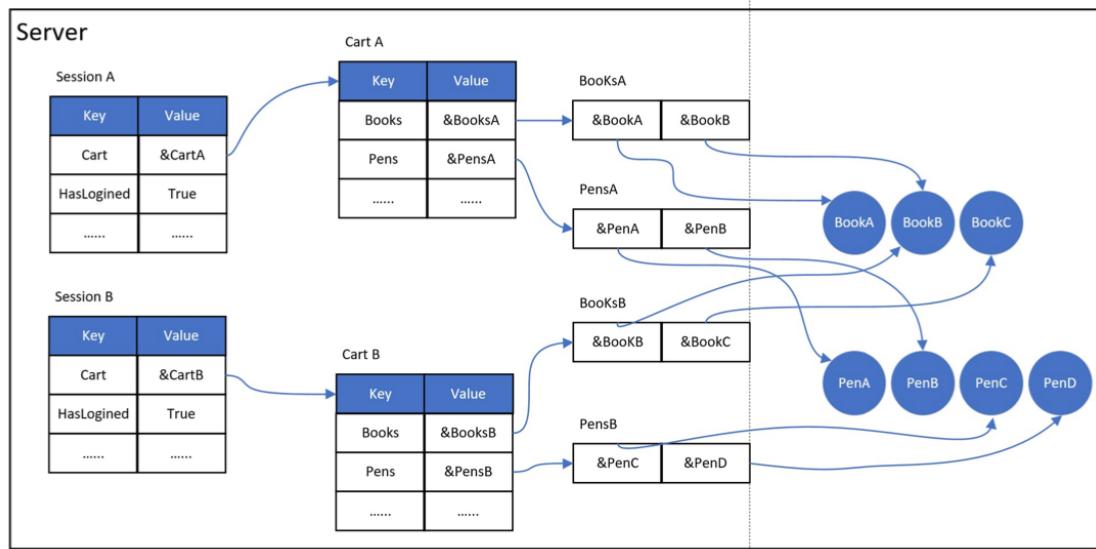
运行hive

partition column：新建分区表

RCFile：Hive中一种新的存储格式，将数据按列存储

# lec1:Overview of Enterprise Applications

- HTTP is a **stateless** protocol



整个运行的过程中只有一个controller & service

一个用户对应一个service → 一个uid

有状态和无状态

无：3个用户上来只有1个uid

有：一一对应，3个uid

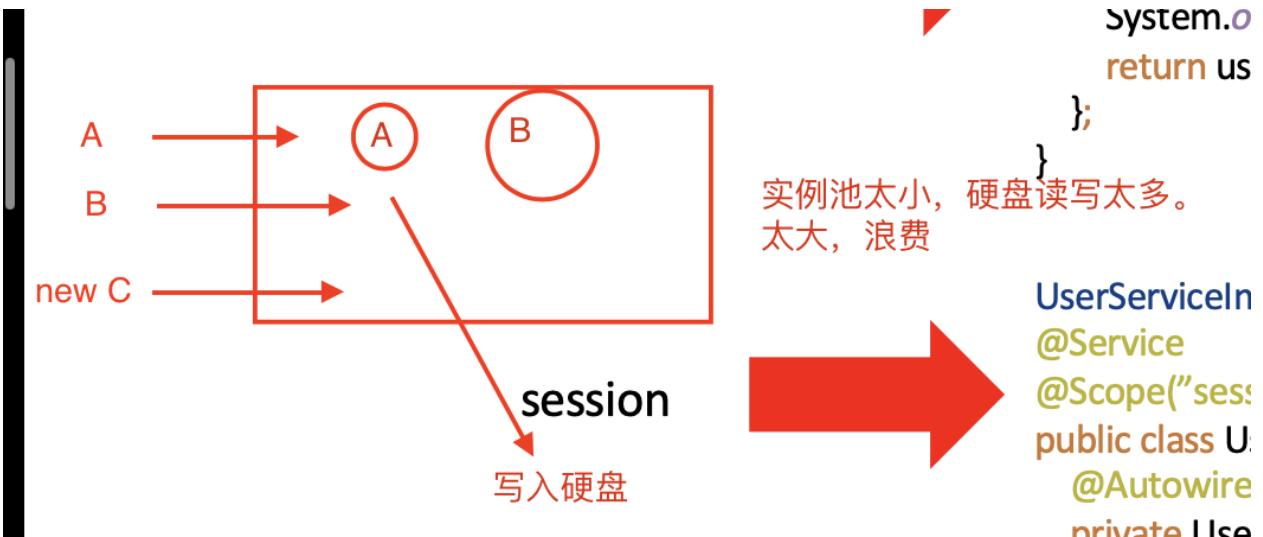
无状态：比如一个bill 出来计算一下不会根据个性化

有状态：比如cart,就不能？？

无状态就只有一个实例，当方便但是无法满足一些需求，有状态会比较麻烦。

□ 跑一下软件包。

区别：内存占用区分。Tomcat 实例池， Instance



## State

作用域

Table 4.4. Bean scopes

Scope	Description
<a href="#">singleton</a>	Scopes a single bean definition to a single object instance per Spring IoC container.
<a href="#">prototype</a>	Scopes a single bean definition to any number of object instances.
<a href="#">request</a>	Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
<a href="#">session</a>	Scopes a single bean definition to the lifecycle of a HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
<a href="#">global session</a>	Scopes a single bean definition to the lifecycle of a global HTTP session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

<https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html>

Singleton: 无状态， 默认的情况。单例模式。比如两个浏览器发请求，是由同一个service， controller，接收并服务的。

prototype : 创建多个对象，一个请求过来创建一个新实例。注意controller 和service都需要注明是 prototype

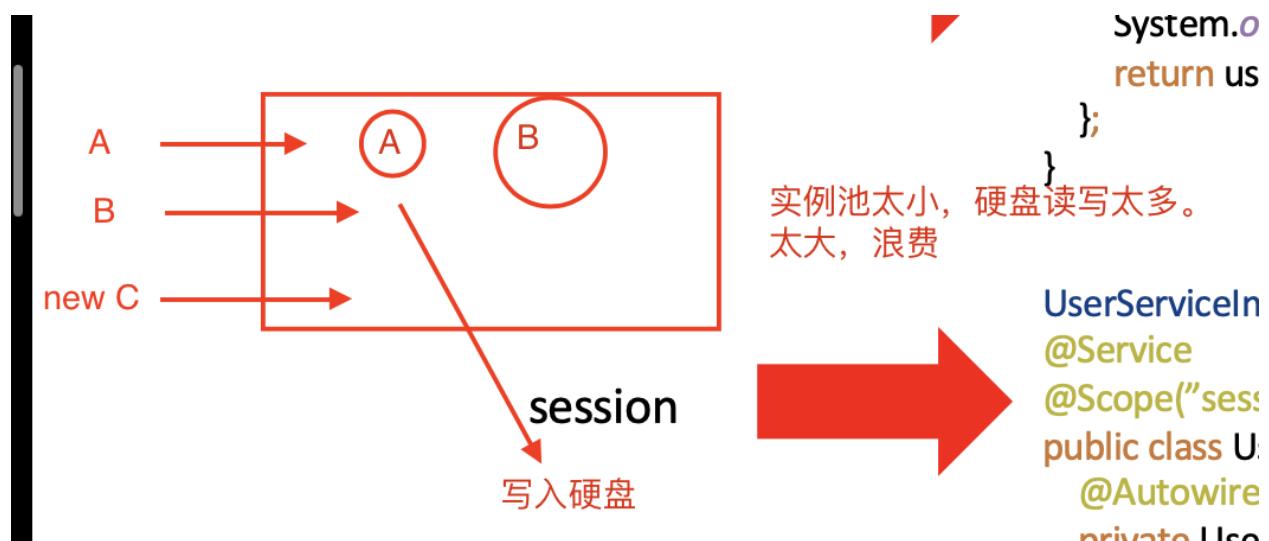
session : 但其实最好的是每个浏览器都有一个controller和service，而不是每次请求都创建。在每个会话里面创建一个对象。所以用applicationContext.getBean

		Controller	
		Prototype	session
Service	prototype	4 service instances 4 controller instances	4 service instances 2 controller instances
	session	2 service instances 4 controller instances	2 service instances 2 controller instances

## 如何选定scope

举例：比如一个用户对应一个user id, 如果使用单例模式，另一个用户进来就会把这个id改写掉，而希望的是，在这一个会话里面记住这个user id，这个uid就是状态，在一个HTTP调用是无状态的，要以这种方式维护状态。

会有内存的消耗问题。对象需要占用内存。在Tomcat中有一个配置叫做实例池。



有状态中，假设实例池只能放两个实例，A来了看一下有没有A的对象：没有，有没有空，有→ 创建一个，B来了，看一下有没有A的对象：没有，有没有空：有→ 创建一个。C来了，没了，但是不能让它等，把最久远的对象状态写入硬盘，就是A的状态。然后A让给C，但是这时候A, B, C按照这个顺序循环来了，就造成了硬盘频繁读写。实例池大小有很难控制，大了内存浪费，小了放不下而无状态中，

如果有靠不是http session维护的东西，那就得用有状态的了。

订单可以无状态，给你一个就算。但是购物车就不行了。

## lect2: Messaging

异步通信：页面的其他部分会先渲染，需要计算结果的等产生了再渲染，这样就不用等了。

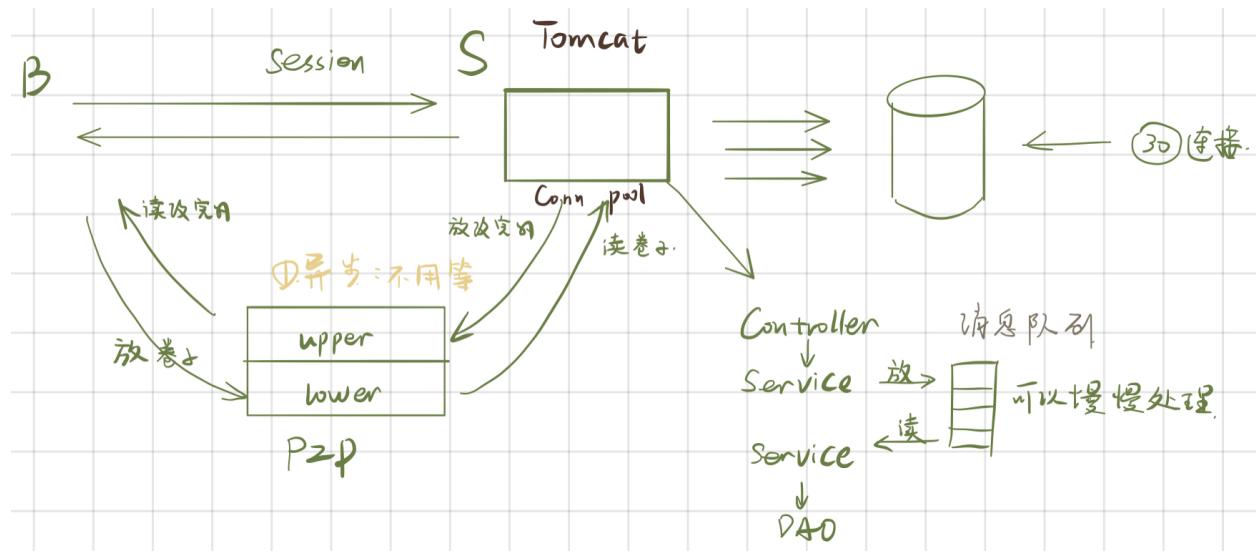
连接池：eg.Mysql 的最多连接是30个，这些连接可以分时复用，就是一个连接可以服务好多个客户。但是如果客户实在太多，session就超时了。

## Java Message Service

好处：

**Asynchronous:** 异步, 不用等。

**Reliable:** 如果一调用失败，就会重复尝试一直到超时为止，而不是像同步的，失败之后就bug.



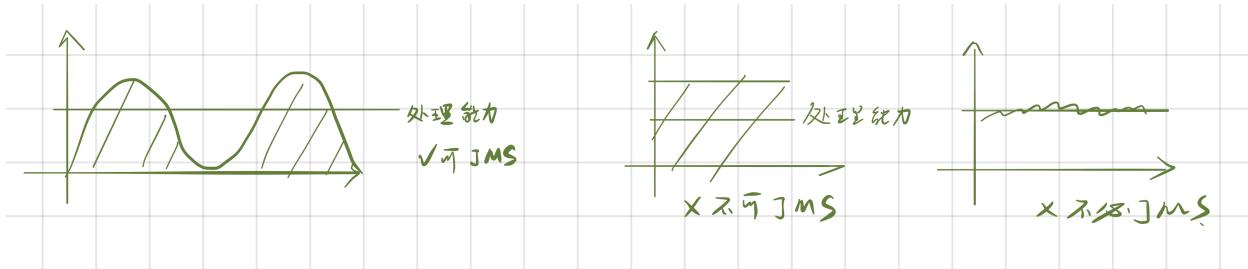
上面的service放的什么类是它定义的。注意解耦的思想。

问题：

1. 弱类型，如果经过中间件， $A \rightarrow B \rightarrow C$ ，若AC的参数不配，是没法检测出来的，因为 $A \rightarrow B$ 是纯文本表示。
2. 效率低，有中介带来性能损耗。

是否选择：

1. 如果事务简单又比较少，那就用同步。如果实在扛不住了，就用异步。
2. 如果确实有不忙的时候可以让异步有好处才行



## Message

就是一个对象，包含三部分

- A header (信封, eg 收发件地址)

- Includes some pre-defined fields
  - JMSEdestination (S)
  - JMSEDeliveryMode (S)
  - JMSEMessageID (S)
  - JMSTimestamp (S)
  - JMSCorrelationID (C) 发送模式 1. 转发不成功, 丢了, 需要保证顺序的应用  
2. 转发不成功再转
  - JMSEReplyTo (C)
  - JMSERedelivered (P)
  - JMSType (C)
  - JMSEExpiration (S)
  - JMSPriority (S)

- Properties(optional)

可扩展，增加一个tag, 一个信箱很多人用，只有某些条件的人才会收到信件。

<b>Message Type</b>	<b>Body Contains</b>
TextMessage	A java.lang.String object (for example, the contents of an XML file).
MapMessage	A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

– A body(optional) 内容

## Messaging Styles

point-to-point (PTP)

每个消息只有一个接收者

publish/subscribe 发送/订阅

- 一条消息过来了可以被很多人接收到。
- 每个人可以接受其中几类订阅的消息。

## Get a Connection and a Session

```
JMSContext context = connectionFactory.createContext();
```

创建一个context自动创建 connection & session.

## Create a Producer or a Consumer

```
try (JMSContext context = connectionFactory.create
```

## Durable subscription 持久化预定

```
String subName = "MySub";
JMSConsumer consumer =
    context.createDurableConsumer(myTopic, subName);
consumer.close();
context.unsubscribe(subName);
JMSConsumer consumer = context.
    createSharedDurableConsumer(topic, "MakeItLast");
```

发现消息队列里面有上次没有推送的（上次离线到现在上线这段时间内）推送过来。

## JMS Browser

看一下message 显示头

```
QueueBrowser browser = context.createBrowser(queue);
```

### 3.2 value的设计

#### ◆ 设计原则

- 1 ) 单个key的体积不要过大，大key拆分成多个小key：
  - ✓ String类型避免存储长度超过2k的字符串；
  - ✓ List、Hash、Set和Zset的元素个数不要超过5000个。
- 2 ) 尽可能使用简单的数据类型。
- 3 ) 选择合适的数据类型。

## lec3: WebSocket

需求：之前都是前台请求，然后给相应，但是如果向股票实时相应需要后端给请求。如果在前端写，控制不好时间间隔，最好是后端按需给前端。—— web socket

前后台都有同步异步通信，然后又有解耦。

full-duplex 双工，对等可以互发收

## WebSocket protocol

WebSocket protocol = handshake + data transfer.

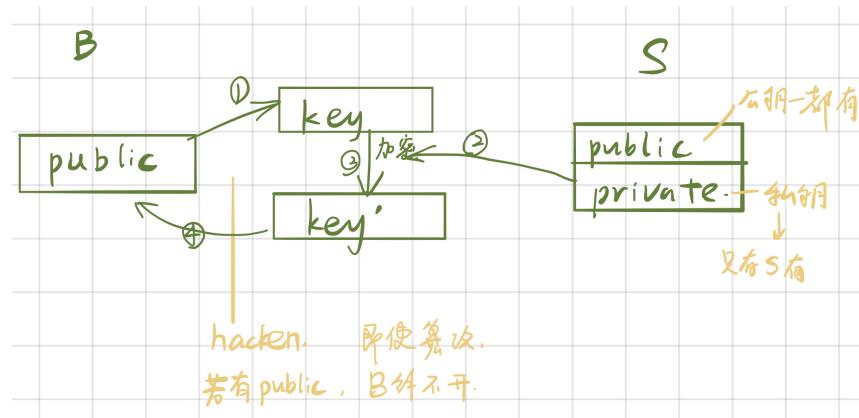
### handshake

客户端发

```
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEF1kg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```

sec-WebSocket-Key : 密钥但是这里还是明文，没有加密

服务器收using its URI.如果双方都用证书对上面的密钥进行处理。



公钥加密的公钥解不开只能私钥解开，私钥加密的，私钥解不开只能公钥解开。

## Endpoint

### The process for creating and deploying a WebSocket endpoint

1. Create an endpoint class.
2. Implement the lifecycle methods of the endpoint.
3. Add your business logic to the endpoint.
4. Deploy the endpoint inside a web application.

Annotation	Event	Example
OnOpen	Connection opened.	@OnOpen public void open(Session session, EndpointConfig conf) {}
OnMessage	Message received.	@OnMessage public void message (Session session, String msg) {}
OnError	Connection error.	@OnError public void error(Session session, Throwable error) {}
OnClose	Connection closed.	@OnClose public void close(Session session, CloseReason reason) {}

onMessage 有多个重载的版本，可以根据消息的类型有多个函数，onMessage 是收到消息的反应

- **Receive messages**

```

@ServerEndpoint("/receive")
public class ReceiveEndpoint {
    @OnMessage
    public void textMessage(Session session, String msg)
    { System.out.println("Text message: " + msg); }

    @OnMessage
    public void binaryMessage(Session session, ByteBuffer msg)
    { System.out.println("Binary message: " + msg.toString()); }

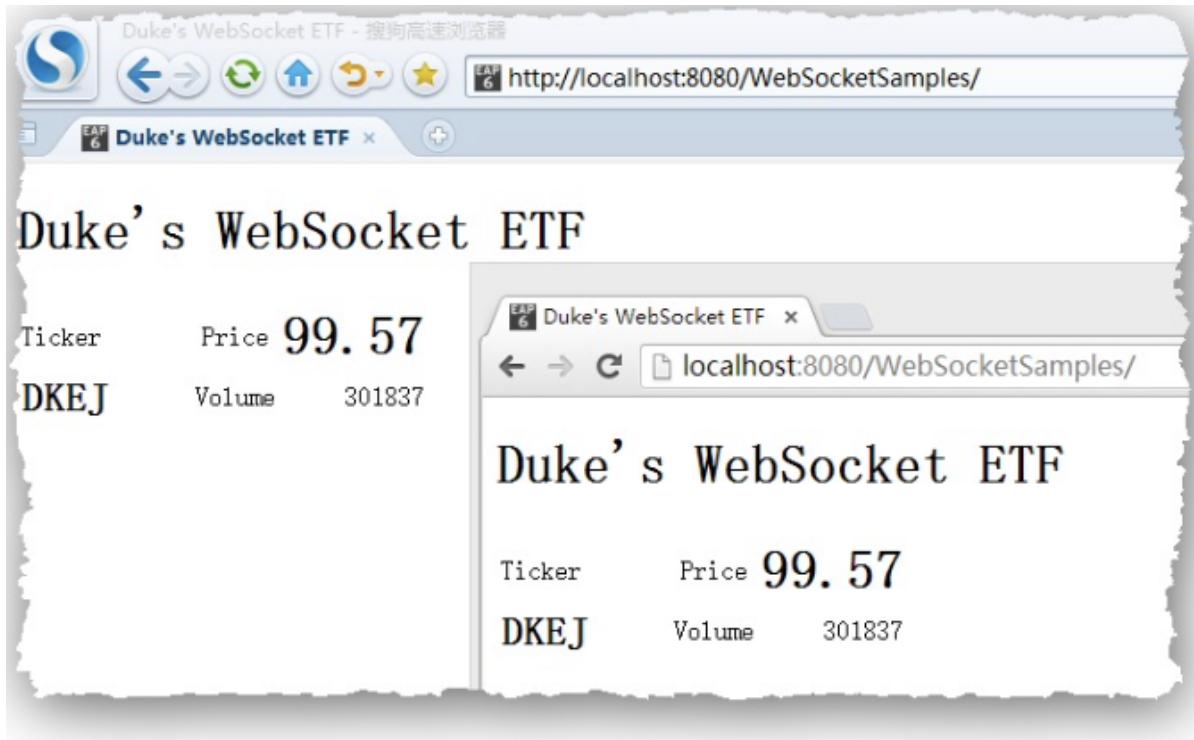
    @OnMessage
    public void pongMessage(Session session, PongMessage msg)
    {
        System.out.println("Pong message: " +
                           msg.getApplicationData().toString());
    }
}

```

可以和多个客户端推送一样的东西。

有可能的问题：客户端太多了，还没发完就更新新的东西了。解决方案发，做集群，分布式服务器。

例子：定时器，研究一下



例子：聊天室

发送的是java对象，需要一个encoder转化为字符串，解开需要decoder

Encoder: 给一个msg的对象，作为string 发过去

Decoder: 把字符传转化为对象

## lec4: Transaction Management

### Def transaction

case1

```
transfer{
    withdraw();
    deposit();
}
```

A(-100) —100—> B(+100) 如果 B+100 失败了，那就必须什么都不做，不能中间失败，所以 transfer 是一个原子性的操作。

## case2

A(-100) —100—> B(+200) ← -100— C(-100)

需要保证B是200，必须要让AC串行执行。所以事务之间需要有隔离性。

## 实现

有一个cahce, 如果cache里面成功就一起写入数据库，否则就使用原来的内存的东西。

同时又rollback机制，如果其中一条失败了，前面的都要回去

TM : DBMS, JMS, Java MailServer

## 事务特性

**Consistency:** 如果数据上有约束，那么这个约束是不可以违反的，保证数据的一致性。

**Durable:** 事务一旦提交，它的状态应该是持久化的。

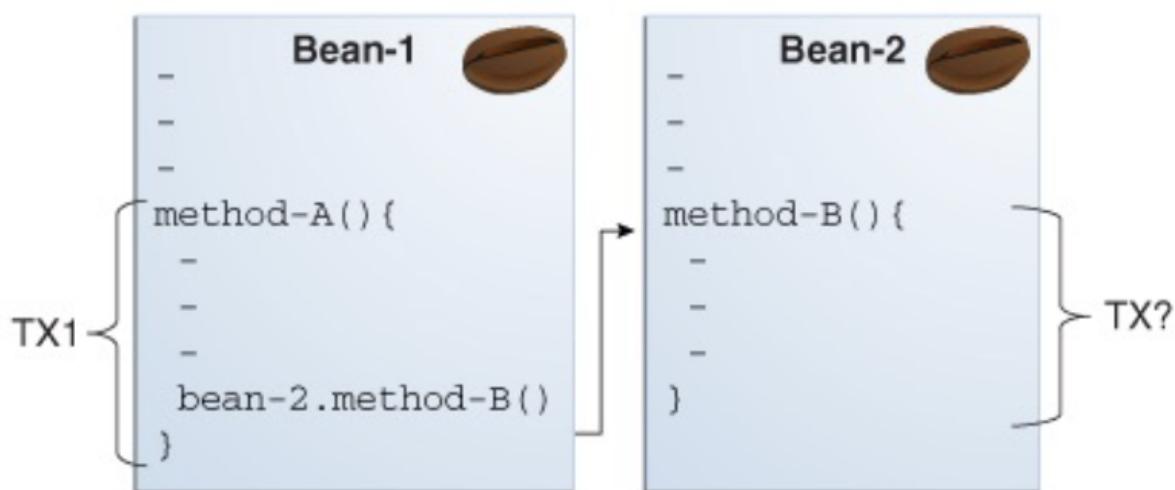
ACID

事务的4个特性：原子性，consistency（数据有约束），isolation,durable(一旦提交事务是持久化的)

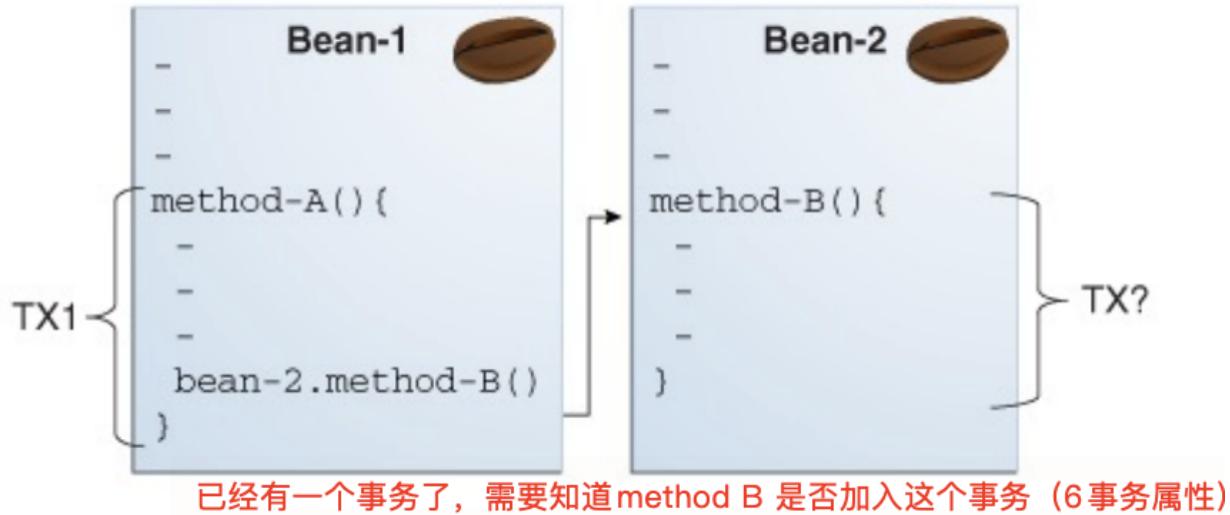
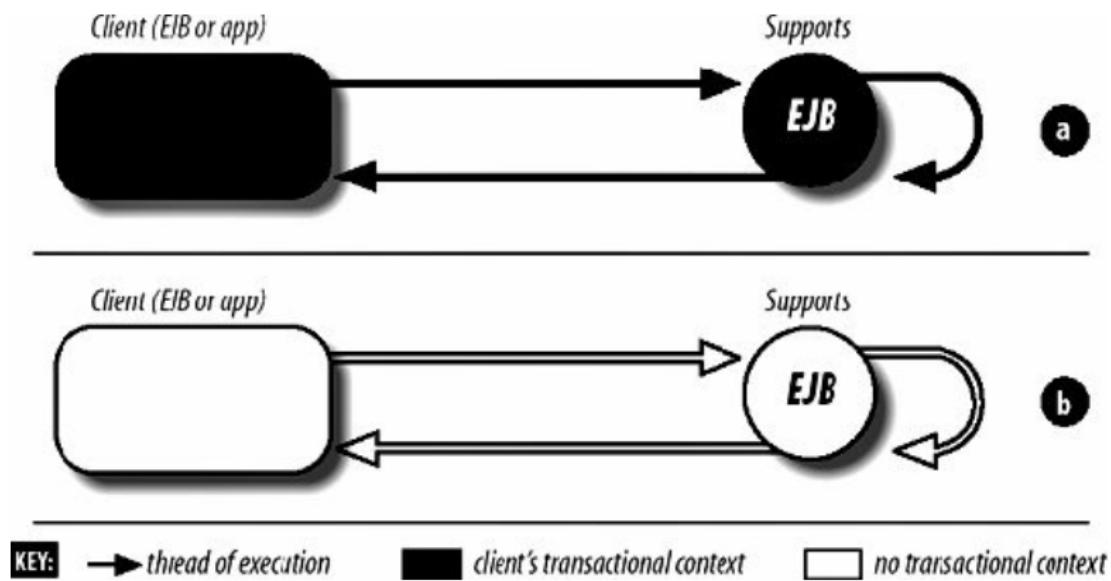
ACID

## container-managed transaction demarcation 6个属性

功能：划分事务的边界



## SECTION SCOPE IT IS INVOLVED WITHIN A TRANSACTION



### 事务属性

Transfer一开始新建一个事务，是require，就开一个，Withdraw发现已经有一个事务了，就加入这个事务，同理和deposit，只有当transfer, 完成了，事务才可以提交。其中任何一个事务失败就会roll back. 这样就实现了要么都执行，要么都不执行。

传播属性的值	外部不存在事务	外部存在事务	用法	备注	其他
REQUIRED	开启新的事务	融合到外部事务中	@Transactional(propagation = Propagation.REQUIRED)	增删改方法	
SUPPORTS	不开启事务	融合到外部事务中	@Transactional(propagation = Propagation.SUPPORTS)	查询方法	
REQUIRES_NEW	开启新的事务	挂起外部事务，创建新的事务	@Transactional(propagation = Propagation.REQUIRES_NEW)	日志记录方法中	失败了日志也需要记录的
NOT_SUPPORTED	不开启事务	挂起外部事务	@Transactional(propagation = Propagation.NOT_SUPPORTED)	极其不常用	
NEVER	不开启事务	抛出异常	@Transactional(propagation = Propagation.NEVER)	极其不常用	
MANDATORY	抛出异常	融合到外部事务中	@Transactional(propagation = Propagation.MANDATORY)	极其不常用	

### 19-Spring中的事务属性 (Transaction Attribute) \_前路无畏的博客-CSDN博客

上一篇：18-Spring事务控制@Transactional

<https://blog.csdn.net/fsjwin/article/details/109497305> 属性：描述事务特征的一系列值 性别、身高、年龄、体重等... ... 事务的属性：就是能在@Transactional（这个括内些的

C [https://yuhongliang.blog.csdn.net/article/details/109500885?spm=1001.2101.3001.6650.3&utm\\_medium=distribute.pc\\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-3.no\\_search\\_link&depth\\_1-utm\\_source=distribute.pc\\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-3.no\\_search\\_link&utm\\_relevant\\_index=6](https://yuhongliang.blog.csdn.net/article/details/109500885?spm=1001.2101.3001.6650.3&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-3.no_search_link&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-3.no_search_link&utm_relevant_index=6)

这些都只是改了annotation，只是改了事务的边界

```
al(
ation = Isolation.READ_COMMITTED,
agation = Propagation.REQUIRED, //
Only = true, //3. 只读属性
out = -1, //4. 超时属性
backFor = Exception.class, //5. 异常
llbackFor = NullPointerException.c
```

eg. TWD 后面还有一个log, 但是它不靠谱, 所以就使用require new 或者Not support 来跳过它。

<b>Transaction Attribute</b>	<b>Client's Transaction</b>	<b>Business Method's Transaction</b>
Required	None	T2
Required	T1	T1
RequiresNew	None	T2
RequiresNew	T1	T2
Mandatory	None	Error
Mandatory	T1	T1
NotSupported	None	None
NotSupported	T1	None
Supports	None	None
Supports	T1	T1
Never	None	None
Never	T1	Error

<b>Transaction Attribute</b>	<b>TransactionAttributeType Constant</b>
Required	TransactionAttributeType.REQUIRED
RequiresNew	TransactionAttributeType.REQUIRES_NEW
Mandatory	TransactionAttributeType.MANDATORY
NotSupported	TransactionAttributeType.NOT_SUPPORTED
Supports	TransactionAttributeType.SUPPORTS
Never	TransactionAttributeType.NEVER

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
    ...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}
    public void thirdMethod() {...}
    public void fourthMethod() {...}
}
```

#### 事务属性使用举例

然而，现在数据库可以回滚，如果里面有一个int count; count++ 就回不去，他不属于数据库管辖的范畴 需要使用 SessionSynchronization

```
afterBegin:
    oldvalue = value;
afterCompletion(boolean b):
    if (!b)
        value = oldvalue;
```

String... 可变长变量

h2 database <pom.xml>

- The **SessionSynchronization** interface, which is optional, allows stateful session bean instances to receive transaction synchronization notifications.
  - The container invokes the **SessionSynchronization** methods (**afterBegin**, **beforeCompletion**, and **afterCompletion**) at each of the main stages of a transaction.
    - afterBegin**: 事务开始之后
    - beforeCompletion**: 事务提交或者回滚之后
    - afterCompletion(boolean b)**: 传输bool判断成功还是失败

```

afterBegin: 不受事务管理器自己写的变量是无法 rollback 的
               比如变量 ++
   oldvalue = value;

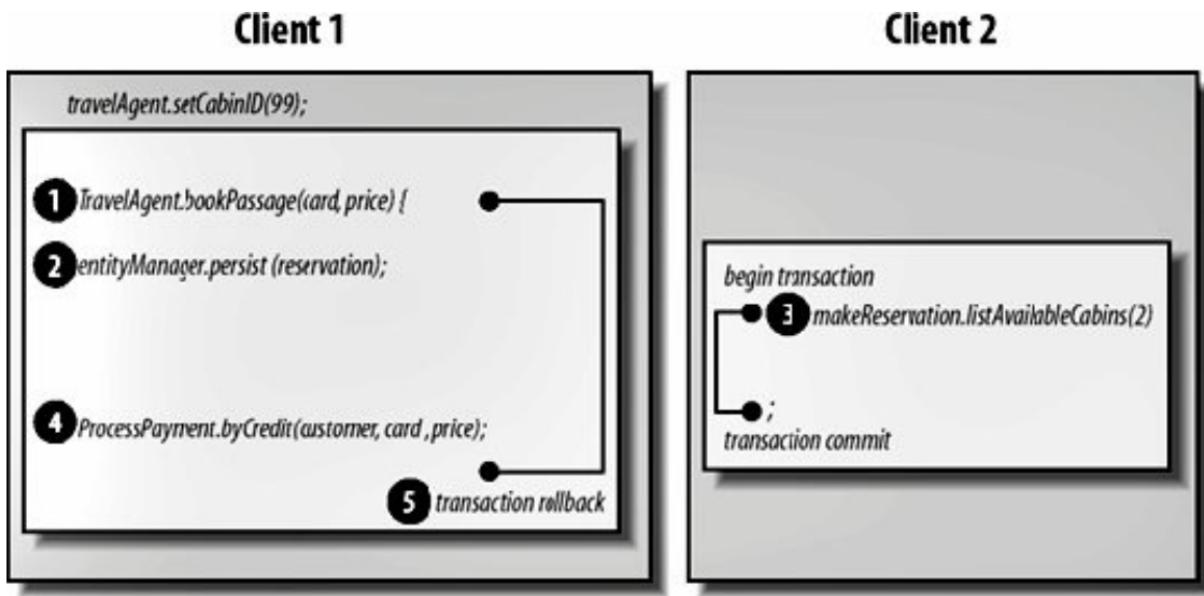
afterCompletion(boolean b):
  if (!b)
    value = oldvalue;

```

## @Transactional isolation

事务隔离级别	说明
@Transactional(isolation = Isolation.READ_UNCOMMITTED)	读取未提交数据(会出现脏读, 不可重复读), 基本不使用
@Transactional(isolation = Isolation.READ_COMMITTED) (SQLSERVER默认)	读取已提交数据(会出现不可重复读和幻读)
@Transactional(isolation = Isolation.REPEATABLE_READ)	可重复读(会出现幻读)
@Transactional(isolation = Isolation.SERIALIZABLE)	串行化

@Transactional 不写就是require



1. 一个写之前的中间状态被别人读到了 Dirty reads
2. 在读的时候数据被改了 repeatable reads, 如果读的时候锁死，别人不能改写。反复读数据就是一致的不发生变化
3. 按照某种需求改写？ Phantom Reads

- **Read locks**
  - Read locks prevent other transactions from changing data read during a transaction until the transaction ends, thus preventing nonrepeatable reads.
  - Other transactions can read the data but not write to it. The current transaction is also prohibited from making changes.
- **Write locks**
  - Write locks are used for updates. A write lock prevents other transactions from changing the data until the current transaction is complete but allows dirty reads by other transactions and by the current transaction itself.
  - In other words, the transaction can read its own uncommitted changes.

Nonreapeatable read和Phantom read 的区别：

Non-repeatable Read不可重复读：在一个事务中，同样的数据被2次读取，得到不同的结果集

Phantom Read幻读：在一个事务中，同样的sql被2次执行，得到不同的结果集。

不可重复读的重点是修改:同样的条件, 你读取过的数据, 再次读取出来发现值不一样了

幻读的重点在于：新增或者删除同样的条件, 第1次和第2次读出来的记录数不一样

从锁的角度来看, 两者的区别就比较大：

对于前者, 只需要锁住满足条件的记录

对于后者, 要锁住满足条件及其相近的记录

## 读写锁

读锁：如果我在读别人不能写

写锁：我在写，别人不能写，但是别人可以读甚至可以读到我正在写的东西。但是可能触发别人读到 dirty reads.

Exclusive write locks: 只能读到我事务执行之前或者提交事务之后的数据，不会有正在写的中间的东西。就不会有脏读。

Snapshots：整个快照出来在这个里面操作，然后覆盖之前所有的，可以解决所有问题但是性能比较差。（就是OOC）

## Transaction Isolation Levels

- **Read Uncommitted**

- The transaction can read uncommitted data (i.e., data changed by a different transaction that is still in progress).
- Dirty reads, nonrepeatable reads, and phantom reads can occur.
- Bean methods with this isolation level can read uncommitted changes.

- **Read Committed**

- The transaction cannot read uncommitted data; data that is being changed by a different transaction cannot be read.
- Dirty reads are prevented; nonrepeatable reads and phantom reads can occur.
- Bean methods with this isolation level cannot read uncommitted data.

- **Repeatable Read**

- The transaction cannot change data that is being read by a different transaction.
- Dirty reads and nonrepeatable reads are prevented; phantom reads can occur.
- Bean methods with this isolation level have the same restrictions as those in the Read Committed level and can execute only repeatable reads.

- **Serializable**

- The transaction has exclusive read and update privileges; different transactions can neither read nor write to the same data.
- Dirty reads, nonrepeatable reads, and phantom reads are prevented.
- This isolation level is the most restrictive.

序列化：防止换读？？

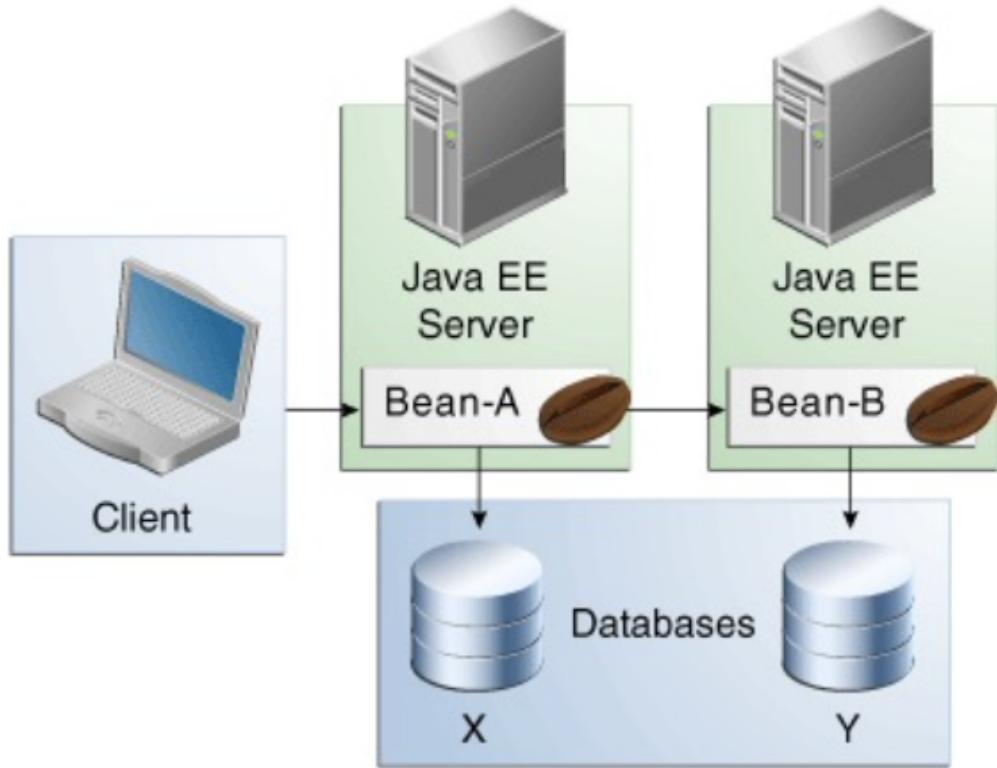
无隔离 → 防止脏读 → 防止重复读 → 防止换读

```
DataSource source = (javax.sql.DataSource)
                    jndiCntxt.lookup("java:comp/env/jdbc/titanDB");
Connection con = source.getConnection();
con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
性能很差，隔离级别高，完全不能做任何操作
```

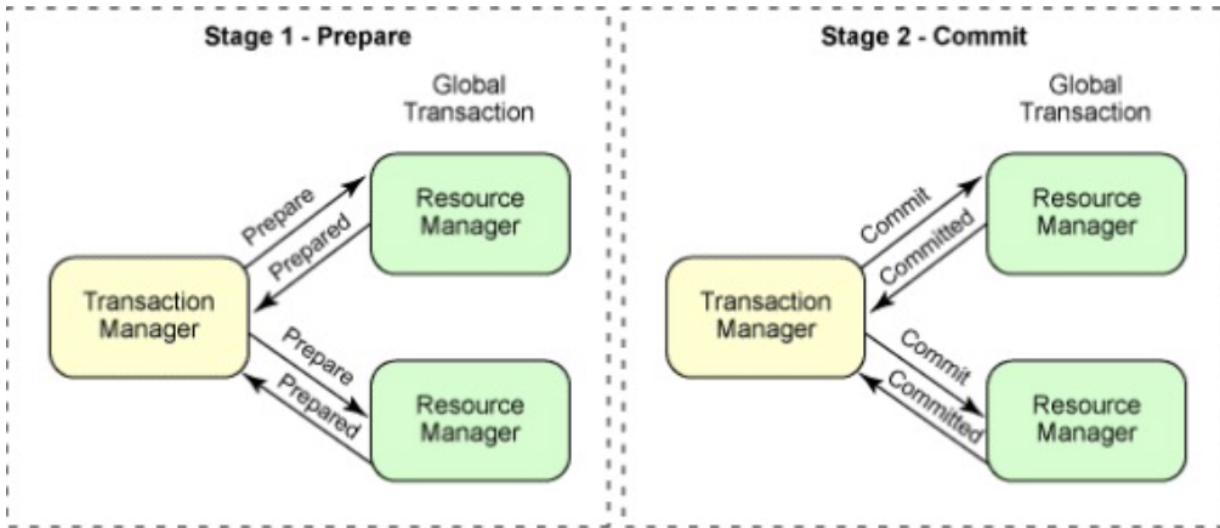
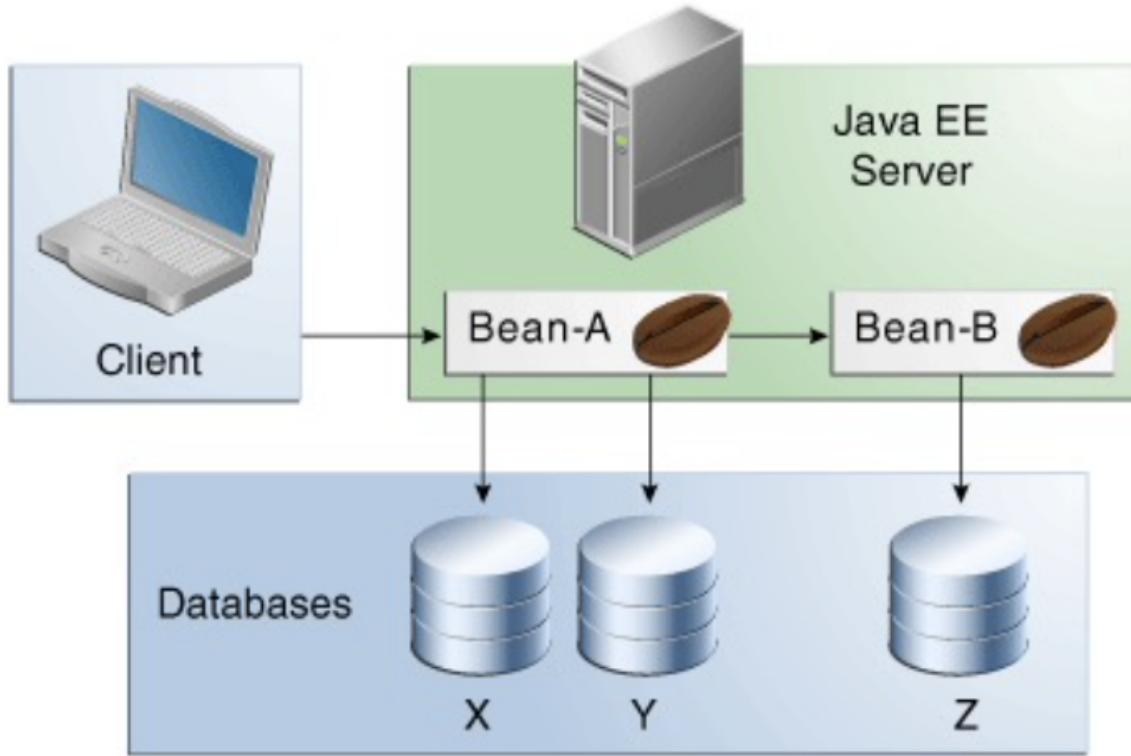
需要根据需求选择合适的隔离级别。

## Updating Multiple Databases

如果有其中有一个失败了，那由tomcat 决定是否都回滚，分布式事务。



Bean-A , Bean-B 是什么意思， 不一定是两台机器，就算是同一台机器的mysql,只要在两个库就是分布式的。

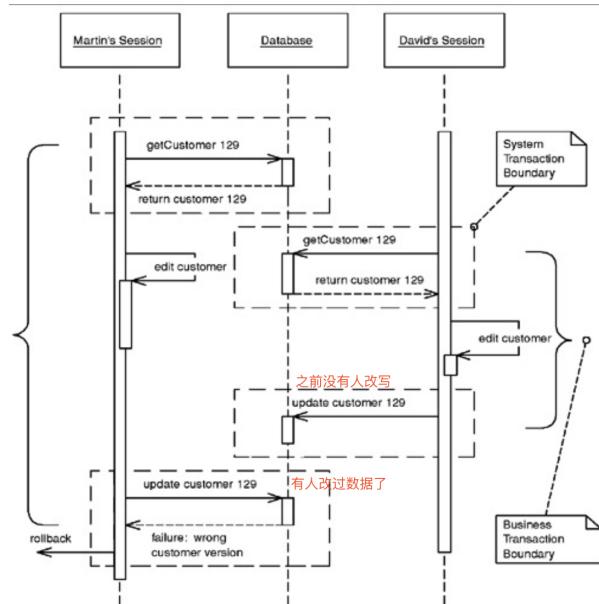


1. 投票阶段（通常情况一票否决） 2. 提交/回滚阶段（两者执行的操作是一样的，即使一个能提交一个不能提交） 虽然两个人不知道对方，但是可以做出一致性的操作。因为加了锁，都需要加个超时就放锁。如果让他们回滚网络断了，就不知道是该回滚还是commit了，就启发式动作（猜）。这里就需要人为干预了。

## Concurrency

## Optimistic Offline Lock

认为发生这种事情的概率比较低，不锁，在写入数据的时候检查一下当时读走的数据和现在的数据是不是已经被改写了。方法：在每个数据后面都加一个版本号/时间戳



A      B      C

AC 读B

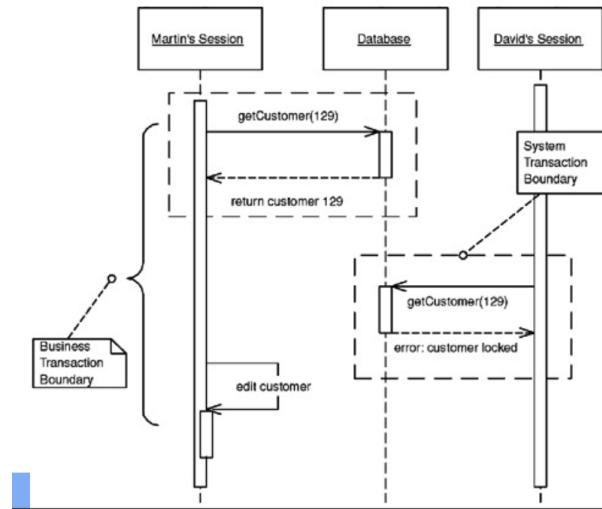
0 —> 0            0

1 ←— 0

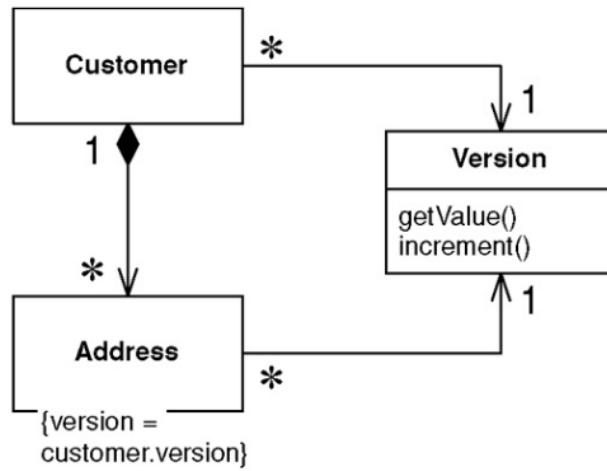
所以出现了版本号的抛异常

## Pessimistic Offline Lock 悲观锁真的要加锁

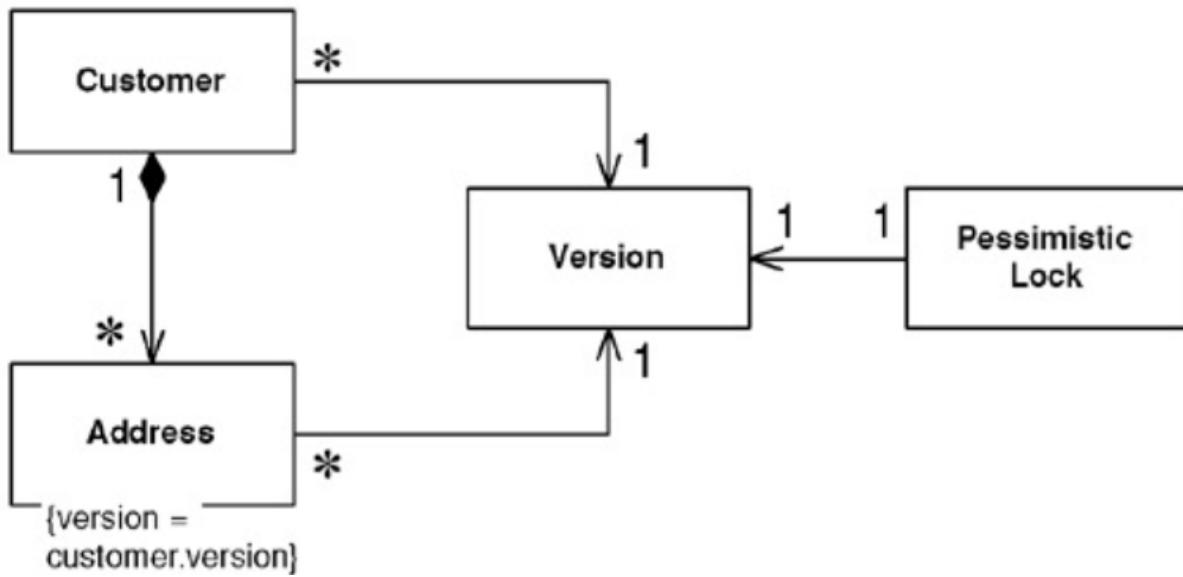
REliable, INtelligent & Scalable Systems



Coarse-Grained Lock



OOC: customer 或者 address 被修改了，就看一下两个人的version是不是和现在的一样



2pl: 真的只有一个version可以读，一个transaction读了之后就把它锁住，等结束了下一个才可以用，

考试：如果给一个6种机制的组合，会有什么结果

## lect5 : Multithreading

进程有独立的存储空间 (eg. java 虚拟机), 线程是共享的, A线程可以访问到B线程的成员和数据, 这样速度快但是可能出错。

### 两种创立线程的方式

- *Subclass Thread*

– The `Thread` class itself implements `Runnable`, though its `run` method does nothing. An application can subclass `Thread`, providing its own implementation of `run`, as in the `HelloThread` example:

```

public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}

```

- *Provide a Runnable object.*

- The Runnable interface defines a single method, `run`, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the `HelloRunnable` example:

```
public class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

单根继承，对于更复杂的类，推荐

The screenshot shows an IDE interface with several tabs at the top: Deadlock.java, HelloRunnable.java, RunnableTest.java (which is currently selected), and HelloThread.java. The left sidebar shows a project structure with various Java files like Consumer, Counter, Deadlock, Drop, ForkBlur, HelloRunnable, HelloThread, ImagePanel, ImmutableRG, Producer, and ProducerCons. The RunnableTest.java code is displayed in the main editor:

```
public class RunnableTest implements Runnable{  
    public void run() {  
        System.out.println(11);  
        System.out.println("Love you");  
        System.out.println("Hello from a thread!");  
    }  
}  
public static void main(String args[]) {  
    (new Thread(new RunnableTest())).start();  
    (new Thread(new RunnableTest())).start();  
}
```

Below the code, the 'Run' section shows the command: `/Library/Java/JavaVirtualMachines/jdk-16.0.1.jdk/Contents/Home/bin/java -javaagent:/Applications/`. The output window shows the following text:

```
11  
11  
Love you  
Love you  
Hello from a thread!  
Hello from a thread!
```

A red annotation '两个线程交互执行' (Two threads interacting) is placed next to the last two lines of output.

## Interrupts

1. try catch 抛出异常

```

for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {      中断, 中断位会被标注。
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}

```

## 2. 使用中断位检查是否异常

```

RECEIVED. FOR EXAMPLE.
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) { 判断中断位
        // We've been interrupted: no more crunching.
        return;
    }
}

```

## 3. 打断t: t.interrupt();

# Join

把现在执行的挂起，然后去执行t, 等t结束/挂起的时候，执行当前的。 (cause the current thread pause until t thread terminates)

t.join();

Waits at most millis milliseconds for t thread to die. A timeout of 0 means to wait forever.

当双方产生了资源的征用

当一个thread sleep, CPU就会让给另外一个线程。

```

public class SimpleThreads {
    // Display a message, preceded by
    // the name of the current thread
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }

    private static class MessageLoop implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats", "Does eat oats", "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0; i < importantInfo.length; i++) {
                    // Pause for 4 seconds
                    Thread.sleep(4000);
                    // Print a message
                    threadMessage(importantInfo[i]);
                }
            } 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;

    // If command line argument
    // present, gives patience
    // in seconds.
    if (args.length > 0) {
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }
}

```

## 工具函数

```
static void threadMessage(String message) {  
    String threadName =  
        Thread.currentThread().getName();  
    System.out.format("%s: %s%n",  
        threadName,  
        message);  
}
```

## 多线程交互

- 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.

在调用写数据的时候，是串行的。否则产生了Memory consistency errors。

- 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++;
```

 在counter++和 print之间有别的线程在改
  - Then, shortly afterwards, thread B prints out counter:  

```
System.out.println(counter);
```

两个线程做计数器结果偶数的跳，有地方连续，很混乱。原因，当一个计数器++ 之后再print，++和print中间另外一个线程又跑过来++ 了。

变成了++ ; ++ ; out;out

解决方法：使用synchronized 关键字

当syn的函数在调用的时候，所有其他都有synchronized 的函数都不可以被调用。

如果是static对象，就只能放到类上

## **happens- before relationship**

### **synchronized 关键字**

当syn的函数在调用的时候，所有其他都有synchronized 的函数都不可以被调用。

机制：自动建立了一个happens-before relationship, java自动建立了锁。

**intrinsic lock or monitor lock.**

#### **对象上的锁**

谁获得谁释放，别人是无法释放的。

#### **类上的锁**

如果是static 方法，和object 没有绑定，需要用类上的锁。

#### **细粒度的锁**

如果整个方法加锁，效率低下，但是把几行给锁起来就可以。

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

#### **多个锁**

lock1 和lock2 是不会相互影响的。所以inc1和inc2可以并行，但是同时inc1多个thread不可以。

#### **防止死锁——可重入的锁**

## **Deadlock & livelock & starvation**

死锁：产生了征用，等一个条件，双方的条件是互斥的。打破结局需要回滚。

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

活锁：大家都还没有占用资源，只要有一方占用一下就可以了。

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked - they are simply too busy responding to each other to resume work.

starvation:

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.

饥饿是指某一线程或多个线程在某种情况下无法获取所需要的资源，导致程序无法执行。比如，当某个线程的优先级太低的时候，那么高优先级的线程会始终霸占着资源，而低优先级的线程由于无法得到相应的资源而无法工作。

上桥，双方都上了，不让对方：死锁

上方都没上，让对方上：活锁

## Producer-Consumer

信箱不能出现太多和太少的问题。

o'iuyiouyipoipoiupyouiypoiupyoiuupouyipoioiuyoiupo

新建了一个drop, 同时给消费者（放入信箱）和生产者（获取信件），这里很重要的是当信箱为空，双方就主动释放现在的锁（wait()），这样就打破了死锁。

```

public synchronized void put(String message) {
    // Wait until message has been retrieved.
    while (!empty) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    // Toggle status.
    empty = false;
    // Store message.
    this.message = message;
    // Notify consumer that status has changed.
    notifyAll();
}
}

```

即使getRGB , getName 都是synchronized 在中间也可能被改写

```

...
int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2

```

中间也有一个线程  
调用了一下set

所以我们就有了不可变的object, 一旦被赋值再也没有改变。每一次用的时候都要创建新的对象而不是使用原来的对象，线程绝对安全。

## 活锁

如果两个锁一方不能同时拿到，就这方同时释放掉这两把锁，让对方可能拿到。

bow的例子

## Executor线程池

线程对象，设置上限，如果满了，就挂起最老的，加入新的

```
mSplit = mLength / 4,  
invokeAll(  
    new ForkBlur(mSource, mStart, split, mDestination),  
    new ForkBlur(mSource, mStart + split, mLength - split, mDestination));  
}
```

切成两个线程交给线程池

## Atomic Variables

- One way to make Counter safe from thread interference is to make its methods synchronized, as in [SynchronizedCounter](#):

这样就递增不会跳了。

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

但是要自己写synchronized 改成了AtomicInteger

Replacing the `int` field with an `AtomicInteger` allows us to prevent thread interference without resorting to synchronization, as in `AtomicCounter`:

```
import java.util.concurrent.atomic.AtomicInteger;
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {
        c.incrementAndGet();
    }
    public void decrement() {
        c.decrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

如果两个线程就算是一个对象的static 成员也是他们私有的，如果是随机数就不一样。

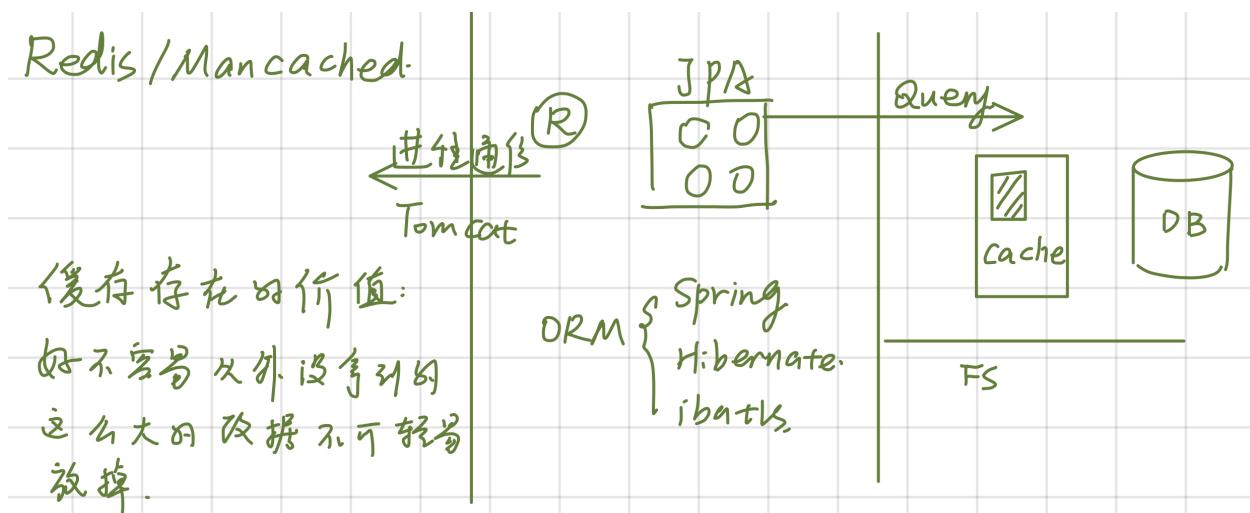
`AtomicLong`

如需要每个线程的随机数都不一样就每次调用这个

`ThreadLocalRandom.current().nextBoolean/ nextInt`

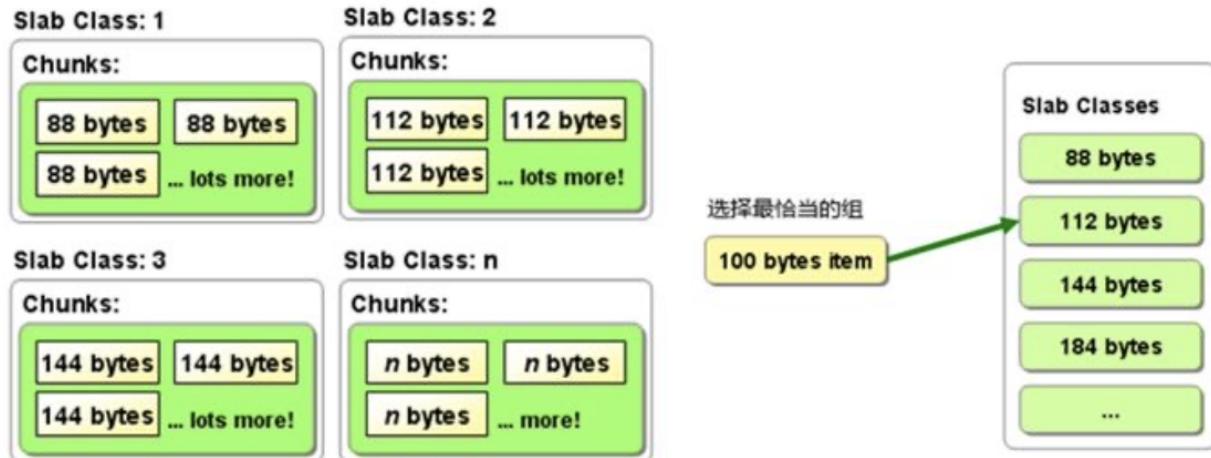
这个函数只和当前的线程有关，每次用这个工厂方法得到新的实例。

## lect6: Memory Caching & redis

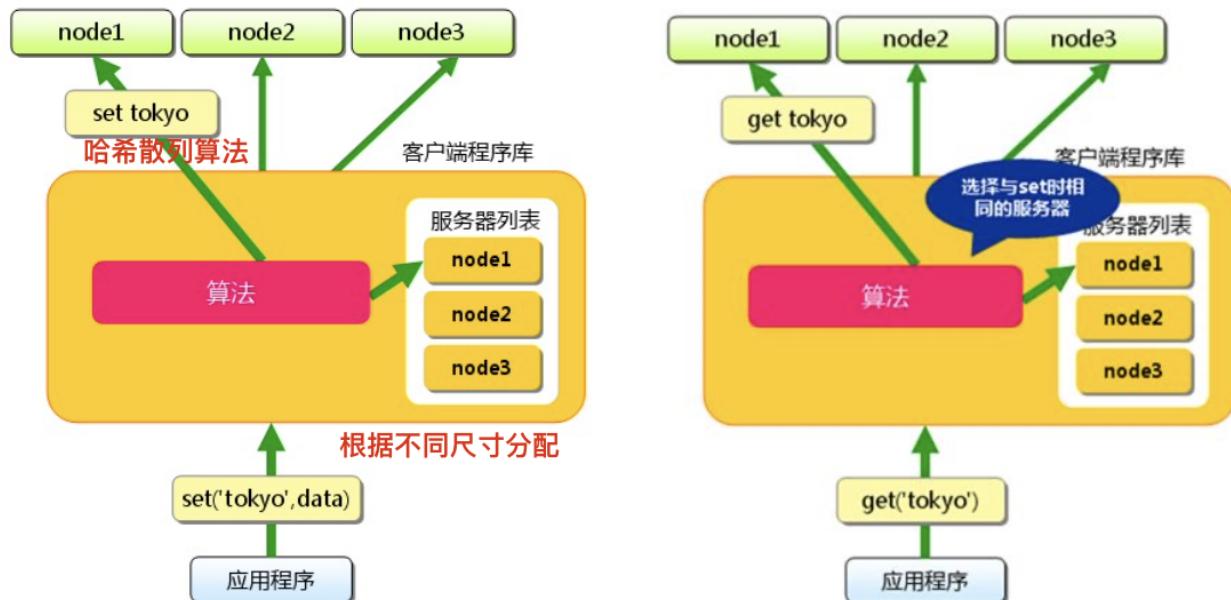


# Memory Management

问题1：如何合理使用内存，会有小的东西也要浪费一个大格子。

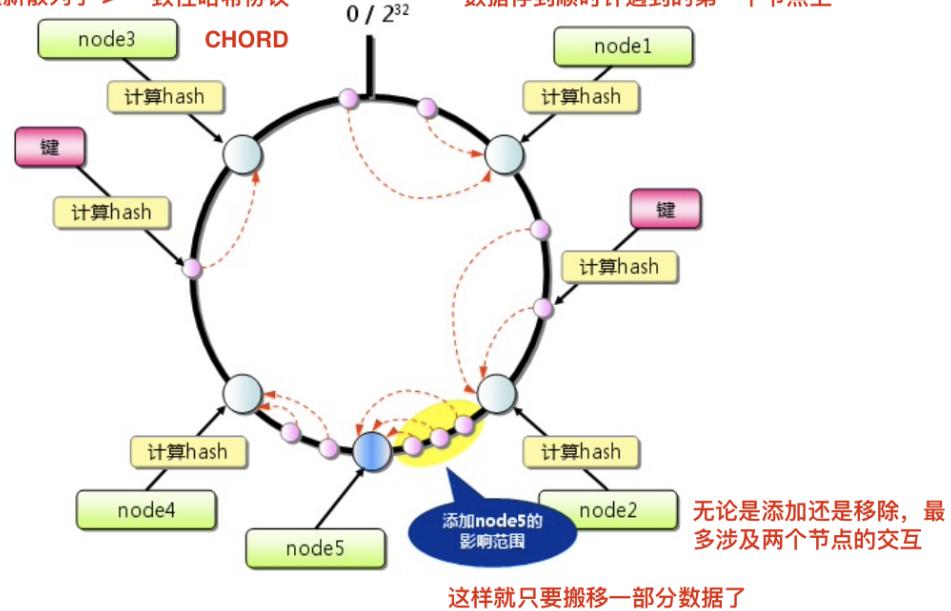


设计不同尺寸的缓存工具。



一致性哈希

如果增加一个机器就要重新散列了-> 一致性哈希协议



## Redis

- Redis is what is called a **key-value store**,
  - often referred to as a **NoSQL** database.
  - The essence of a key-value store is the ability to store some data, called a value, inside a key.
- Redis is an open source, BSD licensed, advanced key-value store.
  - It is often referred to as a data structure server since keys can contain **strings, hashes, lists, sets** and **sorted sets**.
- In order to achieve its outstanding performance, Redis works with an **in-memory dataset**.
  - Depending on your use case, you can persist it either by dumping the dataset to disk every once in a while, or by appending each command to a log.
- Redis also supports trivial-to-setup **master-slave replication**,
  - with very fast **non-blocking** first synchronization, auto-reconnection on net split and so forth.

19

Redis 写硬盘: cache如果放不下了，就在硬盘上做持久化,但是就之前本来就是想避免读外设。原因，前面是json → java, 需要做反序列化。这里直接在硬盘上存java对象，就省去了反序列化的一步，省略了网络开销。而且Redis是在同一个计算机上写，前面是不同的。硬盘上可以存SSD的，而不是hard disk还是快一点。

当数据内存访问非常频繁，且内存足够大，可以一次把一整个表都放到redis.如果写操作比较多的话，就不要放到redis了。

## 适合放的元素

所以redis适合只读的，经常读的。

如果是经常写的不要放redis

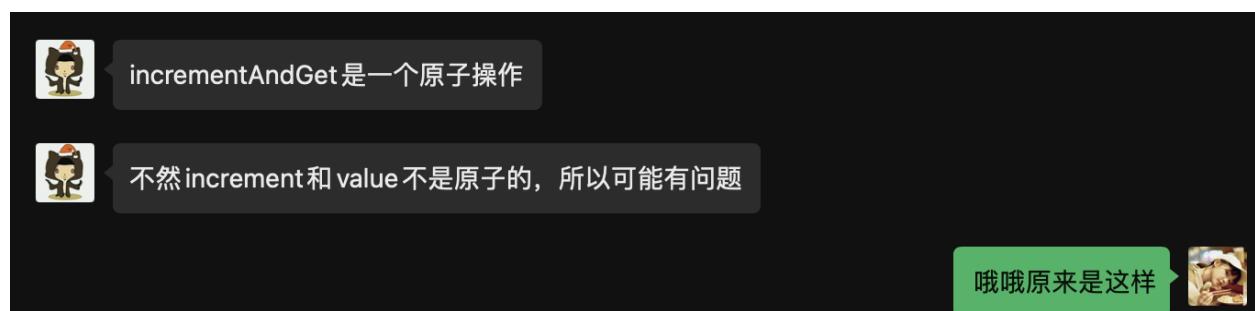
## 什么时候放到redis

当数据内存访问非常频繁，且内存足够大，可以第一次就把一整个表都放到redis.

当数据有写，就要写cache,写到数据库，不能后来再更新数据库，因为这样会有consistency的问题，可能不只有bookStore来访问，有别的应用访问，要把写cache和写数据库变成同一个事务，一个不行就回滚。

## 作业思考

虽然使用了AtomicInteger 和synchronize,但是如果increase和获得值是分开调用的话，其实还是一样的，所以如果两个步骤需要在一起的话，要么放在一个synchronize函数，要么用一个sync



## lect7: Full-text Searching

能够根据业务需求，识别适合全文搜索引擎的场景，设计并实现基于搜索引擎的全文搜索方案，  
lucene, sole, elasticsearch

## 索引的概念

### 1. 定义：

索引Index: a data structure that allows fast random access to words stored inside it.

Searching is the process of looking up words in an index to find documents where they appear.

### 2. 索引建立原则：

1、索引应该经常建在Where子句经常用到的列上。如果某个大表经常使用某个字段进行查询，并且检索行数小于总表行数的5%。则应该考虑。

2、对于两表连接的字段，应该建立索引。如果经常在某表的一个字段进行Order By，则也经过进行索引。

3、不应该在小表上建设索引。

4、索引应该尽可能小（不能老读硬盘，内存和硬盘的换进换出），一般用B+树（树的高度决定了搜索的次数）

3. 索引的使用场景：非结构化的文本，又要搜索。需要全文搜索，如果遍历会很慢，所以去建立索引。

4. 索引衡量标准：找回率（有10个满足要求找到了几个），准确率。优化：是否有高级搜索，and or，结果排序...

## Lucene

1. 建立索引的过程：Index 这个函数才真正在 indexDir 目录(里面是很多很多文本)建立了索引，然后写到了 dataDir 中去。其中的 StandardAnalyzer 就是去断开单词、自动去掉标点符号，最终得到的是一个个去掉标点符号的单词。然后 indexDirectory 就是把建立的索引绑定到了 dataDir 中，只有 writer.optimize() 后才把多线程的结果合并成一个文件，真正写到了硬盘上。（BC）

2. 四种不同的索引

- **Keyword**—Isn't analyzed, but is indexed and stored in the index verbatim.
- **UnIndexed**—Is neither analyzed nor indexed, but its value is stored in the index as is.
- **UnStored**—The opposite of UnIndexed. This field type is analyzed and indexed but isn't stored in the index.
- **Text**—Is analyzed, and is indexed. This implies that fields of this type can be searched against, but be cautious about the field size.

3. 一些细节：只能做增量式的增加

# lect8: Web Services

## web services

能够根据业务需求，识别需要封装为Web服务的业务功能，并能够将其实现为Restful Web服务

### 1. 什么是SOAP？

SOAP、**WSDL**(*WebServicesDescriptionLanguage*)、**UDDI**(*UniversalDescriptionDiscovery andIntegration*)之一，soap用来描述传递信息的格式，WSDL用来描述如何访问具体的接口，uddi用来管理，分发，查询webService。具体实现可以搜索Web Services简单实例；SOAP可以和现存的许多因特网协议和格式结合使用，包括超文本传输协议（HTTP），简单邮件传输协议（SMTP），多用途网际邮件扩充协议（MIME）。它还支持从消息系统到远程过程调用（RPC）等大量的应用程序。SOAP使用基于XML的数据结构和超文本传输协议(HTTP)的组合定义了一个标准的方法来使用Internet上各种不同操作环境中的分布式对象。

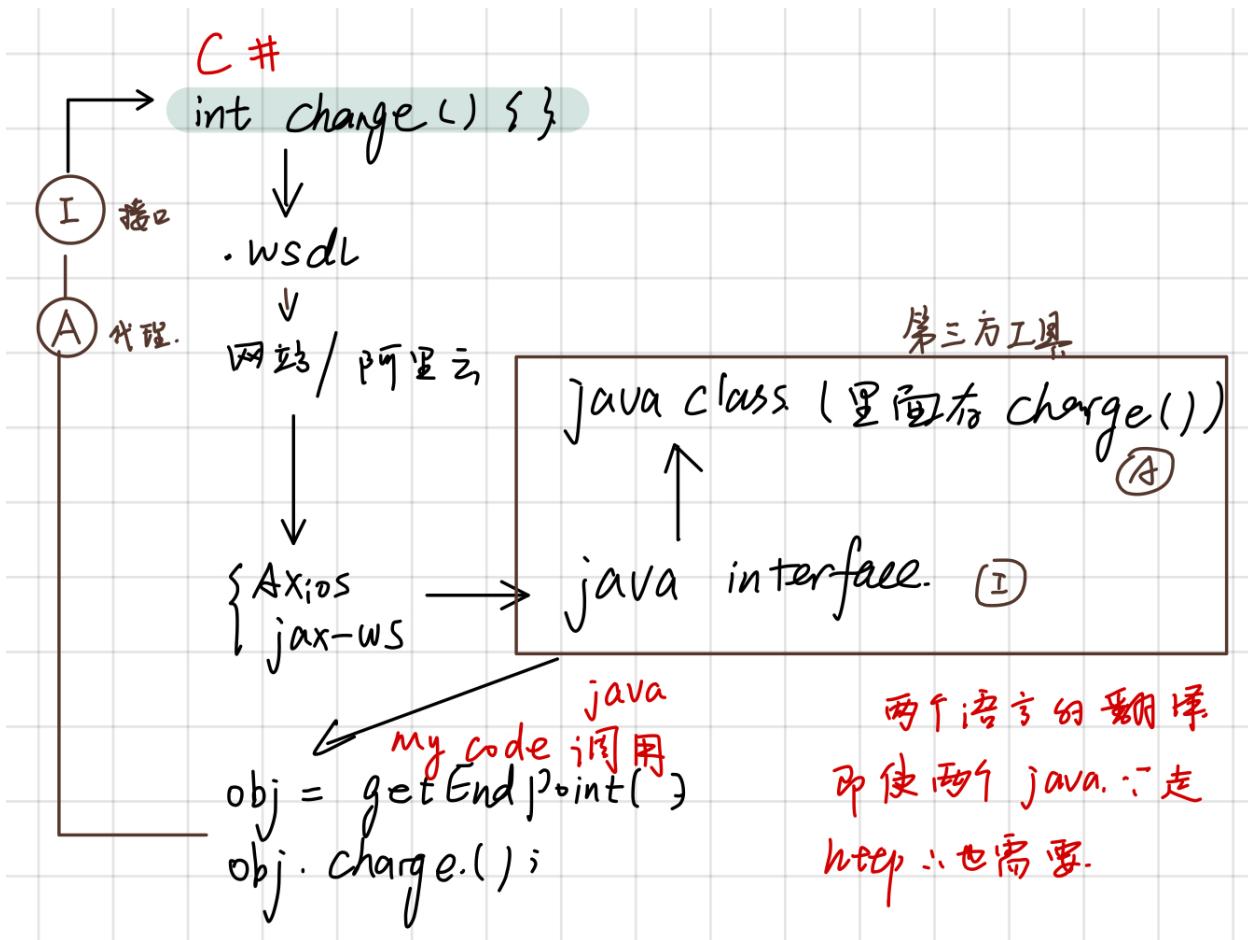
### 2. 什么是web service？

Web Service是一个平台独立的，低耦合的，自包含的、基于可编程的web的应用程序，可使用开放的XML（标准通用标记语言下的一个子集）标准来描述、发布、发现、协调和配置这些应用程序，用于开发分布式的交互操作的应用程序。[1]

Web Service技术，能使得运行在不同机器上的不同应用无须借助附加的、专有的第三方软件或硬件，就可相互交换数据或集成(即独立于具体实现)。依据Web Service规范实施的应用之间，无论它们所使用的语言、平台或内部协议是什么，都可以相互交换数据。Web Service是自描述、自包含的可用网络模块，可以执行具体的业务功能。Web Service也很容易部署，因为它们基于一些常规的产业标准以及已有的一些技术，诸如标准通用标记语言下的子集XML、HTTP。Web Service减少了应用接口的花费。Web Service为整个企业甚至多个组织之间的业务流程的集成提供了一个通用机制。

### 3 为什么要使用web service？

Web Service希望实现不同的系统之间能够用“软件软件对话”的方式相互调用，打破了软件应用、网站和各种设备之间的格格不入的状态，实现“基于Web无缝集成”的目标。



为什么要有 webservice，也就是我们需要以纯文本的格式进行调用。比如我们用 java 写了一个服务，叫做 billing(支付)，这个服务是用 java 写的，能够进行调用。我们怎么样 使用 C# 的程序去调用 java 的这个 billing 服务呢？大家想到了要纯文本，比如 JSON，它的 本质就是在传送输出；还有一种想法，billing 一定会有 api，比如里面有一个函数叫做 pay，C#说想要调用这个 api，然后返回对应的值，因为 C#不能直接调用 pay，需要把调用什么函数、参数是什么这件事情使用纯文本进行描述，这就是 SOAP(简单对象传输协议)。什么样的结构才能让它知道你想要干什么呢？我们需要定义这件事情，SOAP 也实际上在定义这件事情，在调用 pay 函数的时候也要按照固定的格式(xml 格式)。SOAP 相当于去规定在 xml 文件里存在哪些标签、每个标签的子标签能有多少个。

#### 4. WebService 原理

XML, SOAP 和 WSDL 就是构成 WebService 平台的三大技术。

WebService 采用 HTTP 协议来在客户端和服务端之间传输数据。WebService 使用 XML 来封装数据，XML 主要的优点在于它是跨平台的。

WebService 通过 HTTP 协议发送请求和接收结果时，发送的请求内容和结果内容都采用 XML 格式封装，并增加了一些特定的 HTTP 消息头，以说明 HTTP 消息的内容格式，这些特定的 HTTP 消息头和 XML 内容格式就是 SOAP 协议规定的。

WebService 服务器端首先要通过一个 **WSDL** 文件来说明自己有什么服务可以对外调用。简单的说，**WSDL** 就像是一个说明书，用于描述 WebService 及其方法、参数和返回值。WSDL 文件保存在 Web 服务器上，通过一个 url 地址就可以访问到它。客户端要调用一个 WebService 服务之前，要知道该服

务的WSDL文件的地址。WebService服务提供商可以通过两种方式来暴露它的WSDL文件地址：1. 注册到UDDI服务器，以便被人查找；2.直接告诉给客户端调用者。

#### 5. webservice缺陷

- 指明input 和output的格式，一旦server的函数参数改了，重新生成wsdl, 客户端得重新再生成一遍
- webservice需要解析SOAP格式的纯文本，肯定没有两边同种语言通信高。但是webservice的作用是给异构的系统去交互。
- 通讯的过程中要翻译成SOAP消息，他的信息比较冗余，之后出现了restful service.

6. SOAP 和wsdl区别：wsdl和soap虽然是web service的两大标准，但是两者并没有必然的联系，都可以独立使用。wsdl提供了一个统一的接口，目前已经成为一个国际上公认的标准。  
soap（简单对象访问协议）是一种基于http的传输协议，用来封装消息，访问远程服务。soap协议是一种请求和应答协议规范，而http是web传输协议，soap的传输是可以基于http的，但也可以基于其他的传输协议，如ftp、smtp等。

## restful web services

1. 特征：REST式的Web Services架构意味着，对该Web Services进行请求时的方法信息(method information)都在报文（如图1）的请求方法（HTTP method）中，作用域信息(scoping information)都在报文的URL里，服务器在接受到HTTP请求的时候只需要关注请求报文中第一行的请求方法(HTTP method)和URL就能知道用户想做什么事了(如某报文的第一行：GET /data/username/xxx HTTP/1.1)，而报文中其余数据只是一些细节问题。method包括：get, put, post, delete. 这样以数据为中心，那两边就只有纯数据没有API，（数据比API稳定），这样就实现了前端解耦。eg.把之前的getBook, deleteBook, 都改为Book,只是改一下方法get, delete.

restful API无状态。

## lect9: Microservices & Serverless

– 能够根据业务需求，抽象出系统中的微服务，并能够基于Spring框架开发对应的微服务

### microservices

## Microservice architectures are the ‘new normal’.

- Building small, self-contained, ready to run applications can bring great flexibility and added resilience to your code.
- Spring Boot’s many purpose-built features make it easy to build and run your microservices in production at scale.
- And don’t forget, no microservice architecture is complete without [Spring Cloud](#) – easing administration and boosting your fault-tolerance.

## What are microservices?

- Microservices are a modern approach to software whereby **application code is delivered in small, manageable pieces, independent of others.**

## Why build microservices?

- Their small scale and relative isolation can lead to many additional benefits, such as **easier maintenance, improved productivity, greater fault tolerance, better business alignment**, and more.

### 1. 微服务的特点：

它的主要特点是组件化、松耦合、自治、去中心化，体现在以下几个方面：

**\*\*A. 细粒度的服务分解 \*\***

服务粒度要小，而每个服务是针对一个单一职责的业务能力的封装，专注做好一件事情。

**\*\*B. 独立部署运行和扩展 \*\***

每个服务能够独立被部署并运行在一个进程中。这种运行和部署方式能够赋予系统灵活的代码组织方式和发布节奏，使得快速交付和应对变化成为可能。

**\*\*C. 独立开发和演化 \*\***

技术选型灵活，不受遗留系统技术约束。合适的业务问题选择合适的技术可以独立演化。服务与服务之间采取与语言无关的API进行集成。相对单体架构，微服务架构是更面向业务创新的一种架构模式。

**\*\*D. 独立团队和自治 \*\***

团队对服务的整个生命周期负责，工作在独立的上下文中，自己决策自己治理，而不需要统一的指挥中心。团队和团队之间通过松散的社区部落进行衔接。

### 2. 微服务需要考虑的内容：

服务解耦、数据库独立、消息队列、故障处理、监控-发现故障、定位问题-链路追踪、日志收集、分析问题-日志分析、网关、服务注册与发现-动态扩容

细节：

**微服务架构整个应用分散成多个服务，定位故障点非常困难。**

**微服务架构中，一个服务故障可能会产生雪崩效应，导致整个系统故障。**

拆分成微服务后，出现大量的服务，大量的接口，使得整个调用关系乱糟糟的。经常在开发过程中，写着写着，忽然想不起某个数据应该调用哪个服务。为了应对这些情况，微服务的调用需要

一个把关的东西，也就是网关。在调用者和被调用者中间加一层网关，每次调用时进行权限校验。另外，网关也可以作为一个提供服务接口文档的平台。

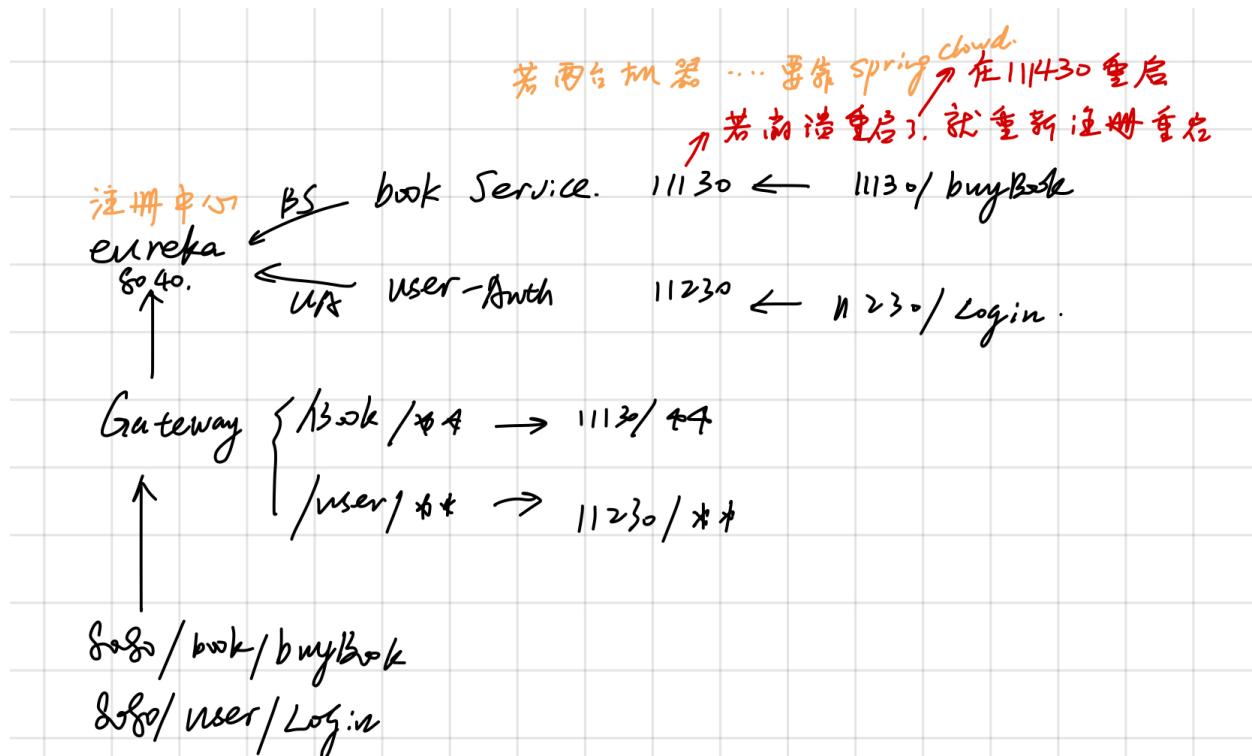
一般来说新增实例的操作为：

1. 部署新实例
2. 将新实例注册到负载均衡或DNS上

第二部人工做麻烦 → 解决这个问题的方案是服务自动注册与发现。

## Eureka & routing

1. 作用：负载均衡，发现中间件（middle-tier server）的故障



通过gateway,eureka 实现了注册和网关，对用户的动态路由

## serverless

## What is serverless?

- Serverless workloads are “**event-driven**” workloads that aren’t concerned with aspects normally handled by server infrastructure.”
- Concerns like “how many instances to run” and “what operating system to use” are all managed by a **Function as a Service platform (or FaaS)**, leaving developers free to focus on business logic.

## Serverless characteristics?

- Serverless applications have a number of specific characteristics, including:
  - Event-driven code execution with triggers
  - Platform handles all the starting, stopping, and scaling chores
  - Scales to zero, with low to no cost when idle
  - Stateless

Serverless (无服务器架构) 指的是由开发者实现的服务端逻辑运行在无状态的计算容器中，它由事件触发，完全被第三方管理，其业务层面的状态则被开发者使用的数据库和存储资源所记录。

## function service

- [Spring Cloud Function](#)
  - provides capabilities that lets Spring developers take advantage of serverless or FaaS platforms.
- [Spring Cloud Function](#)
  - provides adaptors so that you can run your functions on the most common FaaS services including :
    - [Amazon Lambda](#), [Apache OpenWhisk](#), [Microsoft Azure](#), and [Project Riff](#).
- [Spring Cloud Function](#) is a project with the following high-level goals:
  - Promote the implementation of business logic via functions.
  - Decouple the development lifecycle of business logic from any specific runtime target so that the same code can run as a web endpoint, a stream processor, or a task.
  - Support a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS).
  - Enable Spring Boot features (auto-configuration, dependency injection, metrics) on serverless providers.

云提供 Serverless，我们不需要去处理事务管理、安全管理，云的基础设施帮我们搞定，我们只需要关注业务开发。整个服务应该是无状态的。如果云提供的环境足够好的话，我们应用特有的逻辑就可以只写成函数。

## lect10: security

### Message digest 消息摘要

Message Digests : A message digest is a digital fingerprint of a block of data

## A message digest has two essential properties:

- If one bit or several bits of the data are changed, then the message digest also changes.
- A forger who is in possession of a given message cannot construct a fake message that has the same message digest as the original.

1. 固定长短20个字节
2. 就算数据里面改了一点，摘要大相径庭
3. 生成算法是固定的。
4. 数据给对方之前生成摘要，给对方数据。生成算法是固定的，对方就拿数据生成摘要，如果两边的摘要是一样的，那就说明数据在中间没有被篡改。

缺陷：如果把律师买通了把摘要也改了，中间人改完之后两边就对上了。

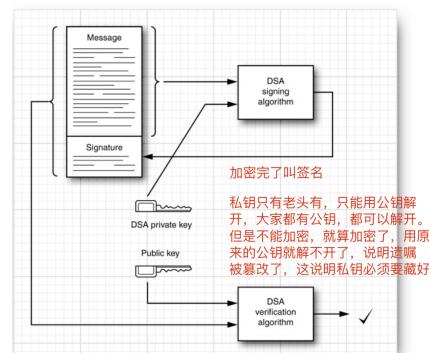
原因：The message digest algorithms are publicly known, and they don't require secret keys.

解决方法：非对称加密，message signing

## 非对称加密

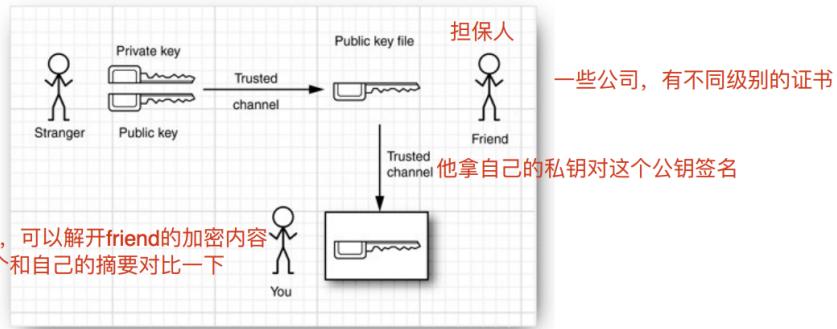
性质：

1. 公钥加密只能拿私钥解密
2. 私钥加密只能用公钥解密
3. 公钥和私钥之间无法互推
4. 私钥是自己存，公钥是导出分发给别人的



## Be careful:

- You still have no idea who wrote the message. Anyone could have generated a pair of public and private keys, signed the message with the private key, and sent the signed message and the public key to you.
- The problem of determining the identity of the sender is called the authentication problem.



因为任何人都可以写公钥私钥，所以我们需要CA

[alice\\_signedby\\_acmeroot.cer](#)

- Now Cindy imports the signed certificate into her keystore: 签名之后的
  - `keytool -importcert -keystore cindy.certs -alias alice -file alice_signedby_acmeroot.cer`这里需要校验，看Cindy的证书库里面是否有可以解开这个的，如果没有说明这个是不被信任的，就导入不成功了。这就实现了陌生人之间的信息传递。

32

传递秘钥的过程

- Create a keystore `acmesoft.certs`.
- Generate a key pair and export the public key:
  - `keytool -genkeypair -keystore acmesoft.certs -alias acmeroot`
  - `keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer`
- The public key is exported into a "self-signed" certificate.
- Then add it to every employee's keystore.
  - `keytool -importcert -keystore cindy.certs -alias acmeroot -file acmeroot.cer`
- An authorized staff member at ACME Software would verify Alice's identity and generate a signed certificate as follows:
  - `java CertificateSigner -keystore acmesoft.certs -alias acmeroot -infile alice.cer -outfile alice_signedby_acmeroot.cer`
- Now Cindy imports the signed certificate into her keystore:
  - `keytool -importcert -keystore cindy.certs -alias alice -file alice_signedby_acmeroot.cer`

## Symmetric Ciphers 对称加密

加密和解密用同样的秘钥，这样安全性稍微弱，但是效率高。但是我们可以结合两者：先使用非对称密钥通信(公钥/私钥)传递一个对称密钥，之后双方都是用对称密钥对数据进行加密/解密。在对称密钥分发的过程中，是安全的，哪怕有人截获了他也解不开，不知道传递的是什么。

对称加密算法比如 AES，在生成实例的时候，一般要传递一个随机数。我们可以使用 SecureRandom 来生成更加安全的随机数。

## https 三次握手

### 第一步：证书验证

客户端向服务器发送 HTTPS 请求，等待服务器确认。

服务器将 crt 公钥以证书的形式发送到客户端，该证书还包含了用于认证目的的服务器标识，服务器同时还提供了一个用作产生密钥的随机数。服务器端存放 crt 私钥和 crt 公钥。

客户端验证证书是否合法，如果不合法则提示告警。

### 第二步：获取对称密钥

客户端用随机数和 hash 签名生成一串对称密钥（即随机钥，客户钥），然后用 crt 公钥对对称密钥进行加密。

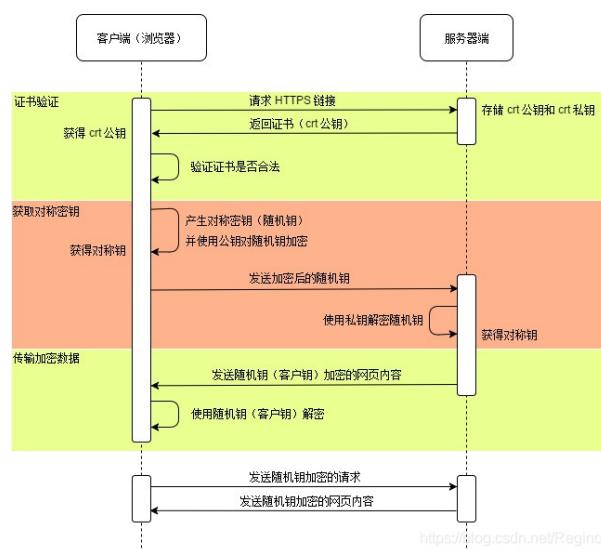
客户端将加密后的对称密钥发送给服务器。

服务器用 crt 私钥解密，取出对称密钥。

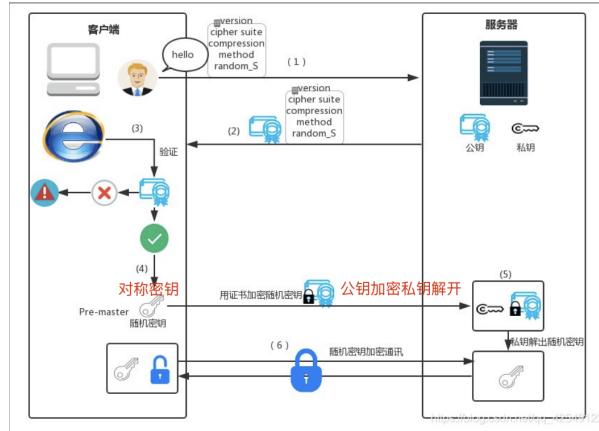
### 第三步：传输加密数据

服务器用随机钥来加密数据，发送加密的网页内容。

客户端通过本地存储的随机钥对密文进行解密并判断是否被篡改，如果没有篡改，后面的数据通讯将使用对称加密传输所有内容。



HTTPS 在内容传输的加密上使用的是对称加密，非对称加密只作用在证书验证阶段。因为非对称加密的加解密效率是非常低的，而 HTTP 的应用场景中通常端与端之间存在大量的交互，非对称加密的效率是无法接受的。



## 单点认证 Single sign-on (SSO)

只要一个地方认证，就可以传播到其他应用去。

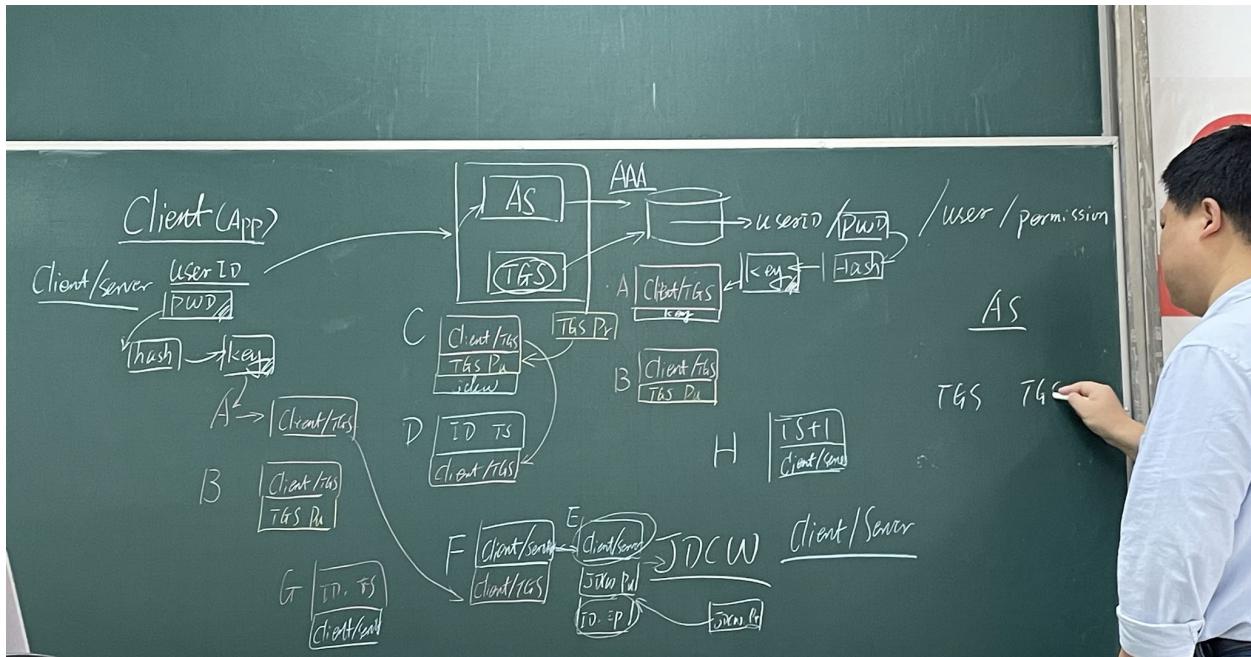
Single sign-on (SSO) is a property of access control of multiple related, but independent software systems.

好处：

### Benefits include:

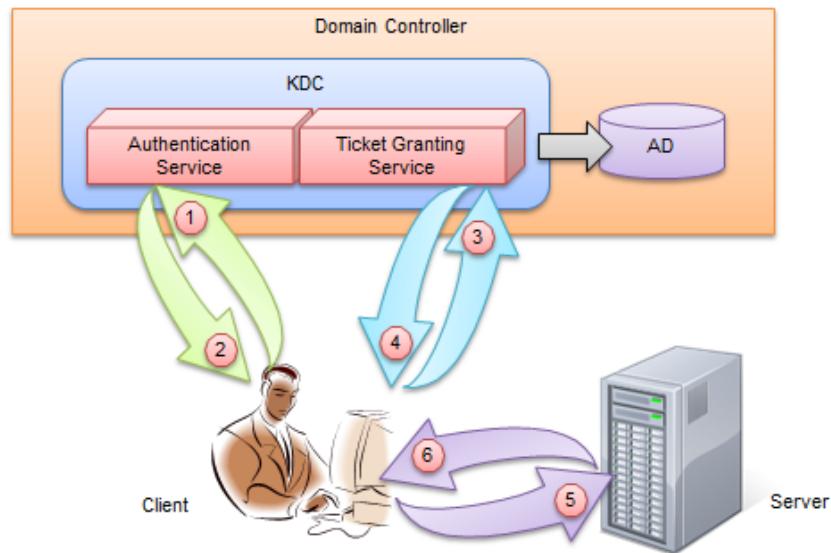
- Reduces phishing success, because users are not trained to enter password everywhere without thinking.
- Reducing password fatigue from different user name and password combinations
- Reducing time spent re-entering passwords for the same identity
- Reducing IT costs due to lower number of IT help desk calls about passwords
- Security on all levels of entry/exit/access to systems without the inconvenience of re-prompting users
- Centralized reporting for compliance adherence.

## Kerberos



## kerberos 认证流程

<https://www.cnblogs.com/zhangwuji/p/9154153.html>



### 1. KDC

全称：key distributed center

作用：整个安全认证过程的票据生成管理服务，其中包含两个服务，AS和TGS

### 2. AS

全称：authentication service

作用：为client生成TGT的服务

### 3. TGS

全称：ticket granting service

作用：为client生成某个服务的ticket

### 4. AD

全称：account database

作用：存储所有client的白名单，只有存在于白名单的client才能顺利申请到TGT

### 5. TGT

全称：ticket-granting ticket

作用：用于获取ticket的票据

### 6.client

想访问某个server的客户端

### 7. server

提供某种业务的服务

图1展示了kerberos的认证流程，总体分为3步。

1. client与AS交互
2. client与TGS交互
3. client与server交互

## 详细分析

kerberos为什么要采用3步交互的形式来完成安全认证，那就要从kerberos的使用场景说起。

相比kerberos，https可能更为熟悉一点，通过证书和非对称加密的方式，让客户端可以安全的访问服务端，但这仅仅是客户端安全，通过校验，客户端可以保证服务端是安全可靠的，而服务端却无法得知客户端是不是安全可靠的。这也是互联网的一种特性。而kerberos可以支持双向认证，就是说，可以保证客户端访问的服务端是安全可靠的，服务端回复的客户端也是安全可靠的。

想证明client和server都是可靠的，必然要引入第三方公证平台，这里就是AS和TGS两个服务。

1. client向kerberos服务请求，希望获取访问server的权限。kerberos得到了这个消息，首先得判断client是否是可信赖的，也就是白名单黑名单的说法。这就是AS服务完成的工作，通过在AD中存储黑名单和白名单来区分client。成功后，返回AS返回TGT给client。
2. client得到了TGT后，继续向kerberos请求，希望获取访问server的权限。kerberos又得到了这个消息，这时候通过client消息中的TGT，判断出了client拥有了这个权限，给了client访问server的权限ticket。

3. client得到ticket后，终于可以成功访问server。这个ticket只是针对这个server，其他server需要像TGS申请。

通过这3步，一次请求就完成了。当然这里会有个问题，这样也没比https快啊。解释一下

1. 整个过程TGT的获取只需要一次，其中有超时的概念，时间范围内TGT都是有效的，也就是说一般情况访问server只需要直接拿到ticket即可
2. 整个过程采用的是对称加密，相对于非对称加密会有性能上的优势
3. kerberos的用户管理很方便，只需要更新AD中的名单即可

当然整个过程的通信都是加密的，这里设计到两层加密，因为所有的认证都是通过client，也就是说kerberos没有和server直接交互，这样的原因是kerberos并不知道server的状态，也无法保证同时和server，client之间通信的顺序，由client转发可以让client保证流程顺序。

第一层加密，kerberos对发给server数据的加密，防止client得到这些信息篡改。

第二层加密，kerberos对发给client数据的加密，防止其他网络监听者得到这些信息。

client和server的通信也是如此。

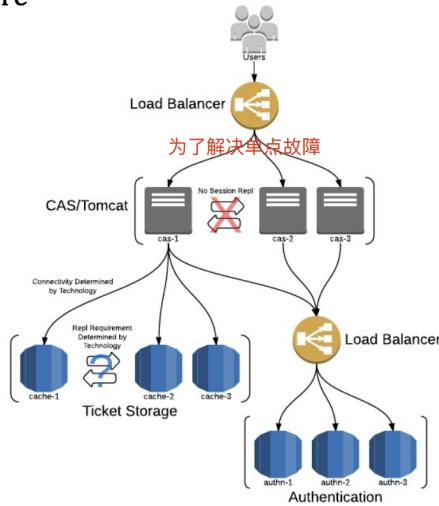
## Drawbacks and Limitations

- Single point of failure.
- Kerberos has strict time requirements, which means the clocks of the involved hosts must be synchronized within configured limits.
- The administration protocol is not standardized and differs between server implementations.
- Since all authentication is controlled by a centralized KDC, compromise of this authentication infrastructure will allow an attacker to impersonate any user.
- Each network service which requires a different host name will need its own set of Kerberos keys. This complicates virtual hosting and clusters.

缺陷是AS TGS 成为了单一故障节点，所以需要用分布式于是有了CAS。

CAS 就是基于 Kerberos 的实现，它为了解决单点故障的问题，就去做了集群，前面有多个 load balancer。

## Architecture



## Security Characteristics

### Nonrepudiation

所有的东西都要留下痕迹，如果黑客来黑能不能找到源头。比如同学在开电子书店，我们一下子下了一万块的单，然后他去进货了。也就是你在网站上的所有动作不能否认。比如我们所有的数据都通过你的私钥加密，那么如果能用公钥解密，那么就一定是你做的。

### Confidentiality

- is the property that data or services are protected from unauthorized access.
- This means that a hacker cannot access your income tax returns on a government computer.  
对方的公钥加密只能用对方的私钥解开。

### Integrity

- – is the property that data or services are being delivered as intended.
- – This means that your grade has not been changed since your instructor assigned it.  
数据在传输的时候需要是完整的，不能丢东西。那就是生成一个摘要 (digest)，如果数据的摘要不相等，那么我们很快就知道内容不一样了。

### Assurance

保证不是钓鱼网站

- – is the property that the parties to a transaction are who they purport to be.
- – This means that, when a customer sends a credit card number to an Internet merchant, the merchant is who the customer thinks they are.

### Availability

未授权的用户进来不能搞破坏

- is the property that the system will be available for legitimate use.
- This means that a denial-of-service attack won't prevent your ordering this book.

## Auditing

log很重要，需要有分析，可视化的工具

- is the property that the system tracks activities within it at levels sufficient to reconstruct them.
- This means that, if you transfer money out of one account to another account, in Switzerland, the system will maintain a record of that transfer.

## 安全策略Security Tactics

Tactics for achieving security can be divided into

- those concerned with **resisting attacks**,
- those concerned with **detecting attacks**,
- and those concerned with **recovering from attacks**.

### Security Tactics-resisting attacks



- we identified
  - **nonrepudiation, confidentiality, integrity**, and **assurance** as goals in our security characterization.
- The following tactics can be used in combination to achieve these goals.
  - **Authenticate users**.
  - **Authorize users**.
  - **Maintain data confidentiality**.
    - Encryption
    - Communication links
      - virtual private network (VPN)
      - Secure Sockets Layer (SSL)
  - **Maintain integrity**.
    - checksums
    - hash results
  - **Limit exposure**
  - **Limit access**
    - Firewalls

- The detection of an attack is usually through an **intrusion detection system**.
  - Such systems work by comparing network traffic patterns to a database.
  - In the case of misuse detection, the traffic pattern is compared to historic patterns of known attacks.
  - In the case of anomaly detection, the traffic pattern is compared to a historical baseline of itself.
  - Frequently, the packets must be filtered in order to make comparisons.
  - Filtering can be on the basis of protocol, TCP flags, payload sizes, source or destination address, or port number.
- Intrusion detectors must have
  - some sort of sensor to detect attacks,
  - managers to do sensor fusion,
  - databases for storing events for later analysis,
  - tools for offline reporting and analysis,
  - and a control console so that the analyst can modify intrusion detection actions.

- Tactics involved in recovering from an attack can be divided into
  - those concerned with **restoring state** and
  - those concerned with **attacker identification**.
- The tactics used in restoring the system or data to a correct state overlap with those used for availability
  - since they are both concerned with **recovering a consistent state from an inconsistent state**.
- The tactic for identifying an attacker is
  - **to maintain an audit trail**.

其他：

扩展脚本攻击， 比如在书评写<script>...</script>

方法：1. 过滤输入。把所有的<,>， 转译成&gt; 2. 前端展示用富文本展示。

拒绝服务攻击， 让很多机器发请求把服务器搞死。

需要：防范攻击， 入侵检测（有人大量登陆请求， 撞户）， 保险， 有限暴露（接口和实现分离）， 防火墙

## lect11: MySQL Optimization

- 能够根据数据访问的具体场景，设计数据库，包括数据库结构和索引

### 1. 数据库设计需要考虑的问题

## 数据库问题

id	name	author	introduction
	null / NOT NULL		

Text / Blob 有指针 → 指向 introduction 文件  
的位置。

Vchar.

① 看是否分两张表，看存储大的字段是否能串用，

∴ 要以 DB load 到内存

② 存什么

③ 类型：NULL, NOT NULL

↳ 怎么索引，还有一位存 0/1 皇否空。

④ 怎么建索引，4列建 4个索引 or 复合索引。

要根据 query 建立

∴ 在 CRUD 时，索引都要改 ∴ 用复合索引较好。

⑤ 行存储来压缩节省空间，但以后 speed ↓

⑥ 缓存的尺寸要开多大呢？

- The most important factor in making a database application fast is its basic design:
  - Are the tables structured properly? In particular, do the columns have the right data types, and does each table have the appropriate columns for the type of work?
  - Are the right indexes in place to make queries efficient?
  - Are you using the appropriate storage engine for each table, and taking advantage of the strengths and features of each storage engine you use?
  - Does each table use an appropriate row format?
  - Does the application use an appropriate locking strategy?
  - Are all memory areas used for caching sized correctly?

## 硬件层优化：行存or列存？

1. 按照行还是列存呢？要做在线分析处理（OLAP），需要列存，比如要找所有人的同一个数据。OLTP（on-line transaction processing），对每行做处理。

如果两个都要就是HTAP(hybrid)

方法1：存两种形式（行，列），这样有同步问题，且空间大

方法2：行和列转换，技术难度大

## 索引的性质

- 要建立一个比较平衡的二叉索引树，搜索效率就是看树的高度。但是数据太多，需要改为B树，但是问题是它的查找效率不太稳定，改为B+ 树，所有的关键字都在叶子节点，可以放多个key，最后一个key又指向了下一个叶子节点，这个链接就是为了范围查找。
- 索引就是这些需要查找的值来做一棵树，一个duplicate，这样就很大。所以索引要尽量小，否则会大量的页在内存的换进换出，所以索引的值一定不能是特别长的取值，比如拿book的introduction文字做索引。

### 如何使用索引提高select 的效率？

建立联合索引，但是要注意性能。

比如我们在 title、author、price 上都建立了一个索引(建立索引就是复制出来建立一棵树)。我们是建立四个索引合适呢，还是建立包含四个的复合索引合适呢？建立复合索引肯定省空间，比如我们建立了 title:author:price:introduction，但是在我们 select 的时候，如果直接查询 author = ...，或者 title = ... or author = ...。那么我们的索引就失效了，这就要求我们需要根据业务环境 query 的形式就去建立什么样的索引。

为什么复合索引更好一点呢？我们在数据库中 CRUD 的时候，索引也需要同步更新。更新一个索引(树)的效率肯定大于更新四个索引(树)的效率。

The best way to improve the performance of **SELECT** operations is

- to create indexes on **one or more of the columns** that are tested in the query.
- The index entries act like **pointers to the table rows**, allowing the query to quickly determine which rows match a condition in the **WHERE** clause, and retrieve the other column values for those rows.
- **All** MySQL data types can be indexed.

Although it can be tempting to create an indexes for every possible column used in a query,

- **unnecessary indexes waste space and waste time** for MySQL to determine which indexes to use.
- Indexes also add to the **cost** of inserts, updates, and deletes because each index must be updated.
- You must find the **right balance** to achieve fast queries using the optimal set of indexes.

## mysql 使用index

- 索引在多个列上建立，有这样的需求（比如高级查找）
- 索引有很多种，比如基于地理位置的索引，有大量的计算，这肯定不是B+树，有专门的叫做R tree.
- 数据量小的时候就没必要建立索引。
- 建立索引的原则：按照经常查找的建立索引，一般数据库会按照主键 (id)做索引。但是这个主键是用户都不知道的信息。原因是可能有另外一张表有外键，引用id，在做Join操作的时候，就可以快速定位了。
- 除了空间数据 → R tree , memory table InnoDB ， 其他都用B-

## SPATIAL Index Optimization



- MySQL permits creation of SPATIAL indexes on **NOT NULL geometry-valued** columns.
  - For comparisons to work properly, each column in a **SPATIAL** index must be SRID-restricted.
  - That is, the column definition must include an **explicit SRID attribute**, and all column values must have the same SRID.
- The optimizer considers SPATIAL indexes **only for SRID-restricted columns**:
  - Indexes on columns restricted to a Cartesian SRID enable Cartesian bounding box computations.
  - Indexes on columns restricted to a geographic SRID enable geographic bounding box computations.

## 索引设计原则

- Indexes are used to find rows with specific column values quickly.
  - 如果一个索引搜出来一堆东西是有害的，最好的索引是能定位到唯一的元素。eg. 主键建立索引是唯一的。

- 索引不能太大，比如拿整个introduction来做，这有大量的页的换进换出，就使用前缀，比如前10个字符。
- 对于较小的表，建立索引的意义不大。但是如果大表每个元素都用到也不好，最好的是一张很大的表里面对几个元素进行查询。

Indexes are **less important** for queries on **small tables**,

- or **big tables** where report queries process **most or all of the rows**.
- When a query needs to access most of the rows, reading sequentially is **faster** than working through an index. Sequential reads minimize disk seeks, even if not all the rows are needed for the query.

- 主键挺特别适合做索引The primary key for a table represents
  - 主键非空，可以唯一标识一行元素。
  - 做外键关联的时候经常是和另外一个表的主键关联的这样用主键就很有优势
  - innodb在硬盘里面也是按照主键的顺序一行一行存储
  - 可以多个字段合起来当主键索引
  - 因为不知道其他字段是不是可以是唯一的，那么不如自动生成的。但其实可以自己可以设定主键。Key可以由很多列构成，如果太多很冗余，并且自己定义很难保证是不是唯一的，主键管理很麻烦。而且这些数字索引比较简单，之后做外键关联的时候也很方便。
- If your table is big and important,
  - but does **not** have an obvious column or set of columns to use as a **primary key**,
  - you might create a **separate column** with **auto-increment** values to use as the primary key.
  - These unique IDs can serve as pointers to corresponding rows in other tables when you join tables using **foreign keys**.
- 主键除了自增还可以UUID，当大量请求的时候，可能也有几个服务器集群，基于整数递增的id可能重复，因为插入两个不同的表。但是UUID是用机器IP + timestamp + 进程PID+当前对象ID+ 随机数，凑成一个32位的16进制数 (16byte) 几乎不会重复。
  - 缺陷：uuid太大了，索引比较耗费空间。不可能知道插入顺序，还需要记录时间戳。
  - 平衡：是需要性能，还是需要分布式不能出错，看需求。

### B-tree VS hash索引

B- 可以范围查找，like类型操作

hash的直接查找快，但是范围查找X

## B-Tree Index Characteristics

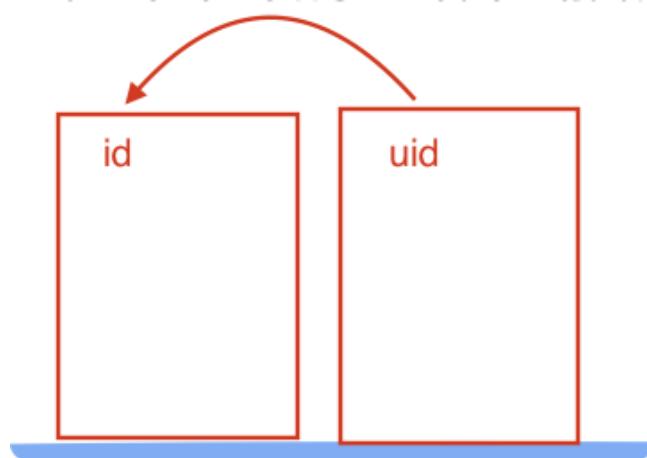
- A B-tree index can be used for column comparisons in expressions that use the `=`, `>`, `>=`, `<`, `<=`, or `BETWEEN` operators.
- The index also can be used for `LIKE` comparisons if the argument to `LIKE` is a constant string that does not start with a wildcard character.

## Hash Index Characteristics

- They are used only for equality comparisons that use the `=` or `<=>` operators (but are *very fast*). They are **not used for comparison** operators such as `<` that find a range of values.
- Systems that rely on this type of **single-value lookup** are known as “**key-value stores**”; to use MySQL for such applications, use hash indexes wherever possible.
- The optimizer **cannot** use a hash index to speed up `ORDER BY` operations. (This type of index cannot be used to search for the **next entry in order**.)
- MySQL **cannot** determine approximately **how many rows** there are **between two values** (this is used by the range optimizer to decide which index to use).
- Only **whole** keys can be used to search for a row.

## Foreign Key Optimization

- 一张大表有一些数据不太被访问
  - 将经常访问的信息和不太访问的信息虽然语义上关联，拆开成两个表，用uid和id关联。不访问的加一个UID指向第一张表，这样下次需要load的时候，就可以一次加载大量经常访问的数据。剩下的数据（图片文本）可能还不存在MongoDB别的数据库。



- Indexing only a prefix of column values in this way can make the index file much **smaller**. When you index a **BLOB** or **TEXT** column, you **must** specify a prefix length for the index. For example:  
`CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));` **BLOB** 字符串的指针

如果建立10个前缀的索引，但是需要查20个字节的，就MySQL先找10个字节匹配的，再在这些里面找20个字节匹配的。

- Index Prefixes**

- With **col\_name(N)** syntax in an index specification for a string column, you can create an index that uses only the **first N characters of the column**.
  - Indexing only a prefix of column values in this way can make the index file much **smaller**. When you index a **BLOB** or **TEXT** column, you **must** specify a prefix length for the index. For example:  
`CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));` Blob: 字符串的地址
- Prefixes can be up to **767 bytes** long for InnoDB tables that use the **REDUNDANT** or **COMPACT** row format.  
mysql 对索引有压缩
- The prefix length limit is **3072 bytes** for InnoDB tables that use the **DYNAMIC** or **COMPRESSED** row format.
- For MyISAM tables, the prefix length limit is **1000 bytes**.
- If a search term **exceeds the index prefix length**, the index is used to exclude non-matching rows, and the remaining rows are examined for possible matches.

如果建立10个前缀的索引，但是需要查20个字节的，就MySQL先找10个字节匹配的，再在这些里面找20个字节匹配的。

## Multiple-Column Index

多列上建立一个索引，组合索引，最多16个key，key没有限制。这里的**顺序**极其重要。和KDtree很像，第一个以lastname分裂，第二层按照firstname分裂，所以输入的顺序必须是这样的。所以需要设计索引满足查询条件，比如查询是顺序的。

- Suppose that a table has the following specification:

```
CREATE TABLE test (
    id INT NOT NULL,
    last_name CHAR(30) NOT NULL,
    first_name CHAR(30) NOT NULL,
    PRIMARY KEY (id),
    INDEX name (last_name,first_name));
```

顺序

- Therefore, the **name** index is used for lookups in the following queries:

```
SELECT * FROM test
    WHERE last_name='Jones'; ✓
SELECT * FROM test
    WHERE last_name='Jones' AND first_name='John';
SELECT * FROM test
    WHERE last_name='Jones' AND (first_name='John' OR first_name='Jon');
SELECT * FROM test
    WHERE last_name='Jones' AND first_name >='M' AND first_name < 'N';
```

- A multiple-column index can be considered a sorted array, the rows of which contain values that are created by concatenating the values of the indexed columns.

- Suppose that a table has the following specification:

```
CREATE TABLE test (
    id INT NOT NULL,
    last_name CHAR(30) NOT NULL,
    first_name CHAR(30) NOT NULL,
    PRIMARY KEY (id),
    INDEX name (last_name,first_name));
```

和KDtree 很像，第一个以lastname 分裂，  
第二层按照firstname分裂，所以输入的顺序  
必须是这样的

顺序

- However, the **name** index is **not** used for lookups in the following queries:

```
SELECT * FROM test
    WHERE first_name='John'; ✗
SELECT * FROM test
    WHERE last_name='Jones' OR first_name='John'; ↗
```

把last\_name 给跳过了  
只要找first name必须经过找last name

同时复合索引也会遇到JPA的问题

- JPA-OR映射需要完成tomcat 和mysql的通信，需要涉及对象和数据的相互转化，性能一定没有JDBC好，
- index是你建立的，如何保证sql 语句翻译的准确性（顺序）。

## B-Tree Index Characteristics

- The following WHERE clauses use indexes:

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3
/* index = 1 OR index = 2 */
... WHERE index=1 OR A=10 AND index=2 ← 优先级and > or
/* optimized like "index_part1='hello'" */
... WHERE index_part1='hello' AND index_part3=5
/* Can use index on index1 but not on index2 or index3 */
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
```

拿出index=1,3

- These WHERE clauses do **not** use indexes:

```
/* index_part1 is not used */
... WHERE index_part2=1 AND index_part3=2
/* Index is not used in both parts of the WHERE clause */
... WHERE index=1 OR A=10      这里用索引也没用，因为A所以需要扫描整张表
/* No index spans all rows */
... WHERE index_part1=1 OR index_part2=10
```

## Hash Index Characteristics

之后可能不知道中间有几个了？所以索引基本都是升序的。

## 建立多个升降序列组合索引

Consider the following table definition

```
CREATE TABLE t (
    c1 INT, c2 INT,
    INDEX idx1 (c1 ASC, c2 ASC),      建立多个索引。
    INDEX idx2 (c1 ASC, c2 DESC),
    INDEX idx3 (c1 DESC, c2 ASC),
    INDEX idx4 (c1 DESC, c2 DESC)
);
```

```
ORDER BY c1 ASC, c2 ASC -- optimizer can use idx1
ORDER BY c1 DESC, c2 DESC -- optimizer can use idx4
ORDER BY c1 ASC, c2 DESC -- optimizer can use idx2
ORDER BY c1 DESC, c2 ASC -- optimizer can use idx3
```

这样拿出来之后就不用排序了。

## Optimizing Database Structure

As when tuning application code,

- you minimize I/O, 每次读多到内存
- keep related items together,
- and plan ahead so that performance stays high as the data volume increases.

## Optimizing Data Size 行存储

- Design your tables to **minimize their space on the disk**.
  - This can result in huge improvements by **reducing the amount of data written to and read from disk**.
  - Smaller tables normally require **less main memory** while their contents are being actively processed during query execution.
  - Any space reduction for table data also results in **smaller indexes** that can be processed faster.
- You can get better performance for a table and minimize storage space by using the techniques listed here:
  - Table Columns
  - Row Format
  - Indexes
  - Joins
  - Normalization

Normalization 范式，范式话程度越高，数据的冗余越少，说明数据库之间的外键关联特别多。比如order 和 orderItem,只存扩展数据。其实全部存在一起压缩了也还好，因为join操作特别慢的，可以获得较大的性能提升。

减小存储的空间，需要压缩。比如多列重复很多，比如学号很多前面是重复的，比如句子的存储是varchar (xx)还是存指针指向文件。到底是两张表还是一张。

Normalization 范式，范式话程度越高，数据的冗余越少，说明数据库之间的外键关联特别多，需要频繁的做join操作才能拿到，但是join太慢了。比如order 和 orderItem,只存扩展数据。其实全部存在一起压缩了也还好，因为join操作特别慢的，可以获得较大的性能提升。

### • Table Columns

- Use the most efficient (**smallest**) data types possible. MySQL has many specialized types that save disk space and memory.
  - For example, use the smaller integer types if possible to get smaller tables. **MEDIUMINT** is often a better choice than **INT** because a **MEDIUMINT** column uses 25% less space.
- Declare columns to be **NOT NULL** if possible. 否则需要增加一个bit来标志是否为空。
  - It makes SQL operations faster, by enabling better use of indexes and eliminating overhead for testing whether each value is NULL.
  - You also save some storage space, one bit per column.
  - If you really need NULL values in your tables, use them. Just avoid the default setting that allows NULL values in every column.

尽量使用NOT NULL (10个字节) 否则需要增加一个bit来标志是否为空。

有很多压缩算法：1. 有好多一样的数据，存哪些是一样的 2. 存偏移量而不是每个值

population

1 1 1 0 3403

+300

-100

compact: VCHAR (1000) , 可以检测有多少字符再加一个数字存储了。这个不算压缩。

增删改查也要改索引，所以索引太多长会变慢。

组合索引比每一列建立好，但是最左匹配方式带来问题，所以越靠组左边的列是越常查询的列。

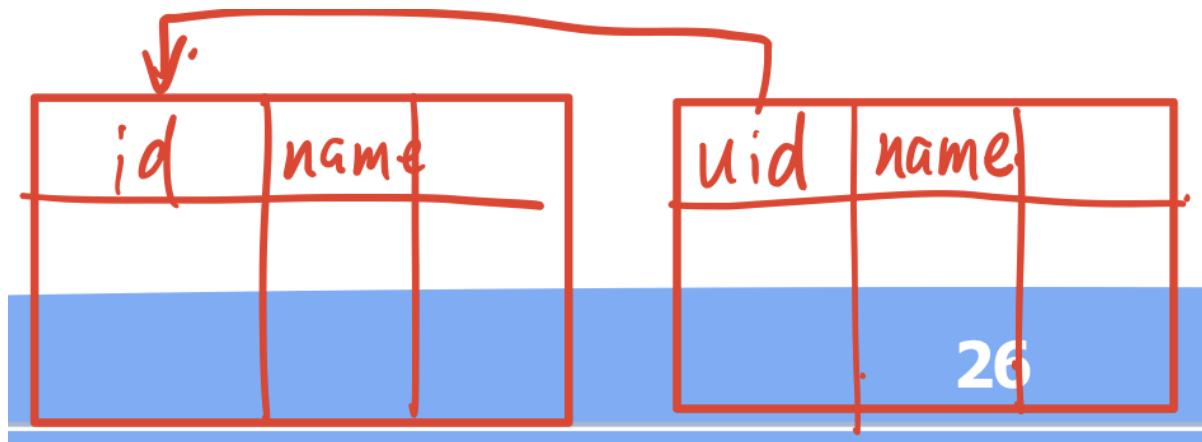
## Indexes

- The primary index of a table should be as short as possible.
  - This makes identification of each row easy and efficient.
  - For InnoDB tables, the primary key columns are duplicated in each secondary index entry, so a short primary key saves considerable space if you have many secondary indexes.
- Create only the indexes that you need to improve query performance.
  - Indexes are good for retrieval, but slow down insert and update operations.
  - If you access a table mostly by searching on a combination of columns, create a single composite index on them rather than a separate index for each column. The first part of the index should be the column most used.
  - If you always use many columns when selecting from the table, the first column in the index should be the one with the most duplicates, to obtain better compression of the index.

## • Joins

- In some circumstances, it can be beneficial to split into two a table that is scanned very often.
  - This is especially true if it is a dynamic-format table
  - and it is possible to use a smaller static format table that can be used to find the relevant rows when scanning the table.
- Declare columns with identical information in different tables with identical data types,
  - to speed up joins based on the corresponding columns.
- Keep column names simple, so that you can use the same name across different tables and simplify join queries.
  - For example, in a table named customer, use a column name of name instead of customer\_name.
  - To make your names portable to other SQL servers, consider keeping them shorter than 18 characters.

外键UID和ID必须要一样，这样才不会做强制类型转化又有额外开销。如果两张表里面有一样的东西也需要是类型一样的，名字也要一样，比如下面不能第二章表叫做username



范式化太高太慢了，要放松一些。

eg. 导出属性是不是需要存呢？比如order的总价可以通过item算出来的。缺陷是如果单价改了，就得改这个price，注意数据的一致性。范式化程度低，但是效率高，所以我们还是加上了，也就是放宽了范式化的要求。

### 数据类型的选择

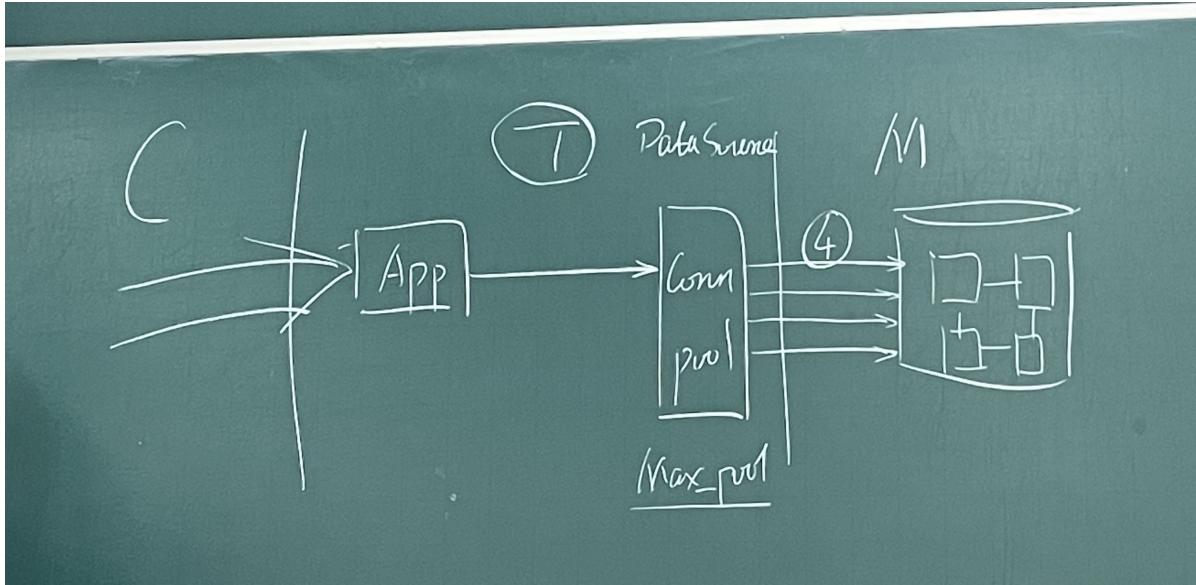
学号可以用string或者数字，所以能用数字存就别用字符串存

如果原本的内容很大的时候，就存成外面搞一个指针，小的存里面

## Optimizing MySQL Data Types

### Optimizing for Many Tables

如果表太多了，元数据很大



认为有4个并发的会话，为了让每个线程都有独占表的样子，就可能一张表打开了4次。打开：把表读到内存里。所以需要把空间开大，浪费空间但是性能得到很大的提高。

`table_open_cache` is related to `max_connections`.

`connection * 最多一个connection多少表`

For example, for 200 concurrent running connections, specify a table cache size of at least  $\$200 * N\$$ , where  $N\$$  is the maximum number of tables per join in any of the queries which you execute. You must also reserve some extra file descriptors for temporary tables and files.

## How MySQL Opens and Closes Tables

### 关掉表的情况

MySQL closes an unused table and removes it from the table cache under the following circumstances:

- When the cache is full and a thread tries to open a table that is not in the cache. 换进换出
- When the cache contains more than table open cache entries and a table in the cache is no longer being used by any threads. 表不用了关了
- When a table-flushing operation occurs. This happens when someone issues a FLUSH TABLES statement or executes a mysqladmin flush-tables or mysqladmin refresh command. 最近最少使用如果都不适合换出就临时扩大内存，当加入的表关闭了，就回收内存

- **How MySQL Opens and Closes Tables**

- When you execute a [mysqladmin status](#) command, you should see something like this:  
Uptime: 426 Running threads: 1 Questions: 11082  
Reloads: 1 Open tables: 12
- The Open tables value of 12 can be somewhat puzzling if you have **fewer** than 12 tables.
- MySQL is **multithreaded**, so there may be many clients issuing queries for a given table simultaneously.
  - To minimize the problem with multiple client sessions having different states on the same table, **the table is opened independently by each concurrent session**.
  - This uses **additional memory** but normally **increases performance**.
  - With **MyISAM** tables, one extra file descriptor is required for the data file for each client that has the table open.

- **How MySQL Opens and Closes Tables**

- The [table open cache](#) and [max connections](#) system variables affect the **maximum number** of files the server keeps open.
  - If you increase one or both of these values, you may run up against a limit imposed by your operating system on the per-process number of open file descriptors.
- [table open cache](#) is related to [max connections](#).
  - For example, for 200 concurrent running connections, specify a table cache size of at least  $200 * N$ , where  $N$  is the maximum number of tables per join in any of the queries which you execute. You must also reserve some extra file descriptors for temporary tables and files.
- Make sure that your operating system can handle the number of open file descriptors implied by the [table open cache](#) setting.
  - If [table open cache](#) is set **too high**, MySQL may run out of file descriptors and exhibit symptoms such as refusing connections or failing to perform queries.

- **How MySQL Opens and Closes Tables**

- MySQL **closes** an unused table and removes it from the table **cache** under the following circumstances:
  - When the cache is **full** and a thread tries to open a table that is **not in the cache**.
  - When the cache contains **more than table open cache** entries and a table in the cache is **no longer being used** by any threads.
  - When a table-flushing operation occurs. This happens when someone issues a [FLUSH TABLES](#) statement or executes a [mysqladmin flush-tables](#) or [mysqladmin refresh](#) command.

- **How MySQL Opens and Closes Tables**

- When the table cache **fills up**, the server uses the following procedure to locate a cache entry to use:
  - Tables **not currently** in use are released, beginning with the table **least recently used**.
  - If a new table **must be opened**, but the cache is full and no tables can be released, the cache is **temporarily extended** as necessary.
  - When the cache is in a **temporarily extended state** and a table goes from a used to **unused state**, the table is **closed** and released from the cache.

- **How MySQL Opens and Closes Tables**

- To determine whether your table cache is **too small**, check the [Opened\\_tables](#) status variable, which indicates the number of table-opening operations since the server started:

```
mysql> SHOW GLOBAL STATUS LIKE 'Opened_tables';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Opened_tables | 2741 |
+-----+-----+
```

- If the value is **very large or increases rapidly**, even when you have not issued many [FLUSH TABLES](#) statements, increase the [table open cache](#) value at server startup.

## Limits on Number of Databases and Tables



- MySQL has **no limit** on the number of databases.
  - The [underlying file system](#) may have a limit on the number of directories.
- MySQL has **no limit** on the number of tables.
  - The [underlying file system](#) may have a limit on the number of files that represent tables.
  - Individual storage engines may impose engine-specific constraints. [InnoDB](#) permits up to **4 billion** tables.

### Row Size Limits

VARCHAR 有动态存储，有额外的两个字节存长度，这样可以来压缩

NULL 额外加空间(额外的bit)来判断是否为空

CHAR 就没有动态，每个就存到这么长度补齐

TEXT 可以存到外面

### Row Size Limits Examples

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
   c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
   f VARCHAR(10000), g TEXT(6000)) ENGINE=MyISAM CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
   c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
   f VARCHAR(10000), g TEXT(6000)) ENGINE=InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)      Text 是存在外面? ? ? ? 怎么存呢我们要管吗?
```

# lect12: MySQL Optimization 2

字段<8K，设置为VCHAR， 否则必须要变成一个BLOG或者Text类型

切开常用和不常用的数据

主键用自动生成的方式好处是存的少，也可以用UUID，但是会比较长，同时不能大小比较

## Optimizing for innodb tables

1. 使用VCHAR而不是CHAR
2. 重复文本多的就用压缩模式eg. 学号
3. 落到硬盘是一整页操作，就多个写操作定义到一个事务里面，所以把autocommit 关掉用事务来。搞一个cache来存。但是如果cache里面放太多了一部分一部分落入硬盘，但是这样roll back就麻烦了。内存里面要放一个Log buffer，来记住每个操作/操作得到的结果。
4. 长时间运行的事务要加锁，不好，要拆开成小事务。读写的事务最好分开，做READ-ONLY事务，这样会优化很多（比如不会存事务ID。。）
5. 如果设置数据UNIQUE， 这样每次操作都要检查是否唯一，如果要开，关掉autocommit，这样就不在硬盘上做检查，在内存检查
6. 关掉外键检查（是否存在）
7. 一句sql里面插入多行
8. 主键整数递增设置为2， 不是1，可能是多线程，这样可以使得冲突的概率减小
9. query：复合索引不要用太多的列，索引里面包含NULL的值越多越不好。→ 字段尽量设置为NOT NULL。NULL多占空间
10. CPU，磁盘，内存可能成为瓶颈。检查CPU占用率
11. DMA 直接硬盘访问
12. Optimizing InnoDB Disk I/O 换磁盘用SSD，用非旋转的。随机读取HDD<< SSD
13. 表格定期optimize，进行数据整合。整理破碎的数据，这样可以把随机读取变成顺序读取。
14. 表格内容删除，表不删除，用TRUNCATE，不是一行一行删除。主键不要轻易改动主键，改动的代价比较大。
15. 磁盘flush可以设置内存里面的脏页达到了多少需要flush。如果设置的比较大万一内存崩溃了，需要恢复。

- Optimizing Storage Layout for InnoDB Tables
  - Once your data reaches a **stable size**, or a growing table has increased by tens or some hundreds of megabytes, consider using the **OPTIMIZE TABLE** statement to reorganize the table and compact any wasted space.
    - **OPTIMIZE TABLE** copies the data part of the table and rebuilds the indexes. The benefits come from improved packing of data within indexes, and reduced fragmentation within the tablespaces and on disk.
  - In InnoDB, having a **long PRIMARY KEY** (either a single column with a lengthy value, or several columns that form a long composite value) wastes a lot of disk space.
    - Create an **AUTO\_INCREMENT** column as the primary key if your primary key is long, or index a prefix of a long VARCHAR column instead of the entire column.
  - Use the **VARCHAR** data type instead of **CHAR** to store variable-length strings or for columns with **many NULL values**.
  - For tables that are **big**, or contain **lots of repetitive text or numeric data**, consider using **COMPRESSED** row format.
- Optimizing InnoDB Transaction Management
  - The default MySQL setting **AUTOCOMMIT=1** can impose performance limitations on a busy database server.
    - Where practical, **wrap several related data change operations into a single transaction**, by issuing **SET AUTOCOMMIT=0** or a **START TRANSACTION** statement, followed by a **COMMIT** statement after making all the changes.
  - Alternatively, for transactions that consist only of a single **SELECT** statement, turning on **AUTOCOMMIT** helps InnoDB to recognize read-only transactions and optimize them.
  - **Avoid performing rollbacks** after **inserting, updating, or deleting huge numbers of rows**.
    - If a big transaction is slowing down server performance, rolling it back can make the problem worse, potentially taking several times as long to perform as the original data change operations.
    - Killing the database process does not help, because the rollback starts again on server startup.
- Optimizing InnoDB Transaction Management
  - To minimize the chance of this issue occurring:
    - Increase the size of the **buffer pool** so that all the data change changes can be cached rather than immediately written to disk.
    - Set **innodb\_change\_buffering=all** so that update and delete operations are buffered in addition to inserts.
    - Consider issuing **COMMIT** statements periodically during the big data change operation, possibly breaking a single delete or update into multiple statements that operate on smaller numbers of rows.
  - **A long-running transaction** can prevent InnoDB from purging data that was changed by a different transaction.
    - When rows are modified or deleted within a long-running transaction, other transactions using the **READ COMMITTED** and **REPEATABLE READ** isolation levels have to do more work to reconstruct the older data if they read those same rows.
    - When a long-running transaction modifies a table, queries against that table from other transactions do not make use of the **covering index** technique.

## Optimizing InnoDB Read-Only Transactions

- InnoDB can avoid the overhead associated with setting up the [transaction ID](#) (TRX\_ID field) for transactions that are known to be read-only.
- InnoDB detects read-only transactions when:
  - The transaction is started with the [START TRANSACTION READ ONLY](#) statement.
  - The [autocommit](#) setting is turned on, so that the transaction is guaranteed to be a single statement, and the single statement making up the transaction is a “non-locking” [SELECT](#) statement.
  - The transaction is started without the **READ ONLY** option, but **no updates or statements that explicitly lock rows** have been executed yet.
- Thus, for a read-intensive application such as a report generator, you can tune a sequence of InnoDB queries
  - by grouping them inside [START TRANSACTION READ ONLY](#) and [COMMIT](#), or
  - by turning on the [autocommit](#) setting before running the [SELECT](#) statements, or
  - simply by avoiding any data change statements interspersed with the queries.

16. InnoDB 配置变量：

### Optimizing InnoDB Configuration Variables

- Controlling the types of data change operations for which InnoDB buffers the changed data, to **avoid frequent small disk writes**.
- Turning the **adaptive hash indexing feature** on and off using the [innodb adaptive hash index](#) option.
- Setting a limit on **the number of concurrent threads** that InnoDB processes, if context switching is a bottleneck.
- Controlling **the amount of prefetching** that InnoDB does with its read-ahead operations.
- Increasing **the number of background threads** for read or write operations, if you have a high-end I/O subsystem that is not fully utilized by the default values
- Controlling **how much I/O** InnoDB performs in the background.
- Controlling the algorithm that determines when InnoDB performs **certain types of background writes**.

索引选择：hash or B+ tree 如果搜索用 = hash 范围搜索 B+ tree

prefetching : cache 读取附近数据到内存

根据不同的数据访问模式，使用存储介质组合

## Buffering and Caching

### Configuring Multiple Buffer Pool Instances

对于有很大的buffer pool (几G, 至少1G), 将buffer pool 分成多个实例，可以提高并发性。(缓存越大越好，这样就可以写内存不用写硬盘了，但是大了就要分实例)

如果硬盘满了， LRU algorithm, 最新最少。

对于每一个新数据，不要放到最上面，因为可能虽然一开始最热，但是不是经常访问，需要很久才能出去。因为最新的数据不是最热的数据。所以放到一个比较冷的地方比如3/8(靠近远端) 的位置，这样就比较好。所以3/8 之前在比较冷的数据，之后是是比较热的数据，刚进来就在冷热的边界。

## Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)

预测哪些数据需要预先读进来。

## Configuring Buffer Pool Flushing

有一个位图 (bit map) 来描述哪个page是dirty page, 下次换出的时候需要触发flush。以及比如看有多少dirty的页了，dirty page 超过了10%，就做一次同步，把这些脏页落到硬盘上。

缓存需要做备份，放到硬盘上，为了防止缓存断电了，这样恢复warm-up比较快。

有innodb max dirty pages参数

## Caching of Prepared Statements and Stored Programs

## MyISAM

myisam 轻量级，适合比较小，适合事务级别的操作。？？？？

线程共享：可以一个写，多个读

multiple key caches

hot 20% 因为要建  来实现，如果太大的话效率低

warm 60% 内存中，stack 向下，heap 向上，这两个数据的组织方式不一样。

cold 20%

- For a busy server, you can use a strategy that involves three key caches:
- A "hot" key cache that takes up 20% of the space allocated for all key caches. Use this for tables that are heavily used for searches but that are not updated.
- A "cold" key cache that takes up 20% of the space allocated for all key caches. Use this cache for medium-sized, intensively modified tables, such as temporary tables.
- A "warm" key cache that takes up 60% of the key cache space. Employ this as the default key cache, to be used by default for all other tables.

在叶子节点占据内存很大的时候，加载索引的时候，不加载叶子节点，只加载分支。

Index preloading: 经常用到的索引在开始的时候就全部加载进来，访问的时候就很快找到，在需要找的时候再加载叶子节点。

- **Index Preloading**
  - If there are enough blocks in a key cache to hold blocks of an **entire index**,
    - or at least the blocks corresponding to its **nonleaf nodes**, it makes sense to preload the key cache with index blocks before starting to use it.
    - Preloading enables you to put the table index blocks into a key cache buffer in the most efficient way: **by reading the index blocks from disk sequentially**.
  - To preload an index into a cache, use the [LOAD INDEX INTO CACHE](#) statement.
  - For example, the following statement preloads nodes (index blocks) of indexes of the tables t1 and t2:

```
mysql> LOAD INDEX INTO CACHE t1, t2 IGNORE LEAVES;
+-----+-----+-----+
| Table | Op      | Msg_type | Msg_text |
+-----+-----+-----+
| test.t1 | preload_keys | status   | OK      |
| test.t2 | preload_keys | status   | OK      |
+-----+-----+-----+
```

如果需要把cache 尺寸变大，需要重建cache, 所以需要重新写缓存，所以这个尺寸不要轻易修改。

### **prepared statement**

给个模板填写内容，编译生成，缓存下生产的sql语句。

### **Stored programs (stored procedures and functions, triggers, and events)**

#### **trigger**

比如有一张表，要求某个参数 $\geq 0$ ，在insert的时候就触发trigger检查函数。

或者当某个值处于某个范围的时候，同一行的另外一个值也有变化。

#### **store procedure**

业务逻辑直接放到数据库里面做，这样可以减少网络传输，虽然不具备了移植性，比如mysql 改到了oracle 就不行了。那么我们需要把这个存储过程也缓存下来。

数据库存在硬盘上，数据库内容要放到内存，Client都是对内存里面的数据操作，所以所有参数，设计都是针对这个流程在进行。调整各个参数。

## **lect 13: MySQL Backup & Recovery**

全量备份：数据大，麻烦

对于文件是使用copy on write

增量备份：每次增加了什么数据，做了什么操作。

备份完了怎么回复。备份的最好周期性的备份。备份最好压缩，加密。

## **物理备份和逻辑备份**

物理备份：

1. 拷贝表。如果数据库很大，会脚本小。但是如果是mysql导出到oracle肯定不行 或者mysql升级，如果兼容性不好就拷贝不过去。需要同一个版本的mysql, 硬件要相同（mac 和 windows 就可能不行）
2. 导入的时候要求mysql停机，因为数据要覆盖。冷备份，适合大数据库，直接覆盖比较快，简单。
3. 不能做更细的划分，最小粒度是文件，不能文件里面满足一些条件的东西拿出来。
4. 使用mysqlbackup (外设，不是mysql自带)
5. 生成的文件是无法看懂的，安全性好。

逻辑备份：

1. 生成一个mysql脚本(先创建表格，再一条一条插入)，恢复直接执行一遍就好了。
2. 转化成脚本时间长，适合比较小的。
3. 这里不需要停机，因为是执行脚本，并且也不会影响到其他的用户。在线，热备份，适合小数据库。想要恢复的数据库需要处于锁定状态，其他用户没法读。
4. 脚本不能备份log, 配置文件这些东西。
5. 数据都是纯文本格式，所以没有加密了。
6. 因为数据是一行一行的，所以数据的粒度更细。
7. 数据是可移植的，因为是用了sql语言，即使移植到oracle里面也可以的。
8. 导出来的格式可以是sql脚本，也可以是自己制定的，比如CSV，这样就可以用load data statement和mysqlimport 。在这之前需要把数据库的框架做好。这个和物理备份的区别是，物理备份导出的就是.MYD , .MYI 文件，不关心内容是什么，而这个CSV甚至可以是excel写出来的

- Physical backup methods have these characteristics:
  - The backup consists of **exact copies of database directories and files**. Typically this is a copy of all or part of the MySQL data directory.
  - Physical backup methods are **faster** than logical because they involve **only file copying without conversion**.
  - Output is **more compact** than for logical backup.
  - Because backup speed and compactness are important for **busy, important databases**, the MySQL Enterprise Backup product performs **physical backups**.
  - Backup and restore **granularity** ranges from the level of the entire data directory down to the level of individual files. This may or may not provide for table-level granularity, depending on storage engine.
    - For example, InnoDB tables can each be in a **separate file**, or share file storage with other InnoDB tables; each MyISAM table corresponds uniquely to a **set of files**.
  - In addition to databases, the backup can include any related files such as **log or configuration files**.

### Physical backup methods have these characteristics:

- Data from **MEMORY** tables is tricky to back up this way
  - because their contents are **not** stored on disk.
- Backups are **portable** only to other machines that have **identical or similar hardware characteristics**.
- Backups can be performed while the MySQL server is **not running**.
  - If the server is running, it is necessary to perform appropriate **locking** so that the server does not change database contents during the backup.
  - MySQL Enterprise Backup does this locking **automatically** for tables that require it.
- Physical backup tools include the **mysqlbackup** of MySQL Enterprise Backup for **InnoDB** or any other tables, or file system-level commands (such as **cp, scp, tar, rsync**) for **MyISAM** tables.
- For restore:
  - MySQL Enterprise Backup restores **InnoDB** and other tables that it backed up.
  - **ndb restore** restores **NDB** tables.
  - Files copied at the file system level can be copied back to their **original locations** with file system commands.

Logical backup methods have these characteristics:

- The backup is done by **querying** the MySQL server to **obtain database structure and content information**.
- Backup is **slower** than physical methods because the server must **access** database information and **convert** it to logical format.
- Output is **larger** than for physical backup, particularly when saved **in text format**.
- Backup and restore **granularity** is available at the server level (all databases), database level (all tables in a particular database), or table level.
- The backup does **not** include log or configuration files, or other database-related files that are **not** part of databases.
- Backups stored in logical format are **machine independent and highly portable**.
- Logical backups are performed with the MySQL server **running**.
- Logical backup tools include the **mysqldump** program and the **SELECT ... INTO OUTFILE** statement. These work for any storage engine, even **MEMORY**.
- To restore logical backups, **SQL-format dump files** can be processed using the **mysql** client. To load **delimited-text files**, use the **LOAD DATA** statement or the **mysqlimport** client.

## online vs offline backup

### Online Versus Offline Backups

- **Online** backups take place while the MySQL server is **running** so that the database information can be obtained from the server.
- **Offline** backups take place while the server is **stopped**.
- This distinction can also be described as “**hot**” versus “**cold**” backups;
  - a “**warm**” backup is one where the server remains **running but locked against modifying data** while you access database files externally.
- **Online** backup methods have these characteristics:
  - The backup is **less intrusive to other clients**, which can connect to the MySQL server during the backup and may be able to access data depending on what operations they need to perform.
  - Care must be taken to **impose appropriate locking** so that data modifications do not take place that would compromise backup integrity. The MySQL Enterprise Backup product **does such locking automatically**.

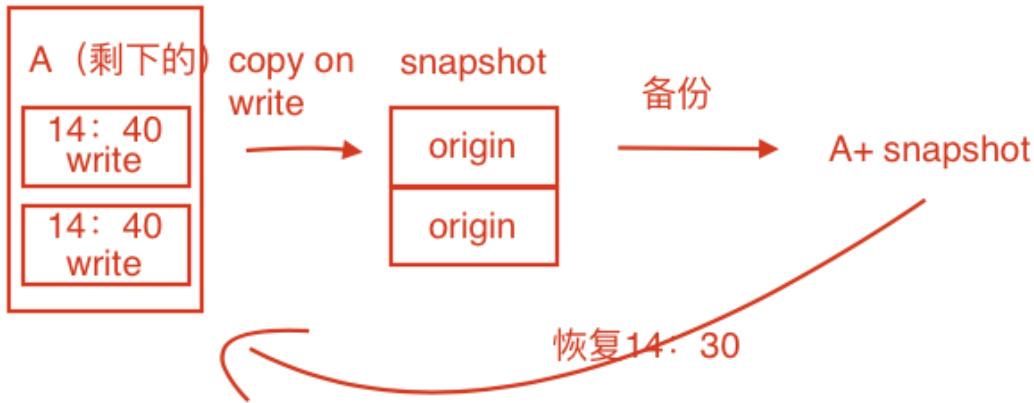
### Online Versus Offline Backups

- **Offline** backup methods have these characteristics:
  - Clients can be **affected adversely** because the server is **unavailable during backup**. For that reason, such backups are often taken from a **replica** that can be taken offline without harming availability.
  - The backup procedure is **simpler** because there is no possibility of interference from client activity.
- A similar distinction between **online** and **offline** applies for recovery operations, and similar characteristics apply.
  - However, it is more likely for clients to be affected by online recovery than by online backup because **recovery requires stronger locking**.
  - During backup, clients might be able to read data while it is being backed up. Recovery modifies data and does not just read it, so clients must be **prevented from accessing data while it is being restored**.

## local VS remote backup

备份可以本机也可以远程做，

## snapshot



没有改的部分+ 改的部分的原始值= 没改之前的所有数据。

## binary log

binary log记住对所有数据库的操作。

只有全量备份（别的机器上的）才能防止硬盘坏了

比如19：00 一台机器硬盘和内存坏了，就在18：00 别的机器上的全量备份拿到，再用另外机器上的bin-log做的备份来恢复。全量备份和增强备份的组合。

在每次做全量备份（比如每隔6h）的时候就把之前的bin-log删除了。（类似commit）因为bin-log很大。

bin-log 之类的备份需要加密，压缩

```
mysql> SHOW BINARY LOGS;  
shell> mysqlbinlog --start-position=155 --stop-position=232  
/var/lib/mysql/bin.123456 \| mysql -u root -p
```

这个即使跳过了一条做错了的指令，后面的也不一定正确，因为后面的指令可能都是基于这个错误的指令算出来的错误的值做的。后面的如果是UPDATE，那这个update (4000) 的值就是基于错误的算出来的，这里已经不是做计算了，所以也可能失效。

## mysql enterprise backup

- Making a Hot Backup with MySQL Enterprise Backup
  - Customers of MySQL Enterprise Edition can use the [MySQL Enterprise Backup](#) product to do **physical** backups of **entire** instances or **selected** databases, tables, or both.
  - This product includes features for **incremental** and **compressed** backups.
  - Backing up the **physical** database files makes restore much **faster** than **logical** techniques such as the **mysqldump** command.
  - InnoDB tables are copied using a **hot backup** mechanism.
    - (Ideally, the InnoDB tables should represent a substantial majority of the data.)
  - Tables from other storage engines are copied using a **warm backup** mechanism.

## 物理备份

- Making Backups by Copying Table Files
  - MyISAM tables can be backed up by **copying table files** (\*.MYD, \*.MYI files, and associated \*.sdi files). To get a consistent backup, stop the server or lock and flush the relevant tables:  
`FLUSH TABLES tbl_list WITH READ LOCK;`
  - You need only a **read lock**; this enables other clients to continue to query the tables while you are making a copy of the files in the database directory.

拷贝这些文件之前需要是最新的，所以需要flush一下。

FLUSH之后需要加一个锁让别人不能读

## 逻辑备份

- Making Delimited-Text File Backups csv
  - To create a **text file** containing a table's data, you can use `SELECT * INTO OUTFILE 'file name'`  
`FROM tbl name.` 写到外部的xx文件，列之间使用tab隔开
  - The file is created on the MySQL server host, **not** the client host. For this statement, the output file **cannot** already exist because permitting files to be overwritten constitutes a security risk.
  - This method works for any kind of data file, but saves **only table data, not the table structure**.
  - Another way to create text data files (along with files containing [CREATE TABLE](#) statements for the backed up tables) is to use **mysqldump** with the **-tab** option.
  - To reload a delimited-text data file, use [LOAD DATA](#) or [mysqlimport](#).

## Making Incremental Backups by Enabling the Binary Log

- MySQL supports **incremental** backups using the **binary log**.
- The binary log files provide you with the information you need to **replicate changes to the database** that are made subsequent to the point at which you performed a backup.
- At the moment you want to make an **incremental** backup (containing all changes that happened since the last full or incremental backup), you should rotate the binary log by using [FLUSH LOGS](#).
- The next time you do a full backup, you should also rotate the binary log using [FLUSH LOGS](#) or [mysqldump --flush-logs](#).

导出bin-log

## Making Backups Using Replicas

备份的时候使用slave 不用master，这样性能就可以提高

## 崩溃的情况

- Operating system crash
- Power failure
- File system crash
- Hardware problem (hard drive, motherboard, and so forth)

如果hardware problem

格式化硬盘吧，装个新的。这个备份就自己用除了MySQL以外的工具了。每隔一段时间，生成脚本的全量备份。每次dump的时候就要加锁，所以不能太频繁。导出所有数据库文件的时候需要把bin-log文件删除。

如果崩溃了，先恢复到之前的全量备份（2天前）+前两天的bin-log文件（不会增长了）+当天的还在增长（bin-log）的文件。注意全量备份，bin-log都不可以和数据文件放到一起，否则就会一起挂掉。**这就是一个全量备份和增量备份的组合。**

## Using mysqldump for Backups

dump 的两种方式：

1. 导出纯sql 脚本，全都是insert语句。

尺寸大，不直观，但是平台迁移性比较好。

2. 导出一个建库的脚本和数据格式的txt。

数据紧凑，但是需要专门的import 工具导入，像matlab的import。可以被excel 之类的打开，很直观。

数据库里面如果写了一些小程序，比如trigger,存储过程... 这样mysql 和tomcat 的传输少了，但是这样做数据库迁移的时候这些都得写。但是不同的数据库对sql有方言dialect, hibernate 当时就有配置方言。

JPA的作用就是把sql的结构化的编程变成了java的面向对象的编程。

但是如果有大量的数据处理和传输，可能就是放在mysql里面处理会更好了。如果是呈现式的大量数据传输免不了，还是放在后端java写比较好。（数据库迁移和更新比较好）

1. 请你详细描述如何通过全量备份和增量备份来实现系统状态恢复。(2分)
2. 请你根据MySQL缓存的工作原理，描述预取机制的优点。(1分)
3. 请你按照你的理解，阐述Partition机制有什么好处？如果数据文件在一台机器上有足够的存储空间存储，是否还需要进行Partition ?(2分)

## Q1

### 概念介绍

全量备份 full backup : MySQL server在给定时间点的**所有**data. 于是通过它可以实现给定时间点的恢复。

实现：MySQL对全量备份有多种方式。

优势：备份的数据最全面且最完整，当发生数据丢失灾难时只要用一盘磁带（即灾难发生前一天的备份磁带）就可以恢复全部的数据，恢复数据的速度快。

缺点：需要很大的存储空间，带宽，完成备份过程相对耗时。

增量备份 incremental backup : 在一个时间段（两个时间点之间）增量式的备份所有的对数据的改变。

实现：通过server的binary log 来记录数据改变，也叫做point-in-time recovery 因为通过全量+增量可以实现任意时间点的恢复。

优势：备份速度快，没有重复的备份数据，节省了磁带空间，缩短了备份时间。

缺点：恢复需要把多个备份集合的数据拼凑在一起，如果一个备份集crash，完全恢复的可能性较小。

系统恢复通过结合 全量备份和增量备份来实现：

1. 比如现在11月3日12：05 数据库崩溃了需要恢复到之前的状态。

2. 查找所有的全量备份，看到了最新的11月1日 00：00做了全量备份，就先恢复到这个状态
3. 找到11月1日00：00-11月2日：00 和11月2日00：00-11月3日00：00 的已经写完成了bin-log文件（这两个文件不会修改了），按照顺序恢复。
4. 再找到现在正在写的（11月3日00：00-12：05）bin-log文件进行恢复。

这样通过全量备份和增量备份就可以恢复到任意的时间点了。然而需要注意的是，无论是全量还是增量备份，都需要存到非这个数据库的机器/硬盘上，否则可能一起crash，这样就没有意义了。

## Q2

好处：总结自讲课和板书

目的：通过预读操作控制InnoDB的预取量。

背景：磁盘数据被optimize之后，变成连续存放，内部碎片减少。当系统有未使用的I/O容量时，更多的预读可以提高查询的性能。比如磁盘转一圈就可以把整个表的数据或者使用的数据的周围数据都读出来。所以预读可以提高性能。但是过多的预读可能会导致高负载系统的性能周期性下降。

对于线性预读，基于数据locality，因为是在一个表里做操作，临近数据被取用或者整个表的数据被频繁使用是很可能的。

对于随机预读，上课提到的背景是：但是如果是交易性的操作，只要读一个订单，就应该使用随机处理（因为不需要周围其他的order）。有一张order表，关联了orderItem，orderItem又关联了book，需要一定的预测机制来得到对应的orderItem和book。在这里，预测算法就显得非常重要了，预测的越准，性能越好。

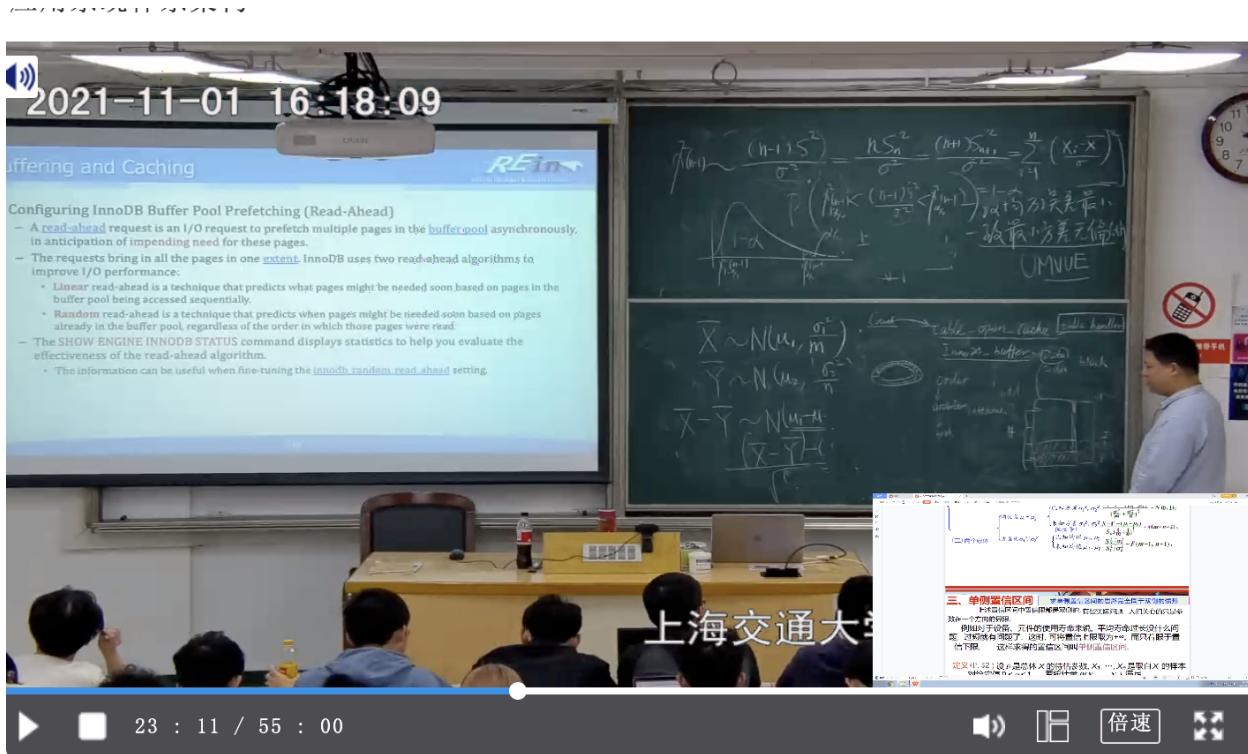
技术：总结自mysql官方文档和PPT

预读请求是指预取缓冲池中的多个页面的异步请求，预计这些页面很快就会被使用。请求在一个区段中引入所有页面。InnoDB使用两种预读算法来提高I/O性能：

**线性预读**是一种基于按顺序访问的缓冲池中的页面来预测可能很快需要哪些页面的技术。通过配置参数innodb\_read\_ahead\_threshold，可以通过调整触发异步读请求所需的顺序页访问次数来控制InnoDB执行预读操作的时间。在此之前，InnoDB只会在读取当前extent的最后一页时，计算是否对整个下一个extent发出异步预取请求。

**随机预读**是一种技术，它可以根据缓冲池中已经存在的页面预测何时可能需要页面，而不管这些页面的读取顺序如何。如果在缓冲池中发现同一个区段的13个连续页面，InnoDB会异步发出一个请求来预取该区段的剩余页面。

- Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)
  - A **read-ahead** request is an I/O request to prefetch multiple pages in the **buffer pool** **asynchronously**, in anticipation of **impending need** for these pages.
  - The requests bring in all the pages in one **extent**. InnoDB uses two read-ahead algorithms to improve I/O performance:
    - Linear** read-ahead is a technique that predicts what pages might be needed soon based on pages in the buffer pool being accessed sequentially.
    - Random** read-ahead is a technique that predicts when pages might be needed soon based on pages already in the buffer pool, regardless of the order in which those pages were read.
  - The **SHOW ENGINE INNODB STATUS** command displays statistics to help you evaluate the effectiveness of the read-ahead algorithm.
  - The information can be useful when fine-tuning the **innodb\_random\_read\_ahead** setting.



## Q3

### 分区的好处：

#### (1) 可伸缩性：

可能表太大了，单个文件或者硬盘放不下。文件一次性读进来占用很多时间和内存。

将数据分区分在不同磁盘，可以解决单磁盘容量瓶颈问题，存储更多的数据，也能解决单磁盘的IO瓶颈问题。

#### (2) 提升数据库的性能：

减少数据库检索时需要遍历的数据量，在查询时只需要在数据对应的分区进行查询。

避免Innodb的单个索引的互斥访问限制

对于聚合函数，例如sum()和count()，可以在每个分区进行并行处理，最终只需要统计所有分区得到的结果

(3) 方便对数据进行运维管理：

方便管理，对于失去保存意义的数据，通过删除对应的分区，达到快速删除的作用。比如删除某一时间的历史数据，直接执行truncate，或者直接drop整个分区，这比delete删除效率更高；在某些场景下，单个分区表的备份很恢复会更有效率。

分区一个最大的优点就是可以非常高效的进行历史数据的清理。也容易把整个分区比较容易增删改查，比如对所有同一类数据操作，如果是删除，这些数据在一个分区里，直接删除掉这个分区就可以了。查询也可以只查询这一个分区。甚至sql语句里面可以指定在哪个分区里面查。

如果在一台机器上有足够的存储空间存储，在表很大，经常需要对一类数据进行处理的时候，仍然需要分区。

上面提到的(2)(3) 两点仍然起作用，分区进行增删改查，使得顺序读写的概率增加，从而性能得到了提升

但是因为bookstore的数据量太少了，分区反而降低性能了。

## lect14 : MySQL Partitioning

### 背景：需要做分区&分区优势

Partitioning takes this notion a step further,

- by enabling you to **distribute portions of individual tables across a file system according to rules which you can set largely as needed.**
- In effect, different portions of a table are stored **as separate tables** in different locations.
- The user-selected rule by which the division of data is accomplished is known as a **partitioning function**, which in MySQL can be the modulus, simple matching against a set of ranges or value lists, an internal hashing function, or a linear hashing function.
- The function is selected according to the partitioning type specified by the user, and takes as its parameter the value of **a user-supplied expression**.
- This expression can be a column value, a function acting on one or more column values, or a set of one or more column values, depending on the type of partitioning that is used.

1. 可能表太大了，单个文件或者硬盘放不下。文件一次性读进来很麻烦

2. 容易把整个分区比较容易增删改查，比如对所有同一类数据操作，如果是删除，直接删除掉这个分区就可以了。查询也可以只查询这一个分区。甚至sql语句里面可以指定在哪个分区里面查。
3. 不同的区域可以在不同的存储设备上存储，这样就可以存下更多的数据了。

Some advantages of partitioning are listed here:

- Partitioning makes it possible to **store more data in one table than can be held on a single disk or file system partition**.
- Data that loses its usefulness can often be **easily removed from a partitioned table** by dropping the partition (or partitions) containing only that data. Conversely, the process of **adding new data** can in some cases be greatly facilitated by adding one or more new partitions for storing specifically that data.
- **Some queries can be greatly optimized** in virtue of the fact that data satisfying a given WHERE clause can be stored only on one or more partitions, which automatically excludes any remaining partitions from the search.
- In addition, MySQL supports **explicit partition selection for queries**. For example, `SELECT * FROM t PARTITION (p0,p1) WHERE c < 5` selects only those rows in partitions p0 and p1 that match the WHERE condition.
  - In this case, MySQL does not check any other partitions of table t; this can greatly speed up queries when you already know which partition or partitions you wish to examine.
  - Partition selection is also supported for the data modification statements `DELETE`, `INSERT`, `REPLACE`, `UPDATE`, and `LOAD DATA`, `LOAD XML`.

## 分区策略

The types of partitioning which are available in MySQL 8.0 are listed here:

- **RANGE partitioning**. This type of partitioning assigns rows to partitions based on **column values falling within a given range**.
- **LIST partitioning**. Similar to partitioning by RANGE, except that the partition is selected based on columns **matching one of a set of discrete values**.
- **HASH partitioning**. With this type of partitioning, a partition is selected based on **the value returned by a user-defined expression** that operates on column values in rows to be inserted into the table. The function may consist of any expression valid in MySQL that yields a **nonnegative integer value**.
- **KEY partitioning**. This type of partitioning is similar to partitioning by HASH, except that **only one or more columns to be evaluated are supplied**, and the **MySQL server provides its own hashing function**. These columns can contain other than integer values, since the hashing function supplied by MySQL guarantees an integer result regardless of the column data type.

1. 按范围分区，这种会根据列的值是否落在一个给定的范围内进行分区，好处是范围可以根据需要调节。
2. 按照List分区，分区基于列是否等于一些离散的值来进行分区。和range的区别是按照成员关系的归属来区分，比如某个地区的storeId num
- 2.5 column 和 range column partition (见ppt)
3. 按照哈希分区，分区是基于程序员定义的哈希表达式来进行划分的。哈希函数输入是每行的列的

值，并且这个哈希函数必须对任何有效的 Mysql 表达式都有效。好处是比较平均。使用线性hash可以比常规hash使得分布更均匀。

4. 按照Key来分区，和哈希分区类似，只是可以通过一列或多列进行算，并且MySQL提供了其哈希函数。这些列可以包括任意类型的值，因为这个 MySQL 函数可以保证输出一个整数的结果。也就是hash交给mysql做。

(这部分见BC)

对某一列的求值范围进行划分（离散值：本科生，研究生、博士生，列举出所有的值），（连续值：GPA=4； $\geq 3.75$  ; $\geq 3$ ）（hash一下）重要的是分区的逻辑是什么。

不同区域不相交，合并为全集。

## 按照多列分区

多列比较， $(0,25,50) < (10,20,100)$  是从左到右依次比较。如果=向下看，如果有大小区分就比出来了。

```
-> PARTITION p0 VALUES LESS THAN (5,10,'ggg'),  
-> PARTITION p1 VALUES LESS THAN (10,20,'mmm'),  
-> PARTITION p2 VALUES LESS THAN (15,30,'sss'),
```

但是这样容易引起歧义  $(6, 9, 'ttt')$

对于离散值，先规定每个值在哪个列，然后再分类。

但是如果漏掉了一些值，就会报错

## Subpartitioning

分区里面还可以有子分区

## How MySQL Partitioning Handles NULL

NULL对应的是最小值， $-X_{MAX}$ ,最小的负数

那些空值不好找，可以专门找一个很小的值-10000来做范围，来映射到NULL

```

the lowest partition.
mysql> CREATE TABLE t1 (
    -> c1 INT,
    -> c2 VARCHAR(20)
    -> )
    -> PARTITION BY RANGE(c1) (
    -> PARTITION p0 VALUES LESS THAN (0),
    -> PARTITION p1 VALUES LESS THAN (10),
    -> PARTITION p2 VALUES LESS THAN MAXVALUE
    -> );
Query OK, 0 rows affected (0.09 sec)

mysql> CREATE TABLE t2 (
    -> c1 INT,
    -> c2 VARCHAR(20)
    -> )
    -> PARTITION BY RANGE(c1) ( 其他正常的值都>-5, 只有NULL放在这个分区
    -> PARTITION p0 VALUES LESS THAN (-5), 一定要升序写下
    -> PARTITION p1 VALUES LESS THAN (0),
    -> PARTITION p2 VALUES LESS THAN (10),
    -> PARTITION p3 VALUES LESS THAN MAXVALUE
    -> );
Query OK, 0 rows affected (0.09 sec)

```

## 改变partition

range,只写小于等于，这样可以保证删除了之后整个覆盖的范围是完整的。而不是有个洞了。

```

PARTITION p0 VALUES LESS THAN (1980),
PARTITION p1 VALUES LESS THAN (1990),
PARTITION p2 VALUES LESS THAN (2000)

```

如果是增加分区， 只能在后面追加。如果在中间添加，只能重新组织这个分区 (reorganize)

```

PARTITION BY RANGE( YEAR(dob) ) (
    PARTITION p0 VALUES LESS THAN (1980),
    PARTITION p1 VALUES LESS THAN (1990),
    PARTITION p2 VALUES LESS THAN (2000)
);

```

```
ALTER TABLE members ADD PARTITION (PARTITION p3 VALUES LESS THAN (2010));
```

```
mysql> ALTER TABLE members
    > ADD PARTITION (
    > PARTITION n VALUES LESS THAN (1970));
ERROR 1463 (HY000): VALUES LESS THAN value must be strictly » increasing for each partition
```

```
ALTER TABLE members
REORGANIZE PARTITION p0 INTO (
    PARTITION n0 VALUES LESS THAN (1970),
    PARTITION n1 VALUES LESS THAN (1980)
);
                                         reorganize
```

## manage hash and key partition

对于哈希函数散列的删除追加特殊处理。就是所有的数据要重新散列

- However, you can **merge** HASH or KEY partitions using ALTER TABLE ... COALESCE PARTITION.  
CREATE TABLE clients (  
 id INT,  
 fname VARCHAR(30),  
 lname VARCHAR(30),  
 signed DATE  
)  
PARTITION BY HASH( MONTH(signed) )  
PARTITIONS 12;

```
mysql> ALTER TABLE clients COALESCE PARTITION 4;
Query OK, 0 rows affected (0.02 sec)
```

```
ALTER TABLE clients ADD PARTITION PARTITIONS 6;
```

## 交换分区 exchange partition and subpartition

详细见BC

```

mysql> CREATE TABLE e2 LIKE e;
Query OK, 0 rows affected (0.04 sec)
mysql> ALTER TABLE e2 REMOVE PARTITIONING;
Query OK, 0 rows affected (0.07 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> SELECT PARTITION_NAME, TABLE_ROWS
      FROM INFORMATION_SCHEMA.PARTITIONS
     WHERE TABLE_NAME = 'e';    交换区域
+-----+-----+
| PARTITION_NAME | TABLE_ROWS |
+-----+-----+
| p0 | 1 |
| p1 | 0 |
| p2 | 0 |
| p3 | 3 |
+-----+
2 rows in set (0.00 sec)

```

- **Nonmatching Rows**

```

mysql> INSERT INTO e2 VALUES (51, "Ellen", "McDonald");
Query OK, 1 row affected (0.08 sec)
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2; 交換过去的东西不满足target分区的规则
ERROR 1707 (HY000): Found row that does not match the partition

                    强制换，这样可能就有些条件下搜索失败了
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2 WITHOUT VALIDATION;
Query OK, 0 rows affected (0.02 sec)

```

请你根据上课内容，针对你在E-BookStore项目中的数据库设计，详细回答下列问题：

1. 你认为在你的数据库中应该建立什么样的索引？为什么？
2. 你的数据库中每个表中的字段类型和长度是如何确定的？为什么？
3. 你认为在我们大二上课时讲解ORM映射的Person例子时，每个用户的邮箱如果只有一个，是否还有必要像上课那样将邮箱专门存储在一张表中，然后通过外键关联？为什么？
4. 你认为为主键使用自增主键和UUID各自的优缺点是什么？
5. 请你搜索参考文献，总结InnoDB和MyISAM两种存储引擎的主要差异，并详细说明你的E-BookStore项目应该选择哪种存储引擎。

–请提交包含上述问题答案的文档，文档中附上你更新的数据库的设计方案，包括库结构、表结构和表与表之间的关联

评分标准：

–上述每个问题 1 分，答案不唯一，只要你的说理合理即可视为正确。

## Q1

下划线部分为我针对原则做的数据库改进。

1. 主键适合作为索引的场景及其原因：
  - a. 一般按照经常查找的列来建立索引。虽然主键（一般是ID）是用户不知道的信息，但是如果其他的表关联到这张表，用它的主键关联。用主键做索引就很需要了。因此如果有这样的情况就使用主键作为索引，比如OrderItem表关联Order 表的主键，因此Order表应该使用主键作为索引。
  - b. innodb 在硬盘中也是按照主键的顺序存储的，这样有利于顺序读写。
  - c. 如果不知道其他字段是否唯一，不如使用递增或者UUID生成的保证唯一性的主键。
  - d. 主键的设置方式一般有自增或者UUID。自增算法简单，并且可以得知数据的插入顺序。而UUID一般运用在分布式中。如果用服务器集群处理大量请求时，基于整数递增可能导致重复，因为插入的是两张不同机器上的表，但是UUID用机器IP + timestamp + 进程PID+当前对象ID+ 随机数，凑成一个32位的16进制数（16byte）几乎不会重复。
    - uuid 缺陷：uuid太大了，索引比较耗费空间。不可能知道插入顺序，如果需要知道顺序可以在前面加入前缀。
    - 平衡：是需要性能(自增)，还是需要分布式不能出错（UUID），看需求进行选择。
2. 如果有多个条件查找可以考虑建立复合索引。如果是复合索引：不可以有太多列，索引里面包含NULL的值越少越好。
3. 数据少的时候就没有必要建立索引。对于较小的表，建立索引的意义不大。但是如果是太大的表，如果每个元素都被使用，顺序读写的效率反而比索引高。
4. 索引大小不能太大，比如用book 的introduction做索引就不合适，可以选前面10个字符。因为这样会导致索引占用内存太多，会涉及大量页的换进换出导致性能降低。
5. 索引最好每个索引都定位到唯一的元素，这样使用主键做索引就比较好。
6. 拆开表

## Q2

你的数据库中每个表中的字段类型和长度是如何确定的？为什么？

1. 能用数字存储的尽量不用字符串存储，因为这样能存储的数据量更大。
2. 每个数据尽量都使用NOT NULL。因为NULL会比NOT NULL多使用一个字节表示该字段是否空。
3. 关于字符串的类型选择：  
一般在保存少量字符串的时候，我们会选择CHAR或者VARCHAR，

- char长度固定，即每条数据占用等长字节空间，适合用在身份证号码、手机号码等定。超过255字节的只能用varchar或者text。在我的实现中，书籍的ISBN是固定格式的，使用char格式。
- varchar可变长度，可以设置最大长度；适合用在长度可变的属性。字符串尽量使用VARCHAR，参数根据最大长度确定，因为VARCHAR可以动态存储，可以压缩空间。CHAR就没有动态存储，每个数据都会占用固定长度的空间。我的实现中，不确定长度的字符串使用varchar

保存较大文本时(>8KB)，通常会选择使用TEXT或者BLOB

- blob。blob 和text 二者之间的主要差别是BLOB能用来保存二进制数据，比如照片；而TEXT只能保存字符数据，比如一遍文章或日记
- text不设置长度，当不知道属性的最大长度时，适合用text，能用varchar的地方不用text。当text列的内容很多的时候，text列的内容会保留一个指针在记录中，这个指针指向了磁盘中的一块区域，当对这个表进行select \*的时候，会从磁盘中读取text的值，影响查询的性能，而varchar不会存在这个问题。在我的实现中，由于book表中的introduciton 是字符不确定长度，并且很长，因此选用了text类型。

## Q3

根据具体需求而定。需要考虑：使用频率、文本长度、一个用户的邮箱个数

1. 一般情况下，邮箱不是很长的文本，而且很多时候作为用户的登陆id，这个时候就使用频率较高，如果是一个用户单一邮箱，就直接存在用户信息的表里，不需要外键关联比较好。因为频繁使用Join操作会带来很大的花销。同时增加一个占用空间比较小的字段，在从磁盘读到内存的时候也不会带来太多占用内存的情况。
2. 特殊情况下，如果使用频率不大，或者极端情况下邮箱的格式特殊，都很长很长（虽然觉得这种情况很少见但是也要考虑），或者一个用户有好几个邮箱，这样如果存在另外一张表的join操作带来的代价比放入内存的性能代价要小的话，放在另外一张表更合适

## Q4

4. 主键的设置方式一般有自增或者UUID。

- 自增优势：
  - 算法简单，并且可以得知数据的插入顺序。

- 在进行数据库插入时，位置相对固定（B+树中的右下角）增加数据插入效率，减少插入的磁盘IO消耗，每页的空间在填满的情况下再去申请下一个空间，底层物理连续性更好，能更好的支持区间查找
- 自增主键 缺点：
  1. 安全性不高，容易被得知业务量和数据量等，而且容易被爬取数据。
  2. 高并发时，竞争自增锁会降低数据库的吞吐能力。
  3. 数据迁移时，尤其是发生表格合并时，多个表容易主键冲突。
- UUID优势：
  1. UUID一般运用在分布式中。如果用服务器集群处理大量请求时，基于整数递增可能导致重复，因为插入的是两张不同机器上的表，但是UUID用机器IP + timestamp + 进程PID+当前对象ID+随机数，凑成一个32位的16进制数（16byte）几乎不会重复。使用UUID，生成的ID不仅是表独立的，而且是库独立的。对以后的数据操作很有好处，可以说一劳永逸。
  2. 相对安全，不能简单的从uuid获取信息，但是如果自增，则容易暴露信息。
- uuid 缺陷：
  - uuid太大了，索引比较耗费空间。不可能知道插入顺序，如果需要知道顺序可以在前面加入前缀。
  - 由于UUID是随机生成的 插入时位置具有一定的不确定性，无序插入，会存在许多内存碎片，造成硬盘使用率低
  - 降低查询速度。
- 平衡：是需要性能(自增)，还是需要分布式不能出错（UUID），看需求进行选择。

## Q5

- 一、InnoDB支持事务，MyISAM不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而MyISAM就不可以了。
- 二、MyISAM适合查询以及插入为主的应用，InnoDB适合频繁修改以及涉及到安全性较高的应用
- 三、InnoDB支持外键，MyISAM不支持
- 四、MyISAM是默认引擎，InnoDB需要指定
- 五、InnoDB不支持FULLTEXT类型的索引
- 六、InnoDB中不保存表的行数，如select count(\*) from table时，InnoDB需要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count(\*)语句包含where条件时MyISAM也需要扫描整个表

七、对于自增长的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中可以和其他字段一起建立联合索引

八、清空整个表时，InnoDB是一行一行的删除，效率非常慢。MyISAM则会重建表

九、InnoDB支持行锁（某些情况下还是锁整表，如 `update table set a=1 where user like '%lee%'`

## Mysql中InnoDB和MyISAM的比较

### MyISAM：

每个MyISAM在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义。数据文件的扩展名为.MYD (MYData)。

MyISAM表格可以被压缩，而且它们支持全文搜索。不支持事务，而且也不支持外键。如果事物回滚将造成不完全回滚，不具有原子性。在进行update时进行表锁，并发量相对较小。如果执行大量的SELECT，MyISAM是更好的选择。

MyISAM的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而Innodb是索引和数据是紧密捆绑的，没有使用压缩从而会造成Innodb比MyISAM体积庞大不小

MyISAM缓存在内存的是索引，不是数据。而InnoDB缓存在内存的是数据，相对来说，服务器内存越大，InnoDB发挥的优势越大。

**优点：**查询数据相对较快，适合大量的select，可以全文索引。

**缺点：**不支持事务，不支持外键，并发量较小，不适合大量update

### InnoDB：

这种类型是事务安全的。.它与BDB类型具有相同的特性,它们还支持外键。InnoDB表格速度很快。具有比BDB还丰富的特性,因此如果需要一个事务安全的存储引擎，建议使用它。在update时表进行行锁，并发量相对较大。如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。

**优点：**支持事务，支持外键，并发量较大，适合大量update

**缺点：**查询数据相对较快，不适合大量的select

对于支持事物的InnoDB类型的表，影响速度的主要原因是AUTOCOMMIT默认设置是打开的，而且程序没有显式调用BEGIN 开始事务，导致每插入一条都自动Commit，严重影响了速度。可以在执行sql前调用begin，多条sql形成一个事物（即使autocommit打开也可以），将大大提高性能。

**基本的差别为：**MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持。

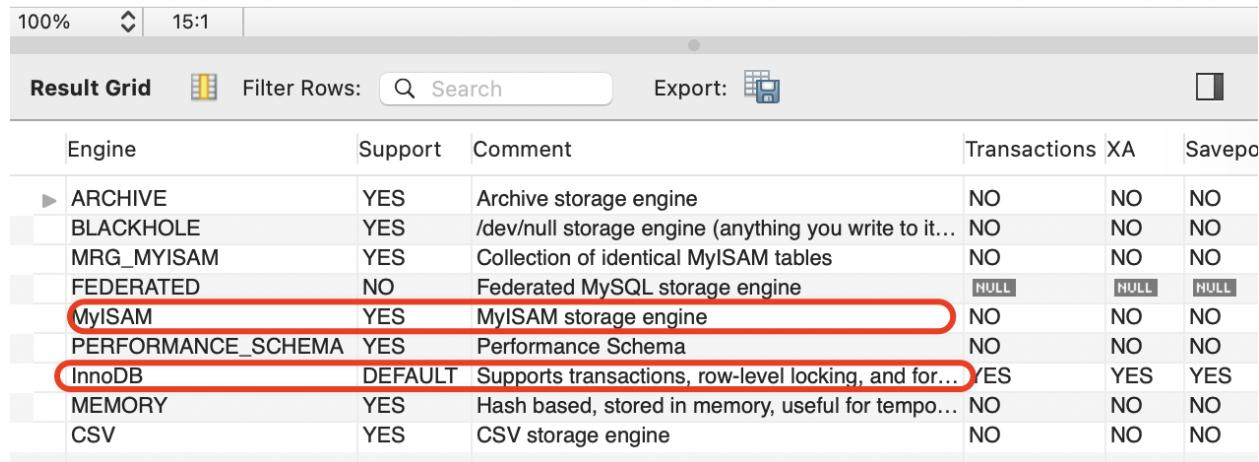
MyISAM类型的表强调的是性能，其执行速度比InnoDB类型更快，但是不提供事务支持，而InnoDB提供事务支持以及外部键等高级数据库功能。

### 其他比较：

MyISAM是ISAM表的新版本，有如下扩展：

- 二进制层次的可移植性。
- NULL列索引。
- 对变长行比ISAM表有更少的碎片。
- 支持大文件。
- 更好的索引压缩。
- 更好的键吗统计分布。
- 更好和更快的auto\_increment处理。

```
1 •  show engines;
2
3
```



Engine	Support	Comment	Transactions	XA	Savepo
► ARCHIVE	YES	Archive storage engine	NO	NO	NO
BLACKHOLE	YES	/dev/null storage engine (anything you write to it...)	NO	NO	NO
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
FEDERATED	NO	Federated MySQL storage engine	NUL	NUL	NUL
MyISAM	YES	MyISAM storage engine	NO	NO	NO
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO
InnoDB	DEFAULT	Supports transactions, row-level locking, and for...	YES	YES	YES
MEMORY	YES	Hash based, stored in memory, useful for tempo...	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO

在我的实现中，因为需要实现外键、事务。所以选择innodb

## 数据库更新方案

表包括：book, cart, orders, order\_items, user, user\_auth

### book

Column	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	G
book_id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>				
author	VARCHAR(50)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
description	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
image	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
inventory	INT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
isbn	CHAR(13)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
name	VARCHAR(50)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
price	INT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
type	VARCHAR(15)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				

- book\_id 是主键设成自增，建立唯一性索，便于其他表格外键引用，能够提升表格查询性能。因为目前还没有做分布式的系统，不容易有高并发或与其他数据库合并等操作，用自增的方式性能比较好。
- author 是作者名，因为名字可变长并且可能多个，故采用 varchar(50)。满足了绝大部分书籍作者名称长度要求且节约了存储空间。
- description 是书籍的简介。本数据库中，大部分的简介在100字左右。于是为了扩展起见，使用 varchar(255)，虽然是比较大的数据，但是上课提到基于性能考虑：“不超过 8 KB 的数据不用 BLOB/TEXT 用 VARCHAR ”
- image 是书籍的图片路径，使用了图床，平均的长度大概在160个字符，扩展起见以及同 introduction一样的考虑，使用VARCHAR
- inventory 是书籍的库存。由于小型书籍网站的书籍库存不会太大，因此使用 int 来存储，节约磁盘空间。
- isbn 是书籍的 isbn 编号，经过查阅资料，目前 isbn 的长度均为 13 位，因此用固定长度CHAR 的字符串存储。
- name 是书籍名称。因为名字可变长并且可能多个，故采用 varchar(50)。满足了绝大部分书籍作者名称长度要求且节约了存储空间。
- price 是书籍的价格，用 int 存储而不是 float 或 double 存储，避免了加减计算时的精度丢失问题，且满足了小型书籍网站的需求。

- *type* 是书籍种类对应的种类，考虑到种类之后可能有很多，并且可能有大分类下面的小分类，用数字存可能不能表达意思，于是采用字符串。
- 关于分表问题：虽然在首页不需要拿到*description* 和 *image* 这两个稍微大一些的数据。但是由于 *book* 是很多类里面的 *oneToMany* 的对象，比如 *orderItem*，这样就使得使用这些对象的时候要加载 *book*，再加载一张分开的表，有很多的 *join* 操作，极大的降低了性能。因为这两个数据并没有那么大，比起 *join* 产生的数据下降，不分表是合理的。

## lect15: NoSQL & MongoDB

- 能够根据数据特性，设计综合运用NoSQL数据库和关系型数据库的数据存储方案，以实现数据访问性能的优化
- 能够通过分层架构设计并实现跨类型数据存储机制下的数据访问

### 1. 数据有相互交互，但是要分在不同的机器上，怎么分呢？

处理完局部的结果不一定是最后的结果。→ map reduce (数据需要预处理)

一张表太大，数据的完整性使用锁来实现的，但是这样就性能低。就算乐观锁冲突也变多了。单张表的索引树会很大，

→ 水平切开，前  $1-n$  行放到一个机器，后面的  $n-2n$  另外一台，或者大一学生放到一台，大二学生放到一台。管理的复杂度比较高。划分的策略有 range, list, key, hash, composite 等

→ 垂直切开，对列进行划分。对有些不经常访问的数据放到另外一张表，然后外键关联。但 mysql 没有机制帮忙做这个事情。关系型数据库集群版价格高，但是性能提升很大。

→ 分区存储，但是仍然解决不了 三张表联动的问题。

### 2. DBMS 里面的三种数据的结构

*structured data*

所有的数据都遵循预定义的模式（表格数据）

*Semi-structured data*

没有字段和有这个字段为NULL是两个概念

有些数据没有一些字段有些有，松散的 schema。

*Unstructured data*

比如一张图片 base64，没有结构化的文本。因为单体的数据比较大，所以在大数据情况下会有很多这样的数据。does not have internal structure

所以说，semi 和 un 在 mysql 做不了，就用 nosql 了

### 3. 关系型数据库的缺陷：关系型数据库很难胜任非结构性的 schema

如果用关系型数据库，没有的属性用 NULL存

于是都不可设为非空

Screen	Standby	Q	P
30	80	NULL	Y
32	NULL	Y	NULL

关联

id	brand	name
1	Apple	iphone10
2	Sumsung	S10

id	pid	name	value
1	1	Screen	30
2	1	Standby	80
3	1	P	Y
4	2	Screen	32
5	2	Q	Y

化列为行。

缺陷：每个搜索

要join，且若一行

很长，则太大了

## Mongodb

介绍：

- Mongo DB is a document-oriented database, not a relational one , which makes scaling out easier
  - 针对文档，所以可以面对semi 和un的数据
- It can balance data and load across a cluster, redistributing documents automatically
  - KVstore sharding 机制
- It supports MapReduce and other aggregation tools, and supports generic secondary indexes
- MongoDB could do some administration by itself, if a master server goes down, MongoDB can automatically failover to a backup slave and promote the slave to the master

features:

- Document-OrientedStorage
- Full Index Support
- Replication&HighAvailability
- Auto-Sharding
- Querying
- FastIn-PlaceUpdates
- Map/Reduce

- GridFS
- Commercial Support

Basic concept:

组成：document → collection → database

- A **document** is the basic unit of data for MongoDB
- A **collection** can be thought of as the **schema-free** equivalent of a **table**, the documents in the same collection could have different shapes or types.
- Every document has a special key “**\_id**”, it is unique across the document’s collection
- Mongo DB **groups collections into database** and each database has its own permission and be stored in separate disks
- **Simple document**
  - A document is roughly equivalent to a **row** in a relational database, which contain one or multiple key-value pairs
  - `{"greeting" : "Hello, world!"}`
  - Most documents will be more complex than this simple one and often will contain multiple key/value pairs:
    - `{"greeting" : "Hello, world!", "foo" : 3}`
    - Key/value pairs in documents are **ordered**—the earlier document is distinct from the following document:
      - `{"foo" : 3, "greeting" : "Hello, world!"}`
  - Values in documents are **not just “blobs.”** They can be one of several different data types

- **Embedded document** 可以有嵌套

embedded documents are entire Mongo DB documents that are used as the values for a key in another document,

```
{
  "name" : "John Doe",
  "address" : {
    "street" : "123 Park Street",
    "city" : "Anytown",
    "state" : "NY"
  }
}
```



Collections are schema-free. 重点：键不一样但可以放到一个表里面

- This means that the documents within a single collection can have any number of different “shapes.”
- `{"greeting" : "Hello, world!"}`
- `{"foo" : 5}`

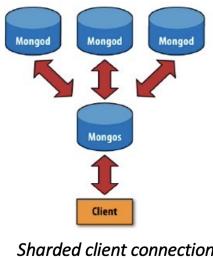
我们结构化的原因是数据应该：高内聚，低耦合，相关的放在一起，这样建立索引的话也让索引小一点。

### Why should we use more than one collection? 总之还是分而治之

- 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.

### autosharding in MongoDB

sharding is to break up collections into small chunks。使用shard key把内容分布到不同的机器



需要shard的场景

When the situations like this, you should probably to shard

- You've run out of disk space on your current machine.
- You want to write data faster than a single mongod can handle.
- You want to keep a larger proportion of data in memory to improve performance

## lect16: Neo4J & Graph Computing

能够根据数据特性和数据访问模式，识别适合图数据库存储的数据，设计并实现其在图数据库中的存储和访问方案

## Graph database

A *graph database management system* (henceforth, a *graph database*) is

- an online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model.
- Graph databases are generally built for use with transactional (OLTP) systems.
- Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

There are two properties of graph databases we should consider when investigating graph database technologies:

- *The underlying storage*
- *The processing engine*

四个特性：

### The Labeled Property Graph Model

A *labeled property graph* is made up of *nodes*, *relationships*, *properties*, and *labels*.

- Nodes contain properties. Think of nodes as documents that store properties in the form of arbitrary key-value pairs. In Neo4j, the keys are strings and the values are the Java string and primitive data types, plus arrays of these types.
- Nodes can be tagged with one or more labels. Labels group nodes together, and indicate the roles they play within the dataset.
- Relationships connect nodes and structure the graph. A relationship always has a direction, a single name, and a *start node* and an *end node*—there are no dangling relationships. Together, a relationship's direction and name add semantic clarity to the structuring of nodes.

#### 1. 什么场景适合使用图数据库？

- 高性能关系查询

需要快速遍历许多复杂关系的任何用例。这实际上包括欺诈检测，社交网络分析，网络和数据库基础设施等。

- 模型的灵活性

任何依赖于添加新数据而不会中断现有查询池的用例。模型灵活性包括链接元数据，版本控制数据和不断添加新关系。

- 快速和复杂的分析规则

当必须执行许多复杂的规则时，例如子图的比较。这包括推荐，相似度计算和主数据管理。

## 2. 与传统关系型数据库相比，图数据库的优势有：

1. 可以很自然的表达现实世界中的实体及其关联关系（对应图的顶点及边）；
2. 灵活的数据模型可以适应不断变化的业务需求；
3. 灵活的图查询语言，轻松实现复杂关系网络的分析；
4. 关系型数据库在遍历关系网络并抽取信息的能力非常弱，图数据库则为此而生；
5. 关系型数据库在数据规模庞大时很难做多层关联关系分析（Join操作往往消耗过长时间而失败），图数据库则天然把关联数据连接在一起，无需耗时耗内存的Join操作，可以保持常数级时间复杂度。

## 3. 图数据库典型查询示例：

- a. 多层关联：查询一个人的好友的好友有哪些？
- b. 最短路径：查询两个点之间的最短路径。
- c. 连通子图：查询一个点在K步以内相连接的所有邻接点（ $K=1,2,3\dots$ ）。
- d. 协同推荐/过滤：查询一个人的好友里面，哪些人喜欢哪些东西，然后把那些东西推荐给这个人。
- e. 集中度测量：如PageRank、PersonalRank、特征向量集中度、亲密度等。
- f. 节点可以表示抽象的事实（图见PPT）

# lect 17: Log-Structured Database

能够根据数据特性和数据访问模式，识别适合日志数据库存储的数据，设计并实现其在日志数据库中的存储和访问方案。

B树和B+树的例子直接见PPT（增删改查）

### 1. B+ tree vs B tree

相比于b树，区别在于

- 1. 非叶子节点是索引节点，只存储部分key和子节点地址，不存储value
- 2. 所有key和value都会存储在叶子节点中
- 3. 叶子节点之间存在顺序指向

查找和更新和B树类似

插入和删除思想一致，但是略有差别。

- 优点
  - 快速查找
  - B+ :
    - 对scan (range query, 范围扫描顺序访问) 更加友好
    - 索引节点内单个页存储元素个数可以更多，降低树的深度，尤其适合value较大的场景
  - B : 查找更加加快 (不需要每次访问到子节点)
  - value热更新开销小
- 缺点
  - 插入操作慢
  - 空间放大率高 (空间利用率不高)，数据结构变更容易产生文件空洞
  - 不同节点随机存储在磁盘上，会产生大量的随机IO

## LSTM tree

### SSTABLE

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
- 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

### LSM-Tree : Log-Structured Merge Tree

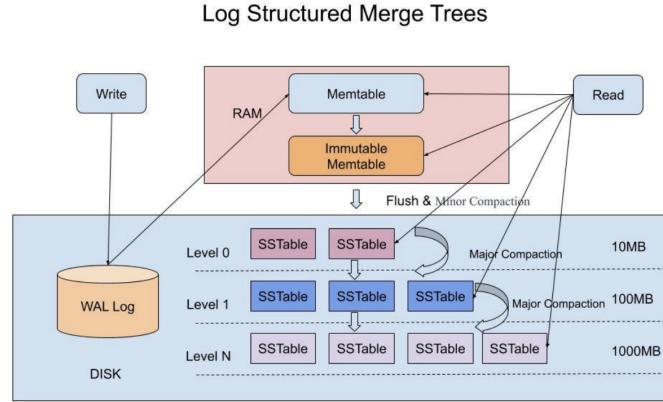
- 是一种分层、有序、面向磁盘的数据结构，其核心思想是充分利用磁盘批量顺序写性能远高于随机写性能的特点

### Log需要提高写入性能

- 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

- C0 树 (常驻内存)
- C1-N 树(位于磁盘)





### LSM tree/rocksdb 优点

- 大幅度提高插入(修改、删除)性能，具体表现为：低延迟插入，写入内存后直接返回，后台异步写入磁盘
- 空间放大率降低（数据紧密排列），但是有时间放大率（没有搜到的话要一直往下搜到底）。
- 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

### LSM tree/rocksdb 缺点

- 牺牲了读性能（一次可能访问多个层）
- 读、写放大率提升，读放大限制了AP查询的性能
- 写阻塞问题，降低了TP事务的可用性。

1. 什么是写阻塞问题：L0层满时将阻塞内存到磁盘的Flush过程 • L0层下沉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TP事务的可用性.
2. 什么是读放大、写放大问题：

读放大(Read Amplification)问题：log-structure db一般使用层次化的存储。不同层级存储着不同版本的数据，需要一层一层（从上到下）查找，直到找到需要的数据。这个过程可能需要不止一次的I/O，有可能寻找一份数据需要遍历所有的数据文件。特别是range query，影响特别明显。

eg, 在LSMtree中，如果查询一个没有在数据库的值，需要遍历完所有的文件才知道这个数据不在数据库中。

写放大 (Write amplification) 问题：实际写入 HDD/SSD 的数据大小和程序要求写入数据大小之比。正常情况下，HDD/SSD 观察到的写入数据多于上层程序写入的数据。由于写数据可能触发合并，压缩操作，写一个数据可能导致很多层的合并压缩，甚至可能到最后一层的合并压缩。

### 3.解决读放大问题：

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。

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

- o 以文件为格式调整粒度

- o 额外记录每个文件的历史操作

- o 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

- o 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

- o 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.请阐述日志结构数据库适合什么样的应用场景？(1分)

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。

因为WAL是数据库文件，如果遇到crash,可以在执行过程中重建丢失的索引，保持事务的一致性，因此也适用于**需要开启很多事务且需要高容错**的系统。



In computer science, the **log-structured merge-tree** (also known as **LSM tree**, or **LSMT**[1]) is a data structure with performance characteristics that make it attractive for providing indexed access to files with **high insert volume**, such as transactional log data. (适合大量插入数据的场景) —— from Wikipedia Log-structured merge-tree

### 4. OLAP & OLTP 是什么？

LTP (on-line transaction processing) 翻译为联机事务处理，OLAP (On-Line Analytical Processing) 翻译为联机分析处理，从字面上来看OLTP是做事务处理，OLAP是做分析处理。从对数据库操作来看，OLTP主要是对数据的增删改，OLAP是对数据的查询。

OLTP System		OLAP System
Online Transaction Processing		Online Analytical Processing
业务目的	处理业务，如订单、合同等	业务支持决策
面向对象	业务处理人员	分析决策人员
主要工作负载	增、删、改	查询
主要衡量指标	事务吞吐量	查询响应速度 (QPS)
数据库设计	3NF 或 BCNF	星型/雪花模型

## 5. 日志结构合并树中，WAL的作用是什么

WAL : Write Ahead Log

WAL主要作用是用来恢复发生 crash时 memtable中的未committed中的数据。所以WAL 的写入需要优先于memtable，且每一次写入都需要flush，这也是write head的由来。

对RocksDB 的每次写操作都必写到两个地方：

- 1) 基于内存的数据结构memtable（达到quota 后会flush 至SST file）。
- 2) 预写日志-Write Ahead Log (WAL)。

WAL 的好处：

1. 如果出现异常情况，WAL 可以用来完整恢复memtable 中的数据，恢复db 的原有的状态。  
通过每次用户写之后flush WAL，来保证进程crash 后的一致性。
2. WAL可以减少DB的写入操作， 达到更好的效率。

## 6. 行存储和列存储的区别

行式存储的适用场景包括：

- 1、适合随机的增删改查操作;
- 2、需要在行中选取所有属性的查询操作;
- 3、需要频繁插入或更新的操作，其操作与索引和行的大小更为相关。
- 4、适合OLAP

列式存储引擎的适用场景包括：

- 1、查询过程中，可针对各列的运算并发执行(SMP)，最后在内存中聚合完整记录集，最大可能降低查询响应时间;

2、可在数据列中高效查找数据，无需维护索引(任何列都能作为索引)，查询过程中能够尽量减少无关IO，避免全表扫描；

3、因为各列独立存储，且数据类型已知，可以针对该列的数据类型、数据量大小等因素动态选择压缩算法，以提高物理存储利用率；如果某一行的某一列没有数据，那在列存储时，就可以不存储该列的值，这将比行式存储更节省空间。

#### 4、适合OLTP

当然，跟行数据库一样，列式存储也有不太适用的场景，主要包括：

- 1、数据需要频繁更新的交易场景
- 2、表中列属性较少的小量数据库场景
- 3、不适合做含有删除和更新的实时操作

## 7. 基于LSTM的数据库优化

已有的行列混合存储的研究针对非LSM-Tree的数据库：

常见的列式存储格式：

- Row Group模式
- 行列折中方案

常见的行列混合存储决策算法：

- 行列对等存储 行列都存
- 行列差异存储 有些是行有些是列
- 基于查询的分析

基于LSM-Tree的数据库进行行列混合存储的研究尚属热点

基于LSM-Tree的KV数据库优化主要针对写放大：

- 同级内增加局部分区(PebblesDB)
- 块大小优化

不同level存储策略优化

- 混合存储结构可以参考的方面：
- 根据冷热数据进行区分行列，如更新频率，读频率
- 基于历史查询的分析

# lect18: Timeseries Database

## 1. 定义：

A time series database (TSDB) is a database optimized for time-stamped or time series data.

- Time series data are simply measurements or events that are tracked, monitored, downsampled, and aggregated over time.
- This could be server metrics, application performance monitoring, network data, sensor data, events, clicks, trades in a market, and many other types of analytics data.

A time series database is built specifically for handling metrics and events or measurements that are time-stamped.

- A TSDB is optimized for measuring change over time.
- Properties that make time series data very different than other data workloads are data lifecycle management, summarization, and large range scans of many records.

2. 时序数据库的应用场景：

- a. 该类数据以时间排序
- b. 由于该类数据通常量级大（因此Sharding和Scale非常重要）或逻辑复杂（大量聚合，上取，下钻），关系数据库通常难以处理

## InfluxDB

- InfluxDB data elements
  - InfluxDB structures data using elements such as **timestamps**, **field keys**, **field values**, **tags**, etc.
  - The sample data below is used to illustrate data elements concepts

bucket: my\_bucket

可以很精确

_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

合起来才是这个时间点得到的所有数据

field set

field set

timestamps measurement Tag value Tag value Field key Filed value

### Timestamp

时间就很重要了

### Measurement

A measurement acts as a container for tags, fields, and timestamps.

一个容器，就是把数据分组之后描述一下。

### field set

一个时间点采到的所有数据的和

**field**：不建立索引，搜索是扫描全表。不建立索引的原因：数据存储的方式，增量式存储，每隔一段时间记录一个原始值。存一个10000，之后是-0.001，+2.99，所以这样就很难建立索引了。

**tag**：存完整数据，这样就比较大了。

Q : tag和field的差异？

A: field 没有建立索引，tag建立了索引速度加快tag是人为设定的。对于经常搜索的字段需要独立出来变成tag。

### Fields aren't indexed:

- Fields are required in InfluxDB data and are **not** indexed.
- Queries that filter field values must **scan all field values** to match query conditions.
- As a result, queries on tags > are more performant than queries on fields.
- **Store commonly queried metadata in tags.**

### Tags are indexed:

- Tags are **optional**.
- You **don't** need tags in your data structure, but it's typically a good idea to include tags.
- Because tags are indexed, queries on tags are **faster** than queries on fields.
- This makes tags ideal for storing commonly-queried metadata.

tag和field可以相互转换，和关系型的行列转化一样的。关系型里面只有在列上做索引，行不能做索引。

## InfluxDB design principle

## Time-ordered data

- To improve performance, data is written in **time-ascending order**.

## Strict update and delete permissions

- To increase query and write performance, InfluxDB **tightly restricts update and delete permissions**.
- Time series data is **predominantly new data** that is never updated.
- Deletes generally only affect data that isn't being written to, and **contentious updates never occur**.

## Handle read and write queries first

- InfluxDB prioritizes read and write requests **over** strong consistency.
- InfluxDB returns results when a query is executed.
- Any transactions that affect the queried data are processed **subsequently** to ensure that data is eventually consistent.
- Therefore, if the ingest rate is high (multiple writes per ms), query results may **not** include the most recent data.

## Schemaless design

- InfluxDB uses a **schemaless design** to better manage discontinuous data.
- Time series data are often ephemeral, meaning the data appears for a few hours and then goes away. For example, a new host that gets started and reports for a while and then gets shut down.

## Datasets over individual points

- Because the data set is more important than an individual point, InfluxDB implements powerful tools to aggregate data and handle large data sets.
- Points are differentiated by timestamp and series, so **don't** have IDs in the traditional sense.

## Duplicate data

- To simplify conflict resolution and increase write performance, InfluxDB assumes **data sent multiple times is duplicate data. Identical points aren't stored twice**.
- If a new field value is submitted for a point, InfluxDB updates the point with the most recent field value. In rare circumstances, data may be overwritten.

大规模的数据都是只读，严格限制删除和更新。

每次查询query就停住插入，保证查到的和数据库的内容一样。

如果同一个field, tag, time都一样，但是数值不一样，influxdb认为time已经缩小到很小了，这样的概率很小，就会被筛掉。

## **InfluxDB storage engine**

The storage engine includes the following components:

- WriteAheadLog(WAL)
- Cache
- Time-StructuredMergeTree(TSM) 和LSM结构都是分层
- Time Series Index(TSI) 对tag,field , timesatmp整个建立索引

写放大：和LSM一样，写的合并要合并到最后一层。 (3 : 03)

写的数据需要压缩数据。写数据的时候不要一条一条写，而是踩到很多数据之后合并成一个batch，一起写入。eg. floom 这样的工具

cache: 原始数据是不压缩的，只有老的数据才会压缩（合并的时候）。cache相当于LSM 的skiplist 每次都从它这里开始查询。

## InfluxDB shards and shard groups

shard: 按照时间顺序分shard.

– A shard contains encoded and compressed time series data for a given time range defined by the shard group duration.

shard group: 一段时间的shard.

shard的增删改查见PPT.

# lect20: Data Lake

能够了解主流云原生数据库的基本概念和特点，理解数据湖的特征的技术难点

概念：

Data lake是指使用大型二进制对象或文件这样的自然格式储存数据的系统，通常把所有的企业数据统一存储，既包括源系统中的原始副本，也包括转换后的数据

—— Wikipedia

是一种不断演进中、可扩展的大数据存储、处理、分析的基础设施

以数据为导向，实现任意来源、速度、  
规模、类型数据的全生命周期管理



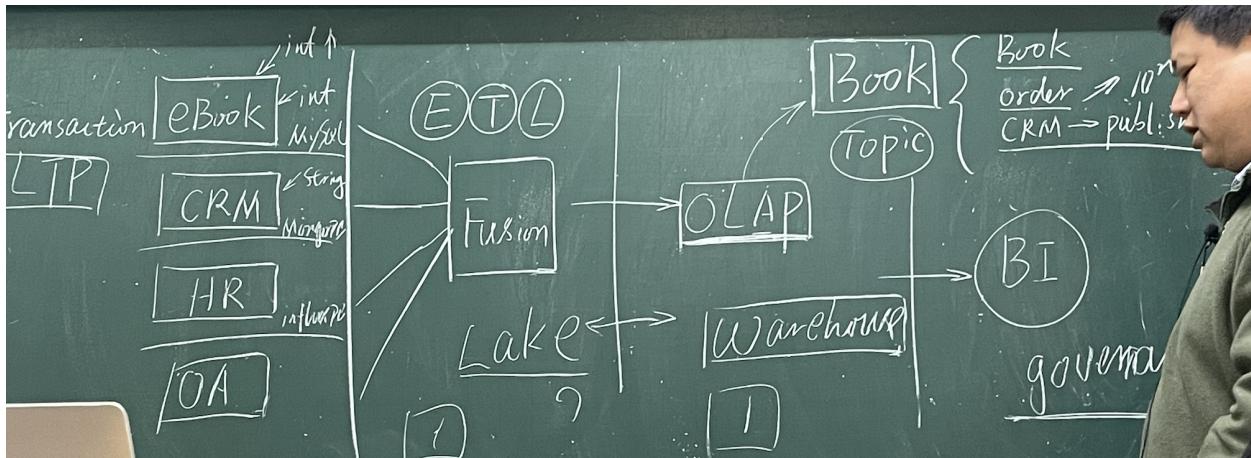
上课提到：

数据仓库：数据要先进行ETL，清理聚合才放进去

数据湖：直接存二进制，要分析的时候再抽取出来放到数据仓库里面。它面向的是一个企业的大量的数据，有些数据甚至都不需要被分析

但是问题是，如果所有的数据都被分析了，那从湖到仓库就复制了一份，所以提出了湖仓一体，数据在从湖到仓来回迁移，数据有一份。还有人说仓变成湖的缓存。

Data warehouse	Data lake
数据体系严格，提前建模	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，各引擎有限优化



一个公司有各种各样的数据库，但是要存的话都直接用二进制文件存，之后要用的话再转移到数据仓库里面

## 1. 数据湖和数据仓库的区别：

- 在数据湖中没有特定的预定架构，它可以轻松容纳结构化或非结构化数据。但数据仓库却不是这种情况，数据仓库通常由确定的架构组成并处理主数据。
- 在数据湖中，无论其结构如何，都可以存储数据，并以原始形式保存数据，直到需要使用为止。但是在数据仓库中，提取的数据组成了定量指标，其中对数据进行了清理和转换。
- 数据湖具有存储所有数据的能力，可以存储当前数据和将来需要使用的数据。在数据仓库中，需要花费大量时间专门用于分析多个源。
- 数据湖可以收集所有类型的数据，包括结构化和非结构化。但是，在数据仓库中，它会收集结构化数据并将其按照专门为数据仓库设计的架构进行排列。
- 数据湖包含所有类型的数据，并促使用户在处理和清除数据之前访问数据。数据仓库提供对预定义数据类型的预定义问题的见解。

## 2. 什么是湖仓一体？

湖仓一体（KeenData LakeHouse）是一种新型开放式架构，将数据湖和数据仓库的优势充分结合，它构建在数据湖低成本的数据存储架构之上，又继承了数据仓库的数据处理和管理功能。作为新一代大数据技术架构，将逐渐取代单一数据湖和数据仓库架构。

湖仓一体的关键属性：

- 1) 事物支持：在企业级应用中，支持ACID满足大量的SQL并发读写
- 2) 模式实施和治理：湖仓一体有一种支持模式实施和演变的方法，支持 DW 模式规范，能对数据类型做推断，并有数据治理和审计机制。
- 3) BI 工具支持：湖仓一体可以直接在源数据上使用BI工具，免去像之前在数据仓库和数据湖中拷来拷去的操作，减少陈旧度和等待时间，提高新近度，并且降低必须在数据湖和仓库中操作两个数据副本的成本。
- 4) 存储与计算分离：支持更大的并发量和更大的数据规模。
- 5) 兼容性：湖仓一体使用的存储格式是开放式和标准化的，它提供了多种 API，各种工具和引擎都可以直接有效地访问数据。
- 6) 支持从非结构化数据到结构化数据的多种数据类型：湖仓一体可用于存储、优化、分析和访问许多新数据应用程序所需的数据类型，包括图像，视频，音频，半结构化数据和文本。
- 7) 支持各种工作场景：包括数据科学，机器学习和 SQL 分析。这些可能依赖于多种工具来支持的工作场景，它们都依赖于相同的数据存储库。

## lect21: Clustering

能够根据业务需求和系统被访问的模式，设计合理的集群部署方式和负载均衡策略

负载均衡的策略：

1. 轮流 2. 最小负载 3. 随机

会话粘滞性：

all requests in a client's session are directed to the same server.

会话粘滞性问题：loadbalance 一开始把一个session开到Aserver上，如果下次调到B上，A上面写的看不见了。

方法1：无论A多忙每次都调用到同一台机器上，这样破坏了负载均衡的意义

方法2：session 放到一个第三方上（redis），redis成为了瓶颈如果redis 崩坏了就die了

# nginx

## Load balancing methods

- The following load balancing mechanisms (or methods) are supported in nginx:
  - **round-robin** — requests to the application servers are distributed in a round-robin fashion,  
只考虑连接数，但是每个连接化多少时间不看
  - **least-connected** — next request is assigned to the server with the least number of active connections,
  - 把用户的ip hash一下 这样就不均衡了  
**ip-hash** — a hash-function is used to determine what server should be selected for the next request (based on the client's IP address).

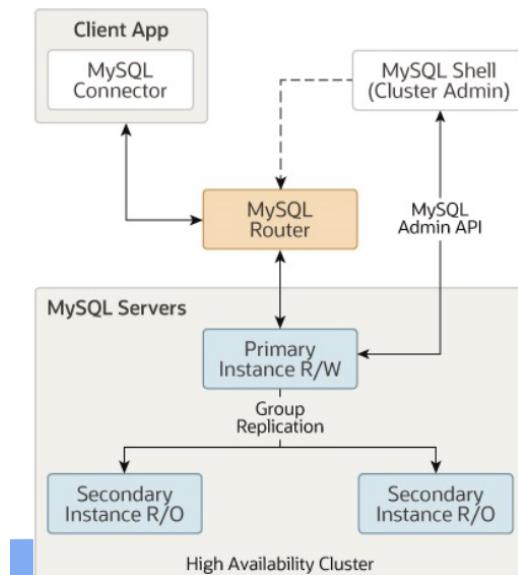
小集群：ip\_hash, 不保证负载均衡，保证session一定在一个机器上

大集群：第三方（redis、memcache），有一台机器来存redis，但是这样的话redis就成为了单一故障节点，需要多备份一台机器。

## MySQL InnoDB cluster

上面讲了tomcat如何建立集群，那么数据库如何集群？

写只能在主节点，读可以负载均衡



## primary和secondary的备份

默认刚开始的是primary，后来加入的是secondary

但是如果一个secondary一直是secondary就consistency不太好，最好是每隔一段时间secondary和primary角色互换。

cold 冷备份：secondary一直是secondary，由primary同步

warm 暖备份：每隔一段时间secondary和primary角色互换。

hot 热备份：一个事务在primary和secondary都跑，如果返回两个返回值，丢了一个就行。

cold → warm → hot : 可靠性+++ 耗电+++

### Q：热备份、冷备份、暖备份对比

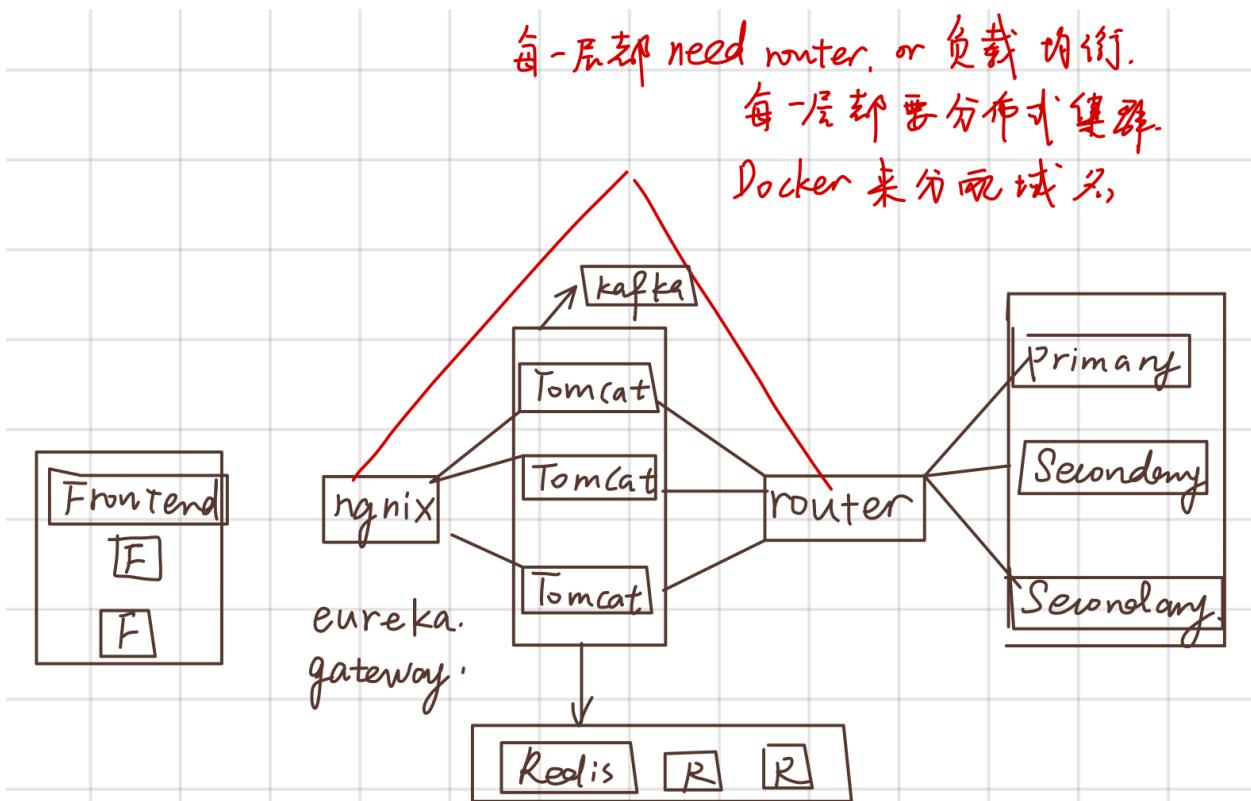
冷备份：备份系统未安装或未配置成与当前使用的系统相同或相似的运行环境，应用系统数据没有及时装入备份系统。一旦发生灾难，需安装配置所需的运行环境，用数据备份介质(磁带或光盘)恢复应用数据，手工逐笔或自动批量追补孤立数据，将终端用户通过通讯线路切换到备份系统，恢复业务运行 优点：设备投资较少，节省通信费用，通信环境要求不高 缺点：恢复时间较长，一般要数天至1周，数据完整性与一致性较差

2> 暖备份：将备份系统已安装配置成与当前使用的系统相同或相似的系统和网络运行环境，安装应用系统业务定期备份数据。一旦发生灾难，直接使用定期备份数据，手工逐笔或自动批量追补孤立数据或将终端用户通过通讯线路切换到备份系统，恢复业务运行 优点：设备投资较少，通信环境要求不高 缺点：恢复时间长，一般要十几个小时至数天，数据完整性与一致性较差

3> 热备份：备份处于联机状态，当前应用系统通过高速通信线路将数据实时传送到备份系统，保持备份系统与当前应用系统数据的同步；也可定时在备份系统上恢复应用系统的数据。一旦发生灾难，不用追补或只需追补很少的孤立数据，备份系统可快速接替生产系统运行，恢复营业 优点：恢复时间短，一般几十分钟到数小时，数据完整性与一致性最好，数据丢失可能性最小 缺点：设备投资大，通信费用高，通信环境要求高，平时运行管理较复杂

## 改变拓扑结构

可以多primary



总结：tomcat 集群：维护内存里的状态

database 集群：维护持久化状态

## lect 22: Virtualization & Container

能够根据系统容器化部署的需求，将本地应用迁移到云中容器集群中进行部署

1. 半虚拟化vs完全虚拟化 Paravirtualization vs. Hardware VirtualMachine

### 1、半虚拟化(PV)

半虚拟化（Paravirtualization）有些资料称为“超虚拟化”，简称为PV，是Xen主导的虚拟化技术。这种技术允许虚拟机操作系统感知到自己运行在Xen Hypervisor上而不是直接运行在硬件上，同时也可以识别出其他运行在相同环境中的客户虚拟机。

在Xen Hypervisor上运行的半虚拟化的操作系统，为了调用系统管理程序（Xen Hypervisor），要有选择地修改操作系统，然而却不需要修改操作系统上运行的应用程序。由于Xen需要修改操作系统内核，所以您不能直接让当前的Linux内核在Xen系统管理程序中运行，除非它已经移植到了Xen架构。不过，如果当前系统可以使用新的已经移植到Xen架构的Linux内核，那么您就可以不加修改地运行现有的系统。

### 2、完全虚拟化(HVM)

完全虚拟化（Hardware Virtual Machine）又称“硬件虚拟化”，简称HVM，是指运行在虚拟环境上的虚拟机在运行过程中始终感觉自己是直接运行在硬件之上的，并且感知不到在相同硬件环境下运行着其他虚拟机的虚拟技术。

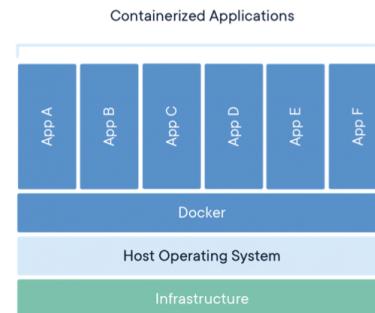
在Xen Hypervisor运行的完全虚拟化虚拟机，所运行的操作系统都是标准的操作系统，即：无需任何修改的操作系统版本。同时也需要提供特殊的硬件设备。

值得注意的是，在Xen上虚拟的Windows虚拟机必须采用完全虚拟化技术。

## 2. 什么是容器

- **What is a Container?** <https://www.docker.com>

- A **container** is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.
- A **Docker container image** is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.
- **Container images become containers at runtime** and in the case of Docker containers - images become containers when they run on **Docker Engine**.
- Containers **isolate** software from its environment and ensure that it works uniformly despite differences for instance between development and staging.



10

- 容器是软件的标准单元，它将代码及其所有依赖项打包，以便应用程序在一个计算环境到另一个计算环境之间快速可靠地运行。
- Docker容器镜像是一个轻量级的、独立的、可执行的软件包，它包含运行应用程序所需的一切：代码、运行时、系统工具、系统库和设置。
- 容器镜像在运行时变成容器，在Docker容器的情况下-镜像在Docker引擎上运行时变成容器。
- 容器将软件从它的环境中隔离出来，并确保它一致地工作，尽管在开发和登台之间存在差异。

一些细节：

在现有的image上面开发应用。

run 先在本地找，找不到到远端拉。

container之间是隔离的，而且每一个有独立的IP。

## Container Volume

With the previous experiment, we saw that each container starts from the image definition each time it starts.

- While containers can create, update, and delete files, those changes are **lost** when the container is **removed** and all changes are isolated to that container.
- With **volumes**, we can change all of this.

**Volumes** provide the ability to connect specific filesystem paths of the container back to the host machine.

- If a directory in the container is mounted, changes in that directory are also seen on the host machine.
- If we mount that same directory across container restarts, we'd see the same files.

There are two main types of volumes.

- We will eventually use both, but we will start with **named volumes**.

docker的持久化，如果docker停了，新写的文件没了。要带一个参数保存这个文件。没有存到image

Volume就是一些卷，相当于硬盘一样，image可以写到不同的volume，如果image被销毁了，重启可以得到数据，挂在不同的volume可以获得不同的数据。

持久化方法1：让docker给你管理volume的地址，如果不指定volumeName，将随机生成一个文件→name volume

持久化方法2：自己指定volume地址→bind mount

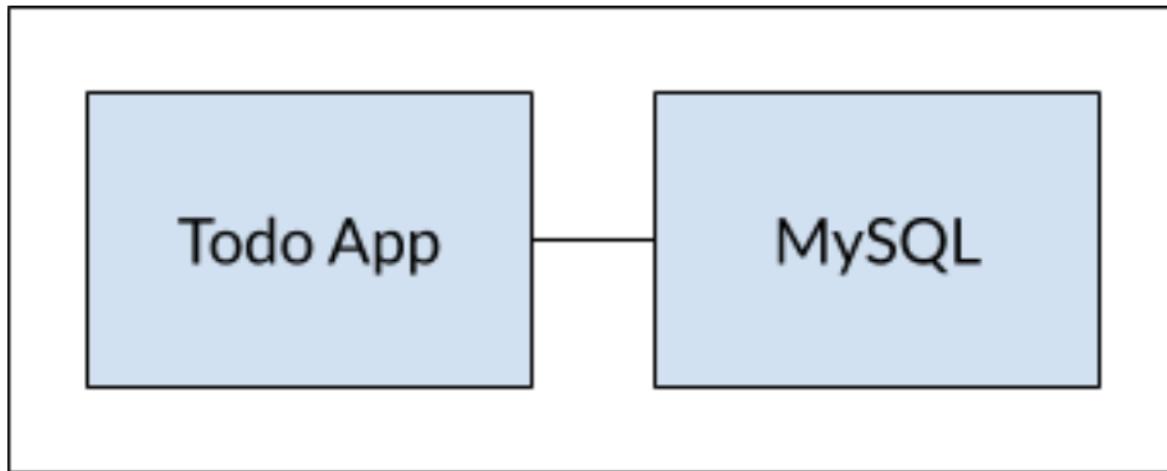
### Volume的适用场景

- 多个容器间需要共享数据。如果volume没有手动被创建，它将会在首次挂载到某个容器之前被自动创建，当容器被停止或删除时，这个volume不会随之被删除。多个容器可以同时以rw或ro的方式挂载这个volume。只有手动指定删除volume，它才会被删除。
- 当宿主机并没有专用于Docker的文件系统结构时。使用volume可以使宿主机的配置与容器的运行解耦。
- 当你希望将数据保存到远程主机或云上。
- 当你希望在不同的宿主机直接备份/恢复/迁移数据时，volume是一个很好的选择。你可以停止运行使用volume的容器，然后直接备份volume所在的目录即可，如`/var/lib/docker/volumes/<volume-name>`

### bind mounts的适用场景

- 将宿主机的系统配置文件共享给容器，这是Docker为容器提供DNS配置的默认方式，即通过bind mounts的方式将宿主的`/etc/resolv.conf`文件挂载到容器中。
- 将宿主机开发环境中的源代码或实验结果共享给容器。例如：你在宿主机上进行一个项目Maven的测试，每次你在宿主机上对Maven项目进行了更改后，容器就可以直接获取更改后的结果
- 当你可以确定宿主机的文件系统结构应该与容器内部完全一致时。

## Multi container apps



这两个是一个pool, 跑K8S 调用的最小颗粒。

1. 可能MySQL后端跑两个，这样放在一起不合适
2. 如果一个崩了，放在一起整个都崩了
3. image 是分层的。每一层可能有些东西是完全一样的，在pull的时候只pull不一样的地方。
4. 不同的container之间通过网络通信net-working。
  - a. `docker network create todo-app` 创造了叫做todo-app 的逻辑网络。

## 多个容器一起启动：compose

因为启动的循序也是对应用有关的。如果先启动数据库，再启动应用可能报错了。通过写docker-compose.yml 就可以指定顺序。

[Docker Compose](#) is a tool that was developed to help define and share multi-container applications.

- With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

The **big** advantage of using Compose is

- you can define your application stack in a file, keep it at the root of your project repo (it's now version controlled), and easily enable someone else to contribute to your project.
- Someone would only need to clone your repo and start the compose app. In fact, you might see quite a few projects on GitHub/GitLab doing exactly this now.

# lect23:Kubernetes

能够根据系统容器化部署的需求，了解K8S的基本原理

Kubernetes 是一个可移植的、可扩展的开源平台，用于管理容器化的工作负载和服务，可促进声明式配置和自动化。Kubernetes 拥有一个庞大且快速增长的生态系统。Kubernetes 的服务、支持和工具广泛可用。

**容器部署时代：**

容器类似于 VM，但是它们具有被放宽的隔离属性，可以在应用程序之间共享操作系统（OS）。因此，容器被认为是轻量级的。容器与 VM 类似，具有自己的文件系统、CPU、内存、进程空间等。由于它们与基础架构分离，因此可以跨云和 OS 发行版本进行移植。

容器因具有许多优势而变得流行起来。下面列出的是容器的一些好处：

- 敏捷应用程序的创建和部署：与使用 VM 镜像相比，提高了容器镜像创建的简便性和效率。
- 持续开发、集成和部署：通过快速简单的回滚（由于镜像不可变性），支持可靠且频繁的容器镜像构建和部署。
- 关注开发与运维的分离：在构建/发布时而不是在部署时创建应用程序容器镜像，从而将应用程序与基础架构分离。
- 可观察性：不仅可以显示操作系统级别的信息和指标，还可以显示应用程序的运行状况和其他指标信号。
- 跨开发、测试和生产的环境一致性：在便携式计算机上与在云中相同地运行。
- 跨云和操作系统发行版本的可移植性：可在 Ubuntu、RHEL、CoreOS、本地、Google Kubernetes Engine 和其他任何地方运行。
- 以应用程序为中心的管理：提高抽象级别，从在虚拟硬件上运行 OS 到使用逻辑资源在 OS 上运行应用程序。
- 松散耦合、分布式、弹性、解放的微服务：应用程序被分解成较小的独立部分，并且可以动态部署和管理 - 而不是在一台大型单机上整体运行。
- 资源隔离：可预测的应用程序性能。
- 资源利用：高效率和高密度。

## 为什么需要 Kubernetes，它能做什么？

容器是打包和运行应用程序的好方式。在生产环境中，你需要管理运行应用程序的容器，并确保不会停机。例如，如果一个容器发生故障，则需要启动另一个容器。如果系统处理此行为，会不会更容易？

这就是 Kubernetes 来解决这些问题的方法！Kubernetes 为你提供了一个可弹性运行分布式系统的框架。Kubernetes 会满足你的扩展要求、故障转移、部署模式等。例如，Kubernetes 可以轻松管理系统的 Canary 部署。

Kubernetes 为你提供：

- **服务发现和负载均衡**

Kubernetes 可以使用 DNS 名称或自己的 IP 地址公开容器，如果进入容器的流量很大，Kubernetes 可以负载均衡并分配网络流量，从而使部署稳定。

- **存储编排**

Kubernetes 允许你自动挂载你选择的存储系统，例如本地存储、公共云提供商等。

- **自动部署和回滚**

你可以使用 Kubernetes 描述已部署容器的所需状态，它可以以受控的速率将实际状态 更改为期望状态。例如，你可以自动化 Kubernetes 来为你的部署创建新容器，删除现有容器并将它们的所有资源用于新容器。

- **自动完成装箱计算**

Kubernetes 允许你指定每个容器所需 CPU 和内存（RAM）。当容器指定了资源请求时，Kubernetes 可以做出更好的决策来管理容器的资源。

- **自我修复**

Kubernetes 重新启动失败的容器、替换容器、杀死不响应用户定义的 运行状况检查的容器，并且在准备好服务之前不将其通告给客户端。

- **密钥与配置管理**

Kubernetes 允许你存储和管理敏感信息，例如密码、OAuth 令牌和 ssh 密钥。你可以在不重建容器镜像的情况下部署和更新密钥和应用程序配置，也无需在堆栈配置中暴露密钥。

Kubernetes provides you with:

- Service discovery and load balancing.
- Storage orchestration.
- Automated rollouts and rollbacks.
- Automatic bin packing. 分配让物理机的负载一样，或者由用户请求来决定。
- Self-healing. 物理机如果不能正常运行了，要回滚。这个物理机自己释放自己的内容，重新要求 K8S 分配一个新的物理机跑这个程序。
- Secret and configuration management.

## Kubernetes 组件

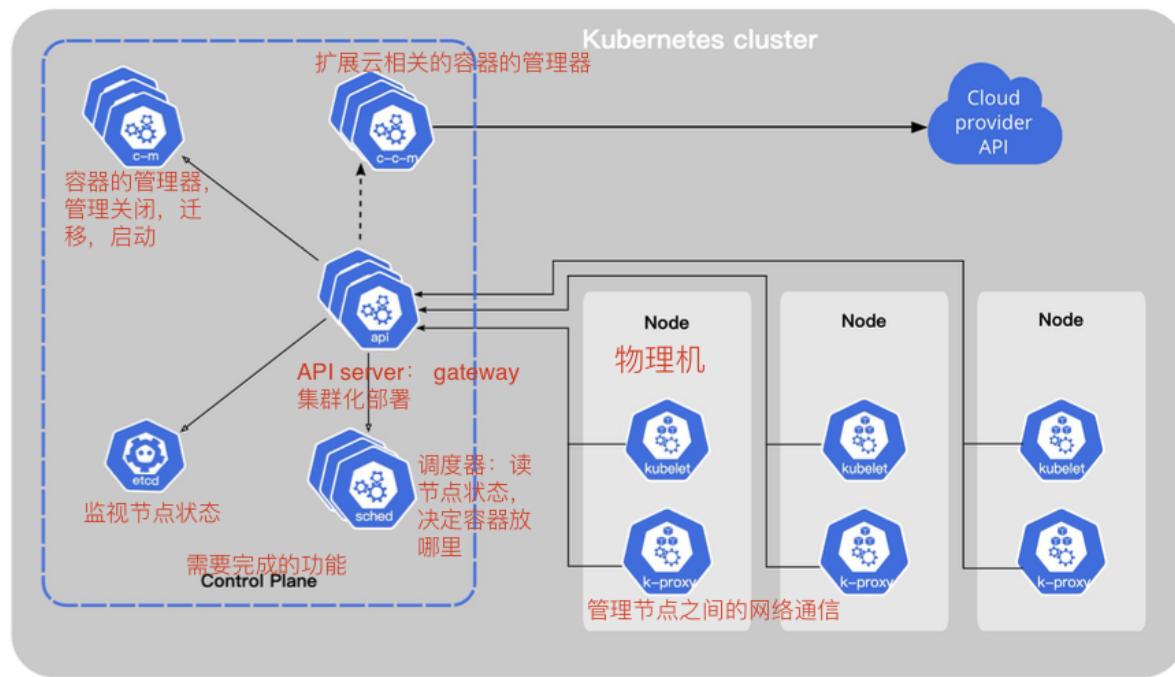
当你部署完 Kubernetes，即拥有了一个完整的集群。

一个 Kubernetes 集群由一组被称作节点的机器组成。这些节点上运行 Kubernetes 所管理的容器化应用。集群具有至少一个工作节点。

工作节点托管作为应用负载的组件的 Pod。控制平面管理集群中的工作节点和 Pod。为集群提供故障转移和高可用性，这些控制平面一般跨多主机运行，集群跨多个节点运行。

本文档概述了交付正常运行的 Kubernetes 集群所需的各种组件。

Kubernetes 集群的组件



## 控制平面组件 (Control Plane Components)

控制平面的组件对集群做出全局决策(比如调度)，以及检测和响应集群事件（例如，当不满足部署的 `replicas` 字段时，启动新的 pod）。

控制平面组件可以在集群中的任何节点上运行。然而，为了简单起见，设置脚本通常会在同一个计算机上启动所有控制平面组件，并且不会在此计算机上运行用户容器。请参阅[使用 kubeadm 构建高可用性集群](#)中关于多 VM 控制平面设置的示例。

### **kube-apiserver**

API 服务器是 Kubernetes 控制面的组件，该组件公开了 Kubernetes API。API 服务器是 Kubernetes 控制面的前端。

Kubernetes API 服务器的主要实现是 kube-apiserver。kube-apiserver 设计上考虑了水平伸缩，也就是说，它可通过部署多个实例进行伸缩。你可以运行 kube-apiserver 的多个实例，并在这些实例

之间平衡流量。

### **etcd**

etcd 是兼具一致性和高可用性的键值数据库，可以作为保存 Kubernetes 所有集群数据的后台数据库。

您的 Kubernetes 集群的 etcd 数据库通常需要有个备份计划。

要了解 etcd 更深层次的信息，请参考 [etcd 文档](#)。

### **kube-scheduler**

控制平面组件，负责监视新创建的、未指定运行节点（node）的 Pods，选择节点让 Pod 在上面运行。

调度决策考虑的因素包括单个 Pod 和 Pod 集合的资源需求、硬件/软件/策略约束、亲和性和反亲和性规范、数据位置、工作负载间的干扰和最后时限。

### **kube-controller-manager**

运行控制器进程的控制平面组件。

从逻辑上讲，每个控制器都是一个单独的进程，但是为了降低复杂性，它们都被编译到同一个可执行文件，并在一个进程中运行。

这些控制器包括：

- 节点控制器（Node Controller）：负责在节点出现故障时进行通知和响应
- 任务控制器（Job controller）：监测代表一次性任务的 Job 对象，然后创建 Pods 来运行这些任务直至完成
- 端点控制器（Endpoints Controller）：填充端点（Endpoints）对象（即加入 Service 与 Pod）
- 服务帐户和令牌控制器（Service Account & Token Controllers）：为新的命名空间创建默认帐户和 API 访问令牌

### **cloud-controller-manager**

云控制器管理器是指嵌入特定云的控制逻辑的控制平面组件。云控制器管理器使得你可以将你的集群连接到云提供商的 API 之上，并将与该云平台交互的组件同与你的集群交互的组件分离开来。

`cloud-controller-manager` 仅运行特定于云平台的控制回路。如果你在自己的环境中运行 Kubernetes，或者在本地计算机中运行学习环境，所部署的环境中不需要云控制器管理器。

与 `kube-controller-manager` 类似，`cloud-controller-manager` 将若干逻辑上独立的 控制回路组合到同一个可执行文件中，供你以同一进程的方式运行。你可以对其执行水平扩容（运行不止一个副本）以提升性能或者增强容错能力。

下面的控制器都包含对云平台驱动的依赖：

- 节点控制器（Node Controller）：用于在节点终止响应后检查云提供商以确定节点是否已被删除

- 路由控制器（Route Controller）：用于在底层云基础架构中设置路由
- 服务控制器（Service Controller）：用于创建、更新和删除云提供商负载均衡器

## Node 组件

节点组件在每个节点上运行，维护运行的 Pod 并提供 Kubernetes 运行环境。

### kubelet

一个在集群中每个节点（node）上运行的代理。它保证容器（containers）都运行在 Pod 中。

kubelet 接收一组通过各类机制提供给它的 PodSpecs，确保这些 PodSpecs 中描述的容器处于运行状态且健康。kubelet 不会管理不是由 Kubernetes 创建的容器。

### kube-proxy

[kube-proxy](#) 是集群中每个节点上运行的网络代理，实现 Kubernetes 服务（Service）概念的一部分。

kube-proxy 维护节点上的网络规则。这些网络规则允许从集群内部或外部的网络会话与 Pod 进行网络通信。

如果操作系统提供了数据包过滤层并可用的话，kube-proxy 会通过它来实现网络规则。否则，kube-proxy 仅转发流量本身。

### 容器运行时（Container Runtime）

容器运行环境是负责运行容器的软件。

Kubernetes 支持多个容器运行环境：[Docker](#)、[containerd](#)、[CRI-O](#) 以及任何实现 [Kubernetes CRI](#)（容器运行环境接口）。

### 插件（Addons）

插件使用 Kubernetes 资源（[DaemonSet](#)、[Deployment](#) 等）实现集群功能。因为这些插件提供集群级别的功能，插件中命名空间域的资源属于 `kube-system` 命名空间。

下面描述众多插件中的几种。有关可用插件的完整列表，请参见 [插件（Addons）](#)。

### DNS

尽管其他插件都并非严格意义上的必需组件，但几乎所有 Kubernetes 集群都应该有集群 DNS，因为很多示例都需要 DNS 服务。

集群 DNS 是一个 DNS 服务器，和环境中的其他 DNS 服务器一起工作，它为 Kubernetes 服务提供 DNS 记录。

Kubernetes 启动的容器自动将此 DNS 服务器包含在其 DNS 搜索列表中。

### Pods

*Pod* 是可以在 Kubernetes 中创建和管理的、最小的可部署的计算单元。

*Pod* (就像在鲸鱼英或者豌豆英中) 是一组 (一个或多个) 容器；这些容器共享存储、网络、以及怎样运行这些容器的声明。Pod 中的内容总是并置 (colocated) 的并且一同调度，在共享的上下文中运行。Pod 所建模的是特定于应用的“逻辑主机”，其中包含一个或多个应用容器，这些容器是相对紧密的耦合在一起的。在非云环境中，在相同的物理机或虚拟机上运行的应用类似于在同一逻辑主机上运行的云应用。

除了应用容器，Pod 还可以包含在 Pod 启动期间运行的 Init 容器。你也可以在集群中支持临时性容器的情况下，为调试的目的注入临时性容器。

A **workload** is an application running on Kubernetes.

pod 由多个容器构成

Whether your workload is a single component or several that work together, on Kubernetes you run it inside a set of **pods**.

### **workload**

工作负载是在 Kubernetes 上运行的应用程序。

无论你的负载是单一组件还是由多个一同工作的组件构成，在 Kubernetes 中你可以在一组 Pods 中运行它。在 Kubernetes 中，Pod 代表的是集群上处于运行状态的一组容器。

Kubernetes Pods 有确定的生命周期。例如，当某 Pod 在你的集群中运行时，Pod 运行所在的节点出现致命错误时，所有该节点上的 Pods 都会失败。Kubernetes 将这类失败视为最终状态：即使该节点后来恢复正常运行，你也需要创建新的 Pod 来恢复应用。

不过，为了让用户的日子略微好过一些，你并不需要直接管理每个 Pod。相反，你可以使用 负载资源 来替你管理一组 Pods。这些资源配置 控制器 来确保合适类型的、处于运行状态的 Pod 个数是正确的，与你所指定的状态相一致。

## **lect 24: Cloud Computing**

能够根据系统云部署的需求，将本地应用迁移到云中虚拟机集群中进行部署

云和云计算的定义：

There is little consensus on how to define the Cloud

- A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet.
  - In "*Cloud Computing and Grid Computing 360-Degree Compared*"
- Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS), so we use that term. The datacenter hardware and software is what we will call a Cloud.
  - In "*Above the Clouds: A Berkeley View of Cloud Computing*"
- Cloud computing is a general term for anything that involves delivering hosted services over the Internet.

云计算指的是：在Internet上作为服务交付的应用程序，以及提供这些服务的数据中心的硬件和系统软件。这两者

**Cloud 的特征：**

1. 按需收费
  2. 弹性扩展，可以增加资源
  3. 管理的复杂性交给了云的提供商
  4. 虚拟化，大家的需求（操作系统，内核数，资源）不一样，要模拟出这样的机子。
- Common attributes of clouds
- Flexible pricing  
不同的机器定价不同eg, 按照需要重启的时间定价 on-demand, reserved
  - Elastic scaling
  - Rapid provisioning (快速供给)
    - 拿到新的服务器之后马上部署，就像容器分层，一些基础的层大家的机器都有，一些依赖下载一下就可以。
  - Advanced virtualization

**云服务的种类**

SaaS	Software-as-a-Service	Google Apps, Microsoft "Software+Services" <a href="#">百度地图</a> , 支出宝API
PaaS	Platform-as-a-Service	IBM IT Factory, Google AppEngine, Force.com
IaaS	Infrastructure-as-a-Service	基础设施 Amazon EC2, IBM Blue Cloud, Sun Grid
dSaaS	data-Storage-as-a-Service	Nirvanix SDN, Amazon S3, Cleversafe dsNet 网盘

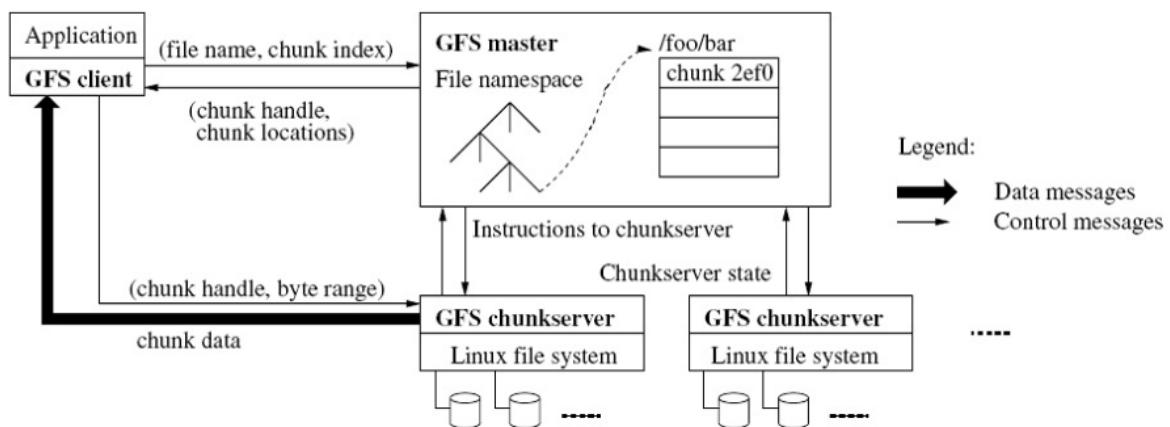
These services are broadly divided into three categories:

- Infrastructure-as-a-Service (IaaS)
- Platform-as-a-Service (PaaS)
- and Software-as-a-Service (SaaS).

## 作业调度 → mapreduce

batch data 处理的很好（数据已经有了），但是流数据不行。有频繁的IO磁盘操作，性能很差。  
map1-spill1.out, 是内存放不下，写到小文件。之后就放弃了java, 用scala, 函数式编程（一个变量被定义了之后不能改写了）

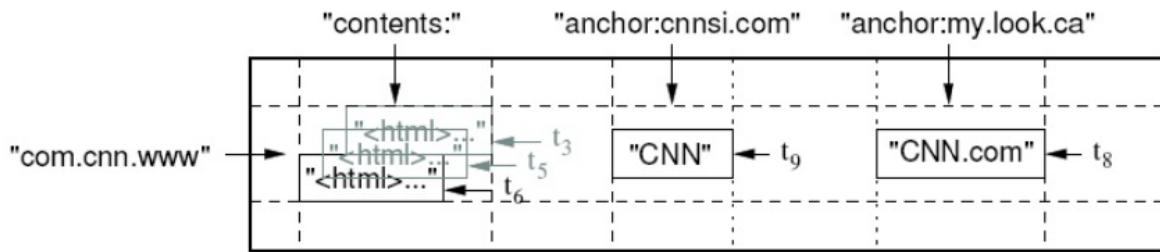
## 文件系统 → GFS



master存储这个文件在哪个chunck, 类似原来的文件系统说这个文件的block在哪里, 就是大文件切成小文件, 然后维护文件的存放位置。还有多备份, data flow 和control flow

## Bigtable

列有二级化的处理。列族, 可以按照时间戳



## hadoop



- The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing. Hadoop includes these subprojects:
  - **Hadoop Common**: The common utilities that support the other Hadoop subprojects.
  - **HDFS**: A distributed file system that provides high throughput access to application data.
  - **MapReduce**: A software framework for distributed processing of large data sets on compute clusters.
- IBM, Amazon, Yahoo
  - Base stone

批处理有基础开销，输入多的时候就适合用批处理。

## two phase commit —— google percolator

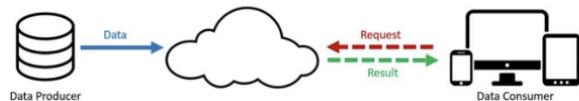
在表里加行锁，而不是锁住整个表

## edge computing

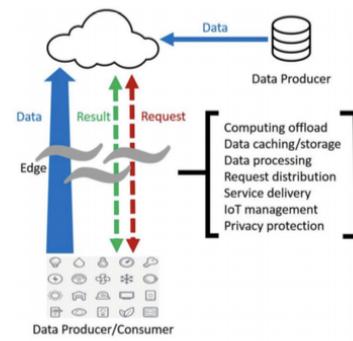
让摄像头直接连接的服务器计算，再之后摄像头就可以有计算能力。把计算的任务推到了网络的边缘，

定义：边缘计算，是指在靠近物或数据源头的一侧，采用网络、计算、存储、应用核心能力为一体的开放平台，就近提供最近端服务。其应用程序在边缘侧发起，产生更快的网络服务响应，满足行业在实时业务、应用智能、安全与隐私保护等方面的基本需求。

- **Cloud computing**



- **Edge computing**



终端到云的每一层都可处理。

数据在靠近他的地方处理效率高。就近存储，就近处理。

好处：

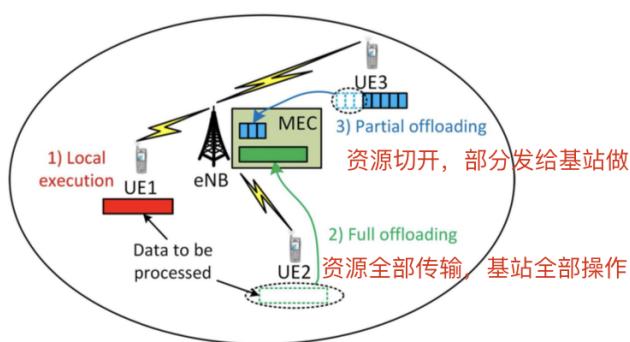
- This reduces the communications bandwidth needed between sensors and the central data center by performing analytics and knowledge generation at or near the source of the data.
- This approach requires leveraging resources that may not be continuously connected to a network such as laptops, smartphones, tablets and sensors

但是什么数据在云什么在边也是需要考虑的。

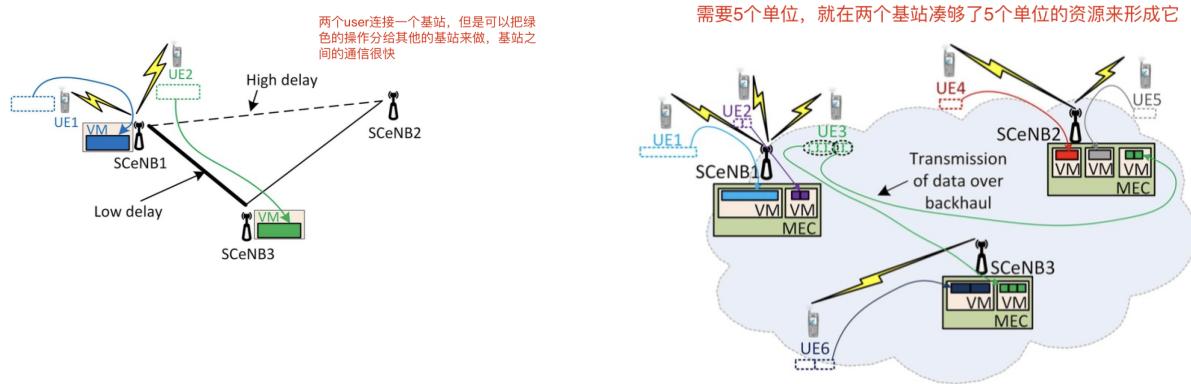
边缘计算兴起的背景：越来越多的终端计算能力增强。

### Computation Offloading (计算迁移)

好处：From the user perspective, a critical use case regarding the MEC is a computation offloading as this can save energy and/or speed up the process of computation.



这个基站要是做不了，还可以投到更上的基站做，这样可以比较充分的利用资源。

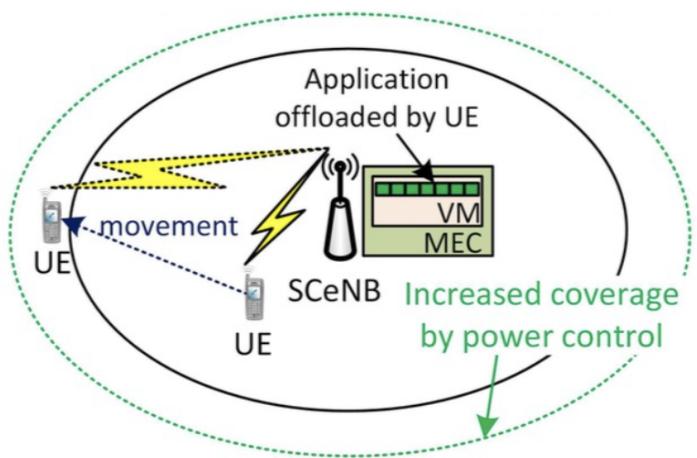


无限通信在于带宽是可变的，数据在传输的时候就带宽会不一样。

## Mobility Management

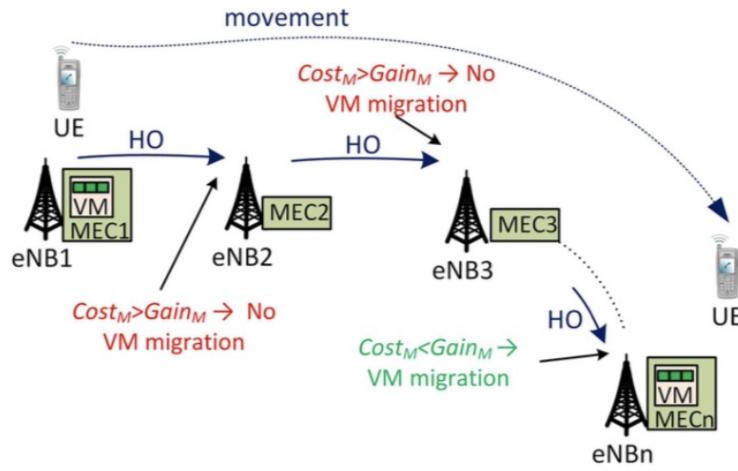
### 1. Power control

出了圈，能耗增大，带宽降低，这个时候是否需要换一个基站



### 2. VM migration

因为用户在移动，所以到了另外一个地区就要换了一个基站了，但是在城市里面，人的移动是有规律的，可以提前预测，就提前启动这个新的虚拟机。eg，人从家里到上班，在不同的基站做 prefetch，基于你的移动速度，路径等等。



summary

边缘计算就是雾计算，就是近在眼前的。

但是每一台的资源都是受限的，在云计算里面是看起来资源无限的

## lect25: Hadoop

– 能够针对大数据批处理需求，设计并实现基于MapReduce/YARN的并行处理方案

### Mapper

对于 Mapper 来说，它需要把输入的键值对映射成中间体的键值对表示，每个都是一个独立的工作。所以在 Hadoop 框架中，每个输入的 InputSplit 都会创建一个 map task。

## How Many Mappers?



- The number of maps is usually driven by the **total size of the inputs**, that is,
  - the **total number of blocks** of the input files.
- The right level of parallelism for maps seems to be around **10-100 maps per-node**,
  - although it has been set up to 300 maps for very cpu-light map tasks.
  - **Task setup takes a while**, so it is best if the maps take at least a minute to execute.
- Thus,
  - if you expect 10TB of input data and have a blocksize of **128MB**, you'll end up with 82,000 maps,
  - unless **Configuration.set(MRJobConfig.NUM\_MAPS, int)** (which only provides a hint to the framework) is used to set it even higher.

小文件一个文件一个mapper, 超过128分开block

## combiner

combiner 可以减少网络传输的重复数据，可以直接求和。在Word cout,就是在本地map结构做一下reduce的事情。但是combiner 的计算是有条件的，eg, 如果求单词的出现频率，不能每次都在中间求频率。

## reducer

Reducer分为三个主要的阶段shuffle, sort和reduce。

**shuffle:** the framework fetches the relevant partition of the output of all themappers, via HTTP.

**sort:** The framework groups Reducer inputs by keys (和sort 同时)

### Secondary Sort

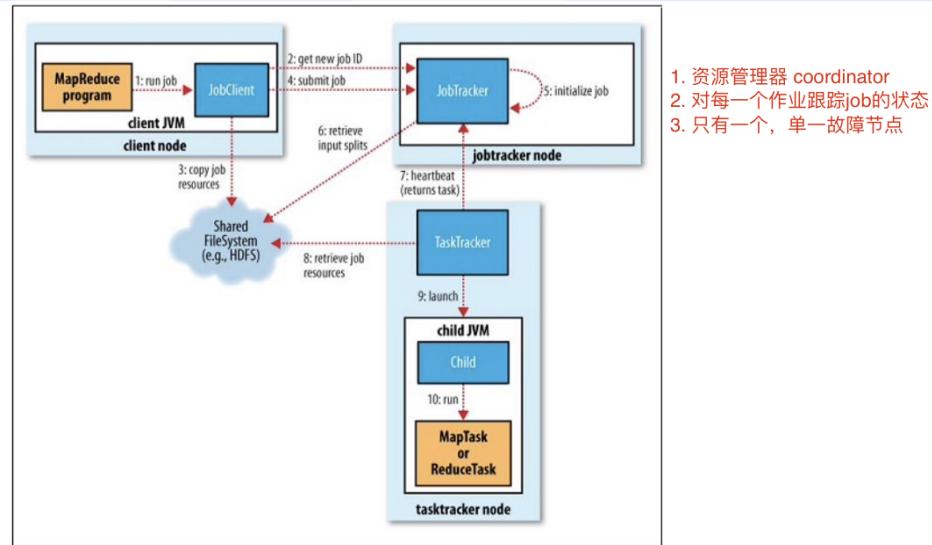
**Reduce :** In this phase the reduce(WritableComparable, Iterable \$<\$ Writable \$>\$, Context) method is called for each < key, (list of values)> pair in the grouped inputs.

The output of the reduce task is typically written to the FileSystem via Context.write(WritableComparable, Writable).

## How Many Reducers?



- The right number of reduces seems to
  - be **0.95** or **1.75** multiplied by (**<no. of nodes> \* <no. of maximum containers per node>**).
  - With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish.
  - With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.
- Increasing the number of reduces
  - increases the framework overhead,
  - but increases load balancing and lowers the cost of failures.
- The scaling factors above are slightly less than
  - whole numbers to reserve a few reduce slots in the framework for speculative-tasks and failed tasks.
- **Reducer NONE**
  - It is legal to set the number of reduce-tasks to **zero** if no reduction is desired.
  - In this case the outputs of the map-tasks go directly to the FileSystem, into the output path set by [FileOutputFormat.setOutputPath\(Job, Path\)](#).
  - The framework **does not sort** the map-outputs before writing them out to the FileSystem.
  - ...



但是这样有单一故障节点，不好，于是有了YARN，把资源管理器和作业监管分开来，就没有单一故障了，写代码的方式没有区别，只是

## Yarn

对旧的mapreduce框架的改进

- 将资源管理和作业控制分离，减小JobTracker压力
- YARN的设计大大减小了 JobTracker（也就是现在的 ResourceManager）的资源消耗，并且让监测每一个 Job 子任务 (tasks) 状态的程序分布式化了，更安全、更优美。
- 老的框架中，JobTracker一个很大的负担就是监控job下的tasks的运行状况，现在，这个部分就扔给ApplicationMaster做了而ResourceManager中有一个模块叫做 ApplicationsManager(ASM)，它负责监测ApplicationMaster的运行状况。
- 能够支持不同的计算框架

## map reduce优缺点

优点

MapReduce 易于编程 : 它简单的实现一些接口，就可以完成一个分布式程序，这个分布式程序可以分布到大量廉价的PC机器上运行。也就是说你写一个分布式程序，跟写一个简单的串行程序是一模一样的。就是因为这个特点使得MapReduce编程变得非常流行。

良好的扩展性 : 当你的计算资源不能得到满足的时候，你可以通过简单的增加机器来扩展它的计算能力。

高容错性 : MapReduce设计的初衷就是使程序能够部署在廉价的PC机器上，这就要求它具有很高的容错性。比如其中一台机器挂了，它可以把上面的计算任务转移到另外一个节点上运行，不至于这个任务运行失败，而且这个过程不需要人工参与，而完全是由 Hadoop内部完成的。

适合PB级以上海量数据的离线处理 : 这里加红字体离线处理，说明它适合离线处理而不适合在线处

理。

### 缺点

实时计算：MapReduce无法像MySQL一样，在毫秒或者秒级内返回结果。

流式计算：流式计算的输入数据是动态的，而MapReduce的输入数据集是静态的，不能动态变化。这是因为MapReduce自身的设计特点决定了数据源必须是静态的。

DAG（有向图）计算：多个应用程序存在依赖关系，后一个应用程序的输入为前一个的输出。在这种情况下，MapReduce并不是不能做，而是使用后，每个MapReduce作业的输出结果都会写入到磁盘，会造成大量的磁盘IO，导致性能非常的低下。

### hadoop优缺点

#### 1、高可靠性

Hadoop按位存储和处理数据的能力值得人们信赖。

#### 2、高扩展性

Hadoop是在可用的计算机集簇间分配数据并完成计算任务的，这些集簇可以方便地扩展到数以千计的节点中。

#### 3、高效性

Hadoop能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此处理速度非常快。

#### 4、高容错性。

Hadoop能够自动保存数据的多个副本，并且能够自动将失败的任务重新分配。Hadoop带有用Java语言编写的框架，因此运行在Linux生产平台上是非常理想的。Hadoop上的应用程序也可以使用其他语言编写，比如C++。

#### 缺点：

Hadoop作为一个处理大数据的软件框架，虽然受到众多商业公司的青睐，但是其自身的技术特点也决定了它不能完全解决大数据问题。在当前Hadoop的设计中，所有的metadata操作都要通过集中式的NameNode来进行，NameNode有可能是性能的瓶颈。当前Hadoop单一NameNode、单一Jobtracker的设计严重制约了整个Hadoop可扩展性和可靠性。

首先，NameNode和JobTracker是整个系统中明显的单点故障源。再次，单一NameNode的内存容量有限，使得Hadoop集群的节点数量被限制到2000个左右，能支持的文件系统大小被限制在10-50PB，最多能支持的文件数量大约为1.5亿左右。实际上，有用户抱怨其集群的NameNode重启需要数小时，这大大降低了系统的可用性。

### 补充问答：

#### 1、运行Hadoop集群需要哪些守护进程？

DataNode，NameNode，TaskTracker和JobTracker都是运行Hadoop集群需要的守护进程。

#### 2、Hadoop常见输入格式是什么？

三种广泛使用的输入格式是：

文本输入：Hadoop中的默认输入格式。

Key值：用于纯文本文件

序列：用于依次读取文件

3、RDBMS和Hadoop的主要区别是什么？

RDBMS用于事务性系统存储和处理数据，而Hadoop可以用来存储大量数据。

4、假如Namenode中没有数据会怎么样？

没有数据的Namenode就下能称之为Namenode，通常情况下，Namenode肯定全有数据。

5、当Job Tracker宕掉时，Namenode会发生什么？

当Job Tracker失败时，集群仍然可以正常工作，只要Namenode没问题。

## lect26: Spark

能够针对高性能计算需求，设计并实现基于Spark和Spark SQL的内存并行计算方案

### 1. 什么是spark？

Apache Spark 是专为大规模数据处理而设计的快速通用的计算引擎。Spark是UC Berkeley AMP lab (加州大学伯克利分校的AMP实验室)所开源的类Hadoop MapReduce的通用并行框架，Spark，拥有Hadoop MapReduce所具有的优点；但不同于MapReduce的是——Job中间输出结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的MapReduce的算法。

Spark 是一种与 Hadoop 相似的开源集群计算环境，但是两者之间还存在一些不同之处，这些有用的不同之处使 Spark 在某些工作负载方面表现得更加优越，换句话说，Spark 启用了内存分布数据集，除了能够提供交互式查询外，它还可以优化迭代工作负载。

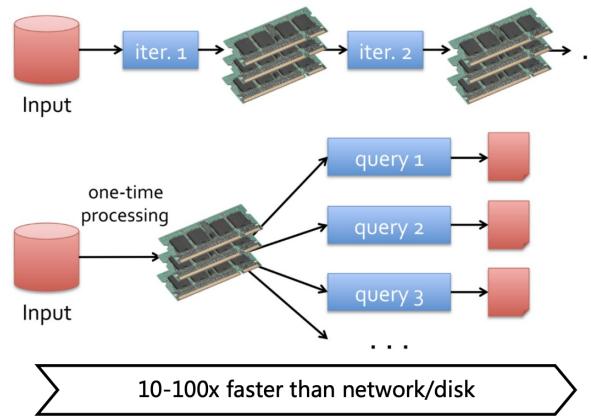
Spark 是在 Scala 语言中实现的，它将 Scala 用作其应用程序框架。与 Hadoop 不同，Spark 和 Scala 能够紧密集成，其中的 Scala 可以像操作本地集合对象一样轻松地操作分布式数据集。

### 2. spark vs mapreduce

hadoop框架在之前侧重于离线大批量计算，而spark则侧重内存以及实时计算。

job中间输出结果可以保存在内存中，从而不再需要读写HDFS，计算的中间结果可以高效地转给下一个计算步骤，提高算法性能。它的算法性能相比MapReduce 提高了10~100 倍。

但是大数据模式，还是需要大量写磁盘，所以只有很大内存的机器才能看出来spark的优势。



## 一些细节

### 1, 交换数据的方式

MR 多使用hdfs做数据交换, 多节点会带来IO压力; Spark多是基于本地磁盘做数据交换。

### 2 , 执行单元

\$M R\$ 的task的执行单元是进程, 进程的创建销毁的开销较大; Spark的task执行单元是线程, 开销较小。

### 3 , 缓存

MR 基本上没有使用缓存, 读取效率低; Spark是用Buffle KV做缓存, 很适合算法场景的多次迭代计算, 也可以了解一下"鸽丝计划"及向量化查询。

### 4, shuffle

MR的shuffle 过程每一个task 会产生多个小文件, task之间的文件无法共用; Spark 的shuffle过程会合并小文件, 保持task和file是一对一, 来精简小文件的数量。

### 5, 数据

MR的数据多是hdfs, 一对Map和Reduce 与其他MR是独立的, 上下游执行阶段是没有血缘记录的; Spark的数据是RDD, 多MR形式来共享数据, 构建了一个DAG (有向无环图), 上下游执行阶段是有血缘记录的, 减少了重复拉取数据的成本。

### 6, 资源申请粒度

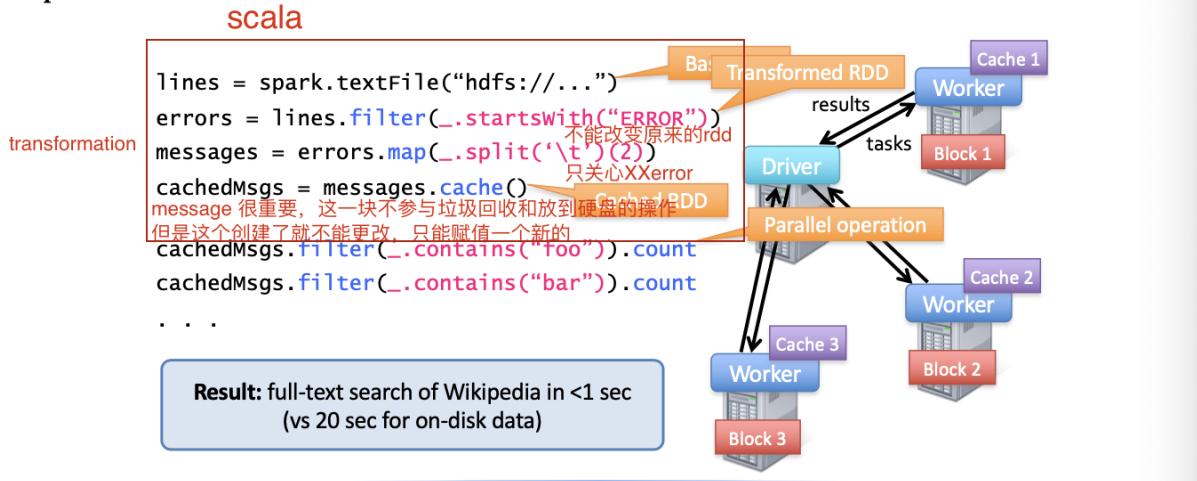
MapReduce是细粒度每一个task去独自做资源申请, Spark是粗粒度是一个整体job来资源申请。

## Example: Log Mining

lines加载进入后进入弹性分布式数据集 (rdd,大数组)

操作分两类 : transformation(输入输出都是数据集rdd, 类似map)

- Load error messages from a log into memory, then interactively search for various patterns



filter和map没有做，只有这个.cache()才需要做涉及的内容，前面的两条transform才会做，就像lazy fetch

如果只是filter就是transform，但是如果调用了count, 就需要调用所有机器的数据，这个是一个action，就是类似reduce的操作。这个没法lazy, 需要立即执行，是并行执行的。

cache 是程序员告诉的spark的一个希望，但是如果要求不合理（cache放不下了）就不满足这个，比如可能压缩一下，或者放到硬盘里面。

map reduce会放到硬盘上，spark就放到内存里面。但是思想是基于mapreduce的，

### spark 基本流程示例

- 从外部数据创建一些作为输入的RDD
- 使用类似filter之类的变换(Transformations)来定义出新的RDD
- 要求Spark对需要重用的任何中间RDD进行persist
- 启用类似count之类的作用(Actions)进行并行计算

### 3. 宽依赖vs窄依赖

窄依赖是指 1个父RDD分区对应1个子RDD的分区。

宽依赖是指 1个父RDD分区对应多个子RDD分区。

## Narrow Dependencies VS. Wide Dependencies

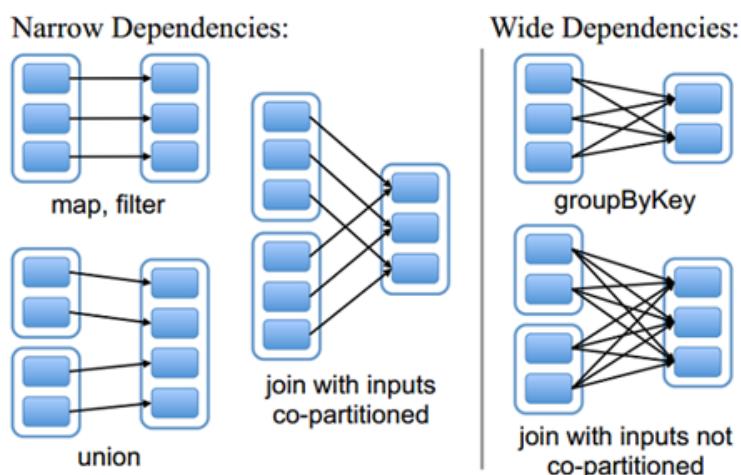
- 宽依赖往往意味着Shuffle操作，可能涉及多个节点的数据传输
- 当RDD分区丢失时，Spark会对数据进行重算
- 窄依赖只需计算丢失RDD的父分区，不同节点间可以并行计算，能更有效地进行节点的恢复
- 宽依赖中，重算的子RDD分区往往来自多个父RDD分区，其中只有一部分数据用于恢复，造成了不必要的冗余，甚至需要整体重新计算

### 2.1 窄依赖(narrow dependency)

可以支持在同一个集群Executor上，以pipeline管道形式顺序执行多条命令，例如在执行了map后，紧接着执行filter。分区内的计算收敛，不需要依赖所有分区的数据，可以并行地在不同节点进行计算。所以它的失败恢复也更有效，因为它只需要重新计算丢失的parent partition即可

### 2.2 宽依赖(shuffle dependency)

则需要所有的父分区都是可用的，必须等RDD的parent partition数据全部ready之后才能开始计算，可能还需要调用类似MapReduce之类的操作进行跨节点传递。从失败恢复的角度看，shuffle dependency牵涉RDD各级的多个parent partition。



需要特别说明的是对join操作有两种情况：

- (1) 图中左半部分join：如果两个RDD在进行join操作时，一个RDD的partition仅仅和另一个RDD中已知个数的Partition进行join，那么这种类型的join操作就是**窄依赖**，例如图1中左半部分的join操作(join with inputs co-partitioned)；
- (2) 图中右半部分join：其它情况的join操作就是宽依赖，例如图1中右半部分的join操作(join with inputs not co-partitioned)，由于是需要父RDD的所有partition进行join的转换，这就涉及到了shuffle，因此这种类型的join操作也是**宽依赖**。

### 3. DAG

RDD之间的依赖关系就形成了DAG（有向无环图）

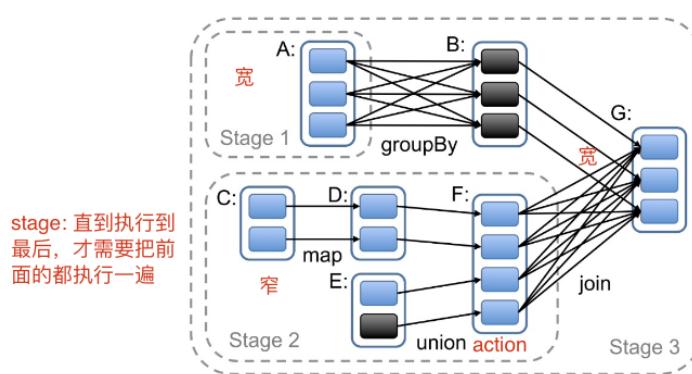
在Spark作业调度系统中，调度的前提是判断多个作业任务的依赖关系，这些作业任务之间可能存在因果的依赖关系，也就是说有些任务必须先获得执行，然后相关的依赖人物才能执行，但是任务之间显然不应出现任何直接或间接的循环依赖关系，所以本质上这种关系适合用DAG表示

### 4. stage划分

由于shuffle依赖必须等RDD的父RDD分区数据全部可读之后才能开始计算，因此Spark的设计是让父RDD将结果写在本地，完全写完之后，通知后面的RDD。后面的RDD则首先去读之前RDD的本地数据作为输入，然后进行运算。

由于上述特性，讲shuffle依赖就必须分为两个阶段(stage)去做：

- (1) 第1个阶段(stage)需要把结果shuffle到本地，例如reduceByKey，首先要聚合某个key的所有记录，才能进行下一步的reduce计算，这个汇聚的过程就是shuffle。
  - (2) 第二个阶段(stage)则读入数据进行处理。
- Stage
  - 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)
  - 边界有两种情况：一是宽依赖上的Shuffle操作；二是已缓存分区

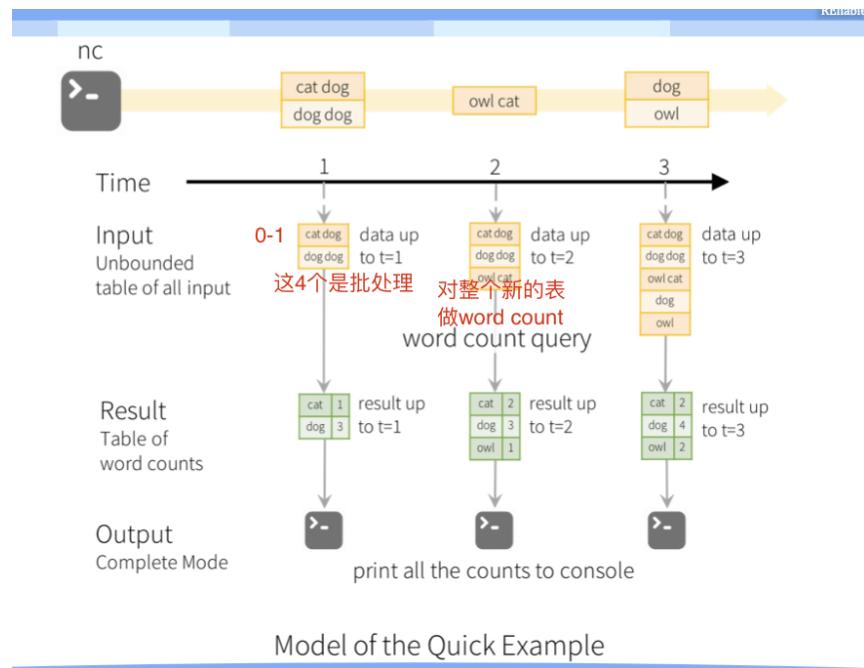


stage执行过的结果是可以cache住的。若G崩了，可以回退F（被cache）如果丢了，就找那个有向无环图，再跑一次

#### Removing Data —— LRU策略

Structured Streaming

word count: 批处理。



就是切成很多时间片，微观看每个时间片做批处理。



- 什么是RDD？

RDD叫做**弹性分布式数据集**，是**Spark中最基本的数据抽象**，它代表一个不可变、可分区、里面的元素可并行计算的集合。RDD具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

- RDD的属性

(1) 一组分片 (Partition)，即数据集的基本组成单位。对于RDD来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建RDD时指定RDD的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分配到的CPU Core的数目。

(2) 一个计算每个分区的函数。Spark中RDD的计算是以分片为单位的，每个RDD都会实现compute函数以达到这个目的。compute函数会对迭代器进行复合，不需要保存每次计算的结果。

(3) RDD之间的依赖关系。RDD的每次转换都会生成一个新的RDD，所以RDD之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark可以通过这个依赖关系重新计算丢失的分区数据，而不是对RDD的所有分区进行重新计算。

(4) 一个Partitioner，即RDD的分片函数。当前Spark中实现了两种类型的分片函数，一个是基于哈希的HashPartitioner，另外一个是基于范围的RangePartitioner。只有对于key-value的RDD，才会有Partitioner，非key-value的RDD的Partitioner的值是None。Partitioner函数不但决定了RDD本身的分片数量，也决定了parent RDD Shuffle输出时的分片数量。

(5) 一个列表，存储取每个Partition的优先位置（preferred location）。对于一个HDFS文件来说，这个列表保存的就是每个Partition所在的块的位置。按照“移动数据不如移动计算”的理念，Spark在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

## lect 27: storm

能够针对高性能计算需求，设计并实现基于Storm的流数据处理方案

### storm 简介

Apache Storm是一个免费开源、分布式、高容错的实时计算系统。Storm令持续不断的流计算变得容易，弥补了Hadoop批处理所不能满足的实时要求。Storm经常用于在实时分析、在线机器学习、持续计算、分布式远程调用和ETL等领域。Storm主要分为两种组件Nimbus和Supervisor。这两种组件都是快速失败的，没有状态。任务状态和心跳信息等都保存在Zookeeper上的，提交的代码资源都在本地机器的硬盘上。

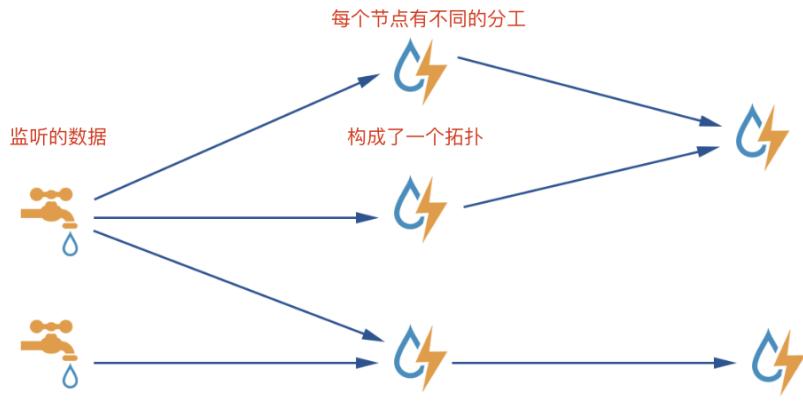
Nimbus 负责在集群里面发送代码，分配工作给机器，并且监控状态。全局只有一个。

Supervisor 会监听分配给它那台机器的工作，根据需要启动/关闭工作进程Worker。每一个要运行Storm的机器上都要部署一个，并且，按照机器的配置设定上面分配的槽位数。

Zookeeper是Storm重点依赖的外部资源。Nimbus和Supervisor甚至实际运行的Worker都是把心跳保存在Zookeeper上的。Nimbus也是根据Zookeeper上的心跳和任务运行状况，进行调度和任务分配的。

Topology 处理的最小的消息单位是一个Tuple，也就是一个任意对象的数组。Storm提交运行的程序称为Topology。

Topology 由Spout和Bolt构成。Spout是发出Tuple的结点。Bolt可以随意订阅某个Spout或者Bolt发出的Tuple。Spout和Bolt都统称为component。

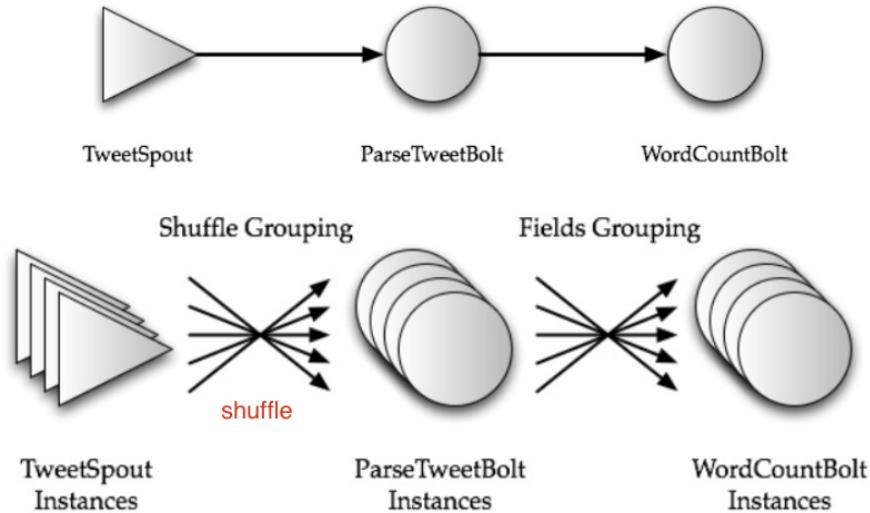


## Why use Apache Storm ?

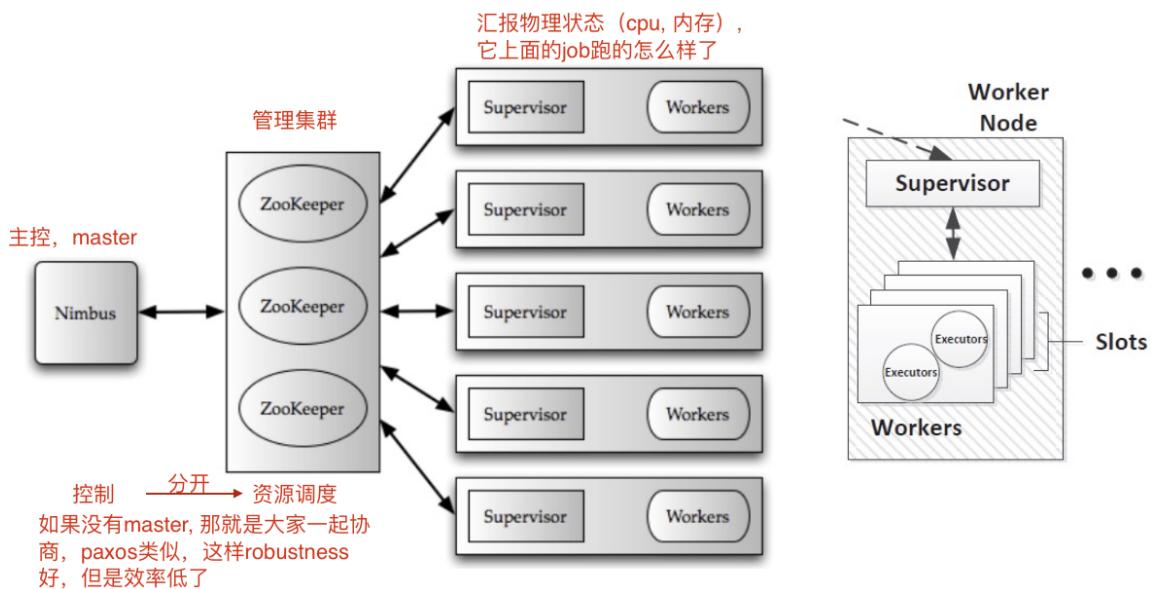


- Apache Storm is a free and open source distributed realtime computation system.
  - Apache Storm makes it easy to reliably process **unbounded streams of data**, doing for realtime processing what Hadoop did for batch processing.
  - Apache Storm is simple, can be used with **any programming language**, and is a lot of fun to use!
- Apache Storm has many use cases:
  - realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Apache Storm is fast: a benchmark clocked it at over **a million tuples processed per second per node**. It is **scalable, fault-tolerant**, guarantees your data will be processed, and is easy to set up and operate. zookeeper 集群管理
- Apache Storm integrates with the **queueing** and **database** technologies you already use.
  - An Apache Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed. 数据来源：队列，图数据库...，因此可以和他们做集成  
Read more in the tutorial

## Word Count

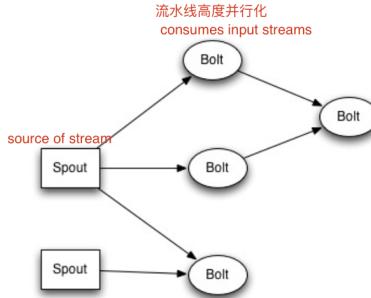


## Apache Storm Architecture



## Apache storm

- A **topology** is a graph of computation.
- A **stream** is an unbounded sequence of tuples.
- A **bolt** consumes input streams
- A **spout** is a source of streams.



## zookeeper

### What is ZooKeeper?

- ZooKeeper is a high-performance coordination service for **distributed applications**.
- It exposes common services - such as naming, configuration management, synchronization, and group services - in a **simple interface** so you don't have to write them from scratch.
- You can use it off-the-shelf to implement consensus, group management, leader election, and presence protocols. And you can build on it for your own, specific needs.

## Guarantees

**REIN**  
REliable, INtelligent & Scalable Systems

- ZooKeeper is very fast and very simple. It provides a set of guarantees. These are:
  - **Sequential Consistency** - Updates from a client will be applied in the order that they were sent.
  - **Atomicity** - Updates either succeed or fail. No partial results.
  - **Single System Image** - A client will see the same view of the service regardless of the server that it connects to. i.e., a client will never see an older view of the system even if the client fails over to a different server with the same session.
  - **Reliability** - Once an update has been applied, it will persist from that time forward until a client overwrites the update.
  - **Timeliness** - The clients view of the system is guaranteed to be up-to-date within a certain time bound.

## Storm Cluster的组成部分

Supervisor向Zookeeper报告自己状态和任务进度，Zookeeper告诉Nimbus可以调用什么机器，Zookeeper负责**资源管理**，Nimbus负责**任务调度**

总结：一个分布式系统最好有master，并将资源管理和任务调度分开来管！

Hadoop/Spark/Storm其实在结构上都是类似的思想，但是侧重的方面不一样

没有master的好处是任何一台机器挂了都没有关系，但是难以维护统一的结果

The basic primitives Storm provides for doing stream transformations are “spouts” and “bolts”.  
Bolt计算，并行处理

## Storm与Spark、Hadoop三种框架对比

1. Storm是最佳的流式计算框架，Storm由Java和Clojure写成，Storm的优点是全内存计算，所以它的定位是分布式实时计算系统，按照Storm作者的说法，Storm对于实时计算的意义类似于Hadoop对于批处理的意义。

Storm的适用场景：

- 1) 流数据处理

Storm可以用来处理源源不断地流进来的消息，处理之后将结果写入到某个存储中去。

- 2) 分布式RPC。由于Storm的处理组件是分布式的，而且处理延迟极低，所以可以作为一个通用的分布式RPC框架来使用。

2. Spark是一个基于内存计算的开源集群计算系统，目的是更快速地进行数据分析。Spark使用Scala开发，类似于Hadoop MapReduce的通用并行计算框架，Spark基于Map Reduce算法实现的分布式计算，拥有Hadoop MapReduce所具有的优点，但不同于MapReduce的是Job中间输出和结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的Map Reduce的算法。

Spark的适用场景：

- 1) 多次操作特定数据集的应用场合

Spark是基于内存的迭代计算框架，适用于需要多次操作特定数据集的应用场合。需要反复操作的次数越多，所需读取的数据量越大，受益越大，数据量小但是计算密集度较大的场合，受益就相对较小。

- 2) 粗粒度更新状态的应用

由于RDD的特性，Spark不适用那种异步细粒度更新状态的应用，例如Web服务的存储或者是增量的Web爬虫和索引。就是对于那种增量修改的应用模型不适合。

总的来说Spark的适用面比较广泛且比较通用。

3. Hadoop是实现了MapReduce的思想，将数据切片计算来处理大量的离线数据。Hadoop处理的数据必须是已经存放在HDFS上或者类似HBase的数据库中，所以Hadoop实现的时候是通过移动计算到这些存放数据的机器上来提高效率。

Hadoop的适用场景：

- 1) 海量数据的离线分析处理
- 2) 大规模Web信息搜索
- 3) 数据密集型并行计算

顺提一下ch30: **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，并提供完整的sql查询功能，可以将sql语句转换为MapReduce任务进行运行，这套SQL简称HQL。

简单来说：

Hadoop适合于离线的批量数据处理适用于对实时性要求极低的场景

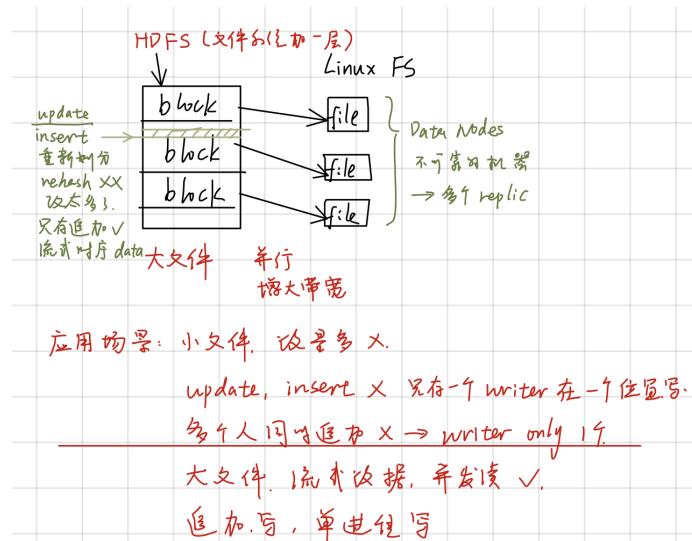
Storm适合于实时流数据处理，实时性方面做得极好

Spark是内存分布式计算框架，试图吞并Hadoop的Map-Reduce批处理框架和Storm的流处理框架，但是Spark已经做得很不错了，批处理方面性能优于Map-Reduce，但是流处理目前还是弱于Storm，产品仍在改进之中

## lect28: HDFS

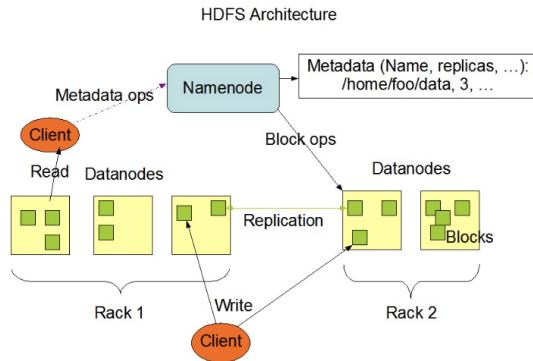
– 能够根据大尺寸数据文件的存储需求，设计使用HDFS存储的方案，设计并实现对HDFS中存储文件的读写操作

针对大文件的存取。



namenode 告诉client，它需要的文件块在哪个机器上，如果有多个replica, 会有选取的策略，就近选取之类。然后告诉了之后就client直接和这个机器打交道了。类似nginx的负载均衡。master: namenode(存储元数据) slave : datanode,

- HDFS has a master/slave architecture.



- A typical deployment has **a dedicated machine that runs only the NameNode software**. Each of the other machines in the cluster runs **one instance of the DataNode software**.

namenode 和datanode的机器分开。namenode 是专门指派的跑，datanode 一般也是一台机器一个。

datanode不知道大文件的存在。不知道全貌。

namenode 类似inode, 要记住某一块数据叫什么。

## 文件管理（文件夹）

Namenode存储的文件名和datanode的文件名不同。eg.HDFS : /users/1.txt, datanode: /hdfs/1\_1.txt

如果datanode全部都存在一个文件夹下，导致索引查找太慢，datanode就会按照一定的逻辑建立树形的结构，就是创建一些目录.namenode也可以自己创建目录。不支持hardlink 和softlink.

eg,20210101/1.txt 注意这个怎么创建完全是datanode决定的，这个和hdfs里面的文件名没有任何关系。

HDFS supports a traditional hierarchical file organization.

- A user or an application can create **directories** and store files inside these directories.
- The file system namespace hierarchy is similar to most other existing file systems; one can create and remove files, move a file from one directory to another, or rename a file.
- HDFS supports **user quotas** and **access permissions**.
- HDFS does **not** support hard links or soft links. However, the HDFS architecture does **not** preclude implementing these features.

The existence of a single NameNode in a cluster greatly simplifies the architecture of the system.

- The NameNode is the **arbitrator** and **repository** for all HDFS metadata.
- The system is designed in such a way that user data **never flows through the NameNode**.

一旦namenode指派了client和datanode的连线，就再也不会经过namenode了。

## file system namespace

namenode 可以指定某个文件的replica 的个数，重要的就多一些。副本不会在同一台机器上。如果有两个机架， 要两个机架都有

The **NameNode** maintains the file system namespace.

- Any change to the file system namespace or its properties is recorded by the NameNode.
- An application can specify **the number of replicas** of a file that should be maintained by HDFS.  
每个这个文件的block都会有这么多的replica.
- The number of copies of a file is called the **replication factor** of that file. This information is stored by the NameNode.

## Data replication

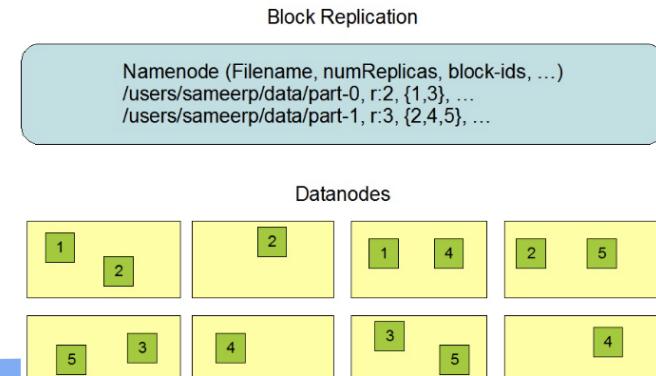
一个文件block会被复制到很多block。

- **The NameNode**

- makes all decisions regarding replication of blocks.
- It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster.
- Receipt of a Heartbeat implies that the DataNode is functioning properly.
- A Blockreport contains a list of all blocks on a DataNode.

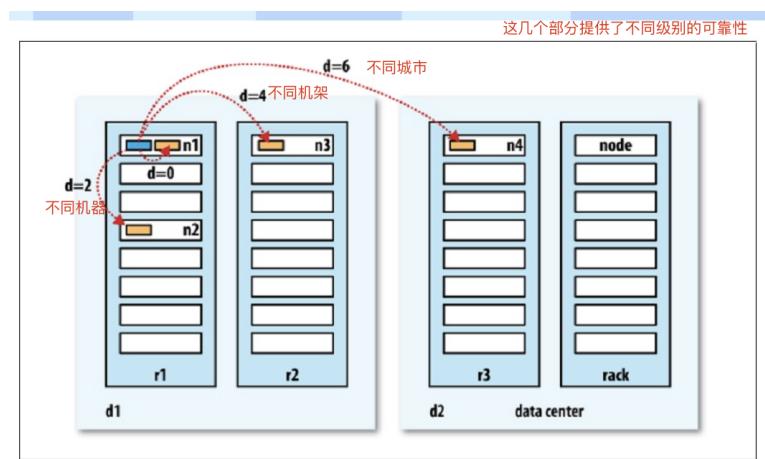
datanode给namenode发心跳和blcokreport(我现在有什么文件)。一旦死了，1.转发请求就不考虑这个节点了2.复制这个节点的所有数据到别的replica。heartbeat 和blockreport 是分开的，心跳频率大， blockreport 可能很大，占的带宽很大，可以频率小一点。

心跳不能这么严格，一旦收不到就认为死了，用守护进程（一个死循环，在某个地方触发心跳）发心跳，这个死循环是这个程序里面的eg.opengl里面的渲染画面循环，而不是定时器（和这个程序分开，eg, 这个机器的读写坏了，但是计时器还是好的，这样就不行了。）发心跳，循环的时间就不能保证了。所以不是一次没有收到就不行，而是一段时间（设定）没有收到就不行了。



datanode写副本是pipeline, 不是namenode 告诉这三个一起写, 而是一个datanode写完了, 这个datanode 写完了告诉下一个datanode写数据。

- **Replication Pipelining**
- When a client is writing data to an HDFS file with a **replication factor of three**,
  - the NameNode retrieves a list of DataNodes using a **replication target choosing algorithm**.
  - This list contains the DataNodes that will host a replica of that block.
  - The client then writes to the **first** DataNode.
  - The **first** DataNode starts receiving the data in portions, writes each portion to its local repository and transfers that portion to the **second** DataNode in the list. The **second** DataNode, in turn starts receiving each portion of the data block, writes that portion to its repository and then flushes that portion to the **third** DataNode. Finally, the **third** DataNode writes the data to its local repository.
  - Thus, a DataNode can be receiving data from the **previous one** in the pipeline and at the same time forwarding data to the **next one** in the pipeline.
  - Thus, the data is **pipelined from one DataNode to the next**.



但是写replica由于是流水线, 所以有很大的网络开销。因此也不能太多。一般情况3.

If the replication factor is greater than 3,

- the placement of the 4th and following replicas are determined randomly while keeping the number of replicas per rack below the upper limit (which is basically  $(\text{replicas} - 1) / \text{racks} + 2$ ).  
每一个机架上面数据有上限，可以使得每个数据比较平均的存在某个服务器上。

replica: 1. 读的负载均衡 2.容错防止断电。

## The Persistence of File System Metadata

### Safemode

启动过程先看一下是不是每个文件都有足够的副本了，有了才开始提供服务，这个启动过程就是叫做处于安全模式。

On startup, the **NameNode** enters a special state called **Safemode**.

- Replication of data blocks does **not** occur when the NameNode is in the Safemode state.
- The NameNode receives **Heartbeat** and **Blockreport** messages from the DataNodes.
- A Blockreport contains the list of data blocks that a DataNode is hosting.
- Each block has a specified **minimum number of replicas**.
- A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode.
- After **a configurable percentage of safely replicated data blocks** checks in with the NameNode (plus an additional 30 seconds), the NameNode exits the Safemode state.
- It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas.
- The NameNode then replicates these blocks to other DataNodes.

写的时候要写log, 类似write ahead log, 要写EditLog. 本身不能存在HDFS, 因为目的就是恢复HDFS。这个要写本地的文件系统。

- The NameNode uses a file in its **local host OS file system** to store the EditLog.
- The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the **FsImage**.
- The FsImage is stored as a file in the NameNode's **local file system** too.

先写editlog, 再写FsImage. 一旦之前写FsImage成功了，就会删除editlog。正常关机，清空editlog, 但是要是断电了，FsImage没有写，就再执行以下editlog. 这个就是一个transaction。

- The NameNode keeps an image of the entire file system namespace and file Blockmap in memory.
  - When the NameNode starts up, or a checkpoint is triggered by a configurable threshold, it reads the FsImage and EditLog from disk, applies all the transactions from the EditLog to the in-memory representation of the FsImage, and flushes out this new version into a new FsImage on disk.
  - It can then truncate the old EditLog because its transactions have been applied to the persistent FsImage.
  - This process is called a checkpoint.
  - The purpose of a checkpoint is to make sure that HDFS has a consistent view of the file system metadata by taking a snapshot of the file system metadata and saving it to FsImage.
  - A checkpoint can be triggered at a given time interval (dfs.namenode.checkpoint.period) expressed in seconds, or after a given number of filesystem transactions have accumulated (dfs.namenode.checkpoint.txns).
  - If both of these properties are set, the first threshold to be reached triggers a checkpoint.

## snapshot

Fsimage 是使用快照，就是和上一个不一样的东西，增量备份的感觉。

Blockreport：大block里面的第几个小block在我这台机器里面的文件位置

过程调用的是TPC/ IP 协议。 使用RPC通信。

## re-replication

### Data Disk Failure, Heartbeats and Re-Replication

- Each DataNode sends a Heartbeat message to the NameNode periodically.
- A network partition can cause a subset of DataNodes to lose connectivity with the NameNode.
- The NameNode detects this condition by the absence of a Heartbeat message.
- The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them. Any data that was registered to a dead DataNode is not available to HDFS any more.

### Data Integrity

数据完整性

就是用checksum 来验证文件的完整性。一旦验证发现checksum 坏了，就从别的备份来。

## Data Integrity

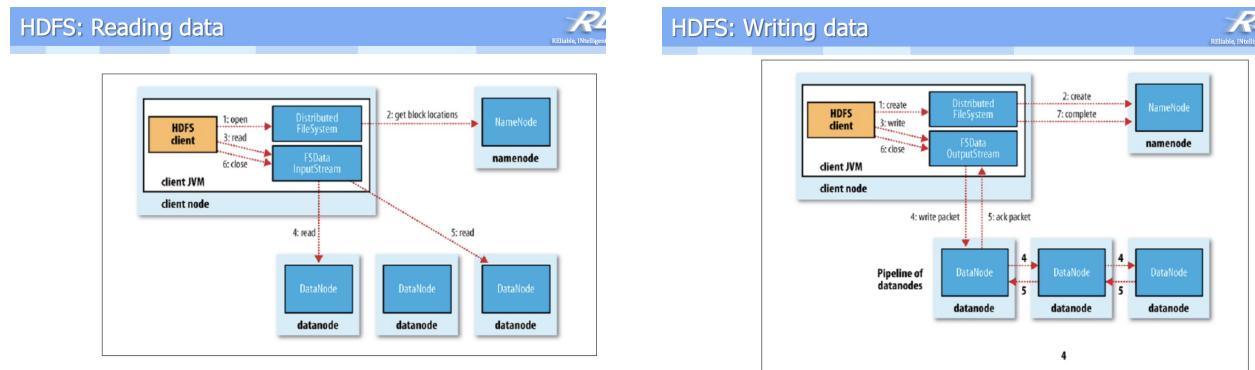
It is possible that a block of data fetched from a DataNode arrives **corrupted**.

- This corruption can occur because of faults in a **storage device, network faults, or buggy software**.
- The HDFS client software implements **checksum checking** on the contents of HDFS files.
- When a client creates an HDFS file, it computes a **checksum of each block of the file** and **stores these checksums in a separate hidden file** in the same HDFS namespace.
- When a client retrieves file contents it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file.
- If **not**, then the client can opt to retrieve that block from **another DataNode** that has a replica of that block.

FsImage 和 Editlog也要有多个副本，有一个改了就集群要一起改。这些就是存在机器的文件系统，而不是HDFS，因为这个是来恢复HDFS的。

### • Metadata Disk Failure

- The **FsImage** and the **EditLog** are central data structures of HDFS.
  - A corruption of these files can cause the HDFS instance to be non-functional.
  - For this reason, the NameNode can be configured to support **maintaining multiple copies of the FsImage and EditLog**.
  - Any update to either the FsImage or EditLog causes each of the FsImages and EditLogs to get updated **synchronously**.
  - When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use.
- **Snapshots**
  - **Snapshots** support storing a copy of data at a particular instant of time. One usage of the snapshot feature may be to roll back a corrupted HDFS instance to a previously known good point in time.



读的负载均衡

如果是大家一起写，要所有人写完第一个，才能写第二个，太慢了，pipeline 可以dup2写block1的时候dup1可以写block2了。

- **FS Shell**
- HDFS allows user data to be organized in the form of files and directories.
  - It provides a **commandline** interface called **FS shell** that lets a user interact with the data in HDFS.
  - The syntax of this command set is similar to other shells (e.g. bash, csh) that users are already familiar with. Here are some sample action/command pairs:

Action	Command
Create a directory named /foodir	bin/hadoop dfs -mkdir /foodir
Remove a directory named /foodir	bin/hadoop fs -rm -R /foodir
View the contents of a file named /foodir/myfile.txt	bin/hadoop dfs -cat /foodir myfile.txt

- **DFSAdmin**
- The **DFSAdmin** command set is used for administering an HDFS cluster.
  - These are commands that are used **only by an HDFS administrator**.
  - Here are some sample action/command pairs:

Action	Command
Put the cluster in Safemode	bin/hdfs dfsadmin -safemode enter
Generate a list of DataNodes	bin/hdfs dfsadmin -report
Recommission or decommission DataNode(s)	bin/hdfs dfsadmin -refreshNodes

给管理员用的，

## 删除文件

有一个垃圾箱，如果没有清空，有一个时间间隔，如果过期了就倾倒垃圾箱，只是把文件的目录改了一下，就是一个改名的过程。

skipTrash 直接删除，不是放到垃圾箱。

## 删除replica

降低某个文件的replica个数。

namenode决定要删除哪个。在某个heartbeat的应答说把这个replica删除掉。

web 应用的端口和提供应用的端口不一样。

# lec29: HBase

HBase is a **distributed column-oriented** database built on top of HDFS.

按照列存储编码机制减少数据 (60 +1, +1, +1, -1, +2) , 压缩机制减少数据+1\*4

相关内容就近存储。

比较适合做分布式的存储, region不断追加, 不同的region存在不同的服务器上, 所以可扩展性比较好, 元数据的管理也比较方便。元数据, 有多少表, 每个表被切成了多少块。元数据的元数据(索引)存在root。分布式的存储性能和一台上的存储性能差不多。

因为是非结构化存储, 所以不支持sql. schema不严格。比如有的行有的列是空的。

- Apache HBase™ is the Hadoop database, a **distributed, scalable, big data store**.
  - Use Apache HBase™ when you need **random, realtime read/write access** to your Big Data.
  - This project's goal is the hosting of very large tables -- **billions of rows X millions of columns** -- atop clusters of commodity hardware.
  - Apache HBase is an **open-source, distributed, versioned, non-relational database** modeled after Google's Bigtable: A Distributed Storage System for Structured Data by Chang et al.
  - Just as Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides **Bigtable-like capabilities** on top of **Hadoop** and **HDFS**.
- <https://hbase.apache.org/>

## Features

- Linear and modular scalability. region太多了, eg. 100个就全部打包压缩放到硬盘。
- Strictly consistent reads and writes.
- Automatic and configurable sharding of tables
- Automatic failover support between RegionServers. 数据有冗余可以容错
- Convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables.
- Easy to use Java API for client access.
- Block cache and Bloom Filters for real-time queries. 查询效率提升
- Query predicate push down via server side Filters
- Thrift gateway and a REST-ful Web service that supports XML, Protobuf, and binary data encoding options
- Extensible jruby-based (JIRB) shell
- Support for exporting metrics via the Hadoop metrics subsystem to files or Ganglia; or via JMX 微内核

## Applications store data into labeled tables.

- Tables are made of rows and columns. 多版本存储，没有限制个数
- Table cells—the intersection of row and column coordinates—are versioned.
  - By default, their version is a timestamp auto-assigned by HBase at the time of cell insertion.
  - A cell's content is an uninterpreted array of bytes.
- Table row keys are also byte arrays, 不能太长否则索引占太多空间
  - so theoretically anything can serve as a row key from strings to binary representations of long or even serialized data structures.
- Table rows are sorted by row key, the table's primary key.
  - The sort is byte-ordered.
  - All table accesses are via the table primary key.

不同于关系型数据库

▼ 列族的数量尽量不要超过3个！

Create a table.

- Use the `create` command to create a new table. You must specify the **table name** and the **ColumnFamily name**.  
`hbase(main):001:0> create 'test', 'cf'`  
0 row(s) in 0.4170 seconds  
=> Hbase::Table - test

List Information About your Table

- Use the `list` command to confirm your table exists  
`hbase(main):002:0> list 'test'`  
TABLE  
test  
1 row(s) in 0.0180 seconds  
=> ["test"]

Put data into your table.

- To put data into your table, use the `put` command.  

```
hbase(main):003:0> put 'test', 'row1', 'cf:a', 'value1'
0 row(s) in 0.0850 seconds
hbase(main):004:0> put 'test', 'row2', 'cf:b', 'value2'
0 row(s) in 0.0110 seconds
hbase(main):005:0> put 'test', 'row3', 'cf:c', 'value3'
0 row(s) in 0.0100 seconds
```
- Here, we insert three values, one at a time.
- The first insert is at `row1`, column `cf:a`, with a value of `value1`.
- Columns in HBase are comprised of a column family prefix, `cf` in this example, followed by a colon and then a column qualifier suffix, `a` in this case.

常用操作：

Get

- [Get](#) returns attributes for a specified row. Gets are executed via [Table.get](#)

Put

- [Put](#) either adds new rows to a table (if the key is new) or can update existing rows (if the key already exists). Puts are executed via [Table.put](#) (non-writeBuffer) or [Table.batch](#) (non-writeBuffer)

Scans

- [Scan](#) allow iteration over multiple rows for specified attributes.

Delete

- [Delete](#) removes a row from a table. Deletes are executed via [Table.delete](#).

## 数据存储

- Cells in this table that appear to be **empty do not take space**, or in fact exist, in HBase.
- This is what makes HBase "**sparse**."

Row Key	只存一行	Time Stamp	ColumnFamily contents	ColumnFamily anchor	ColumnFamily people
"com.cnn.www"		t9		anchor:cnnsi.com = "CNN"	
"com.cnn.www"		t8		anchor:my.look.ca = "CNN.com"	
"com.cnn.www"		t6	contents:html = "<html>..."		
"com.cnn.www"		t5	contents:html = "<html>..."		
"com.cnn.www"		t3	contents:html = "<html>..."		
"com.example.www"		t5	contents:html = "<html>..."		people:author = "John Doe"

把域名倒过来排序

- *ColumnFamily anchor*

Row Key	Time Stamp	Column Family anchor
"com.cnn.www"	t9	anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8	anchor:my.look.ca = "CNN.com"

- *ColumnFamily contents*

Row Key	Time Stamp	Column Family contents
"com.cnn.www"	t6	contents:html = "<html>..."
"com.cnn.www"	t5	contents:html = "<html>..."
"com.cnn.www"	t3	contents:html = "<html>..."

## Namespace

限制容量、安全权限、分组，引用的话在表之前再加一个：



Namespace

- A namespace is a logical grouping of tables analogous to a **database** in relation database systems. This abstraction lays the groundwork for upcoming multi-tenancy related features:
  - Quota Management ([HBASE-8410](#)) - Restrict the amount of resources (i.e. regions, tables) a namespace can consume.
  - Namespace Security Administration ([HBASE-9206](#)) - Provide another level of security administration for tenants.
  - Region server groups ([HBASE-6721](#)) - A namespace/table can be pinned onto a subset of RegionServers thus guaranteeing a coarse level of isolation.
- A namespace can be created, removed or altered.
  - Namespace membership is determined during table creation by specifying a fully-qualified table name of the form:
  - `<table namespace>:<table qualifier>`

## Version管理

- A  $\{row, column, version\}$  tuple exactly specifies a **cell** in HBase.
  - It's possible to have **an unbounded number of cells** where the row and column are the same but the cell address **differs only in its version dimension**.
  - The HBase version dimension is stored in **decreasing order**, so that when reading from a store file, the **most recent values** are found first.
- Specifying the Number of Versions to Store
  - The maximum number of versions to store for a given column is part of the column schema and is specified at table creation, or via an **alter** command, via **HColumnDescriptor.DEFAULT VERSIONS**.
  - *Modify the Maximum Number of Versions for a Column Family*
  - `hbase> alter 't1', NAME => 'f1', VERSIONS => 5`
  - *Modify the Minimum Number of Versions for a Column Family*
  - `hbase> alter 't1', NAME => 'f1', MIN VERSIONS => 5`

## HBase和RDBMS的对比

- **HBase是一种分布式、面向列的数据存储系统。**
  - 表模式反映了物理存储，为高效的数据结构序列化、存储和检索创建了一个系统
  - 应用程序开发人员有责任以正确的方式使用此存储和检索
- **典型的RDBMS**
  - 具有ACID属性和复杂SQL查询引擎的固定模式、面向行的数据库
  - 重点在于强大的一致性、引用完整性、物理层的抽象以及通过SQL语言进行的复杂查询
  - 您可以轻松创建二级索引，执行复杂的内部和外部联接，跨多个表、行和列对数据进行计数、求和、排序、分组和分页

以下是RDBMS与HBase之间的重要区别。

序号	键	关系数据库管理系统	HBase的
1 个	定义	RDBMS stands for Relational DataBase Management System.	HBase没有完整格式。
2	的SQL	RDBMS requires SQL, Structured Query Language.	HBase不需要SQL。
3	架构图	RDBMS has a fixed schema.	HBase没有固定的架构。
4	方向	RDBMS is row oriented.	HBase是面向列的。
5	可伸缩性	RDBMS faces problems in scalability.	HBase具有高度可扩展性。
6	性质	DBMS is static in nature.	HBase本质上是动态的。
7	资料检索	RDBMS data retrieval is slow.	HBase数据检索速度很快。
8	规则	RDBMS follows ACID(Atomicity, Consistency, Isolation and Durability) Rule.	HBase遵循CAP (一致性, 可用性, 分区容忍) 规则。
9	数据结构	RDBMS handles structural data.	HBase处理结构, 非结构和半结构数据。
10	稀疏数据处理	Sparse data handling is not present.	存在稀疏数据处理。

## lec30: hive

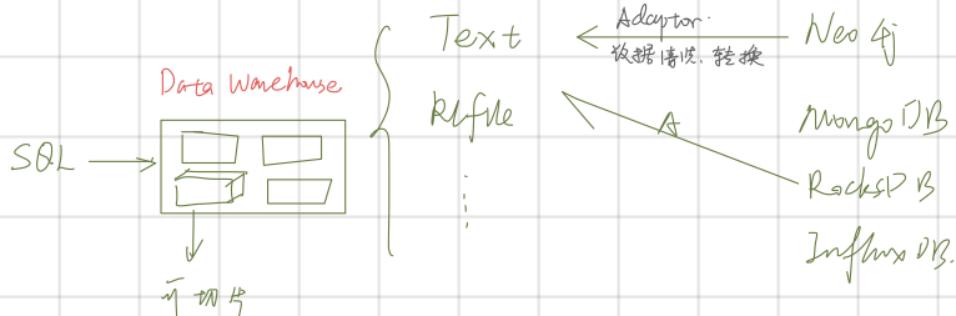
### Datawarehouse (数据仓库)

数据仓库是一个面向主题的 (Subject Oriented) 、集成的 (Integrate) 、相对稳定的 (Non-Volatile) 、反映历史变化 (Time Variant) 的数据集合，用于支持管理决策。

对于数据仓库的概念我们可以从两个层次予以理解，首先，数据仓库用于支持决策，面向**分析型数据处理**，它不同于企业现有的操作型数据库；其次，数据仓库是对**多个异构的数据源有效集成**，集成后按照主题进行了重组，并包含历史数据，而且存放在数据仓库中的数据一般不再修改。

**数据仓库中是一次写入多次读的，不会有改写，因为不是原始数据改它没有意义。**

# Hive 背景: Lake & repository.



数据库: 先定义好 schema 再写 : schema on write.

数据仓库: 先定义好表格式, 数据清洗转化, 检验 时间长, Lazy

↓  
problem: 原数据没了.

只有 SQL 语句用到了,  
才会做.

数据湖: 把原始数据放入, 要用到 才写入 Schema on Read.

数据仓库 everybody, 把 data 放到 Lake.

需求: 用一个 SQL 可对多个 data 操作

① 整理 data → 数据仓库

② 翻译 SQL: 把 SQL 翻译为不同 DB 认识的 SQL, → 性能差

只存数据湖

③ 判断 data 的访问频率, 高于一定值, 放入 Lake → repository.

∴ hot data → repository    cold data → lake.

湖仓一体: lakehouse.

## ETL: 清洗、转换、加载

目前数据湖和数据仓库的三种关系：

- 1、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

2、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

3、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



- Hive, a framework for **data warehousing on top of Hadoop**.
    - Hive grew from a need to manage and learn from the **huge volumes of data** that Facebook was producing every day from its burgeoning social network.
    - After trying a few different systems, the team chose Hadoop for storage and processing, since it was cost-effective and met their scalability needs.
  - Hive was created to make it possible for analysts with strong **SQL** skills (but meager Java programming skills) to run queries on the huge volumes of data that Facebook stored in **HDFS**.
    - Today, Hive is a successful **Apache** project used by many organizations as a general-purpose, scalable data processing platform.
  - <https://hive.apache.org/>
- 
- A yellow and black striped bumblebee logo with the word "HIVE" written below it in a bold, black, sans-serif font.

## 运行hive

1、先把分布式系统跑起来

- Hive uses Hadoop, so:
  - you must have Hadoop in your path OR
  - `$ export HADOOP_HOME=<hadoop-install-dir>`
- Start NameNode daemon and DataNode daemon:
  - `$ sbin/start-dfs.sh`
- In addition,
  - you must use below HDFS commands to create `/tmp` and `/user/hive/warehouse`
  - (aka `hive.metastore.warehouse.dir`) and set them `chmod g+w` before you can create a table in Hive.
  - `$ HADOOP_HOME/bin/hadoop fs -mkdir /tmp`
  - `$ HADOOP_HOME/bin/hadoop fs -mkdir /user/hive/warehouse`
  - `$ HADOOP_HOME/bin/hadoop fs -chmod g+w /tmp`
  - `$ HADOOP_HOME/bin/hadoop fs -chmod g+w /user/hive/warehouse`

`/tmp`用于存数据ETL过程的临时文件

`/user/hive/warehouse`

需要检查namenode和datanode是否都跑起来了。

配置文件：`<install-dir>/conf/hive-default.xml`

所有的操作都是基于mapreduce, 但会自己优化, 有时候就变成本地了。

保证mapreduce尽量高效

## partition column : 新建分区表

按hash分区 ? 按range分区 ? 是否合理

Data Load可以指定文件搜索 :

Insert数据存在hdfs里面, 可选overwrite

Note that in all the examples that follow, `INSERT` (into a Hive table, local directory or HDFS directory) is optional.

- `hive> INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT a.* FROM invites a WHERE a.ds='2008-08-15';`
- selects all rows from partition ds=2008-08-15 of the invites table `into an HDFS directory`.
- The result data is `in files (depending on the number of mappers)` in that directory.  
NOTE: partition columns if any are selected by the use of `*`. They can also be specified in the projection clauses.
- Partitioned tables must always have a partition selected in the `WHERE` clause of the statement.

## 数据仓库 : 数据一旦进入一个表格就基本不会改, 改数据在源文件改

各种数据库都支持一种数据格式, 这种数据格式又可以被hive接受

hive文件存储格式包括：TEXTFILE、SEQUENCEFILE、RCFILE、自定义格式

其中TEXTFILE为默认格式，建表时不指定默认为这个格式，导入数据时会直接把数据文件拷贝到hdfs上不进行处理。

SequenceFile、RCFile格式的表不能直接从本地文件导入数据，数据要先导入到textfile格式的表中，然后再从textfile表中用insert导入到SequenceFile、RCFile表中。

## RCFile : Hive中一种新的存储格式，将数据按列存储

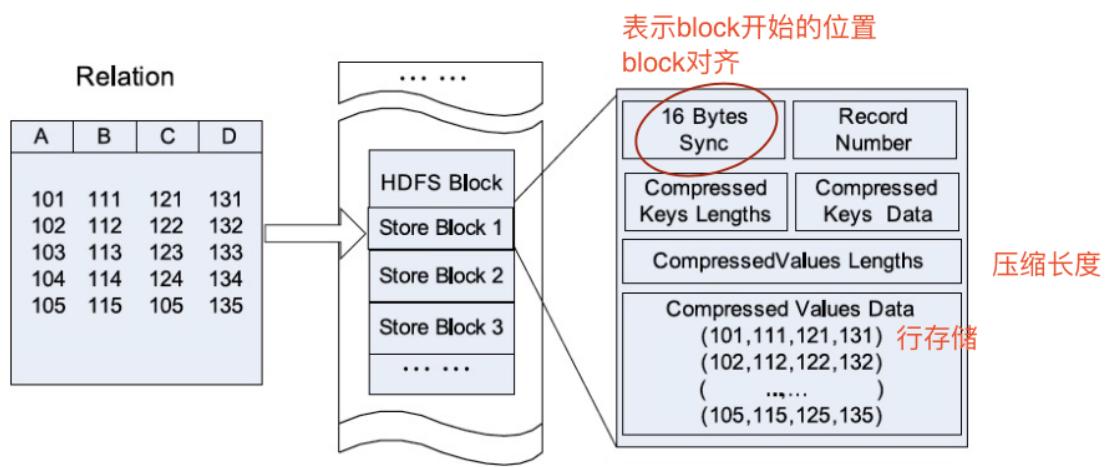


Fig. 1: An example of row-store in an HDFS block.

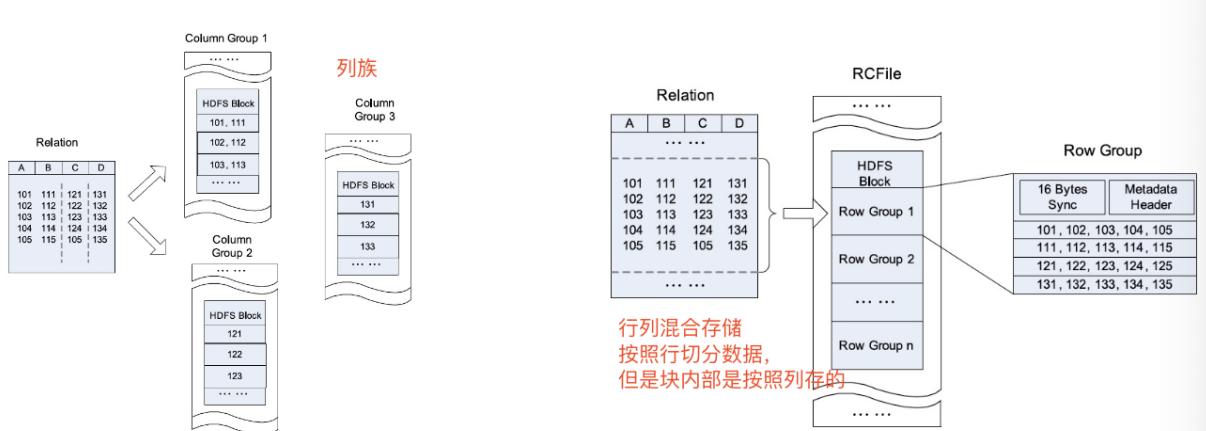


Fig. 2: An example of column-group in an HDFS block. The four columns are stored into three column groups, since column A and B are grouped in the first column group.

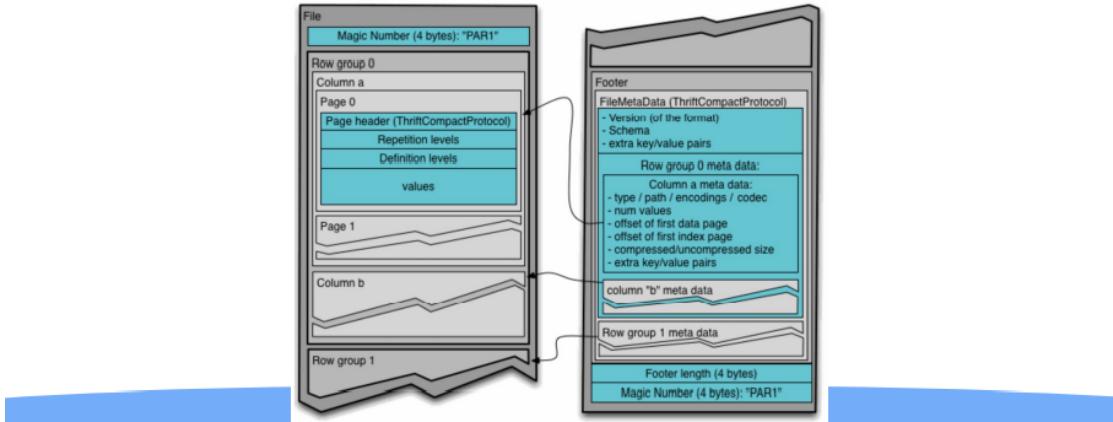
Fig. 3: An example to demonstrate the data layout of RCFile in an HDFS block.

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

- 几乎所有数据库都支持Parquet

## Apache Parquet

- is a [columnar storage](#) format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.
- <http://parquet.incubator.apache.org/>



如果文本文件太大要压缩：hive可以直接将压缩文件加载到表里面，但是没有压缩不能进行分区，所以在mapreduce时只能用一个map和一个reduce，对这个表的操作就比较麻烦了