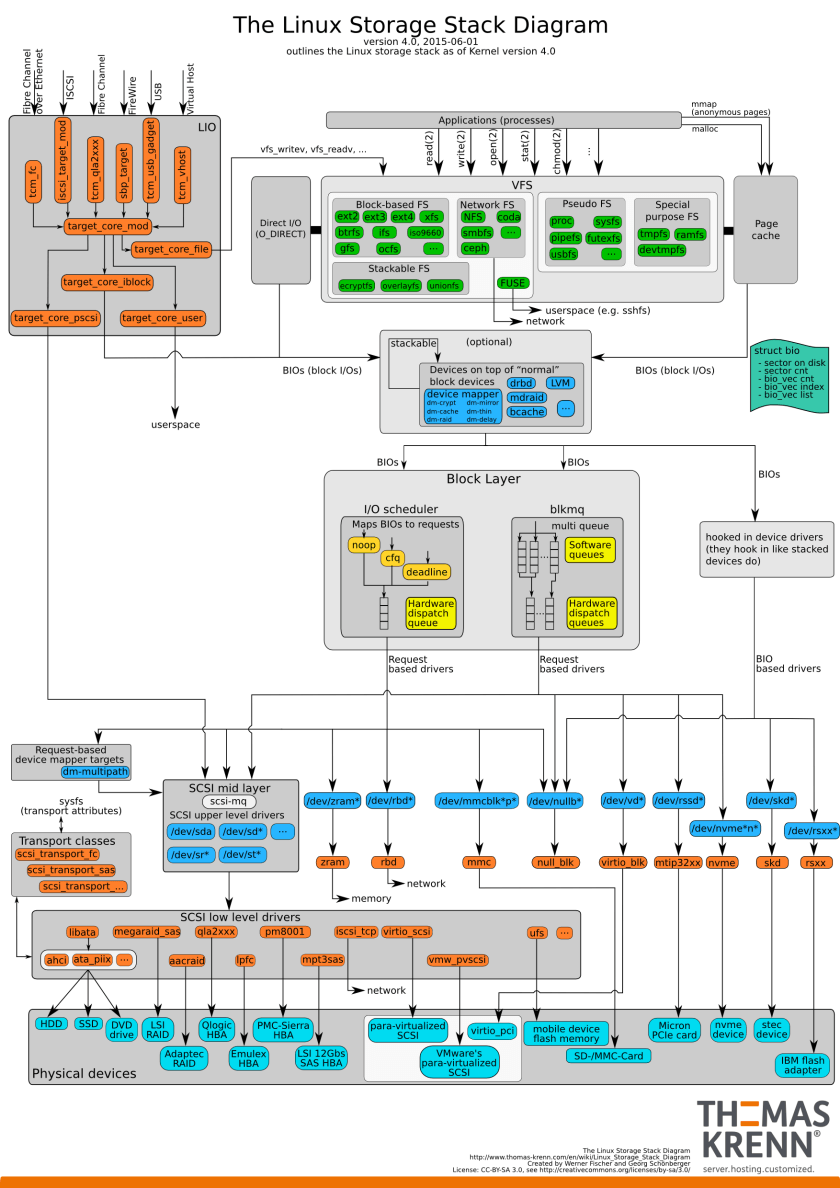
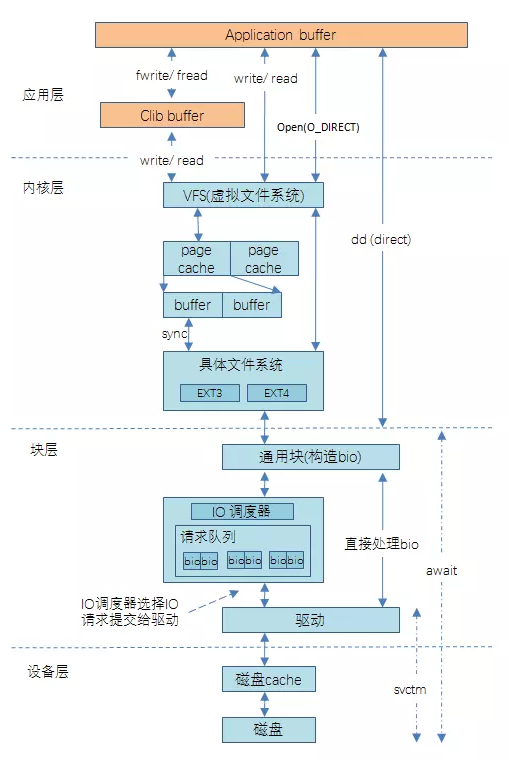
# IO

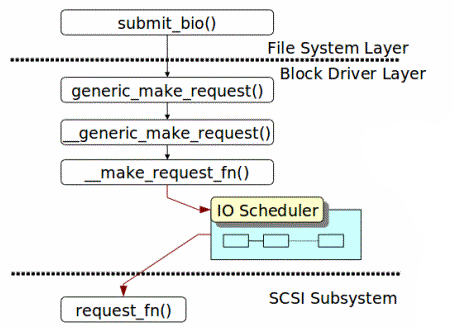
## 磁盘IO流程

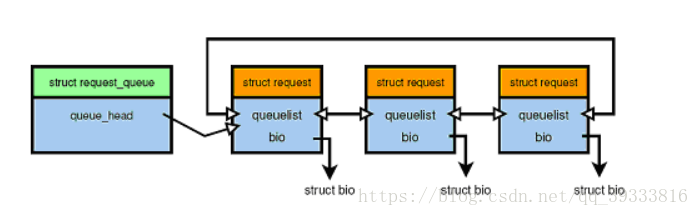
详细图：





下图是文件系统与块层的交互





在块层，文件系统下来的数据，先要构造成 bio 数据结构，这里是可以优化的，如果文件系统要访问的地址有相邻的，可以合并。Bio 提交给驱动有两种方式，一种是通过调度器，一种是直接提交。合并后的request会放入每个device对应的queue（一个机械硬盘即使有多个分区，也只有一个queue）里，之后，磁盘设备驱动程序通过调用peek\_request从queue里取出request，进行下一步的处理。

# 分布式系统

## Paxos算法

N个人闯关，每个关卡有 m条路，并且每条路都有一个大石头，只有超过一半 N/2 的人来才能合力抬得起石头，要到达终点，要么超过 N/2 个人到达终点，要么都堵在半路上。

## Raft算法

汽车包车，但是票是一张一张卖的，车上有 N 个座位，规定只有抢到 N/2 个座位才能包这辆车，如果在30分钟内没有人成功预订到 N/2 个座位，抢票重新开始，直到成功为止。

在Raft中使用了一个可以理解为任期的概念，用Term作为一个周期，每个Term都是一个连续递增的编号，每一轮选举都是一个Term周期，在一个Term中只能产生一个Leader；

其中Term的变化流程：

Raft开始时所有Follower的Term为1，其中一个Follower逻辑时钟到期后转换为Candidate，Term加1这是Term为2，然后开始选举，这时候有几种情况会使Term发生改变：

　　1：如果当前Term为2的任期内没有选举出Leader或出现异常，则Term递增，开始新一任期选举

　　2：当这轮Term为2的周期选举出Leader后，过后Leader宕掉了，然后其他Follower转为Candidate，Term递增，开始新一任期选举

　　3：当Leader或Candidate发现自己的Term比别的Follower小时Leader或Candidate将转为Follower，Term递增

　　4：当Follower的Term比别的Term小时Follower也将更新Term保持与其他Follower一致；

选举（Election）

Raft的选举由定时器来触发，每个节点的选举定时器时间都是不一样的，默认是0~1000ms之间，开始时状态都为Follower某个节点定时器触发选举后Term递增，状态由Follower转为Candidate，向其他节点发起RequestVote RPC请求，这时候有三种可能的情况发生：

　　1：该RequestVote请求接收到n/2+1（过半数）个节点的投票，从Candidate转为Leader，向其他节点发送heartBeat以保持Leader的正常运转

　　2：在此期间如果收到其他节点发送过来的AppendEntries RPC请求，如该节点的Term大则当前节点转为Follower，否则保持Candidate拒绝该请求

　　3：Election timeout发生则Term递增，重新发起选举

在一个Term期间每个节点只能投票一次，所以当有多个Candidate存在时就会出现每个Candidate发起的选举都存在接收到的投票数都不过半的问题，这时每个Candidate都将Term递增、重启定时器并重新发起选举，由于每个节点中定时器的时间都是随机的，所以就不会多次存在有多个Candidate同时发起投票的问题。

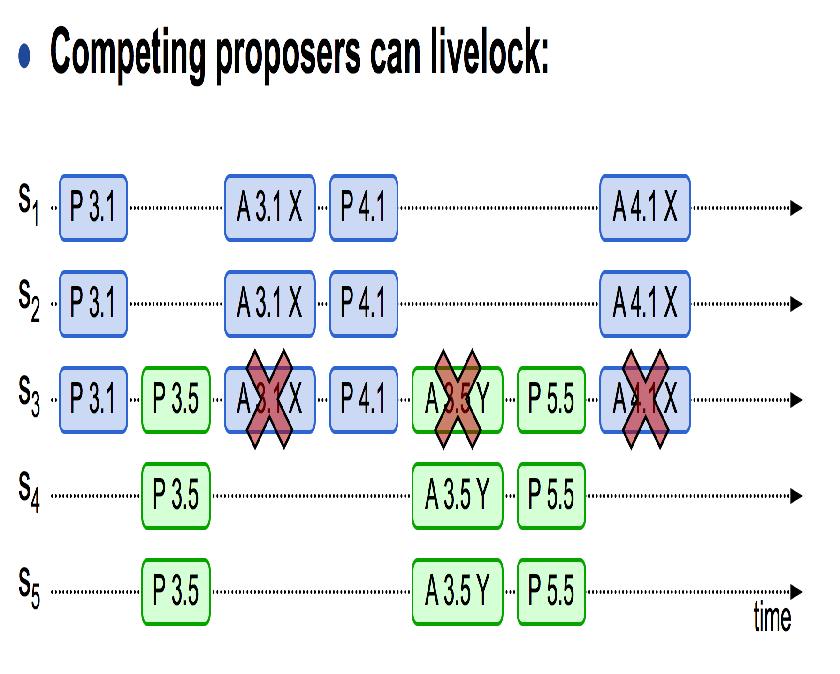
有这么几种情况会发起选举，1：Raft初次启动，不存在Leader，发起选举；2：Leader宕机或Follower没有接收到Leader的heartBeat，发生选举超时从而发起选举;

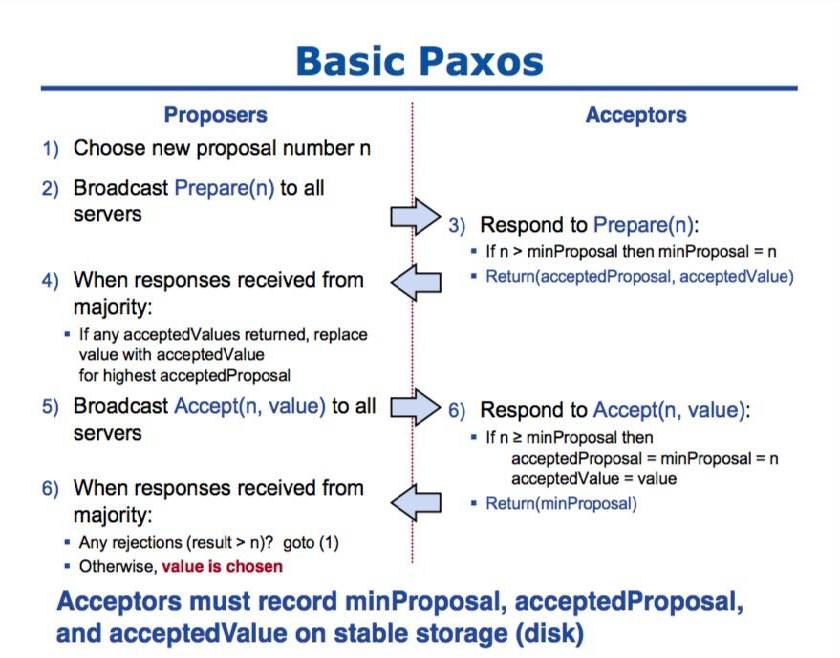
所以在选举过程中，很关键一点就是选举定时器时间，由于这个时间是随机的，在0~1000ms之间，所以最新醒来的机器能够很快给其他机器发送投票，最终能很快达成一致，选出Leader

## Base Paxos算法

Paxos 算法的核心思想是少数服从多数，通过提案编号保证选定决议值的过程有序。在一个提案周期内，不允许新的提案产生，否则重新选定决议值。

Base paxos 算法收敛性无法保证，在 prepare 和 acceptor 交叉执行时会产生活锁。





1. Prepare

Proposer 生成有序提案编号 N，将该提案号通过Perpare(N) 向 acceptor 广播。Acceptor 记录三个值 promiseNo (承诺过最大提案编号)、acceptedProposalNo（已接受的最大提案编号）、acceptedProposalValue（已接受最大提案值），这三个值的初始值都是 NULL。

如果 N > promiseNo, 返回 (acceptedProposalNo, acceptedProposalValue)

否则可以不反回，或者返回错误信息

1. Accept

Proposer 端每次收到acceptor端返回值后，令 proposalValue = max{ proposalValue, acceptedProposalValue }（若没返回值可以不执行），超过半数 acceptor 返回后，即可调用 Accept(N, proposalValue) 来更新Acceptor 端数据。

当 N > promiseNo 时，promiseNo = acceptedProposalNo = N 并且acceptedProposalValue = proposalValue

返回promiseNo 的值，即使 N 不大于promiseNo 也要返回。

Proposer 端判断返回的promiseNo 是否大于 N，若大于 N 说明又被更新了，要重新进行下一个提案。

## FLP理论概要

在异步的系统中，当一个进程出现故障，或者响应丢失时，是无法检测到的。在这样的条件下，如果其中有任意一个进程出现问题，没有任何一个分布式算法，可以让所有的非故障进程，达成一致性共识。

因为有FLP不可能性的限制，大部分[区块链](http://www.bitcoin86.com/block/)项目的共识算法都把大部分节点是诚实的和满足一定的同步性作为前提。POW认为51% 的节点是诚实的，并且有一定的同步性。POS和PBFT也认为大部分节点（66%）是诚实的.

## CAP理论

* 一致性（Consistency）：Every read receives the most recent write or an **error**

这里强调的是每一次读操作，它并不要求在某一时刻，所有节点上的数据都是一样的，只是要求，每次都能读到最近一次写入的数据。

* 可用性（Availability）：Every request receives a (non-error) response ，但是不保证能返回最后一次写入的数据

这里要求每次请求都能返回数据，即使是历史数据，都行。

* 分区容错（Partition tolerance）：The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network betwee nodes。

保证集群一直能正常运行。

CA (consistency + availability)，这样的系统关注一致性和可用性，它需要非常严格的全体一致的协议，比如“两阶段提交”（2PC）。CA 系统不能容忍网络错误或节点错误，一旦出现这样的问题，整个系统就会拒绝写请求，因为它并不知道对面的那个结点是否挂掉了，还是只是网络问题。唯一安全的做法就是把自己变成只读的。

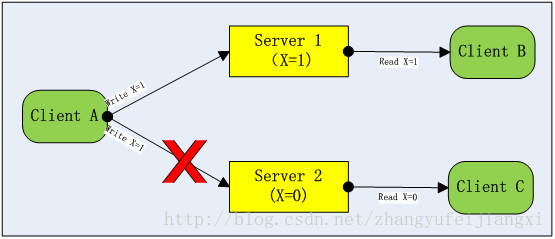
CP (consistency + partition tolerance)，这样的系统关注一致性和分区容忍性。它关注的是系统里大多数人的一致性协议，比如：Paxos 算法 (Quorum 类的算法)。这样的系统只需要保证大多数结点数据一致，而少数的结点会在没有同步到最新版本的数据时变成不可用的状态。这样能够提供一部分的可用性。

AP (availability + partition tolerance)，这样的系统关心可用性和分区容忍性。因此，这样的系统不能达成一致性，需要给出数据冲突，给出数据冲突就需要维护数据版本。Dynamo 就是这样的系统。

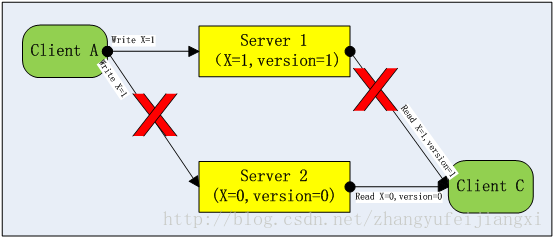
自己的理解：

一致性：

如下图1中所示，Client A负责更新数据，为了保证Server 1和Server 2上的数据是一致的，Client A会将X=1的写操作同时发给Server 1和Server 2，但是当Client A和Server 2之间发生网络分区（网络无法连接）时，此时如果让write X=1的写操作在Server 1上成功，那Client B和Client C将从Server 1和Server 2上读取到不一致的X值；此时如果要保持X值的一致性，那么write X=1的写操作在Server 1和Server 2上都必须失败，这就是著名的CAP理论：在容忍网络分区的前提下，要么牺牲数据的一致性，要么牺牲写操作的可用性。



解决这个问题你可能会想到让Client C同时读取Server 1和Server 2上的X值和版本信息，然后取Server 1和Server 2最新版本的X值, 如下图2所示。但Client C和Server 1之间也可能发生网络分区，这本质上是牺牲读可用性换取写可用性，并没有突破CAP理论。

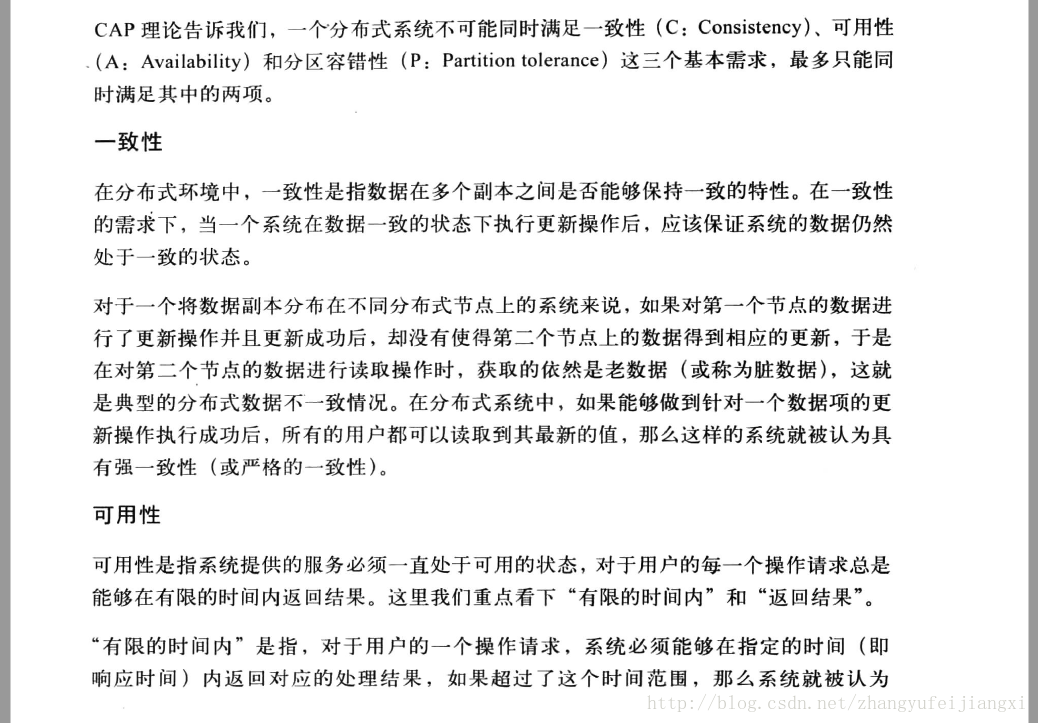


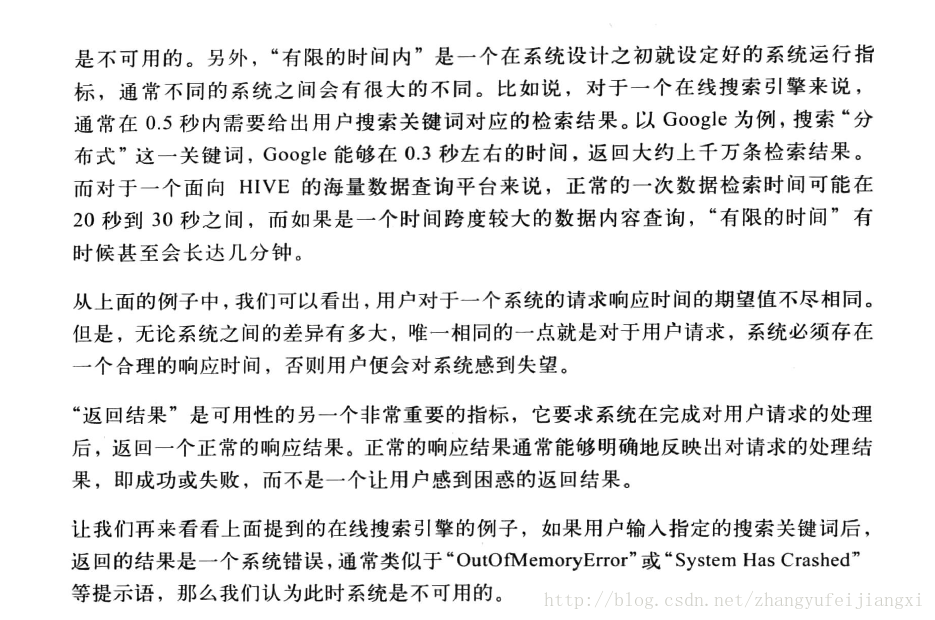
可用性：读写操作在单台服务器出问题后，在其他服务器上依然能够完成读写操作

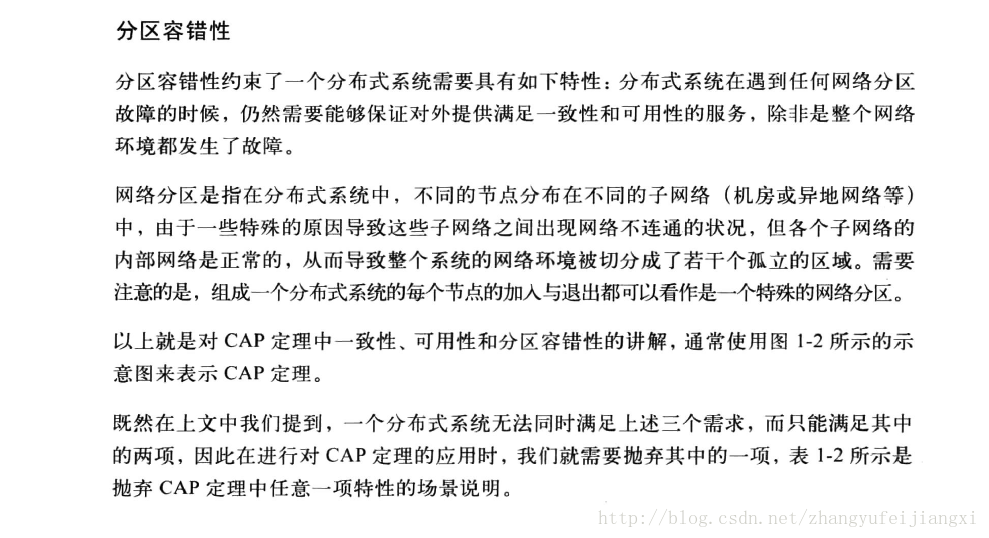
重点在于：某个读写操作在出问题的机器上不能读写了，但是在其他机器可以完成

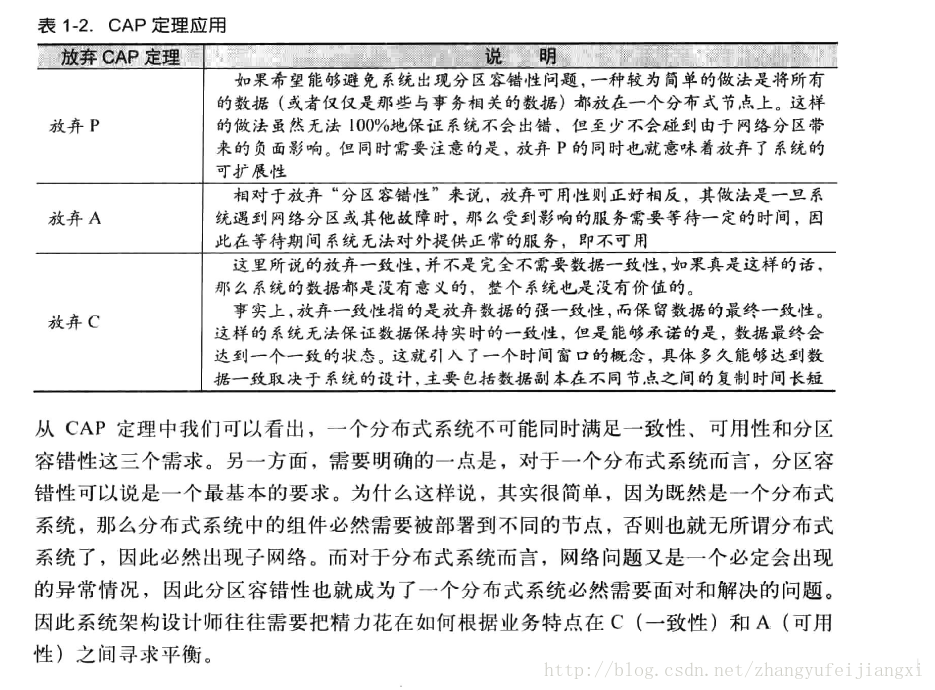
分区容错性：单台服务器，或多台服务器出问题（主要是网络问题）后，正常服务的服务器依然能正常提供服务，并且满足设计好的一致性和可用性

重点在于：部分服务器因网络问题，业务依然能够继续运行









1. Fallacies of Distributed Computing

## 经典资料

1. Distributed systems theory for the distributed systems engineer
2. FLP Impossibility Result
3. An introduction to distributed systems
4. Distributed Systems for fun and profit
5. Distributed Systems: Principles and Paradigms
6. Scalable Web Architecture and Distributed Systems
7. Principles of Distributed Systems
8. Making reliable distributed systems in the presence of software errors
9. Designing Data Intensive Applications

## 通信方式

进程间通过网络直接通信或者读写共享存储，zookeeper采用读写共享存储的方式实现的。

## 任务

未分配（未完成）、已分配（完成、失败、进行中），可以看出一共是四种状态。

## 通信问题

在一个分布式系统中，没有收到某个进程的消息，可能是进程所在节点崩溃了，也可能是时延，这是没法区别对待的。

主节点崩溃，任务无法调度；从节点崩溃，任务无法执行；通信故障，一致性受干扰。

分布式系统要解决这些问题，主节点崩溃要有替代的，新的主节点要知道全部任务的状态。从节点崩溃或者一定时间内无反馈，主节点要处理这种问题；通信故障时怎么办。

### 消息传输延迟

消息到达时间点不可控，先发的消息可能后到达。

### 消息处理延时

操作系统调度和超出负载，处理时间会变化。

### 时钟偏移

各设备接收到信号的时间是不一致的。

# Zookeeper

## Windows伪集群搭建

本次准备启动 3 个实例，其中 zookeeper 配置文件需要修改，并且根据配置文件创建相应的目录，供实例使用。后面以 S1、S2、S3 作为这 3 个实例的名字。

分别为S1、S2、S3 创建配置文件zoo1.cfg，zoo2.cfg, zoo3.cfg内容如下：

|  |
| --- |
| #这个时间是作为 Zookeeper 服务器之间或客户端与服务器之间维持心跳的时间间隔，#也就是每个 tickTime 时间就会发送一个心跳。  tickTime=2000  #集群中的follower服务器(F)与leader服务器(L)之间初始连接时能容忍的tickTime的数 #量，总时间10\*tickTime = 20s  initLimit=10  #集群中的follower服务器与leader服务器之间请求和应答之间能容忍的tickTime的数  #量，5\*tickTime = 10s  syncLimit=5  #数据目录  ***dataDir=D:\\kafkaData\\zookeeper\\z1\\data***  #日志目录  ***dataLogDir=D:\\kafkaData\\zookeeper\\z1\\log***  #zookeeper 对外提供服务的端口  clientPort=2181  #下面三行是配置集群的关键  #2888 所在列是服务器与集群中的 Leader 服务器交换信息端口  #2889所在列是选举时服务器相互通信的端口  server.1=localhost:2888:2889  server.2=localhost:2890:2891  server.3=localhost:2892:2893  #之所以所有的实例，都要有这同样的三行配置，原因很简单，每个实例都要了解彼此  #同时，每个实例既是客户端，在必要时候也做服务端。 |
| #S2配置文件不同的地方如下：  #数据目录  ***dataDir=D:\\kafkaData\\zookeeper\\z2\\data***  #日志目录  ***dataLogDir=D:\\kafkaData\\zookeeper\\z2\\log***  #zookeeper 对外提供服务的端口  clientPort=2182 |
| #S3配置文件不同的地方如下：  #数据目录  ***dataDir=D:\\kafkaData\\zookeeper\\z3\\data***  #日志目录  ***dataLogDir=D:\\kafkaData\\zookeeper\\z3\\log***  #zookeeper 对外提供服务的端口  clientPort=2183 |

由于我们在配置文件中，自定义了数据目录和日志目录，所以我们要手动创建这些目录，创建命令如下

mkdir D:\kafkaData\zookeeper\z1\data

mkdir D:\kafkaData\zookeeper\z2\data

mkdir D:\kafkaData\zookeeper\z3\data

mkdir D:\kafkaData\zookeeper\z1\log

mkdir D:\kafkaData\zookeeper\z2\log

mkdir D:\kafkaData\zookeeper\z3\log

除配置文件外，还要为每个实例编一个号，这个编号存在 myid 这个文件中，只能有一行，创建命令如下

echo 1>>D:\kafkaData\zookeeper\z1\data\myid

echo 2>>D:\kafkaData\zookeeper\z2\data\myid

echo 3>>D:\kafkaData\zookeeper\z3\data\myid

用文本编辑器打开这三个文件，确保他们只有一行。这样配置文件和相关目录我们已经有了，剩下的就是修改启动脚本，方便我们启动实例。

复制zkServer.cmd，分别为zkServer1.cmd、zkServer2.cmd、zkServer3.cmd，在每个脚本 set ZOOMAIN=org.apache.zookeeper.server.quorum.QuorumPeerMain这句下面加入下面设置自己的配置文件路径：

set ZOOCFG=%ZOOCFGDIR%\zoo1.cfg

set ZOOCFG=%ZOOCFGDIR%\zoo2.cfg

set ZOOCFG=%ZOOCFGDIR%\zoo3.cfg

依次双击zkServer1.cmd、zkServer2.cmd、zkServer3.cmd 就会启动这三个实例，这样一个 zookeeper 集群就搭建好了。

## 连接集群

zkCli.sh -server 127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183

## 客户端命令

cZxid: creation ID

ctime: 创建时间

mZxid: Modified ID

mtime: 修改时间

pZxid: 节点ID

cversion: 子节点版本

dataVersion: 数据版本

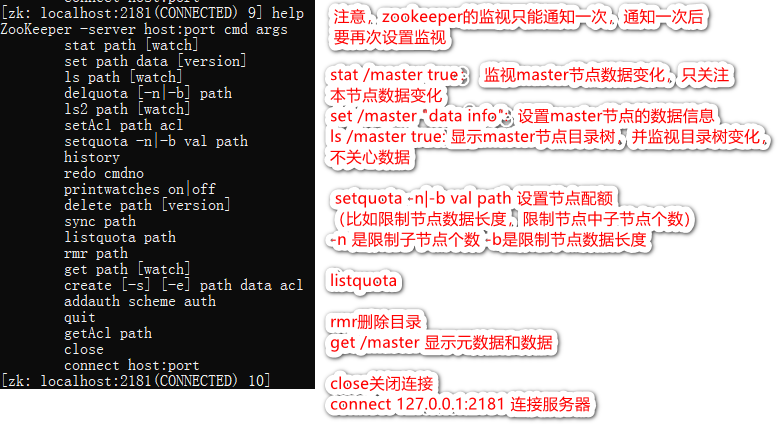
aclVersion: acl 版本

ephemeralOwner:

dataLength: 数据长度

numChildren: 子节点数量

直接双击 zkCli.cmd 可以启动，或者zkCli.sh -server 127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183。最终还是通过 java 工具启动的。



## 会话

会话标识着客户端与服务端的交互，会话不同于连接，会话可以从一个服务器实例转移到另外一个服务器实例，连接不行。

## 节点类型

1. 持久节点（persistent）

持久节点，只能通过 delete 指令来删除。

1. 临时节点（ephemeral）

当创建该节点的客户端与 zk 的会话不存在了的时候，会自动清除。当然也可以手动清除。

1. 持久有序节点（persistent\_sequential）

当创建有序节点时，一个序号会被追加到路径之后。例如，如果一个客户端创建了一个有序znode节点，其路径为/tasks/task-，那么ZooKeeper将会分配一个序号，如1，并将这个数字追加到路径之后，最后该znode节点为/tasks/task-1。有序znode通过提供了创建具有唯一名称的znode的简单方式。同时也通过这种方式可以直观地查看znode的创建顺序。

1. 临时有序节点（ephemeral\_sequential）

## 监视与通知

可以对节点设置观察，客户端注册要被观察的节点，当节点上有事件发生时，采用通知机制来告知客户端数据的变化。

## 版本

每个节点都有一个版本，只有版本匹配时，才能修改数据。

会话

## 原理

zookeeper启动后，会将所有数据加载到内存中，但是也会在硬盘上留下数据文件和日志文件便于恢复。与此相关的配置项如下：

dataDir：保存内存快照的目录

dataLogDir：保存事务日志的目录

globalOutstandingLimit ：单台服务器上未完成的最大客户端请求数(当客户端发送请求太快而集群来不及处理时)，默认1000.

preAllocSize：事务日志大小，默认是64M

snapCount：进行快照的阀值，默认10000；当事务日志条数达到 snapCount/2 + rand.nextInt(snapCount/2)时就做一次快照，使用随机数的原因是避免所有服务器会同时做快照

zookeeper的快照数据保存在 dataDir 中，文件的命名格式为：[snapshot + lastZxid的16进制字符串]。相关序列化和反序列化的类可以看 FileSnap 类实现。快照文件中的内容包括文件头信息，目录树及一些session数据。

事物日志则保存在 dataLogDir 中，日志文件的命名格式 [log + lastZxid的16进制字符串]，具体实现看 FileTxnLog 。日志文件中的数据格式稍微复杂一点，如下：

\* 事务日志文件:

\*

<blockquote>

<pre> \* LogFile:

\* FileHeader TxnList ZeroPad

\*

\* FileHeader: {

\* magic 4bytes (ZKLG)

\* version 4bytes

\* dbid 8bytes

\* }

\*

\* TxnList:

\* Txn || Txn TxnList

\*

\* Txn:

\* checksum Txnlen TxnHeader Record 0x42

\*

\* checksum: 8bytes Adler32 is currently used

\* calculated across payload -- Txnlen, TxnHeader, Record and 0x42

\*

\* Txnlen:

\* len 4bytes

\*

\* TxnHeader: {

\* sessionid 8bytes

\* cxid 4bytes

\* zxid 8bytes

\* time 8bytes

\* type 4bytes

\* }

\*

\* Record:

\* See Jute definition file for details on the various record types

\*

\* ZeroPad:

\* 0 padded to EOF (filled during preallocation stage)

如果需要分析日志文件的话，可以考虑对FileTxnLog和FileTxnIterator做简单修改来暴露数据读取接口。

数据恢复过程

当进行一次快照时，就会重新生成一个新的日志文件，二者结合可以把数据恢复(断点+动作重放)，日志文件记录的事务比快照小，这样好进行回放，启动时数据恢复的具体的逻辑在 FileTxnSnapLog 中：

snapLog.deserialize(dt, sessions);

FileTxnLog txnLog = new FileTxnLog(dataDir);

TxnIterator itr = txnLog.read(dt.lastProcessedZxid+1);

long highestZxid = dt.lastProcessedZxid;

代码中先读取快照数据生成断点映像，然后根据断点映像中最大zxid开始进行事务重放

zookeeper节点间的数据同步

当各个节点已经自我恢复并选举出leader后，leader就开始和follows进行数据同步了，具体的逻辑可以见 LearnerHandler 中：

leader构建NEWLEADER包，内含leader最大数据的zxid, 广播给follows，然后leader根据follower数量为每个follower创建一个LearnerHandler线程来处理同步请求：leader主线程阻塞，等待超过半数follower同步完数据之后成为正式leader。

follower接收到NEWLEADER包后响应FOLLOWERINFO给leader，告知本方数据最大的zxid值； leader接收到回馈后开始判断：

如果follower和leader数据一致，则直接发送DIFF告知已经同步；判断这一阶段内有无已经北提交的决议值，如果有，那么

a) 如果有部分数据没有同步，leader发送DIFF包将有差异的数据同步过去，同时将follower没有的数据逐个发送commit包给follower要求记录下来；

b) 如果follower数据zxid更大，发送TRUNC包给follower要求删除多余数据

如果这一阶段没有提交的决议，直接发送SNAP包将快照同步给follower

以上消息完毕后，LEADER发送UPTODATE包告知follower当前数据已同步，等待follower的ACK完成同步过程。

Zab协议有两种模式，分别是恢复模式和广播模式。

恢复模式

当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和server具有相同的系统状态。

广播模式

一旦Leader已经和多数的Follower进行了状态同步后，他就可以开始广播消息了，即进入广播状态。这时候当一个Server加入ZooKeeper服务中，它会在恢复模式下启动，发现Leader，并和Leader进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper服务一直维持在Broadcast状态，直到Leader崩溃了或者Leader失去了大部分的Followers支持。

Zxid是64 位，可以保证，新的 leader 产生的 zxid 总是大于旧的 leader 产生的 zxid。

# Redis

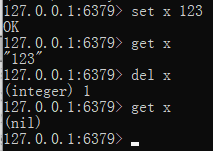
## redis主要特性

使用内存存储的非关系型数据库，数据类型有字符串、列表、集合、散列、有序集合。

## 数据类型

### string字符串

#### Get set del



### list 链表结构

1. 右端只支持栈操作，分别为RPUSH、RPUSHX、RPOP、RPOPLPUSH

RPUSH lst\_1 a 右端添加 a

RPUSHX lst\_1 a 若存在 lst\_1 链表，添加 a

RPOP lst\_1 弹出lst\_1 链表栈顶元素

RPOPLPUSH lst\_1 lst\_2弹出lst\_1 链表栈顶元素加到lst\_2链表左边

1. 左端增、删、改、查 LINSET、LREM (一次删除多个，不能准确删除)、LSET、LRANGE、LINDEX

长度 LLEN

栈操作 LPOP、LPUSH、LPUSHX

1. 阻塞操作 BLPOP、BRPOP、BRPOPLPUSH

### set 集合，无序、互异

1. 元素—集合运算

增、删、查 sadd、srem、smembers、sismember、spop随意弹出一个、

1. 集合—结合运算

并交差 sunion、sinter、sdiff、sdiffstore、sinterstore、sunionstore

Smove 从一个 set 移动到另一个set

计数 scard

### hset 实际上时hashmap

增或者更新 hset、新增键值对hsetnx ，更新则失败

删 hdel hashmap key1、key2.。。。

查Hget hashmap key、查全部 hget hashmap、查全部key, hkeys hashmap、查全部value, hvals hashmap、多值查询 hmget hashmap key1 key2 ... 、多值设置 hmset key1 value1 key2 value2

存在key, Hexists hashmap key

大小hlen hashmap

### ZSET

<STRING, 浮点数> 有序map，根据浮点值排序, 跟set 的操作差不多

## windows服务端

redis-server.exe 命令，按照默认配置启动

## windows客户端

redis-cli.exe 按照默认方式连接

### 命令

#### 禁止提示nohints

:set nohints

#### 获取string全部函数

Help @string， help zadd 获取zadd命令帮助

## Redis原理

### redis-sentinel 哨兵

sentinel可保证redis高可用，无法保证一致性。

#### 启动

redis-server /path/to/your/sentinel.conf –sentinel 指定redis服务器以 sentienl 模式运行。

# Kafka

## 原理

### offset更新原理

主要是 log end offset (LEO)日志最终偏移量，high watermark (HW) 高水印表示当前已经 commit 的消息。

1. producer 写入 n 条消息记录，leader\_leo += n

2. leader尝试更新 leader\_hw

如果 min(follower\_leo, leader\_leo) 大于 leader\_hw

则更新 leader\_hw = min(follower\_leo, leader\_leo) 。

这是消息写入成功后相关值的变化。

3. follower 发来 fetch 请求到leader

leader 可以更新 follower\_leo 和 leader\_hw

4. 响应 fetch 请求

follower端会先把 log 数据写入，同时更新 follower\_leo，

然后如果 min{follower\_leo, leader\_hw} 大于 follower\_hw

还要更新 follower\_hw 的值，所以 follower\_hw 不可能大于

### consumer源码分析

#### group rebalance

方案一：

监控Zookeeper上 consumer/group/ids 节点变化，问题是通知过多(羊群效应)，zookeeper的脑裂问题导致错误的rebalance 尝试。

方案二：

Consumer group交给服务器端 GroupCoordinator 管理。每个 group 对应一个 GroupCoordinator 对象。GroupCoordinator 监控consumer/group/ids 变化，并执行 rebalance 操作。

1. 新消费者先向任一 broker 发送 ConsumerMetadataRequest请求，包含了 groupid信息，Broker 会返回ConsumerMetadataResponse的应答，包含了GroupCoordinator信息。
2. Consumer 通过 HeartBeatRequest 与GroupCoordinator 保持联系。若GroupCoordinator长时间未收到HeartBeatRequest 会执行 rebalance操作
3. 若 HeartbeatRequest 包含错误，则 rebalance正在进行，发送 JoinGroupReqest 请求，GroupCoordinator 将分配信息写入 zookeeper 并返回分配结果。

问题是 consumer rebalance策略由服务端控制，不能由消费者掌握自己的命运，不灵活。

方案三：

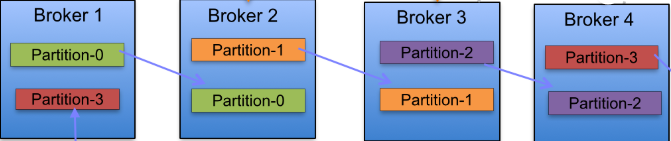
GroupCoordinator先指定一个 group leader 并将 consumer group的信息发给 leader，leader将 rebalance 结果反给GroupCoordinator ，最终发给所有的 consumer

### Controller源码

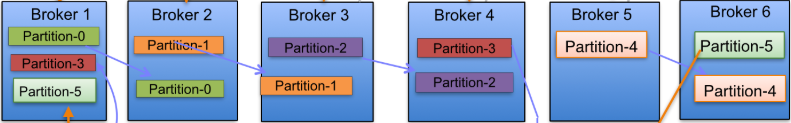
## Partion与broker

Partition一旦分配到 Broker 上，就不能随便移动了，我们可以增加 Broker 这很简单，但是如何把 Partation 往 Broker上搬呢？

下面以一个Kafka集群中4个Broker举例，创建1个topic包含4个Partition，2 Replication；数据Producer流动如图所示：



(2)当集群中新增2节点，Partition增加到6个时分布情况如下：



副本分配算法如下：

假设 Broke数量为 N，partition 数量为 M ，先将所有 Broker和待分配的 Patition 排序。将第i个Partition分配到第(i mod N)个Broker上，将第i个Partition的第j个副本分配到第((i + j) mod n)个Broker上。

本例中，刚开始 4 个可以均匀分配，增加 2 个Broker 和 Partition 数量变成 6 后，Partition 5、6是待分配的，Broker 5、6分别分配 第 5、6 个 Partition，根据算法第 6 个 Partition的第 1个副本 (6 + 1) mod 6 = 1，要分配到第一个 broker 上，这就导致第一个Broker 分配了 3 个 Partition。所以一般第一个Partition 是随机分配到 Broker 上，再依次往下分。

## Broker配置

advertised.host.name = null

advertised.listeners = null

advertised.port = null

alter.config.policy.class.name = null (更改配置策略类)

alter.log.dirs.replication.quota.window.num = 11 (更改日志副本配额窗口数量)

alter.log.dirs.replication.quota.window.size.seconds = 1

authorizer.class.name = (权限类)

auto.create.topics.enable = true (往不存在的topic写，会创建topic)

auto.leader.rebalance.enable = true

background.threads = 10 (后台辅助线程数)

broker.id = 0

broker.id.generation.enable = true

broker.rack = null

client.quota.callback.class = null

compression.type = producer (消息压缩)

connection.failed.authentication.delay.ms = 100

connections.max.idle.ms = 600000 (超时连接被关闭)

connections.max.reauth.ms = 0

control.plane.listener.name = null

controller 跟 broker 通信

broker配置如下：

listeners = INTERNAL://192.1.1.8:9092, EXTERNAL://10.1.1.5:9093, CONTROLLER://192.1.1.8:9094

listener.security.protocol.map = INTERNAL:PLAINTEXT, EXTERNAL:SSL, CONTROLLER:SSL

control.plane.listener.name = CONTROLLER

broker元数据如下：

["INTERNAL://broker1.example.com:9092","EXTERNAL://broker1.example.com:9093","CONTROLLER://broker1.example.com:9094"]

controller配置如下:

listener.security.protocol.map = INTERNAL:PLAINTEXT, EXTERNAL:SSL, CONTROLLER:SSL

control.plane.listener.name = CONTROLLER

那么，broker 与 controller 都是使用 9094端口来通信

controlled.shutdown.enable = true (允许控制器关闭broker)

controlled.shutdown.max.retries = 3

controlled.shutdown.retry.backoff.ms = 5000

controller.socket.timeout.ms = 30000

create.topic.policy.class.name = null

#默认副本数量，不能大于broker数量

default.replication.factor = 1

delegation.token.expiry.check.interval.ms = 3600000

delegation.token.expiry.time.ms = 86400000

delegation.token.master.key = null

delegation.token.max.lifetime.ms = 604800000

delete.records.purgatory.purge.interval.requests = 1

delete.topic.enable = true (允许删除topic)

fetch.purgatory.purge.interval.requests = 1000

group.initial.rebalance.delay.ms = 0

group.max.session.timeout.ms = 300000

group.max.size = 2147483647

group.min.session.timeout.ms = 6000

host.name =

inter.broker.listener.name = null

inter.broker.protocol.version = 2.2-IV1

kafka.metrics.polling.interval.secs = 10

kafka.metrics.reporters = []

leader.imbalance.check.interval.seconds = 300 (partion不均衡检测频率)

leader.imbalance.per.broker.percentage = 10

listener.security.protocol.map = PLAINTEXT:PLAINTEXT,SSL:SSL,SASL\_PLAINTEXT:SASL\_PLAINTEXT,SASL\_SSL:SASL\_SSL

listeners = null

log.cleaner.backoff.ms = 15000

log.cleaner.dedupe.buffer.size = 134217728

log.cleaner.delete.retention.ms = 86400000

log.cleaner.enable = true

log.cleaner.io.buffer.load.factor = 0.9

log.cleaner.io.buffer.size = 524288

log.cleaner.io.max.bytes.per.second = 1.7976931348623157E308

log.cleaner.min.cleanable.ratio = 0.5

log.cleaner.min.compaction.lag.ms = 0

log.cleaner.threads = 1

log.cleanup.policy = [delete]

log.dir = /tmp/kafka-logs (如果dirs没设置，就用这个)

log.dirs = D:\kafkaData\kafka\k\log (数据文件目录)

log.flush.interval.messages = 9223372036854775807 (buffer记录数量)

log.flush.interval.ms = null (buffer寿命，没设置计用scheduler.interval.ms)

log.flush.offset.checkpoint.interval.ms = 60000 (偏移量更新频率)

log.flush.scheduler.interval.ms = 9223372036854775807 (定时检查日志buffer)

log.flush.start.offset.checkpoint.interval.ms = 60000 (起始偏移量更新频率)

log.index.interval.bytes = 4096

log.index.size.max.bytes = 10485760

log.message.downconversion.enable = true

log.message.format.version = 2.2-IV1

log.message.timestamp.difference.max.ms = 9223372036854775807

log.message.timestamp.type = CreateTime

log.preallocate = false

log.retention.bytes = -1 (默认不管大小，配置后超多的消息记录干掉)

log.retention.check.interval.ms = 300000

log.retention.hours = 168 (ms,min,hours级别依次下降，超时日志记录干掉)

log.retention.minutes = null

log.retention.ms = null

log.roll.hours = 168 (日志段超时，新建段，无视日志段大小)

log.roll.jitter.hours = 0

log.roll.jitter.ms = null

log.roll.ms = null

log.segment.bytes = 1073741824 (日志段大小)

log.segment.delete.delay.ms = 60000 (删除延时)

max.connections.per.ip = 2147483647 (socket数量)

max.connections.per.ip.overrides = (争对特定IP指定 socket数量)

max.incremental.fetch.session.cache.slots = 1000

message.max.bytes = 1000012 (最大消息大小)

metric.reporters = []

metrics.num.samples = 2

metrics.recording.level = INFO

metrics.sample.window.ms = 30000

min.insync.replicas = 1 (producer端 ack = -1时有效，副本数少于这个值时拒绝写入)

num.io.threads = 8 (children线程数量)

num.network.threads = 3 (parent线程数量,netty)

num.partitions = 2 (topic默认partition数量)

num.recovery.threads.per.data.dir = 1

num.replica.alter.log.dirs.threads = null

num.replica.fetchers = 1

offset.metadata.max.bytes = 4096

offsets.commit.required.acks = -1

offsets.commit.timeout.ms = 5000

offsets.load.buffer.size = 5242880

offsets.retention.check.interval.ms = 600000

offsets.retention.minutes = 10080

offsets.topic.compression.codec = 0

offsets.topic.num.partitions = 50

offsets.topic.replication.factor = 1

offsets.topic.segment.bytes = 104857600

password.encoder.cipher.algorithm = AES/CBC/PKCS5Padding

password.encoder.iterations = 4096

password.encoder.key.length = 128

password.encoder.keyfactory.algorithm = null (密钥算法)

password.encoder.old.secret = null (改密钥时对比)

password.encoder.secret = null (编码密钥)

port = 9092

principal.builder.class = null

producer.purgatory.purge.interval.requests = 1000

queued.max.request.bytes = -1

queued.max.requests = 500

quota.consumer.default = 9223372036854775807

quota.producer.default = 9223372036854775807

quota.window.num = 11

quota.window.size.seconds = 1

replica.fetch.backoff.ms = 1000

replica.fetch.max.bytes = 1048576

replica.fetch.min.bytes = 1

replica.fetch.response.max.bytes = 10485760

replica.fetch.wait.max.ms = 500

replica.high.watermark.checkpoint.interval.ms = 5000

replica.lag.time.max.ms = 10000

replica.socket.receive.buffer.bytes = 65536

replica.socket.timeout.ms = 30000

replication.quota.window.num = 11

replication.quota.window.size.seconds = 1

request.timeout.ms = 30000

reserved.broker.max.id = 1000

sasl.client.callback.handler.class = null

sasl.enabled.mechanisms = [GSSAPI]

sasl.jaas.config = null

sasl.kerberos.kinit.cmd = /usr/bin/kinit

sasl.kerberos.min.time.before.relogin = 60000

sasl.kerberos.principal.to.local.rules = [DEFAULT]

sasl.kerberos.service.name = null

sasl.kerberos.ticket.renew.jitter = 0.05

sasl.kerberos.ticket.renew.window.factor = 0.8

sasl.login.callback.handler.class = null

sasl.login.class = null

sasl.login.refresh.buffer.seconds = 300

sasl.login.refresh.min.period.seconds = 60

sasl.login.refresh.window.factor = 0.8

sasl.login.refresh.window.jitter = 0.05

sasl.mechanism.inter.broker.protocol = GSSAPI

sasl.server.callback.handler.class = null

security.inter.broker.protocol = PLAINTEXT

socket.receive.buffer.bytes = 102400

socket.request.max.bytes = 104857600

socket.send.buffer.bytes = 102400

ssl.cipher.suites = []

ssl.client.auth = none

ssl.enabled.protocols = [TLSv1.2, TLSv1.1, TLSv1]

ssl.endpoint.identification.algorithm = https

ssl.key.password = null

ssl.keymanager.algorithm = SunX509

ssl.keystore.location = null

ssl.keystore.password = null

ssl.keystore.type = JKS

ssl.principal.mapping.rules = [DEFAULT]

ssl.protocol = TLS

ssl.provider = null

ssl.secure.random.implementation = null

ssl.trustmanager.algorithm = PKIX

ssl.truststore.location = null

ssl.truststore.password = null

ssl.truststore.type = JKS

transaction.abort.timed.out.transaction.cleanup.interval.ms = 60000

transaction.max.timeout.ms = 900000

transaction.remove.expired.transaction.cleanup.interval.ms = 3600000

transaction.state.log.load.buffer.size = 5242880

transaction.state.log.min.isr = 1

transaction.state.log.num.partitions = 50

transaction.state.log.replication.factor = 1

transaction.state.log.segment.bytes = 104857600

transactional.id.expiration.ms = 604800000

unclean.leader.election.enable = false (默认值false，ISR为空停止选举leader)

zookeeper.connect = localhost:2181

zookeeper.connection.timeout.ms = 6000

zookeeper.max.in.flight.requests = 10

zookeeper.session.timeout.ms = 6000

zookeeper.set.acl = false

zookeeper.sync.time.ms = 2000 (zookeeper距离)

## 启动服务端

Kafka每一个服务端实例，都叫一个 broker

### kafka 单机9092

bin\windows\kafka-server-start.bat config\server.properties

### kafka 集群9093，9094，9095

配置文件要改三个地方：

broker.id=2

listeners=PLAINTEXT://:9094

log.dirs=D:\\kafkaData\\kafka\\k2\\log

bin\windows\kafka-server-start.bat config\server1.properties

bin\windows\kafka-server-start.bat config\server2.properties

bin\windows\kafka-server-start.bat config\server3.properties

## topic管理

### 创建topic

bin\windows\kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic test

创建名为 test 的 topic，只有 1 个副本 (replication-factor=1)，就是没副本的意思，只有一个分区(patriitons=1)也可以用下面这个命令

bin\windows\kafka-topics.bat --create --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test

### 查 topic 列表

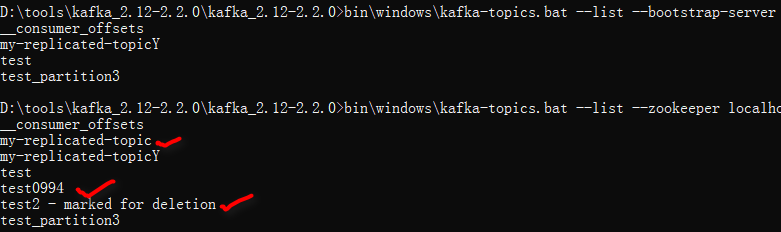
#### broker相关 topic列表

bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092

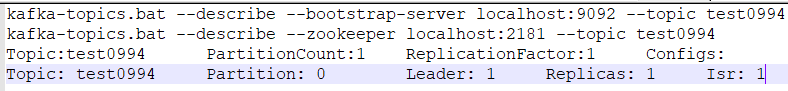
#### 全部topic

全部topic要通过 zookeeper才能查到，连删除了的 topic都能查到， 名为 topic2 的 topic 被标记为 marked for deletion也被查了出来

bin\windows\kafka-topics.bat --list --zookeeper localhost:2181



#### 查 topic详细信息



由于 localhost:9092这个 Broker 没有存放 test0094 这个 topic 的 Partition，所以没法通过它查到 topic 的具体信息，但是可以通过 zookeeper 查出来。

#### 修改topic

增加topic 的 Partition 数量，不能减少 Partition 的数量

bin\windows\kafka-topics.bat --zookeeper localhost:2181 --alter --topic test\_partition3 -partitions 4

如果 Broker 上有该 topic ，也可以用 –bootstrap-server 参数

### 增加replica数

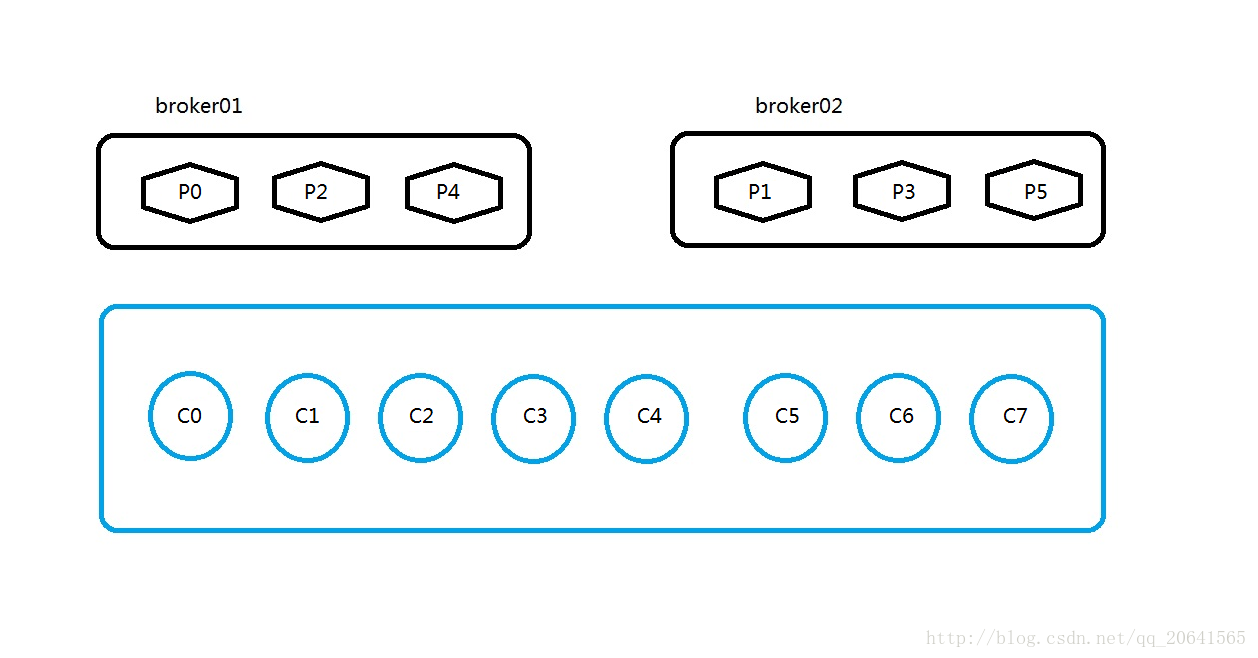
{"version":1,"partitions":[{"topic":"my-replicated-topicY","partition":0,"replicas":[3,0]},{"topic":"my-replicated-topicY","partition":1,"replicas":[0,3]}]}

刚开始Partition =2， replication-factory=1， Partition-0、Partition-1分别在 Broker-3、和 Broker-0 上，通过

bin\windows\kafka-reassign-partitions.bat --zookeeper localhost:2181 --reassignment-json-file replica.json –execute可以令replication-factory=2

## Consumer Group管理

### Consumer均衡算法



算法如下：

1.A=(partition数量/同分组消费者总个数)

2.M=对上面所得到的A值小数点第一位向上取整

3.计算出该消费者拉取数据的patition合集：Ci = [P(M\*i ),P((i + 1) \* M -1)]

按照如图所示，那么这里：

A=6/8=0.75

M=1

C0=[P(1\*0),P((0+1)\*1-1)]=[P0,P0]

C1=[P(1\*1),P((1+1)\*1-1)]=[P1,P1]

C2=[P(1\*2),P((2+1)\*1-1)]=[P2,P2]

C3=[P(1\*3),P((3+1)\*1-1)]=[P3,P3]

C4=[P(1\*4),P((4+1)\*1-1)]=[P4,P4]

C5=[P(1\*5),P((5+1)\*1-1)]=[P5,P5]

C6=[P(1\*6),P((6+1)\*1-1)]=[P6,P6]

C7=[P(1\*7),P((7+1)\*1-1)]=[P7,P7]

那么按照上面的算法：

C0消费者消费P0的数据

C1消费者消费P1的数据

C2消费者消费P2的数据

C3消费者消费P3的数据

C4消费者消费P4的数据

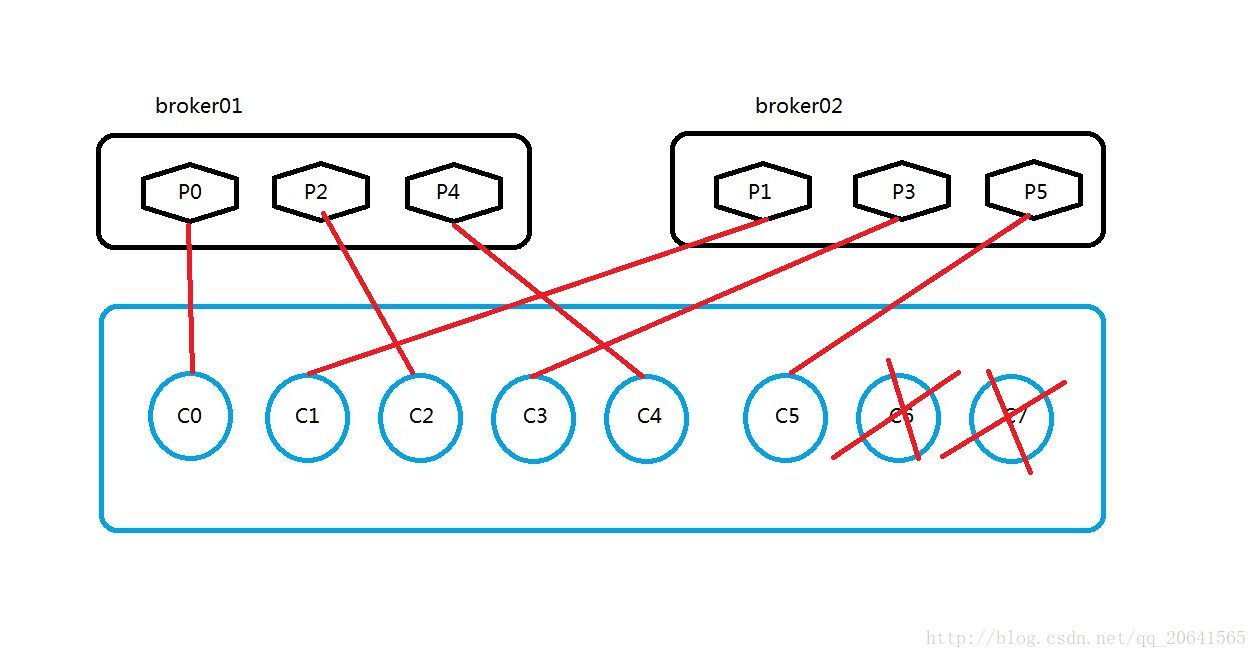
C5消费者消费P5的数据

C6消费者消费P6的数据

C7消费者消费P7的数据

但是partition只有P0-P5根本就没有P6和P7，所以这两个消费者相当于是会被闲置的，就相当于占用资源，却没什么用，所以在这里真正起到作用的就是C0-C5。

如下图所示：



如果这个消费组里面的消费者少于partition数量呢（比如5个）？

那么还是依葫芦画瓢，根据上面的算法：

A=6/5=1.2

M=2

C0=[P(2\*0),P((0+1)\*2-1)]=[P0,P1]

C1=[P(2\*1),P((1+1)\*2-1)]=[P2,P3]

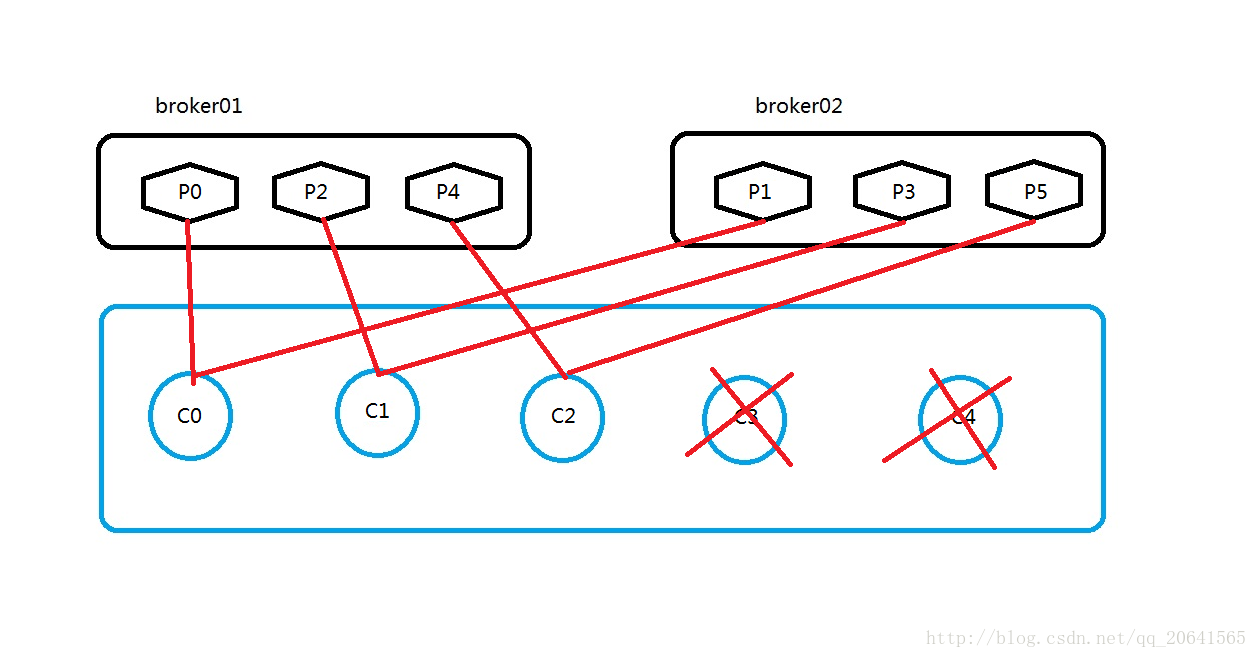
C2=[P(2\*2),P((2+1)\*2-1)]=[P4,P5]

C3=[P(2\*3),P((3+1)\*2-1)]=[P6,P7]

C4=[P(2\*4),P((4+1)\*2-1)]=[P8,P9]

同上面一样C3和C4没有起到任何作用。

如下所示：



总结：

先按照Partition 是 consumer 的倍数分段，然后再按段分配

1.按照如上的算法，所以如果kafka的消费组需要增加组员，最多增加到和partition数量一致，超过的组员只会占用资源，而不起作用；

2.kafka的partition的个数一定要大于消费组组员的个数，并且partition的个数对于消费组组员取模一定要为0，不然有些消费者会占用资源却不起作用；

3.如果需要增加消费组的组员个数，那么也需要根据上面的算法，调整partition的个数

### 查 group 表

查 Broker 相关消费组列表

bin\windows\kafka-consumer-groups.bat --bootstrap-server localhost:9092 –list

### 查 group偏移量

bin\windows\kafka-consumer-groups.bat --bootstrap-server localhost:9092 --describe --group console-consumer-32099

这里是查了console-consumer-32099 消费组 Partition 偏移量

### 指定consumer的group

指定group，也可以指定偏移量

bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test\_partition3 --consumer-property group.id=test-group2

为 test\_partition3 这个主题，添加了一个消费者，它的 consumer group 是 test-group2。这个消费者的偏移量是 topic 的 Partion 的当前偏移量(最后的位置)

### 修改消费者组的Partition偏移量

bin\windows\kafka-consumer-groups.bat --bootstrap-server localhost:9092 --group test-group1 --reset-offsets --all-topics --to-earliest –execute

--reset-offsets：这个参数表示要修改 group 的偏移量

-all-topics：表示要修改全部 topic 的偏移量

--to-earliest： 这是新的偏移量的位置

--execute：表示立即生效。

### consumer和producer压力测试

1.生产者测试

kafka-producer-perf-test.bat --num-records 1000000 --topic test --record-size 200 --throughput 100000 --producer-props bootstrap.servers=localhost:9092

--num-records 发送消息的数量

--topic 主题

-- record-size 单条消息的大小字节

--throughput 吞储量阀值 10万 就是每秒不超过10万条数据

--producer-props producer的配置，可以写多个配置用逗号隔开

2.消费者测试

kafka-consumer-perf-test.bat --messages 1000000 --threads 1 --zookeeper localhost:2181 --num-fetch-threads 3 --topic test

--messages 消费多少消息

--threads 线程数量

--zookeeper zookeeper的地址

--num-fetch-threads 拉取数据的线程数量 即为消费者的数量

测试参数

./kafka-producer-perf-test.sh --num-records 10000000000 --topic test-rep-one --record-size 200 --throughput 1000000000 --producer-props bootstrap.servers=hadoop.23.51fbops.com:9092,hadoop.22.51fbops.com:9092,hadoop.24.51fbops.com:9092

./kafka-consumer-perf-test.sh --messages 100000000 --threads 3 --zookeeper hadoop.22.51fbops.com:2181,hadoop.23.51fbops.com:2181,hadoop.24.51fbops.com:2181 --num-fetch-threads 3 --topic test-rep-one

生产者测试结果 16.39Mb/秒流量，160万/秒消息，无堆积

8274379 records sent, 1654875.8 records/sec (15.78 MB/sec), 0.7 ms avg latency, 14.0 max latency.

8410914 records sent, 1682182.8 records/sec (16.04 MB/sec), 0.7 ms avg latency, 11.0 max latency.

8385566 records sent, 1677113.2 records/sec (15.99 MB/sec), 0.8 ms avg latency, 9.0 max latency.

8293790 records sent, 1658758.0 records/sec (15.82 MB/sec), 0.7 ms avg latency, 11.0 max latency.

7911355 records sent, 1582271.0 records/sec (15.09 MB/sec), 0.8 ms avg latency, 9.0 max latency.

8268134 records sent, 1653626.8 records/sec (15.77 MB/sec), 0.7 ms avg latency, 13.0 max latency.

8198960 records sent, 1639792.0 records/sec (15.64 MB/sec), 0.8 ms avg latency, 15.0 max latency.

8349350 records sent, 1669870.0 records/sec (15.93 MB/sec), 0.7 ms avg latency, 12.0 max latency.

8140649 records sent, 1628129.8 records/sec (15.53 MB/sec), 0.8 ms avg latency, 12.0 max latency.

8592471 records sent, 1718494.2 records/sec (16.39 MB/sec), 0.7 ms avg latency, 9.0 max latency.

8183905 records sent, 1636781.0 records/sec (15.61 MB/sec), 0.8 ms avg latency, 7.0 max latency.

消费者

[root@hadoop.22.51fbops.com bin]# ./kafka-consumer-perf-test.sh --messages 100000 --threads 3 --zookeeper hadoop.22.51fbops.com:2181,hadoop.23.51fbops.com:2181,hadoop.24.51fbops.com:2181 --num-fetch-threads 3 --topic test-rep-one

start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg, nMsg.sec

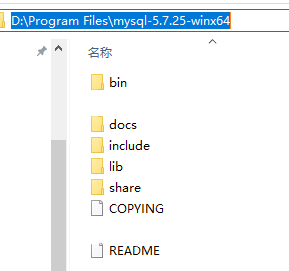
2017-12-29 10:52:59:047, 2017-12-29 10:53:10:508, 286.1023, 24.9631, 30000000, 2617572.6376

目前业务 需求为9000条/秒。

# MYSQL

## 安装

1. 下载免安装包，解压到**D:\Program Files\mysql-5.7.25-winx64** 下，如图



1. 创建配置文件

Windows环境：my.ini Linux环境：my.cnf 内容如下：

**[mysql]**

# 设置mysql客户端默认字符集

default-character-set=utf8

**[mysqld]**

#设置3306端口

port = 3306

# 设置mysql的安装目录

**basedir=D:\\Program Files\\mysql-5.7.25-winx64**

# 设置mysql数据库的数据的存放目录

**datadir=D:\\Program Files\\mysql-5.7.25-winx64\\data**

# 允许最大连接数

max\_connections=200

# 服务端使用的字符集默认为8比特编码的latin1字符集

character-set-server=utf8

# 创建新表时将使用的默认存储引擎

default-storage-engine=INNODB

1. 初始化mysql

初始化mysql 实际就是初始化数据，根据my.ini配置文件，生成数据目录**D:\Program Files\mysql-5.7.25-winx64\data**

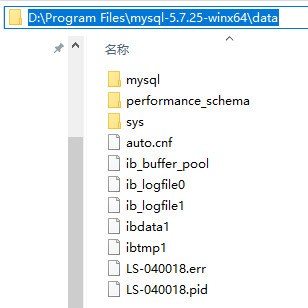
mysqld --defaults-file="D:\Program Files\mysql-5.7.25-winx64\my2.ini" --initialize –console

这里有两个参数可选 [--initialize](https://dev.mysql.com/doc/refman/5.7/en/server-options.html#option_mysqld_initialize) 和 [--initialize-insecure](https://dev.mysql.com/doc/refman/5.7/en/server-options.html#option_mysqld_initialize-insecure) ，使用[--initialize](https://dev.mysql.com/doc/refman/5.7/en/server-options.html#option_mysqld_initialize) 参数系统为 root 用户生成随机密码，这个密码要记住。[--initialize-insecure](https://dev.mysql.com/doc/refman/5.7/en/server-options.html#option_mysqld_initialize-insecure)  参数root 账户没有密码，登陆后使用

ALTER USER 'root'@'localhost' IDENTIFIED BY 'root';

将 root 账户的密码设置为 root。

如下：



1. 安装成 windows 系统服务

mysqld --install *myServiceName* --defaults-file= D:\Program Files\mysql-5.7.25-winx64\my.ini

这里的myServiceName 自己定，它是服务的名字，不是必须的参数 --defaults-file 这个是配置文件的位置

1. 启动/关闭mysql

mysqld –console 直接启动mysql.exe

net start *myServiceName*

net stop *myServiceName*

## 系统库介绍

information\_schema

mysql

performance\_schema

sys

## 语法

### 参数

[IN | OUT | INOUT] param\_name type

**Eg:**

create procedure test\_pro(IN name char(2), OUT age int, INOUT city varchar(255))

### 变量

declare语句的位置，不能在set语句的位置之后，否则报错。

declare param\_name varchar(30) default 100;

declare var1, var2 int;

set var1 = 10;

set var2 = var1;

### condition

condition 是 sql 运行过程中出现的 error 和 exception 的别名。

### handler

是 condition 处理程序，声明 handler时可以不使用 condition 而直接处理 exception 或者 error：

declare exit handler for sqlstate 42000 set @temp = 1;

### cursor

声明游标的declare 语句，必须在声明 handler 的语句之前，否则报错。必须在申明变量的语句之后，否则报错

1. **drop** **procedure** if exists deleteResource;
2. **create** **procedure** deleteResource(IN resourceName **varchar**(64))
3. **begin**
4. **declare** t\_id **varchar**(64);
5. **declare** c\_status **int** **default** 0;
7. **declare** c\_id **cursor** **for** **select** id
8. **from** ctg\_stable\_resource
9. **where** resource\_name = resourceName;
10. **declare** **continue** handler **for** not found **set** c\_status = 1;
12. **open** c\_id;
13. id\_loop: loop
14. **fetch** c\_id **into** t\_id;
15. if c\_status = 1
16. **then** **select** 'error';
17. leave id\_loop; **end** if;
18. **select** \*  **from** ctg\_stable\_resource  **where** id = t\_id;
19. **end** loop id\_loop;
20. **close** c\_id;
21. **end**;

### 索引

普通索引可以有空值，重复值；唯一索引可以有空值，值不能重复；全文索引可以有空值、重复值支持值的全文检索，只能在 CHAR, VARCHAR, TEXT列上建，且只有MyISAM引擎支持；空间索引

#### 创建索引

index(column\_name) 匿名普通索引

index indexName(column\_1, column\_2, ...) 具名普通索引，多列

unique index indexName(column\_1, column\_2, ...)具名唯一索引，多列

fulltext index indexName(column\_1, column\_2, ...)具名全文索引，多列

alter table add [index, unique index, fulltext index] indexName(column\_1, column\_2, ...) 增加索引

create [index, unique index, fulltext index] indexName on tableName(column\_1, column\_2, ...) 增加索引

### delimiter

cmd命令行中，完整命令的结束标志为分号 "**;**"。这与存储过程中sql语句结束符是一样的。为了避免冲突，要修改命令结束标识符。在脚本中不存在这种情况。

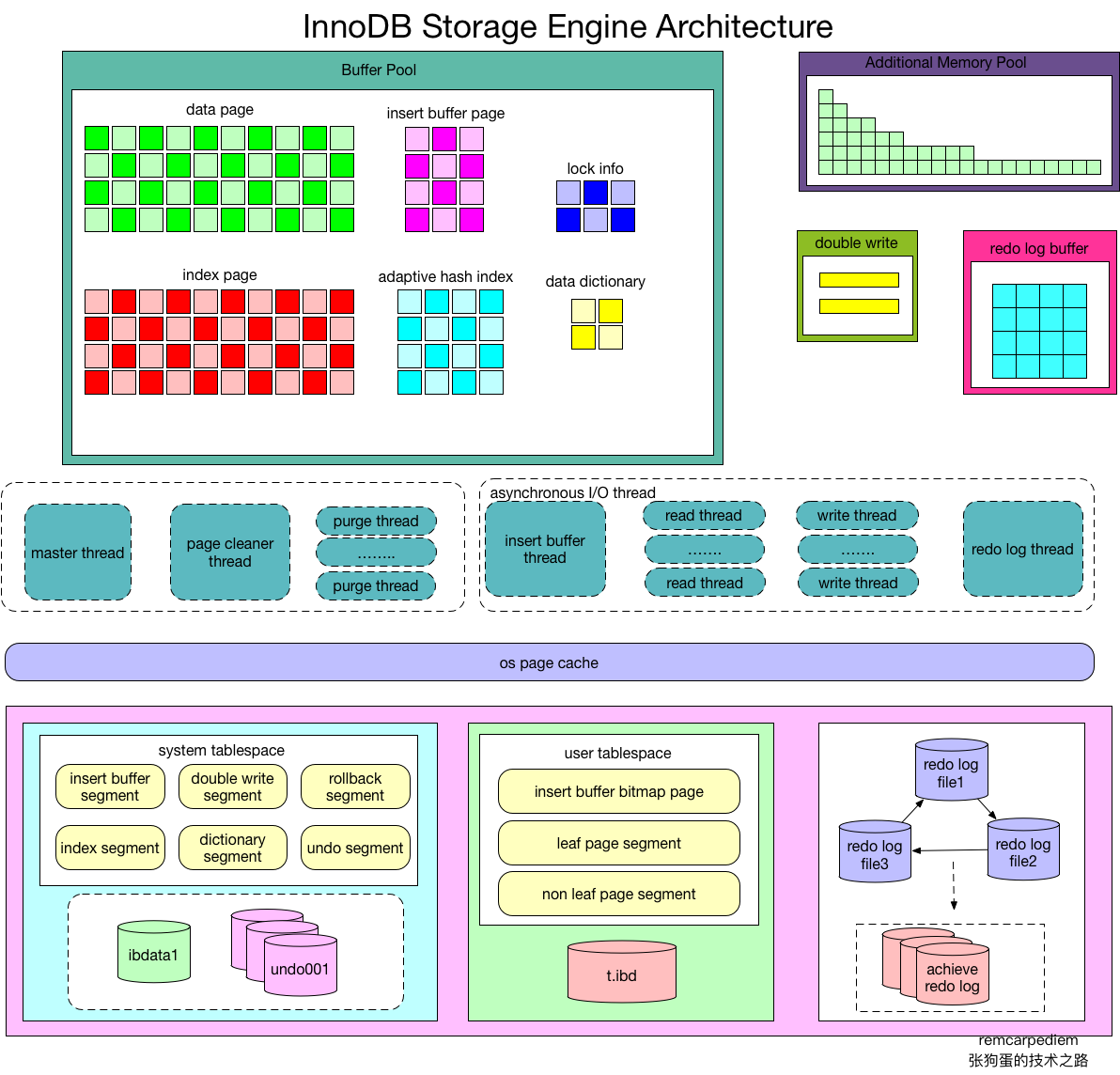
1. **mysql**> delimiter $$
2. **mysql**> **create** **procedure** Proc()
3. -> **begin**
4. -> **select** now();
5. -> **end** $$
6. **mysql**> delimiter ;

上面一共有 3 条mysql.exe 的命令。 第 1行将命令的结束标志设置为 $$，第 5 行使用了这个命令结束标志。第 4 行是正常的 sql 语句，不是 mysql.exe 的命令。第 6 行将命令的结束标志又换回来。

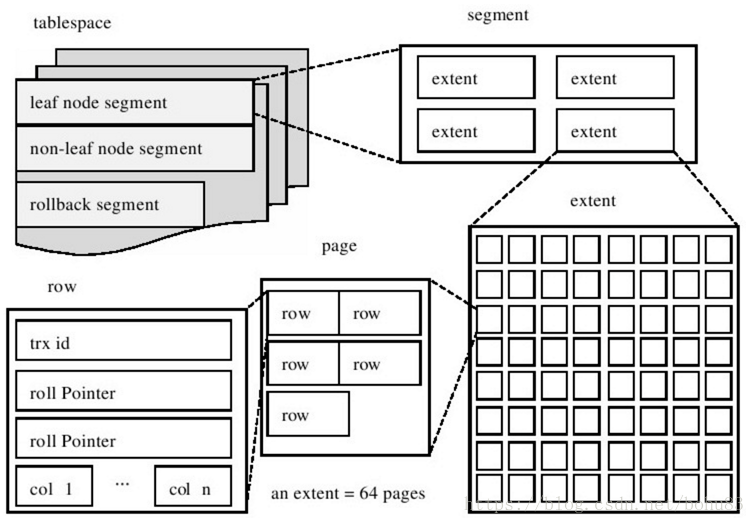
## 原理

### Innodb构架

old 加入 new 时，page made yound。 没有移动的 page not made yound。



### Innodb存储结构



### Checkpoints

#### Redo log buffer 冲刷时机

* Master每秒刷一次
* 事务提交
* Redo log pool 剩余空间小于1/2

#### 脏页冲刷

* Master thread checkout：master按时冲刷
* Flush-lru-list checkpoint：保证空闲页数量
* Async/symc flush checkpoint：重做日志不够用
* Dirty page too much checkpoint：脏页太多

## 工具使用

### Mysqldump

* 导出所有数据库

mysqldump -uroot -proot --all-databases > /tmp/all.sql

* 导出db1、db2两个数据库的所有数据

mysqldump -uroot -proot --databases db1 db2 >/tmp/user.sql

* 导出db1中的a1、a2表

mysqldump -uroot -proot --databases db1 --tables a1 a2 >/tmp/db1.sql

* 条件导出，导出db1表a1中id=1的数据

mysqldump -uroot -proot --databases db1 --tables a1 --where='id=1' >/tmp/a1.sql

* 只导出表结构不导出数据，--no-data

mysqldump -uroot -proot --no-data --databases db1 >/tmp/db1.sql

* 跨服务器导出导入数据

mysqldump --host=h1 -uroot -proot --databases db1 |mysql --host=h2 -uroot -proot db2

* 导入

mysql -uroot -p123456 <f:\all.sql

## 运维

### 集群

#### NDB network database

