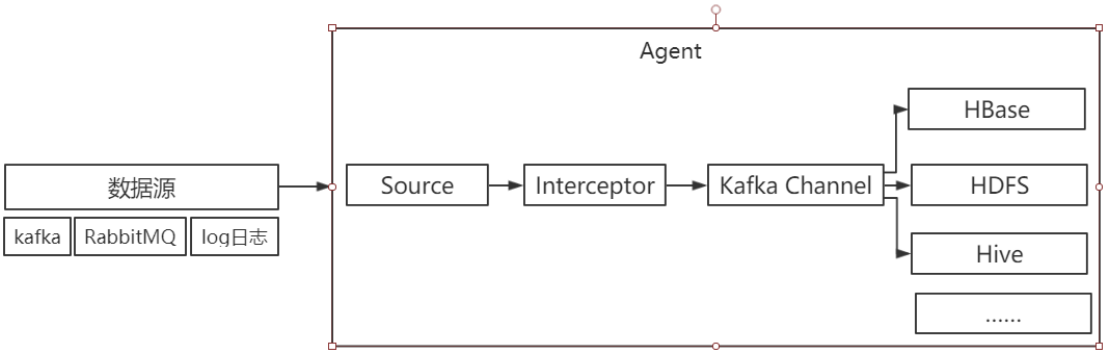
面试简述基于 Flume 数据采集流程



二次开发 flume 各个组件(因为 flume 默认的一些功能还不能满足开发需求, 如需要在 flume 拦截器中处理中文乱码问题，更加灵活的 Sink)

启动命令为 nohup bin/flume-ng agent -n a1 -c conf -f conf/flume-conf.properties -Dflume.monitoring.type=http -Dflume.monitoring.port=41414 &

启动时监听 41414 端口，后台 zabbix 对该端口进行监控，有异常邮件告警
curl localhost:41414/metrics | grep ChannelSize 可以查看各个 Channel 之间的数据积压情况

Flume 开发示例代码:

1.使用官方的组件， 搭配一个 从 netcat source -> file channel -> logger sink 的 demo

```
# example.conf: A single-node Flume configuration

    # Name the components on this agent
    a1.sources = r1
    a1.sinks = k1
    a1.channels = c1

    # Describe/configure the source
    a1.sources.r1.type = netcat
    a1.sources.r1.bind = localhost
    a1.sources.r1.port = 44444

    # Describe the sink
    a1.sinks.k1.type = logger

    # Use a channel which buffers events in memory
    a1.channels.c1.type = memory
```

```
    a1.channels.c1.capacity = 1000
    a1.channels.c1.transactionCapacity = 100

    # Bind the source and sink to the channel
    a1.sources.r1.channels = c1
    a1.sinks.k1.channel = c1
```

输出结果：

```
aa
OK


2018-11-13 14:14:56,786 (lifecycleSupervisor-1-2) [INFO -
org.apache.flume.source.NetcatSource.start(NetcatSource.java:164)]
Created
serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:44444]
2018-11-13 14:15:34,795 (SinkRunner-PollingRunner-DefaultSinkProcessor)
[INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:70)]
Event: { headers:{}
body: 61 61 61 0D                                       aaa. }
```

2. 编写自定义 source ， 抓取模拟数据 经过 channel -> logger sink

```
package flume.plugin;

import java.nio.charset.Charset;


public class CustomSource extends AbstractSource implements Configurabl
e,PollableSource{

    @Override
    public Status process() throws EventDeliveryException {

        Random random = new Random();
        int randomNum = random.nextInt(100);
        String text = "Hello world" + random.nextInt(100);
        HashMap<String, String> header = new HashMap<String,String>();
```

```
        header.put("id",Integer.toString(randomNum));
        //模拟数据

        this.getChannelProcessor()
            .processEvent(EventBuilder.withBody(text,Charset.forName("U
TF-8"),header));
        return Status.READY;
    }

    @Override
    public void configure(Context context) {


    }
}
```

配置文件如下

```
a1.sources = r1
a1.sinks = k1
a1.channels = c1
#
# # Describe/configure the source
a1.sources.r1.type = flume.plugin.CustomSource
#
# # Describe the sink
a1.sinks.k1.type = logger
#
# # Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
#
# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

3. 编写自定义 的 拦截器， 过滤 source 接收的数据

3.1 过滤包含 XXX 的数据

```
public class CustomInterceptor implements Interceptor {
```

```
        @Override
        public void initialize() {
            // TODO Auto-generated method stub
        }

        @Override
        public Event intercept(Event event) {
            if(new String(event.getBody()).contains("xxx")){
                return null;
            };
            return event;
        }

        @Override
        public List<Event> intercept(List<Event> events) {
            for(Event e:events) {
                intercept(e);
            }
            return events;
        }
        @Override
        public void close() {
        }


    }
```

Builder 拦截器创建类

```
public class CustomInterceptorBuilder implements Builder{

        @Override
        public void configure(Context context) {
            // TODO Auto-generated method stub

        }

        @Override
        public Interceptor build() {
            // TODO Auto-generated method stub
            return new CustomInterceptor();
        }

    }
```

配置文件如下

```
a1.sources = r1
```

```
a1.sinks = s1
a1.channels = c1

a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type =
flume.plugin.CustomInterceptorBuilder
#flume.plugin.CustomInterceptor$CustomInterceptorBuilder
a1.sources.r1.interceptors.i1.perserveExisting = true

a1.sinks.s1.type = logger

a1.channels.c1.type = memory
a1.channels.c1.capacity = 2
a1.channels.c1.transactionCapacity = 2

a1.sources.r1.channels = c1
a1.sinks.s1.channel = c1
```

自定义 header

```
public class CustomCountInterceptor implements Interceptor{

    private final String headerKey;
    private static final String CONF_HEADER_KEY = "header";
    private static final String DEFAULT_HEADER = "count";
    private final AtomicLong currentCount;

    public CustomCountInterceptor(Context ctx) {
        headerKey = ctx.getString(CONF_HEADER_KEY,DEFAULT_HEADER);
        currentCount = new AtomicLong();
    }


    @Override
    public void initialize() {
        // TODO Auto-generated method stub
    }

    @Override
    public Event intercept(Event event) {

        long count = currentCount.incrementAndGet();
```

```
        event.getHeaders().put(headerKey, String.valueOf(count));
        return event;
    }

    @Override
    public List<Event> intercept(List<Event> events) {
        for(Event e:events) {
            intercept(e);
        }
        return events;
    }
    @Override
    public void close() {
    }
    public static class CustomInterceptorBuilder implements Builder {
        private Context ctx;

        @Override
        public Interceptor build() {
            return new CustomCountInterceptor(ctx);
        }

        @Override
        public void configure(Context context) {
            this.ctx = context;
        }
    }
}
```

4. 编写自定义 sink ， 将数据接入 mysql 数据库中

```
public class CustomSink extends AbstractSink implements Configurable{

    private Connection connect;
    private Statement stmt;
    private String columnName;
    private String url;
    private String user;
    private String password;
    private String tableName;

    @Override
    public synchronized void start() {
        try {
```

```
            Class.forName("com.mysql.jdbc.Driver");
            connect = DriverManager.getConnection(url,user,password);
            stmt = connect.createStatement();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public Status process() throws EventDeliveryException {

        Channel ch = getChannel();
        Transaction txn = ch.getTransaction();
        Event event = null;
        txn.begin();
        try {
            while (true) {
                event = ch.take();
                if (event != null) {
                    break;
                }
            }

            String rawbody = new String(event.getBody());


            //insert into t1(content) values("zhangsan");
            String sql = "insert into" + tableName +"(" + columnName
+ ")"+"values("+rawbody+")";

            stmt.execute(sql);
            txn.commit();
            return Status.READY;
        } catch (Exception e) {
            txn.rollback();
            e.printStackTrace();
            return null;
        }finally {
            txn.close();
        }
    }

    @Override
    public void configure(Context context) {
```

```
        url = "jdbc:mysql://localhost:3306/test";
        user = "root";
        password = "123456";
        tableName = "test";
        columnName = "content";
    }

}
```

配置文件如下:

```
a1.sources = r1
a1.sinks = k1
a1.channels = c1
#
# # Describe/configure the source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444
#
# # Describe the sink
a1.sinks.k1.type = flume.plugin.CustomSink
#
# # Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
#
# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

改进：

参数不写死，而是写在配置文件里，如：

```
a1.sinks.k1.type = flume.plugin.CustomSink
a1.sinks.k1.url =jdbc:mysql://127.0.0.1:3306/test
a1.sinks.k1.tableName= test
a1.sinks.k1.user=root
a1.sinks.k1.password=123456
a1.sinks.k1.column_name=content
```

```
//从配置文件中读取各种属性，并进行一些非空验证
public void configure(Context context) {
    columnName = context.getString("column_name");
```

```
    Preconditions.checkNotNull(columnName, "column_name must be
set!!");
    url = context.getString("url");
    Preconditions.checkNotNull(url, "url must be set!!");
    user = context.getString("user");
    Preconditions.checkNotNull(user, "user must be set!!");
    password = context.getString("password");
    Preconditions.checkNotNull(password, "password must be set!!");
    tableName = context.getString("tableName");
    Preconditions.checkNotNull(tableName, "tableName must be set!!");
}
```

5. 配置 flume， 尝试 2 或者多个 source， 写入多个 channel， 然后多个 sink

```
a1.sources = r1 r2
a1.sinks = k1 k2
a1.channels = c1 c2

# Describe/configure the source
a1.sources.r1.type = exec
a1.sources.r1.shell = /bin/bash -c
a1.sources.r1.channels = c1 c2
a1.sources.r1.command = tail -F /opt/apps/logs/tail1.log
a1.sources.r1.selector.type=replicating

a1.sources.r2.type = exec
a1.sources.r2.shell = /bin/bash -c
a1.sources.r2.channels = c1 c2
a1.sources.r2.command = tail -F /opt/apps/logs/tail2.log
a1.sources.r2.selector.type=replicating


# channel1
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

#channel2
a1.channels.c2.type = file
a1.channels.c2.checkpointDir = /opt/apps/flume-1.7.0/checkpoint
a1.channels.c2.dataDirs = /opt/apps/flume-1.7.0/data
```

```
#sink1
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path = hdfs://hadoop101:9000/testout
a1.sinks.k1.hdfs.fileType = DataStream
a1.sinks.k1.hdfs.rollCount = 0
a1.sinks.k1.channel =c2
a1.sinks.k1.sink.rollInterval=0

#sink2
a1.sinks.k2.type = file_roll
a1.sinks.k2.channel =c1
#a1.sinks.k2.sink.rollInterval=0
a1.sinks.k2.sink.directory = /opt/apps/tmp
```

6. 1 source 1channel 多 sink

```
a1.sources = r1
a1.sinks = k2
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = exec
a1.sources.r1.shell = /bin/bash -c
a1.sources.r1.channels = c1
a1.sources.r1.command = tail -F /opt/apps/logs/tail1.log



# channel
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

#sink1
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path = hdfs://hadoop101:9000/testout
a1.sinks.k1.hdfs.fileType = DataStream
a1.sinks.k1.hdfs.rollCount = 0
a1.sinks.k2.sink.rollInterval=0
a1.sinks.k2.channel = c1
```

```
#sink2
a1.sinks.k2.type = file_roll
a1.sinks.k2.channel = c1
a1.sinks.k2.sink.rollInterval=0
a1.sinks.k2.sink.directory = /opt/apps/tmp
```

配置可以从 配置文件中取得：

```
    @Override
     public void configure(Context context) {
         url = "jdbc:mysql://localhost:3306/test";
         user = "root";
         password = "123456";
         tableName = "test";
         columnName = "content";
```

可以写成

```
    @Override
     public void configure(Context context) {

         url = context.getString("url", "默认值");

         user = context.getString("user", "root");

         password = .....
```

在配置文件中设置

a1.sinks.k1.url = jdbc:xxxxxxx

a1.sinks.k1.user = root

a1.sinks.k1.password = xxx

即可取到配置内容。

面试简述数据库采集流程

数据增量采集可以根据数据库更新时间或者是自增 ID 来采集 如果是 MySQL 可以考虑根据 binlog 日志来采集

顺便提一句 由于在杭州面试，大部分阿里系的公司都是采用阿里自己研发的 Maxcomputer，在数据采集上，直接在上面做一些配置上的修改。所以在面试阿里系公司的时候，可以先看看有关的 maxcomputer 数据采集与任务型计算

关于数据仓库分层模型的设计

数据的一些分层思想
Ods 层（原始数据层）：存放着最原始的数据 如：flume 采集过来的数据

Cdm 层（公共数据层）：作为公用的数据 如 不同业务都需要的数据
Cdm 层又分为两部分
Dwd 层：存放着从原始数据加工后的数据（包括一些数据维度的加工，剔除暂时不需要的维度数据）
Dws 层：存放着无关业务的一些聚合类数据
Ads 层（结果数据层）：存放着强业务数据结果的数据 如（每个店铺售卖商品的 TopN）
数据的流转可以是单机也可以是分布式如（MR，Spark，Flink）
由于项目是 T+1 的数据采集处理，因此大部分都是定时任务进行（Linux crontab）

一些面试常问的面试题

HashMap 和 Hashtable 的区别以及 HashMap 的底层实现

这个是问的频率比较多的

1. **线程是否安全：** HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过 `synchronized` 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；
2. **效率：** 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；
3. **对 Null key 和 Null value 的支持：** HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。
4. **初始容量大小和每次扩充容量大小的不同 ：** ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为 11，之后每次扩充，容量变为原来的 2n+1。HashMap 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小（HashMap 中的 `tableSizeFor()`方法保证，下面给出了源代码）。也就是说 HashMap 总是使用 2 的幂作为哈希表的大小,后面会介绍到为什么是 2 的幂次方。
5. **底层数据结构：** JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

HashMap 的底层实现

JDK1.8 之前

JDK1.8 之前 HashMap 底层是 **数组和链表** 结合在一起使用也就是 **链表散列**。

HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过 (n - 1) & hash 判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

JDK 1.8 HashMap 的 hash 方法源码:

JDK 1.8 的 hash 方法 相比于 JDK 1.7 hash 方法更加简化，但是原理不变。

```
    static final int hash(Object key) {
      int h;
      // key.hashCode()：返回散列值也就是 hashcode
      // ^ ：按位异或
      // >>>:无符号右移，忽略符号位，空位都以 0 补齐
      return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
  }
```

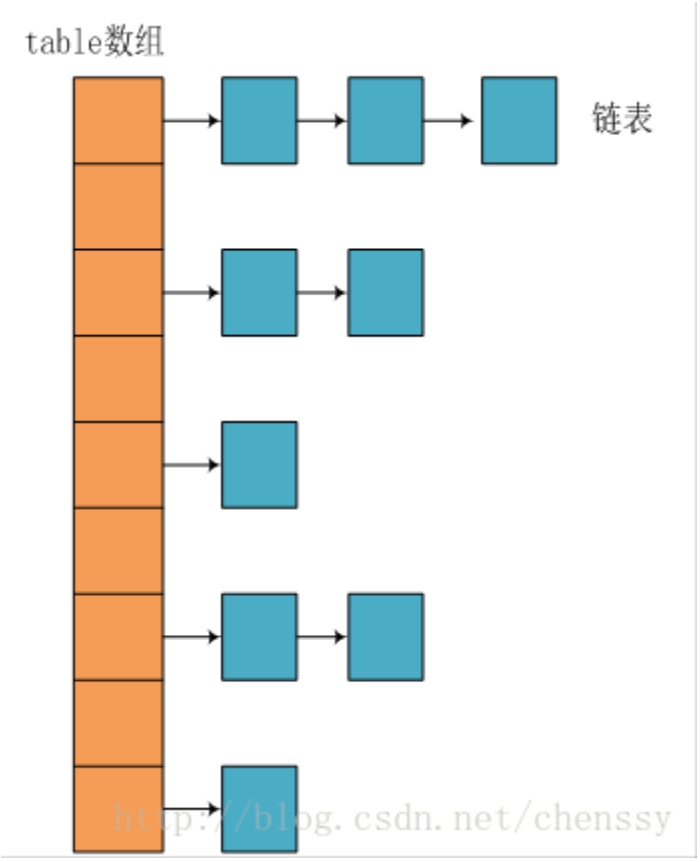
对比一下 JDK1.7 的 HashMap 的 hash 方法源码.

```
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).

    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

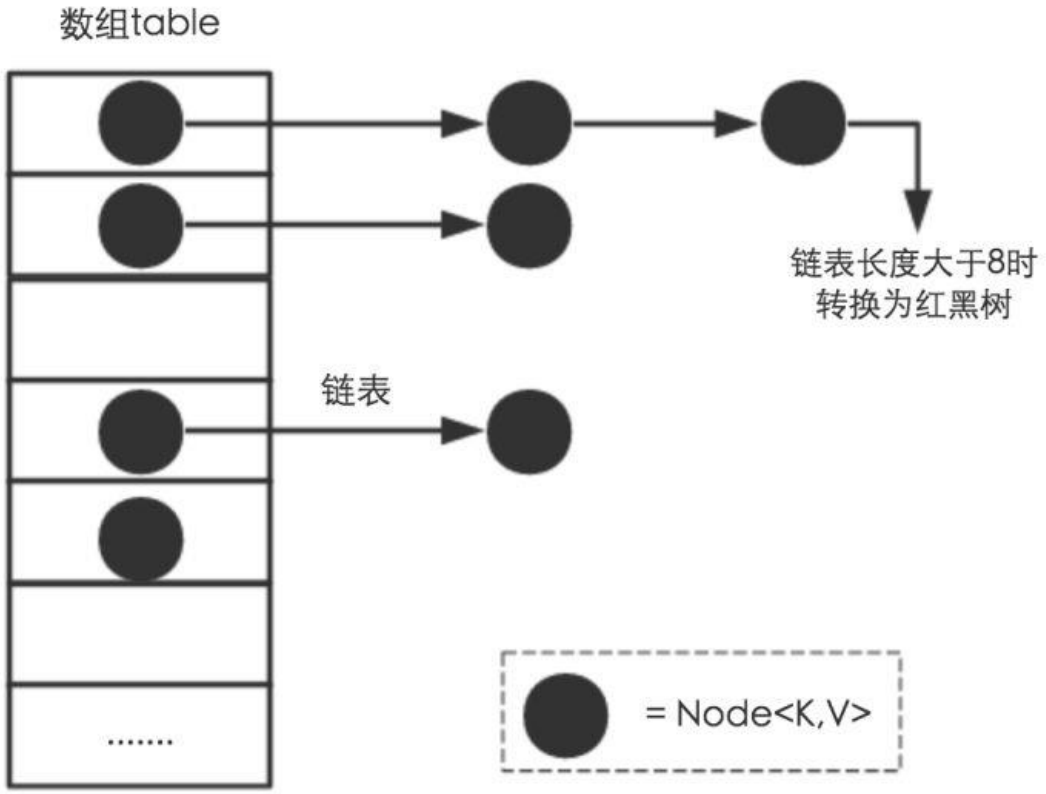
相比于 JDK1.8 的 hash 方法 ，JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次。

所谓**“拉链法”**就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



JDK1.8 之后

相比于之前的版本， JDK1.8 之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间。



TreeMap、TreeSet 以及 JDK1.8 之后的 HashMap 底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

ConcurrentHashMap 和 Hashtable 的区别

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

- **底层数据结构：** JDK1.7 的 ConcurrentHashMap 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟 HashMap1.8 的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数

据结构类似都是采用 **数组+链表** 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；

- **实现线程安全的方式（重要）：** ① **在 JDK1.7 的时候，ConcurrentHashMap（分段锁）** 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 **到了 JDK1.8 的时候已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6 以后 对 synchronized 锁做了很多优化）** 整个看起来就像是优化过且线程安全的 HashMap，虽然在 JDK1.8 中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；

  ② **Hashtable(同一把锁)** :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

HashMap 的长度为什么是 2 的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648 到 2147483647，前后加起来大概 40 亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个 40 亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算

方法是“(n - 1) & hash”。（n 代表数组长度）。这也就解释了 HashMap 的长度为什么是 2 的幂次方。

这个算法应该如何设计呢？

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：**“取余(%)操作中如果除数是 2 的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是 2 的 n 次方；）。”**并且 **采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是 2 的幂次方。**

HBase 默认 MemStore 大小：

hbase.hregion.memstore.flush.size 默认是 128M 的时候,会触发 MemStore 的刷新。

HDFS 读写原理：（这个网上有很多介绍）

Kafka 如何保证消息的高并发写入和读取：刚好前段时间看了相关的文章 可以参考

http://mp.weixin.qq.com/s?__biz=MzU0OTk3ODQ3Ng==&mid=2247484700&idx=1&sn=fbfdb57ea53882828e4e3bd0b3b61947&chksm=fba6ed

1fccd16409c43baa7f941e522d97a72e63e4139f663b327c606c6bb5dfe516b6f61424&scene=21#wechat_redirect