

CSDN

首页

博客

学院

下载

GitChat

TinyMind

...

搜博主文章

Q

写博客

发Chat

登录

注册

Darker

将所有的碎片化时间紧凑好，投入无限的学习中去。

RSS订阅

个人资料

07H\_JH

关注

原创233

粉丝181

喜欢30

评论76

等级：博客5

积分：7470

勋章：恒

访问：51万+

排名：3741

全息沙盘

Github

https://github.com/jinhang

最新文章

一个分布式测试系统利器

Kudu

Spark Streaming和Flink的Word Count对比

Java反射在JVM的实现

快排的思考

博主专栏

spark/hadoop学习

阅读量：9001636篇

java研究者

阅读量：103636106篇

个人分类

ACM笔记36篇

大数据框架106篇

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册

X

https://blog.csdn.net/jianghuxiaojin/article/details/51377600

1/16

机器学习

19篇

展开

归档

2017年8月

1篇

2017年6月

1篇

2017年5月

1篇

2017年2月

7篇

2016年12月

4篇

展开

热门文章

Spark-Spark Streaming例子整理(一)  
阅读量：43216

阿里面试题——天猫部  
阅读量：13893

服务器上的Mysql表全丢了情况下恢复数据  
阅读量：11212

华为软件精英挑战赛2016题解  
阅读量：9649

java 泛型中 T 和 问号（通配符）的区别  
阅读量：7639

最新评论

Spark-Spark Strea...  
yulianglin：当blacklist变化的时候怎么处理？

java-并发-Concurren...  
duoduo18up：很棒很全面~谢谢博主~

基于web的可视化数据库管理  
marko39：基于web的可视化数据库管理 ,还是用treesoft数据库管理系统 吧，支持mysql, ora...

Spark-zeppelin-大数...  
qq\_35022142：这还叫原创？？？！！

java-并发-Concurren...  
Ys8888N：厉害,写的真详细

转

Spark-SparkSQL深入学习系列九（转自OopsOutOfMemory）

2016年05月11日 19:36:05

阅读数：1094

/\*\* Spark SQL源码分析系列文章\*/

Spark SQL 可以将数据缓存到内存中，我们可以见到的通过调用cache table tableName即可将一张表缓存到内存中，来极大的提高查询效率。

这就涉及到内存中的数据的存储形式，我们知道基于关系型的数据可以存储为基于行存储结构 或者基于列存储结构，或者基于行和列的混合存储，即Row Based Storage、Column Based Storage、PAX Storage。

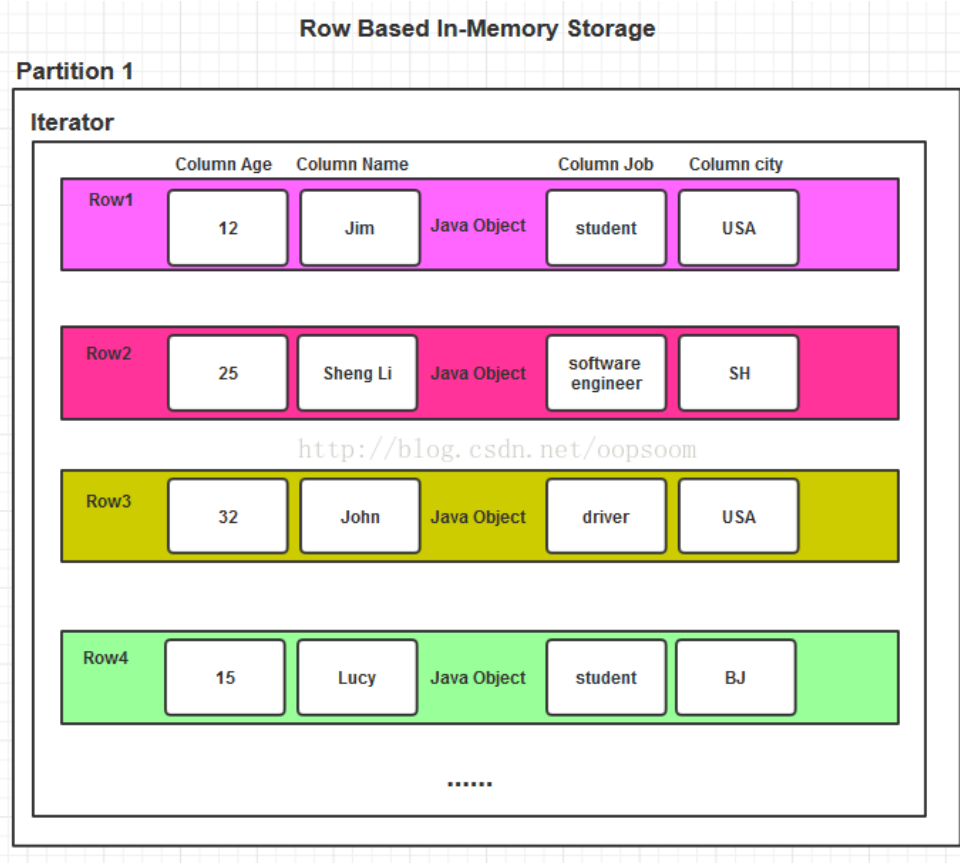
Spark SQL 的内存数据是如何组织的？

Spark SQL 将数据加载到内存是以列的存储结构。称为In-Memory Columnar Storage。

若直接存储Java Object 会产生很大的内存开销，并且这样是基于Row的存储结构。查询某些列速度略慢，虽然数据以及载入内存，查询效率还是低于面向列的存储结构。

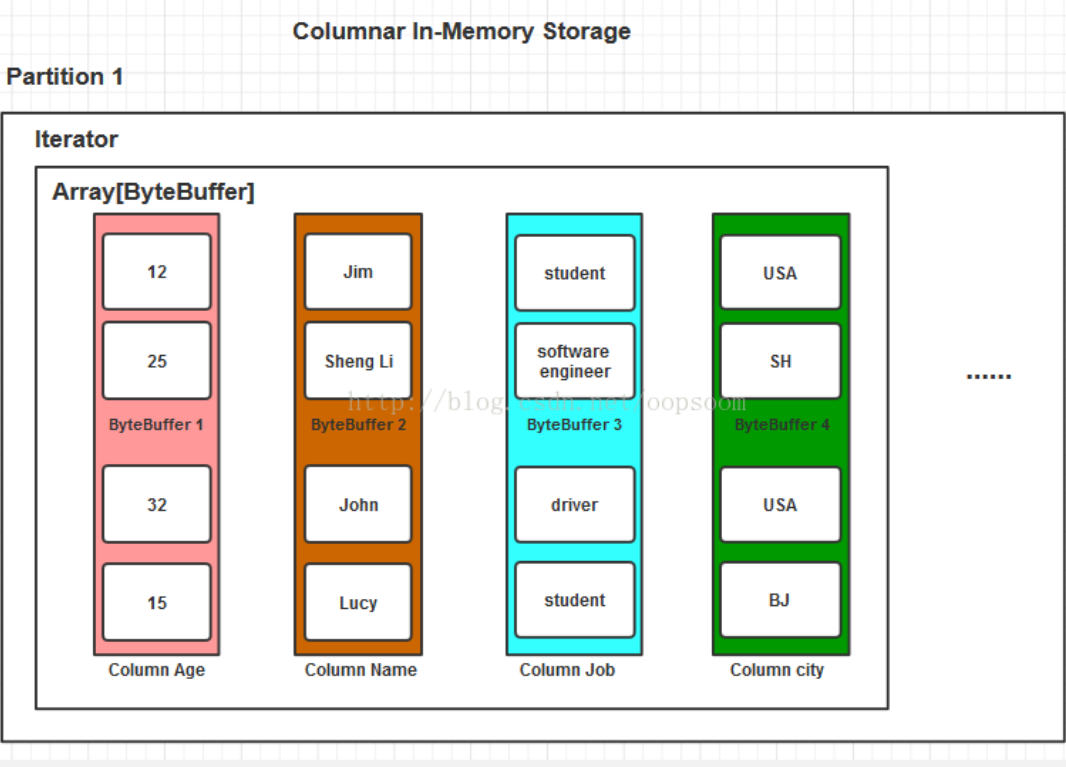
基于Row的Java Object存储：

内存开销大 且容易Full GC 按列查询比较慢



基于Column的ByteBuffer存储(Spark SQL)：

内存开销小，按列查询速度较快。



Spark SQL的In-Memory Columnar Storage是位于spark列下面org.apache.spark.sql.columnar包内: 核心的类有 ColumnBuilder, InMemoryColumnarTableScan, ColumnAccessor, ColumnType. 如果列有压缩的情况：compression包下面有具体的build列和access列的类。

```

└─ spark-sql
  └─ src/main/scala
    └─ org.apache.spark.sql
      └─ api.java
        └─ columnar
          └─ compression
            ├── CompressibleColumnAccessor.scala
            ├── CompressibleColumnBuilder.scala
            ├── CompressionScheme.scala
            ├── compressionSchemes.scala
            ├── ColumnAccessor.scala
            ├── ColumnBuilder.scala
            ├── ColumnStats.scala
            ├── ColumnType.scala
            ├── InMemoryColumnarTableScan.scala
            ├── NullableColumnAccessor.scala
            └── NullableColumnBuilder.scala

```

<http://blog.csdn.net/ooonsoom>

## 一、引子

当我们调用spark sql 里的cache table command时，会生成一CacheCommand，这个Command是一个物理计划。

```

[java]
01. scala> val cached = sql("cache table src")

```

```

[java]
01. cached: org.apache.spark.sql.SchemaRDD =
02. SchemaRDD[0] at RDD at SchemaRDD.scala:103
03. == Query Plan ==
04. == Physical Plan ==
05. CacheCommand src, true

```

这里打印出来tableName是src， 和一个是否要cache的boolean flag.

我们看下CacheCommand的构造：

CacheCommand支持2种操作，一种是把数据源加载带内存中，一种是将数据源从内存中卸载。

对应于SQLContext下的cacheTable和uncacheTable。

```

[java]
01. case class CacheCommand(tableName: String, doCache: Boolean)(@transient context: SQLContext)
02. extends LeafNode with Command {
03.
04. override protected[sql] lazy val sideEffectResult = {
05.   if (doCache) {
06.     context.cacheTable(tableName) //缓存表到内存
07.   } else {
08.     context.uncacheTable(tableName) //从内存中移除该表的数据
09.   }
10.   Seq.empty[Any]
11. }
12. override def execute(): RDD[Row] = {
13.   sideEffectResult
14.   context.emptyResult
15. }
16. override def output: Seq[Attribute] = Seq.empty
17. }

```

如果调用cached.collect(),则会根据Command命令来执行cache或者uncache操作，这里我们执行cache操作。

cached.collect()将会调用SQLContext下的cacheTable函数：

首先通过catalog查询关系，构造一个SchemaRDD。

```

[java]
01. /** Returns the specified table as a SchemaRDD */
02. def table(tableName: String): SchemaRDD =
03.   new SchemaRDD(this, catalog.lookupRelation(None, tableName))

```

找到该Schema的analyzed计划。匹配MemoryRelation：

```

[java]

```

```

04. val asInMemoryRelation =
05.   case _: InMemoryRelation => {
06.     currentTable.logicalPlan
07.   }
08.   case _ => //如果不是（默认是空的）则构建一个内存关系InMemoryRelation
09.     InMemoryRelation(useCompression, columnBatchSize, executePlan(currentTable).executedPlan)
10. }
11. //将构建好的InMemoryRelation注册到catalog里。
12. catalog.registerTable(NonEmptyRelation, asInMemoryRelation)
13. }

```

## 二、InMemoryRelation

InMemoryRelation继承自LogicalPlan，是Spark1.1 Spark SQL里新添加的一种TreeNode，也是catalyst里的一种plan。现在TreeNode变成了4种：

- 1、BinaryNode 二元节点
- 2、LeafNode 叶子节点
- 3、UnaryNode 单孩子节点
- 4、InMemoryRelation 内存关系型节点

```

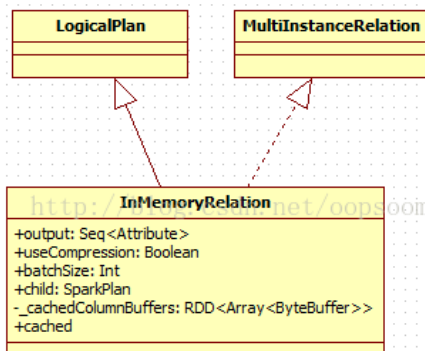
Object - java.lang
├── TreeNodes
│   ├── QueryPlan - org.apache.spark.sql.catalyst.plans
│   └── LogicalPlan - org.apache.spark.sql.catalyst.plans.logical
│       ├── BinaryNode - org.apache.spark.sql.catalyst.plans.logical
│       ├── InMemoryRelation - org.apache.spark.sql.catalyst.plans.logical
│       ├── LeafNode - org.apache.spark.sql.catalyst.plans.logical
│       └── UnaryNode - org.apache.spark.sql.catalyst.plans.logical

```

类图如下：

值得注意的是，\_cachedColumnBuffers这个类型为RDD[Array[ByteBuffer]]的私有字段。

这个封装就是面向列的存储ByteBuffer。前面提到相较于plain java object存储记录，用ByteBuffer能显著的提高存储效率，减少内存占用。并且按列查询的速度会非常快。



InMemoryRelation具体实现如下：

构造一个InMemoryRelation需要该Relation的输出 Attributes，是否需要useCoompression来压缩，默认为false，一次处理的多少行数据batchSize，child 即SparkPlan。

```

[java]
01. private[sql] case class InMemoryRelation(
02.   output: Seq[Attribute], //输出属性，比如src表里就是[key,value]
03.   useCompression: Boolean, //操作时是否使用压缩，默认false
04.   batchSize: Int, //批的大小量
05.   child: SparkPlan) //spark plan 具体child

```

可以通过设置：

spark.sql.inMemoryColumnarStorage.compressed 为true来设置内存中的列存储是否需要压缩。

spark.sql.inMemoryColumnarStorage.batchSize 来设置一次处理多少row

spark.sql.defaultSizeInBytes 来设置初始化的column的bufferbytes的默认大小，这里只是其中一个参数。

这些参数都可以在源码中设置，都在SQL Conf

[java]

```
04. | val DEFAULT_SIZE_IN_BYTES = "spark.sql.defaultSizeInBytes"
```

再回到case class InMemoryRelation :

\_cachedColumnBuffers就是我们最终将table放入内存的存储句柄，是一个RDD[Array[ByteBuffer]]。

## 缓存主流程：

- 1、判断\_cachedColumnBuffers是否为null，如果不是null，则已经Cache了当前table，重复cache不会触发cache操作。
- 2、child是SparkPlan，即执行hive table scan，测试我拿sbt/sbt hive/console里test里的src table为例，操作是扫描这张表。这个表有2个字的key是int, value 是string
- 3、拿到child的输出, 这里的output就是 key, value 2个列。
- 4、执行mapPartitions操作，对当前RDD的每个分区的数据进行操作。
- 5、对于每一个分区，迭代里面的数据生成新的Iterator。每个Iterator里面是Array[ByteBuffer]
- 6、对于child.output的每一列，都会生成一个ColumnBuilder，最后组合为一个columnBuilders是一个数组。
- 7、数组内每个CommandBuilder持有一个ByteBuffer
- 8、遍历原始分区的记录，将对于的行转为列，并将数据存到ByteBuffer内。
- 9、最后将此RDD调用cache方法，将RDD缓存。
- 10、将cached赋给\_cachedColumnBuffers。

**此操作总结下来是：执行hive table scan操作，返回的MapPartitionsRDD对其重新定义mapPartition方法，将其行转列，并且最终cache到内存中。**

所有流程如下：

```
[java]
01. // If the cached column buffers were not passed in, we calculate them in the constructor.
02. // As in Spark, the actual work of caching is lazy.
03. if (_cachedColumnBuffers == null) { //判断是否已经cache了当前table
04.     val output = child.output
05.     /**
06.      * child.output
07.      * res65: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = ArrayBuffer(key#6, value#7)
08.      */
09.     val cached = child.execute().mapPartitions { baseIterator =>
10.         /**
11.          * child.execute()是Row的集合，迭代Row
12.          * res66: Array[org.apache.spark.sql.catalyst.expressions.Row] = Array([238,val_238])
13.          *
14.          * val row1 = child.execute().take(1)
15.          * res67: Array[org.apache.spark.sql.catalyst.expressions.Row] = Array([238,val_238])
16.          */
17.         /**
18.          * 对每个Partition进行map，映射生成一个Iterator[Array[ByteBuffer]]，对应java的Iterator<List<ByteBuffer>>
19.          */
20.         new Iterator[Array[ByteBuffer]] {
21.             def next() = {
22.                 //遍历每一列，首先attribute是key 为 IntegerType，然后attribute是value是String
23.                 //最后封装成一个Array， index 0 是 IntColumnBuilder， 1 是StringColumnBuilder
24.                 val columnBuilders = output.map { attribute =>
25.                     val columnType = ColumnType(attribute.dataType)
26.                     val initialBufferSize = columnType.defaultSize * batchSize
27.                     ColumnBuilder(columnType.typeId, initialBufferSize, attribute.name, useCompression)
28.                 }.toArray
29.                 //src表里Row是[238,val_238] 这行Row的length就是2
30.                 var row: Row = null
31.                 var rowCount = 0
32.                 //batchSize默认1000
33.                 while (baseIterator.hasNext && rowCount < batchSize) {
34.                     //遍历每一条记录
35.                     row = baseIterator.next()
36.                     var i = 0
37.                     //这里row length是2, i的取值是0 和 1
38.                     while (i < row.length) {
39.                         //获取columnBuilders， 0是IntColumnBuilder，
40.                         //BasicColumnBuilder的appendFrom
41.                         //Appends `row(ordinal)` to the column builder.
42.                         columnBuilders(i).appendFrom(row, i)
43.                         i += 1
44.                     }
45.                     //该行已经插入完毕
46.                     rowCount += 1
47.                 }
48.                 //limit and rewind,Returns the final columnar byte buffer.
49.                 columnBuilders.map(_.build())
```

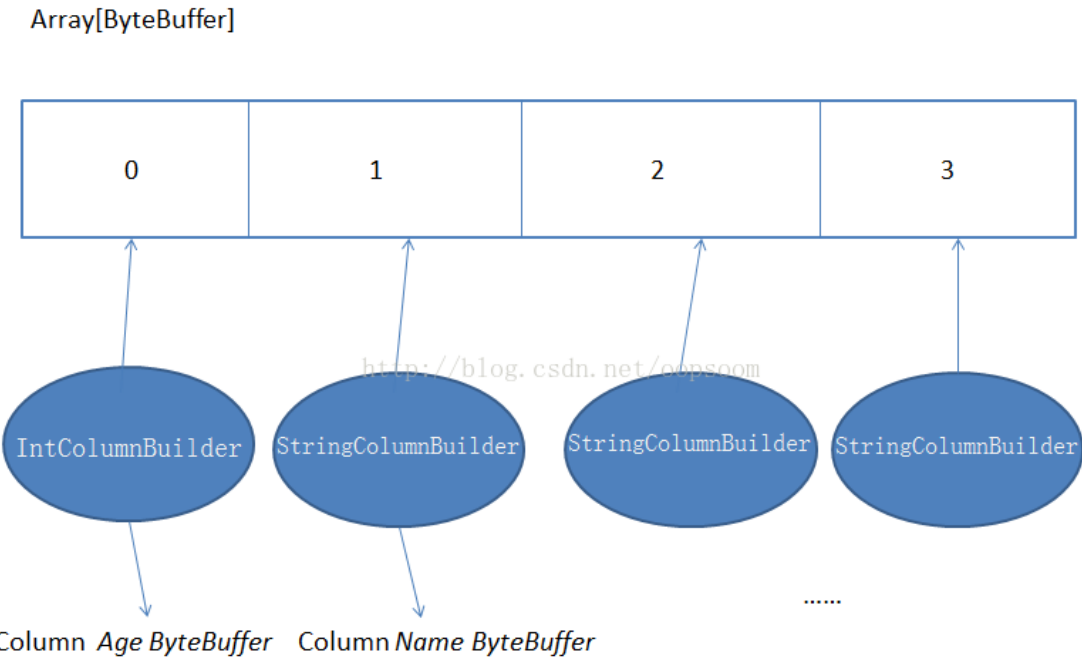
```
53.     }
54.     }.cache()
55.
56.     cached.setName(child.toString)
57.     _cachedColumnBuffers = cached
58. }
```

### 三、Columnar Storage

#### 初始化ColumnBuilders：

```
[java]
01. val columnBuilders = output.map { attribute =>
02.     val columnType = ColumnType(attribute.dataType)
03.     val initialBufferSize = columnType.defaultSize * batchSize
04.     ColumnBuilder(columnType.typeId, initialBufferSize, attribute.name, useCompression)
05. }.toArray
```

这里会声明一个数组，来对应每一列的存储，如下图：



然后初始化类型builder的时候会传入的参数：

initialBufferSize：文章开头的图中会有ByteBuffer，ByteBuffer的初始化大小是如何计算的？

initialBufferSize = 列类型默认长度 × batchSize，默认batchSize是1000

拿Int类型举例，initialBufferSize of IntegerType = 4 \* 1000

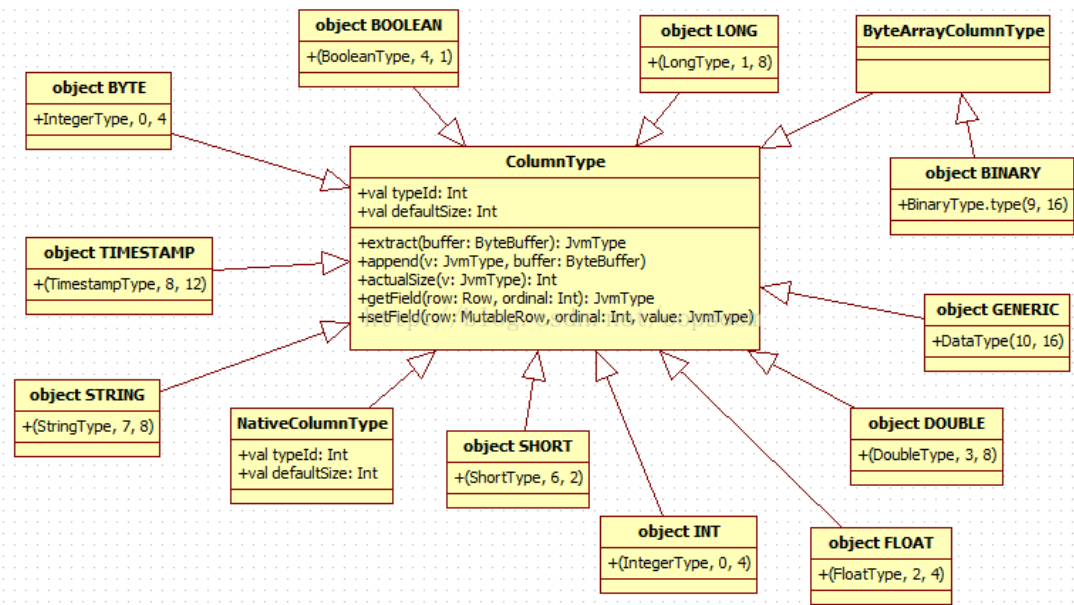
attribute.name即字段名age,name etc。。。。

#### ColumnType：

ColumnType封装了 该类型的 typeId 和 该类型的 defaultSize。并且提供了extract、append\getField方法，来向buffer里追加和获取数据。

如IntegerType typeId 为0，defaultSize 4 .....

详细看下类图，画的不是非常严格的类图，主要为了展示目前类型系统：



ColumnBuilder :

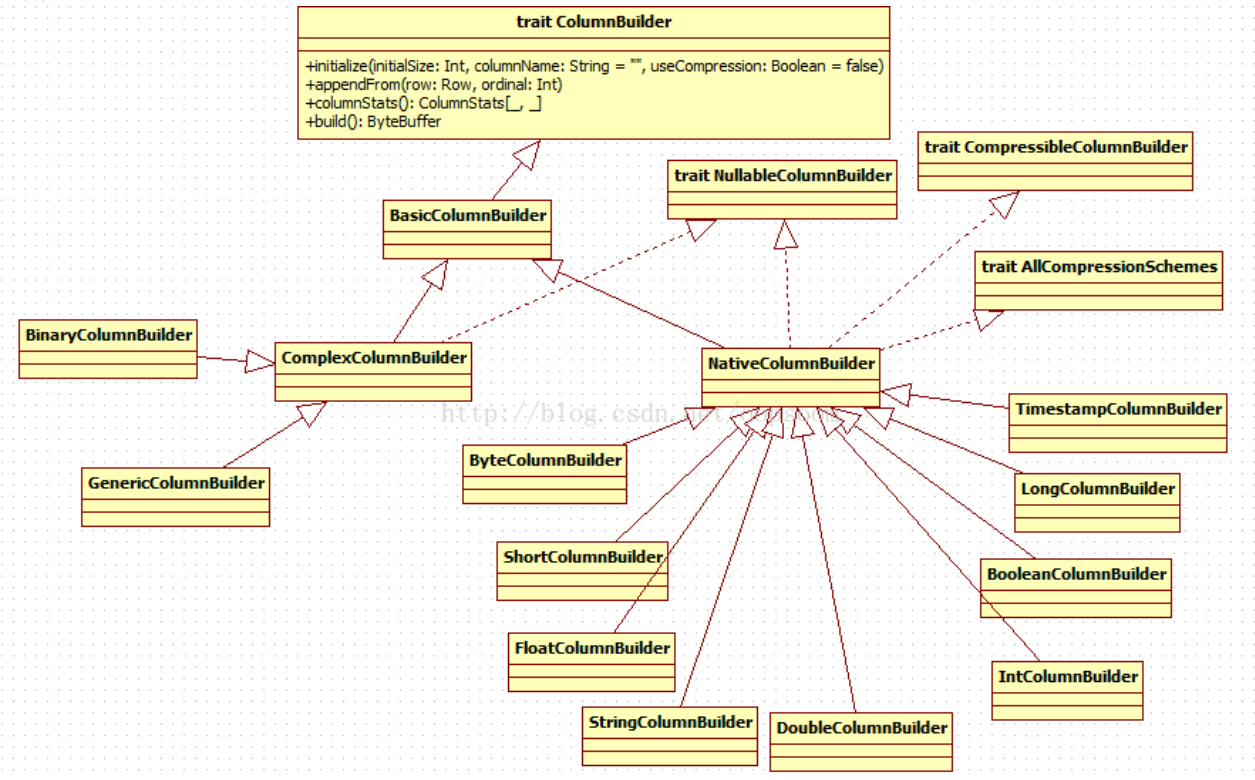
ColumnBuilder的主要职责是：管理ByteBuffer，包括初始化buffer，添加数据到buffer内，检查剩余空间，和申请新的空间这几项主要职责。

initialize负责初始化buffer。

appendFrom是负责添加数据。

ensureFreeSpace确保buffer的长度动态增加。

类图如下：



ByteBuffer的初始化过程：

初始化大小initialSize：拿Int举例，在前面builder初始化传入的是4×batchSize=4\*1000，initialSize也就是4KB，如果没有传入initialSize,则默认是1024×1024。

列名称，是否需要压缩，都是需要传入的。

ByteBuffer声明时预留了4个字节，为了放column type id,这个在ColumnType的构造里有介绍过。

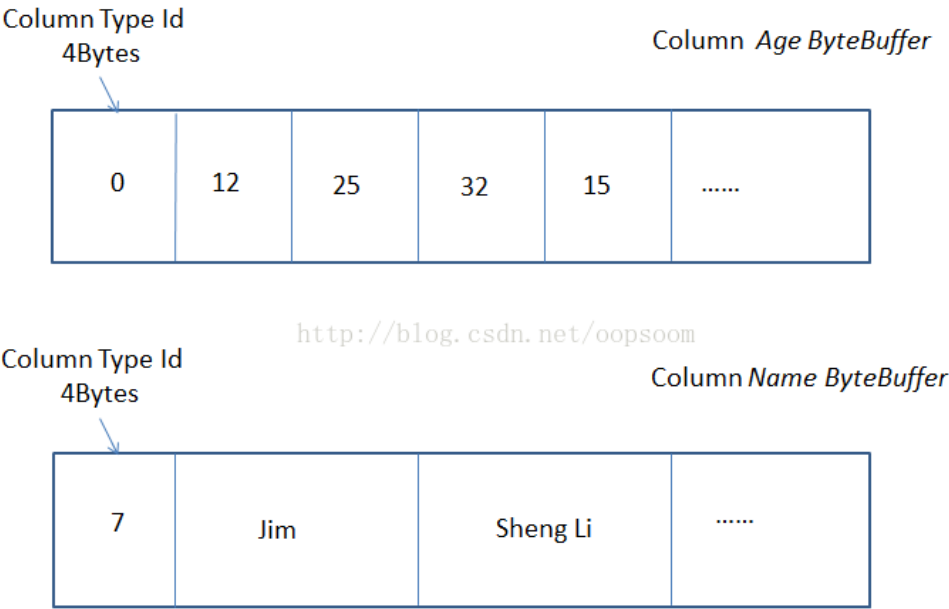
[java]



```
04.         useCompression: Boolean = false) = {
05.
06.         val size = if (initialSize == 0) DEFAULT_INITIAL_BUFFER_SIZE else initialSize //如果没有默认1024×1024 byte
07.         this.columnName = columnName
08.
09.         // Reserves 4 bytes for column type ID
10.         buffer = ByteBuffer.allocate(4 + size * columnType.defaultSize) // buffer的初始化长度，需要加上4byte类型ID空间。
11.         buffer.order(ByteOrder.nativeOrder()).putInt(columnType.typeId)//根据nativeOrder排序，然后首先放入typeId
12.     }
```

存储的方式如下：

Int的type id 是0，string的 type id 是 7. 后面就是实际存储的数据了。



ByteBuffer写入过程:

存储结构都介绍完毕，最后开始对Table进行scan了，scan后对每一个分区的每个Row进行操作遍历：

- 1、读每个分区的每条Row
- 2、获取每个列的值，从builders数组里找到索引 i 对应的bytebuffer，追加至bytebuffer。

```
[java]
01. while (baseIterator.hasNext && rowCount < batchSize) {
02.     //遍历每一条记录
03.     row = baseIterator.next()
04.     var i = 0
05.     //这里row length是2, i的取值是0 和 1 Ps:还是拿src table做测试，每一个Row只有2个字段，key, value所有长度为2
06.     while (i < row.length) {
07.         //获取columnBuilders, 0是IntColumnBuilder,
08.         //BasicColumnBuilder的appendFrom
09.         //Appends `row(ordinal)` to the column builder.
10.         columnBuilders(i).appendFrom(row, i) //追加到对应的bytebuffer
11.         i += 1
12.     }
13.     //该行已经插入完毕
14.     rowCount += 1
15. }
16. //limit and rewind>Returns the final columnar byte buffer.
17. columnBuilders.map(_.build())
```

追加过程：

根据当前builder的类型，从row的对应索引中取出值，最后追加到builder的bytebuffer内。

```
[java]
01. override def appendFrom(row: Row, ordinal: Int) {
02.     //ordinal是Row的index, 0就是第一列值, 1就是第二列值，获取列的值为field
03.     //最后在该列的值put到该buffer内
04.     val field = columnType.getField(row, ordinal)
05.     buffer = ensureFreeSpace(buffer, columnType.actualSize(field))//动态扩容
```

**ensureFreeSpace :**

主要是操作buffer，如果要追加的数据大于剩余空间，就扩大buffer。

```
[java]
01. //确保剩余空间能容下，如果剩余空间小于 要放入的大小，则重新分配一看内存空间
02. private[columnar] def ensureFreeSpace(orig: ByteBuffer, size: Int) = {
03.     if (orig.remaining >= size) { //当前buffer剩余空间比要追加的数据大，则什么都不做，返回自身
04.         orig
05.     } else { //否则扩容
06.         // grow in steps of initial size
07.         val capacity = orig.capacity()
08.         val newSize = capacity + size.max(capacity / 8 + 1)
09.         val pos = orig.position()
10.
11.         orig.clear()
12.         ByteBuffer
13.             .allocate(newSize)
14.             .order(ByteOrder.nativeOrder())
15.             .put(orig.array(), 0, pos)
16.     }
17. }
```

.....

最后调用MapPartitionsRDD.cache()，将该RDD缓存并添加到spark cache管理中。

至此，我们将一张spark sql table缓存到了spark的jvm中。

## 四、总结

对于数据的存储结构，我们常常关注持久化的存储结构，并且在长久时间内有了很多种高效结构。

但是在实时性的要求下，内存**数据库**越来越被关注，如何优化内存数据库的存储结构，是一个重点，也是一个难点。

对于Spark SQL 和 Shark 里的列存储 是一种优化方案，提高了关系查询中列查询的速度，和减少了内存占用。但是中存储方式还是比较简单的，没有额外的元数据和索引来提高查询效率，希望以后能了解到更多的In-Memory Storage。

——EOF——

创文章，转载请注明：

转载自：[OopsOutOfMemory盛利的Blog](#)，作者：[OopsOutOfMemory](#)

本文链接地址：<http://blog.csdn.net/oopsoom/article/details/39525483>

注：本文基于署名-非商业性使用-禁止演绎 2.5 中国大陆(CC BY-NC-ND 2.5 CN)协议，欢迎转载、转发和评论，但是请保留本文作者署名和文章链接。如若需要用于商业目的或者与授权方面的协商，请联系我。



个人分类：[大数据框架](#)

[查看更多>>](#)

想对作者说点什么？

[我来说一句](#)

spark SQL (三) 数据源 Data Source----通用的数据 加载/保存功能

 m0\_37138008 2017-12-30 11:25:25 阅读数：158

## spark持久化（缓存）

1、惰性求值：RDD转化过程都是惰性求值的。这意味着在被调用行动操作之前spark不会开始计算，spark会在内部记录下所要求执行的操作的相关信息，我们可以把每个RDD看作我们通过转化操作构建出来...

 little\_nai 2016-12-12 16:48:30 阅读数：1436

广告

## Spark Streaming：缓存与持久化机制

Spark Streaming：缓存与持久化机制

 kwu\_ganymede 2016-01-25 22:32:27 阅读数：2084

## Spark SQL 源码分析之 In-Memory Columnar Storage 之 cache table

Spark SQL缓存到内存中的数据的存储策略

 u014388509 2014-09-25 18:20:23 阅读数：8570

## spark学习6-spark模拟hive的列转行和行转列

继续上一篇学习spark 本次将通过一个实际场景来综合学习下spark如何实现hive中的列转行和行转列（关于hive的）列转行和行转列介绍见：[http://blog.csdn.net/jthink\\_...](http://blog.csdn.net/jthink_...)

 JThink\_ 2015-10-14 09:47:09 阅读数：4667

## Spark-SQL之DataFrame操作大全

Spark SQL中的DataFrame类似于一张关系型数据表。在关系型数据库中对单表或进行的查询操作，在DataFrame中都可以通过调用其API接口来实现。可以参考，Scala提供的DataFra...

 dabokele 2016-10-12 23:31:35 阅读数：60318

## 终于曝光微赚钱内情，网友：白玩了这么多年

金泉投资·顶新

## 整理对Spark SQL的理解

Catalyst定位 其他系统如果想基于Spark做一些类sql、标准sql甚至其他查询语言的查询，需要基于Catalyst提供的解析器、执行计划树结构、逻辑执行计划的处理规则体系等类体系来实现执行计...

 zbf8441372 2014-07-15 10:18:51 阅读数：25223

## spark sql

Spark SQL运行机制 Spark SQL 对 SQL 语句的处理和关系型数据库对 SQL 语句的处理采用了类似的方法,首先会将 SQL 语句进行解析(Parse),然后形成一个 Tree,在后续...

 lsshls 2014-11-08 11:00:28 阅读数：2134

## Spark SQL入门用法与原理分析

sparkSQL是为了让开发人员摆脱自己编写RDD等原生Spark代码而产生的，开发人员只需要写一句SQL语句或者调用API，就能生成（翻译成）对应的SparkJob代码并去执行，开发变得更简洁，1...

 silviakafka 2017-01-05 11:48:14 阅读数：10543

## [Spark]Spark RDD 指南五 持久化

1. 概述Spark中最重要的功能之一是操作时在内存中持久化(缓存)数据集(persisting (or caching) a dataset in memory across operations)...

### SparkSQL性能调优

最近在[学习spark](#)时，觉得Spark SQL性能调优比较重要，所以自己写下来便于更过的博友查看，同时也希望大家给我指出我的问题和不足 在spark中，Spark SQL性能调优只要是通过下面的一些选...

 YQLakers

2017-03-31 14:54:48

阅读数：4402

### spark sql cache

1.几种缓存数据的方法例如有一张hive表叫做activity1.CACHE TABLE//缓存全表 sqlContext.sql("CACHE TABLE activity")//缓存过滤结果 sq...

 Isshlsiw

2015-09-22 21:58:21

阅读数：5971

### sql行转列和列转行

最近建立数据立方体需要将表的数据结构进行转换，进行列转行，觉得用途还很大，所以就整理一下，当做自己的笔记拉。 1、列转行（主要）表weath  
erdata结构： create table WEA...

 jiyang\_1

2017-01-15 17:04:53

阅读数：698

### 有道智云：智能翻译api为开发者赋能

翻译准确率业界领先，已通过开发者为7.5用户提供优质的翻译服务



### SQLserver行转列，列转行

目录结构如下：行转列列转行 [一]、行转列 1.1、初始测试数据 表结构：TEST\_TB\_GRADE create table TEST\_...

 fengzhongdeyuyi1992

2015-11-06 10:10:15

阅读数：288

### Spark-SQL 之DataFrame操作大全

```
package com.sdctet import org.apache.spark.sql.SQLContext import org.apache.spark.{SparkConf, SparkC...
```

 xfg0218

2017-02-18 19:56:47

阅读数：890

### SQL列转行及行转列

-----作者：王运亮(wwwwgou)时间：2011-06-10博客：http://blog.csdn.net/...

 wwwwgou

2011-06-10 09:04:00

阅读数：7092

### 重温SQL——行转列，列转行（转）

行转列，列转行是我们在开发过程中经常碰到的问题。行转列一般通过CASE WHEN 语句来实现，也可以通过 SQL SERVER 2005 新增的运算符PIVOT 来实现。用传统的方法，比较好理解。层次清...

 bai449083657

2016-11-10 11:39:40

阅读数：179

### ORACLE 列转行和行转列的SQL

网络上关于行转列和列转行的文章不少，但要么太复杂，要么太凌乱，此处用一个小例子说明如何通过简单SQL实现行列转换。表test NAME KM CJ ...

 thy822

2012-08-28 10:59:40

阅读数：6993

### 「包图网」设计服务生活

1000万套ppt模板,每日更新,找PPT模板就上[包图网]

百度广告



### Spark-SparkSQL深入学习系列四（ 转自OopsOutOfMemory ）

/\*\* Spark SQL源码分析[系列文章](#)\*/ 前几篇文章介绍了Spark SQL的Catalyst的核心运行流程、SqlParser，和Analyzer，本来打算直接写Optimize...

 youdianjinjin

2016-05-11 19:28:08

阅读数：600

/\*\* Spark SQL源码分析系列文章\*/ Spark SQL的核心执行流程我们已经分析完毕，可以参见Spark SQL核心执行流程，下面我们来分析执行流程中各个核心组件的工作职...

 youdianjinjin

2016-05-11 19:25:19

阅读数：748

### Spark-SparkSQL深入学习系列十（ 转自OopsOutOfMemory ）

/\*\* Spark SQL源码分析系列文章\*/ 前面讲到了Spark SQL In-Memory Columnar Storage的存储结构是基于列存储的。 那么基于以上存...

 youdianjinjin

2016-05-11 19:37:32

阅读数：505

### Spark-SparkSQL深入学习系列一（ 转自OopsOutOfMemory ）

/\*\* Spark SQL源码分析系列文章\*/ 自从去年Spark Submit 2013 Michael Armbrust分享了他的Catalyst，到至今1年多了,Spark SQL的...

 youdianjinjin

2016-05-11 19:22:41

阅读数：829

### Spark-SparkSQL深入学习系列十一（ 转自OopsOutOfMemory ）

上周Spark1.2刚发布，周末在家没事，把这个特性给了解一下，顺便分析下源码，看一看这个特性是如何设计及实现的。 /\*\* Spark SQL源码分析系列文章\*/ （Ps: E...

 youdianjinjin

2016-05-11 19:38:23

阅读数：633

### 专注翻译，所以卓越\_网易有道

翻译api已接入微信、QQ国际版、网易邮箱等头部app、服务7亿用户



### Spark-SparkSQL深入学习系列三（ 转自OopsOutOfMemory ）

/\*\* Spark SQL源码分析系列文章\*/ 前面几篇文章讲解了Spark SQL的核心执行流程和Spark SQL的Catalyst框架的Sql Parser是怎样接受用户输入s...

 youdianjinjin

2016-05-11 19:26:30

阅读数：473

### Spark-SparkSQL深入学习系列五（ 转自OopsOutOfMemory ）

/\*\* Spark SQL源码分析系列文章\*/ 前几篇文章介绍了Spark SQL的Catalyst的核心运行流程、SqlParser，和Analyzer 以及核心类库TreeNode，本文...

 youdianjinjin

2016-05-11 19:29:27

阅读数：671

### Spark-SparkSQL深入学习系列八（ 转自OopsOutOfMemory ）

/\*\* Spark SQL源码分析系列文章\*/ 在SQL的世界里，除了官方提供的常用的处理函数之外，一般都会提供可扩展的对外自定义函数接口，这已经成为一种事实的标准。 在前面Sp...

 youdianjinjin

2016-05-11 19:35:45

阅读数：452

### Spark-SparkSQL深入学习系列六（ 转自OopsOutOfMemory ）

/\*\* Spark SQL源码分析系列文章\*/ 前面几篇文章主要介绍的是Spark sql包里的的spark sql执行流程，以及Catalyst包内的SqlParser，Analyze...

 youdianjinjin

2016-05-11 19:30:39

阅读数：1087

### Spark-SparkSQL深入学习系列七（ 转自OopsOutOfMemory ）

/\*\* Spark SQL源码分析系列文章\*/ 接上一篇文章Spark SQL Catalyst源码分析之Physical Plan，本文将介绍Physical Plan的toRDD的具...

 youdianjinjin

2016-05-11 19:33:22

阅读数：1297

### 人工智能网站

电商巨头抢滩人工智能

百度广告



### 【PSI/SI学习系列】2.PSI/SI深入学习1——预备知识

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册

×

## Spring+SpringMVC+MyBatis深入学习及搭建(三)——MyBatis全局配置文件解析

转载请注明出处：<http://www.cnblogs.com/Joanna-Yan/p/6874672.html> MyBatis的全局配置文件SqlMapConfig.xml，配置内容和顺序如下：pr...

 gozhuyinglong 2018-03-16 10:04:06 阅读数：27

## 深入Java集合学习系列（一）

HashMap的实现原理 HashMap 概述：HashMap 是基于哈希表的 Map 接口的非同步实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序...

 chengyunyi123 2016-11-25 21:24:43 阅读数：196

## 深入Java集合学习系列：深入CopyOnWriteArraySet

[http://www.cnblogs.com/skywang12345/p/3498497.html?utm\\_source=tuicool](http://www.cnblogs.com/skywang12345/p/3498497.html?utm_source=tuicool) 概要 本章是JUC系列中的Cop...

 lihui6636 2015-10-07 11:11:05 阅读数：3565

## 深入学习spring-boot系列（一）--spring-boot系列开篇

本文是spring boot系列的开篇，spring boot系列文章至少会有10来20篇，用于记录工作和学习中的问题与解决方案。 spring boot是什么？spring boot就是类似...

 u012558400 2016-11-24 15:00:18 阅读数：1432


## 程序猿学炒股投资，拒绝死工资！

网易官方股票交流群！免费送您3支牛股




## 深入理解Bootstrap -- 学习从现在开始1

当下最流行的前端开发框架Bootstrap，可大大简化网站开发过程，从而深受广大开发者的喜欢。本文总结了Bootstrap之所以广泛流传的11大原因。如果你还没有使用Twitter Bootstrap...

 z742182637 2016-01-05 14:53:29 阅读数：1592

## 深入研究Windows内部原理绝对经典的资料

（为了方便大家下，我打包了放在一下地址：1-6：<http://download.csdn.net/detail/wangqiulin123456/4601530> 7-12：<http://down...>

 wangqiulin123456 2012-09-27 08:10:35 阅读数：5903


## netty深入学习之一: 入门篇

Netty是Java NIO之上的网络库（API）。Netty 提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。我构建了一个Netty项目模板：av...

 cheungmine 2015-03-13 11:52:54 阅读数：2040

## WebRTC学习之七：精炼的信号和槽机制

关于信号和槽有一个非常精炼的C++实现，作者是Sarah Thompson，该实现只有一个头文件sigslot.h，源码在：<http://sigslot.cvs.sourceforge.net/vi...>

 caoshangpa 2017-01-05 09:06:52 阅读数：1365

## 深入Java集合学习系列(二)：ArrayList实现原理

深入Java集合学习系列(二)：ArrayList实现原理

下载 2018年01月30日 11:27

## k系列减速机

新宝减速机 原装进口





## 【SignalR学习系列】2. 第一个SignalR程序

新建项目 1.使用VisualStudio 2015 新建一个Web项目 2.选择空模板 3.添加一个新的SignalR Hub Class (v2)类文件，并修改类名为ChatHub ...

Andrewniu 2017-11-01 16:02:54 阅读数：178

## 【PSI/SI学习系列】2.PSI/SI深入学习3——SI信息解析1(NIT,BAT)

SI 信息 INFORMATION OF SI "SI是对多个TS流的描述，它包含了PSI" PSI只提供了单个TS流的信息，使接收机能够对单个TS流中的不同节目进行解码；但是， ...

u010090005 2013-10-24 11:04:03 阅读数：5141

## 【Python学习系列十】Python机器学习库scikit-learn实现Decision Trees案例

学习网址：<http://scikit-learn.org/stable/modules/tree.html> scikit-learn这个官网很好，里面有算法案例也有算法原理说明。案例代码： ...

fjssharpword 2017-06-08 16:49:14 阅读数：2034

## dubbo深入理解（1）

dubbo学习，分布式服务架构，国内最好的一个分布式服务框架,致力于提供高性能和透明化的RPC远程服务调用方案,以及SOA服务治理方案...

maybe\_fly 2017-09-11 14:18:44 阅读数：682

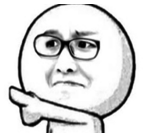
## Java基础重新学习--开篇

接下来一系列的Java基础文章意在巩固Java基础知识。学习路线参考了传智播客给出的Java学习路线图。附录一张Java学习路线图（来自图转传智播客）：从上图中可以看出Java学习路线可以分为6大...

lovewaterman 2016-11-20 17:47:52 阅读数：784

## 码农怎能不懂英语？！试试这个数学公式

老司机教你一个数学公式秒懂天下英语



## 深入理解Spring系列之一：开篇

Spring经过大神们的构思、编码，日积月累而来，所以，对其代码的理解也不是一朝一夕就能快速完成的。源码学习是枯燥的，需要坚持！坚持！坚持！当然也需要技巧，第一遍学习的时候，不用关注全部细节，不重要的...

tianrui 2016-10-30 20:18:39 阅读数：3427

## 调试九法中文版

调试九法中文版，目录齐全

下载 2016年05月25日 20:15

## springboot(九)：定时任务

在我们的项目开发过程中，经常需要定时任务来帮助我们来做一些内容，springboot默认已经帮我们实行了，只需要添加相应的注解就可以实现 1、pom包配置 pom包里面只需要引入sprin...

gebitan505 2017-02-08 17:45:56 阅读数：570

## 深入理解Android系列书籍资源分享更新

由于115网盘限制礼包下载，我现在将深入理解Android系列书籍或其他资源转移到百度网盘上，供兄弟姐妹们下载分享。1 深入理解Android：Wi-Fi,NF C和GPS卷下载地址：<http://pa...>

Innost 2015-01-31 16:53:46 阅读数：37864

## 【PSI/SI学习系列】2.PSI/SI深入学习2——PSI信息解析(PAT,PMT,CAT)

PSI 信息 INFORMATION OF PSI "PSI是对单一TS流的描述，是TS流中的引导信息" PSI信息由节目关联表PAT、条件接收表CAT、节目映射表PM

1克拉的钻戒大概多少钱

1克拉钻戒价格

百度广告



强化学习系列之九:Deep Q Network (DQN)

文章目录 [隐藏] 1. 强化学习和深度学习结合 2. Deep Q Network (DQN) 算法 3. 后续发展 3.1 Double DQN 3.2 Prioriti...

bbbeoy 2018-01-16 10:45:34 阅读数：203

深入Java集合学习系列：LinkedList的实现原理

1. LinkedList概述： List 接口的链接列表实现。实现所有可选的列表操作，并且允许所有元素（包括 null）。除了实现 List 接口外，LinkedList 类还为列表的开头...

zheng0518 2014-12-27 21:38:16 阅读数：6944

没有更多推荐了，[返回首页](#)