单例模式

单例模式的几种实现方式

1、懒汉式，线程不安全

是否 Lazy 初始化：是

是否多线程安全：否

实现难度：易

描述：这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。

这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。

实例

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

接下来介绍的几种实现方式都支持多线程，但是在性能上有所差异。

2、懒汉式，线程安全

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：易

描述：这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。

优点：第一次调用才初始化，避免内存浪费。

缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。

getInstance() 的性能对应用程序不是很关键（该方法使用不太频繁）。

实例

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

3、饿汉式

是否 Lazy 初始化：否

是否多线程安全：是

实现难度：易

描述：这种方式比较常用，但容易产生垃圾对象。

优点：没有加锁，执行效率会提高。

缺点：类加载时就初始化，浪费内存。

它基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

实例

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton (){}

public static Singleton getInstance() {

return instance;

}

}

4、双检锁/双重校验锁（DCL，即 double-checked locking）

JDK 版本：JDK1.5 起

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：较复杂

描述：这种方式采用双锁机制，安全且在多线程情况下能保持高性能。

getInstance() 的性能对应用程序很关键。

实例

public class Singleton {

private volatile static Singleton singleton;

private Singleton (){}

public static Singleton getSingleton() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

5、登记式/静态内部类

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：一般

描述：这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。

这种方式同样利用了 classloader 机制来保证初始化 instance 时只有一个线程，它跟第 3 种方式不同的是：第 3 种方式只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有通过显式调用 getInstance 方法时，才会显式装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，所以想让它延迟加载，另外一方面，又不希望在 Singleton 类加载时就实例化，因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比第 3 种方式就显得很合理。

实例

public class Singleton {

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

private Singleton (){}

public static final Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

6、枚举

JDK 版本：JDK1.5 起

是否 Lazy 初始化：否

是否多线程安全：是

实现难度：易

描述：这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。

这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还自动支持序列化机制，防止反序列化重新创建新的对象，绝对防止多次实例化。不过，由于 JDK1.5 之后才加入 enum 特性，用这种方式写不免让人感觉生疏，在实际工作中，也很少用。

不能通过 reflection attack 来调用私有构造方法。

实例

public enum Singleton {

INSTANCE;

public void whateverMethod() {

}

}

经验之谈：一般情况下，不建议使用第 1 种和第 2 种懒汉方式，建议使用第 3 种饿汉方式。只有在要明确实现 lazy loading 效果时，才会使用第 5 种登记方式。如果涉及到反序列化创建对象时，可以尝试使用第 6 种枚举方式。如果有其他特殊的需求，可以考虑使用第 4 种双检锁方式。

三大范式

1、1NF:字段不可分，每个字段是原子级别的，上节中看到第一个字段为ID，它就是ID不能在分成两个字段了，不能说我要把这个人的ID、名称、班级号都塞在一个字段里面，这个是不合适的，对以后的应用造成很大影响；

2、2NF:有主键，非主键字段依赖主键，ID字段就是主键，它能表示这一条数据是唯一的，有的读者朋友记性很好，“unique”表示唯一的、不允许重复的，确实它经常会修饰某个字段，保证该字段唯一性，然后再设置该字段为主键；

3、3NF:非主键字段不能相互依赖，这个怎么理解呢，比如student表，班级编号受人员编号的影响，如果在这个表中再插入班级的班主任、数学老师等信息，你们觉得这样合适吗？肯定不合适，因为学生有多个，这样就会造成班级有多个，那么每个班级的班主任、数学老师都会出现多条数据，而我们理想中的效果应该是一条班级信息对应一个班主任和数学老师，这样更易于我们理解，这样就形成class表，那么student表和class表中间靠哪个字段来关联呢，肯定是通过“classNo”，这个字段也叫做两个表的外键，后面讲约束的时候老韩会重点讲这个，读者朋友先有个大致了解；

二级索引

隔离级别

不可重复读

可重复读

序列化

Hdfs写流程

读流程

hive调优的思路

1 好的模型设计

2 解决数据倾斜

3 减少job数

4 设置合理的map reduce的task数

5 对小文件进行合并

6 单个作业最优不如整体最优

写Sql

手写wordcount

手写 spark 求 π、wordcount、二次排序

简单总结一下：

消费端重复消费：很容易解决，建立去重表。

消费端丢失数据：也容易解决，关闭自动提交offset，处理完之后受到移位。

生产端重复发送：这个不重要，消费端消费之前从去重表中判重就可以。

生产端丢失数据：这个是最麻烦的情况。

Kafka ISR机制

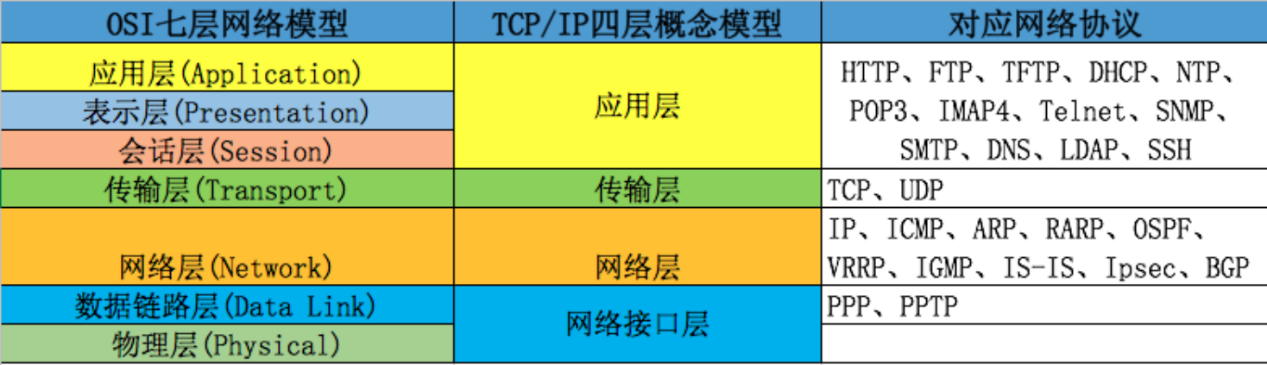
解决spark数据倾斜方法：

1. hive ETL预处理：将数据倾斜提前到ETL阶段，治标不治本
2. 过滤掉少数导致倾斜的key：主要用在产生大量异常数据的情况下
3. 提高shuffle的 并行度：防止多个key在同一task中计算
4. 两阶段聚合（局部聚合+全局聚合）：在key前加上前缀，使key平均，但只适用于reducebykey和groupby这类shuffle中
5. 使用map join：适合小表join大表
6. 采样倾斜key并分拆join操作：将导致倾斜的key单独拆成RDD算子，并加前缀拆分成n个任务，适合只有少数key倾斜的情况

Js跨域

1. jsonp
2. document.domain
3. windows.name
4. 等

TCP/IP



Spark效率比mapreduce高的原因：

1、Spark是基于内存的，而MapReduce是基于磁盘的迭代

2、有向无环图是指：一个图从顶点出发，无法再回到原点，那么这种图叫做有向无环图。

3、DAG计算模型在spark任务调度

4、Spark是粗粒度资源调度，MapReduce是细粒度资源调度

**1、Spark vs MapReduce ≠ 内存 vs 磁盘**

其实Spark和MapReduce的计算都发生在内存中，区别在于：

* MapReduce通常需要将计算的中间结果写入磁盘，然后还要读取磁盘，从而导致了频繁的磁盘IO。
* Spark则不需要将计算的中间结果写入磁盘，这得益于Spark的RDD（弹性分布式数据集，很强大）和DAG（有向无环图），其中DAG记录了job的stage以及在job执行过程中父RDD和子RDD之间的依赖关系。中间结果能够以RDD的形式存放在内存中，且能够从DAG中恢复，大大减少了磁盘IO。

**2、Spark vs MapReduce Shuffle的不同**

Spark和MapReduce在计算过程中通常都不可避免的会进行Shuffle，但Spark因为DAG，Shuffle尽可能减少；

**3、多进程模型 vs 多线程模型的区别**

* MapReduce采用了多进程模型，而Spark采用了多线程模型。多进程模型的好处是便于细粒度控制每个任务占用的资源，但每次任务的启动都会消耗一定的启动时间。就是说MapReduce的Map Task和Reduce Task是进程级别的，而Spark Task则是基于线程模型的，就是说mapreduce 中的 map 和 reduce 都是 jvm 进程，每次启动都需要重新申请资源，消耗了不必要的时间（假设容器启动时间大概1s，如果有1200个block，那么单独启动map进程事件就需要20分钟）
* Spark则是通过复用线程池中的线程来减少启动、关闭task所需要的开销。（多线程模型也有缺点，由于同节点上所有任务运行在一个进程中，因此，会出现严重的资源争用，难以细粒度控制每个任务占用资源）

4、序列化方式不同

总结：关于Spark为什么比MapReduce快，或者Spark速度快于MapReduce的原因，总结至少有这几点不同之处吧。

**1、消除了冗余的HDFS读写**  
Hadoop每次shuffle操作后，必须写到磁盘，而Spark在shuffle后不一定落盘，可以cache到内存中，以便迭代时使用，而且针对于spark-shuffle spark使用的是bypass的sortshuffle机制，进一步提高效率。如果操作复杂，很多的shufle操作，那么Hadoop的读写IO时间会大大增加，相对的spark就很快。  
**2、消除了冗余的MapReduce阶段**  
Hadoop的shuffle操作一定连着完整的MapReduce操作，冗余繁琐。而Spark基于RDD提供了丰富的算子操作，且reduce操作产生shuffle数据，可以缓存在内存中。  
**3、JVM的优化**  
Hadoop每次MapReduce操作，启动一个Task便会启动一次JVM，基于进程的操作。而Spark每次MapReduce操作是基于线程的，只在启动Executor是启动一次JVM，内存的Task操作是在线程复用的。每次启动JVM的时间可能就需要几秒甚至十几秒，那么当Task多了，这个时间Hadoop不知道比Spark慢了多少。  
**4.序列化方式**  
Spark序列化方式提供了kryo序列化–是一种轻量级的序列化，MR只有java的序列化–重量级。  
**5. DAG Scheduler**  
Spark 计算比 MapReduce 快的根本原因在于 DAG 计算模型。一般而言，DAG 相比MapReduce 在大多数情况下可以减少 shuffle 次数。Spark 的 DAGScheduler 相当于一个改进版的 MapReduce，如果计算不涉及与其他节点进行数据交换，Spark 可以在内存中一次性完成这些操作，也就是中间结果无须落盘，减少了磁盘 IO 的操作。

Spark和flink对比

Flink 和 Spark 对比

通过前面的学习，我们了解到，Spark和Flink都支持批处理和流处理，接下来让我们对这两种流行的数据处理框架在各方面进行对比。首先，这两个数据处理框架有很多相同点。

都基于内存计算；

都有统一的批处理和流处理APl，都支持类似SQL的编程接口；

都支持很多相同的转换操作，编程都是用类似于Scala Collection APl的函数式编程模式；

都有完善的错误恢复机制；

都支持Exactly once的语义一致性。

当然，它们的不同点也是相当明显，我们可以从4个不同的角度来看。

从流处理的角度来讲，Spark基于微批量处理，把流数据看成是一个个小的批处理数据块分别处理，所以延迟性只能做到秒级。而Flink基于每个事件处理，每当有新的数据输入都会立刻处理，是真正的流式计算，支持毫秒级计算。由于相同的原因，Spark只支持基于时间的窗口操作（处理时间或者事件时间），而Flink支持的窗口操作则非常灵活，不仅支持时间窗口，还支持基于数据本身的窗口，开发者可以自由定义想要的窗口操作。

从SQL 功能的角度来讲，Spark和Flink分别提供SparkSQL和Table APl提供SQL

交互支持。两者相比较，Spark对SQL支持更好，相应的优化、扩展和性能更好，而Flink在SQL支持方面还有很大提升空间。

从迭代计算的角度来讲，Spark对机器学习的支持很好，因为可以在内存中缓存中间计算结果来加速机器学习算法的运行。但是大部分机器学习算法其实是一个有环的数据流，在Spark中，却是用无环图来表示。而Flink支持在运行时间中的有环数据流，从而可以更有效的对机器学习算法进行运算。

从相应的生态系统角度来讲，Spark 的社区无疑更加活跃。Spark可以说有着Apache

旗下最多的开源贡献者，而且有很多不同的库来用在不同场景。而Flink由于较新，现阶段的开源社区不如Spark活跃，各种库的功能也不如Spark全面。但是Flink还在不断发展，各种功能也在逐渐完善。

Spark任务提交流程

简述SparkSQL 中RDD、DataFrame、DataSet 三者的区别与

联系

SparkStreaming 有哪几种方式消费Kafka 中的数据，它们

之间的区别是什么？

1. 基于receiver
2. 基于direct

优点：简化并行读取、高性能：spark自己存储offset，不需要zookeeper

Watermark

Mapreduce、spark、Flink的容错机制

项目相关

1. 增量表
2. 拉链表
3. 七天内连续3天活跃用户
4. 连续三周活跃用户

mysql索引原理

b+树

多进程和多线程的优缺点

hiveSQL怎么执行

解析-编译-优化-执行

-

hbase为什么快

region分区

LMS树

blockcache-memstore-hfile

访问一个网址会经过哪些步骤

单核cpu同一时刻能处理多少个进程

Sql优化

1.对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

2.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num is null

最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库.

备注、描述、评论之类的可以设置为 NULL，其他的，最好不要使用NULL。

不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL也包含在内），都是占用 100个字符的空间的，如果是varchar这样的变长字段， null 不占用空间。

可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

select id from t where num = 0

3.应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描。

4.应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num=10 or Name = 'admin'

可以这样查询：

select id from t where num = 10

union all

select id from t where Name = 'admin'

5.in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between 1 and 3

很多时候用 exists 代替 in 是一个好的选择：

select num from a where num in(select num from b)

用下面的语句替换：

select num from a where exists(select 1 from b where num=a.num)

6.下面的查询也将导致全表扫描：

select id from t where name like ‘%abc%’

若要提高效率，可以考虑全文检索。

7.如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

select id from t where num = @num

可以改为强制查询使用索引：

select id from t with(index(索引名)) where num = @num

.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/2 = 100

应改为:

select id from t where num = 100\*2

9.应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where substring(name,1,3) = ’abc’ -–name以abc开头的id

select id from t where datediff(day,createdate,’2005-11-30′) = 0 -–‘2005-11-30’ --生成的id

应改为:

select id from t where name like 'abc%'

select id from t where createdate >= '2005-11-30' and createdate < '2005-12-1'

10.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

12.不要写一些没有意义的查询，如需要生成一个空表结构：

select col1,col2 into #t from t where 1=0

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

create table #t(…)

13.Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志。

14.对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

15.select count(\*) from table；这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。

16.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。

17.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

18.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连 接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

19.尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

20.任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

21.尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

22. 避免频繁创建和删除临时表，以减少系统表资源的消耗。临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件， 最好使用导出表。

23.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

24.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

25.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

26.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

27.与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

29.尽量避免大事务操作，提高系统并发能力。

30.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

数据倾斜

解决spark数据倾斜方法：

1. hive ETL预处理：将数据倾斜提前到ETL阶段，治标不治本
2. 过滤掉少数导致倾斜的key：主要用在产生大量异常数据的情况下
3. 提高shuffle的 并行度：防止多个key在同一task中计算
4. 两阶段聚合（局部聚合+全局聚合）：在key前加上前缀，使key平均，但只适用于reducebykey和groupby这类shuffle中
5. 使用map join：适合小表join大表
6. 采样倾斜key并分拆join操作：将导致倾斜的key单独拆成RDD算子，并加前缀拆分成n个任务，适合只有少数key倾斜的情况

手写wordcount：

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("WordCount")

val sc = new SparkContext(conf)

sc.textFile("/input")

.flatMap(\_.split(" "))

.map((\_,1))

.reduceByKey(\_+\_)

.saveAsTextFile("/output")

sc.stop()