一：Spark程序数据结构的优化

1. Java的对象：对象头是16个字节（例如指向对象的指针等元数据信息），如果对象中只有一个int的property，则此时会占据20个字节，也就是说对象的元数据占用了大部分的空间，所有在封装数据的时候尽量不要使用对象！例如说使用JSON格式来封装数据；
2. Java中基本的数据类型会自动的封箱操作，例如int会自动变成Integer，这回额外增加对象头的空间占用
3. Java的String在实际占用内存方面要额外使用40个字节（String的内部使用char[]来保存字符序列），另外需要注意的是String中的字符是2个字节（UTF-16），如果内部有5个字符的话，实际上会占用50个字节；
4. Java中的集合List、HashMap等等其内部一般使用链表来实现，具体的每个数据使用Entry等，这些也非常消耗内存；
5. 优先使用原生数组，尽可能不要直接去使用ArrayList、HashMap、LinkedList等数据结构，例如说List<Integer> list = new ArrayList<Integer> 需要考虑替换为int[] arrary = new int[];
6. 优先使用String（推荐使用JSON），而不是采用HashMap、List等来封装数据，例如Map<Integer, Worker> workers = new HashMap< Integer, Worker >()，建议使用JSON字符串或者自己构建的String字符串对象，例如“id:name,salary|| id:name,salary|| id:name,salary”；

补充：

1）java对象大小分为shallow size和deep size。Shallow size是指对象自身占用的内存大小，其引用对象的大小不算在内。而deep size，则是自身所占内存大小和其递归引用的所有对象所占内存大小的总和。大多数情况下，你会希望获得一个对象的deep size，但是为了知道这个值，首先要知道怎么算shallow size

没有实例属性的类的内存布局

在Sun JVM中，（除了数组之外的）对象都有两个机器字（words）的头部。第一个字中包含这个对象的标示哈希码以及其他一些类似锁状态和等标识信息，第二个字中包含一个指向对象的类的引用。另外，任何对象都是8个字节为粒度进行对齐的。这就是对象内存布局的第一个规则：

规则1：任何对象都是8个字节为粒度进行对齐的。

比如，如果调用new Object()，由于Object类并没有其他没有其他可存储的成员，那么仅仅使用堆中的8个字节来保存两个字的头部即可。

继承了Object的类的内存布局

除了上面所说的8个字节的头部，类属性紧随其后。属性通常根据其大小来排列。例如，整型（int）以4个字节为单位对齐，长整型（long）以8个字节为单位对齐。这里是出于性能考虑而这么设计的：通常情况下，如果数据以4字节为单位对齐，那么从内存中读4字节的数据并写入到处理器的4字节寄存器是性价比更高的。

为了节省内存，Sun VM并没有按照属性声明时的顺序来进行内存布局。实际上，属性在内存中按照下面的顺序来组织：

1. 双精度型（doubles）和长整型（longs）

2. 整型（ints）和浮点型（floats）

3. 短整型（shorts）和字符型（chars）

4. 布尔型（booleans）和字节型（bytes）

5. 引用类型（references）

内存使用率会通过这个机制得到优化。例如，如下声明一个类：

class MyClass {

       byte a;

       int c;

       boolean d;

       long e;

       Object f;

}

如果JVM并没有打乱属性的声明顺序，其对象内存布局将会是下面这个样子：

[HEADER:  8 bytes]  8

[a:       1 byte ]  9

[padding: 3 bytes] 12

[c:       4 bytes] 16

[d:       1 byte ] 17

[padding: 7 bytes] 24

[e:       8 bytes] 32

[f:       4 bytes] 36

[padding: 4 bytes] 40

此时，用于占位的14个字节是浪费的，这个对象一共使用了40个字节的内存空间。但是，如果用上面的规则对这些对象重新排序，其内存结果会变成下面这个样子：

[HEADER:  8 bytes]  8

[e:       8 bytes] 16

[c:       4 bytes] 20

[a:       1 byte ] 21

[d:       1 byte ] 22

[padding: 2 bytes] 24

[f:       4 bytes] 28

[padding: 4 bytes] 32

这次，用于占位的只有6个字节，这个对象使用了32个字节的内存空间。

规则2：类属性按照如下优先级进行排列：长整型和双精度类型；整型和浮点型；字符和短整型；字节类型和布尔类型，最后是引用类型。这些属性都按照各自的单位对齐。

现在我们知道如何计算一个继承了Object的类的实例的内存大小了。下面这个例子用来做下练习: java.lang.Boolean。这是其内存布局：

[HEADER:  8 bytes]  8

[value:   1 byte ]  9

[padding: 7 bytes] 16

Boolean类的实例占用16个字节的内存！

继承其他类的子类的内存布局

JVM所遵守的下面3个规则用来组织有父类的类的成员。对象内存布局的规则3如下：

规则3：不同类继承关系中的成员不能混合排列。首先按照规则2处理父类中的成员，接着才是子类的成员。

class A {

   long a;

   int b;

   int c;

}

class B extends A {

   long d;

}

类B的实例在内存中的存储如下：

[HEADER:  8 bytes]  8

[a:       8 bytes] 16

[b:       4 bytes] 20

[c:       4 bytes] 24

[d:       8 bytes] 32

规则4：当父类中最后一个成员和子类第一个成员的间隔如果不够4个字节的话，就必须扩展到4个字节的基本单位。

class A {

   byte a;

}

class B {

   byte b;

}

[HEADER:  8 bytes]  8

[a:       1 byte ]  9

[padding: 3 bytes] 12

[b:       1 byte ] 13

[padding: 3 bytes] 16

注意到成员a被扩充了3个字节以保证和成员b之间的间隔是4个字节。这个空间不能被类B使用

规则5：如果子类第一个成员是一个双精度或者长整型，并且父类并没有用完8个字节，JVM会破坏规则2，按照整形（int），短整型（short），字节型（byte），引用类型（reference）的顺序，向未填满的空间填充。

class A {

  byte a;

}

class B {

  long b;

  short c;

  byte d;

}

其内存布局如下：

[HEADER:  8 bytes]  8

[a:       1 byte ]  9

[padding: 3 bytes] 12

[c:       2 bytes] 14

[d:       1 byte ] 15

[padding: 1 byte ] 16

[b:       8 bytes] 24

类A“结束”的地方，JVM没有遵守规则2，而是在长整型之前插入一个短整型和一个字节型成员，这样可以避免浪费3个字节

数组有一个额外的头部成员，用来存放“长度”变量。数组元素以及数组本身，跟其他常规对象同样，都需要遵守8个字节的边界规则

下面是一个有3个元素的字节数组的内存布局：

[HEADER:  12 bytes] 12

[[0]:      1 byte ] 13

[[1]:      1 byte ] 14

[[2]:      1 byte ] 15

[padding:  1 byte ] 16

下面是一个有3个元素的长整型数字的内存布局：

[HEADER:  12 bytes] 12

[padding:  4 bytes] 16

[[0]:      8 bytes] 24

[[1]:      8 bytes] 32

[[2]:      8 bytes] 40

2）Hotspot VM将其堆空间分为三个分代空间：

1. 年轻代（Young Generation）

○     Java应用在分配Java对象时，这些对象会被分配到年轻代堆空间中去

○     这个空间大多是小对象并且会被频繁回收

○     由于年轻代堆空间的垃圾回收会很频繁，因此其垃圾回收算法会更加重视回收效率

2. 年老代（Old Generationn）

○     年轻代堆空间的长期存活对象会转移到（也许是永久性转移）年老代堆空间

○     这个堆空间通常比年轻代的堆空间大，并且其空间增长速度较缓

○     由于大部分JVM堆空间都分配给了年老代，因此其垃圾回收算法需要更节省空间，此算法需要能够处理低垃圾密度的堆空间

3. 持久代（Permanent Generation）

○     存放VM和Java类的元数据（metadata），以及interned字符串和类的静态变量

次收集（Minor GC）和全收集（Full GC）

当这三个分代的堆空间比较紧张或者没有足够的空间来为新到的请求分配的时候，垃圾回收机制就会起作用。有两种类型的垃圾回收方式：次收集和全收集。当年轻代堆空间满了的时候，会触发次收集将还存活的对象移到年老代堆空间。当年老代堆空间满了的时候，会触发一个覆盖全范围的对象堆的全收集。

次收集

当年轻代堆空间紧张时会被触发

相对于全收集而言，收集间隔较短

全收集

当老年代或者持久代堆空间满了，会触发全收集操作

可以使用System.gc()方法来显式的启动全收集

全收集一般根据堆大小的不同，需要的时间不尽相同，但一般会比较长。不过，如果全收集时间超过3到5秒钟，那就太长了[1]

全收集通常时间最长，并且是程序无法延迟执行或者无法达到吞吐量目标的主因。GC的目标是去减少程序运行过程中垃圾回收的频率。为了达到这个目的，可以从这两方面入手：

从系统方面考虑：

○    尽量采用大堆，但是不要大到需要系统从磁盘上“换”页。一般而言，可用的RAM(没有被系统进程占用的)的80%都应该分配给JVM。

○    Java堆空间越大，垃圾回收器和java应用在吞吐量（throughput）和延迟执行（latency）方面的效果越好。

从应用方面考虑：

○    减少对象分配（object allocations）操作，或者采用对象保留（object retention）方式有助于减小存活的数据大小，这也可以反过来帮助垃圾回收做的更好。

二：Spark内存消耗诊断

1. JVM自带众多内存消耗诊断的工具例如JMap、JSonsole等，第三方IBM JVM Profile Tools等；
2. 在开发、测试、生产环境下用的最对的是日志？! Driver产生的日志！！！最简单也是最有效的方式就是调用RDD.cache，当进行cache操作的时候Driver上的BlockManagerMaster会记录该信息并写进日志中！！！

三：persist和checkpoint

1. 当反复使用某个（些）RDD的时候强烈建议使用persist来对数据进行缓存（MEMORY\_AND\_DISK）;

*/\*\* Persist this RDD with the default storage level (*`*MEMORY\_ONLY*`*). \*/*def persist(): this.type = persist(StorageLevel.**MEMORY\_ONLY**)  
  
*/\*\* Persist this RDD with the default storage level (*`*MEMORY\_ONLY*`*). \*/*def cache(): this.type = persist()

总结：

1）RDD的cache()方法其实调用的就是persist方法，缓存策略均为MEMORY\_ONLY；

2）可以通过persist方法手工设定StorageLevel来满足工程需要的存储级别；

3）cache或者persist并不是action；

1. 如果某个步骤的RDD计算特别耗时或者经历了很多步骤的计算，如果数据丢失则重新计算的代价特别大，此时考虑使用checkpoint，因为checkpoint是把数据写入HDFS，天然具有高可靠性；

def checkpoint(): Unit = RDDCheckpointData.synchronized {  
 // NOTE: we use a global lock here due to complexities downstream with ensuring  
 // children RDD partitions point to the correct parent partitions. In the future  
 // we should revisit this consideration.  
 if (context.**checkpointDir**.isEmpty) {  
 throw new SparkException("Checkpoint directory has not been set in the SparkContext")  
 } else if (**checkpointData**.isEmpty) {  
 **checkpointData** = Some(new ReliableRDDCheckpointData(this))  
 }  
}

一般在进行checkpoint方法调用前通过都要进行persist来把当前RDD的数据持久化到内存或者磁盘上，这是因为checkpoint是Lazy级别，必须有Job的执行且在Job执行完成后才会从后往前回溯哪个RDD进行了Checkpoint标记，然后对该标记了要进行Checkpoint的RDD新启动一个Job执行具体的Checkpoint的过程。