> 主动做工，一旦松懈，熵增立马开始

RDD缓存和checkpoint

RDD缓存

> RDD通过persist方法或cache方法可以将前面的计算结果缓存，默认情况下 persist() 会把数据以序列化的形式缓存在 JVM 的堆空间中。但是并不是这两个方法被调用时立即缓存，而是**触发后面的action时**，该RDD将会被缓存在计算节点的内存中，并供后面重用。

```
/** 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()
```

通过查看源码发现**cache最终也是调用了persist方法**，默认的存储级别都是仅在内存存储一份，Spark的存储级别还有好多种，存储级别在object StorageLevel中定义的。在存储级别的末尾加上“_2”来把持久化数据存为两份

```
object StorageLevel {
  val NONE = new StorageLevel(false, false, false, false)
  val DISK_ONLY = new StorageLevel(true, false, false, false)
  val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
  val MEMORY_ONLY = new StorageLevel(false, true, false, true)
  val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
  val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
  val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
  val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
  val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
  val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
  val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
  val OFF_HEAP = new StorageLevel(false, false, true, false)
```

级　别	使用的空间	CPU时间	是否在内存中	是否在磁盘上	备　注
MEMORY_ONLY	高	低	是	否	
MEMORY_ONLY_SER	低	高	是	否	
MEMORY_AND_DISK	高	中等	部分	部分	如果数据在内存中放不下，则溢写到磁盘上
MEMORY_AND_DISK_SER	低	高	部分	部分	如果数据在内存中放不下，则溢写到磁盘上。在内存中存放序列化后的数据
DISK_ONLY	低	高	否	是	

> 缓存有可能丢失，或者存储存储于内存的数据由于内存不足而被删除，RDD的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于RDD的一系列转换，**丢失的数据会被重算**(因为RDD的血统)，由于RDD的各个Partition是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部Partition。

（1）创建一个RDD

```
scala> val rdd = sc.makeRDD(Array("zhutian"))
rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[19] at makeRDD at
<console>:25
```

（2）将RDD转换为携带当前时间戳不做缓存

```
scala> val nocache = rdd.map(_.toString+System.currentTimeMillis)
nocache: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[20] at map at
<console>:27
```

（3）多次打印结果

```
scala> nocache.collect
res0: Array[String] = Array(zhutian1538978275359)

scala> nocache.collect
res1: Array[String] = Array(zhutian1538978282416)

scala> nocache.collect
res2: Array[String] = Array(zhutian1538978283199)
```

（4）将RDD转换为携带当前时间戳并做缓存

```
scala> val cache =  rdd.map(_.toString+System.currentTimeMillis).cache
cache: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[21] at map at
<console>:27
```

（5）多次打印做了缓存的结果，后面结果都是从缓存中拿出的

```
scala> cache.collect
res3: Array[String] = Array(atguigu1538978435705)

scala> cache.collect
res4: Array[String] = Array(atguigu1538978435705)

scala> cache.collec
res5: Array[String] = Array(atguigu1538978435705)
```

RDD Checkpoint

> 奇怪，为什么有了缓存机制还需要，检查点机制呢？不都是保存数据快照吗？
>
> 两者都是保存数据没有错，但是缓存机制会将RDD的血缘关系保存下来，如果RDD的血缘过长会导致数据丢失时候恢复RDD的成本过高，所以这就需要一种机制，直接在长血缘中每隔一段时间保存中间的RDD，那么恢复数据的时候就不是从头开始恢复，而是中间的RDD开始恢复，降低了容错成本，这就是检查点机制

> 为当前RDD设置检查点。该函数将会创建一个二进制的文件，并存储到checkpoint目录中，该目录是用SparkContext.setCheckpointDir()设置的。在checkpoint的过程中，**该RDD的所有依赖于父RDD中的信息将全部被移除。**对RDD进行checkpoint操作并不会马上被执行，**必须执行Action操作才能触发。**

（1）设置检查点

```
scala> sc.setCheckpointDir("hdfs://hadoop102:9000/checkpoint")
```

（2）创建一个RDD

```
scala> val rdd = sc.parallelize(Array("zhutiansama"))

rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[14] at parallelize
at <console>:24
```

（3）将RDD转换为携带当前时间戳并做checkpoint

```
scala> val ch = rdd.map(_+System.currentTimeMillis)
ch: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[16] at map at
<console>:26

scala> ch.checkpoint
```

（4）多次打印结果

```
scala> ch.collect
res55: Array[String] = Array(atguigu1538981860336)

scala> ch.collect
res56: Array[String] = Array(atguigu1538981860504)

scala> ch.collect
res57: Array[String] = Array(atguigu1538981860504)

scala> ch.collect
res58: Array[String] = Array(atguigu1538981860504)
```