RDD

Introducción a los RDD

Un conjunto de datos distribuido resistente (RDD)es una colección de objetos distribuida inmutable. Los RDD de Spark son resistentes o tolerantes a errores, lo que permite a Spark recuperar el RDD en caso de errores. La inmutabilidad hace que los RDD sean de solo lectura una vez creados. Las transformaciones permiten que las operaciones en el crear un nuevo RDD, pero el RDD original nunca se modifica una vez creado. Esto hace que los RDD sean inmunes a las condiciones de la carrera y otros problemas de sincronización.

La naturaleza distribuida de los RDD funciona porque un RDD solo contiene una referencia a los datos, mientras que los datos reales se encuentran dentro de las particiones entre los nodos del clúster.

Nota

Conceptualmente, un RDD es una colección distribuida de elementos distribuidos en varios nodos del clúster. Podemos simplificar un RDD para

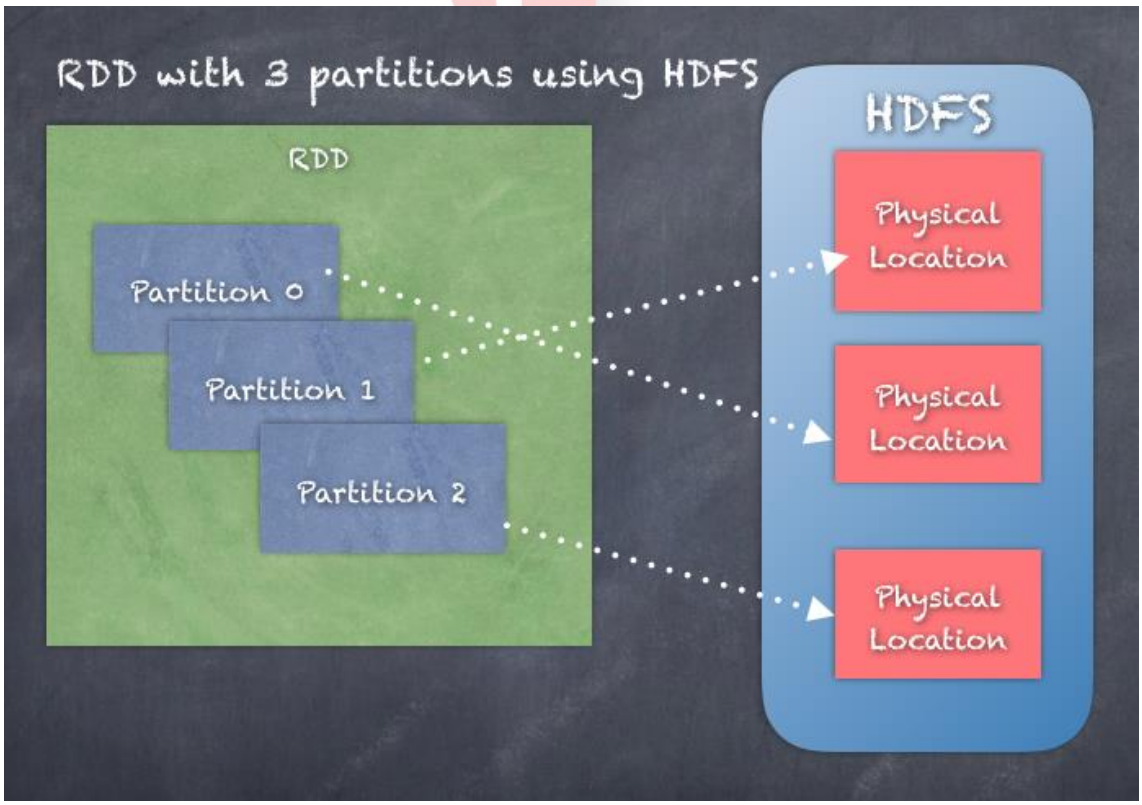
comprender mejor pensando en un RDD como una gran variedad de enteros distribuidos entre máquinas.

Un RDD es en realidad un conjunto de datos que se ha particionado en el clúster y los datos particionados podrían ser de HDFS (Hadoop Distributed File System),tabla HBase, tabla Cassandra, Amazon S3.

Internamente, cada RDD se caracteriza por cinco propiedades principales:

- Una lista de particiones
- Una función para calcular cada división
- Una lista de dependencias en otros RDD
- Opcionalmente, un particionador para RDD de clave-valor (por ejemplo, para decir que el RDD tiene particiones hash)
- Opcionalmente, una lista de ubicaciones preferidas para calcular cada división (por ejemplo, ubicaciones de bloque para un archivo HDFS)

Eche un vistazo al siguiente diagrama:



Dentro del programa, el controlador trata el objeto RDD como un identificador de los datos distribuidos. Es análogo a un puntero a los datos, en lugar de los datos reales utilizados, para llegar a los datos reales cuando es necesario.

El RDD utiliza de forma predeterminada el particionador hash para particionar los datos en todo el clúster. El número de particiones es independiente del número de nodos del clúster. Podría suceder muy bien que un solo nodo en el clúster tiene varias particiones de datos. El número de particiones de datos que existen depende totalmente de cuántos nodos tiene el clúster y del tamaño de los datos. Si observa la ejecución de tareas en los nodos, una tarea que se ejecuta en un ejecutor en el nodo de trabajo podría procesar los datos que están disponibles en el mismo nodo local o en un nodo remoto. Esto se denomina la localidad de los datos y la tarea de ejecución elige la mayor cantidad de datos locales posible.

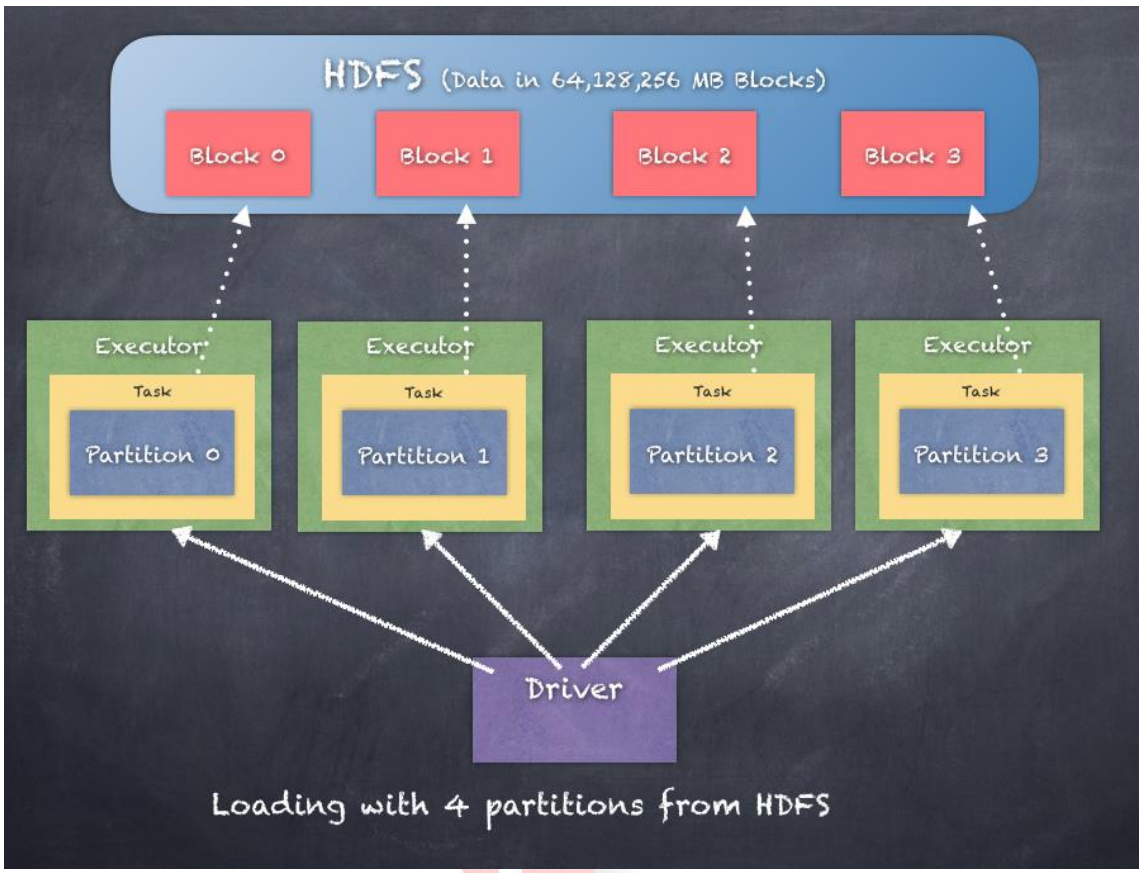
Nota

La localidad afecta significativamente al rendimiento de su trabajo. El orden de preferencia de la localidad por defecto se puede mostrar como `PROCESS_LOCAL > NODE_LOCAL > NO_PREF > RACK_LOCAL > ANY`

No hay ninguna garantía de cuántas particiones podría obtener un nodo. Esto afecta a la eficiencia de procesamiento de cualquier ejecutor, ya que si tiene demasiadas particiones en un único nodo que procesan varias particiones, el tiempo necesario para procesar todas las particiones también crece, sobrecargando los núcleos en el ejecutor y ralentizando así toda la etapa de procesamiento, lo que ralentiza directamente todo el trabajo. De hecho, la partición es uno de los principales factores de ajuste para mejorar el rendimiento de un trabajo de Spark. Consulte el siguiente comando:

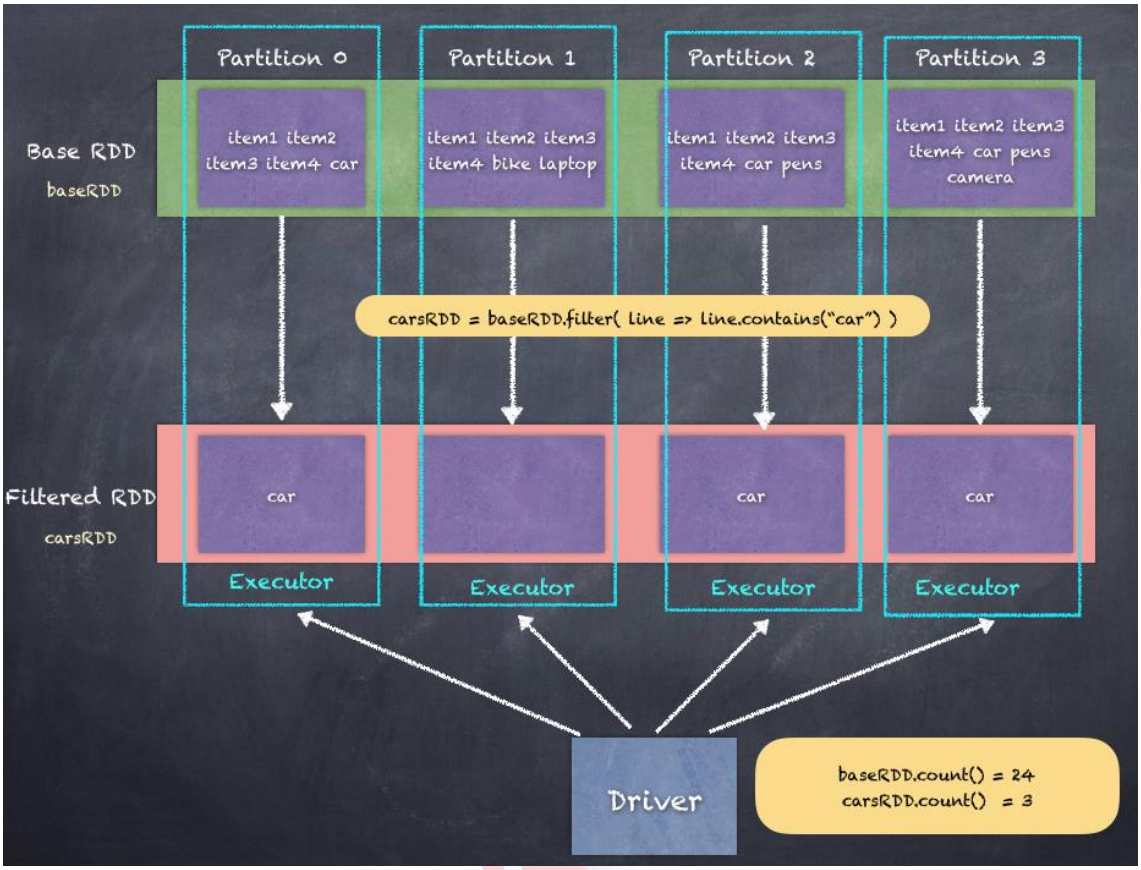
```
classRDD[T:ClassTag]
```

Echemos un vistazo más a lo que se verá un RDD cuando carguemos datos. A continuación, se muestra un ejemplo de cómo Spark utiliza diferentes trabajadores para cargar diferentes particiones o divisiones de los datos:



Independientemente de cómo se cree el RDD, el RDD inicial normalmente se denomina RDD base y los RDD posteriores creados por las distintas operaciones forman parte del linaje de los RDD. Este es otro aspecto muy importante para recordar, ya que el secreto de la tolerancia a errores y la recuperación es que el conductor mantiene el linaje de los RDD y puede ejecutar el linaje para recuperar cualquier bloque perdido de los RDD.

A continuación, se muestra un ejemplo que muestra varios RDD creados como resultado de operaciones. Comenzamos con el RDD base, que tiene 24 artículos y deriva otro carsRDD RDD que contiene sólo los elementos (3) que coinciden con los coches:



Nota

El número de particiones no cambia durante estas operaciones, ya que cada ejecutor aplica la transformación de filtro en memoria, generando una nueva partición RDD correspondiente a la partición RDD original.

Creación de RDD

Un RDD es el objeto fundamental utilizado en Apache Spark. Son colecciones que representan conjuntos de datos y tienen la capacidad integrada de confiabilidad y recuperación de errores. Por naturaleza, los RDD crean nuevos RDD en cualquier operación, como la transformación o la acción. También hemos visto en el capítulo anterior algunos detalles sobre cómo se pueden crear RDD y qué tipo de operaciones se pueden aplicar a rdDs.

Un RDD se puede crear de varias maneras:

- Paralelizar una colección
- Lectura de datos de una fuente externa
- Transformación de un RDD existente
- Streaming API

Paralelizar una colección

Paralelizar una colección se puede hacer llamando a la colección del programa de controlador. El controlador, cuando intenta paralelizar una colección, divide la colección en particiones y distribuye las particiones de datos entre el clúster.parallelize()

A continuación se muestra un RDD para crear un RDD a partir de una secuencia de números mediante SparkContext y la función. La función esencialmente divide la secuencia de números en una colección distribuida también conocida como RDD.parallelize()parallelize()

```
scala> val rdd_one = sc.parallelize(Seq(1,2,3))

rdd_one:           org.apache.spark.rdd.RDD[Int]           =
ParallelCollectionRDD[0] at parallelize at <console>:24



scala> rdd_one.take(10)

res0: Array[Int] = Array(1, 2, 3)
```

Lectura de datos de una fuente externa

Un segundo método para crear un RDD es mediante la lectura de un origen distribuido externo como Amazon S3, Cassandra, HDFS, etc. Por ejemplo, si está creando un RDD a partir de HDFS, los bloques distribuidos en HDFS son leídos por los nodos individuales del clúster de Spark.

Cada uno de los nodos del clúster de Spark esencialmente está realizando sus propias operaciones de entrada-salida y cada nodo lee de forma independiente uno o más bloques de los bloques HDFS. En general, Spark hace el mejor

esfuerzo para poner tanto RDD como sea posible en la memoria. Existe la capacidad de los datos para reducir las operaciones de entrada y salida habilitando los nodos del clúster de chispas para evitar operaciones de lectura repetidas, por ejemplo, desde los bloques HDFS, que podrían ser remotos al clúster de Spark. Hay un montón de estrategias de almacenamiento en caché que se pueden utilizar dentro de su programa Spark, que examinaremos más adelante en una sección para el almacenamiento en caché.cache

A continuación, se muestra un RDD de líneas de texto que se cargan desde un archivo de texto mediante el contexto de Spark y la función. La función carga los datos de entrada como un archivo de texto (cada parte terminada de nueva línea se convierte en un elemento en el RDD). La llamada a la función también utiliza automáticamente HadoopRDD (que se muestra en el capítulo siguiente) para detectar y cargar los datos en forma de varias particiones según sea necesario, distribuidos por el clúster.textFile()textFile\n

```
scala> val rdd_two = sc.textFile("wiki1.txt")

rdd_two:   org.apache.spark.rdd.RDD[String]   =   wiki1.txt
MapPartitionsRDD[8] at textFile at <console>:24


scala> rdd_two.count

res6: Long = 9


scala> rdd_two.first

res7: String = Apache Spark provides programmers with an
application programming interface centered on a data
structure called the resilient distributed dataset (RDD), a
read-only multiset of data items distributed over a cluster
of machines, that is maintained in a fault-tolerant way.
```

Transformación de un RDD existente

Los RDD, por naturaleza, son inmutables; por lo tanto, sus RDD podrían ser mediante la aplicación de transformaciones en cualquier RDD existente. El filtro es un ejemplo típico de una transformación.

A continuación se muestra una simple transformación y enteros multiplicando cada entero por . Una vez más, usamos la función y paralelizar para crear una secuencia de enteros en un RDD distribuyendo la secuencia en forma de particiones. A continuación, usamos la función para transformar el RDD en otro RDD multiplicando cada número por .rdd2SparkContextmap()2

```
scala> val rdd_one = sc.parallelize(Seq(1,2,3))

rdd_one:          org.apache.spark.rdd.RDD[Int]          =
ParallelCollectionRDD[0] at parallelize at <console>:24


scala> rdd_one.take(10)

res0: Array[Int] = Array(1, 2, 3)


scala> val rdd_one_x2 = rdd_one.map(i => i * 2)

rdd_one_x2:         org.apache.spark.rdd.RDD[Int]          =
MapPartitionsRDD[9] at map at <console>:26


scala> rdd_one_x2.take(10)

res9: Array[Int] = Array(2, 4, 6)
```

Streaming API

Los RDD también se pueden crear a través de la transmisión de spark. Estos RDD se denominan RDD de secuencia (RDD DStream).