
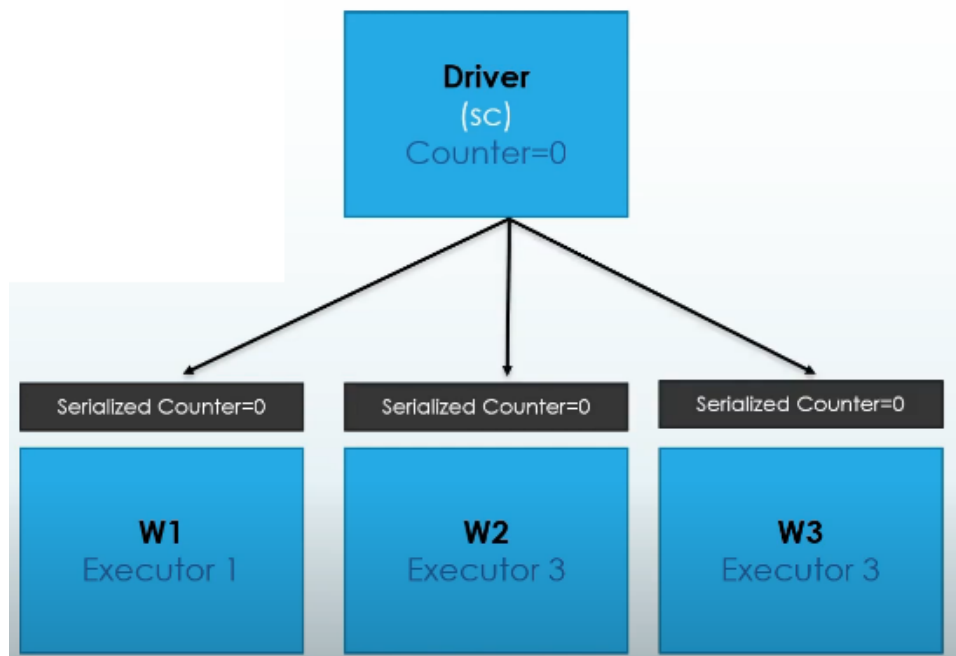


Разделяемые переменные

☰ Tags	
📎 Files & media	

▼ Замыкания (Closure)



Простым языком:

Предположим, над RDD мы хотим сделать **преобразования (transformation) + действие (action)**. Program Driver делит её на задачи (tasks), которые позже будут обрабатываться на executor'ах. Но для обработки им нужна информация также о **переменных** или **методах**, которые мы будем использовать при вычислении. Например, в плохом примере на картинке есть пересенная **counter**, копию которой нужно иметь обязательно. Значит, надо сериализовать эти данные и отправить

каждому исполнителю, и эти данные **называются CLOSURE**. То есть это те переменные и методы, которые необходимы нашему исполнителю для того, чтобы совершить вычисление над RDD.



Closure — это функция или переменная, которая передается в узлы кластера для выполнения задач. Когда вы пишете код на языке программирования, таком как Scala или Python, который использует переменные и функции из окружения, Spark должен отправить этот код на все узлы кластера для выполнения. Эти переменные и функции становятся частью "closure".

```
counter = 0
rdd = sc.parallelize(data)

# Wrong: Don't do this!!
def increment_counter(x):
    global counter
    counter += x
rdd.foreach(increment_counter)

print("Counter value: ", counter)
```



Обрати внимание!

Из этого определения легко догадаться, что исполнитель работает над копией данных, а не с оригиналами. Поэтому нужно всегда иметь это в виду. Например на картинке выше есть **counter**, который (если мы запускаем спарк локально) может находиться на той же JVM что и Program driver, поэтому значение будет меняться. А если у нас кластер — то переменная не изменится, и все будет криво работать.

Как работает closure:

1. **Локальные переменные:** Если вы используете переменные, которые находятся вне контекста вашей функции (например, переменные из основного кода), Spark должен отправить их на каждый узел, чтобы эти переменные были доступны при выполнении задачи.
2. **Сериализация:** Closure (включая функции и переменные) сериализуется и отправляется на каждый рабочий узел (worker node) в кластере. Это значит, что все переменные, использованные в ваших функциях, должны быть сериализуемыми.
3. **Обновления переменных:** Важно помнить, что изменения переменных внутри closure не будут возвращены обратно в драйвер. Это означает, что переменные, используемые в closure, как правило, копируются на каждый узел, и любые изменения на узлах кластера не влияют на оригинальные переменные в драйвере.

Для того, чтобы использовать одну переменную в параллельных операциях Spark предоставляет два ограниченных типа общих переменных: **широковещательные переменные** и **аккумуляторы**.

▼ Аккумуляторы



Аккумуляторы — это переменные, которые позволяют безопасно **накапливать (каунтеры)** значения в ходе распределённых вычислений. Их основная цель — собирать данные с рабочих узлов (worker nodes) обратно в драйвер (driver) без необходимости обмениваться данными между узлами, что позволяет эффективно решать задачи агрегации.

Основные особенности аккумуляторов:

1. **Только для записи:** Аккумуляторы можно изменять **только на рабочих узлах**, но значения можно только накапливать (например, суммировать, добавлять). **Читать** значение такой переменной можно **только на драйвере**.
2. **Ассоциативность и коммутативность:** Аккумуляторы должны быть реализованы с операциями, которые являются **ассоциативными** и **коммутативными**, то есть **порядок выполнения операций не должен влиять на результат**. Это необходимо для корректности при параллельной обработке данных.

3. Типы аккумуляторов:

- **Числовые аккумуляторы:** Это наиболее часто используемый тип аккумуляторов, который суммирует числовые значения.
- **Аккумуляторы коллекций:** Можно использовать для накопления списков или других коллекций данных, но это встречается реже из-за потенциально больших объемов данных.
- **Пользовательские аккумуляторы:** В Spark можно создавать собственные аккумуляторы, например для подсчета специфических метрик или данных.

4. **Ограниченное использование:** Аккумуляторы предназначены для агрегирования информации, но не подходят для передачи данных между узлами или для логики управления, поскольку изменение их значений на рабочих узлах не гарантировано при сбоях задач.

```
from pyspark import SparkContext

sc = SparkContext("local", "Accumulator Example")

# Создаем аккумулятор
accum = sc.accumulator(0)

def add_to_accum(x):
    global accum
    accum += x

# Применяем функцию к элементам RDD
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.foreach(add_to_accum)

# Выводим результат
print(f"Сумма значений: {accum.value}")
```

Когда использовать/не использовать аккумуляторы?

- **Аккумуляторы** лучше всего работают в **действиях** (actions), таких как `foreach()`. Причина этого в том, что действия выполняются один раз на всех данных, и это позволяет избежать возможных проблем с некорректным изменением значений аккумуляторов.
- В `map()` аккумулятор может работать криво и некорректно. **Преобразования плохая идея для аккумуляторов.**

Почему? Потому что если она начнет сбоить (например, будет ошибка в задаче и Spark попытается сделать её еще раз или мы отдадим задаче при крахе executor'a или при спекулятивном исполнении). Он **КАЖДЫЙ РАЗ будет менять наш аккумулятор** и значение может быть кривым, ибо вместо 1 раза мы сделали манипуляцию с аккумулятором много раз.

Зачем нужны аккумуляторы:

1. **Сбор статистики:** Например, вы можете использовать аккумуляторы для подсчета количества записей, которые не прошли фильтрацию, или для отслеживания ошибок обработки.
2. **Мониторинг прогресса:** Аккумуляторы полезны для мониторинга выполнения задач без изменения самих данных, например для отслеживания прогресса обработки.

Типы данных в аккумуляторах Spark

Spark поддерживает аккумуляторы только для **числовых типов данных** (например, `Float`, `Double`, `Integer`). Однако, при необходимости, можно создать кастомный аккумулятор, расширив класс `AccumulatorParam` или используя новый API для аккумуляторов — `AccumulatorV2`. Это позволяет реализовать не только простое увеличение числового значения, но и другие операции, если они обладают свойствами **коммутативности** и **ассоциативности** (порядок выполнения не влияет на результат).

Пример: вместо суммирования можно реализовать аккумулятор, который сохраняет **максимальное значение** среди всех элементов.

Новый API для аккумуляторов в Spark 2.0: AccumulatorV2

В версии Spark 2.0 был представлен новый API для аккумуляторов — `AccumulatorV2`. Старый API (`Accumulator`) всё еще поддерживается, но **помечен как устаревший**.

(deprecated).

- **AccumulatorV2** — это абстрактный класс, который необходимо расширить. При этом нужно определить три ключевых метода:
 - `add` : операция добавления значений в аккумулятор.
 - `merge` : объединение значений аккумуляторов.
 - `isZero` : проверка, является ли текущее состояние аккумулятора "нулевым".
 - Также важно указать, какое значение будет использоваться как "ноль".

Пример реализации можно найти в исходном коде Spark в классе [CollectionAccumulator](#).

Сравнение старого и нового API: accumulator vs AccumulatorV2

Ранее, в Spark, существовали два API для работы с аккумуляторами:

1. **Accumulable** — когда **типы ввода и вывода различаются**.
2. **Accumulator** — когда **типы ввода и вывода одинаковы** (то есть `Accumulable<T, T>`).

Для создания `Accumulable<OUT, IN>` необходимо было определить две операции:

- `merge` — объединение двух значений типа OUT (например, суммирование двух значений).
- `add` — добавление значения типа IN в OUT.
- Также требовалось определить **нулевое значение** для типа OUT.

С появлением **AccumulatorV2** необходимость создания отдельных классов для `Accumulable` исчезла, и теперь все аккумуляторы могут быть реализованы через `AccumulatorV2`, что **значительно упрощает процесс разработки**. В отличие от старого API, в новом API достаточно **зарегистрировать** аккумулятор в `SparkContext` с уникальным именем, и Spark автоматически отслеживает его состояние.

Преимущества нового API: унифицированный подход, который требует только определения ключевых операций (`add` , `merge`) и регистрации аккумулятора в `SparkContext` .

▼ Широковещательные переменные (Broadcast variables)



Широковещательные переменные — это переменные, которые **доступны для чтения (и только для чтения) во всех исполнителях** (executors), участвующих в выполнении приложения Spark. Они **кэшируются** и доступны для использования всеми задачами (tasks) на исполнителях. Эти переменные **отправляются исполнителям только один раз**, что предотвращает избыточные передачи данных и делает их доступными для всех задач, выполняемых в рамках этого приложения..

Когда использовать broadcast переменные?

Широковещательные переменные полезны, когда нужно эффективно передать небольшой объем данных, которые часто используются в задачах. Например, если у вас есть таблица с "кодами стран" и соответствующими названиями стран, и вы хотите объединить их с другим набором данных, используя операцию **join**, это может стать **медленным и затратным** процессом. Операция join требует **shuffle** — перестановки данных между узлами кластера, что может значительно замедлить выполнение задач и увеличить нагрузку на сеть.

Вместо того чтобы передавать данные для каждого join, можно **смапить** данные в виде "код страны: название страны" и использовать их в качестве **broadcast переменной**. В этом случае переменная будет **однократно отправлена на каждый исполнитель** и **закэширована**, что значительно ускорит выполнение задач, так как каждому исполнителю потребуется загрузить данные только один раз. Например, если у вас 100 задач, вместо 100 пересылок данных для каждой задачи будет только 10 пересылок для 10 исполнителей. Это уменьшает объем передаваемых данных по сети и ускоряет выполнение.

Особенности использования broadcast переменных:

- **Размер данных:** Broadcast переменные предназначены для **небольших объемов данных** (в мегабайтах, обычно до 10 мб), которые могут быть эффективно

кэшированы на каждом исполнителе. Для больших наборов данных (в **гигабайтах**) использование broadcast переменных может быть неэффективным и приводить к проблемам с памятью (например, кэш может "лопнуть").

- Также широковещательная переменная должна поместиться как и в оперативную память исполнителя, так и драйвера.
- **Оптимизация производительности:** Broadcast переменные помогают избежать избыточных пересылок данных и reduce (уменьшают) shuffle, что значительно ускоряет задачи, особенно при повторном использовании одной и той же информации в разных частях приложения.

Пример сценария:

Допустим, у вас есть список кодов стран и их названий. Вместо того чтобы делать shuffle и пересылать этот список каждый раз при выполнении join, вы можете сделать broadcast этого списка и разослать его на все узлы кластера:

```
# Создаем broadcast переменную с данными
country_map = sc.broadcast({"US": "United States", "FR": "France", "IN": "India"})

# Используем ее внутри RDD
rdd.map(lambda x: (x, country_map.value.get(x))).collect()
```

В этом примере `country_map` отправляется на каждый исполнитель только один раз и затем доступен для всех задач без необходимости повторной передачи данных.

Когда не следует использовать:

- Не используйте broadcast переменные для **больших наборов данных** (например, в гигабайтах), так как это может привести к нехватке памяти на узлах исполнителей.
- **Нерационально** использовать broadcast переменные для данных, которые редко используются или не используются повторно в разных задачах приложения.

Таким образом, **broadcast переменные** являются мощным инструментом для оптимизации производительности Spark-приложений, если правильно применять их для небольших, часто используемых данных.

