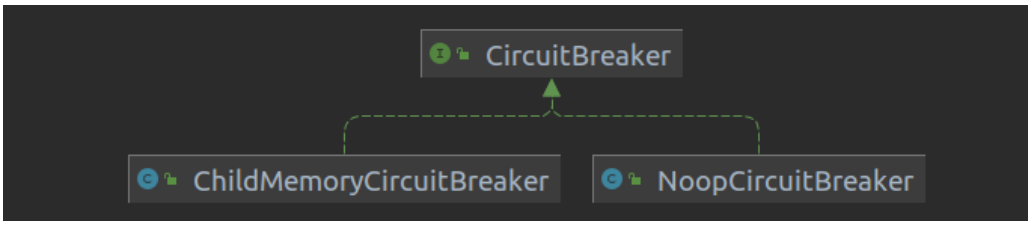
熔断器原理

目的

为了更好理解熔断器是怎么工作的， 从而正确配置熔断器，甚至改造熔断器以达到避免节点OOM的目的。

熔断器监控哪些内存，怎么监控的，还有哪些不能监控到的？

CircuitBreaker



属性分析

熔断器类型

`Type`：枚举类型，表示熔断器的种类。包括MEMORY（常规或子内存熔断器）、PARENT（层次熔断器服务的特殊父类型）和NOOP（每个操作都是空操作，它永远不会触发熔断）。

- String PARENT = "parent";
  - 父熔断器是所有后续熔断器之和。这样，我们可以在为所有熔断器设置“总”限制的同时，允许单个熔断器拥有大量可用内存。注意，它不是一个“真正的”熔断器，因为它不能自己进行加或减操作。
- String FIELDDATA = "fielddata";
  - fielddata熔断器追踪用于fielddata（在字段上）的数据，以及用于父/子查询的id缓存。
- String REQUEST = "request";
  - request熔断器追踪用于特定请求的内存。这包括用于诸如基数聚合之类的事情的分配，以及在聚合请求中使用的桶的数量。通常在请求完成后，添加到这个熔断器的量会被释放。
- String IN_FLIGHT_REQUESTS = "in_flight_requests";
  - 在途请求熔断器追踪为网络层的读写请求分配的字节数。
- String ACCOUNTING = "accounting";
  - accounting熔断器追踪在内存中持有的与请求生命周期无关的事物。这包括Lucene用于段的内存。

Durability

表示触发熔断器后的修复方式。TRANSIENT表示熔断器在触发后会自动恢复，PERMANENT表示需要人工干预来恢复熔断器。

接口分析

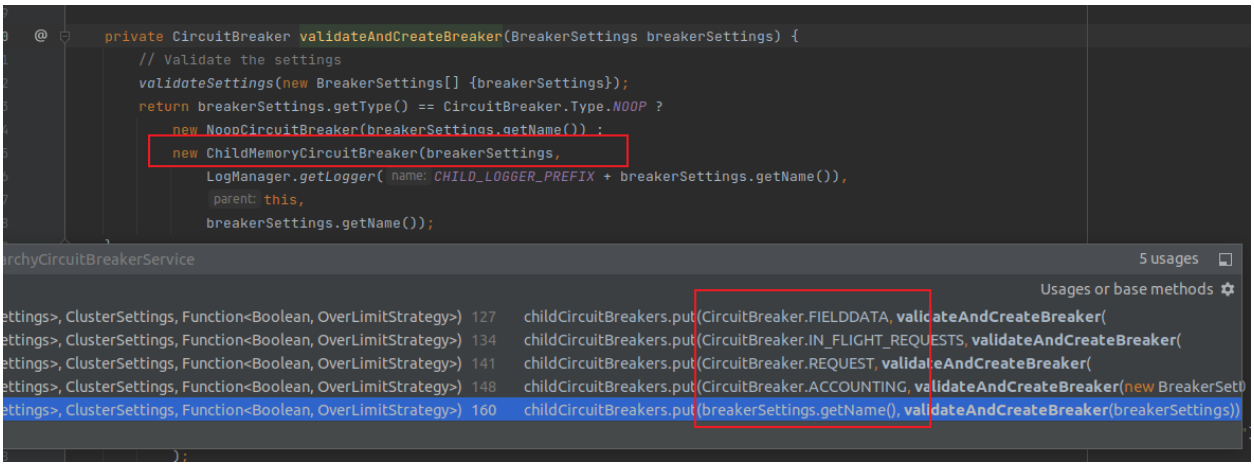
```
/**
     * Trip the circuit breaker
     * 触发熔断器
     * @param fieldName name of the field responsible for tripping the breaker
     * 触发熔断器的字段的名称
     * @param bytesNeeded bytes asked for but unable to be allocated
```

```
     * 请求的但无法分配的字节大小
     */
    void circuitBreak(String fieldName, long bytesNeeded);

    /**
     * add bytes to the breaker and maybe trip
     * 向熔断器添加字节，可能会触发熔断
     * @param bytes number of bytes to add
     * 要添加的字节数
     * @param label string label describing the bytes being added
     * 描述正在添加的字节的字符串标签
     * @return the number of "used" bytes for the circuit breaker
     * 熔断器的“已使用”字节的数量
     */
    double addEstimateBytesAndMaybeBreak(long bytes, String label) throws CircuitBreakingException;

    /**
     * Adjust the circuit breaker without tripping
     * 调整熔断器，但不触发熔断
     * @param bytes number of bytes to add
     * 要添加的字节数
     */
    long addWithoutBreaking(long bytes);

    /**
     * @return the currently used bytes the breaker is tracking
     * 返回熔断器当前正在追踪的已使用字节数
     */
    long getUsed();

    /**
     * @return maximum number of bytes the circuit breaker can track before tripping
     * 返回熔断器在触发之前可以追踪的字节数的最大值
     */
    long getLimit();

    /**
     * @return overhead of circuit breaker
     * 返回熔断器的开销
     */
    double getOverhead();

    /**
     * @return the number of times the circuit breaker has been tripped
     * 返回熔断器已被触发的次数
     */
    long getTrippedCount();

    /**
     * @return the name of the breaker
     * 返回熔断器的名称
     */
    String getName();

    /**
     * @return whether a tripped circuit breaker will reset itself (transient) or requires manual intervention (permanent).
     * 返回触发的熔断器是否会自我重置（暂时的）或者需要人工干预（永久的）。
     */
    Durability getDurability();

    /**
     * sets the new limit and overhead values for the circuit breaker.
     * 为熔断器设置新的限制值和开销值。
     * The resulting write should be readable by other threads.
     * 其他线程应能读取结果写入。
     * @param limit the desired limit
     * 需要设置的限制值
     * @param overhead the desired overhead constant
     * 需要设置的开销常数
     */
```

```
    void setLimitAndOverhead(long limit, double overhead);
}
```

ChildMemoryCircuitBreaker

NoopCiruitBroker是永远不熔断，不分析。我们看ChildMemoryCircuitBreaker. 这个类在增加内存时会去check parent的限制。以上说的REQUEST等熔断器类型都是这个对象。



属性分析

```
// 熔断器的当前限制和开销。 对应着limit和over的配置
private volatile LimitAndOverhead limitAndOverhead;
// 人工恢复还是自动
private final Durability durability;
// 熔断器当前使用字节量
private final AtomicLong used;
// 熔断次数
private final AtomicLong trippedCount;
private final Logger logger;
// 层级熔断服务， 这个熔断器在增加值时会检查parent。
private final HierarchyCircuitBreakerService parent;
private final String name;
```

addEstimateBytesAndMaybeBreak 增加预估字节判断可能熔断

1. 获取当前熔断器的限制和开销的当前值。
2. 如果内存字节限制设置为0，立即触发熔断器。
3. 如果内存字节限制设置为-1，我们可以通过使用.addAndGet()（因为我们不必检查限制）来进行一些优化，这使得RamAccountingTermsEnum情况更快。
4. 否则，我们计算新的已使用字节，考虑到开销和内存字节限制。
5. 额外地，我们需要检查是否已经超过了父级的限制。

6. 如果父级熔断器被触发，这个熔断器必须再调低，因为分配是"被阻塞"的，但熔断器已经被递增了。
7. 断言新的已使用字节总是大于等于0。

```
/**
 * Add a number of bytes, tripping the circuit breaker if the aggregated
 * estimates are above the limit. Automatically trips the breaker if the
 * memory limit is set to 0. Will never trip the breaker if the limit is
 * set &lt; 0, but can still be used to aggregate estimations.
 */
@Override
public double addEstimateBytesAndMaybeBreak(long bytes, String label) throws CircuitBreakingException {
    // 获取当前熔断器的限制和开销值
    final LimitAndOverhead limitAndOverhead = this.limitAndOverhead;
    final long memoryBytesLimit = limitAndOverhead.limit;
    final double overheadConstant = limitAndOverhead.overhead;

    // 如果内存限制设为0，立即触发熔断
    if (memoryBytesLimit == 0) {
        circuitBreak(label, bytes);
    }

    long newUsed;
    // 如果内存限制设为-1，无需检查限制，直接使用addAndGet()方法，这在某些情况下能加速操作
    if (memoryBytesLimit == -1) {
        // 这里会直接增加
        newUsed = noLimit(bytes, label);
    } else {
        // 如果存在内存限制，我们需要考虑开销并检查是否超出限制
        newUsed = limit(bytes, label, overheadConstant, memoryBytesLimit);
    }

    // 还需检查我们是否超出了父级熔断器的限制,这里将要申请的内存 * 熔断器自身的overhead
    try {
        parent.checkParentLimit((long) (bytes * overheadConstant), label);
    } catch (CircuitBreakingException e) {
        // 如果触发了父级熔断器，我们需要回滚本熔断器的计数，因为尽管申请的内存被阻塞了，但我们已经增加了计数
        this.addWithoutBreaking(-bytes);
        throw e;
    }

    // 断言新的使用量不应为负数
    assert newUsed >= 0 : "Used bytes: [" + newUsed + "] must be >= 0";

    // 返回到目前为止的"使用"字节数
    return newUsed;
}
```

limit

修改used是一个原子类，我们这里采用乐观锁compareAndSet来处理并发问题, 并且注意，这里会会让新使用的内存乘以overheadConstat。 也就是你配置的over

```
private long limit(long bytes, String label, double overheadConstant, long memoryBytesLimit) {
    long newUsed;
    long currentUsed;

    // 这是一个乐观锁的使用情况，尝试多次操作直到成功
    do {
        // 获取当前已使用的内存
        currentUsed = this.used.get();
        // 计算添加后的新使用量
        newUsed = currentUsed + bytes;
        // 考虑到开销，计算新使用量的估算值
        long newUsedWithOverhead = (long) (newUsed * overheadConstant);
        // 如果启用了trace日志，则记录添加操作的详细信息
```

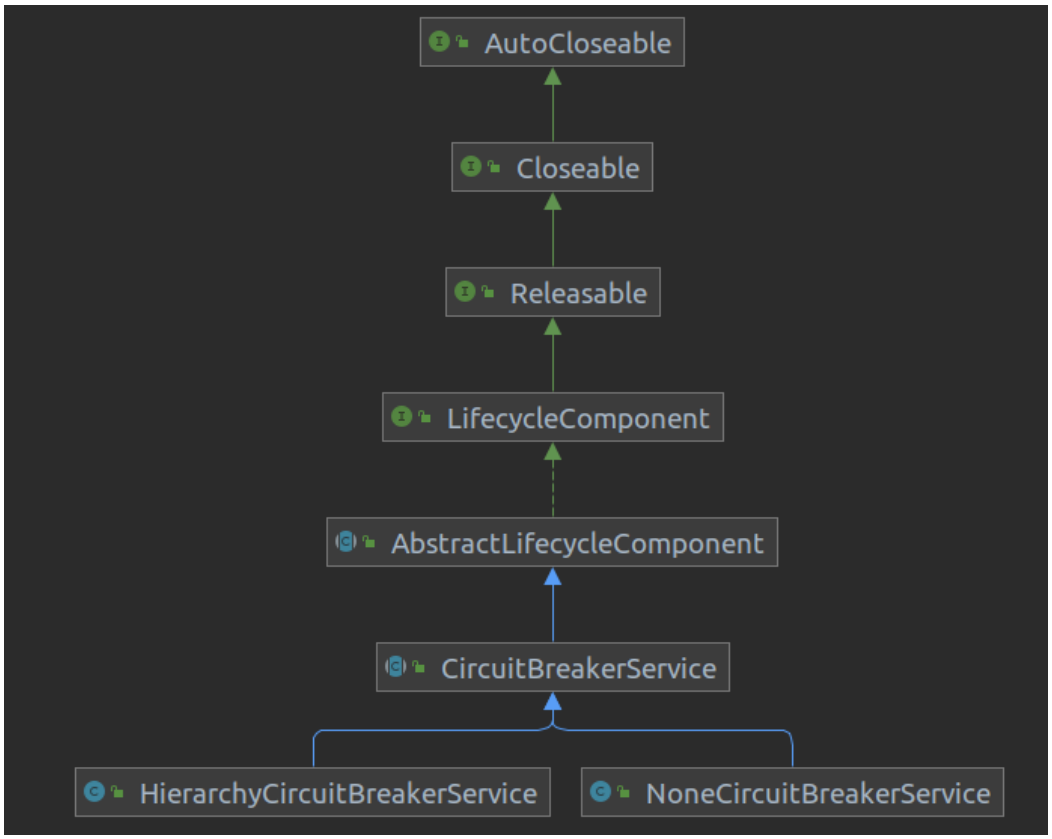
```
        if (logger.isTraceEnabled()) {
            logger.trace("[{}] Adding [{}][{}] to used bytes [new used: [{}], limit: {} [{}], estimate: {} [{}]]",
                    this.name,
                    new ByteSizeValue(bytes), label, new ByteSizeValue(newUsed),
                    memoryBytesLimit, new ByteSizeValue(memoryBytesLimit),
                    newUsedWithOverhead, new ByteSizeValue(newUsedWithOverhead));
        }

        // 如果新使用量的估算值超过了内存限制，则触发熔断
        if (memoryBytesLimit > 0 && newUsedWithOverhead > memoryBytesLimit) {
            logger.warn("[{}] New used memory {} [{}] for data of [{}] would be larger than configured breaker: {} [{}],
breaking",
                    this.name,
                    newUsedWithOverhead, new ByteSizeValue(newUsedWithOverhead), label,
                    memoryBytesLimit, new ByteSizeValue(memoryBytesLimit));
            circuitBreak(label, newUsedWithOverhead);
        }

        // 尝试更新使用量，如果在此期间使用量已被其他线程修改，则再次尝试
    } while (!this.used.compareAndSet(currentUsed, newUsed));

    // 返回新的使用量
    return newUsed;
}
```

触发熔断 circuitBreak

这里简单将熔断次数+1， 然后打印日志。抛出CircuitBreakingException

```
@Override
    public void circuitBreak(String fieldName, long bytesNeeded) {
        final long memoryBytesLimit = this.limitAndOverhead.limit;
        this.trippedCount.incrementAndGet();
        final String message = "[" + this.name + "] Data too large, data for [" + fieldName + "]" +
                " would be [" + bytesNeeded + "/" + new ByteSizeValue(bytesNeeded) + "]" +
                ", which is larger than the limit of [" +
                memoryBytesLimit + "/" + new ByteSizeValue(memoryBytesLimit) + "]";
        logger.debug(() -> new ParameterizedMessage("{}", message));
        throw new CircuitBreakingException(message, bytesNeeded, memoryBytesLimit, durability);
    }
```

CircuitBreakerService



CircuitBreakerService 是对外暴露的，

- public abstract CircuitBreaker getBreaker(String name);
  - 获取指定的熔断器
- public abstract AllCircuitBreakerStats stats();
  - 返回所有熔断器状态
- public abstract CircuitBreakerStats stats(String name);
  - 返回特定熔断器的状态

HierarchyCircuitBreakerService

属性分析

- private final Map<String, CircuitBreaker> breakers;
  - 存储各个熔断器的映射，从这里拿到熔断器，然后进行addEstimateBytesAndMaybeBreak
- private static final MemoryMXBean MEMORY_MX_BEAN = ManagementFactory.getMemoryMXBean();
  - Java管理扩展（Java Management Extensions，JMX）中的MemoryMXBean，用于获取Java虚拟机的内存使用情况
- USE_REAL_MEMORY_USAGE_SETTING

    - 指示是否使用真实的内存使用情况来计算熔断器的阈值
        - indices.breaker.total.use_real_memory
        - 如果为true，就会org.elasticsearch.indices.breaker.HierarchyCircuitBreakerService#checkParentLimit → org.elasticsearch.indices.breaker.HierarchyCircuitBreakerService#memoryUsed中从MemoryMXBean中获取内存，否则就使用预估的。具体见memoryUsed.
- TOTAL_CIRCUIT_BREAKER_LIMIT_SETTING
    - 设置总熔断器的限制，
        - indices.breaker.total.limit
        - 如果没有设置就看indices.breaker.total.use_real_memory ，true就是95， 否则就是70
- 一些熔断配置
- private final AtomicLong parentTripCount = new AtomicLong(0);
    - 父熔断器跳闸次数
- private final OverLimitStrategy overLimitStrategy
    - 超限策略，见下文

超限策略

创建超限策略

```
// 这个方法用于创建超出限制的策略
    static OverLimitStrategy createOverLimitStrategy(boolean trackRealMemoryUsage) {
        // 获取 JVM 信息
        JvmInfo jvmInfo = JvmInfo.jvmInfo();
        // 如果 trackRealMemoryUsage 是 true 且 JVM 使用 G1 垃圾收集器，
        // 并且系统属性 "es.real_memory_circuit_breaker.g1_over_limit_strategy.enabled" 被设置为 true （默认为 true），
        // 则创建 G1OverLimitStrategy 对象
        if (trackRealMemoryUsage && jvmInfo.useG1GC().equals("true")
            // 修改 GC 行为被认为是"危险的"，因此我们提供了一个退出机制。这不是预期的使用方式。
            && Booleans.parseBoolean(System.getProperty("es.real_memory_circuit_breaker.g1_over_limit_strategy.enable
d"), true)) {
            // 获取锁超时时间，如果系统属性 "es.real_memory_circuit_breaker.g1_over_limit_strategy.lock_timeout_ms" 没有被设
置，则默认为 500 毫秒
            TimeValue lockTimeout = TimeValue.timeValueMillis(
                Integer.parseInt(System.getProperty("es.real_memory_circuit_breaker.g1_over_limit_strategy.lock_timeout_
ms", "500"))
            );
            // hardcode interval, do not want any tuning of it outside code changes.
            // 返回 G1OverLimitStrategy 对象，其中包含 JVM 信息，内存使用方法，young GC 计数提供者，当前系统时间获取方法，间隔时间（硬
编码为 5000），以及锁超时时间
            return new G1OverLimitStrategy(jvmInfo, HierarchyCircuitBreakerService::realMemoryUsage, createYoungGcCountS
upplier(),
                System::currentTimeMillis, 5000, lockTimeout);
        } else {
            // 如果不满足上述条件，返回一个只返回传入参数的函数
            return memoryUsed -> memoryUsed;
        }
    }
```

G1OverLimitStrategy G1 策略

```
static class G1OverLimitStrategy implements OverLimitStrategy {
        // G1垃圾收集器区域的大小
        private final long g1RegionSize;
        // 当前内存使用的供应器
        private final LongSupplier currentMemoryUsageSupplier;
        // GC计数供应器
        private final LongSupplier gcCountSupplier;
        // 时间供应器
        private final LongSupplier timeSupplier;
        // 锁超时时间
        private final TimeValue lockTimeout;
        // 最大堆大小
        private final long maxHeap;

        // 上次检查时间
        private long lastCheckTime = Long.MIN_VALUE;
        // 最小间隔时间
        private final long minimumInterval;

        // 黑洞变量，用于防止编译器优化
        private long blackHole;
        // 可释放的锁对象
        private final ReleasableLock lock = new ReleasableLock(new ReentrantLock());
```

overLimit 尝试分配内存

这个方法被org.elasticsearch.indices.breaker.HierarchyCircuitBreakerService#checkParentLimit 调用

```
public void checkParentLimit(long newBytesReserved, String label) throws CircuitBreakingException {
        // 被addEstimateBytesAndMaybeBreak 调用，带上需要分配的字节，计算目前内存使用情况
        final MemoryUsage memoryUsed = memoryUsed(newBytesReserved);
        long parentLimit = this.parentSettings.getLimit();
        // 拿着预估之后的内存使用情况到G1OverLimitStrategy策略中尝试分配
        if (memoryUsed.totalUsage > parentLimit && overLimitStrategy.overLimit(memoryUsed).totalUsage > parentLimit) {
            //
        }
}
```

`overLimit` 方法主要目标是尝试触发JVM的G1垃圾收集器进行内存回收，从而降低当前的内存使用量。如果成功降低了内存使用量，那么返回新的内存使用状态；否则，返回原来的内存使用状态。

```
@Override
public MemoryUsage overLimit(MemoryUsage memoryUsed) {
    boolean leader = false;
    //
    int allocationIndex = 0;
    long allocationDuration = 0;
    try (ReleasableLock locked = lock.tryAcquire(lockTimeout)) {
        if (locked != null) {
            long begin = timeSupplier.getAsLong();
            // // 如果当前时间超过上次检查时间加上最小间隔，则设 leader 为 true
            leader = begin >= lastCheckTime + minimumInterval;

            overLimitTriggered(leader);
            if (leader) {
                // 获取目前的 young gc的次数
                long initialCollectionCount = gcCountSupplier.getAsLong();
                logger.info("attempting to trigger G1GC due to high heap usage [{}]", memoryUsed.baseUsage);
                // 于储存每次分配的数组的哈希值之和
                long localBlackHole = 0;
                // 计算需要分配的次数，取堆的最大可能大小\maxHeap减去当前已经使用的堆内存大小memoryUsed.baseUsage，然后除以G1垃圾收集器的区域大小g1RegionSize。
                // 这样做的目的是尽可能地填满剩余的内存区域。+1是为了确保超过内存限制，这样更可能触发垃圾收集
                // +1是为了确保超过内存限制，这样更可能触发垃圾收集。
```

```
                // number of allocations, corresponding to (approximately) number of free regions + 1
                int allocationCount = Math.toIntExact((maxHeap - memoryUsed.baseUsage) / g1RegionSize + 1);

                // 分配大小为 G1 区域大小的一半
                // 在G1垃圾收集器中，如果一个对象大于区域大小的一半，那么它会被认为是一个“巨大对象”（Humongous object）。
                // 巨大对象在内存分配和垃圾收集中有特殊的处理方式：它们会直接占用一个或多个整个区域，并且它们的回收不会在普通的GC过程中进行，只有在全局GC（也称为STW，Stop-The-World）的时候才会被回收。因此，创建大量的巨大对象可以更有可能触发全局GC。
                // allocations of half-region size becomes single humongous alloc, thus taking up a full region.
                int allocationSize = (int) (g1RegionSize >> 1);
                long maxUsageObserved = memoryUsed.baseUsage;
                // 按照需要分配的次数进行循环
                for (; allocationIndex < allocationCount; ++allocationIndex) {
                    long current = currentMemoryUsageSupplier.getAsLong(); // 当前内存使用量
                    if (current >= maxUsageObserved) {
                        maxUsageObserved = current;
                    } else {
                        // 如果观察到内存使用量下降，可能有 GC 发生，中断循环
                        // we observed a memory drop, so some GC must have occurred
                        break;
                    }
                    // 如果 GC 计数有变化，中断循环，这里判断young gc就可以了
                    if (initialCollectionCount != gcCountSupplier.getAsLong()) {
                        break;
                    }
                    // 假假的分配一下内存， 因为没有具体的引用，所以回收得会比较快，目的是通过创建大量临时对象，意图触发gc
                    localBlackHole += new byte[allocationSize].hashCode();
                }

                blackHole += localBlackHole;
                logger.trace("black hole [{}]", blackHole);

                long now = timeSupplier.getAsLong();
                this.lastCheckTime = now;
                // 计算分配的持续时间
                allocationDuration = now - begin;
            }
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        // fallthrough
    }
    //
    // 获取当前的内存使用情况
    final long current = currentMemoryUsageSupplier.getAsLong();
    // 如果当前的内存使用量小于之前的内存使用量（说明GC已经成功运行并释放了一部分内存，这里肯定包含我们上面生成的new byte[allocationSize]）
    if (current < memoryUsed.baseUsage) {
        if (leader) {
            logger.info("GC did bring memory usage down, before [{}], after [{}], allocations [{}], duration [{}]",
                memoryUsed.baseUsage, current, allocationIndex, allocationDuration);
        }
        return new MemoryUsage(current, memoryUsed.totalUsage - memoryUsed.baseUsage + current,
            memoryUsed.transientChildUsage, memoryUsed.permanentChildUsage);
    } else {
        // 如果当前的内存使用量没有小于之前的内存使用量（说明GC没有成功运行，或者运行了但是没有释放内存）
        if (leader) {
            logger.info("GC did not bring memory usage down, before [{}], after [{}], allocations [{}], duration [{}]",
                memoryUsed.baseUsage, current, allocationIndex, allocationDuration);
        }
        // 返回原来的 MemoryUsage 对象，因为内存使用情况没有变化
        // prefer original measurement when reporting if heap usage was not brought down.
        return memoryUsed;
    }
}
```

- 其中localBlackHole，blackHole的意义

  `blackHole`和`localBlackHole`是一种防止 JVM 对创建的大量空数组（new byte[allocationSize]）进行过早优化（尤其是激进的垃圾收集或内存回收）的技术。本质上，这是一种对 JVM 的 "hack"，用于创建一种假的内存压力，以尝试触发垃圾收集。

`new byte[allocationSize]`创建了一个新的字节数组，但数组并未被使用。在大多数情况下，JVM 的优化器会识别到这是一个无用的操作，因此可能不会真正分配内存，也可能在极短的时间内就进行垃圾收集。为了防止这种情况，代码将每个数组的 hashCode（这个操作会强制 JVM 计算并缓存数组的哈希值）累加到`localBlackHole`变量中。这样做的目的是确保 JVM 认为这些数组是 "有用的"，并为它们在内存中分配空间。

然后，`localBlackHole`的值会被累加到`blackHole`中，`blackHole`是一个对象级别的字段，它的生命周期超过了这个方法，这进一步确保了 JVM 不会过早地回收这些内存。

注，这只是理想做法，具体还是要依赖jvm实现。

思考

这种方式，和我每次都调用gc， 然后再从ManagementFactory.getMemoryMXBean(); 获取剩余的堆空间有啥区别？

- 每次调用gc 肯定很有影响
- overLimit方法有锁，直接调用 `System.gc()` 没有锁的机制，虽然可以加
- overLimit的方法是为了更利用G1 特性，如果使用`System.gc()`可能产生不同结果
- overLimit 方法尝试在需要时只触发最小的GC，而 `System.gc()` 则可能导致全面的垃圾收集，即使只需要释放少量内存。
- overLimit 方法是基于其内部追踪的内存使用情况(各种熔断器的内存)来判断是否需要触发GC，而不是单纯依赖ManagementFactory.getMemoryMXBean()， 这样统计也更精准

MemoryUsage 计算内存使用情况

估算并返回内存使用情况。这种估算考虑到每个熔断器的已用内存量以及它们的开销，并根据熔断器的持久性把这些内存使用量分配给`transientUsage`和`permanentUsage`两个变量。

- 如果启用了实际内存跟踪(`trackRealMemoryUsage`标志为`true`)，那么这个方法会返回当前的内存使用量和预计的内存使用量，其中预计的内存使用量是通过把新预留的字节数添加到当前内存使用量得到的。
- 如果未启用实际内存跟踪，那么这个方法会返回预估的内存使用量，这个预估量是所有熔断器的内存使用量之和。

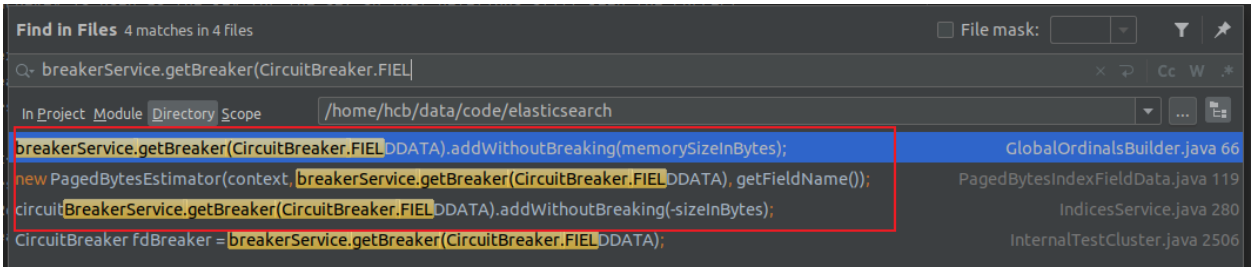
```
// 这个方法用于计算内存使用情况，考虑新预留的字节
    private MemoryUsage memoryUsed(long newBytesReserved) {
        // 定义临时内存使用和永久内存使用的变量
        long transientUsage = 0;
        long permanentUsage = 0;

        // 遍历所有的熔断器
        for (CircuitBreaker breaker : this.breakers.values()) {
            // 对于每个熔断器器，获取其已使用的内存量*overhead
            long breakerUsed = (long)(breaker.getUsed() * breaker.getOverhead());
            // 根据熔断器的持久性，将其使用的内存添加到相应的变量
            if (breaker.getDurability() == CircuitBreaker.Durability.TRANSIENT) {
                transientUsage += breakerUsed;
            } else if (breaker.getDurability() == CircuitBreaker.Durability.PERMANENT) {
                permanentUsage += breakerUsed;
            }
        }
        // 如果trackRealMemoryUsage标志被设置为true，则返回当前的实际内存使用情况和预计的内存使用情况
        if (this.trackRealMemoryUsage) {
            final long current = currentMemoryUsage();  // 这里会去MEMORY_MX_BEAN 中获取当前使用的内存量
            return new MemoryUsage(current, current + newBytesReserved, transientUsage, permanentUsage);
        } else {
            // 否则，返回预计的内存使用情况，这是所有熔断器的内存使用的总和， 这里不会再用到传递进来的newBytesReserved。有点奇怪
```

```
            long parentEstimated = transientUsage + permanentUsage;
            return new MemoryUsage(parentEstimated, parentEstimated, transientUsage, permanentUsage);
        }
    }
```

```
static class MemoryUsage {
        final long baseUsage;
        final long totalUsage;
        final long transientChildUsage;
        final long permanentChildUsage;

        MemoryUsage(final long baseUsage, final long totalUsage, final long transientChildUsage, final long permanentChi
ldUsage) {
            this.baseUsage = baseUsage;
            this.totalUsage = totalUsage;
            this.transientChildUsage = transientChildUsage;
            this.permanentChildUsage = permanentChildUsage;
        }
    }
```

fileddata 熔断

- indices.breaker.fielddata.limit
  - 默认40%
- indices.breaker.fielddata.overhead
  - 默认1.03
- indices.breaker.fielddata.type
  - 无法配置，默认memory



global ordinals

先了解什么是"global ordinals"：

- 当你对某个字段进行terms或者composite等聚合时，Elasticsearch需要找出哪些文档包含了哪些项。在内部，Elasticsearch使用了Lucene的"ordinal"值，它是每个唯一项的数字表示形式。这就需要维护一个全局查找表，该表将全局ordinal值映射回原始的项值。然而，构建这个全局的查找表是非常耗费资源的，需要大量的计算和存储空间。因此，Elasticsearch使用了一个叫做"global ordinals"的技术来优化这个过程。Global ordinals是预先计算并缓存这个全局查找表的技术。这样，在进行聚合查询时，就可以直接使用缓存的global ordinals，而不需要重新计算。

- 注意，由于 global ordinals 占用内存空间比较大，Elasticsearch 并不知道要对哪些字段进行聚合，所以默认情况下，Elasticsearch 不会加载所有字段的 global ordinals（只有聚合的时候会去加载）。可以通过修改 mapping 进行预加载。
- 对于经常进行join查询的字段，启用eager_global_ordinals也会提高查询效率。

可动态开启和关闭。

```
PUT my_index/_mapping
{
  "properties": {
    "tags": {
      "type": "keyword",
      "eager_global_ordinals": false
    }
  }
}
```

```
/**
 * Utility class to build global ordinals.
 */
public enum GlobalOrdinalsBuilder {
    /**
     * 为提供的{@link IndexReader}构建全局序数。
     * "Global ordinals"是一种在Elasticsearch中用来加速聚合操作的数据结构。
     * 它预先计算并缓存了一个全局查找表，该表将全局序数值映射回原始的项值。
     * 在进行聚合查询时，可以直接使用缓存的global ordinals，而不需要重新计算。
     *
     * 这个全局查找表的构建和存储是由"field data"内存管理器处理的，它也会被Elasticsearch的熔断器系统所监控。
     * 当内存使用超过某个阈值时，熔断器会触发，防止过度使用内存。
     *
     * @param indexReader 用于读取索引数据
     * @param indexFieldData 字段的ordinal数据
     * @param breakerService 用于监控内存使用的熔断器服务
     * @param logger 日志对象
     * @param scriptFunction 脚本函数
     * @throws IOException 如果发生I/O错误
     */
    public static IndexOrdinalsFieldData build(final IndexReader indexReader, IndexOrdinalsFieldData indexFieldData,
            CircuitBreakerService breakerService, Logger logger,
            Function<SortedSetDocValues, ScriptDocValues<?>> scriptFunction) throws IOException {
        assert indexReader.leaves().size() > 1;
        long startTimeNS = System.nanoTime();

        final LeafOrdinalsFieldData[] atomicFD = new LeafOrdinalsFieldData[indexReader.leaves().size()];
        final SortedSetDocValues[] subs = new SortedSetDocValues[indexReader.leaves().size()];
        for (int i = 0; i < indexReader.leaves().size(); ++i) {
            atomicFD[i] = indexFieldData.load(indexReader.leaves().get(i));
            subs[i] = atomicFD[i].getOrdinalsValues();
        }
        final OrdinalMap ordinalMap = OrdinalMap.build(null, subs, PackedInts.DEFAULT);
        final long memorySizeInBytes = ordinalMap.ramBytesUsed();
        // 构建完全局序数后，使用熔断器的addWithoutBreaking方法更新当前field data的内存使用量。 注意这里是不熔断的。
        breakerService.getBreaker(CircuitBreaker.FIELDDATA).addWithoutBreaking(memorySizeInBytes);
    }
}
```

FileddataCache

org.elasticsearch.indices.IndicesService#IndicesService: 当缓存移除的时候，调整熔断内存

```
this.indicesFieldDataCache = new IndicesFieldDataCache(settings, new IndexFieldDataCache.Listener() {
            @Override
            public void onRemoval(ShardId shardId, String fieldName, boolean wasEvicted, long sizeInBytes) {
                assert sizeInBytes >= 0 : "When reducing circuit breaker, it should be adjusted with a number higher or
 " +
                    "equal to 0 and not [" + sizeInBytes + "]";
                circuitBreakerService.getBreaker(CircuitBreaker.FIELDDATA).addWithoutBreaking(-sizeInBytes);
            }
        });
```

PagedBytesIndexFieldData

org.elasticsearch.index.fielddata.plain.PagedBytesIndexFieldData#loadDirect

这里会先预估一个大小，（org.elasticsearch.index.fielddata.plain.PagedBytesIndexFieldData.PagedBytesEstimator#beforeLoad）然后再fileddata加载之后再重新调整熔断器大小（org.elasticsearch.index.fielddata.plain.PagedBytesIndexFieldData.PagedBytesEstimator#afterLoad），这个和request的熔断还是挺类似的。

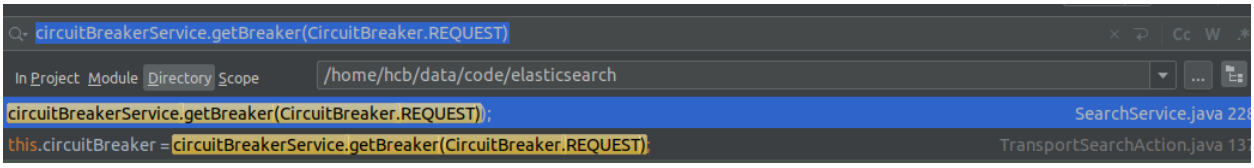
熔断怎么恢复

需要手动恢复，可以清理fileddata缓存恢复。

request 熔断

请求断路器允许 Elasticsearch 防止每个请求数据结构（比如请求中需要聚合的数据，和分片的merge）超过一定量的内存。

- indices.breaker.request.limit
  - 默认60%
- indices.breaker.request.overhead
  - 默认1.0
- indices.breaker.request.type
  - 无法配置，默认memory



搜索上的熔断 - Query 阶段

在协调节点转发Query请求的时候，就会构建一个QueryPhaseResultConsumer， 这里有熔断器， 然后会被放到SearchQueryThenFechAsyncAction中

org.elasticsearch.action.search.TransportSearchAction#searchAsyncAction:

```
// circuitBreaker
final QueryPhaseResultConsumer queryResultConsumer = searchPhaseController.newSearchPhaseResults(executor,
                circuitBreaker, task.getProgressListener(), searchRequest, shardIterators.size(), exc -> cancelTask(tas
k, exc));



SearchQueryThenFetchAsyncAction(...
                                    final QueryPhaseResultConsumer resultConsumer, final SearchRequest request,
                                   ) {
        // SearchQueryThenFetchAsyncAction 继承AbstractSearchAsyncAction
        // QueryPhaseResultConsumer会变成AbstractSearchAsyncAction的result属性
        super("query", logger, searchTransportService, nodeIdToConnection, aliasFilter, concreteIndexBoosts, indexRoutin
gs,
                executor, request, listener, shardsIts, timeProvider, clusterState, task,
                resultConsumer, request.getMaxConcurrentShardRequests(), clusters);

        // 注册queryPhaseResultConsumer的释放，以在搜索结束时释放熔断器内存
        addReleasable(resultConsumer);
```

那么再看来看具体申请内存的地方： 在 org.elasticsearch.action.search.AbstractSearchAsyncAction#performPhaseOnShard中，会注册query成功的回调函数，其中会调用QueryPhaseResultConsumer的onShardResult

```
protected void onShardResult(Result result, SearchShardIterator shardIt) {

        results.consumeResult(result, () -> onShardResultConsumed(result, shardIt));
    }


@Override
    public void consumeResult(SearchPhaseResult result, Runnable next) {
        super.consumeResult(result, () -> {});
        QuerySearchResult querySearchResult = result.queryResult();
        progressListener.notifyQueryResult(querySearchResult.getShardIndex());
        pendingMerges.consume(querySearchResult, next);
    }
```

QueryPhaseResultConsumer的consume方法，它的作用是处理搜索结果并可能触发部分聚合的合并操作。其逻辑如下：

1. 对于每一个搜索结果，首先检查是否已经存在失败或者搜索结果是否为空。如果任一条件满足，就消耗所有的结果并检查结果是否为空。如果为空，则将对应的分片添加到emptyResults中。
2. 如果没有失败并且搜索结果不为空，接下来会检查buffer中的结果数量（如果存在部分聚合合并则要加一）。如果这个数量大于或等于批处理的合并大小，那么将标记hasPartialReduce为true，并不会立即执行下一步。
    a. 批处理大小的判断：
        i. int batchReduceSize = (hasAggs || hasTopDocs) ? Math.min(request.getBatchedReduceSize(), expectedResultSize) : expectedResultSize;
        ii. request.getBatchedReduceSize()是请求中的batchedReduceSize（请求体中设置 **batched_reduce_size，默认512**）， 如果分片数据较多的时候，指定这个参数会降低协调节点内存，而expectedResultSize使用的是请求的索引的分片大小
    b. 在这种情况下，方法会创建一个新的MergeTask，这个任务将包含当前缓冲区中的所有搜索结果、当前聚合缓冲区的大小、所有空结果，以及一个指向下一个任务的引用。然后，清空当前聚合缓冲区的大小，清空缓冲区和空结果，并将新创建的任务添加到队列中, 再调用tryExecuteNext。

我们来看tryExecuteNext， tryExecuteNext和consume的联系在于，consume方法在处理每个搜索结果的时候，可能会创建一个MergeTask并添加到任务队列中，然后调用tryExecuteNext来尝试执行下一个任务

```
private void tryExecuteNext()
    final MergeTask task;
    synchronized (this) {
        // 如果队列为空、存在失败或正在运行的任务不为空，则直接返回
        if (queue.isEmpty()
                || hasFailure()
                || runningTask.get() != null) {
            return;
        }
        // 从队列中获取mergetask，并设置为正在运行的任务
        task = queue.poll();
        runningTask.compareAndSet(null, task);
    }

    // 执行一个新的任务
    executor.execute(new AbstractRunnable() {
        @Override
        protected void doRun() {
            // 获取当前的合并结果
            final MergeResult thisMergeResult = mergeResult;
            // 估算总的大小，包括当前的合并结果和任务中聚合缓冲区的大小
            long estimatedTotalSize = (thisMergeResult != null ? thisMergeResult.estimatedSize : 0) + task.aggsB
ufferSize;
            final MergeResult newMerge;
            try {
                // 获取任务中的搜索结果
                final QuerySearchResult[] toConsume = task.consumeBuffer();
                if (toConsume == null) {
                    return;
                }
                // 估算合并所需的大小
                // 这里的估算的是执行聚合 reduce 操作所需的内存大小
                // 这个估计是基于聚合结果的序列化大小的1.5倍，然后再减去原始的大小。如果你有一个序列化后的聚合结果占用了10个字节，
那么这个函数将会返回5，表示预计执行 reduce 操作还需要额外的5个字节的内存。
                long estimatedMergeSize = estimateRamBytesUsedForReduce(estimatedTotalSize);
                // 如果合并所需的大小超过阈值，则抛出熔断异常
                addEstimateAndMaybeBreak(estimatedMergeSize);
                // 更新总的大小
                estimatedTotalSize += estimatedMergeSize;
                // 合并计数增加
                ++ numReducePhases;
                // 执行部分聚合的合并
                newMerge = partialReduce(toConsume, task.emptyResults, topDocsStats, thisMergeResult, numReduceP
hases);
            } catch (Exception t) {
                // 如果合并失败，则调用合并失败的处理函数
                onMergeFailure(t);
                return;
            }
            // 合并完成后的操作，包括更新合并结果和重新估算大小，这里会调整熔断器的内存
            // 先从当前内存使用中减去预估的大小，然后再加上新的合并结果的大小。这样就将熔断器的内存使用情况更新为准确的值了
            onAfterMerge(task, newMerge, estimatedTotalSize);
            // 尝试执行下一个任务
            tryExecuteNext();
        }

        @Override
        public void onFailure(Exception exc) {
            // 如果任务失败，这里会让把内存释放
            onMergeFailure(exc);
        }
    });
}
```

3. 如果存在聚合，它会计算搜索结果的大小，并调用addWithoutBreaking在不触发熔断器的情况下添加到当前的聚合缓冲区大小。然后，将搜索结果添加到缓冲区。
    a. 调用ramBytesUsedQueryResult，通过将查询中的结果进行序列化之再计算，因为在Java中，对象的实际内存占用量并不容易直接计算，但序列化后的二进制数据的大小就可以直接测量。
    b. 为什么这里是addWithoutBreaking， 而在tryExecuteNext处理MergeTask的时候是addEstimateAndMaybeBreak？
        i. 因为前者并没有触发真正的聚合操作，而在tryExecuteNext会准备执行实际的聚合操作
        ii. 另外这里也有加2次，一次是原始聚合结果的内存占用，另一次是执行 reduce 操作本身所需的额外内存
4. 如果没有触发部分聚合合并，那么会立即执行下一步，对应query阶段搜索来讲，下一步就是onShardResultConsumed(result, shardIt)。

注意： 经过代码走读以及debug调试，发现上面的只有聚合过程中的内存暂用

再看具体释放的时机：

- 上面我们看到，如果MergeTask失败的话，会释放。
- 执行完搜索是怎么释放的呢？ 在org.elasticsearch.action.search.AbstractSearchAsyncAction#sendSearchResponse 发送完搜索请求（向客户端发送，也就说已经执行完了query阶段和fetch阶段）之后

```
回调：
listener.onResponse

这个listener是：
this.listener = ActionListener.runAfter(listener, this::releaseContext);

public void releaseContext() {
        Releasables.close(releasables);
    }

会调用到QueryPhaseResultConsumer的close方法释放query阶段的内存
@Override
public synchronized void close() {
    assert hasPendingMerges() == false : "cannot close with partial reduce in-flight";
    if (hasFailure()) {
        assert circuitBreakerBytes == 0;
        return;
    }
    assert circuitBreakerBytes >= 0;
    circuitBreaker.addWithoutBreaking(-circuitBreakerBytes);
    circuitBreakerBytes = 0;
}
```

搜索上的熔断 - Fetch阶段

在Query阶段结束，之后就会进入到Fetch阶段：

org.elasticsearch.action.search.FetchSearchPhase#innerRun，从查询阶段的shard列表中遍历，跳过查询结果为空的shard，对特定目标shard执行executeFetch方法来获取数据，其中包括分页信息。

```
private void innerRun() throws Exception {
        final int numShards = context.getNumShards();
        final boolean isScrollSearch = context.getRequest().scroll() != null;
        final List<SearchPhaseResult> phaseResults = queryResults.asList();
        // 这里会将结果进行最终的聚合，
        final SearchPhaseController.ReducedQueryPhase reducedQueryPhase = resultConsumer.reduce();
```

在Fetch节点还会去调用QueryPhaseResultConsumer的方法进行

```
@Override
public SearchPhaseController.ReducedQueryPhase reduce() throws Exception {
    // 如果还有等待合并的子任务，那么抛出错误，因为此时不应该进行最终的reduce操作
    if (pendingMerges.hasPendingMerges()) {
        throw new AssertionError("partial reduce in-flight");
    }
    // 如果有子任务失败，那么抛出错误
    else if (pendingMerges.hasFailure()) {
        throw pendingMerges.getFailure();
    }

    // 确保buffer中的数据按照一致的顺序进行排序
    pendingMerges.sortBuffer();
    // 获取TopDocs的统计信息
    final TopDocsStats topDocsStats = pendingMerges.consumeTopDocsStats();
    // 获取TopDocs列表
    final List<TopDocs> topDocsList = pendingMerges.consumeTopDocs();
    // 获取聚合结果列表
    final List<InternalAggregations> aggsList = pendingMerges.consumeAggs();
    // 获取当前已经使用的内存大小
    long breakerSize = pendingMerges.circuitBreakerBytes;
    // 如果有聚合操作，那么增加最终reduce所需内存的估计值
    if (hasAggs) {
        breakerSize = pendingMerges.addEstimateAndMaybeBreak(pendingMerges.estimateRamBytesUsedForReduce(breakerSize));
    }
    // 进行最终的reduce操作，获取reduce后的结果
    SearchPhaseController.ReducedQueryPhase reducePhase = controller.reducedQueryPhase(results.asList(), aggsList,
        topDocsList, topDocsStats, pendingMerges.numReducePhases, false, aggReduceContextBuilder, performFinalReduce);
    // 如果有聚合操作，那么更新熔断器的内存使用情况，将之前的估计值替换为实际的序列化后的大小
    if (hasAggs) {
        long finalSize = reducePhase.aggregations.getSerializedSize() - breakerSize;
        pendingMerges.addWithoutBreaking(finalSize);
        logger.trace("aggs final reduction [{}] max [{}]",
            pendingMerges.aggsCurrentBufferSize, pendingMerges.maxAggsCurrentBufferSize);
    }
    // 通知监听器最终reduce操作已完成
    progressListener.notifyFinalReduce(SearchProgressListener.buildSearchShards(results.asList()),
        reducePhase.totalHits, reducePhase.aggregations, reducePhase.numReducePhases);
    // 返回reduce后的结果
    return reducePhase;
}
```

释放的时机和上面一样，返回响应之后会触发。

BigArrays中对熔断的控制

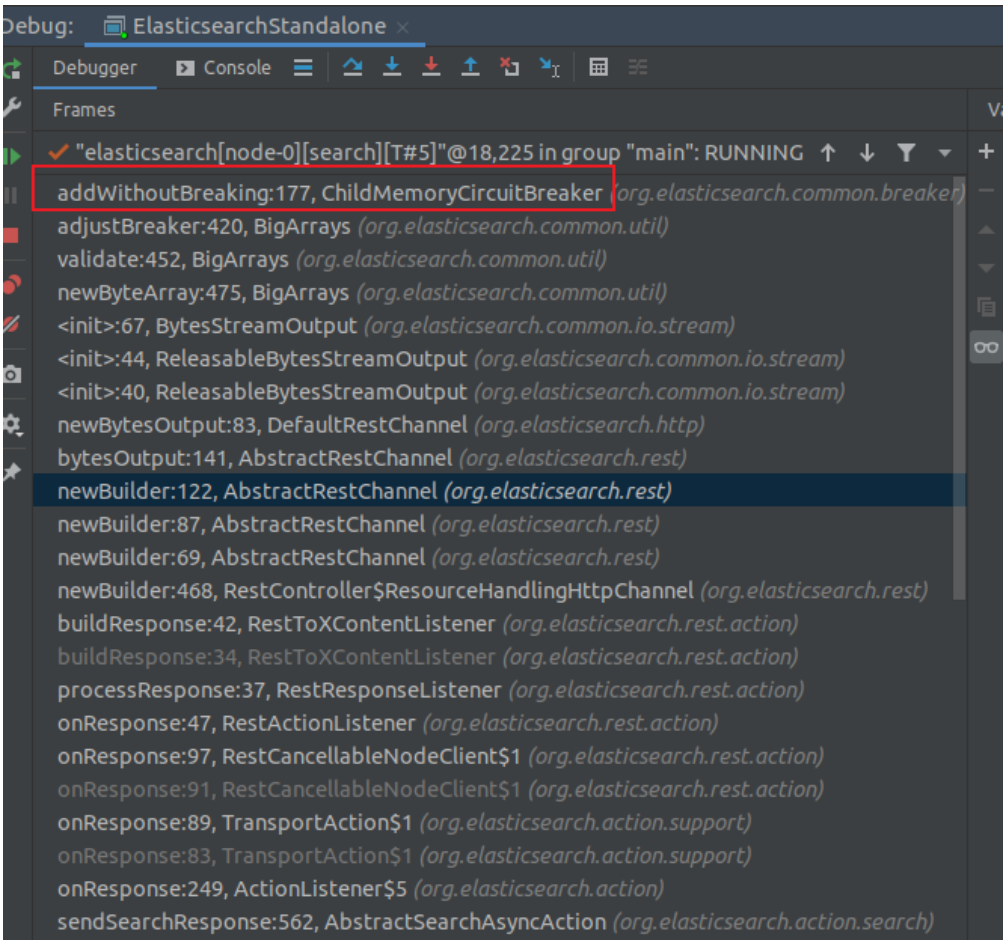
上面分析的是只有有聚合操作才会有，`如果单纯是query，或者其他API请求 。`则会去在返回响应时，在BigArrays中调整熔断器内存，

这个是在node构造函数中创建的的：

```
BigArrays bigArrays = createBigArrays(pageCacheRecycler, circuitBreakerService);


BigArrays createBigArrays(PageCacheRecycler pageCacheRecycler, CircuitBreakerService circuitBreakerService) {
        return new BigArrays(pageCacheRecycler, circuitBreakerService, CircuitBreaker.REQUEST);
    }

// 注意这里的checkBreaker为false. 因为都要返回数据了，没必要熔断了，赶紧把这个请求返回出现，就能回收了。
public BigArrays(PageCacheRecycler recycler, @Nullable final CircuitBreakerService breakerService, String breakerName) {
        // Checking the breaker is disabled if not specified
        this(recycler, breakerService, breakerName, false);
    }
```

当要返回响应时：org.elasticsearch.action.search.AbstractSearchAsyncAction#sendSearchResponse: listener.onResponse(buildSearchResponse(internalSearchResponse, failures, scrollId, searchContextId));

```
static <Response> ActionListener<Response> runAfter(ActionListener<Response> delegate, Runnable runAfter) {
        return new ActionListener<Response>() {
            @Override
            public void onResponse(Response response) {
                try {
                    delegate.onResponse(response);  // 这里会去调整熔断器的内存， 根据返回的字节大小增加
                } finally {
                    runAfter.run();  // 这里会去释放内存query和fetch阶段聚合的内存，但是bigArray申请的内存并不是这里释放
                }
            }
```

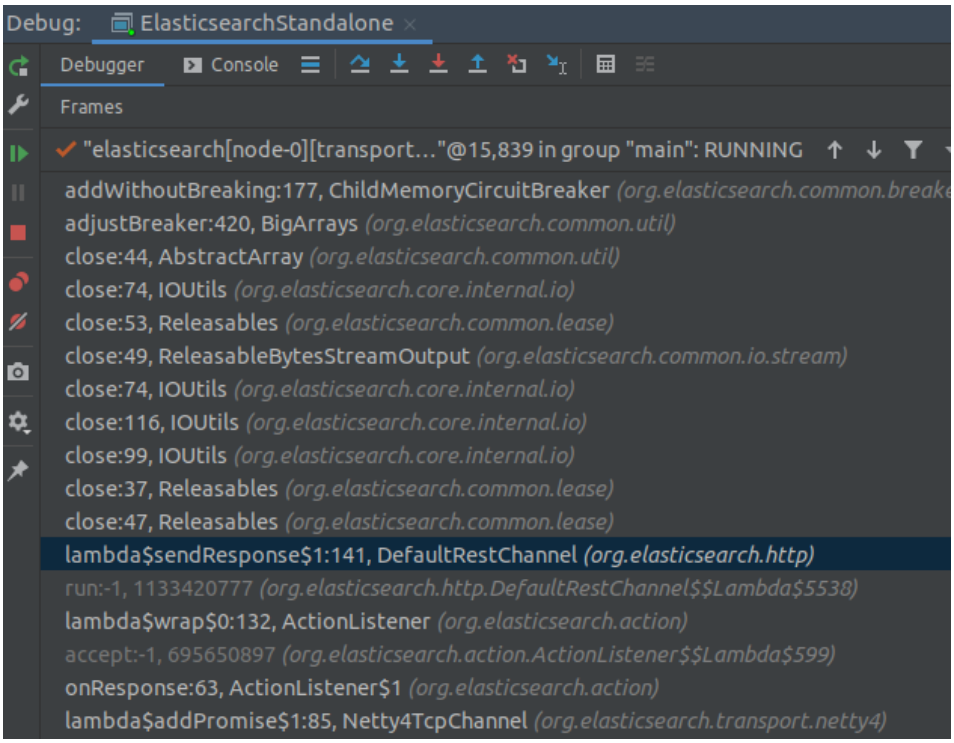


跟随调用栈会进入到adjustBreaker→addWithoutBreaking 中。

```
/**
     * 调整熔断器的状态，具体调整量由参数 delta 决定。调整方式取决于以下几个因素：
     * - 如果 delta 为负数，或者 `checkBreaker` 参数为 false，那么将在不触发熔断器的情况下调整熔断器。
     * - 如果 delta 为正数，并且 `checkBreaker` 参数为 true，则首先会检查增加 delta 是否会触发熔断器。如果会，则抛出 `CircuitBreaking
Exception` 异常。
     * - 如果与 delta 相关的数据已经被创建并占用内存（由 `isDataAlreadyCreated` 参数决定），即使熔断器触发，也会将 delta 加到熔断器中。
     *
     * @param delta 要调整熔断器的 delta 值。如果为正，则表示增加内存使用量；如果为负，则表示减少内存使用量。
     * @param isDataAlreadyCreated 标志位，表示与 delta 相关的数据是否已经被创建并占用内存。如果为 true，则将 delta 加到熔断器中，即使
这样做会触发熔断器。
     */
    void adjustBreaker(final long delta, final boolean isDataAlreadyCreated) {
        // 如果设置了熔断器服务，那么通过名称从熔断器服务中获取熔断器
        if (this.breakerService != null) {
            CircuitBreaker breaker = this.breakerService.getBreaker(breakerName);

            // 如果 `checkBreaker` 为 true，表示熔断器可以被触发
            if (this.checkBreaker) {
                // 如果 delta 为正数，那么检查增加 delta 是否会触发熔断器
                if (delta > 0) {
                    try {
                        breaker.addEstimateBytesAndMaybeBreak(delta, "<reused_arrays>");
                    } catch (CircuitBreakingException e) {
                        // 如果数据已经被创建，那么在不触发熔断器的情况下将 delta 加到熔断器中
                        if (isDataAlreadyCreated) {
                            breaker.addWithoutBreaking(delta);
                        }
```

```
                    // 重新抛出原始异常
                    throw e;
                }
            } else {
                // 如果 delta 为负数，那么在不触发熔断器的情况下将 delta 加到熔断器中
                breaker.addWithoutBreaking(delta);
            }
        } else {
            // 如果 `checkBreaker` 为 false，那么在不触发熔断器的情况下将 delta 加到熔断器中
            breaker.addWithoutBreaking(delta);
        }
    }
}
```

再看释放： 基本是一个套路调用,在发送响应完成之后会调用onResponse，然后会去调用BigArrays的close函数



熔断怎么恢复

可以自动恢复

accounting 熔断

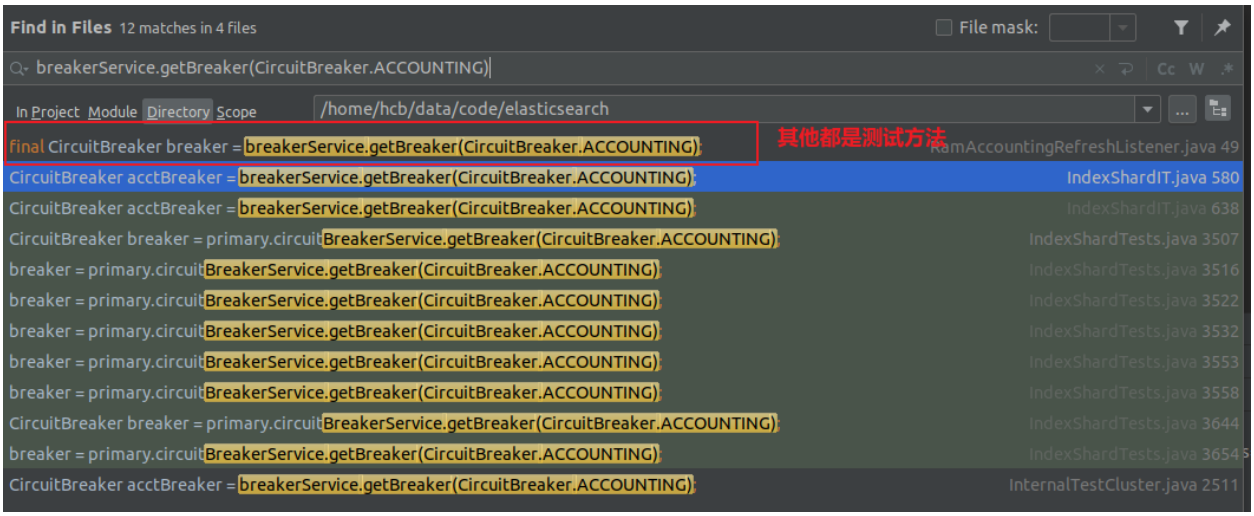
追踪在内存中持有的与请求生命周期无关的事物。这包括Lucene用于段的内存。

- indices.breaker.accounting.limit
  - 100%

- indices.breaker.accounting.overhead
  - 1.0
- indices.breaker.accounting.type
  - 无法配置，默认memory

segment 内存限制

查找源码发现这个目前只用在segment中，去跟踪segment使用的内存量。一般发生在启动的时候，或者生成新的segment的时候。




```
/**
 * 这是一个刷新监听器，用于在accounting熔断器中跟踪段使用的内存量。
 */
final class RamAccountingRefreshListener implements BiConsumer<ElasticsearchDirectoryReader, ElasticsearchDirectoryReade
r> {
    /**
     * 接受两个ElasticsearchDirectoryReader，然后对其执行操作。
     * 第一个参数reader是当前的读取器，第二个参数previousReader是先前的读取器。
     */
    @Override
    public void accept(ElasticsearchDirectoryReader reader, ElasticsearchDirectoryReader previousReader) {

        final CircuitBreaker breaker = breakerService.getBreaker(CircuitBreaker.ACCOUNTING);

        // 构造一个prevReaders集合，包含先前的segment读取器
        // 只追踪新读取器的内存使用，排除在熔断器计数外的老读取器
        final Set<IndexReader.CacheKey> prevReaders;
        if (previousReader == null) {
            prevReaders = Collections.emptySet();
        } else {
            final List<LeafReaderContext> previousReaderLeaves = previousReader.leaves();
            prevReaders = new HashSet<>(previousReaderLeaves.size());
            for (LeafReaderContext lrc : previousReaderLeaves) {
                prevReaders.add(Lucene.segmentReader(lrc.reader()).getCoreCacheHelper().getKey());
            }
        }

        // 遍历当前读取器的所有叶子读取器上下文
        // 如果当前的segment读取器在prevReaders中不存在（即它是新的segment），
```

```
        // 那么获取该segment读取器使用的RAM字节数，并将此值（不触发熔断）添加到熔断器中
        // 同时，为该segment读取器注册一个关闭监听器，
        // 当该segment关闭时，监听器会将该segment所使用的RAM字节数从熔断器中减去
        for (LeafReaderContext lrc : reader.leaves()) {
            final SegmentReader segmentReader = Lucene.segmentReader(lrc.reader());
            if (prevReaders.contains(segmentReader.getCoreCacheHelper().getKey()) == false) {
                final long ramBytesUsed = segmentReader.ramBytesUsed();
                breaker.addWithoutBreaking(ramBytesUsed);
                segmentReader.getCoreCacheHelper().addClosedListener(k -> breaker.addWithoutBreaking(-ramBytesUsed));
            }
        }
    }
}
```

熔断怎么恢复

需要手动恢复，暂时还没遇到过accounting熔断，估计只能调整大小重启了。

inflight_requests 熔断

请求中的熔断器，允许 Elasticsearch 限制在传输或 HTTP 级别上的所有当前活动的传入请求的内存使用超过节点上的一定量的内存。 内存使用是基于请求本身的内容长度。

- network.breaker.inflight_requests.limit
  - 100%
- network.breaker.inflight_requests.overhead
  - 2.0
- network.breaker.inflight_requests.type
  - 无法配置，默认memory

REST 层面的申请和释放

REST层面：

RestController中使用， 从之前源码分析中，我们知道，这个RestController是接受rest请求的。

org.elasticsearch.rest.RestController#dispatchRequest(org.elasticsearch.rest.RestRequest, org.elasticsearch.rest.RestChannel, org.elasticsearch.rest.RestHandler)

```
  private void dispatchRequest(RestRequest request, RestChannel channel, RestHandler handler) throws Exception {
        // 这个会按照请求体的长度大小作为预估字节大小做申请
        final int contentLength = request.content().length();
        if (contentLength > 0) {
            final XContentType xContentType = request.getXContentType();
            if (xContentType == null) {
                sendContentTypeErrorMessage(request.getAllHeaderValues("Content-Type"), channel);
                return;
            }
            if (handler.supportsContentStream() && xContentType != XContentType.JSON && xContentType != XContentType.SMI
LE) {
                channel.sendResponse(BytesRestResponse.createSimpleErrorResponse(channel, RestStatus.NOT_ACCEPTABLE,
                    "Content-Type [" + xContentType + "] does not support stream parsing. Use JSON or SMILE instead"));
                return;
            }
        }
```

```
        RestChannel responseChannel = channel;
        try {
            if (handler.canTripCircuitBreaker()) {  // 默认就是true的
                // 这个会按照请求体的长度大小作为预估字节大小做申请
                inFlightRequestsBreaker(circuitBreakerService).addEstimateBytesAndMaybeBreak(contentLength, "<http_reque
st>");
            } else {
                inFlightRequestsBreaker(circuitBreakerService).addWithoutBreaking(contentLength);
            }
          // 后期内存的释放通过这个ResourceHandlingHttpChannel
          responseChannel = new ResourceHandlingHttpChannel(channel, circuitBreakerService, contentLength);
```

REST 在发送请求响应的时候，会去释放内存：

```
@Override
public void sendResponse(RestResponse response) {
    close();
    delegate.sendResponse(response);
}

private void close() {
    // attempt to close once atomically
    if (closed.compareAndSet(false, true) == false) {
        throw new IllegalStateException("Channel is already closed");
    }
    // 释放内存
    inFlightRequestsBreaker(circuitBreakerService).addWithoutBreaking(-contentLength);
}
```

TCP 层面的申请和释放

另外还有TCP层面的：

在InboundPipeline中，有一个InboundAggregator。 它会去聚合多个数据包，当数据包都到达的时候，就会去判断是否内容长度是否熔断

```
private void forwardFragments(TcpChannel channel, ArrayList<Object> fragments) throws IOException {
        for (Object fragment : fragments) {
            if (fragment instanceof Header) {
                assert aggregator.isAggregating() == false;
                aggregator.headerReceived((Header) fragment);
            } else if (fragment == InboundDecoder.PING) {
                assert aggregator.isAggregating() == false;
                messageHandler.accept(channel, PING_MESSAGE);
            } else if (fragment == InboundDecoder.END_CONTENT) {

                assert aggregator.isAggregating();
                // finishAggregation 会去判断tcp 数据包的大小 checkBreaker
                try (InboundMessage aggregated = aggregator.finishAggregation()) {
                    statsTracker.markMessageReceived();
                    messageHandler.accept(channel, aggregated);
                }
            } else {
                assert aggregator.isAggregating();
                assert fragment instanceof ReleasableBytesReference;
                // 聚合多个数据包
                aggregator.aggregate((ReleasableBytesReference) fragment);
            }
        }
    }


private void checkBreaker(final Header header, final int contentLength, final BreakerControl breakerControl) {
        if (header.isRequest() == false) {
            return;
```

```
        }
        assert header.needsToReadVariableHeader() == false;

        if (canTripBreaker) {
            try {
                // 申请
                circuitBreaker.get().addEstimateBytesAndMaybeBreak(contentLength, header.getActionName());
                // 记录要释放的内存
                breakerControl.setReservedBytes(contentLength);
```

释放也是同理

```
private void setReservedBytes(int reservedBytes) {
    final boolean set = bytesToRelease.compareAndSet(0, reservedBytes);
    assert set : "Expected bytesToRelease to be 0, found " + bytesToRelease.get();
}

@Override
public void close() {
    final int toRelease = bytesToRelease.getAndSet(CLOSED);
    assert toRelease != CLOSED;
    if (toRelease > 0) {
        circuitBreaker.get().addWithoutBreaking(-toRelease);
    }
}
```

熔断之后如何自动恢复

inflight_reqeust是自动恢复。

配置最佳实践

务必开启indices.breaker.total.use_real_memory ，默认就是开启的，方便使用G1 overlimit策略。而且只有这样，才能让parent 熔断器起作用，在检查父熔断的时候会去判断当前真实使用内存，不然的话，就是各个熔断器的内存相加，准确其实很低。

其他配置感觉默认就差不多了，需要根据实际情况再做调整。

总结思考

目前熔断内存的有

- 请求体的内存
- 搜索过程，返回结果的内存占用
- fielddata 内存
- segment内存

发现还有很多内存，熔断器是无法监控到的,不过还好有parent 熔断器兜底，不过必须开启indices.breaker.total.use_real_memory才可以。

- 节点加载的索引和分片个数， 索引和分片对象都需要占用堆内存的，大小和加载的索引和分片个数成正比
- 索引的mapper 也需要占用一定的内存， 索引的mapping有关系，mapping的属性越多，就越占内存
- 对应压缩字段读取，（默认开启压缩），读取的时候也会占用堆内存， 具体和字段的个数有关系
- ..

可能存在的优化点：

- 缺少熔断的持续时间， 目前request熔断器和inflight_requests熔断器都是可以自动恢复的，但是这个缺少显示恢复的时间，感觉有这个可以更好调优
- 目前，当一个熔断器被触发时，Elasticsearch 通常会拒绝新的请求，直到内存使用降下来。如果能够提供更多的熔断策略，比如优先拒绝一些不重要的请求（给请求设置优先级，如果判断即将熔断，可以先处理高优先级的），或者暂时降低服务的性能（fileddata 自动清理），可能会更加灵活和高效。