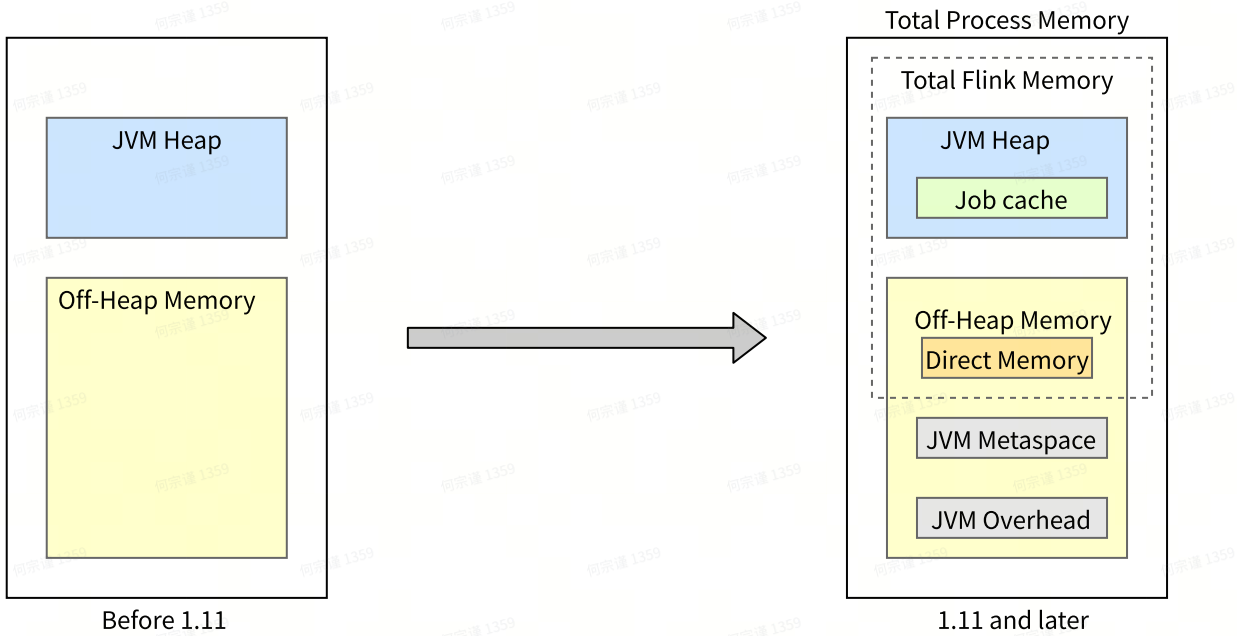
Flink 内存模型

JobManager 内存模型

Flink JobManager 内存模型在 1.11 前后的变化如下如所示：



1.9 JM 内存配置

相关的配置如下表所示：

1.9 版本
• `jobmanager.heap.mb`： JM container 申请总内存大小限制； • `containerized.heap-cutoff-ratio`：堆外内存比例设置； • `containerized.heap-cutoff-min`：堆外内存最小配置，默认 600MB；
内部版本变化： • `containerized.jobmanager.heap-cutoff-ratio`：单独为 JM 引入了一个 cut-off 的配置，线上默认为 0.1，优先使用这个配置，如果这个没有配置，会使用 `containerized.heap-cutoff-ratio`；

示例

举个例子，在 1.9 版本中，如果用户配置 `jobmanager.heap.mb` 为 4096 mb，最后得到的结果是：

1. JVM off-Heap 设置：max(600, 0.1 * 4096) = 600 mb；
2. JVM Heap 设置：4096 - 600 = 3496 mb；

也就是我们看到的 JM 的启动命令里的配置：`-Xms3496m -Xmx3496m -XX:MaxDirectMemorySize=600m`。

1.9 内存配置存在的问题

1.9 JM 内存配置比较简单，但存在着一些问题：

1. `jobmanager.heap.mb` 并不是 heap 的内存，而是一个 container 总内存大小，容易给用户误解；
2. Cut-off memory 也容易让用户困惑，主要有以下三部分用途：
   - Flink 自身或用户代码使用；
   - JVM Metaspace
   - Other JVM overhead
3. JVM 没有控制 *Direct Memory 和 Metaspace 的大小，可能有内存泄露的风险。*

1.11 JM 内存配置

1.11 JM 内存配置变得比较复杂，引入的概念如下：

1. Total Process Memory: JM JVM 使用的总内存（包括 Total Flink Memory、Metaspace 和 other Overhead），也就是申请 container 时配置的内存，在使用时，用户可以只这个值，其他的内存配置由默认参数进行推算；
2. Total Flink Memory：主要是框架和用户作业代码需要的内存，不包括 JVM Metaspace and other Overhead 部分；
3. JVM Heap：JM JVM 启动时设置 heap 大小；
   a. Job Cache：可以通过 `jobstore.cache-size` 来配置，缓存 Job ExecutionGraph 信息，也是占用的 Heap 内存，它是使用 Guava 的 Cache 实现；
4. Off-heap Memory：`-XX:MaxDirectMemorySize` 限制的内存，主要是应用程序调用 native 方法使用的，包括 JM 网络通信的部分（akka）；
   a. 在进程地址空间内分配的内存，这部分内存不在堆内。JVM 的 GC 是不会自动回收这个部分的内存的。
5. JVM Metaspace：`-XX:MaxMetaspaceSize` 限制的内存，主要用于 class load 相关；
   a. 从 JDK8 开始，类的一些元数据放在叫做 Metaspace 的 Native Memory 中；

6. JVM Overhead：保留给JVM其他的内存开销，例如：Thread Stack、code cache、GC 回收空间等等；

JM 1.11 内存配置

JM 这些内存配置参数如下所示：

Memory component	options	Default value
Total Process Memory	jobmanager.memory.process.size	None (“1472m” in default *flink-conf.yaml*)
Total Flink Memory	jobmanager.memory.flink.size	None
JVM Heap	jobmanager.memory.heap.size	None
Off-heap memory	jobmanager.memory.off-heap.size	“128m”
JVM Metaspace	jobmanager.memory.jvm-metaspace.size	“256m”
JVM Overhead	jobmanager.memory.jvm-overhead.min	“192m”
	jobmanager.memory.jvm-overhead.max	“1g”
	jobmanager.memory.jvm-overhead.fraction	0.1

内部当前 1.11 JM 的配置

当前在 Dorado 页面配置的 JM 内存，对应的是 jobmanager.memory.process.size，也就是 JM 总的内存大小。

`jobmanager.memory.enable-jvm-direct-memory-limit` 这个参数社区版默认是 false，内部版本改为了 true，也就是默认会对 `MaxDirectMemorySize` 这部分的内存做限制。

代码块

```
# JM 内存之间的计算关系如下:
Total Process Memory = Total Flink Memory + JVM Metaspace + JVM Overhead
Total Flink Memory = JVM Heap + Off-heap memory
JVM Overhead = Math.max(min, Math.min(max, fraction * Total Process Memory ))
```

举个例子，假设 JM 配置的总内存大小为 1GB，那么 JM 在启动时，生成对应 JVM 命令如下：

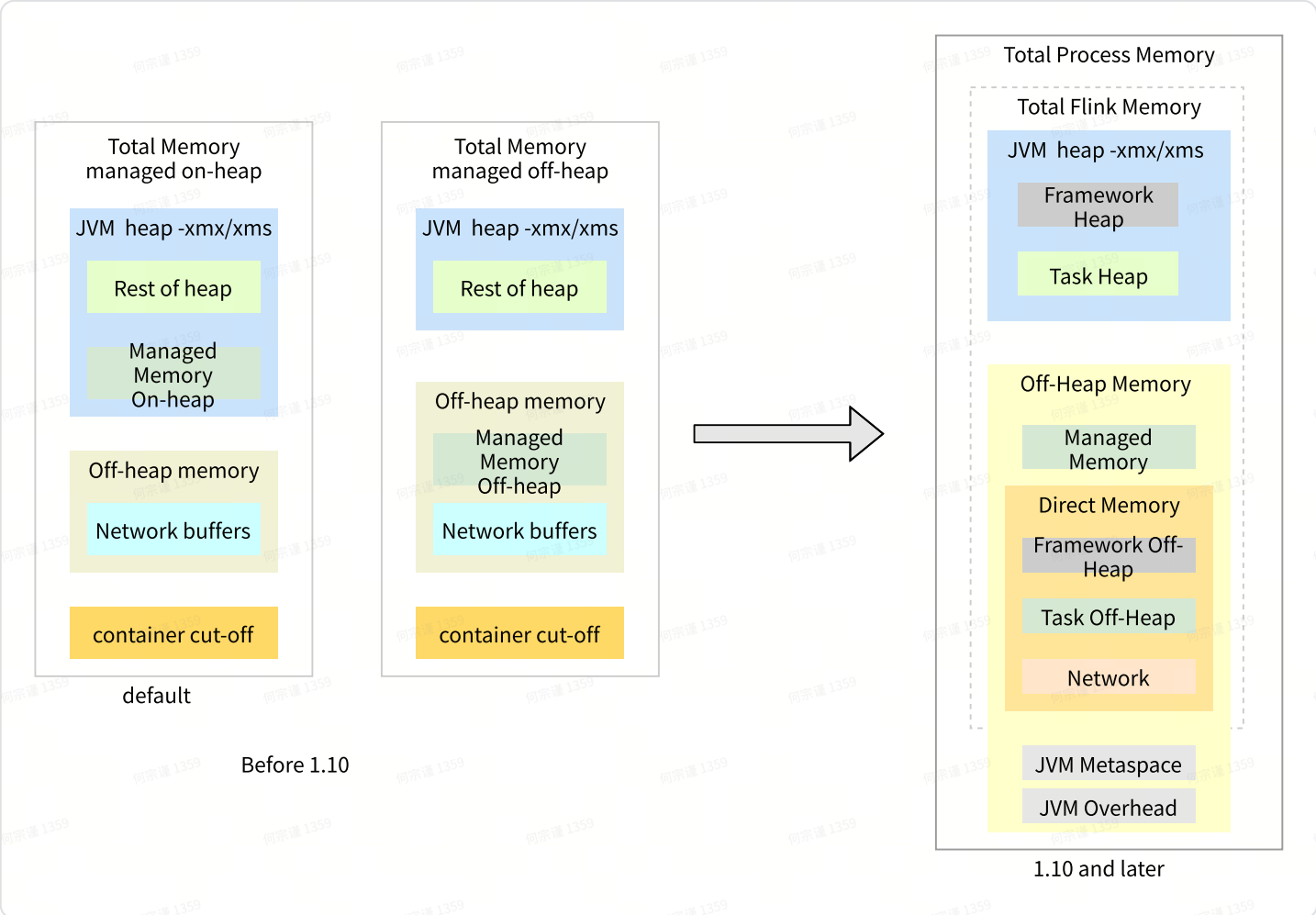
`-Xmx469762048 -Xms469762048 -XX:MaxDirectMemorySize=134217728 -XX:MaxMetaspaceSize=268435456`，算下来大概就是：

1. heap：448MB；
2. DirectMemory：128MB；
3. Metaspace：256MB；
4. Overhead：192MB。

> 参考：
>
> 1. FLIP-116: Unified Memory Configuration for Job Managers；
> 2. Set up JobManager Memory - 1.11；

TaskManager 内存模型

Flink TM 内存模型从 1.10 做了一个比较大调整，变化如下图所示：



1.9 的内存配置

1.9 可以提供的配置参数如下：

Memory component	options	Default value
Total TM Memory	**taskmanager.heap.size**	1024m
Managed Memory	taskmanager.memory.fraction	0.7
	taskmanager.memory.size	0
Managed Memory 是否分配在 off-heap	taskmanager.memory.off-heap	false
off-heap	**containerized.heap-cutoff-ratio**	0.25（python 是 0.4）
	containerized.heap-cutoff-min	600m
Network Buffers	taskmanager.network.memory.fraction	0.1（内部：0.3）
	taskmanager.network.memory.min	64m
	taskmanager.network.memory.max	1gb（内部：2gb）

对应计算逻辑如下：

代码块

```
total = all - cutoff
network = Min(max, Max(min, fraction x total)
managed = (total - network) x fraction
heap = total - managed (if off-heap) - network
```

举一个例子，一个作业的配置如下：

1. TM size：28086 MB；
2. containerized.heap-cutoff-ratio： 0.15
3. taskmanager.network.memory.fraction： 0.5
4. taskmanager.network.memory.max： 3447483648

作业运行时的配置是： `-Xms21084m -Xmx21084m -XX:MaxDirectMemorySize=7588m`

- Cutoff: 28086 * 0.15 = 4,212.9 MB
- Total: 28086 - 4,212.9 = 23,873.1 MB
- Network: 3447483648 = 3,287.8 MB
- Managed: (23,873.1 - 3,287.8)* 0.7 = 14,409.71MB (1.9 在 heap上，这块可以忽略)
- Heap: 23,873.1 - 3,287.8 = 20,585.3 MB

1.9 内存配置的缺陷

在 1.9 的内存配置中，有以下几个问题：

1. Streaming 和 batch 的内存配置不相同
    a. Streaming
        i. Memory is implicitly consumed, either on-heap by memory state backend, or off-heap by RocksDB.
        ii. Users have to manually align heap size and choice of state backend.
        iii. Users have to manually configure RocksDB to use enough memory for good performance, but not too much to exceed the budget.
        iv. No predictability in the memory consumption, neither on-heap by memory state backend, nor off-heap by RocksDB.
    b. Batch
        i. Users configure total memory size, and whether to use on-heap or off-heap memory in operators.
        ii. Flink reserves a fraction of the total memory as managed memory. It adjusts the heap size and “max direct memory” parameters automatically to account for managed memory on-heap or off-heap.
        iii. Flink allocates memory segments for the managed memory, to be used by operators. It guarantees that the reserved memory segments are never exceeded.
2. Streaming 模式 RocksDB 配置非常复杂；
3. 复杂，内存配置不容易理解；

1.11 的内存配置

内存模型改动的出发点：

- 统一 Batch 和 Streaming 的内存配置，用户的作业能够不修改配置的情况下在 Streaming 和 Batch 两种模式转换。

> 不再有 cut-off 这个概念了。

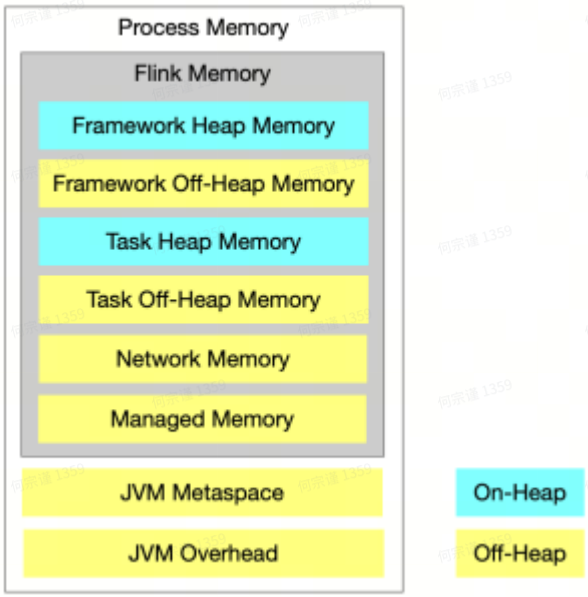
特别是对于 Managed Memory 部分。

统一显式和隐式的内存分配：

- 内存申请者当前可以通过以下两种方式申请内存：
    - 显式地以 MemorySegment 的形式从 MemoryManager 申请；
    - 从 MemoryManager 申请一部分内存，由申请者自身再进行分配（如 RocksDB Backend）。
- MemoryManager 不会预先分配任何 page；

- 对于从 MemoryManager 显式获取的堆外内存，Flink 都是使用 `Unsafe.allocateMemory()` 分配，它并不会被 JVM 的 `-XX:MaxDirectMemorySize` 限制：
  - 通常情况，堆外是 DirectMemory 的大头，现在可以消除这种不确定性；
  - 但是 JDK12 不再支持 `Unsafe` 这种机制；

> Flink 通过 ByteBuffer.allocateDirect(numBytes) 来申请堆外内存，用 sun.misc.Unsafe 来操作堆外内存。



1.11 TM 配置

JVM 参数

- JVM heap memory
  - Includes Framework Heap Memory, Task Heap Memory
  - Explicitly set both -Xmx and -Xms to this value
- JVM direct memory
  - Includes Framework Off-Heap Memory, Task Off-heap Memory and Network Memory
  - Explicitly set -XX:MaxDirectMemorySize to this value
  - For Managed Memory, we always allocate memory with Unsafe.allocateMemory(), which will not be limited by this parameter.
- JVM metaspace
  - Set -XX:MaxMetaspaceSize to configured JVM Metaspace

		JVM Parameters	内存细分	参数配置
	Total	`-Xmx and -Xms`	Framework Heap Memory	taskmanager.memory.framework.heap.size: 128MB

Total process Memory	Flink Memory			• 为Task Executor本身所配置的堆内存大小；
			Task Heap Memory	taskmanager.memory.task.heap.size • 专门用于执行 Flink 任务的堆内存空
		`MaxDirectMemorySize`	Framework Off-Heap Memory	taskmanager.memory.framework.off-heap.size: 128MB • Task Executor 保留的 off-heap memory；
			Task Off-heap Memory	taskmanager.memory.task.off-heap.size: 0 • 执行的 Task 所使用的堆外内存；
			Network Memory • Task Executor 管理的 off-heap 内存	taskmanager.memory.network.min: 64MB taskmanager.memory.network.max: 2Gb taskmanager.memory.network.fraction: 0.3 * TotalFlinkMem
		使用 `Unsafe.allocateMemory()` 分配	Managed Memory	taskmanager.memory.managed.size: 0 taskmanager.memory.managed.fraction: 0.4 * TotalFlinkMem
		`MaxMetaspaceSize`	JVM Metaspace	taskmanager.memory.jvm-metaspace.size: 256MB
			JVM Overhead	taskmanager.memory.jvm-overhead.min: 192MB taskmanager.memory.jvm-overhead.max: 1GB taskmanager.memory.jvm-overhead.fraction: 0.1 * TotalProcessMem

- **Total process Memory**：taskmanager.memory.process.size；
- **Total Flink Memory**：taskmanager.memory.flink.size；

变化较大的点：

- Managed Memery 变成了堆外内存，RocksDB 使用的是这部分内存。
- RocksDB 默认情况下遵守其限制（仅自 Flink 1.10 起），可以增加 Managed Memory 的大小以提高 RocksDB 的性能，也可以减小 Managed Memory 的大小以节省资源。

Component	Configuration options	Description
Total process Memory	taskmanager.memory.process.size	
Total Flink Memory	taskmanager.memory.flink.size	Task Executor 消耗的所有内存，也就是除了 JVM Metaspace 和 JVM Overhead 其他的加在一起就是 Total Flink Memory
Framework Heap Memory	taskmanager.memory.framework.heap.size : 128MB	JVM Heap memory dedicated to Flink framework (advanced option) 为Task Executor本身所配置的堆内存大小
Task Heap Memory	taskmanager.memory.task.heap.size	JVM Heap memory dedicated to Flink application to run operators and user code 专门用于执行Flink任务的堆内存空间
Managed memory	taskmanager.memory.managed.size: 0 taskmanager.memory.managed.fraction: 0.4 * TotalFlinkMem	Native memory managed by Flink, reserved for sorting, hash tables, caching of intermediate results and RocksDB state backend 由 Flink 直接管理的 off-heap 内存，它主要用于排序、哈希表、中间结果缓存、RocksDB 的 backend。其实它是 Task Executor 管理的 off-heap 内存
Framework Off-heap Memory	taskmanager.memory.framework.off-heap.size: 128MB	Off-heap direct (or native) memory dedicated to Flink framework (advanced option) Task Executor 保留的 off-heap memory，不会分配给任何 slot
Task Off-heap Memory	taskmanager.memory.task.off-heap.size: 0	Off-heap direct (or native) memory dedicated to Flink application to run operators Task Executor 执行的 Task 所使用的堆外内存。如果在 Flink 应用的代码中调用了 Native 的方法，需要用到 off-heap 内存，这些内存会分配到 Off-heap 堆外内存中

Network Memory	taskmanager.memory.network.min: 64MB taskmanager.memory.network.max: 1GB --> 2Gb taskmanager.memory.network.fraction: 0.1 --> 0.3 * TotalFlinkMem	Direct memory reserved for data record exchange between tasks (e.g. buffering for the transfer over the network), it is a capped fractionated component of the total Flink memory. Using in ShuffleEnvironment
JVM metaspace	taskmanager.memory.jvm-metaspace.size: 256MB	Metaspace size of the Flink JVM process 从 JDK 8 开始，JVM 把永久代拿掉了，类的一些元数据放在叫做 Metaspace 的 Native Memory
JVM Overhead	taskmanager.memory.jvm-overhead.min: 192MB taskmanager.memory.jvm-overhead.max: 1GB taskmanager.memory.jvm-overhead.fraction: 0.1 * TotalFlinkMem	Native memory reserved for other JVM overhead: e.g. thread stacks, code cache, garbage collection space etc, it is a capped fractionated component of the total process memory 保留给 JVM 其他的内存开销。例如：Thread Stack、code cache、GC 回收空间等等

应该如何配置？

目前推荐的方法有以下三种，其他的模块会根据相应参数来计算：

1. 配置 Task Heap Memory 和 Managed Memory 两项
    a. 根据 Managed Memory 值倒推出 TotalFlinkMemory；
    b. 有了 TotalFlinkMemory 之后，其他几个模块都可以推算出来；
2. 配置 Total Flink Memory
    a. 可以推算出 Network Memory 和 Managed Memory 的值，当然两个也可以由用户指定，进而其他部分也可以计算出来；
    b. JVM Overhead 的计算？
        i. 前面可以推算出 totalFlinkAndJvmMetaspaceSize（TotalFlinkMem + Metaspace），得到一个大概的值 tmp = totalFlinkAndJvmMetaspaceSize * 0.1 / (1 - 0.1)，然后再根据这个 `capToMinMax(min, max tmp)` 来计算（`jvm-overhead.fraction` 为 0.1 的情况下）；
3. **配置 Total Process Memory**

a. 首先可以根据 Metaspace 和 Overhead 得到 TotalFlinkMem 值，其他的部分就可以顺利计算出来了。

内部当前 1.11 的配置

1. Dorado 平台配置的 TM 内存参数是 TotalProcessMemory；

Conf 文件中的配置如下，只做了 NetworkBuffer 相关的配置：

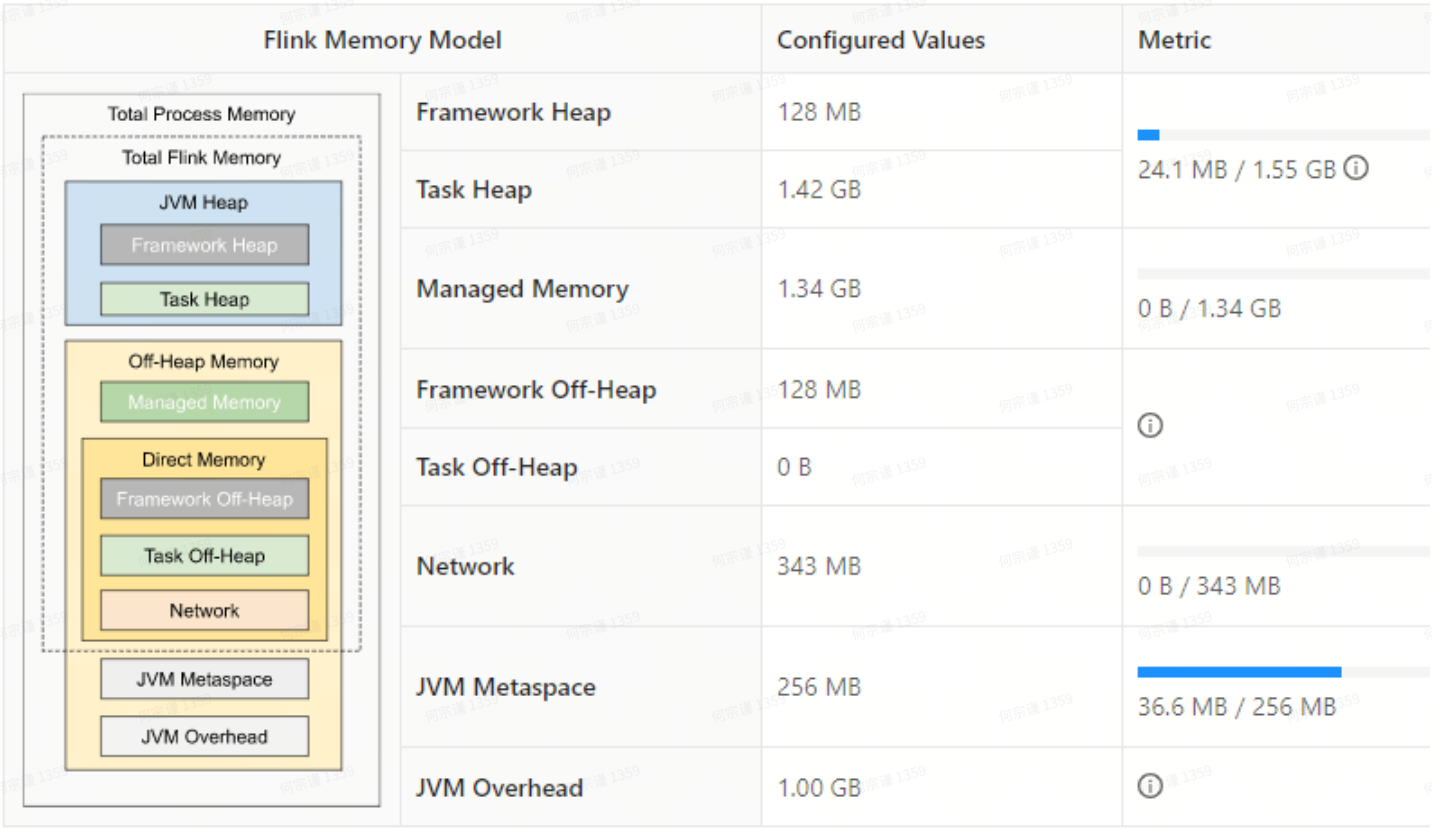
代码块

```
## Memory Configuration
taskmanager.memory.network.fraction: 0.3
taskmanager.memory.network.max: 2147483648
```

举个例子，一个任务 TM 内存配置了 4196 MB，运行时的 JVM 参数被设置如下：

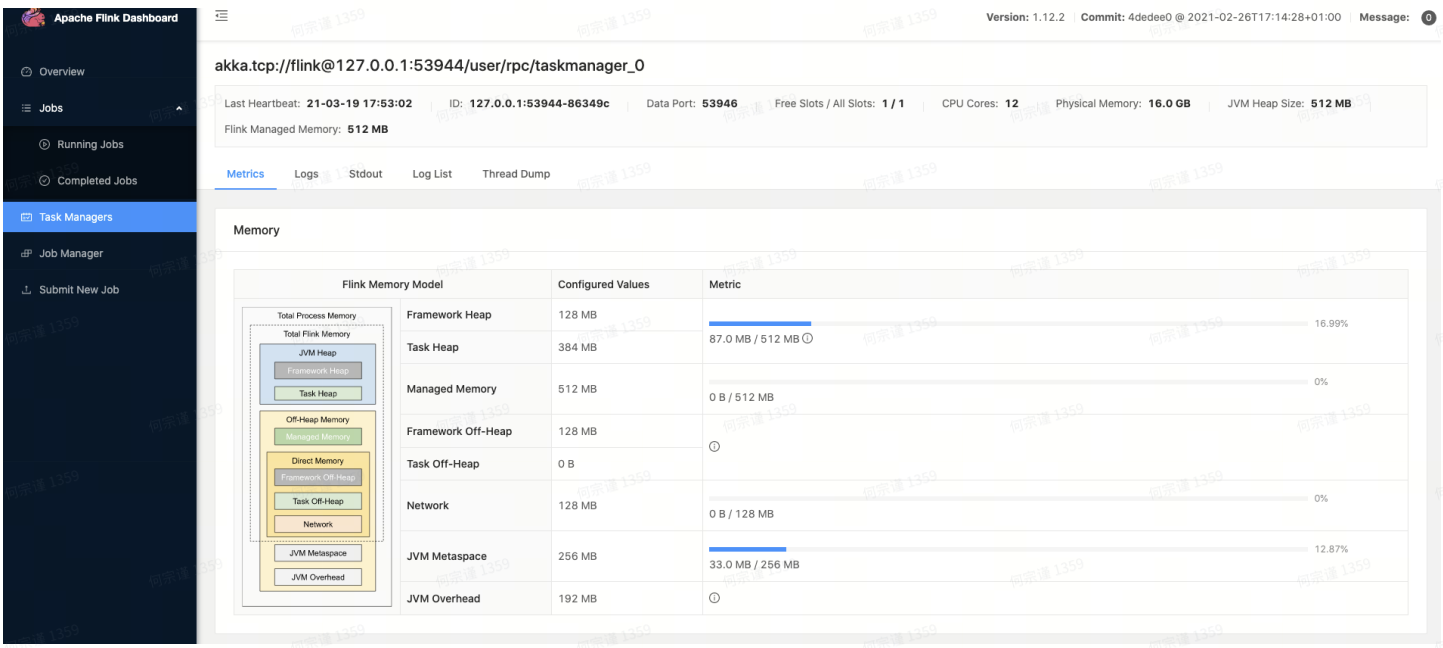
代码块

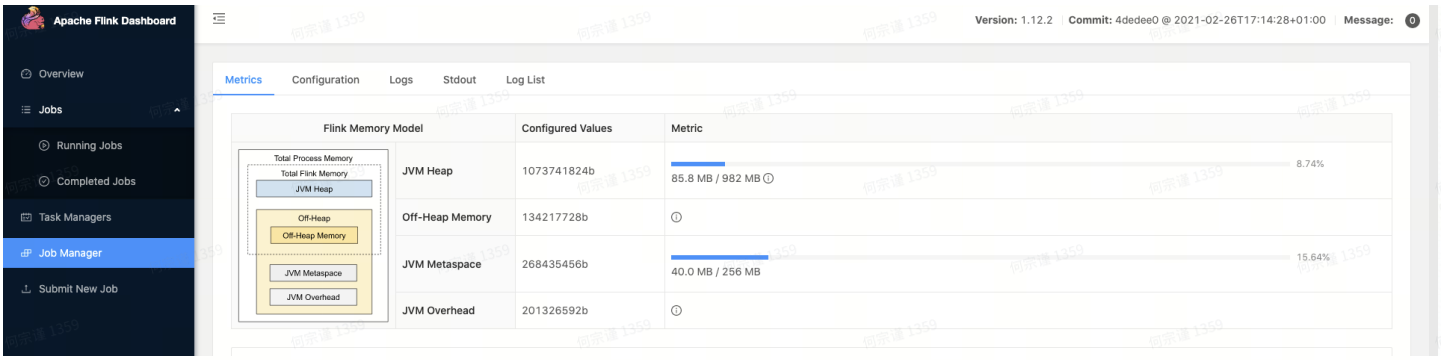
```
# JVM 配置
-Xmx973204290  # 928.12MB
-Xms973204290
-XX:MaxDirectMemorySize= 1241639855  # 1184.12MB
-XX:MaxMetaspaceSize=268435456    # 256MB

# 计算的配置
taskmanager.memory.framework.off-heap.size=134217728b
taskmanager.memory.network.max=1107422127b
taskmanager.memory.network.min=1107422127b
taskmanager.memory.task.off-heap.size=0b
taskmanager.memory.framework.heap.size=134217728b
taskmanager.memory.task.heap.size=838986562b
taskmanager.memory.managed.size=1476562799b
```



> 1.11 测试任务：http://n11-043-031.byted.org:8060/proxy/application_1614151620117_94744/#/overview

1.12 在 TM 部分，metrics 展示比较清晰：





~~Flink 1.10&1.11 中改动~~

内存这部分在1.10&1.11 有较大的改动，这里做下梳理。

1. Flink 1.10 中的变动

- 主要是 TM 内存模型的变化。见 FLIP-49: Unified Memory Configuration for TaskExecutors。

2. Flink 1.11 中的变动

- **JobManager 使用新的内存模型**：可以参考 FLIP-116，介绍了 JobManager 新的内存模型，提供了新的配置选项来控制 JobManager 的进程内存消耗，这种改变会影响 Standalone、YARN、Mesos 和 Active Kubernetes。如果你尝试在不做任何调整的情况下重用以前的Flink 配置，则新的内存模型可能会导致 JVM 的计算内存参数不同，从而导致性能发生变化甚至失败，可以参考 Migrate Job Manager Memory Configuration 文档进行迁移变更。`jobmanager.heap.size` 和 `jobmanager.heap.mb` 配置参数已经过期了，如果这些过期的选项还继续使用的话，为了维持向后兼容性，它们将被解释为以下新选项之一：
- 下面两个选项已经删除了并且不再起作用了：
- JVM 参数，JobManager JVM 进程的 direct 和 metaspace 内存现在通过下面两个参数进行配置：
- 如果没有正确配置或存在相应的内存泄漏，这些新的限制可能会产生相应的 OutOfMemoryError 异常，可以参考 OutOfMemoryError 文档进行解决。
    - jobmanager.memory.off-heap.size
    - jobmanager.memory.jvm-metaspace.size
    - containerized.heap-cutoff-ratio
    - containerized.heap-cutoff-min
    - jobmanager.memory.heap.size：JVM Heap，为了 Standalone 和 Mesos 部署
    - jobmanager.memory.process.size：进程总内存，为了容器部署（Kubernetes 和 YARN）
- **移除过期的** `mesos.resourcemanager.tasks.mem` **参数**

参考

1. A Deep-Dive into Flink's Network Stack;
2. Netty memory allocation;
3. Set up Flink's Process Memory;
4. Set up TaskManager Memory;
5. Set up JobManager Memory;
6. FLIP-49: Unified Memory Configuration for TaskExecutors;
7. FLIP-116: Unified Memory Configuration for Job Managers;
8. Off-heap Memory in Apache Flink and the curious JIT compiler;
9. Flink 原理与实现：如何处理反压问题;
10. Flink的内存管理;
11. Flink 源码阅读笔记（7）- 内存管理;

Flink 堆外&堆内存：https://blog.csdn.net/khxu666/article/details/80775635

MaxDirectMemorySize 设置：https://www.jianshu.com/p/e1503204a059