READ MODE · ANSWER FIRST

HOW JAVA JFR WORKS

An interactive, visual explanation of HOW JAVA JFR WORKS, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

The whole story in 6 lines

HOW JAVA JFR WORKS becomes easier to reason about when every stage is connected as one system.

1. Structural scene showing event class → field descriptors + metadata annotations.
2. Flow scene: multiple threads each writing to their own buffer independently.
3. Flow scene: thread buffers filling up and flushing to global storage.
4. Illustrative mechanism: SIGPROF timer, AsyncGetCallTrace, safepoint contrast.
5. Data representation: events referencing shared constant pool entries via integer IDs.
6. Structural scene: .jfr chunk layout with header, metadata, constant pools, event data.

STAGE 1 OF 8

Setup

Java Flight Recorder is a profiling and diagnostics framework built directly into the JVM. It records what your application does at runtime with extremely low overhead. Let us learn the vocabulary first.

An Event is the fundamental unit of data in JFR. Every measurement, from a garbage collection pause to a method sample, is recorded as a typed event with fields and timestamps.

Each Java thread owns a Thread-Local Buffer. Events are written into these private buffers with no locks and no contention between threads. This is the key to JFR's low overhead.

An Epoch is a generation counter that JFR uses to coordinate buffer flushes. When the epoch flips, the JVM knows which buffers contain data from the previous recording window.

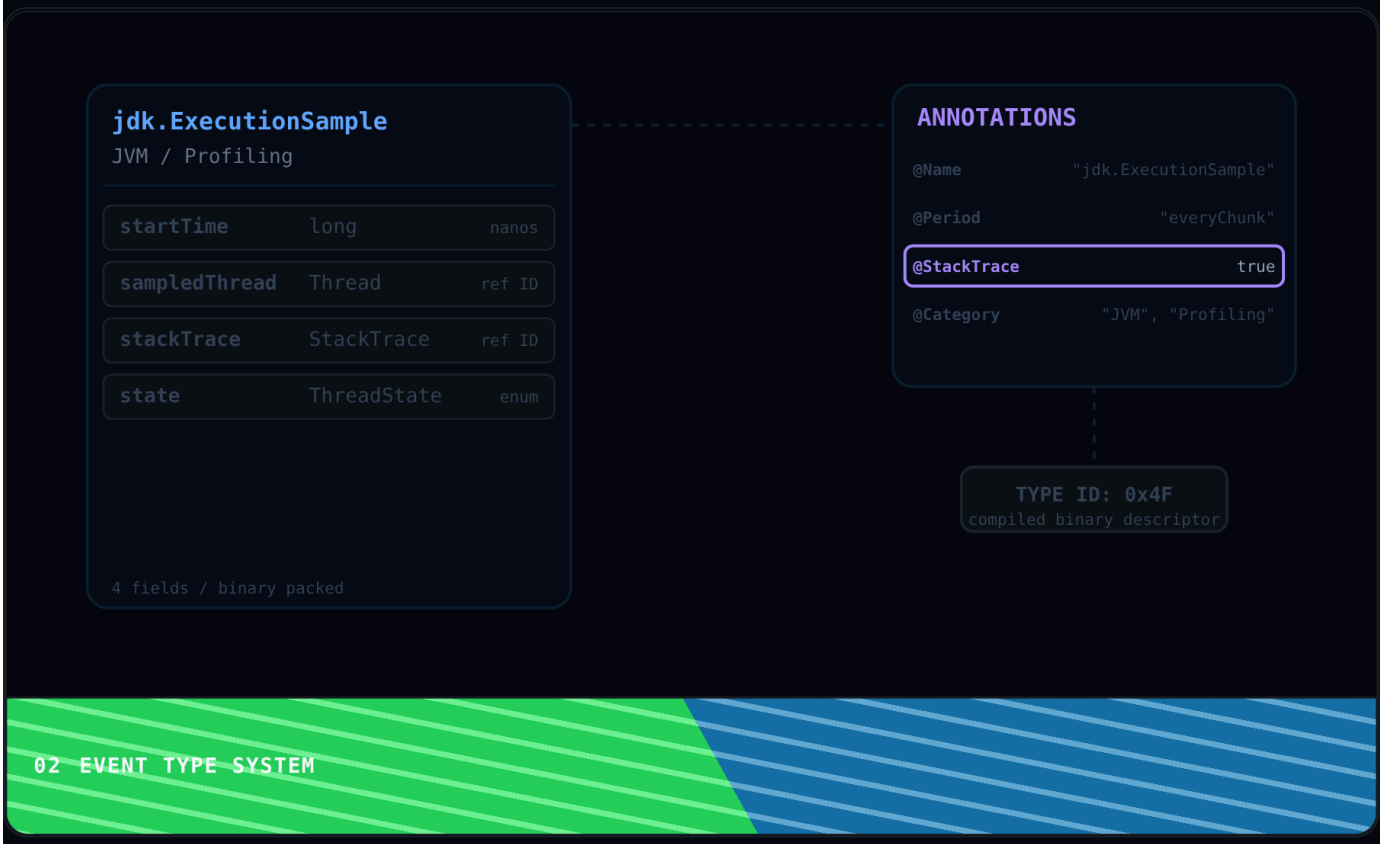
A Constant Pool stores deduplicated metadata like class names, method signatures, and stack traces. Instead of repeating "java.lang.String" in every event, JFR writes a small integer ID that points into the pool.

A Chunk is the on-disk container. Each .jfr file is a sequence of self-contained chunks, each carrying its own metadata, constant pools, and events. Chunks are what make JFR files streamable.

A Safepoint is a moment when the JVM can safely pause all application threads. Traditional profilers sample only at safepoints, which skews results. JFR can sample asynchronously between safepoints.

Now let us see the full journey. We will walk through how JFR defines events, writes them into thread-local buffers, promotes them to global storage, samples running code, deduplicates metadata, and rotates chunks to disk.

Event Type System



02 EVENT TYPE SYSTEM

Every JFR event starts life as a Java class annotated with metadata. These annotations tell the JVM how to record and filter the event. Let us look at how a single event class becomes a compact binary descriptor.

The @Name annotation assigns a stable identifier like "jdk.ExecutionSample". This ID stays the same across JDK versions so that analysis tools can reliably find events in any recording.

Fields are the payload. An ExecutionSample carries a sampled thread, a stack trace reference, and a thread state. Each field has a type ID that tells the parser how to decode it.

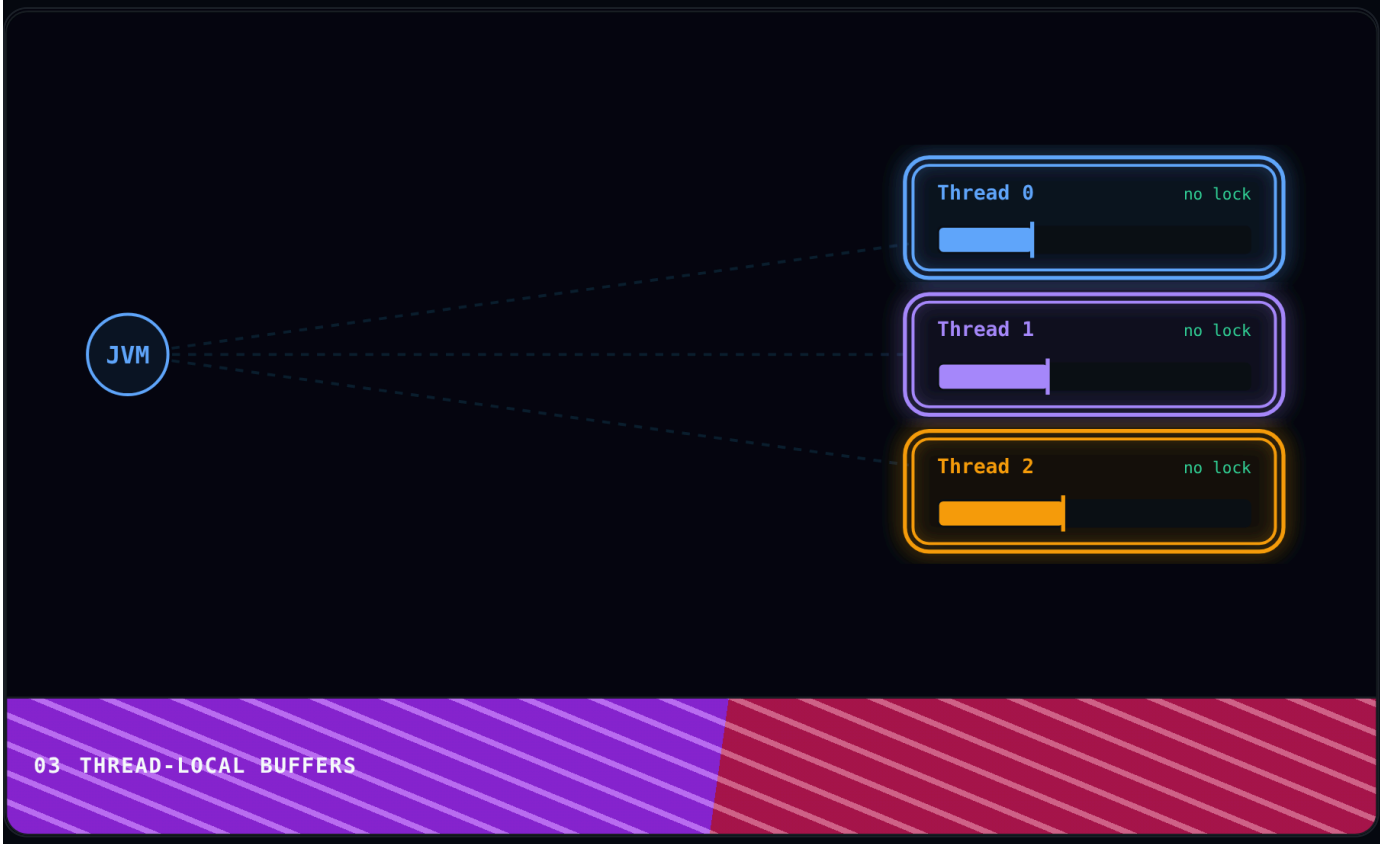
@Threshold sets a minimum duration. Events shorter than the threshold are silently dropped before they ever reach a buffer. A GC pause event with @Threshold("20 ms") ignores minor collections that finish in microseconds.

@StackTrace tells JFR to capture the call stack when this event fires. Stack capture is expensive, so not every event type requests it. ExecutionSample always does. A simple counter event might skip it entirely.

Try switching the Event Type control in the sidebar. Notice how the fields and annotations change. A GC Pause event carries cause and longest pause fields. A File I/O event carries path and bytes read. The schema is specific to the domain.

At JVM startup, the event metadata is compiled into a binary descriptor table. Each event type gets a numeric type ID. When an event fires at runtime, JFR writes just the type ID followed by the raw field values. No strings, no reflection. That is the first layer of JFR's speed. Next, let us see where those bytes actually land.

Thread-Local Buffers



03 THREAD-LOCAL BUFFERS

Now that we know what events look like, the question is: where do they go? The answer is thread-local buffers. Each application thread owns a private memory region where it writes events without touching any shared state.

When Thread 0 fires an event, it serializes the type ID and field values directly into its own buffer. There is no lock, no CAS, no contention. The write is a simple memory copy into the next free position.

At the same time, Thread 1 writes its own events into a completely separate buffer. The two threads never coordinate. This is how JFR stays under 1% overhead even in highly threaded applications.

Each buffer has a write cursor that advances with every event. The buffer is a flat byte array, typically 32 KB to 512 KB depending on configuration. Events are packed tightly with no alignment padding between them.

Try changing the Threads control in the sidebar. With more threads, more buffers appear. Each one fills independently. JFR scales linearly because no thread ever waits for another thread's buffer.

What happens when a buffer fills up? The thread does not allocate more memory. Instead, it promotes the full buffer to a global storage area and gets a fresh empty one. That promotion is the subject of the next stage.

STAGE 4 OF 8

Buffer Promotion

We just saw that each thread writes events into its own buffer. Now let us see what happens when those buffers fill up. The thread-local buffer must be promoted to a global storage area so that events can eventually reach disk.

Thread 0's buffer reaches capacity. The thread signals the JFR subsystem that it needs a fresh buffer. The current buffer is tagged with the active epoch number and moved into the global buffer pool.

The promotion itself is a pointer swap. The thread gets a new empty buffer, and the full buffer's ownership transfers to the global pool. The thread never blocks. If no empty buffer is available, events are silently dropped rather than stalling the application.

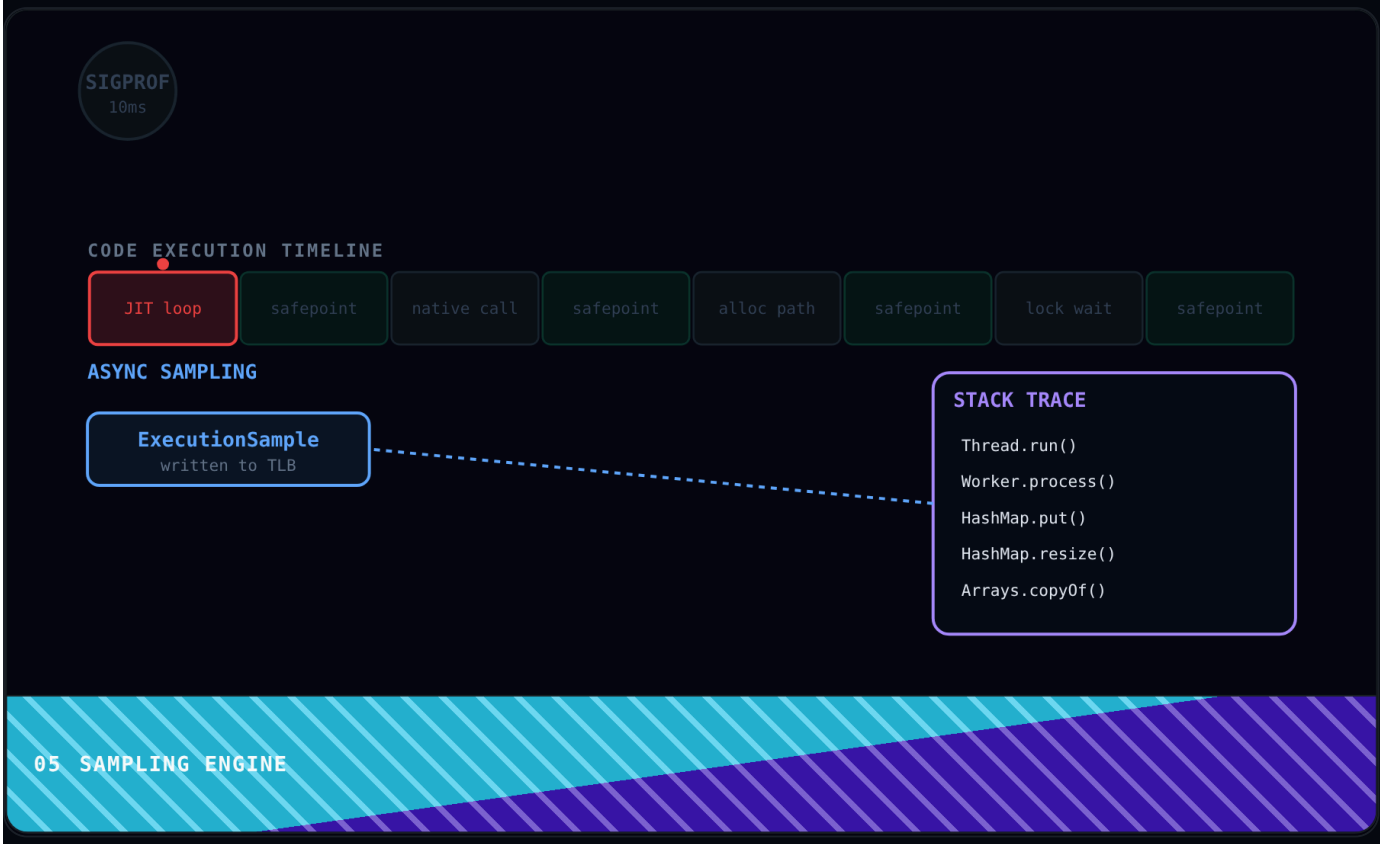
The epoch counter is the coordination mechanism. When a chunk rotation begins, the epoch increments. Buffers tagged with the old epoch are safe to flush to disk because no thread will write new events into them.

Switch the Event Rate control to High. Under heavy load, buffers promote more frequently. The global pool absorbs bursts without back-pressure on the application threads. But if the flush thread cannot keep up, old buffers are reclaimed and their events are lost. JFR trades completeness for predictable overhead.

The global buffer pool is the staging area between thread-local writes and on-disk persistence. But before events reach the disk, JFR might also inject sampled execution data. Let us look at how that sampling works.

STAGE 5 OF 8

Sampling Engine



05 SAMPLING ENGINE

We now know how events get written and promoted. But where do profiling samples come from? JFR uses a timer-driven sampling engine that captures stack traces of running threads at regular intervals.

A POSIX timer fires SIGPROF at a configurable interval, typically every 10 to 20 milliseconds. The signal handler interrupts whichever thread the OS chose to deliver it to. That thread is frozen mid-instruction.

Inside the signal handler, JFR calls AsyncGetCallTrace. This is the critical function. It walks the interrupted thread's stack frames without waiting for a safepoint. The thread does not need to be at a "safe" location in the JIT-compiled code.

The stack walk produces a list of method IDs and bytecode indexes. JFR wraps these into an ExecutionSample event and writes it to the interrupted thread's local buffer. The entire operation takes microseconds.

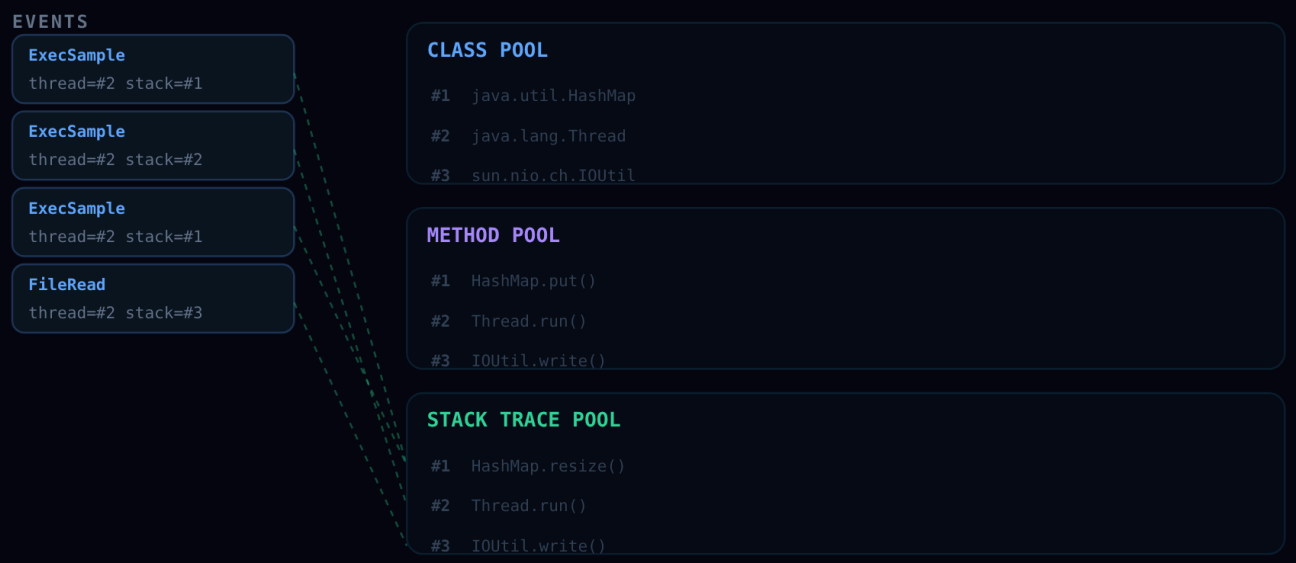
Switch the Sampling control to Safepoint. Traditional profilers like VisualVM only sample at safepoints. Notice how samples cluster at safepoint-safe code and miss the hot loops between them. This is safepoint bias. It makes counted loops and native calls invisible to the profiler.

Switch back to Async. JFR's async sampling sees the full picture because it interrupts threads wherever they happen to be. This is the same technique that async-profiler uses and the reason JFR profiles are more representative than safepoint-based alternatives.

The sampled stack traces reference method names and class names that appear thousands of times across events. Storing them inline would bloat the recording. That is where constant pools come in. Let us see how JFR deduplicates all that metadata.

STAGE 6 OF 8

Constant Pools



06 CONSTANT POOLS

Remember the event type system from Stage 1? Fields like "sampled thread" and "stack trace" are not stored as raw strings. They are stored as integer IDs that point into constant pools. This stage shows how that deduplication works.

The Class constant pool maps integer IDs to fully qualified class names. When three different events reference "java.util.HashMap", they all store the same small integer, say 42, instead of the 20-byte string.

The Method pool maps IDs to method descriptors: the owning class ID, the method name, and the parameter signature. A method entry like "HashMap.put(Object,Object)" occupies a single row in the pool and gets referenced by ID everywhere else.

The StackTrace pool is the most impactful. A single stack trace might appear in hundreds of ExecutionSample events. Instead of repeating 20 frames per sample, every event stores one integer that indexes into the shared trace table.

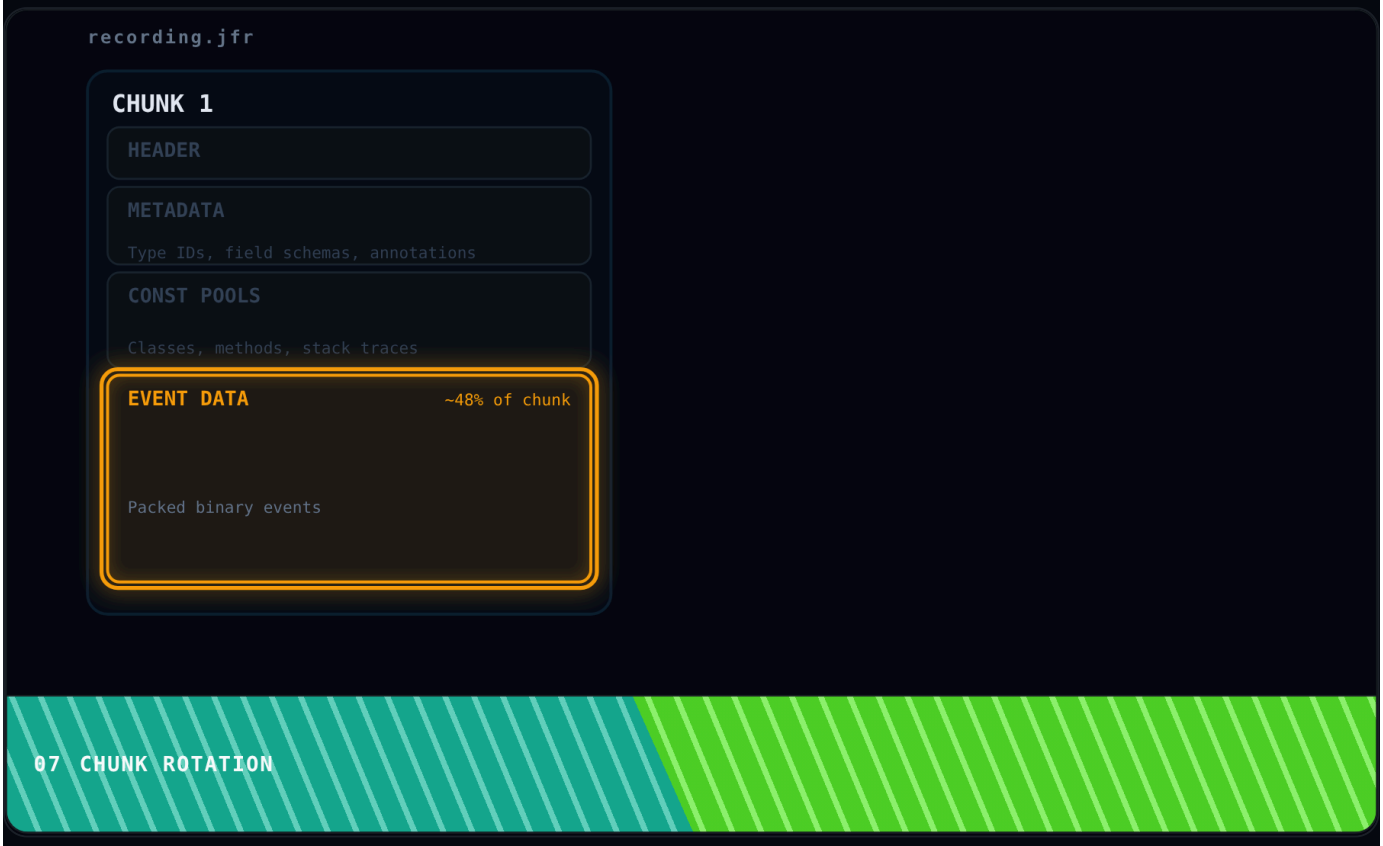
Switch the Pool View control to Expanded. You can see the actual entries and how many events reference each one. In a real recording, the top stack trace might be referenced by thousands of samples. The deduplication ratio is often 100:1 or better.

Constant pools are built incrementally as events are written. When a new class or method appears for the first time, it gets the next available ID. Subsequent references reuse that ID. The pools are flushed alongside the event data when a chunk is written to disk.

Now we have events, buffers, promotion, sampling, and deduplication. The final piece is getting it all onto disk in a format that analysis tools can stream and parse. That is the chunk format.

STAGE 7 OF 8

Chunk Rotation



07 CHUNK ROTATION

We have events in global buffers and metadata in constant pools. The last step is writing them to disk. JFR organizes its output as a sequence of self-contained chunks inside a .jfr file. Let us look inside one chunk.

Every chunk begins with a fixed-size header: a magic number ("FLR\0"), a format version, a chunk size in bytes, and a start timestamp. The header also records the position of the metadata descriptor and the constant pools within the chunk.

The metadata descriptor follows. It is a binary schema that describes every event type in this chunk: type IDs, field names, field types, and annotations. A standalone parser can read any chunk without external schema files because the schema travels with the data.

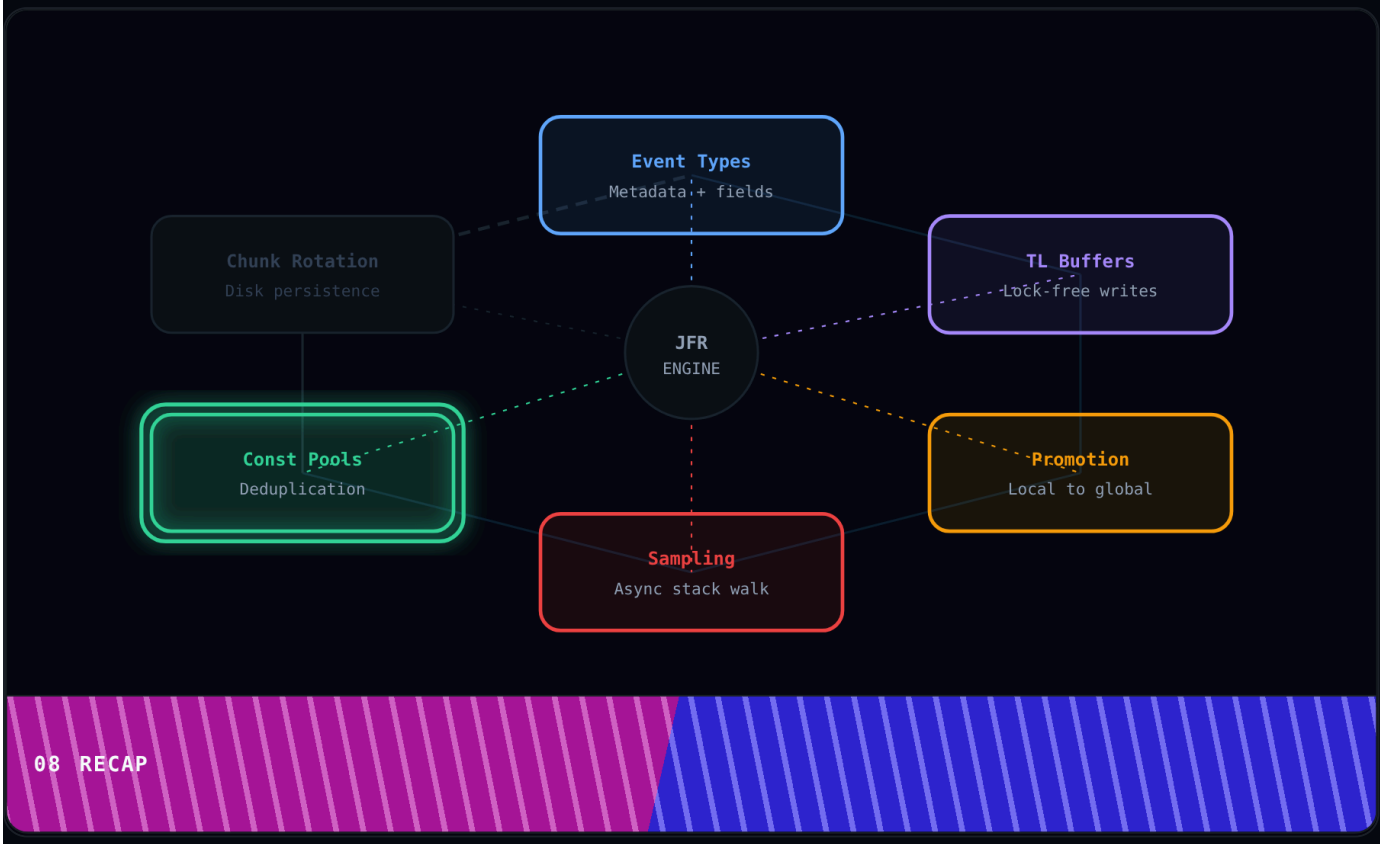
Next come the constant pools. All the class names, method descriptors, and stack traces that events in this chunk reference are serialized here. The pool data uses delta encoding to reduce the size of sequential IDs.

The event body is the largest section. Events are packed sequentially as raw binary. Each event starts with its type ID and size, followed by field values in schema order. A parser walks forward event by event using the size prefix.

Switch the Rotation control to Size-based. By default, JFR rotates chunks on a time interval (every 1 second for continuous recordings, every 12 hours for disk recordings). Size-based rotation caps each chunk at a byte threshold instead. Both strategies keep chunks self-contained and independently parseable.

When rotation triggers, the epoch flips. Thread-local buffers tagged with the old epoch are promoted and flushed. The flush thread writes the header, metadata, constant pools, and events as one atomic chunk. The previous chunk's header is finalized with the exact byte count. Now let us step back and see the whole picture together.

Recap



We started with the Event Type System. Every JFR event is a Java class compiled into a binary descriptor with a type ID, field layout, and filtering annotations like @Threshold and @StackTrace.

Those events land in Thread-Local Buffers. Each thread writes into its own private buffer with no locks and no contention. This design is why JFR can record millions of events per second under 1% CPU overhead.

When buffers fill, they get promoted to a global pool via Buffer Promotion. The epoch counter coordinates this handoff so the flush thread knows which buffers are safe to write to disk.

The Sampling Engine adds profiling data by firing SIGPROF signals and calling AsyncGetCallTrace to walk stacks without waiting for safepoints. This eliminates the bias that plagues traditional profilers.

Constant Pools deduplicate all the metadata. Class names, method signatures, and stack traces are stored once and referenced by integer IDs. This keeps recordings compact even when thousands of samples share the same hot methods.

Finally, Chunk Rotation writes everything to disk as self-contained binary chunks. Each chunk carries its own schema, constant pools, and events. Any chunk can be parsed independently, which is what makes JFR files streamable.

That is the full JFR pipeline. Events are defined as metadata-annotated classes, written lock-free into thread-local buffers, promoted to global storage, enriched with async profiling samples, deduplicated through constant pools, and flushed to disk as self-describing chunks. Every layer is designed for one goal: capture everything, disturb nothing.