# 官方的更新列表如下 :

JEP 181: Nest-Based Access Control

JEP 309: Dynamic Class-File Constants

JEP 315: Improve Aarch64 Intrinsics

JEP 318: Epsilon: A No-Op Garbage Collector

JEP 320: Remove the Java EE and CORBA Modules

JEP 321: HTTP Client (Standard)

JEP 323: Local-Variable Syntax for Lambda Parameters

JEP 324: Key Agreement with Curve25519 and Curve448

JEP 327: Unicode 10

JEP 328: Flight Recorder

JEP 329: ChaCha20 and Poly1305 Cryptographic Algorithms

JEP 330: Launch Single-File Source-Code Programs

JEP 331: Low-Overhead Heap Profiling

JEP 332: Transport Layer Security (TLS) 1.3

JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)

JEP 335: Deprecate the Nashorn JavaScript Engine

JEP 336: Deprecate the Pack200 Tools and API

注 : JEP (JDK Enhancement Proposal 特性增强提议)

1. [JShell。](#_JShell。)(java9开始支持)
2. Dynamic Class-File Constants类文件新添的一种结构

[03. 局部变量类型推断（var关键字）。](#_03._局部变量类型推断（var关键字）。)(java10 开始支持)

[04. 新加的一些更实用的API](#_04._新的API)

[05. 移除的一些其他内容](#_05._移除的一些其他内容)

[06. 标准Java异步HTTP客户端。](#_05._标准Java异步HTTP客户端。)

[07. 更简化的编译运行](#_06._更简化的编译运行程序)

[08. Unicode 10](#_08._Unicode_10)

[09. Remove the JavaEE and CORBA Moudles](#_09._Remove_the)

[10. JEP : 335 : Deprecate the Nashorn JavaScript Engine](#_10._JEP_:)

[11. JEP : 336 : Deprecate the Pack200 Tools and API](#_11._JEP_:)

[12. 新的Epsilon垃圾收集器。](#_12._新的Epsilon垃圾收集器。)

[13. 新的ZGC垃圾收集器](#_13.__ZGC,)

[14. 完全支持Linux容器（包括Docker）。](#_14._完全支持Linux容器（包括Docker）。)

[15. 支持G1上的并行完全垃圾收集。](#_15._支持G1上的并行完全垃圾收集。)

[16. 免费的低耗能堆分析仪。](#_16._JEP_331)

[17. JEP 329 : 实现ChaCha20和Poly1305两种加密算法](#_17._JEP_329)

[18. 新的默认根权限证书集。](#_18._新的默认根权限证书集。)

[19. JEP 332最新的HTTPS安全协议TLS 1.3。](#_19._JEP_332最新的HTTPS安全协议TLS)

[20. Java Flight Recorder 飞行记录仪](#_20._Java_Flight)

# JShell。

用过Python的童鞋都知道，Python 中的读取-求值-打印循环（ Read-Evaluation-Print Loop ）很方便。它的目的在于以即时结果和反馈的形式。

java9引入了jshell这个交互性工具，让Java也可以像脚本语言一样来运行，可以从控制台启动 jshell ，在 jshell 中直接输入表达式并查看其执行结果。当需要测试一个方法的运行效果，或是快速的对表达式进行求值时，jshell 都非常实用。

除了表达式之外，还可以创建 Java 类和方法。jshell 也有基本的代码完成功能。我们在教人们如何编写 Java 的过程中，不再需要解释 “public static void main（String [] args）” 这句废话。

# 02. Dynamic Class-File Constants类文件新添的一种结构

Java的类型文件格式将被拓展，支持一种新的常量池格式：CONSTANT\_Dynamic，加载CONSTANT\_Dynamic会将创建委托给bootstrap方法。

目标

其目标是降低开发新形式的可实现类文件约束带来的成本和干扰。

# 03. 局部变量类型推断（var ”关键字”）。

什么是局部变量类型推断？

var javastack = "javastack";

System.out.println(javastack);

大家看出来了，局部变量类型推断就是左边的类型直接使用 var 定义，而不用写具体的类型，编译器能根据右边的表达式自动推断类型，如上面的 String 。

var javastack = "javastack";

就等于：

String javastack = "javastack";

在声明隐式类型的lambda表达式的形参时允许使用var

使用var的好处是在使用lambda表达式时给参数加上注解

(@Deprecated var x, @Nullable var y) -> x.process(y);

# 04. 新加一些实用的API

## 1. 新的本机不可修改集合API。

自 Java 9 开始，Jdk 里面为集合（List/ Set/ Map）都添加了 of 和 copyOf 方法，它们两个都用来创建不可变的集合，来看下它们的使用和区别。

示例1：

var list = List.of("Java", "Python", "C");

var copy = List.copyOf(list);

System.out.println(list == copy); // true

示例2：

var list = new ArrayList<String>();

var copy = List.copyOf(list);

System.out.println(list == copy); // false

示例1和2代码差不多，为什么一个为true,一个为false?

来看下它们的源码：

static <E> List<E> of(E... elements) {

switch (elements.length) { // implicit null check of elements

case 0:

return ImmutableCollections.emptyList();

case 1:

return new ImmutableCollections.List12<>(elements[0]);

case 2:

return new ImmutableCollections.List12<>(elements[0], elements[1]);

default:

return new ImmutableCollections.ListN<>(elements);

}

}

static <E> List<E> copyOf(Collection<? extends E> coll) {

return ImmutableCollections.listCopy(coll);

}

static <E> List<E> listCopy(Collection<? extends E> coll) {

if (coll instanceof AbstractImmutableList && coll.getClass() != SubList.class) {

return (List<E>)coll;

} else {

return (List<E>)List.of(coll.toArray());

}

}

可以看出 copyOf 方法会先判断来源集合是不是 AbstractImmutableList 类型的，如果是，就直接返回，如果不是，则调用 of 创建一个新的集合。

示例2因为用的 new 创建的集合，不属于不可变 AbstractImmutableList 类的子类，所以 copyOf 方法又创建了一个新的实例，所以为false.

注意：使用of和copyOf创建的集合为不可变集合，不能进行添加、删除、替换、排序等操作，不然会报 java.lang.UnsupportedOperationException 异常。

上面演示了 List 的 of 和 copyOf 方法，Set 和 Map 接口都有。

除了更短和更好阅读之外，这些方法也可以避免您选择特定的集合实现。在创建后，继续添加元素到这些集合会导致 “UnsupportedOperationException” 。

## 2. Stream 加强

Stream 是 Java 8 中的新特性，Java 9 开始对 Stream 增加了以下 4 个新方法。

1) 增加单个参数构造方法，可为null

Stream.ofNullable(null).count(); // 0

2) 增加 takeWhile 和 dropWhile 方法

Stream.of(1, 2, 3, 2, 1)

.takeWhile(n -> n < 3)

.collect(Collectors.toList()); // [1, 2]

从开始计算，当 n < 3 时就截止。

Stream.of(1, 2, 3, 2, 1)

.dropWhile(n -> n < 3)

.collect(Collectors.toList()); // [3, 2, 1]

这个和上面的相反，一旦 n < 3 不成立就开始计算。

3）iterate重载

这个 iterate 方法的新重载方法，可以让你提供一个 Predicate (判断条件)来指定什么时候结束迭代。

## 3. 增加了一系列的字符串处理方法

如以下所示。

// 判断字符串是否为空白

" ".isBlank(); // true

// 去除首尾空白

" Javastack ".strip(); // "Javastack"

// 去除尾部空格

" Javastack ".stripTrailing(); // " Javastack"

// 去除首部空格

" Javastack ".stripLeading(); // "Javastack "

// 复制字符串

"Java".repeat(3);// "JavaJavaJava"

// 行数统计

"A\nB\nC".lines().count(); // 3

## 4. Optional 加强

Opthonal 也增加了几个非常酷的方法，现在可以很方便的将一个 Optional 转换成一个 Stream, 或者当一个空 Optional 时给它一个替代的。

Optional.of("javastack").orElseThrow(); // javastack

Optional.of("javastack").stream().count(); // 1

Optional.ofNullable(null)

.or(() -> Optional.of("javastack"))

.get(); // javastack

## 5. 改进的文件API。

InputStream 加强

InputStream 终于有了一个非常有用的方法：transferTo，可以用来将数据直接传输到 OutputStream，这是在处理原始数据流时非常常见的一种用法，如下示例。

var classLoader = ClassLoader.getSystemClassLoader();

var inputStream = classLoader.getResourceAsStream("javastack.txt");

var javastack = File.createTempFile("javastack2", "txt");

try (var outputStream = new FileOutputStream(javastack)) {

inputStream.transferTo(outputStream);

}

# 05. 移除的一些其他内容

移除项

移除了com.sun.awt.AWTUtilities

移除了sun.misc.Unsafe.defineClass，

使用java.lang.invoke.MethodHandles.Lookup.defineClass来替代

移除了Thread.destroy()以及 Thread.stop(Throwable)方法

移除了sun.nio.ch.disableSystemWideOverlappingFileLockCheck、sun.locale.formatasdefault属性

移除了jdk.snmp模块

移除了javafx，openjdk估计是从java10版本就移除了，oracle jdk10还尚未移除javafx，而java11版本则oracle的jdk版本也移除了javafx

移除了Java Mission Control，从JDK中移除之后，需要自己单独下载

移除了这些Root Certificates ：Baltimore Cybertrust Code Signing CA，SECOM ，AOL and Swisscom

废弃项

-XX+AggressiveOpts选项

-XX:+UnlockCommercialFeatures

-XX:+LogCommercialFeatures选项也不再需要

# 06. 标准Java异步HTTP客户端。

这是 Java 9 开始引入的一个处理 HTTP 请求的的 HTTP Client API，该 API 支持同步和异步，而在 Java 11 中已经为正式可用状态，你可以在 java.net 包中找到这个 API。

来看一下 HTTP Client 的用法：

var request = HttpRequest.newBuilder()

.uri(URI.create("https://javastack.cn"))

.GET()

.build();

var client = HttpClient.newHttpClient();

// 同步

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());

// 异步

client.sendAsync(request, HttpResponse.BodyHandlers.ofString())

.thenApply(HttpResponse::body)

.thenAccept(System.out::println);

上面的 .GET() 可以省略，默认请求方式为 Get！

更多使用示例可以看这个 API，后续有机会再做演示。

现在 Java 自带了这个 HTTP Client API，我们以后还有必要用 Apache 的 HttpClient 工具包吗？

# 07. 更简化的编译运行程序

JEP 330 : 增强java启动器支持运行单个java源代码文件的程序.

注意点 :

1. 执行源文件中的第一个类, 第一个类必须包含主方法
2. 并且不可以使用别源文件中的自定义类, 本文件中的自定义类是可以使用的.

一个命令编译运行源代码

看下面的代码。

// 编译

javac Javastack.java

// 运行

java Javastack

在我们的认知里面，要运行一个 Java 源代码必须先编译，再运行，两步执行动作。而在未来的 Java 11 版本中，通过一个 java 命令就直接搞定了，如以下所示。

java Javastack.java

# 08. Unicode 10

Unicode 10 增加了8518个字符, 总计达到了136690个字符. 并且增加了4个脚本.同时还有56个新的emoji表情符号.

# 09. Remove the JavaEE and CORBA Moudles

在java11中移除了不太使用的JavaEE模块和CORBA技术

CORBA来自于二十世纪九十年代，Oracle说，现在用CORBA开发现代Java应用程序已经没有意义了，维护CORBA的成本已经超过了保留它带来的好处。

但是删除CORBA将使得那些依赖于JDK提供部分CORBA API的CORBA实现无法运行。目前还没有第三方CORBA版本，也不确定是否会有第三方愿意接手CORBA API的维护工作。

在java11中将java9标记废弃的Java EE及CORBA模块移除掉，具体如下：

1. xml相关的，

java.xml.ws,

java.xml.bind，

java.xml.ws，

java.xml.ws.annotation，

jdk.xml.bind，

jdk.xml.ws被移除，

只剩下java.xml，java.xml.crypto,jdk.xml.dom这几个模块；

1. java.corba，

java.se.ee，

java.activation，

java.transaction被移除，

但是java11新增一个java.transaction.xa模块

# 10. JEP : 335 : Deprecate the Nashorn JavaScript Engine

废除Nashorn javascript引擎，在后续版本准备移除掉，有需要的可以考虑使用GraalVM

# 11. JEP : 336 : Deprecate the Pack200 Tools and API

Java5中带了一个压缩工具:Pack200，这个工具能对普通的jar文件进行高效压缩。其 实现原理是根据Java类特有的结构，合并常数 池，去掉无用信息等来实现对java类的高效压缩。由于是专门对Java类进行压缩的，所以对普通文件的压缩和普通压缩软件没有什么两样，但是对于Jar 文件却能轻易达到10-40%的压缩率。这在Java应用部署中很有用，尤其对于移动Java计算，能够大大减小代码下载量。

Java5中还提供了这一技术的API接口，你可以将其嵌入到你的程序中使用。使用的方法很简单，下面的短短几行代码即可以实现jar的压缩和解压：

压缩

Packer packer=Pack200.newPacker();

OutputStream output=new BufferedOutputStream(new FileOutputStream(outfile));

packer.pack(new JarFile(jarFile), output);

output.close();

解压

Unpacker unpacker=Pack200.newUnpacker();

output=new JarOutputStream(new FileOutputStream(jarFile));

unpacker.unpack(pack200File, output);

output.close();

Pack200的压缩和解压缩速度是比较快的，而且压缩率也是很惊人的，在我是使用 的包4.46MB压缩后成了1.44MB（0.322%），而且随着包的越大压缩率会根据明显，据说如果jar包都是class类可以压缩到1/9的大 小。其实JavaWebStart还有很多功能，例如可以按不同的jar包进行lazy下载和 单独更新，设置可以根据jar中的类变动进行class粒度的下载。

但是在java11中废除了pack200以及unpack200工具以及java.util.jar中的Pack200 API。因为Pack200主要是用来压缩jar包的工具，由于网络下载速度的提升以及java9引入模块化系统之后不再依赖Pack200，因此这个版本将其移除掉。

# 12. 新的Epsilon垃圾收集器。

A NoOp Garbage Collector

JDK上对这个特性的描述是: 开发一个处理内存分配但不实现任何实际内存回收机制的GC, 一旦可用堆内存用完, JVM就会退出.

如果有System.gc()调用, 实际上什么也不会发生(这种场景下和-XX:+DisableExplicitGC效果一样), 因为没有内存回收, 这个实现可能会警告用户尝试强制GC是徒劳.

用法 : -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC

**class** Garbage {

**int** n = (**int**)(Math.*random*() \* 100);

@Override

**public** **void** finalize() {

System.***out***.println(**this** + " : " + n + " is dying");

}

}

**public** **class** EpsilonTest {

**public** **static** **void** main(String[] args) {

**boolean** flag = **true**;

List<Garbage> list = **new** ArrayList<>();

**long** count = 0;

**while** (flag) {

list.add(**new** Garbage());

**if** (list.size() == 1000000 && count == 0) {

list.clear();

count++;

}

}

System.***out***.println("程序结束");

}

}

如果使用选项-XX:+UseEpsilonGC, 程序很快就因为堆空间不足而退出

使用这个选项的原因 :

提供完全被动的GC实现, 具有有限的分配限制和尽可能低的延迟开销,但代价是内存占用和内存吞吐量.

众所周知, java实现可广泛选择高度可配置的GC实现. 各种可用的收集器最终满足不同的需求, 即使它们的可配置性使它们的功能相交. 有时更容易维护单独的实现, 而不是在现有GC实现上堆积另一个配置选项.

主要用途如下 :

性能测试(它可以帮助过滤掉GC引起的性能假象)

内存压力测试(例如,知道测试用例 应该分配不超过1GB的内存, 我们可以使用-Xmx1g –XX:+UseEpsilonGC, 如果程序有问题, 则程序会崩溃)

非常短的JOB任务(对象这种任务, 接受GC清理堆那都是浪费空间)

VM接口测试

Last-drop 延迟&吞吐改进

# 13. ZGC, 这应该是JDK11最为瞩目的特性, 没有之一. 但是后面带了Experimental, 说明这还不建议用到生产环境.

ZGC, A Scalable Low-Latency Garbage Collector(Experimental)

ZGC, 这应该是JDK11最为瞩目的特性, 没有之一. 但是后面带了Experimental, 说明这还不建议用到生产环境.

GC暂停时间不会超过10ms

既能处理几百兆的小堆, 也能处理几个T的大堆(OMG)

和G1相比, 应用吞吐能力不会下降超过15%

为未来的GC功能和利用colord指针以及Load barriers优化奠定基础

初始只支持64位系统

ZGC的设计目标是：支持TB级内存容量，暂停时间低（<10ms），对整个程序吞吐量的影响小于15%。 将来还可以扩展实现机制，以支持不少令人兴奋的功能，例如多层堆（即热对象置于DRAM和冷对象置于NVMe闪存），或压缩堆。

GC是java主要优势之一. 然而, 当GC停顿太长, 就会开始影响应用的响应时间.消除或者减少GC停顿时长, java将对更广泛的应用场景是一个更有吸引力的平台. 此外, 现代系统中可用内存不断增长,用户和程序员希望JVM能够以高效的方式充分利用这些内存, 并且无需长时间的GC暂停时间.

STW – stop the world

ZGC是一个并发, 基于region, 压缩型的垃圾收集器, 只有root扫描阶段会STW, 因此GC停顿时间不会随着堆的增长和存活对象的增长而变长.

ZGC : avg 1.091ms max:1.681

G1 : avg 156.806 max:543.846

用法 : -XX:+UnlockExperimentalVMOptions –XX:+UseZGC, 因为ZGC还处于实验阶段, 所以需要通过JVM参数来解锁这个特性

# 14. 完全支持Linux容器（包括Docker）。

许多运行在Java虚拟机中的应用程序（包括Apache Spark和Kafka等数据服务以及传统的企业应用程序）都可以在Docker容器中运行。但是在Docker容器中运行Java应用程序一直存在一个问题，那就是在容器中运行JVM程序在设置内存大小和CPU使用率后，会导致应用程序的性能下降。这是因为Java应用程序没有意识到它正在容器中运行。随着Java 10的发布，这个问题总算得以解决，JVM现在可以识别由容器控制组（cgroups）设置的约束。可以在容器中使用内存和CPU约束来直接管理Java应用程序，其中包括：

遵守容器中设置的内存限制

在容器中设置可用的CPU

在容器中设置CPU约束

Java 10的这个改进在Docker for Mac、Docker for Windows以及Docker Enterprise Edition等环境均有效。

容器的内存限制

在Java 9之前，JVM无法识别容器使用标志设置的内存限制和CPU限制。而在Java 10中，内存限制会自动被识别并强制执行。

Java将服务器类机定义为具有2个CPU和2GB内存，以及默认堆大小为物理内存的1/4。例如，Docker企业版安装设置为2GB内存和4个CPU的环境，我们可以比较在这个Docker容器上运行Java 8和Java 10的区别。

首先，对于Java 8：

docker container run -it -m512 --entrypoint bash openjdk:latest

$ docker-java-home/bin/java -XX:+PrintFlagsFinal -version | grep MaxHeapSize

uintx MaxHeapSize := 524288000 {product}

openjdk version "1.8.0\_162"

1

2

3

4

最大堆大小为512M或Docker EE安装设置的2GB的1/4，而不是容器上设置的512M限制。

相比之下，在Java 10上运行相同的命令表明，容器中设置的内存限制与预期的128M非常接近：

docker container run -it -m512M --entrypoint bash openjdk:10-jdk

$ docker-java-home/bin/java -XX:+PrintFlagsFinal -version | grep MaxHeapSize

size\_t MaxHeapSize = 134217728 {product} {ergonomic}

openjdk version "10" 2018-03-20

1

2

3

4

设置可用的CPU

默认情况下，每个容器对主机CPU周期的访问是无限的。可以设置各种约束来限制给定容器对主机CPU周期的访问。Java 10可以识别这些限制：

docker container run -it --cpus 2 openjdk:10-jdk

jshell> Runtime.getRuntime().availableProcessors()

$1 ==> 2

1

2

3

分配给Docker EE的所有CPU会获得相同比例的CPU周期。这个比例可以通过修改容器的CPU share权重来调整，而CPU share权重与其它所有运行在容器中的权重相关。此比例仅适用于正在运行的CPU密集型的进程。当某个容器中的任务空闲时，其他容器可以使用余下的CPU时间。实际的CPU时间的数量取决于系统上运行的容器的数量。这些可以在Java 10中设置：

docker container run -it --cpu-shares 2048 openjdk:10-jdk

jshell> Runtime.getRuntime().availableProcessors()

$1 ==> 2

1

2

3

cpuset约束设置了哪些CPU允许在Java 10中执行。

docker run -it --cpuset-cpus="1,2,3" openjdk:10-jdk

jshell> Runtime.getRuntime().availableProcessors()

$1 ==> 3

1

2

3

分配内存和CPU

使用Java 10，可以使用容器设置来估算部署应用程序所需的内存和CPU的分配。我们假设已经确定了容器中运行的每个进程的内存堆和CPU需求，并设置了JAVA\_OPTS配置。例如，如果有一个跨10个节点分布的应用程序，其中五个节点每个需要512Mb的内存和1024个CPU-shares，另外五个节点每个需要256Mb和512个CPU-shares。

请注意，1个CPU share比例由1024表示。

对于内存，应用程序至少需要分配5Gb。

512Mb × 5 = 2.56Gb

256Mb × 5 = 1.28Gb

该应用程序需要8个CPU才能高效运行。

1024 x 5 = 5个CPU

512 x 5 = 3个CPU

最佳实践是建议分析应用程序以确定运行在JVM中的每个进程实际需要多少内存和分配多少CPU。但是，Java 10消除了这种猜测，可以通过调整容器大小以防止Java应用程序出现内存不足的错误以及分配足够的CPU来处理工作负载。

# 15. 支持G1上的并行完全垃圾收集。

对于 G1 GC，相比于 JDK 8，升级到 JDK 11 即可免费享受到：并行的 Full GC，快速的 CardTable 扫描，自适应的堆占用比例调整（IHOP），在并发标记阶段的类型卸载等等。这些都是针对 G1 的不断增强，其中串行 Full GC 等甚至是曾经被广泛诟病的短板，你会发现 GC 配置和调优在 JDK11 中越来越方便。

# 16. JEP 331 : Low-Overhead Heap Profiling免费的低耗能飞行记录仪和堆分析仪。

通过JVMTI的SampledObjectAlloc回调提供了一个开销低的heap分析方式

提供一个低开销的, 为了排错java应用问题, 以及JVM问题的数据收集框架, 希望达到的目标如下 :

提供用于生产和消费数据作为事件的API

提供缓存机制和二进制数据格式

允许事件配置和事件过滤

提供OS,JVM和JDK库的事件

# 17. JEP 329 : 实现RFC7539中指定的ChaCha20和Poly1305两种加密算法, 代替RC4

实现 RFC 7539的ChaCha20 and ChaCha20-Poly1305加密算法

RFC7748定义的秘钥协商方案更高效, 更安全. JDK增加两个新的接口

XECPublicKey 和 XECPrivateKey

KeyPairGenerator kpg = KeyPairGenerator.getInstance(“XDH”);

NamedParameterSpec paramSpec = new NamedParameterSpec(“X25519”);

kpg.initialize(paramSpec);

KeyPair kp = kgp.generateKeyPair();

KeyFactory kf = KeyFactory.getInstance(“XDH”);

BigInteger u = new BigInteger(“xxx”);

XECPublicKeySpec pubSpec = new XECPublicKeySpec(paramSpec, u);

PublicKey pubKey = kf.generatePublic(pubSpec);

KeyAgreement ka = KeyAgreement.getInstance(“XDH”);

ka.init(kp.getPrivate());

ka.doPhase(pubKey, true);

byte[] secret = ka.generateSecret();

# 18. 新的默认根权限证书集。

# 19. JEP 332最新的HTTPS安全协议TLS 1.3。

实现TLS协议1.3版本, TLS允许客户端和服务器端通过互联网以一种防止窃听, 篡改以及消息伪造的方式进行通信.

# 20. Java Flight Recorder

Flight Recorder源自飞机的黑盒子

Flight Recorder以前是商业版的特性，在java11当中开源出来，它可以导出事件到文件中，之后可以用Java Mission Control来分析。可以在应用启动时配置java -XX:StartFlightRecording，或者在应用启动之后，使用jcmd来录制，比如

$ jcmd <pid> JFR.start

$ jcmd <pid> JFR.dump filename=recording.jfr

$ jcmd <pid> JFR.stop

是 Oracle 刚刚开源的强大特性。我们知道在生产系统进行不同角度的 Profiling，有各种工具、框架，但是能力范围、可靠性、开销等，大都差强人意，要么能力不全面，要么开销太大，甚至不可靠可能导致 Java 应用进程宕机。

而 JFR 是一套集成进入 JDK、JVM 内部的事件机制框架，通过良好架构和设计的框架，硬件层面的极致优化，生产环境的广泛验证，它可以做到极致的可靠和低开销。在 SPECjbb2015 等基准测试中，JFR 的性能开销最大不超过 1%，所以，工程师可以基本没有心理负担地在大规模分布式的生产系统使用，这意味着，我们既可以随时主动开启 JFR 进行特定诊断，也可以让系统长期运行 JFR，用以在复杂环境中进行“After-the-fact”分析。还需要苦恼重现随机问题吗？JFR 让问题简化了很多。

在保证低开销的基础上，JFR 提供的能力也令人眼前一亮，例如：我们无需 BCI 就可以进行 Object Allocation Profiling，终于不用担心 BTrace 之类把进程搞挂了。对锁竞争、阻塞、延迟，JVM GC、SafePoint 等领域，进行非常细粒度分析。甚至深入 JIT Compiler 内部，全面把握热点方法、内联、逆优化等等。JFR 提供了标准的 Java、C++ 等扩展 API，可以与各种层面的应用进行定制、集成，为复杂的企业应用栈或者复杂的分布式应用，提供 All-in-One 解决方案。而这一切都是内建在 JDK 和 JVM 内部的，并不需要额外的依赖，开箱即用。