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)

01. JShell。(java9 开始支持)

02. 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）。(java10 开始支持)

04. 新加的一些更实用的 API

05. 移除的一些其他内容

06. 标准 Java 异步 HTTP 客户端。

07. 更简化的编译运行

08. Unicode 10

09. Remove the JavaEE and CORBA Moudles

10. JEP : 335 : Deprecate the Nashorn JavaScript Engine

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

12. 新的 Epsilon 垃圾收集器。

13. 新的 ZGC 垃圾收集器

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。

16. 免费的低耗能堆分析仪。

17. JEP 329： 实现 ChaCha20 和 Poly1305 两种加密算法

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。

20. Java Flight Recorder 飞行记录仪

01. 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；

（2） 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。