GIAC | 全球互联网架构大会
GLOBAL INTERNET ARCHITECTURE CONFERENCE

Java 9 In Practice

杨晓峰，Java Core Libraries

2017.thegiac.com

杨晓峰

- Principal Member of Technical Staff, OpenJDK Committer, Oracle Java Platform
- 目前领导Java核心类库北京团队，负责部分Java核心类库新特性相关任务
- 个人兴趣主要关注Java等语言在云计算等前沿领域的演进

日程

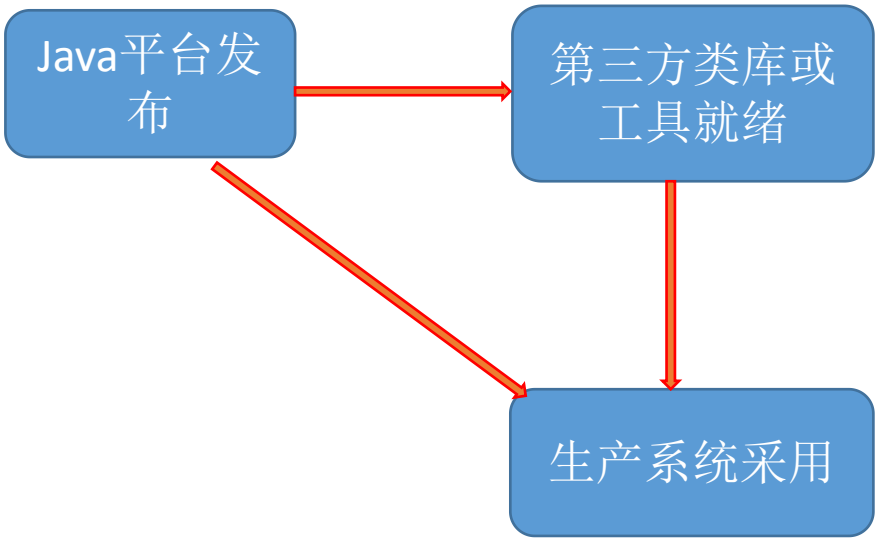
- 让应用在Java 9上跑起来！
- JPMS模块化尝试
- Flow API和JVM特性实践

预备知识

- 假设您对Java 9新特性有基本了解
- 熟悉模块化的基本概念

山高路远坑深（多）？

- 其实兼容性差异并没那么夸张
- 基础平台的传导路径本就是曲折的



Java 9逐步得到了支持

- 工具:
  - 主流IDE已经提供了比较完善的支持
  - 构建等工具如Maven，对编译、打包等各阶段都提供了支持
- 部分类库或开发框架提供了支持或相应计划，如：
  - 后面举例的一些Apache项目
  - 随便查了一下，如Springboot 2.0
    https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-with-Java-9
  - 请根据实际需要查询

让应用先能跑起来（构建和运行）

- Java 9移植中的可能坑儿
  - 版本号影响不小
  - JDK内部API封装
  - 适应JDK的结构变化
  - Classloader架构和行为的变化
- 更多请参考:

  https://docs.oracle.com/javase/9/migrate/toc.htm
  https://github.com/CodeFX-org/java-9-wtf
  https://blog.codefx.org/java/java-9-migration-guide/

Java和内部组件版本号变化(1)

- JEP 223: New Version-String Scheme
  - http://openjdk.java.net/jeps/223
- 为什么这么复杂？
  Java生态需要太多的信息:
  - 主版本
  - 更新
  - 安全级别
  - build号
- 好一些项目躺枪，后面提到的Cassandra和Log4j、Maven等无一幸免...

Java和内部组件版本号变化(2)

- Java自身version：
  - 以前：1.8.0_66-b18
  - 现在：9-ea+121
  - 如果是正式发布版本没有“-ea”
- java.vm.version：
  - 以前：25.92-b14
  - 现在：9-ea+121
- 后面还要改...

Cassandra实验（1）

- 基于master branch + Java 9， build+launch
- 构建阶段就无法通过

```
[javac] /home/felix/cassandra/src/java/org/apache/cassandra/utils/Throwables.java:80: error: unreported exception
Exception; must be caught or declared to be thrown
[javac]       perform(Stream.of(actions));
[javac]            ^
[javac] /home/felix/cassandra/src/java/org/apache/cassandra/utils/concurrent/Locks.java:46: error: cannot find symbol
[javac]           unsafe.monitorEnter(object);
[javac]                 ^
[javac]   symbol:   method monitorEnter(Object)
[javac]   location: variable unsafe of type Unsafe
[javac] /home/felix/cassandra/src/java/org/apache/cassandra/utils/concurrent/Locks.java:52: error: cannot find symbol
[javac]           unsafe.monitorExit(object);
[javac]                 ^
[javac]   symbol:   method monitorExit(Object)
[javac]   location: variable unsafe of type Unsafe
```

Cassandra实验（2）

- Java 9 Type inference bug
  - JDK-8160244, capture conversion行为有瑕疵

```
// Compile well in Java 8
 public static <E extends Exception> void perform(DiscreteAction<? extends E> ... actions) throws E
   {
      perform(Stream.of(actions));
   }
   public static <E extends Exception> void perform(Stream<DiscreteAction<? extends E>> actions) throws E
   {
      ...
   }

# workaround
   Throwables.<E>perform(Stream.of(actions));
```

Cassandra实验（3）

- 引用JDK内部API -- Unsafe.moniterEnter()/monitorExit()
- 可能的解决办法(1)：

```
private volatile long lock;
private static final AtomicLongFieldUpdater<AtomicBTreePartition> lockFieldUpdater =
   AtomicLongFieldUpdater.newUpdater(AtomicBTreePartition.class, "lock");

private void acquireLock(){
   long t = Thread.currentThread().getId();

   while (true){
      if (lockFieldUpdater.compareAndSet(this, 0L, t))
         return;
      // park for a while
       …
   }
}

private void releaseLock(){
   long t = Thread.currentThread().getId();
   boolean r = lockFieldUpdater.compareAndSet(this, t, 0L);
   …
}
```

Cassandra实验（4）-- 使用Java 9 VarHandle API

```
private long lock;
private static final VarHandle lockHandle;
static {
   try {
      lockHandle = MethodHandles.lookup().findVarHandle(AtomicBTreePartition.class,
            "lock", long.class);
   } catch (NoSuchFieldException |IllegalAccessException e) {
      // Handle as your requirement
      throw new Error(e);
   }
}
private void acquireLock(){
   long t = Thread.currentThread().getId();
   while (true){
      if (lockHandle.compareAndSet(this, 0L, t))
         return;
      // park for a while
       …
   }
}
private void releaseLock(){
   long t = Thread.currentThread().getId();
   boolean r = lockHandle.compareAndSet(this, t, 0L);
   …
}
```

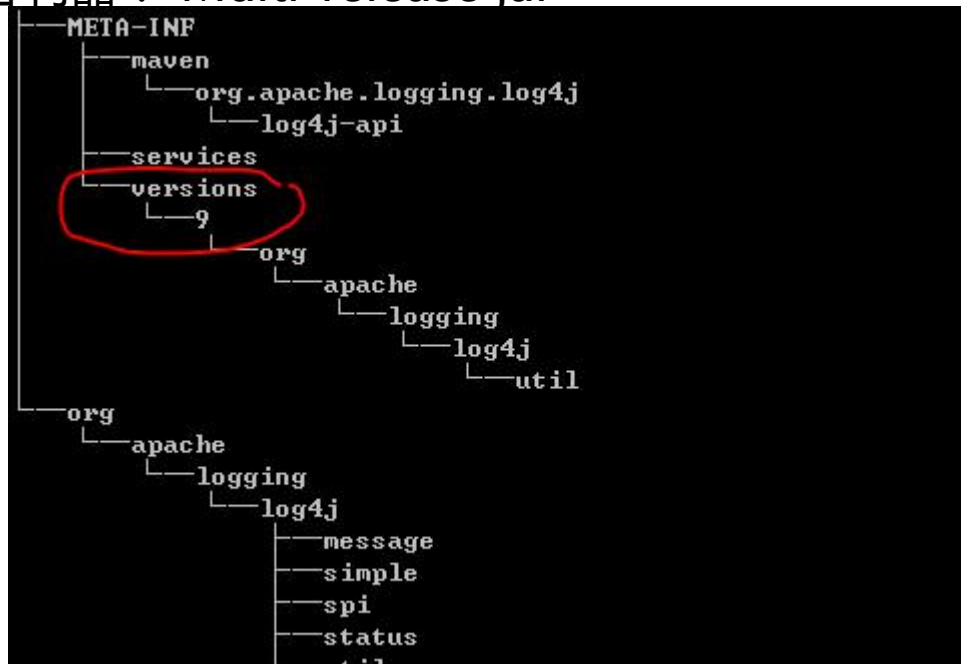
Why VarHandle API?

在提供底层能力的同时，JDK API实现保证了高标准：

- 安全，避免JVM内存corruption. 如：
  - 自动更新的类型要保证是兼容的（castable）
  - 数组元素操作不能越界
- 一致性： 如，读写操作贯彻一致的访问规则
- 高性能： 与Unsafe相似或对等的操作性能，并且，要保证生成的汇编代码大致是一致的，不会优化掉安全检查
- 易用性:与Unsafe相比，使用起来要简单、方便很多

处理版本间API不兼容（1）

- 怎么才能无缝的为不同Java版本实现不同代码？
- Cassandra： “We rely on sun.nio.ch.DirectBuffer.cleaner().clean() to free memory mapped files. DirectBuffer.cleaner() before Java 9 is returns a sun.misc.Cleaner, but Java 9 returns jdk.internal.ref.Cleaner ”
- Log4j：使用StackWalker和新的Process API
- 兼容利器： Multi-release jar



处理版本间API不兼容（2）

- 怎么编译不同Java版本代码？需要准备一系列JDK吗？
- 兼容利器(2)： JEP 247: Compile for Older Platform Versions
- Javac --release N
- 等价于：Javac -source N -target N -bootclasspath rtN.jar
- Javac拥有历史版本的API信息
  - 以特定格式压缩存储
  - 支持平台中立的公共API

处理版本间API不兼容 -- Maven的支持

```
<artifactId>maven-compiler-plugin</artifactId>
  <version>xxx</version>
  <executions>
   <execution>
    <id>default-compile</id>
    <configuration>
     <jdkToolchain>
      <version>9</version>
     </jdkToolchain>
     <release>9</release>
    </configuration>
   </execution>
   <execution>
    <id>base-compile</id>
    <goals>
     <goal>compile</goal>
    </goals>
    <!-- recompile everything for target VM except the module-info.java -->
    <configuration>
     <excludes>
      <exclude>module-info.java</exclude>
     </excludes>
    </configuration>
   </execution>
  </executions>
```

- 支持 --module-path
- 两阶段编译，以处理module-info
- 利用--release控制编译等级

实践中还是有坑儿！

- 现有的个别开发工具：
  - 会去META-INF/versions下寻找class文件
  - 暂时还不能忽略module-info.class
  - 如 Android dx
- 需要相关工具逐步进行修正

JDK结构变化的影响

- 新增了conf目录，请注意相关配置文件
- 很多项目依赖于rt.jar, tools.jar, 如:
  - IDE
  - 基于Java的其他语言平台（如Jython）
- 需要一种新的方式：
  - 获取JDK自身类文件
  - 低版本Java理解高版本Java

JDK结构变化的影响

- 解决办法

```
// JDK8可以使用${JAVA9_HOME}/lib/jrt-fs.jar
   Map<String, String> env = new HashMap<>();
    env.put("java.home", System.getProperty("java.home"));
    FileSystem fs = FileSystems.newFileSystem(URI.create("jrt:/"), env);
    Path sampleClass = fs.getPath("/modules/java.base/java/lang/Object.class");
```

Classloader变化(1)

- Bootstrap + Platform + Application classloaders
- 部分代码被降级，具体可以参考：
  http://java9.wtf/class-loading/
  如，java.sql.*, javax.activation.*,jaxb, jax-ws ...
- 解决办法：
  使用--add-modules module1,module2...

Classloader变化(2)

- URLClassLoader风光不再
- 曾经广泛使用的 Class::getResource* 、 ClassLoader::getResource* 不再能获取JDK内部资源
- 解决办法：
  使用Module::getResourceAsStream 访问相应模块内资源

- -Xbootclasspath被弱化，问题是怎么在初始阶段插入定制代码？
- 解决方法： --patch-module

前面解决的主要是兼容问题，下面试下模块化

尝试利用JPMS模块化(1)

- 理想状态：
  假设目前的jar模块划分合理，直接jdeps生成模块描述符，按照相应结构进行重构
- 但，现实世界是非常复杂的！
  - 就算不考虑目前包划分是否合理
  - 大型项目，如Hadoop，Cassandra，要做大量工作才能可能生成部分
  - 即使，较小规模项目,如log4j, guaua，也比想象中更复杂

尝试利用JPMS模块化(2)

- 如相对简单的log4j

```
${JAVA_HOME|/bin/jdeps --generate-module-info mods lib/*.jar dist/*.jar
writing to mods/stax2.api/module-info.java
writing to mods/commons.csv/module-info.java
Missing dependence: mods/log4j.core/module-info.java not generated
writing to mods/jeromq/module-info.java
…
writing to mods/javax.persistence/module-info.java
Missing dependence: mods/slf4j.api/module-info.java not generated
Missing dependence: mods/woodstox.core/module-info.java not generated
ERROR: missing dependencies
org.apache.logging.log4j.core.config.plugins.convert.TypeConverters$ByteArrayConverter ->
javax.xml.bind.DatatypeConverter                not found
  org.apache.logging.log4j.core.net.SmtpManager      -> javax.activation.DataSource
not found
...
```

尝试利用JPMS模块化(3) -- Split packages问题

- 在实验的大部分开源项目中：
  - 不同jar包具有同名package是比较普遍的
  - 将测试用例和源代码放置同一package下也是常见的实践
- 建议和可能办法：
  - 将公共类库抽象出来，进行重构和合并
  - 对于需要package级别访问的情况：
    - 可以创建代理接口
    - 利用模块更新（--patch-module）机制

尝试利用JPMS模块化(4) -- 其他不建议的情形

- Circling references并不鲜见
- Unnamed packages -- 即使Java初期就不推荐...

```
//模块化Cassandra时的问题
${JAVA_HOME}/bin/jdeps --gen-module-info src/ *.jar ./lib/jars/*.jar
Exception in thread "main" java.lang.module.FindException: Unable to derive module descriptor
for: ./lib/jars/hsqldb-1.8.0.10.jar
    at java.lang.module.ModulePath.readJar(java.base@9-ea/ModulePath.java:552)
    …
    at com.sun.tools.jdeps.Main.main(jdk.jdeps@9-ea/Main.java:48)
Caused by: java.lang.IllegalArgumentException: Empty package name
    …
```

尝试利用JPMS模块化(5) -- 新的服务绑定机制逐渐被采用

- Log4j java 9已经采用了SPI和ServiceLoader机制
- 项目（厂商）间商定统一的服务接口，分别提供服务的不同实现

```
//声明使用
module org.apache.logging.log4j {
    exports org.apache.logging.log4j;
    exports org.apache.logging.log4j.spi;
    // …
    exports org.apache.logging.log4j.util;
    uses org.apache.logging.log4j.spi.Provider;
}

// 其他模块实现SPI，细节不在这里展开
```

Automatic module是很多开源项目目前阶段的选择

- Automatic Module是为兼容性考虑的过渡手段
- 将jar包放置在module path下，JDK会为其生成一个名字
- 开发中，可以通过打包工具进行定制
- 简要规则如下：
  1. 如果JAR main manifest有属性“Automatic-Module-Name”，就用该值命名。
  2. 否则移除“.jar”后缀：
     利用表达式“-(\\d+(\\.|$))”匹配，前后段经简单处理作为名和版本
  3. 如：“foo-bar-1.2.3-SNAPSHOT.jar” 意味着
     模块名：“foo.bar”
     版本号："1.2.3-SNAPSHOT"

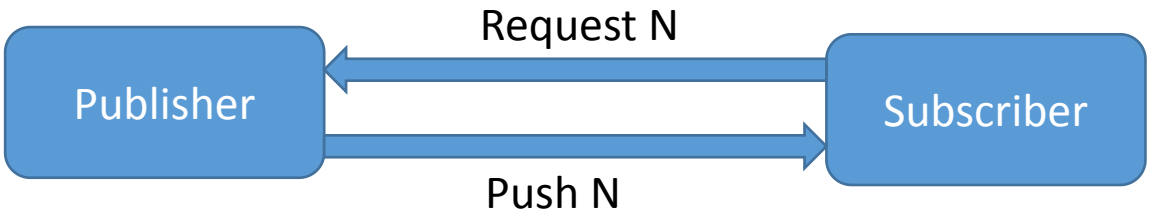
尝试利用JPMS模块化 --总结

- JPMS认为不合法的情况，其实满世界都是~~
- 社区实践反馈是JPMS进一步完善的关键
- 模块化注定是持久战，而且未必是适合每个人的选择
- 更现实的是：
  - 使用Automatic module机制来平滑过渡
  - 从核心的类库等部分开始，逐步模块化

生活中还有诗和远方
-- 谈谈其他特性吧

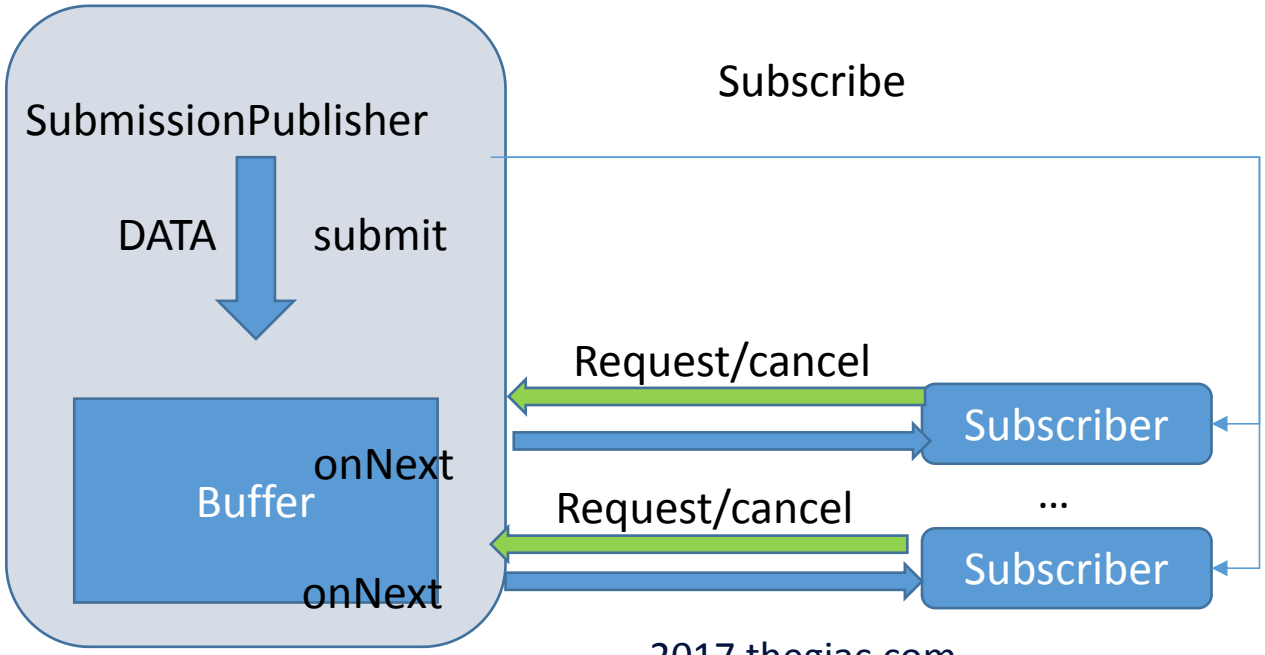
Flow API初探

- Reactive Stream的最小集合API（Flow API）
- 在Java语言层面提供相应抽象，可以作为不同的Reactive框架交互的媒介
- Publish-Subscribe模式进行Flow（PUSH/PULL-based）控制
- 更多细节阐述可以参考相关API文档，或这篇介绍



Flow API的能力范围

- Java 9并没有提供Concerete Processor实现
- 提供了一个参考Concerete Publisher实现
  - java.util.concurrent.SubmissionPublisher
  - 如果可以估计subscriber数量，那么可以指定Executors.newFixedThreadPool(int)
  - 默认使用ForkJoinPool.commonPool()



Java 9内部使用

- 介绍一下JEP 110： HTTP/2 Client API中的部分抽象

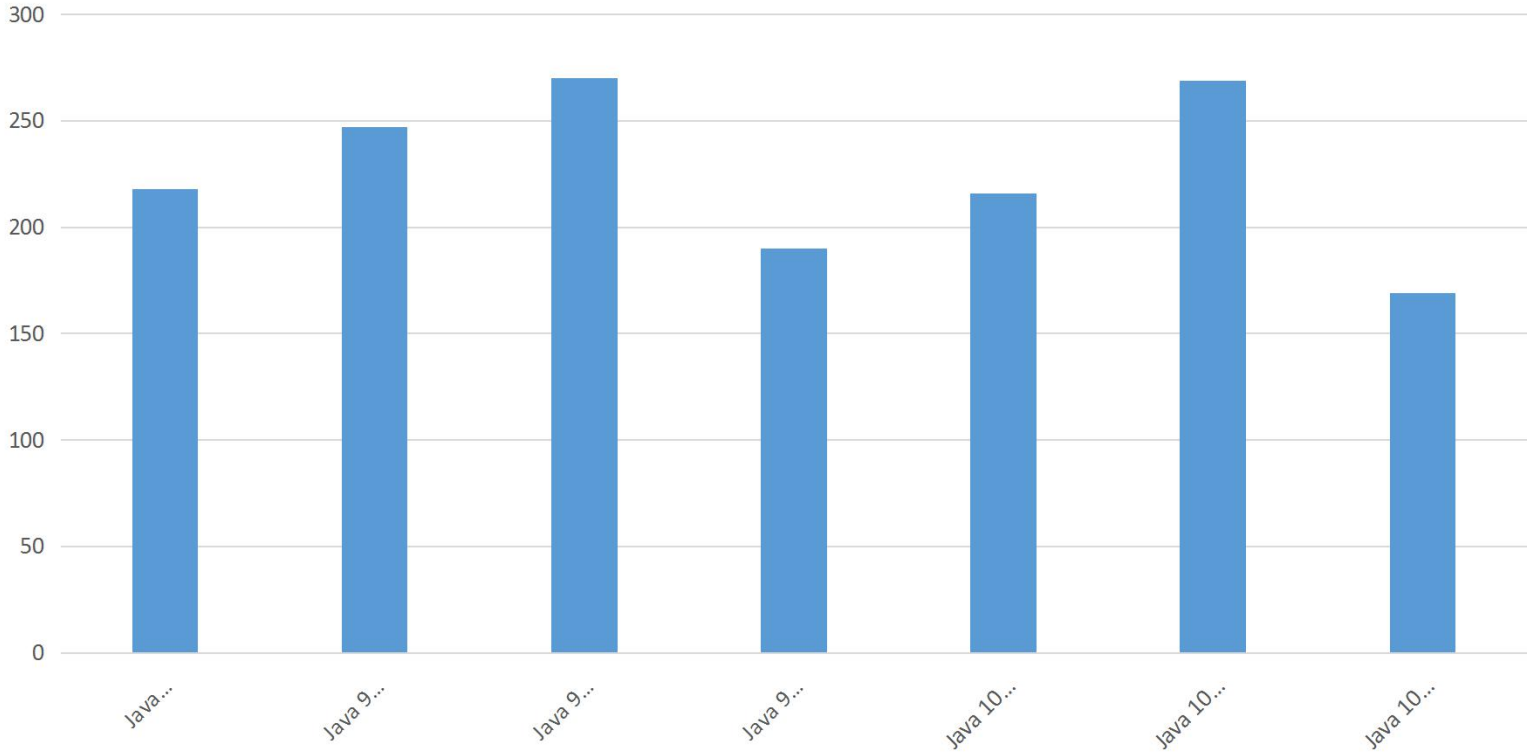
```
// 作为Flow.Publisher, 处理outgoing data, 即发送request body
// 将high level Java objects转换为ByteBuffer flow.
jdk.incubator.http.HttpRequest.BodyProcessor

// 作为Flow.Subscriber
// 将接收到的数据从ByteBuffer转为普通数据对象
jdk.incubator.http.HttpResponse.BodyProcessor

// Flow control的逻辑还是根据需求自己实现
```

Flow API使用社区反馈

- Reactive未必就一定是memory-efficient
  - 它只是从架构上，提供了反馈和协商处理能力的机制
  - 在本文提到的实践中，如何有效利用ByteBuffer是个基本问题（JDK-8186750）
  - 当然这本质不是Flow API导致的
- Publisher/Subscriber实现，需要考虑和其他Reactive框架的交互：
  - http://mail.openjdk.java.net/pipermail/net-dev/2017-December/thread.html#11063
  - 解决思路是提供工具方法进行适配
- 请注意异常处理的建议policy

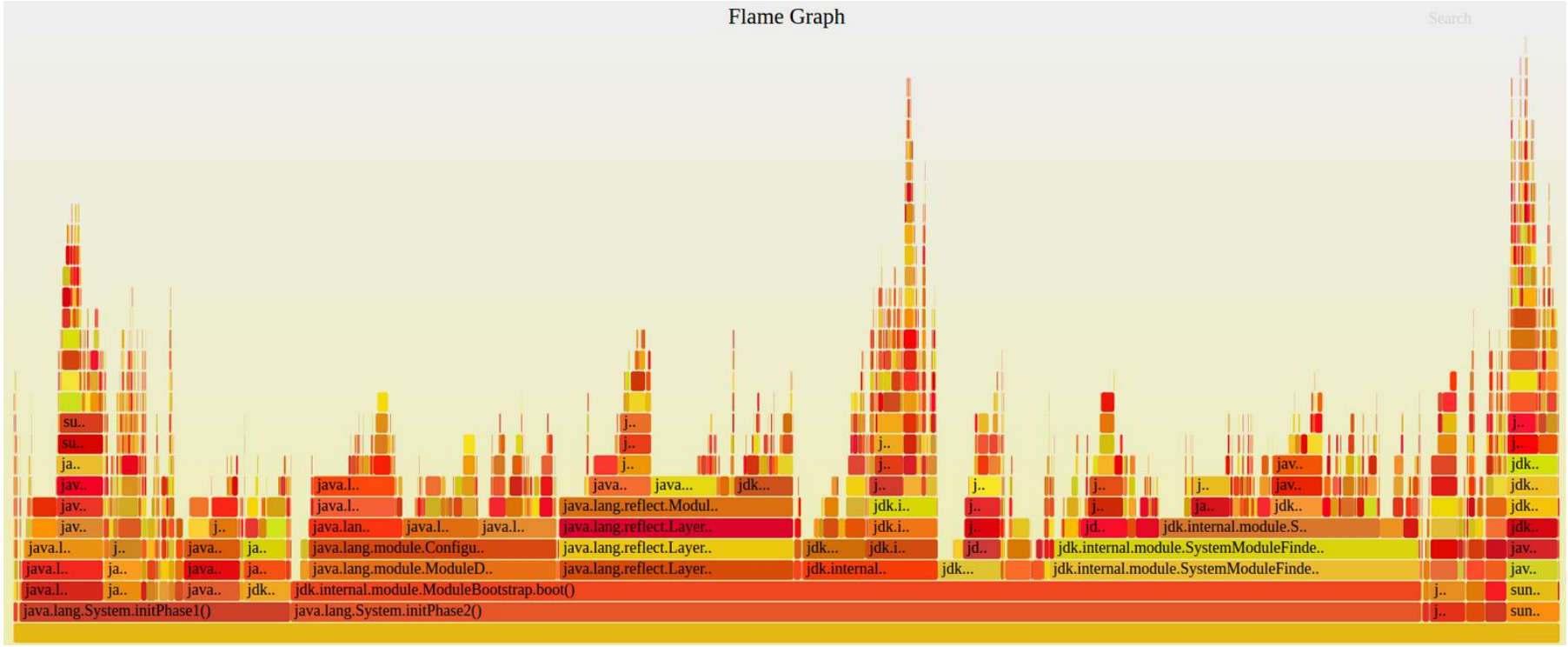
Java启动性能分析

- Java 9默认情况启动稍稍变慢了:
- 可以参考https://mjg123.github.io/2017/10/02/JVM-startup.html



理解Java 9启动阶段做了什么（1）

- Java fastdebug build可以利用-XX:+TraceBytecodes -Xint收集比较精确的stack
- 然后用FlameGraph作图：https://github.com/brendangregg/FlameGraph



理解Java 9启动阶段做了什么（2）-- initPhase1

- 这一阶段算是VM初始化的相对早期，紧接着Thread初始化
- 初始化System Class
- 初始化操作系统和其他环境变量
- 设置VM基本配置，比如某些cache，DirectBuffer最大值等
- 输入输出等
- 初始化Level 1

理解Java 9启动阶段做了什么（2）-- initPhase2

- 初始化Java 平台模块化系统
- Java.base
- 初始化Level 2
- 这一阶段是总体比较耗时的

Java启动性能优化

- 从JDK自身来说，一个简单的优化思路就是
  -- 能不做就不做，能晚做就晚做
  -- 或者简单点 -- 赖，拖~~
- 对于应用来说：
  - 不同GC也有区别，比如G1就通常比Parallel GC初始化慢少许
  - Jlink定制Runtime体现了优势
  - 充分利用JVM优化：
    - 利用jlink定制runtime
    - CDS/AppCDS
    - AOT

CDS + AppCDS(商业特性)

- 内存映射JVM metadata，避免class-loading开销
- 改进启动速度和Footprint
- 用例：

```
#生成映射归档文件
java -XX:SharedClassListFile=<class_list_file> -Xshare:dump

#打开 CDS
java -Xshare:on ...
```

AOT(Ahead of Time Compilation)

- 将Bytecode转换成native code
- 实验性，Java 9仅支持Linux x86
- 用例：

```
#编译类库
jaotc --output java_base.so --module java.base

#改进启动
java -XX:AOTLibrary=./java_base.so yourApp
```

AOT与动态编译

- AOT是静态的，能否利用JVM Compiler优化？
  - 如果应用是短暂的，JIT可能是不必要的负担
  - 长时间运行的应用，建议选择tiered-AOT
- 参数：

  --compile-for-tiered

  默认是关闭的

AOT编译期与运行时参数

- 部分运行时参数最好在编译期指明，比如GC相关
- 用例：

```
#编译类库
jaotc --output java_base.so --module java.base -J-Xmx4g

#改进启动
java -XX:AOTLibrary=./java_base.so -Xmx4g yourApp
```

AOT编译期与运行时参数

- 如何验证AOT确实生效？
- 用例：

```
java -XX:AOTLibrary=./java_base.so \
      -XX:+UnlockDiagnosticVMOptions \
      -XX:+UseAOTStrictLoading
      yourApp
```

GIAC | 全球互联网架构大会
GLOBAL INTERNET ARCHITECTURE CONFERENCE

GIAC

全球互联网架构大会

GLOBAL INTERNET ARCHITECTURE CONFERENCE

扫码关注GIAC公众号

2017.thegiac.com