写运行友好的代码

liach
Reviewer in JDK project in OpenJDK

什么是对运行友好？- 过去

- Java最早是30年前设计的
- 当时数字计算开销相对大，锁、指针、内存分配相对开销小
- JIT compiler 比较原始甚至不存在
- 同样思想的代码不管好坏，跑起来好像没有什么大差异……
- 对应设计模式：对象锁、可变对象加 volatile 字段、setter……
- 典型例子：StringBuffer、NumberFormat、setter 直接改后端数据库

什么是对运行友好？ - 现在

- 现在的运行环境大不同
- 数字计算效率高，指针、分配对缓存不友好，特殊锁支持也已经移除
- C2 特别重要，是 Java 程序运行快速“不输C++”的关键
- 错误代码写法和使用库会导致大幅性能下降！
- 对应设计模型：栈上进行数值变化操作然后分享全final对象（例如 record）锁用更高级数据结构封装，专门的单线程 builder
- 典型例子：java.time、ScopedValue、ClassFile 等各种新 API

写代码的注意事项 - 方法划分

- 方法长度、调用深度、分支量直接影响 C1 采样成本和 C2 编译效率
  - MH 表达式树为确保性能，只有一个 `selectAlternative` 支持分支
  - 控制方法长度、调用深度，减少单个方法中的分支
  - 方法调用处出现的实现太多叫 Profile Pollution，大幅影响性能
- C2 不可靠，就算有长期稳定测试还是有偶发情况导致 JVM 崩溃
  - JDK-8327247 字符串加号 MH 表达式树长度和深度太大，编译先大幅变慢，而后虚拟机崩溃
- 有些求值代码可以**常量折叠**，编译后成为一个常量值，运行也更快
  - 例如 MethodHandle 和 VarHandle 的一般用途

写代码的注意事项 - 方法划分

写小方法！把常用代码和非常用代码分段。

```
private int hashCode;

public int hashCode() {
    var hash = hashCode;
    if (hash == 0) {
        return hashCode = computeHashCode();
    }
    return hash;
}

private int computeHashCode() { /* 复杂计算 */ }
```

写代码的注意事项 - 方法划分

```
var hash = hashCode;
if (hash == 0) {
    return hashCode = computeHashCode();
}
return hash;
```

这里把复杂而且仅调用一次（忽略高并发情况）的的计算过程提取至单独方法 `computeHashCode`，这样 JIT 编译就能专心编译缓存路线，减少 JIT 编译使用的资源，同时可能优化最终编译的代码。如果 `hashCode` 字段能常量折叠，编译后甚至能直接移除对计算方法的调用。

写代码的注意事项 - 多态化

避免过分多态化，条件允许也可以用强类名帮助 JIT 编译器！很多情况如果 JIT 编译发现一段代码只有几种特定实现运行，那么编译时会专门对那个类进行优化。不然性能会大幅下降，叫“Profile Pollution”。

- 同一个接口的使用处出现很多实现，比如 Stream 的函数代码；
  - 函数式在此处的优点是执行的次数和线程自由，不然不需要函数式
- `Object.equals` vs `Objects.equals` — `AbstractCollection.contains`
  - `Arrays.equals` 的对象数组版也有相同问题。PR 14944
- 如果 JIT 编译发现一个接口或者抽象类只有一个实现，比如新 API 中的一些接口，JIT 编译能优化呼叫

写代码的注意事项 - 多态化

Hotspot 虚拟机在多态化统计时会计算每个字段中各种实现的出现频率。但是统计对象不包括数组元素，所以固定长度数组用单独字段表示在 C2 编译后会获得更高的性能。

```
private CharSequence s0, s1, s2; // 如果某一个成员是固定类型，这个更快
private final CharSequence[] strings = new CharSequence[3];
```

有时单独字段优势极其明显，以至于为常见的数组长度单独生成类，获得更高的运行性能。当然这样的代码不容易维护，所以条件允许推荐通过程序生成此类代码。（OpenJDK 社区成员也可以问问有没有考虑过针对固定数组元素进行 Profile 测量）

写代码的注意事项 - lambda 表达式

Java API 中 lambda 表达式的主要用途是封装代码块。如果代码块没有独立封装的必要，可以合并 lambda 表达式或者写直接的代码。

- `Optional.orElseThrow` 避免立刻计算错误栈
- `Stream` 保证操作以深度优先顺序执行，同时如果无最终操作则避免调用操作
- `Comparator` 封装可多处复用的比较代码

没必要则避免使用 lambda 表达式！源代码中每个 lambda 表达式都是一个独立类。有的 lambda 表达式还需要从环境获得本地变量，实际开销等同于匿名类。比如 for 循环够简单就不需要用 Stream。

写代码的注意事项 - lambda 表达式

例如

```
stream
    .filter(String.class::isInstance)
    .map(String.class::cast)
```

可以合并，改写成更轻量的：

```
stream
    .<String>mapMulti((item, sink) -> {
        if (item instanceof String s) {
            sink.accept(s);
        }
    })
```

写代码的注意事项 - 逃逸分析

逃逸分析（Escape Analysis）是 IR 编译器计算如果一个对象没有在栈外使用，那么这个对象可以简化成一个栈上对象，类似值对象。

`for (Item item : list)` 这种用 `Iterator` 实现的加强循环是现在逃逸分析处理的主要对象。有了逃逸分析后，遍历 `List.of` 才能比数组下标遍历数组更快。

- 虽然逃逸分析能够移除大多数 `Iterator` 的额外对象开销，但是有的 `Iterator` 还是有非对象的额外开销
- `ArrayList` 遍历有额外 `modCount` 检查，所以 `Iterator` 遍历还是比用下标遍历慢。

写代码的注意事项 - 逃逸分析

逃逸分析要优化的对象类，例如 `Iterator`，最好不要继承上级类！

- 继承上级类可能会导致在共享代码中出现 Profile Pollution！
- C2 的 profiling 测量是以 Java 字节码为单位的！（MH 除外）
- https://github.com/openjdk/jdk/pull/15615#issuecomment-1710058719

C2 的逃逸分析相对比较基础。据说 Graal JIT 中逃逸分析更完善，针对 `Stream` 和各种 `Builder` 等的效果也更佳；不知道这里的限制是否还存在。

写代码的注意事项 - 对象身份

非必要不依赖对象身份（Object identity）！

- 对象身份可以理解为一个不可变实例字段
- 如果 C2 发现不使用对象身份（==、identityHashCode、对象锁）就能解锁很多性能优化，例如 Vector API 对此有依赖
- 同时为 Value Objects 升级做准备！这也是 Valhalla 中的重要概念。
- 用 `equals`，避免用 `==`！甚至 `equals` 实现都避免头上加个 `if (this == o)`！
- Optimizing your equals() methods with Pattern Matching - JEP Cafe #21

写代码的注意事项 - 读字段

读字段和数组元素应只读一次然后存入本地变量。生成的字节码更小，而且线程安全，可以保证结果一致性。第二次读取结果可能因多线程改动或 CPU 重排序而不同或者提前。

```
var hash = hashCode; // 两次读取字段，则指令重排后，后一个读可能出现 0
if (hash == 0) {
    return hashCode = computeHashCode();
}
return hash;
```

- JDK-8311906 String 构建时读 char 数组相同位置两次获得不同结果导致创造非正确状态的 String

写代码的注意事项 - 数组读写

有些操作需要处理连续数组元素，比如字符串读写。可以注意以下一些事项：

- 指针变动先读字段，存本地变量，操作全部完成后再从变量存回字段
- 本地变量用 `+1` `+2` `+3` 等而非 `++` 变动
- 两种技巧都生成的字节码更小，和堆的交互更少，对 CPU 指令重排序优化更友好
- 同时连续对数组读写现在在 C2 中有 MergeStore 优化，减少零碎内存读写操作，效率更高

写代码的注意事项 - 数组读写

系统库中 `BufWriterImpl` 的例子：

```
public void writeU2U1(int x1, int x2) {
    reserveSpace(3);
    byte[] elems = this.elems;
    int offset = this.offset;
    elems[offset    ] = (byte) (x1 >> 8);
    elems[offset + 1] = (byte) x1;
    elems[offset + 2] = (byte) x2;
    this.offset = offset + 3;
}
```

写代码的注意事项 - 数组读写

系统库中提供了 `MethodHandles.byteArrayViewVarHandle` 提供地址不对齐多字节读写功能，提升读写性能。比 MergeStore 优化更可靠。

- 是 `Unsafe.get/setXxxUnaligned` 的封装
- 用此工具可以快速读取 `byte[]` 内容，读取字符串时可以靠前面几位进行排除
  - 比如读取 JSON 字符串，可以用前几位判断分支，如果只有一种可能就进行大块匹配，加速随后读取速度
  - 这是 fastjson 使用的技巧，适用于字节码生成
  - 向量 API 也许可以如此使用，但是要考虑硬件上的向量传输成本

写代码的注意事项 - 数组查询表

有些查询表可以用数组下标快速访问，用数组元素储存信息。
这些数组有时候长度接近 2 的倍数，比如字节码编号，此时可以用一个小技巧加快访问。

- 数组定义时定义长度为 2 的倍数，比如 256
- 访问前对下标编号进行位运算，比如 `i & 0xFF` 运算
- 如此 C2 便可以推断下标永远不会越出数组长度，编译后会移除抛出 AIOOBE 报错的数组下标检查，加快数组读写。

```
private static final int[] cache = new int[256];
return cache[i & 0xFF];
```

写代码的注意事项 - 冷加载代码

Java 平台的扩展性很好，任何代码都可以定义新类加载器 ClassLoader 加载任意类，甚至支持用 Lookup 加载 hidden classes。

- 要消耗资源准备字节码，加载的类只能 JIT 编译，造成额外开销
- 一些之前运行中未被 JIT 编译的分支代码也可能产生类似开销
- 有的平台是封闭环境，可以证明代码一致，激进复用之前的编译和测量结果，但是对通用的 Java 代码不适用。
- 这种冷代码也一般是病毒的加载途径，见 log4j

检查运行效果 - 检查 Java 字节码

如何检查最后的字节码？用 `javap -c -p -v` 解构 `class` 格式文件找到想要的方法，里面的最后指令编号加1（各种return和athrow的长度）一般就是方法的长度。

- C2 编译对字节码长度敏感，争取控制在 325 之内。
- 有些字节码很大！尤其是 `switch` 生成的 `tableswitch` 、`lookupswitch` 字节码。写 `switch` 记得检查生成的字节码大小！尤其是 `switch` char，中间很多空 case！
  - JDK-8335252 (PR 19926) `Formatter.Conversion::isValid`
- `javap` 也可以反编译自己的运行环境中的类，直接输入类名即可

检查运行效果 - JIT 编译情况

`java` 命令行介绍了一些 JIT 编译器的参数，可以观测运行时代码的编译情况。这些都需要 `-XX:+UnlockDiagnosticVMOptions` 解锁，各个命令都需要 `-XX:` 前缀。

- C2 常用的参数在 `c2_globals.hpp` 中。
- `+PrintInlining` 会输出调用方法树中方法的编译情况。
- `FreqInlineSize` 针对高频方法编译的字节码大小限制，默认 325。
- `MaxInlineSize` 针对一般方法编译的字节码大小限制，默认 35。
- `MaxInlineLevel` 编译的最深调用嵌套层数，不在文档中，默认 15。
- 这些值调整会加减编译范围同时影响编译开销，可按需求选择调优。

检查运行效果 - JIT 编译情况

自从 23 起 Oracle JDK 发布自带 Graal JIT，文档也有介绍。它有另一套调优参数。

- OpenJDK 有 Galahad 项目，Graal JIT 有可能像 Lilliput 和 Shenandoah GC 并入 OpenJDK 的 JDK 项目本身
- 据说 Graal JIT 差不多全面比 C2 更强，个人没试验过
- Graal JIT 使用的 Graal 编译器也用于 AOT 编译和 GraalVM 编译其他语言，可能比 C2 测试更多更稳定

检查运行效果 - 虚拟机日志

Java 中类加载和初始化是影响启动性能的关键因素之一。JVM 有一套日志系统可以汇报各种信息。

- 比如 `-Xlog:class+init` 可以汇报类加载顺序，确认启动未加载类，确保启动性能

运行友好的 API

- 自从 Java 7 JSR 292 以来，Java 平台添加了很多为运行效率设计的新接口和库
- MethodHandle、VarHandle、Foreign Function and Memory API、未来 Vector 向量 API、Stable Values 稳定值、strict final严格不可变字段等
- 甚至新的语言特性都对运行友好
  - sealed classes 有助于限制多态化取样；record 字段默认强不可变，可以常量折叠

运行友好的 API - 通用新接口

一般高版本 API 都比之前和其他库提供的同类 API 优化更好

- `DateTimeFormatter` 性能优于 `SimpleDateFormat` 和 apache common 或者 joda-time 的转换
  - `java.text.Format` 整体较老，设计和实现不运行友好
- 之前对象身份例子，pattern 优于现有库，避免 Profile Pollution
- 字符串模板 `FMT` 优于 `String.format`
- `MethodHandle` 优于 `Method.invoke`

运行友好的 API - MH & VH

MethodHandle 和 VarHandle 包含语法糖方法，语言上支持用任何调用类型调用。

- 运行时链接至不同的实现方法
- MethodHandle 支持类型转换等，有自己的解读运行系统支持懒生成
- 在 Java 核心库中，主要用途是它的编译后性能，同时允许老版本代码调用新版本平台时生成新版本字节码
- 例如 enum 是编译器生成，有些启动代码无法优化；enum switch；record 的hashCode equals toString 是 indy，平台可以提供最新最优化实现

运行友好的 API - MH & VH

MH 和 VH 实现有优化机会，很多与 Leyden 相关。

- `InnerClassLambdaMetafactory` 可能同一接口用共享类实现
  - 字段存 MH，类似现在的新 `MethodHandleProxies`
  - 减少初始化和单次开销（比如 `orElseThrow`）
- 其他一些加快冷启动的优化，主要是 `VarHandle`
- Leyden 可能缓存用户 BSM、hidden class
- JEP 303?

运行友好的 API - 常量折叠

MethodHandle 和 VarHandle 运行快速有很多注意点，最重要的是常量折叠（constant folding）

- 整个调用链必须全能常量折叠
- 见 ciField.cpp `ciField::initialize_from` `_is_constant`
- 如果有个非常量值，例如 `MethodHandle::asType` 类型转换，其中缓存非常量，性能就会大幅下降！
- VarHandle 呼叫中还有分支，可能对常量折叠失败更加敏感

运行友好的 API - 常量折叠

常量折叠能大幅提升 JIT 编译后代码性能！
可以常量折叠的字段：

- static final 字段
- 未来的 strict final 字段 （Valhalla）
  - 现在的 record 成员字段
- `@Stable` 字段（影响数组）
  - 未来的 Stable Values（Leyden，Panama）
- 未来的冷冻数组（Frozen Array）

运行友好的 API - 常量折叠

最简单的反射例子：

```
private static final MethodHandle myTarget;

static { /* 获得 MH，报错就报 ExceptionInIntializerError */ }

private boolean call(MyObject obj, int a, long b) {
    // 调用类型是 (MyObject, int, long) -> boolean
    return (boolean) myTarget.invokeExact(obj, a, b);
}
```

因为有常量折叠，这段代码编译后性能和直接调用 myTarget 指向的代码无异。（`Method.invoke` 有检查权限开销，只能部分常量折叠）

运行友好的 API - 常量折叠

举个例子，遍历枚举类成员的 JMH 测试

```
private static final MyEnum[] VALUES_ARRAY = MyEnum.values();
private static final List<MyEnum> VALUES_LIST = List.of(MyEnum.values());

@Benchmark
public void benchArray(Blackhole bh) {
    for (var me : VALUES_ARRAY) bh.consume(me);
}

@Benchmark
public void benchList(Blackhole bh) {
    for (var me : VALUES_LIST) bh.consume(me);
}
```

运行友好的 API - 常量折叠

测试特点：

- 两个成员来源对象 ARRAY、LIST 都是 static final 可以常量折叠；
- 预先调用 `values()` 避免运行时开销

运行结果是 List 遍历更快。为什么？

- List.of 实现的数组 `ImmutableCollections$ListN.elements` 有 `@Stable` 是常量数组
- Java 普通数组可变，不能常量折叠
- 加上逃逸分析消灭 Iterator 分配后 List 就常量折叠更快了

运行友好的 API - 严格字段

Valhalla 项目准备添加 strict field，严格字段。准备复用 ACC_STRICT，以前给 `strictfp` 用的参数。暂时还没有 JEP，可能在研究实现技巧。

- 严格字段保证在第一次读取前，已经写入了一个合法值
- 类字段通过类加载锁等保证无非法读取
- 实例字段通过在调用上级构造器前写入合法值
  - JEP 482：在调用上级构建器前写好字段！
- JVMLS 2024 Devoxx Belgium 2024
- 未来 record 实例字段或将变成严格字段

运行友好的 API - 严格字段

- 严格字段允许虚拟机能保证空指针安全
- final 严格字段可永远确保不变可以安全常量折叠
  - 上级构造器调用的覆写方法只能看到已经写入的值
- 普通严格字段也可定义额外写入检查
  - 可能对 JIT 编译 profiling 也有帮助
- 原意是封装未初始化的 value object，避免未初始化状态泄露
- 更早的历史：如果反编译内部类会发现 `this$0` 字段是在调用上级构造器之前写入的。这是 John Rose 二十年前发现后给内部类添加的……

运行友好的 API - 严格字段

- 大多数 final 字段能无痛移植严格 final
- 指针中有环结构的怎么办？
  - 可以用 Stable Value 稳定值，一个指针推迟设置
- 反射、序列化、unsafe？
  - 反射新检查，unsafe 退役、新序列化 Marshalling，关联 Pattern 语言支持

运行友好的 API - 稳定值

Hotspot 一直有一个 `@Stable` 注解，表示一个字段一开始是默认值（0、null、false）。如果这个值不是默认值，那么 C2 会认为读取的值是常量，编译会进行常量折叠。如果字段是数组类型，数组每个位置都相当于这样的可常量折叠字段。性能优化显著。此功能因为线程安全和多次赋值问题，是内部专用注解。

- 现有的接口有 `List.of` 和 `Map.of` 的实现用此注解，性能优异
- 计划用 Stable Value 稳定值 API 向用户导出用户可控常量折叠功能

Stable Value 准备在 25 开始 Preview。PR 19625

运行友好的 API - 稳定值

举例：`StableValue<String>` 字段相当于 `@Stable String` 字段。

- StableValue 有安全保证，防止用户多次赋值
  - 读写有线程安全检查，JIT 编译常量折叠后可以消除
- 有更高级封装，例如 memoized supplier、function、stable list。
  - memoized 记忆函数确保初始化函数只执行一次，有额外锁
  - stable list 是数组 Stable 的封装，类型 `List<StableValue<T>>`
  - stable map 也是封装，现在针对固定范围内的输入

运行友好的 API - 稳定值

老代码：

```
private static final class Holder {
    static final Item INSTANCE = new Item();
}
```

新代码：

```
// lambda 会影响冷启动性能，但是编译后能消除
private static final Supplier<Item> ITEM = StableValue.newCachingSupplier(Item::new);
```

基础的 `StableValue.of` 不提供初始化锁，更适合高级用户。

运行友好的 API - 稳定值

实现细节：

- final StableValue 字段现在在 ciField 常量折叠检查添加检查类型允许
- 如果严格字段实现，可能直接复用严格字段的常量折叠检查
- 暂时不清楚会不会像 `@Stable` 优化成定义类 layout 的一部分
- 对 Leyden 缓存友好，Leyden 可以直接缓存 StableValue 已有的值

用途：

- 在系统库中，可能会先替换所有的 holder class pattern
- 非核心代码的 `@Stable` 也有可能被替换成此封装

运行友好的 API - FFM

FFM 外部函数和内存 API 替代 JNI，用来调用 native 平台代码、操作平台内存，已经在 JDK 22 正式稳定。个人不是太了解。

- 现在 FFM 函数性能暂时不如 C2 里面添加的 intrinsic，有优化空间
- C2 现在无法给外部函数构建 IR 进行编译，不知道未来的计划
- 冷启动也有优化的机会，比如 VarHandle 成本

现在的性能测试都是测试长期热使用性能，对冷启动和一次性使用缺少测量和修复。但是 API 稳定，开发中有提升性能的机会。

运行友好的 API - Vector

Vector API 相当于直接编译器指令，进行向量化操作。现在在孵化。个人不是太了解。

- 用户模型对非专业用户不是特别友善，不过好像也难以提升
- 长期孵化，暂时没看到进展，也没看到对 Valhalla 的强依赖性
- 暂时不知道具体的发展方向是什么，这几年 AI 都大发展了
- Java 第三方库难以替代，还是需要是 JDK 的一部分，不然不如直接转 native
- 希望大家能提出对它的反馈

Leyden

因为 Java 的可扩展性，Leyden 现在思想是动态保存 Java 虚拟机的状态给虚拟机复用，可以避免处理很多一次性代码。

- 思想类似 CRaC，不过跨操作系统的支持更好
- 快照适合在 serverless 或抢占式实例运行
  - CRaC 或者 Graal native image 这种不需预热的也适合
- 现在好像只有 linking 数据，主线还没有开始保存 JIT 编译数据
  - 不过系统生成的 lambda 和字符串拼接类似乎支持

总结

- 新时代的设计理念
- 有益程序安全和性能提升的代码习惯和风格
- 测量性能的一些方法
- 性能相关 API 的现状和使用
- 未来的性能相关 API 和 Leyden
- 问题和反馈

感谢各位听众！