

JVM 实战

阿里云开发者学堂配套教材



掌握 Java 虚拟机原理，
学习 JNI、类加载器原理、safepoint 机制等知识





扫一扫
免费领取同步课程



钉钉扫一扫
进入官方答疑群



开发者学院【Alibaba Java 技术图谱】
更多好课免费学



阿里云开发者“藏经阁”
海量电子书免费下载

| 目录

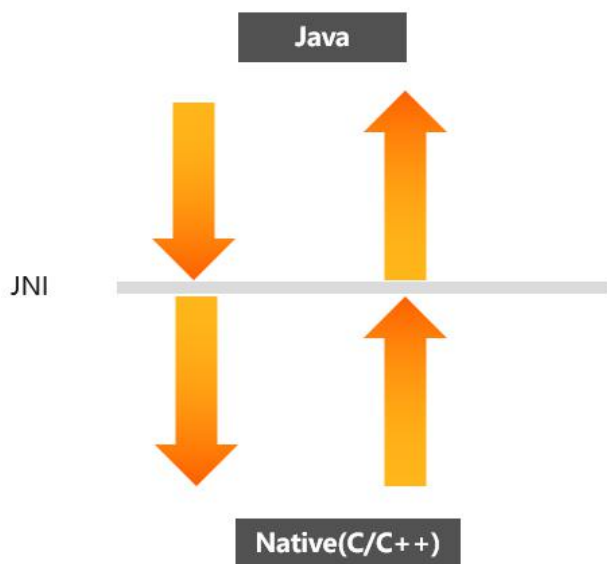
JNI in Java	4
Safepoint 机制	20
类加载器原理	31
Dragonwell 特性: 多租户	46
Dragonwell 特性: JWarmup	55
Dragonwell 特性: Wisp	62

JNI in Java

一、什么是 JNI

(一) 什么是 JNI (Java Native Interface)

JNI 全称是 Java Native Interface，顾名思义是 Java 和 Native 间的通信桥梁，如下图所示，图的上方是 Java 世界，下面是 Native 世界，中间是 JNI 通信，左边箭头从上往下是 Java 调用 Native 的方法，右边是 Native 调用 Java，彼此可以互通。



这种方式带来的好处，Java 调用 Native，可以去调用非 Java 实现的库，扩充 Java 的使用场景，比如调用 Tensorflow；反之 Native 调用 Java，可以在别的语言里面调用 Java，比如 java launcher 可以命令启动 Java 程序。

(二) 为什么要学习 JNI

掌握 Java 和 Native 之间的互相调用，大大丰富 java 的使用场景。了解原理，对于学习 JVM/故障定位更加得心应手。

经典例子，如下图所示，在主函数里面用 `Selector.open` 创建一个 `select`，叫 `select` 方法，这是 Java 里面通过 NIO 取允许网络的方法。

```
"main" #1 prio=5 os_prio=0 tid=0x00007f9d8fa4e000 nid=0xb62e runnable [0x00007f9d8fa4e000]
  java.lang.Thread.State: RUNNABLE
    at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
    at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:280)
    at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:96)
    at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
    - locked <0x000000008027b820> (a sun.nio.ch.Util$3)
    - locked <0x000000008027b798> (a java.util.Collections$UnmodifiableSet)
    - locked <0x0000000080276bf8> (a sun.nio.ch.EPollSelectorImpl)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:101)
    at A.main(A.java:5)
```

```
public static void main(String[] args) throws Exception {
    java.nio.channels.Selector.open().select();
}
```

这个方法会阻塞其当前线程，通过 `java.lang` 呈现状态是 `RUNNABLE`，看到 `RUNNABLE` 总觉得会消耗 CPU、NIO 的 BUG，其实是一个经典谬误，实际上线程是禁止的。

二、JNI 实践和思考

实战一：从 native 调用 Java

首先要#include <jni.h> ,这个头文件定义了各种 Java 和 Native 交互的数据结构以及定义；在主函数里面，首先声明一个 JVM 的指针，然后一个 JNIEnv *env 的指针，JVM 表示的 Java 虚拟实例，我通过实例消耗资源进行各种操作。

env 其实对应的是一个线程，然后创建 JavaVMInitArgs 结构体，结构体里面要填充 Java 参数，用 JavaVMOption 表示。因为这里不需要参数，场景比较简单，所以用 options[0]，把 options 传入 vm_args.options 结构体，最后调用 JNI_CreateJavaVM 创建 Java 虚拟机，如果返回的是“JNI_OK”，说明这次调用成功。

有了 JNI 指针表示实例以后，就可以用标准方法使用 JNI，在这里调用一个 Java 方法，比如 Java 数据结构，先通过 EMC 的 FindClass，找到 SelectorProvider 类，中间有个 printf 变量叫 lock，先通过 GetStaticField 获取 field，再通过 GetStaticObjectField 从 cls 对象上获取 fid，就是 lock 对象，然后把它打印出来，最后 jvm->DestroyJavaVM。详情操作如下图所示：

```

#include <jni.h>
#include <stdio.h>
#include <stdlib.h>

int
main()
{
    JVM *jvm;
    JNIEnv *env;
    JVMInitArgs vm_args; /* JDK/JRE 6 VM initialization arguments */
    JVMOption options[0];
    vm_args.version = JNI_VERSION_1_6;
    vm_args.nOptions = 0;
    vm_args.options = options;
    vm_args.ignoreUnrecognized = false;

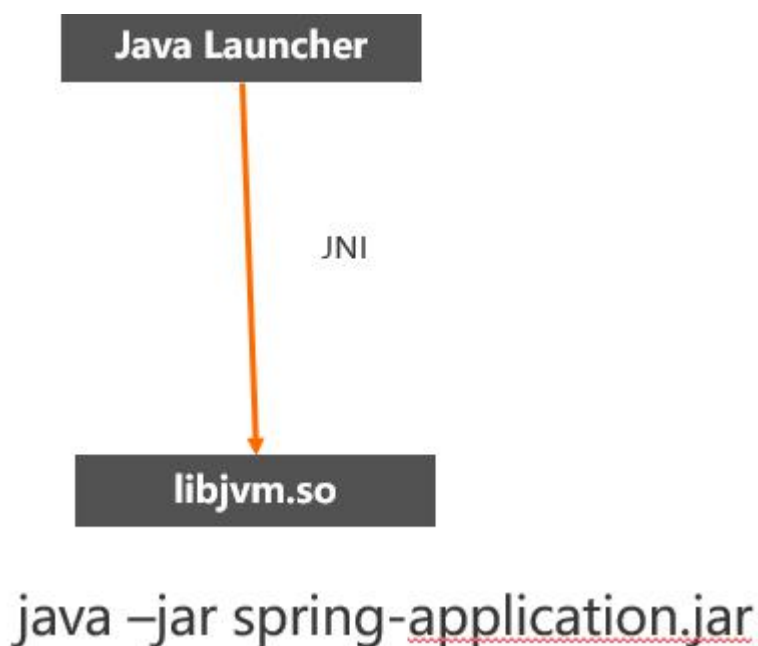
    if (JNI_CreateJavaVM(&jvm, (void**) &env, &vm_args) != JNI_OK) {
        exit(-1);
    }

    jclass cls = env->FindClass("java/nio/channels/spi/SelectorProvider");
    printf("class = %p\n", cls);
    jfieldID fid = env->GetStaticFieldID(cls, "lock", "Ljava/lang/Object;");
    printf("fid = %p\n", fid);
    jobject lock = env->GetStaticObjectField(cls, fid);
    printf("lock = %p\n", lock);

    return jvm->DestroyJavaVM() == JNI_OK ? 0 : -1;
}

```

还有一个比较经典的例子 Java Launcher, `java -jar spring-application` 执行程序的时候, 在后台默默的创建了一个 `jvm`, 把 Java 参数作为 `arguments` 传进去, 调用 Java 入口方法, 通过 JNI 实现。



平时所说，开发 jvm 其实就是开发 jvm 的动态库，“libjvm.so”基本上本身是作为“os”提供出去，好处是非常灵活，可以作为独立应用使用，也可以在别的像 cer 这样的语言调用，使 Java 调用 Native，Native 调用 Java 更加灵活。

JNI 实战二: Java 调用 C

Java 调用 C 是使用 JNI 最常见的方式，首先定一个类叫 HelloJNI，里面有 `System.loadLibrary("hello");` 系统会自动去找到 library libhello.so，这个类里面定义方法叫 say Hello，加了 C 以后调用它，但这是调不通的，因为并没有提供真正的 Native 实现。实现要通过一个头文件去告诉这个方法的签名，这里实现 Java 文件，然后通过 jni.h 生成头文件，这个是自动生成的。

签名是 Java，然后是 Java_HelloJNI_sayHello(JNIEnv *, jobject)规范，类名加上方法名，参数第一个是环境；第二个是 jobject，无参数，但是 Java 的方法默认是有一个 this 指针作为第一个参数，最后编写它，实现 HelloJNI.c，根据这个声明定义实现，然后里面只是 printf 了一下，把 HelloJNI.c 定义成 libhelloHello.so 这个程序就可以运行起来了。详情如下图所示：

```
public class HelloJNI {
    static {
        System.loadLibrary("hello"); // Load native library libhello.so
    }
    private native void sayHello();

    public static void main(String[] args) {
        new HelloJNI().sayHello();
    }
}

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>

#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      HelloJNI
 * Method:     sayHello
 * Signature:  ()V
 */
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif

#include <jni.h>          // JNI header provided by JDK
#include <stdio.h>         // C Standard IO Header
#include "HelloJNI.h"     // Generated

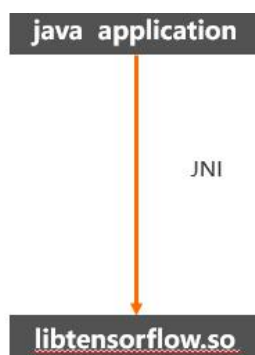
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
    printf("Hello World!\n");
    return;
}
```

HelloJNI.java

HelloJNI.h

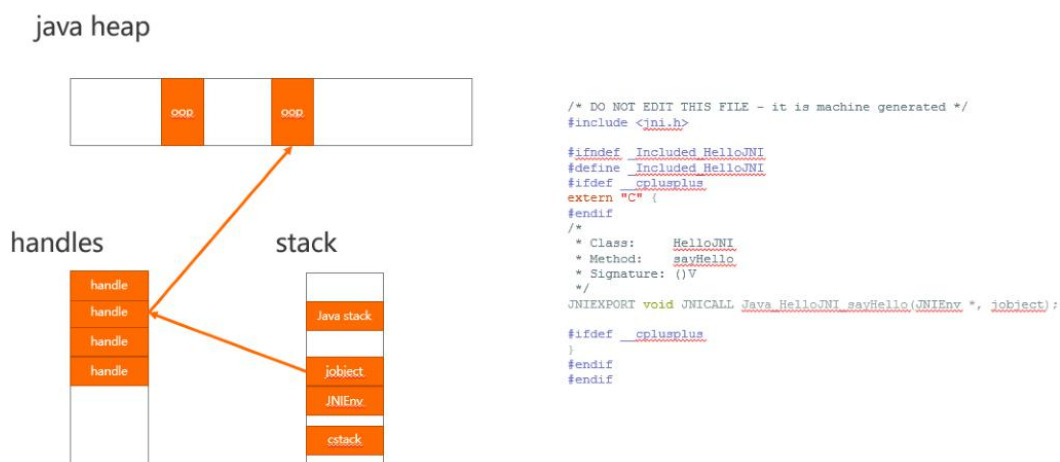
HelloJNI.c

在 Java 应用里面，可以调通过 JNI 调用各种库，调用到 native 以后，因为任何语言跟 native 都互相交互，大大丰富了 Java 使用场景。



思考一：Java 和 Native 的数据是怎么传递的

在执行 Java 方法时，用的是 java heap，假设暂时向下增长，需要调用 c 函数的时候，它需要去压站，把 object 压站、把 JNIEnv 压站、cstack 压站，进入 seat stack。然后 object 本质上是指向 handle 的指针，handle 指向战上真正的 OOP，使用二级指针结构，稍微有点复杂。详情如下图所示：



思考二: 回到问题, 为什么 select()的线程状态是 RUNNABLE

JNI 只是提供一种机制, 让 Java 程序可以进入 Native 状态, Native 状态基本上没有办法管理。这段 Native 代码在做一种非常复杂的数学运算, 肯定是 RUNNABLE 状态, 也可以调用系统形象去阻塞, 但这个阻塞基本上不知情, 所以会一直显示为 RUNNABLE, 除非通过 JNI 的特殊接口改变现实状态, 到其他状态才会显示为其他状态, 所以这里显示为 RUNNABLE 为正常, 不用担心 RUNNABLE 状态消耗很多 CPU 等问题。

```
"main" #1 prio=5 os_prio=0 tid=0x00007f9d8fa4e000 nid=0xb62e runnable [0x00007f9d9
java.lang.Thread.State: RUNNABLE
    at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
    at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:280)
    at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:96)
    at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
    - locked <0x000000008027b820> (a sun.nio.ch.Util$3)
    - locked <0x000000008027b798> (a java.util.Collections$UnmodifiableSet)
    - locked <0x000000008027b6f8> (a sun.nio.ch.EPollSelectorImpl)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:101)
    at A.main(A.java:5)
```

三、JNI 与 safepoint

首先有这样两个问题:

- 1) JNI 是否会影响 GC 进行?
- 2) GC 时 JNI 修改 Java Heap 怎么保证一致?

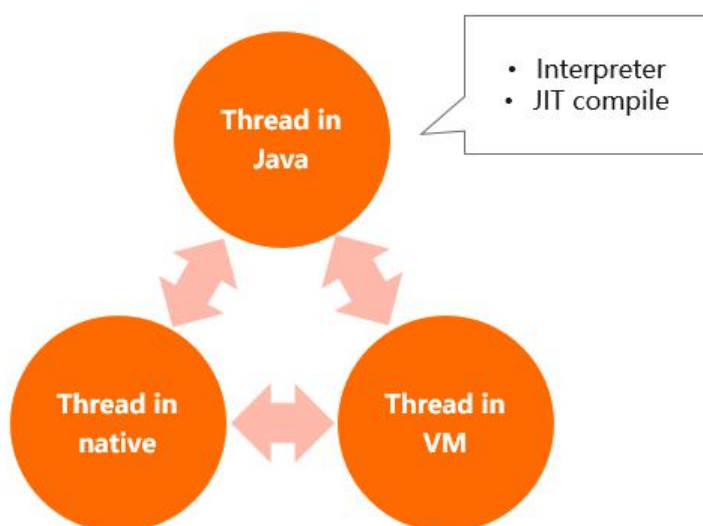
看到第二个问题的时候, 已经回答第一个问题, 假如 GC 是不能运行 JNI, 那也就没有一致性问题, 所以在 GC 时可以执行 JNI。

(一) JNI 与 Safepoint 的协作

首先要知道 Java 的信任状态，Java 最主要信任状态是“Thread in Java”状态，这个状态里面在执行一个解释器或者已经编译的方法，纯 Java 执行。这时候如果发生 Safepoint，会通过 Interpreter 机制把这个线程直接挂起，暂停下来，然后去 Safepoint 里面进行 GC 的各种操作。

在 Java 里面，调用 JNI 进入 Native，会切换到 Thread in native 状态，这里执行 Native 函数，在执行的时候跟 GC 可以并行执行，因为理论上要么执行，要么通过 JNI 和 JNI 交互，所有的跟 JNI 相关的数据结构都可以被管理。然后 Native 还可以去切换到 JVM 状态，这是非常关键的状态，这个状态不能发生 GC，不用关心。

JNI 与 Safepoint 交互，假如 JNI 执行时发生 Safepoint 能并行。JNI 执行的时候返回 Java，这时候会被阻塞，需要检查状态，卡在 Safepoint 状态，直到 Safepoint 结束，继续回到 Java。



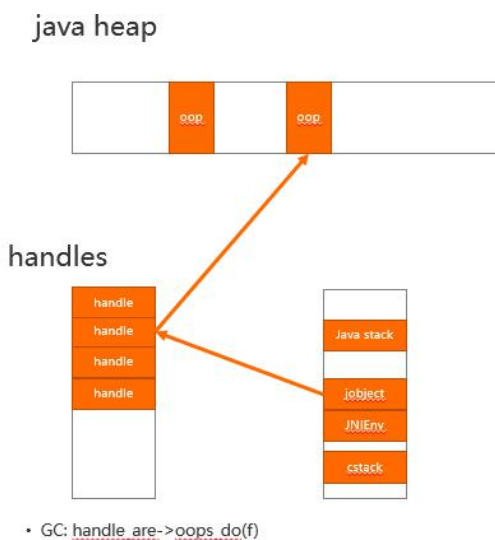
(二) JNI 与 GC

透过几个 JNI 管中窥豹，了解这个机制：

```
void * GetPrimitiveArrayCritical(JNIEnv *env, jarray array, jboolean *isCopy);  
void ReleasePrimitiveArrayCritical(JNIEnv *env, jarray array, void *carray, jint mode);
```

这个函数叫做 `GetPrimitiveArrayCritical`。Critical 作用是把一段内存返回给用户，用户可以直接编辑里面的数据，这时如果发生 GC 被移动，编辑肯定会导致 heap 乱掉，有 Critical 这段时间里锁住 heap，没法发生 GC。假如 critical 状态发生期间，基本上不会影响 GC，会等待，直 `ReleasePrimitiveArrayCritical` 发出，这是比较巧妙的互相协作。

下图所示的二级指针模型，还是前面 Java 调到 Native，参数通过 `jobject` 到 `handle` 保存使用，`jobject` 指向 `handle`，`handle` 指向 oop。



java heap 时候，假如 OOP 对象被移动 handle，同时会更新 handle 里面的地址。所以只要 C 程序都是通过 JNI 访问对象，每次对象被移动它都可以被感知，不会出现数据布局之后突然情况。

“GC: handle_are->oops_do(f)”

指有区域专门存放 handle，里面所有 handle 在 GC 里，都会进行一次指针修正，保证数据一致性。

四、JNI 与 Intrinsic

（一）高级主题: intrinsic

如下图所示，以非常常见 JNA “currentThread” 为例子，说明 Intrinsic 机制。Intrinsic 在看到 currentThread 的时候，不会去 JNI，而是通过形成更高效的版本。

这里 inline_native_currentThread 的时候，最终会调用 generate_curent_thread 工具。然后看里面的实现核心部分，创建 “ThreadLocalNode()”，代表当前 JavaThread 结构的指针，再通过 JavaThread 结构里的 threadObj_offset() 拿到它，通常是一个偏移量，拿到 Object 以后作为返回值返回。这里是一段 AI，真正生成代码时被翻译成非常简约的几条指令，直接返回。所以 “currentThread” 变得非常高效，这就是 Intrinsic 机制，主要为性能而生。

```
case vmIntrinsics::_loadFence:
case vmIntrinsics::_storeFence:
case vmIntrinsics::_fullFence:      return inline_unsafe_fence(intrinsic_id());

case vmIntrinsics::_currentThread:  return inline_native_currentThread();
case vmIntrinsics::_isInterrupted:  return inline_native_isInterrupted();
```

```
//-----generate_current_thread-----
Node* LibraryCallKit::generate_current_thread(Node* &tls_output) {
  ciKlass* thread_klass = env()->Thread_klass();
  const Type* thread_type = TypeOopPtr::make_from_klass(thread_klass)->cast_to_ptr_type(TypePtr::NotNull);
  Node* thread = _gvn.transform(new (C) ThreadLocalNode());
  Node* p = basic_plus_adr(top()/*!loop*/, thread, in_bytes(JavaThread::threadObj_offset()));
  Node* threadObj = make_load(NULL, p, thread_type, T_OBJECT, MemNode::unordered);
  tls_output = thread;
  return threadObj;
}
```

(二) Intrinsic 性能分析

对比一下 Intrinsic 与非 Intrinsic 性能，如下图所示，是用 jmh 写的 Benchmark，可以规避掉一些具体的预热不够，导致性能测试不准的问题，用它进行测试，也是官方推荐的版本。

Intrinsic 版本，下面测试叫“jni”，主要区别就是 Intrinsic 后面接了一个叫 isAlive 的调用。isAlive 本身状态调用看起来非常轻量，但因为他没有做 Intrinsic，所以最终会走 JNI。

```
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.infra.Blackhole;

public class JNI {

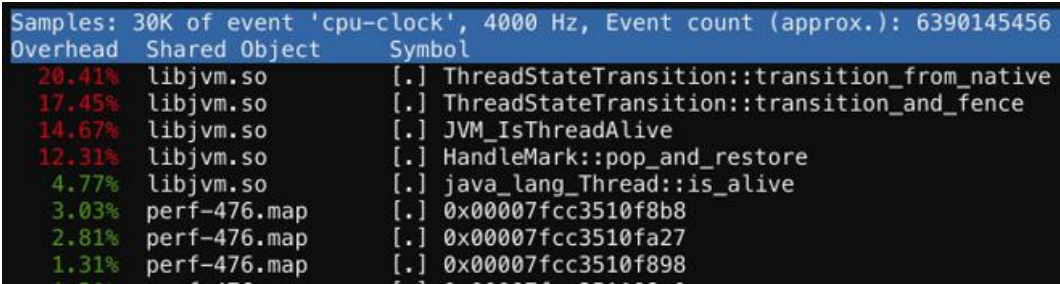
    @Benchmark
    public void intrinsic(Blackhole bh) throws Exception {
        bh.consume(Thread.currentThread());
    }

    @Benchmark
    public void jni(Blackhole bh) throws Exception {
        bh.consume(Thread.currentThread().isAlive());
    }
}
```

下图所示，对比普通 Intrinsic 与加上 JNI 的 Intrinsic 性能，普通 Intrinsic 的性能大概是 3 亿次每秒；加上 JNI 的 Intrinsic 版本的性能是 2000 万次每秒，差了十几倍，差距很大。

Benchmark	Mode	Cnt	Score	Error	Units
JNI.intrinsic	thrpt		311014387.385		ops/s
JNI.jni	thrpt		20046812.057		ops/s

进一步看性能问题，最重要的是 performing，performing 手段是 perform，public 第二段 JNI 版本，前面两个热点方法都是“ThreadStateTransition”现任状态转换。前面说到，假如 JNI 回到 Java 时候做 GC 肯定要停下来，所以这有个内存同步比较好资源，要等的时间比较长，所以这两个函数是最热的。



```
Samples: 30K of event 'cpu-clock', 4000 Hz, Event count (approx.): 6390145456
Overhead Shared Object Symbol
20.41% libjvm.so [.] ThreadStateTransition::transition_from_native
17.45% libjvm.so [.] ThreadStateTransition::transition_and_fence
14.67% libjvm.so [.] JVM_IsThreadAlive
12.31% libjvm.so [.] HandleMark::pop_and_restore
4.77% libjvm.so [.] java_lang_Thread::is_alive
3.03% perf-476.map [.] 0x00007fcc3510f8b8
2.81% perf-476.map [.] 0x00007fcc3510fa27
1.31% perf-476.map [.] 0x00007fcc3510f898
```

下面是“JVM_IsThreadAlive”实现。后面是“HandleMark::pop_and_restore”在调 JNI 时需要把 oop 包装成 handle, JNI 退出时, 需要消费 handle, restore 指有开销。再后面“java_lang_Thread::is_alive”占比 4.77% 非常小。

由此可以看出 Intrinsic 提供性能非常好的机制, 直接调用 JNI, 性能可能差一点, 但也可以接受。

(三) 案例分析: RocketMQ Intrinsic 导致应用卡顿

RocketMQ 是阿里巴巴开源的 MQ 产品, 使用非常广泛, 里面有个函数叫“warmMappedFile”, 指的是 RocketMQ 是通过 warmMapped 机制内存映射磁盘去做 IO, 在申请完一块磁盘映射的内存以后, 会去做预热。

这里有 for 循环“for (int i = 0, j = 0; i < this.fileSize”, 每隔一个 PACG_SIZE 去“byteBuffer.put(i, (byte) 0)”; 这样的话, 操作系统就会发生缺页, 把内存真正分配出来, 而不只是 EMV 数据结构。分配出来以后, 等到程序真正使用这块内存的时候, 就是纯内存 IO, 不太会触发这种缺页了, 可以变得更快, 目的是减少程序卡顿。

```
public void warmMappedFile(FlushDiskType type, int pages) {
    long beginTime = System.currentTimeMillis();
    ByteBuffer byteBuffer = this.mappedByteBuffer.slice();
    int flush = 0;
    long time = System.currentTimeMillis();
    for (int i = 0, j = 0; i < this.fileSize; i += MappedFile.OS_PAGE_SIZE, j++) {
        byteBuffer.put(i, (byte) 0);

        if (j % 1000 == 0) {
            log.info("j={}, costTime={}", j, System.currentTimeMillis() - time);
            time = System.currentTimeMillis();
            try {
                Thread.sleep(0);
            } catch (InterruptedException e) {
                log.error("Interrupted", e);
            }
        }
    }
}
```

但是后面加了 if 这一段，可以想到刚开始这个循环有问题，因为 `byteBuffer.put` 是 Intrinsic，最底层是 Intrinsic，方法返回的时候，没有方法调用。JVM 在方法返回以及循环末尾检查，是否有 Safepoints，来看是否要进入 GC，但是因为这是一个 Intrinsic，所以没有到检查点，同样这是一个 CountedLoop，也没法去进入检查点。因为 JVM 有个机制，如果这是一个 int 作为 index 去 Counted 次数的话，为了性能是不会去检查，因为它认为这是有限次的循环，所以不用检查次数。

这种机制循环里面非常简单，中间有可能因为操作系统原因带来卡顿，导致循环，基本上没法进入 GC，因为线程没有进入 Safepoints，整个界面都没法进入 GC，夯住很长时间，当时大家觉得很不可思议，但是通过一个很简单方法修好了，就是每隔 1000 字循环的时候，去调一个 “`Thread.sleep(0)`”。

刚刚提到，“`byteBuffer.put`” 没法出发，`Thread.sleep` 是个 JNI，返回的时候会检查 Safepoints，所以就可以让这个程序能够进入到 Safepoints，这个代码就不会影响 JVM 进入到 GC 了，代码目前还可以从开源软件上看到。

“-XX:+UseCountedLoopSafepoints”

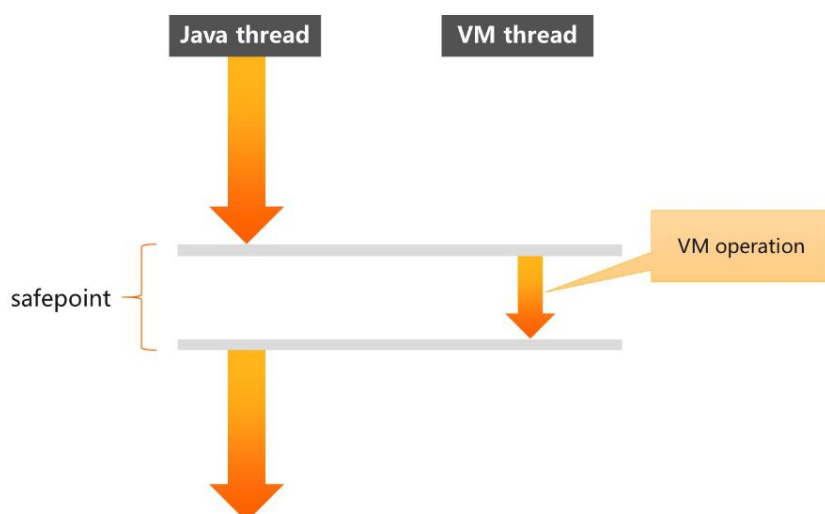
解决这个问题，还有另一种方式，通过一个选项叫“-XX:+UseCountedLoopSafepoints”，可以 JVM 自动在 CountedLoop 结尾检查这 Safepoints，当然这带来的副作用是，CountedLoop 末尾都会检查 Safepoints，这样就会影响整体性能。

Safepoint 机制

一、Hotspot safepoint 介绍

(一) 什么是 Safepoint

在 hotspot 内部，有时候它会把 Java 线程暂停掉，有时候又会把它叫做 Stop The World，在 hotspot 里可以做很多 vm 级别操作，如 GC、HeapDump/Stack trace、JVMTI、Check vmOperations.hpp，这里列了一个 vmOperations.hpp 这个头文件里面列出了绝大部分的这些 vm operation。下图演示，如正常的 java 的线程，运行的过程中，有一个 VMthread，有些特殊的条件，触发了一个 vm 的操作请求，这时候就会发起一个请求，要求 Java thread 都进入 safepoint，Java thread 收到请求以后，会自己暂停，等所有的 Java thread 停下来，整个 JVMTI 都安全了，可以做一些比较复杂的 vm 的操作，等操作做完了以后，就可以要求这些 Java 线程再重新恢复。



举例来说像 GC 会把在 Heap 中的 Java 对象移来移去，如果这时 Java 线程正在运行的时候，一边运行对象一边移动，Java 线程有可能就会访问到一个非法的地址，造成整个 JVM 的 crash，所以这时候需要进入 safepoint，把整个 Java 线程给暂停，Stop The World，会很影响性能。

（二）Safepoint 中还会做什么

从上述那些操作可知，在 hotspot 中会做很多事，平均下来，也许一秒钟之内会有两三次都会进入到一个 safepoint，所以 hotspot 会借用这个机会，用 safepoint 做一些常规的一些清理工作。

举例，如有些空的 monitor，他觉得可以回收了，就可以把它回收到一个 monitor 的 list，还有与 inline cache 相关的，会把它更新或者是清理掉。

还有些内部数据 stringtable 或者 symbol table 这类数据结构，在 safepoint 中觉得可以有必要做一些 rehash 的操作的话也会在这里做，这些都是一些很短的操作，一般来说并不是特别需要关心，这里主要提一下，在进入 safepoint 的时候，hotspot 除了做 vm operation 以外，还会做一些这种常规的动作。

（三）对 Safepoint 我们关注那些指标

safepoint 会把整个 jvm 的那些应用线程给暂停掉这里主要是关心的当 vm thread 发出请求的时候，Java 的实验者能够及时的响应 safepoint 的请求，能够马上自己给停下来，如果有一些线程它停下来了，另一些线程还在运行，这样的话其他的线程就会

等于是浪费时间在等待，所以说及时响应是它一个很重要的指标。

进入了 safepoint 后，vm operation 它本身操作，也希望能够在尽快短的时间内完成，完成了以后，还要能够快速的退出，这里一般没有问题，因为 safepoint 的退出都比较简单，一般来说不太会造成什么影响，前面三个点从进到做 vm operation 和退出，整个是影响了一次暂停的时间，如果你业务方比较关心这种延迟、响应时间这些指标的话，也许就要关注这几个性能。

有可能进一次 safepoint 很短，很快，但是 safepoint 的发生的时间频率又很高，这样的话，就会发现它总体暂停的时间就会很长，所以频率和总体时间也是一个需要关注的指标，如果对应用的吞吐量和性能比较关注的话，就要关注 safepoint 的总的暂停时间和它的那些频率，这里就是对 safepoint 有可能要关注一些性能。

(四) Safepoint 内部实现

safepoint 采用的是一种协作式的方式，就是当它发起了 safepoint 的请求后，那些 Java 线程来检测这个请求，然后再把自己给暂停，而不是通过强迫式，例如 VMthread 调用某一个 API 强行把一个 thread 给占进，强行暂停也许可以很快的把种线程给暂停住，但是这里会有很多不确定的状态在里面，安全性就很容易形成问题。

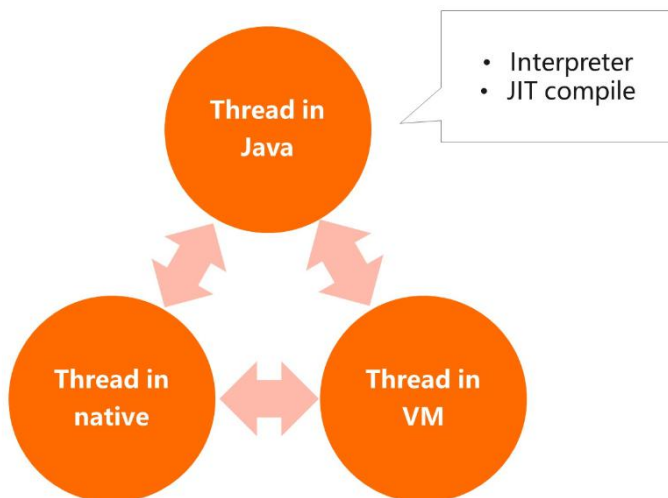
Hotspot 是所以就采用了这种协作式的方式，每个 Java 线程它能够及时的判断出来 safepoint 的请求，能够到一个他自认为可以安全的一个点上把自己给停下来。

与此同时，既然是协作式，就是说这些 Java 线程怎样能够确保它能够及时的响应，

因为有可能在做自己很复杂的业务逻辑，什么时候去检查 safepoint，做这么多的检查，会不会影响到 Java 本身的性能，这些都是需要综合考虑的一件事。

(五) Java thread 状态转换

在 Hotspot 里，对于这种 Java 的线程，其实主要有三个状态，在互相这样转换，第一个就是说是 Thread in Java，这个是说明这个线程现在执行的代码是 Java 的代码，如下图中标注，在执行 Java 代码中，在 hotspot 里它其实又分成两种模式，一种是解释器模式就 interpreter，第二种是 JIT，生成的那种 native 的 code，这两种模式它在这个里面处理也是不一样的。



另外两了状态 Thread in native 和 Thread in VM,他们其实执行的都是类似于像 c 和 c++的一些代码。

Thread in vm 的话主要是 hotspot 本身自己的那些代码；

Thread initiative 的话主要是一些 JMI, 如 Java code 有的时候需要调一些 GMI 的接口去访问, 去调用一些 c 的库和方法, 这时候它会进入的是 Thread in native 的状态。

以上就是他的三个状态, 在 safepoint 的时候, 要针对这三种不同的情况来做不同的处理。

(六) Thread in vm

Thread in vm 主要执行的像 hotspot 内部代码, 如 arraycopy, 如现在要执行一个 arraycopy 拷贝到一半的时候, GC 如果把 array 移到另一个位置, 肯定就出问题了, 拷贝的都是一个非法的数据, 做 arraycopy 的时候, 其实是会把自己 Java 线程的状态标志为 Thread in VM, 类似的像反射, 有的时候做一些 resolve 或 link, hotspot 里有很多的这种操作, 因为这些动作它往往是直接去操作 hot spot 内部的那些数据结构, 所以不会希望有一些 vm operation 类似像 GC 那些动作, 来做这些事情, 所以需要把线程状态标志为 Thread in VM, 在 Thread in VM 的状态下, 这个时候 VM thread 必须要等这个操作给做完以后才能做, 所以 hot spot 里对这些在 VM 状态的代码, 其实做得很小心, 它必须要保证这些这些事情能够很快的完成, 不会有这种长时间的阻塞或者这样的动作。

(七) Thread in native

Thread in native 其实是通过 JMI 接口去执行了 c 和 c++ 的一些 native 的 code, 在这种状态下, 其实在 JMI 中已经认为它进入了 safepoint, 即使已经在运行, 与前面提到的 stop the world 好像理解上有点不一样, 这时候这个线程其实还是可以一直在运行

的，因为如果这个代码是 native 的 code，其实 hotspot 是没法知道是什么状态的，而且也没法控制行为，有可能在做一个很长的 Loop，在那里不停的执行，所以这个时候如果要等的话，肯定会出问题 safepoint 就进不去了，但这时候认为已经是 safepoint 了，就可以做那些 vm operation，因为我的 Java 线上还在运行，当 native code 执行自己的东西的时候，是不会去碰到那些 Java 内部的那些 hip hop object 的那些东西，当想访问那些 object 的时候，需要通过那些 JMI 的接口，当调用接口的时候，这个时候 JVM 就会来检查这时候是不是正在做 safepoint，如果正在做 safepoint，就会把调用给阻塞，然后线程就会被停下来，等 vm operation 结束了以后再继续执行下去。

所以虽然在 Thread in native 状态你仍然在运行，但实际上不会造成造成危害，因为要访问那种 Java object 或者访问 hip 的时候，这里的 JMI 接口会挡住。

(八) Thread in java-interpreter

Thread in Java 的解释器模式，hotspot 中解释器其实是通过一个叫 dispatch table 的一个数据结构来实现的，Dispatch table 就是一个很大的 table，对于每个 bite code，它对应的就是一小段的执行代码，所以它执行的时候，是哪个 bite code 就执行 dispatch table 中的哪一段代码，然后在不停的跳转。

在解释器里面，在 hotspot 中，其实是维护了两套 dispatch table，一个就是 normal table，这就是刚刚说对每个 bite code 做解释执行的代码，另一个 safept table，除了做正常的解释执行之外，对每个 bite code 执行之前会加入一小段代码来检测，Jvm 是不是发起了 safepoint 的请求，如果发起 safepoint 的请求，就可以把自己给停下来。

通过这样一个方式来 safepoint 的 check 的，正常的话，Java 执行的都是 normal table 里的 bite code，如果 vm Thread 决定发起一次 safepoint 的请求的时候，hotspot 内部有个 active table 的指针，它会做一次切换，从 normal table 中切换到了 safept table。

一个 bite code 执行完，会去取下一次 bite code 的执行代码，因为这时候已经被切换到了 safept table，会执行 ssafept table 中对应的代码，然后就会检查 safepoint，然后再暂停。

所以基本上可以理解在解释器模式中，在每一次的 bite code 的最后都会做一次检查，但实际上它是通过一个 table 的表的一个切换来做的，正常运行的话，其实并没有做检查，所以它的性能并不会受影响。

(九) Thread in java-jit

Jet 最关注的是它的性能，在 jet 生成的 code 中，如何来检查 safepoint,在 hotspot 里，在它启动的时候，会先申请一个全局的 polling page 的这一个页，是一个 4k 大小的页,然后在 jit 生成的代码中，在某些特定的一个点，它会生成一两条指令，直接去访问页，就去读一下页里面这个内容是不是可读,特定点大概有两个，第一个是在 jit code 的返回的时候，在 return 的地方会去检查一次;另一个是循环，如果代码里面有循环，它会在循环的 loop 的 back edge 中，他\也会去检查一次，只在这两个点上去做检查，一方面是确保他\检查尽可能的少，另一方面要确保它的 jit 能够及时的响应 safepoint 的请求，本身只是读一下，并没有做任何的动作，这里如何把自己给停下来，就是 vm Thread 开始要触发 sfepoint 的时候，会做一个动作，会把全局的 pulling

page 把他的权限给改了，会用 `nprotect` 类似 API 把权限设成不可访问。

这样如果读取 `polling page` 的这条指令就会触发一次 `SIGSEGV` 的异常，但 `hotspot` 本身在 `signal handle` 里面，会对这种 `SIGSEGV` 做进行一些特殊处理，它会捕获住这种异常，会看触发异常的地址，是不是 `polling page`，然后如果是个 `polling page` 的，就知道是 `jit` 里面触发的 `safepoint`，所以这里并不是一个真正的异常，而是一次 `safepoint` 的请求。

后续的操作，会把 `Java` 线程给暂停，然后把自己的状态标志为已经进入了 `safepoint`。

如下图所示这段 `jit` 深层的代码，里面有一个 `Loop` 的 `polling`，又有一个 `return` 的 `polling`，可以看这两条 `test` 的指令，用红框标出来的，最上面的是一个 `polling` 是一个在 `back edge` 中他用来做 `polling` 的，其实只是做一次 `test`，把 `polling` 地址放到了 `20` 寄存器中，然后就去读一下 `test` 一下，后续对这个其实根本没有任何操作，`Test` 的结果对他来说没有任何作用，就是为了去读一次，能读这个代码就可以继续往下执行。

```
0x00007f52744e25cb: test    %eax, (%r10)      ; {poll}
0x00007f52744e25ce: cmp     %eax, %edi
0x00007f52744e25d0: jl      0x00007f52744e2588
0x00007f52744e25d2: mov     %rbx, %rax
0x00007f52744e25d5: add     $0x30, %rsp
0x00007f52744e25d9: pop     %rbp
0x00007f52744e25da: mov     0x108(%r15), %r10
0x00007f52744e25e1: test    %eax, (%r10)      ; {poll_return}
0x00007f52744e25e4: retq
```

下面的一条 `test`，旁边的标注是 `poll return`，紧接着下面就是一个 `return` 的指令，所以这一条指令就是在 `return` 之前，也会去做一次 `polling`，来判断下是不是有人在发起了 `safepoint` 的请求。

这就是在 jit code 中，大概会在这样的两个地方去做 polling, 第二个 test，如果看上一条，可能会看到 20 的地址其实是从二十五中读取了一个偏移量过来，25 在现在 X86 的 hotspot，主要是用来做一个 thread，所以它其实是从 thread 中去读了一个。

这里说明一下，牵涉到新的一个 jdk10 引入了一个技术，引入了一个叫 thread local handkerchief, 因为上述的 polling page 是 global 的，实际上把 global 的 page 把它作为这个地址记下来，然后每次 polling 的时候就直接去访问这个地址，这就是一个常量，根本没有任何动作不需要去到 thread 上去读。

(十) Global polling vs Thread Local handshake

在 jdk10 它这里引入了一个叫 thread the local 的 hand shake，这是一个新的协议品，主要的一个目的是要能够对一个特定的 thread 来触发 safepoint，前面讲过触发 safepoint 以后是会让所有的线程都停下来，但对某些操作，也许只是对一个线程来做动作的话，做一个把整个 Java 线程全部停下来的操作，是一个比较比较浪费的一个行为。

所以希望就是说能够用 thread local 的机制，只对一个特定的 thread 来把它给暂停，在 11 里面，都是用 thread local，这时候他取 polling page 的时候，都是从通过自己的 thread 里面去读一个 polling page 的地址。

实际上怎么做到 thread local，其实上述的 polling page 中，做了两个页，一个就是好的每次都能读，另一个是坏的，读就肯定会失败，good page 和 bad page 这样两个页，所以如果要对某个线程进行暂停的话，进入 safepoint 的话，其实就是把线程上的 page 的地址改写一下，改成坏页，这样 thread 就会触发到异常来进入 safepoint，

这里有一个开关,叫 User ThreadLocal Handshakes,它现在默认是打开的,基本上默认都会去走 thread local 的 safepoint,如果还是想用 global pulling,可以把它关掉。

实际上用到 thread local,用特定的线程来进入 safepoint 的这种 win 其实也没有多少,主要是现在的 cgc 大概会用到它。

Jit 因为比较关注性能,如果那种 loop 在一个循环里面,每个 loop 的回编中都要去做一个 polling,虽然只是一两条指令,但如果是在一个大循环里面,加起来的性能其实还是会有影响的,所以 hotspot 为了提高它的性能,可以把 counted loop 的 polling 给去掉,counted loop 就是一般看到的 for loop,可以认为是那种 for 的循环,因为这种循环中会有一个循环变量,循环变量有初始值,有它的边界,有它的布长,基本上都是固定的,在 hotspot 里面,就会认为这种循环叫 counted loop,在 counted loop 里面 hotspot 可以做一个优化,把这种 polling 的指令去掉,来提高它的性能,但这样会造成它的一个 trade off,如果你的 counted loop 比较大,这样进 safepoint 的时间就会就会被推迟了。

因为在整个循环中都不会去检查 polling,都不会去检查 safepoint,要等这个循环执行完一直到最后退出的时候,才会检查,造成的一个可能负面影响,就是说对进 safepoint 的时间它会延迟掉。

像 G1/ZGC 一些新的 GC,这些机器更关注的是说暂停的时间,为了要把暂停时间给减少,所以这些 GC 的时候,又会默认把 counted loop 中的 pulling 给生成出来。

总的开关，就是 UseCountedLoopSafepoints，打开就会生成，关掉就不生成这些 polling。

（十一）监控 safepoint

在日常的维护中，一般来说希望能知道 safepoint 究竟造成了一些行为是怎样的，这里提供的一些选项，像 JDK8，主要是提供了，能够打印 safepoint 的统计信息，能够知道它大概发生了多少次，总的暂停时间，可以计算一下它的平均时间等。

但在 JDK11 中，已经把这一个选项基本上已经是废弃了，因为在 JDK11 中，已经用了一个新的一套 Log 的机制，这套 Log 机制中对 safepoint 就可以用这个命令 `logsafepoint=debug` 打开这个开关，会打印出很多的跟 safepoint 的详细信息，如进入 safepoint 的花了多少时间，出来大概多少时间，总的时间是多少，这些详细的这些信息都能够在用 log 来记，所以在 JDK11 中，其实是比较推荐用这种方式来看 safepoint 的这些数据。

类加载器原理

一、类加载

(一) TraceClassLoading

TraceClassLoading 参数可以显示 JVM 从进程开始到运行结束的时候，所有 ClassLoad 的相关信息。在 JDK8 上，用 “-XX:+ TraceClassLoading” 就可以显示，在 JDK11 上的话，要加上 “-Xlog: class+load=info”。

下方是 JDK11 上打出来的一些日志，可以看到时间，类，还有类从哪个模块里来，信息非常详细。

```
[0.931s][info][class,load] java.lang.Object source: jrt:/java.base
[0.932s][info][class,load] java.io.Serializable source: jrt:/java.base
[0.932s][info][class,load] java.lang.Comparable source: jrt:/java.base
[0.932s][info][class,load] java.lang.CharSequence source: jrt:/java.base
[0.932s][info][class,load] java.lang.String source: jrt:/java.base
[0.932s][info][class,load] java.lang.reflect.AnnotatedElement source: jrt:/java.base
[0.932s][info][class,load] java.lang.reflect.GenericDeclaration source: jrt:/java.base
[0.932s][info][class,load] java.lang.reflect.Type source: jrt:/java.base
[0.932s][info][class,load] java.lang.Class source: jrt:/java.base
[0.932s][info][class,load] java.lang.Cloneable source: jrt:/java.base
[0.933s][info][class,load] java.lang.ClassLoader source: jrt:/java.base
[0.933s][info][class,load] java.lang.System source: jrt:/java.base
[0.933s][info][class,load] java.lang.Throwable source: jrt:/java.base
[0.933s][info][class,load] java.lang.Error source: jrt:/java.base
[0.933s][info][class,load] java.lang.ThreadDeath source: jrt:/java.base
[0.933s][info][class,load] java.lang.Exception source: jrt:/java.base
[0.933s][info][class,load] java.lang.RuntimeException source: jrt:/java.base
[0.933s][info][class,load] java.lang.SecurityManager source: jrt:/java.base
[0.933s][info][class,load] java.security.ProtectionDomain source: jrt:/java.base
[0.933s][info][class,load] java.security.AccessControlContext source: jrt:/java.base
[0.934s][info][class,load] java.security.SecureClassLoader source: jrt:/java.base
[0.934s][info][class,load] java.lang.ReflectiveOperationException source: jrt:/java.base
[0.934s][info][class,load] java.lang.ClassNotFoundException source: jrt:/java.base
[0.934s][info][class,load] java.lang.LinkageError source: jrt:/java.base
[0.934s][info][class,load] java.lang.NoClassDefFoundError source: jrt:/java.base
[0.934s][info][class,load] java.lang.ClassCastException source: jrt:/java.base
[0.934s][info][class,load] java.lang.ArrayStoreException source: jrt:/java.base
[0.934s][info][class,load] java.lang.VirtualMachineError source: jrt:/java.base
[0.934s][info][class,load] java.lang.OutOfMemoryError source: jrt:/java.base
```

(二) 类加载与虚拟机

关于类加载部分，首先用户有 Java 文件，然后 Java 文件用 Java c 去编译就可以得到.class 文件，接着虚拟机会加载.class 文件变成虚拟机的元数据。比如在 c++里边会变成 Klass *，Method *，ConstantPool * 等，这些都是 Java 虚拟机里元数据的描述。

比如一个 Class 会变成一个 Klass*的结构体，这个 Class 里面所有方法会变成虚拟机里面 Method*的结构体，然后常量池会被包装成一个 ConstantPool*，这些在虚拟机里都有相关描述。

(三) ClassFile

```
#865 = Utf8          java/lang/Long
#866 = Utf8          java/lang/Float
#867 = Utf8          java/lang/Double
#868 = Utf8          (I)Ljava/lang/StringBuilder;
#869 = Utf8          fill
#870 = Utf8          inflate
#871 = Utf8          ([BI[BII)V
#872 = Utf8          compress
#873 = Utf8          ([CII)[B
#874 = Utf8          isBmpCodePoint
#875 = Utf8          isSupplementaryCodePoint
#876 = Utf8          toBytesSupplementary
#877 = Utf8          (I)Ljava/lang/Integer;
{
  private final byte[] value;
  descriptor: [B
  flags: (0x0012) ACC_PRIVATE, ACC_FINAL
  RuntimeVisibleAnnotations:
    0: #247()
      jdk.internal.vm.annotation.Stable
```

上图为 ClassFile 的结构，它的反汇编是 Java.lang.string。

如果用户想构造一个 String，就必须传给它一个字符串的自变量，自变量会被传到 Value 的数组里。可以看到，在 JDK11 当中 Value 是用 Stable Annotation 修饰的。


```

124 public final class String
125     implements java.io.Serializable, Comparable<String>, CharSequence {
126
127     /**
128      * The value is used for character storage.
129      *
130      * @implNote This field is trusted by the VM, and is a subject to
131      * constant folding if String instance is constant. Overwriting this
132      * field after construction will cause problems.
133      *
134      * Additionally, it is marked with (@Link Stable) to trust the contents
135      * of the array. No other facility in JGM provides this functionality (yet).
136      * (@Link Stable) is safe here, because value is never null.
137      */
138     @Stable
139     private final byte[] value;

```

和上图对比，可以发现 Private Final 以及 Byte 的数组全都被很好地描述在 Java p 反汇编的 Class 文件中，Stable annotation 被描述在 ClassFile 里。

我们来看一个例子。

```

277 @ private static void rangeCheck(char[] value, int offset, int count) {
278     checkBoundsOffCount(offset, count, value.length);
279     return null;
280 }

```

rangeCheck 是 String 里边的一个 Static 方法，这个方法有三个参数 value、offset 和 count，它内部会调用一个 static 的方法，并且返回 null。

```

private static java.lang.Void rangeCheck(char[], int, int);
descriptor: ([CII)Ljava/lang/Void;
flags: (0x000a) ACC_PRIVATE, ACC_STATIC
Code:
    stack=3, locals=3, args_size=3
    0: iload_1
    1: iload_2
    2: aload_0
    3: arraylength
    4: invokestatic #8          // Method checkBoundsOffCount:([III)V
    7: aconst_null
    8: areturn
LineNumberTable:
    line 280: 0
    line 281: 7
LocalVariableTable:
    Start Length Slot Name Signature
    0      9      0 value  [C
    0      9      1 offset I
    0      9      2 count I

```

对照上方的 Java p 反汇编的 class 文件，反汇编的文件分为三个部分，第一个部分是 Code，第二个部分是 LineNumberTable，以及 LocalVariableTable。

Code 当中 iload_1，iload_2 以及 aload_10 都是字节码，可以看到 LineNumberTable 里的第 280 行对应的 0，这个 0 是上面 Code 的第 0 行，也就是 iload_1。下面的 line 281 行的 7 对应的是 aconst_null 字节码。

LocalVariableTable 的 Start、Length 对应的都是字节码的位置，后面还有名字等信息。

例如 value 这个变量是从第 0 号字节码，它的生命周期一直从 0 号到第 9 位字节码，第 9 位是左开右闭区间，因此不包括第 9 号字节码。可以看到，所有的信息都会被完整保存在 ClassFile 里。

```
@II
@IH
public class Annotations <@IG T> extends @IG Enum implements @IG Interface{

    @IM
    Annotations() {}

    @IA
    @IB(name = "ha")
    @IC(name = "ok", a = @IA, b = { @IB(name = "a"), @IB(name = "b"), @IB{}})
    int iz;

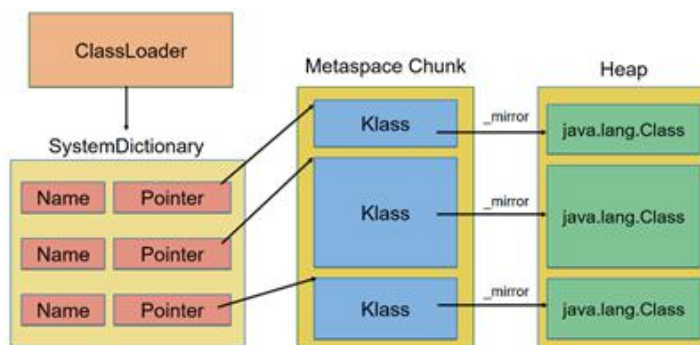
    @IA
    @IB(name = "ha")
    @IC(name = "ok", a = @IA, b = { @IB(name = "a"), @IB(name = "b"), @IB{}})
    @SuppressWarnings("hahaha")
    <@IF @IG @IJ T extends @IG ArrayList<@IG T>>
    @IF @IG T check(@ID @IE @IF @IG T in) throws @IG RuntimeException {
        try{
            @IA @IK String i = "haha";
            System.out.println(i instanceof @IF String);
            int j = 0;
            System.out.println(i[@IF char]);
            String @IG [] ss = new String[3];
            @IG String[] sss = new String[3];
            @IG ArrayList<@IG String> a = new ArrayList<>();
            throw new @IG ClassNotFoundException();
        } catch (@IG RuntimeException | @IG ClassNotFoundException g){
        }
    }
    return iz;
}
}
```

可以看到，上图所示的 Annotations 类上面有无数的注解，例如 IA、IB、IC，它们都是 Annotations 的定义，Annotations 可以插在程序的各个地方，这张图只是为了一个直观表示，然后来看一下 Annotations 是怎么样被 Incode 进 ClassFile 里面的，可以直观对比下图的变量。

```

RuntimeInvisibleAnnotations:
  0: #20(#21=s#22)
    IB(
      name="ha"
    )
  1: #23(#21=s#24,#25=@#18(),#26=@#20(#21=s#25),@#20(#21=s#26),@#20())
    IC(
      name="ok"
      a=@IA
      b=@IB(
        name="a"
      ),@IB(
        name="b"
      ),@IB()
    )
RuntimeVisibleTypeAnnotations:
  0: #37(): METHOD_TYPE_PARAMETER, param_index=0
    IG
  1: #37(): METHOD_TYPE_PARAMETER_BOUND, param_index=0, bound_index=0
    IG
  2: #37(): METHOD_TYPE_PARAMETER_BOUND, param_index=0, bound_index=0, location=[TYPE_ARGUMENT(0)]
    IG
  3: #37(): THROWS, type_index=0
    IG
  4: #37(): METHOD_RETURN
    IG
  5: #37(): METHOD_FORMAL_PARAMETER, param_index=0
    IG
RuntimeInvisibleTypeAnnotations:
  0: #39(): METHOD_TYPE_PARAMETER, param_index=0
    IF
  1: #43(): METHOD_TYPE_PARAMETER, param_index=0
    IF
  2: #39(): METHOD_RETURN
    IF
  3: #39(): METHOD_FORMAL_PARAMETER, param_index=0
    IF
RuntimeVisibleParameterAnnotations:
  parameter 0:
    0: #45()
    IE
RuntimeInvisibleParameterAnnotations:
  parameter 0:
    0: #47()
    ID
  
```

(四) ClassLoader 结构



Class 这些元数据在 JVM 当中是如何被表示的？

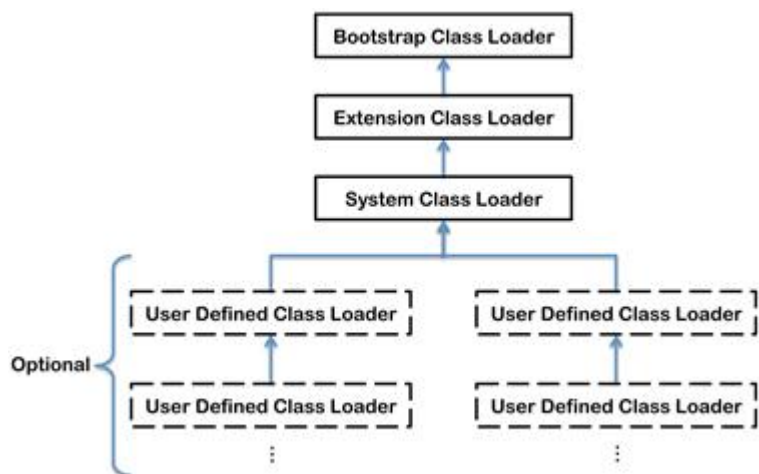
假设有一个 ClassLoader 正在 Loader 一些类，然后把它们 Load 进虚拟机当中。JVM 当中有一个结构体叫做 SystemDictionary，它是一个 Meta，会把 Class 的类名 Meta 到 Class 的 Pointer 当中，然后 Pointer 指向的就是 Metaspace 当中真正的 Class 结构描述。

Class 当中有一些 Mirror 的字段，这些 Mirror 指向 java.lang.Class。Mirror 和上层的.class 是一样的，是一个反射接口的作用。

可以看到，ClassLoader 会索引到 SystemDictionary，然后索引到 Metaspace Chunk，接着索引到 Heap，这几个可以互相引用。

图中 Metaspace Chunk 的 Klass 以及 Heap 里的 java.lang.Class 图形大小是不同的。因为用户自己写的 Class 有可能会继承自不同的父类以及不同的接口，它有可能实现了若干个父类和接口，实现接口和父类的数量有所不同，Class 里的东西也是不尽相同，因此元数据的大小也是不一样的。

（五）双亲委派机制



Java 的 `ClassLoader` 有双亲委派机制，先使用双亲类加载器进行加载，当 Parent 加载失败的时候，再自己加载。

`Bootstrap Class Loader`、`Extension Class Loader` 和 `System Class Loader`（即 `APP Class Loader`）这三个 `Class Loader` 是父子的关系。如果先从 `APP Class Loader` 加载用户的命令 `Class`，会先去 `Extension Class Loader` 加载，然后去 `Bootstrap Class Loader` 加载，如果它们都没有加载到，最后才会轮到 `System Class Loader` 加载。

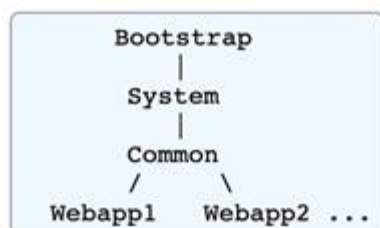
所有 `User Defined Class Loader` 的 Parent 基本都是 `System Class Loader`，用户可以选择自己是否要写一个新的 `Class Loader`。

`LoadClass` 类是 `ClassLoader` 内部一个非常通用性的类，如果要重写一个 `ClassLoader` 的话，会选择重写里面的 `findLoadedClass` 这个方法，而不会选择 `LoadClass`。

```
protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException {
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            try {
                if (parent != null) {
                    c = parent.loadClass(name, resolve);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }
            if (c == null) {
                c = findClass(name);
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
```

如上图所示，首先是一个 synchronized，加上 get ClassLoadingLock 的同步锁。它下面会先调用一个 findLoadedClass，这个函数会去 SystemDictionary 里去寻找相应的类。如果它没有，那么就会到 Parent 中 loadClass，如果 Parent 里也没有，就会到 findClass 的方法。

(六) 破坏双亲委派机制



Ø Tomcat ClassLoader 为例，它会经过以下过程：

- 1) 在本地 ResourceEntry 当中查找
- 2) 调用 ClassLoader.findLoadedClass()
- 3) 默认情况下调用 AppClassLoader.loadClass()
- 4) 用自身加载
- 5) 依旧没有加载出来的情况，最后才委派给 Parent

Ø 意义：可以实现一个 VM 进程下加载不同版本的 jar 包

(七) ParallelCapable

从 JDK1.7 开始，ClassLoader 引入了一个叫 ParallelCapable 的特性。

之前的 JDK 当一个 ClassLoader 在 LoadClass 的时候，它会锁 ClassLoader，锁的粒度是整个 ClassLoader。在 1.7 引入了 ParallelCapable 特性之后，锁的粒度变成了 Class，大幅提高 ClassLoader 的性能。

先 ClassLoader 在 loadClass 时同步整个 loader 对象，现在把锁变成了单个类名对应的 Placeholder。如果要 Load 一个 Class，检查类名就可以找到相应的 Placeholder。

下面我们来看一下它到底是怎么实现的。

```
protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException {
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            try {
                if (parent != null) {
                    c = parent.loadClass(name, resolve);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }
            if (c == null) {
                c = findClass(name);
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
```

如上图所示，第一行的关键字 `synchronized` 锁住了 `getClassLoadingLock`。这个方法会从非权限命名所对应的 `Object` 的 `Map` 里边搜索到它对应的 `Placeholder`，也就是占位符，它只要锁住了占位符，后面的过程就全是进程安全了。

下面我们来看一下虚拟机里面的实现。

```
bool DoObjectLock = true;
if (is_parallelCapable(class_loader)) {
    DoObjectLock = false;
}

// Class is not in SystemDictionary so we have to do loading.
// Make sure we are synchronized on the class loader before we proceed
Handle lockObject = compute_loader_lock_object(class_loader, THREAD);
check_loader_lock_contention(lockObject, THREAD);
ObjectLocker ol(lockObject, THREAD, DoObjectLock);

// ... real class loading ...
```

`DoObject` 变量决定了当前的 `ClassLoader` 是否要锁住整个 `ClassLoader` 来加载一个类。如果是 `true`，就会去锁住整个 `ClassLoader`。如果它是 `false` 的话，它就会像刚才一

样做 synchronized 操作，synchronized 锁住的是它加载的类对应的名字所对应的 Placeholder。这样的话它就把 C++ 层锁住整个 ClassLoader 的代价，转移到了 Java 层，去锁住 Class。

二、链接

Ø 链接的过程如下：

- 1) 先递归地对所有父类和接口进行链接操作；
- 2) verify 当前类；
- 3) rewrite 当前类：

n 比如会把 java.lang.Object.<init> 构造函数的 `_return` 字节码重写为 `_return_register_finalizer` 字节码；

```
public java.lang.Object();
descriptor: ()V
flags: (0x0001) ACC_PUBLIC
Code:
    stack=0, locals=1, args_size=1
    0: return
LineNumberTable:
    line 50: 0
LocalVariableTable:
    Start Length Slot Name Signature
    0      1      0  this  Ljava/lang/Object;
RuntimeVisibleAnnotations:
    0: #27()
    jdk.internal.HotSpotIntrinsicCandidate
```

n 比如 `_lookupswitch` 这种不连续的 `switch`，跳转分支数在 `BinarySwitchThreshold` (default 5) 以下会被重写成 `_fast_linearswitch` 字节码，否则会变成 `_fast_binaryswitch` 字节码；

```
public static void main(String[] args) {  
    switch (args.length) {  
        case 1:  
        case 2:  
        case 3:  
        case 100:  
        case 101:  
    }  
}
```

```
public static void main(java.lang.String[]);  
descriptor: ([Ljava/lang/String;)V  
flags: (0x0009) ACC_PUBLIC, ACC_STATIC  
Code:  
    stack=1, locals=1, args_size=1  
    0: aload_0  
    1: arraylength  
    2: lookupswitch { // 5  
        1: 52  
        2: 52  
        3: 52  
        100: 52  
        101: 52  
        default: 52  
    }  
    52: return
```

n 比如 `_aload_0 + _getfield (integer)` 的组合最终会被 rewrite 成 `_fast_iaccess_0` 字节码

4) 对类内部的所有方法进行链接操作，使其生效（设置方法执行的入口为解释器的入口）。

三、初始化

（一）初始化操作

在虚拟机规范当中，我们可以看到这样的描述：

- 1) 在 `_new/_getstatic/_putstatic/_invokestatic` 字节码时/反射/lambda 解析发现 callee 是一个 static 函数时触发；
- 2) 调用 class 的 `<clinit>` 方法；
- 3) 实例化。

```
static {};  
descriptor: ()V  
flags: (0x0008) ACC_STATIC  
Code:  
  stack=0, locals=0, args_size=0  
    0: invokestatic #16          // Method registerNatives:()V  
    3: return  
LineNumberTable:  
  line 43: 0  
  line 44: 3
```

我们写 Java 程序的时候用的 Static 变量，在虚拟机内部会转化成一个叫 `<clinit>` 的方法，然后实例化。如果用反射去 New 一个 Object，或者是走 New 字节码的时候，都会进行初始化的操作。

上图是一个 `<clinit>` 的方法，截取的是 `java.lang.Object` 的 Static 块，它只有一条的代码。

（二）编写自己的 ClassLoader

Ø 方法：

- 1) 按照 `ClassLoader.loadClass()` 的模板来重写（不推荐）；
- 2) 仅重写 `findClass()` 方法，拿到并解析.class 文件为一个 `byte[]` 数组，并调用 `defineClass()`方法进入 VM。

（三）Class Unloading

Ø JDK8 与 JDK11 中都有 `-XX:+ClassUnloading` (default true)

Ø Class Unloading 发生在当一个类不被任何引用所引用时，就可以被 unload 掉

- 1) 一个类被加载的时候，会产生 `ClassLoader -> Class` 的引用，因此 `ClassLoader` 自身需要先不被任何引用所引用
- 2) 其他 GC roots 无对此类的引用，比如栈帧等

（四）向 JDK11 迁移

Ø JDK8 和 JDK11 中 JDK library 中的 `ClassLoader` 有所不同

- 1) `ExtClassLoader` 更名为了 `PlatformClassLoader`；
- 2) `PlatformClassLoader` 和 `AppClassLoader` 不再继承自 `URLClassLoader`；

- 3) 如果指定了 `-Djava.ext.dirs` 这个变量，需要用 `-classpath` 来加以替代；
- 4) 如果指定了 `-Djava.endorsed.dirs` 来覆盖 JDK 内部的 API，需要删掉参数。

(五) AppCDS (APPLication Class Data Sharing)

Ø AppCDS 是 OpenJDK 做的一个特性，它有以下特点：

- 1) 用程序加载的 classes 产生 *.jsa 文件 (shared archive)，给应用的启动进行加速；
- 2) JDK 1.5 时为 CDS，只能用 dump BootstrapClassLoader 加载的类；
- 3) JDK10 后变为 AppCDS，可以用于 AppClassLoader 和 custom ClassLoaders。

Ø AppCDS 本质是动态分析流程，使用步骤如下：

- 1) 第一次： `java -Xshare:off -XX:DumpLoadedClassList=list.log <app>`
- 2) 第二次： `java -Xshare:dump -XX:SharedClassListFile=list.log XX:SharedArchiveFile=dump.jsa <app>`
- 3) 第三次： `java -Xshare:on -XX:SharedArchiveFile=dump.jsa <app>`

JDK 在 build 的时候，会使用 Java 加上 AppCDS 的参数自动产生一份 .jsa 文件来加速启动，放在 `${JAVA_HOME}/lib/server` 下，会什么参数都不加，裸跑一个 .jsa 文件，产生的文件叫 `classes.jsa`，用户搜自己 JDK11 的目录都可以搜到。

Dragonwell 特性: 多租户

一、JAVA 语言的特点

JAVA 语言的特点如下图可以看到企业级性能好,稳定,生态丰富, JavaEE 的标准非常好,后面还有跨平台的特性,按层次来看 Java 平台,Java 硬件上是 arm 或者 x86 或者 PowerPC 等等不同 CPU, 在此之上操作系统是如 windows、Linux、Mac OS,再往上是 JAVA platform,是一个平台,因为只要根据 Java 平台来写应用,可以完全不管操作系统以及硬件的一些具体实现细节,跨平台性是 Java 的一个主要卖点,在此之上 Java 定义了 Java1g2e 的标准,然后 Tom Kate、APACHE 等各种组件其实都是从 JAVA EE 的标准里面衍生出来的,帮助 Java 很好的成长,按照 Java 一开始设计的人目标去成长。

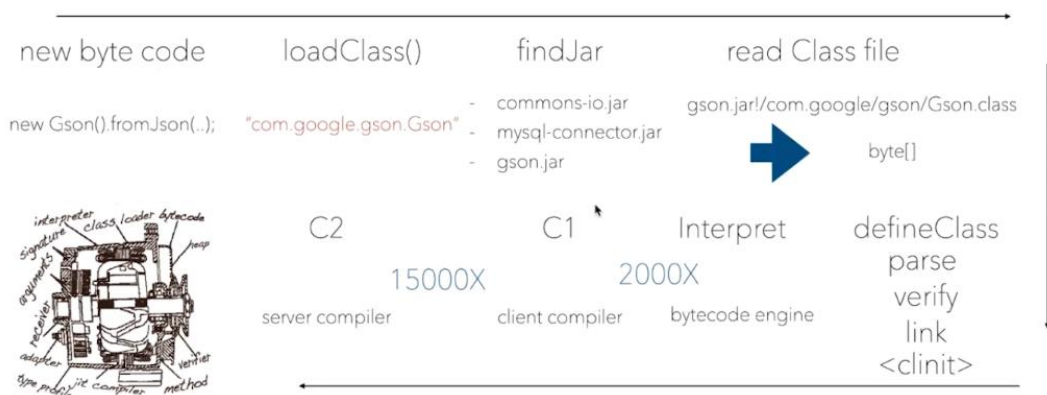
生态丰富的特点,因为 Java 上有 Tom Kate、APACHE、spring 等各种生态,开箱即用,开发应用非常简单;企业级性能好,Java 它适合于运行一种大型、长期运行的程序,稳定性非常好,如运行一个 Linux server 加 JVM 的一个组合,可能几年都不用重启,可以跑得非常好;

二、一次编译到处执行

一次编译到处执行是 Java 的一个很大的卖点,通过一段代码来了解在 Java 内是如何被加载和执行的,如在应用代码里面去调用 `new Gson(). from Json(..)`;然后有一个 `new byte code` 会触发 `loadClass ()` 机制,还要去找 `com.google.gson.Gson` 用 `loadClass` 方法,去找 Jar,因为应用的 class 下有很多 Jar 包,如 `commons-io.jar`、

mysql-connector.jar、gson.jar，找到 gson.jar 里面有 com.google.gson/Gson.class 文件，然后会把 Class 给读出来，读成一个 byte 的数组，调用一个 define class JVM 的接口，define class 会进行 parse、verify、link，调出<clinit>,最终达到一个可以让 Jvm 识别的 byte code，Jvm 解释器会去执行 byte code 到 2000 次以后，会运行一个 client compiler 让代码编译到 c1 级别，c1 级别其实已经在 native 执行了，同时会收集一些 provide 信息，帮助编译到更高的优化级别 c2，然后到 15,000 次以后会进入到最快的 c2 级别，interpret 和 c2 之间可能大概有 50 倍的差距，所以 Java1 开始是很慢的，但只要跑稳后是非常快的。

代码装载的开销很高

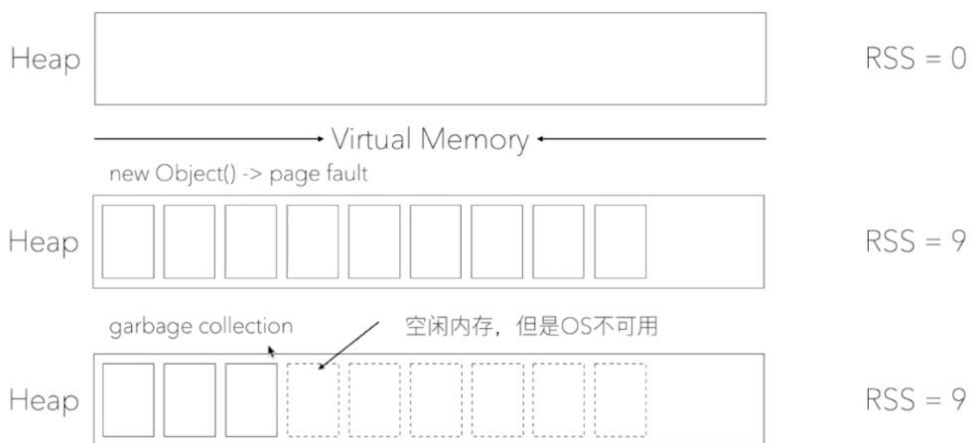


通过上述，一段代码想要被执行，生命周期是非常长的，优点是这种跨平台性，可以收集的信息越跑越快，缺点就是 Java 代码装载的开销非常大。

三、内存管理

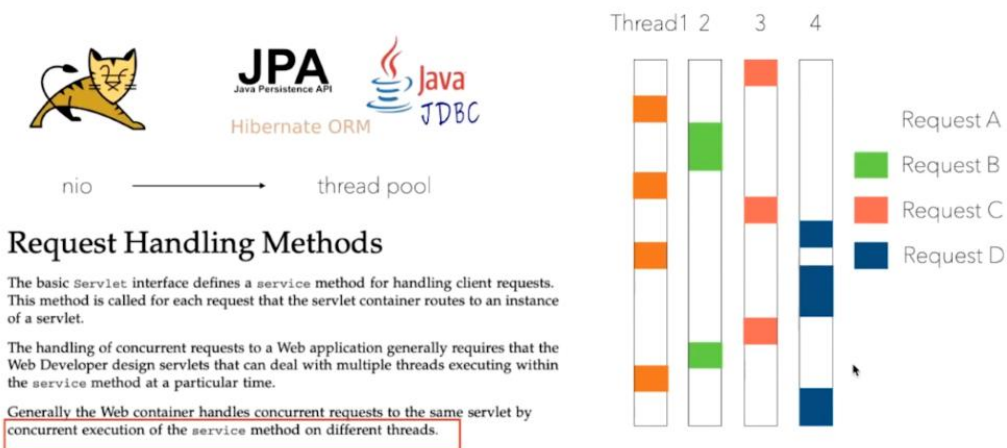
Java 的垃圾回收管理 GC， heap 就是 Java 里面实际做管理的内存，是虚拟内存，

一开始如设 4Gheap, 会想到操作系统, 申请 4G 的 heap, 这些配置其实都没有被申请出来, 是按需分配的, 这是操作系统的推荐机制, 随着各种 new Object, 页面就会被分配出来, 如下图所示分为 9 个页面, 就是 RSS=9, 中间如果发生 garbage collection, 即便会压缩内存的, 如这里面有 6 个页面是空闲的, 他就把三个排到一起, 然后剩余的内存是空闲的, 但是操作系统认为依然占着这些内存, 然后它是不可用的, 所以 RSS 依然很高, 因为内存很有可能随着后面的分配马上会被使用, Java 也是积攒到一定程度来释放的, 所以很有可能占据很大的面积。



四、线程模型

用 Java 去写 server 端应用的时有很多框架, 如 Tom Kate 后面可能会有 JPA、Hibernate ORM、JDBC 等组件是通过一个 NIO 接收请求, 接收到请求以后是交到一个 thread pool, thread pool 多线程可能并不是一种特别高效的处理并发的模型, 下图可以看到一个 g2e 的规范, 在 thread 的规范里面, 一个外部容器怎样去处理并发的请求是通过在多线程里面去并行的调用 service 函数来达到的, 标准导致这个事件必须通过线程池来提供并发能力。



线程多会导致的问题，上图是一张实际的执行图，每一个竖条代表一个线程，每一个色块代表一个请求，如在只有一个核的机器上去启动 4 个线程，其实操作系统提供能力，会觉得请求是并发并行来执行的，实际上都是交替的执行分时复用的，看起来是并发的，实际是交替来执行，这中间就有一个切换开销是比较大的，是多线程的一个弊端。

五、云原生 vs Java

总结了 Java 的优势，云原生的应用应该是微服务的，但 Java 是一个企业级的模式都非常庞大；云原生是低内存开销，这样可以创造很多微服务，把他们更快的合并更好的和部署，但 Java 是 GC 管理是大块内存的；云原生要求应用是快速交付，Java 应用启动慢，需要编译预热，中间是有一个抵触的，

六、Dragonwell

所有的市面上的 JDK 产品大多都基于 Open JDK，Open JDK 加上 oracle 的一些

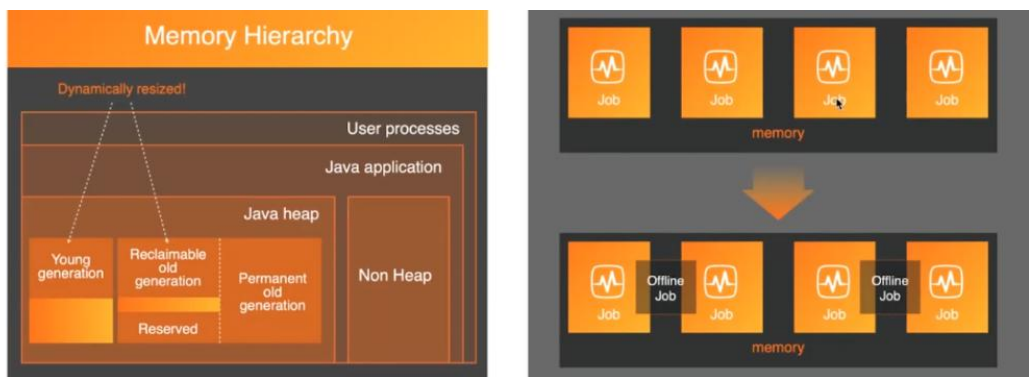
商业特性，然后形成的是 oracle JDK，其它还有一些三方厂商，如说亚马逊等都通过 Open JDK 进行了简单的扩展，形成自己的 JDK，阿里云也是通过 Open JDK 作为上游扩展，然后加上阿里云的一些原生特性，最终形成了阿里巴巴 Dragonwell 这个产品。



七、Dragonwell: Elastic Heap

阿里云自己的一些特性也会回归社区，这里介绍 Dragonwell 的几个主要特性，第一个是弹性堆，就是可以把前面看到的一些那问题给解决掉，把内存还给操作系统；往下是进程级别，然后 Java 应用级别，然后里面有 JavaHeap，Java heap 里面之间内存默认都是固定的，但是通过 Elastic Heap 机制，可能用不到这么多内存就可以把应用跑得很好，会动态的去减少 Heap 大小，把内存实际还给操作系统。

下图右半部为特性的使用场景，可能有一些在线的 job，比如 4 个应用，平时用户量很大，需要大量内存去应付很多请求，所以内存是不共享的，但是到晚上可能没有很多用户在访问在线应用了，就可以动态的把内存给释放出来，然后提供给其他一些同等级以上的离线服务，离线服务和在线服务可以捆绑，或者是复用内存，能达到一个很好的资源复用的目的。



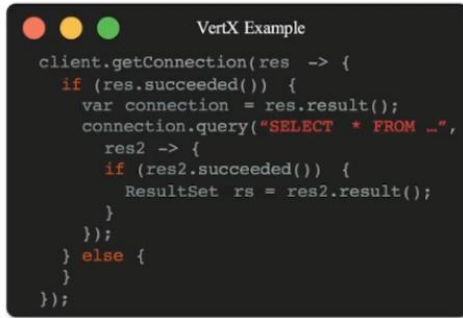
八、Dragonwell: wisp

还有一个非常有用的特性叫做 wisp 协程, 这个特性可以把 Java 的线程映射为协程, 去提高 Java 的并发处理能力, 因为目前都是微服务的状态, 微服务把应用拆掉之后, 自然不同服务就要通信, 通信的就会很多 wisp 提高这些 io 的效率。

现在 Java 有很多异步编程框架, 如 VERT.X 这个词的含义就是节点, 跟 node 的 js 在图算法里面其实是可以互相代替使用的, VERT.X 和 node.js 就是想制造一种 Java 里的 node, 希望用 node 的这种异步编程模型得带给 Java。

下图是 Verte.x 编写一个数据库的访问应用, 用他官方的一个 API: `client.getConnection` 到这里不是直接的返回的 connection, 而是写一个 call back, 因为这是一个组词函数, 异步的他就要写 call back, 然后 call back 里面去判断请求是否成功, 成功的话可以拿到 result 里面一个 connection, 然后 Connection 去查一个 SQL 语句, 比如 “select*FROM...”, 这里同样要写一个 call back result to, 而不是说直接拿到 connection, 一层层的回调, 如果我们要继续 result 再去经营什么, 比如说把它放到缓存里面, 那里面又会进行一层嵌套的回调。

异步编程



```
client.getConnection(res -> {
  if (res.succeeded()) {
    var connection = res.result();
    connection.query("SELECT * FROM ..",
      res2 -> {
        if (res2.succeeded()) {
          ResultSet rs = res2.result();
        }
      });
  } else {
  }
});
```



协程

ES7

```
suspend fun Client.aGetConnection(): .. =
  suspendCoroutine { cont ->
    getConnection( conn ->
      { cont.resume(conn) })
  }

conn = client.aGetConnection();
rs = conn.aQuery("SELECT * FROM ..");
```

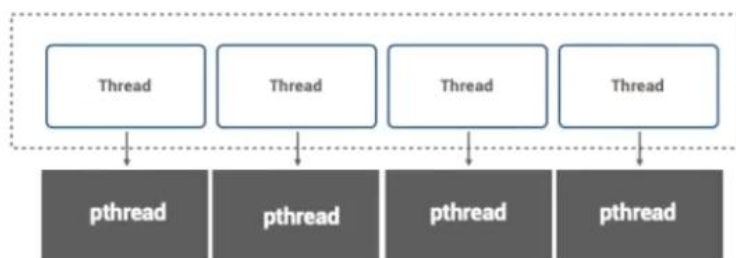
这样代码的控制就被反转了，并且异常也看不到，如在某一层发生一个异常，因为都是从回调直接执行的，是看不到从哪调进来的，这是异步编程带给我们一个问题，当然异步编程是有解决方法的。

C sharp 和 ES7 所提供的新特性，主要是提供了协程机制，Kotlim 程序作为例子，来解释这种机制如何帮助去改变义务变成模型,在 Kotlim 里面，如果想让一段代码可以被协程切换的话，可以用 suspend 标记这段代码，然后可以对一个异步函数进行封装，如 client.getconnection，把它封装成 client.aGet connection，实际做调用了一个非方法叫 suspendCoroutine 做的是调用这一段代码，并且把当前协程切走,会马上调用 getConnection，并且得到一个 continuation 的回调,在 get connection 的 call back, 如 get connection 实际完成的时候，会恢复协程马上挂起，然后再完成恢复,这样就可以让代码临时挂起,过一段时间再回来执行,只有协程可以带给的优势,整个执行站都是保留完整的，通过这种封装可以这样去写代码;Conn=clientaGetconnection，rs=connection=，aQuery 同样要进行包装。

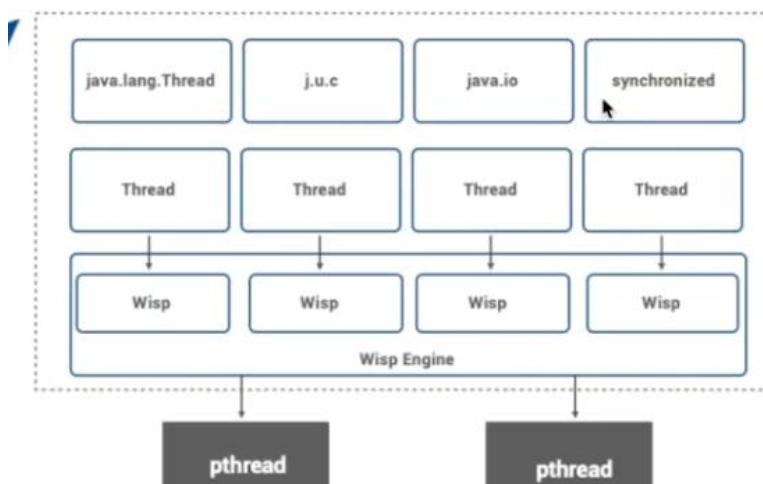
这样可以达到异步的一个性能，同时使用完全阻塞方式去写这个代码，非常高效，C sharp 的协程的一个解决方案，但是这里对这些带 Kotlin 的进行封装，其实是非常繁琐的。

九、Dragonwell: wisp 原理

wisp 其实就是把所有这些需要封装注册函数全部可以做到 JDK，可以看到像 g.u.c 等，其实都被 wisp 做了像上述的封装，会切走协程，等到事件 ready 的时候再把这协程切回来，用户不用关心。



All thread as Wisp



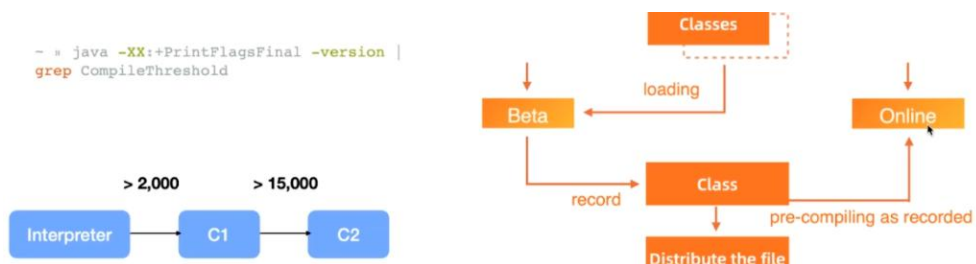
以前阻塞的 API 是支持的完全不需要改代码, 可以把以前的用协程写代码直接切换到协程模型, 进行一个模型转换, 这样从 java thread 和 Pthread 就操作系统 1:1 的模型变到调用大量 wisp 变成少量操作系统线程, 性能大大提高。

十、Dragonwell: JWarmup

JWarmup 特性, 前面提到, Java 方法要执行 2000 次, 然后在 15,000 次才达到一个非常高的效率, 这些执行其实是非常慢的, 要比慢 50 倍, 并且解释执行的时候, 其他编译器其实也是比较少有资源的, 就边编解释执行还要边编译这个方法。

执行的时候 CPU 会打得非常高, 响应特别慢, JWarmu 就是说让 JVM 提前知道哪些方法热的, 在处理请求之前就让这些方法提前被编译掉, 从而避免了前面边解释, 边编译的开销。

模型如图所示, 首先应用被在被他环境被跑的时候, class 被加载的时候, 会产生一些日志, 告诉哪些 class 是热的, 然后这里进行一个 record, 记了一个 class list, 然后把 class list 就是真正热的方法的一个 logo 文件, 给分发到线上环境, 然后线上环境的机器就知道哪些方法热的, 在真正处理用户请求之前, 会根据列表去提前把方法给编译好, JWarmup 它可以大大减少应用预热的一个开销。



Dragonwell 特性: JWarmup

一、JWarmup 背景

(一) 应用程序预热

Java 的方法要被执行时，首先这个方法所在的类需要被 JVM 加载，这个过程包括各类文件的验证、解析、链接以及类的初始化。当这个类被加载进来了以后，JVM 就可以去执行这个方法。

JVM 在刚开始的时候会使用模板解释器去解释执行方法，模板解释器除了一个个去执行方法中的 Bytecodes 之外，还会额外收集关于方法执行动态运行的信息，例如方法执行的调用次数，调用时一些类型的信息等。这些信息都会提供给 JVM 的即时编译器，由它利用这些信息将刚才解释执行发现的热点方法编译成为 Native Code。这样 JVM 就不用模板解释器去执行这些方法，而是去执行性能更高的 Native Code，从而使应用程序的性能得到大幅提升。

当大多数热点方法都被编译成为 Native Code 以后，应用程序的预热就完成了。

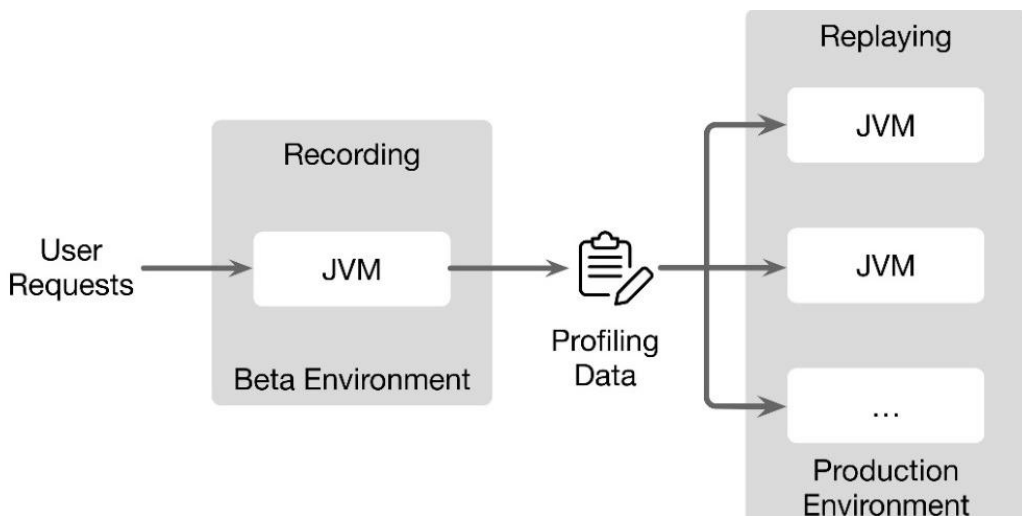
(二) 遇到的问题

Java 有非常丰富的应用场景，一个典型的场景就是我们会用 Java 写一些 Web 服务。在 Web 服务的部署过程中，会发现预热给我们带来非常大的困扰。当我们把一个 Web

服务部署到线上后，应用启动完成，此时就会有大量的用户请求进入。

这个时候由于有大量的热点方法需要被编译，JVM 的编译线程会非常忙碌，因为它需要占用大量的 CPU 将这些方法编译成为 Native Code。同时又因为用户的线程需要执行解决用户的请求，因此它也会占用大量的 CPU，并且会抢占编译线程所使用的 CPU，这样就会导致编译线程无法尽快地把这些热点方法编译到 Native Code，使得应用程序长时间运行在解释执行的状态，降低服务的质量。同时，服务的 RT 会增加，服务所使用的 CPU 也会非常的高，这就是在真实场景中所遇到应用程序预热的问题，接下来我们来看一下阿里巴巴 Dragonwell 8 中的 JWarmup 特性是如何解决这个问题。

二、JWarmup 功能



上方为 JWarmup 的流程图，它将应用程序的发布分成了两个阶段，分别是 Recording 和 Replaying。在 Recording 阶段，JVM 会接受线上的请求，同时记录 JVM 即时编译器它所编译方法的信息，并且将这些信息都输出到一个文件之中。

等到第二次再去启动的时候, JVM 就可以去读取刚刚所记录的这些方法编译的信息, 同时会主动的触发即时编译器编译刚刚记录的热点方法, 使得在用户请求到来之前, 就把热点方法编译成为性能较高的 Native Code, 避免了在用户请求大量进入的时候做编译, 这样就能够进一步提高应用程序的性能, 节约 CPU 使用率。

三、案例演示

(一) Demo 代码

```
1
2 public class TestJWarmup {
3     public static final int FREQ = 60000;
4
5     public static void main(String[] args) {
6         TestJWarmup test = new TestJWarmup();
7         test.foo();
8     }
9
10    public void foo() {
11        for (int i = 0; i < FREQ; i++) {
12            bar();
13        }
14    }
15
16    private void bar() {
17        try {
18            Thread.sleep(1);
19        } catch (Exception e) {
20            e.printStackTrace();
21        }
22    }
23 }
```

下面根据一个简单的例子展示如何使用 JWarmup 功能。

上方为一个简单的 Java 程序, 在这个程序之中有一个循环用来模拟线上应用的一个热点, 循环会反复调用一个方法。

(二) Recording

如何去使用 JWarmup 的记录功能 Recording?

- 首先, 从 <http://dragonwell-jdk.io/> 下载 Dragonwell 8;
- 添加 JVM 参数启动 JWarmup 的记录功能:
 - XX:+CompilationWarmUpRecording
 - XX: CompilationWarmUpRecording=30
 - XX: CompilationWarmUpLogfile=./jwarmup.log
 - XX: -ClassUnloading
- 第一个参数表示去打开 JWarmup 的记录功能;
- 第二个参数表示需要记录的时间, 在当前 Demo 之中选择记录 30 秒;
- 第三个参数表示记录编译信息生成文件的路径, 在这个 Demo 中, 我们将这个文件生成在当前目录下的 jwarmup.log 这个文件;
- 第四个参数是由于 JWarmup 的 Recording 功能不支持 ClassUnloading, 所以需要将这一功能关闭。

当设定的记录时间到了以后, JWarmup 会将记录好的编译信息输出到指定的文件之中, 同时会在应用程序的输出中看到以下这样一条日志, 表明记录是成功的。

```
[JitWarmUp] output profile info has done, file is ./jwarmup.log
```

(三) Replaying

如何使用 JWarmup 的 Replaying 功能？

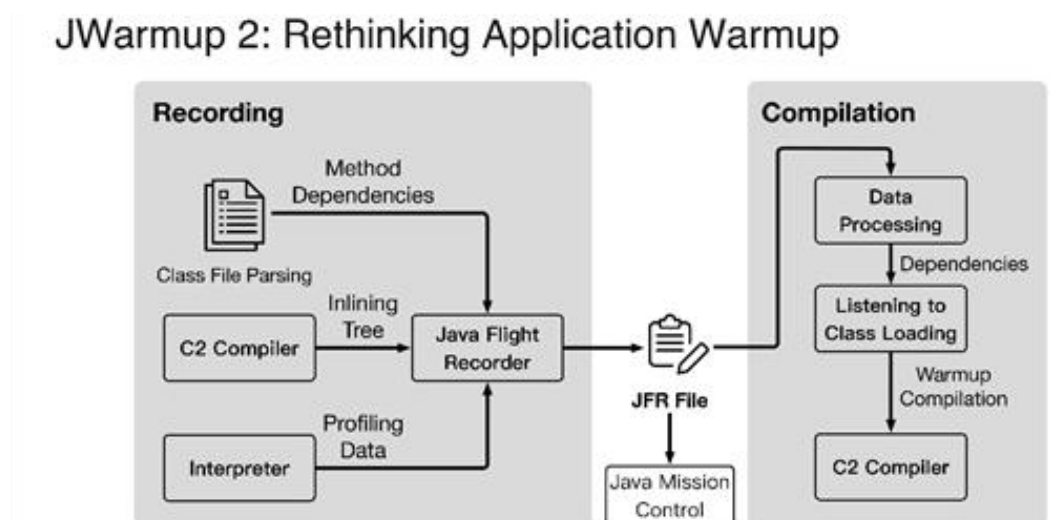
- 添加 JVM 参数：
-XX:+CompilationWarmUp
-XX:+CompilationWarmUpLogfile=./jwarmup.log
-XX:+PrintCompilationWarmUpDetail
- 第一个参数表示要使用 JWarmup 的编译功能；
- 第二个参数需要指定刚刚记录的包含编译信息的文件，在当前 Demo 之中，就是刚刚所记录的当前目录下的 jwarmup.log 文件；
- 第三个参数表示我希望 JWarmup 打印出一些详细的日志，帮助我记录 JWarmup 工具的一些行为。

当把这些参数配置好以后，将服务启动，等待一些关键的类的加载完成，可以使用 `jcmd <pid> JWarmup -notify` 主动触发 Warmup 的编译。

当 Warmup 编译完成后，可以在程序的标准输出中看到下面三条 log，就表示这一次 Warmup 编译是成功的。

```
JitWarmUp [INFO]: start eager loading classes from constant pool  
JitWarmUp [INFO]: start warmup compilation  
JitWarmUp [INFO]: warmup compilation is done
```

以上就是关于 JWarmup 的基本介绍，包含 JWarmup 所需要解决的问题，解决方法，以及用案例讲述如何使用 JWarmup 功能。同时，我们对 JWarmup 这个功能还做了非常多的改进，形成了我们另外一份工作：JWarmup2。



上方为 JWarmup2 整体流程图，可以看到在这份工作之中，我们记录了更加丰富的信息，去更好的解决应用程序预热的问题。

首先我们会使用 JFR（Java Flight Recorder）统一所记录的编译的信息，这些信息也可以形成一个 JFR 文件，使用 JDK 官方所提供的 Java Mission Control 浏览所记录的所有热点方法的信息。

此外，我们除了记录一些方法的编译，还记录了每一个方法它所依赖的类的信息。这样子在我们第二步预热的时候，就可以根据这些依赖的信息，当我们看到一个方法，它所有的依赖的类都被加载了以后，就会自动触发这个方法的 Warmup 编译，避免了人工手动触发 Warmup 编译。

此外，我们还额外记录了这些方法所有的 Profiling 信息，这些信息能够帮助我们在第二步去更好地生成 Warmup 的代码，从而进一步提高应用程序的性能。

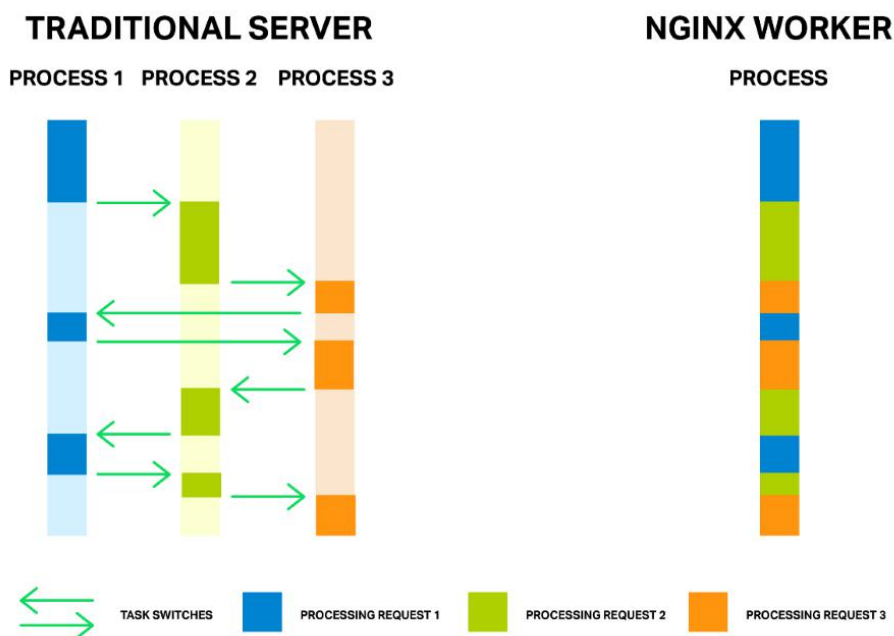
JWarmup2 未来将在阿里巴巴 Dragonwell 11 中开源，欢迎大家使用。

Dragonwell 特性: Wisp

一、协程与异步编程

(一) 多线程和事件模型

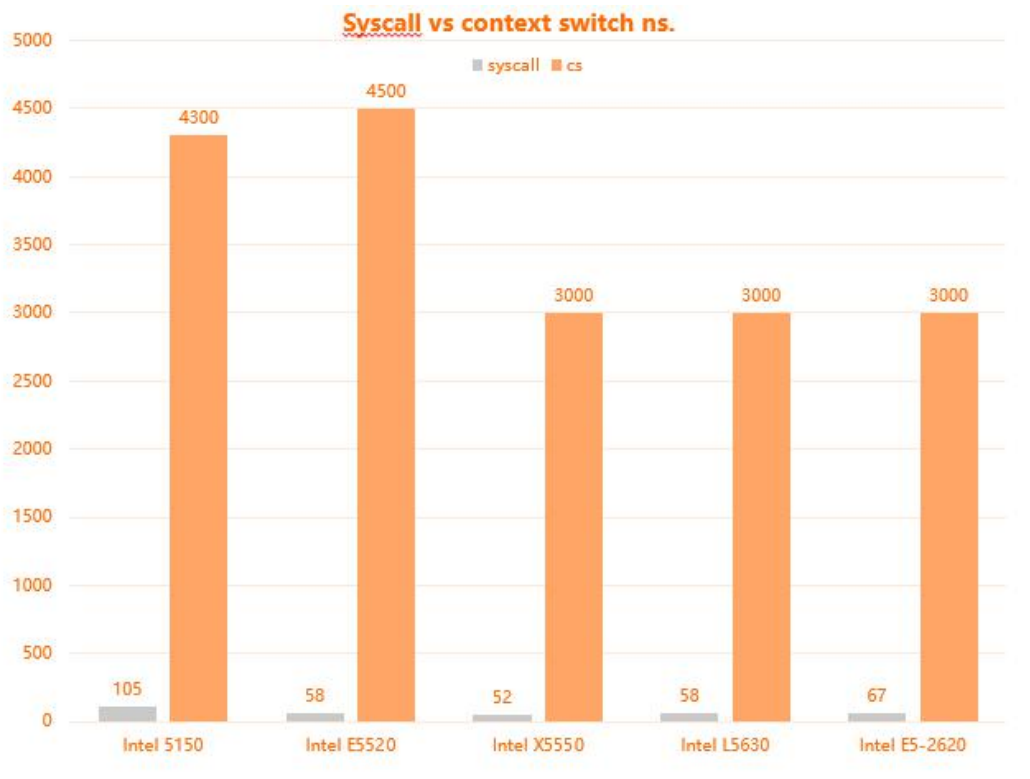
在 Web Server 领域，最早像 Apache Server，大家都是使用多线程模型处理并发。下图左边这张图是通过多个进程去处理多个用户不同的请求，可能在一个单核系统上也可以去创建多个进程来处理这些请求，但这实际上操作系统给大家一个假象：操作系统通过分时互动机制去不停的切换线程，表现出一种正在同时执行的假象。



实际上这个切换是非常消耗资源的，然后我们看右边这张图 NGINX，他率先使用了事件模型，让大家科学认知到在单个线程里通过业务代码去切换不同的上下文，这样可以大大减少操作系统里面限制切换开销，很好的提高性能。

(二) 上下文切换

上下文切换会吃掉宝贵的 CPU 资源，大家很多情况下对上下文有误区，进出内核和调度之间其实很大差异的。假如像刚才这种场景，我们看到多个线程来回调用，那一个线程当它资源耗尽或者比较阻塞的时候，下个线程选谁？其实操作系统需要进行调度，真正的损耗远大于想象。

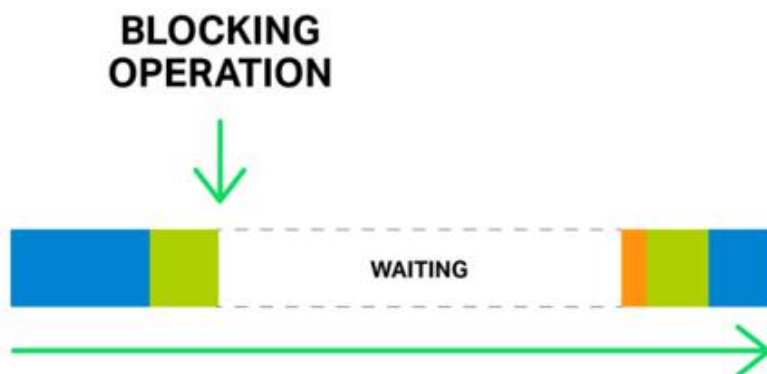


我们可以看到进出内核是上图左边灰色这一列，它的耗时是很小的，可能在几十到一百纳秒级别。然后假如这一次系统调用它触发了切换，比如读一个程序里面有数据，信令要挂起会触发上下文切换，如果希望有调度，开销就会很大，会达到 40 倍左右。

（三）使用异步编程

所以如果在编程中引发调度的切换开销是很大的，我们应该尽量避免。怎么避免呢？答案就是异步编程，在 node.js 里面，我们可以使用大量 callback 区域处理业务逻辑。当使用 callback 以后，代码可能会变成这样一种三角形，因为每一个组织方式，它后面返回值都要带 callback 调用，都会缩进去一层。这样业务逻辑非常难以维护。

其次是即便我使用了异步编程，但可能还是不小心在现实里面使用了一段阻塞代码，下图是 NGINX 官网所提供的图片，虽然我自己去切换不同的请求处理，但是中间可能还是不小心调用了操作系统的一个阻塞方法。



为了解决这个问题，NGINX 虽然是一个号称纯异步事件驱动模型，但是它最近也引入了线程池去处理这种可能阻塞现实的情况。

(四) 引入协程

其实最早在操作系统里没有协程的概念，大家都是通过协程做逻辑上抽象来帮助我们写并发代码。

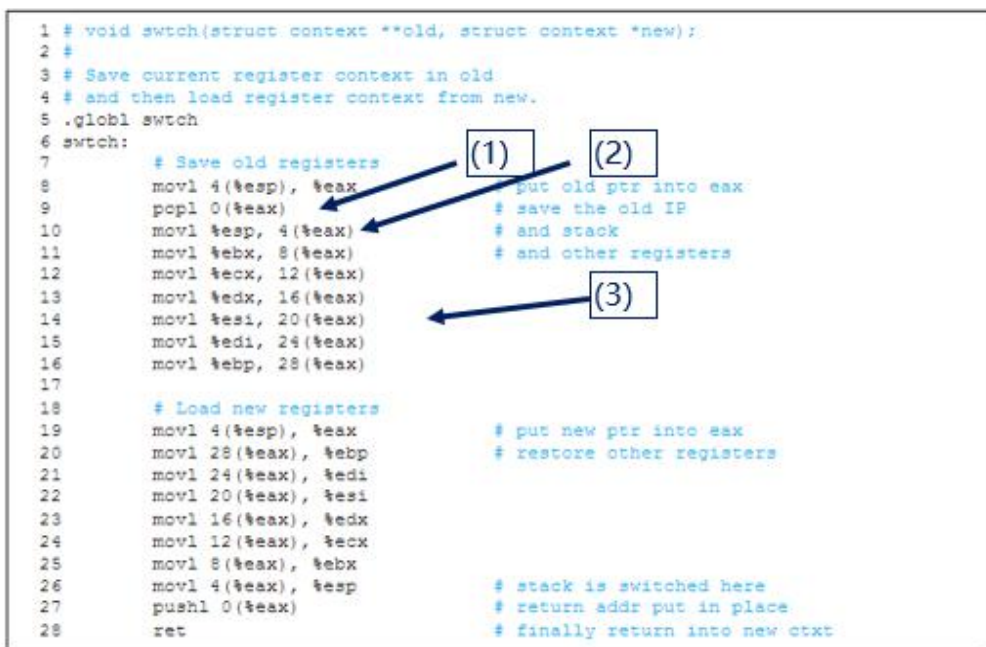
```
/* Parser code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (isalpha(c)) {
        do {
            add_to_token(c);
            c = getchar();
        } while (isalpha(c));
        got_token(WORD);
    }
    add_to_token(c);
    got_token(PUNCT);
}
```

```
/* Decompression code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (c == 0xFF) {
        len = getchar();
        c = getchar();
        while (len--)
            emit(c);
    } else
        emit(c);
}
emit EOF;
```

比如说这里有两段 code，一段是解压的 code，一段是 parser 的 code。大家要从解压数据结构里面去解析数据，这里对数据进行简单的 encode，如果 char 是普通字符，会直接返回。若是特殊字符，可能就进行一个长度 encode。用协程来组织逻辑，emit() 和 parser::getchar() 会切换到另一个协程，如果没有协程需要两个线程结合 pipe 来组织，但如果有协程，我们可以在 frame 里面直接控制，逻辑清晰且性能高。

我们看怎么实现协程。协程的执行上下文其实包括这几个部分，当前的栈、局部变量、当前代码位置，这些其实都可以通过数据表示。

The xv6 Context Switch Code



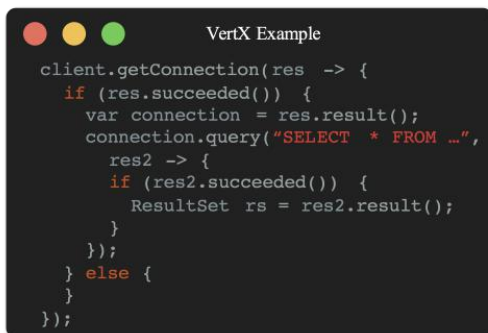
与 OS 内的线程切换方式一致

- 1) 保存 pc
- 2) 保存 sp
- 3) 保存 callee-save 寄存器

保存完这些后，将来想回去，只要通过反向计算器 `pop` 出来，就会回到之前上下文。协程场景下，`emit` 和 `getchar` 都是通过这种方式去实现的。

(五) 现代编程语言中的协程

异步编程



```

VertX Example

client.getConnection(res -> {
    if (res.succeeded()) {
        var connection = res.result();
        connection.query("SELECT * FROM ...",
            res2 -> {
                if (res2.succeeded()) {
                    ResultSet rs = res2.result();
                }
            });
    } else {
    }
});
  
```



协程

ES7



NGINX Wisp



```

Kotlin 库支持

suspend fun Client.aGetConnection(): .. =
    // suspendCoroutine { cont ->
    getConnection( conn ->
        标记 { cont.resume(conn) })
    }

conn = client.aGetConnection();
rs = conn.aQuery("SELECT * FROM ...");
  
```

左边是 VERT.X，Java 里面最近比较流行的框架，想要制作的就是 Java 里的 node.js 的生态，我们可以看到官方所提供的连接数据库例子。

Client.getConnection, 来获取数据库连接，但它不是说立马返回一个连接给到我们，而是提供 callback，然后这个 result 里面表示执行是否成功，如果成功的话，我们可以通过 result 去拿到 connection。这就是通过义务编程的方式，去让我们在线程里面处理大的逻辑，NGINX 就是这样的一种方式。这样代码其实看起来是非常难以维护的，比如在里面需要通过 result set 去把数据放到缓存里面，又是一个远程调用需要阻塞，可能又是一种 callback，这个嵌套会非常深，非常难处理，由于我们都是 callback，所以这个站就没法被维持，假如在这个地方有异常就非常难以处理。

现代编程语言是怎么解决这个问题，我们给的答案是协程。ES7、C# 他们都提供协程来帮助解决这类问题。我们以一段 Kotlin 代码为例，看协程怎么帮助代码改写成非常直观的代码，Kotlin 里面通过 suspend 关键字来表示，函数是可以被挂起的，然后它也可以在 client 上新加的方法，新的方法叫 Agetconnection。里面调用 Kotlin 提供的非常

medical 的方法，他会获取一个当前执行上下文的 connection，让我们 getConnection 直接调用。getConnection 的 callback 是恢复当前协程的执行，并且把拿到 connection 作为返回值。这样实际上不用一直占着 CPU 资源，实际上调度器会继续去调度其他执行，一旦进行这类封装以后，我们看到代码可以被简化为下面这种形式。

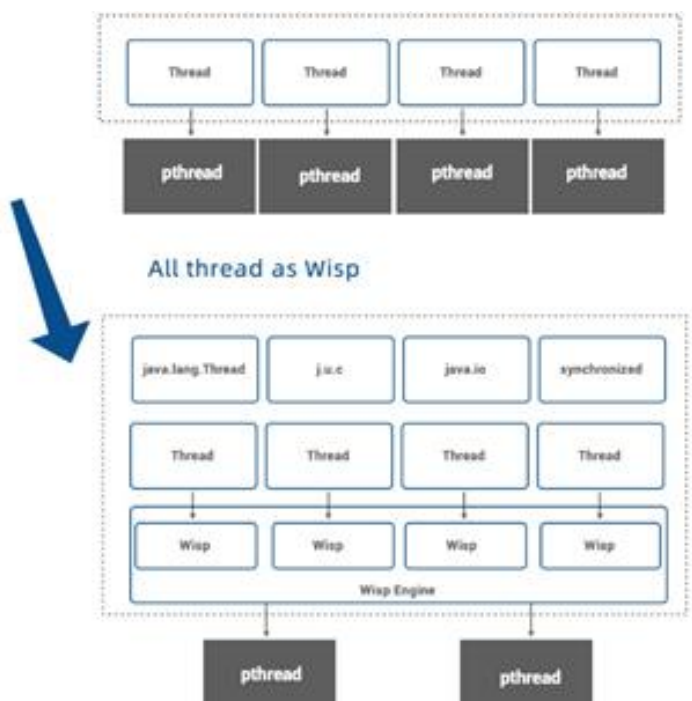
```
conn = client.aGetConnection();  
rs = conn.aQuery("SELECT * FROM ...");
```

```
Conn=clinet.AGetConnection () ;
```

```
然后 rs= Conn .aQureythat( "SELECT * FROM ..." )
```

这段代码相比左边这段代码那就是大大简化了，但我们要做对这种回调形式进行封装。

(六) Dragonwell: Wisp 原理



既然要对这么多回调形式进行封装，工作量是非常大的，能不能在更底层去解决，为什么就提供了这一层帮助？因为 jdk 提供所有的阻塞方式都是在 jdk 里面提供的。比如说 `Java.lang.Thread`、`j.u.c`、`java.io`、`synchronized` 这些都是有可能阻塞 API。在这些 API 上我们都做了封装，Wisp 把这些脏活苦活全部给做掉了。Wisp 还对现成模型进行一个映射。我们知道 Java 里面的 Java thread 和操作系统 pthread 是 1:1 的映射关系，大量线程使用的话就会导致前面提到的上下文切换问题。但是在 Wisp 下我们每一个线程都被映射到一个 Wisp，wisp 执行过程中可能阻塞 CPU，然后这时候就可以让 pthread 调动其他 Wisp，调度效率非常高，可以免费提高应用的性能。

二、使用 Wisp 提升微服务性能

Dragonwell: Wisp demo

下面在 Dragonwell 下用 使用 Wisp 提高性能的例子

```
Running 6s test @ http://192.168.1.101:8080
2 threads and 16 connections
Thread Stats      Avg          Stdev         Max      +/-  Stdev
  Latency    522.67us    0.89ms    13.60ms    93.02%
  Req/Sec    24.62k     1.02k    26.90k    69.67%
Latency Distribution
  50%    255.00us
  75%    441.00us
  90%     0.99ms
  99%     4.93ms
298824 requests in 6.10s, 107.15MB read
Requests/sec: 48992.87
Transfer/sec: 17.57MB
```

```
Running 6s test @ http://192.168.1.101:8080
2 threads and 16 connections
Thread Stats      Avg          Stdev         Max      +/-  Stdev
  Latency    271.91us   163.31us    5.16ms    98.65%
  Req/Sec    30.18k     518.61   31.26k    66.39%
Latency Distribution
  50%    257.00us
  75%    299.00us
  90%    341.00us
  99%    458.00us
366196 requests in 6.10s, 145.98MB read
Requests/sec: 60038.35
Transfer/sec: 23.93MB
```

上边这张图是不开 Wisp，使用 wrk 压测工具去压这台机器，192.168.1.101，8080 端口，平均的延迟是 522 微秒，QBS 是不到 5 万，在同个应用完全不改代码情况下，我们调整一下界面参数把 Wisp 打开，然后线程就被完全意识到协程了，latency 降低到 270 多微秒，QBS 变成了 6 万多，大概有 20% 多的性能提升，这不需要修改任何应用代码，是一个免费的性能午餐，所以推荐大家可以通过 Wisp 提高我们微服务的性能表现。



扫一扫
免费领取同步课程



钉钉扫一扫
进入官方答疑群



开发者学院【Alibaba Java 技术图谱】
更多好课免费学



阿里云开发者“藏经阁”
海量电子书免费下载