Alibaba Group
阿里巴巴集团

Extreme Scaling with Alibaba JDK

Sanhong Li

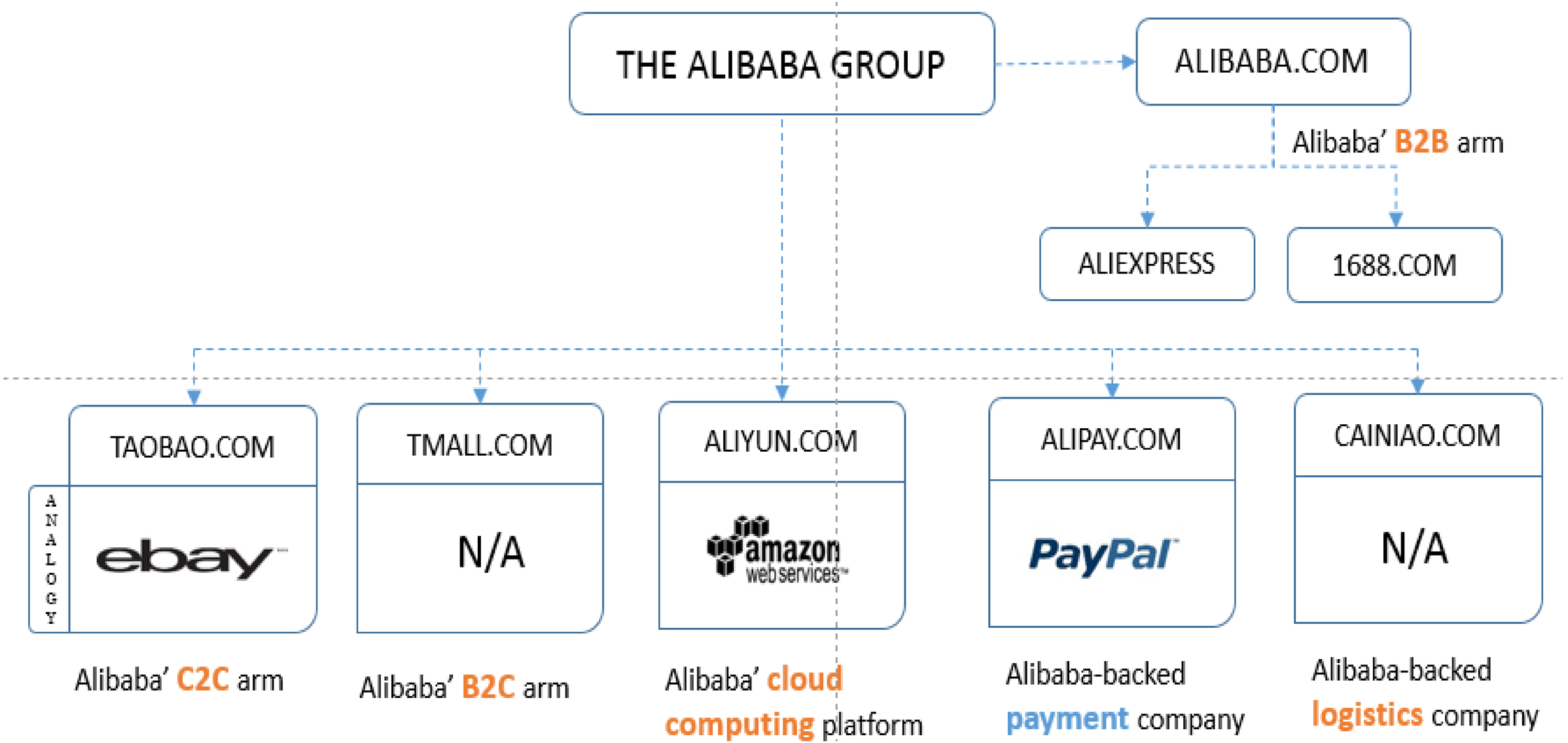
JVM Lead at Alibaba

系统软件事业部：打造具备全球竞争力、效率最优的系统软件

Agenda

- Introduction: Java at Alibaba
- AJDK: Optimizing OpenJDK for our needs
- Tools: Diagnostics and Troubleshooting

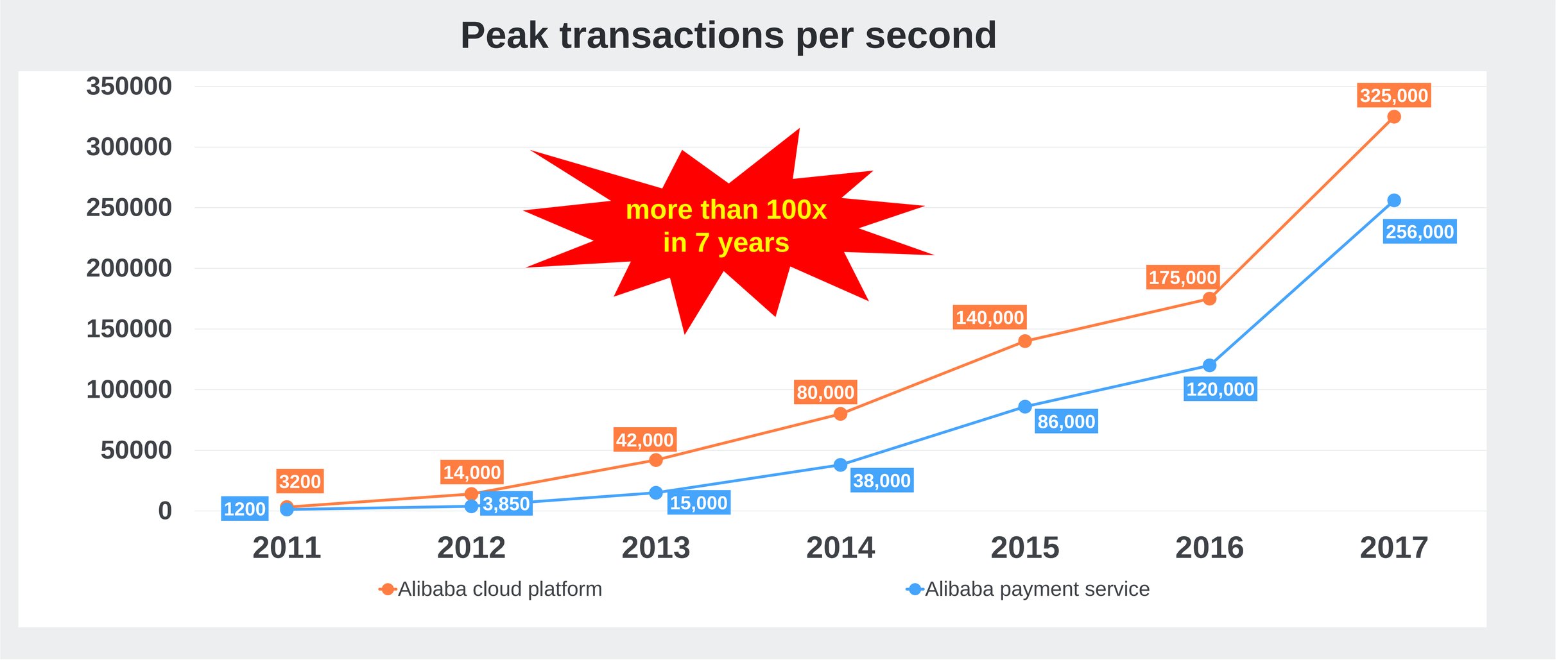
Alibaba Family Tree



Alibaba in Russia : AliExpress

- Biggest e-commerce platform in Russia
- Global B2C division of Alibaba
  - 100 million customers+ from 220 countries
- Engineering team in Russia is responsible for building entire distributed software stack to serve AliExpress' global customers, merchants.

Singles 'Day in China



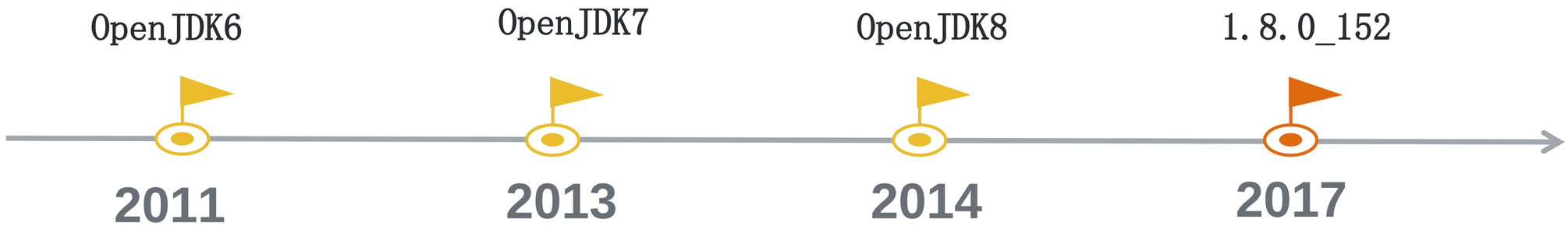
China's Singles Day: the world's biggest online shopping day

https://www.practicalecommerce.com/sales-report-2017-thanksgiving-day-black-friday-cyber-monday
https://www.forbes.com/sites/helenwang/2017/11/12/alibabas-singles-day-by-the-numbers-a-record-25-billion-haul/

Java usage in Alibaba

- **Building most of its software based on rich open-source ecosystem**
  - Implemented for **online trading, payments, logistics**, and a lot of other things.
- **Running Java at massive scale**
  - Million instances of JVMs, serving insurmountable number of requests every second
- **Service oriented architecture**
  - Services communicate with each other via RPC
  - typically, many JVMs as one cluster per service
- **Heterogeneous**
  - The native libraries written in C/C++ communicate with Java via JNI

OpenJDK at Alibaba



OpenJDK open source ecosystem is critical for Alibaba business, which enables us to:

- Incubate new Java technology to meet Alibaba's rapid business needs
- Handle infrastructure Java problems more quickly, no need waiting for official release.
- Build more better troubleshooting services to our Java developers

Challenge #1

- Thousands of thousands of JVMs are deployed in our datacenter
- Challenge with increasing cost
  - How to reduce the Java appetite for resources

Challenges #2

- Ability to handle huge transaction volume
- Java promises:
  - “Infinite memory” by Garbage-Collector
  - Native performance speeds by Just-in-Time
- In opposite side, suffering from
  - Notorious “Stop-the-World” & Time to warmup
  - Both get worse and worse when complexity of application increased

Challenge #3

- Hundreds of thousands of Java applications with above a billion lines of code.
- Challenges with diagnostic
  - Not enough depth to diagnose
  - Current tools not suited for production

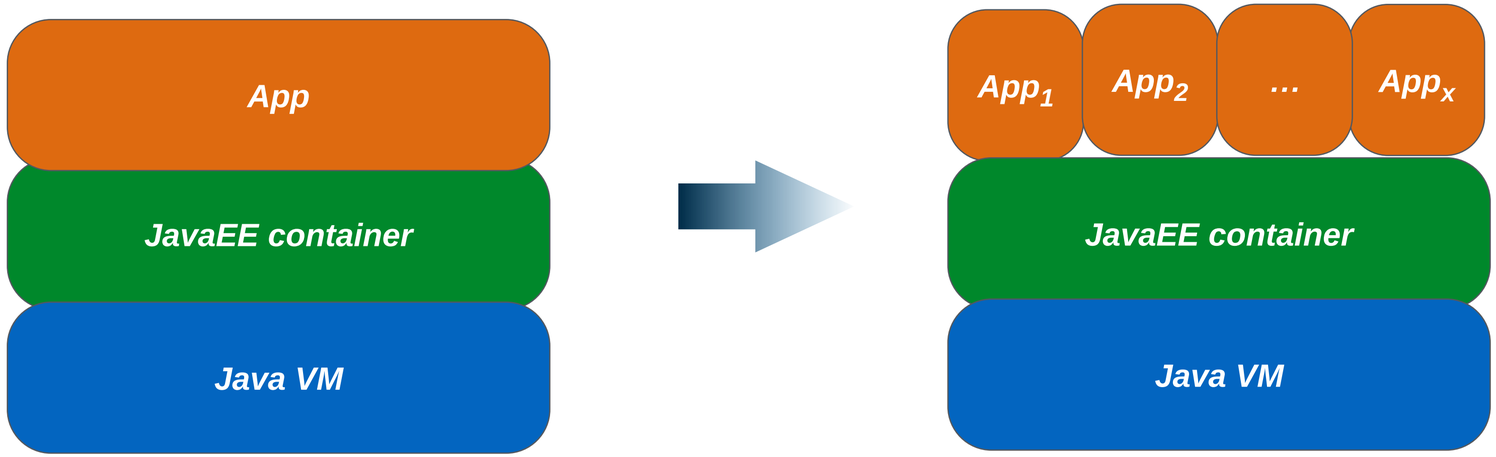
AJDK: customize OpenJDK for our own needs

- ✓ *Multi-tenant:* run multiple apps in same instance safely
- ✓ *GCIH:* new GC free mechanism for data cache
- ✓ *Wisp:* simplicity of synchronous, performance of asynchronous
- ✓ *JWarmup:* priming online application for speed
- ✓ *ZProfiler:* fine grained, low-overhead Java performance profiler

#1 Multi-tenant JVM

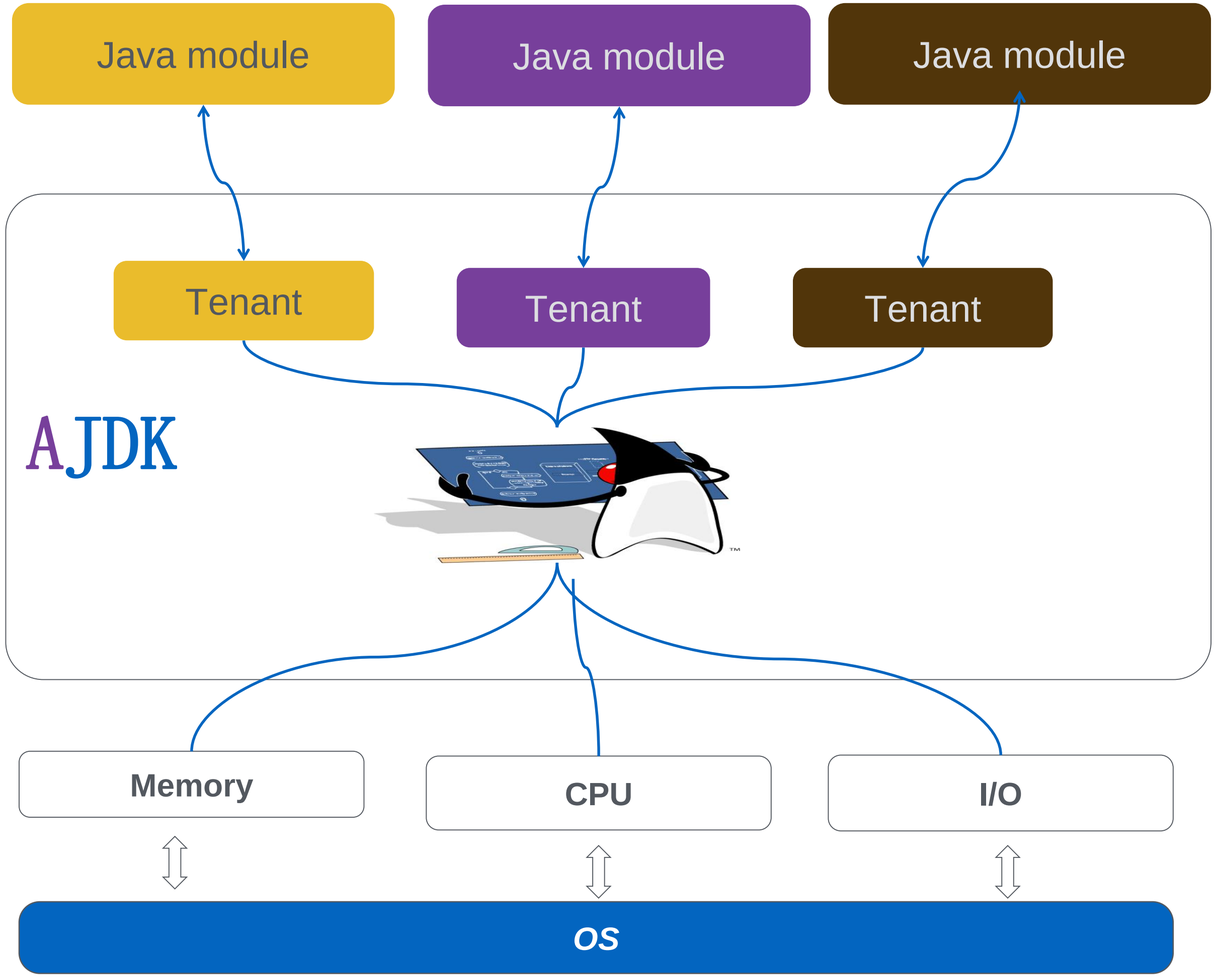
run multiple apps(tenant) in same instance safely

Single vs multiple instance deployment



- JavaEE containers such as Apache Tomcat support deploying multiple web applications into the same container
- We don’t do this in real production environment , one of major reasons is the absence of resource consumption control

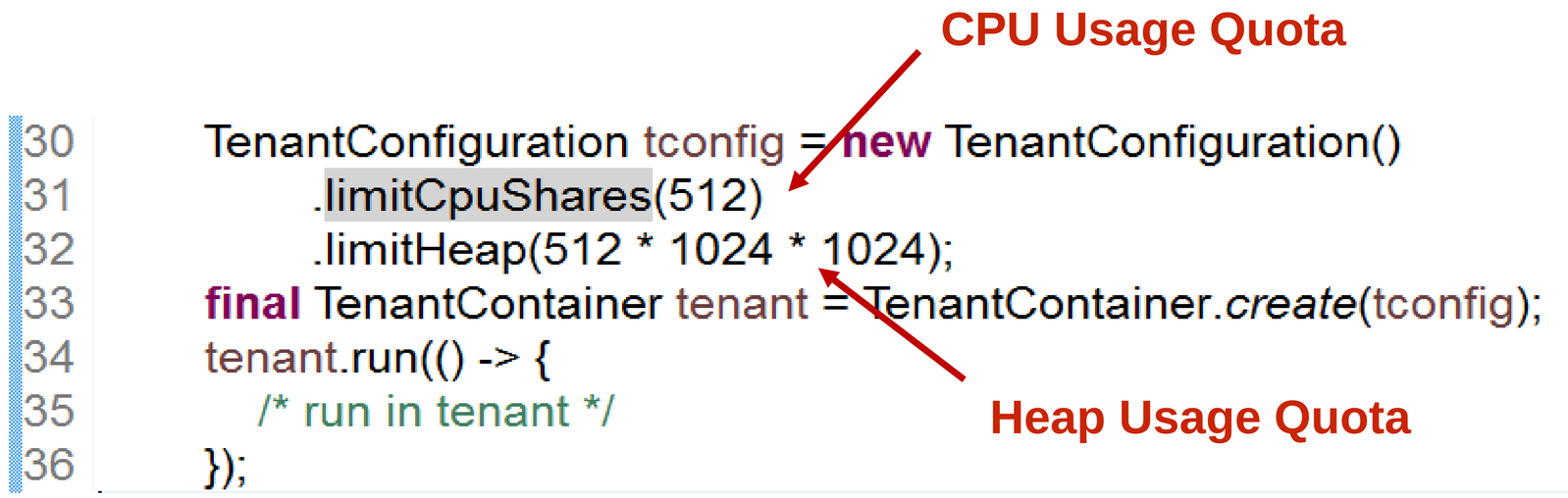
Building containers inside JVM



- ✓ Create “tenant” resource container per java module referred as “tenant”.
- ✓ The java module may be mapped as a single thread, a thread group in runtime.
- ✓ The resources consumed by an application unit are accounted for per tenant

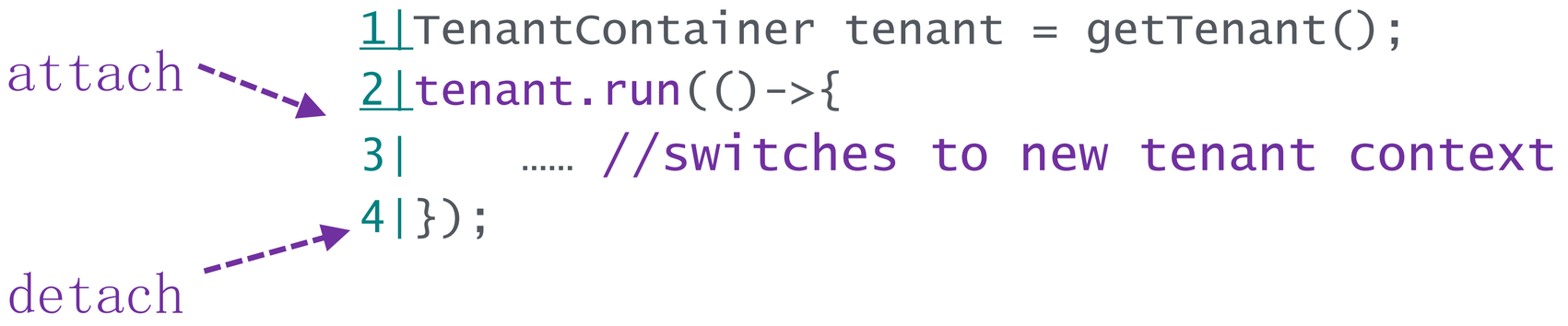
Tenant Container API

- TenantConfiguration: maintains resource configuration information for tenant
- TenantContainer: represents resource container for tenant, thread switches to the context of tenant by calling TenantContainer.run()



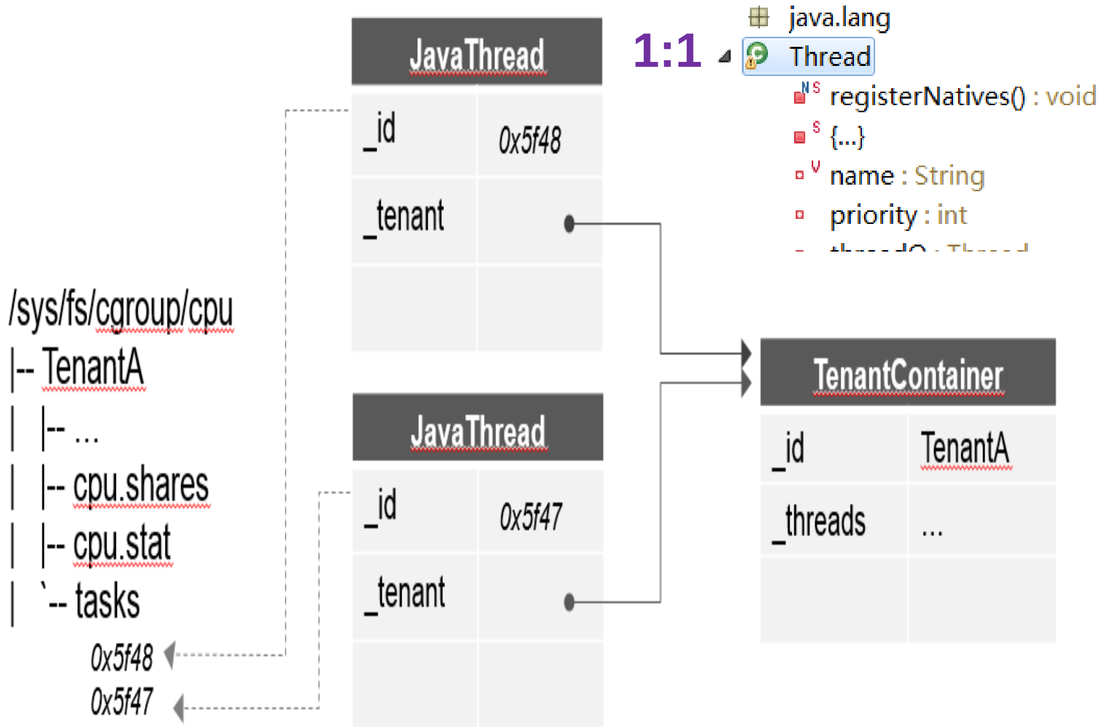
Thread and tenant

- Attach/detach
  - Attached to tenant automatically after entering into TenantContainer.run()
  - Detached after leaving from run block
- Inheritance
  - Threads forked in TenantContainer.run() will be attached to parent thread's tenant automatically



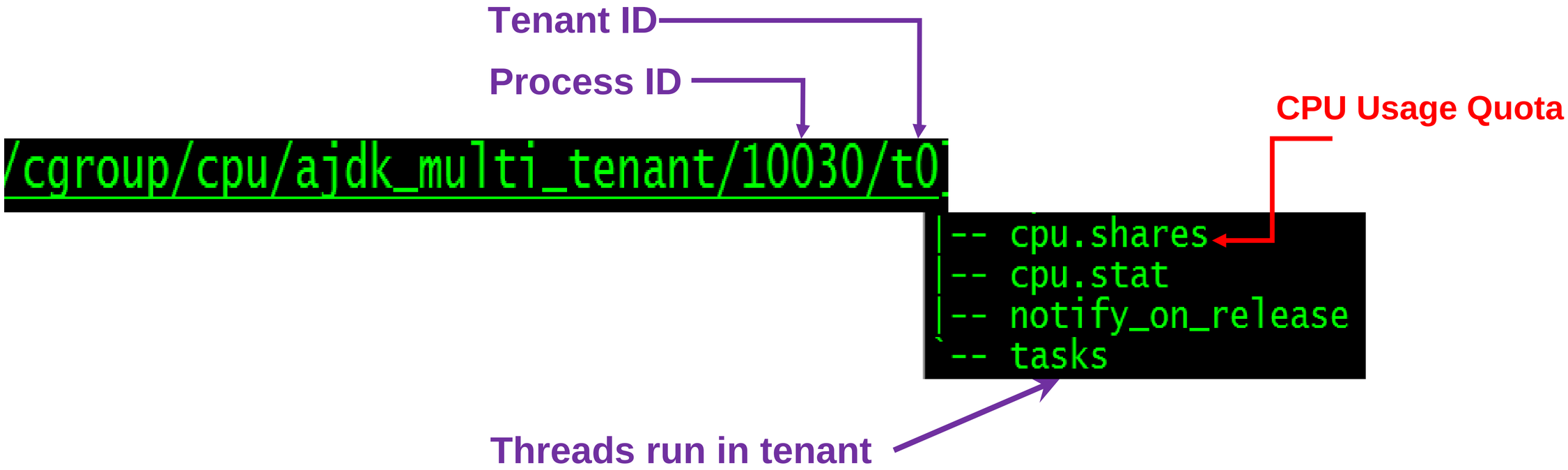
CPU Throttling per Tenant

- Control Group directly provided by Linux kernel
  - Allocate resources for aggregated processes(threads)
    - Partition all processes and children into groups
    - Organize groups in hierarchies
    - Associate groups with particular resources
    - Manage resource among groups
- Delegate CPU management to Control Group(cgroup)



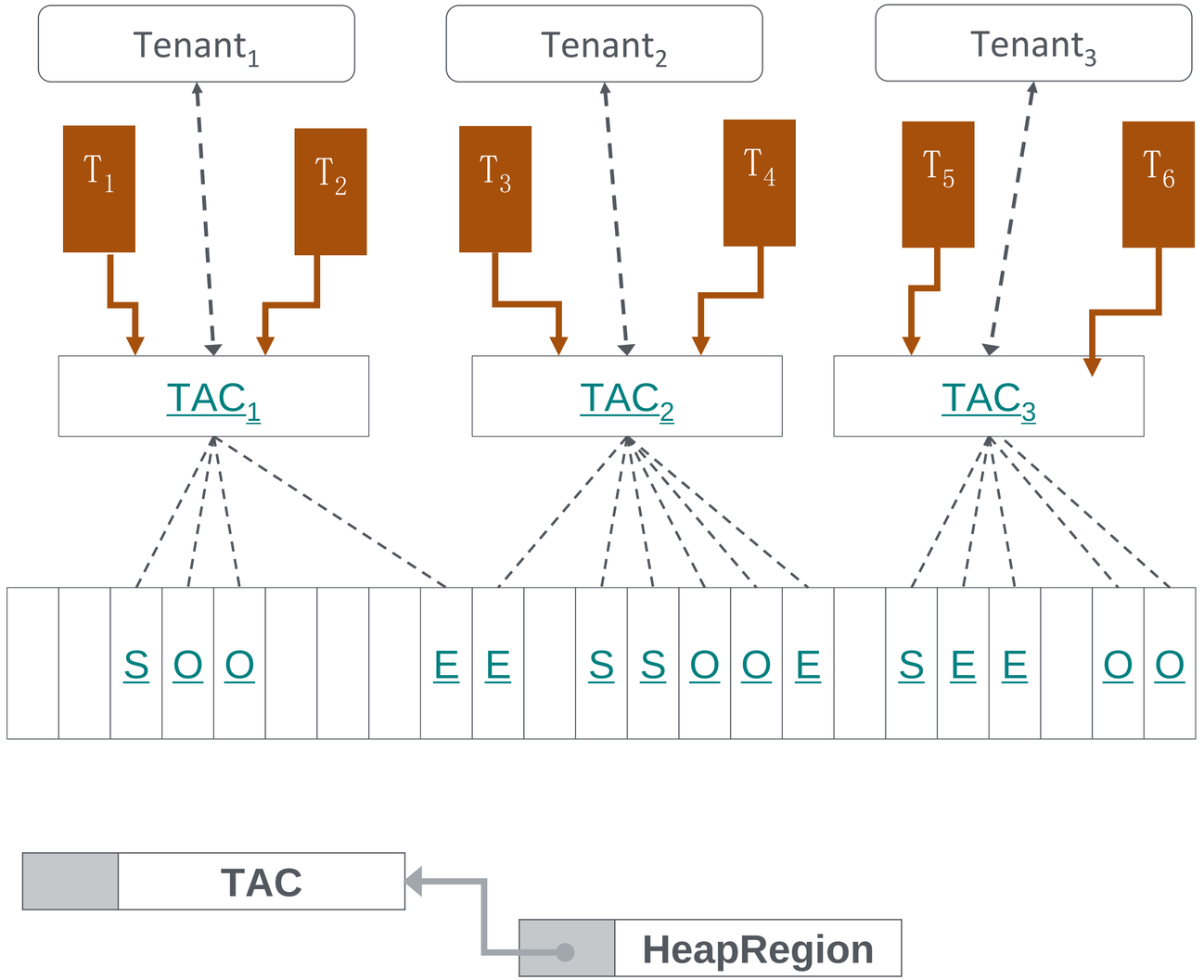
CPU Throttling per Tenant (2)

- The directory tree mirrors the control group



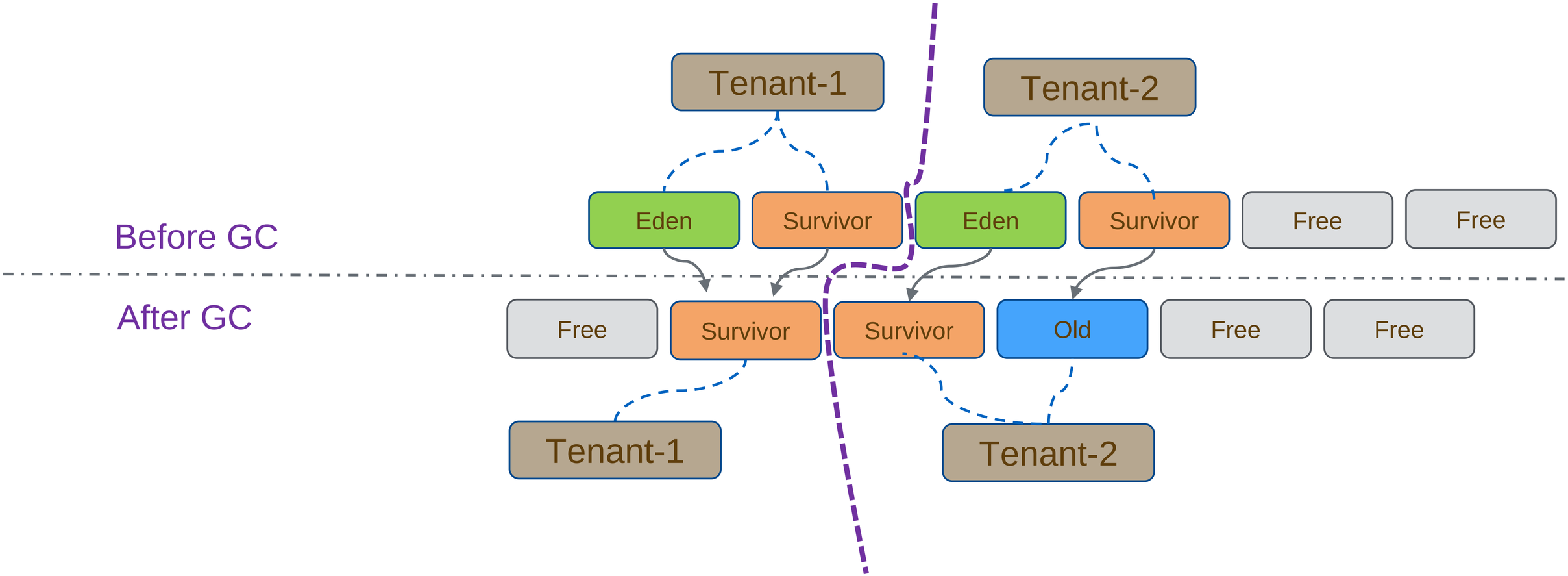
Tenant Heap Management

- G1GC divides the entire heap into equally-sized regions, well suited for tenant management
- TAC(tenant allocation context) manages the regions for tenant
- Code running in a tenant allocates objects in a region it owns
- New regions can be requested up to tenant maximum reservation



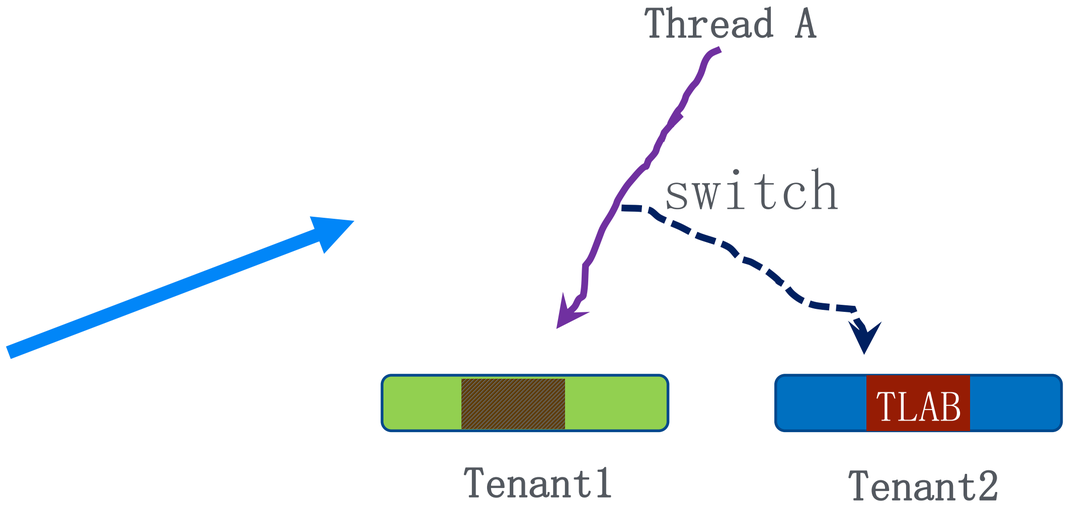
Tenant Heap Management (2)

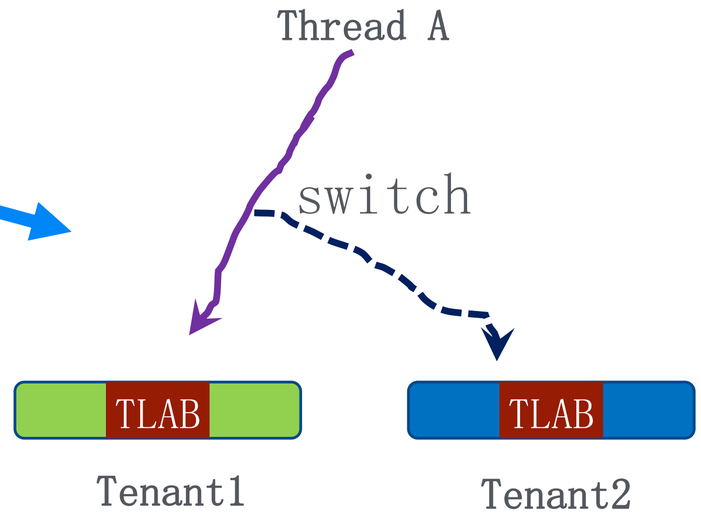
- Mark and Copy: copy object from “from space” to “to space”
  - Be sure to copy object into correct tenant region



Other Changes for Tenant Isolation

- TLAB(Thread Local Allocation Buffer)
  - Option 1: Retire TLAB when thread switches into different tenant
    - Easy to implement, but waste space on TLAB, might cause frequent YGC
  - Option 2: Tenant aware TLAB, maps TLAB into correct heap region of tenant





Other Changes for Tenant Isolation

- IHOP(Initiating Heap Occupancy Percent)
  - Calculate IHOP per tenant
  - Otherwise, may suffer FULL GC when only some tenant reaches to limit

```
product(uintx, InitiatingHeapOccupancyPercent, 45,                          \
        "Percentage of the (entire) heap occupancy to start a "             \
        "concurrent GC cycle. It is used by GCs that trigger a "            \
        "concurrent GC cycle based on the occupancy of the entire heap, "   \
        "not just one of the generations (e.g., G1). A value of 0 "         \
        "denotes 'do constant GC cycles'.")                                 \
                                                                            \
```

GCIH: GC Invisible Heap

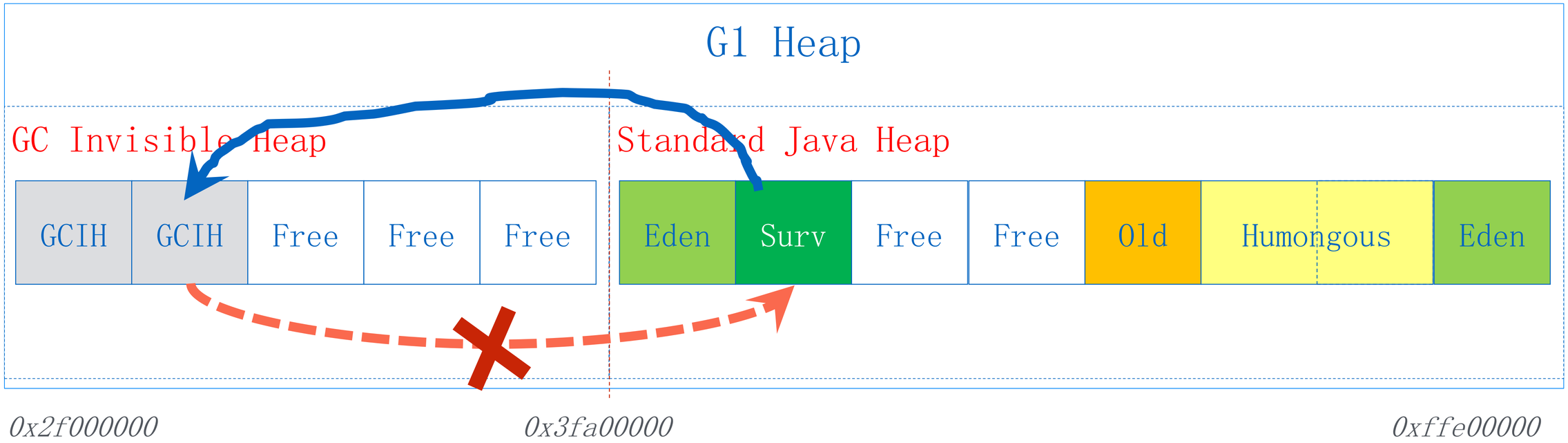
- Based on tenant heap management
- Heap regions are managed by GCIH tenant
- Allows users to allocate(moveIn) normal java objects in GCIH region via explicit API
- Advantages
  - Use as normal java objects
  - Don’t incur GC overhead

Deep copy objects into GCIH region

```
5  public static void main(String[] args) {
6      Object gcihObj = GCInvisibleHeap.moveIn(new Object());
7      assertTrue(GCInvisibleHeap.contains(gcihObj));
8      //.....
9  }
```

Tips: Define your GC according to your application logic

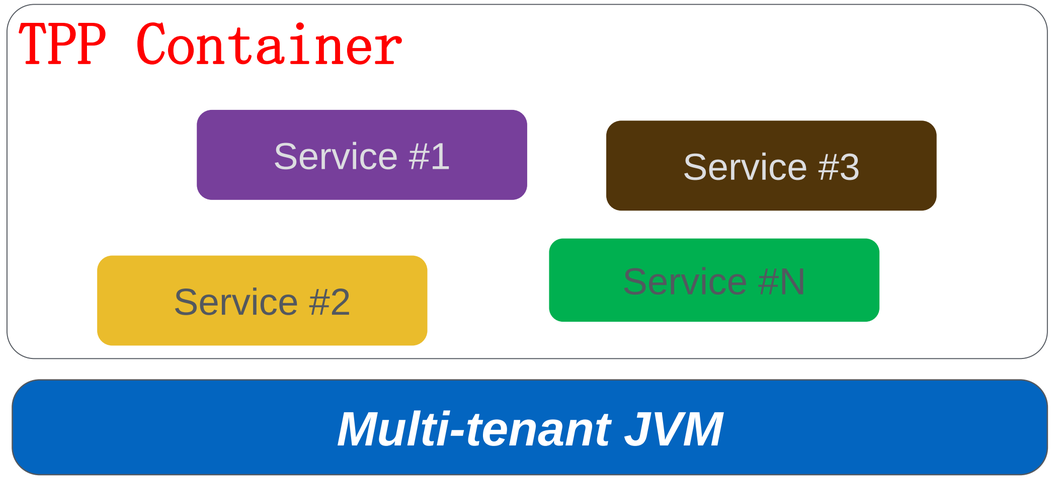
Heap Overview of GCIH



- GCIH objects cannot point to objects in standard heap
  - Implement via pre-write barrier
  - Throw exception if rule is violated

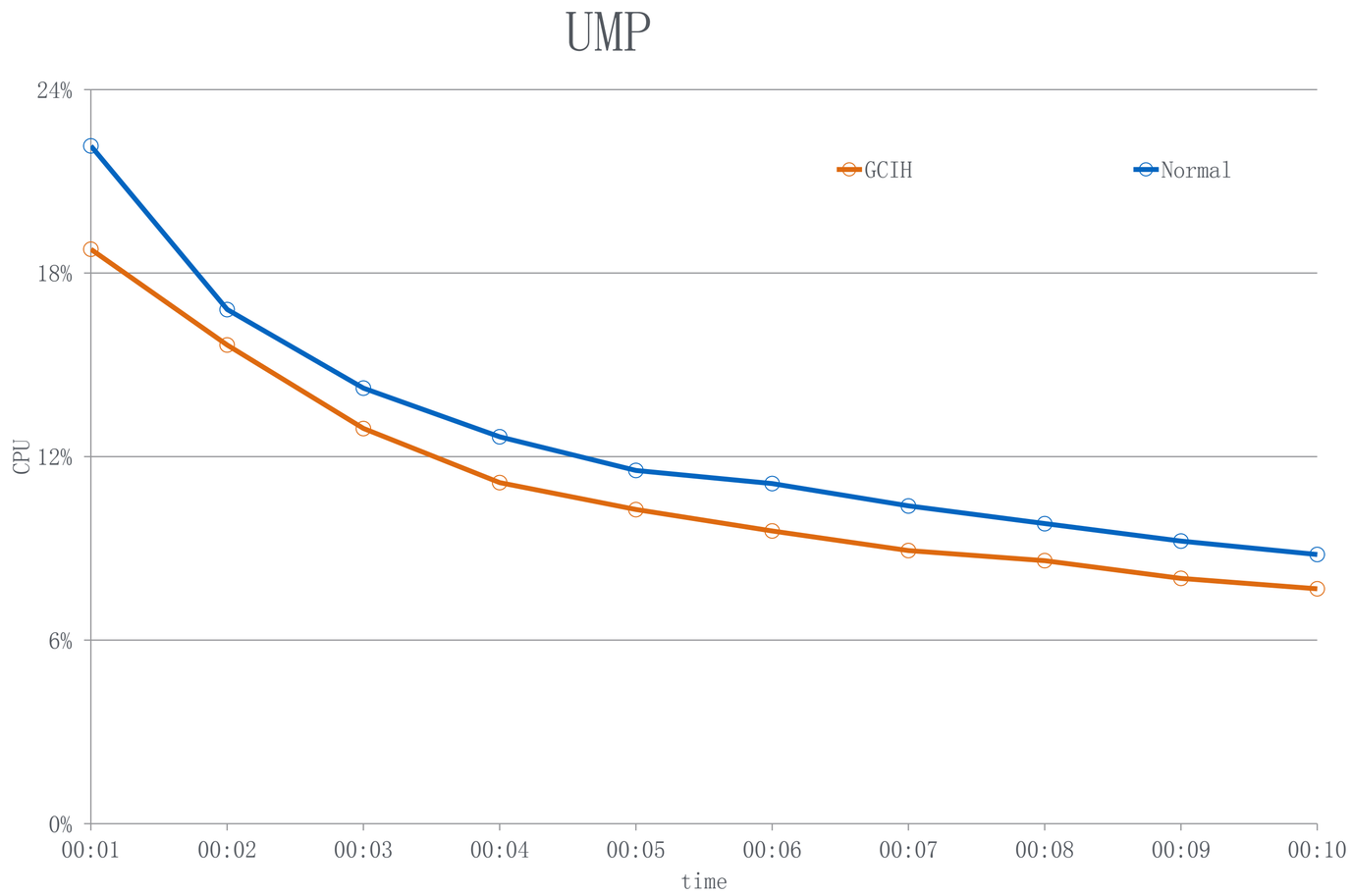
Multi-tenant JVM: where we use this?

- Taobao Personalization Platform(TPP), which is recommendation system for personalization in online shopping applications.
- Singles 'day in 2017, the peak QPS(queries per second) of TPP reached 2,000,000~
- TPP builds resource management capacity based on Multi-tenant JVM.
  - Consolidate multiple micro services (100+) to run in same container
  - Each of them is generally developed by different Business Unit
  - Limit the CPU/HEAP usage of each of service



GCIH: Where we use it?

- GCIH technology has been used in UMP.
- By using GCIH, the data can be cached in local, no need retrieving them from remote(Cache server or Database)
- CPU saving: 18+%



UMP: ecommerce application for complex discounting calculation

#2 Coroutine in Java

simplicity of synchronous, performance of asynchronous

Threads and Context Switch Overhead

- Java provided powerful “threads and synchronization” primitives since 1.0
- In Hotspot JavaVM, it has been implemented as OS kernel thread
- Context switching happens between Threads are heavy

- 4 IO threads, and 200 business logic threads
- Much more of CPU time(**sy**) are consumed by thread context switch, **~100k**

```
$vmstat -w 1
procs -------------------memory------------------ ---swap-- -----io---- --system-- -----cpu-------
 r  b       swpd       free       buff      cache   si   so    bi    bo   in   cs  us sy  id wa st
 4  0          0    5796820          0    1742716    0    0     3   135   28   14  16 10  73  0  0
 7  0          0    5796812          0    1742716    0    0     0     0 143129 105630  54 35  11  0  0
15  0          0    5796812          0    1742716    0    0     0     0 143230 105439  54 35  12  0  0
16  0          0    5796812          0    1742716    0    0     0    12 142786 104645  54 35  11  0  0
18  0          0    5796812          0    1742716    0    0     0     0 143969 106339  53 36  11  0  0
31  0          0    5796812          0    1742716    0    0     0     0 142915 105740  52 36  11  0  0
13  0          0    5796412          0    1742716    0    0     0    16 143267 106121  53 34  12  0  0
15  0          0    5796412          0    1742716    0    0     0     0 142993 105464  52 35  12  0  0
11  0          0    5796412          0    1742716    0    0     0     0 143461 105810  53 35  12  0  0
```

Wisp: build user-mode thread in Java

- Add continuation primitive support into AJDK (Thanks JKU’ previous work)
- Implement user-mode scheduler which takes overs scheduling coroutines in same thread
- Fully transparent to java developers, almost zero code change to application

Wisp: is the light phenomenon traditionally ascribed to ghosts

Wisp: build user-mode thread in Java

Wisp technology has been proved in large scale real production at Alibaba

Demo for Our Approach

```
23          ExecutorService executorService = Executors.newCachedThreadPool();
24          CountDownLatch done = new CountDownLatch(1);
25          executorService.execute(() -> {
26              try {
27                  ServerSocket serverSocket = new ServerSocket(PORT);
28                  Socket inputSocket = serverSocket.accept();
29                  serverSocket.close();
30                  // coroutine for reading data from socket.
31                  InputStream is = inputSocket.getInputStream();
32                  is.read(buf);
33                  done.countDown();
34              } catch (Exception e) {
35              }
36          });
37          executorService.execute(() -> {
38              try {
39                  Socket outputSocket = new Socket("localhost", PORT);
40                  OutputStream os = outputSocket.getOutputStream();
41                  // coroutine for writing data from socket.
42                  os.write(buf);
43                  outputSocket.close();
44              } catch (Exception e) {
45              }
46          });
47          done.await();
48          System.in.read();
```

Create normal thread pool

Create Runnable for

- accept new socket
- Read in data

Create runnable for

- Connect to server
- Write out data

Demo for Our Approach Revisited

```
23          ExecutorService executorService = Executors.newCachedThreadPool();
24          CountDownLatch done = new CountDownLatch(1);
25          executorService.execute(() -> {
26              try {
27                  ServerSocket serverSocket = new ServerSocket(PORT);
28                  Socket inputSocket = serverSocket.accept();
29                  serverSocket.close();
30                  // coroutine for reading data from socket.
31                  InputStream is = inputSocket.getInputStream();
32                  is.read(buf);
33                  done.countDown();
34              } catch (Exception e) {
35              }
36          });
37          executorService.execute(() -> {
38              try {
39                  Socket outputSocket = new Socket("localhost", PORT);
40                  OutputStream os = outputSocket.getOutputStream();
41                  // coroutine for writing data from socket.
42                  os.write(buf);
43                  outputSocket.close();
44              } catch (Exception e) {
45              }
46          });
47          done.await();
48          System.in.read();
```

- Two coroutines instead of two threads

```
"main" #1 prio=5 os_prio=0 tid=0x00007ff3d404f000 nid=0x49ee runnable [0x00007ff3d6ca3000]
   java.lang.Thread.State: RUNNABLE
        at java.io.FileInputStream.readBytes(Native Method)
        at java.io.FileInputStream.read(FileInputStream.java:255)
        at java.io.BufferedInputStream.fill(BufferedInputStream.java:246)
        at java.io.BufferedInputStream.read(BufferedInputStream.java:265)
        - locked <0x00000006cbc1c680> (a java.io.BufferedInputStream)
        at WispCoroutineTest.main(WispCoroutineTest.java:93)

- Coroutine [0x7ff3d40a4ac0]
        at java.dyn.CoroutineSupport.symmetricYieldTo(CoroutineSupport.java:157)
        at java.dyn.Coroutine.yieldTo(Coroutine.java:110)
- Coroutine [0x7ff3d40a4880]
        at java.dyn.CoroutineSupport.symmetricYieldTo(CoroutineSupport.java:157)
        at java.dyn.Coroutine.yieldTo(Coroutine.java:110)
```

Thread Dump

- Yield at possibly blocking points(IO)

Demo for Our Approach Revisited

goal of our approach

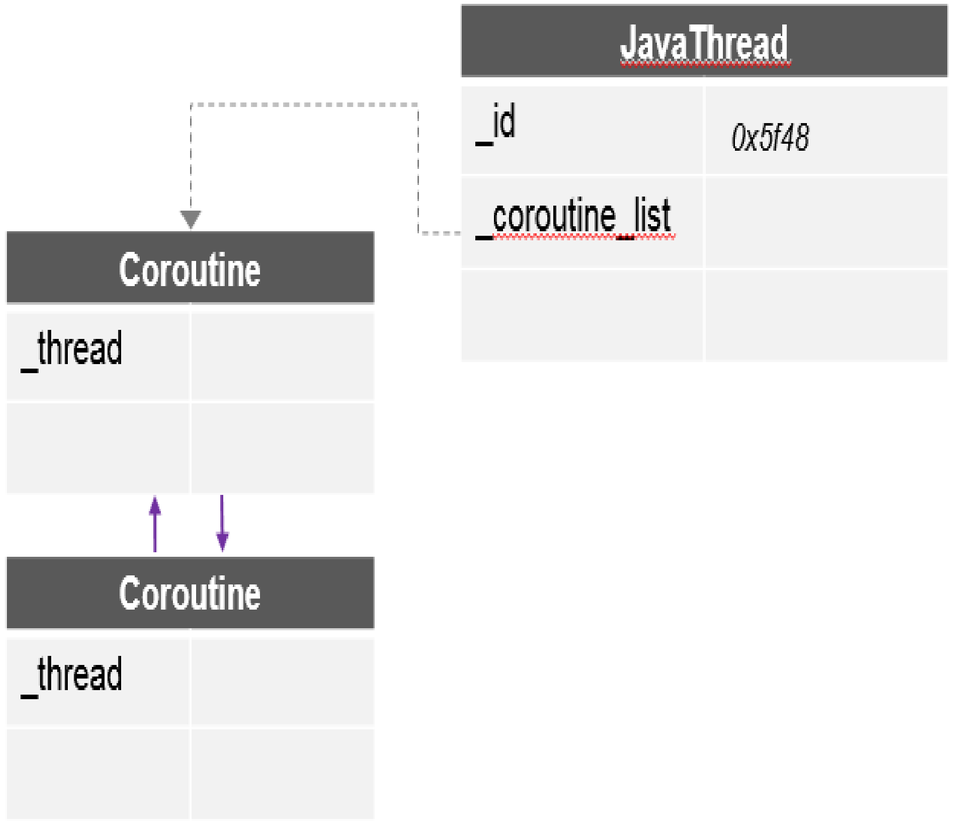
simple synchronous blocking code but with asynchronous performance

Continuation primitive support in HotSpot

- Implemented as part of the MLVM project
- Highlights:
  - Separate stack is allocated per coroutine
  - Support symmetric coroutine
    - yieldTo API is used to transfer control to the next coroutine

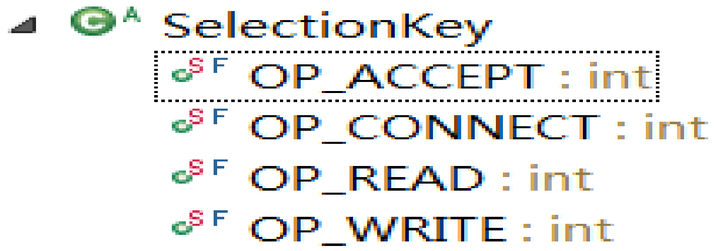
java.dyn
Coroutine
Coroutine()
Coroutine(Runnable)
Coroutine(long)
Coroutine(Runnable, long)
yield() : void
yieldTo(Coroutine) : void

1:1



Java NIO Selector

- Selector: allows only single thread to be used for managing multiple channels
  - Monitoring events for multiple socket channels

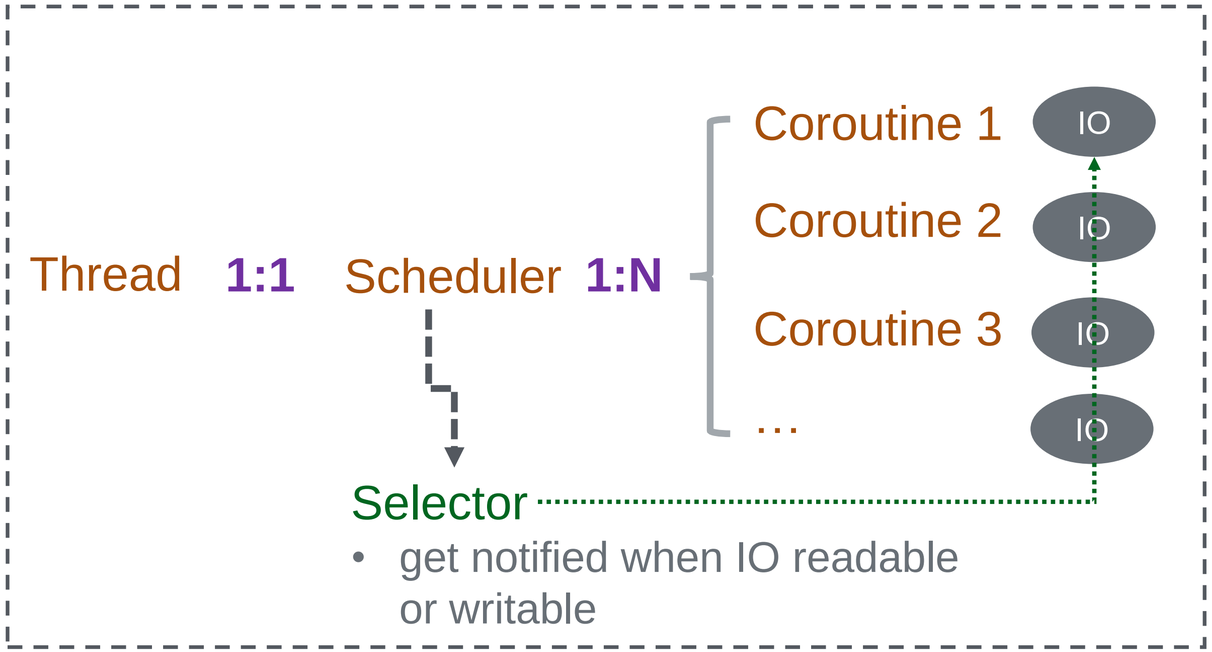


  - Recognizing when one or more become available for data transfer

- Key idea : use Selector mechanism for coroutine scheduling for blocked IO.

How Does It Work: IO Read Demystified

```
17        SocketChannel client = SocketChannel.open(address);
18        while (client.read(buffer) != -1) {
19            buffer.flip();
20            out.write(buffer);
21            buffer.clear();
22        }
```




```
read(ByteBuffer bb) {
   if ((n = ch.read(bb)) != 0)
      return n;
   do {
      engine.registerEvent( ch,OP_READ);
      engine.scheduleNext();
   }  while ((n = ch.read(bb)) == 0);
   return n;
}
```

Register ‘read’ event with scheduler(using selector)

Block current and yield to another available co-routine.

WispEngine : Coroutine Scheduler

- Allocated per thread
- Interested event will be registered with Wisp engine when coroutine gets blocked
  - IO(network r/w)
  - Thread parking(e.g. Thread.sleep)

WispEngine : Coroutine Scheduler (2)

- Schedule next coroutine when:
  - Interested events happen:
    - ✓ IO completed(via Selector)
    - ✓ Timeout(e.g. Thread.sleep)
    - ✓ Thread unparking(e.g. Object.notify)
  - New request arrives (newly submitted Runnable)

WispThreadExecutor API

- WispThreadExecutor, which extends AbstractExecutorService, which allows user to submit tasks into executor, and then run in coroutine

```
 9        ExecutorService wispES = new WispThreadExecutor ();
10        wispES.execute(() -> {
11            //do something
12        });
13        wispES.submit(() -> {
14            //do something
15        }).get();
```

✓ Use like a normal executor service
✓ But run with coroutine

Coroutine and Synchronization

```
8       1) WispEngine.dispatch(test::foo);
9       WispEngine.dispatch(test::bar);
10      }
11      private synchronized void foo() {
12          try {
13      2)      wait();
14          } catch (InterruptedException e) {
15          }
16      }
17      private synchronized void bar() {
18          notify();
19      }
```

coroutine A

coroutine B

1) Schedule and run test::foo in coroutine A
2) Current thread get blocked by wait();
3) No chance to schedule next coroutine B due to the block in 2)

Thread Dump

```
"main" #1 prio=5 os_prio=0 tid=0x00007f5f8fc4f000 nid=0x610e in Object.wait() [0x00007f5f8fe49000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00000006cbc634c0> (a PrimitiveLockTest)
        at java.lang.Object.wait(Object.java:502)
        at PrimitiveLockTest.foo(PrimitiveLockTest.java:11)
        - locked <0x00000006cbc634c0> (a PrimitiveLockTest)
        at PrimitiveLockTest$$Lambda$4/1943105171.run(Unknown Source)
        at com.alibaba.wisp.engine.WispEngine$CacheableCoroutine.run(WispEngine.java:891)
        at java.dyn.CoroutineBase.startInternal(CoroutineBase.java:60)

 - Coroutine [0x7f5f8fca2480]
        at java.dyn.CoroutineSupport.symmetricYieldTo(CoroutineSupport.java:157)
        at java.dyn.Coroutine.yieldTo(Coroutine.java:110)
```

Approach for synchronization (1)

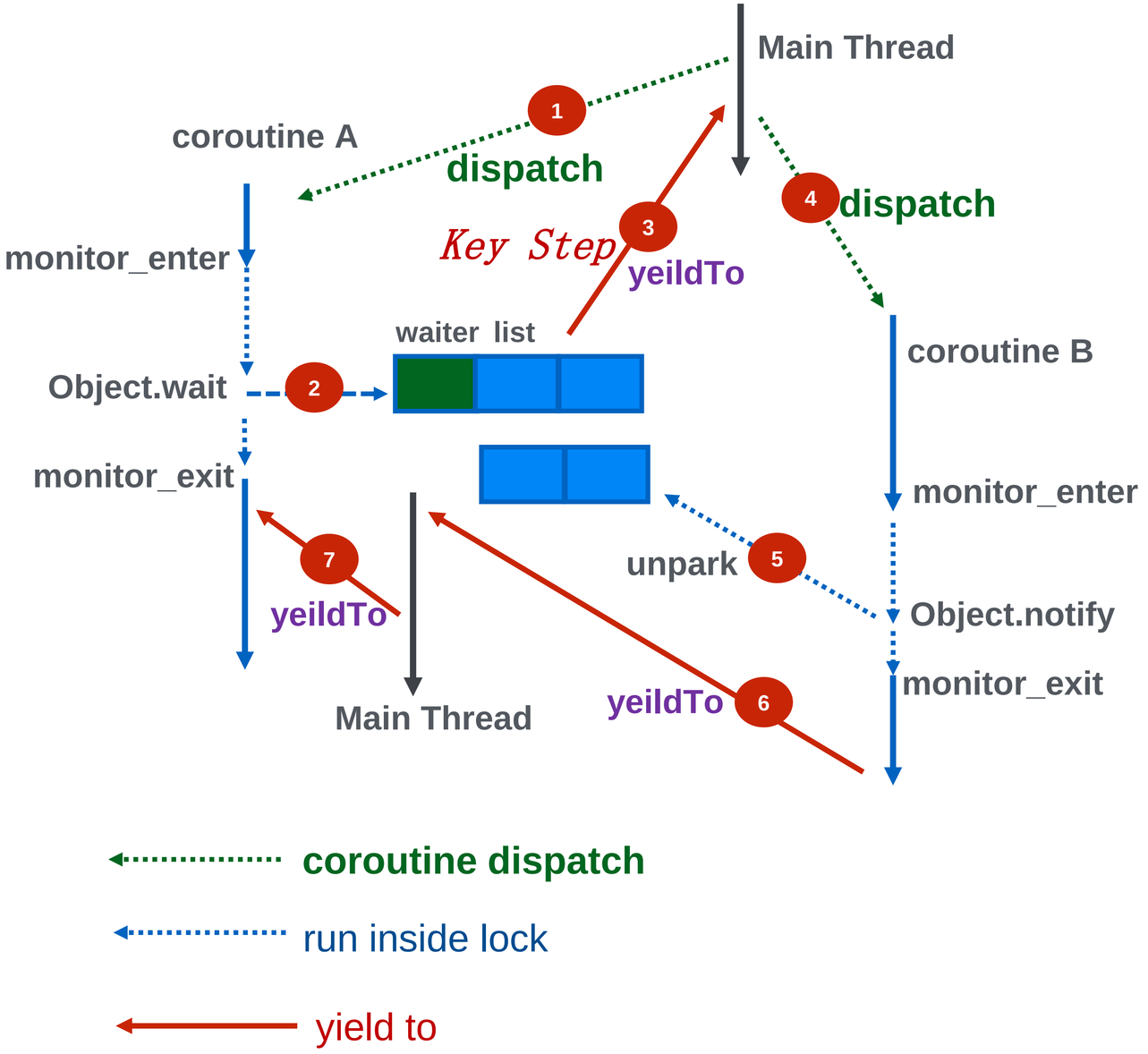
Modify synchronization in HotSpot, support coroutine scheduling at all ‘blocked’ places.

- Fast Lock
  - Determining lock ownership by address on stack, natural support due to the fact: stack is allocated per coroutine.
- Biased lock
  - Use –XX:- UseBiasedLocking to work around

Approach for synchronization (2)

- Inflated Lock(Very complex case)
- Modify monitor enter/exit implementation in HotSpot to support coroutine scheduling

```
8          WispEngine.dispatch(test::foo);
9          WispEngine.dispatch(test::bar);
10     }
11     private synchronized void foo() {
12         try {
13             wait();
14         } catch (InterruptedException e) {
15         }
16     }
17     private synchronized void bar() {
18         notify();
19     }
```



Performance Results

host	10.185.52.173	host	10.185.53.219
idc	su18	idc	su18
cpu_util	60.41%	cpu_util	54.76%
cpu_user	50.37%	cpu_user	47.00%
cpu_sys	8.70%	cpu_sys	6.36%
cpu_iowait	0.00%	cpu_iowait	0.00%
cpu_hardirq	0.00%	cpu_hardirq	0.00%
cpu_softirq	0.00%	cpu_softirq	0.00%
load_load1	13.28	load_load1	9.71
load_load5	15.51	load_load5	10.46
load_load15	15.93	load_load15	10.68
mem_util	13.16%	mem_util	13.76%
mem_used	7.89G	mem_used	8.26G
mem_buff	0.00B	mem_buff	0.00B
mem_cach	8.74G	mem_cach	10.86G
mem_free	43.36G	mem_free	40.89G
mem_total	0.00B	mem_total	0.00B
traffic_bytin	41.01M	traffic_bytin	41.08M
traffic_bytout	16.07M	traffic_bytout	16.25M
traffic_pktin	25309	traffic_pktin	25517
traffic_pktout	25158	traffic_pktout	25254
traffic_pkterr	0	traffic_pkterr	0
traffic_pktdrp	0	traffic_pktdrp	0
jvm_ygc	54	jvm_ygc	54
jvm_ygc_time	53ms	jvm_ygc_time	21ms
jvm_fgc	0	jvm_fgc	0
jvm_fgc_time	0ms	jvm_fgc_time	0ms

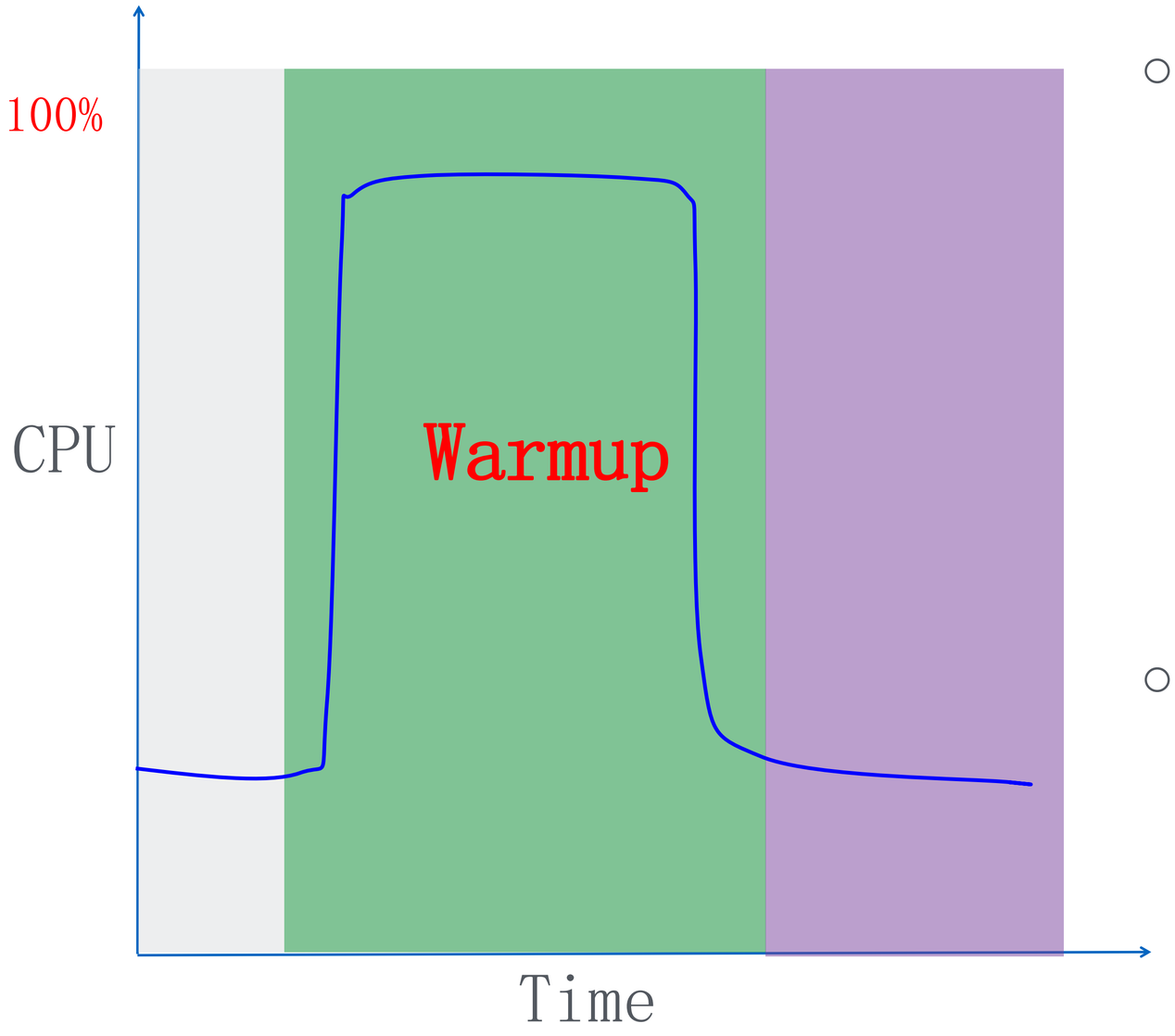
- Used in real world Alibaba application, **Carts**, which is an e-commerce application for shopping carts.
- Coroutine is used to handle all network IO requests, running each thread on each logical processor.
- Reduced CPU usage from 60% to 54% (~10% saving)while serving the same number of requests.

Tips: Use coroutine library to boost performance if high thread context switch cost was observed

#3 JWarmup

priming online application for speed

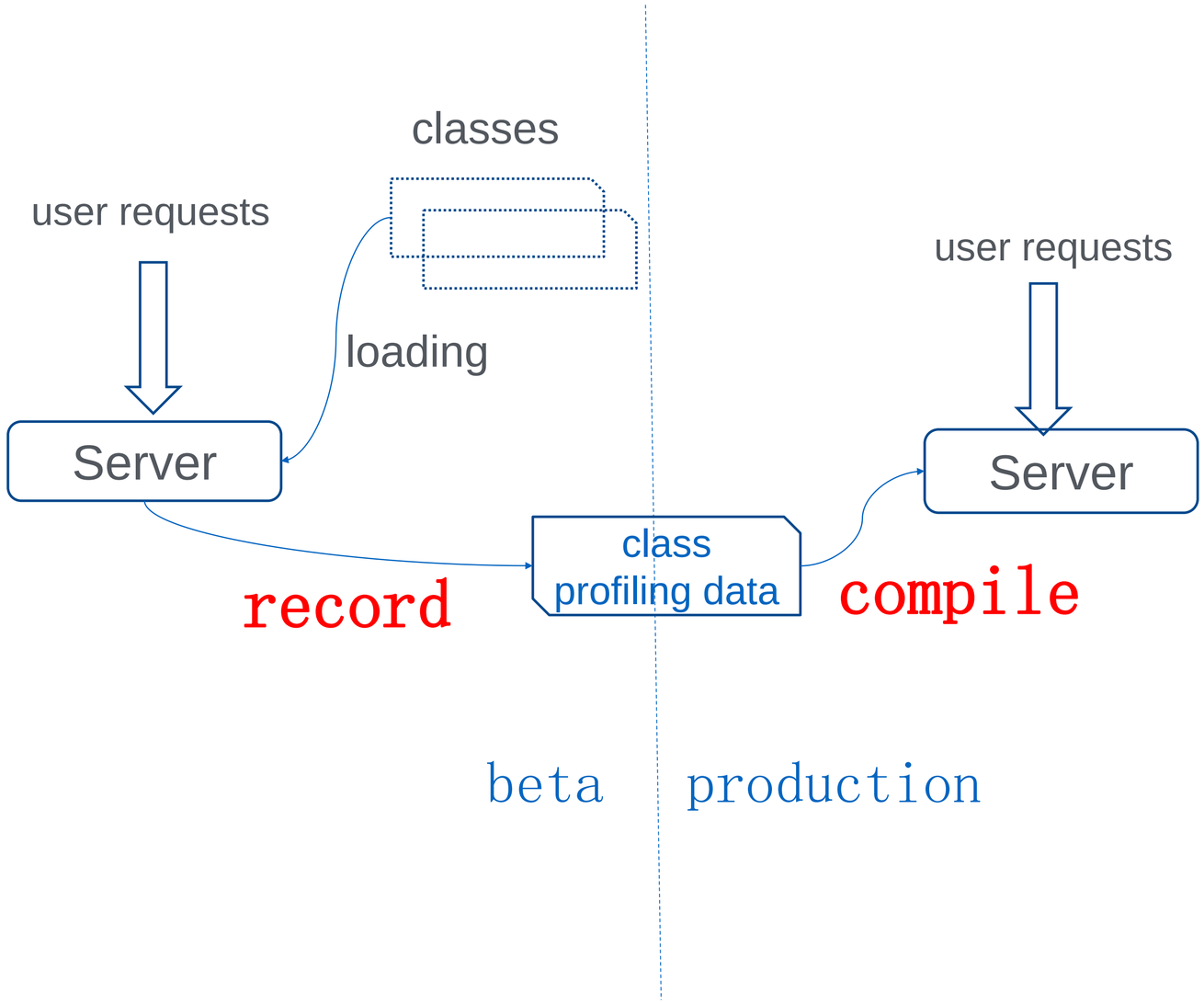
Java Warmup Issue



- High CPU consumption during warmup
  - Complex application with **20,000+** classes loading and **50,000+** methods compilation
  - TieredCompilation can alleviate it, but can not completely resolve it.
- Problems occurred when lots of user requests come in after application online
  - Much longer response time , hence lots of ‘timeout’ errors.

JWarmup: eliminate JVM warm-up

- Before JWarmup, our application owners usually use ‘mock’ data to warm-up JVM to let JIT optimize before actual requests come in.
- ‘JWarmup’ used to obviate the need for "warming-up" by:
  - **Record** the profiling data (in beta testing)
  - Let JIT **compile** code based on recorded profiling data before requests come in (in online run)



‘Under the hook view 'of JWarmup

- In recording phase
  - Record class initialization info
  - Record method compilation info
  - Flush them onto disk as record file after enough time
- In compiling phase
  - Eagerly load classes recorded in previous run
  - Eagerly initialize loaded classes
    - Tricky case: <clinit> MUST be executed in determined order
      - Initializing them only through constant pool might be problem
  - Submit for compilation

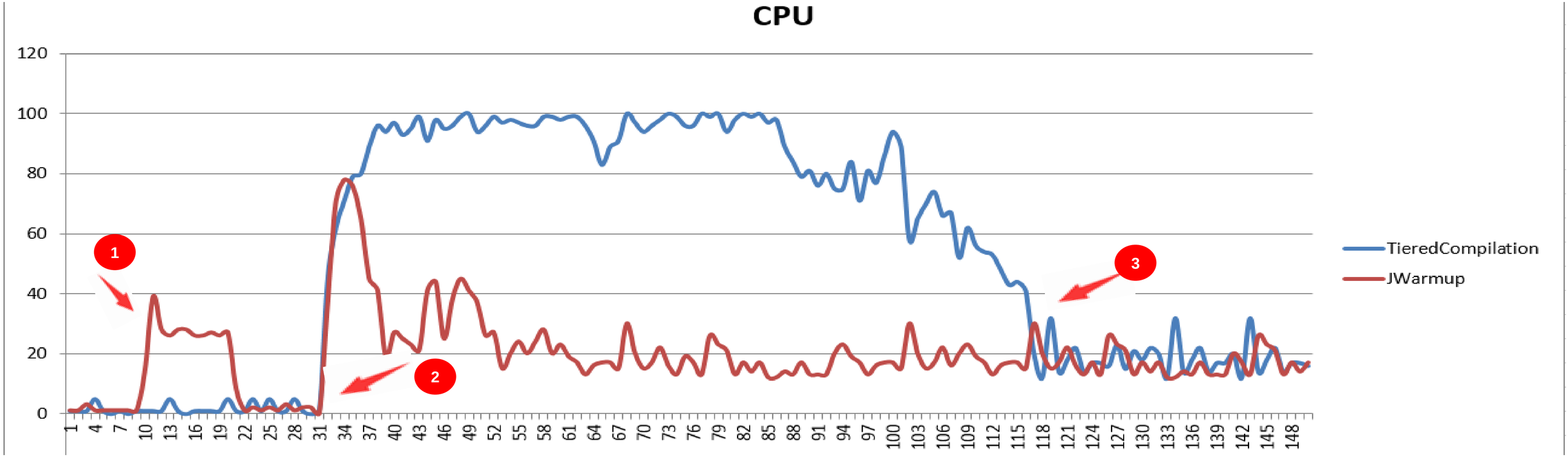
```
2 public class Foo {
3     public static int count = 0;
4     public void test() {
5         count++;
6         Bar.test();
7     }
```

dependent

```
2 public class Bar {
3     public static int count = 0;
4     static {
5         count = Foo.count;
6         count ++;
7     }
```

Bar.<clinit> is dependent on the execution of Foo.test();

Results from Alibaba Online Application(UMP)



(1) Warmup compiles code in advance before requests come in

(2) At the point when real user requests come in.

(3) Most of hot methods are completely compiled by TieredCompilation

Other considerations

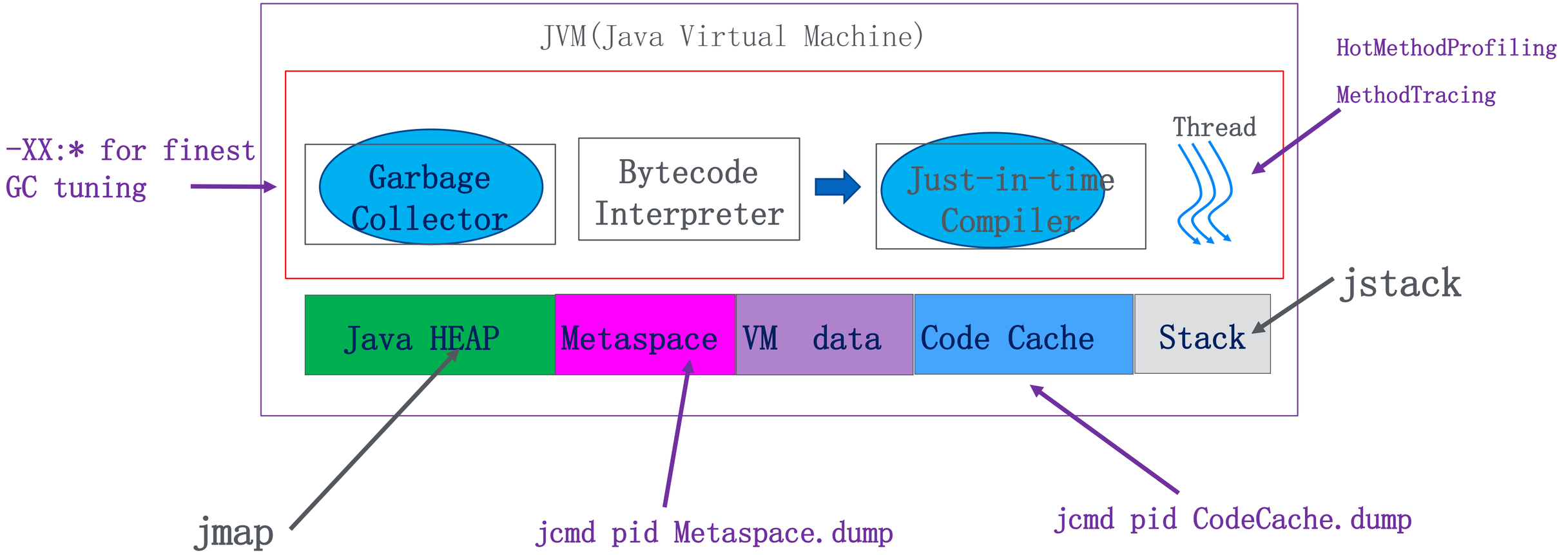
- Don’t record dynamically generated classes
  - Generated by groovy, classes themselves might be changed between runs due to the change of rules.
  - Generated by Java Reflection/Proxy because class names get changed in next run.
- Disable ‘Null check elimination’ optimization for avoiding unexpected de-optimization.

- Not compatible with –XX:+ TieredCompilation, consider it in next step.

#4 Diagnostics and Troubleshooting

technical support for java developers

Modify to build-in more profiling capacity

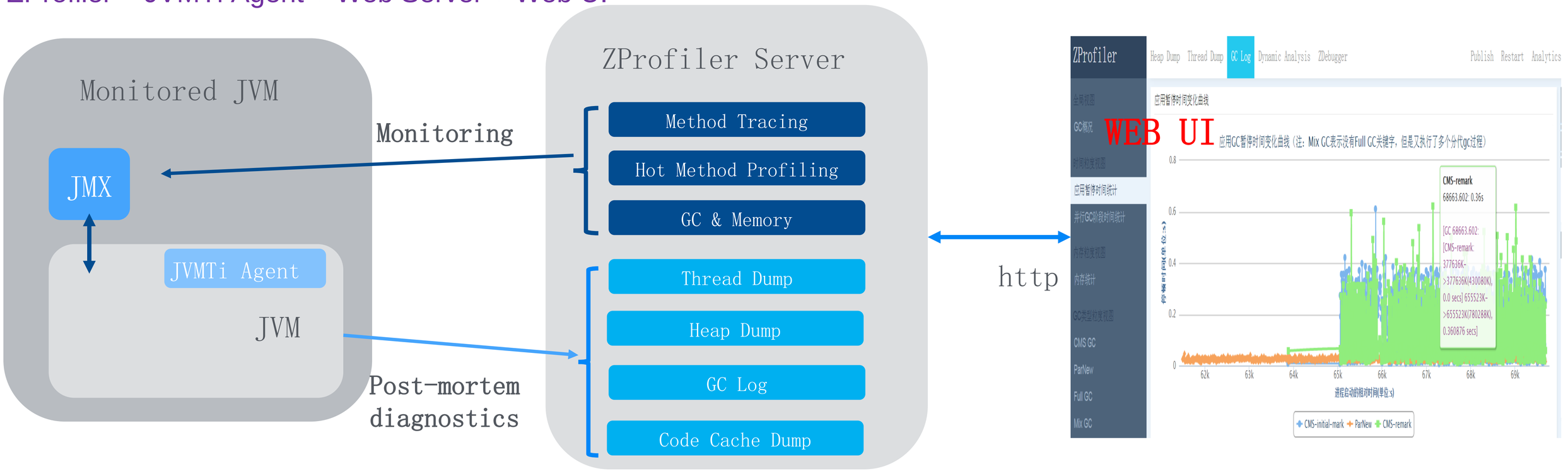


HotMethodProfiling : sampling based, use undocumented AsyncGetCallTrace API
MethodTracing: capture every method enter/exit per thread at compiled code level

https://github.com/jvm-profiling-tools/honest-profiler

ZProfiler: tools for profiling and diagnostics

ZProfiler = JVMTi Agent + Web Server + Web UI



- ZProfiler server is deployed in datacenter directly, developer use **browser** to diagnose the java applications on any machine.
  - Dump files could be copied into ZProfiler server for further analysis.
  - Could monitor the runtime behavior of target JVM directly via JMX connection.

Recap

- What we have mainly covered in this talk
  - Multi-tenant JVM & GICH
  - Wisp coroutine
  - JWarump
  - ZProfiler
- Contribution to OpenJDK
  - Two OpenJDK committers so far
  - Would like to contribute back improvements made in AJDK to OpenJDK

Alibaba Group 阿里巴巴集团

谢谢观看

thanks

Mail: sanhong.lsh@alibaba-inc.com

Twttier: @sanhong_li