init class

new
⇓
check if class exists in
constant pool
if not load class
⇓
JVM allocated memory for obj
from Heap
⇓
1. Bump the pointer } Serial, ParNew ← compact
2. Free list {CMS ← mark-swap

to avoid
thread safety issue
1. CAS → Atomic operation
2. TLAB → in Heap
⇓
init assigned memory to 0 (obj ~~header~~ not header)
⇓
setup obj Header

⇓

< init > executed.

---

obj Memory Structure.

{ Header
Instance Data
Padding.

obj clean.

init address
must be 8bits. × X

{ Mark Word.
class ref
Array length

{ hash code.
GC age
ref to lock
thread id.
time.

decide class type

if obj is Array

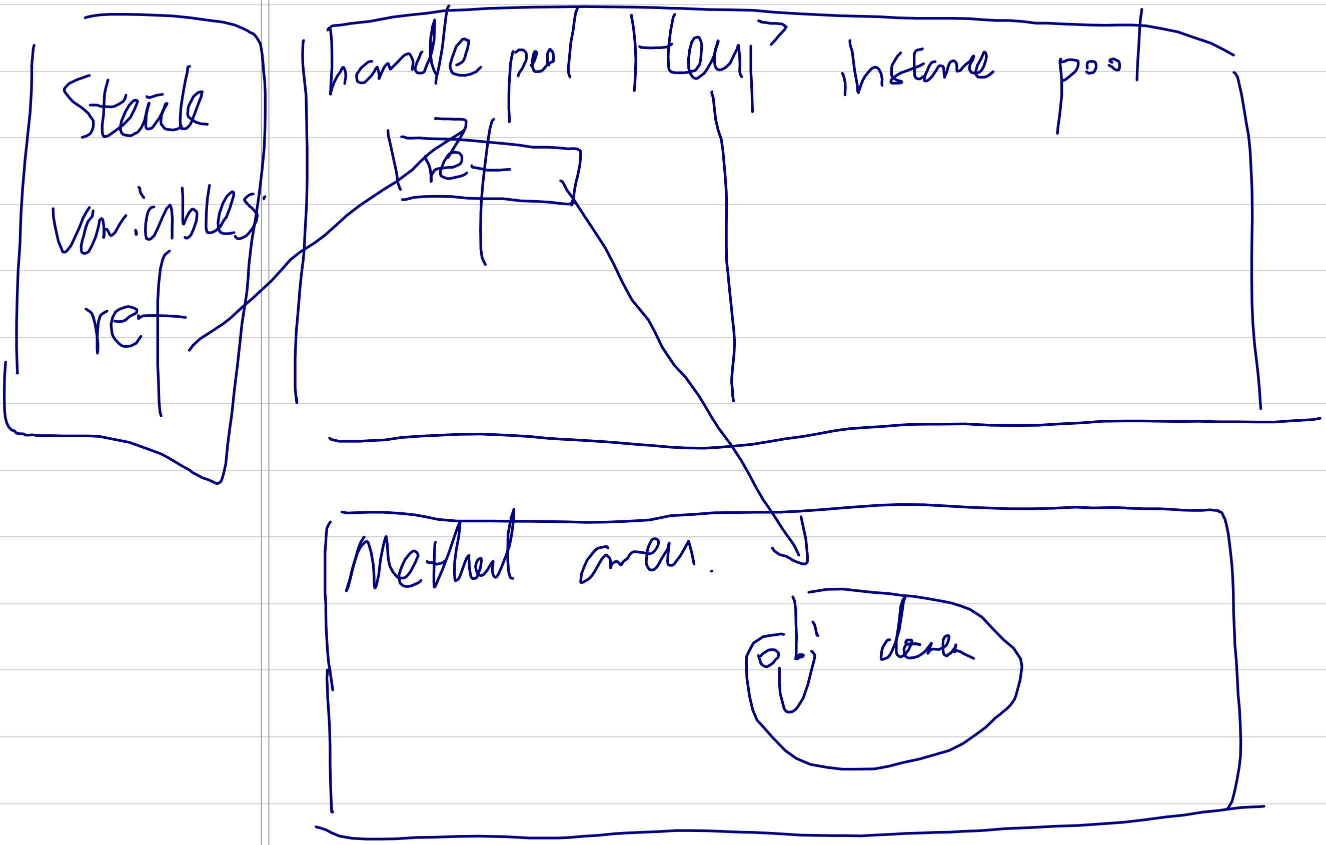
---

reference visit

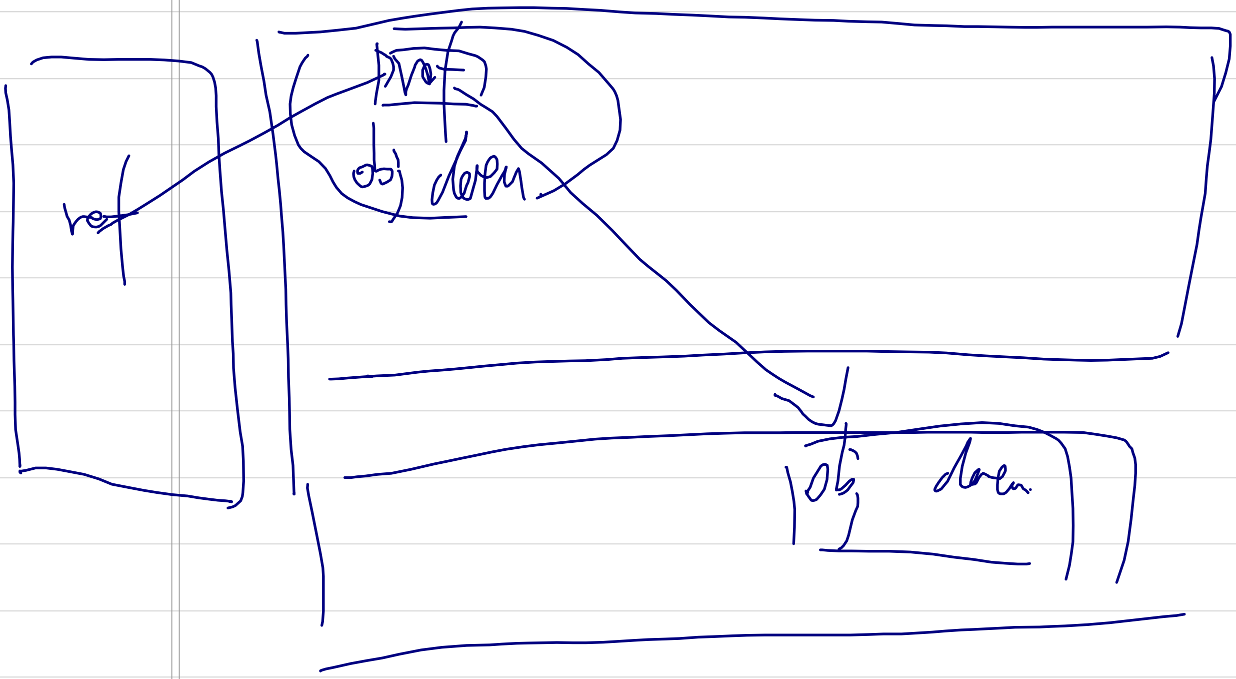
1, use handler

2. directly use ref

Handle.



Ref Directly.



Out of Memory Error

Java Heap

└ OutOfMemory {
① memory leak
② memory overflow

Jvm stack

└ StackOverflow ⇐ { usually this happen 单线程

└ OutOfMemory { 不断 build thread

if sys memory is 2GB

2GB − Xms (Heap) − MaxPermSize

计数器消耗少可以忽略

剩下的内存由 stack 分

thread space 越大 thread 越少

内存容易耗尽

Method Area

└ Constant pool { CGlib.

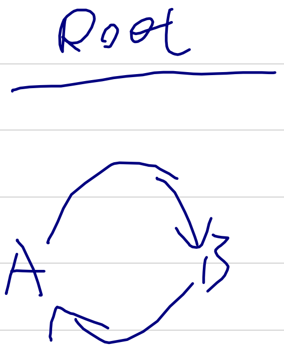
Direct Memory.

└ if NIO used?

Heap Dump file size small

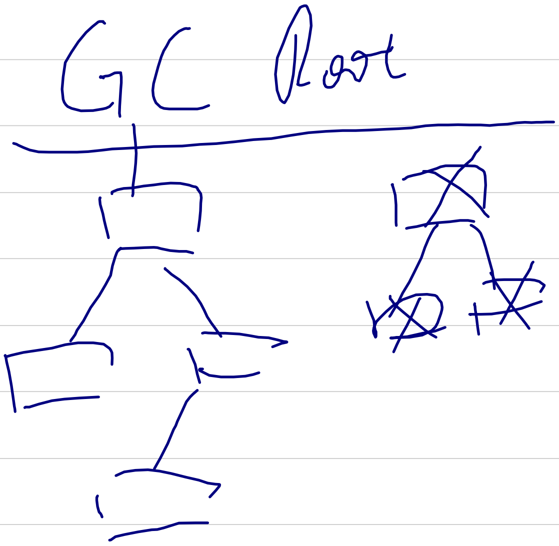
GC

Reference counting → cannot solve



↑ JVM not use

Reachability Analysis



GC ROOT: JVM Stack ref obj
Method Area static ref obj
constant ref obj
Java Native ref obj

Reference type:

Strong Reference: Object obj = new Object()
never GC if ref exists

Soft Reference: useful but not essential
if OOM will occur, SR will be GC
if memory still small, then OOM

Weak Reference: if WR, the obj will alive
until next GC

Phantom Reference: if GC, get a sys
notification

GC life cycle:
if not linked with GC Root
mark first time
if need to finalize()
if not overrided by finalize
or finalize has been called

return false
if need to finalise
↳ F-Queue

avoid dead lock
execute: trigger but not wait
↑ Finalizer Thread

← can relink with Root during finalise( )

GC ↗ F-Queue
mark second time
if still not linked
GC

GC

Metaspace:
GC: constant
class: 1. no object
2. no class loader
3. no java.lang.Class ref

GC algorithms

1、Mark — Swap

step 1: mark all GC object.

2: Swap marked obj

Cons:

1、not effentient

2、space, memory is not together

may be not good

for large obj → another GC

2、Copying

sperate space into 2

Only use half of it

If one a- half space is full,

Copy alive objs to another half

Then clean the original half

Pros:
  1. easy to implement
  2. efficient

Cons: use half space

it's been used for new generation
8:1:1 Eden : Survivor : Survivor
use Eden and one Survivor
only use 90%
If space of survivor is OOM,
 we use old generation for handle
Promotion
if alive rate is high, Copy is not good

Mark-Compact:
 step 1: mark
   2: move everything to one side
    clean the rest of space

Generation Collection:
  new G: copy
  old G: mark-sweep mark-compact

HotSpot: — solution:
Oop records references

SafePoint
- Preemptive Suspension → not used → stop all threads
- Voluntary Suspension →

set a mark
threads will poll the mark
if true, suspened

Cons: cannot solve sleep or blocked threads

⇓

Safe Region

GC

Serial collector:
new G single thread.
stop others.

ParNew collector:
multi-thread

can work with CMS

Parallel: multi-thread $\overset{GC}{\vee}$ working together
but customer thread still waiting

Concurrent: customer & GC threads work
at the same time
GC thread runs on another CPU

Parallel Scavenge:

new G copying

Throughput First

Serial old:
old G single mark-swap
as a back up of CMS

Parallel Old:
old G mark- compact
Throughput First

CMS :
concurrent mark sweep
step 1: initial mark
2: concurrent mark
3: remark
4: concurrent sweep

Cons:
1. CPU usage high requirement
CPU > 4
2. cannot solve Floating Garbage
lead to Concurrent Mode Failure

3. mark-sweep space compact required

G1: Garbage-First
best for now

Pros:
1. concurrent use multi CPU core
2. Generation collect
3. space compact
4. predicted stop

use Remembered set → Region.
↓
reference

GC Log

Full GC → stop the world.
ParNew
DefNew
Tenured } → GC where it happened.
Perm

new G1 [3324k → 152k (XXX k)]
area usage before GC → area usage after GC
↓ second number data
heap usage before → after

Object Memory allocate

1、In Eden
   if not enough space, minor GC

2、Big Object goes into Old generation

3、Long living object into old generation
   Age 15 → old.

4、Dynamic Object Age Judge:
   if the size of all same age obj's is larger than half space of Survivor space
   All objs' age >= ↑ age will come into old generation

5、Handle Promotion

Chapter4 JVM monitor and tools

jps : JVM Process Status Tool

jstat: JVM Statistics Monitoring Tool

jmap → heapdump

jhat → get dump analysis

jstack

VisualVM : all in one tool

JVM Child Process

6、 Class : 8 bits basis

Magic Number : first 4 bits

Version : second 4 bit

constant pool

access flags : class or Interface public?

this class :

super class :

field info :

7 Class loader

$$\begin{tikzcd}
\text{Loading} \arrow[d, Rightarrow] & & \\
\text{verification} \arrow[d, Rightarrow] & & \\
\text{Preparation} \arrow[d, Rightarrow] & \text{Linking} & \\
\text{Resolution} \arrow[d, Rightarrow] & & \\
\text{Initialization} \arrow[r, Rightarrow] & \text{using} \arrow[r, Rightarrow] & \text{unloading}
\end{tikzcd}$$

Loading:

(1) new, read or set static attribute
or call class static method.

(2) java.lang.reflect

(3) if parent class not loaded

4) main()

5) java.lang.invoke.MethodHandle

Parents Delegation Model

$$\begin{tikzcd}
 & \text{Bootstrap CL} & \\
 & \text{Extension CL} \arrow[u, Rightarrow] & \\
 & \text{Application CL} \arrow[u, Rightarrow] & \\
\text{UserCL} \arrow[ur, Rightarrow] & & \text{User CL} \arrow[ul, Rightarrow]
\end{tikzcd}$$

java.lang.ClassLoader

---

12 Memory Model & Threads.