

<http://mashibing.com>

JUC: `java.util.concurrent`

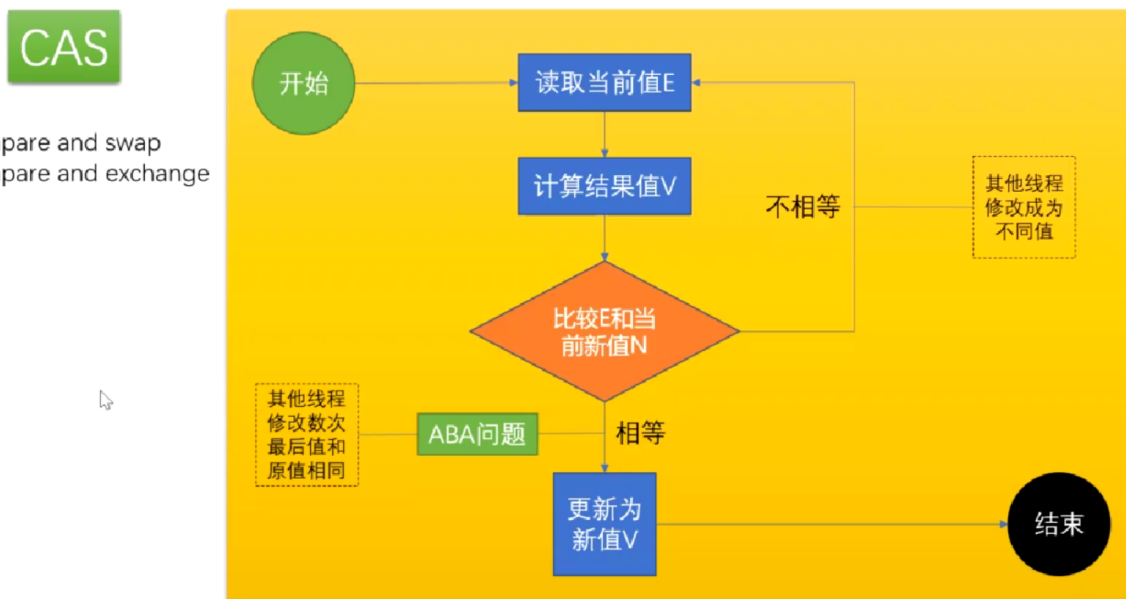
CAS: compare and swap

JNI: java native interface

JIT: just-in-time compilation 即时编译

identityHashCode: 一致性的hashcode

1. CAS



CAS是乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

这里再强调一下，乐观锁是一种思想。CAS是这种思想的一种实现方式。

JAVA对CAS的支持：

在JDK1.5 中新增 `java.util.concurrent` (J. U. C) 就是建立在CAS之上的。相对于对于 `synchronized` 这种阻塞算法，CAS是非阻塞算法的一种常见实现。所以J. U. C在性能上有了很大的提升。

2. ABA问题

解决：当前的值加版本号

或者JDK中加boolean类型:从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题

3. CAS的底层实现

JVM是一个标准

hotspot--oracle

J9--IBM

taobaoVM

openJDK--开源

Jrocket--BEA---oracle

以 java.util.concurrent 中的 AtomicInteger 为例，看一下在不使用锁的情况下是如何保证线程安全的。主要理解 incrementAndGet方法，该方法的作用相当于 ++i 操作。

```
AtomicInteger i = new AtomicInteger();
```

自旋锁、无锁是怎么实现的呢

```
i.incrementAndGet();
```

```
/**
 * Atomically increments by one the current value.
 *
 * @return the updated value
 */
public final int incrementAndGet() {
    return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}
```

在没有锁的机制下, 字段value要借助volatile原语, 保证线程间的数据是可见性。这样在获取变量的值的时候才能直接读取。然后来看看 ++i 是怎么做到的。

incrementAndGet采用了CAS操作, 每次从内存中读取数据然后将此数据和 +1 后的结果进行CAS操作, 如果成功就返回结果, 否则重试直到成功为止。

而 compareAndSwapInt利用JNI (Java Native Interface) 来完成CPU指令的操作:

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
```

```

        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}

```

CAS原理：

CAS通过调用JNI的代码实现的。而compareAndSwapInt就是借助C来调用CPU底层指令实现的。

下面从分析比较常用的CPU（intel x86）来解释CAS的实现原理。

下面是sun.misc.Unsafe类的compareAndSwapInt()方法的源代码：

```

public final native boolean compareAndSwapInt(
    Object var1, long var2, int var4, int var5);

```

compareAndSwapInt这个本地方法在JDK中依次调用的C++代码为

```

#define LOCK_IF_MP(mp) __asm cmp mp, 0 \
    __asm je L0 \
    __asm _emit 0xF0 \
    __asm L0:

inline jint Atomic::cmpxchg (jint exchange_value, volatile jint* dest, jint
compare_value) {
    // alternative for InterlockedCompareExchange
    int mp = os::is_MP();
    __asm {
        mov edx, dest
        mov ecx, exchange_value
        mov eax, compare_value
        LOCK_IF_MP(mp)
        cmpxchg dword ptr [edx], ecx
    }
}

```

如上面源代码所示，程序会根据当前处理器的类型来决定是否为cmpxchg指令添加lock前缀。如果程序是在多处理器上运行，就为cmpxchg指令加上lock前缀（lock cmpxchg）。反之，如果程序是在单处理器上运行，就省略lock前缀（单处理器自身会维护单处理器内的顺序一致性，不需要lock前缀提供的内存屏障效果）。

_asm 汇编语言

LOCK_IF_MP

一个cpu, 直接执行cmpxchg指令

多个cpu, 执行lock cmpxchg指令

cmpxchg汇编指令

lock cmpxchg指令:

cmpxchg指令, 没有原子性, 不能保证读和写之间不能被其他cpu改写

lock指令, 保证当前cpu对当前值修改的时候, 其他cpu不能进行修改, 保证了原子性

4. Object o = new Object() 在内存中占了多少字节? - 顺丰

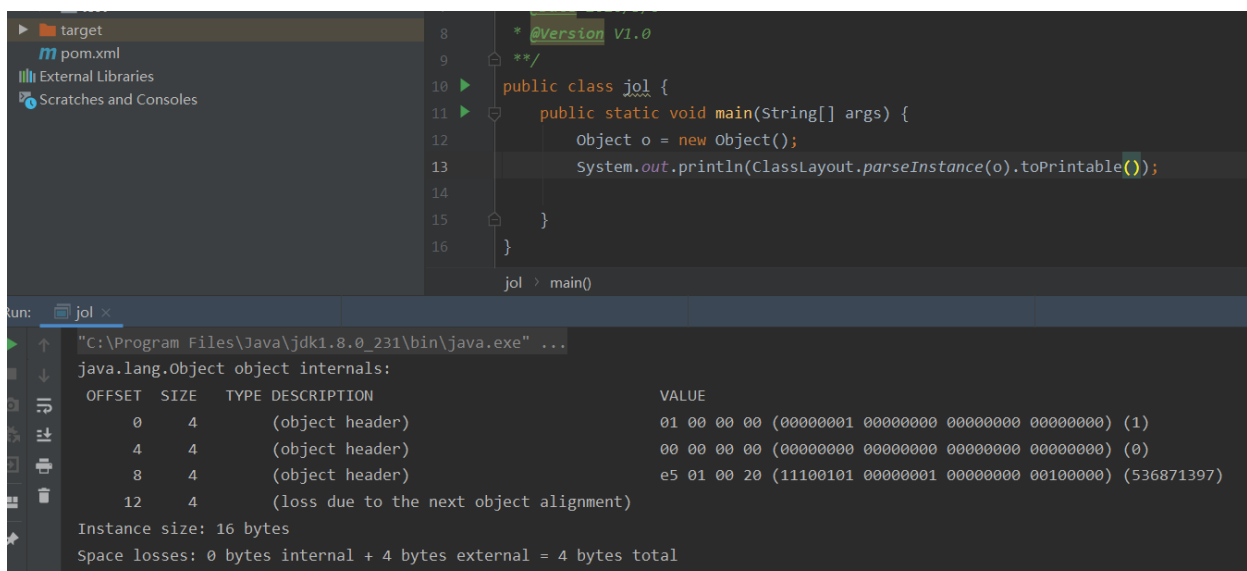
1.64位的机器, 因为hotspot默认开启了oops(useCompressOops)压缩, 对象头(markword)为8个字节+class pointer类型指针 (4个字节) + instance data实例数据 (0个字节) +padding补齐(4个字节)=16个字节

如果没有开始oops, 对象头(markword)为8个字节+class pointer类型指针 (8个字节) + instance data实例数据 (0个字节) +padding补齐(0个字节)=16个字节

openJDK提供的工具: java object layout工具

[mavan版本信息](#), 我使用的是最新的0.9

```
<dependency>
  <groupId>org.openjdk.jol</groupId>
  <artifactId>jol-core</artifactId>
  <version>0.9</version>
</dependency>
```



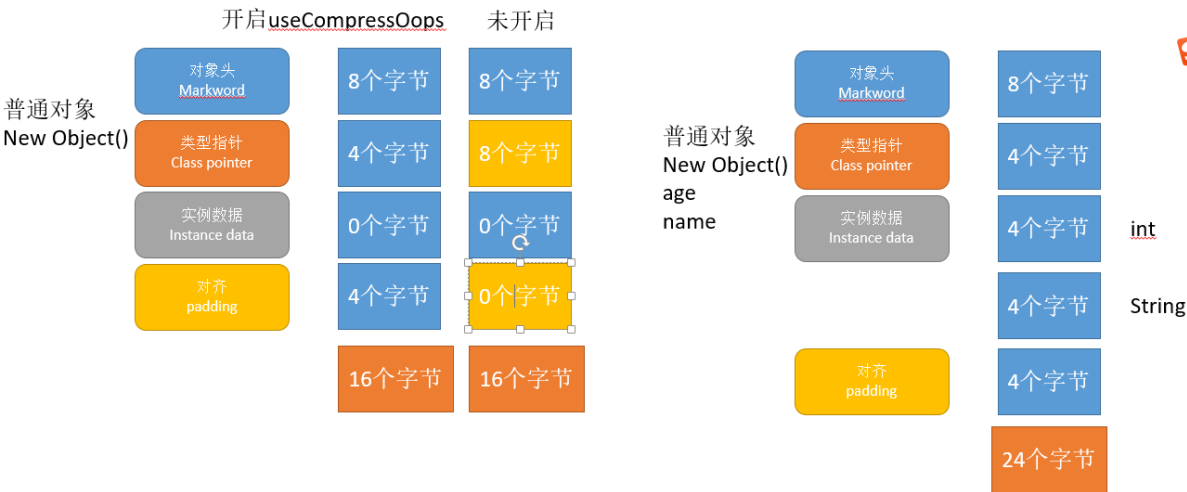
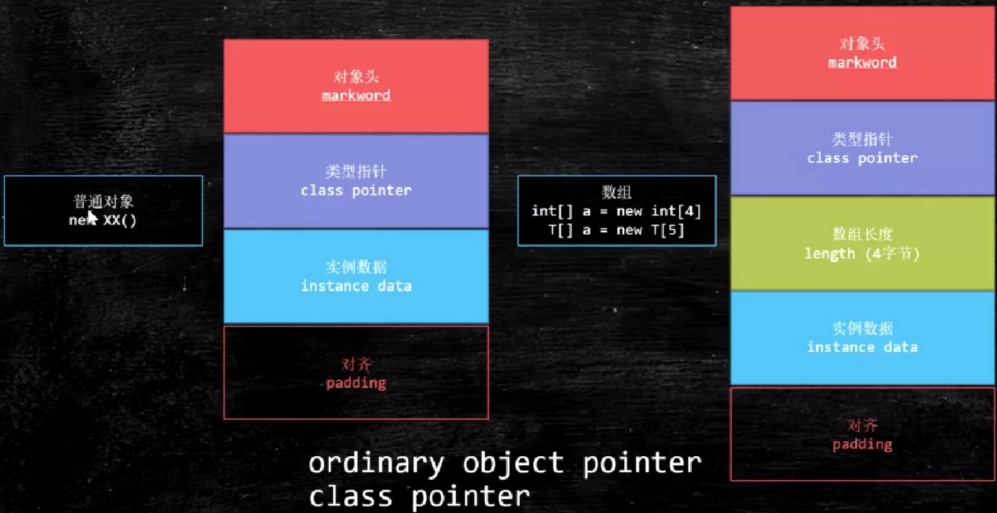
```
8  * @Version V1.0
9  **/
10 public class jol {
11     public static void main(String[] args) {
12         Object o = new Object();
13         System.out.println(ClassLayout.parseInstance(o).toPrintable());
14     }
15 }
16 }

jol > main()

"C:\Program Files\Java\jdk1.8.0_231\bin\java.exe" ...
java.lang.Object object internals:
OFFSET  SIZE  TYPE DESCRIPTION                               VALUE
0       4      (object header)                               01 00 00 00 (00000001 00000000 00000000 00000000) (1)
4       4      (object header)                               00 00 00 00 (00000000 00000000 00000000 00000000) (0)
8       4      (object header)                               e5 01 00 20 (11100101 00000001 00000000 00100000) (536871397)
12      4      (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

object Header 对象头

3:对象在内存中的存储布局



```
java
├── CasCode
├── Jol
├── resources
│   ├── test
│   └── target
├── pom.xml
├── External Libraries
└── Scratches and Consoles

Jol o = new Jol();
o.setAge(1);
o.setName("test");
System.out.println(ClassLayout.parseInstance(o).toPrintable());
}
```

Jol object internals:

OFFSET	SIZE	TYPE DESCRIPTION	VALUE
0	4	(object header)	01 00 00 00 (00000001 00000000 00000000 00000000) (1)
4	4	(object header)	00 00 00 00 (00000000 00000000 00000000 00000000) (0)
8	4	(object header)	05 c1 00 20 (00000101 11000001 00000000 00100000) (536920325)
12	4	int Jol.age	1
16	4	java.lang.String Jol.name	(object)
20	4	(loss due to the next object alignment)	

Instance size: 24 bytes

Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

java -XX:+PrintCommandLineFlags -version指令查看JVM打印出那些已经被用户或者JVM设置过的详细的XX参数的名称和值，默认开启了Ops

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.17134.885]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\limiao>java -XX:+PrintCommandLineFlags -version
-XX:InitialHeapSize=132272960 -XX:MaxHeapSize=2116367360 -XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
java version "1.8.0_231"
Java(TM) SE Runtime Environment (build 1.8.0_231-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.231-b11, mixed mode)

C:\Users\limiao>
```

-XX:+UseCompressedOops压缩

一个指针的长度为64位，即8个字节，开启useCompressedOops后，压缩为4个字节

reference类型（类型指针）在32位JVM下占用4个字节，但是在64位下可能占用4个字节或8个字节，这取决于是否启用了64位JVM的指针压缩参数UseCompressedOops

5. synchronize的底层原理

锁的信息在markword对象头中，

synchronize(o) {

...

}

上锁的位置在synchronize(o)，而不是synchronize内的代码块；举例是上厕所，是把门锁了，而不是把马桶锁了，触发的对象为o对象

6. 锁升级过程

new对象---->偏向锁--->轻量级锁（自旋锁【CAS技术】、无锁、自适应锁）---->重量级锁

Hotspot的实现

锁状态	25位	31位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
无锁态 (new)	unused	hashCode (如果有调用)	unused	分代年龄	0	0	1

锁状态	54位	2位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
偏向锁	当前线程指针 <code>JavaThread*</code>	Epoch	unused	分代年龄	1	0	1

锁状态	62位	2bit 锁标志位	
轻量级锁 自旋锁 无锁	指向线程栈中Lock Record的指针	0	0
重量级锁	指向互斥量（重量级锁）的指针	1	0
GC标记信息	CMS过程用到的标记信息	1	1

markword对象头为64位，上图就是这64位在锁升级过程的变化

markword记录了锁信息+GC信息+hashCode

identityHashCode：一致性的hashCode

hashCode可以重复

偏向锁升级轻量级锁的过程中，分代年龄和hashCode会备份在线程栈中，通过LR指针（local Record）又保存在markword中

JDK1.0, 1.2中synchronize为重量级锁

=====

【1】new对象进入无锁态

1. 偏向锁位：0 锁标志位：0, 1

2. 4bit的分代年龄

分代年龄：每gc一次，加1

在做jvm优化的时候，把分代年龄值改大，由15改为31，是没有效果的，因为分代年代是4位，最大值就是15，所以设置为31是不起作用的

3. 31位的hashCode

【2】来一个线程，升级为偏向锁

1. 偏向锁位：1，锁标志位：0, 1

2. 4bit的分代年龄

3. 54位的当前线程指针`JavaThread*`

【3】再来一个线程，升级为轻量级锁

1. 锁标志位：0, 0
2. 62位的指向当前线程栈中的Lock Record (LR) 的指针

使用CAS来抢锁

【4】当一个线程请求资源超过10次，或者cpu线程使用超过1/2，升级为重量级锁
原先可以一个线程请求资源超过的多少次升级为重量级锁，其实没必要

1. 锁标志位：1, 0
2. 62位的指向互斥量（重量级锁）的指针

3. 用户态：3，大部分应用程序

内核态：锁通过内核来执行

4. 重量级锁下面有队列，一个线程占用资源的时候，其他线程是wait状态（wait状态并不消耗cpu）

7. 锁降级

锁降级只会发生在特殊情况下，GC情况下会发生
一旦进行GC，再做锁降级，其实没什么作用

8. 锁消除

```
Object o = new Object();
synchronized (o){
    StringBuffer sb = new StringBuffer();
    sb.append("a").append("b");
}
```

9. 锁粗化

10. Synchronize的底层原理

字节码：monitorenter和monitorexit

汇编指令：lock cmpxchg

解锁检测VM参数??

优化: C1

C2

11. Voliate的底层原理

Voliate保证了线程之间的可见性, 禁止指令重排

12. CPU三层缓存

cpu的执行速度-----100倍 内存-----100百万的硬盘