

Summary of Common Java Concurrency Interview Questions (Part 2)

👤 [Guide](#) 📄 Java 🔑 Java Concurrency 🕒 About 11,469 words ⏱ About 38 minutes

JavaGuide官方知识星球
(限时优惠) 📄

专属面试小册 / 一对一提问 / 简历修改
专属求职指南 / 不定时福利 / 学习打卡

— 点击图片即可详细了解 —

★JMM (Java Memory Model)

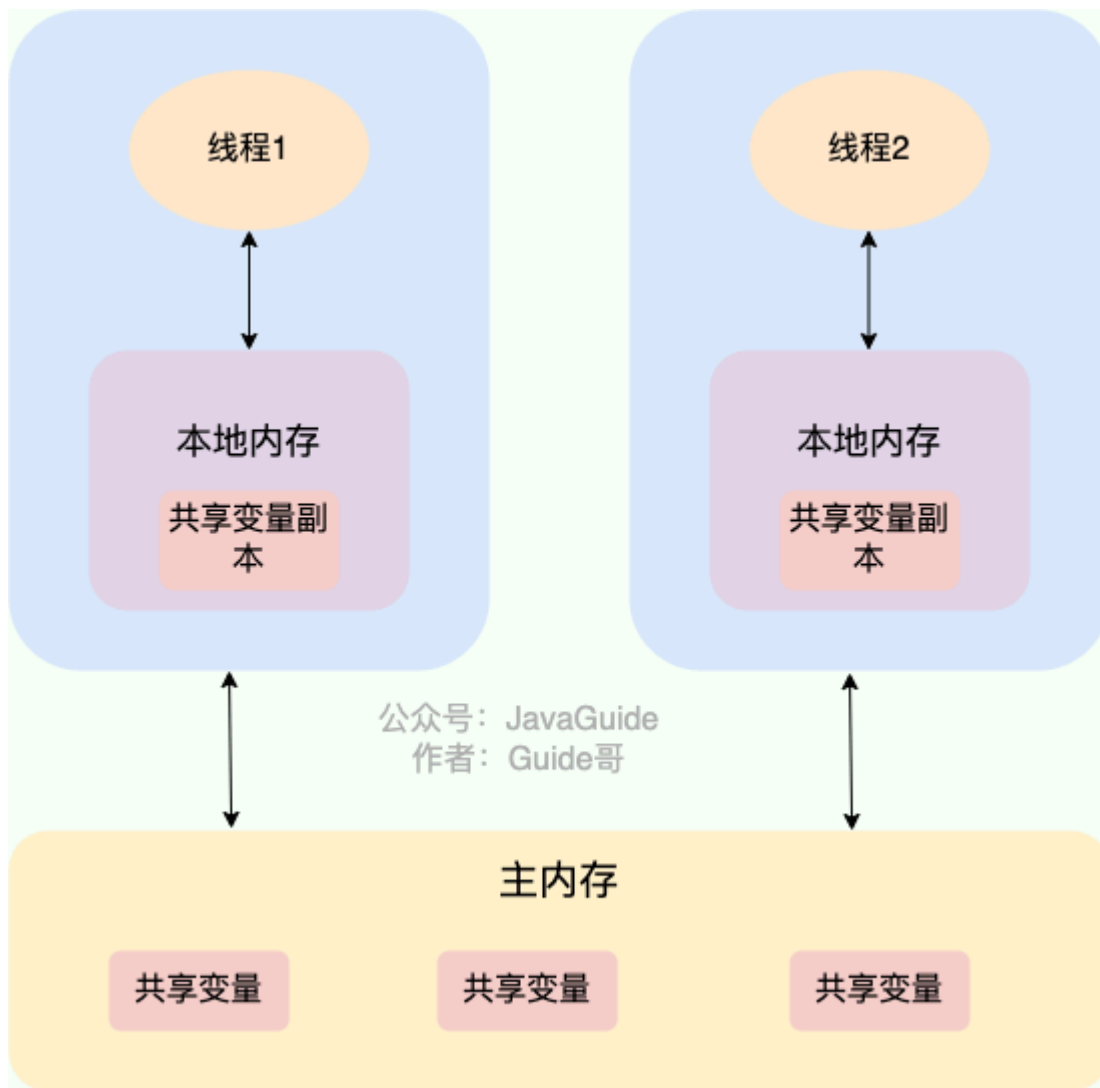
There are many issues related to JMM (Java Memory Model), and they are also quite important, so I have written a separate article to summarize the knowledge points and issues related to JMM: [Detailed Explanation of JMM \(Java Memory Model\)](#).

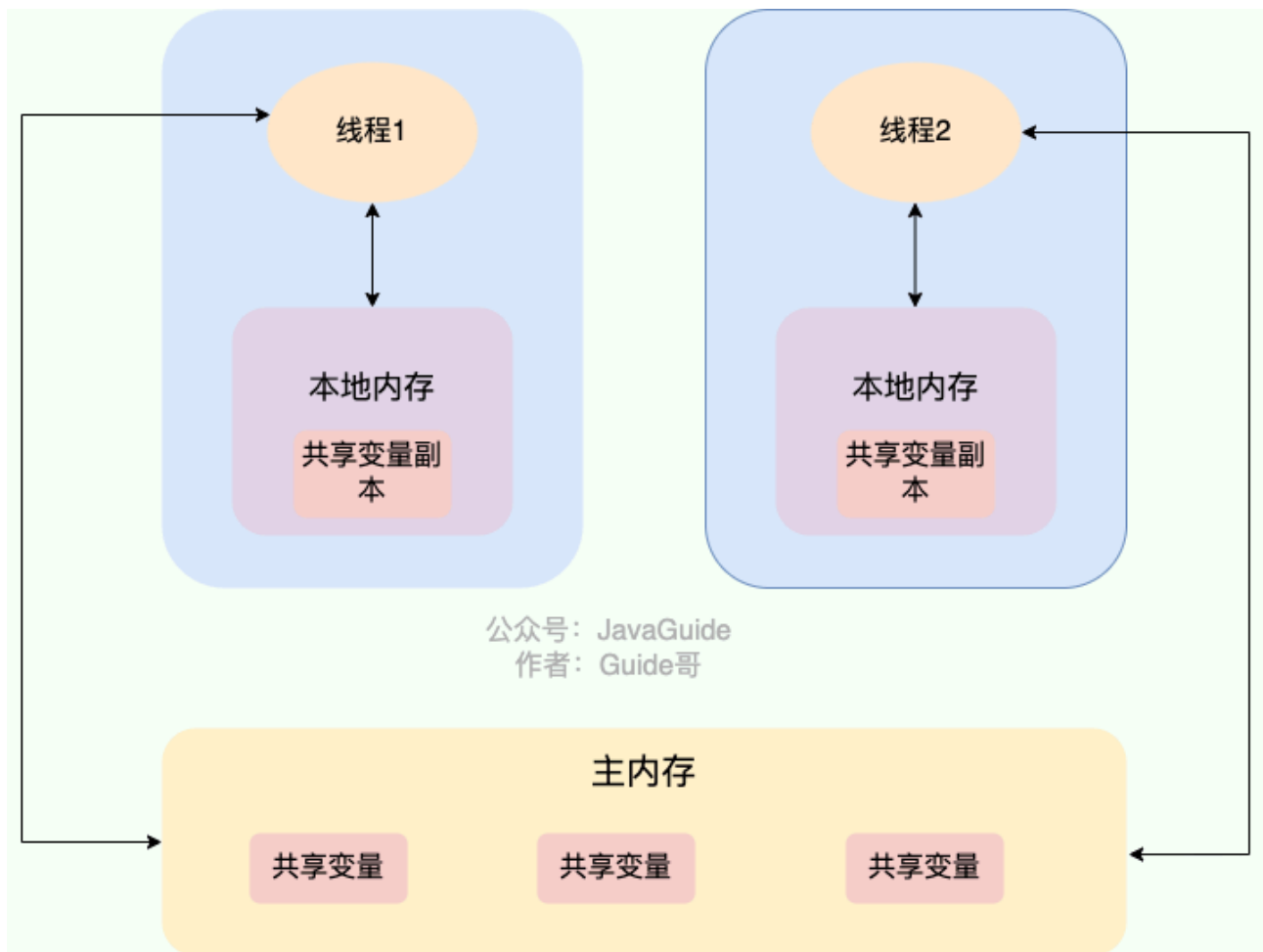
★volatile keyword

How to ensure variable visibility?

In Java, `volatile` the keyword can ensure the visibility of the variable. If we declare a variable as `volatile`, this instructs the JVM that this variable is shared and unstable, and each time it is used, it must be read from the main memory.







`volatile` The keyword is not specific to Java; it also exists in C. Its original meaning was to disable the CPU cache. If we `volatile` modify a variable with , it instructs the compiler that the variable is shared and unstable, requiring it to be read from main memory each time it is used.

`volatile` Keywords can guarantee data visibility, but not data atomicity.

`synchronized` Keywords can guarantee both.

How to disable instruction reordering?

In Java, `volatile` the keyword not only ensures variable visibility, but also plays an important role in preventing JVM instruction reordering. If a variable is declared as `volatile` , when reading or writing this variable, a specific **memory barrier** will be inserted to prevent instruction reordering.

In Java, `Unsafe` classes provide three memory barrier-related methods out of the box shielding the underlying differences in the operating system:



```
1 public native void loadFence();
2 public native void storeFence();
3 public native void fullFence();
```

java

Theoretically, you can achieve `volatile` the same effect as disabling reordering through these three methods, but it will be more troublesome.

Below I will use a common interview question as an example to explain `volatile` the effect of the keyword prohibiting instruction reordering.

During an interview, the interviewer will often ask: "Do you understand the singleton pattern? Write it down for me! Explain to me how the double-checked locking method implements the singleton pattern!"

Double-checked locking implements object singleton (thread-safe) :

```
1 public class Singleton {
2
3     private volatile static Singleton uniqueInstance;
4
5     private Singleton() {
6     }
7
8     public static Singleton getUniqueInstance() {
9         //
10        if (uniqueInstance == null) {
11            //
12            synchronized (Singleton.class) {
13                if (uniqueInstance == null) {
14                    uniqueInstance = new Singleton();
15                }
16            }
17        }
18        return uniqueInstance;
19    }
20 }
```

java

`uniqueInstance volatile` It is also necessary to use keyword modification.
`uniqueInstance = new Singleton();` This code is actually executed in three steps:



1. `uniqueInstance` Allocate memory space for
2. initialization `uniqueInstance`

3. Point `uniqueInstance` to the allocated memory address

However, due to the JVM's instruction reordering, the execution order may become 1 -> 3 -> 2. This instruction reordering is not a problem in a single-threaded environment, but in a multi-threaded environment, it may cause one thread to obtain an uninitialized instance.

For example, thread T1 executes 1 and 3. At this time, thread T2 calls `getUniqueInstance()` and finds `uniqueInstance` that is not null, so it returns `uniqueInstance`, but at this time `uniqueInstance` is not initialized.

Can volatile guarantee atomicity?

volatile Keywords can ensure the visibility of variables, but cannot guarantee that operations on variables are atomic.

We can prove this with the following code:

```
1  /**
2   *      JavaGuide      "      "      Java
3   *
4   * @author Guide
5   * @date 2022/08/03 13:40
6   */
7  public class VolatileAtomicityDemo {
8      public volatile static int inc = 0;
9
10     public void increase() {
11         inc++;
12     }
13
14     public static void main(String[] args) throws
15     InterruptedException {
16         ExecutorService threadPool =
17         Executors.newFixedThreadPool(5);
18         VolatileAtomicityDemo volatileAtomicityDemo = new
19         VolatileAtomicityDemo();
20         for (int i = 0; i < 5; i++) {
21             threadPool.execute(() -> {
22                 for (int j = 0; j < 500; j++) {
23                     volatileAtomicityDemo.increase();
24                 }
25             });
26     }
```



```

27         //      1.5
28         Thread.sleep(1500);
29         System.out.println(inc);
        threadPool.shutdown();
    }
}

```

Under normal circumstances, running the above code should output `2500` . But after you actually run the above code, you will find that the output result is less than `2500` .

Why does this happen? Isn't it said that `volatile` the visibility of variables can be guaranteed?

In other words, if the atomicity of the operation `volatile` is guaranteed , after each thread increments the variable, the other threads can immediately see the modified value. If five threads each perform 500 operations, the final value of `inc` should be $5 * 500 = 2500$. `inc++ inc`

Many people mistakenly believe that the increment operation `inc++` is atomic. In fact, `inc++` it is a compound operation consisting of three steps:

1. Read the value of `inc`.
2. Add 1 to `inc`.
3. Write the value of `inc` back to memory.

`volatile` There is no guarantee that these three operations are atomic, which may lead to the following situation:

1. Thread 1 `inc` reads , but does not modify it. Thread 2 reads `inc` the value of and modifies it (+1), then `inc` writes the value of back to memory.
2. After thread 2 completes the operation, thread 1 `inc` modifies the value of (+1) and then `inc` writes the value of back to the memory.

This results in the two threads each `inc` performing a self-increment operation on , `inc` but actually only increasing 1.

In fact, if you want to ensure that the above code runs correctly, it is very simple, you can use `synchronized` , `Lock` or `AtomicInteger` .

Use `synchronized` Improvement:

```

1     public synchronized void increase() {
2         inc++;

```



Java

```
3     }
```

Use AtomicInteger Improvement:

```
1     public AtomicInteger inc = new AtomicInteger();  
2  
3     public void increase() {  
4         inc.getAndIncrement();  
5     }
```

Use ReentrantLock Improvement:

```
1     Lock lock = new ReentrantLock();  
2     public void increase() {  
3         lock.lock();  
4         try {  
5             inc++;  
6         } finally {  
7             lock.unlock();  
8         }  
9     }
```

★ Optimistic locking and pessimistic locking

What is pessimistic locking?

Pessimistic locking always assumes the worst-case scenario, assuming that a problem will occur every time a shared resource is accessed (such as shared data being modified). Therefore, a lock is acquired every time a resource is acquired. This blocks other threads from accessing the resource until the lock is released by the previous owner. In other words, **a shared resource is only available to one thread at a time, blocking other threads until the lock is released. Once the shared resource is available, the resource is transferred to another thread .**

synchronized Exclusive locks like those in Java ReentrantLock are the implementation of pessimistic locking ideas.



```
1 public void performSynchronisedTask() {
2     synchronized (this) {
3         //
4     }
5 }
6
7 private Lock lock = new ReentrantLock();
8 lock.lock();
9 try {
10    //
11 } finally {
12    lock.unlock();
13 }
```

java

In high-concurrency scenarios, intense lock contention can cause thread blocking. A large number of blocked threads can lead to context switching, increasing system performance overhead. Furthermore, pessimistic locking can lead to deadlock, impacting the normal operation of the code.

What is optimistic locking?

Optimistic locking always assumes the best case scenario, assuming that there will be no problems each time a shared resource is accessed. Threads can execute continuously without locking or waiting. They only verify whether the corresponding resource (that is, data) has been modified by other threads when submitting modifications (the specific method can use the version number mechanism or the CAS algorithm).

In Java, `java.util.concurrent.atomic` the atomic variable classes in the package (for example `AtomicInteger` ,) are implemented `LongAdder` using **CAS, an implementation method of optimistic locking.**




```

1 // LongAdder AtomicInteger AtomicLong java
2 //
3 LongAdder sum = new LongAdder();
4 sum.increment();

```

In high-concurrency scenarios, optimistic locking, compared to pessimistic locking, eliminates thread blocking caused by lock contention and avoids deadlock issues, often resulting in superior performance. However, if conflicts frequently occur (e.g., when writes account for a high percentage of transactions), frequent failures and retries can significantly impact performance and cause CPU usage to spike.

However, the problem of a large number of failed retries can also be solved. As we mentioned earlier, `LongAdder` the method of exchanging space for time solves this problem.


In theory:

- Pessimistic locking is typically used in write-heavy scenarios (high-contention scenarios) to prevent frequent failures and retries from impacting performance. The overhead of pessimistic locking is fixed. However, if optimistic locking solves the problem of frequent failures and retries (for example `LongAdder`), then it can be considered. The decision depends on the actual situation.
- Optimistic locking is typically used in scenarios with low write activity (read-heavy scenarios with low contention) to avoid frequent locking that can impact performance. However, optimistic locking primarily targets single shared variables (see `java.util.concurrent.atomic` the atomic variable class in the package).

How to implement optimistic locking?

Optimistic locking is generally implemented using a version number mechanism or a CAS algorithm. The CAS algorithm is relatively more common and requires special attention here.

Version number mechanism

Typically, a data version number `version` field is added to the data table to indicate the number of times the data has been modified. Each time the data is modified, `version` value is incremented by one. When thread A updates a data value, it reads `version` 

value simultaneously with the data. When submitting the update, the update is only performed if the previously read version value `version` matches the value in the current database. Otherwise, the update is retried until the update succeeds.

Let's take a simple example : suppose there is a version field in the account information table in the database, and the current value is 1; and the current account balance field (`balance`) is \$100.

1. Operator A now reads it out (`version =1`) and deducts \$50 from his account balance ($\$100 - \50).
2. During operator A's operation, operator B also reads this user information (`version =1`) and deducts \$20 from his account balance ($\$100 - \20).
3. Operator A completes the modification and submits the data version number (`version =1`) and the account balance after deduction (`balance =\$50`) to the database for update. At this time, since the submitted data version is equal to the current version of the database record, the data is updated and the database record `version` is updated to 2.
4. Operator B completes the operation and `version` attempts to submit data ($=\$80$) to the database with the version number (`=1`) `balance` . However, when comparing the database record versions, it is found that the data version number submitted by operator B is 1, and the current version recorded in the database is also 2. This does not meet the optimistic locking strategy of "the submitted version must be equal to the current version to perform the update". Therefore, operator B's submission is rejected.

This prevents operator B from `version` overwriting operator A's operation results with the results modified based on old data `= 1`.

CAS algorithm

CAS (**Compare And Swap**) is used to implement optimistic locking and is widely used in various frameworks. The idea behind CAS is simple: it compares an expected value with the value of the variable to be updated, and only updates the variable if the two values are equal.

CAS is an atomic operation that relies on a CPU atomic instruction at the bottom layer.

An atomic operation is the smallest indivisible operation, which means that once an operation starts, it cannot be interrupted until it is completed.



CAS involves three operands:

- **V** : variable value to be updated (Var)

- **E** : Expected
- **N** : New value to be written (New)

If and only if the value of V is equal to E, CAS atomically updates the value of V with the new value N. If not, it means that another thread has updated V, and the current thread abandons the update.

Let's take a simple example : thread A wants to modify the value of variable i to 6, and the original value of i is 1 (V = 1, E = 1, N = 6, assuming there is no ABA problem).

1. i is compared with 1. If they are equal, it means that it has not been modified by other threads and can be set to 6.
2. i is compared with 1. If they are not equal, it means that it has been modified by other threads. The current thread gives up the update and the CAS operation fails.

When multiple threads use CAS to operate a variable at the same time, only one will win and update successfully, and the rest will fail. However, the failed thread will not be suspended, but will only be informed of the failure and allowed to try again. Of course, the failed thread is also allowed to give up the operation.

The Java language does not directly implement CAS. CAS-related implementations are implemented through C++ inline assembly (JNI calls). Therefore, the specific implementation of CAS is related to both the operating system and the CPU.

`sun.misc` The classes under the package `Unsafe` provide `compareAndSwapObject` methods to implement CAS operations on types of `Object`, `int`, and `long`.

```

1  /**
2   * CAS
3   * @param o          field
4   * @param offset      field
5   * @param expected
6   * @param update
7   * @return           true | false
8   */
9  public final native boolean compareAndSwapObject(Object o, long
10 offset, Object expected, Object update);
11
12
13

```

java

```
public final native boolean compareAndSwapInt(Object o, long
offset, int expected, int update);

public final native boolean compareAndSwapLong(Object o, long
offset, long expected, long update);
```

For `Unsafe` a detailed introduction to the class, see this article: [Detailed Explanation of Java Magic Class Unsafe - JavaGuide - 2022](#) .

How is CAS implemented in Java?

In Java, a key class for implementing CAS (Compare-And-Swap) operations is `Unsafe` .

`Unsafe` This class `sun.misc` , located in the package `Unsafe`, provides low-level, unsafe operations. Due to its powerful functionality and potential dangers, it is typically used internally within the JVM or in libraries that require extremely high performance and low-level access. It is not recommended for general developers to use in applications. For

`Unsafe` a detailed introduction to this class, please read this article: [A Detailed Explanation of the Java Magic Class Unsafe](#) .

`sun.misc` The classes under the package `Unsafe` provide `compareAndSwapObject` methods to implement CAS operations on types of `Object` `int` `long` :

```
1  /**
2   *
3   *
4   * @param o
5   * @param offset
6   * @param expected
7   * @param x
8   * @return true false
9   */
10 boolean compareAndSwapObject(Object o, long offset, Object
11 expected, Object x);
12
13 /**
14 * int
15 */
16 boolean compareAndSwapInt(Object o, long offset, int expected, int
17 x);
```

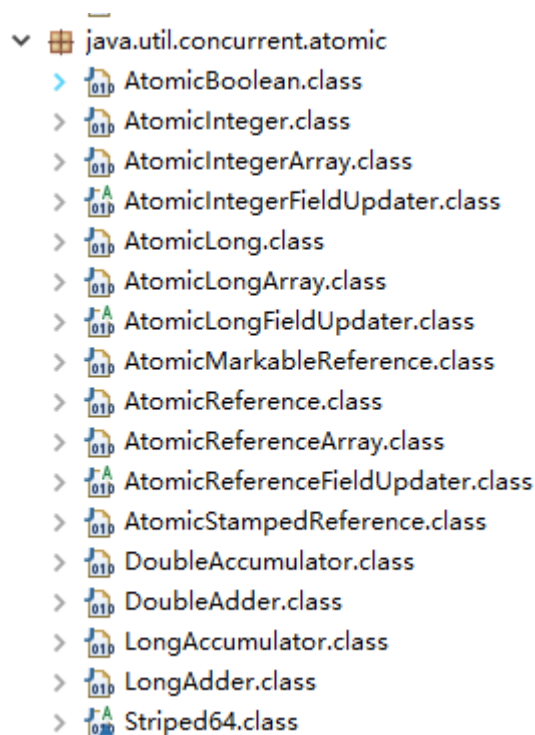
```

18  /**
19   *           long
20   */
   boolean compareAndSwapLong(Object o, long offset, long expected,
   long x);

```

Unsafe CAS methods in classes are native methods. The keyword indicates that these methods are implemented in native code (usually C or C++) rather than Java. These methods directly call underlying hardware instructions to perform atomic operations. In other words, Java doesn't implement CAS directly in Java, but rather in C++ inline assembly (via JNI calls). Therefore, the specific implementation of CAS is closely related to the operating system and CPU.

`java.util.concurrent.atomic` The package provides some classes for atomic operations. These classes use the underlying atomic instructions to ensure that operations in a multi-threaded environment are thread-safe.



For the introduction and use of these Atomic atomic classes, you can read this article: [Atomic Atomic Class Summary](#) .

`AtomicInteger` It is one of Java's atomic classes, mainly used to `int` perform atomic operations on variables of type `int`. It uses `Unsafe` the low-level atomic operation methods provided by the class to achieve lock-free thread safety.



Next, we will explain how Java uses class methods to implement atomic operations by interpreting `AtomicInteger` the core source code (JDK1.8) . `Unsafe`

`AtomicInteger` The core source code is as follows:

```

1      //      Unsafe
2      private static final Unsafe unsafe = Unsafe.getUnsafe();
3      private static final long valueOffset;
4
5      static {
6          try {
7              //      "value"      AtomicInteger
8              valueOffset = unsafe.objectFieldOffset
9                  (AtomicInteger.class.getDeclaredField("value"));
10         } catch (Exception ex) { throw new Error(ex); }
11     }
12     //      "value"
13     private volatile int value;
14
15     //      newValue
16     //      Unsafe#compareAndSwapInt      CAS
17     public final boolean compareAndSet(int expect, int update) {
18         return unsafe.compareAndSwapInt(this, valueOffset, expect,
19         update);
20     }
21
22     //      delta
23     public final int getAndAdd(int delta) {
24         return unsafe.getAndAddInt(this, valueOffset, delta);
25     }
26
27     //      1
28     //      Unsafe#getAndAddInt      CAS
29     public final int getAndIncrement() {
30         return unsafe.getAndAddInt(this, valueOffset, 1);
31     }
32
33     //      1
34     public final int getAndDecrement() {
35         return unsafe.getAndAddInt(this, valueOffset, -1);

```

`Unsafe#getAndAddInt` Source code:

```

1 // java
2 public final int getAndAddInt(Object o, long offset, int delta) {
3     int v;
4     do {
5         // volatile o offset
6         v = getIntVolatile(o, offset);
7     } while (!compareAndSwapInt(o, offset, v, v + delta));
8     //
9     return v;
10 }

```

As you can see, a loop `getAndAddInt` is used `do-while` : if `compareAndSwapInt` an operation fails, it will be retried until it succeeds. That is, `getAndAddInt` the method will `compareAndSwapInt` try to update value the value of using the method. If the update fails (the current value has been modified by another thread during this time), it will re-obtain the current value and try to update it again until the operation succeeds.


Because CAS operations may fail due to concurrency conflicts, they are often while used in conjunction with loops to repeatedly retry after failure until the operation succeeds. This is **the spin lock mechanism** .

What are the problems with the CAS algorithm?

The ABA problem is the most common problem of the CAS algorithm.

ABA Issues

If a variable V is initially read with value A and then checked to be still A when being assigned a value, can we conclude that its value has not been modified by another thread? Obviously not, because during this time, its value could be changed to another value and then back to A. In this case, the CAS operation would mistakenly believe that it had never been modified. This problem is known as **the "ABA" problem of CAS operations**.

The solution to the ABA problem is to prepend **a version number or timestamp** to the variable . The `__register__` class, available since JDK 1.5 `AtomicStampedReference` , is designed to address the ABA problem. Its `__register__` `compareAndSet()` method first checks whether the current reference is equal to the expected reference and whether the current flag is equal to the expected flag. If so, it atomically sets the reference and flag  the given updated value.

```

1  public boolean compareAndSet(V    expectedReference,      java
2                                V    newReference,
3                                int    expectedStamp,
4                                int    newStamp) {
5      Pair<V> current = pair;
6      return
7          expectedReference == current.reference &&
8          expectedStamp == current.stamp &&
9          ((newReference == current.reference &&
10             newStamp == current.stamp) ||
11             casPair(current, Pair.of(newReference, newStamp)));
12  }

```

Long cycle time and high overhead

CAS often uses spin operations to perform retries, which means that if the retries fail, the retries will continue in a loop until they succeed. If the retries fail for a long time, it will cause a lot of CPU overhead.

If the JVM can support the instructions provided by the processor `pause`, the efficiency of the spin operation will be improved. `pause` The instructions have two important functions:

1. **Delay pipeline execution instructions** : `pause` This instruction can delay the execution of instructions, thereby reducing CPU resource consumption. The specific delay time depends on the processor implementation version. On some processors, the delay time may be zero.
2. **Avoid memory order conflicts** : When exiting a loop, `pause` the instruction can prevent the CPU pipeline from being flushed due to memory order conflicts, thereby improving the CPU's execution efficiency.

Only one atomic operation of a shared variable can be guaranteed

CAS operations only work on a single shared variable. When multiple shared variables need to be manipulated, CAS becomes ineffective. However, starting with JDK 1.5, Java provides `AtomicReference` classes that enable atomicity between referenced objects. By encapsulating multiple variables in a single object, we can use `AtomicReference` CAS operations.



In addition `AtomicReference` to this method, locking can also be used to ensure it.

synchronized keyword

What is synchronized? What is it used for?

`synchronized` It is a keyword in Java, which means synchronization when translated into Chinese. It mainly solves the synchronization of resource access between multiple threads, and can ensure that the method or code block modified by it can only be executed by one thread at any time.

In early Java versions, monitor locks `synchronized` were considered **heavyweight** and inefficient. This was because monitor locks relied on the underlying operating system `Mutex Lock` for implementation, and Java threads were mapped to native operating system threads. Suspending or waking a thread required the assistance of the operating system, which, in turn, required transitions from user mode to kernel mode. This state transition took a relatively long time and was costly.

However, Java 6 and later `synchronized` introduced numerous optimizations, such as spin locks, adaptive spin locks, lock elision, lock coarsening, biased locks, and lightweight locks, to reduce the overhead of locking operations. These optimizations have `synchronized` significantly improved locking efficiency. Therefore, `synchronized` they can still be used in real-world projects, including extensively in the JDK source code and many open source frameworks `synchronized` .

A further point about biased locking: Because biased locking increases JVM complexity and doesn't necessarily improve performance for all applications, it was disabled by default in JDK 15 (though it can still be `-XX:+UseBiasedLocking` enabled using). In JDK 18, biased locking has been completely deprecated (it can no longer be enabled from the command line).

How to use synchronized?

`synchronized` There are three main ways to use keywords:

1. Decorating instance methods
2. Modifying static methods
3. Decorating code blocks

1. Modify instance method (lock current object instance)



Lock the current object instance and obtain the lock of the current object instance before entering the synchronization code .

```
1   synchronized void method() {  
2       //  
3   }
```

java

2. Modify static methods (lock the current class)

Locking the current class will affect all object instances of the class. **The lock of the current class** must be obtained before entering the synchronization code .

This is because static members do not belong to any instance object, but to the entire class. They are independent of the specific instance of the class and are shared by all instances of the class.

```
1   synchronized static void method() {  
2       //  
3   }
```

java

Are calls between static synchronized and non-static methods mutually exclusive? They are not! If thread A calls a non-static method of an instance object, and thread B calls a static method of the class to which the instance object belongs, both are allowed and no mutual exclusion occurs. This is because the lock occupied by accessing a static method is the lock of the current class, while the lock occupied by accessing a non-static method is the lock of the current instance object. synchronized synchronized synchronized synchronized synchronized

3. Modify the code block (lock the specified object/class)

Lock the object/class specified in the brackets:

- `synchronized(object)` **Indicates that the lock of the given object** must be obtained before entering the synchronized code block .
- `synchronized(.class)` **Indicates that the lock of the given Class** must be obtained before entering the synchronized code block

```
1   synchronized(this) {  
2       //  
3   }
```

java

Summarize:

- `synchronized` Keywords added to `static` static methods and `synchronized(class)` code blocks are used to lock the Class class;
- `synchronized` The keyword is added to the instance method to lock the object instance;
- Try not to use it `synchronized(String a)` because the string constant pool in JVM has a cache function.

Can the constructor be modified with `synchronized`?

Constructors cannot be modified with the `synchronized` keyword. However, `synchronized` code blocks can be used inside constructors.

In addition, the construction method itself is thread-safe, but if the construction method involves operations on shared resources, appropriate synchronization measures need to be taken to ensure the thread safety of the entire construction process.

★ Do you understand the underlying principles of `synchronized`?

The underlying principle of the `synchronized` keyword belongs to the JVM level.

The case of `synchronized` statement blocks

```
1 public class SynchronizedDemo {
2     public void method() {
3         synchronized (this) {
4             System.out.println("synchronized ");
5         }
6     }
7 }
```

java

Use the JDK's built-in `javap` command to view `SynchronizedDemo` the relevant bytecode information of the class: first switch to the corresponding directory of the class and execute `javac SynchronizedDemo.java` the command to generate the compiled `.class` file, and then execute it `javap -c -s -v -l SynchronizedDemo.class`.



```

public void method();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
    0: aload_0
    1: dup
    2: astore_1
    3: monitorenter
    4: getstatic #2                // Field java/lang/System.out:Ljava/io/PrintStream;
    7: ldc #3                      // String Method 1 start
    9: invokevirtual #4            // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   12: aload_1
   13: monitorexit
   14: goto #22
   17: astore_2
   18: aload_1
   19: monitorexit
   20: aload_2
   21: athrow
   22: return
  Exception table:
    from   to   target type
    4      14    17    any
    17     20    17    any
  LineNumberTable:
    line 5: 0
    line 6: 4
    line 7: 12
    line 8: 22
  StackMapTable: number_of_entries = 2
    frame_type = 255 /* full_frame */
    offset_delta = 17
    locals = [ class test/SynchronizedDemo, class java/lang/Object ]
    stack = [ class java/lang/Throwable ]
    frame_type = 250 /* chop */
    offset_delta = 4
}
SourceFile: "SynchronizedDemo.java"


```

From the above we can see that **synchronized the implementation of the synchronized statement block uses the monitorenter and monitorexit instructions, where monitorenter the instruction points to the starting position of the synchronized code block and monitorexit the instruction indicates the end position of the synchronized code block.**

The above bytecode contains one monitorenter instruction and two monitorexit instructions, which are to ensure that the lock can be released correctly in both cases when the synchronized code block code is executed normally and when an exception occurs.

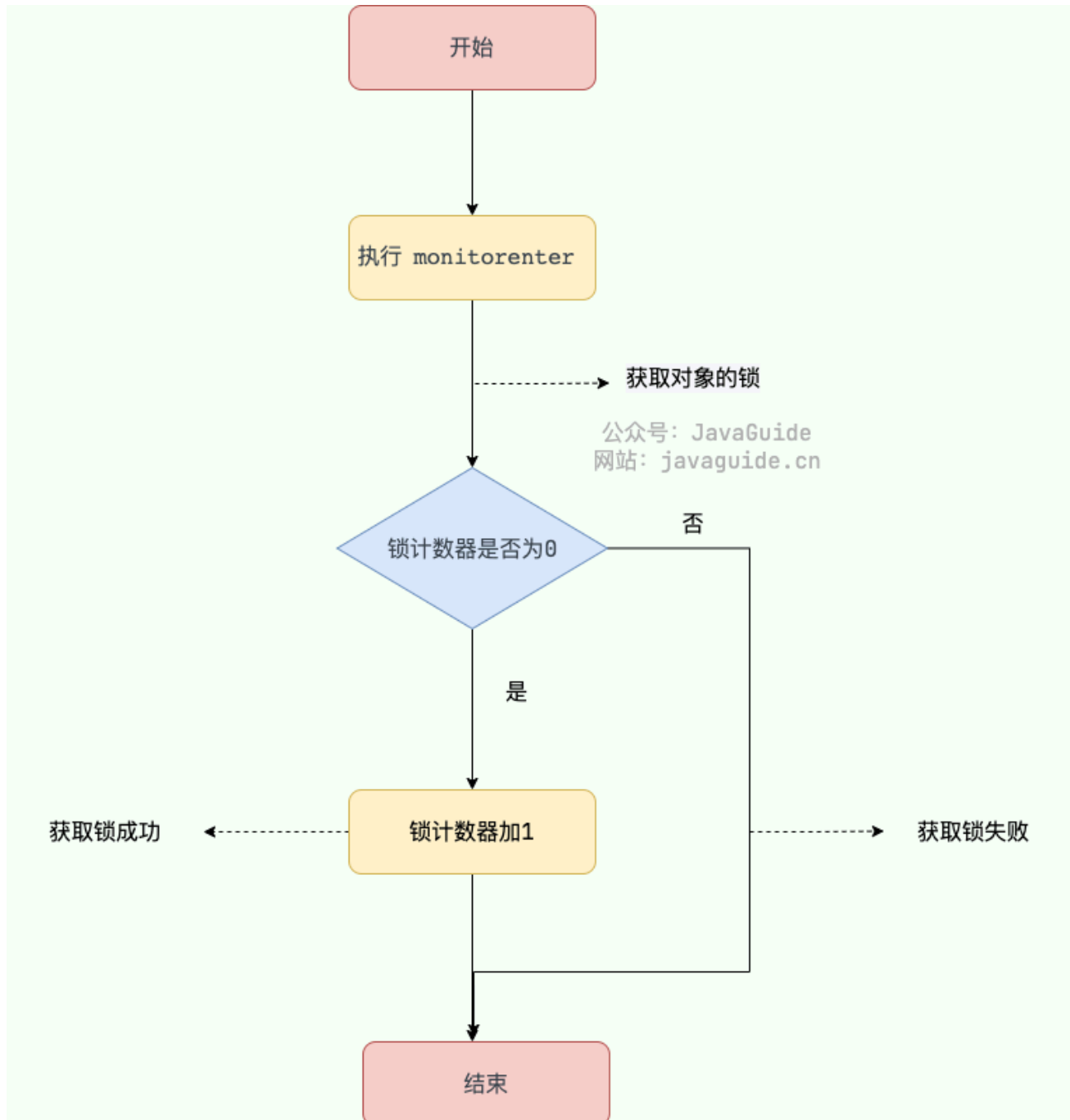
When executing monitorenter the instruction, the thread attempts to acquire the lock, that is, to obtain ownership of **the object monitor . monitor**

In the Java Virtual Machine (HotSpot), Monitor is implemented in C++ and is implemented by ObjectMonitor . Each object has an ObjectMonitor object built into it.

In addition, wait/notify the wait method also depends on monitor the object,  is why the wait method can only be called in a synchronized block or method wait/notify , otherwise java.lang.IllegalMonitorStateException an exception

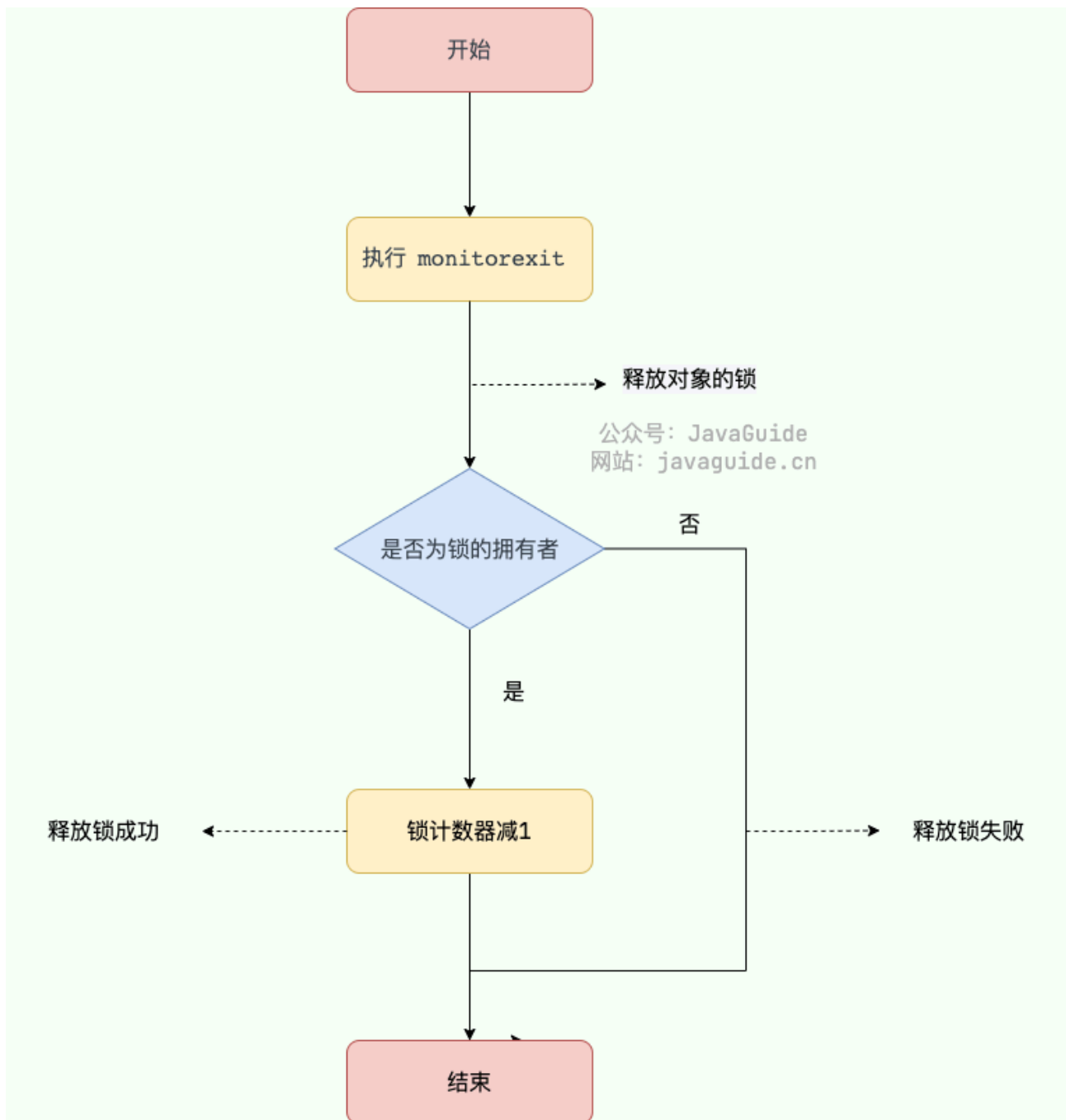
will be thrown.

During execution `monitorenter`, an attempt is made to acquire the lock of the object. If the lock counter is 0, it means that the lock can be acquired. After acquisition, the lock counter is set to 1, that is, plus 1.



The owner thread of the object lock can execute `monitorexit` the instruction to release the lock. After executing `monitorexit` the instruction, the lock counter is set to 0, indicating that the lock is released and other threads can try to acquire the lock.





If the object lock acquisition fails, the current thread will be blocked and wait until the lock is released by another thread.

The case of synchronized modified methods

```
1 public class SynchronizedDemo2 {
2     public synchronized void method() {
3         System.out.println("synchronized ");
4     }
5 }
```

java



```

public test.SynchronizedDemo2();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
    0: aload_0
    1: invokespecial #1           // Method java/lang/Object.<init>:()V
    4: return
 LineNumberTable:
    line 3: 0

public synchronized void method();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=1, args_size=1
    0: getstatic     #2           // Field java/lang/System.out:Ljava/io/PrintStream;
    3: ldc          #3           // String synchronized 钼规砗
    5: invokevirtual #4           // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    8: return
 LineNumberTable:
    line 5: 0
    line 6: 8
}
SourceFile: "SynchronizedDemo2.java"

```

synchronized The modified method does not have `monitorenter` the `___access__` flag or `monitorexit` the `___access__` flag. Instead, it has `ACC_SYNCHRONIZED` the `___access__` flag, which indicates that the method is a synchronized method. The JVM uses this `ACC_SYNCHRONIZED` `___access__` flag to identify whether a method is declared as a synchronized method and executes the corresponding synchronization call.

If it is an instance method, the JVM will try to acquire the lock of the instance object. If it is a static method, the JVM will try to acquire the lock of the current class.

Summarize

synchronized The implementation of the synchronized statement block uses the `monitorenter` and `monitorexit` instructions, where `monitorenter` the instruction points to the start position of the synchronized code block and `monitorexit` the instruction indicates the end position of the synchronized code block.

synchronized The modified method does not have `monitorenter` the directive and `monitorexit` the directive. Instead, it has `ACC_SYNCHRONIZED` the identifier, which indicates that the method is a synchronized method.

However, the essence of both is to obtain the object monitor monitor.

Related recommendations: [Things about Java locks and threads - Youzan Technology Team](#) .



Advanced: Those who have more time to learn can take the time to study object monitors in detail `monitor` .



What optimizations have been made to the underlying synchronized implementation since JDK 1.6? Do you understand the principles of lock escalation?

After Java 6, synchronized a large number of optimization technologies such as spin locks, adaptive spin locks, lock elimination, lock coarsening, biased locks, and lightweight locks were introduced to reduce the overhead of lock operations. These optimizations have greatly improved the efficiency of locks (in JDK18, biased locks have been completely abandoned, as mentioned earlier).

Locks exist in four main states: unlocked, biased, lightweight, and heavyweight. These states gradually escalate as competition intensifies. Note that locks can be upgraded but not downgraded. This strategy improves the efficiency of acquiring and releasing locks.

synchronized Lock escalation is a relatively complex process and is rarely asked in interviews. If you want to learn more about it, you can read this article: [A Brief Analysis of the Principles and Implementation of Synchronized Lock Escalation](#) .

Why is synchronized biased lock abandoned?

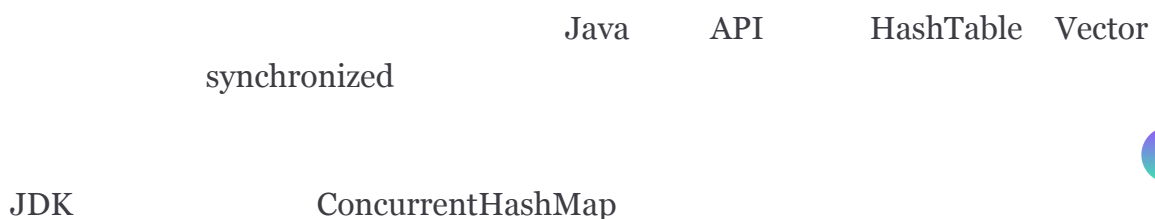
Open JDK official statement: [JEP 374: Deprecate and Disable Biased Locking](#)

In JDK 15, biased locking is disabled by default (you can still – XX:+UseBiasedLocking enable biased locking using). In JDK 18, biased locking has been completely deprecated (it cannot be enabled from the command line).

In the official statement, there are two main reasons:

- **Performance gains are not obvious:**

Biased locking is an optimization technology of the HotSpot virtual machine that can improve the performance of single-threaded access to synchronized code blocks.



safe point

- JVM

HotSpot
OpenJDK

★synchronized volatile

synchronized

volatile

- volatile volatile synchronized
- volatile synchronized volatile synchronized
- volatile synchronized
- volatile synchronized

ReentrantLock

ReentrantLock

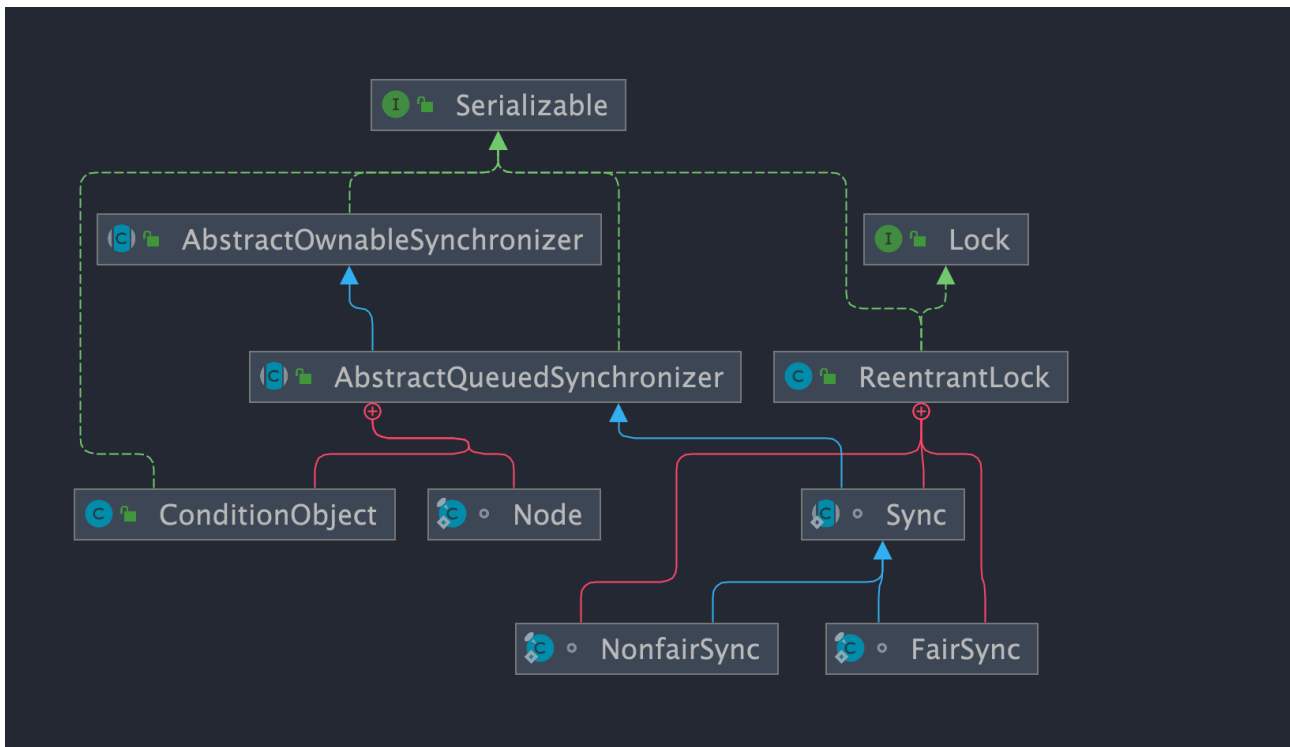
ReentrantLock Lock synchronized
ReentrantLock

1

```
public class ReentrantLock implements Lock, java.io.Serializable {
```

ReentrantLock Sync Sync AQS
AbstractQueuedSynchronizer Sync FairSync NonfairSync Sync





ReentrantLock

```

1 //      boolean    true      false      java
2 public ReentrantLock(boolean fair) {
3     sync = fair ? new FairSync() : new NonfairSync();
4 }

```

ReentrantLock AQS AQS

AQS_____

- :
-

★synchronized ReentrantLock



JDK	Lock	synchronized
method1()	method1() method2()	method2() synchronized

```

1 public class SynchronizedDemo {
2     public synchronized void method1() {
3         System.out.println("    1");
4         method2();
5     }
6
7     public synchronized void method2() {
8         System.out.println("    2");
9     }
10 }

```

synchronized	method1()
method2()	
synchronized	
method2()	

synchronized	JVM	ReentrantLock	API
synchronized	JVM		JDK1.6
synchronized			
ReentrantLock	JDK	API	lock() unlock()
try/finally			

ReentrantLock synchronized

synchronized ReentrantLock

- : ReentrantLock
lock.lockInterruptibly()



interrupt()

InterruptedException

- `ReentrantLock` `synchronized`
`ReentrantLock` `ReentrantLock(boolean fair)`
- `notify()` / `notifyAll()` `synchronized` `wait()`
`Condition` `newCondition()` `ReentrantLock`
- `ReentrantLock` `tryLock(timeout)`

`ReentrantLock`

`Condition`

`Condition` `JDK1.5`

`Lock`

`Condition`

`Condition`

`notify()/notifyAll()`

JVM

`ReentrantLock`

`Condition`

“

”

`Condition`

`synchronized`

`Lock`

`Condition`

`notifyAll()`

`Condition`

`signalAll()`

`Condition`

`lockInterruptibly()`

`interrupt()`

`InterruptedException`

`Stack Overflow`

`lockInterruptibly()`

```

1 public class MyReentrantlock {
2     Thread t = new Thread() {
3         @Override
4         public void run() {
5             ReentrantLock r = new ReentrantLock();
              // 1.1

```

java



```

6         r.lock();
7
8         // 1.2                                1
9         System.out.println("lock() : lock count :" +
10 r.getHoldCount());
11
12         // 2
13 Thread.currentThread().isInterrupted()
14 true
15         interrupt();
16         System.out.println("Current thread is intrupted");
17
18         // 3.1
19         r.tryLock();
20         // 3.2                                2
21         System.out.println("tryLock() on intrupted thread
22 lock count :" + r.getHoldCount());
23         try {
24             // 4                                true
25 lockInterruptibly()                        InterruptedException
26             System.out.println("Current Thread
27 isInterrupted:" + Thread.currentThread().isInterrupted());
28             r.lockInterruptibly();
29             System.out.println("lockInterruptibly() --NOT
30 executable statement" + r.getHoldCount());
31         } catch (InterruptedException e) {
32             r.lock();
33             System.out.println("Error");
34         } finally {
35             r.unlock();
36         }
37
38         // 5                                lockInterruptibly()
39
40         System.out.println("lockInterruptibly() not able to
41 Acquirie lock: lock count :" + r.getHoldCount());
42
43         r.unlock();
44         System.out.println("lock count :" +
45 r.getHoldCount());
46         r.unlock();
47         System.out.println("lock count :" +
48 r.getHoldCount());

```



```

        }
    };
    public static void main(String str[]) {
        MyReentrantLock m = new MyReentrantLock();
        m.t.start();
    }
}

```

```

1 lock() : lock count :1
2 Current thread is interrupted
3 tryLock() on interrupted thread lock count :2
4 Current Thread isInterrupted:true
5 Error
6 lockInterruptibly() not able to Acquire lock: lock count :2
7 lock count :1
8 lock count :0

```

bash

```

        tryLock(timeout)
tryLock(timeout)
true                                false

•                                tryLock(timeout)

•

•                                tryLock(timeout)

```

-
- ReentrantLock
- synchronized



ReentrantReadWriteLock

ReentrantReadWriteLock

JDK 1.8

StampedLock

ReentrantReadWriteLock

ReentrantReadWriteLock

ReadWriteLock

```
1 public class ReentrantReadWriteLock
2     implements ReadWriteLock, java.io.Serializable{
3 }
4 public interface ReadWriteLock {
5     Lock readLock();
6     Lock writeLock();
7 }
```

-
-

ReentrantReadWriteLock

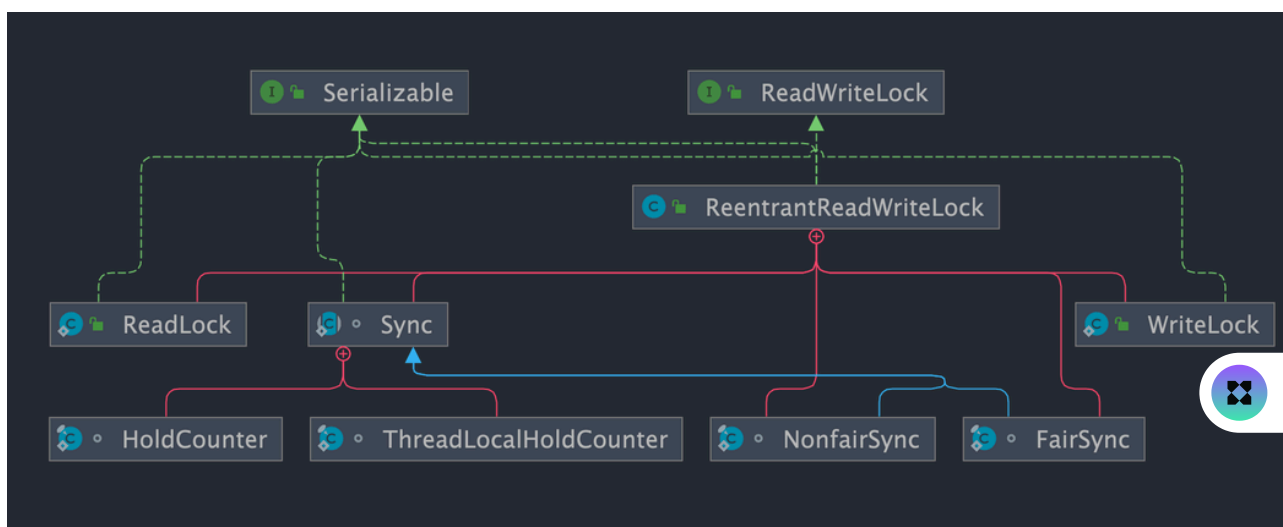
WriteLock ()

ReadLock

ReentrantLock

ReentrantReadWriteLock

AQS



ReentrantReadWriteLock

```
1 //          boolean      true          false          java
2 public ReentrantReadWriteLock(boolean fair) {
3     sync = fair ? new FairSync() : new NonfairSync();
4     readerLock = new ReadLock(this);
5     writerLock = new WriteLock(this);
6 }
```

ReentrantReadWriteLock

ReentrantReadWriteLock

ReentrantReadWriteLock

-
-

-
-

()

Java JVM -



StampedLock

StampedLock

StampedLock

StampedLock JDK 1.8

Condition

Lock
CLH

StampedLock
AQS

Lock

ReadWriteLock

```
1 public class StampedLock implements java.io.Serializable {
2 }
```

StampedLock

-

ReentrantReadWriteLock

-

ReentrantReadWriteLock

-

StampedLock

```
1 long tryConvertToWriteLock(long stamp){}
2 long tryConvertToReadLock(long stamp){}
3 long tryConvertToOptimisticRead(long stamp){}
```

StampedLock

long

0

StampedLock



```

1 //
2 public long writeLock() {
3     long s, next; // bypass acquireWrite in fully unlocked case
4     only
5     return (((s = state) & ABITS) == 0L &&
6         U.compareAndSwapLong(this, STATE, s, next = s + WBIT))
7     ?
8         next : acquireWrite(false, 0L));
9 }
10 //
11 public long readLock() {
12     long s = state, next; // bypass acquireRead on common
13     uncontended case
14     return ((whead == wtail && (s & ABITS) < RFULL &&
15         U.compareAndSwapLong(this, STATE, s, next = s +
16         RUNIT)) ?
17         next : acquireRead(false, 0L));
18 }
19 //
20 public long tryOptimisticRead() {
21     long s;
22     return (((s = state) & WBIT) == 0L) ? (s & SBITS) : 0L;
23 }

```

StampedLock

StampedLock ReadWriteLock

StampedLock

StampedLock

ReentrantReadWriteLock StampedLock

ReentrantReadWriteLock

StampedLock

CPU

StampedLock

Condition

ReentrantLock



StampedLock

StampedLock

StampedLock

StampedLock

StampedLock

Lock

ReadWriteLock

CLH

AQS

CLH

StampedLock

CLH

state

StampedLock

AQS

- AQS
- StampedLock

AQS

StampedLock

Atomic

Atomic

Atomic

- Java
- Java
- Guide to the Volatile Keyword in Java - Baeldung <https://www.baeldung.com/java-volatile>
- Java“ ” - <https://tech.meituan.com/2018/11/15/java-lock.html>
- ReadWriteLock <https://cloud.tencent.com/developer/article/1176230>
- StampedLock <https://mp.weixin.qq.com/s/2Acujjr4BHIhFsCLGwYSg>
- Java ThreadLocal - <https://droidyue.com/blog/2016/03/13/learning-threadlocal-in-java/>



- ThreadLocal (Java Platform SE 8) - Oracle Help Center

<https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html>

JavaGuide官方公众号 (微信搜索JavaGuide)



- 1、公众号后台回复“PDF”获取原创PDF面试手册
- 2、公众号后台回复“学习路线”获取Java学习路线最新版
- 3、公众号后台回复“开源”获取优质Java开源项目合集
- 4、公众号后台回复“八股文”获取Java面试真题+面经

2025/8/7 15:13

Contributors: SnailClimb , Farahani , halle , yellowgg , Ryze-Zhao , Snailclimb , shuang.kou , guide , Lshare , qiuyukang , pengchen211 , drlifeL , Tan Jiuding , 2293736867 , kaka2634 , chengejk , HangdianGhostMr. , cxhello , WangjiaW , Curvature , Itswag , Evan He , JuiceApp1e , Verne.Chung , Carbda , Guide , ale , Nicolas , Mr.Hope , paigeman , liuxiaocs7 , aucub , liangyi , smy1999 , suppered , 11 , Kisa-Dong , Slade , Wenweigood , uncle-lv

Copyright © 2025 Guide

