Summary of Common Java Concurrency Interview Questions (Part 2)

Guide Java Java Concurrency About 17,324 words About 58 minutes

ThreadLocal

What is ThreadLocal used for?

Typically, variables we create can be accessed and modified by any thread. This can lead to data races and thread safety issues in a multi-threaded environment. So, **if we want each thread to have its own dedicated local variables, how can we achieve this?**

The JDK provides `ThreadLocal` the class to address this problem. **`ThreadLocal` The class allows each thread to bind its own values** , figuratively likening it to a "data storage box." Each thread has its own independent box for storing private data, ensuring that data between threads does not interfere with each other.

When you create a `ThreadLocal` variable, each thread that accesses it has its own copy. This is `ThreadLocal` where the name derives from it. A thread can `get()` access its own local copy using the __get__ method or `set()` modify the value of that copy using the __get__ method, thus avoiding thread safety issues.

Let's take a simple example: Imagine two people go to a treasure house to collect treasures. If they share a bag, they'll inevitably argue. But if each person has their own bag, there's no such problem. If these two people are like threads, then [the `ThreadLocal` code] is a method used to prevent them from competing for the same resource.

```java
public class ThreadLocalExample {
    private static ThreadLocal<Integer> threadLocal =
ThreadLocal.withInitial(() -> 0);

    public static void main(String[] args) {
        Runnable task = () -> {
            int value = threadLocal.get();
            value += 1;
            threadLocal.set(value);
            System.out.println(Thread.currentThread().getName() + "
Value: " + threadLocal.get());
        };

        Thread thread1 = new Thread(task, "Thread-1");
        Thread thread2 = new Thread(task, "Thread-2");

        thread1.start(); //    : Thread-1 Value: 1
        thread2.start(); //    : Thread-2 Value: 1
    }
}
```

⭐Do you understand the principle of ThreadLocal?

Start with `Thread` the class source code.

```java
public class Thread implements Runnable {
    //......
    //           ThreadLocal     ThreadLocal
    ThreadLocal.ThreadLocalMap threadLocals = null;

    //           InheritableThreadLocal     InheritableThreadLocal

    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
    //......
}
```

`Thread` From the source code above , we can see that `Thread` the class has a `threadLocals` and a `inheritableThreadLocals` variable, both `ThreadLocalMap` of type . We can `ThreadLocalMap` understand as `ThreadLocal` a customization implemented by the class `HashMap` . By default, both variables are null. They are created only when the current thread calls `ThreadLocal` the `set` or `get` method of the class. In fact, when calling these two methods, we are calling `ThreadLocalMap` the corresponding `get()` or `set()` method of the class.

`ThreadLocal` Class `set()` methods

```java
public void set(T value) {
    //
    Thread t = Thread.currentThread();
    //    Thread         threadLocals     (         )
    ThreadLocalMap map = getMap(t);
    if (map != null)
        //
        map.set(this, value);
    else
        createMap(t, value);
}
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

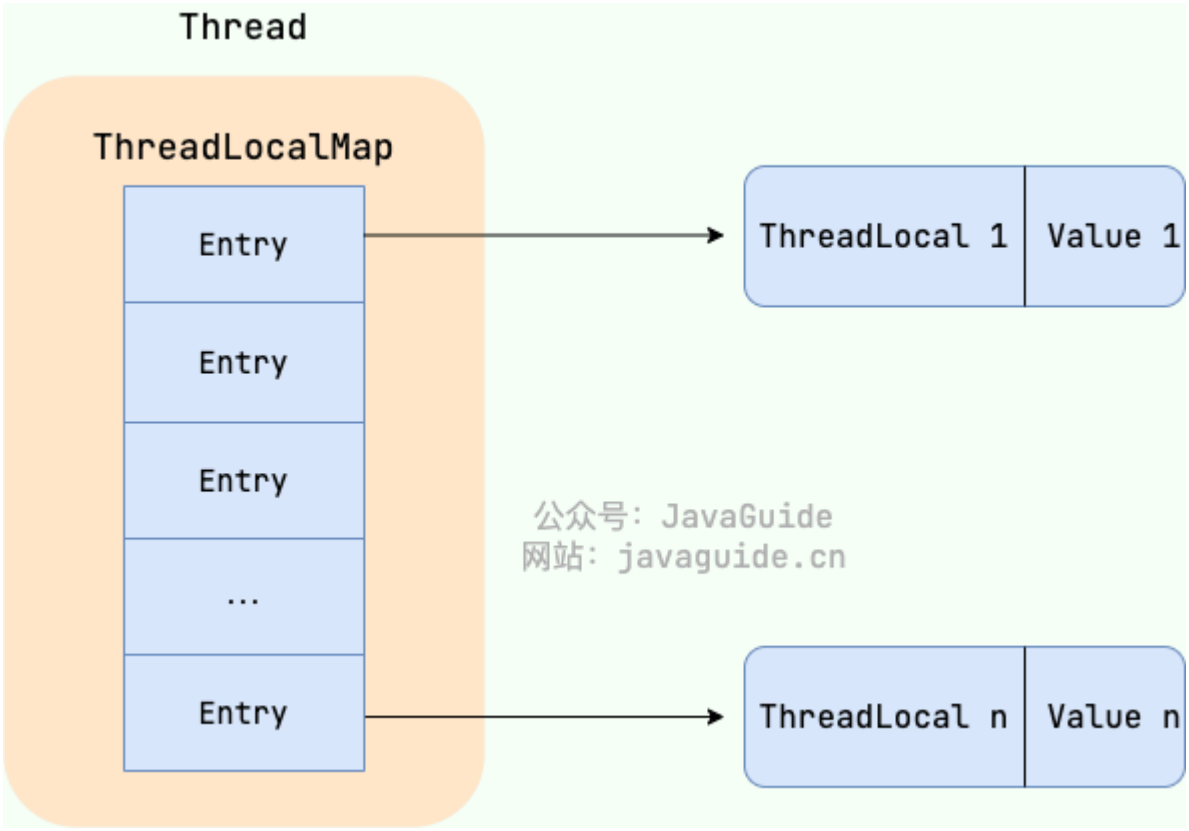
From the above, we can conclude that **the final variable is placed `ThreadLocalMap` in the current thread's , not on `ThreadLocal` . This `ThreadLocal` can be understood as just `ThreadLocalMap` a wrapper around , passing variable values.** `ThrealLocal` The class can access the current thread object `Thread.currentThread()` directly through after obtaining it . `getMap(Thread t)` `ThreadLocalMap`

Each `Thread` has a key-value pair `ThreadLocalMap` that `ThreadLocalMap` can store an object `ThreadLocal` as the key and an object as the value.

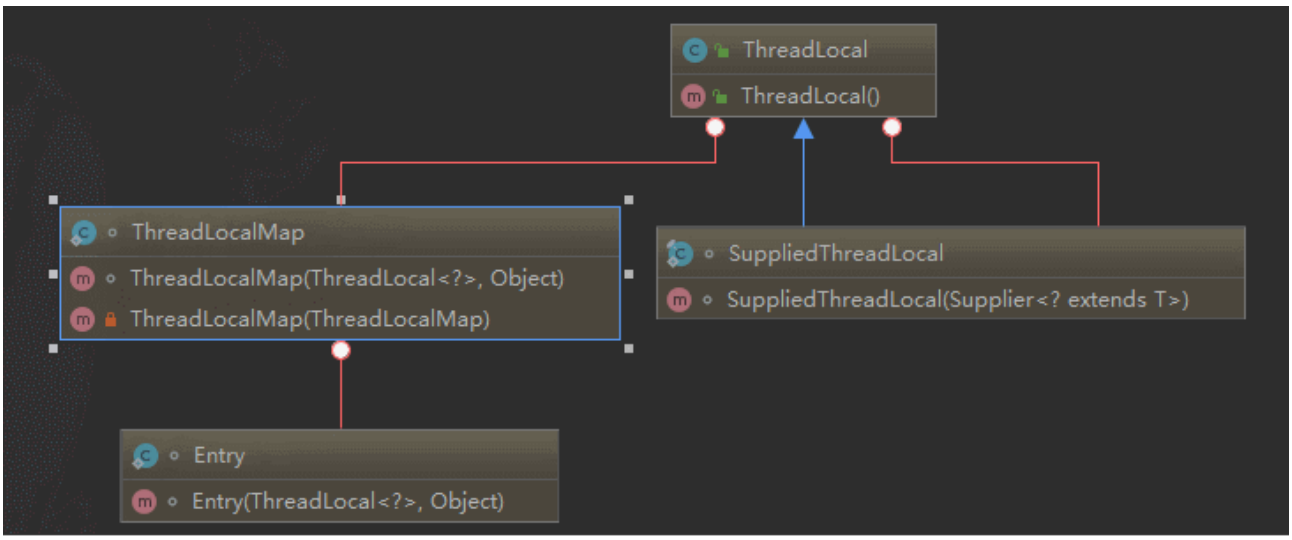
```java
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
    //......
}
```

`ThreadLocal` For example, if we declare two objects in the same thread , `Thread` the only one used to `ThreadLocalMap` store data `ThreadLocalMap` is the key of `ThreadLocal` the object, and the value is the value set by `ThreadLocal` the object calling method. `set`

`ThreadLocal` The data structure is shown in the figure below:



`ThreadLocalMap` Yes `ThreadLocal` , a static inner class.



⭐What causes ThreadLocal memory leaks?

`ThreadLocal` The root cause of memory leaks lies in its internal implementation mechanism.

From the above content, we already know that each thread maintains a `ThreadLocalMap` map called . When you use `ThreadLocal` to store a value, you actually store the value in the of the current thread `ThreadLocalMap` , where `ThreadLocal` the instance itself is the key and the value you want to store is the value.

`ThreadLocal` The method `set()` source code is as follows:

```java
public void set(T value) {
    Thread t = Thread.currentThread(); //
    ThreadLocalMap map = getMap(t);   //
ThreadLocalMap
    if (map != null) {
        map.set(this, value);         //
    } else {
        createMap(t, value);          //         ThreadLocalMap
    }
}
```

`ThreadLocalMap` `set()` In the and methods of `createMap()` , the object itself is not stored directly `ThreadLocal` . Instead, `ThreadLocal` the array index is calculated using the hash value of , and is ultimately stored in `static class Entry extends WeakReference<ThreadLocal<?>>` an array of type .

```java
int i = key.threadLocalHashCode & (len-1);
```

`ThreadLocalMap` The definition of `Entry` is as follows:

```java
static class Entry extends WeakReference<ThreadLocal<?>> {
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
```

`ThreadLocalMap` The `key` and `value` reference mechanisms:

- **key is a weak reference** : `ThreadLocalMap` The key in is `ThreadLocal` a weak reference (`WeakReference<ThreadLocal<?>>`) of . This means that if `ThreadLocal` the instance is no longer pointed to by any strong reference, the garbage

collector will reclaim the instance at the next GC, causing `ThreadLocalMap` the corresponding key in to become `null` .

- **value is a strong reference** : even if `key` it is reclaimed by GC, it `value` still `ThreadLocalMap.Entry` exists by strong reference and cannot be reclaimed by GC.

`ThreadLocal` value `ThreadLocalMap` `Entry` `ThreadLocalMap` key `null` entry

1. `ThreadLocal`
2. `ThreadLocalMap`

`ThreadLocalMap` `get()` , `set()` `remove()` key null entry

1. `ThreadLocal` `remove()` `remove()` `ThreadLocalMap` entry `ThreadLocal` `static final` `remove()`
2. `try-finally` `remove()`

ThreadLocal

`ThreadLocal` `Thread` `Thread` `ThreadLocal`

`ThreadLocal`

- `InheritableThreadLocal` `InheritableThreadLocal` JDK1.2 `ThreadLocal` `InheritableThreadLocal` `ThreadLocal` `ThreadLocal`
- `TransmittableThreadLocal` `TransmittableThreadLocal` TTL `InheritableThreadLocal` `ThreadLocal` https://github.com/alibaba/transmittable-thread-local

InheritableThreadLocal

InheritableThreadLocal ThreadLocal
JDK JDK Thread
ThreadLocal

InheritableThreadLocal

Thread ThreadLocalMap
inheritableThreadLocals ThreadLocal

```java
class Thread implements Runnable {
    ThreadLocal.ThreadLocalMap threadLocals = null;
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
}
```

ThreadLocal

Thread Thread
inheritableThreadLocals

```java
// Thread init()
private void init(/* ... */) {
    // 1
    Thread parent = currentThread();
    // 2 inheritableThreadLocals
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =

ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
}
```

TransmittableThreadLocal

JDK ThreadLocal
TransmittableThreadLocal

JDK

ThreadLocal

TTL

- Thread run() ThreadLocal
- execute() JDK Thread Thread

Maven

```xml
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>transmittable-thread-local</artifactId>
    <version>2.12.0</version>
</dependency>
```

1. ThreadLocal
2. Trace ID

?

⭐

HTTP

1.

2. “ ”

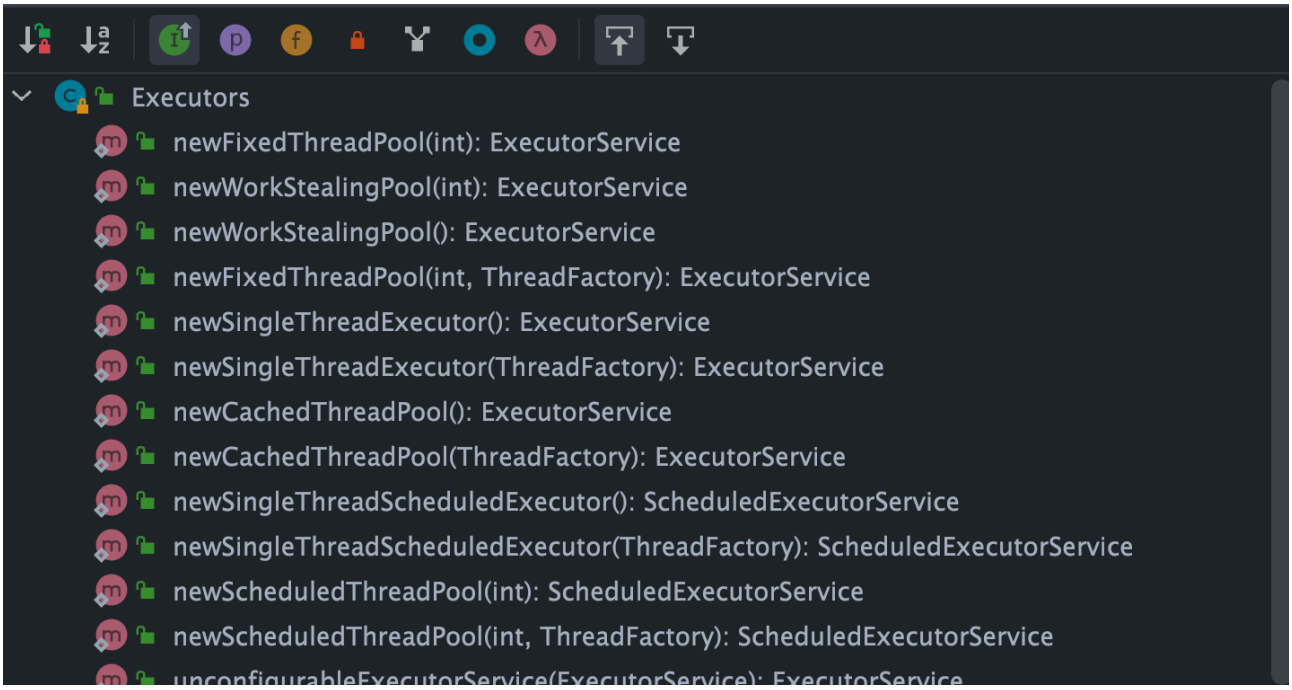
3.

Java

ThreadPoolExecutor ( )

Executors ( )

Executors



Executors

- FixedThreadPool
- SingleThreadExecutor
- CachedThreadPool
- ScheduledThreadPool

⭐

Java “ ”

> “ ”

Java Executors
ThreadPoolExecutor

Executors ( )

- FixedThreadPool SingleThreadExecutor :
  LinkedBlockingQueue Integer.MAX_VALUE
  OOM
- CachedThreadPool : SynchronousQueue ,
  Integer.MAX_VALUE
  OOM
- ScheduledThreadPool SingleThreadScheduledExecutor :
  DelayedWorkQueue Integer.MAX_VALUE ,
  OOM

```java
public static ExecutorService newFixedThreadPool(int nThreads) {
    // LinkedBlockingQueue Integer.MAX_VALUE

    return new ThreadPoolExecutor(nThreads, nThreads,0L,
TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>());

}

public static ExecutorService newSingleThreadExecutor() {
    // LinkedBlockingQueue Integer.MAX_VALUE

    return new FinalizableDelegatedExecutorService (new
ThreadPoolExecutor(1, 1,0L, TimeUnit.MILLISECONDS,new
LinkedBlockingQueue<Runnable>()));

}

//  SynchronousQueue Integer.MAX_VALUE`
public static ExecutorService newCachedThreadPool() {

    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,60L,
TimeUnit.SECONDS,new SynchronousQueue<Runnable>());

}

// DelayedWorkQueue
```

```
27  public static ScheduledExecutorService newScheduledThreadPool(int
    corePoolSize) {
        return new ScheduledThreadPoolExecutor(corePoolSize);
    }
    public ScheduledThreadPoolExecutor(int corePoolSize) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
              new DelayedWorkQueue());
    }
```

java

```
1   /**
2    *                        ThreadPoolExecutor
3    */
4   public ThreadPoolExecutor(int corePoolSize,//
5                             int maximumPoolSize,//
6                             long keepAliveTime,//
7
8                             TimeUnit unit,//
9                             BlockingQueue<Runnable> workQueue,//
10
11                            ThreadFactory threadFactory,//
12
13                            RejectedExecutionHandler handler//
14
15                             ) {
16      if (corePoolSize < 0 ||
17          maximumPoolSize <= 0 ||
18          maximumPoolSize < corePoolSize ||
19          keepAliveTime < 0)
20          throw new IllegalArgumentException();
21      if (workQueue == null || threadFactory == null || handler
    == null)
22          throw new NullPointerException();
23      this.corePoolSize = corePoolSize;
24      this.maximumPoolSize = maximumPoolSize;
25      this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

ThreadPoolExecutor 3

- corePoolSize :
- maximumPoolSize :

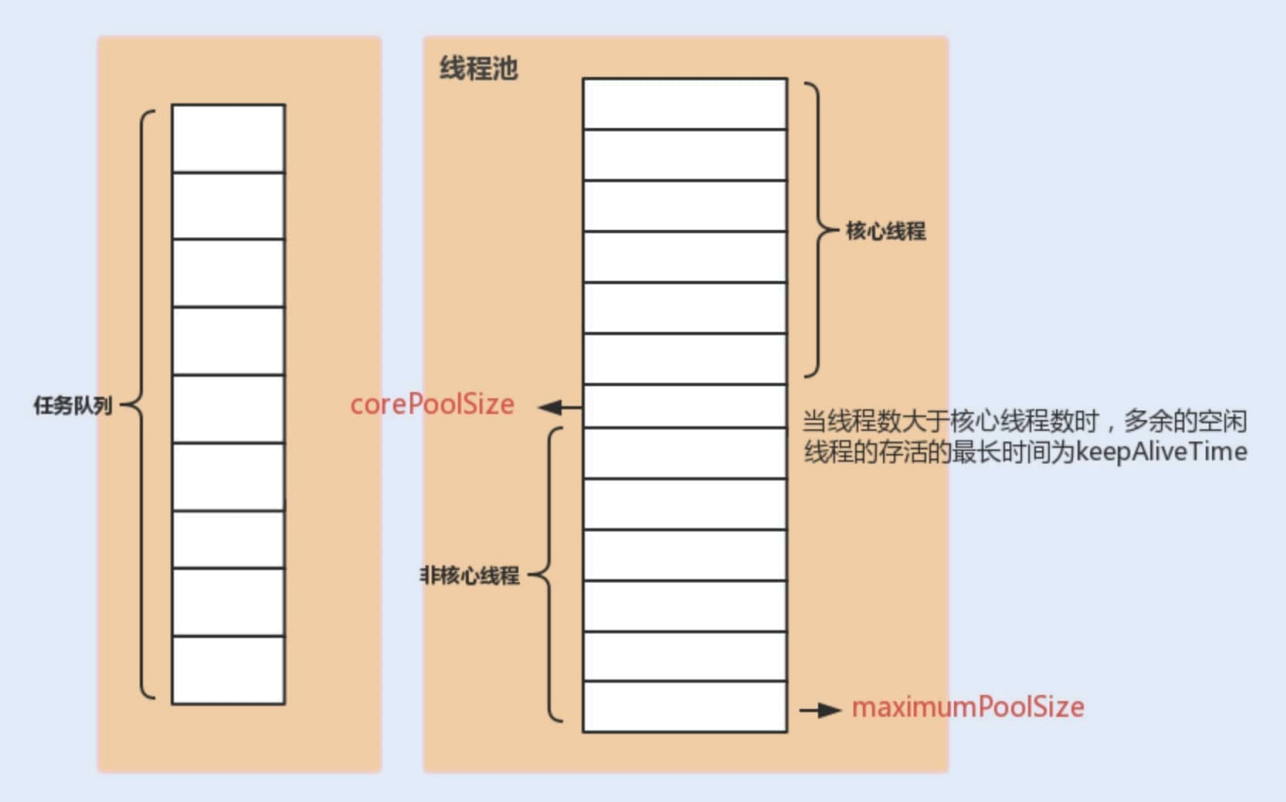
- workQueue :

ThreadPoolExecutor :

- keepAliveTime : corePoolSize

keepAliveTime
- unit : keepAliveTime
- threadFactory :executor
- handler :

Java



ThreadPoolExecutor

allowCoreThreadTimeOut(boolean value) true
keepAliveTime

```java
public void allowCoreThreadTimeOut(boolean value) {
    //          keepAliveTime         0
    if (value && keepAliveTime <= 0) {
        throw new IllegalArgumentException("Core threads must have
nonzero keep alive times");
    }
    //     allowCoreThreadTimeOut
    if (value != allowCoreThreadTimeOut) {
        allowCoreThreadTimeOut = value;
        //
        if (value) {
            interruptIdleWorkers();
        }
    }
}
```

- WAITING
  TERMINATED
- WAITING
  WAITING
  RUNNABLE

Worker Worker

timed BlockingQueue

timed true poll()

- timed == true poll(timeout, unit) poll(timeout, unit) TERMINATED
- timed == false take() take() WAITING

```java
// ThreadPoolExecutor
private Runnable getTask() {
    boolean timedOut = false;
    for (;;) {
        // ...

        // 1
          timed    true
        boolean timed = allowCoreThreadTimeOut || wc >
corePoolSize;
        // 2
        // wc > maximuimPoolSize
   wc
        // timed && timeOut  timeOut
        //                                  &&
                              &&
        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }
        try {
            // 3      timed    true        poll()
take()
            Runnable r = timed ?
```

```
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)
:
                workQueue.take();
            // 4
            if (r != null)
                return r;
            timedOut = true;
        } catch (InterruptedException retry) {
            timedOut = false;
        }
    }
}
```

ThreadPoolExecutor :

- ThreadPoolExecutor.AbortPolicy RejectedExecutionException
- ThreadPoolExecutor.CallerRunsPolicy
  execute (run)

- ThreadPoolExecutor.DiscardPolicy
- ThreadPoolExecutor.DiscardOldestPolicy

Spring ThreadPoolTaskExecutor
ThreadPoolExecutor
RejectedExecutionHandler
AbortPolicy ThreadPoolExecutor
RejectedExecutionException
CallerRunsPolicy CallerRunsPolicy

```java
public static class CallerRunsPolicy implements
RejectedExecutionHandler {

        public CallerRunsPolicy() { }

        public void rejectedExecution(Runnable r,
ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                //
                r.run();
            }
        }
    }
```

CallerRunsPolicy

CallerRunsPolicy

```java
public static class CallerRunsPolicy implements
RejectedExecutionHandler {

        public CallerRunsPolicy() { }


        public void rejectedExecution(Runnable r,
ThreadPoolExecutor e) {
            //                          execute
            if (!e.isShutdown()) {

                r.run();
            }
        }
    }
```

execute

CallerRunsPolicy

CallerRunsPolicy

CallerRunsPolicy

2 1(4) ThreadUtil Hutool

```java
public class ThreadPoolTest {

    private static final Logger log = LoggerFactory.getLogger(ThreadPoolTest.class);

    public static void main(String[] args) {
        //                          1           2
        //                                                  60
        //              1  ArrayBlockingQueue
CallerRunsPolicy
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(1,
                2,
                60,
                TimeUnit.SECONDS,
                new ArrayBlockingQueue<>(1),
                new ThreadPoolExecutor.CallerRunsPolicy());

        //
        threadPoolExecutor.execute(() -> {
            log.info("                    ");
            ThreadUtil.sleep(1, TimeUnit.MINUTES);
        });

        //
        threadPoolExecutor.execute(() -> {
            log.info("                          ");
            ThreadUtil.sleep(1, TimeUnit.MINUTES);
        });
```

```
        //
        threadPoolExecutor.execute(() -> {
            log.info("                    ");
            ThreadUtil.sleep(1, TimeUnit.MINUTES);
        });

        //
CallerRunsPolicy
        threadPoolExecutor.execute(() -> {
            log.info("                 ");
            ThreadUtil.sleep(2, TimeUnit.MINUTES);
        });

        //

        threadPoolExecutor.execute(() -> {
            log.info("                   ");
        });

        //
        threadPoolExecutor.shutdown();
    }
}
```

bash

```
18:19:48.203 INFO  [pool-1-thread-1] c.j.concurrent.ThreadPoolTest
-
18:19:48.203 INFO  [pool-1-thread-2] c.j.concurrent.ThreadPoolTest
-
18:19:48.203 INFO  [main] c.j.concurrent.ThreadPoolTest -

18:20:48.212 INFO  [pool-1-thread-2] c.j.concurrent.ThreadPoolTest
-
18:21:48.219 INFO  [pool-1-thread-2] c.j.concurrent.ThreadPoolTest
-
```

CallerRunsPolicy

OOM

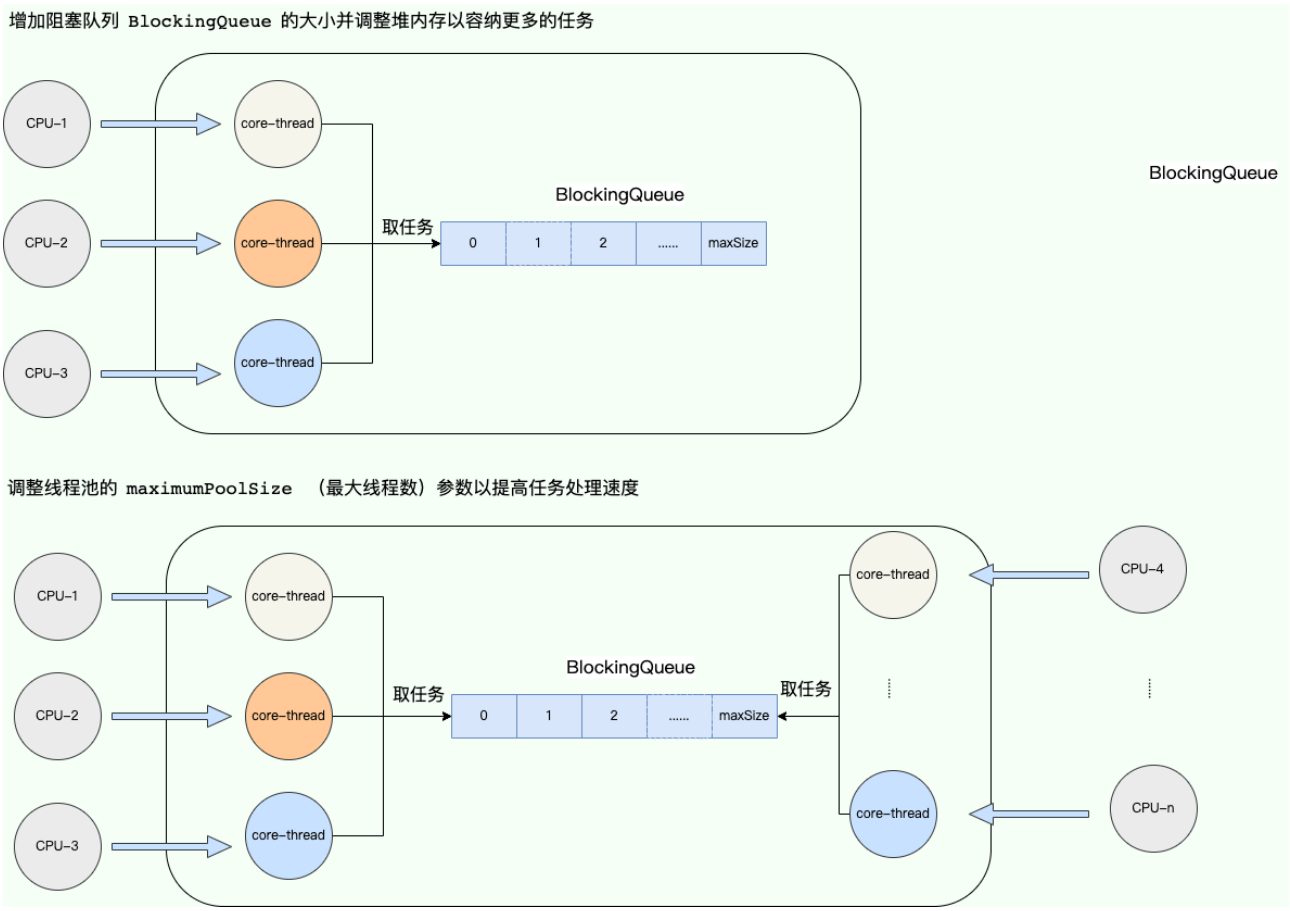
`CallerRunsPolicy`

`BlockingQueue`

`BlockingQueue`

CPU `maximumPoolSize`

`BlockingQueue`



:

1. MySQL
2. Redis
3.

1. `RejectedExecutionHandler`

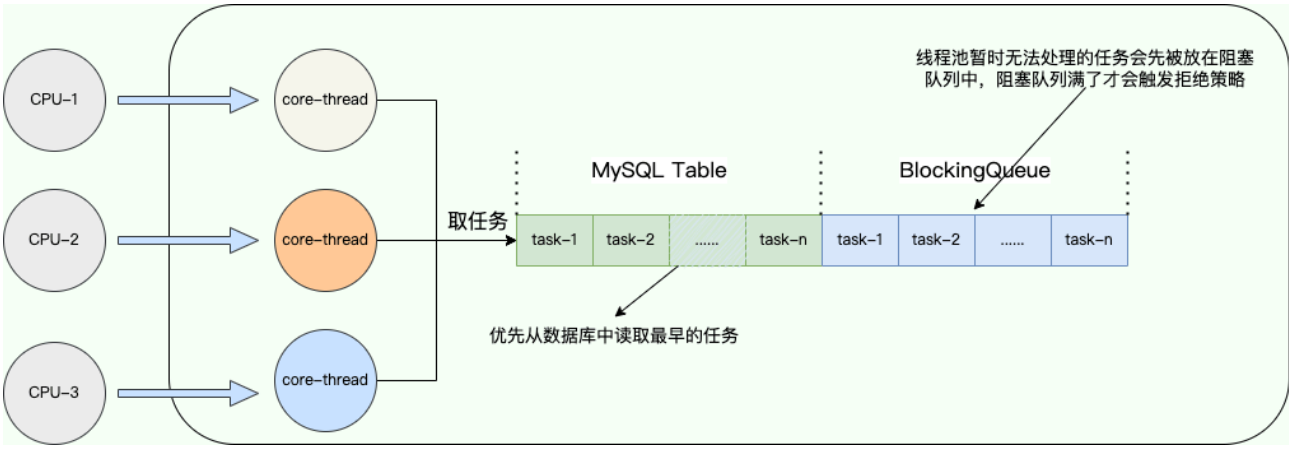
MySQL

2. BlockingQueue JDK

ArrayBlockingQueue

take()

ArrayBlockingQueue



MySQL

" "

Netty

```java
private static final class NewThreadRunsPolicy implements
RejectedExecutionHandler {
    NewThreadRunsPolicy() {
        super();
    }
    public void rejectedExecution(Runnable r, ThreadPoolExecutor
executor) {
        try {
            //
            final Thread t = new Thread(r, "Temporary task
executor");
            t.start();
        } catch (Throwable e) {
```

```
            throw new RejectedExecutionException(
                    "Failed to start a new thread", e);
        }
    }
}
```

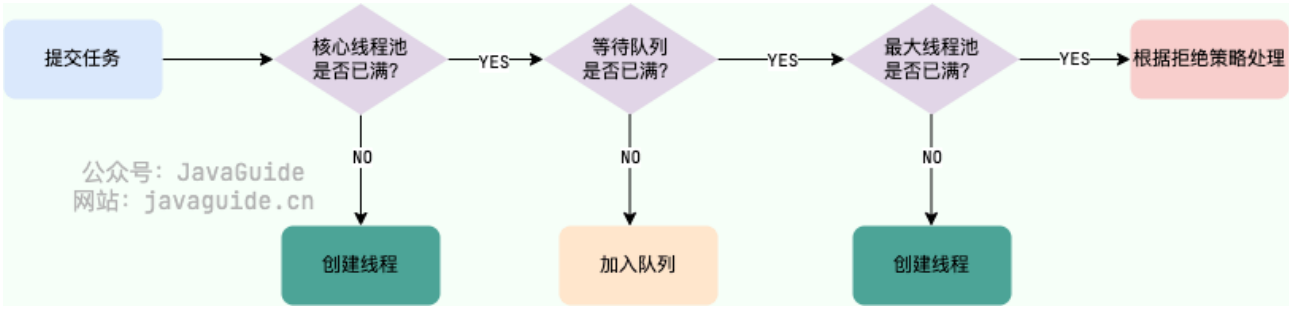
ActiveMQ

```java
new RejectedExecutionHandler() {
                @Override
                public void rejectedExecution(final Runnable r,
final ThreadPoolExecutor executor) {
                    try {
                        //
                        executor.getQueue().offer(r, 60,
TimeUnit.SECONDS);
                    } catch (InterruptedException e) {
                        throw new
RejectedExecutionException("Interrupted waiting for
BrokerService.worker");
                    }
                    throw new RejectedExecutionException("Timed Out
while attempting to enqueue Task.");
                }
            });
```

- Integer.MAX_VALUE LinkedBlockingQueue FixedThreadPool SingleThreadExecutor FixedThreadPool SingleThreadExecutor 1
- SynchronousQueue CachedThreadPool SynchronousQueue CachedThreadPool

Integer.MAX_VALUE
OOM

- DelayedWorkQueue ScheduledThreadPool
  SingleThreadScheduledExecutor DelayedWorkQueue
  “ ”
  DelayedWorkQueue
  50%
  Integer.MAX_VALUE
- ArrayBlockingQueue



1.
2.
3.
4.

RejectedExecutionHandler.rejectedExecution()

ThreadPoolExecutor

- prestartCoreThread() :
  false true
- prestartAllCoreThreads() :

⭐

- **execute()** execute()

- **submit()** submit()
  submit() Future
  Future.get() ExecutionException

execute()
submit() Future

submit()
execute()

⭐

pool-1-thread-n

1 guava ThreadFactoryBuilder

```java
ThreadFactory threadFactory = new ThreadFactoryBuilder()
                        .setNameFormat(threadNamePrefix + "-%d")
                        .setDaemon(true).build();
ExecutorService threadPool = new ThreadPoolExecutor(corePoolSiz
maximumPoolSize, keepAliveTime, TimeUnit.MINUTES, workQueue,
threadFactory);
```

2 ThreadFactory

```java
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.atomic.AtomicInteger;

/**
 *
 */
public final class NamingThreadFactory implements ThreadFactory {

    private final AtomicInteger threadNum = new AtomicInteger();
    private final String name;

    /**
     *
     */
    public NamingThreadFactory(String name) {
        this.name = name;
    }

    @Override
    public Thread newThread(Runnable r) {
        Thread t = new Thread(r);
        t.setName(name + " [#" + threadNum.incrementAndGet() +
"]");
        return t;
    }
}
```

3 6

> CPU CPU
> CPU

> CPU
>
> CPU
> Linux Unix

- / / / OOM CPU
- CPU

- **CPU (N+1)** CPU N CPU +1 CPU CPU CPU
- **I/O (2N)** I/O I/O CPU CPU I/O 2N

CPU IO

CPU CPU IO CPU IO IO

> issue#1737
>
> = N CPU * 1+WT /ST WT = - ST

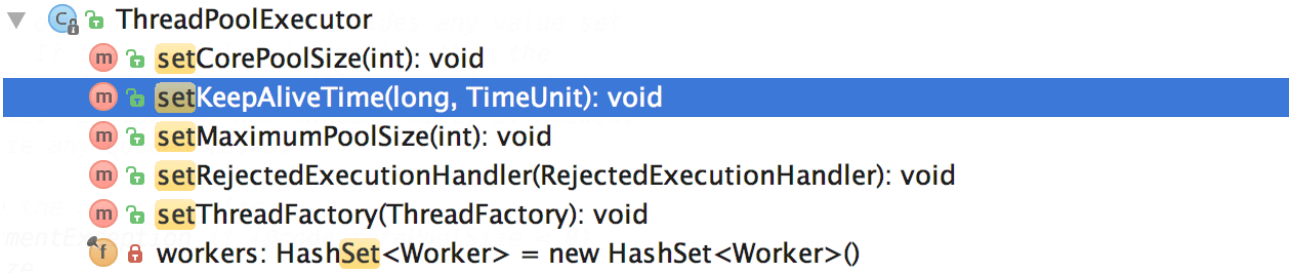
> JDK VisualVM WT/ST
> CPU WT/ST 0 N CPU
> * 1+0 = N N CPU +1
> IO
> 2N WT/ST 2N

Java

- **corePoolSize :**
- **maximumPoolSize :**
- **workQueue :**

Java ThreadPoolExecutor

ThreadPoolExecutor



corePoolSize
setCorePoolSize()
corePoolSize

`ResizableCapacityLinkedBlockIngQueue` `LinkedBlockingQueue`
capacity final

👏👏👏

&

- **Hippo4j** & &
- **Dynamic TP**

Nacos Apollo Zookeeper Consul Etcd

SPI

⭐

FixedThreadPool LinkedBlockingQueue
Integer.MAX_VALUE FixedThreadPool

PriorityBlockingQueue ThreadPoolExecutor
workQueue

```
/**
 * 用给定的初始参数创建一个新的ThreadPoolExecutor。
 */
public ThreadPoolExecutor(int corePoolSize,//线程池的核心线程数量
                          int maximumPoolSize,//线程池的最大线程数
                          long keepAliveTime,//当线程数大于核心线程数时，多余的空闲线程存活的最长时间
                          TimeUnit unit,//时间单位
                          BlockingQueue<Runnable> workQueue,//任务队列，用来储存等待执行任务的队列
                          ThreadFactory threadFactory,//线程工厂，用来创建线程，一般默认即可
                          RejectedExecutionHandler handler//拒绝策略，当提交的任务过多而不能及时处理
时，我们可以定制策略来处理任务
                           ) {
```

PriorityBlockingQueue
PriorityQueue
PriorityQueue

PriorityBlockingQueue

1. Comparable compareTo

2. PriorityBlockingQueue Comparator
( )

- PriorityBlockingQueue OOM

-
-
  `ReentrantLock`

  OOM `PriorityBlockingQueue`
  `offer` ( ) false

Future

`CompletableFuture`

CompletableFuture

Future

`Future`

`Future`

Future

Java

Java `Future` `java.util.concurrent`
5 4

-
- ;
- ;
-

```java
// V      Future
public interface Future<V> {
    //
    //            true         false
    boolean cancel(boolean mayInterruptIfRunning);
    //
    boolean isCancelled();
    //
    boolean isDone();
    //
    V get() throws InterruptedException, ExecutionException;
    //                              TimeOutException
    V get(long timeout, TimeUnit unit)

        throws InterruptedException, ExecutionException,
TimeoutExceptio

}
```

Future

Future

Callable Future

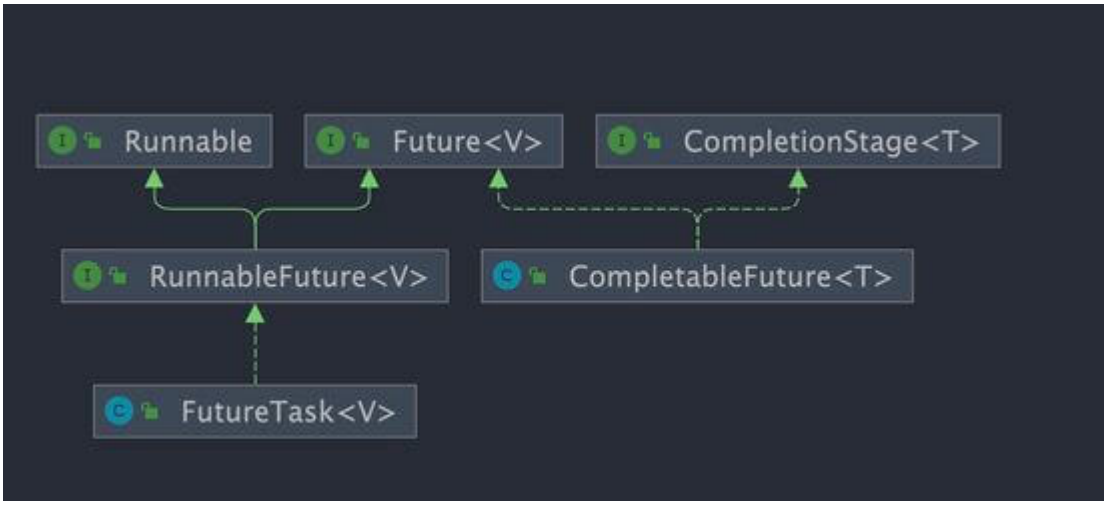
FutureTask Callable Future

FutureTask Future Callable Runnable

ExecutorService.submit() Future FutureTask

```java
<T> Future<T> submit(Callable<T> task);
Future<?> submit(Runnable task);
```

FutureTask Future Runnable



FutureTask Callable Runnable
Runnable Callable

```java
public FutureTask(Callable<V> callable) {
    if (callable == null)
        throw new NullPointerException();
    this.callable = callable;
    this.state = NEW;
}
public FutureTask(Runnable runnable, V result) {
    //         RunnableAdapter   Runnable   runnable     Callable

    this.callable = Executors.callable(runnable, result);
    this.state = NEW;
}
```

FutureTask Callable
Callable call

Future Java
Future

CompletableFuture

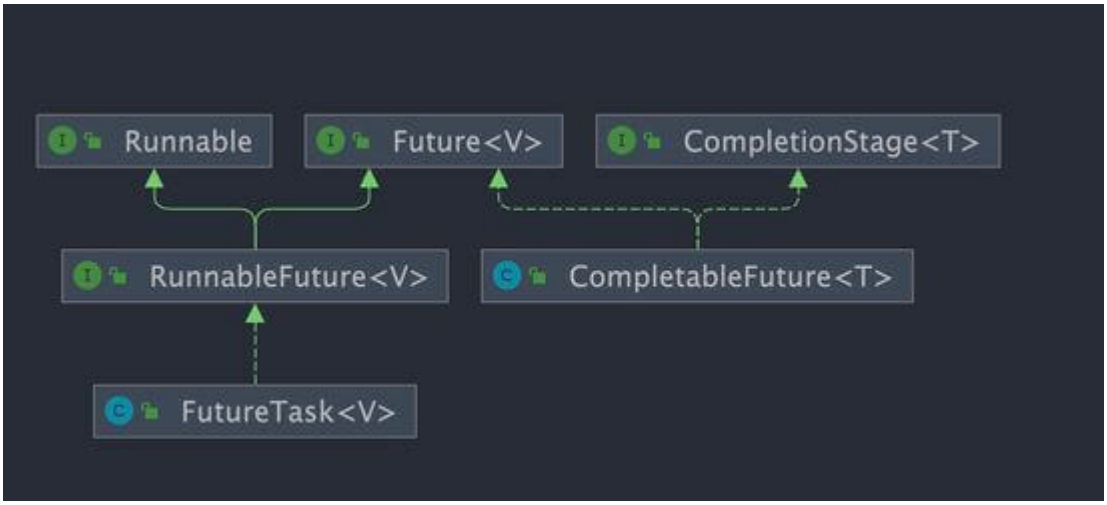
Future
get()

Java 8 CompletableFuture Future
CompletableFuture Future

CompletableFuture

```java
public class CompletableFuture<T> implements Future<T>,
CompletionStage<T> {
}
```

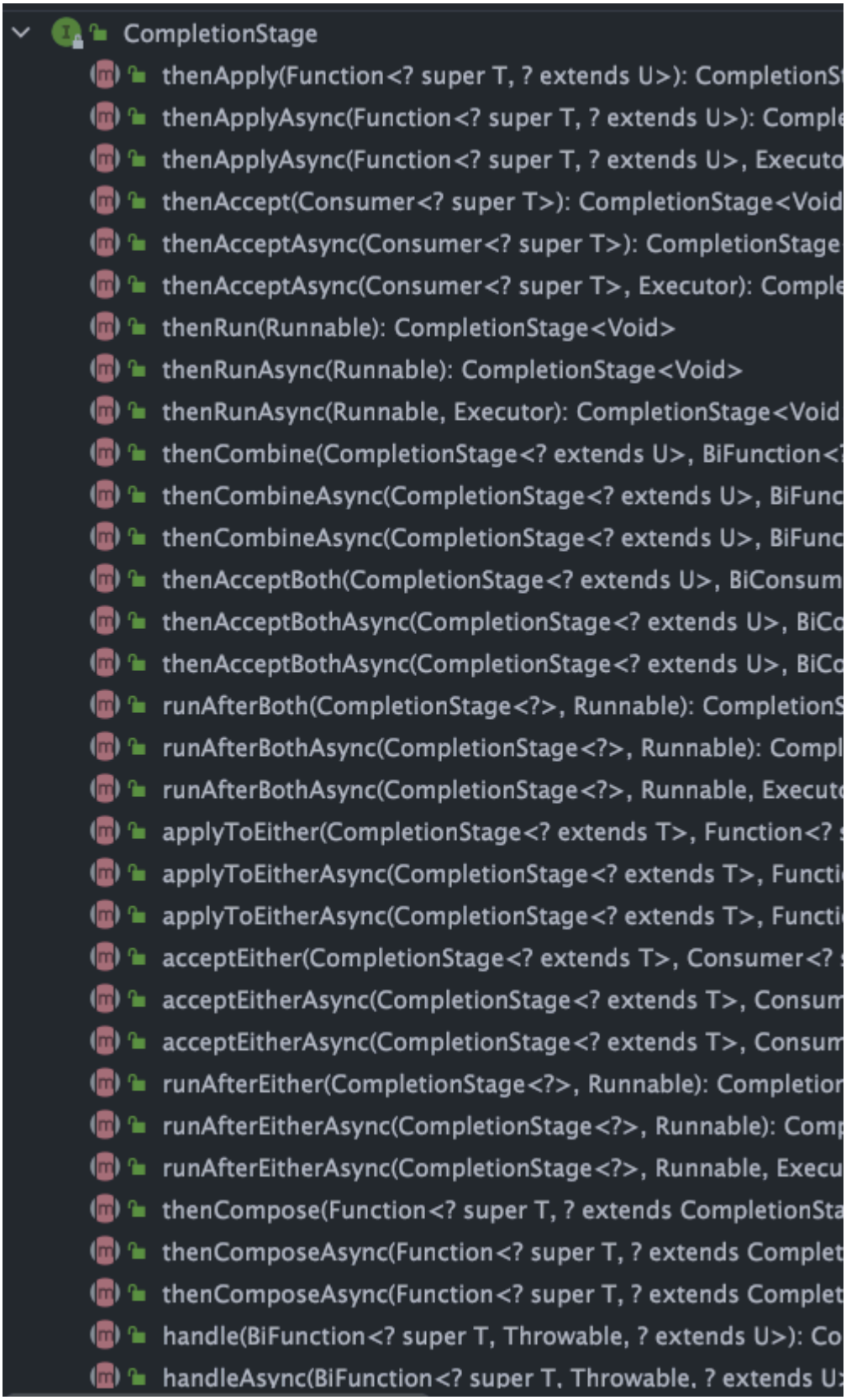
CompletableFuture Future CompletionStage



CompletionStage

CompletionStage CompletableFuture

Java8



`CompletableFuture` T3 T2 T1

Hutool ThreadUtil DateUtil

```java
// T1
CompletableFuture<Void> futureT1 = CompletableFuture.runAsync(() ->
{
    System.out.println("T1 is executing. Current time  " +
DateUtil.now());
    //
    ThreadUtil.sleep(1000);
});
// T2
CompletableFuture<Void> futureT2 = CompletableFuture.runAsync(() ->
{
    System.out.println("T2 is executing. Current time  " +
DateUtil.now());
    ThreadUtil.sleep(1000);
});

//    allOf()       T1  T2  CompletableFuture
CompletableFuture<Void> bothCompleted =
CompletableFuture.allOf(futureT1, futureT2);
//   T1  T2            T3
bothCompleted.thenRunAsync(() -> System.out.println("T3 is
executing after T1 and T2 have completed.Current time  " +
DateUtil.now()));
//
ThreadUtil.sleep(3000);
```

CompletableFuture allOf() T1 T2 T1 T2 T3

⭐ CompletableFuture

CompletableFuture

- whenComplete

- `exceptionally`
- `handle`
- `CompletableFuture.allOf` `CompletableFuture`
- ……

⭐ CompletableFuture

`CompletableFuture` `ForkJoinPool.commonPool()`

Spring `CompletableFuture`

`ForkJoinPool`

`CompletableFuture`

-
-
- `ThreadFactory`

```java
private ThreadPoolExecutor executor = new ThreadPoolExecutor(10, 10,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());

CompletableFuture.runAsync(() -> {
     //...
}, executor);
```

AQS

AQS AQS

AQS

AQS `AbstractQueuedSynchronizer` JDK1.5 Java

AQS

`ReentrantLock` `Semaphore` `CountDownLatch` AQS

AQS

AQS “ ” AQS “ ”

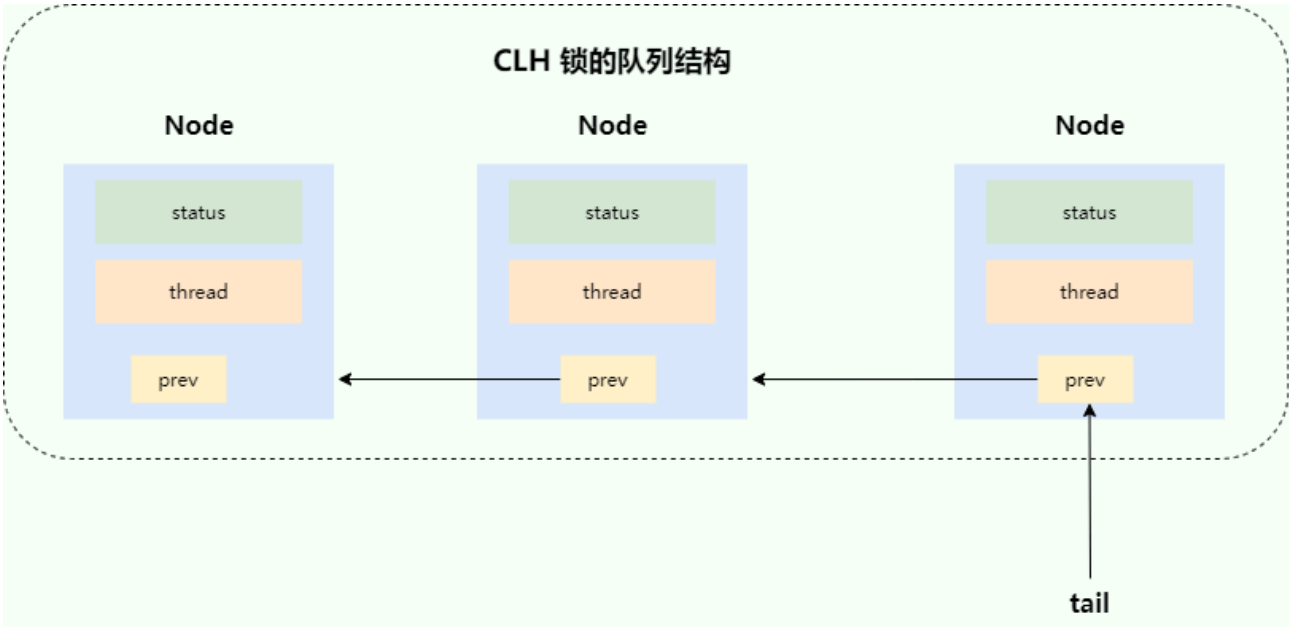
⭐AQS

AQS

AQS **CLH** Craig, Landin, and Hagersten locks

CLH

CLH



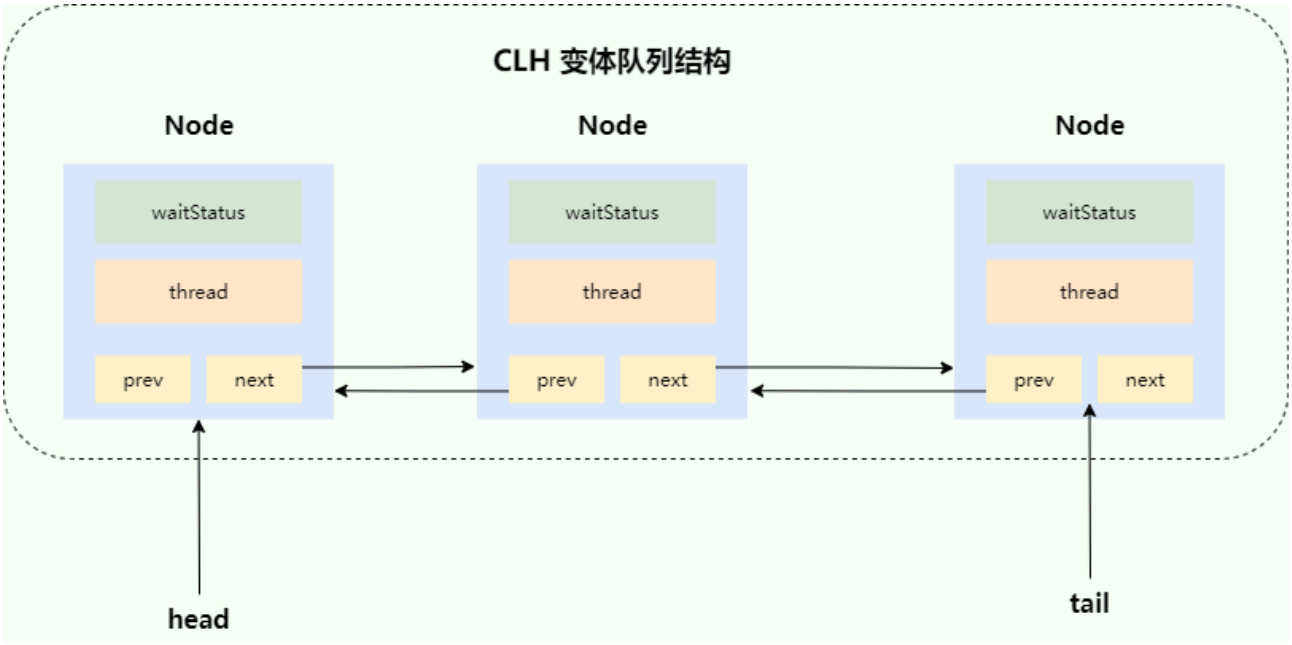
AQS CLH CLH

AQS CLH
CLH CLH

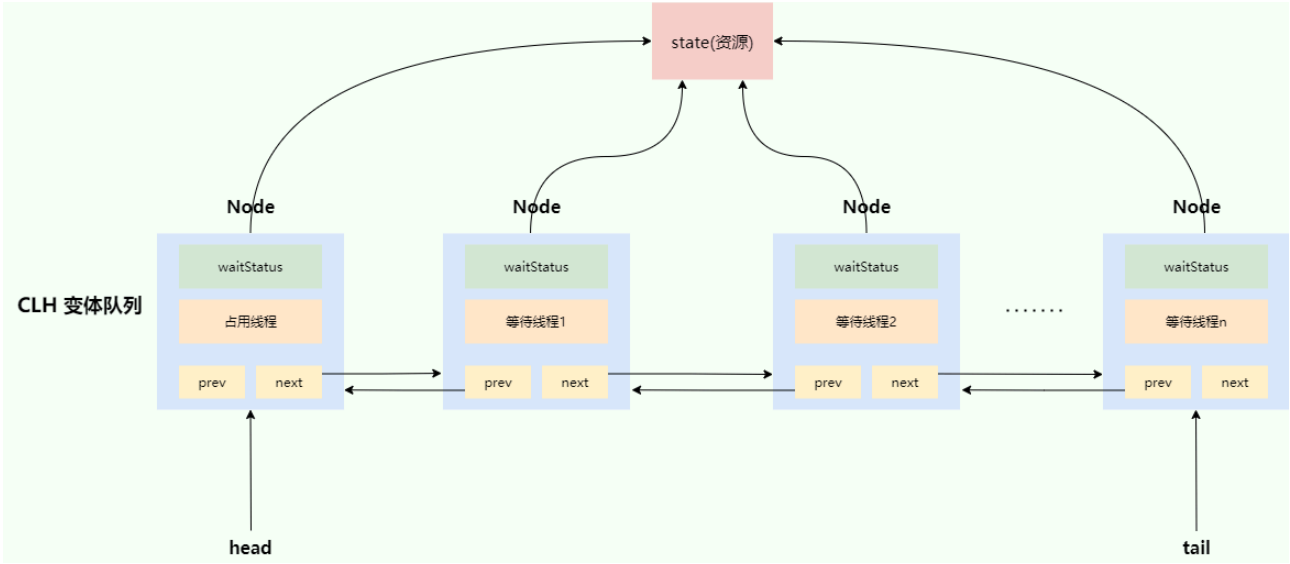
- + CPU
  CLH
- CLH
  next

AQS CLH Node
CLH thread
waitStatus prev next

AQS CLH



AQS(`AbstractQueuedSynchronizer`)



AQS **`int`** **`state`**

`state` `volatile`

```java
//             volatile
private volatile int state;
```

`state` `protected` `getState()` `setState()` `compareAndSetState()` `final`

```java
//
protected final int getState() {
     return state;
}
 //
protected final void setState(int newState) {
     state = newState;
}
//       CAS                          update
expect
protected final boolean compareAndSetState(int expect, int update)
{
      return unsafe.compareAndSwapInt(this, stateOffset, expect,
update);
}
```

ReentrantLock state 0 A lock() tryAcquire() state+1 tryAcquire() A unlock() state= 0 A state state

CountDownLatch N state N N N countDown() state CAS(Compare and Swap) 1 (state=0) unpark() await()

Semaphore

synchronized ReentrantLock Semaphore ( )

Semaphore N(N>5) Semaphore N 5

```java
//
final Semaphore semaphore = new Semaphore(5);
//     1
semaphore.acquire();
//     1
semaphore.release();
```

1 Semaphore

Semaphore

- acquire() FIFO
-

Semaphore

```java
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new NonfairSync(permits);
}
```

Semaphore

Redis +Lua

Semaphore

Semaphore AQS state permits
permits

semaphore.acquire() state >= 0
CAS state state=state-1
state<0 Node

```java
/**
 *     1
 */
public void acquire() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}
/**
 *
 */
public final void acquireSharedInterruptibly(int arg)
    throws InterruptedException {
    if (Thread.interrupted())
      throw new InterruptedException();
        //              arg
            0,
    if (tryAcquireShared(arg) < 0)
      doAcquireSharedInterruptibly(arg);
}
```

semaphore.release(); CAS state state=state+1

state state=state-1 state>=0

```java
//
public void release() {
    sync.releaseShared(1);
}

//
public final boolean releaseShared(int arg) {
    //
    if (tryReleaseShared(arg)) {
      //
      doReleaseShared();
      return true;
    }
    return false;
}
```

CountDownLatch

CountDownLatch count

CountDownLatch
CountDownLatch

CountDownLatch

CountDownLatch , AQS state count
countDown() , tryReleaseShared CAS
state , state o await() state o
await() await()
count countDown() state o
await() await()

CountDownLatch

CountDownLatch count

CountDownLatch

6 6

count 6 CountDownLatch
count-1 CountDownLatch await()

```java
public class CountDownLatchExample1 {
    //
    private static final int threadCount = 6;

    public static void main(String[] args) throws
```

```
InterruptedException {
        //
        ExecutorService threadPool =
Executors.newFixedThreadPool(10);
        final CountDownLatch countDownLatch = new
CountDownLatch(threadCount);
        for (int i = 0; i < threadCount; i++) {
            final int threadnum = i;
            threadPool.execute(() -> {
                try {
                    //
                    //......
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    //
                    countDownLatch.countDown();
                }

            });
        }
        countDownLatch.await();
        threadPool.shutdown();
        System.out.println("finish");
    }
}
```

CompletableFuture Java8 CompletableFuture

java

```
CompletableFuture<Void> task1 =
    CompletableFuture.supplyAsync(()->{
        //
    });
......
CompletableFuture<Void> task6 =
    CompletableFuture.supplyAsync(()->{
    //
    });
......
```

```
CompletableFuture<Void>
headerFuture=CompletableFuture.allOf(task1,.....,task6);

try {
    headerFuture.join();
} catch (Exception ex) {
    //......
}
System.out.println("all done. ");
```

task

java

```
//
List<String> filePaths = Arrays.asList(...)
//
List<CompletableFuture<String>> fileFutures = filePaths.stream()
    .map(filePath -> doSomeThing(filePath))
    .collect(Collectors.toList());
//
CompletableFuture<Void> allFutures = CompletableFuture.allOf(
    fileFutures.toArray(new CompletableFuture[fileFutures.size()])
);
```

CyclicBarrier

CyclicBarrier CountDownLatch

CountDownLatch CountDownLatch

> CountDownLatch AQS CyclicBarrier
> ReentrantLock (ReentrantLock AQS) Condition

CyclicBarrier Cyclic Barrier

CyclicBarrier

CyclicBarrier count count parties 1 count 0

```java
//
private final int parties;
//
private int count;
```

1 CyclicBarrier CyclicBarrier(int parties) await() CyclicBarrier

```java
public CyclicBarrier(int parties) {
    this(parties, null);
}

public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
    this.parties = parties;
    this.count = parties;
    this.barrierCommand = barrierAction;
}
```

parties

2 CyclicBarrier await() dowait(false, 0L) await() parties

```java
public int await() throws InterruptedException,
BrokenBarrierException {
  try {
      return dowait(false, 0L);
  } catch (TimeoutException toe) {
      throw new Error(toe); // cannot happen
  }
}
```

dowait(false, 0L)

```java
    //                        count    await
        count         5
    private int count;
    /**
     * Main barrier code, covering the various policies.
     */
    private int dowait(boolean timed, long nanos)
        throws InterruptedException, BrokenBarrierException,
               TimeoutException {
        final ReentrantLock lock = this.lock;
        //
        lock.lock();
        try {
            final Generation g = generation;

            if (g.broken)
                throw new BrokenBarrierException();

            //
            if (Thread.interrupted()) {
                breakBarrier();
                throw new InterruptedException();
            }
            // cout  1
            int index = --count;
            //    count         0
              await
            if (index == 0) {  // tripped
                boolean ranAction = false;
                try {
                    final Runnable command = barrierCommand;
```

```
                    if (command != null)
                        command.run();
                    ranAction = true;
                    //    count       parties
                    //
                    //
                    nextGeneration();
                    return 0;
                } finally {
                    if (!ranAction)
                        breakBarrier();
                }
            }

            // loop until tripped, broken, interrupted, or timed
out
            for (;;) {
                try {
                    if (!timed)
                        trip.await();
                    else if (nanos > 0L)
                        nanos = trip.awaitNanos(nanos);
                } catch (InterruptedException ie) {
                    if (g == generation && ! g.broken) {
                        breakBarrier();
                        throw ie;
                    } else {
                        // We're about to finish waiting even if we
had not
                        // been interrupted, so this interrupt is
deemed to
                        // "belong" to subsequent execution.
                        Thread.currentThread().interrupt();
                    }
                }

                if (g.broken)
                    throw new BrokenBarrierException();

                if (g != generation)
                    return index;

                if (timed && nanos <= 0L) {
```

```
                    breakBarrier();
                    throw new TimeoutException();
                }
            }
        } finally {
            lock.unlock();
        }
    }
```

Java 21

1.
2.
3.
4.
5.

- Java
- Java
- Java :
  https://mp.weixin.qq.com/s/icrrxEsbABBvEU0Gym7D5Q
- SynchronousQueue
  https://juejin.cn/post/7031196740128768037
- — DelayedWorkQueue https://zhuanlan.zhihu.com/p/310621485

- Java ——FutureTask/CompletableFuture
  https://www.cnblogs.com/iwehdio/p/14285282.html
- Java AQS https://www.cnblogs.com/waterystone/p/4920797.html
- Java -AQS
  https://www.cnblogs.com/chengxiao/archive/2017/07/24/7141160.html

2025/8/7 15:13

Contributors: SnailClimb , Farahani , halle , yellowgg , Ryze-Zhao , Snailclimb , shuang.kou , guide , Lshare , qiuyukang , pengchen211 , drlifeL , Tan Jiuding , 2293736867 , kaka2634 , chengcjk , HangdianGhostMr. , cxhello , WangjiaW , Curvature , Itswag , Evan He , JuiceApp1e , Verne.Chung , Raxcl , Guide , Mr.Hope , Nicolas , shikaibin , paigeman , OSrange , jun , viosay , zcx-666 , shark-chili , tim_zhangyu , qiliq , WindLYLY , suppered , xiaodongxu , 11 , Mister-Hope , suaxi , wenzhuo4657 , 26684 , flying-pig-z , wayne , Joycn2018