# Java并发编程与高并发解决方案

## 前期准备

### 1.1并发编程初体验

|  |
| --- |
| G:\concurrent\_programming>mvn -B archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DgroupId=com.byf -DartifactId=concurrent |

实现一个计数器

|  |
| --- |
| @Slf4j **public class** CountExample {  **private static int** *threadTotal* = 200;  **private static int** *clientTotal* = 5000;   **private static long** *count* = 0;   **public static void** main(String[] args) {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(*threadTotal*);  **for** (**int** index=0; index<*clientTotal*; index++){  exec.execute(() -> {  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (Exception e) {  ***log***.error(**"exception"**, e);  }  });  }  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  }   **private static void** add() {  *count*++;  } } |

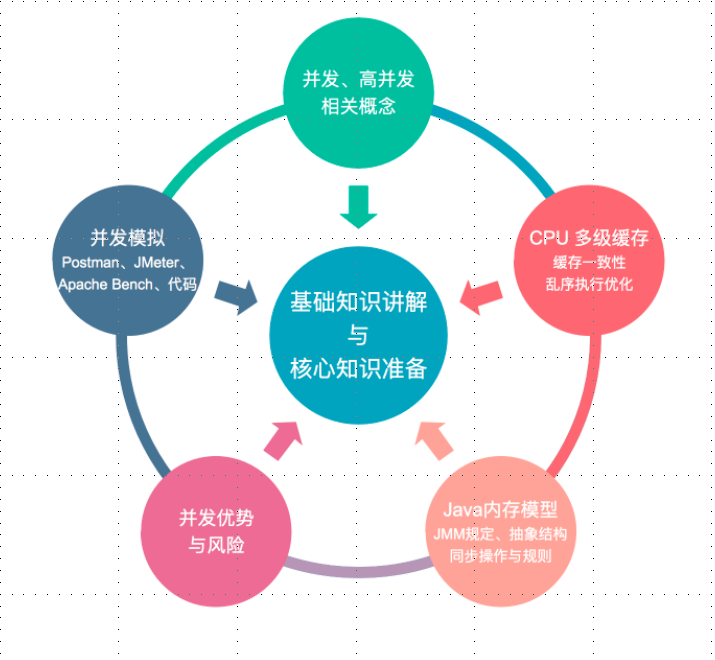
### 1.2并发与高并发基本概念

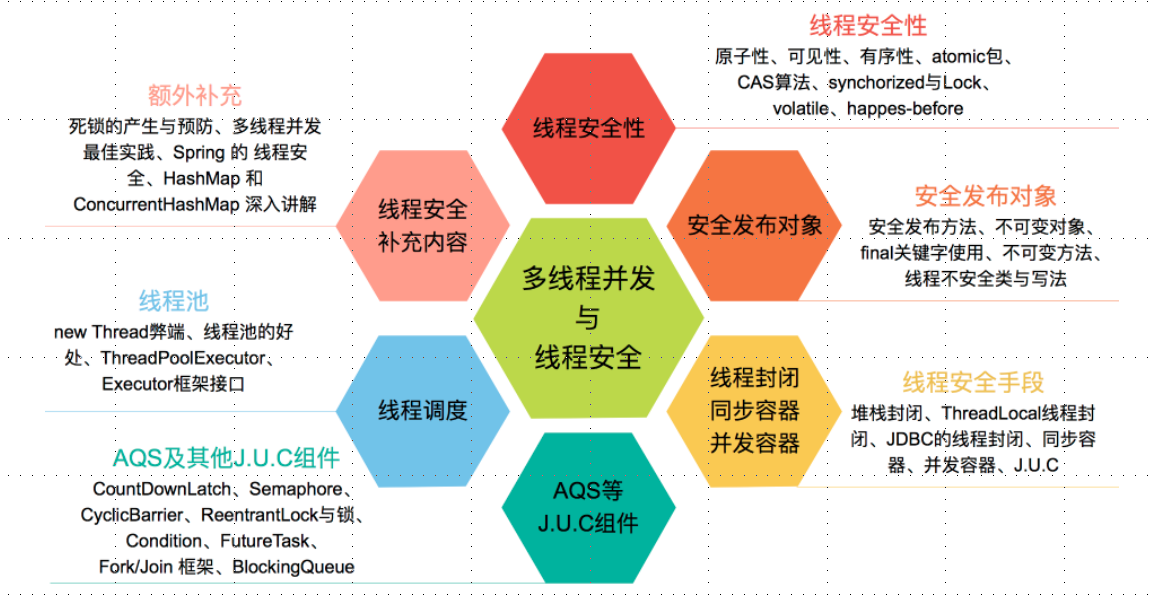
·并发：单核，多个线程交替执行；多核，每个线程分配到一个处理器。

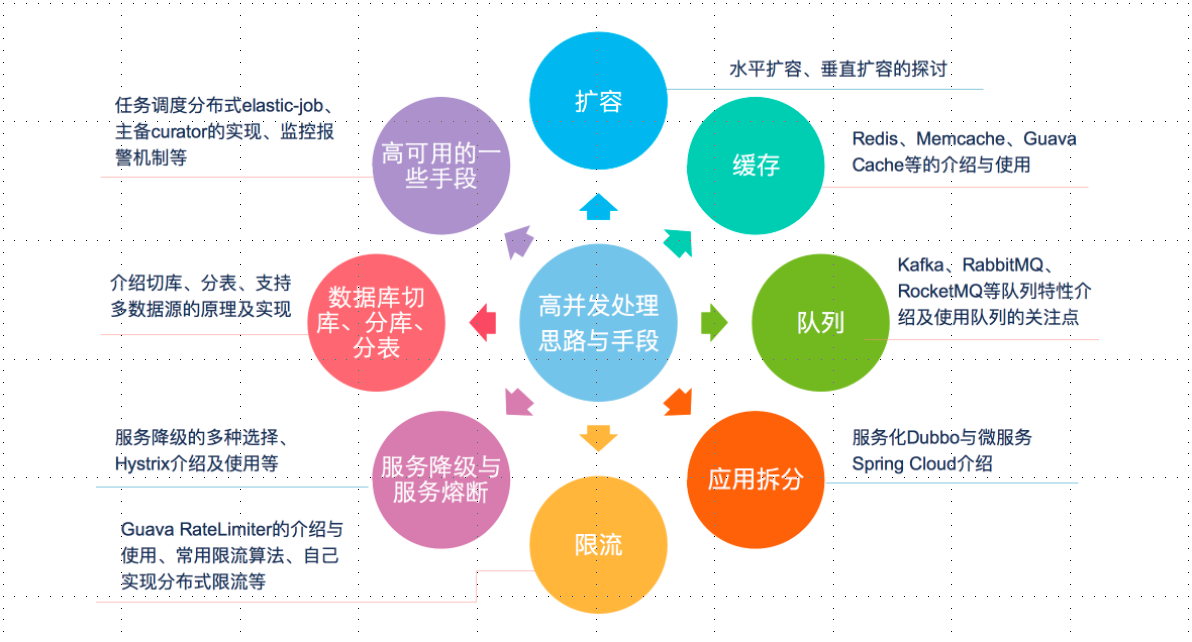
·高并发：互联网分布式架构设计必须考虑的因素，通常指，通过设计保证系统能够同时并行处理多个请求。

·并发：多个线程操作相同资源，保证线程安全，合理使用资源；

·高并发：服务能够同时处理很多请求，提高程序性能。



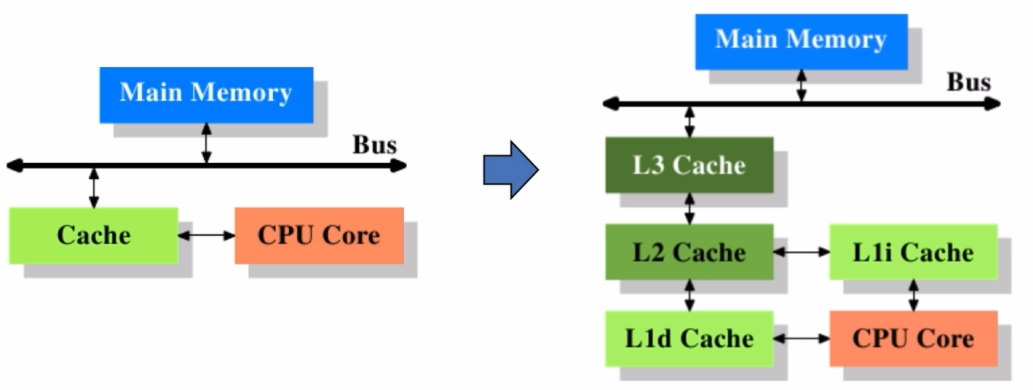


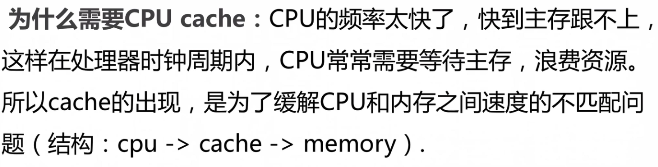


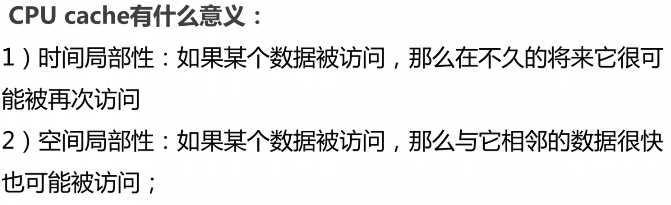


## 第2节并发基础

### 2.1CPU多级缓存--缓存一致性







·缓存一致性（MESI，Cache line的四种状态）

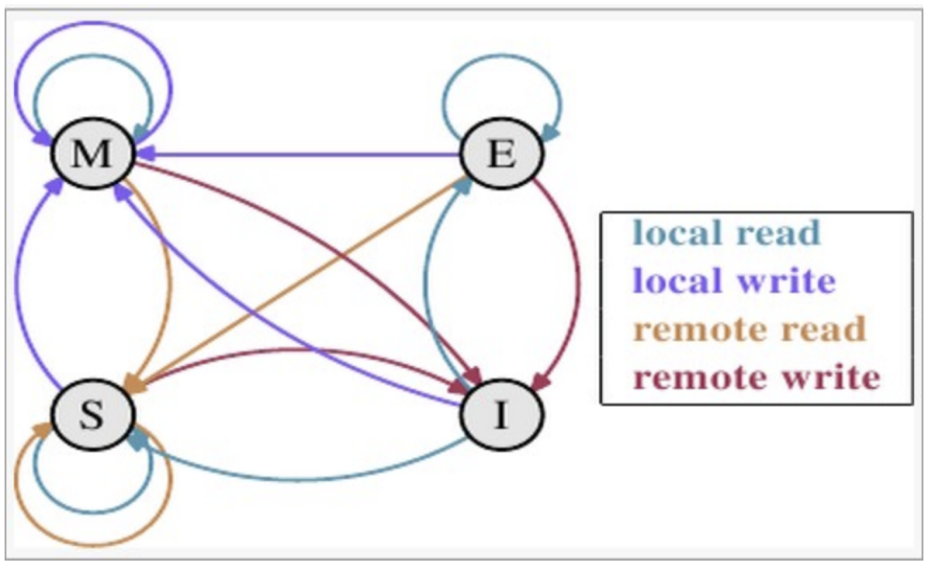
M：Modified，被修改

E：Exclusive，独享

S：Shared，共享

I：Invalid，无效

·CPU对缓存的四种操作可能会产生不一致的状态，因此缓存控制器监听到本地操作和远程操作，需要对地址一定的Cache line做出一定的修改。



M E S I

M × × × √

E × × × √

S × × √ √

I √ √ √ √

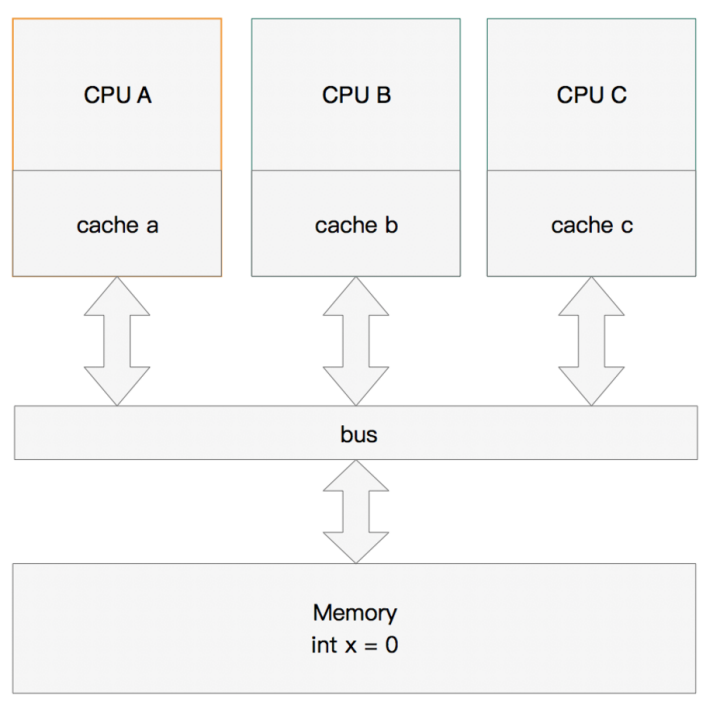
举个栗子来说：

假设cache 1 中有一个变量x = 0的cache line 处于S状态(共享)。

那么其他拥有x变量的cache 2、cache 3等x的cache line调整为S状态（共享）或者调整为 I 状态（无效）。

**多核缓存协同操作**

假设有三个CPU A、B、C，对应三个缓存分别是cache a、b、 c。在主内存中定义了x的引用值为0。

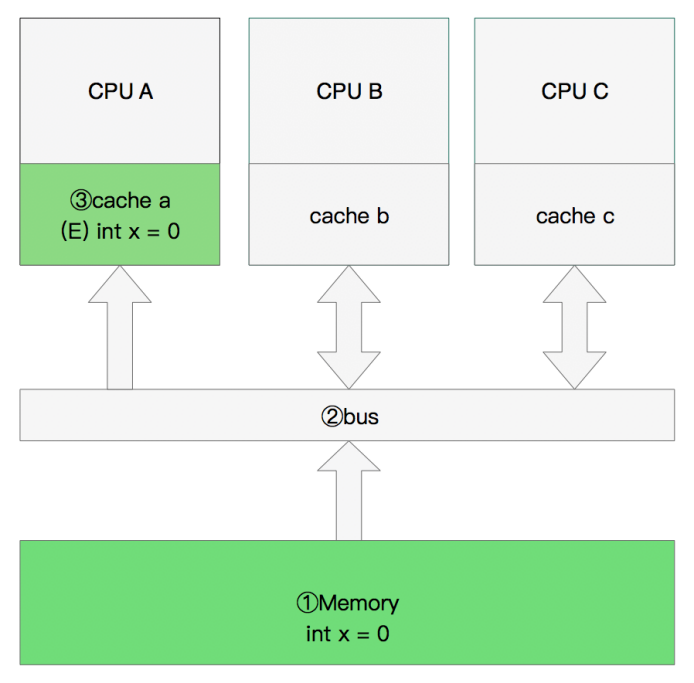


**单核读取**

那么执行流程是：

CPU A发出了一条指令，从主内存中读取x。

从主内存通过bus读取到缓存中（远端读取Remote read）,这是该Cache line修改为E状态（独享）.



**双核读取**

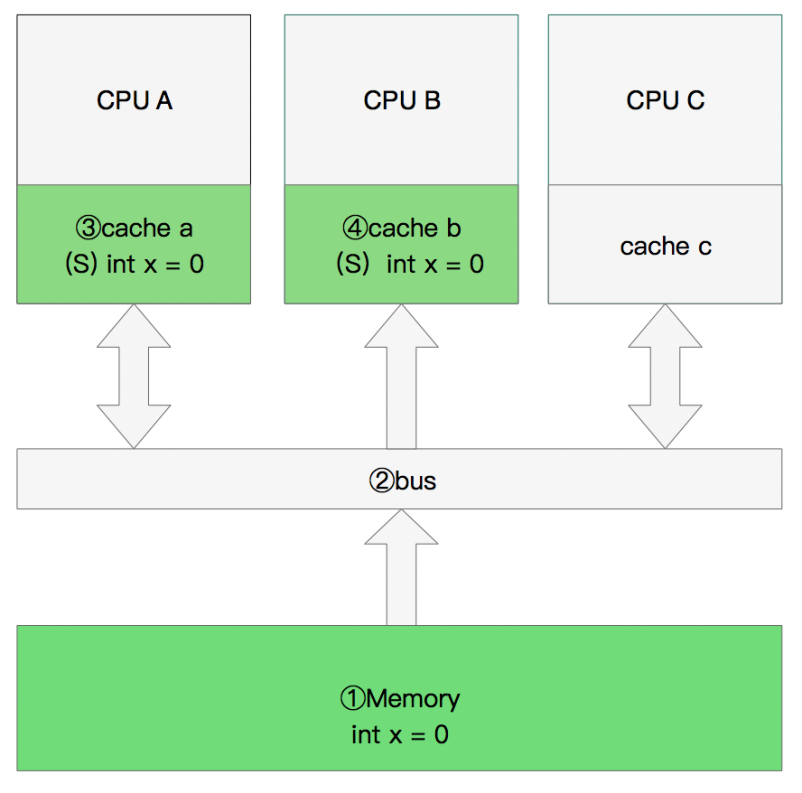
那么执行流程是：

CPU A发出了一条指令，从主内存中读取x。

CPU A从主内存通过bus读取到 cache a中并将该cache line 设置为E状态。

CPU B发出了一条指令，从主内存中读取x。

CPU B试图从主内存中读取x时，CPU A检测到了地址冲突。这时CPU A对相关数据做出响应。此时x 存储于cache a和cache b中，x在chche a和cache b中都被设置为S状态(共享)。



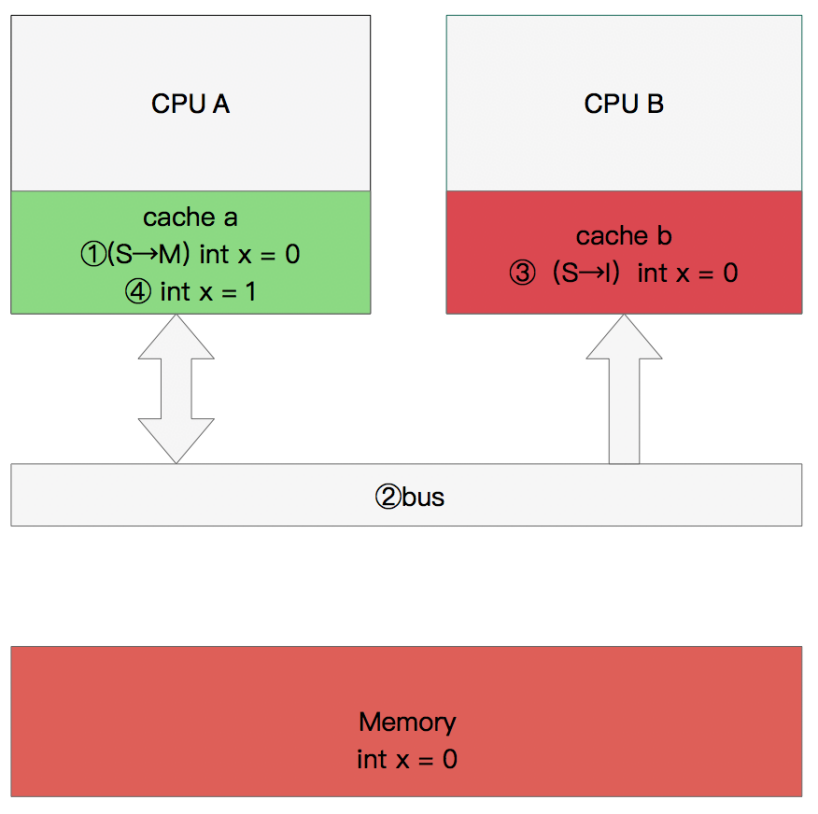
**修改数据**

那么执行流程是：

CPU A 计算完成后发指令需要修改x.

CPU A 将x设置为M状态（修改）并通知缓存了x的CPU B, CPU B将本地cache b中的x设置为I状态(无效)

CPU A 对x进行赋值。



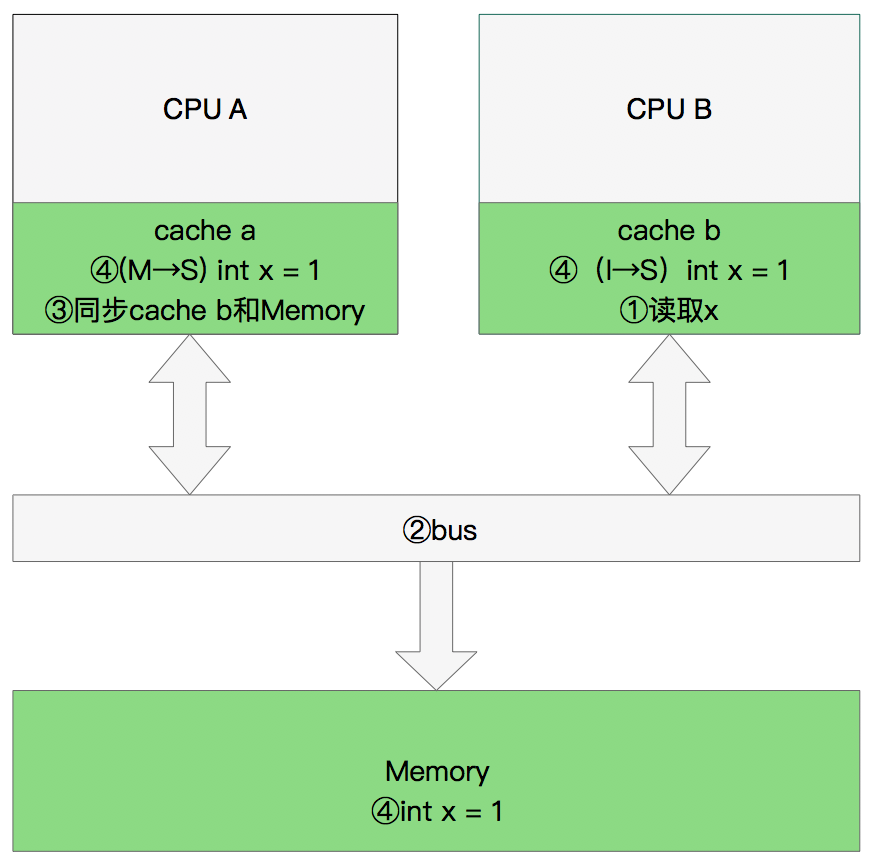
**同步数据**

那么执行流程是：

CPU B 发出了要读取x的指令。

CPU B 通知CPU A,CPU A将修改后的数据同步到主内存时cache a 修改为E（独享）

CPU A同步CPU B的x,将cache a和同步后cache b中的x设置为S状态（共享）。



**MESI优化和他们引入的问题**

缓存的一致性消息传递是要时间的，这就使其切换时会产生延迟。当一个缓存被切换状态时其他缓存收到消息完成各自的切换并且发出回应消息这么一长串的时间中CPU都会等待所有缓存响应完成。可能出现的阻塞都会导致各种各样的性能问题和稳定性问题。

**CPU切换状态阻塞解决-存储缓存（Store Bufferes）**

为了避免这种CPU运算能力的浪费，Store Bufferes被引入使用。处理器把它想要写入到主存的值写到缓存，然后继续去处理其他事情。当所有失效确认（Invalidate Acknowledge）都接收到时，数据才会最终被提交。

这么做有两个风险

1. 就是处理器会尝试从存储缓存（Store buffer）中读取值，但它还没有进行提交。这个的解决方案称为Store Forwarding，它使得加载的时候，如果存储缓存中存在，则进行返回。  
   第二、保存什么时候会完成，这个并没有任何保证。

|  |
| --- |
| value = 3；  void exeToCPUA(){  value = 10;  isFinsh = true;  }  void exeToCPUB(){  if(isFinsh){  //value一定等于10？！  assert value == 10;  }  } |

写屏障 Store Memory Barrier(a.k.a. ST, SMB, smp\_wmb)是一条告诉处理器在执行这之后的指令之前，应用所有已经在存储缓存（store buffer）中的保存的指令。

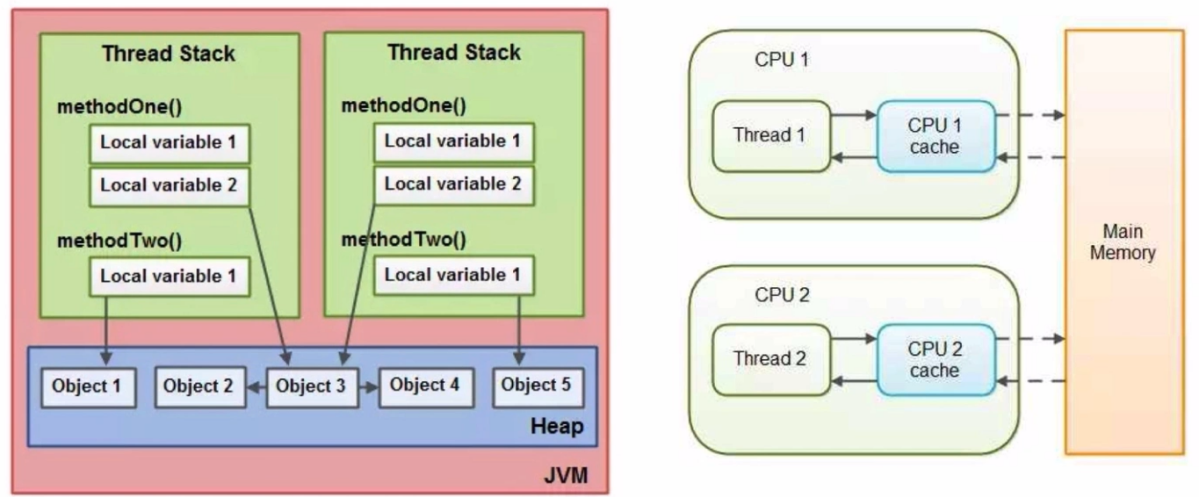
读屏障Load Memory Barrier (a.k.a. LD, RMB, smp\_rmb)是一条告诉处理器在执行任何的加载前，先应用所有已经在失效队列中的失效操作的指令。

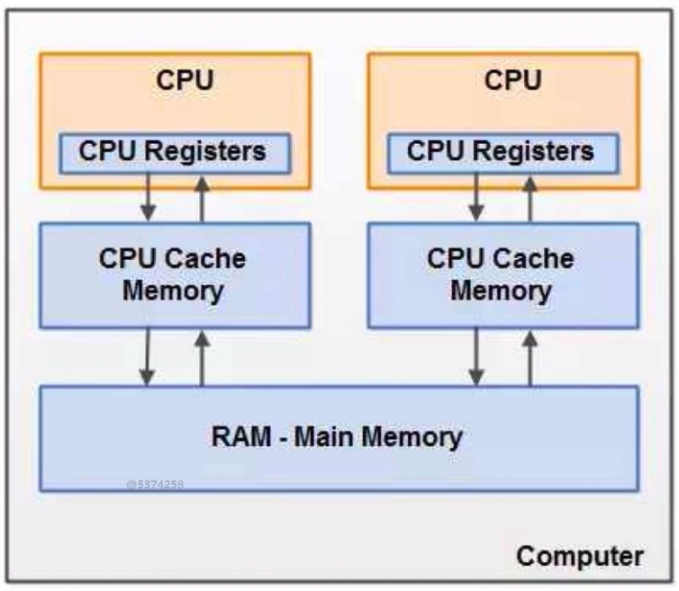
|  |
| --- |
| void executedOnCpu0() {  value = 10;  //在更新数据之前必须将所有存储缓存（store buffer）中的指令执行完毕。  storeMemoryBarrier();  finished = true;  }  void executedOnCpu1() {  while(!finished);  //在读取之前将所有失效队列中关于该数据的指令执行完毕。  loadMemoryBarrier();  assert value == 10;  } |

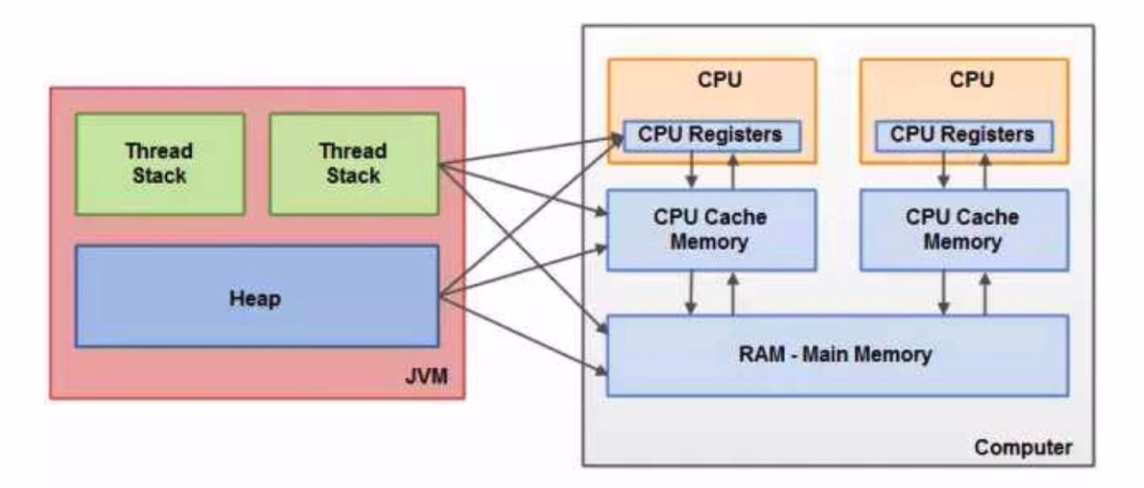
### 2.2CPU多级缓存--乱序执行优化

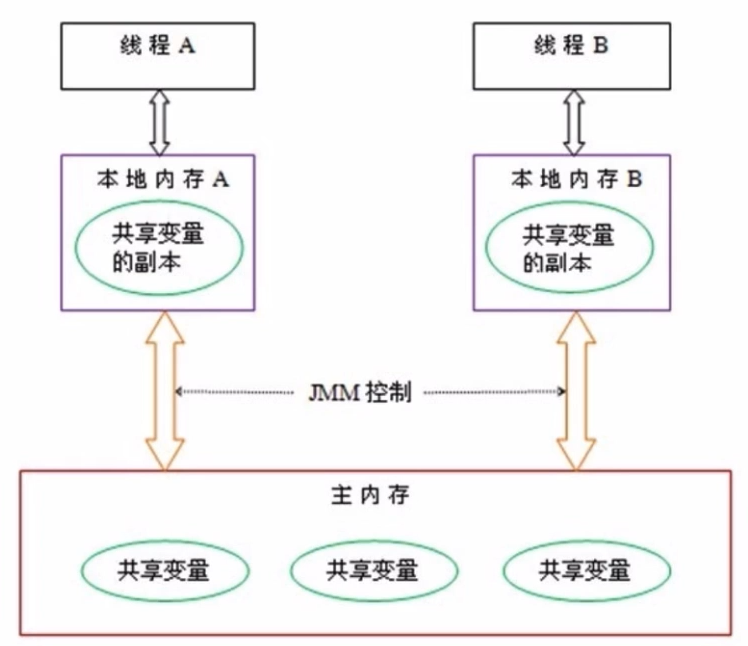


### 2.3Java内存模型

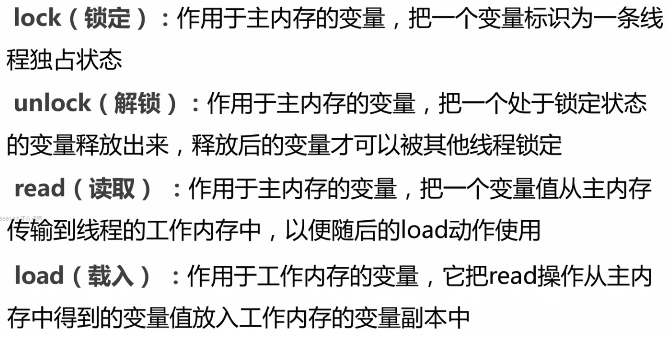


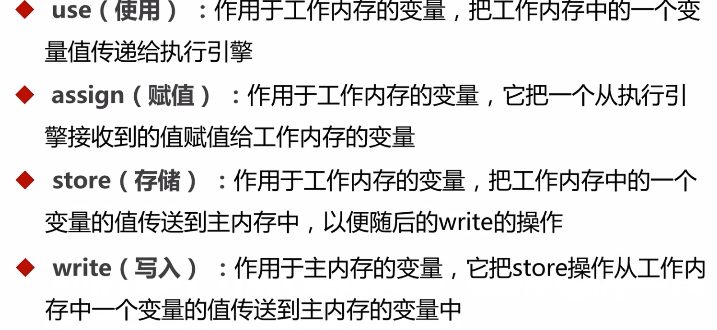




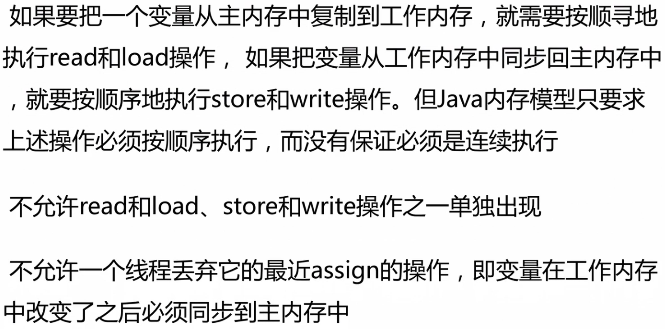


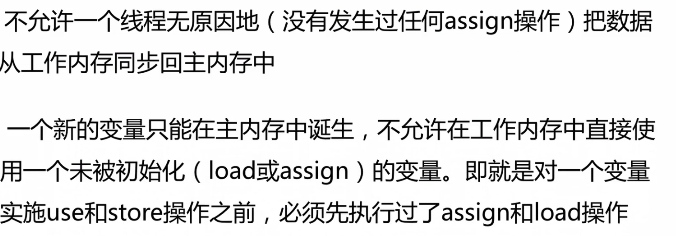
同步八种操作

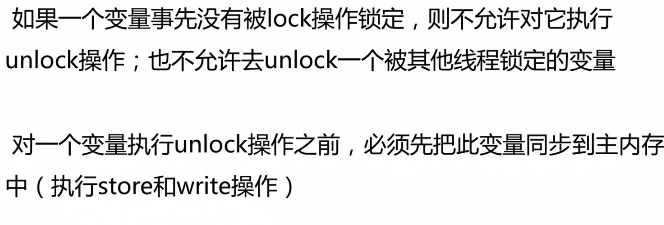


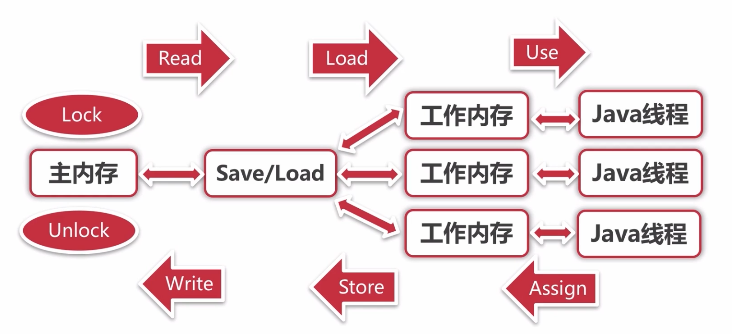


同步规则

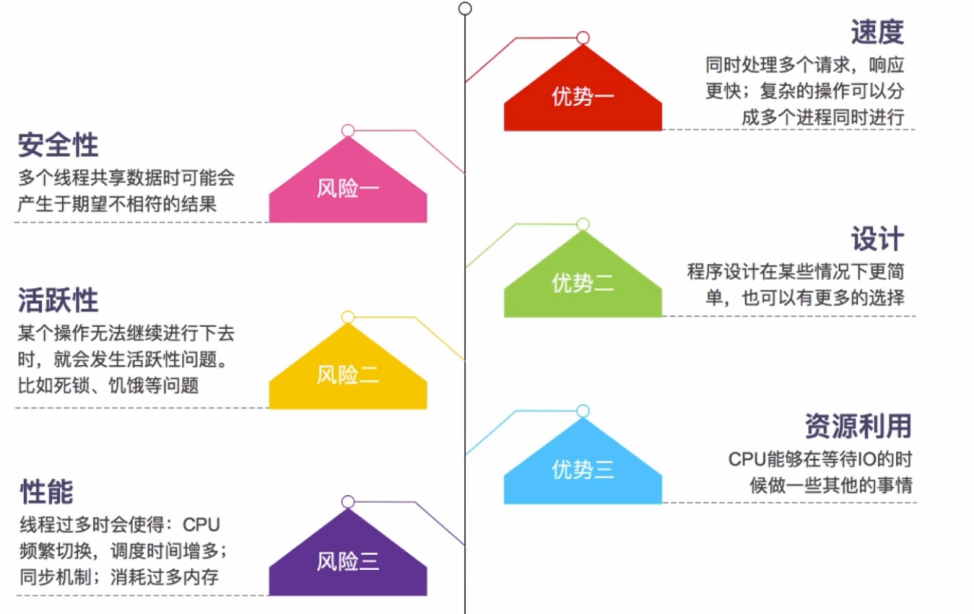








### 2.4并发的优势与风险



## 第3节 项目准备

### 3.1环境初始化

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>  <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>1.5.21.RELEASE</**version**>  <**relativePath**/> *<!-- lookup parent from repository -->* </**parent**>  <**groupId**>com.byf</**groupId**>  <**artifactId**>concurrency</**artifactId**>  <**version**>0.0.1-SNAPSHOT</**version**>  <**packaging**>war</**packaging**>  <**name**>concurrency</**name**>  <**description**>Demo project for Spring Boot</**description**>   <**properties**>  <**java.version**>1.8</**java.version**>  </**properties**>   <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot</**artifactId**>  <**version**>1.5.21.RELEASE</**version**>  </**dependency**>  </**dependencies**>   <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**>  </**build**>  *<!-- 配置jar包的私服仓库 -->* <**repositories**>  <**repository**>  <**id**>app\_repository</**id**>  <**name**>app\_repository</**name**>  <**url**>http://127.0.0.1:8081/nexus/content/groups/public/</**url**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  <**snapshots**>  <**enabled**>true</**enabled**>  </**snapshots**>  </**repository**>  </**repositories**>  *<!-- 配置插件的私服仓库 -->* <**pluginRepositories**>  <**pluginRepository**>  <**id**>app\_repository</**id**>  <**name**>app\_repository</**name**>  <**url**>http://127.0.0.1:8081/nexus/content/groups/public/</**url**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  <**snapshots**>  <**enabled**>true</**enabled**>  </**snapshots**>  </**pluginRepository**>  </**pluginRepositories**> </**project**> |

### 3.2案例准备

·github新建仓库concurrency

·本地在G:\concurrent\_programming下克隆项目

|  |
| --- |
| > git clone git@github.com:byf312358196/concurrency.git |

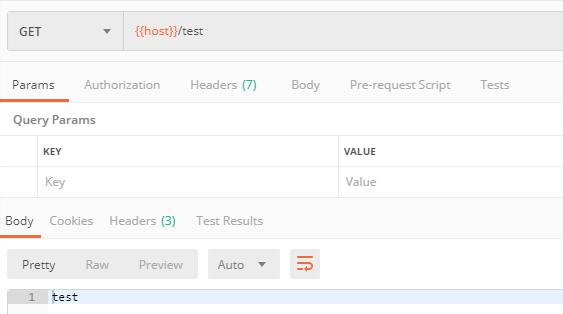
·start.spring.io新建springboot项目

·git提交代码

### 3.3并发模拟-工具



·postman



·ab

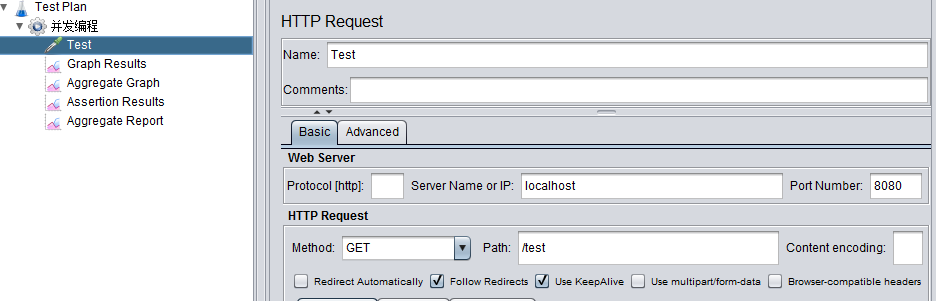
ab -n 100 -c 5 <http://localhost:8080/test>

·Jmeter

|  |
| --- |
| Error: Java version -- 1.7.0\_80 -- is too low to run JMeter. Needs a Java version greater than or eq |

Jmeter指定jdk版本

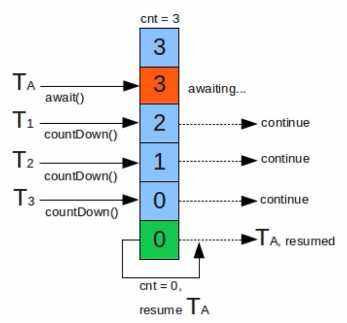
|  |
| --- |
| setlocal  set MINIMAL\_VERSION=1.8.0  set JAVA\_HOME=F:\Program Files\Java\jdk1.8.0\_172  set PATH=%JAVA\_HOME%\bin;%PATH% |



### 3.4并发模拟-代码

·CountDownLatch

等待countdown线程执行完，再进行其他处理



·Semaphore

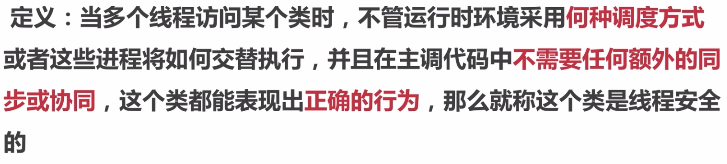
控制线程的并发数，同一时间仅有两个线程在抢占CPU，一个执行，一个阻塞，其他线程处于挂起，等待被唤醒



|  |
| --- |
| @Slf4j @NotThreadSafe **public class** ConcurrencyTest {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static int** *count* = 0;   **private static void** add(){  *count*++;  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  } } |
| 10:04:01.783 [main] INFO com.byf.concurrency.counter.ConcurrencyTest - count:4963 |

## 第4节线程安全性

### 4.1原子性-AtomicLong、LongAdder



|  |
| --- |
| **private static** AtomicInteger *count* = **new** AtomicInteger(0);  **private static void** add(){  *count*.getAndIncrement(); } |
| **public final int** getAndAddInt(Object var1, **long** var2, **int** var4) {  **int** var5;  **do** {  // 从底层获取对象var1当前的值var2，赋给var5  var5 = **this**.getIntVolatile(var1, var2);  // 比较从对象var1取得当前值var2如果期待的是var5，就把var5加上var4，与预期不符，则重新从底层获取对象var1当前的值var2赋给var5，继续比较  } **while**(!**this**.compareAndSwapInt(var1, var2, var5, var5 + var4));   **return** var5; } |

|  |
| --- |
| **private static** LongAdder *count* = **new** LongAdder();  **private static void** add(){  *count*.increment(); } |
| **原理：LongAdder将不同线程通过Hash算法对数组中的Hash坐标对应的值进行增加，最后求和，分散了Atomic单点统计的压力。但并发更新可能导致统计的数据有些误差。**  **实际使用中，低并发使用Atomic优先，无需分cell求和计算；**  **高并发统计如果不是要求全局唯一，高精度的统计，优先使用LongAdder。** |

### 4.2原子性-AtomicReference、AtomicReferenceFieldUpdater

|  |
| --- |
| @Slf4j @ThreadSafe **public class** AtomicReferenceTest {  **private static** AtomicReference<Integer> *count* = **new** AtomicReference<>(0);   **public static void** main(String[] args) {  *count*.compareAndSet(0,2); *// 2  count*.compareAndSet(0,2); *// no  count*.compareAndSet(1,3); *// no  count*.compareAndSet(2,4); *// 4  count*.compareAndSet(3,5); *// no* ***log***.info(**"count:{}"**,*count*);  } } |
| 11:01:07.790 [main] INFO com.byf.concurrency.counter.atomic.AtomicReferenceTest - count:4 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** AtomicReferenceFieldUpdaterTest {   @Getter  **private volatile int count** = 100;  **private static** AtomicIntegerFieldUpdater<AtomicReferenceFieldUpdaterTest> *updater* =  AtomicIntegerFieldUpdater.*newUpdater*(AtomicReferenceFieldUpdaterTest.**class**, **"count"**);   **private static** AtomicReferenceFieldUpdaterTest *test* = **new** AtomicReferenceFieldUpdaterTest();  **public static void** main(String[] args) {  **if** (*updater*.compareAndSet(*test*,100,200)){  ***log***.info(**"count{}"**, *test*.getCount());  }   **if** (*updater*.compareAndSet(*test*,100,200)){  ***log***.info(**"update success count:{}"**, *test*.getCount());  } **else** {  ***log***.error(**"update failed count:{}"**, *test*.getCount());  }  } } |
| 11:00:13.660 [main] INFO com.byf.concurrency.counter.atomic.AtomicReferenceFieldUpdaterTest - count200  11:00:13.660 [main] ERROR com.byf.concurrency.counter.atomic.AtomicReferenceFieldUpdaterTest - update failed count:200 |

AtomicStampReference：CAS的ABA问题

线程1在访问共享数据是的值时，线程2将A的值改为B，很快又将B改回A，此时，线程1操作的值的版本其实已经发生了改变，这与设计思想不符。

线程1：我期望的值是A时，才进行修改；

线程2：我期望是A改成B，然后期望是B改成A；

线程1：比较与自己期望的值，发现此时是A，进行修改操作；

期望并发场景下，某段代码只执行一次

|  |
| --- |
| **private static** AtomicBoolean *isHappend* = **new** AtomicBoolean(**false**);  **private static void** test(){  **if** (*isHappend*.compareAndSet(**false**,**true**)){  ***log***.info(**"execute, isHaddped:{}"**, *isHappend*);  } } |
| **11:13:11.391 [pool-1-thread-1] INFO com.byf.concurrency.counter.atomic.ConcurrencyAtomicBooleanTest - execute, isHaddped:true**  **11:13:11.407 [main] INFO com.byf.concurrency.counter.atomic.ConcurrencyAtomicBooleanTest - isHapped:true** |

### 4.3原子性-Synchronized

1. 修饰代码块、修饰静态

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** SynchronizedTest1 {   *// 修饰代码块* **public void** test1(**int** j){  **synchronized** (**this**){  **for** (**int** i=0;i<10;i++){  ***log***.info(**"test1 {} -> {}"**,j, i);  }  }  }  *// 修饰方法* **public synchronized void** test2(**int** j){  **for** (**int** i=0;i<10;i++){  ***log***.info(**"test2 {} -> {}"**, j, i);  }  }   **public static void** main(String[] args) {  SynchronizedTest1 t1 = **new** SynchronizedTest1();  SynchronizedTest1 t2 = **new** SynchronizedTest1();  ExecutorService exec = Executors.*newCachedThreadPool*();  exec.execute(()->{  t1.test1(1);  });   exec.execute(()->{  t2.test2(2);  });  exec.shutdown();  }  } |

1. 修饰静态方法、使用类锁

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SynchronizedTest2 {   *// 修饰类* **public void** test1(**int** j){  **synchronized** (SynchronizedTest2.**class**){  **for** (**int** i=0;i<10;i++){  ***log***.info(**"test1 {} -> {}"**,j, i);  }  }  }  *// 修饰方法* **public synchronized static void** test2(**int** j){  **for** (**int** i=0;i<10;i++){  ***log***.info(**"test2 {} -> {}"**, j, i);  }  }   **public static void** main(String[] args) {  SynchronizedTest2 t1 = **new** SynchronizedTest2();  SynchronizedTest2 t2 = **new** SynchronizedTest2();  ExecutorService exec = Executors.*newCachedThreadPool*();  exec.execute(()->{  t1.test1(1);  });   exec.execute(()->{  t2.*test2*(2);  });  exec.shutdown();  }  } |

1. 修改计数器使成为线程安全

|  |
| --- |
| @Slf4j @ThreadSafe **public class** ConcurrencyTest3 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static int** *count* = 0;   **private synchronized static void** add(){  *count*++;  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  } } |

原子性--对比

·Synchronized：不可中断锁，适合竞争不激烈，可读性好。在并发高的场景下，性能下降，偏向锁-->轻量级锁-->自旋锁

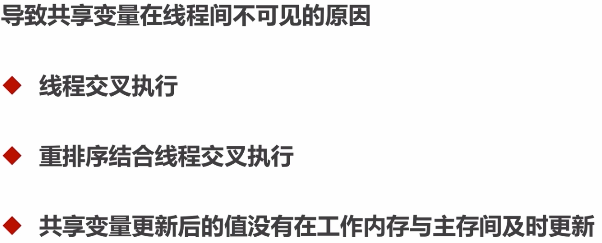
·Lock：可中断锁，多样化同步，竞争激烈能维持同步；

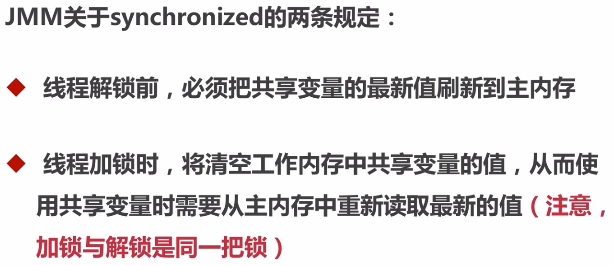
·Atomic：竞争激烈能维持常态，比Lock性能好（CAS，存在ABA问题）；只能维持同步一个值；

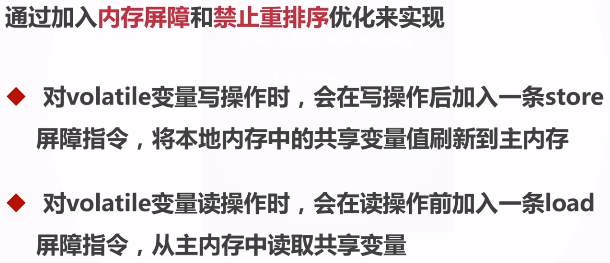
### 4.4可见性

1. 导致共享变量在线程间不可见的原因

·主内存与工作内存







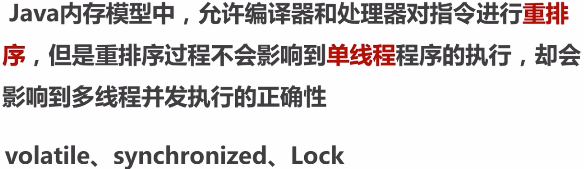




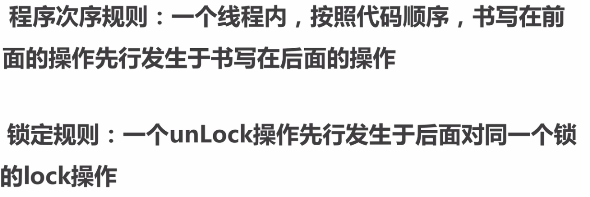
不适合做计数的修改型操作

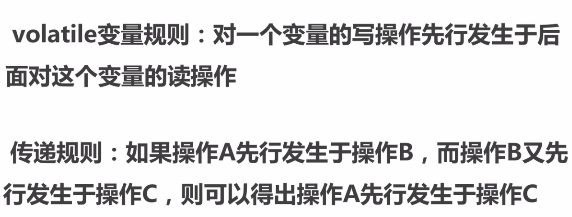
|  |
| --- |
| @Slf4j @NotThreadSafe **public class** ConcurrencyTest4 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private volatile static int** *count* = 0;   **private static void** add(){  *count*++;  *// 1.读count值  // 2.count+1  // 3.写count值* }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  } } |

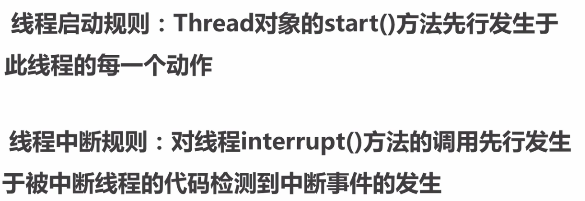
### 4.5有序性与总结

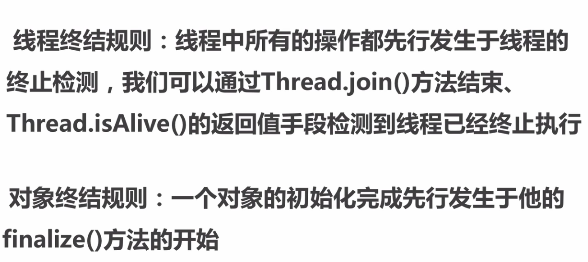


1. happens-before原则

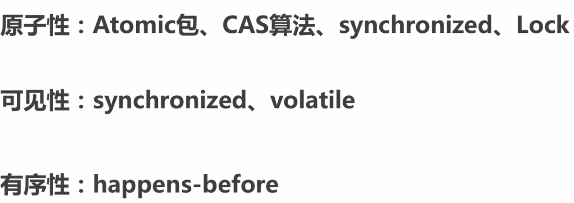






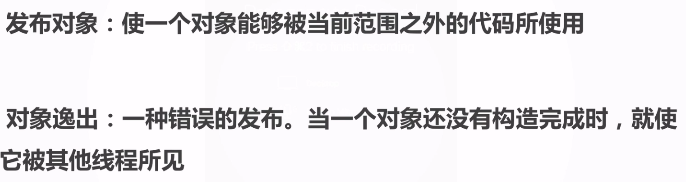


1. 总结



## 第5节安全发布对象

### 5.1发布与逸出



|  |
| --- |
| @Slf4j  @NotThreadSafe **public class** UnsafePublish {  **private** String[] **states** = {**"a"**, **"b"**, **"c"**};   **public** String[] getStates() {  **return states**;  }   **public static void** main(String[] args) {  UnsafePublish publish = **new** UnsafePublish();  ***log***.info(**"{}"**, Arrays.*toString*(publish.getStates()));   publish.getStates()[0] = **"d"**;  ***log***.info(**"{}"**, Arrays.*toString*(publish.getStates()));  } } |
| 17:34:20.254 [main] INFO com.byf.concurrency.publish.UnsafePublish - [a, b, c]  17:34:20.264 [main] INFO com.byf.concurrency.publish.UnsafePublish - [d, b, c] |

对象未构造完成前，不可以发布，防止新线程在构造期间，拿到发布的过期对象。

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** UnsafePublish {  **private** String[] **states** = {**"a"**, **"b"**, **"c"**};   **public** String[] getStates() {  **return states**;  }   **public static void** main(String[] args) {  UnsafePublish publish = **new** UnsafePublish();  ***log***.info(**"{}"**, Arrays.*toString*(publish.getStates()));   publish.getStates()[0] = **"d"**;  ***log***.info(**"{}"**, Arrays.*toString*(publish.getStates()));  } } |
| 18:03:09.618 [main] INFO com.byf.concurrency.publish.UnsafePublish - [a, b, c]  18:03:09.618 [main] INFO com.byf.concurrency.publish.UnsafePublish - [d, b, c] |

|  |
| --- |
| @Slf4j @NotRecommend **public class** Escape {    **public** Escape(){  **new** InnerClass();  }   **private class** InnerClass{  **public** InnerClass(){  ***log***.info(**"{}"**, Escape.**this**.**thisCanBeEsape**);  }  }  **private int thisCanBeEsape** = 0;   **public static void** main(String[] args) {  **new** Escape();  } } |

### 5.2安全发布对象

单例的几种模式：

1. 不安全懒汉模式；
2. 安全饿汉模式（构造单例逻辑复制时，存在程序加载性能问题）；
3. 安全懒汉模式(Synchronized,并发高时存在性能问题)；
4. 双重检测枷锁懒汉单例模式（volatile禁止指令重排序）；

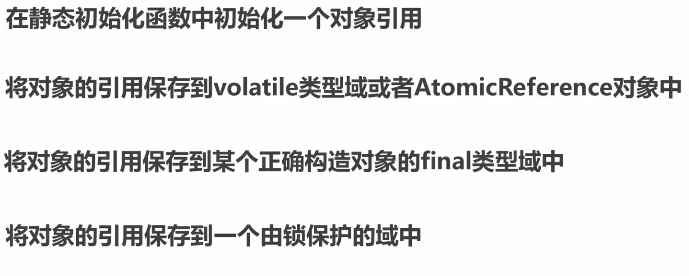
|  |
| --- |
| Instance = new Singleton();  // 1. memory = allocate() 分配对象的内存空间  // 2.ctorInstance() 初始化对象  // 3. instance = memory 设置instance指向刚分配的内存  如果上述步骤没有使用volatile，而指令重排序，步骤1和3执行后，步骤2还未执行，双重检测判断instance拿到的实例还未初始化构造，从而引入线程安全问题。 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SingletonExample1 {  **private** SingletonExample1(){   }  **private volatile static** SingletonExample1 *instance* = **null**;   **public static** SingletonExample1 getInstance(){  **if** (*instance* == **null**){  **synchronized** (SingletonExample1.**class**){  **if** (*instance* == **null**){  *instance* = **new** SingletonExample1();  }  }  }  **return** *instance*;  }  **public static void** main(String[] args) {  SingletonExample1 singletonExample1 = **new** SingletonExample1();  }  } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SingletonExample2 {  **private** SingletonExample2(){   }  *// 静态代码块的初始化顺序，从上往下；* **private static** SingletonExample2 *instance* = **null**;   **static** {  *instance* = **new** SingletonExample2();  }   **public static** SingletonExample2 getInstance(){  **return** *instance*;  }  **public static void** main(String[] args) {  System.***out***.println(*getInstance*().hashCode());  System.***out***.println(*getInstance*().hashCode());  } } |

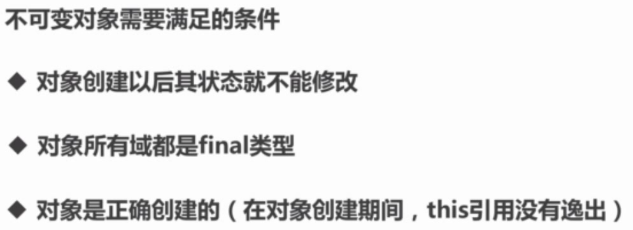
### 5.3推荐枚举单例模式

|  |
| --- |
| @Slf4j @ThreadSafe @Recommend **public class** SingletonExample3 {  *// 私有构造函数* **private** SingletonExample3(){   }   **public static** SingletonExample3 getInstance(){  **return** Singleton.***INSTANCE***.getInstance();  }   **private enum** Singleton {  INSTANCE;  **private** SingletonExample3 singleton = **null**;  *// JVM保证这个方法绝对只调用一次* Singleton(){  singleton = **new** SingletonExample3();  }  **private** SingletonExample3 getInstance(){  **return** singleton;  }  }   **public static void** main(String[] args) {  System.***out***.println(SingletonExample3.*getInstance*());  } } |



## 第6节线程安全策略

### 6.1不可变对象



1. final关键字：类、方法、变量

·修饰类：不能被继承

·修饰方法：

1. 锁定方法不能被继承类修改；
2. 效率：早期的java方法中，final会被转为内嵌调用（方法过于庞大，会引起效率问题），private方法会被隐式修饰为final；
3. 修饰变量：基本数据类型变量、引用类型变量；

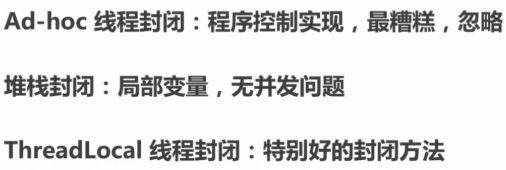
|  |
| --- |
| @Slf4j @NotThreadSafe **public class** ImmutableExample1 {  **private final static** Integer ***a*** = 1;  **private final static** String ***b*** = **"2"**;  **private final static** Map<Integer, Integer> ***map*** = Maps.*newHashMap*();   **static** {  ***map***.put(1,2);  ***map***.put(3,4);  ***map***.put(5,6);  }   **public static void** main(String[] args) {  */\*a = 2;  b = "3";  map = Maps.newHashMap();\*/* ***map***.put(1,3);  ***log***.info(**"{}"**,***map***.get(1));  }    **private void** test(**final int** a ){  *// a = 1;* } } |



|  |
| --- |
| @Slf4j @ThreadSafe **public class** ImmutableExample2 {  **private final static** Integer ***a*** = 1;  **private final static** String ***b*** = **"2"**;  **private static** Map<Integer, Integer> *map* = Maps.*newHashMap*();   **static** {  *map*.put(1,2);  *map*.put(3,4);  *map*.put(5,6);  *map* = Collections.*unmodifiableMap*(*map*);  }   **public static void** main(String[] args) {  *map*.put(1,3);  } } |
| Exception in thread "main" java.lang.UnsupportedOperationException  at java.util.Collections$UnmodifiableMap.put(Collections.java:1457)  at com.byf.concurrency.immutable.ImmutableExample2.main(ImmutableExample2.java:25) |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** ImmutableExample3 {   **private final static** ImmutableList ***list*** = ImmutableList.*of*(1,2,3);  **private final static** ImmutableList ***set*** = ImmutableList.*copyOf*(***list***);  **private final static** ImmutableMap<Integer, Integer> ***map*** = ImmutableMap.*of*(1,3,2,4);  **private final static** ImmutableMap<Integer, Integer> ***map2*** = ImmutableMap.<Integer,Integer>*builder*().put(1,3).put(2,4).build();   **public static void** main(String[] args) {  *// list.add(1);  // set.add(4);  // map.put(1,4);* ***map2***.put(4,4);  } } |
| Exception in thread "main" java.lang.UnsupportedOperationException  at com.google.common.collect.ImmutableMap.put(ImmutableMap.java:407)  at java.util.Map.putIfAbsent(Map.java:744)  at com.byf.concurrency.immutable.ImmutableExample3.main(ImmutableExample3.java:25) |

### 6.2线程封闭



示例：通过ThreadLocal保存线程id或其他信息（Controller无需向下传递request对象中的数据），http响应结束后，将资源释放。

1. 定义线程封闭对象，保存Http请求的参数或线程id；

|  |
| --- |
| **public class** RequestHolder {  **private final static** ThreadLocal<Long> ***requestHolder*** = **new** ThreadLocal<Long>();   **public static void** add(Long id){  ***requestHolder***.set(id);  }   **public static** Long getId(){  **return *requestHolder***.get();  }   **public static void** remove(){  ***requestHolder***.remove();  } } |

2.Controller处理前过滤，将当前线程id保存下来；

|  |
| --- |
| @Slf4j **public class** HttpFilter **implements** Filter {  @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException {   }   @Override  **public void** doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) **throws** IOException, ServletException {  HttpServletRequest request = (HttpServletRequest) servletRequest;  ***log***.info(**"do filter, {}, {}"**, Thread.*currentThread*().getId(), request.getServletPath());  *// request.getSession().getAttribute("user");* RequestHolder.*add*(Thread.*currentThread*().getId());  filterChain.doFilter(servletRequest,servletResponse);   }   @Override  **public void** destroy() {   } } |

3.Controller处理后将ThreadLocal保存的资源释放，防止线程泄漏；

|  |
| --- |
| @Slf4j **public class** HttpIntercepter **extends** HandlerInterceptorAdapter {  @Override  **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  ***log***.info(**"preHandle"**);  **return true**;  }   @Override  **public void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) **throws** Exception {  ***log***.info(**"afterCompletion"**);  RequestHolder.*remove*();  **return**;  } } |

4.SpringBoot启动加载过滤器和拦截器的Bean

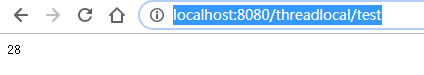
|  |
| --- |
| @SpringBootApplication **public class** ConcurrencyApplication **extends** WebMvcConfigurerAdapter {   **public static void** main(String[] args) {  SpringApplication.*run*(ConcurrencyApplication.**class**, args);  }   @Bean  **public** FilterRegistrationBean httpFilter(){  FilterRegistrationBean registrationBean = **new** FilterRegistrationBean();  registrationBean.setFilter(**new** HttpFilter());  registrationBean.addUrlPatterns(**"/threadlocal/\*"**);  **return** registrationBean;  }   @Override  **public void** addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(**new** HttpIntercepter()).addPathPatterns(**"/\*\*"**);  } } |

1. Controller实现

|  |
| --- |
| @Controller @RequestMapping(**"/threadlocal"**) **public class** ThreadLocalController {  @RequestMapping(**"/test"**)  @ResponseBody  **public** Long test(){  **return** RequestHolder.*getId*();  } } |

访问：<http://localhost:8080/threadlocal/test>

返回线程ID：



日志打印：

|  |
| --- |
| 2019-07-09 22:27:57.738 INFO 4348 --- [nio-8080-exec-1] com.byf.concurrency.HttpFilter : do filter, 28, /threadlocal/test  2019-07-09 22:27:57.746 INFO 4348 --- [nio-8080-exec-1] c.b.concurrency.example.HttpIntercepter : preHandle  2019-07-09 22:27:57.803 INFO 4348 --- [nio-8080-exec-1] c.b.concurrency.example.HttpIntercepter : afterCompletion |

在JDBC连接池取Connection对象也是线程封闭，线程之间无需关注Connection是否线程安全，CURD使用完毕后，释放连接；

### 6.3线程不安全的类与写法

·StringBuilder -> StringBuffer

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** StringExample1 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** StringBuilder *stringBuilder* = **new** StringBuilder();   **private static void** add(){  *stringBuilder*.append(**"1"**);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"length:{}"**,*stringBuilder*.length());  } } |
| 22:58:55.493 [main] INFO com.byf.concurrency.example.commonUnsafe.StringExample1 - length:4980 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** StringExample2 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** StringBuffer *stringBuilder* = **new** StringBuffer();   **private static void** add(){  *stringBuilder*.append(**"1"**);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"length:{}"**,*stringBuilder*.length());  } } |
| 22:59:50.155 [main] INFO com.byf.concurrency.example.commonUnsafe.StringExample2 - length:5000 |

·SimpleDateFormat -> JodaTime

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** DateFormatExample1 {  **private static** SimpleDateFormat *simpleDateFormat* = **new** SimpleDateFormat(**"yyyy-mm-dd"**);   **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;   **private static void** update(){  **try** {  *simpleDateFormat*.parse(**"2018-02-08"**);  } **catch** (ParseException e) {  ***log***.error(**"parse exception"**, e);  }   }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *update*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"length:{}"**,*simpleDateFormat*);  }  } |
| java.lang.NumberFormatException: For input string: ""  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)  at java.lang.Long.parseLong(Long.java:601)  at java.lang.Long.parseLong(Long.java:631)  at java.text.DigitList.getLong(DigitList.java:195)  at java.text.DecimalFormat.parse(DecimalFormat.java:2084)  at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:2160)  at java.text.SimpleDateFormat.parse(SimpleDateFormat.java:1514)  at java.text.DateFormat.parse(DateFormat.java:364)  at com.byf.concurrency.example.commonUnsafe.DateFormatExample1.update(DateFormatExample1.java:23)  at com.byf.concurrency.example.commonUnsafe.DateFormatExample1.lambda$main$0(DateFormatExample1.java:38)  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)  at java.lang.Thread.run(Thread.java:748)  java.lang.NumberFormatException: multiple points  at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1890)  at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)  at java.lang.Double.parseDouble(Double.java:538)  at java.text.DigitList.getDouble(DigitList.java:169)  at java.text.DecimalFormat.parse(DecimalFormat.java:2089)  at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:2162)  at java.text.SimpleDateFormat.parse(SimpleDateFormat.java:1514)  at java.text.DateFormat.parse(DateFormat.java:364)  at com.byf.concurrency.example.commonUnsafe.DateFormatExample1.update(DateFormatExample1.java:23)  at com.byf.concurrency.example.commonUnsafe.DateFormatExample1.lambda$main$0(DateFormatExample1.java:38)  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)  at java.lang.Thread.run(Thread.java:748) |

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** DateFormatExample2 {   **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;   **private static void** update(){  **try** {  SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat(**"yyyy-mm-dd"**);  simpleDateFormat.parse(**"2018-02-08"**);  } **catch** (ParseException e) {  ***log***.error(**"parse exception"**, e);  }   }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *update*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  }  } |
| Process finished with exit code 0 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** DateFormatExample3 {   **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** DateTimeFormatter *dateTimeFormatter* = DateTimeFormat.*forPattern*(**"yyyy-mmm-dd"**);   **private static void** update(**final int** count){  ***log***.info(**"{}, {}"**, count, DateTime.*parse*(**"2018-02-28"**, *dateTimeFormatter*));  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  }  } |
| ......  23:12:30.191 [pool-1-thread-2639] INFO com.byf.concurrency.example.commonUnsafe.DateFormatExample3 - 4997, 2018-01-28T00:02:00.000+08:00  23:12:30.191 [pool-1-thread-2637] INFO com.byf.concurrency.example.commonUnsafe.DateFormatExample3 - 4999, 2018-01-28T00:02:00.000+08:00 |

·ArrayList、HashSet、HashMap等Collection

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** ArrayListExample {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** List<Integer> *list* = **new** ArrayList<>();   **private static void** update(**final int** count){  *list*.add(1);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"list size:{}"**,*list*.size());  } } |
| 20:01:27.711 [main] INFO com.byf.concurrency.example.commonUnsafe.ArrayListExample - size:4970 |

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** HashSetExample {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Set<Integer> *list* = **new** HashSet<>();   **private static void** update(**final int** count){  *list*.add(count);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"list size:{}"**,*list*.size());  } } |
| 20:01:10.157 [main] INFO com.byf.concurrency.example.commonUnsafe.HashSetExample - size:4949 |

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** HashMapExample {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Map<Integer,Integer> *list* = **new** HashMap<>();   **private static void** update(**final int** count){  *list*.put(count,count);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"list size:{}"**,*list*.size());  } } |
| 20:01:43.742 [main] INFO com.byf.concurrency.example.commonUnsafe.HashMapExample - size:4913 |

·先检查后执行不是线程安全的操作，if(condition(a)) {handle(a)};

### 6.4同步容器Collections.newSynchronizedXXX

·ArrayList -> Vector/Stack

·HashMap -> HashTable(key/value不能为null)

·Collections.synchronizedxxx(List/Set/Map)

|  |
| --- |
| @Slf4j @ThreadSafe **public class** VectorExample1 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Vector<Integer> *list* = **new** Vector<>();   **private static void** update(**final int** count){  *list*.add(1);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**,*list*.size());  } } |
| 20:28:28.382 [main] INFO com.byf.concurrency.example.syncContainer.VectorExample1 - size:5000 |

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** VectorExample2 {   **private static** Vector<Integer> *vector* = **new** Vector<>();   **public static void** main(String[] args) {  **while** (**true**) {  **for** (**int** i = 0; i < 10; i++) {  *vector*.add(i);  }   Thread thread1 = **new** Thread() {  @Override  **public void** run() {  **for** (**int** i = 0; i < vector.size(); i++) {  vector.remove(i);  }  }  };  Thread thread2 = **new** Thread() {  @Override  **public void** run() {  **for** (**int** i = 0; i < vector.size(); i++) {  vector.get(i);  }  }  };  thread1.start();  *// java.lang.ArrayIndexOutOfBoundsException: Array index out of range: 5* thread2.start();  }  } } |
| Exception in thread "Thread-1098" java.lang.ArrayIndexOutOfBoundsException: Array index out of range: 12  at java.util.Vector.remove(Vector.java:834)  at com.byf.concurrency.example.syncContainer.VectorExample2$1.run(VectorExample2.java:24) |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** HashTableExample {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Map<Integer,Integer> *hashtable* = **new** Hashtable<>();   **private static void** update(**final int** count){  *hashtable*.put(count,count);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**, *hashtable*.size());  } } |
| 20:30:55.527 [main] INFO com.byf.concurrency.example.syncContainer.HashTableExample - size:5000 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** CollectionsExample1 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** List<Integer> *list* = Collections.*synchronizedList*(Lists.*newArrayList*());   **private static void** update(**final int** count){  *list*.add(1);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**,*list*.size());  } } |
| 20:33:56.834 [main] INFO com.byf.concurrency.example.syncContainer.CollectionsExample1 - size:5000 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** CollectionsExample2 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Set<Integer> *set* = Collections.*synchronizedSet*(Sets.*newHashSet*());   **private static void** update(**final int** count){  *set*.add(count);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**, *set*.size());  } } |
| 20:36:14.971 [main] INFO com.byf.concurrency.example.syncContainer.CollectionsExample2 - size:5000 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** CollectionsExample3 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Map<Integer,Integer> *map* = Collections.*synchronizedMap*(Maps.*newHashMap*());   **private static void** update(**final int** count){  *map*.put(count,count);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**, *map*.size());  } } |
| 20:37:49.552 [main] INFO com.byf.concurrency.example.syncContainer.CollectionsExample3 - size:5000 |

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** VectorExample3 {   *// java.util.ConcurrentModificationException* **private static void** test1(Vector<Integer> vector){ *// foreach* **for** (Integer i : vector) {  *// vector.get(i).equals(3)不会抛出 java.util.ConcurrentModificationException* **if** (i.equals(3)) {  vector.remove(i);  }  }  }   **private static void** test2(Vector<Integer> vector){ *// iterator* Iterator iterator = vector.iterator();  **while** (iterator.hasNext()) {  */\*if (iterator.next().equals(3)) {  iterator.remove();  } // 不会抛出 java.util.ConcurrentModificationException\*/* Integer i = (Integer) iterator.next();  **if** (i.equals(3)){  vector.remove(i);  }  }  }  *// success* **private static void** test3(Vector<Integer> vector){ *// for* **for** (**int** i=0;i<vector.size();i++) {  **if** (vector.get(i).equals(3)){  vector.remove(i);  }  }  }   **public static void** main(String[] args) {  Vector<Integer> vector = **new** Vector<>();  vector.add(1);  vector.add(2);  vector.add(3);  *test3*(vector);  ***log***.info(**"size:{}"**,vector.size());  } |
| **// Vector的检查机制，存在并发修改抛异常**  **final void** checkForComodification() {  **if** (**modCount** != **expectedModCount**)  **throw new** ConcurrentModificationException(); } |

### 6.5并发容器J.U.C

·ArrayList -> CopyOnWriteArrayList(读多写少)

|  |
| --- |
| @Slf4j @ThreadSafe **public class** CopyOnWriteArrayListExample {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** List<Integer> *list* = **new** CopyOnWriteArrayList<>();   **private static void** update(**final int** count){  *list*.add(1);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**,*list*.size());  } } |
| **public boolean** add(E e) {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  Object[] elements = getArray();  **int** len = elements.**length**;  Object[] newElements = Arrays.*copyOf*(elements, len + 1);  newElements[len] = e;  setArray(newElements);  **return true**;  } **finally** {  lock.unlock();  } } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** CopyOnWriteArraySetExample {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Set<Integer> *set* = **new** CopyOnWriteArraySet<>();   **private static void** update(**final int** count){  *set*.add(count);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**, *set*.size());  } } |
| 22:44:51.317 [main] INFO com.byf.concurrency.example.concurrent.CopyOnWriteArraySetExample - size:5000 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** ConcurrentHashMapExample {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Map<Integer,Integer> *map* = **new** ConcurrentHashMap<>();   **private static void** update(**final int** count){  *map*.put(count,count);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**, *map*.size());  } } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** ConcurrentSkipListMapExample {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Map<Integer,Integer> *map* = **new** ConcurrentSkipListMap<>();   **private static void** update(**final int** count){  *map*.put(count,count);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**, *map*.size());  } } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SynchronizedSortedMapExample {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static** Map<Integer,Integer> *map* = Collections.*synchronizedSortedMap*(Maps.*newTreeMap*());   **private static void** update(**final int** count){  *map*.put(count,count);  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  **final int** count = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *update*(count);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"size:{}"**, *map*.size());  } } |

### 6.6总结

·ArrayList --> CopyOnWriteArrayList

CopyOnWriteArrayList保证最终一致性要求，但无法满足实时性要求，因此适合读多写少的操作，不确定读写比例不要使用；

设计思想：

1. 读写分离；
2. 最终一致性，拷贝写需要时间，保证最终并发修改List最终是一致的；
3. 使用时另外开辟空间，解决并发冲突。

注意：读操作在原数组操作，无需加锁，写操作加锁拷贝一份，防止多个副本，数据混乱；

·HashSet、TreeSet --> CopyOnWriteArraySet、ConcurrentSkipListsSet、SynchronizedSortedSet

CopyOnWriteArraySet：底层是HashSet

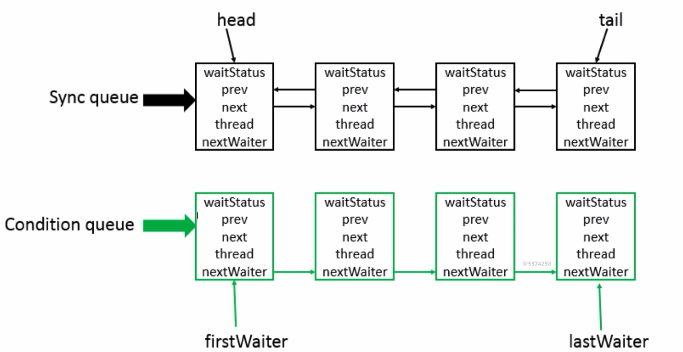
1. 4个线程16000次add操作性能是ConcurrentSkipListsSet四倍；
2. 无序；
3. 允许null；

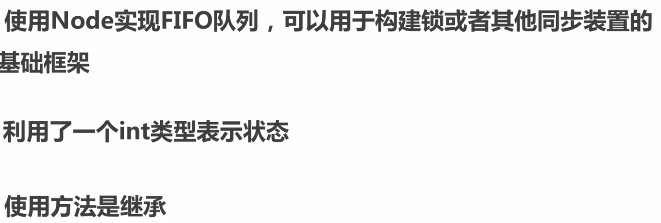
ConcurrentSkipListsSet：底层是TreeMap

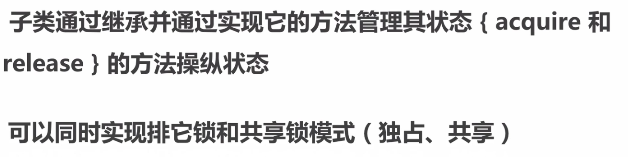
1. 有序；
2. 不允许null；
3. 适用高并发场景的读写操作，并发较少时建议适用CopyOnWriteArraySet，如果低并发且要求有序，推荐Collections集合的SynchronizedSortedSet，是对有序TreeSet的包装。
4. addAll、removeAll、containsAll批量操作需要控制并发，否则会有线程安全问题。

## 第7节J.U.C之AQS

### 7.1介绍







·CountDownLatch

·Semephone

·CycleBarrie

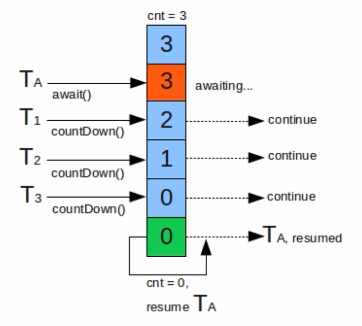
·ReentranLock

·Condition

·FutureTask

### 7.2AQS之CountDownLatch

·同步辅助类



|  |
| --- |
| @Slf4j @ThreadSafe **public class** CountDownLatchExample1 {  **private static int** *threadCount* = 200;   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** CountDownLatch countDownLatch = **new** CountDownLatch(*threadCount*);  **for** (**int** i=0;i<*threadCount*;i++){  **final int** threadNum = i;  exec.execute(()->{  **try** {  *test*(threadNum);  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  } **finally** {  countDownLatch.countDown();  }  });  }  *// countDownLatch在等待指定时间，超过则不再等待  // countDownLatch.await(10, TimeUnit.MILLISECONDS);* countDownLatch.await();  ***log***.info(**"finish"**);  exec.shutdown();  }   **private static void** test(**int** threadNum) **throws** InterruptedException {  Thread.*sleep*(100);  ***log***.info(**"threadNum:{}"**,threadNum);  Thread.*sleep*(100);  } } |
| ......  22:56:23.149 [pool-1-thread-193] INFO com.byf.concurrency.example.aqs.CountDownLatchExample1 - threadNum:192  22:56:23.259 [main] INFO com.byf.concurrency.example.aqs.CountDownLatchExample1 - finish |

### 7.3AQS之Samaphone



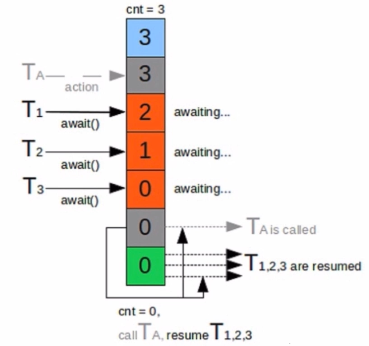
|  |
| --- |
| @Slf4j @ThreadSafe **public class** SemaphoneExample1 {  **private static int** *threadCount* = 20;   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();   **final** Semaphore semaphore = **new** Semaphore(3);  **for** (**int** i=0;i<*threadCount*;i++){  **final int** threadNum = i;  exec.execute(()->{  **try** {  semaphore.acquire();  *test*(threadNum);  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  });  }  exec.shutdown();  }   **private static void** test(**int** threadNum) **throws** InterruptedException {  ***log***.info(**"threadNum:{}"**,threadNum);  Thread.*sleep*(1000);  } } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SemaphoneExample2 {  **private static int** *threadCount* = 20;   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();   **final** Semaphore semaphore = **new** Semaphore(3);  **for** (**int** i=0;i<*threadCount*;i++){  **final int** threadNum = i;  exec.execute(()->{  **try** {  semaphore.acquire(3);  *test*(threadNum);  semaphore.release(3);  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  });  }  exec.shutdown();  }   **private static void** test(**int** threadNum) **throws** InterruptedException {  ***log***.info(**"threadNum:{}"**,threadNum);  Thread.*sleep*(1000);  } } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SemaphoneExample3 {  **private static int** *threadCount* = 20;   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();   **final** Semaphore semaphore = **new** Semaphore(3);  **for** (**int** i=0;i<*threadCount*;i++){  **final int** threadNum = i;  exec.execute(()->{  **try** {  **if**(semaphore.tryAcquire(1)){ *// 超过3个则丢弃  test*(threadNum);  semaphore.release();  }   } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  });  }  exec.shutdown();  }   **private static void** test(**int** threadNum) **throws** InterruptedException {  ***log***.info(**"threadNum:{}"**,threadNum);  Thread.*sleep*(1000);  } } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SemaphoneExample4 {  **private static int** *threadCount* = 20;   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();   **final** Semaphore semaphore = **new** Semaphore(3);  **for** (**int** i=0;i<*threadCount*;i++){  **final int** threadNum = i;  exec.execute(()->{  **try** {  **if**(semaphore.tryAcquire(5000, TimeUnit.***MILLISECONDS***)){ *// 允许时间范围内执行  test*(threadNum);  semaphore.release();  }   } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  });  }  exec.shutdown();  }   **private static void** test(**int** threadNum) **throws** InterruptedException {  ***log***.info(**"threadNum:{}"**,threadNum);  Thread.*sleep*(1000);  } } |

### 7.4AQS之CyclicBarrier



各个线程之间相互等待的场景。

·计算发生错误，重置计数器，让线程们重新执行一次；

·getNumberWaiting方法，获得CyclicBarrier阻塞的数量；

·isBroken方法，判断当前阻塞的线程是否被中断；

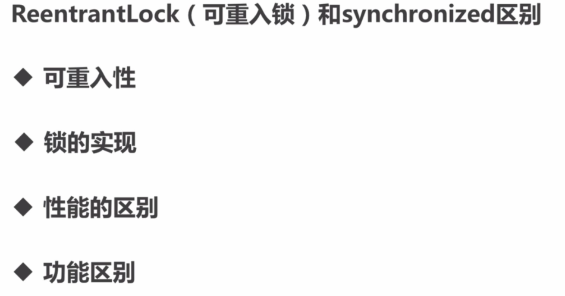
|  |
| --- |
| @Slf4j **public class** CyclicBarrierExample1 {  **private static** CyclicBarrier *cyclicBarrier* = **new** CyclicBarrier(5);   **public static void** main(String[] args) **throws** Exception{  ExecutorService exec = Executors.*newCachedThreadPool*();  **for** (**int** i=0;i<10;i++){  **final int** threadNum = i;  Thread.*sleep*(1000);  exec.execute(()->{  **try** {  *race*(threadNum);  } **catch** (InterruptedException | BrokenBarrierException e) {  ***log***.error(**"exception"**,e);  }  });  }  exec.shutdown();  }   **private static void** race(**int** threadNum) **throws** InterruptedException, BrokenBarrierException {  Thread.*sleep*(100);  ***log***.info(**"{}, ready"**,threadNum);  *cyclicBarrier*.await();  ***log***.info(**"{}, continue"**,threadNum);  } } |
| 19:31:51.306 [pool-1-thread-1] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 0, ready  19:31:52.306 [pool-1-thread-2] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 1, ready  19:31:53.308 [pool-1-thread-3] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 2, ready  19:31:54.308 [pool-1-thread-4] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 3, ready  19:31:55.308 [pool-1-thread-5] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 4, ready  19:31:55.308 [pool-1-thread-5] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 4, continue  19:31:55.308 [pool-1-thread-2] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 1, continue  19:31:55.308 [pool-1-thread-3] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 2, continue  19:31:55.308 [pool-1-thread-1] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 0, continue  19:31:55.308 [pool-1-thread-4] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 3, continue  19:31:56.308 [pool-1-thread-3] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 5, ready  19:31:57.308 [pool-1-thread-5] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 6, ready  19:31:58.308 [pool-1-thread-1] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 7, ready  19:31:59.308 [pool-1-thread-4] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 8, ready  19:32:00.308 [pool-1-thread-2] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 9, ready  19:32:00.308 [pool-1-thread-3] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 5, continue  19:32:00.308 [pool-1-thread-2] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 9, continue  19:32:00.308 [pool-1-thread-5] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 6, continue  19:32:00.308 [pool-1-thread-1] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 7, continue  19:32:00.308 [pool-1-thread-4] INFO com.byf.concurrency.example.aqs.CyclicBarrierExample1 - 8, continue |

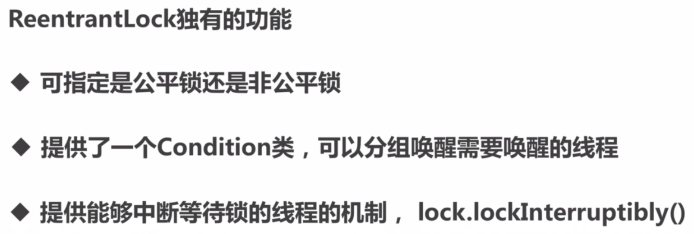
设置cyclicBarrier到达一定时间后执行，此时该线程状态发生改变，将会抛出打破Barrier的异常，如果需要保证业务继续执行，则需要处理BrokenBarrierException异常。

|  |
| --- |
| @Slf4j **public class** CyclicBarrierExample2 {  **private static** CyclicBarrier *cyclicBarrier* = **new** CyclicBarrier(5);   **public static void** main(String[] args) **throws** Exception{  ExecutorService exec = Executors.*newCachedThreadPool*();  **for** (**int** i=0;i<10;i++){  **final int** threadNum = i;  Thread.*sleep*(1000);  exec.execute(()->{  **try** {  *race*(threadNum);  } **catch** (Exception e) {  ***log***.warn(**"exception"**,e);  }  });  }  exec.shutdown();  }   **private static void** race(**int** threadNum) **throws** Exception {  Thread.*sleep*(100);  ***log***.info(**"{}, ready"**,threadNum);  **try**{  *cyclicBarrier*.await(2000, TimeUnit.***MILLISECONDS***);  } **catch** (BrokenBarrierException | TimeoutException e){  ***log***.warn(**"BarrierException"**, e);  }   ***log***.info(**"{}, continue"**,threadNum);  } } |

|  |
| --- |
| @Slf4j **public class** CyclicBarrierExample2 {  **private static** CyclicBarrier *cyclicBarrier* = **new** CyclicBarrier(5);   **public static void** main(String[] args) **throws** Exception{  ExecutorService exec = Executors.*newCachedThreadPool*();  **for** (**int** i=0;i<10;i++){  **final int** threadNum = i;  Thread.*sleep*(1000);  exec.execute(()->{  **try** {  *race*(threadNum);  } **catch** (Exception e) {  ***log***.warn(**"exception"**,e);  }  });  }  exec.shutdown();  }   **private static void** race(**int** threadNum) **throws** Exception {  Thread.*sleep*(100);  ***log***.info(**"{}, ready"**,threadNum);  **try**{  *cyclicBarrier*.await(2000, TimeUnit.***MILLISECONDS***);  } **catch** (BrokenBarrierException | TimeoutException e){  *// 如果一定要保证下面的流程执行，此处可以捕获Exception异常，和定时器Timer一样，定时器抛出异常后没有捕获，则定时器被中断* ***log***.warn(**"BarrierException"**, e);  }   ***log***.info(**"{}, continue"**,threadNum);  } } |

### 7.5ReentrantLock与锁





只有少量竞争者，Synchronized

线程增长趋势可预估，使用ReentrantLock

|  |
| --- |
| @Slf4j @ThreadSafe **public class** LockExample1 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static int** *count* = 0;   **private synchronized static void** add(){  *count*++;  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  } } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** LockExample2 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static int** *count* = 0;  **private final static** Lock ***lock*** = **new** ReentrantLock();  **private static void** add(){  ***lock***.lock();  **try**{  *count*++;  } **finally** {  ***lock***.unlock();  }  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  } } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** LockExample3 {  **private final** Map<String, Date> **map** = **new** TreeMap<>();  **private final static** ReentrantReadWriteLock ***lock*** = **new** ReentrantReadWriteLock();  **private final static** Lock ***readLock*** = ***lock***.readLock();  **private final static** Lock ***writeLock*** = ***lock***.writeLock();   **public** Date get(String key){  ***readLock***.lock();  **try** {  **return map**.get(key);  } **finally** {  ***readLock***.unlock();  }  }   **public** Set<String> getAllKeys(){  ***readLock***.lock();  **try** {  **return map**.keySet();  } **finally** {  ***readLock***.unlock();  }  }   **public** Date put(String key, Date value){  ***writeLock***.lock();  **try**{  **return map**.put(key, value);  } **finally** {  ***writeLock***.unlock();  }  } } |

|  |
| --- |
| **public class** LockExample4 {   **class** Point {  **private double x**, **y**;  **private final** StampedLock **sl** = **new** StampedLock();   **void** move(**double** deltaX, **double** deltaY) { *// an exclusively locked method* **long** stamp = **sl**.writeLock();  **try** {  **x** += deltaX;  **y** += deltaY;  } **finally** {  **sl**.unlockWrite(stamp);  }  }   *//下面看看乐观读锁案例* **double** distanceFromOrigin() { *// A read-only method* **long** stamp = **sl**.tryOptimisticRead(); *//获得一个乐观读锁* **double** currentX = **x**, currentY = **y**; *//将两个字段读入本地局部变量* **if** (!**sl**.validate(stamp)) { *//检查发出乐观读锁后同时是否有其他写锁发生？* stamp = **sl**.readLock(); *//如果没有，我们再次获得一个读悲观锁* **try** {  currentX = **x**; *// 将两个字段读入本地局部变量* currentY = **y**; *// 将两个字段读入本地局部变量* } **finally** {  **sl**.unlockRead(stamp);  }  }  **return** Math.*sqrt*(currentX \* currentX + currentY \* currentY);  }   *//下面是悲观读锁案例* **void** moveIfAtOrigin(**double** newX, **double** newY) { *// upgrade  // Could instead start with optimistic, not read mode* **long** stamp = **sl**.readLock();  **try** {  **while** (**x** == 0.0 && **y** == 0.0) { *//循环，检查当前状态是否符合* **long** ws = **sl**.tryConvertToWriteLock(stamp); *//将读锁转为写锁* **if** (ws != 0L) { *//这是确认转为写锁是否成功* stamp = ws; *//如果成功 替换票据* **x** = newX; *//进行状态改变* **y** = newY; *//进行状态改变* **break**;  } **else** { *//如果不能成功转换为写锁* **sl**.unlockRead(stamp); *//我们显式释放读锁* stamp = **sl**.writeLock(); *//显式直接进行写锁 然后再通过循环再试* }  }  } **finally** {  **sl**.unlock(stamp); *//释放读锁或写锁* }  }  } } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** LockExample5 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static int** *count* = 0;  **private final static** StampedLock ***lock*** = **new** StampedLock();  **private static void** add(){  **long** stamp = ***lock***.writeLock();  **try**{  *count*++;  } **finally** {  ***lock***.unlockWrite(stamp);  }  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  } } |

|  |
| --- |
| @Slf4j **public class** LockExample6 {  **public static void** main(String[] args) {  ReentrantLock reentrantLock = **new** ReentrantLock();  Condition condition = reentrantLock.newCondition();  **new** Thread(()->{  reentrantLock.lock();  log.info(**"{},wait signal"**,Thread.currentThread().getName());  **try** {  condition.await();  } **catch** (InterruptedException e) {  log.warn(**"exception"**, e);  }  log.info(**"{},get signal"**,Thread.currentThread().getName());  reentrantLock.unlock();  }).start();  **new** Thread(()->{  reentrantLock.lock();  log.info(**"{},get lock"**,Thread.currentThread().getName());  **try** {  Thread.sleep(2000);  } **catch** (InterruptedException e) {  log.warn(**"exception"**, e);  }  log.info(**"{},send signal"**,Thread.currentThread().getName());  condition.signal();  reentrantLock.unlock();  }).start();  } } |

JVM层面：

1. Synchronized：不但能够实现监控，而且代码执行异常JVM也会释放异常，并发不高时使用最优；

对象层面：

2.ReentrantLock线程增长趋势可预估；

3.ReentrantReadWriteLock提高读多写少的并发效率；

4.StampedLock写、读、乐观读，版本和模式；

5.Condition信号通知协作。

## 第8节J.U.C组件拓展

### 8.1FutureTask

·Callable与Runnable接口对比；

被线程执行后能返回值，并能够抛出异常。

·Future接口

查询的任务是否被取消，查询是否完成以及获取结果。通常线程都是义务计算模型。通常不能从其他线程返回结果，Future可以从得到其他线程的返回值。Future可以监视目标线程调用的情况。

·FutureTask类实现Future和Runnable接口

有一个很费时的逻辑需要计算，并且返回这个值，同时该值又不是马上需要，那么就可以使用这个组合。用另外一个线程计算返回值，而当前线程在使用返回值之前，可以做其他操作，等到需要结果，通过FutureTask的get其他线程获取结果。

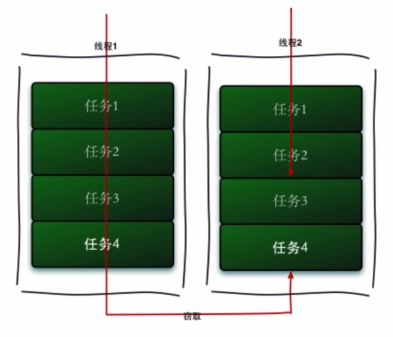
|  |
| --- |
| @Slf4j **public class** FutureExample {  **static class** MyCallable **implements** Callable<String>{  @Override  **public** String call() **throws** Exception {  ***log***.info(**"do something in callable."**);  Thread.*sleep*(5000);  **return "Done"**;  }  }   **public static void** main(String[] args) **throws** InterruptedException, ExecutionException {  ExecutorService service = Executors.*newCachedThreadPool*();  Future<String> future = service.submit(**new** MyCallable());  ***log***.info(**"do somethind in main."**);  Thread.*sleep*(1000);  String result = future.get();  ***log***.info(**"result:{}"**, result);  service.shutdown();  } } |
| 16:00:20.797 [pool-1-thread-1] INFO com.byf.concurrency.example.aqs.FutureExample - do something in callable.  16:00:20.797 [main] INFO com.byf.concurrency.example.aqs.FutureExample - do somethind in main.  16:00:25.802 [main] INFO com.byf.concurrency.example.aqs.FutureExample - result:Done |

|  |
| --- |
| @Slf4j **public class** FutureTaskExample {    **public static void** main(String[] args) **throws** InterruptedException, ExecutionException {  FutureTask<String> futureTask = **new** FutureTask<>(**new** Callable<String>() {  @Override  **public** String call() **throws** Exception {  ***log***.info(**"do something in callable."**);  Thread.*sleep*(5000);  **return "Done"**;  }  });   **new** Thread(futureTask).start();  ***log***.info(**"do something in main."**);  Thread.*sleep*(1000);  String result = futureTask.get();  ***log***.info(**"result:{}"**, result);  } } |
| 16:11:08.062 [main] INFO com.byf.concurrency.example.aqs.FutureTaskExample - do something in main.  16:11:08.062 [Thread-0] INFO com.byf.concurrency.example.aqs.FutureTaskExample - do something in callable.  16:11:13.068 [main] INFO com.byf.concurrency.example.aqs.FutureTaskExample - result:Done |

### 8.2Fork/Join框架

·JDK1.7，分而治之

·工作窃取



·双端队列（被窃取线程从头取，窃取线程从尾部取，只有一个任务存在竞争）

·充分利用线程，并减少竞争；

·消耗更多资源，例如需要更多的双端队列、多个双端队列；

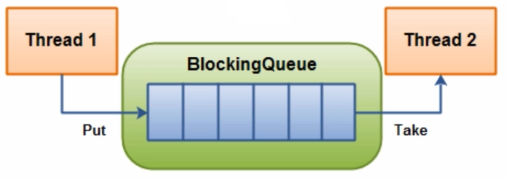
·对于Fork/Join框架而言，当一个任务正在等待它使用Join操作创建的子任务结束时，执行这个任务的工作线程，查找其他未被执行的任务，并开始它的执行。通过这种方式，充分利用线程的执行时间，来提高性能。

·任务局限性：

1. 任务只能使用Fork/Join操作来作为同步机制，如果使用其他同步机制，在同步操作时，工作线程不能执行其他任务。例如：在Fork/Join框架中，是任务进入睡眠，睡眠期间内，正在执行这个任务的工作线程就不能执行其他任务；
2. 拆分的任务不应去执行IO操作，如写数据文件；
3. 任务不能抛出检查异常，必须通过build代码处理他们。

|  |
| --- |
| @Slf4j **public class** ForkJoinTaskExample **extends** RecursiveTask<Integer> {  **public static final int *threshold*** = 2;  **private int start**;  **private int end**;  **public** ForkJoinTaskExample(**int** start, **int** end){  **this**.**start** = start;  **this**.**end** = end;  }  @Override  **protected** Integer compute() {  **int** sum = 0;   *// 如果任务足够小，就计算任务* **boolean** canCompute = (**end** - **start**) <= ***threshold***;  **if** (canCompute){  **for** (**int** i = **start**; i <= **end**; i++) {  sum += i;  }  } **else** {  *// 如果子任务大于阈值，就分裂成两个种子任务计算；* **int** middle = (**start** + **end**) / 2;  ForkJoinTaskExample leftTask = **new** ForkJoinTaskExample(**start**, middle);  ForkJoinTaskExample rightTask = **new** ForkJoinTaskExample(middle +1, **end**);  leftTask.fork();  rightTask.fork();  *// 等待任务执行结束合并结果；* **int** leftResult = leftTask.join();  **int** rightResult = rightTask.join();  sum = leftResult + rightResult;  }  **return** sum;  }   **public static void** main(String[] args) {  ForkJoinPool forkJoinPool = **new** ForkJoinPool();  *// 生成一个计算任务，计算1+2+3+4* ForkJoinTaskExample task = **new** ForkJoinTaskExample(1,100);  *// 执行一个任务* Future<Integer> future = forkJoinPool.submit(task);  **try**{  ***log***.info(**"result:{}"**,future.get());  } **catch** (Exception e){  ***log***.error(**"exception"**, e);  }  } } |
| 16:49:51.114 [main] INFO com.byf.concurrency.example.aqs.ForkJoinTaskExample - result:5050 |

### 8.3BlockingQueue



·ArrayBlockingQueue：FIFO

·DelayQueue：

1. Delay接口继承自Comparable接口，排序，一般按照元素过期时间的优先级排序；
2. 应用场景：定时关闭连接，缓存对象，超时处理等；
3. 内部实现PriorityQueue和Lock。

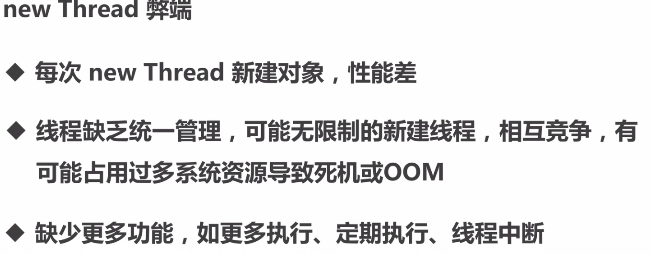
·LinkedBlockingQueue：不指定大小则无界；

·PriorityBlockingQueue：允许null，添加到队列的元素需要实现Comparable接口，而该队列提供的iterator并不一定按照实现Comparable接口的规则遍历；

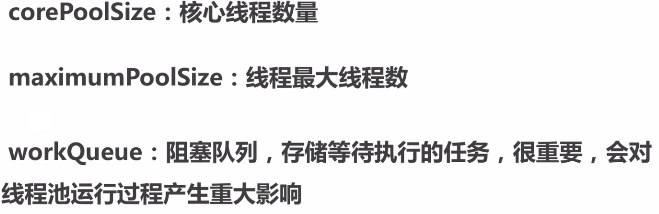
·SynchronizedQueue：仅允许容纳一个元素，同步队列，无界非缓存；

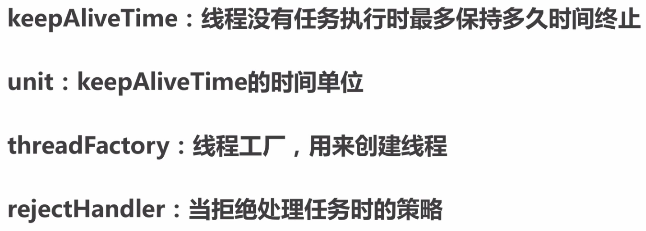
## 第9节线程池

1. new Thread

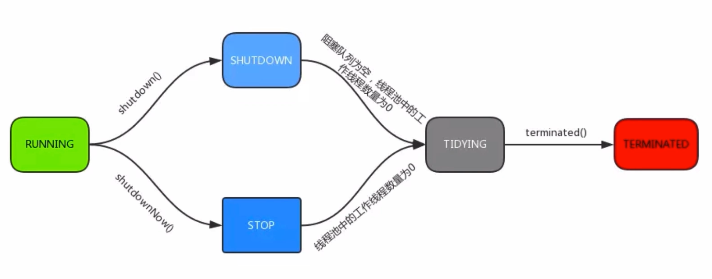


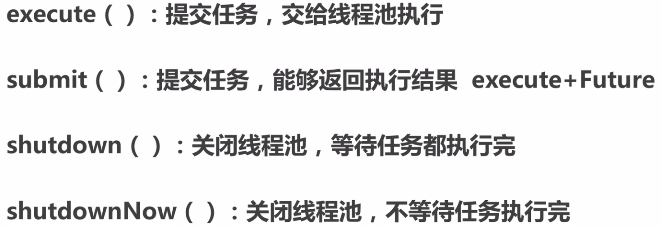
1. ThreadPoolExecutor

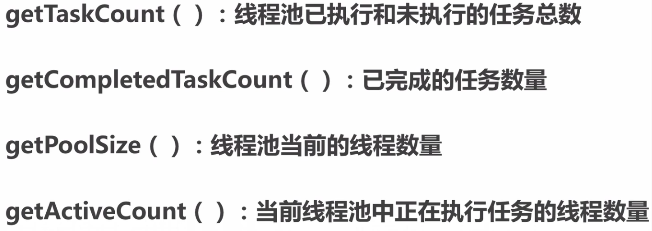




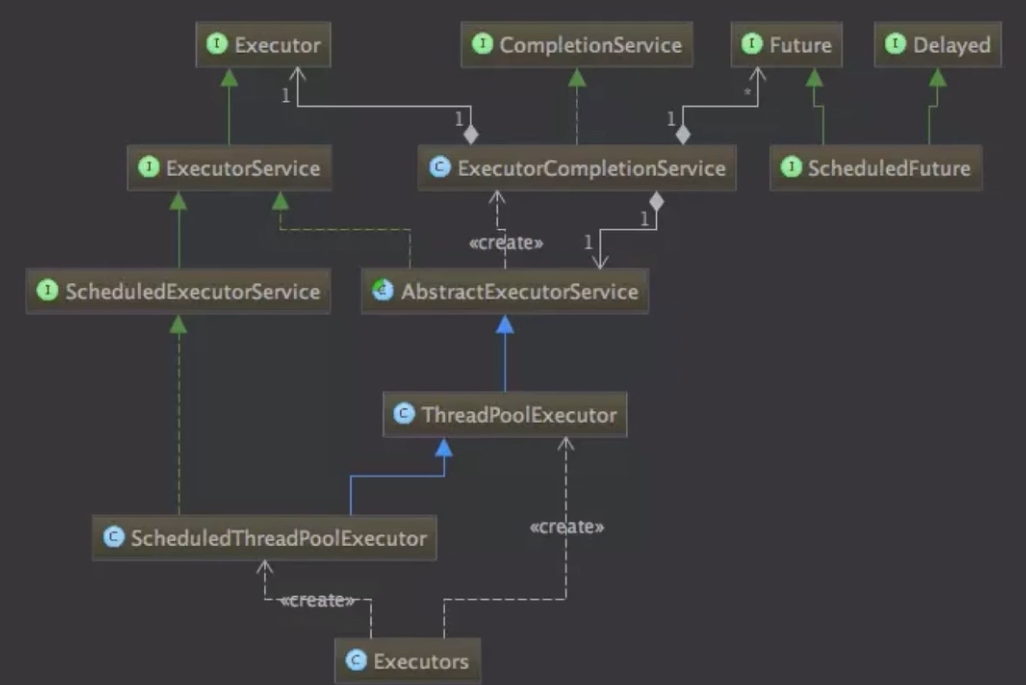
（3）状态转换







1. 线程池类图



1. Executor框架接口

·Executors.newCachedThreadPool：创建一个可缓存的线程池，需要处理的请求超过了线程池的长度，回收空闲线程，如果没有可回收线程，则创建新的线程；

·Executors.newFixedThreadPool：定长线程池，请求数量超过线程池长度，等待；

·Executors.newSecheduledThreadPool：定长线程池，定时周期性的执行请求；

·Executors.newSingleThreadExecutor：创建一个单线程化的线程池，创建一个唯一的线程执行任务，保证所有任务按指定顺序执行，顺序可以是FIFO，优先级。

|  |
| --- |
| @Slf4j **public class** ThreadPoolExample1 {  **public static void** main(String[] args) {  ExecutorService executorService = Executors.*newCachedThreadPool*();  **for** (**int** i = 0; i < 10; i++) {  **final int** index = i;  executorService.execute(**new** Runnable() {  @Override  **public void** run() {  ***log***.info(**"task:{}"**,index);  }  });  }  executorService.shutdown();  } } |

|  |
| --- |
| @Slf4j **public class** ThreadPoolExample2 {  **public static void** main(String[] args) {  ExecutorService executorService = Executors.*newFixedThreadPool*(5);  **for** (**int** i = 0; i < 10; i++) {  **final int** index = i;  executorService.execute(**new** Runnable() {  @Override  **public void** run() {  ***log***.info(**"task:{}"**,index);  }  });  }  executorService.shutdown();  } } |
| 10:40:45.584 [pool-1-thread-2] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:1  10:40:45.584 [pool-1-thread-4] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:3  10:40:45.594 [pool-1-thread-2] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:5  10:40:45.584 [pool-1-thread-3] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:2  10:40:45.584 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:0  10:40:45.584 [pool-1-thread-5] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:4  10:40:45.594 [pool-1-thread-4] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:6  10:40:45.594 [pool-1-thread-2] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:7  10:40:45.594 [pool-1-thread-3] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:8  10:40:45.594 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample2 - task:9 |

|  |
| --- |
| @Slf4j **public class** ThreadPoolExample3 {  **public static void** main(String[] args) {  ExecutorService executorService = Executors.*newSingleThreadExecutor*();  **for** (**int** i = 0; i < 10; i++) {  **final int** index = i;  executorService.execute(**new** Runnable() {  @Override  **public void** run() {  ***log***.info(**"task:{}"**,index);  }  });  }  executorService.shutdown();  } } |
| 10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:0  10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:1  10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:2  10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:3  10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:4  10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:5  10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:6  10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:7  10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:8  10:40:20.733 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample3 - task:9 |

|  |
| --- |
| @Slf4j **public class** ThreadPoolExample4 {  **public static void** main(String[] args) {  ScheduledExecutorService executorService = Executors.*newScheduledThreadPool*(5); *// executorService.schedule(new Runnable() { // @Override // public void run() { // log.info("schedule run"); // } // }, 3, TimeUnit.SECONDS);* executorService.scheduleAtFixedRate(**new** Runnable() {  @Override  **public void** run() {  ***log***.info(**"schedule run"**);  }  },1,3,TimeUnit.***SECONDS***);  *// executorService.shutdown();* Timer timer = **new** Timer();  timer.schedule(**new** TimerTask() {  @Override  **public void** run() {  ***log***.info(**"task run"**);  }  }, **new** Date(), 5 \* 1000);  } } |
| 10:39:22.675 [Timer-0] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample4 - task run  10:39:23.675 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample4 - schedule run  10:39:26.675 [pool-1-thread-1] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample4 - schedule run  10:39:27.675 [Timer-0] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample4 - task run  10:39:29.675 [pool-1-thread-2] INFO com.byf.concurrency.example.threadpool.ThreadPoolExample4 - schedule run |

## 第10节多线程并发拓展

### 10.1死锁

·互斥；

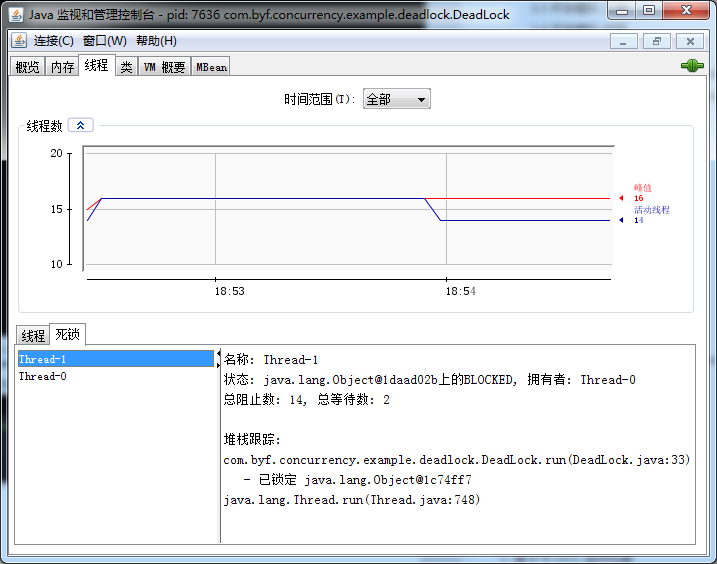
·请求和保持；

·不可剥夺；

·环路等待。

|  |
| --- |
| @Slf4j **public class** DeadLock **implements** Runnable {  **public int flag** = 1;  **private static** Object *o1* = **new** Object(), *o2* = **new** Object();  @Override  **public void** run() {  **if** (**flag** == 1){  ***log***.info(**"flag : {}"**,**flag**);  **synchronized** (*o1*){  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **synchronized** (*o2*){  ***log***.info(**"flag : {}"**,**flag**);  }  }  }  **if** (**flag** == 0){  ***log***.info(**"flag : {}"**,**flag**);  **synchronized** (*o2*){  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  **synchronized** (*o1*){  ***log***.info(**"flag : {}"**,**flag**);  }  }  }  }   **public static void** main(String[] args) {  DeadLock td1 = **new** DeadLock();  DeadLock td2 = **new** DeadLock();  td1.**flag** = 1;  td2.**flag** = 0;  **new** Thread(td1).start();  **new** Thread(td2).start();  } } |
|  |

在控制台打开jconsole检测死锁：



### 10.2并发最佳实践

·使用本地变量；

·使用不可变类；

·最小化锁的作用范围：S=1/(1 - a + a/n);

·使用线程池的Executor，而不是new Thread()；

·宁可使用同步也不要使用线程的wait和notify；

·使用BlockingQueue实现生产--消费模式；

·使用并发集合而不是加了锁的同步；

·使用Semaphone创建有界的访问；

·宁可使用同步代码块，也不实用同步方法；

·避免使用静态变量；

### 10.3Spring与线程安全

·Spring bean：singleton(default)、prototype；

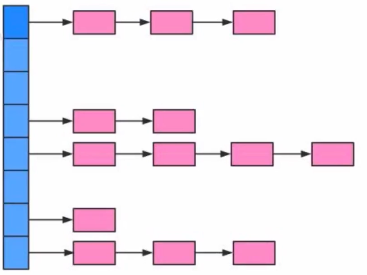
·无状态对象：不会应为多线程而导致状态被破坏的对象，线程安全；

DTO、VO、Controller（Singleton基于方法；structs基于类）、Service、局部变量；

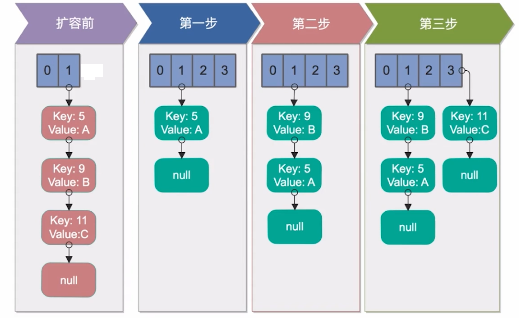
·有状态对象：包装在ThreadLocal的map容器，管理使其变为线程私有；

### 10.4HashMap与ConcurrentHashMap

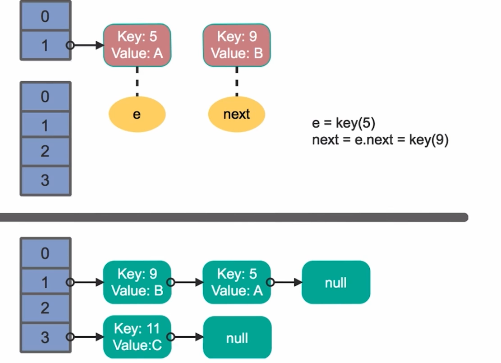
·HashMap容量是2的n次方，由于取余效率问题，2的n次方哈希取余与位移相同；

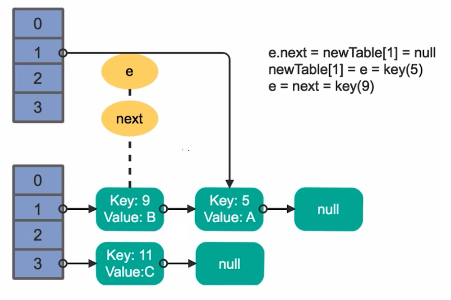


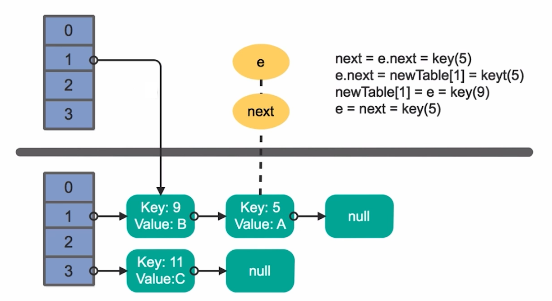
·单线程rehash，位置反序

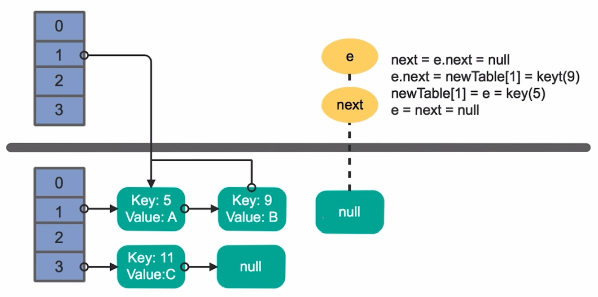


·多线程rehash引入环路问题

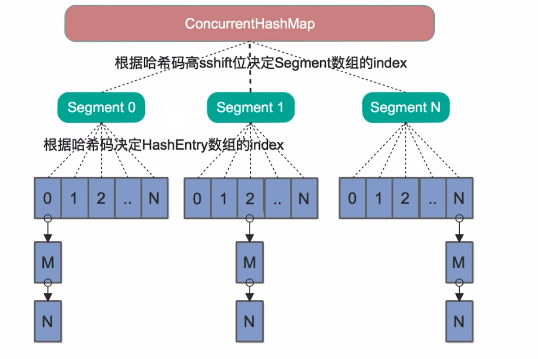








ConcurrentHashMap：Segment



HashMap与ConcurrentHashMap区别：

·HashMap线程不安全，ConcurrentHashMap线程安全；

·HashMap允许null值null键，ConcurrentHashMap不允许key为空；

·HashMap不允许Iterator遍历的同时对HashMap修改，而ConcurrentHashMap允许，并且更新对后续遍历可见；

JDK7和JDK8中ConcurrentHashMap区别：

·JDK7在hash取余设计使用Segment，防止Hash冲突引起性能问题，而JDK8中引入红黑树，并且没有在Hash冲突上做过多的设计，而是在碰撞后，存入红黑树；

JDK1.7和JDK1.8中ConcurrentHashMap的区别

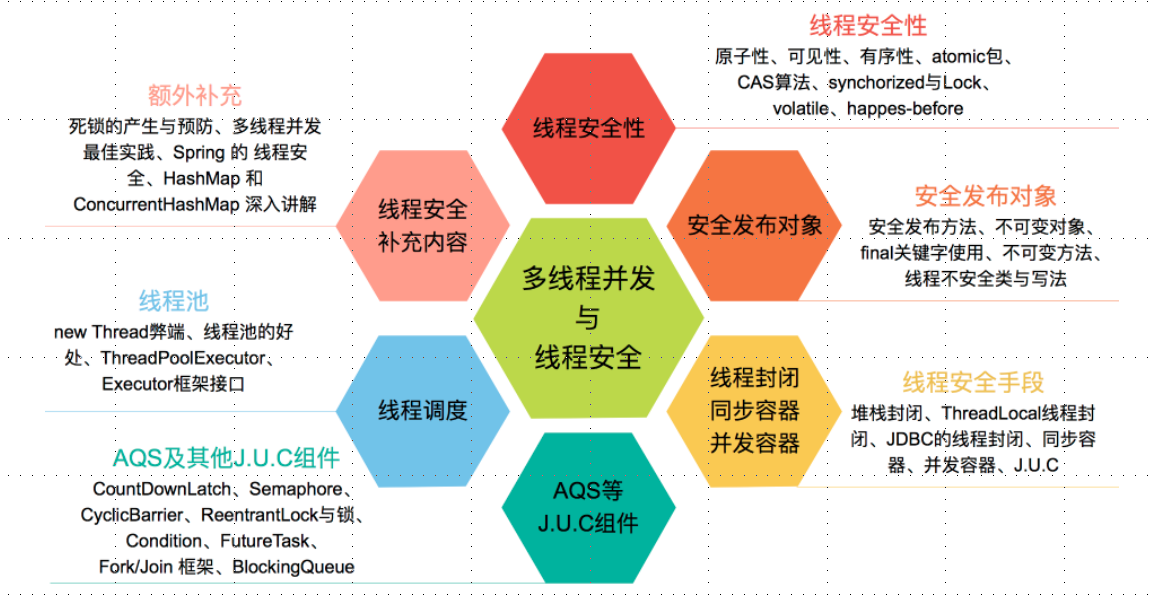
1、在JDK1.8中ConcurrentHashMap的实现方式有了很大的改变，在JDK1.7中采用的是Segment + HashEntry，而Sement继承了ReentrantLock，所以自带锁功能，而在JDK1.8中则取消了Segment，作者认为Segment太过臃肿，采用Node + CAS + Synchronized（ps:Synchronized一直以来被各种吐槽性能差，但java一直没有放弃Synchronized，也一直在改进，既然作者在这里采用了Synchronized，可见Synchronized的性能应该是有所提升的，当然只是猜想哈哈哈。。。）

2、在上篇HashMap中我们知道，在JDK1.8中当HashMap的链表个数超过8时，会转换为红黑树，在ConcurrentHashMap中也不例外。这也是新能的一个小小提升。

3、在JDK1.8版本中，对于size的计算，在扩容和addCount()时已经在处理了。JDK1.7是在调用时才去计算。

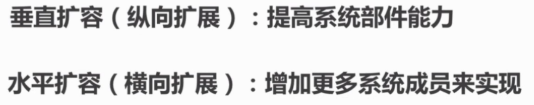
为了帮助统计size，ConcurrentHashMap提供了baseCount和counterCells两个辅助变量和CounterCell辅助类，1.8中使用一个volatile类型的变量baseCount记录元素的个数，当插入新数据或则删除数据时，会通过addCount()方法更新baseCount。

### 10.5多线程总结



## 第11节高并发之扩容思路

·扩容



·扩容--数据库

