

Dozer

1.简介

2.应用场景

3.使用方法

4.Spring配置

1.简介

dozer是一种JavaBean的映射工具，类似于Apache的BeanUtils。它可以灵活地处理复杂类型之间的映射。不但可以进行简单的属性映射、复杂的类型映射、双向映射、递归映射等，并且可以通过XML配置文件进行灵活的配置。

2.使用场景

1.数据存储之间Bean的转化

数据既要存储在MySQL，又要存储在ElasticSearch中，中间有一个Bean的转换，在大多数情况下，只是简单的赋值而已，如右：

```
public AdvertisementEsBean(DspAdvertisementDO advertisementDO) {  
    this.id = advertisementDO.getId();  
    this.planId = advertisementDO.getPlanId();  
    this.planName = advertisementDO.getPlanName();  
    this.userId = advertisementDO.getUserId();  
    this.status = advertisementDO.getStatus();  
    this.name = advertisementDO.getName();  
}
```

2.数据传输过程DO,DTO,VO层之间的转化

表现层与应用层之间是通过数据传输对象（DTO）进行交互的，数据传输对象的目的是为了对领域对象进行数据封装，实现层与层之间的数据传递。在此过程中，需要DO层，DTO，VO层之间的转换，如右：

```
public DspAdvertisementListVO(AdvertisementEsBean m) {  
    setId(m.getId());  
    setName(m.getName());  
    setPlanId(m.getPlanId());  
    setPlanName(m.getPlanName());  
    setStatus(m.getStatus());  
    setUserId(m.getUserId());  
    setBiddingPrice(NumUtils.get100DiviceNum(m.getBiddingPrice()));  
    setReason(m.getReason());  
    setCreateTime(m.getCreateTime());  
    setMaterialsInfo(m.getMaterialsInfo());  
    setPv(0L);  
    setClick(0L);  
    setConsume(BigDecimal.ZERO);  
    setCtr(BigDecimal.ZERO);  
    setDownload(0L);  
    setCpc(BigDecimal.ZERO);  
}
```

3.使用方法

1.源数据对象和目标数据对象都共享相同的公共属性名称。3.使用方法

1.源对象：

```
public class Source {  
    private String name;  
    private int age;  
  
    public Source() {  
    }  
  
    public Source(String name, int age) {  
        super();  
        this.name = name;  
        this.age = age;  
    }  
  
    //省略 get/set方法  
}
```

```
@Before  
public void initMapper() {  
    mapper = new DozerBeanMapper();  
}
```

2.目标对象：

```
public class DozerSource {  
    private String name;  
    private int age;  
  
    public DozerSource() {  
    }  
  
    public DozerSource(String name, int age) {  
        super();  
        this.name = name;  
        this.age = age;  
    }  
  
    //省略 get/set方法  
}
```

```
@Test  
public void getDozerSourceObject(){  
    Source source = new Source("wade", 35);  
    DozerSource dest = mapper.map(source, DozerSource.class);  
    assertEquals(dest.getName(), "wade");  
    assertEquals(dest.getAge(), 35);  
  
    assertEquals(dest.getAge(), source.getAge());  
    assertEquals(dest.getName(), source.getName());  
}
```

3.使用方法

2.源数据对象与目标对象属性名称相同，数据类型不同

1.源对象：

```
public class Source2 {  
    private String id;  
    private double points;  
  
    public Source2() {  
    }  
  
    public Source2(String id, double points) {  
        super();  
        this.id = id;  
        this.points = points;  
    }  
}
```

2.目标对象：

```
public class DozerSource2 {  
    private int id;  
    private int points;  
  
    public DozerSource2() {  
    }  
  
    public DozerSource2(int id, int points) {  
        super();  
        this.id = id;  
        this.points = points;  
    }  
}
```

3.使用方法

```
@Before  
public void initMapper() {  
    mapper = new DozerBeanMapper();  
}
```

```
@Test  
public void givenDozerSource2() {  
    Source2 source = new Source2("320", 15.2);  
    DozerSource2 dest = mapper.map(source, DozerSource2.class);  
  
    assertEquals(dest.getId(), 320);  
    assertEquals(dest.getPoints(), 15);  
}
```

3.使用方法

3.源数据对象与目标对象属性名称不同（xml与注解）

1.源对象：

```
public class Person {  
    private String name;  
    private String address;  
    private int age;  
    public Person() {  
  
    }  
    public Person(String name, String address, int age) {  
        this.name = name;  
        this.address = address;  
        this.age = age;  
    }  
}
```

2.目标对象：

```
public class People {  
    private String peopleName;  
    private String peopleAddress;  
    private int age;  
  
    public People() {  
    }  
  
    public People(String peopleName, String peopleAddress,  
int age) {  
        this.peopleName = peopleName;  
        this.peopleAddress = peopleAddress;  
        this.age = age;  
    }  
}
```

3.使用方法

3.1配置xml配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>  
<mappings xmlns="http://dozer.sourceforge.net"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://dozer.sourceforge.net  
        http://dozer.sourceforge.net/schema/beanmapping.xsd">  
    <mapping>  
        <class-a>com.learn.dozer.People</class-a>  
        <class-b>com.learn.dozer.Person</class-b>  
        <field>  
            <a>peopleName</a>  
            <b>name</b>  
        </field>  
        <field>  
            <a>peopleAddress</a>  
            <b>address</b>  
        </field>  
    </mapping>  
</mappings>
```

3.2使用

```
//读取配置文件  
private void configureMapper(String... configure) {  
    mapper.setMappingFiles(Arrays.asList(configure));  
}  
//根据配置文件来进行bean的装换  
@Test  
public void giveObjectByXML() {  
    configureMapper("dozer_mapping.xml");  
    Person a=new Person("a","b",2);  
    People b=mapper.map(a,People.class);  
    assertEquals(b.getPeopleName(),"a");  
    assertEquals(b.getPeopleAddress(),"b");  
    assertEquals(b.getAge(),2);  
}
```

4.Spring配置与原理

1.注解方式配置

```
@Bean
public DozerBeanMapper instance(){
    DozerBeanMapper mapper=new DozerBeanMapper();
    mapper.setMappingFiles(Arrays.asList("dozer_mapping.xml"));
    return mapper;
}
```

2.xml方式配置

```
<bean id="mapper" class="org.dozer.DozerBeanMapper">
    <property name="mappingFiles">
        <list>
            <value>classpath*:dozer/dozer-mapping.xml</value>
        </list>
    </property>
</bean>
```

3.原理

底层用的是Java反射与Java克隆机制，根据Field生成目标类

```
// Perform mappings for each field. Iterate through Fields Maps for this
class mapping
for (FieldMap fieldMapping : classMap.getFieldMaps()) {
    //Bypass field if it has already been mapped as part of super class
    mappings.
    String key = MappingUtils.getMappedParentFieldKey(destObj,
fieldMapping);
    if (mappedParentFields != null && mappedParentFields.contains(key)) {
        continue;
    }
    mapField(fieldMapping, srcObj, destObj);
}
```

Disruptor

1.简介

2.应用场景

3.使用方法

4.原理介绍

5.Disruptor为什么这么快原因

1.简介

Disruptor是一个开源框架，研发的初衷是为了解决高并发下队列锁的问题，最早由LMAX（一种新型零售金融交易平台）提出并使用，能够在无锁的情况下实现队列的并发操作，并号称能够在单线程里每秒处理6百万笔订单。

2.使用场景

- 1.异步操作，如记录日志等
- 2.高并发，阻塞队列不能满足需求
- 3.生产者-消费者模式

2.使用场景-日志打印分析-logback

1.代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <!--无缓存，立即输出-->
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>e:/log.out</file>
        <append>true</append>
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} %p %c - %m%n</pattern>
        </encoder>
    </appender>

    <!--有缓存，不立即输出-->
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>e:/log.out</file>
        <append>true</append>
        <immediateFlush>false</immediateFlush>
        <bufferSize>8192</bufferSize>
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} %p %c - %m%n</pattern>
        </encoder>
    </appender>

    <!--异步appender-->
    <appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
        <discardingThreshold>0</discardingThreshold>
        <queueSize>128</queueSize>
        <appender-ref ref="FILE"/>
    </appender>

    <root level="info">
        <appender-ref ref="ASYNC" />
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

```
public class LogBackDemo {
    Logger logger = LoggerFactory.getLogger(LogBackDemo.class);

    //多线程
    @Test
    public void testThread() throws InterruptedException {
        int THREAD_NUM = 100;
        final int LOOP_NUM = 100000;

        final CountDownLatch countDownLatch = new CountDownLatch(THREAD_NUM);
        long start = System.currentTimeMillis();
        for(int x= 0;x < THREAD_NUM;x++){
            new Thread(new Runnable() {
                public void run() {
                    for (int y = 0; y < LOOP_NUM; y++) {
                        logger.info("Info Message!");
                    }
                    countDownLatch.countDown();
                }
            }).start();
        }
        countDownLatch.await();
        System.out.println(System.currentTimeMillis() - start);
    }

    //单线程
    @Test
    public void test() {
        int X_NUM = 100;
        int Y_NUM = 100000;

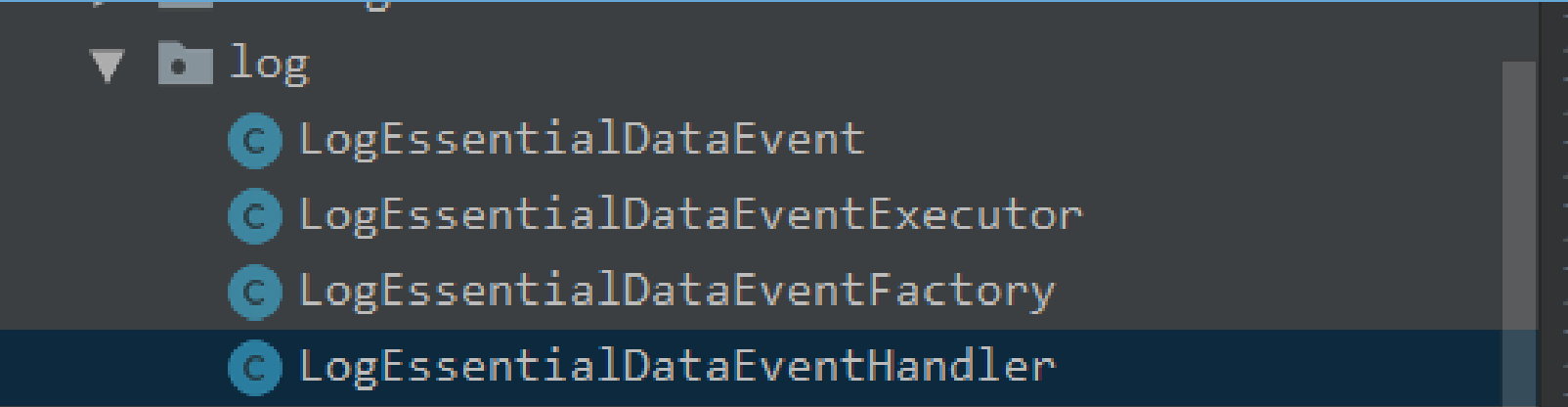
        long start = System.currentTimeMillis();
        for(int x=0;x<X_NUM;x++){
            for (int y = 0; y < Y_NUM; y++) {
                logger.info("Info Message!");
            }
        }
        System.out.print(System.currentTimeMillis()-start);
    }
}
```

2.使用场景-日志打印分析-logback

- 2.结论（只针对输出）：
- 1.开启缓存性能更好
 - 2.异步性能更好

	未开启缓存	开启缓存	异步未开启缓存	异步开启缓存
单线程	25423	5561	33102	9112
多线程	35644	6547	44203	5319

3.若日志收集也是异步的，性能更好



```
▼ log
  LogEssentialDataEvent
  LogEssentialDataEventExecutor
  LogEssentialDataEventFactory
  LogEssentialDataEventHandler
```

3.使用方法

1.注册事件

```
public class LogEvent {  
  
    private String message;  
  
    public String getMessage() {  
        return message;  
    }  
  
    public void setMessage(String message) {  
        this.message = message;  
    }  
}
```

2.事件工厂

```
public class LogEventFactory implements  
EventFactory<LogEvent> {  
    @Override  
    public LogEvent newInstance() {  
        return new LogEvent();  
    }  
}
```

3.事件处理器

```
public class LogEventHandler implements  
EventHandler<LogEvent> {  
    @Override  
    public void onEvent(LogEvent logEvent, long l, boolean  
b) throws Exception {  
        System.out.println(logEvent.getMessage());  
    }  
}
```

```
public class DisruptorMain {  
  
    public static void main(String[] args) throws InterruptedException {  
        Executor executor = Executors.newCachedThreadPool();  
        LogEventFactory factory = new LogEventFactory();  
        //ringBuffer的大小  
        int bufferSize = 256;  
        Disruptor<LogEvent> disruptor = new Disruptor<>(factory, bufferSize,  
executor,      ProducerType.SINGLE, new BlockingWaitStrategy());  
        //设置事件执行者  
        disruptor.handleEventsWith(new LogEventHandler());  
        disruptor.start();  
        RingBuffer<LogEvent> ringBuffer = disruptor.getRingBuffer();  
        //设置事件单生产者:  
        for (int x = 0; x < 256; x++) {  
            // 获取下一个可用位置的下标  
            long sequence = ringBuffer.next();  
            try {  
                // 返回可用位置的元素  
                LogEvent event = ringBuffer.get(sequence);  
                // 设置该位置元素的值  
                event.setMessage("hello,world");  
            } finally {  
                //发布事件  
                ringBuffer.publish(sequence);  
            }  
        }  
    }  
}
```

4.原理介绍

1.主要类功能介绍

类名	作用
Disruptor	Disruptor的入口，主要封装了环形队列RingBuffer、消费者集合ConsumerRepository的引用；主要提供了获取环形队列、添加消费者、生产者向RingBuffer中添加事件（可以理解为生产者生产数据）的操作
RingBuffer	Disruptor中队列具体的实现，底层封装了Object[]数组；在初始化时，会使用Event事件对数组进行填充，填充的大小就是bufferSize设置的值；此外，该对象内部还维护了Sequencer（序列生产器）具体的实现
Sequencer	序列生产器，分别有MultiProducerSequencer（多生产者序列生产器）和SingleProducerSequencer（单生产者序列生产器）两个实现类，以及维护了生产者与消费者序列冲突时候的等待策略WaitStrategy
Sequence	序列对象，内部维护了一个long型的value，这个序列指向了RingBuffer中Object[]数组具体的角标。生产者和消费者各自维护自己的Sequence；但都是指向RingBuffer的Object[]数组
WaitStrategy	等待策略。当没有可消费的事件时，消费者根据特定的策略进行等待；当没有可生产的地方时，生产者根据特定的策略进行等待
Event	事件对象，就是我们Ringbuffer中存在的数据，在Disruptor中用Event来定义数据
EventProcessor	事件处理器，单独在一个线程内执行，判断消费者的序列和生产者序列关系，决定是否调用我们自定义的事件处理器，也就是是否可以消费
EventHandler	事件处理器，由用户自定义实现，也就是最终的事件消费者，需要实现EventHandler接口
Producer	事件生产者，也就是我们上面代码中最后那部门的for循环

4.原理介绍

2.等待策略

类名	作用
BlockingWaitStrategy	通过线程阻塞的方式，等待生产者唤醒，被唤醒后，再循环检查依赖的sequence是否已经消费
BusySpinWaitStrategy	线程一直自旋等待，可能比较耗cpu
LiteBlockingWaitStrategy	线程阻塞等待生产者唤醒，与BlockingWaitStrategy相比，区别在signalNeeded.getAndSet, 如果两个线程同时访问一个访问waitfor,一个访问signalAll时，可以减少lock加锁次数
LiteTimeoutBlockingWaitStrategy	与LiteBlockingWaitStrategy相比，设置了阻塞时间，超过时间后抛异常
PhasedBackoffWaitStrategy	根据时间参数和传入的等待策略来决定使用哪种等待策略
TimeoutBlockingWaitStrategy	相对于BlockingWaitStrategy来说，设置了等待时间，超过后抛异常
YieldingWaitStrategy	尝试100次，然后Thread.yield()让出cpu

5.Disruptor为什么这么快的原因

1.解决伪共享

1.1 CPU缓存

在现代计算机当中，CPU是大脑，最终都是由它来执行所有的运算。而内存(RAM)则是血液，存放着运行的数据；但是，由于CPU和内存之间的工作频率不同，CPU如果直接去访问内存的话，系统性能将会受到很大的影响，所以在CPU和内存之间加入了三级缓存，分别是L1、L2、L3。

当CPU执行运算时，它首先会去L1缓存中查找数据，找到则返回；如果L1中不存在，则去L2中查找，找到即返回；如果L2中不存在，则去L3中查找，查到即返回。如果三级缓存中都不存在，最终会去内存中查找。对于CPU来说，走得越远，就越消耗时间，拖累性能。

从CPU到	大约需要的 CPU 周期	大约需要的时间
主存		约60-80纳秒
QPI 总线传输 (between sockets, not drawn)		约20ns
L3 cache	约40-45 cycles,	约15ns
L2 cache	约10 cycles,	约3ns
L1 cache	约3-4 cycles,	约1ns
寄存器	1 cycle	

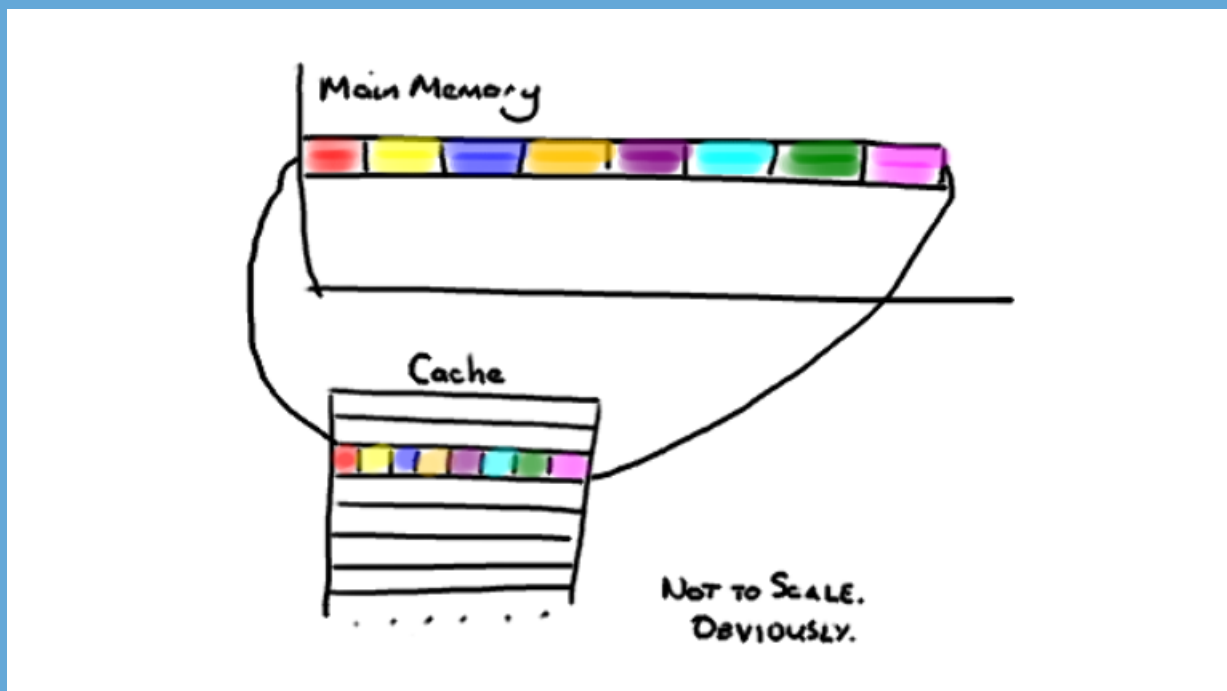
5.Disruptor为什么这么快的原因

1.2缓存行

在CPU缓存中，数据是以缓存行(cache line)为单位进行存储的，每个缓存行的大小一般为32--256个字节，常用CPU中缓存行的大小是64字节；CPU每次从内存中读取数据的时候，会将相邻的数据也一并读取到缓存中，填充整个缓存行

当我们遍历数组的时候，CPU遍历第一个元素时，与之相邻的元素也会被加载到了缓存中，对于后续的遍历来说，CPU在缓存中找到了对应的数据，不需要再去内存中查找，效率得到了巨大的提升

Disruptor底层是Object[]数组，因此性能比较高



5.Disruptor为什么这么快的原因

1.3缓存是否命中例子—模拟缓存是否命中

```
public class CacheHit {  
    //二维数组  
    private static long[][] longs;  
    //一维数组长度  
    private static int length = 1024 * 1024;  
    public static void main(String[] args) {  
        //创建二维数组,并赋值:  
        longs = new long[length][6];  
        for (int x = 0; x < length; x++) {  
            longs[x] = new long[6];  
            for (int y = 0; y < 6; y++) {  
                longs[x][y] = 1L;  
            }  
        }  
        cacheHit();  
        cacheMiss();  
    }  
    //缓存命中  
    private static void cacheHit() {  
        long sum = 0L;  
        long start = System.nanoTime();  
        for (int x = 0; x < length; x++) {  
            for (int y = 0; y < 6; y++) {  
                sum += longs[x][y];  
            }  
        }  
        System.err.println("sum : " + sum);  
        System.out.println("命中耗时: " + (System.nanoTime() - start));  
    }  
    //缓存未命中  
    private static void cacheMiss() {  
        long sum = 0L;  
        long start = System.nanoTime();  
        for (int x = 0; x < 6; x++) {  
            for (int y = 0; y < length; y++) {  
                sum += longs[y][x];  
            }  
        }  
        System.err.println("sum : " + sum);  
        System.out.println("未命中耗时: " + (System.nanoTime() - start));  
    }  
}
```

在Java中，一个long类型是8字节，而一个缓存行是64字节，因此一个缓存行可以存放8个long类型。但是，在内存的布局中，对象不仅包含了实例数据(long类型变量)，还包含了对象头。对象头在32位系统上占用8字节，而64位系统上占用16字节。

二维数组中填充了6个元素，占用了48字节

命中耗时：15877654

sum : 6291456

未命中耗时：57731743

sum : 6291456

差不多相差4倍

5.Disruptor为什么这么快的原因

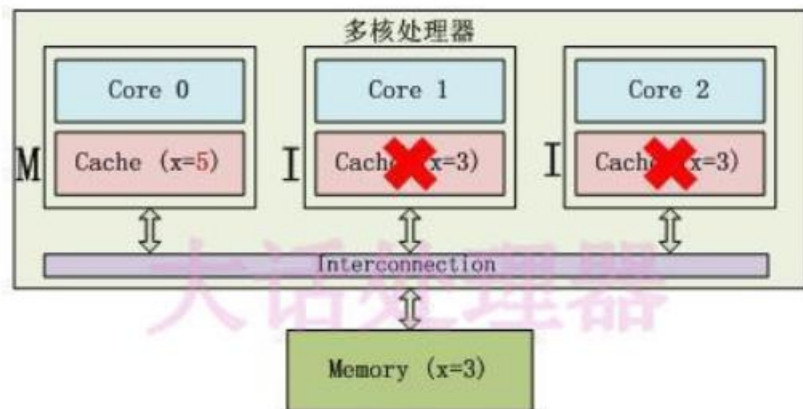
1.4多线程模拟伪共享

java程序中，当多个线程修改两个独立变量的时候，如果这两个变量存在于一个缓存行中，那么就有很大的概率产生伪共享

现如今，CPU都是多核处理器，一般为2核或者4核，当我们程序运行时，启动了多个线程。例如：核心1启动了1个线程，核心2启动了1个线程，这2个线程分别要修改不同的变量，其中核心1的线程要修改x变量，而核心2的线程要修改y变量，但是x、y变量在内存中是相邻的数据，他们被加载到了同一个缓存行当中，核心1的缓存行有x、y，核心2的缓存行也有x、y。那么，只要有一个核心中的线程修改了变量，另一个核心的缓存行就会失效，导致数据需要被重新到内存中读取，无意中影响了系统的性能，这就是伪共享。这就是缓存一致性协议(MESI)

CPU的伪共享问题本质是：

几个在内存中相邻的数据，被CPU的不同核心加载在同一个缓存行当中，数据被修改后，由于数据存在同一个缓存行当中，进而导致缓存行失效，引起缓存命中降低。



5.Disruptor为什么这么快的原因

1.4多线程模拟伪共享

	未注释代码1	注释掉代码1
第一次	16451743461	26666708578
第二次	17124046655	25214990415
第三次	17220128914	25372634597

1.5Disruptor中如何解决伪共享

通过缓存行填充的方式来解决伪共享
jdk并发包下也有这样解决伪共享

```
class LhsPadding
{
    protected long p1, p2, p3, p4, p5, p6, p7;
}

class Value extends LhsPadding
{
    protected volatile long value;
}

class RhsPadding extends Value
{
    protected long p9, p10, p11, p12, p13, p14,
    p15;
}
```

```
public class FalseShare implements Runnable {
    private static int NUM_THREADS = 4; //线程数、数组大小:
    private final static long ITERATIONS = 500L * 1000L * 1000L; //数组迭代的次数:
    private final int handleArrayIndex; //线程需要处理的数组元素角标:
    private static VolatileLong[] longs = new VolatileLong[NUM_THREADS]; //操作数组:
    static{
        for (int i = 0; i < longs.length; i++) {
            longs[i] = new VolatileLong();
        }
    }
    public FalseShare(final int handleArrayIndex) {
        this.handleArrayIndex = handleArrayIndex;
    }
    public static void main(final String[] args) throws Exception {
        final long start = System.nanoTime();
        Thread[] threads = new Thread[NUM_THREADS];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new FalseShare(i));
        }
        for (Thread t : threads) {
            t.start();
        }
        for (Thread t : threads) {
            t.join();
        }
        System.out.println(System.nanoTime() - start);
    }
    //对数组的元素进行操作:
    @Override
    public void run() {
        long i = ITERATIONS;
        while (0 != --i) {
            longs[handleArrayIndex].value = i;
        }
    }
    //数组元素:
    public final static class VolatileLong {
        public volatile long value = 0L;
        public long p1, p2, p3, p4, p5; //代码1
        public int p6; //代码1
    }
    //注释代码1前后理解伪共享
}
```

5.Disruptor为什么这么快的原因

2. RingBufferSize为2的倍数

求余操作本身也是一种高耗费的操作, 所以ringbuffer的size设成2的n次方, 可以利用位操作来高效实现求余

3.不使用锁, 用CAS机制

disruptor不使用锁, 使用CAS (Compare And Swap/Set) ,严格意义上说仍然是使用锁, 因为CAS本质上也是一种乐观锁, 只不过是CPU级别指令, 不涉及到操作系统, 所以效率很高(AtomicLong实现Sequence)

4.预读或者批量读

disruptor底层是数组, 借助数组底层存储是连续的, 可以批量加载到缓存中

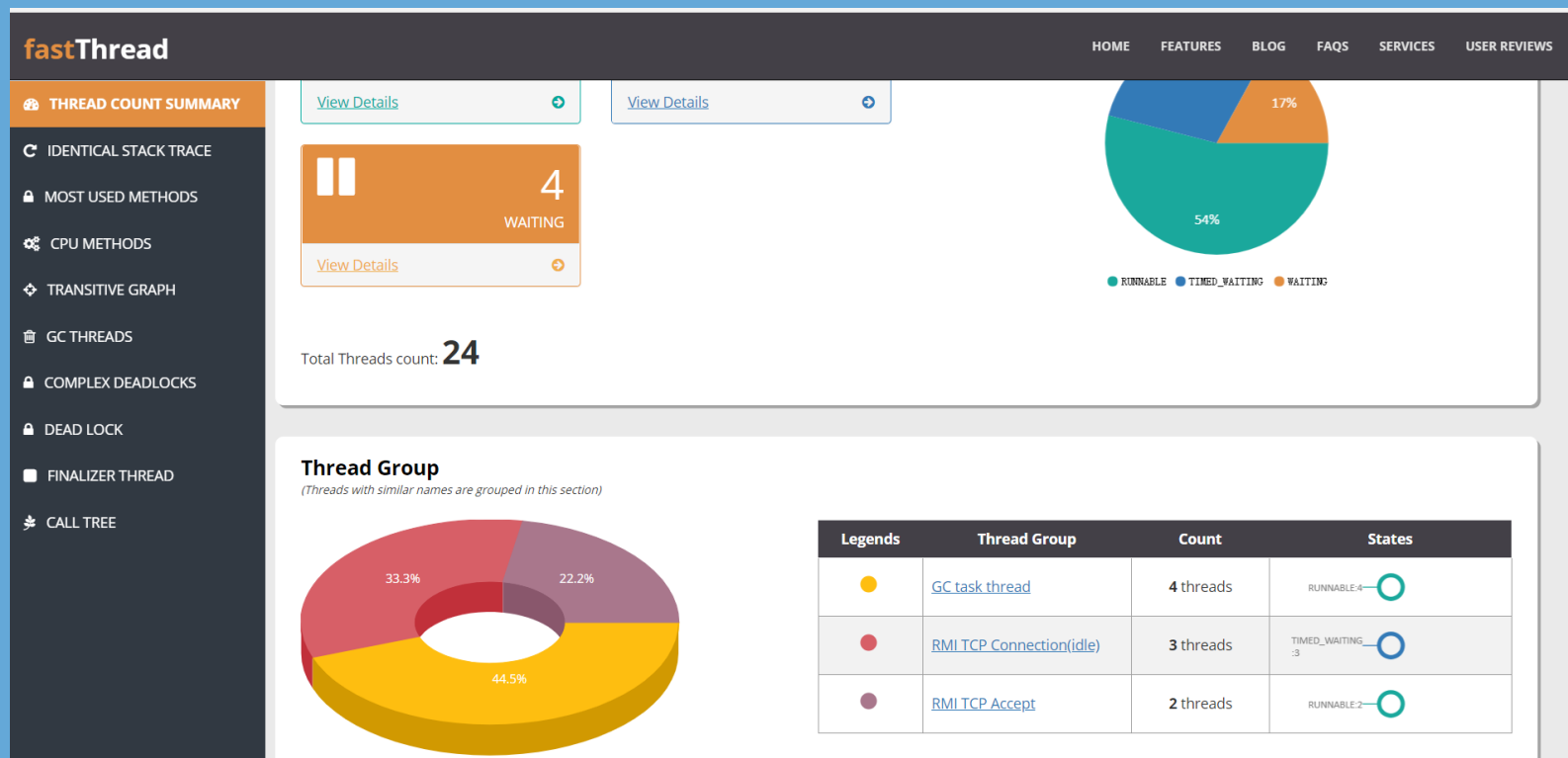
拓展：工具介绍

1.Jvm参数配置工具

<http://xxfox.perfma.com/jvm/generate>

2. JVM线程堆栈在线分析工具

<http://gceasy.io/>



谢谢