17 | ReadWriteLock：如何快速实现一个完备的缓存？

2019-04-06 王宝令

前面我们介绍了管程和信号量这两个同步原语在Java语言中的实现，理论上用这两个同步原语中任何一个都可以解决所有的并发问题。那Java SDK并发包里为什么还有很多其他的工具类呢？原因很简单：**分场景优化性能，提升易用性**。

今天我们就介绍一种非常普遍的并发场景：读多写少场景。实际工作中，为了优化性能，我们经常会使用缓存，例如缓存元数据、缓存基础数据等，这就是一种典型的读多写少应用场景。缓存之所以能提升性能，一个重要的条件就是缓存的数据一定是读多写少的，例如元数据和基础数据基本上不会发生变化（写少），但是使用它们的地方却很多（读多）。

针对读多写少这种并发场景，Java SDK并发包提供了读写锁——ReadWriteLock，非常容易使用，并且性能很好。

那什么是读写锁呢？

读写锁，并不是Java语言特有的，而是一个广为使用的通用技术，所有的读写锁都遵守以下三条基本原则：

1. 允许多个线程同时读共享变量；
2. 只允许一个线程写共享变量；
3. 如果一个写线程正在执行写操作，此时禁止读线程读共享变量。

读写锁与互斥锁的一个重要区别就是**读写锁允许多个线程同时读共享变量**，而互斥锁是不允许

的，这是读写锁在读多写少场景下性能优于互斥锁的关键。但**读写锁的写操作是互斥的**，当一个线程在写共享变量的时候，是不允许其他线程执行写操作和读操作。

快速实现一个缓存

下面我们就实践起来，用ReadWriteLock快速实现一个通用的缓存工具类。

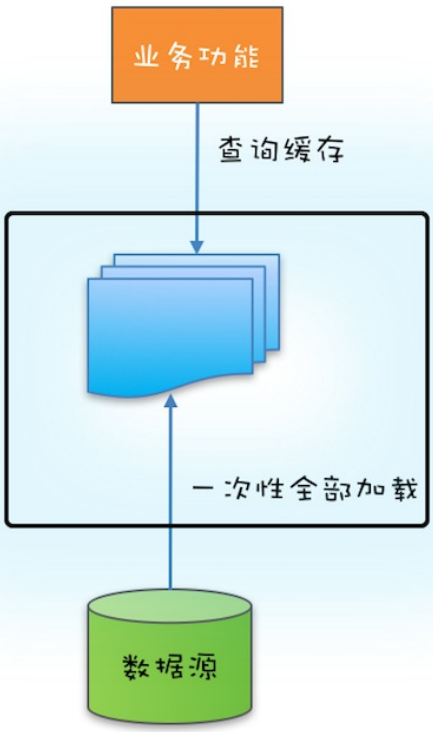
在下面的代码中，我们声明了一个Cache<K, V>类，其中类型参数K代表缓存里key的类型，V代表缓存里value的类型。缓存的数据保存在Cache类内部的HashMap里面，HashMap不是线程安全的，这里我们使用读写锁ReadWriteLock 来保证其线程安全。ReadWriteLock 是一个接口，它的实现类是ReentrantReadWriteLock，通过名字你应该就能判断出来，它是支持可重入的。下面我们通过rwl创建了一把读锁和一把写锁。

Cache这个工具类，我们提供了两个方法，一个是读缓存方法get()，另一个是写缓存方法put()。读缓存需要用到读锁，读锁的使用和前面我们介绍的Lock的使用是相同的，都是try{}finally{}这个编程范式。写缓存则需要用到写锁，写锁的使用和读锁是类似的。这样看来，读写锁的使用还是非常简单的。

```
class Cache<K,V> {
  final Map<K, V> m =
    new HashMap<>();
  final ReadWriteLock rwl =
    new ReentrantReadWriteLock();
  // 读锁
  final Lock r = rwl.readLock();
  // 写锁
  final Lock w = rwl.writeLock();
  // 读缓存
  V get(K key) {
    r.lock();
    try { return m.get(key); }
    finally { r.unlock(); }
  }
  // 写缓存
  V put(String key, Data v) {
    w.lock();
    try { return m.put(key, v); }
    finally { w.unlock(); }
  }
}
```

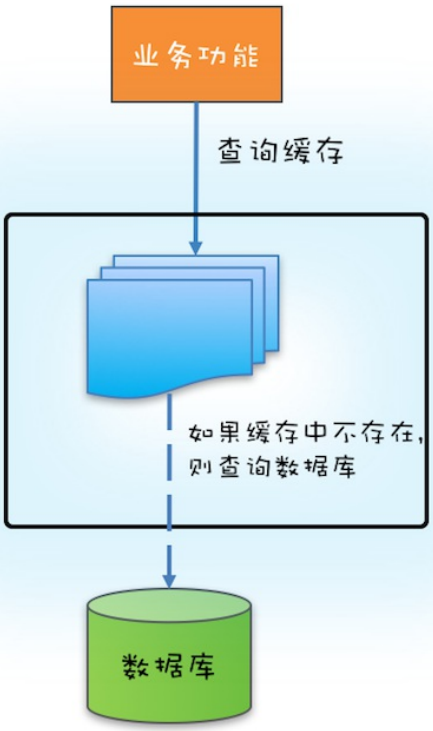
如果你曾经使用过缓存的话，你应该知道**使用缓存首先要解决缓存数据的初始化问题**。缓存数据的初始化，可以采用一次性加载的方式，也可以使用按需加载的方式。

如果源头数据的数据量不大，就可以采用一次性加载的方式，这种方式最简单（可参考下图），只需在应用启动的时候把源头数据查询出来，依次调用类似上面示例代码中的put()方法就可以了。



缓存一次性加载示意图

如果源头数据量非常大，那么就需要按需加载了，按需加载也叫懒加载，指的是只有当应用查询缓存，并且数据不在缓存里的时候，才触发加载源头相关数据进缓存的操作。下面你可以结合文中示意图看看如何利用ReadWriteLock 来实现缓存的按需加载。



缓存按需加载示意图

实现缓存的按需加载

文中下面的这段代码实现了按需加载的功能，这里我们假设缓存的源头是数据库。需要注意的是，如果缓存中没有缓存目标对象，那么就需要从数据库中加载，然后写入缓存，写缓存需要用到写锁，所以在代码中的⑤处，我们调用了 w.lock() 来获取写锁。

另外，还需要注意的是，在获取写锁之后，我们并没有直接去查询数据库，而是在代码⑥⑦处，重新验证了一次缓存中是否存在，再次验证如果还是不存在，我们才去查询数据库并更新本地缓存。为什么我们要再次验证呢？

```
class Cache<K,V> {
  final Map<K, V> m =
    new HashMap<>();
  final ReadWriteLock rwl =
    new ReentrantReadWriteLock();
  final Lock r = rwl.readLock();
  final Lock w = rwl.writeLock();

  V get(K key) {
    V v = null;
    //读缓存
    r.lock();         ①
    try {
      v = m.get(key); ②
    } finally{
      r.unlock();     ③
    }
    //缓存中存在，返回
    if(v != null) {   ④
      return v;
    }
    //缓存中不存在，查询数据库
    w.lock();         ⑤
    try {
      //再次验证
      //其他线程可能已经查询过数据库
      v = m.get(key); ⑥
      if(v == null){  ⑦
        //查询数据库
```

```
        //查询数据库
        v=省略代码无数
        m.put(key, v);
      }
    } finally{
      w.unlock();
    }
    return v;
  }
}
```

原因是在高并发的场景下，有可能会有多线程竞争写锁。假设缓存是空的，没有缓存任何东西，如果此时有三个线程T1、T2和T3同时调用get()方法，并且参数key也是相同的。那么它们会同时执行到代码⑤处，但此时只有一个线程能够获得写锁，假设是线程T1，线程T1获取写锁之后查询数据库并更新缓存，最终释放写锁。此时线程T2和T3会再有一个线程能够获取写锁，假设是T2，如果不采用再次验证的方式，此时T2会再次查询数据库。T2释放写锁之后，T3也会再次查询一次数据库。而实际上线程T1已经把缓存的值设置好了，T2、T3完全没有必要再次查询数据库。所以，再次验证的方式，能够避免高并发场景下重复查询数据的问题。

读写锁的升级与降级

上面按需加载的示例代码中，在①处获取读锁，在③处释放读锁，那是否可以在②处的下面增加验证缓存并更新缓存的逻辑呢？详细的代码如下。

```
//读缓存
r.lock();         ①
try {
  v = m.get(key); ②
  if (v == null) {
    w.lock();
    try {
      //再次验证并更新缓存
      //省略详细代码
    } finally{
      w.unlock();
    }
  }
} finally{
  r.unlock();     ③
}
```

这样看上去好像是没有问题的，先是获取读锁，然后再升级为写锁，对此还有个专业的名字，叫**锁的升级**。可惜ReadWriteLock并不支持这种升级。在上面的代码示例中，读锁还没有释放，此时获取写锁，会导致写锁永久等待，最终导致相关线程都被阻塞，永远也没有机会被唤醒。锁的升级是不允许的，这个你一定要注意。

不过，虽然锁的升级是不允许的，但是锁的降级却是允许的。以下代码来源自ReentrantReadWriteLock的官方示例，略做了改动。你会发现在代码①处，获取读锁的时候线程还是持有写锁的，这种锁的降级是支持的。

```
class CachedData {
  Object data;
  volatile boolean cacheValid;
  final ReadWriteLock rwl =
    new ReentrantReadWriteLock();
  // 读锁
  final Lock r = rwl.readLock();
  //写锁
  final Lock w = rwl.writeLock();

  void processCachedData() {
    // 获取读锁
    r.lock();
    if (!cacheValid) {
      // 释放读锁，因为不允许读锁的升级
      r.unlock();
      // 获取写锁
      w.lock();
      try {
        // 再次检查状态
        if (!cacheValid) {
          data = ...
          cacheValid = true;
        }
        // 释放写锁前，降级为读锁
        // 降级是可以的
        r.lock(); ①
      } finally {
        // 释放写锁
        w.unlock();
      }
    }
    // 此处仍然持有读锁
    try {use(data);}
    finally {r.unlock();}
  }
}
```

总结

读写锁类似于ReentrantLock，也支持公平模式和非公平模式。读锁和写锁都实现了java.util.concurrent.locks.Lock接口，所以除了支持lock()方法外，tryLock()、lockInterruptibly()等方法也都是支持的。但是有一点需要注意，那就是只有写锁支持条件变量，读锁是不支持条件变量的，读锁调用newCondition()会抛出UnsupportedOperationException异常。

今天我们用ReadWriteLock实现了一个简单的缓存，这个缓存虽然解决了缓存的初始化问题，但是没有解决缓存数据与源头数据的同步问题，这里的数据同步指的是保证缓存数据和源头数据的一致性。解决数据同步问题的一个最简单的方案就是**超时机制**。所谓超时机制指的是加载进缓存的数据不是长久有效的，而是有时效的，当缓存的数据超过时效，也就是超时之后，这条数据在缓存中就失效了。而访问缓存中失效的数据，会触发缓存重新从源头把数据加载进缓存。

当然也可以在源头数据发生变化时，快速反馈给缓存，但这个就要依赖具体的场景了。例如MySQL作为数据源头，可以通过近实时地解析binlog来识别数据是否发生了变化，如果发生了变化就将最新的数据推送给缓存。另外，还有一些方案采取的是数据库和缓存的双写方案。

总之，具体采用哪种方案，还是要看应用的场景。

课后思考

有同学反映线上系统停止响应了，CPU利用率很低，你怀疑有同学一不小心写出了读锁升级写锁的方案，那你该如何验证自己的怀疑呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言

密码123456 59

有多少跟我一样，发的内容能够看的懂。一到思考题，要么不会，要么心里的答案答非所问。

2019-04-06

crazypokerk 8

老师，可不可以这样理解，ReadWirteLock不支持锁的升级，指的是：在不释放读锁的前提下，无法继续获取写锁，但是如果在释放了读锁之后，是可以升级为写锁的。锁的降级就是：在不释放写锁的前提下，获取读锁是可以的。请老师指正，感谢。

2019-04-06

作者回复

可以这样理解，不过释放了读锁，也就谈不上升级了

2019-04-06

linqw 7

1、课后习题感觉可以使用第一种方法：①ps -ef | grep java查看pid②top -p查看java中的线程③使用jstack将其堆栈信息保存下来，查看是否是锁升级导致的阻塞问题。第二种方法：感觉可以调用下有获取只有读锁的接口，看下是否会阻塞，如果没有阻塞可以在调用下写锁的接口，如果阻塞表明有读锁。

2、读写锁也是使用volatile的state变量+加上happens-before来保证可见性么？

3、写下缓存和数据库的一致性问题的理解，如果先写缓存再写数据库，使用分布式锁，如果先写数据库再写缓存，①比如文中所说的使用binlog，canal+mq，但是感觉这个还得看具体情况

，有可能binlog使用了组提交，不是马上写的binlog文件中，感觉也是会有延迟②感觉也可以使用定时任务定时的扫描任务表③使用消息队列

2019-04-07

缪文@有赞 5

老师，感觉这里的读写锁，性能还有可以提升的地方，因为这里可能很多业务都会使用这个缓存懒加载，实际生产环境，写缓存操作可能会比较多，那么不同的缓存key，实际上是没有并发冲突的，所以这里的读写锁可以按key前缀拆分，即使是同一个key，也可以类似ConcurrentHash 一样分段来减少并发冲突

2019-04-07

作者回复

可以这样

2019-04-08

WL 3

老师我们现在的项目全都是集群部署, 感觉在这种情况下是不是单机的Lock,和Synchronized都用不上, 只能采用分布式锁的方案? 那么这种情况下, 如何提高每个实例的并发效率?

2019-04-09

作者回复

分布式有分布式的锁，单机的效率就是靠多线程了

2019-04-09

西西弗与卡夫卡 3

考虑到是线上应用，可采用以下方法

1. 源代码分析。查找ReentrantReadWriteLock在项目中的引用，看下写锁是否在读锁释放前尝试获取

2. 如果线上是Web应用，应用服务器比如说是Tomcat，并且开启了JMX，则可以通过JConsole等工具远程查看下线上死锁的具体情况

2019-04-06

作者回复

[]

2019-04-23

iron_man 2

王老师，写锁降级为读锁的话，前面的写锁是释放了么？后面可不可以讲一下这个读写锁的实现机制呢，这样可以对这种锁有更深入的理解，锁的升级降级也就不会用错了

2019-04-06

Dylan 2

一般都说线程池有界队列使用ArrayBlockingQueue，无界队列使用LinkedBlockingQueue，我很奇怪，有界无界不是取决于创建的时候传不传capacity参数么，我现在想创建线程池的时候，new LinkedBlockingQueue(2000)这样定义有界队列，请问可以吗？

2019-04-06

作者回复

可以，ArrayBlockingQueue有界是因为必须传capacity参数，LinkedBlockingQueue传capacity参数就是有界，不传就是无界

2019-04-06

ycfHH

1

问题1：获取写锁的前提是读锁和写锁均未被占用？
问题2：获取读锁的前提是没有其他线程占用写锁？
基于以上两点所以只支持锁降级而不允许锁升级。
问题3
高并发下，申请写锁时是不是中断其他线程申请读锁，然后等待已有读锁全部释放再获取写锁？因为如果没有禁止读锁的申请的话在读多写少的情况下写锁可能一直获取不到。
这块不太懂，希望老师能指点一下。

2019-05-07

作者回复

获取写锁的前提是读锁和写锁均未被占用

获取读锁的前提是没有其他线程占用写锁

申请写锁时不中断其他线程申请读锁

公平锁如果过有写申请，能禁止读锁

2019-05-12

xuery

1

读锁不能升级为写锁：好理解，本线程在释放读锁之前，想要获取写锁是不一定能获取到的，因为其他线程可能持有读锁（读锁共享），可能导致阻塞较长的时间，所以java干脆直接不支持读锁升级为写锁。
写锁可以降级为读锁：也好理解，本线程在释放写锁之前，获取读锁一定是可以立刻获取到的，不存在其他线程持有读锁或者写锁（读写锁互斥），所以java允许锁降级

2019-05-01

老杨同志

1

老师，如果读锁的持有时间较长，读操作又比较多，会不会一直拿不到写锁？

2019-04-06

作者回复

不会一直拿不到，只是等待的时间会很长

2019-04-06

zhangtnty

1

老师好，首先在机器启动未挂机时，监控JVM的GC运行指标，Survivor区一定持续升高，GC次数增多，而且释放空间有限。说明有线程肯定被持续阻塞。然后可以查看JVM的error.log，可以看到lock.BLOCK日志。可排查出锁的阻塞异常。要进一步排查，可review代码的锁使用情况。

2019-04-06

密码123456

1

系统停止了响应，说明线程可能被占满了。cpu利用率低为什么会推断出，是读锁升级为写锁？是因为锁升级后，线程都是等待状态吗？是不是cpu高是锁竞争？还有怎么验证读锁升级为写锁？

2019-04-06

作者回复

系统停止了响应,cpu利用率低大概率是死锁了，没法推断，只能大胆假设，小心求证

2019-04-06

Lemon 1

看线程的堆栈

2019-04-06

Jxin 0

有写禁止读。读锁无法升级成写锁。那么写锁持有锁时已经在执行逻辑的读锁会怎么样？有读不禁止写，那么读写锁的写锁啦饿又是什么情况下出现的？

2019-06-13

墨雨 0

关于思考题，直接在代码里找读写锁应用的地方看他有没有写错来验证呀，原谅我啥也不会233

2019-06-13

GEEKBANG_6638780 0

老杨同志

老师，如果读锁的持有时间较长，读操作又比较多，会不会一直拿不到写锁？

作者回复: 不会一直拿不到，只是等待的时间会很长

老师，你这个回答不对吧。我记得之前看源码是，读写锁，对于写锁是优先的。意思就是说，当有写线程时，哪怕有读线程在等待，都会优先分配给写线程的

2019-06-10

作者回复

进到等待队列，都是按顺序的，没听说写操作会强行排到前面。你可以参考《Java并发编程之美》，里面有代码分析。也可以参考https://sourceforge.net/projects/javaconcurrenta/来理解原理。

老杨同志的意思是，我猜测是，可能存在1000个读线程在读，而且读的慢，这个时候写操作就会等待的比较久。一旦有一个写操作等待，再有读操作的请求，就会进入等待队列里了，所以写操作不会一直拿不到锁。

2019-06-11

Delong 0

用jstack看是不是在waiting自己

2019-06-08

Geek_ebda96

0

老师既然读写锁的都是可以多线程读的，那为什么还要读锁，不是可以读的时候不加锁了么，有读锁的原因是不是因为有写锁，读到的时候要判断有没有写吧，如果没有写就读锁是没用的？

2019-06-01

文灏

0

王老师你好，有个问题想请教一下。既然允许多个线程同时读，那么这个时候的读锁意义在哪里？

2019-05-22

作者回复

不用关心可见性，原子性，读到的都是对的

2019-05-23