17-ReadWriteLock：如何快速实现一个完备的缓存？

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。

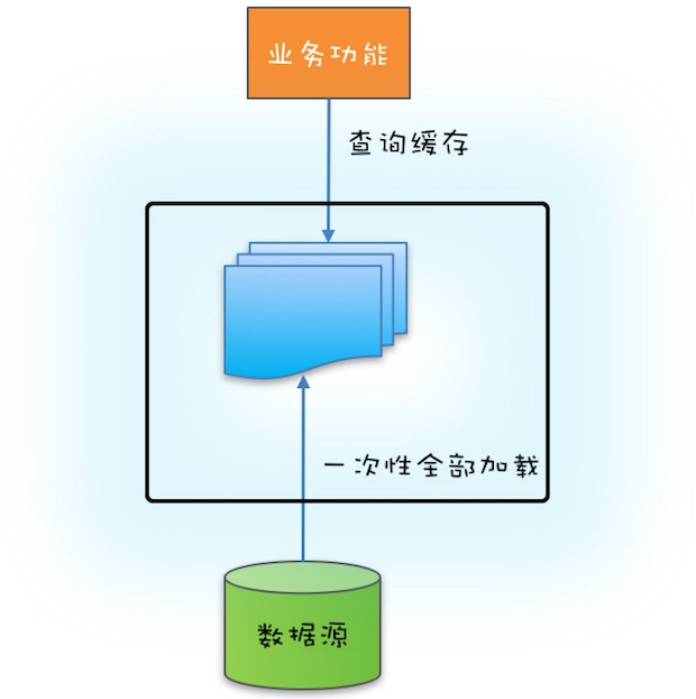
Cache这个工具类，我们提供了两个方法，一个是读缓存方法get()，另一个是写缓存方法put()。读缓存需要用到读锁，读锁的使用和前面我们介绍的Lock的使用是相同的，都是try{}finally{}这个编程范式。写缓存则需要用到写锁，写锁的使用和读锁是类似的。这样看来，读写锁的使用还是非常简单的。

```
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(); }
  }
}
```

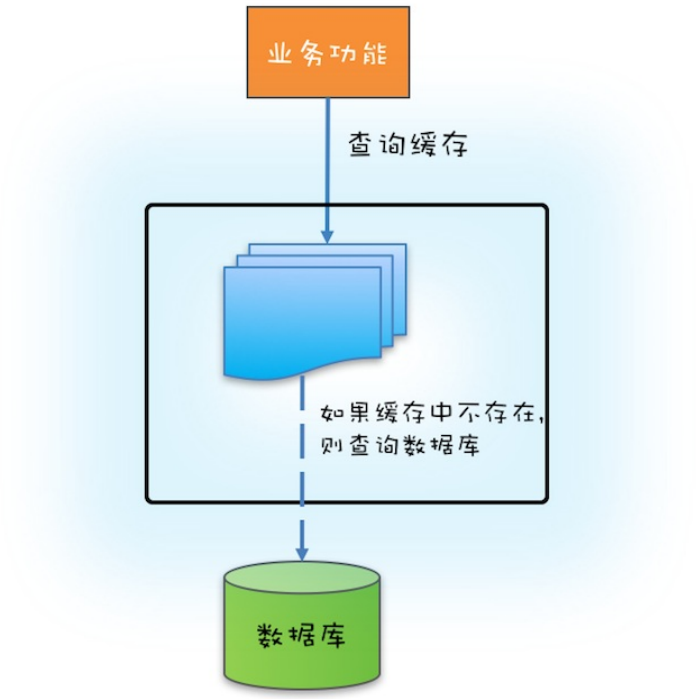
如果你曾经使用过缓存的话，你应该知道**使用缓存首先要解决缓存数据的初始化问题**。缓存数据的初始化，可以采用一次性加载的方式，也可以使用按需加载的方式。

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
        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。

猜你喜欢

精选留言：

- 密码123456 2019-04-06 10:06:44
  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。 [33赞]

- crazypokerk 2019-04-06 09:44:34
  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。 [7赞]

  作者回复2019-04-06 10:09:50
  可以这样理解，不过释放了读锁，也就谈不上升级了

- linqw 2019-04-07 23:40:00
  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 [4赞]

- 缪文@有赞 2019-04-07 21:13:18

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 [3赞]

作者回复2019-04-08 14:05:40
可以这样

- WL 2019-04-09 20:25:36
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? [1赞]

作者回复2019-04-09 22:24:33
分布式有分布式的锁，单机的效率就是靠多线程了

- iron_man 2019-04-06 18:27:08
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 [1赞]

- Dylan 2019-04-06 17:20:53
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？ [1赞]

作者回复2019-04-06 21:05:06
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

- 老杨同志 2019-04-06 13:46:10
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？ [1赞]

作者回复2019-04-06 16:01:57
不会一直拿不到，只是等待的时间会很长

- zhangtnty 2019-04-06 10:14:57
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。 [1赞]

- Lemon 2019-04-06 07:11:48
看线程的堆栈 [1赞]

- 西西弗与卡夫卡 2019-04-06 01:04:03
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 [1赞]

- JGOS 2019-04-15 23:17:28
所谓升级降级，本质上是一把锁的排它性的变化，此处的读写锁其实是一把锁的不同视图。

- Zach_ 2019-04-12 07:56:31
  老师，我想问一个“超纲”的问题:

  我看分布式锁使用redis实现，主要就是在while循环里使用jedis的get(key); setnx(key,value);

  那这里分布式锁的实现，有没有必要采用今天讲的读写所+双检查来实现啊？

  恳请老师回答，谢谢老师!

  作者回复2019-04-14 00:40:26
  我觉得没必要用双重检查。分布式下面的锁和进程内部的锁区别很大，要考虑容错，redis的锁都有时效，假设是5分钟，获取redis锁之后，sleep（6分钟）后再执行业务代码，那么理论上起不到互斥的作用，这种情况不能说没有（有些进程就是突然假死，一会又活过来了），但是一般都是忽略了。

- Sunny_Lu 2019-04-11 09:30:13
  老师，读锁释放，获取写锁的时候，会存在并发，获取不到写锁阻塞的情况吧？

  作者回复2019-04-13 09:19:17
  存在

- Geek_c33c8e 2019-04-10 21:26:41
  老师，看到你说写锁可以降级为读锁，是指在同一线程吗？如果是两个线程的话，读写就是互斥的了吧？

  作者回复2019-04-10 21:45:18
  是的

- Geek_c33c8e 2019-04-10 14:20:47
  老师还是有点不明白这个读写锁，譬如在get方法的是，r.lock和r.unlock没有什么意义啊，因为r锁是支持并发，在get方法获取不到的时候，才加write锁，那这样岂不是ReentrantLock应该也可以满足，不同的是，ReentrantReadWriteLock写的时候，可以控制其他线程不同读，ReentrantLock不能做到这点而已。这对于多线程的影响会大吗？求解答下

  作者回复2019-04-10 20:19:47
  互斥锁能满足，只是慢

- 空知 2019-04-09 14:32:12
  锁降级为读锁可以直接获取当前锁,避免可能发生其他线程的写锁获取写操作

- 空知 2019-04-09 14:30:48
  写锁没结束,锁降级为读锁,读锁可以操作共享变量 这还是同一个线程操作 所以跟原则第三条互斥 不矛盾~~~

- 悟空 2019-04-09 13:20:55
  我会dump出线程信息，查看有没有线程在等待获取写锁

- 胡桥 2019-04-09 11:41:57
  课后思考：因为读锁与读锁直接不会互斥，所以可以看看系统中仅有“读操作”的模块是否还能响应。