**17、ReadWriteLock: 如何快速实现一个完备的缓存？**[](" \l "17readwritelock" \o "Permanent link)

* 什么是读写锁？
* 快速实现一个缓存
* 实现缓存按需加载
* 读写锁的升级与降级
* 总结

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。那么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。**

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，为了我优化性能，我们会经常使用到缓存，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的应用场景。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，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。

**什么是读写锁？**[](" \l "_1" \o "Permanent link)

**读写锁必须遵循以下三条基本原则：**[](" \l "_2" \o "Permanent link)

* 允许多个线程同时读共享变量；
* 只允许一个线程写共享变量；
* 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，这时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**[](" \l "_3" \o "Permanent link)

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 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** ReentrantReadWritLock**();**

*//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**();**

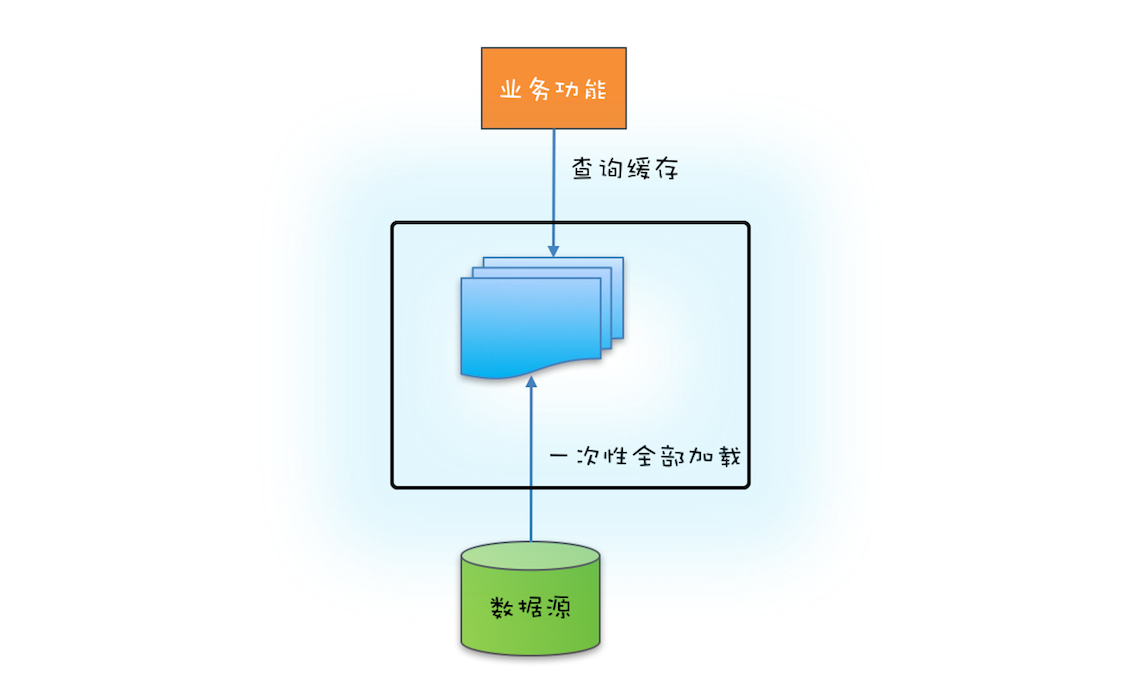
**}**

**}**

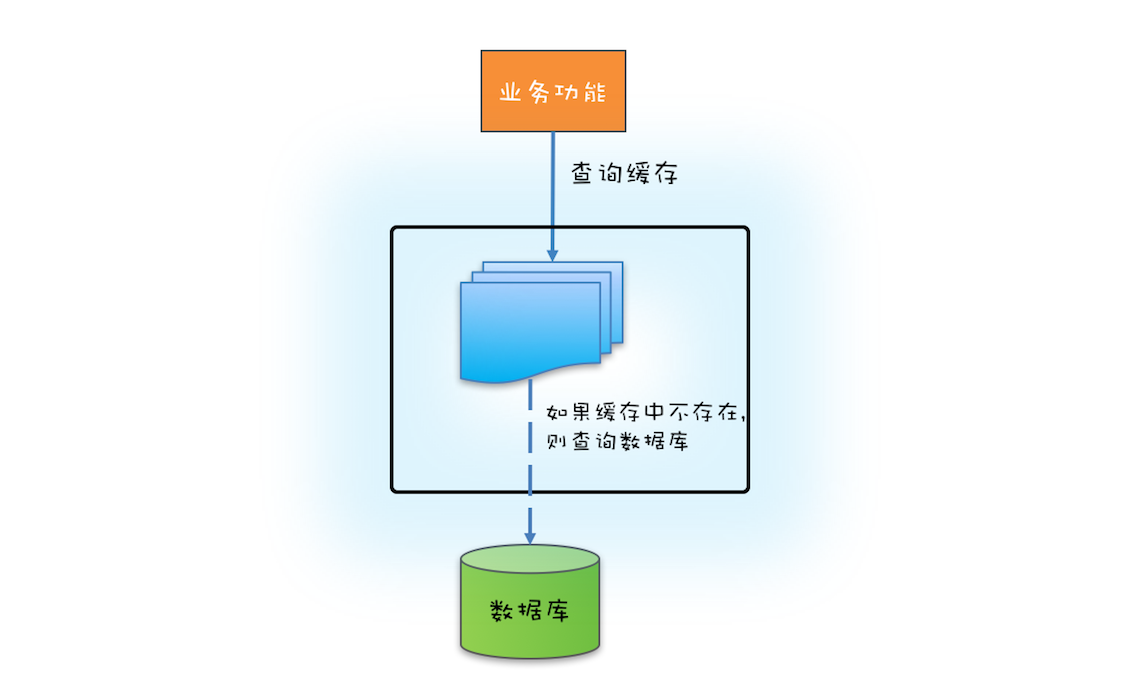
**}**

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。



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。



**实现缓存的按需加载**[](" \l "_4" \o "Permanent link)

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，有可能会有很多线程竞争写锁。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。那么此时他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**[](" \l "_5" \o "Permanent link)

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问题的，先是获取读锁，然后再升级为写锁，对此还有个专业名字，叫 **锁的升级**。可惜ReadWriteLock并不支持这种升级。在上面的代码实例中，读锁还没有释放，此时获取写锁，对导致写锁永久等待，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。

虽然锁的升级是不允许的，但是锁的降级却是允许的，以下代码来源于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**[](" \l "_6" \o "Permanent link)

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。