记一次死锁问题的排查和解决

缘由

说起来这个事情还是挺悲催的，记得上周忙的不亦乐乎，目标是修改之前另外一个团队留下来的一坨代码中的一些bug，这个项目是做OLAP分析的，分为两个模块，逻辑服务器主要负责一些元数据的操作，例如页面上展示的一些信息，而分析服务器负责执行查询语句，因为他们之前使用的是mondrian作为OLAP分析引擎，所以输入的查询是MDX语句，然后结果是一个二维的数据。这是基本的项目背景，当然使用mondrian的过程中发现他的确够慢的。

而且mondrian还有一个问题，它的确在内部实现了一些缓存，缓存好像是基于cell的，但是它的缓存全部是保存在进程内部的，这就导致每一个分析服务器是有状态的，不能扩展成多个，否则就不能利用这些缓存了，另外，因为我们需要支持大量的数据源（每一个产品可能有一个或者多个数据源），每一个数据源可能定义多个报表，每一个报表对应着一个MDX查询语句，这就导致缓存的数据很大，很容易就造成OOM的现象，因此我们接下来的任务就是把这个缓存移出去，放到第三方的缓存系统中。

回到正题，正当忙完准备周五上线呢，上线之后没怎么验证就匆匆在用户群里面吼了一声，因此大家都打开点啊点，突然老大过来说怎么现在打开报表什么的这么慢啊，我查了一下发现的确挺慢的，为什么在测试环境中没有发现呢？多次验证之后开始怀疑自己可能真的改错了什么了，立马回滚到之前的版本，然后就剩下我一头汗水中排查到底出现了什么问题。

好在将线上的环境切到测试环境中很容易就把这个现象给复现了，主要是点开某个报表，然后经过一段时间的加载，接下来点开该报表之后就会快很多，因为接下来的操作都是从缓存中获取的，但是当我在页面上点击“清除缓存”之后（这个操作其实时清除整个报表的缓存和mondrian内部的缓存），发现会等待很长的时间才能返回，然后这个操作是异步的，在页面上我还能进行其他操作。但是当我再次点击其它报表的“清除缓存“的操作就会出现卡顿，然后发现打开其他的报表可能要等待一段时间，问题就这么很容易的复现了。

排查

之前没有针对java这方面的排查经验，但是也知道jstack，jmap之类的工具，于是立即用jstack把整个进程的堆栈抓取下来（很是后悔没有在回滚之前执行jstack），发现的确出现了问题：

```
Found one Java-level deadlock:
=============================
"mondrian.rolap.RolapResultShepherd$executor_160":
  waiting to lock monitor 0x0000000043b2bf70 (object 0x0000000702080db0, a mondrian.rolap.MemberCacheHelper),
  which is held by "mondrian.rolap.RolapResultShepherd$executor_152"
"mondrian.rolap.RolapResultShepherd$executor_152":
  waiting to lock monitor 0x00007f4e6c0751c8 (object 0x0000000702081270, a mondrian.rolap.MemberCacheHelper),
  which is held by "http-8182-11"
"http-8182-11":
  waiting to lock monitor 0x0000000043b2bf70 (object 0x0000000702080db0, a mondrian.rolap.MemberCacheHelper),
  which is held by "mondrian.rolap.RolapResultShepherd$executor_152"
```

这意味着程序里面出现了死锁，这里牵扯到了三个线程，但是其中的两个线程都持有了一个锁并且希望锁住对方持有的锁，而第三个线程正在等待前两个线程中某个线程已经持有的锁，有了这个堆栈就很容易排查问题了，并且在堆栈信息中发现很多线程都在等待这两个线程中已经持有的锁，但是因为这两个线程已经处于死锁状态了，其他的线程只能同步的等待，这样继续在前端操作这些报表迟早把tomcat中的线程消耗完。

根据堆栈找到对应的代码，代码执行的是清理缓存的操作，但是缓存是对于每一个cube下的hierarchy创建的，因此根据具体的堆栈中的调用信息如下：

```
    at mondrian.rolap.SmartMemberReader.flushCacheSimple(SmartMemberReader.java:577)
    - waiting to lock <0x00000007020a8990> (a mondrian.rolap.MemberCacheHelper)
    at mondrian.rolap.RolapCubeHierarchy$CacheRolapCubeHierarchyMemberReader.flushCacheSimple(RolapCubeHierarchy.java:883)
    at mondrian.rolap.RolapCubeHierarchy.flushCacheSimple(RolapCubeHierarchy.java:458)
    at mondrian.rolap.MemberCacheHelper.flushCache(MemberCacheHelper.java:166)
    - locked <0x00000007020a8e50> (a mondrian.rolap.MemberCacheHelper)
    at mondrian.rolap.RolapCubeHierarchy$CacheRolapCubeHierarchyMemberReader.flushCache(RolapCubeHierarchy.java:878)
    at mondrian.rolap.RolapCubeHierarchy.flushCache(RolapCubeHierarchy.java:451)
```

最先进入的这个flushCache函数是hierarchy级别的缓存清理，它其实是调用它的成员变量reader对象的clearCache方法，这个reader用于读取这个hierarchy下的members，可以直接从数据源（关系数据库）中读取，也维护了members的缓存，因此调用reader的clearCache方法也就是调用它的cache对象的方法，这个cache对象名为rolapCubeCacheHelper，类型为MemberCacheHelper，但是发现在reader中的clearCache方法执行的具体操作如下：

```
     @Override
    public void flushCache(){
        super.flushCache();
        rolapCubeCacheHelper.flushCache();
    }
```

首先调用父类的flushCache方法，父类又是什么鬼，打开父类的flushCache方法发现更奇怪的事情：

```
public void flushCache(){
  synchronized( cacheHelper){
        cacheHelper .flushCache();
 }
}
```

这是父类的flushCache方法，它其实就是对成员变量的cacheHelper对象加锁，然后使用cacheHelper的flushCache方法，打开cacheHelper对象才发现它又是一个MemberCacheHelper对象，这时候问题来了，为什么父类和子类都保存了一个MemberCacheHelper对象呢？其实MemberCacheHelper这个对象就是一个缓存的结构体，父类有一些公有的缓存数据，子类有自己的缓存信息，这样也能说得过去，继续到MemberCacheHelper类的flushCache方法：

```
 // Must sync here because we want the three maps to be modified together.
public synchronized void flushCache() {
    mapMemberToChildren.clear();
    mapKeyToMember.clear();
    mapLevelToMembers.clear();

    if (rolapHierarchy instanceof RolapCubeHierarchy){
         ((RolapCubeHierarchy)rolapHierarchy ).flushCacheSimple();
    }
    // We also need to clear the approxRowCount of each level.
    for (Level level : rolapHierarchy.getLevels()) {
        ((RolapLevel)level ).setApproxRowCount(Integer. MIN_VALUE);
    }
}
```

这里对缓存中的每一个map进行clear，然后又对这个hierarchy执行flushCacheSimple方法，我勒个擦，怎么又回来了，这个hierarchy对象不就是我们进出进入flushCache的那个hierarchy对象吗？过了一遍flushCacheSimple方法发现它最终又调用了reader的flushCacheSimple方法，这个函数执行的操作类似于flushCache：

```
 public void flushCacheSimple(){
       super.flushCacheSimple();
       rolapCubeCacheHelper.flushCacheSimple();
    }
```

好了，继续到MemberCacheHelper的flushCacheSimple方法：

```
public void flushCacheSimple(){
 synchronized(cacheHelper){
       cacheHelper.flushCacheSimple();
 }
}
```

我勒个擦，这里又加锁，之前不是已经加过了吗？当然这个锁因该是可重入的，这里自然不会造成死锁，但是下面的rolapCubeCacheHelper对象也是MemberCacheHelper对象啊！最后进入flushCacheSimple方法，这彻底凌乱了：

```
public synchronized void flushCacheSimple() {
    mapMemberToChildren.clear();
    mapKeyToMember.clear();
    mapLevelToMembers.clear();
    // We also need to clear the approxRowCount of each level.
    for (Level level : rolapHierarchy.getLevels()) {
        ((RolapLevel)level).setApproxRowCount(Integer.MIN_VALUE);
    }
```

```
}
```

这里面执行的操作和flushCache方法不是一样的吗？！这到底是在做什么，当然理了这么多也发现了出现死锁的根源了，就在于reader执行的flushCache方法，这里面分别调用了父类和当前类的cacheHelper对象的flushCache，但是这个方法还会调用flushCacheSimple方法，这个方法再次调用reader的flushCacheSimple方法，这里再次调用父类和当前类的cacheHelper对象的flushCacheSimple方法，而且每次调用都需要加锁，这就导致了如下的死锁情况：

A线程执行flushCache方法，它已经完成了super.flushCache方法，然后执行当前reader对象的flushCache方法，首先及时需要持有这个helper对象的锁，然后再执行到flushCacheSimple的时候申请父类的helper对象的锁。B线程可能在执行super.flushCache进入这个函数意味着需要持有父类的helper，但是当它执行flushCacheSimple的时候有需要申请当前类的helper对象的锁，于是就造成了死锁。

开始没有定位到这个问题之前不晓得死锁到底是怎么回事造成的，于是想着让所有的线程顺序执行flushCache方法就可以避免死锁了（不要并发了），但是尝试了一下发现不能这样，因为其他线程还是有可能调用这个flushCache方法，这个不是由我控制的，于是只能具体了解这个函数到底执行了什么，发现flushCache和flushCacheSimple方法其实是重复的，不晓得当初写这段代码的人是怎么想的，于是就把所有的flushCacheSimple方法的调用去掉，这样就不会再有持有A锁再去申请B锁的情况了。

总结

问题算是解决了，最终hotfix版本也算是上线了，一颗悬着的心也算放下了，着这个过程中我也学到了不少知识：

- 学会并且善于使用java提供的分析工具，例如jstack、jstat、jmap、以及开源的MAT等等。
- 遇到问题不要害怕，不要一味的埋怨这个问题不是我造成的，我也不知道怎么回事之类的，静下心来思考整个流程，运用以前的理论知识和经验一定能够把问题解决的，没有什么问题是偶然的，如果出错一定是代码有问题。
- 测试很重要，尤其压力测试，我们项目目前人手紧缺，QA也没有专职的，所以基本上是开发在开发环境上测试一下功能，并没有做过性能测试之类的东西，我觉得测试应该尽可能覆盖线上可能出现的各种情况。
- 上线之前做好回滚，否则你会很狼狈，幸亏这点我每次操作之前都先备份。
- 在编码的时候，尤其一个操作会涉及到多个synchronized操作的时候尤其要注意，回忆一下当初避免死锁的几个方法，按顺序加锁往往是最好的解决办法。
- 搞清楚一个方法到底想要做什么？输入是什么，输出是什么，会造成什么影响，在写完一个方法之后在脑子中模拟一下整个函数的执行流程是否符合预想。
- 如果真的遇到这样的需求：父类和子类都持有一个类型的对象，让他们独立操作，父类对象的操作完成之后在执行子类对象的操作，而不要穿插着调用。

接下来的一段时间要开始搞mondrian了，希望能够从这个OLAP执行引擎中学到一些东西，不过自己的编译原理方面的知识几乎为0，这方面需要补强啊，我对于mondrian中重点要看的东西应该是：* 如何解析MDX（类似于如何解析SQL）* 如何将MDX动态的翻译成一串SQL（类似于如何生成执行计划）* 缓存如何实现 * 执行MDX或者SQL时如何使用缓存 * 如果使用聚合表进行优化。

希望顺利~