二

17 案例分析：分库分表后，我的应用崩溃了

本课时我们主要分析一个案例，那就是分库分表后，我的应用崩溃了。

前面介绍了一种由于数据库查询语句拼接问题，而引起的一类内存溢出。下面将详细介绍一下这个过程。

假设我们有一个用户表，想要通过用户名来查询某个用户，一句简单的 SQL 语句即可：

```
select * from user where fullname = "xxx" and other="other";
```

为了达到动态拼接的效果，这句 SQL 语句被一位同事进行了如下修改。他的本意是，当 fullname 或者 other 传入为空的时候，动态去掉这些查询条件。这种写法，在 MyBaits 的配置文件中，也非常常见。

```
List<User> query(String fullname, String other) {
        StringBuilder sb = new StringBuilder("select * from user where 1=1 ");
        if (!StringUtils.isEmpty(fullname)) {
            sb.append(" and fullname=");
            sb.append(" \"" + fullname + "\"");
        }
        if (!StringUtils.isEmpty(other)) {
            sb.append(" and other=");
            sb.append(" \"" + other + "\"");
        }
        String sql = sb.toString();
        ...
 }
```

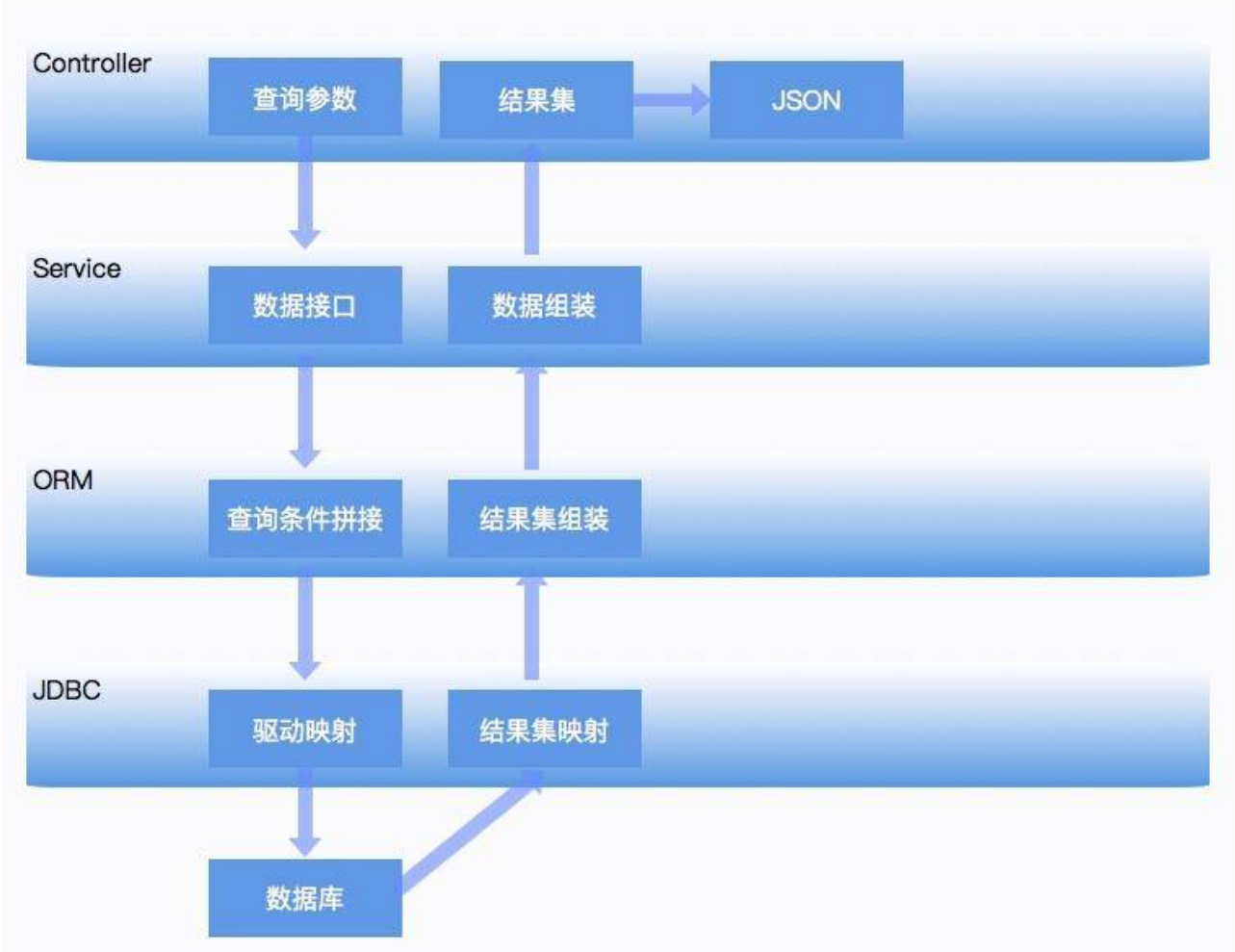
大多数情况下，这种写法是没有问题的，因为结果集合是可以控制的。但随着系统的运行，用户表的记录越来越多，当传入的 fullname 和 other 全部为空时，悲剧的事情发生了，SQL 被拼接成了如下的语句：

```
select * from user where 1=1
```

数据库中的所有记录，都会被查询出来，载入到 JVM 的内存中。由于数据库记录实在太多，直接把内存给撑爆了。

在工作中，由于这种原因引起的内存溢出，发生的频率非常高。通常的解决方式是强行加入**分页功能**，或者对一些必填的参数进行校验，但不总是有效。因为上面的示例仅展示了一个非常简单的 SQL 语句，而在实际工作中，这个 SQL 语句会非常长，每个条件对结果集的影响也会非常大，在进行数据筛选的时候，一定要小心。

内存使用问题



拿一个最简单的 Spring Boot 应用来说，请求会通过 Controller 层来接收数据，然后 Service 层会进行一些逻辑的封装，数据通过 Dao 层的 ORM 比如 JPA 或者 MyBatis 等，来调用底层的 JDBC 接口进行实际的数据获取。通常情况下，JVM 对这种数据获取方式，表现都是非常温和的。我们挨个看一下每一层可能出现的一些不正常的内存使用问题（仅限 JVM 相关问题），以便对平常工作中的性能分析和性能优化有一个整体的思路。

首先，我们提到一种可能，那就是类似于 Fastjson 工具所产生的 bug，这类问题只能通过升级依赖的包来解决，属于一种极端案例。具体可参考这里

Controller 层

Controller 层用于接收前端查询参数，然后构造查询结果。现在很多项目都采用**前后端分离**架构，所以 Controller 层的方法，一般使用 @ResponseBody 注解，把查询的结果，解析成 JSON 数据返回。

这在数据集非常大的情况下，会占用很多内存资源。假如结果集在解析成 JSON 之前，占用的内存是 10MB，那么在解析过程中，有可能会使用 20M 或者更多的内存去做这个工作。如果结果集有非常深的嵌套层次，或者引用了另外一个占用内存很大，且对于本次请求无意义的对象（比如非常大的 byte[] 对象），那这些序列化工具会让问题变得更加严重。

因此，对于一般的服务，保持结果集的精简，是非常有必要的，这也是 DTO（Data Transfer Object）存在的必要。如果你的项目，返回的结果结构比较复杂，对结果集进行一次转换是非常有必要的。互联网环境不怕小结果集的高并发请求，却非常恐惧大结果集的耗时请求，这是其中一方面的原因。

Service 层

Service 层用于处理具体的业务，更加贴合业务的功能需求。一个 Service，可能会被多个 Controller 层所使用，也可能会使用多个 dao 结构的查询结果进行计算、拼装。

Service 的问题主要是对底层资源的不合理使用。举个例子，有一回在一次代码 review 中，发现了下面让人无语的逻辑：

```
//错误代码示例
int getUserSize() {
        List<User> users = dao.getAllUser();
        return null == users ? 0 : users.size();
}
```

这种代码，其实在一些现存的项目里大量存在，只不过由于项目规模和工期的原因，被隐藏了起来，成为内存问题的定时炸弹。

Service 层的另外一个问题就是，职责不清、代码混乱，以至于在发生故障的时候，让人无从下手。这种情况就更加常见了，比如使用了 Map 作为函数的入参，或者把多个接口的请求返回放在一个 Java 类中。

```
//错误代码示例
Object exec(Map<String,Object> params){
        String q = getString(params,"q");
        if(q.equals("insertToa")){
            String q1 = getString(params,"q1");
            String q2 = getString(params,"q2");
            //do A
        }else if(q.equals("getResources")){
```

```
            String q3 = getString(params,"q3");
            //do B
        }
        ...
        return null;
}
```

这种代码使用了万能参数和万能返回值，exec 函数会被几十个上百个接口调用，进行逻辑的分发。这种将逻辑揉在一起的代码块，当发生问题时，即使使用了 Jstack，也无法发现具体的调用关系，在平常的开发中，应该严格禁止。

ORM 层

ORM 层可能是发生内存问题最多的地方，除了本课时开始提到的 SQL 拼接问题，大多数是由于对这些 ORM 工具使用不当而引起的。

举个例子，在 JPA 中，如果加了一对多或者多对多的映射关系，而又没有开启懒加载、级联查询的时候就容易造成深层次的检索，内存的开销就超出了我们的期望，造成过度使用。

另外，JPA 可以通过使用缓存来减少 SQL 的查询，它默认开启了一级缓存，也就是 EntityManager 层的缓存（会话或事务缓存），如果你的事务非常的大，它会缓存很多不需要的数据；JPA 还可以通过一定的配置来完成二级缓存，也就是全局缓存，造成更多的内存占用。

一般，项目中用到缓存的地方，要特别小心。除了容易造成数据不一致之外，对堆内内存的使用也要格外关注。如果使用量过多，很容易造成频繁 GC，甚至内存溢出。

JPA 比起 MyBatis 等 ORM 拥有更多的特性，看起来容易使用，但精通门槛却比较高。

这并不代表 MyBatis 就没有内存问题，在这些 ORM 框架之中，存在着非常多的类型转换、数据拷贝。

举个例子，有一个批量导入服务，在 MyBatis 执行批量插入的时候，竟然产生了内存溢出，按道理这种插入操作是不会引起额外内存占用的，最后通过源码追踪到了问题。

这是因为 MyBatis 循环处理 batch 的时候，操作对象是数组，而我们在接口定义的时候，使用的是 List；当传入一个非常大的 List 时，它需要调用 List 的 toArray 方法将列表转换成数组（浅拷贝）；在最后的拼装阶段，使用了 StringBuilder 来拼接最终的 SQL，所以实际使用的内存要比 List 多很多。

事实证明，不论是插入操作还是查询动作，只要涉及的数据集非常大，就容易出现问题。由于项目中众多框架的引入，想要分析这些具体的内存占用，就变得非常困难。保持小批量操

作和结果集的干净，是一个非常好的习惯。

分库分表内存溢出

分库分表组件

如果数据库的记录非常多，达到千万或者亿级别，对于一个传统的 RDBMS 来说，最通用的解决方式就是分库分表。这也是海量数据的互联网公司必须面临的一个问题。

根据切入的层次，数据库中间件一般分为编码层、框架层、驱动层、代理层、实现层 5 大类。典型的框架有驱动层的 sharding-jdbc 和代理层的 MyCat。

MyCat 是一个独立部署的 Java 服务，它模拟了一个 MySQL 进行请求的处理，对于应用来说使用是透明的。而 sharding-jdbc 实际上是一个数据库驱动，或者说是一个 DataSource，它是作为 jar 包直接嵌入在客户端应用的，所以它的行为会直接影响到主应用。

这里所要说的分库分表组件，就是 sharding-jdbc。不管是普通 Spring 环境，还是 Spring Boot 环境，经过一系列配置之后，我们都可以像下面这种方式来使用 sharding-jdbc，应用层并不知晓底层实现的细节：

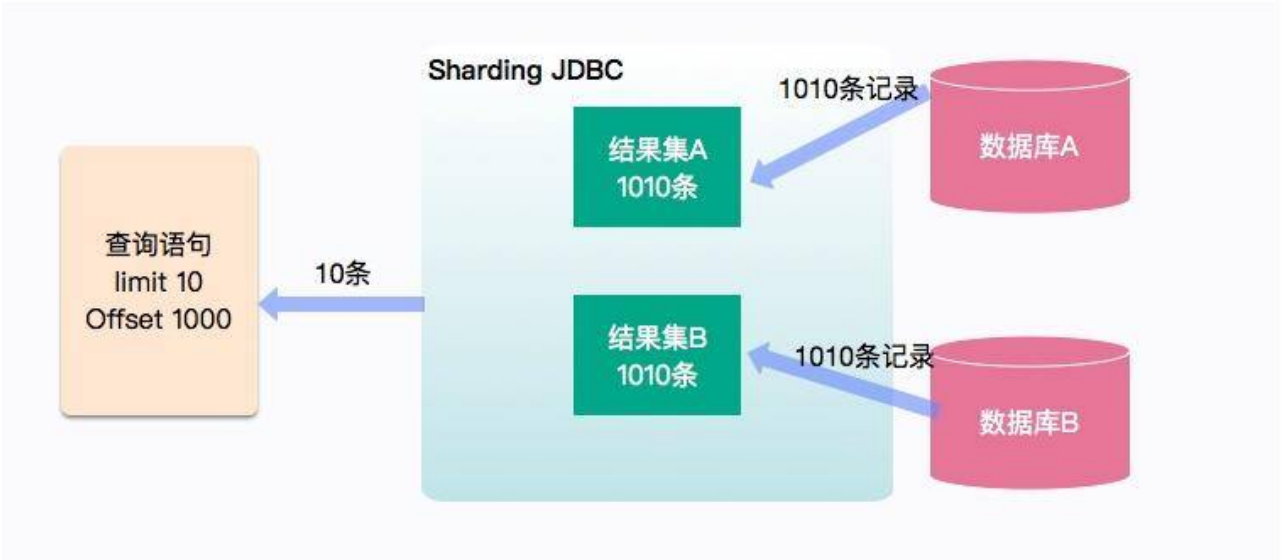
```
@Autowired
private DataSource dataSource;
```

我们有一个线上订单应用，由于数据量过多的原因，进行了分库分表。但是在某些条件下，却经常发生内存溢出。

分库分表的内存溢出

一个最典型的内存溢出场景，就是在订单查询中使用了深分页，并且在查询的时候没有使用“切分键”。使用前面介绍的一些工具，比如 MAT、Jstack，最终追踪到是由于 sharding-jdbc 内部实现所引起的。

这个过程也是比较好理解的，如图所示，订单数据被存放在两个库中。如果没有提供切分键，查询语句就会被分发到所有的数据库中，这里的查询语句是 limit 10、offset 1000，最终结果只需要返回 10 条记录，但是数据库中间件要完成这种计算，则需要 (1000+10)*2=2020 条记录来完成这个计算过程。如果 offset 的值过大，使用的内存就会暴涨。虽然 sharding-jdbc 使用归并算法进行了一些优化，但在实际场景中，深分页仍然引起了内存和性能问题。



下面这一句简单的 SQL 语句，会产生严重的后果：

```
select * from order  order by updateTime desc limit 10 offset 10000
```

这种在中间节点进行归并聚合的操作，在分布式框架中非常常见。比如在 ElasticSearch 中，就存在相似的数据获取逻辑，不加限制的深分页，同样会造成 ES 的内存问题。

另外一种情况，就是我们在进行一些复杂查询的时候，发现分页失效了，每次都是取出全部的数据。最后根据 Jstack，定位到具体的执行逻辑，发现分页被重写了。

```
private void appendLimitRowCount(final SQLBuilder sqlBuilder, final RowCountToken r
```

```
        SelectStatement selectStatement = (SelectStatement) sqlStatement;
        Limit limit = selectStatement.getLimit();
        if (!isRewrite) {
            sqlBuilder.appendLiterals(String.valueOf(rowCountToken.getRowCount()));
        } else if ((!selectStatement.getGroupByItems().isEmpty() || !selectStatemen
            sqlBuilder.appendLiterals(String.valueOf(Integer.MAX_VALUE));
        } else {
            sqlBuilder.appendLiterals(String.valueOf(limit.isNeedRewriteRowCount()
        }
        int beginPosition = rowCountToken.getBeginPosition() + String.valueOf(rowCo
        appendRest(sqlBuilder, count, sqlTokens, beginPosition);
    }
```

如上代码，在进入一些复杂的条件判断时（参照 SQLRewriteEngine.java），分页被重置为 Integer.MAX_VALUE。

总结

本课时以 Spring Boot 项目常见的分层结构，介绍了每一层可能会引起的内存问题，我们把结论归结为一点，那就是**保持输入集或者结果集的简洁**。一次性获取非常多的数据，会让中间过程变得非常不可控。最后，我们分析了一个驱动层的数据库中间件，以及对内存使用的一些问题。

很多程序员把这些耗时又耗内存的操作，写了非常复杂的 SQL 语句，然后扔给最底层的数据库去解决，这种情况大多数认为换汤不换药，不过是把具体的问题冲突，转移到另一个场景而已。

上一页 下一页