如何实现超高并发的无锁缓存？

2017-02-23 58沈剑 中生代技术

作者 | 58沈剑
来源 | 架构师之路

编者按

本文转自架构师之路，作者沈剑详细描述了在超高并发条件下如何实现缓存数据的一致性。作者提供了几种可行的解决方案，并详细描述了这几种解决方案的优缺点，最终指出了如何实现无锁缓存这个终极目标。对于致力于超高并发数据处理的你，本文非常具有借鉴意义。

一、需求缘起

【业务场景】

有一类写多读少的业务场景：大部分请求是对数据进行修改，少部分请求对数据进行读取。

例子1：滴滴打车，某个司机地理位置信息的变化（可能每几秒钟有一个修改），以及司机地理位置的读取（用户打车的时候查看某个司机的地理位置）。

void SetDriverInfo(long driver_id, DriverInfoi); // 大量请求调用修改司机信息，可能主要是GPS位置的修改

DriverInfo GetDriverInfo(long driver_id); // 少量请求查询司机信息

例子2：统计计数的变化，某个url的访问次数，用户某个行为的反作弊计数（计数值在不停的变）以及读取（只有少数时刻会读取这类数据）。

void AddCountByType(long type); // 大量增加某个类型的计数，修改比较频繁

long GetCountByType(long type); // 少量返回某个类型的计数

【底层实现】

具体到底层的实现，往往是一个Map（本质是一个定长key，定长value的缓存结构）来存储司机的信息，或者某个类型的计数。

Map<driver_id, DriverInfo>

Map<type, count>

【临界资源】

这个Map存储了所有信息，当并发读写访问时，它作为临界资源，在读写之前，一般要进行加锁操作，以司机信息存储为例：

```
void SetDriverInfo(long driver_id, DriverInfoinfo){
        WriteLock (m_lock);
        Map<driver_id>= info;
        UnWriteLock(m_lock);
}

DriverInfo GetDriverInfo(long driver_id){
```

```
    DriverInfo t;
    ReadLock(m_lock);
    t= Map<driver_id>;
    UnReadLock(m_lock);
    return t;
}
```

【并发锁瓶颈】

假设滴滴有100w司机同时在线，每个司机没5秒更新一次经纬度状态，那么每秒就有20w次写并发操作。假设滴滴日订单1000w个，平均每秒大概也有300个下单，对应到查询并发量，可能是1000级别的并发读操作。

上述实现方案没有任何问题，但在并发量很大的时候（每秒20w写，1k读），锁m_lock会成为潜在瓶颈，在这类高并发环境下写多读少的业务仓井，如何来进行优化，是本文将要讨论的问题。

二、水平切分+锁粒度优化

上文中之所以锁冲突严重，是因为所有司机都公用一把锁，锁的粒度太粗（可以认为是一个数据库的“库级别锁”），是否可能进行水平拆分（类似于数据库里的分库），把一个库锁变成多个库锁，来提高并发，降低锁冲突呢？显然是可以的，把1个Map水平切分成多个Map即可：

```
void SetDriverInfo(long driver_id, DriverInfoinfo){
    i= driver_id % N; // 水平拆分成N份，N个Map，N个锁
    WriteLock (m_lock [i]);  //锁第i把锁
    Map[i]<driver_id>= info;  // 操作第i个Map
    UnWriteLock (m_lock[i]); // 解锁第i把锁
}
```

每个Map的并发量（变成了1/N）和数据量都降低（变成了1/N）了，所以理论上，锁冲突会成平方指数降低。

分库之后，仍然是库锁，有没有办法变成数据库层面所谓的“行级锁”呢，难道要把x条记录变成x个Map吗，这显然是不现实的。

三、MAP变Array+最细锁粒度优化

假设driver_id是递增生成的，并且缓存的内存比较大，是可以把Map优化成Array，而不是拆分成N个Map，是有可能把锁的粒度细化到最细的（每个记录一个锁）。

```
void SetDriverInfo(long driver_id, DriverInfoinfo){
        index= driver_id;
        WriteLock (m_lock [index]);  //超级大内存，一条记录一个锁，锁行锁
        Array[index]= info; //driver_id就是Array下标
        UnWriteLock (m_lock[index]); // 解锁行锁
}
```

和上一个方案相比，这个方案使得锁冲突降到了最低，但锁资源大增，在数据量非常大的情况下，一般不这么搞。数据量比

较小的时候，可以一个元素一个锁的（典型的是连接池，每个连接有一个锁表示连接是否可用）。

上文中提到的另一个例子，用户操作类型计数，操作类型是有限的，即使一个type一个锁，锁的冲突也可能是很高的，还没有方法进一步提高并发呢？

四、把锁去掉，变成无锁缓存

【无锁的结果】

```
void AddCountByType(long type /*, int count*/){
        //不加锁
        Array[type]++; // 计数++
        //Array[type] += count; // 计数增加count
}
```

如果这个缓存不加锁，当然可以达到最高的并发，但是多线程对缓存中同一块定长数据进行操作时，有可能出现不一致的数据块，这个方案为了提高性能，牺牲了一致性。在读取计数时，获取到了错误的数据，是不能接受的（作为缓存，允许cache miss，却不允许读脏数据）。

【脏数据是如何产生的】

这个并发写的脏数据是如何产生的呢，详见下图：

1）线程1对缓存进行操作，对key想要写入value1

2）线程2对缓存进行操作，对key想要写入value2

3）如果不加锁，线程1和线程2对同一个定长区域进行一个并发的写操作，可能每个线程写成功一半，导致出现脏数据产生，最终的结果即不是value1也不是value2，而是一个乱七八糟的不符合预期的值value-unexpected。

【数据完整性问题】

并发写入的数据分别是value1和value2，读出的数据是value-unexpected，数据的篡改，这本质上是一个数据完整性的问题。通常如何保证数据的完整性呢？

例子1：运维如何保证，从中控机分发到上线机上的二进制没有被篡改？

回答：md5

例子2：即时通讯系统中，如何保证接受方收到的消息，就是发送方发送的消息？

回答：发送方除了发送消息本身，还要发送消息的签名，接收方收到消息后要校验签名，以确保消息是完整的，未被篡改。

当当当当 => “签名”是一种常见的保证数据完整性的常见方案。

【加上签名之后的流程】

加上签名之后，不但缓存要写入定长value本身，还要写入定长签名（例如16bitCRC校验）：

1）线程1对缓存进行操作，对key想要写入value1，写入签名v1-sign

2）线程2对缓存进行操作，对key想要写入value2，写入签名v2-sign

3）如果不加锁，线程1和线程2对同一个定长区域进行一个并发的写操作，可能每个线程写成功一半，导致出现脏数据产生，最终的结果即不是value1也不是value2，而是一个乱七八糟的不符合预期的值value-unexpected，但签名，一定是v1-sign或者v2-sign中的任意一个

4）数据读取的时候，不但要取出value，还要像消息接收方收到消息一样，校验一下签名，如果发现签名不一致，缓存则返回**NULL**，即**cache miss**。

当然，对应到司机地理位置，与URL访问计数的case，除了内存缓存之前，肯定需要timer对缓存中的数据定期落盘，写入数

据库，如果cache miss，可以从数据库中读取数据。

五、总结

在【超高并发】，【写多读少】，【定长value】的【业务缓存】场景下：

1）可以通过水平拆分 来降低锁冲突

2）可以通过**Map**转**Array**的方式来最小化锁冲突，一条记录一个锁

3）可以把锁去掉 ，最大化并发，但带来的数据完整性的破坏

4）可以通过签名的方式保证数据的完整性，实现无锁缓存