不談架構，看看如何從代碼層面優化系統性能！

原創 2016-06-14 程超 聊聊架構

我們以前看到的很多架構變遷或者演進方面的文章大多都是針對架構方面的介紹，很少有針對代碼級別的性能優化介紹，這就好比蓋樓一樣，樓房的基礎架子搭的很好，但是蓋房的工人不夠專業，有很多需要注意的地方忽略了，那麼在往裡面填磚加瓦的時候出了問題，後果就是房子經常漏雨，牆上有裂縫等各種問題出現，雖然不至於樓房塌陷，但樓房也已經變成了危樓。那麼今天我們就將針對一些代碼細節方面的東西進行介紹，歡迎大家吐槽以及提建議。

服務器環境

- 服務器配置：4核CPU，8G內存，共4台
- MQ：RabbitMQ
- 數據庫：DB2
- SOA框架：公司內部封裝的Dubbo
- 緩存框架：Redis、Memcached
- 統一配置管理系統：公司內部開發的系統

問題描述

1. 單台40TPS，加到4台服務器能到60TPS，擴展性幾乎沒有。
2. 在實際生產環境中，經常出現數據庫死鎖導致整個服務中斷不可用。
3. 數據庫事務亂用，導致事務佔用時間太長。
4. 在實際生產環境中，服務器經常出現內存溢出和CPU時間被佔滿。
5. 程序開發的過程中，考慮不全面，容錯很差，經常因為一個小bug而導致服務不可用。
6. 程序中沒有打印關鍵日誌，或者打印了日誌，信息卻是無用信息沒有任何參考價值。
7. 配置信息和變動不大的信息依然會從數據庫中頻繁讀取，導致數據庫IO很大。
8. 項目拆分不徹底，一個Tomcat中會佈署多個項目WAR包。

9. 因為基礎平台的bug，或者功能缺陷導致程序可用性降低。
10. 程序接口中沒有限流策略，導致很多VIP商戶直接拿我們的生產環境進行壓測，直接影響真正的服務可用性。
11. 沒有故障降級策略，項目出了問題後解決的時間較長，或者直接粗暴的回滾項目，但是不一定能解決問題。
12. 沒有合適的監控系統，不能准實時或者提前發現項目瓶頸。

優化解決方案

1、數據庫死鎖優化解決

我們從第二條開始分析，先看一個基本例子展示數據庫死鎖的發生：

时间	会话 A	会话 B
1	BEGIN;	
2	mysql>SELECT * FROM t WHERE a = 1 FOR UPDATE; ********* 1. row ********** a: 1 1 row in set (0.00 sec)	BEGIN
3		mysql>SELECT * FROM t WHERE a = 2 FOR UPDATE; ********* 1. row ********** a: 2 1 row in set (0.00 sec)
4	mysql>SELECT * FROM t WHERE a = 2 FOR UPDATE; # 等待	
5		mysql>SELECT * FROM t WHERE a = 1 FOR UPDATE; ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

註：在上述事例中，會話B會拋出死鎖異常，死鎖的原因就是A和B二個會話互相等待。

分析：出現這種問題就是我們在項目中混雜了大量的事務+for update語句，針對數據庫鎖來說有下面三種基本鎖：

- Record Lock：單個行記錄上的鎖
- Gap Lock：間隙鎖，鎖定一個範圍，但不包含記錄本身
- Next-Key Lock：Gap Lock + Record Lock，鎖定一個範圍，並且鎖定記錄本身

當for update語句和gap lock和next-key lock鎖相混合使用，又沒有注意用法的時候，就

非常容易出現死鎖的情況。

那我們用大量的鎖的目的是什麼，經過業務分析發現，其實就是為了防重，同一時刻有可能會有多筆支付單發到相應系統中，而防重措施是通過在某條記錄上加鎖的方式來進行。

針對以上問題完全沒有必要使用悲觀鎖的方式來進行防重，不僅對數據庫本身造成極大的壓力，同時也會把對於項目擴展性來說也是很大的擴展瓶頸，我們採用了三種方法來解決以上問題：

- 使用Redis來做分佈式鎖，Redis採用多個來進行分片，其中一個Redis掛了也沒關係，重新爭搶就可以了。
- 使用主鍵防重方法，在方法的入口處使用防重表，能夠攔截所有重複的訂單，當重複插入時數據庫會報一個重複錯，程序直接返回。
- 使用版本號的機制來防重。

以上三種方式都必須要有過期時間，當鎖定某一資源超時的時候，能夠釋放資源讓競爭重新開始。

2、數據庫事務佔用時間過長

偽代碼示例：

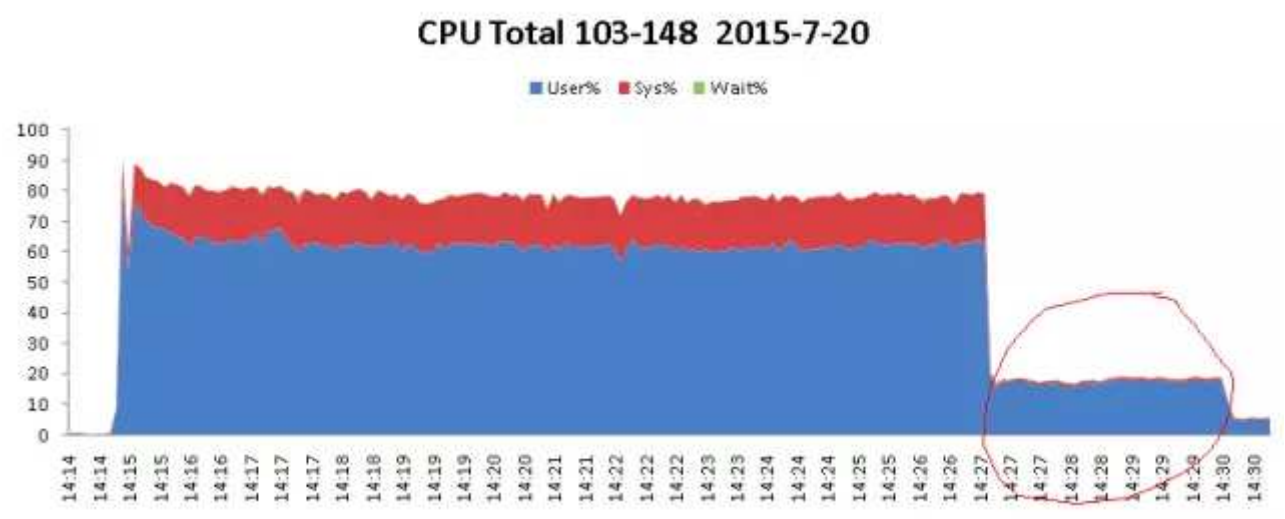
```
public void test() {
    Transaction.begin  //事务开启
    try {
        dao.insert //插入一行记录
        httpClient.queryRemoteResult()  //请求访问
        dao.update //更新一行记录
        Transaction.commit()  //事务提交
    } catch(Exception e) {
        Transaction.rollFor //事务回滚
    }
}
```

項目中類似這樣的程序有很多，經常把類似httpClient，或者有可能會造成長時間超時的操作混在事務代碼中，不僅會造成事務執行時間超長，而且也會嚴重降低並發能力。

那麼我們在用事務的時候，**遵循的原則是快進快出，事務代碼要儘量小**。針對以上偽代碼，我們要用httpClient這一行拆分出來，避免同事務性的代碼混在一起，這不是一個好習慣。

3、CPU時間被佔滿分析

下面以我之前分析的一個案例作為問題的起始點，首先看下面的圖：



項目在壓測的過程中，CPU一直居高不下，那麼通過分析得出如下分析：

1）數據庫連接池影響

我們針對線上的環境進行模擬，儘量真實的在測試環境中再現，採用數據庫連接池為咱們默認的C3P0。

那麼當壓測到二萬批，100個用戶同時訪問的時候，並發量突然降為零！報錯如下：

> com.yeepay.g3.utils.common.exception.YeepayRuntimeException: Could not get JDBC Connection; nested exception is java.sql.SQLException: An attempt by a client to checkout a Connection has timed out.

那麼針對以上錯誤跟蹤C3P0源碼，以及在網上搜索資料發現C3P0在大並發下表現的性能不佳。

2）線程池使用不當引起

```
private static final ExecutorService executorService = Executors.newCachedThreadPool();
/**
* 异步执行短耗时的任务
* @param task
*/
public static void asynShortTask(Runnable task){
 executorService.submit(task);
 //task.run();
}

        CommonUtils.asynShortTask(new Runnable() {
            @Override
            public void run() {
                String sms = sr.getSmsContent();
                sms = sms.replaceAll(finalCode, AES.encryptToBase64(finalCode, ConstantUtils.getDB_AES_KEY()));
                sr.setSmsContent(sms);
                smsManageService.addSmsRecord(sr);
            }
        });
```

以上代碼的場景是每一次並發請求過來，都會創建一個線程，將DUMP日誌導出進行分析發現，項目中啟動了一萬多個線程，而且每個線程都極為忙碌，徹底將資源耗盡。

那麼問題到底在哪裡呢？？？就在這一行！

> private static final ExecutorService executorService = Executors.newCachedThreadPool();

在並發的情況下，無限制的申請線程資源造成性能嚴重下降，在圖表中顯拋物線形狀的元兇就是它！！！那麼採用這種方式最大可以產生多少個線程呢？？答案是：Integer的最大值！看如下源碼：

```
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>(),
                                  threadFactory);
}
```

那麼嘗試修改成如下代碼：

> private static final ExecutorService executorService = Executors.newFixedThreadPool(50);

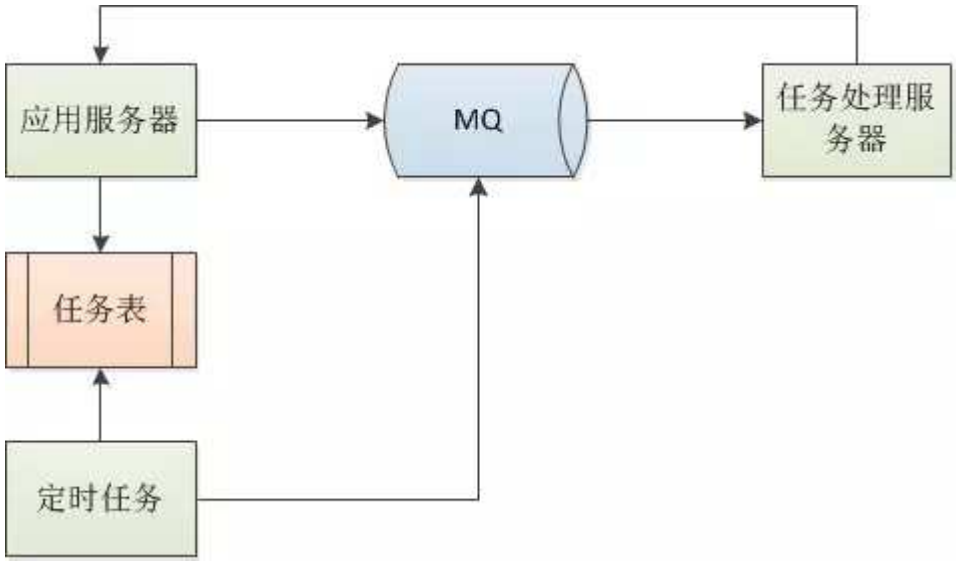
修改完成以後，並發量重新上升到100以上TPS，但是當並發量非常大的時候，項目GC（垃圾回收能力下降），分析原因還是因為Executors.newFixedThreadPool(50)這一行，雖然解決了產生無限線程的問題，但是當並發量非常大的時候，採用newFixedThreadPool這種方式，會造成大量對象堆積到隊列中無法及時消費，看源碼如下：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

可以看到採用的是無界隊列，也就是說隊列是可以無限的存放可執行的線程，造成大量對象無法釋放和回收。

2）最終線程池技術方案

方案一



註：因為服務器的CPU只有4核，有的服務器甚至只有2核，所以在應用程序中大量使用線程的話，反而會造成性能影響，針對這樣的問題，我們將所有異步任務全部拆出應用項目，以任務的方式發送到專門的任務處理器處理，處理完成回調應用程序器。後端定時任務會定時

掃瞄任務表，定時將超時未處理的異步任務再次發送到任務處理器進行處理。

方案二

使用AKKA技術框架，下面是我以前寫的一個簡單的壓測情況：

http://www.jianshu.com/p/6d62256e3327

4、日誌打印問題

先看下面這段日誌打印程序：

```
QuataDTO quataDTO = null;
        try {
            quataDTO = getRiskLimit(payRequest.getQueryRiskInfo(),
 payRequest.getMerchantNo(), payRequest.getIndustryCatalog(),
cardBinResDTO.getCardType(), cardBinResDTO.getBankCode(), bizName);
        } catch (Exception e) {
            logger.info("获取风控限额异常", e);
        }
```

像這樣的代碼是嚴格不符合規範的，雖然每個公司都有自己的打印要求。

首先日誌的打印必須是以logger.error或者logger.warn的方式打印出來。

日誌打印格式：[系統來源] 錯誤描述 [關鍵信息]，日誌信息要能打印出能看懂的信息，有前因和後果。甚至有些方法的入參和出參也要考慮打印出來。

在輸入錯誤信息的時候，Exception不要以e.getMessage的方式打印出來。

合理的日誌格式是：

```
logger.warn("[innersys] - [" + exceptionType.description + "] - [" + methodName + "] - "
                + "errorCode:[" + errorCode + "], "
                + "errorMsg:[" + errorMsg + "]", e);

logger.info("[innersys] - [入參] - [" + methodName + "] - "
                    + LogInfoEncryptUtil.getLogString(arguments) + "]");

logger.info("[innersys] - [返回结果] - [" + methodName + "] - "
 + LogInfoEncryptUtil.getLogString(result));
```

我們在程序中大量的打印日誌，雖然能夠打印很多有用信息幫助我們排查問題，但是更多是日誌量太多不僅影響磁盤IO，更多會造成線程阻塞對程序的性能造成較大影響。

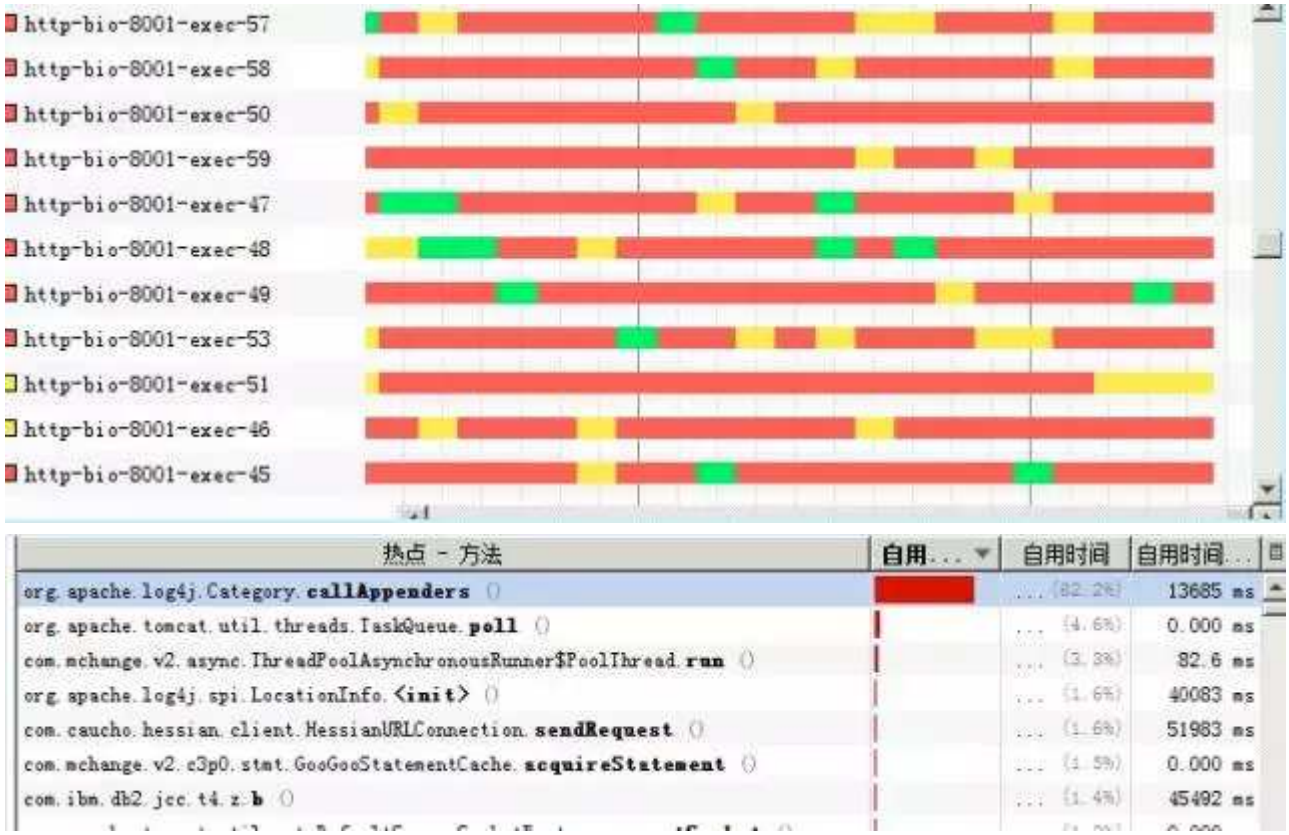
在使用Log4j1.2.14版本的時候，使用如下格式：

> %d %-5p %c:%L [%t] - %m%n

那麼在壓測的時候會出現下面大量的線程阻塞，如下圖：

```
"pool-nocard-1-thread-107" prio=10 tid=0x00007fbeb415c800 nid=0x538a in Object.wait() [0x00007fbe71bd9000]
   java.lang.Thread.State: BLOCKED (on object monitor)
at java.lang.Object.wait(Native Method)
at com.mchange.v2.resourcepool.BasicResourcePool.awaitAvailable(BasicResourcePool.java:1315)
at com.mchange.v2.resourcepool.BasicResourcePool.prelimCheckoutResource(BasicResourcePool.java:557)
- locked <0x00007fbf252387c0> (a com.mchange.v2.resourcepool.BasicResourcePool)
at com.mchange.v2.resourcepool.BasicResourcePool.checkoutResource(BasicResourcePool.java:477)
at com.mchange.v2.c3p0.impl.C3P0PooledConnectionPool.checkoutPooledConnection(C3P0PooledConnectionPool.java:525)
at com.mchange.v2.c3p0.impl.AbstractPoolBackedDataSource.getConnection(AbstractPoolBackedDataSource.java:128)
```

再看壓測圖如下：



原因可以根據log4j源碼分析如下：

```
public LocationInfo(Throwable t, String fqnOfCallingClass) {
  if(t == null)
return;

  String s;
  // Protect against multiple access to sw.
  synchronized(sw) {
t.printStackTrace(pw);
s = sw.toString();
sw.getBuffer().setLength(0);
  }
  //System.out.println("s is ["+s+"].");
  int ibegin, iend;

  // Given the current structure of the package, the line
  // containing "org.apache.log4j.Category." should be printed j
  // before the caller.

  // This method of searching may not be fastest but it's safer
  // than counting the stack depth which is not guaranteed to be
  // constant across JVM implementations.
  ibegin = s.lastIndexOf(fqnOfCallingClass);
  if(ibegin == -1)
return;
```

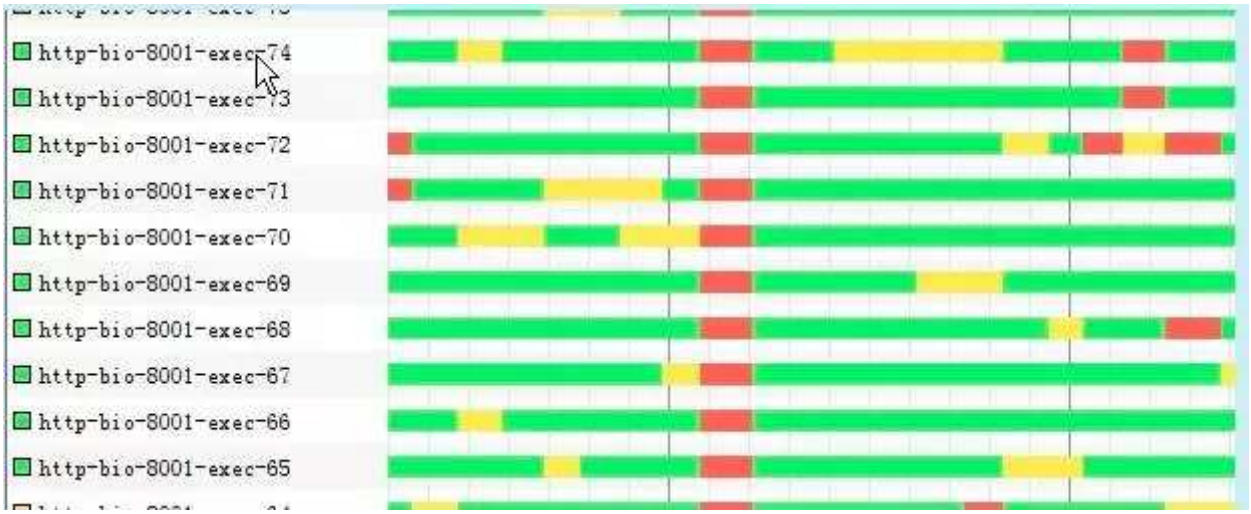
註：Log4j源碼裡用了synchronized鎖，然後又通過打印堆棧來獲取行號，在高並發下可能

就會出現上面的情況。

於是修改Log4j配置文件為：

```
%d %-5p %c [%t] - %m%n
```

上面問題解決，線程阻塞的情況很少出現，極大的提高了程序的並發能力，如下圖所示：



作者介紹

程超，易寶支付架構師，10年Java工作經驗，擅長和感興趣的技術領域是分佈式和大數據方面，目前主要從事金融支付類方向。