美团点评技术团队

常见性能优化策略的总结

晓明 ·2016-12-02 21:52

本文要感谢我职级评定过程中的一位评委，他建议把之前所做的各种性能优化的案例和方案加以提炼、总结，以文档的形式沉淀下来，并在内部进行分享。力求达到如下效果：

1. 形成可实践、可借鉴、可参考的各种性能优化的方案以及选型考虑点，同时配合具体的真实案例，其他人遇到相似问题时，不用从零开始。

2. 有助于开阔视野，除了性能优化之外，也能提供通用的常见思路以及方案选型的考虑点，帮助大家培养在方案选型时的意识、思维以及做各种权衡的能力。

文章在内部分享后，引起强烈分享，得到了不少同事和朋友的认可和好评，觉得对日常的工作有很好的指导作用。考虑到这些经验可能对业界同行也有帮助，所以在美团点评技术团队博客公开。

常见性能优化策略分类

代码

之所以把代码放到第一位，是因为这一点最容易引起技术人员的忽视。很多技术人员拿到一个性能优化的需求以后，言必称缓存、异步、JVM等。实际上，第一步就应该是分析相关的代码，找出相应的瓶颈，再来考虑具体的优化策略。有一些性能问题，完全是由于代码写的不合理，通过直接修改一下代码就能解决问题的，比如**for循环次数过多、作了很多无谓的条件判断、相同逻辑重复多次**等。

数据库

数据库的调优，总的来说分为以下三部分：

SQL调优

这是**最常用、每一个技术人员都应该掌握基本的SQL调优手段（包括方法、工具、辅助系统等）**。这里以MySQL为例，最常见的方式是，由**自带的慢查询日志或者开源的慢查询系统**定位到具体的出问题的SQL，然后使用**explain、profile**等工具来逐步调优，最后经过测试达到效果后上线。这方面的细节，可以参考MySQL索引原理及慢查询优化 (http://tech.meituan.com/mysql-index.html)。

架构层面的调优

这一类调优包括**读写分离、多从库负载均衡、水平和垂直分库分表**等方面，一般需要的改动较大，但是频率没有SQL调优高，而且一般需要DBA来配合参与。那么什么时候需要做这些事情？我们可以通过内部监控报警系统（比如Zabbix），定期跟踪一些指标数据是否达到瓶颈，一旦达到瓶颈或者警戒值，就需要考虑这些事情。通常，DBA也会定期监控这些指标值。

连接池调优

我们的应用为了实现数据库连接的高效获取、对数据库连接的限流等目的，通常会采用连接池类的方案，即每一个应用节点都管理了一个到各个数据库的连接池。随着业务访问量或者数据量的增长，原有的连接池参数可能不能很好地满足需求，这个时候就需要结合**当前使用连接池的原理、具体的连接池监控数据和当前的业务量**作一个综合的判断，通过反复的几次调试得到最终的调优参数。

缓存

分类

本地缓存（HashMap/ConcurrentHashMap、Ehcache、Guava Cache等），缓存服务（Redis/Tair/Memcache等）。

使用场景

什么情况适合用缓存？考虑以下两种场景：

- 短时间内相同数据重复查询多次且数据更新不频繁，这个时候可以选择先从缓存查询，查询不到再从数据库加载并回设到缓存的方式。此种场景较适合用单机缓存。
- 高并发查询热点数据，后端数据库不堪重负，可以用缓存来扛。

选型考虑

- 如果数据量小，并且不会频繁地增长又清空（这会导致频繁地垃圾回收），那么可以选择本地缓存。具体的话，如果需要一些策略的支持（比如缓存满的逐出策略），可以考虑Ehcache；如不需要，可以考虑HashMap；如需要考虑多线程并发的场景，可以考虑ConcurentHashMap。
- 其他情况，可以考虑缓存服务。目前从资源的投入度、可运维性、是否能动态扩容以及配套设施来考虑，我们优先考虑Tair。除非目前Tair还不能支持的场合（比如分布式锁、Hash类型的value），我们考虑用Redis。

设计关键点

什么时候更新缓存？如何保障更新的可靠性和实时性？

更新缓存的策略，需要具体问题具体分析。这里以门店POI的缓存数据为例，来说明一下缓存服务型的缓存更新策略是怎样的？目前约10万个POI数据采用了Tair作为缓存服务，具体更新的策略有两个：

- 接收门店变更的消息，准实时更新。
- 给每一个POI缓存数据设置5分钟的过期时间，过期后从DB加载再回设到DB。这个策略是对第一个策略的有力补充，解决了手动变更DB不发消息、接消息更新程序临时出错等问题导致的第一个策略失效的问题。通过这种双保险机制，

有效地保证了POI缓存数据的可靠性和实时性。

缓存是否会满，缓存满了怎么办？

对于一个缓存服务，理论上来说，随着缓存数据的日益增多，在容量有限的情况下，缓存肯定有一天会满的。如何应对？
① 给缓存服务，选择合适的缓存逐出算法，比如最常见的LRU。
② 针对当前设置的容量，设置适当的警戒值，比如10G的缓存，当缓存数据达到8G的时候，就开始发出报警，提前排查问题或者扩容。
③ 给一些没有必要长期保存的key，尽量设置过期时间。

缓存是否允许丢失？丢失了怎么办？

根据业务场景判断，是否允许丢失。如果不允许，就需要带持久化功能的缓存服务来支持，比如Redis或者Tair。更细节的话，可以根据业务对丢失时间的容忍度，还可以选择更具体的持久化策略，比如Redis的RDB或者AOF。

缓存被“击穿”问题

对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑另外一个问题：缓存被**“击穿”**的问题。

- 概念：缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

- 如何解决：业界比较常用的做法，是使用mutex。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。类似下面的代码：

```
public String get(key) {
    String value = redis.get(key);
    if (value == null) { //代表缓存值过期
        //设置3min的超时，防止del操作失败的时候，下次缓存过期一直不能load db
        if (redis.setnx(key_mutex, 1, 3 * 60) == 1) {  //代表设置成功
             value = db.get(key);
                    redis.set(key, value, expire_secs);
                    redis.del(key_mutex);
            } else {  //这个时候代表同时候的其他线程已经load db并回设到缓存了，这时候重试获取缓存值l
                    sleep(50);
                    get(key);  //重试
            }
        } else {
            return value;
        }
}
```

异步

使用场景

针对某些客户端的请求，在服务端可能需要针对这些请求做一些附属的事情，这些事情其实用户并不关心或者用户不需要立即拿到这些事情的处理结果，这种情况就比较适合用异步的方式处理这些事情。

作用

- 缩短接口响应时间，使用户的请求快速返回，用户体验更好。
- 避免线程长时间处于运行状态，这样会引起服务线程池的可用线程长时间不够用，进而引起线程池任务队列长度增大，从而阻塞更多请求任务，使得更多请求得不到技术处理。
- 线程长时间处于运行状态，可能还会引起系统Load、CPU使用率、机器整体性能下降等一系列问题，甚至引发雪崩。异步的思路可以在不增加机器数和CPU数的情况下，有效解决这个问题。

常见做法

一种做法，是额外开辟线程，这里可以采用额外开辟一个线程或者使用线程池的做法，在IO线程（处理请求响应）之外的线程来处理相应的任务，在IO线程中让response先返回。

如果异步线程处理的任务设计的数据量非常巨大，那么可以引入阻塞队列BlockingQueue作进一步的优化。具体做法是让一批异步线程不断地往阻塞队列里扔数据，然后额外起一个处理线程，循环批量从队列里拿预设大小的一批数据，来进行批处理（比如发一个批量的远程服务请求），这样进一步提高了性能。

另一种做法，是使用消息队列（MQ）中间件服务，MQ天生就是异步的。一些额外的任务，可能不需要我这个系统来处理，但是需要其他系统来处理。这个时候可以先把它封装成一个消息，扔到消息队列里面，通过消息中间件的可靠性保证把消息投递到关心它的系统，然后让这个系统来做相应的处理。

比如C端在完成一个提单动作以后，可能需要其它端做一系列的事情，但是这些事情的结果不会立刻对C端用户产生影响，那么就可以先把C端下单的请求响应先返回给用户，返回之前往MQ中发一个消息即可。而且这些事情理应不是C端的负责范围，所以这个时候用MQ的方式，来解决这个问题最合适。

NoSQL

和缓存的区别

先说明一下，这里介绍的和缓存那一节不一样，虽然可能会使用一样的数据存储方案（比如Redis或者Tair），但是使用的方式不一样，这一节介绍的是把它作为DB来用。如果当作DB来用，需要有效保证数据存储方案的可用性、可靠性。

使用场景

需要结合具体的业务场景，看这块业务涉及的数据是否适合用NoSQL来存储，对数据的操作方式是否适合用NoSQL的方式来操作，或者是否需要用到NoSQL的一些额外特性（比如原子加减等）。

如果业务数据不需要和其他数据作关联，不需要事务或者外键之类的支持，而且有可能写入会异常频繁，这个时候就比较适合用NoSQL（比如HBase）。

比如，美团点评内部有一个对exception做的监控系统，如果在应用系统发生严重故障的时候，可能会短时间产生大量exception数据，这个时候如果选用MySQL，会造成MySQL的瞬间写压力飙升，容易导致MySQL服务器的性能急剧恶化以及主从同步延迟之类的问题，这种场景就比较适合用Hbase类似的NoSQL来存储。

JVM调优

什么时候调？

通过监控系统（如没有现成的系统，自己做一个简单的上报监控的系统也很容易）上对一些机器关键指标（gc time、gc count、各个分代的内存大小变化、机器的Load值与CPU使用率、JVM的线程数等）的监控报警，也可以看gc log和jstat等命令的输出，再结合线上JVM进程服务的一些关键接口的性能数据和请求体验，基本上就能定位出当前的JVM是否有问题，以及是否需要调优。

怎么调？

1. 如果发现高峰期CPU使用率与Load值偏大，这个时候可以观察一些JVM的thread count以及gc count（可能主要是young gc count），如果这两个值都比以往偏大（也可以和一个历史经验值作对比），基本上可以定位是young gc频率过高导致，这个时候可以通过适当增大young区大小或者占比的方式来解决。
2. 如果发现关键接口响应时间很慢，可以结合gc time以及gc log中的stop the world的时间，看一下整个应用的stop the world的时间是不是比较多。如果是，可能需要减少总的gc time，具体可以从减小gc的次数和减小单次gc的时间这两个维度来考虑，一般来说，这两个因素是一对互斥因素，我们需要根据实际的监控数据来调整相应的参数（比如新生代与老生代比值、eden与survivor比值、MTT值、触发cms回收的old区比率阈值等）来达到一个最优值。
3. 如果发生full gc或者old cms gc非常频繁，通常这种情况会诱发STW的时间相应加长，从而也会导致接口响应时间变慢。这种情况，大概率是出现了“内存泄露”，Java里的内存泄露指的是一些应该释放的对象没有被释放掉（还有引用拉着它）。那么这些对象是如何产生的呢？为啥不会释放呢？对应的代码是不是出问题了？问题的关键是搞明白这个，找到相应的代码，然后对症下药。所以问题的关键是转化成寻找这些对象。怎么找？综合使用**jmap和MAT**，基本就能定位到具体的代码。

多线程与分布式

使用场景

离线任务、异步任务、大数据任务、耗时较长任务的运行**，适当地利用，可达到加速的效果。

注意：**线上对响应时间要求较高的场合，尽量少用多线程，尤其是服务线程需要等待任务线程的场合（很多重大事故就是和这个息息相关）**，如果一定要用，可以对服务线程设置一个最大等待时间。

常见做法

如果单机的处理能力可以满足实际业务的需求，那么尽可能地使用单机多线程的处理方式，减少复杂性；反之，则需要使用多机多线程的方式。

对于**单机多线程**，可以引入**线程池**的机制，作用有二：

- **提高性能，节省线程创建和销毁的开销**
- **限流，给线程池一个固定的容量，达到这个容量值后再有任务进来，就进入队列进行排队，保障机器极限压力下的稳定处理能力**在使用JDK自带的线程池时，一定要仔细理解构造方法的各个参数的含义，如**core pool size、max pool size、keepAliveTime、worker queue**等，在理解的基础上通过不断地测试调整这些参数值达到最优效果。

如果单机的处理能力不能满足需求，这个时候需要使用**多机多线程**的方式。这个时候就需要一些分布式系统的知识了。首先就必须引入一个单独的节点，作为调度器，其他的机器节点都作为执行器节点。调度器来负责拆分任务，和分发任务到合适的执行器节点；执行器节点按照多线程的方式（也可能是单线程）来执行任务。这个时候，我们整个任务系统就由单击演变成一个集群的系统，而且不同的机器节点有不同的角色，各司其职，各个节点之间还有交互。这个时候除了有多线程、线程池等机制，像RPC、心跳等网络通信调用的机制也不可少。后续我会出一个简单的分布式调度运行的框架。

度量系统（监控、报警、服务依赖管理）

严格来说，度量系统不属于性能优化的范畴，但是这方面和性能优化息息相关，可以说为性能优化提供一个强有力的数据参考和支撑。没有度量系统，基本上就没有办法定位到系统的问题，也没有办法有效衡量优化后的效果。很多人不重视这方面，但我认为它是系统稳定性和性能保障的基石。

关键流程

如果要设计这套系统，总体来说有哪些关键流程需要设计呢？
① 确定指标
② 采集数据
③ 计算数据，存储结果
④ 展现和分析

需要监控和报警哪些指标数据？需要关注哪些？

按照需求出发，主要需要二方面的指标：

1. 接口性能相关，包括单个接口和全部的QPS、响应时间、调用量（统计时间维度越细越好；最好是，既能以节点为维度，也可以以服务集群为维度，来查看相关数据）。其中还涉及到服务依赖关系的管理，这个时候需要用到服务依赖管理系统
2. 单个机器节点相关，包括CPU使用率、Load值、内存占用率、网卡流量等。如果节点是一些特殊类型的服务（比如MySQL、Redis、Tair），还可以监控这些服务特有的一些关键指标。

数据采集方式

通常采用异步上报的方式，具体做法有两种：第一种，发到本地的Flume端口，由Flume进程收集到远程的Hadoop集群或者Storm集群来进行运算；第二种，直接在本地运算好以后，使用异步和本地队列的方式，发送到监控服务器。

数据计算

可以采用离线运算（MapReduce/Hive）或者实时/准实时运算（Storm/Spark）的方式，运算后的结果存入MySQL或者HBase；某些情况，也可以不计算，直接采集发往监控服务器。

展现和分析

提供统一的展现分析平台，需要带报表（列表/图表）监控和报警的功能。

真实案例分析

案例一：商家与控制区关系的刷新job

背景

这是一个每小时定期运行一次的job，作用是用来刷新商家与控制区的关系。具体规则就是根据商家的配送范围（多个）与控制区是否有交集，如果有交集，就把这个商家划到这个控制区的范围内。

业务需求

需要这个过程越短越好，最好保持在20分钟内。

优化过程

原有代码的主要处理流程是：

1. 拿到所有门店的配送范围列表和控制区列表。
2. 遍历控制区列表，针对每一个控制区：
   a. 遍历商家的配送范围列表，找到和这个控制区相交的配送范围列表。
   b. 遍历上述商家配送范围列表，对里面的商家ID去重，保存到一个集合里。
   c. 批量根据上述商家ID集合，取到对应的商家集合。
   d. 遍历上述商家集合，从中拿到每一个商家对象，进行相应的处理（根据是否已是热门商家、自营、在线支付等条件来判断是否需要插入或者更新之前的商家和控制区的关系）。
   e. 删除这个控制区当前已有的，但是不应该存在的商家关系列表。

分析代码，发现第2步的a步骤和b步骤，找出和某控制区相交的配送范围集合并对商家ID去重，可以采用R树空间索引的方式来优化。具体做法是：

- 任务开始先更新R树，然后利用R树的结构和匹配算法来拿到和控制区相交的配送范围ID列表。
- 再批量根据配送范围ID列表，拿到配送范围列表。

- 然后针对这一批配送范围列表（数量很小），用原始多边形相交匹配的方法做进一步过滤，并且对过滤后的商家ID去重。

这个优化已经在第一期优化中上线，整个过程耗时**由40多分钟缩短到20分钟以内。**

第一期优化改为R树以后，运行了一段时间，随着数据量增大，性能又开始逐渐恶化，一个月后已经恶化到50多分钟。于是继续深入代码分析，寻找了两个优化点，安排第二期优化并上线。

这两个优化点是：

- 第2步的c步骤，原来是根据门店ID列表从DB批量获取门店，现在可以改成mget的方式从缓存批量获取（此时商家数据已被缓存）；
- 第2步的d步骤，根据是否已是热门商家、自营、在线支付等条件来判断是否需要插入或者更新之前的商家和控制区的关系。

上线后效果

通过日志观察，执行时间**由50多分钟缩短到15分钟以内**，下图是截取了一天的4台机器的日志时间（单位：**毫秒**）：

可以看到，效果还是非常明显的。

案例二：POI缓存设计与实现

背景

2014年Q4，数据库中关于POI（这里可以简单理解为外卖的门店）相关的数据的**读流量**急剧上升，虽然说加入从库节点可以解决一部分问题，但是毕竟节点的增加是会达到极限的，达到极限后主从复制会达到瓶颈，可能会造成数据不一致。所以此时，急需引入一种新的技术方案来分担数据库的压力，降低数据库POI相关数据的读流量。另外，任何场景都考虑加DB从库的做法，会对资源造成一定的浪费。

实现方案

基于已有的经过考验的技术方案，我选择Tair来作为缓存的存储方案，来帮DB分担来自于各应用端的POI数据的读流量的压力。理由主要是从**可用性、高性能、可扩展性、是否经过线上大规模数据和高并发流量的考验、是否有专业运维团队、是否有成熟工具**等几个方面综合考量决定。

详细设计

第一版设计

缓存的更新策略，根据业务的特点、已有的技术方案和实现成本，选择了用MQ来接收POI改变的消息来触发缓存的更新，但是这个过程有可能失败；同时启用了key的过期策略，并且调用端会先判断是否过期，如过期，会从后端DB加载数据并回设到缓存，再返回。通过两个方面双保险确保了缓存数据的可用。

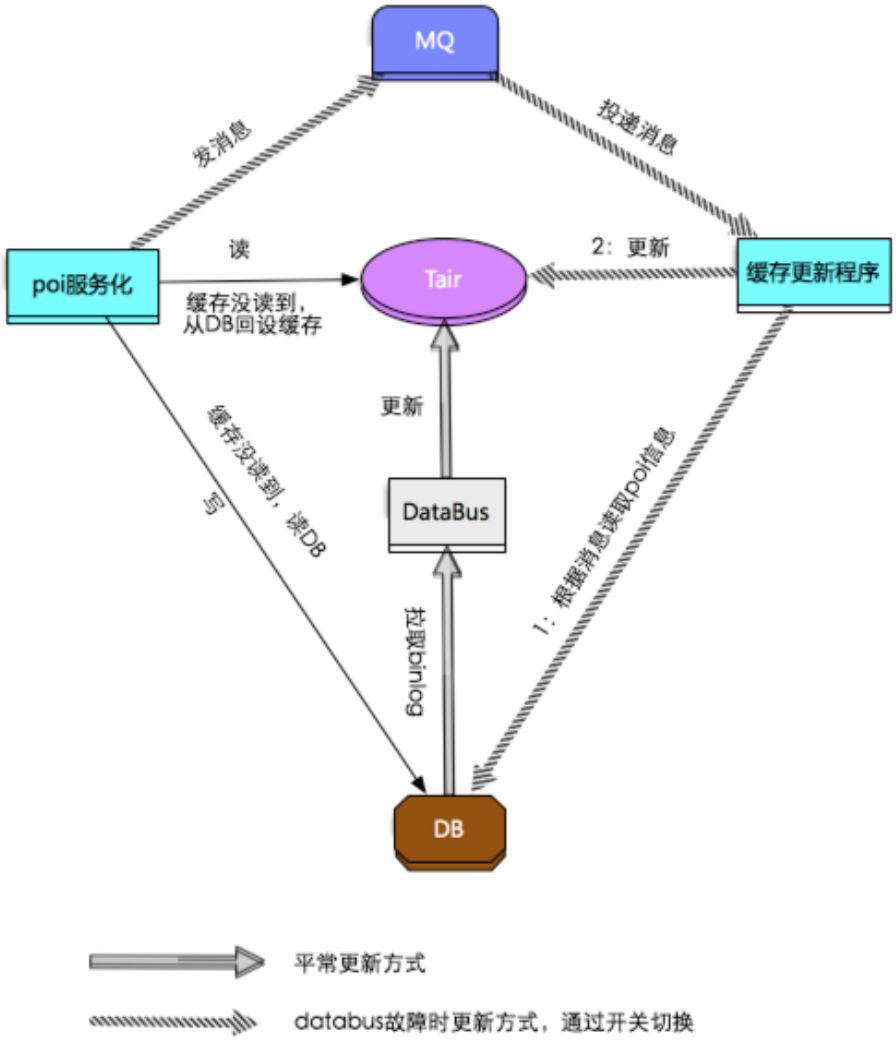
第二版设计

第一版设计运行到一段时间以后，我们发现了两个问题：

1. 某些情况下不能保证数据的实时一致（比如技术人员手动改动DB数据、利用MQ更新缓存失败），这个时候只能等待5分钟的过期时间，有的业务是不允许的。
2. 加入了过期时间导致另外一个问题：Tair在缓存不命中的那一刻，会尝试从硬盘中Load数据，如果硬盘没有再去DB中Load数据。这无疑会进一步延长Tair的响应时间，这样不仅使得业务的超时比率加大，而且会导致Tair的性能进一步变差。

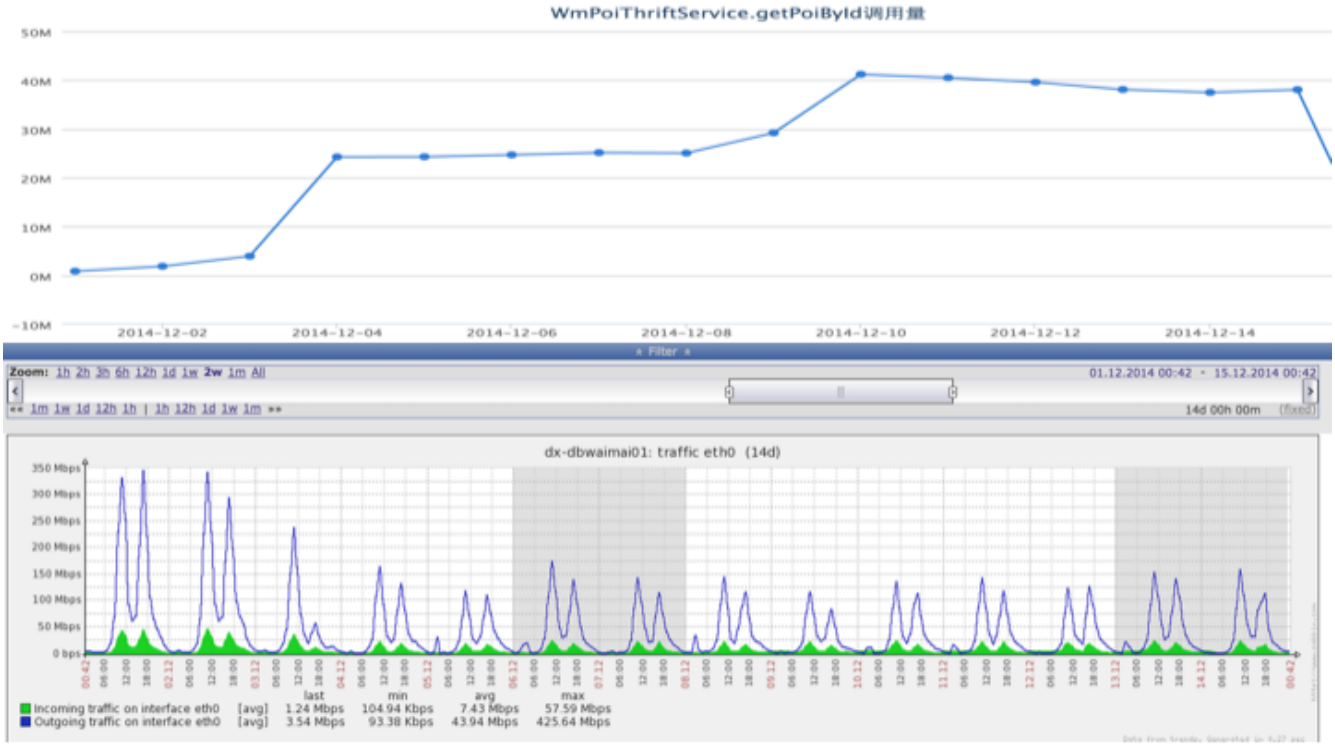
为了解决上述问题，我们从美团点评负责基础架构的同事那里了解到Databus (https://github.com/linkedin/databus)可以解决缓存数据在某些情况下不一致的问题，并且可以去掉过期时间机制，从而提高查询效率，避免tair在内存不命中时查询硬盘。而且为了防止DataBus单点出现故

障影响我们的业务，我们保留了之前接MQ消息更新缓存的方案，作了切换开关，利用这个方案作容错，整体架构如下：



上线后效果

上线后，通过持续地监控数据发现，随着调用量的上升，到DB的流量有了明显地减少，极大地减轻了DB的压力。同时这些数据接口的响应时间也有了明显地减少。缓存更新的双重保障机制，也基本保证了缓存数据的可用。见下图：



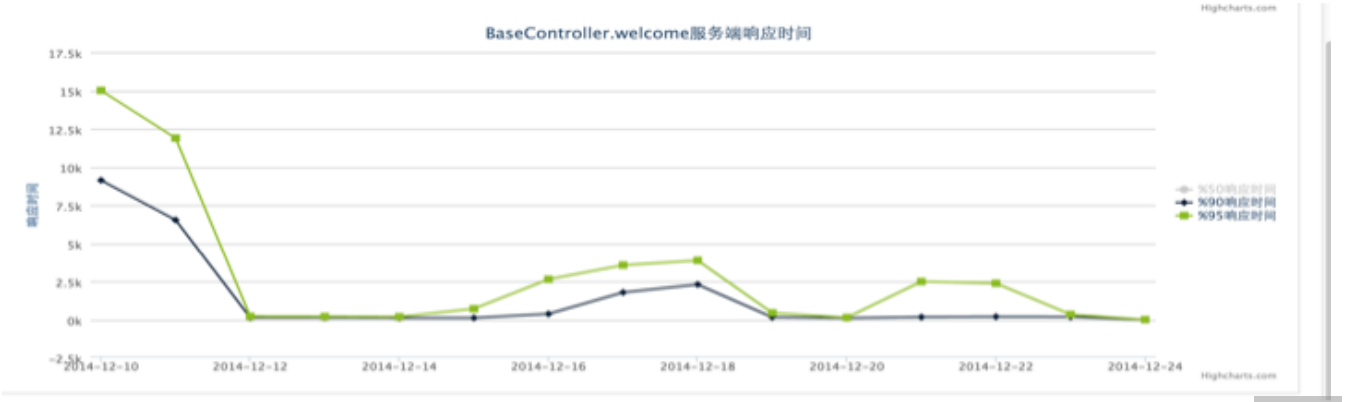
案例三：业务运营后台相关页面的性能优化

背景

随着业务的快速发展，带来的访问量和数据量的急剧上升，通过我们相应的监控系统可以发现，系统的某些页面的性能开始出现恶化。 从用户方的反馈，也证明了这点。此时此刻，有必要迅速排期，敏捷开发，对这些页面进行调优。

欢迎页

- 需求背景：欢迎页是地推人员乃至总部各种角色人员进入外卖运营后台的首页，会显示地推人员最想看到最关心的一些核心数据，其重要性不言而喻，所以该页面的性能恶化会严重影响到用户体验。因此，首先需要优化的就是欢迎页。通过相应定位和分析，发现导致性能恶化的主要原因有两个：数据接口层和计算展现层。
- 解决方案：对症下药，分而治之。经过仔细排查、分析定位，数据接口层采用接口调用批量化、异步RPC调用的方式来进行有效优化，计算展现层决定采用预先计算、再把计算好的结果缓存的方式来提高查询速度。其中，缓存方案根据业务场景和技术特点，选用Redis。定好方案后，快速开发上线。
- 上线效果：上线后性能对比图，如下：



组织架构页