getArticleList

无特殊参数查询

getArticleList

getTotal优化(TODO)

getTagListByArticleId优化(TODO)

Limit优化(doing)

**关于本博客在本地运行环境下的性能优化**

博客上线后，可能会被很多人同时访问，因此，在上线前，最好对博客进行一定程度的优化。本博客分为前端项目，后端项目，在本文章中，作者主要探讨、实施了对后端项目的优化。这些优化包括对getArticleList接口，getCategoryList

[深入了解Spring Security的实现原理 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/72305502)

Threadlocal context --代码

1. getArticleList接口

getArticleList由4钟分页查询组成：用标签tagId查询，用类别categoryId查询，用内容queryContent查询，以及普通分页查询。

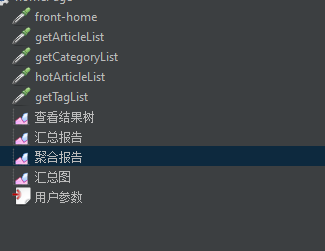
1-1.getArticleList作为普通分页查询接口的性能优化。

* 1. 背景：

用户在进入首页后，会调用很多接口，包括前端工程接口front-home，后端工程接口：普通分页查询getArticleList,getCategoryList,hotArticleList,getTagList。通过jmeter进行测试后，发现getArticleList在进入首页这个功能中占用的时间相对于其他后端工程接口所占用的时间比较长，因此我们考虑对其进行优化。而用户在首页点击“加载更多”按钮后，调用的后端接口只有普通分页查询getArticleList。因此我们在此进行统一优化。

***进入首页功能性能测试如下：***

用50线程数/秒对进入首页进行模拟并发请求，结果如下图：



（内容为模拟进入首页的线程组，其中front-home是对前端服务的取样器，其余为后端取样器）



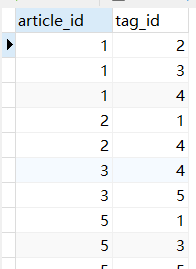
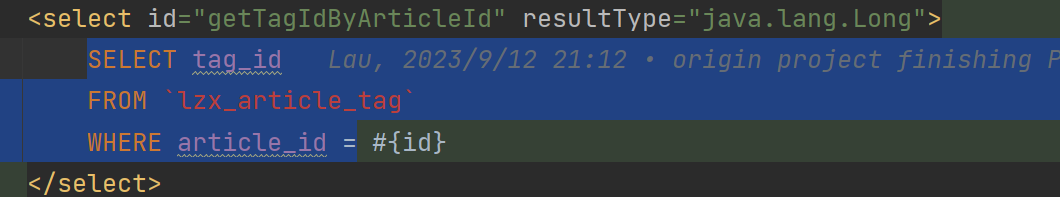
（内容为吞吐量平稳之后，各取样器的相应请求数据）

从上面两图不难看出，在后端接口中，getArticleList接口的平均访问时间为11390ms，中位数为9299ms，在后端接口中明显性能相对不好，因此，优化getArticleList接口成了优化进入首页的重要一环。

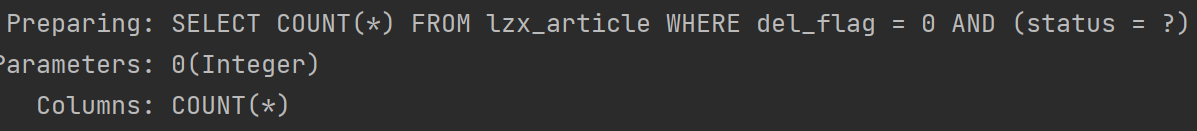
* 1. “进入首页”的getArticleList优化-可以考虑缓存

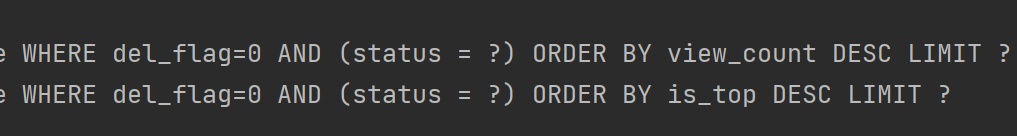
如何优化？“进入首页”的getArticleList接口的代码逻辑都比较简单，主要是从数据库获取内容，因此重点放在优化查询数据库上。同时，不难想到mybatisplus的数据库操作日志也占用性能，因此也要考虑关闭后端mybatisplus日志，当然关闭日志（TODO）可以暂时放一放。

要想优化查询数据库，首先我们需要看进入首页的getArticleList包括哪些查询。从代码不难得知，该接口的查询主要包括getCategoryName(categoryId)，getTagList(articleId),以及分页查询。getCategoryName查询的是根据lzx\_category表的主键进行查询，而这张表又很小（少于100行），所以暂不考虑优化。对于getTagList(articleId),该查询又分为从article\_tag这张2个字段的relation表根据主键查询TagId（如下面的两张图），以及从lzx\_tag这张不大（少于100行）表根据主键查询，所以也暂时不考虑优化。



那么我们就来看分页查询。从前端项目进入首页，然后查看后端日志，如下两图所示。从下两图不难看出，“进入首页”涉及到2条sql查询，其中第一条（TODO，第一条也尝试优化）是用来查询所有记录数。第二条包含del\_flag,status,view\_count,is\_top字段。



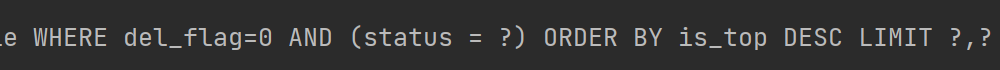


（“进入首页”接口的后端sql语句）

首先看del\_flag,删除字段只有两个值0和1，因此没有必要对此建立索引，status，is\_top同理。因此，想要使用索引优化，并不合适。但是鉴于访问首页的这些article经常访问，而且对于数据一致性要求并不高，我们可以考虑将这些article先放入redis缓存（TODO）中。

* 1. 点击“加载更多”的性能优化

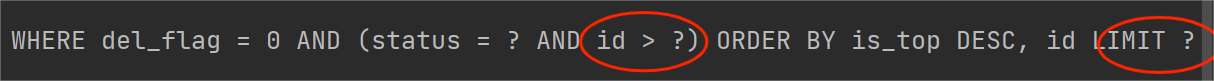
点击“加载更多”按钮的getArticleList代码与“进入首页”一致，因此根据“进入首页”优化的讨论，我们把优化“加载更多”的重点放在优化分页查询。这时的sql仍然为2条，与“进入首页”不同的是，第二条sql语句变为了如下图：



（Sql从limit m变成了 limit m, n）

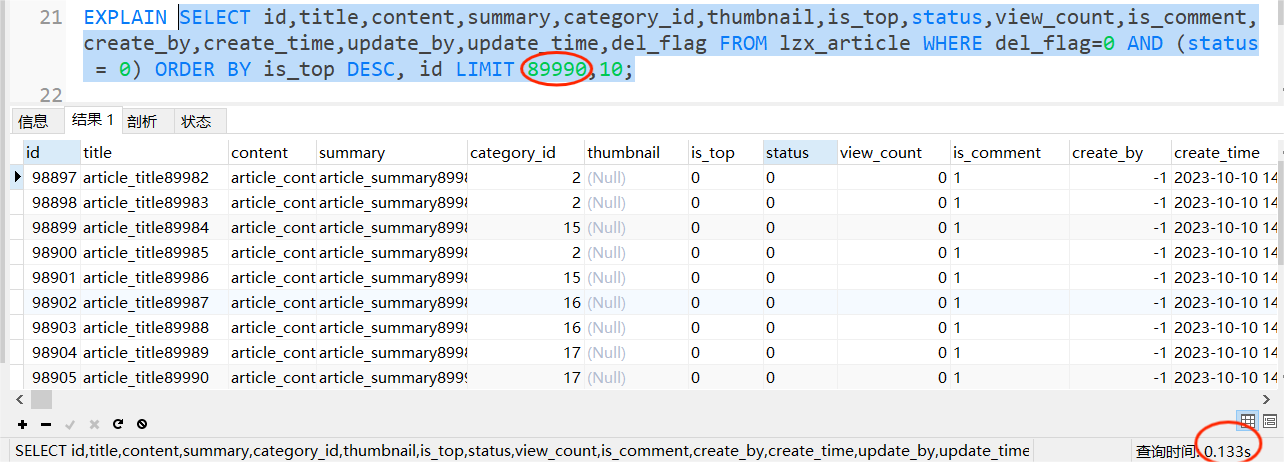
Sql从limit m变成了 limit m, n。这种sql在article数据量特别大时会导致全局扫描太多数据并丢弃，很浪费性能。因此我们对此尝试优化。我们需要自定义分页查询，但这时我发现原代码并不好仅仅针对首页中的getArticleList进行自定义分页查询，因此我对其进行了一定程度的更改。

那么这种情况下如何进行优化？我的想法是，既然利用del\_flag,is\_top,status建立索引都不合适，那么，不妨引入上一页的最大id，利用主键索引先进行范围查找，然后再排序与截取，这样就避免了扫描浪费。于是，更改代码，使原语句变成如下图所示（TODO，无论优不优化，会出现数据重复的问题）。

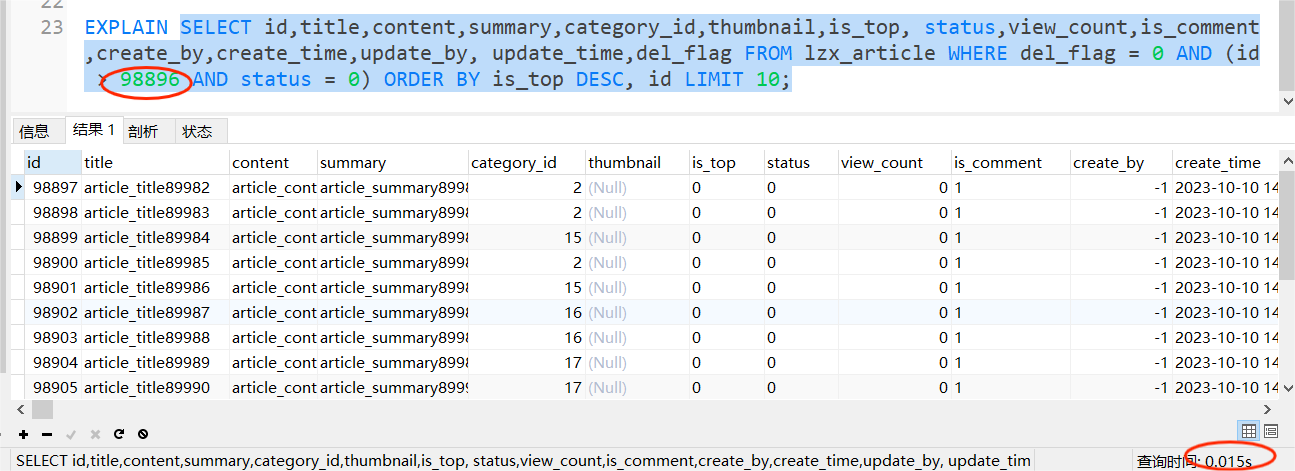


（先用id排序，再进行后续操作）

那么，优化的效果如何？拿查询行数为89991-90000的文章来进行测试，首先来看sql执行时间，从下图来看，不难看出，优化后的sql比优化前的sql在页数比较多时，消耗时间减少了80%多。



（优化前，可以看到查询时间为0.133s）



（优化后，可以看到查询时间缩短为0.015s）

可以看到，sql优化效果不错。那么实际业务优化情况如何？我们在jmeter中，对首页中的“加载更多”进行测试，结果如下图所示，可以看到，当页数比较大时，吞吐量从10/s变成了300/sec。（注：1.经过测试，页数比较小时，两者性能相近，在这里不予展示。2.“加载更多”的业务接口主要是后端的getArticleList，因此本次测试忽略前端接口）



（“首页查询”的“加载更多”测试，上图为接口参数）



（“首页查询”的“加载更多”测试，上图为优化后测试数据）



（“首页查询”的“加载更多”测试，上图为优化前测试数据，其中异常量太大是因为请求处理比较慢，很多请求还没收到信息就被停止了）

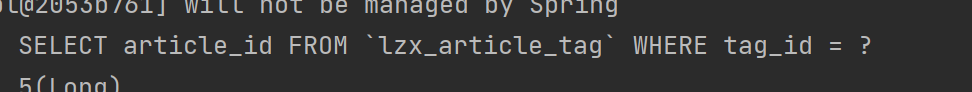
1-2.getArticleList作为根据tagId查询接口的优化（TODO，不用分页，还可以省去第二条sql，进一步优化）(TODO, 目前会导致page总数获取不对，导致前端无法加载更多)

(1) 背景

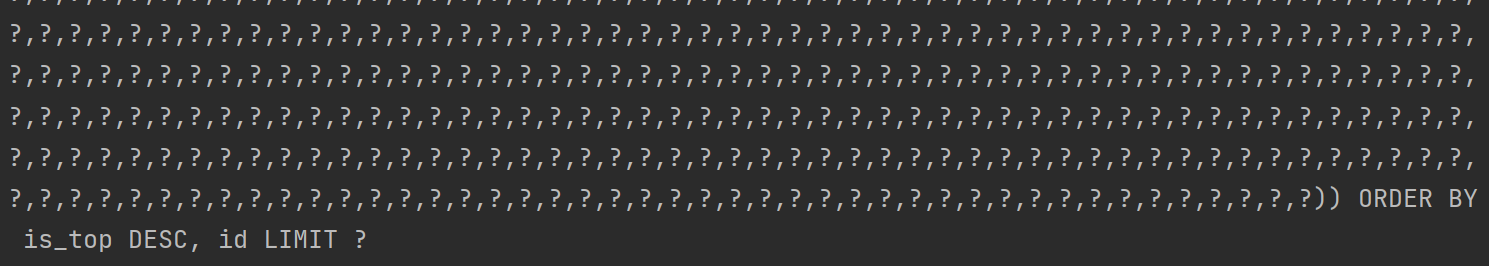
在文章的tagList或者任一页面右侧的tagList点击某一个tag，会查询有此tag的文章。但此接口涉及到查询lzx\_article\_tag，lzx\_article两张行数比较大的表（都在90000行以上），因此我们考虑对此进行优化。

1. 优化

该接口查询顺序为：首先根据tagId查询articleId，其次根据这些articleId去lzx\_article表中查询article（包括判断id，status，del\_flag，orderby以及limit）。一般情况下，第一次查询会查询出包含指定tagId的所有article\_id。然后再由第二条查询先查询出这些article\_id所对应的所有记录，然后排序，limit筛选指定数量记录；在页数比较大时，limit又会扫描太多行并丢弃，又浪费了性能。这两条sql语句如下图所示。



（tagId查询articleId，可以看到没有对行数做出限制）

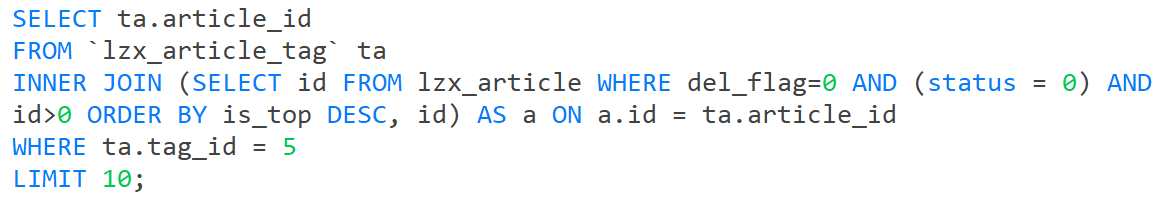


（根据这些articleId去lzx\_article表中查询article，可以看到传入的article\_id太多，浪费太多）

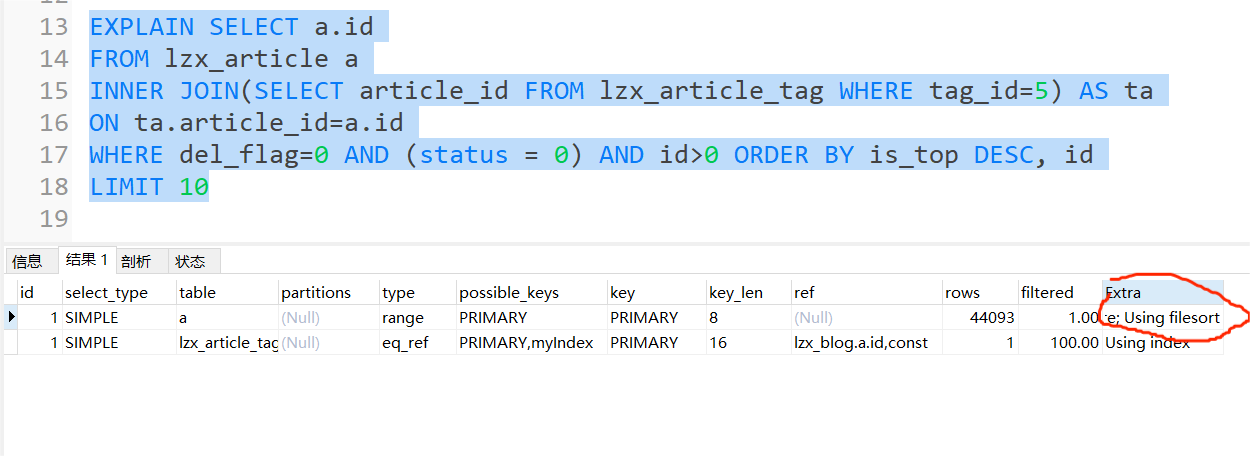
经过测试，发现无论页数是多少，第二条查询的时间都明显要高于第一条查询。因此我们尝试重点放在优化第二条查询。上面的分析不难看出，传入的article\_id太多，以及页数太多时limit的扫描与丢弃，是第二条查询性能差的主要原因。很容易想到，只传每一页大小的article\_id，会避免article\_id太多以及页数太多时limit语句的浪费。因此我们对此进行尝试。想要第二条传入的article\_id减少，就要在第一条语句时进行limit行数控制，以及status，del\_flag，orderby的业务逻辑处理（如果只进行limit行数控制而不进行其他业务逻辑处理，可能导致第二条查询得到的结果数少于预期）。

想要指定tagId的article，就要去lzx\_article\_tag表去查询，而其他例如status,del\_flag的逻辑，离不开lzx\_article表，因此我们很自然的想到了连表查询，因为是求交集，因此使用inner join。

因为sql其实比较简单，因此这里直接贴出优化后的第一条sql如下图所示：



这里用lzx\_article创建临时表。如果用lzx\_article\_tag创建临时表，在mysql选择lzx\_article作为驱动表时，因为我们没有设置相关索引，因此会被mysql进行filesort排序，如下图所示。但当用lzx\_article创建临时表时，会用内存排序，比filesort排序要好很多（TODO，有没有id>0会导致rows折半，为什么）（TODO，内存排序只有using where,为什么）。

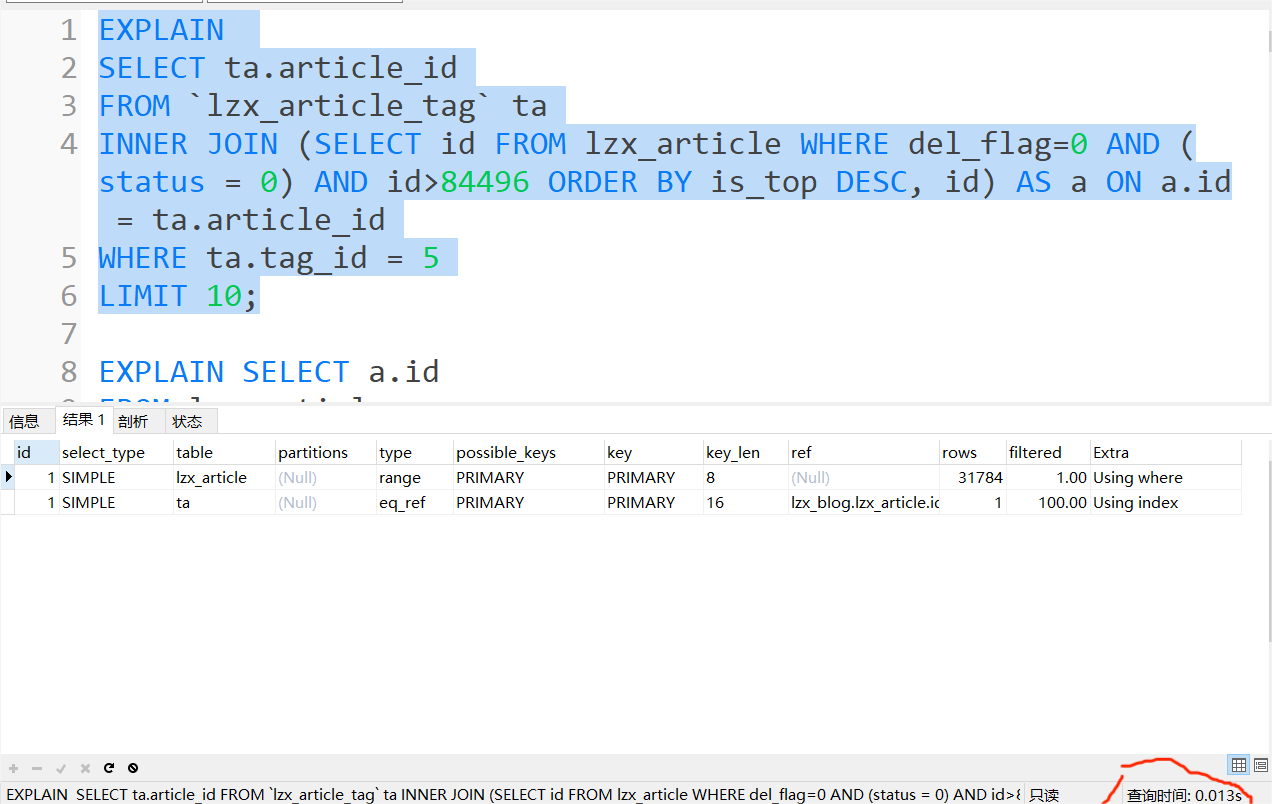
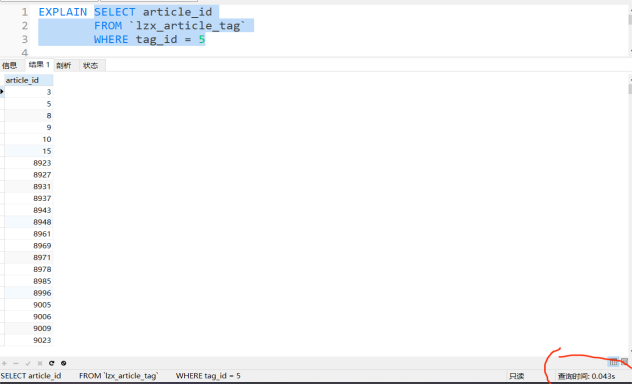


（外部排序，效率不高）

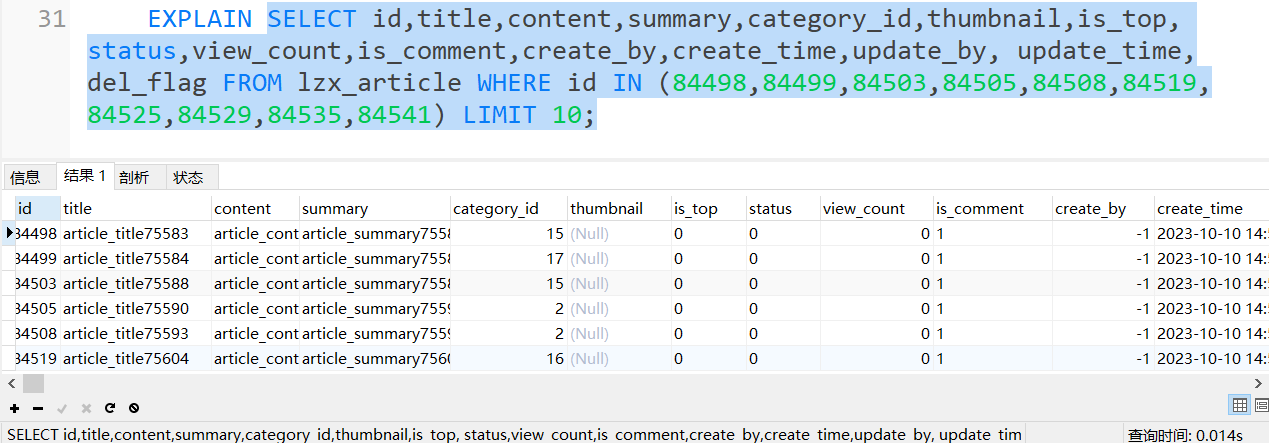
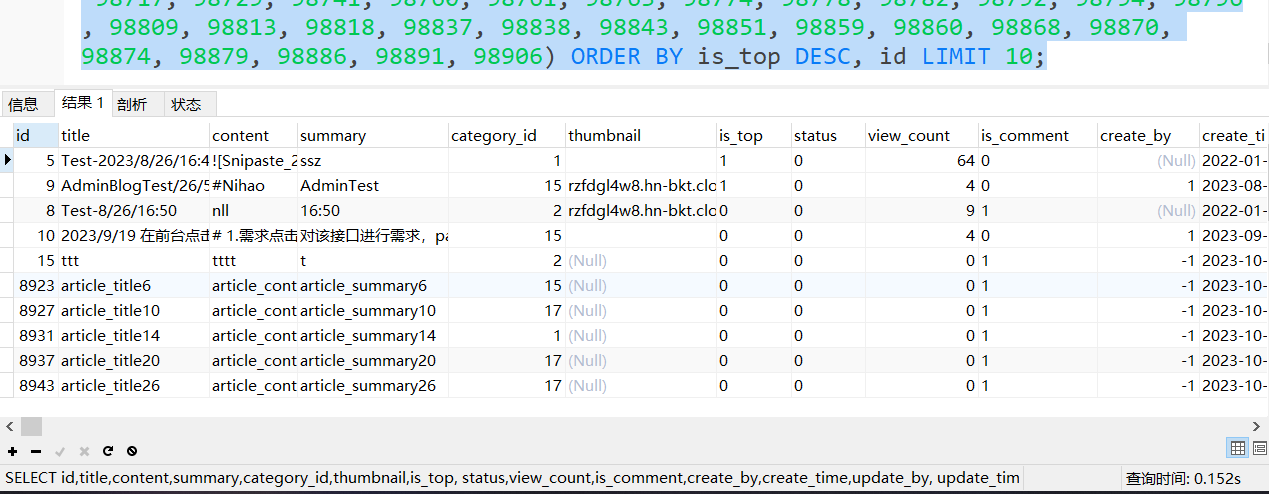
我们再来看第二条sql，既然传入了del\_flag,status,id,tagId都满足业务需求的article\_id，那么没有必要再重新去执行一遍业务了。只需要利用id进行主键索引查询即可。将在讲述下面的优化效果展示该sql。

接下来我们看看优化效果如何？分为sql优化效果和接口优化效果。

首先看sql优化效果。可以看到，两条sql的优化效果都很明显。



（第一条sql的优化，可以看到查询时间从平均0.043s变成了0.013s，且经过测试，无论页数多大，优化后查询时间都在0.013s左右）



（第二条sql的优化，可以看到查询时间从平均0.152s变成了0.014s，且经过测试，无论页数多大，优化前查询平均时间在0.152s，优化后为0.014s）

再来看接口优化效果。从下两图可以看到，优化后的效果很明显。（注：经过测试，无论页数是小，中，大，优化前吞吐量在7-9/s，优化后吞吐量在280-300/s，这里只展示部分）



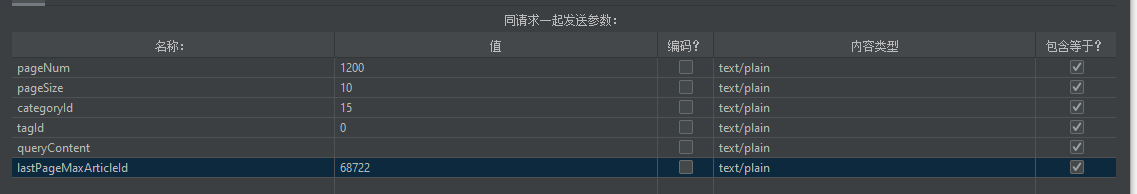
（优化前，可以看到吞吐量非常低）



（优化后，可以看到吞吐量从8/s变成了290/s左右）

1-3 getArticleList作为根据categoryId查询接口的优化

该项业务逻辑与1-1几乎相同，仅仅是查询时多在where后多添加了一个category字段的判断。因此优化逻辑和1-1相同，在这里略去优化分析。只展示页数大时的优化结果。从下三张图可以看出，在页数比较大时，优化后性能明显高于优化前（经过测试，当页数比较小时，优化前后性能相近，优化前吞吐量为90/s-57/s随页数增大而减小，优化后吞吐量为90/s-170/s随页数增大而减小）。（TODO，为什么优化后与1-1性能相差如此大）



（本次测试的参数）



（优化前性能指标）



（优化后性能指标，可以看到吞吐量等指标明显比优化前好）

1-4 getArticleList作为根据queryContent查询接口的优化

(1) 背景

用户可以在任意页面的上方输入文章内容进行模糊查询。当文章的内容比较大或者文章数量比较多，查询会显著变慢，因此对此尝试优化。

(2)优化--mysql的两种优化方案都不行，采用其他数据库

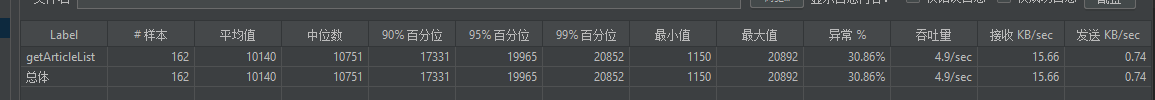
考虑到上面接口的优化方案，作者先是考虑了传上一页的最大articleId，但模糊查询是一个很耗时的操作，因此无论页数大小，经过测试，吞吐量都不是很好。因此，作者又考虑在传入articleId的基础上，通过建立全文索引解决模糊查询的问题。但经过测试，建立全文索引，吞吐量没有增加，sql执行时间也可能更慢，因此这个方案也不行。

因此这里使用搜索性能上比较好的elasticsearch，作为根据内容查询接口的数据库。

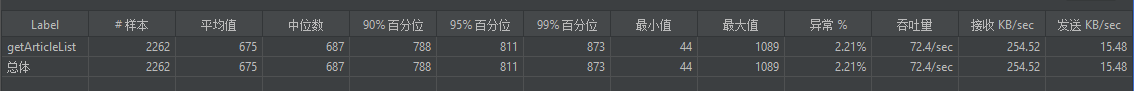
优化效果如何？从下图可以看到，无论是页数小还是大，优化后的性能都明显好于优化前。（注：因为ealsticsearch并不是很支持跳页，因此这里的“大页”只展示到999页）



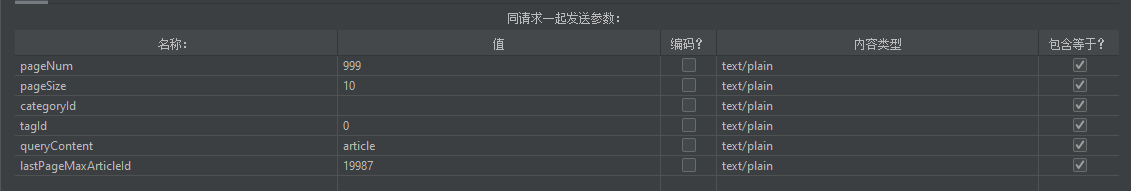
（页数小的请求参数）



（优化前，可以看到吞吐量很小，异常比例大是因为太多请求还没来得及收到response就关闭了）



（优化后，可以看到吞吐量明显增大）



（页数大的请求参数）



（优化后的吞吐量为17-20/s，优化前的吞吐量仍然为2-5/s，这里忽略优化前的图片）

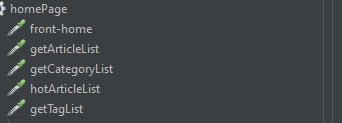
1. getCategoryList接口
2. 背景

客户端前端的头部菜单，每次加载时会获取categoryList。但原处理逻辑是：获取所有article->得到不重复的categoryId->根据categoryId去获取category。获取所有article，无疑大幅度减少了性能。

1. 优化--更改业务逻辑

有两种优化方案：要么像获取tag列表那样，不在乎category有没有存在于某个文章中，要么优化“获取所有article”这条语句。考虑到博客本身的category可能并不多，category本身的实时性需求也不是很大，因此采用类似获取tag列表那样。而根据categoryId查询本身的性能也不错，因此无需担心根据categoryId查询的请求会相对于原方案更多。

优化效果如何？从下图可以看到，优化后各项性能明显好于优化前。这里值得注意的是getCategoryList优化前的getArticleList也是非常慢，但优化后的优化程度要明显高于hotArticleList和getTagList,我认为这是由于优化后lzx\_article数据表的压力由于getCategoryList不再查询而减小。



（请求参数）

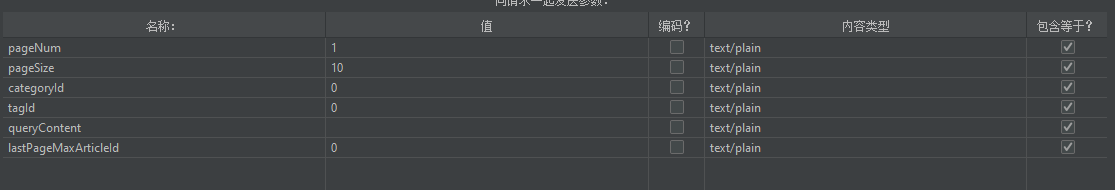


（优化前，可以看到吞吐量非常少）

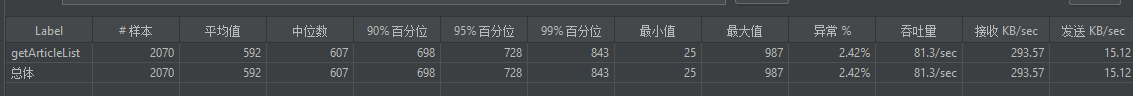


（优化后，可以看到性能明显提升）

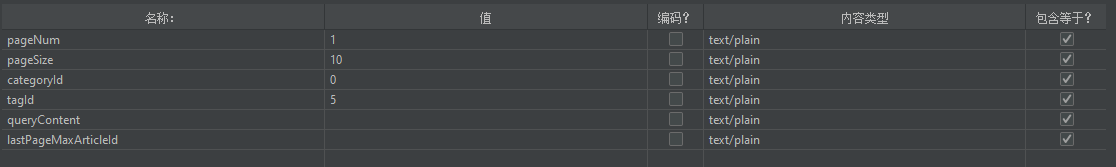
-----------------------------



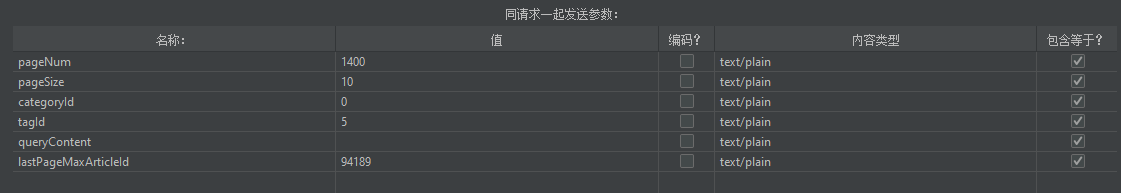




=============



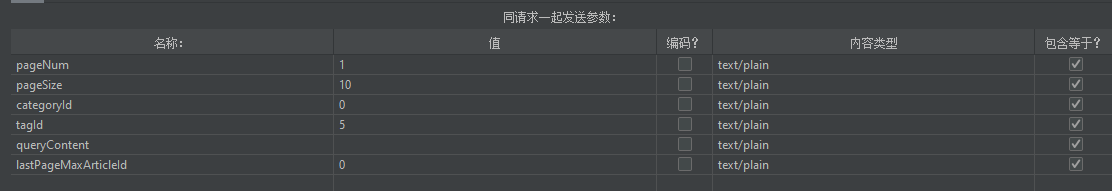






进阶后

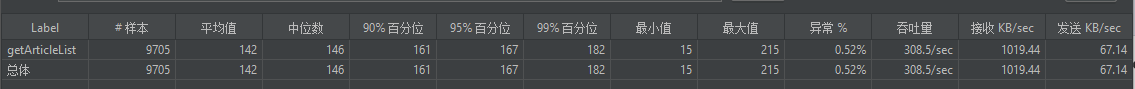
pageNum=1





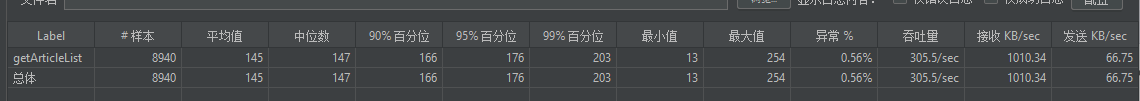
pageNum=320



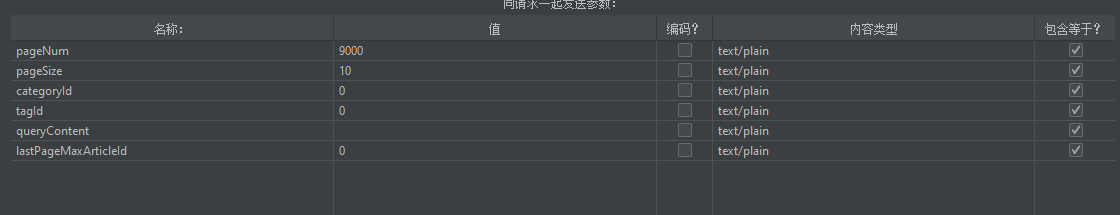


pageNum=1390





-----------------------------





======================

-------------------

数据量小



=============