

二级缓存架构极致提升系统性能

原创 木将 阿里云开发者 2024年09月11日 08:30 浙江



微信扫一扫
关注该公众号



阿里妹导读



本文详细阐述了如何通过二级缓存架构设计提升高并发下的系统性能。

前言

随着k8s成为用云新界面，容器成为众多用户“弹性”的利器，因此容器的创建天生具备高并发特性。

高并发、大数据量下，为了更好的容器弹性体验，笔者通过二级缓存的设计，成功优化了系统性能、资源消耗、系统容量。

但持续压榨性能的道路是曲折的。各种缓存方案需要考虑非常多因素，包括缓存的多级架构、预热、击穿、刷新、运维等。

让我们先看看系统性能、资源消耗、系统容量，到底优化了多少？然后详细介绍带来性能提升的二级缓存架构，以及各种缓存方案的设计、比较、落地。

优化结果

笔者对系统进行多次压测，观察了二级缓存等优化手段上线前后，高QPS下的资源消耗、RT和系统容量。

1、资源消耗大幅下降

可以看到，上线了二级缓存等优化手段后，相同QPS下，CPU使用率大幅下降。



2、RT和系统容量优化

由于计算资源消耗的下降，同QPS下的平均RT也大幅下降。此外我们发现，优化前，随着QPS不断提升，RT明显变慢、变慢幅度很大，优化后，RT变慢的幅度明显减缓。这意味着，系统容量也更加深不可测了。

系统瓶颈在哪？

持续压榨性能，前提是知道瓶颈在哪。笔者怎么定位瓶颈的呢？-> cpuProfile火焰图。

高QPS下，分析了系统cpu火焰图，发现50%+的CPU瓶颈在大量的业务数据处理，主要两块：

1、从redis取业务pojoList时的fastjson array反序列化。背景是和redis交互的数据量特别大，笔者对缓存value进行过压缩，在解压缩后的string转为List<Pojo>的过程，本质用时间换了空间。

2、调用redis获取数据后的alibaba.cachejson的POJO解析。

在高并发、大数据量的业务背景下，这些过程的耗时被笛卡尔积地放大。

既然分布式缓存CPU资源消耗是瓶颈，那么引入本地缓存，就能解这个问题。另外，本地缓存还能提供服务容灾能力。

数据属性和缓存选型

数据属性

设计缓存方案前，先分析业务数据属性：

- 符合key -> value的数据模式
- 读多写少
- 变更频率低
- 数据一致性要求不高

简直就是缓存的绝佳使用场景！那么用本地缓存，还是分布式缓存呢？还是二级缓存？

本地/分布式缓存特性

基于业务属性，我们来match本地/分布式缓存特性：

本地缓存

- 优点
 - 访问速度快；
 - 减少网络开销：不存在redis流量瓶颈(之前由于数据valueList量过大，导致redis网络流量超过redis实例SLA，还做过一次gzip压缩)；
- 缺点
 - 可用性相对差：如果应用实例宕机/重启, 缓存数据会丢失；
 - 资源限制：受限于单机内存大小，不适合大规模数据缓存(相对内存大小来说，本业务数据量还是较小，也是相对静态数据，在内存还是妥妥可存的核心业务数据)；

分布式缓存

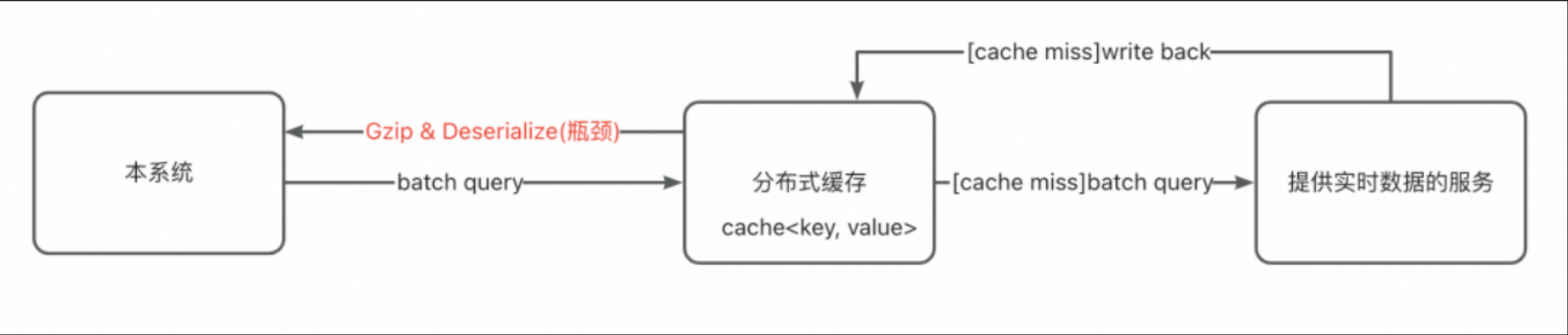
- 优点
 - 高可用：数据共享，不会因为业务机器重启丢失；
 - 大容量：可以水平扩展增加存储容量；
- 缺点
 - 序列化cpu瓶颈(本系统大数据量+高调用量下尤其突出)；
 - 网络开销：存在redis流量瓶颈(本系统大数据量+高调用量下尤其突出)；

本地+分布式二级缓存

- 优点
 - 结合两者优势，层次化加速；
- 缺点
 - 对本系统代码侵入较大，实现较复杂；
 - 资源消耗：额外的缓存层级会占用更多计算和存储资源；

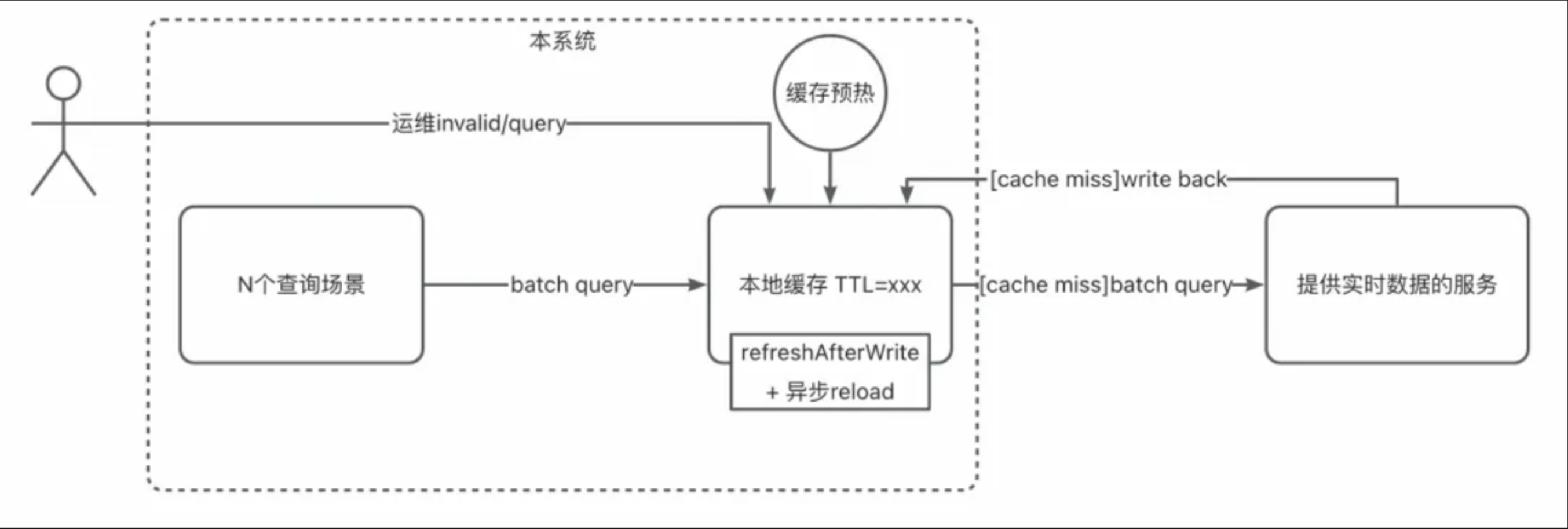
缓存方案比较

系统现状



结合业务场景和系统现状，提出以下三种方案。对多级架构、预热、击穿、刷新、运维多个角度分析。

方案1：本地缓存+guava refreshAfterWrite



1、架构

仅本地缓存。根据业务场景设置失效时间。

2、cache miss处理、刷新

使用Guava cache原生的refreshAfterWrite+异步reLoader机制，进行缓存的刷新，保证时效性。当对应key距离写入时间点存活超过TTL后，guava会自动执行我们在reLoader中写的业务逻辑，从提供实时数据的系统自动拉取最新的<key, value>，更新缓存值。

在本业务场景下，缓存的key&value设计有两种方案。

- a. <key, value>。由于guava cache loader只能by key更新，所以，如果在单地域有N台机器，每个机器都查M个key，在缓存刷新时最大有M*N个查询请求打到提供实时数据的系统，无法使用批量查询能力，而M数量非常大，对下游压力会翻倍。
- b. <"all_data"(hardCode), Map<key, value>>。这样可以使用到批量查询的能力，但是当某些<key, value>在本地缓存没有时，guava的loader就无法识别cache miss并从下游系统捞数据了。

3、预热

启动时全量预热。

4、运维

暴露dubbo/http服务，by host运维；或者重启机器。

因此：

- 优点

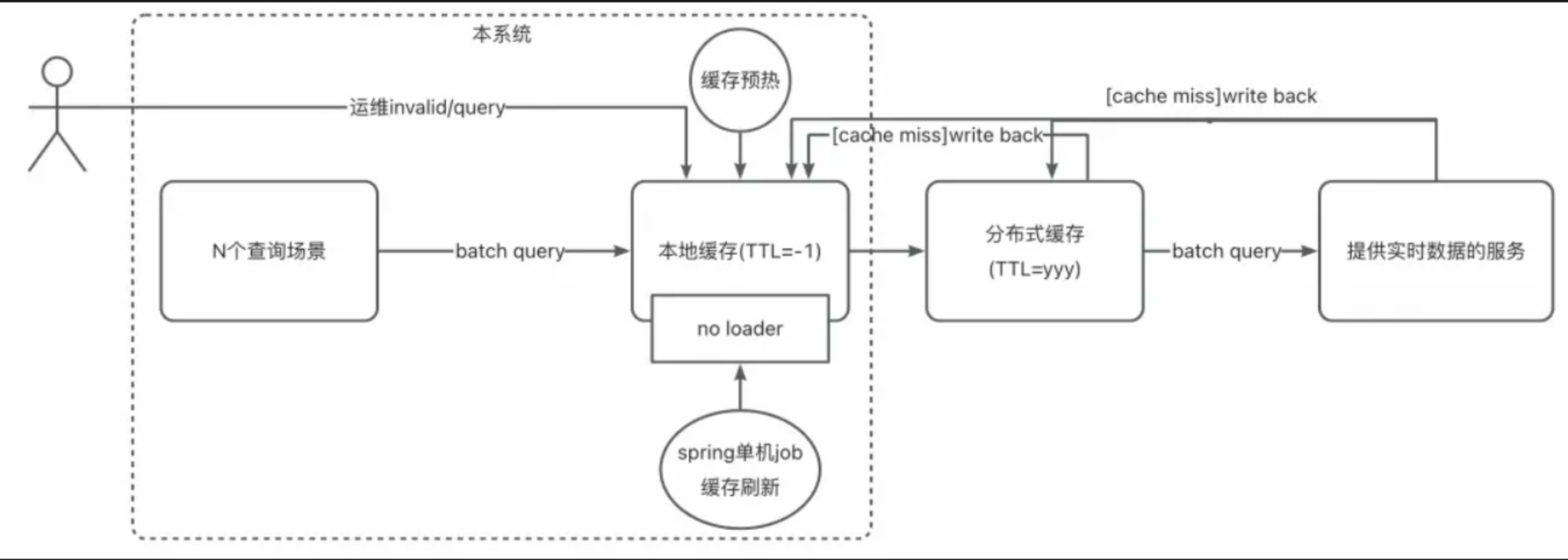
▪ 无redis相关cpu资源、网络资源损耗；

▪ 直接使用了Guava cache原生的loader机制；
- 缺点

▪ 对下游压力翻倍；

由于对下游压力太大，放弃此方案。

方案2：二级缓存+刷新job



1、架构

本地+分布式二级缓存。本地缓存失效时间无穷大。

2、预热

启动时全量预热。

3、cache miss处理

不用guava cache loader。cache miss时业务上调用分布式缓存，再miss则调用下游服务。

4、刷新

使用spring单机job定时全量刷新缓存，保证一定时效性(数据变动频率很低，所以job频率设低即可)。但是由于本系统在间接引入quartz分布式定时任务框架时，没有直接支持单机job(quartz本身是支持的)，所以需要额外使用spring单机job框架，会导致系统任务管理框架不统一。

5、运维

本地缓存：暴露dubbo/http服务，by host运维；或者重启机器。

分布式缓存：通过redis服务管理。

因此:

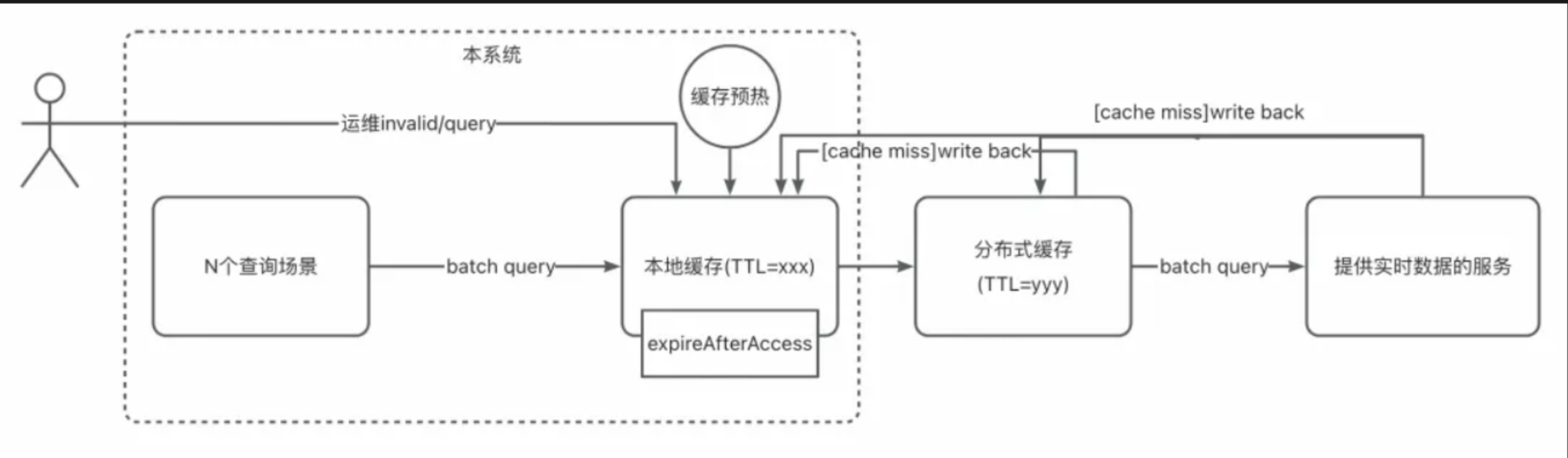
- 优点

▪ 本地缓存命中率很高，基本不会发生redis瓶颈。
- 缺点

▪ 业务自行load+write back。

▪ 使用了新的定时任务框架(for单机)，系统任务管理框架不统一。

方案3：二级缓存+guava expireAfterAccess



1、架构

本地+分布式二级缓存。根据业务设定本地缓存失效时间(expireAfterAccess)。

2、预热

启动时全量预热。这会导致应用重启，然后缓存预热后，部分本地缓存TTL可能批量到期失效，后面请求过来后可能直接大量击穿到下游服务，这是典型的缓存雪崩场景！

而且，当本地缓存雪崩，或者miss时，请求即使hit分布式缓存，也会导致redis相关序列化cpu瓶颈，会导致偶发的系统性能长尾。

3、cache miss处理

不用guava cache loader。cache miss时业务上调用分布式缓存，再miss则调用下游服务。

4、刷新

无刷新job。

5、运维

本地缓存：暴露dubbo/http服务，by host运维；或者重启机器。

分布式缓存：通过redis服务管理。

因此：

- 优点
 - 不需要额外刷新job；
 - 不强依赖运维方案(因为本地和分布式缓存都有失效时间)；
- 缺点
 - 存在缓存雪崩风险；
 - redis相关序列化造成cpu瓶颈发生概率仍然较大；

落地方案

根据上述优缺点、改造量评估，基本按照方案2来执行。

1、架构

本地+分布式二级缓存，以本地缓存为主，保证本地缓存极高命中率。

2、CacheMiss处理

业务自行处理，和现有链路保持一致，localCache -> redis -> dubbo，这样能够快速上线。

3、预热

应用启动，提供服务前预热。

4、刷新

基于SpringCronJob。

5、运维

per host运维，提供dubbo/http服务。


除此之外，对guava cache还进行了包装，各个业务场景的各种local cache，统一存储在一个Map容器Map<\${prefix}, Cache<\${key}, \${value}>>里，对现有的本地缓存也进行了重构，通过缓存包装层统一交互统一管理，增加可复用性和代码简洁度。

写在最后

本文对多种缓存方案的架构、预热、击穿、刷新、运维等进行了比较分析，最终进行工程落地，完成了容器场景高并发、大数据量下的系统性能极致提升。后续的优化点有：

- 1、其他系统瓶颈：压测pattern较单一，可能因为缓存掩盖其他依赖服务的性能瓶颈。未来会用更全面的压测pattern继续压榨系统性能。
- 2、批量刷新能力：对于方案1的批量刷新能力缺陷，caffeine其实有相关feature，见Bulk refresh这个issue。
- 3、运维方案：缺乏集群批量invalid能力，待建设。
- 4、架构统一：分布式和单机job使用同一个job管理框架。

这个方案的落地，特别要感谢团队同学的宝贵建议，是大家多次对技术方案“battle”后的结果。虽然笔者不是哈工大的，但借用哈工大的校训来总结，reviewers的意见、对技术的追求，如同追求“规格严格，功夫到家”。Creating great software is crafting a piece of art. 继续加油！



快速部署定制化文生图应用

PAI Stable Diffusion WebUI 解决方案为企业提供云上快速部署定制化的文生图应用。提供了方便、高效的模型部署产品，并支持根据实际需求，配置不同的服务版本及服务参数。具有分钟级部署上线，方便快捷、开箱即用，多版本部署方案，参数可定制化调整的优势。

[点击阅读原文查看详情。](#)

[阅读原文](#)