缓存这匹“野马”，你驾驭得了吗？- 简书

俗话说得好，工欲善其事，必先利其器，有了好的工具肯定得知道如何用好这些工具，本篇将分为如下几个方面介绍如何利用好缓存：

你真的需要缓存吗

如何选择合适的缓存

多级缓存

缓存更新

缓存挖坑三剑客

缓存污染

序列化

GC 调优

缓存的监控

一款好的框架

总结

你真的需要缓存吗

在使用缓存之前，需要确认你的项目是否真的需要缓存。使用缓存会引入一定的技术复杂度，一般来说从两个方面来判断是否需要使用缓存：

CPU 占用

如果你有某些应用需要消耗大量的 CPU 去计算，比如正则表达式；如果你使用正则表达式比较频繁，而它又占用了很多 CPU 的话，那你就应该使用缓存将正则表达式的结果给缓存下来。

数据库 IO 占用

如果你发现你的数据库连接池比较空闲，可以不用缓存。但是如果数据库连接池比较繁忙，甚至经常报出连接不够的报警，那么是时候应该考虑缓存了。

笔者曾经有个服务被很多其他服务调用，其他时间都还好，但是在每天早上 10 点的时候总是会报出数据库连接池连接不够的报警。

经过排查，我发现有几个服务选择了在 10 点做定时任务，大量的请求打过来，DB 连接池不够，从而产生连接池不够的报警。

这个时候有几个选择，我们可以通过扩容机器来解决，也可以通过增加数据库连接池来解决。

但是没有必要增加这些成本，因为只有在 10 点的时候才会出现这个问题。后来引入了缓存，不仅解决了这个问题，而且还增加了读的性能。

如果并没有上述两个问题，那么你不必为了增加缓存而缓存。

如何选择合适的缓存

缓存分为进程内缓存和分布式缓存。包括笔者在内的很多人在开始选缓存框架的时候都会感到困惑：网上的缓存太多了，大家都吹嘘自己很牛逼，我该怎么选择呢？

选择合适的进程缓存

首先看几个比较常用缓存的比较，具体原理可以参考《你应该知道的缓存进化史》：

比较项	ConcurrentHashMap	LRUMap	Ehcache	Guava Cache	Caffeine
读写性能	很好，分段锁	一般，全局加锁	好	好，需要做淘汰操作	很好
淘汰算法	无	LRU，一般	支持多种淘汰算法，LRU，LFU，FIFO	LRU，一般	W-TinyLFU，很好
功能丰富程度	功能比较简单	功能比较单一	功能很丰富	功能很丰富，支持刷新和虚引用等	功能和Guava Cache类似
工具大小	JDK自带类，很小	基于LinkedHashMap，较小	很大，最新版本1.4MB	是Guava工具类中的一个小部分，较小	一般，最新版本644KB
是否持久化	否	否	是	否	否
是否支持集群	否	否	是	否	否

对于 ConcurrentHashMap 来说，比较适合缓存比较固定不变的元素，且缓存的数量较小的。

虽然从上面表格中比起来有点逊色，但是由于它是 JDK 自带的类，在各种框架中依然有大量的使用。

比如我们可以用来缓存反射的 Method，Field 等等；也可以缓存一些链接，防止重复建立。在 Caffeine 中也是使用的 ConcurrentHashMap 来存储元素。

对于 LRUMap 来说，如果不想引入第三方包，又想使用淘汰算法淘汰数据，可以使用这个。

对于 Ehcache 来说，由于其 jar 包很大，较重量级。对于需要持久化和集群的一些功能的，可以选择 Ehcache。

笔者没怎么使用过这个缓存，如果要选择的话，可以选择分布式缓存来替代 Ehcache。

对于 Guava Cache 来说，Guava 这个 jar 包在很多 Java 应用程序中都有大量的引入。

所以很多时候直接用就好了，并且它本身是轻量级的而且功能较为丰富，在不了解 Caffeine 的情况下可以

选择 Guava Cache。

对于 Caffeine 来说，笔者是非常推荐的，它在命中率，读写性能上都比 Guava Cache 好很多。

并且它的 API 和 Guava Cache 基本一致，甚至会多一点。在真实环境中使用 Caffeine，取得过不错的效果。

总结一下：如果不需要淘汰算法则选择 ConcurrentHashMap；如果需要淘汰算法和一些丰富的 API，这里推荐选择 Caffeine。

选择合适的分布式缓存

这里我选取三个比较出名的分布式缓存来作为比较，MemCache(没有实战使用过)，Redis(在美团又叫 Squirrel)，Tair(在美团又叫 Cellar)。

比较项	MemCache	Squirrel/Redis	Cellar/Tair
数据结构	只支持简单的Key-Value结构	String，Hash，List，Set，Sorted Set	String，HashMap，List，Set
持久化	不支持	支持	支持
容量大小	数据纯内存，数据存储不宜过多	数据全内存，资源成本考量不宜超过100GB	可以配置全内存或内存+磁盘引擎，数据容量可无限扩充
读写性能	很高	很高(RT0.5ms左右)	String类型比较高(RT1ms左右)，复杂类型比较慢(RT5ms左右)

不同的分布式缓存功能特性和实现原理方面有很大的差异，因此它们所适应的场景也有所不同：

MemCache：这一块接触得比较少，不做过多的推荐。其吞吐量较大，但是支持的数据结构较少，并且不支持持久化。

Redis：支持丰富的数据结构，读写性能很高，但是数据全内存，必须要考虑资源成本，支持持久化。

Tair：支持丰富的数据结构，读写性能较高，部分类型比较慢，理论上容量可以无限扩充。

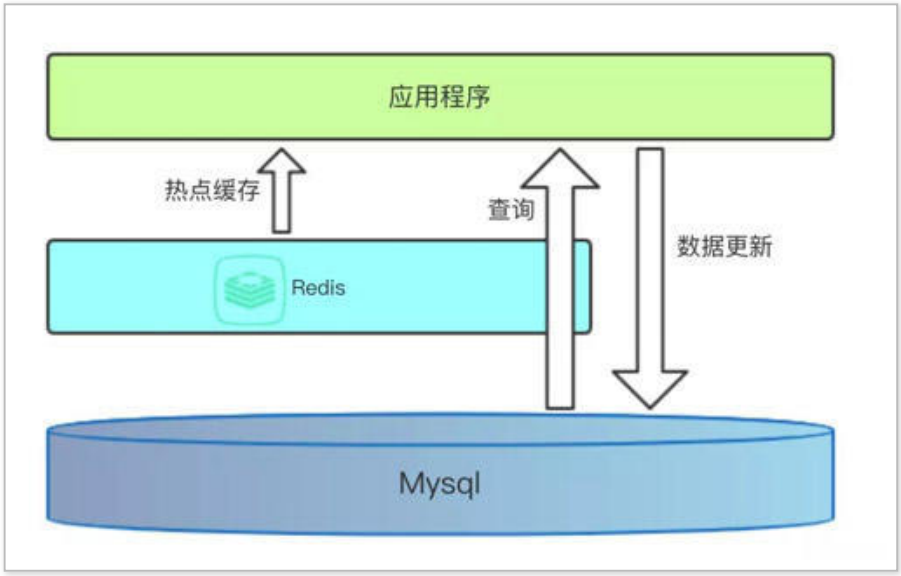
总结：如果服务对延迟比较敏感，Map/Set 数据也比较多的话，比较适合 Redis。

如果服务需要放入缓存量的数据很大，对延迟又不是特别敏感的话，那就可以选择 Tair。

在美团的很多应用中对 Tair 都有应用，在笔者的项目中使用其存放我们生成的支付 Token，支付码，用来替代数据库存储。大部分的情况下两者都可以选择，互为替代。

多级缓存

一说到缓存，很多人脑子里面马上就会出现下面的图：



Redis 用来存储热点数据，Redis 中没有的数据则直接去数据库访问。

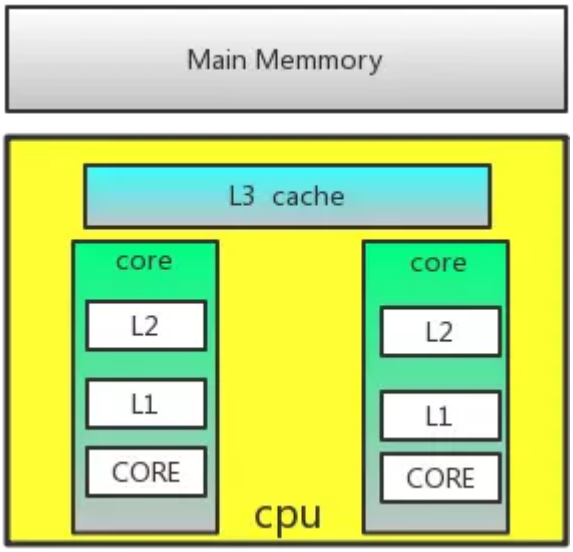
在之前介绍本地缓存的时候，很多人都问我，我已经有 Redis 了，我为什么还需要了解 Guava，Caffeine 这些进程缓存呢？

我统一回复下，有如下两个原因：

Redis 如果挂了或者使用老版本的 Redis，会进行全量同步，此时 Redis 是不可用的，这个时候我们只能访问数据库，很容易造成雪崩。

访问 Redis 会有一定的网络 I/O 以及序列化反序列化，虽然性能很高但是终究没有本地方法快，可以将最热的数据存放在本地，以便进一步加快访问速度。

这个思路并不是我们做互联网架构独有的，在计算机系统中使用 L1，L2，L3 多级缓存，用来减少对内存的直接访问，从而加快访问速度。



所以如果仅仅是使用 Redis，能满足我们大部分需求，但是当需要追求更高性能以及更高可用性的时候，那就不得不了解多级缓存。

使用进程缓存

对于进程内缓存，它本来受限于内存大小的限制，以及进程缓存更新后其他缓存无法得知，所以一般来说进程缓存适用于：

数据量不是很大，数据更新频率较低，之前我们有个查询商家名字的服务，在发送短信的时候需要调用，由于商家名字变更频率较低，并且就算是变更了没有及时变更缓存，短信里面带有老的商家名字客户也能接受。

利用 Caffeine 作为本地缓存，Size 设置为 1 万，过期时间设置为 1 个小时，基本能在高峰期解决问题。

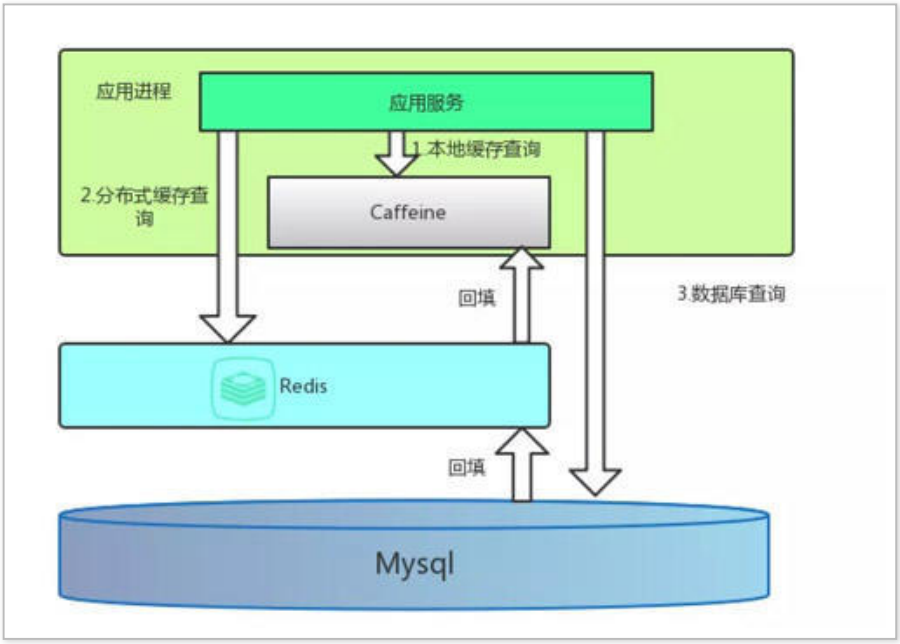
如果数据量更新频繁，也想使用进程缓存的话，那么可以将其过期时间设置为较短，或者设置其较短的自动刷新的时间。这些对于 Caffeine 或者 Guava Cache 来说都是现成的 API。

使用多级缓存

俗话说得好，世界上没有什么是一个缓存解决不了的事，如果有，那就两个。

一般来说我们选择一个进程缓存和一个分布式缓存来搭配做多级缓存，一般来说引入两个也足够了。

如果使用三个，四个的话，技术维护成本会很高，反而有可能会得不偿失，如下图所示：



利用 Caffeine 做一级缓存，Redis 作为二级缓存，步骤如下：

首先去 Caffeine 中查询数据，如果有直接返回。如果没有则进行第 2 步。

再去 Redis 中查询，如果查询到了返回数据并在 Caffeine 中填充此数据。如果没有查到则进行第 3 步。

最后去 MySQL 中查询，如果查询到了返回数据并在 Redis，Caffeine 中依次填充此数据。

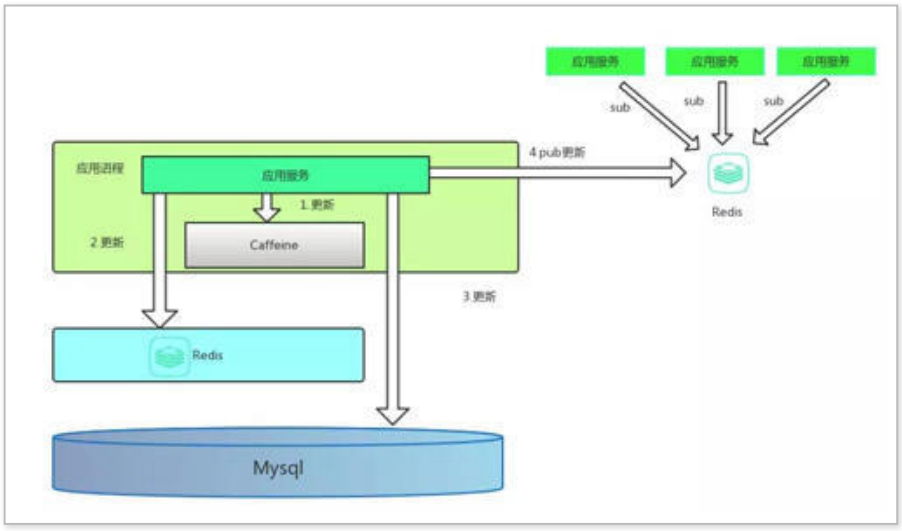
对于 Caffeine 的缓存，如果有数据更新，只能删除更新数据的那台机器上的缓存，其他机器只能通过超时来过期缓存，超时设定可以有两种策略：

设置成写入后多少时间后过期。

设置成写入后多少时间刷新。

对于 Redis 的缓存更新，其他机器立刻可见，但是也必须要设置超时时间，其时间比 Caffeine 的过期长。

为了解决进程内缓存的问题，设计进一步优化：



通过 Redis 的 Pub/Sub，可以通知其他进程缓存对此缓存进行删除。如果 Redis 挂了或者订阅机制不靠谱，依靠超时设定，依然可以做兜底处理。

缓存更新

一般来说缓存的更新有两种情况：

先删除缓存，再更新数据库。

先更新数据库，再删除缓存。

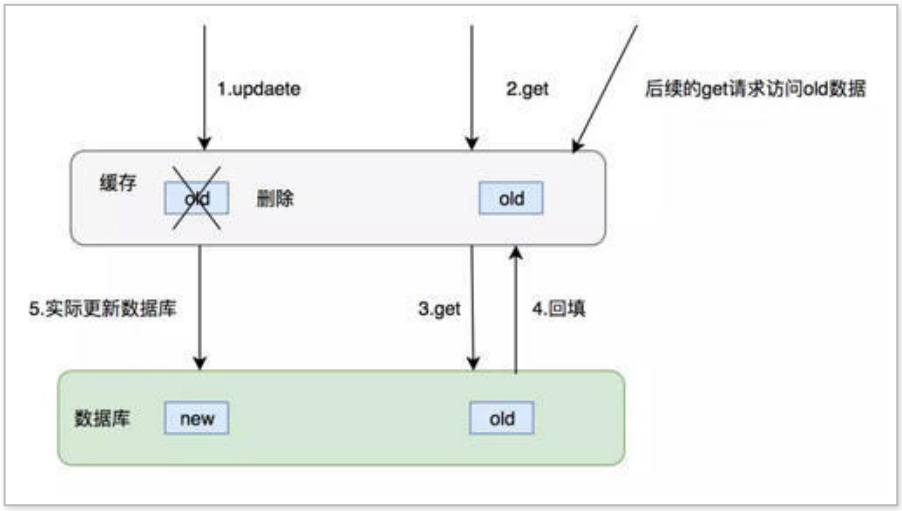
这两种情况在业界，大家都有自己的看法。具体怎么使用还得看各自的取舍。当然肯定有人会问为什么要删除缓存呢？而不是更新缓存呢？

当有多个并发的请求更新数据，你并不能保证更新数据库的顺序和更新缓存的顺序一致，那么就会出现数据库中和缓存中数据不一致的情况。所以一般来说考虑删除缓存。

先删除缓存，再更新数据库

对于一个更新操作简单来说，就是先对各级缓存进行删除，然后更新数据库。

这个操作有一个比较大的问题，在对缓存删除完之后，有一个读请求，这个时候由于缓存被删除所以直接会读库，读操作的数据是老的并且会被加载进入缓存当中，后续读请求全部访问的老数据。



对缓存的操作不论成功失败都不能阻塞我们对数据库的操作，那么很多时候删除缓存可以用异步的操作，但是先删除缓存不能很好的适用于这个场景。

先删除缓存也有一个好处是，如果对数据库操作失败了，那么由于先删除的缓存，最多只是造成 Cache Miss。

先更新数据库，再删除缓存 (推荐)

如果我们使用更新数据库，再删除缓存就能避免上面的问题。但是同样引入了新的问题。

试想一下有一个数据此时是没有缓存的，所以查询请求会直接落库，更新操作在查询请求之后，但是更新操作删除数据库操作在查询完之后回填缓存之前，就会导致我们缓存中和数据库出现缓存不一致。

为什么我们这种情况有问题，很多公司包括 Facebook 还会选择呢？因为要触发这个条件比较苛刻：

首先需要数据不在缓存中。

其次查询操作需要在更新操作先到达数据库。

最后查询操作的回填比更新操作的删除后触发，这个条件基本很难出现，因为更新操作的本来在查询操作之后，一般来说更新操作比查询操作稍慢。但是更新操作的删除却在查询操作之后，所以这个情况比较少出现。

对比上面先删除缓存，再更新数据库的问题来说这种问题出现的概率很低，况且我们有超时机制保底所以基本能满足我们的需求。

如果真的需要追求完美，可以使用二阶段提交，但是成本和收益一般来说不成正比。

当然还有个问题是如果我们删除失败了，缓存的数据就会和数据库的数据不一致，那么我们就只能靠过期超时来进行兜底。

对此我们可以进行优化，如果删除失败的话 我们不能影响主流程那么我们可以将其放入队列后续进行异步删除。

缓存挖坑三剑客

大家一听到缓存有哪些注意事项，首先想到的肯定是缓存穿透，缓存击穿，缓存雪崩这三个挖坑的小能手，这里简单介绍一下他们具体是什么以及应对的方法。

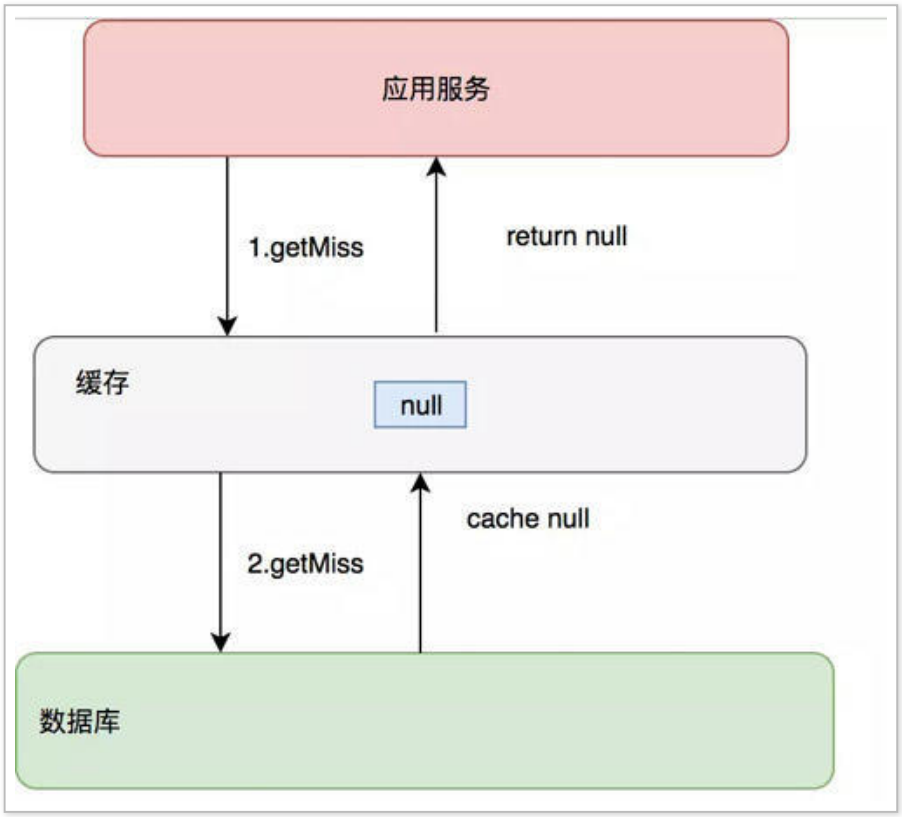
缓存穿透

缓存穿透是指查询的数据在数据库是没有的，那么在缓存中自然也没有，所以在缓存中查不到就会去数据库查询，这样的请求一多，我们数据库的压力自然会增大。

为了避免这个问题，可以采取下面两个手段：

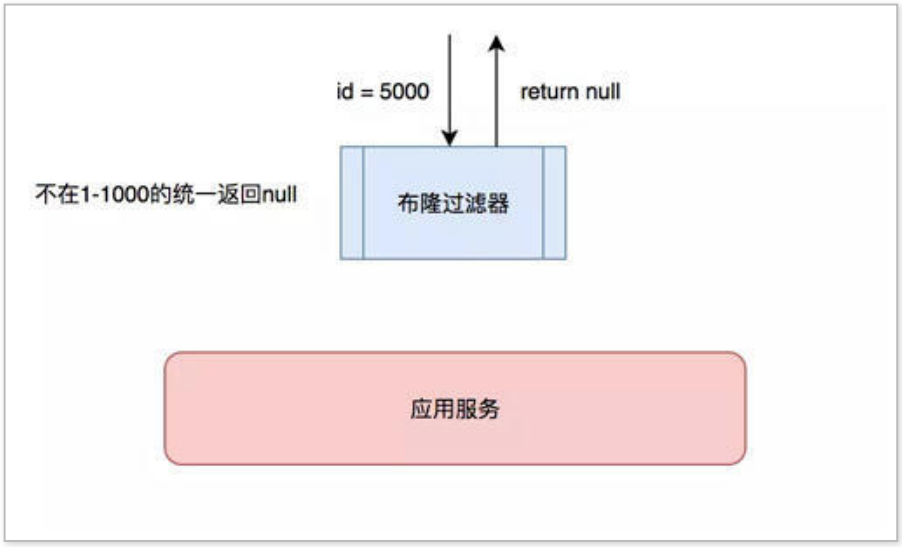
约定：对于返回为 NULL 的依然缓存，对于抛出异常的返回不进行缓存，注意不要把抛异常的也给缓存了。

采用这种手段会增加我们缓存的维护成本，需要在插入缓存的时候删除这个空缓存，当然我们可以通过设置较短的超时时间来解决这个问题。



制定一些规则过滤一些不可能存在的数据，小数据用 BitMap，大数据可以用布隆过滤器。

比如你的订单 ID 明显是在一个范围 1-1000，如果不是 1-1000 之内的数据那其实可以直接给过滤掉。



缓存击穿

对于某些 Key 设置了过期时间，但是它是热点数据，如果某个 Key 失效，可能大量的请求打过来，缓存未命中，然后去数据库访问，此时数据库访问量会急剧增加。

为了避免这个问题，我们可以采取下面的两个手段：

加分布式锁：加载数据的时候可以利用分布式锁锁住这个数据的 Key，在 Redis 中直接使用 SetNX 操作即可。

对于获取到这个锁的线程，查询数据库更新缓存，其他线程采取重试策略，这样数据库不会同时受到很多线程访问同一条数据。

异步加载：由于缓存击穿是热点数据才会出现的问题，可以对这部分热点数据采取到期自动刷新的策略，而不是到期自动淘汰。淘汰也是为了数据的时效性，所以采用自动刷新也可以。

缓存雪崩

缓存雪崩是指缓存不可用或者大量缓存由于超时时间相同在同一时间段失效，大量请求直接访问数据库，数据库压力过大导致系统雪崩。

为了避免这个问题，我们采取下面的手段：

增加缓存系统可用性，通过监控关注缓存的健康程度，根据业务量适当的扩容缓存。

采用多级缓存，不同级别缓存设置的超时时间不同，即使某个级别缓存都过期，也有其他级别缓存兜底。

缓存的 Key 值可以取个随机值，比如以前是设置 10 分钟的超时时间，那每个 Key 都可以随机 8-13 分钟过期，尽量让不同 Key 的过期时间不同。

缓存污染

缓存污染一般出现在我们使用本地缓存中。可以想象，在本地缓存中如果你获得了缓存，但是你接下来修改了这个数据，这个数据却并没有更新在数据库，这样就造成了缓存污染：

```
ConcurrentHashMap<Integer, Customer> cache = new ConcurrentHashMap();
    public Customer getCustomer(int id){
        Customer customer = cache.get(id);
        //需求是需要给Customer中的名字加上店名
        customer.setName(customer.getName() + "_" + tenantName);
        return customer;
    }
```

上面的代码就造成了缓存污染，通过 ID 获取 Customer，但是需求需要修改 Customer 的名字。

所以开发人员直接在取出来的对象中直接修改，这个 Customer 对象就会被污染，其他线程取出这个数据就是错误的数据。

要想避免这个问题需要开发人员从编码上注意，并且代码必须经过严格的 Review，以及全方位的回归测试，才能从一定程度上解决这个问题。

序列化

序列化是很多人都不注意的一个问题，很多人忽略了序列化的问题，上线之后马上报出一下奇怪的错误异常，造成了不必要的损失，最后一排查都是序列化的问题。

列举几个序列化常见的问题：

Key-Value 对象过于复杂导致序列化不支持：笔者之前出过一个问题，在美团的 Tair 内部默认是使用 protostuff 进行序列化。

而美团使用的通讯框架是 thfift，thrift 的 TO 是自动生成的，这个 TO 里面有很多复杂的数据结构，但是将它存放到了 Tair 中。

查询的时候反序列化也没有报错，单测也通过，但是到 QA 测试的时候发现这一块功能有问题，有个字段是 boolean 类型默认是 False，把它改成 true 之后，序列化到 Tair 中再反序列化还是 False。

定位到是 protostuff 对于复杂结构的对象 (比如数组，List 等等) 支持不是很好，会造成一定的问题。

后来对这个 TO 进行了转换，用普通的 Java 对象就能进行正确的序列化反序列化。

添加了字段或者删除了字段，导致上线之后老的缓存获取的时候反序列化报错，或者出现一些数据移位。

不同的 JVM 的序列化不同，如果你的缓存有不同的服务都在共同使用 (不提倡)，那么需要注意不同 JVM 可能会对 Class 内部的 Field 排序不同，而影响序列化。

比如 (举例，实际情况不一定如此) 下面的代码，在 JDK7 和 JDK8 中对象 A 的排列顺序不同，最终会导致反序列化结果出现问题：

//jdk7class A{inta;intb;} //jdk8class A{intb;inta; } 复制代码

序列化的问题必须得到重视，解决的办法有如下几点：

测试：对于序列化需要进行全面的测试，如果有不同的服务并且他们的 JVM 不同，那么你也需要做这一块的测试。

在上面的问题中笔者的单测通过的原因是用的默认数据 False，所以根本没有测试 true 的情况，还好 QA 给力，将它给测试出来了。

对于不同的序列化框架都有自己不同的原理，对于添加字段之后如果当前序列化框架不能兼容老的，那么可以换个序列化框架。

对于 protostuff 来说它是按照 Field 的顺序来进行反序列化的，对于添加字段我们需要放到末尾，也就是不能插在中间，否则会出现错误。

对于删除字段来说，用 @Deprecated 注解进行标注弃用，如果贸然删除，除非是最后一个字段，否则肯定会出现序列化异常。

{} 如果你想了解更多，可以加群：855355016，群内有 Java 高架构师、分布式架构、高可扩展、高性能、

高并发、性能优化、Spring boot、Redis、ActiveMQ、Nginx、Mycat、Netty、Jvm 大型分布式项目实战学习架构师视频免费获取} 可以使用双写来避免，对于每个缓存的 Key 值可以加上版本号，每次上线版本号都加 1。

比如现在线上的缓存用的是 Key_1，即将要上线的是 Key_2，上线之后对缓存的添加是会写新老两个不同的版本 (Key_1，Key_2) 的 Key-Value，读取数据还是读取老版本 Key_1 的数据。

假设之前的缓存的过期时间是半个小时，那么上线半个小时之后，之前的老缓存存量的数据都会被淘汰，此时线上老缓存和新缓存的数据基本是一样的，切换读操作到新缓存，然后停止双写。

采用这种方法基本能平滑过渡新老 Model 交替，但是不好的就是需要短暂的维护两套新老 Model，下次上线的时候需要删除掉老 Model，这样增加了维护成本。

GC 调优

对于大量使用本地缓存的应用，由于涉及到缓存淘汰，那么 GC 问题必定是常事。如果出现 GC 较多，STW 时间较长，那么必定会影响服务可用性。

这一块给出下面几点建议：

经常查看 GC 监控，如何发现不正常，需要想办法对其进行优化。

对于 CMS 垃圾收集算法，如果发现 Remark 过长，如果是大量本地缓存应用的话这个过长应该很正常，因为在并发阶段很容易有很多新对象进入缓存，从而 Remark 阶段扫描很耗时，Remark 又会暂停。

可以开启 XX：CMSScavengeBeforeRemark，在 Remark 阶段前进行一次 YGC，从而减少 Remark 阶段扫描 GC Root 的开销。

可以使用 G1 垃圾收集算法，通过 XX：MaxGCPauseMillis 设置最大停顿时间，提高服务可用性。

缓存的监控

很多人对于缓存的监控也比较忽略，基本上线之后如果不报错，然后就默认它就生效了。

但是存在这个问题，很多人由于经验不足，有可能设置了不恰当的过期时间，或者不恰当的缓存大小导致缓存命中率不高，让缓存成为了代码中的一个装饰品。

所以对于缓存各种指标的监控，也比较重要，通过不同的指标数据，我们可以对缓存的参数进行优化，从而让缓存达到最优化：

上面的代码中用来记录 Get 操作的，通过 Cat 记录了获取缓存成功，缓存不存在，缓存过期，缓存失败(获取缓存时如果抛出异常，则叫失败)。

通过这些指标，我们就能统计出命中率，我们调整过期时间和大小的时候就可以参考这些指标进行优化。

```
private void parseSingleGet(CacheGetResult result) {
        LOGGER.debug("getResult code:{}, message:{}", result.getResultCode(), result.getMessage());
        switch (result.getResultCode()) {
            //记录成功
            case SUCCESS:
                CatMonitorBase.monitorMetricForCountByTags(name, GETSUCCESS_TAGS);
                break;
            //记录不存在
            case NOT_EXISTS:
                CatMonitorBase.monitorMetricForCountByTags(name, GETMISSCOUNT_TAGS);
                break;
            //记录过期
            case EXPIRED:
                CatMonitorBase.monitorMetricForCountByTags(name, GETEXPIRECOUNT_TAGS);
                break;
            //记录失败
            case FAIL:
                CatMonitorBase.monitorMetricForCountByTags(name, GETFAILCOUNT_TAGS);
                break;
            default:
                LOGGER.warn("springCache get return unexpected code:{},name:{} ", result.getResultCode(),
this.name);
        }
    }
```

一款好的框架

一个好的剑客没有一把好剑怎么行呢？如果要使用好缓存，一个好的框架也必不可少。

在最开始使用的时候，大家使用缓存都用一些 util，把缓存的逻辑写在业务逻辑中：

```
public Customer getCustomer(int id){
        //通过Redis的Util获取这个key，然后把value转换成Customer
        Customer customer = ConvertorUitl.convert(JedisUtil.get(id));
        if (customer == null){
            //没有缓存落库查询
            customer = customerMapper.getById(id);
            JedisUtil.set(id, customer);
        }
        return customer;
    }
```

上面的代码把缓存的逻辑耦合在业务逻辑当中，如果我们要增加成多级缓存那就需要修改我们的业务逻辑，不符合开闭原则，所以引入一个好的框架是不错的选择。

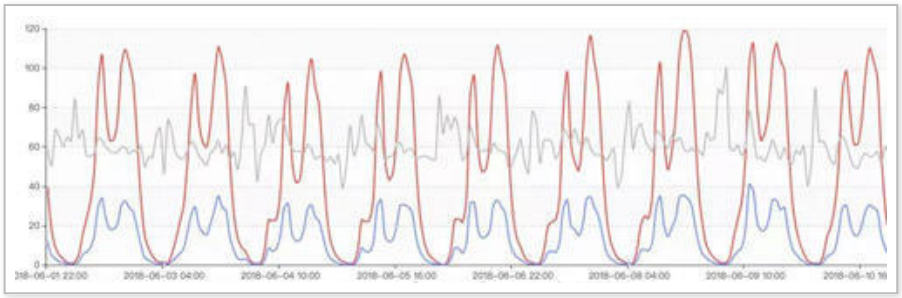
推荐大家使用 JetCache 这款开源框架，它实现了 Java 缓存规范 JSR107 并且支持自动刷新等高级功能。

笔者参考 JetCache 结合 Spring Cache，监控框架 Cat 以及美团的熔断限流框架 Rhino 实现了一套自有的缓存框架，让操作缓存，打点监控，熔断降级，业务人员无需关心。

上面的代码可以优化成：

```
@Cacheable(value = "tairCache", key = "#id" ,unless = "#result == null")
    public Customer getCustomer(int id){
        return customerMapper.getById(id);
    }
```

对于一些监控数据也能轻松从大盘上看到：



总结

想要真正的使用好一个缓存，必须要掌握很多的知识，并不是看几个 Redis 原理分析，就能把 Redis 缓存用得炉火纯青。

对于不同场景，缓存有各自不同的用法，同样的不同的缓存也有自己的调优策略，进程内缓存你需要关注的是它的淘汰算法和 GC 调优，以及要避免缓存污染等。

分布式缓存你需要关注的是它的高可用，如果它不可用了，如何进行降级，以及一些序列化的问题。

全文完