美团点评技术团队

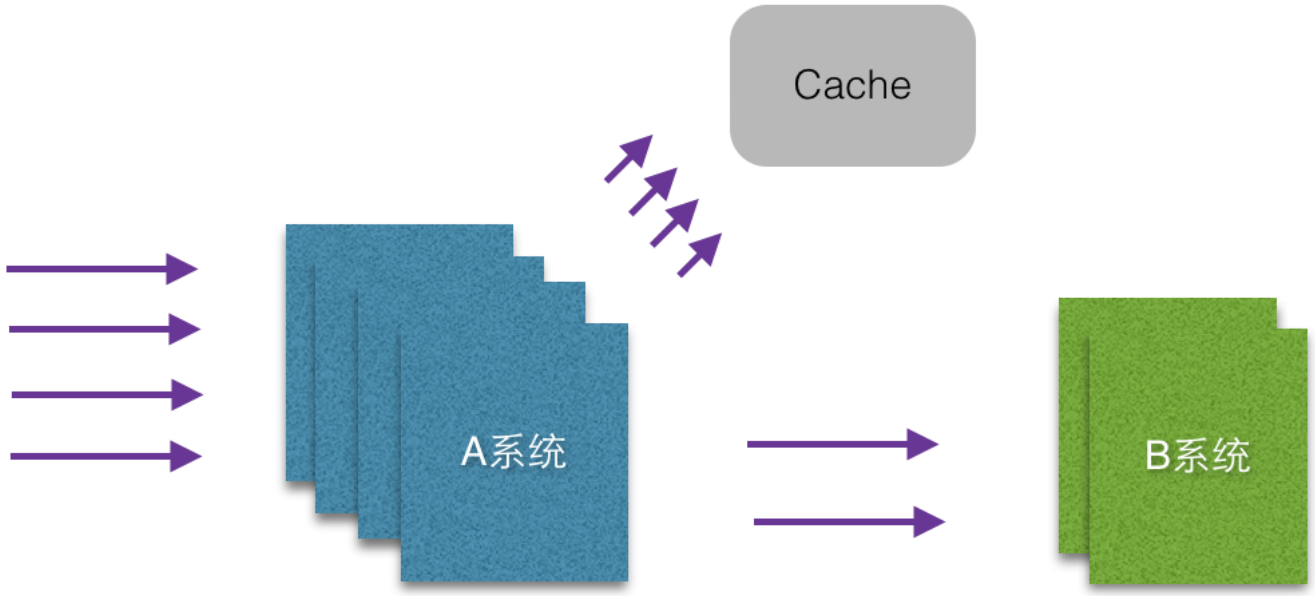
Cache应用中的服务过载案例研究

张杨 ·2016-06-16 17:00

简单地说，过载是外部请求对系统的访问量突然激增，造成请求堆积，服务不可用，最终导致系统崩溃。本文主要分析引入Cache可能造成的服务过载，并讨论相关的预防、恢复策略。Cache在现代系统中使用广泛，由此引入的服务过载隐患无处不在，但却非常隐蔽，容易被忽视。本文希望能为开发者在设计和编写相关类型应用，以及服务过载发生处理时能够有章可循。

一个服务过载案例

本文讨论的案例是指存在正常调用关系的两个系统（假设调用方为A系统，服务方为B系统），A系统对B系统的访问突然超出B系统的承受能力，造成B系统崩溃。造成服务过载的原因很多，这里分析的是严重依赖Cache的系统服务过载。首先来看一种包含Cache的体系结构（如下图所示）。



A系统依赖B系统的读服务，A系统是60台机器组成的集群，B系统是6台机器组成的集群，之所以6台机器能够扛住60台机器的访问，是因为A系统并不是每次都访问B，而是首先请求Cache，只有Cache的相应数据失效时才会请求B。

这正是Cache存在的意义，它让B系统节省了大量机器；如果没有Cache，B系统不得不组成60台机器的集群，如果A也同时依赖除B系统外的另一个系统（假设为C系统）呢？那么C系统也要60台机器，放大的流量将很快耗尽公司的资源。

然而Cache的引入也不是十全十美的，这个结构中如果Cache发生问题，全部的流量将流向依赖方，造成流量激增，从而引发依赖系统的过载。

回到A和B的架构，造成服务过载的原因至少有下面三种：

1. B系统的前置代理发生故障或者其他原因造成B系统暂时不可用，等B系统系统服务恢复时，其流量将远远超过正常值。
2. Cache系统故障，A系统的流量将全部流到B系统，造成B系统过载。
3. Cache故障恢复，但这时Cache为空，Cache瞬间命中率为0，相当于Cache被击穿，造成B系统过载。

第一个原因不太好理解，为什么B系统恢复后流量会猛增呢？主要原因就是缓存的超时时间。当有数据超时的时候，A系统会访问B系统，但是这时候B系统偏偏故障不可用，那么这个数据只好超时，等发现B系统恢复时，发现缓存里的B系统数据已经都超时了，都成了旧数据，这时当然所有的请求就打到了B。

下文主要介绍服务过载的预防和发生后的一些补救方法，以预防为主，从调用方和服务方的视角阐述一些可行方案。

服务过载的预防

所谓Client端指的就是上文结构中的A系统，相对于B系统，A系统就是B系统的Client，B系统相当于Server。

Client端的方案

针对上文阐述的造成服务过载的三个原因：B系统故障恢复、Cache故障、Cache故障恢复，我们看看A系统有哪些方案可以应对。

合理使用Cache应对B系统宕机

一般情况下，Cache的每个Key除了对应Value，还对应一个过期时间T，在T内，get操作直接在Cache中拿到Key对应Value并返回。但是在T到达时，get操作主要有五种模式：

1. 基于超时的简单（stupid）模式

在T到达后，任何线程get操作发现Cache中的Key和对应Value将被清除或标记为不可用，get操作将发起调用远程服务获取Key对应的Value，并更新写回Cache，然后get操作返回新值；如果远程获取Key-Value失败，则get抛出异常。

为了便于理解，举一个码头工人取货的例子：5个工人（线程）去港口取同样Key的货（get），发现货已经过期被扔掉了，这时5个工人各自分别去对岸取新货，然后返回。

2. 基于超时的常规模式

在T到达后，Cache中的Key和对应Value将被清除或标记为不可用，get操作将调用远程服务获取Key对应的Value，并更新写回Cache；此时，如果另一个线程发现Key和Value已经不可用，**get操作还需要判断有没有其他线程发起了远程调用**，如果有，那么自己就等待，直到那个线程远程获取操作成功，Cache中得Key变得可用，get操作返回新的Value。如果远程获取操作失败，则get操作抛出异常，不会返回任何Value。

还是码头工人的例子：5个工人（线程）去港口取同样Key的货（get），发现货已经过期被扔掉了，那么只需派出一个人去对岸取货，其他四个人在港口等待即可，而不用5个人全去。

基于超时的简单模式和常规模式区别在于对于同一个超时的Key，前者每个get线程一旦发现Key不存在，则发起远程调用获取值；而后者每个get线程发现Key不存在，则还要判断当前是否有其他线程已经发起了远程调用操作获取新值，如果有，自己就简单的等待即可。

显然基于超时的常规模式比基于超时的简单模式更加优化，减少了超时时并发访问后端的调用量。

实现基于超时的常规模式就需要用到经典的Double-checked locking (https://en.wikipedia.org/wiki/Double-checked_locking)惯用法了。

3. 基于刷新的简单（stupid）模式

在T到达后，Cache中的Key和相应Value不动，但是如果有线程调用get操作，将触发refresh操作，根据get和refresh的同步关系，又分为两种模式：

- 同步模式：任何线程发现Key过期，都触发一次refresh操作，get操作等待refresh操作结束，refresh结束后，get操作返回当前Cache中Key对应的Value。注意refresh操作结束并不意味着refresh成功，还可能抛了异常，没有更新Cache，但是get操作不管，get操作返回的值可能是旧值。
- 异步模式：任何线程发现Key过期，都触发一次refresh操作，get操作触发refresh操作，不等refresh完成，直接返回Cache中的旧值。

举上面码头工人的例子说明基于刷新的常规模式：这次还是5工人去港口取货，这时货都在，但

是已经旧了，这时5个工人有两种选择：

- 5个人各自去远程取新货，如果取货失败，则拿着旧货返回（同步模式）
- 5个人各自通知5个雇佣工去取新货，5个工人拿着旧货先回（异步模式）

4. 基于刷新的常规模式

在T到达后，Cache中的Key和相应Value都不会被清除，而是被标记为旧数据，如果有线程调用get操作，将触发refresh更新操作，根据get和refresh的同步关系，又分为两种模式：

- 同步模式：get操作等待refresh操作结束，refresh结束后，get操作返回当前Cache中Key对应的Value，注意：refresh操作结束并不意味着refresh成功，还可能抛了异常，没有更新Cache，但是get操作不管，get操作返回的值可能是旧值。如果其他线程进行get操作，Key已经过期，**并且发现有线程触发了refresh操作**，则自己不等refresh完成直接返回旧值。
- 异步模式：get操作触发refresh操作，不等refresh完成，直接返回Cache中的旧值。如果其他线程进行get操作，发现Key已经过期，**并且发现有线程触发了refresh操作**，则自己不等refresh完成直接返回旧值。

再举上面码头工人的例子说明基于刷新的常规模式：这次还是5工人去港口取货，这时货都在，但是已经旧了，这时5个工人有两种选择：

- 派一个人去远方港口取新货，其余4个人拿着旧货先回（同步模式）。
- 5个人通知一个雇佣工去远方取新货，5个人都拿着旧货先回（异步模式）。

基于刷新的简单模式和基于刷新的常规模式区别就在于取数线程之间能否感知当前数据是否正处在刷新状态，因为基于刷新的简单模式中取数线程无法感知当前过期数据是否正处在刷新状态，所以每个取数线程都会触发一个刷新操作，造成一定的线程资源浪费。

而基于超时的常规模式和基于刷新的常规模式区别在于前者过期数据将不能对外访问，所以一旦数据过期，各线程要么拿到数据，要么抛出异常；后者过期数据可以对外访问，所以一旦数据过期，各线程要么拿到新数据，要么拿到旧数据。

5. 基于刷新的续费模式

该模式和基于刷新的常规模式唯一的区别在于refresh操作超时或失败的处理上。在基于刷新的常规模式中，refresh操作超时或失败时抛出异常，Cache中的相应Key-Value还是旧值，这样下一个get操作到来时又会触发一次refresh操作。

在基于刷新的续费模式中，如果refresh操作失败，那么refresh将把旧值当成新值返回，这样就相当于旧值又被续费了T时间，后续T时间内get操作将取到这个续费的旧值而不会触发refresh操作。

基于刷新的续费模式也像常规模式那样分为同步模式和异步模式，不再赘述。

下面讨论这5种Cache get模式在服务过载发生时的表现，首先假设如下：

- 假设A系统的访问量为每分钟M次。
- 假设Cache能存Key为C个，并且Key空间有N个。
- 假设正常状态下，B系统访问量为每分钟W次，显然W\<\<M。

这时因为某种原因，比如B长时间故障，造成Cache中得Key全部过期，B系统这时从故障中恢复，五种get模式分析表现分析如下：

1. 在基于超时和刷新的简单模式中，B系统的瞬间流量将达到和A的瞬时流量M大体等同，相当于Cache被击穿。这就发生了服务过载，这时刚刚恢复的B系统将肯定会被大流量压垮。
2. 在基于超时和刷新的常规模式中，B系统的瞬间流量将和Cache中Key空间N大体等同。这时是否发生服务过载，就要看Key空间N是否超过B系统的流量上限了。
3. 在基于刷新的续费模式中，B系统的瞬间流量为W，和正常情况相同而不会发生服务过载。实际上，在基于刷新的续费模式中，不存在Cache Key全部过期的情况，就算把B系统永久性地干掉，A系统的Cache也会基于旧值长久的平稳运行。

第3点，B系统不会发生服务过载的主要原因是基于刷新的续费模式下不会出现chache中的Key全部长时间过期的情况，即使B系统长时间不可用，基于刷新的续费模式也会在一个过期周期内把旧值当成新值继续使用。所以当B系统恢复时，A系统的Cache都处在正常工作状态。

从B系统的角度看，能够抵抗服务过载的基于刷新的续费模式最优。

从A系统的角度看，由于一般情况下A系统是一个高访问量的在线web应用，这种应用最讨厌的一个词就是“线程等待”，因此基于刷新的各种异步模式较优。

综合考虑，**基于刷新的异步续费模式是首选。**

然而凡是有利就有弊，有两点需要注意的地方：

1. 基于刷新模式最大的缺点是Key-Value一旦放入Cache就不会被清除，每次更新也是新值覆盖旧值，JVM GC永远无法对其进行垃圾收集，而基于超时的模式中，Key-Value超时后如果新的访问没有到来，内存是可以被GC垃圾回收的。所以如果你使用的是寸土寸金的本地内存做Cache就要小心了。
2. 基于刷新的续费模式需要做好监控，不然有可能Cache中的值已经和真实的值相差很远了，应用还以为是新值而使用。

关于具体的Cache，来自Google的Guava本地缓存库支持上文的第二种、第四种和第五种get操作模式。

但是对于Redis等分布式缓存，只提供原始的get、set方法，而提供的get仅仅是获取，与上文提到的五种get操作模式不是一个概念。开发者想用这五种get操作模式的话不得不自己封装和实现。

五种get操作模式中，基于超时和刷新的简单模式是实现起来最简单的模式，但遗憾的是这两种模式对服务过载完全无免疫力，这可能也是服务过载在大量依赖缓存的系统中频繁发生的一个重要原因吧。

本文之所以把第1、3种模式称为stupid模式，是想强调这种模式应该尽量避免，Guava里面根本没有这种模式，而Redis只提供简单的读写操作，很容易就把系统实现成了这种方式。

应对分布式Cache宕机

如果是Cache直接挂了，那么就算是基于刷新的异步续费模式也无能为力了。这时A系统铁定无法对Cache进行存取操作，只能将流量完全打到B系统，B系统面对服务过载在劫难逃……

本节讨论的预防Cache宕机仅限于分布式Cache，因为本地Cache一般和A系统应用共享内存和进程，本地Cache挂了A系统也挂了，不会出现本地Cache挂了而A系统应用正常的情况。

首先，A系统请求线程检查分布式Cache状态，如果无应答则说明分布式Cache挂了，则转向请求B系统，这样一来大流量将压垮B系统。这时可选的方案如下：

1. A系统的当前线程不请求B系统，而是打个日志并设置一个默认值。
2. A系统的当前线程按照一定概率决定是否请求B系统。
3. A系统的当前线程检查B系统运行情况，如果良好则请求B系统。

方案1最简单，A系统知道如果没有Cache，B系统可能扛不住自己的全部流量，索性不请求B系统，等待Cache恢复。但这时B系统利用率为0，显然不是最优方案，而且当请求的Value不容易设置默认值时，这个方案就不行了。

方案2可以让一部分线程请求B系统，这部分请求肯定能被B系统hold住。可以保守的设置这个概率 u =（B系统的平均流量）/（A系统的峰值流量）

方案3是一种更为智能的方案，如果B系统运行良好，当前线程请求；如果B系统过载，则不请求，这样A系统将让B系统处于一种宕机与不宕机的临界状态，最大限度挖掘B系统性能。这种方案要求B系统提供一个性能评估接口返回Yes和No，Yes表示B系统良好，可以请求；No表示B系统情况不妙，不要请求。这个接口将被频繁调用，必须高效。

方案3的关键在于如何评估一个系统的运行状况。一个系统中当前主机的性能参数有CPU负载、内存使用率、Swap使用率、GC频率和GC时间、各个接口平均响应时间等，性能评估接口需要根据这些参数返回Yes或者No，是不是机器学习里的二分类问题？😁关于这个问题已经可以单独写篇文章讨论了，在这里就不展开了，你可以想一个比较简单傻瓜的保守策略，缺点是A系统的请求无法很好的逼近B系统的性能极限。

综合以上分析，方案2比较靠谱。如果选择方案3，建议由专门团队负责研究并提供统一的系统性能实时评估方案和工具。

应对分布式Cache宕机后的恢复

不要以为成功hold住分布式Cache宕机就万事大吉了，真正的考验是分布式Cache从宕机过程恢复之后，这时分布式Cache中什么都没有。

即使是上文中提到了基于刷新的异步续费策略这时也没用，因为分布式Cache为空，无论如何都要请求B系统。这时B系统的最大流量是Key的空间取值数量。

如果Key的取值空间数量很少，则相安无事；如果Key的取值空间数量大于B系统的流量上限，服务过载依然在所难免。

这种情况A系统很难处理，关键原因是**A系统请求Cache返回Key对应Value为空，A系统无法知道是因为当前Cache是刚刚初始化，所有内容都为空；还是因为仅仅是自己请求的那个Key没在Cache里。**

如果是前者，那么当前线程就要像处理Cache宕机那样进行某种策略的回避；如果是后者，直接请求B系统即可，因为这是正常的Cache使用流程。

对于Cache宕机的恢复，A系统真的无能为力，只能寄希望于B系统的方案了。

Server端的方案

相对于Client端需要应对各种复杂问题，Server端需要应对的问题非常简单，就是如何从容应对过载的问题。无论是缓存击穿也好，还是拒绝服务攻击也罢，对于Server端来说都是过载保护的问题。对于过载保护，主要给出两种可行方案，以及一种比较复杂的方案思路。

流量控制

流量控制就是B系统实时监控当前流量，如果超过预设的值或者系统承受能力，则直接拒绝掉一部分请求，以实现对系统的保护。

流量控制根据基于的数据不同，可分为两种：

1. 基于流量阈值的流控：流量阈值是每个主机的流量上限，流量超过该阈值主机将进入不稳定状态。阈值提前进行设定，如果主机当前流量超过阈值，则拒绝掉一部分流量，使得实际被处理流量始终低于阈值。
2. 基于主机状态的流控：每个接受每个请求之前先判断当前主机状态，如果主机状况不佳，则拒绝当前请求。

基于阈值的流控实现简单，但是最大的问题是需要提前设置阈值，而且随着业务逻辑越来越复杂，接口越来越多，主机的服务能力实际应该是下降的，这样就需要不断下调阈值，增加了维护成本，而且万一忘记调整的话，呵呵……

主机的阈值可以通过压力测试确定，选择的时候可以保守些。

基于主机状态的流控免去了人为控制，但是其最大的确定上文已经提到：如何根据当前主机各个参数判断主机状态呢？想要完美的回答这个问题目测并不容易，因此在没有太好答案之前，我推荐基于阈值的流控。

流量控制基于实现位置的不同，又可以分为两种：

1. 反向代理实现流控：在反向代理如Nginx上基于各种策略进行流量控制。这种一般针对HTTP服务。
2. 借助服务治理系统：如果Server端是RMI、RPC等服务，可以构建专门的服务治理系统进行负载均衡、流控等服务。
3. 服务容器实现流控：在应用代码里，业务逻辑之前实现流量控制。

第3种在服务器的容器（如Java容器）中实现流控并不推荐，因为流控和业务代码混在一起容易混乱；其次实际上流量已经全量进入到了业务代码里，这时的流控只是阻止其进入真正的业务逻辑，所以流控效果将打折；还有，如果流量策略经常变动，系统将不得不为此经常更改。

因此，推荐前两种方式。

最后提一个注意点：当因为流控而拒绝请求时，务必在返回的数据中带上相关信息（比如“当前请求因为超出流量而被禁止访问”），如果返回值什么都没有将是一个大坑。因为造成调用方请求没有被响应的原因很多，可能是调用方Bug，也可能是服务方Bug，还可能是网络不稳定，这样一来很可能在排查一整天后发现是流控搞的鬼……

服务降级

服务降级一般由人为触发，属于服务过载造成崩溃恢复时的策略，但为了和流控对比，将其放到这里。

流量控制本质上是减小访问量，而服务处理能力不变；而服务降级本质上是降低了部分服务的处理能力，增强另一部分服务处理能力，而访问量不变。

服务降级是指在服务过载时关闭不重要的接口（直接拒绝处理请求），而保留重要的接口。比如服务由10个接口，服务降级时关闭了其中五个，保留五个，这时这个主机的服务处理能力将增强到二倍左右。

然而，服务过载发生时动辄就超出系统处理能力10倍，而服务降级能使主机服务处理能力提高10倍么？显然很困难，因此服务过载的应对不能只依靠服务降级策略。

动态扩展

动态扩展指的是在流量超过系统服务能力时，自动触发集群扩容，自动部署并上线运行；当流量过去后又自动回收多余机器，完全弹性。

这个方案是不是感觉很不错。但是目前互联网公司的在线应用跑在云上的本身就不多，要完全实现在线应用的自动化弹性运维，要走的路就更多了。

崩溃恢复

如果服务过载造成系统崩溃还是不幸发生了，这时需要运维控制流量，等后台系统启动完毕后循序渐进的放开流量，主要目的是让Cache慢慢预热。流量控制刚开始可以为10%，然后20%，然后50%，然后80%，最后全量，当然具体的比例，尤其是初始比例，还要看后端承受能力和前端流量的比例，各个系统并不相同。

如果后端系统有专门的工具进行Cache预热，则省去了运维的工作，等Cache热起来再发布后台系统即可。但是如果Cache中的Key空间很大，开发预热工具将比较困难。

结论

“防患于未然”放在服务过载的应对上也是适合的，预防为主，补救为辅。综合上文分析，具体的预防要点如下：

1. 调用方（A系统）采用基于刷新的异步续费模式使用Cache，或者至少不能使用基于超时或刷新的简单（stupid）模式。
2. 调用方（A系统）每次请求Cache时检查Cache是否可用（available），如果不可用则按照一个保守的概率访问后端，而不是无所顾忌的直接访问后端。
3. 服务方（B系统）在反向代理处设置流量控制进行过载保护，阈值需要通过压测获得。

崩溃的补救主要还是靠运维和研发在发生时的通力合作：观察流量变化准确定位崩溃原因，运维控流量研发持续关注性能变化。

未来如果有条件的话可以研究下主机应用健康判断问题和动态弹性运维问题，毕竟自动化比人为操作要靠谱。

服务过载 Cache 并发 可用性 SOA