(/book/4900)

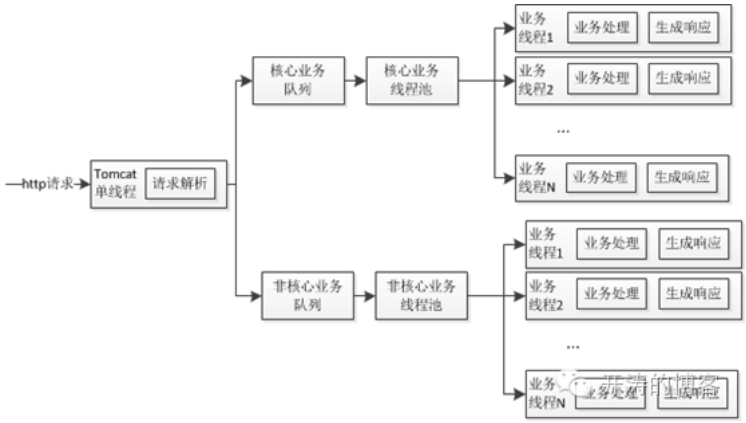
聊聊高并发之隔离术

张开涛 (/space/index/29518)
2017-04-21

隔离是指将系统或资源分割开，系统隔离是为了在系统发生故障时能限定传播范围和影响范围，即发生故障后不会出现滚雪球效应，从而保证只有出问题的服务不可用，其他服务还是可用的；而资源隔离有脏数据隔离、通过隔离后减少资源竞争提升性能等。我遇到的比较多的隔离手段有线程隔离、进程隔离、集群隔离、机房隔离、读写隔离、动静隔离、爬虫隔离等。而出现系统问题时可以考虑负载均衡路由、自动/手动切换分组或者降级等手段来提升可用性。

线程隔离

线程隔离主要有线程池隔离，在实际使用时我们会把请求分类，然后交给不同的线程池处理，当一种业务的请求处理发生问题时，不会将故障扩散到其他线程池，从而保证其他服务可用。

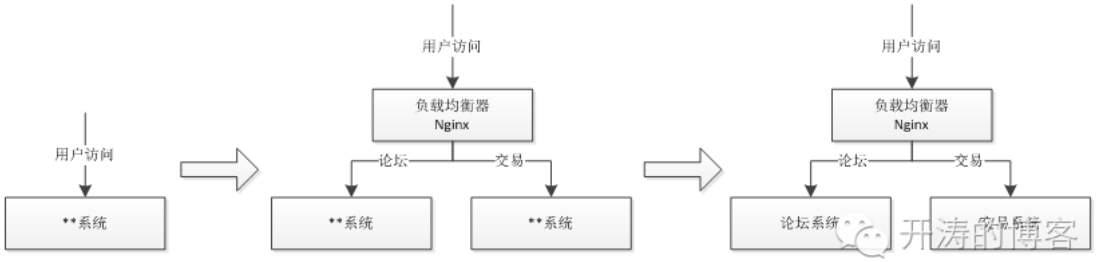


我们会根据服务等级划分两个线程池，以下是池的抽象：

```
<bean id="zeroLevelAsyncContext" class="com.jd.noah.base.web.DynamicAsyncContext" destroy-method="stop">
<property name="asyncTimeoutInSeconds" value="${zero.level.request.async.timeout.seconds}"/>
<property name="poolSize" value="${zero.level.request.async.pool.size}"/>
<property name="keepAliveTimeInSeconds" value="${zero.level.request.async.keepalive.seconds}"/>
<property name="queueCapacity" value="${zero.level.request.async.queue.capacity}"/>
</bean>
<bean id="oneLevelAsyncContext" class="com.jd.noah.base.web.DynamicAsyncContext" destroy-method="stop">
<property name="asyncTimeoutInSeconds" value="${one.level.request.async.timeout.seconds}"/>
<property name="poolSize" value="${one.level.request.async.pool.size}"/>
<property name="keepAliveTimeInSeconds" value="${one.level.request.async.keepalive.seconds}"/>
<property name="queueCapacity" value="${one.level.request.async.queue.capacity}"/>
</bean>
```

进程隔离

在公司发展初期，一般是先进行从0到1，不会一上来就进行系统的拆分，这样就会开发出一些比较大而全的系统，系统中的一个模块/功能出现问题，整个系统就不可用了。首先想到的解决方案是通过部署多个实例，然后通过负载均衡进行路由转发，但是这种情况无法避免某个模块因BUG而出现如OOM导致整个系统不可用的风险。因此此种方案只是一个过渡，较好的解决方案是通过将系统拆分为多个子系统来实现物理隔离。通过进程隔离使得某一个子系统出现问题不会影响到其他子系统。



集群隔离

随着系统的发展，单实例服务无法满足需求了，此时需要服务化技术，通过部署多个服务，形成服务集群来提升系统容量，如下图所示

随着调用方的增多，当秒杀服务被刷会影响到其他服务的稳定性，此时应该考虑为秒杀提供单独的服务集群，即为服务分组，从而当某一个分组出现问题不会影响到其他分组，从而实现了故障隔离，如下图所示

比如注册生产者时提供分组名：

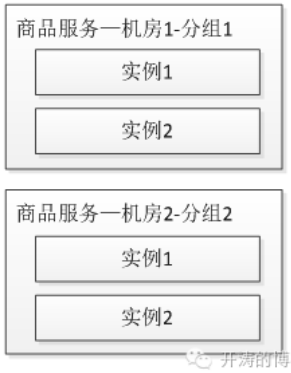
```
<jsf:provider id="myService" interface="com.jd.MyService" alias="${分组名}" ref="myServiceImpl"/>
```

消费时使用相关的分组名即可：

```
<jsf:consumer id="myService" interface="com.jd.MyService" alias="${分组名}"/>
```

机房隔离

随着对系统可用性的要求，会进行多机房部署，每个机房的服务都有自己的服务分组，本机房的服务应该只调用本机房服务，不进行跨机房调用；其中一个机房服务发生问题时可以通过DNS/负载均衡将请求全部切到另一个机房；或者考虑服务能自动重试其他机房的服务从而提升系统可用性。

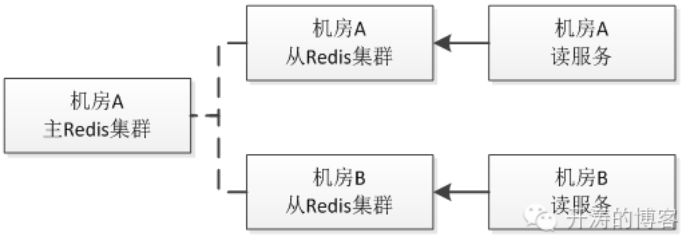


一种办法是根据IP（不同机房IP段不一样）自动分组，还一种较灵活的办法是通过在分组名中加上机房名解决：

```
<jsf:provider id="myService" interface="com.jd.MyService" alias="${分组名}-${机房}" ref="myServiceImpl"/>
<jsf:consumer id="myService" interface="com.jd.MyService" alias="${分组名}-${机房}"/>
```

读写隔离

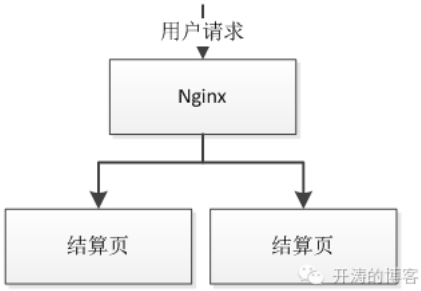
如下图所示，通过主从模式将读和写集群分离，读服务只从从Redis集群获取数据，当主Redis集群出现问题时，从Redis集群还是可用的，从而不影响用户访问；而当从Redis集群出现问题时可以进行其他集群的重试。



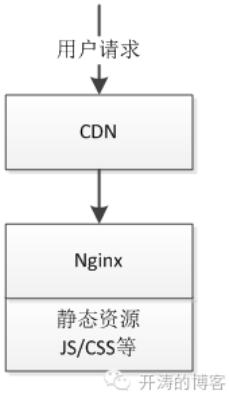

```
--先读取从
status, resp = slave_get(key)
if status == STATUS_OK then
    return status, value
end
--如果从获取失败了，从主获取
status, resp = master_get(key)
```

动静隔离

当用户访问如结算页时，如果JS/CSS等静态资源也在结算页系统中时，很可能因为访问量太大导致带宽被打满导致出现不可用。

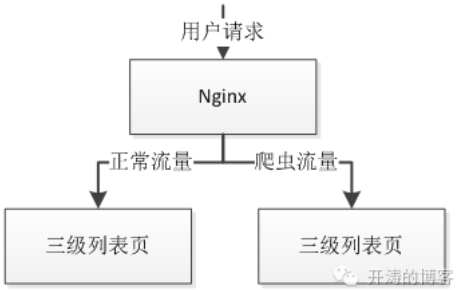


因此应该将动态内容和静态资源分离，一般应该将静态资源放在CDN上，如下图所示



爬虫隔离

在实际业务中我们曾经统计过一些页面型应用的爬虫比例，爬虫和正常流量的比例能达到5:1，甚至更高。而一些系统是因为爬虫访问量太大而导致服务不可用；一种解决办法是通过限流解决；还一种解决办法是在负载均衡层面将爬虫路由到单独集群，从而保证正常流量可用，爬虫流量尽量可用。



比如最简单的使用Nginx可以这样配置：

```
set $flag 0;
if ($http_user_agent ~* "spider") {
    set $flag "1";
}
if($flag = "0") {
    //代理到正常集群
}
if ($flag = "1") {
    //代理到爬虫集群
}
```

实际场景我们使用了Openresty，不仅仅对爬虫user-agent过滤，还会过滤一些恶意IP（统计IP访问量，配置阀值），将他们分流到固定分组。还有一种办法是种植Cookie，访问特殊服务前先种植Cookie，访问服务时验证该Cookie，如果没有或者不对可以考虑出验证码或者分流到固定分组。

热点隔离

秒杀、抢购属于非常合适的热点例子；对于这种热点是能提前知道的，所以可以将秒杀和抢购做成独立系统或服务进行隔离，从而保证秒杀/抢购流程出现问题不影响主流程。

还存在一些热点可能是因为价格或突发事件引起的；对于读热点我使用多级缓存搞定；而写热点我们一般通过缓存+队列模式削峰，可以参考《前端交易型系统设计原则》。

资源隔离

最常见的资源如磁盘、CPU、网络；对于宝贵的资源都会存在竞争问题。

在《构建需求响应式亿级商品详情页》中我们使用JIMDB数据同步时要dump数据，SSD盘容量用了50%以上，dump到同一块磁盘时遇到了容量不足的问题，我们通过单独挂一块SAS盘来专门同步数据。还有如使用Docker容器时，有的容器写磁盘非常频繁，因此要考虑为不同的容器挂载不同的磁盘。

默认CPU的调度策略在一些追求极致性能的场景下可能并不太适合，我们希望通过绑定CPU到特定进程来提升性能。如我们一台机器会启动很多个Redis实例，通过将CPU通过taskset绑定到Redis实例上可以提升一些性能；还有Nginx提供了worker_processes和worker_cpu_affinity来绑定CPU。还有如系统网络应用比较繁忙的话，可以考虑绑定网卡IRQ到指定的CPU来提升系统处理中断的能力，从而提升性能。

还有如大数据计算集群、数据库集群应该和应用集群隔离到不同的机架，并尽量隔离网络；因为大数据计算或数据库同步时时会有比较大的网络带宽，可能拥塞网络导致应用响应慢。

还有一些其他类似的隔离术，如环境隔离（测试环境、预发布环境/灰度环境、正式环境）、压测隔离（真实数据、压测数据隔离）、ABTest（为不同的用户提供不同版本的服务）、缓存隔离（有些系统混用缓存，而有些系统会扔大字节值到如Redis，造成Redis慢查询）、查询隔离（简单、批量、复杂条件查询分别路由到不同的集群）等。通过隔离后可以将风险降低到最低、性能提升至最优。