宅米网性能优化实践

胡勇、李智慧/宅米

宅米是一家专注校园电子商务的互联网企业，目前主营校园超市O2O。公司成立于2014年11月，仅仅一年多的时间，公司即经过4轮融资，覆盖近200座城市，1000多所大中专院校，10000多栋宿舍楼，日均订单20万，峰值订单50万。

像所有高速发展的初创互联网企业一样，宅米的成长是一部野蛮成长的历史。公司成立之初，只有三个工程师，是创始人CEO孙高峰在上海交通大学计算机学院和软件学院挨个宿舍敲门敲出来，他逢人便问：『同学，要不要创业？』。就这样，三个尚未毕业休学创业的学生开发上线了宅米的第一个版本。

早期，为了迅速开发，技术人员选择了Ruby作为开发语言。由于业务快速增长，技术人员缺乏经验，系统甫一上线，即经历了各种bug，各种系统崩溃。往往在业务最繁忙的时候系统宕机了，公司上下焦头烂额匆忙应对，工程师每天工作近20个小时，困了就在桌子上趴一会，醒来接着写代码，修bug。

但是就是在这样的跌跌撞撞中，公司业务仍然快速增长，只几个月的时间就成为该领域中最主要的竞争者，公司顺利获得A轮融资。有钱了，公司便期望在技术研发方面投入更多资源，招聘更多专业技术人才，开发出更完善更稳定的系统迎接下一轮更快速的发展。但是招聘的时候才发现，市面上Ruby工程师非常稀缺，难以招募，技术团队迅速决定转型，使用Java作为主要的后端开发语言。于是几个工程师一边自己学Java，一边招Java，不到两个月的时间，组建了一个20多人的Java技术团队，完成对原有几个核心系统的Java重构。

开发人员增加了，可以更加从容开展开发工作，应对新增业务和需求变更，Bug减少了，系统稳定了。但是这时候的系统架构依然是一个非常简单的Web架构，如图1。

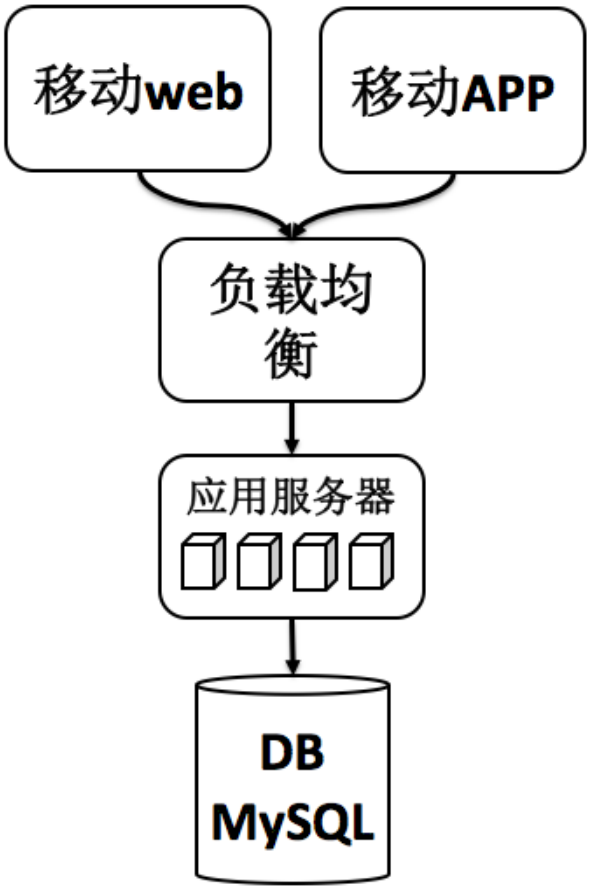

图1 最开始的系统架构

这样的系统能不能应对今后快速的业务发展？性能问题会不会成为持续增长的交易量的瓶颈？系统能不能撑得住访问高峰期的大规模并发访问？

性能优化成为这个时候最重要的工作，于是安排专门的工程师进行性能测试和性能优化，从架构、代码、数据库、运维各个层面梳理系统状况，发现系统瓶颈，进行针对性优化。

一、 性能测试

校园零食购物的特点是在晚上10点左右进入高峰，在此前后一小时的交易量大概占整天交易量的一半，也就是说，如果要设计一个日订单100万的系统，其实要承受的交易压力是每小时50万单。

当初按照二八法则推算峰值每秒单量为556笔『500000 * 0.8 / (60 * 60 * 0.2)』，以此为基准根据Nginx日志分析后端接口调用频率，推算出接口调用比率前20的请求，以此构造测试场景。

在执行性能测试时，我们使用Jmeter作为性能测试工具，利用了云服务提供的系统资源监控作为基础，同时抓取应用服务线程快照和MySQL数据库slow.log分析系统瓶颈。脚本分别如下：

```
//抓取应用服务线程快照
jstack `jps | grep -v grep | grep -v Jps|awk '{print $1}'`
//MySQL数据库slow.log分析
mysqldumpslow mysql-slow.log
```

二、架构优化

性能测试结果并不乐观，我们结合互联网领域常用技术架构模式以及自身性能瓶颈，进行了架构优化重构。

虽然系统此前使用了分布式缓存对热点数据进行缓存，但是比较随意，哪些数据需要缓存，失效策略如何设置都没有认真分析和设计。性能测试后决定规范缓存使用，尽可能将各种频繁读取的数据全部缓存起来，并将Redis服务器做集群和主从复制部署。

此外还使用第三方CDN服务进行静态文件访问加速，产品图片、JavaScript文件、CSS文件等都通过CDN加速，同时通过Nginx反向代理服务器提供静态文件的前端缓存。

性能测试发现，系统主要瓶颈点在数据库上，虽然使用Redis将热点数据缓存起来，但是数据库依然在并发量达到一定程度后表现出系统过载的情况。于是对数据库进行主从分离。

优化后的系统架构如图2。

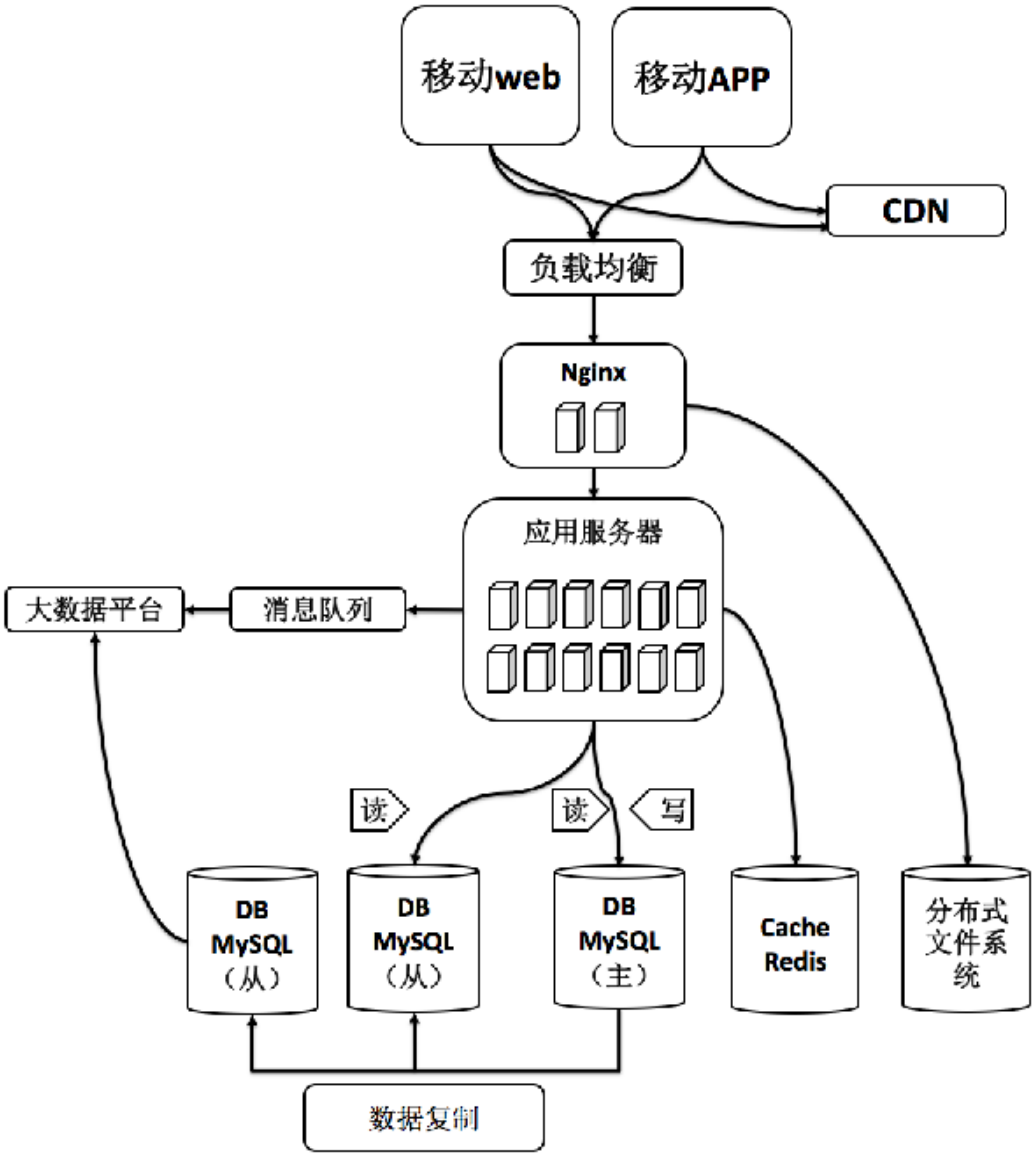

图2 优化后的系统架构

三、H5响应压缩优化

性能测试发现App应用比移动Web端响应速度更快，分析发现H5响应内容因为包含了大量HTML，数据包大小远远大于App响应包。因此决定采用Nginx作为反向代理的同时，对HTML内容进行压缩。

开启Nginx gzip压缩的指令如下：

```
#config gzip;
    gzip on;
    gzip_min_length 1k;
    gzip_buffers 4 16k;
    gzip_comp_level 2;
    gzip_types text/plain application/javascript application/x-javascript text/css application/xml text/javascript image/jpeg image/gif image/png application/json;
```

关于gzip_types，我们针对JSON数据也开启gzip压缩，降低App响应数据包大小，提高响应性能。图3是开启gzip前后的性能测试结果对比：

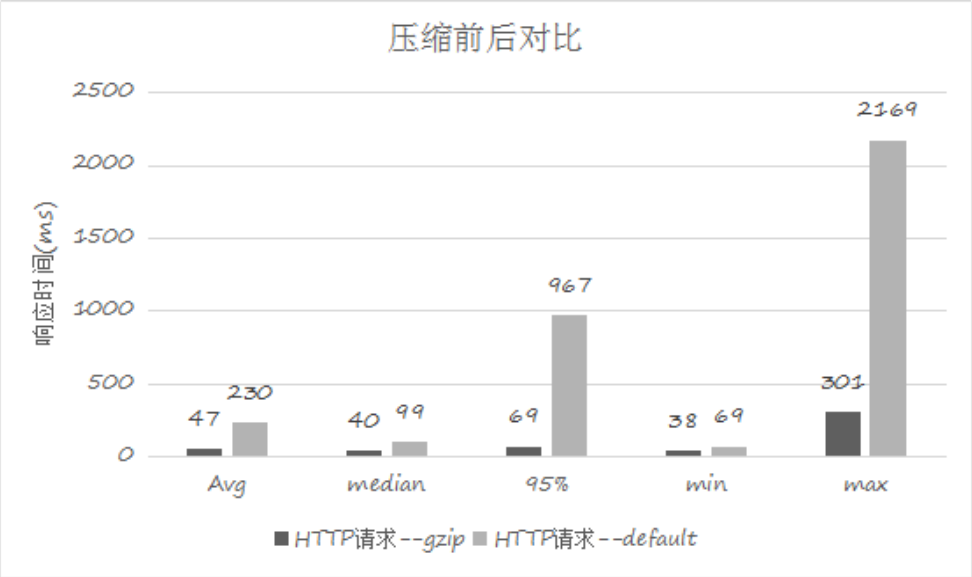

图3 H5页面开启压缩前后性能对比

四、SQL语句与索引优化

性能测试过程中发现，由于此前主要精力都在关注如何快速实现业务，大量数据库查询语句写得比较随意，索引设计非常不合理。

结合性能测试中Mysql数据库slow.log分析，定位慢查询SQL追加index，然后利用解释执行计划explain优化SQL。在此简要列举几处示例。

（1）某字段类型为varchar类型，根据查询关键字段查询时，写入值为Int类型，导致无法命中索引。

优化前：

```
select * from aa where aa.bb = 1449220364536130715;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	aa	ALL	index_aa_on_bb	[Null]	[Null]	[Null]	119392	Using where

优化后：

```
select * from aa where aa.bb = '1449220364536130715';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	aa	const	index_aa_on_bb	index_aa_or	194	const	1	

（2）查询条件左边写入函数，导致无法命中索引。

优化前：

```
select * from cc where date_format(dd,'%Y-%m-%d')=
(DATE_SUB(CURDATE(),INTERVAL 1 DAY));
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	cc	ALL	[Null]	[Null]	[Null]	[Null]	119392	Using where

优化后：

```
select * from cc where dd=(DATE_SUB(CURDATE(),INTERVAL 1 DAY))
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	cc	ref	index_cc_on_dd	index_cc_on	4	const	1	

（3）追加Index时，计算数据唯一性巧妙添加左前缀索引，提高索引命中率，保证索引字段唯一性。

利用如下SQL计算索引命中率：

```
select count(distinct left(pinyin_initial,3))/count(*) as sel3,
count(distinct left(pinyin_initial,4))/count(*) as sel4,
count(distinct left(pinyin_initial,5))/count(*) as sel5,
count(distinct left(pinyin_initial,6))/count(*) as sel6,
count(distinct left(pinyin_initial,7))/count(*) as sel7
from city;
```

以此算出城市拼音缩写长度为3时，命中率和唯一性比较高，则写下如下SQL：

```
ALTER TABLE `city` ADD INDEX `index_on_pinyinInitial` USING BTREE
(pinyin_initial(3));
```

五、数据库连接池优化

数据库的访问优化也比较重要，宅米后台系统开发使用了Mybatis + C3P0组合，在做性能测试的时候发现在某些情况下有较为严重的性能问题。在高并发情况下，长时间施加压力，应用程序出现不能访问的状况。

上网查找资料，发现很多人也遇到了C3P0的”APPARENT DEADLOCK”问题。

将C3P0切换成国产数据库连接池Druid之后，状况明显好转，类似问题再未出现过。

六、缓存使用优化

经过对数据库和缓存应用的一系列优化后，缓存的命中率保持在90%以上，进一步研究后发现，Redis使用依然有提升的空间。

应用程序访问Redis的时候，可以通过使用Jedis的pipeline减少redis通信次数，有效提升性能。Jedis是基于socket通信实现的，每次与Redis通信都会消耗相当的网络连接时间，pipeline则是以打包批量的形式执行命令，图4是执行5000次set操作的响应时间对比：

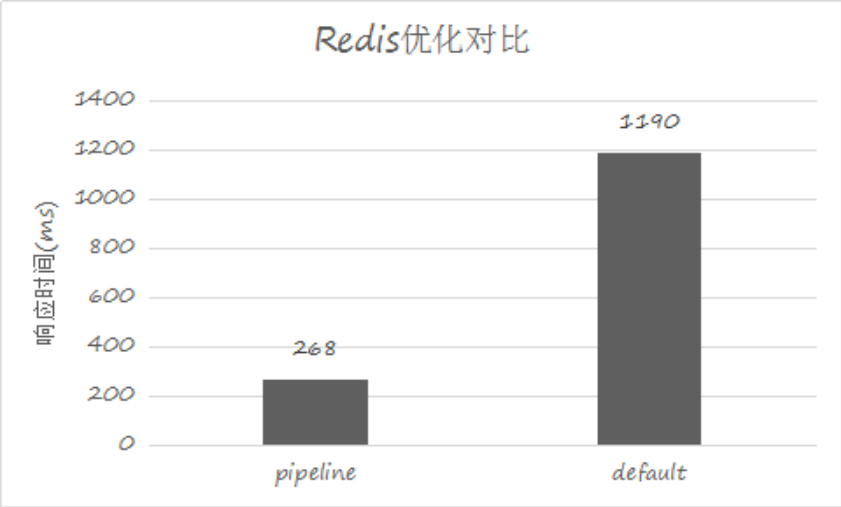

图4 Jedis pipeline性能测试结果

七、订单数据冷热分离

随着业务的持续发展，订单表的数据会越来越多。按我们现在日订单量20万单预估，月订单量则为600万单，年订单量则达到7200万单，而且日订单量还在不断的增加，用不了多久，数据量就会超过MySQL的极限。

一开始我们考虑使用分布式数据库的方案，对订单表进行水平切分，使用订单号进行hash，将订单数据切分到多张表上。 进一步分析后发现，订单数据具有明显的冷热不均的特点，即刚刚创建的订单是热数据，不同应用以各种方式访问修改这些订单。经过一段时间以后，特别是订单完成后，订单访问频率急剧降低，而且只有订单查询这一种操作。于是我们考虑采取冷热数据分离的策略，以控制热库中数据总量，保障订单表数据量始终维持在一个可以接受的范围内，进而提供稳定的数据访问性能。订单数据冷热分离方案如图5。

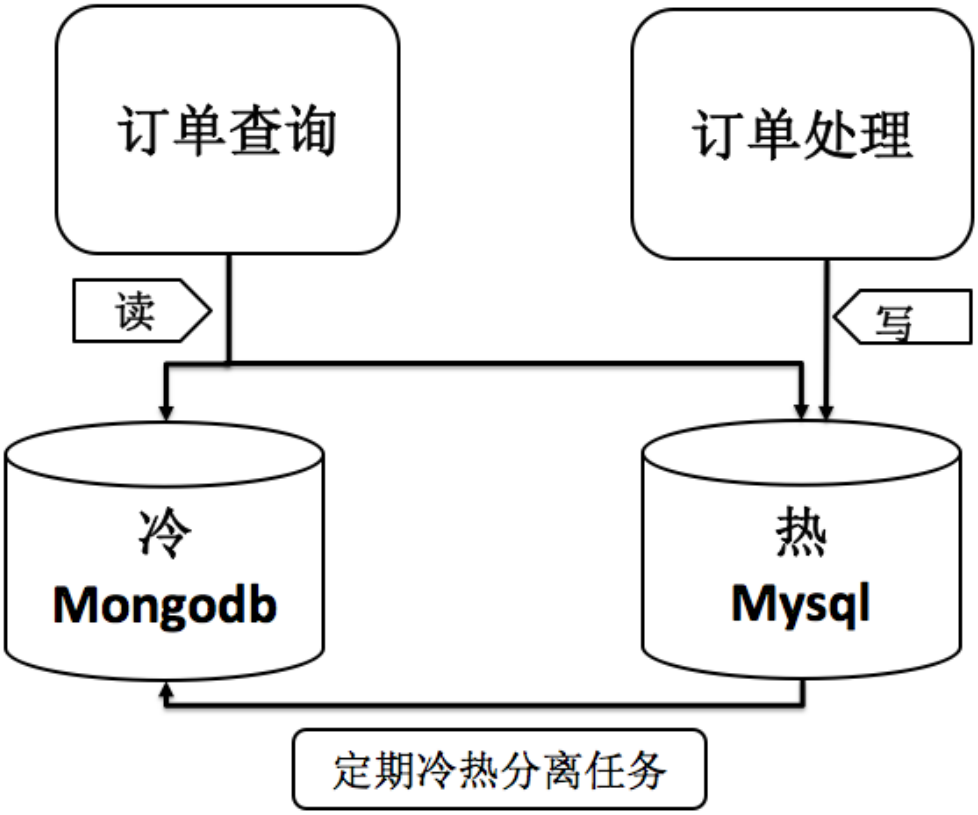

图5订单数据冷热分离

八、系统性能监控

性能测试和性能优化虽然对系统做了充分的改进，但是实际线上性能表现究竟如何，出现紧急性能问题时如何快速应对，还必须要对生产环境进行性能监控。在此简要列举一些宅米的性能监控报警要点：

层次	监控要点
系统	CPU/内存/网络/磁盘
基础应用：数据库	连接数、QPS、TPS、慢查询、缓存命中率、全表扫描、主从延迟等
基础应用：分布式文件系统	流量、带宽、存储空间、读写请求频率等
基础应用：分布式缓存	操作数、命中率、连接数、key数量等
业务应用	API：可用性、延迟、吞吐率 SQL语句：查询分布、频率、耗时 第三方服务调用：耗时、吞吐率

除了系统自身监控，很多系统故障和性能问题会直接反应到业务上。如果系统响应缓慢甚至宕机，那么实时订单量也会受到影响，因此监控实时交易也可以发现系统问题。图6是实时交易监控图，在这里例子中，21:33订单量突然降到零，虽然系统监控指标正常，但是可以断定系统必定出了问题，马上打开应用日志查看，发现有个Bug导致某个外部资源死锁，立刻手工释放该资源，系统恢复正常。

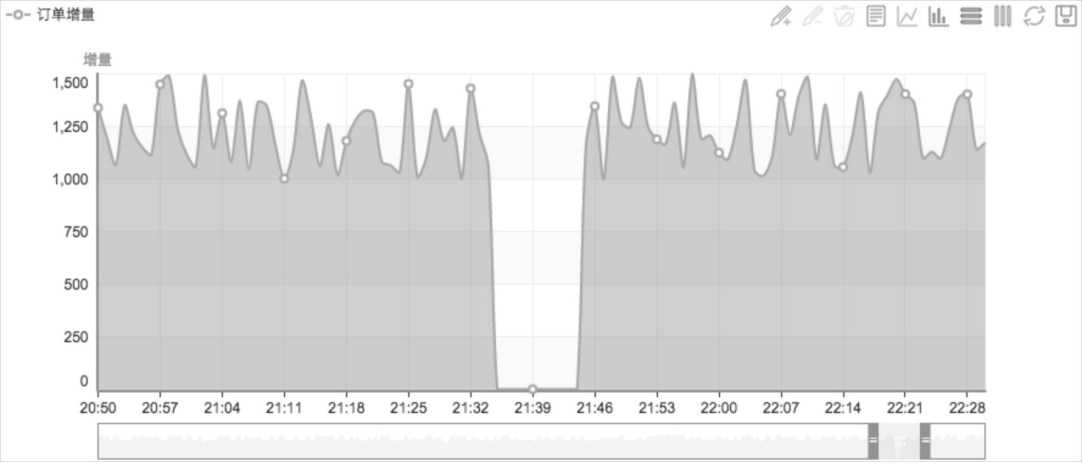

图6实时订单监控

九、总结

性能问题是实打实的问题，解决办法也应该针对具体问题各个击破。通过性能测试了解系统现状，通过瓶颈分析发现具体问题，针对具体问题寻找解决方案，实现解决方案再进行性能测试，整个性能优化形成闭环，系统得以持续优化。

经过一系列各种性能优化，虽然宅米主要系统性能现阶段能够满足需求，但是技术永远要走到业务的前面，才能在业务增长以后从容应对。而初创互联网公司的野蛮成长速度，永远也不要猜测，技术必须要做好充分准备，才能不拖业务的后腿，从容应对各种局面。