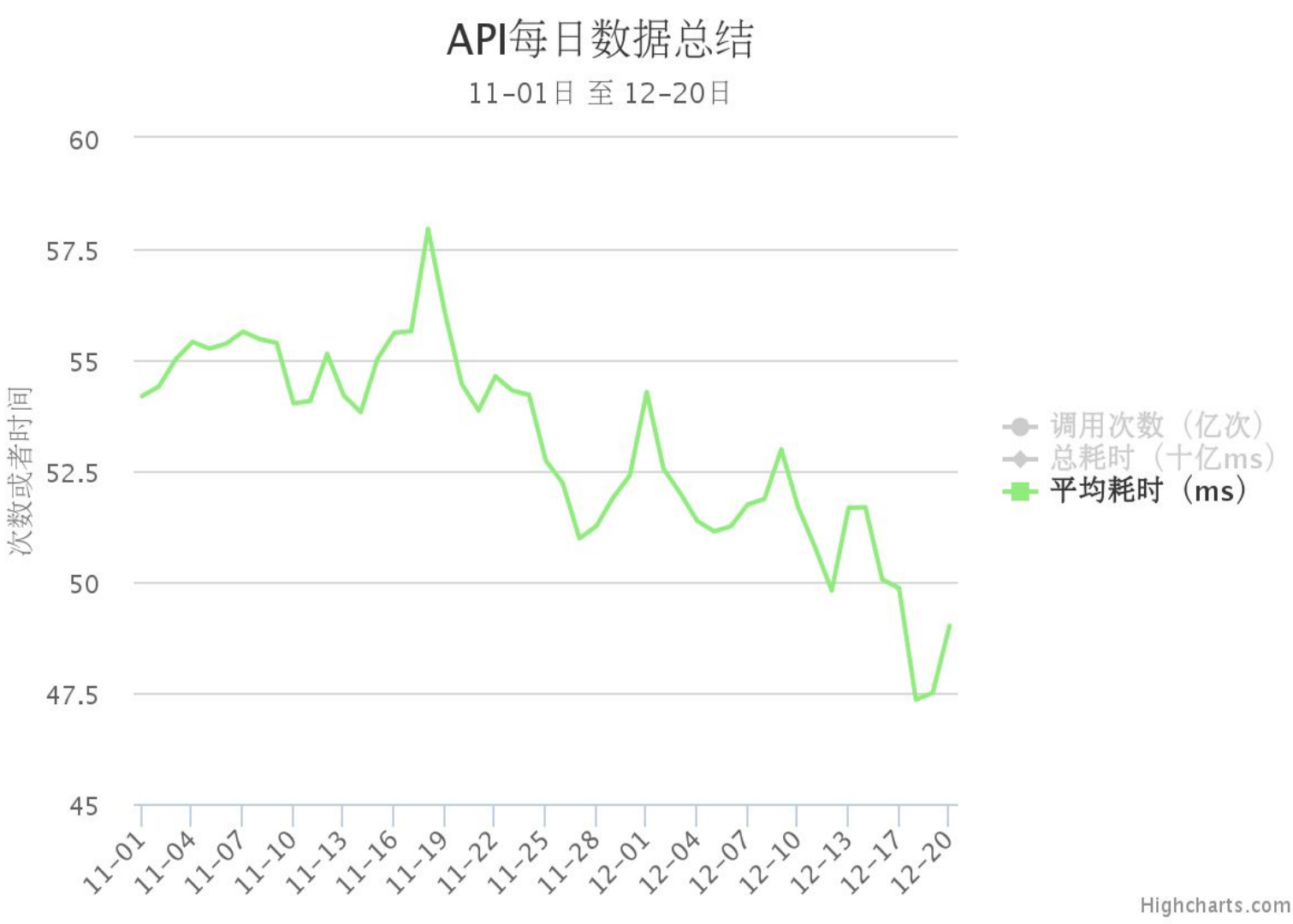
Performance Optimization From Momo Api Team

Glowdan & Zoco

2015 12 24

优化成果

- 平均耗时从55ms降低到48ms
- 部分接口响应时间提高了一倍
- 降低了服务器负载
- 提高了单机QPS能力
- 一套完整的性能分析平台


API每日数据总结
11-01日 至 12-20日
60
57.5
55
52.5
50
47.5
45
次数或者时间
11-01
11-04
11-07
11-10
11-13
11-16
11-19
11-22
11-25
11-28
12-01
12-04
12-07
12-10
12-13
12-17
12-20
调用次数（亿次）
总耗时（十亿ms）
平均耗时（ms）
Highcharts.com

接口优化数据『请求量超过100万，平均耗时超过150毫秒』

	平均响应时间(ms)		
接口	优化前	优化后	变化
接口1	165	147	**10.5%**
接口2	236	213	**9.8%**
接口3	213	116	**45.5%**
接口4	395	279	**29.3%**
接口5	402	231	**42.7%**
接口6	194	155	**20.2%**
接口7	244	153	**37.4%**
接口8	193	141	**27.0%**
接口9	216	156	**28.0%**
接口10	224	125	**44.3%**
接口11	150	143	**4.5%**

发现问题

怎样获取线上性能数据？

每天30万日志，怎样分析？

选取什么样的点进行优化？

思路？？？

思路是怎样的？

优化哪些接口

重点接口 -> 有问题的接口

可视化，自动化

趁手的工具是成功的关键

邮件＋消息通知

关注哪些数据，以什么角度

请求量、平均耗时、总耗时

变化率

整合分析

这么做的原因是

在什么情况下做了这个决定？现在是否有更好的方法了？

数据来源：Xhprof

- Xhprof是一个函数级的分层的性能分析扩展。使用C开发。它能方便的统计函数级别的调用关系，内存使用，CPU时间和调用次数
- 线上环境部署，API使用了万分之一的采样率
- 每天供性能分析的数据在30万左右

高阶优化法

原始数据

- 耗时排名
- 次数排名
- 查看调用链
- 关注耗时较多函数
- 超过150ms的接口

分类统计

统计什么参数

- **耗时**

  找到耗时最多的调用：发现递归，循环，不合理的用法，异常的服 务

- **次数**

  找到调用次数异常的调用：寻找批量方案

分类统计

关注哪一类数据

1. RPC
2. Redis
3. Function

整合统计

将某段时间的请求合并成一次请求

Link

1. 能看到当前接口所有运行过的函数。
2. 平均值处理，也避免了程序运行中的不规律事件。
3. 统计值比孤立值更具说服力
4. 为增量统计做准备

ps: 可以使用Xhprof展示平台

整合统计变化趋势

RPC 耗时变化，请求量变化

Redis 耗时变化，请求量变化

Function 耗时变化，请求量变化

业务新增检测

检测多余出的方法以及此类方法的耗时

从而得出业务变化对性能的影响

手段

- 短路
- 六脉神剑
- 兵马未动，粮草先行
- 诸葛连弩
- 蝴蝶效应
- 降维攻击

优化方法 之一 兵马未动，粮草先行

将需要的数据事先准备好

案例解析：附近广告位的输出

优化前：三个广告位分别请求，需要请求广告三次

第一次优化：将请求到的数据存到内存中，虽然还是需要请求三次，但是下一次的请求直接读取内存即可

第二次优化：直接将三次请求归并到一次请求

优化方法 之二 一劳永逸

分布式缓存Redis	单机缓存Yac
static静态变量，类变量	CPU计算结果
发现并避免重复调用	单例模式

优化方法 之三 诸葛连弩

将多次对RPC的请求改为一次批量请求

优化前：

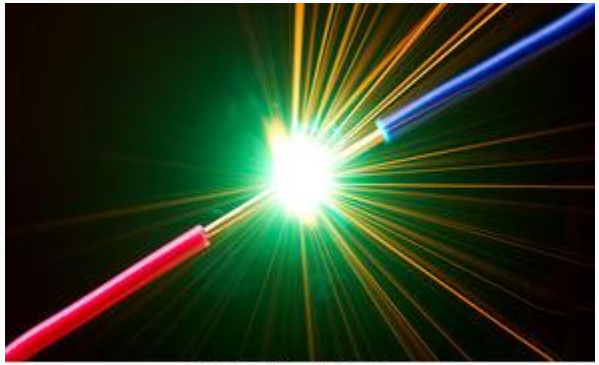
```
foreach($momoids as $momoid) {
        doSomething_once($momoid);
}
```

优化后：

```
doSomething_mulit($momoids);
```

优化方法 之四 短路

将**经常**判断的条件放在 if( ) 中条件的前面



优化前：

```
if($sex == 'F' && $region == '美国' && $age >90) {}
```

优化后：

```
if($age > 90 && $region == '美国' && $sex == 'F') {}
```

代码实例：

```
if (($charlet = $this->outputChatlet()) && ($this->ctx->old->loginUser->isLocal())) {}
```

优化方法 之五 降维攻击

《三体》里面的维度攻击，能把三维变二维，实现毁灭性打击。

案例：

API output时候需要对输出结果进行过滤

优化前：array_walk_recursive递归进行正则处理

优化后：将数组通过json_encode降维成字符串，一次正则即可

优化方法 之六 六脉神剑

在完全串行运行的系统里，一次请求总响应时间满足如下公式：

一次请求总耗时=解析请求耗时 + ∑(获取数据耗时+处理数据耗时) + 组装返回结果耗时

多次请求耗时=∑(一次请求总耗时)

问题：

PHP没有非常完善的并行处理机制。Swoole有解决方案，但是会影响PHP的部署机制。

优化方法 之六 六脉神剑

解决方法：

将RPC调用并行化，将互相不依赖的RPC使用专门的服务打包调用。

并行化后请求时间总耗时公式：

多次请求总耗时=解析请求耗时 + Max(单次请求耗时) + 组装返回结果耗时

上线后：

全部API平均响应时间从50ms达到了48ms

优化方法 之七 蝴蝶效应

某复杂业务系统依赖于多个服务，其中某个服务的响应时间变长，随之系统整体响应时间变长，进而出现CPU、内存报警。

反其道而行之，降低某一个关键服务的响应时间会提升整个系统的性能。

越是**底层**的优化这种效果越明显。

其他建议

1. 少用@错误抑制符
2. 不用array_walk_recursive()或者其他形式的递归，可以采用降维的方法简化问题
3. 不要写$arr[1][2][3][4][5] = $i
4. 变量先定义再引用
5. 用===代替==
6. 用null === $a代替is_null($a)
7. 用str开头的函数代替preg开头的函数

优化实践

Instruction 1

在PPT中罗列的编码时候的优化建议，不是为了让大家在出现性能问题的时候，以这些作为准则去对代码进行优化，而是希望能在大家最初写代码的时候，就有一个心理的认知，应该怎么写会更好。

Instruction 2

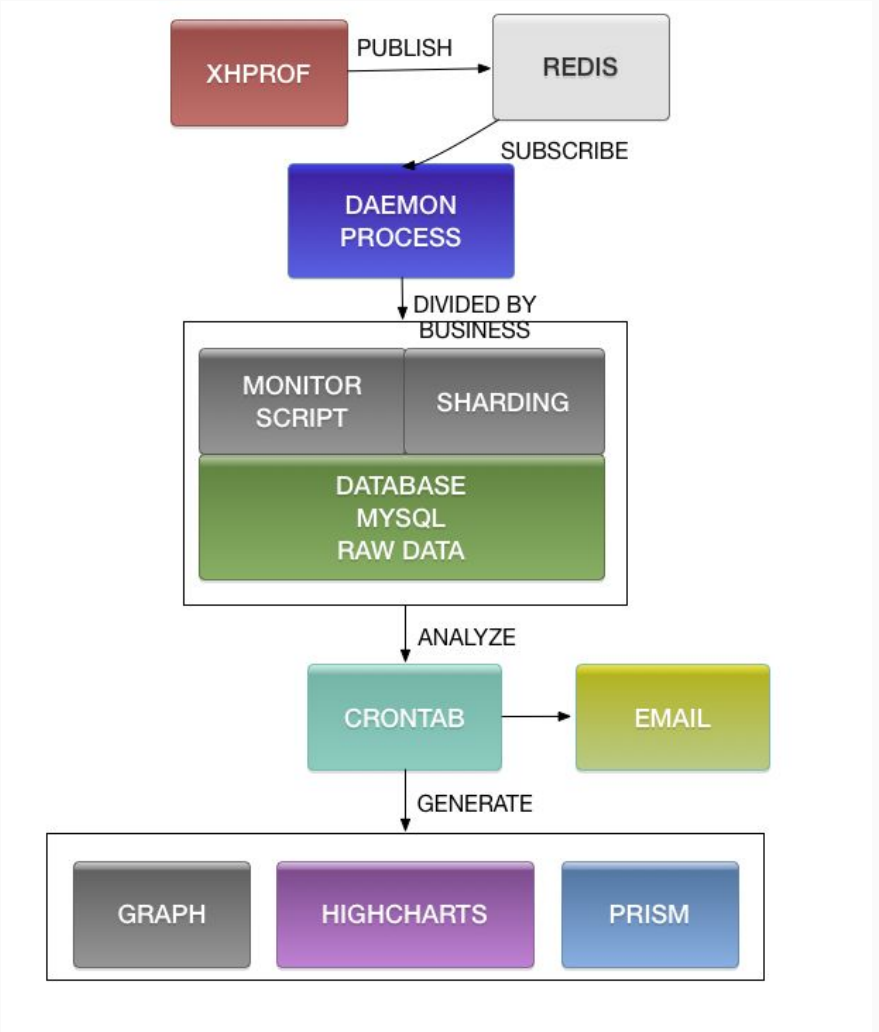
优化的建议，是建议，是防止大家滥用，如果你能在写代码的时候，能意识到，什么慢，什么快，从而避免一些没有必要的调用，那就是这个优化建议所追求的效果了。

PHP对程序员的要求更高

对于C来说，它有一个很强大的编译“优化”器，可以为你做很多优化，而PHP是解释型脚本语言，它只会重视的执行你的代码，不会做任何优化，所以，你写的代码的风格，直接影响到最终的执行风格。

分析系统结构图

1. Redis Pub 数据
2. Sub 脚本收集数据，保存到不同的库，表中
3. PubSub还可以双写，给程序开发带来了极大的便利性
4. 每天凌晨分析并前一天的数据并入库
5. 发送总结邮件到相关负责人



工具

- 工具选型
- 数据量逐渐加大怎么办
- 数据库怎么设计
- 报警+通知

分析平台数据

1. 存储空间1T->3T
2. API每天日志量9G，全部20G
3. 分析数据为200M
4. 数据库使用Postgresql -> Mysql

存储经历的问题

1. 迁库（数据删除不释放空间）
2. 分库（单表数据过大）
3. 分表（进一步缩小单表数据）
4. 数据可视化（获取数据量，查看进程，数据量，存储空间）
5. 对指定机器赋予相应权限

Tips

1. 每日报表，持续关注各种数据，实时分析
2. 放量策略，1% 5% 20% 60% 100%
3. 胆大心细，敢于实践
4. 优化的过程中了解业务，随时修改Bug
5. 业务是推进架构的演化和升级的原动力
6. 加大数据度量，数据是所有决策的基础，不要赶时髦，人云亦云
7. 利用自己熟悉的技术解决问题，数据化衡量新技术引入复杂度和新问题

未来的方向

1. 核武器（PHP-7）
2. RPC并行化
3. 耗时处理使用扩展重写
4. 从客户端到客户端全链优化