如何设计出高可用、高性能的接口

发起这个Chat只是一时兴起，想了一些点就写出来了，但自己一读，感觉一点干货都没有，真是汗颜。但还是也希望此拙文能带来一些你的思考，欢迎交流。

接口设计需要考虑哪些方面

1. 接口的命名。
2. 请求参数。
3. 支持的协议。
4. TPS、并发数、响应时长。
5. 数据存储。DB选型、缓存选型。
6. 是否需要依赖于第三方。
7. 接口是否拆分。
8. 接口是否需要幂等。
9. 防刷。
10. 接口限流、降级。
11. 负载均衡器支持。
12. 如何部署。
13. 是否需要服务治理。
14. 是否存在单点。
15. 接口是否资源包、预加载还是内置。
16. 是否需要本地缓存。
17. 是否需要分布式缓存、缓存穿透怎么办。
18. 是否需要白名单。

当我们设计接口，我们或多或少都会有上面列举的一些考虑，我们只有想的更多才能让让我们的接口更加完善，我个人觉得100%完美的接口是不存在，只有适合才是最重要。

接口设计原则

原则一：必须符合Restful，统一返回格式，约定业务层错误编码，每个编码可以携带可选的错误信息。

原则二： 命名必须规范、优雅。

原则三：单一性。

单一性是指接口要做的事情应该是一个比较单一的事情，比如登陆接口，登陆完成应该只是返回登陆成功以后一些用户信息即可，但很多人为了减少接口交互，返回一大堆额外的数据。比如有人设计一个用户列表接口，接口他返回每一条数据都是包含用户了一大堆跟另外无关的数据，结果一问，原来其他无关的数据是他下一步想要获取的，想达成数据的懒加载

原则四：可扩展。

接口扩展性，是指设计接口的时候多想想多种情况，多考虑各个方面，其实我觉得单独将扩展性放在这里也是不妥的，感觉说的跟单一性有点相反的意思，其实这个不是这个意思，这边的扩展性是指我们的接口充分考虑客户端，想想他们是如何调用的，他要怎样使用我的代码，他会如何扩展我的代码，不要把过多的工作写在你的接口里面，而应该把更多的主动权交给客户程序员。如获取不同的列表数据接口，我们不可能将每个列表都写成一个接口。 还有一点，我这里特别想指出来的是很多开发人员为了省事（姑且只能这么理解），将接口设计当成只是app页面展示，这些人将一个页面展示就用一个接口实现，而不考虑这些数据是不是属于不同的模块、是不是属于不同的展示范畴、结果下次视觉一改，整个接口又得重写，不能复用。

原则五：必须有文档。

良好的接口设计，离不开清晰的接口文档表述。文档表述一定要足够详细

原则六：产品心。

为什么我说要有产品心？因为我觉得很多人忽略了这一点。我来说一下假如开发一个app，如果一开始连个交互文档给你都没有的话，你怎么设计接口？所以我觉得作为一个服务端后台开发人员应该要有产品心，特别是对于交互文档应该好好理解，因为这些都会对我们的接口设计有很大的影响，我在设计接口的时候就很常发现很多交互文档根本就走不通，产品没有考虑到位，交互文档缺失，这时候作为一个开发要主动推动，完善。

原则七：第三方服务接口数据能缓存就缓存。

原则八：第三方服务需要做降级。

原则九：建议消除单点。

原则十：接口粒度要小。

原则十一：客户端能处理的逻辑就不要给服务端处理，减少服务端压力。

原则十二：资源预加载。

原则十三：不要过度设计。

原则十四：缓存尽量不要穿透。

原则十五：接口能缓存就缓存。

如何保证接口的高可用、高性能

上面也列举很多需要考虑和设计的原则，其实还有很多方面，我这边也不是特别全面。居于上面列举的这些考虑点，其实这边说服务是更恰当，能把上面说的点做好，其实接口也是比较可靠，如何设计以及保证接口的高可用和高性能。可以思考一下以下几个point

高性能：如果我们发现这个接口tps和响应时间没有达到我们的要求怎么办。

- A：数据存储方面：我们会想数据库有没有分库、分表、有没有做主从，有没有读写分离、字段是否有加索引、是否存在慢sql，数据库引擎是否选用合适、是不是用了事务；其次我们会想到是不是引用了分布式缓存、缓存key大小是否合适，失效时间是否设置合理，会不会大量缓存穿透、有没有引入本地缓存。

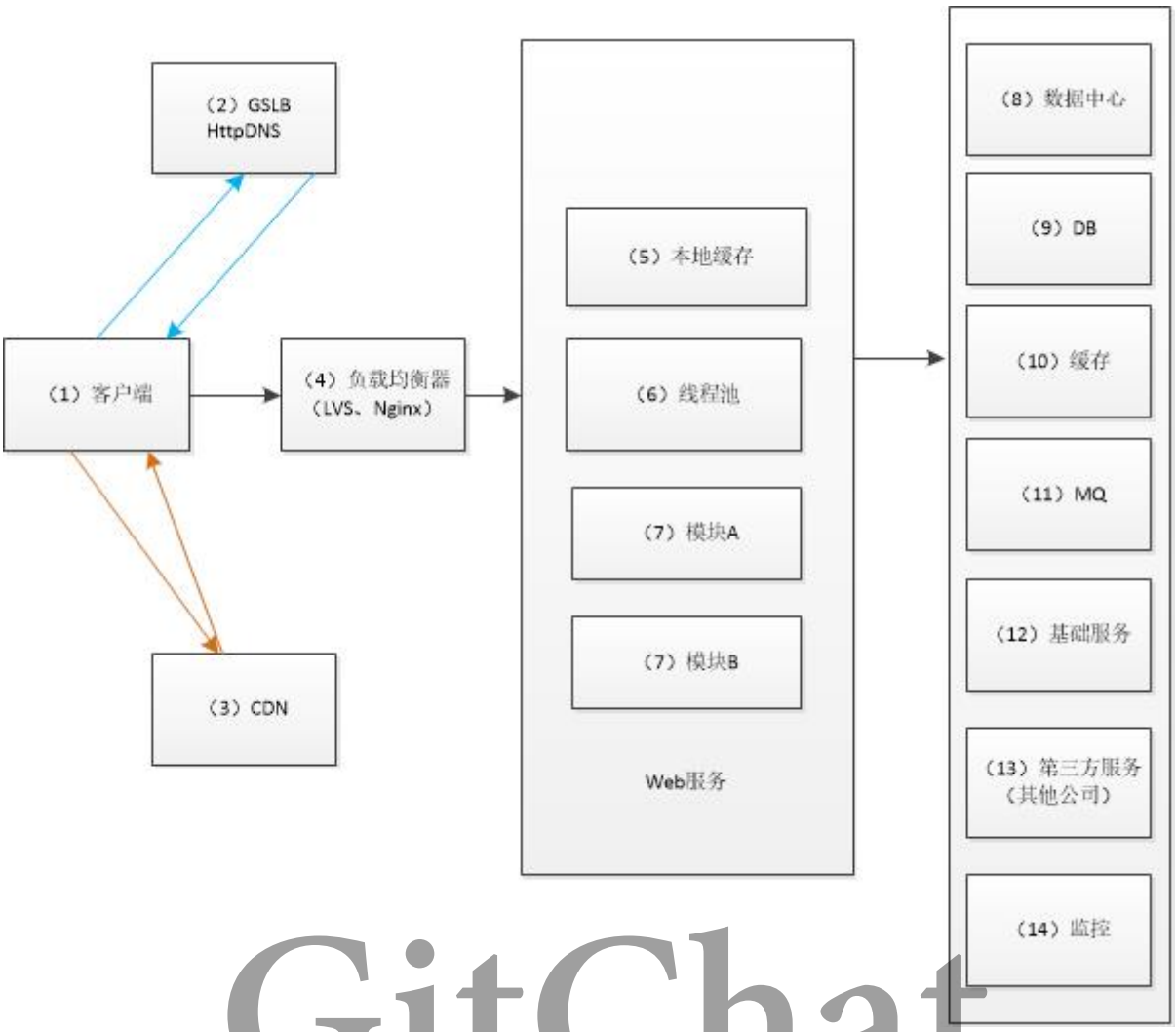
- B：业务方面：是否有大量的计算、能否异步处理。是否需要引入线程池或者MQ来异步处理任务。有没有必要将接口进行垂直拆分和水平拆分、将接口粒度变小。

- C：其他方面：nginx层面做缓存、加机器、用ssd，资源放cdn，多机房部署、资源文件预加载。

高可用：如何保证服务高可用，需要从几个维度来实现：

- A：消除单点，基于高可用第二位。
- B：能做集群的全部做集群。譬如Redis集群、mysql集群、MongoDB副本集。
- C：能做读写分离的都做读写分离。
- D：异地多机房部署，接入GSLB
- E：必须有限流、降级机制。
- F：监控。高可用的保证，基于第一位。

下图是从一个基本的请求出发来梳理需要涉及到各个段，以及各个端能做的事情。谈谈接口服务，但不局限于接口本身。



1. 客户端：资源预加载、限制请求、数据上报。我这边就拿客户端来举个例子。接口服务所依赖的资源包或者一些公共配置预加载在本地，减少接口的交互，通过请求配置文件是否更新，code是否是304等来；接口做一些请求限制，比如抢红包、抢券等，单位时间内N次点击只请求一次等；接口失败数据上报来；这就是客户端可以做到的对接口有帮助的事情

2. GSLB/HttpDNS：多机房部署、流量切换、域名劫持，一般技术和业务比较成熟的公司这一层。

3. 资源文件放CDN。

4. 负载均衡器：lVS+Nginx是互联网常用的做负载均衡，可以实现四层/七层负载均衡；这里除了可以分流、转发以外，我们用的更多的基于令牌桶限流、缓存。

5. 本地缓存。本地缓存能减少我们访问DB或者分布式缓存，本地缓存推荐使用guava，guava里面有很多特性很好用，例如基于令牌桶的限流；当缓存失效时只穿透一个请求去访问后端。

6. 线程池。

7. 模块拆分。将一个项目按功能模块拆分，一个接口也可以按业务粒度进行拆分。

8. 数据中心。提供数据支撑，譬如黑名单。

9. 数据库。加索引、分库、分表、读写分离

10. 分布式缓存。数据分片、拆分大key，并做集群，采用分布式锁

11. MQ。做接口拆分利器，异步操作。

12. 其他服务。限流、防刷以及降级（特别是第三方服务，保证第三方服务down掉不要影响我们自身的服务）。在这里也需要考虑做第三方数据的缓存或者持久化，譬如实名认证、身份证认证等。

13. 监控。监控永远是必须的，能让你第一时间知道接口服务是否ok

个人小分享

1）接口**Restful**，统一返回格式，约定业务层错误编码，每个编码可以携带可选的错误信息

在前司，客户端和服务之间是有统一的数据返回格式，约定各层的编码，可以通过编码位数以及编码就可以看出是那一层出问题，我觉得这对我们定位问题以及维护来说具有莫大的意义，并对异常也进行捕捉，封装成对应的code，我之前阅读一些人的代码发现其项目根本没有做这一层，因为简单而不做我觉得有失所望。

2）采用**hybird**模式

采用hybird模式涉及到资源预加载的问题，在很多项目里面都大量使用，譬如前司的生活服务，就采用了hybird模式，先将资源文件（包含图片、前端页面）打包放到服务器并通过版本号进行管理，并通过一个总的配置文件来管理，如果是H5页面可以进行模板预先设计，down到本地。
配置文件格式：

```
*文件1*
     name：xxx
     url：http:xxxx
     md5：xxxx

*文件2*
     name：zzz
     url：http：zzzz
     md5：zzz
```

客户端每次启动应用或者定时请求总的配置文件，通过http code是否是304判断是否需要下载这个总的配置文件，如果code是200，那么下载这个配置，比较那个文件发生变化，并将其下载。这样的好处：

1. 减少接口的交互；

2. 资源预加载，节省流量，打开页面更加流畅，对于服务端来说字需要返回数据json串就行，而不需要其他，减少服务端压力；
3. 方便开发人员，资源管理更加简洁，比如做活动需要的h5页面，只需要前端上传对应的h5资源包到服务端，不需要通过后端开发人员就可以搞定。

虽然这个原理很简单，但是现在很多app还是没有做这个，都是通过填写一个url，加载网页的方式去打开，体验性太不友好。

3）客户端

客户端跟服务端就是接口请求的关系，很多时候需要要求客户端做一些数据缓存的工作以及一些检验工作。在前司已经好几次给客户端的同学坑过了，客户端同学接口乱调用，死循环调用。一次是做一个关于事件提醒的功能，需要每天定时调用调用服务端一个接口，结果客户端的同学写了一个bug导致请求每隔一两秒就调用一次，导致服务器这边此接口pv翻了N倍，而且这个bug通过测试同学很难测试出来；还有一次发现服务端一段时间以后UV不见涨，但是PV却涨的很猛，定位发现是客户端同学A图省事在一个方法里面调用了N个接口，也就是模板方法，因为版本更新，同学B需要做一个新的功能，然后也调用了A同学的接口导致，从而导致PV上升，其实B同学完全不需要调用这么多接口。这些都是真实案例，所以这里需要有一个监控接口异常的机制。

4）思辨大于执行

写到这里觉得这个非常重要，思辨大于执行，意味着我们不是一股脑就去干，也不是不去干，我们做事情需要思考、辨别；从而让事情更高效、更好、更有力的执行。接口设计也一样，需要我们去思辨。

5）本地缓存、分布式缓存以及异步

缓存在前司主要分为客户端缓存、CDN缓存、本地缓存（guava）、Redis缓存。在MZ早期是接口是采用DB+本地缓存的方式提供数据，但这种模式DB压力大，接口吞吐量小，本地缓存多机难一致性、更新不及时问题。为了解决这些问题，引入分布式缓存，并通过Task将业务数据刷到Redis，接口只访问redis，不会访问DB，及时DB故障也不会影响功能。不同的业务系统系统通过MQ来解耦，多机房不是通过MQ来实现数据的一直。比如，评论，先通过写Redis，写MQ来实现数据在多机房同步，再通过task将Redis中评论同步到DB中。

接口设计涉及方方面面，这边也只谈到一个大概，虽然有点泛泛而谈，希望此拙文对你有所启示。

6）数据库

数据库分库分表，一般都是通过userId或者imei或者mac地址来分表，单表数据量控制在500w以内，这需要我们提前估算好数据量，尽量避免数据的迁移。在前司，数据库一般都是采用mysql+MongoDB两种，MySQL存储用户的用户数据，MongoDB存储业务数据，就像阅读和生活服务里面的业务数据就存储在MongoDB里面。在数据库这层，我们主要也是通过主从模式、读写分离、分库、分表来实现数据的可用性。

7）业务

业务尽可能拆分、独立部署、将项目按业务划分、按功能划分等。譬如生活服务，我们当时主要拆分成管理后台admin、任务task、活动、web、数据展示模块。

8）数据中心

每个大一点的公司都有数据部门，我们这边可以通过数据中心的数据分析来达到我们需要的数据。
比如黑名单，推广效果、活动数据。我们可以通过这些完善我们的接口功能。之前在前司做了个数据处理后异步加载到Redis来实现数据利用的项目。

以上都是我个人的一些拙见，请大家思辨。