# 架构

## 400% 的飞跃－web 页面加载速度优化实战 - 推酷

http://www.tuicool.com/articles/6nUvuaj

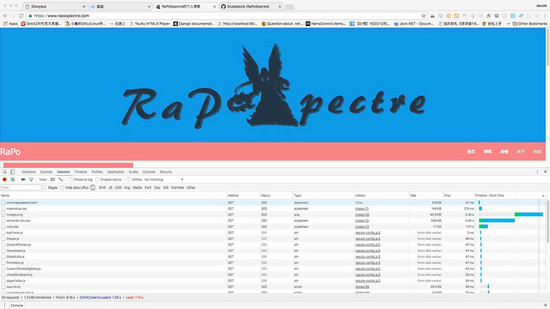
时间 2016-09-28 23:42:32  [SegmentFault](http://www.tuicool.com/sites/3uEjYv)

原文  [https://segmentfault.com/a/1190000007035107](https://segmentfault.com/a/1190000007035107?utm_source=tuicool&utm_medium=referral)

主题 [JavaScript](http://www.tuicool.com/topics/11060004)[CSS](http://www.tuicool.com/topics/11060002)[Nginx](http://www.tuicool.com/topics/11090014)

前言

一个网站的加载速度有多重要？ 反正我相信之前来 [***博主网站***](https://www.rapospectre.com/) 的人至少有 50% 在加载完成前关闭了本站。 为啥捏？ 看图

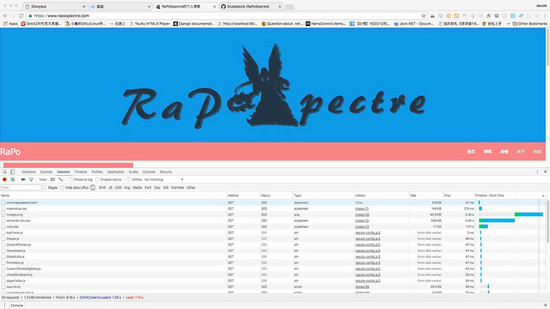


首页完整加载时间 8.18s，看来能进来看博主网站的人都是真爱呀，哈哈。 正常来讲一个网页 4s 加载不完就会流失很大一部分用户，而博主的网站加载时间竟然达到了 8s 还是在电脑端，如果在移动端，加载时间会更久，体验会更差。 这样的话网站做得再难看批判者进不来不是白搭嘛，于是针对 web 页面加载速度的优化迫在眉睫。

基于博主以前优化过其他网站，于是博主准备把这次的优化过程记录下来分享给大家借鉴。

1. 页面分析

先来看优化前的页面：



加载时间 8.18s ，一共 33 个 请求，加载 1.38MB 。 可以看到对于网速较慢的浏览者光加载资源就需要 5s 以上，再加上 33 个请求切换开销，简直不能愉快的玩耍。 所以接下来的优化手段就要从加载流量和请求数量入手：

2. 优化图片

图片在网络流量中占有很大的比重，因此优化图片对于减少流量有着至关重要的作用。

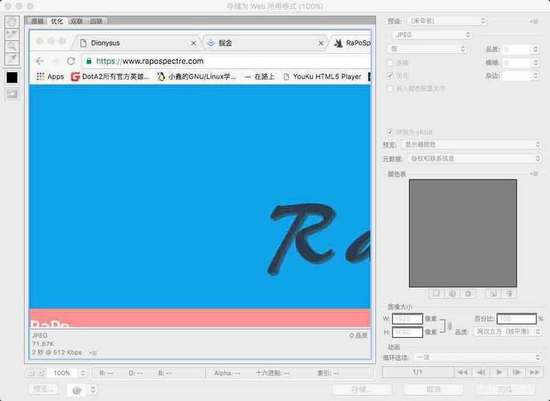
合并小图片:

很多页面有很多小图标，一个一个加载就相当于一个一个请求，将这些小图片合并成一个大图片，用css 控制显示范围，这样就只需要一个请求即可加载完所有小图片，瞬间就会减少很多网络请求。

优化图片格式：

很多图片没有经过优化直接上传到网页中会占用很多额外的流量，比如一张屏幕大小的截图，用截图工具直接截图后的大小大概有 1MB ，此时直接上传到网页中就直接占用了 1MB 流量，但其实我们完全可以只牺牲它 40% 的质量换取缩小 10 倍的大小，网上有很多转化 web 图片的网站，当然如果你有 photoshop 的话完全可以自己导出：

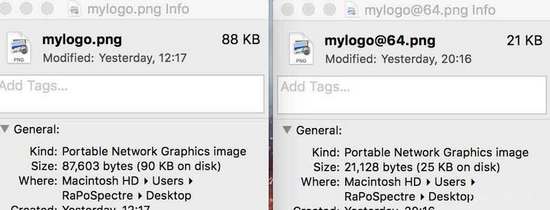
将图片在 ps 中打开，然后点击菜单栏 “文件” 菜单，选择 “储存为 web 所用格式”，出现如下对话框：



一般情况下 jpg 图片选择品质中即可，png 格式图片选择 png8 即可，但注意有透明背景的 png 图片要选择 png24 ，否则透明背景中会出现白边，gif 图片选择 gif64 无仿色即可。

一般经过优化的图片大小至少会有 3倍 之差，图片原大小越大优化的结果会越好。

博主的网站最显眼的图片就是页眉上那个幽鬼的图片啦，所以就先拿它开刀，经过以上步骤优化：



瞬间减小 4 倍， [***实际效果***](https://www.rapospectre.com/) 可以看看，代表着博主门面的图片经过优化后和优化前显示效果并没有明显区别，而文件大小却相差了 4 倍。

3. 使用免费 cdn 加载第三方资源

所有网站都会用到第三方资源，对于第三方资源，如果选择让自己的服务器提供，那么对于小型站点，本就不大的带宽相当一部分还要被公共资源占用，无形之中压缩了服务器带宽，如果把这部分资源让第三方 cdn 提供，那么对于网站加载速度会有不小的提升。

博主选用的是 bootstrap 中文站提供的 [***cdn 静态库***](https://www.bootcdn.cn/) ，博主看过不少国内 cdn 静态库，可以说 bootstrap 家的还是很良心的，更新及时，资源现在也很丰富，基本博主用的三方资源都能在上面找到，于是接下来就是搜索静态资源 ＋ 替换静态资源：

<script src="/s/js/jquery.min.js"></script>

改为

<script src="//cdn.bootcss.com/jquery/3.1.1/jquery.min.js"></script>

这里不要写协议头，让网页自动判断使用 http 还是 https （ 关于 https 网站的部署可以看博主之前的文章： [***给你的网站穿上外衣－ HTTPS 免费部署指南***](https://www.rapospectre.com/blog/https-deploy-guide) ）

4. 使用 cdn 储存静态资源

一般网站 90% 的流量都用于静态资源的加载，除了用免费 cdn 加载第三方资源，还可以自己申请云空间储存自己的静态资源，进一步减小服务器的开销，让服务器只专注于提供数据或者网页渲染服务。 比如博主使用的是 X牛 ，将自己的图片什么都存在 x牛上，每个月都有免费流量，对于个人网站来说应该够用。

5. 合并压缩 js css

除去引用公共库，网页中还有许多自己写的 js 与 css，如果我们直接把开发环境的文件拿来用无疑很浪费流量，因此在编写好网页测试完毕后，我们应该将 css 和 js 压缩合并成一个或者几个文件，这样既减少了请求次数又减少了流量消耗，一箭双雕。 当然还有 html 压缩，不过 ms 现阶段还有一些坑，就先不用了。 说到合并压缩，第一时间播追就想到了 webpack ，前端工程化神器，简单配置一下就可以完全搞定任务：

博主网站自己的 js 工程文件放在 /webroot/static/src/js/ 中，假如我们要将压缩合并后的文件放在 /webroot/static/dist/js/ 中：

在 /webroot/ 下新建文件夹 webpack , 进入文件夹，新建文件 package.json:

{

"name": "RaPo3",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "rapospectre",

"license": "ISC",

"devDependencies": {

"css-loader": "^0.24.0",

"style-loader": "^0.13.1",

"webpack": "^1.13.2",

"webpack-dev-server": "^1.15.1"

}

}

保存后执行：

npm install

*//或*

cnpm install

*//如果你有的话*

然后新建 webpack 配置文件 webpack.config.js:

**var** webpack = **require**('webpack');

module.exports = {

entry: {

base: ['../static/js/src/http.js', '../static/js/stickUp.min.js', '../static/js/src/base.js'],

index: ['../static/js/src/index.js'],

detail: ['../static/js/editormd.js', '../static/js/src/article.js'],

know: ['../static/js/editormd.js', '../static/js/src/know.js'],

**list**: ['../static/js/src/list.js']

},

output: {

path: '../static/js/dist/',

filename: '[name].js'

},

plugins: [

**new** webpack.optimize.UglifyJsPlugin({

output: {

comments: **false**

},

compress: {

warnings: **true**

}

}),

]

}

这里要注意的是如果你的 js 文件间的引用是传统的 html 引入后引用那么在这里合并时记得把你被引用的方法对象等等设置为全局，比如 b.js 要引用 a.js 中的函数 c，合并前要在 a.js 中加上（ 当然如果你一直用 es6/node 写 js 就不用看这里了 ）：

window.c = c;

或

**this**.c = c;

不然 c 就会被当作局部函数封装起来。

改完后运行 webpack 提示成功后看到 dist 目录里已经输出了合并压缩好的文件，之前 12kb 的文件经过压缩合并后只有 6kb 大小，然后我们将其替换到网页中即可。

6. 代码优化

页面代码的优化对于页面加载速度也有不小的影响，最广为人知的：

HTML头部的JavaScript和写在HTML标签中的Style会阻塞页面的渲染，因此CSS放在页面头部并使用Link方式引入，JavaScript的引入放在页面尾

其次还有：

1. 按需加载，把统计、分享等 js 在页面 onload 后再进行加载，可以提高访问速度；
2. 优化 cookie ，减少 cookie 体积；
3. 避免 <img> 的 src 为空；
4. 尽量避免设置图片大小，多次重设图片大小会引发图片的多次重绘，影响性能；
5. 合理使用display属性：

a.display:inline后不应该再使用width、height、margin、padding以及float

b.display:inline-b**lock**后不应该再使用**float**

c.display:block后不应该再使用vertical-align

d.display:**table**-\*后不应该再使用margin或者**float**

1. 不滥用Float 和 web 字体；
2. 尽量使用CSS3动画；
3. 使用 ajax 异步加载部分请求；

7. HTTP2 与 gzip

HTTP2 是以 SPDY 为基础开发的。 SPDY 系列协议由谷歌开发，于 2009 年公开。它的设计目标就是降低 50% 的页面加载时间，所以 HTTP2 在很大程度也是为了优化页面加载时间，同时 HTTP2 支持多路复用，简单说就是所有的请求都通过一个 TCP 连接并发完成。 而 gzip 大家都不陌生，就是一种压缩网页的技术，当然压缩网页进行传输的代价就是给服务器增加一些压缩的负担，当然这种牺牲是值得的。

如何开启 HTTP2 与 gzip？ 博主的网站基于 nginx + uWSGI 进行服务，因此只要在 nginx 开启 HTTP2 与 gzip 就好：

开启 HTTP2

nginx 1.9.5 之后才支持 HTTP2 ，而且需要配置编译参数，关于 nginx 开启 HTTP2 请直接移步博主之前的文章： [***nginx 配置 http2***](https://www.rapospectre.com/blog/https-deploy-guide#h4-3-http2)

开启 gzip

直接打开 nginx 配置文件, 比如博主的在 /etc/nginx/nginx.conf ， 然后加上：

server{

gzip on;

gzip\_comp\_level 6;

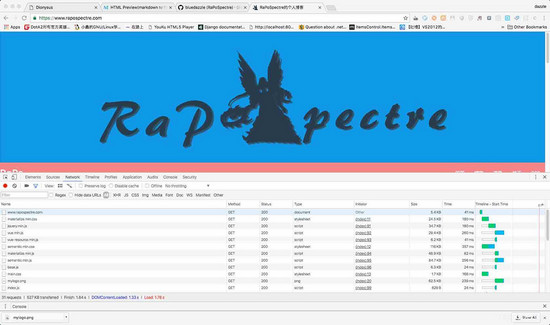
gzip\_proxied any;

gzip\_types text/plain text/css application/json application/javascript application/x-javascript text/xml application/xml application/xml+rss text/javascript application/x-font-woff;

}

然后重启 nginx 即可

最后，让我们清除缓存，再次打开网站：



总加载流量 527kb ，页面完成加载时间 1.84s，对比之前加载时间 8.18s ，1.38MB 流量，整体时间提升了 4 倍多！用手机端访问测试，简直快的飞起，不信你也来[访问]1 试试呀～

最后，附上本文网站的源码以及目录结构，可以通过 commit 记录更加直观的看到优化的过程：

[***https://github.com/bluedazzle...***](https://github.com/bluedazzle/django-vue.js-blog)

欢迎来颗 star 哈哈。

作者： [***rapospectre***](https://www.rapospectre.com/)

## @跟着毛老师学高可用 - 推酷

http://www.tuicool.com/articles/qmAnI3I

时间 2016-09-08 00:53:27  [谢权'blog](http://www.tuicool.com/sites/RZBvYjn)

原文  [https://xiequan.info/ha-（high-availability）/](https://xiequan.info/ha-%EF%BC%88high-availability%EF%BC%89/?utm_source=tuicool&utm_medium=referral)

主题 [高可用性](http://www.tuicool.com/topics/11110042)[网络设备](http://www.tuicool.com/topics/11000139)

HA （High Availability）

(本文根据毛老师（浪客剑心）在b站内部技术分享会总结)

高可用性H.A.（High Availability）指的是通过尽量缩短因日常维护操作（计划）和突发的系统崩溃（非计划）所导致的停机时间，以提高系统和应用的可用性。它与被认为是不间断操作的容错技术有所不同。HA系统是目前企业防止核心计算机系统因故障停机的最有效手段。

一：网络高可用

由于网络存储的快速发展，网络冗余技术被不断提升，提高IT系统的高可用性的关键应用就是网络高可用性，网络高可用性与网络高可靠性是有区别的，网络高可用性是通过匹配冗余的网络设备实现网络设备的冗余，达到高可用的目的。

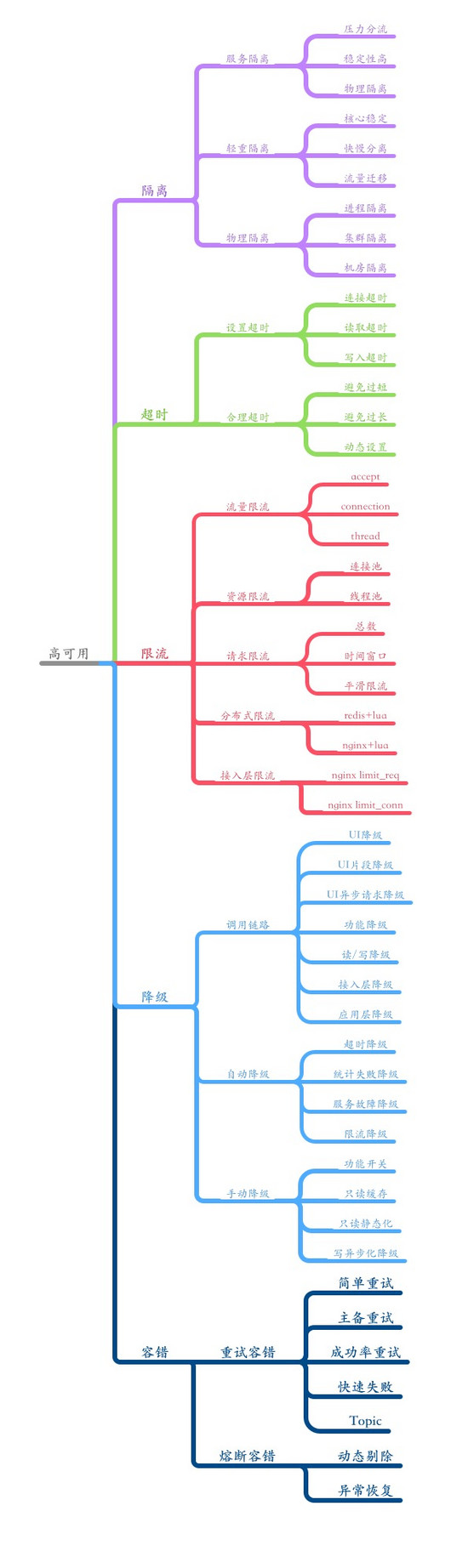
比如冗余的交换机，冗余的路由器等

二：服务器高可用

服务器高可用主要使用的是服务器集群软件或高可用软件来实现。

三：存储高可用

使用软件或硬件技术实现存储的高度可用性。其主要技术指标是存储切换功能，数据复制功能，数据快照功能等。当一台存储出现故障时，另一台备用的存储可以快速切换，达一存储不停机的目的。

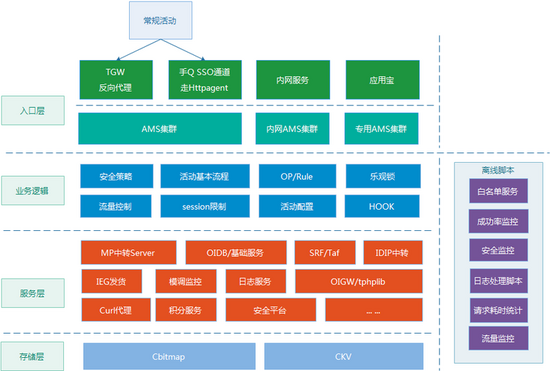


隔离

由于业务的复杂性，常采用服务组件化的架构策略来达到解耦和可扩展的目的。在服务按业务功能切分后，如何提高整个系统的可用性是一个非常重要的问题。文章在对大型网站服务组件化设计架构研究的基础上，结合服务组件之间强弱依赖关系的特点与典型场景，总结了服务隔离技术的基本思想和维度，提供了服务调用解耦与隔离方法、容器级别服务故障隔离方法、组件服务间接依赖故障隔离方法等来提高系统的可用性。

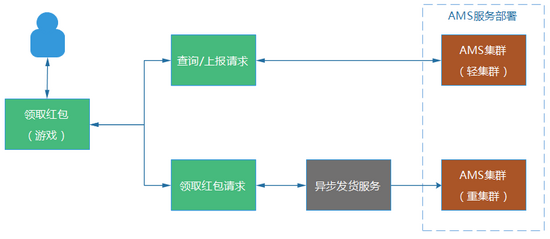
服务解耦、物理隔离

大家都知道一个服务的设计，要尽可能小和分离部署，如此，服务之间的耦合会比较小，一旦某个模块出问题，受到影响的模块就比较少，容错能力就会更强。可是，从设计之初，就将每一个服务有序的切割地很小，这个需要设计者具备超前的意识，能够提前意识到业务和系统的发展形态，而实际上，业务的发展往往是比较难以预知的，因为业务的形态会随着产品的策略的改变而变化。在业务早期流量比较小的时候，通常也没有足够的人力和资源，将服务细细的切分。



轻重分离，物理隔离

以我们支持2016年“手Q春节红包”活动项目的服务集群。就将负责信息查询和红包礼包发货的集群分别独立部署，信息查询的服务相对没有那么重要，业务流程比较轻量级，而红包礼包发货则属于非常核心的业务，业务流程比较重。



轻重分离的这个部署方式，可以给我们带来一些好处：

（1） 查询集群即使出问题，也不会影响发货集群，保证用户核心功能正常。

（2） 两边的机器和部署的服务基本一致，在紧急的情况下，两边的集群可以相互支援和切换，起到容灾的效果。

（3） 每个集群里的机器，都是跨机房部署，例如，服务器都是分布在ABC三个机房，假设B机房整个网络故障了，反向代理服务会将无法接受服务的B机房机器剔除，然后，剩下AC机房的服务器仍然可以正常为外界提供服务。

超时

设置合理超时

调用任何一个服务或者存储，一个合理的超时时间（超时时间，就是我们请求一个服务时，等待的最长时间），是非常重要的，而这一点往往比较容易被忽视。通常Web系统和后端服务的通信方式，是同步等待的模式。这种模式，它会带来的问题比较多。

对于服务端，影响比较大的一个问题，就是它会严重影响系统吞吐率。假设，我们一个服务的机器上，启用了100个处理请求的worker，worker的超时时间设置为5秒，1个worker处理1个任务的平均处理耗时是100ms。那么1个work在5秒钟的时间里，能够处理50个用户请求，然而，一旦网络或者服务偶尔异常，响应超时，那么在本次处理的后续整整5秒里，它仅仅处理了1个等待超时的失败任务。一旦比较大概率出现这类型的超时异常，系统的吞吐率就会大面积下降，有可能耗尽所有的worker（资源被占据，全部在等待状态，知道5s超时才释放），最终导致新的请求无worker可用，只能陷入异常状态。

快慢分离

比如我们的在app-interface项目中需要聚合不同功能的接口，接口的返回时间都不一样，有快有慢。所以我们采用了快慢分离方案

[httpClient]

dial      = "100ms"

timeout  = "500ms"

keepAlive = "60s"

timer    = 128

[httpIm9]

dial      = "100ms"

timeout  = "800ms"

keepAlive = "60s"

timer    = 128

[httpSearch]

dial      = "100ms"

timeout  = "800ms"

keepAlive = "60s"

timer    = 128

[httpWrite]

dial      = "100ms"

timeout  = "800ms"

keepAlive = "60s"

timer    = 128

限流

在开发高并发系统时有三把利器用来保护系统：缓存、降级和限流。缓存的目的是提升系统访问速度和增大系统能处理的容量，可谓是抗高并发流量的银弹；而降级是当服务出问题或者影响到核心流程的性能则需要暂时屏蔽掉，待高峰或者问题解决后再打开；而有些场景并不能用缓存和降级来解决，比如稀缺资源（秒杀、抢购）、写服务（如评论、下单）、频繁的复杂查询（评论的最后几页），因此需有一种手段来限制这些场景的并发/请求量，即限流。

限流的目的是通过对并发访问/请求进行限速或者一个时间窗口内的的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务（定向到错误页或告知资源没有了）、排队或等待（比如秒杀、评论、下单）、降级（返回兜底数据或默认数据，如商品详情页库存默认有货）。

一般开发高并发系统常见的限流有：限制总并发数（比如数据库连接池、线程池）、限制瞬时并发数（如nginx的limit\_conn模块，用来限制瞬时并发连接数）、限制时间窗口内的平均速率（如Guava的RateLimiter、nginx的limit\_req模块，限制每秒的平均速率）；其他还有如限制远程接口调用速率、限制MQ的消费速率。另外还可以根据网络连接数、网络流量、CPU或内存负载等来限流。

限流算法

常见的限流算法有：令牌桶、漏桶。计数器也可以进行粗暴限流实现。

令牌桶算法

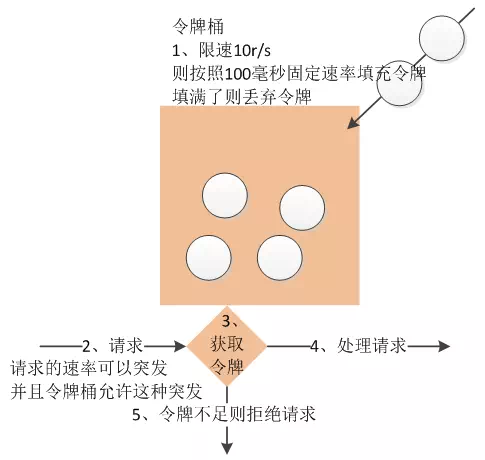
令牌桶算法是一个存放固定容量令牌的桶，按照固定速率往桶里添加令牌。令牌桶算法的描述如下：

假设限制2r/s，则按照500毫秒的固定速率往桶中添加令牌；

桶中最多存放b个令牌，当桶满时，新添加的令牌被丢弃或拒绝；

当一个n个字节大小的数据包到达，将从桶中删除n个令牌，接着数据包被发送到网络上；

如果桶中的令牌不足n个，则不会删除令牌，且该数据包将被限流（要么丢弃，要么缓冲区等待）。



漏桶算法

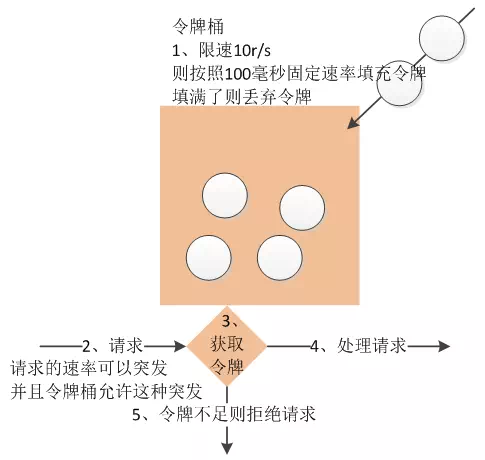
漏桶作为计量工具（The Leaky Bucket Algorithm as a Meter）时，可以用于流量整形（Traffic Shaping）和流量控制（TrafficPolicing），漏桶算法的描述如下：

一个固定容量的漏桶，按照常量固定速率流出水滴；

如果桶是空的，则不需流出水滴；

可以以任意速率流入水滴到漏桶；

如果流入水滴超出了桶的容量，则流入的水滴溢出了（被丢弃），而漏桶容量是不变的。



令牌桶和漏桶对比：

令牌桶是按照固定速率往桶中添加令牌，请求是否被处理需要看桶中令牌是否足够，当令牌数减为零时则拒绝新的请求；

漏桶则是按照常量固定速率流出请求，流入请求速率任意，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝；

令牌桶限制的是平均流入速率（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，4个令牌），并允许一定程度突发流量；

漏桶限制的是常量流出速率（即流出速率是一个固定常量值，比如都是1的速率流出，而不能一次是1，下次又是2），从而平滑突发流入速率；

令牌桶允许一定程度的突发，而漏桶主要目的是平滑流入速率；

两个算法实现可以一样，但是方向是相反的，对于相同的参数得到的限流效果是一样的。

降级

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级。 为了保证用户体验，在资源有限的条件下，我们必须保证关键系统的稳定性。通过对不同业务级别定义不同的降级策略，对除核心主流程以外的功能，根据系统压力情况进行有策略的关闭，从而达到服务降级的目的

（1）拒绝服务

拒绝优先级应用的调用，减少服务调用并发数，保证核心应用正常使用；或者随机拒绝部分请求调用。

（2）关闭功能

关闭不重要的服务，或者服务内部关闭不重要的功能，以节约系统开销，为重要的服务和功能让出资源。

（2）幂等性设计

应用调用服务失败后，会将调用请求重新发送到其他服务器，但是这个失败可能是虚假的，如：处理成功，但未受到相应。

等幂性设计在服务层保证服务重复调用和调用一次产生的结果相同，即服务据有等幂性。

容错

“火”是永远救不完的，让系统能够自动”灭火”，才是解决问题的正确方向。简而言之，系统的异常不能总是依赖于“人”去恢复，让系统本身具备“容错”能力，才是根本解决之道。

重试机制

最容易也最简单被人想到的容错方式，当然就是“失败重试”，总而言之，简单粗暴！简单是指它的实现通常很简单，粗暴则是指使用不当，很可能会带来系统“雪崩”的风险，因为重试意味着对后端服务的双倍请求。

简单重试

我们请求一个服务，如果服务请求失败，则重试一次。假设，这个服务在常规状态下是99.9%的成功率，因为某一次波动性的异常，成功率下跌到95%，那么如果有重试机制，那么成功率大概还能保持在99.75%。而简单重试的缺陷也很明显，如果服务真的出问题，很可能带来双倍流量，冲击服务系统，有可能直接将服务冲垮。而在实际的真实业务场景，往往更严重，一个功能不可用，往往更容易引起用户的“反复点击”，反而制造更大规模的流量冲击。比起服务的成功率比较低，系统直接被冲击到“挂掉”的后果明显更严重。

简单重试，要使用在恰当的场景。或者，主动计算服务成功率，成功率过低，就直接不做重试行为，避免带来过高的流量冲击。

主备服务自动切换

既然单一服务的重试，可能会给该带来双倍的流量冲击，而最终导致更严重的后果，那么我们不如将场景变为主备服务的自动重试或者切换。例如，我们搭建了两套获取openid的服务，如果服务A获取失败，则尝试从服务B中获取。因为重试的请求压力是压到了服务B上，服务A通常不会因为重试而产生双倍的流量冲击。

这种重试的机制，看似比较可用，而实际上也存在一些问题：

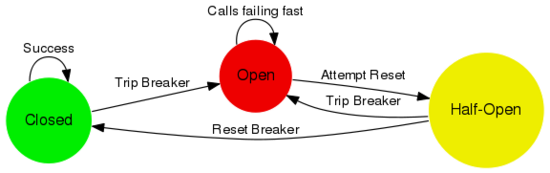
（1） 通常会存在“资源浪费”的问题。因为备份服务系统，很可能长期处于闲置状态，只有在主服务异常的时候，它的资源才会被比较充分地使用。不过，如果对于核心的服务业务（例如核心数据、营收相关）进行类似的部署，虽然会增加一些机器成本和预算，但这个付出通常也是物有所值的。

（2） 触发重试机制，对于用户的请求来说，耗时必然增加。主服务请求失败，然后再到备份服务请求，这个环节的请求耗时就至少翻倍增长，假设主服务出现连接（connect）超时，那么耗时就更是大幅度增加。一个服务在正常状态下，获取数据也许只要50ms，而服务的超时时间通常会设置到500-1000ms，甚至更多，一旦出现超时重试的场景，请求耗时必然大幅度增长，很可能会比较严重地影响用户体验。

（3） 主备服务一起陷入异常。如果是因为流量过大问题导致主服务异常，那么备份服务很可能也会承受不住这种级别的流量而挂掉。

熔断机制

该模式的原理类似于家里的电路熔断器，如果家里的电路发生短路，熔断器能够主动熔断电路，以避免灾难性损失。在分布式系统中应用电路熔断器模式后，当目标服务慢或者大量超时，调用方能够主动熔断，以防止服务被进一步拖垮；如果情况又好转了，电路又能自动恢复，这就是所谓的弹性容错，系统有自恢复能力。下图Fig 8是一个典型的具备弹性恢复能力的电路保护器状态图，正常状态下，电路处于关闭状态(Closed)，如果调用持续出错或者超时，电路被打开进入熔断状态(Open)，后续一段时间内的所有调用都会被拒绝(Fail Fast)，一段时间以后，保护器会尝试进入半熔断状态(Half-Open)，允许少量请求进来尝试，如果调用仍然失败，则回到熔断状态，如果调用成功，则回到电路闭合状态。



跟着毛老师学高可用