1. LB三种架构

**VS/DR**利用大多数Internet服务的非对称特点（上传与下载速度不同），负载调度器只负责调度请求，而服务器直接将响应返回给客户，可以极大地提高整个集群系统的吞吐量。（吞吐量是指在单位时间内[中央处理器](https://baike.baidu.com/item/%E4%B8%AD%E5%A4%AE%E5%A4%84%E7%90%86%E5%99%A8" \t "_blank)（CPU）从存储设备读取->处理->存储信息的量。）在DR模式中，调度器根据各个真实服务器的负载情况，连接数多少等，动态地选择一台服务器，不修改目标IP地址和目标端口，也不封装IP报文，而是将请求报文的数据帧的目标MAC地址改为真实服务器的MAC地址。然后再将修改的数据帧在服务器组的局域网上发送。因为数据帧的MAC地址是真实服务器的MAC地址，并且又在同一个局域网。那么根据局域网的通讯原理，真实服务器是一定能够收到由LB发出的数据包。真实服务器接收到请求数据包的时候，解开IP包头查看到的目标IP是VIP。（此时只有自己的IP符合目标IP才会接收进来，所以我们需要在本地的回环借口上面配置VIP。PS：在环回口上配IP而不在网卡上配是通过BGP引流的话，Loopback口只要Router还健在，则它就会一直保持Active，这样，只要BGP的Peer的Loopback口之间满足路由可达，就可以建立BGP回话，总之BGP中使用loopback口可以提高网络的健壮性。另：由于网络接口都会进行ARP广播响应，但集群的其他机器都有这个VIP的lo接口，都响应就会冲突。所以我们需要把真实服务器的lo接口的ARP响应关闭掉。）然后真实服务器做成请求响应，之后根据自己的路由信息将这个响应数据包发送回给客户，并且源IP地址还是VIP。lvs基本上能支持所有应用，因为lvs工作在4层，所以它可以对几乎所有应用做负载均衡，包括http、数据库、聊天室等等。抗负载能力强。抗负载能力强、性能高，能达到F5硬件的60%；对内存和cpu资源消耗比较低，工作在网络4层，通过vrrp协议转发（仅作分发之用），具体的流量由linux内核处理，因此没有流量的产生。咱们还通过DPDK绕过内核，稳定性、可靠性好，自身有完美的热备方案；（如：LVS+Keepalived）配置 复杂，对网络依赖比较大。

**HAPROXY**

Nginx为支持长连接，需要配置使用upstream 模块的keepalive选项，根据QPS和响应时间计算出需要的长连接量，HaProxy能够补充Nginx的一些缺点比如Session的保持，Cookie的引导等工作。haproxy 将后端服务器产生的session和后端服务器标识存在haproxy中的一张表里。客户端请求时先查询这张表。

疑问：当客户通过TCP连接访问网络访问时，服务所需的时间和所要消耗的计算资源是千差万别的，它依赖于很多因素。例如，它依赖于请求的服务类型、当前网 络带宽的情况、以及当前服务器资源利用的情况。一些负载比较重的请求需要进行计算密集的查询、数据库访问、很长响应数据流；而负载比较轻的请求往往只需要 读一个HTML页面或者进行很简单的计算。

请求处理时间的千差万别可能会导致服务器利用的倾斜（Skew），即服务器间的负载不平衡。例如，有一个WEB页面有A、B、C和D文件，其中D是 大图像文件，浏览器需要建立四个连接来取这些文件。当多个用户通过浏览器同时访问该页面时，最极端的情况是所有D文件的请求被发到同一台服务器。所以说， 有可能存在这样情况，有些服务器已经超负荷运行，而其他服务器基本是闲置着。同时，有些服务器已经忙不过来，有很长的请求队列，还不断地收到新的请求。反 过来说，这会导致客户长时间的等待，觉得系统的服务质量差。

简单连接调度可能会使得服务器倾斜的发生。在上面的例子中，若采用轮叫调度算法，且集群中正好有四台服务器，必有一台服务器总是收到D文件的请求。这种调度策略会导致整个系统资源的低利用率，因为有些资源被用尽导致客户的长时间等待，而其他资源空闲着。

二、

DNS如何实现解析

实际上，DNS解析是通过NSSwitch这个框架实现的，这个框架提供了名称解析的平台，它的工作方式就是通过找寻在此平台下名称解析的工具实现名称解析的，DNS只需要在NSSwitch框架上找寻能够实现名称解析的工具，那这工具就是libnss\_files.so和libnss\_nss.so。NSSwitch这个框架对我们而言展示的其实就是一个配置文件:/etc/nsswitch.conf。

当我们去访问一个主机名时，主机名是没法直接建立联系的，这时候它会调用一个库文件完成从主机名到ip的转换，这个转换机制在本机上就叫stub resolver（名称解析器），它会通过库调用去/etc/nsswitch.conf找查询次序，根据查询次序用相应机制去完成解析。

DNS查询

DNS是分布式数据库，上级仅知道其直接下级，默认下级只知道根的位置（得配置）。它通过分级的方式自顶向下来进行查询的。最上面的一级是根域，第二层是顶级域。

顶级域分为三种：

组织域：.com , .org , .net ,.cc

国家域：.cn , .tw , .hk , .jp

反向域：IP–>FQDN(正向：FQDN–>IP,正向和反向不在同一个数据库)

若是每次查询都去根域，则大部分带宽都会被用来做DNS解析，所以就有了缓存，将查询的结果缓存下来，且缓存到一个公共位置，建立个缓存服务器，服务器专门接收每台主机的访问和请求，这样就能节省带宽。但是服务器缓存的数据是不具有权威的，由主机的直接上级告诉缓存服务器缓存的超时时间。因此，DNS服务器向缓存服务器发送回应时，不止要包含主机与IP的对应，还要包含TTL。TTL时间越长，服务器越空闲，对服务器压力越小。

DNS的查询分为递归查询和迭代查询：

递归查询

递归查询是一种DNS服务器的查询模式，在该模式下DNS 服务器接收到客户机请求，必须使用一个准确的查询结果回复客户机。如果DNS 服务器本地没有存储查询DNS 信息，那么该服务器会询问其他服务器，并将返回的查询结果提交给客户机。客户机只发出一次请求。

迭代查询

DNS 服务器另外一种查询方式为迭代查询，DNS 服务器会向客户机提供其他能够解析查询请求的DNS服务器地址，当客户机发送查询请求时，DNS 服务器并不直接回复查询结果，而是告诉客户机另一台DNS 服务器地址，客户机再向这台DNS 服务器提交请求，依次循环直到返回查询的结果。这种情况下客户机会发出多次请求。

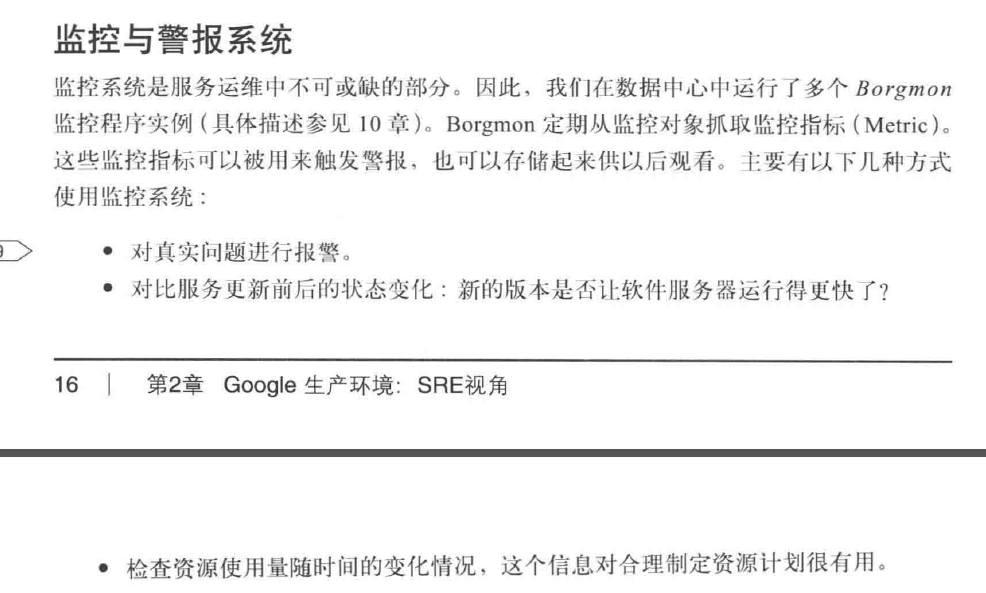
实际情况下，根是不参与递归的。互联网是两段式查询，对于客户端，使用递归查询，即客户端只发出一次请求，而对于服务器，使用迭代查询，即服务器发出多次请求直至得到结果，再将结果返回给客户端。对于服务器来说，一般只给其内部或兄弟主机提供递归服务。

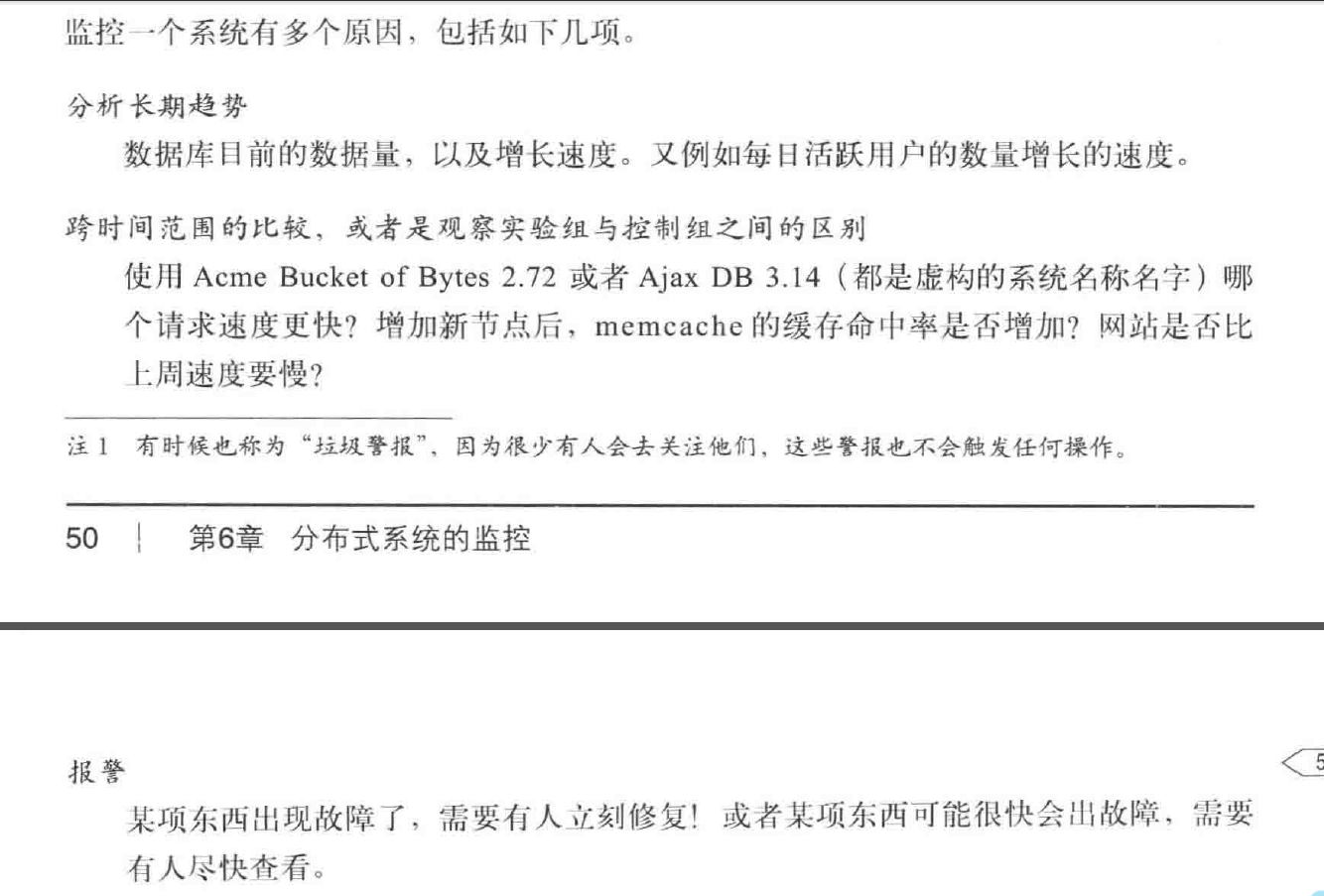
监控

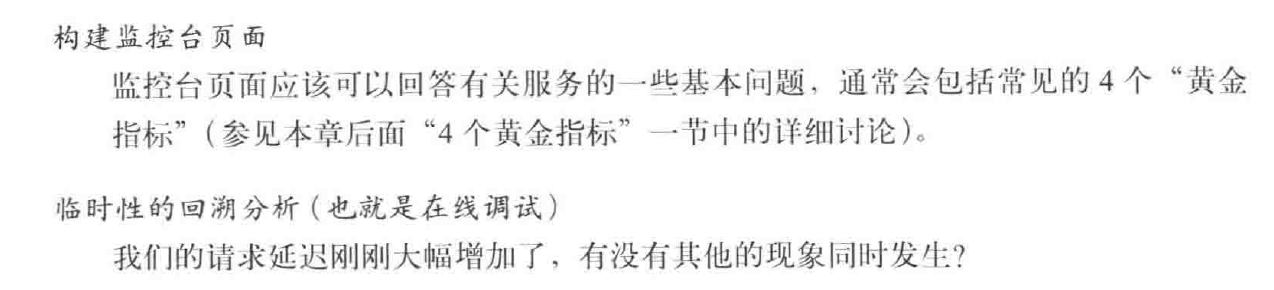
监控系统应当解决两个问题：**现象**（什么东西出故障了？），**原因**（为什么出故障？）

**现象**—— 用户可感知的现象，比如：登陆不了、支付订单变慢；

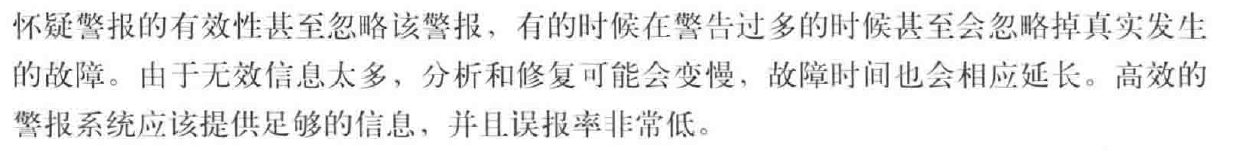
**原因**—— 造成现象的潜在因素，可能只是中间因素或者相关因素，并非根本原因，根本原因需要SRE介入分析并确定。比如：login 服务CPU超过警戒值，订单服务器的CLOSE\_WAIT状态的TCP链接数猛增等等。



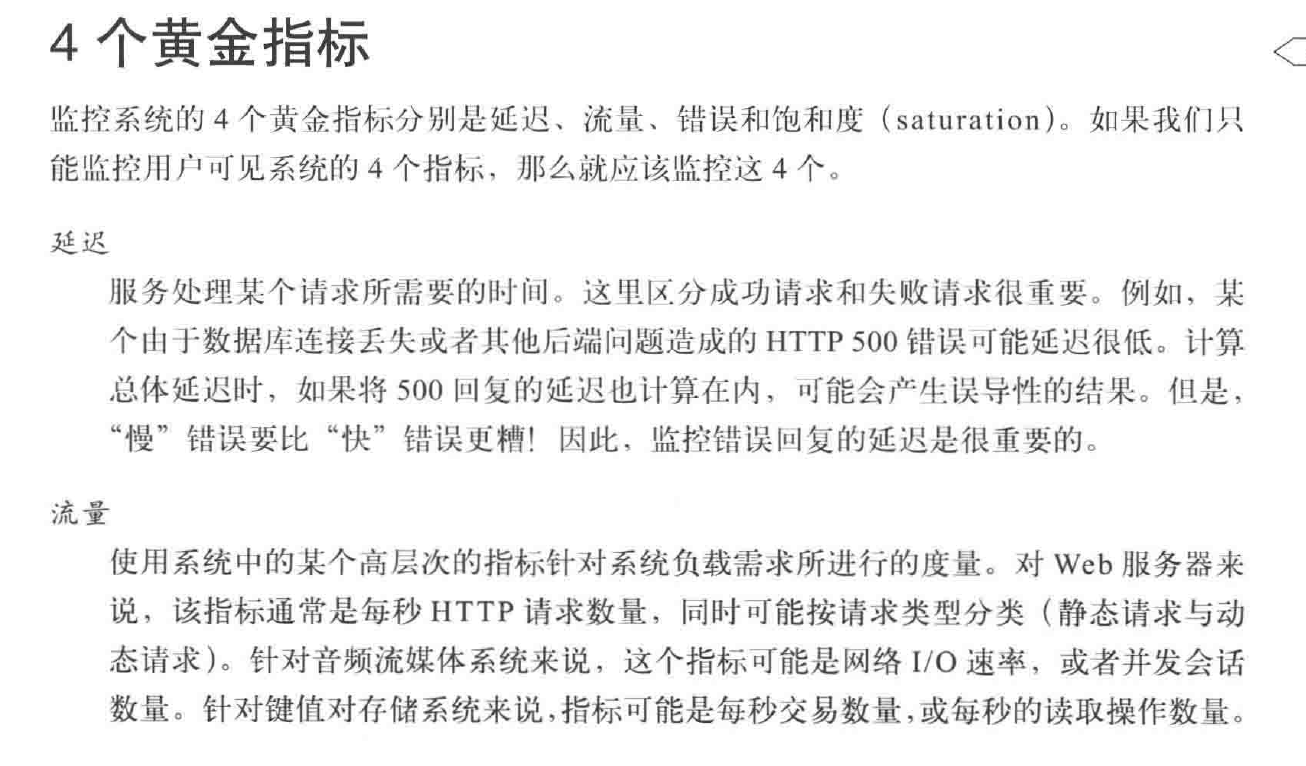


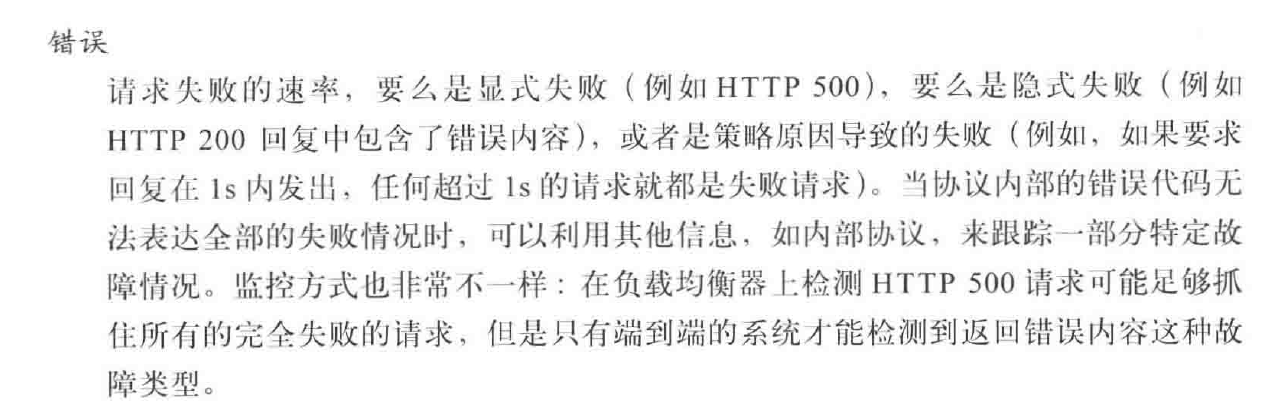


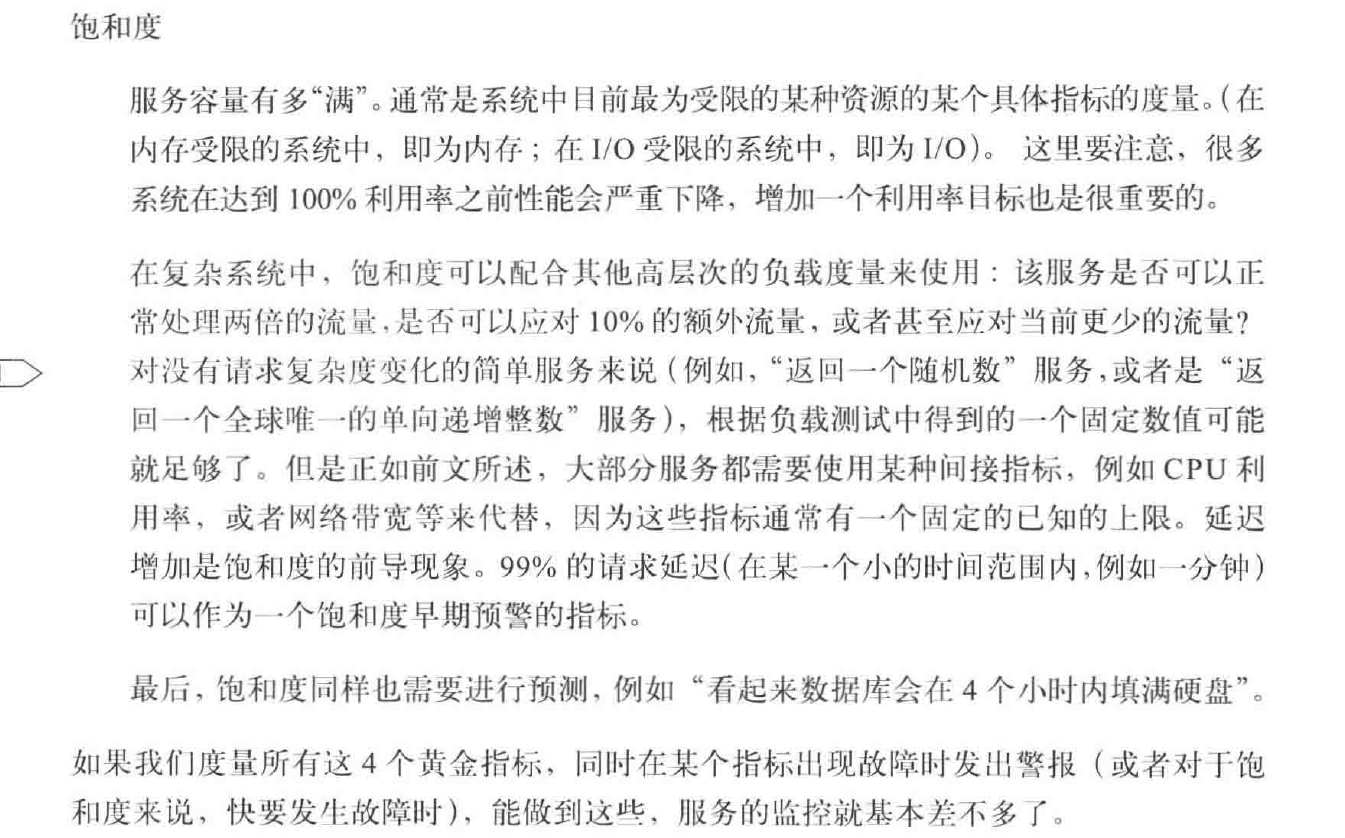
思考：



报警分时间，若多长时间未恢复，才短信提醒。每周做一次报警数据分析

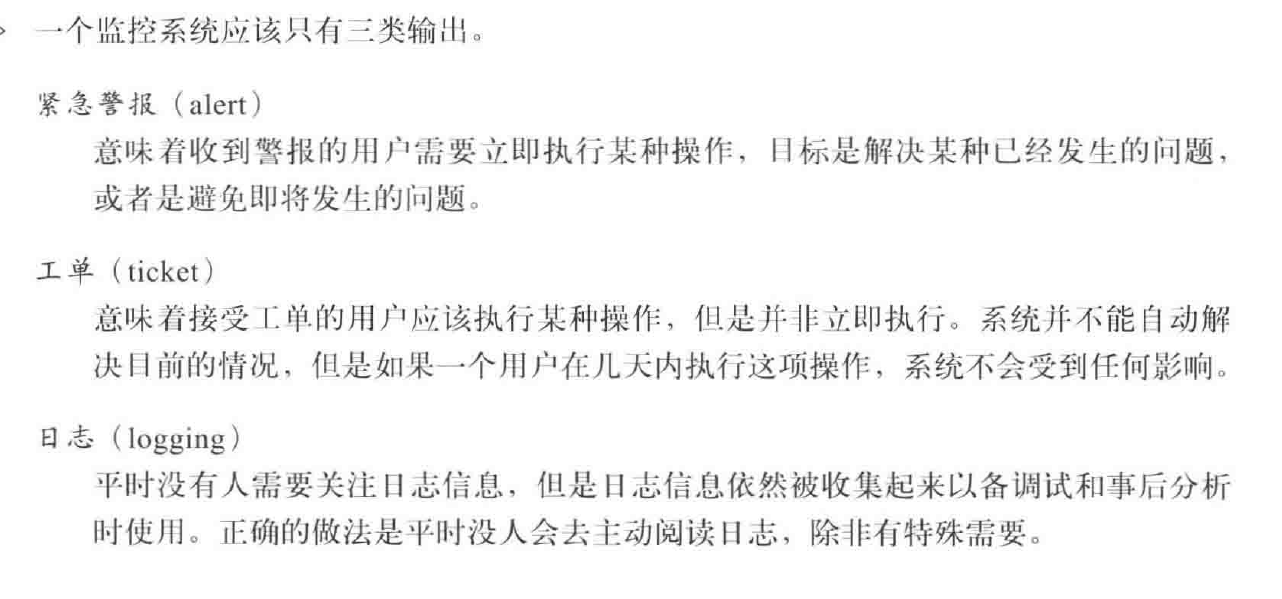


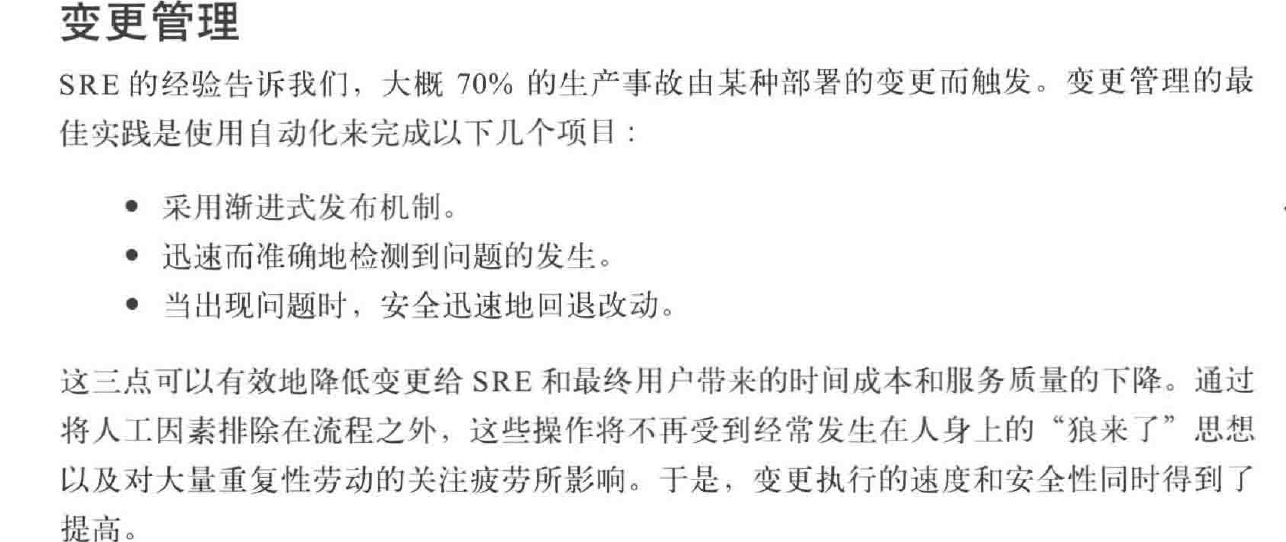


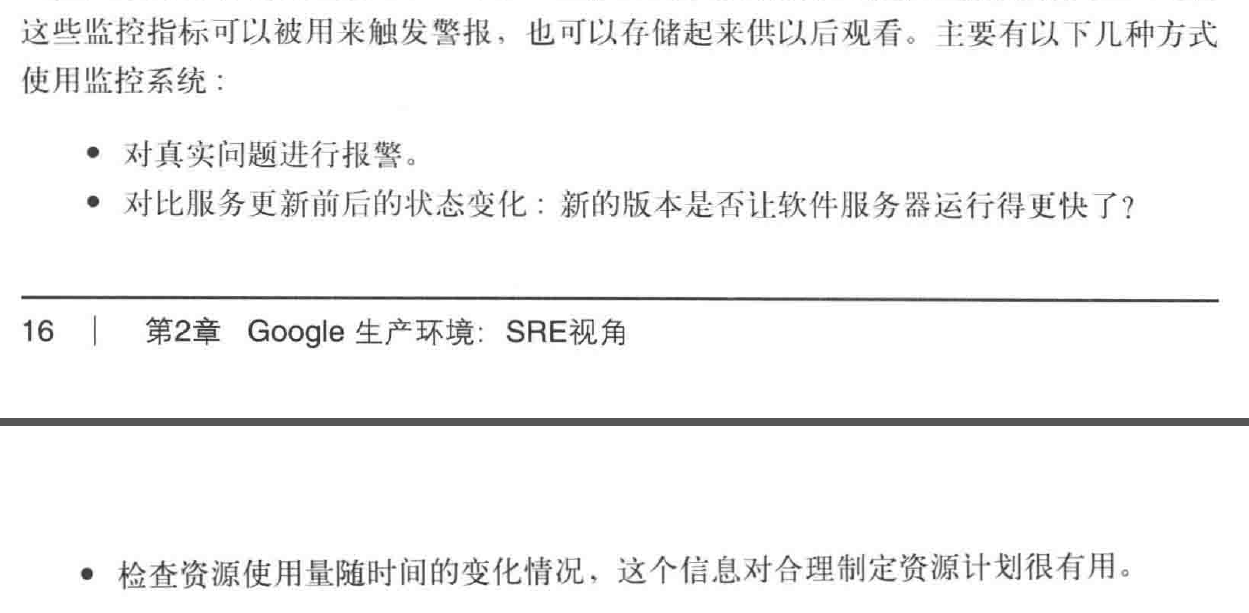


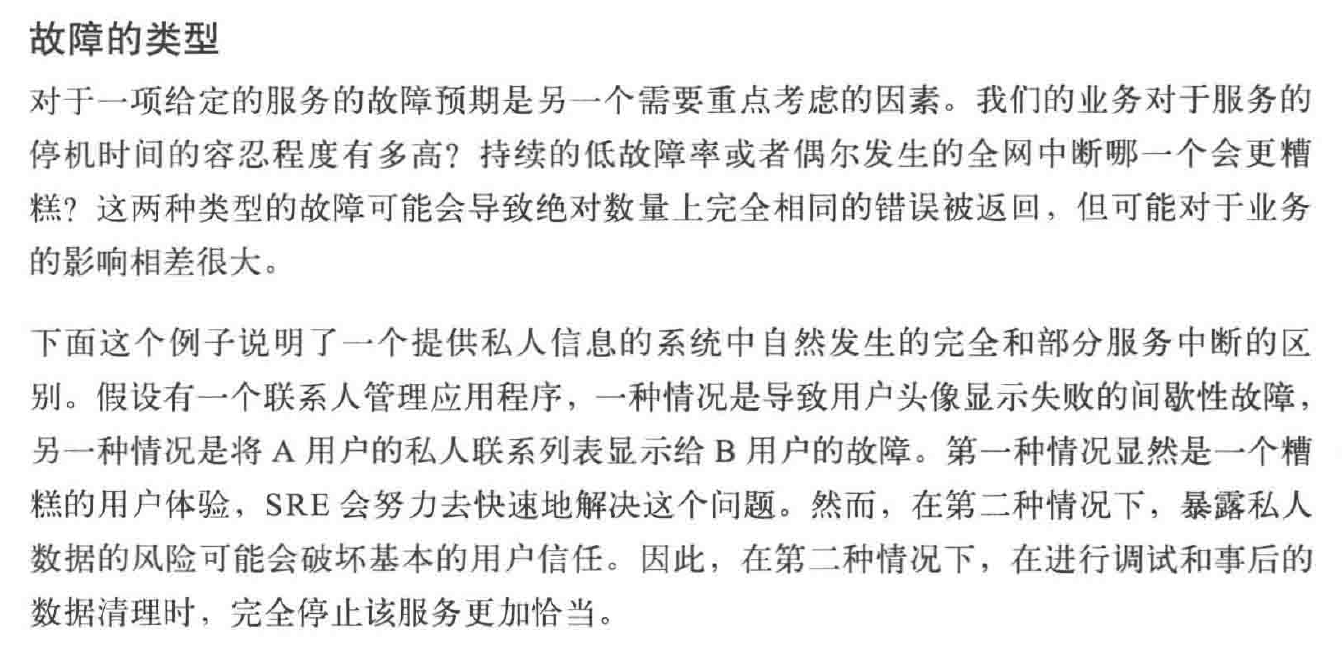
前三个是对服务进行监控，后一个是对服务器进行监控，当然也可以包含容器的状态监控，比如线程池、GC等。











三、SRE

# 拥抱风险

传统运维是厌恶风险的，但是开发和产品却更关注变化速度，他们都希望迭代速度越快越好，但是这回给系统运行带来风险，所以这天生是矛盾。

为了解决风险和变化的矛盾，google提出了SLI-->SLO-->SLA的机制。

**SLI**——服务质量指标，如：延时、吞吐量、错误率、可用性等

**SLO**——服务质量目标，服务的某个SLI的目标值，或者目标范围。比如：SLI<=目标值，min=

**SLA**——服务质量协议（Agreement），服务（SRE）和用户（开发、产品）之间的一个明确的、或者不明确的协议，描述了在达到或者没有达到SLO之后的后果。或者可以转化为先行的KPI，比如系统可用性99.99%等。

开发和运维针对某个系统协商好一个SLA后，大家有一个量化的指标，一旦出现冲突时，算一下，看看是否违反SLA，如果违反，那么就升级走流程。这样既灵活，也有章可循。如果开发团队牛逼，代码质量高或者运气好，你可以迭代快，反之你需要慢点来，间接地，大家都对线上系统负责了。

SRE中定义了一个概念叫SLO(服务质量目标)，通过SLO合理评判一个服务要达成的服务质量。

首先我先说下”故障“这个词，这个词对运维人员来说，是非常不想听到和遇到的。运维人员有一个重要任务是确保服务的稳定，换句话说就是没有故障。

所以我们或多或少谈到“故障”就会色变，遇到故障马上第一时间解决，为了避免下次还出现，我们可能还会开“事故总结会”，优化流程和工具。

其实我们很多时候对于“故障”的理解是简单粗暴的，从一线员工到老板都认为“故障不能有”，“故障必须消除”，我们耗费很大精力“消除了一切故障”，系统平稳运行了，自己也会萌生成就感，感觉干的还不赖。可是并没有进一步去思考一下，故障存在的意义。

我们常见的所谓“99.9%”，“99.99%”的服务可用性，但是并没有使用科学方法来分析和规划业务到底应该3个9还是4个9。

SRE中说到一句话“100%稳定的系统是不存在的”，它把这个做为一个前提，那也意味着系统是肯定要出故障的。

SLO就是用来解决这个事情的，首先服务的故障不可避免，每个服务的级别不同，不可能所有服务都是99.999999，要针对业务的不通特性制定不同的SLO。

比如： 谷歌的企业服务，针对企业用户是有签署服务中断赔偿协议的，那么稳定性要求很高，所以它的SLO级别必须很高。 谷歌的youtube（当时），针对终端用户且版本迭代很快，业务在不断变化和创新，SLO级别可以放低。

SLO的制定通常是产品经理，开发团队，SRE一起协商完成，大家根据业务的规模，产品特性，产品处于的阶段制定。

SLO的制定，我觉得就是科学的面对“故障”这个问题，故障不可避免，不应该以消灭故障为目的，合理的接受它，确保它在SLO标准的范围内，高于这个标准会浪费人力和成本，低于这个标准就需要进行优化。

SLO的制定很大程度在于各个团队之间的协商，大家都有基于数据的科学评判方法，比如产品预估的用户数，产品发版周期，使用带宽等。

中国的国情更多的是拍脑袋，老板的态度，上面的一句话“不能有事故”，那就是99.999999999999999无限，没有科学的进行评估。

SLO解决的问题

通过这样一个SLO，之前很多令人头疼的问题就迎刃而解了。

成本和收益的矛盾

大家都知道维护服务可用性的成本不是线性增长的，到一定程度，增加一个9可能需要10倍100倍的成本，通过SLO让成本和收益取得很好的平衡，假设一个业务增加SLO等级，可以计算一下需要的成本和带来的收益，如果得不偿失就可以不用增加SLO等级。

科学的运维

有了SLO，对于运维工作有了可量化的标准，运维工程师不用每天提心吊胆，生怕出现故障，只要故障在SLO范围内就是可接受的，节省出很多精力用在更重要的事情上。

稳定和创新的矛盾

大家都知道运维工程师最不喜欢的就是“线上变更”，一个服务如果不做变更一般都是很稳定的，问题往往出现在变更上。  
可是一个新业务往往需要大量变更，不停的迭代创新。  
这个时候运维会说：别做变更了，稳定是第一位的，出了故障，我们得背锅。  
开发会说：我们得变更，这样才有新功能，才能获取更多用户啊。  
矛盾因此产生了。  
通过SLO很好的解决了这个矛盾，我们先一起给这个业务制定好SLO的等级，如果是需要频繁的变更的，可能SLO等级就会低一些。  
这样在满足业务创新的需求上，只要在SLO范围内，就认为业务是稳定的。  
反之，如果变更太频繁，使故障率超出了SLO可接受的范围，可以要求开发调低变更频率，或者重新制定SLO等级。  
这样就解决了业务既要“稳定”又要“创新“的矛盾。