# 大规模服务设计部署经验谈

本文中提出的最佳实践，来自于作者多年大规模服务设计和部署的经验，为设计、开发对运营友好的服务提供了一系列良好的解决方案。

■ 文 / James Hamilton 译 / 赖嶲翔

## 1 引言

本文就设计和开发运营友好的服务的话题进行总结，得出一系列最佳实践。设计和部署大规模服务是一个高速发展的领域，因而随着时间的流逝，任何最佳实践集合都可能成熟并完善。我们的目的是为了帮助人们：

- 快速交付运营友好的服务；
- 避免清早电话铃声的骚扰，帮助备受运营不友好的服务侵扰的客户尽量摆脱窘境。

这篇论文是我们在过去的20年中在大规模以数据为中心的软件系统和互联网级大规模服务的智慧结晶，包括Exchange Hosted Services团队、Microsoft Global Foundation Services Operations团队以及Windows Live! 平台多个团队的经验。这些贡献经验的服务中，有不少规模已经增长到拥有超过二亿五千万名用户。同时，本论文也大量吸取了加州大学伯克利分校在面向恢复计算（Recovery Oriented Computing）方面取得的成果和斯坦福大学在只崩溃软件（Crash-Only Software）方面的研究经验。

Bill Hoffman为本论文贡献许多最佳实践。此外，他还提出三条简单原则，值得大家在进入正题之前进行考量：

1. 做好发生故障的心理准备。
2. 保持简单化。
3. 将所有的工作自动化。

这三条原则形成了贯穿后续讨论的主轴。

本文的十个小节涵盖了设计和部署运营友好服务所必须做到的各个方面。它们是：整体服务设计；以自动化和预置（Provisioning）为目标进行设计；依赖关系管理；发布周期及测试；硬件的选择和标准化；运营和容量规划；审核、监控和警报；体面降级和管理控制；客户及媒体沟通计划；以及客户自我预置和自我帮助。

## 2 整体服务设计

一直以来，人们都相信80%的运营问题源于设计和开发，因此本节关于整体服务设计的内容篇

幅最长，也最重要。系统出故障时，人们很自然倾向于首先去审视运营工作，因为这是问题实际产生的地方。不过，绝大多数运营问题都可以归因于设计和开发，或者最适合在设计和开发中解决。随后的内容凸显一个共识，即在服务领域，将开发、测试和运营严格分离不是最有效的方式。在环顾众多服务之后， 我们发现了这样一个趋势——管理的低成本与开发、测试和运营团队间协作的紧密程度密切相关。

除了在这里讨论的服务设计最佳实践以外，随后一节“以自动化管理和预置为目标进行设计”对服务设计也有实质性的影响。有效的自动化管理和预置通常以一个受限的服务模型来实现。简单是高效率运营的关键，

这是贯穿本文重复出现的主题。在硬件选择、服务设计和部署模型上的理性约束，是降低管理成本和提高服务可靠性的强心针。

在运营友好的基础原则中，为整体服务设计带来最大影响的几条包括：

## 2.1 设计时为故障做好准备（Design for failure）。

在开发包含多个协同运作的组件的大型服务时，这是一条核心概念。这些组件会有故障发生，而且故障的产生很频繁。它们之间不会总是稳定地协作，也不会单独出现故障。一旦服务的规模达到10,000台以上的服务器和50,000块以上的磁盘，每天就会有多次故障发生。如果一有硬件故障产生就得采取紧急措施来应对，那么服务就无法以合理的成本可靠地伸缩。整个服务必须有承受故障而无须人工干预的能力。故障恢复的步骤必须非常简单，而且这些步骤应当进行频繁测试。斯坦福大学的Armando Fox 主张说，对故障恢复步骤进行测试的最佳方法，就是绝对不要用正常方式使服务停机，用粗暴的方式让它停转就可以了。乍听起来怎么做是违背直觉的，但是如果没有频繁使用故障步骤，那么它们在真正临阵时就可能溃不成军 。

## 2.2 冗余和错误恢复（Redundancy and fault recovery）。

大型机模型是指购买一台价高块头大的服务器。大型机拥有冗余的电源供应，CPU可以热交换，而且总线架构也超乎寻常，使得这样一个紧密耦合的系统能够有可观的I/O吞吐量。这样的系统，最明显的问题就是它们的成本；而且即便有了所有这些费用高昂的设计，它们仍然不够可靠。为了达到99.999%的可靠性，冗余是必须存在的。实际上，在一台机器上实现4个9的可靠性都是相当困难的。这个概念在整个业界都耳熟能详，不过，将服务构建在脆弱而又非冗余的数据层之上的现象，到目前为止都屡见不鲜。要保证设计的服务其中的任何系统可以随时崩溃（或者因为服务原因被停止）但又仍然能符合服务水平协定（Service Level Agreement， 简称SLA），是需要非常仔细的设计的。保证完全遵守这项设计原则的严格测试（acid test） 步骤如下：首先， 运营团队是否有意愿并且有能力随时让任意一台服务器停机并且不会让负载被榨干？如果答案是确定的，那么肯定存在同步冗余（无数据丢失）、故障侦测和自动接管。我们推荐一条普遍使用的设计方法，

用于查找和纠正潜在的服务安全问题：安全威胁建模（Security Threat Modeling）。在安全威胁建模中，我们要考虑每一条潜在的安全威胁，并且相应实现恰当的缓和方案。同样的方法也适用于以错误适应和恢复为目标的设计。

将所有可以想象到的组件故障模式及其相应组合用文档记录下来。要保证服务在每个故障发生后都能继续运行，且不会在服务质量上出现不可接受的损失；或者判断这样的故障风险对于这样一个特定的服务是否可以接受（例如，在非地理冗余的服务中损失掉整个数据中心）。我们可能会认定某些非常罕见的故障组合出现的可能性微乎其微，从而得出确保系统在发生这种故障之后还能继续运行并不经济的结论。但是，在做这样的决定时请谨慎从事。在运行数以千计的服务器的情况下，每天都会有几百万种组件故障产生的可能，这时那些事件的“罕见”组合亮相的频繁程度，足以让我们瞠目结舌。小概率组合可能变成普遍现象。

## 2.3 廉价硬件切片（Commodity hardware slice）。

服务的所有组件都应当以廉价硬件切片为目标。例如，存储量轻的服务器可以是双插槽的2至4核的系统，带有启动磁盘，价格在1,000 至2,500 美元之间；存储量大的服务器则可以是带有16至24个磁盘的类似服务器。主要的观察结果如下：

- 大型的廉价服务器集群要比它们替代的少数大型服务器便宜得多；
- 服务器性能的增长速度依然要比I/O性能的增长速度快很多，这样一来，对于给定容量的磁盘，小型的服务器就成为了更为稳定的系统；
- 电量损耗根据服务器的数量呈线性变化，但随系统时钟频率按立方级别变化，这样一来性能越高的机器运营成本也越高；
- 小型的服务器在故障转移（Fail over）时只影响整体服务工作负荷的一小部分。

## 2.4 单版本软件（Single-version software）。

使某些服务比多数打包产品开发费用更低且发展速度更快的两个因素是：

- 软件只需针对一次性的内部部署。
- 先前的版本无须得到十年的支持——针对企业的产品正是如此。相对而言，单版本软件更容易实现，附带客户服务，特别是无须费用的客户服务。但是在向非客户人员销售以订阅为基础的服务时，单版本软件也是同样重要的。企业通常习惯在面对他们的软件提供商时拥有重要的影响力，并且在部署新版本时（通常是个缓慢的过程），他们会习惯性想去掌握全部的控制权。这样做会导致他们的运营成本和支持成本急剧上升，因为软件有许多版本需要得到支持。最经济型的服务是不会把对客户运行的版本的控制权交给他们的，并且通常只提供一个版本。要把握好单一版本软件的产品线，必须：
- 注意在每次发布之间不要产生重大的用户体验变更。

- 愿意让需要相应级别控制能力的客户可以在内部托管，或者允许他们转向愿意提供这类人员密集型支持服务的应用服务提供商。

## 2.5 多重租赁（Multi-tenancy）。

多重租赁是指在同一个服务中为该服务的所有公司或最终用户提供主机服务，且没有物理上的隔离；而单一租赁（Single-tenancy）则是将不同组别的用户分离在一个隔离的集群中。主张多重租赁的理由基本上和主张单版本支持的理由一致，且它的基础论点在于提供从根本上成本更低、构建在自动化和大规模的基础之上的服务。

回顾起来，上面我们所展示的基本设计原则和思考如下：

- 设计时为故障做好准备
- 实现冗余和错误恢复
- 依靠廉价硬件切片
- 支持单一版本软件
- 实现多重租赁

我们约束服务设计和运营的模型，以此最大化自动化的能力，并且减少服务的总体成本。在这些目标和应用服务提供商或IT 外包商的目标之间，我们要划一道清楚的界限。应用服务提供商和IT外包商往往人员更加密集，并且更乐于运行面向客户的复杂配置。

## 2.6 设计运营友好的服务的实践

设计运营友好的服务更具体的最佳实践包括：

### 2.6.1 快速服务健康测试。

这是构建验证测试的服务版本。这是一个嗅探型测试，可以快速在开发者的系统上运行，以保证服务不会以独立方式出错。要保证所有的边界条件都被测试到，是不可能的，但如果快速健康测试通过的话，那么代码就可以检入了。

### 2.6.2 在完整的环境中开发。

开发人员不但应当对他们的组件进行单元测试，而且还要对出现组件变更的整个服务进行测试。要高效实现这个目标，必须得有单服务器的部署，以及前一条最佳实践——快速的服务健康测试。

### 2.6.3 对下层组件的零信任。

设想下层组件会出现故障，并且确保组件会有能力恢复并继续提供服务。恢复的技巧和服务相关，但常见的技巧包括：

- 在只读模式中依靠缓存的数据继续运转；
- 在服务访问故障组件的冗余拷贝的短暂时间内，继续向用户群一小部分的所有人提供服务。

### 2.6.4 不要把同一个功能构建在多个组件中。

预见未来的交互是一件极其困难的事情，如果不慎引入冗余的代码，那么将来就不得不在系统的多处做修复。服务的增长和发展是很快的，稍不留神代码的质量就会很快恶化。

### 2.6.5 不同的集群之间不能互相影响。

大多数服务由小型集合或者系统的子集群组成，它们之间相互协作，共同提供服务，其中每个小型集合都可以相对独立地运作。每个小型集合应达到接近100%的独立程度，且不会有跨群的故障。全局服务，甚至包括冗余，是故障的中心点。有时候，这样的问题是不可避免的，但还是要设法保证每个集群都拥有各自所需的资源。

### 2.6.6 允许（少量）在紧急情况的人工干预

见场景是在灾难性事件或者其他紧急情况下移动用户数据。把系统设计成完全无须人工交互，但也要清楚小概率事件可能会发生，其中组合的故障或者未预期的故障都会需要人工交互。这些事件是会发生的，而在这些情况下，操作员的错误往往是导致灾难性数据丢失的常见来源。一名在半夜2点顶压工作的运营工程师可能会犯错误。将系统设计成一开始在多数情况下无须运营干预，但请和运营团队协作制定在他们需要进行干预时的恢复计划。比起将这些计划写进文档，变成多步骤易出错的过程，更好的办法是把这些规则写成脚本，并在生产环境中进行测试，以确保它们正常工作。没有经过产品环境试验的代码是不可行的，因此运营团队应当定时指挥使用这些工具进行“防火演习”。如果演习的服务可用性风险非常高，那么可以认为之前在工具的设计、开发和测试上的投资不足。

### 2.6.7 保持一切简单健壮。

复杂的算法和组件交互会给调试和部署带来成倍困难。简单到近乎傻瓜式的结构在大规模服务中几乎总是更胜一筹， 因为在复杂的优化工作交付之前， 交互中故障模式的数量早就足以磨灭人们的信心。通常我们的惯例是， 能够带来一个数量级以上改善的优化工作才值得考虑， 而只有百分之几或者甚至于只是低系数级别的提升，就不值得了。

### 2.6.8 全面推进准入控制。

所有良好的系统会在设计时开门见山地引入许可控制，这样符合一条长期以来为人们所认可的原则，那就是避免将更多的工作引入一个已经过载的系统，要比持续接受工作然后开始翻来覆去地检查好一些。在服务入口引入某些形式的节流或者准入控制是很常见的做法，但在所有的主要组件边界上都应该有准入控制。工作性质的变更最终会导致子组件的过载，即使整体服务仍然运行在可接受的负载级别。总体的惯例就是尝试采用优雅降级的方式，而不用在统一给所有用户低质量服务之前进行硬停机并阻断服务的入口。

### 2.6.9 给服务分区。

分区应当可以无限调整，并且高度细粒度化，并且不受任何现实实体（人、集合等）的限制。如果按公司分区，那么对于大的公司，就有可能超过单个分区的规模；而如

果按名称前缀进行分区，那么例如所有以P打头的最终一台服务器就可能会装不下。我们推荐在中间层使用一张查询表，将细粒度的实体，通常是在设计大规模服务的目标和应用服务提供商或IT外包商的目标之间，要划一道清楚的界限。应用服务提供商和IT 外包商往往人员更加密集，并且更乐于运行面向客户的复杂配置。

用户，映射到其数据相应被管理的系统上。这些细粒度的分区随后就可以自由在服务器之间移动。

### 2.6.10 理解网络的设计。

提早进行测试，了解机柜内的服务器之间、跨柜的服务器之间以及跨数据中心之间有哪些负载。应用程序开发人员必须理解网络的设计，且设计应当尽早交给来自运营团队的网络专员审核。

## 2.7 对吞吐量和延迟进行分析。

应当对核心服务的用户交互进行吞吐量和延迟的分析，从而了解它们的影响。结合其他运营操作，比如定期数据库维护、运营配置（加入新用户，用户迁移）和服务调试等，进行吞吐量和延迟的分析。这样做对于捕捉由周期性管理任务所带动的问题是颇有裨益的。对于每个服务，都应当形成一个度量标准，用于性能规划，比如每个系统的每秒用户访问数，每个系统的并发在线人数，或者某些将关联工作负载映射到资源需求的相关度量标准。

## 2.8 把运营的实用工具作为服务的一部分对待。

把运营的实用工具作为服务的一部分对待。由开发、测试、项目管理和运营所产生的运营实用工具应当交给开发团队进行代码审查，提交到主源码树上，用同一套进度表跟踪，进行同样的测试。

最频繁出现的现象是，这样的实用工具对于任务有至关重要的影响，但几乎都没有经过测试。

### 2.9 理解访问模式。

理解访问模式。在规划新特性时，一定要记得考虑它们会给后端存储带来哪些负载。通常，服务模型和服务开发人员与存储抽象得非常开，以至于他们全然忘记了它们会给后端数据库带来的额外负载。对此的最佳实践把它作为规范建立起来，里面可以有这样的内容：“这项功能会给基础结构的其他部分带来什么样的影响？”然后在这项特性上线之后，对它进行负载的测量和验证。

### 2.10 让所有工作版本化。

做好在混合版本的环境中运行的准备。我们的目标是运行单版本的软件，但是多个版本可能会在首次展示和生产环境测试的过程中并存。所有组件的n 版和n+1版都应当能够和平共存。

#### 2.10.1 保持最新发布版的单元/功能测试。

这些测试是验证n － 1 版本的功能是否被破坏的重要方法。我们推荐大家更进一步，定期在生产环境中运行服务验证（后面会详细介绍）。

#### 2.10.2 避免单点故障。

单点故障的产生会导致整个服务或者服务的某些部分停工。请选择无状态的实现。不要将请求或者客户端绑定在特定的服务器上，相反，将它们负载均衡在一组有能力处理这样负载的服务器上。静态哈希或者任何静态的服务器负载分配都会时不时遭受数据和/ 或查询倾斜问题的困扰。在一组机器可以互换时，要做到横向伸展是非常容易的。数据库通常会发生单点故障，而数据库伸缩仍然是互联网级大规模服务的设计中最为困难的事情之一。良好的设计一般会使用细粒度的分区，且不支持跨分区操作，以在多台数据库服务器之间进行高效伸展。所有的数据库状态都会进行冗余存储（至少在一台）完全冗余的热待机服务器上，并且在生产环境中会频繁进行故障转移测试。

## 3 自动管理和预置

许多服务编写的目的是为了在故障时向运营部门发出警报，以得到人工干预完成恢复。这种模式凸显出在24 x 7 小时运营人员上的开支问题；更重要的是，如果运营工程师被要求在充满压力的情况下做出艰难的决定，那么有20%的可能他们会犯错误。这种模式的代价高昂，容易引入错误，而且还会降低服务的整体可靠性。然而，注重自动化的设计会引入明显的服务模型约束。比如说，当前某些大型服务依靠的数据库系统，会以异步方式复制到次级备份服务器，因为复制以异步方式完成，在主数据库无法服务请求后，故障转移到次级数据库会引起部分客户数据丢失。然而，不把

故障转移到次级数据库，则会引起数据被储存在故障服务器上的用户面临服务停工。在这种情况下，要自动化故障转移的决策就很困难了，因为这个决策依赖于人为判断，以及对数据损失量和停机大致时长相比的准确估计。注重自动化设计出的系统会在延迟和同步复制的吞吐量开销上付出代价。在做到这一步以后，故障转移变成了一个很简单的决策：如果主服务器宕机，将请求转到次服务器上。这种方式更适用于自动化，而且被认为更不容易产生错误。在设计和部署后将服务的管理过程自动化可能是一件相当具有难度的工作。成功的自动化必须保证简单性，以及清晰并易于确定的运营决策；这又要依靠对服务的谨慎设计，甚至在必要时以一定的延迟和吞吐量为代价作出牺牲，让自动化变得简单。通常这样的折中方案并不容易确定，但是对于大规模服务来说在管理方面的节约可能不止数量级。事实上，目前根据我们的观察，在人员成本方面，完全手动管理的服务和完全自动化的服务之间的差别足足有两个数量级。注重自动化的设计包含以下最佳实践：

## 3.1 可以重启动，并保持冗余。

可以重启动，并保持冗余。所有的操作都必须可以重新启动，并且所有持久化状态也必须冗余存储。

## 3.2 支持地理分布

支持地理分布。所有大规模服务都应当支持在多个托管数据中心运行。我们所描述的自动化和绝大多数功效在无地理分布的情况下仍是可行的。但缺乏对多服务中心部署方式的支持， 会引起运营成本显著提升。没有了地理分布， 很难使用一个数据中心的空闲容量来减缓另外一个数据中心所托管的服务的负载。缺乏地理分布是一项会导致成本提高的运营约束。

## 3.3 自动预置与安装。

自动预置与安装。手动进行预置和安装是相当劳民伤财的，故障太多，而且微小的配置差异会慢慢在整个服务中蔓延开来，导致问题的确定越来越困难。

## 3.4 将配置和代码作为整体。

将配置和代码作为整体。请确保做到：

- 开发团队以整体单元的形式交付代码和配置；
- 该单元经过测试部门的部署，并严格按照运营部门将会部署的方式；
- 运营部门也按照整体单元的方式部署。通常说，将配置和代码作为一整个单元处理并且只把它们放在一起修改的服务会更可靠。

## 3.5 生成环境的变更必须审核和记录

如果配置必须在生产环境中变更，那么请保证所有的变更都要产生审核日志记录，这样什么东西被修改，在什么时候被谁修改，哪些服务器受到影响，就一目了然了。频繁扫描所有的服务器，确保它们当前的状态与预期状态相符。这样做对捕获安装和配置故障颇有裨益，而且能在早期侦测到服务器的错误配置，还能找到未经审核的服务器配置变更。

## 3.6 管理服务器的角色或者性质，而不是服务器本身

管理服务器的角色或者性质，而不是服务器本身。每个系统的角色或性质都应当按需要支持尽可能多或少的服务器。

## 3.7 多系统故障是常见的。

多系统故障是常见的。请做好多台主机同时发生故障的准备（电源、网络切换和首次上线）。遗憾的是，带有状态的服务必须得注意它们的拓扑分布。有相互联系的故障一直以来都是不可避免的。

## 3.8 在服务级别上进行恢复。

在服务级别上进行恢复。在服务级别处理故障，相比软件底层来说，其中的服务执行上下文更加完整。例如，将冗余纳入服务当中，而不是依靠较低的软件层来恢复。

## 3.9 对于不可恢复的信息，绝对不要依赖于本地存储。

对于不可恢复的信息，绝对不要依赖于本地存储。保证总是复制所有的非瞬时服务状态。

## 3.10 保持部署的简单性

保持部署的简单性。文件复制是最理想的方式，因为这样能带来最大的部署灵活性。最小化外部依赖性；避免复杂的安装脚本；避免任何阻止不同组件或者同一个组件不同版本在同一台机器运行的情况。

## 3.11 定期使服务停转

定期使服务停转。停掉数据中心，关闭柜式服务器，断掉服务器电源。定期进行受控的关闭操作，能够主动暴露出服务器、系统和网络的缺陷。没有意愿在生产环境中测试的人，实际上是还没有信心保证服务在经历故障时仍能继续运转。此外，若不进行生产测试，在真正出事时，恢复不一

定能派上用场。■

# 4 依赖管理

在大规模服务中，依赖管理这个话题通常得不到应有的关注。一般的准则是，对于小型组件和服务的依赖关系，对于判断管理它们的复杂性来说，并不足以节约成本。在以下情况中，依赖关系存在重要意义：

1．被依赖的组件在大小和复杂度上有重要价值；

2．被依赖的服务在作为单一的中央实例时存在价值。

第一类的例子有存储和一致性算法（consensus algorithm） 的实现。

第二类的例子包括身份和群组管理系统。这些系统的整体价值在于它们是一个单一且共享的实力，因此使用多实例来避免依赖关系就不是可选方案。假定要根据上面的标准判断依赖关系，那么用来管理它们的最佳实践有：

## 4.1 为延迟做好准备。

为延迟做好准备。对外部组件的调用可能需要很长时间才能完成。不要让一个组件或者服务中的延迟在完全不相关的领域中引发延迟；确保所有的交互都存在长度合适的超时时长，避免资源阻塞更长的时间。运营等幂性允许请求在超时后重启，即便这些请求可能已经部分或完全完成。确保所有的重启操作都得到报告，并给重启操作设定界限，从而避免反复故障的请求消耗更多的系统资源。

## 4.2 隔离故障。

网站的架构必须能防止层叠的故障，要总是“快速失败（failfast）”。当依赖服务出现故障时，把这些服务标注为停机，并停止使用这些服务，以避免线程因等待故障组件而阻塞。

## 4.3 使用已经交付的历经考验的组件。

使用已经交付的历经考验的组件。历经考验的技术通常总是要比大胆前卫地走在潮流尖端运行要好很多。稳定的软件要优于新版本的早期，不管新特性如何有价值。这条原则也适用于硬件。批量生产的稳定硬件，往往要比从早期发布的硬件所获得的些许性能提升要有价值得多。

## 4.4 实现跨服务的监控和警报。

实现跨服务的监控和警报。如果服务中有一个附属服务过载，那么被依赖的服务就必须了解这

个情况，并且如果服务无法自动备份，那么必须发送警报。如果运营部门无法快速地解决这个问题，那么服务就得设计得能容易迅速地联系到两个团队的工程师。所有相关的团队都应当在附属团队中安排工程联络人。

### 4.5 附属服务需要同一个设计点。

附属服务需要同一个设计点。附属的服务和附属组件的生产者至少必须遵循与所属服务相同的SLA（服务水平协议）。

### 4.6 对组件解耦。

对组件解耦。在所有可能的地方保证在其他组件故障时，组件可以继续运行，可能在一个降级的模式中。例如，比起在每一个连接上重新验证，维护一个Session键值，并且无论连接状况如何，每过N小时就刷新这个键值会更好些。在重新建立连接时，只使用现有的Session键值。这样的话在验证服务器上的负载就会更加一致，而且也不会在临时网络故障和相关事件之后的重新连接过程中出现登录高峰期的情况。

## 5 发布周期及测试

在生产环境中进行测试是一件很现实的事情，必须成为所有Internet级服务所必须的质量保证方式之一。在尽可能（ 可以掏得起钱）的情况下，绝大多数服务都应当有至少一个与生产环境相似的测试实验室，并且所有优秀的工程师团队应当使用生产级的负载，以反映现实的方式测试服务器。不过，我们的经验是，这些测试实验室即便模拟得再好，也绝不可能百分之百的逼真；它们至少总是会在一些细微的方式上和生产环境有所差别。由于这些实验室在真实性上接近生产系统，因此相应的费用也呈渐进趋势，在部署互联网级大规模服务时，有关依赖管理和发布周期及测试方面的经验和最佳实践在本文中得以体现。新的服务发布版本可以顺着标准单元测试、功能测试和生产测试实验室测试一路走下来，一直进入受限的生产环境作为最后的测试阶段。

快速逼近生产系统的开支。与此不同，我们推荐让新的服务发布版本顺着标准单元测试、功能测试和生产测试实验室测试一路走下来，一直进入受限的生产环境作为最后的测试阶段。显然，我们不想让无法正常工作并给数据一致性带来风险的软件进入生产环境，因此这就不得不小心翼翼地实施。下面的原则一定要遵守：

1．生产系统必须有足够的冗余，以保证在灾难性的新服务故障发生时，能够快速地恢复到原来的状态；

2．必须让数据损坏或者状态相关的故障极难发生（一定要首先通过功能测试）；

3．故障一定要能检测得到，并且开发团队（而不是运营团队）必须监控受测代码的系统健康

度；

4．必须可以实现对所有变更的回滚操作，并且回滚必须在进入生产环境之前经过测试。这听起来有点让人心惊胆战。不过我们发现，使用这个技术实际上能够在新服务发布时提升客户体验。与尽可能快地进行部署的做法不同，我们在一个数据中心中将一个系统放到生产环境数天。随后在每个数据中心内把新系统引入生产环境。接着，我们会将整个数据中心带入生产环境。最后，如果达到了质量和性能的目标，我们就进行全局部署。这种方式可以在服务面临风险之前发现问题，事实上还可以通过版本过渡提供更优秀的客户体验。一锤定音的部署是非常危险的。我们青睐的另一种可能违反直觉的方式是，在每天正午而不是半夜部署。在晚上部署，出现错误的风险更高，而且在半夜部署时如果有异常情况突然发生，那么能处理这些问题的工程师肯定会少些。这样做的目标是

为了使开发和运营团队与整体系统的互动最小化，尤其在普通的工作日之外，使得费用得到削减的同时，质量得到提高。对于发布周期和测试的最佳实践包括：

## 5.1 经常性地交付。

经常性地交付。直觉上讲，人们会认为更频繁地交付难度要更大，而且会错误频出。然而我们发现，频繁的交付中突兀的变更数量很少，从而使得发布的质量变得更高，并且客户体验更棒。对一次良好的发布所进行的酸性测试，用户提供可能会有所变化，但是关于可用性和延迟的运营

问题的数量应当在发布周期中不受改变。我们会喜欢三个月一次的交付，但也可以有支持其他时长的论调。我们从心底认为，标准最终会比三个月更少，并且有许多服务已经是按周交付的了。比三个月更长的周期是很危险的。

## 5.2 使用生产数据来发现问题。

使用生产数据来发现问题。在大规模系统中的质量保证，是个数据挖掘和可视化的问题，而不是一个测试问题。每个人都必须专注于从生产环境的海量数据中获得尽可能多的信息。这方面的策略有：

- 可度量的发布标准。定义出符合预期用户体验的具体标准， 并且对其进行持续监控。如果可用性应当为99%， 那么衡量可用性是否达到目标。如果没有达到， 发出警报并且进行诊断。
- 实时对目标进行调优。不要停顿下来考虑到底目标应当是99%、99.9%还是任何其他目标，设定一个可以接受的目标，然后随着系统在生产环境中稳定性的建立，让目标渐进式地增长。
- 一直收集实际数据。收集实际的度量，而不是那些红红绿绿的或者其他的报表。总结报表和图像很有用，不过还是需要原始数据用来诊断。
- 最小化“ 假阳性（falsepositive）”现象。在数据不正确时，人们很快就不再关注它们。不要过度警报，真是很重要的，否则运营人员会慢慢习惯于忽略这些警报。这非常重要，以至于

把真正的问题隐藏成间接损害常常是可以接受的。

- 分析趋势。这个可以用来预测问题。例如，当系统中数据移动的速度有异于往常的时候，常常能够预测出更大的问题。这时就要研究可用的数据。
- 使系统健康程度保持高度透明。要求整个组织必须有一个全局可用且实时显示的系统健康报告。在内部安置一个网站，让大家可以在任意时间查看并了解当前服务的状态。
- 持续监控。值得一提的是，人们必须每天查看所有数据。每个人都应当这么做，不过可以把这项工作明确给团队的一部分人专职去做。

## 5.3 在设计开发上加大投入。

在设计开发上加大投入。良好的设计开发可以使运营需求降到最小，还能在问题变成实际运营矛盾之前解决它们。非常常见的一个现象就是组织不断给运营部门增加投入，处理伸缩问题，却从没花时间设计一套伸缩的可靠架构。如果服务一开始没有进行过宏伟的构思，那么以后就得手忙脚乱地追赶了。

## 5.4 支持版本回滚。

支持版本回滚。版本回滚是强制的，而且必须在发布之前进行测试和验证。如果没有回滚，那么任何形式的产品级测试都会存在非常高的风险。回复到先前的版本应该是一个可以在任意部署过程中随时打开的降落伞扣。

## 5.5 保持前后版本的兼容性

保持前后版本的兼容性。这一点也是至关紧要的，而且也前面一点关系也非常密切。在组件之间更改文件类型、接口、日志/ 调试、检测（instrumentation）、监控和联系点，都是潜在的风险来源。除非今后没有机会回滚到之前的老版本的可能，否则不要放弃对于老版本文件的支持。

## 5.6 单服务器部署。

单服务器部署。这既是测试的需求也是开发的需求，整个服务必须很容易被托管到单一的系统中。在对于某些组件单服务器无法实现的地方（比如说一个对于外部、非单箱的可部署服务），编写模拟器来使单服务器测试成为可能。没有这个的话，单元测试会的难度会很大，而且不会完全覆盖到实际条件。而且，如果运行完整的系统很困难的话，开发人员会倾向于接受从组件的角度看问题，而不是从系统的角度。

### 5.7 针对负载进行压力测试。

针对负载进行压力测试。使用两倍（或者更多倍的）负载来运行生产系统的某些小部分，以确保系统在高于预期负载情况下的行为得到了解，同时也确保系统不会随着负载的增加而瓦解。

### 5.8 在新发布版本之前进行功能和性能测试。

在新发布版本之前进行功能和性能测试。在服务的级别上这么做，并针对每个组件这么做，因为工作负载的特征会一直改变。系统内部的问题和降级现象必须在早期捕获。

### 5.9 表象性且迭代地进行构建和部署。

表象性且迭代地进行构建和部署。在开发周期中早早地把完整服务的骨架先搭建起来。这个完整服务可能几乎做不了什么，也可能在某些地方出现偏差，但是它可以允许测试人员和开发人员更有效率，而且也能让整个团队在一开始就从用户的角度进行思考。在构建任何一个软件系统时，这都是一个好方法。不对，对于服务来说这尤为重要。

### 5.10 使用真实数据测试。

使用真实数据测试。将用户请求和工作量从生产到测试环境分门别类。选择生产数据并把它放到测试环境中。产品形形色色的用户，在发现bug的时候总是显得创意无穷。显然，隐私承诺必须保持，使得这样的数据永远不会泄漏回到产品环境中，这是至关紧要的。

### 5.11 运行系统级的验收测试。

运行系统级的验收测试。在本地运行的测试提供可以加速迭代开发的健康测试。要避免大量维护费用，这些测试应当放在系统级别。

### 5.12 在完全环境中做测试和开发

在完全环境中做测试和开发。把硬件放在一边，在专注的范围内测试。作重要的是，使用和在这些环境中的生产条件下同样的数据集合和挖掘技术，以保证投资的最大化。■(中结束)

# 6 运营和功能计划

要高效地运营服务，关键在于让构建的系统有效地消除运营团队的绝大部分管理交互。这样做的目标，是让一个高度可靠的24 x 7 小时运行的服务，由一个8 x 5小时工作的运营团队就足以

维护起来。

不过世事难料，一组或者多组系统救火不成，无法恢复上线的事情是时有发生的。在熟知这些可能性的情况下，实现把损坏的系统标为当机这个过程的自动化。依赖运营团队手动更新SQL表或者使用特别的技术移动数据，都会招致灾难。与故障交战正酣时，往往错误也容易迭出。先预估运营团队需要采取的补救措施，然后预先编写和测试这些过程。一般来说，开发团队必须将紧急恢复措施自动化，而且他们必须对之进行测试。显然，百密一疏，并非所有故障都能预估到，但通常一小组恢复措施就可以用来恢复多种类型的故障。从根本上说，要构建并测试可以根据灾难的范围和性质以不同方式使用及结合的“恢复内核”。

恢复脚本应当在生产环境中进行测试。这里有一条普适规则，即如果没经过频繁测试，什么程序都无法正常工作，因此不要实现团队没勇气使用的任何东西。如果在生活环境中测试风险过高，那么脚本就没有达到能在紧急情况下使用的标准，或者说在紧急情况下不安全。这里很关键的一点是，灾难总是可能发生的，由无法按预期结果运行的恢复步骤所导致的小问题酿成大灾难的例子是屡见不鲜的。要预见到这样的事件，设计出自动化措施，让服务回复正常状态，而不至于丢失更多数据或损耗更多正常运行时间。

## 6.1 让开发团队承担责任。

让开发团队承担责任。Amazon也许是沿着这条道路贯彻得最坚定最矢志不渝的公司了——他们的口号是“你创建就该你管”。这样的立场也许要比我们会选择的更坚定一些，但这显然是一个正确的大方向。如果不得不频繁在深更半夜给开发团队打电话，那么你就得做出一套自动化方案。如果需要频繁给运营团队打电话，那么通常的反应就是要增加运营团队的人手。

## 6.2 只进行软删除。

只进行软删除。绝不要删除任何东西，只可以把这些东西标记成删除状态。在有新数据进入时，即时将请求记录下来。每两周（或者更长时间）保存一份所有操作的历史记录，可以有助于从软件或者管理上的错误进行恢复。如果有谁犯了错误，忘记在delete 语句加上where 子句（这样的错误以前发生过，以后也可能再犯），那么数据的所有逻辑拷贝都会被删除。不管是RAID还是镜像都无法防止这样的错误。具备数据恢复的能力，可以让原来会十分令人窘迫难堪的大问题转化为一个小到甚至可以忽略不计的小障碍。对于那些已经做过离线备份的系统，只需要从上次备份开始记录进入服务的附加数据就可以了。不过谨慎起见，不管怎么说我们还是推荐进行更进一步的备份。

## 6.3 跟踪资源分配。

跟踪资源分配。了解性能规划的额外负载所带来的开销。每个服务都需要开发出一些使用的度量标准，例如并发在线用户数量、每秒用户访问数或者其它合适的标准。不管度量标准是什么，在

这个负载度量和所需的硬件资源之间肯定存在一个已知的直接相互关系。估算的负载数字应当由销售和营销部门提供，并在性能规划过程中为运营团队所使用。不同的服务会拥有不同的变更速率，也要求不同的订购周期。在我们开发过的服务中，我们每90天更新一次市场预报，每30天更新一次性能规划和订购一次设备。

### 6.4 每次变更一样东西

每次变更一样东西。在出现麻烦时，应当每次只向环境应用一个变更。这条准则看起来显而易见，但我们也看见过许多场合里出现多个变更，导致起因和效果不能吻合。

### 6.5 使所有资源都可以配置。

使所有资源都可以配置。在生产环境中，任何存在变更需求的资源，都应无需经过任何代码改变，就能在生产环境中进行配置和调优。即使你没什么好理由说明为什么某个值会有必要在生产环境中更改，只要实现起来没什么难度，还是让它可变更好些。不应在生产环境中随意更改这些开关，而应该使用为生产环境所规划的配置对整个系统进行彻底测试。不过，在出现生产环境问题时，比起编码、编译、测试再部署代码变更的过程，进行简单的配置变更永远是更加简单、安全和快速的。

## 7 审核、监控和预警

运营团队不能在部署环境中装配服务。要在部署过程中付出实质性的努力，以确保系统中的每个组件都可以生成性能数据、健康数据和吞吐量数据等。

在任何有配置变更发生的时候，都必须在审核日志中记录详细变更内容、变更人和变更时间。在生产环境出现异常时，第一个要回答的问题就是最近到底进行过哪些变更。离开了审核跟踪，那么这个答案就是“什么都没被改过”，而且情况往往会是，被人们忘掉的就是引发问题的变更。预警是一门艺术。人们总是倾向于对所有事件都做警报，因为开发人员认为这些事件可能值得关注。正是如此，多数服务的第一版通常都会产生长篇累牍的无用预警，结果再也没

有人去理睬。要提高效率，每个警报都得说明一个问题，否则运营团队会慢慢学会对这些警报置之不理。进行互动慢慢跳出需要警报的条件，保证所有关键性事件得到预警，以及在无需采取应对措施时没有警报。除此之外，要实现正确的预警别无灵丹妙药。要得到正确的预警标准，有两条度量标准很有用，也值得尝试：

一、警报和实际故障比（同时要设定一个较为接近的目标）；

二、没有相应警报的系统健康问题数量（并设定一个近于0的目标）。

## 7.1 对所有资源进行检测。

对所有资源进行检测。测量通过系统的每一次用户交互或事务，报告异常情况。尝试“运行器（人为的工作负载，用来模拟生产环境中用户和服务的交互）”也是可以的，不过这还远远不够。如果只是单独使用运行器，我们发现需要花费数日才能检测到一个严重错误，因为运行器的标准工作负载也会被继续良好地处理，接下来还要再花几天才能查出原因。

## 7.2 数据是最有价值的资产。

数据是最有价值的资产。如果没有充分理解正常的操作行为，那么要对非正常行为做出响应就不是件容易的事情。我们需要汇集许多系统内发生的事件信息，才能知道系统是否真的正常运行。许多服务都经历过灾难性故障，而只有电话铃响起的时候，人们才意识到故障的发生。

## 7.3 从客户的角度看服务。

从客户的角度看服务。进行端到端的测试。虽然单有运行器不够，但还是需要它们来保证服务器的完整运行。保证例如新用户登录这样重要的复杂过程经过运行器的测试。避免误警，如果在某个运行器上的故障没被当作重要故障，那么变更测试对象，换到是重要故障的运行器上。重申一下，一旦人们开始对数据视而不见，真正的损失就会让人们措手不及。

## 7.4 检测是生产环境测试所必不可少的。

检测是生产环境测试所必不可少的。检测是生产环境测试所必不可少的。要在生产环境中实现安全的测试，就有必要进行全面监控和预警。如果某个组件开始出现故障，就必须快速检测出来。

## 7.5 延迟是最棘手的问题。

延迟是最棘手的问题。缓慢的I/O，以及尚未出现故障但处理缓慢的现象，都是很好的例子。这些问题发现起来很困难，所以一定要仔细检测，保证这样的现象可以检测出来。

## 7.6 要有足够的生产环境数据

要有足够的生产环境数据。为了发现问题，数据是不可或缺的。在早期就要建立细粒度的监控机制，否则放到后面再翻新成本就高得多了。

我们所依赖的数据中最重要的包括：

- 对所有操作使用性能计数器。至少记录操作的延迟和每秒钟的操作次数。这些数值突然出现的此消彼长现象，是一个十分危险的信号。
- 审核所有操作。每次有人进行操作之后，尤其是那些明显的操作，一定要记入日志。这样做

有两个目的：首先，可以对日志进行挖掘，找出用采取的操作类型（在我们的例子里是用户查询的种类）；其次，一旦发现问题，这样做有助于调试问题。相关视点：如果每个人使用相同的账号管理系统的话，这么做带来不了多少好处。相反这是一个非常糟的办法，不过这种情况不多。

- 跟踪所有容错机制。容错机制会把故障隐藏起来。每次出现重试、某个数据被一处复制到另一处，以及机器或者服务重启这类现象时，都要进行跟踪。要了解容错机制在何时隐藏了小故障，这样就可以对这些小故障进行追溯，以防其变成大面积故障。我们曾经碰到过这样一个问题：一个跨2000台机器的服务在几天内慢慢地瘫痪，最后只剩400 台机器可用，而一开始这个问题却没有发现。
- 跟踪对重要实体的操作。为某个特殊实体的所有重要操作记录一份“审核日志”，不管这个实体是一个文档，或者一组文档。在运行数据分析时，常常能在数据中发现异常现象。要了解数据的来源及其经历的处理过程。到了后期才往项目加入这样的功能是非常困难的。
- 断言（asserts）。不要吝惜断言的使用，而且要贯彻在整个产品中。收集因此产生的日志或者崩溃转储（crashdump），并进行调查研究。对于在同一个进程边界内运行不同服务并且无法使用断言的系统，要写下跟踪记录。不管哪种实现，都要能够对错误进行标记，频繁挖掘不同问题的频率。
- 保留历史记录。历史性能和日志数据对于趋势的总结和问题的诊断都是非常必要的。

## 7.7 可配置的日志功能。

可配置的日志功能。对可配置的日志功能提供支持，这些日志记录可以有选择性开启或关闭，以便进行错误调试。在故障过程中，如果不得不部署带额外监控功能的新构建版本是非常危险的。

## 7.8 外部化健康信息，用于监控

外部化健康信息，用于监控。考虑能实现外部监控服务健康程度的方式，并使对生产环境进行监控容易实现。

## 7.9 保证所有报告的错误可以应对

保证所有报告的错误可以应对。问题总会发生，系统也总会出错。如果在代码中检测到无法恢复的错误，并且在日志或者报告中归为错误，那么错误信息应当指出错误可能发生的原因，并提供修复的建议。无法采取应对措施的错误报告毫无用处，而且时间一久这些错误报告会像“狼来了”一样被人们忽略，那时候真正的错误就可能被错过。

### 7.10 启用生产环境问题的快速诊断。

启用生产环境问题的快速诊断。

为诊断提供足够信息。当问题被标出时，要提供足够的信息，人们才可以对其进行诊断；否则门槛会非常高，标注也会被忽略。例如，不要只说“10个查询都没返回结果”，还得补上“列表在这里，还有问题出现的次数”。

证据链。保证存在一条贯穿始终的路径，可供开发人员诊断之用。通常这都是由日志实现的。

在生产环境中的调试。我们偏爱这样的一种模式：系统没有被包括运营团队在内的任何人触碰过，并且调试是通过镜像快照、内存转储并将所得数据发送到生产环境之外来实现的。当生产环境成为唯一选择是，开发人员就是最好的选择。要确保开发人员得到良好的培训，知晓生产环境的操作限制。我们的经验是，系统在生产环境中动的次数越少，客户通常也越开心。因此我们推荐，一定要多努一把力，尽量避免接触生产环境中的系统。

◇ 记录所有重要的操作。每次系统执行了重要的操作，特别是对网络请求和数据修改上，要进行记录。这既包括用户发送命令的事件，也包括系统内部的行为。有了这样的记录，调试问题的时候就会获益无穷。更重要的是，还可以开发出相应的挖掘工具，用来找出有用的集合，比如用户的查询都是什么样的（也就是说，用了哪些关键字，有多少关键字等等）。

## 8 优雅降级和许可控制

当发生DoS攻击或者由于用户使用模式变化而带来的负载激增时，服务器需要能实现优雅降级和管理控制。例如，911恐怖袭击发生后，大多数的新闻服务都发生瘫痪，无法向所有用户群体提供任何可用的服务。与此相比，如果能够将一部分文章可靠地提供给用户，总比什么都不能提供好。最佳实践有两种：“大红开关（big red switch）”和许可控制，如果能够针对服务进行量身定制，作用将会十分强大。

### 8.1 支持“大红开关”。

支持“大红开关”。这个概念最初来源于Windows Live Search，它拥有很大的权力。我们对这个概念进行了一定程度的推广，更事务性的服务与搜索差别迥异。不过这个想法非常有效，而且可以应用于任何地方。一般来说，“大红开关”是当服务已经或者即将无法满足SLA时，采用的一个经过精心设计和测试的措施。将优雅降级比喻成“大红开关”稍微有些不恰当，但核心意思是指在紧急时刻卸掉非关键的负载。“大红开关”的主要思想是保证关键任务的运行，同时卸掉或者延迟非关键的负载。从设计角度来说，这种情况应该不会发生，但在发生紧急情况时不失为一个好的救火办法。在紧急情况已经发生之后再来考虑这些问题是十分危险的。如果有哪个负载可以被加入队列并延后处理，或者在关掉高级查询后仍能继续运行事务系统，那么这都是作为“大红开关”

的理想对象。关键在于，判断在整个系统出现问题时有哪些任务是必不可少的，然后实现并测试在问题出现时关闭非关键服务的选项。切记“大红开关”必须是可逆的，也就是说当紧急情况解除后，必须保证开关能正确地让所有的批处理工作和先前被停止的非关键任务恢复原状，这是应当经过测试的。

## 8.2 控制许可

控制许可。许可控制是另外一个重要的概念。如果都无法对当前的任务进行处理，向系统引入更多的工作负载只会将不好的体验扩散到更大的用户群中。这要如何实现和系统有关，有些系统实现很容易，有些则比较困难。拿我们上次的邮件处理服务作为例子，如果系统超过负荷，开始排队列的话，我们最好是拒绝接受后续邮件进入系统，然后让这些邮件在来源处列队等候。这样做很有意义并实际上减少了整体服务延迟，主要原因在于一旦队列在我方形成，系统会处理得更慢。如果我们拒绝形成队列，吞吐量也会得以提升。另外一个诀窍是：重要客户优先于非重要用户，注册用户优先于访客（但如果是以吸引新访客作为商业模式的，另当别论）。

## 8.3 对许可进行计量

对许可进行计量。还有一个无比重要的概念，就是前面所说的许可控制观点的修改方案。如果系统故障并当机，必须能够实现逐步恢复用户使用，并确保系统运行正常。比如先允许1 个用户进入，然后允许每秒进入10 个用户，随后再逐渐放宽限制。对于重新上线或者从严重错误中恢复过来的系统，应该确保每项服务都有一个细粒度的开关，来实现用户的缓慢增加，这是至关紧要的。大多数系统的初始版本中很少有包括这项功能。

对于有客户端的系统，一定会存在一种方式，用来通知客户端服务器当机，并且告知可能的恢复时间。这样一方面使客户端尽可能继续基于本地数据来运行，另一方面也可以避免客户端打扰服务器，以便后者更快恢复。这样同时也给了服务拥有者一个直接和用户沟通的机会（ 见下节），用来调整用户的期望值。另外一个关于客户端的技巧是刻意设置扰动（jitter），和自动备份来防止客户端在同一时刻“扑向”脆弱的服务器。客户和媒体沟通计划当系统出错时需要就发生的延误和其它相关事项与客户进行沟通。沟通应当能以可选的方式通过多个渠道完成 ：RSS、Web，还有即时消息等等。另外，对于拥有客户端的系统，通过客户端来进行和用户的沟通也是很有效的。可以告诉客户端在一定时间内或者特定时间点之前避免访问服务器，或者如果支持的话，可以告诉客户端在离线和缓存的方式运行。客户端可以把系统状态告诉用户，并说明可以预计完整的功能在何时恢复。即使在没有客户端的情况下，比如用户通过网页来和系统交互，仍然可以告知用户系统当前的状态。当用户了解发生的状况并对系统的恢复时间有一个合理的期望值，他们的满意度会大大提高。系统管理员常常会不自觉地倾向于隐藏系统发生的问题，但是跟据我们的经验，我们确信，将系统的状态告知用户会极大地提高其满意度。即便是非付费系统，如果人们知道系统发生了状况

并被告知其何时会恢复，他们放弃使用这项服务的可能性也会减小。某些事件会引发媒体报导。如果这样的场景有预先备好的应对方案，那么会更加真实地反应服务的情况。大量数据丢失或损坏、安全遭到破坏、违反隐私以及服务器长时间当机，这样的情况都可能引起媒体的关注。事先准备好一份沟通计划。清楚电话通知谁，并能主导谈话。沟通方案的框架应当事先搭好。应针对每一种事故制定一份沟通方案，内容包括该给谁电话通知、电话时间，还有如何掌控沟通过程。客户自预置及自助服务客户自己进行预置可以大幅度降低成本，同时还能提升客户满意度。如果客户可以访问网站，输入所需数据，然后就可以开始使用服务，那么他们要比不得不在电话处理队列中浪费时间开心得多。我们一直认为，主流移动运营商因为没为那些不想致电客户支持组的用户提供自助服务，错过了一次拯救并提升客户满意度的好机会。

# 9 结语

要降低大规模互联网服务的运营成本并改善服务的可靠性，一切从编写服务时注重运营友好开始。在这篇论文中，我们为“运营友好”做了诠释，并根据从事大规模服务的工程师的经验，总结了服务设计、开发、部署和运营的最佳实践

# 10 作 者 简 介

James Hamilton，

是微软LivePlatform Service团队的架构师，在微软有11 年工作经验， 此前他曾带领Exchange Hosted Service团队，该团队为超过两百万用户提供Email 相关服务。在微软的前八年，他曾是SQL Server 团队成员，并带领大部分的核心引擎开发团队。在加入微软之前，James曾担任IBM 的DB2 UDB 团队的首席架构师，更早时曾带领团队交付IBM的第一个C++ 编译器。在20 世纪70 年代末、80 年代初，他曾持有汽车机械师和意大利汽车竞赛驾驶执照。