AWS 白皮书

可用性及其他：了解和提高分布式系统的弹性 AWS

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

可用性及其他：了解和提高分布式系统的弹性 AWS: AWS 白皮书

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商标和商业外观不得用于任何非 Amazon 的商品或服务，也不得以任何可能引起客户混淆、贬低或诋毁 Amazon 的方式使用。所有非 Amazon 拥有的其他商标均为各自所有者的财产，这些所有者可能附属于 Amazon、与 Amazon 有关联或由 Amazon 赞助，也可能不是如此。

Table of Contents

摘要和简介 i
简介 1
了解可用性 2
分布式系统可用性 3
分布式系统中的故障类型 5
可用性和依赖项 6
可用性和冗余 7
CAP 定理 10
容错能力和故障隔离 11
衡量可用性 13
服务器端和客户端请求成功率 13
年度停机时间 15
延迟 15
在上设计高度可用的分布式系统 AWS 17
减少 MTTD 17
减少 MTTR 18
绕过故障 18
恢复到已知良好状态 19
故障诊断 21
运行手册和自动化 21
越来越多 MTBF 21
越来越多的分布式系统 MTBF 21
依赖性增加 MTBF 23
减少常见的影响源 24
结论 26
附录 1 — MTTD 和 MTTR 关键指标 28
贡献者 29
延伸阅读 30
文档历史记录 31
注意事项 32
AWS 术语表 33
..... xxxiv

可用性及其他：了解和提高 AWS 上的分布式系统的韧性

发布日期：2021 年 11 月 12 日（文档历史记录）

如今的企业会在云端和本地运行各种复杂的分布式系统。他们希望这些工作负载具有韧性，以便为客户提供服务并实现业务成果。这篇论文概述了把可用性作为韧性的衡量标准这一共识，建立了构建高可用性工作负载的规则，并就如何提高工作负载可用性提供了指导。

简介

构建高可用性工作负载意味着什么？ 如何衡量可用性？ 怎样才能提高工作负载的可用性？ 本文将会帮助您回答这类问题。本文分为三个主要部分。第一部分了解可用性是偏理论性的论述。它针对可用性的定义和影响可用性的因素建立了共识。第二部分衡量可用性提供了实证式衡量工作负载可用性方面的指导。第三部分在 AWS 上设计高可用性分布式系统是第一部分中介绍的想法的实际应用。此外，本文还在上述内容中明确了构建韧性工作负载的规则。本文的目的是为 AWS Well-Architected 可靠性支柱中提出的指导和最佳实践提供支持。

在这篇论文中，我们会遇到很多代数学知识。我们应该关注数学支持的概念，而不是数学本身。这篇论文的目的也包括提出一项挑战。当您运行高可用性工作负载时，您需要能够在数学上证明自己构建的东西实现了预期目的。即使是建立在良好意愿之上的最佳设计也可能无法始终如一地实现预期结果。这意味着您需要能够衡量解决方案有效性的机制，因此，在构建和运行有韧性并且高度可用的分布式系统时，必须进行一定程度的数学运算。

了解可用性

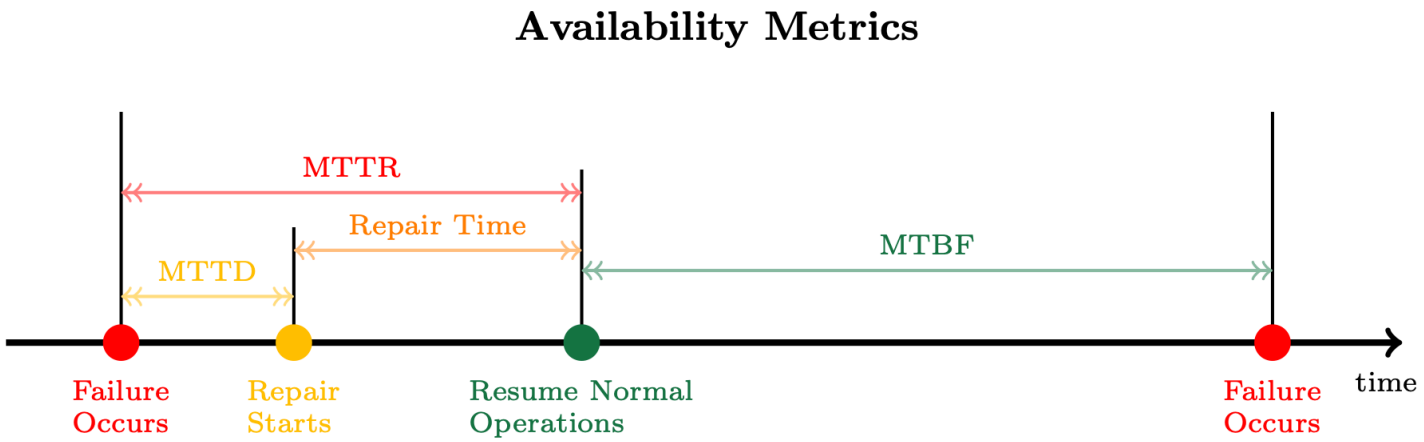
可用性是我们定量衡量韧性的主要方法之一。我们将可用性 A 定义为工作负载可供使用的时间百分比。这是其预期“正常运行时间”（可用时间）与衡量的总时间（预期“正常运行时间”加上预期“停机时间”）的比率。

$$A = \frac{uptime}{uptime + downtime}$$

公式 1 - 可用性

为了更好地理解这个公式，我们来分析一下如何衡量正常运行时间和停机时间。首先，我们需要知道工作负载能持续多长时间不出现故障。我们称之为平均故障间隔时间 (MTBF)，即工作负载开始正常运行与下一次故障之间的平均时间。然后，我们需要知道发生故障后需要多长时间才能恢复。

我们称之为平均修复（或恢复）时间 (MTTR)，即在发生故障的子系统被修复或恢复服务时，工作负载不可用的时长。MTTR 中的一个重要时间段是平均检测时间 (MTTD)，即从故障发生到修复操作开始之间的时间长度。下图显示了所有这些指标之间的关联。



MTTD、MTTR 和 MTBF 之间的关系

因此，我们可以使用 MTBF（工作负载运行的时间）和 MTTR （工作负载关闭的时间）来表示可用性 A。

$$A = \frac{MTBF}{MTBF + MTTR}$$

公式 2 - MTBF 和 MTTR之间的关系

而工作负载“停机”（即不可用）的概率就是发生故障的概率 F。

$$F = 1 - A$$

公式 3 - 故障概率

可靠性是指工作负载收到请求后在指定响应时间内执行正确操作的能力。这是可用性衡量的对象。降低工作负载故障频率（提高 MTBF）或缩短修复时间（缩短 MTTR）都可以提高可用性。

> 规则 1
>
> 降低故障频率（提高 MTBF）、缩短故障检测时间（缩短 MTTD）和缩短修复时间（缩短 MTTR）是提高分布式系统可用性的三项因素。

主题

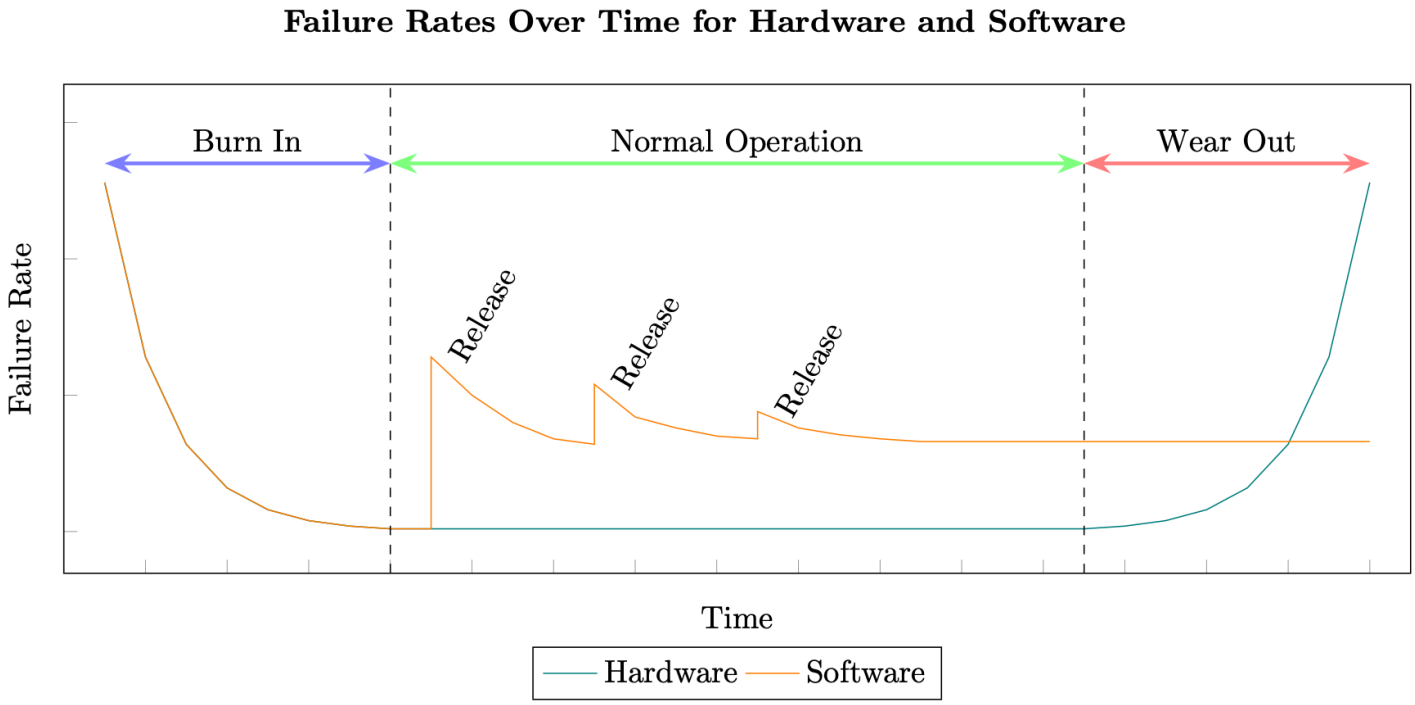
- 分布式系统可用性
- 可用性和依赖项
- 可用性和冗余
- CAP 定理
- 容错能力和故障隔离

分布式系统可用性

分布式系统由软件组件和硬件组件组成。有些软件组件本身可能是另一个分布式系统。底层硬件和软件组件的可用性会影响工作负载的可用性。

使用 MTBF 和 MTTR 来计算可用性的方法起源于硬件系统。但是，分布式系统发生故障的原因与硬件故障原因截然不同。制造商始终可以计算出硬件组件失效前的平均时间，但分布式系统的软件组件却无

法进行这种测试。硬件通常遵循“浴缸式”故障率曲线，而软件因为每次发布新版本时都会引入额外的缺陷，所以会形成交错式曲线（参见软件可靠性）。



硬件和软件故障率

此外，分布式系统中软件的变化速度往往比硬件高很多。例如，标准磁性硬盘的平均年化故障率 (AFR) 可能为 0.93%，对于硬盘驱动器来说，这实际上意味着在达到失效期之前其至少有 3-5 年的使用寿命，甚至可能更长（参见 Backblaze 硬盘数据和统计，2020年）。在这段生命周期中，硬盘驱动器不会发生实质性变化，而在 3-5 年内，以亚马逊为例，其各种软件系统可能要部署 4.5 亿至 7.5 亿次更改。（参见 Amazon Builders's Library — 自动实现无需干预的安全部署。）

硬件还存在计划性报废的概念，即具有设定的使用寿命，需要在一定时间后更换。（参见灯泡大阴谋。）而软件在理论上不受这一限制，它没有失效期，可以无限期地运行。

所有这些都意味着用于确定硬件 MTBF 和 MTTR 的各种测试和预测模型并不适用于软件。自 20 世纪 70 年代以来，人们数百次尝试通过建立模型来解决这个问题，但一般都没有脱离预测建模和估计建模的范畴。（参见软件可靠性模型列表。）

因此，要计算分布式系统未来的 MTBF 和 MTTR，从而确定未来的可用性，我们始终需要依赖某种类型的预测。我们可以通过预测建模、随机仿真、历史分析或严格测试来生成计算结果，但这些计算结果并不能成为正常运行时间或停机时间的保证。

过去导致分布式系统出现故障的原因可能永远不会再次出现。未来造成故障的原因可能截然不同，甚至完全不可知。对于未来的故障，所需的恢复机制也可能与过去的机制不同，所花费的时间也大不相同。

此外，MTBF 和 MTTR 均为平均值。平均值与实际值之间会有一些差异（通过标准差 σ 来衡量这种差异）。因此，在实际生产使用中，工作负载在故障与恢复之间的时间可能会更短或更长。

但是，构成分布式系统的软件组件的可用性仍然非常重要。软件可能由于多种原因（下一节会详细介绍）而发生故障，并影响工作负载的可用性。因此，对于高可用性分布式系统来说，软件组件可用性的计算、衡量和提高应该得到与硬件和外部软件子系统同等的重视。

> 规则 2
>
> 工作负载中的软件可用性是决定工作负载总体可用性的一项重要因素，应与其他组件同等重视。

值得注意的是，尽管分布式系统的 MTBF 和 MTTR 很难预测，但它们仍然可以为提高可用性提供重要信息。降低故障频率（提高 MTBF）和缩短故障发生后的恢复时间（缩短 MTTR）都可以提高实证可用性。

分布式系统中的故障类型

分布式系统中通常存在两类影响可用性的错误，分别叫做波尔错误和海森堡错误（参见“Bruce Lindsay 访谈”，ACM Queue 第 2 卷，第 8 号 – 2004 年 11 月）。

波尔错误是可以重复出现的功能性软件问题。给定相同的输入，就能始终产生相同的错误输出（如同确定性玻尔原子模型一样，稳定并且容易检测）。工作负载进入生产环境后，这类错误非常少见。

海森堡错误是一种短暂的错误，只发生在特定和不常见条件下。这些条件通常与硬件（例如瞬时设备故障或寄存器大小等硬件实现细节）、编译器优化和语言实现、限制条件（例如存储空间暂时不足）或竞争条件（例如不使用信号量进行多线程操作）等内容相关。

生产环境中的大部分错误都是海森堡错误，这种错误难以捉摸，当我们尝试进行观察或调试时，它们似乎会改变行为或消失，因此很难被发现。但是，如果重新启动程序，那么失败的操作很可能会成功，因为操作环境略有不同，消除了引发海森堡错误的条件。

因此，生产环境中的大多数故障都是暂时性的，当重试操作时，故障不太可能再次出现。为了保持韧性，分布式系统必须能够承受海森堡错误。我们将在提高分布式系统 MTBF 一节探讨如何实现这一目标。

可用性和依赖项

在上一节中，我们提到工作负载的组件包括硬件、软件，还可能包括其他分布式系统。我们将这些组件称为依赖项，即工作负载为了实现功能而需要依赖的事物。依赖项包括硬依赖项，即工作负载要发挥作用就离不开的事物，还包括软依赖项，其不可用性在一段时间内可能会被忽视或容许。硬依赖项会直接影响工作负载的可用性。

我们可能想要尝试计算工作负载的理论最大可用性。这个数值是包括软件本身在内的所有依赖项的可用性的乘积，（α_n 是单个子系统的可用性），因为每个依赖项都必须正常运行。

$$A = \alpha_1 \times \alpha_2 \times \ldots \times \alpha_n$$

公式 4 - 理论最大可用性

这种计算中使用的可用性数字通常与 SLA 或服务级别目标 (SLO) 之类的内容相关联。SLA 定义了客户将获得的预期服务级别、用于衡量服务的指标，以及未能达到服务级别时采取的补救措施或惩罚（通常是金钱）。

使用上面的公式，我们可以得出结论：纯粹从数学上讲，工作负载的可用性不会高于其任何依赖项。但实际上，我们通常看到的情况并非如此。如果一个工作负载两个或三个具有 99.99% 可用性 SLA 的依赖项，那么工作负载本身仍然可以实现 99.99% 或更高的可用性。

这是因为像上一节所说那样，这些可用性数字是估计值。我们会估计或预测故障发生的频率以及修复故障的速度。估计的数字并不代表一定会停机。依赖项的表现经常会超出其可用性 SLA 或 SLO 的规定。

依赖项为发挥性能而设定的内部可用性目标也可能高于公开 SLA 中的数字。这样可以提高在发生未知或不可知情况时仍然符合 SLA 的可能性。

最后，工作负载的依赖项可能具有不为人知的 SLA，或者不提供 SLA 或 SLO。例如，全球互联网路由是许多工作负载的常见依赖项，但我们很难知道自己全球流量正在使用哪家互联网服务提供商、他们是否具有 SLA，以及 SLA 在不同提供商之间的一致性如何。

这一切都告诉我们，计算最大理论可用性只可能得出粗略的数量级计算结果，但其本身可能并不准确，或者无法提供有意义的见解。从数学公式中可以看出，工作负载依赖的东西越少，发生故障的总体可能性就越低。小于一的数字越少，乘积就越大。

> 规则 3
>
> 减少依赖项可以对可用性产生积极影响。

数学计算还可以对依赖项选择过程起到辅助作用。选择过程会影响您如何设计工作负载、如何利用依赖项中的冗余来提高其可用性，以及您将其视为软依赖项还是硬依赖项。我们应该谨慎选择可能影响工作负载的依赖项。下一条规则提供了这方面的指导。

> 规则 4
>
> 通常应该选择可用性目标等于或高于工作负载目标的依赖项。

可用性和冗余

当工作负载使用多个独立的冗余子系统时，它可以实现比使用单个子系统更高的理论可用性水平。例如，假设某个工作负载由两个完全相同的子系统组成。只要有一个子系统正常运行，工作负载就能正常运行。要让整个系统停机，两个子系统必须同时关闭。

如果一个子系统的故障概率为 1 − α，则两个冗余子系统同时停机的概率就是每个子系统的故障概率的乘积：F = (1−α_1) × (1−α_2)。对于具有两个冗余子系统的工作负载，根据公式 (3)，其可用性为：

$$A = 1 - F$$
$$F = (1 - \alpha_1) \times (1 - \alpha_2)$$
$$A = 1 - (1 - \alpha)^2$$

公式 5

因此，对于两个可用性为 99% 的子系统，如果一个子系统出现故障的概率为 1%，那么两个子系统都出现故障的概率则为 (1−99%) × (1−99%) = 0.01%。这使得使用两个冗余子系统的可用性达到 99.99%。

这一规律也适用于增加冗余备件 s。在公式 (5) 中，我们只假设有一个备件，但是一个工作负载可能有两个、三个或更多备件，从而可以在多个子系统同时失效的情况下正常运行，不会影响可用性。如果某个工作负载有三个子系统并且其中两个是备件，则所有三个子系统同时出现故障的概率为 (1−α) × (1−α) × (1−α)，即 $(1-\alpha)^3$。一般来说，具有 s 个备件的工作负载只有在 s + 1 个子系统发生故障时才会失效。

对于具有 n 个子系统和 s 个备件的工作负载来说，f 代表故障次数，也是 n 个子系统中有 s + 1 个子系统发生故障的概率。

这实际上是二项式定理，即从 n 个元素中选择 k 个元素的组合数学（“n 选 k”）。在本例中，k 为 s + 1。

$$k = s + 1$$

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

$$\binom{n}{s+1} = \frac{n!}{(s+1)!\,\big(n-(s+1)\big)!}$$

$$f = \frac{n!}{(s+1)!\,(n-s-1)!}$$

公式 6

然后，我们可以得出一个包含故障次数和备件数量的广义可用性近似值。（要理解为什么是近似值，请参阅 Highleyman 等人的著作 Breaking the Availability Barrier 的附录 2。）

$$s = \ Number\ of\ spares$$
$$\alpha = Availability\ of\ subcomponent$$
$$f = Number\ of\ failure\ modes$$
$$A = 1 - F \approx 1 - f(1-\alpha)^{s+1}$$

公式 7

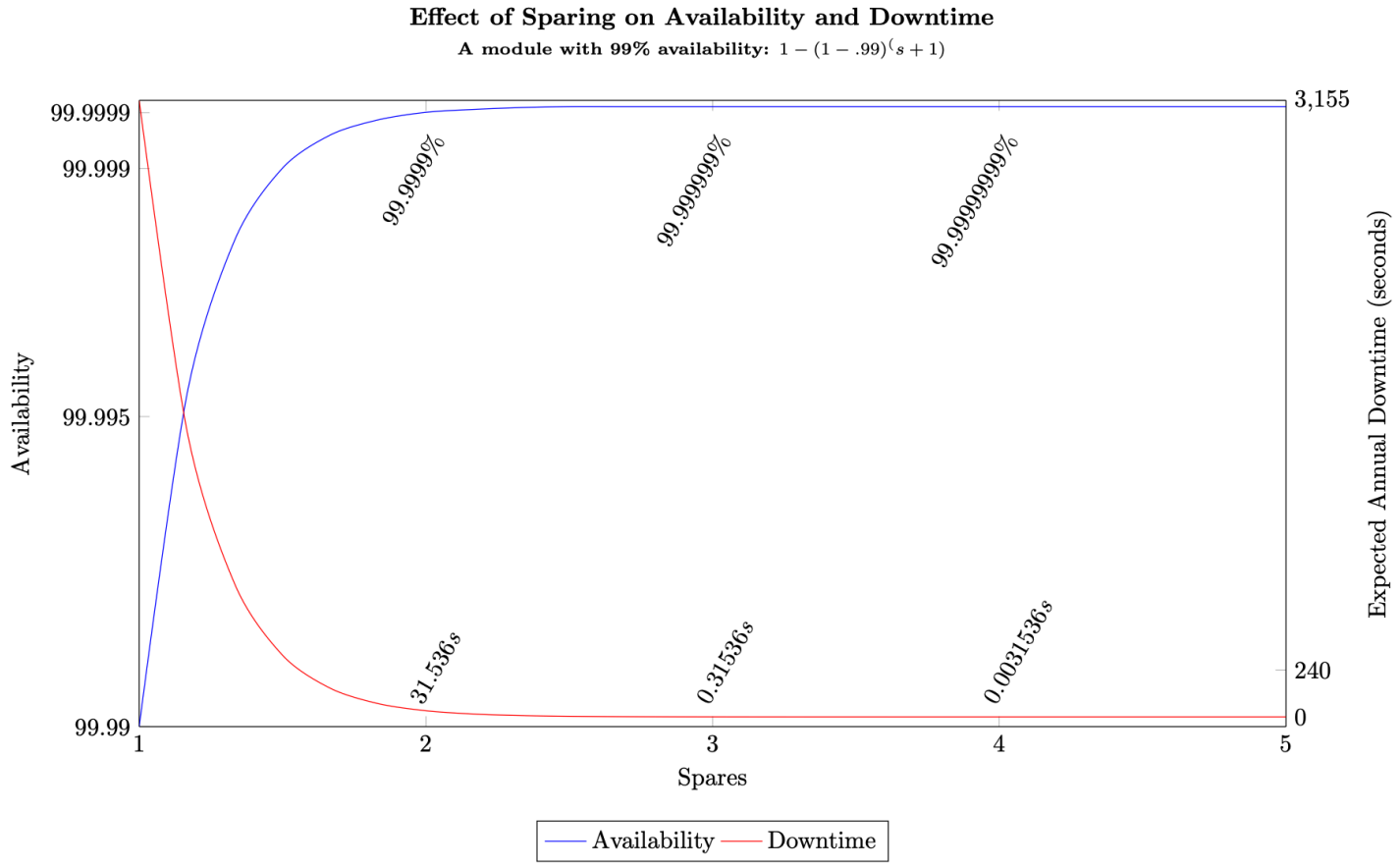
提供独立失败的资源的任何依赖项都可以设置备件。位于不同 AWS 区域中的不同可用区或不同 Amazon S3 桶中的 Amazon EC2 实例就是一个这方面的例子。使用备件可以帮助依赖项实现更高的总体可用性，从而支持工作负载的可用性目标。

> **规则 5**
>
> 使用备件以便提高工作负载中的依赖项的可用性。

但是，备件需要投入成本。增加的每个备件的成本与原始模块相同，所以成本至少会线性提高。构建可以使用备件的工作负载也会增加其复杂性。工作负载必须识别依赖项故障、将工作转移到正常运行的资源上，并管理总体容量。

冗余是一种优化问题。备件太少，工作负载可能比预期更频繁地出现故障；备件太多，工作负载的运行成本就会过高。超出某个阈值之后，增加更多备件的成本将会超过额外获得的可用性带来的收益。

根据成本与备件的一般公式，即公式 (7)，如果子系统的可用性为 99.5%，设置两个备件，那么工作负载的可用性 $A \approx 1 - (1)(1-.995)^3 = 99.9999875\%$（每年停机时间约为 3.94 秒）；而如果设置 10 个备件，那么可用性 $A \approx 1 - (1)(1-.995)^{11} = 25.5$ 个 9（每年停机时间约为 1.26252×10^{-15} 毫秒，实际上等于 0）。比较这两种工作负载可以发现，每年减少四秒钟的停机时间所产生的备件成本提高了 5 倍。对于大多数工作负载来说，成本的增加与可用性的提升显然不成比例。下图显示了这种关系。



增加备件导致收益递减

当备件不少于三个时，每年的预期停机时间不到一秒，这意味着在此之后进入了收益递减区间。有人可能想要不断增加备件以便实现更高的可用性，但实际上，成本效益很快就会消失。当子系统本身的可用性至少达到 99% 时，使用三个以上的备件并不能为几乎任何工作负载打来实质性的明显效益。

> 规则 6
>
> 备件的成本效益存在上限。利用最少的备件来实现所需的可用性。

在选择正确的备件数量时，您应该考虑故障单元。例如，我们假设某个工作负载需要 10 个 EC2 实例来处理峰值容量，并且这些实例部署在单个可用区内。

可用区是一种故障隔离边界，因此故障单元不仅仅是单个 EC2 实例，因为整个可用区内的 EC2 实例可能会一起出现故障。在这种情况下，您需要通过另一个可用区来添加冗余，也就是再部署 10 个 EC2 实例以便在可用区出现故障时处理负载，这样 EC2 实例的总数就是 20 个（遵循静态稳定模式）。

虽然看起来有 10 个备用 EC2 实例，但实际上我们只部署了一个备用可用区，还没有超过收益递减的边界。但是，您还可以使用 3 个可用区，并在每个可用区部署 5 个 EC2 实例，这样可以进一步提高成本效益和可用性。

这时有 1 个可用区处于备用状态，总共有 15 个 EC2 实例（而不是 2 个可用区和 20 个实例）。如果出现影响单个可用区的事件，这种配置仍然可以提供所需的 10 个实例来处理峰值容量。因此，您应该设置备件，以便在工作负载使用的所有故障隔离边界（实例、单元、可用区和区域）内建立容错能力。

CAP 定理

可用性的另一个方面与 CAP 定理有关。该定理指出，一个由存储数据的多个节点组成的分布式系统最多只能同时实现以下三点中的两点：

- 一致性 (C)：如果保证不了一致性，每个读取请求都会收到最新的写入内容或错误。
- 可用性 (A)：即使节点已关闭或不可用，每个请求也都会收到非错误响应。
- 分区容错性 (P)：尽管节点之间丢失了任意数量的消息，但系统仍能继续运行。

（有关更多详细信息，请参阅 Seth Gilbert 和 Nancy Lynch 的文章：Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services，ACM SIGACT News，第 33 卷第 2 期（2002 年），第 51—59 页。）

大多数分布式系统都必须容许网络故障，因此就必须允许网络分区。这意味着在出现网络分区时，工作负载就必须在一致性和可用性之间做出选择。如果选择可用性，那么工作负载就会始终返回响应，但数

据可能不一致。如果选择一致性，那么在网络分区期间，工作负载会返回错误，因为其无法确定数据的一致性。

对于以提供更高级别的可用性为目标的工作负载来说，他们可能会选择可用性和分区容错性（A 和 P），以防止在网络分区期间返回错误（不可用）。这就要求使用更宽松的一致性模型，例如最终一致性或单调一致性。

容错能力和故障隔离

这是与可用性有关的两个重要概念。容错能力是承受子系统故障并保持可用性（满足已有 SLA 的要求）的能力。为了实现容错能力，工作负载会使用备用（或冗余）子系统。当冗余中的一个子系统出现故障时，另一个子系统通常会以几乎无缝的方式接手其工作。在这种情况下，备件是真正的备用容量，可以承担故障子系统的 100% 的工作。使用真正的备件时，需要多个子系统故障才能对工作负载产生不利影响。

故障隔离可以尽可能缩小故障发生时的影响范围。这一结果通常通过模块化来实现。工作负载被分解为多个小型子系统，这些子系统的故障互不影响，可以单独修复。一个模块的故障不会传播到模块之外。这种效果既可以在一个工作负载中的不同功能之间纵向实现，也可以在提供相同功能的多个子系统之间横向实现。这些模块充当故障容器，可以限制事件的影响范围。

控制平面、数据平面和静态稳定性架构模式可以直接支持容错能力和故障隔离的实现。Amazon Builders Library 文章使用可用区的静态稳定性针对这几个术语给出了准确的定义，并介绍了它们如何应用于具有韧性和高可用性的工作负载的构建过程。本白皮书在设计高可用性分布式系统AWS一节中使用了这些模式，并在这里总结了它们的定义。

- 控制平面 — 工作负载中涉及做出更改的部分：添加资源、删除资源、修改资源以及将这些更改传播到所需位置。与数据平面相比，控制平面通常更加复杂并具有更多活动部件，因此从统计学上讲，控制平面失效的可能性更大，可用性也更低。
- 数据平面 — 工作负载中提供日常业务功能的部分。与控制平面相比，数据平面往往更加简单，运行容量也更高，因此可用性更高。
- 静态稳定性 — 工作负载在依赖项受损的情况下继续正常运行的能力。一种实现方式是从数据平面中移除控制平面依赖项。另一种方式是松散地耦合工作负载依赖项。工作负载可能看不到其依赖项本应提供的任何更新信息（例如新内容、删除的内容或修改过的内容）。但是，它在依赖项受损之前所做的一切仍然在发挥作用。

当工作负载受损时，我们可以考虑两种恢复方式。第一种是在损伤发生后对其做出反应，比如用 AWS Auto Scaling 来增加新的容量。第二种是在损伤发生之前做好准备，比如超额配置工作负载的基础架构，让它不需要额外资源就可以继续正常运行。

静态稳定的系统采用后一种方式。它预先配置了备用容量，以便在故障期间保持可用性。采用这种方式，我们就不需要在工作负载恢复路径中在控制平面上创建依赖项以便配置新容量，从而从故障中恢复。此外，为各种资源配置新容量需要耗费时间。在等待新容量时，工作负载可能会因现有需求而超负荷并进一步降级，从而降低或完全失去可用性。但是，您还应该对照可用性目标，考虑使用预配置容量所产生的成本影响。

静态稳定性针对高可用性工作负载提出了以下两项规则。

规则 7

不要在数据平面中的控制平面上设置依赖项，尤其是在恢复期间。

规则 8

尽可能松散地耦合依赖项，让工作负载在依赖项受损时也能正常运行。

衡量可用性

如上文所述，针对分布式系统创建有前瞻性的可用性模型是一项难以实现的任务，而且可能无法提供所需的见解。更实用的方式是建立起一致的方法来衡量工作负载的可用性。

如果用正常运行时间和停机时间来衡量可用性，那么故障就是一种二元选择：工作负载要么正在运行，要么就没有运行。

但是，这种情况很少见。故障会产生特定程度的影响，通常发生在工作负载的某些子集中，并会影响一定比例的用户、请求、位置，或者对延迟产生一定影响。这些都属于部分故障模式。

尽管 MTTR 和 MTBF 可以帮助我们了解影响系统可用性的因素以及如何提高可用性，但它们并不是衡量可用性的实证指标。此外，工作负载由许多组件组成。例如，像支付处理系统这样的工作负载包含许多应用程序编程接口 (API) 和子系统。所以，整个工作负载的可用性实际上是一个复杂而微妙的概念。

在本节中，我们将探讨以实证方式衡量可用性的三种方法：服务器端请求成功率、客户端请求成功率和年度停机时间。

服务器端和客户端请求成功率

前两种方法非常相似，只是从测量的角度来看有所不同。服务器端指标可以从服务中的工具中收集。但这些指标并不完整。如果客户无法访问服务，您就无法收集客户端指标。为了了解客户体验，我们可以不依赖客户对失败请求的反馈，而是使用金丝雀这种定期探测您的服务并记录指标的软件来模拟客户流量，从而收集客户端指标。

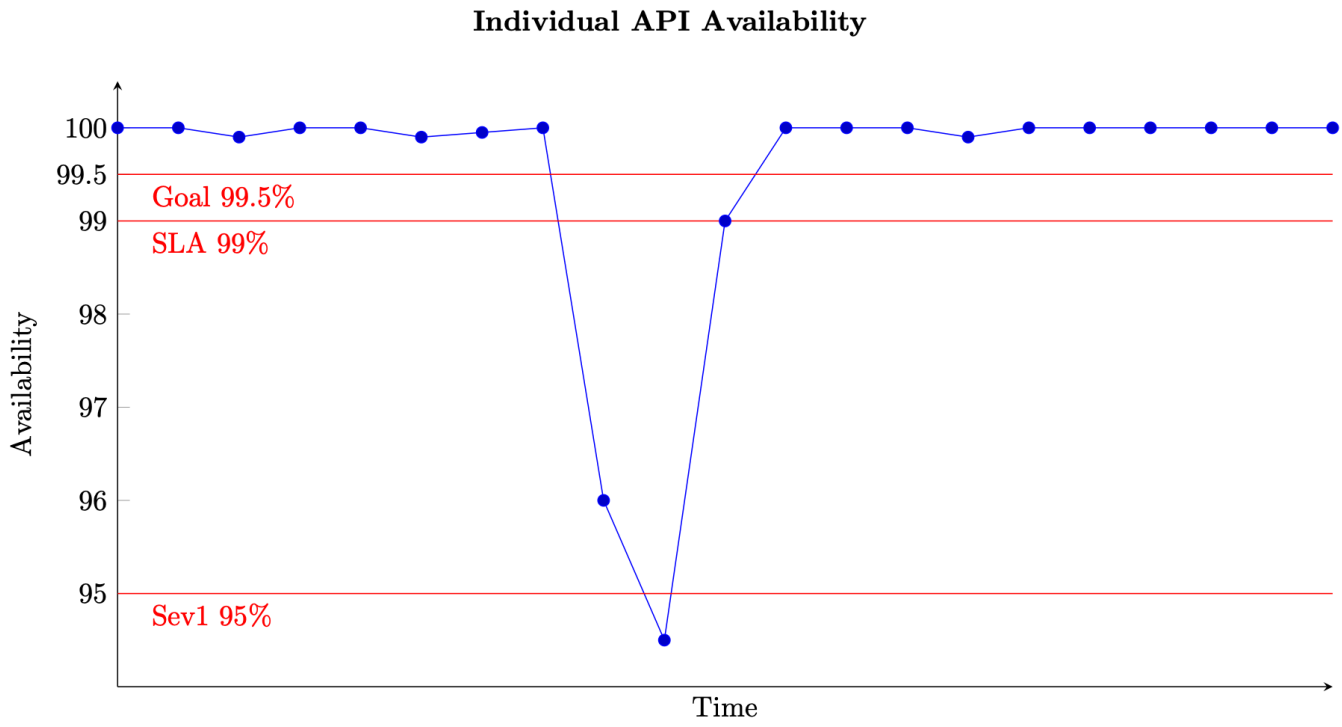
利用这两种指标，我们可以用成功处理的有效工作单元除以服务收到的有效工作单元的总数（忽略无效的工作单元，例如导致 404 错误的 HTTP 请求），从而得出可用性。

$$A = \frac{Successfully\ Processed\ Units\ of\ Work}{Total\ Valid\ Units\ of\ Work\ Received}$$

公式 8

对于基于请求的服务，工作单元是请求，例如 HTTP 请求。对于基于事件或基于任务的服务，工作单元是事件或任务，例如处理队列中的消息。这种可用性衡量标准在短时间间隔内是有意义的，例如一分

钟或五分钟内。它也最适合从精细的角度分析，例如基于请求的服务在 API 层面的可用性。下图显示了以这种方式计算出的可用性随着时间的变化。图表上的每个数据点都通过公式 (8) 计算得出，时间窗口为五分钟（您可以选择其他时间维度，例如一分钟或十分钟的间隔）。例如，数据点 10 显示可用性为 94.5%。这意味着在 t+45 到 t+50 分钟内，如果服务收到了 1000 个请求，则只有 945 个请求被成功处理。



单个 API 的可用性随时间变化的示例

图中还显示了 API 的可用性目标为 99.5%，为客户提供的服务级别协议 (SLA) 规定的可用性为 99%，高严重性警报阈值为 95%。如果没有不同的阈值作为背景，可用性图表就可能无法提供有关服务运行情况的重要信息。

我们也希望能够跟踪和描述更大的子系统（例如控制平面或整个服务）的可用性。实现这一目标的一种方法是计算每个子系统的每个五分钟数据点的平均值。生成的图表与上图类似，但输入值更多。构成服务的所有子系统都具有相同的权重。另一种方法可以是汇总从服务中的所有 API 收到的和被成功处理的所有请求，用来计算五分钟时间间隔内的可用性。

但是，后一种方法可能会隐藏吞吐量和可用性都比较低的 API。举一个简单的例子：假设某项服务有两个 API。

第一个 API 在五分钟内收到了 100 万个请求，并成功处理了 99.9 万个请求，可用性达到 99.9%。第二个 API 在五分钟内收到了 100 个请求，但仅成功处理了 50 个请求，可用性只有 50%。

如果我们将来自每个 API 的请求相加，则总共有 1000100 个有效请求，其中 999050 个请求被成功处理，因此该服务的总体可用性为 99.895%。但是，如果我们将前一种方法计算出的两个 API 的可用性取平均值，则得出的可用性为 74.95%，这也许更能反映实际体验。

这两种方法都没有错误，重点在于我们想通过可用性指标了解哪些信息。如果您的工作负载的每个子系统收到的请求量相似，则您可以优先考虑汇总计算所有子系统的请求。这种方法侧重于请求本身及其成功与否，以此作为可用性和客户体验的衡量标准。如果请求量存在差异，您也可以选择将子系统的可用性取平均值，均衡地体现每个子系统的重要性。这种方法侧重于子系统以及每个子系统反映客户体验的能力。

年度停机时间

第三种方法是计算年度停机时间。这种形式的可用性指标更适合长期目标的设定和回顾。它需要对工作负载的停机时间作出定义。然后，您可以计算工作负载未处于“中断”状态的时间占给定的总时间的比例，从而衡量可用性。

对于某些工作负载，停机时间可以定义为单个 API 或工作负载的单项功能在一分钟或五分钟时间间隔内的可用性降至 95% 以下的时间（如之前的可用性图表所示）。您也可以只考虑停机时间，因为它适用于关键数据平面操作的子集。例如，针对 SQS 可用性的 Amazon 消息收发 (SQS, SNS) 服务水平协议适用于 SQS Send、Receive 和 Delete API。

更大、更复杂的工作负载可能需要定义系统范围的可用性指标。对于大型电子商务网站，系统范围指标可以是客户订单率等指标。在这种情况下，停机时间可以是在任何五分钟时间段内，与预测数量相比，订单数量下降 10% 或以上的时间。

无论采用哪种方法，您都可以将所有停机时间相加以计算年度可用性。例如，如果在一个日历年内，有 27 段五分钟停机时间（其定义为任何数据平面 API 的可用性降至 95% 以下），则总停机时间为 135 分钟（有些五分钟时段可能是连续的，有些是单独的），则年度可用性为 99.97%。

这种衡量可用性的方法可以提供客户端和服务器端指标中缺少的数据和见解。例如，假设某个工作负载受损，错误率明显升高。这种工作负载的客户可能会完全停止调用其服务。他们可能已经激活了断路器，或者按照灾难恢复计划在其他区域中使用该服务。如果我们只测量失败的响应，那么工作负载在受损期间的可用性实际上可能会提高，但原因不是损害被减轻或消除，而是客户停止使用该工作负载。

延迟

最后，测量工作负载中的工作单元的处理延迟也很重要。按照既定的 SLA 完成工作是可用性的一种体现。如果在返回响应前客户端已经超时，则客户端会认为请求失败，工作负载不可用。但是在服务器端，请求可能会被视为已成功处理。

测量延迟为可用性的评估提供了另一个视角。百分位数和切尾均值都适合用来测量延迟。测量对象通常是第 50 个百分位数（P50 和 TM50）和第 99 个百分位数（P99 和 TM99）。我们应该使用金丝雀来测量延迟以便反映客户体验，同时也使用服务器端指标进行测量。当某些延迟百分位数（例如 P99 或 TM99.9）的平均值超过目标 SLA 时，您可以认为出现停机，并将其纳入年度停机时间的计算范围。

在上设计高度可用的分布式系统 AWS

前面的章节主要介绍了工作负载的理论可用性及其作用。它们是构建分布式系统时要记住的一组重要概念，可以为依赖项的选择和冗余的实现提供决策依据。

我们还研究了MTTDMTTR、和与可用MTBF性的关系。这一部分将会基于之前的理论提供一些实用的指导。简而言之，高可用性的工程工作负载旨在增加MTBF和减少MTTR以及MTTD。

最理想的结果是消除所有故障，但这并不现实。依赖项严重堆叠的大型分布式系统一定会发生故障。“一切都会失败”（参见 Werner VogelsCTO，Amazon.com，《亚马逊网络服务 10 年的十个教训》。）和 “你无法立法防止失败 [所以] 专注于快速检测和响应。”（参见 Chris Pinkham，Amazon EC2 团队创始成员，《为失败而ARC335设计：在上面架构弹性系统》）AWS

这意味着我们往往无法控制故障是否发生。我们可以控制的，是自己检测故障并采取措施的速度。因此，尽管提高可用性仍然MTBF是高可用性的重要组成部分，但客户可以控制的最重大变化是减少MTTD和MTTR。

主题

- 减少 MTTD
- 减少 MTTR
- 越来越多 MTBF

减少 MTTD

减少故障意味着要尽快发现故障。MTTD缩短MTTD是基于可观察性，或者你如何对工作负载进行检测以了解其状态。客户应监控其工作负载关键子系统中的客户体验指标，以便主动识别问题何时发生（请参阅附录1），MTTD并监控MTTR关键指标以了解有关这些指标的更多信息。)。 客户可以使用 Amazon CloudWatch Synthetics 创建用于监控您APIs和控制台的金丝雀，从而主动衡量用户体验。还有许多其他运行状况检查机制可用于最大限度地减少运行状况，例如 Elastic Load Balancing (ELB) 运行状况检查、Amazon Route 53 运行状况检查等。MTTD（参见 Amazon Builders's Library — 实施运行状况检查。）

您的监控机制还需要能够检测整个系统和单个子系统的部分故障。您的可用性、故障和延迟指标应使用故障隔离边界的维CloudWatch 度作为指标维度。例如，假设一个属于基于单元的架构、use1-az1 AZ、useast-1 区域中的单个EC2实例，它是工作负载更新的一部分，也是其控制平面子系统的一部分。API当服务器推送其指标时，它可以使用其实例 ID、可用区、区域、API名称和子系统名称作为维度。这让您可以进行观测，并针对每个维度设置警报以便检测故障。

减少 MTTR

发现故障后，剩下的MTTR时间是实际修复或减轻影响。要修复或缓解故障，您必须知道发生了什么问题。在这一阶段，有两组关键指标可以提供见解：1/影响评估指标和 2/运营状况指标。第一组指标可以告诉您故障的影响范围，并衡量受影响的客户、资源或工作负载的数量或百分比。第二组指标有助于确定产生影响的原因。发现原因后，操作人员和自动化机制可以响应和消除故障。有关这些指标的更多信息，请参阅附录 1 MTTD 和MTTR关键指标。

> 规则 9
>
> 可观察性和仪器化对于减少MTTD和MTTR至关重要。

绕过故障

减轻影响的最快方法是使用能够绕过故障的快速失效子系统。这种方法使用冗余，MTTR通过将故障子系统的工作快速转移到备用子系统来减少工作量。冗余范围从软件进程到EC2实例，再到多个区域AZs，再到多个区域。

备用子系统可以将MTTR降至几乎为零。恢复时间仅仅是将工作重定向到备件所花费的时间。这通常以最小的延迟发生，并允许在定义的范围内完成工作SLA，从而保持系统的可用性。MTTRs这会产生轻微的、甚至可能难以察觉的延迟，而不是长时间的不可用。

例如，如果您的服务使用的是 Application Load Balancer (ALB) 之后的EC2实例，则您可以以短至五秒的间隔配置运行状况检查，并且只需要两次失败的运行状况检查即可将目标标记为不健康。这意味着您可以在 10 秒钟内检测到故障，并停止向运行状况不佳的主机发送流量。在这种情况下，MTTR实际上与相同，MTTD因为一旦检测到故障，它也会得到缓解。

这就是高可用性或持续可用性工作负载想要实现的目标。我们希望快速检测到发生故障的子系统，将其标记为故障，停止向其发送流量，然后将流量发送到冗余子系统，从而快速绕过工作负载中的故障。

请注意，使用这种快速失效机制会让您的工作负载对瞬时错误非常敏感。在上面的示例中，我们应该确保负载均衡器运行状况检查仅对实例执行浅层或活性和本地运行状况检查，而不会对依赖项或工作流程执行测试（通常称为深度运行状况检查）。这有助于防止在影响工作负载的瞬时错误期间不必要地替换实例。

可观测性和检测子系统故障的能力对于成功绕过故障至关重要。您必须知道影响范围，这样才能将受影响的资源标记为运行状况不佳或出现故障，然后停止服务，将其绕过。例如，如果单个可用区出现部分服务受损，则您的检测工具需要能够识别出存在可用区范围内的问题，以便绕过该可用区内的所有资源，直到其恢复为止。

绕过故障可能还需要额外的工具，具体取决于环境。使用前面的示例，EC2实例位于后面ALB，假设一个可用区中的实例可能正在通过本地运行状况检查，但是孤立的可用区缺陷导致它们无法连接到另一可用区中的数据库。在这种情况下，负载均衡运行状况检查不会让这些实例停止服务。这就需要一种不同的自动化机制来从负载均衡器中移除可用区或让实例无法通过运行状况检查，而这又需要确定影响范围是否为可用区。对于未使用负载均衡器的工作负载，需要使用类似的方法来防止特定可用区中的资源接受工作单元或完全移除可用区的容量。

在某些情况下，我们无法将工作自动转移到冗余子系统，例如在没有主节点选择机制的情况下将主数据库故障转移到辅助数据库。这是 AWS 多区域架构中的一种常见场景。这种类型的故障转移需要一定的停机时间才能完成，无法立即恢复，并且会让工作负载在一段时间内没有冗余，因此需要让人参与决策过程。

MTTRs通过使用多区域故障转移自动化来绕过故障，可以采用不太严格的一致性模型的工作负载可以缩短时间。Amazon S3 跨区域复制或 Amazon DynamoDB 全局表等功能可以通过最终一致性复制来实现多区域故障转移。此外，当我们考虑CAP定理时，使用宽松的一致性模型是有益的。当有状态子系统的连接性受到网络故障影响时，如果工作负载重视可用性而不是一致性，它仍然可以提供非错误响应，这是绕过故障的另一种方式。

我们可以通过两种不同的策略来绕过故障。第一种策略是预先配置足够的资源来处理故障子系统的全部负载，从而实现静态稳定性。这可以是单个EC2实例，也可以是整个可用区的容量。在故障期间尝试配置新资源会增加恢复路径中的控制平面MTTR并增加对控制平面的依赖性。但是，预先配置资源需要额外付费。

第二种策略是将部分流量从出现故障的子系统路由到其他子系统，并卸除剩余容量无法处理的多余流量。在性能下降期间，您可以扩展新资源以替换出现故障的容量。这种方法的使用时间更长MTTR，会产生对控制平面的依赖，但待机、备用容量的成本更低。

恢复到已知良好状态

修复期间的另一种常见缓解措施是将工作负载恢复到先前的已知良好状态。如果导致故障的可能是近期发生的更改，则我们可以回滚该更改以便恢复到先前状态。

如果导致故障的可能是瞬时情况，那么重新启动工作负载可能会减轻影响。我们来分析一下这两种场景。

在部署过程中，最小化MTTD并MTTR依赖于可观察性和自动化。您的部署过程必须持续监视工作负载，以防出现错误率增加、延迟增加或异常情况。发现这些问题后，部署过程应该暂停。

我们可以采用多种部署策略，例如就地部署、蓝绿部署和滚动部署。每种策略都可以利用不同的机制来恢复到已知良好状态。系统可以自动回滚到先前状态、将流量转移回蓝色环境，或者要求手动干预。

CloudFormation 提供了自动回滚的功能，这是其创建和更新堆栈操作的一部分，也是如此AWS CodeDeploy。 CodeDeploy 还支持蓝/绿和滚动部署。

要利用这些功能并最大限度地减少您的能力MTTR，请考虑通过这些服务自动部署所有基础架构和代码。在无法使用这些服务的情况下，可以考虑使用实现传奇模式 AWS Step Functions 来回滚失败的部署。

在考虑重启时，我们有很多不同的方法。我们可以重启服务器（用时最长），也可以重新启动线程（用时最短）。下表列出了一些重启方法和完成的大致时间（体现数量级差异，数字并不准确）。

故障恢复机制	估计 MTTR
启动和配置新虚拟服务器	15 分钟
重新部署软件	10 分钟
重启服务器	5 分钟
重启或启动容器	2 秒
调用新无服务器函数	100 毫秒
重启进程	10 毫秒
重启线程	10 微秒

查看该表，使用容器和无服务器函数（例如 AWS Lambda）有一些明显的好处。MTTR它们MTTR比重启虚拟机或启动新虚拟机快几个数量级。但是，通过软件模块化来实现故障隔离也有好处。如果能讲故障限制在单个进程或线程内，那么从故障中恢复的速度要比重启容器或服务器快得多。

作为一般的恢复方法，您可以从下到上做出选择：1/重启、2/重新引导、3/重新创建映像/重新部署、4/替换。但是，进入重新引导步骤后，绕过故障通常是更快的方法（一般最多需要 3-4 分钟）。因此，为了在尝试重启后最快速地缓解影响，请绕过故障，然后在后台继续恢复过程以恢复工作负载的容量。

> 规则 10
>
> 侧重于缓解影响，而不是解决问题。以最快的速度恢复正常运行。

故障诊断

检测到故障之后，修复过程进入诊断阶段。这是操作人员试图确定问题所在的阶段。这一过程可能包括查询日志、查看运行状况指标或登录主机进行故障排除。所有这些操作都需要时间，因此创建工具和运行手册来加快这些操作MTTR也有助于减少。

运行手册和自动化

同样，在确定了问题和修复措施之后，操作人员通常需要执行一些步骤来实现修复目的。例如，在发生故障后，修复工作负载的最快方法可能是重启工作负载，而这可能涉及多个有序的步骤。您可以使用运行手册来自动执行这些步骤或为操作人员提供具体指导，从而加快修复流程并降低执行出现操作的风险。

越来越多 MTBF

提高可用性的最后一个要素是提高MTBF. 这既适用于软件，也适用于用于运行该软件的 AWS 服务。

越来越多的分布式系统 MTBF

增加的一种方法MTBF是减少软件中的缺陷。我们可以通过多种方式来实现这一目的。客户可以使用诸如 Amazon CodeGuru Reviewer 之类的工具来查找和修复常见错误。在将软件部署到生产环境之前，您还应该对软件进行全面的同行代码审查、单元测试、集成测试、回归测试和负载测试。增加测试中的代码覆盖率有助于确保即使是不常见的代码执行路径也能得到测试。

部署较小规模的更改也可以降低更改的复杂性，从而帮助防止意外结果。每项活动都提供了在缺陷被调用之前识别和修复缺陷的机会。

防止故障的另一种方法是定期测试。实施混沌工程可以帮助测试工作负载如何发生故障、验证恢复程序，并有助于在生产环境中出现故障之前发现和修复故障。客户可以将 AWS Fault Injection Simulator 用作混沌工程实验工具。

建立容错能力是防止分布式系统出现故障的另一种方法。快速失效模块、采用指数回退和抖动的重试、事务和幂等性都是可以帮助工作负载获得容错能力的技术。

交易是一组符合ACID属性的操作。这些特性如下所示：

- 原子性 — 所有操作要么全部发生，要么全部不发生。
- 一致性 — 每个事务都让工作负载处于有效状态。
- 隔离性 — 并发执行的事务会让工作负载处于相同状态，如同它们是按顺序执行的一样。

- 持久性 — 事务提交之后，即使工作负载出现故障，其所有影响也会保持不变。

采用指数回退和抖动的重试可以克服由海森堡错误、过载或其他条件引起的暂时故障。当事务具有幂等性时，它们可以多次重试而不会产生副作用。

对于海森堡错误对容错硬件配置的影响，我们可以将其完全忽略，因为海森堡错误同时出现在主子系统和冗余子系统上的可能性微乎其微。（参见 Jim Gray，“Why Do Computers Stop and What Can Be Done About It?”，1985 年 6 月，Tandem 技术报告 85.7。）在分布式系统中，我们希望通过软件实现同样的结果。

出现海森堡错误时，软件必须快速检测到错误的操作并快速失效，以便可以重试。这通过防御性编程以及验证输入、中间结果和输出来实现。此外，进程是隔离的，不与其他进程共享任何状态。

这种模块化方法可以确保故障的影响范围受到限制。进程会独立失效。当某个进程失效时，软件应该使用“进程对”来重试该工作，这意味着新进程可以承担失效进程的工作。为了保持工作负载的可靠性和完整性，应将每项操作视为一项ACID事务。

这可以让进程的失效不会因为中止事务并回滚所做的任何更改而破坏工作负载的状态。这可以让恢复过程在已知良好状态下重试事务并正常重启。这就是软件容许海森堡错误的方式。

但是，您的目标不应该是让软件能够容许海森堡错误。您必须在工作负载进入生产环境之前就发现并消除缺陷，因为任何程度的冗余都无法实现正确的结果。（参见 Jim Gray，“Why Do Computers Stop and What Can Be Done About It?”，1985 年 6 月，Tandem 技术报告 85.7。）

增加的最终方法MTBF是缩小失败的影响范围。正如前面的容错能力和故障隔离部分所述，实现这一目标的主要方式是通过模块化来创建故障容器，从而实现故障隔离。降低故障率可以提高可用性。 AWS 使用诸如将服务划分为控制平面和数据平面、可用区独立性 (AZI)、区域隔离、基于单元的架构和洗牌分片等技术来提供故障隔离。这些模式也可以由 AWS 客户使用。

例如，假设工作负载将不同客户置于其基础架构的不同故障容器中，每个容器最多为 5% 的客户提供服务。其中一个故障容器中发生了一个事件，该事件让 10% 的请求延迟超过了客户端超时时间。在这次事件中，对于 95% 的客户来说，该服务具有 100% 的可用性。对于剩余 5% 的客户来说，该服务具有 90% 的可用性。这时的可用性为 1 − (5% 的客户×10% 的请求) = 99.5%，而不是 100% 的客户的 10% 的请求无效（可用性为 90%）。

> 规则 11
>
> 故障隔离通过降低总体故障率来缩小影响范围并增加工作负载。MTBF

依赖性增加 MTBF

增加 AWS 依赖性的第一种方法MTBF是使用故障隔离。许多 AWS 服务在可用区提供一定程度的隔离，这意味着一个可用区的故障不会影响另一个可用区的服务。

在多个中使用冗余EC2实例可AZs提高子系统的可用性。AZI在单个区域内提供备用功能，允许您提高AZI服务的可用性。

但是，并非所有 AWS 服务都在可用区级别运行。许多其他服务可以提供区域性隔离。在这种情况下，如果区域性服务的设计可用性无法实现您的工作负载所需的总体可用性，则您可以考虑采用多区域方法。每个区域都对服务进行隔离的实例化，相当于创建备件。

有多种服务可以帮助简化多区域服务的构建。例如：

- Amazon Aurora 全球数据库
- Amazon DynamoDB 全局表
- 亚马逊 ElastiCache (RedisOSS) — 全球数据存储
- AWS 全球加速器
- Amazon S3 跨区域复制
- Amazon Route 53 应用程序恢复控制器

本文并未深入探讨构建多区域工作负载的策略，但您应该权衡多区域架构的可用性优势和实现所需可用性目标所需的额外成本、复杂性和运营实践。

下一个增加依赖性的方法MTBF是将工作负载设计为静态稳定。例如，您有一个提供产品信息的工作负载。当客户针对产品发出请求时，您的服务会向外部元数据服务发出请求以检索产品详细信息。然后，您的工作负载会将所有这些信息返回给用户。

但是，如果元数据服务不可用，您的客户发出的请求就会失败。因此，您可以将元数据异步拉取或推送到本地服务，用于回复请求。这样您就无需从关键路径同步调用元数据服务。

此外，因为您的服务在元数据服务不可用时仍然保持可用，所以您可以在计算可用性时删除这个依赖项。本示例假设元数据不会经常更改，并且提供过时的元数据要好于请求失败。另一个类似的例子是serve-staleDNS，它允许在TTL过期后将数据保存在缓存中，并在刷新后的答案不容易获得时用于响应。

增加依赖性的最后一种方法MTBF是缩小失败的影响范围。如上文所述，故障不是一个二元事件，存在着不同的程度。利用模块化设计，我们可以将故障限制在故障容器所服务的请求或用户范围内。

这样可以减少事件期间的故障，从而通过限制影响范围来最终提高整个工作负载的可用性。

减少常见的影响源

1985年，Jim Gray 在 Tandem Computers 的一项研究中发现，故障主要源自两种事物：软件和操作。（参见 Jim Gray，“Why Do Computers Stop and What Can Be Done About It?”，1985 年 6 月，Tandem 技术报告 85.7。）即使在 36 年之后，情况仍然如此。尽管技术取得了进步，但这些问题并没有简单的解决方案，而且故障的主要原因也没有改变。本节开头探讨了如何解决软件故障，因此这里的重点将放在操作和降低故障频率上。

稳定性与功能的比较

如果再看一次 the section called “分布式系统可用性” 部分的软件与硬件故障率图表，我们可以发现软件的每个版本都会引入缺陷。这意味着工作负载的任何变化都会提高故障风险。这些变化通常是新功能之类的东西，会带来必然的结果。可用性更高的工作负载会更重视稳定性而不是新功能。因此，提高可用性的一种最简单的方法，就是降低部署频率或减少提供的功能。部署频率更高的工作负载的可用性天然低于比不频繁部署的工作负载。但是，不添加新功能的工作负载无法满足客户需求，并且作用会随着时间的推移而降低。

那么，我们如何继续创新并安全地发布新功能呢？ 答案是标准化。正确的部署方式是什么？ 如何确定部署顺序？ 测试标准是什么？ 不同阶段之间要等待多长时间？ 单元测试是否涵盖了足够的软件代码？ 标准化可以回答这些问题，并防止由于未进行负载测试、跳过部署阶段或过快地部署到太多主机等而引发问题。

实现标准化的方法是推行自动化。自动化可以减少人为错误，让计算机处理其擅长的任务，即以同样的方式反复执行同样的操作。将标准化与自动化结合的方法是设定目标。目标包括不进行手动更改、仅通过临时授权系统访问主机、为每个API系统编写负载测试等。卓越运营是一种文化规范，可能需要进行重大变革。根据目标来确定和跟踪效果有助于推动文化变革，这将对工作负载的可用性产生广泛影响。AWS Well-Architected 卓越运营支柱针对卓越运营供了全面的最佳实践。

操作人员安全

导致故障的运营事件的另一个主要引发因素是人。人会犯错误。他们可能会使用错误的凭证、输入错误的命令、过早按回车键，或者错过关键步骤。始终采取手动操作会导致错误，从而引发故障。

造成操作人员错误的一项主要原因是用户界面混乱、不直观或不一致。Jim Gray 在 1985 年的研究中还指出，“要求操作人员提供信息或执行某些功能的界面必须简单、一致并能容许操作人员错误。”（参见 Jim Gray，“Why Do Computers Stop and What Can Be Done About It?”，1985 年 6 月，Tandem 技术报告 85.7。）这一见解在今天仍然是正确的。在过去的三十年中，整个行业中有许多例子表明，混乱或复杂的用户界面、缺乏确认或说明，甚至只是不友好的人类语言就导致操作人员犯下了错误。

规则 12

让操作员可以轻松采取正确操作。

防止过载

最后一个常见影响来源是您的客户，即工作负载的实际用户。成功的工作负载往往会被大量使用，但有时这种使用量会超出工作负载的扩展能力。可能出现的情况有很多，比如磁盘已满、线程池耗尽、网络带宽饱和，或者达到数据库连接限制。

没有万无一失的方法可以消除这些故障，但是通过运行状况指标对容量和使用情况进行主动监控，我们就能在可能发生这些故障时提前收到警告。卸除负载、断路器以及采用指数回退和抖动的重试等技术可以帮助我们尽可能减少影响并提高成功率，但这些都是发生故障之后采取的措施。基于运行状况指标的自动扩展可以帮助降低过载导致的故障的发生频率，但可能无法足够快地响应使用率的变化。

如果您需要确保客户具有持续可用的容量，则必须在可用性和成本之间做出权衡。要确保容量不足不会导致不可用，您可以为每位客户提供配额并确保扩展工作负载的容量，以便提供 100% 的分配配额。当客户超过配额时，他们会受到限制，这不是故障，也不会影响可用性的计算。您还需要密切跟踪您的客户群并预测未来的使用情况，以便预置足够的容量。这样可以确保您的工作负载不会因为客户过度使用而产生故障。

- Amazon Builders' Library – 通过卸除负载来避免过载
- Amazon Builders' Library – Fairness in multi-tenant systems

我们来看一个提供存储服务的工作负载。工作负载中的每台服务器可以支持每秒 100 次下载，为客户提供每秒 200 次下载的配额，共有 500 个客户。要支持这一数量的客户，该服务需要提供每秒 10 万次的下载容量，这需要 1000 台服务器。如果任何客户超出其配额，他们就会受到限制，这可以确保其他所有客户都有足够的容量。这是一个简单的示例，说明了一种在不拒绝工作单元的情况下避免过载的方法。

结论

在本文中，我们确立了 12 条高可用性规则。

- 规则 1 — 降低故障频率（提高 MTBF）、缩短故障检测时间（缩短 MTTD）和缩短修复时间（缩短 MTTR）是提高分布式系统可用性的三项因素。
- 规则 2 — 工作负载中的软件可用性是决定工作负载总体可用性的一项重要因素，应与其他组件同等重视。
- 规则 3 — 减少依赖项可以对可用性产生积极影响。
- 规则 4 — 通常应该选择可用性目标等于或高于工作负载目标的依赖项。
- 规则 5 — 使用备件以便提高工作负载中的依赖项的可用性。
- 规则 6 — 备件的成本效益存在上限。利用最少的备件来实现所需的可用性。
- 规则 7 — 不要在数据平面中的控制平面上设置依赖项，尤其是在恢复期间。
- 规则 8 — 尽可能松散地耦合依赖项，让工作负载在依赖项受损时也能正常运行。
- 规则 9 — 可观测性和检测对于缩短 MTTD 和 MTTR 至关重要。
- 规则 10 — 侧重于缓解影响，而不是解决问题。以最快的速度恢复正常运行。
- 规则 11 — 故障隔离可以降低总体故障率，从而缩小影响范围并延长工作负载的 MTBF。
- 规则 12 — 让操作员可以轻松采取正确操作。

缩短 MTTD 和 MTTR 以及延长 MTBF 可以提高工作负载可用性。总结一下，我们探讨了涵盖技术、人员和流程的以下提高可用性的方法。

- MTTD
  - 通过主动监控客户体验指标来缩短 MTTD。
  - 利用精细的运行状况检查实现快速故障转移。
- MTTR
  - 监控影响范围和运行状况指标。
  - 按照 1/重启、2/重新引导、3/重新创建映像/重新部署、4/替换的顺序来缩短 MTTR。
  - 通过了解影响范围来绕过故障。
  - 使用重启时间更快的服务，例如虚拟机或物理主机上的容器和无服务器功能。
  - 尽可能自动回滚发生故障的部署。
  - 为诊断操作和重启程序建立运行手册和操作工具。

- MTBF
  - 在软件发布到生产环境之前，通过严格的测试来消除软件中的错误和缺陷。
  - 实施混沌工程和故障注入。
  - 在依赖项中使用适量的备件来容许故障。
  - 通过故障容器最大限度地减少故障的影响范围。
  - 实施部署标准和变更标准。
  - 设计简单、直观、一致且有据可查的操作人员界面。
  - 针对卓越运营设定目标。
  - 当可用性是工作负载的关键维度时，优先考虑稳定性而不是新功能的发布。
  - 通过节流和/或卸除负载来实现使用配额，以避免过载。

记住自己永远无法完全成功地防止故障。专注于具有最佳故障隔离的软件设计，以便限制影响的范围和程度，理想情况下将影响保持在“停机”阈值以下，并采取非常快速、非常可靠的检测和缓解措施。现代分布式系统仍然需要将故障视为不可避免的，并且需要在各个层面上进行设计以实现高可用性。

附录 1 — MTTD 和 MTTR 关键指标

以下是一个分析与观察标准化框架，可以帮助缩短事件期间的 MTTD 和 MTTR。

客户体验指标。这些指标可以体现服务是否响应迅速，能够处理客户的请求。例如控制平面延迟。这些指标衡量错误率、可用性、延迟、容量和限制率。

影响评估指标。这些指标可以让用户深入了解事件的影响范围。例如受数据平面事件影响的客户数量或百分比。衡量受影响的事物的数量或百分比。

运营状况指标。这些指标可以体现服务是否响应迅速，能够处理客户的请求，但侧重于常见的基础设施子系统和资源。例如，EC2 实例集的 CPU 使用率百分比。这些指标应该衡量利用率、容量、吞吐量、错误率、可用性和延迟。

贡献者

本文档的贡献者包括：

- Michael Haken，Amazon Web Services 首席解决方案架构师

延伸阅读

如需了解其他信息，请参阅：

- Well-Architected Reliability Pillar
- Well-Architected Operational Excellence Pillar
- Amazon Builders' Library — 确保部署期间安全回滚
- Amazon Builders' Library – Beyond five 9s: Lessons from our highest available data planes
- Amazon Builders' Library — 自动实现无需干预的安全部署
- Amazon Builders’ Library – Architecting and operating resilient serverless systems at scale
- Amazon Builders’ Library – Amazon’s approach to high-availability deployment
- Amazon Builders’ Library – Amazon’s approach to building resilient services
- Amazon Builders’ Library – Amazon’s approach to failing successfully
- AWS 架构中心

文档历史记录

如需获取有关本白皮书更新的通知，请订阅 RSS 源。

变更	说明	日期
初次发布	白皮书首次发布。	2021 年 11 月 12 日

> **Note**
>
> 要订阅 RSS 更新，您必须为当前使用的浏览器启用 RSS 插件。

注意事项

客户有责任对本文档中的信息进行单独评测。本文档：（a）仅供参考，（b）代表当前的 AWS 产品和实践，如有更改，恕不另行通知，以及（c）不构成 AWS 及其附属公司、供应商或许可方的任何承诺或保证。AWS 产品或服务“按原样”提供，不附带任何明示或暗示的保证、陈述或条件。AWS 对其客户承担的责任和义务受 AWS 协议制约，本文档不是 AWS 与客户直接协议的一部分，也不构成对该协议的修改。

© 2021，Amazon Web Services, Inc. 或其附属公司。保留所有权利。

AWS 术语表

有关最新的 AWS 术语，请参阅《AWS 词汇表参考》中的 AWS 词汇表。

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。