键值存储正确性验证方法

夏谦[1)]

[1)](华中科技大学 湖北 武汉 430070)

摘　要　应用和系统崩毁后，可能会导致状态不一致，数据丢失等严重后果。应用程序在系统崩毁后依赖持久存储来恢复一致性状态，即需要非易失性内存键值存储保持崩毁一致性，又需要定义和检查文件系统的崩毁一致性模型。非易失性内存存储崩毁一致性检测可能会遇到测试空间爆炸，有一种新的崩毁一致性测试框架 Witcher， 它系统地探索 NVM 状态测试空间（没有测试空间爆炸）并自动验证每个可行的 NVM 状态是否一致（没有手动测试预言机）。Amazon S3 正在开发一种新键值存储结点系统，这是一个规模很大且非常复杂的系统，需要保障客户数据的持久性和正确性。键值存储系统的正确性至关重要，但要验证一个键值存储系统的正确性是非常困难的，需要考虑各种情况的发生，在系统发生故障之后仍然保持其正确性，维护数据的持久化。关于键值存储系统正确性验证研究有很多，本文主要参考了分布式存储正确性验证、可串行的事务键值存储验证、非易失内存键值存储崩溃一致性检测、轻量级方法验证键值存储系统。这些论文在验证键值存储系统的正确性时，都无一例外地只验证了系统的一部分，而不是完全验证系统正确性，因为这样所耗费的时间和成本太大。他们都在各自的研究方向上，采取了经过测试真实可行的方法，可以明显地发现键值存储系统的正确性错误。

关键字　键值存储；正确性验证；崩毁一致性

中国法分类号　T　　**DOI 号**　10

The method of key-value storage correctness verification

XIA Qian[1)]

[1)](Huazhong University of Science and Technology， Wuhan，Hubei 430070)

Abstract　After the application and system collapse, it may lead to inconsistent state, data loss and other serious consequences. Applications rely on persistent storage to restore consistency after system crash, that is, they need nonvolatile memory key value storage to maintain crash consistency, and they also need to define and check the crash consistency model of file system. Nonvolatile memory collapse consistency detection may encounter test space explosion. There is a new collapse consistency testing framework, Witcher, which systematically explores the NVM state test space (no test space explosion) and automatically verifies whether each feasible NVM state is consistent (no manual test Oracle). Amazon S3 is developing a new key value storage node system, which is a large-scale and very complex system and needs to ensure the persistence and correctness of customer data. The correctness of key value storage system is very important, but it is very difficult to verify the correctness of a key value storage system. We need to consider the occurrence of various situations, maintain its correctness after system failure, and maintain the persistence of data. There are many researches on the correctness verification of key value storage system. This paper mainly refers to distributed storage correctness verification, serializable transaction key value storage verification, non-volatile memory key value storage crash consistency detection, and lightweight methods to verify key value storage system. When verifying the correctness of the key value storage system, these papers, without exception, only verify part of the system rather than completely verify the correctness of the system, because it costs too much time and cost. In their respective research directions, they have adopted tested and

practical methods, which can obviously find the correctness and error of the key value storage system

Key Words Key value storage; Correctness verification; Collapse consistency

1 引 言

应用程序崩毁是一件很令人头疼的事情，可能会导致应用程序状态损坏，最坏的情况下会导致灾难性的数据丢失。在突然断电的情况下，大部分的随机存取存储器中的数据会在电力切断以后很快消失，因此随机存储器是一种易失性存储器，如果在断电前，在随机存储器中的数据没有持久存储下来，应用程序的状态就会损坏，无法恢复。非易失性主存储器（Non Volatile Memory）的出现使开发具有崩毁一致性的应用软件成为可能，而无需支付存储堆栈开销。非易失存储器具有非易失、按字节存取、存储密度高、低能耗、读写性能接近 DRAM 的特点，但读写速度不对称，读远快于写，寿命有限，断电时，所存储的数据不会消失。非易失存储器价格昂贵，还有使用寿命，且原来的内存模型不兼容 NVM，其完全取代目前流行的电脑主存动态随机存储器（DRAM）或大规模使用，都不太现实。所以存在易失性缓存的情况下，构建正确的崩毁一致性程序仍然非常具有挑战性。非易失性主存储器技术正在兴起，它提供存储的持久性以及传统的 DRAM 特性，例如字节寻址能力和低访问延迟。使用常规加载和存储指令直接访问 NVM 的能力提供了构建崩毁一致性软件的新机会。在软件崩溃或突然断电的情况下，程序可以从持久的 NVM 状态恢复到一致的状态。但是，很难设计和实现正确且高效的崩溃一致性 NVM 程序，易失性缓存上的 NVM 数据可能不会在崩溃后持久化。缓存还可以按任意顺序逐出缓存行，因此，对不同 NVM 位置的更新可能不会以与程序存储顺序相同的顺序持久化。为确保崩溃一致性，当前的 NVM 编程模型要求（应用程序或库）开发人员显式添加缓存行刷新和存储栅栏（或称为屏障）指令并设计自定义机制以强制执行适当的持久性排序和原子性保证。并且误用 NVM 原语可能会导致正确性错误，例如错位的刷新/栅栏，或性能错误，例如，冗余的刷新/栅栏。正确性错误特别重要，因为程序可能会在崩溃时导致不一致的 NVM 状态，并且由于永久性数据损坏、不可恢复的数据丢失等而不能恢复到正确的一致性状态。一种新的系统崩溃一致性测试框架 Witcher 用于测试 NVM 的键值存储。

在非易失性存储具有崩溃一致性的基础上，应用程序在系统崩溃后，依赖持久存储来恢复状态，但是可移植操作系统接口（POSIX）文件没有定义崩溃的可能结果，因此应用程序编写者很难正确理解文件系统操作之间的顺序和依赖关系。构建应用程序的崩溃一致性模型也非常重要，类似于内存一致性模型。崩毁一致性模型包括证明允许和禁止行为的测试，以及公理和规范操作。许多应用程序与文件系统交互，并将它们用作持久存储，在硬件或软件崩溃时将被保留。为了实现他们想要的完整性属性，这些应用程序必须正确使用文件系统接口。不这样做可能会导致应用程序状态损坏和灾难性的数据丢失。应用程序编写者面临的主要挑战是了解文件系统在系统崩毁时的准确行为。POSIX 标准基本上没有提交文件系统接口在发送崩溃时应提供的保障。在实践中，应用程序编写者对文件系统提供的崩溃保证做出假设，并将他们的应用程序基于这些假设，对崩溃保证过于乐观，会导致严重的数据丢失，但是过于保守会导致昂贵且不必要的同步，从而消耗能源、性能和硬件寿命。因此，以明确且可访问的方式描述文件系统的崩溃保证至关重要。可以将文件系统在崩溃时的行为描述为崩溃一致性模型，类似于用于描述宽松内存排序的内存一致性模型。

今天的云数据库提供强大的属性，包括可序列化性，有时称为黄金标准数据库正确性属性。但云数据库是复杂的黑匣子，运行在与其客户端不同的管理域中。因此，

客户可能想知道数据库是否符合他们的合同。为此，我们要保证键值存储的正确性，持续验证其事务性。云数据库是分布式的，为了验证分布式系统，先前的工作引入了一种方法来验证在单个机器上运行的代码以及当这些机器通过异步分布式环境交互时整个系统的正确性。该方法既不需要特定于域的逻辑也不需要工具。然而，分布式系统只是与异步环境交互的系统代码这一更普遍现象的一个例子。我们认为存储系统的软件可以被类似地看待。有研究者引入一种新技术自动验证分布式存储系统的正确性，以一种可扩展的方式指定了崩溃安全性的方法。Amazon 网络工程师，在前人工作的基础上，将轻量级形式化方法用于验证云对象存储服务的键值存储节点的正确性，这种验证不是实现完整的形式验证，而是强调自动化、可用性以及随着软件及其规范随着时间的推移不断发展而不断确保正确性的能力。他们将正确性分解为独立的属性，每个属性都由最合适的工具检查，并开发可执行的参考模型作为要针对实现进行检查的规范。这种轻量级方法用于指定和验证生产规模存储系统，以应对频繁的更改和新功能。本文将根据所参考文献，概述 NVM 内存键值存储崩溃一致性的测试，文件系统崩溃一致性模型的指定和检查，以及使用轻量级形式化方法验证 Amazon S3 键值存储结点。

2 原理和优势

保证键值存储的正确性，使得应用程序和存储系统在崩毁后恢复到一致性状态成为可能。首先，非易失性内存键值存储的崩溃一致性是关键，在遇到突然断电等特殊情况，NVM 中还有保留有应用程序或存储系统的一些持久性数据，程序和系统可以从持久的 NVM 状态中恢复一致的状态。本文介绍一种新的系统崩溃一致性测试框架 Witcher，它检测基于 NVM 的持久键值存储和底层 NVM 库中的正确性和性能错误，没有测试空间爆炸，也没有手动注释或崩溃一致性检查器。为了检测正确性错误，Witcher 通过分析数据和控制 NVM 访问之间的依赖关系来自动推断可能的正确性条件。然后 Witcher 通过检查有和没有崩溃的执行之间的输出等效性来验证是否有任何违反它们的行为是真正的崩溃一致性错误。为了解决测试空间的挑战，Witcher 通过分析 NVM 访问之间的程序数据/控制依赖关系，推断出一组可能正确的条件，这些条件被认为是崩溃一致的。 Witcher 然后只测试那些违反可能正确性条件的 NVM 状态，显着减少 NVM 状态测试空间。尽管 Witcher 可能会不合理地修剪一些 NVM 状态以进行测试，但评估表明它可以有效地修剪测试空间并检测许多新的正确性错误。由于可能正确性条件源自常见的 NVM 编程模式，因此它们适用于非键值应用程序。为了缓解测试预言机问题，Witcher 在有和没有（模拟）崩溃的程序执行之间采用了输出等效性检查。如果从违反可能正确性条件的 NVM 状态恢复的键值存储产生与没有崩溃的执行不同的输出，那么可以自信地得出结论，该程序不是崩溃一致的，并且违规确实是一个真正的崩溃一致性错误。输出等效性检查需要确定性并检查持久的线性化能力。因此，Witcher 可能不适用于允许非确定性输出（例如时间戳）或非持久线性化行为的 NVM 程序。Witcher 是第一个使用与应用程序无关的规则来查找特定于应用程序的正确性错误的 NVM 测试工具。Witcher 不会受到测试空间爆炸的影响，也不需要手动测试预言机来检测它们。当前的 Witcher 原型专注于测试键值存储，其中操作接口是众所周知的，因此可以自动执行输出等效性检查。所提出的想法可以扩展并应用于键值存储之外的其他 NVM 程序。一系列试图彻底测试所有可能的 NVM 状态的测试工具，有可能检测到许多错误，但经常受到测试空间爆炸的影响。此外，对于测试语言机，现有的 NVM 测试工具要求用户提供手动设计的、特定于应用程序的一致性检查器来验证，设计特定于应用程序的测试预言机不仅需要大量的手动工作，而且容易出错。本文将介绍一种新的崩毁一致性测试框架，它系统地探索测试空间，

没有测试空间爆炸，并自动验证每个可行的 NVM 状态是否一致，使用与应用程序无关的规则检测特定于应用程序的正确性错误，还可以通过分析执行跟踪来识别性能错误。

文件系统崩毁一致性模型类似于内存一致性模型，它描述了文件系统在崩溃时的行为。崩毁一致性模型包括证明允许和禁止行为的试金石测试，以及公理和操作规范。与内存一致性模型一样，崩溃一致性模型有两种形式：Litmus 测试：演示文件系统在崩溃时允许或禁止的行为的小程序；正式规范：分别使用逻辑和（非确定性）状态机对碰撞一致性行为的公理化和操作性描述。这些形式的规范服务于不同的目的，两者都不能取代另一个。正式规范提供了（禁止）允许的崩溃行为的完整描述，因此，为应用程序崩溃保证的自动推理提供了基础。另一方面，石蕊测试提供精确和直观的描述，非常适合与应用程序开发人员交流，以及根据文件系统实现验证形式模型。崩毁一致性模型可以帮助我们编写安全崩毁的应用程序。

Amazon S3 的工程师在验证键值存储结点的正确性时，寻求由非正规方法专家自动化和可用的轻量级方法来自行验证新功能；他们标是将形式化方法主流化到我们的日常工程实践中。他们寻求形式化方法验证 API 级调用的实现、功能正确性、磁盘数据结构的崩溃一致性以及 API 调用和后台维护任务的并发正确性。Amazon S3 工程师确定了一种包含三个要素的方法。首先，使用定义系统预期语义的可执行参考模型提取所需行为的规范。参考模型定义了系统中某个组件允许的顺序、无崩溃行为。参考模型使用与实现相同的语言编写并嵌入其代码库中，允许工程团队编写和维护它们，而不是作为单独的专家编写的工件而萎靡不振。为 ShardStore（Amazon S3 一种新的存储结点）开发的参考模型是小的可执行规范（实现代码的 1%），强调简单性；例如，日志结构化合并树，实现的参考模型是哈希映射。我们还重用这些参考模型作为 ShardStore 单元测试的模拟实现，有效地要求工程师在开发新代码时自己更新参考模型规范。其次，通过检查 ShardStore 实现是否改进了参考模型来验证它是否满足期望的正确性属性。为了充分利用可用的轻量级形式方法工具，他们分解这些属性并为每个组件应用最合适的工具。为了功能的正确性，应用基于属性的测试来检查实现和模型在 API 操作的随机序列上是否一致。为了崩溃一致性，增加了参考模型以定义软更新允许在崩溃期间丢失的特定最近突变，并再次应用基于属性的测试来检查包括任意崩溃的历史记录的一致性。对于并发，应用无状态模型检查来表明实现是可线性化的，这与参考模型有关。在所有三种情况下，检查器都是高度自动化的，基础设施很紧凑（组合的属性定义和测试工具占 ShardStore 代码库的 12%）。第三，Amzon S3 工程师通过培训工程团队开发他们自己的参考模型和检测。Amazon S3 将轻量级正式方法集成到生产工程团队的实践中并交出由他们而不是正式方法专家维护的验证工件方面的经验。

3 研究进展

3.1 非易失性内存键值存储测试框架 Witcher

最近，已经提出了几种解决方案来检测 NVM 程序中的持久性错误，但是有两个关键问题，分别是针对测试可能的 NVM 状态和需要手动测试预言机来验证测试的正确性。这些问题仍然使有效测试 NVM 支持的持久键值存储具有挑战性。非易失性内存存储测试框架 Witcher，不同于已有的一些测试工具，例如 Yat，PMReorder，它们试图彻底测试所有可能的 NVM 状态，者虽然可能会发现很多错误，但经常会遇到测试空间爆炸的情况。持久的键值存储为了限制搜索成本，通常由具有重新平衡操作的数据结构支持，如在哈希表中重新散列，在 B 树种拆分合并。要触发这类操作并检查其中的持久性错误，需要具有大量的操作和长时间执行的测试用例，使得穷举测试在实践中不可行。更何况，现有的 NVM

测试工具要求用户提供手动设计的，用于特定应用程序的一致性检查器来验证被测程序。崩毁一致性框架 Witcher 既没有测试空间爆炸，也不需要设计特定应用程序的检查器，而是使用与应用程序无关的规则来检测应用程序中的正确性错误。Witcher 能在崩毁一致性 NVM 程序中发现正确性错误和性能错误。

3.1.1 正确性错误和性能错误

由于处理器可以按任意顺序逐行输出缓存，并且不支持多个 NVM 位置的原子更新，因此崩毁一致性是根据程序语义保证 NVM 位置的持久性排序和原子性问题，违法这些是 NVM 程序中正确性错误的主要原因。即持久化顺序违规和持久化原子违规，很多崩毁一致性和恢复机制依赖于特定应用程序 NVM 变量的持久性排序，但是，当更新多个 NVM 位置时，有缺陷的 NVM 程序可能无法使用缓存行刷新和存储栅栏指令来维持正确的持久性排序。为了确保 NVM 数据的完整性，许多 NVM 程序依赖于 NVM 变量的原子更新，可是有缺陷的 NVM 程序可能无法正确地在多个 NVM 更新之间强制执行持久化原子性，如果程序在 NVM 更新过程中崩毁，可能出现不一致的状态。

以前的研究发现，性能错误在现实世界的 NVM 程序中很普遍。性能错误不会导致不一致的状态，但需要开发人员花费大量的时间和精力来发现和修复它们。与之前的工作类似，NVM 性能错误分类如下：非持久的性能错误，额外的冲洗和栅栏，额外的日志。

3.1.2 正确性错误查找

为了检测正确性错误，Witcher 推断可能的正确性条件并执行输出等效性检查以验证违反它们的 NVM 状态。

Witcher 提出了一种分析 NVM 访问之间的程序数据/控制依赖性的新方法，以推断在 NVM 访问之间强制执行持久性排序和持久性原子性保证的可能正确性条件。主要观察结果是程序员经常以数据/控制依赖关系的形式在源代码中留下一些关于他们想要确保什么的提示，可以推断出相应的可能的持久性排序/原子性条件。Witcher 仅测试违反推断的可能正确性条件的 NVM 状态。通过这种方式，Witcher 使用可能的正确性条件来减少 NVM 状态测试空间，而无需手动注释。Witcher 不需要事先了解真相，也不假设条件总是正确的；如果两个条件相互矛盾，我们会测试这两种情况，以使用输出等效性检查来辨别哪一种情况是正确的。

Witcher 使用输出等效性检查来验证违反推断的可能正确性条件的 NVM 状态是否确实不一致，表明存在崩溃一致性错误。许多 NVM 程序，包括持久键值存储，旨在操作粒度（例如，插入、删除）上提供持久的线性化。也就是说，在崩溃时，NVM 程序应该表现得好像发生崩溃的操作要么完全执行要么根本不执行（即，全部或不执行语义）。因此，Witcher 可以通过比较有和没有崩溃的执行输出来验证崩溃的一致性。如果从违反可能正确性条件的 NVM 状态恢复的程序产生与没有崩溃的执行不同的输出，那么我们可以自信地得出结论，该程序不是崩溃一致的。如果是这样，违反可能的正确性条件就是一个真正的错误。

Witcher 使用基于跟踪的方法来检测性能错误。与查找正确性错误需要搜索可能的崩溃 NVM 状态不同，检测性能错误不需要崩溃模拟，只需要按程序顺序跟踪 NVM 持久状态。为了检测额外的刷新性能错误，需要在执行刷新指令之前要刷新的缓存行的持久状态。 Witcher 利用收集到的动态程序跟踪并在 NVM 持久性模拟期间检测性能错误。

3.1.3 性能错误检测

Witcher 基于跟踪的缓存/NVM 模拟检测以下性能错误。如果在模拟结束时存储仍然保留在缓存中（未持久化），但它通过了输出等效性检查，Witcher 会报告一个未持久化的性能错误。在模拟刷新指令时，如果所有先前的存储都已被先前的刷新指令刷新，Witcher 会报告一个额外的刷新

性能错误。在模拟围栏指令时，如果没有前面的刷新指令，Witcher 会报告一个额外的围栏性能错误。对于事务性 NVM 程序，如果一个内存区域或其子集已经被同一事务中的先前日志操作记录了，Witcher 会报告一个额外的日志性能错误。

3.1.4 总结

目前的 Witcher，一个用于 NVM 支持的持久键值存储的系统崩溃一致性测试框架。Witcher 推断可能的正确性条件并执行输出等效性检查以验证其违规。这种方法允许 Witcher 使用与应用程序无关的规则来查找特定于应用程序的正确性错误，而无需手动注释、用户提供的一致性检查器或详尽的测试。 Witcher 还检测 NVM 状态模拟期间的性能错误。

3.2 指定和检查文件系统崩溃一致性模型

作者提出了一个用于开发崩溃一致性模型的正式框架，以及一个名为 FERRITE 的工具包，用于根据实际文件系统实现验证这些模型。他们为 ext4 开发了崩溃一致性模型，并使用 FERRITE 来演示 ext4 实现的非直观崩溃行为。为了向应用程序编写者展示崩溃一致性模型的实用性，他们使用模型来构建概念验证和综合工具的原型，以及用于崩溃安全应用程序的新库接口。作者使用在此框架中表达的正式崩溃一致性模型来开发自动化验证和综合工具。

崩溃一致性模型定义了崩溃后文件系统的允许状态。我们开发了两种规范风格：公理式和操作式。公理模型使用一组公理和排序关系以声明方式描述有效的崩溃行为，而操作模型是模拟文件系统行为的相关方面（例如崩溃、缓存等）的抽象机器。作者展示了两个示例文件系统的两种模型：seqfs，一个具有强崩溃一致性保证（类似于顺序一致性）的理想文件系统，和 ext4，一个具有弱一致性保证（类似于弱内存）的真实文件系统模型）

FERRITE，这是一套用于推理崩溃一致性模型的自动化工具。 FERRITE 由两个工具组成——一个显式枚举器和一个有界模型检查器——它们详尽地探索了石蕊测试的所有可能的崩毁行为。模型检查器根据公理规范象征性地执行试金石测试。这些工具一起使文件系统开发人员创建崩溃行为的高保真规范：模型检查器确保形式化允许在 litmus 测试中编码的代表性行为，并且枚举器确保 litmus 测试确实代表实际文件的行为系统实现。

崩溃一致性的公理规范仅在它们忠实地捕获文件系统实现的行为时才有用。先前使用内存模型的经验表明，很容易意外地编写一个约束不足（允许应该禁止的行为）或过度约束（禁止应该允许的行为）的规范。为了帮助文件系统开发人员在指定崩溃一致性模型时避免这些陷阱，FERRITE 包含一个模型检查器，它象征性地针对规范执行试金石测试，确保规范（禁止）允许测试中编码的行为。

正式的崩溃一致性规范不需要应用程序编写者手动确保其程序的崩溃安全，而是允许我们为实现综合足够的障碍，使其满足所需的崩溃安全属性。

最近对内存一致性模型的研究设计了编程抽象，在可能的情况下隐藏内存重新排序的细节（使用并发库），但在需要达到峰值性能时公开它们（例如通过 C11 原子）。除了为综合和验证工具提供基础之外，崩溃一致性模型还为提供崩溃一致性保证的编程抽象提出了类似的设计。作者提出了两个接口，第一个是一个简单的接口，用于向应用程序公开文件系统崩溃一致性行为的关键特征，类似于 C11 原子。第二个是一种激进的设计，它为应用程序提供连续的碰撞一致性 (SCC) 保证。我们使用我们的石蕊测试和 Dafny 验证框架对这两种设计进行了原型设计。

作者的工作受到过去的研究的启发，这些研究发现了应用程序和文件系统中的碰撞安全错误，探索了更好的碰撞安全编程抽象，并提供了文件系统的正式模型。

3.3 使用轻量级形式化方法验证键值存储节点

ShardStore 是目前部署在 Amazon S3 云对象存储服务中的键值存储。每个存储节点都存储客户对象的分片，这些分片在多个节点之间复制以实现持久性，因此存储节点不需要在内部复制其存储的数据。

3.3.1 设计概述

ShardStore 为 S3 系统的其余部分提供了一个键值存储接口，其中键是分片标识符，值是客户对象数据的分片。客户请求由 S3 的元数据子系统映射到分片标识符。ShardStore 的键值存储包含一个日志结构合并树（LSM 树），但分片数据存储在树外以减少写入放大，类似于 WiscKey。LSM 树将每个分片标识符映射到一个（指向）块的列表，每个块都存储在一个范围内。盘区是磁盘上物理存储的连续区域；一个典型的磁盘有数万个盘区。ShardStore 要求每个区段内的写入是顺序的，由定义下一个有效写入位置的写指针跟踪，因此不能立即覆盖区段上的数据。每个 extent 都有一个 reset 操作，将写指针返回到 extent 的开头并允许覆盖。ShardStore 的磁盘布局。每个范围都提供仅附加写入。日志结构合并树（LSM 树）将分片映射到它们的数据，以块的形式存储在范围上。 LSM 树本身也作为块存储在磁盘上。ShardStore 不是将所有分片数据集中在磁盘上的单个共享日志中，而是将分片数据分布在不同的范围内。这种方法让我们可以灵活地将每个分片的数据放在磁盘上，以优化预期的热量和访问模式（例如，最大限度地减少寻道延迟）。

3.3.2 崩溃一致性

ShardStore 使用了一种受软更新启发的崩溃一致性方法。软更新实现协调将写入发送到磁盘的顺序，以确保磁盘的任何崩溃状态都是一致的。软更新避免了通过预写日志重定向写入的成本，并允许灵活地将数据物理放置在磁盘上。正确实现软更新需要对所有可能的磁盘写回顺序进行全局推理。为了降低这种复杂性，ShardStore 的实现以声明方式指定崩溃一致的排序，使用 Dependency 类型在运行时构造依赖图，以指示有效的写入顺序。ShardStore 的 extent append 操作，是写入磁盘的唯一方式。Amazon S3 旨在实现的数据持久性，并跨多个存储节点复制对象数据，因此单节点崩溃一致性问题不会导致数据丢失。相反，我们将崩溃一致性视为降低存储节点故障的成本和运营影响。从丢失整个存储节点数据的崩溃中恢复会在整个存储节点队列中产生大量修复网络流量和 IO 负载。崩溃一致性还确保存储节点在崩溃后恢复到安全状态，因此不会出现可能需要操作员手动干预的意外行为。

3.3.3 验证存储系统

生产存储系统结合了几个难以实现的复杂性：复杂的磁盘数据结构、并发访问和对它们的更改，以及在崩溃时保持一致性的需要。实际实现的规模反映了这种复杂性：ShardStore 超过 40,000 行代码并且经常更改。在 ShardStore 设计过程的早期面临这些挑战，我们从最近在存储系统验证中取得的成功中汲取灵感，并决定应用形式化方法来增加我们的信心。我们选择正式方法是因为它们允许我们验证 ShardStore 实现的深层属性，这些属性很难用 S3 的规模和复杂性的现成工具进行测试 - API 级调用的功能正确性，磁盘数据结构的崩溃一致性以及并发执行的正确性，包括 API 调用和垃圾收集等维护任务。鉴于系统的复杂性和快速变化的速度，我们需要我们的结果比形式方法专家的参与更持久，因此我们寻求一种可以由工程团队自己开发和自动化的轻量级方法。

3.3.4 一致性检查

一旦我们有了指定系统预期行为的参考模型，我们就需要根据上述正确性属性检查实现是否符合参考模型。本节详细介绍

了我们如何使用基于属性的测试来实现此检查，以及我们如何确保对实现的可能行为的良好覆盖。

(1) 基于属性的测试

为了检查实现是否满足参考模型规范，我们使用基于属性的测试，它生成测试用例的输入，并在测试运行时检查用户提供的属性是否成立。基于属性的测试可以被认为是对用户提供的正确性属性和结构化输入进行模糊测试的扩展，这使得它可以检查比单独模糊测试更丰富的行为。当基于属性的测试失败时，生成的输入允许重放失败（假设测试是确定性的）。我们使用基于属性的测试来检查实现代码是否完善了参考模型：模型必须允许实现的任何可观察行为。我们将此检查构建为基于属性的测试，它将从我们定义的字母表中提取的一系列操作作为输入。对于序列中的每个操作，测试用例将操作应用于参考模型和实现，比较每个输出的等价性，然后检查与两个系统相关的不变量。为确保确定性和测试性能，被测实现使用内存用户空间磁盘，但磁盘层以上的所有组件都使用其实际实现代码。

（2）覆盖率

基于属性的测试选择随机的操作序列进行测试，因此可能会遗漏错误。我们通过增加测试规模减少这种风险。我们经常在每次 ShardStore 部署之前运行数千万个随机测试序列，但这些测试只能检查测试工具能够达到的系统状态。使用基于属性的测试进行验证的关键挑战是确保它可以达到系统的有趣状态。为此，我们通过偏置将领域知识引入参数选择中，并使用代码覆盖机制来监控测试有效性。

3.3.5 检查崩溃一致性

崩溃一致性可能是存储系统中错误的来源。 ShardStore 使用基于软更新的崩溃一致性协议，这引入了实现复杂性。对于ShardStore，关于崩溃一致性的推理是在开发过程中引入正式方法的主要动机，因此它是我们努力的重点。我们通过使用用户空间依赖项来验证崩溃一致性。我们根据这些依赖项定义了两个崩溃一致性属性：1. 持久性：如果依赖项说一个操作在崩溃前已经存在，那么它在崩溃后应该是可读的（除非被后来的持久化操作取代） 2.前进进度：非崩溃关机后，每个操作的依赖都应该表明它是持久的。

3.3.6 检查并发执行

到目前为止，我们的验证方法仅处理顺序正确性，因为之前的一致性检查方法仅测试确定性单线程执行。在实践中，像 ShardStore 这样的生产存储系统是高度并发的，每个磁盘都服务于多个并发请求和后台维护任务（例如，LSM 树压缩、缓冲区缓存刷新等）。 Rust 的类型系统保证了语言安全片段内的数据竞争自由，但不能保证更高级别的竞争条件（例如，原子性违规），它们是难以测试和调试，因为它们在执行中引入了不确定性。

3.3.7 相关工作

最近人们对经过正式验证的存储系统产生了浓厚的兴趣。 Yggdrasil 是一个经过“按钮验证”的文件系统实现，它形式化了一种改进，以定义崩溃后的允许状态。FSCQ 是一个正式验证的文件系统，它采用崩溃感知 Hoare 逻辑来定义每个 POSIX 系统调用的允许崩溃行为。 VeriBetrKV 是最近的一项验证键值存储的努力，重点是自动证明，它提供比我们更多的开销（用他们的术语来说是“tediumž”）更强大的保证：7 行证明每一条执行线。我们的方法没有这些示例的健全性保证——我们可以错过这些系统可能会发现的错误——但我们的轻量级工具极大地提高了我们的工程团队将形式化方法采用和集成到他们的开发实践中的能力。

我们认为处理并发对于我们的方法取得成功至关重要。最近对并发存储系统的验证工作集中在简单的复制磁盘和日志文件系统。ShardStore 比这些设计复杂得多，因此我们专注于轻量级无状态模型检查，并使用 Shuttle 模型检查器为并发测试失

败提供良好的调试体验，同样以可能丢失错误为代价。

4 总结与展望

键值存储的正确性验证是非常重要的，同时也是非常复杂的。键值存储的崩溃一致性得到了保障，在此基础上的应用和系统就有可能在崩毁后恢复到一致的状态。

在测试 NVM 崩毁一致性时，如果列举所有的 NVM 状态，则可能会导致测试空间爆炸，这种穷举的方式不切合实际。崩毁一致性测试框架 Witcher 推断可能正确的条件，剪去一定的测试空间，根据这些可能正确性条件去发现 NVM 程序的正确性错误和性能错误。Amazon S3 在验证新的存储结点正确性时，没有采用完全形式化验证，因为这样会耗费大量的时间和人力，而是使用轻量级的形式化方法，不需要专业的形式化专家团队，由工程师团队负责编写和更新验证结点正确性的代码。虽然保障结点正确性的能力不如完全形式化验证，但这种方法可用性强，自动化测试，而且随着软件的更新，这些测试的规范也会随之更新，在未来的开发当中都能够起作用。这些验证键值存储的方法，都在避开正确性的完全验证，因为其代价太高了，而是在一定的筛选条件下，验证键值存储的一部分正确性，且都取得了显著的效果。Amazon S3 工程师目前没有对资源耗尽故障（例如磁盘空间不足）应用基于属性的测试，这些场景很难自动测试，资源耗尽测试需要区分真正的错误（例如，空间泄漏）和预期的失败，因为测试分配的空间比环境可用的空间多。然而，所有存储系统都会引入一定量的空间开销和放大（由于元数据、校验和、块对齐、分配等）。区分真正的故障需要考虑这些开销，但它们是实现所固有的，难以抽象计算，采用资源分析中的想法将是解决这个问题的有希望的未来工作。

参考文献

[1] Fu X , Kim W H , Shreepathi A P , et al. WITCHER : Detecting Crash Consistency Bugs in Non-volatile Memory Programs[J]. 2020.

[2] Bornholt J , Kaufmann A , Li J , et al. Specifying and Checking File System Crash-Consistency Models[C]// the Twenty-First International Conference. 2016.

[3] Bornholt J, Joshi R, Astrauskas V, et al. Using lightweight formal methods to validate a key-value storage node in Amazon S3[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 836-850

[4] Tan C, Zhao C, Mu S, et al. Cobra: Making transactional key-value stores verifiably serializable[C]//14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20). 2020: 63-80.

[5] Hance T, Lattuada A, Hawblitzel C, et al. Storage Systems are Distributed Systems (So Verify Them That Way!)[C]//14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20). 2020: 99-115.