Testing Distributed Systems for Linearizability

测试分布式系统的线性化

4 Jun 2017 — shared on Lobsters, Reddit, and Twitter

2017 年 6 月 4 日 — 在 Lobsters、Reddit 和 Twitter 上分享

Distributed systems are challenging to implement correctly because they must handle concurrency and failure. Networks can delay, duplicate, reorder, and drop packets, and machines can fail at any time. Even when designs are proven correct on paper, it is difficult to avoid subtle bugs in implementations.

分布式系统很难正确实现，因为它们必须处理并发和故障。网络可能会延迟、复制、重新排序和丢弃数据包，并且机器可能随时出现故障。即使设计在纸面上被证明是正确的，也很难避免实现中的细微错误。

Unless we want to use formal methods[1], we have to test systems if we want assurance that implementations are correct. Testing distributed systems is challenging, too. Concurrency and nondeterminism make it difficult to catch bugs in tests, especially when the most subtle bugs surface only under scenarios that are uncommon in regular operation, such as simultaneous machine failure or extreme network delays.

除非我们想使用正式方法 [1] ，否则如果我们想确保实现正确，就必须测试系统。测试分布式系统也具有挑战性。并发性和不确定性使得在测试中捕获错误变得困难，特别是当最微妙的错误仅在常规操作中不常见的情况下才会出现时，例如同时机器故障或极端网络延迟。

Correctness 正确性

Before we can discuss testing distributed systems for correctness, we need to define what we mean by “correct”. Even for seemingly simple systems, specifying exactly how the system is supposed to behave is an involved process[2].

在我们讨论测试分布式系统的正确性之前，我们需要定义“正确”的含义。即使对于看似简单的系统，准确指定系统的行为方式也是一个复杂的过程 [2] 。

Consider a simple key-value store, similar to etcd, that maps strings to strings and supports two operations: `Put(key, value)` and `Get(key)`. First, we consider how it behaves in the sequential case.

考虑一个简单的键值存储，类似于 etcd，它将字符串映射到字符串并支持两种操作： `Put(key, value)` 和 `Get(key)` 。首先，我们考虑它在顺序情况下的表现。

Sequential Specifications

顺序规格

We probably have a good intuitive understanding of how a key-value store is supposed to behave under sequential operation: `Get` operations must reflect the result of applying all previous `Put` operations. For example, we could run a `Put("x", "y")` and then a subsequent `Get("x")` should return `"y"`. If the operation returned, say, a `"z"`, that would be incorrect.

我们可能对键值存储在顺序操作下的行为有一个很好的直观理解： `Get` 操作必须反映应用所有先前 `Put` 操作的结果。例如，我们可以运行 `Put("x", "y")` ，然后后续的 `Get("x")` 应返回 `"y"` 。如果操作返回，例如 `"z"` ，那将是不正确的。

More formal than an English-language description, we can write a specification for our key-value store as executable code:

比英语描述更正式，我们可以为我们的键值存储编写一个规范作为可执行代码：

```
class KVStore:
    def __init__(self):
        self._data = {}

    def put(self, key, value):
        self._data[key] = value

    def get(self, key):
        return self._data.get(key, "")
```

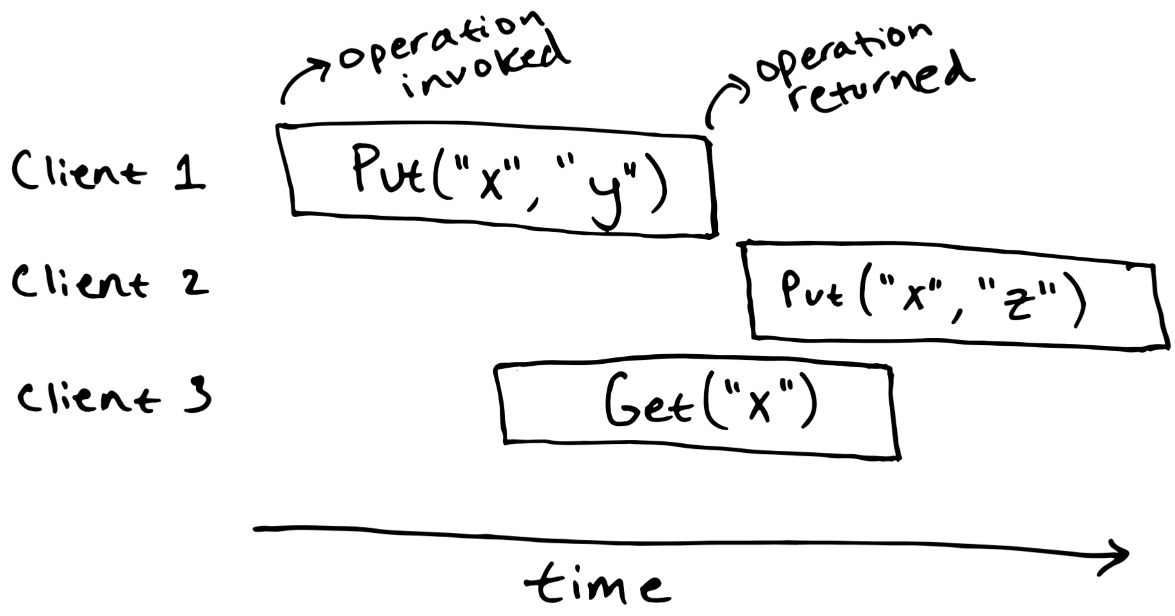
The code is short, but it nails down all the important details: the start state, how the internal state is modified as a result of operations, and what values are returned as a result of calls on the key-value store. The spec solidifies some details like what happens when `Get()` is called on a nonexistent key, but in general, it lines up with our intuitive definition of a key-value store.

代码很短，但它明确了所有重要的细节：开始状态、内部状态如何作为操作的结果进行修改，以及作为键值存储调用的结果返回什么值。该规范固化了一些细节，例如在不存在的键上调用 `Get()` 时会发生什么，但总的来说，它与我们对键值存储的直观定义一致。

Linearizability 线性度

Next, we consider how our key-value store can behave under concurrent operation. Note that the sequential specification does **not** tell us what happens under concurrent operation. For example, the sequential spec doesn't say how our key-value store is allowed to behave in this scenario:

接下来，我们考虑我们的键值存储在并发操作下如何表现。请注意，顺序规范并没有告诉我们并发操作下会发生什么。例如，顺序规范没有说明我们的键值存储在这种情况下如何表现：



It's not immediately obvious what value the `Get("x")` operation should be allowed to return. Intuitively, we might say that because the `Get("x")` is concurrent with the `Put("x", "y")` and `Put("x", "z")`, it can return either value or even `""`. If we had a situation where another client executed a `Get("x")` much later, we might say that the operation must return `"z"`, because that was the value written by the last write, and the last write operation was not concurrent with any other writes.

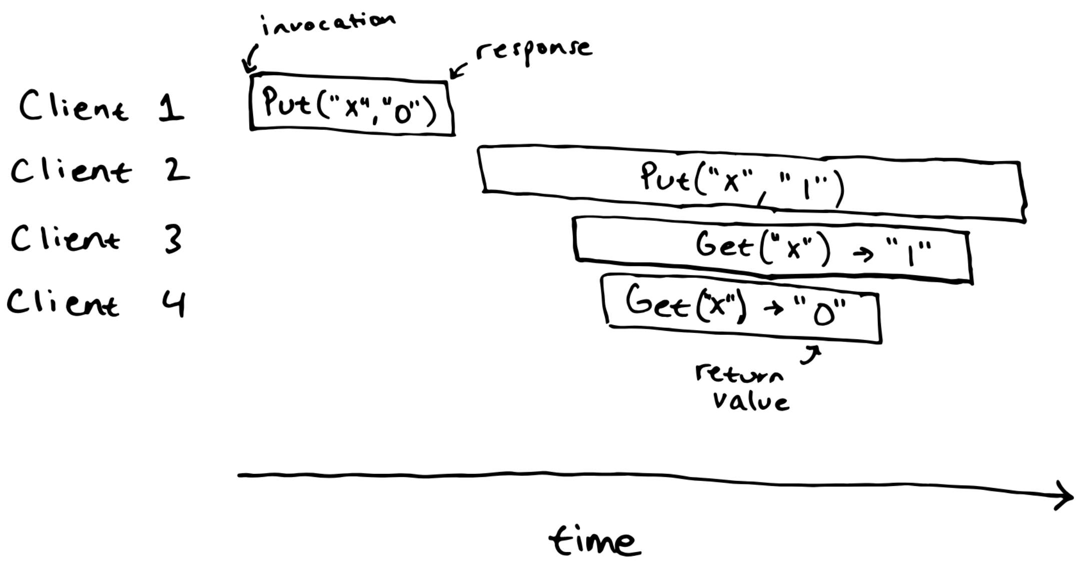
应该允许 `Get("x")` 操作返回什么值并不是很明显。直观上，我们可能会说，因为 `Get("x")` 与 `Put("x", "y")` 和 `Put("x", "z")` 并发，所以它可以返回值，甚至可以返回 `""` 。如果我们遇到另一个客户端很晚才执行 `Get("x")` 的情况，我们可能会说该操作必须返回 `"z"` ，因为这是最后一次写入写入的值，也是最后一次写入的值写入操作不与任何其他写入同时进行。

We formally specify correctness for concurrent operations based on a sequential specification using a consistency model known as **linearizability**. In a linearizable system, **every operation appears to execute atomically and instantaneously at some point between the invocation and response**. There are other consistency models besides linearizability, but many distributed systems provide linearizable behavior: linearizability is a strong consistency model, so it's relatively easy to build other systems on top of linearizable systems.

我们使用称为线性化的一致性模型，基于顺序规范正式指定并发操作的正确性。在线性化系统中，每个操作似乎都在调用和响应之间的某个时刻以原子方式即时执行。除了线性化之外，还有其他一致性模型，但许多分布式系统都提供线性化行为：线性化是一种强一致性模型，因此在线性化系统之上构建其他系统相对容易。

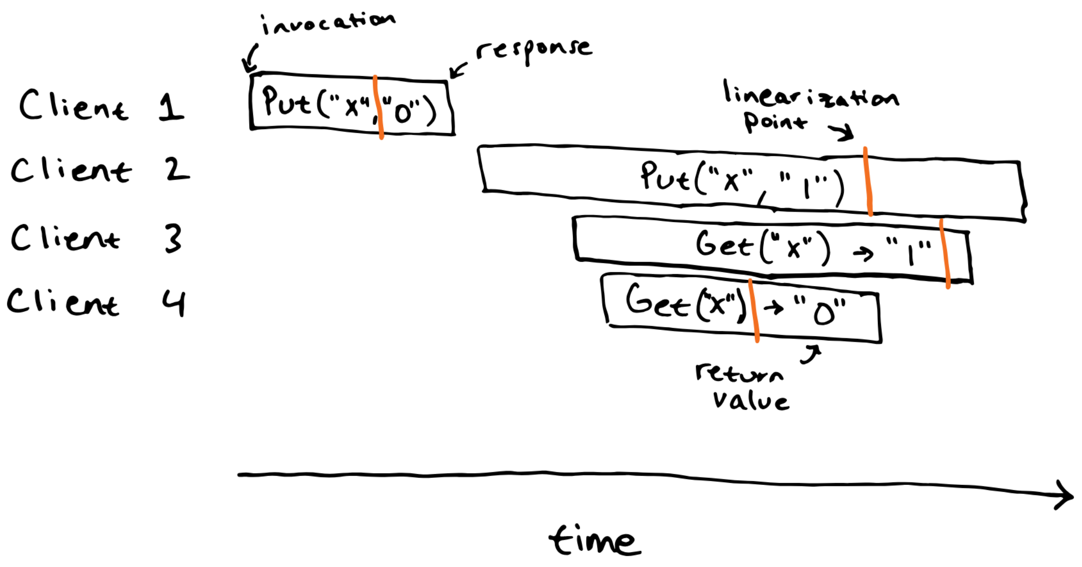
Consider an example history with invocations and return values of operations on a key-value store:

考虑一个示例历史记录，其中包含键值存储上的操作的调用和返回值：



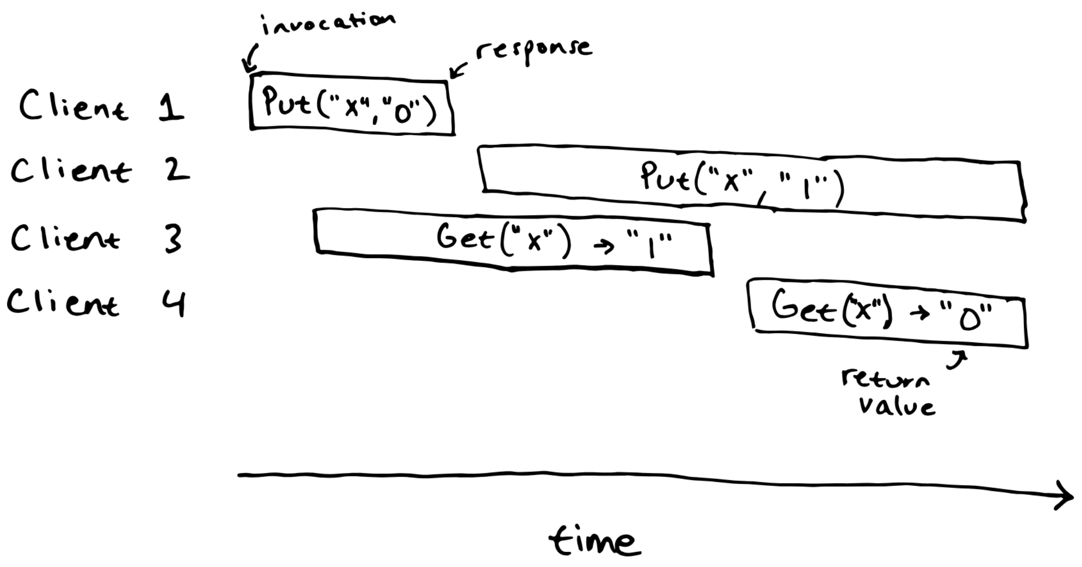
This history **is linearizable**. We can show this by explicitly finding linearization points for all operations (drawn in orange below). The induced sequential history, `Put("x", "0")`, `Get("x") -> "0"`, `Put("x", "1")`, `Get("x") -> "1"`, is a correct history with respect to the sequential specification.

这段历史是线性化的。我们可以通过显式查找所有操作的线性化点（下面用橙色绘制）来展示这一点。诱导的顺序历史 `Put("x", "0")` 、 `Get("x") -> "0"` 、 `Put("x", "1")` 、 `Get("x") -> "1"` 是关于顺序规范的正确历史。



In contrast, this history is **not** linearizable:

相反，这段历史是不可线性化的：



There is no linearization of this history with respect to the sequential specification: there is no way to assign linearization points to operations in this history. We could start assigning linearization points to the operations from clients 1, 2, and 3, but then there would be no way to assign a linearization point for client 4: it would be observing a stale value. Similarly, we could start assigning linearization points to the operations from clients 1, 2, and 4, but then the linearization point of client 2’s operation would be after the start of client 4’s operation, and then we wouldn’t be able to assign a linearization point for client 3: it could legally only read a value of `""` or `"0"` .

相对于顺序规范，该历史记录没有线性化：无法将线性化点分配给该历史记录中的操作。我们可以开始为客户端 1、2 和 3 的操作分配线性化点，但随后就无法为客户端 4 分配线性化点：它将观察到过时的值。类似地，我们可以开始为客户端 1、2 和 4 的操作分配线性化点，但是客户端 2 的操作的线性化点将在客户端 4 的操作开始之后，然后我们将无法分配线性化点。客户端 3 的线性化点：它只能合法读取 `""` 或 `"0"` 的值。

Testing 测试

With a solid definition of correctness, we can think about how to test distributed systems. The general approach is to test for correct operation while randomly injecting faults such as machine failures and network partitions. We could even simulate the entire network so it’s possible to do things like cause extremely long network delays. Because tests are randomized, we would want to run them a bunch of times to gain assurance that a system implementation is correct.

有了正确性的可靠定义，我们就可以考虑如何测试分布式系统了。一般的方法是测试是否正确运行，同时随机注入机器故障和网络分区等故障。我们甚至可以模拟整个网络，这样就可以做一些事情，比如造成极长的网络延迟。由于测试是随机的，因此我们希望多次运行它们以确保系统实现是正确的。

Ad-hoc testing 临时测试

How do we actually test for correct operation? With the simplest software, we test it using input-output cases like `assert(expected_output == f(input))` . We could use a similar approach with distributed systems. For example, with our key-value store, we could have the following test where multiple clients are executing operations on the key-value store in parallel:

我们如何实际测试操作是否正确？使用最简单的软件，我们使用像 `assert(expected_output == f(input))` 这样的输入输出情况来测试它。我们可以对分布式系统使用类似的方法。例如，对于我们的键值存储，我们可以进行以下测试，其中多个客户端在键值存储上并行执行操作：

```
for client_id = 0..10 {
    spawn thread {
        for i = 0..1000 {
            value = rand()
            kvstore.put(client_id, value)
            assert(kvstore.get(client_id) == value)
        }
    }
}
wait for threads
```

It is certainly the case that if the above test fails, then the key-value store is not linearizable. However, this test is not that thorough: there are non-linearizable key-value stores that would always pass this test.

当然，如果上述测试失败，则键值存储不可线性化。然而，这个测试并不是那么彻底：有一些非线性的键值存储总是可以通过这个测试。

Linearizability 线性度

A better test would be to have parallel clients run **completely random operations**: e.g. repeatedly calling `kvstore.put(rand(), rand())` and `kvstore.get(rand())` , perhaps limited to a small set of keys to increase contention. But in this case, how would we determine what is “correct” operation? With the simpler test, we had each client operating on a separate key, so we could always predict exactly what the output had to be.

更好的测试是让并行客户端运行完全随机的操作：例如重复调用 `kvstore.put(rand(), rand())` 和 `kvstore.get(rand())` ，可能仅限于一小组键以增加争用。但在这种情况下，我们如何判断什么是“正确”的操作呢？通过更简单的测试，我们让每个客户端在单独的密钥上运行，因此我们始终可以准确预测输出必须是什么。

When clients are operating concurrently on the same set of keys, things get more complicated: we can’t predict what the output of every operation has to be because there isn’t only one right answer. So we have to take an alternative approach: we can test for correctness by recording an entire history of operations on the system and then checking if the history is linearizable with respect to the sequential specification.

当客户端同时对同一组键进行操作时，事情会变得更加复杂：我们无法预测每个操作的输出是什么，因为不只有一个正确答案。因此，我们必须采取另一种方法：我们可以通过记录系统上操作的整个历史记录来测试正确性，然后检查历史记录是否相对于顺序规范可线性化。

Linearizability Checking 线性度检查

A linearizability checker takes as input a sequential specification and a concurrent history, and it runs a decision procedure to check whether the history is linearizable with respect to the spec.

线性化检查器将顺序规范和并发历史记录作为输入，并运行决策过程来检查历史记录是否相对于规范可线性化。

NP-Completeness NP完备性

Unfortunately, linearizability checking is NP-complete. The proof is actually quite simple: we can show that linearizability checking is in NP, and we can show that an NP-hard problem can be reduced to linearizability checking. Clearly, linearizability checking is in NP: given a linearization, i.e. the linearization points of all operations, we can check in polynomial time if it is a valid linearization with respect to the sequential spec.

不幸的是，线性化检查是 NP 完全的。证明实际上非常简单：我们可以证明线性化检查是在 NP 中进行的，并且我们可以证明 NP 难问题可以简化为线性化检查。显然，线性化检查是 NP 形式的：给定线性化，即所有操作的线性化点，我们可以在多项式时间内检查它是否是相对于顺序规范的有效线性化。

To show that linearizability checking is NP-hard, we can reduce the subset sum problem to linearizability checking. Recall that in the subset sum problem, we are given a set $S = \{s_1, s_2, \ldots, s_n\}$ of non-negative integers and a target value t, and we have to determine whether there exists a subset of S that sums to t. We can reduce this problem to linearizability checking as follows. Consider the sequential spec:

为了证明线性化检查是 NP 困难的，我们可以将子集和问题简化为线性化检查。回想一下，在子集和问题中，我们给出了一组非负整数 $S = \{s_1, s_2, \ldots, s_n\}$ 和一个目标值 t ，我们必须确定是否存在 S 总和为 t 。我们可以将这个问题简化为线性化检查，如下所示。考虑顺序规范：

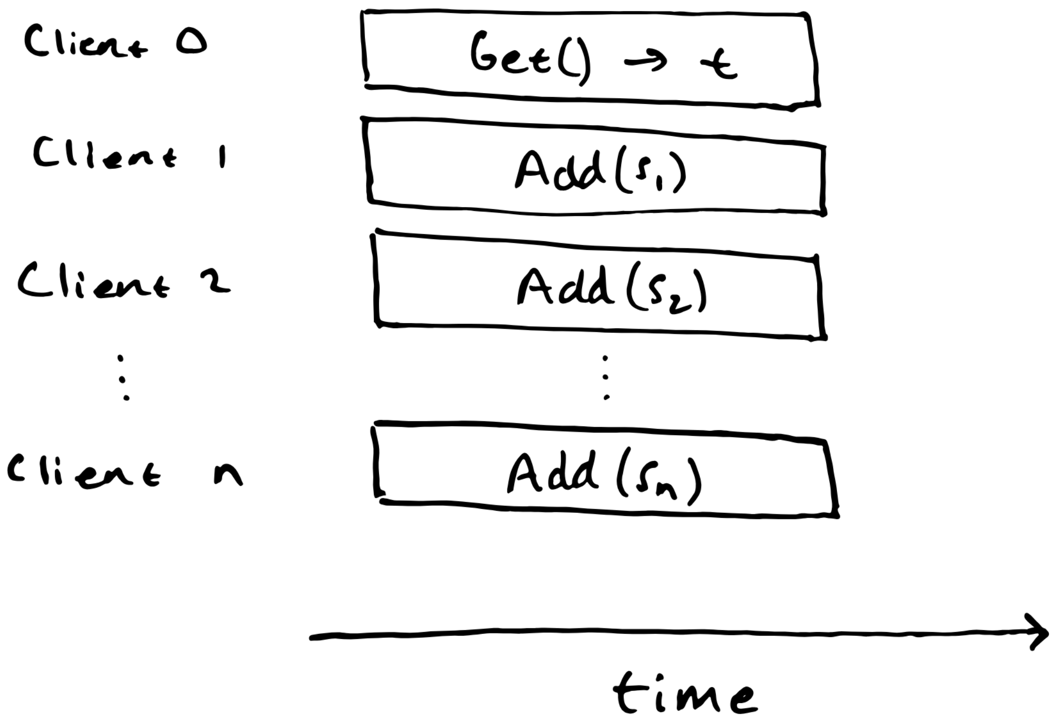
```
class Adder:
    def __init__(self):
        self._total = 0

    def add(self, value):
        self._total += value

    def get(self):
        return self._total
```

And consider this history:

并考虑这段历史：



This history is linearizable if and only if the answer to the subset sum problem is “yes”. If the history is linearizable, then we can take all the operations `Add(s_i)` that have linearization points before that of the `Get()` operation, and those correspond to elements s_i in a subset whose sum is t. If the set does have a subset that sums to t, then we can construct a linearization by having the operations `Add(s_i)` corresponding to the elements s_i in the subset take place before the `Get()` operation and having the rest of the operations take place after the `Get()` operation.

当且仅当子集和问题的答案为“是”时，该历史才可线性化。如果历史是可线性化的，那么我们可以采用在 `Get()` 操作之前具有线性化点的所有操作 `Add(s_i)` ，并且这些操作对应于中的元素 s_i 总和为 t 的子集。如果该集合确实有一个总和为 t 的子集，那么我们可以通过与子集中的元素 s_i 对应的操作 `Add(s_i)` 来构造线性化放置在 `Get()` 操作之前，并使其余操作发生在 `Get()` 操作之后。

Implementation 执行

Even though linearizability checking is NP-complete, in practice, it can work pretty well on small histories. Implementations of linearizability checkers take an executable specification along with a history, and they run a search procedure to try to construct a linearization, using tricks to constrain the size of the search space.

尽管线性化检查是 NP 完全的，但实际上，它在小历史上可以很好地工作。线性化检查器的实现采用可执行规范和历史记录，并运行搜索过程来尝试构建线性化，使用技巧来限制搜索空间的大小。

There are existing linearizability checkers like Knossos, which is used in the Jepsen test system. Unfortunately, when trying to test an implementation of a distributed key-value store that I had written, I couldn't get Knossos to check my histories. It seemed to work okay on histories with a couple concurrent clients, with about a hundred history events in total, but in my tests, I had tens of clients generating histories of thousands of events.

现有的线性化检查器如 Knossos，用于 Jepsen 测试系统。不幸的是，当尝试测试我编写的分布式键值存储的实现时，我无法让 Knossos 检查我的历史记录。它似乎在有几个并发客户端的历史上工作得很好，总共有大约一百个历史事件，但在我的测试中，我有数十个客户端生成了数千个事件的历史。

To be able to test my key-value store, I wrote Porcupine, a fast linearizability checker implemented in Go. Porcupine checks if histories are linearizable with respect to executable specifications written in Go. Empirically, Porcupine is thousands of times faster than Knossos. I was able to use it to test my key-value store because it is capable of checking histories of thousands of events in a couple seconds.

为了能够测试我的键值存储，我编写了 Porcupine，一个用 Go 实现的快速线性化检查器。Porcupine 检查历史记录是否可相对于用 Go 编写的可执行规范进行线性化。根据经验，Porcupine 的速度比 Knossos 快数千倍。我能够使用它来测试我的键值存储，因为它能够在几秒钟内检查数千个事件的历史记录。

Effectiveness 效力

Testing linearizable distributed systems using fault injection along with linearizability checking is an effective approach.

使用故障注入和线性化检查来测试可线性化分布式系统是一种有效的方法。

To compare ad-hoc testing with linearizability checking using Porcupine, I tried testing my distributed key-value store using the two approaches. I tried introducing different kinds of design bugs into the implementation of the key-value store, such as modifications that would result in stale reads, and I checked to see which tests failed. The ad-hoc tests caught some of the most egregious bugs, but the tests were incapable of catching the more subtle bugs. In contrast, I couldn't introduce a single correctness bug that the linearizability test couldn't catch.

为了将临时测试与使用 Porcupine 的线性化检查进行比较，我尝试使用这两种方法来测试我的分布式键值存储。我尝试在键值存储的实现中引入不同类型的设计错误，例如会导致过时读取的修改，并且我检查了哪些测试失败了。临时测试发现了一些最严重的错误，但测试无法发现更微妙的错误。相比之下，我无法引入线性化测试无法捕获的任何正确性错误。

---

1. Formal methods can provide strong guarantees about the correctness of distributed systems. For example, the UW PLSE research group has recently verified an

implementation of the Raft consensus protocol using the Coq proof assistant. Unfortunately, verification requires specialized knowledge, and verifying realistic systems involves huge effort. Perhaps one day systems used in the real world will be proven correct, but for now, production systems are tested but not verified. ↩️

形式化方法可以为分布式系统的正确性提供强有力的保证。例如，UW PLSE 研究小组最近使用 Coq 证明助手验证了 Raft 共识协议的实现。不幸的是，验证需要专业知识，并且验证现实系统需要付出巨大的努力。也许有一天，现实世界中使用的系统将被证明是正确的，但目前，生产系统已经过测试，但尚未得到验证。 ↩️

2. Ideally, all production systems would have formal specifications. Some systems that are being used in the real world today do have formal specs: for example, Raft has a formal spec written in TLA+. But unfortunately, the majority of real-world systems do not have formal specs. ↩️

   理想情况下，所有生产系统都应该有正式的规范。如今现实世界中使用的一些系统确实有正式的规范：例如，Raft 有一个用 TLA+ 编写的正式规范。但不幸的是，大多数现实世界的系统没有正式的规格。 ↩️