小米开源分布式KV存储系统Pegasus

原创 2017-10-19 孙伟杰 InfoQ

作者 | 孙伟杰

编辑 | 小智

小米近日开源了分布式 KV 存储系统 Pegasus，这个小米自造的轮子背后，有着什么样的设计理念与技术细节?

写在前面

这次我给大家带来的主题是“分布式实现那些事儿——Pegasus 背后的故事”。在这次演讲中，我着重讲解的是我们在 Pegasus 的架构和实现之中遇到的一些的坑。

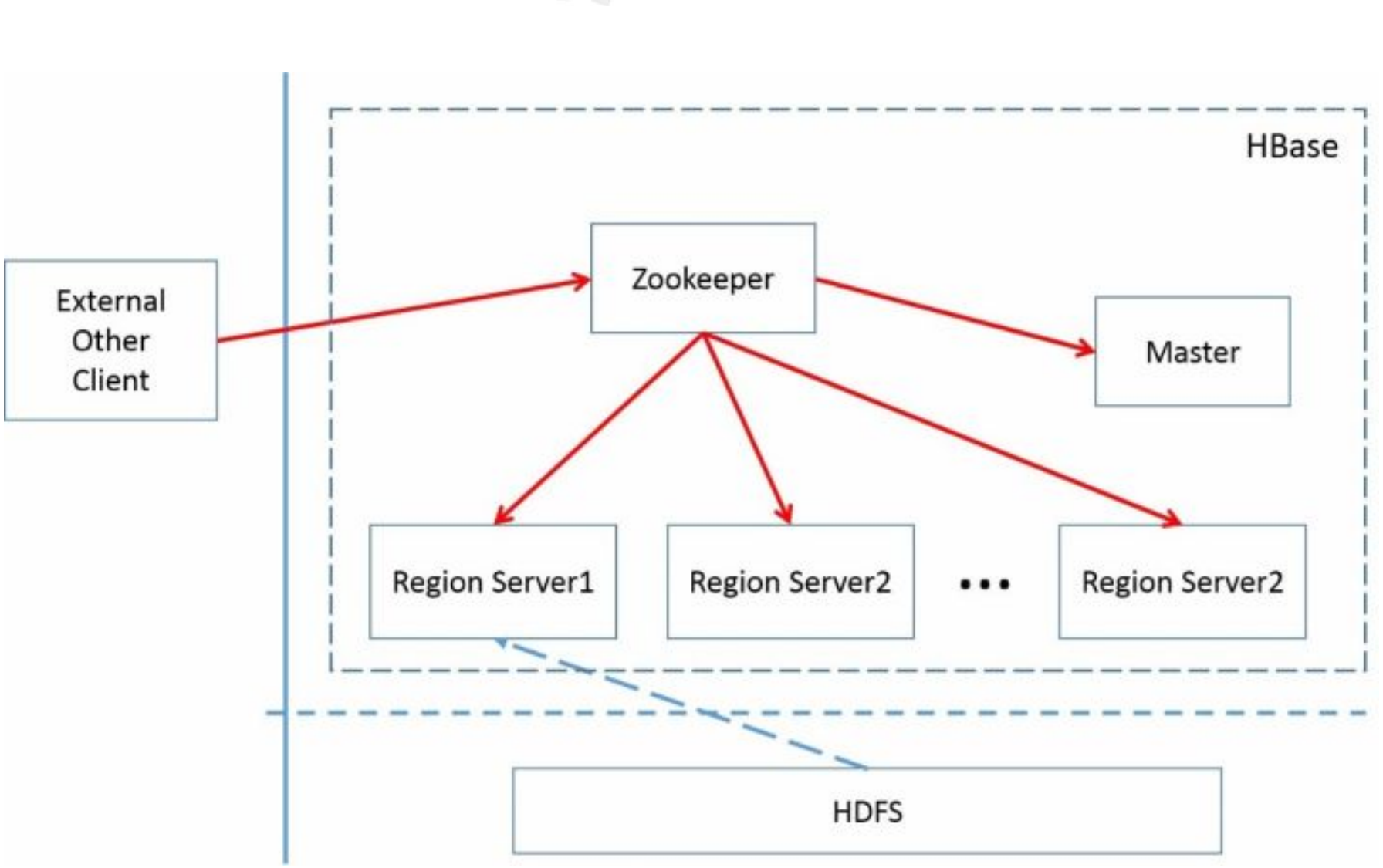
我来自小米，主要做 Pegasus 的研发工作。Pegasus 是小米自己造的一个分布式 KV 存储的轮子。对于造轮子这件事情，不知道各位是什么样的想法，就我个人而言，造轮子是要竭力避免的。如果有一个较好的方案能解决问题，则应该优先使用别的方案；除非随着业务的发展，没什么好的方案可供选择了，那我们只能自己造一个出来了。

做 Pegasus 的过程就是这个样子的。

Pegasus 的产生

在 Pegasus 立项之前，小米的分布式存储主要使用 HBase。经过 10 多年的发展，HBase 本身还是比较稳定的，并且功能接口上也较为方便。但由于设计和实现上的一些问题，我们在使用上还是会有一些坑。

记一次 HBase 事故



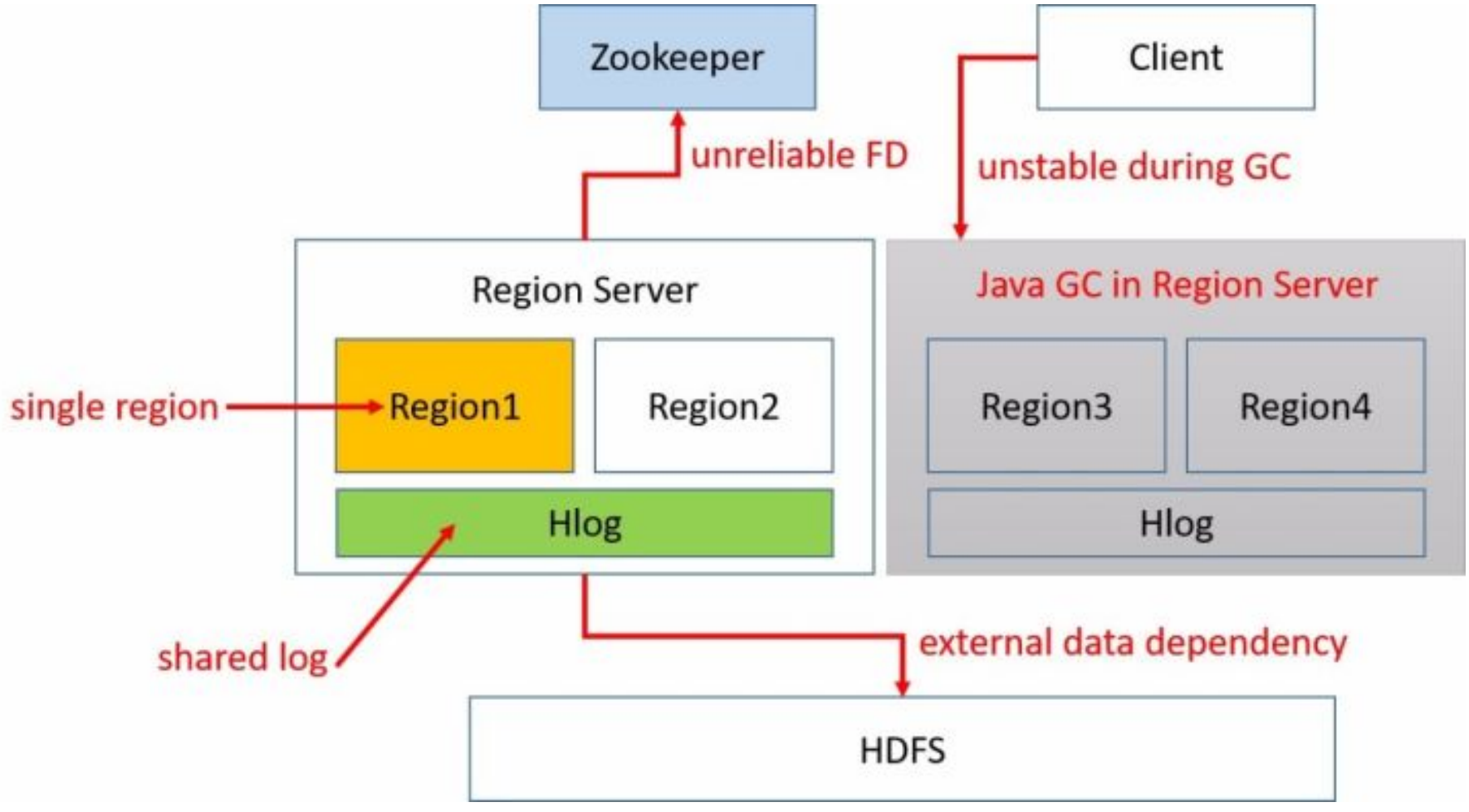
HBase 的架构，我相信大家应该对其也比较熟悉了。如上图所示，HBase 由一个 Master 节点来管理整个集群的状态。在 Master 节点的统筹管理之下，Region Server 节点负责客户端

的读写请求；Region Server 的数据，存在一个外部的分布式文件系统 HDFS 之上。Region Server 的心跳探活，以及 Master 的高可用都由 Zookeeper 来负责。

我们在实际使用时，很多问题都由 Zookeeper 而起。Zookeeper 在设计之初，它是一个要求负载比较低的系，所以在小米这边，Zookeeper 大多都由多个业务混合使用。除了 HBase 之外，很多其他业务的节点探活，服务注册等都依赖 Zookeeper。但一些业务在使用 Zookeeper 时，会将超额的压力加到 Zookeeper 上，从而导致 Zookeeper 的不稳定。除此之外，一些对 Zookeeper 的误用，如访问时不共用 socket 连接，也是 Zookeeper 服务不稳定的来源。

各种误用因素的存在，会导致 Zookeeper 经常遇到崩溃。对 HBase 而言，Zookeeper 的崩溃就意味着 Region Server 也随之崩溃掉。而小米的很多业务都依赖 HBase，一旦 HBase 崩掉，小米的很多业务也随之无法提供服务了。

HBase 的不足



通过对 HBase 的一系列问题进行复盘，我们认为有几点值得提一下：

1、对于 HBase 而言，它将“节点探活”这一重要的任务交给 Zookeeper 来做，是可以商榷的。因为如果运维不够细致的话，会使得 Zookeeper 成为影响 HBase 稳定性的一个坑。

在 HBase 中，Region Server 对“Zookeeper 会话超时”的处理方式是“自杀”。而 Region Server 上“多个 Region 合写一个 WAL 到 HDFS”的实现方式会使得“自杀”这一行为的成本比较高，因为自杀之后 Server 重启时会拆分和重放 WAL。这就意味着假如整个 HBase 集群挂了，想要将 HBase 重新给拉起来，时间会比较长。

2、即使我们能保证 Zookeeper 的稳定性，“节点探活”这一功能也不能非常稳定的运行。因为 HBase 是用 Java 实现的。GC 的存在，会使得 Zookeeper 把正常运行的 Region Server 误判为死亡，进而又会引发 Region Server 的自杀；在其之上的 Region，需要其他的 Server 从 HDFS 上加载重放 WAL 才能提供服务。而这一过程，同样也是比较耗时的。在此期间内，Region 所服务的 Key 都是不可读写的。

对于这一问题，可以通过将“节点探活”的时间阈值拉长来解决。但这会使得真正的“Region Server 死亡”不能被及时发现，从而另一个方面引发可用性的问题。

3、GC 的另一个问题是，HBase 在读写延时上存在毛刺。我们希望在广告、推荐这种业务上能够尽量避免出现这种毛刺，即能够有一个比较稳定的延时。

把上面的三点概括一下，我们认为 HBase 在可用性和性能延时方面还是存在一些瑕疵的。对于这些问题，通过修修补补、调整参数的方式能够得以缓解。但想从根本上解决，还是不太容易。

Pegasus 的定位

既然搞不定 HBase，那么我们只好自己重新造一个。由于已有 HBase 这样一个标杆，我们在定位上也非常明确：对 HBase 取长补短。具体来看：

- HBase 的一致性的视图和动态伸缩，是存储系统里非常好的两个特质，我们希望可以保留；
- 对于 HBase 的性能延时和可用性问题，我们应该通过架构和实现着力弥补，从而将系统覆盖给更多的业务使用。

有了这几个定位之后，我们架构也比较清晰的浮出了水面：

Pegasus 架构一览

总体来说，该架构借鉴了不少 HBase 的内容：

- 从总体来看，Pegasus 也是一个中心化管理的分布式存储系统。MetaServer 负责管理集群的全局状态，类似 HBase 的 HMaster；ReplicaServer 负责数据读写，类似 HBase 的 RegionServer。
- 为了扩展性考虑，我们也对 key 分了不同的 partition。

和 HBase 不同之处在于以下几点：

- 心跳并没有依赖 Zookeeper，而是单独抽出来，直接由 MetaServer 进行管理
- 数据不写到第三方的 DFS 上，而是直接落入 ReplicaServer。
- 为了对抗单个 ReplicaServer 的失效，每个 Partition 都有三个副本，分散到不同的 ReplicaServer 上。

目前 MetaServer 的高可用依赖于 Zookeeper，这是为了项目开发简单上的一个权益之计。后面可能引入 Raft，从而彻底消除对 Zookeeper 的依赖。

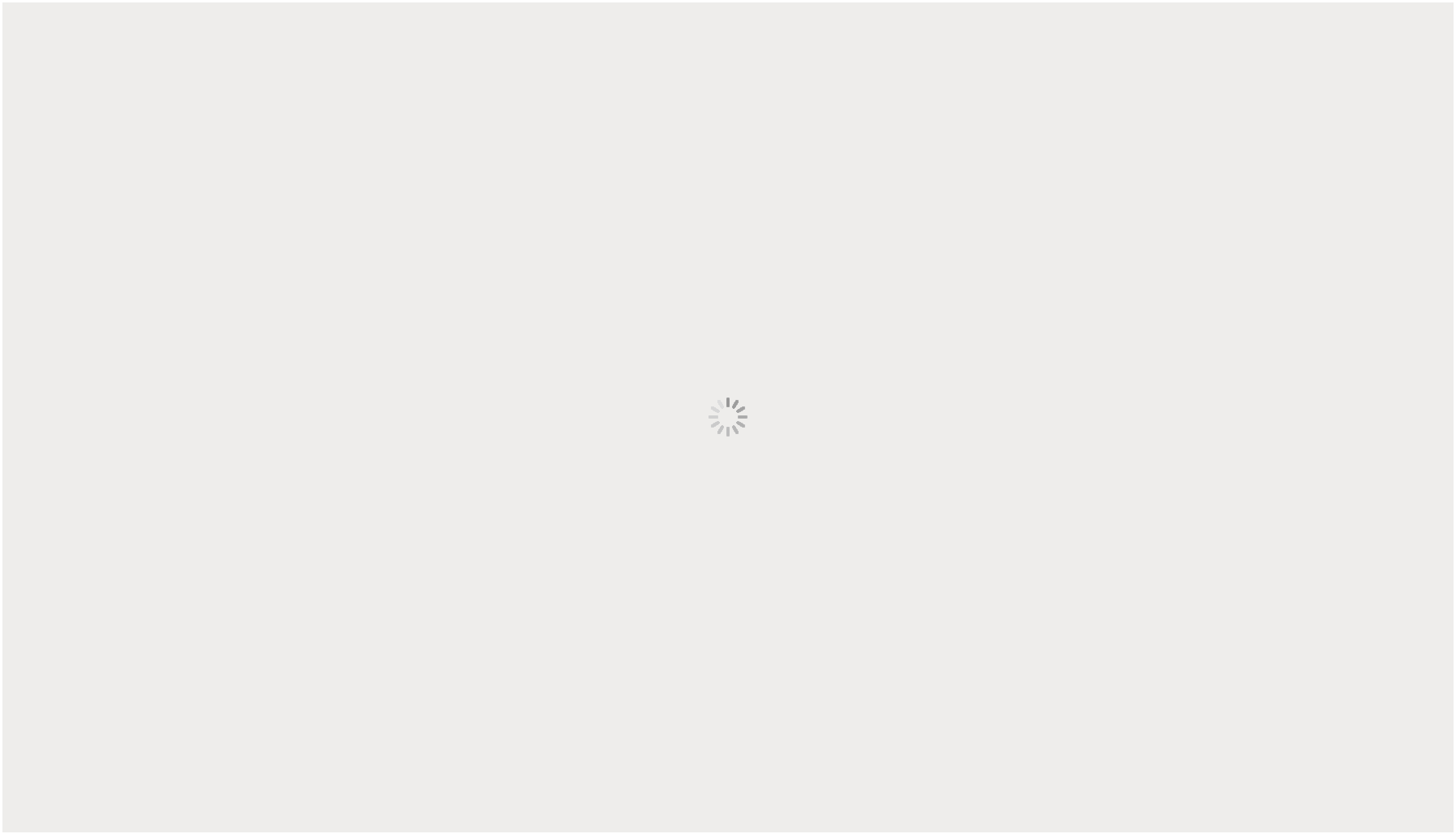
多副本的一致性协议

前面提过，我们的每个 Partition 都是有多副本的。为了满足多副本情况下的强一致性，我们必须得采用一致性协议算法。我们使用的是 MSRA 发表的 PacificA。PacificA 和 Raft 的对比可以参考我们在 github 项目中的文档，这里不再详细展开。总的来说，我们认为依照 PacificA 的论文来实现一个可用的存储系统，其难度要比 Raft 更低。

Pegasus 写请求流程

在 PacificA 协议里边，负责接受读写请求的叫做 primary，相当于 Raft 中的 leader；另外两个接受 replication 请求的是 secondary，相当于 Raft 中的 follower。在如此架构下，对一个 partition 写请求也比较简单：如果客户端有一个写请求，首先需要向 MetaServer 来查询 Key 的位置，之后再向 primary 所在的 ReplicaServer 发起写请求，然后 primary 将其同步给 secondary，二者都成功后再返回客户端写请求成功的回复。

Pegasus 的读请求流程图



读请求操作更加简单，客户端直接和 primary 发起读请求，primary 因为拥有全部数据，所以直接进行响应。

实现上的那些坑

前面大致回顾了 Pegasus 的设计考虑和总体架构。现在进入我们的正题，来看看这样的架构在实现上会有哪些问题?

扩展性

首先来看扩展性。扩展性分为两点：

- partition schema 的问题：一个完整的 key 空间，怎么样把它分成不同的 partition；
- load balance(负载均衡) 的问题：如果 partition schema 定了之后，怎么样把这些分片用一个比较好的算法分到不同的机器上。

Partition schema 的选取

对于 partition schema，业界解决方案一般有两种，第一种是哈希，第二种是排序。

对于哈希，就是将所有的 key 分成很多桶，一个桶就是一个 partition，然后根据 key 的哈希值来决定分配到某个桶中；对于排序而言，所有 key 一开始都在一个桶里边，然后依据桶的容量进行不停地动态分裂、合并。

对比这两种方案，排序比哈希多一个天生的优点是排序方案有全局有序的效果。但由于排序需要不停地做动态分裂，因此在实现上更麻烦。

另外，如果采用 hash 的 partition schema，数据库不太容易因为业务的访问模式而出现热点问题。排序由于将所有的 key 都按序排列，相同前缀的请求很有可能会被集中在一个或者相邻的 partition 中，而这些请求则很有可能落在同一台机器上。而对于哈希来说，所有的 key 已经预先都打散了，它自然而然就没有这个问题。

不过，单纯对比两个方案的优劣意义并不大，还是要从业务出发。我们认为，在互联网的业务场景中，不同的 key 之间并不需要有一个偏序关系 (比如两个用户的名字)，所以在权衡之后我们采用哈希方案。

Hash schema 实现方式

在实现 HashSchema 的时候，我们遇到的第一个问题就是“数据怎么存储”的问题。“把一个 key 依据 hash 值落到一个桶中”，这只是一个理想化的抽象而已。在真实的系统实现上，你必须还得提供一层“表”的概念把不同的业务给分割开。有了表的概念后，我们在实现上就开始产生分歧了。具体来说，我们在存储上一共有“多表混存”和“分开存”两种方案。

所谓多表混存，就是将表 ID 以及 hash key 合起来算一个新的哈希值，根据这个哈希值从全局唯一的 partition space 里选取一个 partition 做存取。如上图的左半部分所示。

而对于分开存而言，把表的语义下推到了存储层。如上图的右半部分所示：每个表都有一个单独的 partition space，当有一个读写请求时，要先根据表 ID 找到对应的 partition space，再根据 hash key 去找对应的 partition。

那么这两种方案到底哪一个更好呢？从理论上来说，我们认为多表混存方案更好，因为它更符合软件工程中分层的思想；混存也更容易实现，因为只需要管理一份元数据，负载均衡也更为简单。但是从业务角度来看，采用各表分开存的方案更好。因为分开存意味着更简单的表间资源限制、表级别的监控和删除表的操作。考虑到运维上的误操作，分开存储也更有优势，即使你误删了一个 partition，该错误操作扩散给不同业务的影响也是要少一些。

下表给出了我们对两种方案的对比：

两者相比，一个理论上更优美，一个实践上对业务更友好。最后我们还是从业务出发，我们放弃了理论更优美也更容易实现的方案，而选择了对业务友好的方案。

Hash schema 的负载均衡

说完 hash schema，接下来就是负载均衡的问题。下面列出了一般分布式 KV 存储在负载均衡上的目标：

- 单个 Key 的请求过热：不论哪种存储方案都不太容易解决。如果是读，可以考虑再加上层的只读缓存；如果是写，只能让业务将 key 做进一步拆分
- 单个 Partition 的请求过热：在 hash schema 下无需考虑，因为 key 是 hash 分散的。
- Partition 的容量分布不均：在 hash schema 下无需考虑，因为 key 是 hash 分散的。
- Partition 的个数在不同机器上分布不均匀：这一点需要处理，而且这一点处理好之后，读写请求在不同 ReplicaServer 的请求就比较均衡了。

负载均衡 - 目标

具体来看，负载均衡的目标有两点：

A.primary 和 secondary 不能共享 ReplicaServer

B. 对于每个表， primary 和 secondary 都能在不同的 ReplicaServer 上平均分配

上图是对目标 B 的一个简单说明：假如一张表有四个 Partition，而一共有三个 ReplicaServer，我们希望 12 个 Replica 的分布情况是 (1, 3), (2, 2), (1, 3)。

负载均衡 - 算法

在实现我们的负载均衡算法时，有一个很重要的注意点是：角色切换要优于数据的拷贝，因为角色切换的成本非常低。

对于这点的说明，可以参考上图中的两种情况：

1. 在左图里，4 个 Primary 分布在 ReplicaServer2 和 ReplicaServer3 上，而 ReplicaServer1 上没有任何 Primary。这时候，通过把 ReplicaServer3 上的 Primary A 和 ReplicaServer1 上的 Secondary A 做角色对调，就可以满足 Primary 的均衡。
2. 在右图中，4 个 Primary 在四个 ReplicaServer 上的分布是 (2, 1, 1, 0)。如果想要做 Primary 的均衡，需要把 ReplicaServer1 上的一个 Primary 迁移到 ReplicaServer4 上，但直接迁移 Primary 需要拷贝数据。此时，我们如果引入中间节点 ReplicaServer2，先把 ReplicaServer1 的 Primary A 和 ReplicaServer2 的 Secondary A 角色互换，再把 ReplicaServer2 的 Primary D 和 ReplicaServer4 的 Secondary D 互换，就能实现 Primary 的均衡。

为了处理这些情况，我们对集群中 Primary 可能的流向建立了一个有向图，然后利用了 Ford-Fulkerson 的方法进行 Primary 的迁移调换。具体的算法不再展开，可以参见我们的开源项目代码。

在真实的负载均衡中，还有很多情况需要考虑：

- 为了更好的容错，不应该把一个 Partition 的几个副本都放在同一个机架上。
- 不同的 ReplicaServer 可能有不同的存储容量，副本个数的绝对均衡可能不很合理。
- 在做负载均衡的数据拷贝时，一定要注意限流，绝对不能占用过多的系统资源

在当前 Pegasus 的开源项目上，有部分情况还没有做考虑。后面在负载均衡上我们会持续做优化，大家可以持续保持关注。

一致性和可用性

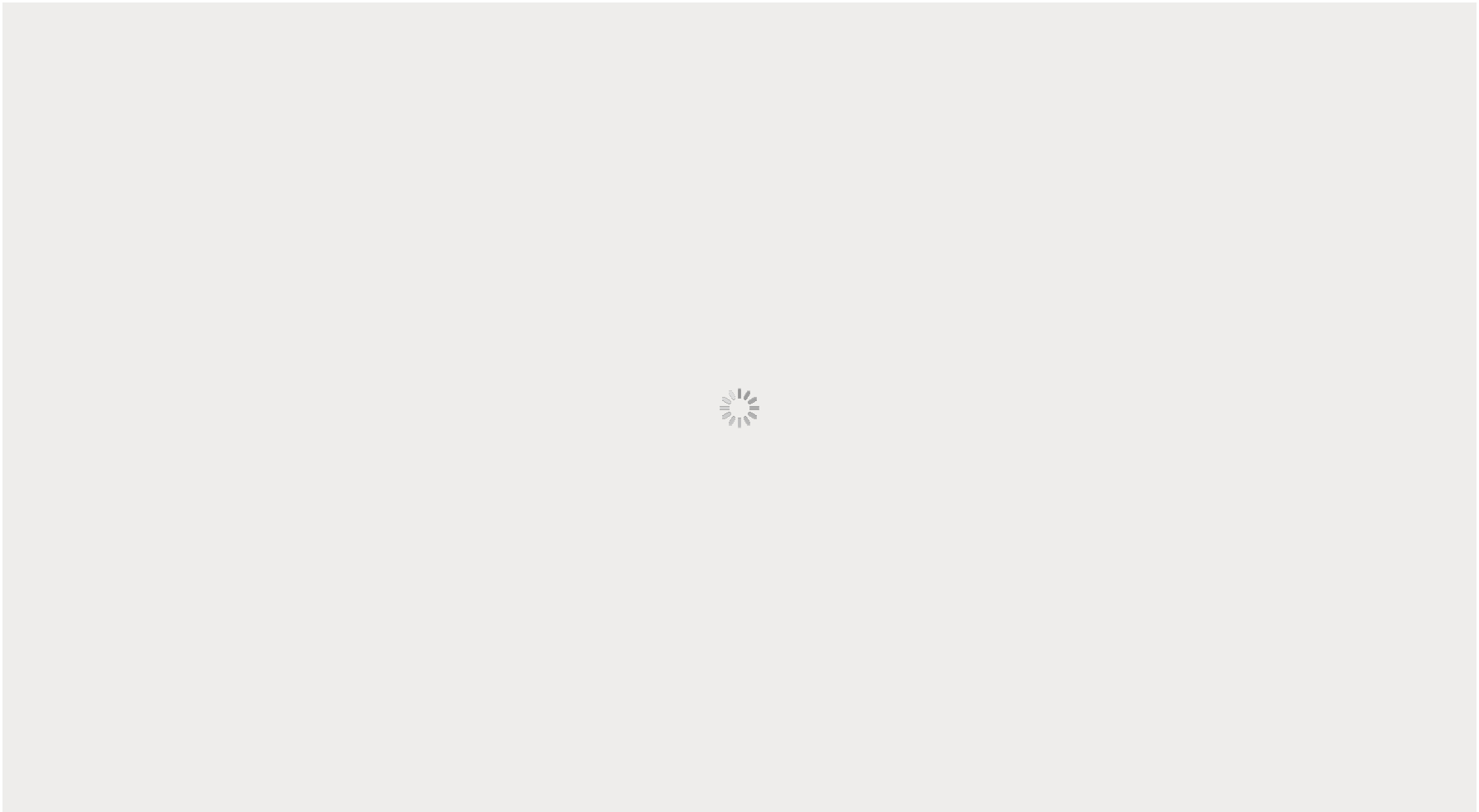
前面介绍了扩展性，接下来是一致性和可用性。我们在设计上的考虑前面已经介绍过了：

- 减少对 zookeeper 的依赖
- 数据不写在 DFS 上
- 多副本、用 PacificA 算法来保证强一致

当我们按照这些设计目的将系统实现出来，并准备找业务小试牛刀时，业务一句话就将我们顶了回来：“你们有双机房热备么？”

为什么要单独说这一点？因为对于一个强一致性的分布式存储系统而言，跨机房的容错是一

件比较麻烦的事情：



- 一方面，假如把一个 Partition 的多个副本部署到两个机房中，一定会有一个机房拥有多数派。而这个机房的整体宕机，意味着集群不可服务 (因为要保证强一致性)。
- 另一方面，假如把 Partition 的多副本部署的三个机房中，跨机房的安全性的确得到了保障。但是这意味着每一条写请求都一定会有跨机房的 replication，从而影响性能。

通过对这个问题进行反思，我们认为我们其实钻了一个“要做完美系统”的牛角尖。就业务的角度来看，作为一个完善的存储系统，的确要对各种各样的异常情况都需要做好处理。但随着异常情况发生概率的降低，业务对一致性的要求其实也是逐步放宽的：

在了解了业务的需求后，我们为 Pegasus 设计了多级的冗余策略来应对不同的风险：

- 一致性协议：用于单机房部署，抵御单机失效的风险
- 跨机房 replication：异步复制每个 partition 的 log，双机房都可写，基于 NTP 时间戳的最终一致性
- Snapshot 的定期冷备份：跨地域复制，发生极端故障时的兜底策略，可能会丢失部分数据

对上述冗余策略的几点说明：

- NTP 的时间戳可能不够精确。两个机房对同一个 key 的先后写，最终结果可能会是前者覆盖后者。目前我们并没有更进一步的做法来规避这点，因为我们的目标是“最终一致性”，即“随着时间的流逝，两个机房最后的结果是一样的”，所以即便是先写入的覆盖后写入的，最终在两个机房的状态表现也是相同的。
- 就我们在维护 HBase 的经验来看，业务两个机房同时写一个 key 的可能性是很小的。所以 NTP 带来的“先写覆盖后写”的问题，需要业务自己做注意。
- 多级的冗余策略可能会造成数据的备份数比较多。首先，并不是所有业务都需要这么多的冗余级别，而是根据具体需求进行在线配置的。所以这些限制冗余目前我们做成了表级别的可以修改的参数，而非集群级别强制的。再者，有些备份，如 snapshot 冷备，其存储介质是非常廉价的。
- 跨机房 replication 和 snapshot 冷备份的功能还在内部测试阶段，开源版本中还没有开放出来。

延时保证

最后介绍一下保证延时性能方面的问题。对于这个问题，有两点需要强调一下：

- 选择什么样的实现语言
- 怎样高效的实现一致性协议

在实现语言上，我们选择了 C++。原因前面也说过，为了性能保障，我们必须采用没有运行时 GC 的语言。另一个选择可能是 Rust。在为什么选 C++ 而不选 Rust 上，我们的考虑如下：

- 当时项目立项时 Rust 还不太火，第三方库也远没有 C++ 完善。
- 只要遵守一些编程规范，C++ 还不算特别难以驾驭。
- 使用 C++ 招人更方便些

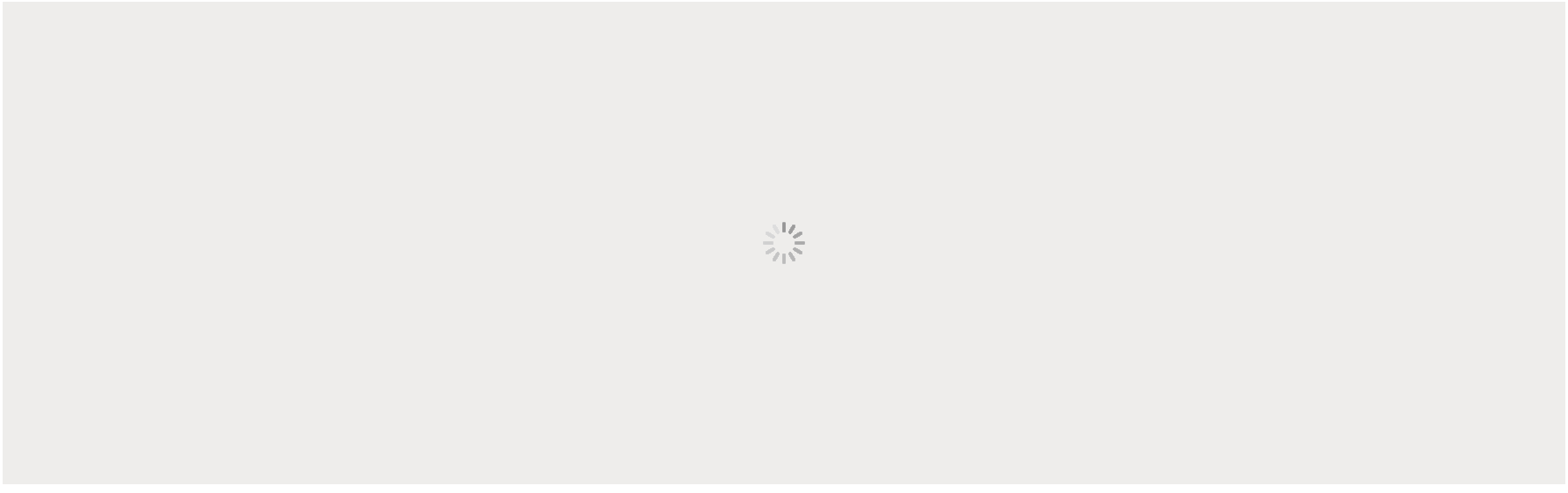
当然这个选择是偏保守型的，在语言层面的探讨也到此为止。重点还是说下我们一致性协议实现的问题。

就实践来看，想要把一致性协议实现的正确、高效，并且还要保证代码是易维护、可测试的，是一件比较难的事情，主要原因就在于一个完整的请求会涉及很多个阶段，阶段之间会产生 IO，再加上并发上的要求，往往需要对代码进行很细粒度的加锁。

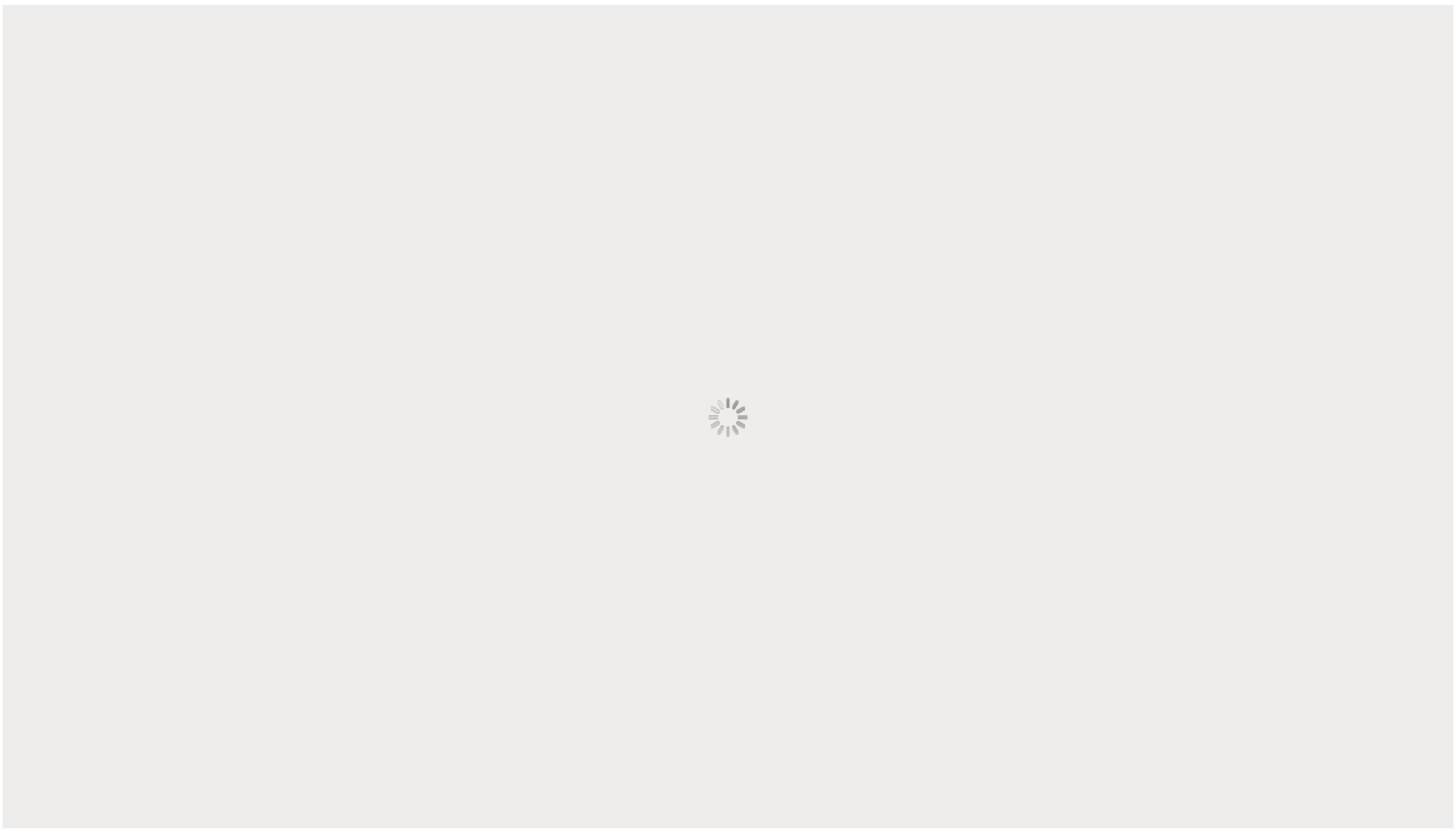
下图给出了一个完整写请求的流程简图：

从上图可以看出，当客户端发起一个写请求后，这条请求会先后产生写本地磁盘、以及发送网络 RPC 等多个事件，无论事件成功还是失败，都需要对公共的一致性状态进行访问或修改。这样的逻辑，会使得我们要对一致性状态进行非常繁琐的线程同步，是非常容易产生 bug 的。

那么这种问题该怎么处理呢？就我们这边的经验来看，是要把代码的组织方式从“对临界区的争抢”到“无锁串行化的排队”。我先用一张图来说明一下我们的这种代码架构：



具体来看，就是以“一致性状态”为核心，把所有涉及到状态更改的事件串到一个队列中，通过单独的线程取队列事件执行。所谓执行，就是更改一致性状态。如果执行过程中触发了 IO，就用纯异步的方式，IO 完成后的响应事件又会串到队列中来。如下图所示：



另外，为了避免创建过多的线程，我们采用了线程池的做法。一个“一致性状态”的数据结构只会被一个线程进行修改，但多个结构可能共享一个线程；replica 和线程的，是多对一的关系。

总的来说，通过这种事件驱动和纯异步的方式，我们得以在访问一致性状态时避免细粒度的锁同步。CPU 没有陷入 IO 等待中，以及线程池的使用，性能方面也是有保障的。此外，这种实现方式由于把一个写请求清晰的分成了不同的阶段，是非常方便我们对读写流程进行监控的，这对项目的性能分析是非常有好处的。

Determinnistic 测试

接下里我们重点讲一下测试是怎么做的。

分布式系统稳定的问题

当我们把系统做下来之后，发现真正长期困扰着我们的其实是如何将系统做稳定。

这个难题主要体现在哪几个方面呢？总结之后有以下三点：

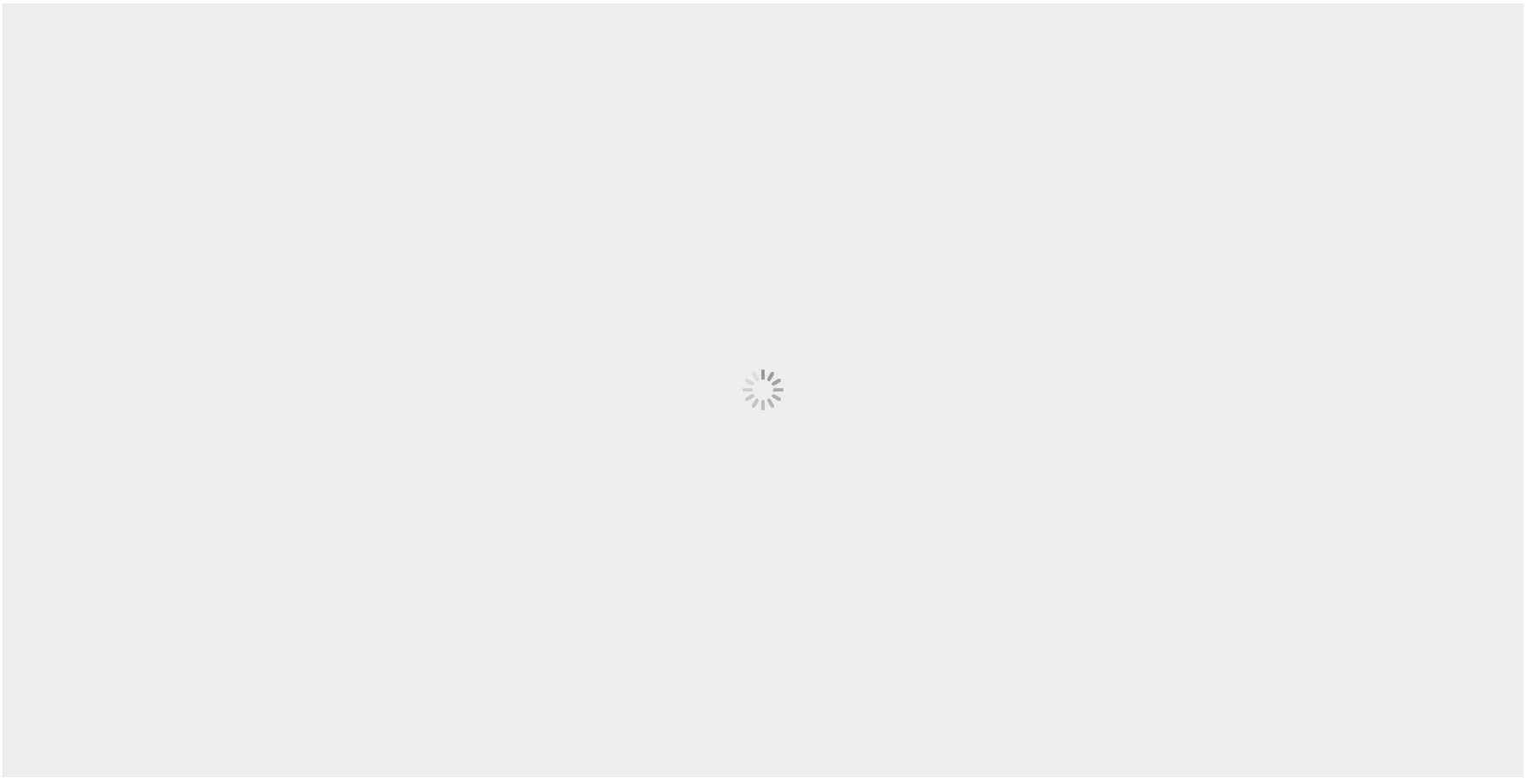
1. 难以测试，没有较有效的方法测试系统的问题。现在的经验便是将它当成一个黑盒，读写的同时杀任务，想办法使它的某一个模块出现问题，再去查看全局是否出现问题。然而这样的测试方法只能去撞 bug，因为问题是概率性的出现的。
2. 难以复现。因为 bug 是概率性出现的，就算通过测试发现了问题，这个问题也不太容易复现，从而进一步对调试带来困扰。
3. 难以回归。假如通过看 log、观察现象、分析代码找到了问题的症结。你的修复方法是不是有效也没有说服力。这样修是不是能解决问题？会不会引发新的问题？因为没有稳定复现的方法，这两个问题是很难回答的。

根源：不确定性

那么造成以上那些难点的根源在哪里？总结一下，我们认为是程序自身的不确定性。该不确定性体现在两个方面：

- 程序自身的随机：系统 API 如调度、定时器、随机数的使用，以及多节点间的并行。

- IO：程序可能要涉及到外设的 IO。IO 的概率性失败、超时、丢包、乱序等情况，也导致了程序不确定性。



用一个公式概括一下：

> 小概率的 IO 错误 + 随机执行路径 = 不容易复现的异常状况

那么对于这个问题，我们应该怎么解决?

既然在线上很难复现问题，那么能不能构造一种模拟的场景：在这个模拟场景里边，我们可以模拟 IO 错误的概率，也可以控制程序的执行顺序。再在这样的模拟场景里运行代码，如果逻辑真的出现了问题，我们就可以按照相同的执行顺序把问题复现出来。逻辑修改后，还可以进一步做成单元测试，这样难以回归的问题也就解决了。

当然这样描述这个问题，还是非常抽象。这里我举个简单的例子来说明下：

假设有这样的一个账户系统，里面有 Alice 和 Bob 两个人，两人账户的余额各为 100 元，分别存储在两台机器上，两人均可以向对方发起转账的交易。现在 Alice 要向 Bob 转账 5 元，最简单的实现是 Alice 把自己账号上的余额扣减 5 元，然后向 Bob 所在的机器发起一个增加 5 元的请求。等 Bob 的账号增加 5 元后，就可以通知 Alice 说转账成功了。

如果机器不会宕机、磁盘不会出故障、网络也不会出问题，那么这种简单的实现方式并无不可。但这样的假设绝对不会成立，所以为了应对这些方面的问题，我们可能加入一系列的手

段来让账号系统变得可靠起来，诸如增加交易日志，把交易和账户信息备份到多个机器上，引入一些分布式事务的技术。这些手段的引入，会使得我们的系统变得复杂起来。

面对这样一个复杂的系统，任何一个异常的数据库状态都会使我们抓耳挠腮，比如 Alice 和 Bob 的账户信息的如下变迁：

```
(Alice_100, Bob_100) -> (Alice_105, Bob_105)
```

程序没有 bug 时，我们可以假定 Alice 和 Bob 账户的余额之和是等于 200 的。现在余额和变成了 210，一定是某个环节出了问题。也许是两人同时向对方转账时，然后触发了什么 bug；也许是数据包被发送了两次。进入非法状态的可能情况有很多种，但硬件概率性的失败以及 Alice、Bob 间复杂 (可能是并行) 的转账记录会使得我们不太容易把产生非法状态的原因定位并复现出来。

而对这样的一个问题，我们的武器是模拟和伪随机。通过这两种手段，我们希望程序能按着可复现的事件序列运行。这样假如程序因为进入非法状态而 crash，我们可以对该状态进行复现、调试和回归。

还拿上面 Alice 和 Bob 的交易为例。一个双方同时转账的流程在模拟的环境里可能是这样运行的：Alice 发起转账 –> Bob 发起转账 -> Alice 发起写盘 -> Alice 发起 RPC -> Alice RPC 成功 -> Bob 发起写盘 -> Alice 写盘失败

换句话说：

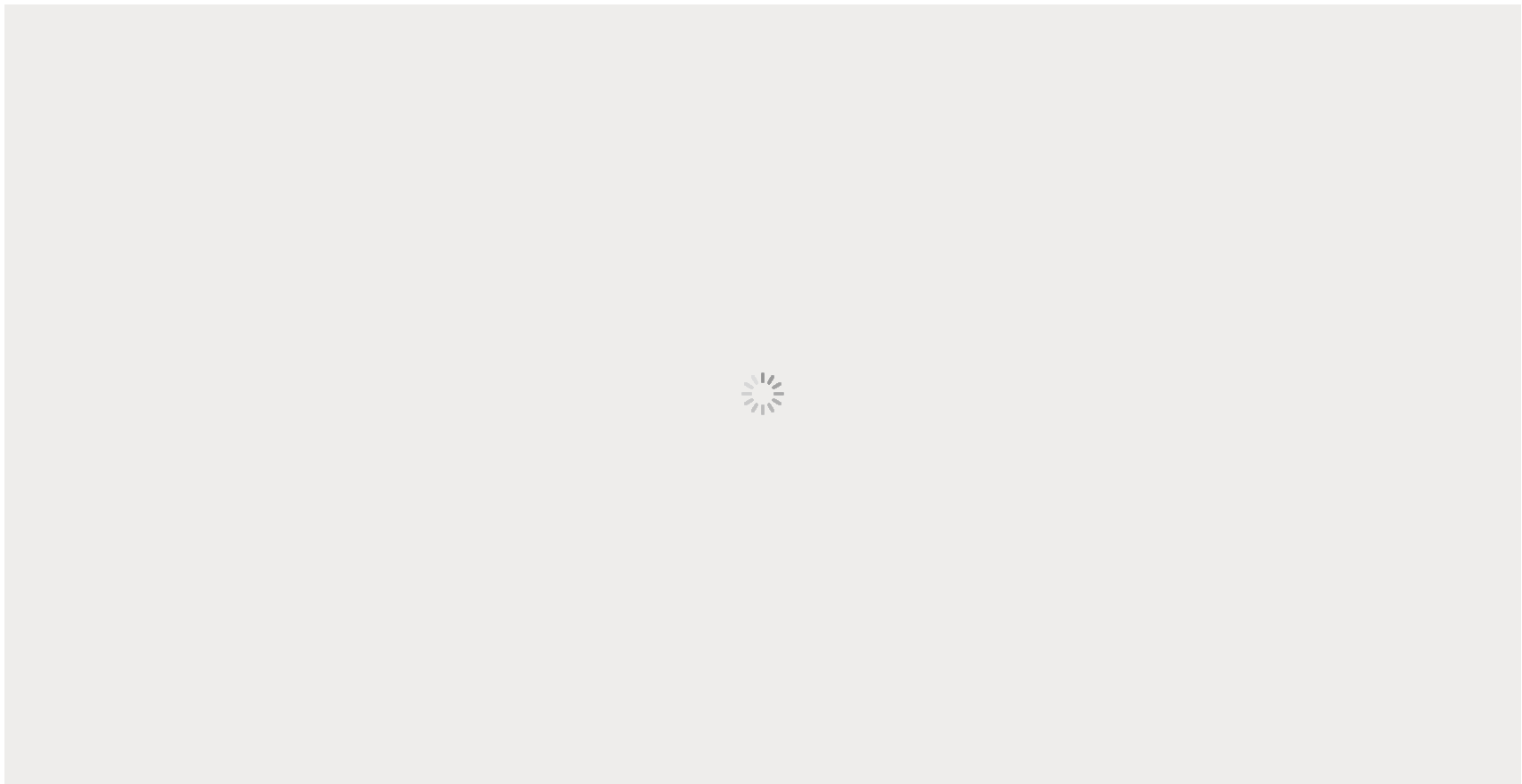
- 原本在并行执行的两套逻辑可能在模拟器里变成了并发交错的方式
- 交错时候的调度选择是伪随机的，运行不同的模拟器实例，它们的状态变迁流程是完全不同的；但对于任何一个实例，只要选定了相同的随机数种子，其变迁流程就是确定的
- IO 的失败也都是按照一定的伪随机概率进行错误注入的

假设能有这样的一个环境，分布式业务逻辑的 debug 难度就一定会降很多。一方面，随机的存在使得我们有测试各种执行序列的能力；另一方面，伪随机和错误注入，使得我们可以复现遇到的问题。

控制不确定性

那么，具体的控制不确定是怎么做的呢?

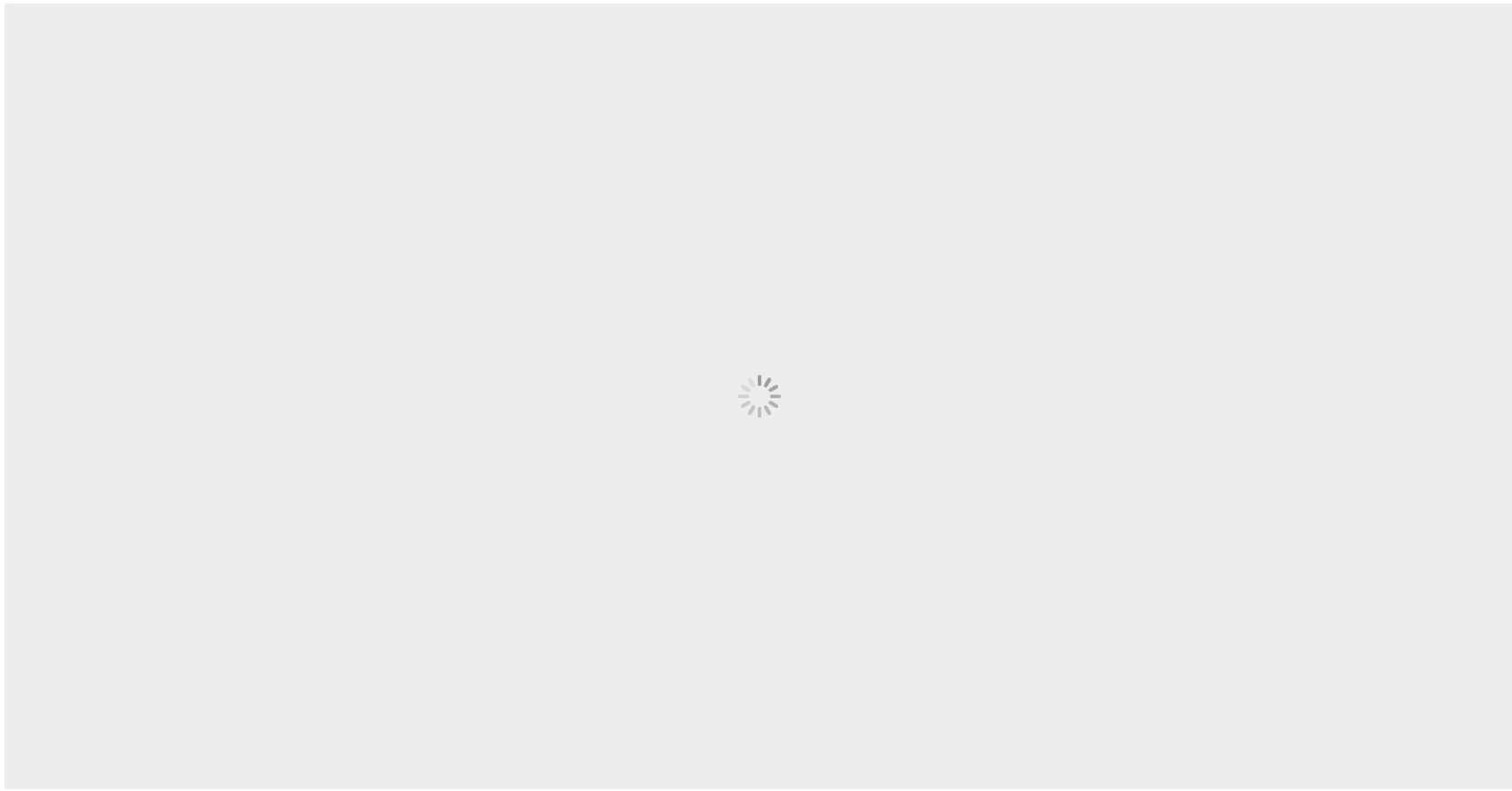
- 产生不确定性的接口提供抽象层。当前系统一共提供了线程池、RPC、磁盘 IO、线程同步和环境操作 5 种抽象接口。每种接口都有模拟实现 (测试) 和非模拟实现 (部署运行)，分布式业务逻辑 (Application Logic) 的代码只调用系统的抽象接口 (Runtime)
- 系统提供能在单进程中模仿多个服务节点的能力，从而达到模拟分布式系统的效果。



Runtime 层的模拟实现是一个难点，重点包括：

- 只提供纯异步的编程接口，节点的运行模式就是线程执行事件队列中任务的过程。
- 当业务逻辑通过 Runtime API 调用到 Runtime 的模块内部时，Runtime 会将当前节点挂起，并按照伪随机的方式选取一个新的节点进行执行。整个过程请类比单核多任务的操作系统，runtime 就像内核态，分布式逻辑的 Node 就像用户态进程，runtime API 就像系统调用。如下图所示：

- 对于提交到事件队列中的 IO 事件，我们按照一定的概率伪随机的注入错误。另外，引入了一些全局的状态检测模块，在每次状态变化之时检测全局状态是否合法 (比如检查 Alice 和 Bob 的账户余额之和是不是小于 200)；同时，Application Logic 的代码也加入了很多的 assert 点来检查 Node 状态是否合法。程序 crash 后，就可以用相同的随机数种子进行复现和调试。如下图所示：



Pegasus 的单元测试也是利用这套测试框架进行的。在单元测试中，我们会用一个脚本来描述为一个场景施加的动作以及预期的状态，上图的 App Logic Checker 的功能就是加载脚本，然后按照脚本的方式检测程序状态是否符合预期。

Pegasus 所采用的这套测试框架，其理念和实现均来源于微软的开源框架 rDSN。整个框架较难以理解，这里也只是简述下其原理。大家如果感兴趣可以直接查看源代码。

现状和计划

现在 Pegasus 在小米内部已经稳定服务将近一年左右，服务了近十个业务。有关 Pegasus 的存储引擎、性能以及设计方面的更多阐述，大家可以移步 Arch Summit 2016 的另一次分享（文后有链接），也可以参考我们在 github 上的相关文档。

后续我们会把数据的冷热备份、Partition 分裂等功能都开源出来，请持续保持关注。也欢迎各路人士加入小米，加入我们团队。

写在最后

最后概括一下全篇的内容。

当我们在调研或者实现一个项目时，一定有三点是要关注的：

- 关注业务：项目是否满足业务需求
- 关注架构：项目的架构是否合理，对于分布式存储系统，就是一致性、可用性、扩展性、性能这几点
- 关注软件工程：项目的测试是否完善，代码的可维护性是不是很好，项目能不能进行监控

相关链接

- Pegasus 项目地址：
  https://github.com/XiaoMi/pegasus
- Pegasus 在 Arch Summit 2016 上的分享：
  http://bj2016.archsummit.com/presentation/3023
- rDSN 项目地址：
  https://github.com/Microsoft/rDSN
- Pegasus 维护的 rDSN 分支：
  https://github.com/XiaoMi/rdsn

- PacificA 论文链接：
  https://www.microsoft.com/en-us/research/wp-content/uploads/2008/02/tr-2008-25.pdf

关于我们

我们是小米公司人工智能与云平台部门的云存储团队，团队的职责是开发和维护分布式存储系统，为整个小米公司及生态链企业提供分布式存储解决方案。

我们团队开发和维护的系统包括：Zookeeper、HDFS、HBase、Pegasus，以及基于这些系统封装的 FDS、SDS、EMQ 等服务。

如果有问题（不限于技术上的问题），欢迎联系我们。同时，我们也随时欢迎有志之士加入。

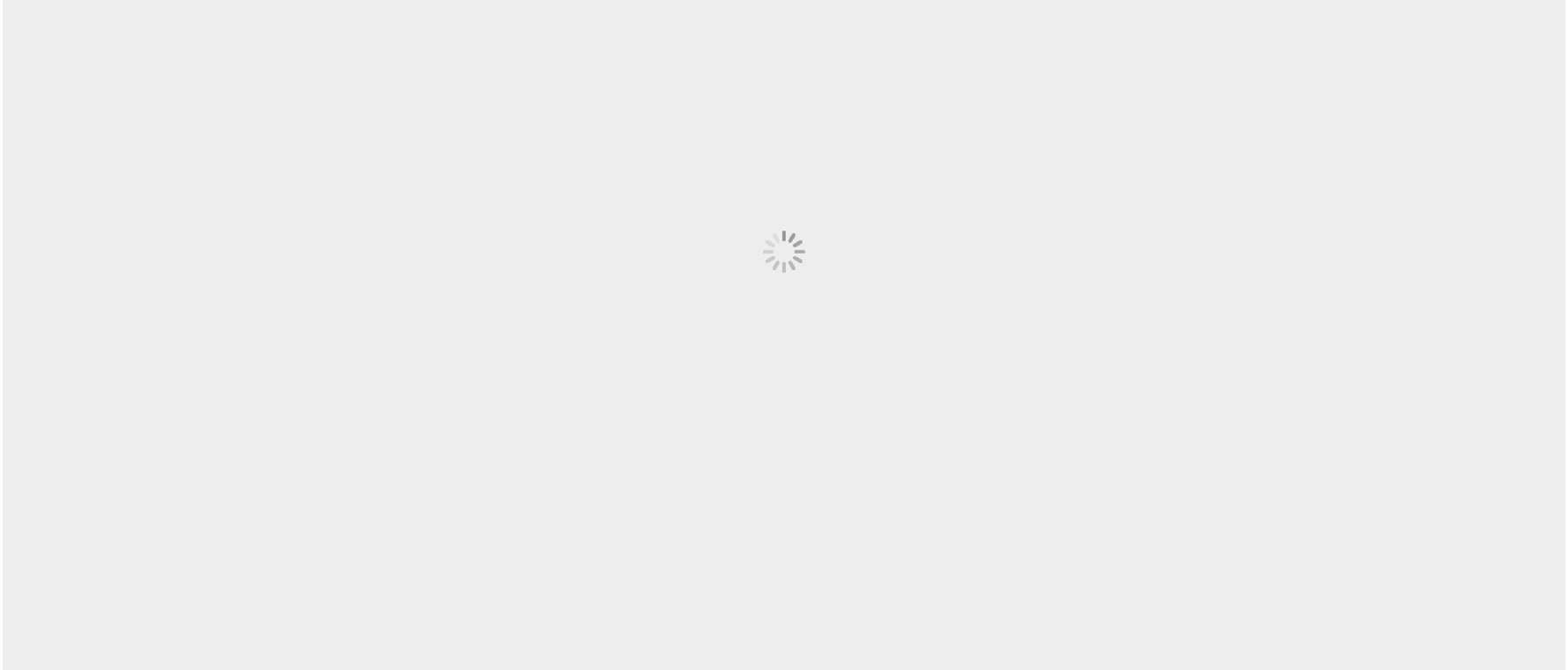
- 邮箱：pegasus-help [at] xiaomi.com
- 微博：小米云技术

作者介绍

孙伟杰，云存储工程师，浙江大学硕士毕业，硕士期间主要研究方向为操作系统和虚拟化。目前就职于小米，致力于分布式存储系统 Pegasus 的研发工作。热爱底层技术，热爱开源，是分布式框架系统 rDSN 的重要开发者。微博：shengofbig，知乎：zhihu.com/people/niney。

今日荐文

点击下方图片即可阅读



一名优秀的 CTO，需要具备哪些特质?