松耦合系统中的 chubby 锁服务

Mike Burrows, Google Inc.

译：何磊，Baidu Inc.

摘要

chubby 是一项可以在松耦合系统中提供粗粒度锁服务，同时还保证可靠小型存储的技术。接下来我们将对 chubby 进行详细描述。尽管 chubby 本身提供了类似分布式文件系统接口，但 chubby 的设计重点仍是提供可靠性与高可用性，而并非高性能。目前已经实际 chubby 系统运行多年，其中一些需要并发处理来自客户端的上万个连接。本文会讲述 chubby 的设计及期望的使用方式，同时和实际使用情况进行对比，并且描述为了满足实际需求对 chubby 设计上必须进行的改进

介绍

本文描述了一项名为 chubby 的锁服务。chubby 被设计用于大规模，松耦合的分布式系统中，整个系统通过高速网络连接。例如，一个 chubby 可能会服务千兆带宽环境中的上万台 4 核机器。多数的 chubby cell 可以放置于一个数据中心，然而我们还是至少会将一个 chubby cell 的副本部署在千里之外

锁的目前是使其客户端行为得到同步，并且对环境信息的认识达成一致。chubby 的主要目标是提高大规模集群的可靠性，可用性，此外接口语义的清晰也很重要，相比之下吞吐和存储容量则次要考虑。chubby 的客户端接口类似于普通文件系统，并且提供了全文件读

-写操作接口(即对文件的读写都是基于全文而非部分)，同时增加了建议锁及文件修改时的各项事件通知机制。

我们期望 chubby 能够帮助开发者在其他系统中解决粗粒度锁同步的问题，特别的，在大规模服务器中解决选主问题。举例来说，GFS 中使用 chubby 来选举 GFS master；bigtable 则使用的更多：选主，主发现从，客户端的服务器发现等。不仅如此，GFS 和 bigtable 还将 chubby 做为全局可见的高可用存储，用来存放小规模元数据；实际上他们使用 chubby 做为其分布式数据结构的 root。还有一些服务用锁划分多个服务器间的工作。

在 chubby 出现以前，google 的许多分布式系统使用 ad hoc 方式来进行选主(重复工作不会有损失的环境下)，或者需要 op 介入人工操作(要求高的正确性环境下)。在之前的例子中，chubby 提升了计算能力；后一个例子中，chubby 提供了显而易见的高可用性，使得不需要人工介入进行故障处理

熟悉分布式计算的读者都知道选主问题实际上是分布式一致性的一个特例，我们需要一个异步方案来解决选主问题；接下来将描述基于实际网络的解决方式。异步一致性在 paxos 协议中得到解决。实际上，目前为止的分布式协议都能找到 paxos 的影子。paxos 协议并不依赖于时间信息，时钟用来维持整个进程的持续性。这一点克服了 Fischer 的缺点(什么缺点？这文章没读过)

创建 chubby 实际上是为满足上述需求而进行的一项工程性工作，而并不是研究。我们没有引入新的算法或技术。本文将会描述我们的做法及原因，而不单单是做个广告:)。下面的章节中，我们将会介绍 chubby 的设计和实现，以及它如何在实践经验中被改变的。我们介绍了对 chubby 意料之外的用法，同时也介绍了一些在实践中被证明是错误的特性。我们省略了一些常见的细节，比如一致性协议以及 RPC 系统相关内容

设计

基本理论

可能有些人会觉得与其完成一个中心化的锁服务库，即使这个服务是高可用的，也还不如直接完成一个基于 paxos 的分布式库。一个客户端的 paxos 库可以不依赖于任何其他的服务器，并且如果上层应用可以表示为某种状态机的话，则它可以为程序员提供一套标准的框架。事实上，我们的确提供了独立于 chubby 的客户端库。

然而，相对于客户端库来说，一个锁服务也有其固有的优势。

第一，开发者之间并不总是期望同样的高可用性。通常他们的系统总是起始于较低的负载，并且对高可用也没有很高的要求，其代码也还没有专门针对一致性做特殊的组织优化。随着服务的成熟和客户的增多，高可用性变得很重要。冗余和选主被加入到一个已有的设计中。比起一个提供分布式一致性的库来说，锁服务将使这个过程更加简单，并且不需要修改程序的结构及通信。例如：选主并往文件服务器中写入信息的过程仅仅需要增加几行代码就可以搞定，首先请求锁，请求成功则意味着自身成为主，再往文件服务器上发一个 RPC 请求来写入信息即可。这一过程并不需要复杂的多服务器参与的一致性过程

第二，我们的很多项服务，在出现选主或数据分区时都需要对结果进行广播。这使得我们的客户端必须能够保存和获取小量数据—这就是说，读写小文件。这可以通过一个名字服务来完成，然而我们的经验说明锁服务本身其实非常适合这种应用，因为它不仅降低了服务的依赖性，同时还提供了一致性保证。chubby 做为名字服务器的成功很大部分归功于其选择了一致性的客户端 cache，而不是基于时间进行更新。特别是我们发现开发者们非常高兴不用去选择类似于 dns time-to-live 那样的 cache 超时时间，因为这个时间如果选择不当就会导致高 dns 负载或长时间的客户端失效问题。

第三，基于锁的接口对开发者来说更为熟悉。

最后，分布式一致性算法采用集群来确定做决定，因此它依赖于多个副本的机制来达成高可用。例如，chubby 在每个 cell 中通常使用 5 个副本，其中至少需要有 3 台可以正常运行才能保证整个 cell 正常。然而，即使是单客户端也可以通过锁服务来保证过程安全性。即，锁服务减少了为提供客户端过程安全的可靠服务所需要服务器的数量。

这部分提出了两个主要的设计方向：

1. 我们选择了锁服务而不是一致性服务或库
2. 为了允许选主及通知机制，我们提供小文件服务

为了达到我们期望的使用方式并且满足使用环境，我们有如下决定：

1. 一个通过一个 chubby 文件广播其主服务器的服务可能有上千个客户端。因此，我们必须允许上千个客户端 watch 这个文件，而不需要增加服务器数量
2. 客户端及服务器副本都期望在主服务器发生变更时获得通知。这使我们必须引入事件通知机制来避免轮询
3. 即使客户端并不需要定期去拉取文件，有些客户端也会这么做(林子大了，什么鸟都有)。因此，客户端缓存是必须的
4. 开发者们搞不清楚抽象的 cache 语义，因此我们需要提供一致性 cache，将开发者们从对 cache 关注中解放出来
5. 为避免财务损失及进监狱的风险，我们提供安全机制，包括 acl

我们并没有提供细粒度的锁服务，这种细粒度的锁仅仅只能被持有较短时间，一般是秒级或更短。相反的，我们期望应用使用粗粒度的锁。例如，一个应用可以使用锁服务进行选主，选主完成后可能需要保持数小时甚至数天时间。这两种不同的使用方式对锁服务提出了不同的要求。

粗粒度锁使得锁服务器的负载大幅度降低，并且锁请求的频度与客户端的事务频度基本无关。粗粒度锁只会偶尔被请求，因此暂时性的锁服务不可用不会导致大范围客户端延迟。另一方面，客户端间的锁传递将要求更高的恢复代价，因此我们不希望锁服务器的故障转移导致锁丢失。也就是说，粗粒度锁应当在锁服务器失效过程中存活。我们需要考虑这样做的代价，同时即使锁服务器处于相对较低可用性保证时，也能使客户端得到充分的服务。

细粒度的锁将会导致完全不同的结论。即使短暂的锁服务不可用也会导致许多客户端停止服务。增加新服务器的时，整个服务性能和可用性之间会有复杂的关系，因为事务频度的提升同时也会导致对锁服务请求的提升。如果锁服务器发生失效，则它可以通过在此时放弃锁来减少锁开销，由于锁本身被持有的时间很短，因此放弃锁的消耗并不会显得很严重。(由于客户端随时准备因为网络分割等因素放弃锁服务，因此服务器失效引发的放锁并不需要新的恢复机制)

chubby 仅仅为了提供粗粒度锁服务。幸运的是，客户端要实现自身定制化的细粒度锁服务也相当简单。应用可以将自身的锁划分 group，并且使用 chubby 提供的粗粒度锁来分配 lock group 到应用特定的锁服务器。服务器仅仅需要保存一个非易失，单调增加的获取计数器。解锁时客户端应该知道锁丢失，假设指定一个简单的定长租期，那么这个协议将会简单并且有效。这种设计最重要的好处是使客户端开发者负责服务器规则以使其能支持相应的负载，同时降低了他们自身引入一致性的复杂度

系统结构

chubby 有两个主要部件通过 rpc 通信：一个服务端，一个应用链接的客户端库。所有 chubby 客户端与服务器的通信都通过客户端库进行。代理服务器做为一个优化的组件将会在 3.1 中介绍

chubby cell 由一组服务器组成(通常是 5 台)，为了要减少关联失效需要对部署也做一定考虑，例如放置到不同机架位。各个副本间使用分布式一致性协议来选主，主必须获取多数副本的选票，同时保证在某一段时期内副本不会再选举不同的主，这个时期被叫做主租期。当主在新一轮选举过程中胜出时，主租期得以更新

副本都保存了一个数据库的拷贝，只有主对数据库进行初始化读写，其他的所有副本只是使用一致性协议对数据库进行 update 操作

客户端通过向列在 dns 中的副本发送主定位请求来获取主信息。非主的副本会响应主 id。一量客户端定位了主，则客户端将所有请求直接发给主，直到它停止响应或明确已经不再是主为止。写请求会通过一致性协议在所有的副本中提交，这种请求会在多数派响应之后被回复。读请求则仅由主进行回复。当主租期未到期时，由于没有其他主的存在，这种操作是安全的。如果主失效了，其他的副本将会在主租期过期时重新选举，新主通常在数秒之内选举完成。

如果一个副本宕机后数小时内没有恢复，则一个简单的替换系统会从可用机器池中选择一台新的机器进行替换。首先它从新机器中启动锁服务，然后对 dns 表进行更新，替换掉失效的机器 ip。当前的主周期性的从 dns 中拉取机器列表。当发现机器变更时会更新本地的 cell 成员数据库，此列表会在所有的成员中使用普通的一致性协议保持一致。同时，新的副本从文件服务器上获取到最新的数据库副本，并且从活跃的副本中进行更新。当新的副本服务器访问到当前主等待提交的请求时，即认为此副本达到最新，则允许此副本在后续的选举过程中进行投票。

文件，目录和句柄

chubby 提供了类似 unix 文件系统的接口，但更简单。chubby 由一棵包含目录和文

件的树组成，由’/’进行分隔。一个标准的名字类似于：

/ls/foo/wombat/pouch

ls 是 chubby 的标准前缀，代表着锁服务(lock service)。第二个组成部分(foo)是 chubby cell 的名字；这个名字通过 dns 会被解析为一台或多台 chubby 服务器。local 是一个特殊的 cell 名字，意味着应该使用的是客户端本地的 cell；通常情况下这应该与客户端处于同一建筑中，因此是最应该被访问的。名字的剩余部分/wombat/pouch 会在 chubby cell 中被解析。与 unix 一样，每一个目录包含了一系列的目录和文件，而每个文件则包含了一系列字节流

正因为 chubby 的名字结构类似于文件系统，我们即可以使用其提供的特殊 API 来访问 chubby ,也可以使用类似于 GFS 提供了文件系统接口进行访问。这降低了为使用 chubby 所需要编写工具的难度，同时也降低了潜在用户的学习成本。

和 unix 系统不一样的地方在于，chubby 很容易分布式化。为支持不同目录中的文件可以被多个 chubby 主服务，我们没有提供对文件的 move 操作，也没有保存文件的修改时间，同时文件权限与路径无关(即一个文件的访问仅由文件自身控制，与其所在的目录无关)。为使它能更简单的缓存元数据，系统也没有保存最后访问时间。

chubby 中保存了目录和文件，我们统一命名为节点。每一个节点在 cell 中有一个惟一的名字，没有符号链接或研链接存在。

节点可能是永久的，也可能是临时的。任意节点都可以被显式删除，此外当一个临时节点没有被任一个客户端打开时，此临时节点也会被自动删除(如果目录为空时也一样)。临时节点被用做临时文件，用来表示一个客户端是否还存活。任意节点都可以做为一把建议性的读写锁使用，章节 2.4 将会对此进行详细描述

每一个节点都有不同的元数据，其中包括三个 ACL 标志，分别对应着读，写和更改 ACL

权限。每个节点在创建时会自动从其所在的父节点(即目录)继承。ACL 是位于 ACL 目录的一些文件，此目录是在 cell 本地名字空间中全局可见的。一个 ACL 文件实际上包含了一些名字列表。因此 ,如果一个文件 F 的写 ACL 名字为 foo ,并且 ACL 目录下有 foo 这个文件，此文件中含有 bar 的入口，则用户 bar 可以对文件 F 进行写操作。用户在 RPC 系统中被授权。因为 chubby 的 ACL 机制由非常简单的文件组成，因此他们可以直接被其他类似的系统使用

节点元数据还包含 4 个单调增加的 64 字节数字

1. 一个实例号，比之前创建的同名节点的实例号都大
2. 内容号，当文件内容被改变时自增
3. 锁号，当节点的锁从 free 到 held 状态时自增
4. ACL 号，当节点 ACL 名字更改时自增

同时，chubby 还提供了 64 字节的文件内容较验，以便客户端对文件内容进行校验

类似于 unix 系统，chubby 通过打开文件来获取句柄。句柄内容包括：

1. 客户端进行创建文件时的检查位，因此完成的 ACL 检查可以只在句柄创建时进行
2. 一个序列号，此序列号用来区别一个文件是否是在某个主创建的
3. 打开时的模式信息。这允许发生主切换时，旧的文件句柄可以被重新创建并且维护一致的状态信息

锁和序列

任何一个 chubby 的节点都可以被用做读-写锁。跟临界区的概念一样，chubby 提供的是建议性的锁。我们没有选择强制锁：

1. chubby 的锁服务经常会被用于对其它服务进行互斥访问，而不仅仅是对锁文件本

身互斥。强制锁要求我们对其他服务必须进行额外的修改

2. 出于 debug 和管理的目的，有时我们需要对锁文件进行查看，此时并不希望强制用户关闭他们的应用。而强制性锁往往会导致用户关闭其应用以支持对节点的管理操作
3. 开发者通常会使用“lock X is held”这样的断言来进行错误检查，因此强制锁并不能带来额外的好处。错误或者是恶意的操作可能会造成数据错误，因此我们使用强制锁提供的额外监视来确保

在 chubby 系统中，任意加锁操作都需要拥有写权限。一个读者不能通过加锁阻止写进程。

分布式系统中的锁问题比较复杂，因为通信有不确定性，并且每个进程都可能独立终止。比如，一个进程持有锁 L，并且发出请求 R，之后此进程意外退出了；另一个进程请求锁 L，然后在请求 R 到达前完成一些操作。随后 R 到达，它可能会在锁 L 保护外做一些操作，并且导致数据的不一致。消息的乱序问题已经有很多的研究，并且已经有一些解决方案，比如 virtual time，virtual synchrony 等，它们都通过在所有消息参与者之间保持一致的顺序来解决此问题

但是往一个已经在运行的复杂系统中加入序列号的代价往往很大。chubby 提供了一种替代的解决方案：序列号仅在锁的交互过程中被使用到。任意时间，锁的持有者可以请求一个序列发生器，此序列发生器由描述锁当前状态的字节串组成。它包含锁名字，锁请求模式(独占式还是共享式)，锁序号。如果希望操作被锁保护，客户端把序列发生器传至服务器。参与的服务器都应该测试序列发生器是否处于 valid 状态，是否有正确的 mode，如果不是，则它应该拒绝请求。序列发生器的状态可以通过检测服务器本身的 chubby cache 来得到；如果服务器不希望与 chubby 保留一个长连接，那么也可以通过检测最近的序列发生器来获

取。序列机制仅仅需要在消息中加入一个字串，对开发者来说很容易理解

虽然我们认为序列发生器机制使用起来很简单，然而一些重要的协议并不那么容易进行修改。对于不支持序列发生器的服务，chubby 提供了一种不完美但更简单的机制来减少延迟及乱序请求带来的风险。如果一个客户端以正常方式放锁，那么这把锁将能够立即被其它客户端获取。然而，如果因为客户端失效导致的放锁，则锁服务器将会在一段时间内阻止其他客户端获取同一把锁，这个时间被称为锁延迟。客户端可以在范围内任意指定锁延迟时间，目前最大为 1 分钟；这个限制将会阻止一个已失效客户端永久持有一把锁。虽然并不完美，但锁延迟机制确实解决了消息延迟和乱序的问题

事件

当创建一个 handle 之后，chubby 客户端会接收到一系列的事件。这些事件会异步的发送给客户端，其中包括：

1. 文件内容更改。通常情况下用于通过文件来监控服务器地址
2. 子节点增加，减少，以及修改。用于实现镜像。(此事件可以在不增加节点引用计数的前提下监控到临时节点的变更)
3. 主故障转移。通知客户端可能有事件丢失，需要重新扫描数据
4. 句柄(包括其包含的锁)失效。这通常意味着网络错误
5. 锁请求。可以被用来检查是否已经选主
6. 锁冲突。允许缓存锁

如果一个客户端被通知文件内容更改，那么当它紧接着对文件进行读时，可以保证它能读到新的数据。

最后的两个事件事实上很少被使用，因此目前来看可以省略不用。以选主为例，客户端

通常都需要与新的主通信，而不仅仅是收到有主选出的通知。因此，客户端会等待直到新主的信息被写入到文件中，从而通过一个文件内容更改的事件来得到通知。锁冲突事件是为了从理论上支持客户端进行数据缓存，为了在多服务器缓存时保证数据一致性使用。一个锁冲突事件将会引起客户端结束对缓存数据的使用：结束 pending 的操作，将修改下刷，丢弃缓存数据并且释放锁。目前为止，还没有人如此使用。

API

对于客户端来说，一个 chubby 句柄是一个支持多种操作的应用不透明的指针。句柄通过 open()创建，通过 close()销毁

open()通过打开一个文件或目录来创建一个类似于 unix 文件描述符的句柄。只有此操作基于文件名进行，其他操作都基于已经打开的句柄进行

文件名必须基于一个已经存在的绝对路径。chubby 提供了一个基本的'/'做为根，这个目录总是有效的。绝对路径避免了在多线程环境下使用当前目录的晦涩

客户端支持多种操作：

1. 句柄的使用操作(读，写，加锁，更新 ACL)；仅当客户端拥有正确权限时才可以创建句柄
2. 事件通过句柄进行传递
3. 锁延迟时间
4. 判断是否可以新建一个文件或目录。如果文件被创建，调用者应当提供初始化内容及 ACL 信息。返回值会标志文件是否被创建

close()会关闭一个已经被打开的句柄，close()操作不会失败。对一个已经关闭的句柄的访问是非法的。另一个相关的操作是 poison()，此操作可以在不关闭句柄的条件下使对此句

柄未完成和后续的操作都失败；它用来退出其他线程对句柄进行的操作，同时不用担心会释放掉可能在被使用的内存

对于句柄的主要操作有：

1. GetContentsAndStat()，返回一个文件的内容及元数据。文件内容会被完全读取。我们不支持部分读写以避免大文件。另一个相关的 GetStat()操作则仅仅返回元数据。ReadDir()返回一个目录的子节点的名字及其元数据
2. SetContents()，写文件内容。此操作有一个文件内容号的可选项，客户端使用此号来对文件模拟一个 CAS 操作，文件仅在内容号与当前相同时才会更新。对文件内容的写操作也是完全写。一个相关的操作 SetACL()则会修改一个节点的 ACL 名字
3. Delete()，当一个节点没有子节点时进行删除
4. Acquire()，TryAcquire()，Release()用来获取和释放锁
5. GetSequencer()，返回一个序列发生器，此发生器描述了句柄中被持有的锁
6. SetSequencer()，将句柄和序列发生器相关联。如果此发生器失效，则后续的操作会失败
7. CheckSequencer()，检查序列发生器是否有效

如果句柄相关的节点被删除，即使后续同名的节点被重新创建起来，对此句柄的操作也会失败。即，一个句柄是与一个文件实例相关联而不是文件名。chubby 应该在所有调用中进行 acl 检查，然后通常情况下仅在 open()调用中进行了检查

所有的调用中除了正常的参数外，还需要一个操作参数，此参数用来保存调用中可能用到的数据及控制相关信息。客户端通常使用这个参数：

1. 提供一个 callback 以支持异步
2. 等待调用完成

3. 获取额外的错误及诊断信息

客户端可以按以下步骤使用此参数以完成选主过程 :所有参与者打开锁文件并且尝试获取锁。成功的一个成为主，其他为副本。主调用 SetContent()将其信息写入到锁文件，副本和客户端收到文件内容更改事件后可以通过 GetContentsAndStat()获取这些信息。理想情况下，主通过 GetSequencer()调用获取一个序列发生器，然后将此发生器传给它将要通信的服务器 然后通过 CheckSequencer()来确认此过程中主没有发生变化。如果不支持序列发生器，则可以通过锁延迟来进行后续操作

缓存

为了减少读请求带来了性能损耗，chubby 客户端对文件数据及节点元数据进行一致性，写穿方式的内存缓冲。缓存基于一个租期机制，此机制将会在后文中进行详细介绍，并且通过接收从主发来的失效事件来保持一致性。主维护了一个客户端缓存的列表。此协议保证了所有的客户端都看到一致性的数据，或者出错

当文件数据或元数据发生变更时，修改操作被阻塞，主发送失效命令到每一个可能缓存被修改数据的客户端；此机制基于 KeepAlive RPC，此 rpc 会在下面的章节中进行详细讨论。当接收到失效命令，客户端下刷无效状态，并且在其下一次 KeepAlive 调用中进行通知。修改操作当且仅当服务器确定所有客户端已经失效其本地缓冲后才可以继续 ,这种失效可能是由于客户端进行了失效通知，或者客户端允许缓冲租期过期失效

在缓冲失效过程中，主会将节点设置于不可缓冲状态，因此节点失效命令仅需要进行一轮。这种行为使得读操作总是不会被延迟，在读多写少的环境中会带来性能上的好处。另一种方式是对处于缓存失效过程中节点的所有访问都阻塞 ,这种方式不会导致某些客户端在此过程中由于没有 cache 而对主进行大量访问并导致偶尔的延迟。如果偶尔的延迟确实是问题，

那么当负载过大时应该使用一些复杂的策略

缓存的协议很简单：当变化发生时，失效相关数据而不是更新数据。更新数据可能比直接失效更加简单，但数据的更新协议会有效率上的问题：客户端访问过文件后会进行缓存，此后会获取关于此文件的更新信息，导致大量无效的更新操作

尽管强一致性导致开销增大，由于一个弱一致性的模型很难使用，我们还是采用了强一致性模型。同样，类似于虚拟同步这样需要客户端在所有消息中进行序列号交换的机制，在现存多种网络通信协议的环境下也被认为是不合适的

除了缓存数据和元数据，chubby 客户端还对打开的句柄进行了缓存。就是说，如果一个客户端打开一个已经打开过的文件，仅第一个 open()调用会引起一个到主的 rpc 调用。这个缓存被限制为一种辅助方式使用以确保它不会影响客户端的语义 :当临时文件的句柄被应用关闭后不能在另外的线程还保持打开状态；用做锁的句柄可以被重复使用，但不能被多个应用并发使用。后一个限制是因为客户端可能使用 Close()或 Poison()操作，而这两个操作会导致未完成的 Acquire()调用退出

chubby 协议允许客户端缓存锁，即可以持有锁比必要的持有时间更长。这是为了使这些锁在后续可以被同一客户端所重用。当另一个客户端希望持有锁时，会发生锁冲突事件，此时锁的保持者可以选择释放锁

Session 和 KeepAlives

chubby session 是 chubby cell 和客户端之间的关系，它由周期性的被称之为 keepalives 的握手所保持。除非 chubby 客户端对主进行了通知，否则在 session 有效期间，客户端的句柄，锁，缓存的数据等都有效。(当然，如下述，为了保持连接，session 连接协议可能会要求客户端失效其 cache)

客户端第一次初始化跟 chubby cell 的连接时会请求一个新的 session。在它关闭时，或者 session 空闲(一分钟内没有打开的文件也没有调用)时会被结束掉

每个 session 都有一个租期，在租期内主承诺不会主动断掉连接。我们称租期结束为租期超时。主可以对此超时进行延长，但可能无法及时将其回退

主通过三种场景下会提出租期超时时间：session 创建时，主发生故障转移时，主回复客户端的 KeepAlive 请求时。当接收到一个 KeepAlive 请求时，主通常会阻塞这个请求直至客户端之前的租期快要超时。之后主返回 KeepAlive 请求，并且指定一个新的租期时间。主可以任意增加超时时间长度。默认的超时时间是 12s，但负载过重的主可能会延长这个时间以减少 KeepAlives 请求数量。客户端接收到上一个 KeepAlives 回复后立刻初始化一个新的 KeepAlive 请求，即客户端会保证总是一个 KeepAlives 请求被阻塞在主上

除用来延长客户端租期外，KeepAlives 应答还用来传输事件，发送 cache 失效消息。当发生上述事件时，主会提前发出 KeepAlives 应答。KeepAlives 应答承载事件这种机制保证客户端不会在连接保持过程中丢失 cache 失效请求，同时也使得从客户端到主形成 chubby rpc 流。这种方式简化了客户端，并且使得此协议可以在配置了单向连接防火墙的环境下正常运行

客户端本身维护了一个 session 超时，此超时比主的租期超时时间要小，因为客户端必须考虑到网络交互及主请求的时间。为了保持一致性，我们要求主的处理时钟周期不能比客户端的处理周期的某个倍数更快

当客户端的本地租期到期时，它将无法确认是否主已经将 session 关闭。客户端此时会清除其 cache，并且设置 cache 为不可用状态，此时客户端处于 jeopardy 状态。客户端会等待一个默认 45 秒的时长，此时期被称为 grace period，如果在此期间客户端与主完成了一次成功的 KeepAlives 交互，则客户端再次启动 cache。否则，客户端会认为 session 过期。这样，

当 chubby cell 无法访问的时候客户端调用也不会永久被阻塞；如果 grace period 到期时还没有恢复连接，则 chubby API 会返回错误

当进入 grace period 时， chubby 库会通过 jeopardy 事件通知应用。如果通讯故障及时恢复，则应用会收到 safe 事件；如果 session 超时了，则会收到 expired 事件。这使得当应用不清楚当前 session 状态时可以静默，当故障排除时可以进行恢复而不是重启动。这对于启动很麻烦的服务有很大好处

当有操作在句柄 H 上由于 session 过期而失败时，后续除 Close()与 Poison()外的任何操作都会因为同样的原因失败。客户端可以利用这一点来保证网络或服务器的中断仅会导致部分而不是所有的后续请求丢失 ,因此可以通过最终的写操作来标记一系列复杂变化的最终状态

故障转移

当主故障或丢失主身份时，它会丢弃所有关于 session，句柄，锁的内存信息。由于仅主有权进行 session 租期的计时，因此新主选出之前对于 session 的计时暂停；因为这相当于延长客户端的租期，因此不会引起其他问题。如果选主过程迅速完成，则客户端可以在其本地租期过期前连接到新主。如果选主过程较长，则客户端丢失 cache，并且在寻找新主的过程中进入 grace period。grace period 允许 session 在发生故障转移时保持。

图 2 展示了一次主故障转移过程，在此过程中客户端将使用其 grace period 来保持其 session。时间从左到右增加，但并不等分。客户端租期以箭头显示。向上方向的箭头表示 KeepAlive 请求，向下的箭头则代表着回复。原始的主时间租期为 M1，客户端租期为 C1.当主在进入租期 M2 后通过 KeepAlive 的回复 2 通知客户端，客户端收到回复后进入租期 C2.当主回复下一个 KeepAlive 前死掉，又花费了一些时间进行选主。此过程中客户端的租期 C2

过期。客户端下刷 cache 并且开始进入 grace period

在这期间 ,客户端并不能确定自身的 session 是否已在主端过期。它并不会销毁 session，但会阻塞应用对它的访问，以避免访问到不一致的数据。在 grace period 的开始，chubby 库向应用发出 jeopardy 事件，以便应用在确定状态前使自身处于静默状态

新主选择终于成功。新主初始化一个接近但比其前辈更保守的租期 M3。来自客户端的第一个 KeepAlive 请求(4)会被拒绝 ,因为此请求中带有的 epoch 号不正确(此处后续会讲到)。重试的请求(6)成功了，但由于 M3 是保守的，故并不能使主的租期延长。然而应答(7)客户端延长其租期(C3)，并且可以通知应用其状态不再是 jeopardy。因为 grace period 足够覆盖租期 C2 结束至租期 C3 开始这段时间间隔，对客户端的影响仅仅是一点延迟。如果 grace period 比这段时间间隔还要短，则 session 可能被抛弃，应用会收到失败消息

当客户端连接到新的主时，客户端库和主向应用屏蔽此次错误。为了达成这一目标，新主必须重建其前辈的内存状态。其部分数据通过磁盘获得，部分通过客户端获得，还有部分通过保守的假设获得。数据库记录着所有 session，持有的锁，以及临时文件

新主将会访问如下内容：

1. 首先选择一个新的 epoch number，客户端的每次请求都需要提供。主会拒绝使用旧的 epoch number 的客户端请求，并且在应答中提供新的。这保证了即使新主与老主处于同一台机器，新主也不会回复一个非常老的发给老主的请求。
2. 新主可以响应主定位请求，但此时还不能进行 session 相关操作
3. 通过数据库记录，主重建包括 session，锁相关的内存信息。session 租期被设置为老主可能使用过的最长租期时间
4. 此时主可以响应客户端发起的 KeepAlive 请求，但还不能处理其他 session 操作
5. 主向每个 session 发起一个故障转移事件，客户端丢弃 cache，同时通知应用可能丢

失了事件

6. 主等待所有的 session 通知故障转移或延长其租期
7. 此时主可以处理所有请求
8. 当客户端使用故障转移前打开的句柄时，主会重新创建此句柄相关信息。如果重新创建的句柄被关闭，则主会标记此句柄，并且在当前的主中将不能够被重新创建，这保证了重复的网络包不会意外的重建一个已经被关闭的句柄。出错的客户端可能在将来使用新的 epoch number 会重建一个关闭的句柄，但这也仅仅会给已经出错的客户端带来问题
9. 一段时间后(例如 1 分钟)，客户端删除没有打开句柄的临时文件。客户端应该在此时间内对临时文件的句柄进行刷新。此机制带来的问题是当保留临时文件句柄的最后一个客户端丢失 session 时，此临时文件将不能被迅速删除

读者将会发现由于故障转移的代码极少跑到，这块会有较多的 bug

数据库实现

第一版 chubby 使用分布式的 Berkeley DB 做为其数据库。Berkeley DB 通过 B 树将字节串映射为字节流。我们插入了一个比较函数，它会首先对路径进行排序处理；这允许节点在保证其兄弟邻接保存的情况下通过其路径名被检索。由于 chubby 并不使用基于路径的权限，因此对每个文件的访问仅需要在数据库中进行一次查询即可

Berkeley DB 使用一个分布式一致性协议来在其多个副本中保存 log。当有主租期增加时，将会直接被广播到所有副本中

尽管 Berkeley DB 的 B 树代码被广泛使用，但其分布式代码是最近刚刚加入，使用者很少，而软件维护者必须投入大量精力去升级其最广泛使用的产品特性。当 Berkeley DB 的维

护者解决一个我们遇到的问题后，我们发现使用分布式这部分代码给我们带来了风险。因此，我们自己实现了一个预写日志和快照的简单数据库，类似于 Birrell 等。跟以前一样，数据库 log 使用分布式一致性协议被分发到所有副本上。chubby 只使用了 Berkeley DB 的少部分特性，因此我们的重写也使整个系统变得简单并且更像一个整体。比如，当我们需要一个原子操作时，不再需要生成一个事务了

备份

每几个小时 chubby cell 的主会写一个快照至异地的 GFS 文件服务器中。之所以使用异地服务器是为了避免建筑损毁的情况；另外，也可以避免循环依赖，因为同一机房的 GFS 很有可能会使用 chubby 进行选主

备份不仅仅可以用来做灾难恢复，当需要替换一个副本时，可以先从备份中对新副本进行初始化而不需要增加已经服务的副本的负载

镜像

chubby 允许将一个文件集从一个 cell 镜像到另一个 cell。由于文件都很小，并且有事件通知机制对文件更改，新增，删除进行通知，镜像的速度很快。假设没有网络问题，则 1 秒内变化就会影响到全球范围内的数十个镜像。当镜像不可达时，在连接恢复前会一直保持不变状态。恢复后通过对更新文件的校验来确保一致

镜像最常见的使用在保存分布在全球的计算集群的配置文件。有一个特殊的 chubby cell，叫做 global，包含/ls/global/master 子树，其中的数据会被镜像至子树/ls/cell/slave 中指定的其他 chubby cell。global cell 很特殊，它的 5 个副本分布在全球，因此它应该随时可以被访问

global cell 中镜像的文件包括 chubby 自身的访问控制列表，chubby cell 或其他系统广播其存在性的文件，或是保存着大数据集合位置信息数据，以及其他系统的配置文件