Google利器之Chubby
写完了Google Cluster，该轮到Chubby了。

参考文献：
[1] The Chubby lock service for loosely-coupled distributed systems
[2] Paxos Made Simple

声明文中大部分的观点来自于文献[1]中的描述，但也夹杂了部分本人自己的理解，所以不能保证本文的正确性。真想深入了解Chubby还是好好读原版论文吧：）

前言

MapReduce 很多人已经知道了，但关于Chubyy似乎熟悉它的就非常有限，这倒是不奇怪，因为MapReduce是一个针对开发人员的 ProgrammingModel，自然会有很多人去学习它，而Chubby更多的是一种为了实现MapReduce或者Bigtable而构建的内部的 工具，对于开发人员来说基本上是透明的。文献[1]我反复读了至少有两三天，但感觉也只是一个囫囵吞枣的结果，里面有很多工程实现上的细节，如果不是自己 亲自去设计或者实现，很难体会到其中的道理和奥妙。但是，对于这样一个分布式service的研究，还是让我对一个分布式系统的结构和设计思想有了更加直 观的感觉。

从distributed consensus problem说起distributed consensus problem(分布的一致性问题)是分布式算法中的一个经典问题。它的问题描述大概是这样的：在一个分布式系统中，有一组的Process，它们需要确 定一个Value。于是每个Process都提出了一个Value，consensus就是指只有其中的一个Value能够被选中作为最后确定的值，并且 当这个值被选出来以后，所有的Process都需要被通知到。
表面上看，这个问题很容易解决。比如设置一个server，所有的process都 向这个server提交一个Value，这个server可以通过一个简单的规则来挑选出一个Value（例如最先到达的Value被选中），然后由这个 server通知所有的Process。但是在分布式系统中，就会有各种的问题发生，例如，这个server崩溃了怎么办，所以我们可能需要有几台 server共同决定。还有，Process提交Value的时间都不一样，网络传输过程中由于延迟这些Value到达server的顺序也都没有保证。
为 了解决这个问题，有很多人提出了各种各样的Protocol，这些Protocol可以看做是一组需要遵循的规则，按照这些规则，这些Process就能 够选举出一个唯一的Value。其中，最有名的一个Protocol就是Paxos算法。（八卦一下，Paxos的提出者叫做Lamport，有很多分布 式的算法都是他提出的，他还是Latex的作者，大牛啊...）。想更加了解Paxos算法可以参考文献[2]，很漂亮的一篇文章。

那么 这些和Chubby有什么关系呢？其实Chubby就是为了这个问题而构建出来的。只是它并不是一个Protocol或者是一个算法，而是google精 心设计的一个service。这个service不仅能够解决一致性问题，还有其它的一些很实用的好处，会在下文慢慢介绍。

一个实例在Google File System(GFS)中，有很多的server，这些server需要选举其中的一台作为master server。这其实是一个很典型的consensus问题，Value就是master server的地址。GFS就是用Chubby来解决的这个问题，所有的server通过Chubby提供的通信协议到Chubby server上创建同一个文件，当然，最终只有一个server能够获准创建这个文件，这个server就成为了master，它会在这个文件中写入自己 的地址，这样其它的server通过读取这个文件就能知道被选出的master的地址。

Chubby是什么

从 上面的这个实例可以看出，Chubby首先是一个分布式的文件系统。Chubby能够提供机制使得client可以在Chubby service上创建文件和执行一些文件的基本操作。说它是分布式的文件系统，是因为一个Chubby cell是一个分布式的系统，一般包含了5台机器，整个文件系统是部署在这5台机器上的。

但是，从更高一点的语义层面上，Chubby是一个 lock service，一个针对松耦合的分布式系统的lock service。所谓lock service，就是这个service能够提供开发人员经常用的“锁”，“解锁”功能。通过Chubby，一个分布式系统中的上千个client都能够 对于某项资源进行“加锁”，“解锁”。

那么，Chubby是怎样实现这样的“锁”功能的？就是通过文件。Chubby中的“锁”就是文件，在上例 中，创建文件其实就是进行“加锁”操作，创建文件成功的那个server其实就是抢占到了“锁”。用户通过打开、关闭和读取文件，获取共享锁或者独占锁；并且通过通信机制，向用户发送更新信息。

综上所述，Chubby是一个lock service，通过这个lock service可以解决分布式中的一致性问题，而这个lock service的实现是一个分布式的文件系统。

可能会有人问，为什么不是直接实现一个类似于Paxos算法这样的Protocol来解决一致性问题，而是要通过一个lock service来解决？文献[1]中提到，用lock service这种方式有几个好处：

1. 大部分开发人员在开始开发service的时候都不会考虑到这种一致性的问题，所以一开始都不会使用consensus protocol。只有当service慢慢成熟以后，才开始认真对待这个问题。采用lock service可以使得在保持原有的程序架构和通信机制的情况下，通过添加简单的语句就可以解决一致性问题；
2. 正如上文实例中所展现，很多时候并不仅仅是选举出一个master，还需要将这个master的地址告诉其它人或者保存某个信息，这种时候，使用 Chubby中的文件，不仅仅是提供锁功能，还能在文件中记录下有用的信息（比如master的地址）。所以，很多的开发人员通过使用Chubby来保存 metadata和configuration。
3. 一个基于锁的开发接口更容易被开发人员所熟悉。并不是所有的开发人员都了解consensus protocol的，但大部分人应该都用过锁。
4. 一个consensus protocol一般来说需要使用到好几台副本来保证HA（详见Paxos算法），而使用Chubby，就算只有一个client也能用。

可以看出，之所以用lock service这样的形式，是因为Chubby不仅仅想解决一致性问题，还可以提供更多更有用的功能。事实上，Google有很多开发人员将Chubby当做

name service使用，效果非常好。

关于lock service，还有两个名词需要提及。
一 个是advisory lock。Chubby中的lock都是advisory lock。所谓的advisory lock，举个例子，就是说当有人将某个文件锁住以后，如果有其他的人想不解锁而直接访问这个文件，这种行为是不会被阻止的。和advisory lock对应的是mandatory lock，即如果某个文件被锁住以后，如果有其他的人直接访问它，那么这种行为是会产生exception的。
另 一个是coarse-grained（粗颗粒度的）。Chubby的lock service是coarse-grained，就是说Chubby中的lock一般锁住的时间都比较长，可能是几小时或者几天。与之对应的是 fined-grained，这种lock一般只维持几秒或者更少。这两种锁在实现的时候是会有很多不同的考虑的，比如coarse-grained的 lock service的负载要小很多，因为加锁解锁并不会太频繁。其它的差别详见文献[1]。

Chubby的架构

上图就是Chubby的系统架构。
基本上分为了两部分：服务器一端，称为Chubby cell；client一端，每个Chubby的client都有一个Chubby library。这两部分通过RPC进行通信。
client端通过Chubby library的接口调用，在Chubby cell上创建文件来获得相应的锁的功能。
由于整个Chubby系统比较复杂，且细节很多，我个人又将整个系统分为了三个部分：
Chubby cell的一致性部分
分布式文件系统部分
client与Chubby cell的通信和连接部分

先从Chubby cell的一致性部分说起。
一般来说，一个Chubby cell由五台server组成，可以支持一整个数据中心的上万台机器的lock service。
cell中的每台server我们称之为replicas（副本）。
当 Chubby工作的时候，首先它需要从这些replicas中选举出一个master。注意，这其实也是一个distributed consensus problem，也就是说Chubby也存在着分布式的一致性问题。Chubby是通过采用consensus protocol（很可能就是Paxos算法）来解决这个问题的。所以，Chubby的client用Chubby提供的lock service来解决一致性问题，而Chubby系统内部的一致性问题则是用consensus protocol解决的。
每个master都具有一定的期限，成为master lease。在这个期限中，副本们不会再选举一个其它的master。
为 了安全性和容错的考虑，所有的replicas（包括master）都维护的同一个DB的拷贝。但是，只有master能够接受client提交的操作对 DB进行读和写，而其它的replicas只是和master进行通信来update它们各自的DB。所以，一旦一个master被选举出来后，所有的 client端都之和master进行通信（如图所示），如果是读操作，那么master一台机

器就搞定了，如果是写操作，master会通知其它的 replicas进行update。这样的话，一旦master意外停机，那么其它的replicas也能够很快的选举出另外一个master。

再说说Chubby的文件系统
前 文说过，Chubby的底层实现其实就是一个分布式的文件系统。这个文件系统的接口是类似于Unix系统的。例如，对于文件名“/ls/foo /wombat/pouch”，ls表示的是“lock service”，foo表示的是某个Chubby cell的名字，wombat/pouch则是这个cell上的某个文件目录或者文件名。如果一个client端使用Chubby library来创建这样一个文件名，那么这样一个文件就会在Chubby cell上被创建。
Chubby的文件系统由于它的特殊用途做了很多 的简化。例如它不支持文件的转移，不记录文件最后访问时间等等。整个文件系统只包含有文件和目录，统一称为“Node”。文件系统采用Berkeley DB来保存Node的信息，主要是一种map的关系。Key就是Node的名字，Value就是Node的内容。
还有一点需要提及的 是，Chubby cell和client之间用了event形式的通知机制。client在创建了文件之后会得到一个handle，并且还可以订阅一系列的event，例 如文件内容修改的event。这样的话，一旦client相关的文件内容被修改了，那么cell会通过机制发送一个event来告诉client该文件被 修改了。

最后谈谈client与cell的交互部分
这里大致包含两部分的内容：cache的同步机制和KeepAlive握手协议。
为 了降低client和cell之间通信的压力和频率，client在本地会保存一个和自己相关的Chubby文件的cache。例如如果client通过 Chubby library在cell上创建了一个文件，那么在client本地，也会有一个相同的文件在cache中创建，这个cache中的文件的内容和cell 上文件的内容是一样的。这样的话，client如果想访问这个文件，就可以直接访问本地的cache而不通过网络去访问cell。
cache 有两个状态，有效和无效。当 有一个 client 要改变某个 File 的时候，整个修改会被 master block，然后 master 会发送无效标志给所有 cache 了这个数据的 client（它维护了这么一个表），当其它 client 端收到这个无效标志 后，就会将 cache 中的状态置为无效，然后返回一个 acknowledge；当 master 确定收到了所有的 acknowledge 之后，才完成整个 modification。需要注意的是，master 并不是发送 update 给 client 而是发送无效标志给 client。这是因为如果发送 update 给 client，那么每 一次数据的修改都需要发送一大堆的 update，而发送无效标示的话，对一个数据的很多次修改只需要发送一个无效标示，这样大大降低了通信量。至于 KeepAlive 协议，则是为了保证 client 和 master 随时都保持着联系。client 和 master 每隔一段时间就会 KeepAlive 一次，这样的话，如果 master 意外停机，client 可以很快的知道这个消息，然后迅速的转移到新的 master 上。并且，这种转移对于 client 端的 application 是透明的，也就是说 application 并不会知道 master 发生了错误。关于 cache 和 KeepAlive 还有很多的 细节，想了解的读文献[1]吧。总结其实在我的这篇文章中，还有一个很大的主题没有提及，那就是 Chubby 的容错机制。基本上，容错这个思想贯穿了文献[1]的始终，也正是因此，我很难将 它单独提取出来解释，因为它散落在了 Chubby 系统设计的所有角落。我个人感觉，容错是一个分布式系统设计的核心思想，在设计的时候要求考虑到所有

可能 会发生的错误，不仅仅包括了硬件的错误，网络的故障，还包括了开发人员可能出现的错误。我想，这是我读这篇文章[1]最大的收获。