Slacker: Fast Distribution With Lazy Docker Container

张桐

2017 年 12 月 8 日

摘要

容器化的应用现在越来越受到欢迎，但是不幸的是，目前部署容器的方法非常的慢。我们开发了一个全新的容器benchmark（HelloBench）来测量57种不同的容器化应用的启动时间。我们使用HelloBench来细致地分析工作负载，研究启动过程和容器镜像压缩过程中展现出的块I/O的模式。我们的分析展示出，启动容器所花时间有76%是用来做pulling，但是pulling下来的数据中，只有6.4%被读取了。我们使用这和其他的发现来指导设计出了Slacker，一个全新的，启动优化的Docker存储驱动。Slaker是基于中心化的，被Docker workers和registries共享的存储。Workers能够经过backend clone和通过lazy fetching容器数据来进行快速启动和确定容器的存储。Slacker能够提高中间容器的部署速度20倍，提高总体的部署速度5倍。

1 Introduction

隔离是一种在云计算或者其他多租户平台中高度需求的属性。如果没有隔离机制，用户会面临性能的不稳定，可能存在的服务崩溃，还有严重的隐私泄露的威胁。

Hypervisors，或者说虚拟机监控器（VMM）已经为应用提供了传统的隔离机制，每个应用都被部署到它们各自的虚拟机中，每个虚拟机为它们提供各自的环境和资源。不幸的是，hypervisors需要做出很多的特权级的操作，并且使用的是一些间接的技术来推断资源的使用情况。这就导致了hypervisors通常都很笨重，启动时间也很长，运行时的开销也非常大。

随着Docker越来越受到欢迎，容器最近是作为一种轻量级的备择方案来替代基于hypervisor的虚拟化。在一个容器内部，所有的处理资源都被操

作系统虚拟化了，包括网络端口和文件系统挂载点。容器本质上就是进程，它用虚拟化的方式使用所有资源，不仅仅是CPU与内存。正如我们以上提到的，不存在一些固有的因素导致容器的启动速度比正常的进程启动速度慢。

但不幸的是，由于文件系统配置的瓶颈，启动容器要比启动进程慢得多。在启动容器时，初始化网络，计算，和内存资源都是很快的。一个容器化的应用需要一个完全初始化的文件系统，包含应用二进制，完整的Linux发行版，和包的依赖。在Docker或者Google Borg的集群中部署一个容器一般需要显著的复制和安装开销。

如果启动时间能够被优化，那么就会有大量的优势：面对突然增大的访问量应用能够立即的扩展，集群调度可以在低代价的情况下频繁地进行负载均衡，当严重漏洞被修复后，更新地软件能迅速的进行部署，开发者也能更好地构建和测试已经分发地应用。

我们采用了分两步的方法来解决容器启动的问题。第一，我们开发了一个新的开源的Docker benchmark，它会在容器启动的时候进行检测并记录。HelloBench基于57种不同的容器工作负载，确定容器从开始部署到能正常使用之间所耗费的时间。我们使用HelloBench和静态分析来表示Docker镜像和I/O模式的特征。我们的分析表明了（1）复制包数据耗费了76%的启动时间，（2）被复制的数据中只有6.4%是开始工作所实际需要的，（3）简单地对镜像进行block级的去重相比于使用gzip压缩单独的镜像能够有更好的压缩率。

第二，我们使用我们的发现构建了Slacker，一个新的Docker存储驱动。它通过在stack中多个层次利用特殊的存储系统支持来完成快速分发。特殊地，Slacker使用我们后端服务器的快照和克隆来极大的减小Docker基本操作的开销。Slacker是使用lazily pull必要的数据，而不是预先下载整个容器的镜像，这能显著的减小网络I/O所花费的时间。我们还对Linux的内核进行修改来提升cache sharing，这也能提高Slacker的效率。

使用这些技术的结果就是极大的提升了Docker基本操作的性能。镜像的push能够比以前快153倍，pull能比以前快72倍。基本的Docker操作用例能够在由于这个特性受益。举例来说，Slacker能够加快5倍中间容器的部署周期，20倍的整个部署周期。

我们也构建了MultiMake，一个新的基于容器的构建工具来展示Slacker快速启动的优势。MultiMake展示了从相同源代码生成16种不同的二进制文

件，不同的文件由不同的容器化的GCC生成。在使用Slacker的情况下，MultiMake被加速了10倍。

2 Docker Backgroud

我们现在描述Docker框架，存储接口和默认存储驱动。

2.1 Version Control for Container

Linux总是利用虚拟化来隔离内存，cgroup通过提供6个新的名字空间，虚拟了一个资源的边界。名字空间包括文件系统挂载点，IPC队列，网络，主机名字，进程ID号和用户ID。Linux的cgroups是在2007年提出来的，但是真正火起来是2013年提出新的容器管理工具Docker之后。当使用Docker时，使用一条命令“docker run -it ubuntu bash”将会从Internet中pull下Ubuntu的包，并初始化一个文件系统来进行新的Ubuntu的安装，然后执行必要的cgroup的配置启动，最后运行交互式的bash会话。

这个样例命令有几个部分，第一，“ubuntu”是一个镜像的名字。镜像是文件系统的只读拷贝，典型的包括有应用的二进制，Linux发行版，还有应用依赖的其他包。在Docker镜像中绑定应用会很方便，我们可以在配置文件中指定该应用依赖的软件包，当运行时会自动下载安装。第二，“run”是我们对镜像的一个操作，run操作会创建一个初始化的，基于镜像的根文件系统，这个根文件系统是给这个新容器使用的。其他的操作包括“push”（发布新的镜像）和“pull”（从中心化的存储获取发布的镜像），如果本地没有选中的镜像的话，镜像会被自动的pull下来。第三，“bash”是我们指定的在容器内启动的程序，用户可以指定容器内可用的任何程序。

Docker管理镜像数据的方式与传统的版本控制系统很像。使用这个模型是因为以下两点。第一，对于同一个镜像可能有不同的分支。第二，镜像是很自然的从其他镜像中构建。距离来说，Ruby on Rails镜像就是基于Rails镜像构建，Rails又是基于Debian构建的。每个镜像都表示了，在以前的commit之上有一个新的commit，而且可能还有很多commit没有被标记为可运行的镜像。当一个容器启动时，它从一个被commit的镜像开始，但是文件可能被修改过。用版本控制的说法，这些修改是指unstaged 修改。Docker的commit操作将一个容器和对它的修改转化为一个只读的镜像，

layer指的时commit的数据或者对容器的unstaged修改。

Docker的work机器运行一个本地的Docker守护进程。用户通过发送命令到守护进程来创建新的容器和镜像。镜像的分享一般是在中性化的registries上，镜像一般是通过push指令来从守护进程发送到集群的registries中，通过执行pull可以从registries获取镜像然后执行。只是Layer在传输完后，在接收端目前还不可用。Layer是一种使用gzip压缩的归档文件，存储在registries中并在网络中进行传输。

2.2 Storage Driver Interface

Docker容器对存储的访问有两种方式。第一，用户可能要将主机上的目录挂载到容器中。举例来讲，一个用户使用容器化的编译器时，他可能要将源代码的目录挂载到容器中，这样编译器就能访问源代码文件然后创建二进制文件。第二，容器需要访问Docker的layer，来表示应用二进制与函数库。通过将挂载点作为容器的根目录，Docker可以表示出应用数据的视图。容器存储和挂载是被Docker存储驱动管理的，不同的驱动可能选择不同的方法表现layer的数据。但是驱动必须实现这几个方法Get(id)=dir，Put(id)，Create(parent, id)，Diff(parent, id)=tar，ApplyDiff(id, tar)，所有这些方法参数都有一个id来指定layer。

Get函数请求驱动挂载layer然后返回挂载点的路径，挂载点返回的并不只是id指定的layer的路径，还包括它所有祖先的路径（在id指定的layer路径的父目录下的所有文件应该能被看到）。Put将一个layer进行umount。Create从上级layer中copy数据然后创建一个新的layer，如果上级是NULL，那么创建的新layer就是空的。Docker使用create用来（1）划分新容器的文件系统，（2）从pull获取数据之后分配layer。

Diff和ApplyDiff是分别在Docker的push操作和pull操作中执行的。当Docker正在pushing一个layer的时候，Diff将本地layer的进行压缩打包为tar文件。ApplyDiff刚好相反。

2.3 AUFS Driver Implementation

AUFS存储驱动是Docker发行版默认的存储驱动。驱动是基于AUFS文件系统（Another Union File System），Union file system 不会直接在磁盘上存储数据，它会使用底层其他的文件系统。一个Union的挂载点提供了底层文件系统多个目录的视图。AUFS维护一个列表，是被挂载的底层文件系

统目录路径。在路径解析的过程中，AUFS迭代搜索这个列表，包含这个路径的第一个目录会被选中并使用它的inode。AUFS支持特殊的whiteout文件来标记那些已经在底层layer中被删除的文件。AUFS同时支持在文件粒度下的COW（copy on write）。在写操作进行之前，在底层layer中的文件会被复制到顶层layer中。

AUFS的驱动利用了AUFS文件系统分层和COW的优点来直接访问AUFS之下的文件系统。驱动在底层文件系统中为每一个存储的layer创建一个目录。ApplyDiff操作就是简单的将tar文件解压到layer对应的目录。当接受到一个Get时，驱动使用AUFS来创建layer和它祖先的一个union视图。当Create操作执行时，驱动使用AUFS的COW技术来有效的复制layer的数据，不幸的是，在文件粒度层面进行的COW会造成一些其他的问题。

3 HelloBench

我们提出了HelloBench，来评估Docker启动的速度。HelloBench直接执行Docker的独立命令例如push，pull，和run。它由两部分组成：（1）container的镜像的集合和（2）一个在上述container中执行简单任务的测试套件。我们选择的镜像是可以在最小配置下独立运行的，不会依赖于其他container。例如WordPress没有是因为它依赖于独立的MySQL的container。

HelloBench套件用一下的方式来确定容器的启动时间，运行一个最简单的任务或者等到容器自己来报告准备完成。对于编程语言的容器，典型的任务就是编译或解释一段简单的“hello world”程序。Linux发行版的镜像执行一段非常简单的shell命令，典型就是“echo hello”。对于长时间运行的服务，典型包括部分数据库和web server。HelloBench等到这些容器写一个“up and ready”或者类似地信息时来确定时间。对于某些不会输出地服务，就一直对它进行访问直到它有一个response。

4 Workload analysis

在这个section中我们分析了HelloBench工作负载的行为和性能，然后问了4个问题：容器镜像有多大？有多少数据是在执行的时候是必要的？push，pull，和运行镜像所需的时间是多少？镜像数据是如何在layer中分发的？我们能通过性能得到什么启示？不同的运行有没有什么相似的访问

模式？

4.1 Image Data

我们首先研究使用HelloBench在Docker Hub中pull下来的镜像来开始我们的分析，对于每一个镜像我们设置三个必要属性，它压缩后的大小，未压缩的大小和当HelloBench执行时所读取的字节数。我们通过使用blktrace来跟踪工作负载在块设备上的读取过程。Figure 5展示了这三个属性的分布(CDF)，我们观察到读取数据的中位数是20MB，但是镜像压缩后的大小中位数是117MB，未压缩的大小的中位数是329MB。

我们按照不同的种类将读取的大小进行了分类。最大的相对浪费是Linux发行版的负载（30倍的压缩大小比上85倍的未压缩大小），但是绝对的最大浪费是language和web框架。对于所有的镜像来说，平均只有27MB被读取，平均未压缩大小是读取的15倍，所以大概只有6.4%的镜像数据是在启动的时候所必要的。

虽然Docker的镜像在进行gzip压缩的时候会变得更小，但是这个格式在已经启动的容器中无法修改数据，所以不怎么合适。于是worker典型地都是将镜像文件解压存储在磁盘上，这就意味着压缩仅仅减小了网络的I/O而没有减小磁盘的I/O。相对于压缩而言，去重是一个简单的备择方案并且满足更新的需求。我们扫描了HelloBench的镜像来计算在文件块中去重方案的有效性。Figure 7比较了gzip的压缩率和去重，在文件粒度和数据块（4KB）的粒度。我们发现gzip的压缩率是在2.3到2.7之间，对每一个镜像进行单独的去重的表现很差，只能对全局进行去重。然而，对所有的全局的文件或者数据块进行去重可以有2.6和2.8。

推断：在执行的过程中读取的数据比整个镜像的大小要小很多，不管是压缩过的还是未经压缩的。镜像数据在网络中进行传输的时候是压缩的，但是在本地存储进行读取或者写入的时候未经压缩的，所以对于网络和存储其实开销都很大。有一个减小开销的方法就是构建更加轻量级的镜像，减少安装的包的数量。可选的是我们可以对镜像数据进行缓pull操作，只有当我们需要的时候才会pull它。我们同时可以看到全局的基于数据块的去重操作也同样是一个有效的方法。

4.2 Operation Performance

一旦构建完毕，容器化的应用就会按以下的步骤进行部署：开发者

将应用的镜像push到中心化的registry中，worker就会从registry中将镜像进行pull下来，然后run每个镜像。我们将这些操作带来的延迟用HelloBench进行记录，然后放在figure 8中。push，pull和run的时间的中位数分别是61，16和0.97秒。

我们在Figure 9中将时间根据不同的工作负载进行了分类。所有类别的模式都是通用的，run很快但是pull和push都很慢。Run在distro和language类别中是最快的，分别是0.36和1.9秒。push，pull和run的平均时间分别是72，20，6.1秒。所以大概有76%的启动时间被花费在从远程的registry中将镜像pul下来的过程中。l

push和pull都很慢，我们想知道这些操作是不是仅仅只是高延时，或者说在多操作并发的情况下是否同样开销很大。为了研究可扩展性，我们并发push和pull多种个数量，多种大小的自己生成的镜像。每个镜像包括一个随机生成大小的文件。我们没有使用HelloBench的镜像使用自己生成的镜像是因为我们可以确定生成镜像的大小。Figure 10表明了总的时间随着镜像的数量和镜像的大小成线性增加。所以push和pull不仅仅是高延迟，他们还消耗网络和磁盘资源，限制了可扩展性。

得出结论：容器的启动时间主要是pull操作共享的，在一个全新的部署中大概有有76%的时间是花费在pull操作上。使用push来发布镜像对迭代开发的程序员来说也是非常痛苦的，因为他们要不断的地将自己做的小更改push到registry中。大多数push的工作是由存储驱动的Diff函数完成的，大多数pull工作是由ApplyDiff完成的。优化存储驱动的这些函数应该能够显著的提高发布的性能。

4.3 Layers

镜像数据典型地被划分为几个layer。AUFS驱动在运行时将镜像的layers进行组合为容器提供文件系统完整的视图。我们首先分析layered文件系统的两个性能问题：查找深层次layer和对不在顶层layer的small写。

首先我们create了16个layer然后使用AUFS进行compose，每个layer包含1K个空文件。然后在没有提前进行cache的情况下我们在每个layer随机打开10个文件来计量工作延迟。Figure 11a展示了这个的结果：在layer深度和延迟之间有一很强的关联关系。然后我们create了两个layer，底部的layer包含了不同大小的大文件。我们计量了向底部layer中的文件增加一个字节的延迟。在Figure 11b中的结果显示，这些small write的延迟对应的是文件

的大小，而不是写入的大小，这是因为AUFS是在文件粒度上进行COW。在文件修改之前它会被复制到顶层的layer中，所以写一个字节可能会花费20秒。幸运的是对一个layer进行small write对于容器来说是一次花费，之后再对这个大文件继续操作是不会额外复制该文件的。

我们之前已经考虑了layer的深度与性能的关系，我们现在考虑，一般来说数据在HelloBench的镜像中存储的有多深？Figure 12展示了全部数据（文件数量，目录数量，字节大小）存储在每个layer的百分比。这三个度量值具有很密切的联系，基本在分布上很严格的对应。有一些数据到了28层，但是多数数据都是集中在比较前的层次中，几乎一半的数据都是在9层以下。

我们现在来考察数据在多个layer之间分布的方差，来估计对于每个镜像来说有多少数据是存储在最顶层的layer，多少是存在最底层的layer，哪个layer存储的数据量是最多的。Figure 13展示了该数据的分布情况，对于79%的镜像来讲，最顶层的layer并存储了0%数据的，相对来讲大概有27%的数据是存储在最底层的layer中。大部分的数据典型的是在某单个的layer中存储的。

推导：对于layered文件系统，数据存储在越深的layer中，访问就会变得越慢。不幸的是Docker的镜像是倾向于变得更深的，我们也发现至少有一半的数据是存储在第9层或者更深的层次。将多层layer展开是一个避免出现性能问题的技术，然而flattening可能会潜在的需求额外的拷贝同时减小了其他由COW带来的受益。

4.4 Caching

我们现在来考虑这一种情景：同样的worker将相同的镜像运行多次。我们想知道是否第一次的I/O会产生一个Cache来避免相同I/O序列的重复运行。最后我们运行HelloBench每个工作负载两次，收集每次的时间信息。我们计算了第一次读的时候的那部分来判断他们是否从Cache中受益。

Figure 14展示了第二次运行时的读写性能。读被划分为hit和miss，对于给定的块，我们只记录了第一次的访问。对于所有的工作负载，读/写的比率是88/12。对于ditro，database，还有language的工作负载几乎完全都是读请求。对于读请求大概99%的都能从之前运行的cache中获益。

推导：我们对于同一个镜像运行多次时，读取的数据大多数都是相同的，当相同的镜像在相同的机器上运行时Cache是非常有用的。在很多容器

化的应用组成的大型集群里，一台机器上很可能不会多次重复执行一个应用，除非容器的位置是严格确定的。所以其他的一些特性（负载均衡和容错机制）可能会导致Cache的失效。然而在一些小型集群的一些容器化应用（python或者gcc）中，重复执行可能会存在。在后者的条件下，集群的性能可能会从Cache共享中受益。

5 Slacker

在这一节我们描述了Slacker，一个新的Docker存储驱动。我们的设计是基于对容器负载的分析和以下的五个目标：（1）使得push和pull变得很快，（2）不会减慢长时间运行的容器，（3）尽可能的重复利用存在的存储系统，（4）使用最新存储驱动提供的primitive，（5）除了存储驱动插件不会对Docker registry和Docker 守护进程进行任何修改。

Figure 15说明了运行Slacker的Docker集群的架构。设计是基于一个中性化的NFS存储，它被所有Docker守护进程和registries共享。容器内的大多数数据是在执行启动容器时不需要的，所以Docker的worker仅仅是在需要的时候才在共享存储中获取数据（Lazy）。我们使用的是Tinri VMstore server来作为NFS存储。Docker的是由VMstore的只读镜像表示的，Registries现在也不再是作为layer数据的主机，只是作为一个name server来关联镜像元数据和对应的快照。Push和pull不再造成大的网络传输，相反他们现在仅仅会共享快照的ID。Slacker使用VMstore的snapshot将一个容器转化为一个可共享的镜像，然后指定快照ID，使用clone来将容器存储从registries中pull下来。在内部VMstore将会使用块级的COW来有效实现snapshot和clone。

Slacker的设计是基于我们对容器工作负载的研究，下列的四个subsection对应了我们之前提到的分析subsection。我们同时还总结了如何让在不使用Slacker的时候对Docker框架进行修改来提高性能。

5.1 Storage Layers

我们之前的分析已经揭示了，在总共pull下来的数据中，平均只有6.4%是在容器启动时必要的。为了避免不被使用的数据浪费I/O资源，Slacker将所有的数据存储在中性化的NFS server上，被所有的worker共享。只有当需要的时候worker才会获取数据（Lazy）。Figure 16说明了这个设计：每个容器的存储表示为单个的NFS文件。Linux的loopbacks将每个NFS文件看作

是一个虚拟块设备，这就可以将它挂载成容器的根文件系统。Slacker将每个NFS文件格式化为Ext4文件系统。

Figure 16b 将Slacker 的结构和AUFS的结构做了一个比较。虽然两者都将Ext 4作为了一个key 文件系统，但是这里有三个重要的不同点。第一，ext4是Slacker网络磁盘的后端，而在AUFS中它是一个本地的磁盘。于是Slacker能够lazily的从网络获取数据，但是AUFS必须在容器启动之前复制全部的数据到本地磁盘。

第二，AUFS是在ext4之上使用文件粒度的COW，所以这个容易受到layered文件系统的性能影响。相对的，Slacker在文件级别是非常flatten的，也就是没有那么多层次的影响。然而，Slacker仍然利用VMstore实现的块级的COW获得性能的提升。不仅这样，VMstore还可以在内部进行块级的去重操作，为运行容器的Docker worker提供进一步的空间节省。

第三，AUFS是用一个ext4实例上的不同目录来用作容器的存储，对于Slacker则是每个容器对应不同的ext4实例。这个不同点表示了一个有意思的权衡：每个ext4实例有他自己的日志。在AUFS的作用下，所有的容器共享着一个日志，提供了极大的有效性。但是我们知道共享日志可能会导致优先级倒置的问题，这可能会破坏多租户平台的QoS保证的特性。Internal fragmentation是另一个潜在的问题，当NFS的存储被划分为很多小的，none-full的ext4实例的时候可能引发这个问题。幸运的是，VMstore的文件存储是稀疏的，所以Slacker并不会受到这个问题影响。

5.2 VMstore Integration

在前面我们发现Docker的push操作和pull操作是相比于run非常慢。Run启动的非常快是因为一个新容器的存储是由镜像初始化的，而且这个操作由于AUFS提供的COW所以完成的很快。相对来讲，push和pull要与其他的机器传输数据，需要复制大量的layer所以会很慢。由于Slacker是基于共享存储进行构建的，所以我们也可以对于pull和push进行COW的分享。

幸运的是，VMstore使用基于REST的API扩展了它的基础NFS接口，包括两个函数snapshot和clone。snapshot是从一个NFS文件中创建一个只读的快照，clone是从一个快照中创建一个NFS文件。NFS的名字空间中不会出现快照，只有独一的ID来表示快照。文件级别的快照和克隆是构建有效的日志，去重和其他公共存储操作的primitives。我们在Slacker中使用snapshot和clone分别来实现Diff和ApplyDiff操作。同时这个驱动的函数

表示为push和pull操作。

Figure 17展示了运行Slacker的守护进程是如何与VMstore和Docker registries交互的。Figure 17a 是push/Diff操作，17b是pull/ApplyDiff操作。对于push操作，Slacker要求VMstore来创建一个NFS的快照来表示layer。VMstore创建快照之后返回一个ID，例如“212”。Slacker将这个ID内嵌进一个tar文件（为了兼容性）然后将它发送到registry。pull操作正好相反。Slacker的实现很快是因为：（a）layer数据没有经过压缩或者解压操作。（b）layer数据从来没有离开过VMstore，只有元数据才在网络上传输。

Diff和ApplyDiff这两个名字在Slacker的实现中命名可能有些不准。尤其是Diff(A, B)是用来返回和其他守护进程不同的一个增量的，能够在A的基础上重构出B。在Slacker的情况下，layer已经在名字空间级别被有效的flatten了，Diff（A, B）并不是返回一个增量，而是返回一个其他worker的引用，通过这个引用我们可以得到B的一个clone，有没有A都没有关系。

Slacker和其他没用运行Slacker的守护进程是部分兼容的，当Slacker pull一个tar的时候，它首先检测tar包的前几个字节，如果tar包含了layer文件（而不是内嵌的snapshot）就解压，和未修改的操作一样。但是其它类似AUFS无法处理内嵌snapshot的tar文件。

5.3 Optimizing Snapshot and Clone

镜像通常由多个layer组成，一半的镜像将数据存储在9层之后。同时对于这种类型的数据，块级的COW比文件记得COW性能要强得多。然而底层的数据还是为Slacker带来了挑战。前面讨论过，Slacker的layer已经flatten了，所以挂载任何一个layer都会提供一个完整的文件系统的视图，然后就可以被容器使用。不幸的是Docker的框架没有flatten layer的概念。当Docker pull一个镜像的时候，它获取了所有的layers，将每一个layer使用ApplyDiff传给驱动。对于Slacker来说，最顶端的layer已经足够了，但是对于最深的镜像，额外的克隆操作花费很大。

我们目标之一就是在已经有的框架中进行工作，所以我们并没有修改整个框架来消除不必要的驱动调用，我们是使用lazy cloning进行优化。我们发现在pull阶段主要的耗时并不是快照tar文件的网络传输，而是VMstore的clone。虽然clone并没有花费多少时间，但是执行28次还是有一些明显的延迟。于是Slacker并没有把每个layer表示为一个NFS文件，而是将他们表示为本地的元数据，然后记录下快照ID。ApplyDiff只是设置以下元数据而不是立即

进行clone。如过Docker使用get命令时才对它进行实际的clone操作。

我们同样使用snapshot-ID元数据做snapshot caching。Slacker实现了Create，就对layer做了一个逻辑的copy。如果多个容器从相同的镜像创建，Create就会被调用很多次，但是Slacker仅仅只在第一次做一次，之后就重用快照ID。