调研报告

马凯, 金孜达, 刘时, 徐亦尧, 李文睿

2018 年 4 月 9 日

目录

1 项目成员

马凯、金孜达、刘时、徐亦尧、李文睿。

2 项目简介

该项目提出一种可靠、实时、可伸缩、可扩展（可裁剪）、资源消耗低、面向流数据优化且能与网络集成的嵌入式文件系统，因此适用于多种物联网应用场景，尤其适用于资源有限的无线传感器网络的数据采集工作。

该文件系统的预期目标如下：

1. 可靠性：在硬件故障情况下仍可正常工作，不会导致系统崩溃；
2. 低资源消耗：适用于资源有限的设备；
3. 可伸缩：插入新的闪存卡后将立刻为系统所用；
4. 可扩展、可裁剪：根据自身需求，可配置是否需要某些功能；
5. 面向流数据优化：适合存储和传输传感器抓取数据得到的流式数据；
6. 网络集成：将数据写入相应文件即相当于将数据发送给服务器，毋须手动进行网络操作。并可在网络失去连接时自动在本地存储进行数据缓冲。

本项目提出这样的问题：物联网需要怎样的文件系统？并将给出一个初步的答案。

3 项目背景

3.1 物联网技术

3.1.1 物联网简介

物联网（Internet of Things）是物理设备、交通工具、家用电器及内嵌电子设备、软件、传感器、执行器和互联通性的设备的网络。每个东西都可唯一地通过其嵌入式计算系统识别，但也可以通过现有的互联网架构进行互操作。

图 1: 物联网示意图

传统互联网的信息来源严重依赖于人的输入，而人的精力有限，虽然物质世界的信息极为丰富，但人类无法尽数采取。因此物联网技术将物质与资讯连接，使得物品可以自主地采集信息并作出行动，依据信息进一步地反作用于物质世界。

在物联网上，每个人都可以应用电子标签将真实的物体上网联结，在物联网上都可以查出它们的具体位置。通过物联网可以用中心计算机对机器、设备、人员进行集中管理、控制，也可以对家庭设备、汽车进行遥控，以及搜寻位置、防止物品被盗等，类似自动化操控系统，同时透过收集这些小事的数据，最后可以聚集成大数据，包含重新设计道路以减少车祸、都市更新、灾害预测与犯罪防治、流行病控制等等社会的重大改变。

3.1.2 物联网技术前景

物联网将现实世界数位化，应用范围十分广泛。物联网拉近分散的资讯，统整物与物的数位资讯，物联网的应用领域主要包括以下方面：运输和物流领域、健康医疗领域范围、智慧环境（家庭、办公、工厂）领域、个人

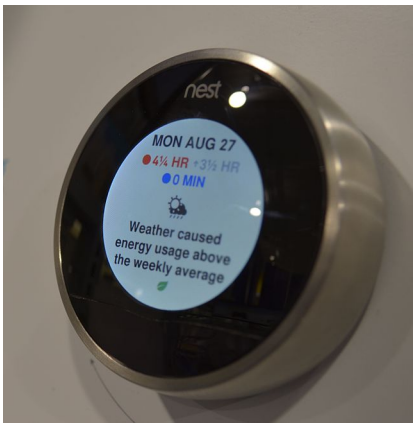


图 2: Nest 学习型恒温器

和社会领域等，具有十分广阔的市场和应用前景。

2016 年到 2017 年在线设备数量增长 31% 达到 84 亿。专家预测，2020 年前，物联网将连接 300 亿个物体，预估全球市值将达 7.1 万亿美元。

3.1.3 物联网技术简史

早在 1982 年，就出现过智能设备网络的概念。卡内基梅隆大学的修改版可乐机成了第一个接入互联网的电器。

1991 年，Mark Weiser 中提出了“无处不在的计算”（Ubiquitous Computing）。1994 年，Reza Raji 进一步发展了这个概念，描述为“移动的小数据包发送到大的节点集，从而整合和自动化从家用电器到工厂的一切”。

1993 年到 1996 年之间，多家公司提出了自己的解决方案，如微软的 at Work 或 Novell 的 NEST。

然而，直到 1999 年该领域才得到了重视。术语“物联网”（Internet of Things）是由 Kevin Ashton 在 1999 年提出的。

3.1.4 物联网设备及应用的特点

物联网设备的重要特点是硬件种类极为丰富，从低功率的 MCU 到新一代节能 32 位处理器都有。物联网应用的重要特点是需要可靠性、实时行为和与互联网的无缝集成。

这些需求对基础架构（如操作系统）开发者来说是重要的挑战。

图 3: 使用手机控制台灯的色彩和亮度

3.1.5 开发与推广应用中的困难

目前，物联网是业界的热点，各种基础架构开发方兴未艾，生态尚不够成熟。从消费者层面来看，不同厂商的设备可能无法互相沟通，隐私和安全性也很值得担忧。

从开发者的角度来看，生态的不健全使得开发物联网应用难度不必要地高于开发相应的计算机软件。

4 实时操作系统

4.1 从非实时操作系统到实时操作系统

传统的操作系统以提高吞吐量（单位时间内处理数据的总量）为目标，从而在达到平均处理时间意义上的高性能。对响应时间有严格要求的软件系统在这种系统下无法合理地工作。例如，核反应堆监测系统需要有严格的响应时间限制，一旦某项任务超出了限制，将有可能导致整个系统崩溃，甚至对人的生命安全造成危害。由此可见，实时操作系统是以提高响应速度为目标，而不追求吞吐量，因此衡量一个实时操作系统坚固性的重要指标，是系统从接收一个任务，到完成该任务所需的时间，其时间的变化称为抖动。

可以依抖动将实时性分为三种：硬实时（Hard real-time）、严格实时（Firm real-time）、软实时（Soft real-time）。

硬实时 一旦出现某期限被错过的情况，即被视为系统失败。即所有任务必须在确定期限内完成。
例子：美国的火星探路者因一个高优先级任务被低优先级任务阻塞，未在期限内完成，险些丢失。

严格实时 允许不频繁的期限被错过的情况。即绝大部分任务在确定时间内完成。
例子：装配生产线的机器人如果出错频率较低，造成的影响较小，可以认为任务是成功的。

软实时 允许频繁的期限被错过的情况，只要错过期限。
例子：传感器互相之间可以不同步，只要获取数据的时间足够相近，就可以认为是同时获取（而不是读取失败）。

物联网应用一般对实时性要求不高，达到软实时或严格实时一般即可满足要求。

4.2 案例研究：RIOT

RIOT [1]是面向物联网的操作系统，致力于填补无线传感器网络的轻量级操作系统和传统的全面的操作系统之间的空白。其特性为：可靠、实时、自适应网络栈。

操作系统可以被三个关键特点描述：

1. 内核的结构
   (a) 宏内核
   (b) 分层方法
   (c) 微内核
2. 调度策略
   (a) 是否支持实时调度
3. 编程模型

[1] http://www.riot-os.org/

(a) 所有任务执行在同一个上下文，内存地址空间没有分段

(b) 每个进程运行在自己的线程里，有自己的内存栈

其中，编程模型也与开发者的开发体验紧密相关。

RIOT 实现一个继承自 FireKernel 的为内核架构，支持多线程及标准 API。此外，RIOT 也支持 C++ 和 TCP/IP 网络栈。RIOT 的优势为

1. 高可靠性；
2. 对开发者友好的 API；
3. 模块化微内核架构使得单点故障无法影响整个系统。

为了满足常数时间的要求，RIOT 内核大量使用静态内存分配（向用户程序提供动态内存分配接口）。调度器使用线程的循环链表，从而实现 $O(1)$ 的调度。常数时间的计时器操作利用了 MCU 通常会提供的多个比较寄存器。

为了减少 IoT 设备的耗电，RIOT 提供毋须任何周期性事件的调度器。当没有任务时，RIOT 执行 idle 任务，该任务将设备置于深度睡眠状态。

RIOT 中，上下文切换发生在两种情况

1. 一个对应的内核操作被调用
2. 一个产生线程切换的中断

内核的低复杂度使得 RIOT 极为节能。

5 文件系统

文件系统是用于控制数据存储和获取的操作系统的重要组成部分。

5.1 面向物联网的文件系统

在一个物联网系统中，文件系统可以用于缓存远程数据和缓冲传感器产生的数据，或用于存储应用的长期或临时信息。这是在物联网系统中使用文件系统的基本动机，但为何不使用传统的文件系统？这是因为现有的文件系统都有一些不足之处，不适合物联网设备使用[2]。

[2] “不适合”是指不能满足物联网系统的一些需求，使得开发者需要手动开发一些支持程序库。

传统的用于计算机的文件系统有以下缺陷，使得这样的文件系统不能适用于物联网设备使用：

1. 内存消耗过大：许多文件系统在内存中缓存大量数据，以加快文件操作；
2. 不适合闪存：早期文件系统多面向传统的磁盘，数据常被放在一起，而适合嵌入式设备的闪存反而不适合这样的设计特点。由于成本原因，嵌入式设备的闪存一般不会搭载 FTL，这容易导致闪存磨损而寿命减少。
3. 设计复杂：如层次式目录结构在用途比较单一的嵌入式系统中没有用处；日志等容易造成闪存磨损。最重要的是，过于复杂的实现会导致性能、能耗、资源消耗不能满足人意。

物联网系统中嵌入式系统将直接接入网络，但若不搭载本地可擦写存储，普通的分布式文件系统也不能适用于物联网设备使用：

1. 对网络要求高：物联网设备的网络环境一般并不稳定，一旦网络断开，在无本地存储的情况下数据将完全丢失。
2. 对硬件性能要求高：分布式文件系统常常要求全功能操作系统，这在嵌入式设备上很难达到。

此外，物联网设备还有传输流数据的需求，如传感器不断收集数据并不断发送到中心服务器。

由于以上原因，目前物联网系统的开发仍然常常手动读写闪存，开发者需要区分本地存储和网络传输，相关系统的可扩展性问题亟待解决。本项目提出这样的问题：物联网需要怎样的文件系统？并将给出一个初步的答案。

6 相关工作

为开展相关工作，除项目背景中提到的实时操作系统外，我组在嵌入式文件系统和分布式文件系统上也进行了深入的调研。

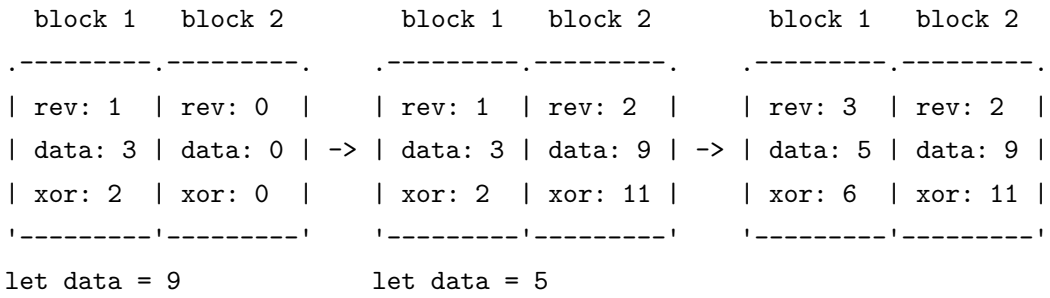


图 4: LittleFS 元数据更新

6.1 LittleFS

LittleFS 是适用于微控制器的小型故障安全文件系统[3]，它用一种简单的方法实现了随机掉电时的文件一致性，使用写时复制技术，并且可以实现闪存磨损均衡。

6.1.1 元数据

LittleFS 将元数据储存在两个块上，每个块上有一个修订号，每修改一次则增加 1。使用简单的异或进行校验。进行修改时，首先选择最旧版本修改。一旦在此过程中掉电，要么版本号未增加，要么异或未修改，则可以检测到数据没有成功更新，而旧版本仍存在，可以成功读取。

6.1.2 文件数据

文件数据是一系列块的逆向链表，从尾部依次指向头部，从而实现 $O(1)$ 时间的追加。

当文件进行修改时，首先将被修改的块写到一个新块上，然后修改文件元数据，从而保证文件数据在掉电时的一致性。

由于逆向链表的顺序读取性能较差，LittleFS 让每个编号 n 可以被 2^x 整除的块逆向 $n - 2^x$，从而实现最坏 $O(n \log n)$ 的读取。

[3] https://github.com/geky/littlefs

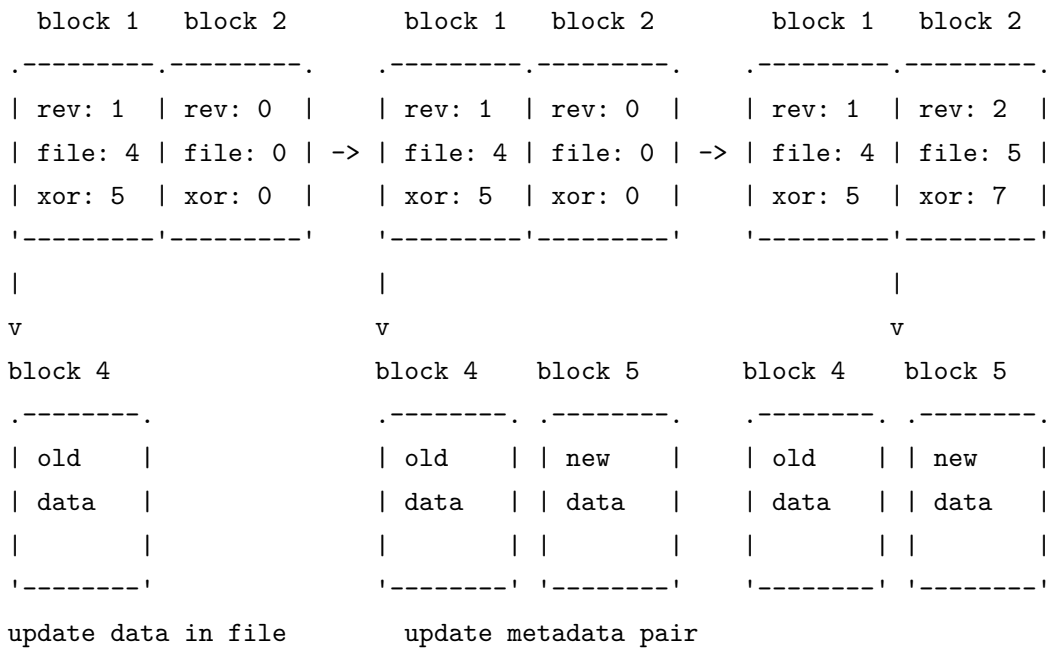


图 5: LittleFS 文件更新

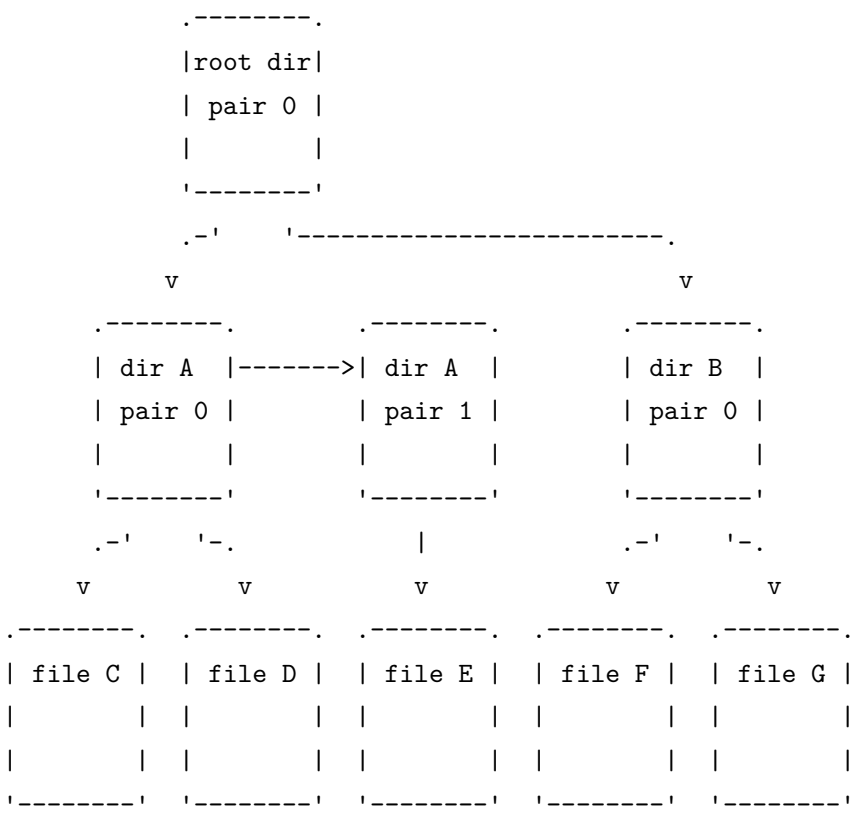


图 6: LittleFS 目录数据结构示意图

6.1.3 块分配

传统的文件系统为分配空闲块需要维护一个空闲块列表，这会造成代码膨胀、内存占用多、BUG 多等问题。为此，LittleFS 不存储哪些块是空闲的，LittleFS 只需用所有的块减去正在使用的块即可。具体实现为：使用一个固定大小的前向检查位向量，每次分配时向前扫描，直到找到空闲块。

该方法的渐进复杂度为 $O(n^2)$，但实际使用时效率已经很好。如块大小为 4 KB 的 4 MB 的闪存芯片，只需 8 次扫描即可扫完整个闪存。

6.1.4 目录

目录使用树存储。由于内存极为有限，不能使用递归遍历内存，所以使用线索树存储。

6.2 总结

以上所述是 LittleFS 最基本的特征，它在一些细节上还有更仔细的考虑。LittleFS 是一个嵌入式文件系统的典型案例，它强调了设计的简单（从而达到体积小、效率高、BUG 少的目标），同时考虑了常见的使用场景，适合嵌入式系统使用。

6.3 Google FS

6.3.1 GFS

Google（gfs）是一个可扩展的分布式文件系统，用于大型分布数据密集型应用程序，在提供容错的同时运行在廉价的商品型硬件上，并为大量用户提供高集成性能。[**?**]

6.3.2 概况

GFS 是为 Google 日益增加的数据处理需求设计的，GFS 与以前的分布式文件系统有着如性能，可扩展性，可靠性，可用性等很多共同的目标，但其设计由 Google 的应用工作负载和当前和预期的技术环境的关键观察所驱动，反映了一些早期的显着偏离文件系统设计假设，因此研究团队重新审视了传统选择并探索了根本上不同的观点。

- 首先，组件故障是规范而非概念，由于数量庞大，某些组件会发生故障。持续监视，错误检测，容错和自动恢复必须要保证。
- 其次，系统上存储的文件大多都很大，定期处理大量 TB 级的文件时管理几十亿 KB 大小的文件是不切实际的，因此设计假设和参数要重新考虑。
- 第三，大多数文件都是以附加而不是覆盖的形式被修改，文件内的随机写入实际不存在，一旦被写入则只能被顺序读取，鉴于这种大型文件的访问模式，追加成为性能优化和原子性保证的重点，而在客户端缓存数据块则失去了吸引力。
- 第四，通过增加灵活性，共同设计应用程序和文件系统 API 可以使整个系统受益，如放宽了 GFS 的一致性模型，大大简化了文件系统，而不会给应用程序造成沉重的负担。还引入了一个原子追加操作，以便

多个客户端可以并发地追加到一个文件中，而无需在它们之间进行额外的同步。

6.3.3 设计综述

设想

- 要定期监视检测故障，以及容错性和故障恢复
- 系统主要存储大文件，小文件也要支持，但不对其进行优化
- 两种文件读取方式：大型流阅读和小型的随机阅读
- 系统必须有有效的对同一文件进行并发附加的处理机制
- 高持续带宽比低时延更为重要

6.3.4 Interface

支持一般的创建、删除、打开、关闭、读写文件的功能。

GFS 还支持快照和记录追加，记录追加可以让多个客户端同时向同一个文件追加数据而无需加锁。

6.3.5 Architecture

GFS 由 master 和 chunkservers 组成，也可以轻易的在同一台机器上运行 chunkserver 和 client。

文件被分为等大的块，每一块被 64 位的 chunk handle（master 根据块创建的时间产生）确定。每一块都在多个 chunkserver 上有备份，这样能确保稳定性，默认三份，用户可以指定备份级别。

master 存储所有的元数据，如命名空间，访问控制信息，文件与 chunk 的对应关系，chunk 的位置等，也控制系统级别的活动如垃圾清理，孤儿 chunk 的处理，chunk 的合并等。master 定期与 chunkserver 联系以传递指令和收集 chunkserver 的状态。

client 则通过与 master 和 chunkserver 的沟通处理数据，将元数据处理交给 master，而数据承载则直接交给 chunkserver。

不论是 client 还是 chunkserver 都不缓存数据，因为这没有好处。

6.3.6 single master

用户添加文件时，直接将文件名及 offset 等信息发送给 master，master 添加文件信息并给 client 传回 location 及 index，client 再用此信息将文件传给 chunkserver。

6.3.7 chunk size

chunk 大小指定为 64MB
好处：

- 减少了 client 与 master 的沟通
- client 可以在给定的 chunk 上进行更多操作，减少建立 TCP 连接需要的时间。
- 减少了 master 上元数据的大小。

缺点：hotspot

6.3.8 metadata

master 上的 metadata 有 namespace，文件与 chunk 的对应关系，chunk 的位置。使用 log 使得系统升级更简单稳定。

master 负责周期性的扫描，包括垃圾处理，错误恢复，为平衡负载的 chunk 合并等。

master 不维护特定 chunk 的位置记录，只是在开始的时候与 chunkserver 通信，这解决了 chunkserver 的加入，离开，故障等时 master 与其的同步。

operation log 包括元数据改变的历史记录，这非常重要，因此在远程的部分机器上也进行备份，master 通过 replay operation log 来恢复文件系统状态。当 log 的 size 达到一定程度时，master 检查状态，这样恢复的时候只要恢复最新的 log 上几条记录即可。checkpoint 在能直接被 map 入内存的一个紧凑的 B+ 树内，在启动时系统可以将 log 记录进新的 log 文件中，这样就不用 delay 启动期间的修改。checkpoint 期间的 fail 不影响准确性，因为恢复操作会跳过未完成的 checkpoint。

6.3.9 Consistency Model

GFS 的一致性：

- defined: 状态已定义，从客户端角度来看，客户端完全了解已写入集群的数据，例如，客户端串行写入且成功，此时的状态是 defined
- consistent: 客户端来看 chunk 多副本的数据完全一致，但不一定 defined，一般发生在多客户端并发更新时
- unconsistent: 多副本数据不一致
- undefined: 数据未定义

从修改过程看：

- 串行 over-write: over-write 由客户端指定文件更新 offset。当客户端是串行更新时，客户端自己知道写入文件范围以及写入数据内容，且本次写入在数据服务器的多副本上均执行成功。因此，本次写结果对于客户端来说就是明确的，且多副本上数据一致，故而结果是 defined
- 并行 Over-Write: 并行写入时多个客户端由于写入范围可能交叉而形成交织写。这时候，由于单个客户端无法决定写入顺序（只有主副本才能决定谁先写谁后写），因此，即使写入成功，客户端仍无法确定在并发写入时交叉部分最终写入结果，但是因为写入成功，所以多副本数据必然一致，即为 consistent but undefined
- append: 客户端 append 操作无需指定 offset，由 chunk 主副本根据当前文件大小决定写入 offset，在写入成功后将该 offset 返回给客户端。因此，客户端能够根据 offset 确切知道写入结果，无论是串行写入还是并发写入，其行为是 defined
- 在一系列的成功修改后，mutated file region 被定义并且包含最后一次修改后的版本。之后按文件中的 chunk 顺序修改，并检测是否有过期的 chunk，旧的 chunk 不会在与 client 通信时使用。
- 客户端缓存 chunk 的位置可能过期，于是 client 对缓存有时间限制，这会清除所有缓存信息。

- 组件的 failure 可能会损坏文件，因此 GFS 定期与 chunkserver 握手并使用 checksum 检查文件完整性，一旦出现问题，数据会迅速由其余可用的备份中恢复。除非在能够联系之前所有备份全部丢失，否则文件不可能彻底损坏（这种情况用户会收到 error 而不是损坏的文件）。
- 检查点允许编写者以增量方式重新启动，并无法成功处理在应用程序看来不完整的数据。

6.3.10 租约 (Lease)

租约（Lease）是由 GFS 中心节点 Master 分配给 chunk 的某个副本的锁。持有租约的副本方可处理客户端的更新请求，客户端更新数据前会从 Master 获取该 chunk 持有租约的副本并向该副本发送更新请求。

租约本质上是一种有时间限制的锁：租约的持有者（chunk 的某个副本）需要定期向 Master 申请续约。如果超过租约的期限，那么该租约会被强制收回并重新分配给其他副本。

6.3.11 优点与不足

优点 负载均衡，容错性好，可扩展性强，满足 POSIX 语义，应用程序无需任何修改可直接使用 GFS。

缺点 Master 启动时将所有元数据加载至内存中，由于 gfs 应用场景主要是大文件，对其影响并不严重，但限制了文件系统的可扩展性。

6.4 星际文件系统

6.4.1 摘要

星际文件系统（InterPlanetary File System，缩写 IPFS，下用 IPFS 代替）是一个试图用相同的文件系统来连接所有计算机的端对端（P2P）分布文件系统。IPFS 与万维网的结构类似，但它也可以视作使用一个 Git 仓库交换数据的 BitTorrent 单群。IPFS 提供了一个高吞吐的区块存储模型，并带有基于内容寻址的超链接，从而形成一个广义的 Merkle 有向无环图。它由分布的散列表、带有激励机制的区块交换系统和自我认证的命名空间构成，没有单点故障，节点之间也不需要互相信任。

6.4.2 IPFS 设计

IPFS 借鉴了以前端对端系统中的优秀想法，比如 DHT，BitTorrent，Git 和 SFS 等。在 IPFS 中，每份文件以及其中所有的区块都有一个唯一的指纹，被称作加密散列；IPFS 对每个文件追踪历史版本并删除冗余的复制版本；每个网络节点无需存储全部数据，仅存储它感兴趣的一部分，以及一些索引信息来帮助它从其他机器上下拉数据；查询文件的时候，它通过网络来寻找含有此文件的节点；IPFS 使用星际命名空间（InterPlanetary NameSpace，缩写 IPNS，下用 IPNS 代替）来将文件的散列映射为人类可读的名字。

IPFS 协议采用端对端模式，所有节点都是平等的。文件与数据结构统称 IPFS 对象。节点在本地存储 IPFS 对象并使用网络互相传递。IPFS 协议分为 7 个不同功能的子协议：

1. 身份协议：管理节点身份生成与验证。
2. 网络协议：使用各种已有的底层协议管理端对端连接。
3. 路由协议：通过维护信息帮助定位特定的端或对象，同时对本地或者远程请求相应。
4. 交换协议：使用新的区块交换协议（BitSwap）有效地分配区块，该协议类似于市场交易机制。
5. 对象协议：使用 Merkle 有向无环图的结构，用链接来建立不可变动的对象，比如文件数据结构和交流系统等。
6. 文件协议：一个类似 Git 的带版本管理的文件系统。
7. 命名协议：一个自我认证的可变的命名空间。

身份协议 节点通过 NodeID 辨认身份，NodeID 是通过 S/Kademlia 加密谜题来对公钥做散列得到的，公钥和私钥使用口令加密存储在各自的节点中。

在每次启动时，用户都可以选择实例化一个新的节点，不过这个操作会流失他们在网络上原先的累计收益，所以一般他们都愿意保留原先的节点。

在第一次连接时，双端交换公钥，并检查公钥的散列与 NodeID 是否一致，若不一致则拒绝连接。

IPFS 也接受使用多种散列函数，所以在交换公钥的时候也可以在头部加入自己的散列函数的代码。

网络协议 通常，一个节点时常与数百个网络中的其他节点沟通信息，有时也可能会跨越整个网络。IPFS 网络协议带有如下特性：

1. 传输：IPFS 可以使用任何传输协议，推荐使用 WebRTC DataChannels 或者 uTP。
2. 可靠性：如果下层的协议没有保证可靠性，那么 IPFS 也会提供可靠性。
3. 连通性：使用 ICE NAT 遍历技术。
4. 完整性：可选，使用散列校验和检查消息的完整性。
5. 权威性：可选，使用发送者的公钥和 HMAC 算法确认消息的权威性。

路由协议 节点需要一套路由系统来帮助它们找到其他端的网络地址，以及那些可以提供特定对象的端。IPFS 使用一个基于在 S/Kademlia 和 Coral 上的 DSHT 来达成此功能。IPFS DHT 通过区块的大小区分它们的存储方法，例如小于等于 1KB 的区块 DHT 直接存储，大于 1KB 的区块 DHT 将存储引用信息（能提供此区块的对象的 NodeID 列表）。

交换协议 IPFS 使用 BitSwap 协议交换区块。类似于 BitTorrent，BitSwap 中的端有一条需求列表记录它想要的区块，也有一条提供列表记录它能提供区块。BitSwap 的机制类似于一个永恒的市场，在这里节点可以自由地获取和提供任意存在的区块。一些区块可能来自于毫无关联的文件，不过节点也可以一起打包出售。

在 BitTorrent 里，为了防止有些节点只是抓取数据但从不提供，它使用了投桃报李的策略来惩罚这些节点。在 BitSwap 里也定义了类似的交换意愿。在端对端中，记债务比

$$r = \frac{bits\_I\_sent}{bits\_I\_receive + 1}$$

并定义发送意愿

$$P_{(message\_willing\_to\_send|r)} = 1 - \frac{1}{1 + e^{6-3r}}$$

在每次发送或者接收数据后，更新 r。可知，如果对方接受的数据与发送的数据的比例上升，那么我方的发送意愿将降低。当这个比例在 1 附近的时候，发送意愿大约在 0.95；若比例提升到 2，其意愿将降低到 0.5；若比例超过 3.6，则意愿迅速降为 0.01 以下。该策略可以防止通过大量创建全新节点然后纯粹为了拉取数据的行为。

BitSwap 协议要求每个节点各自保存自己的一份账本存储这些信用信息，也包括自己的。在双端建立连接时，它们需要交换账本并保持信息一致。如果不一致，也可建立新账本，但会损失过去的信誉或债务。为防止恶意节点故意丢失账本来欺骗节点，对方也可以认为丢失账本者不可信，从而直接拒绝交易。

对象协议

DHT 和 BitSwap 为 IPFS 构成一个庞大的端对端系统，用于快速可靠地储存和分布区块。IPFS 建立了一个 Merkle 有向无环图，其将对象用加密散列连接起来。这是一个广义的 Git 数据结构。Merkle 有向无环图为 IPFS 提供了许多有用的特性，如：

- 内容寻址：所有内容都用它的多个散列校验和唯一定位。
- 防篡改：所有的内容都可通过校验和检验，IPFS 可以据此检测被篡改或损坏的数据。
- 去重：拥有一致内容的对象视作等价，也只存储一次。

IPFS 用散列唯一确定路径，所以以下路径都是等价的：/ipfs/<hash of a>/b/c /ipfs/<hash of b>/c /ipfs/<hash of c>

文件协议 IPFS 的文件结构与 Git 非常接近，故不做表述。

命名协议 IPFS 使用内容来给文件寻址，但是文件是变动的，需要找到一种方法使得 IPFS 能在相同的路径找到变动的数据。IPFS 认为，对象是永恒存在的，从而其采用了类似 Git 的版本管理，其中对象是不变的，但是引用确是可变的。

在 IPNS 中，为每个节点建立如下命名空间：

/ipns/<NodeID>

其下的内容不按照内容寻址方式建立，而是使用子目录路由协议。

6.4.3 IPFS 实际使用案例

土耳其官方不能封禁的维基百科

土耳其在 2017 年 4 月封禁了维基百科。一些黑客行为主义者制作了一份维基百科的副本，并使用 IPFS 的方式在线发布。因为 IPFS 搜索内容的方式是从最近可用的副本中选择一个下载，用一个 IPFS 地址理论能找到任何一个副本，所以官方封掉一部分副本后，还可以找到另一份副本解决问题。

加泰罗尼亚独立公投网站复刻

加泰罗尼亚在 2017 年 10 月 1 日举行公投，此前几天，西班牙法院判定公投官网违法并予以阻拦。加泰罗尼亚海盗党使用 IPFS 复刻公民投票网并成功举行公投，随后在同年 10 月 27 日宣布独立。

6.4.4 优点与不足

优点 分布式的文件系统减少了存储需求，版本管理使得历史可追溯，同时也可以设置成允许有多个副本，使得其能够做到更强的容错性，弱化垄断力量的介入，保证民主自由的进行。

不足 路由的算法实质上在大网络里效率并不是特别高，而市场化的区块交易机制也依然会出现所有市场经济该出现的典型问题，且保存所有版本实质也增加了存储开销，在没必要保存历史的情况下是没有意义的。

6.5 Gnutella 项目

6.5.1 关于 Gnutella 项目

Gnutella 是一个用于分布式搜索和数字资源共享的协议，他的特色是它的点对点、非中心的模型。

在这个模型中，任何一个客户端同时也是一个服务器，反之亦然。所以 Gnutella 中给它们起了一个专门的名字叫做 servent (是从 “SERVer” 和 “cliENT” 各取一部分组成的)。Servent 提供客户端接口，用户通过这个接口可以提交查询并查看查询结果，同时，它们也可以接受查询请求，在本地数据中检索，并返回符合条件的结果。因为这种分布式属性，实现 Gnutella 协议的由 servent 组成的网络具有高度的容错性，一部分 servent 掉线并不会中断正在进行的网络操作。

6.5.2 Gnutella 协议

1. 连接 Gnutella 网络（Bootstrapping 过程）

   为了连接 Gnutella 网络，servent 需要发现并存储多个主机的地址，有四种方式

   (a) 调用一个 GWebCache。这是一个有人上传到他们的 Web 服务器的 PHP 脚本。运行在用户计算机上的 Gnutella 程序调用脚本来获取计算机的 IP 地址，并获取最近完成相同事情的程序的 IP 地址。这让 Gnutella 程序能够找到彼此。

   (b) 在握手期间，Gnutella 程序在 X-Try 和 X-Try-Ultrapeers 头文件中告诉对方更多的 IP 地址。

   (c) 有些地址以相同的方式洒在 Pong message(reply to ping(discover hosts on network)) 中。

   (d) Query hit(reply to query) 也包含 IP 地址。

2. 启动算法

   一般情况下，在一个会话期内，不要向 web cache 发送太多的请求，查询 GWebCache 的次数不应该超过一次。也就是说，一个用户掉线的时候，代替的方案是用本地主机的 cache 来存储发现的主机地址，并和 GWebCache 一起使用。

   - 启动客户端并调入本地主机 cache
   - 使用本地主机 cache 连接 GNet.
   - 如果在 X 秒后还没有连接到 GNet，查询一个随机的 GWebCache.
   - 等待最少 Y 秒后，向另一个 GWebCache 发送请求。(在第一次选取的 GWebCache 死机或者反应非常慢的情况下)
   - 在 Y 秒后，如果第一次 GWC 调用还没有返回结果，每隔 Z 秒调用一个新的 GWC，直到收到其中一个的响应。
   - 一旦至少收到一个 GWC 的返回结果，只有在本地主机 cache 为空的情况下，servent 才应该在接下来的 session 中发送新的请求 (如果软件提供初始的主机 cache，并且没有清除主机 cache 的选项，这种情况就不应该发生).

假设 X，Y 和 Z 的值分别为 5 秒、10 秒和 2 秒，这样应该能够确保 servent 能够很快连接到 GNet 并把查询 GWebCache 的次数控制到最低。

启动过程还应该确保 servent 不用经常连接一个远程主机，每秒钟少于一次是一个合理的界限。为了达到这个要求，一个解决办法就是确保本地主机 cache 中总是有足够多的主机，这样就能保证任何一个主机都不会在一分钟之内被再次试图连接。

每一个 servent 在握手时都应该发送一个 X-Try 头，这个头给出这个 servent 可以尝试连接的主机列表，允许它在不用调用 GWebCache 的情况下得到新的主机地址。例如：

X-Try: 1.2.3.4:1234, 1.2.3.5:6346, 1.2.3.6:6347

servent 应该发送适当数量的主机，一般在 10 到 20 之间。这些随着 X-Try 头发送的主机信息应该是确切的。所以，servent 应该只选择那些最近（在发送 X-Try 头之前几分钟内）所知还在运行的主机。如果最近发现还在运行的主机数量少于 10 个，那发送时就少发送一些。如果 servent 实现了响应 cache（参见 3.4 节），在响应信息中发现的主机可以认为具有空闲的时隙。但这不适用于在监控 QueryHits 中得到的主机信息。这样，在返回信息中得到的主机信息可以替换 X-Try 头中主机的位置，而 QueryHits 中发现的主机却不行。

6.5.3 握手协议

一个 Gnutella servent 通过与当前在网络上的另一个 servent 建立连接的方式把自己连接到 Gnutella 网络。一旦第一个连接建立起来，网络可以提供更多的当前连接在网上上的主机地址。在得到当前在网上的另一个 servent 的地址时，首先建立一个到 servent 的 TCP/IP 连接，然后开始一个握手的消息序列。客户端是发起连接的主机，服务器是接受连接的主机。

（以下步骤来自于网络）

1. 客户端建立一个到服务器的 TCP 连接
2. 客户端发送 “GNUTELLA CONNECT/0.6<cr><lf>”
3. 客户端发送带有所有域的头，除了供应商自定义的特殊域，每个域以”<cr><lf>” 结束，在整个头的最后添加一个额外的”<cr><lf>”

4. 服务器以"GNUTELLA/0.6 200 <string><cr><lf>" 应答,这里 <string> 应该是"ok", 但是 servent 应该只查找"200" 这个代码作为状态标志。

5. 服务器发送自己信息头的全部，按照 3 规定的格式

6. 如果客户端在解析完服务器发送的信息头以后，还希望继续，就发送"GNUTELLA/0.6 200 OK <cr><lf>" 给服务器作为应答, 就像 4 一样。否则，它需要返回一个错误码，并关闭这个连接。

7. 客户端发送供应商自定义的信息头，与 3 的格式一样。

8. 客户端和服务器按照 3 和 5 中得到的信息，按照自己的要求发送和接收二进制消息。

   例如：First-Field: this is the value of the first field<cr><lf> Second-Field: this is the value<cr><lf> of the<cr><lf> second field<cr><lf> <cr><lf>

   下面是一个客户端和服务器交互的例子。从客户端发送到服务器的数据列在左边，从服务器发送到客户端的数据列在右边。

Client	Server
GNUTELLA CONNECT/0.6<cr><lf> User-Agent: BearShare/1.0<cr><lf> Pong-Caching: 0.1<cr><lf> GGEP: 0.5<cr><lf> <cr><lf>	
	GNUTELLA/0.6 200 OK<cr><lf> User-Agent: BearShare/1.0<cr><lf> Pong-Caching: 0.1<cr><lf> GGEP: 0.5<cr><lf> Private-Data: 5ef89a<cr><lf> <cr><lf>
GNUTELLA/0.6 200 OK<cr><lf> Private-Data: a04fce<cr><lf> <cr><lf>	
[binary messages]	[binary messages]

6.6 百度文件系统（Baidu File System）

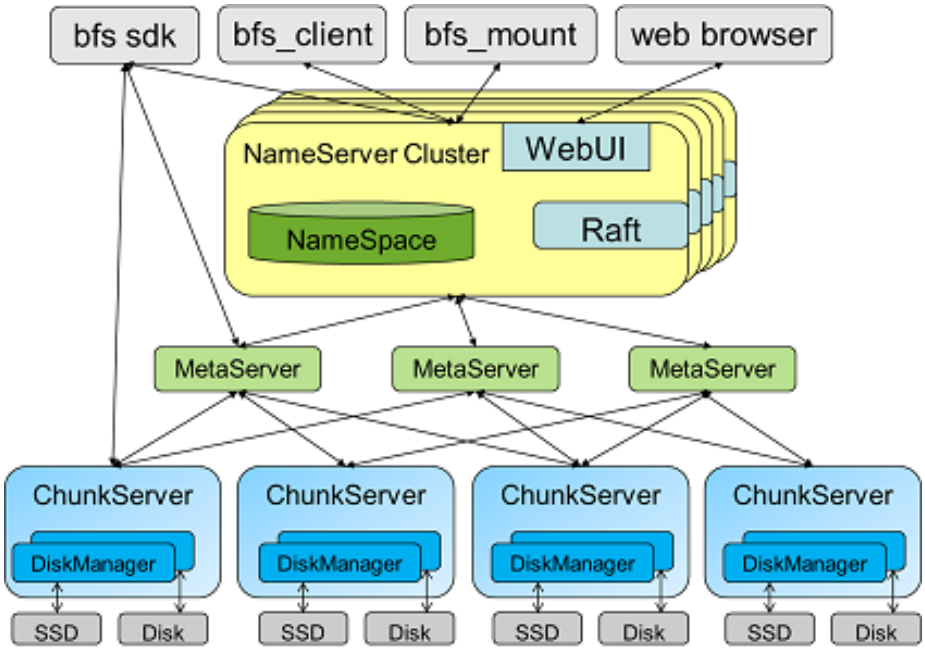
6.6.1 介绍

百度文件系统（以下简称 BFS）是一个为支持实时应用而设计的分布式文件系统。类似于其他分布式文件系统，BFS 具有高容错性。但与此同时，BFS 还兼具低读写延迟和高吞吐量的特性。[?]

6.6.2 特性

1. 持续可用性：Nameserver 多节点冗余，并使用 Raft 共识算法维护其一致性，单个节点宕机不影响整体可用性。
2. 高吞吐：高性能的数据引擎能最大化 IO 的吞吐量。
3. 低延时：全局负载均衡，自动检测并规避慢速节点。//似乎尚未实现
4. 可线性扩展：支持多数据中心的架设及 1w+ 数据节点管理。

6.6.3 结构：共五层



由底至上：

1. 硬件
2. 单机节点 ChunkServer：管理底层硬件
3. MetaServer：为 NameServer 分担压力
4. NameServer：类似于 hdfs 的 namenode，但是一个集群而非单节点，这样可以解决部分可用性和性能瓶颈的问题。[4]
5. 外层：用户接口/本地服务/sdk 等

6.6.4 总体设计

1. 主备 master 结构：
   - master 之间通过一致性协议保持同步，每个 master 都存储所有信息。
   - master 的数据是落地（即被储存在持久化存储设备，如硬盘中）的，不一定非得全内存，通过 LRU cache 算法保证热点访问的快速响应。
2. 文件分块
   - 文件是否分块决定了能否支持大文件
   - 文件名信息是否存储在 master 决定了是否支持小文件
   - 折衷是对大文件才分块，小文件 (<100G) 不分块；创建文件夹时指定可否 list，可 list 的必须存在目录树中，不可 list 的可以哈希到 chunkserver，让 chunkserver 去维护 meta 信息。
3. 文件 meta 信息（元数据：描述文件属性的数据）
   - 文件 meta 信息存储在 master 会导致 master 负载过高（内存，cpu)。我们选择对于不分块的文件，将文件 meta 存储在 chunkserver，如果用户要 list，那仅需要访问 master，但如果要获取文件大小等信息，那就必须得访问对应的 chunkserver 了，master 仅存储 namespace 和每个文件在哪些 chunkserver 上。

[4]hdfs 中分 Namenode 和 Datanode 两种节点，是主从式结构。其中，Namenode 负责管理文件系统的名字空间 namespace 以及客户端对文件的访问，而 Datanode 则负责节点存储。这种设计简化了系统的架构，因为存储数据无需经过 Namenode。

4. 写数据流有两个选择：
   - client 写一份，然后 chunkserver 通过复制链创建多副本，优势是 client 带宽占用低；
   - client 一次写多份，优势是可以多写一份，并在写的过程中抛弃慢节点。
5. 名称空间使用 LevelDB 存储：可以简单的将整个目录结构平展开，并提供高效的文件创建、删除、list 和 rename 操作。例. 对于实际目录结构：

   /home/dirx/

   /filex

   diry/

   /filey

   /tmp/

   filez

   存储格式为：

   1home -> 2

   1tmp -> 3

   2dirx -> 4

   2diry -> 5

   3filez -> 6

   4filex -> 7

   5filey -> 8

6.6.5 Master（NameServer）高可用设计方案：

1. 背景：在 BFS 中，NameServer 负责管理文件 meta 信息以及所有 ChunkServer 的状态信息，是整个系统中唯一拥有全局信息的模块，同时也成为了系统的单点。NameServer 的不可用会直接导致文件系统的瘫痪，于是提高 NameServer 的可用性至关重要。

2. 结构：将 NameServer 扩展为集群，Client 只与集群 Leader 进行交互。每一个 NameServer 中有一个同步模块（Sync），负责集群间的状态同步，保证元数据的一致性。

   - Sync 模块

     Sync 模块主要有两个功能，选主和日志同步。选主操作就是在 NameServer 中确定唯一一个 Leader，并且在 Leader 异常后迅速选出新任 Leader。日志同步操作实际上就是同步 NameServer 中所有需要落地的数据写操作。Sync 模块设计为与 NameServer 松耦合，仅暴露必须接口，内部可以采用任意一致性协议实现。这样的设计使得我们可以实现多种一致性方案，以满足不同程度的可用性及性能需求。当前我们采用 Raft 协议实现。

   - Leader

     NameServer 集群中的 Leader 负责接收和响应 Client 的请求，并把所做出的决定通过 Sync 通知集群中其他 NameServer。Leader 只在通知成功后才会向 Client 返回操作成功。

   - Leader

     NameServer 集群中的 Leader 负责接收和响应 Client 的请求，并把所做出的决定通过 Sync 通知集群中其他 NameServer。Leader 只在通知成功后才会向 Client 返回操作成功。

3. 主要流程操作方案：

   - 对写操作（改变文件 meta 信息的操作，包括 Create、Delete 和 Rename）：

     同步结果的方案：Leader 在收到操作后先对数据库打一个快照，检查完合法性后，在提交给 Sync 之前就将数据写入本地数据库。之后再将操作提交给 Sync，Sync 返回扩散成功后，Leader 将快照删除并给向 Client 返回成功。在 Sync 返回成功前，所有的读请求都会读快照之前的数据，也就是说 Client 不会读到还未扩散成功的数据。

     同步操作的方案：Client 向 Leader Nameserver 发起请求，Leader 收到请求后，对所需操作的路径加锁，检查操作合法性。如果操作是合法的，Leader 将需要落地的数据通过 Sync 扩散给从

Nameserver。Sync 模块返回扩散成功后，Leader 向 Client 返回操作成功。从 NameServer 收到提交操作的指令后，无需检查操作合法性，直接根据指令执行操作更改状态机和内存结构。

两种方案各自有一定的局限性：同步结果的问题在于当操作的结果很大（例如删除目录下所有文件），可能超出内存大小范围，从而很难保证操作原子性。同步操作基于一个假设：Leader 和从 NameServer 将同一个指令应用到状态机及更改内存状态所产生的结果严格一致。

对扩散的失败，例如 RPC 超时、写失败等需要重试处理，如果超过一定重试次数，则均视为 Nameserver 集群不可用，需人工介入处理。各一致性协议对于扩散失败的定义不同，例如 Raft 中，大于半数成员收到消息便认为扩散成功；主从模式中主或从任意一方写失败均认为是扩散失败。

- 对读操作：

  因为数据扩散存在延时，只有 Leader 掌握最新的数据，所以只有 Leader 可以响应 Client 的读请求。Client 向 Nameserver 集群中任一实例发起读请求，如果正好是 Leader 接收到了请求，则直接返回给 Client 结果；否则，从 Nameserver 返回读失败并告知当前 Leader 地址，Client 向新 Leader 请求。

- 关于模式切换：

  主从方案有两种模式：主从模式和单主模式。严格的主从模式，需要日志成功同步给从后再向用户返回成功。这种情况一旦从宕机，会导致集群不可用。所以引入了单主模式，在从故障的情况下，依然可以提供服务。单主模式下，主将日志持久化到本地 log，之后直接向用户返回结果。

- 关于主从切换：

  当主宕机时，需要将从切换为主。这时向从发送切换命令，从将自己的 term 加一，然后切换为主对外提供服务。term 的作用主要为了标识曾经发生过主从切换，因为当前实现中，主的状态机中可能存在脏数据，之前的主作为从重启，而又没有清理自己的状态机，脏数据将不会被发现。在 term 机制下，之前的主以从的身份重启后，会发现有比自己 term 更高的主存在，会将自己的状态机和本地日志清理干净，然后等待主发送镜像。

6.6.6 启发

BFS 的特殊之处有两点,其一是将系统分为 NameServer 和 ChunkServer 两个部分：众多 ChunkServer 各负责一块盘上文件的存储，而 NameServer 负责与 Client 通信控制存储。NameServer 是整个系统的中心，但由于无需存储所有的文件内容，结构相较于传统的中心化存储系统有所简化。如果我们要做物联网设备的文件系统，可以考虑加入这样一个控制中枢，所有的系统控制操作全部在中心节点完成，而其他物联网设备全部视为存储节点，只需负责发送和接收文件内容信息，可能可以实现减轻物联网设备工作压力的目的。第二个特点是将 NameServer 集群化，提高了系统的可靠性。在规模不太大的系统中，为中心节点添加一个从节点作为备份就可能有效提高系统的可靠性，在我们的设计中也可以考虑这一点。