



# 第9章 谷歌文件系统(GFS)

# Google的观点

- 90%计算任务都能够通过“云计算”技术完成
- 桌面软件正在向Web软件转型
- 云计算是开放标准，业界不会有公司独裁
- 中小企业、大学、消费者会相对迅速地转向基于Web的“云计算”技术
- 新的赢利模式
  - 低廉的云计算给Google带来更多的流量，进而带来更多的广告收入
- 承认“云计算”不会在一夜之间普及
  - 大公司通常会慢慢地改变自己的习惯
  - 其它问题，例如“飞机问题”，以及在不能上网时用户如何工作。



Google CEO  
埃立克.施米特

# Microsoft的观点

- 在计算机上安装的传统软件是微软的根本
- 比尔·盖茨(Bill Gates)接受媒体采访时曾提出：“我们致力于推动PC成为一切的中心”
- 微软将自身的战略称为“**软件加服务**”
- 微软将Google的乐观称作是一厢情愿。
  - 利用Web软件收发电子邮件、处理文档和电子表格、进行协作很方便吗？
  - 高速宽带连接会象Google断言的那样普及和可靠吗？
  - 企业、大学、消费者会让Google保存他们的资料吗？



Microsoft CEO  
史蒂夫·鲍尔默

孰优孰劣，市场检验！



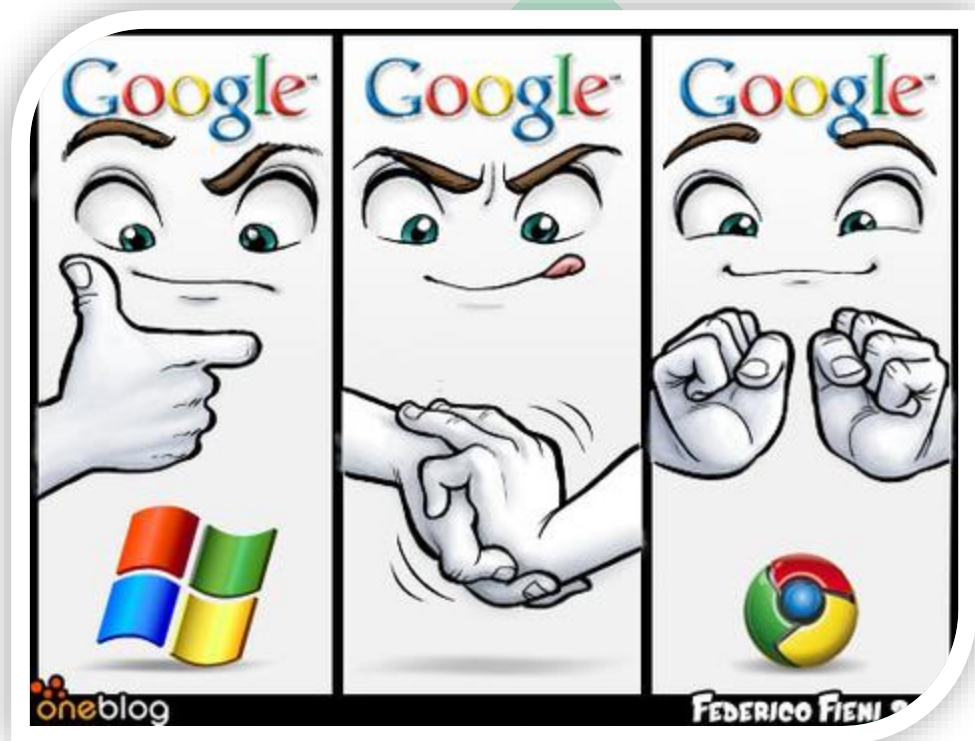
# Google的秘密武器

---

- 应用规模对于系统架构设计的重要性
- Google应用的特性
  - 海量用户+海量数据
  - 需要具备较强的可伸缩性
  - 如何又快又好地提供服务？

秘密武器： 云计算平台

# Google的云计算梦想



“浏览器=操作系统”

# Google 云计算的应用分类

## SaaS

Google Docs

Google Maps

Gmail

Google Calendar

Google Wave

.....

## PaaS

Google App Engine

Google 文件

Google 地图  
谷歌 中国

Google 日历

Gmail™  
by Google BETA

Google™  
app engine

# Google云计算应用场景

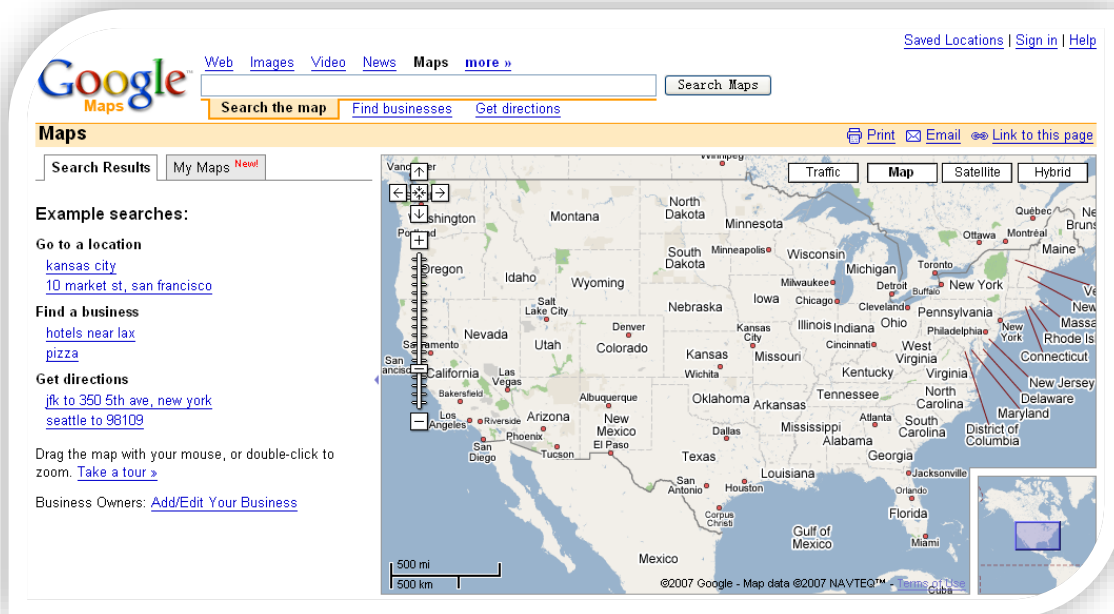
## ■ Google在线文档

Google 文件



# Google 云计算应用场景

## ■ Google 地图





# Google 云计算应用场景

## ■ Google 邮件



欢迎使用 Gmail

Google 提供的电子邮件服务。

Gmail 的开发理念是，电子邮件可以更加直观、高效而实用，甚至可能很有趣。毕竟，Gmail 具有以下特点：



### 减少垃圾邮件

利用 Google 的创新技术可以将垃圾邮件拒于收件箱之外。



### 手机邮箱

将手机的网络浏览器指向 <http://gmail.com>，便可以在您的手机上查阅Gmail。 [了解详情](#)



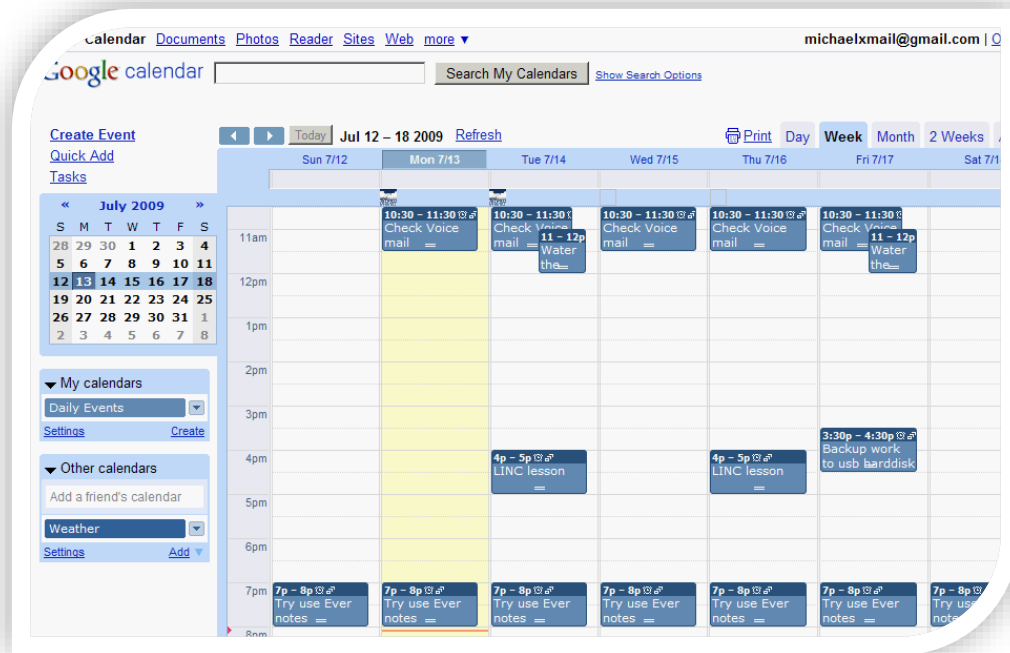
### 超大空间

超过 7439.188374 MB（还在不断增加）的免费存储空间。

# Google 云计算应用场景

## ■ Google 日历

Google 日历



# Google 云计算应用场景

## ■ Google Wave

■ 信息分享、协作、发布平台



# Google云计算应用场景

- 隶属于PaaS的Google云计算
  - 属于部署在云端的应用执行环境
  - 支持Python和Java两种语言
  - 通过SDK提供Google的各种服务，如图形、MAIL和数据存储等
  - 用户可快速、廉价（可免费使用限定的流量和存储）地部署自己开发的应用（如创新的网站、游戏等）



# Google 云计算应用场景

## ■ 应用场景特点



应用（功能实现）在云端  
存储在云端  
计算在云端



# Google服务器猜想

---

- 不论何时，不论何地，也不论你搜索多么冷门的词汇，只要你的电脑连接互联网，只要你轻轻点击“**google**搜索”，那么这一切相关内容**google**都会在**1**秒钟之内全部完成，这甚至比你查询“我的文档”都要快捷。
- 这也就是为什么**Google**创业**12**年，市值超过**2000**亿美元的原因。
- 有人可能认为**Google**拥有几台“蓝色基因”那样的超级计算机来处理各种数据和搜索，事实是怎样的呢？



# Google服务器猜想

---

- 2006年，Google大约有**45万台**服务器，2010年增长到**100万台**，这些服务器都不是什么昂贵的服务器，而是非常普通的**PC级别**服务器，其中的服务器硬盘还普遍是**IDE**接口、并且采用**PC级**主板而非昂贵的服务器专用主板。

# Google服务器







# Google需求

---

- 跨数据中心的高可靠性
- 成千上万的网络节点的伸缩性
- 大读写带宽的需求
- 支持大块的数据，可能为上千兆字节
- 高效的跨节点操作分发来减少瓶颈



# Google数字

---

- 在2005年Google索引了80亿Web页面，现在没有人知道具体数目，近千亿并不断增长中。
- 目前在Google有超过500个GFS集群。一个集群可以有1000或者甚至5000台机器。成千上万的机器从运行着5000000000000000000字节存储的GFS集群获取数据，集群总的读写吞吐量可以达到每秒40兆字节。
- 目前在Google有6000个MapReduce程序，而且每个月都写成百个新程序。
- BigTable伸缩存储几十亿的URL，几百E的卫星图片和几亿用户的参数选择。



# Google存储

---

- Google存储着海量的资讯，近千亿个网页、数百亿张图片。早在2004年，Google的存储容量就已经达到了5PB。
- Google没有使用任何磁盘阵列，哪怕是低端的磁盘阵列也没用。Google的方法是将集群中的每一台PC级服务器，配备两个普通IDE硬盘来存储。
- 这样的以一个PC级别服务器搭建起来的系统，怎么能承受巨大的工作负载呢？怎么能保证高可用性呢？



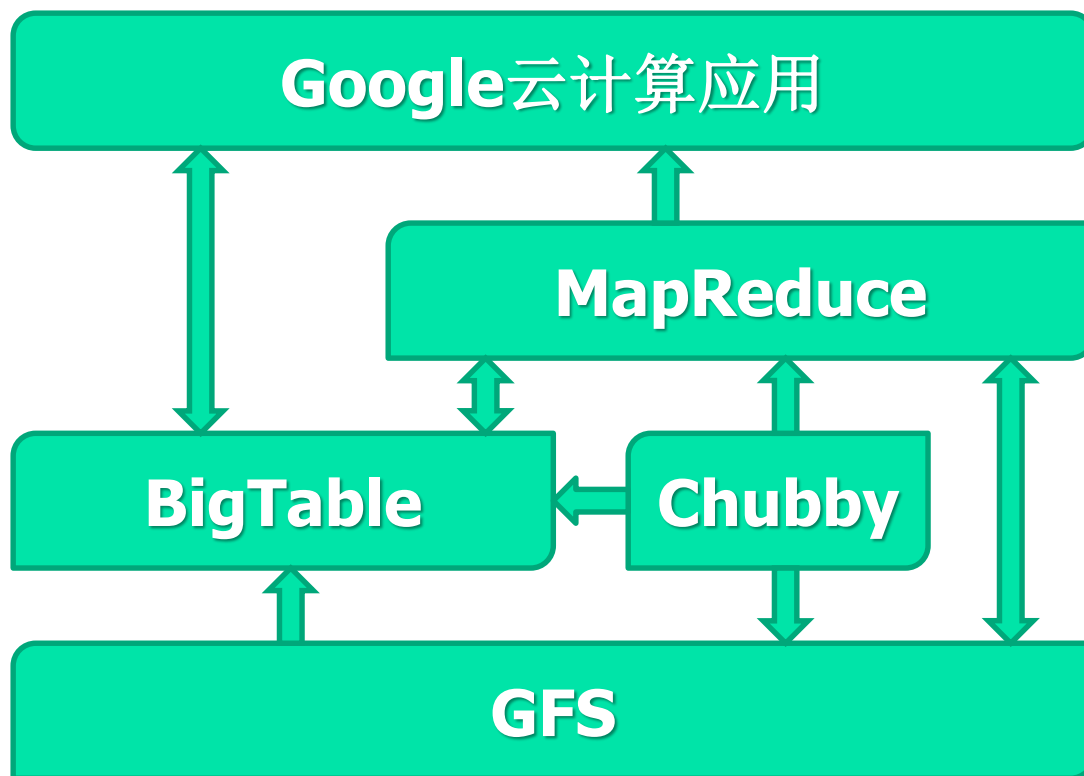
# Google云计算基础组件

- Google的云计算应用均依赖于四个基础组件
  - GFS, 分布式文件存储
  - BigTable, 结构化数据表
  - MapReduce, 并行数据处理模型
  - Chubby, 分布式锁 (GFS, BigTable, MapReduce都依赖)

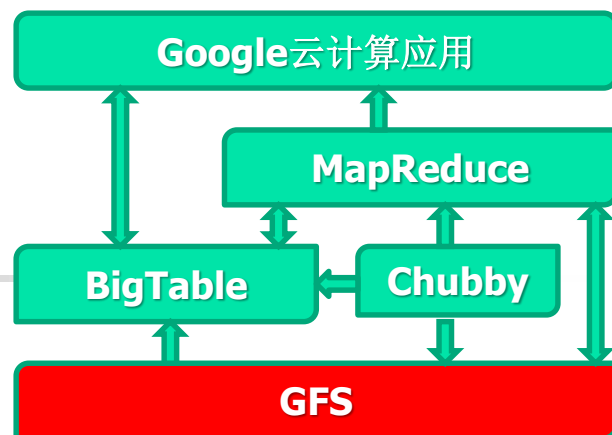


# 组件之间的关系

- 组件调用关系分析



# GFS



## ■ GFS的作用

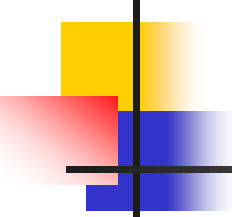
- 存储BigTable的子表文件
- 提供大尺寸文件存储功能

## ■ 文件被分成块（Chunk）

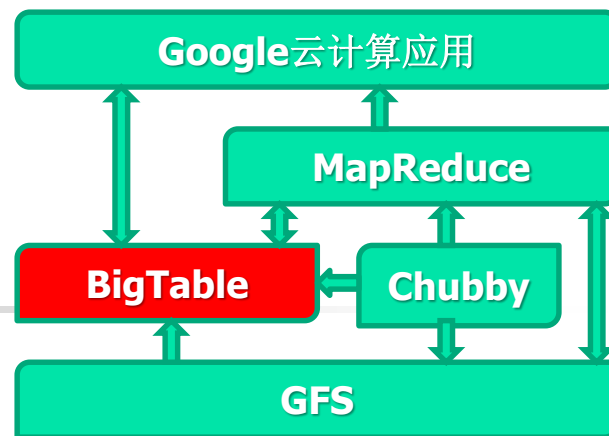
- 64MB/块
- 分布和复制在服务器上

## ■ 两个实体

- 一个Master
- 多个Chunkserver

- 
- 
- **Master** 维护所有文件系统元数据
    - 命名空间
    - 访问控制信息
    - 文件名到块的映射
    - 块的当前位置
  - **Master** 复制其数据以实现容错
  - **Master** 定期与所有**Chunkserver**通信
    - 通过心跳消息
    - 获取状态并发送命令
  - **Chunkserver**响应read/write请求和**Master**的命令

# Bigtable



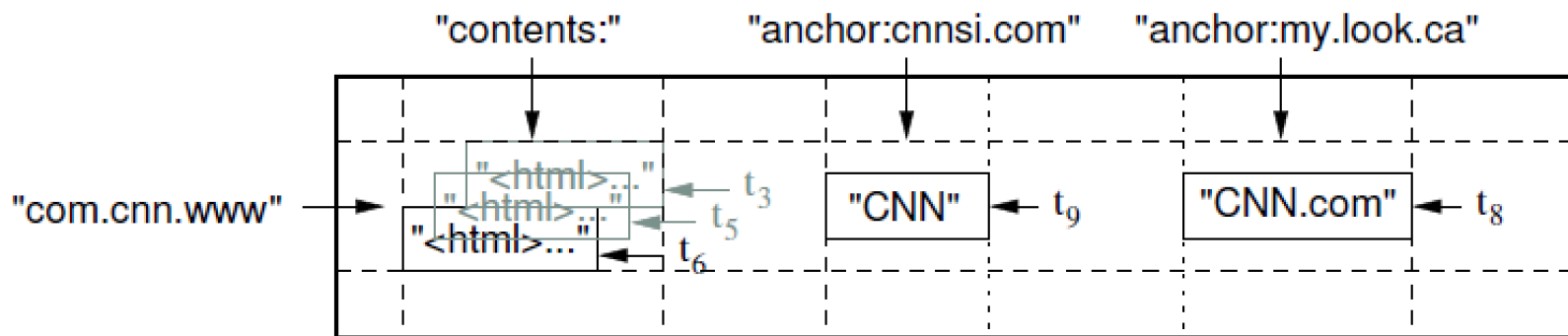
## ■ BigTable的作用

- 为Google服务提供数据结构化存储功能
  - Google Analytics
  - Google Finance
  - 个人搜索
  - Google Earth & Google Maps 等
- 为客户提供一个大的逻辑表视图
  - 逻辑表被分成片（tablets）并分布在 Bigtable 服务器上

## ■ 三个实体

- Client库
- 一个Master
- 多个Tablet服务器





- Row Key

- 行名是一个反向URL

- Column Key

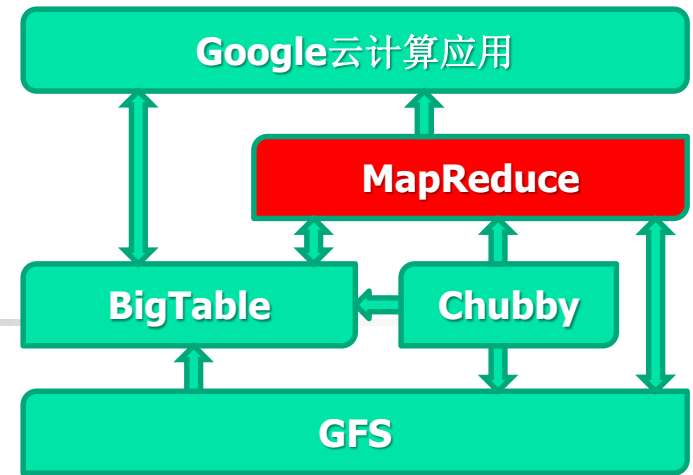
- Contents column family包含页面内容

- 例如，CNN主页有3个版本，分别是t3, t5和t6

- Anchor column family包含引用页面的所有anchor的文本

- 例如，cnnsi.com和my.look.ca都引用了CNN主页，所以包含anchor:cnnsi.com和anchor:my.look.ca两列，每个anchor只有1个版本

# MapReduce

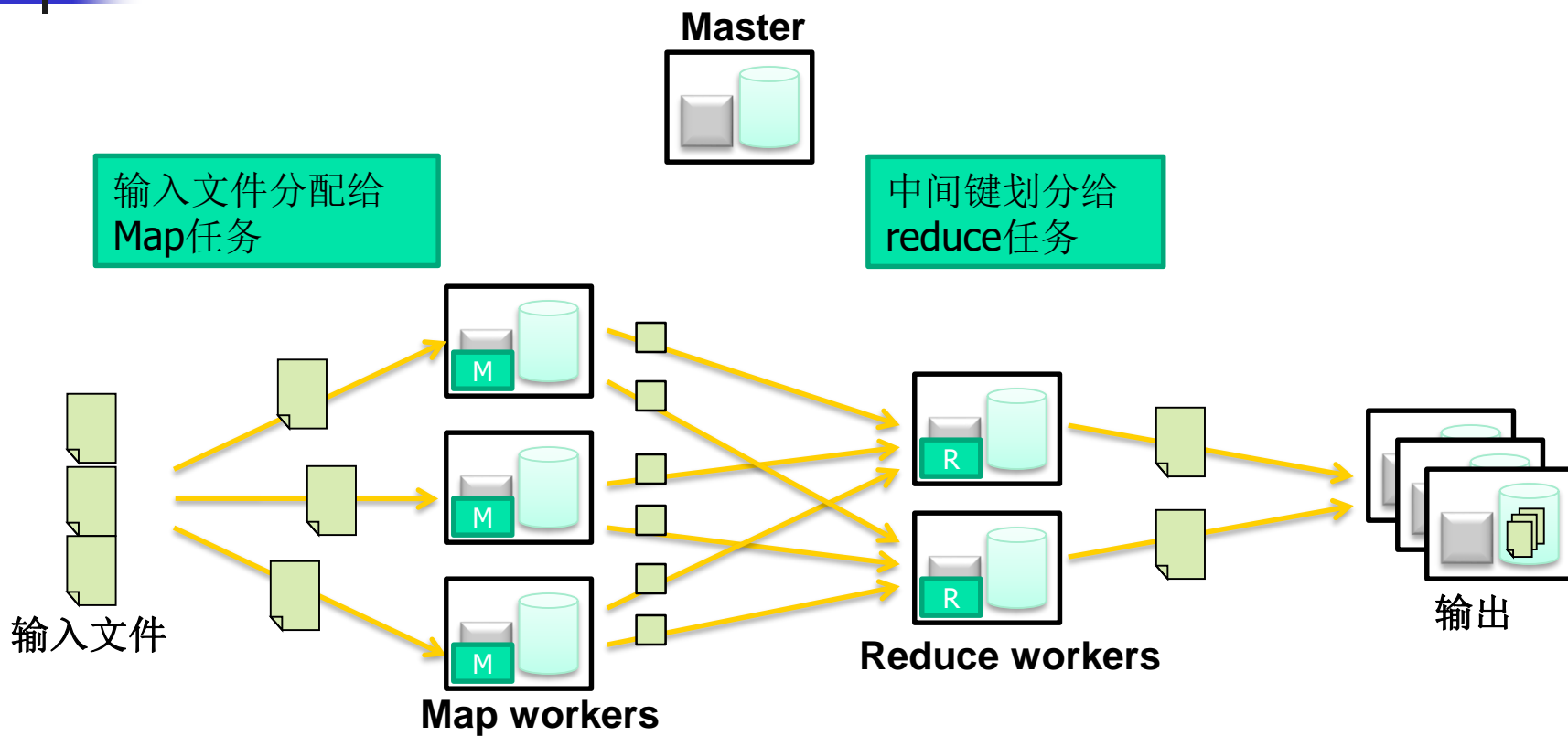
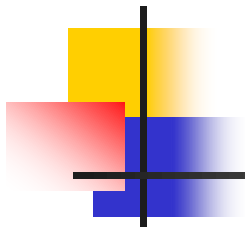


## ■ MapReduce的作用

- 对BigTable中的数据进行并行计算处理（如统计、归类等）
- 实现Map和Reduce两个功能
  - Map: 分配和处理任务（任务分解）
  - Reduce: 分类和归纳结果（结果聚合）

## ■ 执行框架

- 在一组服务器上执行Map和Reduce功能
- 一个Master
- 多个workers





# 为什么需要MapReduce?

- 计算问题简单，但求解困难
  - 待处理数据量巨大（**PB级**），只有分布在成百上千个节点上并行计算才能在可接受的时间内完成
  - 如何进行并行分布式计算？
  - 如何分发待处理数据？
  - 如何处理分布式计算中的错误？

简单的问题，计算并不简单！

# 为什么需要MapReduce?

Jeffery Dean设计一个新的抽象模型，使我们只要执行的简单计算，而将并行化、容错、数据分布、负载均衡的等杂乱细节放在一个库里，使并行编程时不必关心它们，这就是MapReduce。



Google MapReduce  
架构设计师  
Jeffrey Dean



# MapReduce示例：单词计数

- 使用MapReduce求解该问题
  - 定义Map和Reduce函数

```
function map(String name, String document):
```

```
    // name: document name
```

```
    // document: document contents
```

```
    for each word w in document:
```

```
        emit (w, 1)
```

任务分散

```
function reduce(String word, Iterator partialCounts):
```

```
    // word: a word
```

```
    // partialCounts: a list of aggregated partial counts
```

```
    sum = 0
```

```
    for each pc in partialCounts:
```

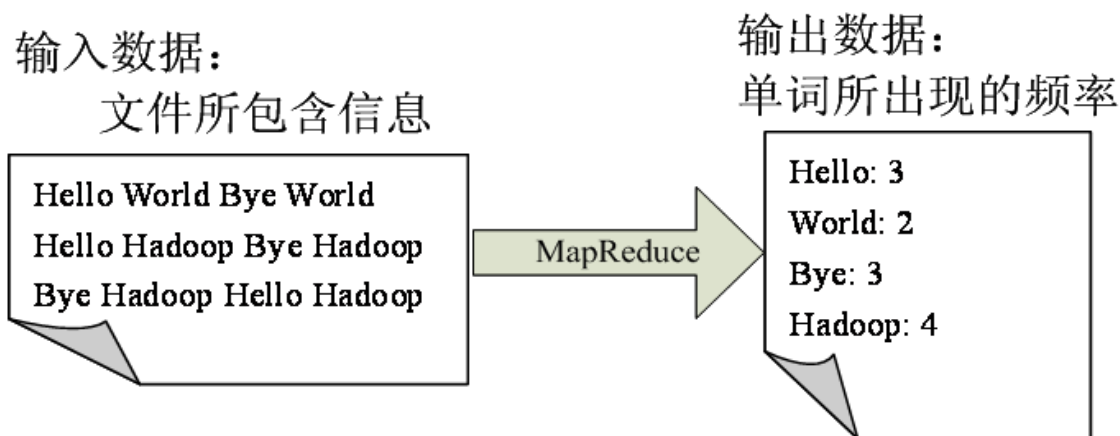
```
        sum += pc
```

```
    emit (word, sum)
```

结果聚合

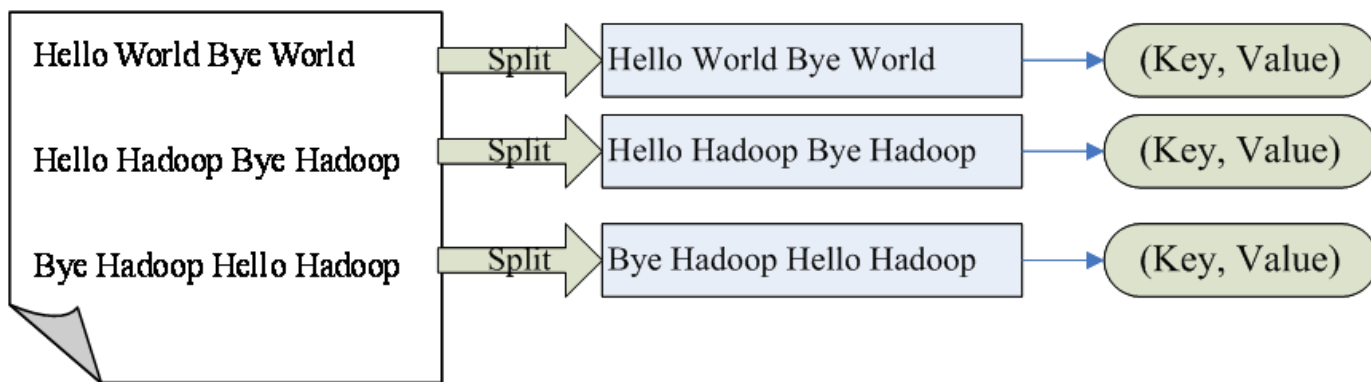
# MapReduce示例：单词计数

- 案例：单词记数问题(Word Count)
  - 给定一个巨大的文本（如1TB），如何计算单词出现的数目？



# MapReduce示例：单词计数

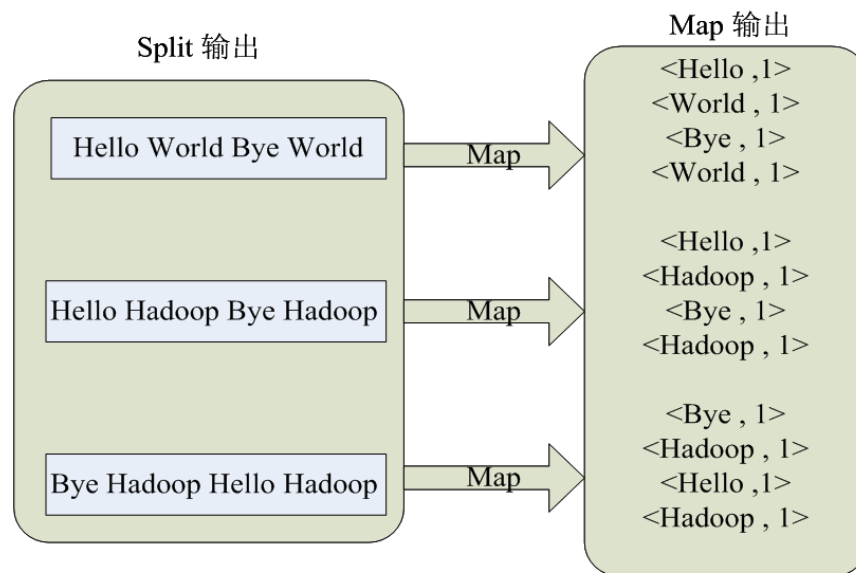
- 使用MapReduce求解该问题
  - Step 1: 自动对文本进行分割





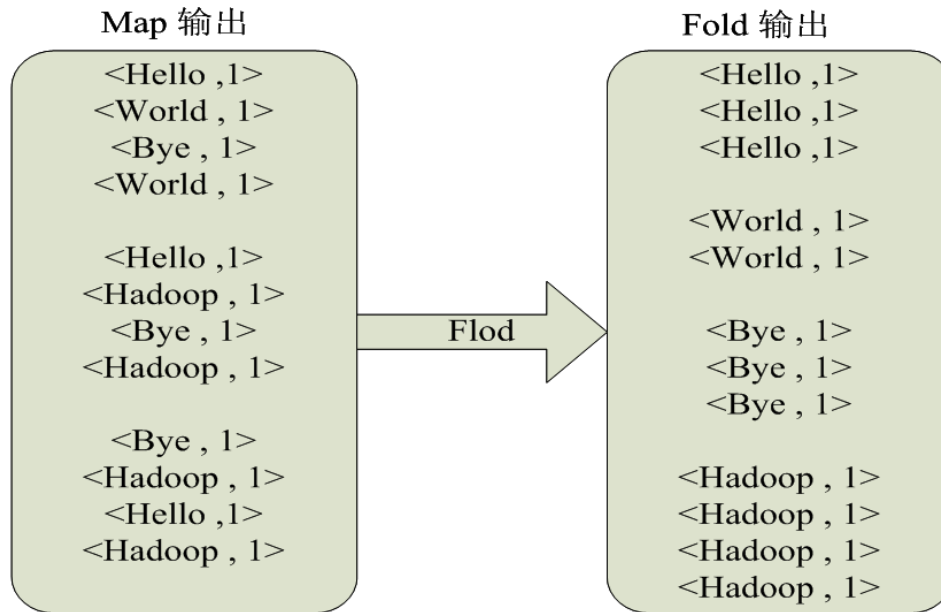
# MapReduce示例：单词计数

- 使用MapReduce求解该问题
  - Step 2:在分割之后的每一对<key,value>进行用户定义的Map进行处理，再生成新的<key,value>对



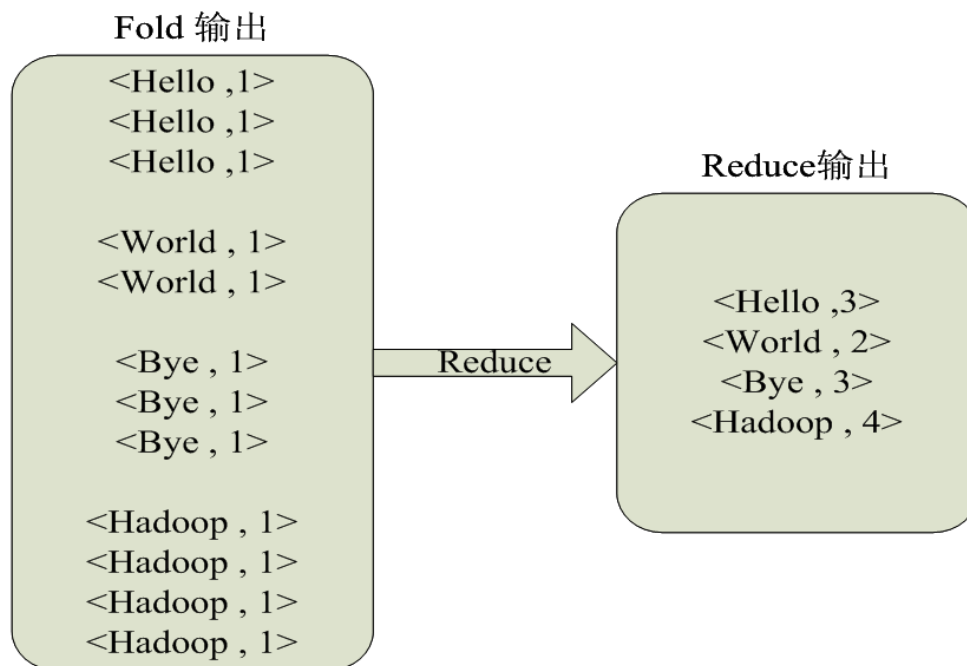
# MapReduce示例：单词计数

- 使用MapReduce求解该问题
  - Step 3:对输出的结果集归拢、排序（shuffle）



# MapReduce示例：单词计数

- 使用MapReduce求解该问题
  - Step 4:通过Reduce操作生成最后结果



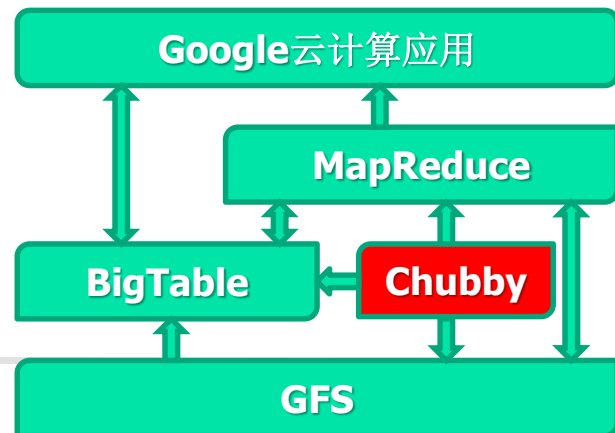


# GFS, BigTable和MapReduce的共同点

---

- 为什么**只有一个Master**?
  - 这种设计简化了系统复杂度
  - 主要用于处理元数据，减少单个主服务器的负载很重要
  - 无需处理一致性问题
  - 适合内存访问→速度快
- 主要问题：单点失效
  - 一个Primary和几个Backup
  - 从对等节点中选出Primary
- 选择Master时需要Chubby的锁服务

# Chubby



## ■ Chubby的作用

帮助开发人员处理系统中**粗粒度**的同步问题，特别是**选择Master**

### ■ 为什么是粗粒度锁？

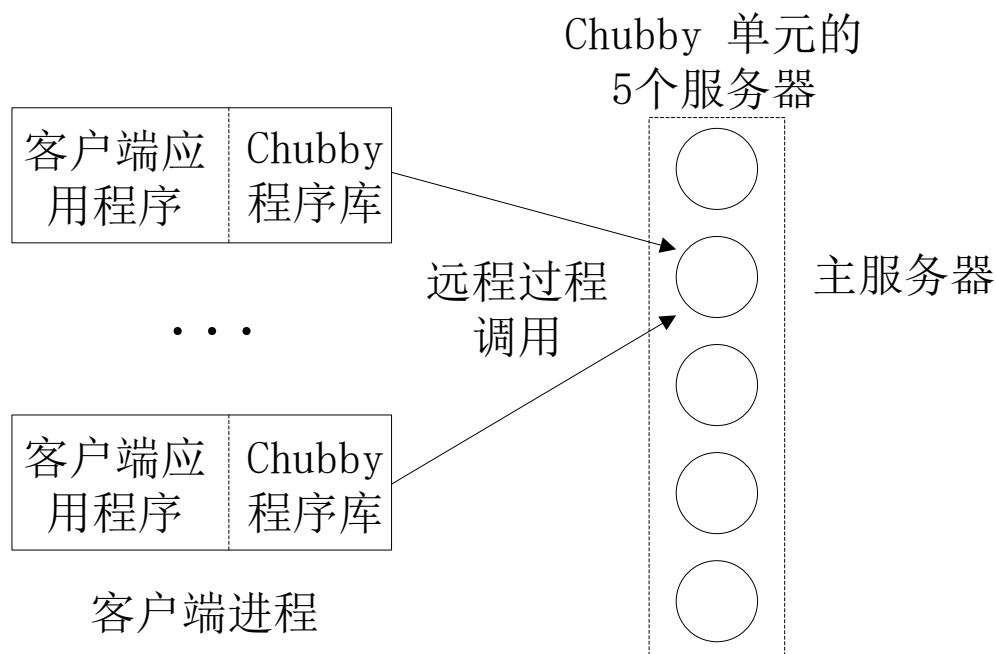
- **细粒度**锁通常只保持很短时间（几秒或更少），**粗粒度**锁持数小时和数天（**Master**的作用时间）

### ■ 如何选择Master？

- 潜在的**Master**尝试在Chubby上创建一个锁
- 第一个获得锁的成为**Master**

- 
- 
- **GFS使用Chubby**
    - 指定Master服务器
    - 存储一小部分元数据
  - **BigTable使用Chubby**
    - 选择一个Master
    - 允许Master发现它控制的其它服务器
    - 允许Client发现Master
    - 存储一小部分元数据

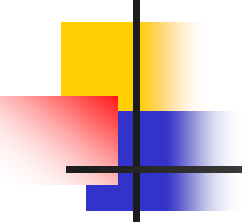
# Chubby的系统架构



Chubby单元由一小部分服务器（通常为5台）组成，这些服务器称为副本服务器，它们放置位置不同，以减少相关故障的可能性（例如，在不同的机架中）。

副本服务器使用分布式共识协议选举主服务器。主副本服务器必须从大多数副本服务器获得投票，并保证这些副本服务器不会在几秒钟的间隔内选出另一个主副本服务器，这称为主租约。

副本服务器维护简单数据库的副本，但是只有主副本服务器会启动数据库的读取和写入。所有其他副本服务器仅复制使用共识协议发送的来自主服务器的更新。

- 
- 
- The **google file system**, SOSP'03
  - **MapReduce**: simplified data processing on large clusters, OSDI'04
  - The **chubby** lock service for loosely-coupled distributed system, OSDI'06
  - **Bigtable**: a distributed storage system for structured data, OSDI'06





# GFS 动机

---

- 首先，**组件失效**被认为是**常态事件**，而不是意外事件。
  - **GFS**包括几百甚至几千台普通的廉价设备组装的存储机器，同时被相当数量的客户机访问。
  - **GFS**组件的数量和质量导致在事实上，任何给定时间内都有可能发生某些组件无法工作，某些组件无法从它们目前的失效状态中恢复。



# GFS 动机

---

- 以通常的标准衡量，文件**非常巨大**。
  - 数**GB**的文件非常普遍
  - 当我们经常需要处理快速增长的、并且由数亿个对象构成的、数以**TB**的数据集时，采用管理数亿个**KB**大小的小文件的方式是非常不明智的，尽管有些文件系统支持这样的管理方式。
  - 设计的假设条件和参数，比如**I/O操作**和**Block**的尺寸都需要重新考虑。



# GFS 动机

---

- 绝大部分文件的**修改**是采用在文件尾部**追加数据**，而不是覆盖原有数据的方式。
  - 对文件的随机写入操作在实际中几乎不存在。
  - 一旦写完之后，对文件的操作就只有读，而且通常是按顺序读。
  - 对于这种针对海量文件的访问模式，**Client**对数据块缓存是没有意义的，数据的追加操作是性能优化和原子性保证的主要考量因素。



# GFS 动机

---

- 应用程序和文件系统**协同**设计以提高整个系统的灵活性。
  - 放松了对一致性模型的要求。
  - 原子性的记录追加操作，从而保证多个**Client**能够同时进行追加操作，不需要额外的同步操作来保证数据的一致性。



# GFS 假设

---

- 系统由许多廉价的普通组件组成，**组件失效是一种常态。**
  - 系统必须持续监控自身的状态
  - 必须能够迅速地侦测、冗余并恢复失效的组件
- 系统存储一定数量的大文件。
  - 预期会有几百万文件，大小通常在**100MB**或者以上。
  - 数个**GB**大小的文件也是普遍存在，需有效管理。
  - 必须支持小文件，但是不针对小文件做专门的优化。



# GFS 假设

---

- **大规模的流式读取**和小规模的随机读取
  - 大规模的流式读取通常一次读取数百**KB**的数据，更常见的是一次读取**1MB**甚至更多的数据。
  - 来自同一个**Client**的连续操作通常是读取同一个文件中连续的一个区域。
- 大规模的、顺序的、**数据追加**方式的写操作。
- 系统必须高效的、行为定义明确的实现多客户端**并行追加**数据到同一个文件里的语义。
- 高性能的**稳定网络带宽**远比**低延迟**重要。

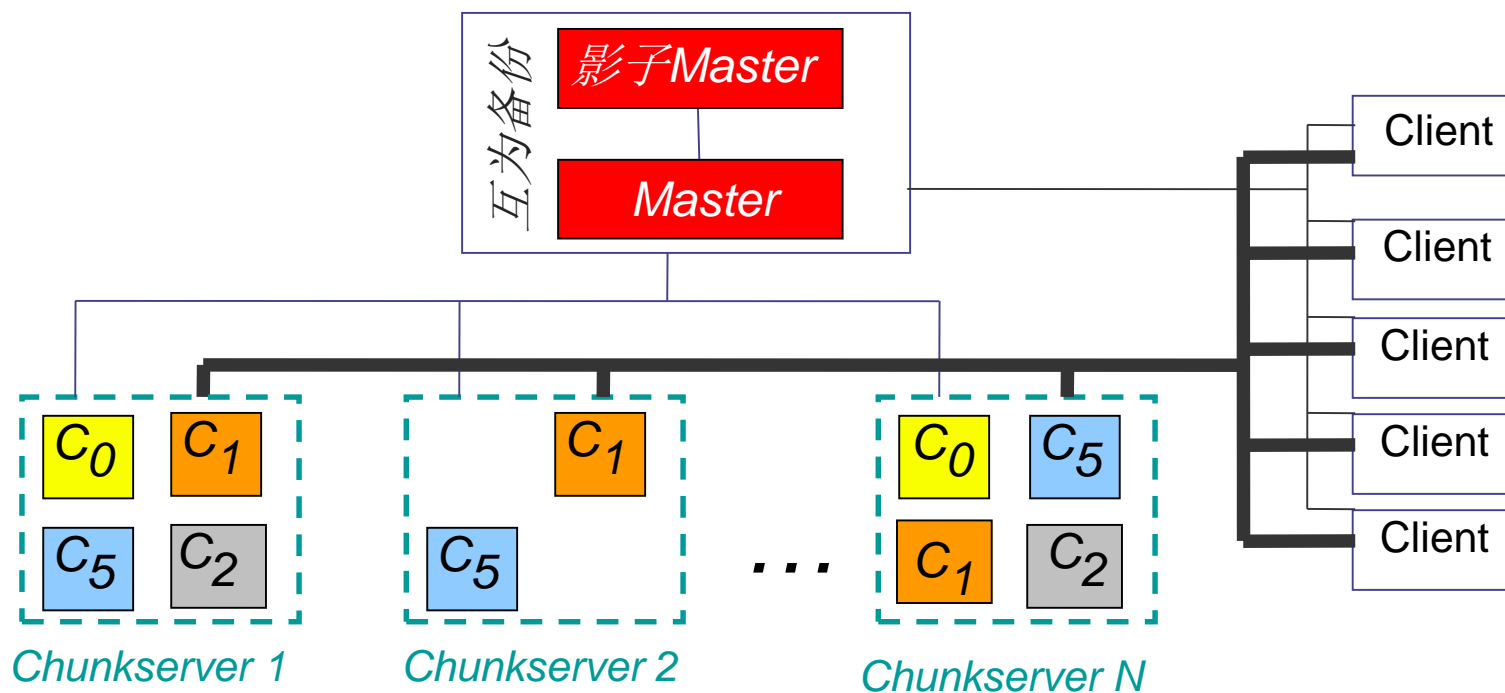


# GFS的设计思想

---

- 文件以**数据块 (Chunk)** 的形式存储
  - 数据块大小固定，每个数据块拥有句柄。
- 利用副本技术保证可靠性
  - 每个数据块至少在3个Chunkserver上存储副本。
  - 每个数据块作为本地文件存储在Linux文件系统中。
- **Master**维护所有文件系统的元数据 (metadata)
  - 每个GFS簇只有一个Master。
  - 利用周期性的心跳消息向Chunkserver发送命令和收集状态

# Google文件系统(GFS)



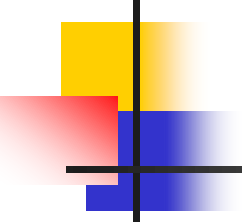




# GFS的设计思想

---

- Master服务器在不同的数据文件里保持**元数据**。数据以64MB为单位存储在文件系统中。Client与Master服务器通讯在文件上做元数据操作并且找到包含用户需要的数据
  - 只存储元数据，不存储文件数据，不让**磁盘容量**成为Master瓶颈；
  - 元数据会存储在磁盘和内存里，不让**磁盘IO**成为Master瓶颈；
  - 元数据大小内存完全能装得下，不让**内存容量**成为Master瓶颈；
  - 所有数据流，数据缓存，都不通过Master，不让**带宽**成为Master瓶颈；
  - 元数据可以缓存在Client，每次从Client本地缓存访问元数据，只有元数据不准确的时候，才会访问Master，不让**CPU**成为成为Master瓶颈

- 
- 
- **Chunkserver**在硬盘上存储实际数据。
  - 每个块跨**3**个不同的**Chunkserver**备份以创建冗余避免服务器崩溃。
  - 一旦被**Master**服务器指明，**Client**会直接从**Chunkserver**读取文件。

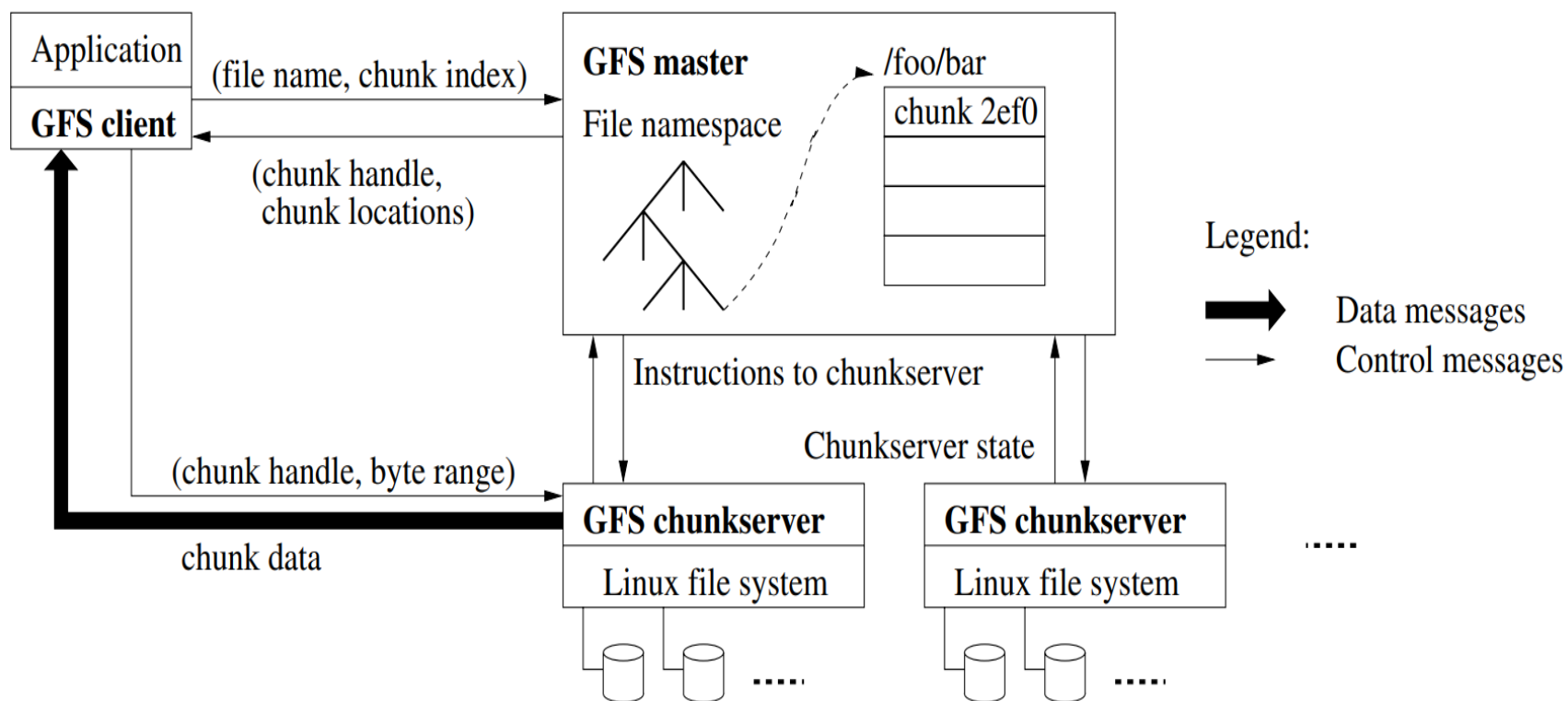


# 缓存

---

- 无论是**Client**还是**Chunkserver**都不需要缓存**文件**数据（不过，**Client**会缓存元数据）。
- **Client** 缓存数据几乎没有作用，因为大部分程序要么以流的方式读取一个巨大文件，要么工作集太大而无法被缓存。
- 无需考虑缓存相关的问题也简化了**Client**和整个系统的设计和实现。
- **Chunkserver** 也不需要缓存文件数据，因为**Linux**操作系统的文件系统缓存会把经常访问的数据缓存在内存中。

# GFS的体系结构





# GFS的体系结构

---

- 系统的流程从**Client**开始
  - **Client**以块偏移量制作目录索引并发送请求
  - **Master**收到请求通过块映射表映射反馈**Client**
  - **Client**获得块句柄和块位置，将文件名和块的目录索引缓存，并向**Chunkserver**发送请求
  - **Chunkserver**回复请求传输块数据。



# GFS的体系结构

---

- **Master**是在独立的主机上运行的一个**进程**
- 存储的**元数据**信息：
  - 文件命名空间
  - 文件到数据块的映射信息
  - 数据块的位置信息
  - 访问控制信息
  - 数据块版本号



# GFS的体系结构

---

## ■ 内存数据结构

- **Master**可以在后台定期扫描整个系统状态。
  - 块垃圾收集。
  - 为平衡负载和磁盘空间而进行的块迁移。
  - **Chunkserver**出现故障时的副本复制。
- 整个系统的容量受限于**Master**的内存，每个块（**64MB**）保留少于**64B**的元数据。
- 若要支持更大的文件系统，只需增加一些保存元数据的内存即可完成扩展，这种设计简单、可靠、高效和灵活。



# GFS的体系结构

---

文件数据块：64MB的大数据块

■ 优点：

- 减少Master上保存的元数据的规模，使得可以将元数据(metadata) 放在内存中
- Client在一个给定块上很可能执行多个操作，和一个Chunkserver保持较长时间的TCP连接可以减少网络负载
- 在Client中缓存更多的块位置信息

■ 缺点：

- 一个文件可能只包含一个块，如果很多Client访问该文件，存储块的Chunkserver可能会成为访问热点





# GFS的体系结构

---

- 块位置信息
  - **Master**并不为**Chunkserver**的所有块的副本保存一个不变的记录
  - **Master**在启动时或者在有新的**Client**加入这个簇时通过简单的查询获取这些信息。
- **Master**可以保持这些信息的更新，因为它控制所有块的放置并通过心跳消息监控。



# GFS的体系结构

---

- Master和Chunkserver之间的通信
  - 定期地获取状态信息
    - Chunkserver是否关闭？
    - Chunkserver上是否有硬盘损坏？
    - 是否有副本出错？
    - Chunkserver维护哪些块的副本？
    - Master发送命令给Chunkserver:
      - 删除已存在的块
      - 创建新的块



# GFS的体系结构

---

## ■ 操作日志

- 操作日志包含了对**metadata**所作的修改的历史记录，被复制在多个远程**Chunkserver**上。
- 它可以从本地磁盘装入最近的检查点来恢复状态。
- 它作为逻辑时间基线定义了并发操作的执行顺序。
- 文件、块以及它们的版本号都由它们被创建时的逻辑时间而唯一地、永久地被标识。
- **Master**可以用操作日志来恢复它的文件系统的状态。

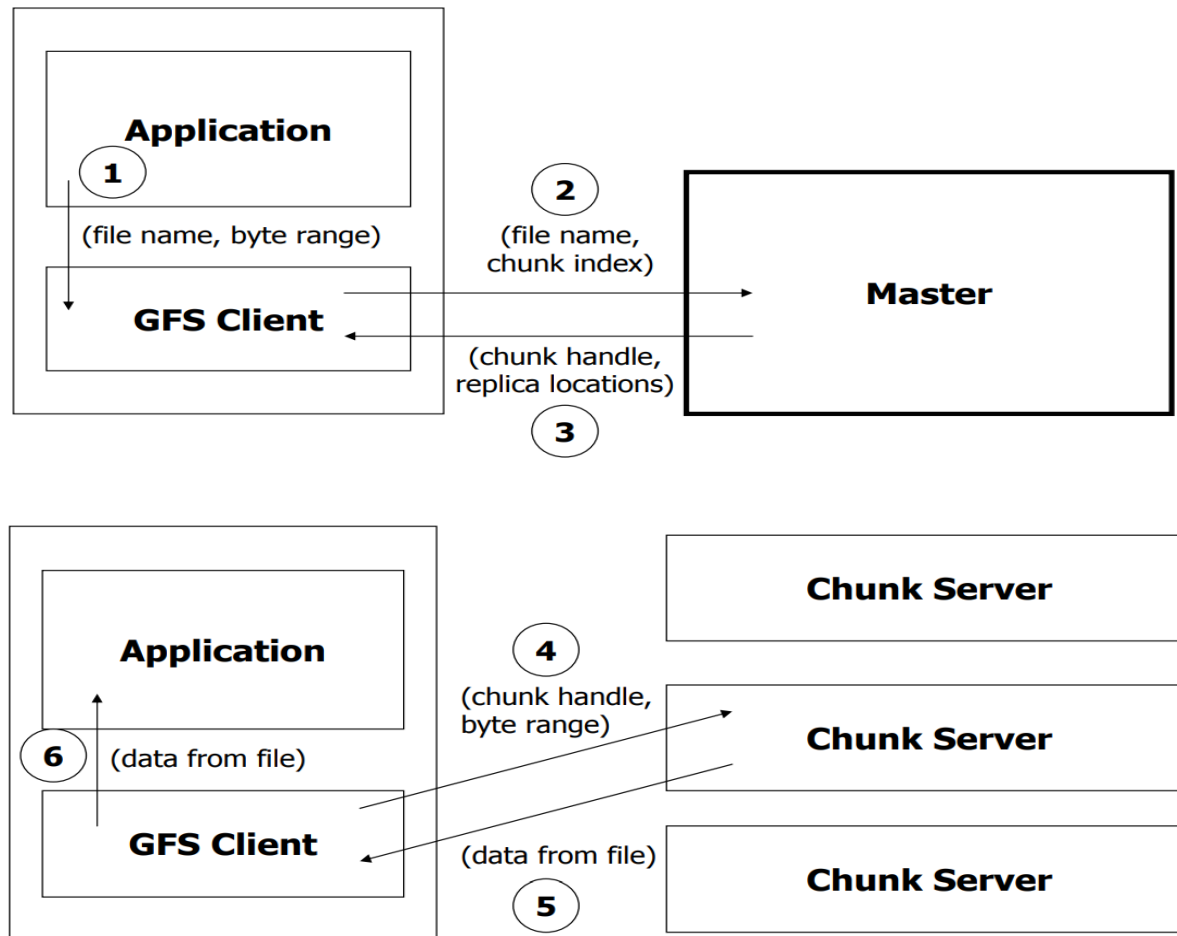


# GFS的体系结构

---

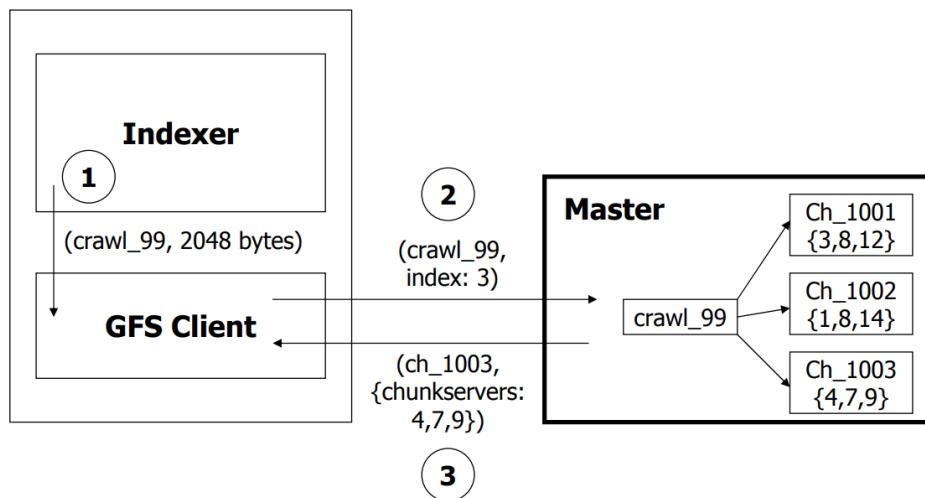
- 服务请求：
  - Client 从Master检索元数据（metadata）
  - 单个Master并不会成为瓶颈，因为Master仅提供查询数据块所在的Chunkserver以及详细位置
  - Client直接与Chunkserver通讯，传输数据块。

# GFS的读操作



# GFS的读操作

计算数据块位置信息：（假设：文件位置在134,250,297 bytes）



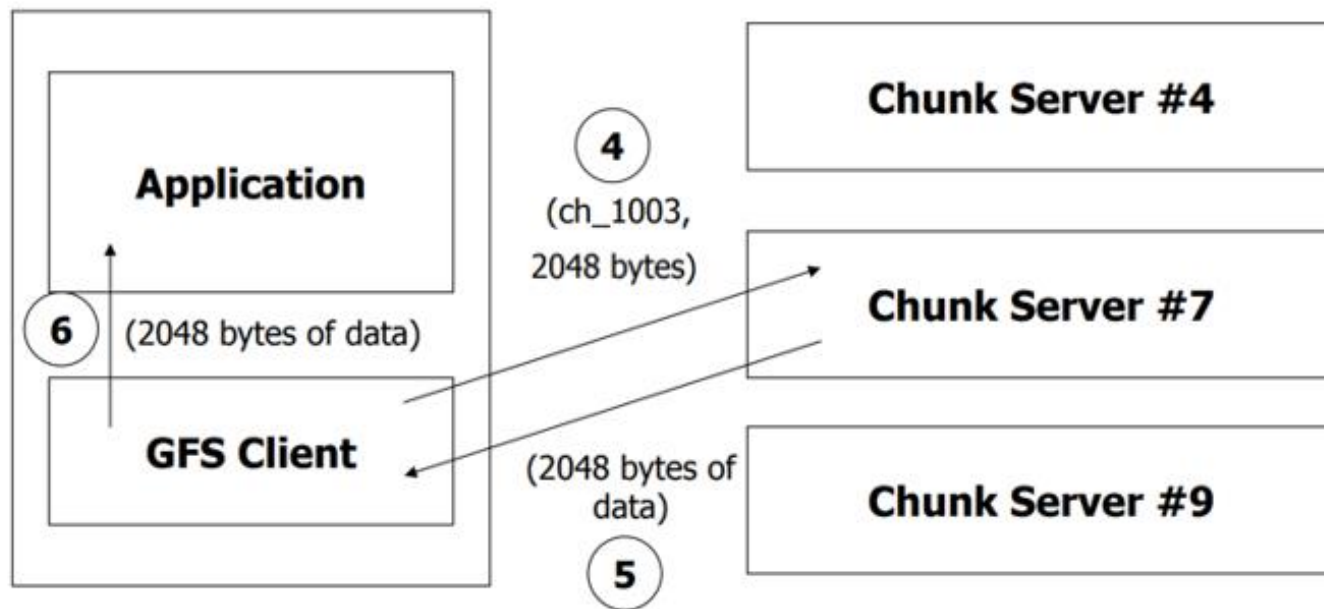
块大小=64MB

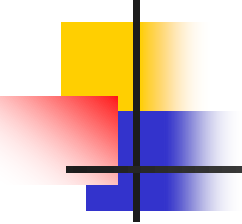
$64\text{MB} = 1024 * 1024 * 64 \text{ bytes}$   
 $= 67,108,864 \text{ bytes}$

$134,250,297 \text{ bytes} =$   
 $67,108,864 * 2 + 32,569 \text{ bytes}$

所以，Client的位置索引是3

# GFS的读操作

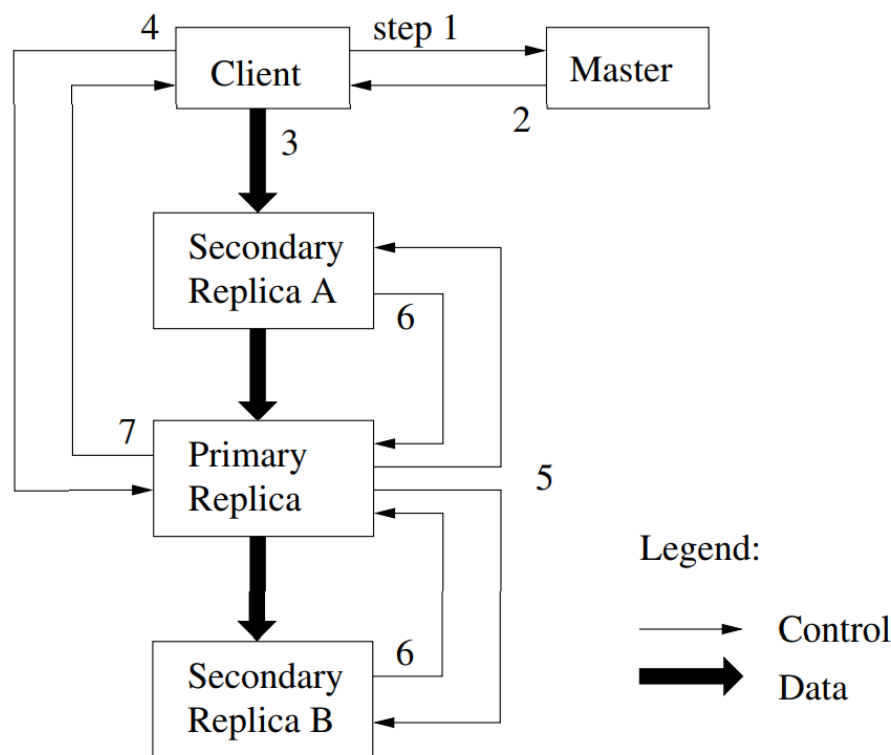


- 
1. 应用程序发起读取请求。
  2. **Client**从（文件名，字节范围）->（文件名，组块索引）转换请求，并将其发送到**Master**。
  3. **Master**以块句柄和副本位置（即存储副本的**Chunkserver**）作为响应。
  4. **Client**选择一个位置，然后将（块句柄，字节范围）请求发送到该位置。
  5. **Chunkserver**将请求的数据发送到**Client**。
  6. **Client**将数据转发到应用程序。

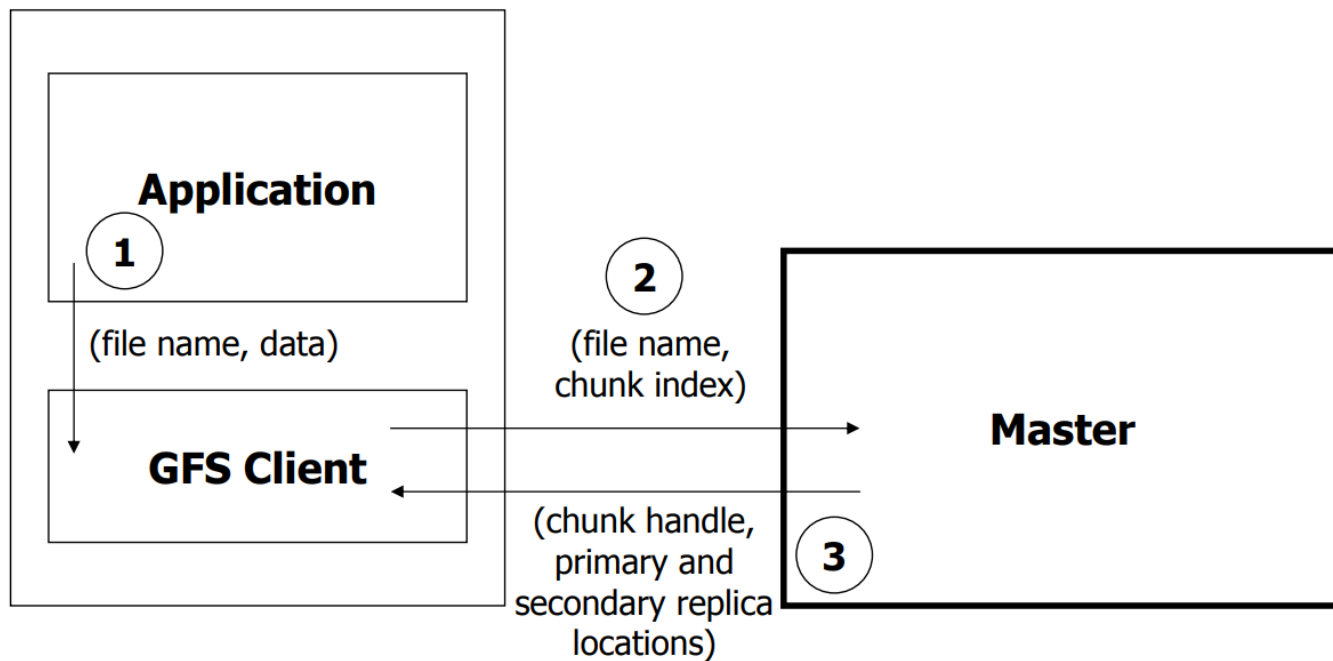


# GFS的互斥操作

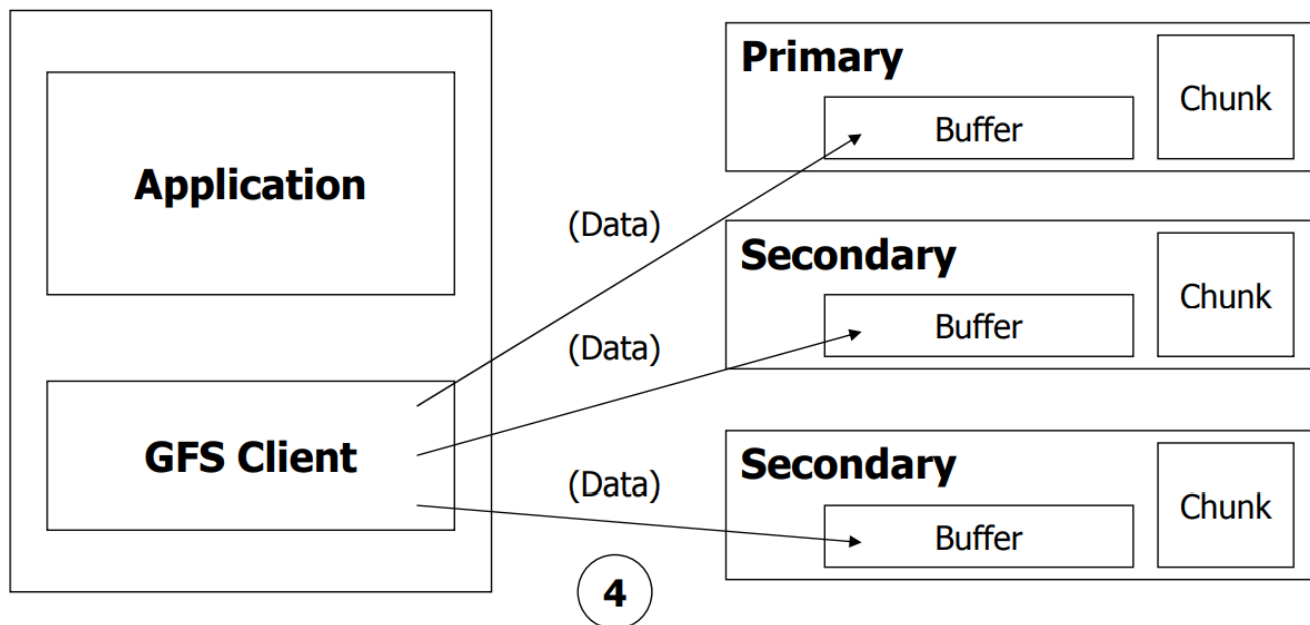
- 互斥：任何的写或者追加操作
  - 数据需要被写到所有的Replica上
  - 当多个Client请求修改操作时，保证同样的次序。



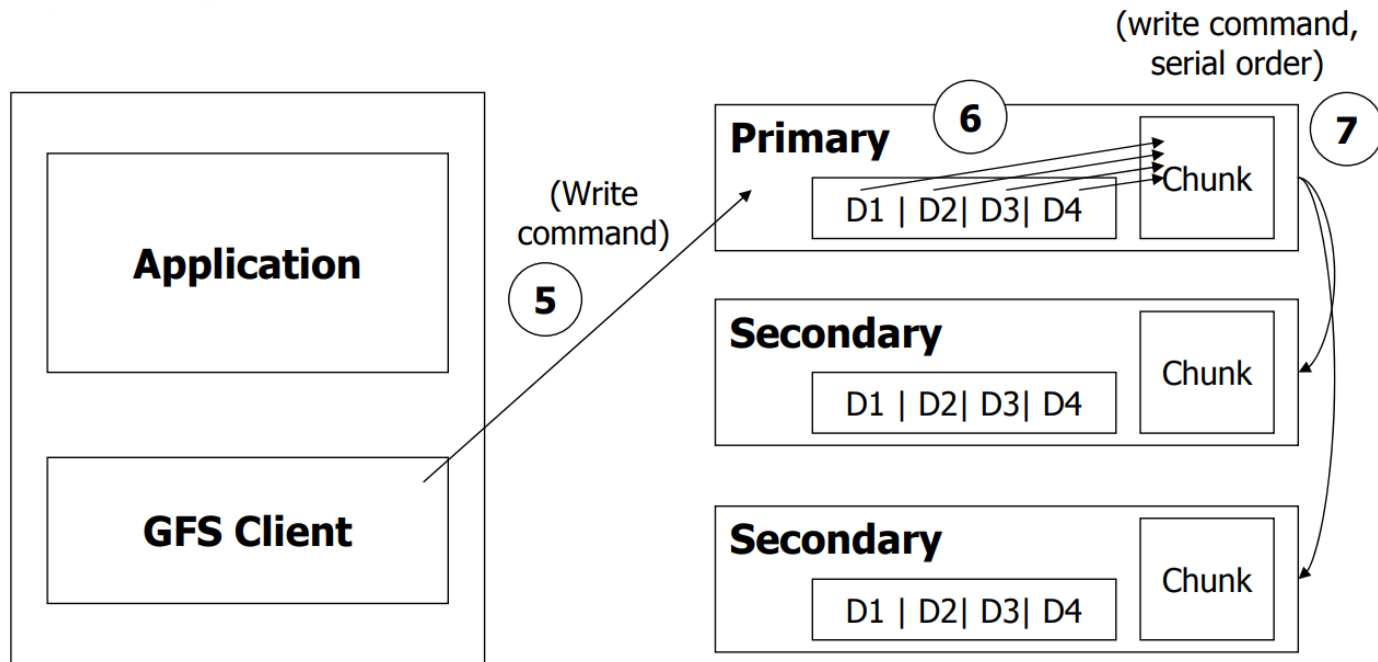
# GFS的互斥操作



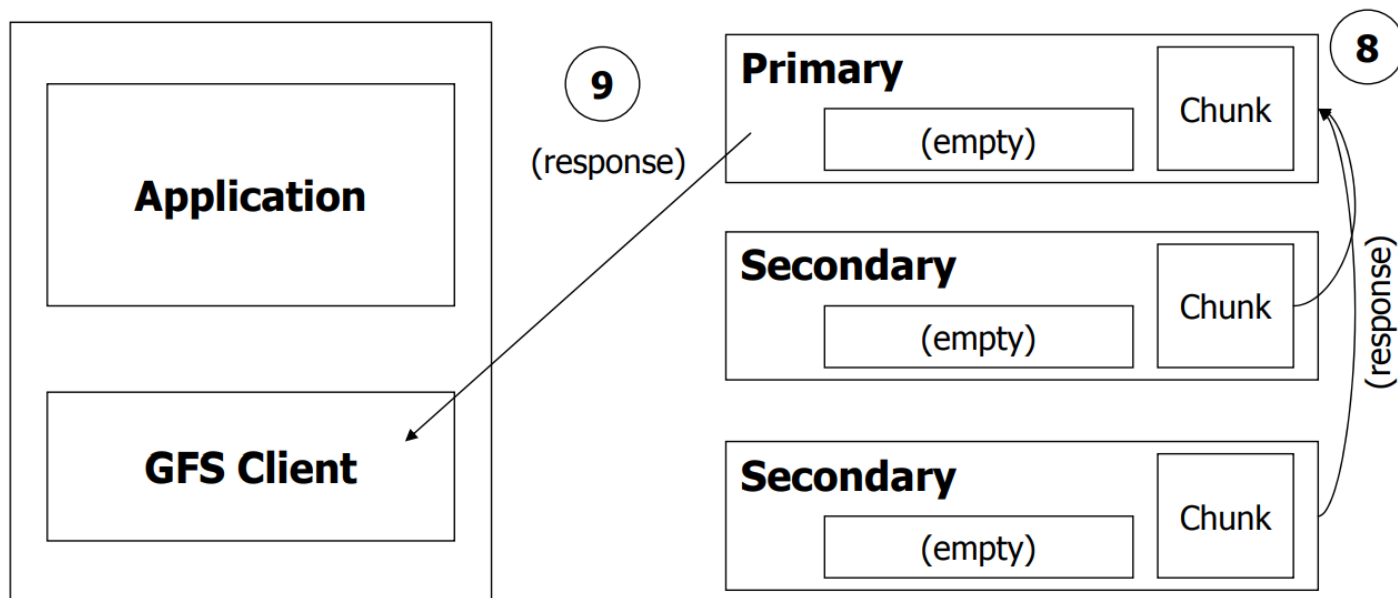
# GFS的互斥操作



# GFS的互斥操作



# GFS的互斥操作





# GFS的互斥操作

---

1. Client发送请求到Master;
2. Master返回块的句柄和Replica位置信息;
3. Client将写数据推送给所有Replica（可以根据网络拓扑）;
4. 数据存储在Replica的缓存中;
5. Client发送写命令到Primary;
6. Primary给出写的次序（可能请求来自多个Client）;
7. Primary将该次序发送给Secondaries;
8. Secondaries响应Primary;
9. Primary响应Client。



# Append操作

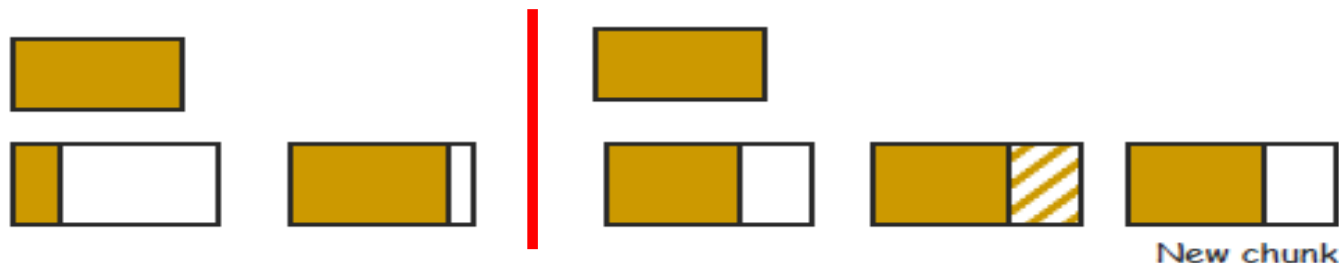
---

谷歌文件系统中非常重要的操作：

- 把多个主机的结果合并到一个文件中。
- 将文件组织成生产者消费者队列。
- **Clients**可以并发读。
- **Clients**可以并发写。
- **Clients**可以并发地执行添加操作。

# Append操作（续）

1. Client将数据推送给所有Replica，然后向Primary发送请求
2. Primary检查Append是否会导致该块超过64MB
  - 如果小于64MB，按正常情况处理。
  - 如果超过64MB，将该块扩充到最大范围（写0），并要求所有Secondary做同样的操作，同时通知Client该操作需要在下一个块上重新尝试。



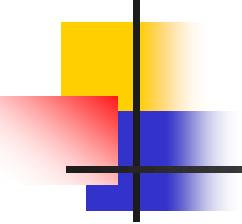


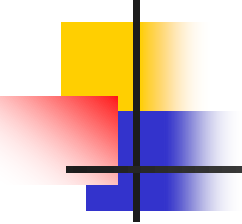


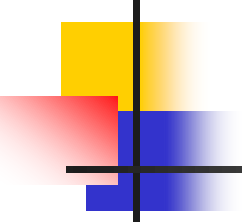
# 一致性模型

- 并发的写将导致一致性问题。不同的Client对同一文件区域执行写。
  - 一致：所有的Client读取**相同**数据。
  - 确定：所有的Client读取**有效**数据。

	Write	Record Append
串行化成功	确定的 ( <b>defined</b> )	确定的，但穿插不一致 ( <b>defined intersperse with inconsistent</b> )
并发成功	一致但不确定 ( <b>consistent but undefined</b> )	
失败	不一致 ( <b>inconsistent</b> )	

- 
- 
- **Serial success:** 当 多个Client 串行写时，写入并没有相互干扰，所有Client可以看到明确的写的过程，写的区域是 **Defined**，也是**Consistent**

- 
- **Concurrent success:** Primary决定Client写的顺序。当多个Client并发写多个存在交叉的Chunk时，由于Primary之间并不通信，不同Primary可能选择不同的Client写顺序。如果执行成功，会导致所有Client看到相同的数据 (**Consistent**)，但数据无效 (**Undefined**)

- 
- **Record append:** Primary根据当前文件大小决定写入的offset, GFS不保证所有Replica上字节都相同, 只保证至少一次写 (**at-least-once semantics**), 因此副本的同一个块可能包含重复的数据, Append成功的区域数据是**Defined**, 但Append失败重试会导致介于中间的区域是**Inconsistent** (也是**Undefined**)



# 数据完整性

---

- **Writer**为每条记录增加额外的校检和信息用于验证记录的有效性。
- 一个数据块被分为**64KB**大小的小块，每个小块有一个**32bit**的校检和。
- 读取时，**Reader**先验证数据块的校检和，检测数据块的错误和重复。



# 容错

---

- 恢复：不管如何终止服务，**Master**和**Chunkserver**都会在几秒钟内恢复状态和运行。
- 数据块备份：每个数据块都会被备份到放到不同机架上的多个**Chunkserver**上。
- **Master备份**：为确保可靠性，**Master**的状态、操作记录和检查点都在多台机器上进行了备份。一个操作只有在**Chunkserver**硬盘上刷新并被记录在**Master**和其备份的上之后才算成功。如果**Master**或是硬盘失败，系统监视器会发现并通过改变域名启动一个影子**Master**，而**Client**并不会发现**Master**改变。



# 创建、复制、平衡数据块

---

- 当**Master**创建新数据块时，如何放置新数据块要考虑以下因素：
  - 放置在磁盘利用率低的**Chunkservers**
  - 控制在一个**Chunkserver**上的“新创建”次数
  - 把数据块放置于不同的机架上

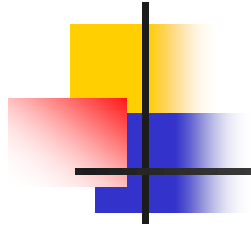


# 垃圾收集

---

- 文件删除后，**GFS** 不会立即回收可用的存储空间
  - 删除文件会被重命名为包含删除时间戳的隐藏名
  - 在**Master**定期扫描文件系统命名空间期间（常规后台活动），如果隐藏文件已存在超过**3**天（间隔可配置），删除此隐藏文件
- 存储回收比立即回收具有以下优点：
  - 在组件故障常见的大规模分布式系统中，存储回收简单可靠
  - 将存储回收合并到**Master**常规的后台活动，实现成本摊销
  - 回收存储的延迟可以防止意外、不可逆删除





END