《分布式系统开发》课程大作业

姓　名　葛玉菲

专业班级 软件 2003 班

学　号　04203103

日　期　2022 年 12 月 30 日

一、实验题（web 集群应用，60 分）

题目叙述：某影院公司开发了一套基于 tomcat 服务器，面向互联网用户的在线票务管理系统，向用户提供在线订票、退票和改票服务。在运营初期，系统部署在一台物理服务器上，位于公司局域网内。随着公司业务的快速发展，在线用户人数呈现迅猛增长，在用户访问高峰时段对票务管理系统造成了巨大的压力，通过提升软件和物理服务器的配置，仍旧无法满足的用户需求，该服务器不能保证良好的用户体验，也无法保证公司不断增长的业务需求。因此，公司管理层听取了技术部门的意见，决定将此票务管理系统部署在若干台物理服务器上，每台服务器上运行一个 tomcat,并在其中一台物理服务器部署反向代理，即采用负载均衡调度分配和副本复制的服务器集群方案来提高整个系统的性能和可靠性。

问题：

1、在该公司业务量扩张过程中，从分布式系统软件有关性能和可靠性技术的角度，试分析采用何种方案维护系统，并简要说明方案的优缺点。（15 分）

答案：分布式系统的出现，就是为了用廉价的、普通的机器解决单个计算机处理复杂、大规模数据和任务时存在的性能问题、资源瓶颈问题，由此看来，性能、资源、可用性和可扩展性时分布式系统的重要指标，它们是分布式系统的“三围”。

一、有关性能

高性能。正是因为单个节点的 scale up 不能完成任务，因此我们才需要 scale out，用大量的节点来完成任务，影院系统中的分布式方案的理想目标是任务与机器节点按一定比例线性增长。

性能指标，主要用于衡量一个系统处理各种任务的能力。不同的系统、服务要达成的目标不同，关注的性能自然也不同。常见的性能指标，包括吞吐量(Throughput)、响应时间(Response Time)和完成时间(Turnaround Time)

(1) **吞吐量。**指系统在一定时间内可以处理的任务数。常见的吞吐量指标有 QPS（Queries Per Second）、TPS（Transactions Per Second）和 BPS（Bits Per Second）。如果要提升分布式系统性能，就需要从这几方面入手.

1. **QPS。**即查询数秒，用于衡量一个系统每秒处理的查询数，并以此作为系统处理能力的标准。由公式 QPS = 并发数/平均响应时间 可以看出，要提升 QPS，需要两个方向的努力。

增加并发数。比如增加 tomcat 并发的线程数，数据库的连接数使后端服务尽量无状态化，可以更好的支持横向扩容；调用链路上的各个系统和服务尽量不要单点；PRC 调用的尽量使用线程池，预先建立合适的连接数。

减少平均响应时间。请求尽量提前结束，使压力从后端系统转移，在各个层面上加上缓存；放行适当流量；减少调用链；优化程序；减少网络开销，适当使用长链接；优化数据库，建立索引

2. TPS。即事务数每秒，用于衡量一个系统每秒处理的事务数。这个指标对应写操作。

DB **优化**。筛选率低的字段不需要索引，数据库读写分离。

业务优化。Log4j2 异步，增加缓存，简化业务步骤等。

(2) 响应时间。指系统相应一个请求或输入需要花费的时间。

1. 串行转并发，使用线程池并发处理请求

2. 同步转异步，使用消息队列

3. 使用缓存，读写分离

4. 减少日志打印，留意日志打印中的序列化、长报文或者异步打印日志。

5. 优化 SQL，批量查询。

(3) **本地化计算**。

在分布式系统中，数据的分布方式也深深影响着计算的分布方式。在分布式系统中计算节点和保存计算数据的存储节点可以在同一台物理机器上，也可以位于不同的物理机器。如果计算结点和存储节点位于不同的物理机器，则计算的数据需要通过网络传输，此种方式的开销很大，甚至网络带宽会成为系统的总体瓶颈。另一种思路是，将计算尽量调度到与存储节点在同一台物理机器上的计算节点上进行，这称之为本地化计算。

二、有关可靠性

高可用，是分布式系统架构设计中必须考虑的因素之一，它通常是指系

统不间断对外提供服务的能力。

那么如何保障系统的高可用呢？

1. **冗余**。我们都知道，单点是系统高可用的大敌，单点往往是系统高可用最大的风险和敌人，应该尽量避免。方法论上，高可用保证的原则是“集群化”，或者叫做“冗余”：允许在一定范围内出现故障，而系统不受影响，在需要状态维护的场景中，比如分布式存储中广泛应用。我们使用两台或者两台以上的机器处理影院系统的相同的任务，以保证即使有一台机子出现故障，数据也不会丢失，从而给影院造成损失。但冗余也有弊端，如果没有出现故障，那么冗余的这部分资源就浪费了，不能发挥任何作用。

2. **自动故障转移**。有了冗余还不够，每次出现故障需要人工介入势必会增加系统的不可服务实践，所以，往往通过“自动故障转移”来实现系统的高可用。

3. **去中心化**。指所有节点地位相等，都能够发起数据的更新，其优点是完全消除了中心节点故障带来的全盘出错的风险，但缺点是更高的节点间协作成本。影院公司使用 quorum、vector clock 等算法来尽量保证去中心化环境下的一致性。

4. **数据分片**。又称 partition、sharding。狭义上是指数据存储系统，把大表切分成更小的切片的过程，分割后的数据块会分布在多个服

务器中。它具有以下几个优点：其一，支持系统的水平扩展，性能不够时，只要增加机器数量，而不用升级机器配置，且机器的配置可以很普通；其二，通过数据不同分片之间的热点均衡，可以减少资源征用并提高性能。

2、该公司采用集群支持业务扩展，如何用 Apache httpd 和 tomcat 搭建集群系统，反向代理服务器配置文件需要添加哪些模块，需要添加哪些命令？Tomcat 服务器配制文件要相应做哪些修改？请列出并说明其作用。（15 分）

答案：

一、反向代理服务器配置文件

1. 配置 Apache httpd 24 为反向代理服作为请求负载均衡器

2. 编辑 httpd.conf,修改 Define SERVER 路径为 Apache httpd 文件所在路径；

3. 修改 Listen 端口为 8050；

4. 添加反向代理 ProxyPass/ balancer://mycluster/ 、ProxyPassReserve/ balancer://mycluster/

5. 编辑 httpd.conf，添加以下模块：

mod_lbmethod_bybusyness.so 、 mod_lbmethod_byrequests.so 、mod_lbmethod_bytraffic.so 、 mod_lbmethod_heartbeat.so 、

mod_proxy.so 、 mod_proxy_balancer.so 、 mod_proxy_http.so、 mod_slotmem_shm.so

6. 编辑 httpd.conf，添加指令，配置三个 tomcat 实例形成集群，供负载均衡算法 byrequests 调用.

```
<Proxy balancer://mycluster>
BalancerMember http://localhost:8060
BalancerMember http://localhost:8070
BalancerMember http://localhost:8090
ProxySet lbmethod=byrequests
</Proxy>
```

二、 Tomcat 服务器配制文件

1. 编辑 server.xml ，找到元素以下内容：<!--<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"/> -->,添加以下内容：

```
<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
          channelSendOptions="6">
      <Manager
className="org.apache.catalina.ha.session.BackupManager"
           expireSessionsOnShutdown="false"
```

```
            notifyListenersOnReplication="true"
            mapSendOptions="6"/>
      <!--
  <Manager className="org.apache.catalina.ha.session.DeltaManager"
            expireSessionsOnShutdown="false"
            notifyListenersOnReplication="true"/>    -->
<Channel className="org.apache.catalina.tribes.group.GroupChannel">
<Membership
className="org.apache.catalina.tribes.membership.McastService"
              address="228.0.0.4"
              port="45564"
              frequency="500"
              dropTime="3000"/>
<Receiver
className="org.apache.catalina.tribes.transport.nio.NioReceiver"
             address="auto"
             port="5000"
             selectorTimeout="100"
             maxThreads="6"/>
```

```
<Sender
className="org.apache.catalina.tribes.transport.ReplicationTransmit
ter">
<Transport
className="org.apache.catalina.tribes.transport.nio.PooledParallel
Sender"/>
</Sender>
<Interceptor
className="org.apache.catalina.tribes.group.interceptors.TcpFailure
Detector"/>
<Interceptor
className="org.apache.catalina.tribes.group.interceptors.MessageDis
patch15Interceptor"/>
<Interceptor
className="org.apache.catalina.tribes.group.interceptors.Throughput
Interceptor"/>
</Channel>
<Valve      className="org.apache.catalina.ha.tcp.ReplicationValve"
filter=".*\.gif|.*\.js|.*\.jpeg|.*\.jpg|.*\.png|.*\.htm|.*\.html|.*
```

```
\.css|.*\.txt"/>
<Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
            tempDir="/tmp/war-temp/"
            deployDir="/tmp/war-deploy/"
            watchDir="/tmp/war-listen/"
            watchEnabled="false"/>
<ClusterListener
className="org.apache.catalina.ha.session.ClusterSessionListener"/>
</Cluster>
```

2. 使用 apache tomcat7.0.77 运行影院管理系统 ttms 作为服务器集群。

3、说明 Apache httpd 负载均衡器工作原理，并举例说明请求计数调度算法。（15 分）

答案：一、**Apache httpd 负载均衡器工作原理**：

使用 Apache httpd 提供的 web 服务器是由守护进程 httpd，通过 http 协议进行文本传输，默认使用 80 端口的明文传输方式，也有 443 加密传输的方式。想要给 Apache 添加功能，只需要添加相应的模块，让 Apache

主程序加载即可。Apache 支持插入式并行模块，称为多路处理模块(MPM)，有多种 MPM 模块类型，Worker、Prefork 和 exent 模块。

程序的逻辑：一个无限循环，一个请求。创建一个线程。之后线程处理函数处理每个请求，然后解析 HTTP 请求，作出判断处理，判断文件是否可以执行，不可执行则打开文件，输出给客户端，可执行就创建管道，父子进程通信。

main 函数中，首先调用 startup 函数，在服务端设置 socket，绑定端口号，然后建立监听 socket。进入 while 循环，服务器通过调用 accept 等待客户端连接，accept 会以阻塞的方式运行，指导客户端连接才会返回。连接成功后，服务器调用 pthread_create 启动一个新的线程来处理客户端请求，处理完成后，重现等待新的客户端请求。

处理请求的流程：

1. 从客户端读取请求行

2. 判断请求方法，如果不是 GET 或者 POST，返回服务端无法实现错误。

3. 读取 buf 中请求行的 url。

4. 对于 GET 请求，得到请求文件路径，判断请求文件是否有效，如果有效，根据 cgi 的值，进行静态解析或者 cgi 动态展示。

5. 关闭与客户端连接

二、**请求计数调度算法：**

1. **轮询**。这种方法会将收到的请求循环分配到服务器集群中的每台机器，即有效服务器。如果使用这种方式，所有的标记进入虚拟服务的服务器应该有相近的资源容量以及负载形同的应用程序。如果所有的服务器有相同或者相近的性能，那么选择这种方式会使服务器负载形同。基于这个前提，轮询调度是一个简单而有效的分配请求方式。

2. **加权轮询**。这种算法解决了简单轮询调度算法的缺点：传入的请求按顺序被分配到集群中的服务器，但是会考虑提前为每一台服务器分配的权重.

3. **最少连接数。**以上两种方法都没有考虑到的是系统不能识别在给定时间里保持了多少连接。因此可能发生，服务器 B 服务器收到的连接比服务器 A 少但是它已经超载，因为服务器 B 上的用户打开连接持续的时间更长。这就是说连接数即服务器的负载是累加的。这种潜在的问题可以通过"最少连接数"算法来避免：传入的请求是根据每台服务器当前所打开的连接数来分配的。即活跃连接数最少的服务器会自动接收下一个传入的请求。

4. **最少连接数慢启动时间。**对最少连接数和带权重的最小连接数调度方法来说，当一个服务器刚加入线上环境是，可以为其配置一个时间段，在这段时间内连接数是有限制的而且是缓慢增加的。这为服务器提供了一个'过渡时间'以保证这个服务器不会因为刚启动后因为分配的连接数过多而超载。这个值在 L7 配置界面设置。

5. **加权最少连接**。如果服务器的资源容量各不相同，那么"加权最少连接"方法更合适：由管理员根据服务器情况定制的权重所决定的活跃连接数一般提供了一种对服务器非常平衡的利用，因为他它借鉴了最少连接和权重两者的优势。通常，这是一个非常公平的分配方式，因为它使用了连接数和服务器权重比例，集群中比例最低的服务器自动接收下一个请求。

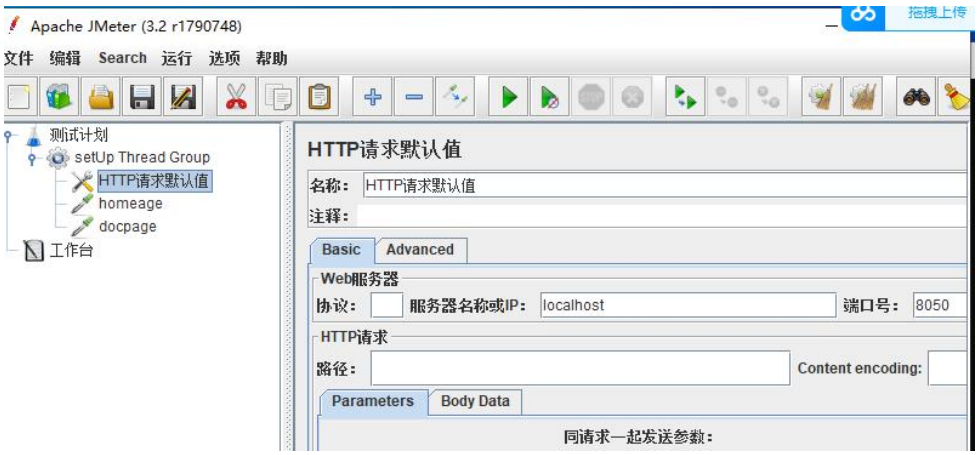
6. **基于代理的自适应负载均衡**。除了上述方法之外，负载主机包含一个自适用逻辑用来定时监测服务器状态和该服务器的权重。当所有服务器的负载低于管理员定义的下限时，负载主机就会自动切换为加权轮循方式来分配请求;如果负载大于管理员定义的下限，那么负载主机又会切换回自适应方式。

7. **固定权重**。最高权重只有在其他服务器的权重值都很低时才使用。然而，如果最高权重的服务器下降，则下一个最高优先级的服务器将为客户端服务。这种方式中每个真实服务器的权重需要基于服务器优先级来配置。

4、公司技术部门准备在票务管理系统集群正式上线之前，完成系统的可用性测试，并计划采用测试工具 Jmeter 完成性能测试，请给出集群的测试方案并截图。（15 分）

答案：

1. 集群测试，一主（master）多从（slave）



2. 找到 Jmeter 的 bin 目录下 jmeter.properties 文件，修改配置，IP 和 Port 是 slave 机的 IP 以及自定义的端口

3. 打开 Jmeter，选择运行，有运程启动、运程全部启动两个选项

4. 选择远程启动，master 结果

二、理论题（2PL 事务并发调度研究进展，40 分）

题目要求：首先，根据本课程教材说明事务并发调度的基本情况和热点数据不加以控制可能出现的现象，例如脏读（10 分）；然后，请查阅 10 篇文献资料，精读你认为最有价值的 1 篇文献，撰写研究进展并提出你自己的观点（20 分）；在撰写过程中，要从参考文献进行引用（5 分）；在报告最后要说明引用的参考文献（5 分）。

题目叙述：在分布式系统中，在协调者的控制之下，多个参与者可以完成诸如分布式事务提交的计算任务。在参与者节点上，多个分布式事务往往会并发地访问热点数据（同一数据对象），为了保证访问这些热点数据结果的正确性，需要并发调度执行结果与可串行性执行结果相同，因此，采用加锁机制（读锁、写锁）的动态并发调度、时间戳机制的静态并发调度和乐观的并发调度。在本课程中，我们提到过 2PL，它就是保证事务并发调度的加锁协议，但是此协议严重降低了参与者节点并发执行的性能，即其它访问热点数据的并发事务只能等待锁释放，之后，才能申请这些热点数据的锁，如何能改变这种情况，研究人员又提出了很多新类型锁，破坏严格释放锁的要求，期望它们既保证事务执行的正确性又提高事务并发执行的程度。

答案：**一、事务并发调度的基本情况**

在同一时间内，多个事务同时存取相同的数据库数据，这样的操作叫做并发，并发操作不去隔离就会产生很多问题，例如丢失修改、脏读和不可重

复读。所以我们必须要对事务的执行进行一些控制，我们需要构造一个事物执行的这个命令的正确的执行队列，即并发调度。

并发调度：指并发的事物的命令按照时间顺序组成的一个执行队列。多个事务从宏观上看是并行执行的，但其微观上的基本操作(读、写)则是交叉执行的。串行调度：如果调度的动作首先是一个事务的所有动作，然后是另一个事务的所有动作，以此类推，而没有动作的混合，那我们说这一调度是串行的。串行调度效率低，它要等到一个事务结束才能进行。

一致性调度：串行调度顺序如果和业务顺序一致的话就是一致性调度

可串行化调度：根据事务的正确性原则，每个串行调度都将保持数据库状态的一致性。通常，不管数据库初态怎样，一个调度对数据库的影响都和某个串行调度相同，我们就说这个调度是可串行化的。

冲突可串行化：某一并行调度 S 经过非冲突指令转换成串行调度，且与该串行调度的执行结果一致，则 S 是冲突可串行化的。

事务不同的隔离级别：事务有很多不同的级别，read uncommited、read commited、repeat read、snapshot isolation、serializable 等，通常指的级别是 serializable

并发控制协议：并发控制协议是数据库用来调度多个事务操作并发执行并保证执行结果符合预期的方式。通常，可以认为并发控制协议就是用来保证事务并发执行时的 schedule 是冲突可串行化的。并发协议可以分为”悲

观并发控制协议”和”乐观并发控制协议”两大类。

基于两阶段锁的并发控制协议：可以通过”交换不冲突的操作”或者”优先图”的方式来验证某个事务调度是否是”冲突可串行化的”。通过加锁的方式可以控制冲突操作的并发问题，并且如果事务在访问或修改某个数据项的过程中都通过相应的锁来保护，直到事务结束的时候才释放锁，那么其他事务与该事务所有的冲突的操作必然是可串行化的，这就是两阶段锁的基本逻辑。

锁管理器：锁管理器负责锁的分配，如果请求的锁与其他的锁是相容的，那么成功分配，否则加锁请求需要等待或者失败。锁的类型可以分为 s-lock(共享锁)、 x-lock(排他锁)。一般来说，读取操作需要持有 s-lock、写入操作需要持有 x-lock。只有两个加 s-lock 的请求是相容的，其他任意类型的两个加锁请求都是不相容的。

	s-lock	x-lock
s-lock	yes	no
x-lock	no	no

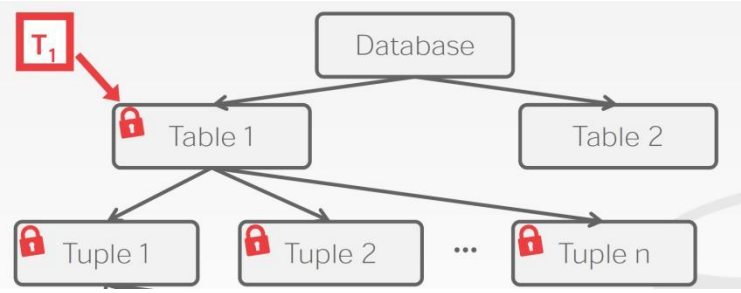
两阶段锁协议内容：简称 2PL 协议，为了保证事务并发调度的正确性。阶段一(Growing)：事务向锁管理器申请所需要的锁，锁管理器分配或拒绝对

应的锁。阶段二(Shrinking)：事务释放在 Growing 阶段申请的锁，并不能再申请新的锁。

死锁问题：对比预防死锁使用的一次封锁法，区别在于两段锁协议只是说加锁的时候在一个阶段完成，没说要用的所有数据都必须加锁，而后者则要求所有使用到的数据必须加锁而且是在刚开始的加锁时期；所以只要是使用一次封锁法的协议都遵循两段锁协议，同时也说明两段锁协议也有死锁问题。

死锁的预防：死锁预防是通过协议来保证系统永不进入死锁状态。两种方法解决，一是预防：一次封锁法、顺序封锁法；二是解决：超时法、等待图法。

多粒度锁：数据库中数据管理的粒度从上到下有 database、table、tuple、column。一般来说数据库操作的最小单位是行，所以最小粒度的锁也就是行锁。整体多级粒度锁的关系是如下图的树状的结构，树中的每个节点都可以单独加锁。当事务对一个节点加锁(共享锁或排他锁)时，该事务隐式的给这个节点的所有后代节点加上了同类型的锁。层次越高的锁，整体加锁和释放锁的代价越低，但是会限制整体的并发度；层次越低的锁，整体加锁和释放锁的代价越高，但是有利于并发。



意向锁：意向锁建立在一个多粒度树上面，多粒度树就是将整个数据库按照对象的大小建立一棵树，这个时候你在一个节点上面加上一个意向锁，那么它以及他的子节点默认被加锁。具有意向锁的多粒度封锁方法提高了系统的并发度，减少了加锁和解锁的开销。

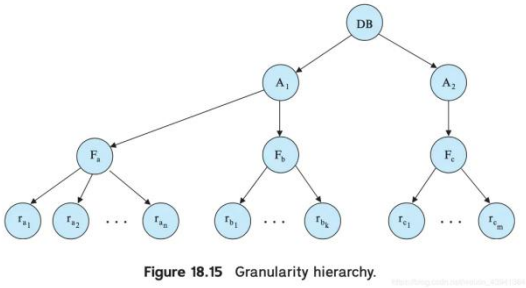


Figure 18.15 Granularity hierarchy.

多粒度树

基于时间戳的调度协议：每个事务在启动数据库系统会赋予这个事务唯一的时间标记，以标记事务的开始，这个时间标记为时间戳。这个时间标记可以是系统时钟或逻辑计数值。时间戳基本原理：事务的时间戳决定串行化顺序，若 TS(Ti) < TS(Tj)，则系统必须保证产生的调度等价于 Ti，Tj 串行调度。

二、热点数据不加以控制出现的问题

1. **热点**：

点表示我们在系统的业务路径上有一个地方存在性能的瓶颈，比如数据库，件系统，网络，甚至于内存等，这个点一般有 io，锁等问题构成。热表示其被访问的频率很高。就是说一个被访问频率很高的 io 或锁自然而然就造成了我们系统业务路径上的性能瓶颈。

这样的热点问题会影响系统稳定性和性能瓶颈，例如支付系统中的热点账户进出款，电商系统中的热点商品参与秒杀，金融系统中的热点理财产品抢购等，都会因为热点问题而影响系统性能。

2. **热点问题**：

我们需要弄清楚我们的热点问题是属于读热点问题还是写热点问题，两种热点问题的处理方案完全不一样，比如我们对一个热门的秒杀商品详情页的访问就属于是读热点问题，对一个秒杀商品的库存抢操作就是一个写热点问题。虽然分离了读热点问题和写热点问题，但是往往在读热点问题时也需要处理写热点问题的解决方案，比如我对一个热门的秒杀商品详情的读热点问题使用了缓存解决方案，但因为商家对商品做了更新价格的东西时立马需要对写热点而造成的缓存脏数据做清理的操作，因此就变成了一个读写混合的热点问题。

3. 解决读热点

读热点最常规的思路是本地 cache 方案和二级缓存方案，这是从

业务层面解决，而不是从分布式缓存框架的层面上解决。

(1) **mysql 优化**：一般我们做系统之出使用数据库，直接对用户的请求做 sql 的 select 操作，那对于此类热点问题我们首先想到的是需要优化数据库的读操作，我们对应的查询是否走了索引，走的是否是唯一索引甚至于主健效果最佳，优化了 sql 性能后，可以借助于 mysql innodb 的 buffer 做一些文章，在数据库层面就提供足够的缓冲区，加速对应的性能，实验证明，只要走的是主健或唯一索引，innodb 缓冲区足够大的情况下，进行 mysql 的主从分离和集群化部署，mysql 抗上亿的数据也是没有任何问题的。

(2) **redis 缓存**：真正出问题的不是点而是热，由于访问频次太高，mysql 的 cpu 扛不住了，这个时候我们考虑到的是将对应的读热点放到例如 redis 的缓存中用于卸载压力，由于 redis4 版本以后就可以支持 cluster 的集群模式，其借助分片集群的效果理论上可以扩展到 1000 个左右的节点，如此一来我们可以依靠缓存去解决读热点问题，一旦商家变更了读热点的数据，我们可以在业务应用中使用提交后异步清除缓存的方式将 redis 的数据清除，这样在下一次的请求中可以依靠数据库的回源更新 redis 数据。

(3) **redis 分片**：那既然我们讨论的是读热点问题，就和 redis 的水平扩展能力无关，因为是个热点数据，则必定会被分片路由到一个

redis 节点上，当热度大到连 redis 节点都无法承受的时候，我们可以考虑将原本的一个热点做三分拷贝，比如我们的热点 key 叫"item_1"，我们可以考虑随机的生成三个 key 分别叫"item_1_key_1"，“item_1_key_2”，“item_1_key_3”，对应的 value 都是这个商品 value 本身，这样当用户请求过来后可以随机的生成 1-3 的数字以决定这次请求我们访问哪个 key，这样人为的将一个热点的 tps 降到了原来的三分之一，以空间换时间。

(4) **本地缓存**：另外我们还可以考虑在应用服务器上做本地的 cache 内存，由于应用服务器本身容量有限，内存中不能放太多数据，也不能存很长时间，我们推荐使用 google 研发的 guava cache 包，提供给我们很好的 lru cache 队列的能力，一般本地的缓存不要设置太长时间，一是出于内存容量考虑，二是出于清理本地缓存不会同步清理 redis，需要我们的每台应用服务感知到数据的变更，一般可以用广播型的 mq 消息解决，推荐 rocketmq 的广播型消息，使得订阅对应商品信息变更的所有应用服务器都有机会清理本地缓存。

4. 解决写热点

(1) **mysql 优化**：针对点我们一般写操作会选用数据库之类的文件存储设备，mysql 对数据存储有比较好的优化，其基于写事务日志，也就是 redo，undo log，然后等系统空闲的时候将数据刷入磁盘的，

由于事务日志的存在，即使系统挂了再启动的时候也可以根据 redo log 恢复数据，那为啥写 log 比写数据快那么多的，因为写日志是一个顺序追加写的方式，磁盘的磁头不需要随机的移动寻找写入点，只要顺序的写下去即可，配合 ssd 固态硬盘，整个写入性能可以做到很高。

(2) **redis 缓存**：但是磁盘操作终究是磁盘操作，我们试着可以将写入的目标点移到缓存中，比如将秒杀的库存移到 redis 中，这么一来，点的瓶颈的天花板瞬间就提升到了很多倍，但是一旦将数据落到没有办法保证磁盘落地能力的缓存中就需要去靠一些机制去保证可靠性，不至于在缓存丢失的情况下造成超卖等灾难，我们可以依靠 rocketmq 异步事务型的消息保持 redis 和数据库之间的数据同步，解决缓存异常情况下我们可以依靠数据库恢复对应的数据。

(3) **缓冲入账**：那异步化是解决问题的最终方案吗？显然不是，异步化只是将对应的写热点问题延迟到后面去解决，不至于卡住前端的用户体验，但是一旦这个点多了起来，后端服务器和磁盘的压力还在，那我们还有什么方法去解决呢？我们都知道写入操作之所以在热点问题的情况下那么难解决，是因为写同一份数据的操作不能并发，必须得要通过竞争锁的机制去竞争以获得线程的写入权限，我们突然可以想到，锁这个东西本身就是一个耗性能的来源，设想两个人要抢

同一个食堂阿姨拿出来的饭，你争我抢，我抢到了吃晚了再给后面的其他人在争，在竞争的过程中所有人，所有线程的资源都被白白消耗掉了，最终还是只有一个人在那个时刻可以吃到饭。那针对这种情况我们是否可以有更好的解决方案呢？还是考虑抢饭吃这个场景，在现实生活中最高效的方式是什么，就是排队，大家都不要竞争，按照先到先得的方式将所有对热点的写入访问操作队列化，使用单线程的方式去队列中取得下一个写入操作，然后写完后再取下一个，这样可以避免掉写锁竞争的无谓 cpu 和内存消耗，也可以使用单线程的方式解决，没有 cpu 调度切换的开销。就是我们常说的在无锁的情况下，单线程排队比多线程更高效，这种解决写热点的方式叫做"缓冲入账"。

三、文献阅读——The Google file system

Google 文件系统是一个一个可扩展的分布式文件系统，适用于大型分布式数据密集型应用程序，它不仅需要在廉价的商品服务器上运行，而且需要保持高聚合性能，以支持全球上亿人的搜索量。

相比其他分布式文件系统，Google 文件系统的设计是由当前和预期的应用程序工作负载和技术环境的观察所驱动的，这是有别于其他文件系统的地方。

GFS 在 Google 被作为存储平台广泛部署，用于生成、处理服务所需要的数据

或用于需要大型数据集的研发工作。这篇文章主要介绍了为分布式应用程序而设计的文件系统接口扩展，并给出了小批量的 benchmark 与在现实场景中的使用表现。

为了满足快速增长的数据处理需求，Google 设计并实现了 GFS。GFS 与过去的分布式系统有着很多相同的目标，如性能（performance）、可伸缩性（scalability）、可靠性（reliability）和可用性（availability）。但是我们的设计来自于我们对 Google 的应用负载与技术环境的观察。

首先，在应用程序中，故障时有发生。GFS 由成百上千台由廉价设备组成的存储节点组成，并被与其数量相当的客户端访问。设备的数量和质量决定了几乎在任何时间都会有部分设备无法正常工作，甚至部分设备无法从当前故障中分恢复。Google 遇到过的问题包括：应用程序 bug、操作系统 bug、人为错误和硬盘、内存、插头、网络、电源等设备故障。因此，系统必须具有持续监控、错误检测、容错与自动恢复的能力。

其次，文件比传统标准更大。数 GB 大小的文件是十分常见的。每个文件一般包含很多引用程序使用的对象，如 Web 文档等。因为 Google 的数据集由数十亿个总计数 TB 的对象组成，且这个数字还在快速增长，所以管理数十亿个几 KG 大小的文件是非常不明智的，即使操作系统支持这种操作。因此需要重新考虑像 I/O 操作和 chunk 大小等设计和参数。

第三，大部分文件会以“追加”（append）的方式变更（mutate），而非“覆写”（overwrite）。在实际场景中，几乎不存在对文件的随机写入。文件一旦被写入，即为只读的，且通常仅被顺序读取。很多数据都有这样的特征。如数据分析程序扫描的大型数据集、流式程序持续生成的数据、归档数据、由一台机器生产并同时或稍后在另一台机器上处理的数据等。鉴于这种对大文件的访问模式，追加成了为了性能优化和原子性保证的重点关注目标，而客户端中对 chunk 数据的缓存则不再重要。

第四，同时设计应用程序和文件系统 API 便于提高整个系统的灵活性。例如，放宽了 GFS 的一致性协议，从而大幅简化了系统，减少了应用程序的负担。还引入了一种在不需要额外同步操作的条件下允许多个客户端并发将数据追加到同一个文件的原子性操作。

Google 在设计 GFS 时，提出了一些假设。

系统有许多可能经常发生故障的廉价的商用设备组成。它必须具有持续监控自身并检测故障、容错、及时从设备故障中恢复的能力。

系统存储一定数量的大文件。我们的期望是能够存储几百万个大小为 100MB 左右或更大的文件。系统中经常有几 GB 的文件，且这些文件需要被高效管理。系统同样必须支持小文件，但是不需要对其进行优化。

系统负载主要来自两种读操作：大规模的流式读取和小规模的随机读取。在大规模的流式读取中，每次读取通常会读几百 KB、1MB 或更多。来自

同一个客户端的连续的读操作通常会连续读文件的一个区域。小规模的随机读取通常会在文件的某个任意偏移位置读几 KB。性能敏感的应用程序通常会将排序并批量进行小规模的随机读取，这样可以顺序遍历文件而不是来回遍历。

系统负载还来自很多对文件的大规模追加写入。一般来说，写入的规模与读取的规模相似。文件一旦被写入就几乎不会被再次修改。系统同样支持小规模随机写入，但并不需要高效执行。

系统必须良好地定义并实现多个客户端并发向同一个文件追加数据的语义。我们的文件通常在生产者-消费者队列中或多路归并中使用。来自不同机器的数百个生产者会并发地向同一个文件追加写入数据。因此，最小化原子性需要的同步开销是非常重要的。文件在被生产后可能同时或稍后被消费者读取。

持续的高吞吐比低延迟更重要。我们的大多数应用程序更重视告诉处理大量数据，而很少有应用程序对单个读写操作有严格的响应时间的需求。

尽管 GFS 没有实现像 POSIX 那样的标准 API，但还是提供了大家较为熟悉的文件接口。文件被路径名唯一标识，并在目录中被分层组织。GFS 支持如创建（create）、删除（delete）、打开（open）、关闭（close）、读（read）、写（write）文件等常用操作。

此外，GFS 还支持快照（snapshot）和追加记录（record append）操作。快照操作会以最小代价创建一个文件或一个目录树的拷贝。追加记录操作允许多个客户端在保证每个独立的客户端追加操作原子性的同时能够并发地向同一个文件追加数据。这对实现如多路归并、生产者-消费者队列等多个客户端不需要额外的锁即可同时向同一文件追加数据非常有益。

如下图所示，一个 GFS 集群包括单个 master（主服务器）和多个 chunkserver（块服务器），并被多个 client（客户端）访问。每个节点通常为一个运行着用户级服务进程的 Linux 主机。如果资源允许且可以接受不稳定的应用程序代码所带来的低可靠性，那么可以轻松地在一台机器上同时运行 chunkserver 和 client。

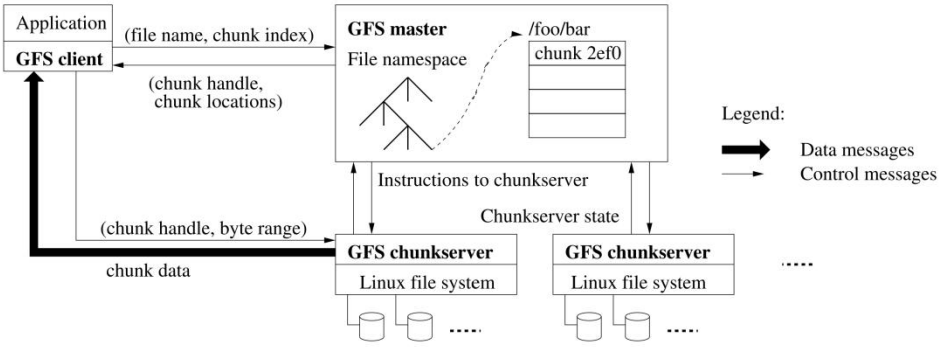


Figure 1: GFS Architecture

文件被划分为若干固定大小的块。每个块被一个不可变的全局唯一的 64 位块标识符唯一标识，块标识符在块被创建时由主节点分配。chunkserver 将块作为 Linux 文件存储到本地磁盘中，通过 chunk handle 和

byte range（字节范围）来确定需要被读写的块和块中的数据。为了可靠性考虑，每个块会在多个 chunkserver 中有副本。我们默认存储三份副本，用户也可以为不同的命名空间的域指定不同的副本级别。

master 维护系统所有的元数据。元数据包括命名空间（namespace）、访问控制（access control）信息、文件到 chunk 的映射和 chunk 当前的位置。master 还控制系统级活动如 chunk 租约（chunk lease）管理、孤儿 chunk 垃圾回收（garbage collection of orphaned chunks）和 chunkserver 间的 chunk 迁移（migration）。master 周期性地通过心跳（HeartBeat）消息与每个 chunkserver 通信，向其下达指令并采集其状态信息。

被链接到应用程序中的 GFS client 的代码实现了文件系统 API 并与 master 和 chunkserver 通信，代表应用程序来读写数据。进行元数据操作时，client 与 master 交互。而所有的数据（译注：这里指存储的数据，不包括元数据）交互直接由 client 与 chunkserver 间进行。因为 GFS 不提供 POXIS API，因此不会陷入到 Linux vnode 层。

无论 client 还是 chunkserver 都不需要缓存文件数据。在 client 中，因为大部分应用程序需要流式地处理大文件或者数据集过大以至于无法缓存，所以缓存几乎无用武之地。不使用缓存就消除了缓存一致性问题，简化了 client 和整个系统。（当然，client 需要缓存元数据。）chunkserver 中的

chunk 被作为本地文件存储，Linux 系统已经在内存中对经常访问的数据在缓冲区缓存，因此也不需要额外地缓存文件数据。

读工作负载主要由两种读方式构成：大规模的串行读以及小规模的随机读。大规模顺序读：顺序（磁盘地址连续地）读取数百及以上个 KB 大小的数据（或者单位改成 MB）；小规模随机读：以任意偏移量读取几个 KB 大小的数据；小规模随机读会有优化，比如进行排序后的批处理化，以稳定地遍历文件（排序可能是按照索引的指针大小），而不是来回地随机读取。

写工作负载主要是大规模的、连续（即串行的）的写操作，这些操作将数据追加到文件末尾。写操作的规模通常和大规模串行读的规模类似；这要求：文件一旦写好，就几乎不会进行覆写，虽然 GFS 支持在文件的任意位置进行修改，但是并不会进行优化，存在并发安全问题，因此应当尽量避免使用。系统需要支持并发写，即支持数百台机器并发地追加数据到一个文件。操作的原子性和同步开销是主要指标；

高持续带宽（High sustained bandwidth）比低延迟更重要；

GFS 作为一个分布式文件系统，对外提供了一个传统的单机文件系统接口。但是出于效率和使用性的角度，并没有实现标准的文件系统 POSIX API。

文件通过目录进行分层管理，通过路径名来定位，支持文件的 create，delete，open，close，read 以及 write 操作。

此外 GFS 还支持如下两个特性：

（1）Snapshot **快照**：快照指的是以低成本方式创建文件和目录的副本；

（2）Record Append **记录追加**：记录追加指的是 GFS 允许多个客户机并发安全地向同一文件追加数据，同时保证每个客户追加操作的原子性；

一个 GFS cluster 分为两个组件：单个 matser 节点和多和 chunkserver 节点。一个 GFS 集群可以用以下图片表示，可见 GFS 集群是一个典型的 Master + Worker 结构。

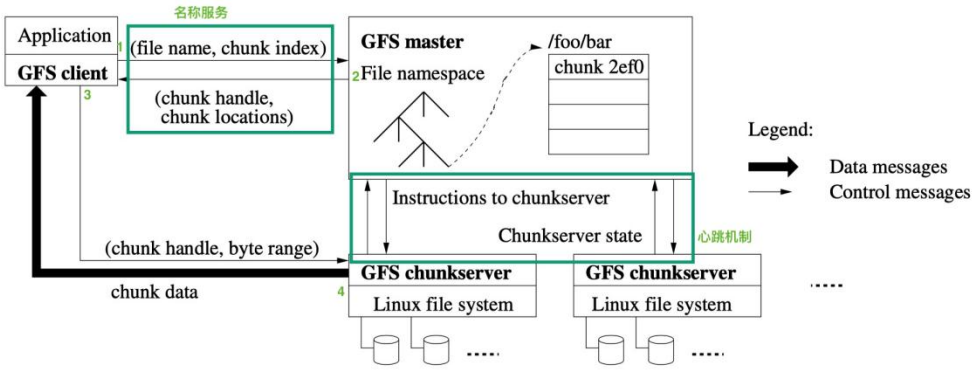


Figure 1: GFS Architecture

GFS 基础概念介绍到此为止，在 GFS 中，大文件为什么要分块存储？论文中说，大文件分块存储和 MySQL 的水平扩展、垂直扩展的理念是一样的，

或者说类似于 Redis 的主从节点的设计。不过，如果要讨论最基本的原理，那便是：将串行通为并行。并行操作则是分布式系统在各个工厂领域广泛应用的原因。

大的文件块有自己的优点：

1. 首先，它减少了 Client 与 Master 服务器交互的次数，因为对同一块进行多次读写仅仅需要向 Master 服务器发出一次初始请求，就能获取全部的块位置信息。这可以有效地减少 Master 的工作负载；

2. 其次，减少了 GFS Client 与 GFS chunkserver 进行交互的数据开销，这是因为数据的读取具有连续读取的倾向，即读到 offset 的字节数据后，下一次读取有较大的概率读紧挨着 offset 数据的后续数据，chunk 的大尺寸相当于提供了一层缓存，减少了网络 I/O 的开销；

3. 第三，它减少了存储在主服务器上的元数据的大小。这允许我们将元数据保存在内存中。

但同时大的文件块也有缺点，即小数据量（比如仅仅占据一个 chunk 的文件，文件至少占据一个 chunk）的文件很多时，当很多 GFS Client 同时将 record 存储到该文件时就会造成局部的 hot spots 热点。

参考文献

[1]Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung,《The Google file system》, SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principlesOctober 2003 Pages 29—43

[2]Jeffrey Dean, Sanjay Ghemawat, jeff@google.com, sanjay@google.com,《MapReduce: simplified data processing on large clusters》, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), pp. 137-150

[3]Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E.Gruber,《Bigtable: A Distributed Storage System for Structured Data》, ACM Transactions on Computer Systems, Volume 26, Issue 2, June 2008 , Article No.: 4, pp 1—26

[4]Mike Burrows,《The Chubby lock service for loosely-coupled distributed systems》, OSDI '06 Paper, September 2006,Pp. 335—350

[5]Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel,《Finding a needle in Haystack: Facebook's photo storage》, OSDI 2010, October 4-6, 2010

[6]Brad Calder，Ju Wang，Aaron Ogus，Niranjan Nilakantan，Arild Skjolsvold，Sam McKelvie，Yikang Xu，Shashwat Srivastav，Jiesheng Wu，Huseyin Simitci，Jaidev Haridas，Hemal Khatri，Andrew Edwards，Vaman Bedekar，Shane Mainali，Rafay Abbasi，Arpit Agarwal，Mian Fahim ul Haq，Muhammad Ikram ul Haq，Deepali Bhardwaj，Sowmya Dayanand，Anitha Adusumilli，Marvin McNett，Sriram Sankaran，Kavitha Manivannan· Leonidas Rigas，《Windows Azure Storage：a highly available cloud storage service with strong consistency》，SOSP '11：Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles，October 2011，Pages 143—157

[7]Yucheng Low，Joseph E. Gonzalez，Aapo Kyrola，Danny Bickson，Carlos E. Guestrin，Joseph Hellerstein，《GraphLab：A New Framework For Parallel Machine Learning》，UAI-P-2010-PG-340-349

[8]Matei Zaharia，Mosharaf Chowdhury，Tathagata Das，Ankur Dave，Justin Ma，Murphy McCauley，Michael J. Franklin，Scott Shenker，and Ion Stoica，University of California，Berkeley，《Resilient Distributed Datasets：A Fault-Tolerant Abstraction for In-Memory Cluster Computing》

[9]Mu Li，《Scaling Distributed Machine Learning with System and Algorithm Co-design》，CMU-CS-17-102，February 2017

[10]Sergey Melink, Andrey Gubarey, Jing Jing Long, Geoffrey, Shiva Shivakumar, Matt Tolton, Theo Vassilakis,《Dremel: interactive analysis of web-scale datasets》 ,Proceedings of the VLDB Endowment, Volume 3, Issue 1-2, September 2010 ,pp 330—339