软件系统分析与设计作业

**学术成果分享平台**

**技术调研说明书**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **项目组成员信息** | | | | |
| **小组编号** | **13** | | | |
| **小组名称** | **MSI** | | | |
| **小组联系人** | **周星达** | | | |
| **学号** | **姓名** | **联系电话** | **本次实践中主要承担的工作内容** | **本次作业占比**  **(95%-105%之间)** |
| 21373339 | 周星达 | 15269992619 | 架构设计PPT制作 | 100% |
| 20101021 | 梁坤 | 18712085680 | 非功能需求架构设计 | 100% |
| 21373330 | 吴浩宇 | 13552318175 | 部署策略 | 100% |
| 21373019 | 曾尔文 | 18979618851 | 文档分析完善 | 100% |
| 21373278 | 赵乾宇 | 18355750747 | 文档分析完善 | 100% |
| 21373326 | 秦茂凯 | 15611759462 | 文档分析完善 | 100% |
| 21373328 | 王陆昊 | 13273673856 | 文档审核与汇总 | 100% |
| 21371220 | 杨硕 | 18098715927 | 技术选型 | 100% |
| 21371134 | 田琦 | 13225685550 | 前后端架构设计 | 100% |
| 71066002 | 温舒盈 | 19658727170 | 文档分析完善 | 100% |

2023年

版本变更历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本 | 提交日期 | 主要编制人 | 审核人 | 版本说明 |
| 1.0 | 2023.11.05 | 王陆昊、赵乾宇、曾尔文、田琦、杨硕、温舒盈、周星达、梁坤 | 周星达 | 完成大体部分，合并完成 |
| 1.1 | 2023.11.08 | 周星达 | 周星达 | 进行内容完善 |

目 录

[1. 引言 2](#_Toc150201187)

[1.1 项目概述 2](#_Toc150201188)

[1.2 问题引入 2](#_Toc150201189)

[1.3 文档概述 2](#_Toc150201190)

[2. 总体架构 2](#_Toc150201191)

[2.1 前端架构 2](#_Toc150201192)

[2.1.1 MVVM设计模式 3](#_Toc150201193)

[2.1.2 组件化开发 3](#_Toc150201194)

[2.1.3 Vue CLI工具 3](#_Toc150201195)

[2.1.4 指令系统 3](#_Toc150201196)

[2.1.5 html、css、javascript设计 3](#_Toc150201197)

[2.2 后端架构 4](#_Toc150201198)

[3. 部署策略 5](#_Toc150201199)

[3.1 部署架构 5](#_Toc150201200)

[3.2 部署方案 6](#_Toc150201201)

[3.3 部署设备性能实例 6](#_Toc150201202)

[3.3.1 高性能服务器实例 6](#_Toc150201203)

[3.3.2 普通服务器实例 6](#_Toc150201204)

[4. 技术选型 7](#_Toc150201205)

[4.1 技术框架表格 7](#_Toc150201206)

[4.1.1 后端技术 7](#_Toc150201207)

[4.1.2 前端技术 7](#_Toc150201208)

[4.2 重要技术说明 7](#_Toc150201209)

[4.2.1 Django 7](#_Toc150201210)

[4.2.2 MySQL 8](#_Toc150201211)

[4.2.3 Elasticsearch 8](#_Toc150201212)

[4.2.4 Celery 8](#_Toc150201213)

[4.2.5 Nginx 8](#_Toc150201214)

[4.2.6 Redis 8](#_Toc150201215)

[4.2.7 Vue.js 9](#_Toc150201216)

[5. 质量属性 9](#_Toc150201217)

[5.1 设计质量 9](#_Toc150201218)

[5.1.1 概念完整性 9](#_Toc150201219)

[5.1.2 可移植性 9](#_Toc150201220)

[5.1.3 可维护性 10](#_Toc150201221)

[5.1.4 复用性 10](#_Toc150201222)

[5.2 运行质量 10](#_Toc150201223)

[5.2.1 优化性能 10](#_Toc150201224)

[5.2.2 可延展性 10](#_Toc150201225)

[5.2.3 安全性 10](#_Toc150201226)

[5.3 系统质量 10](#_Toc150201227)

[5.3.1 可支持性 10](#_Toc150201228)

[5.3.2 可测试性 10](#_Toc150201229)

[5.4 用户质量 11](#_Toc150201230)

[5.4.1 可用性 11](#_Toc150201231)

[6. 横切关注点 11](#_Toc150201232)

[6.1 安全性 11](#_Toc150201233)

[6.1.1 安全威胁 11](#_Toc150201234)

[6.1.2 安全模式 11](#_Toc150201235)

[6.2 异常处理 12](#_Toc150201236)

[6.2.1 异常检测 12](#_Toc150201237)

[6.2.2 异常恢复 12](#_Toc150201238)

[6.3 缓存 12](#_Toc150201239)

[6.4 日志 12](#_Toc150201240)

# 引言

## 问题引入

在学术成果分享平台展开架构设计之前，针对该系统具有亿级数据量的特点，我们对大数据技术展开了调研工作，选取了典型的分布式存储技术、大数据处理框架和缓存技术，生成了此调研报告。

# 整体介绍

## 基本简介

面对亿级数据量的平台，如何处理大数据是重中之重。想要做好学术成果分享平台的数据管理，显然不能闭门造车，要去了解处理大数据的技术。而随着信息技术的飞速发展，尤其是互联网数据量的爆炸性增长，一系列创新技术应运而生，值得参考借鉴。其中包括Hadoop Distributed File System（HDFS）、MapReduce计算框架以及Redis内存数据结构存储系统。

HDFS是Hadoop框架中的一个关键组件，它被设计用于高效存储和处理海量数据集。这个文件系统借鉴了GFS的设计，提供了高度容错的特性和高吞吐量的访问能力，适合那些需要大规模数据存储的企业和组织。通过分布式的架构，HDFS可以将数据分散存储在多个节点上，确保了数据的可靠性和系统的可伸缩性。

为了处理这些分散的数据，MapReduce作为一个大数据处理框架被引入。它的核心理念是“分而治之”，将一个大计算任务分解成多个小任务，分发到多个计算节点上并行处理，最后再汇总结果。这种方法大大提高了处理大数据集时的效率和速度。

Redis是一个开源的内存中数据结构存储系统，用于构建高性能的键值数据库。Redis支持多种类型的数据结构，如字符串、列表、集合、哈希表、有序集合等。由于其数据都存储在内存中，Redis可以提供极快的读写速度，非常适合需要快速访问的应用场景，如缓存、会话存储、实时分析等。

将HDFS、MapReduce和Redis结合起来，就可以构建一个功能强大、高效且可扩展的大数据处理和存储平台。HDFS提供了坚实的基础存储能力，MapReduce确保了数据处理的效率和可靠性，而Redis则贡献了快速访问的能力。这三者的结合，能够更好的处理大数据问题，也同样值得学习。

## 应用成效

HDFS、MapReduce和Redis是当前大数据领域内核心的技术组件，在不在支持大数据系统上发挥着不同的效果。

HDFS具有高容错性、高伸缩性等优点，允许用户将Hadoop部署在廉价的硬件上，构建分布式文件存储系统，用于解决海量文件存储问题。

MapReduce分布式计算框架则允许用户在不了解分布式系统底层细节的情况下开发并行、分布式的应用程序，充分利用大规模的计算资源，解决传统高性能单机无法解决的大数据处理问题。

而HDFS和MapReduce都是Hadoop的核心组件，在直接处理大数据上有着重要作用。

而Redis则经常用作缓存，提供了极高的读写速度。

HDFS的可靠性体现在可以有效备份数据，数据不易丢失。

MapReduce的分布式计算使得多个机器可同时计算一个任务的一部分，然后把每个计算的结果进行汇总，大大增加运算效率。

Redis将所有数据存储在内存中，实现了高速数据读写。

这三项技术已经被广泛应用于需要分布式与大数据技术支持的场景。

（1）Twitter

Twitter使用Hadoop的文件系统（HDFS）来存储和处理巨量的社交媒体数据。在Twitter中，HDFS用于日志存储和用户行为数据的聚合，这些数据是用于生成用户时间线和广告定向的关键数据。

（2）Linkedln

LinkedIn使用MapReduce来处理其庞大的社交网络数据，例如，构建用户之间的关系图谱、推荐系统（如“人脉推荐”功能），以及对大量的用户生成数据进行分析以提供更个性化的职业机会。

（3）GitHub

GitHub使用Redis作为缓存和队列系统，以优化站点性能。例如，它用于缓存用户会话和存储后台作业队列，这样可以快速响应用户的请求，并保持平台处理大量请求时的高性能和稳定性。

# 技术分析

接下来，我们将基于搜索引擎数据处理的相关问题，对HDFS,MapRedure,Redis的技术架构展开讲解与分析。

## HDFS

### 相关问题

HDFS作为大数据处理的核心文件系统，拥有远超过单个磁盘容量的数据，例如分散存储在整个Hadoop集群中的大型数据集，日志文件，以及各种数据处理中间文件。HDFS的设计目标是提供一个可靠、分布式的文件系统，能够存储和管理大规模数据，同时支持高吞吐量和容错能力。HDFS分割数据为较小的数据块，通常为128 MB或256 MB，然后将这些块复制到不同的数据节点上，以确保数据冗余和容错性。这使得HDFS能够快速并行访问海量数据，支持大规模数据处理工具，如MapReduce。与GFS类似，HDFS也具有全局有效性，这意味着各种不同的大数据应用程序都可以从中读取数据，而不需要为每个应用程序建立单独的存储系统。在HDFS中，大量的数据可以方便地被多个应用程序和用户共享，因为它们都访问同一个分布式文件系统。这使得任何Hadoop集群内部的用户都可以根据文件名或路径访问HDFS中的可被共享的数据，从而实现了数据的全局共享和协同处理。

## 技术架构

### 架构概览



图3-1 HDFS架构概览

NameNode：NameNode是HDFS的主要管理节点，负责存储文件系统的元数据信息，如文件和目录结构，文件块的位置，权限等。它维护了一个命名空间树，记录了文件和目录的层次结构。NameNode是单点故障，因此需要进行备份和故障恢复操作以保持元数据的高可用性。

DataNode：DataNode是HDFS中的数据存储节点。它们负责实际存储数据块，并向NameNode报告它们所存储的块的状态。数据块通常被复制到多个DataNode上，以提供数据冗余和容错性。DataNode可以动态加入或离开集群，使HDFS具备可伸缩性。

数据块：HDFS将大文件分成较小的数据块，通常默认大小为128MB或256MB。这些数据块在不同的DataNode上分布，以提供并行存储和处理能力。数据块的复制策略由管理员配置，通常会复制到不同的DataNode上，以防止数据丢失。

Secondary NameNode：Secondary NameNode并不是NameNode的备份，而是用于协助NameNode进行元数据的检查点和维护工作。它定期合并和压缩NameNode的事务日志，以减少NameNode恢复所需的时间。

客户端：HDFS客户端用于与HDFS进行交互，包括读取、写入和删除文件，以及监视文件的状态。客户端通过HDFS的Java API或命令行工具与HDFS集群通信。

副本策略：HDFS使用数据块的复制策略来确保数据冗余和容错性。通常，每个数据块都会被复制到多个DataNode上，默认情况下是三个副本。这些副本通常分布在不同的机架上，以提高容错性和数据访问的性能。

### 分布式数据存储和复制策略

Hadoop Distributed File System (HDFS)的分布式数据存储和复制策略是一个渐进的过程，涵盖了数据的切分、复制、副本策略和数据局部性等关键方面。

首先，当一个大文件需要存储到HDFS时，系统会将该文件切分成较小的数据块。这个切分过程允许大文件被分成多个更小的部分，通常是128MB或256MB大小的数据块。这一步是数据的分割过程，将大数据划分为更容易管理和处理的单元。

每个数据块都会被复制到多个不同的DataNode上。这是为了确保数据的冗余存储和容错性。每个数据块通常会有多个副本，默认情况下是三个副本。这意味着每个数据块的多个拷贝会分布在不同的DataNode上，分散在整个HDFS集群中。这一过程是数据的冗余复制，以防止数据丢失。

复制策略是可配置的，管理员可以根据需要进行调整。这使得复制过程可以根据不同的应用场景和需求进行调优。例如，对于关键数据，可以选择更高的复制因子以增加冗余性，而对于临时数据，可以选择较低的复制因子以减少存储成本。这一步是数据的自定义冗余策略。

HDFS通过机架感知来确保数据块的一个副本位于与客户端或计算任务最接近的机架上。这一过程是渐进的，确保数据块的位置逐渐调整，以减少数据的网络传输。这意味着数据块的一个副本通常位于离用户或计算任务更近的节点上，而其他副本分布在不同的机架上。这提高了数据的访问性能，因为数据通常可以在本地或附近的节点上访问。

整个过程是渐进的，以平衡数据的冗余、容错性和性能。它使HDFS成为一个适合大规模数据存储和分布式计算的可靠文件系统，同时确保了数据的可用性和安全性。

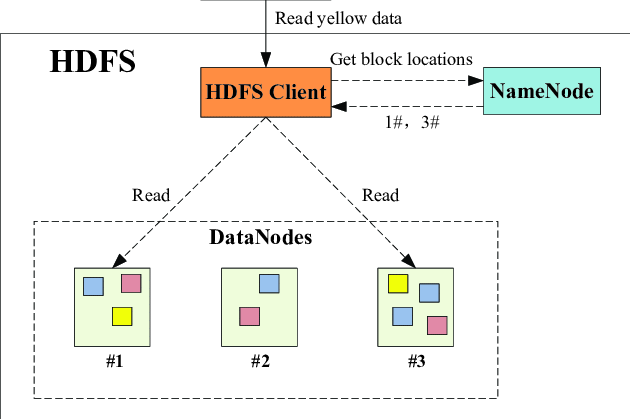


图3-2 HDFS数据副本存放策略

### 单点故障和元数据管理

Hadoop Distributed File System (HDFS)的单点故障问题涉及到文件系统的核心组件，即NameNode，负责管理元数据，如文件和目录结构、文件块的位置和权限。由于NameNode是单一的，一旦NameNode发生故障，整个文件系统的元数据将不可用，导致无法访问文件和数据块，严重威胁数据可靠性和可用性。为了解决这个问题，HDFS采用了复杂的元数据管理流程如图3-3所示：

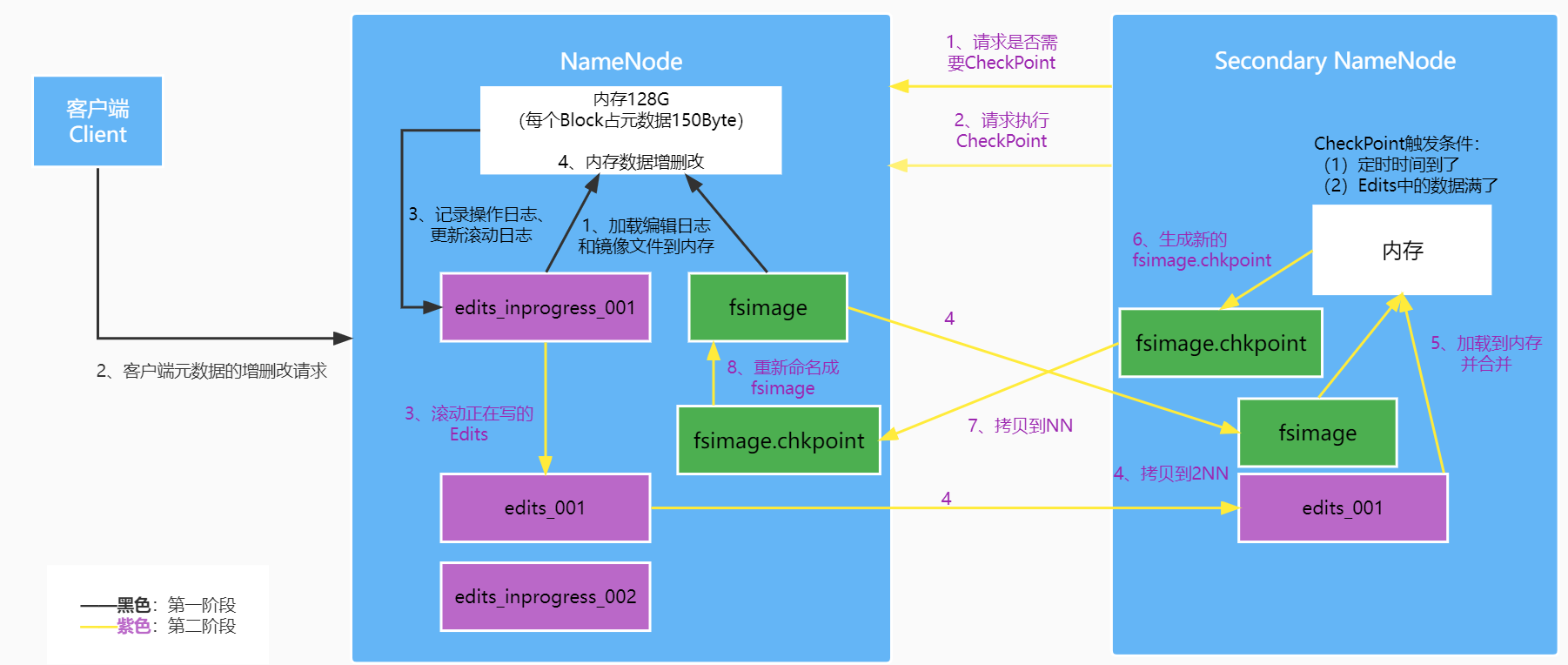


图3-3 HDFS元数据管理

NameNode的责任： NameNode是HDFS的元数据管理节点，负责维护整个文件系统的元数据信息。它不存储实际数据块，但记录了文件和目录的结构、数据块的位置、权限等。

Secondary NameNode： Secondary NameNode并不是NameNode的备份，而是一个辅助节点。它的主要任务是定期合并和压缩NameNode的事务日志（edits log）。这些日志包含了对元数据的更新操作。 Secondary NameNode通过将这些日志合并成一个新的检查点文件，以降低检查点操作的时间和复杂度。

检查点（Checkpoint）： 检查点是元数据的快照，包括文件系统的当前状态。它存储在Secondary NameNode的本地文件系统上。检查点的生成是一个周期性的过程，它将当前元数据状态持久化到本地磁盘，以备份元数据。

Standby NameNode： 为进一步提高可用性，HDFS引入了Standby NameNode。Standby NameNode是一个备份NameNode，维护一个与主NameNode相同的元数据副本。它可以随时接管主NameNode的角色，以维护文件系统的连续可用性。这意味着如果主NameNode发生故障，Standby NameNode能够迅速接管，并继续提供元数据服务。

元数据管理流程的核心是定期生成检查点，这使得在NameNode发生故障时，可以从最近的检查点中快速恢复元数据状态，以维持文件系统的可用性。同时，Standby NameNode提供了元数据的备份和故障切换的能力，确保了HDFS的高可用性和数据可靠性。

这一系列的元数据管理机制和流程确保了HDFS在大规模数据存储和处理环境中的稳定运行，减少了单点故障对数据的风险。

## MapReduce

### 相关问题

随着数据的不断增长，当今企业面临的数据处理需求已远远超过了单台服务器能够处理的能力。不管是大家都在用的微信、QQ或京东、淘宝还是学校内的课程网站，每天都会产生大量的用户行为日志、交易记录、实时通信数据等。这些信息和数据需要非常高的处理和计算效率才能使这些平台正常运行。处理这些大数据最大的难题不是逻辑，而是对计算能力和数据分发合理性的需求。单台服务器满足不了，就需要多台，就又涉及了在分布式机器上处理数据一致性，任务调度等问题。可谓问题重重。

为了应对这些挑战，产生了像MapReduce这样的编程模型，它让开发者可以专注于编写数据处理的逻辑，而不必深入到底层架构的实现细节中去。MapReduce将数据分布、任务调度、故障恢复等复杂问题封装起来，使得开发分布式数据处理应用变得更加简单和高效。通过将复杂的任务拆解成Map（映射）和Reduce（归约）两个易于理解的操作，程序员可以轻松地构建出能够处理PB级数据集的强大应用程序。

### 技术架构

MapReduce框架的设计源自于函数式编程领域的Map和Reduce两个概念。在此模型中，计算过程首先通过Map函数对每条输入数据执行映射操作，产生一系列键值对；然后，Reduce函数针对相同键的所有值执行归约操作，最终获得所需的结果集。

该模型的极大优势在于其并行处理能力，能在数量众多的标准配置计算机上高效运行。MapReduce自动处理数据切分、任务分配、故障恢复以及集群内节点间的协调通信等繁琐细节，极大降低了并行计算的复杂性。

如在Google内部，MapReduce框架被部署于庞大的服务器群组上，通常这些集群由成千上万台服务器构成，日常处理多达数TB的数据量。由此，Google日复一日地运行着上千个MapReduce作业，处理各种数据分析和计算任务。

#### MapReduce工作步骤

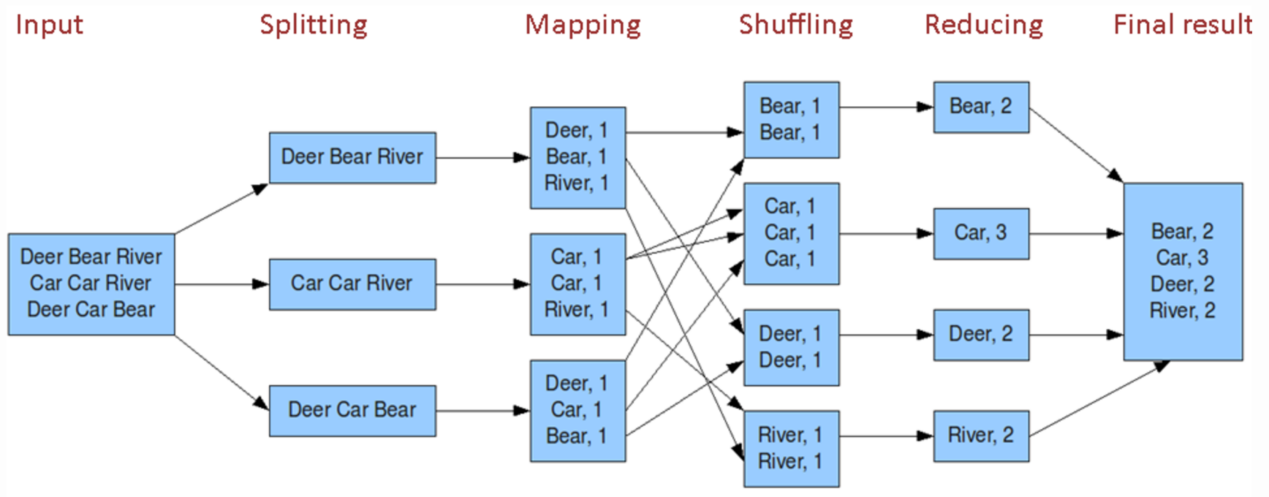


图3-4 任务执行概览

当用户调用MapReduce函数时，将发生如上图所示的一系列动作：

1、输入数据分片（Input Splitting）：数据集被分割成多个小块，通常是文件系统中的数据块大小，如HDFS中的块。

2、Map阶段（Mapping）:对每个分片执行Map操作，由用户定义的Map函数处理输入数据块。Map函数接收输入键值对，处理后生成中间键值对。

3、洗牌阶段（Shuffling）:系统自动将所有Map操作生成的中间键值对进行排序和分组，确保相同键的值发送到同一个Reduce任务。

4、Reduce阶段（Reducing）:对每个键及其相关的值集合执行Reduce操作。Reduce函数合并那些具有相同键的中间数据值，输出最终结果。

5、最终阶段（Final Result）：Reduce操作的结果被写入到文件系统中，通常是HDFS中，形成最终的输出数据集

最后在成功完成任务之后，MapReduce的输出存放在R个输出文件中（对应每个Reduce任务产生一个输出文件，文件名由用户指定）。一般情况下，用户不需要将这R个输出文件合并成一个文件–他们经常把这些文件作为另外一个MapReduce的输入，或者在另外一个可以处理多个分割文件的分布式应用中使用。同时，框架还能进行错误处理，如重启失败的任务，这使得MapReduce可以确保整个计算过程的容错性。

#### 关键词统计任务示例

在学术平台进行检索时，可能需要关键词来进行检索，同时需要对关键词进行统计和分类，这，可以这样来设计Map和Reduce函数：

Map函数读取每一篇论文的标题和摘要，分词后输出每个关键词及其出现次数（在这个例子里是1）。

Reduce函数把每个词的计数累加起来。

伪代码如下：

map<String key, String value>:

*// key: paper id*

*// value: title + abstract of the paper*

for each word w in value:

EmitIntermediate(w, “1″);

reduce<String key, Iterator values>:

*// key: a word*

*// values: a list of counts*

int result = 0;

for each v in values:

result += ParseInt(v);

Emit(AsString(result));

#### 其它任务实例

1、图像处理：如对一个图库中的图像进行分类。Map函数对每张图片进行识别和分类，Reduce函数则统计每个类别的图像数量。

2、天气数据分析：如为了统计一定时期北京各区的平均气温。Map函数对于每条气温记录输出地区标识和对应气温。Reduce函数把相同地区的气温记录下来，并计算平均值。

日志分析：如需分析web服务器日志以确定最频繁访问的网页。Map函数对于每个日志条目，输出访问的URL和出现次数（计数为1）。Reduce函数对于每个URL，累加其被访问的次数。

4、社交网络分析：如需计算社交网络中每个用户的朋友数量以进行初步的社交网络分析。Map函数对于社交网络中的每个用户，输出该用户与其每个朋友的配对关系。Reduce函数对于每个用户，统计其朋友的总数。

5、机器算法学习：使用MapReduce来运行大规模机器学习算法，如K-Means聚类。Map函数负责分配数据点到最近的中心点，Reduce函数则负责对每个类别计算新的中心点。

#### 故障处理

MapReduce是为了处理大规模数据而设计的，因此故障处理机制是其核心特性之一。以下是MapReduce的主要故障处理机制：

任务重试（Task Retry）：

当一个任务失败时，MapReduce框架会自动重新安排该任务在另一台机器上执行。这通常是由于个别机器上的软件或硬件故障。

数据备份（Data Replication）

输入和输出数据通常在HDFS（Hadoop Distributed File System）上有多个副本，如果处理某个数据块的节点失败，其他节点上的副本可以用来重新执行该任务。

心跳检测（Heartbeat Monitoring）

工作节点会定期向主节点（如Hadoop中的JobTracker）发送心跳信号。如果主节点在预定时间内没有收到某个节点的心跳信号，它会将该节点标记为失败，并将其任务分配给其他节点。

节点黑名单（Node Blacklisting）

如果某个节点连续失败了一定次数的任务，MapReduce框架可能会认为这个节点不可靠，将其加入黑名单，不再分配任务给这个节点。

超时机制（Timeout Mechanism）

MapReduce框架设有超时机制，如果任务执行超过一定时间没有进展，系统会将其标记为失败，并重新调度。

容错计算（Fault-Tolerant Computation）

Map和Reduce操作都是无状态的，并且只有在操作完全完成后才会对外提交结果，这意味着任何时候失败的任务都可以安全地重新计算而不影响最终结果。

## Redis

### 相关问题

在21世纪初，互联网上的内容不断膨胀，而当时的数据库市场主要由传统的关系型数据库系统主导，这些系统在面对不断增加的互联网流量和实时数据需求时表现出一些明显的弊端和限制。传统数据库系统面临读写性能瓶颈，因为它们通常基于磁盘存储数据，而磁盘访问速度相对较慢，导致了性能瓶颈。此外，这些数据库系统的数据模型复杂，需要大量时间和资源来设计和维护，而高并发读写要求更多的硬件和管理员，增加了成本和复杂性。缺乏弹性也是问题之一，因为传统数据库难以在不停机的情况下扩展，这对于处理突发的数据量增加或高负载时，可能导致系统不稳定或服务中断。

Redis的诞生解决了这些问题，并具有内存存储、简单的数据模型、高性能和低延迟等优势。Redis主要将数据存储在内存中，这使得它能够提供非常快速的读写访问，远远超过了传统数据库系统。这种内存存储还使得Redis成为优秀的缓存系统，有效减轻了后端数据库的负担。Redis的数据模型相对简单，包括字符串、列表、集合、有序集合和哈希等数据结构，简单性有助于更快速的应用开发。由于Redis数据存储在内存中，它能够以极低的延迟提供数据访问，适合实时应用程序、消息队列、计数器和实时分析等应用。此外，Redis支持分布式部署，允许将数据分布在多个服务器上，以实现更高的容量和可用性，这使得它适用于大规模互联网应用，能够应对高并发的情况。简单概括：Redis的诞生背景源于当时数据库市场存在的性能、复杂性和扩展性等问题，Redis通过采用内存存储、简单的数据模型和高性能特点，填补了这些不足，成为一个强大的数据存储和缓存解决方案，被广泛用于互联网应用程序和实时数据处理场景。其设计理念和优势使其成为当时市场上的一个重要创新，为高性能数据存储提供了新的方向。

### 技术架构

#### 架构概览

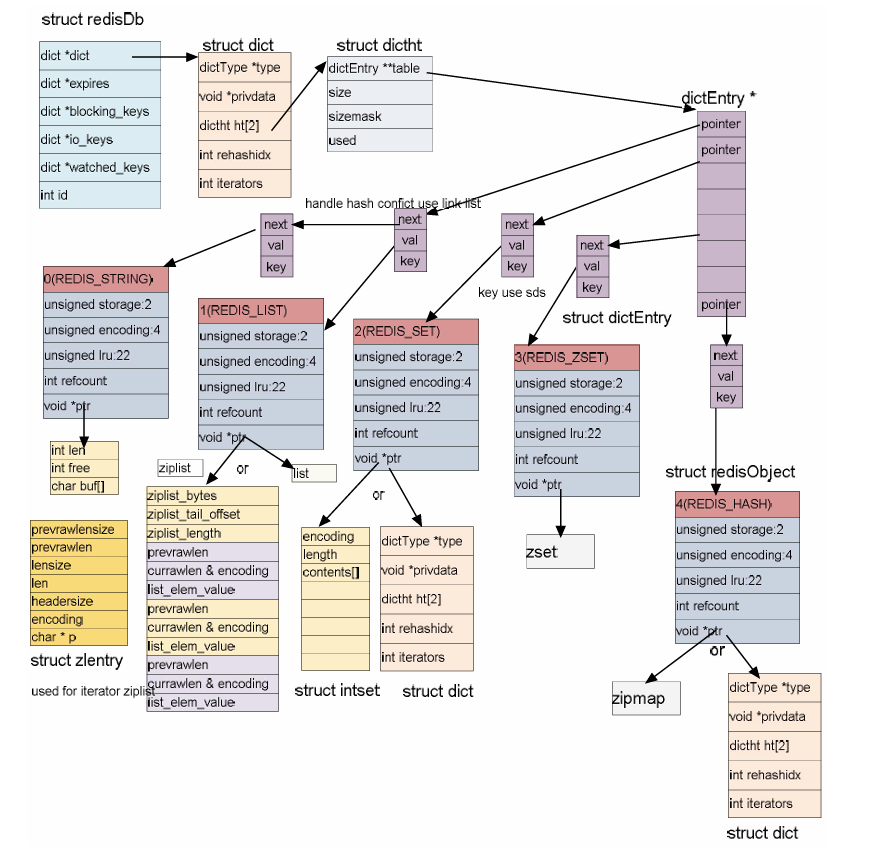
Redis 是支持多key-value 数据库(表)的,并用 RedisDb 来表示一个key-value 数据库

图3-6 Redis 内存架构

redisServer 中有一个 redisDb \*db; 成员变量, RedisServer 在初始化 时,会根据配置文件的 db 数量来创建一个 redisDb 数组. 客户端在连接后,通过 SELECT 指令来选择一个 reidsDb,如果不指定,则缺省是 redisDb 数组的第 1 个 (即下标是 0 ) redisDb. 一个客户端在选择 redisDb 后,其后续操作都是在此 redisDb 上进行的. 下面会详细介绍一下 redisDb 的内存结构.redisServer 中有一个 redisDb \*db; 成员变量, RedisServer 在初始化 时,会根据配置文件的 db 数量来创建一个 redisDb 数组. 客户端在连接后,通过 SELECT 指令来选择一个 reidsDb,如果不指定,则缺省是 redisDb 数组的第 1 个 (即下标是 0 ) redisDb. 一个客户端在选择 redisDb 后,其后续操作都是在此 redisDb 上进行的.

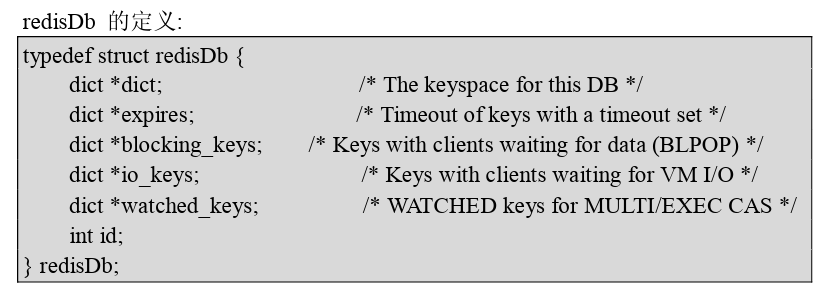


图3-7 redisDb 定义

struct redisDb 中 ,dict 成员是与实际存储数据相关的. dict 的定义如下:

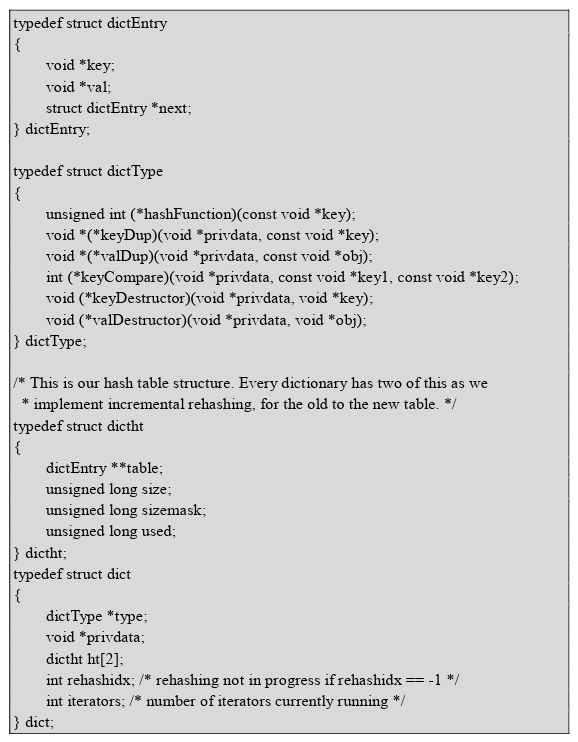


图3-8 dict 定义

dict 是主要是由 struct dictht 的 哈唏表构成的, 之所以定义成长度为2的 ( dictht ht[2] ) 哈唏表数组,是因为 redis 采用渐进的 rehash,即当需要 rehash 时, 每次像 hset,hget 等操作前,先执行N 步 rehash. 这样就把原来一次性的 rehash 过程拆散到进行, 防止一次性 rehash 期间 redis 服务能力大幅下降. 这种渐进 的 rehash 需要一个额外的 struct dictht 结构来保存. struct dictht 主要是由一个 struct dictEntry 指针数组组成的, hash 表的冲突 是通过链表法来解决的. struct dictEntry 中的 key 指针指向用 sds 类型表示的 key 字符串, val 指 针指向一个 struct redisObject 结构体, 其定义如下:

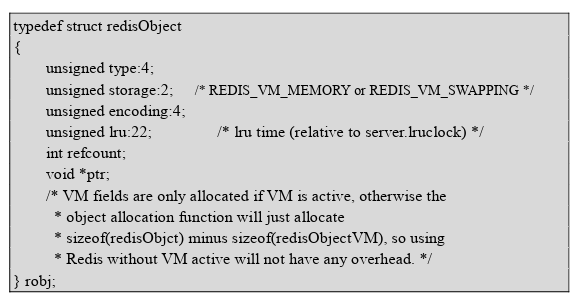


图3-9 redisObject结构体定义

* + - 1. 数据结构

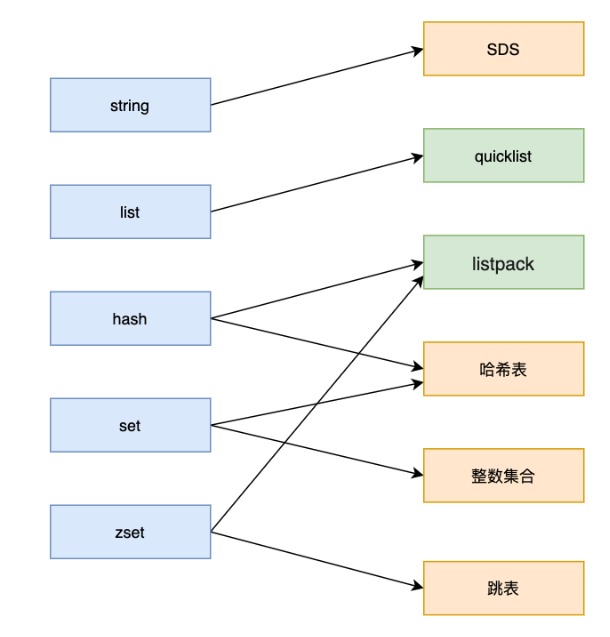
Redis是一个非关系型数据库，通常被归类为键值存储（Key-Value Store）数据库。在Redis中，数据以键值对的形式存储，其中键是唯一的标识符，而值可以是各种不同类型的数据结构。

图3-10 Redis 数据结构

在Redis中，你可以根据键来查询对应的值，而不需要像传统关系型数据库指定行、列和版本。zipmap 结构

如果 redisObject 的 type 成员值是 REDIS\_HASH 类型的,则当该 hash 的 entry 小于配置值: hash-max-zipmap-entries 或者 value 字符串的长度小于 hash-max-zipmap-value, 则可以编码成 REDIS\_ENCODING\_ZIPMAP 类型存储, 以节约内存. 否则采用 Dict 来存储. zipmap 其实质是用一个字符串数组来依次保存key和value,查询时是依次遍 列每个 key-value 对,直到查到为止. 其结构示意图如下:

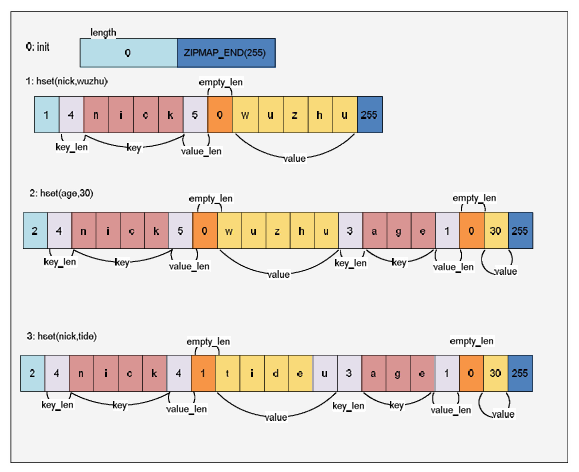


图3-11 zipmap数据结构

如果redisObject的type 成员值是 REDIS\_LIST 类型的,则当该list 的 elem数小 于配置值: hash-max-ziplist-entries 或 者 elem\_value 字符 串的长度小于 hash-max-ziplist-value, 则可以编码成 REDIS\_ENCODING\_ZIPLIST 类型存储, 以节约内存. 否则采用 Dict 来存储. ziplist 其实质是用一个字符串数组形式的双向链表. 其结构示意图如下:

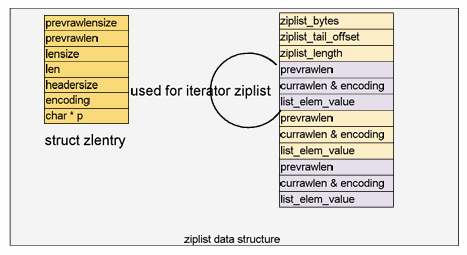


图3-12 ziplist数据结构

adlist为常见的双向列表，其结构定义如下：

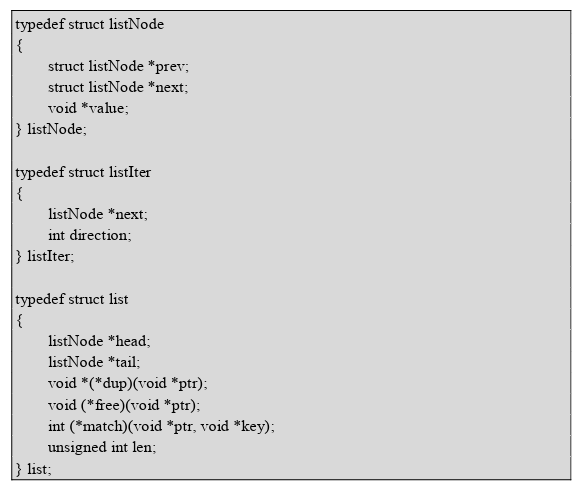


图3-13 adlist 定义

intset 是用一个有序的整数数组来实现集合(set). struct intset 的定义如下:

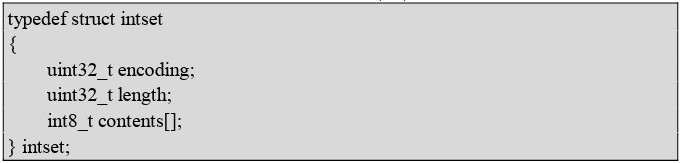


图3-14 intset 定义

Redis中的Skip List（跳跃表）是一种用于实现有序集合数据结构的底层数据结构。Skip List是一种有序数据结构，它允许高效地执行范围查询和插入操作，因此非常适合用于实现有序集合，如有序集合数据类型（Sorted Set）。

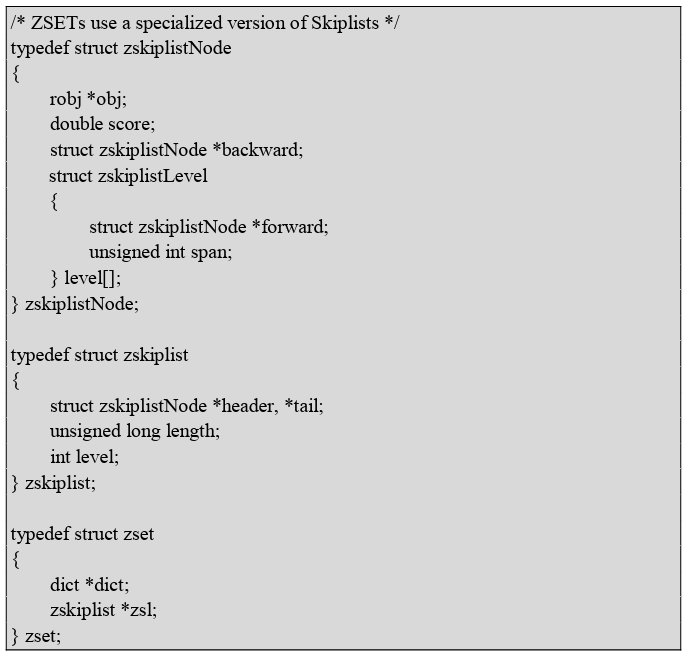


图3-15 skipList 定义

zset 的实现用到了2个数据结构: hash\_table 和 skip list (跳跃表),其中 hash table 是使用 redis 的 dict 来实现的,主要是为了保证查询效率为 O(1),而 skip list (跳跃表) 是用来保证元素有序并能够保证 INSERT 和 REMOVE 操作是 O(logn)的复杂度。

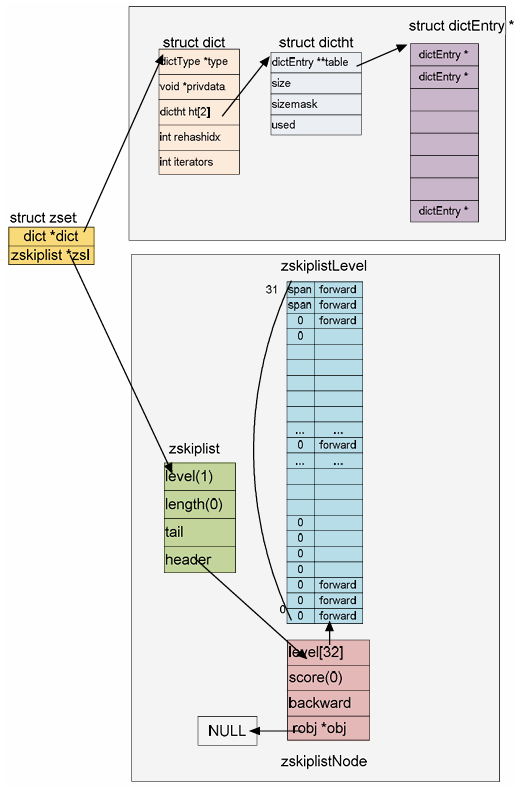


图3-15 zset数据结构图

# 非功能特性分析

* 1. 扩展性分析

大数据系统需要能够处理大量数据，同时还能够应对不断增长的数据量。可扩展性包括水平扩展和垂直扩展，以确保系统能够满足不断增长的需求。

### HDFS（Hadoop Distributed File System）

HDFS具有很高的扩展性，能够有效地处理大规模数据存储需求。这是因为它将数据划分成块，并分布在多个节点上，这些节点可以随时扩展。当需要更多存储容量时，可以简单地添加新的存储节点，而无需对整个系统进行重大改变。HDFS还支持数据的冗余存储，通过多次复制数据块以提高容错性。这种冗余机制进一步增加了系统的扩展性，因为即使某些节点出现故障，数据仍然可以访问。

### MapReduce

MapReduce是一种编程模型，用于处理大规模数据集，通常与Hadoop一起使用。其扩展性取决于底层的分布式计算框架，通常是Hadoop MapReduce。MapReduce任务可以非常容易地分发到集群中的多个节点上，充分利用并行计算。这使得MapReduce非常适合处理大规模数据，可以通过添加更多的计算节点来水平扩展计算能力。MapReduce的扩展性还受到编写的Map和Reduce任务的质量和效率的影响。优化任务可以提高性能和扩展性。

### Redis

Redis是一个内存数据库系统，其扩展性在某种程度上受到内存和计算资源的限制。因为数据通常存储在内存中，所以存储大规模数据集需要足够的内存容量。Redis支持分片（sharding），允许数据分布在多个Redis节点上。这提高了性能和可伸缩性，但需要管理多个Redis实例。Redis还可以配置为运行在主从复制模式，以提高可用性和数据冗余，但这也需要更多的节点。

## 可靠性分析

可靠性涉及到系统的稳定性和持久性。大数据系统应当能够持续工作，不容易崩溃或丢失数据。

### 创建冗余以实现更高的可用性

具有高可靠性需求的系统必须避免单点故障，并且必须在多个故障网域之间复制其资源。故障网域是可以单独出现故障的资源池，例如虚拟机实例、可用区或地区。HDFS通过数据的冗余副本机制来实现高可用性。通过在不同的故障域中存储数据副本，HDFS还可以防止在某个故障域出现问题时导致数据不可用。

### 跨区域复制数据以进行灾难恢复

将数据复制或归档到远程地区，以便在发生地区级服务中断或数据丢失时进行灾难恢复。在使用复制功能时，除了因复制延迟而可能丢失少量数据之外，远程区域中的存储系统已经具有几乎最新的数据，因此恢复速度会更快。使用定期归档而非持续复制时，灾难恢复涉及从新区域的备份或归档中恢复数据。此过程通常会导致服务停机时间比激活持续更新的数据库副本更长，并且可能因连续备份操作之间的时间间隔而造成数据丢失。无论使用何种方法，整个应用堆栈都必须重新部署并在新区域中启动，并且在发生这种情况时服务将不可用。HDFS还提供了故障恢复机制，可以检测并自动修复数据块损坏或丢失的情况。这有助于保持数据的完整性和可靠性。

### 故障恢复机制

MapReduce底层环境提供了故障恢复机制，以保证任务的可靠执行。MapReduce任务通常可以重新启动或重新分配给其他节点，以处理节点故障或任务失败的情况。这有助于确保作业的完成和结果的正确性。在MapReduce中，开发人员通常可以编写容错的Mapper和Reducer任务，以处理数据处理过程中的错误情况。

### 高速缓存和消息队列

Redis可以配置为在内存中保存数据，并定期将数据快照到磁盘，以防止数据丢失。这提供了一定程度的数据可靠性。Redis还支持主从复制，以提高可用性和容错性。如果主节点发生故障，备份节点可以接管服务，减少中断时间。

### 过载时逐步降低服务级别

服务应该检测过载，并向用户返回降低质量的响应或减少部分流量，而不是在遇到过载时完全不能工作。例如，服务可以使用静态网页响应用户请求，并暂时停用处理费用较昂贵的动态效果。

### 预防和缓解流量峰值

在服务器端实施高峰缓解策略，例如限制、排队、减载或熔断、优雅降级和确定关键请求的优先级。客户端上的缓解策略包括客户端限制和使用抖动的指数退避算法。如果问题导致发生故障，系统组件应以可让整个系统继续运行的方式发生故障。这些问题可能为软件错误、输入或配置有误、计划外实例中断或人为错误。

## 性能分析

### HDFS的性能优化

#### 数据块大小的传输

HDFS的数据块大小是一个重要的性能参数。较小的数据块适合存储大量小文件，因为它们减少了空间浪费。较大的数据块适用于大型数据集，因为它们减少了元数据的开销，提高了数据的存储和访问效率。选择适当的数据块大小取决于应用需求。如果工作负载包含大量小文件，可以考虑减小数据块大小。如果有大型数据集，可以增大数据块大小以降低元数据管理的开销。调整数据块大小需要仔细考虑，并可能需要重新加载数据以适应新的块大小。

#### 副本数优化

HDFS默认会将数据块复制到三个不同的DataNode上，以提供数据冗余和容错性。然而，根据存储需求和性能要求，可以考虑调整副本数。减少副本数量可以降低存储开销，因为每个副本都占用存储空间。但要注意，减少副本数量会降低数据的容错性，因此需要在性能和容错之间进行权衡。对于重要的数据，可以保留更多的副本以提高容错性。对于临时或不太重要的数据，可以减少副本数量以节省存储空间。

#### 数据本地性

数据本地性是指尽量在计算节点上执行数据处理，以减少数据的网络传输。优化数据本地性可以显著提高HDFS性能。为了实现数据本地性，可以考虑以下几个方面：

将计算任务分配给距离数据最近的节点。

使用HDFS的块放置策略，确保数据块在计算节点的本地磁盘上。

避免数据移动，尽量减少数据的复制和传输。

### Redis的性能优化

#### 写缓存

设立写缓存Memtable(内存表)，当客户端请求写入数据时直接写入内存中的 Memtable 后就返回成功，写入的数据在Memtable中将按照Key排序，当内存表中写入的数据达到一定大小后，将数据刷写到磁盘磁盘文件中，称之为SSTable，SSTable 一旦写入完成后续将不会变更其中的数据（只会在合并后删除），后续如果有对同一个Key的Value的变更或删除也只会是新的SSTable中的一条记录(删除也是一条数据，只是它的类型是delete，称之为墓碑标记)。Memtable在这里起到了对写数据进行集散的作用，从而能够实现写入BigTable的速度要远快于对磁盘的随机读写，且能一次写入多条数据。

#### 读缓存

设立读缓存，Bigtable有Scan Cache和BlockCach两级缓存，Scan Cache缓存的是刚Scan的数据，对于经常要重复读取相同数据的应用程序来说，Scan Cache非常有效；Block Cache 缓存的是从GFS读取的SSTable的Block，当经常要读取刚刚读过的数据附近的数据时，BlockCache 很有用。

### MapReduce的性能优化

#### 本地存储

MapReduce所需的输入数据大部分都存储在集群中机器的本地磁盘上由GFS管理，master在调度Map任务时会考虑输入文件的位置信息，尽量将一个任务调度在包含所需数据的备份的机器上或者附近的机器上执行，这样可以节省网络带宽。当在一个足够大的集群上运行MapReduce时，大部分的输入数据都能从本地机器读取。

#### 任务粒度

为了使上述的本地存储优化策略发挥最大效用，一个Map Worker理想情况下需要处理16M到64M的输入数据，而Reduce Worker的数量通常使用户所需的输出文件数量而定，因此需要考虑Map Worker的数量；master在对Map和Reduce任务调度时需要消耗一定的计算资源，master机器的内存中还需要存储Map Worker和Reduce Worker的对应关系，这需要浪费一定的存储资源，为了尽量避免浪费，Map Worker的数量也需要加以限制。

#### 备用任务

在实际的应用场景中，可能会出现一部分机器花了很长时间才完成最后的几个任务，导致总操作时间过长，这一般是由于硬件损坏或者系统为这台机器调度了其他任务导致的。为了减少此类情况的发生，当一个MapReduce操作接近完成时，master会调度备用(backup)任务来执行剩下的处于处理状态中的任务，原进程和备用进程中的任意一个完成了该任务，都视作任务已完成。这个机制通常只比正常操作多占用几个百分点的计算资源，但却对减少总处理时间很有效。

* 1. 安全性分析

大数据系统中的数据通常是敏感的，因此必须具备强大的安全性措施，包括数据加密、身份验证、授权等。

### HDFS（Hadoop Distributed File System）

HDFS提供了一些安全机制，包括访问控制列表（ACLs）、身份验证和授权。它可以集成到Kerberos等身份验证系统中，以确保只有授权用户可以访问数据。HDFS还支持数据加密，包括数据传输和数据在磁盘上的加密。这有助于保护数据的机密性。HDFS可以配置为使用访问令牌（Token）来控制客户端对数据的访问，从而提供额外的安全性。

### MapReduce

MapReduce框架本身没有强制性的安全性机制，但通常在底层的Hadoop集群上实施安全性。Hadoop提供了用户身份验证和授权机制，以确保只有授权用户可以提交作业和访问集群资源。这可以集成到Kerberos等身份验证系统中。作业和任务数据可以加密，以保护数据的机密性。

### Redis

Redis的安全性较为有限，因为它最初是设计用于内部使用和性能，而不是强调安全性。Redis可以配置为需要密码进行身份验证，以防止未授权访问。但是，密码是明文存储在配置中的，因此需要小心管理。Redis 6及更高版本引入了ACL（访问控制列表）系统，允许更细粒度的用户和命令级别的控制。在实际使用中，建议采取额外的安全措施，例如网络隔离和定期审计，以确保这些系统的安全性

# 调研总结

经过对大数据典型技术的调研，可以看出在处理大数据问题时，多用分布式技术和缓存技术来处理，提高效率。我们也考虑在自己系统的架构之中引入分布式部署策略和缓存。