

Contents

分布式系统作业-1

1. 为什么有时候最大程度地实现透明性并不总是好的？

① 透明性的实现需要额外的计算和性能开销。过度地实现透明性可能会导致系统性能地下降。② 透明性会隐藏一些系统内部实现的细节，同时还会增加系统实现的复杂度。过度地实现透明性可能会导致系统难以调试和优化。当系统发生错误时，我们或许会难以区分失效和性能变慢的节点，或者是不能确定系统失效前的操作是什么。

2. 可以通过应用多种技术来取得扩展性，都有哪些技术？

① 隐藏通信延迟，尽量避免等待远程服务对请求的响应。这可以通过利用异步通信技术、设计分离的响应消息处理器以及将计算任务移动从服务端移动到客户端来实现。② 分布式技术，在多个机器上划分数据和计算。③ 副本和缓存技术，在多个不同的机器上创建多个数据副本。

3. 分布式系统的软件体系结构有哪几种类型，各自有什么优缺点？

• 分层结构

— 优点：

- * (1) 系统被划分为一个层次结构，每层具有高内聚性，每一层的功能清晰明确，便于维护和扩展；
- * (2) 支持软件资源的复用：软件层与下层通过特定 API 进行交互，只要保持 API 不变，可以使用不同的实现方式实现下层应用。

— 缺点：

- * (1) 设计难度大，将系统合理地分解为层次结构并不容易；
- * (2) 数据需要在不同的层间进行传输，增加了通信的开销，影响系统的性能。

• 面向对象结构

— 优点：

- * (1) 代码的抽象性和重用性高，便于代码的组织管理和维护；
- * (2) 模块化与高内聚的设计理念降低了系统模块之间的耦合度，便于组件的开发与系统的扩展；
- * (3) 多态机制使得系统能够以统一的方式处理不同类型的对象，增加了系统的灵活性和适应性。

— 缺点：

- * (1) 高聚合的对象之间的通信将会引入额外的性能开销；
- * (2) 当一个对象的改变时，所有调用它的对象也要连锁地作出相应的调整。

• 事件驱动结构

— 优点：

- * (1) 提高了系统的可复用性，构建名和事件名所注册的过程名不变，原构件可以被重新构建，提高系统的适应性；
- * (2) 组件之间通过事件通信，降低了组件之间的耦合度，提高了系统的灵活性与可扩展性；
- * (3) 事件的生产 and 消费可以异步进行，提高系统的响应速度。

— 缺点：

- * 系统设计难度大。需要设计合理的事件处理顺序、数据同步机制、事件一致性处理机制等。

• 共享数据空间结构

— 优点：

- * 简化了组件之间的通信，通过共享的数据空间可以方便地实现组件之间数据的共享交换，这个优点在数据量越大时越为明显。

— 缺点：

- * (1) 共享数据将会引入数据的一致性和竞态条件，增大系统的设计复杂度；
- * (2) 可能会导致系统的可靠性下降，如果共享数据空间出现故障或数据丢失，可能会影响整个系统的运行。

4. 点对点网络中，并不是每个节点都能成为超级对等节点，满足超级对等节点的合理要求是什么？

节点需要是高度可用的，因为其他节点需要依赖该节点。并且，它应该有足够的能力处理各种请求。最重要的是依靠它能够高效快速地处理任务。

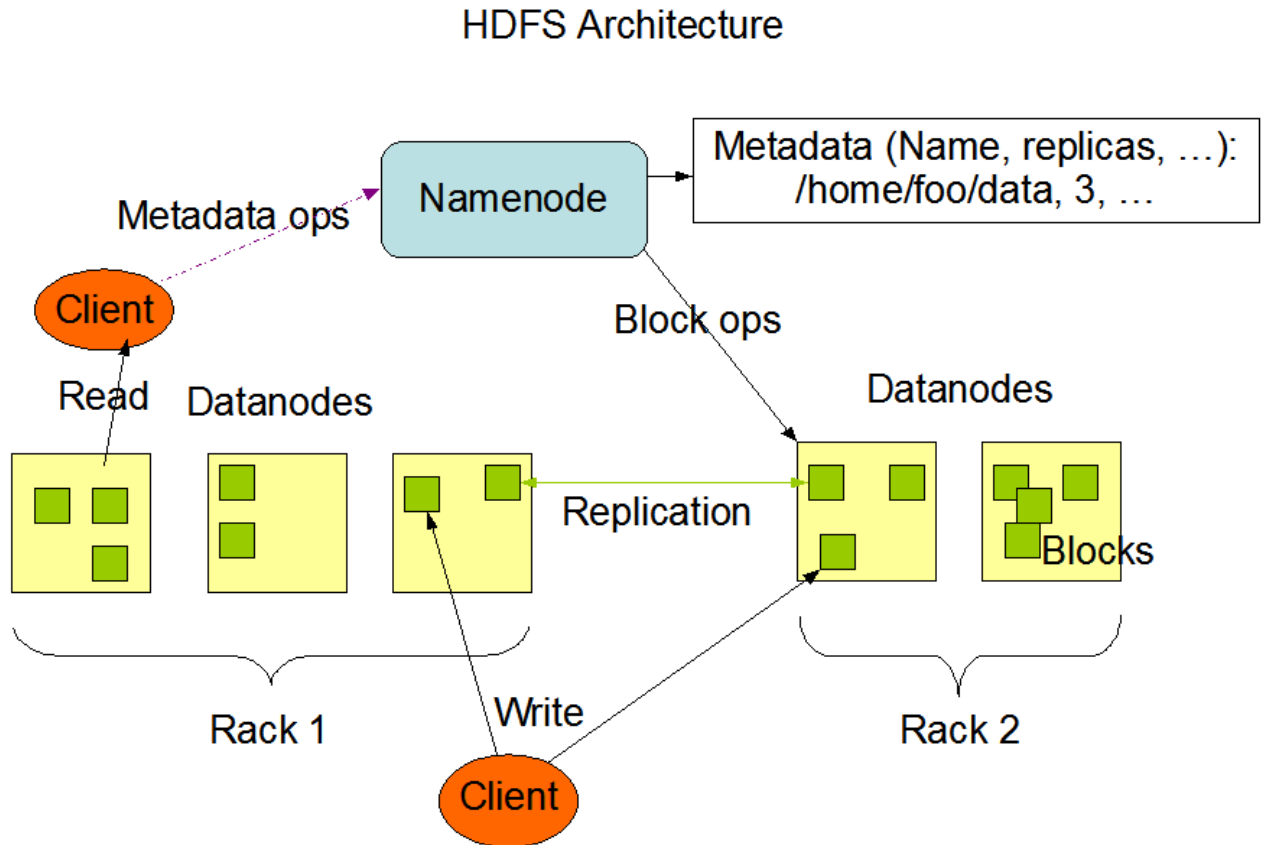
5. 代码迁移有哪些场景？为什么要进行代码迁移？

从单机部署迁移到分布式部署、从一个平台迁移到另一个系统平台等。代码迁移的目的可能是提高系统的可扩展性、可靠性和性能，也可能是为了满足更多用户的使用需求，也可能是原平台不可用需要迁移至新平台。

6. 请从一些开源分布式软件如 Hadoop、Ceph 分布式文件系统、Apache Httpd、Spark 等找出至少 2 处能够体现透明性的样例代码，并解释是何种类型的透明性。

分析 Hadoop

首先需要知道，Hadoop 的 HDFS 采用 master/slave 架构。一个 Hadoop 分布式文件系统（HDFS）集群是由一个 Namenode 和一定数目的 Datanodes 组成。Namenode 是一个中心服务器，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。集群中的 Datanode 一般是一个节点一个，负责管理它所在节点上的存储。HDFS 暴露了文件系统的名字空间，用户能够以文件的形式在上面存储数据。从内部看，一个文件其实被分成一个或多个数据块，这些块存储在一组 Datanode 上。Namenode 执行文件系统的名字空间操作，比如打开、关闭、重命名文件或目录。它也负责确定数据块到具体 Datanode 节点的映射。Datanode 负责处理文件系统客户端的读写请求。在 Namenode 的统一调度下进行数据块的创建、删除和复制。



从上述介绍中得知，一个文件被分成一个或多个数据块，且这些块存储在一组 Datanode 上。而我们用户在进行操作时，我们仿佛是在对单个系统进行操作，并没有感受到我们的文件可能分布在不同的节点上，其中便蕴含着 HDFS 的透明性。

在这里，我尝试从文件的重命名方法入手，来探索 Hadoop 的透明性实现：

很快，我们可以在 HDFS 中的 `DistributedFileSystem` 类中找到 `rename` 的两个重载：

```

1  @SuppressWarnings("deprecation")
2  @Override
3  public boolean rename(Path src, Path dst) throws IOException {
4      statistics.incrementWriteOps(1);
5      storageStatistics.incrementOpCounter(OpType.RENAME);
6
7      final Path absSrc = fixRelativePart(src);
8      final Path absDst = fixRelativePart(dst);
9
10     // Try the rename without resolving first
11     try {
12         return dfs.rename(getPathName(absSrc), getPathName(absDst));
13     } catch (UnresolvedLinkException e) {
14         // Fully resolve the source
15         final Path source = getFileLinkStatus(absSrc).getPath();
16         // Keep trying to resolve the destination
17         return new FileSystemLinkResolver<Boolean>() {
18             @Override
19             public Boolean doCall(final Path p) throws IOException {
20                 return dfs.rename(getPathName(source), getPathName(p));
21             }
22             @Override
23             public Boolean next(final FileSystem fs, final Path p)
24                 throws IOException {
25                 // Should just throw an error in FileSystem#checkPath
26                 return doCall(p);
27             }
28         }.resolve(this, absDst);
29     }
30 }
31
32 /**
33  * This rename operation is guaranteed to be atomic.
34  */
35 @SuppressWarnings("deprecation")
36 @Override
37 public void rename(Path src, Path dst, final Options.Rename... options)
38     throws IOException {
39     statistics.incrementWriteOps(1);
40     storageStatistics.incrementOpCounter(OpType.RENAME);
41     final Path absSrc = fixRelativePart(src);
42     final Path absDst = fixRelativePart(dst);
43     // Try the rename without resolving first
44     try {
45         dfs.rename(getPathName(absSrc), getPathName(absDst), options);
46     } catch (UnresolvedLinkException e) {
47         // Fully resolve the source
48         final Path source = getFileLinkStatus(absSrc).getPath();
49         // Keep trying to resolve the destination
50         new FileSystemLinkResolver<Void>() {
51             @Override
52             public Void doCall(final Path p) throws IOException {
53                 dfs.rename(getPathName(source), getPathName(p), options);
54                 return null;
55             }
56             @Override
57             public Void next(final FileSystem fs, final Path p)
58                 throws IOException {
59                 // Should just throw an error in FileSystem#checkPath
60                 return doCall(p);
61             }
62         }.resolve(this, absDst);
63     }
64 }


```

两个方法之间主要区别在于 options 的传递，其余差别不大。这里直接分析简洁的 rename 方法。可以看到，用户输入的 src 与 dst 首先会被 fixRelativePart 方法解析为绝对地址，诸如/usr/hadoop/file.txt。随后 absSrc 与 absDst 会被传入一个 getPathName 方法内，我们来看一下这个方法：

```
1  /**
2   * Checks that the passed URI belongs to this filesystem and returns
3   * just the path component. Expects a URI with an absolute path.
4   *
5   * @param file URI with absolute path
6   * @return path component of {file}
7   * @throws IllegalArgumentException if URI does not belong to this DFS
8   */
9  String getPathName(Path file) {
10     checkPath(file);
11     String result = file.toUri().getPath();
12     if (!DFSUtilClient.isValidName(result)) {
13         throw new IllegalArgumentException("Pathname " + result + " from " +
14             file+" is not a valid DFS filename.");
15     }
16     return result;
17 }
```

阅读上面的代码，我们发现，在 getPathName 内，我们输入的路径将会在我们不知情的时候解析为一个 URL 路径！我们以为自己在对同一个文件系统进行操作，但是实际上我们的操作将会通过 URL 路径被重定向到不同的节点上的文件上，这便体现了 Hadoop 的位置透明性。

再次阅读代码，我们会发现对于 delete 方法，Hadoop 也有类似的处理：



```
1  @Override
2  public boolean delete(Path f, final boolean recursive) throws IOException {
3      statistics.incrementWriteOps(1);
4      storageStatistics.incrementOpCounter(OpType.DELETE);
5      Path absF = fixRelativePart(f);
6      return new FileSystemLinkResolver<Boolean>() {
7          @Override
8          public Boolean doCall(final Path p) throws IOException {
9              return dfs.delete(getPathName(p), recursive);
10         }
11         @Override
12         public Boolean next(final FileSystem fs, final Path p)
13             throws IOException {
14             return fs.delete(p, recursive);
15         }
16     }.resolve(this, absF);
17 }
```