大数据基础加强阶段

第二天

一、 课程计划

目录

二、 ZooKeeper

1. Zookeeper

1.1. ZooKeeper 概述

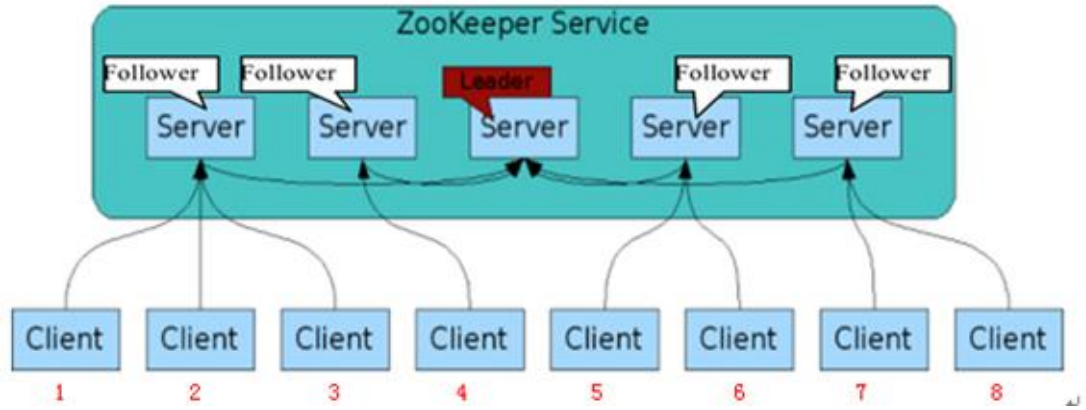
Zookeeper 是一个分布式协调服务的开源框架。主要用来解决分布式集群中应用系统的一致性问题，例如怎样避免同时操作同一数据造成脏读的问题。

ZooKeeper 本质上是一个分布式的小文件存储系统。提供基于类似于文件系统的目录树方式的数据存储，并且可以对树中的节点进行有效管理。从而用来维护和监控你存储的数据的状态变化。通过监控这些数据状态的变化，从而可以达到基于数据的集群管理。诸如：统一命名服务、分布式配置管理、分布式消息队列、分布式锁、分布式协调等功能。

1.2. ZooKeeper 特性

1. 全局数据一致：集群中每个服务器保存一份相同的数据副本，client 无论连接到哪个服务器，展示的数据都是一致的，这是最重要的特征；
2. 可靠性：如果消息被其中一台服务器接受，那么将被所有的服务器接受。
3. 顺序性：包括全局有序和偏序两种：全局有序是指如果在一台服务器上消息 a 在消息 b 前发布，则在所有 Server 上消息 a 都将在消息 b 前被发布；偏序是指如果一个消息 b 在消息 a 后被同一个发送者发布，a 必将排在 b 前面。
4. 数据更新原子性：一次数据更新要么成功（半数以上节点成功），要么失败，不存在中间状态；
5. 实时性：Zookeeper 保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。

1.3. ZooKeeper 集群角色



Leader:

Zookeeper 集群工作的核心

事务请求（写操作）的唯一调度和处理者，保证集群事务处理的顺序性；

集群内部各个服务器的调度者。

对于 create，setData，delete 等有写操作的请求，则需要统一转发给 leader 处理，leader 需要决定编号、执行操作，这个过程称为一个事务。

Follower:

处理客户端非事务（读操作）请求，转发事务请求给 Leader；

参与集群 Leader 选举投票。

此外，针对访问量比较大的 zookeeper 集群，还可新增观察者角色。

Observer:

观察者角色，观察 Zookeeper 集群的最新状态变化并将这些状态同步过来，其对于非事务请求可以进行独立处理，对于事务请求，则会转发给 Leader 服务器进行处理。

不会参与任何形式的投票只提供非事务服务，通常用于在不影响集群事务处理能力的前提下提升集群的非事务处理能力。

1.4. ZooKeeper 集群搭建

Zookeeper 集群搭建指的是 ZooKeeper 分布式模式安装。通常由 2n+1 台 servers 组成。这是因为为了保证 Leader 选举（基于 Paxos 算法的实现）能过得到多数的支持，所以 ZooKeeper 集群的数量一般为奇数。

Zookeeper 运行需要 java 环境，所以需要提前安装 jdk。对于安装 leader+follower 模式的集群，大致过程如下：

- 配置主机名称到 IP 地址映射配置
- 修改 ZooKeeper 配置文件
- 远程复制分发安装文件
- 设置 myid
- 启动 ZooKeeper 集群

如果要想使用 Observer 模式，可在对应节点的配置文件添加如下配置：

```
peerType=observer
```

其次，必须在配置文件指定哪些节点被指定为 Observer，如：

```
server.1:localhost:2181:3181:observer
```

详细步骤请参考附件安装资料。

2. ZooKeeper shell

2.1. 客户端连接

运行 `zkCli.sh -server ip` 进入命令行工具。

输入 help，输出 zk shell 提示：

```
ZooKeeper -server host:port cmd args
        stat path [watch]
        set path data [version]
        ls path [watch]
        delquota [-n|-b] path
        ls2 path [watch]
        setAcl path acl
        setquota -n|-b val path
        history
        redo cmdno
        printwatches on|off
        delete path [version]
        sync path
        listquota path
        rmr path
        get path [watch]
        create [-s] [-e] path data acl
        addauth scheme auth
        quit
        getAcl path
        close
        connect host:port
```

2.2. shell 基本操作

创建节点

create [-s] [-e] path data acl

其中，-s 或-e 分别指定节点特性，顺序或临时节点，若不指定，则表示持久节点；acl 用来进行权限控制。

创建顺序节点：

```
[zk: node-22(CONNECTED) 4] create -s /test 123
Created /test0000000003
```

创建临时节点：

```
[zk: node-22(CONNECTED) 5] create -e /test-temp 123temp
Created /test-temp
```

创建永久节点：

```
[zk: node-22(CONNECTED) 1] create /test-p 123p
Created /test-p
```

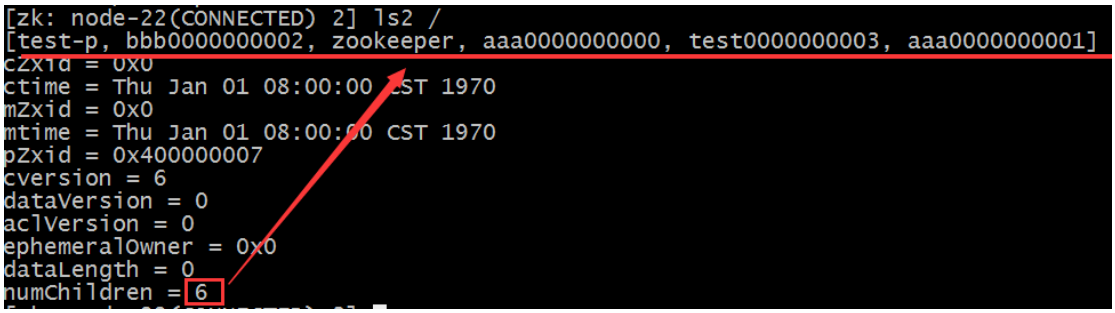
读取节点

与读取相关的命令有 ls 命令和 get 命令，ls 命令可以列出 Zookeeper 指定节点下的所有子节点，只能查看指定节点下的第一级的所有子节点；get 命令可以获取 Zookeeper 指定节点的数据内容和属性信息。

ls path [watch]

get path [watch]

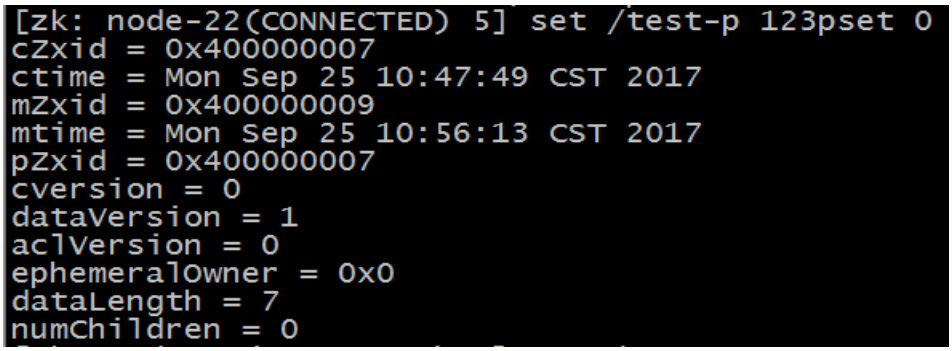
ls2 path [watch]



更新节点

set path data [version]

data 就是要更新的新内容，version 表示数据版本。



现在 dataVersion 已经变为 1 了，表示进行了更新。

删除节点

delete path [version]

若删除节点存在子节点，那么无法删除该节点，必须先删除子节点，再删除父节点。

Rmr path

可以递归删除节点。

quota

setquota -n|-b val path 对节点增加限制。

n:表示子节点的最大个数

b:表示数据值的最大长度

val:子节点最大个数或数据值的最大长度

path:节点路径

```
[zk: node-22(CONNECTED) 13] setquota -n 2 /quota
Comment: the parts are option -n val 2 path /quota
```

listquota path 列出指定节点的 quota

```
[zk: node-22(CONNECTED) 14] listquota /quota
absolute path is /zookeeper/quota/quota/zookeeper_limits
Output quota for /quota count=2,bytes=-1
Output stat for /quota count=1,bytes=1
```

子节点个数为 2,数据长度-1 表示没限制

delquota [-n|-b] path 删除 quota

其他命令

history ： 列出命令历史

```
[zk: node-22(CONNECTED) 16] history
6 - get /test-p
7 - delete /test-p
8 - ls
9 - ls /
10 - stat /
11 - ls /
12 - create /quota 1
13 - setquota -n 2 /quota
14 - listquota /quota
15 - delquota -n /quota
16 - history
```

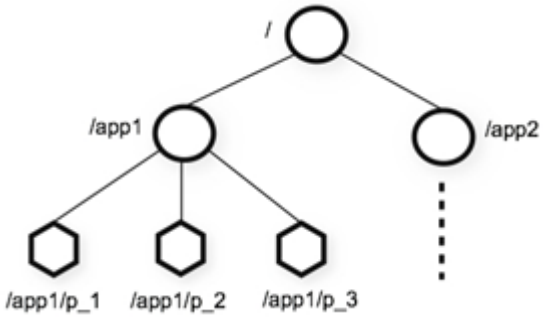
redo：该命令可以重新执行指定命令编号的历史命令,命令编号可以通过 history 查看

3. ZooKeeper 数据模型

ZooKeeper 的数据模型，在结构上和标准文件系统的非常相似，拥有一个层次的命名空间，都是采用树形层次结构，ZooKeeper 树中的每个节点被称为—Znode。和文件系统的目录树一样，ZooKeeper 树中的每个节点可以拥有子节点。但也有不同之处：

1. Znode 兼具文件和目录两种特点。既像文件一样维护着数据、元信息、ACL、时间戳等数据结构，又像目录一样可以作为路径标识的一部分，并可以具有子 Znode。用户对 Znode 具有增、删、改、查等操作（权限允许的情况下）。
2. Znode 具有原子性操作，读操作将获取与节点相关的所有数据，写操作也将替换掉节点的所有数据。另外，每一个节点都拥有自己的 ACL(访问控制列表)，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作。
3. Znode 存储数据大小有限制。ZooKeeper 虽然可以关联一些数据，但并没有被设计为常规的数据库或者大数据存储，相反的是，它用来管理调度数据，比如分布式应用中的配置文件信息、状态信息、汇集位置等等。这些数据的共同特性就是它们都是很小的数据，通常以 KB 为大小单位。ZooKeeper 的服务器和客户端都被设计为严格检查并限制每个 Znode 的数据大小至多 1M，当时常规使用中应该远小于此值。
4. Znode 通过路径引用，如同 Unix 中的文件路径。路径必须是绝对的，因此他们必须由斜杠字符来开头。除此以外，他们必须是唯一的，也就是说每一个路径只有一个表示，因此这些路径不能改变。在 ZooKeeper 中，路径由 Unicode 字符串组成，并且有一些限制。字符串"/zookeeper"用以保存管理信息，比如关键配额信息。

3.1. 数据结构图



图中的每个节点称为一个 Znode。 每个 Znode 由 3 部分组成:

① stat：此为状态信息，描述该 Znode 的版本，权限等信息

② data：与该 Znode 关联的数据

③ children：该 Znode 下的子节点

3.2. 节点类型

Znode 有两种，分别为临时节点和永久节点。

节点的类型在创建时即被确定，并且不能改变。

临时节点：该节点的生命周期依赖于创建它们的会话。一旦会话结束，临时节点将被自动删除，当然可以也可以手动删除。临时节点不允许拥有子节点。

永久节点：该节点的生命周期不依赖于会话，并且只有在客户端显示执行删除操作的时候，他们才能被删除。

Znode 还有一个序列化的特性，如果创建的时候指定的话，该 Znode 的名字后面会自动追加一个不断增加的序列号。序列号对于此节点的父节点来说是唯一的，这样便会记录每个子节点创建的先后顺序。它的格式为“%10d”(10 位数字，没有数值的数位用 0 补充，例如“0000000001”)。

```
[zk: localhost:2181(CONNECTED) 4] create -s /aaa helloallen
Created /aaa0000000000
[zk: localhost:2181(CONNECTED) 5] create -s /aaa helloallen
Created /aaa0000000001
[zk: localhost:2181(CONNECTED) 6] get /aaa
Node does not exist: /aaa
[zk: localhost:2181(CONNECTED) 7] ls /
[zookeeper, aaa0000000000, aaa0000000001]
[zk: localhost:2181(CONNECTED) 8] create -s /bbb helloallen
Created /bbb0000000002
[zk: localhost:2181(CONNECTED) 9] ls /
[bbb0000000002, zookeeper, aaa0000000000, aaa0000000001]
[zk: localhost:2181(CONNECTED) 10]
```

这样便会存在四种类型的 Znode 节点，分别对应：

PERSISTENT：永久节点

EPHEMERAL：临时节点

PERSISTENT_SEQUENTIAL：永久节点、序列化

EPHEMERAL_SEQUENTIAL：临时节点、序列化

3.3. 节点属性

每个 znode 都包含了一系列的属性，通过命令 get，可以获得节点的属性。

```
[zk: node-22(CONNECTED) 2] get /aaa0000000001
hello22
cZxid = 0x200000003
ctime = Fri Sep 22 16:47:35 CST 2017
mZxid = 0x200000007
mtime = Fri Sep 22 17:26:15 CST 2017
pZxid = 0x200000003
cversion = 0
dataVersion = 2
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 7
numChildren = 0
```

dataVersion：数据版本号，每次对节点进行 set 操作，dataVersion 的值都会增加 1（即使设置的是相同的数据），可有效避免了数据更新时出现的先后顺序问题。

cversion ：子节点的版本号。当 znode 的子节点有变化时，cversion 的值就会增加 1。

aclVersion ：ACL 的版本号。

cZxid ：Znode 创建的事务 id。

mZxid ：Znode 被修改的事务 id，即每次对 znode 的修改都会更新 mZxid。

对于 zk 来说，每次的变化都会产生一个唯一的事务 id，zxid（ZooKeeper Transaction Id）。通过 zxid，可以确定更新操作的先后顺序。例如，如果 zxid1 小于 zxid2，说明 zxid1 操作先于 zxid2 发生，zxid 对于整个 zk 都是唯一的，即使操作的是不同的 znode。

ctime：节点创建时的时间戳.

mtime：节点最新一次更新发生时的时间戳.

ephemeralOwner:如果该节点为临时节点，ephemeralOwner 值表示与该节点绑定的 session id. 如果不是，ephemeralOwner 值为 0.

在 client 和 server 通信之前,首先需要建立连接,该连接称为 session。连接建立后,如果发生连接超时、授权失败,或者显式关闭连接,连接便处于 CLOSED 状态，此时 session 结束。

4. ZooKeeper Watcher

ZooKeeper 提供了分布式数据发布/订阅功能，一个典型的发布/订阅模型系统定义了一种一对多的订阅关系，能让多个订阅者同时监听某一个主题对象，当这个主题对象自身状态变化时，会通知所有订阅者，使他们能够做出相应的处理。

ZooKeeper 中，引入了 Watcher 机制来实现这种分布式的通知功能。ZooKeeper 允许客户端向服务端注册一个 Watcher 监听，当服务端的一些事件触发了这个 Watcher，那么就会向指定客户端发送一个事件通知来实现分布式的通知功能。

触发事件种类很多，如：节点创建，节点删除，节点改变，子节点改变等。

总的来说可以概括 Watcher 为以下三个过程：客户端向服务端注册 Watcher、服务端事件发生触发 Watcher、客户端回调 Watcher 得到触发事件情况

4.1. Watch 机制特点

一次性触发

事件发生触发监听，一个 watcher event 就会被发送到设置监听的客户端，这种效果是一次性的，后续再次发生同样的事件，不会再次触发。

事件封装

ZooKeeper 使用 WatchedEvent 对象来封装服务端事件并传递。

WatchedEvent 包含了每一个事件的三个基本属性：

通知状态（keeperState），事件类型（EventType）和节点路径（path）

event 异步发送

watcher 的通知事件从服务端发送到客户端是异步的。

先注册再触发

Zookeeper 中的 watch 机制，必须客户端先去服务端注册监听，这样事件发送才会触发监听，通知给客户端。

4.2. 通知状态和事件类型

同一个事件类型在不同的通知状态中代表的含义有所不同，下表列举了常见的通知状态和事件类型。

KeeperState	EventType	触发条件	说明
	None (-1)	客户端与服务端成功建立连接	
SyncConnected (0)	NodeCreated (1)	Watcher监听的对应数据节点被创建	
	NodeDeleted (2)	Watcher监听的对应数据节点被删除	此时客户端和服务器处于连接状态
	NodeDataChanged (3)	Watcher监听的对应数据节点的数据内容发生变更	
	NodeChildChanged (4)	Wather监听的对应数据节点的子节点列表发生变更	
Disconnected (0)	None (-1)	客户端与ZooKeeper服务器断开连接	此时客户端和服务器处于断开连接状态
Expired (-112)	Node (-1)	会话超时	此时客户端会话失效，通常同时也会受到SessionExpiredException异常
AuthFailed (4)	None (-1)	通常有两种情况，1：使用错误的schema进行权限检查 2：SASL权限检查失败	通常同时也会收到AuthFailedException异常

其中连接状态事件(type=None，path=null)不需要客户端注册，客户端只要有需要直接处理就行了。

4.3. Shell 客户端设置 watcher

设置节点数据变动监听：

```
[zk: localhost:2181(CONNECTED) 12] get /aaa0000000001 watch
helloallen
```

通过另一个客户端更改节点数据：

```
[zk: localhost:2181(CONNECTED) 0] set /aaa0000000001 hello22
```

此时设置监听的节点收到通知：

```
[zk: localhost:2181(CONNECTED) 13]
WATCHER::

WatchedEvent state:SyncConnected type:NodeDataChanged path:/aaa0000000001
get /aaa0000000001 watch
hello22
```

5. ZooKeeper Java API

org.apache.zookeeper.Zookeeper

Zookeeper 是在 Java 中客户端主类，负责建立与 zookeeper 集群的会话，并提供方法进行操作。

org.apache.zookeeper.Watcher

Watcher 接口表示一个标准的事件处理器，其定义了事件通知相关的逻辑，包含 KeeperState 和 EventType 两个枚举类，分别代表了通知状态和事件类型，同时定义了事件的回调方法：process（WatchedEvent event）。

process 方法是 Watcher 接口中的一个回调方法，当 ZooKeeper 向客户端发送一个 Watcher 事件通知时，客户端就会对相应的 process 方法进行回调，从而实现对事件的处理。

5.1. 基本使用

建立 java maven 项目,引入 maven pom 坐标。

```
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.9</version>
</dependency>
```

```
public static void main(String[] args) throws Exception {
        // 初始化 ZooKeeper 实例(zk 地址、会话超时时间，与系统默认一致、watcher)
        ZooKeeper zk = new ZooKeeper("node-1:2181,node-2:2181", 30000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("事件类型为：" + event.getType());
                System.out.println("事件发生的路径：" + event.getPath());
                System.out.println("通知状态为：" +event.getState());
            }
        });
        zk.create("/myGirls", "性感的".getBytes("UTF-8"), Ids.OPEN_ACL_UNSAFE,
                    CreateMode.PERSISTENT);
        zk.close();
```

5.2. 更多操作示例

```
public static void main(String[] args) throws Exception {
        // 初始化 ZooKeeper 实例(zk 地址、会话超时时间，与系统默认一致、watcher)
        ZooKeeper zk = new ZooKeeper("node-21:2181,node-22:2181", 30000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("事件类型为：" + event.getType());
                System.out.println("事件发生的路径：" + event.getPath());
                System.out.println("通知状态为：" +event.getState());
            }
        });
    // 创建一个目录节点
   zk.create("/testRootPath", "testRootData".getBytes(), Ids.OPEN_ACL_UNSAFE,
   CreateMode.PERSISTENT);
 // 创建一个子目录节点
 zk.create("/testRootPath/testChildPathOne", "testChildDataOne".getBytes(),
   Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
 System.out.println(new String(zk.getData("/testRootPath",false,null)));
 // 取出子目录节点列表
 System.out.println(zk.getChildren("/testRootPath",true));
 // 修改子目录节点数据
 zk.setData("/testRootPath/testChildPathOne","modifyChildDataOne".getBytes(),-1);
 System.out.println("目录节点状态：["+zk.exists("/testRootPath",true)+"]");
 // 创建另外一个子目录节点
 zk.create("/testRootPath/testChildPathTwo", "testChildDataTwo".getBytes(),
   Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
 System.out.println(new String(zk.getData("/testRootPath/testChildPathTwo",true,null)));
 // 删除子目录节点
 zk.delete("/testRootPath/testChildPathTwo",-1);
 zk.delete("/testRootPath/testChildPathOne",-1);
 // 删除父目录节点
 zk.delete("/testRootPath",-1);
 zk.close();
}
```

6. ZooKeeper 选举机制

zookeeper 默认的算法是 FastLeaderElection，采用投票数大于半数则胜出的逻辑。

6.1. 概念

服务器 ID

比如有三台服务器，编号分别是 1,2,3。

编号越大在选择算法中的权重越大。

选举状态

LOOKING，竞选状态。

FOLLOWING，随从状态，同步 leader 状态，参与投票。

OBSERVING，观察状态,同步 leader 状态，不参与投票。

LEADING，领导者状态。

数据 ID

服务器中存放的最新数据 version。

值越大说明数据越新，在选举算法中数据越新权重越大。

逻辑时钟

也叫投票的次数，同一轮投票过程中的逻辑时钟值是相同的。每投完一次票这个数据就会增加，然后与接收到的其它服务器返回的投票信息中的数值相比，根据不同的值做出不同的判断。

6.2. 全新集群选举

假设目前有 5 台服务器，每台服务器均没有数据，它们的编号分别是 1,2,3,4,5,按编号依次启动，它们的选择举过程如下：

- 服务器 1 启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器 1 的状态一直属于 Looking。
- 服务器 2 启动，给自己投票，同时与之前启动的服务器 1 交换结果，由于服务器 2 的编号大所以服务器 2 胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是 LOOKING。
- 服务器 3 启动，给自己投票，同时与之前启动的服务器 1,2 交换信息，由于服务器 3 的编号最大所以服务器 3 胜出，此时投票数正好大于半数，所以服务器 3 成为领导者，服务器 1,2 成为小弟。
- 服务器 4 启动，给自己投票，同时与之前启动的服务器 1,2,3 交换信息，尽管服务器 4 的编号大，但之前服务器 3 已经胜出，所以服务器 4 只能成为小弟。
- 服务器 5 启动，后面的逻辑同服务器 4 成为小弟。

6.3. 非全新集群选举

对于运行正常的 zookeeper 集群，中途有机器 down 掉，需要重新选举时，选举过程就需要加入数据 ID、服务器 ID 和逻辑时钟。

数据 ID：数据新的 version 就大，数据每次更新都会更新 version。

服务器 ID：就是我们配置的 myid 中的值，每个机器一个。

逻辑时钟：这个值从 0 开始递增，每次选举对应一个值。 如果在同一次选举中，这个值是一致的。

这样选举的标准就变成：

1、逻辑时钟小的选举结果被忽略，重新投票；

2、统一逻辑时钟后，数据 id 大的胜出；

3、数据 id 相同的情况下，服务器 id 大的胜出；

根据这个规则选出 leader。

7. ZooKeeper 典型应用

7.1. 数据发布与订阅（配置中心）

发布与订阅模型，即所谓的配置中心，顾名思义就是发布者将数据发布到 ZK 节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。

应用在启动的时候会主动来获取一次配置，同时，在节点上注册一个 Watcher，这样一来，以后每次配置有更新的时候，都会实时通知到订阅的客户端，从来达到获取最新配置信息的目的。比如：

分布式搜索服务中，索引的元信息和服务器集群机器的节点状态存放在 ZK 的一些指定节点，供各个客户端订阅使用。

注意：适合数据量很小的场景，这样数据更新可能会比较快。

7.2. 命名服务(Naming Service)

在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。被命名的实体通常可以是集群中的机器，提供的服务地址，远程对象等等——这些我们都可以统称他们为名字（Name）。其中较为常见的就是一些分布式服务框架中的服务地址列表。通过调用 ZK 提供的创建节点的 API，能够很容易创建一个全局唯一的 path，这个 path 就可以作为一个名称。

阿里巴巴集团开源的分布式服务框架 Dubbo 中使用 ZooKeeper 来作为其命名服务，维护全局的服务地址列表。

7.3. 分布式锁

分布式锁，这个主要得益于 ZooKeeper 保证了数据的强一致性。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

所谓保持独占，就是所有试图来获取这个锁的客户端，最终只有一个可以成功获得这把锁。通常的做法是把 zk 上的一个 znode 看作是一把锁，通过 create znode 的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。

控制时序，就是所有试图来获取这个锁的客户端，最终都是会被安排执行，只是有个全局时序了。做法和上面基本类似，只是这里 /distribute_lock 已经预先存在，客户端在它下面创建临时有序节点（这个可以通过节点的属性控制：CreateMode.EPHEMERAL_SEQUENTIAL 来指定）。Zk 的父节点（/distribute_lock）维持一份 sequence,保证子节点创建的时序性，从而也形成了每个客户端的全局时序。

三、 网络编程

1. 概述

通过通信线路（有线或无线）可以把不同地理位置且相互独立的计算机连同其外部设备连接起来，组成计算机网络。在操作系统、网络管理软件及网络通信协议的管理和协调下，可以实现计算机之间的资源共享和信息的传递。

网络编程是指用来实现网络互联的不同计算机上运行的程序间可以进行数据交换。对我们来说即如何用编程语言 java 实现计算机网络中不同计算机之间的通信。

2. 网络通信三要素

IP 地址

网络中计算机的唯一标识；

32bit（4 字节），一般用“点分十进制”表示，如 192.168.1.158；

IP 地址=网络地址＋主机地址　可分类：

A 类：第 1 个 8 位表示网络地址。剩下的 3 个 8 位表示主机地址

B 类：前 2 个 8 位表示网络地址。剩下的 2 个 8 位表示主机地址

C 类：前 3 个 8 位表示网络地址。剩下的 1 个 8 位表示主机地址

D 类地址用于在 IP 网络中的组播

E 类地址保留作研究之用。

Java 编程中可使用 InetAddress 类来操纵 IP 地址。

```
public static void main(String[] args) throws UnknownHostException {
        InetAddress localHost = InetAddress.getLocalHost();
        System.out.println(localHost.getHostAddress());
        System.out.println(localHost.getHostName());
    }
```

端口号

用于标识进程的逻辑地址，不同进程的标识；

有效端口：0-65535，其中 0-1024 系统使用或保留端口。

传输协议

通讯的规则

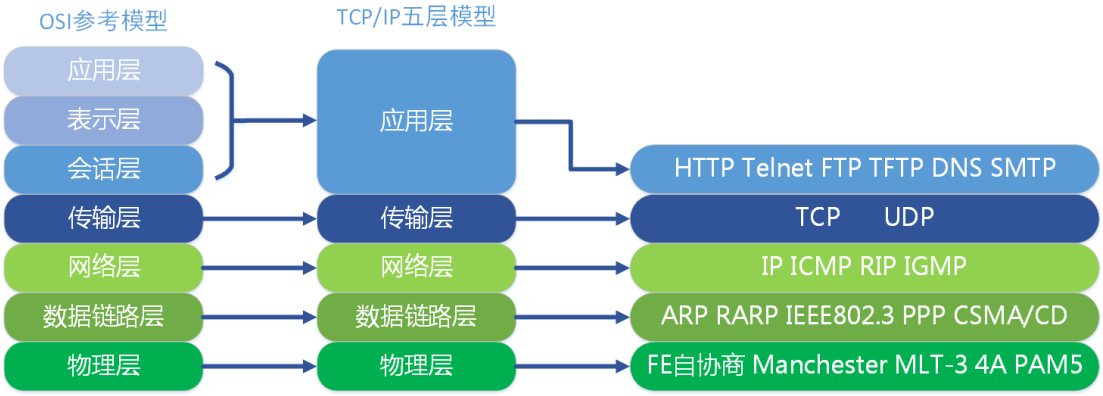
常见协议： UDP（用户数据报协议）、TCP（传输控制协议）

- UDP
  - 将数据及源和目的封装成数据包中，不需要建立连接
  - 每个数据报的大小在限制在64k内
  - 因无连接，是不可靠协议
  - 不需要建立连接，速度快
- TCP
  - 建立连接，形成传输数据的通道。
  - 在连接中进行大数据量传输
  - 通过三次握手完成连接，是可靠协议
  - 必须建立连接，效率会稍低

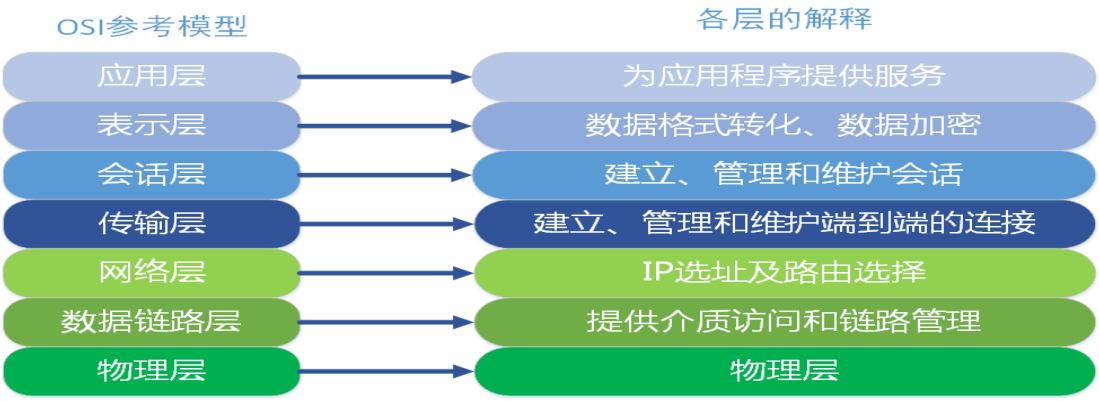
3. 网络模型

计算机网络之间以何种规则进行通信，就是网络模型所研究的问题。

网络模型一般是指 OSI 七层参考模型和 TCP/IP 五层参考模型。



每一层实现各自的功能和协议，并且都为上一层提供业务功能。为了提供这种业务功能，下一层将上一层中的数据并入到本层的数据域中，然后通过加入报头或报尾来实现该层业务功能，该过程叫做数据封装。用户的数据要经过一次次包装，最后转化成可以在网络上传输的信号，发送到网络上。当到达目标计算机后，再执行相反的数据拆包过程。



物理层：

主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。

主要作用是将数据最终编码为用 0、1 标识的比特流，通过物理介质传输。

这一层的数据叫做比特。

数据链路层：

主要将接收到的数据进行 MAC 地址（网卡地址）的封装与解封装。

常把这一层的数据叫做帧。这一层常工作的设备是交换机。

网络层：

主要将接收到的数据进行 IP 地址的封装与解封装。

常把这一层的数据叫做数据包。这一层设备是路由器。

传输层：

定义了一些数据传输的协议和端口号。

主要将接收的数据进行分段和传输，到达目的地址后在进行重组。

常把这一层的数据叫做段。

会话层：

通过传输层建立数据传输的通路。

主要在系统之间发起会话或者接收会话请求。

表示层：

主要进行对接收数据的解释、加密与解密、压缩与解压缩。

确保一个系统的应用层发送的数据能被另一个系统的应用层识别。

应用层：

主要是为一些终端应用程序提供服务。直接面对着用户的。

4. Socket 机制

4.1. Socket 概述

Socket，又称为套接字，用于描述 IP 地址和端口。应用程序通常通过 socket 向网络发出请求或者应答网络请求。Socket 就是为网络编程提供的一种机制：

通信两端都有 socket；

网络通信其实就是 socket 之间的通信；

数据在两个 socket 之间通过 IO 传输。

网络编程也称作为 Socket 编程，套接字编程。

Socket 通信是 Client/Server 模型。

4.2. 基于 UDP 协议的 Socket 通信

核心类：DatagramSocket

发送端：

```
// 创建发送端 Socket 服务对象
DatagramSocket dSocket = new DatagramSocket();

// 创建数据，打包数据
String message = "hello ,are u UDP ?";
byte[] bys = message.getBytes();
int length = bys.length;
InetAddress address = InetAddress.getByName("localhost");
int port = 12621;
DatagramPacket dPacket = new DatagramPacket(bys, length, address, port);

// 发送数据
dSocket.send(dPacket);

// 资源释放
dSocket.close();
```

接收端：

```
//创建接收端 Socket 服务对象
DatagramSocket dSocket = new DatagramSocket(12621);

//创建数据包（接收容器）
byte[] bys = new byte[1024];
DatagramPacket dPacket = new DatagramPacket(bys, bys.length);

//调用接收方法
dSocket.receive(dPacket);

//数据包解析
InetAddress address = dPacket.getAddress();
String hostAddress = address.getHostAddress();

byte[] data = dPacket.getData();
String message = new String(data);

System.out.println(hostAddress+"*********:"+message);

//资源释放
dSocket.close();
```

4.3. 基于 TCP 协议的 Socket 通信

服务端

核心 API：ServerSocket

流程：

创建 ServerSocket 服务，然后绑定在服务器的 IP 地址和端口

监听连接请求

接受请求，建立了 TCP 连接

获取输入流读取数据，并显示

释放资源

```
//建立服务端 socket 服务，并且监听一个端口
ServerSocket ss =  new ServerSocket(13131);

//监听连接
Socket s = ss.accept();

//获取输入流，读取数据
```

```
InputStream inputStream = s.getInputStream();
byte[] bys = new byte[1024];
int len = inputStream.read(bys);

System.out.println(new String(bys, 0, len));

//关闭客户端
s.close();

//关闭服务端，一般服务端不关闭
ss.close();
```

客户端

核心 API：Socket

流程： 创建客户端 socket 对象

向服务端请求建立 tcp 连接

从 tcp 连接中获取输出流，写数据

释放资源

```
//创建客户端的 socket 服务，指定目的主机和端口
Socket s = new Socket("127.0.0.1", 13131);

//通过 socket 获取输出流，写数据
OutputStream outputStream = s.getOutputStream();
outputStream.write("hello ,this is tcp?".getBytes());

//释放资源
s.close();
```

5. IO 通信模型

网络通信的本质是网络间的数据 IO。只要有 IO，就会有阻塞或非阻塞的问题，无论这个 IO 是网络的，还是硬盘的。原因在于程序是运行在系统之上的，任何形式的 IO 操作发起都需要系统的支持。

5.1. BIO（阻塞模式）

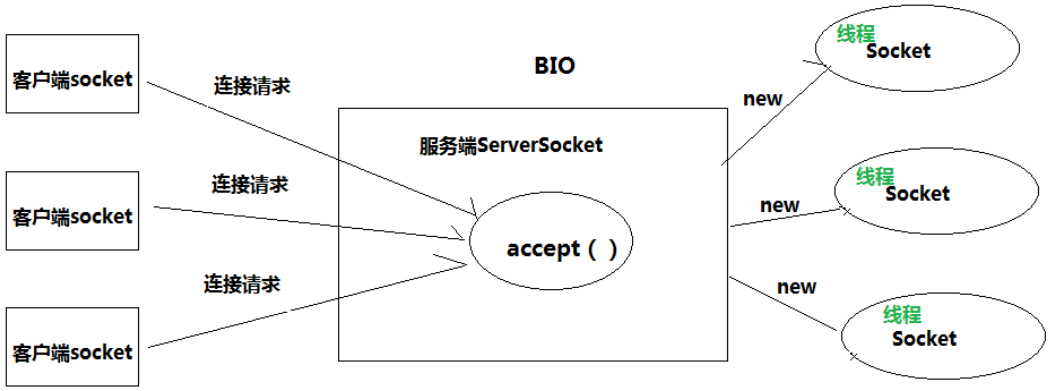
BIO 即 blocking IO，是一种阻塞式的 IO。

jdk1.4 版本之前 Socket 即 BIO 模式。

BIO 的问题在于 accept()、read()的操作点都是被阻塞的。

服务器线程发起一个 accept 动作，询问操作系统是否有新的 socket 信息从端口 X 发送过来。注意，是询问操作系统。如果操作系统没有发现有 socket 从指定的端口 X 来，那么操作系统就会等待。这样 serverSocket.accept()方法就会一直等待。这就是为什么 accept()方法为什么会阻塞。

如果想让 BIO 同时处理多个客户端请求，就必须使用多线程，即每次 accept 阻塞等待来自客户端请求，一旦收到连接请求就建立通信，同时开启一个新的线程来处理这个套接字的数据读写请求，然后立刻又继续 accept 等待其他客户端连接请求，即为每一个客户端连接请求都创建一个线程来单独处理。



5.2. NIO（非阻塞模式）

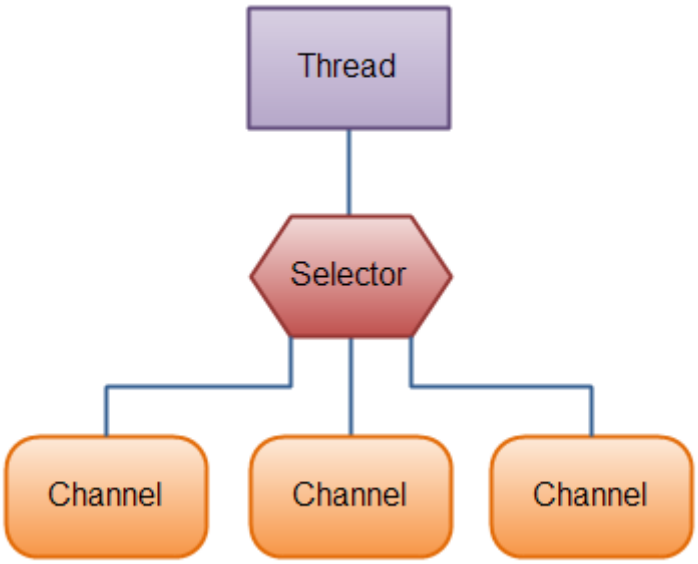
NIO 即 non-blocking IO，是一种非阻塞式的 IO。jdk1.4 之后提供。

NIO 三大核心部分：Channel(通道)，Buffer(缓冲区)，Selector(选择器)。

Buffer：容器对象，包含一些要写入或者读出的数据。在 NIO 库，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，也是写入到缓冲区中。任何时候访问 NIO 中的数据，都是通过缓冲区进行操作。

Channel：通道对象，对数据的读取和写入要通过 Channel，它就像水管一样。通道不同于流的地方就是通道是双向的，可以用于读、写和同时读写操作。Channel 不会直接处理字节数据，而是通过 Buffer 对象来处理数据。

Selector：多路复用器，选择器。提供选择已经就绪的任务的能力。Selector 会不断**轮询**注册在其上的 Channel，如果某个 Channel 上面发生读或者写事件，这个 Channel 就处于就绪状态，会被 Selector 轮询出来，进行后续的 I/O 操作。这样服务器只需要一两个线程就可以进行多客户端通信。



5.3. 阻塞/非阻塞、同步/非同步

阻塞 IO 和非阻塞 IO 这两个概念是程序级别的。主要描述的是程序请求操作系统 IO 操作后，如果 IO 资源没有准备好，那么程序该如何处理的问题：前者等待；后者继续执行（并且使用线程一直轮询，直到有 IO 资源准备好了）。

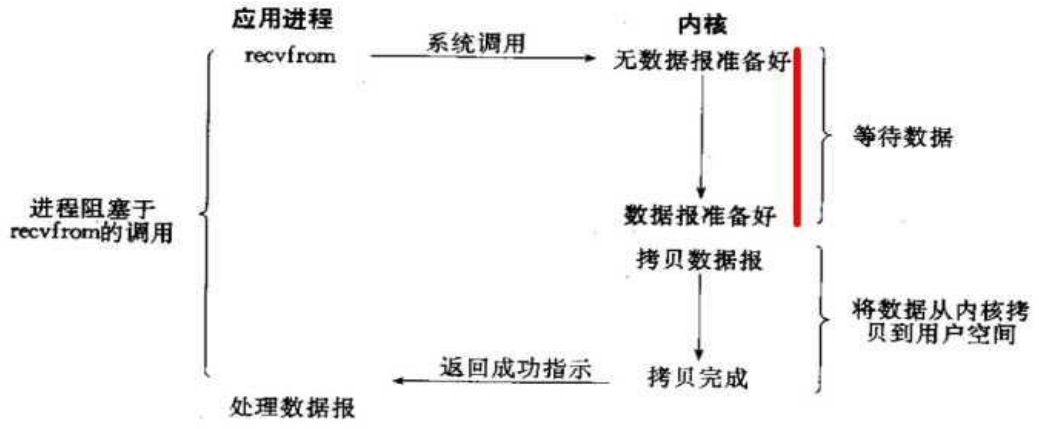


图 6.1 阻塞 I/O 模型

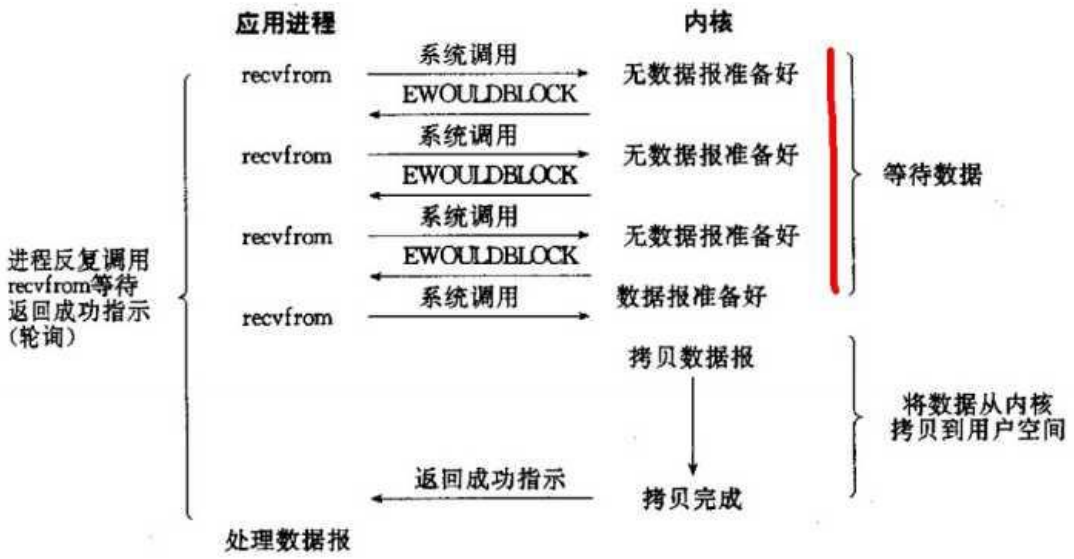


图 6.2 非阻塞 I/O 模型

同步 IO 和非同步 IO，这两个概念是操作系统级别的。主要描述的是操作系统在收到程序请求 IO 操作后，如果 IO 资源没有准备好，该如何响应程序的问题：前者不响应，直到 IO 资源准备好以后；后者返回一个标记（好让程序和自己知道以后的数据往哪里通知），当 IO 资源准备好以后，再用事件机制返回给程序。

权威参考：《UNIX 网络编程：卷一》第六章——I/O 复用

6. RPC

6.1. 什么是 RPC

RPC（Remote Procedure Call Protocol）远程过程调用协议。

通俗的描述是：客户端在不知道调用细节的情况下，调用存在于远程计算机上的某个过程或函数，就像调用本地应用程序中的一样。

正式的描述是：一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

6.2. RPC 主要特质

RPC 是协议：协议意味着规范。目前典型的 RPC 实现包括：Dubbo、Thrift、Hetty 等。但这些实现往往都会附加其他重要功能，例如 Dubbo 还包括了服务管理、访问权限管理等功能。

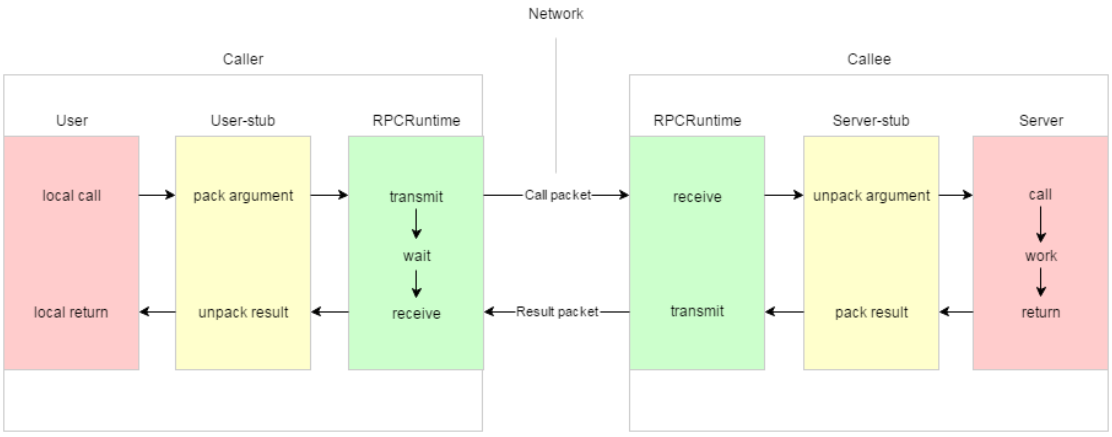
网络协议和网络 IO 模型对其透明：既然 RPC 的客户端认为自己是在调用本地对象。那么传输层使用的是 TCP/UDP 还是 HTTP 协议，又或者是一些其他的网络协议它就不需要关心了。既然网络协议对其透明，那么调用过程中，使用的是哪一种网络 IO 模型调用者也不需要关心。

信息格式对其透明：远程调用过程中，需要传递一些参数，并且会返回一个调用结果。至于这些参数会以某种信息格式传递给网络上的另外一台计算机，这个信息格式是怎样构成的，调用方是不需要关心的。

跨语言能力：对于调用方来说，不知道也无需知道远程的程序使用的是什么语言运行的，无论服务器方使用的是什么语言，本次调用都应该成功，并且返回值也应该按照调用方程序语言所能理解的形式进行描述。

6.3. RPC 原理

实现 RPC 的程序包括 5 个部分：User、User-stub、RPCRuntime、Server-stub、Server。



user 就是发起 RPC 调用的 client ，当 user 想发起一个远程调用时，它实际是通过本地调用 user-stub。 user-stub 负责将调用的接口、方法和参数通过约定的协议规范进行编码并通过本地的 RPCRuntime 实例传输到远端的实例。远端 RPCRuntim 实例收到请求后交给 server-stub 进行解码后发起本地端调用，调用结果再返回给 user 端。

stub：为屏蔽客户调用远程主机上的对象，必须提供某种方式来模拟本地对象，这种本地对象称为存根(stub)，存根负责接收本地方法调用，并将它们委派给各自的具体实现对象。