手撸rpc远程调用框架

代码gitee地址：https://gitee.com/zhangnan716/ydlclass-yrpc

我们先更新前两天的课程，文档会继续升级大家先看这个。

第一章 概述

一、为什么要手写一个rpc项目

只要你学习过dubbo，学习过微服务，你一定会对rpc这个概念很熟悉。最近几年，不管是在实际的工作中，还是面试中，我们经常可以看到rpc的影子，现在，他几乎成了我们工作中必不可少的一个基础组件，值得我们学习。

有些朋友可能会提出疑问，我们已经有了很成熟的rpc框架，比如dubbo、grpc那么我们为什么还要自己去手写呢?

这个问题我们要这样来看：

1、学习本身就是为了增长知识，只有懂其原理，才能更好的利用框架，才能更好的解决问题。

2、大厂都有自己的中间件部门，很多组件必须要自行研发，一方面是自研组件可以更好的满足自己的业务需要，一方面是防止开源作者停更无法维护、甚至卡脖子。

很重要的一点：如果我们可以在简历中写出一个**很完善的rpc项目**，并且能在**面试中讲出其中的核心知识点**，那对于校招来说，简直无敌。

二、什么是rpc

rpc 的全称是 Remote Procedure Call，即**远程过程调用。**从字面上的来看，rpc就是通过网络通信访问另一台机器的应用程序接口。但随着近几年的技术在不断发展，rpc也有了一些新的含义。

目前，我们的rpc组件的基本能力就是**屏蔽网络编程细节，实现调用远程方法就跟调用本地（同一个项目中的方法）一样。**事实上一个合格的可用于生产的rpc框架还应该具备**负载均衡、优雅启停、链路追踪、灰度发布**等等功能。这次的课程我们会择优选择其中的一部分功能进行讲解。

三、rpc怎么使用

目前我们对rpc有了一个基本的了解，那 rpc 在项目中应该如何使用呢，如果有同学熟悉dubbo、grpc、或者openFegin那一定很清楚这个问题的答案。

我们就以dubbo为例，大家可以参考dubbo的官网讲解，在springboot中，服务提供者和消费者只要依赖相同接口，就可以使用如下的方式进行远程调用了。

1、服务端只需要使用 @DubboService 注解将对应接口的实现暴露出去。

```
@DubboService
public class DemoServiceImpl implements DemoService {

    @Override
    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

2、客户端就可以直接使用 @DubboReference 直接注入使用了。

```
@Component
public class Task implements CommandLineRunner {
    @DubboReference
    private DemoService demoService;

    @Override
    public void run(String... args) throws Exception {
        String result = demoService.sayHello("world");
        System.out.println("Receive result ======> " + result);

        new Thread(()-> {
            while (true) {
                try {
                    Thread.sleep(1000);
                    System.out.println(new Date() + " Receive result ======> " +
demoService.sayHello("world"));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    Thread.currentThread().interrupt();
                }
            }
        }).start();
    }
}
```

这样就是实现了调用远程接口就和调用本地接口一样的能力。事实上，我们使用 MQ 来处理异步流程、Redis 处理缓存热点数据、MySQL 进行持久化数据，调用其他业务系统接口，都可以说是属于 yrpc 调用，由此可见，rpc 确实是我们日常开发中经常接触的东西，只是被包装成了各种框架，导致我们很少意识到这就是 rpc，让 rpc 变成了我们最“熟悉的陌生人”。

rpc 是整个应用系统的“经络”，这不为过吧？我们真的很有必要学好 rpc，不仅因为 rpc 是构建复杂系统的基石，还是提升自身认知的利器。

四、rpc的通信流程

rpc能实现**调用远程方法就跟调用本地（同一个项目中的方法）一样，**发起调用请求的那一方叫做**调用方**，被调用的一方叫做**服务提供方。**

发起远程调用的核心是网络通信，那我们手动实现rpc框架的核心就是**封装通信细节**，本小节我们主要和大家一起来思考整个调用过程中的一些流程和细节：

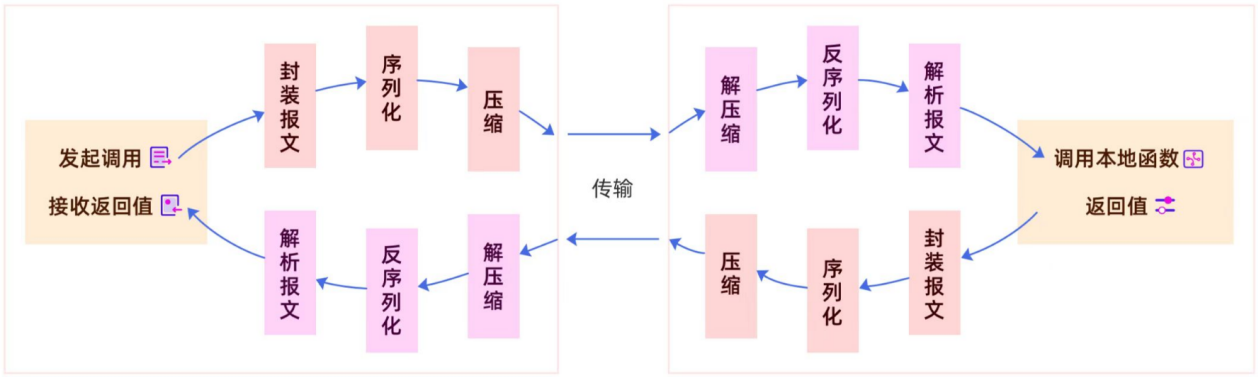
1、传输协议：既然 yrpc 存在的核心目的是为了实现远程调用，既然是远程调用那肯定就需要通过网络来传输数据，并且 yrpc 常用于业务系统之间的数据交互，需要保证其可靠性，所以 yrpc 一般默认采用 TCP 来传输。事实上。我们常用的 HTTP 协议也是建立在 TCP 之上的。选择tcp的核心原因还是因为他的效率要比很多应用层协议高很多。

2、封装一个可用的协议：选择了合适的传输层协议之后，我们需要基于此建立一个我们自己的通用协议，和http一样需要封装自己的应用层协议，详细的内容会在后边的课程里详细介绍。

3、序列化：网络传输的数据必须是二进制数据，但调用方请求的出入参数都是对象。对象是肯定没法直接在网络中传输的，需要提前把它转成可传输的二进制，并且要求转换算法是可逆的，这个过程我们一般叫做“序列化”。

4、压缩：如果我们觉得序列化后的字节数组体积比较大，我们还可以对他进行压缩，压缩后的字节数组体积更小，能在传输的过程中更加节省带宽和内存。

到这里，一个简单版本的 yrpc 框架就实现了。我把整个流程都画出来了，供你参考：



那上述几个流程就组成了一个完整的 rpc 吗?

在我看来，还缺点东西。因为对于研发人员来说，这样做要掌握太多的 rpc 底层细节，需要手动写代码去构造请求、调用序列化，并进行网络调用，整个 API 非常不友好。

那我们有什么办法来简化 API，屏蔽掉 rpc 细节，让使用方**只需要关注业务接口**，像调用本地一样来调用远程呢?

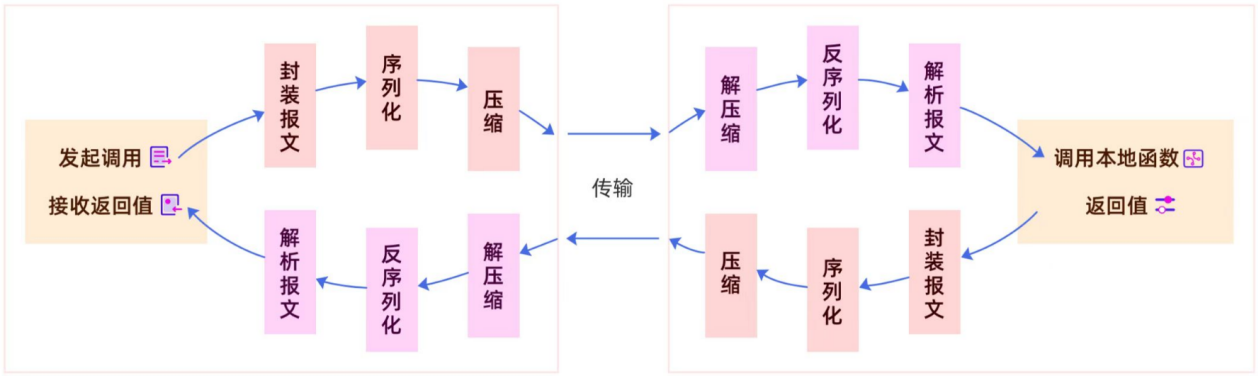
如果你了解 Spring，一定对其 AOP 技术很佩服，其核心是采用动态代理的技术，通过字节码增强对方法进行拦截增强，以便于增加需要的额外处理逻辑。其实这个技术也可以应用到 rpc 场景来解决我们刚才面临的问题。

由服务提供者给出业务接口声明，在调用方的程序里面，rpc 框架根据调用的服务接口提前生成动态代理实现类，并通过依赖注入等技术注入到声明了该接口的相关业务逻辑里面。该代理实现类会拦截所有的方法调用，在提供的方法处理逻辑里面完成一整套的远程调用，并把远程调用结果返回给调用方，这样调用方在调用远程方法的时候就获得了像调用本地接口一样的体验。

了解了我们的一些基础概念和知识，我们开始设计并编写我们自己的rpc框架，我们把他**命名为yrpc**，y就是元动力的元。

第二章 网络传输

实现远程调用，网络传输是基石，本章我们从网络传输的技术选型上给大家讲解一下，还是这张图：



客户端发起调用过程　　服务端接收请求返回结果

我们可以使用java中的`socket api`来实现我们的yrpc远程调用框架，但毋庸置疑`netty`**是最好的选择。**他提供了非常友好的api供我们使用，同时还完美的提供了**IO 多路复用以及零拷贝**的实现，作为一个基础框架，性能、扩展性和易用性是最重要的几个方面之一。本课程的主要内容是讲解yrpc，而非netty，所以需要大家先自行学习一下netty相关的知识，这里我们只对面试中常问的**io多路复用和零拷贝**给大家做一个详细介绍：

一、零拷贝

1、零拷贝的概念

学习零拷贝时我们先了解几个buffer缓冲区：

- 当某个程序或已存在的进程需要某段数据时，它只能在用户空间中属于它自己的内存中访问、修改，这段内存暂且称之为`user buffer`
- 正常情况下，数据只能从磁盘(或其他外部设备)加载到内核的缓冲区，且称之为`kernel buffer`
- TCP/IP协议栈维护着两个缓冲区：`send buffer`和`recv buffer`，它们合称为`socket buffer`

（1）DMA操作

DMA 的全称叫直接内存存取（Direct Memory Access），是一种允许外围设备（硬件子系统）直接访问系统主内存的机制。

DMA下读取磁盘数据流程如下:·

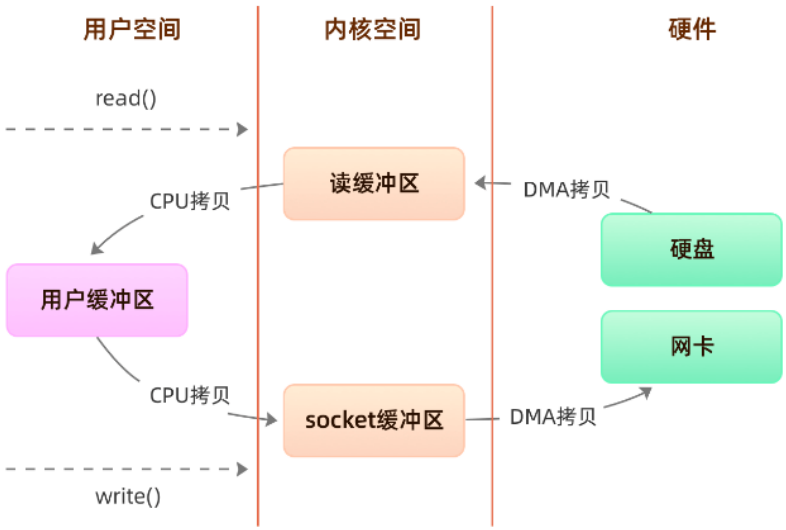
1. 用户进程向 CPU 发起 read 系统调用读取数据，由用户态切换为内核态，然后一直阻塞等待数据的返回。
2. CPU 在接收到指令以后对 DMA 磁盘控制器发起调度指令。
3. DMA 磁盘控制器对磁盘发起 I/O 请求，将磁盘数据先放入**磁盘控制器缓冲区**，CPU 全程不参与此过程。

4. 数据读取完成后，DMA 磁盘控制器会接受到磁盘的通知，**将数据从磁盘控制器缓冲区拷贝到内核缓冲区。**
5. **DMA 磁盘控制器向 CPU 发出数据读完的信号**，由 **CPU 负责将数据从内核缓冲区拷贝到用户缓冲区。**
6. 用户进程**由内核态切换回用户态，解除阻塞状态。**

整个数据传输操作是在一个 DMA 控制器的控制下进行的。CPU 除了在数据传输开始和结束时做一点处理外（开始和结束时候要做中断处理），在传输过程中 CPU 可以继续进行其他的工作。这样在大部分时间里，CPU 计算和 I/O 操作都处于并行操作，使整个计算机系统的效率大大提高。

（2）传统读取数据和发送数据

程序传统IO实际上是调用系统的`read()`和`write()`实现，通过`read()`把数据从**硬盘读取到内核缓冲区**，再复制到用户缓冲区；然后再通过`write()`写入到**socket缓冲区，最后写入网卡设备**：



整个过程发生了**四次用户态和内核态的切换还有四次IO拷贝**， 具体流程是：

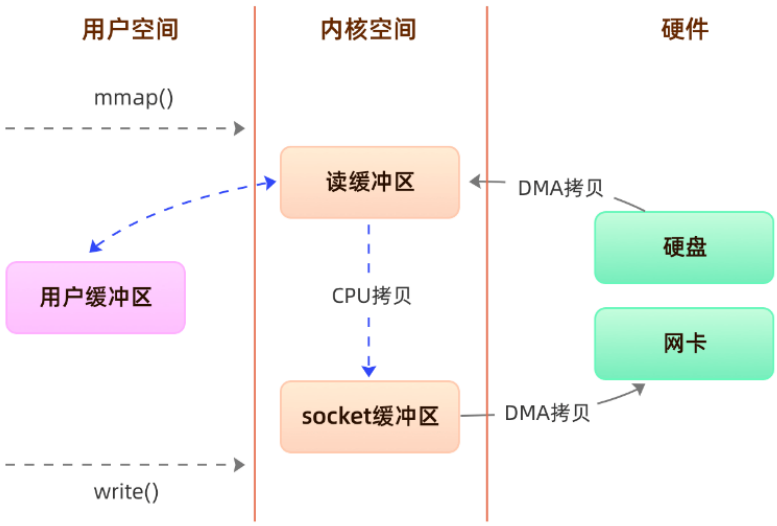
1. 用户进程通过`read()`方法向操作系统发起调用，此时上下文从用户态转向内核态
2. DMA控制器把数据从硬盘中拷贝到读缓冲区
3. CPU把读缓冲区数据拷贝到应用缓冲区，上下文从内核态转为用户态，`read()`返回
4. 用户进程通过`write()`方法发起调用，上下文从用户态转为内核态
5. CPU将应用缓冲区中数据拷贝到socket缓冲区
6. DMA控制器把数据从socket缓冲区拷贝到网卡，上下文从内核态切换回用户态，`write()`返回

（3）零拷贝实现方式

方案一、内存映射(mmap+write)

mmap 是 Linux 提供的一种内存映射文件方法，即将一个**进程的地址空间中的一段虚拟地址映射到磁盘文件地址。**

mmap 主要实现方式是将**读缓冲区的地址和用户缓冲区的地址进行映射，内核缓冲区和应用缓冲区共享，**从而减少了从读缓冲区到用户缓冲区的一次CPU拷贝，然而内核读缓冲区（read buffer）仍需将数据拷贝到内核写缓冲区（socket buffer）。



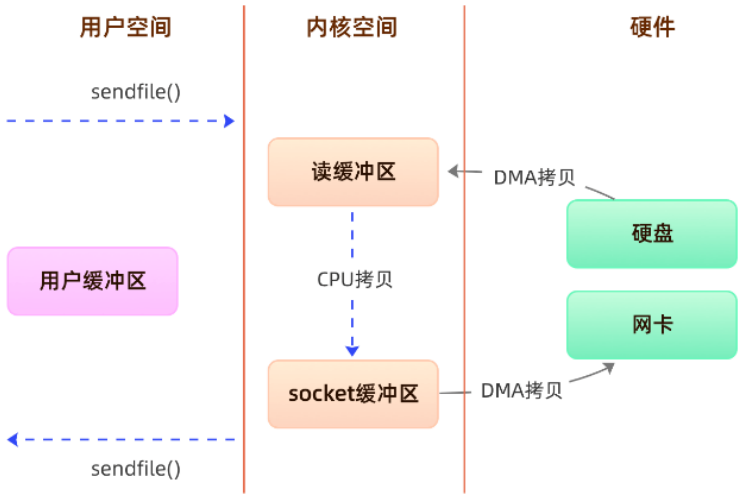
基于 mmap + write 系统调用的零拷贝方式，整个过程发生了4次用户态和内核态的上下文切换和3次拷贝，具体流程如下：

1. 用户进程通过mmap()方法向操作系统发起调用，上下文从用户态转向内核态
2. DMA控制器把数据从硬盘中拷贝到读缓冲区
3. 上下文从内核态转为用户态，mmap调用返回
4. 用户进程通过write()方法发起调用，上下文从用户态转为内核态
5. CPU将读缓冲区中数据拷贝到socket缓冲区
6. DMA控制器把数据从socket缓冲区拷贝到网卡，上下文从内核态切换回用户态，write()返回

mmap 主要的用处是提高 I/O 性能，特别是针对大文件。对于小文件，内存映射文件反而会导致碎片空间的浪费，因为内存映射总是要对齐页边界，最小单位是 4 KB，一个 5 KB 的文件将会映射占用 8 KB 内存，也就会浪费 3 KB 内存。

方案二、sendfile

通过使用 `sendfile` 函数，数据可以直接在内核空间进行传输，因此避免了用户空间和内核空间的拷贝，同时由于使用sendfile替代了read+write从而节省了一次系统调用，也就是2次上下文切换。



整个过程发生了2次用户态和内核态的上下文切换和3次拷贝，具体流程如下：

1. 用户进程通过sendfile()方法向操作系统发起调用，上下文从用户态转向内核态

2. DMA控制器把数据从硬盘中拷贝到读缓冲区
3. CPU将读缓冲区中数据拷贝到socket缓冲区
4. DMA控制器把数据从socket缓冲区拷贝到网卡，上下文从内核态切换回用户态，sendfile调用返回

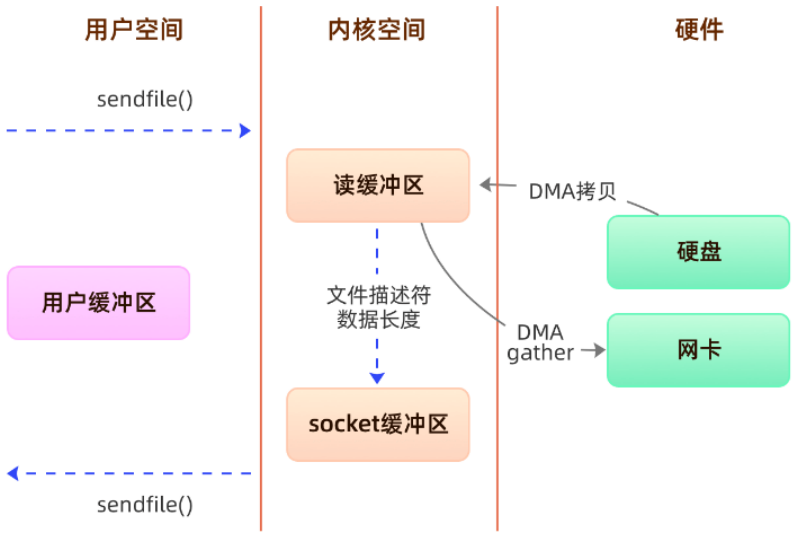
sendfile方法IO数据对用户空间完全不可见，所以**只能适用于完全不需要用户空间处理的情况**，比如静态文件服务器。

sendfile 只适用于把数据从磁盘中读出来往 socket buffer 发送的场景

方案三、sendfile+DMA scatter/gather

Linux2.4内核版本之后对sendfile做了进一步优化，通过引入新的硬件支持，这个方式叫做DMA Scatter/Gather 分散/收集功能。

它将读缓冲区中的数据描述信息–内存地址和偏移量记录到socket缓冲区，由 DMA 根据这些将数据从读缓冲区拷贝到网卡，相比之前版本减少了一次CPU拷贝的过程。



整个过程发生了2次用户态和内核态的上下文切换和2次拷贝，其中更重要的是完全没有CPU拷贝，具体流程如下：

1. 用户进程通过sendfile()方法向操作系统发起调用，上下文从用户态转向内核态
2. DMA控制器利用scatter把数据从硬盘中拷贝到读缓冲区离散存储
3. CPU把读缓冲区中的文件描述符和数据长度发送到socket缓冲区
4. DMA控制器根据文件描述符和数据长度，使用scatter/gather把数据从内核缓冲区拷贝到网卡
5. sendfile()调用返回，上下文从内核态切换回用户态

DMA gather和sendfile一样数据对用户空间不可见，而且需要硬件支持，同时输入文件描述符只能是文件，但是过程中完全没有CPU拷贝过程，极大提升了性能。

总结：

- 由于CPU和IO速度的差异问题，产生了DMA技术，通过DMA搬运来减少CPU的等待时间。
- 传统的 `IO read/write` 方式会产生2次DMA拷贝+2次CPU拷贝，同时有4次上下文切换。
- 而通过 `mmap+write` 方式则产生2次DMA拷贝+1次CPU拷贝，4次上下文切换，通过内存映射减少了一次CPU拷贝，可以减少内存使用，适合大文件的传输。

- `sendfile`方式是新增的一个系统调用函数，产生2次DMA拷贝+1次CPU拷贝，但是只有2次上下文切换。因为只有一次调用，减少了上下文的切换，但是用户空间对IO数据不可见，适用于静态文件服务器。
- `sendfile+DMA gather`方式产生2次DMA拷贝，没有CPU拷贝，而且也只有2次上下文切换。虽然极大地提升了性能，但是需要依赖新的硬件设备支持。

2、Netty中的零拷贝

操作系统层面的零拷贝主要避免在`用户态(User-space)`和`内核态(Kernel-space)`之间来回拷贝数据。

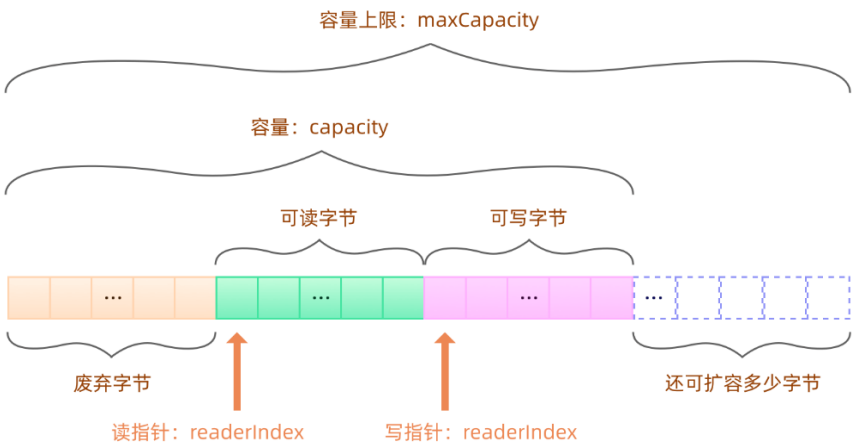
Netty中的`zero-copy`不同于操作系统，它完全是在用户态(java 层面)，更多的偏向于优化数据操作这样的概念，体现在：

- Netty 提供了`CompositeByteBuf`类, 它可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf, 避免了各个 ByteBuf 之间的拷贝
- 通过 wrap 操作, 我们可以将 byte[] 数组、ByteBuf、ByteBuffer等包装成一个 Netty ByteBuf 对象, 进而避免了拷贝操作
- ByteBuf 支持 slice 操作, 因此可以将 ByteBuf 分解为多个共享同一个存储区域的 ByteBuf, 避免了内存的拷贝
- 通过`FileRegion`包装的`FileChannel.transferTo`实现文件传输, 可以直接将文件缓冲区的数据发送到目标 Channel, 避免了传统通过循环 write 方式导致的内存拷贝问题

上述的 Netty 包装了`FileChannel.transferTo`实际上也是对操作系统`sendfile`的一个封装， 我们可以理解为 Netty **即支持了系统层面的零拷贝， 还有一个重要作用就是：防止在 JVM 中进行不必要的复制**

（1）ByteBuf

ByteBuf是Netty进行数据读写交互的单位，结构如下：



1. ByteBuf 是一个字节容器，容器里面的的数据分为三个部分，第一个部分是已经丢弃的字节，这部分数据是无效的；第二部分是可读字节，这部分数据是 ByteBuf 的主体数据， 从 ByteBuf 里面读取的数据都来自这一部分;最后一部分的数据是可写字节，所有写到 ByteBuf 的数据都会写到这一段。最后一部分虚线表示的是该 ByteBuf 最多还能扩容多少容量
2. 以上三段内容是被两个指针给划分出来的，从左到右，依次是读指针（readerIndex）、写指针（writerIndex），然后还有一个变量 capacity，表示 ByteBuf 底层内存的总容量
3. 从 ByteBuf 中每读取一个字节，readerIndex 自增1，ByteBuf 里面总共有`writerIndex-readerIndex`个字节可读,当 readerIndex 与 writerIndex 相等的时候，ByteBuf 不可读

4. 写数据是从 writerIndex 指向的部分开始写，每写一个字节，writerIndex 自增1，直到增到 capacity，这个时候，表示 ByteBuf 已经不可写了
5. ByteBuf 里面其实还有一个参数 maxCapacity，当向 ByteBuf 写数据的时候，如果容量不足，那么这个时候可以进行扩容，直到 capacity 扩容到 maxCapacity，超过 maxCapacity 就会报错

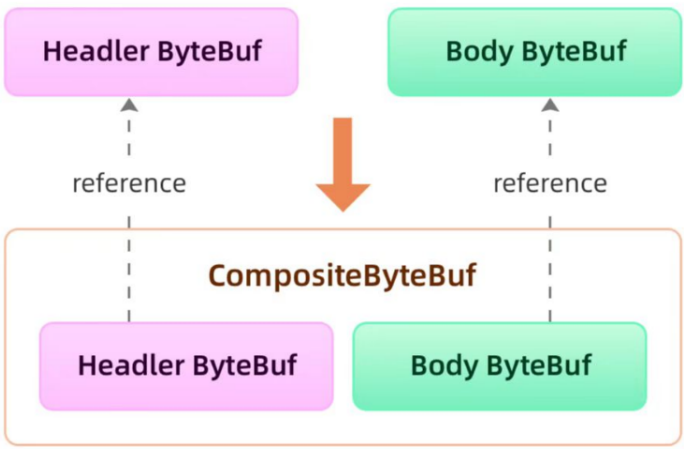
(2) CompositeByteBuf 零拷贝

`Composite buffer` 实现了透明的零拷贝，将物理上的多个 Buffer 组合成了一个逻辑上完整的 CompositeByteBuf.

比如在网络编程中, 一个完整的 http 请求常常会被分散到多个 Buffer 中。用 CompositeByteBuf 很容易将多个分散的Buffer组装到一起，而无需额外的复制：

```
ByteBuf header = Unpooled.buffer();// 模拟http请求头
ByteBuf body = Unpooled.buffer();// 模拟http请求主体
CompositeByteBuf httpBuf = Unpooled.compositeBuffer();
// 这一步，不需要进行header和body的额外复制，httpBuf只是持有了header和body的引用
// 接下来就可以正常操作完整httpBuf了
httpBuf.addComponents(header, body);
```

复制



而 JDK ByteBuffer 完成这一需求:

```
ByteBuffer header = ByteBuffer.allocate(1024);// 模拟http请求头
ByteBuffer body = ByteBuffer.allocate(1024);// 模拟http请求主体

// 需要创建一个新的ByteBuffer来存放合并后的buffer信息，这涉及到复制操作
ByteBuffer httpBuffer = ByteBuffer.allocate(header.remaining() + body.remaining());
// 将header和body放入新创建的Buffer中
httpBuffer.put(header);
httpBuffer.put(body);
httpBuffer.flip();
```

相比于JDK，Netty的实现更合理，省去了不必要的内存复制，可以称得上是JVM层面的零拷贝。

（3）wrap 操作实现零拷贝

例如我们有一个 byte 数组, 我们希望将它转换为一个 ByteBuf 对象, 以便于后续的操作, 那么传统的做法是将此 byte 数组拷贝到 ByteBuf 中, 即:

```
byte[] bytes = ...
ByteBuf byteBuf = Unpooled.buffer();
byteBuf.writeBytes(bytes);
```

这样的操作是有一次额外的拷贝，如果使用`Unpooled`相关的方法，包装这个byte数组生成一个新的的ByteBuf，而不需要进行拷贝，如:

```
byte[] bytes = ...
ByteBuf byteBuf = Unpooled.wrappedBuffer(bytes);
```

`Unpooled.wrappedBuffer` 方法来将 bytes 包装成为一个 `UnpooledHeapByteBuf` 对象, 而在包装的过程中, 是不会有拷贝操作的. 即最后我们生成的**生成的 ByteBuf 对象是和 bytes 数组共用了同一个存储空间**, 对 bytes 的修改也会反映到 ByteBuf 对象中

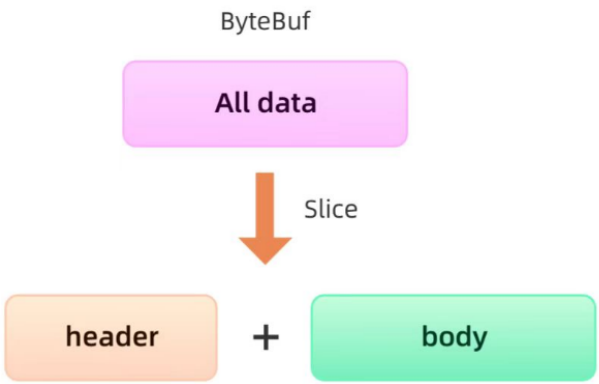
Unpooled 提供的方法可以将一个或多个 buffer 包装为一个 ByteBuf 对象, 从而避免了拷贝操作.

（4）通过 slice 操作实现零拷贝

slice 操作和 wrap 操作刚好相反, `Unpooled.wrappedBuffer` 可以将多个 ByteBuf 合并为一个 而 slice 操作将一个 ByteBuf 切片为多个共享一个存储区域的 ByteBuf 对象,如:

```
ByteBuf byteBuf = ...
ByteBuf header = byteBuf.slice(0, 5);
ByteBuf body = byteBuf.slice(5, 10);
```

用 slice 方法产生 byteBuf 的过程是没有拷贝操作的, header 和 body 对象在内部其实是共享了 byteBuf 存储空间的不同部分而已。



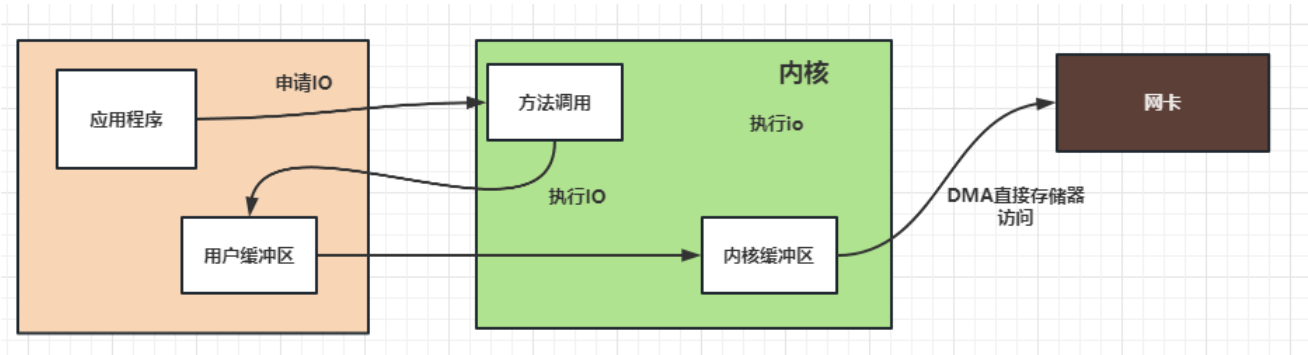
二、IO 多路复用

常见的网络 IO 模型分为四种：

- 同步阻塞 IO（BIO）、
- 同步非阻塞 IO（NIO）、
- IO 多路复用
- 异步非阻塞 IO（AIO）。

在这四种 IO 模型中，只有 AIO 为异步 IO，其他都是同步 IO。

下图是应用程序发起一次网络IO的流程：



第一步：应用程序发起IO申请（阻塞和非阻塞）

第二部：内核执行方法调用（同步和异步）

多路复用 IO 是在高并发场景中使用最为广泛的一种 IO 模型，如 Java 的 NIO、Redis、Nginx 的底层实现就是此类 IO 模型的应用，经典的 Reactor 模式也是基于此类 IO 模型。

那么什么是 IO 多路复用呢？通过字面上的理解，多路就是指多个通道，也就是多个网络连接的 IO，而复用就是指多个通道复用在一个selector上。

多个网络连接的 IO 可以注册到一个selector上，当用户进程调用了 select，那么整个进程会被阻塞。同时，内核会“监视”所有 selector 负责的 socket，当任何一个 socket 中的数据准备好了，select 就会返回。这个时候用户进程再调用 read 操作，将数据从内核中拷贝到用户进程。

这里我们可以看到，当用户进程发起了 select 调用，进程会被阻塞，当发现该 select 负责的 socket 有准备好的数据时才返回，之后才发起一次 read，整个流程要比阻塞 IO 要复杂，似乎也更浪费性能。但它最大的优势在于，用户可以在一个线程内同时处理多个 socket 的 IO 请求。用户可以注册多个 socket，然后不断地调用 select 读取被激活的 socket，即可达到在同一个线程内同时处理多个 IO 请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

同样好比我们去餐厅吃饭，这次我们是几个人一起去的，我们专门留了一个人在餐厅排号等位，其他人就去逛街了，等排号的朋友通知我们可以吃饭了，我们就直接去享用了。

yrpc 调用在大多数的情况下，是一个高并发调用的场景，考虑到系统内核的支持、编程语言的支持以及 IO 模型本身的特点，在 yrpc 框架的实现中，在网络通信的处理上，我们会选择 IO 多路复用的方式。开发语言的网络通信框架的选型上，我们最优的选择是基于 Reactor 模式实现的框架，如 Java 语言，首选的框架便是 Netty 框架（Java 还有很多其他 NIO 框架，但目前 Netty 应用得最为广泛），并且在 Linux 环境下，也要开启 epoll 来提升系统性能（Windows 环境下是无法开启 epoll 的，因为系统内核不支持）。

了解完以上内容，我们可以继续看这样一个关键问题——零拷贝。在我们应用的过程中，他是非常重要的。

三、netty入门

1、为什么要学习netty?

一方面：现在物联网的应用无处不在，大量的项目都牵涉到应用传感器和服务器端的数据通信，Netty作为基础通信组件、能够轻松解决之前有较高门槛的通信系统开发，你不用再为如何解析各类简单、或复杂的通讯协议而薅头发了，有过这方面开发经验的程序员会有更深刻、或者说刻骨铭心的体会。

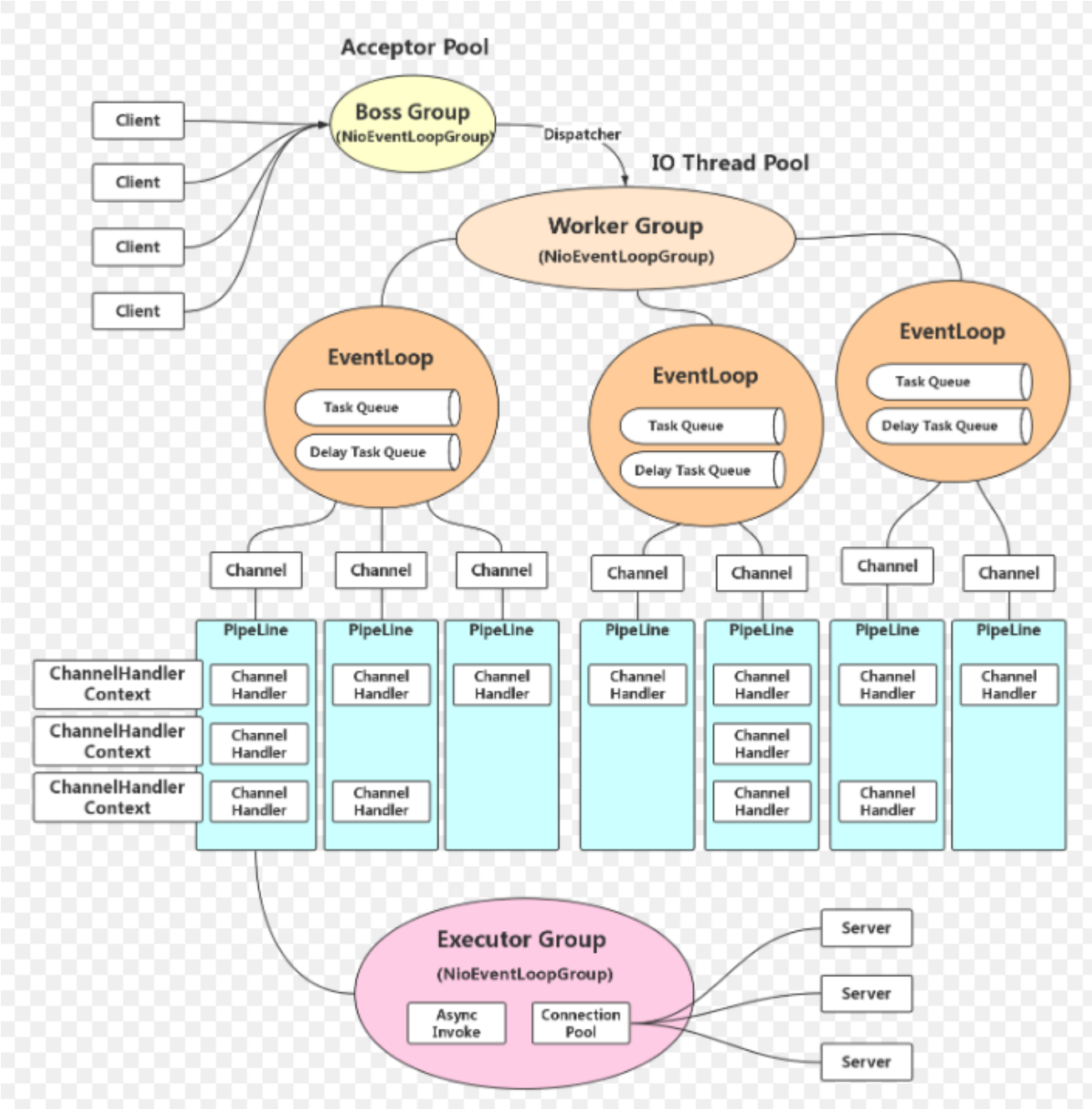
另一方面：现在互联网系统讲究的都是高并发、分布式、微服务，各类消息满天飞（是的，IM系统、消息推送系统就是其中的典型），Netty在这类架构里面的应用可谓是如鱼得水，如果你对当前的各种应用服务器不爽，那么完全可以基于Netty来实现自己的HTTP服务器、FTP服务器、UDP服务器、RPC服务器、WebSocket服务器、Redis的Proxy服务器、MySQL的Proxy服务器等等。

2、netty的基本工作流程

在netty中存在以下的核心组件：

- ServerBootstrap：服务器端启动辅助对象；
- Bootstrap：客户端启动辅助对象；
- Channel：通道，代表一个连接，每个Client请对会对应到具体的一个Channel；
- ChannelPipeline：责任链，每个Channel都有且仅有一个ChannelPipeline与之对应，里面是各种各样的Handler；
- handler：用于处理出入站消息及相应的事件，实现我们自己要的业务逻辑；
- EventLoopGroup：I/O线程池，负责处理Channel对应的I/O事件；
- ChannelInitializer：Channel初始化器；
- ChannelFuture：代表I/O操作的执行结果，通过事件机制，获取执行结果，通过添加监听器，执行我们想要的操作；
- ByteBuf：字节序列，通过ByteBuf操作基础的字节数组和缓冲区。

我们结合其核心组件通过下图，可以清晰的看明白netty的基本工作原理：



在这其中，ChannelPipeline 是一个重要的组件，用于处理 I/O 事件和拦截 I/O 操作。它是一个处理器链，负责将 I/O 操作分发给各个 ChannelHandler 进行处理。通过组合不同的 ChannelHandler，用户可以定制处理网络事件的逻辑，其中大多数的ChannelHandler需要我们手动编写。

一个典型的 Netty ChannelPipeline 可以包含以下几种 ChannelHandler：

1. 解码器（Decoder）：将接收到的字节流（ByteBuf）解码为应用层所使用的数据结构（如 POJO 对象）。常见的解码器有：ByteToMessageDecoder、LengthFieldBasedFrameDecoder 等。
2. 编码器（Encoder）：将应用层的数据结构编码为字节流，以便在网络中传输。常见的编码器有：MessageToByteEncoder、LengthFieldPrepender 等。
3. 业务逻辑处理器：处理应用层的业务逻辑，如数据库操作、业务计算等。业务逻辑处理器通常需要继承 ChannelInboundHandlerAdapter 或 ChannelOutboundHandlerAdapter，并实现相应的事件处理方法。

3、netty中的helloworld

首先创建Handler类，该类用于接收服务器端发送的数据，这是一个简化的类，只重写了消息读取方法channelRead0、捕捉异常方法exceptionCaught。

（1）定义客户端的处理器

客户端的Handler一般继承的是SimpleChannelInboundHandler，该类有丰富的方法，心跳、超时检测、连接状态等等，代码如下：

```
@ChannelHandler.Sharable
public class HandlerClientHello extends SimpleChannelInboundHandler<ByteBuf>
{
    @Override
    protected void channelRead0(ChannelHandlerContext channelHandlerContext, ByteBuf
byteBuf) throws Exception
    {
        /**
        * @Description  处理接收到的消息
        **/
        System.out.println("接收到的消息："+byteBuf.toString(CharsetUtil.UTF_8));
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throwsException
    {
        /**
        * @Description  处理I/O事件的异常
        **/
        cause.printStackTrace();
        ctx.close();
    }
}
```

代码说明：

- 1）@ChannelHandler.Sharable：这个注解是为了线程安全，如果你不在乎是否线程安全，不加也可以；
- 2）SimpleChannelInboundHandler：这里的泛型可以是ByteBuf，也可以是String，还可以是对象，根据具体的实际情况来；
- 3）channelRead0：读取消息的方法，注意名称中有个0；
- 4）ChannelHandlerContext：通道上下文，代指Channel；
- 5）ByteBuf：字节序列，通过ByteBuf操作基础的字节数组和缓冲区，因为JDK原生操作字节麻烦、效率低，所以Netty对字节的操作进行了封装，实现了指数级的性能提升，同时使用更加便利；
- 6）CharsetUtil：这个是JDK原生的方法，用于指定字节数组转换为字符串时的编码格式。

（2）创建客户端

客户端启动类根据服务器端的IP和端口，建立连接，连接建立后，实现消息的双向传输，代码较简洁，如下：

```
public class AppClientHello
```

```
{
    private final String host;
    private fina lint port;

    public AppClientHello(String host, int port)
    {
        this.host = host;
        this.port = port;
    }

    public void run() throws Exception
    {
        //定义干活的线程池，I/O线程池
        EventLoopGroup group = new NioEventLoopGroup();
        try{
            Bootstrap bs = new Bootstrap();//客户端辅助启动类
            bs.group(group)
                .channel(NioSocketChannel.class)//实例化一个Channel
                .remoteAddress(newInetSocketAddress(host,port))
                .handler(newChannelInitializer<SocketChannel>()//进行通道初始化配置
                        {
                            @Override
                            protected void initChannel(SocketChannel socketChannel)
throws Exception
                            {

socketChannel.pipeline().addLast(newHandlerClientHello());//添加我们自定义的Handler
                            }
                        });

            //连接到远程节点；等待连接完成
            ChannelFuture future=bs.connect().sync();

            //发送消息到服务器端，编码格式是utf-8
            future.channel().writeAndFlush(Unpooled.copiedBuffer("Hello World",
CharsetUtil.UTF_8));

            //阻塞操作，closeFuture()开启了一个channel的监听器（这期间channel在进行各项工作），
直到链路断开
            future.channel().closeFuture().sync();

        } finally{
            group.shutdownGracefully().sync();
        }
    }

    public static void main(String[] args) throws Exception
    {
        new AppClientHello("127.0.0.1",18080).run();
    }
}
```

由于代码中已经添加了详尽的注释，这里只对极个别的进行说明：

- 1）ChannelInitializer：通道Channel的初始化工作，如加入多个handler，都在这里进行；
- 2）bs.connect().sync()：这里的sync()表示采用的同步方法，这样连接建立成功后，才继续往下执行；
- 3）pipeline()：连接建立后，都会自动创建一个管道pipeline，这个管道也被称为责任链，保证顺序执行，同时又可以灵活的配置各类Handler，这是一个很精妙的设计，既减少了线程切换带来的资源开销、避免好多麻烦事，同时性能又得到了极大增强。

`ChannelFuture`代表一个异步的I/O操作的结果或状态。在Netty中，几乎所有的I/O操作都是异步执行的，这就意味着当您调用一个方法来执行某个操作时，该方法会立即返回一个`ChannelFuture`对象，而不会阻塞当前线程等待操作完成。

`ChannelFuture`提供了以下几个主要的功能：

1. **异步操作结果**：`ChannelFuture`提供了方法来检查操作是否已完成，是否成功或失败，以及获取操作的结果。您可以通过调用`isDone()`来检查操作是否已完成，`isSuccess()`来检查操作是否成功，`cause()`来获取操作失败的原因，`get()`来获取操作的结果（会阻塞当前线程），或者通过`addListener()`添加监听器，在操作完成时执行回调方法。
2. **操作的连续性**：`ChannelFuture`提供了一系列方法来支持操作的连续性。例如，您可以通过`await()`方法阻塞当前线程，直到操作完成，或者通过`awaitUninterruptibly()`方法以无中断方式等待操作完成。此外，您还可以通过`sync()`方法在操作完成前阻塞当前线程，并在操作失败时抛出异常。
3. **操作的顺序控制**：`ChannelFuture`可以与其他`ChannelFuture`进行组合，以控制操作的顺序。通过调用`addListener()`并在回调方法中处理下一个操作，您可以实现操作的串行执行或者依赖关系。

总之，`ChannelFuture`是Netty中表示异步I/O操作结果的重要概念。它提供了一组方法来管理和处理操作的状态、结果和连续性，以便您可以编写具有高性能和灵活性的异步网络应用程序。

（3）创建服务器处理器

和客户端一样，只重写了消息读取方法channelRead(注意这里不是channelRead0)、捕捉异常方法exceptionCaught。

另外服务器端Handler继承的是ChannelInboundHandlerAdapter，而不是SimpleChannelInboundHandler，至于这两者的区别，这里不赘述，大家自行百度吧。

代码如下：

```
@ChannelHandler.Sharable
public class HandlerServerHello extends ChannelInboundHandlerAdapter
{
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)  throws Exception
    {
        //处理收到的数据，并反馈消息到到客户端
        ByteBuf in = (ByteBuf) msg;
        System.out.println("收到客户端发过来的消息: "+ in.toString(CharsetUtil.UTF_8));

        //写入并发送信息到远端（客户端）
        ctx.writeAndFlush(Unpooled.copiedBuffer("你好，我是服务端，我已经收到你发送的消息",
CharsetUtil.UTF_8));
    }

    @Override
```

```
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws
Exception
    {
        //出现异常的时候执行的动作（打印并关闭通道）
        cause.printStackTrace();
        ctx.close();
    }
}
```

以上代码很简洁，大家注意和客户端Handler类进行比较。

(4) 创建服务器

```
public class AppServerHello
{
    private int port;

    public AppServerHello(int port)
    {
        this.port = port;
    }

    public void run() throws Exception
    {
        //Netty的Reactor线程池，初始化了一个NioEventLoop数组，用来处理I/O操作,如接受新的连接和读/
写数据
        EventLoopGroup boss = new NioEventLoopGroup();
        EventLoopGroup worker = new NioEventLoopGroup();
        try{
            ServerBootstrap b = newServerBootstrap();//用于启动NIO服务
            b.group(boss,worker)
                .channel(NioServerSocketChannel.class) //通过工厂方法设计模式实例化一个
channel
                .localAddress(newInetSocketAddress(port))//设置监听端口
                .childHandler(newChannelInitializer<SocketChannel>() {
                    //ChannelInitializer是一个特殊的处理类，他的目的是帮助使用者配置一个新的
Channel,用于把许多自定义的处理类增加到pipline上来
                    @Override
                    //ChannelInitializer 是一个特殊的处理类，他的目的是帮助使用者配置一个新的
Channel。
                    public void initChannel(SocketChannel ch) throws Exception {
                        //配置childHandler来通知一个关于消息处理的InfoServerHandler实例
                        ch.pipeline().addLast(new HandlerServerHello());
                    }
                });

            //绑定服务器，该实例将提供有关IO操作的结果或状态的信息
            ChannelFuture channelFuture= b.bind().sync();
            System.out.println("在"+ channelFuture.channel().localAddress()+"上开启监
听");
```

```
            //阻塞操作，closeFuture()开启了一个channel的监听器（这期间channel在进行各项工作），直到链路断开
            // closeFuture().sync()会阻塞当前线程，直到通道关闭操作完成。这可以用于确保在关闭通道之前，程序不会提前退出。
            channelFuture.channel().closeFuture().sync();
        } finally{
            group.shutdownGracefully().sync();//关闭EventLoopGroup并释放所有资源，包括所有创建的线程
        }
    }

    public static void main(String[] args)  throws Exception
    {
        new AppServerHello(8080).run();
    }
}
```

代码说明：

- 1）EventLoopGroup：实际项目中，这里创建两个EventLoopGroup的实例，一个负责接收客户端的连接，另一个负责处理消息I/O。
- 2）NioServerSocketChannel：通过工厂通过工厂方法设计模式实例化一个channel，这个在大家还没有能够熟练使用Netty进行项目开发的情况下，不用去深究。

通常我们会将ChannelPipeline的定义放在一个独立的外部类中，如下：

```
public class MyChannelInitializer extends ChannelInitializer<SocketChannel> {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();

        // 添加解码器
        pipeline.addLast(new LengthFieldBasedFrameDecoder(65535, 0, 2, 0, 2));
        pipeline.addLast(new MyMessageDecoder());

        // 添加编码器
        pipeline.addLast(new LengthFieldPrepender(2));
        pipeline.addLast(new MyMessageEncoder());

        // 添加业务逻辑处理器
        pipeline.addLast(new MyBusinessHandler());
    }
}
```

在这个示例中，我们首先创建了一个自定义的 ChannelInitializer，并重写了 initChannel 方法。在该方法中，我们通过 ch.pipeline() 获取 ChannelPipeline 的实例，然后依次添加解码器、编码器和业务逻辑处理器。这样，当有新的连接建立时，Netty 会自动调用 initChannel 方法，为新连接创建一个包含指定处理器的 ChannelPipeline。

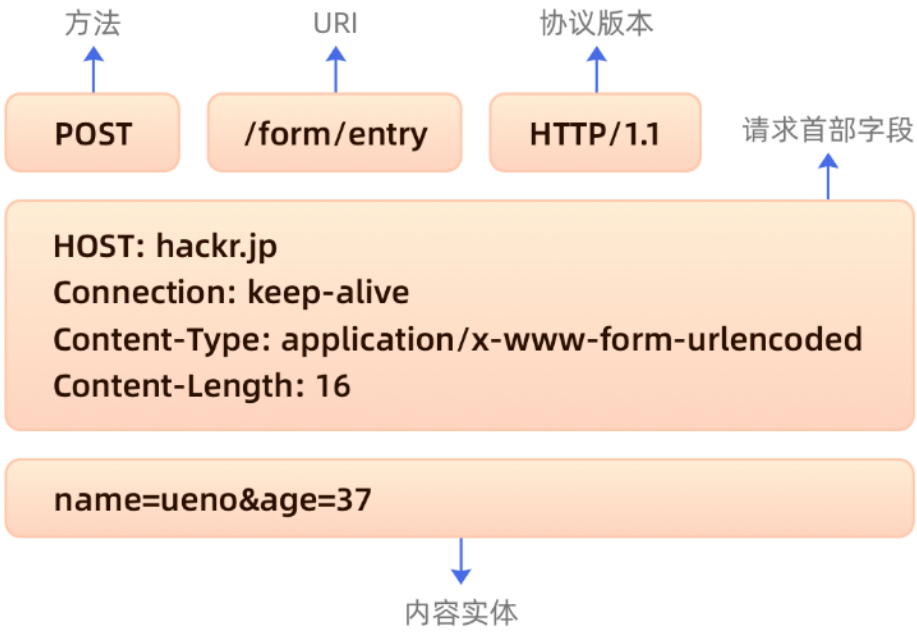
通过灵活地组合不同的 ChannelHandler，用户可以轻松地实现各种网络协议和应用逻辑。

四、封装报文

在设计一个 rpc（Remote Procedure Call）远程调用框架时，需要考虑如何**对请求和响应数据进行封装、以及编码、解码**，以及如何表示调用的方法和参数。此时，我们必须要设计一个私有且通用的私有协议，协议是一种公平对话的模式，有了标准协议调用方和服务提供方就可以互相按照标准进行协商。

1、设计私有协议

我们想发送数据的时候必须遵循一些规范，比如dubbo中就封装了dubbo协议。事实上任何基于tcp上的应用层的通信方式都是一种协议，如下图的http协议，是我们最熟悉不过的应用层协议了：



相对于 HTTP 的而言，rpc 更多的是负责**应用间的通信**，所以性能要求相对更高。但 HTTP 协议的数据包大小相对请求数据本身要大很多，又需要加入很多无用的内容，比如换行符号、回车符等；还有一个更重要的原因是，HTTP 协议属于**无状态协议**，客户端无法对请求和响应进行关联，每次请求都需要重新建立连接，响应完成后再关闭连接。因此，对于要求高性能的 rpc 来说，HTTP 协议基本很难满足需求，所以 rpc 会选择设计更紧凑的私有协议。

2、协议结构

我们的项目设计的协议分为 Header（头部）和 Body（主体）两部分。Header 包含协议的元数据，例如消息类型、序列化类型、请求ID 等。Body 包含实际的 yrpc 请求或响应数据。

```
+-----------------------------------------------+
|                    Header                     |
+-----------------------------------------------+
|                      Body                     |
+-----------------------------------------------+
```

Header 结构

Header 可以包含以下字段：

- Magic Number（4 字节）：魔数，用于识别该协议，例如：0xCAFEBABE。
- Version（1 字节）：协议版本号。
- MessageType（1 字节）：消息类型，例如：0x01 表示请求，0x02 表示响应。
- Serialization Type（1 字节）：序列化类型，例如：0x01 表示 JSON，0x02 表示 Protobuf 等。
- Request ID（8 字节）：请求ID，用于标识请求和响应的匹配。
- Body Length（4 字节）：Body 部分的长度。
- head length(4 字节)

Body 结构

Body 的结构取决于具体的 yrpc 请求或响应数据。

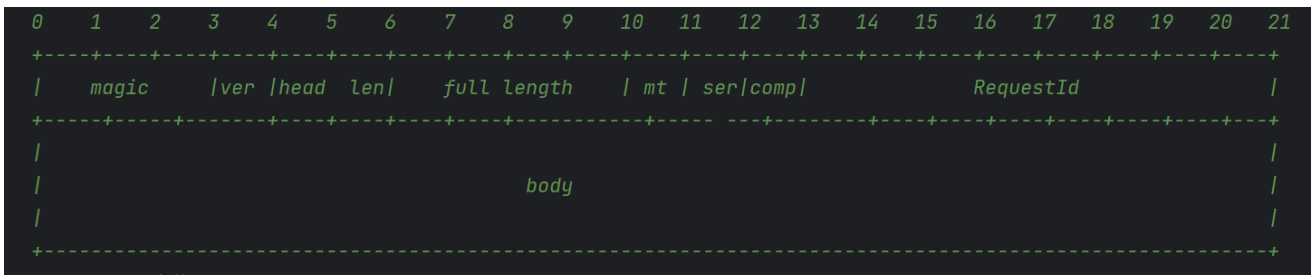
对于 yrpc 请求，Body 可以包含以下字段：

- Service Name：被调用的服务名称。
- Method Name：被调用的方法名称。
- Method Arguments：被调用方法的参数列表。
- Method Argument Types：被调用方法参数的类型列表。

对于 yrpc 响应，Body 可以包含以下字段：

- Status Code：响应状态码，例如：0x00 表示成功，0x01 表示失败。
- Error Message：错误信息，当 Status Code 为失败时，包含具体的错误信息。
- Return Value：方法返回值，当 Status Code 为成功时，包含方法调用的返回值。

大致如下：



代码案例如下：

```
@Test
public void testMessage() throws IOException {
    ByteBuf message = Unpooled.buffer();
    message.writeBytes("ydl".getBytes(StandardCharsets.UTF_8));
    message.writeByte(1);
    message.writeShort(125);
    message.writeInt(256);
    message.writeByte(1);
    message.writeByte(0);
    message.writeByte(2);
    message.writeLong(251455L);
    // 用对象流转化为字节数据
    AppClient appClient = new AppClient();
```

```
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(outputStream);
        oos.writeObject(appClient);
        byte[] bytes = outputStream.toByteArray();
        message.writeBytes(bytes);

        printAsBinary(message);

    }

    public static void printAsBinary(ByteBuf byteBuf) {
        byte[] bytes = new byte[byteBuf.readableBytes()];
        byteBuf.getBytes(byteBuf.readerIndex(), bytes);

        String binaryString = ByteBufUtil.hexDump(bytes);
        StringBuilder formattedBinary = new StringBuilder();

        for (int i = 0; i < binaryString.length(); i += 2) {
            formattedBinary.append(binaryString.substring(i, i + 2)).append(" ");
        }

        System.out.println("Binary representation: " + formattedBinary.toString());
    }
```

五、序列化

网络传输中，我们不能直接将堆内存的对象实例直接进行传输，而是需要将其序列化成一组二进制数据，这样的二进制数据可以是字符序列，最简单的莫过于我们熟悉的json字符序列，当然，事实上只要是一组可逆的转换过程都可以，如：

- 1、jdk的ObjectInputStream
- 2、Hession
- 3、json
- 4、protobuf等

事实上我们会发现一个问题，不同的序列化方式，在序列化后的信息密度是不一样的，像json这样，我们可以轻易读懂，也就意味着他的信息密度是最小的，也就是序列化后的体积是最大的，传输传输过程中需要的带宽也是最大的，选用什么样的序列化方式，也要和我们的系统的特性相结合。当然为了让我们的框架更加的灵活和具备可扩展性，我们可以灵活配置序列化方式。

以下是jdk的序列化代码：

```
@Slf4j
public class JdkSerializer implements Serializer {
    @Override
    public byte[] serialize(Object obj) {
        if(log.isDebugEnabled()){
            log.debug("Serialization is being done using jdk.");
        }
        try(ByteArrayOutputStream baos = new ByteArrayOutputStream();
            ObjectOutputStream out = new ObjectOutputStream(baos)) {
            out.writeObject(obj);
            return baos.toByteArray();
        } catch (IOException e) {
            log.error("对象[{}]序列化过程发生了异常! ",obj,e);
            throw new SerializeException("An exception occurred while implementing
serialization using jdk");
        }

    }

    @Override
    public <T> T deserialize(byte[] bytes, Class<T> clazz) {
        if(log.isDebugEnabled()){
            log.debug("Deserialization is being done using jdk.");
        }
        try (
            ByteArrayInputStream bin = new ByteArrayInputStream(bytes);
            ObjectInputStream ois = new ObjectInputStream(bin);
        ){
            Object readObject = ois.readObject();
            return (T)readObject;
        } catch (IOException | ClassNotFoundException e){
            log.error("class[{}]反序列化过程发生了异常! ",clazz.getName(),e);
            throw new SerializeException("An exception occurred while implementing
deserialization using jdk");
        }
    }

    @Override
    public byte getCode() {
        return 1;
    }
}
```

六、压缩

如果我们觉得序列化后的二进制内容体积任然比较大，任然不能支持当前的业务容量，我们可以选择对序列化的结果进行压缩，但是开启压缩一定要注意，这个操作本是就是一个**cpu资源换取存储和带宽资源**的操作，要判断当前的业务是更需要cpu资源还是内存资源。

通常我们使用gzip的方式进行压缩：

```
public class GzipCompress implements Compress {

    private static final int BUFFER_SIZE = 1024 * 4;
    @Override
    public byte[] compress(byte[] bytes) {
        if (bytes == null) {
            throw new CompressException("We tried to compress a byte array, but it was
null.");
        }
        try (ByteArrayOutputStream out = new ByteArrayOutputStream();
             GZIPOutputStream gzip = new GZIPOutputStream(out)) {
            gzip.write(bytes);
            gzip.flush();
            gzip.finish();
            return out.toByteArray();
        } catch (IOException e) {
            throw new RuntimeException("gzip compress error", e);
        }
    }

    @Override
    public byte[] decompress(byte[] bytes) {
        if (bytes == null) {
            throw new CompressException("We tried to decompress a byte array, but it
was null.");
        }
        try (ByteArrayOutputStream out = new ByteArrayOutputStream();
             GZIPInputStream gunzip = new GZIPInputStream(new
ByteArrayInputStream(bytes))) {
            byte[] buffer = new byte[BUFFER_SIZE];
            int n;
            while ((n = gunzip.read(buffer)) > -1) {
                out.write(buffer, 0, n);
            }
            return out.toByteArray();
        } catch (IOException e) {
            throw new CompressException("gzip decompress error", e);
        }
    }

    @Override
    public byte getCode() {
        return 1;
    }
}
```

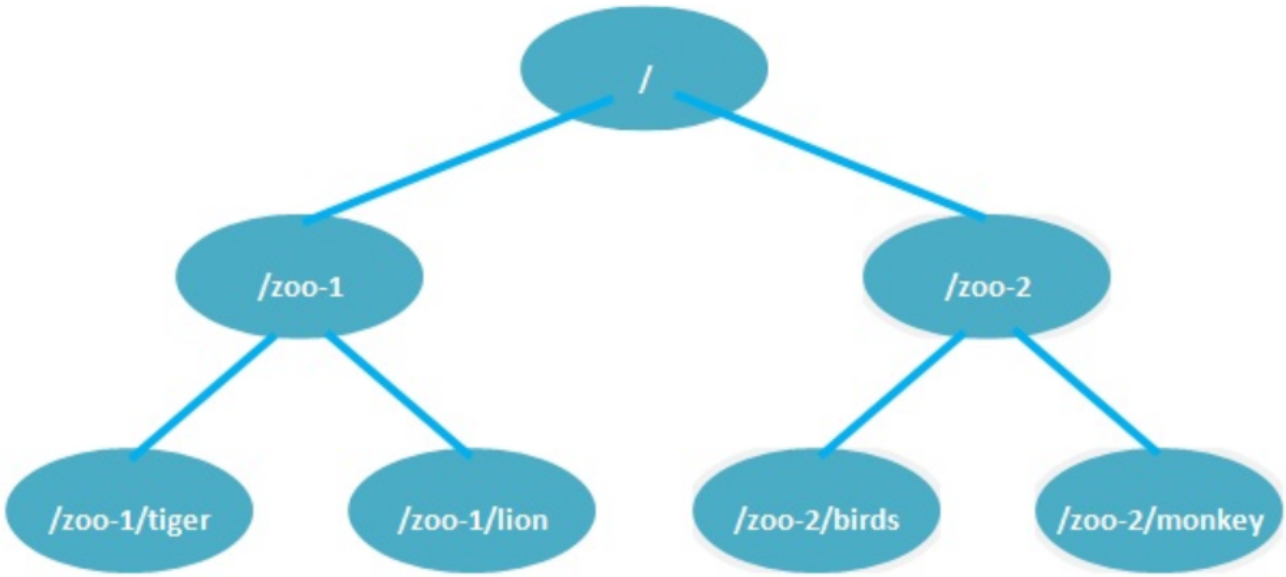
第三章、zookeeper

一、概述

- 概念：解释ZooKeeper是一个开源的**分布式协调服务组件**，用于构建可靠的分布式系统。它提供了一个高性能的、有序的、可靠的分布式数据存储和协调服务，简单的理解为一个内存数据库，特殊的数据结构和一些特性，他可以实现一些特殊的功能。
- 应用场景：在分布式系统中我们经常使用ZooKeeper实现**服务注册发现、分布式锁、配置管理、命名服务和分布式协调**等功能。
- 数据模型：ZooKeeper的数据模型是一个类似于**文件系统的层次结构**。每个节点称为ZNode，它可以存储数据和子节点。zookeeper中的ZNode可以分为持久节点和临时节点。
- Watcher机制：ZooKeeper允许开发人员在节点上设置监视点，以便在节点发生更改时接收通知。

二、数据模型

ZooKeeper的数据模型是一个类似于**文件系统的层次结构**，被组织成一个树形结构，每个节点称为ZNode。ZNode是ZooKeeper中的基本数据单元，它可以存储**数据和子节点**。



以下是ZooKeeper数据模型的详细说明：

1. 树形结构：

   ZooKeeper的数据模型是一个树形结构，类似于文件系统的目录结构。整个树由根节点（称为"/"）开始，每个节点可以有多个子节点。

   每个节点都**有一个路径来唯一标识它**，路径是由斜杠（/）分隔的一系列名称组成。

2. ZNode：

   ZNode是ZooKeeper的**基本数据单元**，类似于文件系统中的文件或目录。

   每个ZNode可以**存储一个字节数组作为其数据，可以是任意类型的数据，例如配置信息、状态信息等。**

   每个ZNode还可以**有多个子节点，形成层次结构。**

3. 持久节点（Persistent Node）：

   持久节点在创建后一直存在，**直到主动删除。**

   持久节点的数据和子节点都是持久的，即它们在节点创建后仍然存在，直到被显式删除。

4. 临时节点（Ephemeral Node）：

临时节点的**生命周期与客户端会话绑定**，当会话结束（例如客户端关闭或与ZooKeeper集合的连接断开）时，**临时节点将自动被删除。**

临时节点的数据和子节点也将随之被删除。

5. 顺序节点（Sequential Node）：

   顺序节点在创建时会自动**分配一个全局唯一且递增的编号。**

   顺序节点的编号由ZooKeeper集合维护，可以用于实现有序性或生成全局唯一的标识符。

三、watcher机制

Watcher机制是ZooKeeper中非常重要的概念，它允许客户端在ZooKeeper上设置监视点，以便在节点发生变化时接收通知。Watcher机制使得开发人员可以及时获取关于数据变化的通知，以便采取相应的操作。

以下是Watcher机制的详细说明：

1. 注册Watcher：

   客户端可以通过在对节点**进行操作时注册Watcher来设置监视点。**例如，在创建、更新或删除节点时都可以注册Watcher。

   客户端在注册Watcher时需要指定监视的路径和Watcher对象。当指定路径的节点**发生变化时，ZooKeeper会将通知发送给客户端。**

2. Watcher通知：

   当一个节点发生变化时，ZooKeeper会将通知发送给注册了Watcher的客户端。

   Watcher通知是一次性的，也就是说，当客户端接收到Watcher通知后，该Watcher将被移除，需要客户端**重新注册Watcher才能再次接收通知。**

3. Watcher的类型：

   数据变更触发的Watcher（Data Watcher）：当节点的数据发生变化时触发，例如节点的值被修改。

   子节点变更触发的Watcher（Child Watcher）：当节点的子节点发生变化时触发，例如新增或删除子节点。

   连接状态变更触发的Watcher（Existence Watcher）：当客户端与ZooKeeper集合的连接状态发生变化时触发，例如连接断开或重新连接。

4. Watcher的触发流程：

   当一个节点发生变化时，ZooKeeper首先会**将通知发送给与该节点相关的Watcher。**

   客户端接收到Watcher通知后，需要处理通知并根据需要采取相应的操作，例如重新读取数据或重新注册Watcher。

5. Watcher的注意事项：

   Watcher通知是异步的，**客户端需要保证处理Watcher通知的代码是线程安全的。**

   Watcher通知是**最终一致性的**，即ZooKeeper不能保证Watcher通知的实时性，只能保证最终一致性。

   Watcher通知是有序的，当多个Watcher同时触发时，ZooKeeper会**按照注册顺序依次发送通知。**

通过理解Watcher机制，开发人员可以更好地利用ZooKeeper来实现分布式系统中的实时数据变更和协调操作。Watcher机制提供了一种高效且可靠的方式，使得分布式系统能够实时响应节点变化，保持数据的一致性。

四、安装和基本操作

ZooKeeper提供了一组命令行工具（CLI）来与ZooKeeper集群进行交互。以下是一些常见的ZooKeeper命令：

1. connect：
   - 连接到ZooKeeper集群。
   - 语法：`connect <host:port>`
2. ls：
   - 列出指定路径下的子节点。
   - 语法：`ls <path>`
3. create：
   - 创建一个节点。
   - 语法：`create <path> <data>`
4. get：
   - 获取指定节点的数据。
   - 语法：`get <path>`
5. set：
   - 设置指定节点的数据。
   - 语法：`set <path> <data>`
6. delete：
   - 删除指定节点。
   - 语法：`delete <path>`
7. stat：
   - 获取指定节点的详细信息，包括数据版本、ACL权限等。
   - 语法：`stat <path>`
8. getAcl：
   - 获取指定节点的ACL（访问控制列表）权限信息。
   - 语法：`getAcl <path>`
9. setAcl：
   - 设置指定节点的ACL权限信息。
   - 语法：`setAcl <path> <acl>`
10. quit/exit：
    - 退出ZooKeeper命令行工具。
    - 语法：`quit` 或 `exit`

这些命令可以通过在ZooKeeper命令行工具中输入相应的命令来执行。使用这些命令，可以查看和操作ZooKeeper集群中的节点和数据，管理ACL权限，以及执行其他与ZooKeeper相关的操作。

五、java操作

1、基础api

当使用Java客户端连接ZooKeeper时，您可以使用Maven来管理项目依赖。下面是使用Maven连接ZooKeeper的基本步骤：

1、创建Maven项目：首先，创建一个新的Maven项目。您可以使用任何IDE或命令行工具来执行此操作。

2、添加Maven依赖：在您的Maven项目的`pom.xml`文件中，添加以下依赖项来引入ZooKeeper客户端库：

```
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.8.1</version>
</dependency>
```

在上述代码中，我们指定了ZooKeeper客户端库的`groupId`为`org.apache.zookeeper`，`artifactId`为`zookeeper`，并指定了所需的版本号。您可以根据需要更新版本号。

3、测试用例

```
public class ZookeeperTest {

    ZooKeeper zooKeeper;

    @Before
    public void createZk(){

        // 定义连接参数
        String connectString = "127.0.0.1:2181";
        // 定义超时时间
        int timeout = 10000;
        try {
            // new MyWatcher() 默认的watcher
            zooKeeper = new ZooKeeper(connectString,timeout,new MyWatcher());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Test
    public void testCreatePNode(){
        try {
            String result = zooKeeper.create("/ydlclass", "hello".getBytes(),
                    ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
            System.out.println("result = " + result);
        } catch (KeeperException | InterruptedException e) {
            e.printStackTrace();
        } finally {
            try {
                if(zooKeeper != null){
                    zooKeeper.close();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
    @Test
    public void testDeletePNode(){
        try {
            // version: cas  mysql  乐观锁,  也可以无视版本号  -1
            zooKeeper.delete("/ydlclass",-1);
        } catch (KeeperException | InterruptedException e) {
            e.printStackTrace();
        } finally {
            try {
                if(zooKeeper != null){
                    zooKeeper.close();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    @Test
    public void testExistsPNode(){
        try {
            // version: cas  mysql  乐观锁,  也可以无视版本号  -1
            Stat stat = zooKeeper.exists("/ydlclass", null);

            zooKeeper.setData("/ydlclass","hi".getBytes(),-1);

            // 当前节点的数据版本
            int version = stat.getVersion();
            System.out.println("version = " + version);
            // 当前节点的acl数据版本
            int aversion = stat.getAversion();
            System.out.println("aversion = " + aversion);
            // 当前子节点数据的版本
            int cversion = stat.getCversion();
            System.out.println("cversion = " + cversion);


        } catch (KeeperException | InterruptedException e) {
            e.printStackTrace();
        } finally {
            try {
                if(zooKeeper != null){
                    zooKeeper.close();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

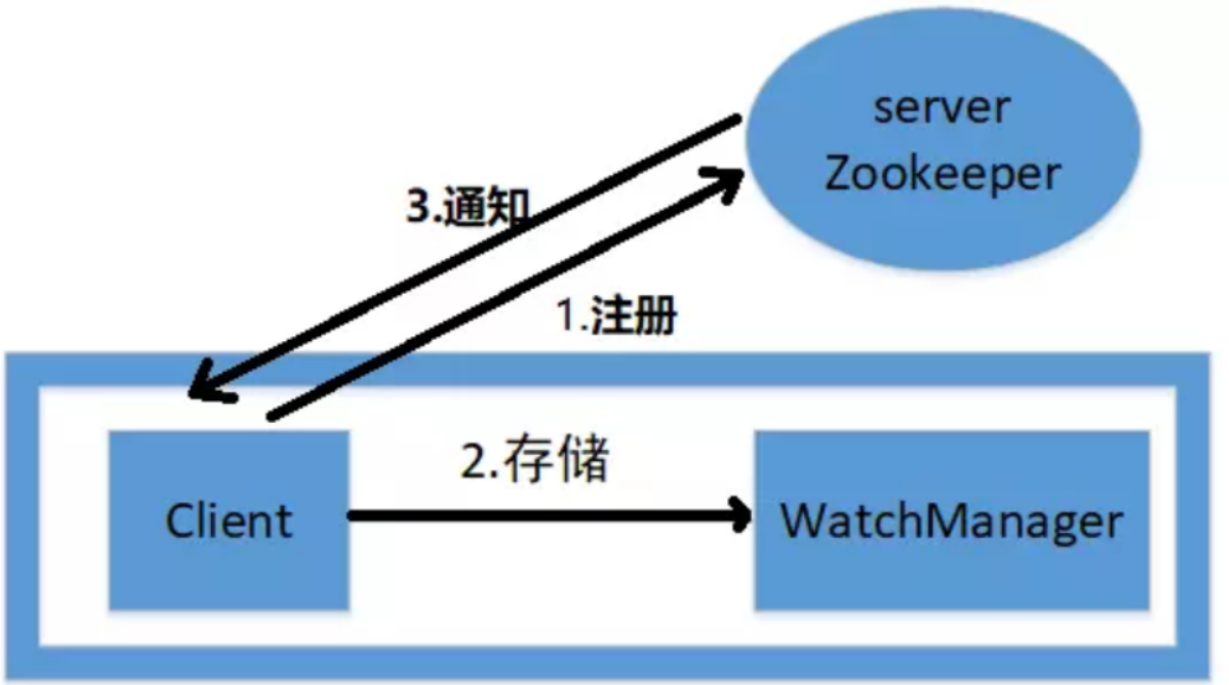
注: （1）version ： 当前结点的数据内容版本号；

（2）cversion：当前数据子节点的版本号；

（3）aversion：当前数据节点ACL变更的版本号；

2、watcher机制

watcher机制的本质是向zookeeper注册关心的事件，然后在本地存储钩子函数，当事件发生后调用钩子函数，如下图：



> 概念

- Zookeeper提供了数据的发布/订阅功能。**多个订阅者可监听某一特定主题对象（节点）**。当主题对象发生改变（数据内容改变，被删除等），会实时通知所有订阅者。该机制在被订阅对象发生变化时，会异步通知客户端，因此客户端不必在注册监听后轮询阻塞。
- Watcher机制实际上与观察者模式类似，也可看作观察者模式在分布式场景中给的一种应用。

> 特性

特性	说明
一次性	Watcher是一次性的，一旦触发，就会被移除，再次使用需要重新注册
轻量级	WatcherEvent是最小的通信单元，结构上只包含连接状态、事件类型和节点路径，并不会告诉数据节点变化前后的具体情况
时效性	watcher只有在当前session彻底时效时才会无效，若在session有效期内重新连接成功，则watcher依然存在

ZooKeeper中的读取操作getData、exist、getChildren 等都可以使用指定参数为节点设置监听。

Zookeeper监听有三个关键点：

1. 客户端对该节点**注册监听**，也就是客户端对该节点进行订阅。
2. 该节点发生改变，触发某一事件后，客户端会收到一个通知。可以执行相应回调执行相应动作。
3. 注册的监听只会生效一次，要想继续生效，就要在回调的方法中继续注册监听。

java api中 有三个方法可以注册监听，getData、exist、getChildren。

监听器：

- 接听器接口Watcher，我们可以实现该接口实现自定义的监听器注册到节点上。
- 事件类型可以分为**连接事件状态类型和节点事件类型。**
- 事件类型：由Watcher.Event.EventType枚举维护。

主要有5种类型：

1. NodeCreated：节点被创建时触发。
2. NodeDeleted：节点被删除时触发。
3. NodeDataChanged：节点数据被修改时触发。
4. NodeChildrenChanged：子节点被创建或者删除时触发。
5. NONE： 该状态就是连接状态事件类型。前面四种是关于节点的事件，这种是连接的事件，具体由Watcher.Event.KeeperState枚举维护。

注册事件的方式与节点事件的关系：

方式	NodeCreated	NodeDeleted	NodeDataChanged	NodeChildrenChanged
exist	可监控	可监控	可监控	不可监控
getData	不可监控	可监控	可监控	不可监控
getChildren	不可监控	可监控	不可监控	可监控

以上表格是对设置监听的方法对相应的事件是否可监控到。比如exist方法对节点删除事件不可监控，假如用该方法注册监听的话，节点删除时并不会触发事件回调。

连接事件类型，是指客户端连接时会触发的事件，由Watcher.Event.KeeperState枚举维护，主要有四个类型：

- SyncConnected： 客户端与服务器正常连接时触发的事件。
- Disconnected： 客户端与服务器断开连接时触发的事件。
- AuthFailed：身份认证失败时触发的事件
- Expired： 客户端会话Session超时时触发的事件。

我们举例并进行如下测试：

```
public class MyWatcher implements Watcher {
     @Override
     public void process(WatchedEvent event) {
         // 判断事件类型,连接类型的事件
         if(event.getType() == Event.EventType.None){
              if(event.getState() == Event.KeeperState.SyncConnected){
                  System.out.println("zookeeper连接成功");
```

```
                } else if (event.getState() == Event.KeeperState.AuthFailed){
                    System.out.println("zookeeper认证失败");
                } else if (event.getState() == Event.KeeperState.Disconnected){
                    System.out.println("zookeeper断开连接");
                }

            } else if (event.getType() == Event.EventType.NodeCreated){
                System.out.println(event.getPath() + "被创建了");
            } else if (event.getType() == Event.EventType.NodeDeleted){
                System.out.println(event.getPath() + "被删除了了");
            } else if (event.getType() == Event.EventType.NodeDataChanged){
                System.out.println(event.getPath() + "节点的数据改变了");
            } else if (event.getType() == Event.EventType.NodeChildrenChanged){
                System.out.println(event.getPath() + "子节点发生了变化");
            }
        }
}
```

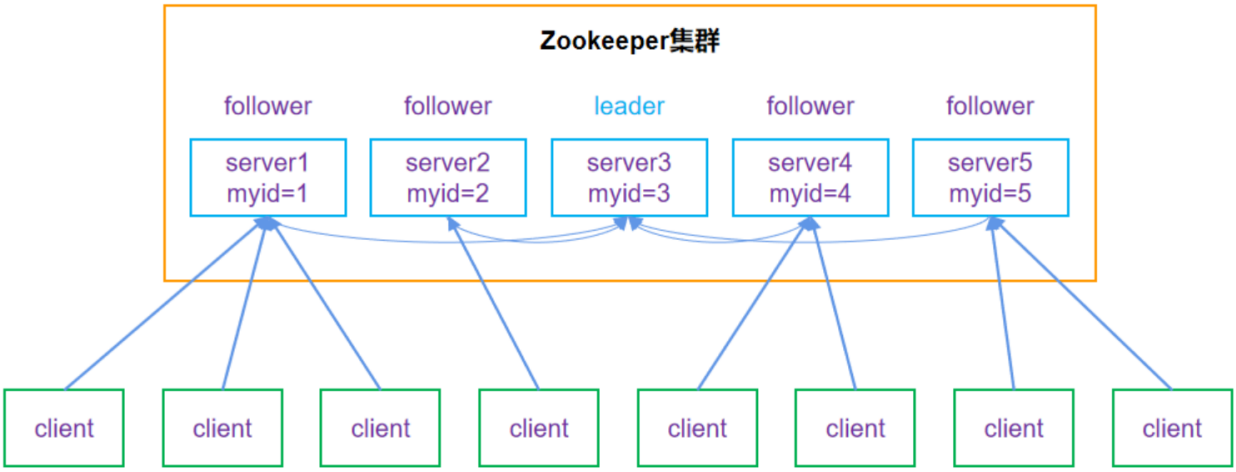
测试 用例:

```
@Test
public void testWatcher(){
    try {
        // 以下三个方法可以注册watcher，可以直接new一个新的watcher，
        // 也可以使用true来选定默认的watcher
        zooKeeper.exists("/ydlclass", true);
        //            zooKeeper.getChildren();
        //            zooKeeper.getData();

        while(true){
            Thread.sleep(1000);
        }

    } catch (KeeperException | InterruptedException e) {
        e.printStackTrace();
    } finally {
        try {
            if(zooKeeper != null){
                zooKeeper.close();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

六、集群安装

zookeeper的运行模式有单机模式，伪集群模式，集群模式三种。单个Zookeeper节点是会存在单点故障的，本小节我们搭建一个zk集群。Zookeeper节点部署越多，服务的可靠性越高，通常建议部署奇数个节点，因为zookeeper集群是以宕机个数过半才会让整个集群宕机的。

zookeeper的集群模式下，节点分为leader和follower两种状态，leader负责所有的写操作，follower负责相关的读操作。具体的细节我们后边详细介绍，本节主要是搭建：



1、准备环境

主机名	系统	IP地址
linux-node1	CentOS release 8	192.168.126.129
linux-node2	CentOS release 8	192.168.126.132
linux-node2	CentOS release 8	192.168.126.133

2、Zookeeper安装

Zookeeper运行需要java环境，需要安装jdk，注：每台服务器上面都需要安装zookeeper、jdk，建议本地下载好需要的安装包然后上传到服务器上面，服务器上面下载速度太慢。

修改zookeeper的配置文件，构建集群的基础配置：

```
dataLogDir=/opt/zookeeper/logs
dataDir=/opt/zookeeper/data
server.1= 192.168.126.129:2888:3888
server.2= 192.168.126.132:2888:3888
server.3= 192.168.126.133:2888:3888
```

server.1中的1指代第几个节点，2888端口用来辅助这个服务器与集群中的leader服务器做交换信息的端口，3888端口是在leader挂掉时专门用来进行选举leader所用的端口。

> 创建日志和持久化目录：

```
mkdir -p /opt/zookeeper/{logs,data}
```

> 创建ServerID标识

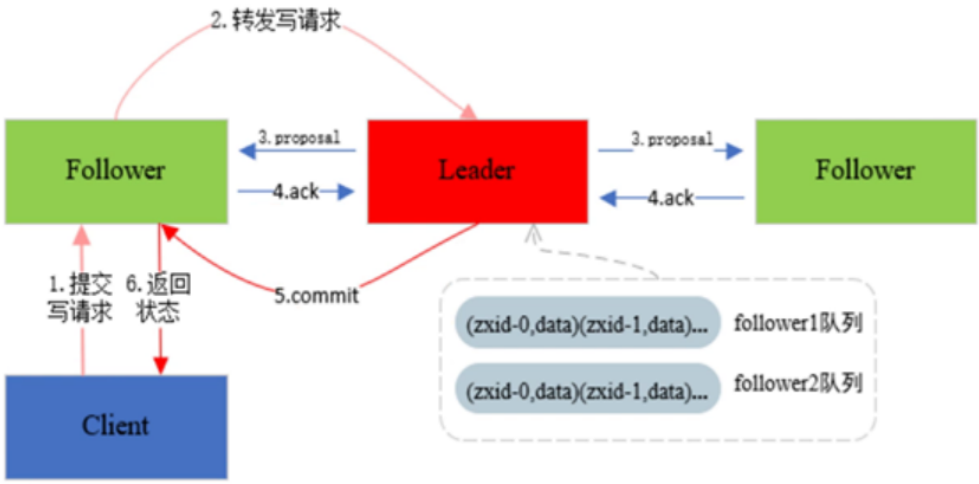
除了修改zoo.cfg配置文件外，zookeeper集群模式下还要配置一个myid文件，这个文件需要放在dataDir目录下。

这个文件里面有一个数据就是服务器编号，在zoo.cfg文件中配置的dataDir路径中创建myid文件。如在server.1服务器上面创建myid文件，就将他的值设置为1。

小节：ZooKeeper集群的启动过程包括以下步骤：

1. 准备配置文件：
    - 创建每个ZooKeeper节点的配置文件。配置文件包含节点的唯一标识（例如，ID）、监听地址和端口、数据目录等信息。
    - 配置文件还包括集群的信息，包括每个节点的连接信息和选举算法。
2. 启动ZooKeeper节点：
    - 在每个节点上运行ZooKeeper进程。可以使用命令行或脚本来启动每个节点。
    - 在启动命令中指定节点的配置文件，确保每个节点使用正确的配置。
3. 节点启动顺序：
    - 为了保证ZooKeeper集群的正常运行，节点的启动顺序很重要。通常，建议先启动节点ID较小的节点，然后依次启动其他节点。
    - 启动过程中，每个节点会尝试连接到其他节点并加入集群。
4. 初始化数据目录：
    - 在节点启动时，ZooKeeper会检查配置文件中指定的数据目录。
    - 如果数据目录不存在，ZooKeeper会自动创建，并初始化节点的数据存储结构。
5. 数据同步和选举：
    - 在节点加入集群后，ZooKeeper会开始进行数据同步和选举过程。
    - 数据同步是指将节点的数据状态与其他节点进行同步，以确保集群中的数据一致性。
    - 选举过程是为了选出一个节点作为Leader（领导者），负责协调和处理客户端请求。
6. Leader选举完成：
    - 一旦Leader选举完成，集群中的节点将分为Leader和Follower两类。
    - Leader节点负责处理所有写操作和协调集群的状态，而Follower节点负责复制和同步Leader节点的数据。
7. 集群就绪：
    - 当ZooKeeper集群中的节点完成数据同步和选举后，整个集群将进入就绪状态。
    - 客户端可以通过连接到任意一个节点来与集群进行通信和操作。

通过以上步骤，ZooKeeper集群中的节点将完成启动过程并开始提供服务。在启动过程中，注意配置文件的正确性和节点启动顺序的保持，这样可以确保集群的正常运行和数据的一致性。



七、CAP理论

前面我们学习了 zookeeper，了解了它的基本使用，接下来我们又学习了zookeeper的集群安装。但对于任何一个分布式系统而言，**数据同步永远都是重中之重。**因为**一个集群当中会有很多节点，那么客户端每次写数据的时候，是只向一个节点写入，还是向所有节点写入就成了一个问题。**

如果向所有节点写入，假设节点个数为 N，那么客户端的一次写请求就会被放大 N 倍，因为每个节点都要写一遍，显然这么做是非常不明智的。因此我们应该**让客户端只向一个节点写入**，然后该节点再将数据同步给集群内的其它节点。

但这就产生了一个问题，如果某个节点的数据同步还没有完成，就收到了客户端的读请求，那么显然会返回旧数据。如果想让客户端看到的一定是新数据，那么就必须等到数据在所有节点之间都同步完成之后，才能让客户端访问，而这又会造成集群服务出现**短暂的不可用。**

因此面对这种情况，我们必须要做出取舍，至于如何取舍，CAP 理论会告诉我们答案。它对分布式系统的特性进行了抽象，掌握了 CAP 理论，我们在面对分布式系统的时候就可以做到心中有数。

CAP 理论对分布式系统的特性做了高度抽象，形成了三个指标：

- 一致性（Consistency）/kənˈsɪstənsi/；
- 可用性（Availability）；
- 分区容错性（Partition Tolerance）；

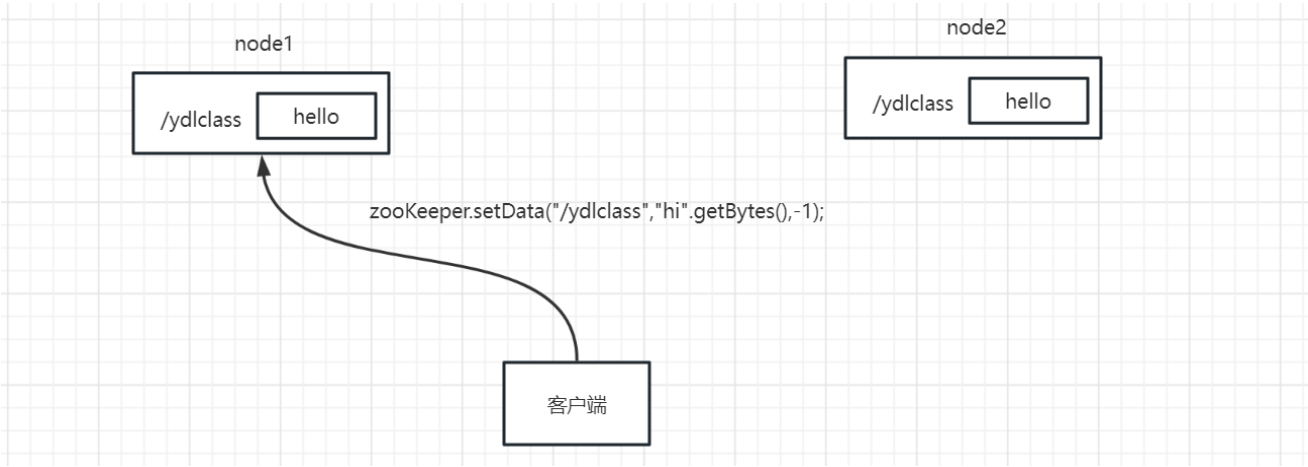
以上这三个指标就称之为 CAP，我们来分别介绍。

1、一致性，即 CAP 中的 C

一致性说的是客户端的每次读操作，不管访问哪个节点，**读到的都是同一份最新的数据**（或者读取失败，说明节点之间还在同步数据）。不会出现读不同节点，得到的数据不同这种情况。

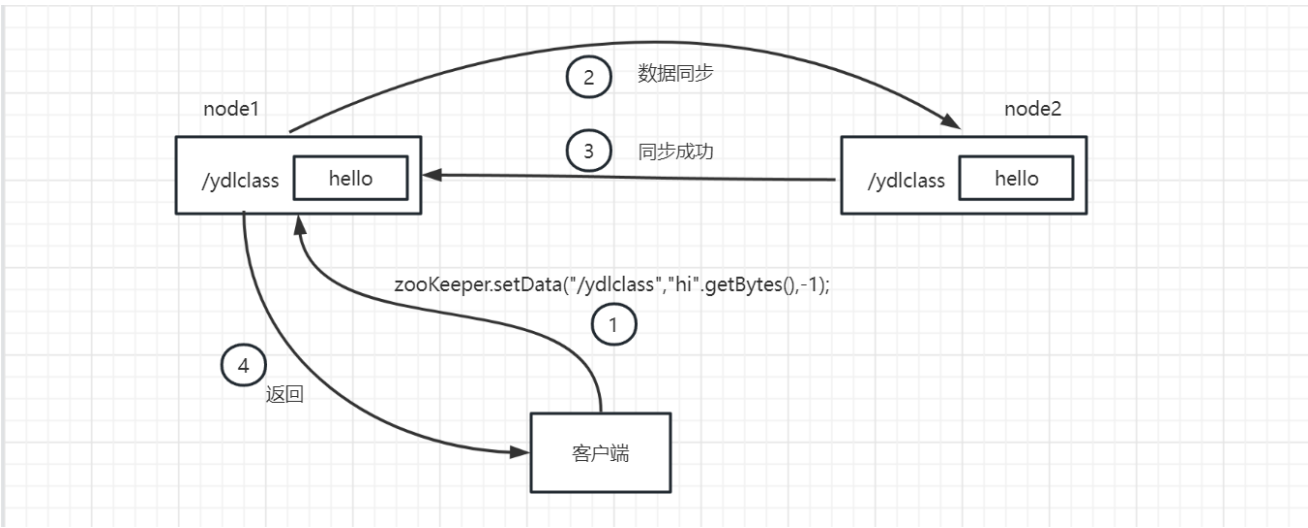
所以一致性强调的不是数据完整，而是各节点间的数据一致。

为了更好地理解一致性这个指标，我们以我们的zookeeper为例。假设当前有两个zookeeper节点，路径/ydlclass中保存的数据是hello，此时客户端请求将数据改为hi。



如果节点 1 收到写请求后，只修改自身数据，然后返回成功给客户端，那么这个时候节点 2 的数据还是hello，此时两个节点的数据就是**非一致的。**

如果节点 1 收到写请求后，**不仅自身更新数据，还通过节点间的通讯，将更新操作发送给节点 2**，等到自身和节点 2 的 数据都更新之后，再返回成功给客户端。那么当客户端完成写请求后，两个节点的数据就是一致的了。之后不管客户端访问哪个节点，读取到的都是同一份最新数据。



一致性这个指标，描述的是分布式系统非常重要的一个特性，强调的是数据的一致。也就是说，在客户端看来，**访问集群和访问单机是等价**的，因为两者在数据一致性上是一样的。

但**集群毕竟不是单机，**总会有**网络故障**的时候，那么当**节点之间无法通信**的时候该怎么办呢？比如节点1在将写请求同步给节点2的时候，发生了网络故障，这时候如果要保证一致性，也就是让客户端访问任何一个节点都能看到相同的数据，那么就应该拒绝服务（客户端读取失败），等到数据同步完成之后再提供服务。否则客户端就可能读到旧数据，比如访问节点 2 的时候，因为网络原因数据还没有同步过来。

因此可以把一致性看成是分布式系统对客户端的一种承诺：**不管访问哪个节点，返回的都是绝对一致的数据，因为数据不一致的时候会读取失败（拒绝提供服务）**。所以再次强调，一致性强调的不是数据完整，而是各节点之间的数据绝对一致。

但有些服务并不追求数据的一致性，返回旧数据也是可以的。当面对这种场景时，再因为节点间出现了通讯问题（会导致节点间的数据不一致）而拒绝提供服务，就有些不合适了。

这个时候我们就需要牺牲数据的一致性，每个节点使用本地数据来响应客户端请求，保证服务可用。这就是我们要说的另外一个指标，可用性。

2、可用性，即 CAP 中的 A

可用性说的是任何**来自客户端的请求，不管访问哪个节点，都能得到响应数据，但不保证是同一份最新数据。**

因此可以把可用性看作是分布式系统对客户端的另一种承诺：**尽量返回数据，不会不响应，但不保证每个节点返回的数据都是最新的。**因此可用性这个指标强调的是服务可用，但不保证数据的绝对一致。

3、分区容错性，即 CAP 中的 P

最后的分区容错性说的是，**当节点间出现任意数量的消息丢失或高延迟的时候，系统仍然可以继续提供服务。**也就是说，分布式系统会告诉客户端：不管我的内部出现什么样的数据同步问题，我会一直运行，提供服务。这个指标，强调的是**集群对分区故障的容错能力。**

比如当节点 1 和节点 2 通信出问题（发生网络分区）的时候，如果系统仍能提供服务，那么两个节点是满足分区容错性的。而分布式系统与单机系统不同，它涉及到多节点之间的通讯和交互，节点间的分区故障不可能完全避免，所以在分布式系统中分区容错性是必须要考虑的。

4、CAP 不可兼得

对于一个分布式系统而言，一致性、可用性、分区容错性 3 个指标不可兼得，只能在 3 个指标中选择两个。

我们知道只要有**网络交互就一定会有延迟和数据丢失**，而这种状况我们必须接受，还必须保证系统不能挂掉。所以就像上面提到的，节点间的分区故障是必然发生的。也就是说，**分区容错性（P）是前提，是必须要保证的**，不能说某些节点之间无法正常通信（发生网络分区）就导致整个集群不可用。

现在就只剩下一致性（C）和可用性（A）可以选择了：**要么选择一致性，保证数据绝对一致；要么选择可用性，保证服务可用。**如果选择 C，那么就是 CP 模型；如果选择 A，那么就是 AP 模型。

- 当选择一致性（C）的时候，如果因为消息丢失、延迟过高发生了网络故障，部分节点无法保证特定信息是最新的。那么这个时候，当集群节点接收到来自客户端的请求时，因为无法保证所有节点都是最新信息，所以系统将返回错误，也就是说拒绝请求。
- 当选择可用性（A）的时候，如果发生了网络故障，一些节点将无法返回最新的特定信息，那么它们将返回自己当前相对新的信息。

这里需要再强调一点，有很多人对 CAP 理论有个误解，认为无论在什么情况下，分布式系统都只能在 C 和 A 中选择 1 个。其实在不发生网络故障的情况下，也就是分布式系统正常运行时（这也是系统在绝大部分时候所处的状态），C 和 A 是能够**大致同时保证**的（如果节点之间的数据同步很快的话）。只有当发生分区故障的时候，也就是说需要 P 时，才会在 C 和 A 之间做出选择。

5、CAP 总结

以上就是 CAP 理论的具体内容，以及 CAP 理论的应用，总结如下：

1）CA 模型：

不支持分区容错，只支持一致性和可用性，但这在分布式系统中不存在。因为不支持分区容错性，也就意味着不允许分区异常，设备、网络永远处于理想的可用状态，从而让整个分布式系统满足一致性和可用性。

但分布式系统是由众多节点通过网络通信连接构建的，设备故障、网络异常是客观存在的，而且分布的节点越多，范围越广，出现故障和异常的概率也越大。因此对于分布式系统而言，分区容错性（P）是无法避免的，如果避免了 P，那么只能把分布式系统回退到单机单实例系统。就比如单机版关系型数据库 MySQL，如果 MySQL 要考虑主备或集群部署时，那么它也必须考虑 P。

2）CP 模型：

因为分区容错客观存在，所以放弃系统的可用性，换取一致性。采用 CP 模型的分布式系统，一旦因为消息丢失、延迟过高而发生了网络分区，就会持续阻塞整个服务，直到分区问题解决，才恢复对外服务，这样就可以保证数据的一致性。

选择 CP 一般都是对**数据一致性特别敏感，尤其是在支付交易领域**，Hbase 等分布式数据库领域，都要优先保证数据的一致性，在出现网络异常时，系统就会暂停服务处理。还有用来分发及订阅元数据的 Zookeeper、Etcd 等等，也是优先保证 CP 的。

3）AP 模型：

由于分区容错 P 客观存在，所以放弃系统的数据一致性，换取可用性。在系统遇到分区异常时，某些节点之间无法通信，数据处于不一致的状态。但为了保证可用性，服务节点在收到用户请求后会立即响应，因此只能返回各自新老不同的数据。

这种舍弃一致性，而保证系统在分区异常下的可用性，在互联网系统中非常常见。比如微博多地部署，如果不同区域出现网络中断，区域内的用户仍然能发微博、相互评论和点赞，但暂时无法看到其它区域用户发布的新微博和互动状态。

还有类似 12306 这种火车购票系统，在节假日高峰期抢票时也会遇到这种情况，明明某车次有余票，但真正点击购买时，却提示说没有余票。就是因为票已经被抢光了，票的可选数量应该更新为 0，但因并发过高导致当前访问的节点还没有来得及更新就提供服务了（和发生网络分区是类似的，都是最新数据还没有同步，就对外提供服务）。因此它返回的是更新之前的旧数据，但其实已经没有票了。

所以相比 CP，采用 AP 模型的分布式系统，更注重服务的高可用。用户访问系统的时候，都能得到响应数据，不会出现响应错误。但当出现分区故障、或者并发量过高导致数据来不及同步时，相同的读操作，访问不同的节点，得到的响应数据可能不一样。典型应用有 Cassandra, DynamoDB, Redis 等 NoSQL。

因此 CAP 理论可以帮助我们思考如何在一致性和可用性之间进行妥协折中，设计出满足场景特点的分布式系统。

最后再提一点，在分布式系统开发中，延迟是非常重要的一个指标。比如名字路由系统，通过延迟评估服务可用性，进行负载均衡和容灾；再比如在 Raft 实现中，通过延迟评估领导者节点的服务可用性，以及决定是否发起领导者选举；再比如类似 Redis 这种查询量非常大的分布式缓存，它的目的是能够快速地返回结果，所以它是 AP 模型。

所以在分布式系统的开发中，要能意识到延迟的重要性，能通过延迟来衡量服务的可用性。总之能否容忍短暂的延迟是关键。

6、BASE 理论

BASE 理论是 CAP 理论中的 AP 的延伸，所以它**强调的是可用性**，这个理论广泛应用在大型互联网的后台当中。它的核心思想就是**基本可用（Basically Available）和最终一致性（Eventually consistent）**。

首先「基本可用」指的是，当分布式系统在出现不可预知的故障时，允许损失部分功能的可用性，来保障核心功能的可用性。说白了就是服务降级，在服务器资源不够、或者说压力过大时，将一些非核心服务暂停，优先保证核心服务的运行。比如：

- 当业务应用访问的是非核心数据（例如电商商品属性）时，拒绝服务，或者直接返回预定义信息、空值或错误信息；当业务应用访问的是核心数据（例如电商商品库存）时，正常查询结果并返回；
- 还可以对用户体验进行降级，比如用小图片来替代原始图片，通过降低图片的清晰度和大小，提升系统的处理能力；

所以基本可用本质上是**一种妥协**，也就是在出现节点故障或系统过载的时候，通过**牺牲非核心功能的可用性，保障核心功能的稳定运行。**而手段也有很多，比如服务降级、体验降级、流量削峰、延迟响应、接口限流、服务熔断等等。

然后是最终一致性，它指的是**系统中所有的数据副本在经过一段时间的同步后，最终能够达到一致的状态。**也就是说在数据一致性上，存在一个短暂的延迟，几乎所有的互联网系统采用的都是最终一致性。比如 12306 买票，票明明卖光了，但还是显示有余票，说明此时数据不一致。但当你在真正购买的时候，又会提示你票卖光了，说明数据最终是一致的。

因此最终一致性应该不难理解，就是节点间的数据存在短暂的不一致，但经过一段时间后，最终会达到一致的状态。所以 BASE 理论除了引入一个基本可用之外，它和 AP 模型本质上没太大区别。

只有对数据有强一致性要求，才考虑 CP 模型或分布式事务，比如：决定系统运行的敏感元数据，需要考虑采用强一致性；与钱有关的支付系统或金融系统的数据，需要考虑采用事务保证一致性。因此，尽管事务型的分布式系统和强一致性的分布式系统，使用起来很方便，不需要考虑太多，就像使用单机系统一样。但是我们要知道，想在分布式系统中实现强一致性，必然会影响可用性。

如果换个角度思考，我们可以将强一致性理解为最终一致性的特例，也就是说可以把强一致性看作是不存在延迟的一致性。因此在实践中我们也可以这样思考：如果业务的某功能无法容忍一致性的延迟（比如分布式锁对应的数据），就需要强一致性；如果能容忍短暂的一致性的延迟（比如APP用户的状态数据），就可以考虑最终一致性。

> 所以我们之前介绍基于 Redis 实现分布式锁的时候，说过 Redis 在主从切换的时候会出问题，就是因为分布式锁需要的是 CP 模型，而 Redis 是 AP 模型。

小结：

BASE 理论是对 CAP 中一致性和可用性权衡的结果，它来源于对大规模互联网分布式系统实践的总结，是基于 CAP 定理逐步演化而来的。它的核心思想是，如果不是必须的话，不推荐使用事务或强一致性，鼓励可用性和性能优先，根据业务的场景特点，来实现非常弹性的基本可用，以及实现数据的最终一致性。

BASE 理论主张通过牺牲部分功能的可用性，实现整体的基本可用，也就是说通过服务降级的方式，努力保障极端情况下的系统可用性。

说到 BASE 理论，应该会有人想到 ACID 理论。ACID 是传统数据库常用的设计理念，追求强一致性模型；而 BASE 理论支持的是大型分布式系统，通过牺牲强一致性获得高可用性。BASE 理论在很大程度上，解决了事务型系统在性能、容错、可用性等方面的痛点。此外 BASE 理论在 NoSQL 中也应用广泛，是 NoSQL 系统设计的理论支撑。

对于任何集群而言，不可预知的故障的最终后果，都是系统过载。如何设计过载保护，实现系统在过载时的基本可用，是开发和运营互联网后台的分布式系统的重点。因此在开发实现分布式系统，要充分考虑如何实现基本可用。

八、选举和同步算法

1、写操作的具体流程

我们假设现在有一个写操作，需要在ZooKeeper集群服务中执行写操作，创建一个/ydlclass节点，其大致流程如下：

1. 客户端连接：首先，要创建节点的客户端需要与ZooKeeper集群中的任何一个服务器建立连接。
2. 发起写请求：客户端向Leader发送写请求，请求创建一个新的节点。
3. Leader处理写请求：Leader接收到写请求后，将生成一个全局唯一的ZooKeeper事务ID（ZXID），用来标识这个写操作。
4. 创建节点过程：Leader将写请求转发给其他Follower节点，并协调它们来完成创建节点的过程。具体步骤如下：

   a. Leader将写请求转发给Follower节点。

   b. Follower节点接收到写请求后，会记录下这个写操作的ZXID，并执行节点的创建操作。

   c. Follower节点将创建节点的操作结果返回给Leader。

   d. Leader收集Follower节点的操作结果，并基于大多数原则决定最终的写操作结果。
5. 数据同步过程：一旦写操作成功并被大多数节点接受，数据同步将在ZooKeeper集群中进行。具体步骤如下：

   a. Leader将写请求成功的结果通知给所有的Follower节点。

   b. Follower节点在收到通知后，会将Leader上的数据进行复制，确保自己的数据与Leader上的数据保持一致。

   c. Follower节点完成数据复制后，会向Leader发送确认通知。

   d. Leader在收到所有Follower节点的确认通知后，确定数据同步成功。
6. 客户端响应：一旦数据同步成功，Leader将向客户端发送操作成功的响应，表示节点创建完成。

> 已经执行了写操作还要数据同步吗？

在ZooKeeper中，Follower节点**执行写操作并返回成功结果给Leader是为了保证写操作的一致性和持久性**。数据同步是确保在整个集群中**所有节点的数据是一致的关键步骤**。

尽管Follower节点会在接收到写请求后**立即执行对应的操作，但在写操作的结果被确认之前，数据同步的过程是必要的。**这是因为ZooKeeper使用了**多数原则来决定写操作的最终结果**，只有在大多数节点都完成写操作并确认成功后，Leader才会确认写操作成功。此时，数据同步的过程才会开始。

值得注意的是，**数据同步是在写操作完成后才开始的**，这意味着在**数据同步期间的某个时间点，集群中的不同节点的数据可能是不一致的。**但是一旦数据同步完成，所有节点都将具有相同的数据，并保持一致性。这样做是为了在写操作期间保持高可用性，并在数据同步完成后确保数据的一致性。

> 数据同步是增量同步还是全量同步？

全量数据同步可能会对性能产生一定的影响，尤其是当数据量较大或者集群规模较大时。因为全量数据同步需要将所有相关节点上的数据进行复制，网络传输和处理的开销都可能会比较大。

为了减小全量数据同步的性能开销，ZooKeeper在设计上采取了一些优化措施：

1. 增量更新：ZooKeeper中的数据是以事务日志的形式进行持久化的。当数据有更新时，只会记录变更的内容，而不会每次都全量复制所有的数据。这样可以减小数据同步的开销。
2. 快速同步：当一个Follower节点加入集群时，它可以通过快速同步（Snapshot）的方式来获取最新的数据。只需要将Leader节点上的整个数据文件传输给新加入的Follower节点，而不是逐个复制增量更新。这样可以加快新节点的数据同步过程。

尽管全量数据同步可能对性能产生一定的影响，但这是为了保证集群中所有节点数据的一致性和可靠性。在实际应用中，可以通过合理的配置和优化集群硬件设施来提高性能，例如使用高性能的网络、增加机器的处理能力等，从而减少全量数据同步的性能开销。

2、选举流程

ZooKeeper是一个分布式协调服务，它的选举过程是为了确保集群中的某个节点成为leader，负责处理客户端的请求。下面是ZooKeeper选取leader的详细过程：

1. 初始化：当一个ZooKeeper服务器加入集群时，它会初始化自己的状态。这包括为自己分配一个唯一的标识符（myid）、创建临时节点（通过在ZooKeeper中创建一个带有自己标识符的临时顺序节点）以及与其他服务器建立连接等。
2. 选举过程的启动：当服务器初始化完成后，它会启动选举过程。在选举过程中，服务器将参与一个Leader选举协议。
3. 选举协议：ZooKeeper使用的是**基于Paxos的Zab协议。**在选举协议中，每个服务器都会成为候选者，并向其他服务器发送选举请求。集群中的服务器会根据规则进行投票。
4. 初始化投票：当一个服务器成为候选者时，它会初始化一个投票。这个投票包括服务器的标识符（myid）、ZXID（ZooKeeper事务ID）和一个状态（如LOOKING）等。
5. 发送投票请求：候选者会向其他服务器发送投票请求。每个服务器会将自己的投票响应发送回给候选者。
6. 收集和处理投票：候选者会收集所有服务器发送的投票，并根据投票的规则进行处理。通常情况下，候选者会根据ZXID和服务器的标识符进行投票的计算和比较。
7. 更改状态：候选者会根据投票结果修改自己的状态。如果它得到了大多数投票（超过半数），则会更改状态为LEADING，并成为leader；否则，它将继续参与选举过程。
8. 选举结果通知：当一个服务器成为leader后，它会将选举结果通知给其他服务器。其他服务器在收到通知后会更新自己的状态，并认可该服务器为leader。
9. 与客户端连接：leader将会为客户端提供服务，并处理客户端的请求。

需要注意的是，如果leader节点宕机或网络分区发生，ZooKeeper会触发新的选举过程以选取新的leader。这样可以确保集群的高可用性和持续的服务。

以上就是ZooKeeper选取leader的详细过程。选举过程保证了在集群中始终有一个可靠的leader节点，同时提供了良好的容错性和可用性。

假设现在有myid为1、2、3的三台ZooKeeper服务器，分别启动1、2、3三台服务，下面是集群启动时的选举过程的详细描述：

1. 初始状态：集群中的所有服务器处于LOOKING状态，即正在寻找leader节点。
2. 服务器1启动：服务器1作为第一台启动的服务器，它将成为Leader选举的候选者，并向其他服务器发送选举请求。
3. 服务器1的选举投票：服务器1将自己的投票信息（包括标识符myid=1、ZXID和状态LOOKING）发送给服务器2和服务器3。
4. 服务器2和服务器3收到选举请求：服务器2和服务器3收到来自服务器1的选举请求，并检查自己的状态。

5. 服务器2和服务器3的选举投票：服务器2和服务器3会分别初始化自己的投票，并将自己的投票信息发送回给服务器1。
6. 服务器1收集和处理投票：服务器1收到来自服务器2和服务器3的投票，它将根据规则进行投票的计算和比较。如果服务器1得到了大多数的投票（超过半数，即两票），它将更改自己的状态为LEADING，并成为leader。
7. 选举结果通知：服务器1成为leader后，它将选举结果通知给服务器2和服务器3。服务器2和服务器3会更新自己的状态，并认可服务器1作为leader。
8. 集群状态稳定：现在，服务器1成为了leader，而服务器2和服务器3成为了follower。整个集群的状态稳定下来，Leader将开始处理客户端的请求。

请注意，以上是一种可能的选举过程。在实际情况中，选举的结果可能因为网络延迟、服务器响应速度等原因而有所不同。此外，在初始启动期间，如果没有绝大多数的服务器参与选举（如只有两台服务器启动），选举过程可能无法达到一致，集群无法选取到leader。

在ZooKeeper中，选举是通过基于Paxos的Zab协议来实现的，它提供了强一致性和高可用性。选举过程确保了在集群中始终有一个leader节点，以提供可靠的服务和数据一致性。

3、脑裂问题

> 什么是脑裂?

在Elasticsearch、ZooKeeper这些集群环境中，有一个共同的特点，就是它们有一个“大脑”。比如，Elasticsearch集群中有Master节点，ZooKeeper集群中有Leader节点。

集群中的Master或Leader节点往往是通过选举产生的。在网络正常的情况下，可以顺利的选举出Leader（后续以Zookeeper命名为例）。但当**两个机房之间的网络通信出现故障时，选举机制就有可能在不同的网络分区中选出两个Leader。当网络恢复时，这两个Leader该如何处理数据同步？又该听谁的？这也就出现了“脑裂”现象。**

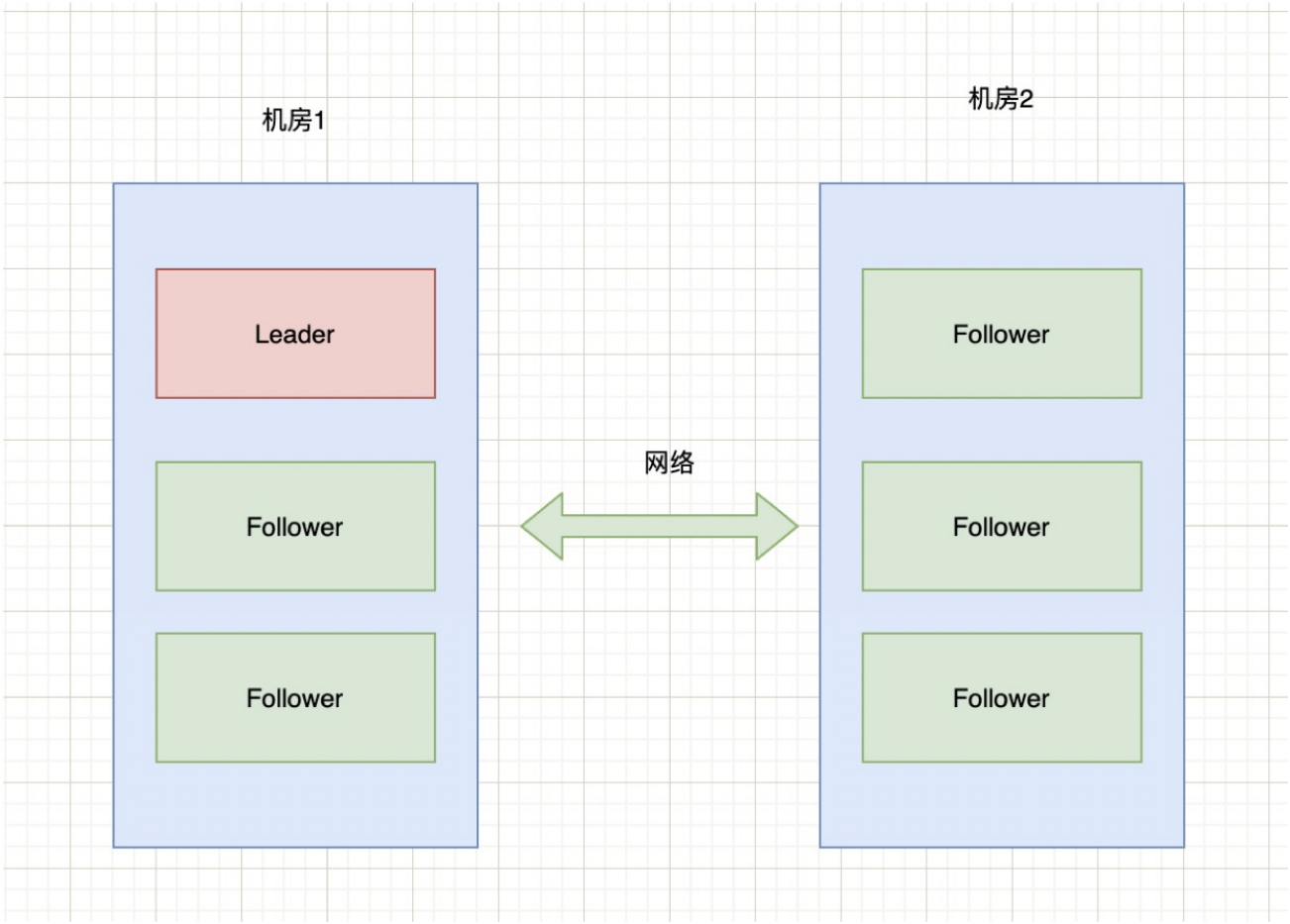
通俗的讲，脑裂(split-brain)就是“大脑分裂”，本来一个“大脑”被拆分成两个或多个。试想，如果一个人有多个脑，且相互独立，就会导致人体“手舞足蹈”，“不听使唤”。

了解了脑裂的基本概念，下面就以zookeeper集群的场景为例，来分析一下脑裂的发生。

> zookeeper集群中的脑裂

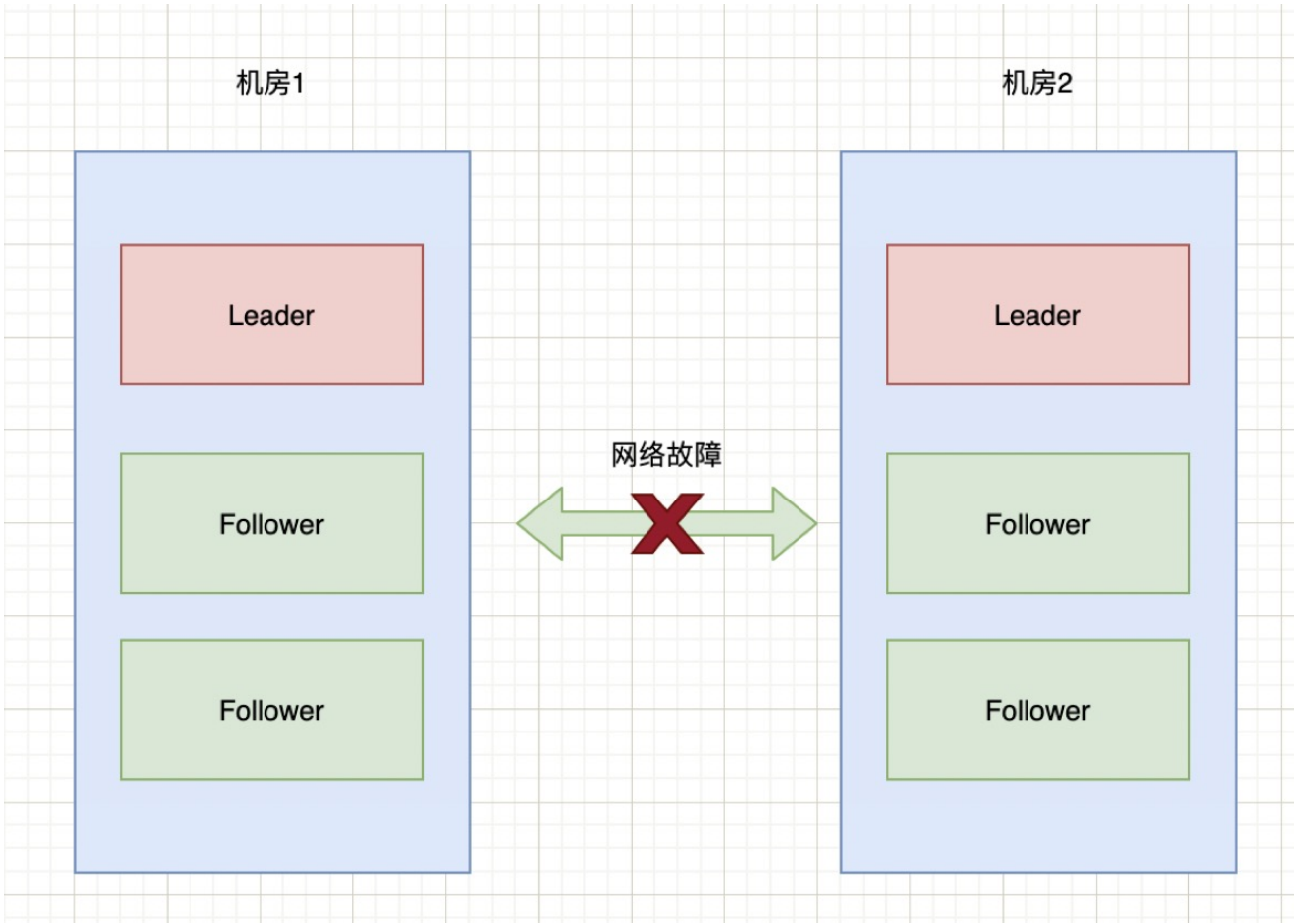
我们在使用zookeeper时，很少遇到脑裂现象，是因为zookeeper已经采取了相应的措施来减少或避免脑裂的发生，这个后面会讲到Zookeeper的具体解决方案。现在呢，先假设zookeeper没有采取这些防止脑裂的措施。在这种情况下，看看脑裂问题是如何发生的。

现有6台zkServer服务组成了一个集群，部署在2个机房：



正常情况下，该集群只有会有个Leader，当Leader宕掉时，其他5个服务会重新选举出一个新的Leader。

如果机房1和机房2之间的网络出现故障，暂时不考虑Zookeeper的过半机制，那么就会出现下图的情况：



也就是说机房2的三台服务检测到没有Leader了，于是开始重新选举，选举出一个新Leader来。原本一个集群，被分成了两个集群，同时出现了两个“大脑”，这就是所谓的“脑裂”现象。

由于原本的一个集群变成了两个，都对外提供服务。一段时间之后，两个集群之间的数据可能会变得不一致了。当网络恢复时，就面临着谁当Leader，数据怎么合并，数据冲突怎么解决等问题。

当然，上面的过程只是我们假设Zookeeper不做任何预防脑裂措施时会出现的问题。那么，针对脑裂问题，Zookeeper是如何进行处理的呢?

> Zookeeper的过半原则

防止脑裂的措施有多种，Zookeeper默认采用的是“过半原则”。所谓的过半原则就是：在Leader选举的过程中，如果某台zkServer获得了超过半数的选票，则此zkServer就可以成为Leader了。

底层源码实现如下：

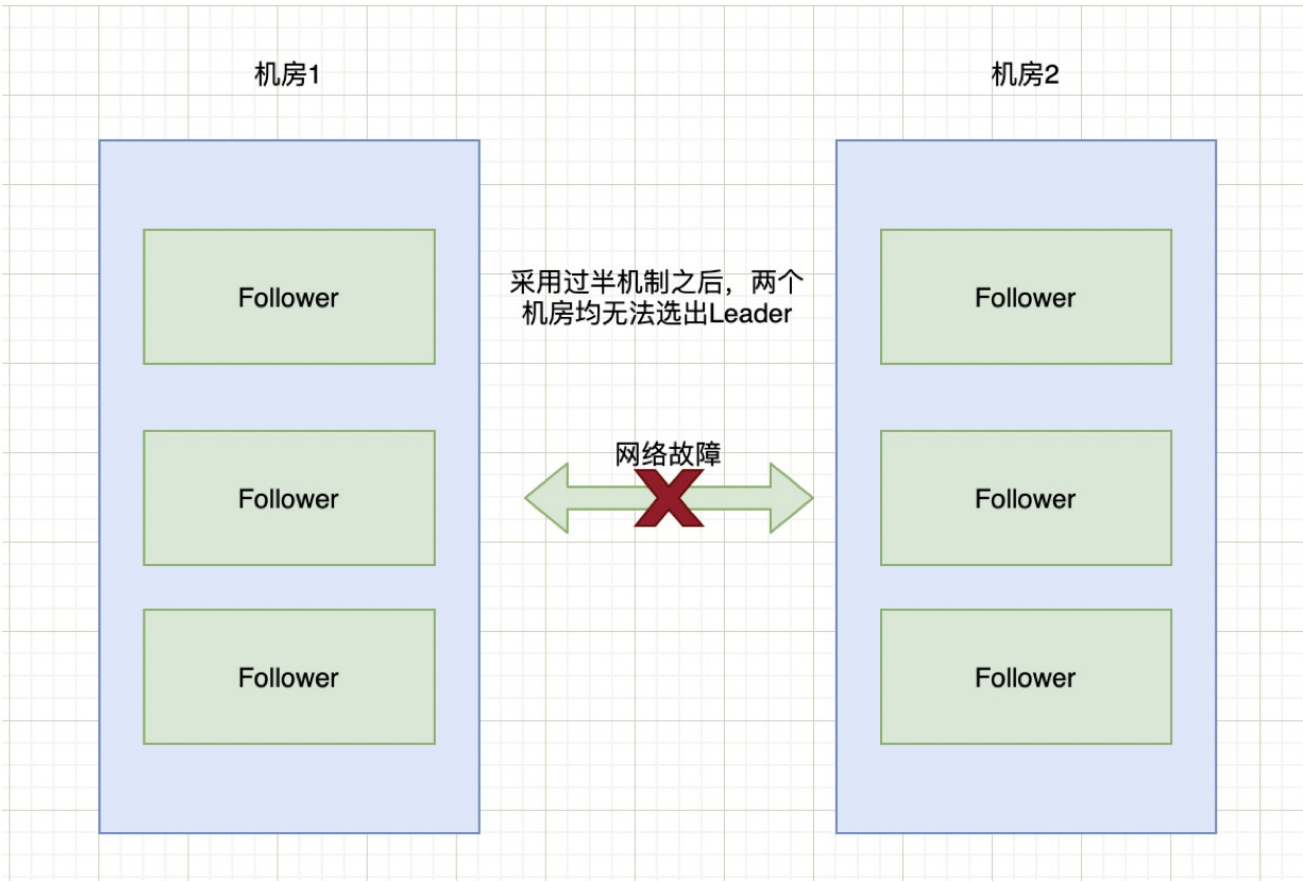
```
public class QuorumMaj implements QuorumVerifier {

    int half;

    // QuorumMaj构造方法。
    // 其中，参数n表示集群中zkServer的个数，不包括观察者节点
    public QuorumMaj(int n){
        this.half = n/2;
    }
```

```
    // 验证是否符合过半机制
    public boolean containsQuorum(Set<Long> set){

        // half是在构造方法里赋值的
        // set.size()表示某台zkServer获得的票数
        return (set.size() > half);
    }
}
```

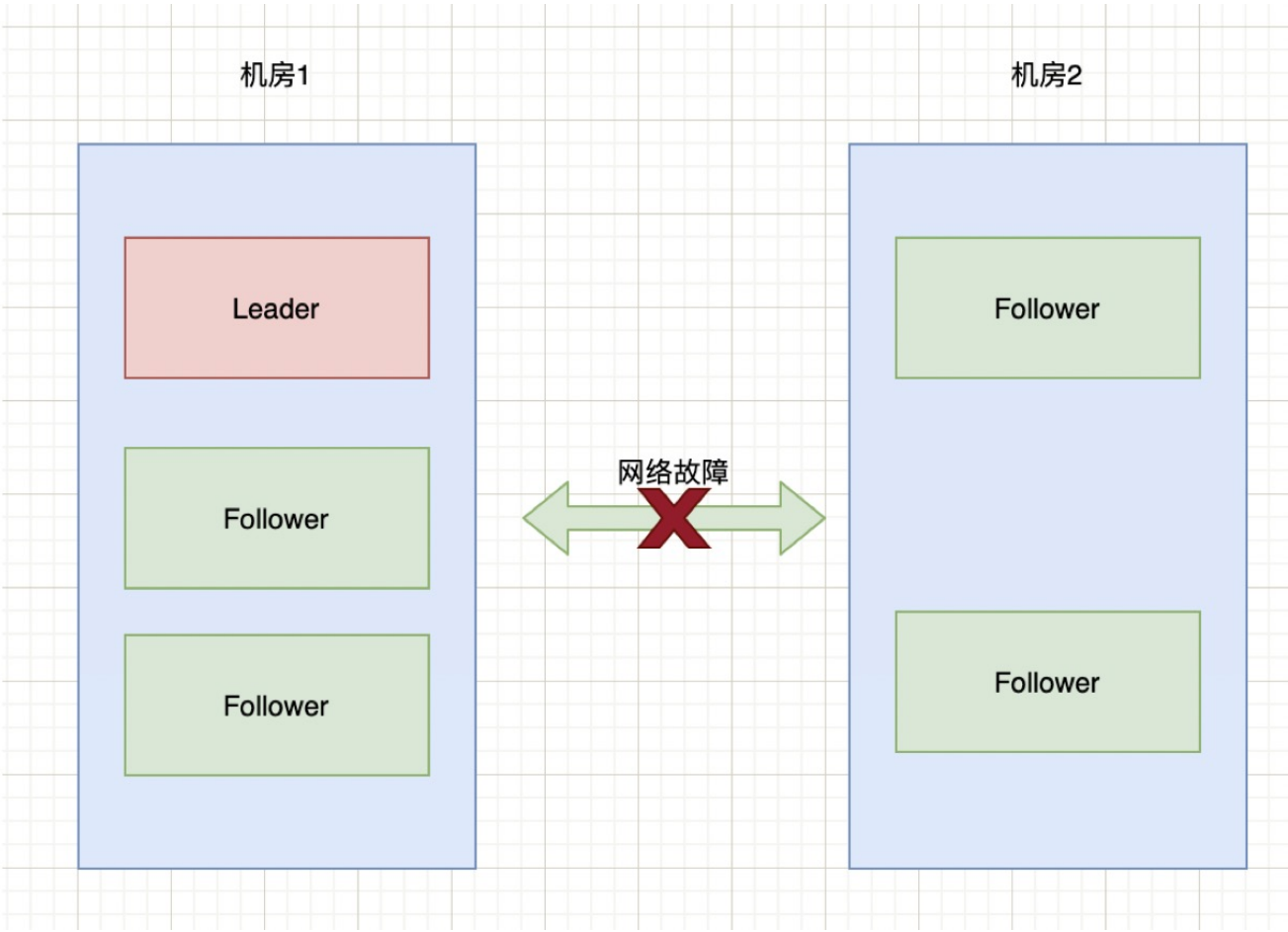
上述代码在构建QuorumMaj对象时，传入了集群中有效节点的个数；containsQuorum方法提供了判断某台zkServer获得的票数是否超过半数，其中set.size表示某台zkServer获得的票数。

上述代码核心点两个：第一，如何计算半数；第二，投票属于半数的比较。

以上图6台服务器为例来进行说明：half = 6 / 2 = 3，也就是说选举的时候，要成为Leader至少要有4台机器投票才能够选举成功。那么，针对上面2个机房断网的情况，由于机房1和机房2都只有3台服务器，根本无法选举出Leader。这种情况下整个集群将没有Leader。

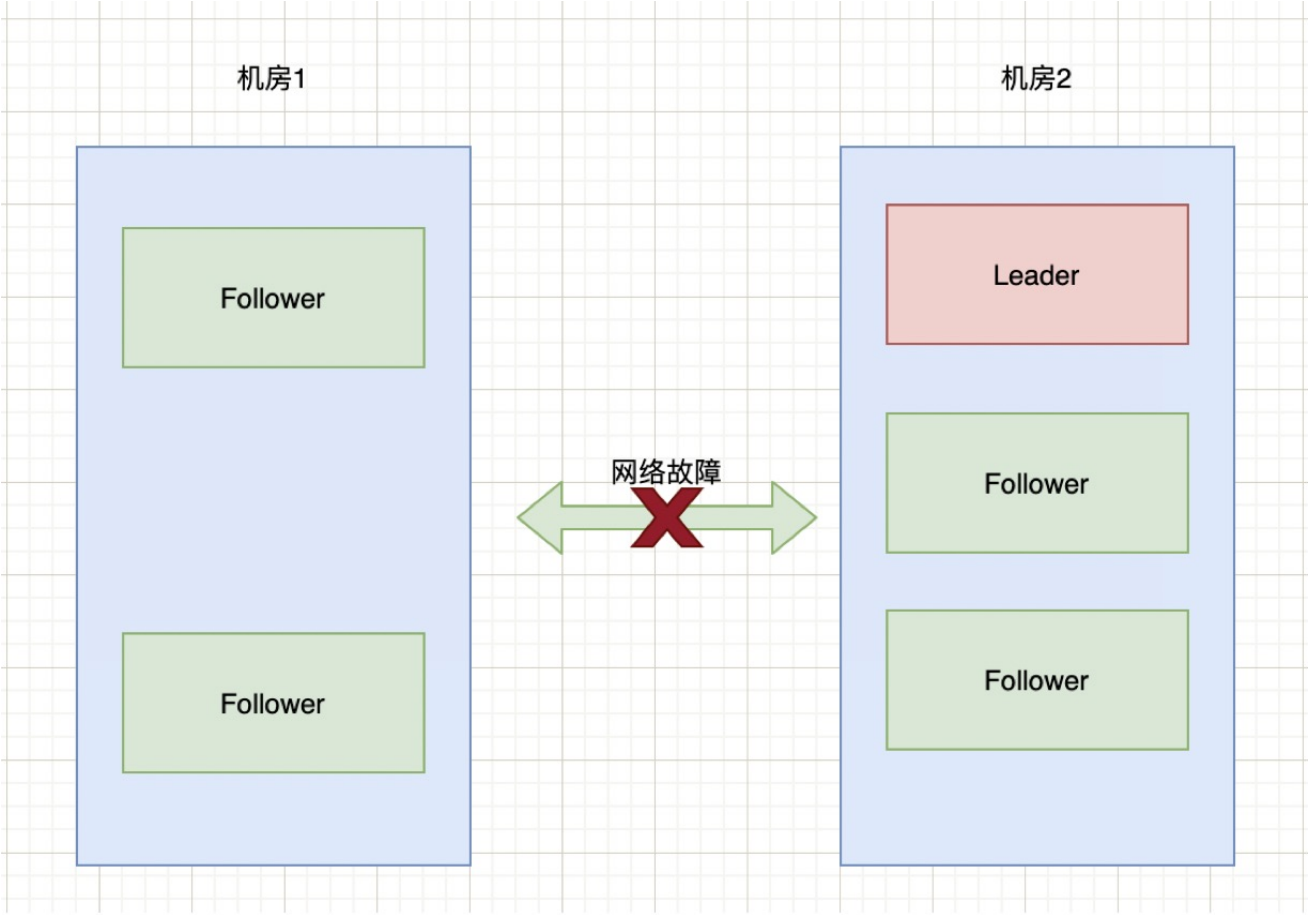


在没有Leader的情况下，会导致Zookeeper无法对外提供服务，所以在设计的时候，我们在集群搭建的时候，要避免这种情况的出现。



如果两个机房的部署请求部署3：3这种状况，而是3：2，也就是机房1中三台服务器，机房2中两台服务器：

在上述情况下，先计算half = 5 / 2 = 2，也就是需要大于2台机器才能选举出Leader。那么此时，对于机房1可以正常选举出Leader。对于机房2来说，由于只有2台服务器，则无法选出Leader。此时整个集群只有一个Leader。



对于上图，颠倒过来也一样，比如机房1只有2台服务器，机房2有三台服务器，当网络断开时，选举情况如下：

Zookeeper集群通过过半机制，达到了要么没有Leader，要没只有1个Leader，这样就避免了脑裂问题。

对于过半机制除了能够防止脑裂，还可以实现快速的选举。因为过半机制不需要等待所有zkServer都投了同一个zkServer就可以选举出一个Leader，所以也叫快速领导者选举算法。

> 新旧Leader争夺

通过过半原则可以防止机房分区时导致脑裂现象，但还有一种情况就是Leader假死。

假设某个Leader假死，其余的followers选举出了一个新的Leader。这时，旧的Leader复活并且仍然认为自己是Leader，向其他followers发出写请求也是会被拒绝的。

因为ZooKeeper维护了一个叫epoch的变量，每当新Leader产生时，会生成一个epoch标号（标识当前属于那个Leader的统治时期），epoch是递增的，followers如果确认了新的Leader存在，知道其epoch，就会拒绝epoch小于现任leader epoch的所有请求。

那有没有follower不知道新的Leader存在呢，有可能，但肯定不是大多数，否则新Leader无法产生。ZooKeeper的写也遵循quorum机制，因此，得不到大多数支持的写是无效的，旧leader即使各种认为自己是Leader，依然没有什么作用。

> ZooKeeper集群节点为什么要部署成奇数

上面讲了过半原则，由于Zookeeper默认采用的就是这种策略，那就带来另外一个问题。集群的数量设置为多少合适呢？而我们所看到的Zookeeper节点数一般都是奇数，这是为什么呢？

首先，只要集群中有过半的机器是正常工作的，那么整个集群就可对外服务。那么我们列举一些情况，来看看在这些情况下集群的容错性。

如果有2个节点，那么只要挂掉1个节点，集群就不可用了。此时，集群对的容忍度为0；

如果有3个节点，那么挂掉1个节点，还有剩下2个正常节点，超过半数，可以重新选举，正常服务。此时，集群的容忍度为1；

如果有4个节点，那么挂掉1个节点，剩下3个，超过半数，可以重新选举。但如果再挂掉1个，只剩下2个，就无法正常选举和服务了。此时，集群的容忍度为1；

依次类推，5个节点，容忍度为2；6个节点容忍度同样为2；

既然3个节点和4个节点、5个节点和6个节点，也就是2n和2n-1的容忍度是一样的，都是n-1。那么，为了节省资源，为了更加高效（更多节点参与选举和通信），为什么不少一个节点呢？这就是为什么集群要部署成奇数的原因。