存储数据结构之 B-tree

极客时间

扫码试看/订阅

《ZooKeeper实战与源码剖析》视频课程

B-tree

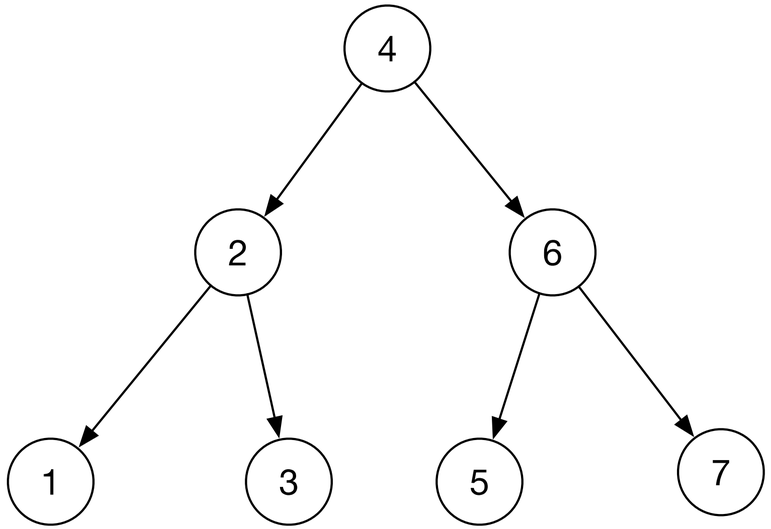
B-tree 的应用十分广泛，尤其是在关系型数据库领域，下面列出了一些知名的 B-tree 存储引擎：

- 关系型数据库系统 Oracle、SQL Server、MySQL 和 PostgreSQL 都支持 B-tree。
- WiredTiger 是 MongoDB 的默认存储引擎，开发语言是 C，支持 B-tree。
- BoltDB：Go 语言开发的 B-tree 存储引擎，etcd 使用 BoltDB 的 fork bbolt。

存储引擎一般用的都是 B+tree，但是存储引擎界不太区分 B-tree 和 B+tree，说 B-tree 的时候其实一般指的是 B+tree。

平衡二叉搜索树

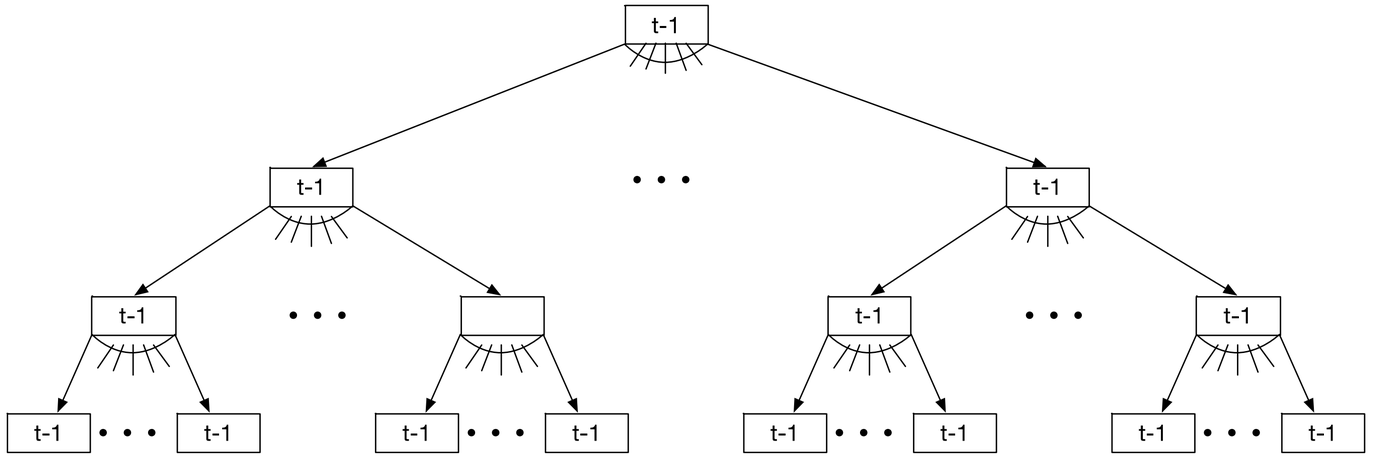
平衡二叉搜索树是用来快速查找 key-value 的有序数据结构。平衡二叉搜索树适用于内存场景，但是不试用于于外部存储。原因在于没访问一个节点都要访问一次外部存储，而访问外部存储是非常耗时的。要减少访问外部存储的次数，就要减少树的高度，要减少树的高度就要增加一个节点保存 key 的个数。B-tree 就是用增加节点中 key 个数的方案来减少对外部存储的访问。



lookup, insertion, and removal:$O(\log n)$

B-tree

B-tree 是一种平衡搜索树。每一个 B-tree 有一个参数 t，叫做 minimum degree。每一个节点的 degree 在 t 和 2t 之间。下图是一个每个节点的 degree 都为 t 的 B-tree。如果 t 为 1024 的话，下面的 B-tree 可以保存 1G 多的 key 值。因为 B-tree 的内部节点通常可以缓存在内存中，访问一个 key 只需要访问一次外部存储。



depth	#node
0	1
1	t
2	t^2
3	t^3

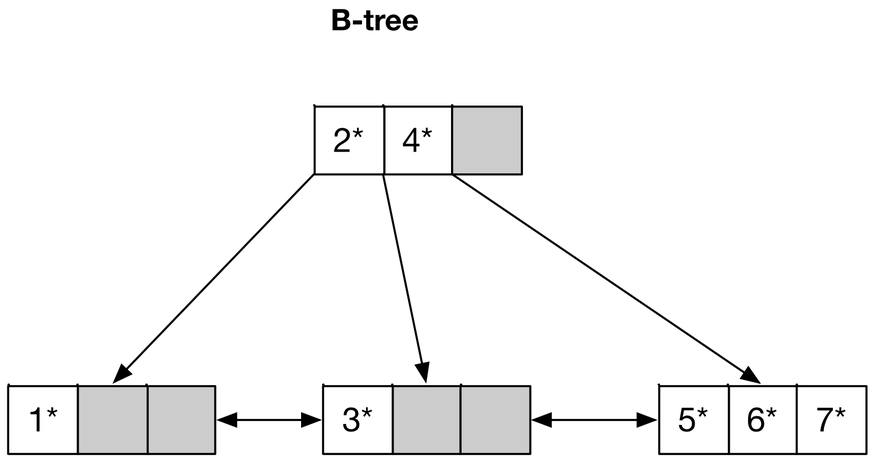
B-tree 和平衡二叉搜索树的算法复杂度一样的，但是减少了对外部存储的访问次数。

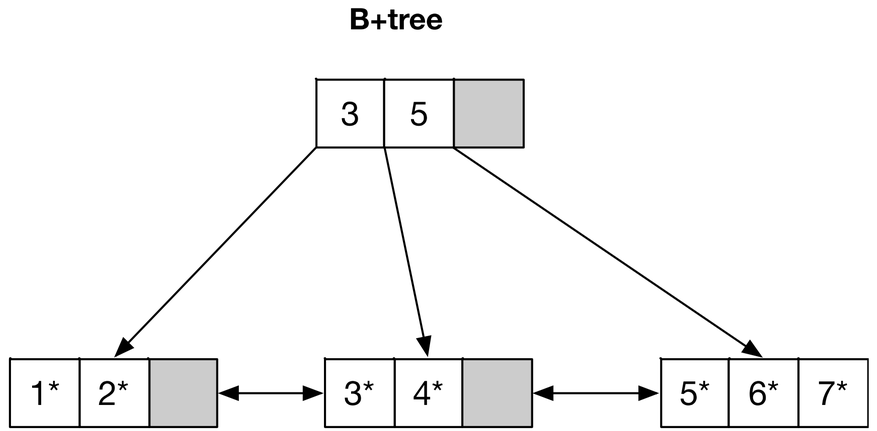
B-tree 特点

- 所有的节点添加都是通过节点分裂完成的。
- 所有的节点删除都是通过节点合并完成。
- 所有的插入都发生在叶子节点。
- B-tree 的节点的大小通常是文件系统 block 大小的倍数，例如 4k，8k 和 16k。

B+tree

为了让 B-tree 的内部节点可以具有更大的 degree，可以规定内部节点只保存 key，不保存 value。这样的 B-tree 叫作 B+tree。另外通常会把叶子节点连成一个双向链表，方便 key-value 升序和降序扫描。

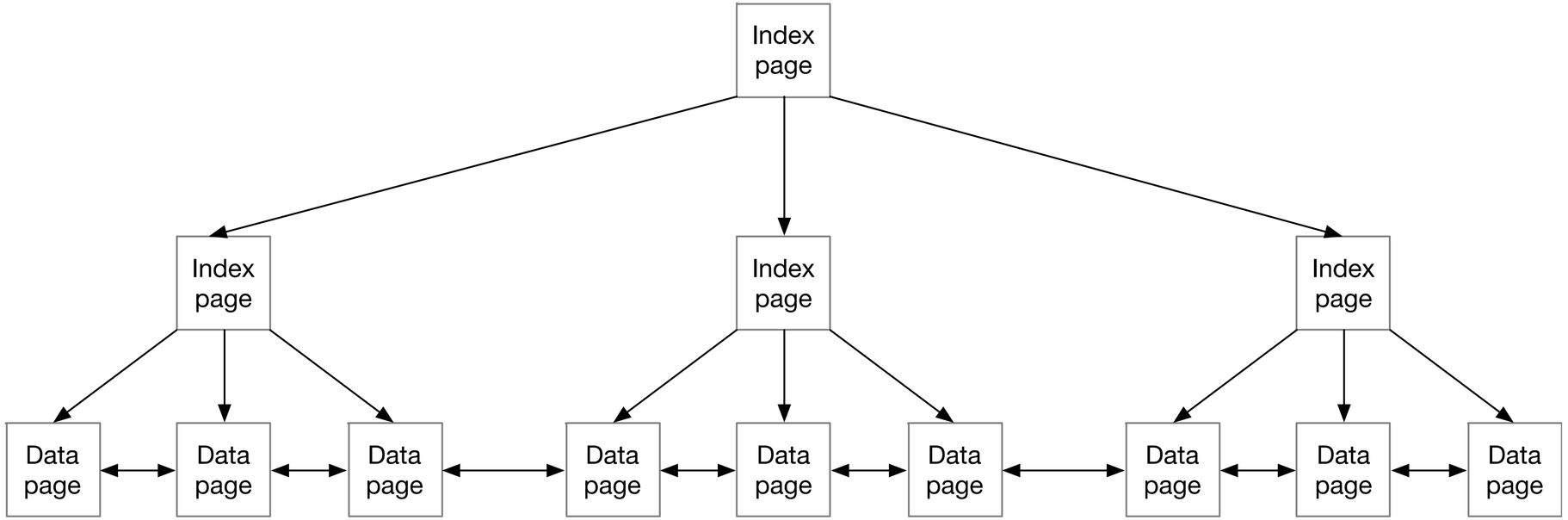




1*代表key为1的key-value

B+tree 索引

大部分关系型数据库表的主索引都是用的 B+tree。B+tree 的叶子节点叫作 data page，内部节点叫作 index page。



存储数据结构之 LSM

Log Structured Merge-tree（LSM）

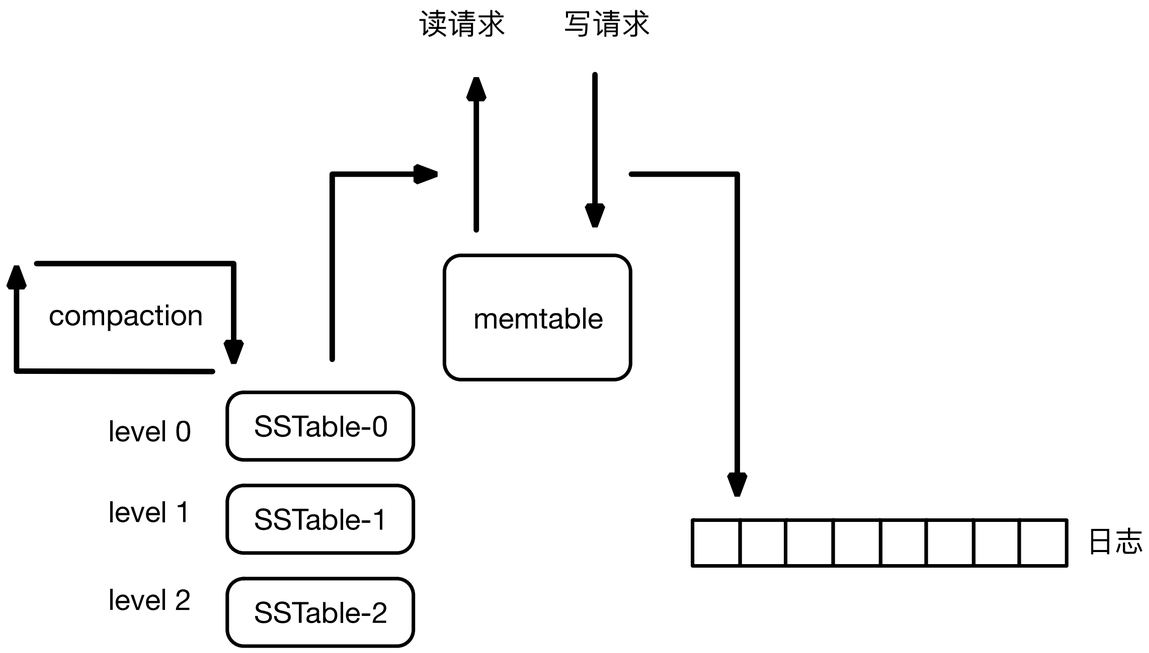
LSM 是另外一种广泛使用的存储引擎数据结构。LSM 是在 1996 发明的，但是到了 2006 年从 Bigtable 开始才受到关注。

LSM 架构

一个基于 LSM 的存储引擎有以下 3 部分组成：

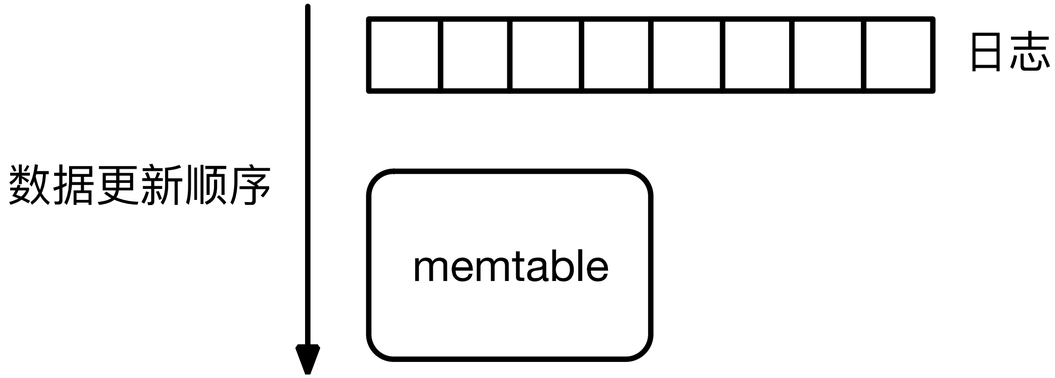
- Memtable：保存有序 KV 对的内存缓冲区。
- 多个 SSTable：保存有序 KV 对的只读文件。
- 日志：事务日志。

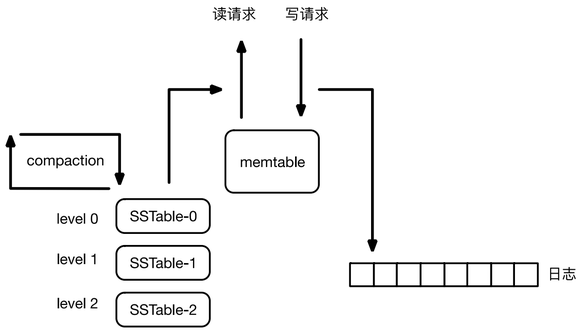
LSM 存储 MVCC 的 key-value。每次更新一个 key-value 都会生成一个新版本，删除一个 key-value 会生成一个 tombstone 的新版本。



LSM 写操作

一个写操作首先在日志中追加事务日志，然后把新的 key-value 更新到 Memtable。LSM 的事务是 WAL 日志。

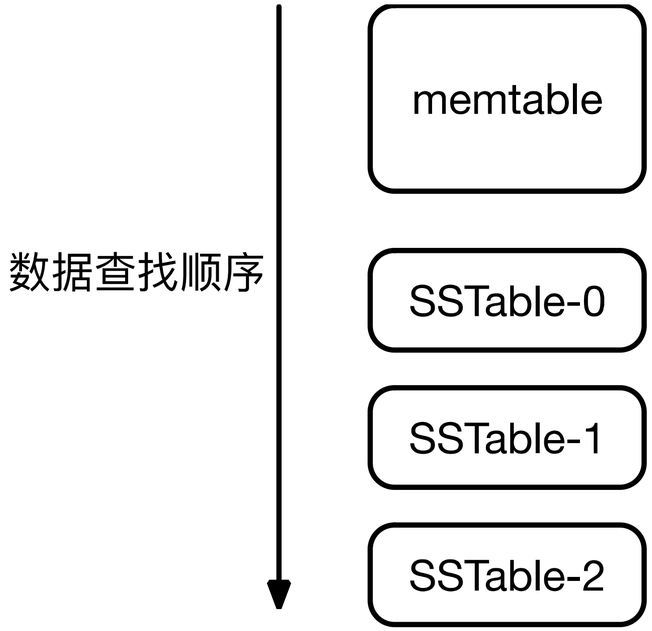


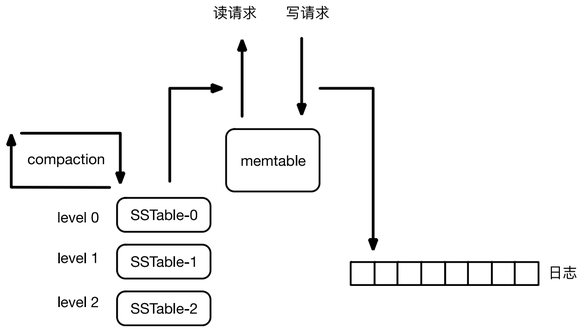


LSM 读操作

在由 Memtable 和 SSTable 合并成的一个有序 KV 视图上进行 Key 值的查找。例如在右图所示的 LSM 中，要查找一个 key a，

1. 在 memtable 中查找，如果查找到，返回。否则继续。
2. 在 SSTable-0 中查找，如果查找到，返回。否则继续。
3. 在 SSTable-1 中查找，如果查找到，返回。否则继续。
4. 在 SSTable-2 中查找，如果查找到，返回。否则返回空值。



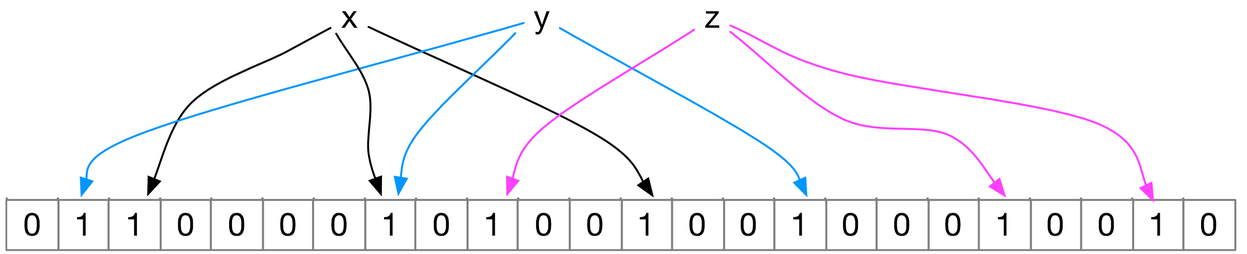


Bloom Filter

使用 Bloom filter 来提升 LSM 数据读取的性能。Bloom filter 是一种随机数据结构，可以在 O(1) 时间内判断一个给定的元素是否在集合中。False positive 是可能的，既 Bloom filter 判断在集合中的元素有可能实际不在集合中，但是 false negative 是不可能的。Bloom filter 由一个 m 位的位向量和k个相互独立的哈希函数 h_1，h_2，…，h_k 构成。这些hash函数的值范围是 {1，…，m}。初始化 Bloom filter 的时候把位向量的所有的位都置为 0。 添加元素 a 到集合的时候，把维向量 $h_1(a)$， $h_2(a)$， $h_k(a)$ 位置上的位置为 1。判断一个元素 b 是否在集合中的时候，检查把维向量 $h_1(b)$， $h_2(b)$， … ，$h_k(a)$ 位置上的位是否都为 1。如果这些位都为 1，那么认为 b 在集合中；否则认为 b 不在集合之中。下图所示的是一个 m 为 14，k 为 3 的 Bloom filter。下面是计算 False positive 概率的公式（n 是添加过的元素数量）：

$$\text{False positive probability} \approx (1 - (1 - \frac{1}{m})^{kn})^k$$

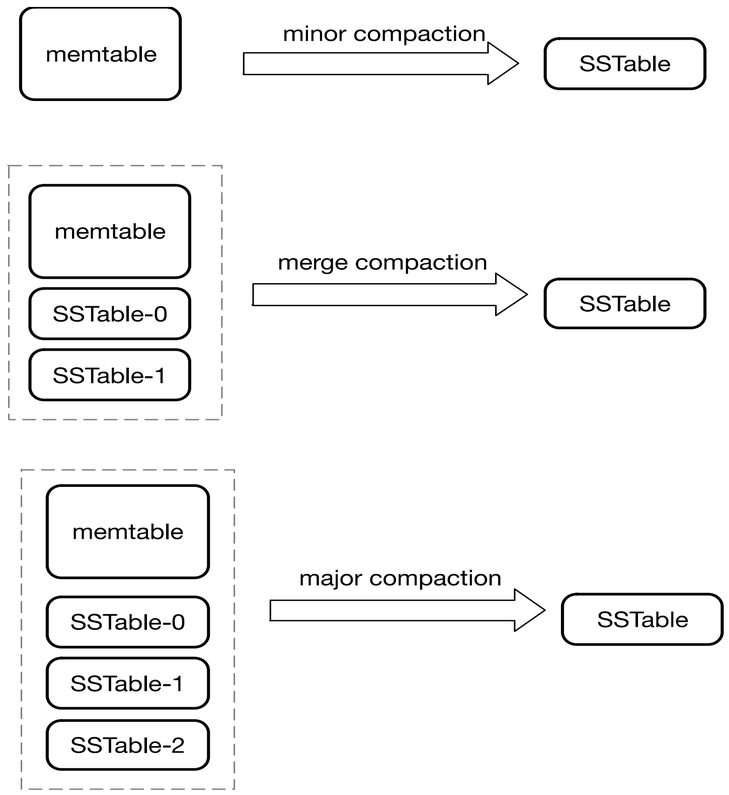
下图所示的是一个 m 为 14，k 为 3 的 Bloom filter。



Compaction

如果我们一直对 memtable 进行写入，memtable 就会一直增大直到超出服务器的内部限制。所以我们需要把 memtable 的内存数据放到 durable storage 上去，生成 SSTable 文件，这叫做 minor compaction。

- Minor compaction：把 memtable 的内容写到一个 SSTable。目的是减少内存消耗，另外减少数据恢复时需要从日志读取的数据量。
- Merge compaction：把几个连续 level 的 SSTable 和 memtable 合并成一个 SSTable。目的是减少读操作要读取的 SSTable 数量。
- Major compaction：合并所有 level 上的 SSTable 的 merge compaction。目的在于彻底删除 tombstone 数据，并释放存储空间。



基于 LSM 的存储引擎

下面列出了几个知名的基于 LSM 的存储引擎：

- LevelDB：开发语言是 C++，Chrome 的 IndexedDB 使用的是 LevelDB。
- RocksDB：开发语言是 C++，RocksDB 功能丰富，应用十分广泛，例如 CockroachDB、TiKV 和 Kafka Streams 都使用了它。
- Pebble：开发语言是 Go，应用于 CockroachDB。
- BadgerDB：一种分离存储 key 和 value 的 LSM 存储引擎。
- WiredTiger：WiredTiger 除了支持 B-tree 以外，还支持 LSM。

存储引擎的放大指标（Amplification Factors）

- 读放大（read amplification）：一个查询涉及的外部存储读操作次数。如果我们查询一个数据需要做 3 次外部存储读取，那么读放大就是 3。
- 写放大（write amplification）：写入外部存储设备的数据量和写入数据库的数据量的比率。如果我们对数据库写入了 10MB 数据，但是对外部存储设备写入了 20BM 数据，写放大就是 2。
- 空间放大（space amplification）：数据库占用的外部存储量和数据库本身的数据量的比率。如果一个 10MB 的数据库占用了 100MB，那么空间放大就是 10。

比较 B-tree 和 LSM

LSM 和 B-tree 在 Read amplification（读放大），Write amplification（写放大）和 Space amplification（空间放大）这个三个指标上的区别:

	LSM	B+-Tree
读放大	一个读操作要对多个 level 上的 SSTable 进行读操作。	一个 key-value 的写操作涉及一个数据页的读操作，若干个索引页的读操作。
写放大	一个 key-value 值的写操作要在多级的 SSTable 上进行。	一个 key-value 的写操作涉及数据页的写操作，若干个索引页的写操作。
空间放大	在 SSTable 中存储一个 key-value 的多个版本。	索引页和页 fragmentation。

LSM 和 B+-Tree 在性能上的比较：

- 写操作：LSM 上的一个写操作涉及对日志的追加操作和对 memtable 的更新。但是在 B+-Tree 上面，一个写操作对若干个索引页和一个数据页进行读写操作，可能导致多次的随机 IO。所以 LSM 的写操作性能一般要比 B+-Tree 的写操作性能好。

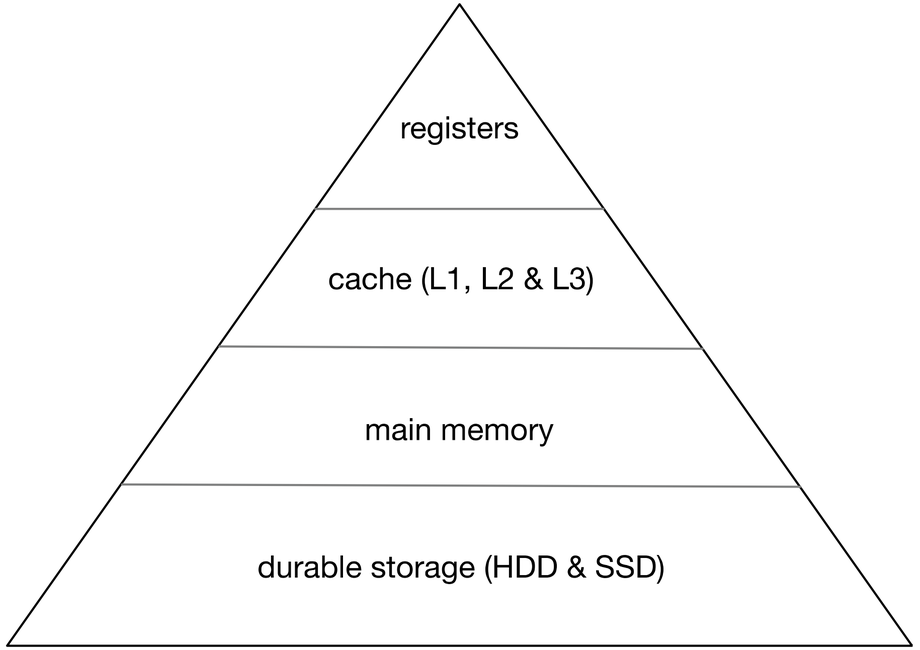
- 读操作：LSM 上的一个读操作需要对所有 SSTable 的内容和 memtable 的内容进行合并 。但是在 B+-Tree 上面，一个读操作对若干个索引页和一个数据页进行读操作 。所以 B+-Tree 的读操作性能一般要比 LSM 的读操作性能好。

本地存储技术总结

数据的随机读写 vs 顺序读写

在右图的 memory hierarchy，越往下的存储方式容量越大延迟越大，越往上容量越小延迟越小。对 main memory 和 durable storage 的数据访问，顺序读写的效率都要比随机读写高。例如 HDD 的 seek time 通常在 3 到 9ms 之前，所以一个 HDD 一秒最多支持 300 多次随机读写。虽然 SSD 和 main memory 的随机读写效率要比 HDD 好的多，顺序读写的效率仍然要比随机读写高。

所以我们设计存储系统的时候，要尽量避免随机读写多使用顺序读写。

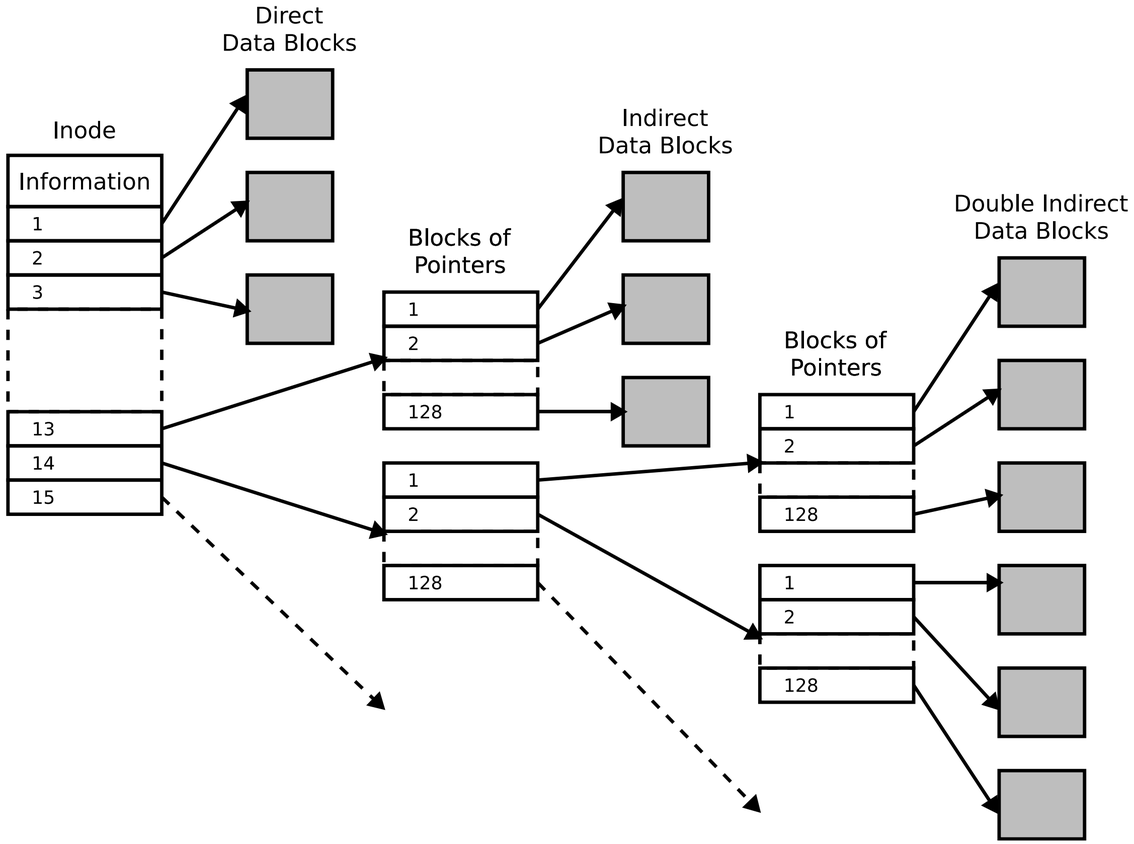


The memory hierarchy

文件系统基础知识

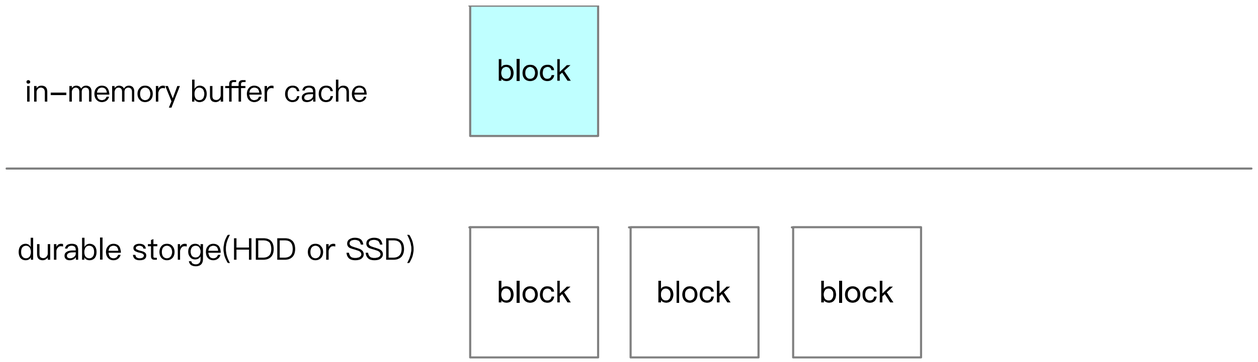
ext4 文件系统

ext4 是 Linux 系统上广泛使用的文件系统。下图列的是 ext4 文件系统 inode 的结构。其中 information 包括文件的 size，last access time 和 last modification time 等。文件的 inode 和 data block 存储在存储设备的不同位置。



文件系统 API

访问文件内容是 read 和 write 两个系统调用。除非使用了 O_DIRECT 选项，read 和 write 操作的都是 block 的buffer cache，Linux OS 会定期把 dirty block 刷新到 durable storage 上去。



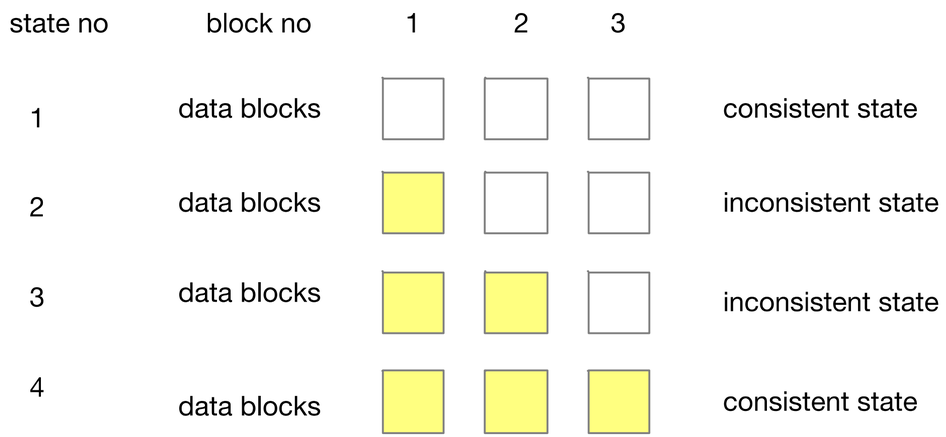
设计可靠的存储系统要求把内容实际写到 durable storage 上去。下面的这两个系统调用提供了把 buffer cache 的内容手动刷新到 durable storage 的机制：

- fsync：把文件的数据 block 和 inode 的 metadata 刷新到 durable storage。
- fdatasync：把把文件的数据 block 刷新到 durable storage。只有修改过的 metadata 影响后面的操作才把 metadata 也刷新到 durable storage。

Write Ahead Logging

如何保证 durable storage 写入的原子性

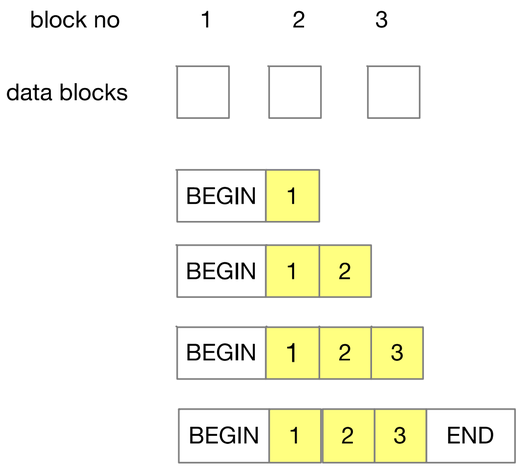
我们在 write 调用之后调用 fsync/fdatasync，文件系统通常可以保证对一个 block 写入的原子性。如果我们的一个数据写入包含对多个 block 的写入。要保证这样整个写入的原子性，就需要另外的机制。



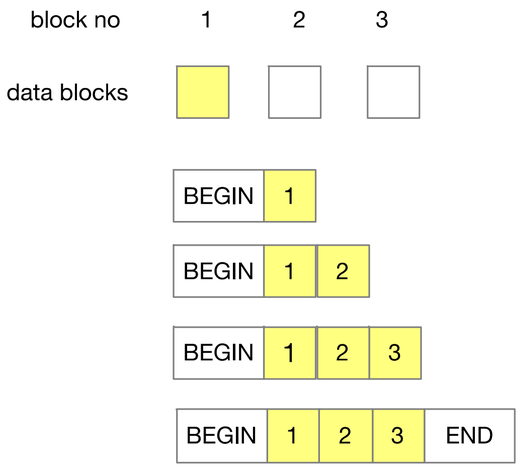
例如在上图中，我们要对 3 个 data block 进行写入。如果我们依次对这些 block 写入，如果在写入 block1 之后发生 crash，数据就会处于状态 2。在状态 2中，block1 保存旧数据、block2 和 block3 保存旧数据，数据是不一致的。

Write Ahead Logging（WAL）

WAL 是广泛使用的保证多 block 数据写入原子性的技术。WAL 就是在对 block 进行写入之前，先把新的数据写到一个日志。只有在写入 END 日志并调用 sync API，才开始对 block 进行写入。如果在对 block 进行写入的任何时候发生 crash，都可以在重启的使用 WAL 里面的数据完成 block 的写入。



完成END的日志写入是整个数据写入的commit point

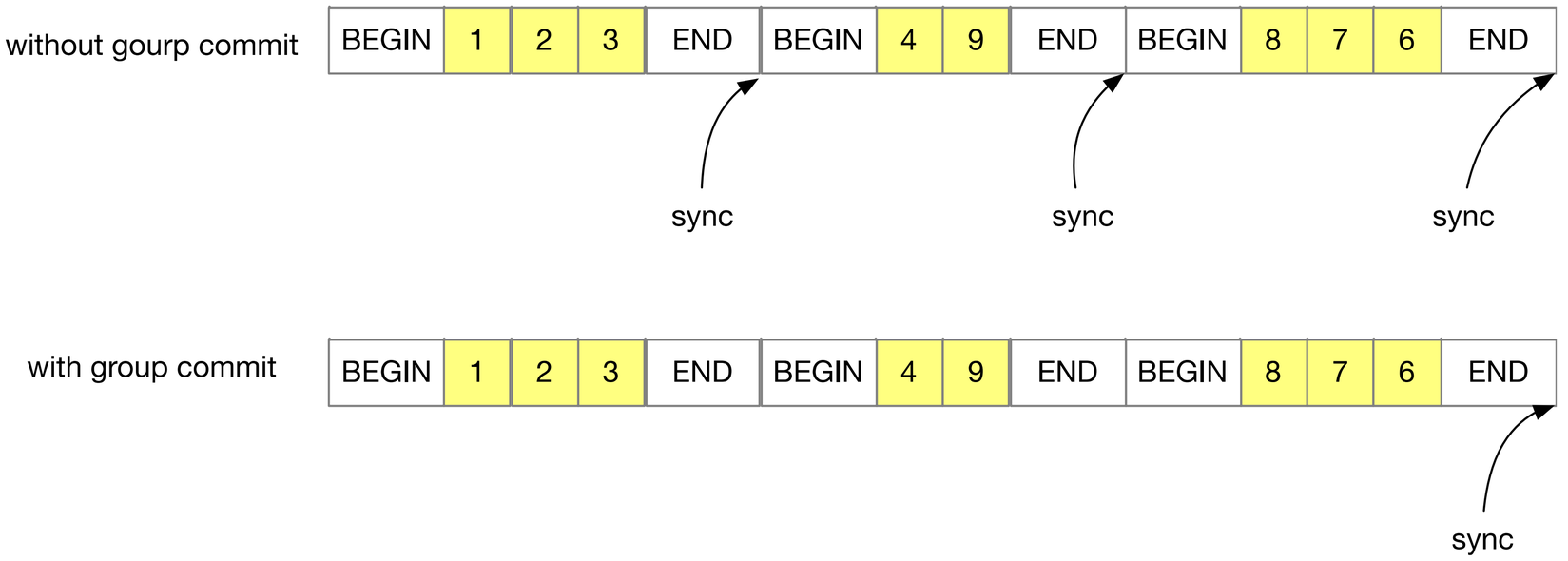


完成block 1写入之后，发生crash

另外通过使用 WAL，我们在提交一个操作之前只需要进行文件的顺序写入，从而减少了包含多 block 文件操作的数据写入时延。

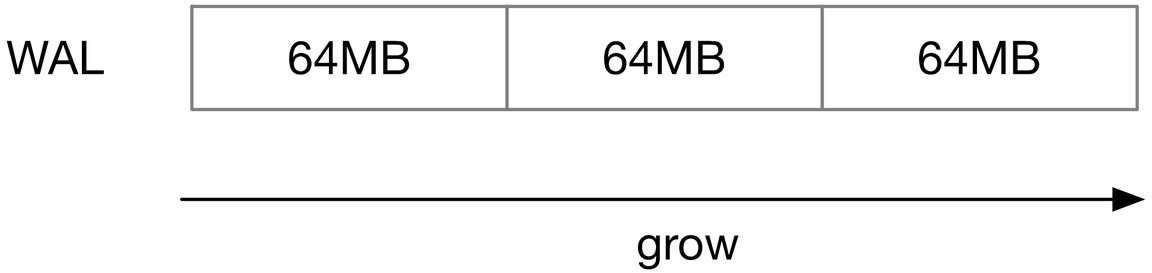
WAL 优化1：Group Commit

上面的 WAL 方案中每次写入完 END 日志都要调用一次耗时的 sync API，会影响系统的性能。为了解决这个问题，我们可以使用 group commit。group commit 就是一次提交多个数据写入，只有在写入最后一个数据写入的 END日志之后，才调用一次 sync API。



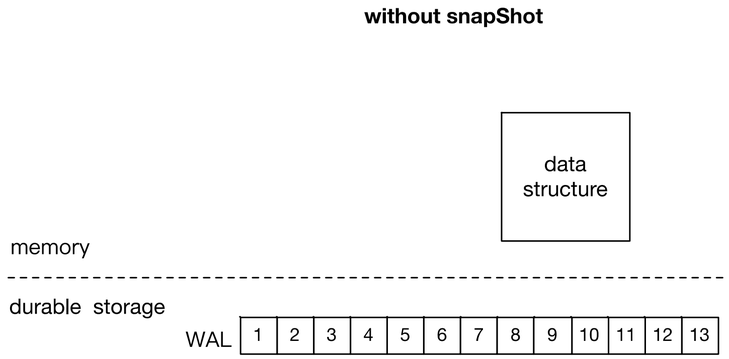
WAL 优化2： File Padding

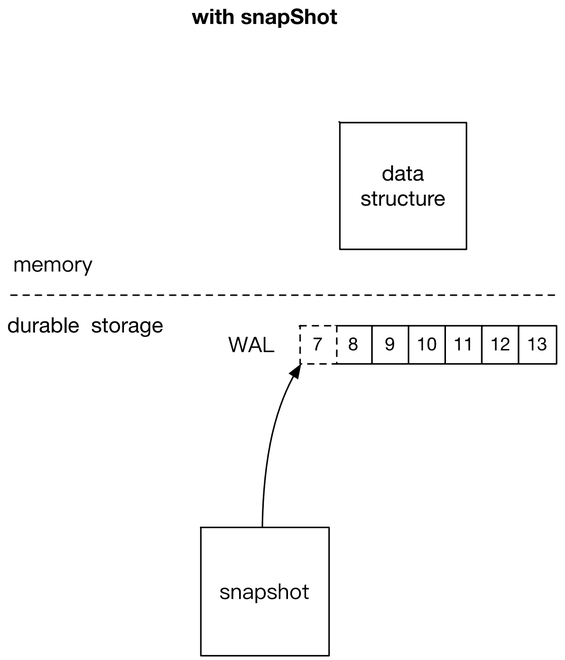
在往 WAL 里面追加日志的时候，如果当前的文件 block 不能保存新添加的日志，就要为文件分配新的 block，这要更新文件 inode 里面的信息（例如 size）。如果我们使用的是 HHD 的话，就要先 seek 到 inode 所在的位置，然后回到新添加 block 的位置进行日志追加。为了减少这些 seek，我们可以预先为 WAL 分配 block。例如 ZooKeeper 就是每次为 WAL 分配 64MB 的 block。



WAL 优化3：快照

如果我们使用一个内存数据结构加 WAL 的存储方案，WAL 就会一直增长。这样在存储系统启动的时候，就要读取大量的 WAL 日志数据来重建内存数据。快照可以解决这个问题。快照是应用 WAL 中从头到某一个日志条目产生的内存数据结构的序列化，例如下图中的快照就是应用从 1 到 7 日志条目产生的。





除了解决启动时间过长的问题之外，快照还可以减少存储空间的使用。WAL 的多个日志条目有可能是对同一个数据的改动，通过快照，就可以只保留最新的数据改动。

数据序列化

现在有众多的数据序列化方案，下面列出一些比较有影响力的序列化方案：

- JSON：基于文本的序列化方案，方便易用，没有 schema，但是序列化的效率低，广泛应用于 HTTP API 中。
- BSON：二进制的 JSON 序列化方案，应用于 MongoDB。
- Protobuf：Google 研发的二进制的序列化方案，有 schema，广泛应用于 Google 内部，在开源界也有广泛的应用（例如 gRPC）。
- Thrift：Facebook 研发的和 Protobuf 类似的一种二进制序列化方案，是 Apache 的项目。
- Avro：二进制的序列化方案，Apache 项目，在大数据领域用的比较多。

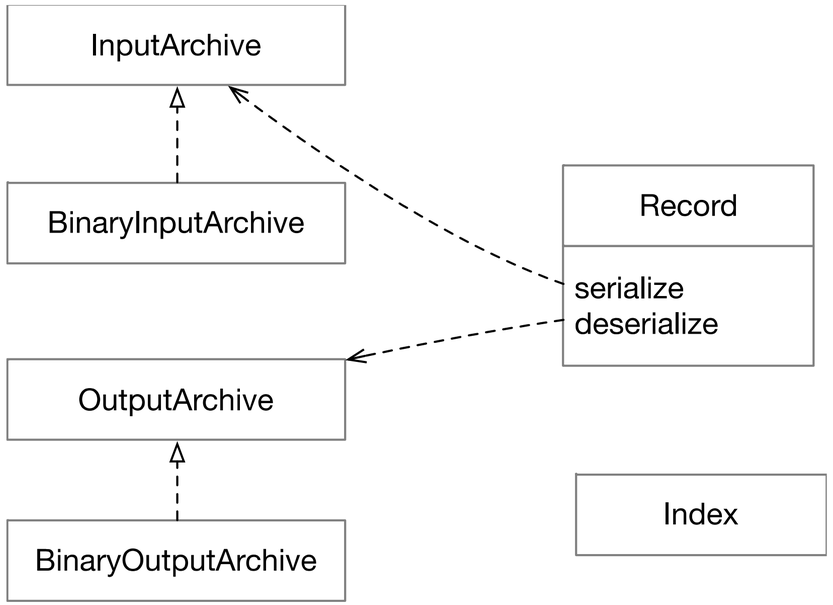
如何研发本地存储

研发一个高效的本地存储引擎需要该领域的专家级技术，所以不建议自己从 0 开始研发。目前开源界有众多的本地存储引擎，建议直接使用现有的方案。如果现有的开源方案不能满足要求，可以在这些方案的基础之上进行二次开发。

ZooKeeper本地存储源码解析

序列化

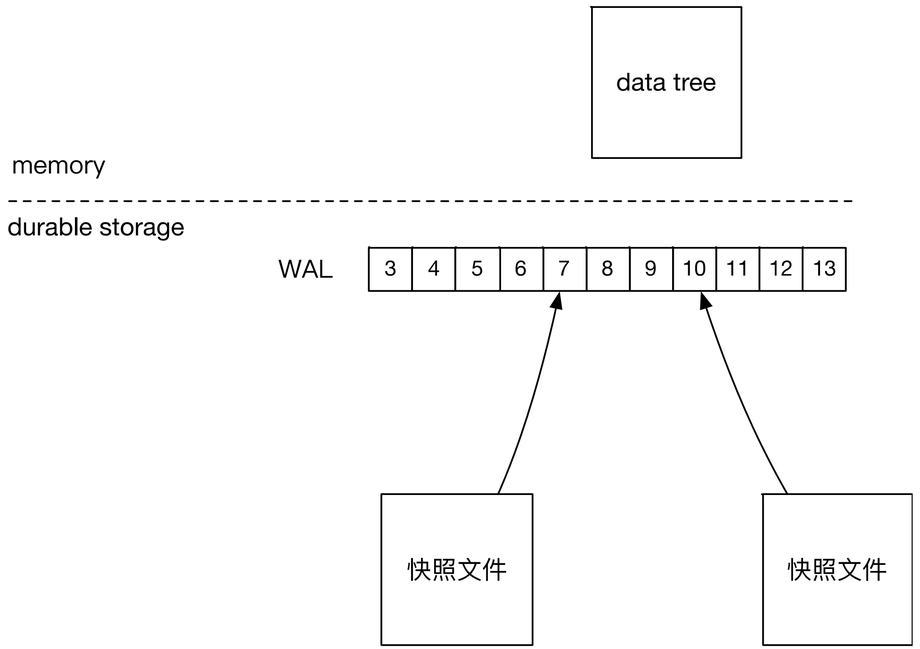
ZooKeeper 使用的序列化方案是 Apache Jute。ZooKeeper.jute 包含所有数据的 schema， Jute 编译器通过编译 jute 文件生成 Java 代码。生成的所有 Java 类实现 Record 接口。下面列出了序列化的核心接口和类。 Jute 的序列化底层使用的是 Java DataInput 的编码方案。



序列化代码展示

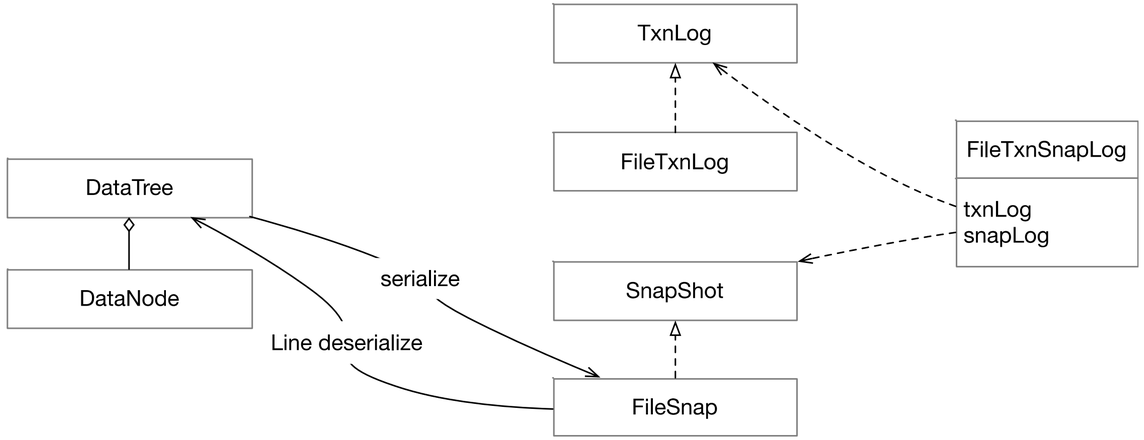
本地存储架构

ZooKeeper 的本地存储采用的是内存数据结构加 WAL 的方案。 ZooKeeper 的 WAL 叫作事务日志 (transaction log).



核心接口和类

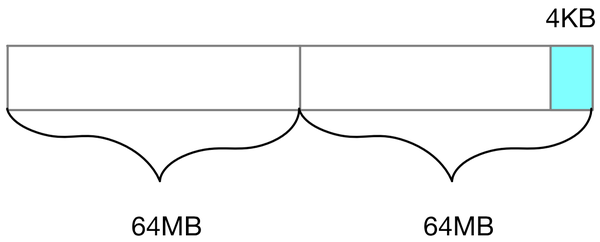
- TxnLog: 接口类型，提供读写事务日志的API。
- FileTxnLog: 基于文件的 TxnLog 实现。
- Snapshot: 快照接口类型，提供序列化、反序列化、访问快照的 API。
- FileSnap: 基于文件的 Snapsho 实现。
- FileTxnSnapLog: TxnLog和 SnapSho t的封装。
- DataTree: ZooKeeper 的内存数据结构，是有所有 znode 构成的树。
- DataNode: 表示一个 znode。



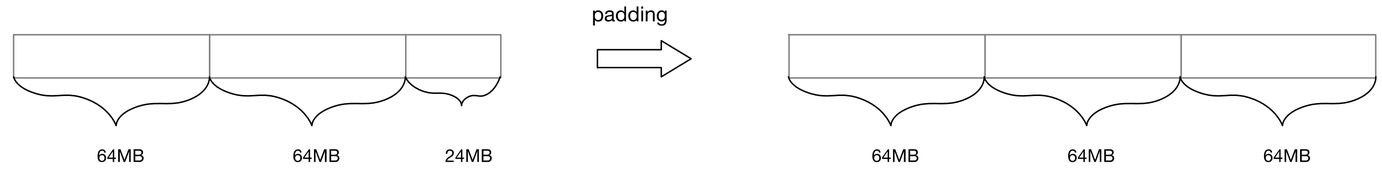
WAL

File Padding

当 file channel 的 position 在 currentSize 结束为止的 4KB 范围之内是进行 padding。



如果 position 已经超出了 currentSize，基于 position 进行空间扩容。



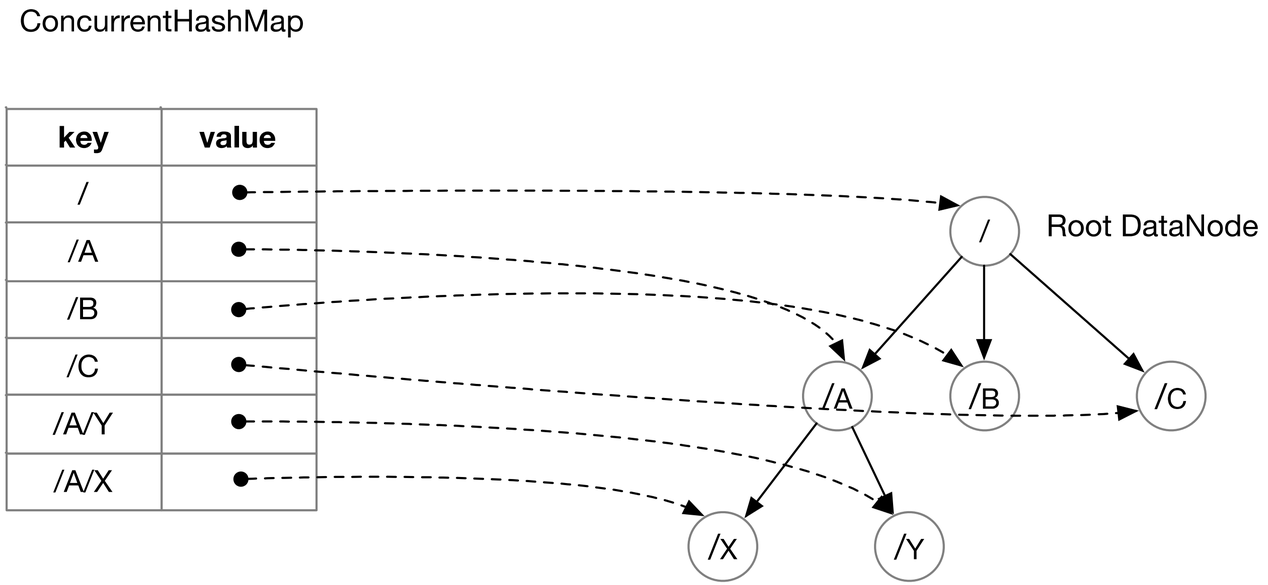
position: 64+64+24=152MB

currentSize: 64+64=128MB

Group Commit

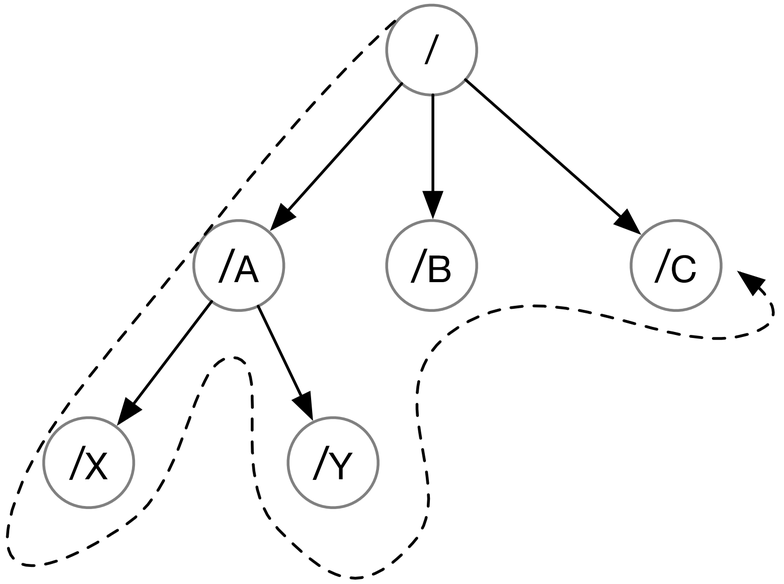
DataTree

DataNode 有一个成员叫作 children，children 保存该 DataNode 的子节点名字，可以从根节点开始通过 children 遍历所有的节点。只有在序列化 DataTree 的时候才会通过 children 进行 DataTree 的遍历。其他对 DataNode 的访问都是通过 DataTree 的成员 nodes 来进行的。nodes 是一个 ConcurrentHashMap，保存的是 DataNode 的 path 到 DataNode 的映射。



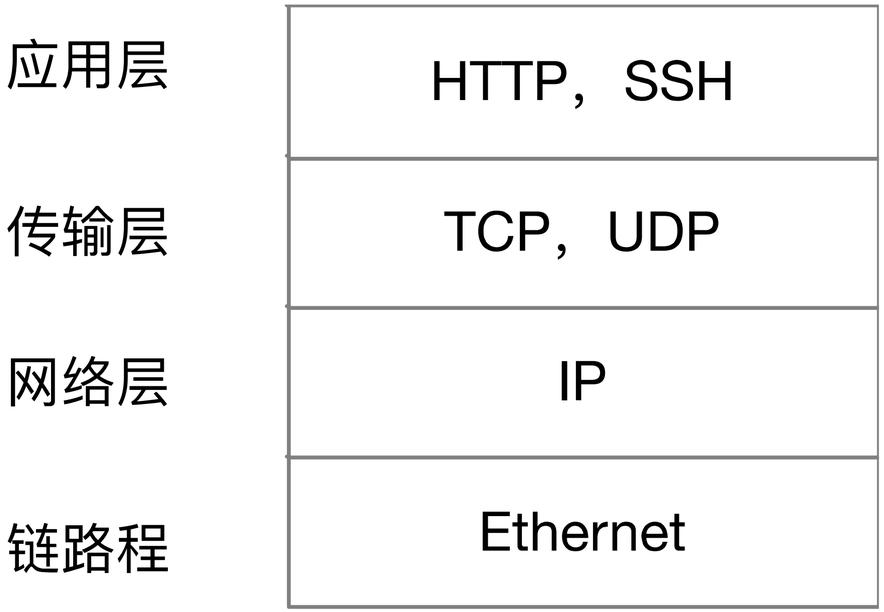
快照

- 序列化：从根 DataNode 开始做前序遍历，依次把 DataNode 写入到快照文件中。
- 反序列化：从头开始顺序读取快照文件的内容，建立 DataTree。因为在序列化的时候使用的是前序遍历，会先反序列化到父亲节点再反序列化孩子节点。因此，在创建新的 DataNode 的同时，可以把新的 DataNode 加到它的父亲节点的 children 中去。



网络编程基础

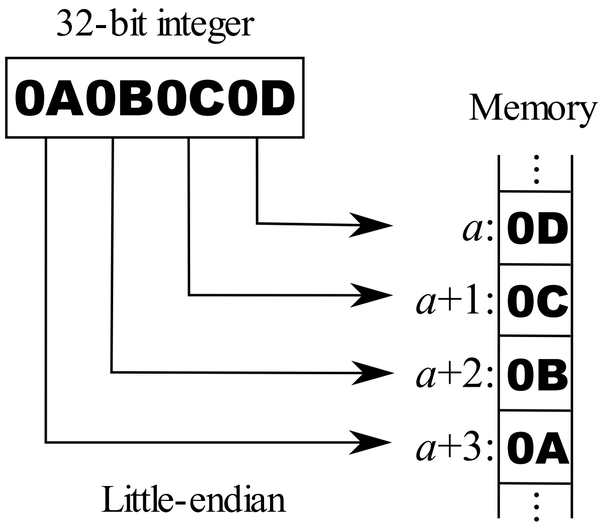
TCP/IP 协议栈

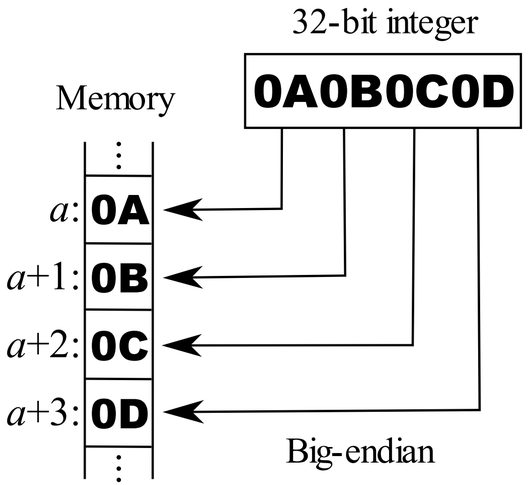


在后面的讨论中我们使用的传输层协议就是 TCP。

Endianness

Java使用的是big-endian，x86使用的是little-endian，TCP/IP使用的是big-endian。

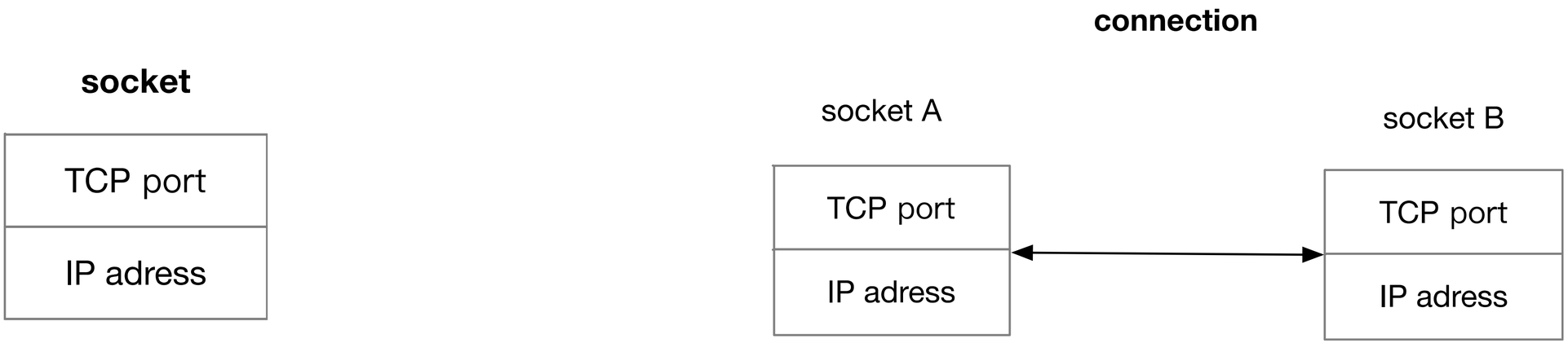




TCP socket和connection

socket：用来表示网络中接收和发送数据的一个 endpoint，由 IP 地址和 TCP 端口号组成。

connection：表示两个 endpoint 之间进行数据转述的一个通道，由代表两个 endpoint 的 socket 组成。

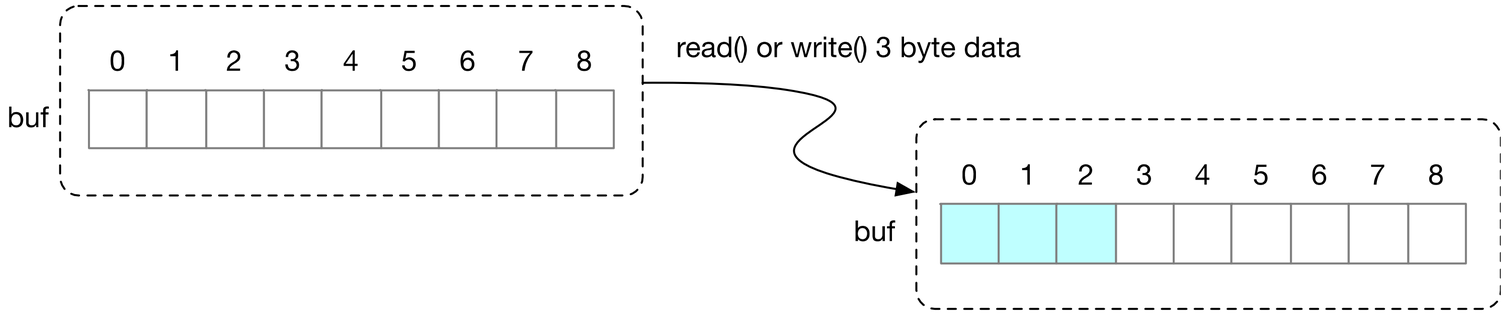


Socket编程API

发送和接受数据API：

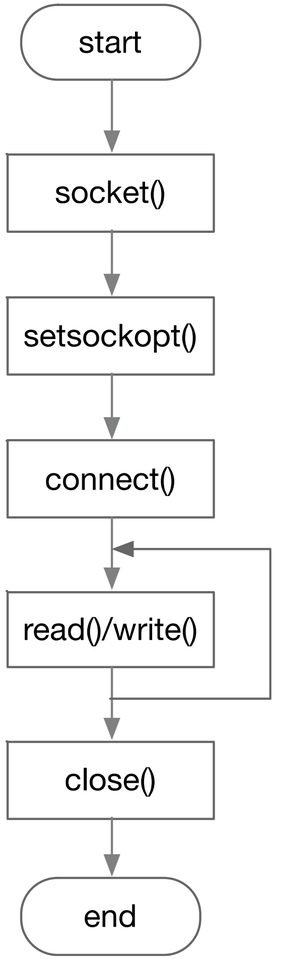
- ssize_t write(int fd, const void *buf, size_t count)
- ssize_t read(int fd, void *buf, size_t count)

返回值大于等于 0，表示发送和接收的字节数；返回值-1表示API调用失败，errno里面会保存相应的错误码。perror() API可以用来输出errno表示的错误。



client端socket编程

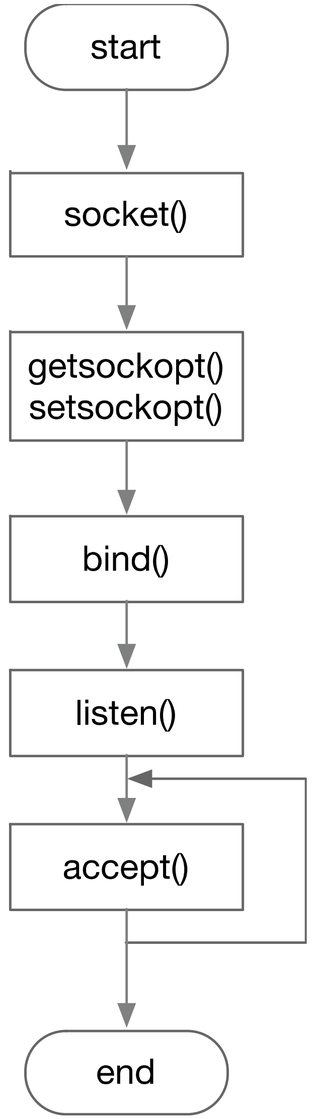
- socket(): 创建一个数据传输socket。
- getsockopt()/setsockopt(): 获取和设置socket的选项。
- connect()：建立TCP连接。
- close()：关闭socket，释放资源。



client端代码示例

server端socket编程

- socket(): 创建一个用于监听的socket。
- bind(): 把socket和一个网络地址关联起来。
- listen()：开始监听连接请求。
- accept()：接受一个连接，返回一个用于数据传输的socket。



server端代码示例

Java Socket编程

Java区分用于数据传输的socket和监听的socket，用Socket这个类表示前者，用ServerSocket这个类表示后者。

Socket的getOutputStream()返回的OutputStream用于发送数据，Socket的getInputStream()返回的InputStream用于接收数据。

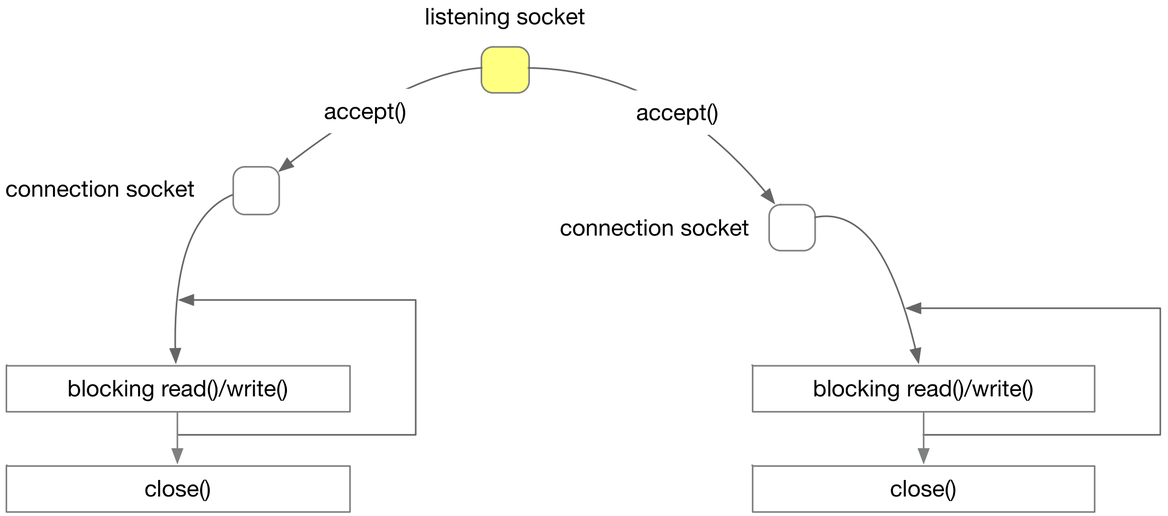
ServerSocket没有listen方法， listen是自动被执行的。

Java Socket编程代码示例

事件驱动的网络编程

阻塞IO的服务架构

这种架构使用一个进程处理一个connection，Apache HTTP server的Prefork MPM就是采用的这种架构。这种架构的问题在于进程和connection的不匹配，connection是一种lightweight的OS资源，而process是一种heavyweight的OS资源。



one-process per connection archirtecture

blocking read() means to wait until some data is read from the recive buffer.
blocking write() means to wait unitl some data is written to the send buffer.

事件驱动的网络编程架构

epoll API

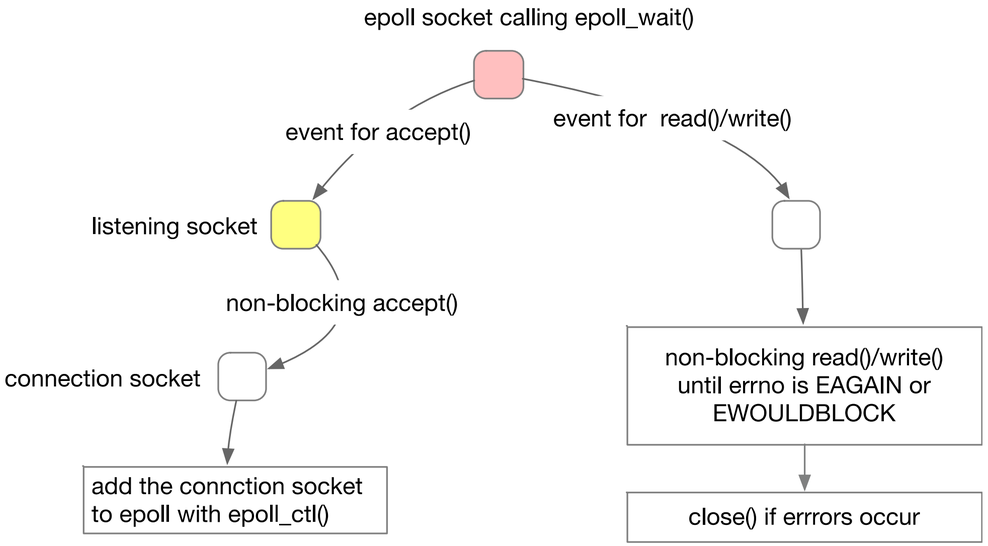
epoll提供以下3个API：

- epoll_create1：创建epoll文件描述符。
- epoll_wait：等待和epoll文件描述符关联的I/O事件。
- epoll_ctl：设置epoll文件描述符的属性，更新文件描述符和epoll文件描述符的关联。

EPOLL事件的两种模型： Level Triggered (LT) 水平触发和Edge Triggered (ET) 边沿触发 。默认是水平触发。

事件驱动的网络编程架构

使用一个event loop进行网络数据的发送和接收。



event-driven architecture

优点：1. 避免了大量创建process的OS资源消耗。2. 减少了耗时的context switch。

缺点：事件驱动的编程麻烦一些。

epoll代码示例

Java的事件驱动网络编程

Java NIO事件驱动网络编程

Java NIO事件驱动网络编程主要由以下3部分组成：

- Buffer: 数据缓冲区，对应epoll模型的数组。
- Channel: I/O操作的数据通道，对应epoll模型的socket文件描述符。
- Selector和selection key: 对应epoll模型的epoll文件描述符。

以上三者构成了对底层事件驱动网络编程的封装。在Linux平台上，Java NIO最初支持的事件驱动网络编程模型是select/poll。从JDK 8开始，JDK开始支持epoll。

Java NIO的事件驱动网络编程和epoll的一样也是要在一个event loop上面加数据处理的handler。

为什么引入Buffer?

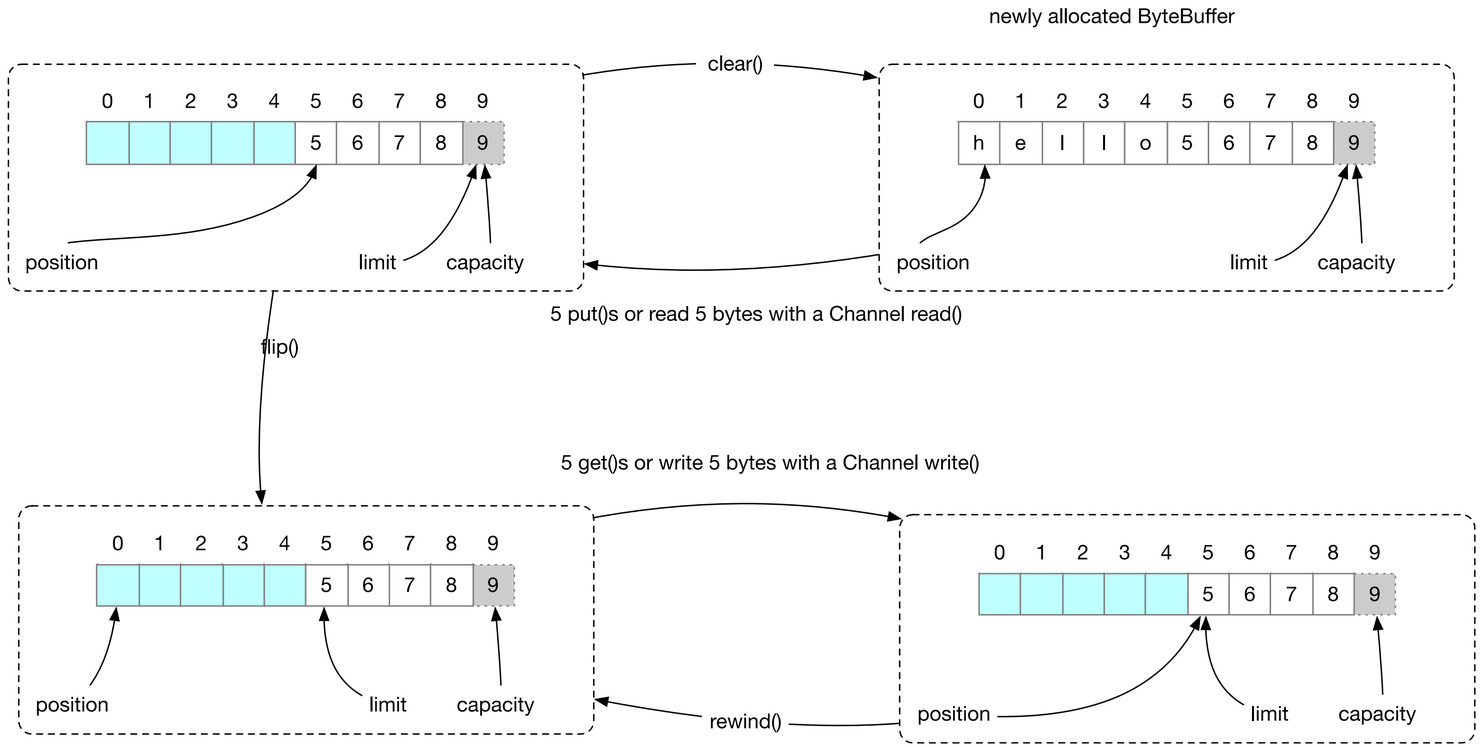
设想这样一个场景，从一个socket里面进行数据读取，直到读到\n为止，然后把读到的数据（可能包括\n后面的一些数据）写入到另外一个socket。下面分别用C和Java NIO实现了这个功能，可以看出Buffer简化了对数组中index的维护工作。

```c
// C Code
#define SIZE 1024
char buf[SIZE];

int ret;
int position = 0;

for (;;) {
  ret = read(fd1, buf + position, SIZE - position);
  // Omit the code that handle the case for ret = 0 and ret = -1
  position += ret;
  if (received data contains '\n') {
    break;
  }
}

int count = position;
position = 0;
for (;;) {
  ret = write(fd2, buf + position, count - position);
  // Omit the code that handle the case for ret = -1
  position += ret;
  if (position == count) {
    break;
  }
}
```

```java
// Java code
// Omit the code for exception handling
int ret;
ByteBuffer buf = ByteBuffer.allocate(1024);
for (;;) {
  ret = channel1.read(buf);
  // Omit the code that handle the case for ret = -1
  if (received data contains '\n') {
    break;
  }
}

buf.flip();
while (buf.remaining() > 0) {
  channel2.write(buf);
}
```

ByteBuffer状态图

一个ByteBuffer除了数据缓存区本身之外，有position、limit，capacity着3个属性。下图列出了ByteBuffer的方法是如何更新position和limit这两个属性。



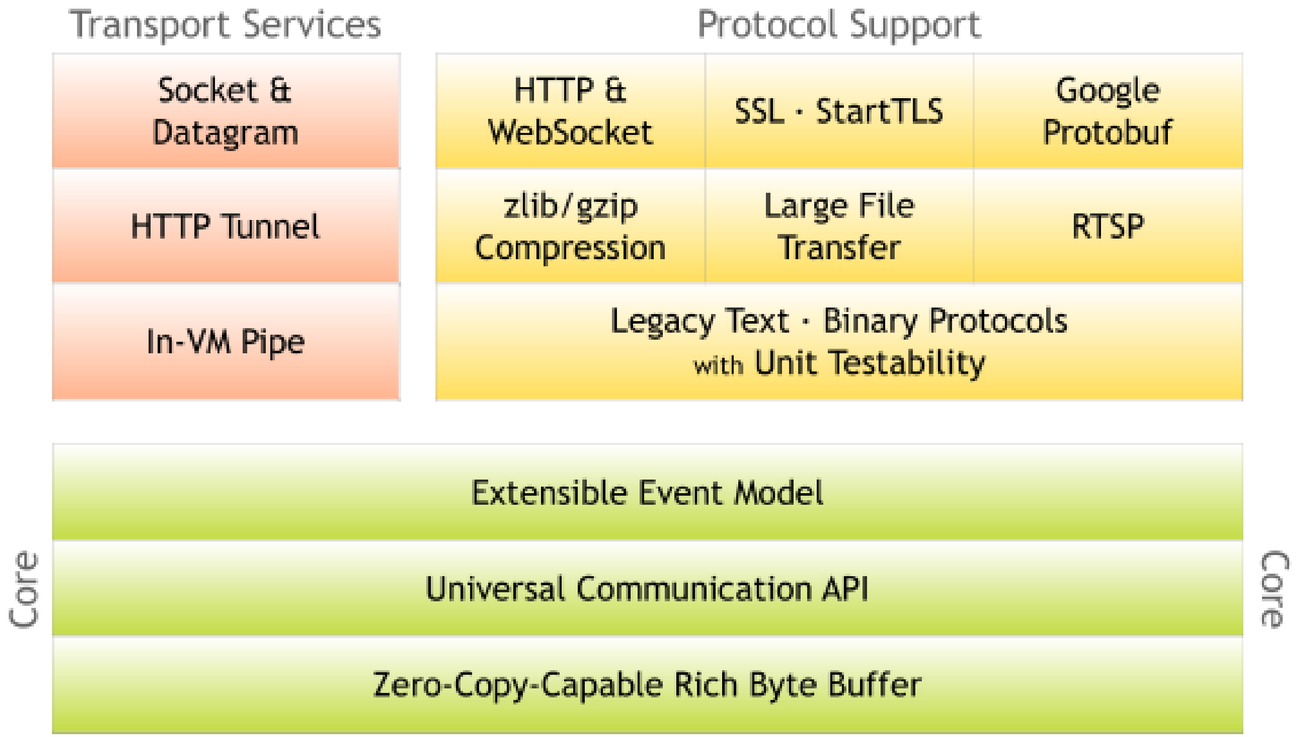
Direct ByteBuffer vs Non-direct Buffer

Direct ByteBuffer: 内存缓冲区存在于JVM之外，不受Java GC的控制。

Non-direct ByteBuffer:内存缓冲区存在于JVM的heap中，受Java GC的控制。

Netty事件驱动网络编程

Netty是在Java NIO基础之上封装的一个事件驱动网络编程。Netty在Java得到了广泛的应用，很多知名的Java项目都使用了Netty，例如gRPC和Apache Spark。



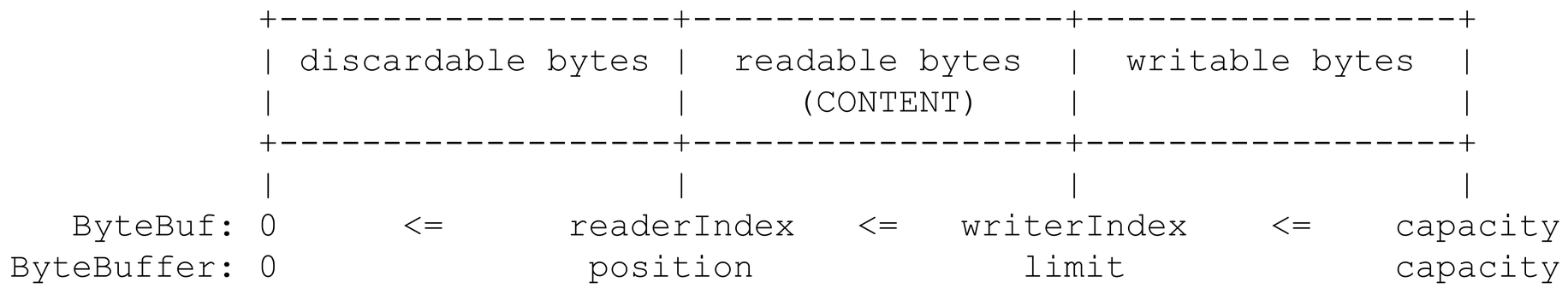
Netty事件驱动网络编程核心类和接口

Netty事件驱动网络编程主要由以下4部分组成：

- Buffers: 数据缓冲区，对应Java NIO的buffers。
- Channel: I/O操作的数据通道，对应Java NIO的Channel 。
- Bootstrap和ServerBootstrap : Bootstrap用来初始化一个Netty client端， ServerBootstrap用来初始化一个Netty server端。
- Handler： Netty的用户通过实现Netty的各种handler的接口方法，来进行数据的处理，不需要在一个event loop上面显示的加handler。

Netty ByteBuf

Netty ByteBuf对应的是Java NIO。下图展示了ByteBuffer和ByteBuf属性的对应关系。



ByteBuf的优点：

- 使用了更准确的属性命名。
- 提供了更丰富的方法：检索ByteBuf内容的方法和通过已有ByteBuf生成新ByteBuf的方法。

Netty Echo Server代码示例

Java的事件驱动网络编程

Java NIO事件驱动网络编程

Java NIO事件驱动网络编程主要由以下3部分组成：

- Buffer: 数据缓冲区，对应epoll模型的数组。
- Channel: I/O操作的数据通道，对应epoll模型的socket文件描述符。
- Selector和selection key: 对应epoll模型的epoll文件描述符。

以上三者构成了对底层事件驱动网络编程的封装。在Linux平台上，Java NIO最初支持的事件驱动网络编程模型是select/poll。从JDK 8开始，JDK开始支持epoll。

Java NIO的事件驱动网络编程和epoll的一样也是要在一个event loop上面加数据处理的handler。

为什么引入Buffer?

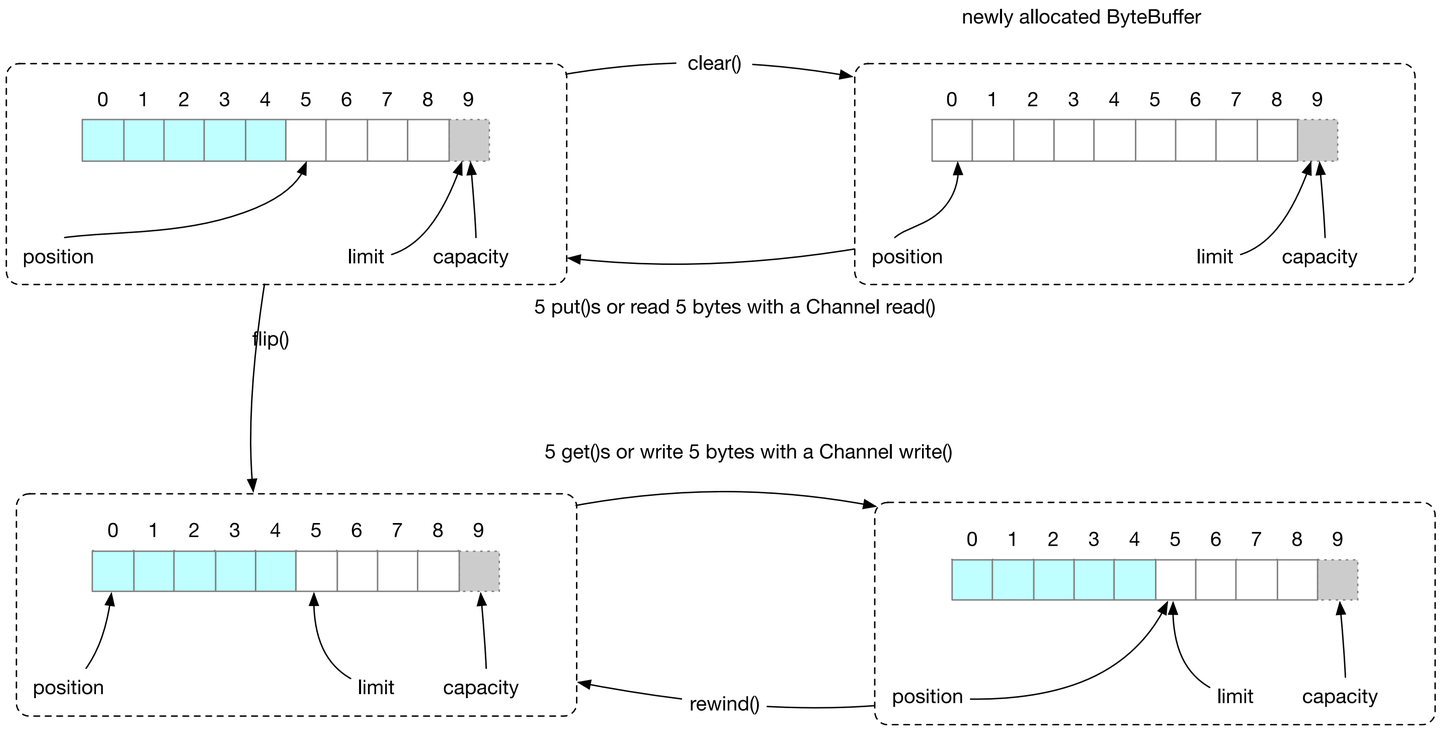
设想这样一个场景，从一个socket里面进行数据读取，直到读到\n为止，然后把读到的数据（可能包括\n后面的一些数据）写入到另外一个socket。下面分别用C和Java NIO实现了这个功能，可以看出Buffer简化了对数组中index的维护工作。

```c
// C Code
#define SIZE 1024
char buf[SIZE];

int ret;
int position = 0;

for (;;) {
  ret = read(fd1, buf + position, SIZE - position);
  // Omit the code that handle the case for ret = 0 and ret = -1
  position += ret;
  if (received data contains '\n') {
    break;
  }
}

int count = position;
position = 0;
for (;;) {
  ret = write(fd2, buf + position, count - position);
  // Omit the code that handle the case for ret = -1
  position += ret;
  if (position == count) {
    break;
  }
}
```

```java
// Java code
// Omit the code for exception handling
int ret;
ByteBuffer buf = ByteBuffer.allocate(1024);
for (;;) {
  ret = channel1.read(buf);
  // Omit the code that handle the case for ret = -1
  if (received data contains '\n') {
    break;
  }
}

buf.flip();
while (buf.remaining() > 0) {
  channel2.write(buf);
}
```

ByteBuffer状态图

一个ByteBuffer除了数据缓存区本身之外，有position、limit，capacity着3个属性。下图列出了ByteBuffer的方法是如何更新position和limit这两个属性。



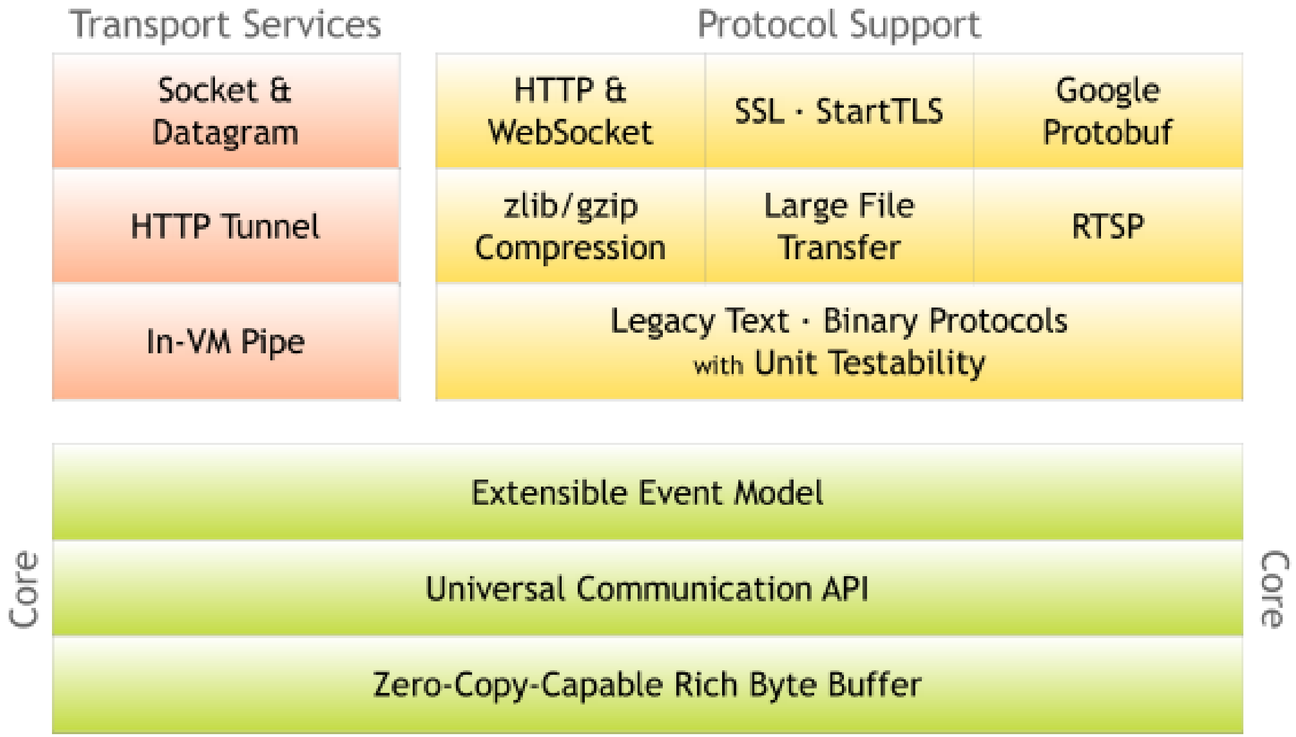
Direct ByteBuffer vs Non-direct Buffer

Direct ByteBuffer: 内存缓冲区存在于JVM之外，不受Java GC的控制。

Non-direct ByteBuffer:内存缓冲区存在于JVM的heap中，受Java GC的控制。

Netty事件驱动网络编程

Netty是在Java NIO基础之上封装的一个事件驱动网络编程。Netty在Java得到了广泛的应用，很多知名的Java项目都使用了Netty，例如gRPC和Apache Spark。



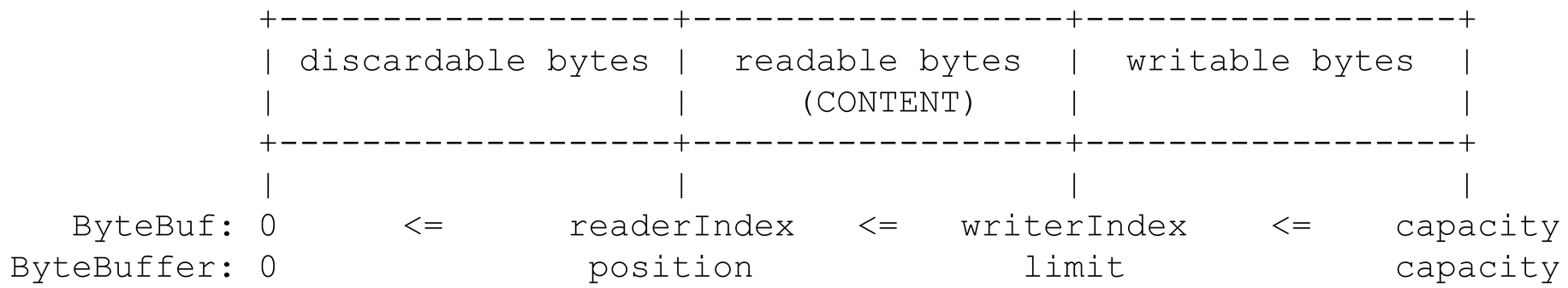
Netty事件驱动网络编程核心类和接口

Netty事件驱动网络编程主要由以下4部分组成：

- Buffers: 数据缓冲区，对应Java NIO的buffers。
- Channel: I/O操作的数据通道，对应Java NIO的Channel 。
- Bootstrap和ServerBootstrap : Bootstrap用来初始化一个Netty client端， ServerBootstrap用来初始化一个Netty server端。
- Handler： Netty的用户通过实现Netty的各种handler的接口方法，来进行数据的处理，不需要在一个event loop上面显示的加handler。

Netty ByteBuf

Netty ByteBuf对应的是Java NIO的ByteBuffer 。下图展示了ByteBuffer和ByteBuf属性的对应关系。



ByteBuf的优点：

- 使用了更准确的属性命名。
- 提供了更丰富的方法：检索ByteBuf内容的方法和通过已有ByteBuf生成新ByteBuf的方法。

Netty Echo Server代码示例

ZooKeeper的客户端网络通信源码解读

源码阅读心得

- 带着问题阅读，每次只关心眼前问题，忽略和目前无关的细节。
- 必要时可以对某些代码的功能做一个猜测。
- 及时记录得到的结论。

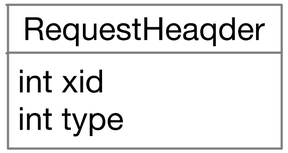
RPC网络数据结构

RPC包结构(请求和响应)

len是后面Data的长度，是一个int。

请求

Request是保存请求数据的各种Record。

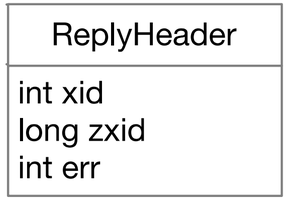


xid代表了一个客户端发请求的序号，
用来保证请求的FIFO，RequestHeader的type
给出了Request是什么，例如CreateRequest。

create API请求

响应

Response是保存响应数据的各种Record。



xid对应request中的xid，zxid表示Zookeeper
最新的事务ID，err是一个错误码（例如Code.OK
和Code.NONODE），表示处理请求的结果状态。

create API响应

ZooKeeper网络通信概述

支持两种事件驱动编程模型，一种是Java NIO，一种是Netty。核心接口和类如下：

- ZooKeeper：用户使用的ZooKeeper客户端库核心类。
- ClientCnxn: 负责和多个ZooKeeper节点的一个建立网络连接，包含ZooKeeper RPC的处理逻辑。
- ClientCnxnSocket: 网络通信的high level逻辑。
- ClientCnxnSocketNetty: 实际进行TCP socket的网络通信。
- StaticHostProvider: 提供一个ZooKeeper节点列表。

配置Netty Event Loop源码和建立连接源码解读

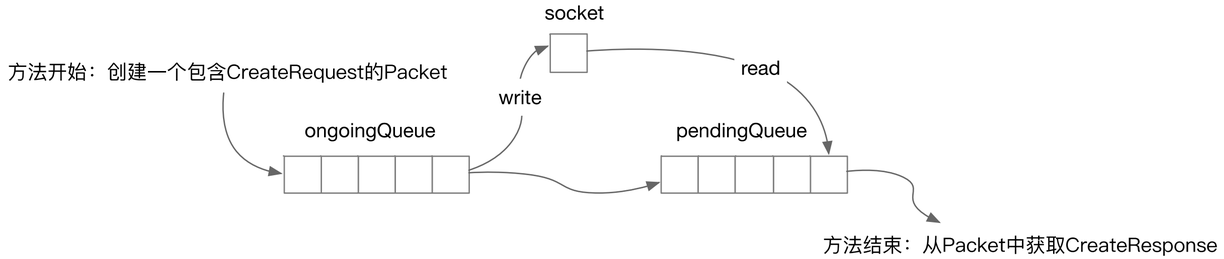
```
ZooKeeper.getClientCnxnSocket()
ClientCnxnSocketNetty.connect()
  bootstrap.connect(addr)
  operationComplete()
    incomingBuffer = lenBuffer;
    sendThread.primeConnection()
  ZKClientPipelineFactory.initChannel()
```

RPC方法流程

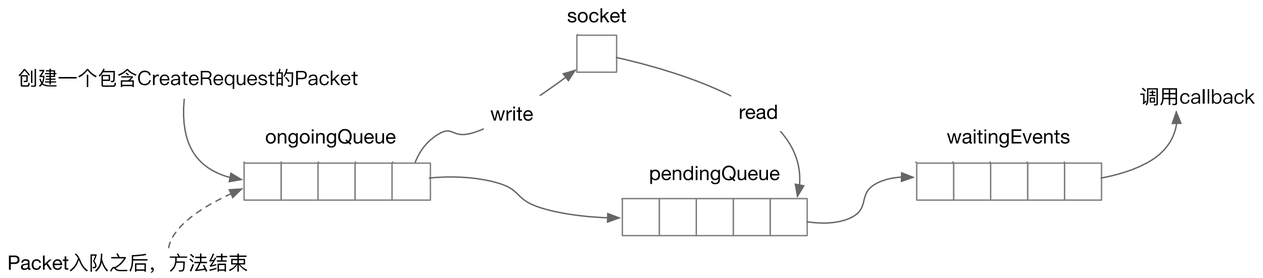
核心数据结构Packet

Packet
requestHeader request replyHeader response callback context

create API同步版



create API异步版



ongoingQueue: 等待发送请求的Packet。
pendingQueue: 等待响应的的Packet。已经收到响应的请求从pendingQueue出队。
waitingEvents：收到响应的Packet。

SendThread源码解读

SendThread负责把ongoingQueue里面的packet出队写入到socket，同时把packet入队到pendingQueue。

```
SendThread.run()
loop
  if !clientCnxnSocket.isConnected()
    serverAddress = hostProvider.next(1000)
    startConnect(serverAddress)
      ClientCnxnSocketNetty.connect()
  ClientCnxnSocketNetty.doTransport()
    doWrite()
      loop
        sendPktOnly()
          channel.write()
        pendingQueue.add(p);
        p = outgoingQueue.remove()
```

create API同步版源码解读

把Packet入队到ongoingQueue，调用packet.wait()进行等待：

```
ZooKeeper.create()
  ClientCnxn.submitRequest()
    ClientCnxn.queuePacket()
      outgoingQueue.add(packet);
  packet.wait()
```

收到响应后，调用packet.notifyAll()进行通知：

```
ZKClientHandler.channelRead0()
  sendThread.readResponse(incomingBuffer)
    packet = pendingQueue.remove()
    finishPacket(packet)
      packet.notifyAll()
```

EventThread源码解读

EventThread调用从waitingEvents里面出队的Packet的callback。下图展示了调用一个create API异步版的callback的call graph：

```
EventThread.run()
  loop:
    Object event = waitingEvents.take()
    processEvent(event)
      StringCallback cb = (StringCallback) p.cb
      cb.processResult()
```

create API异步版源码解读

把Packet入队到ongoingQueue，方法返回：

```
ZooKeeper.create()
  ClientCnxn.queuePacket()
    outgoingQueue.add(packet)
```

收到响应后，把Packet入队到waitingEvents：

```
ZKClientHandler.channelRead0()
  sendThread.readResponse(incomingBuffer)
    packet = pendingQueue.remove()
    finishPacket(packet)
      eventThread.queuePacket(p)
        waitingEvents.add(packet)
```

ZooKeeper的服务器网络通信源码解读

Standalone ZooKeeper启动源码解读

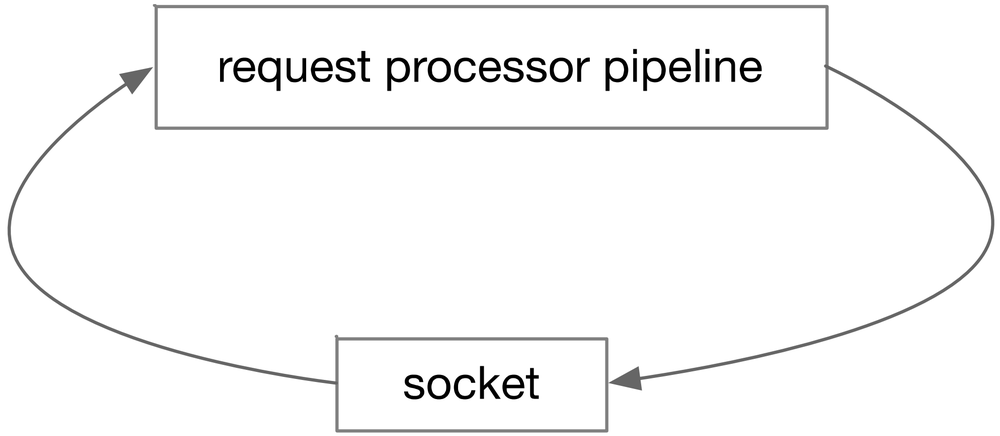
```
ZooKeeperServerMain.main()
  initializeAndRun()
    runFromConfig()
      new FileTxnSnapLog()
      new ZooKeeperServer()
      cnxnFactory = ServerCnxnFactory.createFactory()
      cnxnFactory.startup()
```

配置Netty Event Loop源码解读

```
NettyServerCnxnFactory()
```

服务器端请求处理过程

一个RPC请求在从TCP socket读取之后，经过request processor pipeline的处理，最后把响应写回socket。



接收请求源码解读

```
NettyServerCnxnFactory.CnxnChannelHandler.channelRead()
   cnxn.processMessage(msg)
     receiveMessage()
       zks.processPacket()
         submitRequest(si) // si is a Request
           firstProcessor.processRequest(Request si)
```

发送响应源码解读

```
FinalRequestProcessor.processRequest()
  ServerCnxn.sendResponse()
     NettyServerCnxn.sendBuffer(bb)
      channel.writeAndFlush
```

ZooKeeper的Request Processor源码解读

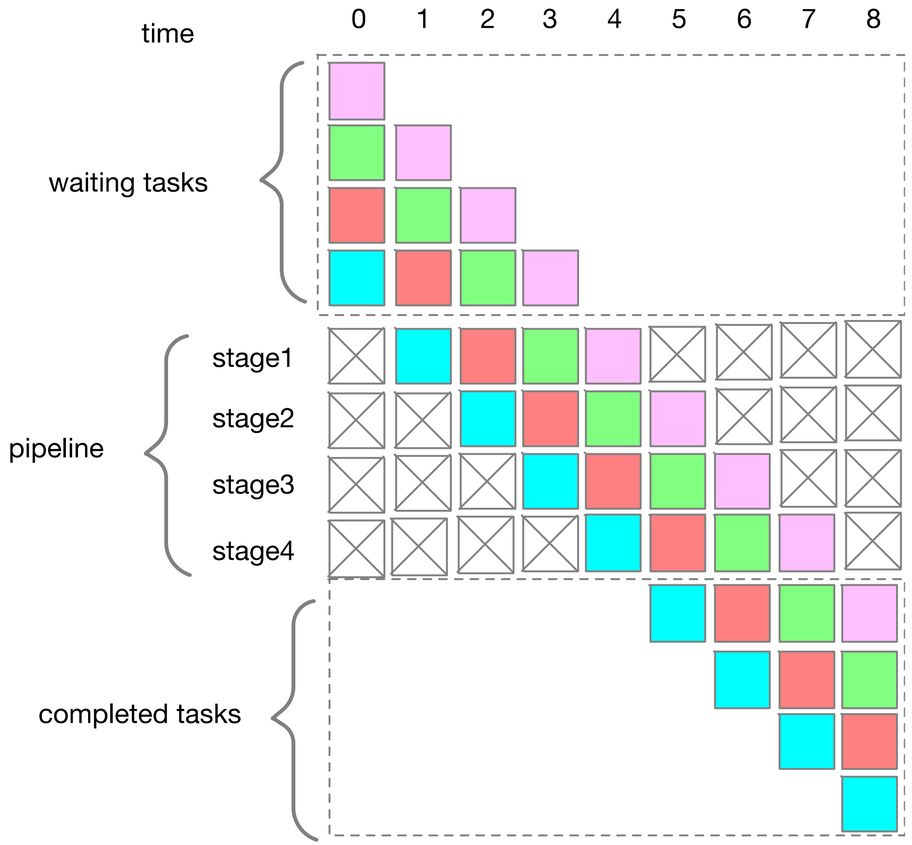
流水线(pipeline)

流水线是工业界广泛应用的技术，在计算机领域更是在诸多领域发挥了重要作用。

- 在体系结构领域，instruction pipelines是提升CPU处理能力的重要手段。
- UNIX pipeline(例如ls -l | grep key | less)是UNIX操作系统经久不衰的经典设计。

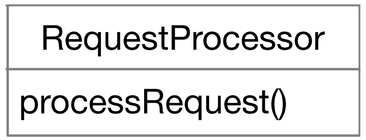
流水线的作用：

- 提升系统的吞吐量。
- 把一个复杂任务分解多个阶段，可以减低系统研发的复杂度，另外可以更方便的重用这些独立的阶段。



Request Processor流水线

ZooKeeper requeest processor流水线的核心接口是RequestProcessor。

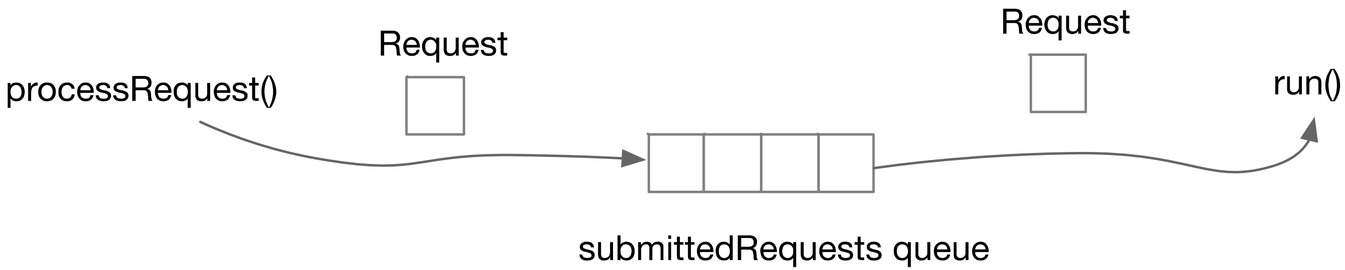


多个RequestProcessor组成一个流水线，完成处理客户端请求的完整流程。

以下是standalone模式下ZooKeeper节点处理客户端请求的流水线。

PrepRequestProcessor源码解读

PrepRequestProcessor是standalone模式下的第一个request procoessor，主要用来生成写请求的事务记录。它的核心逻辑入下图所示：



PrepRequestProcessor 的shutdown使用入队一个requestOfDeath来结束run方法的loop，是poison pill并发模式的应用。

SyncRequestProcessor源码解读

SyncRequestProcessor负责把事务记录写到durable storage上去，做了Group Commit的优化。Group Commit的commit size大于1000。

```
loop:
  if toFlush.isEmpty()
    // Wait for a Request since toFlush is empty
    queuedRequests.take()
  else if
    si = queuedRequests.poll()
    if si == null
      // Flush since there is no Request
      flush(toFlush)
    end
  end
  zks.getZKDatabase().append(si)
  toFlush.add(si);
  if toFlush.size() > 1000)
      flush(toFlush)
  end
```

FinalRequestProcessor源码解读

FinalRequestProcessor使用事务记录操作ZooKeeper的in-memory DataTree，并把操作结果写回到TCP socket。以下是processRequest方法的核心逻辑(以处理Create API为例)：

```
processRequest
  rc = zks.processTxn(request)
  rsp = new CreateResponse(rc.path)
  ServerCnxn.sendResponse(hdr, rsp, "response")
    NettyServerCnxn.sendBuffer(bb)
      channel.writeAndFlush()
```

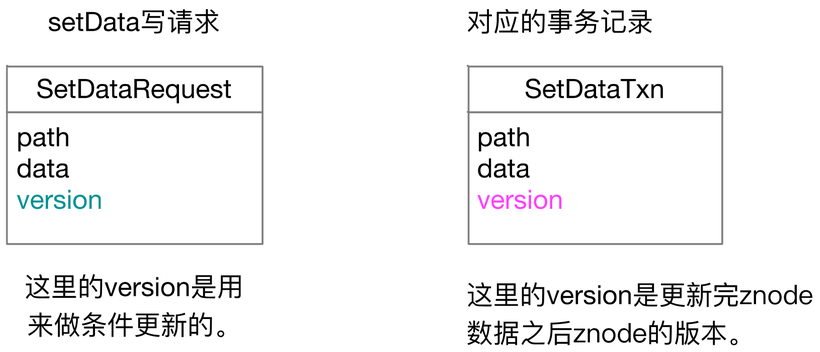
Standalone的ZooKeeper是如何处理客户端请求的?

读请求源码解读

以getData为例

事务日志

PrepRequestProcessor会为每一个客户端写请求生成一个事务，对ZooKeeper in-memory DataTree更新的时候应的不是原始的写请求，而是对应的事务记录。事务记录都是幂等的，多次应用一个事务记录不会影响结构的正确性。



ZooKeeper的事务日志是每个事务包含一条记录的REDO日志，日志记录是physical的。

事务日志源码解读

以setData为例。

为setData创建事务

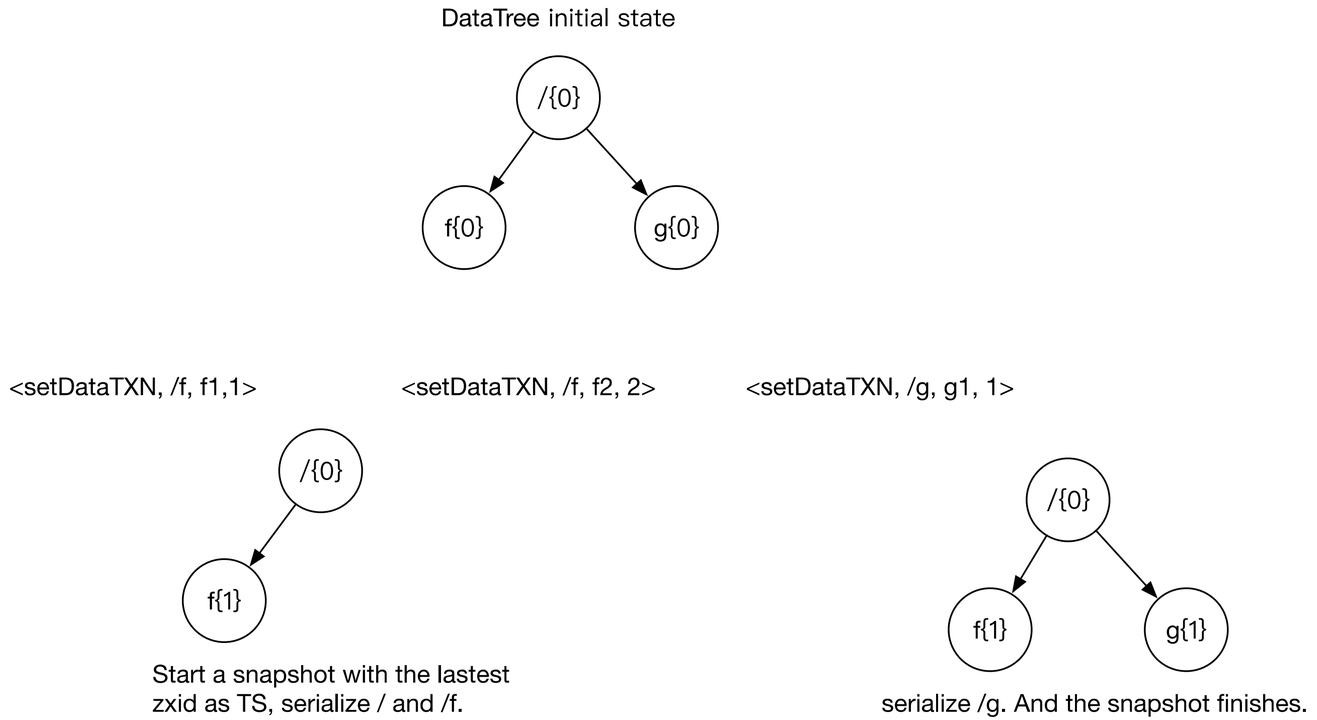
```
PrepRequestProcessor.run()
  pRequest()
    pRequest2Txn()
```

把setData事务应用到DataTree

```
FinalRequestProcessor.processRequest ()
  ZooKeeperServer.processTxn()
      ZKDatabase.processTxn(hdr, txn)
        DataTree.processTxn()
          DataTree.setData()
```

Fuzzy Snapshot

ZooKeeper可以在同时处理客户端请求的时候生成snapshot。snapshot开始时候DataTree上面最新的zxid叫作snapshot的TS。 ZooKeeper的snapshot不是一个数据一致的DataTree。但是在snapshot上面应用比TS新的事务记录之后得到的DataTree是数据一致的。



写请求源码解读(以create为例)

PrepRequestProcessor的outstandingChanges保存的是正在处理的事务要对in-memory DataTree做的变更。后面的记录在读取数据数据的时候，先要到outstandingChanges里面读取。

outstandingChanges使得ZooKeeper不必等前面的事务处理完就可以处理处理后面的事务，保证了事务的流水线处理。

极客时间

扫码试看/订阅

《ZooKeeper实战与源码剖析》视频课程