数据结构课程设计

- 1. Hot-Dump Statstics Database
- 2. File based K-V Database

开始之前

- 一个项目 3-5 人，代码独立完成
  - 可以：沟通算法、实现思路、提供测试用例
  - 不可以：拷贝代码（也无法从网上找到代码）
  - 每周需投入 1~2 小时小组讨论，及 6~10 小时个人学习、实践的时间
- 如何开展：
  - 确定 Stage X 完成计划，每周选一天中午周例会，检查 Checklist 上已完成和未完成部分，分享遇到的问题和解决思路，组织 Code Review（可选）
  - Github 建立个人项目，每周至少 Push 一次代码（要学会用 git）
  - 验收方式：测试用例 + Code Review

开始之前


Baidu 百度
Google C++ 编码规范
百度一下
网页 资讯 视频 图片 知道 文库 贴吧 采购 地图 更多»
百度为您找到相关结果约23,900,000个
搜索工具
Google的C++代码规范 - CSDN博客
2017年12月29日 - 编程规范,没有之一,建议广大国内外it研究使用。“google c++ style guide是一份不错的c++编码指南,下面是一张比较全面的说明图,可以在短时间内快速掌握规范的...
CSDN技术社区 - 百度快照
Google C++编程规范 (中文版) - Hhrock的博客 - CSDN博客
2018年9月4日 - 1414 googlec++styleguide是一份不错的c++编码指南,我制作了一张比较全面的说明图,可以在短时间内快速掌握规范的重点内容。不过规范毕竟是人定的,记得...
CSDN技术社区 - 百度快照
Google C++ 编码规范 - 神马小猿 - 博客园
2018年5月30日 - 刚刚看到一位博主的文章分享Google C++ 编码规范 本人做一下记录,方便以后学习。。 中文在线版本地址: http://zh-google-styleguide.readthedocs.io/e...
https://www.cnblogs.com/lsmhom... - 百度快照
Google C++编码规范 - lilei9110 - 博客园
2017年12月2日 - Google C++编码规范 http://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/contents/ posted @ 2017-12-02 16:38 lilei9110 ...
https://www.cnblogs.com/lilei9... - 百度快照

课题一

Hot-Dump Statistics Database

Statistics Database

- 《写给新人的数据库指南》：
  - https://zhuanlan.zhihu.com/p/25120684
- 分析型数据库：
  - OLTP / OLAP 的概念
  - 列分为维度（dimesion）和指标（measure）两种，所有分析都是在查询符合条件的维度下的聚合 （aggressive）结果
  - 《用户行为数据分析》 一文，以 天池 某数据集为例，介绍了一种典型的分析场景——用户行为分析 下，分析型数据库的应用案例
    - https://zhuanlan.zhihu.com/p/81012119

Statistics Database

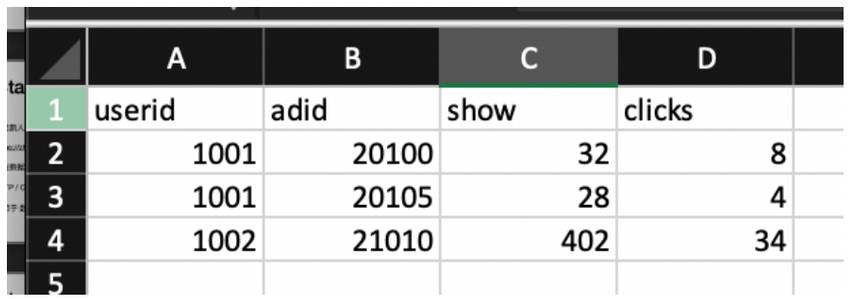
- 我们要做的分析型数据库：
  - 只支持 int 类型
  - 只支持 SUM 型聚合；
  - 只支持 插入 操作；
  - 支持 Primary Key 和 Index；
  - 单表能存储数十列 x 千万行级别数据；
  - 支持通过（简化的）SQL 进行数据库表定义和操作

Statistics Database

- Stage1：实现基于二进制文件的分析型数据库，支持数据插入、查询；
- Stage2：维护 PrimaryKey，实现 Hot-Dump 过程；实现按条件查询及查询结果聚合、排序
- Stage3：实现 SQL 交互，通过 SQL 定义表结构、操作数据表（插入、查询、JOIN）
- Stage4：
  - 实现单表多 Index 支持；实现多副本支持；
  - 实现 行列（Row-Block）混合式存储及数据压缩；

Stage1

- 64位系统中，int 类型定长 8 字节，一行的长度为列数 x sizeof(int)



	A	B	C	D
1	userid	adid	show	clicks
2	1001	20100	32	8
3	1001	20105	28	4
4	1002	21010	402	34
5				

表定义：

```
userid,adid,show,clicks
```

二进制数据文件（8字节*4列*3行=96字节）：

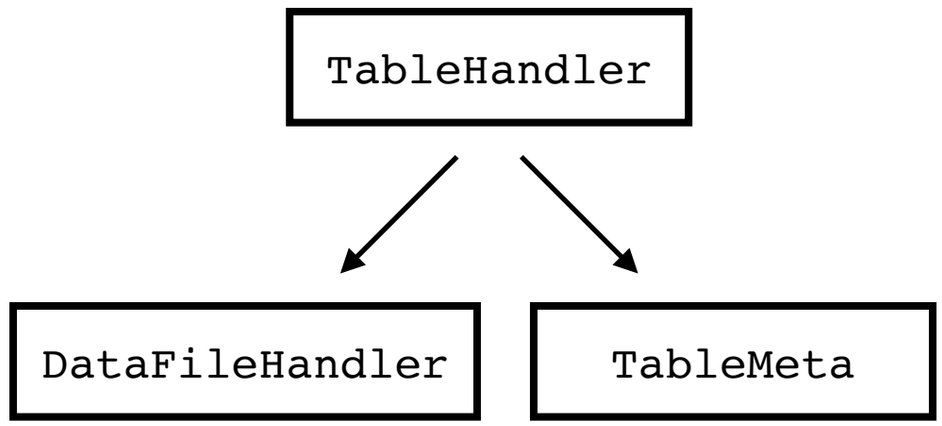
```
0x3E9|0x4E84|0x20|0x8|
0x3E9|0x4E89|0x1C|0x4|
0x3EA|0x5212|0x192|0x22
```

Stage1

- 二进制文件基本操作：
  - 学习 C/C++ 读写二进制文件
  - 实现 API 1.1.1: `DataFileHandler::open()`
    - 逻辑：若文件存在，则打开文件；若文件不存在，则创建文件
- 实现 API 1.1.2：`DataFileHandler::append()`
  - 每次插入的新数据，Append 在文件末尾，避免对文件随机读写，提高性能
- 实现 API 1.1.3：`DataFileHandler::read_line()`
  - 从指定行数开始读取一行数据，返回二进制信息 int*
  - 优化：使用预读技术，每次读入 4K 的 block

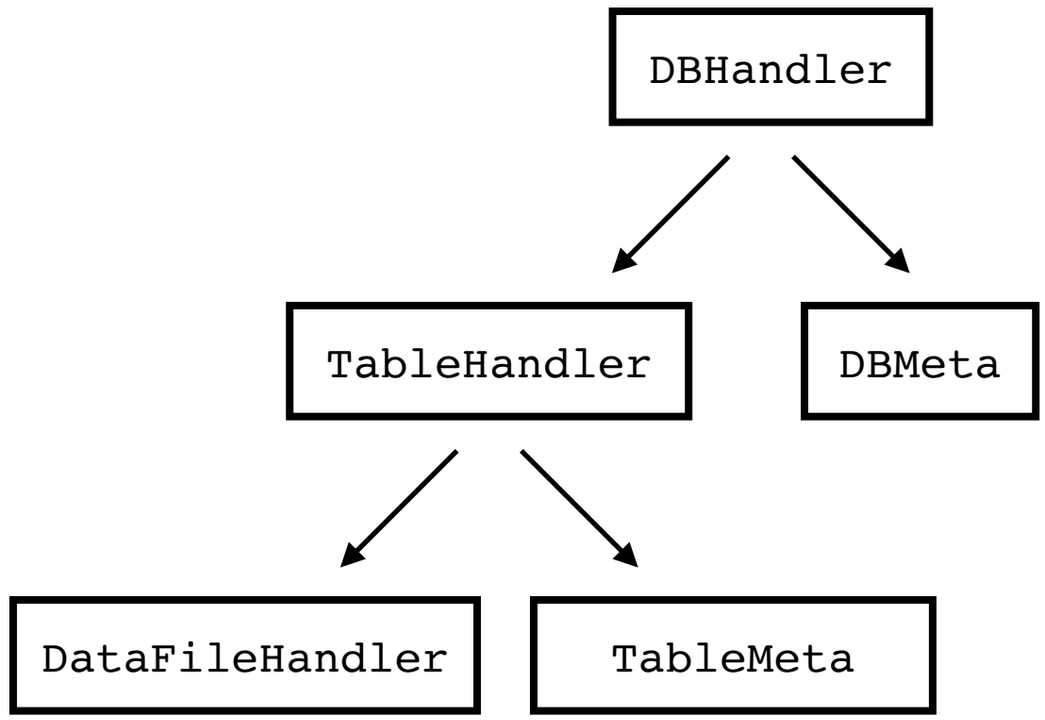
Stage1

- 表的元数据管理
  - 实现 API 1.2.1：`TableMeta::TableMeta()`
    - 根据表的元数据创建对象，元数据包括：表名、列名、主键、表文件存储路径等
  - 实现 API 1.2.2：`TableMeta::save()/load()`
    - 将元数据存储在文件中 / 从文件中加载元数据
- 表的创建、打开与删除：
  - 实现 API 1.3.1：`TableHandler::create`
  - 实现 API 1.3.2：`TableHandler::open`
  - 实现 API 1.3.3：`TableHandler::drop`



Stage1

- 数据库的元数据管理
  - 实现 API 1.4.1：`DBMeta::DBMeta()`
    - 根据数据库的元数据创建对象，元数据包括：数据库存储路径等
  - 实现 API 1.4.2：`DBMeta::save()/load()`
    - 将元数据存储在文件中 / 从文件中加载元数据
- 数据库的初始化、打开与删除：
  - 实现 API 1.5.1：`DBHandler::create()`
  - 实现 API 1.5.2：`DBHandler::open()`
  - 实现 API 1.5.3：`DBHandler::drop()`



Stage1

- 表的插入
  - 只在表的末尾追加写入（append）
  - 实现对象 `TableRow`，描述一行数据：
    - 实现 API 1.6.1 `TableRow::TableRow()`
      - 实现 API 1.6.2 `TableRow::write()` 将一行数据序列化为二进制数据，并通过 `DataFileHandler` 写入文件
- 表的查询
  - 通过 `TableRow` 对象描述一行数据：
    - 实现 API 1.6.3 `TableRow::read()` 通过 `DataHandler` 对象从二进制文件中读取一行数据，并反序列化
    - 实现 API 1.6.4 `TableRow::get_column()` 根据列名获取数据

Stage1

- 总结：
  - 了解关系数据库的定义，分析型数据库的特点
  - 熟悉基本的文件操作
  - 搭起了数据库访问、管理的底层架构

Stage2

- 任务一：从经典数据库 MySQL 中学习 Primary-Key / Index 的概念

  - Primary-Key 和 Index 的区别?

  - 多级 Index 的原理和使用方法?

  - 不同存储引擎（MyISAM/InnoDB）的 Index 是如何实现的?

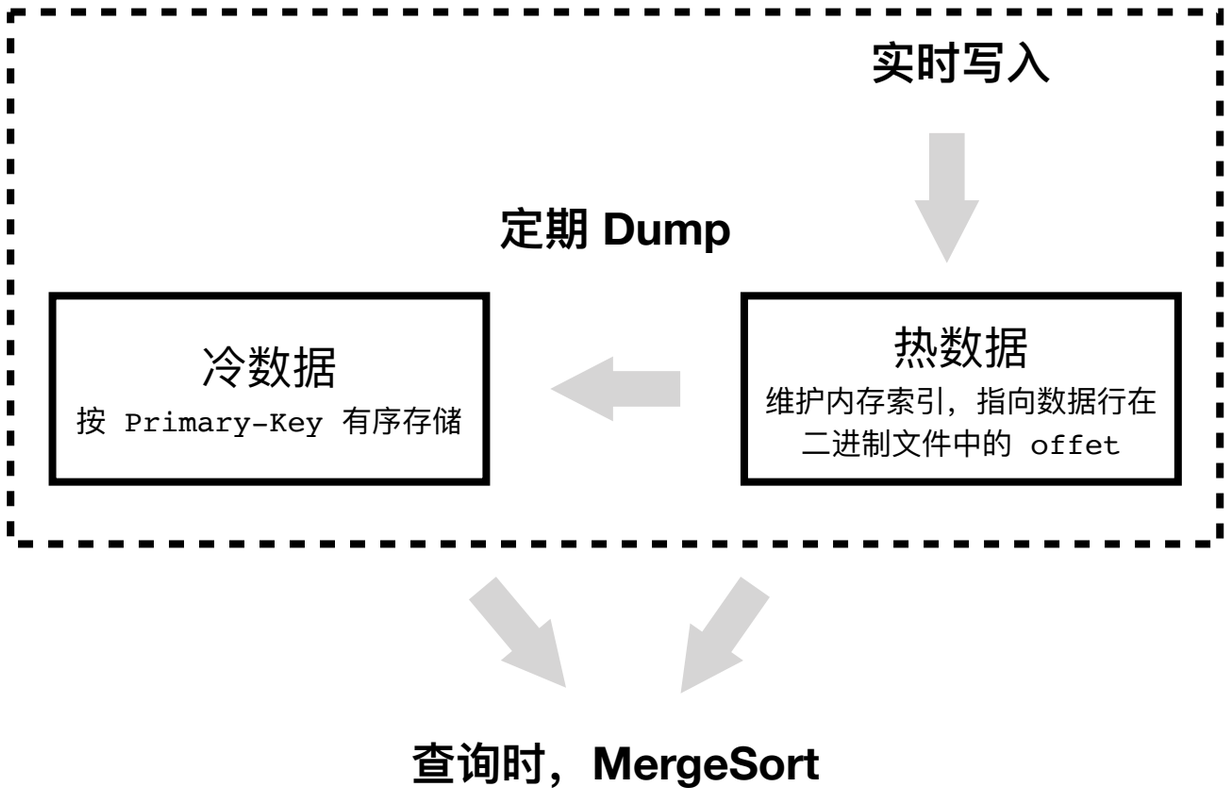
  - Hash/B+Tree 的索引各自的特点?

Stage2

- 任务二：通过 Hot-Dump 维护 Primary-Key
    - Primary-Key 特点：有序性、唯一性
    - （思考）如何实现有序性?
        - 方案一：数据使用 AVLTree、R-B Tree 等方案存储在内存中：
            - 优点：插入、查询性能高
            - 缺点：存储量受内存大小限制；机器故障时数据丢失；
        - 方案二：数据按写入顺序存储，增加内存索引，通过索引有序遍历、快速查找 Primary-Key
            - 优点：插入、查询性能较高，存储量不受内存大小限制
            - 缺点：索引的大小受内存限制；系统启动时需耗时载入索引；批量按序查询时，对物理文件产生大量随机读
        - 方案三：数据文件在物理存储时就按 Primary-Key 排序
            - 优点：查询性能高，不受内存大小影响
            - 缺点：无法实时维护物理文件有序

Stage2

- 任务二：通过 Hot-Dump 维护 Primary-Key
  - Hot-Dump 机制，融合方案一和方案二
    - 热数据使用方案一，保证数据写入实时性
    - 冷数据使用方案三，保证系统可靠性和查询性能
    - 冷、热可按 写入数据量 或 写入时间 划分；使用单独线程，定期将热数据（方案一）转换成冷数据（方案三），这个过程即 Hot-Dump
    - 查询时，分别查询热数据和冷数据，对结果进行归并排序（外部排序）



Stage2

- 任务二：通过 Hot-Dump 维护 Primary-Key
    - 热数据：方案一（的改进）
        - 在内存中，使用 AVLTree / R-B Tree 存储数据，树的 Key 为 Primary-Key，叶子中存储完整数据行
            - 对于新的 Primary-Key，直接插入树中
            - 已存在的 Primary-Key，对 Measure 进行 SUM 聚合
        - 使用 Log 文件（二进制）记录插入的数据
            - 保证机器故障时，已插入数据不丢失；故障后，遍历读取 Log 文件，恢复热数据；
            - 插入时，先写入 Log 文件，再更新内存中的 Tree（为什么？）
    - 实现 DataHot 对象维护热数据：
        - API 2.1.1 DataHot::DataIndex()
        - API 2.1.2 DataHot::load() 从 Log 文件中载入热数据
        - API 2.1.3 DataHot::insert() 插入数据
        - API 2.1.4 DataHot::begin() / end() 遍历热数据的迭代器
        - API 2.1.5 DataHot::find() 按 Primary-Key 查找数据

Stage2

- 任务二：通过 Hot-Dump 维护 Primary-Key
    - 冷数据：方案三
        - 在磁盘中，按 Primary-Key 有序存储
    - 实现 DataCold 对象
        - API 2.2.1 DataCold::DataCold()
        - API 2.2.2 DataCold::load()
        - API 2.2.3 DataCold::insert()
        - API 2.2.4 DataCold::begin() / end()
        - API 2.2.5 DataCold::find()

Stage2

- 任务二：通过 Hot-Dump 维护 Primary-Key
    - 数据查询：
        - 从热数据、冷数据中分别查询结果，并进行归并排序（外部排序）
    - Hot-Dump
        - 定期将热数据转换为冷数据
            - 思考：
                - 转换触发机制如何设计？考虑转换效率和写入速度?
                - **难点：转换过程需使用独立线程，转换过程中如果机器故障，如何处理?**
            - 实现 API 2.3 HotDump

Stage2

- 任务三：实现按条件过滤查询
  - 查询条件筛选
    - 限制：查询条件不支持 column 表达式四则计算，仅包含 &、|、>、>=、=、<、<= 和 ( )
  - 思考：如何用程序描述一组查询条件？ (A & (B | C)) | D
  - 实现一个对象，描述查询条件，输入是一行数据 `TableRow`，输出是 True/False
    - 实现 API 2.4.1 `CondFilter::CondFilter()`
    - 实现 API 2.4.2 `CondTool::filter()`

Stage2

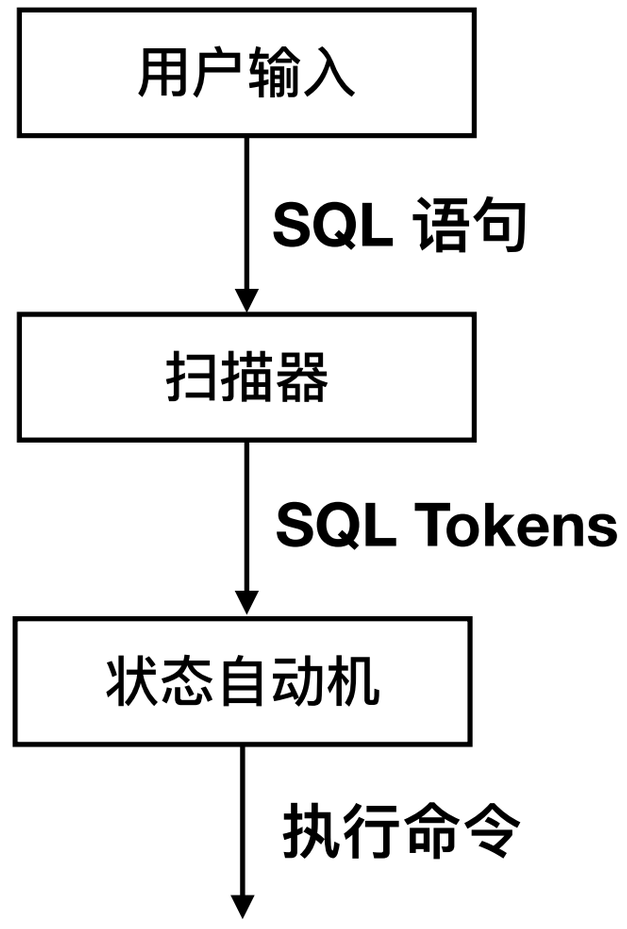
- 任务四：实现查询结果聚合与排序
    - 查询结果聚合
        - 把 Primary-Key 相同的行合并起来
        - TableRow 实现：
            - 实现 API 1.6.5 `TableRow::equal()` ，判断两个 Row 的 Parimary-Key 是否相等
            - 实现 API 1.6.6 TableRow::agg()，将两个 Row 的指标（Meaurse）聚合起来，仅支持 SUM()，可扩展包括 MAX、MIN、MEAN 等
    - 查询结果排序
        - TableRow 实现：
            - 实现 API 1.6.7 `TableRow::compare()` ，根据传入的 column 比较两个 Row

Stage2

- 总结：
- 实现了分析型数据库的完整功能
- 思考：系统设计、实现过程中，哪些地方需要考虑性能和稳定性?
- 思考：如何对系统进行测试?

Stage3

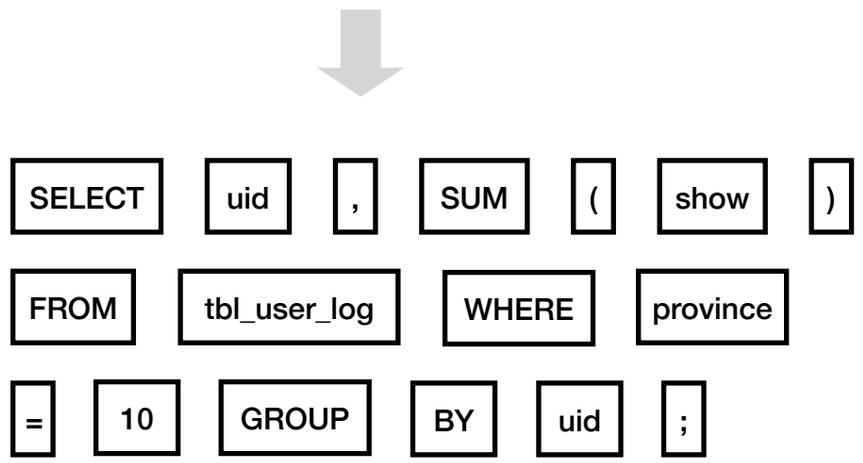
- Stage3：基于（简化的） SQL 的前端
  - 了解编译的基本原理和流程
    - 扫描器的作用与实现
    - 基于有限状态自动机实现简单的词法分析器
- 解析 SQL，并实现：
  - CREATE / DROP：创建、删除数据库表
  - DESCRIBE：显示数据库表定义
  - INSERT：数据插入、Hot-Dump 触发
  - SELECT：实现条件查询、实现 JOIN



Stage3

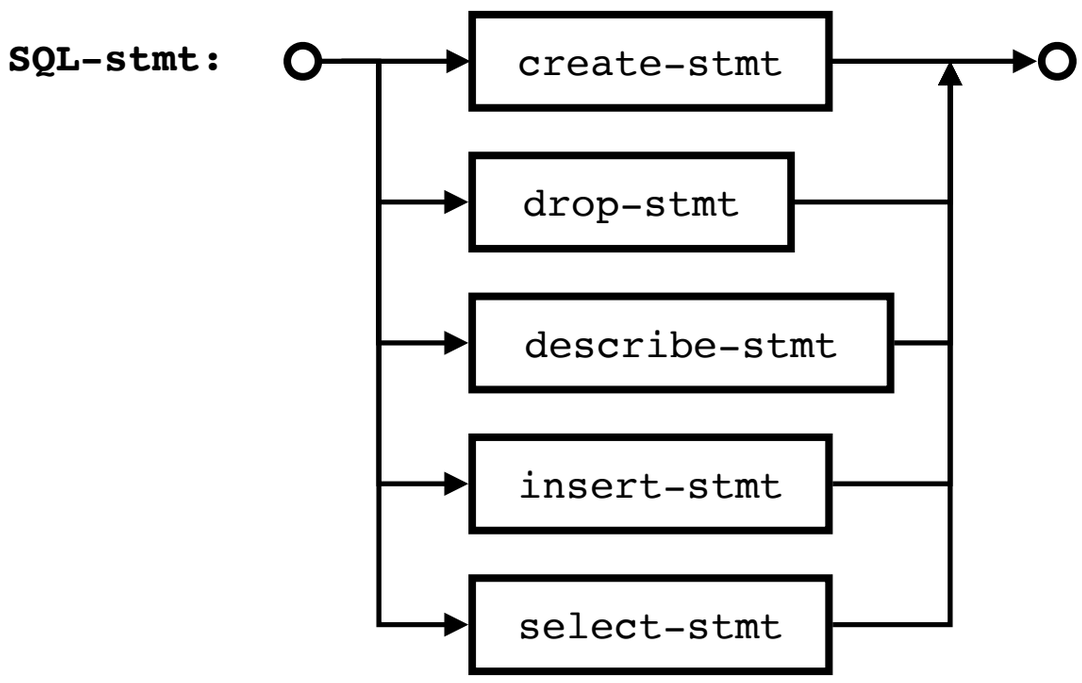
- Stage3：基于（简化的） SQL 的前端
  - 扫描器的作用与实现：
    - 关键字：SELECT、FROM、WHERE、OR、AND、...
    - 标识符：[a-zA-Z] 开头，包含 [a-zA-Z0-9_]
    - 数字
    - 运算符：=, >, <, >=, <=
    - 分隔符：\,, (,)

```
SELECT uid, SUM(show)
FROM tbl_user_log
WHERE province=10
GROUP BY uid;
```



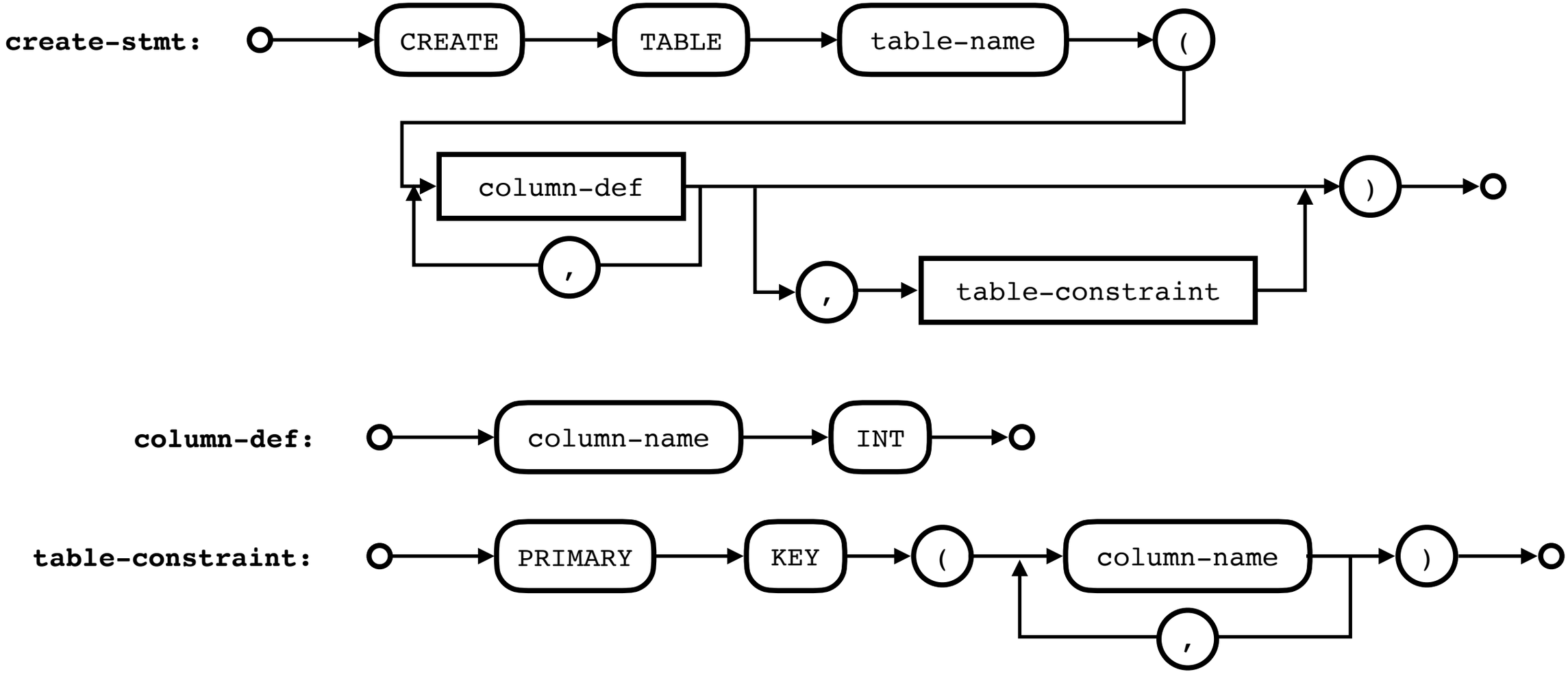
Stage3

- Stage3：基于（简化的） SQL 的前端
  - 使用 有限状态自动机（DFA）分析 SQL 语句
    - 需自己设计状态机的转移过程，并实现。下面是一个参考



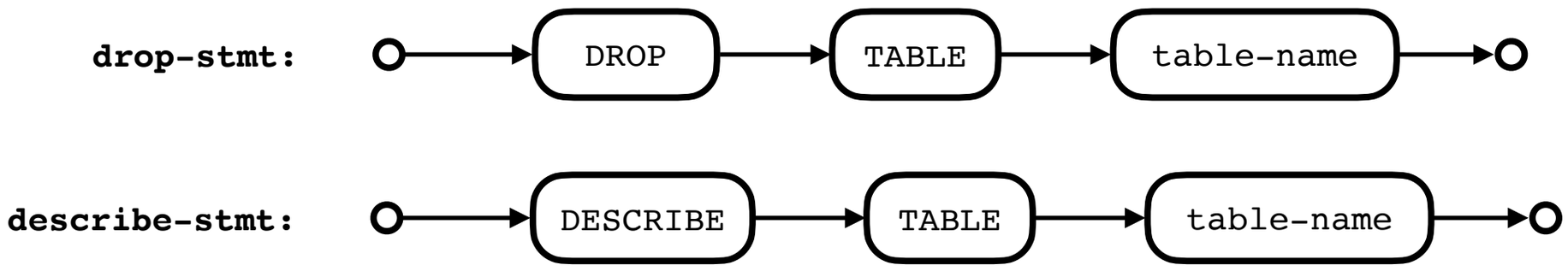
Stage3

- Stage3：基于（简化的） SQL 的前端
  - 使用 有限状态自动机（DFA）分析 SQL 语句
    - 需自己设计状态机的转移过程，并实现。下面是一个参考



Stage3

- Stage3：基于（简化的） SQL 的前端
  - 使用 有限状态自动机（DFA）分析 SQL 语句
    - 需自己设计状态机的转移过程，并实现。下面是一个参考

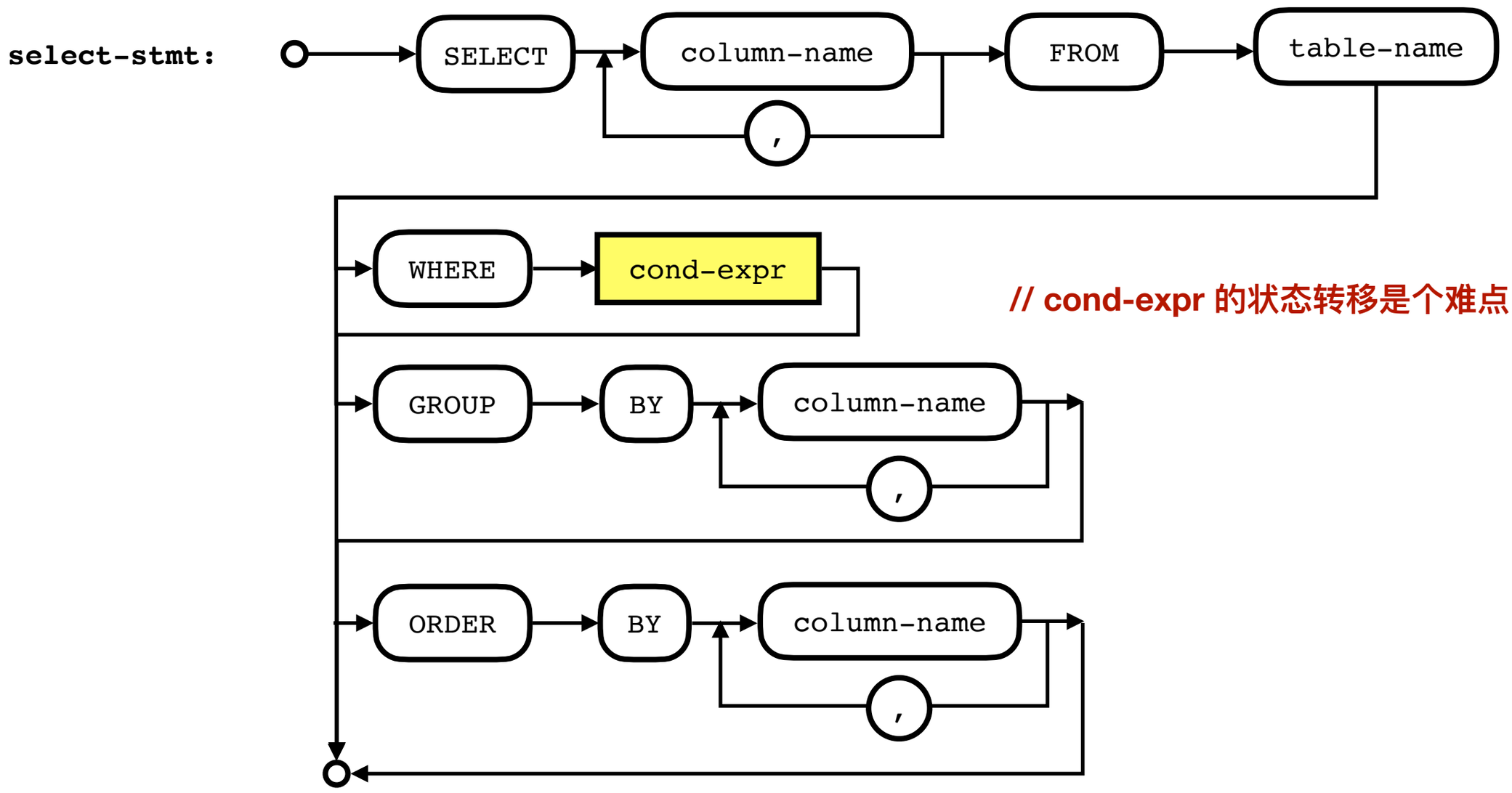


Stage3

- Stage3：基于（简化的） SQL 的前端
  - 使用 有限状态自动机（DFA）分析 SQL 语句
    - 需自己设计状态机的转移过程，并实现。下面是一个参考

Stage3

- Stage3：基于（简化的） SQL 的前端
  - 使用 有限状态自动机（DFA）分析 SQL 语句
    - 需自己设计状态机的转移过程，并实现。下面是一个参考



// cond-expr 的状态转移是个难点

Stage3

- 总结：
  - 实现了简单的 SQL 解析器
  - 自学：了解 MySQL 的插件式存储引擎机制
  - 可将 Stage1、Stage2 实现的分析型数据库作为 MySQL 的一个存储引擎
    - 利用 MySQL 进行 SQL 解析，兼容现有数据库的接口（如 pymysql 等）
    - 利用 SQL 优化、Condition PushDown（5.7+）等特性，实现查询优化

Stage4

- Stage4：针对分析型数据库的优化
  - 多 Index 支持
  - 多副本支持
  - 行列混合式存储
  - …

Statistics Database

- 总结：
  - 业界优秀的开源实现：ClickHouse
    - 《彪悍开源的分析数据库-ClickHouse》
      - https://zhuanlan.zhihu.com/p/22165241
    - 《ClickHouse深度揭秘》
      - https://zhuanlan.zhihu.com/p/98135840
- 《神策分析的技术选型与架构实现》
  - https://www.sensorsdata.cn/blog/technical_implementation_of_sensors_analytics/
- 《VLDB论文解读：阿里云超大规模实时分析型数据库AnalyticDB》
  - https://zhuanlan.zhihu.com/p/80913343

课题二

File based K-V Database

K-V Database

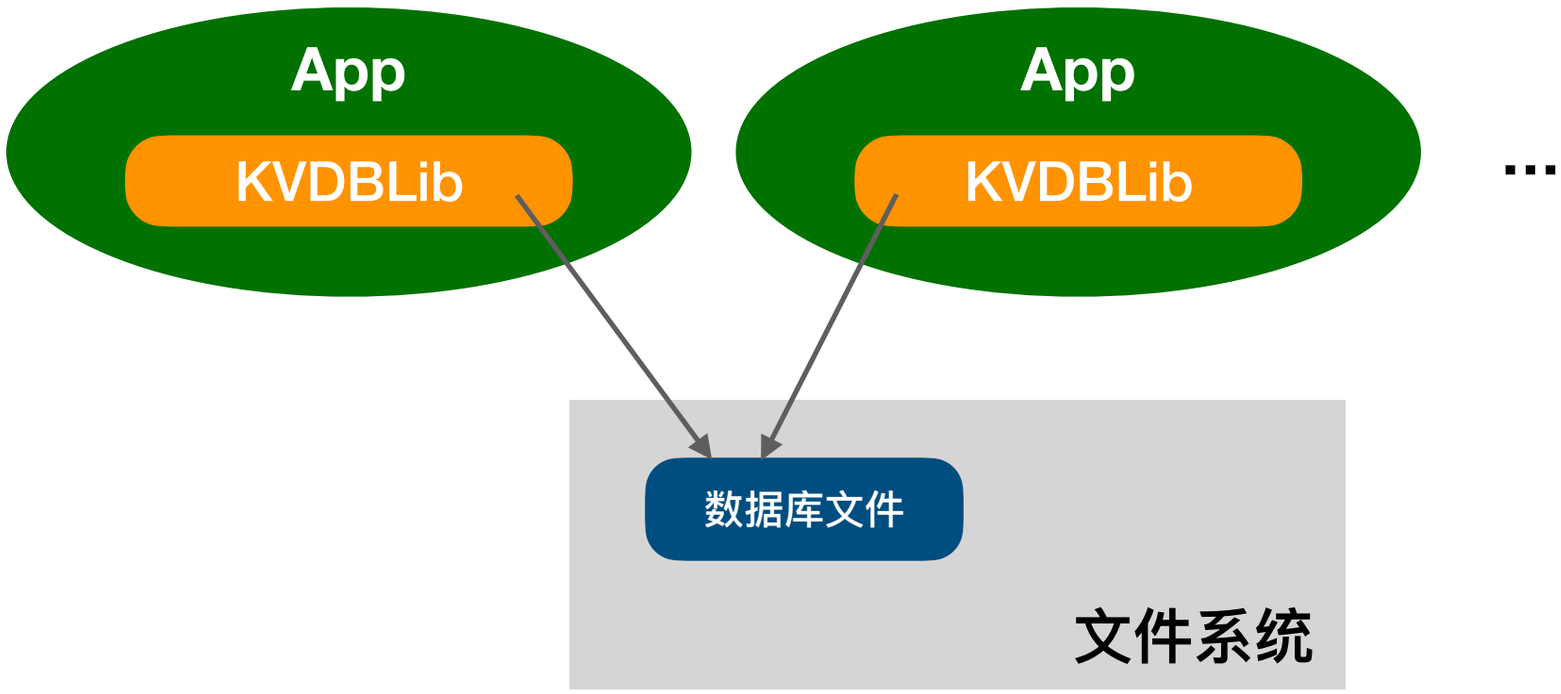
- https://en.wikipedia.org/wiki/Key-value_database
- Key-Value DB & Relational DB:
  - NoSQL & SQL, Primary Key Only Access DB
  - Schemaless
- 天然适合分布式

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

一个表，展示了不同的键关联着不同的格式化数据

文件存储的数据库

- 基于单个文件存储的数据库的典范：SQLite3
  - 提供多语言 API
  - 跨平台，支持 Windows、Linux、Android、iOS 等..
  - 绿色、轻量，Lib 百 KB 级，无第三方依赖库



File Based K-V Database

- Stage1：实现一个基于文件的 K-V 数据库，支持基本操作
- Stage2：支持内存索引、支持超时类接口
- Stage3：支持 LRU Cache、支持 List、Set 类型
- Stage4：使用数据库完成一个真实任务

File Based K-V Database

- Stage1：
    - 创建一个基于文件存储的 K-V 数据，支持基本操作
        - API1: KVDBHandler
        - API2: get / set / del
- 使用 Append-Only File 存储 Key-Value 数据
    - API3: purge
- 处理异常情况
    - Return CODE

文件存储的数据库

- API1：打开、关闭数据库（文件）

```
// File Handler for KV-DB
class KVDBHandler {

public:
    // Constructor, creates DB handler
    // @param db_file {const std::string&} path of the append-only file for database.
    KVDBHandler(const std::string& db_file);

    // Closes DB handler
    ~KVDBHandler();
}
```

注意：

1. **若文件存在，则打开数据库；否则，创建新的数据库；**
2. **处理异常：**
    1) **判断输入的文件路径合法性；**
    2) **判断文件是否正常创建；**

文件存储的数据库

- API2：数据库基本操作

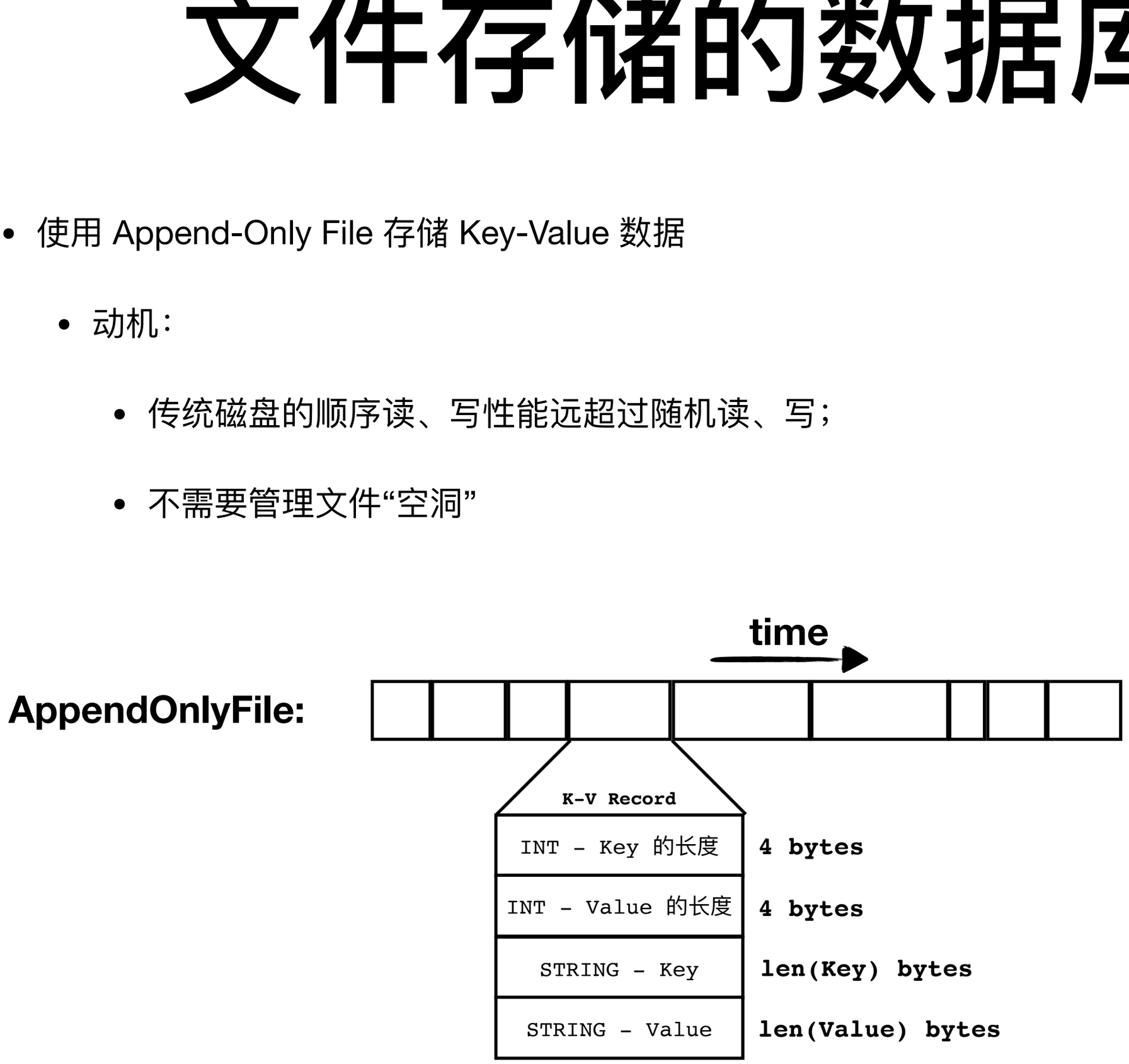
```
// Set the string value of a key
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key of a string
// @param value {const std::string&} the value of a string
// @return {int} return code
int set(KVDBHandler* handler, const std::string& key, const std::string& value);

// Get the value of a key
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key of a string
// @param value {std::string&} store the value of a key after GET executed correctly
// @return {int} return code
int get(KVDBHandler* handler, const std::string& key, std::string& value) const;

// Delete a key
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key to be deleted
// @return {int} return code
int del(KVDBHandler* handler, const std::string& key);
```

注意：

1. **对 Key、Value 做字符串合法性检查**
2. **处理异常操作，如删除不存在的 Key**

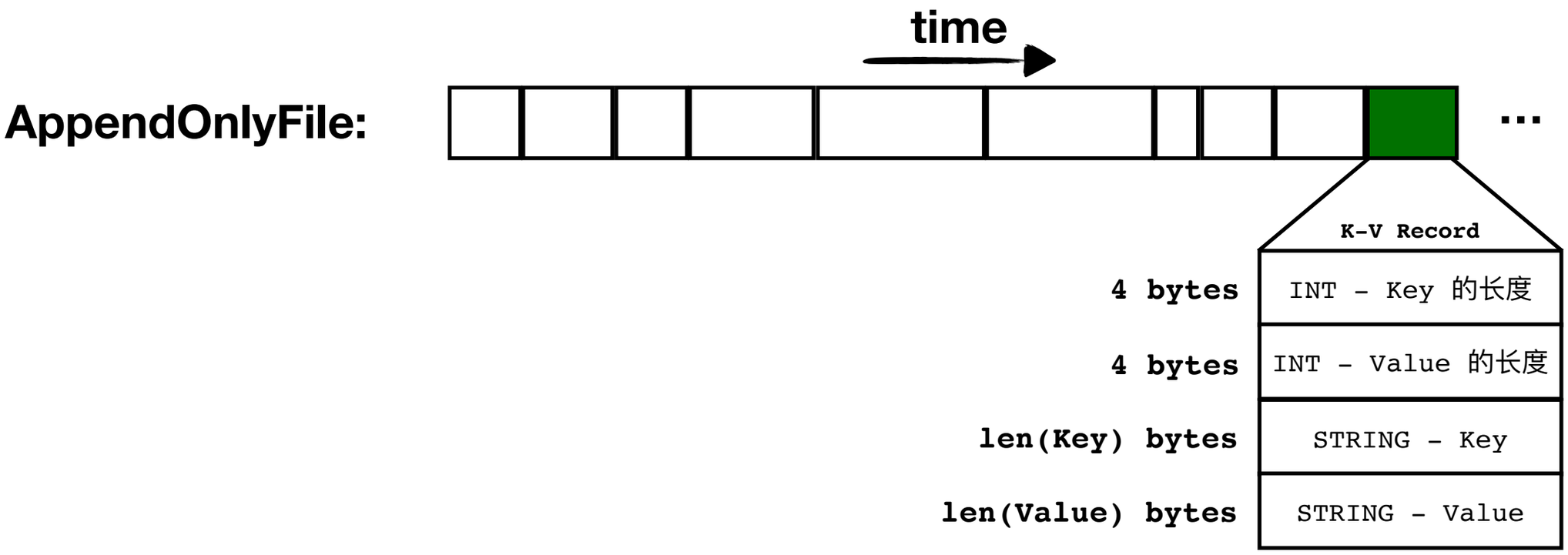
文件存储的数据库

- 使用 Append-Only File 存储 Key-Value 数据
  - 动机：
    - 传统磁盘的顺序读、写性能远超过随机读、写；
    - 不需要管理文件“空洞”



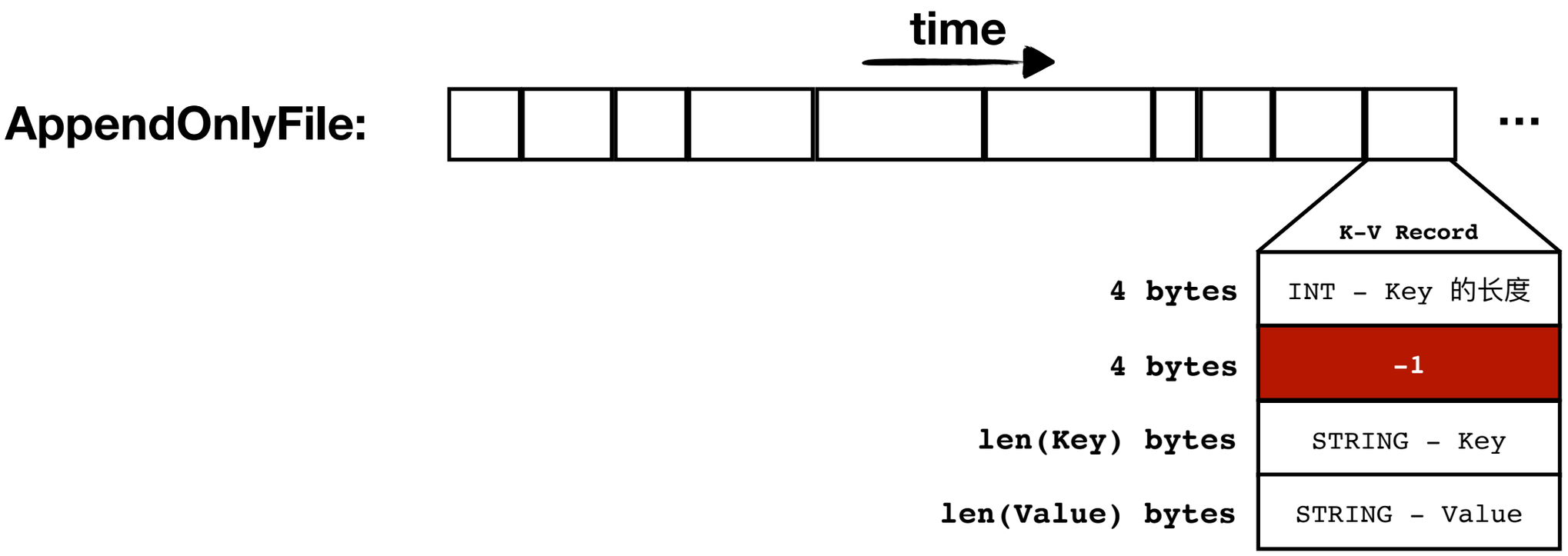
文件存储的数据库

- 使用 Append-Only File 存储 Key-Value 数据
  - 写（Set）操作：
    - 将新的 Key-Value （K-V Record）追加写入（Append）在文件末尾
    - K-V Record 由四个项构成：定长 4 字节存储 Key 的长度；定长 4 字节存储 Value 的长度；变长存储 Key；变长存储 Value；



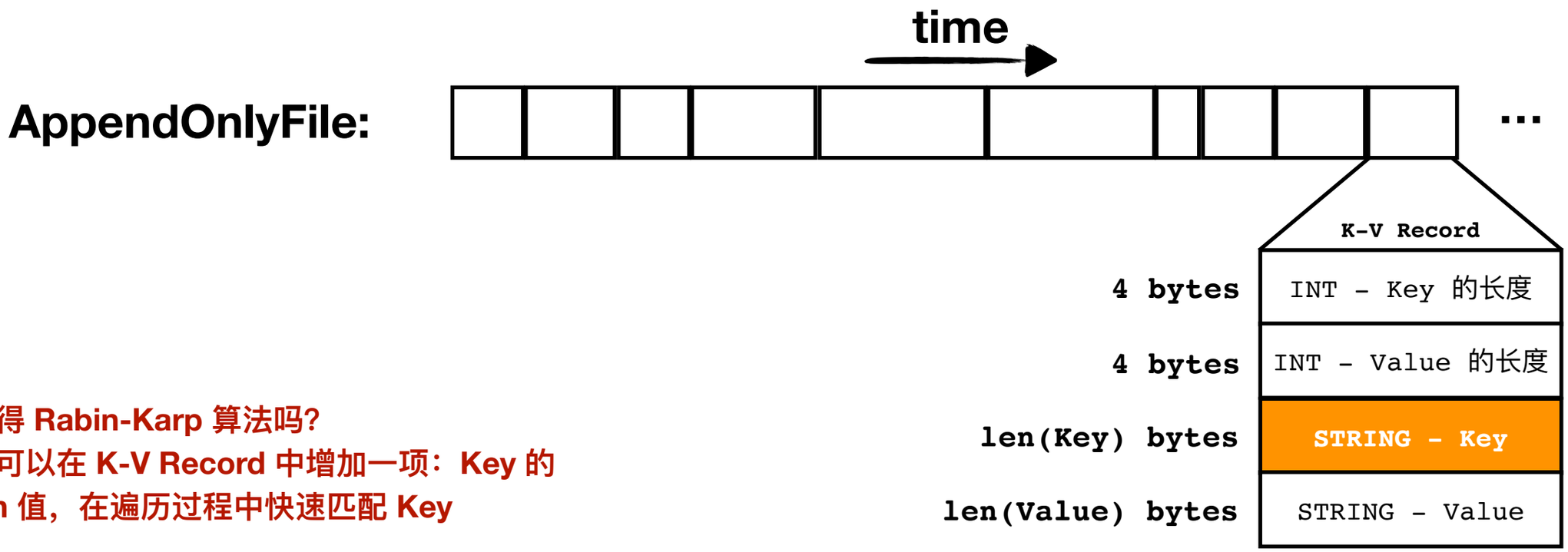
文件存储的数据库

- 使用 Append-Only File 存储 Key-Value 数据
  - 删除（Del）操作：
    - 在文件末尾追加写入一个特殊的 K-V Record，标记为删除
    - 例如，写入一个 Value 的长度为 -1 的特殊 K-V Record，表示该 Key 被删除



文件存储的数据库

- 使用 Append-Only File 存储 Key-Value 数据
  - 读（Get）操作：
    - 顺序遍历文件，比较每个 K-V Record 的 Key，获取满足查询条件的最后一个 Key，并返回其 Value；或返回空；
    - 注意，要处理表示删除操作的 K-V Record；

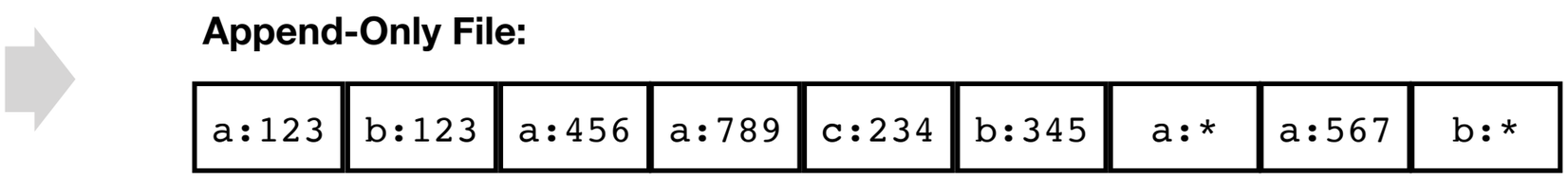


还记得 Rabin-Karp 算法吗?
我们可以在 K-V Record 中增加一项：Key 的 Hash 值，在遍历过程中快速匹配 Key

文件存储的数据库

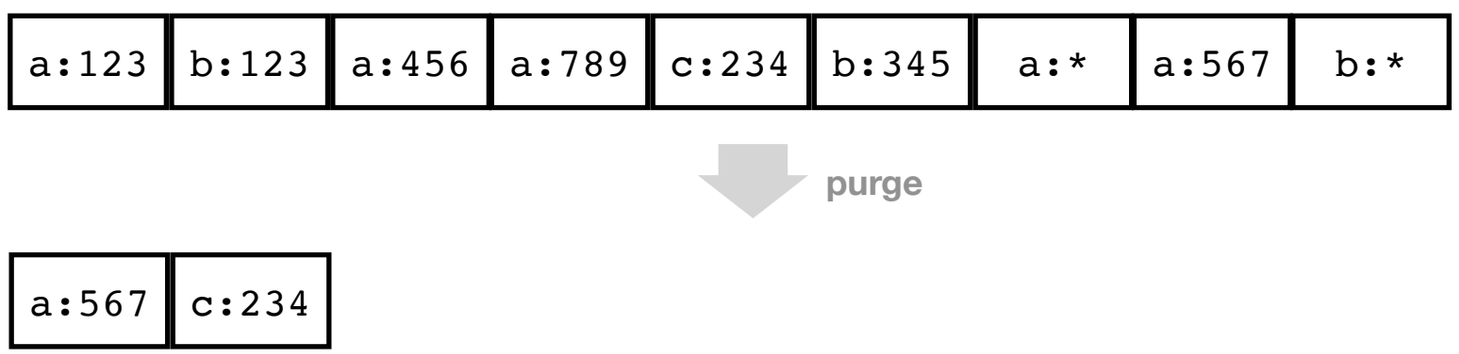
- 使用 Append-Only File 存储 Key-Value 数据
  - 处理文件膨胀问题：
    - 背景：当 Set/Get/Del 操作反复执行后，文件体积会越来越大，但其中有效数据可能很少；

```
SET a 123
SET b 123
SET a 456
GET a
SET a 789
SET c 234
GET b
SET b 345
DEL a
SET a 567
DEL b
```



文件存储的数据库

- 使用 Append-Only File 存储 Key-Value 数据
  - 处理文件膨胀问题：
    - 处理方案：
      - 增加 purge() 操作，整理文件：
        - 合并多次写入（Set）的 Key，只保留最终有效的一项
        - 合并过程中，移除被删除（Del）的 Key
      - 可选：设置一个文件大小上限的阈值，当文件大小达到阈值时，自动触发 perge() 操作



文件存储的数据库

- API3：清理 Append-Only File

```
    // Purge the append-only file for database.
    // @param handler {KVDBHandler*} the handler of KVDB
    // @return {int} return code
    int purge(KVDBHandler* handler);
```

文件存储的数据库

- 处理异常情况
  - 每个 API 都需要处理由输入错误、系统错误 等导致的异常，例如：
    - `KVDBHandler::KVDBHandler(db_file)` - 若 `db_file` 路径不存在，需要返回 `KVDB_INVALID_AOF_PATH`；
    - `set(handler, key, value)` 或 `get(handler, key)` - 若 `key` 长度为0，需返回 `KVDB_INVALID_KEY`；
    - `purge(handler)` - 若清理文件时，没有足够的磁盘空间用于生成新 Append-Only File，需返回 `KVDB_NO_SPACE_LEFT_ON_DEVICES`;

```
// Def of return code
// OK
const int KVDB_OK = 0;
// Invalid path of append-only file
const int KVDB_INVALID_AOF_PATH = 1;
// Invalid KEY
const int KVDB_INVALID_KEY = 2;
// No space on devices for purging.
const int KVDB_NO_SPACE_LEFT_ON_DEVICES = 3;
// ...
```

Stage 1

- 总结：
  - 一个能存储、查询、删除数据的 Key-Value 数据库
  - 基本的软件工程方法

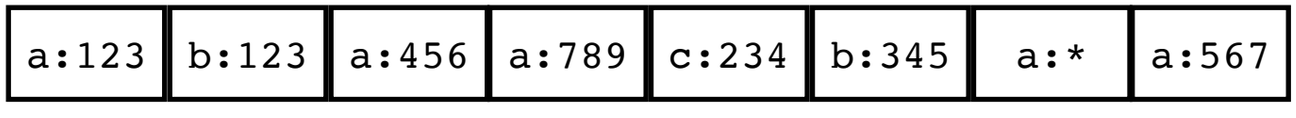
File Based K-V Database

- Stage2：
  - 通过内存索引提升查找速度
    - Append-Only File 的 Key-Offset 索引
- 支持过期删除操作
  - API4: expires

Append-Only File 内存索引

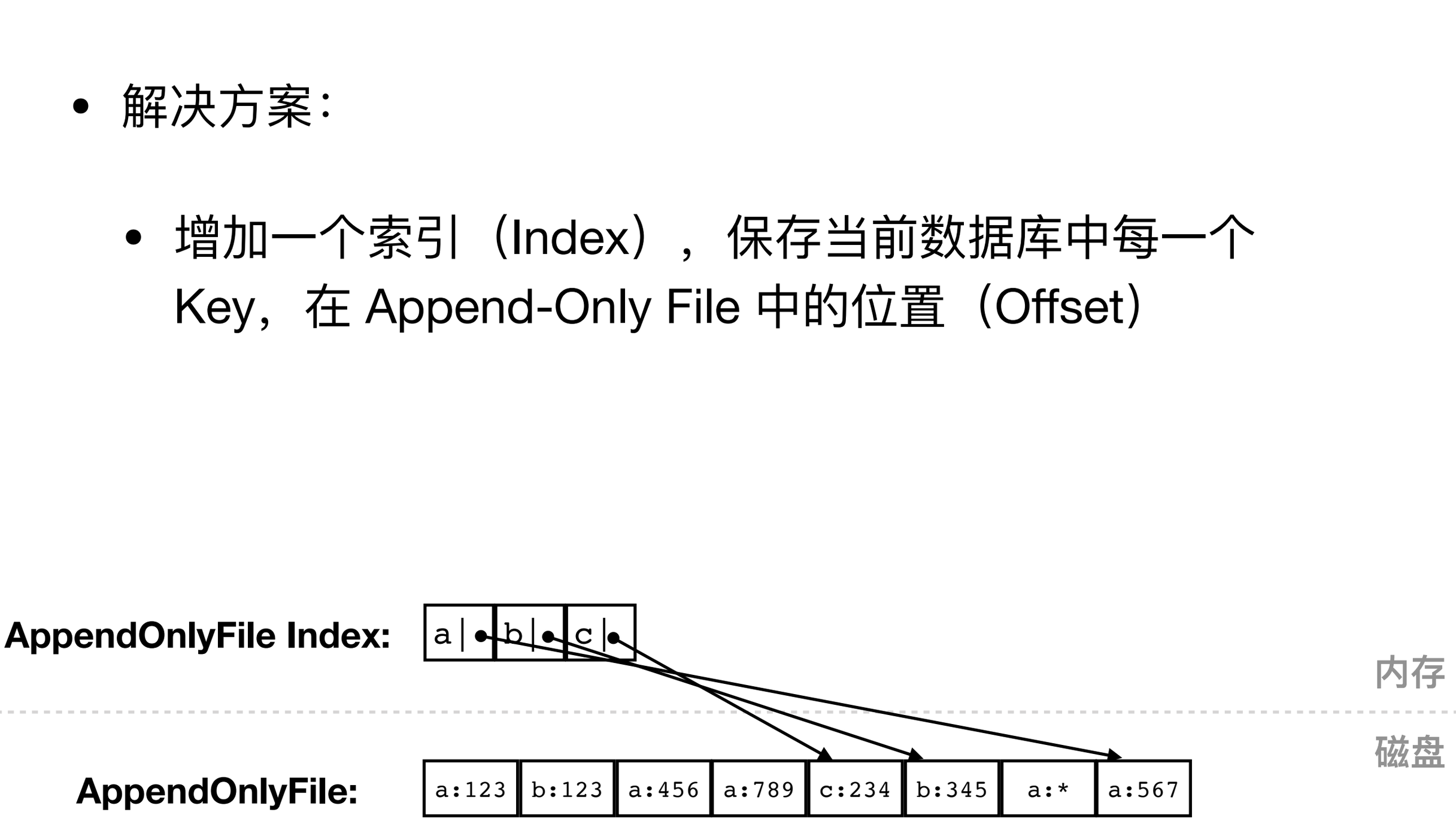
- 动机：
  - 读（GET）操作需要扫描整个 Append-Only File，效率较低

AppendOnlyFile:



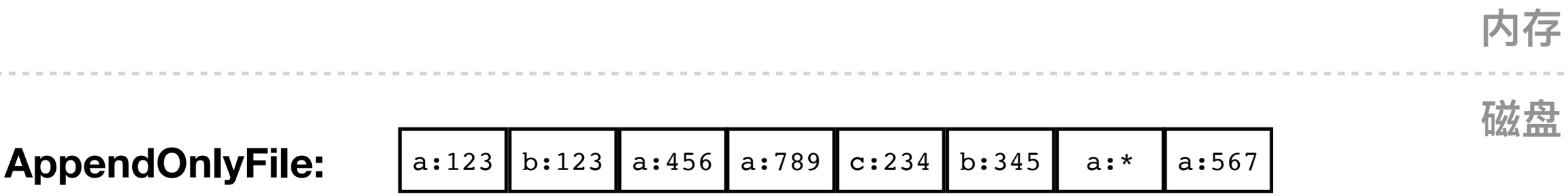
Append-Only File 内存索引

- 解决方案：
  - 增加一个索引（Index），保存当前数据库中每一个 Key，在 Append-Only File 中的位置（Offset）



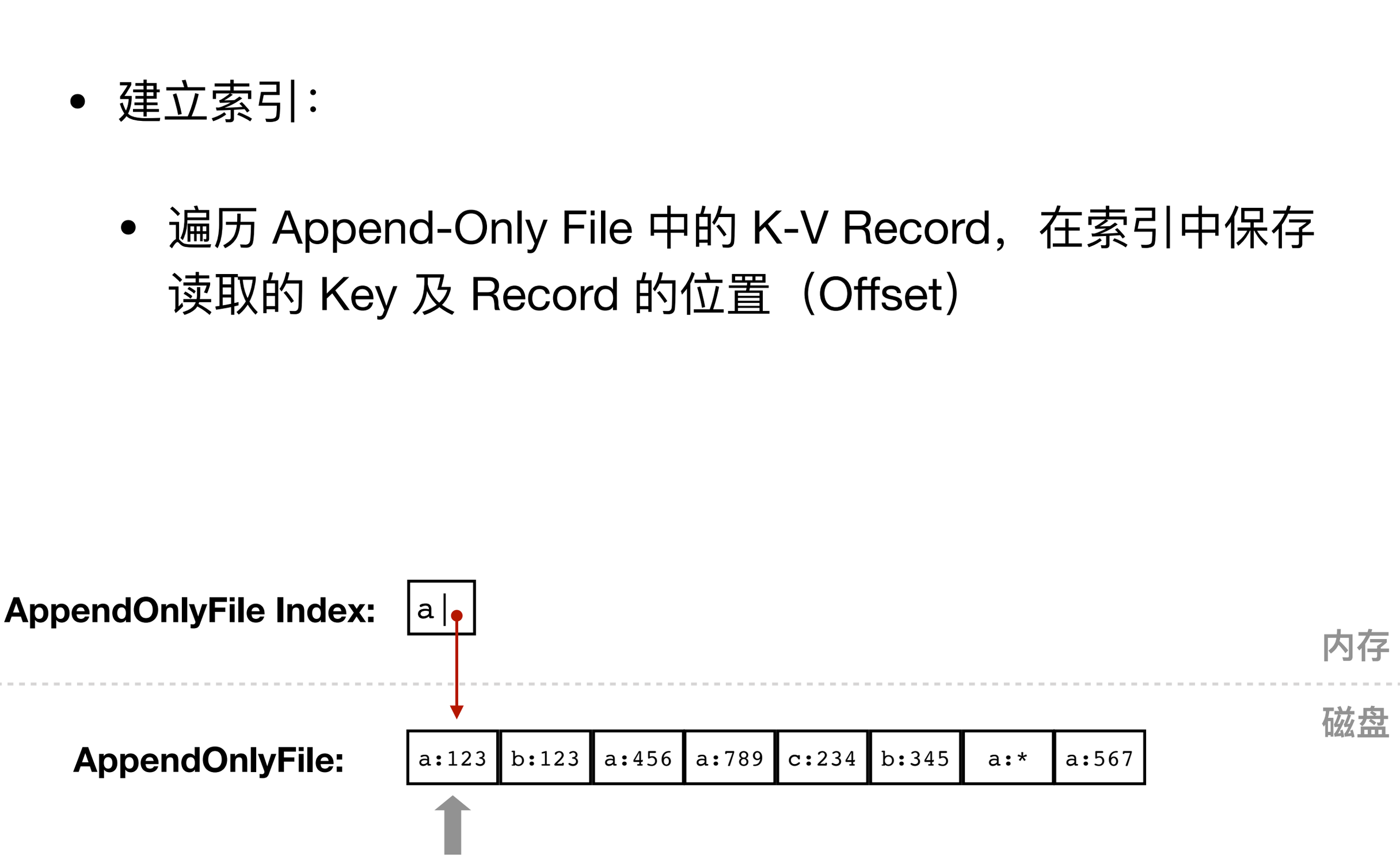
Append-Only File 内存索引

- 建立索引：
  - 遍历 Append-Only File 中的 K-V Record，在索引中保存读取的 Key 及 Record 的位置（Offset）



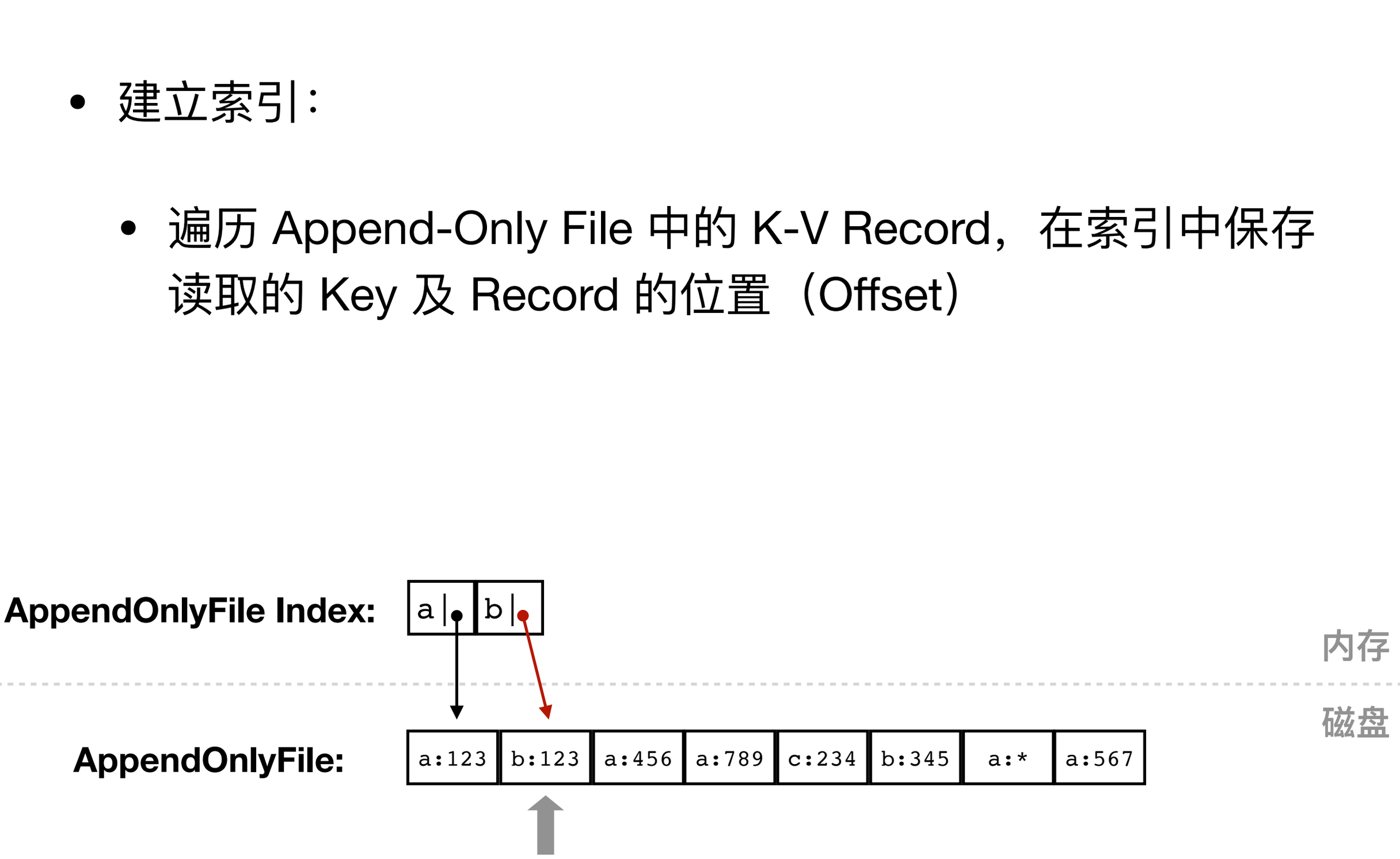
Append-Only File 内存索引

- 建立索引：
  - 遍历 Append-Only File 中的 K-V Record，在索引中保存读取的 Key 及 Record 的位置（Offset）



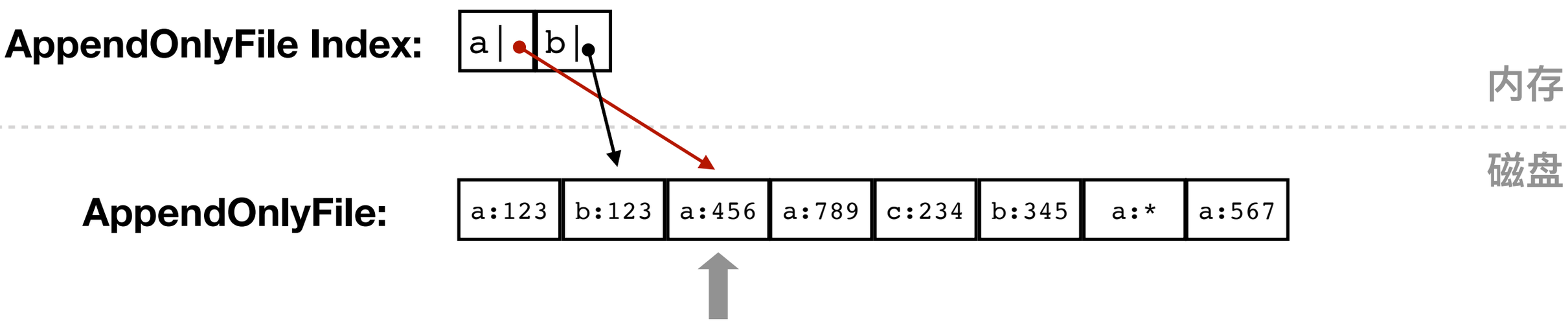
Append-Only File 内存索引

- 建立索引：
  - 遍历 Append-Only File 中的 K-V Record，在索引中保存读取的 Key 及 Record 的位置（Offset）



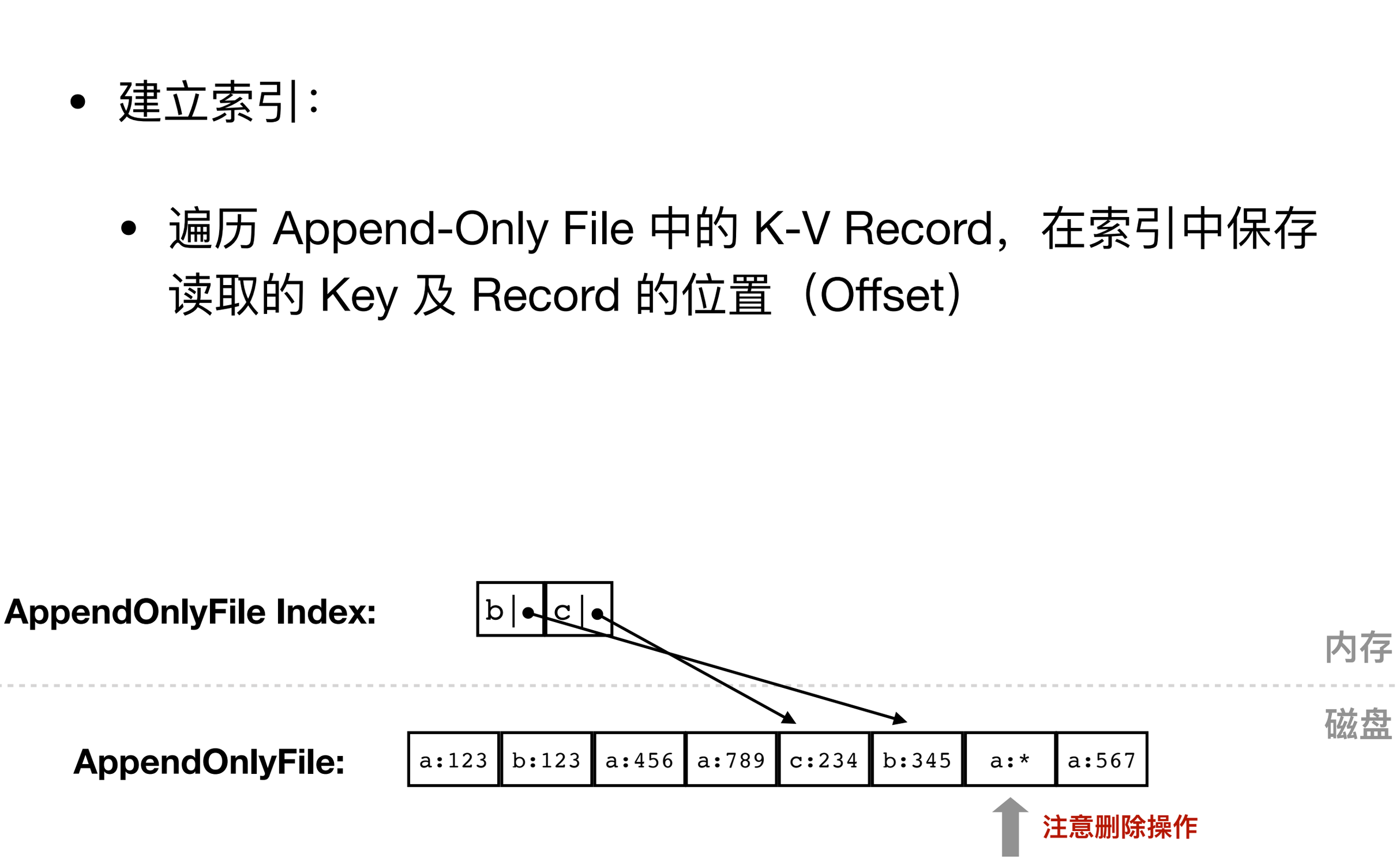
Append-Only File 内存索引

- 建立索引：
  - 遍历 Append-Only File 中的 K-V Record，在索引中保存读取的 Key 及 Record 的位置（Offset）



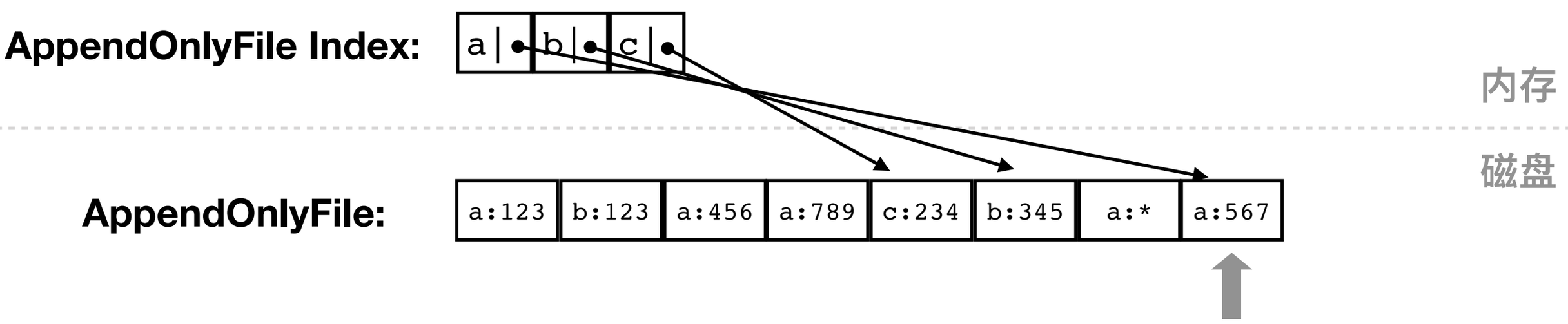
Append-Only File 内存索引

- 建立索引：
  - 遍历 Append-Only File 中的 K-V Record，在索引中保存读取的 Key 及 Record 的位置（Offset）



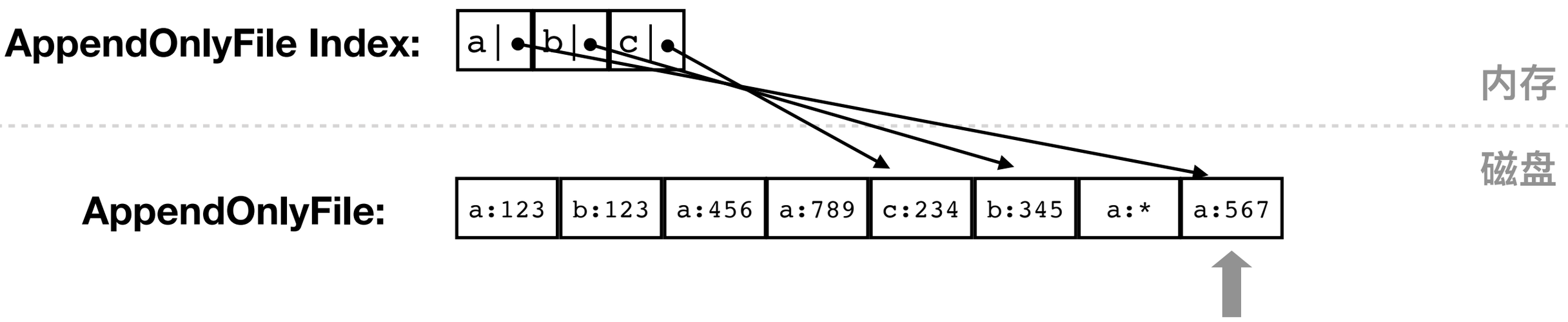
Append-Only File 内存索引

- 建立索引：
  - 遍历 Append-Only File 中的 K-V Record，在索引中保存读取的 Key 及 Record 的位置（Offset）



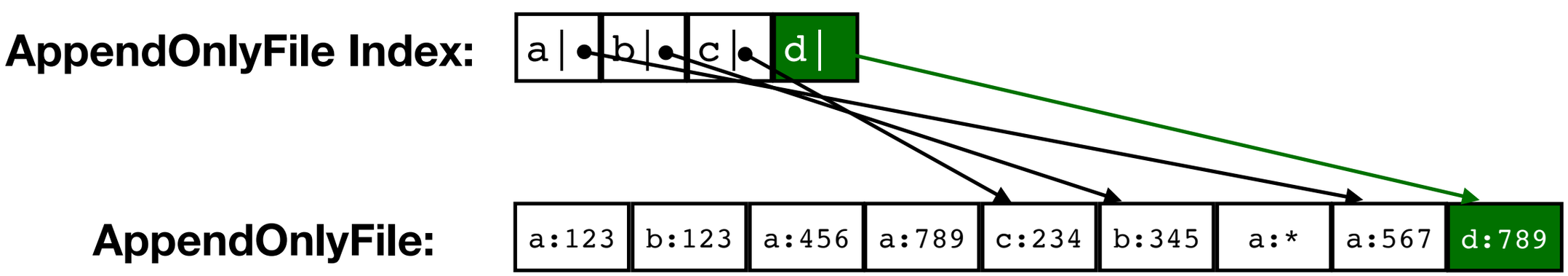
Append-Only File 内存索引

- 使用索引：
  - 读（GET）操作只需要访问索引（Index）：
    - 若 Key 在索引中，则从索引指向的 Append-Only File 中对应的 K-V Record 中读取数据；
    - 若索引中 Key 不存在，则直接返回结果
  - 读操作的时间复杂度从 "O(N)" 降低到 “O(1)"



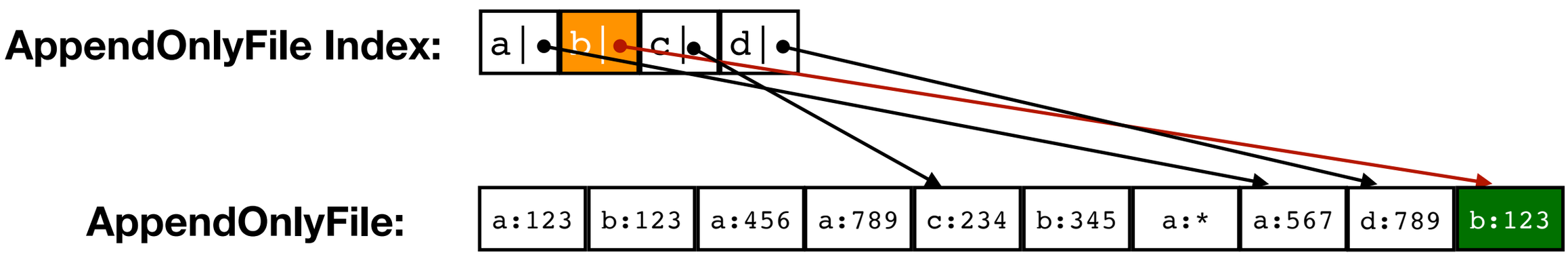
Append-Only File 内存索引

- 维护索引：
  - 写入（SET）操作中，先按原方案将 K-V Record 写入 Append-Only File 中，再修改 Index：
    - 若 Key 不存在，则将它添加到索引中；若 Key 之前已存在于索引中，则修改索引指向的位置（Offset）；



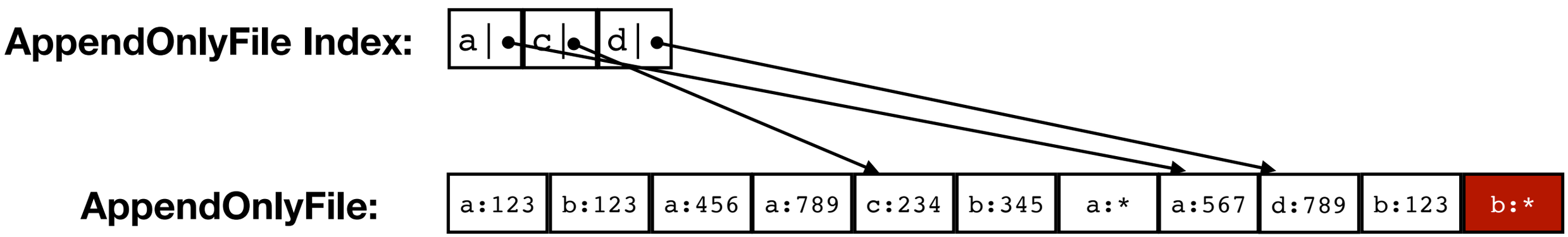
Append-Only File 内存索引

- 维护索引：
  - 写入（SET）操作中，先按原方案将 K-V Record 写入 Append-Only File 中，再修改 Index：
    - 若 Key 不存在，则将它添加到索引中；若 Key 之前已存在于索引中，则修改索引指向的位置（Offset）；
  - 若 Key 已存在，则修改索引中的 Key 指向的位置（Offset）；



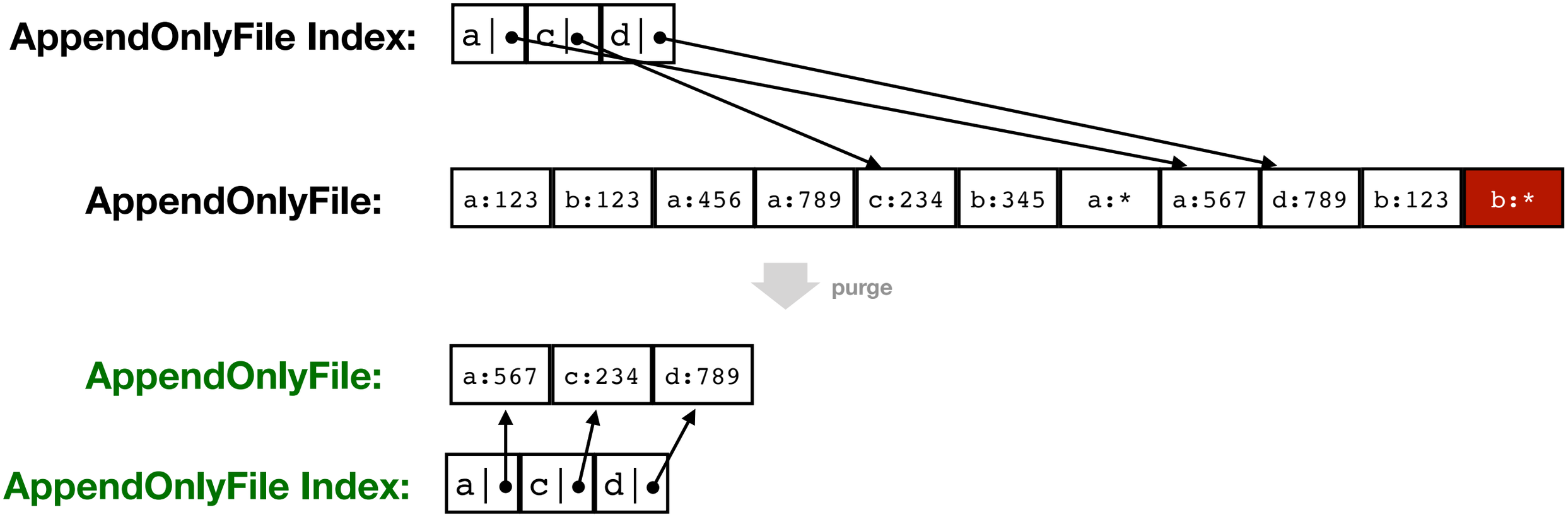
Append-Only File 内存索引

- 维护索引：
  - 删除（DEL）操作中，先按原方案将表示删除操作的特殊的 K-V Record 写入 Append-Only File 中，再将索引中的 Key 删除



Append-Only File 内存索引

- 维护索引：
  - 清理（PURGE）Append-Only File 后，需要重新执行建立索引操作



Append-Only File 内存索引

- 实现索引：
    - 索引需要：
        - 快速插入 Key 和 Offset
        - 快速查找 Key 的 Offset
        - 快速删除 Key

Tips:

1. **HashMap**
2. **Binary Search Tree**

过期删除操作

- API4 定时删除：

  - 过期（EXPIRES）操作，设置 Key 的生存周期（秒），倒计时归零后，自动将 Key 删除

```
    // Set a key's time to live in seconds
    // @param handler {KVDBHandler*} the handler of KVDB
    // @param key {const std::string&} the key
    // @param n {int} life cycle in seconds
    // @return {int} return code
    int expires(KVDBHandler* handler, const std::string& key, int n);
```

Tips:

1. **在 Append-Only File 中增加 K-V Record，记录 Key 的过期时间**
   1) **K-V Record 增加操作类型字段：SET / DEL / EXPIRES**

2. **使用 最小堆（Min-Heap）记录所有 Key 的过期时间：**
   1) **读（GET）操作前，遍历堆顶（Top）元素，将所有已过期的 Key 删除（DEL）；**
   2) **对重复设置过期时间的 Key，需更新 最小堆（Min-Heap）中的过期时间；**
   3) **对已设置过期时间的 Key，过期前执行删除（DEL）操作，或覆盖（SET）操作，需删除最小堆（Min-Heap）中过期时间；**

Stage 2

- 总结：
  - 优化读取效率的 Key-Value 数据库
    - 使用 HashMap 或 BinarySearchTree 实现索引，支持内存索引的重建、维护操作
    - 实现 Min-Heap，支持计时器的重建、维护操作

File Based K-V Database

- Stage3：
  - 实现 LRU Cache 减少磁盘写入次数
  - 支持 List 和 Set 类型
    - API5: lpush / rpush / lpop / rpop / blpop / brpop # List
    - API6: sadd / srem / sunion / sinter / scount # Set

支持 LRU Cache

- 动机：
  - 磁盘操作比内存要慢，特别是支持复杂数据结构后，将复杂数据结构（如 List、Set、Graph）写入磁盘很慢
  - 在内存中短暂存储短期内操作的数据，待内存使用达到上限阈值时，将最久不使用的 Key-Value 写入磁盘
  - LRU (Least Recently Used) Cache

支持 LRU Cache

- 研究资料：
  - 《LRU原理和Redis实现——一个今日头条的面试题》

    https://zhuanlan.zhihu.com/p/34133067
  - 《Leetcode算法题解——LRU缓存机制》

    https://zhuanlan.zhihu.com/p/57733537

支持 LRU Cache

- 方案：
  - 将 LRU Cache 和内存索引结合，在内存中 Cache 部分 Key-Value 对象
  - 为保证内存断电数据不丢失，所有操作必须同时在 Append-Only File 中记录，并可重现

支持更多数据类型

- API5：支持 List 和 Block List 类型

```
// Remove and get a element from list head
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key
// @param value {std::string&} store the element when successful.
// @return {int} return code
int lpop(KVDBHandler* handler, const std::string& key, std::string& value);

// Remove and get a element from list tail
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key
// @param value {std::string&} store the element when successful.
// @return {int} return code
int rpop(KVDBHandler* handler, const std::string& key, std::string& value);

// Add one element to the head of a list
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key
// @param value {const std::string&} the element
// @return {int} return code
int lpush(KVDBHandler* handler, const std::string& key, const std::string& value);

// Add one element to the tail of a list
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key
// @param value {const std::string&} the element
// @return {int} return code
int rpush(KVDBHandler* handler, const std::string& key, const std::string& value);
```

支持更多数据类型

- API5：支持 List 和 Block List 类型

```
    // Remove and get a element from list tail
    // @param handler {KVDBHandler*} the handler of KVDB
    // @param key {const std::string&} the key
    // @param value {std::string&} store the element when successful.
    // @return {int} return code
    int rpop(KVDBHandler* handler, const std::string& key, std::string& value);

    // Remove and get a element from list tail or block until one is available
    // @param handler {KVDBHandler*} the handler of KVDB
    // @param key {const std::string&} the key
    // @param value {std::string&} store the element when successful.
    // @return {int} return code
    int brpop(KVDBHandler* handler, const std::string& key, std::string& value);

    // Get the number of elements in list specified the given key
    // @param handler {KVDBHandler*} the handler of KVDB
    // @param key {const std::string&} the key
    // @return {int} return the number of elements in list, <0 if error
    int llen(KVDBHandler* handler, const std::string& key);
```

支持更多数据类型

- API6：支持 Set 类型

```
// Add one or more member(s) to a set, or update its score if it already exists
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key
// @param members {const std::vector<std::string>&} the set storing members
// @return {int} return code
int sadd(
        KVDBHandler* handler,
        const std::string& key,
        const std::vector<std::string>& members);

// Remove one or more member(s) from a set
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key
// @param members {const std::vector<std::string>&} the removing members of a set
// @return {int} return code
int srem(
        KVDBHandler* handler,
        const std::string& key,
        const std::vector<std::string>& members);
```

支持更多数据类型

- API6：支持 Set 类型

```
// Return the members of a set resulting from the union of all the given sets.
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::vector<std::string> &} the key of all the sets
// @param members {std::vector<std::string>*} stores the result
int zunion(
        KVDBHandler* handler,
        const std::vector<std::string>& key,
        std::vector<std::string>* members);

// Return the members of a set resulting from the intersection of all the given sets.
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::vector<std::string> &} the key of all the sets
// @param members {std::vector<std::string>*} stores the result
int zinter(
        KVDBHandler* handler,
        const std::vector<std::string>& key,
        std::vector<std::string>* members);
```

支持更多数据类型

- API6：支持 Set 类型

```
// Get the number of elements in a set
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key
// @return {int} return the number of elmenets in a set, <0 if error
int zcount(
        KVDBHandler* handler,
        const std::string& key);

// Set a member from a Set's time to live in seconds
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key
// @param n {int} life cycle in seconds
// @return {int} return code
int expires(KVDBHandler* handler, const std::string& key, const std::string& member, int n);
```

Stage 3

- 总结：

  - 理解 LRU Cache 、操作流水日志的概念，以及如何在 Cache 效率和安全间取得平衡

  - 使用双向链表和 HashMap 实现 LRU Cache

  - 实现 List、Set 等结构的序列化与反序列化

File Based K-V Database

- Stage4：
    - 使用 K-V Database 完成一个真实任务
        - 可选：
            - 短视频应用
            - 电商应用
            - 微博
            - 游戏
            - …

File Based K-V Database

- 短视频常见需求
    - 视频：
        - 行为：观看 / 收藏 / 评论 视频
        - 查询：单个商品 播放量 / 收藏量 / 评论数 等
        - 查询：各类视频排行榜
- 社交关系：
    - 行为：用户关注用户
    - 查询：用户好友列表
    - 查询：用户与某用户的共同好友
    - 查询：好友推荐 - 查询与某用户有至少两个共同好友，且互相不是好友的用户

File Based K-V Database

- 短视频常见需求
    - Feed：
        - 行为：好友 收藏视频 / 评论视频
        - 查询：按时间倒序展示所有好友最近 收藏/评论 视频的信息动态
- 用户收藏夹：
    - 行为：用户收藏视频
    - 查询：与某好友共同收藏的商品
    - 查询：都收藏了某商品的好友

File Based K-V Database

- 短视频常见需求
    - 视频播放（实时）：
        - 行为：用户发布评论（弹幕）
        - 查询：当前视频观看人数
        - 查询：按时间顺序实时展示用户评论（弹幕）
- 防攻击（实时）：
    - 查询：1分钟内评论超过2次或5分钟内评论超过5次的用户

Stage 4

- 总结：
  - 业界优秀的 Key-Value Database
  - 阅读 Redis （老版本，例如 1.7.0）的源代码，撰写阅读报告；
  - 《Redis 实战》，https://book.douban.com/subject/26612779/