

# 数据结构实验 (0)

C++编译与高质量工程  
课程设计介绍

# 目录

- 基础开发环境
- 高质量工程建议
- C++ 编译、链接、执行
- 课程设计介绍

# 基础开发环境

- 操作系统：
  - Ubuntu 18.04/20.04 LTS
  - Win10：推荐在 Windows Subsystem for [Linux \(WSL\)](#) 运行
  - Mac：使用 VirtualBox 等虚拟机工具
- 编译器：
  - GCC / G++ 7 以上版本
- 开发工具：
  - Vim / Emacs + tmux
  - VSCode (不推荐)

## 本科四年该做什么

Slide Subtitle

其实只有 3 年，1024 天

- 用同一门“语言”说话：《鸟哥的 Linux 私房菜》
- 扎实的计算机科学基础：
  - C/C++、数据结构、操作系统、计算机网络、计算机体系结构、编译原理、数据库
- 写 5 万行代码：
  - 200 道 ACM / LeetCode 题目
  - 5 场 TopCoder / [Kaggle](#) / 天池 等比赛
- 解决问题的能力：至少一个超过4个月的 Full-Time 实习经历

```

yuhanzou@yuhanzou-MS-7C94: ~ (zsh)      z@z-Parallels-Virtual-Platform: ~ (ssh)      §§1      §§2
64
65     std::unique_ptr<rknn_input[]> inputs(new rknn_input[this->rknn_io_num_.n_input]);
66     memset(inputs.get(), 0, sizeof(rknn_input));
67
68     inputs[0].index = 0;
69     // inputs[0].type = RKNN_TENSOR_INT8;
70     inputs[0].type = RKNN_TENSOR_FLOAT32;
71     inputs[0].fmt = RKNN_TENSOR_NCHW;
72     inputs[0].size = w * h * channels * sizeof(float);
73     inputs[0].buf = data;
74
75     ret = rknn_inputs_set(this->rknn_ctx_, this->rknn_io_num_.n_input, inputs.get());
76     if (0 > ret) {
77         fprintf(
78             stderr,
79             "[%s(%d)] ERROR: rknn_inputs_set failed. [ret=%d]\n",
80             __FILE__, __LINE__,
81             ret);
82         return;
83     }
84
85 #ifdef DEBUG_TIME
86     t = facerecogsdk::get_current_time() - t;
87     fprintf(stdout, "[%s(%d)] DEBUG: rknn_inputs_set. [time=%.2f]\n", __FILE__, __LINE__, t);
88     t = facerecogsdk::get_current_time();
89 #endif
90
91     fprintf(stdout, "[%s(%d)] INFO: rknn_run. \n", __FILE__, __LINE__, t);
92
93     ret = rknn_run(this->rknn_ctx_, NULL);
94     if (0 > ret) {
95         fprintf(
96             stderr,
97             "[%s(%d)] ERROR: rknn_run failed. [ret=%d]\n",
98             __FILE__,
99             __LINE__,
100            ret);
101    return;
102 }
103
104 #ifdef DEBUG_TIME
105     t = facerecogsdk::get_current_time() - t;
106     fprintf(stdout, "[%s(%d)] DEBUG: rknn_run. [time=%.2f]\n", __FILE__, __LINE__, t);
107     t = facerecogsdk::get_current_time();
108
rknn.cpp [+]          102,9          14% CMakeLists.txt          45,1          4%

```

--  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/z/app/git/FaceRecogSDK/build\_rknn  
Scanning dependencies of target FaceRecogSDKCommon  
[ 3%] Building CXX object Common/CMakeFiles/FaceRecogSDKCommon.dir/src/backend.cpp.o  
[ 6%] Building CXX object Common/CMakeFiles/FaceRecogSDKCommon.dir/src/common.cpp.o  
[ 10%] Building CXX object Common/CMakeFiles/FaceRecogSDKCommon.dir/src/common\_iive.cpp.o  
[ 13%] Building CXX object Common/CMakeFiles/FaceRecogSDKCommon.dir/src/inifile.cpp.o  
[ 16%] Building CXX object Common/CMakeFiles/FaceRecogSDKCommon.dir/src/wb\_auth.cpp.o  
[ 20%] Building CXX object Common/CMakeFiles/FaceRecogSDKCommon.dir/src/rknn/rknn.cpp.o  
[ 23%] Linking CXX shared library ../bin/libFaceRecogSDKCommon.so  
[ 23%] Built target FaceRecogSDKCommon  
Scanning dependencies of target FaceRecogSDKFaceQuality  
Scanning dependencies of target FaceRecogSDKFaceTracker  
Scanning dependencies of target FaceRecogSDKFaceRetrieve  
Scanning dependencies of target FaceRecogSDKFaceFeature  
[ 26%] Building CXX object FaceRetrieve/CMakeFiles/FaceRecogSDKFaceRetrieve.dir/src/face\_retrieve.cpp.o  
[ 30%] Building CXX object FaceQuality/CMakeFiles/FaceRecogSDKFaceQuality.dir/src/face\_quality.cpp.o  
[ 33%] Building CXX object FaceFeature/CMakeFiles/FaceRecogSDKFaceFeature.dir/src/face\_feature.cpp.o  
[ 36%] Building CXX object FaceTracker/CMakeFiles/FaceRecogSDKFaceTracker.dir/src/face\_tracker.cpp.o

39 elseif(CMAKE\_SYSTEM\_PROCESSOR MATCHES "i686.\*|i386.\*|x86.\*")  
40 set(X86 1)  
41 set(PROCESSOR\_ARCH "X86")  
42 elseif(CMAKE\_SYSTEM\_PROCESSOR MATCHES "^(aarch64.\*|AARCH64.\*|arm64.\*|ARM64.\*)")  
43 set(AARCH64 1)  
44 set(PROCESSOR\_ARCH "AARCH64")  
45 elseif(CMAKE\_SYSTEM\_PROCESSOR MATCHES "^(arm.\*|ARM.\*)")  
46 set(ARM 1)  
47 set(PROCESSOR\_ARCH "ARM")  
48 elseif(CMAKE\_SYSTEM\_PROCESSOR MATCHES "^(powerpc|ppc)64le")  
49 set(PPC64LE 1)  
50 set(PROCESSOR\_ARCH "PPC64LE")  
51 elseif(CMAKE\_SYSTEM\_PROCESSOR MATCHES "^(powerpc|ppc)64")  
52 set(PPC64 1)  
53 set(PROCESSOR\_ARCH "PPC64")  
54 elseif(CMAKE\_SYSTEM\_PROCESSOR MATCHES "^(mips.\*|MIPS.\*)")  
55 set(MIPS 1)  
56 set(PROCESSOR\_ARCH "MIPS")  
57 endif()  
58  
59 if(APPLE)

[0] 1:buildroot 2:SDKExample 3:SDKTest- 4:ADB 5:bash# 6:SDKCore\* 7:bash# 8:vim "z-Parallels-Virtual-P" 12:24 10-9月 -21

# 高质量工程建议

- 代码规范
- 单元测试与黑盒测试
- 日志

# 代码规范

## Google C++ 编程规范

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

[《Google的C++编码规范》阅读随笔 —— 第一章 头文件 - 简书](#)  
2019年3月10日 - 《Google的C++编码规范》阅读随笔 —— 第一章 头文件 2019.03.10  
14:53:44字数1047阅读19 第一章 头文件 每一个.c文件都有一个对应的.h文件,例外的一...  
简书社区 - 百度快照

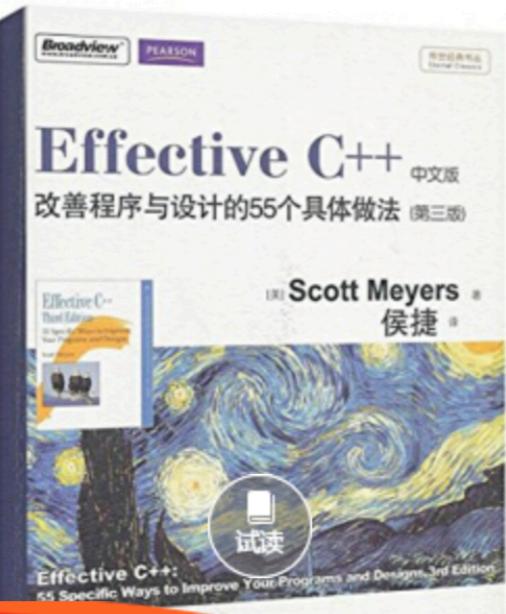
# 代码规范

## 函数参数的规范

```
1 void plus_good(int a, int b, int* c) {
2     *c = a + b;
3 }
4
5 void plus_bad(int a, int b, int& c) {
6     c = a + b;
7 }
8
9 int main(int, char**)
10    int a = 3;
11    int b = 5;
12    int c = 0;
13
14    plus_good(a, b, &c);
15
16    plus_bad(a, b, c);
17
18    return 0;
19 }
```

# 代码规范

## 《Effective C++》，侯捷老师翻译的版本



Effective C++ 中文版  
改善程序与设计的55个具体做法 (第三版)  
Scott Meyers 侯捷

自营每满100减50 (9.6-9.12)

试读

自定义封面

商品评价 10万+

京东价 ¥ 89.00 [10折] [定价 ¥89.00] 降价通知

促销 多买优惠 满1件，总价打6折

满减 每满100元，可减50元现金 详情>>

"满减""多买优惠"仅可任选其一，可在购物车更改

增值业务 助力环保，传递知识，旧书换新

排行榜 入选『讲解详细的编程书籍TOP榜』 详情>>

配送至 广东深圳市南山区蛇口街道 ✓ 有货

京东物流 京准达 | 211限时达 | 京尊达 ✓

由 京东 发货，并提供售后服务。23:10前下单，预计明天(09月11日)送达

重量 0.54kg

服务支持 放心购 上门换新 | 破损包退换 | 闪电退款  
可配送海外49元免基础运费

# 代码规范

## 智能指针

- C++ 没有自动回收内存的机制，每一次 malloc / new 的动态内存必须手动 free/delete
- 智能指针：利用了栈的机制，自动负责释放所指向的对象
  - RAII 基本原理
  - 两种基本的智能指针：
    - unique\_ptr<T>
    - shared\_ptr<T>

# 单元测试与黑盒测试

- 单元测试
  - 针对一个函数或者一个代码片段进行的测试，在反复迭代中，保证代码段的正确性和异常处理
  - **单元测试不是测试，其实是开发过程**
  - 分枝覆盖的完整性
- 黑盒测试
  - 针对整体功能的测试
    - 产品功能
    - 产品性能

```
1 int div(int a, int b) {  
2     if (b == 0) {  
3         return 0;  
4     }  
5     return a / b;  
6 }  
7  
8 int main(int, char**) {  
9     // Unit Test  
10    assert(2 == plus(10, 5));  
11    assert(0 == plus(3, 0));  
12  
13    return 0;  
14 }
```

# 单元测试与黑盒测试

- 推荐的单元测试工具：
  - gtest & gmock
    - <https://github.com/google/googletest>
    - <Unit Testing C++ with Google Test>
      - <https://blog.jetbrains.com/rscpp/2015/09/01/unit-testing-google-test/>
  - 《从头到脚说单测——谈有效的单元测》
    - <https://mp.weixin.qq.com/s/okmWMOeBm7cClZ1zzFr4KQ>

# 日志

- 任何程序都需要记录运行期间的信息，用于：

- 记录流水信息

- 排查可能的异常

- 捕获程序错误

```
74
75     ret = rknn_inputs_set(this->rknn_ctx_, this->rknn_io_num_.n_input, inputs.get());
76     if (0 > ret) {
77         fprintf(
78             stderr,
79             "[%s(%d)] ERROR: rknn_inputs_set failed. [ret=%d]\n",
80             __FILE__,
81             __LINE__,
82             ret);
83         return;
84     }
85 #ifdef DEBUG_TIME
86     t = facerecogsdk::get_current_time() - t;
87     fprintf(stdout, "[%s(%d)] DEBUG: rknn_inputs_set. [time=%.2f]\n", __FILE__, __LINE__, t);
88     t = facerecogsdk::get_current_time();
89 #endif
90
91     fprintf(stdout, "[%s(%d)] INFO: rknn_run. \n", __FILE__, __LINE__, t);
92
93     ret = rknn_run(this->rknn_ctx_, NULL);
94     if (0 > ret) {
95         fprintf(
96             stderr,
97             "[%s(%d)] ERROR: rknn_run failed. [ret=%d]\n",
98             __FILE__,
99             __LINE__,
100            ret);
101    return;
102 }
```

# 日志

- 日志的要素：
  - 时间
  - 日志级别： DEBUG / INFO / WARN / ERROR / FATAL
  - 日志内容
- 推荐的 C++ 日志库：
  - glog
    - <https://github.com/google/glog>
  - log4cxx
    - [https://logging.apache.org/log4cxx/latest\\_stable/](https://logging.apache.org/log4cxx/latest_stable/)
  - 自己写一个：
    - 《C++如何写一个简单Logger?》
      - <https://www.zhihu.com/question/293863155/answer/576148854>

# C++ 编译、链接与执行

- C++ 编译、链接基本原理
- 使用 Makefile

# C++ 编译、链接基本原理

```
1 #include <cstdio>
2
3 int plus(int a, int b) {
4     return a + b;
5 }
6
7 int main(int, char**)
8 {
9     int s = plus(2, 3);
10    #ifdef OUTPUT
11        fprintf(stdout, "2+3=%d\n", s);
12    #endif
13    return 0;
14 }
```

```
~  
~  
~  
~  
~  
~  
~  
~  
~
```

**test.cpp**

"test.cpp" 13L, 186C

---

```
z@z-Parallels-Virtual-Platform:~/Documents$ g++ test.cpp -o test
z@z-Parallels-Virtual-Platform:~/Documents$ ./test
z@z-Parallels-Virtual-Platform:~/Documents$ g++ test.cpp -o test -DOUTPUT
z@z-Parallels-Virtual-Platform:~/Documents$ ./test
2+3=5
z@z-Parallels-Virtual-Platform:~/Documents$ █
```

# C++ 编译、链接基本原理

- `g++ test.cpp -o test` 一般包含四个步骤

- 预处理：

- Macro 展开、添加编译器指令、删除注释等
- g++ -E test.cpp -o test.ii

```
z@z-Parallels-Virtual-Platform:~$ g++ -E test.cpp -o test.ii
z@z-Parallels-Virtual-Platform:~$ vim test.ii
z@z-Parallels-Virtual-Platform:~$ head -n 20 test.ii
# 1 "test.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "test.cpp"
# 1 "/usr/include/c++/7/cstdio" 1 3
# 39 "/usr/include/c++/7/cstdio" 3

# 40 "/usr/include/c++/7/cstdio" 3

# 1 "/usr/include/x86_64-linux-gnu/c++/7/bits/c++config.h" 1 3
# 229 "/usr/include/x86_64-linux-gnu/c++/7/bits/c++config.h" 3

# 229 "/usr/include/x86_64-linux-gnu/c++/7/bits/c++config.h" 3
namespace std
{
    typedef long unsigned int size_t;
    typedef long int ptrdiff_t;
```

# C++ 编译、链接基本原理

- `g++ test.cpp -o test` 一般包含四个步骤

- 编译：

- 将预编译后的程序编译成汇编语言
  - g++ -S test.ii -o test.s

```
1  .file  "test.cpp"
2  .text
3  .globl _Z4plusii
4  .type  _Z4plusii, @function
5 _Z4plusii:
6 .LFB0:
7   .cfi_startproc
8   pushq %rbp
9   .cfi_def_cfa_offset 16
10  .cfi_offset 6, -16
11  movq %rsp, %rbp
12  .cfi_def_cfa_register 6
13  movl %edi, -4(%rbp)
14  movl %esi, -8(%rbp)
15  movl -4(%rbp), %edx
16  movl -8(%rbp), %eax
17  addl %edx, %eax
18  popq %rbp
19  .cfi_def_cfa 7, 8
20  ret
21  .cfi_endproc
22 .LFE0:
23  .size  _Z4plusii, .-_Z4plusii
24  .globl main
25  .type  main, @function
26 main:
27 .LFB1:
28   .cfi_startproc
```



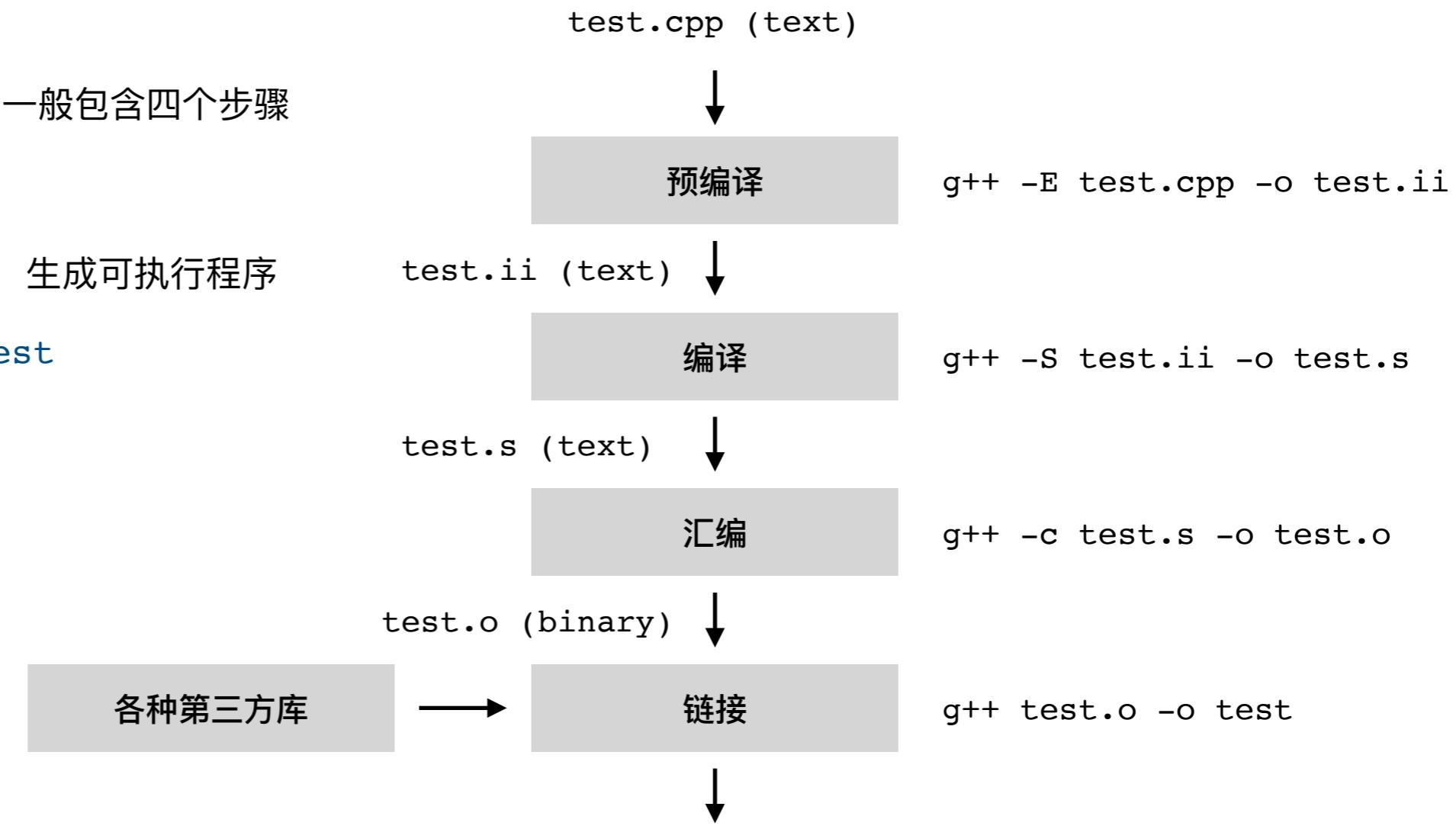
# C++ 编译、链接基本原理

- `g++ test.cpp -o test` 一般包含四个步骤

- 链接：

- 链接静态库和动态库，生成可执行程序

- g++ test.o -o test



- 思考这些问题：

- 为什么 C++ 代码要分成头文件(.h) 和 程序文件 (.c/.cpp)
  - 宏、内联函数展开、模版展开 分别是什么阶段做的？
  - 当调用系统库（例如 stdio.h），或者第三方库的时候，在 每个阶段分别发生什么？
  - 如果要在 Linux 下编译 Windows 的可执行程序，或者在 x86 CPU 上编译 ARM 可执行程序，上述哪些阶段会有差异？

# C++ 编译、链接基本原理

- 为什么需要分成头文件 (.h) 和 程序文件 (.c/.cpp)

test.cpp

```
1 #include <cstdio>
2
3 int plus(int a, int b);
4
5 int main(int, char**)
6 {
7     int s = plus(2, 3);
8     fprintf(stdout, "2+3=%d\n", s);
9     return 0;
10 }
```

test2.cpp

```
1 int plus(int a, int b) {
2     return a + b;
3 }
```

编译、运行：

```
z@z-Parallels-Virtual-Platform:~/Documents$ gcc test.cpp test2.cpp -o test
z@z-Parallels-Virtual-Platform:~/Documents$ ./test
2+3=5
z@z-Parallels-Virtual-Platform:~/Documents$
```

# C++ 编译、链接基本原理

- 为什么需要分成头文件 (.h) 和 程序文件 (.c/.cpp)
  - 基本背景：
    - 在编译大型程序的时候，预编译、编译、汇编 这三个步骤是独立的；
    - 在 汇编 中，需要为程序的静态段分配空间，确定地址（入口）；
    - 在 链接 中，将各种目标码 (.o) 里面的函数与变量提取出来，放到对应的二进制区段里面，解决符号与替换问题，就完成链接与绑定，生成最终目标文件：可执行文件，或动、静态库。

# C++ 编译、链接基本原理

- 为什

```
z@z-Parallels-Virtual-Platform:~$ gcc -c test.cpp -o test.o
z@z-Parallels-Virtual-Platform:~$ gcc -c test2.cpp -o test2.o
z@z-Parallels-Virtual-Platform:~$
z@z-Parallels-Virtual-Platform:~$ objdump -tT test.o

test.o:      file format elf64-x86-64

objdump: test.o: not a dynamic object
SYMBOL TABLE:
0000000000000000 l    df  *ABS*  0000000000000000 test.cpp
0000000000000000 l    d   .text  0000000000000000 .text
0000000000000000 l    d   .data  0000000000000000 .data
0000000000000000 l    d   .bss   0000000000000000 .bss
0000000000000000 l    d   .rodata 0000000000000000 .rodata
0000000000000000 l    d   .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l    d   .eh_frame 0000000000000000 .eh_frame
0000000000000000 l    d   .comment 0000000000000000 .comment
0000000000000000 g    F   .text  00000000000046 main
0000000000000000           *UND*  0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000           *UND*  0000000000000000 _Z4plusii
0000000000000000           *UND*  0000000000000000 stdout
0000000000000000           *UND*  0000000000000000 fprintf

DYNAMIC SYMBOL TABLE:
no symbols
```

# C++ 编译、链接基本原理

- 为什么需要分成头文件 (.h) 和 程序文件 (.c/.cpp)
  - 在 test.cpp 中，对 int plus(int, int) 的 声明，是可以复用的，这个是 头文件的 基本背景
  - 我们通常把以下内容放在头文件中：
    - inline 函数 声明+实现
    - class / function 声明
    - const 常量 声明
  - 参考资料：
    - 《为什么C/C++要分为头文件和源文件？》
    - <https://www.zhihu.com/question/280665935/answer/649503865>

# C++ 编译、链接基本原理

- 更多拓展资料：
  - 静态库、动态库的区别与作用
    - 《Linux的so文件到底是干嘛的？浅析Linux的动态链接库》
      - <https://zhuanlan.zhihu.com/p/235551437>
    - C++ 内存管理与可执行文件的分段
      - 《C/C++内存管理》专栏，建议读(1)、(4)和(5)
        - [https://www.zhihu.com/column/c\\_1277937360727257088](https://www.zhihu.com/column/c_1277937360727257088)

# 使用 Makefile

- 《为什么编译c/c++要用makefile，而不是直接用shell呢？》
  - <https://www.zhihu.com/question/461953861/answer/1914452432>
- 《跟我一起写 Makefile》
  - <https://seisman.github.io/how-to-write-makefile/overview.html>

# 课程设计介绍

<https://riak.com/assets/bitcask-intro.pdf>

## Bitcask

*A Log-Structured Hash Table for Fast Key/Value Data*

# 课程要求

- 一个项目 5-10 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 4 小时以上个人学习、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

# KV 数据库

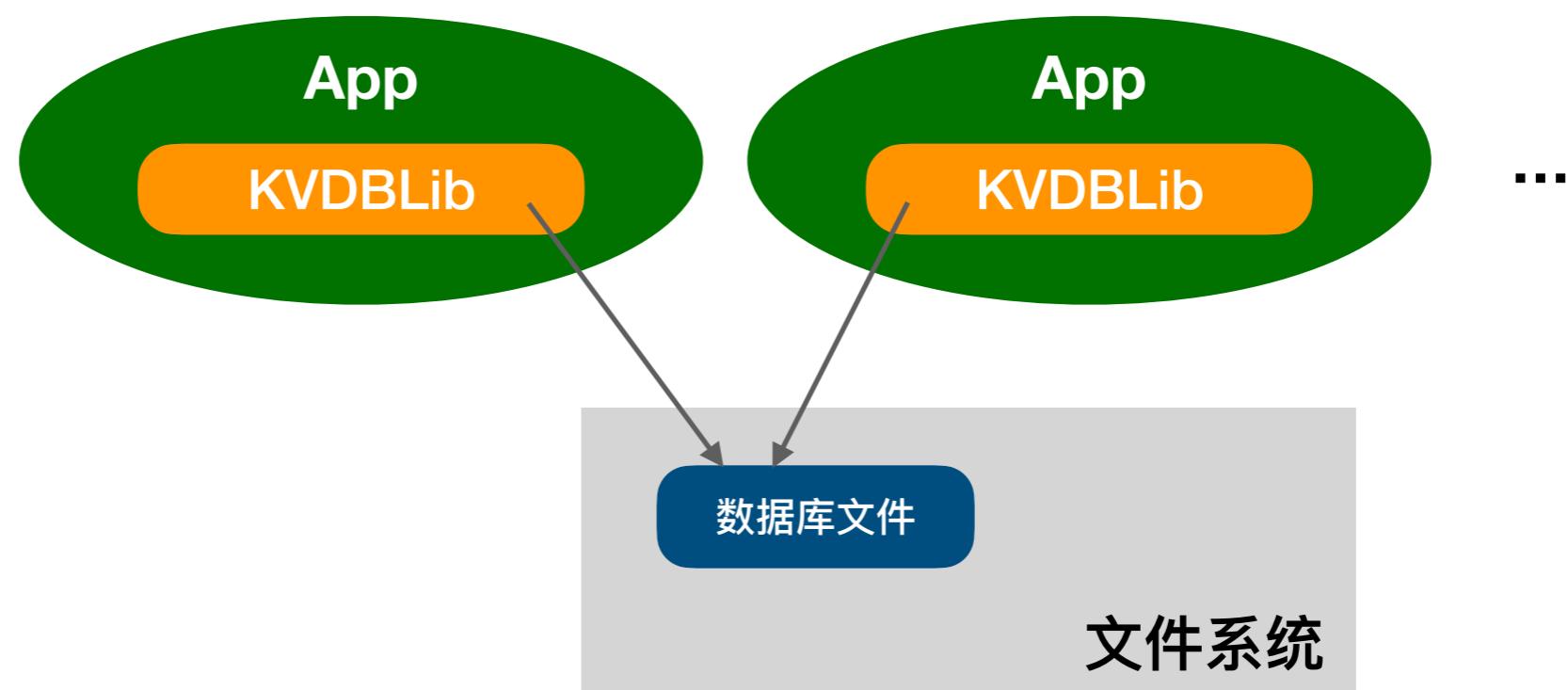
- [https://en.wikipedia.org/wiki/Key-value\\_database](https://en.wikipedia.org/wiki/Key-value_database)
- Key-Value DB & Relational DB:
  - NoSQL & SQL, Primary Key Only Access DB
  - Schemaless
  - 天然适合分布式

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



# 文件存储的数据库

- Pre-Stage：学习编程规范、了解 K-V 数据库的应用、搭建基于 Ubuntu 的开发环境、有能力可以阅读论文
- 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

# 文件存储的数据库

- 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;

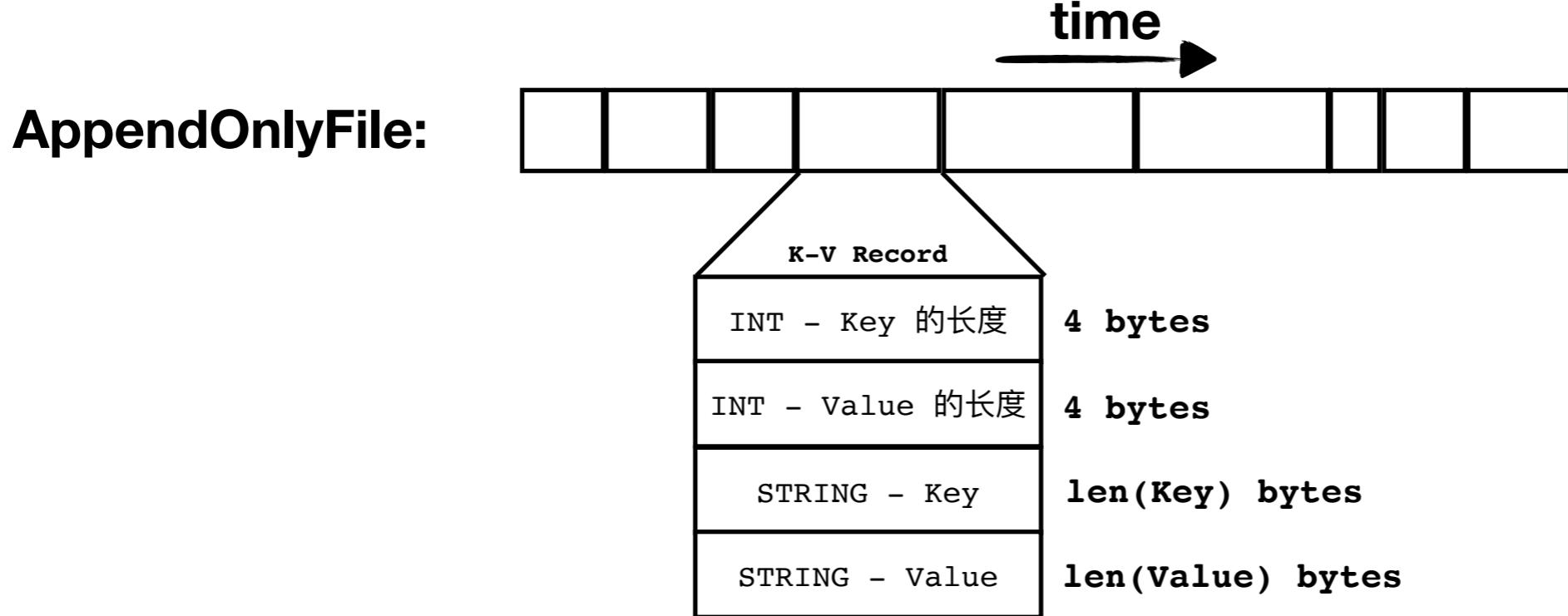
// Delete a key
// @param handler {KVDBHandler*} the handler of KVDB
// @param key {const std::string&} the key to be deleted
// @return {int} return code
int del(KVDBHandler* handler, const std::string& key);
```

## 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

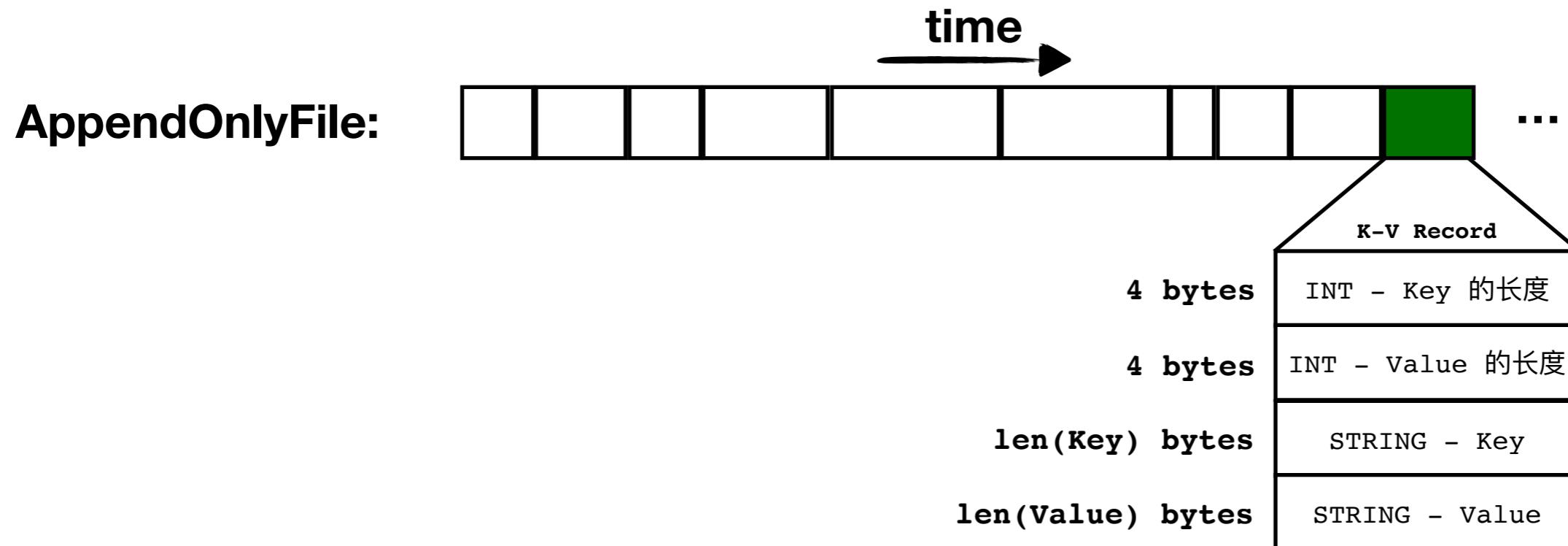
# 文件存储的数据库

- 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”



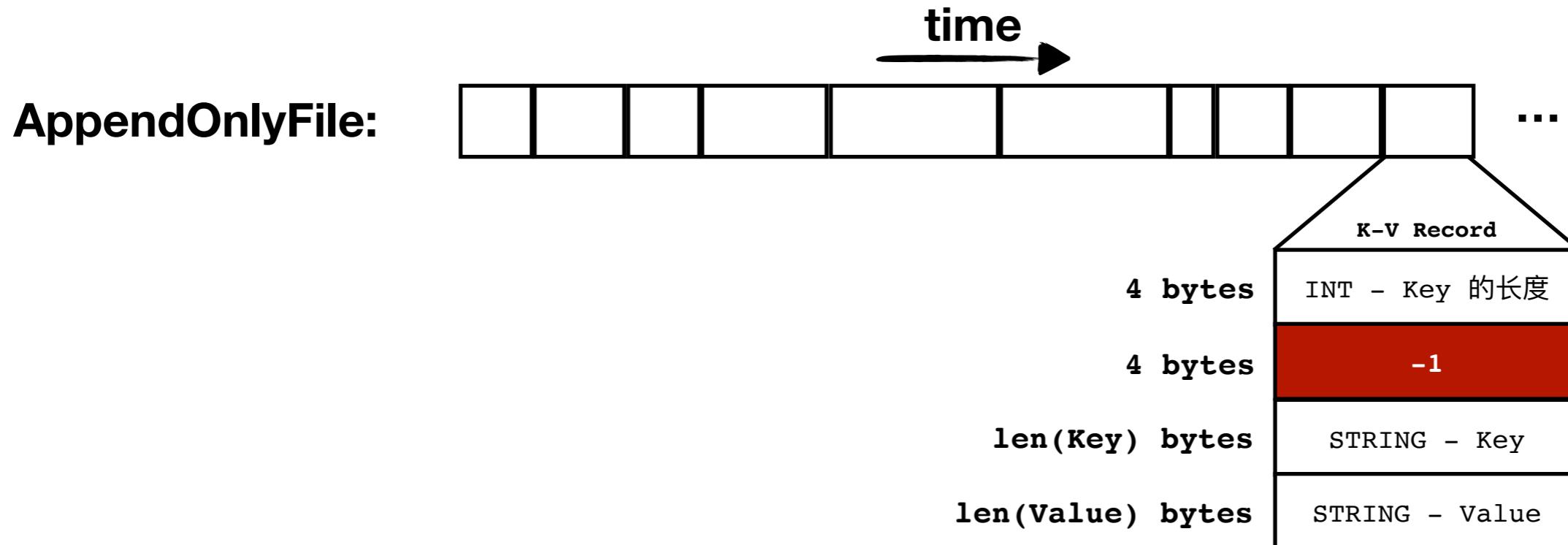
# 文件存储的数据库

- 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；



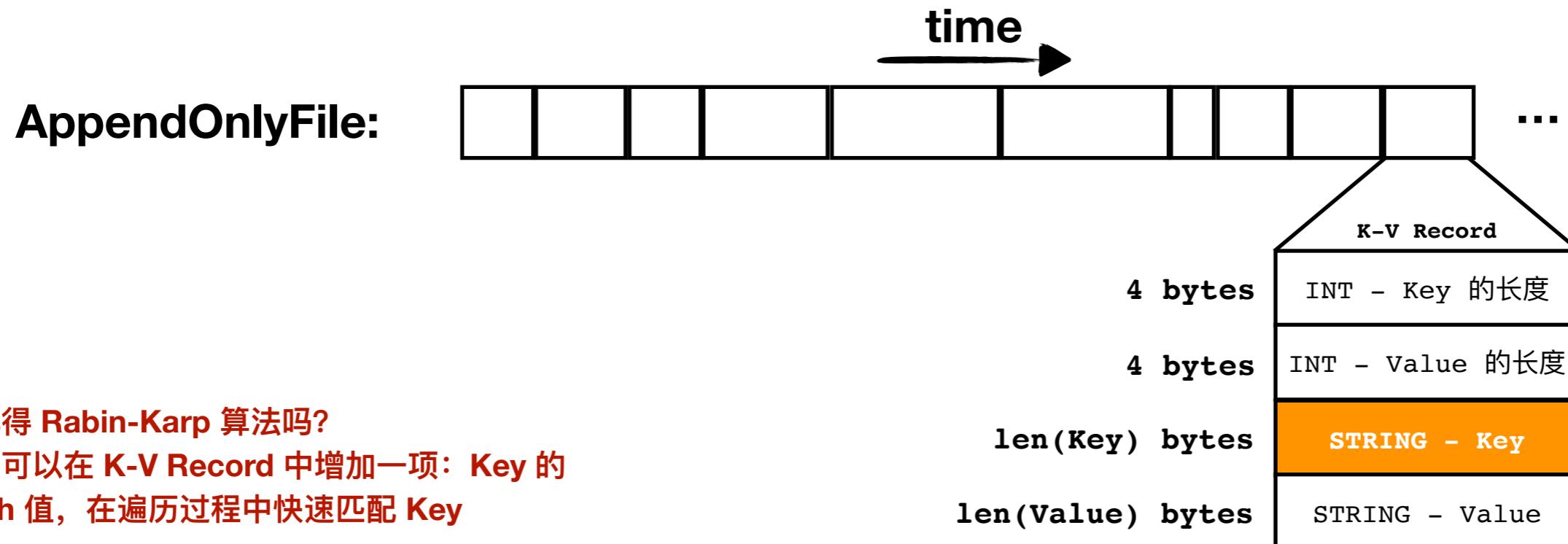
# 文件存储的数据库

- 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；



# 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
```

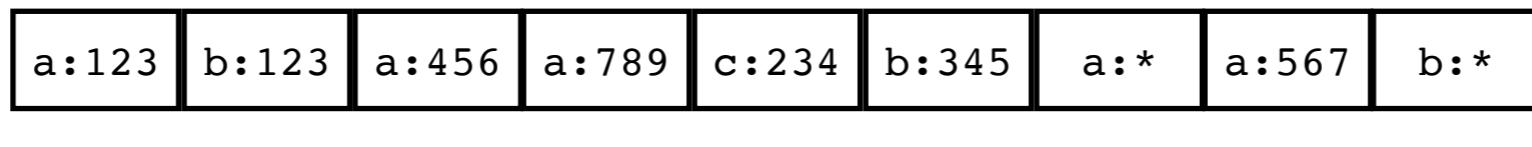


Append-Only File:

a:123	b:123	a:456	a:789	c:234	b:345	a:*	a:567	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 `purge()` 操作



purge

a:567	c:234
-------	-------

# 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

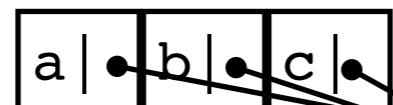
**AppendOnlyFile:**

a:123	b:123	a:456	a:789	c:234	b:345	a:*	a:567
-------	-------	-------	-------	-------	-------	-----	-------

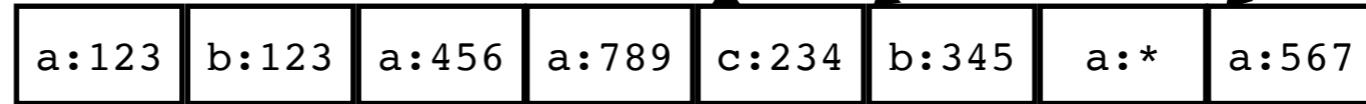
# Append-Only File 内存索引

- 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)

**AppendOnlyFile Index:**



**AppendOnlyFile:**



内存

磁盘

# 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）

**AppendOnlyFile:**

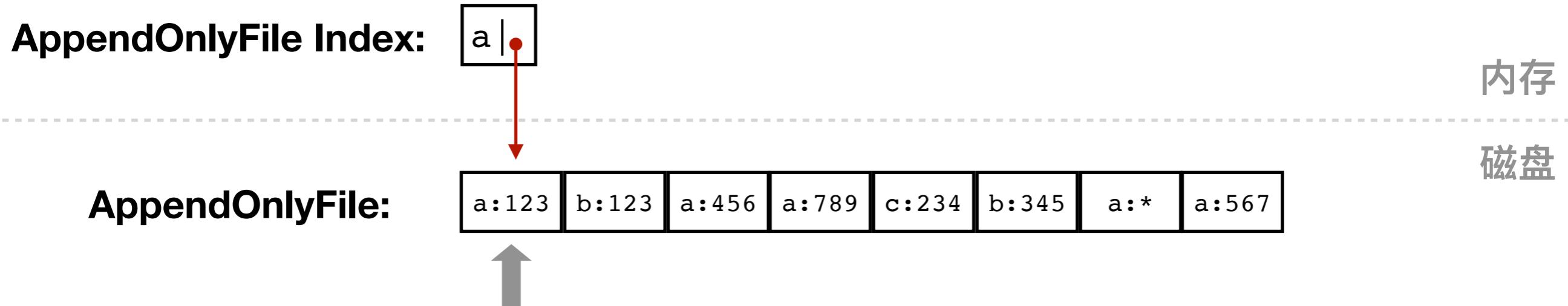
a:123	b:123	a:456	a:789	c:234	b:345	a:*	a:567
-------	-------	-------	-------	-------	-------	-----	-------

内存

磁盘

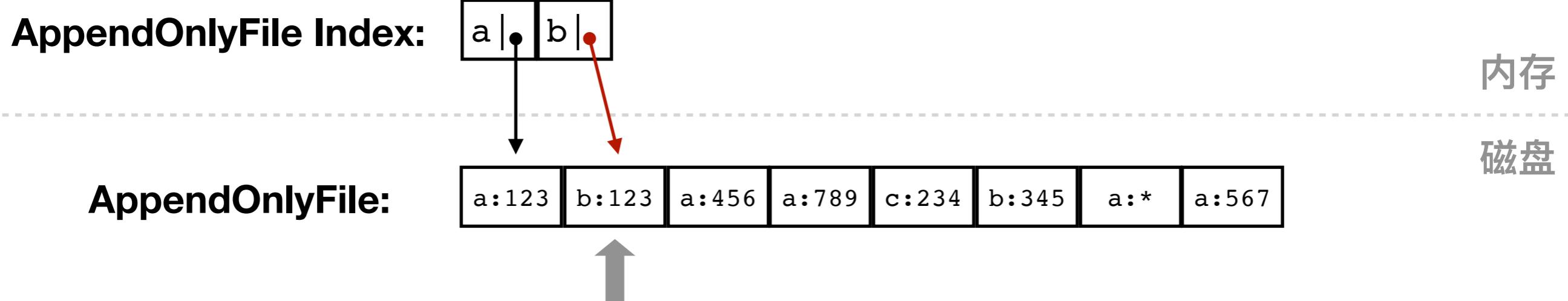
# Append-Only File 内存索引

- 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）

**AppendOnlyFile Index:**



内存

**AppendOnlyFile:**

a:123	b:123	a:456	a:789	c:234	b:345	a:*	a:567
-------	-------	-------	-------	-------	-------	-----	-------

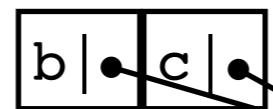
磁盘



# Append-Only File 内存索引

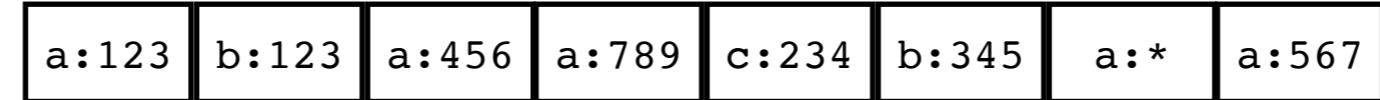
- 建立索引：
  - 遍历 Append-Only File 中的 K-V Record，在索引中保存读取的 Key 及 Record 的位置（Offset）

AppendOnlyFile Index:



内存

AppendOnlyFile:



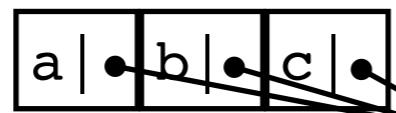
磁盘

↑ 注意删除操作

# 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）

AppendOnlyFile Index:



AppendOnlyFile:

a:123	b:123	a:456	a:789	c:234	b:345	a:*	a:567
-------	-------	-------	-------	-------	-------	-----	-------

内存

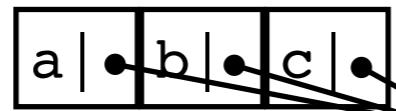
磁盘



# 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 " $O(1)$ "

AppendOnlyFile Index:



AppendOnlyFile:

a:123	b:123	a:456	a:789	c:234	b:345	a:*	a:567
-------	-------	-------	-------	-------	-------	-----	-------

内存

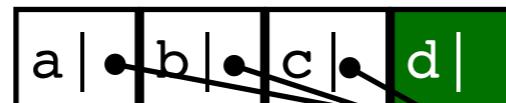
磁盘



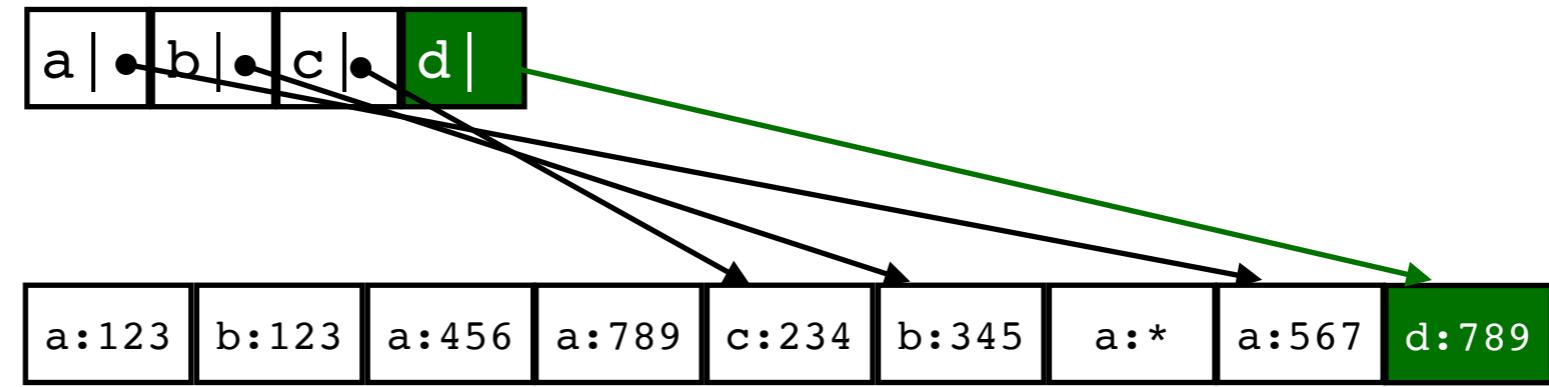
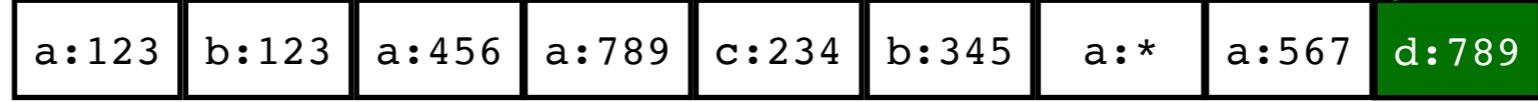
# Append-Only File 内存索引

- 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) ；

**AppendOnlyFile Index:**



**AppendOnlyFile:**



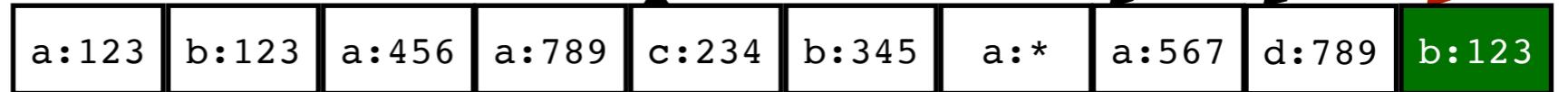
# Append-Only File 内存索引

- 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) ；

**AppendOnlyFile Index:**



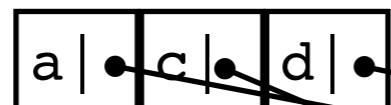
**AppendOnlyFile:**



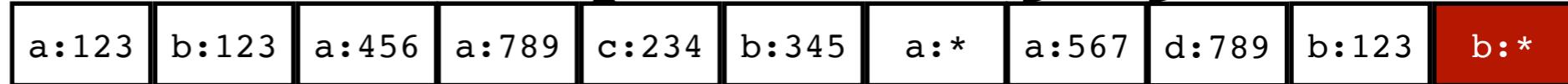
# Append-Only File 内存索引

- 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

**AppendOnlyFile Index:**

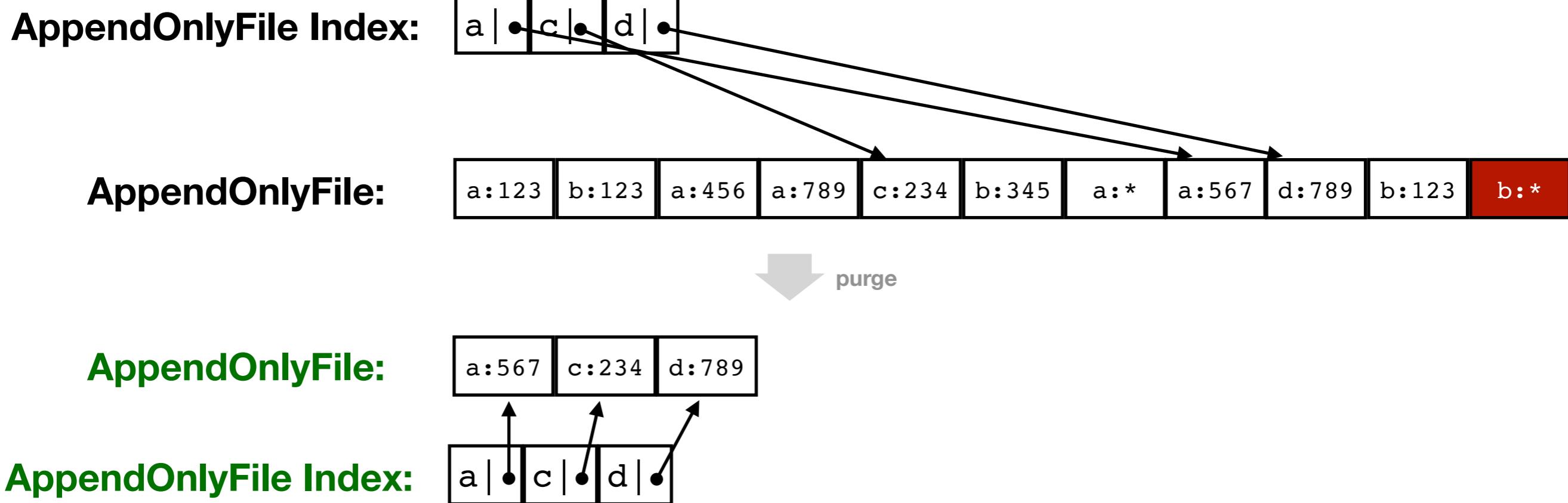


**AppendOnlyFile:**



# 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
    - ...

# 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/>
  - 学习 Redis 使用案例

# 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