作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

缓存的含义

缓存：使用集合结构暂存数据或复用结果，减少重复计算或重复请求，提高性能。

缓存的特点：
容量限制。缓存需要限制容量，超过容量则触发淘汰策略。
过期策略。缓存的数据有过期策略，比如访问或写入后 N 秒过期。
淘汰策略。使用 LFU、LRU 算法，清理缓存的数据，防止数据过久、容量超限。
单值模式或 KV 模式。单值模式只存储 value，KV 模式存储 key-value。
缓存命中率。衡量缓存的重要指标，命中缓存的次数与总的查询次数的比率。
缓存一致性问题。原因在于多副本，数据源与缓存如何保持数据一致。

使用数组缓存加快数字计算

编写代码： num_without_cache.c
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>
#include <sys/time.h>

// 计算次数
uint64_t calc_count = 0;

// 计算数字
uint64_t num_count(int num)
{
    // 边界
    if (num <= 0)
    {
        return 1;
    }
    // 计数
    ++calc_count;
    // 结果
```

```
    return num_count(num - 1) + num_count(num - 3) + num_count(num - 5);
}

// 读毫秒
uint64_t now_millis()
{
    struct timeval time;
    gettimeofday(&time, NULL);
    uint64_t millis = time.tv_sec * 1000 + time.tv_usec / 1000;
    return millis;
}

int main()
{
    // 耗时和结果
    uint64_t millis = now_millis();
    uint64_t num_result = num_count(50);
    uint64_t millis_cost = now_millis() - millis;

    printf(" num_result  = %llu \n", num_result);
    printf(" calc_count  = %llu \n", calc_count);
    printf(" millis_cost = %llu \n", millis_cost);
    return 0;
}
```

编写代码： num_with_cache.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>
#include <sys/time.h>

// 计算次数
uint64_t calc_count = 0;
// 缓存命中次数
uint64_t cache_hit = 0;
// 结果缓存
uint64_t result_cache[1000];

// 计算数字
uint64_t num_count(int num)
{
    // 边界
```

```
    if (num <= 0)
    {
        return 1;
    }

    // 读缓存
    uint64_t tmp = result_cache[num];
    if (tmp > 0)
    {
        cache_hit++;
        return tmp;
    }

    // 计数
    ++calc_count;
    // 结果
    uint64_t ret = num_count(num - 1) + num_count(num - 3) + num_count(num - 5);
    // 写缓存
    result_cache[num] = ret;
    return ret;
}

// 读毫秒
uint64_t now_millis()
{
    struct timeval time;
    gettimeofday(&time, NULL);
    uint64_t millis = time.tv_sec * 1000 + time.tv_usec / 1000;
    return millis;
}

int main()
{
    // 耗时和结果
    uint64_t millis = now_millis();
    uint64_t num_result = num_count(50);
    uint64_t millis_cost = now_millis() - millis;

    printf(" num_result  = %llu \n", num_result);
    printf(" calc_count  = %llu \n", calc_count);
    printf(" cache_hit   = %llu \n", cache_hit);
    printf(" millis_cost = %llu \n", millis_cost);
    return 0;
}
```

编译代码：

```
gcc num_without_cache.c -o num_without_cache
gcc num_with_cache.c -o num_with_cache
```

运行代码：

```
[root@local cache]# ./num_without_cache
 num_result  = 11355378639
 calc_count  = 5677689319
 millis_cost = 40346

[root@local cache]# ./num_with_cache
 num_result  = 11355378639
 calc_count  = 50
 cache_hit   = 92
 millis_cost = 0
```

分析结果：
2 份代码，计算过程一样，计算结果都为 11355378639，差别在于是否使用缓存。
不使用缓存的代码，耗时 40346 毫秒，计算 5677689319 次。
使用缓存的代码，耗时 0 毫秒，计算 50 次，缓存命中 92 次。
对比看出，耗时、计算次数都减少很多，缓存极大优化了性能。

数组缓存有如下优点：
数组结构紧凑，内存利用率高，有利于 CPU 高速缓存。
数组使用下标访问，运算指令少，读写效率高。

多级缓存的协作

很多场景，单级缓存功能不够，需要使用多级缓存，相互协作。
假设某个小公司，需要统计访问的用户数量，以 5 分钟为时间维度，使用用户 ID 去重。考虑到数据量、技术成本，用缓存实现。

编写代码： TimeUserId.java

```
package book.cache;

public class TimeUserId implements Comparable<TimeUserId> {
    long minute;// 时间。分钟
    long userId;// 用户 ID。

    // 排序。去重
    @Override
```

```
    public int compareTo(TimeUserId o) {
        // 先用时间排序
        int ret = Long.compare(this.minute, o.minute);
        if (0 != ret) {
            return ret;
        }
        // 再用 userId 排序
        return Long.compare(this.userId, o.userId);
    }
}
```

编写代码： MultiLevelCache.java

```
package book.cache;

import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.concurrent.ConcurrentSkipListSet;

import com.google.common.collect.ArrayListMultimap;
import com.google.common.collect.ListMultimap;

public class MultiLevelCache {

    // 本机缓存。有排序、去重功能。
    Set<TimeUserId> localCache = new ConcurrentSkipListSet<>();

    // 当前的时间段。5 分钟
    public long minutePeriod() {
        long millis = System.currentTimeMillis();
        return millis / 1000 / 60 / 5;
    }

    // redis，set 结构，去重
    public void redisSetAdd(String key, List<Long> userIdList) {
        // 过程略。
    }

    // redis，设置过期时间，分钟
    public void redisExpire(String key, int minute) {
        // 过程略。
    }

    // redis，set 结构，取大小
```

```
public int redisSetSize(String key) {
    // 过程略。
    return 0;
}

// redis，map 结构，写值
public void redisMapPut(String key, long minute, int userCount) {
    // 过程略。
}

// 记录到本机缓存
public void recordUserIdToLocalCache(long userId) {
    long minute = minutePeriod();
    TimeUserId one = new TimeUserId();
    one.minute = minute;
    one.userId = userId;
    localCache.add(one);
}

// 定时器，每隔几秒，把本机缓存刷到 Redis
public void timerFlushLocalCacheToRedis() {
    // 切换本机缓存
    Set<TimeUserId> tmpCache = localCache;
    localCache = new ConcurrentSkipListSet<>();

    // 按时间分组
    ListMultimap<Long, Long> minuteMap = ArrayListMultimap.create();
    for (TimeUserId one : tmpCache) {
        minuteMap.put(one.minute, one.userId);
    }

    // 批量更新 redis
    for (Long minute : minuteMap.keySet()) {
        List<Long> userIdList = minuteMap.get(minute);
        String key = "user_visit_" + minute;// key
        this.redisSetAdd(key, userIdList);// 批量写
        this.redisExpire(key, 15);// 设置过期时间
    }
}

// 定时器，每隔几秒，统计 Redis 记录的 userid 数量
public void timerCountUserIdInRedis() {
    // 只需要统计最近的 2 个时间段
    long minute1 = minutePeriod();// 当前时间段
```

```
        long minute2 = minute1 - 1;// 前一个时间段
        List<Long> minuteList = Arrays.asList(minute1, minute2);

        // 统计每个 key 的大小
        for (Long minute : minuteList) {
            String key = "user_visit_" + minute;// key
            int userCount = this.redisSetSize(key);// 取大小
            if (userCount <= 0) {// key 可能过期了
                continue;
            }
            // 用 map 汇总统计的结果
            this.redisMapPut("user_visit_map", minute, userCount);
        }
    }
}
```

逻辑流程：
首先，把用户 ID 写到本机缓存，时间段为 5 分钟。用 ConcurrentSkipListSet 实现时间段、用户 ID 去重。
使用方法 recordUserIdToLocalCache(long userId)。
然后，本机启动 1 个定时器，每隔几秒，把本机缓存的用户 ID，刷到记录时间段和用户 ID 的 redis 缓存。
使用方法 timerFlushLocalCacheToRedis()。
最后，启动另 1 个定时器，每隔几秒，读取 redis 缓存的用户 ID 的数量，写到只存用户数量的 redis 缓存。
使用方法 timerCountUserIdInRedis()。

使用 3 级缓存：
本机缓存，使用 ConcurrentSkipListSet，存储几秒的用户 ID。
Redis 缓存，使用 set 结构，存储 15 分钟的用户 ID。
Redis 缓存，使用 map 结构，存储各个时间段的用户数量 UV。

多级缓存的优点：
减少网络请求。如果每次记录用户 ID 都直接写 Redis，会产生很多网络请求，进而增加延时。
近实时。定时器每隔几秒触发刷数据、统计数据，时效性满足一般需求，同时降低系统负载。
模块化设计。体现架构思维，提高扩展性，便于对每个模块做针对性扩展、优化。