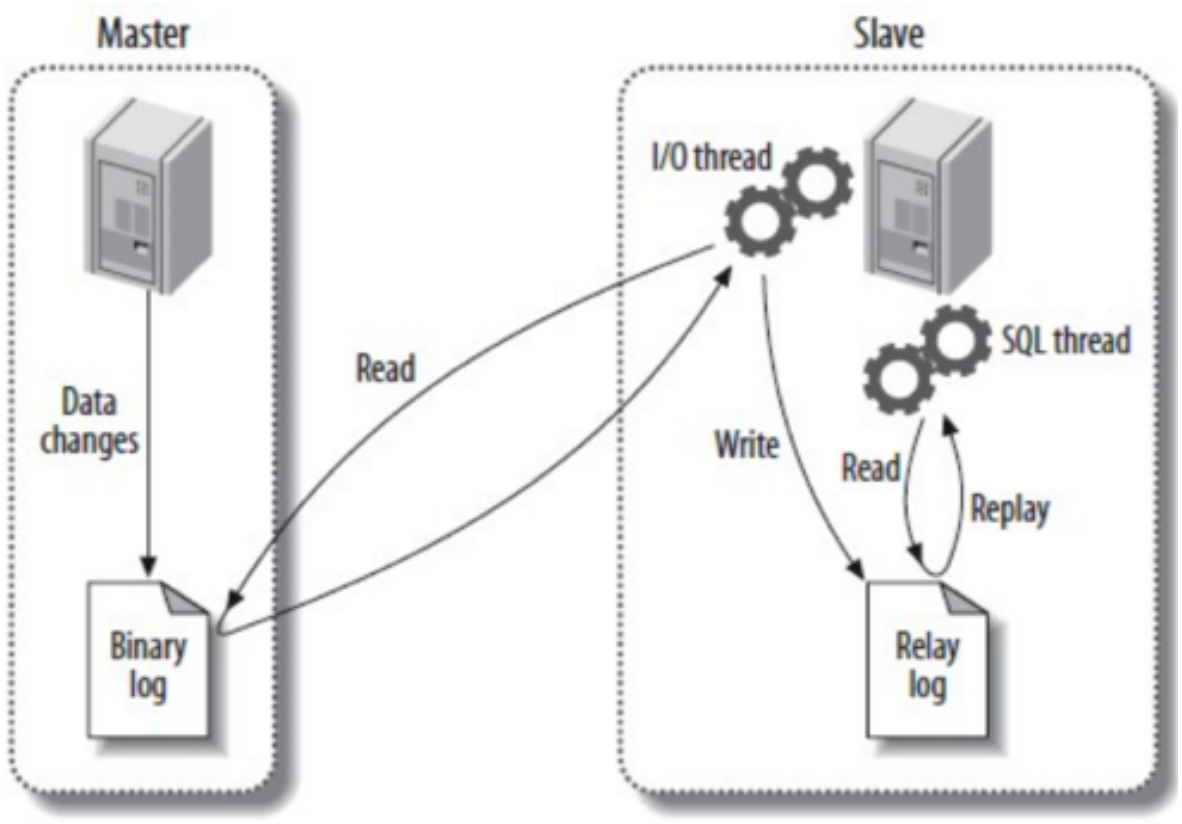
零声教育 Mark 老师 QQ: 2548898954

mysql主从复制

原理图

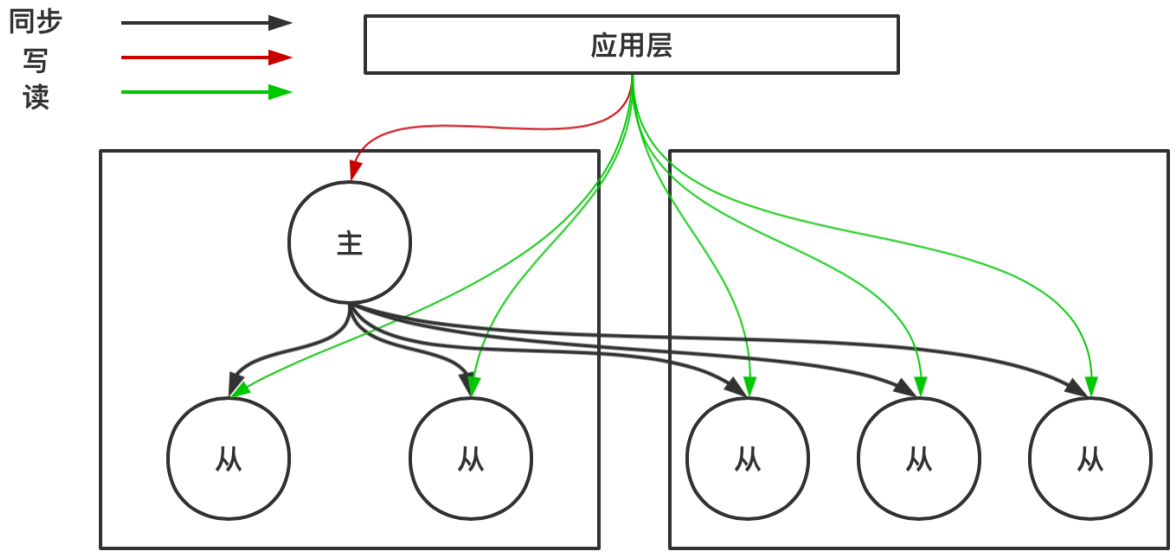


1. 主库更新事件 *DML* 操作(`update`、`insert`、`delete`) 通过 *io-thread* 写到 *binlog*;
2. 从库请求读取 *binlog*，通过 *io-thread* 写入从库本地 *relay-log*（中继日志）；
3. 从库通过 *sql-thread* 读取 *relay-log*，并把更新事件在从库中重放（*replay*）一遍;

复制流程：

1. *Slave* 上面的 IO 线程连接上 *Master*，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容。
2. *Master* 接收到来自 *Slave* 的 *IO* 线程的请求后，负责复制的 *IO* 线程会根据请求信息读取日志指定位置之后的日志信息，返回给 *Slave* 的 *IO* 线程。返回信息中除了日志所包含的信息之外，还包括本次返回的信息已经到 *Master* 端的 *binlog* 文件的名称以及 *binlog* 的位置。
3. *Slave* 的 *IO* 线程接收到信息后，将接收到的日志内容依次添加到 *Slave* 端的 *relay-log* 文件的最末端，并将读取到的 *Master* 端的 *binlog* 的文件名和位置记录到 *master-info* 文件中，以便在下一次读取的时候能够清楚的告诉 *Master* 从何处开始读取日志。
4. *Slave* 的 *Sql* 进程检测到 *relay-log* 中新增加了内容后，会马上解析 *relay-log* 的内容成为在 *Master* 端真实执行时候的那些可执行的内容，并在自身执行。

读写分离（最终一致性）



为什么需要缓冲层?

前提

> 读多写少，单个主节点能支撑项目数据量；数据的主要依据是 *mysql*；

mysql

> *mysql* 有缓冲层，它的作用也是用来缓存热点数据，这些数据包括索引、记录等；*mysql* 缓冲层是从自身出发，跟具体的业务无关；这里的缓冲策略主要是 *lru*；
>
> *mysql* 数据主要存储在磁盘当中，适合大量重要数据的存储；磁盘当中的数据一般是远大于内存当中的数据；一般业务场景关系型数据库（*mysql*）作为主要数据库；

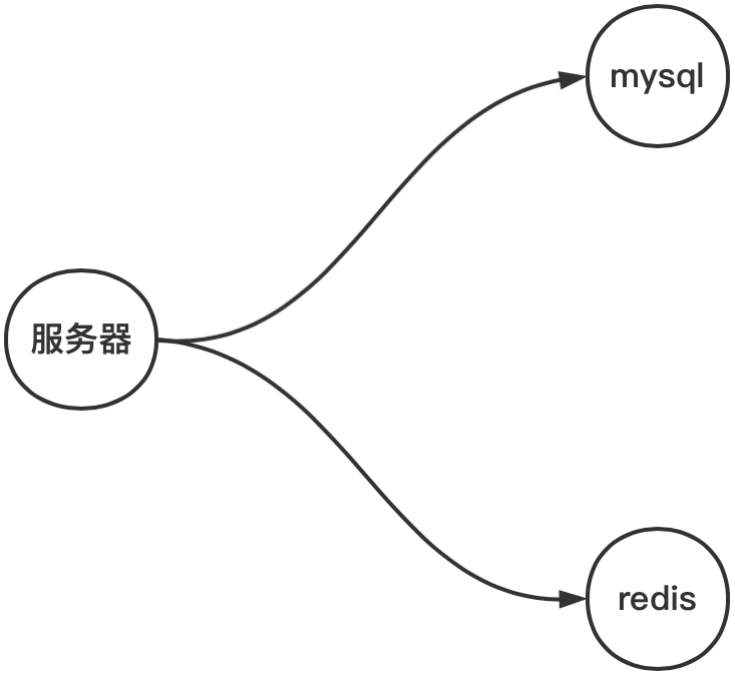
缓冲层

> 缓存数据库可以选用 *redis*，*memcached*；它们所有数据都存储在内存当中，当然也可以将内存当中的数据持久化到磁盘当中；

总结

> 1. 由于 *mysql* 的缓冲层（buffer pool）不由用户来控制，也就是不能由用户来控制缓存具体数据；
> 2. 访问磁盘的速度比较慢，尽量获取数据从内存中获取；
> 3. 主要**解决读的性能**；因为写没必要优化，必须让数据正确的落盘；如果写性能出现问题，那么请使用横向扩展集群方式来解决；
> 4. 项目中需要存储的数据应该远大于内存的容量，同时需要进行数据统计分析，所以数据**存储获取的依据应该是关系型数据库**；
> 5. 缓存数据库可以存储用户自定义的热点数据；以下的讨论都是基于**热点数据的同步问题**；

原理图



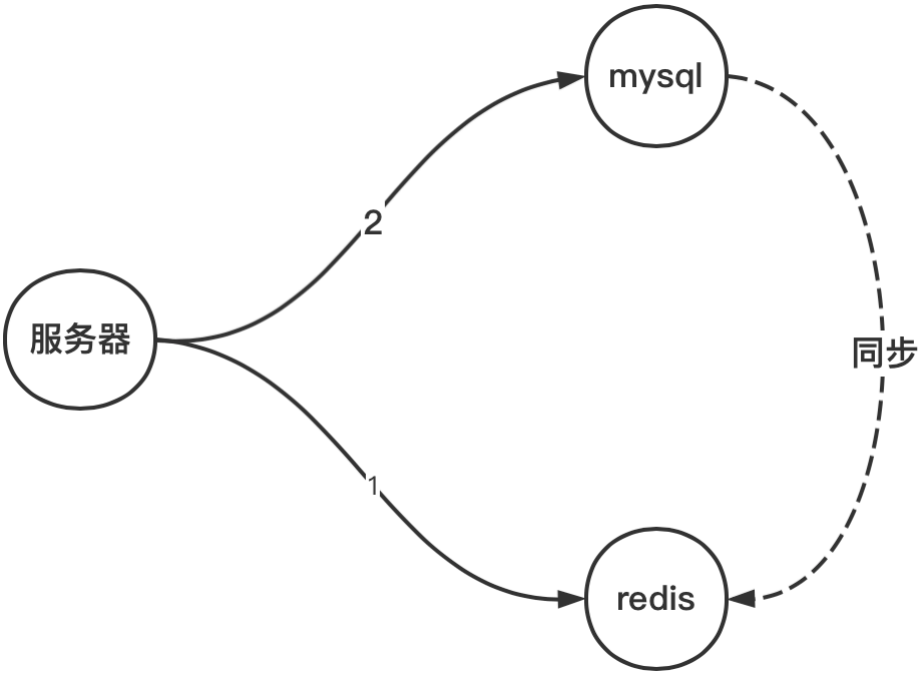
为什么有同步的问题?

> 没有缓冲层之前，我们对数据的读写都是基于 *mysql*；所以不存在同步问题；这句话也不是必然，比如读写分离就存在同步问题（数据一致性问题）；
>
> 引入缓冲层后，我们对数据的获取需要分别操作缓存数据库和 *mysql*；那么这个时候数据可能存在几个状态?
>
> 1. *mysql* 有，缓存无
> 2. *mysql* 无，缓存有
> 3. 都有，但数据不一致
> 4. 都有，数据一致
> 5. 都没有
>
> 4 和 5显然是没问题的，我们现在需要考虑1、2以及3；
>
> 首先明确一点：我们获取数据的主要依据是 *mysql*，只需要将 *mysql* 的数据正确同步到缓存数据库就可以了；同理，缓存有，*mysql* 没有，这比较危险，此时我们可以认为该数据为脏数据；所以我们需要在同步策略中避免该情况发生；同时可能存在 *mysql* 和缓存都有数据，但是数据不一致，这种也需要在同步策略中避免；
>
> **注意**:
>
> 缓存不可用，整个系统依然要保持正常工作；
>
> *mysql* 不可用的话，系统停摆，停止对外提供服务；

解决数据同步问题

策略1

> 读流程：先读缓存，若缓存有，直接返回；若缓存没有，读 *mysql*；若 *mysql* 有，同步到缓存，并返回；若 *mysql* 没有，则返回没有；
>
> 写流程：先删除缓存，再写 *mysql*，后面数据同步交由 *go-mysql-transfer* 等中间件处理；（将问题 3 转化成 1）
>
> 先删除缓存，为了避免其他服务读取旧的数据；也是告知系统这个数据已经不是最新，建议从 *mysql* 获取数据；
>
> 但是对于服务 A 而言，写入 *mysql* 后，接着读操作必须要能读到最新的数据；

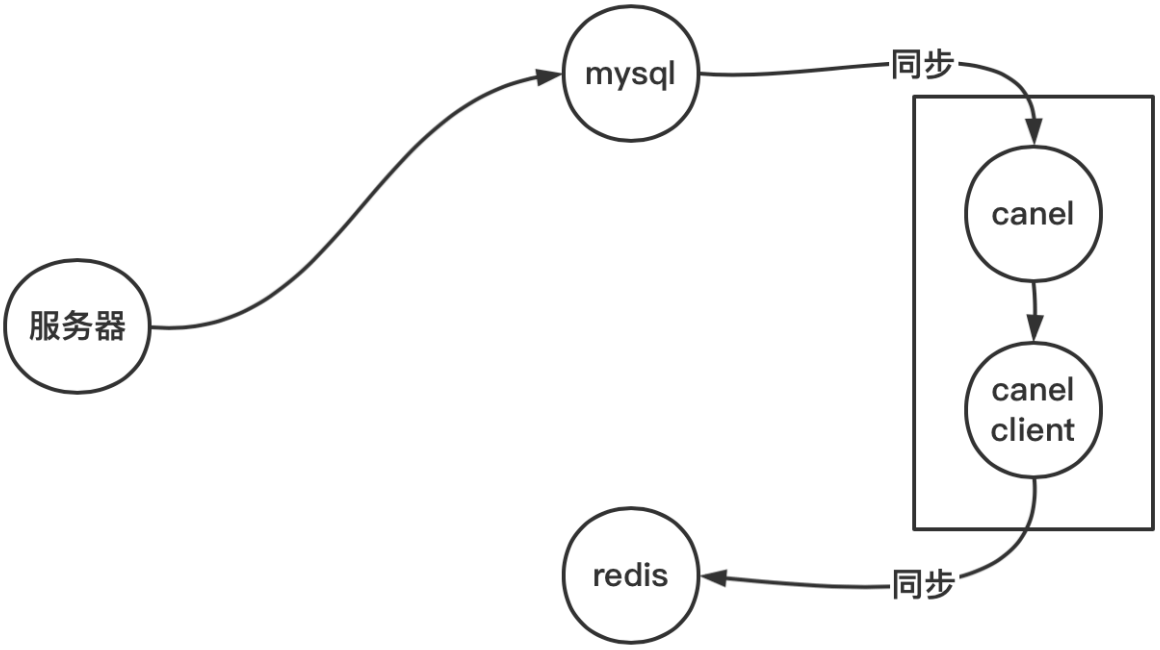


策略2

> 读流程：先读缓存，若缓存有，直接返回；若缓存没有，读 *mysql*；若 *mysql* 有，同步到缓存，并返回；若 *mysql* 没有，则返回没有；
>
> 写流程：先写缓存，并设置过期时间（如 **200ms**），再写 *mysql*，后面数据同步交由其他中间件处理；
>
> 这里设置的过期时间是预估时间，大致上是 *mysql* 到缓存同步的时间；
>
> 在写的过程中如果 *mysql* 停止服务，或数据没写入 *mysql*，则 **200 ms** 内提供了脏数据服务；但仅仅只有 **200ms** 的数据错乱；

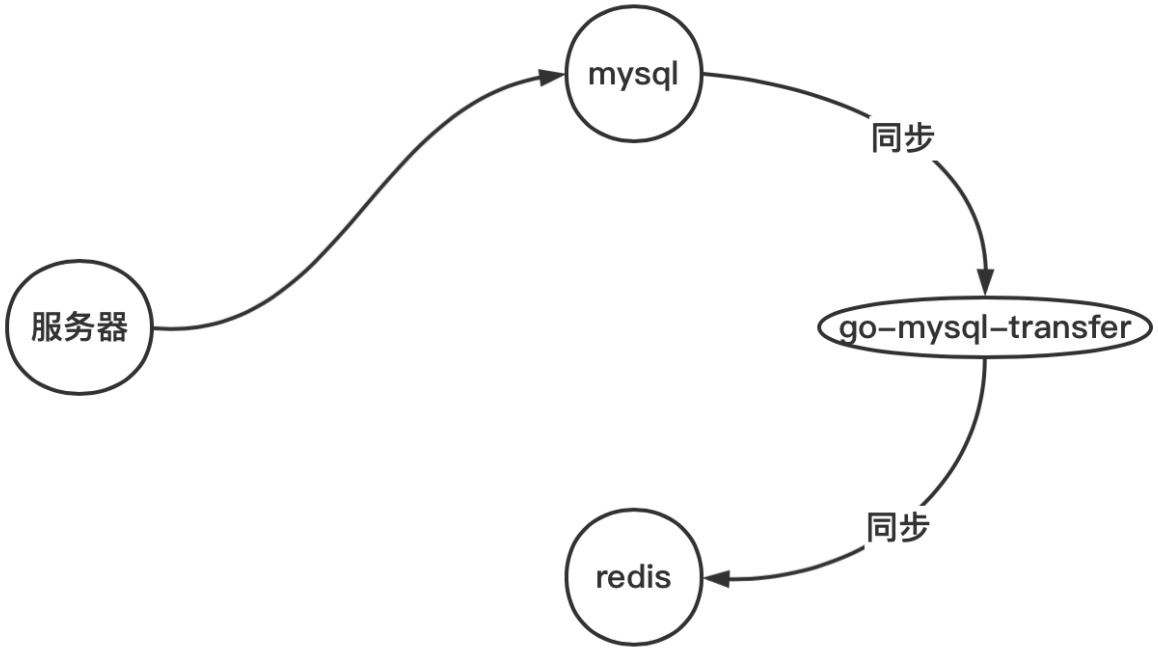
同步方案1

原理图



同步方案2

原理图



代码1

`git clone https://gitee.com/mirrors/go-mysql-transfer.git`

```
# 查询 master 状态，获取 日志名和偏移量
mysql> show master status;
# 重置同步位置 （假设通过上面命令获取到日志名和偏移量为 mysql-bin.000025 993779648）
./go-mysql-transfer -config app.yml -position mysql-bin.000025 993779648
# 全量数据同步  第一次
./go-mysql-transfer -stock
```

```
/*
mysql 配置文件 my.cnf
```

```
log-bin=mysql-bin # 开启 binlog
binlog-format=ROW # 选择 ROW 模式
server_id=1 # 配置 MySQL replaction 需要定义，不要和 go-mysql-transfer 的 slave_id 重复
*/
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
    `id` BIGINT,
    `nick` VARCHAR (100),
    `height` INT8,
    `sex` VARCHAR (1),
    `age` INT8,
    PRIMARY KEY (`id`)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert into `user` values (10001, 'mark', 180, '1', 30);
update `user` set `age` = 31 where id = 10001;
delete from `user` where id = 10001;
```

```
-- go-mysql-transfer
--[[
安装步骤:
GO111MODULE=on
git clone https://gitee.com/0k/go-mysql-transfer.git
go env -w GOPROXY=https://goproxy.cn,direct
go build

修改 app.yml
执行 go-mysql-transfer
]]

local ops = require("redisOps") --加载redis操作模块

local row = ops.rawRow()  --当前数据库的一行数据,table类型，key为列名称
local action = ops.rawAction()  --当前数据库事件,包括：insert、updare、delete

if action == "insert" then -- 只监听insert事件
    local id = row["id"] --获取ID列的值
    local name = row["name"] --获取USER_NAME列的值
    local key = name .. ":" .. id
    local sex = row["sex"]
    local height = row["height"] --获取PASSWORD列的值
    local age = row["age"]
    local createtime = row["createtime"] --获取CREATE_TIME列的值
    ops.HSET(key, "id", id) -- 对应Redis的HSET命令
    ops.HSET(key, "name", name) -- 对应Redis的HSET命令
    ops.HSET(key, "sex", sex) -- 对应Redis的HSET命令
    ops.HSET(key, "height", height) -- 对应Redis的HSET命令
    ops.HSET(key, "age", age) -- 对应Redis的HSET命令
end
```

代码2

```
git clone  https://gitee.com/josinli/mysql_redis.git
```

问题是否解决?

> 没有，我们刚刚思考的方向全是正常流程下的方式，我们来看异常情况；

缓存穿透

> 假设某个数据 *redis* 不存在，*mysql* 也不存在，而且一直尝试读怎么办？缓存穿透，数据最终压力依然堆积在 *mysql*，可能造成 *mysql* 不堪重负而崩溃;

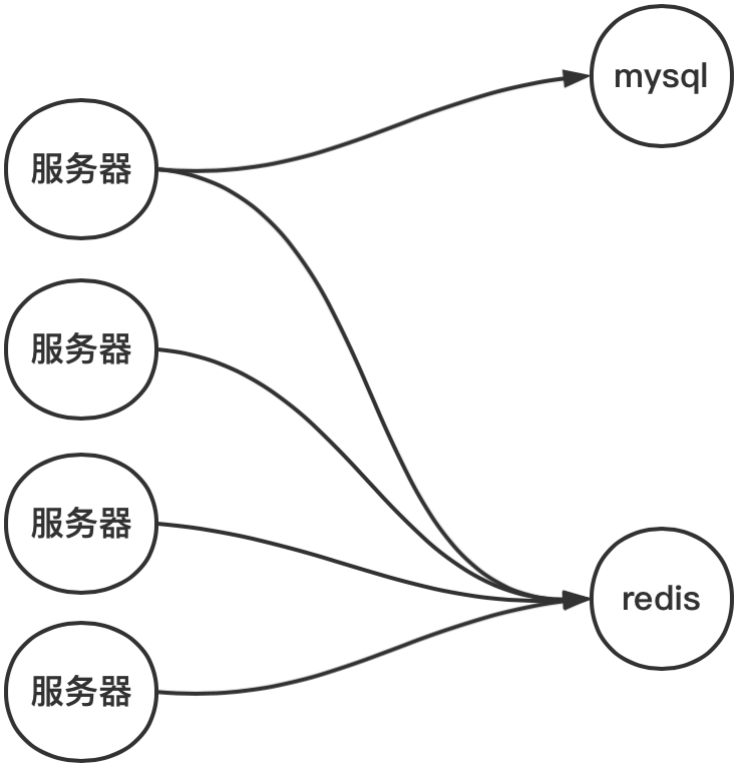
解决

> 1. 发现 *mysql* 不存在，将 *redis* 设置为 `<key, nil>` 设置过期时间 下次访问 *key* 的时候 不再访问 *mysql* 容易造成 *redis* 缓存很多无效数据；
> 2. 布隆过滤器，将 *mysql* 当中已经存在的 *key*，写入布隆过滤器，不存在的直接 *pass* 掉；

缓存击穿

> 缓存击穿某些数据 *redis* 没有，但是 *mysql* 有；此时当大量这类数据的**并发**请求，同样造成 *mysql* 过大；

原理图



解决

> 1. 分布式锁

> 请求数据的时候获取锁，若获取成功，则操作后释放锁；若获取失败，则休眠一段时间（200ms）再去获取，当获取成功，操作后释放锁

> 2. 将很热的 *key*，设置不过期；

缓存雪崩

> 表示一段时间内，缓存集中失效（*redis* 无， *mysql* 有），导致请求全部走 *mysql*，有可能搞垮数据库，使整个服务失效；
>
> *mysql* 主要的数据的依据；*redis* 可有可无的状态；

解决

> 缓存数据库在整个系统不是必须的，也就是缓存宕机不会影响整个系统提供服务；
>
> 1. 如果因为缓存数据库宕机，造成所有数据涌向 mysql；
>
>   采用高可用的集群方案，如哨兵模式、cluster 模式；
> 2. 如果因为设置了相同的过期时间，造成缓存集中失效；
>
>   设置随机过期值或者其他机制错开失效时间；
> 3. 如果因为系统重启的时候，造成缓存数据消失；
>
>   重启时间短，*redis* 开启持久化（过期信息也会持久化）就行了； 重启时间长提前将热数据导入 *redis* 当中；