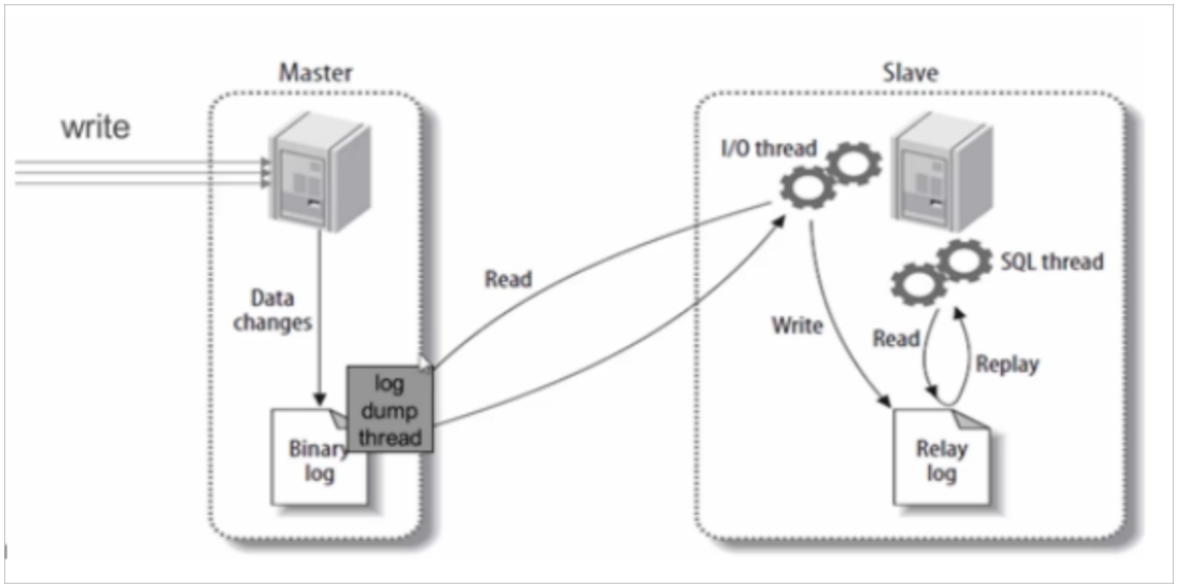
零声教育 Mark 老师 QQ: 2548898954

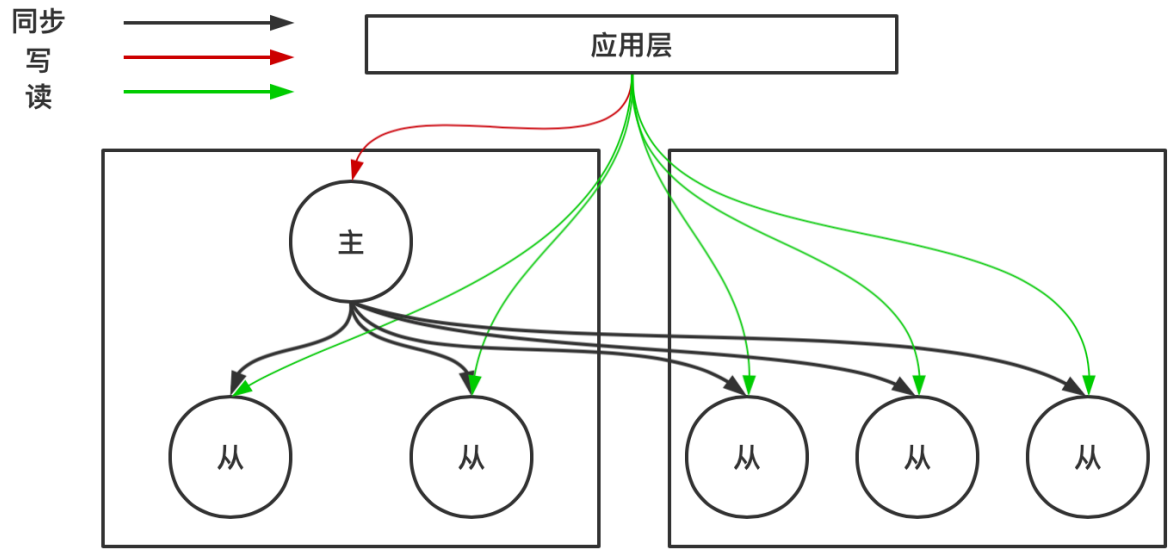
mysql主从复制

原理图



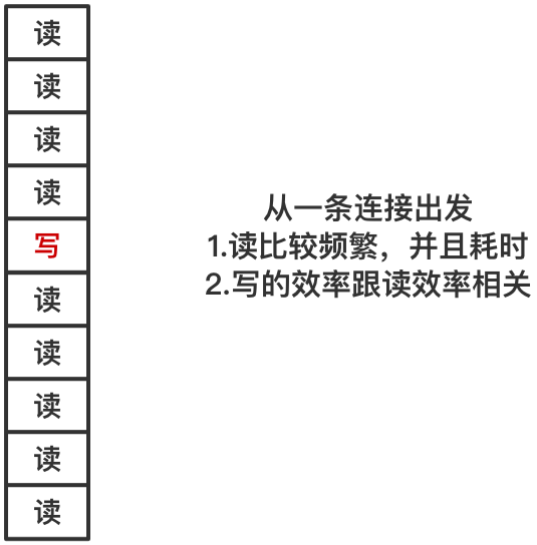
> 此图来源网络
>
> 1. 主库更新事件(update、insert、delete)通过 io-thread 写到binlog；
> 2. 从库请求读取 binlog，通过 io-thread 写入（write）从库本地 relay log（中继日志）；
> 3. 从库通过sql-thread读取（read） relay log，并把更新事件在从库中执行（replay）一遍；
>
> 复制流程：
>
> 1. Slave上面的IO进程连接上Master，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容。
> 2. Master接收到来自Slave的IO进程的请求后，负责复制的IO进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进程。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bin-log文件的名称以及bin-log的位置。
> 3. Slave的IO进程接收到信息后，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 bin-log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

> 写主读从



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；

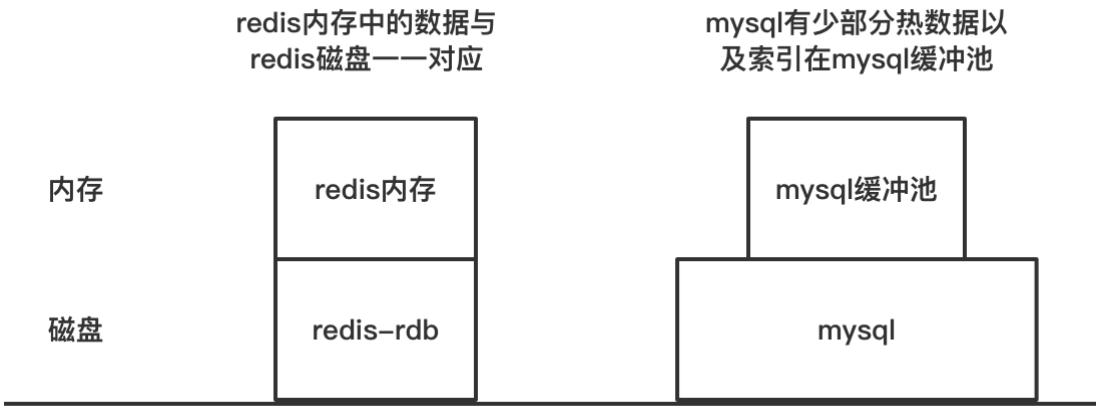
mysql

> mysql 有缓冲层，它的作用也是用来缓存热点数据，这些数据包括数据文件、索引文件等；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，当然是经过优化的 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；内存的数据和磁盘的数据是一比一的；

存储比较



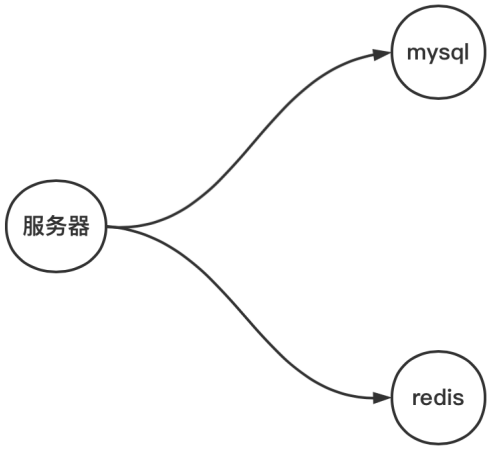
几项重要的数据

> 1. 内存的访问速度是磁盘访问速度的10万倍（数量级倍率）；内存的访问速度大约是100ns，而一次磁盘访问大约是10ms；访问mysql时访问磁盘的次数跟b+树的高度相关；
> 2. 一般大部分项目中，数据库读操作是写操作的10倍左右；

总结

> 1. 由于mysql的缓冲层不由用户来控制，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，所以mysql数据正确就万事大吉，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；
>
> 缓存不可用的话，我们整个系统应该要保持正常工作；
>
> mysql不可用的话，应该停止对外提供服务；
>
> 另外可以将问题 3 转化为问题 1；

尝试解决同步（一致性）

写

> 主要数据存储在mysql当中，所以先写mysql，如果mysql不可用，直接返回；mysql写成功后，再将数据同步给redis就行了，如果此时redis不可用，应该怎么做?

读

> 先从redis当中获取数据，如果redis不可用，直接去mysql获取；如果redis有，直接返回；如果redis没有，转而向mysql请求，如果mysql没有，直接返回；如果MySQL有，则返回并将数据回写到redis当中；

总结

> 1. 业务层引入了两个变化方向，尽量减少redis的流程；
> 2. 业务层控制热数据流程；回写流程控制热数据流程；
> 3. 热数据不是总是热数据；需要将热数据设置超时时间；

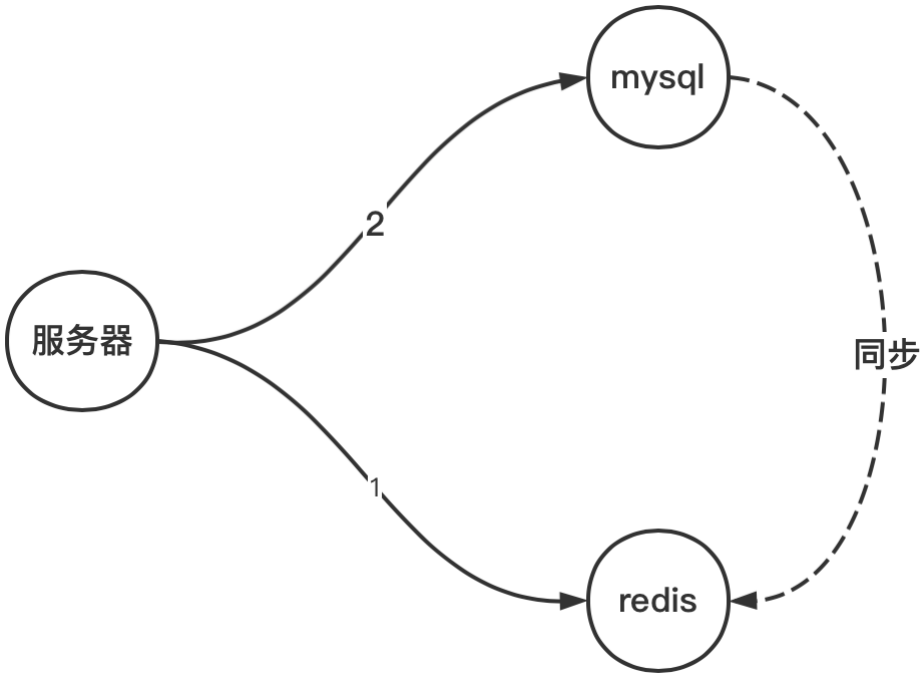
解决数据同步问题

一致性问题

强一致性

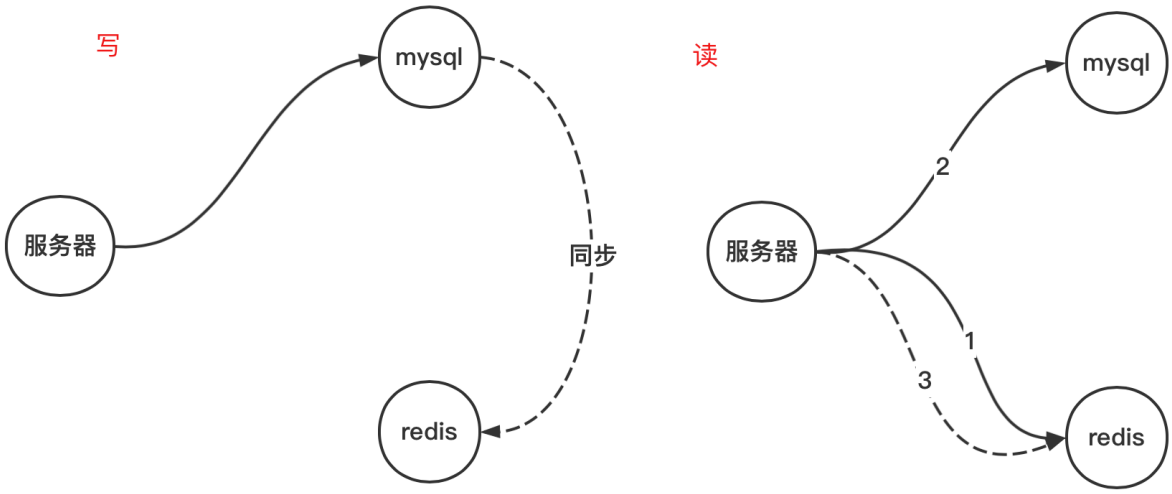
> 同步是否成功的依据来源于mysql是否同步到redis，即使没有同步成功，也没关系；
>
> 写流程：先删除缓存，再写mysql，后面数据的同步交由go-mysql-transfer；
>
> 先删除缓存，为了避免其他服务读取旧的数据；也是告知系统这个数据已经不是最新，建议从mysql获取数据；
>
> 那么在删除缓存和写mysql之间发生读取mysql行为，是否是不一致行为?

> 如下图，可以理解**连续两个**sql操作，发起请求前后顺序并不能解读为数据的一致性；
>
> 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；



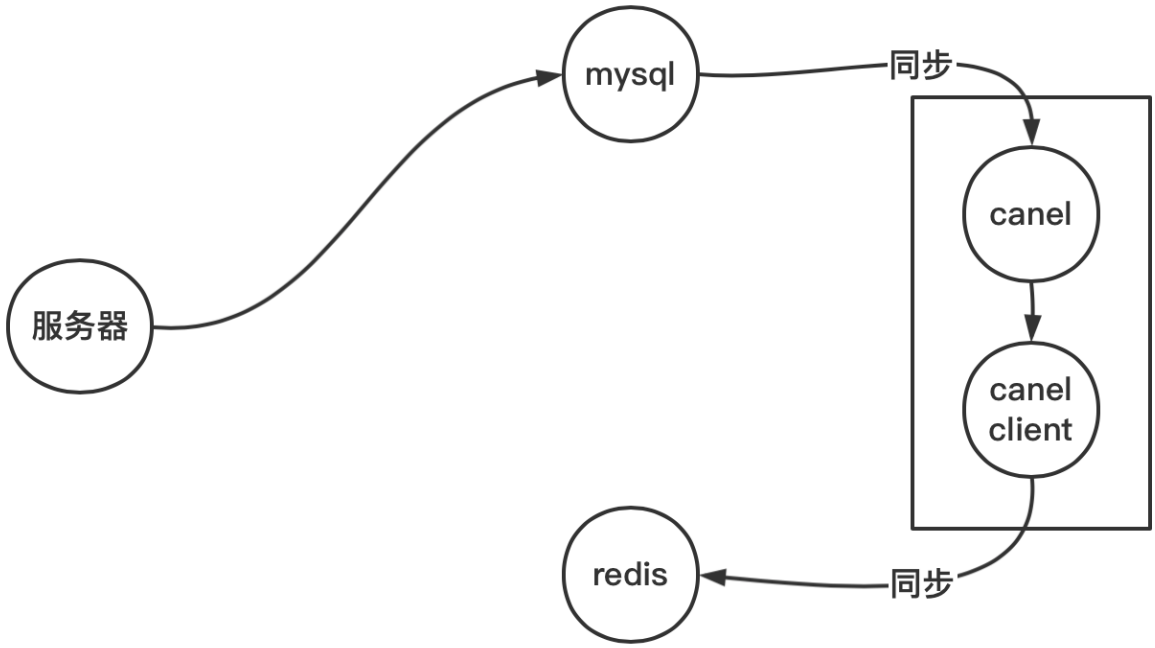
最终一致性

> 读写分离，主库将数据同步到从库，是需要时间，那么在同步期间，主从之间数据有差异；
>
> 这里有写两种方案：
>
> 第一种：直接写mysql，等待mysql同步数据到redis；
>
> 第二种：先写redis，设置key的过期时间为200ms（经验值），等待mysql回写redis，覆盖key，设置更长的过期时间；
>
> 200ms 默认的是 写mysql到mysql同步到redis的时长；这个需要根据实际环境进行设置；



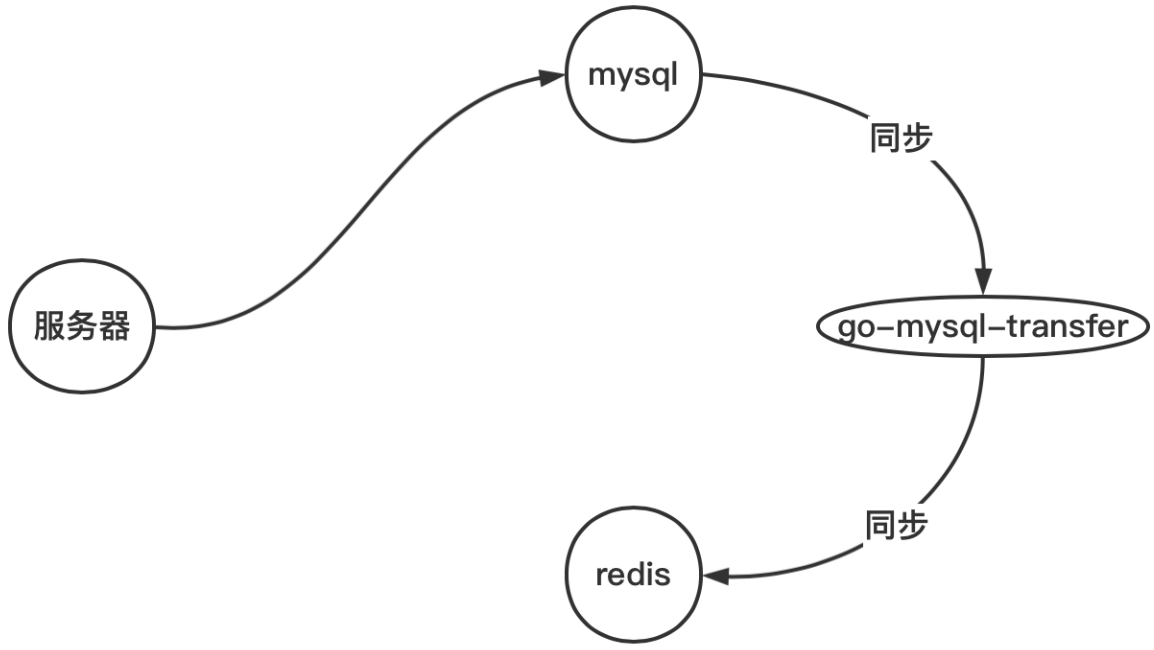
数据同步方案1

原理图



数据同步方案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
./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
*/
CREATE TABLE `user` (
    `id` BIGINT,
    `name` VARCHAR (100),
    `height` INT8,
    `sex` VARCHAR (1),
    `age` INT8
    PRIMARY KEY (`id`)
);

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

```
-- go-mysql-transfer
--[[
安装步骤:
GO111MODULE=on
git clone https://gitee.com/Ok/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
```

> 触发器：具备事务 外键具备事务 如果
>
> 不建议使用，有事务的场景容易出错；虽然保证了真正的强一致性；
>
> 这个实现每次插入修改都需要重新建立redis连接，操作完后又释放redis连接；

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;

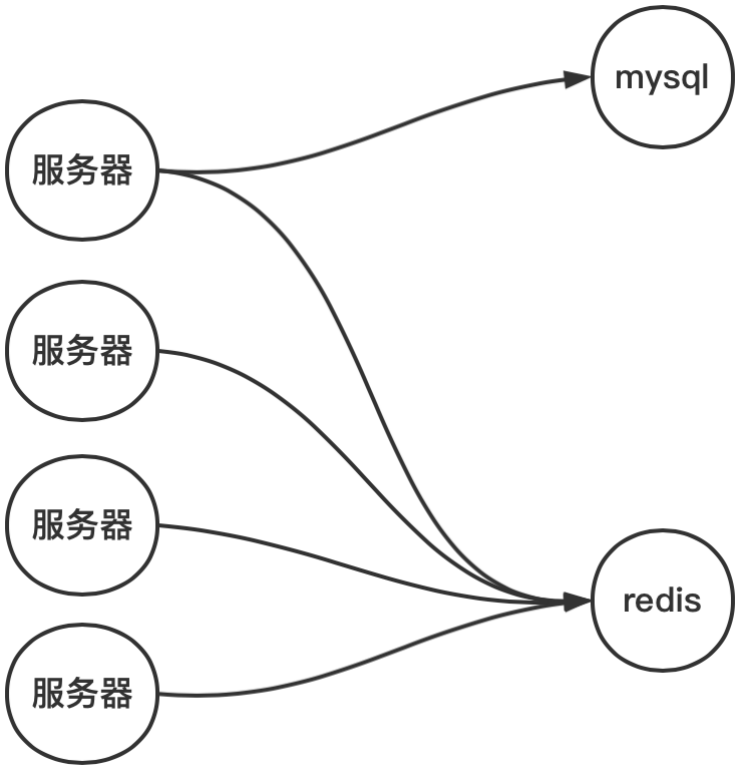
解决

> 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. 加锁
>
> 请求数据的时候获取锁，如果获取成功，则操作，获取失败，则休眠一段时间（200ms）再去获取；获取成功，则释放锁
>
> 首先读redis，不存在，读mysql，存在，写redis　key的锁
>
> 整个流程走完，才让后面的服务器访问

> 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；

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；