Go 进阶训练营
第6课
案例 - 评论系统架构设计答疑

大明

目录

数据库设计——层级结构

场景：需要存储层级数据

- 组织关系
- 评论系统
- 聊天 thread （slack）

基本概念：

- 根节点（root）：没有父亲的节点
- 叶子节点（leaf node）：没有子节点的节点
- 中间结点：同时具有父亲节点和子节点

数据库设计——邻接表Adjacency List

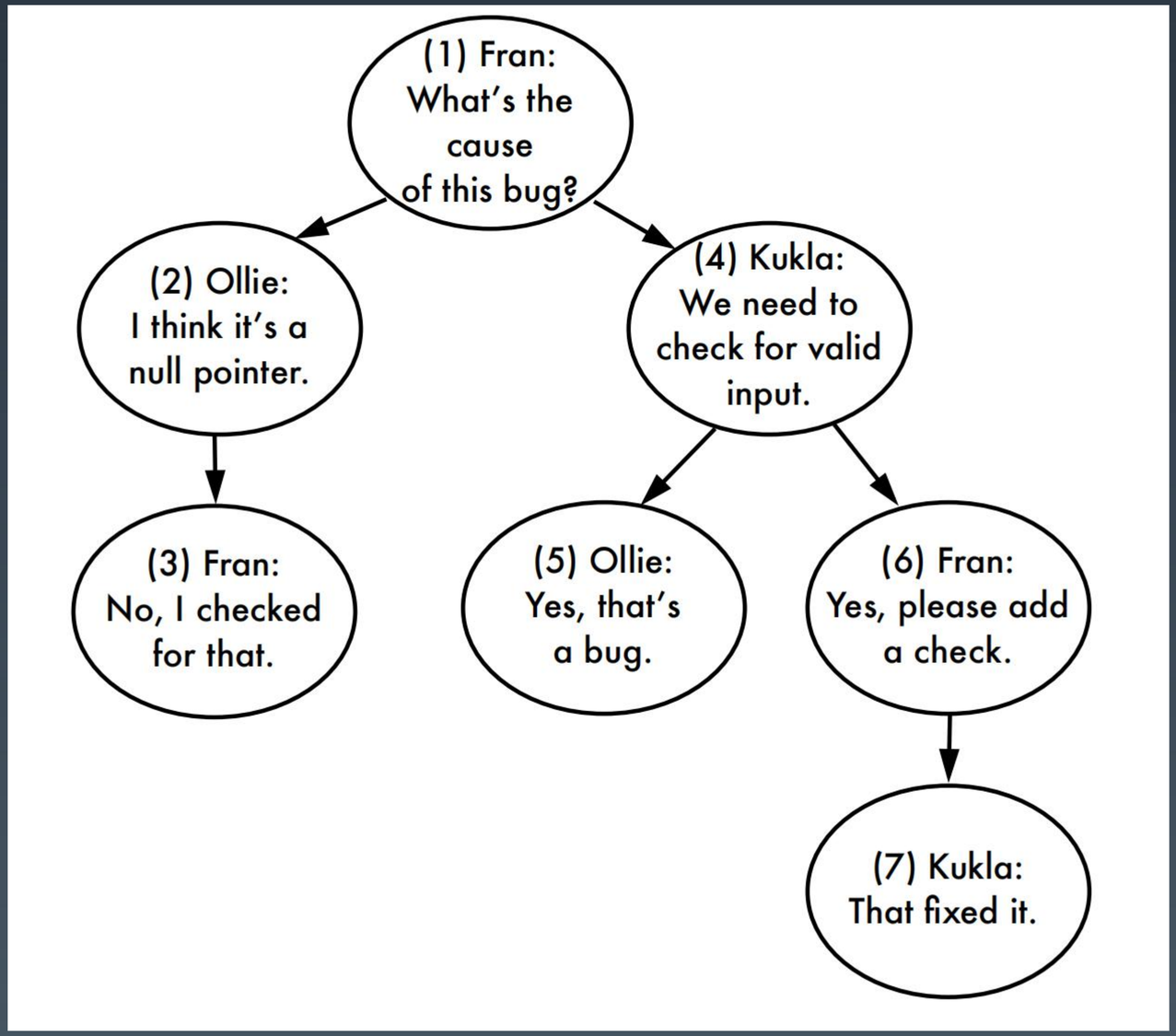
在数据库中设计一个 parent_id 字段

parent_id 为 NULL 表示是根节点

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

comment_id	parent_id	author	comment
1	NULL	Fran	What's the cause of this bug?
2	1	Ollie	I think it's a null pointer.
3	2	Fran	No, I checked for that.
4	1	Kukla	We need to check for invalid input.
5	4	Ollie	Yes, that's a bug.
6	4	Fran	Yes, please add a check.
7	6	Kukla	That fixed it.

数据库设计——邻接表


(1) Fran:
What's the
cause
of this bug?
(2) Ollie:
I think it's a
null pointer.
(4) Kukla:
We need to
check for valid
input.
(3) Fran:
No, I checked
for that.
(5) Ollie:
Yes, that's
a bug.
(6) Fran:
Yes, please add
a check.
(7) Kukla:
That fixed it.

数据库设计——邻接表

缺点：

- 难以查询全部子节点
- 难以查询特定层的节点
- 难以查询聚合函数

Users > ming_ > Downloads > ancestors.sql

```sql
SELECT c1.*, c2.*, c3.*, c4.*
FROM Comments c1                                -- 1st level
  LEFT OUTER JOIN Comments c2
    ON c2.parent_id = c1.comment_id             -- 2nd level
  LEFT OUTER JOIN Comments c3
    ON c3.parent_id = c2.comment_id             -- 3rd level
  LEFT OUTER JOIN Comments c4
    ON c4.parent_id = c3.comment_id; -- 4th level

```

使用 LEFT OUTER JOIN 来查询节点。
查询复杂，而且性能不好

数据库设计——邻接表

缺点：

- 难以查询全部子节点
- 难以查询特定层的节点
- 难以查询聚合函数

```
10
11  SELECT * FROM Comments WHERE bug_id = 1234;
```

一次性查出全部节点，在应用内完成树形结构构建

数据库设计——邻接表

缺点：

- 难以查询全部子节点
- 难以查询特定层的节点
- 难以查询聚合函数

```
12
13  SELECT comment_id FROM Comments WHERE parent_id = 4; -- returns 5 and 6
14  SELECT comment_id FROM Comments WHERE parent_id = 5; -- returns none
15  SELECT comment_id FROM Comments WHERE parent_id = 6; -- returns 7
16  SELECT comment_id FROM Comments WHERE parent_id = 7; -- returns none
17  DELETE FROM Comments WHERE comment_id IN ( 7 );
18  DELETE FROM Comments WHERE comment_id IN ( 5, 6 );
19  DELETE FROM Comments WHERE comment_id = 4;
```

删除节点复杂

数据库设计——使用分段式 Path

Path 可以理解为分段的。一般来说倾向于设计为 /a/b/c的形式。当然也可以是 a_b_c 或者 a#b#c。核心在于自己应当定义一个分隔符。

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  path         VARCHAR(1000),
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

数据库设计——使用分段式 Path

缺点：

- 查找主要依赖于 LIKE 查询
- 无法保证 PATH 正确
- Path 依赖于应用的字符串处理

- 根节点：/1, /1/, 1/, 1
- Like “1/%”, “1/2/%”

comment_id	path	author	comment
1	1/	Fran	What’s the cause of this bug?
2	1/2/	Ollie	I think it’s a null pointer.
3	1/2/3/	Fran	No, I checked for that.
4	1/4/	Kukla	We need to check for invalid input.
5	1/4/5/	Ollie	Yes, that’s a bug.
6	1/4/6/	Fran	Yes, please add a check.
7	1/4/6/7/	Kukla	That fixed it.

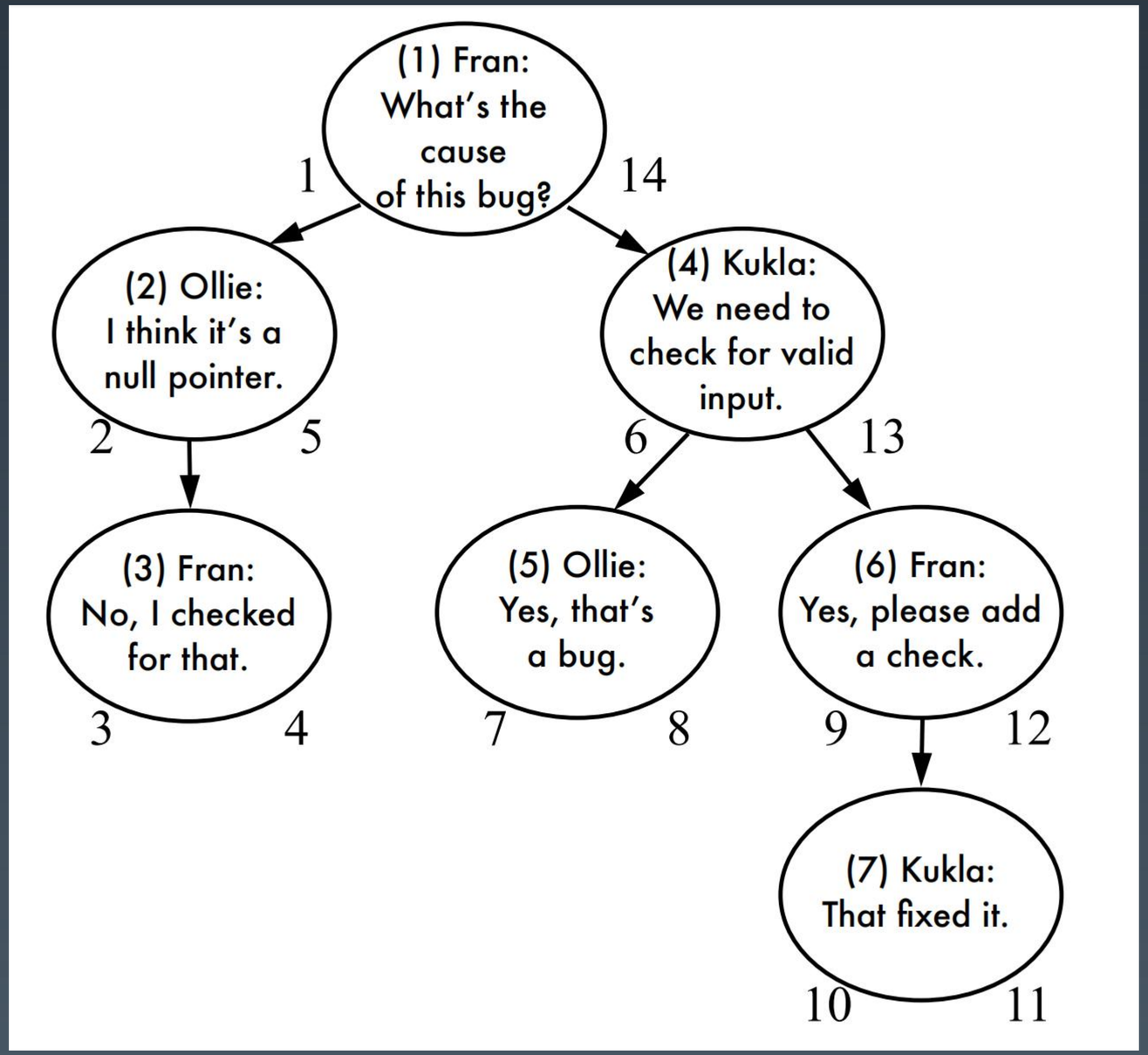
数据库设计——Nested Set

- nsleft 小于所有子节点的 nsleft
- nsright 大于所有子节点的 nsright
- 本质上是一个深度优先遍历的顺序
- 经过删除等操作之后，数字不再连续

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  nsleft       INTEGER NOT NULL,
  nsright      INTEGER NOT NULL,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs (bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

数据库设计——Nested Set

- nsleft 小于所有子节点的 nsleft
- nlright 大于所有子节点的 nright
- 本质上是一个深度优先遍历的顺序
- 经过删除等操作之后，数字不再连续



数据库设计——Nested Set

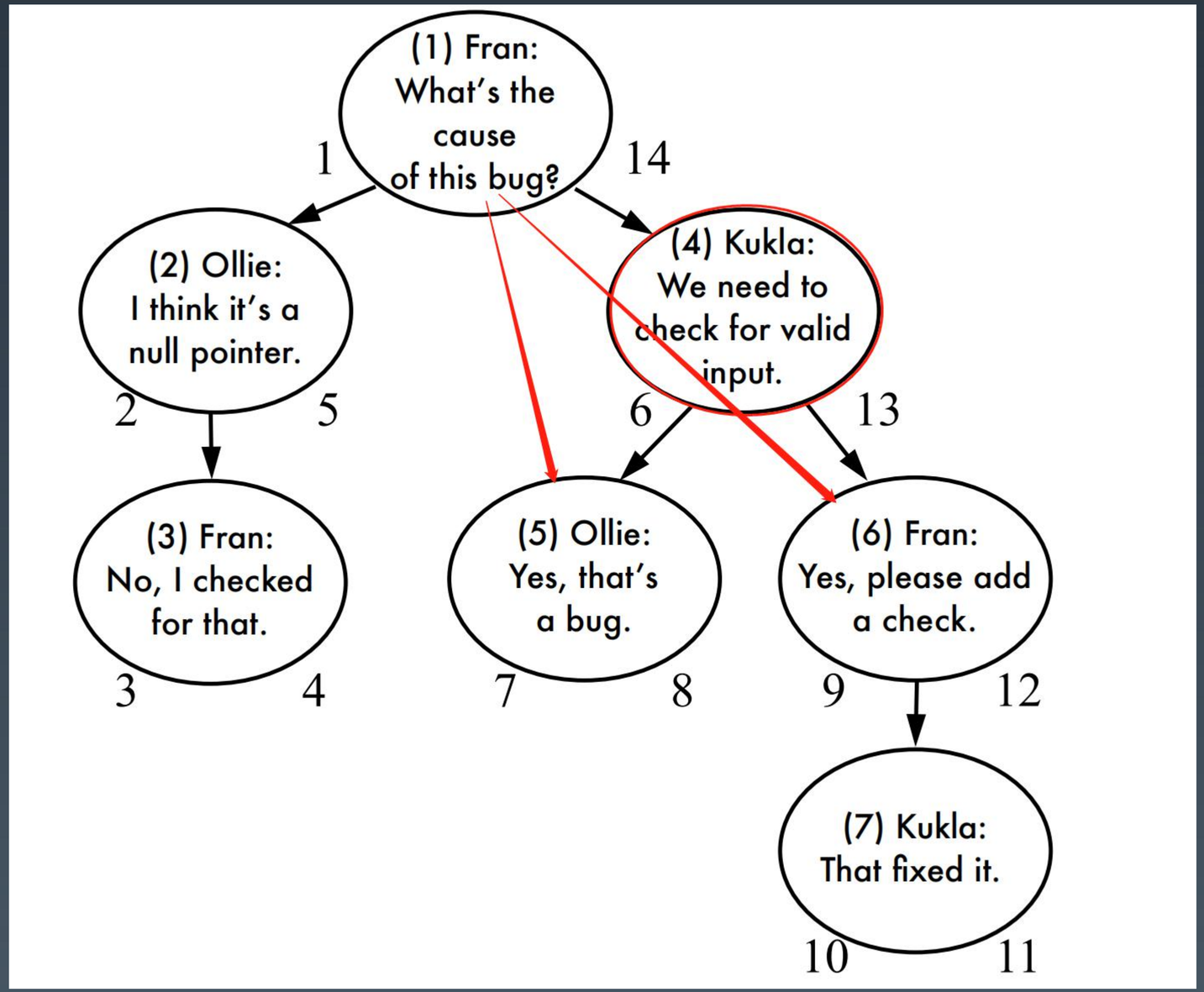
- nsleft 小于所有子节点的 nsleft
- nlright 大于所有子节点的 nright
- 本质上是一个深度优先遍历的顺序
- 经过删除等操作之后，数字不再连续

```
21
22  SELECT c2.* FROM Comments AS c1
23  JOIN Comments as c2
24  ON c2.nsleft BETWEEN c1.nsleft AND c1.nsright
25  WHERE c1.comment_id = 4;
```

查询 4 的子节点

数据库设计——Nested Set

- nsleft 小于所有子节点的 nsleft
- nlright 大于所有子节点的 nright
- 本质上是一个深度优先遍历的顺序
- 经过删除等操作之后，数字不再连续



删除某个中间结点之后，中间结点的子节点变成了其父亲的直属子节点

数据库设计——Nested Set

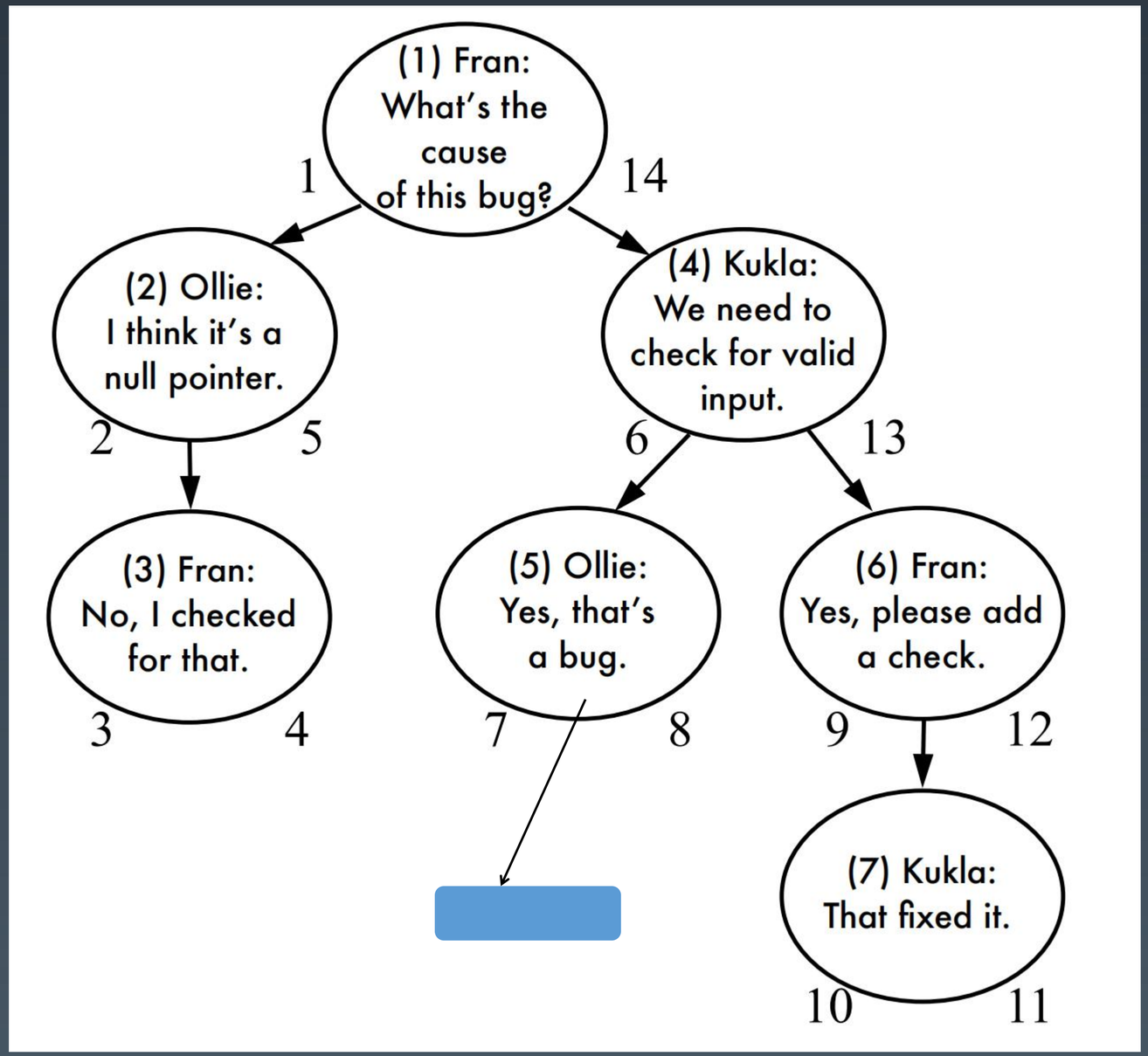
缺点：

- 查找父亲非常困难：c 结点的父亲节点是祖先节点里面，没有别的祖先节点作为其子节点的节点
- 插入非常困难：需要重新计算节点的 nsleft 和 nsright

```
26
27  SELECT parent.* FROM Comment AS c
28  JOIN Comment AS parent
29  ON c.nsleft BETWEEN parent.nsleft AND parent.nsright
30  LEFT OUTER JOIN Comment AS in_between
31  ON c.nsleft BETWEEN in_between.nsleft AND in_between.nsright
32  AND in_between.nsleft BETWEEN parent.nsleft AND parent.nsright
33  WHERE c.comment_id = 6
34  AND in_between.comment_id IS NULL;
```

数据库设计——Nested Set

- nsleft 小于所有子节点的 nsleft
- nlright 大于所有子节点的 nright
- 本质上是一个深度优先遍历的顺序
- 经过删除等操作之后，数字不再连续



数据库设计——Closure Table

用单独的表来存储节点之间的关系

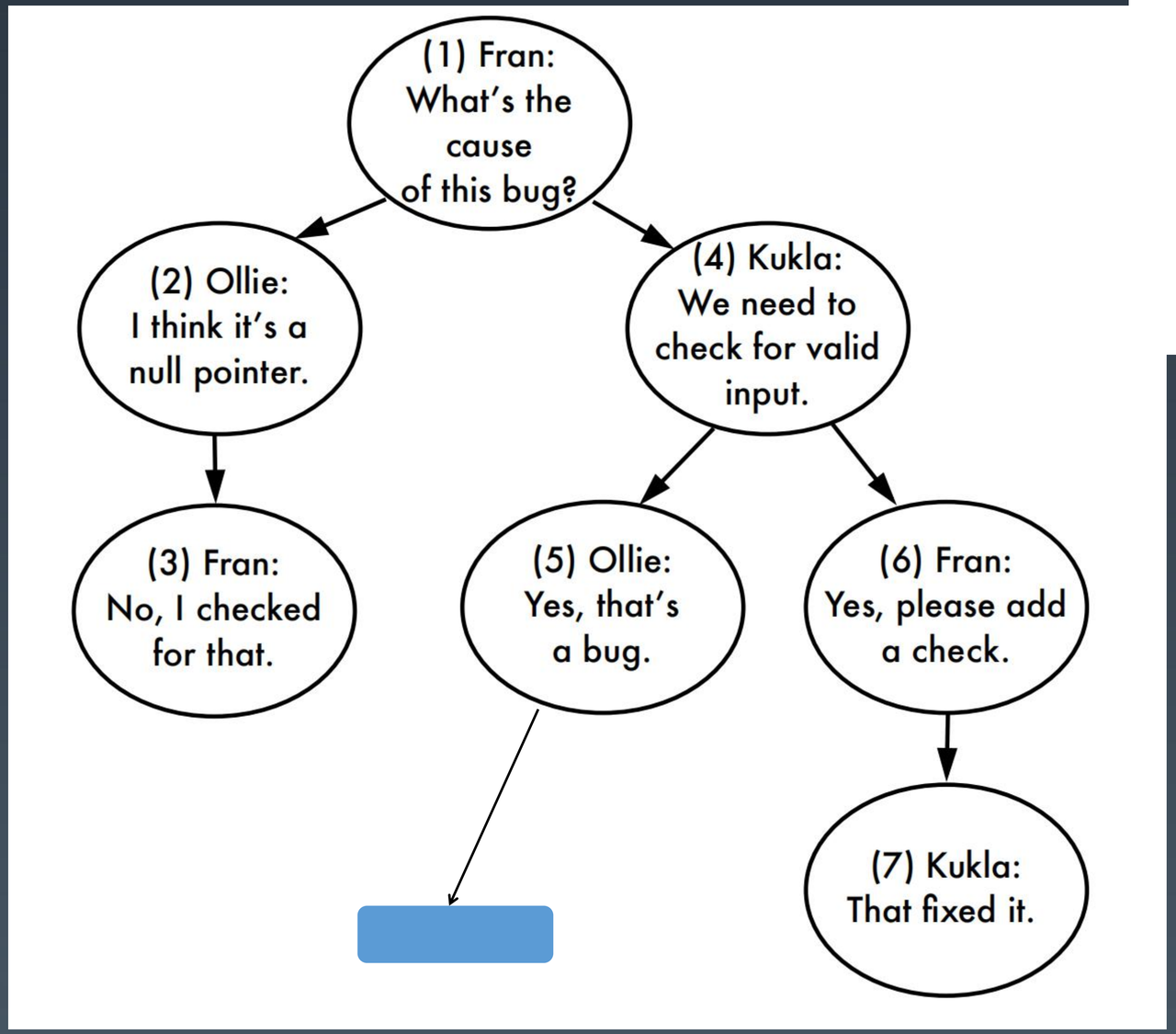
根节点的父亲节点指向自己

叶子节点的子节点指向自己

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE TreePaths (
  ancestor    BIGINT UNSIGNED NOT NULL,
  descendant  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY(ancestor, descendant),
  FOREIGN KEY (ancestor) REFERENCES Comments(comment_id),
  FOREIGN KEY (descendant) REFERENCES Comments(comment_id)
);
```

数据库设计——Closure Table



ancestor	descendant	ancestor	descendant	ancestor	descendant
1	1	1	7	4	6
1	2	2	2	4	7
1	3	2	3	5	5
1	4	3	3	6	6
1	5	4	4	6	7
1	6	4	5	7	7

```
68   1 5
69   4 5
70   5 5
71
72   1 8
73   4 8
74   5 8
75   8 8
```

数据库设计——树形结构对比

Design	Tables	Query Child	Query Tree	Insert	Delete	Ref. Integ.
Adjacency List	1	Easy	Hard	Easy	Easy	Yes
Recursive Query	1	Easy	Easy	Easy	Easy	Yes
Path Enumeration	1	Easy	Easy	Easy	Easy	No
Nested Sets	1	Hard	Easy	Hard	Hard	No
Closure Table	2	Easy	Easy	Easy	Easy	Yes

Resursive Query 直接依赖于数据库特性。

系统设计——数据与数据统计信息分离

comment_index_[0-199]

id	int64	主键
obj_id	int64	对象id
obj_type	int8	对象类型
member_id	int64	发表者用户id
root	int64	根评论ID，不为0是回复评论
parent	int64	父评论ID，为0是root评论
floor	int32	评论楼层
count	int32	评论总数
root_count	int32	根评论总数
like	int32	点赞数
hate	int32	点踩数
state	int8	状态（0-正常、1-隐藏）
attrs	int32	属性
create_time	timestamp	创建时间
update_time	timestamp	修改时间

comment_content_[0-199]

comment_id	int64	主键
at_member_ids	string	对象id
ip	int64	对象类型
platform	int8	发表者用户id
device	string	根评论ID，不为0是回复评论
message	string	评论内容
meta	string	评论元数据: 背景、字体
create_time	int32	创建时间
update_time	int32	修改时间

root 和 parent 是用于维系树形结构

floor, count, root_count, like, hate 属于统计信息

典型场景：统计赞踩，阅读，转发……

系统设计——朴素解决方案

一张主表打天下。

	id	评论 ID
2	content	评论内容
3	hate	踩
4	like	赞
5		

优点：表结构简单，代码简单

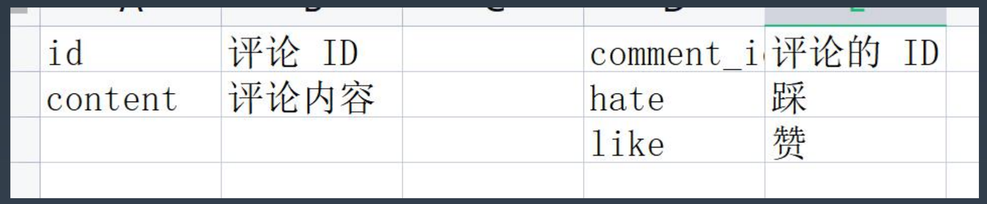
缺点：不适合大系统

系统设计——分离表结构

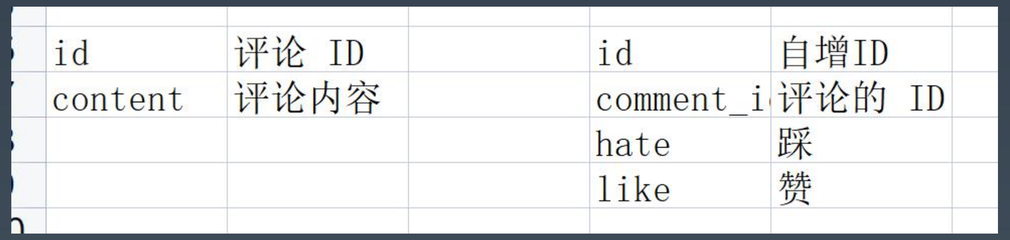
表的数据可以分成两部分：

1. 频繁变更的统计信息
2. 不频繁变更的基本信息

所以可以尝试分离开来



id	评论 ID		comment_i	评论的 ID
content	评论内容		hate	踩
			like	赞



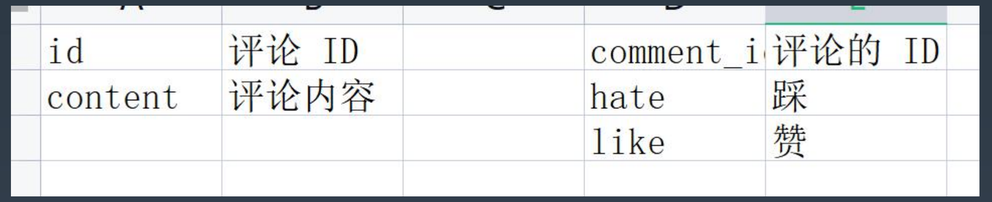
id	评论 ID		id	自增ID
content	评论内容		comment_i	评论的 ID
			hate	踩
			like	赞

系统设计——分离表结构

我要不要在统计表里面使用自增主键？

核心是保证插入是朝着单向插入的，

以保证页分裂是朝着单向增长的

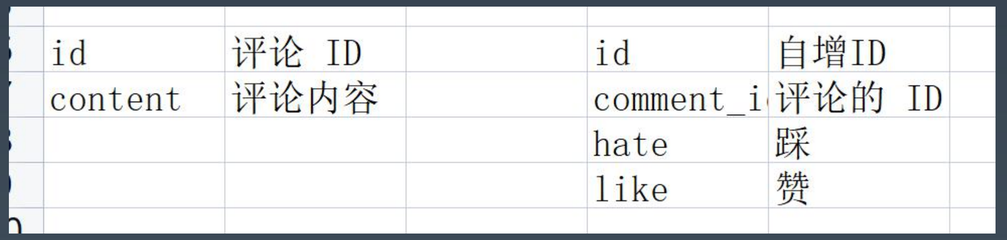


id	评论 ID		comment_i	评论的 ID
content	评论内容		hate	踩
			like	赞

所以：

1. 如果在插入评论的时候也初始化好统计信息，随便用哪个都可以（使用 comment_id 作为主键可以少维护一个唯一索引）

2. 如果在插入评论的时候不会初始化统计信息，那么最好使用自增主键



id	评论 ID		id	自增ID
content	评论内容		comment_i	评论的 ID
			hate	踩
			like	赞

系统设计——多业务统计信息

我不仅仅有评论，还有文章，视频.....
都需要统计信息：

2	id	评论 ID		id	自增ID
3	content	评论内容		biz_id	业务的ID
4				biz_type	业务的类型
5				hate	踩
6	id	视频ID		like	赞
7	address	视频的存储地址		read	观看数
8					
9					
20	id	文章 ID			
21	content	内容			
22	category_	类目			

系统设计——更新统计信息

- 侵入式直接更新。直接在各自的代码里面调用数据库层操作，更新统计数据；

```
38
39  func GetArticle() Article {
40      // ... 一通操作猛如虎拿到了 article
41      increaseReadCnt()
42  }
43
44  func GetVideo() Video {
45      // ... 一通操作猛如虎拿到了 Video 的信息
46      increaseReadCnt()
47  }
```

2	id	评论 ID		id	自增ID
3	content	评论内容		biz_id	业务的ID
4				biz_type	业务的类型
5				hate	踩
6	id	视频ID		like	赞
7	address	视频的存储地址		read	观看数
8					
9					
10	id	文章 ID			
11	content	内容			
12	category_	类目			

- 业务侵入严重；
- 如果业务API在不同的项目上，那么复用代码困难
- 在读里面嵌入了写操作，极大影响性能

系统设计——异步更新统计信息

- 侵入式直接更新。开启 goroutine

```
37
38
39  func GetArticle() Article {
40      // ... 一通操作猛如虎拿到了 article
41      go increaseReadCnt()
42  }
43
44  func GetVideo() Video {
45      // ... 一通操作猛如虎拿到了 Video 的信息
46      go increaseReadCnt()
47  }
```

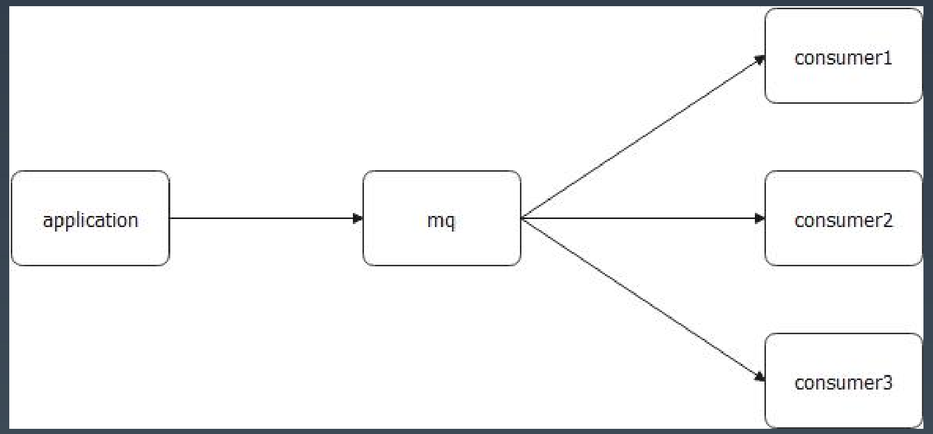
2	id	评论 ID		id	自增ID
3	content	评论内容		biz_id	业务的ID
4				biz_type	业务的类型
5				hate	踩
6	id	视频ID		like	赞
7	address	视频的存储地址		read	观看数
8					
9					
20	id	文章 ID			
21	content	内容			
22	category_	类目			

- 业务侵入严重；
- 如果业务API在不同的项目上，那么复用代码困难
- 开 goroutine 能缓解性能问题，但是要提防高并发导致 goroutine 极多
- 容错难。无法使用事务保证数据一致性

系统设计——异步更新统计信息

- 利用 MQ 解耦
- go + MQ

```
48
49  func GetArticle() Article {
50      // ... 一通操作猛如虎拿到了 article
51      dropMsg()
52  }
53
54  func GetVideo() Video {
55      // ... 一通操作猛如虎拿到了 Video 的信息
56      dropMsg()
57  }
```

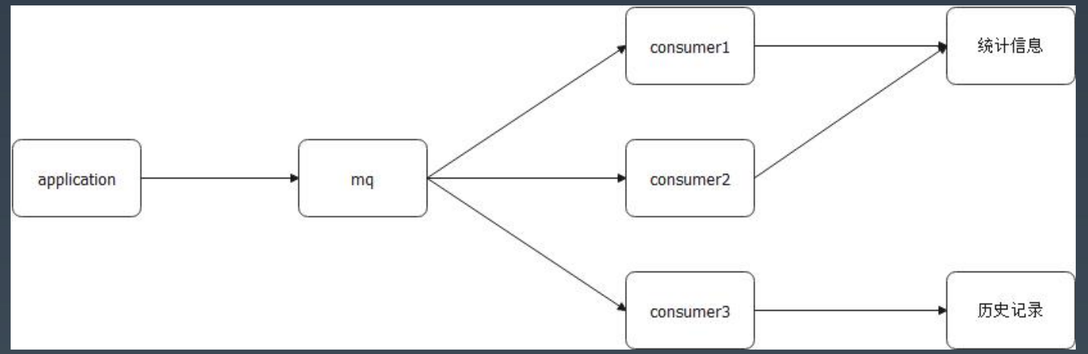


- 引入了额外的中间件，可用性下降
- 数据一致性更加难以保证

系统设计——异步更新统计信息

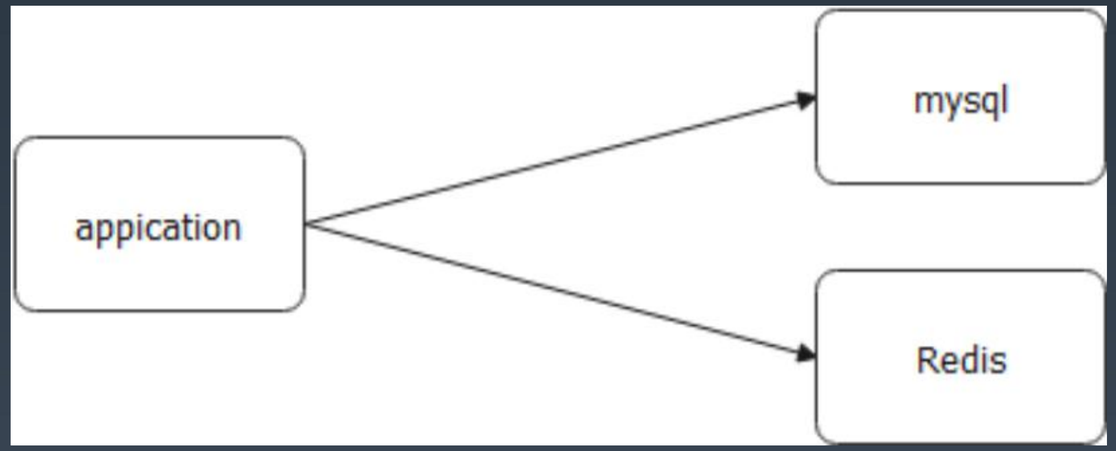
- 利用 MQ 解耦
- go + MQ
- MQ 消息可以用于其它功能，例如历史记录，点赞记录

```
48
49  func GetArticle() Article {
50      // ... 一通操作猛如虎拿到了 article
51      dropMsg()
52  }
53
54  func GetVideo() Video {
55      // ... 一通操作猛如虎拿到了 Video 的信息
56      dropMsg()
57  }
```



系统设计——采用 KV 来存储统计数据

- 数据主体存储在关系型数据库，但是统计放在别的地方。
- 首选是 KV 结构存储：统计信息往往不会做关系型运算（查询），只是根据业务主键来查找



Redis 开启持久化。

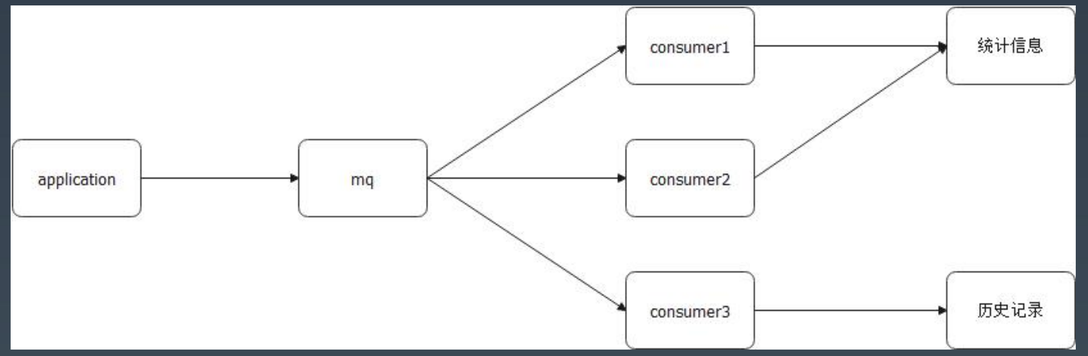
优点：性能好，落地简单

缺点：持久化本身会影响 redis 的性能，可能影响别的业务

系统设计——数据一致性保证

同一个数据库：

1. 严格一致性：开启本地事务
2. 宽松一致性：不需要事务



存储在不同地方（不同数据库或者引入了其它中间件）：

1. 严格一致性：分布式事务 or 本地事务 + 补偿（基本就类似于下单的那种流程）
2. 宽松一致性：允许部分数据丢失 + 离线数据对比

系统设计——缓存模式 cache pattern

- cache aside
- read through
- write through
- refresh-ahead

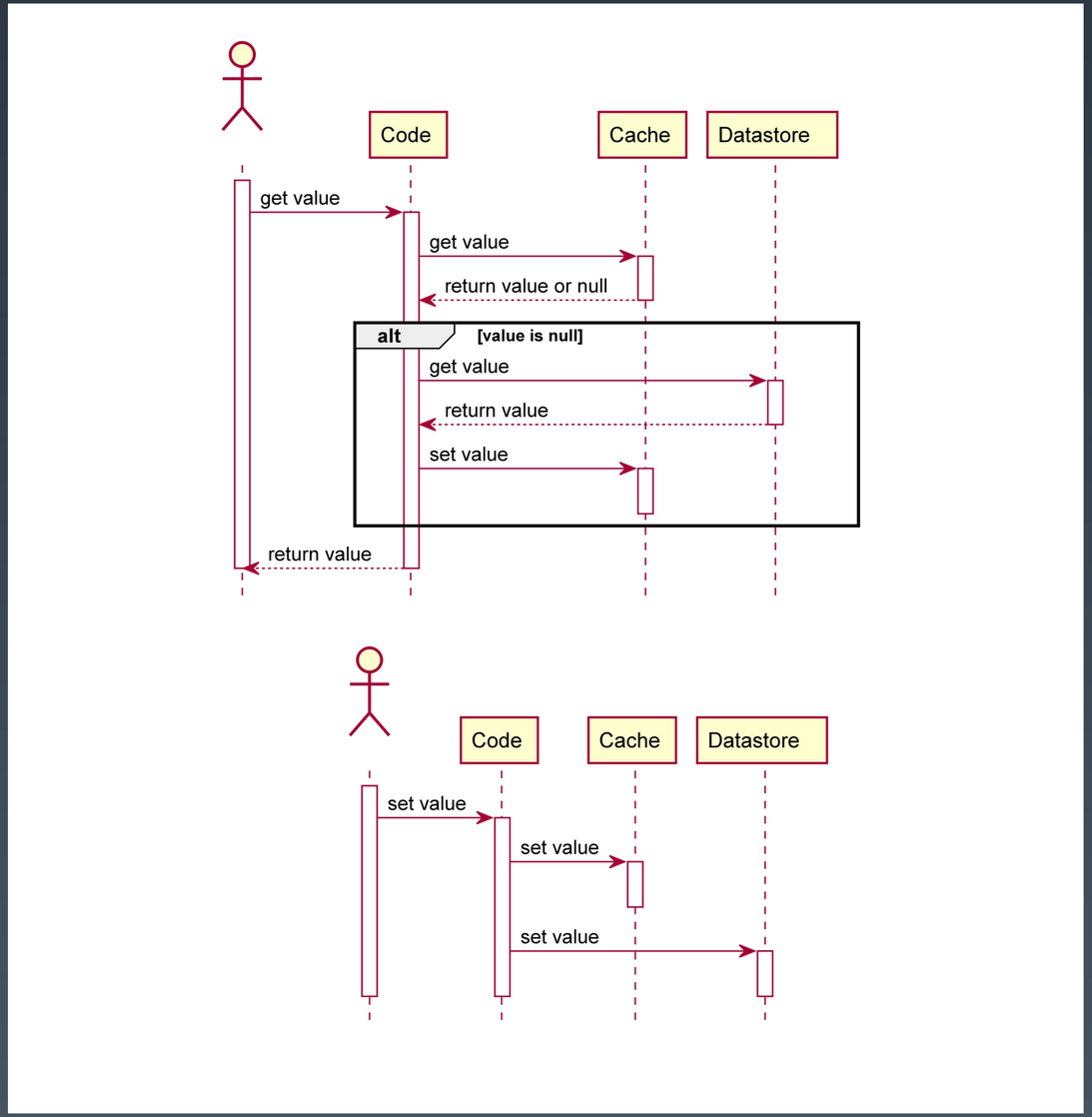
系统设计——缓存模式 cache aside

Cache aside:

- 把 Cache 当成一个普通的数据源
- 更新 Cache 和 DB 都是依赖于开发者自己写代码

读，用户代码负责：

1. 未命中的时候直接返回 null （避免穿透），而后异步查询数据库并且写到 cache；
2. 采用 singleflight



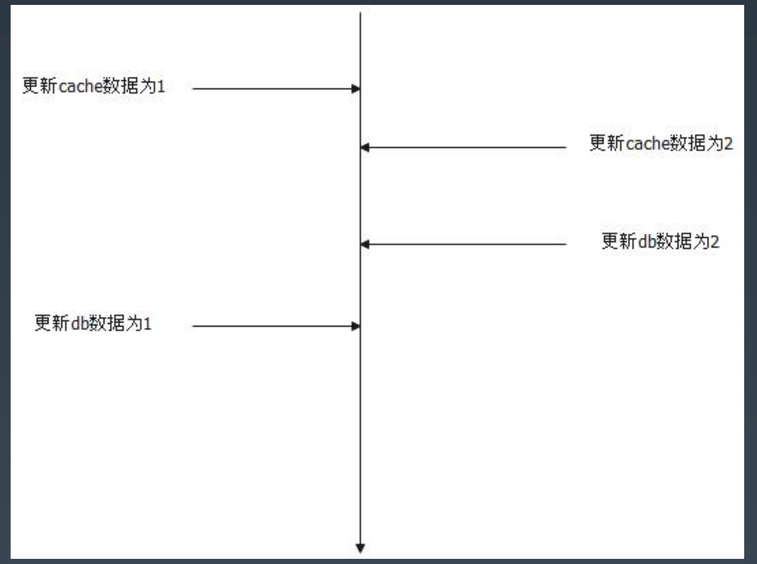
系统设计——缓存模式 cache aside

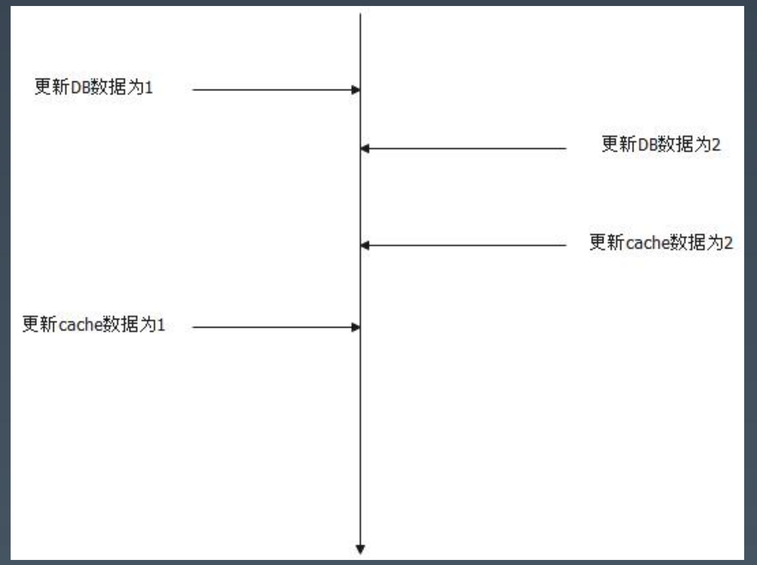
Cache aside:

- 把 Cache 当成一个普通的数据源
- 更新 Cache 和 DB 都是依赖于开发者自己写代码

写：

- 先写 cache，数据库可能更新失败，或者数据不一致
- 先写数据库，再写 cache，cache 失败，或者数据不一致
- 先写数据库，删除原本的 cache，读数据可能导致穿透

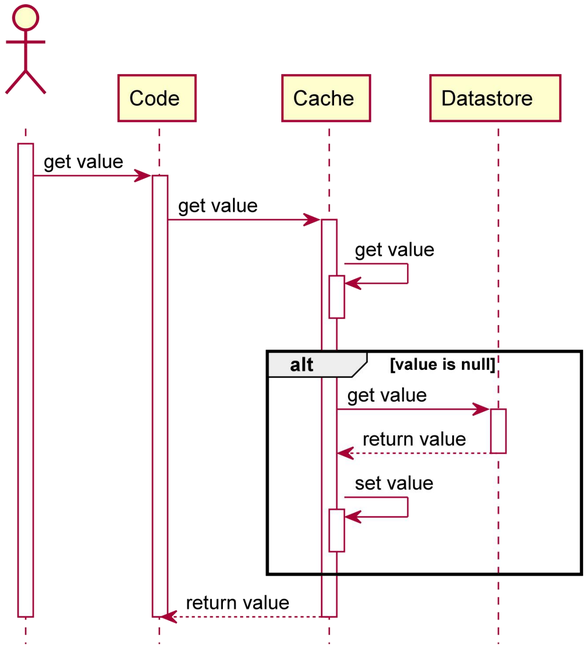




系统设计——缓存模式 read-through

read through:

- 开发者只需要从 cache 中读取数据，cache 会在缓存不命中的时候去读取数据
- 更新数据的时候可以直接更新 DB，等待 Cache 过期。也可以更新 DB 之后同步更新 Cache。如何更新依赖于开发者自己写代码

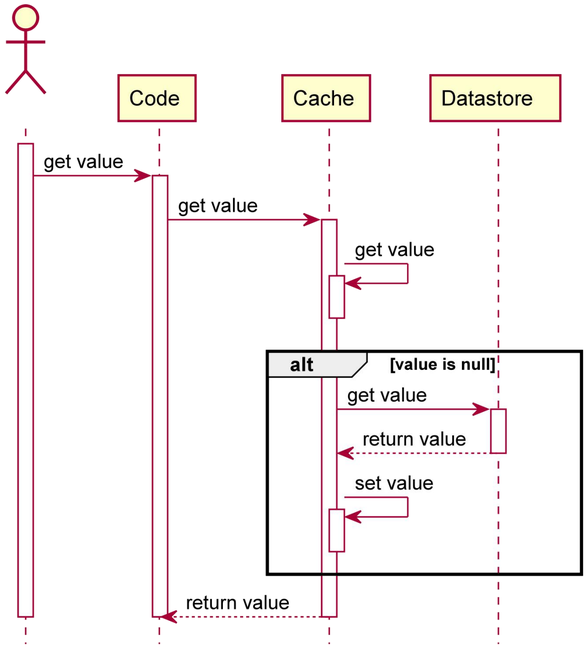


系统设计——缓存模式 read-through

read through:

- 开发者只需要从 cache 中读取数据，cache 会在缓存不命中的时候去读取数据
- 更新数据的时候可以直接更新 DB，等待 Cache 过期。也可以更新 DB 之后同步更新 Cache。如何更新依赖于开发者自己写代码

读写要考虑的问题和 cache aside 一样

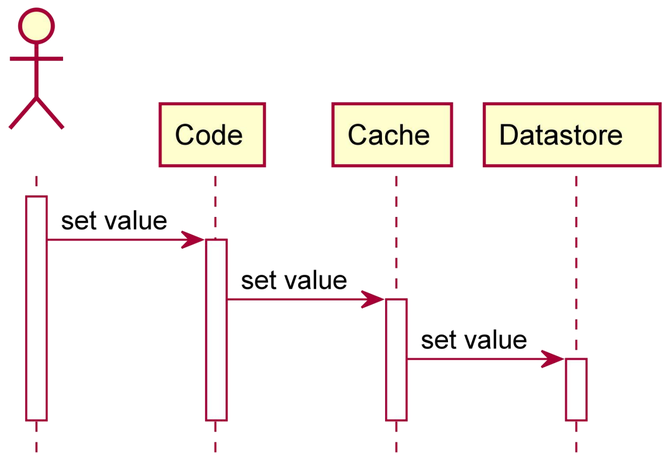


系统设计——缓存模式 write-through

write through:

- 开发者只需要写入 cache，cache 自己会更新数据库。
- 在读未命中缓存的情况下，开发者需要自己去数据库捞数据，然后更新缓存（此时缓存不需要更新 DB 了）

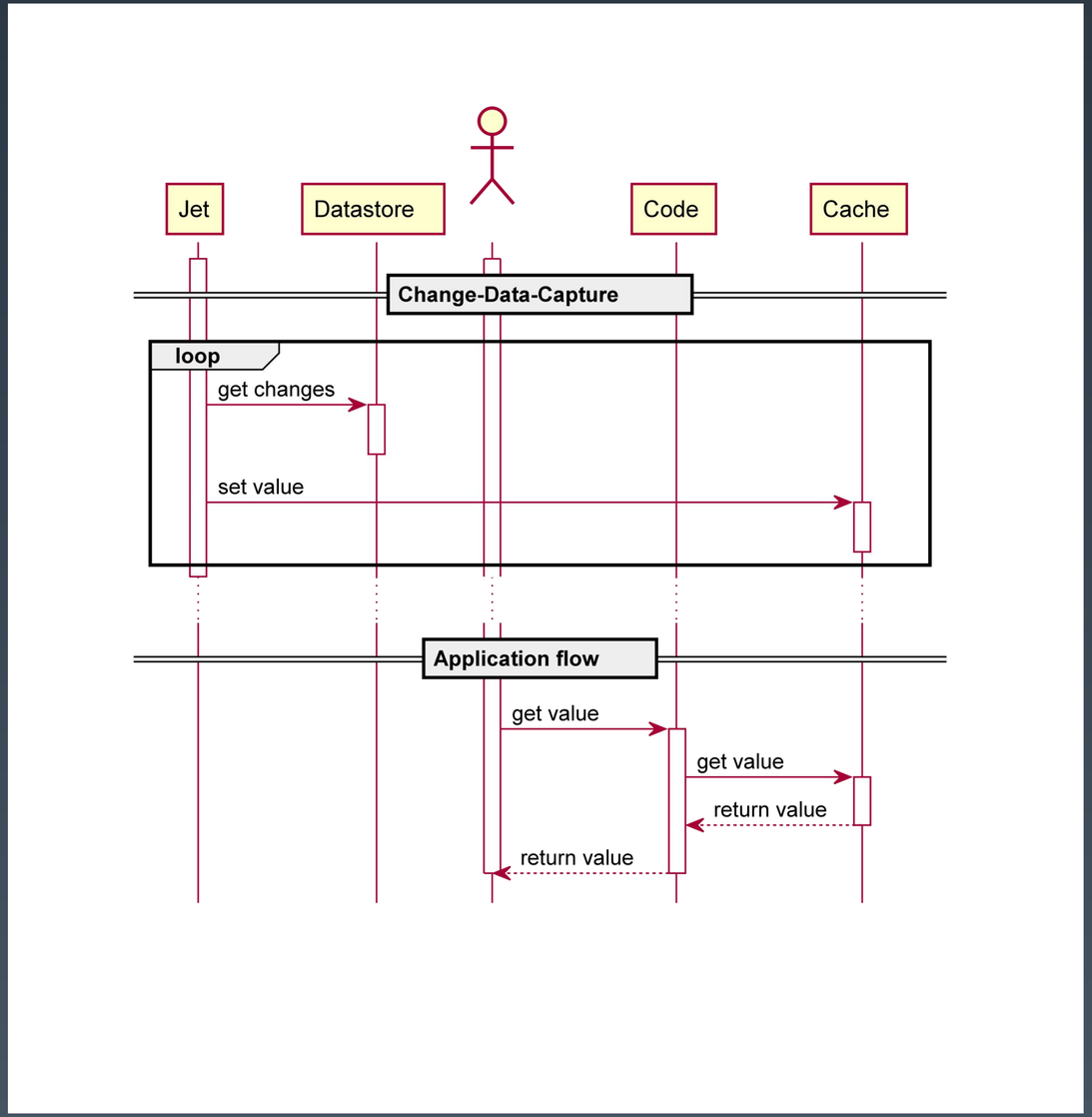
读写要考虑的问题和 cache aside 一样



系统设计——缓存模式 refresh-ahead

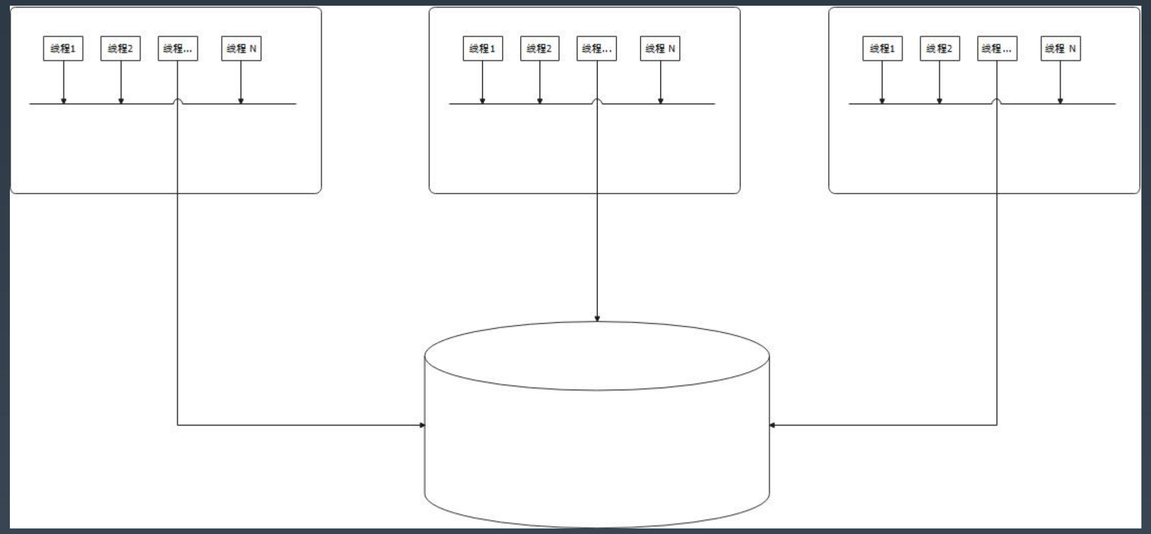
refresh-ahead 依赖于 CDC(changed data capture) 接口：

- 数据库暴露数据变更接口
- cache 或者第四方在监听到数据变更之后自动更新数据



系统设计——singleflight

别的语言也是可以用的。核心在于在单个进程内，所有的线程（协程）抢夺锁，从数据库里面加载资源，除了拿到锁的线程（协程），其余线程（协程）停下来等待。

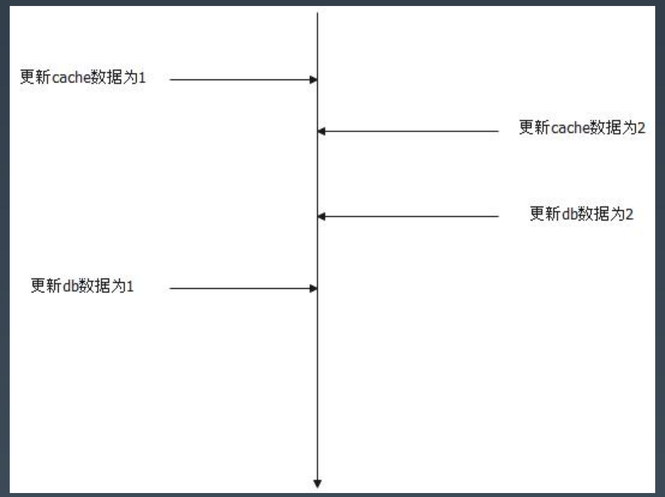


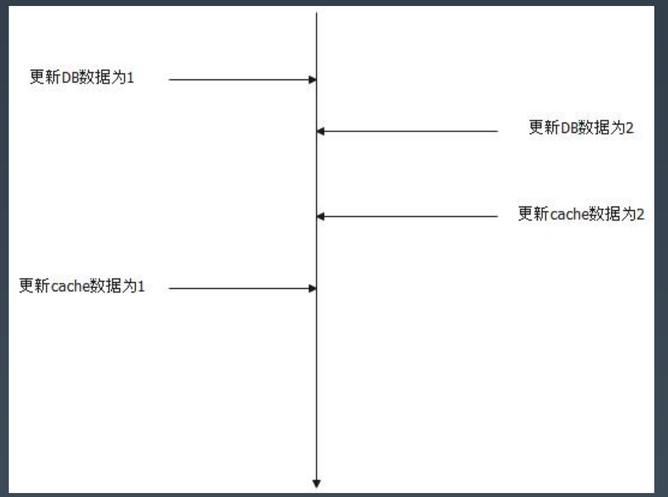
和应用实例个数，key 成正比。前面的 cache pattern 如果尝试从数据库中读取数据，都可以应用

系统设计——删繁就简

模式	miss 的时候谁刷新缓存	更新的时候谁更新 DB	
cache aside	用户	用户	
read through	cache	用户	
write through	用户	cache	
read throgh + write through	cache	cache	用户根本不知道 cache 后面还有啥
refresh ahead	cache 或者第四方	用户	主要是监听数据变更接口，监听者就负责更新 DB

系统设计——删繁就简


更新cache数据为1
更新cache数据为2
更新db数据为2
更新db数据为1


更新DB数据为1
更新DB数据为2
更新cache数据为2
更新cache数据为1

References

- https://www.r-5.org/files/books/computers/languages/sql/style/Bill_Karwin-SQL_Antipatterns-EN.pdf
- https://hazelcast.com/blog/a-hitchhikers-guide-to-caching-patterns/