bluebell用户模块

user表结构设计

```
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
    `id` bigint(20) NOT NULL AUTO_INCREMENT,
    `user_id` bigint(20) NOT NULL,
    `username` varchar(64) COLLATE utf8mb4_general_ci NOT NULL,
    `password` varchar(64) COLLATE utf8mb4_general_ci NOT NULL,
    `email` varchar(64) COLLATE utf8mb4_general_ci,
    `gender` tinyint(4) NOT NULL DEFAULT '0',
    `create_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
    `update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`),
    UNIQUE KEY `idx_username` (`username`) USING BTREE,
    UNIQUE KEY `idx_user_id` (`user_id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

分布式ID生成器

分布式ID的特点

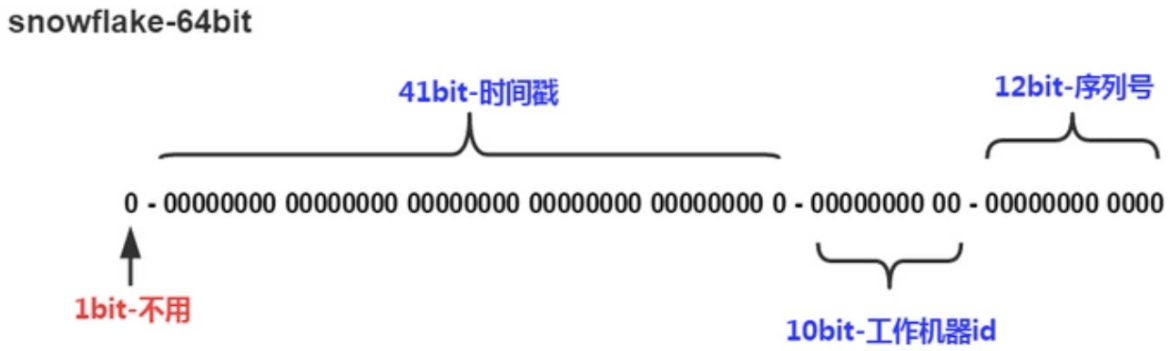
- 全局唯一性：不能出现有重复的ID标识，这是基本要求。
- 递增性：确保生成ID对于用户或业务是递增的。
- 高可用性：确保任何时候都能生成正确的ID。
- 高性能性：在高并发的环境下依然表现良好。

不仅仅是用于用户ID，实际互联网中有很多场景需要能够生成类似MySQL自增ID这样不断增大，同时又不会重复的id。以支持业务中的高并发场景。

比较典型的场景有：电商促销时短时间内会有大量的订单涌入到系统，比如每秒10w+；明星出轨时微博短时间内会产生大量的相关微博转发和评论消息。在这些业务场景下将数据插入数据库之前，我们需要给这些订单和消息先分配一个唯一ID，然后再保存到数据库中。对这个id的要求是希望其中能带有一些时间信息，这样即使我们后端的系统对消息进行了分库分表，也能够以时间顺序对这些消息进行排序。

snowflake算法介绍

雪花算法，它是Twitter开源的由64位整数组成分布式ID，性能较高，并且在单机上递增。



1.第一位 占用1bit，其值始终是0，没有实际作用。

2.时间戳 占用41bit，单位为毫秒，总共可以容纳约69年的时间。当然，我们的时间毫秒计数不会真的从1970年开始记，那样我们的系统跑到`2039/9/7 23:47:35`就不能用了，所以这里的时间戳只是相对于某个时间的增量，比如我们的系统上线是2020-07-01，那么我们完全可以把这个timestamp当作是从`2020-07-01 00:00:00.000`的偏移量。

3.工作机器id 占用10bit，其中高位5bit是数据中心ID，低位5bit是工作节点ID，最多可以容纳1024个节点。

4.序列号 占用12bit，用来记录同毫秒内产生的不同id。每个节点每毫秒0开始不断累加，最多可以累加到4095，同一毫秒一共可以产生4096个ID。

SnowFlake算法在同一毫秒内最多可以生成多少个全局唯一ID呢?

同一毫秒的ID数量 = 1024 X 4096 = 4194304

snowflake的Go实现

1. https://github.com/bwmarrin/snowflake

github.com/bwmarrin/snowflake是一个相当轻量化的snowflake的Go实现。

1 Bit Unused	41 Bit Timestamp	10 Bit NodeID	12 Bit Sequence ID

```
package main

import (
  "fmt"
  "time"
```

```
  "github.com/bwmarrin/snowflake"
)

var node *snowflake.Node

func Init(startTime string, machineID int64) (err error) {
  var st time.Time
  st, err = time.Parse("2006-01-02", startTime)
  if err != nil {
    return
  }
  snowflake.Epoch = st.UnixNano() / 1000000
  node, err = snowflake.NewNode(machineID)
  return
}

func GenID() int64 {
  return node.Generate().Int64()
}

func main() {
  if err := Init("2020-07-01", 1); err != nil {
    fmt.Printf("init failed, err:%v\n", err)
    return
  }
  id := GenID()
  fmt.Println(id)
}
```

2. https://github.com/sony/sonyflake

sonyflake是Sony公司的一个开源项目，基本思路和snowflake差不多，不过位分配上稍有不同:

这里的时间只用了39个bit，但时间的单位变成了10ms，所以理论上比41位表示的时间还要久(174年)。

`Sequence ID`和之前的定义一致，`Machine ID`其实就是节点id。`sonyflake`库有以下配置参数：

```
type Settings struct {
  StartTime      time.Time
  MachineID      func() (uint16, error)
  CheckMachineID func(uint16) bool
}
```

其中：

- `StartTime`选项和我们之前的`Epoch`差不多，如果不设置的话，默认是从`2014-09-01 00:00:00 +0000 UTC`开始。
- `MachineID`可以由用户自定义的函数，如果用户不定义的话，会默认将本机IP的低16位作为`machine id`。
- `CheckMachineID`是由用户提供的检查`MachineID`是否冲突的函数。

使用示例：

```
package main

import (
  "fmt"
  "time"

  "github.com/sony/sonyflake"
)

var (
  sonyFlake     *sonyflake.Sonyflake
  sonyMachineID uint16
)

func getMachineID() (uint16, error) {
  return sonyMachineID, nil
}

// 需传入当前的机器ID
func Init(startTime string, machineId uint16) (err error) {
  sonyMachineID = machineId
  var st time.Time
  st, err = time.Parse("2006-01-02", startTime)
  if err != nil {
    return err
  }
  settings := sonyflake.Settings{
    StartTime: st,
    MachineID: getMachineID,
  }
  sonyFlake = sonyflake.NewSonyflake(settings)
  return
}

// GenID 生成id
func GenID() (id uint64, err error) {
  if sonyFlake == nil {
    err = fmt.Errorf("snoy flake not inited")
    return
```

```
  }

  id, err = sonyFlake.NextID()
  return
}

func main() {
  if err := Init("2020-07-01", 1); err != nil {
    fmt.Printf("Init failed, err:%v\n", err)
    return
  }
  id, _ := GenID()
  fmt.Println(id)
}
```

登录注册流程

注册流程

提交注册信息 --> 参数校验 --> 入库 --> 注册成功

登录流程

提交登录信息 --> 参数校验 --> 查询数据库 --> 登陆成功 --> 下发Token

用户认证

HTTP 是一个无状态的协议，一次请求结束后，下次在发送服务器就不知道这个请求是谁发来的了（同一个 IP 不代表同一个用户），在 Web 应用中，用户的认证和鉴权是非常重要的一环，实践中有多种可用方案，并且各有千秋。

Cookie- Session 认证模式

在 Web 应用发展的初期，大部分采用基于Cookie-Session 的会话管理方式，逻辑如下。

- 客户端使用用户名、密码进行认证
- 服务端验证用户名、密码正确后生成并存储 Session，将 SessionID 通过 Cookie 返回给客户端
- 客户端访问需要认证的接口时在 Cookie 中携带 SessionID
- 服务端通过 SessionID 查找 Session 并进行鉴权，返回给客户端需要的数据

基于 Session 的方式存在多种问题。

- 服务端需要存储 Session，并且由于 Session 需要经常快速查找，通常存储在内存或内存数据库中，同时在线用户较多时需要占用大量的服务器资源。
- 当需要扩展时，创建 Session 的服务器可能不是验证 Session 的服务器，所以还需要将所有 Session 单独存储并共享。
- 由于客户端使用 Cookie 存储 SessionID，在跨域场景下需要进行兼容性处理，同时这种方式也难以防范 CSRF 攻击。

Token 认证模式

鉴于基于 Session 的会话管理方式存在上述多个缺点，基于 Token 的无状态会话管理方式诞生了，所谓无状态，就是服务端可以不再存储信息，甚至是不再存储 Session，逻辑如下。

- 客户端使用用户名、密码进行认证
- 服务端验证用户名、密码正确后生成 Token 返回给客户端
- 客户端保存 Token，访问需要认证的接口时在 URL 参数或 HTTP Header 中加入 Token
- 服务端通过解码 Token 进行鉴权，返回给客户端需要的数据

基于 Token 的会话管理方式有效解决了基于 Session 的会话管理方式带来的问题。

- 服务端不需要存储和用户鉴权有关的信息，鉴权信息会被加密到 Token 中，服务端只需要读取 Token 中包含的鉴权信息即可
- 避免了共享 Session 导致的不易扩展问题
- 不需要依赖 Cookie，有效避免 Cookie 带来的 CSRF 攻击问题
- 使用 CORS 可以快速解决跨域问题

JWT介绍

JWT 是 JSON Web Token 的缩写，是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准（(RFC 7519)。JWT 本身没有定义任何技术实现，它只是定义了一种基于 Token 的会话管理的规则，涵盖 Token 需要包含的标准内容和 Token 的生成过程，特别适用于分布式站点的单点登录（SSO）场景。。

一个 JWT Token 就像这样：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoyODAxODcyNzQ4ODMyMzU4NSwiZXhwIjoxNTk0NTQwMjkxLCJpc3MiOiJibHVlYmVsbCJ9.lk_ZrAtYGCeZhK3iupHxP1kgjBJzQTVTtX0iZYFx9wU
```

它是由`.`分隔的三部分组成，这三部分依次是：

- 头部（Header）
- 负载（Payload）

- 签名（Signature）

头部和负载以 JSON 形式存在，这就是 JWT 中的 JSON，三部分的内容都分别单独经过了 Base64 编码，以 `.` 拼接成一个 JWT Token。

Header

JWT 的 Header 中存储了所使用的加密算法和 Token 类型。

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload

Payload 表示负载，也是一个 JSON 对象，JWT 规定了7个官方字段供选用，

```
iss (issuer)：签发人
exp (expiration time)：过期时间
sub (subject)：主题
aud (audience)：受众
nbf (Not Before)：生效时间
iat (Issued At)：签发时间
jti (JWT ID)：编号
```

除了官方字段，开发者也可以自己指定字段和内容，例如下面的内容。

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

注意，JWT 默认是不加密的，任何人都可以读到，所以不要把秘密信息放在这个部分。这个 JSON 对象也要使用 Base64URL 算法转成字符串。

Signature

Signature 部分是对前两部分的签名，防止数据篡改。

首先，需要指定一个密钥（secret）。这个密钥只有服务器才知道，不能泄露给用户。然后，使用 Header 里面指定的签名算法（默认是 HMAC SHA256），按照下面的公式产生签名。

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload),secret)
```

JWT优缺点

JWT 拥有基于 Token 的会话管理方式所拥有的一切优势，不依赖 Cookie，使得其可以防止 CSRF 攻击，也能在禁用 Cookie 的浏览器环境中正常运行。

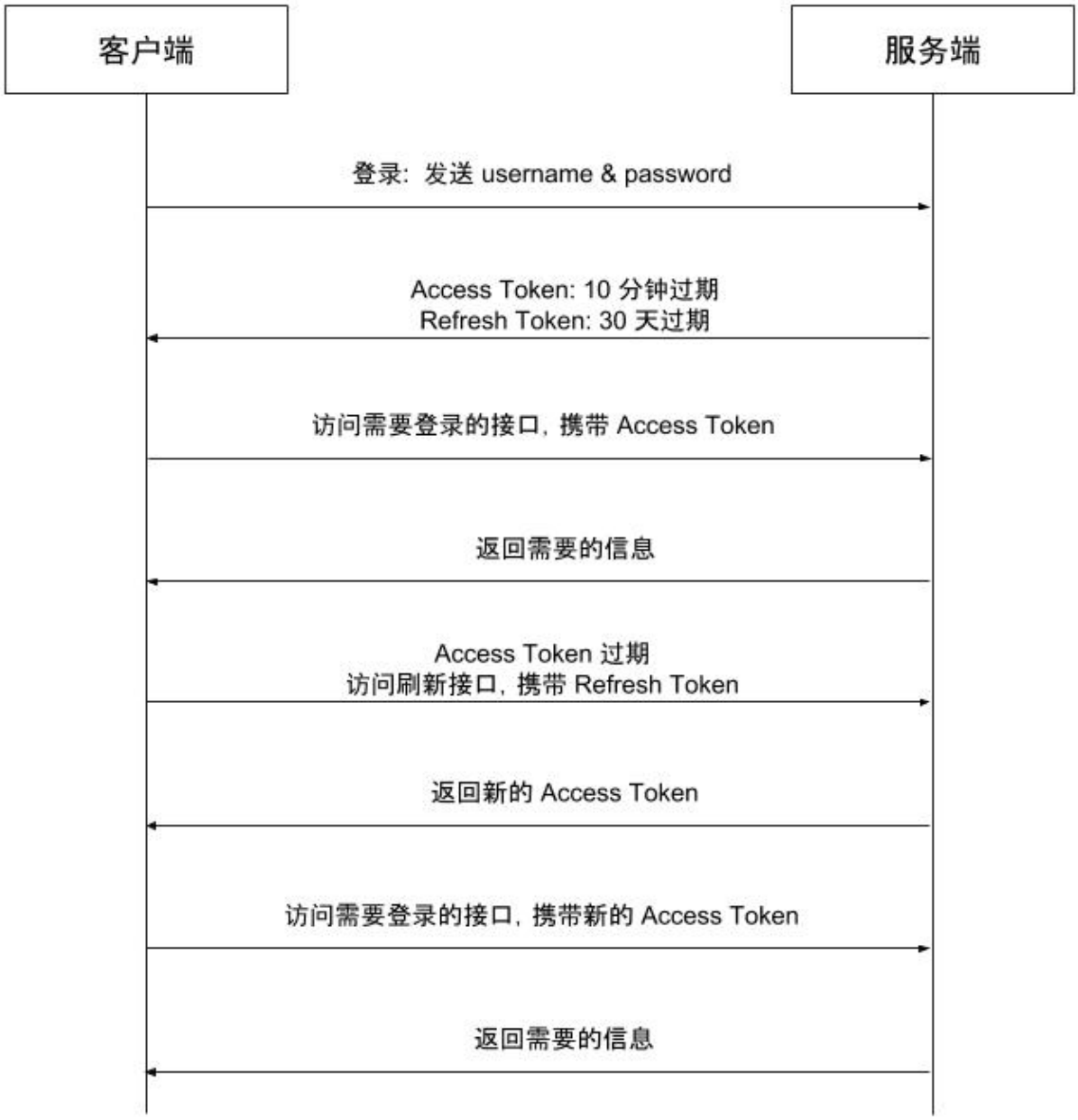
而 JWT 的最大优势是服务端不再需要存储 Session，使得服务端认证鉴权业务可以方便扩展，避免存储 Session 所需要引入的 Redis 等组件，降低了系统架构复杂度。但这也是 JWT 最大的劣势，由于有效期存储在 Token 中，JWT Token 一旦签发，就会在有效期内一直可用，无法在服务端废止，当用户进行登出操作，只能依赖客户端删除掉本地存储的 JWT Token，如果需要禁用用户，单纯使用 JWT 就无法做到了。

基于jwt实现认证实践

前面讲的 Token，都是 Access Token，也就是访问资源接口时所需要的 Token，还有另外一种 Token，Refresh Token，通常情况下，Refresh Token 的有效期会比较长，而 Access Token 的有效期比较短，当 Access Token 由于过期而失效时，使用 Refresh Token 就可以获取到新的 Access Token，如果 Refresh Token 也失效了，用户就只能重新登录了。

在 JWT 的实践中，引入 Refresh Token，将会话管理流程改进如下。

- 客户端使用用户名密码进行认证
- 服务端生成有效时间较短的 Access Token（例如 10 分钟），和有效时间较长的 Refresh Token（例如 7 天）
- 客户端访问需要认证的接口时，携带 Access Token
- 如果 Access Token 没有过期，服务端鉴权后返回给客户端需要的数据
- 如果携带 Access Token 访问需要认证的接口时鉴权失败（例如返回 401 错误），则客户端使用 Refresh Token 向刷新接口申请新的 Access Token
- 如果 Refresh Token 没有过期，服务端向客户端下发新的 Access Token
- 客户端使用新的 Access Token 访问需要认证的接口



后端需要对外提供一个刷新Token的接口，前端需要实现一个当Access Token过期时自动请求刷新Token接口获取新Access Token的拦截器。

gin框架使用jwt

`jwt-go`库的基本使用详见我的博客链接：https://www.liwenzhou.com/posts/Go/jwt_in_gin/

鉴权中间件开发

```
const (
  ContextUserIDKey = "userID"
)

var (
  ErrorUserNotLogin = errors.New("当前用户未登录")
)
```

```
// JWTAuthMiddleware 基于JWT的认证中间件
func JWTAuthMiddleware() func(c *gin.Context) {
  return func(c *gin.Context) {
    // 客户端携带Token有三种方式 1.放在请求头 2.放在请求体 3.放在URI
    // 这里假设Token放在Header的中
    // 这里的具体实现方式要依据你的实际业务情况决定
    authHeader := c.Request.Header.Get("Auth")
    if authHeader == "" {
      ResponseErrorWithMsg(c, CodeInvalidToken, "请求头缺少Auth Token")
      c.Abort()
      return
    }
    mc, err := utils.ParseToken(authHeader)
    if err != nil {
      ResponseError(c, CodeInvalidToken)
      c.Abort()
      return
    }
    // 将当前请求的username信息保存到请求的上下文c上
    c.Set(ContextUserIDKey, mc.UserID)
    c.Next() // 后续的处理函数可以用过c.Get("userID")来获取当前请求的用户信息
  }
}
```

生成access token和refresh token

```
// GenToken 生成access token 和 refresh token
func GenToken(userID int64) (aToken, rToken string, err error) {
  // 创建一个我们自己的声明
  c := MyClaims{
    userID, // 自定义字段
    jwt.StandardClaims{
      ExpiresAt: time.Now().Add(TokenExpireDuration).Unix(), // 过期时间
      Issuer:    "bluebell",                                 // 签发人
    },
  }
  // 加密并获得完整的编码后的字符串token
  aToken, err = jwt.NewWithClaims(jwt.SigningMethodHS256,
c).SignedString(mySecret)

  // refresh token 不需要存任何自定义数据
  rToken, err = jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.StandardClaims{
    ExpiresAt: time.Now().Add(time.Second * 30).Unix(), // 过期时间
    Issuer:    "bluebell",                              // 签发人
  }).SignedString(mySecret)
  // 使用指定的secret签名并获得完整的编码后的字符串token
  return
```

```
}
```

解析access token

```
// ParseToken 解析JWT
func ParseToken(tokenString string) (claims *MyClaims, err error) {
  // 解析token
  var token *jwt.Token
  claims = new(MyClaims)
  token, err = jwt.ParseWithClaims(tokenString, claims, keyFunc)
  if err != nil {
    return
  }
  if !token.Valid { // 校验token
    err = errors.New("invalid token")
  }
  return
}
```

refresh token

```
// RefreshToken 刷新AccessToken
func RefreshToken(aToken, rToken string) (newAToken, newRToken string, err
error) {
  // refresh token无效直接返回
  if _, err = jwt.Parse(rToken, keyFunc); err != nil {
    return
  }

  // 从旧access token中解析出claims数据
  var claims MyClaims
  _, err = jwt.ParseWithClaims(aToken, &claims, keyFunc)
  v, _ := err.(*jwt.ValidationError)

  // 当access token是过期错误 并且 refresh token没有过期时就创建一个新的access token
  if v.Errors == jwt.ValidationErrorExpired {
    return GenToken(claims.UserID)
  }
  return
}
```

参考链接

- http://www.ruanyifeng.com/blog/2018/07/json_web_token-tutorial.html
- https://www.jianshu.com/p/25ab2f456904