# 绪论

随着移动办公和云计算的快速发展，对于多终端实时同步备份系统的需求越来越高。本论文设计和实现了一个基于Golang和云存储技术的多用户多终端实时增量数据同步备份系统，旨在解决现代化信息技术中数据同步备份的问题。

该系统采用了Golang作为服务端和客户端的主要编程语言，服务端使用MinIO和PostgreSQL 进行文件存储和和数据持久化，并利用Redis的发布和订阅功能来广播和接收文件变化信息。客户端利用http和websocket传输协议与服务器进行数据通信，接收并处理云端文件的变化。

在本文中，我们详细介绍了该系统的设计和实现，并对系统的性能和可靠性进行了测试和分析。测试结果表明，该系统能够实现高效、可靠的数据同步备份，并具有较好的性能和稳定性。在系统的研究和实现对于解决多用户多终端数据同步备份问题上具有重要的理论和实际意义，并在移动办公、在线教育、医疗健康等领域具有广阔的应用前景

同步 备份 云存储 多终端 实时 增量 Golang MinIO Websocket

With the rapid development of mobile office and cloud computing, the demand for multi-terminal real-time sync and backup systems is increasing. This paper designs and implements a multi-user multi-terminal real-time incremental data synchronization and backup system based on Golang and cloud storage technology, aimed at solving the problem of data synchronization and backup in modern information technology.

The system uses Golang as the main programming language for both the server and client sides, and employs MinIO and PostgreSQL for file storage and data persistence, and utilizes Redis' publish and subscribe function to broadcast and receive file change information. The client communicates with the server using the HTTP and WebSocket protocols to receive and process changes to cloud files.

In this paper, we detail the design and implementation of the system and test and analyze its performance and reliability. The test results show that the system can achieve efficient and reliable data synchronization and backup, and has good performance and stability. The research and implementation of the system have important theoretical and practical significance for solving the problem of multi-user and multi-terminal data synchronization and backup, and have broad application prospects in fields such as mobile office, online education, and healthcare.

synchronization, backup, cloud storage, multi-terminal, real-time, incremental

# 研究背景和意义

现代社会的信息化程度越来越高，个人和企业的数据存储和备份需求也越来越大。而云存储技术可以为用户提供更加安全和可靠的数据存储和备份方案，避免数据丢失和泄露的风险。 随着操作系统和设备类型的不断更新和更迭，我们通常会使用多个终端设备（例如手机、电脑、平板等）来处理数据和信息。而基于云文件的多用户多终端实时增量数据同步备份系统可以帮助用户在多个终端设备上实时同步数据，提高数据的可用性和便捷性。传统的全量备份技术通常需要备份整个数据集，备份时间和空间成本都很高。而增量备份技术可以只备份变化的部分数据，大大减少备份时间和空间成本。因此，基于云文件的多用户多终端实时增量数据同步备份系统可以大幅降低备份成本。基于云文件的多用户多终端实时增量数据同步备份系统可以为多个用户提供数据共享和协作的平台。例如，团队成员可以共享项目文档，提高协作效率。

因此，基于云文件的多用户多终端实时增量数据同步备份系统具有重要的研究意义和实际应用价值。它可以帮助用户在多个终端之间实现数据的自动备份和同步，并且可以快速地恢复到之前的状态，避免数据的丢失和损坏，提高数据的安全性和可靠性。同时，它还可以帮助用户实现数据的共享和协作，提高工作效率和协同能力。

，用户需要的数据备份和同步系统也需要具备跨平台的能力。基于云文件的多用户多终端实时增量数据同步备份系统可以在多种操作系统和设备类型之间实现数据同步和备份。

提高数据的可靠性：传统的本地备份方法容易受到病毒、硬件故障、误操作等因素的影响，而基于云文件的备份系统可以实现数据的自动备份和同步，提高了数据的可靠性。

提高数据的可用性：基于云文件的备份系统可以实现数据的实时同步，可以随时随地访问和管理备份数据，提高了数据的可用性。

# 国内外的研究现状

在中国，随着云计算技术的发展和普及，云存储备份系统逐渐受到重视。目前，国内的很多企业和研究机构都在探索基于云存储的多用户多终端实时增量数据同步备份系统。例如，百度云盘、阿里云盘，坚果云等厂商都提供了相应的备份服务，可以实现多用户多终端数据的备份和同步。此外，国内的一些大学和科研机构也在进行相关研究，如清华大学、中科院等。他们通过分布式算法、增量备份技术、加密算法等技术手段，不断提高备份系统的效率、可靠性和安全性。

在国外，基于云存储的备份系统也是一个研究热点。美国、欧洲等发达国家在这方面的研究更加深入和广泛，涉及到的领域也更加多元化。例如，Dropbox、Google Drive、Microsoft OneDrive等云存储备份服务已经成为了国际上备受欢迎的备份工具，其多用户多终端的备份功能得到了广泛的应用。此外，还有一些学术界和行业界的研究团队在进行相关研究，他们关注的问题也更加深入和复杂，如数据一致性、网络带宽利用等方面。

总的来说，基于云文件的多用户多终端实时增量数据同步备份系统在国内外都得到了广泛的研究和应用，相关的产品和服务也得到了用户的认可和使用。虽然在研究重点、应用场景等方面存在差异，但是都致力于提高备份系统的效率、可靠性和安全性。随着云计算和大数据等领域的发展，该领域的研究和应用还将不断拓展和深入。

# 研究内容

基于云文件的多用户多终端实时增量数据同步备份系统的研究内容包括以下方面：

1. 系统架构设计：设计基于云存储的多用户多终端实时增量数据同步备份系统的整体架构，包括 存储文件和文件夹关系 服务端和客户端的数据传输协议 客户端与服务端的交互流程、文件数据存储方式。

2. 多终端同步算法设计：设计并实现文件增量同步算法 以提高同步效率和减少带宽开销，文件变化消息广播与接收

3. 客户端设计： 程序启动文件对比 同步备份任务创建和恢复初始话 监控本地同步文件夹 文件变化传递和处理 文件过滤限制，文件新增冲突

4. 适配不同平台： 文件路径 配置文件 界面交互

5. 安全性保障：设计并实现数据加密、身份认证、权限控制等安全机制，以确保用户数据的隐私和安全性。

6. 系统优化：对系统进行优化，提高系统的可扩展性、可靠性和性能。

系统架构

本系统采用C/S架构设计，C/S架构模式是一种常见的客户端/服务器架构模式，其中客户端应用程序通过网络连接到服务器应用程序，以获取所需的数据和服务。C/S架构模式通常包含一个客户端软件和一个服务器软件，客户端软件提供用户界面和应用程序逻辑，而服务器软件提供数据存储和计算能力。

客户端主要负责用户交互和本地文件管理，服务器端主要负责文件数据保存和同步消息处理。为了实现同步文件功能，客户端需要读取并上传用户本地文件，监视本地文件数据变更信息，接受并处理其他客户端的文件变更信息和用户交互，而服务端则需要提供数据存储API来保存用户文件和文件夹等数据，广播发送文件变更消息来实现多客户端实时同步。

系统技术

* 1. Golang

使用Golang 来作为服务端和客户端的主要语言，Golang也称为Go语言，是由Google开发的一种开源编程语言。它具有静态类型、高效、可扩展、并发性强等特点，适用于各种应用程序的开发，包括服务器端应用、网络应用、分布式系统、云计算、数据库应用、操作系统等。它的语法简洁，同时具有高并发和高性能的特点，能够轻松地处理大规模的数据处理任务。

Golang，也被称为Go，是一种由Google开发的开源编程语言，于2009年首次亮相。它是一种高效、快速、简单、安全和并发性强的编程语言，被设计用于大规模分布式系统和网络服务应用程序的开发。Golang的语法简单易学，它拥有丰富的标准库，可以轻松地处理网络编程、并发编程、加密、解密等诸多任务，而不需要依赖第三方库。

Golang的编译速度非常快，生成的二进制文件也非常小，这使得它非常适合用于云计算和网络应用程序的开发。此外，Golang还支持垃圾回收机制，可以有效地管理内存，减少了程序员的负担。

Golang在并发编程方面表现非常突出，它提供了一种称为"goroutine"的并发机制，可以轻松地创建和管理成千上万个并发执行的任务。此外，Golang还提供了一个称为"channel"的机制，用于在不同的goroutine之间通信和同步，这使得编写并发程序变得更加简单和可靠。

总之，Golang是一种高效、快速、简单、安全和并发性强的编程语言，被广泛应用于网络应用程序、云计算和大规模分布式系统等领域。它的语法简单易学，标准库丰富，编译速度快，内存管理效率高，并发编程支持出色，这使得它成为一个非常流行的编程语言。

Gin

使用Gin来作为服务端的Web框架，Gin 基于Golang，使用了Golang的高效性能和易于使用的语法，旨在提供一种轻量级、快速、可扩展的Web框架。Gin框架提供了一些有用的特性，例如路由、中间件、错误管理、日志记录、请求和响应处理等。

它的优点：

1. 快速：Gin采用了基于Radix树的路由算法，能够快速地处理大量的路由请求。
2. 轻量级：Gin框架仅仅依赖于标准库和第三方库，因此非常轻量级，可以方便地集成到现有的项目中。
3. 易于使用：Gin采用了类似于Express的API风格，语法简单易懂，开发者可以快速上手。
4. 可扩展：Gin框架提供了中间件机制，可以方便地扩展框架功能。
5. 错误管理：Gin框架提供了简单易用的错误管理功能，可以很好地处理请求和响应中的错误。
6. 日志记录：Gin框架提供了日志记录功能，方便开发者进行调试和错误追踪。
   1. Sqlite3

Sqlite3是一个轻量级的嵌入式关系型数据库管理系统，它是在公有领域下发行的免费软件。与传统的大型数据库管理系统（如MySQL、Oracle等）不同，SQLite3没有独立的服务器进程，而是将整个数据库作为一个文件存储在主机的文件系统中，通过读写这个文件来实现对数据库的操作。这种设计使得SQLite3非常适合嵌入在应用程序中，尤其是对于需要高效、可靠的本地数据存储的应用程序。

Sqlite3是一种轻量级的关系型数据库，其代码库非常小，仅仅只有几百KB大小。它的设计目标之一是免去繁琐的服务器配置和管理，使用者可以直接在应用程序中调用SQLite3库进行数据库操作，同时它也可以在多种操作系统上运行，包括Windows、Linux、macOS等，这使得它非常适合开发跨平台的应用程序。Sqlite3具有非常简单的API，易于使用和学习，同时也支持SQL语言，可以方便地进行数据操作和查询。

考虑到客户端需要一个简单且性能不错的数据库，综上所述，使用Sqlite3作为客户端数据持久化数据库为最优选择。

* 1. Flutter

Flutter 是一个跨平台的移动应用程序开发框架，可以使用单一代码库构建高性能、高保真度、原生体验的应用程序，支持 iOS、Android、Web、Linux、Windows、MacOS 等多个平台。Flutter 采用 Dart 语言作为开发语言，Dart 是一种面向对象、类似于 Java 和 JavaScript 的语言，具有类型安全、高性能、可扩展性和可移植性等优点。

在同步文件系统的客户端中，选择使用 Flutter 作为开发框架，有以下几个理由：

跨平台支持：Flutter 可以同时支持多个平台的应用程序开发，无需为不同平台编写不同的代码，大大提高了开发效率和代码复用率。

高性能和流畅度：Flutter 使用自己的渲染引擎 Skia 来绘制 UI，可以实现高性能的 UI 绘制和流畅的动画效果，能够提供良好的用户体验。

丰富的组件库：Flutter 提供了丰富的组件库，包括 Material Design 和 Cupertino 风格的组件，可以快速构建漂亮的用户界面，减少开发工作量。

热重载：Flutter 支持热重载，可以在不重新启动应用程序的情况下，快速查看 UI 变化，大大提高了开发效率。

强大的开发工具：Flutter 集成了强大的开发工具，包括 Flutter CLI、Flutter DevTools、Flutter IDE 插件等，可以提高开发效率和代码质量。

综上所述，选择使用 Flutter 作为客户端的开发框架，可以大大提高开发效率、减少开发工作量，并且可以实现高性能、高质量的应用程序。

* 1. Posgr

同步文件系统的服务端需要存储大量的文件信息和用户数据，因此需要使用一个可靠、高效的数据库管理系统。PostgreSQL是一种功能强大、开源、可扩展性好的关系型数据库管理系统，被广泛应用于各种应用场景，包括数据仓库、在线事务处理、地理信息系统等。

在同步文件系统的服务端中，使用PostgreSQL数据库管理系统有以下优势：

可靠性高：PostgreSQL的ACID事务支持和高度的数据一致性保证了数据的完整性和可靠性。

扩展性好：PostgreSQL支持高级扩展和自定义函数，可以满足不同规模和复杂度的应用需求。

开源免费：PostgreSQL是一款开源的数据库管理系统，无需额外购买商业许可证，降低了运维成本。

大量第三方工具支持：PostgreSQL有大量的第三方工具支持，包括图形化管理工具、ORM框架、数据迁移工具等，提高了开发和运维的效率。

综上所述，使用PostgreSQL作为同步文件系统服务端的数据库管理系统，可以保证系统的可靠性、可扩展性和开发效率。

* 1. Redis

Redis是一个开源的基于内存的数据结构存储系统，支持多种数据结构和丰富的命令，同时也提供了高效的发布/订阅消息机制。 使用Redis的发布/订阅功能，服务端可以将变更消息发布到指定的频道，客户端则可以订阅相应的频道以接收消息。当服务端有文件变更时，可以将变更消息发布到相应的频道，客户端通过订阅相应的频道，即可接收到实时的变更消息，从而进行相应的同步操作。

* 1. MinIO

MinIO是一个开源的对象存储服务器，用于存储和检索大量的非结构化数据，如图像、视频、文本等。它是基于Amazon S3设计的，并具有高可用性、可扩展性和兼容性，可以在多种环境中使用. 它还提供了许多安全和数据管理功能，如数据加密、访问控制、数据共享和备份等

MinIO支持分布式架构，可以通过部署多个节点来实现高可用性和可靠性，同时还支持数据冗余备份，即在多个节点之间复制数据以防止数据丢失。它针对大规模数据存储场景进行了优化，具有很高的读写性能和低延迟，可以满足高并发、高吞吐量的数据访问需求。MinIO支持简单的横向扩展，通过增加节点的数量可以扩展存储容量和性能，而无需对现有系统进行修改。

综上所述，使用MinIO作为数据存储可以提供高可用性、高性能、可扩展性、兼容性和开源免费等优势，非常适合用于同步文件系统。

2 需求分析

1. 安全的存储用户 密码信息和身份验证信息

用户的密码是他们账户的唯一凭证，如果密码信息被不当地处理或泄露，黑客或其他不良分子就可以利用这些信息来盗取用户账户、窃取用户个人信息、进行钓鱼攻击等，给用户造成严重的损失和风险。因此，保证用户密码信息的安全性非常重要，这不仅是对用户的尊重和保护，也是网站和企业保护自身安全和声誉的必要措施之一。

此外，还需要使用合适的加密算法来确保密码在传输过程中不被窃取。为了进一步提高安全性，

1. 服务端安全的存储文件
2. 适配多平台和实现多用户同步
3. 需要做到文件实时增量的同步
4. 在客户端登录以后，客户端需要对同步任务 同步文件 进行检查。

功能模块

用户登录模块

云端广播文件变更消息模块

本地监视文件变更模块

登录进度同步模块

文件和信息存储模块

功能流程

3 技术实现

3.1密码身份信息保存和校验

3.1.1 密码保存

当用户注册账号时，他们会提供一个密码用于登录。这个密码通常是用户选择的易于记忆的字符串。然而，这种字符串本身是不安全的，因为它可以被猜测和破解。如果存储在数据库中的密码是以明文形式存储，那么数据库的管理者、黑客或其他不良用户可能会轻易地访问到这些密码，这对用户的安全性和隐私构成了威胁。

为了保护用户的密码，可以使用哈希函数对密码进行加密。哈希函数是一种将任意长度的消息映射到固定长度的消息摘要（或哈希值）的算法。但是，使用哈希函数加密密码仍然存在安全问题。因为相同的密码在加密后会得到相同的哈希值，这使得攻击者可以使用预先计算的哈希表来快速破解哈希值。为了解决这个问题，可以使用盐（salt）来增加哈希函数的安全性。

盐是一个随机数，它与密码一起被哈希。通过将密码和盐组合起来进行哈希，攻击者将需要预先计算每个可能的盐值的哈希值，这样就可以防止使用预先计算的哈希表进行攻击。使用加盐哈希可以保护用户的密码，即使数据库被攻击，黑客也无法获得用户的真实密码。因此，使用加盐哈希可以有效地保护用户的密码和隐私，使用户的账号更加安全。

声明一个函数，在函数内部，首先将盐写入哈希函数中，然后将要加密的字符串写入哈希函数中。接下来，将哈希函数的输出进行十六进制编码，并将编码后的结果作为函数的返回值。

func (s \*saltService) Hashed(password, salt []byte) string {  
 defer s.Hash.Reset()  
 s.Hash.Write(salt)  
 s.Hash.Write(password)  
 return hex.EncodeToString(s.Hash.Sum(nil))  
}

将编码后生成字符串作为密码的加密结果和盐一起存入服务器数据库中，用户登陆时，读取密码加盐哈希值和盐，利用上述代码使用传输过来的密码和数据库中的盐生成哈希值，并和原来的哈希值进行比对。 //todo 增加 一致 不一致

3.1.2 身份权限校验

在用户使用软件时，服务端需要对客户端的请求进行身份权限验证，阻止非法的请求进入系统，确保用户的身份真实有效，并且只有被授权的用户才能访问系统资源。本系统使用JWT来完成身份校验。

JWT (JSON Web Token) 是一种开放标准，它定义了一种简洁、自包含的方式用于在各方之间作为 JSON 对象安全地传输信息。JWT 可以被用于认证、授权信息的传输以及声明从一个系统到另一个系统的安全性质。

JWT 由三部分组成：Header、Payload 和 Signature。Header 指定了 JWT 所使用的签名算法，通常是 HMAC SHA256 或 RSA。Payload 包含了要传输的信息，可以包含标准的注册声明（例如，iss (issuer)、exp (expiration time)、sub (subject) 等）和自定义声明。Signature 用于验证消息在传输过程中没有被篡改。

JWT 的好处之一是，它可以在不需要在服务端存储会话信息的情况下完成身份验证。在客户端完成身份验证之后，客户端将会在每次请求时将 JWT 作为 Authorization header 发送到服务端，服务端只需要对 JWT 进行签名验证即可确认请求的合法性。这种方式可以有效减轻服务端的负担，并使得服务端变得更加可扩展。

if uuid == "" {  
 return "", errors.New("uuid 不能为空")  
 }  
 timeNow := time.Now()  
 customClaims := customClaims{  
 UserID: uuid,  
 RegisteredClaims: jwt.RegisteredClaims{  
 ExpiresAt: jwt.NewNumericDate(timeNow.Add(j.ExpiresAt)),  
 IssuedAt: jwt.NewNumericDate(timeNow),  
 NotBefore: jwt.NewNumericDate(timeNow),  
 },  
 }  
 claims := jwt.NewWithClaims(jwt.SigningMethodHS256, customClaims)  
 signedString, err := claims.SignedString(j.providedKey)  
 return signedString, nil

在用户进行登陆时，使用如上的函数生成一个JWT字符串。在这个函数中，首先判断传入的用户的 ID 是否为空，如果为空则返回错误。然后生成一个包含用户 ID 和过期时间的customClaims对象，将用户的唯一ID存放入customClaims对象， 在后续解析时判断请求发出者是谁,同时限制这个生成的JWT 有效时间来防止JWT泄露后被无限制使用。使用这个customClaims 对象和指定的签名方法（HS256）创建一个 JWT 对象。接着使用提供的密钥对 JWT 进行签名，如果签名出错则返回错误，否则返回签名后的 JWT 字符串。

func (j \*jwtService) Parse(ctx context.Context, tokenStr string) (string, error) {  
 token, err := jwt.ParseWithClaims(tokenStr, &customClaims{}, func(token \*jwt.Token) (interface{}, error) {  
 return j.providedKey, nil  
 })  
 if err != nil {  
 return "", fmt.Errorf("%v", err)  
 }  
 if claims, ok := token.Claims.(\*customClaims); ok && token.Valid {  
 return claims.UserID, nil  
 }  
 return "", fmt.Errorf("%v", err)  
}

当用户登录软件执行同步任务后后，客户端会向服务端发起带JWT参数的HTTP 请求，请求在经过 服务端的Web 框架中的中间件时，回调调用上述函数，在这个函数中，传入刚刚请求内包含的 JWT 的字符串 tokenStr，通过调用 jwt.ParseWithClaims() 方法，将传入的 tokenStr 解析成一个 JWT 对象。这个 JWT 对象包含了一些元数据，如签名算法、过期时间等。在这里，使用了自定义的 customClaims 结构体。接下来，这个函数通过验证 JWT 的有效性和声明是否正确来验证用户的身份。如果 token.Valid 为 true，说明 JWT 是有效的，时间没有过期， JWT包含的信息也没有被其他人所修改，取出JWT内的用户的唯一ID并返回。如果 token.Valid 为 false，做说明JWT已经过期或者JWT数据被修改了，则函数会返回一个错误对象，其中包含了错误信息。

3.1.3 自动登录

在用户登录过一次之后，软件需要通过记录用户的账号和密码等登录凭据，在下一次登录时自动填写并完成登录操作。它的好处是可以方便用户快速登录，提升用户体验，减少用户操作次数。但同时也存在一定的安全风险，有可能会导致账号密码泄露，从而遭受不必要的损失。因此为了保护用户的密码信息，我们直接使用身份验证JWT来进行自动登录。

为了确保JWT 的安全，我们使用常见的非对称加密算法RSA对其进行加密处理，

数据保存

对象文件存储

为了满足用户的多个客户端即使是离线后再上线后也可以保持同步，我们需要将用户的文件存储至服务端。在满足数据的可用性和可靠性的同时能够水平扩展，故我们选择使用 MinIO 来完成文件存储功能。它可以使用 HTTP API 或者对应的SDK 可以很方便进行上传删除等操作文件。

文件夹关系数据库

在同步文件系统时，保存文件和文件夹的关系非常重要，因为文件系统通常是以树形结构组织文件和文件夹的。在同步文件时，如果不保存文件和文件夹的关系，则无法知道文件应该被放置在哪个文件夹中，或者在哪个文件夹中创建一个新文件夹来容纳同步的文件。

保存文件和文件夹的关系也是为了确保同步的一致性。如果文件夹被删除或移动了，与之相关联的文件也应该被删除或移动。保存文件和文件夹的关系可以帮助确保同步操作是正确的，并且保持文件系统的完整性。

另外，保存文件和文件夹的关系也有助于提高同步的效率。在同步大量文件时，如果不保存文件和文件夹的关系，则需要扫描整个文件系统以确定文件应该放置在哪个文件夹中。保存文件和文件夹的关系可以避免这种扫描，从而提高同步的效率。

将文件目录存储至数据库时要考虑到 存储效率和查询效率。最简单的存储方式就是直接将文件或者文件夹的全路径直接保存，但是这样会大大降低存储效率，每个文件夹都有相同的根路径，而这些重复的根路径应该只保存一次，让其他的文件夹对其进行引用，减少存储空间。故我们需要三张表将被同步的目录存储至数据库内，，一张文件夹表 表存储文件夹内所有的目录信息信息，另一张表存储文件夹内所有的文件信息，最后一张同步文件任务表 存储被同步文件夹的根目录，三个表分别存储所需的字段并对某些字段进行引用组成外键，比如文件夹表内不需要存储文件大小，文件夹创建时间修改时间的信息字段，将它们分开存放这样可以进一步减少存储大小，而文件表内则有一个字段代表父文件夹的ID，使用这个ID 将文件和父文件夹联系起来，以便后续查询。

主要的数据库表设计如下：

同步任务表：

为了减小数据库大小同时方便的管理每一个同步文件夹的状态，我们需要一个同步任务表，表中的每一行代表一个被创建的同步任务，其中包含任务ID 任务名 路径和状态等。

---------------------------------

表格

----------------------------------

----------------------------------

路径为被同步文件夹的根目录全路径，

状态为 : create

syncing

sync

pause

update

delete

任务ID 使用雪花ID算法来生成唯一ID，雪花ID（Snowflake ID）是一种分布式系统中生成全局唯一ID的算法，由Twitter公司开发。它的核心思想是将一个64位的整数分为多个部分，分别表示不同的信息，例如机器ID、时间戳、序列号等，以保证生成的ID具有全局唯一性和有序性。

雪花ID的生成规则如下：

第一个部分：1个bit表示符号位，0表示正数，1表示负数。

第二个部分：41个bit表示时间戳，精确到毫秒级别。由于使用的是毫秒级时间戳，因此该部分可以支持到2^41-1毫秒，大约可以支持69年。

第三个部分：10个bit表示机器ID，用于标识不同的机器。这样可以支持1024台不同的机器。

第四个部分：12个bit表示序列号，用于标识同一毫秒内生成的不同ID。由于该部分可以支持到2^12-1，因此同一毫秒内可以生成4096个不同的序列号。

与UUID 相比 雪花ID有以下优点：

生成的ID更短：UUID生成的ID为128位，而雪花ID只有64位，占用的空间更小，更加节省存储空间。

生成速度更快：UUID生成的过程中需要进行伪随机数生成和字符串格式化等操作，而雪花ID只需要进行简单的位运算和数值计算，因此生成速度更快。

有序性更好：雪花ID的生成算法中，时间戳占据了更多的位数，因此生成的ID有更好的有序性，有助于优化数据库索引和排序等操作。

易于分布式系统使用：雪花ID的算法比较简单，易于实现，并且可以很容易地在分布式系统中使用，不需要中心化的ID生成器或者数据库。

综上所述 相比于UUID，唯一ID生成算法 使用雪花ID在长度、生成速度、有序性和易用性方面都有一定的优势，同时还可以很方便在后期扩展为分布式。

文件夹表和文件表 需要存储 从根路径到下面所有文件和文件夹所形成的一个目录树，在数据库中存储这样的树形结构通常采用邻接表、路径枚举和闭包表三种方法。

1. 邻接表

邻接表是一种基于节点的表示方法，用一个节点表和一个边表分别记录树的节点和节点之间的关系。节点表包含了每个节点的信息和唯一标识符（例如节点ID），边表则记录了每个节点和其直接子节点的对应关系。这种方法可以轻松地进行遍历和查询，但是对

于树的深度较大的情况下，查询需要多次的连表操作，效率较低。

1. 路径枚举

路径枚举是一种基于路径的表示方法，使用字符串来表示每个节点的路径。具体来说，节点表中每个节点包含了唯一的标识符和该节点的路径，通过字符串的前缀和后缀来表示节点之间的关系。这种方法的查询效率较高，但是在插入和删除节点时需要修改路径信息，比较繁琐。

1. 闭包表

闭包表是一种基于矩阵的表示方法，使用一个二维矩阵来表示树中节点之间的关系。矩阵中的每个元素代表了节点之间的连通性，使用0和1来表示。通过闭包算法可以计算出任意两个节点之间的路径，查询效率较高，但是需要较大的存储空间，并且在插入和删除节点时需要进行矩阵的重新计算。

这三种方法在实际应用中各有优劣势，其中闭包表通常需要占用更多的存储空间，因为每个文件或文件夹节点对应的闭包集合需要存储在表中。当新增文件或者文件夹时，需要更新闭包表中的数据。如果新增文件或者文件夹过深，更新操作需要占用更大的计算资源，故我们排除掉闭包表。

由于文件存储于父文件夹之下，我们想寻找它的位置，我们只需要查询其父文件夹的ID 即可，且它只需要存储文件名，不需要存储路径信息，故采用邻接表作为文件表的存储结构最为合适，同时再增加一个Level 字段代表这个文件的层级来加快查询速度。

文件表

---------------------------------

表格

----------------------------------

----------------------------------

文件夹ID 用户ID 使用雪花ID算法进行生成，在文件的创建和删除时间字段使用时间戳作为存储类型。 时间戳类型通常只需要4个字节，而日期类型通常需要8个字节，因此使用时间戳类型可以在存储空间上节省一半的空间。由于时间戳类型表示的是一个时间戳数字，因此可以方便地进行时间比较和计算。在后续进行文件更新对比时 可以计算时间戳之差来判断文件是否被更新。

而文件夹在存储时若使用邻接表则需要考虑文件夹名和层级重复问题。在查询获得重复文件夹的情况时，虽然可以向上查询父路径来确定唯一，但是如果遇到父路径也是重复的，那就只能继续向上查询，直到查询到父路径不再重复为止，这样会导致降低查询效率。而使用路径枚举则不会出现此情况，它将文件夹和跟目录的相对路径直接保存起来，而相对路径在一个目录下是唯一存在不会重复的。故存储文件夹采用路径枚举最为合适。

文件夹表

---------------------------------

表格

----------------------------------

----------------------------------

文件夹ID 用户ID 和同步任务ID 都是使用雪花ID进行生成

同步文件实现

1. 客户端创建同步任务

task := ent.SyncTask{  
 Name: st.Name,  
 RootDir: st.Path,  
}  
  
var gs []ent.SyncTask  
s.DB.Find(&gs)  
for \_, st := range gs {  
 if prefix := strings.HasPrefix(task.RootDir, st.RootDir); prefix  
 if prefix := strings.HasPrefix(st.RootDir, task.RootDir);prefix }

添加同步任务的名字和同步文件夹的根路径，同时检测所添加的路径是否包含在已被同步的文件夹内和是否为已被同步文件夹的父路径。例如：如果有已经同步的目录路径为 ‘/root/dir1/folder1’ , 则不能添加新的同步根路径 `/root/dir1` 和 `/root/dir1/folder1/test1`，防止新增父路径和子路径的任务，避免文件被重复同步。

全量同步

filepath.WalkDir(rooPath, func(path string, d fs.DirEntry, err error) error {  
  
 absPath := path[rootPathLen:]  
 if d.IsDir() {  
 dir := ent.Dir{  
 SyncID: syncID,  
 Dir: absPath,  
 Level: len(strings.Split(absPath, "/")),  
 }  
 h.DB.Create(&dir)  
   
 file := ent.File{  
 SyncID: syncID,  
 Name: d.Name(),  
 ParentDirID: dir.ID,  
 Level: level,  
 Deleted: false,  
 CreateTime: time.Now().Unix(),  
 ModTime: info.ModTime().Unix(),  
 }  
  
 fileIO, err := os.Open(path)  
 if err := h.HttpClient.FileCreate(&file, fileIO); err != nil {  
 return err  
 }  
  
 h.DB.Create(&file)  
 return err  
})

创建完成后，需要初始化这个同步任务，使用 filepath.WalkDir 函数循环遍历目录，将根目录下所有的文件夹和文件的名字查询出来， 将路径和元数据等信息存储至本地数据库，同时也上传到服务器数据库内。如果上传的是文件夹，则只需要根据库上述的数据库文件夹表设计存储相应的数据，如果是文件则另需要将文件内的数据读取并发送至存储值服务端的文件对象存储系统进行保存。

1. 监视文件变更

全量同步完成后需要监视器来实时获取文件变更数据，而一个监视器不足以满足监视多个同步文件夹的需求，同时要满足多线程的要求，防止多个监视器相互阻塞 降低程序性能。

1. 过滤

如果在同步文件夹下存在大量的不需要同步的文件 目录或者临时生成的文件，那么同步时过滤掉这些文件或目录可以减少同步所需的时间和系统资源消耗，减少网络传输量，提高同步效率。有些用户在同步文件夹下可能存有敏感数据的文件或目录，将它们过滤掉以确保数据的安全性。

func newRegexpRule(expr string) (Rule, error) {  
 reg, err := regexp.Compile(expr)  
 return &regexpRule{  
 expr: expr,  
 reg: reg,  
 }, nil  
}

过滤文件时使用正则表达式匹配其文件名符不符合要求，使用 Golang 的regexp.Compile（）函数，)的作用是编译一个正则表达式，将字符串形式的正则表达式转换为一个可以匹配文本的正则表达式对象，如果正则表达式不合法，会返回一个错误。编译后的正则表达式对象可以用于匹配文本、查找匹配、替换匹配等操作。

func newFilePathRule(expr string) (Rule, error) {  
 err := filepath.Match(expr, "")  
 return &filePathRule{  
 expr: expr,  
 }, nil  
}

进行路径过滤时，可以使用 filepath.Match（）函数，它可以判断指定字符串是否与文件路径模式匹配。它接受两个参数：一个是文件路径模式（例如"/home/user/.txt"），另一个是待匹配的字符串。如果待匹配的字符串符合指定的文件路径模式，该函数会返回true；否则返回false。

type Rule interface {  
 Match(s string) bool  
 Expression() string  
}

func NewIgnore(ignoreConfig \*types.Ignore) (\*Ignore, error) {  
 rules, err := parseIgnoreConfig(ignoreConfig)  
 return &Ignore{rules}, err  
}  
  
func (ig \*Ignore) Match(s string) bool {  
 for \_, rule := range ig.rules {  
 if rule.Match(s) {  
 return true  
 }  
 }  
 return false  
}

func parseIgnoreConfig(ignoreConfig \*types.Ignore) ([]Rule, error) {  
 var rs []Rule  
 for \_, ig := range ignoreConfig.Filepath {  
 rule, err := newFilePathRule(ig)  
 rs = append(rs, rule)  
 }  
 for \_, ig := range ignoreConfig.Regexp {  
 rule, err := newRegexpRule(ig)  
 rs = append(rs, rule)  
 }  
 return rs, nil  
}

Rule是一个接口，定义了Match和Expression方法，分别用于判断路径是否匹配某个规则和返回规则表达式。通过调用NewIgnore函数从配置文件获得的参数来创建一个新的忽略规则对象。忽略规则对象包含了多个规则。使用parseIgnoreConfig函数分别解析正则和路径匹配解析返回多个规则并初始化这个忽略规则对象，当需要对某些文件或者路径进行判断的时候，使用 忽略规则对象的Match 方法，即可被所有规则进行判定，如果能匹配上则返回true，反之则返回 false。

1. 恢复

多客户端同步

1. 客户端唯一ID
2. Redis 订阅和发布
3. WebSocket 双向信息传输
4. 本地客户端处变更

相互传输接受处理信息实现

本地文件数据变动检测实现

客户端启动 同步任务进度检查