GIT

Materia: Taller de Programación

Universidad: Universidad de Buenos Aires

Proyecto: Git Rústico

Cuatrimestre: 2do Cuatrimestre 2023

Grupo: Messi

Integrantes:

- Clara Ruano Frugoli
- Francisco Martinez
- Ramiro Gestoso
- Maria Paula Bruck

Fecha: 4 de Diciembre 2023

Introduccion:

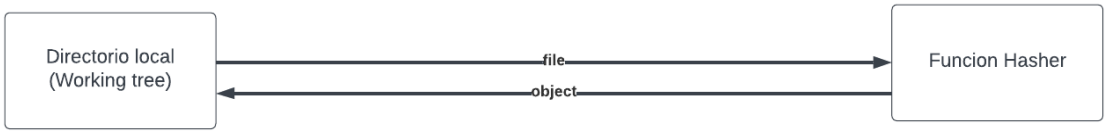
El sistema de control de versiones Git se ha convertido en una herramienta fundamental para el desarrollo colaborativo de software. Su diseño distribuido, eficiencia y flexibilidad lo han posicionado como uno de los sistemas más populares en la industria. En este trabajo práctico, nos sumergimos en el fascinante mundo de Git, explorando su funcionamiento interno y construyendo nuestra propia implementación simplificada.

Comandos Implementados:

Cada uno de los comandos se encuentra en un archivo con el nombre correspondiente al mismo dentro de la carpeta /src .

hash-object

En nuestra implementación el comando la función `store_file` es la que inicia su ejecución al recibir la ruta del archivo (`path`) que se desea almacenar y la ruta al directorio del repositorio Git (`git_dir_path`). En primer lugar, calcula el hash SHA-1 del contenido del archivo mediante la función `hash_file_content`, garantizando la unicidad del archivo en el repositorio. A continuación, construye la estructura de directorios en la carpeta de objetos de Git basándose en los primeros dos caracteres del hash, proporcionando una organización eficiente y distribuida de los objetos. La función procede a comprimir el contenido del archivo utilizando el formato zlib, añadiendo el encabezado correspondiente con el tipo de objeto ("blob"). Este enfoque no solo reduce el espacio ocupado por los archivos, sino que también preserva la integridad de los datos. Finalmente, la función retorna el hash del contenido, sirviendo como identificador único del archivo en el repositorio Git.



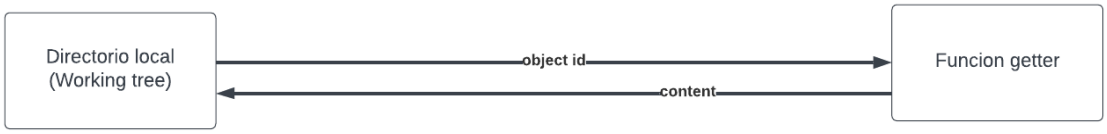
cat-file

Nuestra implementación se compone de dos funciones principales: `cat_file` y `cat_file_return_content`.

La función `cat_file` recibe el hash del archivo, la ruta al directorio del repositorio Git y el destino de salida para el contenido del archivo. En caso de éxito, utiliza la función `cat_file_return_content` para obtener el contenido del archivo y lo escribe en el destino proporcionado. Si el hash no es válido o el archivo no se encuentra, la función retorna un error indicando que el archivo no pudo ser encontrado.

Por otro lado, la función `cat_file_return_content` realiza la tarea principal de obtener el contenido del archivo a partir de su hash. Comienza construyendo la ruta del archivo en la estructura de directorios del repositorio Git y abre el archivo correspondiente. Luego, utiliza la función `decompress_file` para descomprimir el contenido del archivo, que puede estar comprimido en formato zlib. Después de la descompresión, la función procesa el contenido para extraer y retornar el texto del archivo.

La descompresión del contenido zlib se realiza mediante la función `decompress_file`

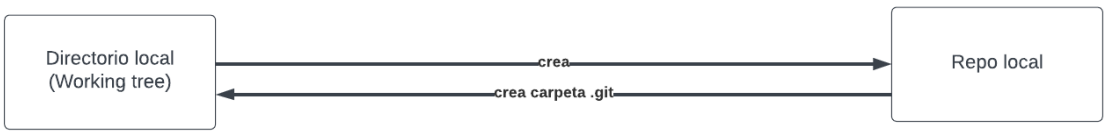


init

Nuestra implementacion del comando se base en la función `git_init` toma varios parámetros, incluyendo la ruta al directorio donde se inicializará el repositorio, el nombre de la rama inicial y, opcionalmente, la ruta a un directorio de plantillas para copiar archivos. La implementación comienza creando directorios necesarios, como el directorio `.mgit`, `objects`, y `refs/heads`. Luego, crea archivos esenciales como `HEAD`, `config`, y `index` que son fundamentales para el funcionamiento de Git.

La función utiliza las utilidades `create_directory_if_not_exists` y `create_file_if_not_exists` para manejar la creación de directorios y archivos de manera segura, evitando duplicados y asegurándose de que los directorios existan antes de intentar crear archivos en ellos. Además, permite la creación de archivos con contenido específico.

La función también puede copiar archivos desde un directorio de plantillas si se proporciona uno. Esto proporciona flexibilidad al permitir la configuración inicial del repositorio con archivos específicos según las necesidades del usuario.

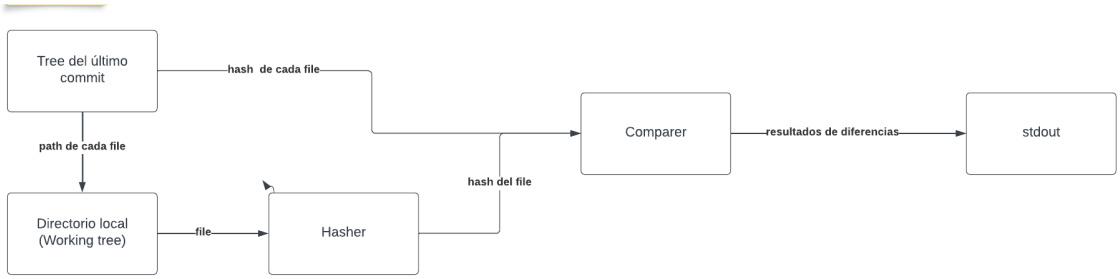


status

Para status creamos las funciones `changes_to_be_committed`, `get_staged_changes`, `find_unstaged_changes`, `get_unstaged_changes` y `find_untracked_files`. Estas funciones verifican cambios en el directorio de trabajo, en el índice y en archivos no rastreados.

1. `changes_to_be_committed`: Compara el hash de archivos en el índice con su contenido actual para identificar archivos modificados que aún no se han preparado para el compromiso. Escribe información sobre estos cambios en la salida proporcionada.

2. `get_staged_changes`: Devuelve una cadena que contiene todos los cambios preparados en el índice del repositorio Git.
3. `find_unstaged_changes`: Compara el hash de archivos en el índice con su contenido actual para identificar archivos modificados que aún no se han preparado para el compromiso. Escribe información sobre estos cambios en la salida proporcionada.
4. `get_unstaged_changes`: Devuelve una cadena que contiene todos los cambios no preparados en el índice del repositorio Git.
5. `find_untracked_files`: Encuentra de manera recursiva y escribe información sobre archivos no rastreados en un repositorio Git. Recorre la estructura de directorios, la compara con los archivos rastreados en el índice de Git e identifica los archivos no rastreados. Escribe información sobre estos archivos no rastreados en la salida proporcionada.



add

Para la implementation de add implementamos las siguientes funciones:

1. `process_file_name`: Esta función procesa un nombre de archivo o directorio y actualiza el índice en consecuencia. Si la ruta es un directorio, agrega todos los archivos dentro de él al índice. Si es un archivo, agrega ese archivo al índice.
2. `add`: Esta es la función principal que implementa el comando `git add`. Añade archivos al índice de Git basándose en la ruta proporcionada. Si la ruta apunta a un directorio, se agregarán todos los archivos dentro de ese directorio. Si algún archivo no existe en el directorio de trabajo, se eliminará del índice. Si el archivo no existe en el índice ni en el directorio de trabajo, se devuelve un error.

   La función toma varios argumentos:

   - `path`: La ruta del archivo o directorio que se va a agregar.

- `index_path`: La ruta al archivo que representa el índice de Git.
- `git_dir_path`: La ruta al directorio de Git.
- `gitignore_path`: La ruta al archivo `.gitignore`.
- `options`: Opciones adicionales. En este caso, se utiliza para procesar la opción "." (agregar todos los archivos en el directorio actual).

La función verifica si la ruta es un subdirectorio de `.mgit` y, en ese caso, no hace nada. Luego, procesa los archivos según las opciones proporcionadas.

- Si se proporciona la opción ".", agrega todos los archivos en el directorio de trabajo al índice.
- Si no se proporcionan opciones o las opciones no son ".", agrega el archivo o directorio especificado al índice.

Finalmente, escribe el índice actualizado en el archivo.

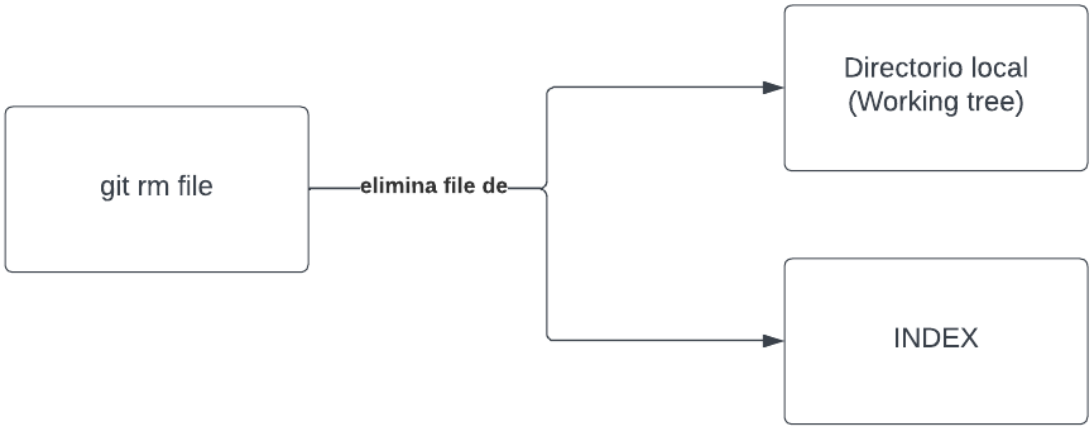
rm

La función `git_rm` es la funcion principal de este comando. Al recibir el nombre del archivo a eliminar, la ruta del archivo de índice, la del directorio Git, y la del archivo `.gitignore`, la función procede a cargar el índice desde el archivo especificado. Luego, verifica si el archivo está presente en el índice y, en caso afirmativo, lo elimina tanto del índice como del directorio de trabajo. La función gestiona posibles errores, como la carga fallida del índice o la ausencia del archivo en el índice, y muestra mensajes detallados en caso de fallos. Finalmente, guarda el índice actualizado.

La función `remove_directory` se encarga de eliminar de forma recursiva un directorio y todos sus contenidos. Recorre los elementos del directorio y elimina archivos y directorios de manera recursiva. Esta función también gestiona posibles errores durante el proceso, asegurando una eliminación segura.

La función `remove_path` es responsable de eliminar un archivo o directorio del índice y del directorio de trabajo. Utiliza la función `remove_directory` si el path especificado es un

directorio, asegurando una eliminación completa y segura. Además, elimina el archivo del índice y del directorio de trabajo, manteniendo la coherencia entre ambos.



commit

En este comando se implementaron distintas funciones para poder realizar el comportamiento esperado.

Función `create_new_commit_file`

Esta función crea un nuevo archivo de commit con el árbol, el commit padre, el mensaje y otros metadatos necesarios. Si no se han realizado cambios desde el último commit, la función devuelve un error. También verifica la existencia del archivo de índice y del archivo `.gitignore`.

Función `get_branch_name`

Recupera el nombre de la rama actualmente activa en el repositorio Git. Lee el contenido del archivo "HEAD" para determinar la rama activa y devuelve su nombre.

Función `new_commit`

Crea un nuevo commit y actualiza el archivo de referencia de la rama correspondiente. Si la rama no existe, la crea. Utiliza la función `create_new_commit_file` para generar el contenido del commit.

Función `new_merge_commit`

Similar a `new_commit`, pero específica para commits de fusión que tienen dos padres. Crea un commit con dos padres y actualiza la referencia de la rama.

Función `get_parent_hash`

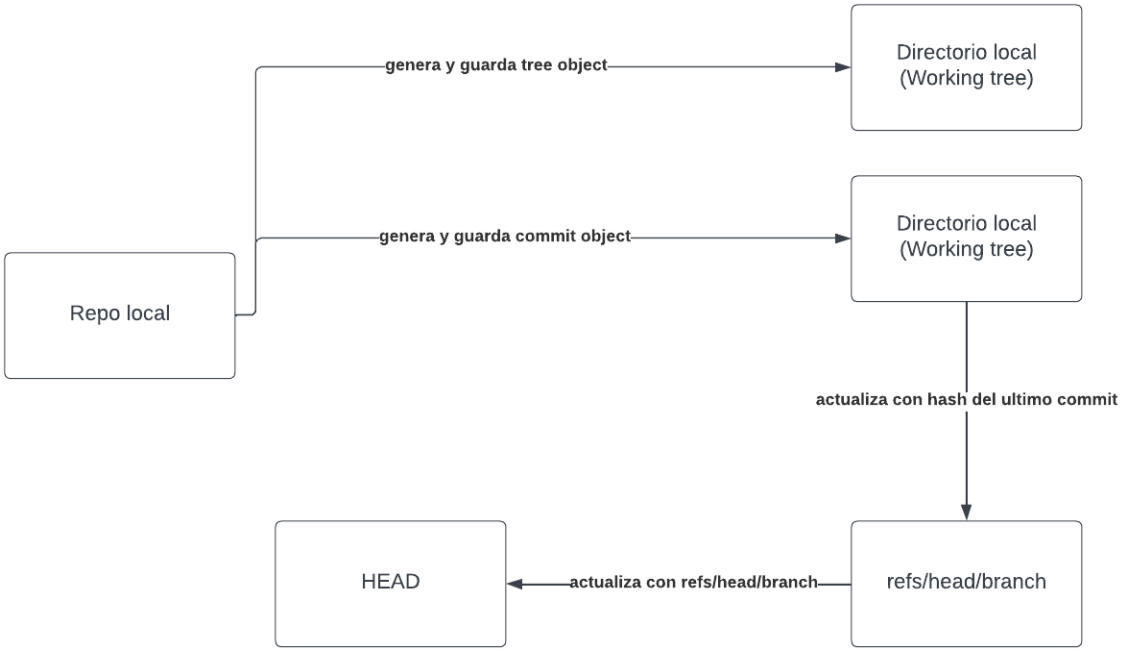
Recupera el hash del commit padre de un commit dado.

Función `get_commit_message`

Recupera el mensaje asociado a un commit dado.

Función `read_head_commit_hash`

Lee y devuelve el hash del commit al que apunta la referencia "HEAD" en el repositorio Git. Resuelve la referencia y devuelve el hash correspondiente.

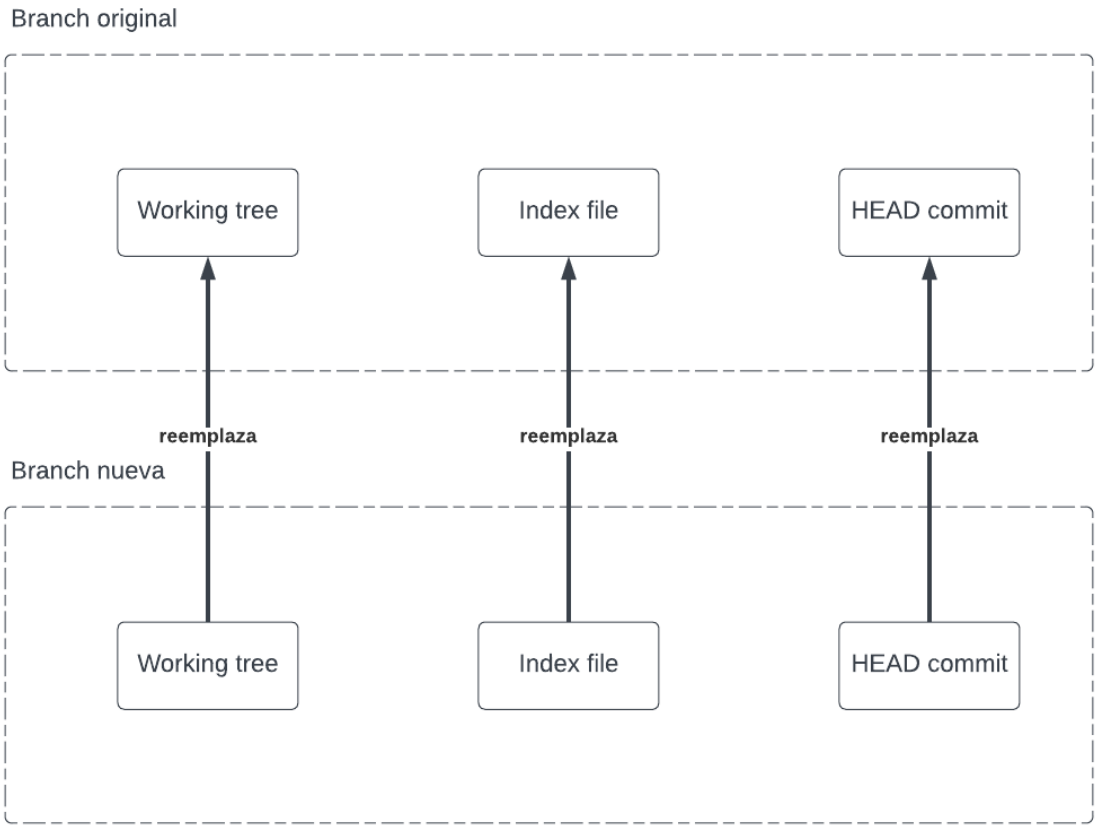


checkout

1. **Función** `log_checkout`**:**
    - Registra detalles del comando 'git checkout' en un archivo llamado 'logger_commands.txt'.
    - Toma la rama actual, la nueva rama, una cadena opcional para opciones adicionales y la ruta del directorio Git como parámetros.

2. **Función `checkout_branch`:**
    - Cambia a una rama específica actualizando la referencia HEAD en un repositorio similar a Git.
    - Maneja errores durante el proceso y registra el comando 'git checkout'.
3. **Función `checkout_branch_references`:**
    - Cambia a una rama diferente en el repositorio Git local actualizando las referencias.
    - Devuelve el identificador de commit de la rama anterior en caso de éxito.
4. **Función `create_and_checkout_branch`:**
    - Crea y cambia a una nueva rama en un repositorio similar a Git.
    - Registra el comando 'git checkout'.
5. **Función `create_and_checkout_branch_references`:**
    - Crea y cambia a una nueva rama Git en el repositorio local.
    - Devuelve el identificador de commit de la rama anterior en caso de éxito.
6. **Función `create_or_reset_branch`:**
    - Crea o reinicia una rama similar a Git en un repositorio.
    - Verifica si la rama existe y, si es así, la reinicia usando `create_and_checkout_branch`.
7. **Función `checkout_commit_detached`:**
    - Cambia a un commit específico en modo desvinculado en un repositorio similar a Git.
    - Actualiza la referencia HEAD y reemplaza el árbol de trabajo con el contenido del nuevo commit.
8. **Función `checkout_commit_detached_references`:**
    - Cambia a un commit específico en modo HEAD desvinculado en el repositorio Git local.
    - Devuelve el identificador de commit de HEAD anterior en caso de éxito.

9. **Función `replace_working_tree`:**
    - Reemplaza el árbol de trabajo de un repositorio Git con el contenido de un nuevo commit.
    - Elimina archivos y directorios del commit anterior y crea nuevos a partir del nuevo commit.
10. **Función `force_checkout`:**
    - Cambia forzosamente a una rama o commit específico en un repositorio similar a Git.
    - Maneja ramas e identificadores de commit, actualiza el archivo HEAD y registra el comando 'git checkout'.

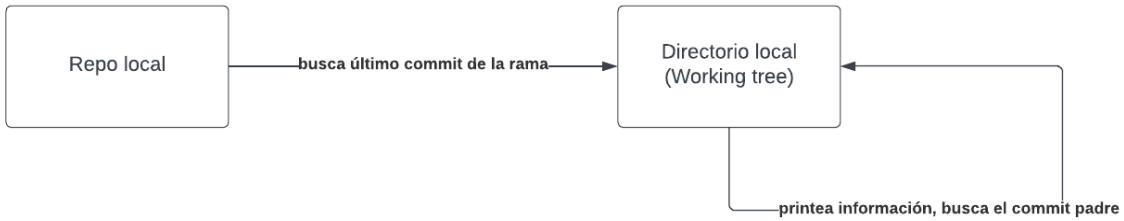


log

La implementación se basa en dos estructuras principales: `LogIter` y `Log`. `LogIter` se utiliza para facilitar la iteración a través de los registros de commit, mientras que `Log` almacena información relevante sobre cada commit.

El método `load` en la estructura `Log` permite cargar información de un commit específico si se proporciona su hash, o cargar el commit actual desde la referencia HEAD del repositorio. La lectura del encabezado del commit se realiza mediante el método `load_from_hash`, que a su vez utiliza la función `cat_file::cat_file_return_content` para obtener el contenido del commit.

La estructura `Log` incluye métodos para analizar líneas del encabezado del commit y actualizar sus campos correspondientes. Además, se proporciona la capacidad de configurar el modo en línea para la formación de mensajes de commit.

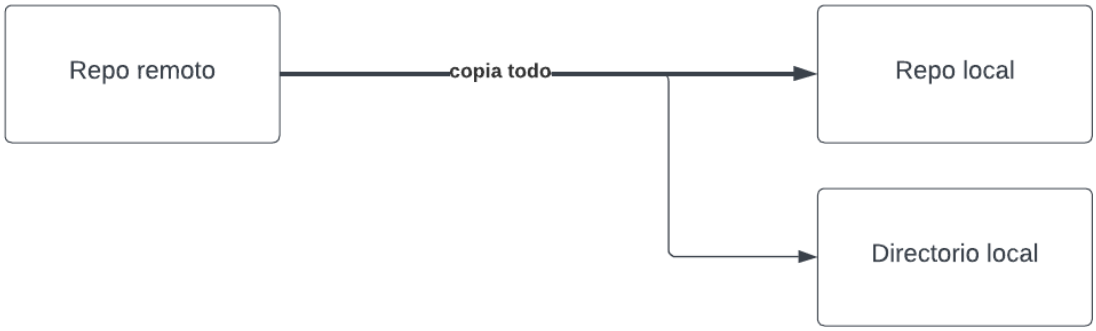


clone

La implementación se basa en varias funciones y estructuras auxiliares. Algunas de las funciones clave incluyen:

1. `get_default_branch_commit`: Esta función recupera el hash del commit de la rama predeterminada del repositorio local, obteniendo la información de la referencia "origin/master".
2. `get_clean_refs`: Extrae el último componente de cada referencia de una lista de referencias. Esto es útil para trabajar con nombres limpios y simplificados.
3. `create_working_dir`: Crea un directorio de trabajo basado en el commit de la rama predeterminada de un repositorio Git local. Esto implica la inicialización de la rama maestra, la carga del árbol de commits y la creación de archivos esenciales.
4. `create_remote_dir`: Crea el directorio "refs/remotes/origin/" necesario para almacenar las referencias remotas.

5. `git_clone`: La función principal que realiza la clonación del repositorio remoto. Inicializa un nuevo repositorio local, obtiene las referencias del servidor remoto, crea directorios y archivos esenciales, y configura las referencias locales y remotas.



fetch

La implementación se basa en varias funciones y estructuras auxiliares. Aquí se destacan algunas de las funciones clave:

1. `FetchEntry`: Representa una entrada en el archivo "FETCH_HEAD" generado durante las operaciones de `git fetch`. Contiene información sobre un commit recuperado, incluido su hash, el nombre de la rama y la URL del repositorio remoto.
2. `FetchHead`: Representa el archivo "FETCH_HEAD" que contiene una lista de entradas recuperadas durante las operaciones de `git fetch`. Mantiene una lista de instancias de `FetchEntry`.
3. `get_clean_refs`: Limpia una lista de referencias Git extrayendo sus últimos componentes. Esto es útil para trabajar con nombres limpios y simplificados.
4. `git_fetch`: Realiza la operación `git fetch` para actualizar el repositorio local con los cambios remotos. Recupera la información de commits más recientes de cada rama, trae objetos faltantes y almacena la información en el archivo "FETCH_HEAD".
5. `git_fetch_for_gui`: Versión de `git_fetch` diseñada para ser utilizada en una interfaz gráfica de usuario (GUI). Retorna una lista de strings que representan las referencias limpias.

merge

La implementación presentada aborda tanto los escenarios de "fast-forward" como de fusión bidireccional ("two-way merge").

Funciones Principales

1. `is_fast_forward`

Esta función determina si una fusión puede realizarse mediante un "fast-forward." En este caso, si el commit de la rama actual (`our_branch_commit`) es el mismo que el commit del ancestro común (`common_commit`), se considera un "fast-forward." La función devuelve `true` en este caso y `false` de lo contrario.

2. `find_common_ancestor`

La función `find_common_ancestor` busca el ancestro común entre dos commits (`our_branch_commit` y `their_branch_commit`). Utiliza la información del historial de commits para encontrar el primer ancestro común. Si no se encuentra un ancestro común, la función devuelve un error.

3. `fast_forward_merge`

Esta función realiza una fusión "fast-forward" de la rama actual (`our_branch`) a la rama remota (`their_branch`). Actualiza el árbol de trabajo y el índice para reflejar el nuevo estado fusionado. También actualiza la referencia de la rama actual al commit de la rama remota.

4. `two_way_merge`

La función `two_way_merge` realiza una fusión bidireccional entre dos ramas (`our_branch` y `their_branch`). Utiliza los árboles de commits correspondientes para fusionar los cambios. En caso de conflictos, la función devuelve las rutas conflictivas. Actualiza el árbol de trabajo, el índice y la referencia de la rama actual al nuevo commit fusionado.

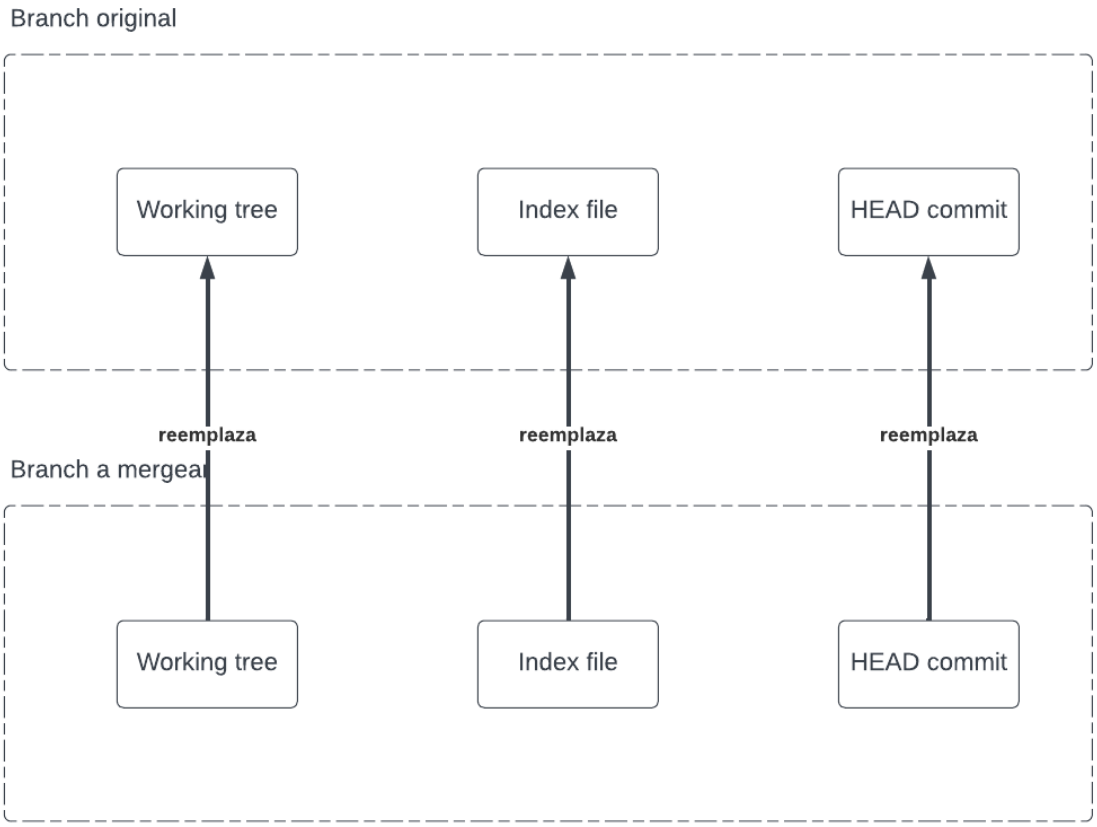
5. `git_merge`

Esta función es la interfaz principal para realizar fusiones. Primero, determina si se puede realizar un "fast-forward" utilizando `is_fast_forward` y `find_common_ancestor`. Si es posible, realiza una fusión "fast-forward"; de lo contrario, realiza una fusión bidireccional utilizando `two_way_merge`. Devuelve el nuevo commit resultante y las rutas conflictivas, si las hay.

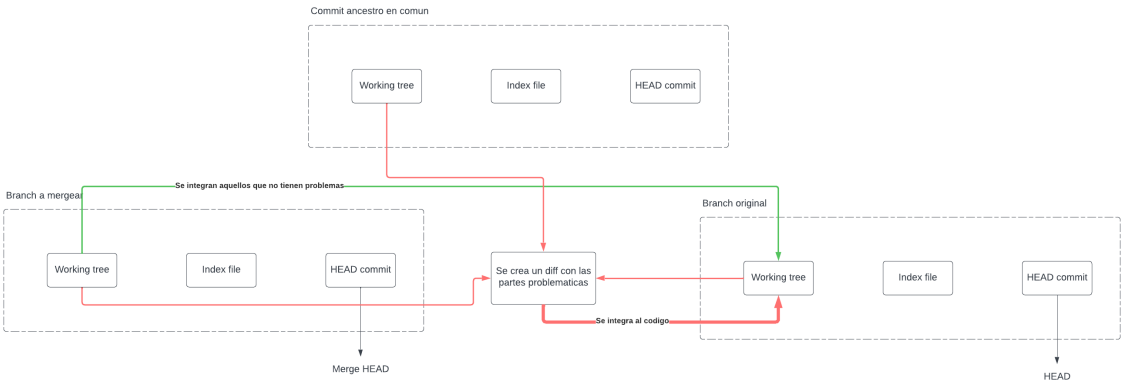
6. `merge_remote_branch`

La función `merge_remote_branch` fusiona cambios de una rama remota en la rama local actual. Utiliza los árboles de commits correspondientes y actualiza el índice y el árbol de trabajo.

Fast foward



True merge



remote

Funciones Principales

1. `git_remote`

Esta función centraliza el manejo de subcomandos para `git remote`. Recibe un conjunto de argumentos desde la línea de comandos, como el nombre del subcomando y sus argumentos asociados. Luego, dirige la ejecución a la función específica correspondiente al subcomando proporcionado.

2. Funciones de Manejo de Subcomandos

Las siguientes funciones manejan subcomandos específicos de `git remote`:

- `handle_list_command`: Lista los remotos configurados en el repositorio.
- `handle_add_command`: Agrega un nuevo remoto con el nombre y la URL proporcionados.
- `handle_remove_command`: Elimina un remoto según el nombre especificado.
- `handle_set_url_command`: Cambia la URL de un remoto existente.
- `handle_get_url_command`: Obtiene y escribe la URL de un remoto existente.
- `handle_rename_command`: Cambia el nombre de un remoto existente.

Cada una de estas funciones realiza validaciones de argumentos y utiliza la estructura de configuración de Git (`Config`) para realizar las operaciones correspondientes.

3. Función `report_error`

Esta función proporciona un mecanismo consistente para informar errores durante la ejecución de `git remote`. Escribe un mensaje de error en la salida especificada y devuelve un error de tipo `io::Error` con el mismo mensaje.

pull

Esta operación consta de dos pasos principales: fetch y merge. La función utiliza módulos previamente implementados para llevar a cabo estas operaciones.

Función Principal

1. `git_pull`

La función `git_pull` es la pieza central que ejecuta la operación de pull. A continuación, se describen los parámetros y la funcionalidad clave:

- **Parámetros:**
  - `branch`: El nombre de la rama local que se actualizará.
  - `local_dir`: La ruta al directorio local que contiene el repositorio Git.
  - `remote_repo_name`: (Opcional) El nombre del repositorio remoto desde el cual realizar el pull. Si no se proporciona, se utiliza "origin".
  - `host`: El host asociado con el repositorio remoto.
- **Operaciones:**
  1. Se ejecuta la operación de fetch llamando a la función `git_fetch` del módulo `fetch`, pasando los parámetros relevantes.
  2. Se construye la ruta al directorio `.mgit` dentro del repositorio Git local.
  3. Se carga el archivo `FETCH_HEAD` desde el directorio `.mgit` utilizando la estructura `FetchHead`.

4. Se obtienen las entradas correspondientes a la rama local desde `FETCH_HEAD`.
5. Se actualizan los archivos de las ramas locales con los nuevos valores obtenidos durante el fetch.
6. Se obtiene el hash del commit remoto desde la entrada correspondiente a la rama local.
7. Se ejecuta la operación de merge utilizando la función `merge_remote_branch` del módulo `merge`.

- **Resultados:**
  - La función devuelve un `Result` indicando el éxito o el fallo de la operación de pull.

push

La función utiliza módulos y estructuras de datos previamente implementados para llevar a cabo esta operación.

Función Principal

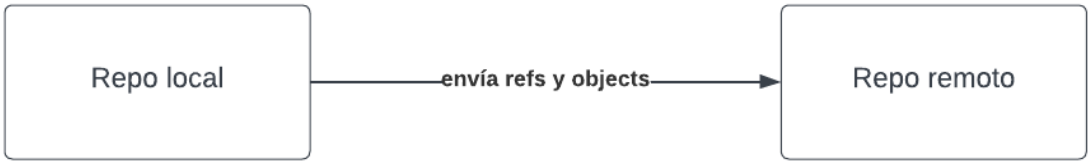
1. `git_push`

La función `git_push` es la pieza central que ejecuta la operación de push. A continuación, se describen los parámetros y la funcionalidad clave:

- **Parámetros:**
  - `branch`: El nombre de la rama que se va a enviar al repositorio remoto.
  - `git_dir`: La ruta al directorio Git.
- **Operaciones:**
  1. Se carga el archivo de configuración Git utilizando la estructura `Config`.

2. Se obtiene la URL remota del repositorio.
3. Se divide la URL remota para obtener la dirección y el nombre del repositorio.
4. Se crea una instancia del cliente Git utilizando la dirección y el nombre del repositorio.
5. Se llama al método `receive_pack` del cliente para enviar la rama al repositorio remoto.

- **Resultados:**
  - La función devuelve un `Result` indicando el éxito o el fallo de la operación de push.



branch

La función `git_branch` realiza varias operaciones relacionadas con las ramas Git, como listar las ramas existentes, crear nuevas ramas, eliminar ramas y modificar el nombre de las ramas. La implementación utiliza módulos y estructuras de datos previamente definidos para lograr estas operaciones.

Funciones Principales

1. `git_branch`

La función `git_branch` es la función principal que realiza diversas operaciones relacionadas con las ramas Git. A continuación, se describen las operaciones principales:

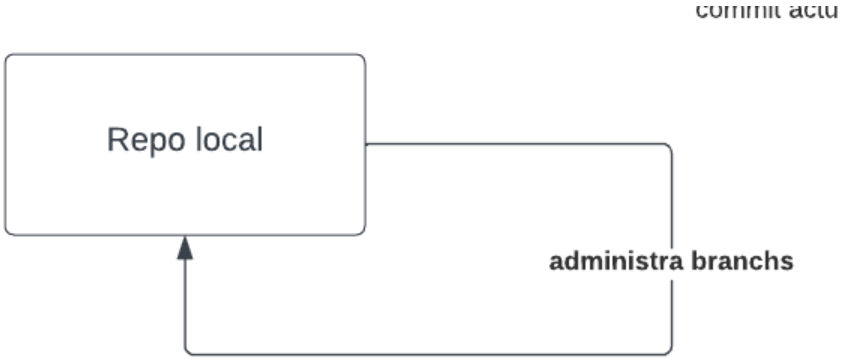
- **Parámetros:**
  - `name`: El nombre de la nueva rama (si se está creando una nueva rama).
  - `option`: La opción que especifica la operación a realizar (`l` para listar, `d` para eliminar, `m` para modificar, `c` para crear).

- `new_name`: El nuevo nombre para la rama (en caso de operación de modificación).
- `output`: Una referencia mutable a un tipo que implementa el trait `Write` para escribir mensajes de salida.

- **Operaciones:**
  - Si se proporciona un nombre, la función determina la operación a realizar según la opción especificada.
    - `l`: Lista todas las ramas en el repositorio.
    - `d`: Elimina la rama especificada.
    - `m`: Modifica el nombre de una rama existente.
    - `c`: Crea una nueva rama.
  - Si se proporciona un nuevo nombre, la función realiza la operación correspondiente (modificación de nombre).
  - Se utilizan funciones auxiliares para realizar las operaciones específicas, como `list_branches`, `delete_branch`, `modify_branch`, y `create_new_branch`.
- **Resultados:**
  - La función devuelve un `Result` indicando el éxito o el fallo de la operación.

2. Otras Funciones

Se implementan varias funciones auxiliares que realizan operaciones específicas relacionadas con las ramas Git, como obtener el camino de la rama actual, obtener el hash de confirmación de una rama, actualizar el hash de confirmación de una rama, obtener el hash de confirmación de la rama actual, eliminar una rama, y otras.



check-ignore

La función `git_check_ignore` permite verificar si se deben ignorar ciertos caminos basándose en un ignorador proporcionado. La implementación utiliza una estructura modular y maneja varios casos de entrada, proporcionando mensajes informativos y manejo de errores.

Función Principal

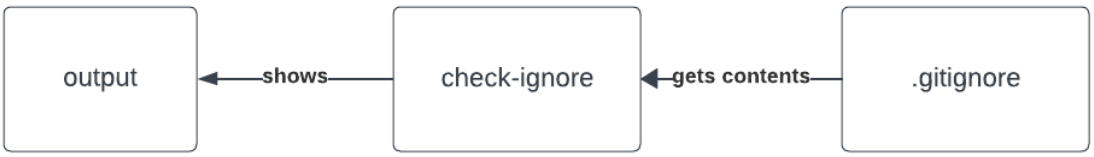
1. `git_check_ignore`

La función `git_check_ignore` realiza la verificación de ignorar caminos basándose en un ignorador proporcionado. A continuación, se describen los aspectos principales de la implementación:

- **Parámetros:**
  - `ignorer_name`: El nombre del ignorador, que se utilizará en la salida en caso de coincidencia.
  - `git_ignore_path`: La ruta al archivo `.gitignore` que contiene las reglas de ignorado.
  - `line`: Un vector de cadenas que representa la línea de entrada, generalmente obtenida de los argumentos de la línea de comandos.
  - `output`: Una referencia mutable a un tipo que implementa el trait `Write` para la salida de resultados.
- **Operaciones:**
  - Se carga un ignorador usando la función `Ignorer::load` proporcionada por la implementación del trait `Ignorer`.

- Se manejan varios casos según la longitud y contenido de la línea de entrada.
  - Si la longitud es 2 o 3 (y la tercera cadena es "-v"), se informa un error indicando que no se especificó ningún camino.
  - Si la tercera cadena es "-v", se imprime la información detallada de coincidencia con el ignorador, incluyendo número de línea y camino.
  - Si la tercera cadena comienza con "-", se informa un error indicando una opción no válida.
  - En otros casos, se verifica si cada camino está siendo ignorado por el ignorador y se imprime el resultado correspondiente.

- **Resultados:**
  - La función devuelve un `Result` indicando el éxito o el fallo de la operación. Si hay un error, se imprime un mensaje de error informativo.



ls-files

La función `git_ls_files` permite listar archivos en un directorio de trabajo, proporcionando opciones adicionales para mostrar archivos no rastreados (`-o`) o archivos modificados (`-m`).

Funciones Principales

1. `list_files_in_index`

La función `list_files_in_index` lista los archivos presentes en el índice proporcionado. Los archivos se imprimen en la salida especificada.

2. `process_untracked`

La función `process_untracked` procesa elementos no rastreados, manejando tanto directorios como archivos. Para directorios, lista recursivamente los archivos utilizando la función `git_ls_files`. Para archivos, escribe la ruta de entrada relativa en la salida.

3. `list_untracked_files`

La función `list_untracked_files` lista recursivamente archivos no rastreados en un directorio especificado, comparando con el índice proporcionado.

4. `list_modified_files`

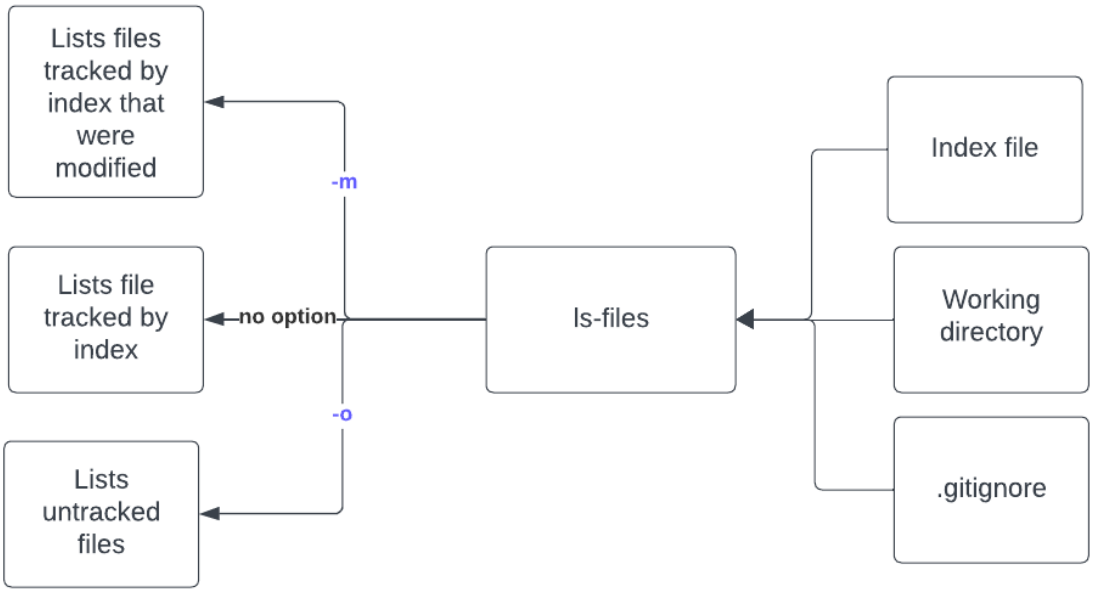
La función `list_modified_files` lista los archivos modificados comparando los hash almacenados en el índice con los hash calculados de los archivos actuales.

5. `git_ls_files`

La función `git_ls_files` realiza la lista de archivos basándose en la entrada de línea de comandos proporcionada. Puede listar archivos en el índice (`-c`), archivos no rastreados (`-o`), o archivos modificados (`-m`).

Consideraciones de Diseño

- **Modularidad:**
  - Las funciones están diseñadas de manera modular para realizar operaciones específicas.
  - Se hace uso de la estructura `Index` para manejar el índice y verificar la existencia de archivos y directorios.
- **Manejo de Errores:**
  - Se utiliza `io::Result` para manejar errores en las operaciones de lectura/escritura.
  - Se informan mensajes de error descriptivos en caso de problemas.
- **Opciones de Comando:**
  - Se implementan opciones adicionales (`o` y `m`) para listar archivos no rastreados y modificados, respectivamente.
  - Se utiliza la opción `c` para listar archivos en el índice.



ls-tree

La función `ls_tree` permite listar blobs en un objeto tipo árbol referenciado por un hash, ya sea un commit o un árbol, y también proporciona opciones para listar recursivamente (`-r`), mostrar solo subárboles (`-d`), o listar recursivamente incluyendo tipos de objetos (`-r-t`).

Función Principal

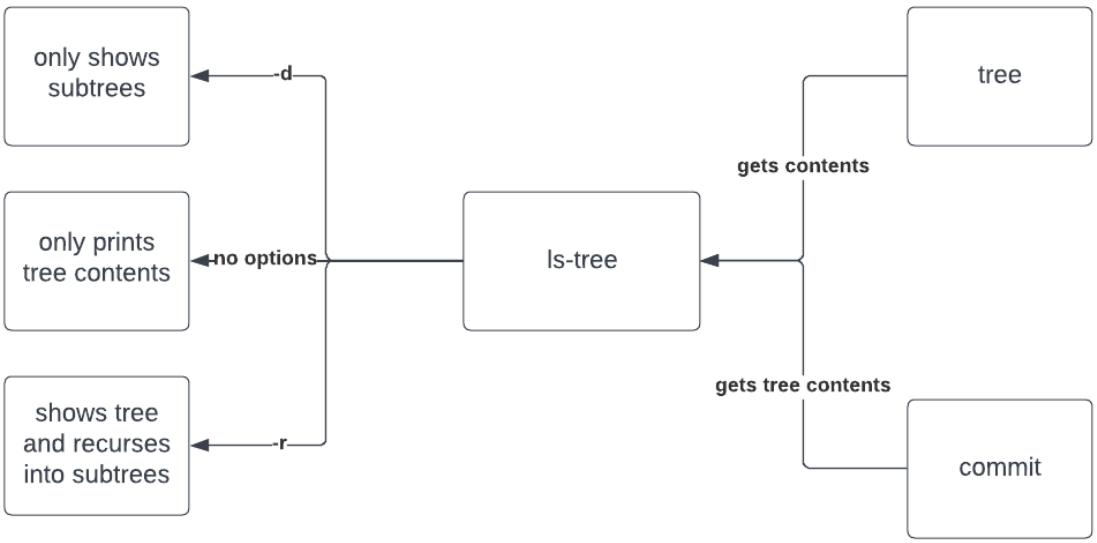
`ls_tree`

La función `ls_tree` toma un hash que apunta a un objeto de tipo árbol (ya sea un commit o un árbol) y realiza la siguiente lógica:

1. Intenta cargar el árbol desde un commit y, si falla, intenta cargarlo desde un archivo. Si ambos fallan, devuelve un error indicando que no es un árbol.
2. Según la opción proporcionada (`""`, `r`, `d`, `r-t`), realiza las siguientes acciones:
   - `""`: Imprime el árbol en la salida.
   - `r`: Imprime recursivamente los blobs sin incluir subárboles.
   - `d`: Imprime solo los subárboles.
   - `r-t`: Imprime recursivamente incluyendo tipos de objetos.

Consideraciones de Diseño

- **Manejo de Errores:**
  - Se manejan errores al intentar cargar el árbol desde un commit o un archivo.
- **Opciones de Comando:**
  - La función acepta diferentes opciones para personalizar la salida, lo que permite una flexibilidad en la visualización de la estructura del árbol.
- **Modularidad:**
  - La función hace uso de un módulo externo (`tree_handler`) para cargar el árbol desde el commit o el archivo.



show-ref

La función `git_show_ref` proporciona información sobre las referencias Git en base a los argumentos proporcionados en la línea de comandos. El comando ofrece funcionalidades para mostrar referencias de heads, tags, y realizar verificación de referencias.

Funciones Principales

`git_show_ref`

La función `git_show_ref` es la entrada principal para mostrar información sobre las referencias Git. Se encarga de manejar diferentes casos según la cantidad de argumentos y opciones proporcionadas. Los casos incluyen la visualización de heads, tags, y la verificación de referencias. También maneja casos de error como opciones inválidas o un número incorrecto de argumentos.

`show_ref`

La función `show_ref` muestra referencias en el directorio Git para heads y tags. Enumera las referencias con información de tipo, imprimiendo el contenido y el tipo asociado, así como el nombre del archivo para heads y tags.

`show_ref_with_options`

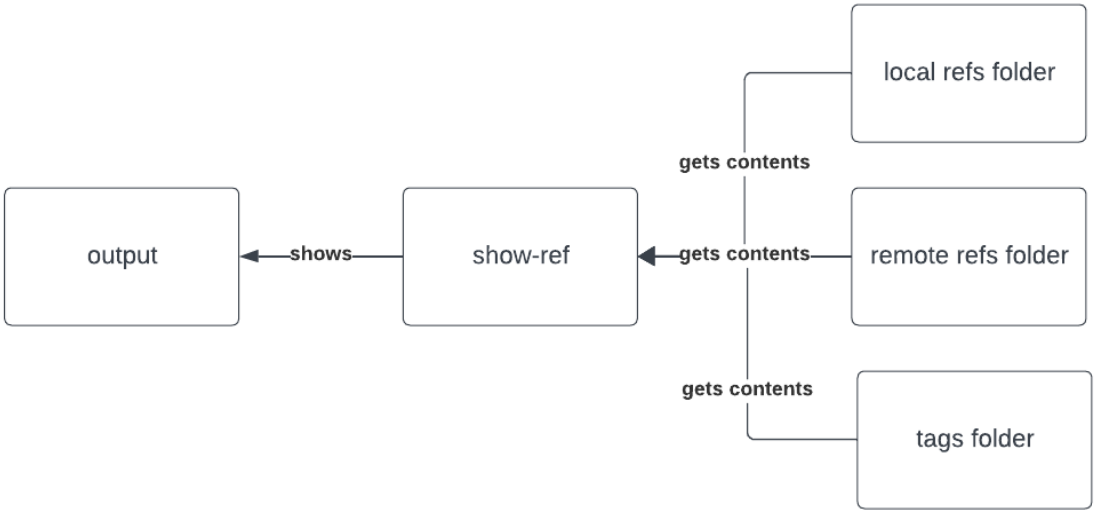
La función `show_ref_with_options` muestra referencias en el directorio Git según las opciones proporcionadas en la línea de comandos. Admite opciones como `--heads`, `--tags`, `--hash`, y `--verify`. Dependiendo de la opción, imprime referencias con o sin información de tipo y maneja la verificación de referencias.

`verify_ref`

La función `verify_ref` verifica y muestra información sobre las referencias Git especificadas. Para cada referencia proporcionada, verifica si existe en el directorio Git y, si es así, imprime el contenido y el nombre de la referencia. Si la referencia no existe, imprime un mensaje de error fatal.

`process_files_in_directory`

La función `process_files_in_directory` procesa archivos en el directorio especificado y es utilizada para mostrar heads, tags, y referencias en la carpeta remotes. Dependiendo de la opción `is_hash`, imprime solo el contenido o incluye información de tipo y nombre de archivo.



rebase

El código de este comando está organizado en funciones que realizan diversas tareas.

Funciones Principales:

1. `create_rebasing_commit`
    - Toma tres commits: nuestro commit actual (`our_commit`), el commit que ha sido aplicado (`rebased_commit`), y el ancestro común de ambos (`common_ancestor`).
    - Carga los árboles de estos commits.
    - Determina los archivos que han cambiado entre el ancestro común y nuestro commit actual y los archivos que no han cambiado en el commit aplicado.
    - Actualiza el árbol de la versión "rebaseada" con los cambios necesarios.
    - Interactúa con la interfaz gráfica para permitir al usuario revisar y realizar cambios antes de finalizar el rebase.
2. `rebase`
    - Recibe información sobre dos ramas en Git (`our_branch` y `their_branch`) y el directorio de trabajo (`git_dir`).
    - Calcula el ancestro común de ambas ramas utilizando `find_common_ancestor`.
    - Obtiene la historia de commits de nuestra rama (`our_branch_commits`).

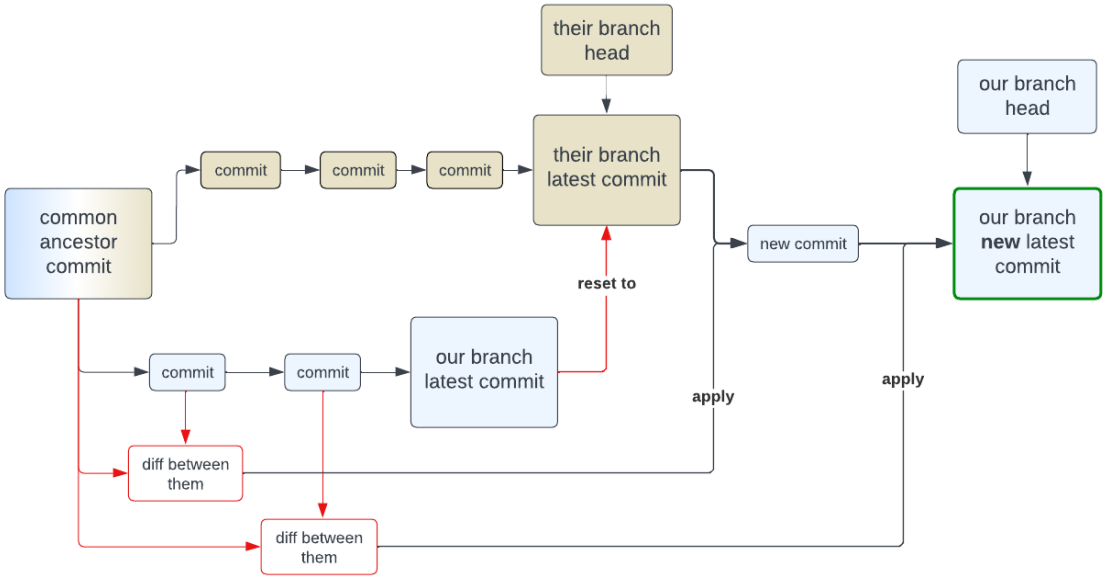
- Itera sobre los commits, utilizando `create_rebasing_commit` para realizar el rebase interactivo.

Funciones de Utilidad:

1. `update_files_to_change`
   - Actualiza un mapa de archivos a cambiar, generando un diff entre dos commits dados y almacenando la diferencia en el mapa.
2. `update_combo_box_text`
   - Actualiza un `ComboBoxText` en la interfaz gráfica con las opciones proporcionadas en un `HashMap`.
3. `combo_box_connect_changed`
   - Conecta el evento de cambio de un `ComboBoxText` a una función específica que actualiza un `TextView` con el contenido seleccionado.
4. `rebase_button_on_clicked`
   - Maneja el evento de clic en un botón, escribiendo el contenido de un `TextView` en un archivo temporal.
5. `write_diffs_in_files`
   - Escribe las diferencias almacenadas en un mapa de archivos en archivos temporales.
6. `update_files_to_change_from_files`
   - Lee los archivos temporales creados por `write_diffs_in_files` y los almacena en un nuevo mapa.
7. `update_rebase_tree`
   - Actualiza un árbol de rebase con la información proporcionada en un mapa.
8. `rebase_ok_all_button_on_clicked`
   - Maneja el evento de clic en un botón, actualiza archivos, crea un nuevo commit de rebase y devuelve su hash.

- El código utiliza hilos para realizar operaciones asíncronas y espera hasta que se complete una operación antes de continuar.

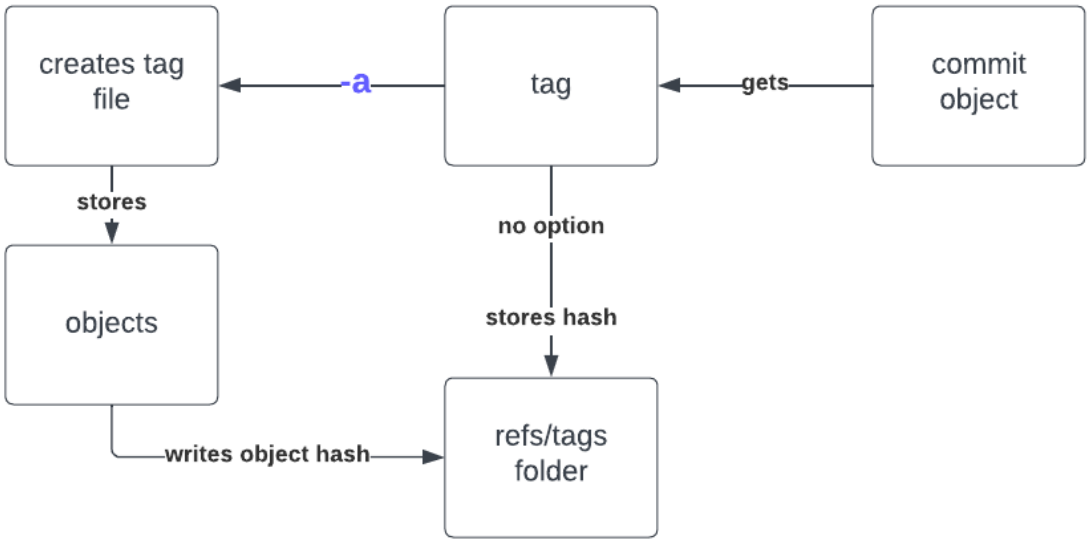
- Se utiliza un enfoque interactivo para permitir al usuario revisar y confirmar cada paso del rebase.



tag

La implementación se organiza en funciones modulares, cada una abordando una tarea específica relacionada con el comando `git tag`.

1. `list_tags`: Lista etiquetas existentes en el directorio de etiquetas (`refs/tags`).
2. `create_tag`: Crea una nueva etiqueta simple asociada al commit actual.
3. `create_annotated_tag`: Crea una etiqueta anotada con información adicional.
4. `copy_tag`: Duplica una etiqueta existente con un nombre diferente.
5. `delete_tag`: Elimina etiquetas existentes y proporciona información sobre la operación.
6. `verify_tag`: Verifica la integridad de una etiqueta existente y muestra su información.
7. `git_tag`: Función principal que interpreta los argumentos de línea de comandos y dirige la ejecución.



Servidor :

La implementación del servidor está compuesta por muchas funciones. Algunas funciones clave incluyen:

1. `ServerInstance::new`**:** Inicializa una nueva instancia del servidor, configurando la ubicación de los repositorios y el directorio Git.
2. `ServerInstance::handle_client`**:** Gestiona las solicitudes del cliente, incluyendo la lectura del comando, manejo de errores y ejecución de las operaciones correspondientes.
3. `ServerInstance::upload_pack`**:** Maneja la solicitud `git-upload-pack`, enviando referencias al cliente y calculando y enviando objetos faltantes.
4. `ServerInstance::receive_pack`**:** Maneja la solicitud `git-receive-pack`, enviando referencias al cliente y actualizando el repositorio con los cambios proporcionados por el cliente.
5. **Funciones auxiliares:** Se han implementado funciones adicionales para tareas específicas, como el manejo de referencias, la lectura de comandos y la manipulación de archivos.

`upload_pack` :

1. **Envío de Referencias:**
    - La función comienza enviando las referencias del servidor al cliente mediante la función `send_refs`. Esto proporciona al cliente información sobre las ramas y etiquetas disponibles en el repositorio remoto.
2. **Recepción de 'Wants' y 'Haves':**
    - Luego, espera a que el cliente envíe los objetos que necesita para completar su historial local. Esto se hace a través de la función `read_wants_haves`. El servidor lee los "wants" (objetos que el cliente desea) y los "haves" (objetos que el cliente ya tiene).
3. **Creación y Envío del Packfile:**
    - Después de recibir esta información, el servidor determina los objetos que faltan y crea un packfile que contiene estos objetos. Este packfile se envía al cliente, optimizando la transferencia de datos al enviar solo lo necesario.

`receive_pack`:

1. **Envío de Referencias:**
    - Similar a `upload_pack`, comienza enviando las referencias del servidor al cliente a través de la función `send_refs`.
2. **Espera de Cambios del Cliente:**
    - Luego, espera a que el cliente envíe las actualizaciones propuestas para las referencias mediante la función `wait_changes`. Esto implica recibir información sobre las referencias afectadas y los nuevos y viejos valores de los hashes asociados.
3. **Manejo de Packfile (si es necesario):**
    - Dependiendo de los cambios propuestos, puede ser necesario esperar a que el cliente envíe un packfile. Esto se gestiona a través de la función `wait_and_unpack_packfile`.
4. **Actualización de Referencias:**
    - Finalmente, el servidor utiliza la función `make_refs_changes` para aplicar las modificaciones propuestas por el cliente. Esto puede incluir la creación de

nuevas referencias, actualización de referencias existentes o eliminación de referencias, según los cambios recibidos.

Cliente:

Estructura del Cliente (`Client`):

- **Atributos:**
  - `address`, `repository`, y `host`: Almacenan la información necesaria para establecer una conexión TCP con el servidor Git.
  - `socket`: Un `TcpStream` que representa la conexión TCP con el servidor.
  - `git_dir` y `remote`: Almacenan información sobre el directorio Git local y el nombre del control remoto.
- **Métodos Principales:**
  - `new`: Constructor para inicializar un nuevo cliente con la información básica.
  - `get_server_refs`: Establece una conexión, solicita las referencias del servidor y las devuelve como un hashmap.
  - `upload_pack`: Realiza la lógica de `git-upload-pack`. Conecta al servidor, solicita referencias, y maneja la transferencia de objetos entre el cliente y el servidor.
  - `receive_pack`: Realiza la lógica de `git-receive-pack`. Conecta al servidor, maneja las referencias locales y actualiza el servidor con las nuevas referencias.
- **Métodos Auxiliares:**
  - `clear`: Limpia el estado del cliente.
  - `connect`: Establece una conexión TCP con el servidor.
  - `initiate_connection`: Inicia la conversación con el servidor para un comando específico (`git-upload-pack` o `git-receive-pack`).
  - `wait_server_refs`: Espera las referencias del servidor y las almacena en `self.server_refs`.
  - `want_branches`, `send_wants`, `send_haves`: Manejan la lógica de intercambio de información sobre ramas y objetos entre el cliente y el servidor.

- `wait_and_unpack_packfile`: Espera la recepción de un packfile y lo desempaqueta en el directorio local de Git.
- `update_remote`: Actualiza una referencia remota en el cliente.
- `receive_pack_create`, `receive_pack_update`: Manejan la lógica de actualización de referencias en el servidor durante un `git-receive-pack`.
- `socket`, `end_connection`, `send`, `send_bytes`, `flush`, `done`: Métodos auxiliares para interactuar con el socket.

Uso del Cliente:

- **Configuración Inicial:**
  - Se crea una instancia de `Client` con la dirección del servidor, el nombre del repositorio y el nombre del host.
  - El cliente puede luego llamar a varios métodos para realizar operaciones específicas de Git, como obtener referencias del servidor, realizar `git-upload-pack` o `git-receive-pack`.
- **Comunicación con el Servidor:**
  - La comunicación con el servidor se realiza a través de operaciones en el socket TCP.
  - Se utilizan funciones auxiliares para enviar y recibir mensajes según el formato del protocolo Git.
- **Manejo de Errores:**
  - El código maneja errores de I/O y proporciona mensajes de error descriptivos en caso de falla.

Manual para usarlo con Git Daemon:

El siguiente manual proporciona instrucciones sobre cómo configurar y utilizar Git Daemon para compartir repositorios de manera sencilla a través de la red. Además, se incluyen pasos para clonar y configurar repositorios tanto desde un cliente local como desde la interfaz gráfica proporcionada en el código Rust.

Configuración del Servidor Daemon:

1. **Inicializar un Repositorio Git Bare:**

```
git init --bare server
```

   - Esto crea un repositorio Git en modo "bare" en la carpeta “server”, que es necesario para el uso con Git Daemon.

2. **Iniciar el Servidor Git Daemon:**

```
git daemon --base-path=. --export-all --enable=receive-pack --reuseaddr --informative
-errors --verbose
```

   - Inicia el servidor Git Daemon con varias opciones:
     - `-base-path`: La ruta base para los repositorios.
     - `-export-all`: Exporta todos los repositorios.
     - `-enable=receive-pack`: Permite la recepción de paquetes.
     - `-reuseaddr`: Reutiliza la dirección.
     - `-informative-errors`: Muestra errores informativos.
     - `-verbose`: Muestra mensajes detallados.

Clonar Repositorio desde Cliente Local:

1. **Clonar desde Cliente Local:**

```
git clone git://127.0.0.1/server
```

   - Clona el repositorio desde el servidor local a una carpeta diferente.
   - Puedes probar realizar cambios y hacer push al repositorio original para luego clonarlo en el nuestro.

2. **Configurar Remotos:**
    - Configura los remotos según sea necesario para gestionar el flujo de trabajo con el servidor.

Configuración en la Interfaz Gráfica (Rust):

1. **Clonar Repositorio en la Interfaz Gráfica:**
    - En la interfaz gráfica, ingresa la siguiente URL para clonar el repositorio:

      ```
      localhost:9418/server
      ```

    - Puedes cambiar el nombre 'server' por el que prefieras.
2. **Iniciar Nuestro Servidor (Rust):**

    ```
    cargo run server localhost 9418 /ruta/a/tu/carpeta/server
    ```

    - Reemplaza "/ruta/a/tu/carpeta/server" con la ubicación donde se encuentra la carpeta 'server'.
    - Esto inicia nuestro servidor Rust, que se conectará como el servidor Git Daemon.

Conclusion

A lo largo de este trabajo práctico, adquirimos una comprensión más profunda de los conceptos fundamentales que respaldan el funcionamiento de Git. Desde la manipulación de objetos hasta la gestión de referencias y la implementación de un protocolo de comunicación eficiente.

Este trabajo fortaleció nuestra comprensión teórica de Git, y también nos brindó la oportunidad de aplicar estos conocimientos en un contexto práctico. Al construir nuestra propia versión simplificada de Git, pudimos internalizar los conceptos y procesos que subyacen en esta herramienta esencial para el desarrollo de software.

A lo largo de este informe, detallamos los pasos clave que seguimos en la implementación.