目录

init 模块

一：整个项目的构成

```
/ (sysmaster)
|...init (init进程)
|...factory (系统配置)
|...libs (对外接口)
|     |...libtests (test lib crate)
|     |...cgroup (cgroup lib crate)
|     |...cmdproto(cmd proto lib crate)
|...exts (sysmaster-extends组件)
|     |...devmaster (daemon)
|     |...random-seed (bin)
|...core (sysmaster-core核心组件)
|     |...sysmaster (bin)
|     |...libcore (internal lib)
|     |...sctl (sysmaster cli)
|     |...coms (插件)
|          |...service (unit type crate)
|          |...socket  (unit type crate)
|          |...target  (unit type crate)
|...tools
|     |...musl_build
|     |...run_with_sd
|...docs (sysmaster.online)
|...build.sh (准备环境)
```

二：runtime.rs 的 impl 块简要介绍

分析的功能主要位于 init 的 runtime.rs 文件，runtime.rs 中不同 impl 块的作用：

new：构造函数，用于初始化 Runtime 结构体。

register 和 deregister：用于注册和注销事件源。

load_config：加载配置文件。

reap_zombies：回收僵尸进程。

handle_signal：处理接收到的信号。

handle_timer：处理计时器事件。

handle_socket：处理套接字事件。

pid_is_running：检查特定 PID 的进程是否在运行。

start_bin：启动二进制文件。

runloop：运行事件循环。

is_running 和 set_state：检查和设置初始化程序的状态。

reload：重新加载配置。

is_reexec 和 reexec：检查是否需要重新执行和执行重新启动。

kill_sysmaster 和 exit：终止系统进程和退出程序。

三：init 的进程回收机制

在 Linux 系统中，子进程的回收是一个重要的管理任务，它涉及到资源的有效利用和系统的稳定性。如果父进程未能妥善回收其子进程，就可能导致僵尸进程（Zombie Process）的产生 。

僵尸进程占用系统资源：当子进程结束时，如果父进程没有通过 wait()或 waitpid()函数来回收子进程的资源和状态信息，子进程就会变成僵尸进程。僵尸进程会占用系统资源，如进程描述符（PCB）。

系统稳定性受到影响：僵尸进程的存在可能会影响系统的稳定性。如果系统资源被大量僵尸进程占用，可能会导致新的进程无法被创建，从而影响系统的正常运行。

信号处理的延迟性：系统向父进程发送 SIGCHLD 信号来通知子进程的结束。如果父进程没有处理这个信号，子进程同样会变成僵尸进程。

master-init 的 init 进程针对上述缺点的改进如下所示

3.1 非阻塞事件驱动：

```rust
// 结合使用 WNOHANG 和 WNOWAIT 标志位，使得 waitid 调用不会阻塞父进程，即使没有子进程退出。
// 父进程可以继续执行其他任务，同时周期性地检查是否有子进程终止，而不会在 waitid 调用上浪费任何时间。

// 定义位置 : use nix::sys::wait::{waitid, Id, WaitPidFlag, WaitStatus};
let flags = WaitPidFlag::WEXITED | WaitPidFlag::WNOHANG | WaitPidFlag::WNOWAIT;
```

使用 WNOHANG 和 WNOWAIT 标志位，使得 waitid ()调用不会阻塞父进程。父进程可以继续执行其他任务，同时周期性地检查是否有子进程终止，而不会在 waitid()调用上浪费任何时间。使用非阻塞模式代替了原来的阻塞模式。

```rust
// pop: recycle the zombie
if let Some((pid, _, _)) = si {
    // 这句话会回收子进程的资源
    if let Err(e) = waitid(Id::Pid(pid), WaitPidFlag::WEXITED) {
        log::error!("Error when reap the zombie({:?}), ignored: {:?}!", pid, e);
    }
}
```

具体而言这句代码完成了子进程的回收

3.2 使用 mio 库达成了高效的事件循环

```rust
use crate::config::Config;
use mio::{unix::SourceFd, Events, Interest, Poll, Token};
```

这句代码指定了事件循环库为 mio 库，mio 是 Rust 语言中的一个快速、低级别的 I/O 库，专注于非阻塞 API 和事件通知，用于构建高性能的 I/O 应用程序，同时尽可能减少操作系统抽象的开销。

3.3 错误处理的改进

```rust
// pop: recycle the zombie
if let Some((pid, _, _)) = si {
    // 这句话会回收子进程的资源
    if let Err(e) = waitid(Id::Pid(pid), WaitPidFlag::WEXITED) {
        log::error!("Error when reap the zombie({:?}), ignored: {:?}!", pid, e);
    }
}
```

错误处理改成了日志打印而不是错误退出，提高了系统的稳定性

3.4 信号量的使用

Master-init 通过提供一种更简洁的接口来初始化和使用信号量，从而减少了信号量的使用复杂性。它可能通过封装信号量的创建和销毁过程，以及提供更直观的等待和释放操作，来帮助开发者更容易地管理线程间的同步。定义信号结构的位置

```rust
pub struct Runtime {
    poll: Poll,
    timerfd: TimerFd,
    signalfd: SignalFd,
    socketfd: UnixListener,
    config: Config,
    state: InitState,
    // sysmaster pid
    pid: u32,
    // sysmaster status
    online: bool,
    deserialize: bool,
}
```

四：监控 sysmaster-core 的部分

Init 进程监控 sysmaster-core 的部分主要是 handle_timer 和 handle_socket。handle_timer 负责系统的定时器事件，handle_socket 负责了系统联网的事件处理。

4.1 handle_timer 模块：

```
if self.online {
    self.online = false;
} else {
    self.start_bin();
    self.config.timecnt -= 1;
}
self.timerfd.set(
    Expiration::OneShot(TimeSpec::seconds(self.config.timewait as i64)),
    TimerSetTimeFlags::empty(),
)?;
Ok(())
```

处理定时器事件，用于定期检查 sysmaster-core 的状态，并在必要时重新启动它。

4.2 handle_socket：

配合 master_core 的联网处理模块，使用 accept()函数接受来自 poll 的传递的调用更新对应 sysmaster-core 的状态值：online 和 pid。

```
let (stream, _) = match self.socketfd.accept() {
    Ok((connection, address)) => (connection, address),
    Err(e) if e.kind() == io::ErrorKind::WouldBlock => {
        // If we get a `WouldBlock` error we know our
        // listener has no more incoming connections queued,
        // so we can return to polling and wait for some
        // more.
        return Ok(());
    }
    Err(e) => {
        // If it was any other kind of error, something went
        // wrong and we terminate with an error.
        log::error!("Error accepting connection: {}!", e);
        return Err(e);
    }
};
```

```
// 检查连接的 PID 是否是 sysmaster-core 进程，如果是，则更新 online 和 pid
let credentials = getsockopt(stream.as_raw_fd(), PeerCredentials)?;
let pid = credentials.pid() as u32;
if self.pid_is_running(pid) {
    // If the incoming PID is not the monitored sysmaster,
    // do not refresh the status.
    self.online = true;
    self.pid = pid;
}
Ok(())
```

sysMaster-core-unit

一：主要功能

unit 是所有 sysmaster 管理的 subunit 的基础，sysmaster 通过配置文件来定义每个具体 subunit 实例的行为，unit 负责出所有 subunit 运行时功能行为的定义，以及设置为开机自动时需要的信息。sysmaster 通过 unit 管理操作系统中各类服务的生命周期，每类服务运行时需要执行生命类型的操作，通过配置文件来进行定义，对公共的行为进行同一的定义，将重复工作降到最低，此部分需求分析，需要结合系统维护过程中经常用到的属性，并需要考虑和 systemd 兼容的兼容性。

二：总体方案

要实现通过 unit 管理系统中的服务，包含以下步骤：

1. 解析 unit 的配置文件，转换成 unit 对象。
2. 执行 unit 对应的动作。

整体模块划分包括：

1. load 模块，将配置文件转换成 unit。
2. unit_dataStore，保存 unit 的状态。
3. unit 模块，包含 unit 属性。

三：功能设计

3.1 功能域划分

结合场景分析，unitManger 功能域的划分逻辑如下（图 1）：

unit 为基本管理单元，并且会有不同类型的 unit，因此需要先抽象一个 unit 接口层，并且需要有不同的实现。

因为不同类型的 unit 的存在，unit 管理框架设计成可扩展的，可以动态加载不同 unit 的实现。

每个 unit 有独立的配置文件，因此需要有一个配置文件管理的模块，来管理 unit 配置文件，并完成从配置文件解析生成 unit 对象。

unit 状态会发生变化，并且还需要维护 unit 的依赖关系，因此需要定义一个 DataStore 模块，统一存储 unit 对象。

unit 对象是有生命周期的，因此需要有一个整体的管理模块，将所有模块串联起来。

unit 对象每次启动，通过 job 出发，因此需要划分出一个 job 引擎。

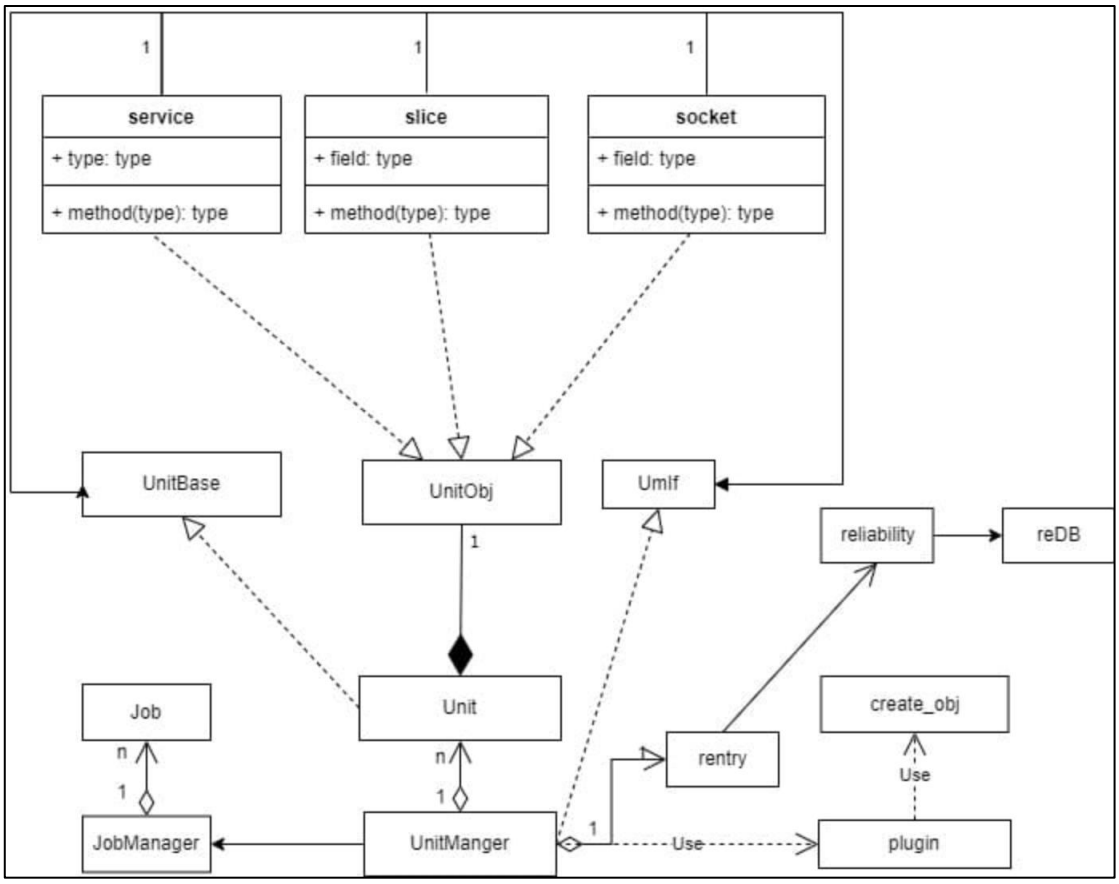


图 1:功能域划分

3.2 unit 加载

3.2.1 unit 的配置管理

每个 unit 都会包含配置文件，并且 unit 的配置文件和 Unit 同名，并且以 unit 的类型作为扩展名，如 XXX.service, XXX.socket。配置目录下可以存在同名的文件，同名的文件高优先级的会覆盖低优先级的的配置，unit 的配置文件和 unit 的对应关系会在首次加载的时候完成缓存，只有在目录发生更新的时候才会刷新缓存的映射关系。unit 配置使用 toml 格式，通过对 toml 文件解析，填充到具体 unit 对象中。

3.2.2 unit 的加载

unit 对象加载是所有 unit 执行后续的动作的前提，通过解析 unit 的对应的配置文件，生成 unit 对象，并加载到 sysmaster 内部，具体流程如下（图 2）：

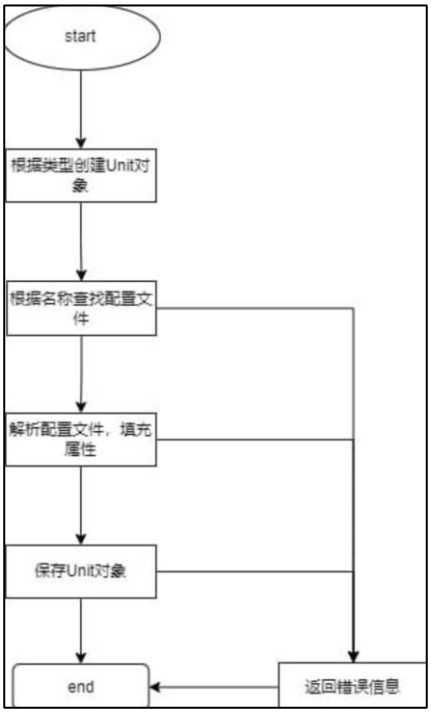


图 2:unit 对象加载

3.2.3 unit 对象创建

unit 对象包含 unit 以及子 unit，对象关系如下图（图 3）：

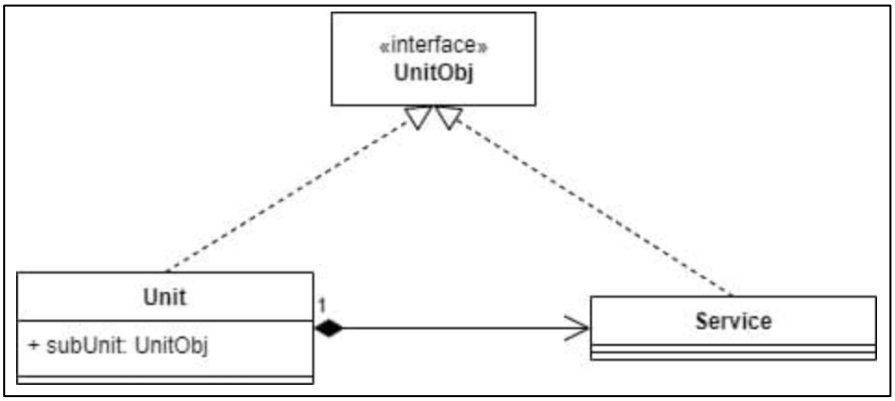


图 3:unit 对象创建

unit 为 unitManger 管理的单元，subUnit 为子 unit，每种类型都要求实现 unitObj 接口，不同类型有不同的的实现，通过 Plugin 框架来创建。

3.2.4 unit 对象的存储

每个配置文件，解析完成之后，会生成一个 unit 对象，unit 全局唯一，同一保存到 datastore 中，datastore 使用 hashmap，并且使用 name 作为 key 来保存 unit 对象。使用过程中，unit 使用引用的方式更新数据，当前 sysmater 使用的是单线程，因此不考虑数据并发问题。

四：具体实现

4.1 代码组成逻辑

unit 模块的代码组成逻辑如下图（图 4）：

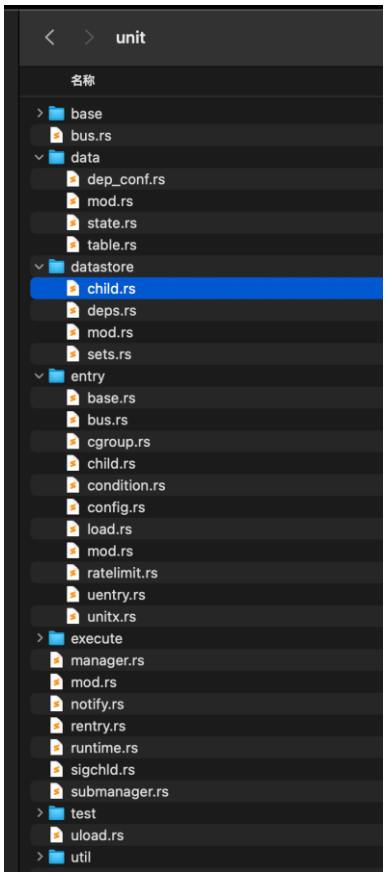


图 4:代码组成

base 模块主要包含了各个 unit 单元之间的依赖关系和关系原子之间的转换，这些依赖关系在启动、停止和管理服务单元时非常重要。

data 模块主要包含了管理 unit 单元相关的不同种类的数据，例如依赖配置，单元状态，启动限制结果和作业结果并实现了 ReStation trait，这可能是用于处理数据重新加载和清除的 trait。

datastore 模块主要包含了用于管理系统服务单元子进程的模块，包括子进程的监视和数据管理。用于管理系统服务单元依赖关系的模块，包括依赖关系的插入、移除和查询。用于管理服务单元集合的模块，包括单元的插入、移除、获取和订阅功能。

entry 模块主要包含了 unit 进程的创建，load 配置文件，状态信息存入外部数据库等等功能。

4.2 unit 创建调度流程

定义了一个名为 UeLoad 的结构体，它似乎用于表示一个 unit 的加载状态和管理其生命周期。UeLoad 结构体包含多个字段，包括一些与单元相关的对象（如 DataManager、UnitFile、UeBase 和 UeConfig）的引用计数对象（Rc），以及一些用于存储单元状态的 RefCell。ReStation trait 的实现提供了 db_map 和 db_insert 方法，这些方法可能用于将单元的状态映

射到数据库，并在需要时插入数据。

```rust
pub(super) fn new(
    dmr: &Rc<DataManager>,
    filer: &Rc<UnitFile>,
    baser: &Rc<UeBase>,
    config: &Rc<UeConfig>,
) -> UeLoad {
    let load = UeLoad {
        dm: Rc::clone(dmr),
        file: Rc::clone(filer),
        base: Rc::clone(baser),
        config: Rc::clone(config),
        transient: RefCell::new(false),
        paths: RefCell::new(Vec::new()),
        load_state: RefCell::new(UnitLoadState::Stub),
        in_load_queue: RefCell::new(false),
        in_target_dep_queue: RefCell::new(false),
        transient_file: RefCell::new(None),
        last_section_private: RefCell::new(-1),
    };

    ///创建的 UeLoad 实例的数据插入到数据库中
    load.db_insert();

    ///创建一个 flags 变量，它是一个 UnitRePps 枚举值的组合，表示要清除的加载队列和目标依赖队列的标志
    let flags = UnitRePps::QUEUE_LOAD | UnitRePps::QUEUE_TARGET_DEPS;

    ///清除相关的处理标志，这可能是为了确保新创建的 UeLoad 实例不会立即进入处理队列。
    load.base.rentry_pps_clear(flags);
    load
```

```rust
fn db_map(&self, reload: bool) {
    if reload {
        return;
    }
    if let Some((load_state, transient, paths, transient_file, last_section_private)) =
        self.base.rentry_load_get()
    {
        *self.load_state.borrow_mut() = load_state;
        *self.transient.borrow_mut() = transient;
        *self.paths.borrow_mut() = paths;
        *self.transient_file.borrow_mut() = transient_file;
        *self.last_section_private.borrow_mut() = last_section_private;
    }
}

fn db_insert(&self) {
    self.base.rentry_load_insert(
        *self.load_state.borrow(),
        *self.transient.borrow(),
        self.paths.borrow().clone(),
        self.transient_file.borrow().clone(),
        *self.last_section_private.borrow(),
    );
}
```

用于解析单元的配置数据并更新依赖关系。

```rust
///函数接收两个参数：files，一个指向 UnitFile 结构体的引用，它包含了单元文件的信息；name，单元的名称
pub(super) fn load_fragment_and_dropin(&self, files: &UnitFile, name: &str) -> Result<()> {

    ///获取单元的配置片段路径缓冲区
    let unit_conf_frag = files.get_unit_id_fragment_pathbuf(name);

    if unit_conf_frag.is_empty() {
        return Err(format!("{} doesn't have corresponding config file", name).into());
    }

    ///加载配置数据
    let mut configer = match UeConfigData::load_config(unit_conf_frag, name) {
        Ok(v) => v,
        Err(e) => {
            log::error!("Invalid Configuration: {}", e);
            return Err(Error::ConfigureError {
                msg: format!("Invalid Configuration: {}", e),
            });
        }
    };

    ///对于每个 wants 类型的符号链接单元，将其添加到 configer.Unit.Wants 和 configer.Unit.After 列表中
    for v in files.get_unit_wants_symlink_units(name) {
        configer.Unit.Wants.push(v.to_string_lossy().to_string());
        configer.Unit.After.push(v.to_string_lossy().to_string());
    }

    ///对于每个 requires 类型的符号链接单元，将其添加到 configer.Unit.Requires 和 configer.Unit.After 列表中
    for v in files.get_unit_requires_symlink_units(name) {
        configer.Unit.Requires.push(v.to_string_lossy().to_string());
        configer.Unit.After.push(v.to_string_lossy().to_string());
    }

    ///创建一个新的 unit_specifier_data 实例
    let mut unit_specifier_data = UnitSpecifierData::new();
    unit_specifier_data.instance = unit_name_to_instance(&self.base.id());

    ///更新配置数据
    configer.update_with_specifier_escape(&unit_specifier_data);

    *self.data.borrow_mut() = configer;

    ///更新数据库
    self.db_update();

    Ok(())
}
```

```rust
fn parse(&self) {
    let mut ud_conf = UnitDepConf::new(); // need get config from config database,and update depends hereW
    let config_data = self.config.config_data();
    let start_limit_interval = config_data.borrow().Unit.StartLimitInterval;
    let start_limit_interval_sec = config_data.borrow().Unit.StartLimitIntervalSec;
    if start_limit_interval != start_limit_interval_sec {
        ///如果它们不相等，那么会根据是否是默认值来决定使用哪个值更新 star_limit_interval
        if start_limit_interval != 10 {
            config_data.borrow_mut().Unit.StartLimitInterval = start_limit_interval;
        } else {
            /* If StartLimitInterval is the default value, use StartLimitIntervalSec. */
            config_data.borrow_mut().Unit.StartLimitInterval = start_limit_interval_sec;
        }
    }

    ///创建一个名为 ud_conf_insert_table 的向量，它包含了一系列的元组，每个元组包含一个 UnitRelations
    ///枚举值和一个与之关联的值（通常是字符串的 Vec）。这些元组代表了单元的依赖关系。
    let ud_conf_insert_table = vec![
        (
            UnitRelations::UnitWants,
            config_data.borrow().Unit.Wants.clone(),
        ),
        (
            UnitRelations::UnitAfter,
            config_data.borrow().Unit.After.clone(),
        ),
        (
            UnitRelations::UnitBefore,
            config_data.borrow().Unit.Before.clone(),
        ),
        (
            UnitRelations::UnitRequires,
            config_data.borrow().Unit.Requires.clone(),
        ),
        (
            UnitRelations::UnitBindsTo,
            config_data.borrow().Unit.BindsTo.clone(),
        ),
        (
            UnitRelations::UnitRequisite,
            config_data.borrow().Unit.Requisite.clone(),
        ),
```

加载一个服务单元，并处理在加载过程中可能发生的错误

```rust
pub(super) fn load_unit(&self) -> Result<()> {
    ///用来标记单元是否已经加入加载队列
    self.set_in_load_queue(false);

    ///这个方法可能是用来处理单元的临时状态
    self.load.finalize_transient()?;

    ///用来加载单元的配置
    match self.load.load_unit_confs() {
        Ok(_) => {
            ///获取单元的路径缓冲区 paths
            let paths = self.load.get_unit_id_fragment_pathbuf();
            log::debug!("Begin exec sub class load");

            ///加载子单元，并处理可能发生的错误
            if let Err(err) = self.sub.load(paths) {
                if let Error::Nix { source } = err {
                    if source == nix::Error::ENOEXEC {
                        self.load.set_load_state(UnitLoadState::BadSetting);
                        return Err(err);
                    }
                }
                self.load.set_load_state(UnitLoadState::Error);
                return Err(err);
            }

            self.load.set_load_state(UnitLoadState::Loaded);
            Ok(())
        }
        Err(e) => {
            self.load.set_load_state(UnitLoadState::NotFound);
            Err(e)
        }
    }
}
```

尝试启动一个服务单元，并在启动过程中进行一系列的检查和条件测试

```rust
pub fn start(&self) -> Result<()> {
    ///获取单元的当前活跃状态
    let active_state = self.current_active_state();
    if active_state.is_active_or_reloading() {
        log::debug!(
            "The unit {} is already active or reloading, skipping.",
            self.id()
        );
        return Err(Error::UnitActionEAlready);
    }

    ///如果单元的活跃状态是维护模式
    if active_state == UnitActiveState::Maintenance {
        log::error!("Failed to start {}: unit is in maintenance", self.id());
        return Err(Error::UnitActionEAgain);
    }

    if self.load_state() != UnitLoadState::Loaded {
        log::error!("Failed to start {}: unit hasn't been loaded.", self.id());
        return Err(Error::UnitActionEInval);
    }

    if active_state != UnitActiveState::Activating && !self.conditions().conditions_test() {
        log::info!("The condition check failed, not starting {}.", self.id());
        return Err(Error::UnitActionEComm);
    }

    if active_state != UnitActiveState::Activating && !self.conditions().asserts_test() {
        log::info!("The assert check failed, not starting {}.", self.id());
        return Err(Error::UnitActionEProto);
    }

    self.sub.start()
}
```

尝试重新加载一个服务单元，并在重新加载过程中进行一系列的检查。

```rust
/// reload the unit
pub fn reload(&self) -> Result<()> {

    ///检查单元是否能够被重新加载
    if !self.sub.can_reload() {
        log::info!("Unit {} can not be reloaded", self.id());
        return Err(Error::UnitActionEBadR);
    }

    let active_state = self.current_active_state();

    if active_state == UnitActiveState::Reloading {
        log::info!("Unit {} is being reloading", self.id());
        return Err(Error::UnitActionEAgain);
    }

    if active_state != UnitActiveState::Active {
        log::info!("Unit {} is not active, no need to reload", self.id());
        return Err(Error::UnitActionENoExec);
    }

    log::info!("Reloading {}", self.id());
    match self.sub.reload() {
        Ok(_) => Ok(()),
        Err(e) => match e {
            Error::UnitActionEOpNotSupp => {
                self.notify(active_state, active_state, UnitNotifyFlags::EMPTY);
                Ok(())
            }
            _ => Err(e),
        },
    }
}
```

调度并执行加载队列中的单元

```rust
pub(self) fn dispatch_load_queue(&self) {
    ///首先检查加载队列 self.load_queue 是否为空。如果为空，它将调用 dispatch_target_dep_queue 方法来处理目标依赖队列
    if self.load_queue.borrow().is_empty() {
        self.dispatch_target_dep_queue();
        return;
    }

    log::debug!("Dispatching load queue");

    ///使用 self.reli.set_last_frame2 方法设置最后一个框架的状态，表示正在处理加载队列
    self.reli
        .set_last_frame2(ReliLastFrame::Queue as u32, ReliLastQue::Load as u32);

    ///进入一个循环，该循环将持续从加载队列中弹出单元直到队列为空
    loop {
        //Limit the scope of borrow of load queue
        //unitX pop from the load queue and then no need the ref of load queue
        //the unitX load process will borrow load queue as mut again
        // pop
        let unit = match self.load_queue.borrow_mut().pop_front() {
            None => break,
            Some(v) => v,
        };

        log::debug!("Loading unit: {}", unit.id());
        self.reli.set_last_unit(&unit.id());
        if let Err(e) = unit.load() {
            log::error!("Failed to load unit [{}]: {}", unit.id(), e);
        }

        let real_name = unit.get_real_name();
        if !real_name.is_empty() {

            ///如果 real_name 不为空，表示正在启动一个别名，需要将其合并到真实单元
            log::debug!("Merging {} to {}", unit.id(), real_name);
            match self.db.units_get(&real_name) {

                ///如果数据库中没有真实单元，将当前单元的 ID 更改为真实单元的 ID，并插入数据库
                None => {
                    unit.set_id(&real_name);
                    self.db.units_insert(real_name.to_string(), unit.clone());
                }

                ///如果数据库中已有真实单元，将当前单元的加载状态设置为 Merged，并将其合并到真实单元中
                Some(u) => {
                    unit.set_load_state(UnitLoadState::Merged);
                    unit.set_merge_into(Some(u.clone()));
                    self.db.units_insert(unit.id().to_string(), u);
                }
            }
        } else {
            /* We are starting a real unit, remember its aliases. */
            for alias_name in unit.get_all_names() {
                log::debug!("Add name {} to {}", alias_name, real_name);
                self.db.units_insert(alias_name, unit.clone());
            }
        }
```

```rust
        ///获取单元的加载状态 load_state。如果状态是 Loaded，则将单元推入目标依赖队列
        let load_state = unit.load_state();
        if load_state == UnitLoadState::Loaded {
            self.push_target_dep_queue(Rc::clone(&unit));
        }

        self.reli.clear_last_unit();
    }

    self.reli.clear_last_frame();
    self.dispatch_target_dep_queue();
}
```

4.3 单元的自恢复机制

4.3.1 实现原理

1. 状态外置：sysMaster 将单元的状态信息外置到数据库或其他持久化存储中，这样即使在崩溃后也能从这些持久化的状态信息中恢复。

2. 多级 Checkpoint：通过设置多个恢复点（Checkpoint），sysMaster 能够在发生故障时回滚到最近的稳定状态。

3. 热升级能力：sysMaster 支持不中断服务的热升级，通过动态加载新的模块或配置来

实现服务的更新。

4. 故障监测：通过监控单元的运行状态，一旦检测到异常，会自动触发恢复流程。

5. 秒级自愈：sysMaster 设计了快速响应机制，能够在秒级时间内恢复服务。

4.3.2 代码实现

定义了一个名为 ReliHistory 的结构体，它用于管理一个或多个数据库实例的历史记录。

switch：一个 RefCell 包装的 ReliSwitch 枚举，用于控制历史记录的行为。

dbs：一个 RefCell 包装的 HashMap，存储数据库实例，键是数据库的名称，值是实现了 ReDbTable trait 的动态分发对象的 Rc 智能指针。

```rust
pub struct ReliHistory {
    // control
    switch: RefCell<ReliSwitch>,

    // database: multi-instance(N)
    dbs: RefCell<HashMap<String, Rc<dyn ReDbTable>>>, // key: name, value: db
}
```

commit 方法用于将所有数据库实例的更改提交到数据库。flush 方法用于将所有数据库实例的更改刷新到数据库，并根据提供的 switch 值决定刷新策略。

```rust
#[allow(dead_code)]
pub fn commit(&self, env: &Env) {
    // create transaction
    let mut db_wtxn = ReDbRwTxn::new(env).expect("history.write_txn");

    // export to db
    for (_, db) in self.dbs.borrow().iter() {
        db.export(&mut db_wtxn);
    }

    // commit
    db_wtxn.0.commit().expect("history.commit");
}

pub(super) fn flush(&self, env: &Env, switch: ReliSwitch) {
    // create transaction
    let mut db_wtxn = ReDbRwTxn::new(env).expect("history.write_txn");

    // flush to db
    for (_, db) in self.dbs.borrow().iter() {
        db.flush(&mut db_wtxn, switch);
    }

    // commit
    db_wtxn.0.commit().expect("history.commit");
}
```

recover 这个方法的目的是提供一个全面的恢复流程，确保系统在面对配置更改、系统重启或其他中断后能够恢复到一个已知的、一致的状态。通过分步骤执行不同的恢复操作，它有助于确保系统的稳定性和数据的完整性。

```rust
pub fn recover(&self, reload: bool) {
    // ignore last's input
    self.last.ignore_set(true);

    ///是从数据库导入历史数据，以便恢复之前的状态
    self.history.import();
    self.input_rebuild();

    ///执行数据库补偿操作，以确保数据库状态与系统状态一致
    self.db_compensate();

    ///根据 reload 参数的值，可能仅映射数据库结果类参数或执行更全面的数据库映射
    self.db_map(reload);

    ///确保系统状态在恢复后是一致的
    self.make_consistent(reload);

    ///恢复上一次的忽略状态
    self.last.ignore_set(false);

    // clear last
    self.last.clear_unit();
    self.last.clear_frame();
}
```

可以在 unit 初始化的时候确认是否开启自恢复功能

```rust
fn main() -> Result<()> {
    let args = Args::parse();
    ignore_all_signals();

    // The registration signal is at the beginning and has the highest priority!
    register_reexec_signal(true);

    remount_sysroot();

    //创建一个系统模式下的 Mode::System 实例，并将其用于初始化 ManagerConfig。
    let system = Mode::System;

    //manager_config 是一个引用计数的可变单元，存储了管理器的配置信息
    let manager_config = Rc::new(RefCell::new(ManagerConfig::new(&system)));

    //初始化日志记录系统 log::init_log，设置日志级别、目标、文件路径、文件大小和文件数量。
    log::init_log(
        "sysmaster",
        Level::from_str(&manager_config.borrow().LogLevel).unwrap(),
        manager_config
            .borrow()
            .LogTarget
            .split(&[' ', '-'][..])
            .collect(),
        LOG_FILE_PATH,
        manager_config.borrow().LogFileSize,
        manager_config.borrow().LogFileNumber,
        false,
    );

    //输出一条信息日志，表明 sysmaster 正在系统模式下运行。
    log::info!("sysmaster running in system mode.");

    setup::mount_setup()?;

    rel::reli_dir_prepare()?;

    //获取自恢复功能的启用状态，并记录一条信息日志。
    let self_recovery_enable = rel::reli_debug_get_switch();
    log::info!("sysmaster self_recovery_enable: {}.", self_recovery_enable);


    //根据自恢复功能的启用状态初始化运行时环境 initialize_runtime。
    initialize_runtime(self_recovery_enable)?;

    let manager = Manager::new(system, Action::Run, manager_config);

    // enable clear 如果没有启用自恢复功能且没有反序列化参数，则清除恢复数据。
    if !self_recovery_enable && !args.deserialize {
        manager.debug_clear_restore();
        log::info!("debug: clear data restored.");
    }
```

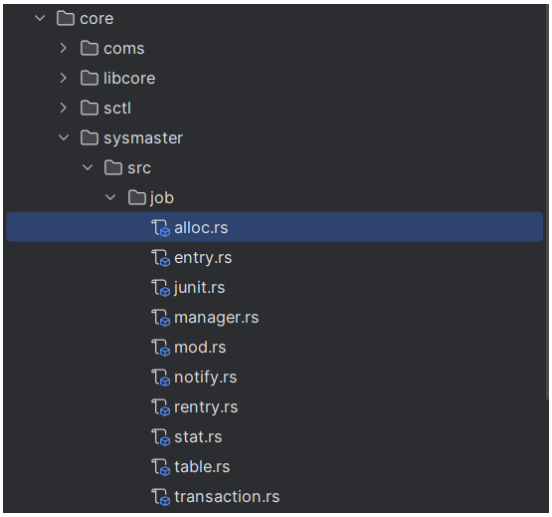
sysMaster-core-job

一、主要功能

job 模块是系统的核心部分，它负责整个作业的生命周期管理。这个模块能够创建作业，这些作业通常对应于对系统中服务的各种操作，比如启动、停止或者重启服务。它使用一个作业调度器来处理作业的执行顺序，这个调度器会考虑作业之间的依赖关系和触发条件，以确保作业按照正确的顺序执行。执行作业时，job 模块会监控作业的状态，包括成功、失败或者超时，并根据这些状态更新作业的进度。如果作业失败，模块会根据配置决定是否重试或者跳过。作业完成后，job 模块能够发送通知给系统其他组件，触发后续的操作或者事件处理。

此外，job 模块还负责作业的持久化，确保在系统重启后能够恢复作业的状态，以及在作业执行过程中能够从失败点恢复。它还收集作业执行的统计数据，为系统管理员提供了审计和监控作业执行历史的能力。在作业执行过程中，如果出现冲突或者依赖问题，job 模块会进行检测和解决，保证系统的稳定性。

二、主要组件



2.1 manager.rs

这个文件定义了 JobManager 结构体，它是作业管理的核心。JobManager 负责作业的生命周期管理，包括作业的创建、执行、完成和清理。它还处理作业的依赖关系和触发事件。

- 主要功能：

作业执行：exec 方法用于执行作业。

作业通知：notify 方法用于在作业完成后发送通知。
作业完成：try_finish 方法用于标记作业为完成状态。
作业移除：remove 方法用于从作业表中移除作业。

- 关键特性：

支持作业的异步执行。
管理作业的依赖性和触发条件。
提供作业执行的统计信息。

2.2 mod.rs

这个文件作为模块的索引，它重导出了模块中各个组件的公共接口。这使得其他模块可以通过 mod 文件访问 job 模块的功能，而不需要直接依赖于内部实现。

- 主要功能：
    - 提供模块内部结构体和枚举的公共访问接口。
    - 简化模块间依赖关系。

2.3 junit.rs

这个文件定义了 JobUnit 结构体，它表示与特定单元（Unit）相关的作业集合。JobUnit 负责管理单个单元的作业，包括作业的挂起、触发和完成。

- 主要功能：

作业挂起和触发：管理单元的挂起作业，并在条件满足时触发它们。
作业合并：合并冲突的作业，确保作业的一致性。
作业顺序：处理作业的执行顺序，确保依赖关系得到尊重。

- 关键特性：

支持作业的暂停和恢复。
管理作业的冲突和依赖关系。

2.4 entry.rs

这个文件定义了作业的基本数据结构，如 JobConf（作业配置）、JobInfo（作业信息）和 JobResult（作业结果）。这些结构体为作业的创建和执行提供了必要的数据。

- 主要功能：

定义作业的配置和状态。
提供作业执行的结果和统计信息。

2.5 alloc.rs

这个文件定义了 JobAlloc 结构体，它用于分配和管理作业 ID。JobAlloc 确保每个作业都有一个唯一的 ID，这对于作业的跟踪和管理至关重要。

- 主要功能：

分配唯一的作业 ID。
管理作业 ID 的分配和回收。

2.6 table.rs

这个文件定义了 JobTable 结构体，它是一个作业表，用于存储和管理系统中的所有作业。JobTable 提供了作业的插入、删除和查询操作。

- 主要功能：

存储和管理作业实体。
提供作业的插入、删除和查询接口。

2.7 transaction.rs

这个文件定义了作业事务处理的相关函数，包括作业的扩展、影响分析和验证。这些函数确保作业的创建和执行不会违反系统的一致性和完整性。

- 主要功能：

作业扩展：根据作业配置和系统状态扩展作业。
作业影响分析：分析作业对系统的影响。
作业验证：验证作业的合法性和一致性。

2.8 rentry.rs

这个文件定义了 JobRe 结构体，它负责作业的可靠性管理。JobRe 处理作业的持久化和恢复，确保系统在故障后能够正确恢复作业状态。

- 主要功能：

作业持久化：将作业状态持久化到存储中。
作业恢复：在系统恢复时重新加载作业状态。

2.9 notify.rs

这个文件定义了作业通知的相关函数，包括作业结果通知和作业事件通知。这些函数在作业完成后触发，用于通知其他系统组件或执行后续操作。

- 主要功能：

作业结果通知：在作业完成后发送结果通知。
作业事件通知：在作业触发特定事件时发送通知。

2.10 stat.rs

这个文件定义了 JobStat 结构体，它用于收集和统计作业的执行结果和操作次数。这些统计信息对于系统的性能分析和问题诊断非常有用。

- 主要功能：

收集作业执行结果。
统计作业操作次数。

这些文件共同构成了 job 模块，提供了作业的创建、执行、管理、通知和统计等功能。模块的设计注重事务性、模块化和可靠性，为系统的稳定运行提供了有力支持。通过对作业的精细管理，job 模块确保了系统作业的正确执行和高效调度。

三、具体实现

3.1 作业创建

当系统需要执行一个操作，如启动或停止服务时，它会通过 JobManager 创建一个新的作业。在 manager.rs 中，JobManager::new 函数初始化作业管理器，准备接收作业请求。

```rust
impl JobManager {
    pub(crate) fn new(
        eventr: &Rc<Events>,
        relir: &Rc<Reliability>,
        dbr: &Rc<UnitDb>,
        dmr: &Rc<DataManager>,
    ) -> JobManager {
        let jm = JobManager {
            event: Rc::clone(eventr),
            sub_name: String::from("JobManager"),
            data: Rc::new(JobManagerData::new(relir, dbr, eventr, dmr)),
        };
        jm.register(eventr, dbr);
        jm
    }
```

使用 JobAlloc 来为新作业分配一个唯一的 ID，并创建一个 Job 实例。这个实例包含了作业的所有必要信息，如类型、状态和相关联的单元（Unit）。

```rust
impl JobAlloc {
    pub(super) fn new(
        relir: &Rc<Reliability>,
        rentryr: &Rc<JobRe>,
        eventsr: &Rc<Events>,
        dmr: &Rc<DataManager>,
    ) -> JobAlloc {
        JobAlloc {
            reli: Rc::clone(relir),
            rentry: Rc::clone(rentryr),
            events: Rc::clone(eventsr),
            dm: Rc::clone(dmr),
            data: RefCell::new(JobAllocData::new()),
        }
    }

    pub(super) fn clear(&self) {
        self.data.borrow_mut().clear();
    }

    pub(super) fn alloc(&self, config: &JobConf) -> Rc<Job> {
        let unit = config.get_unit();
        let kind = config.get_kind();
        self.data.borrow_mut().alloc(
            &self.reli,
            &self.rentry,
            &self.events,
            &self.dm,
            Rc::clone(unit),
            kind,
        )
    }
}
```

3.2 作业调度：

JobTable 负责存储和管理所有作业实例。它根据作业的依赖关系和触发条件来调度作业。

例如，某些服务可能需要在其他服务启动后才能启动。

JobUnit 管理与特定单元相关的所有作业，确保作业按照正确的顺序执行。

3.3 作业执行:

当作业被调度为就绪状态时，JobManager 会调用作业的 run 方法来执行作业。作业的执行可能涉及调用底层的服务管理接口

```rust
pub(self) fn run(&self, unit: Option<&UnitX>) -> usize {
    let mut cnt: usize = 0;
    loop {
        // pop(JobTable.try_trigger()) + {record + action}(Job.run())
        // try to trigger something to run
        *self.text.borrow_mut() = None; // reset every time
        *self.running.borrow_mut() = true;
        let trigger_ret = self.jobs.try_trigger(unit);
        *self.running.borrow_mut() = false;

        if let Some((trigger_info, merge_trigger)) = trigger_ret {
            // something is triggered in this round
            let (lcnt, _) = cnt.overflowing_add(1); // ++
            cnt = lcnt;

            // update statistics
            self.stat.update_change(&(&None, &merge_trigger, &None));

            // try to finish it now in two case, and the case coming from unit has higher
            // case 1. the job has been finished synchronously in context, which is deriv
            // case 2. the trigger is ended(failed or over), which is derived from 'job'
            if let Some((unit, os, ns, flags)) = self.text.take() {
                // case 1: finish it
                self.do_try_finish(&unit, os, ns, flags);
                *self.text.borrow_mut() = None;
            }

            if let Some((t_jinfo, Some(tend_r))) = trigger_info {
                // case 2: remove it if it exists
                if self.jobs.get(t_jinfo.id).is_some() {
                    self.do_remove(&t_jinfo, tend_r, true);
                }
            }
```

作业执行过程中，Job 实例会更新其状态，如从 Wait 变为 Running。这些状态变化在 entry.rs 中定义的 Job 结构体中进行管理。

```rust
pub(crate) enum JobResult {
    Done,
    Cancelled,
    TimeOut,
    Failed,
    Dependency,
    Skipped,
    Invalid,
    Assert,
    UnSupported,
    Collected,
    Once,
    Merged,
}
```

3.4 作业监控和完成：

JobManager 监控作业的执行状态，捕获作业的成功、失败或超时等结果。在 manager.rs 中，JobManager::try_finish 方法用于标记作业为完成状态，并触发任何相关的后续操作。

```rust
#[test]
fn job_try_finish_async() {
    let (event, reli, db, unit_test1, _unit_test2) = prepare_unit_multi(None);
    let jm = JobManager::new(&event, &reli, &db, &Rc::new(DataManager::new()));
    let os = UnitActiveState::InActive;
    let ns = UnitActiveState::Active;
    let flags = UnitNotifyFlags::empty();

    let ret = jm.try_finish(&unit_test1, os, ns, flags);
    assert!(ret.is_ok());
}

#[test]
fn job_try_finish_sync() {
    let (event, reli, db, unit_test1, _unit_test2) = prepare_unit_multi(None);
    let jm = JobManager::new(&event, &reli, &db, &Rc::new(DataManager::new()));
    let os = UnitActiveState::InActive;
    let ns = UnitActiveState::Active;
    let flags = UnitNotifyFlags::empty();

    *jm.data.text.borrow_mut() = None; // reset every time
    *jm.data.running.borrow_mut() = true;
    let ret = jm.try_finish(&unit_test1, os, ns, flags);
    *jm.data.running.borrow_mut() = false;
    assert!(ret.is_ok());
    assert!(jm.data.text.borrow().is_some());
    let (u, o, n, f) = jm.data.text.take().unwrap();
    assert_eq!(u.id(), unit_test1.id());
    assert_eq!(o, os);
    assert_eq!(n, ns);
    assert_eq!(f, flags);
}
```

作业完成后，JobStat（定义在 stat.rs 中）收集作业的执行结果和统计信息，为系统管理员提供审计和监控的能力。

```rust
impl JobStat {
    pub(super) fn new() -> JobStat {
        JobStat {
            data: RefCell::new(JobStatData::new()),
        }
    }

    pub(super) fn clear(&self) {
        self.data.borrow_mut().clear();
    }

    #[allow(clippy::type_complexity)]
    pub(super) fn update_change(
        &self,
        change: &(&Option<Rc<Job>>, &Option<Rc<Job>>, &Option<Rc<Job>>),
    ) {
        self.data.borrow_mut().update_change(change)
    }

    #[allow(clippy::type_complexity)]
    pub(super) fn update_changes(&self, changes: &(&Vec<Rc<Job>>, &Vec<Rc<Job>>, &Vec<Rc<Job>>)) {
        self.data.borrow_mut().update_changes(changes)
    }

    pub(super) fn clear_cnt(&self) {
        self.data.borrow_mut().clear_cnt()
    }
}
```

3.5 作业通知:

根据作业的执行结果，JobManager 会发送通知给其他系统组件。这些通知可以通过事件、回调或其他机制进行，如 notify.rs 中定义的 job_notify_result 和 job_notify_event 函数。

```rust
pub(super) fn job_notify_result(
    db: &UnitDb,
    unit: Rc<UnitX>,
    atom: UnitRelationAtom,
    mode: JobMode,
) -> (Vec<JobConf>, JobMode) {
    let configs = match atom {
        UnitRelationAtom::UnitAtomOnSuccess | UnitRelationAtom::UnitAtomOnFailure => {
            notify_result_start(db, unit, atom)
        }
        _ => unreachable!("kind of notify is not supported."),
    };

    (configs, mode)
}

pub(super) fn job_notify_event(
    db: &UnitDb,
    config: &JobConf,
    mode_option: Option<JobMode>,
) -> Vec<(JobConf, JobMode)> {
    match config.get_kind() {
        JobKind::Start => notify_event_start(db, config, mode_option),
        JobKind::Stop => notify_event_stop(db, config, mode_option),
        JobKind::Reload => notify_event_reload(db, config, mode_option),
        _ => unreachable!("kind of notify is not supported."),
    }
}
```

3.6 作业持久化:

JobRe（定义在 rentry.rs 中）负责将作业状态持久化到存储系统中。这样，即使系统发生故障，作业状态也可以被恢复，确保系统的高可用性和一致性。

```rust
impl JobRe {
    pub(super) fn new(relir: &Rc<Reliability>) -> JobRe {
        let trigger = Rc::new(ReDb::new(relir, RELI_DB_HJOB_TRIGGER));
        let suspends = Rc::new(ReDb::new(relir, RELI_DB_HJOB_SUSPENDS));
        let rentry = JobRe { trigger, suspends };
        rentry.register(relir);
        rentry
    }

    pub(super) fn trigger_insert(&self, unit_id: &str, kind: JobKind, attr: &JobAttr) {
        assert!(job_is_basic_op(kind));
        let jt_data = JobReTrigData::new(kind, attr);
        self.trigger.insert(String::from(unit_id), jt_data);
    }

    pub(super) fn trigger_remove(&self, unit_id: &str) {
        self.trigger.remove(&unit_id.to_string());
    }

    pub(super) fn trigger_get(&self, unit_id: &str) -> Option<(JobKind, JobAttr)> {
        if let Some(jt_data) = self.trigger.get(&unit_id.to_string()) {
            Some((jt_data.kind, jt_data.attr))
        } else {
            None
        }
    }

    pub(super) fn trigger_keys(&self) -> Vec<String> {
        self.trigger.keys()
    }
}
```

3.7 作业事务处理：

在 transaction.rs 中定义的函数处理作业的扩展、影响分析和验证，确保作业的创建和执行不会违反系统的一致性和完整性。

比如，如果在作业执行过程中出现错误，或者作业验证失败，需要回滚已经执行的操作。job_trans_fallback 函数负责这个回滚过程。

```rust
pub(super) fn job_trans_fallback(
    jobs: &JobTable,
    db: &UnitDb,
    unit: &UnitX,
    run_kind: JobKind,
    f_result: JobResult,
) -> Vec<Rc<Job>> {
    let mut del_jobs = Vec::new();
    trans_fallback_body(jobs, db, unit, run_kind, f_result, &mut del_jobs);
    del_jobs
}
```

整个工作流程通过模块化的设计和组件间的协作，确保了作业的原子性、一致性、隔离性和持久性（ACID 属性），同时也支持作业的灵活性和可扩展性。通过这种方式，job 模块可以有效地管理和执行各种系统作业，确保系统的稳定性和可靠性。