Gestione degli errori

Tutte le computazioni possono fallire prematuramente impedendendo quindi che possa essere restituito il valore atteso o che vengano eseguiti tutti gli effetti collaterali richiesti. I fallimenti possono essere catalogati in due gruppi principali:

- malfunzionamenti **recuperarabili** : quelli che non hanno compromesso lo stato del programma ed è possibile attivare una strategia di recupero
- malfunzionamenti **non recuperabili** : quelli che causano un'alterazione impredicibile dello stato o che indicano la possibilità di procedere ulteriormente nella computazione Nei casi irrecuperabili occorre terminare il processo mentre negli altri occorre mettere in atto una strategia di ripristinao dello stato. Non è detto che il punto in cui si verifica il fallimento possegga abbastanza contesto per discriminare come comportarsi. Questo implica un aumento della complessità del codice (rami if/else, match, switch/case lungo i quali procedere). I linguaggi moderni hanno introdotto particolari costrutti a supporto della gestione delle eccezzioni.

Supporto sintattico alla gestione degli errori in Rust

Rust non utilizza il concetto di eccezione ma offre i tipi algebric **Result < T,E > e Option < T >** per esprimere gli esiti delle computazioni. Offre inoltre la macro **panic!(...)** per forzare l'interruzione del thread corretne producendo una descrizione testuale di quanto successo.

Pattern Resource Acquisition is Inialization

E' un pattern che serve a gestire il comportamento inatteso di fallimento : la funzione corrente si interrompe e viene causato un ritorno anticipato senza inizializzazione del valore di ritorno. Il ritorno comporta la contrazione dello stack con il conseguenete rilascio di tutte le variabili locali e l'esecuzione dei relativi distruttori. Questi ultimi possono essere usati per liberare risorse acquisite o disfare effetti collaterali avviati dal costruttore, contando sul fatto che verranno eseguiti sempre e comunque, anche in caso di eccezione.
Rust riprende il concetto di Raii, appoggiandosi su strutture che implementano il tratto Drop.

Gestione delle eccezioni in Rust

Rust offre una risposta funzionale al problema della modellazione degli errori basata sul tipo algebrico generico **Result < T,E >**. Modella l'unione di tutti i possibili risultati con successo T e tutti gli errori che possono verificarsi nell'esecuzione di una funzione

```
enum Result<T, E> {  Ok(T),  Err(E),}
fn read_file(name: &str)-> Result<String,io::Error> {
  let r1 = File::open(name);
  let mut file = match r1 {
    Err(why) => return Err(why),
    Ok(file) => file,
  };
  let mut s = String::new();
  let r2 = file.read_to_string(&mut s);
  match (r2) {
            Err(why) => Err(why),fn read_file(name: &str)->
Result<String,io::Error> {
            let r1 = File::open(name);
            let mut file = match r1 {
            Err(why) => return Err(why),
             Ok(file) => file,
  };
  let mut s = String::new();
  let r2 = file.read_to_string(&mut s);
  match (r2) {
    Err(why) => Err(why),
    Ok(_) => Ok(s),
  }}    Ok(_) => Ok(s),
  }
}
```

Elaborare i risultati

Result è una monade, incapsula un risultato ma non vi si può accedere. Bisogna fare del pattern matching. Result< T,E > mette a disposizione svariati metodi che consentono di accedere ai dati contenuti al suo interno

- I metodi **is_ok(&self)** e **is_err(&self)** permettono, rispettivamente, di determinare se l'esito di un'operazione ha avuto successo o meno
- I metodi **ok(self)** e **err(self)** consumano il risultato trasformandolo in un oggetto di tipo **Option< T > piuttosto che Option< E >**
- Il metodo **map(self, op: F)->Result< U,E >** applica la funzione al valore contenuto nel risultato, se questo è ok, altrimenti lascia l'errore invariato
- Il metodo **contains(&self, x: &U)** restituisce vero se il risultato è valido e contiene un valore che equivale all'argomento
- Il **metodo unwrap(self)->T** restituisce il valore contenuto, se è valido, ma invoca la macro panic!(...) se il risultato contiene un errore

Gestire gli errori

Se l'invocazione di una funzione restituisce un valore di tipo Result contenente un errore, occorre mettere in atto una strategia di gestione

- Terminare, in modo ordinato, il programma o Ritentare l'esecuzione, evitando di entrare in un loop infinito
- Registrare un messaggio nel log e propagare l'errore al chiamante
- Propagare tout court l'errore al chiamante, delegando ad esso la corrispondente responsabilità

Sebbene terminare il programma possa apparire facile (è sufficiente invocare la funzione **std::process::exit(code: i32) -> !)**, farlo in modo pulito può essere complesso. Occorre infatti garantire che non vengano lasciati oggetti persistenti (come file o altre risorse di sistema) in stati non coerenti. **Per questo motivo, Rust offre la macro panic!(...)**

Panic

In alcune situazioni, **lo stato del programma risulta così compromesso che non è pensabile eseguire azioni di ripristino**: questo spesso è dovuto alla presenza di errori logici nel programma stesso. In queste situazioni, Rust offre la **macro panic!(...)** che accetta argomenti simili a quelli offerti dalla macro println!(...) per formulare un messaggio di errore. L'effetto dell'invocazione di questa macro è la contrazione dello stack (come nel caso delle eccezioni C++), la distruzione (via .drop()) delle variabili che implementano il tratto Drop fino alla terminazione del thread corrente. **Se il thread in cui è stato invocato panic!(...) è il thread principale dell'applicazione, il processo termina con un codice di errore non nullo, altrimenti continua**.

Ignorare l'errore

In alcune situazioni, può capitare che l'errore che potenzialmente viene restituito da una funzione non possa di fatto succedere, a conseguenza di quanto si è appena verificato nel corso dell'esecuzione oppure che il programmatore assuma che non sia necessario mettere in atto una strategia di contenimento e gestione dell'errore, lasciando semplicemente terminare il processo. Il tipo Result< T,E > mette a disposizione i **metodi unwrap() e expect(...) che restituiscono il valore di tipo T, se presente ed invocano la macro panic!(...) in caso di errore**. Il metodo **expect(...) **permette di indicare una stringa che sarà usata al posto del messaggio standard generato da unwrap(), nel caso si sia verificato un errore.

Propagare l'errore

Nella maggior parte delle funzioni in cui si verifica un errore, non si sa quale strategia mettere in atto per ripristinare lo stato del programma. Si può facilmente ovviare a questo problema restituendo, a propria volta, un oggetto di tipo Result<T,E>. Questo porta, tuttavia, a costrutti alquanto complessi, con espressioni di controllo difficili da decodificare. **Per semplificare la sintassi, Rust offre l'operatore ? che può essere applicato a qualunque espressione che produce un valore di tipo Result< T,E >**.

- Se il valore risulta **essere Ok(v)**, l'operatore restituisce il valore v racchiuso nell'enum
- Se, invece, risulta essere **Err(e)**, la funzione corrente termina, ritornando il valore che incapsula l'errore

```
? come if result contains Error return error else tira fuori il valore buono
```

Perché questo comportamento possa funzionare, occorre che la funzione in cui è usato l'operatore ? ritorni un oggetto di tipo Result< U,E > e che il tipo dell'errore ritornato sia compatibile con l'errore E che proviene dalla funzione chiamata.

L'utilizzo dell'operatore ? permette di ottenere una sintassi molto più compatta, evidenziando il comportamento della funzione lungo il cammino principale e demandando al compilatore la scrittura delle clausole if / match necessarie a valutare il comportamento da adottare

```
//avanzamento vecchia funzione read file
fn read_file(name: &str) -> Result<String,io::Error> {
  let mut file = File::open(name)?;
  let mut s = String::new();
  file.read_to_string(&mut s)?;
  Ok(s)
}
```

Altri modi di esprimere il fallimento

In alcune situazioni, può essere sufficiente distinguere se la computazione ha prodotto il proprio risultato, oppure indicare che non è stato possibile completarla. (Non si vuole dire cosa ha generato l'errore), Per questi casi esiste il tipo **Option < T >** che racchiude due alternative:

- **Some< T >**, usata per descrivere il risultato atteso
- **None**, che indica l'assenza di risultato, senza descriverne le ragioni. **L'operatore ? può essere applicato anche al tipo Option< T >** a condizione che la funzione in cui viene adottato abbia tipo di ritorno Option< U >, con U qualsiasi<>.
- **Result< T,E > e Option< U >** sono correlati Result offre
- il metodo ok(self) che restituisce Option< T >, valorizzato con il dato in caso di successo (il dato risulta mosso da Result, che quindi diventa inaccessibile)
- metodo err(self) che restituisce Option< E >, valorizzato con l'errore in caso di fallimento (anche in questo caso, con movimento)

Propagare errori eterogenei

Quando una funzione produce diverse tipologie di errore, è necessario propagare i diversi errori così da poter specializzare le contromisure. Rust offre diversi modi per propagare errori eterogenei, la scelta spetta al programmatore sulla base delle sue esigenze. **La libreria standard mette a disposizione l'implementazione generica del tratto From, che converte qualsiasi valore che implementi il tratto Error in Box< dyn error::Error >.** Gli oggetti-tratto richiedono l'utilizzo di fat pointer e vtable con il conseguente costo in termini di memoria. Durante la conversione vengono perse le informazione sul tipo dell'errore. Si può risalire allo specifico errore tramite l'utilizzo del **downcast_ref()** a patto che si conosca l'implementazione della funzione che genera gli errori. La conversione può avvenire in maniera implicita attraverso l'utilizzo dell'operatore ?

```
impl<E: error::Error> From<E> for Box<dyn error::Error>;

//la funzione ora diventa così
fn sum_file(path: &Path) -> Result<i32, Box<dyn error::Error>> {
  let mut file = File::open(path) ? ;      // io::Error -> Box<dyn error::Error>
  let mut contents = String::new();
  file.read_to_string(&mut contents) ? ; // io::Error -> Box<dyn error::Error>
  let mut sum = 0;
  for line in contents.lines() {
    sum += line.parse::<i32>() ? ;       // ParseIntError -> Box<dyn error::Error>
  }
  Ok(sum)
}
// ci sono tre tipi di errori

fn handle_sum_file_errors(path: &Path) {
  match sum_file(path) {
    Ok(sum) => println!("sum is {}", sum),
    Err(err) => {
      if let Some(e) = err.downcast_ref::<io::Error>() {...}  //tratto io::Error
      else if let Some(e) = err.downcast_ref::<ParseIntError>() {...}  //tratto
ParseIntError
      else { unreachable!(); }  //non può capitare
    }
  }
}
```

Per propagare errori eterogenei senza forzare il sistema dei tipi è possibile implementare degli errori custom. **Tutti gli errori custom devono implementare il tratto Error e conseguentemente anche i tratti Debug e Display**. L'utilizzo di un enum permette di racchiudere i diversi tipi di errore da gestire successivamente tramite l'utilizzo del costrutto match. E' necessario implementare il tratto From per convertire i diversi errori nel tipo custom da propagare.

```
#[derive(Debug)]
enum SumFileError {
  Io(io::Error),
  Parse(ParseIntError),
}
impl From<io::Error> for SumFileError {
  fn from(err: io::Error) -> Self { SumFileError::Io(err) }
}
impl From<ParseIntError> for SumFileError {
  fn from(err: ParseIntError) -> Self { SumFileError::Parse(err) }
}
impl fmt::Display for SumFileError {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    match self {
      SumFileError::Io(err) => write!(f, "IO error: {}", err),
      SumFileError::Parse(err) => write!(f, "Parse error: {}", err),
    }
  }
}
impl error::Error for SumFileError {
  fn source(&self) -> Option<&(dyn error::Error + 'static)> {
    Some(match self {
      SumFileError::Io(err) => err,
      SumFileError::Parse(err) => err,
    })
  }
}
fn sum_file(path: &Path) -> Result<i32, SumFileError> {
  let mut file = File::open(path)?;
  let mut contents = String::new();
  file.read_to_string(&mut contents)?;
  let mut sum = 0;
  for line in contents.lines() {
    sum += line.parse::<i32>()?;
  }
  Ok(sum)
}
fn handle_sum_file_errors(path: &Path) {
  match sum_file(path) {
    Ok(sum) => println!("the sum is {}", sum),
    Err(SumFileError::Io(err)) => {...},
    Err(SumFileError::Parse(err)) => {...},
  }
}
```

Un grosso aiuto all'implementazione di tipi che implementano il tratto Error viene dal **crate thiserror**. Esso offre una implementazione della macro derive specializzata per il tratto Error. **Definendo un tipo (enum, struct, unit) preceduto dall'attributo #[derive(Error, Debug)] si ottiene l'implementazione automatica di questi due tratti**.

- L'attributo **#[error("Messaggio con formato")]** posta di fronte alle singole voci enumerative o all'intera struct genera l'implementazione del tratto Display.
- L'attributo **#[from]** posto di fronte ad un campo il cui tipo implementa il tratto Error genera l'implementazione del tratto From a partire dal tipo del campo Questo approccio è particolarmente adatto nella creazione di librerie, dove si vogliono rendere i possibili errori parte della definizione del contratto (API)

```
// in cargo.toml
[dependencies]
thiserror = "1.0"

#[derive(Error, Debug)]
enum SumFileError {

  #[error("IO error {0}")]
  Io(#[from] io::Error),

  #[error("Parse error {0}")]
  Parse(#[from] ParseIntError),
}
```

Il crate anyhow error

Il crate anyhow definisce l'oggetto-tratto anyhow::Error che semplifica la gestione idiomatica degli errori. Si può usare il tipo **anyhow::Result< T >** per incapsulare il valore di ritorno di una funzione che può fallire. Questo tipo offre un'implementazione automatica del tratto **From< T: Error >**, il che permette di utilizzare la notazione basata sull'operatore ? per propagare l'errore ritornato . Quando viene generato un errore è possibile aggiungere una descrizione che contestualizza ciò che è successo tramite i metodi **context(...) e with_context(...)**. Il messaggio di errore associato verrà sostituito dalla stringa indicata seguita dalla causa originale. Questo crate interopera correttamente con** thiserror** e risulta adatto nella scrittura di codice applicativo, piuttosto che di librerie. In questo caso, la semplicità di scrittura del codice domina rispetto al controllo dell'informazione contenuta nell'errore generato, tenuto conto che sarà interpretato da persone

```
fn sum_file(path: &Path) -> anyhow::Result<i32> {
  let mut file = File::open(path).with_context(|| format!("Missing path {}",
path)) ? ;
  let mut contents = String::new();
  file.read_to_string(&mut contents).context("File read error") ? ;
  let mut sum = 0;
  for line in contents.lines() {
    sum += line.parse::<i32>().with_context(|| format!("Not a number: {}", line))
? ;
  }
  Ok(sum)
}
fn handle_sum_file_errors(path: &Path) {
  match sum_file(path) {
    Ok(sum) => println!("sum is {}", sum),
    Err(err) => {
      if let Some(e) = err.downcast_ref::<io::Error>() {...}  //tratto io::Error
      else if let Some(e) = err.downcast_ref::<ParseIntError>() {...}  //tratto
ParseIntError
      else { unreachable!(); }  //non può capitare
    }
  }
}
```