04-26.md

1. Lezione del 26/04 - Gestione degli Errori

Quando parliamo di gestione degli errori, possiamo intendere due cose. La prima è la gestione puntuale del valore del dato, come ad esempio un controllo manuale dei valori di una variabile floating point per assicurarsi che non abbia valore infinito o `NaN`. La seconda invce è il meccanismo delle eccezioni, popolarizzato da linguaggi come Java che lanciano un eccezione utilizzando gli interrupt, e va da sé quindi che tale meccanismo non solo sia costoso ma richieda anche un certo support hardware.

A più alto livello distinguiamo due scenari, che ci dicono quale delle due scelte adottare, ovvero il caso in cui l'errore sia recuperabile e quello in cui non lo sia. Si dice errore irrecuperabile quando in seguito all'errore non sia possibile continuare con l'esecuzione.

Rust ad esempio avvolge anche la semplice lettura da tastiera attorno a un possibile errore. Il linguaggio spezza l'operazione in due parti, la lettura di un buffer e la successiva conversione di tale buffer. Questo tipo di errore è recuparbile in quanto su Rust non siamo chiamati a gestire l'errore su tale operazione. Potremmo aver notato che ad esempio l'operazione `println!` ritorna un `Result`, quindi potremmo sapere se quell'operazione è andata a buon finre o meno, ma non siamo tenuti a gestire nessuno dei due casi.

Puntualizziamo che la gestione dell'errore in un contesto concorrente deve essere eseguita in maniera puntuale: se un thread incappa in un errore che richiede il suo arresto, anche tutti i suoi thread paralleli andranno arrestati.

In Rust non abbiamo quindi il meccanismo delle eccezioni, in quanto per consistenza si gestiscono queste casistiche tramite i tipi `Result` ed `Option`. Per confronto, vediamo come il C++ implementa il meccanismo delle eccezioni:

```
try {
    // codice che puo' lanciare un'eccezione
} catch (ExceptionTypeOne e) {
    // codice che gestisce l'eccezione
} catch (ExceptionTypeTwo e) {
    // codice che gestisce l'eccezione
}
```

Questo tipo di codice non si usa per l'arresto del programma ma è solitamente usato per il recupero dell'errore e quindi la continuazione dell'esecuzione. Queste operazioni richiedono una procedura molto costosa detta `stack unwinding`, in quanto devo retrocedere nelle chiamate a funzioni per trovare la fonte dell'errore, eseguire una disamina sul tipo di eccezione che abbiamo incontrato e saltare alla sua gestione specifica.

Il tipo `Result` come sappiamo utilizza due tipi generici, il tipo `T` che è il nostro tipo di dato ed il tipo `E` che invece è il tipo di errore. Quest'ultimo gioca qui il ruolo di discriminante dell'errore un po' come nel pezzo di codice sopra fa il tipo di eccezione. È molto comodo quindi eseguire il recupero dell'errore grazie alle istruzioni di `match`:

```
let r1 = File::open("file.txt");
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)
}
```

Tuttavia questo ci porta a un problema stilistico. Un linguaggio come Rust che si pone tra gli obiettivi quello di essere più conciso dei suoi predecessori si ritrova ad avere `match` multiple, intese sia come annidate che a cascata per fare un controllo d'errore, che in un applicativo reale deve necessariamente essere presente a differenza degli esempi didattici che possiamo scrivere noi.

Un primo aiuto in tal senso ci è fornito dall'operatore `?`. Esso indica che un'operazione che ha come valore di ritorno una `Result`, ritorna quel valore in caso di errore. Ricordando che anche `println!()` ha come valore di ritorno `Result` e che l'apertura di un file puo' risultare in un errore, la funzione:

```
fn funzione() -> Result<(), Error> {
    let res = println!("Apro il file");
    if res.is_err() {
        return res;
    }

    let file = File::open("file.txt");

    if file.is_err() {
        return file;
    }

    let res = println!("Ho aperto il file");
    if res.is_err() {
        return res;
    }

    return Ok(());
}
```

Puo' essere riscritta con tale operatore come:

```
fn funzione() -> Result<(), Error> {
    println!("Apro il file")?;
    let file = File::open("file.txt")?;
    println!("Ho aperto il file")?;
}
```

Indubbiamente il codice è molto più snello e fruibile.