```
cargo run --example wordcount
```

Wordcount 1.0

Usando la librería **Rayon**

```
let start = Instant::now();
```

Guardamos el momento de inicio del programa para medir cuánto tarda el proceso de conteo de las palabras.

```
let result = read_dir(concat!(env!("CARGO_MANIFEST_DIR"),
"/data")).unwrap()
```

Para leer los archivos dentro del directorio data.

```
.map(|d| d.unwrap().path())
.flat_map(|path| {
      let file = File::open(path);
      let reader = BufReader::new(file.unwrap());
      reader.lines()
})
```

map: convierte cada entrada del directorio en una ruta de archivo
flat_map: abre los archivos y crea un buffer para leer las líneas de los archivos.
reader: lee las líneas de cada archivo y devuelve un iterador de cadenas.

```
.map(|l| {
      let line = l.unwrap();
      let words = line.split(' ');
      thread::sleep(Duration::from_millis(100));
      let mut counts = HashMap::new();
      words.for_each(|w| *counts.entry(w.to_string()).or_insert(0) +=
1);
      counts
})
```

Divide cada línea en palabras, y en un Hashmap va acumulando las frecuencias de las palabras en cada línea.

```
.fold(HashMap::new(), |mut acc, words| {
      words.iter().for_each(|(k, v)| *acc.entry(k.clone()).or_insert(0)
+= v);
      acc
});
```

Con fold se combinan los resultados de las palabras contadas. La acumulación se hace en paralelo con la librería rayon.

Código completo

```
use std::collections::HashMap;
use std::{env, thread};
use std::fs::{File, read_dir};
use std::io::{BufRead, BufReader};

use rayon::prelude::{IntoParallelRefIterator, ParallelBridge,
ParallelIterator};
use std::path::PathBuf;
use std::time::{Instant, Duration};

fn main() {

   let start = Instant::now();
   let result = read_dir(concat!(env!("CARGO_MANIFEST_DIR"),
"/data")).unwrap()
       .map(|d| d.unwrap().path())
       .flat_map(|path| {
           let file = File::open(path);
           let reader = BufReader::new(file.unwrap());
           reader.lines()
       })
       .map(|l| {
           let line = l.unwrap();
           let words = line.split(' ');
           thread::sleep(Duration::from_millis(100));
           let mut counts = HashMap::new();
           words.for_each(|w| *counts.entry(w.to_string()).or_insert(0) +=
1);
           counts
       })
       .fold(HashMap::new(), |mut acc, words| {
           words.iter().for_each(|(k, v)| *acc.entry(k.clone()).or_insert(0)
```

```
+= v);
            acc
        });
    println!("{:?}", start.elapsed());
    println!("{:?}", result);
}
```

Wordcount 2.0

```
let result = read_dir(concat!(env!("CARGO_MANIFEST_DIR"),
"/data")).unwrap()
        .flatten()
        .map(|d| d.path())
        .collect::<Vec<PathBuf>>()
```

flatten(): read_dir devuelve un Result<DirEntry, std::io::Error>. Usando flatten(), el código "desenvuelve" el Result y solamente pasa las rutas válidas, descartando los errores.

```
.par_iter()
.flat_map(|path| {
        let file = File::open(path);
        let reader = BufReader::new(file.unwrap());
        reader.lines().par_bridge()
})
```

par_iter(): para que los archivos sean abiertos en paralelo.
par_bridge(): para que la iteración de las líneas sea paralela, es más eficiente que de forma secuencial.

```
.reduce(|| HashMap::new(), |mut acc, words| {
        words.iter().for_each(|(k, v)| *acc.entry(k.clone()).or_insert(0)
+= v);
        acc
});
```

reduce: se usa de forma más explícita para combinar los resultados de los hilos en paralelo. Es más eficiente en la forma en que se distribuyen las tareas de combinación entre los hilos.

Código completo

```
use std::collections::HashMap;
use std::{env, thread};
use std::fs::{File, read_dir};
use std::io::{BufRead, BufReader};

use rayon::prelude::{IntoParallelRefIterator, ParallelBridge,
ParallelIterator};
use std::path::PathBuf;
use std::time::{Instant, Duration};

fn main() {

   let start = Instant::now();
   let result = read_dir(concat!(env!("CARGO_MANIFEST_DIR"),
"/data")).unwrap()
       .flatten()
       .map(|d| d.path())
       .collect::<Vec<PathBuf>>()
       .par_iter()
       .flat_map(|path| {
           let file = File::open(path);
           let reader = BufReader::new(file.unwrap());
           reader.lines().par_bridge()
       })
       .map(|l| {
           let line = l.unwrap();
           let words = line.split(' ');
           thread::sleep(Duration::from_millis(100));
           let mut counts = HashMap::new();
           words.for_each(|w| *counts.entry(w.to_string()).or_insert(0)
+= 1);
           counts
       })
       .reduce(|| HashMap::new(), |mut acc, words| {
           words.iter().for_each(|(k, v)|
*acc.entry(k.clone()).or_insert(0) += v);
           acc
       });
   println!("{:?}", start.elapsed());
   println!("{:?}", result);
}
```