test

The tracking issue for this feature is: None.

The internals of the `test` crate are unstable, behind the `test` flag. The most widely used part of the `test` crate are benchmark tests, which can test the performance of your code. Let's make our `src/lib.rs` look like this (comments elided):

```
#![feature(test)]

extern crate test;

pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;
    use test::Bencher;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

Note the `test` feature gate, which enables this unstable feature.

We've imported the `test` crate, which contains our benchmarking support. We have a new function as well, with the `bench` attribute. Unlike regular tests, which take no arguments, benchmark tests take a `&mut Bencher`. This `Bencher` provides an `iter` method, which takes a closure. This closure contains the code we'd like to benchmark.

We can run benchmark tests with `cargo bench`:

```
$ cargo bench
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/release/adder-91b3e234d4ed382a
```

```
running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:         1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured
```

Our non-benchmark test was ignored. You may have noticed that `cargo bench` takes a bit longer than `cargo test`. This is because Rust runs our benchmark a number of times, and then takes the average. Because we're doing so little work in this example, we have a `1 ns/iter (+/- 0)`, but this would show the variance if there was one.

Advice on writing benchmarks:

- Move setup code outside the `iter` loop; only put the part you want to measure inside
- Make the code do "the same thing" on each iteration; do not accumulate or change state
- Make the outer function idempotent too; the benchmark runner is likely to run it many times
- Make the inner `iter` loop short and fast so benchmark runs are fast and the calibrator can adjust the run-length at fine resolution
- Make the code in the `iter` loop do something simple, to assist in pinpointing performance improvements (or regressions)

Gotcha: optimizations

There's another tricky part to writing benchmarks: benchmarks compiled with optimizations activated can be dramatically changed by the optimizer so that the benchmark is no longer benchmarking what one expects. For example, the compiler might recognize that some calculation has no external effects and remove it entirely.

```
#![feature(test)]

extern crate test;
use test::Bencher;

#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        (0..1000).fold(0, |old, new| old ^ new);
    });
}
```

gives the following results

```
running 1 test
test bench_xor_1000_ints ... bench:         0 ns/iter (+/- 0)
```

```
test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

The benchmarking runner offers two ways to avoid this. Either, the closure that the `iter` method receives can return an arbitrary value which forces the optimizer to consider the result used and ensures it cannot remove the computation entirely. This could be done for the example above by adjusting the `b.iter` call to

```
# struct X;
# impl X { fn iter<T, F>(&self, _: F) where F: FnMut() -> T {} } let b = X;
b.iter(|| {
    // Note lack of `;` (could also use an explicit `return`).
    (0..1000).fold(0, |old, new| old ^ new)
});
```

Or, the other option is to call the generic `test::black_box` function, which is an opaque "black box" to the optimizer and so forces it to consider any argument as used.

```
#![feature(test)]

extern crate test;

# fn main() {
# struct X;
# impl X { fn iter<T, F>(&self, _: F) where F: FnMut() -> T {} } let b = X;
b.iter(|| {
    let n = test::black_box(1000);

    (0..n).fold(0, |a, b| a ^ b)
})
# }
```

Neither of these read or modify the value, and are very cheap for small values. Larger values can be passed indirectly to reduce overhead (e.g. `black_box(&huge_struct)`).

Performing either of the above changes gives the following benchmarking results

```
running 1 test
test bench_xor_1000_ints ... bench:       131 ns/iter (+/- 3)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

However, the optimizer can still modify a testcase in an undesirable manner even when using either of the above.