```
MyActor { }.start()

Context::new().run(self));

struct Context<A>
  where A: Actor<Context = Context<A>>
{ parts: ContextParts<A>
  mb : Option<Mailbox<A>>
}

let mb = Mailbox::default()          ①

Context { parts = ContextParts::new( mb.sender_producer() )     ③ ②
          mb = Some(mb)
}

let fut = Self.into_future(Act)      ④
let Addr = fut.address();            ⑤
Arbiter::spawn(fut));                ⑥
Addr.
```

① Mailbox : Mailbox::default()

```
struct Mailbox<A>
  { msgs: AddressReceiver<A>
  }
  impl default
     let (_, rx) = channel::channel(DEFAULT_CAPACITY)
     Mailbox{msgs: rx }
```

② mb. = mb.sender_producer()
⇓
self.msgs.sender_producer()
⇓
? 1-3

④. 1-1 , 1-2 , 1-3 实际上调到 channel 的部分
看不是很懂

② ContextParts Impl 实际上是 ContextImpl 中的东西

```
struct ContextParts<A>
  where A : Actor.
     A::Context : AsyncContext<A>.
  { Addr :
    flags:
    wait:
  } items:
    handles:
```

一堆 有点怪怪的东西

实际上这俩和 ④ ⑥ 有关系, future 的 impl.

④
Self.into_future(act) ⑤
let Addr = fut.address(); ⑥
Arbiter::spawn(fut);
addr

④ into_future(act)
Actor的一切实际是都是由future生成来执行的. 因此需要将Actix的Context转换为future.

let mb = self.mb.take().unwrap();
Contextfut::new(self, act, mb)   Contextfut 才真正被执行的 future.

ContextImpl 实际完成有 [illegible] [illegible]. 会被其转换的Contextfut

<A, C>.
Contextfut(
  where C: AsyncContextParts<A>,
        A: Actor<Context = C>,   大部分的操作都是在这个结构下完成的.

{ ctx: C,
  act: A,
  mailbox: Mailbox<A>,
  wait: SmallVec<[ActorWaitItem<A>; 2]>,
  items: SmallVec<[Item<A>; 3]>,

☆ 难.

这个实现的 Actix 较复杂和难懂
大多数对于 ctx的操作实际是直接作用于 [illegible]

cache_future —— ① ContextParts 中的 handles 中 Push spawn handle. why it work.
notify —— spawn
notify_later

{ let handle = self.handles[0].next(); => spawnhandle 实际上就是个递增的数字
  self.handles[0] = handle;
  let fut = Box::new(fut);
  self.items.push((handle, fut));
  handle
}

实际上 equal
self.handle[0] = Self.handle[0].next(),
Self.items.push(Self.handle[0].clone())
将 handles 中的 O(1) 替换为下个 handle. 并将其 放到 self.items 中
(new 出来的, 所以是 [illegible])

ContextFut future.

当我们在 handle 一个 Message 中会频繁对 ctx 做出修改

当我们在 handle 一个 Message 中会频繁对 ctx 做出修改，比如说 spawn、notify、notify-later 这些，实际上就会去更新 handles 和 Items 这两个值。

每次 poll 顶层上做的事情很简单

① 检查是不是要取消某些被 spawn 进来的 spawnhandle,

② 等待直到要 wait 的 fut 执行完

③ 依次 poll 要执行的 task. 每 poll 一次都要再去检查是不是要取消，要不要等待

④ 检查要不要关闭这个 Actor.