1. 查询

查询可以让你访问实体的组件。使用 Query 系统参数，你可以指定要访问的数据，并可选择额外的过滤器。

你在 Query 中输入的类型，作为你想要访问的实体的“规范”。查询将匹配符合你规范的 ECS 世界中的哪些实体。然后，你可以以不同的方式使用查询，从这些实体中访问相关的数据。

2. Query Data

Query 的第一个类型参数是你想要访问的数据。使用 & 进行共享/只读访问，使用 &mut 进行独占/可变访问。

```
fn level_info(levels: Query<&Level>)

fn add_health(mut health: Query<&mut Health>)
```

如果组件不是必须的（有没有这个组件都行的实体），请用 Option 。如果你想要多个组件，请将它们放在一个元组中。

```
fn health_info_my_player_add_marker(health: Query<(&Health,
Option<&MyPlayer>)>)
```

如果你想知道你正在访问的实体的 ID，你可以在 Query 中加入 Entity 类型。如果你需要稍后对这些实体执行特定操作，这将非常有用。

```
fn position_entity_info(positions: Query<(Entity,
&Position)>)
```

3. 迭代

最常见的操作是迭代 Query ，以访问每个匹配实体的组件值。Query 可以通过调用 iter() / iter_mut() 方法转换为迭代器，这样就可以调用迭代器适配器了。

```
fn level_info(levels: Query<&Level>) {
    levels.iter().for_each(|level| {
        info!("level: {level:?}");
    });
}

fn add_health(mut health: Query<&mut Health>) {
    health.iter_mut().for_each(|mut health| {
        health.0 += 1;
    });
}
```

4. 访问特定实体

要从一个特定的实体访问组件，你需要知道实体 ID ：

```
fn level_info_by_parent(parents: Query<&Parent>, levels:
Query<&Level>) {
    let parent = parents.single().get();
    if let Ok(level) = levels.get(parent) {
        info!("parent level: {level:?}");
    }
}
```

如果你想要一次访问多个实体的数据，可以使用 many() / many_mut() （在错误时 panic）或 get_many() / get_many_mut() （返回 Result） 或 iter_many() / iter_many_mut() （返回迭代器）。这些方法请确保所有的 Entity 与查询匹配，否则将产生错误。

```
fn health_info_by_children(children: Query<&Children>,
health: Query<&Health>) {
    let Ok(children) = children.get_single() else {
        return;
    };
    let mut children = children.iter();
    let entity_1 = *children.next().unwrap();
    let entity_2 = *children.next().unwrap();
    let Ok([health_1, health_2]) =
health.get_many([entity_1, entity_2]) else {
        return;
    };
    println!("many child health_1: {health_1:?}, health_2:
{health_2:?}");

    for health in health.iter_many(children) {
        info!("child health: {health:?}");
    }
}
```

5. 独特实体

如果你知道应该只存在一个匹配的实体，你可以使用 single() / single_mut() （在错误时 panic）或 get_single() / get_single_mut() （返回 Result）。这些方法确保存在恰好一个候选实体可以匹配你的查询，否则将产生错误。

```
fn level_info_by_parent(parents: Query<&Parent>, levels:
Query<&Level>) {
    let parent = parents.single().get();
    if let Ok(level) = levels.get(parent) {
        info!("parent level: {level:?}");
    }
}

fn health_info_by_children(children: Query<&Children>,
health: Query<&Health>) {
    let Ok(children) = children.get_single() else {
        return;
    };
    for health in health.iter_many(children) {
        info!("child health: {health:?}");
    }
}
```

6. 组合

如果你想遍历 N 个实体的所有可能组合，Bevy 也提供了一个方法。如果实体很多，这可能会变得非常慢！

```
fn level_combinations(levels: Query<&Level>) {
    levels.iter_combinations().for_each(|[level1, level2]| {
        info!("level 1: {level1:?}, level 2: {level2:?}");
    });
}
```

7. Query Filter

添加查询过滤器以缩小从查询中获取的实体范围。

这是通过使用 Query 类型的第二个（可选）泛型类型参数来完成的。

注意查询的语法：首先指定你想要访问的数据（使用元组访问多个内容），然后添加任何额外过滤条件（也可以用元组添加多个）。

使用 With 来过滤有某个组件的实体，用 Without 来过滤没有某个组件的实体。

```
fn my_player_health(health: Query<&Health, With<MyPlayer>>)
{
    health.iter().for_each(|health| {
        info!("my player health: {health:?}");
    })
}

fn without_my_player_health(health: Query<&Health,
Without<MyPlayer>>) {
    health.iter().for_each(|health| {
        info!("without my player health: {health:?}");
    });
}
```

这在你实际上不关心这些组件内部存储的数据时很有用，但你想确保你的查询只查找具有（或不具有）它们的实体。如果你想要数据，那么将组件放在查询的第一部分，而不是使用过滤器。

可以组合多个过滤器：

- 在一个元组中 （与逻辑）
- 用 Or 包装来检测满足其中任一个 （或逻辑）

```
fn my_player_and_player_level(levels: Query<&Level,
(With<MyPlayer>, With<Player>)>) {
    levels.iter().for_each(|level| {
        info!("my player level: {level:?}");
    })
}

fn my_player_or_player_level(levels: Query<&Level,
Or<(With<MyPlayer>, With<Player>)>>) {
    levels.iter().for_each(|level| {
        info!("player level: {level:?}");
    });
}
```