1. 变化检测

Bevy 允许你轻松检测数据何时发生变化。你可以利用这一点来响应变化执行操作。

主要用例之一是优化——避免不必要的工作，仅在相关数据发生变化时才执行。另一个用例是在更改时触发特殊操作，如配置或将数据发送到某处。

2. 组件

2.1 过滤

你可以创建一个查询，只包含特定组件被修改的实体。

使用查询过滤器：

- Added<T>：检测新组件实例
  - 如果组件被添加到现有实体
  - 如果带有组件的新实体被生成
- Changed<T>：检测已更改的组件实例
  - 当组件被修改时触发
  - 如果组件是新添加的（如 Added），也会触发

```
fn added_info(components: Query<&MyComponent,
Added<MyComponent>>) {
    for component in &components {
        info!("Added MyComponent: {}", component.0);
    }
}

fn changed_info(components: Query<&MyComponent,
Changed<MyComponent>>) {
    for component in &components {
        info!("Changed MyComponent: {}", component.0);
    }
}
```

2.2 检查

如果你想像往常一样访问所有实体，而不管它们是否被修改，并且想知道这些组件的修改情况，你可以使用特殊的 Ref<T> 查询参数代替 & 进行不可变访问。

对于可变访问，变更检测方法始终可用（因为 Bevy 查询实际上返回一个特殊的 Mut<T> 类型，每当你在查询中使用 &mut 时）。

```
fn print_info(components: Query<Ref<MyComponent>>) {
    for component in &components {
        info!("print info: {}", component.0);

        if component.is_added() {
            info!("print info added: {}", component.0);
        }

        if component.is_changed() {
            info!("print info changed: {}", component.0);
        }
    }
}
```

3. 资源

对于资源，变更检测通过 Res<T>/ResMut<T> 系统参数上的方法提供。

```
fn print_resource_info(resource: Res<MyResource>) {
    info!("print resource info: {}", resource.0);

    if resource.is_added() {
        info!("print resource info added: {}",
resource.0);
    }

    if resource.is_changed() {
        info!("print resource info changed: {}",
resource.0);
    }
}
```

4. 检测到什么?

变更检测由 DerefMut 触发。仅通过可变查询访问组件，或通过 ResMut 访问资源，而不实际执行 &mut 访问，不会触发它。这使得变更检测非常准确。

注意：如果你调用一个接受 &mut T（可变借用）的 Rust 函数，即使该函数实际上没有进行任何修改，也会触发变更检测。

此外，当你修改组件时，Bevy 不会检查新值是否与旧值实际不同。它总是会触发变更检测。如果你想避免这种情况，只需自己检查。

变更检测在每个系统的粒度上工作，并且是可靠的。系统只会检测到它之前未见过的更改（更改发生在上次运行之后）。

5. 可能的陷阱

注意帧延迟/1 帧滞后。如果 Bevy 在更改系统之前运行检测系统，检测系统将在下次运行时看到更改，通常是在下一帧更新时。

如果你需要确保更改立即/在同一帧内处理，可以使用显式系统排序。