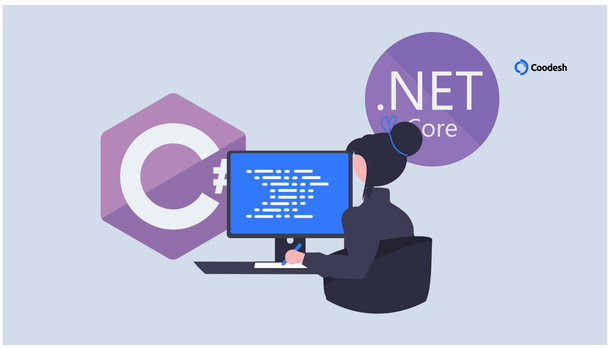
C#语言中 null 检查的最佳实践

作者：微信公众号：【架构师老卢】 1-31 10:51 131



概述：在_C# 应用程序，空引用异常通常是问题和运行时失败的原因。适当的 null 检查对于保护代码免受此类问题的影响是必要的。本文将介绍在 C# 中执行 null 检查的多种方法，介绍推荐的方法，并提供示例来演示如何使用每种方法。传统空检查：检查 null 的最直接方法是通过显式比较。例如：if (variable != null) {　// Do something with variable }使用运算符检查 null 值。 is not null 运算符是检查 null 值的安全方法，即使 not 相等运算符 （！ = null） 已被重写也是如此。is not nullif (

在_C# 应用程序，空引用异常通常是问题和运行时失败的原因。适当的 null 检查对于保护代码免受此类问题的影响是必要的。本文将介绍在 C# 中执行 null 检查的多种方法，介绍推荐的方法，并提供示例来演示如何使用每种方法。

1. **传统空检查**：

检查 null 的最直接方法是通过显式比较。例如：

```
if (variable != null)
{
    // Do something with variable
}
```

使用运算符检查 null 值。 is not null 运算符是检查 null 值的安全方法，即使 not 相等运算符 （！ = null） 已被重写也是如此。is not null

```
if (variable is not null)
{
    // Do something with variable
}
```

如果要编写需要与旧版本的 C# 兼容的代码，则应使用 ，因为直到 C# 9 才引入运算符。!= nullis not null

以下是一些何时使用和 ： is not null!= null

// Use `is not null` if you are working with code that you do not control.

```
object obj = GetObjectFromSomewhere();
if (obj is not null)
{
  // Do something with the object.
}

// Use \`!= null\` if you are working with code that you control, and you know that the equality operator has not been overloaded.

string name = GetName();
if (name != null)
{
  // Do something with the name.
}



// Use \`!= null\` if you need to be compatible with older versions of C#.
int? number = GetNumber();
if (number != null)
{
  // Do something with the number.
}
```

虽然这种方法简单且广泛使用，但它可能容易出错，尤其是在大型代码库中，开发人员可能会忘记包含 null 检查，从而导致意外崩溃。

2. **条件访问运算符 （? .）：**

条件访问运算符在 C# 6.0 中引入，是一个功能强大的工具，可以帮助你编写更简洁、更可靠的代码。通过使用条件访问运算符，可以避免为访问的每个引用类型编写显式 null 检查。例如：

```
// Access a member of a class.
Person person = GetPerson();
string name = person?.Name;

// Access a member of a struct.
DateTime date = GetDate();
int day = date?.Day;

// Access a member of an interface.
IEnumerable<int> numbers = GetNumbers();
int firstNumber = numbers?.First();

// Access a member of a nullable reference type.
int? number = GetNumber();
int value = number?.Value ?? -1;
```

下面是使用条件访问运算符的一些好处：Here are some of the benefits of using the conditional access operator:

它使您的代码更加简洁和可读。

它有助于防止 null 引用异常。

它使您的代码更加健壮和可靠。

3. **空合并算子 （? ? ）**：

C# 中的 null 合并运算符 （? ? ） 是一个二进制运算符，如果它不为 null，则返回其左侧操作数，否则返回其右侧操作数。

null 合并运算符的语法如下：

```
int result = leftOperand ?? rightOperand;
```

其中 和 是任意类型的表达式。leftOperandrightOperand

如果计算结果为 null，则 null 合并运算符将返回 .否则，null 合并运算符将返回 .leftOperandrightOperandleftOperand

null 合并运算符可用于为表达式提供默认值，即使表达式的计算结果为 null 也是如此。这对于避免空引用异常和编写更简洁易读的代码非常有用。

以下是如何使用 null 合并运算符的一些示例：

```
// Provide a default value for a variable.
string name = person?.Name ?? "Unknown";

// Get the first element of a collection, or return null if the collection is empty.
int firstNumber = numbers?.First() ?? 0;

// Get the length of a string, or return 0 if the string is null.
int length = string?.Length ?? 0;
```

4. 保护条款：

保护子句涉及在方法开始时检查 null，并在满足条件时提前退出。这种方法可以增强代码的可读性和可维护性。例：

```
public void SomeMethod(string variable)
{
    if (variable == null)
    {
        throw new ArgumentNullException(nameof(variable));
    }
    // Rest of the method logic
}

```

保护子句通过在方法的入口点捕获 null 引用来防止下游问题特别有用。

首选方式：保护子句

在这些方法中，guard 子句通常被认为是在 C# 中执行 null 检查的首选方法。原因如下：

1. **早期检测**：

保护子句在方法的最早时间点捕获 null 引用，从而防止进一步执行并降低下游问题的可能性。

2. **代码可读性**：

在 guard 子句中显式引发异常可使代码更具可读性。它清楚地表明，对于给定的上下文，null 值是不可接受的。

3.**一致性**：

保护子句强制采用一致的方法跨方法进行 null 检查，确保开发人员遵守标准化做法。

空检查的重要性：

执行可靠的 null 检查至关重要，原因如下：

1. **防止 Null 引用异常**：

空引用异常可能导致应用程序崩溃和不可预知的行为。可靠的 null 检查通过及早捕获 null 引用来帮助防止这些问题。

2. **增强代码可靠性**：

执行良好的 null 检查有助于提高代码库的整体可靠性，降低出现错误的可能性并提高应用程序的健壮性。

3. **提高可维护性**：

包含显式 null 检查的代码更易于维护。它可以帮助开发人员快速了解对变量值的期望，从而更轻松地进行未来的修改和调试。

尽管还有其他方法，但最好的方法是采用保护子句，因为它们具有一致性、可读性和早期检测能力。开发人员可以通过采用这些最佳实践来使其应用程序更能抵抗空引用异常，从而生成更可靠和可预测的软件系统。