در زبان برنامه‌نویسی **گولنگ (Go)**، تست‌نویسی به دلیل فلسفه مینیمالیستی و کارایی این زبان از اهمیت زیادی برخوردار است.

**اهمیت تست در گولنگ**

1. **کیفیت کد بالا:** با نوشتن تست‌ها می‌توان مطمئن شد که کدها به درستی کار می‌کنند و تغییرات بعدی منجر به بروز خطا نمی‌شوند.
2. **یافتن باگ‌ها در مراحل اولیه:** تست‌ها به شما کمک می‌کنند مشکلات احتمالی را پیش از انتشار کد شناسایی کنید.
3. **مستندات غیررسمی:** تست‌ها می‌توانند به عنوان نوعی مستندات برای کد عمل کنند و نشان دهند که هر بخش از برنامه چه رفتاری باید داشته باشد.
4. **حفظ قابلیت توسعه:** تست‌ها به شما اجازه می‌دهند بدون نگرانی از خراب شدن بخش‌های دیگر، کد را تغییر دهید یا گسترش دهید.

**توسعه مبتنی بر تست (TDD)**

**TDD** مخفف **Test-Driven Development** یا **توسعه مبتنی بر تست** است. این روش شامل سه مرحله اصلی است:

1. **نوشتن تست قبل از کد:** ابتدا تستی برای ویژگی یا قابلیت مورد نظر نوشته می‌شود که در ابتدای کار با شکست مواجه خواهد شد.
2. **نوشتن کد برای عبور از تست:** به حداقل کدی که باعث می‌شود تست پاس شود، بسنده می‌شود.
3. **بهبود کد (Refactor):** کدی که برای عبور از تست نوشته شده، بهینه و تمیز می‌شود.

مزایای TDD:

* تضمین می‌کند که کد با نیازمندی‌ها هماهنگ است.
* منجر به تولید کد تمیز و ماژولار می‌شود.
* کاهش هزینه‌ی باگ‌ها در مراحل بعدی.

### ****انواع تست‌ها در گولنگ****

1. **تست واحد (Unit Test):**  
   برای تست کردن کوچک‌ترین بخش‌های برنامه، مانند توابع و متدها.
2. **تست یکپارچگی (Integration Test):**  
   برای بررسی تعامل و هماهنگی میان چند بخش یا ماژول.
3. **تست سیستمی (System Test):**  
   برای تست کردن کل سیستم به عنوان یک واحد کامل.
4. **تست پذیرش (Acceptance Test):**  
   برای بررسی اینکه آیا برنامه نیازهای کاربر یا مشتری را برآورده می‌کند یا خیر.
5. **تست Benchmarking:**  
   گولنگ امکان نوشتن تست‌های بنچمارک برای ارزیابی عملکرد کد را فراهم می‌کند.

### ****بخش‌های مختلف یک تست در گولنگ****

تست‌ها در گولنگ به کمک بسته استاندارد **testing** نوشته می‌شوند. ساختار یک تست معمولاً شامل بخش‌های زیر است:

1. **آماده‌سازی (Setup):**  
   در این بخش، داده‌ها یا پیش‌نیازهای لازم برای اجرای تست فراهم می‌شوند.
2. **اجرای کد (Execution):**  
   کدی که قرار است تست شود، اجرا می‌شود.
3. **ارزیابی (Assertion):**  
   خروجی کد با نتیجه مورد انتظار مقایسه می‌شود. اگر خروجی با انتظارات مطابقت نداشته باشد، تست شکست خواهد خورد.
4. **تمیزکاری (Teardown) [اختیاری]:**  
   در صورت نیاز، منابع استفاده‌شده در تست آزاد یا بازنشانی می‌شوند.



در این مثال:

* در **بخش آماده‌سازی** ورودی‌ها (2 و 3) مشخص شده‌اند.
* در **بخش اجرا** تابع Sum فراخوانی شده است.
* در **بخش ارزیابی** خروجی با مقدار مورد انتظار (5) مقایسه شده است.
* در صورت شکست، پیام خطا با استفاده از t.Errorf نمایش داده می‌شود.
* **۱. تست (Testing)**
* تست‌ها برای اطمینان از صحت عملکرد کد نوشته می‌شوند. در Go، تست‌ها در فایل‌هایی با پسوند \_test.go تعریف می‌شوند و توابع تست با نامی که با Test شروع می‌شود، شناخته می‌شوند.

## **۲. بنچمارک (Benchmark)**

بنچمارک‌ها برای اندازه‌گیری **عملکرد و سرعت اجرای کد** استفاده می‌شوند. توابع بنچمارک با نامی که با Benchmark شروع می‌شود، تعریف می‌شوند.

### ****بنچمارک در زبان Go****

بنچمارک‌ها در Go برای اندازه‌گیری **کارایی کد** و **زمان اجرای توابع** استفاده می‌شوند. این ابزار به شما کمک می‌کند که عملکرد بخش‌های مختلف کد خود را ارزیابی کرده و نقاط ضعف را شناسایی کنید. بنچمارک‌ها به صورت توابعی در فایل‌های تست (با پسوند \_test.go) تعریف می‌شوند و از پکیج داخلی testing استفاده می‌کنند.

## **ساختار کلی یک بنچمارک**

بنچمارک‌ها در Go باید با نامی که با Benchmark شروع می‌شود تعریف شوند و از آرگومانی از نوع \*testing.B استفاده کنند. در این توابع، حلقه‌ای با مقدار b.N وجود دارد که نشان‌دهنده تعداد دفعات اجرای کد مورد نظر است.



* مقدار b.N توسط فریمورک تست Go تنظیم می‌شود و بسته به سخت‌افزار سیستم ممکن است متغیر باشد.
* هدف این است که کد شما چندین بار اجرا شود تا عملکرد آن به‌طور دقیق اندازه‌گیری شود.

برای اجرای بنچمارک، از دستور زیر استفاده کنید

go test -bench=.

توضیحات:

* BenchmarkAdd-8: نام بنچمارک به همراه تعداد رشته‌های CPU که استفاده شده است.
* 1000000000: تعداد دفعات اجرای تابع.
* 0.280 ns/op: زمان اجرای تابع به ازای هر فراخوانی (نانوثانیه).